汇编语言

The Hardware/Software Interface

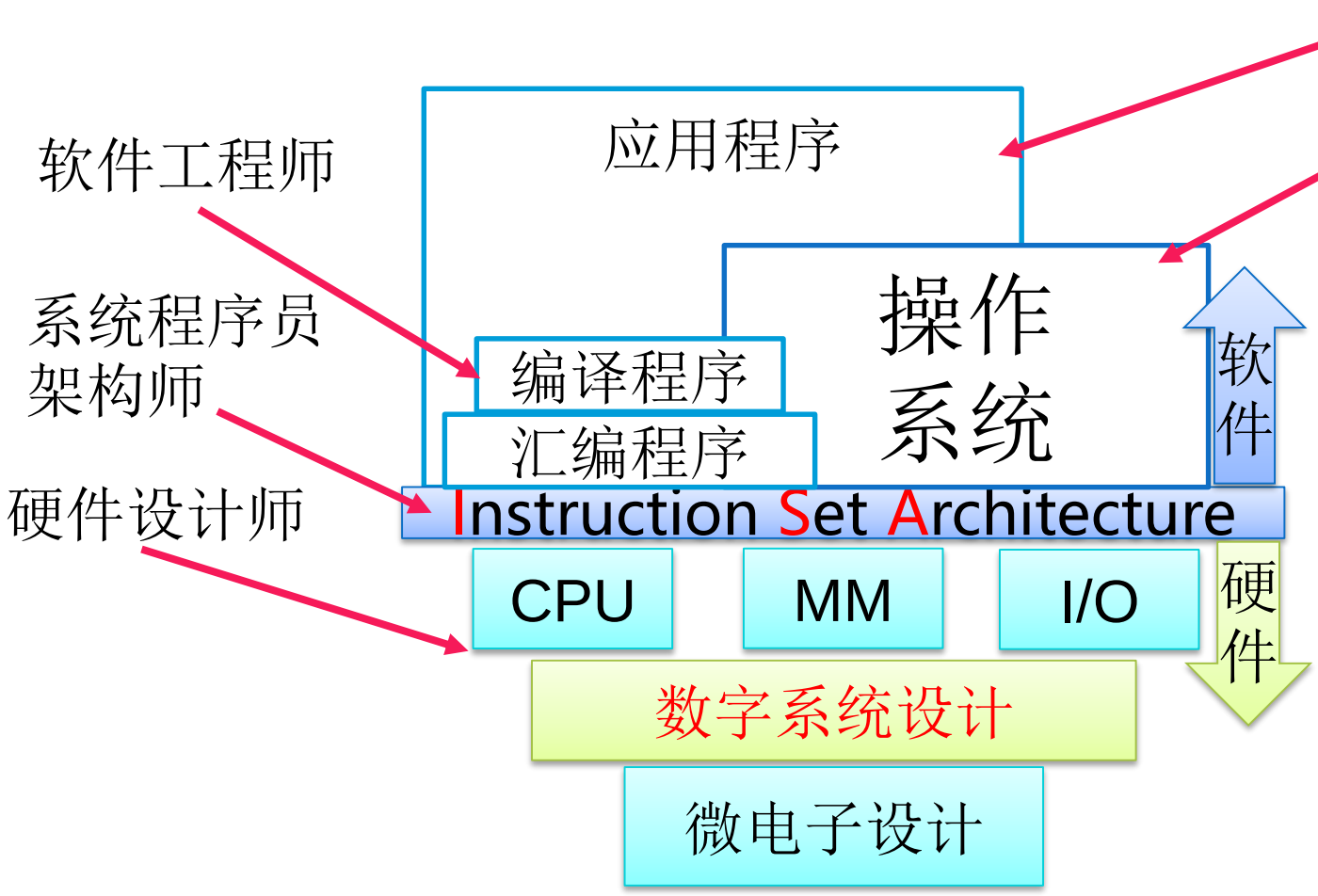
吴海军

南京大学计算机系

主要内容

- 课程简介
- 计算机系统运行机理
- IA-32简介
- 汇编语言介绍

计算机系统的知识体系



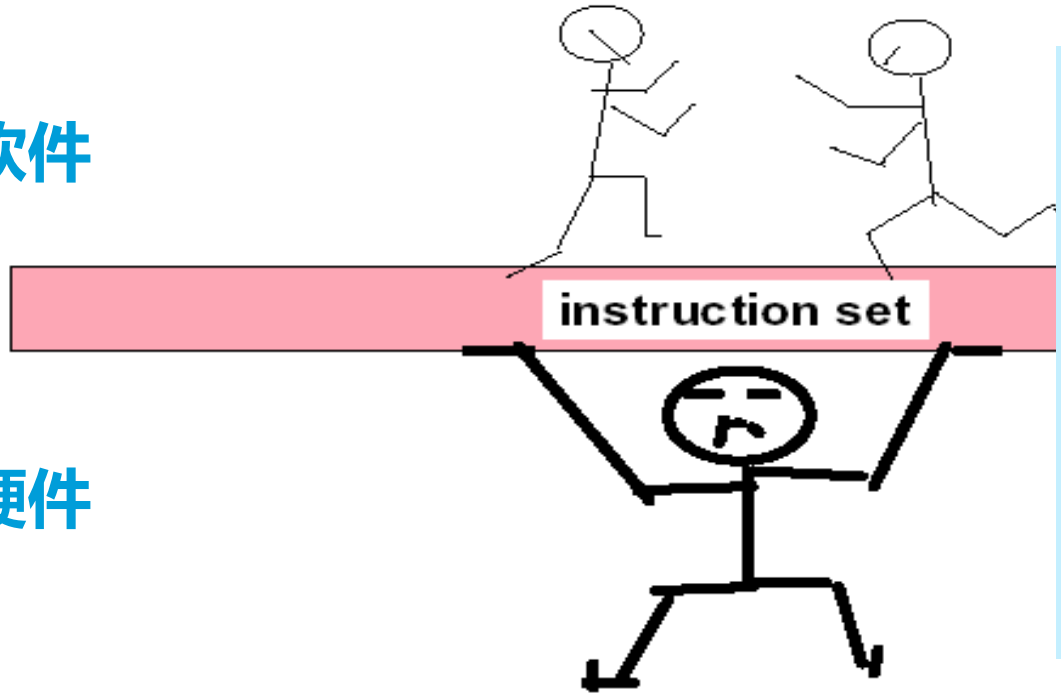
上层是下层的抽象，下层是上层的实现，底层为上层提供支撑环境！

不同计算机课程处于不同层次

必须将各层次关联起来解决问题

ISA处于软件和硬件的交界面（接口）

计算机软硬件关系



ISA是对硬件的抽象

所有软件功能都建立在ISA之上

ISA是最重要的层次！

不同ISA规定的指令集不同，如，IA-32、MIPS、ARM等

计算机系统有硬件和系统软件组成，他们协同完成程序的执行。

硬件为骨骼、软件为血肉，硬件是计算机系统的基石

机器语言由二进制代码构成，能被计算机硬件直接执行。

一个典型程序的转换处理过程

经典的“ hello.c ”C-源程序

```
#include <stdio.h>

int main()
{
printf("hello, world\n");
}
```

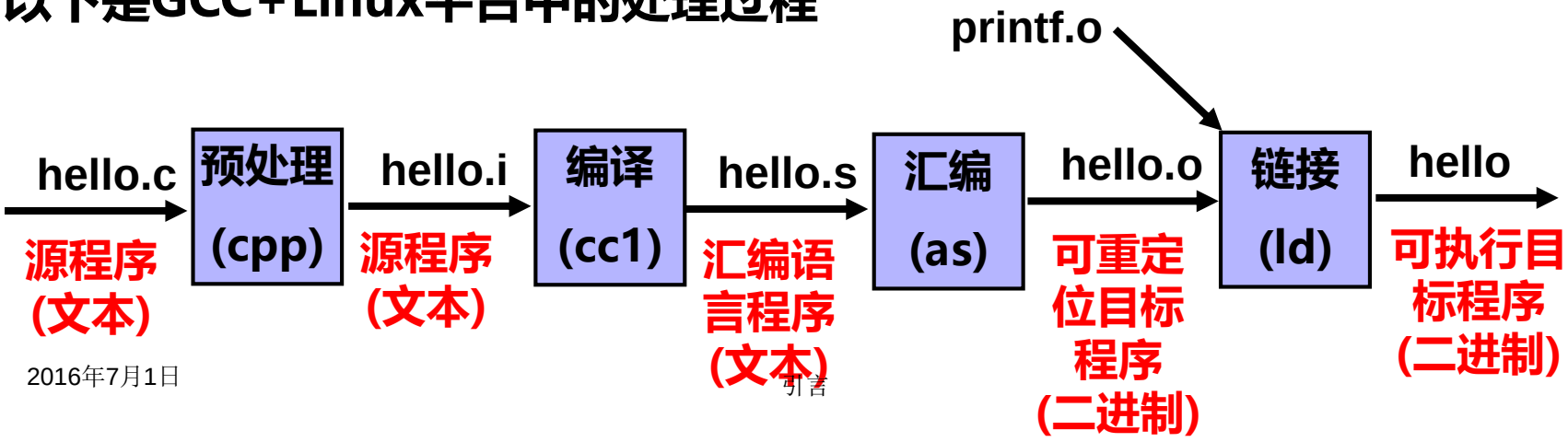
hello.c的ASCII文本表示

```
# i n c l u d e <sp> < s t d i o .
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46
h > \n \n i n t <sp> m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n <sp> <sp> <sp> <sp> p r i n t f ( " h e l
10 32 32 32 32 112 114 105 110 116 102 40 34 104 101 108
l o , <sp> w o r l d \ n " ) ; \n }
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```

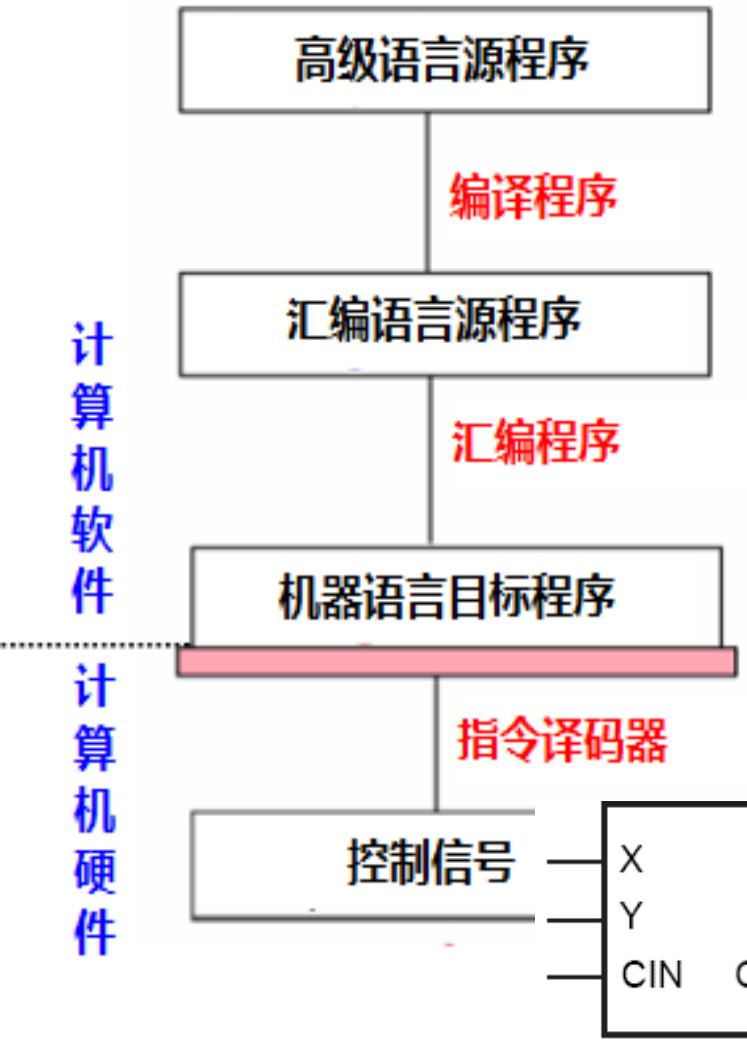
功能：输出“hello,world”

计算机不能直接执行hello.c！

以下是GCC+Linux平台中的处理过程



计算机程序的运行层次



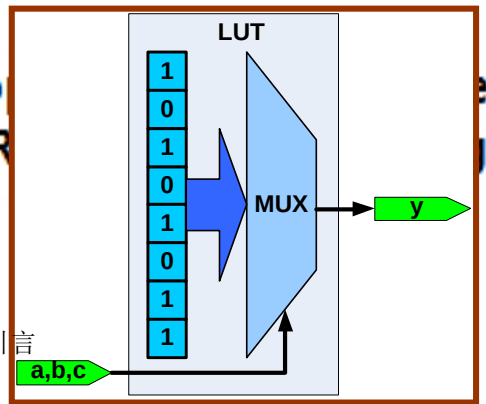

```
tmp = a[i];
a[i] = a[i+1];
a[i+1] = tmp;
```

```
lw $8, 0($2)
lw $9, 4($2)
sw $9, 0($2)
sw $8, 4($2)
```

高级语言到汇编语言到机器语言的转换。

数据的表示和操作

100011	00010	01000	0000000000000000
100011	00010	01001	0000000000000100
101011	00010	01001	0000000000000000
101011	00010	01000	0000000000000100



elB=11,ALUop=add,

Wr=1, ...

课程内容概要

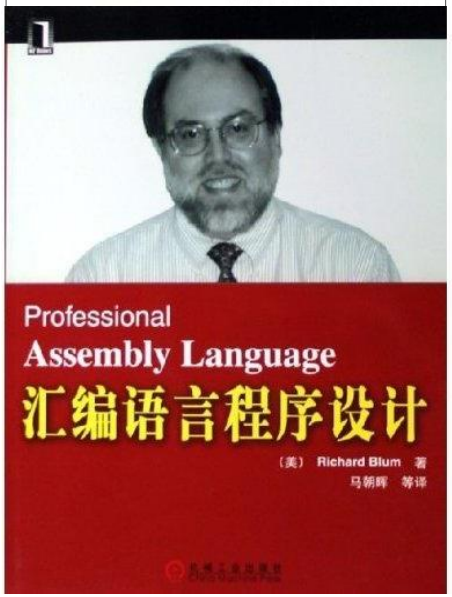
- IA-32体系架构介绍
- 汇编语言-AT&T格式介绍
- 数据的如何表示（Representation）
  - 信息的表示
  - 数据存储
  - 位运算、数据运算
  - 浮点数的表示及运算
- 高级语言程序设计如何转换到底层实现
  - 数据传送、算术和逻辑操作
  - 条件、循环、分支控制、堆栈、过程、递归和指针
  - 数组、结构、联合、内存分配、缓冲区溢出

课程介绍

- 目的：在软件和硬件间建立关联，在高级语言和机器级编码间建立关联，强化理解而不是记忆。
- 内容：
  - 学习Linux下汇编语言的编程、调试和反汇编工具
  - 以 IA-32+Linux+C+gcc+gas 为平台
  - 数据在计算机系统中如何表示
  - 高级语言的结构在汇编中是如何实现的
- 前导课程：C语言程序设计、数字逻辑电路
- 后续课程：计算机系统基础、算法设计与分析等

课程介绍

- TextBook:
  - 《深入理解计算机系统(原书第2版)》1-3章 作者：（美）Randal E.Bryant　David O'Hallaron 著,龚奕利 雷迎春译，机械工业出版社，2010；
- Reference Book:
  - **《计算机系统基础》 1-3章，袁春风，机械工业出版社，2014.7**
  - 《汇编语言程序设计：Professional Assembly Languange》 Richard Blum著 马朝晖等译 机械工业出版社2006



课程介绍

- 教学方法
  - 上午：授课
  - 下午：实验
  - 晚上：作业
- 综合实验：二进制炸弹实验，开学前上交
- 课程考核
  - 平时：20%，课后作业
  - 实验：30%，下午的实验+综合实验
  - 考试：50%，书面考试

助教

- 谭龙海，15950572339；QQ：591191160 学号：MF1533044 591191160@qq.com
- 王申，15996311418 qq:1518288577 学号：MG1533061 wangshen0716@163.com
- 陈英 13989476885，QQ：8472140.学号：DG1533003　8472140@qq.com
- 刘杰，15205164044 qq:1170381285 学号：MG1533026 nju_jie_liu@foxmail.com

课程参考

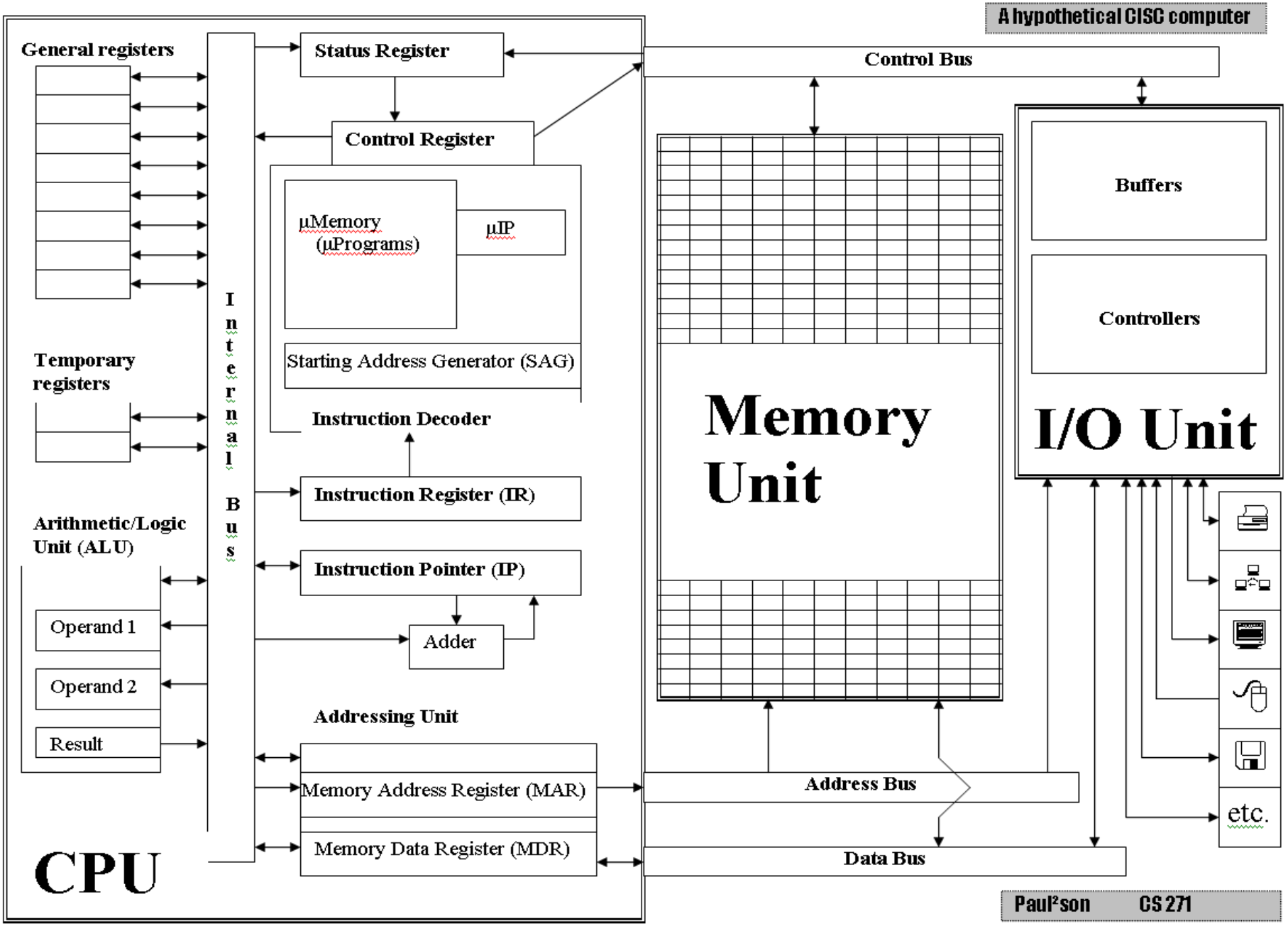
- 课程网站：http://cslabcms.nju.edu.cn/
- QQ群：475233864（2016汇编语言）

- CMU课程：
  http://csapp.cs.cmu.edu/3e/students.html
- MOOC课程：
  - 中国大学MOOC平台《计算机系统基础（一）--程序的表示、转换与链接》袁春风老师
    http://www.icourse163.org/course/nju-1001625001#/info/
  - **《The Hardware/Software Interface》：https://class.coursera.org/hwswinterface-002**

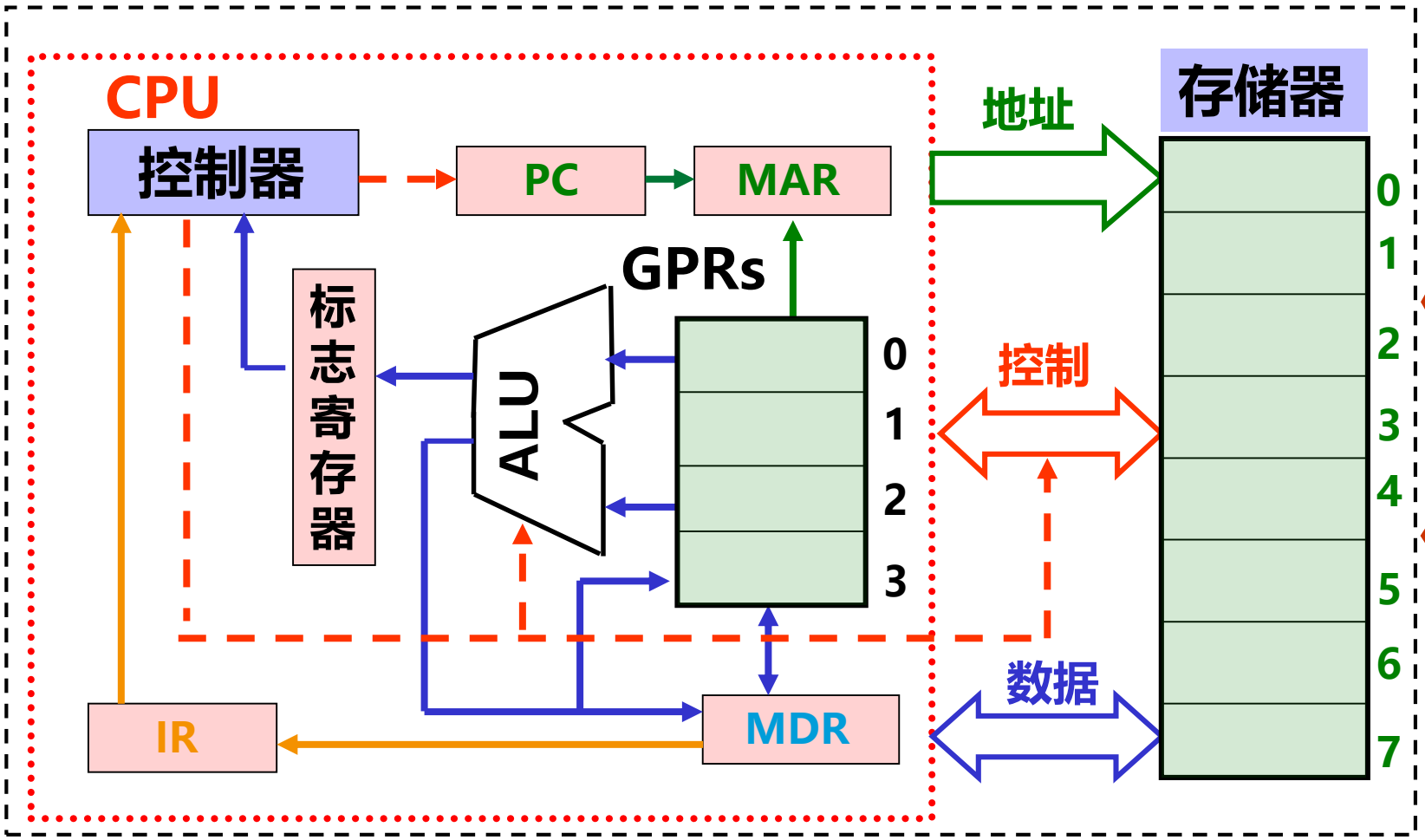
IA-32架构

Intel Architecture, 32-bit

*www.**intel**.com/…/us/…/64-ia-**32**-**architecture**s-software-developer-vol-1-manual.pdf*


A hypothetical CISC computer
General registers
Status Register
Control Bus
Control Register
µMemory
(µPrograms)
µIP
Buffers
Controllers
Starting Address Generator (SAG)
Temporary
registers
Internal Bus
Instruction Decoder
Memory
Unit
I/O Unit
Instruction Register (IR)
Arithmetic/Logic
Unit (ALU)
Instruction Pointer (IP)
Operand 1
Adder
Operand 2
Result
Addressing Unit
Memory Address Register (MAR)
Address Bus
etc.
Memory Data Register (MDR)
Data Bus
CPU
Paul²son CS 271

现代计算机结构模型



你能想到计算机相当于现实生活中的什么呢？

指令集架构

- 指令码：处理器制造厂商在芯片内部**预先定义**的一串二进制代码，它规定了处理器的功能和任务，也称为机器语言。
- 指令集架构（Instruction Set Architecture， ISA）又称指令集或指令集体系，是计算机体系结构中与程序设计有关的部分，包含了**基本数据类型**，**指令集**，**寄存器**，**寻址模式**，存储体系，中断，异常处理以及外部I/O。
  - 指令集架构包含一系列的指令码以及由特定处理器执行的基本命令。
- 微架构：一套用于执行指令集的微处理器设计方法。
  - 使用不同微架构的电脑可以共享一种指令集。例如，Intel的Pentium和AMD的AMD Athlon，两者几乎采用相同版本的x86指令集体系，但是两者在内部设计上有本质的区别。

复杂指令集计算
Complex Instruction Set Computing

- 是一种微处理器指令集架构，每个指令可执行若干低阶操作，如从内存读取、储存和计算操作，全部集于单一指令之中。
- 特点：
  - 指令数目多而复杂，
  - 每条指令字长并不相等，电脑必须加以判读，并为此付出了性能的代价。
- 对常用的简单指令会以硬件线路控制尽全力加速，不常用的复杂指令则交由微码循序器“慢慢解码、慢慢跑”

当时的看法是硬件比编译器更容易设计。
另一因素是缺乏大容量的内存，高消息密度的程序更实用。

精简指令集计算
reduced instruction set computing

- RISC：这种设计思路，对指令数目和寻址方式都做了精简，使其实现更容易，指令并行执行程度更好，编译器的效率更高。
- 目前常见的精简指令集微处理器包括DEC Alpha、ARM、MIPS、PA-RISC、Power Architecture和SPARC等。
- 这种指令集的特点是指令数目少，每条指令都采用标准字长、执行时间短、中央处理器的实现细节对于机器级程序是可见的等等。
- RISC与CISC在竞争的过程中相互学习，现在的RISC指令集也达到数百条，运行周期也不再固定。

Intel X86 处理器

- Intel公司的CPU占领主流地位
- 第一款微处理器:是1971年11月英特尔公司推出的4004微处理器。
- 循序渐进的设计
  - 开始于1978年的8086单芯片
  - 随着时间推移增加很多新的特性
  - 尽管有些特性已经过时，但仍然被兼容着
- 采用复杂指令集系统计算机
  - 指令集由具有不同格式的多种不同指令构成
    - 但是只有小部分子集被用到Linux程序中

Intel X86 CPU 演化

Name	*Date*	*Transistors*	*MHz*
8086	1978	29K	5-10
386	1985	275K	16-33
Pentium4F	2004	125M	2800-3800
Core i7	2008	731M	2667-3333

- 第1个16位处理器. 构成了IBM PC&DOS系统的基础
- 1MB地址寻址空间. DOS系统只留给用户640K
- 第1个32位处理器，被称为IA 32, x86-32,i386
- 加了平面寻址方式
- 可运行Unix，32位的Linux/gcc 没有使用后续推出的新指令
- 64位处理器, 被称为x86-64

通常按照处理器芯片支持的指令码的数量和类型进行分类。

Intel X86 CPU 演化

Name	*Date*	*Transistors*
Itanium	2001	10M
Itanium 2	2002	221M
Itanium 2 Dual-Core	2006	1.7B

- Itanium
  - 扩展到IA64，64位体系结构
  - 为了更高性能**全新设计的指令集**
  - 能够运行已经存在的IA32程序
    - 片上 “x86 引擎”
- Itanium 2
  - 极大性能提升
- Itanium 2 Dual-Core

X86体系结构演化

Architectures	Processors
X86-16	8086
	286
X86-32/IA32	386
	486
	Pentium
MMX	Pentium MMX
SSE	Pentium III
SSE2	Pentium 4
SSE3	Pentium 4E
X86-64 / EM64t	Pentium 4F
	Core 2 Duo
SSE4	Core i7

- 增加支持多媒体的指令
- 支持在1、2、4个字节整数和浮点运算的平行处理

- Intel processors (Wikipedia)
- Intel microarchitectures

time

Linux/GCC 演化：
支持32位386
支持64位 x86-64

IA: often redefined as latest Intel architecture

X86-64

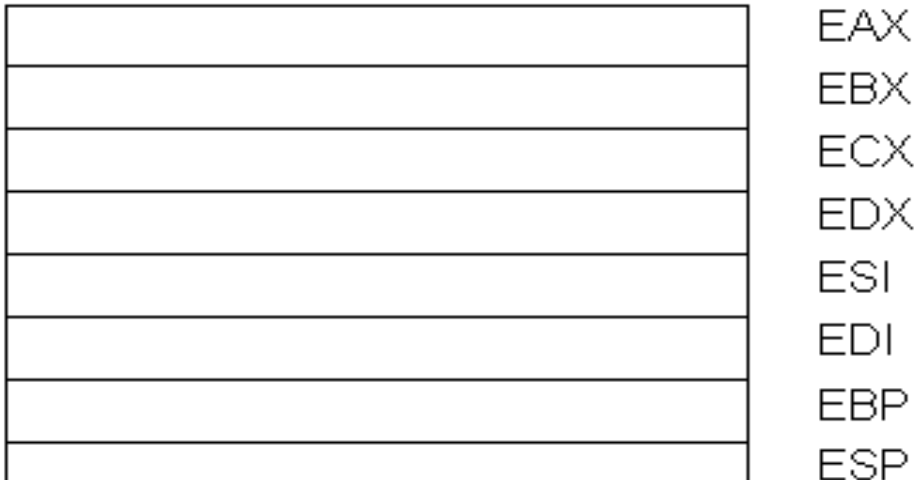
- X86-64于1999年由AMD设计，是64位版本的x86指令集，兼容于16位及32位的x86架构。AMD称为“AMD64”。
- 其后也为英特尔所采用，英特尔称之为“Intel 64”、“IA-32e”及“EM64T”。
- 一般称为：x86-64、x86_64、x64。
- AMD64架构在IA-32基础上，既有支持64位通用寄存器、64位整数及逻辑运算、以及64位虚拟地址，设计人员又为架构作出不少改进。
- 64位寻址模式-长模式，是物理地址扩展（PAE）的超集；内存页大小可以是4KB，2MB，或1GB。
- 虚拟地址有64位的宽度，但目前的实现机制并不允许整个16EB的虚拟地址空间都被使用。Windows在x64上的实现仅应用了16TB，即44位的宽度。

IA-32 CPU执行模式

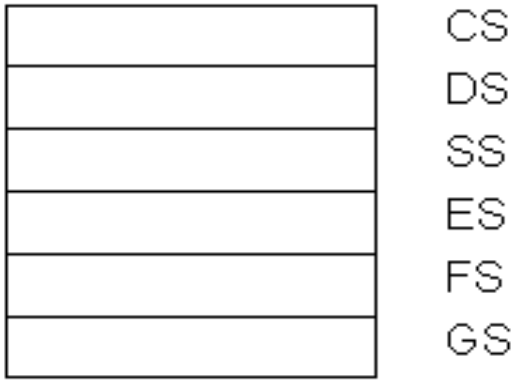
- 实地址模式Real Mode
  - 为与8086/8088兼容而设，加电或复位时
  - 寻址空间为1MB，20位地址：(CS)<<4+(IP)
- 保护模式Protected Mode
  - 加电后进入，采用虚拟存储管理，多任务情况下隔离、保护
  - 80286以上高档微处理器最常用的工作模式
  - 寻址空间为2^{32}B，32位线性地址分段（段基址+段内偏移量）
- 系统管理模式System Management mode SMM：只能通过系统管理中断SMI进入，并只能通过执行RSM指令退出。
- 虚拟86模式Virtual 86 mode:虚拟-真实模式，允许在保护模式操作系统的控制下执行实地址模式的程序。

IA-32 寄存器组

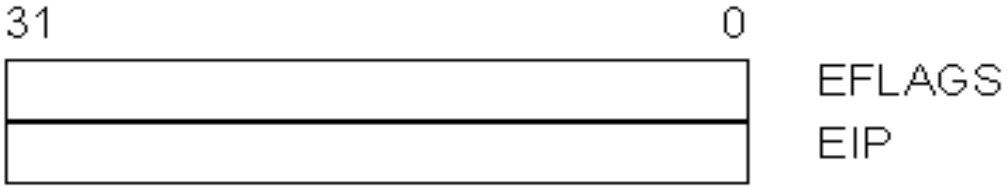
通用数据寄存器：8个



段寄存器：6个



状态和控制寄存器：2个



通用数据寄存器

- 8个32位通用数据寄存器，用于保存逻辑和算术运算的操作数、地址运算和内存指针的操作数：
  - EAX：累加器操作数和结果的数据。
  - EBX：指针在DS段的数据。
  - ECX：计数器字符串和循环操作。
  - EDX：I/O指针。
  - ESI：指针在段数据指向DS注册; 源指针字 符串操作。
  - EDI：指针在段数据（或目标）指出，由ES寄存器; 目的地指针字 符串操作。
  - EBP：指针堆栈上的数据（SS段）。
  - ESP：堆栈指针（SS段）。

通用数据寄存器

General-purpose registers

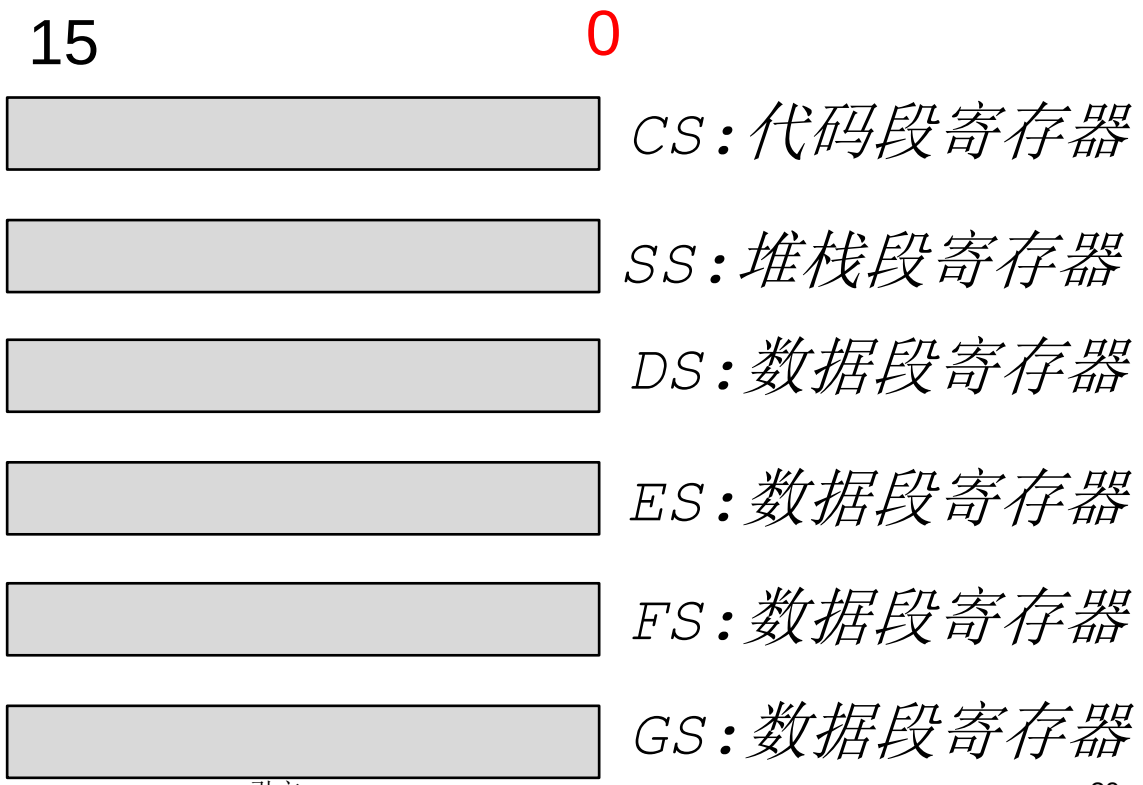
31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			ESI
	SI			EDI
	DI			EBP
	SP			ESP

通用数据寄存器

- 通用寄存器的低16位寄存器称为：AX、BX、CX、DX、BP、SP、SI和DI（对应的32位寄存器的名字都有一个前缀“E”）。 EAX 、EBX、ECX和EDX每个寄存器可以由名称进行引用AH，BH，CH，和DH（高字节）和AL，BL，CL，和DL（低字节）。

段寄存器

- 有6个16位段地址寄存器。 段地址是一个特殊的指针标识的内存段。
- 四个数据段寄存器提供灵活、高效的方式来访问数据的程序。
- 现代的操作系统和应用程序使用（不分段）内存模型，所有段地址的值相同。
- 编写系统程序除外。



标志寄存器EFLAGS

32位EFLAGS寄存器包含一组状态标志、控制标志以及一组系统标志。

6个条件标志

OF、SF、ZF、CF、AF、PF

3个控制标志

DF：方向标志

IF：中断允许标志

TF：陷阱标志

2016年7月1日

功能	EFLAG寄存器位或位
ID标志（ID）	21（系统）
虚拟中断挂起（VIP）	20（系统）
虚拟中断标志（VIF）	19（系统）
对齐检查（AC）	18（系统）
虚拟8086模式（VM）的	17（系统）
恢复标志（RF）	16（系统）
嵌套任务（NT）	14（系统）
I / O特权级别（IOPL）	13至12（系统）
溢出标志（OF）	11（系统）
方向标志（DF）	10（系统）
中断允许标志（IF）	9（系统）
陷阱标志（TF）	8（系统）
符号标志（SF）	7（状态）
零标志（ZF）	6（状态）
辅助进位标志（AF）	4（状态）
奇偶标志（PF）	2（状态）
进位标志（CF）	0（状态）

EIP 指令寄存器

- EIP寄存器（或指令指针），也被称为“程序计数器PC”。 它存放的是在当前代码段中要执行的下一条指令地址的偏移量。 它可以是指令地址顺序产生也可能是由跳转、调用、返回等指令产生。
- EIP的不能由软件直接访问; 它是由控制转移指令、中断产生和异常隐含控制。
- EIP寄存器可以间接通过修改程序堆栈上的返回指令指针的值，并执行返回指令（RET或IRET）。
- EIP的值可能不匹配，因为指令预取了当前指令。
- 读取EIP的唯一方法是执行CALL指令，然后读取从程序堆栈中的返回指令指针的值。

X86-64架构

- 64位架构
  - IA-64：Intel最早推出的64位架构是基于超长指令字VLIW技术的IA-64体系结构，安腾和安腾2分别在2000年和2002年问世。
  - AMD64：AMD公司在2003年推出兼容IA-32的64位版本指令集x86-64，AMD后来将x86-64更名为AMD64。
  - Intel64：Intel在2004年推出IA32-EM64T，它支持x86-64指令集。Intel为了表示EM64T的64位模式特点，又使其与IA-64有所区别，2006年开始把EM64T改名为Intel 64。
- 主流：x86-64架构

X86-64架构

- 字长从32位变为64位。
- 所有GPRs都从32位扩充到64位，Rax、Rbx~Rsp
- 新增8个64位通用寄存器：R8-R15；R8B、R8W、R8D。
- long double型数虽还采用80位扩展精度格式，但所分配存储空间从12B扩展为16B，即改为16B对齐，但不管是分配12B还是16B，都只用到低10B。
- 过程调用时，通常用通用寄存器而不是栈来传递参数。因此大多数情况下执行时间比IA-32代码短。
- 128位的MMX寄存器从原来的8个增加到16个，浮点操作采用基于SSE的面向XMM寄存器的指令集，而不采用基于浮点寄存器栈的指令集 。

X86-64架构

- 过程调用的参数传递
  - 通过寄存器传送参数
  - 最多可有6个整型或指针型参数通过寄存器传递
  - 超过6个入口参数时，后面的通过栈来传递
  - 在栈中传递的参数若是基本类型，则都被分配8个字节

操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

x86-64 整数寄存器

%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

- 拓展了整数寄存器，增加8个新寄存器
- 可以访问8、16、32、64位

x86-64 整数寄存器习惯用法

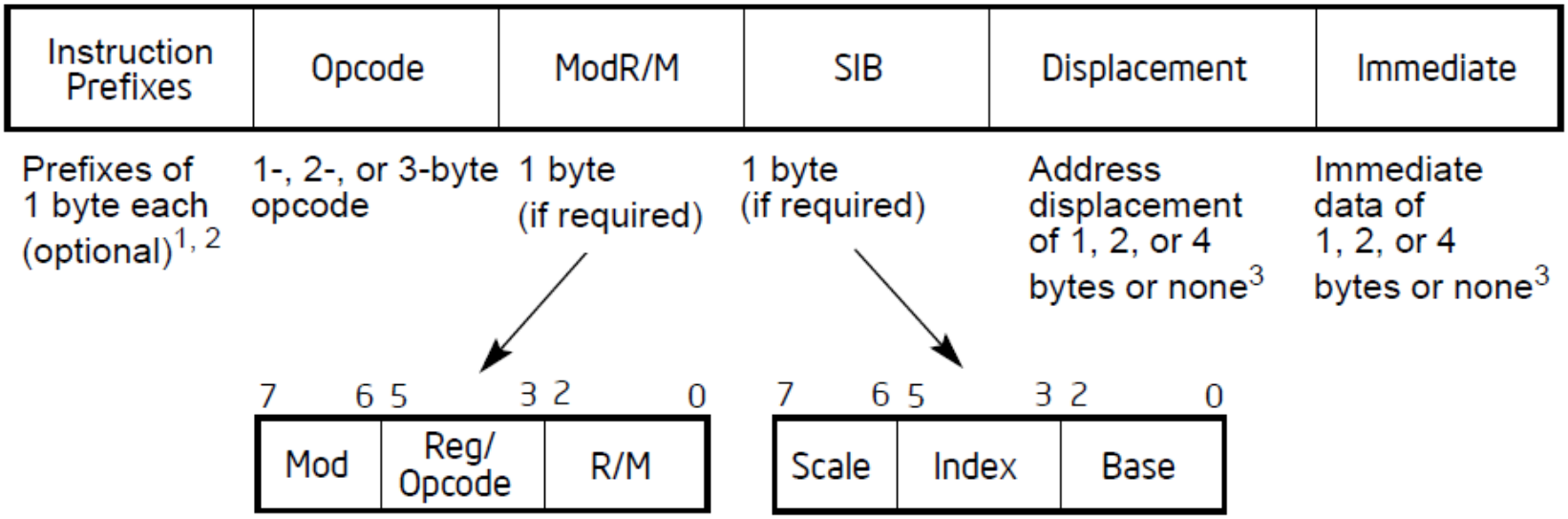
%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Argument #5
%r9	Argument #6
%r10	Caller saved
%r11	Caller Saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

IA32、X86-64寄存器组

Register or Stack	Legacy and Compatibility Modes			64-Bit Mode		
	Name	Number	Size (bits)	Name	Number	Size (bits)
General-Purpose Registers (GPRs)	EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP	8	32	RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8–R15	16	64
128-Bit XMM Registers	XMM0–XMM7	8	128	XMM0–XMM15	16	128
64-Bit MMX Registers	MMX0–MMX7	8	64	MMX0–MMX7	8	64
x87 Registers	FPR0–FPR7	8	80	FPR0–FPR7	8	80
Instruction Pointer	EIP	1	32	RIP	1	64
Flags	EFLAGS	1	32	RFLAGS	1	64
Stack	–		16 or 32	–		64

IA-32指令码格式



1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, “REX Prefixes” for additional information.
2. For VEX encoding information, see Section 2.3, “Intel® Advanced Vector Extensions (Intel® AVX)”.
3. Some rare instructions can take an 8B immediate or 8B displacement.

IA-32指令码格式

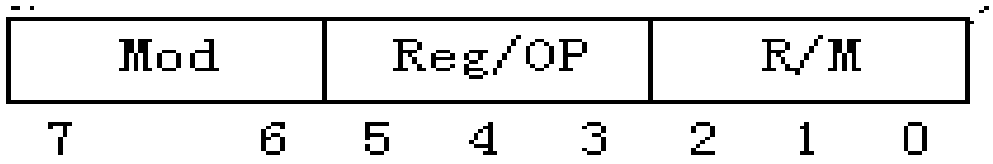
- 处理器芯片按照指令码完成相应的操作。
- 指令码包含1个或多个处理器需要处理的信息。
- IA-32指令码格式，分四个部分：

名称	指令前缀	操作码	修饰符			数据元素
			ModR/M	SIB	移位	
字节数	0~4	1~3	0~1	0~1	0~4	0~4

- 指令前缀：包含1到4个修改操作码行为的1字节前缀
  - 锁定前缀和重复前缀：锁定前缀表示指令将独占共享内存区域
  - 段覆盖前缀和分支提示前缀：段覆盖前缀覆盖定义了的段寄存器的值
  - 操作数长度覆盖前缀：通知处理器，程序将切换16位和32位操作数
  - 地址长度覆盖前缀：通知处理器，程序将切换16位和32位内存地址

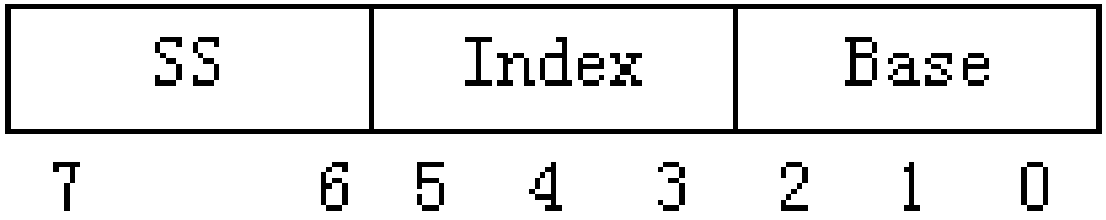
IA-32指令码格式

- 操作码：定义了处理器执行的基本功能和任务，是IA-32指令码格式中唯一必须存在的部分，1到3个字节长。比如0F A2定义了IA-32 CPUID指令
- 修饰符：定义了执行的功能中涉及的寄存器和内存位置，包含3个独立的部分
  - ModR/M字节：寻址方式说明符，1个字节
    - 3个字段的信息构成：Mod(2bit) reg/opcode(3bit) r/m(3bit)
    - Mod字段和r/m字段一起使用，定义指令用使用的寄存器或者寻址模式
    - Reg/opcode字段用于允许使用更多的3位进一步定义操作码功能或者用于定义寄存器。



IA-32指令码格式

- SIB字节：比例-索引-基地址，1个字节
  - 由3个字段组成
  - 比例字段：指定操作的比例因子
  - 索引字段指定内存访问中用作索引寄存器的寄存器
  - 基址字段指定用作内存访问的基址寄存器的寄存器



- 移位字节：1、2或4个的地址移位字节，4个字节，用来指定对于ModR/M和SIB字节中定义的内存位置的偏移量。
- 数据元素：指令中包含的数据，称为立即数。可以包含1、2或4字节的信息。

IA-32指令码格式

- 指令码示例：C7　45　FC　01　00 00 00
  - C7：操作码，定义把值传送到内存指定位置（MOV）
  - 45 FC：修饰符，定义内存位置是从EBP寄存器（45）中值指向的内存位置开始的4个字节（FC）
  - 01 00 00 00：定义传送的数值，立即数1。
- 指令码由01序列组成，难以记忆
- 使用助记符表示—汇编语言
  - 操作码和汇编指令一一对应，都是机器级指令
  - 汇编指令是机器指令的符号表示

Mod	Reg/OP	R/M

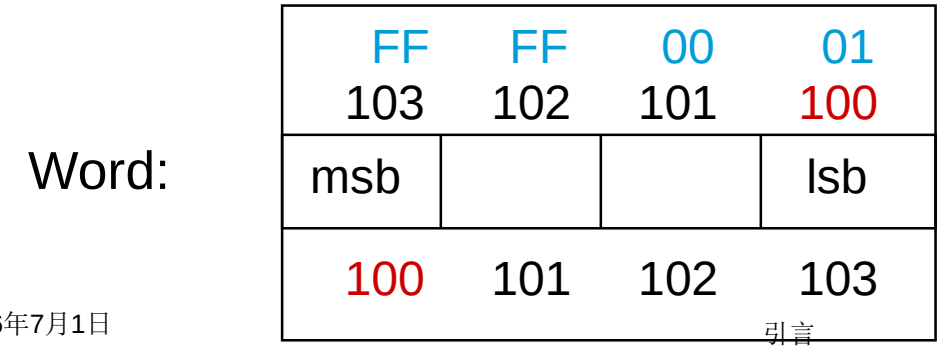
Effective Address	Mod	R/M	7	6	5	4	3	2	1	0		
[EAX]	00	000	00	08	10	18	20	28	30	38		
[ECX]		001	01	09	11	19	21	29	31	39		
[EDX]		010	02	0A	12	1A	22	2A	32	3A		
[EBX]		011	03	0B	13	1B	23	2B	33	3B		
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C		
disp32		101	05	0D	15	1D	25	2D	35	3D		
[ESI]		110	06	0E	16	1E	26	2E	36	3E		
[EDI]		111	07	0F	17	1F	27	2F	37	3F		
disp8[EAX]	01	000	40	48	50	58	60	68	70	78		
disp8[ECX]		001	41	49	51	59	61	69	71	79		
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A		
disp8[EBX];		011	43	4B	53	5B	63	6B	73	7B		
disp8[--][--]		100	44	4C	54	5C	64	6C	74	7C		
disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D		
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E		
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F		
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8		
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9		
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA		
disp32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB		
disp32[--][--]		100	84	8C	94	9C	A4	AC	B4	BC		
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD		
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE		
disp32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF		
EAX/AX/AL	11	000	C0	C8	D0	D8	E0	E8	F0	F8		
ECX/CX/CL		001	C1	C9	D1	D9	E1	E9	F1	F9		
EDX/DX/DL		010	C2	CA	D2	DA	E2	EA	F2	FA		
EBX/BX/BL		011	C3	CB	D3	DB	E3	EB	F3	FB		
ESP/SP/AH		100	C4	CC	D4	DC	E4	EC	F4	FC		
EBP/BP/CH		101	C5	CD	D5	DD	E5	ED	F5	FD		
ESI/SI/DH		110	C6	CE	D6	DE	E6	EE	F6	FE		
EDI/DI/BH		111	C7	CF	D7	DF	E7	EF	F7	FF		

IA-32的数据类型和字节顺序

- IA-32 提供四种数据类型：
  - 一个字节（8比特）
  - 一个字（16位）
  - 一个双字（32位）
  - 一个四倍长字（64位）

若 int i = -65535，存放在100号单元（占100～103），则用“取数”指令访问100号单元取出 i 时，必须清楚 i 的4个字节是如何存放的。

- 80年代开始，几乎所有机器都用字节编址
- IA-32处理器使用“小端模式Little Endian”的字节顺序
  - 数据的最低有效位存放在地址字节的最低位。



IA-32存储器寻址方式

- 寻址方式：根据指令给定信息得到操作数或操作数地址。
- IA-32支持平面存储模式和分段存储模式
- 平面存储模式下：内存使用独立连续的地址空间,称为线性地址空间,程序、数据和堆栈都包含在同一的地址空间中，采用字节寻址，范围0～2^{32}-1.
- 分段存储器模式：代码、数据和堆栈通常包含在单独的段中，为了表示一个段内的一个地址， 程序必须计算一个逻辑地址，由一个段地址加上偏移量组成。段地址表明可以访问的段，偏移量表示在段地址空间中的位置。

保护模式下的寻址方式

寻址方式	说明
立即寻址	指令直接给出操作数
寄存器寻址	指定的寄存器R的内容为操作数
位移	LA=(SR)+A
基址寻址	LA=(SR)+(B)
基址加位移	LA=(SR)+(B)+A
比例变址加位移	LA=(SR)+(I)×S + A
基址加变址加位移	LA=(SR)+(B)+(I)+A
基址加比例变址加位移	LA=(SR)+(B)+(I)×S + A
相对寻址	LA=(PC)+A

存储器操作数

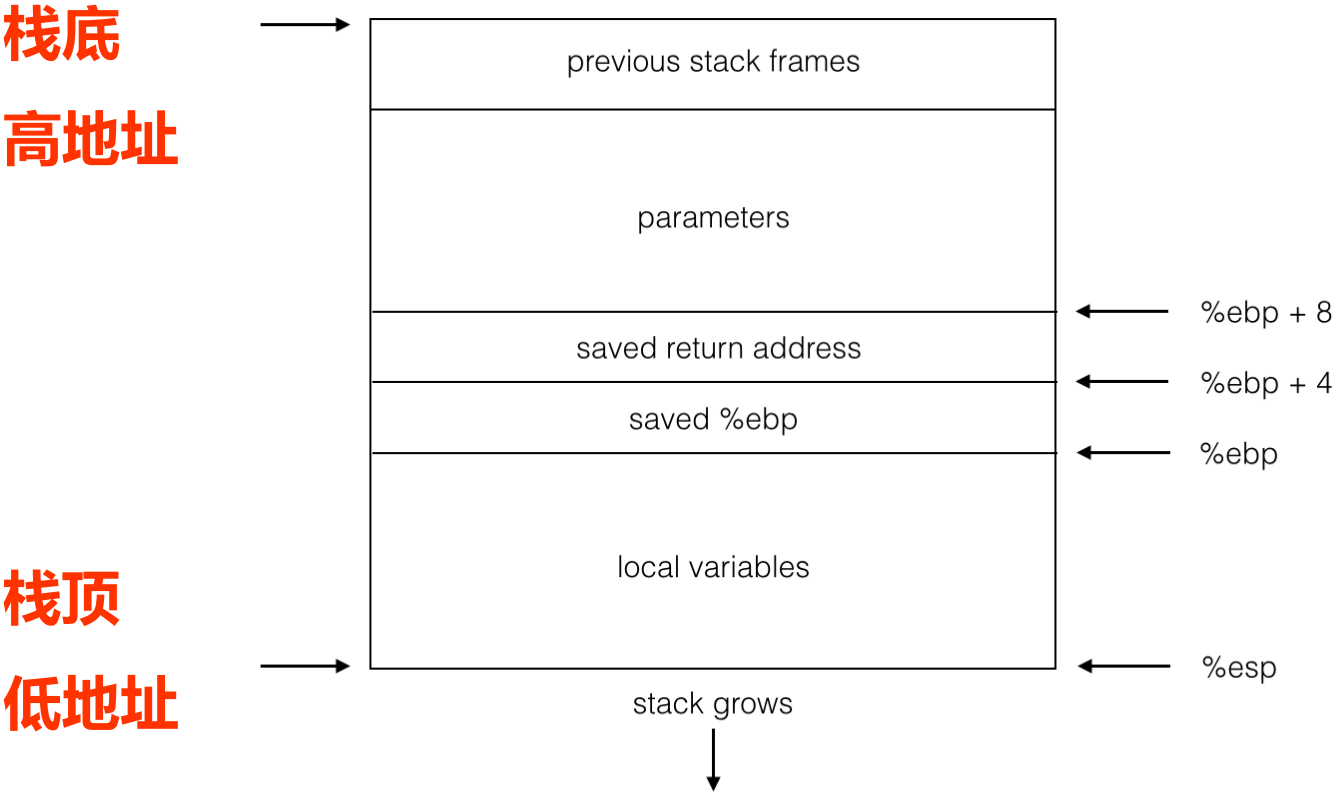
跳转目标指令地址

注：LA:线性地址　(X):X的内容　SR:段寄存器　PC:程序计数器　R:寄存器
A:指令中给定地址段的位移量　B:基址寄存器　I:变址寄存器　S:比例系数

- SR段寄存器（间接）确定操作数所在段的段基址
- 有效地址给出操作数在所在段的偏移地址
- 寻址过程涉及到“分段虚拟管理方式”

IA-32堆栈结构

栈（Stack）是一种采用“先进后出”方式进行访问的一块存储区，用于嵌套过程调用。从高地址向低地址增长



汇编语言

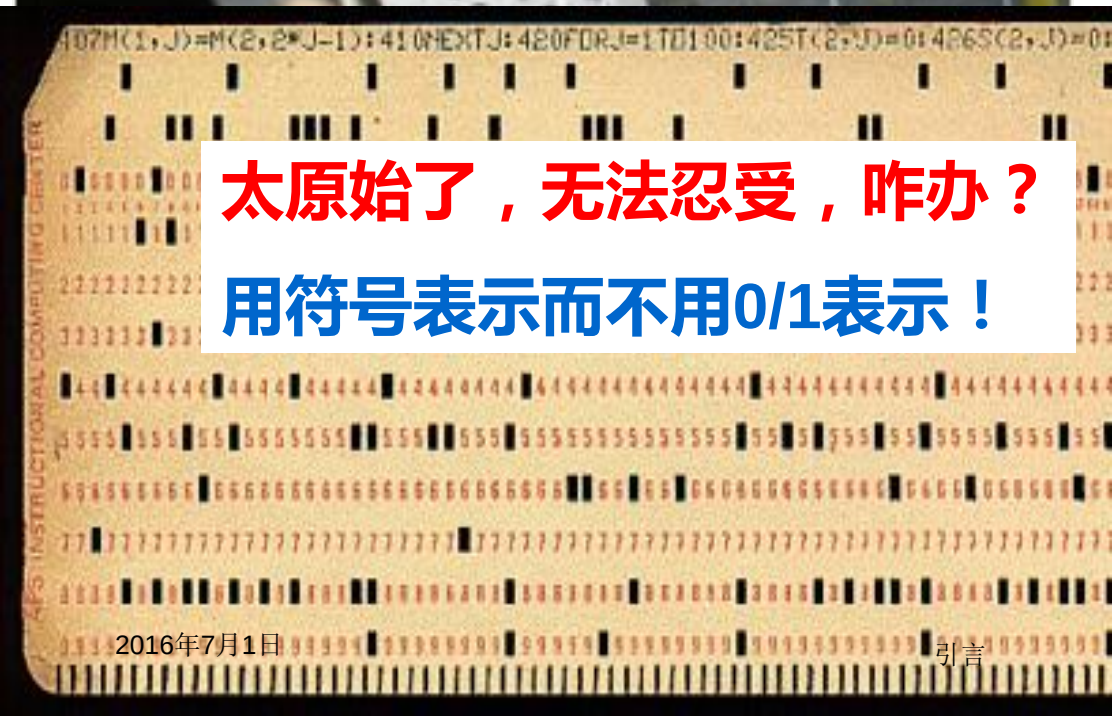
机器语言编程

- 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 : ……
```

你认为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）来代替和表示特定低级机器语言的操作。
- 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，因此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应用。
- 汇编语言能以让学生更深入了解计算机系统的运行原理，因此依然是计算机科学类专业学生的必修课之一。

汇编语言

- 汇编语言使用助记符(mnemonics)表示机器指令码，助记符类似英语单词的格式。
- 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

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
    - 只是在存储器中的连续字节
- 基本操作
  - 在寄存器或存储器的数据上的执行算术操作
  - 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. GNU

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]
```

- AT&T 与 Intel的不同
  - 目标操作数则在源操作数的右边，Intel汇编中操作数顺序相反

    addl $4,%eax　　add eax,4
  - 用‘$’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的最后一个字母决定的，后缀‘b’、‘w’、‘l’、’q’分别表示字长为8、16、32、64位. 如`subl $8,%esp`
  - AT&T中的内存操作数的格式形如：SECTION:DISP(BASE,INDEX,SCALE)，如`-8(%ebp)`

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

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通用编译器也可以用来在一个步骤内完成编译和链接汇编程序
- 但是gcc汇编程序时，查找的是main标签，因此需要把_start标签改为main标签：
  - .section .text
  - .golbl main
  - main:

$ gcc –o cpuid cpuid.s

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	只进行编译和汇编（不连接）
-S	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

调试工具GDB

- gdb（GNU Debugger）是由GNU计划完成的、受通用公共许可证（GPL）保护的自由软件。它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
```

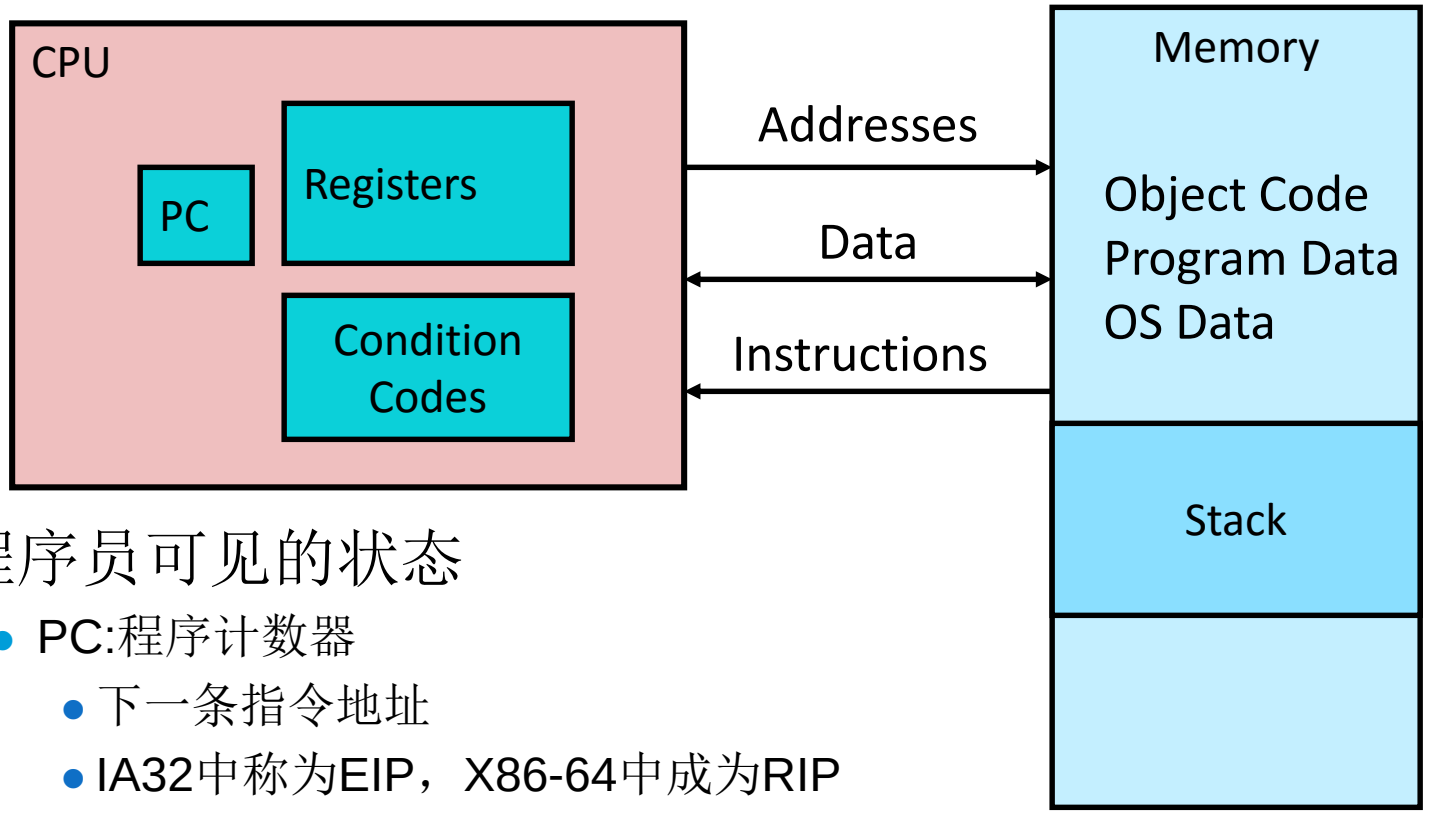
Objdump命令

- 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

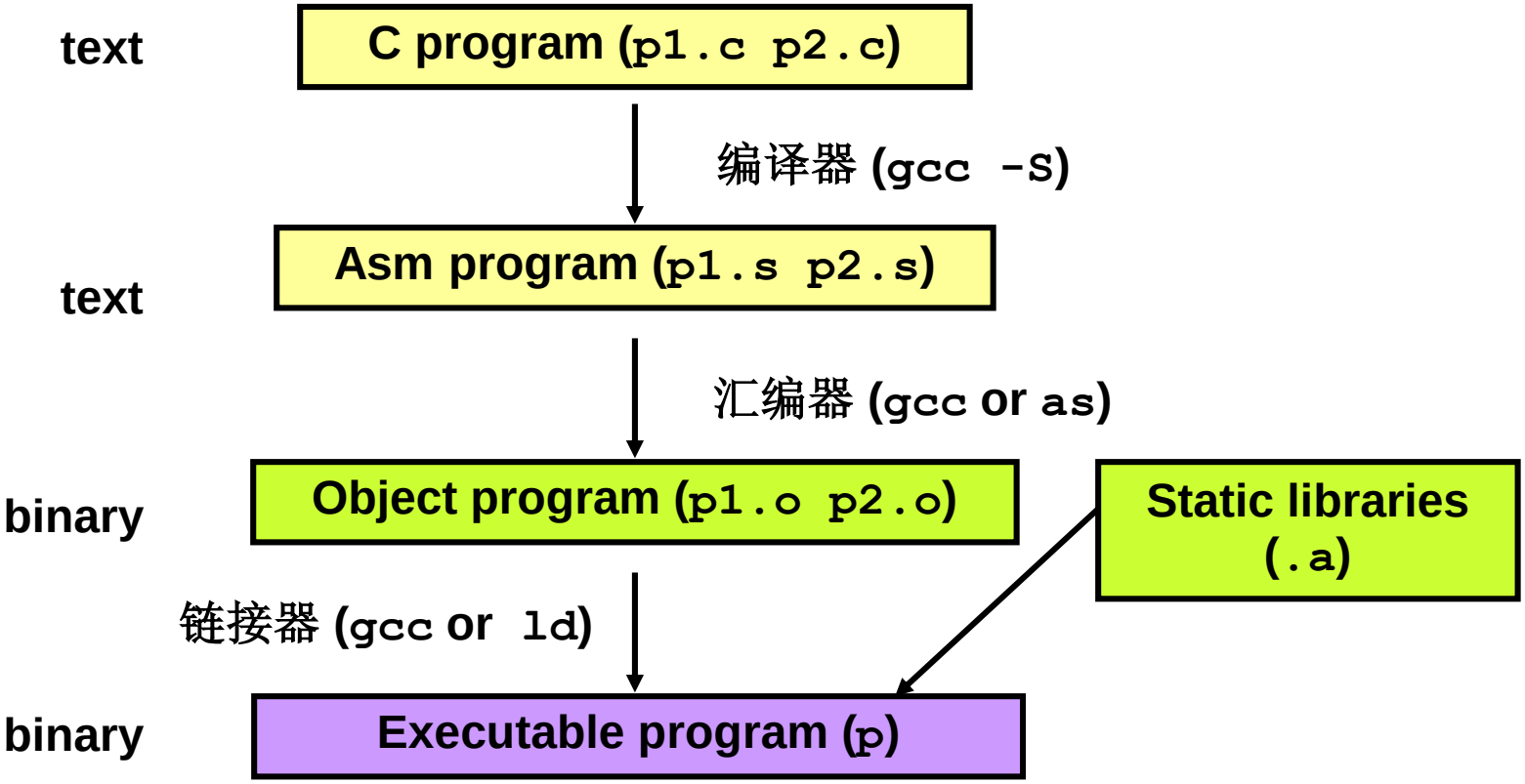
汇编程序员的视角



- 程序员可见的状态
  - PC:程序计数器
    - 下一条指令地址
    - IA32中称为EIP，X86-64中成为RIP
  - 寄存器
    - 程序中频繁使用的数据
  - 条件码
    - 保存最近算术操作状态信息
    - 用于条件分支
  - 内存
    - 字节可寻址数组
    - 代码，用户数据，一些操作系统数据
    - 包含支持过程调用的栈

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



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

世界编程语言排行榜

Jun 2015	Jun 2014	Change	Programming Language	Ratings	Change
1	2	↑	Java	17.822%	+1.71%
2	1	↓	C	16.788%	+0.60%
3	4	↑	C++	7.756%	+1.33%
4	5	↑	C#	5.056%	+1.11%
5	3	↓	Objective-C	4.339%	-6.60%
6	8	↑	Python	3.999%	+1.29%
7	10	↑	Visual Basic .NET	3.168%	+1.25%
8	7	↓	PHP	2.868%	+0.02%
9	9		JavaScript	2.295%	+0.30%
10	17	⇈	Delphi/Object Pascal	1.869%	+1.04%

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

汇编语言的排名

Other programming languages

The complete top 50 of programming languages is listed below. This overview is published unofficially, because it could be the ca
that we missed a language. If you have the impression there is a programming language lacking, please notify us at tpci@tiobe.cc
Please also check the overview of all programming languages that we monitor.

Position	Programming Language	Ratings
21	OpenEdge ABL	0.796%
22	SAS	0.787%
23	Assembly language	0.754%
24	Dart	0.671%
25	Scratch	0.628%
26	D	0.610%
27	Fortran	0.582%
28	Lua	0.546%
29	Logo	0.536%
30	Scala	0.531%

计算机性能评价

计算机性能的基本评价指标

- 计算机有两种不同的性能
  - Time to do the task
    - 响应时间（response time）
    - 执行时间（execution time）
    - 等待时间或时延（latency）
  - Tasks per day, hour, sec, ns. -
    - 吞吐率（throughput）
    - 带宽（bandwidth）

不同应用场合用户关心的性能不同：

-要求吞吐率高的场合，例如：

多媒体应用（音/视频播放要流畅）

-要求响应时间短的场合：例如：

事务处理系统（存/取款速度要快）

-要求吞吐率高且响应时间短的场合：

ATM、文件服务器、Web服务器等

- 基本的性能评价标准是：CPU的执行时间

“机器X的速度（性能）是Y的n倍”的含义：

$$\frac{ExTime(Y)}{ExTime(X)} = \frac{Performance(X)}{Performance(Y)} = n$$

相对性能用执行时间的倒数来表示！

CPU执行时间的计算

CPI：Cycles Per Instruction

CPU 执行时间 = CPU时钟周期数 / 程序 X 时钟周期

= CPU时钟周期数 / 程序 ÷ 时钟频率

= 指令条数 / 程序 X CPI X 时钟周期

CPU时钟周期数 / 程序 = 指令条数 / 程序 X CPI

CPI = CPU时钟周期数 / 程序 ÷指令条数 / 程序

CPI 用来衡量以下各方面的综合结果

- Instruction Set Architecture（ISA）
- Implementation of that architecture
  （Organization & Technology）
- Program（Compiler、Algorithm）

Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	instr. Count	CPI	clock rate
Programming			
Compiler			
Instr. Set Arch.			
Organization			
Technology			

思考：三个因素与哪些方面有关？

例如，{…..
y=4*x;
}

Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	instr. Count	CPI	clock rate
Programming	X	X	
Compiler	X	(X)	
Instr. Set Arch.	X	X	
Organization		X	X
Technology			X

如何计算CPI?

对于某一条特定的指令而言，其CPI是一个确定的值。但是，对于某一个程序或一台机器而言，其CPI是一个平均值，表示该程序或该机器指令集中每条指令执行时平均需要多少时钟周期。

假定CPI_i 和 C_i分别为第i类指令的CPI和指令条数，则程序的总时钟数为：

$$\text{总时钟数} = \sum_{i=1}^{n} CPI_i \times C_i$$

所以，

$$\text{CPU时间} = \text{时钟周期} \times \sum_{i=1}^{n} CPI_i \times C_i$$

假定CPI_i、F_i是各指令CPI和在程序中的出现频率，则程序综合CPI为：

$$\text{CPI} = \sum_{i=1}^{n} CPI_i \times F_i \quad \text{where} \quad F_i = \frac{C_i}{Instructin\_Count}$$

已知CPU时间、时钟频率、总时钟数、指令条数，则程序综合CPI为：

$$\text{CPI} = (\text{CPU 时间} \times \text{时钟频率}) / \text{指令条数} = \text{总时钟周期数} / \text{指令条数}$$

问题：指令的CPI、机器的CPI、程序的CPI各能反映哪方面的性能？

单靠CPI不能反映CPU性能！为什么？

例如，单周期处理器CPI=1，但性能差！

Example1

程序P在机器A上运行需10 s， 机器A的时钟频率为400MHz。 现在要设计一台机器B，希望该程序在B上运行只需6 s.

机器B时钟频率的提高导致了其CPI的增加，使得程序P在机器B上时钟周期数是在机器A上的1.2倍。机器B的时钟频率达到A的多少倍才能使程序P在B上执行速度是A上的10/6=1.67倍？

Answer:

CPU时间A = 时钟周期数A / 时钟频率A

时钟周期数A = 10 sec x 400MHz = 4000M个

时钟频率B = 时钟周期数B / CPU时间B

= 1.2 x 4000M / 6 sec = 800 MHz

机器B的频率是A的两倍，但机器B的速度并不是A的两倍！

Marketing Metrics （产品宣称指标）

MIPS = Instruction Count / Time x10^6

= Clock Rate / CPI x 10^6

Million **I**nstructions **P**er **S**econd （定点指令执行速度）

因为每条指令执行时间不同，所以MIPS总是一个平均值。

- 不同机器的指令集不同
- 程序由不同的指令混合而成
- 指令使用的频度动态变化
- Peak MIPS: （不实用）

用MIPS数表示性能有没有局限?

所以MIPS数不能说明性能的好坏（用下页中的例子来说明）

MFLOPS = FP Operations / Time x10^6

Million **Flo**ating-point Operations **P**er **S**econd（浮点操作速度）

- 不一定是程序中花时间的部分

用MFLOPS数表示性能也有一定局限!

问题：GFLOPS、TFLOPS、PFLOPS等的含义是什么?

全球超级计算机排行2016.6

- TOP 5

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Supercomputing Center in Wuxi China	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway,NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P,NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x,Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom,IBM	1,572,864	17,173.2	20,132.7	7,890
5	RIKEN Advanced Institute for Computational Science (AICS),Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660

Example: MIPS数不可靠！

Assume we build an optimizing compiler for the load/store machine. The compiler discards 50% of the ALU instructions.

1) What is the CPI ?

仅在软件上优化，没涉及到任何硬件措施。

2) Assuming a 20 ns clock cycle time (50 MHz clock rate). What is the MIPS rating for optimized code versus unoptimized code? Does the MIPS rating agree with the rating of execution time?

Op	Freq	Cycle
ALU	43%	1
Load	21%	2
Store	12%	2
Branch	24%	2

Optimizing compiler

21.5/ (21.5+21+12+24)=27%

21 / (21.5+21+12+24)=27%

12 / (21.5+21+12+24)=15%

24 / (21.5+21+12+24)= 31%

New	Freq
	27%
	27%
	15%
	31%

1.57是如何算出来的?

CPI	1.57	50M/1.57=31.8MIPS	1.73
MIPS	31.8	50M/1.73=28.9MIPS	28.9

结果：因为优化后减少了ALU指令（其他指令数没变），所以程序执行时间一定减少了，但优化后的MIPS数反而降低了。

选择性能评价程序（Benchmarks）

- 用基准程序来评测计算机的性能
  - 基准测试程序是专门用来进行性能评价的一组程序
  - 基准程序通过运行实际负载来反映计算机的性能
  - 最好的基准程序是用户实际使用的程序或典型的简单程序
- 基准程序的缺陷
  - 现象：基准程序的性能与某段短代码密切相关时，会被利用以得到不当的性能评测结果
  - 手段：硬件系统设计人员或编译器开发者针对这些代码片段进行特殊的优化，使得执行这段代码的速度非常快
    - 例1：Intel Pentium处理器运行SPECint时用了公司内部使用的特殊编译器，使其性能极高
    - 例2：矩阵乘法程序SPECmatrix300有99%的时间运行在一行语句上，有些厂商用特殊编译器优化该语句，使性能达VAX11/780的729.8倍！