程序的机器级表示V：高级主题

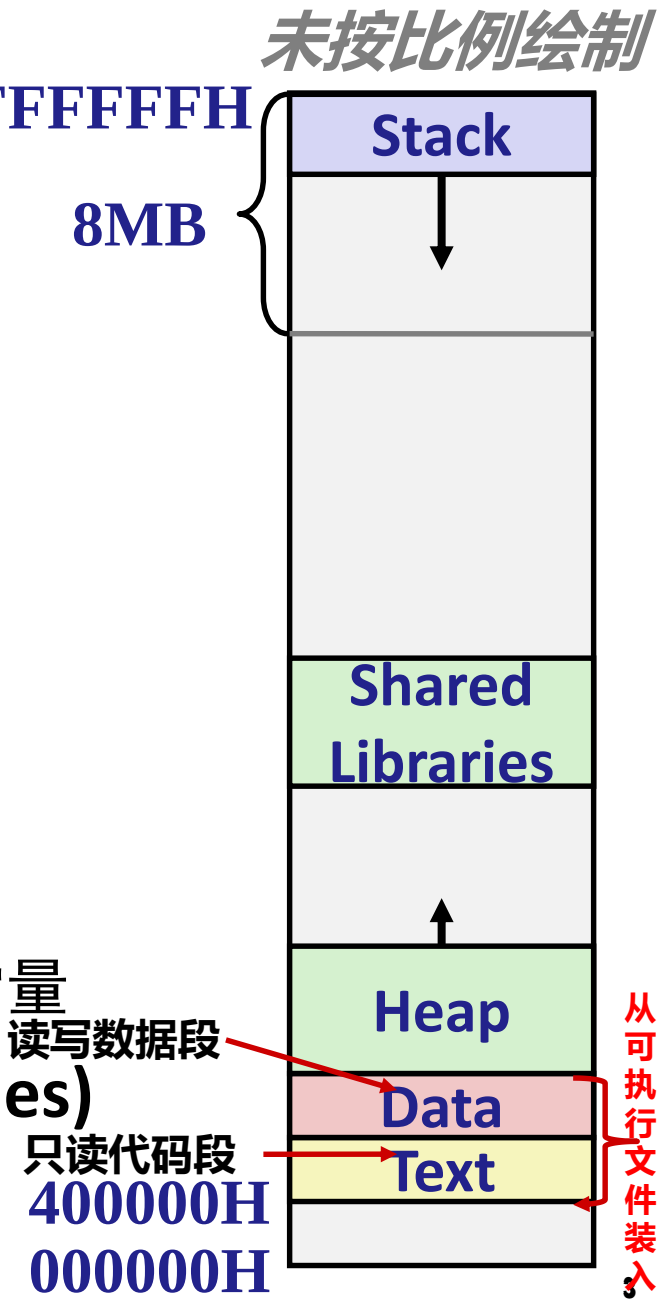
教师：吴锐
计算机科学与技术学院
哈尔滨工业大学

主要内容

- 内存布局
- 缓冲区溢出
  - 安全隐患
  - 防护
- 浮点数

x86-64 Linux 内存布局

- **栈(Stack)**
  - 运行时栈 (8MB limit)
  - 涉及局部变量
- **堆(Heap)**
  - 按需动态分配
  - 时机:调用malloc(), calloc(), new()时
- **数据(Data)**
  - 静态分配的内存中保存的数据
  - 全局变量、static变量、字符串常量
- **代码/共享库(Text / Shared Libraries)**
  - 只读的可执行的机器指令



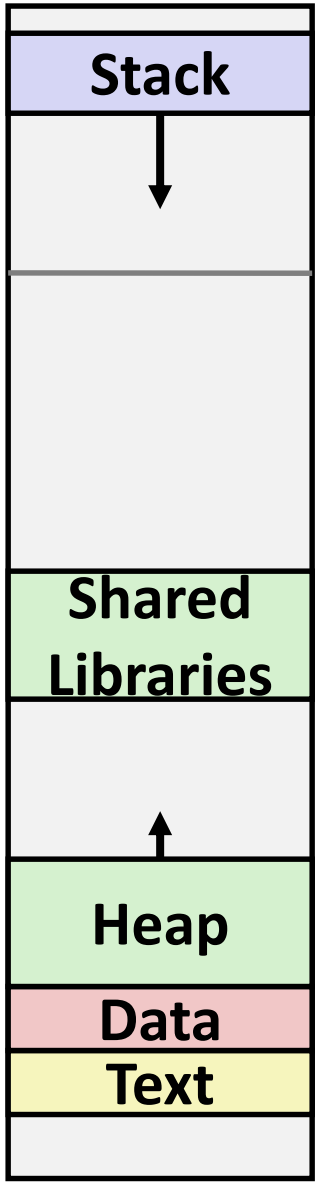
内存分配示例

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;
int useless() { return 0; }
int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```

程序中各个部分都在哪里?

未按比例绘制

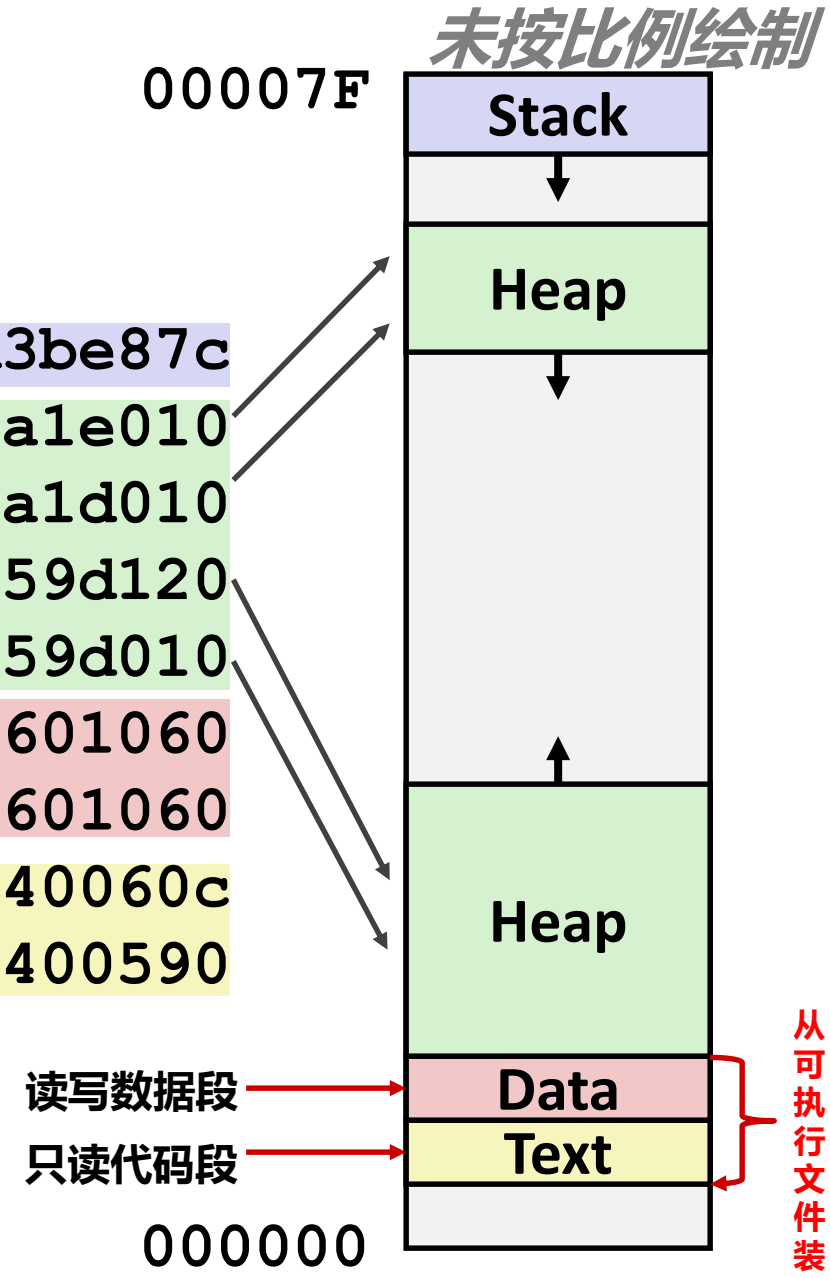


x86-64 例子的地址

未按比例绘制

地址范围 ~2^{47}

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x00000008359d120
p2	0x00000008359d010
big_array	0x00000080601060
huge_array	0x00000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



主要内容

- 内存布局
- **缓冲区溢出**
  - 安全隐患
  - 防护
- 浮点数

回忆: 内存引用的Bug示例

```
typedef struct {
  int a[2];
  double d;
} struct_t;
double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(1)  → 3.14
fun(2)  → 3.1399998664856
fun(3)  → 2.00000061035156
fun(4)  → 3.14
fun(6)  → Segmentation fault
```

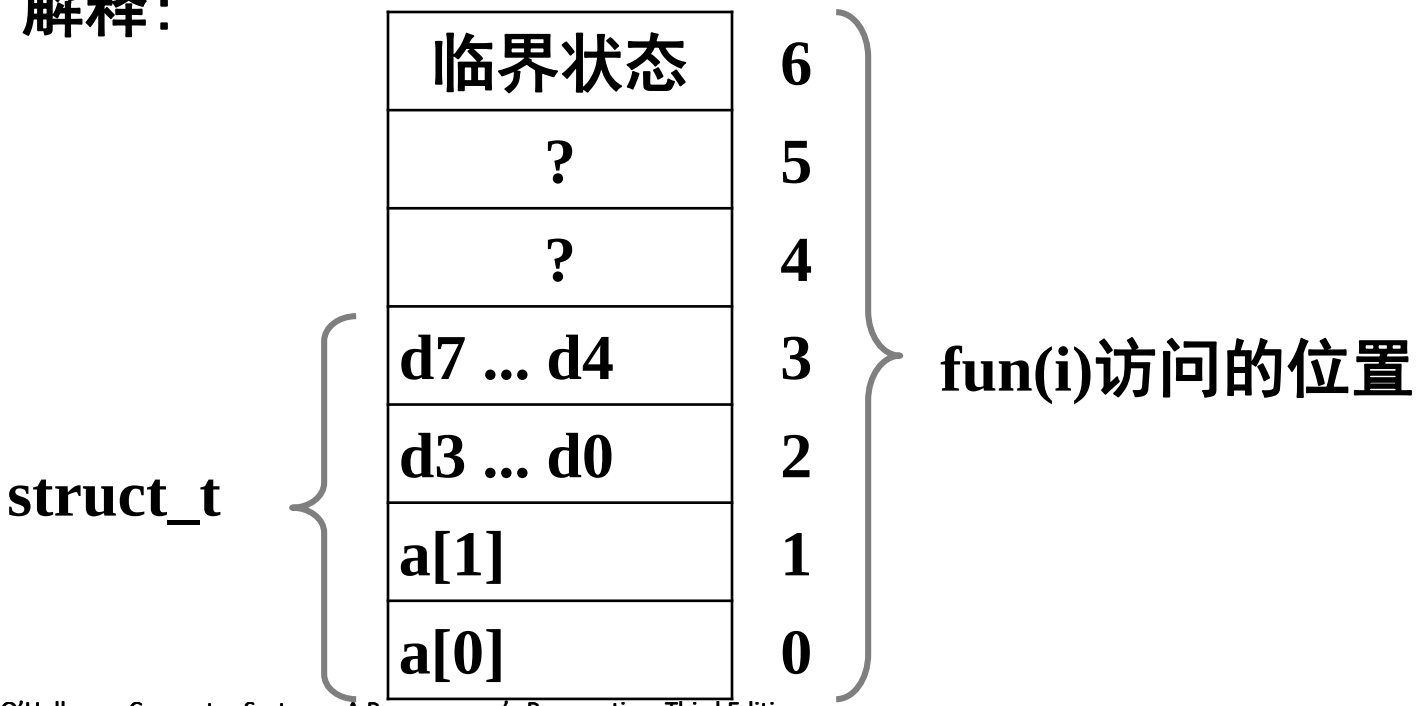
运行结果与系统有关

内存引用的Bug示例

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14
fun(6) → Segmentation fault

解释：



这是个 大 问题。

- 一般称为“缓冲区溢出”
  - 当超出数组分配的内存大小（范围）
- 为何是大问题?
  - 示例 #1安全隐患的技术原因
    - 示例#1总的原因是用户无知
- 更一般的形式
  - 字符串输入不检查长度
  - 特别是堆栈上的有界字符数组
    - 有时称为堆栈粉碎(stack smashing)

字符串库的代码

- Unix函数gets()的实现

```
/* Get string from stdin */
char *gets(char *dest){
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - 无法设定读入字符串的长度限制
- 其他库函数也有类似问题
  - strcpy, strcat: 任意长度字符串的拷贝
  - scanf, fscanf, sscanf, 使用 %s 转换符时

存在安全隐患的缓冲区代码

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

←btw, 多大才足够?

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901230123456789012345678901230
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
Segmentation Fault
```

缓冲区溢出的反汇编

echo:

```
0000000004006cf <echo>:
 4006cf:    48 83 ec 18          sub    $0x18,%rsp
 4006d3:    48 89 e7             mov    %rsp,%rdi
 4006d6:    e8 a5 ff ff ff       callq  400680 <gets>
 4006db:    48 89 e7             mov    %rsp,%rdi
 4006de:    e8 3d fe ff ff       callq  400520 <puts@plt>
 4006e3:    48 83 c4 18          add    $0x18,%rsp
 4006e7:    c3                   retq
```

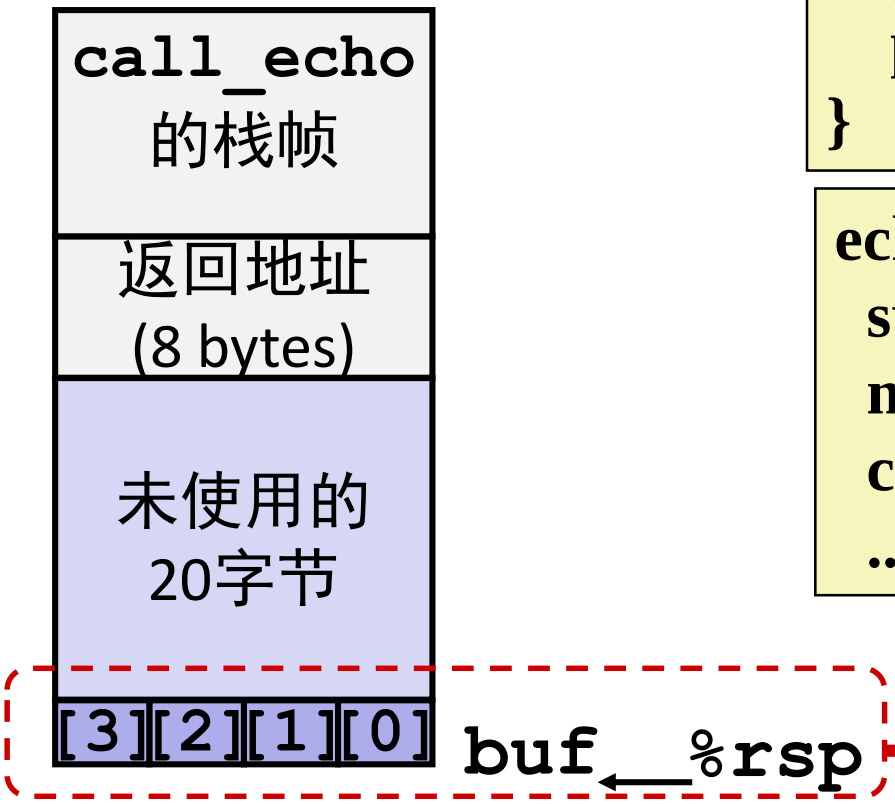
call_echo:

```
4006e8:    48 83 ec 08                sub    $0x8,%rsp
4006ec:    b8 00 00 00 00             mov    $0x0,%eax
4006f1:    e8 d9 ff ff ff       callq  4006cf <echo>
4006f6:    48 83 c4 08                add    $0x8,%rsp
4006fa:    c3                   retq
```

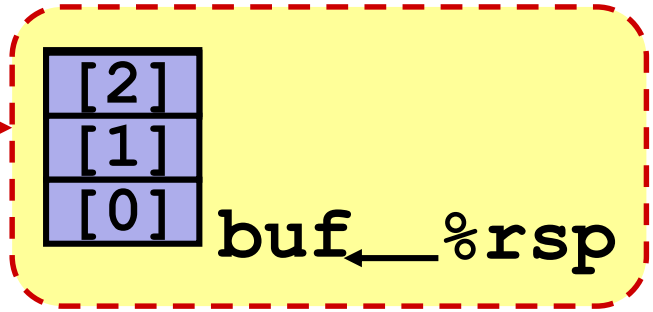
缓冲区溢出的栈示

```
/* Echo Line */
void echo(){
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

调用gets之前

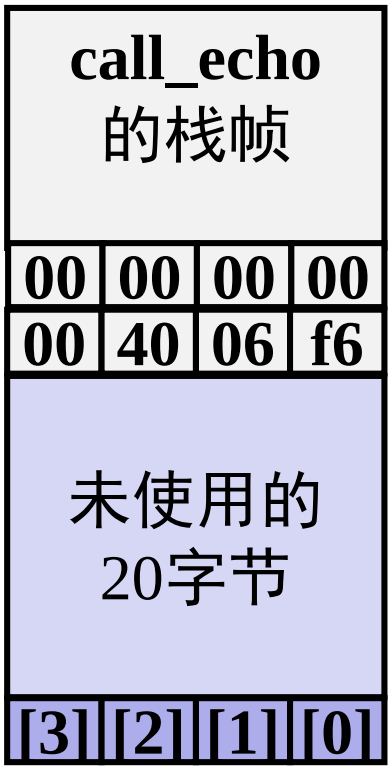



```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  ...
```



缓冲区溢出的栈示例

调用gets之前




```
void echo(){
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
    . . .
    4006f1: callq   4006cf <echo>
    4006f6: add     $0x8,%rsp
    . . .
```

缓冲区溢出的栈示例 #1

调用gets之后

call_echo 的栈帧

00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ⟵ %rsp

```
void echo(){
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

```
call_echo:
  . . .
  4006f1:    callq  4006cf <echo>
  4006f6:    add    $0x8,%rsp
  . . .
```

缓冲区溢出，但没有破坏状态

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901
0123456789012345678901
```

缓冲区溢出的栈示例 #2

调用gets之后

call_echo 的栈帧

00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo(){
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  ...
```

call_echo:

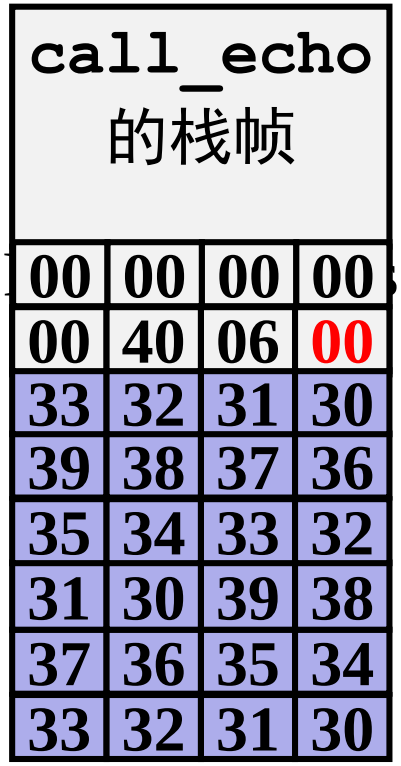
```
  ...
  4006f1:    callq  4006cf <echo>
  4006f6:    add    $0x8,%rsp
  ...
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

溢出的缓冲区,返回地址被破坏

缓冲区溢出的栈示例 #3

调用gets之后




```
void echo(){
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
  . . .
  4006f1:    callq  4006cf <echo>
  4006f6:    add    $0x8,%rsp
  . . .
```

溢出的缓冲区,破坏了返回地址，但程序看起来能工作

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
0123456789012345678901234
```

缓冲区溢出的栈示例 #3 ——解读

调用gets之后

call_echo 的栈帧			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ⟵ %rsp

register_tm_clones:

```
. . .
400600:   mov    %rsp,%rbp
400603:   mov    %rax,%rdx
400606:   shr    $0x3f,%rdx
40060a:   add    %rdx,%rax
40060d:   sar    %rax
400610:   jne    400614
400612:   pop    %rbp
400613:   retq
```

返回到无关的代码

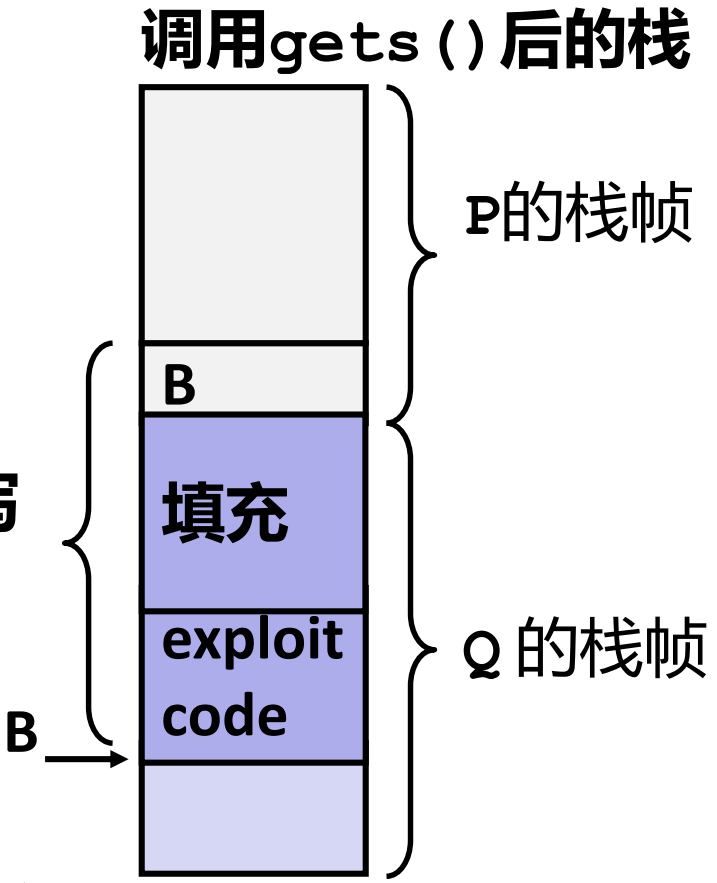
大多数情况不会修改临界状态，最终执行retq 返回主程序

代码注入攻击(Code Injection Attacks)

```
void P(){
  Q();
  ...
}
```

返回地址A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- 输入字符串包含可执行代码的字节序列!
- 将返回地址 A用缓冲区B的地址替换
- 当Q执行ret后，将跳转到B处，执行漏洞利用程序(exploit code)

基于缓冲区溢出的漏洞利用程序

- ***缓冲区溢出错误允许远程机器在受害者机器上执行任意代码。***
- **在程序中常见，令人不安**
  - 程序员持续犯相同的错误 ☹
  - 最近的措施使这些攻击更加困难。
- **经典案例**
  - 原始"互联网蠕虫"(Internet worm),1988
  - 即时通讯战争"IM wars",1999
  - Twilight hack on Wii, 2000s(不改动硬件，直接在Wii上运行自制程序)
- **在相应的实验中会学到一些技巧**
  - 希望能说服你永远不要在程序中留下这样的漏洞！！

例子: 原始互联网蠕虫 (1988)

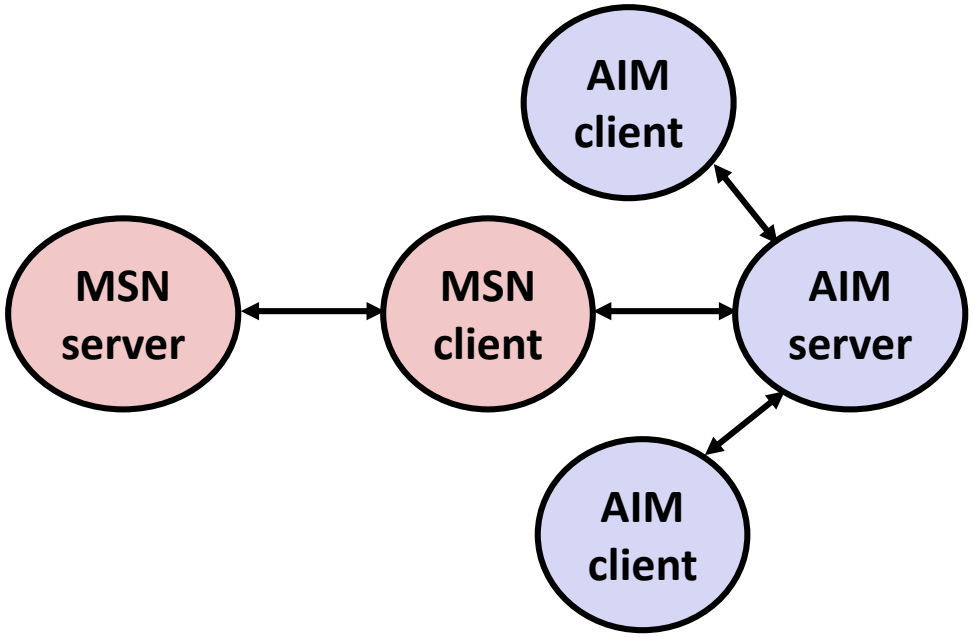
- **利用漏洞传播**
  - 指服务器(finger server)的早期版本用`gets()`读取客户机发来的参数:
    - **`finger droh@cs.cmu.edu`**
  - 蠕虫利用发送假参数的方法攻击指服务器:
    - **`finger "exploit-code padding new-return-address"`**
    - 利用程序:用直接和攻击者相连的TCP链接，在受害者机器上执行根用户shell

例子: 原始互联网蠕虫 (1988)

- 一旦进到机器上，就扫描其他机器攻击
- 几小时内侵入了大概6000台 (互联网机器总数的10%)
  - 参考*Comm. of the ACM*在1989年6月的文章
  - 年轻的蠕虫作者被起诉……
  - 计算机安全应急响应组(Computer Emergency Response Team)成立，…现仍在CMU

例2:即时通讯战争

- 1999年7月
  - 微软发布了即时通讯系统MSN Messenger
  - Messenger 的客户端能获取流行的美国在线(American Online， AOL)即时通讯服务（AIM）服务器



例2:即时通讯战争 (续...)

- 1999年8月
  - 很神秘, Messenger客户端无法再AIM服务器
  - 微软和AOL 开始了即时通讯战争
    - AOL 变动服务器不允许Messenger客户端连接
    - 微软对客户进行更改以挫败AOL的变动
    - 至少有13个这样的小冲突
  - 真正发生的到底是什么?
    - AOL 在他们自己的AIM服务器中发现了缓冲区溢出的漏洞
    - 他们利用这个bug检测并阻塞微软: 漏洞利用程序返回一个4字节的签名(在AIM客户端的某些位置存储的)到服务器
    - 当微软改变签名匹配程序时，AOL改变签名的位置

```
Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you
might find interesting because you are an Internet security expert with
experience in this area. I have also tried to contact AOL but received
no response.

I am a developer who has been working on a revolutionary new instant
messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself
this might not be the end of the world, as MS surely has had its share.
But AOL is now *exploiting their own buffer overrun bug* to help in
its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you
can use this information to help inform people that behind AOL's
friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com
```

后来确定这封电子邮件来源于微软内部！

旁白：蠕虫和病毒

- **蠕虫(Worm):程序**
  - 可以自行运行
  - 可以将自己的完整版本传播到其他计算机上
- **病毒(Virus): 代码**
  - 将自己添加到别的程序中
  - 不独立运行

- **两者通常都能在计算机之间传播并造成破坏。**

针对缓冲区溢出攻击，怎么做？

- 避免溢出漏洞
- 使用系统级的防护
- 编译器使用“栈金丝雀”(stack canaries)

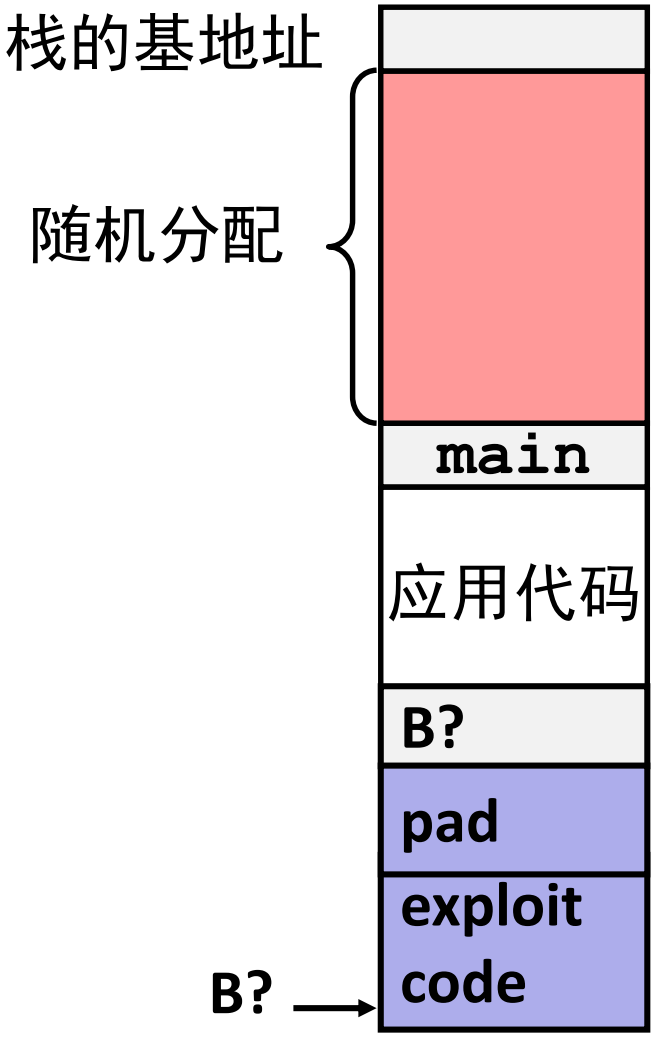
1. 代码中避免溢出漏洞(!)

```
/* Echo Line */
void echo(){
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- 例如，使用限制字符串长度的库例程
  - fgets 代替gets
  - strncpy 代替strcpy
  - 在scanf函数中别用%s
    - 用fgets读入字符串
    - 或用 %ns代替%s，其中n是一个合适的整数

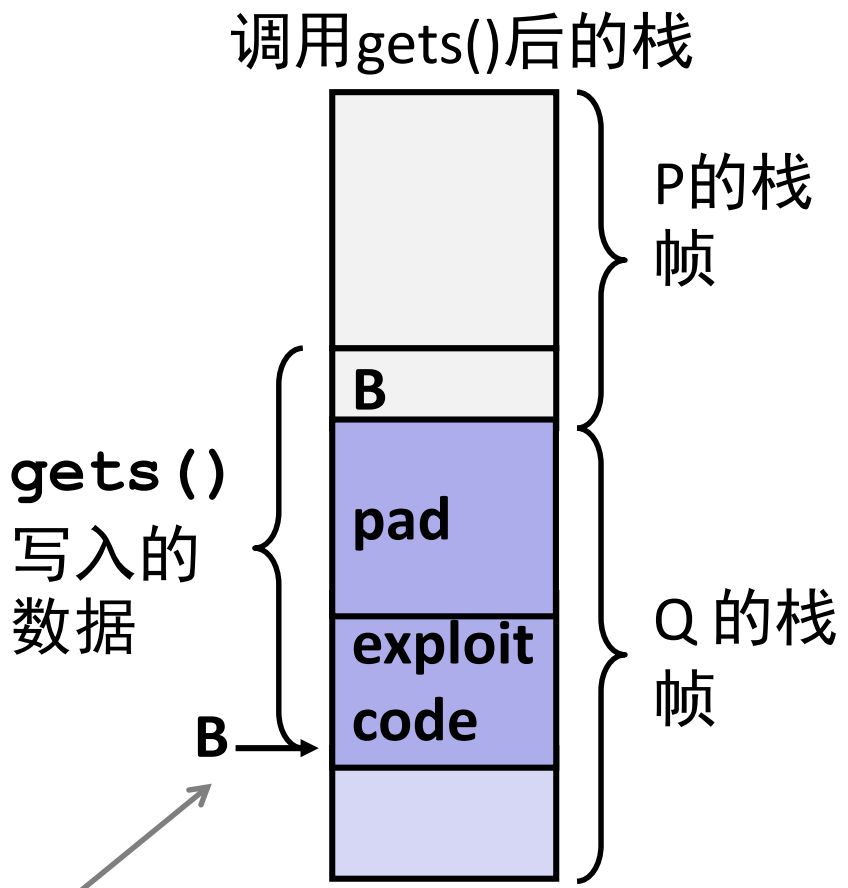
2. 系统级防护

- **随机的栈偏移**
  - 程序启动后，在栈中分配随机数量的空间
  - 将移动整个程序使用的栈空间地址
  - 黑客很难预测插入代码的起始地址
  - 例如：执行5次内存申请代码
    - 每次程序执行，栈都重新定位



2. 系统级防护

- 非可执行代码段
  - 在传统的x86中，可以标记存储区为“只读”或“可写的”
    - 可以执行任何可读的操作
  - x86-64添加显式“执行”权限
  - 将stack标记为不可执行



所有执行该代码的尝试都将失败

3. 栈金丝雀(Stack Canaries)

- 想法
  - 在栈中buffer之后的位置放置特殊的值——金丝雀(“canary”)
  - 退出函数之前，检查是否被破坏
- 用GCC 实现
  - `-fstack-protector`
  - 该选项现在是默认开启的(早期默认关闭)

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

保护缓冲区反汇编

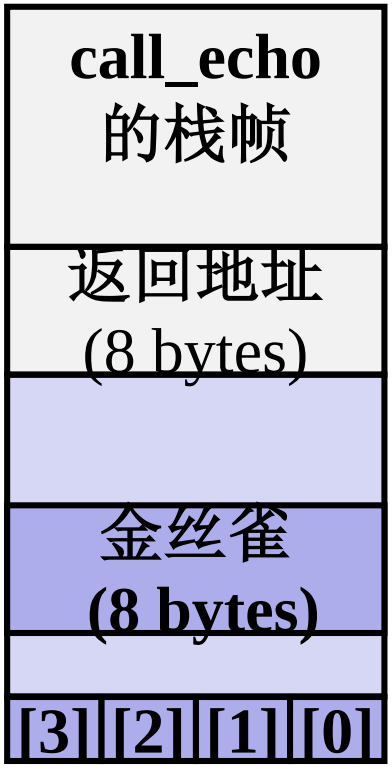
echo:

```
40072f:    sub    $0x18,%rsp
400733:    mov    %fs:0x28,%rax
40073c:    mov    %rax,0x8(%rsp)
400741:    xor    %eax,%eax
400743:    mov    %rsp,%rdi
400746:    callq  4006e0 <gets>
40074b:    mov    %rsp,%rdi
40074e:    callq  400570 <puts@plt>
400753:    mov    0x8(%rsp),%rax
400758:    xor    %fs:0x28,%rax
400761:    je     400768 <echo+0x39>
400763:    callq  400580 <__stack_chk_fail@plt>
400768:    add    $0x18,%rsp
40076c:    retq
```

设立金丝雀(Canary)

```
/* Echo Line */
void echo(){
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

调用gets之前

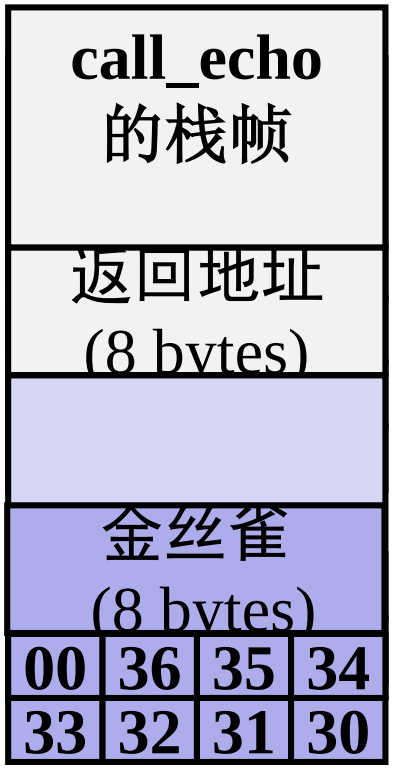



```
echo:
    . . .
    movq    %fs:40, %rax  # Get canary
    movq    %rax, 8(%rsp) # Place on stack
    xorl    %eax, %eax    # Erase canary
    . . .
```

核对金丝雀

```
/* Echo Line */
void echo(){
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

调用gets后




```
echo:
      …
      movq    8(%rsp), %rax    # Retrieve from stack
      xorq    %fs:40, %rax     # Compare to canary
      je      .L6              # If same, OK
      call    __stack_chk_fail # FAIL
.L6:  …
```

Input: *0123456*

面向返回的编程攻击（ROP，return oriented programming）

- **挑战(对黑客)**
  - 栈随机化使缓冲区位置难以预测。
  - 标记栈为不可执行，很难插入二进制代码
- **替代策略**
  - 使用已有代码
    - 例如：stdlib的库代码
  - 将片段串在一起以获得总体期望的结果。
  - 不用克服栈金丝雀
- **从小工具构建攻击程序**
  - 以`ret`结尾的指令序列
    - 单字节编码为`0xc3`
  - 每次运行，代码的位置固定
  - 代码可执行

小工具例子 #1

```
long ab_plus_c(long a, long b, long c)
{
  return a*b + c;
}
```

```
0000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe          imul %rsi,%rdi
  4004d4:  48 8d 04 17          lea (%rdi,%rdx,1),%rax
  4004d8:  c3                   retq
```

rax ← rdi + rdx

小工具地址 = 0x4004d4

- 使用现有功能的尾部

小工具例子 #2

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```

`movq %rax, %rdi`的编码

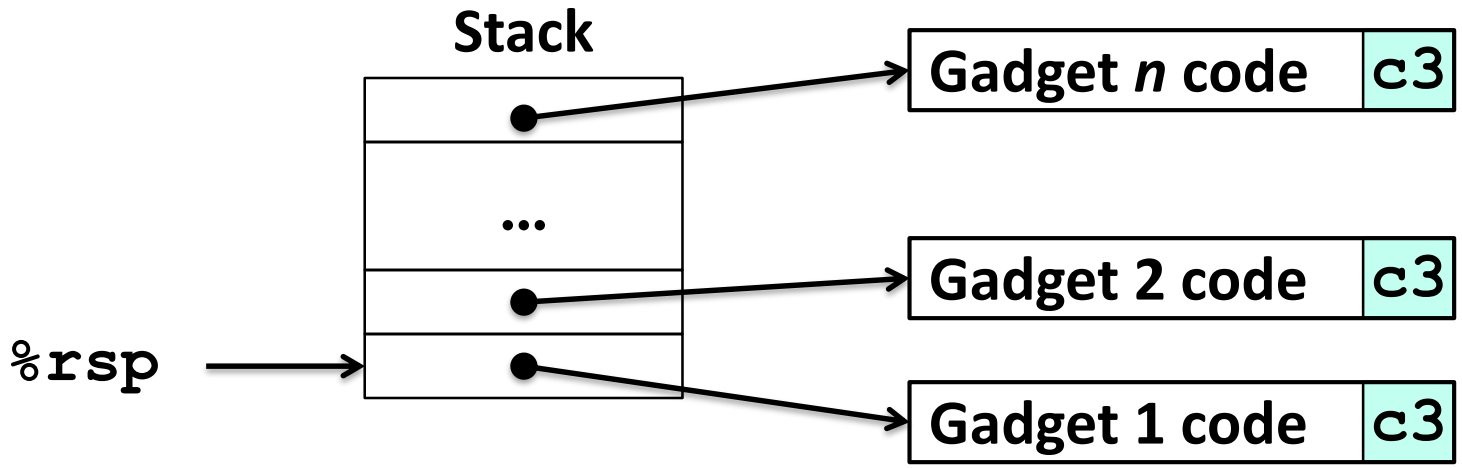
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl  $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

rdi ← rax

小工具地址 = 0x4004dc

- 改变字节码的用途

面向返回编程(ROP)的执行



- `ret` 指令触发
  - 将开始运行 Gadget 1
- 每个小工具最终的 `ret`将启动下一个小工具
- 通过小工具序列的运行，达到攻击目的。

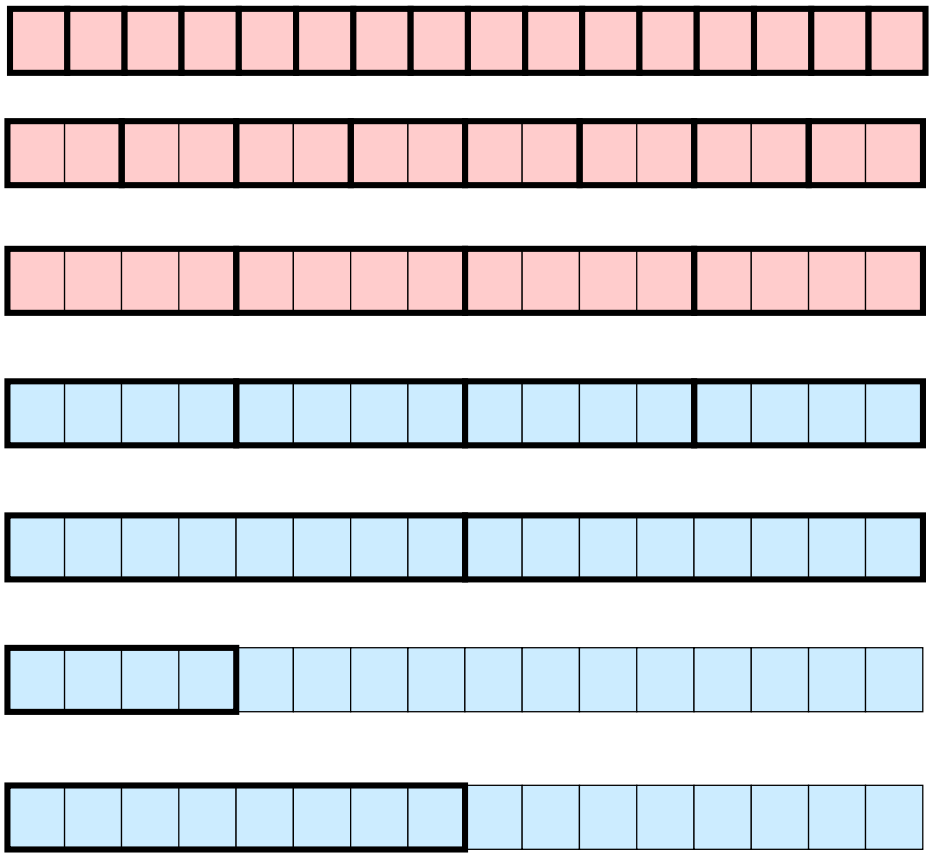
主要内容

- 内存布局
- 缓冲区溢出
  - 安全隐患
  - 防护
- **浮点数**

用SSE3编程（流式SIMD扩展）

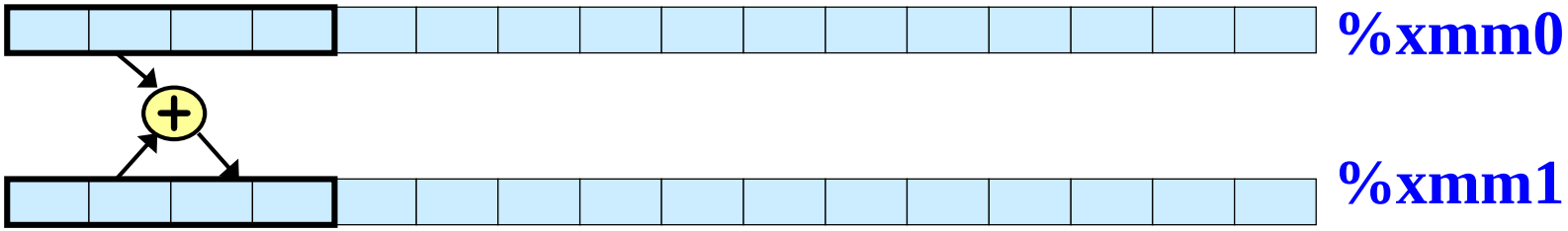
XMM 寄存器

- 共16 个 16字节的寄存器
  - 16个单字节整数
  - 8个16位整数
  - 4个32位整数
  - 4个单精度浮点数
  - 2个双精度浮点数
  - 1个单精度浮点数
  - 1个双精度浮点数

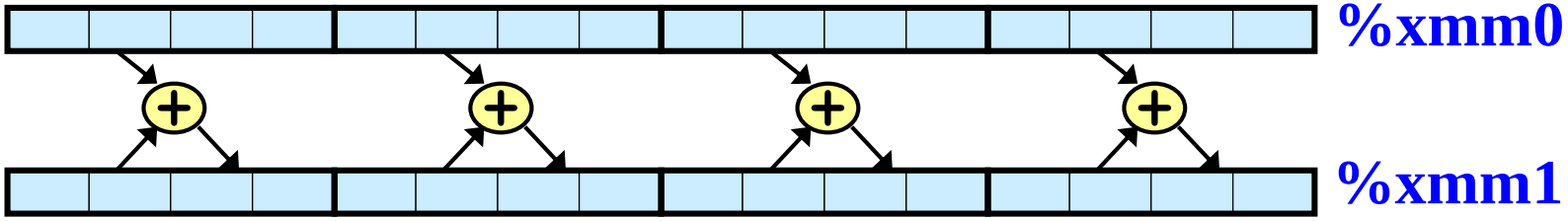


标量和SIMD操作

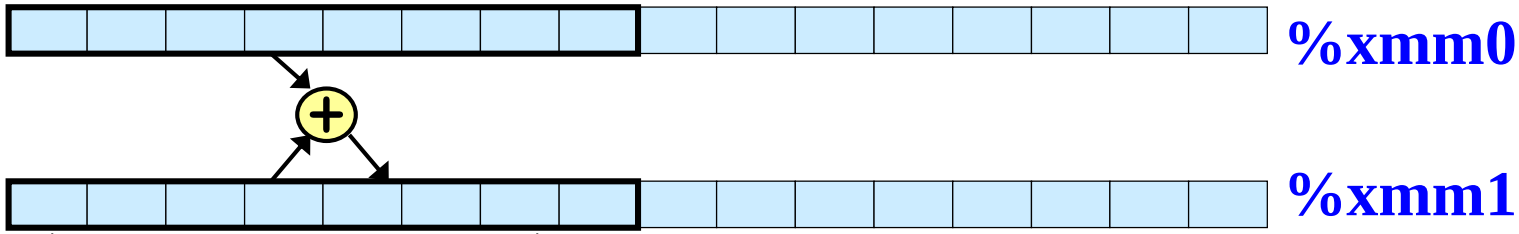
- 标量操作:单精度 addss %xmm0,%xmm1



- SIMD 操作: 单精度 addps %xmm0,%xmm1



- 标量操作:双精度 addsd %xmm0,%xmm1



浮点基础

- 参数传递使用：`%xmm0, %xmm1, ...`
- 返回结果保存：`%xmm0`
- 所有XMM 寄存器都是调用者保存

```
float fadd(float x, float y)
{
   return x + y;
}
```

```
double dadd(double x, double y)
{
   return x + y;
}
```

```
# x in %xmm0, y in %xmm1
addss   %xmm1, %xmm0
ret
```

```
# x in %xmm0, y in %xmm1
addsd   %xmm1, %xmm0
ret
```

浮点数的内存引用

- 单数传递：整数型 (包括指针) 参数用通用寄存器
- 单数传递：浮点型参数用XMM 寄存器
- 使用不同的mov指令在XMM 寄存器之间、或者内存和 XMM 寄存器之间传送数值

```
double dincr(double *p, double v){
    double x = *p;
    *p = x + v;
    return x;
}
```

```
# p in %rdi, v in %xmm0
movapd  %xmm0, %xmm1   # Copy v
movsd   (%rdi), %xmm0  # x = *p
addsd   %xmm0, %xmm1   # t = x + v
movsd   %xmm1, (%rdi)  # *p = t
ret
```

浮点数编程

- **指令多**
  - 不同的操作、格式...
- **浮点数比较**
  - **`ucomiss` 和`ucomisd`（单精度数比较和双精度数比较）**
  - 设置条件码: CF, ZF和PF
- **常量数值的使用**
  - 寄存器XMM0 清零：
    `xorpd %xmm0, %xmm0`
  - 其他：从内存载入

附：课外资料-----典型的指令系统

- **CISC指令系统具有如下特点：**
  - 指令系统庞大，表现为指令条数较多
  - 指令格式多样，表现为指令字长不固定，寻址方式种类较多
  - 指令的执行时间差别较大
  - 大多数CISX指令系统采用微程序控制器
- **RISC指令系统特点：**
  - 指令条数少，所选取的指令都是使用频率较高的简单指令
  - 指令字长固定，且指令格式种类少
  - 只有取数/存数指令可以访问存储器，其他指令的操作都在寄存器之间进行

ARM处理器特点

- 特点
  - 功耗低、成本低、性能高
  - 支持Thumb（16位）/ARM（32位）双指令集
    - Cortex支持Thumb-2（16/32位混合指令系统）
  - 指令长度固定（32位/16位）
  - 大量使用寄存器，指令执行速度更快
  - 寻址方式灵活简单，执行效率高
  - Load-Store体系结构
  - 3地址指令（两个源操作数寄存器和结果寄存器独立设定）
  - 能在单时钟周期执行的单条指令内完成一项普通的移位操作和一项普通的ALU操作
  - 能过协处理器指令集来扩展ARM指令集，包括在编程模式下增加了新的寄存器和数据类型

- **ARM微处理器的指令集是Load/Store型**
  - 注意：可以使用间接寻址等方式访问存储器，注意和纯RISC处理器的区别
- **1 指令格式**
- **2 指令助记符**
- **3 指令条件域**

ARM指令格式

- **<opcode>{<cond>}{S} <Rd>, <Rn>{, operand2}**
  - opcode: 指令助记符，如ADD, LDR, STR
  - *cond:* *执行条件，如NE, EQ*
  - *S:* *是否影响CPSR的值*
  - Rd: 目标寄存器
  - Rn: 第一操作数寄存器
  - *operand2:* *第二操作数*
    - *灵活使用第二操作数能够提高代码质量*
  - 例子：ADDEQS R0, R1, #0x3f

- 实例
  - C代码： if (a>b) a++; else b++;
  - Thumb代码：
```
        CMP     R0, R1
        BHI     A_ADD
        ADD     R1, R1, #1
        B       OVER
A_ADD   ADD     R0, R0, #1
OVER    ……
```
  - ARM代码：
```
        CMP     R0, R1
        ADDHI   R0, R0, #1
        ADDLS   R1, R1, #1
```

- **ARM处理器的寄存器**
  - 37个32位寄存器
    - 31个通用寄存器
    - 6个状态寄存器
  - 这些寄存器不能被同时访问
    - 取决于处理器的
      - 工作状态
      - 工作模式

ARM状态下的寄存器组织

工作模式：	System&User	FIQ	Supervisor	Abort	IRQ	Undefined
未分组寄存器	R0	R0	R0	R0	R0	R0
	R1	R1	R1	R1	R1	R1
	R2	R2	R2	R2	R2	R2
	R3	R3	R3	R3	R3	R3
	R4	R4	R4	R4	R4	R4
	R5	R5	R5	R5	R5	R5
	R6	R6	R6	R6	R6	R6
	R7	R7	R7	R7	R7	R7
分组寄存器	R8	***R8_fiq***	R8	R8	R8	R8
	R9	***R9_fiq***	R9	R9	R9	R9
	R10	***R10_fiq***	R10	R10	R10	R10
	R11	***R11_fiq***	R11	R11	R11	R11
	R12	***R12_fiq***	R12	R12	R12	R12
堆栈指针	R13_usr	***R13_fiq***	***R13_svc***	***R13_abt***	***R13_irq***	***R13_und***
链接寄存器	R14_usr	***R14_fiq***	***R14_svc***	***R14_abt***	***R14_irq***	***R14_und***
程序计数器	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)
当前程序状态寄存器	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
备份程序状态寄存器		***SPSR_fiq***	***SPSR_svc***	***SPSR_abt***	***SPSR_irq***	***SPSR_und***

MIPS

- **MIPS处理器**
  - MIPS计算机公司研发
  - 一种高端嵌入式内核标准
  - MIPS的英文原文：
    - Microprocessor without Interlocked Pipeline Stages
  - 中文意义是内部无互锁流水级微处理器

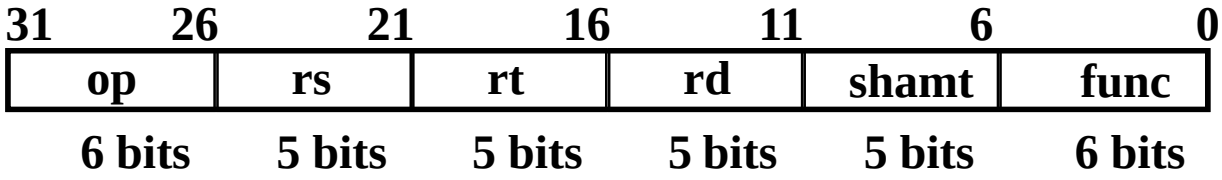
MIPS指令格式

- 所有指令都是32位宽，须按字地址对齐

  字地址为4的倍数！

- 有三种指令格式

R-Type指令



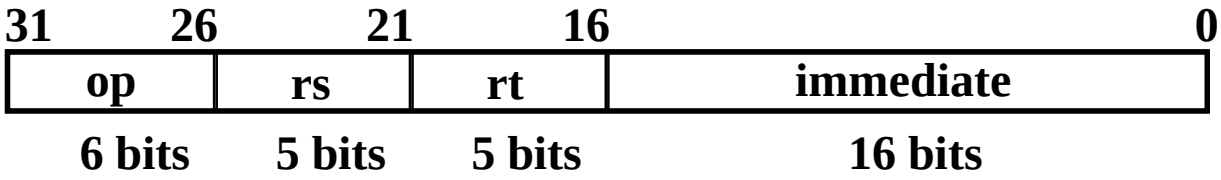
– R-Type

两个操作数和结果都在寄存器的运算指令。如：sub rd, rs, rt

– I-Type

- 运算指令：一个寄存器、一个立即数。如：ori rt, rs, imm16
- LOAD和STORE指令。如：lw rt, rs, imm16
- 条件分支指令。如：beq rs, rt, imm16

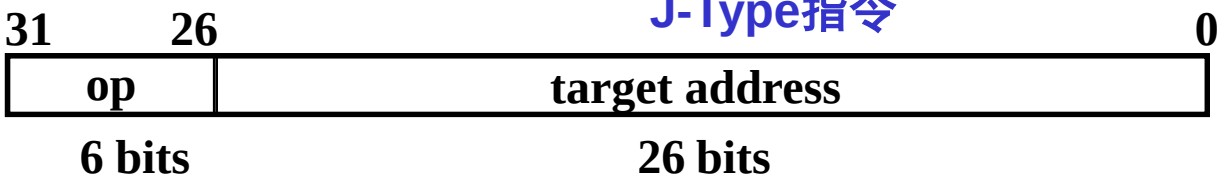
I-Type指令



– J-Type

无条件跳转指令。如：j target

J-Type指令



MIPS指令字段含义

OP：操作码

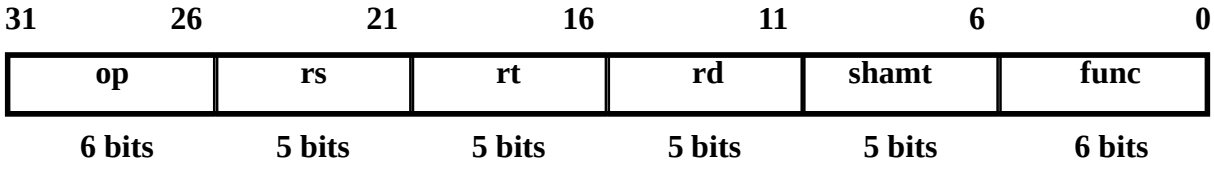
rs：第一个源操作数寄存器

rt：第二个源操作数寄存器

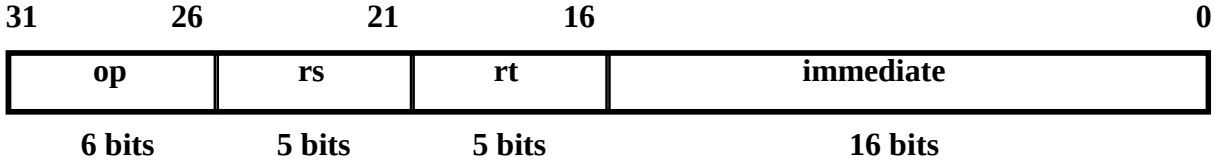
rd：结果寄存器

shamt：移位指令的位移量

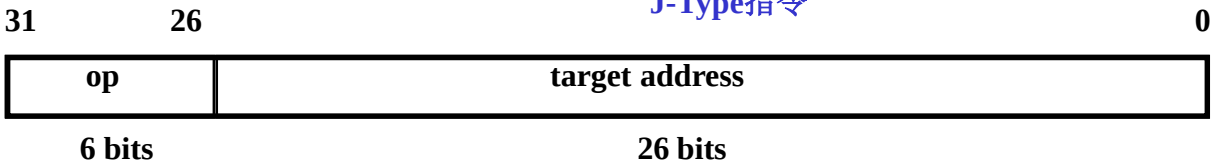
R-Type指令



I-Type指令



J-Type指令



func：R-Type指令的OP字段是特定的“000000”，具体操作由func字段给定。例如：func=“100000”时，表示“加法”运算。

操作码的不同编码定义不同的含义，操作码相同时，再由功能码定义不同的含义!

immediate：立即数或load/store指令和分支指令的偏移地址

target address：无条件转移地址的低26位。将PC高4位拼上26位直接地址，最后添2个“0”就是32位目标地址。为何最后两位要添“0”？

指令按字地址对齐，所以每条指令的地址都是4的倍数（最后两位为0）。

MIPS的算术运算指令

需要判溢出，溢出时发生“异常”

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
multiply	mult $2,$3	Hi, Lo = $2×$3	64-bit signed product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
Move from Hi	mfhi $1	$1=Hi	get a copy of Hi
Move from Lo	mflo $1	$1=lo	