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

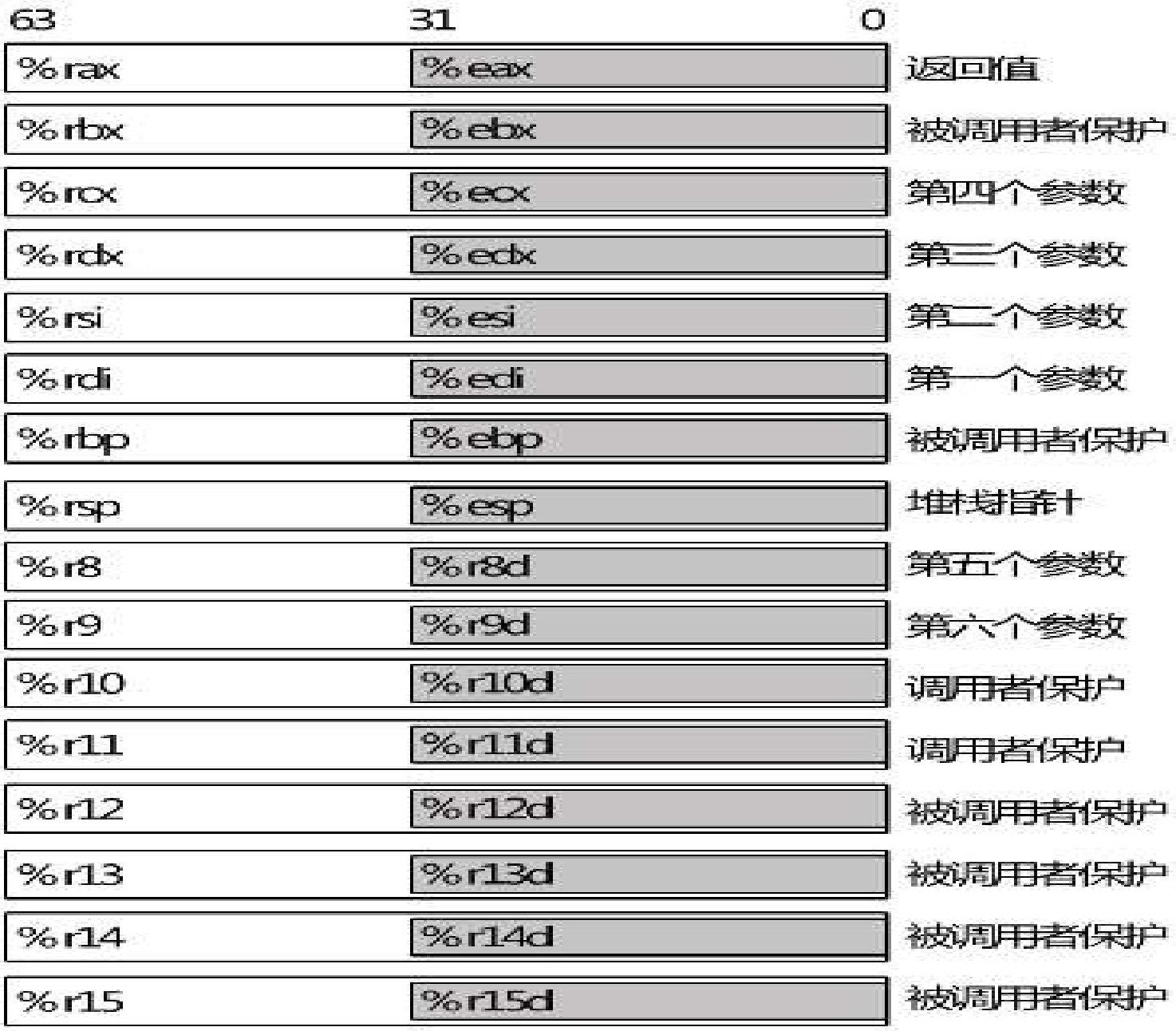
参数传递方式不同

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

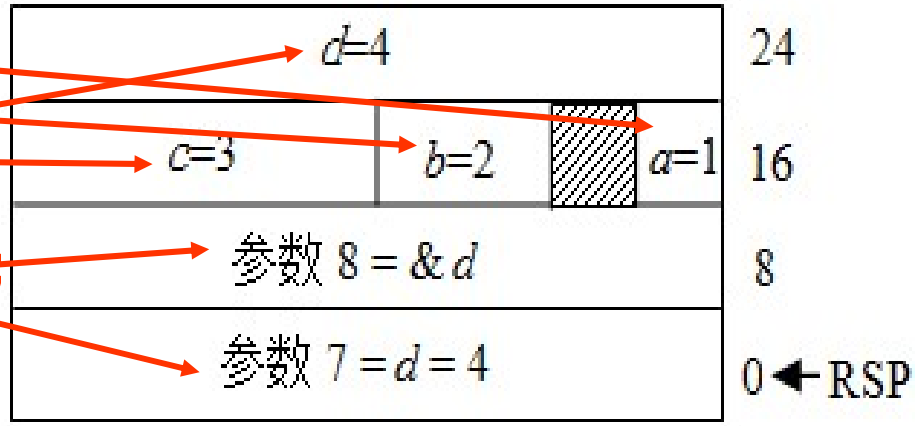
x86-64过程调用举例

```
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？

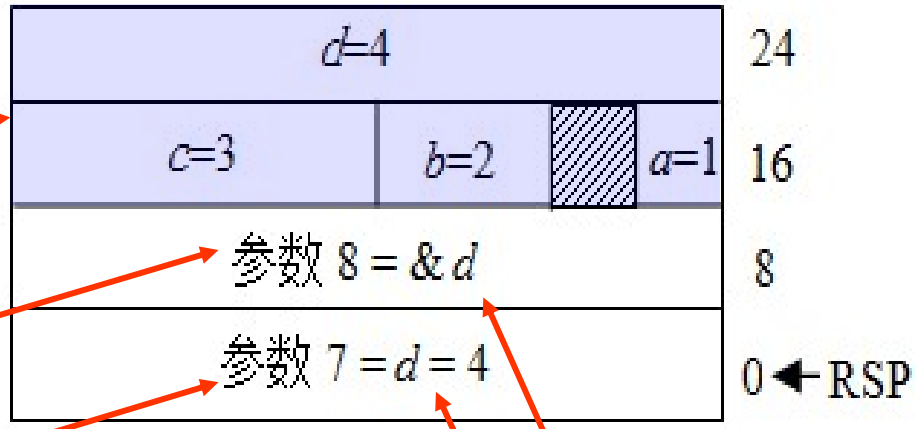
操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

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;

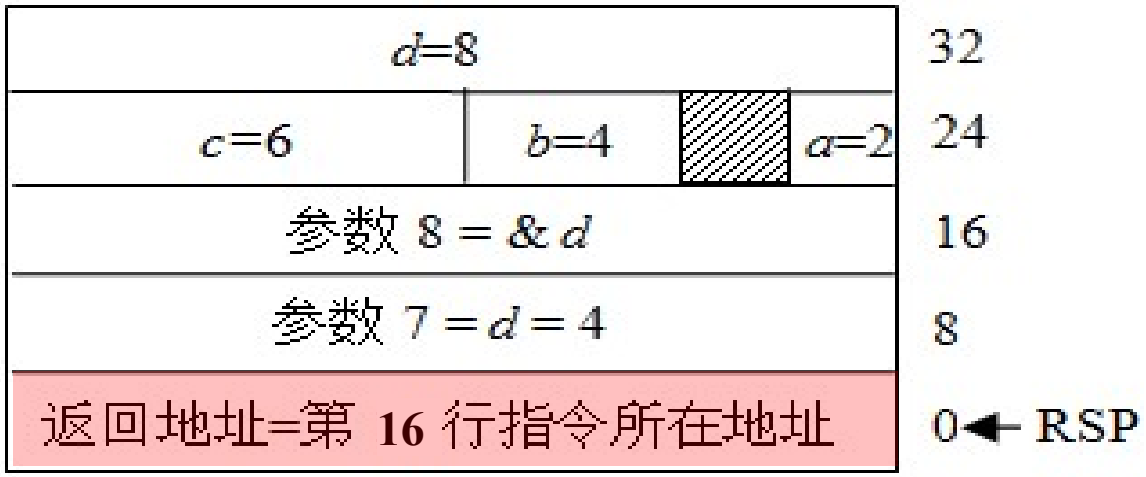
*cp+=c;

*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

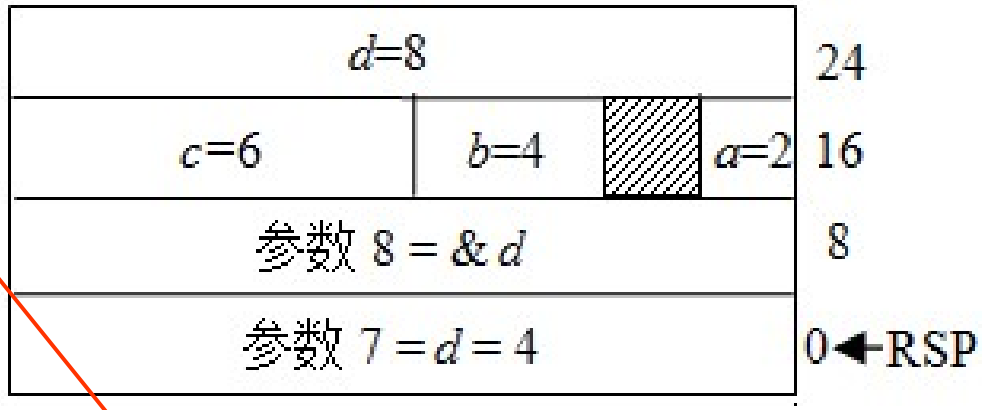
从第16条指令开始

```
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

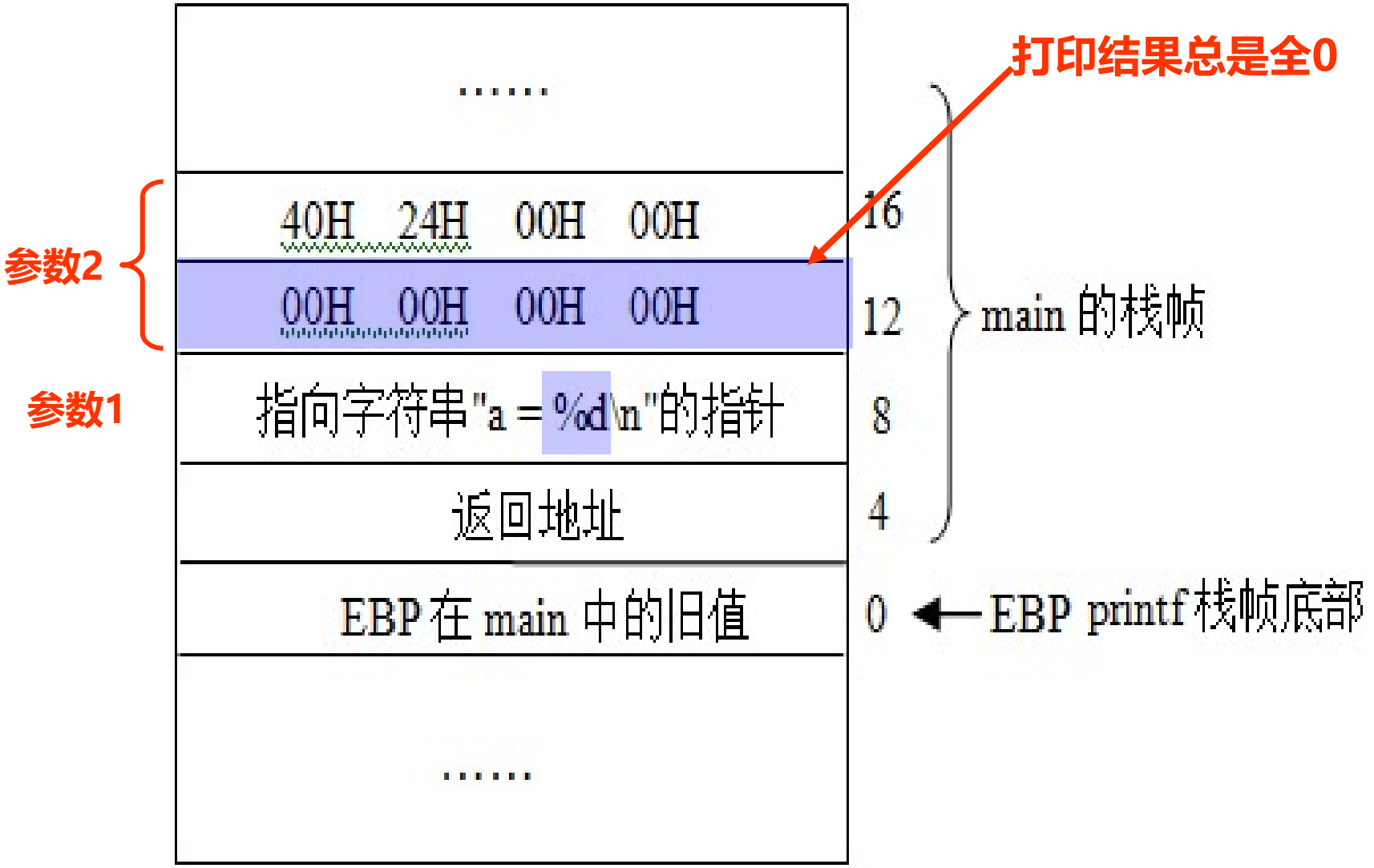
fldl：局部变量区→ST(0)

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。