x86-64指令系统概述

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

回顾：Intel处理器

x86前产品	4004 • 4040 • 8008 • 8080 • iAPX 432 • 8085	
x87（外置浮点运算器）	8/16位总线: 8087 16位总线: 80187 • 80287 • 80387SX 32位总线: 80387DX • 80487	已停产
x86-16（16位）	8086 • 8088 • 80186 • 80188 • 80286	
x86-32/IA-32（32位）	80386 • 80486 • Pentium（OverDrive、Pro、II、III、4、M） • Celeron（M、D） • Core	
x86-64/Intel 64（64位）	Pentium（4（部份型号）、Pentium D、EE） • Celeron D（部份型号） • Core 2	
EPIC/IA-64（64位）	Itanium	
RISC	i860 • i960 • StrongARM • XScale	
微控制器	8048 • 8051 • MCS-96	
x86-32/IA-32	EP80579 • A100 • Atom（CE、SoC）	现有产品
x86-64/Intel 64	Xeon（E3、E5、E7、Phi） • Atom（部分型号） • Celeron • Pentium • Core（i3、i5、i7）	
EPIC/IA-64	Itanium 2	

x86-64架构

- 背景
  - 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，Intel 称其为显式并行指令计算机EPIC（Explicitly Parallel Instruction Computer）。
  - 安腾和安腾2分别在2000年和2002年问世，它们是IA-64体系结构的最早的具体实现，因为是一种全新的、与IA-32不兼容的架构，所以，没有获得预期的市场份额。
  - AMD公司利用Intel在IA-64架构上的失败，抢先在2003年推出兼容IA-32的64位版本指令集x86-64，AMD获得了以前属于Intel的一些高端市场。AMD后来将x86-64更名为AMD64。
  - 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Intel64。

回顾：IA-32支持的数据类型及格式

C 语言声明	Intel 操作数类型	汇编指令长度后缀	存储长度（位）
(unsigned) char	整数 / 字节	b	8
(unsigned) short	整数 / 字	w	16
(unsigned) int	整数 / 双字	l	32
(unsigned) long int	整数 / 双字	l	32
unsigned) long long int	–	–	2×32
char *	整数 / 双字	l	32
float	单精度浮点数	s	32
double	双精度浮点数	l	64
long double	扩展精度浮点数	t	80 / 96

IA-32架构由16位架构发展而来，因此，虽然字长为32位或更大，但一个字为16位，长度后缀为 w；32位为双字，长度后缀为 l

long double实际长度为80位，但分配96位=12B（按4B对齐）

x86-64中各类数据的长度

C 语言声明	Intel 操作数类型	汇编后缀	x86-64（位）	IA-32（位）
(unsigned) char	整数 / 字节	b	8	8
(unsigned) short	整数 / 字	w	16	16
(unsigned) int	整数 / 双字	l	32	32
(unsigned) long int	整数 / 四字	q	64	32
unsigned) long long int	整数 / 四字	q	64	2×32
char *	整数 / 四字	q	64	32
float	单精度浮点数	s	32	32
double	双精度浮点数	d	64	64
long double	扩展精度浮点数	t	80 / 128	80 / 96

x86-64的通用寄存器

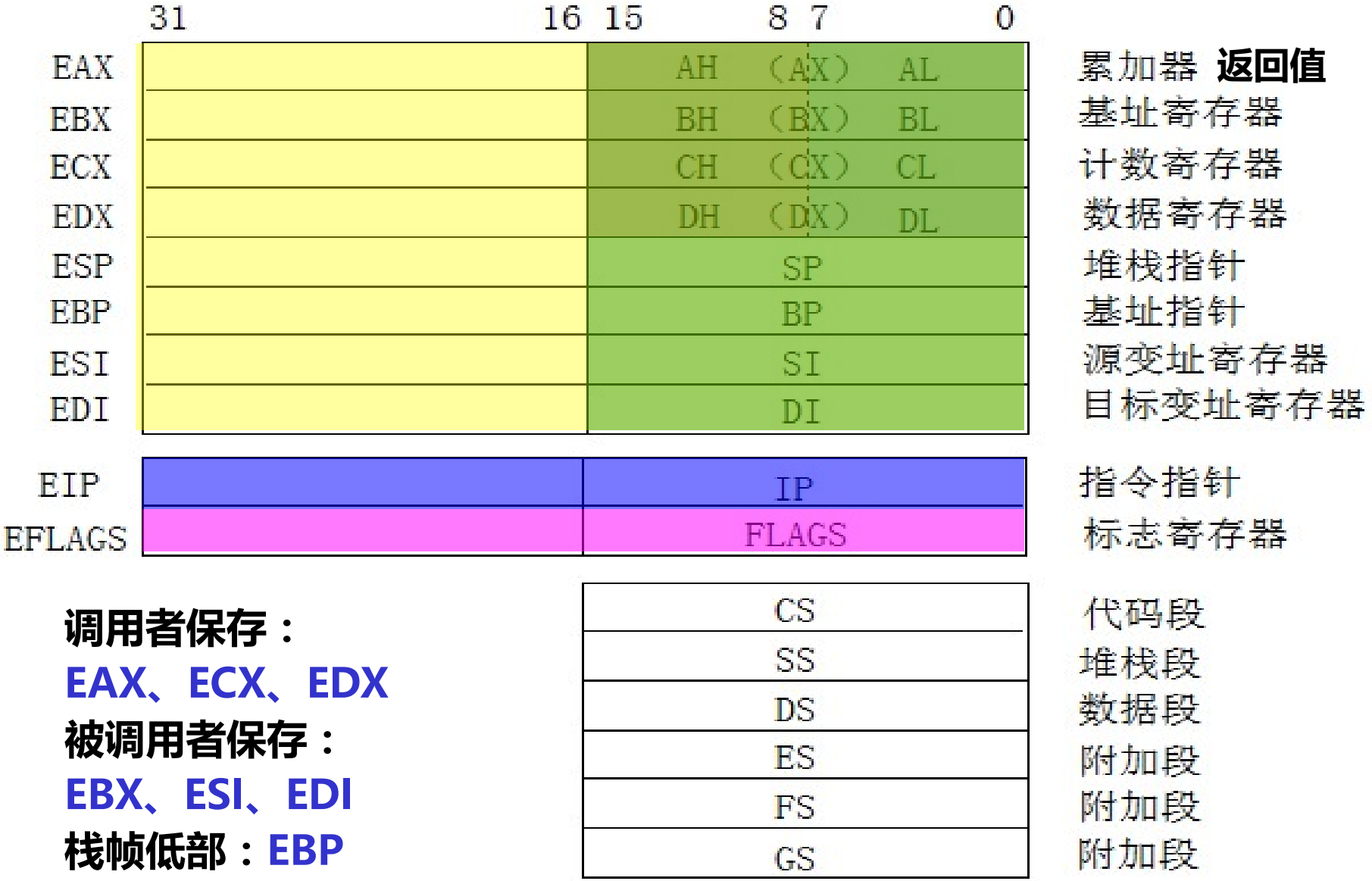
- 新增8个64位通用寄存器（整数寄存器）
  - R8、R9、R10、R11、R12、R13、R14和R15。
  - 可作为8位（R8B~R15B）、16位（R8W~R15W）或32位寄存器（R8D~R15D）使用
- 所有GPRs都从32位扩充到64位
  - 8个32位通用寄存器EAX、EBX、ECX、EDX、EBP、ESP、ESI和 EDI对应扩展寄存器分别为RAX、RBX、RCX、RDX、RBP、RSP、RSI和RDI
  - EBP、ESP、ESI和 EDI的低8位寄存器分别是BPL、SPL、SIL和DIL
  - 可兼容使用原AH、BH、CH和DH寄存器

（使原来IA-32中的每个通用寄存器都可以是8位、16位、32位和64位，如：SIL、SI、ESI、RSI）

x86-64中寄存器的使用

- 指令可直接访问16个64位寄存器：RAX、RBX、RCX、RDX、RBP、RSP、RSI、RDI，以及R8~R15
- 指令可直接访问16个32位寄存器：EAX、EBX、ECX、EDX、EBP、ESP、ESI、EDI，以及R8D~R15D
- 指令可直接访问16个16位寄存器：AX、BX、CX、DX、BP、SP、SI、DI，以及R8W~R15W
- 指令可直接访问16个8位寄存器：AL、BL、CL、DL、BPL、SPL、SIL、DIL，以及R8B~R15B
- 为向后兼容，指令也可直接访问AH、BH、CH、DH
- 通过寄存器传送参数，因而很多过程不用访问栈，因此，与IA-32不同，x86-64不需要帧指针寄存器，即RBP可用作普通寄存器使用
- 程序计数器为64位寄存器RIP

回顾：IA-32的寄存器组织



调用者保存：

EAX、ECX、EDX

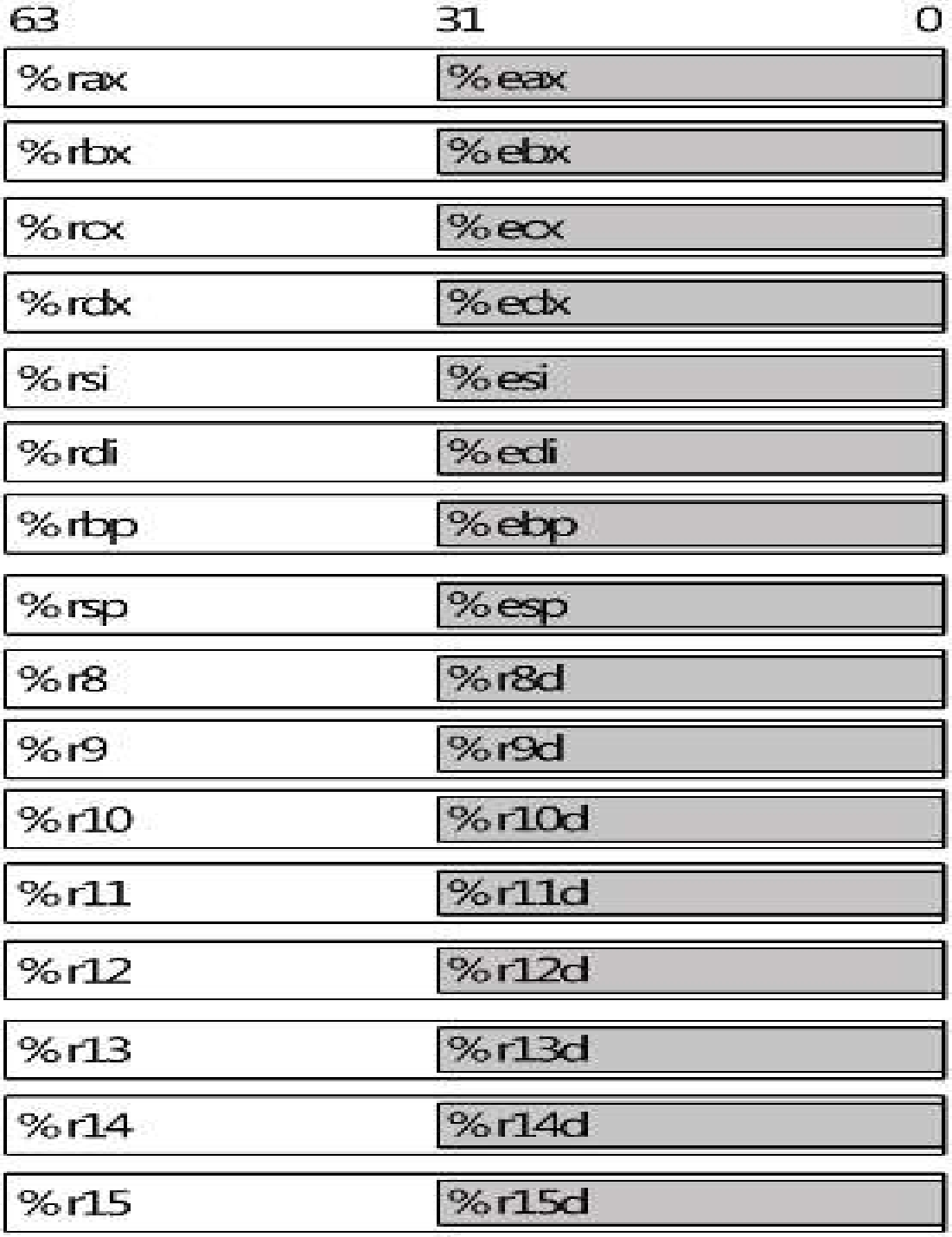
被调用者保存：

EBX、ESI、EDI

栈帧低部：EBP

栈帧顶部：ESP

寄存器



通用寄存器个数从8个增加到16个，宽度从32位增加到64位

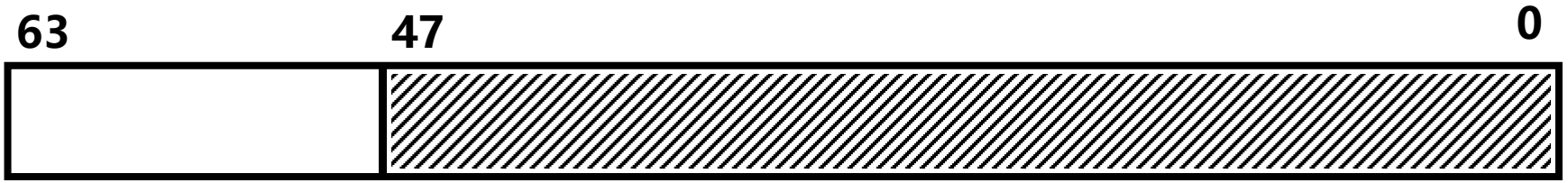
增加了 %sil、%dil、%bpl、%spl 四个8位寄存器

%riw为16位

%rib为8位（i=8~15）

x86-64的地址和寻址空间

- 字长从32位变为64位，64位（8B）数据被称为一个四字（qw: quadword）
- 逻辑地址最长可达为64位，即理论上可访问的存储空间达2^{64}字节或16EB（ExaByte）
- 编译器为指针变量分配64位（8B）
- 基址寄存器和变址寄存器都应使用64位寄存器
- 但实际上，AMD和Intel的x86-64仅支持48位虚拟地址，因此，程序的虚拟地址空间大小为2^{48}=256TB



x86-64的浮点寄存器

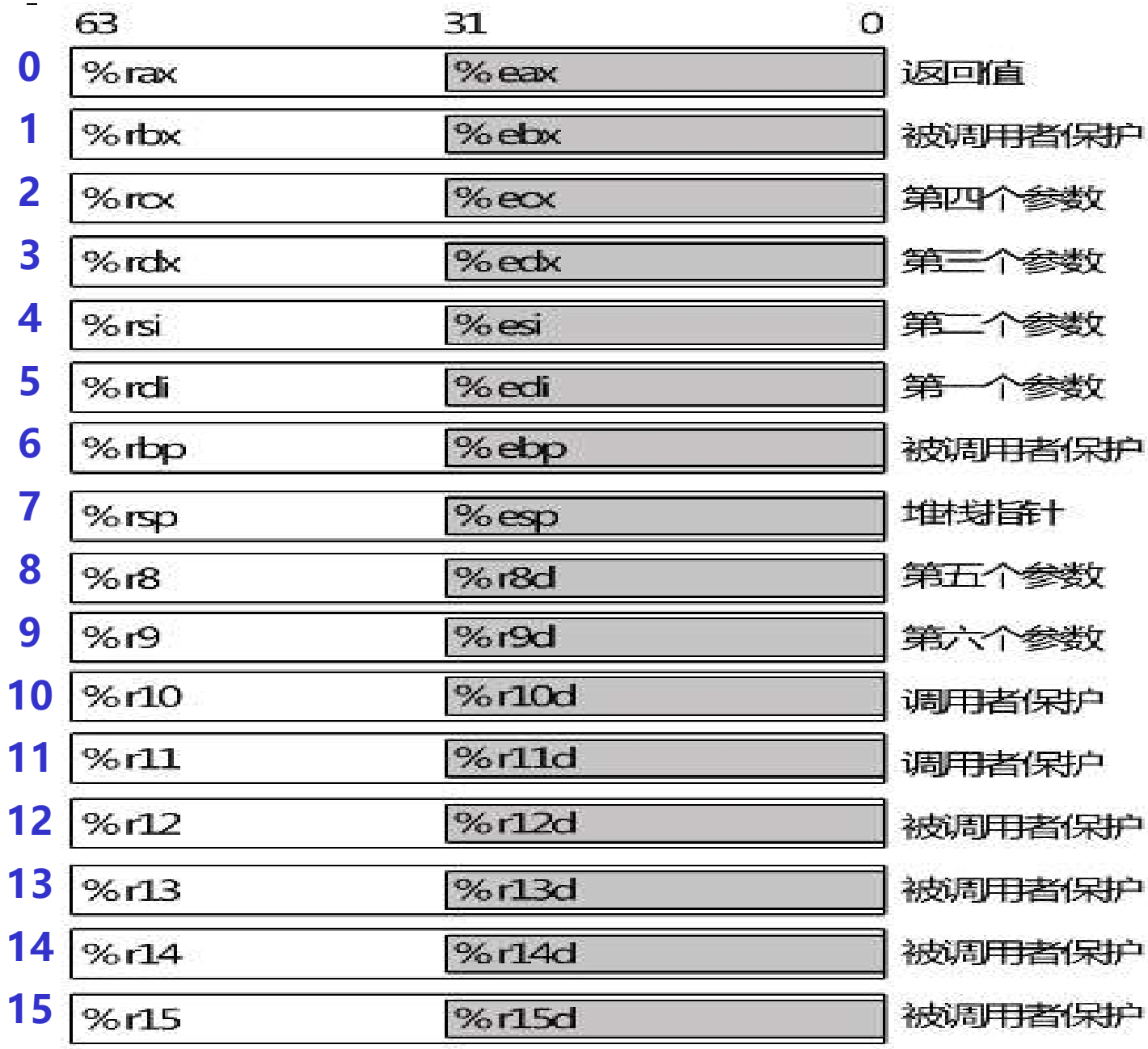
- long double型数据虽然还采用80位（10B）扩展精度格式，但所分配存储空间从12B扩展为16B，即改为16B对齐方式，但不管是分配12B还是16B，都只用到低10B
- 128位的XMM寄存器从原来的8个增加到16个
- 浮点操作指令集采用基于SSE的面向XMM寄存器的指令集，而不采用基于浮点寄存器栈的 x87 FPU 指令集
- 浮点操作数存放在XMM寄存器中

回顾：IA-32的寄存器组织

编号	8 位寄存器	16 位寄存器	32 位寄存器	64 位寄存器	128 位寄存器
000	AL	AX	EAX	MM0 / ST(0)	XMM0
001	CL	CX	ECX	MM1 / ST(1)	XMM1
010	DL	DX	EDX	MM2 / ST(2)	XMM2
011	BL	BX	EBX	MM3 / ST(3)	XMM3
100	AH	SP	ESP	MM4 / ST(4)	XMM4
101	CH	BP	EBP	MM5 / ST(5)	XMM5
110	DH	SI	ESI	MM6 / ST(6)	XMM6
111	BH	DI	EDI	MM7 / ST(7)	XMM7

x86-64继承了IA-32中的8、16、32位通用寄存器和128位XMM寄存器
而取消了IA-32中的80位浮点寄存器栈ST(0)-ST(7)

x86-64的寄存器



增加了 %sil、%dil、%bpl、%spl 四个8位寄存器

%riw为16位
%rib为8位
（i=8~15）

16个128位寄存器%xmmi
（i=0~15）

x86-64中数据的对齐

- 各类型数据遵循一定的对齐规则，而且更严格
- 存储器访问接口被设计成按8字节或16字节为单位进行存取，其对齐规则是，任何K字节宽的基本数据类型和指针类型数据的起始地址一定是K的倍数。
  - short型数据必须按2字节边界对齐
  - int、float等类型数据必须按4字节边界对齐
  - long、double、指针型变量必须按8字节边界对齐
  - long double型数据必须按16字节边界对齐

x86-64的基本指令

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

传送指令

- 数据传送指令（助记符“q”表示操作数长度为四字（即64位））

  movabsq I, R：将64位立即数送64位通用寄存器

  movq：传送一个64位的四字

  movsbq、movswq、movslq：将源操作数进行符号扩展并传送到一个64位寄存器或存储单元中

  movzbq、movzwq：将源操作数进行零扩展后传送到一个64位寄存器或存储单元中

  movl：的功能相当于movzlq指令

  pushq S：R[rsp]←R[rsp]-8; M[R[rsp]] ←S

  popq D： D← M[R[rsp]]; R[rsp]←R[rsp]-8

传送指令

- **数据传送指令举例**

以下函数功能是将类型为source_type的参数转换为dest_type型数据并返回

```
dest_type  convert(source_type x) {
        dest_type y = (dest_type) x;
        return y;
}
```

根据参数传递约定知，x在RDI对应的适合宽度的寄存器（RDI、EDI、DI和DIL）中，y存放在RAX对应的寄存器（RAX、EAX、AX或AL）中，填写下表中的汇编指令以实现convert函数中的赋值语句

source_type	dest_type	汇 编 指 令
char	long	问题：每种情况对应的汇编指令各是什么？
int	long	
long	long	
long	int	
unsigned int	unsigned long	
unsigned long	unsigned int	
unsigned char	unsigned long	

传送指令

- **数据传送指令举例**

以下函数功能是将类型为source_type的参数转换为dest_type型数据并返回

```
dest_type  convert(source_type x) {
        dest_type y = (dest_type) x;
        return y;
}
```

根据参数传递约定知，x在RDI对应的适合宽度的寄存器（RDI、EDI、DI和DIL）中，y存放在RAX对应的寄存器（RAX、EAX、AX或AL）中，填写下表中的汇编指令以实现convert函数中的赋值语句

source_type	dest_type	汇 编 指 令
char	long	movsbq %dil, %rax
int	long	movslq %edi, %rax
long	long	movq %rdi, %rax
long	int	movslq %edi, %rax //符号扩展到 64 位 movl %edi, %eax //只需x的低32位
unsigned int	unsigned long	movl %edi, %eax //零扩展到 64 位
unsigned long	unsigned int	movl %edi, %eax //零扩展到 64 位
unsigned char	unsigned long	movzbq %dil, %rax //零扩展到 64 位

算术逻辑指令

- **常规的算术逻辑运算指令**

  只要将原来IA-32中的指令扩展到64位即可。例如：

  - **addq（四字相加）**
  - **subq（四字相减）**
  - **incq（四字加1）**
  - **decq（四字减1）**
  - **imulq（带符号整数四字相乘）**
  - **orq（64位相或）**
  - **salq（64位算术左移）**
  - **leaq（有效地址加载到64位寄存器）**

算术逻辑指令

以下是C赋值语句“x=a*b+c*d;”对应的x86-64汇编代码

已知x、a、b、c和d分别在寄存器RAX(x)、RDI(a)、RSI(b)、RDX(c)和RCX(d)对应宽度的寄存器中

根据以下汇编代码，推测x、a、b、c和d的数据类型

```
movslq %ecx, %rcx
imulq  %rdx, %rcx
movsbl %sil, %esi
imull %edi, %esi
movslq %esi, %rsi
leaq (%rcx, %rsi), %rax
```

d从32位符号扩展为64位，故d为int型

在RDX中的c为64位long型

在SIL中的b为char型

在EDI中的a是int型

在RAX中的x是long型

算术逻辑指令

- **特殊的算术逻辑运算指令**

  对于x86-64，还有一些特殊的算术逻辑运算指令。例如：

  imulq S：R[rdx]:R[rax]← S * R[rax] (64位*64位带符号整数)

  mulq S：R[rdx]:R[rax]← S * R[rax] (64位*64位无符号整数)

  cltq：R[rax] ← SignExtend(R[eax]) (将EAX内容符号扩展为四字)

  clto： R[rdx]:R[rax]← SignExtend(R[rax]) (符号扩展为八字)

  idivq S：R[rdx] ← R[rdx]:R[rax] mod S (带符号整数相除、余数)

  R[rax] ← R[rdx]:R[rax]÷S (带符号整数相除、商)

  divq S：R[rdx] ← R[rdx]:R[rax] mod S (无符号整数相除、余数)

  R[rax] ← R[rdx]:R[rax]÷S (无符号整数相除、商)

  上述功能描述中，R[rdx]:R[rax]是一个128位的八字（oct word）

算术逻辑指令

不同长度操作数混合运算时，编译器必须选择正确的指令的组合。

```
long samp(int x, int y)
{
    long t1=(long) x+y;
    long t2=(long) (x+y);
    return t1 | t2;
}
```

计算t1：先符号扩展为64位，再进行64位加法

计算t2：先进行32位加法，再符号扩展为64位

对应x86-64汇编代码如下（x在EDI中，y在ESI中）：

```
leal  (%rdi,%rsi), %eax
cltq
movslq %esi, %rsi
movslq %edi, %rdi
addq %rsi, %rdi
orq  %rdi, %rax
```

EDI和ESI中内容相加，低32位送EAX

将EAX符号扩展为四字，送RAX（t2）

将ESI符号扩展为四字，送RSI

将EDI符号扩展为四字，送RDI

RDI和RSI中内容相加，送RDI（t1）

RAX和RDI中内容相或，送RAX

比较和测试指令

- **比较和测试指令**

  与IA-32中比较和测试指令类似。例如：

  cmpq S2, S1 : S1-S2 (64位数相减进行比较)

  testq S2, S1 : S1∧S2 (64位数相与进行比较)

  条件转移指令、条件传送指令、条件设置指令都根据上述比较指令和测试指令生成的标志进行处理

x86-64逆向工程举例

根据汇编代码填写C语句，说明功能

```
long test(unsigned long x)
{
    long val=0;
    int i;
    for ( ①  ; ②  ; ③ ){
        ④ ;
    }
    ⑤ ;
    return ⑥ ;
}
```

①处是: i=0

```
movl    $0, %ecx        R[ecx]=val
movl    $0, %edx        R[edx]=i
movabsq $7234017283807673, %rsi
.L1                     R[rsi]=0x0101010101010101
movq    %rdi, %rax
andq    %rsi, %rax
addq    %rax, %rcx
shrq    %rdi
addl    $1, %edx
cmpl    $8, %edx
jne     .L1
movq    %rcx, %rax
sarq    $32, %rax
addq    %rcx, %rax
movq    %rax, %rdx
sarq    $16, %rdx
addq    %rax, %rdx
movq    %rdx, %rax
sarq    $8, %rax
addq    %rdx, %rax
andl    $255, %eax
ret
```

④处是:
val+=x&0x01...01;
x>>=1;

③处是: i++

②处是: i<8

⑤处是以下语句：
val+=(val>>32);
高、低32位相加
val+=(val>>16);
高、低16位相加
val+=(val>>8);
高、低8位相加

⑥处是: val&0xFF;

GCC生成的x86-64汇编代码如右

$2^{56}+2^{48}+2^{40}+2^{32}+2^{24}+2^{16}+2^{8}+2^{0}$
=7234017283807673

for循环：val各字节记录x中对应字节内1的个数

函数功能：计算x中1的个数。

因最大值为64，故最终析取低8位

x86-64的过程调用

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

先看一个例子

对于以下C语言源文件sample.c：

```
long int sample(long int *xp, long int y)
{
    long int t=*xp+y;
    *xp=t;
    return t;
}
```

- 在x86-64/Linux平台上用以下命令执行汇编操作，得到x86-64汇编指令代码

$ gcc –O1 –S –m64 sample.c

```
sample：
   movq    %rsi, %rax
   addq    (%rdi), %rax
   movq    %rax, (%rdi)
   ret
```

在x86-64/Linux平台上默认生成x86-64格式代码，故可省略-m64

- 在x86-64/Linux平台上用以下命令执行汇编操作，得到与IA-32兼容的汇编指令代码

$ gcc –O1 –S –m32 sample.c

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

Long型数据长度不同

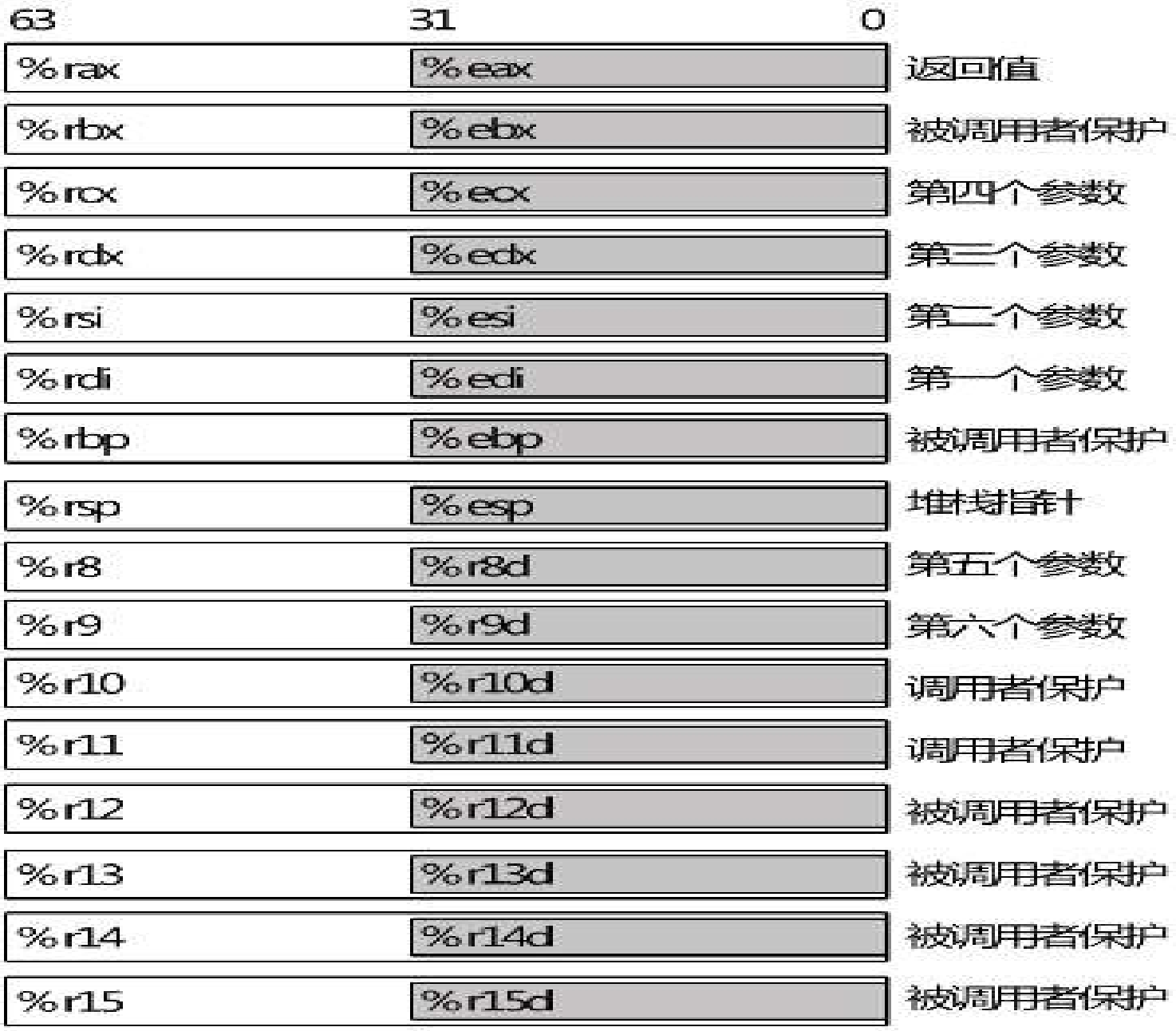
参数传递方式不同

x86-64过程调用的参数传递

- 通过通用寄存器传送参数，很多过程不用访问栈，故执行时间比IA-32代码更短
- 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
- call（或callq）将64位返址保存在栈中之前，执行R[rsp]←R[rsp]-8
- ret从栈中取出64位返回地址后，执行R[rsp]←R[rsp]+8

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过程调用的寄存器使用约定



在过程(函数)中尽量使用寄存器RAX、R10和R11。若使用RBX、RBP、R12、R13、R14和R15，则需要将它们先保存在栈中再使用，最后返回前再恢复其值

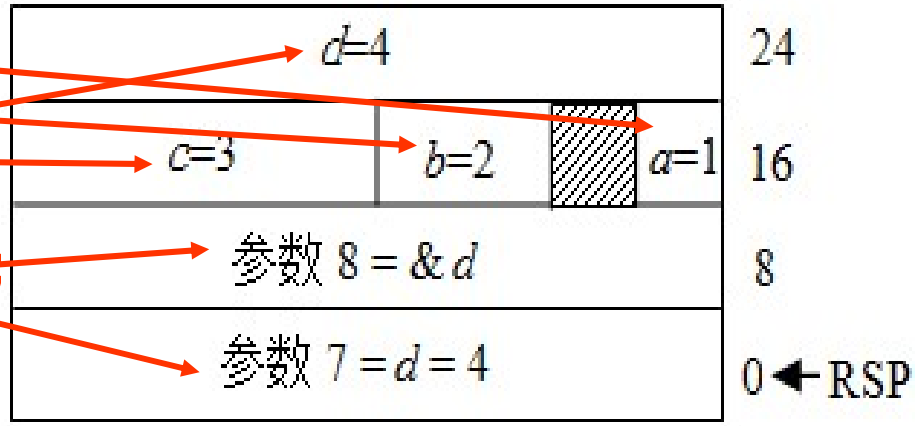
x86-64过程调用举例

```
long caller ( )
{
    char a=1 ;
    short b=2 ;
    int c=3 ;
    long d=4 ;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

其他6个参数在哪里？

```
void test(char a, char *ap,
          short b, short *bp,
          int c, int *cp,
          long d, long *dp)
{
    *ap+=a;
    *bp+=b;
    *cp+=c;
    *dp+=d;
}
```

执行到 caller 的 call 指令时栈中情况



执行到caller的call指令前，栈中的状态如何？

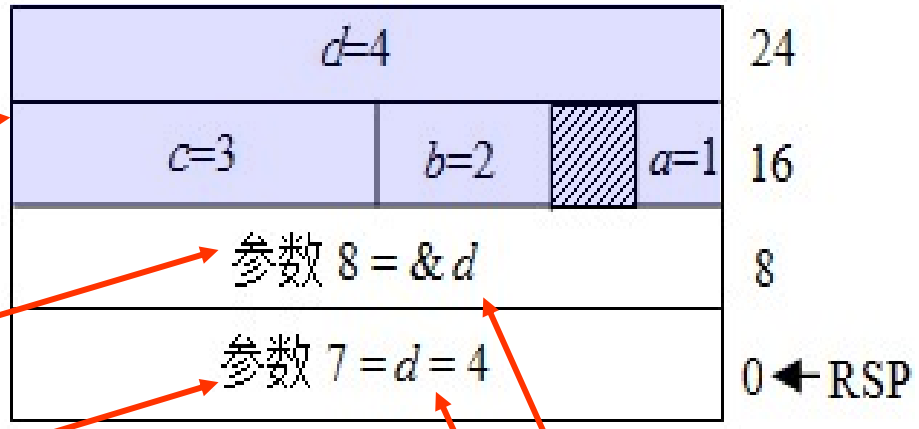
操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

举例：caller函数中部分指令

```
subq $32, %rsp        //R[rsp]←R[rsp]-32
movb $1, 16(%rsp)     //M[R[rsp]+16]←1
movw $2, 18(%rsp)     //M[R[rsp]+18]←2
movl $3, 20(%rsp)     //M[R[rsp]+20]←3
movq $4, 24(%rsp)     //M[R[rsp]+24]←4
leaq 24(%rsp), %rax   //R[rax]←R[rsp]+24
movq %rax, 8(%rsp)    //M[R[rsp]+8]←R[rax]
movq $4, (%rsp)       //M[R[rsp]]←4
leaq 20(%rsp), %r9    //R[r9]←R[rsp]+20
movl $3, %r8d         //R[r8d]←3
leaq 18(%rsp), %rcx   //R[rcx]←R[rsp]+18
movw $2, %dx          //R[dx]←2
leaq 16(%rsp), %rsi   //R[rsi]←R[rsp]+16
movb $1, %dil         //R[dil]←1
call test
```

第15条指令

执行到 caller 的 call 指令时栈中情况




```
long caller ( )
{
    char a=1 ;
    short b=2 ;
    int c=3 ;
    long d=4 ;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

举例：test函数中部分指令

```
movq 16(%rsp), %r10 //R[r10]←M[R[rsp]+16]
addb %dil, (%rsi)   //M[R[rsi]]←M[R[rsi]]+R[dil]
addw %dx, (%rcx)    //M[R[rcx]]←M[R[rcx]]+R[dx]
addl %r8d, (%r9)    //M[R[r9]]←M[R[r9]]+R[r8d]
movq 8(%rsp), %rax  //R[rax]←M[R[rsp]+8]
addq %rax, (%r10)   //M[R[r10]]←M[R[r10]]+R[rax]
ret
```

R[r10]←&d

*ap+=a;

*bp+=b;

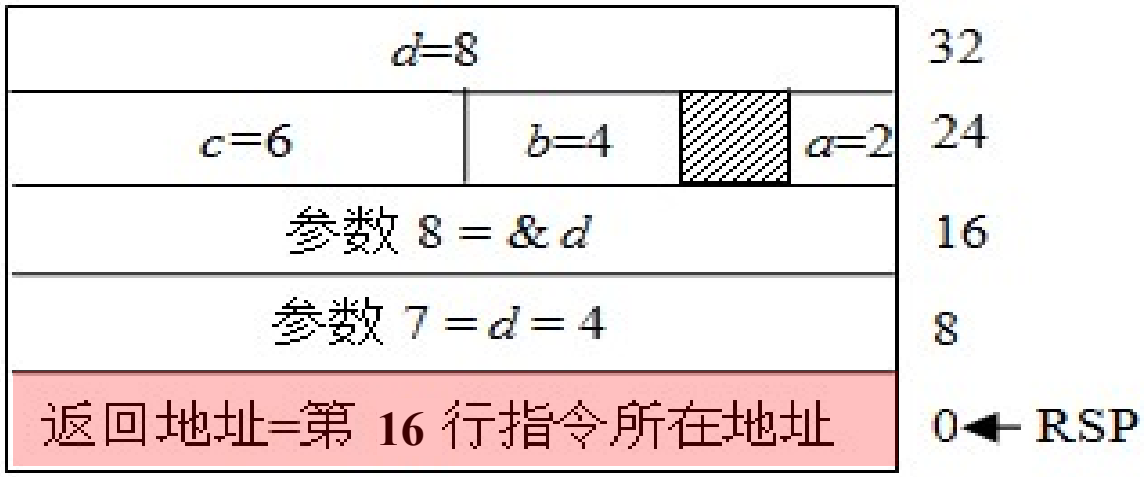
*cp+=c;

*dp+=d;

DIL、RSI、DX、RCX、R8D、R9

```
void test(char a, char *ap,
          short b, short *bp,
          int c, int *cp,
          long d, long *dp)
{
    *ap+=a;
    *bp+=b;
    *cp+=c;
    *dp+=d;
}
```

执行到test的ret指令前，栈中的状态如何？ret执行后怎样？



举例：caller函数中部分指令

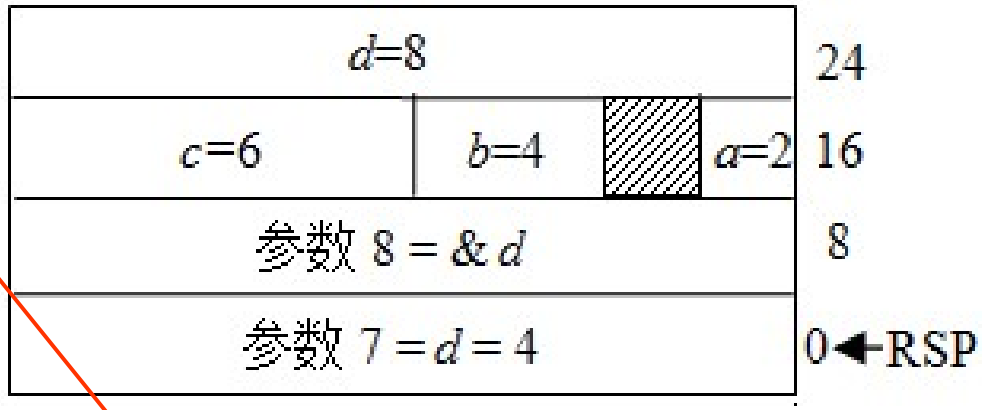
从第16条指令开始

```
movslq 20(%rsp), %rcx
movq   24(%rsp), %rdx
imulq  %rdx, %rcx
movsbw 16(%rsp), %ax
movw   18(%rsp), %dx
imulw  %dx, %ax
movswq %ax, %rax
leaq   (%rax, %rcx),%rax
addq   $32, %rsp
ret
```

释放caller的栈帧

执行到ret指令时，
RSP指向调用caller
函数时保存的返回值

执行test的ret指令后，栈中的状态如何？




```
long caller ( )
{
    char a=1 ;
    short b=2 ;
    int c=3 ;
    long d=4 ;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

IA-32和x86-64的比较

例：以下是一段C语言代码：

```
#include <stdio.h>
main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

在IA-32上运行时，打印结果为a=0

在x86-64上运行时，打印一个不确定值

为什么？

$10=1010B=1.01\times2^3$

阶码e=1023+3=10000000010B

10的double型表示为：

0 10000000010 0100...0B

即4024 0000 0000 0000H

先执行fldl，再执行fstpl

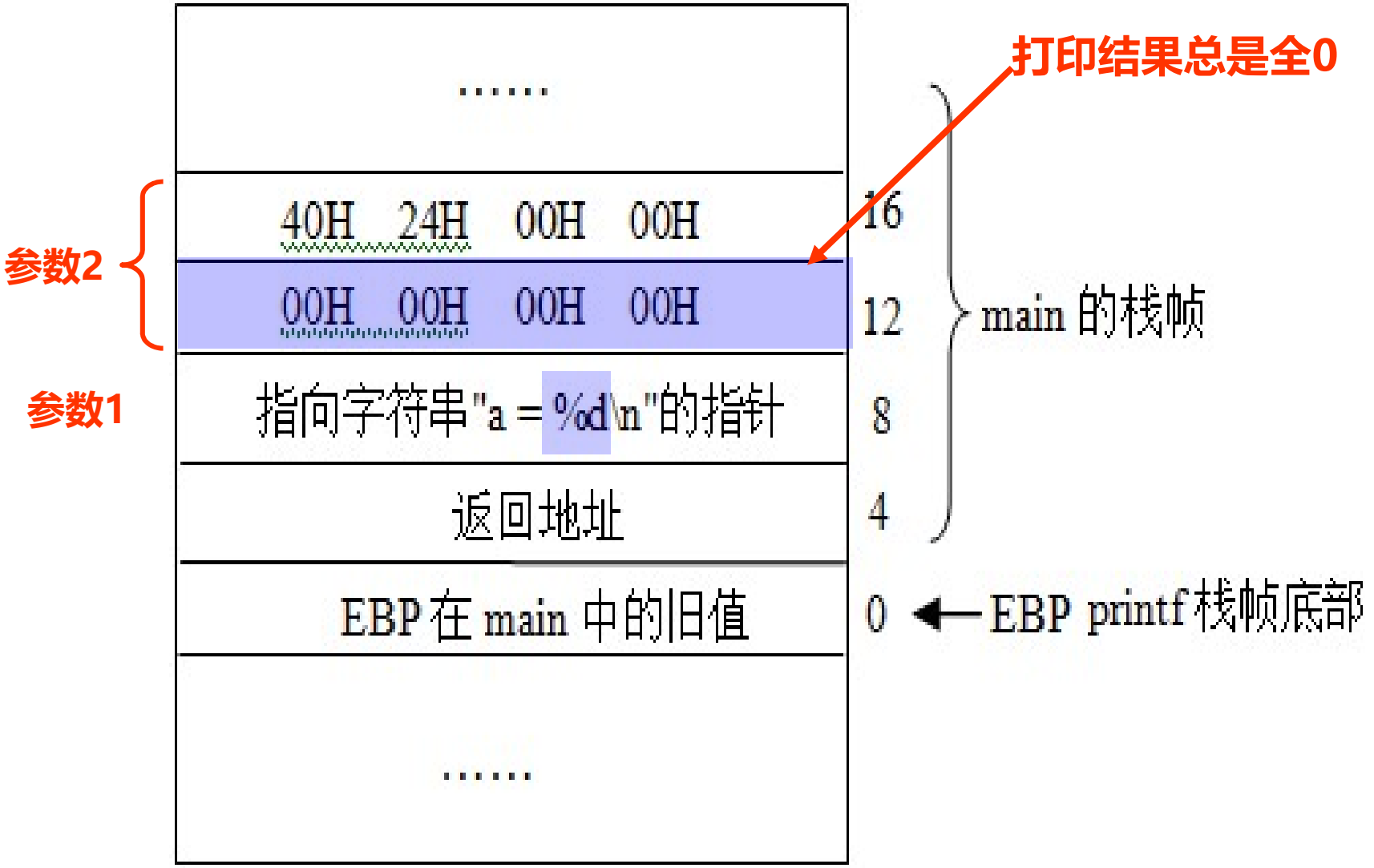
fldl：局部变量区→ST(0)

fstpl：ST(0) →参数区

在IA-32中a为float型又怎样呢？先执行flds，再执行fstpl

即：flds将32位单精度转换为80位格式入浮点寄存器栈，fstpl再将80位转换为64位送存储器栈中，故实际上与a是double效果一样！

IA-32过程调用参数传递



a的机器数对应十六进制为：40 24 00 00 00 00 00 00H

回顾：x86-64的浮点寄存器

- long double型数据虽然还采用80位（10B）扩展精度格式，但所分配存储空间从12B扩展为16B，即改为16B对齐方式，但不管是分配12B还是16B，都只用到低10B
- 128位的XMM寄存器从原来的8个增加到16个
- 浮点操作指令集采用基于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 x87 FPU 指令集
- 浮点操作数存放在XMM寄存器中

x86-64过程调用参数传递

```
main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

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

```
.LC1:
        .string "a = %d\n "
......
movsd   .LC0(%rip), %xmm0 //a 送xmm0
movl    $.LC1, %edi //RDI 高32位为0
movl    $1, %eax    //向量寄存器个数
call     printf
addq    $8, %rsp
ret
......
.LC0:
    .long   0          ←  00000000H
    .long   1076101120 ←  40240000H
```

因为printf第2个参数为double型，故向量寄存器个数为1

小端方式！0存在低地址上

printf中为%d，故将从ESI中取打印参数进行处理；但a是double型数据，在x86-64中，a的值被送到XMM寄存器中而不会送到 ESI 中。故在printf执行时，从ESI 中读取的并不是a的低32位，而是一个不确定的值。