Machine Prog (Assembly Code) ——Procedures & Data

余文凯 康子熙 赵廷昊 许珈铭

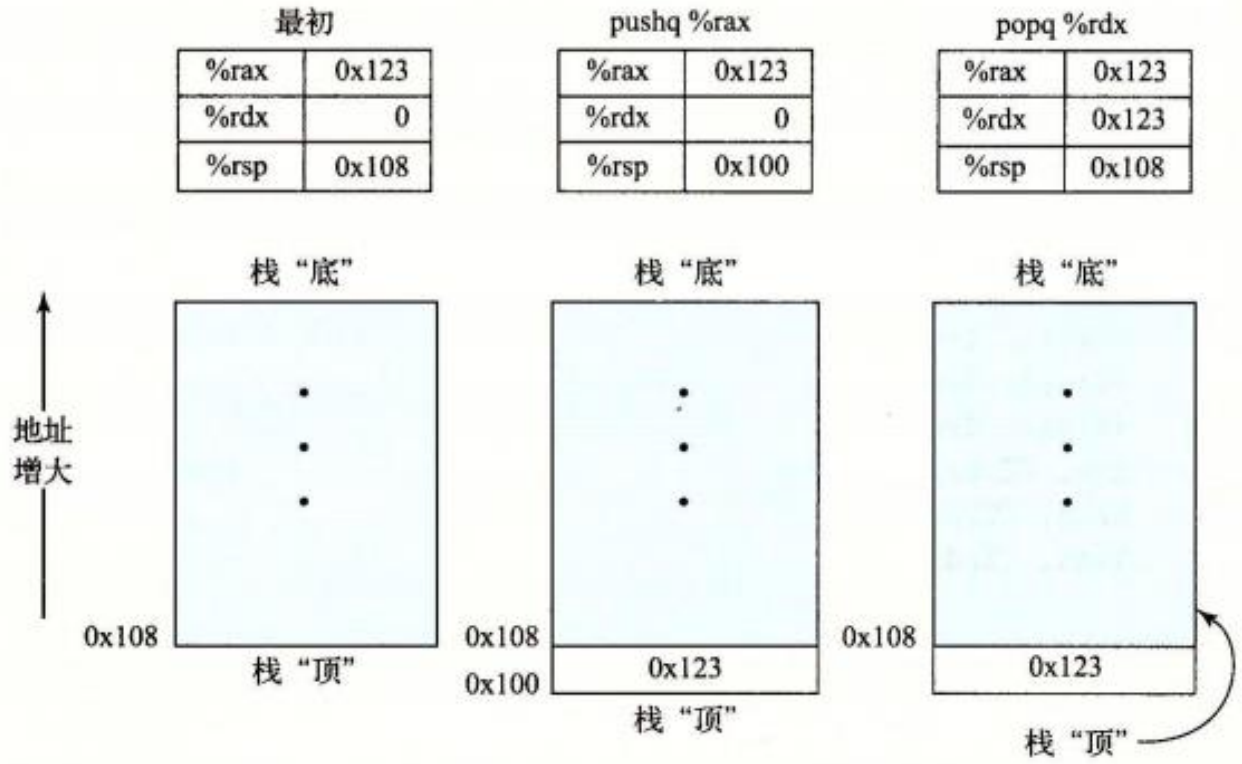
2023.10.11

Procedures (CS:APP Ch. 3.7)

赵廷昊

push pop

- 栈——全局存储
- push数据压入栈 sub+mov
- pop弹出数据 mov+add
- 栈指针%rsp保存栈顶元素地址
- 栈顶元素地址是所有栈中元素地址中最低的



过程调用

- 传递控制
- 传递数据
- 分配和释放内存

调用函数

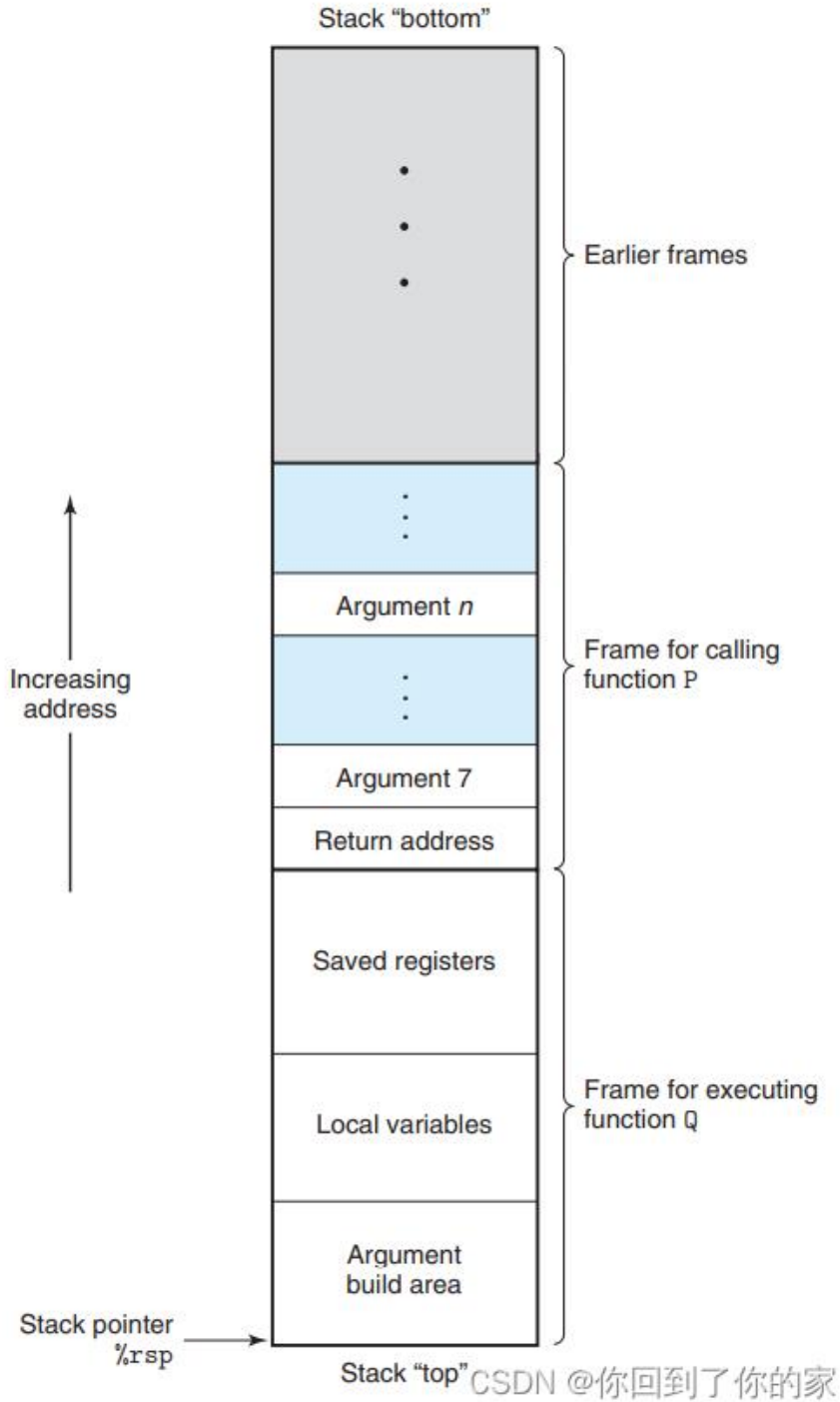
程序计数器地址转移

寄存器与栈

栈帧

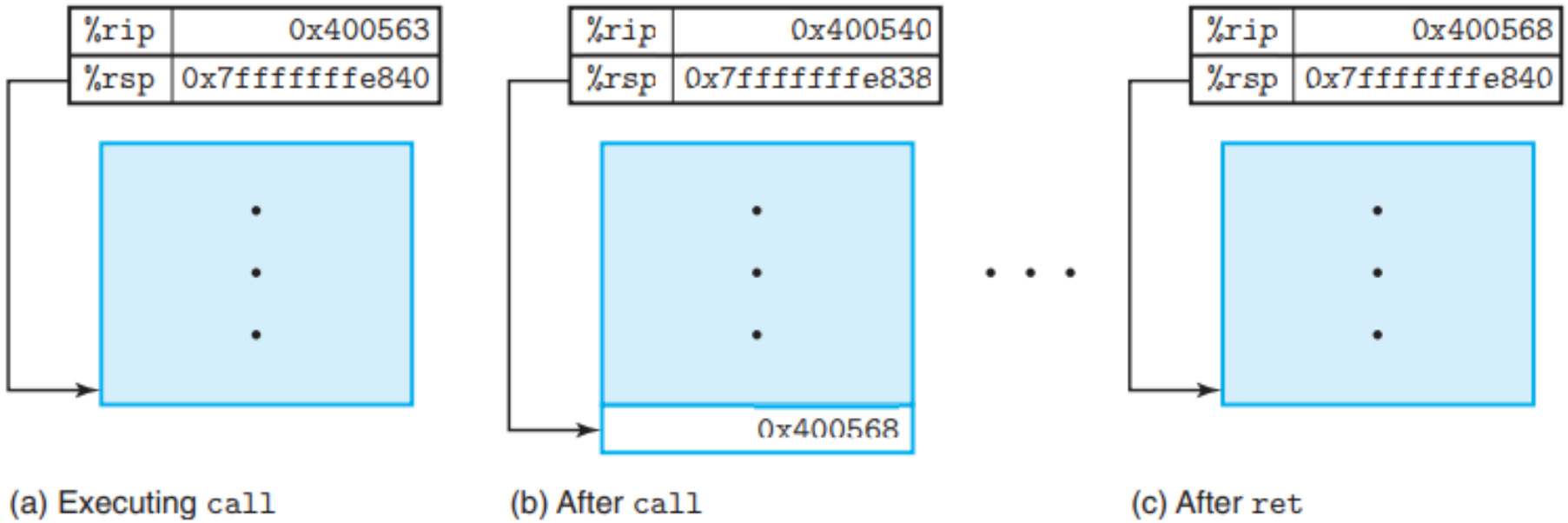
栈帧结构

- 一个栈帧 (stack frame)
  - 单个过程调用所分配的栈的部分
- 大多数过程的栈帧是定长的
- 调用函数P
  - 参数
  - 返回地址
- 正在执行的函数Q
  - 被保存的寄存器
  - 局部变量
  - 参数构造



转移控制

- call Q　　把地址A压入栈中 将PC设置为Q的起始地址
- retq　　从栈中弹出A并把PC设为A
  - 返回值保存在%rax中

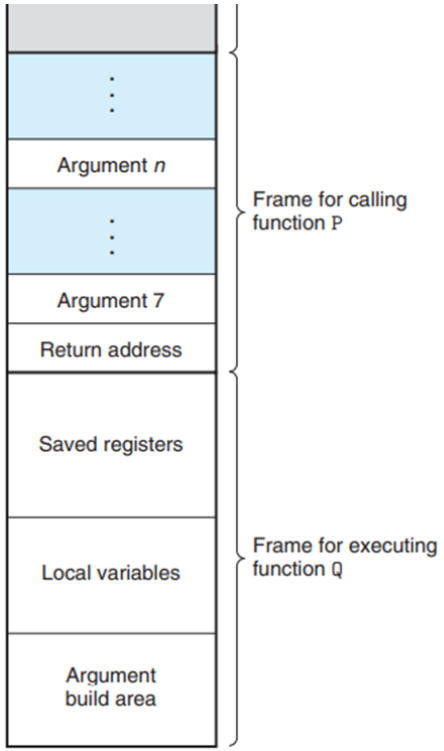


数据传送

函数参数传递

- 在x86-64下，最多可以通过寄存器传递六个integral arguments
- 当一个函数有超过六个integral arguments时，其他参数通过栈进行传递

Operand size (bits)	Argument number					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b



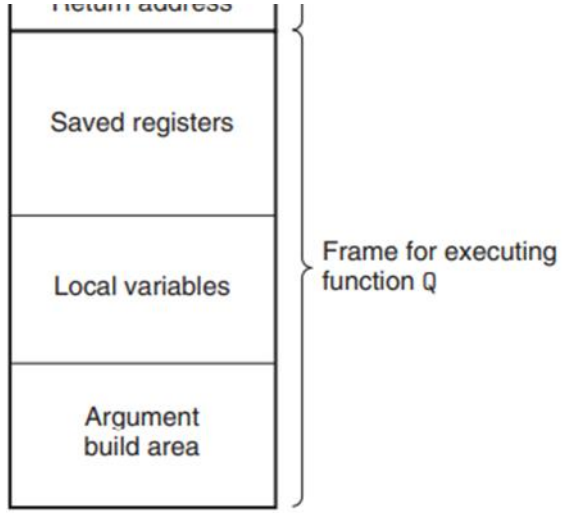
例

```
void proc(long  a1, long  *a1p,
          int   a2, int   *a2p,
          short a3, short *a3p,
          char  a4, char  *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

```
void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
Arguments passed as follows:
  a1  in %rdi          (64 bits)
  a1p in %rsi          (64 bits)
  a2  in %edx          (32 bits)
  a2p in %rcx          (64 bits)
  a3  in %r8w          (16 bits)
  a3p in %r9           (64 bits)
  a4  at %rsp+8        ( 8 bits)
  a4p at %rsp+16       (64 bits)
1 proc:
2   movq    16(%rsp), %rax    Fetch a4p   (64 bits)
3   addq    %rdi, (%rsi)      *a1p += a1  (64 bits)
4   addl    %edx, (%rcx)      *a2p += a2  (32 bits)
5   addw    %r8w, (%r9)       *a3p += a3  (16 bits)
6   movl    8(%rsp), %edx     Fetch a4    ( 8 bits)
7   addb    %dl, (%rax)       *a4p += a4  ( 8 bits)
8   ret                       Return
```

栈上局部存储

- 过程调用 Q 也使用栈存储任何不能存储于寄存器中的局部变量
  - 没有足够的寄存器保存所有的局部变量
  - 一些局部变量是数组或结构体，必须通过数组和结构体引用才能访问
  - 取址操作符 '&' 被应用于局部变量，因此我们必须能为它生成一个地址



例

使用了地址运算符

```
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
```

```
    long caller()
1   caller:
2     subq    $16, %rsp          Allocate 16 bytes for stack frame
3     movq    $534, (%rsp)       Store 534 in arg1
4     movq    $1057, 8(%rsp)     Store 1057 in arg2
5     leaq    8(%rsp), %rsi      Compute &arg2 as second argument
6     movq    %rsp, %rdi         Compute &arg1 as first argument
7     call    swap_add           Call swap_add(&arg1, &arg2)
8     movq    (%rsp), %rdx       Get arg1
9     subq    8(%rsp), %rdx      Compute diff = arg1 - arg2
10    imulq   %rdx, %rax         Compute sum * diff
11    addq    $16, %rsp          Deallocate stack frame
12    ret                        Return
```

寄存器局部存储

- 寄存器%rbx、%rbq和%r12~r15为被调用者保存寄存器
  - 当P调用Q时，Q必须保留这些寄存器的值，确保当Q返回P时这些寄存器的值和P调用Q时是一样的。Procedure Q可以不更改这些寄存器或将原始值压入栈中，修改后再从栈中恢复。

- 除了栈指针%rsp外的其他寄存器为调用者保存寄存器

例

(a) Calling function

```
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

(b) Generated assembly code for the calling function

```
    long P(long x, long y)
    x in %rdi, y in %rsi
1   P:
2     pushq   %rbp            Save %rbp
3     pushq   %rbx            Save %rbx
4     subq    $8, %rsp        Align stack frame
5     movq    %rdi, %rbp      Save x
6     movq    %rsi, %rdi      Move y to first argument
7     call    Q               Call Q(y)
8     movq    %rax, %rbx      Save result
9     movq    %rbp, %rdi      Move x to first argument
10    call    Q               Call Q(x)
11    addq    %rbx, %rax      Add saved Q(y) to Q(x)
12    addq    $8, %rsp        Deallocate last part of stack
13    popq    %rbx            Restore %rbx
14    popq    %rbp            Restore %rbp
15    ret
```

递归过程

- 寄存器与栈使得递归过程调用中每个过程调用在栈中都有自己的私有空间——多个未完成调用的局部变量不会相互影响
- 与调用其他函数没有太大的差别

例

12. 将下列汇编代码翻译成 C 代码

```
func:
    movq    %rsi, %rax
    testq   %rdi, %rdi
    jne .L7
    rep ret
.L7:
    subq    $8, %rsp
    imulq   %rdi, %rax
    movq    %rax, %rsi
    subq    $1, %rdi
    call    func
    addq    $8, %rsp
    ret
```

```
long func(long n, long m) {
    if (__________)
        return _________;
    return func (_____, _____);
}
```

Data (CS:APP Ch. 3.8-Ch. 3.10.1)

康子熙

数组存储的基本原则

对于声明：T A[N]

数组元素i的地址：$X_A + L \cdot i$

所有指针数据类型都是8个字节存储的

```
char    A[12];
char   *B[8];
int     C[6];
double *D[5];
```

These declarations will generate arrays with the following parameters:

Array	Element size	Total size	Start address	Element i
A	1	12	x_A	$x_A + i$
B	8	64	x_B	$x_B + 8i$
C	4	24	x_C	$x_C + 4i$
D	8	40	x_D	$x_D + 8i$

指针运算原则

为什么第7行运行结果是+2？

- 强制类型转换的优先级高于加法
- 指针进行加减法时指针指向的值的加减只与sizeof(T)有关

void* 被称为通用指针，它是一个特殊的指针类型，可以用来指向任何数据类型的对象。

在gcc编译器中，void* 的算术运算会按照 char* 来处理，即每次 p++ 会使指针移动一个字节。但这并不是标准的行为，依赖于此行为是不可移植的

```
1   #include<iostream>
2   using namespace std;
3   int main() {
4       unsigned int A=0x11112222;
5       unsigned int B=0x33336666;
6       void *x = (void *)&A;
7       void *y = 2 + (void *)&B;
8       unsigned short P = *(unsigned short *)x;
9       unsigned short Q = *(unsigned short *)y;
10      printf("0x%08x\n", *(int *)x);
11      printf("0x%08x\n", *(int *)y);
12      printf("0x%04x\n", P);
13      printf("0x%04x\n", Q);
14      printf("0x%04x", Q + P);
15      return 0;
16  }
```

通用指针 void*

通用指针具有以下特点：

- 类型中立：可以将任何类型的指针赋值给void*，且不经过显式类型转换

```
int x = 10;
double y = 20.5;
void* p1 = &x;  // Valid
void* p2 = &y;  // Valid
```

-无法直接解引用：由于 void* 是类型不明确的，所以你不能直接解引用它。

```
int* px = (int*)p1;
int value = *px;
```

需要经过类型转换

使用场景：
- 动态内存分配函数，如 malloc() 和 free()，使用 void*。
- 实现泛型数据结构和函数时，可以使用 void* 来处理各种数据类型。

指针运算原则

```
void *p
int *new_p1 = (int *)p + 7;
   //new_p1 是一个 int 指针，值 = p + 28 --先类型转换后加减
int *new_p2 = (int *)(p + 7);
   //new_p2 是一个 int 指针，值 = p + 7
```

- 指针不能相加
- T * 类型指针相减 = 地址之差 / sizeof (T)
- 不同类型指针不能相减

函数指针

New to C? Function pointers

The syntax for declaring function pointers is especially difficult for novice programmers to understand. For a declaration such as

```
int (*f)(int*);
```

it helps to read it starting from the inside (starting with 'f') and working outward. Thus, we see that `f` is a pointer, as indicated by `(*f)`. It is a pointer to a function that has a single `int *` as an argument, as indicated by `(*f)(int*)`. Finally, we see that it is a pointer to a function that takes an `int *` as an argument and returns `int`.

The parentheses around `*f` are required, because otherwise the declaration

```
int *f(int*);
```

would be read as

```
(int *) f(int*);
```

That is, it would be interpreted as a function prototype, declaring a function `f` that has an `int *` as its argument and returns an `int *`.

指针的阅读——右-左法则 "right-left rule"

- 定位变量名

do {
    向右阅读，直到右括号，确定其类型（是否是数组）
    再向左阅读，直到左括号，确定其内容类型
} while (括号没拆干净)

// 右侧并列的括号可能是函数

例子：int (*(*vtable)[])()

高维数组

对于声明：T D[R][C]

数组元素D[i][j]的地址：$X_D + L(C \cdot i + j)$

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

```
long sum_element(long i, long j)
i in %rdi, j in %rsi
1   sum_element:
2     leaq    0(,%rdi,8), %rdx
3     subq    %rdi, %rdx
4     addq    %rsi, %rdx
5     leaq    (%rsi,%rsi,4), %rax
6     addq    %rax, %rdi
7     movq    Q(,%rdi,8), %rax
8     addq    P(,%rdx,8), %rax
9     ret
```

对于定长数组，优化器可以使用leaq来达到乘法运算的效果

变长数组 VLA

变长数组可以作为局部变量或者函数参数

为什么变长数组不可以作为全局变量？

- **储存方式**：全局变量一般存储在程序的数据段，而不是在栈上。VLA是设计为在栈上分配的，这与全局变量的存储方式不符。

- **初始化时机**：VLA的大小是在运行时确定的，而全局变量的初始化通常发生在程序启动时。这意味着，在程序启动时，我们不会知道VLA的确切大小，这使得为VLA分配和初始化存储变得复杂。

- **安全性**：由于VLA在栈上分配，所以过大的VLA可能会导致栈溢出，这是一个潜在的安全风险。将VLA限制为局部作用域可以减少这种风险，因为程序员更容易控制局部作用域内的资源使用。

变长数组 VLA

变长数组在进行寻址时，由于长度是可变的，汇编时无法使用lea来替代乘法

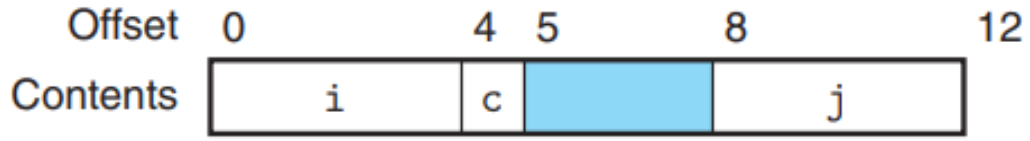
如果使用优化，GCC能够识别出程序访问多维数组元素的步长

```
    Registers: n in %rdi, Arow in %rsi, Bptr in %rcx
               4n in %r9, result in %eax, j in %edx
1   .L24:                                   loop:
2     movl    (%rsi,%rdx,4), %r8d             Read Arow[j]
3     imull   (%rcx), %r8d                    Multiply by *Bptr
4     addl    %r8d, %eax                      Add to result
5     addq    $1, %rdx                        j++
6     addq    %r9, %rcx                       Bptr += n
7     cmpq    %rdi, %rdx                      Compare j:n
8     jne     .L24                            If !=, goto loop
```

结构体 Struct

一个结构体中不同子变量在内存中几乎相邻（对齐可能会影响）

对齐要求某种类型对象的地址必须是某个值K(2/4/8)的倍数



提高数据访问速度：对齐的数据通常可以更快地从内存中加载和存储，因为它们遵循硬件的自然边界。例如，如果一个32位整数从4字节边界开始，那么它可以在单次内存操作中读取，而不需要多次操作。
增强向量化和SIMD效率：对于使用SIMD（单指令多数据）指令的代码，对齐的数据通常提供更好的性能。这是因为SIMD指令经常需要对齐的数据，或者在处理对齐的数据时工作得更快。

联合 Union

- union 的大小是所有子变量大小的最大值
- 所有子变量在内存中起始位置相同
  - 通常情况下各个子变量是互斥的（不会同时被用到）
  - 使用枚举来表示联合中存储的数据是哪种类型，这样就可以在运行时知道如何正确地访问它
- 可用于同位级表示的互转
  - **注意大小端！**

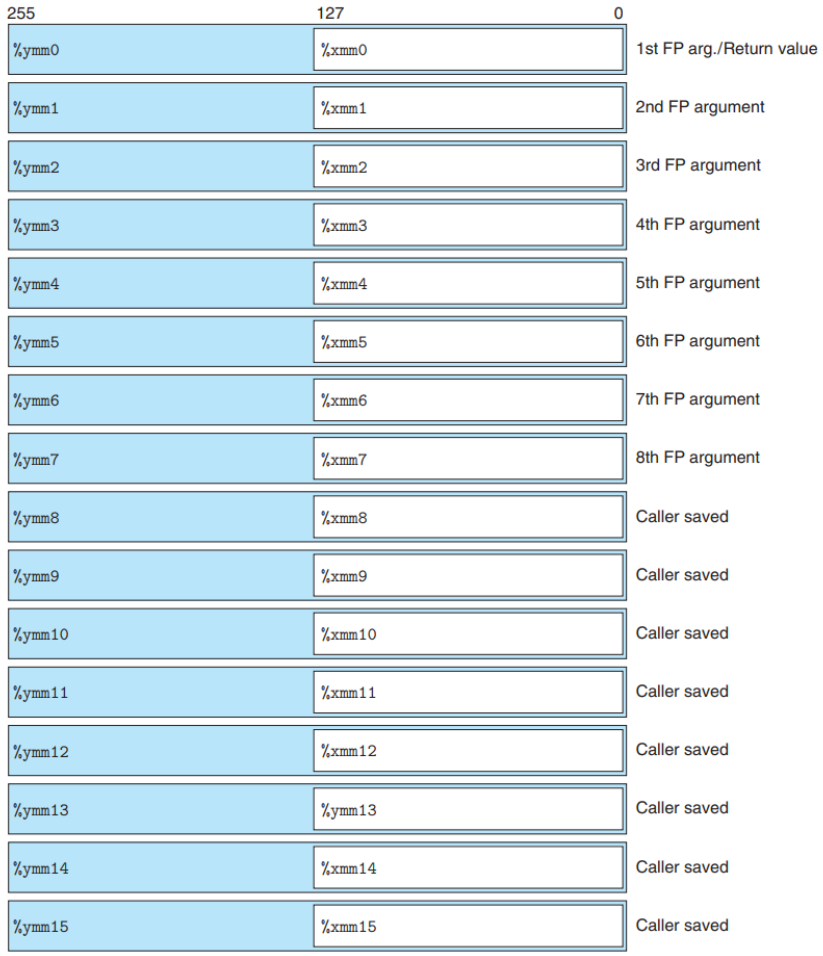
```
double uu2double(unsigned word0, unsigned word1)
{
    union {
        double d;
        unsigned u[2];
    } temp;

    temp.u[0] = word0;
    temp.u[1] = word1;
    return temp.d;
}
```

YMM寄存器

YMM寄存器支持SIMD——单指令多数据。这样做提供了并行模式的优化

- 当对标量数据存储时，这些寄存器只保存浮点数，且只使用低32位或64位
- 函数使用寄存器%xmm0来返回浮点值
- 所有XMM寄存器都是调用者保存的，被调用者可以不用保存就覆盖这些寄存器中的任意一个



浮点传送与转换操作

- **浮点传送指令**

Instruction	Source	Destination	Description
vmovss	M_{32}	X	Move single precision
vmovss	X	M_{32}	Move single precision
vmovsd	M_{64}	X	Move double precision
vmovsd	X	M_{64}	Move double precision
vmovaps	X	X	Move aligned, packed single precision
vmovapd	X	X	Move aligned, packed double precision

浮点传送与转换操作

- **双操作数浮点转换指令：**浮点数——整数

  //在转换时会进行截断，向零舍入

Instruction	Source	Destination	Description
vcvttss2si	X/M_{32}	R_{32}	Convert with truncation single precision to integer
vcvttsd2si	X/M_{64}	R_{32}	Convert with truncation double precision to integer
vcvttss2siq	X/M_{32}	R_{64}	Convert with truncation single precision to quad word integer
vcvttsd2siq	X/M_{64}	R_{64}	Convert with truncation double precision to quad word integer

- **三操作数浮点转换指令：**整数——浮点数

Instruction	Source 1	Source 2	Destination	Description
vcvtsi2ss	M_{32}/R_{32}	X	X	Convert integer to single precision
vcvtsi2sd	M_{32}/R_{32}	X	X	Convert integer to double precision
vcvtsi2ssq	M_{64}/R_{64}	X	X	Convert quad word integer to single precision
vcvtsi2sdq	M_{64}/R_{64}	X	X	Convert quad word integer to double precision

//三操作数的浮点转换指令，第二个操作数只会影响XMM的高64位，可以忽略

浮点传送与转换操作

- float→double

//vunpcklps用来交叉放置两个XMM寄存器的值

```
    Conversion from single to double precision
1   vunpcklps  %xmm0, %xmm0, %xmm0   Replicate first vector element
2   vcvtps2pd  %xmm0, %xmm0          Convert two vector elements to double
```

- double→float

//vmovddup将两个双精度值设置为低位的重复

```
    Conversion from double to single precision
1   vmovddup     %xmm0, %xmm0      Replicate first vector element
2   vcvtpd2psx   %xmm0, %xmm0      Convert two vector elements to single
```

浮点运算

浮点运算指令的第一个源操作数可以是XMM寄存器或内存位置，第二个源操作数和目的操作数必须是XMM寄存器

Single	Double	Effect	Description
vaddss	vaddsd	$D \leftarrow S_2 + S_1$	Floating-point add
vsubss	vsubsd	$D \leftarrow S_2 - S_1$	Floating-point subtract
vmulss	vmulsd	$D \leftarrow S_2 \times S_1$	Floating-point multiply
vdivss	vdivsd	$D \leftarrow S_2 / S_1$	Floating-point divide
vmaxss	vmaxsd	$D \leftarrow \max(S_2, S_1)$	Floating-point maximum
vminss	vminsd	$D \leftarrow \min(S_2, S_1)$	Floating-point minimum
sqrtss	sqrtsd	$D \leftarrow \sqrt{S_1}$	Floating-point square root

浮点运算

浮点运算指令的第一个源操作数可以是XMM寄存器或内存位置，第二个源操作数和目的操作数必须是XMM寄存器。

AVX浮点操作不能以立即数值作为操作数

Single	Double	Effect	Description
vaddss	vaddsd	$D \leftarrow S_2 + S_1$	Floating-point add
vsubss	vsubsd	$D \leftarrow S_2 - S_1$	Floating-point subtract
vmulss	vmulsd	$D \leftarrow S_2 \times S_1$	Floating-point multiply
vdivss	vdivsd	$D \leftarrow S_2/S_1$	Floating-point divide
vmaxss	vmaxsd	$D \leftarrow \max(S_2, S_1)$	Floating-point maximum
vminss	vminsd	$D \leftarrow \min(S_2, S_1)$	Floating-point minimum
sqrtss	sqrtsd	$D \leftarrow \sqrt{S_1}$	Floating-point square root

浮点位级运算

浮点数的位级运算会更新整个XMM寄存器，对所有128位进行位级操作

Single	Double	Effect	Description
vxorps	xorpd	$D \leftarrow S_2 \hat{} S_1$	Bitwise EXCLUSIVE-OR
vandps	andpd	$D \leftarrow S_2 \& S_1$	Bitwise AND

浮点比较运算

浮点数会设置三个条件码：

- ZF：零标志位
- CF：进位标志位
- PF：奇偶标志位

Ordering S_2:S_1	CF	ZF	PF
Unordered	1	1	1
$S_2 < S_1$	1	0	0
$S_2 = S_1$	0	1	0
$S_2 > S_1$	0	0	0

—— 对于浮点比较，当两个操作数任意一个是NaN时，会设置该位

- jp条件跳转：当浮点比较得到无序结果后跳转

Practice

余文凯

The End