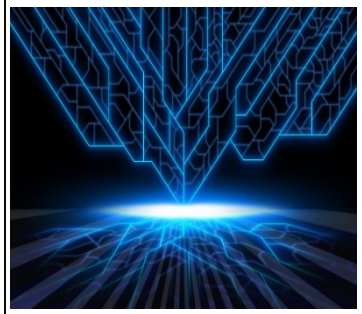
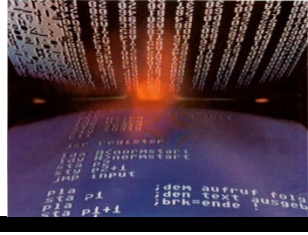
第3讲
汇编语言简介



吴海军
南京大学计算机科学与技术系

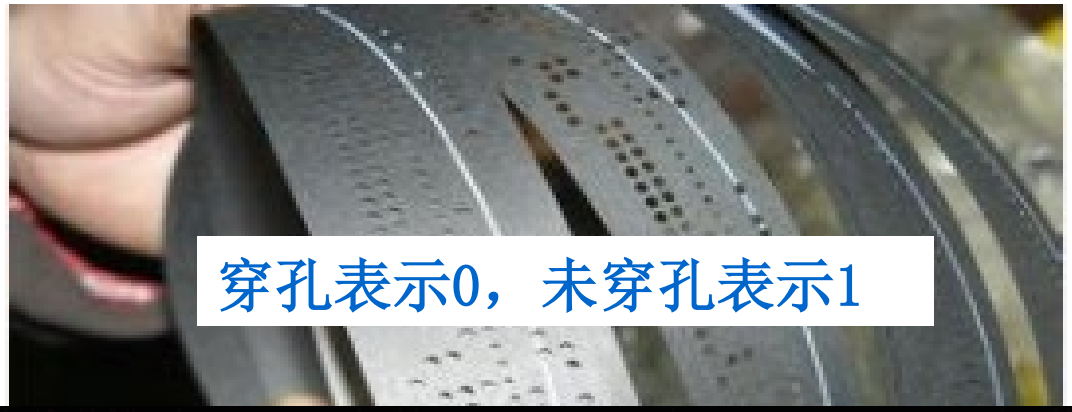
主要内容



- 简介
- 语法格式
- 程序组成
- 编译
- 调试工具
- 反汇编工具

机器语言编程

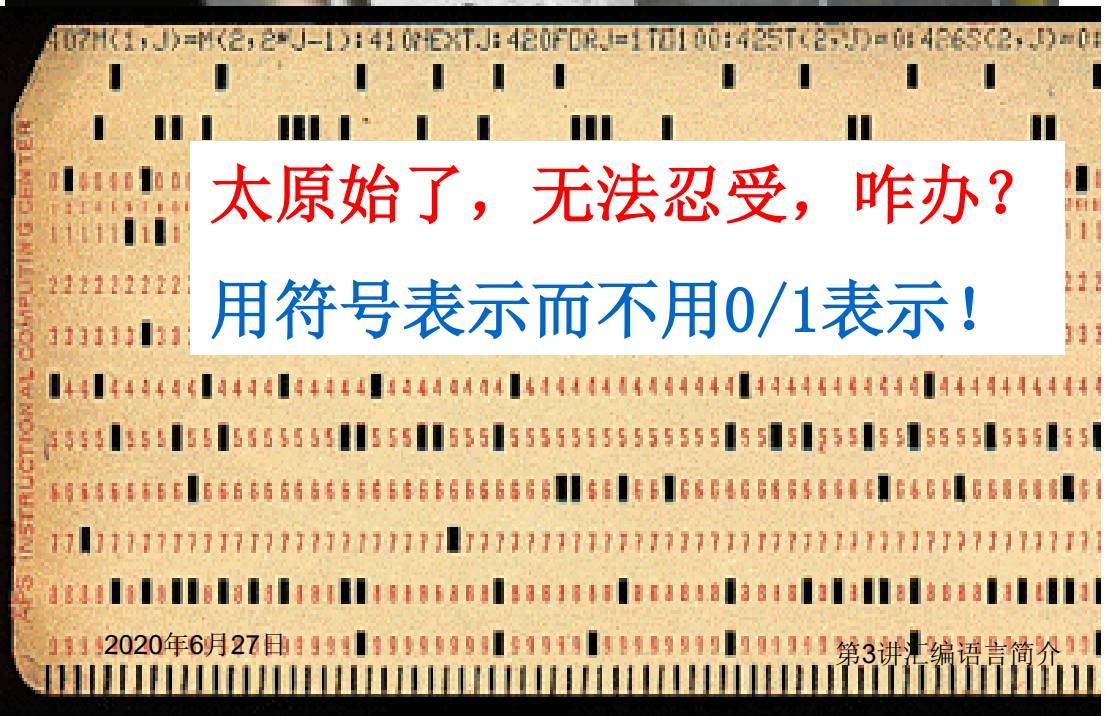
- 用机器语言编写程序，并记录在纸带或卡片上



输入：按钮、开关；所有信息都
输出：指示灯等　　是0/1序列！

假设：0010-jxx

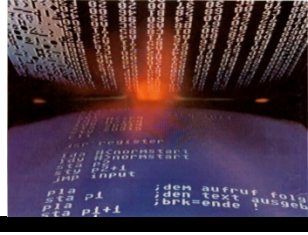
```
0:  0101 0110
1:  0010 0100
2:  ......
3:  ......
4:  0110 0111
5:  ......
6:  ......
```



若在第4条指令前加入指令，则需重新计算地址码（如jxx的目标地址），然后重新打孔。

不灵活！书写、阅读困难！

汇编语言



- 若用符号表示跳转位置和变量位置，是否简化了问题？
- 于是，汇编语言出现
  - 用助记符表示操作码
  - 用标号表示位置
  - 用助记符表示寄存器
  - …

```
0:  0101 0110        add B
1:  0010 0100        jxx L0
2:  ……               ……
3:  ……               ……
4:  0110 0111        L0: sub C
5:  ……               ……
6:  ……               B:  ……
7:  ……               C:  ……
```

用汇编语言编写的优点是：

不会因为增减指令而需要修改其他指令

不需记忆指令码，编写方便

可读性比机器语言强

不过，这带来新的问题，是什么呢？

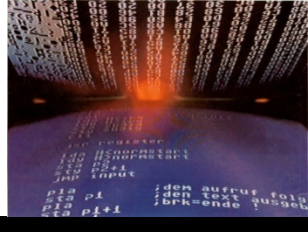
人容易了，可机器不认识这些指令了！

需将汇编语言转换为机器语言！

用汇编程序转换

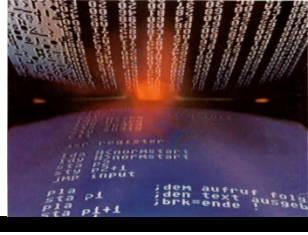
在第4条指令前加指令时不用改变add、jxx和sub指令中的地址码！

汇编语言



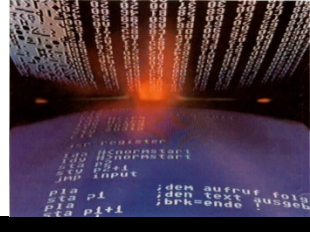
- 是一种用于电子计算机、微处理器、微控制器，或其他可编程器件的低级语言。
- 在不同的设备中，汇编语言对应着不同的机器语言指令集。一种汇编语言**专用**于某种计算机系统架构，而不能像许多高级语言可以在不同系统平台之间移植。
- 使用汇编语言编写的源代码，然后通过相应的汇编程序将它们转换成可执行的机器代码。这一过程被称为**汇编过程**。
- 汇编语言使用**助记符**（Mnemonics）来代替和表示特定机器语言的**指令码**。
- 汇编程序可以识别代表地址和常量的**标签**（Label）和**符号**（Symbols）。
- 通常被应用在底层硬件操作和高性能的程序优化操作。驱动程序、嵌入式操作系统和实时运行程序都会需要汇编语言。

汇编语言



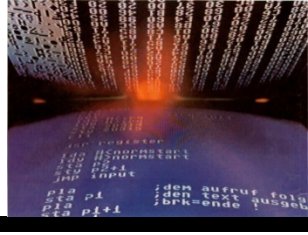
- 汇编更接近机器语言，能够直接对硬件进行操作
- 生成的程序与其他的语言相比具有更高的运行速度，占用更小的内存。
- 在一些对于时效性要求很高的程序、许多大型程序的核心模块以及工业控制方面大量使用汇编语言编程。
- 汇编语言能以让学生更深入了解计算机系统的运行原理。
- 是计算机专业学生的必备知识。

汇编语言



- 汇编语言由3个组件构成，用来定义程序操作
  - 操作码助记符：使用容易记忆的助记符(push、mov、sub和call等)来表示指令码，也成为汇编指令。
  - 命令：由“.”开始+关键字组成，指示汇编器如何执行专门的函数。
    - 如：.long、 .asci 、.float用于表示一个特定的数据类型
    - .section命令用于定义内存段
  - 数据变量：允许声明指向内存中特定位置的变量。
    - 指向一个内存位置的标记
    - 内存字节的数据类型和默认值。
    - 如：testvalue: .long 150

汇编语言



- 汇编语言源程序主要是由汇编指令构成。
- 什么是汇编指令？
  - 用助记符和标号来表示的指令（与机器指令一一对应）
- 指令又是什么呢？
  - 包含操作码和操作数或其地址码

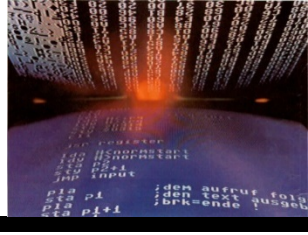
    （机器指令用二进制表示，汇编指令用符号表示）
  - 只能描述：取（或存一个数）

    两个数加（或减、乘、除、与、或等）

    根据运算结果判断是否转移执行

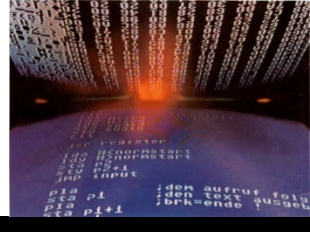
Intel® 64 and IA-32 Architectures Software Developer's Manual：
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

汇编语言特性



- 最少的数据类型
  - 整数：1、 2、4或8字节，表示数值、地址 (无类型指针)等
  - 浮点数：4、 8、12或16字节
  - 没有聚集数据类型，如数组、结构等
    - 只是存储在存储器中的连续字节
- 基本操作
  - 在寄存器或存储器中的数据上执行算术操作
  - 在存储器和寄存器间传送数据
    - 取数据：存储器取到寄存器
    - 存数据：寄存器存入存储器
  - 转移控制
    - 无条件跳转， 函数调用/返回
    - 条件分支

汇编语言格式 Intel vs. AT&T



AT&T 格式

```
leal  (%ecx,%ecx,2),%eax
subl  $8,%esp
cmpl  $0,-8(%ebp)
movl  $0x100(,%eax,4),%eax
```

Intel 格式

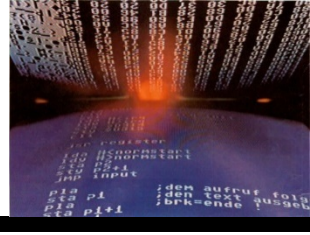
```
lea   eax,[ecx+ecx*2]
sub   esp,8
cmp   dword ptr [ebp-8],0
mov   eax,dword ptr [eax*4+100h]
```

- Intel汇编与AT&T汇编，分别被Microsoft 与GNU采用
- AT&T汇编格式与 Intel的不同
  - 目标操作数则在源操作数的右边：addl $4, %eax
  - 用‘$’前缀表示一个立即操作数，如 $0x100;
  - 寄存器操作数总是以‘%’作为前缀。如：%ecx
  - 操作数的字长是由操作码助记符末尾的字母决定，后缀‘b’、‘w’、‘l’、’q’分别表示字长为8、16、32、64位。

  如subl $8,%esp

项目	AT&T风格	Intel风格
操作数顺序	源操作数在前	目标操作数在前
寄存器名字	加%前缀	原样
立即数	加$前缀	原样
16进制立即数	加前缀0x	用后缀b与h分别表示二进制与十六进制
访问内存长度的表示	后缀b、w、l、q表示字节、字、双字、四字	前缀byte ptr, word ptr, dword ptr
引用全局或静态变量var的值	_var	[_var]
引用全局或静态变量var的地址	$_var	_var
内存直接寻址	seg_reg:immed32 (base, index, scale)	seg_reg: [base + index * scale + immed32]
寄存器间址	(%reg)	[reg]
寄存器变址寻址	_x(%reg)	[reg + _x]
立即数变址寻址	1(%reg)	[reg + 1]
整数数组寻址	_array (,%eax, 4)	[eax*4 + array]

AT&T 指令格式



- 典型的指令格式:

  操作符opcode 源操作数src, 目标操作数dst

what to do

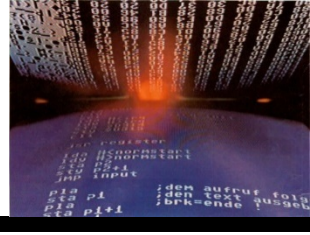
input source

result destination

- 操作符的后缀字母指明了操作数的长度:
  - 后缀‘b’、‘w’、‘l’、’q’分别表示字长为8、16、32、64位
  - 如subl $8,%esp

类别	说明	说明
数据传输	MOV {l,w,b} Source,Dest	把源地址中的数据传送到目标
	Movs{l,w,b} Source,Dest	传送符号扩展的数据
	Movz{l,w,b} Source,Dest	传送零扩展的数据
	xchg {l,w,b} dest1，dest2	交换
	Push/pop {l,w}	推入/弹出堆栈
算术	ADD/SUB {l,w,b} Source,Dest	加/减
	IMUL / MUL {l,w,b} formats	有符号/无符号乘法
	IDIV / DIV {l,w,b} DEST	有符号/无符号除法
	INC / DEC /NEG {l,w,b} DEST	递增/递减/否定
	CMP {l,w,b}source1, source2	比较
逻辑	AND/OR/ XOR / {l,w,b}Source,Dest	逻辑与/或/异或 /取反操作
	SAL / SAR {l,w,b} formats	算术移位左/右
	SHL / SHR {l,w,b}formats	逻辑移位左/右
控制传输	JMP address	无条件转移
	call address	将EIP保存在栈中，跳转到address
	ret	返回到被call语句所保存的EIP
	leave	从堆栈恢复EBP; 弹出堆栈帧
	j{e,ne,l,le,g,ge}address	跳转到地址，if{=,！=,<,<=,>,> =}
	loop address	递减ECX或CX; if= 0 跳转
	rep	重复字符串操作前缀
	Int number	软件中断
	IRET	中断返回; 从栈中弹出EFLAGS

GNU汇编语言程序组成



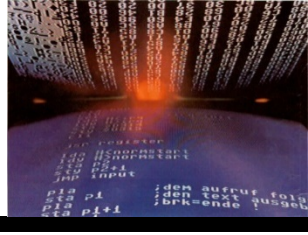
- GNU汇编语言，由不同的段构成，每个段都有不同的目的。三个最常用的段如下：
  - data数据段：用于声明带初始值的数据元素，这些数据元素用着汇编程序中的变量。该段不能扩展，并且在整个程序运行保持静态。
  - bss缓冲段：使用零或(null)值初始化的数据元素，用着汇编程序中的的缓冲区，也是静态的内存区域。
  - text文本段：所有的汇编语言程序必须有文本段，这是在可执行程序内声明指令码的区域。
- 汇编程序起始点定义
  - _start：标签用于表明程序从该条指令开始执行。
  - .globl命令：声明外部程序可以访问的程序标签。

GNU汇编程序模板

```
.section .data
        <initialized data here>
.section .bss
        <uninitialized data here>
.section .text
.globl _start
_start:
        <instruction code gone
here>
```

```
#cpuid.s Sample program to extract the
processor Vendor ID
.section .data
    output:    .ascii "The processor Vendor ID
is 'xxxxxxxxxxxx'\n”
.section .text
.globl _start
_start:    movl $0, %eax
        cpuid
        movl $output, %edi
        movl %ebx, 28(%edi)
        movl %edx, 32(%edi)
        movl %ecx, 36(%edi)
        movl $4, %eax
        movl $1, %ebx
        movl $output, %ecx
        movl $42, %edx
        int $0x80
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

汇编程序编译、链接和执行



- 编译器： $ as –o cpuid.o cpuid.s
  - 转换 .s 到 .o
  - 每条指令进行二进制编码
  - 几乎完整的可执行代码映像
  - 缺少不同文件代码之间的链接
- 链接器： $ ld –o cpuid cpuid.o
  - 解析文件间的相互引用
  - 与静态运行时库函数合并
    - E.g., malloc, printf代码
  - 一些库是动态链接的
    - 只有在程序执行时才发生链接
- 运行可执行文件： $ ./cpuid

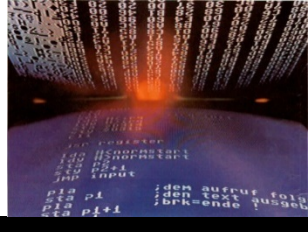
使用了C库函数，必须把C库文件链接到程序目标代码：

```
$ ld -dynamic-linker /lib/ld-linux.so.2 -o cpuid2 -lc cpuid2.o
```

使用了gcc编译时自动链接C库函数：

```
$ gcc -o cpuid2 -lc cpuid2.s
```

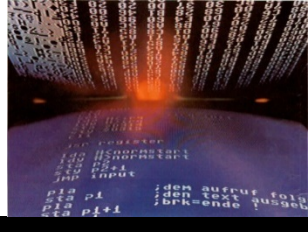
汇编程序使用gcc编译



- gcc（GNU Common Compiler）GNU通用编译器，用来在一个步骤内完成编译和链接C语言、汇编语言源程序。
- gcc编译汇编程序时，查找的是main标签，需要把源程序中的_start标签改为main标签：
  - .section .text
  - .globl main
  - main:

$ gcc –o cpuid cpuid.s

汇编程序使用gcc编译

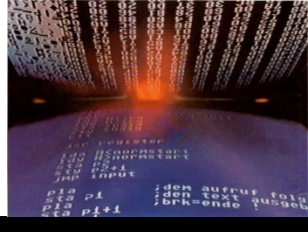


- Gcc（选项）(参数)
  - 选项：
    - -o：指定生成的输出文件；
    - -E：仅执行编译预处理；
    - -S：将C代码转换为汇编代码；
    - -wall：显示警告信息；
    - **-m32：可以将程序编译为32位**
    - -c：仅执行编译操作，不进行连接操作。
  - 参数：源文件，指定C、C++、汇编语言源代码文件。
- 实例：
  - gcc test.c

  将test.c预处理、汇编、编译并链接形成可执行文件。默认输出为a.out。
  - gcc test.c -o test

  将test.c预处理、汇编、编译并链接形成可执行文件test。-o选项用来指定输出文件的文件名。

汇编程序使用gcc编译



- gcc -E test.c -o test.i

将test.c预处理输出test.i文件。

- gcc -S test.i

将预处理输出文件test.i汇编成test.s文件。

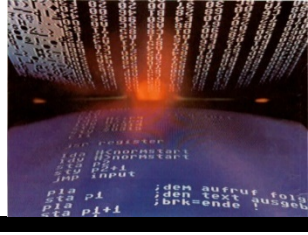
- gcc -c test.s

将汇编输出文件test.s编译输出test.o文件。

- 选项 -O
- gcc -O1 test.c -o test

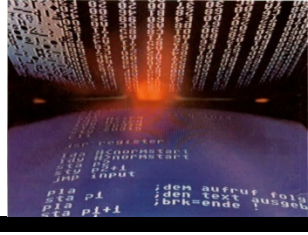
使用编译优化级别1编译程序。级别为1~3，级别越大优化效果越好，但编译时间越长。

GNU C/C++编译器的选项列表



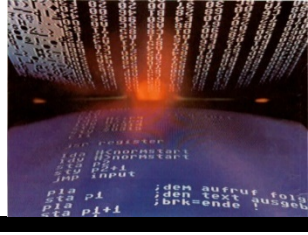
选项	描述
-x language	指定语言（C、C++和汇编为有效值）
-c	只进行预处理、编译和汇编，生成obj文件
-S	只进行预处理、编译（不汇编或连接）
(-s) file	
-E	只进行预处理（不编译、汇编或连接）
-o file	用来指定输出文件名
-l library	用来指定所用库
-L directory	为库文件的搜索指定目录
-I directory	为include文件的搜索指定目录
-w	禁止警告消息
-pedantic	严格要求符合ANSI标准
-Wall	显示附加的警告信息
-g	产生排错信息（同gdb一起使用时）
-ggdb	产生排错信息（用于gdb）
-p	产生proff所需的信息
-pg	产生groff所需的信息
-O	优化，缺省O1，O0表示没有优化

调试工具GDB



- gdb（GNU Debugger）主要工作在字符模式下，是一个功能强大的交互式程序调试工具
  - gdb能在程序运行时观察程序的内部结构和内存的使用情况
- gdb主要提供以下功能
  - 监视程序中变量的值的变化
  - 设置断点，使程序在指定的代码行上暂停执行，便于观察
  - 单步执行代码
  - 分析崩溃程序产生的core文件
- 命令形式如下：gdb filename

调试工具GDB



- 需要使用-gstabs参数重新汇编源代码

  $ as –gstabs –o cpuid.o cpuid.s

  $ ld –o cpuid cpuid.o

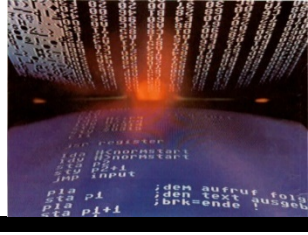
- 查看cpuid文件大小，发现变大了，包含了必要的调试信息

  $ gdb cpuid

- 设置断点：break * label+offset / b 源文件:行号
  - Label：源代码中的标签（_start）

  (gdb) break *_start+1 / b cupid.s:9
  - 使用next或step命令单步调试
  - 使用cont命令、run命令持续执行

调试工具GDB



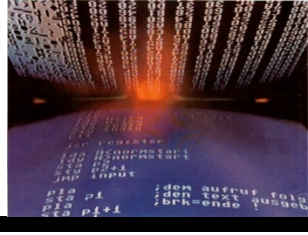
- 查看数据：查看用于变量的寄存器和内存位置的数据元素
- info registers：显示所有寄存器的值
- print：显示特定寄存器或者来自程序的变量的值，加修饰符可以改变输出格式，/d显示十进制的值，/t显示二进制的值，/x显示十六进制的值
- x：显示特定内存位置的内容，加修饰符可以改变输出格式。/nyz，n表示要显示的字段数;y表示输出格式,c字符d十进制x十六进制;z表示要显示的字段的长度，b用于字节h用于16位w用于32位

(gdb) info registers：显示所有寄存器的值

(gdb) print/x $ebx：用十六进制显示ebx寄存器的值

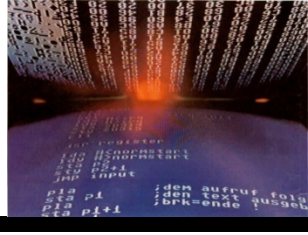
(gdb) x/42cb &output:以字符方式显示从标签output处开始的42个字节的内存内容。

GDB常用命令



命　令	效　果
开始和停止	
quit	退出 GDB
run	运行程序（在此给出命令行参数）
kill	停止程序
断点	
break sum	在函数 sum 入口处设置断点
break *0x8048394	在地址 0x8048394 处设置断点
delete 1	删除断点 1
delete	删除所有断点
执行	
stepi	执行 1 条指令
stepi 4	执行 4 条指令
nexti	类似于 stepi，但是以函数调用为单位的
continue	继续执行
finish	运行直到当前函数返回
检查代码	
disas	反汇编当前函数

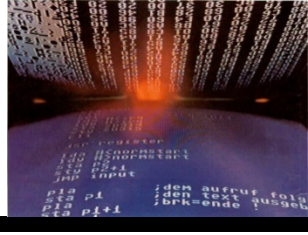
GDB常用命令



disas	反汇编当前函数
disas sum	反汇编函数 sum
disas 0x8048397	反汇编位于地址 0x8048397 附近的函数
disas 0x8048394 0x80483a4	反汇编指定地址范围内的代码
print /x $eip	以十六进制输出程序计数器的值
检查数据	
print $eax	以十进制输出 %eax 的内容
print /x $eax	以十六进制输出 %eax 的内容
print /t $eax	以二进制输出 %eax 的内容
print 0x100	输出 0x100 的十进制表示
print /x 555	输出 555 的十六进制表示
print /x ($ebp+8)	以十六进制输出 %ebp 的内容加上 8
print *(int *) 0xfff076b0	输出位于地址 0xfff076b0 的整数
print *(int *) ($ebp+8)	输出位于地址 %ebp + 8 处的整数
x/2w 0xfff076b0	检查从地址 0xfff076b0 开始的双（4 字节）字
x/20b sum	检查函数 sum 的前 20 个字节
有用的信息	
info frame	有关当前栈帧的
info registers	所有寄存器的值
help	获取有关 GDB 的信息

http://sourceware.org/gdb/

反汇编工具

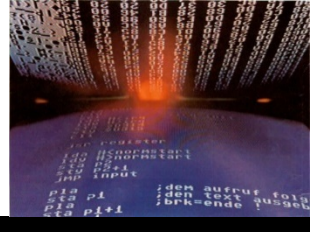


- objdump：根据目标文件生成汇编语言代码
  - 检查目标代码时非常有用的工具
  - 分析指令串的二进制模式
  - 生成与汇编代码近似的代码
  - 对完全可执行的文件 或者 目标代码文件都可以运行
- 最常用-d显示反汇编后的目标代码文件

  $ objdump –d cpuid.o

```
00401040 <_sum>:
   0:       55                      push   %ebp
   1:       89 e5                   mov    %esp,%ebp
   3:       8b 45 0c                mov    0xc(%ebp),%eax
   6:       03 45 08                add    0x8(%ebp),%eax
   9:       5d                      pop    %ebp
   a:       c3                      ret
```

Objdump命令

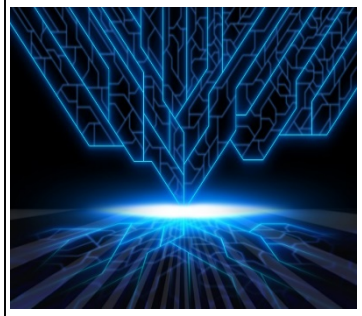


- 使用**objdump**命令对任意一个二进制文件进行反汇编命令：

  objdump -D -b binary -m i386 a.bin

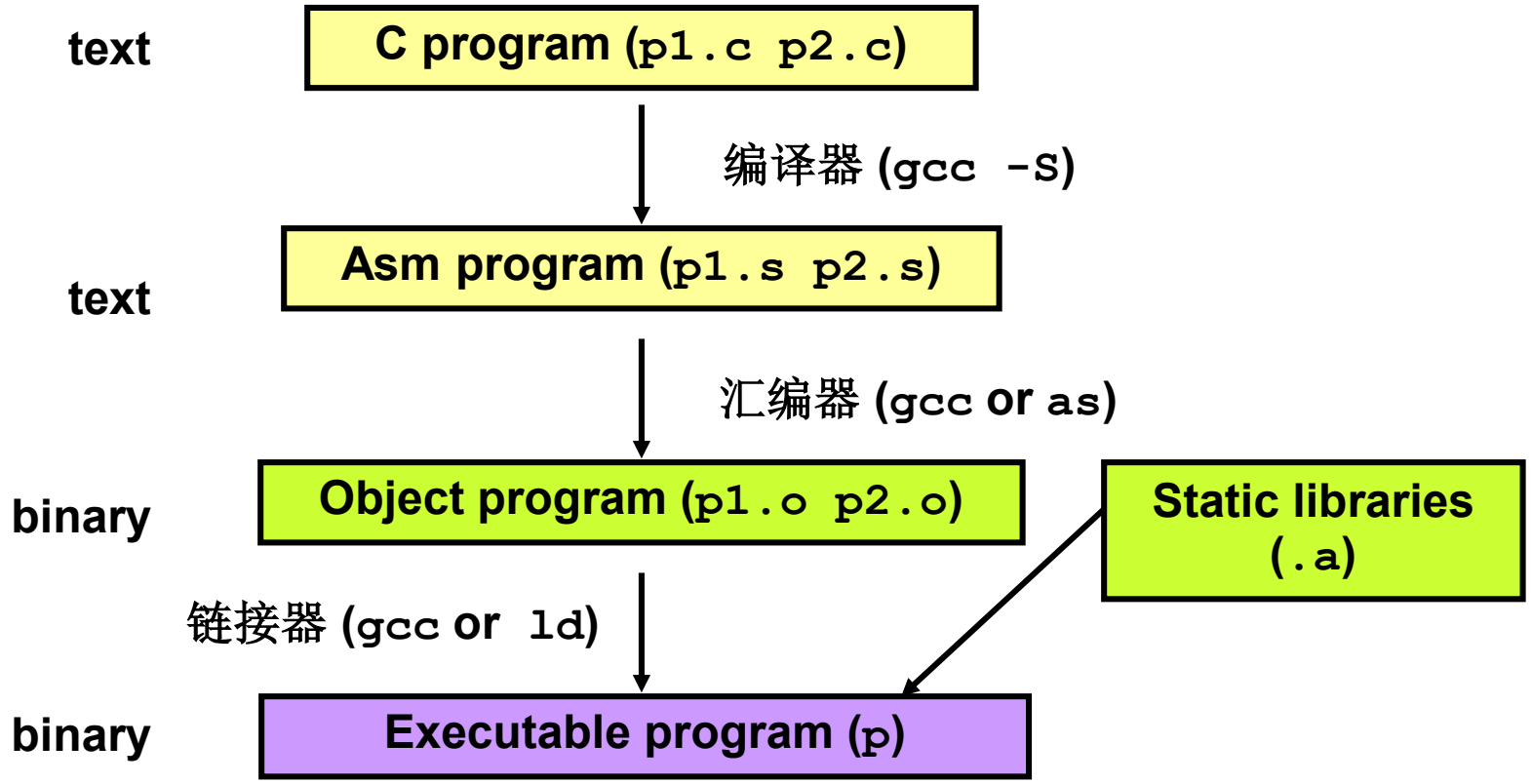
  - -D表示对全部文件进行反汇编，-b表示二进制，-m表示指令集架构，a.bin就是我们要反汇编的二进制文件
- objdump -m可以查看更多支持的指令集架构，如i386:x86-64，i8086等
- 可以指定big-endian或little-endian（-EB或-EL），指定从某一个位置开始反汇编等。
- Objdump --help

C程序编译



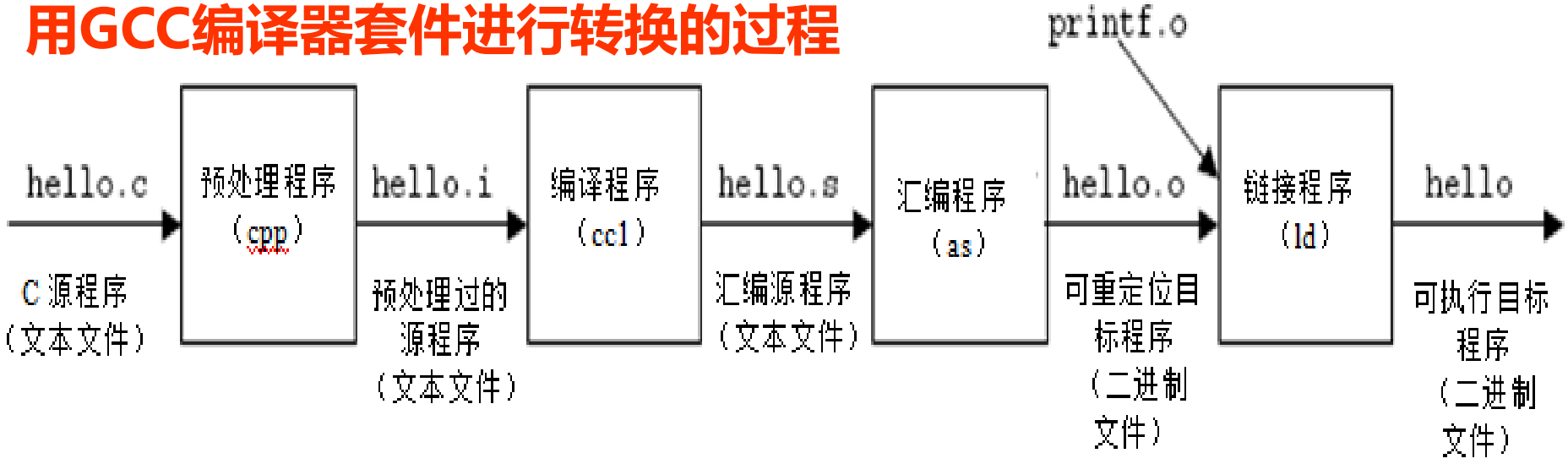
C代码转换成目标代码

- 代码文件　　`p1.c p2.c`
- 编译命令:　　`gcc -O1 p1.c p2.c -o p`
  - 使用优化选项(`-O1`)
  - 把二进制结果放到文件 `p`中



高级语言程序转换为机器代码的过程

用GCC编译器套件进行转换的过程



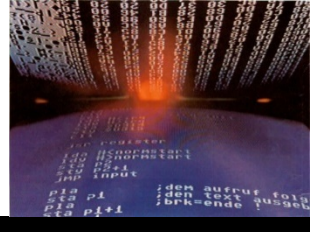
预处理：在高级语言源程序中插入所有用#include命令指定的文件和用#define声明指定的宏。

编译：将预处理后的源程序文件编译生成相应的汇编语言程序。

汇编：由汇编程序将汇编语言源程序文件转换为可重定位的机器语言目标代码文件。

链接：由链接器将多个可重定位的机器语言目标文件以及库例程（如printf()库函数）链接起来，生成最终的可执行目标文件。

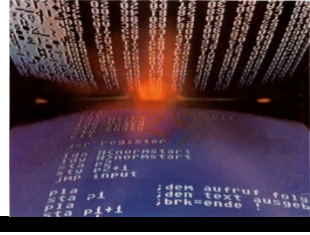
高级语言程序转换为机器代码的过程



```
gcc -E hello.c -o hello.i
gcc -S hello.i -o hello.s
gcc -c hello.i
gcc -o hello hello.o
./hello
```

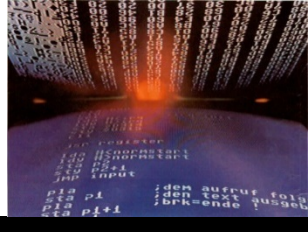
```
 7344 7♦     8 23:37 hello
   63 7♦     8 23:35 hello.c
17505 7♦     8 23:36 hello.i
 1068 7♦     8 23:37 hello.o
  653 7♦     8 23:36 hello.s
```

高级语言程序转换为机器代码的过程



```
main:
.LFB0:
        .cfi_startproc
        leal    4(%esp), %ecx
        .cfi_def_cfa 1, 0
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        .cfi_escape 0x10,0x5,0x2,0x75,0
        movl    %esp, %ebp
        pushl   %ecx
        .cfi_escape 0xf,0x3,0x75,0x7c,0x6
        subl    $4, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    puts
        addl    $16, %esp
        movl    $0, %eax
        movl    -4(%ebp), %ecx
        .cfi_def_cfa 1, 0
        leave
        .cfi_restore 5
        leal    -4(%ecx), %esp
        .cfi_def_cfa 4, 4
        ret
```

GCC使用举例



```
1    // test.c
2    #include <stdio.h>
3    int add(int i, int j )
4    {
5        int x = i +j;
6        return x;
7    }
```

test1.c和test2.c，最终生成可执行文件为test

est2.c -o test

“gcc –c test.s –o test.o”将test.s汇编为test.o

“objdump -d test.o”将test.o 反汇编为

gcc -E test.c -o test.i

gcc -S test.i -o test.s

gcc –S test.c –o test.s

test.s

```
add:
pushl%ebp
movl  %esp, %ebp
subl  $16, %esp
movl  12(%ebp), %eax
movl  8(%ebp), %edx
leal    (%edx, %eax), %eax
movl  %eax, -4(%ebp)
movl  -4(%ebp), %eax
leave
ret
```

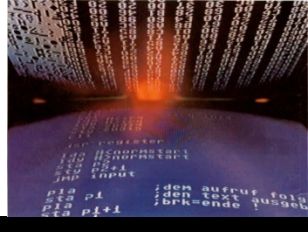
00000000 <add>:

0:	55	push %ebp
1:	89 e5	mov %esp, %ebp
3:	83 ec 10	sub $0x10, %esp
6:	8b 45 0c	mov 0xc(%ebp), %eax
9:	8b 55 08	mov 0x8(%ebp), %edx
c:	8d 04 02	lea (%edx,%eax,1), %eax
f:	89 45 fc	mov %eax, -0x4(%ebp)
12:	8b 45 fc	mov -0x4(%ebp), %eax
15:	c9	leave
16:	c3	ret

位移量　机器指令　汇编指令

编译得到的与反汇编得到的汇编指令形式稍有差异

两种目标文件



```
1    // test.c
2    #include <stdio.h>
3    int add(int i, int j )
4    {
5        int x = i +j;
6        return x;
7    }
```

test.o：可重定位目标文件

test：可执行目标文件

“objdump -d test.o”结果

00000000 <add>:

0:	55	push	%ebp
1:	89 e5	mov	%esp, %ebp
3:	83 ec 10	sub	$0x10, %esp
6:	8b 45 0c	mov	0xc(%ebp), %eax
9:	8b 55 08	mov	0x8(%ebp), %edx
c:	8d 04 02	lea	(%edx,%eax,1), %eax
f:	89 45 fc	mov	%eax, -0x4(%ebp)
12:	8b 45 fc	mov	-0x4(%ebp), %eax
15:	c9	leave	
16:	c3	ret	

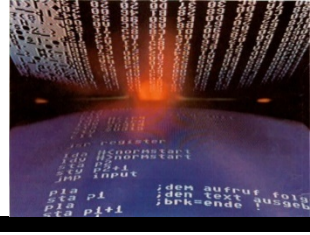
“objdump -d test” 结果

080483d4 <add>:

80483d4:	55	push ...
80483d5:	89 e5	…
80483d7:	83 ec 10	…
80483da:	8b 45 0c	…
80483dd:	8b 55 08	…
80483e0:	8d 04 02	…
80483e3:	89 45 fc	…
80483e6:	8b 45 fc	…
80483e9:	c9	…
80483ea:	c3	ret

test.o中的代码从地址0开始，test中的代码从80483d4开始！

编译成汇编代码



- C 代码

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

生成的IA32汇编代码

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    popl %ebp
    ret
```

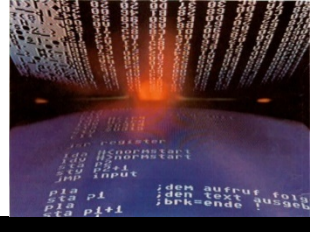
有些编译器使用指令“leave”，生成汇编文件

使用命令

```
/usr/local/bin/gcc -O1 -S code.c
```

生成文件code.s

目标代码



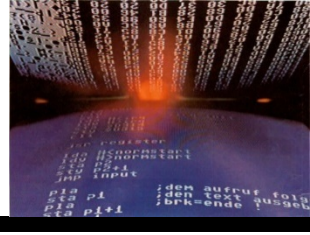
Sum代码

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

- **总共11 字节**
- **每条指令1、2或3字节**
- **起始地址0x401040**

- 汇编器
  - 转换 `.s` 到 `.o`
  - 每条指令进行二进制编码
  - 几乎完整的可执行代码映像
  - 缺少不同文件代码之间的链接
- 链接器
  - 解析文件间的相互引用
  - 与静态运行时库函数合并
    - E.g., `malloc`, `printf`代码
  - 一些库是动态链接的
    - 只有在程序执行时才发生链接

机器指令例子



```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

类似的表达式:

```
x += y
```

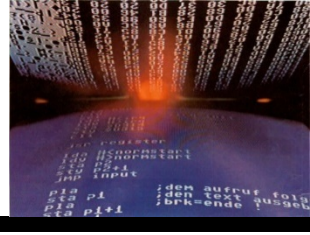
Or

```
int eax;
int *ebp;
eax += ebp[2]
```

```
0x401046:      03 45 08
```

- C 代码
  - 两个有符号整数相加
- 汇编代码
  - 两个4字节整数相加
    - GCC语法中Long数据类型
    - 有符号无符号数指令相同
  - 操作数:
    - y: 寄存器 %eax
    - x: 存储器 M[%ebp+8]
    - t: 寄存器 %eax
      - 返回值存在%eax
- 目标代码
  - 3字节指令
  - 存储在地址0x401046

反汇编目标代码



反汇编

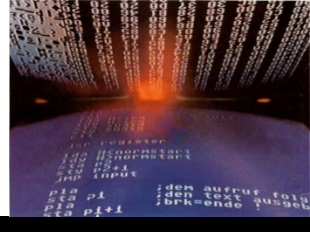
```
00401040 <_sum>:
   0:      55                   push   %ebp
   1:      89 e5                mov    %esp,%ebp
   3:      8b 45 0c             mov    0xc(%ebp),%eax
   6:      03 45 08             add    0x8(%ebp),%eax
   9:      5d                   pop    %ebp
   a:      c3                   ret
```

- 反汇编器

  `objdump -d p`

  - 检查目标代码时非常有用的工具
  - 分析指令串的二进制模式
  - 生成与汇编代码近似的代码
  - 对`a.out`(完全可执行的)或者`.o`文件都可以运行

其他反汇编



Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

反汇编

```
0x401040 <sum>:      push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
```

- 应用Gdb调试器

  `gdb p`

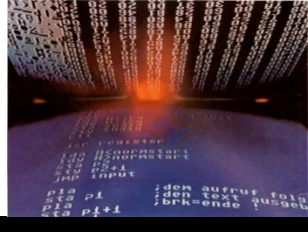
  `disassemble sum`

  - 反汇编程序

  `x/11xb sum`

  - 检查从sum开始的11个字节

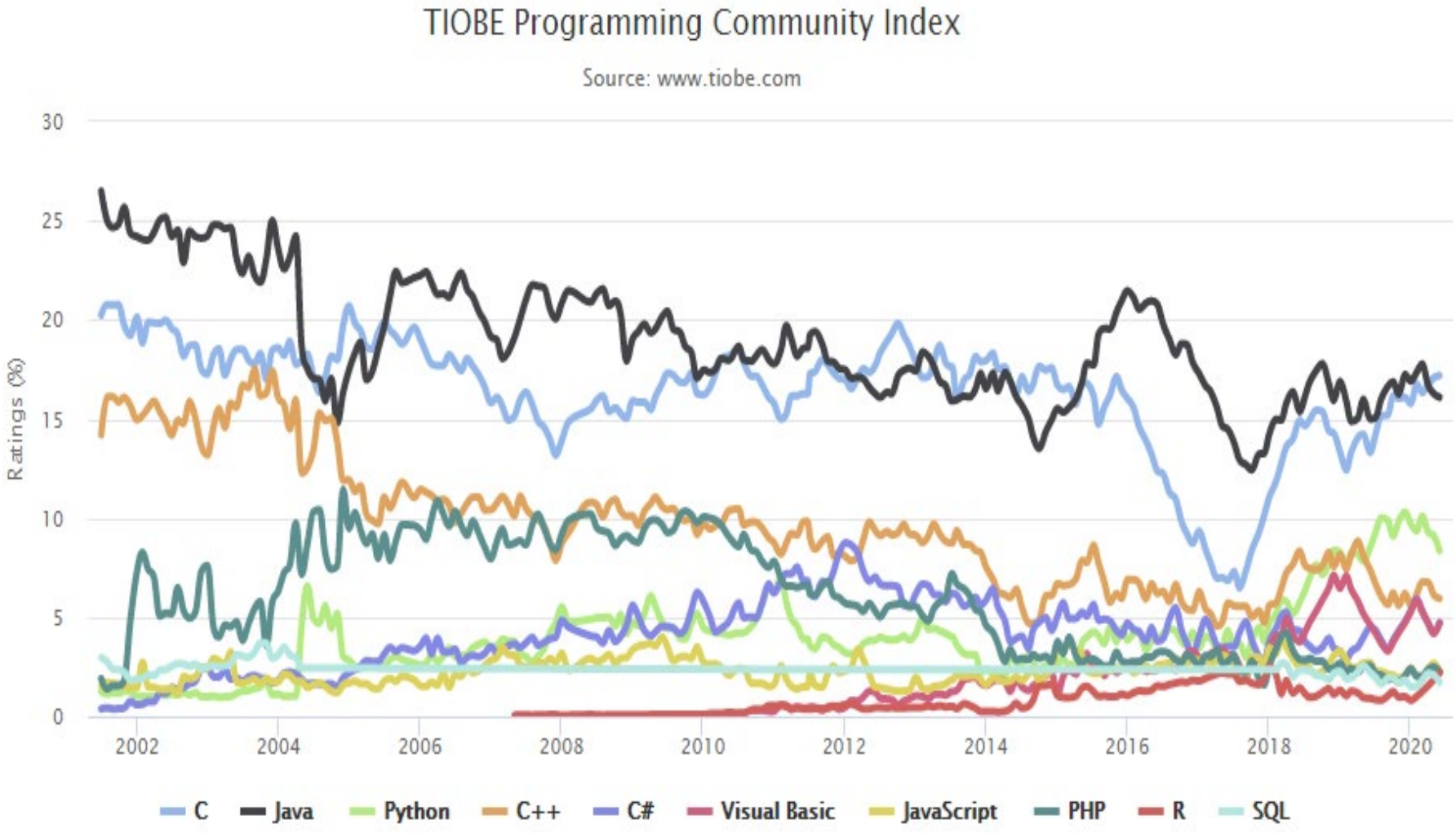
编程语言排行榜



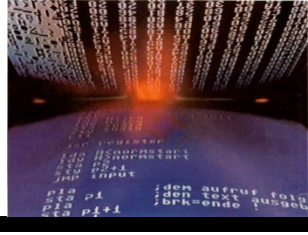
Jun 2020	Jun 2019	Change	Programming Language	Ratings	Change
1	2	↑	C	17.19%	+3.89%
2	1	↓	Java	16.10%	+1.10%
3	3		Python	8.36%	-0.16%
4	4		C++	5.95%	-1.43%
5	6	↑	C#	4.73%	+0.24%
6	5	↓	Visual Basic	4.69%	+0.07%
7	7		JavaScript	2.27%	-0.44%
8	8		PHP	2.26%	-0.30%
9	22	⇈	R	2.19%	+1.27%
10	9	↓	SQL	1.73%	-0.50%
11	11		Swift	1.46%	+0.04%
12	15	↑	Go	1.02%	-0.24%
13	13		Ruby	0.98%	-0.41%
14	10	⇊	Assembly language	0.97%	-0.51%
15	18	↑	MATLAB	0.90%	-0.18%

https://www.tiobe.com/tiobe-index//

编程语言发展历程


TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
C
Java
Python
C++
C#
Visual Basic
JavaScript
PHP
R
SQL

编程语言发展历程



The Assembly language Programming Language

Some information about Assembly language:

Highest Position (since 2015): #8 in May 2017

Lowest Position (since 2015): #32 in Dec 2014