Machine Prog : Data

蔡绘霓

目录

- 数组声明
- 指针运算
- 嵌套数组
- 定长数组VS变长数组
- 数组和指针的关系
- 指针辨析&类型解读
- 结构&数据对齐
- 联合及其应用情况
- structure of arrays VS array of structures
- 浮点数及其指令

数组声明

对于数据类型T和整型常数N，声明如下：

T A[N];

作用：在内存中分配一个连续区域；用标识符A表示指向数组开头的指针（值为x_A）。

可以用0~N-1的整数索引访问该数组元素，数组元素i被存放在地址为$x_A+L\cdot i$的地方。

汇编代码：（假设数据类型为int，E的地址存放在寄存器%rdx中，i存放在寄存器%rcx中）

movl (%rdx,%rcx,4),%eax

将E[i]存放在%eax中

指针运算

对指针进行运算时，计算出的值会根据该指针引用的数据类型的大小进行伸缩。如果p是一个指向类型为T的数据的指针，值为x_p，则表达式p+i的值为$x_p+L\cdot i$，L为数据类型T的大小。

对于一个表示某对象的表达式E，&E给出该对象地址的一个指针

对于一个表示地址的表达式E，*E给出该地址处的值

数组引用A[i]等同于表达式*(A+i)

指针运算

两个指针不能相加。

可以计算同一个数据结构中的两个指针之差，结果的数据类型为ptrdiff_t，值等于两个地址之差除以该数据类型的大小。

ptrdiff_t：有符号整型，宽度依赖于编译平台（类似于size_t），在64位平台上就是long。

思考：选用有符号整型的原因？

有可能为负值。

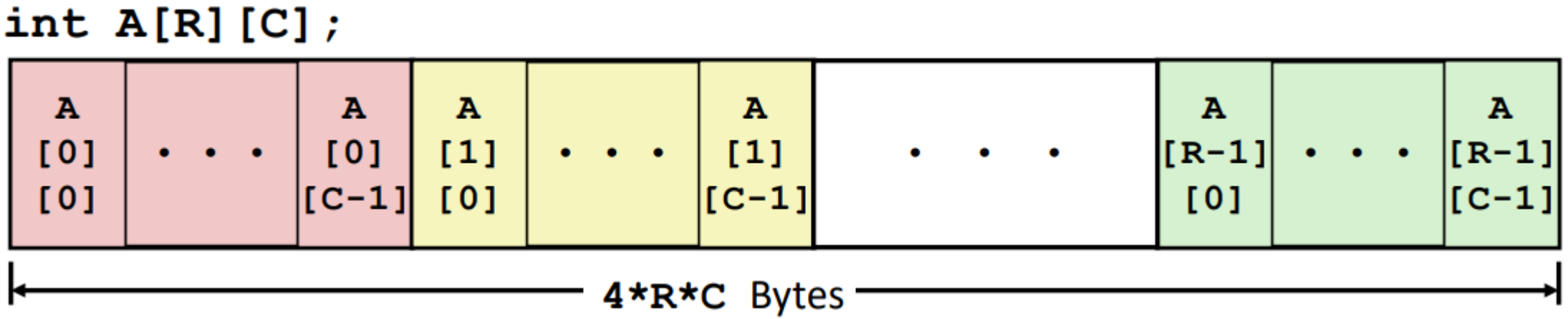
嵌套数组

int A[5][3];

等价于

typedef int row3_t[3]; row3_t A[5];

由此可知，数组元素在内存中按照“行优先”的顺序排列



嵌套数组

对于一个数组：T D[R][C]

$$\&D[i][j]=x_D+L\cdot(C\cdot i+j)$$

L是数据类型T以字节为单位的大小

全局数组VS局部数组

全局数组的元素会被初始化为0，局部数组的元素则是随机值。

全局数组存储在内存中的静态区，局部数组则存储在栈区。占用空间较大的数组建议定义为全局数组。

全局数组分配好内存后其内存大小就不能改变，因此一定是定长数组；局部数组则可以是定长或者变长数组。

定长数组VS变长数组

历史上，C语言只支持大小在编译时就能确定的多维数组。程序员需要变长数组时不得不用malloc或calloc这样的函数为这些数组分配存储空间，而且不得不显式的编码，用行优先索引将多维数组映射到一维数组。

ISO C99引入了一种功能，允许数组的维度是表达式，在数组被分配的时候才计算出来。

定长数组VS变长数组

```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}
```

```
  # a in %rdi, i in %rsi, j in %rdx
  salq    $6, %rsi                # 64*i
  addq    %rsi, %rdi              # a + 64*i
  movl    (%rdi,%rdx,4), %eax     # M[a + 64*i + 4*j]
  ret
```

```
  # n in %rdi, a in %rsi, i in %rdx, j in %rcx
  imulq   %rdx, %rdi              # n*i
  leaq    (%rsi,%rdi,4), %rax     # a + 4*n*i
  movl    (%rax,%rcx,4), %eax     # a + 4*n*i + 4*j
  ret
```

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j)
{
  return a[i][j];
}
```

上图已知数组大小为a[16][16]，下图数组大小则为a[n][n]。

相比定长数组，变长数组必须用乘法指令对i伸缩n倍，而不能用一系列的移位和加法。在一些处理器中，乘法会招致严重的性能处罚。

数组和指针的关系

区别：

1、赋值：同类型指针变量可以相互赋值，而数组不行。

2、求sizeof：sizeof（数组名）表示数组所占内存大小

sizeof（指针名）则固定为4（32位平台）或8（64位平台）

关系：

数组作为参数传给函数时，会退化为指针。

数组和指针的关系（32位机）

```c
#include <stdio.h>

int size(char a[10])
{
    return sizeof(a);
}

int main(void)
{
    char a[] = {'C','h','i','n','a','\0'};
    char *p = "China";
    char *q = a;
    printf("sizeof(a)=%d\n", sizeof(a));  //sizeof(a)=6
    printf("sizeof(p)=%d\n", sizeof(p));  //sizeof(p)=4
    printf("sizeof(q)=%d\n", sizeof(q));  //sizeof(q)=4
    printf("size(a)=%d\n", size(a));      //size(a)=4
     return 0;
}
```

图源：C语言：关于数组退化为指针-CSDN博客

数组和指针的关系

7. 考虑以下 C 语言变量声明：

```
int *(*f[3])();
```

那么在一台 x86-64 机器上，sizeof(f)和 sizeof(*f)的值是多少？

A. 8 24

B. 24 8

C. 8 8

D. 8 不确定

下标运算[]优先级比取值运算*高

指针辨析

int *p()和int (*p)()

前者：p是一个函数，这个函数没有参数且返回值类型为int*

后者：p是一个函数指针，这个函数没有参数且返回值类型为int

int *p[n]和int (*p)[n]

前者：p是一个数组，数组中的n个元素是int*类型的指针

后者：p是一个指针，指向一个int[n]的数组

类型解读

int *(*(**p)[10])(int*)

p是一个指针，它指向一个指针，这个指针指向一个10个元素的数组，数组的元素是函数指针，这个函数返回值是int*，参数是int*。

int (*(*fun(int*(*p)(int *)))[5]) (int*)

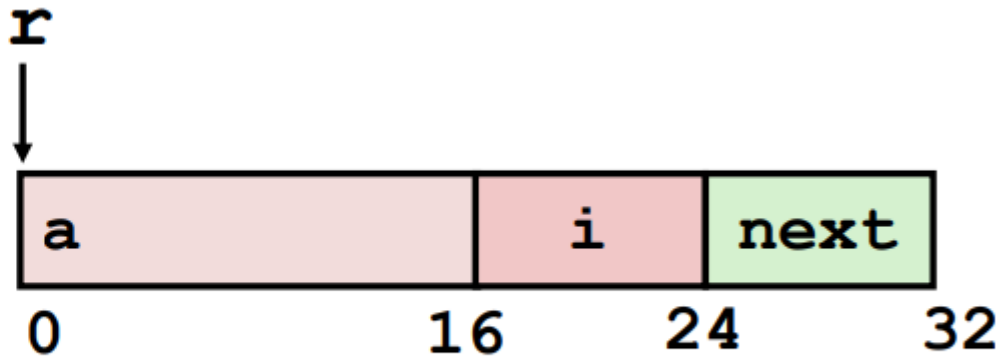
fun是一个函数，它的参数是一个返回值为int*，参数为int*的函数指针p，它的返回值是一个指向5个元素数组的指针，数组的元素是一个返回值为int，参数为int*的函数指针。（来自int (*(*fun(int*(*p)(int *)))[5]) (int*)表示的是什么？ - 知乎 (zhihu.com)）

结构：struct

C语言的struct声明创建一个数据类型，将可能不同类型的对象聚合到一个对象中。类似于数组的实现，结构的所有组成部分都存放在内存中一段连续的区域内，指向结构的指针就是结构第一个字节的地址。

编译器维护关于每个结构类型的信息，指示每个字段的字节偏移。

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



结构

5、假设结构体类型 student_info 的声明如下：

```
struct student_info {
        char id[8];
        char name[16];
        unsigned zip;
        char address[50];
        char phone[20];
}x;
```

若 x 的首地址在%rdx 中，则"unsigned xzip=x.zip;"所对应的汇编指令为：

A. movl 0x24(%rdx), %eax

B. movl 0x18(%rdx), %eax

C. leaq 0x24(%rdx), %rax

D. leaq 0x18(%rdx), %rax

数据对齐

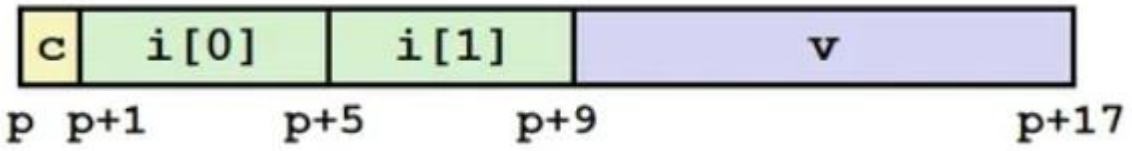
许多计算机系统对基本数据类型的合法地址做出了一些限制，要求某种类型对象的地址必须是某个值K的倍数。这种对齐限制简化了形成处理器和内存系统之间接口的硬件设计。

无论数据是否对齐，x86-64硬件都能正常工作。但Intel还是建议要对齐数据以提高内存系统的性能。

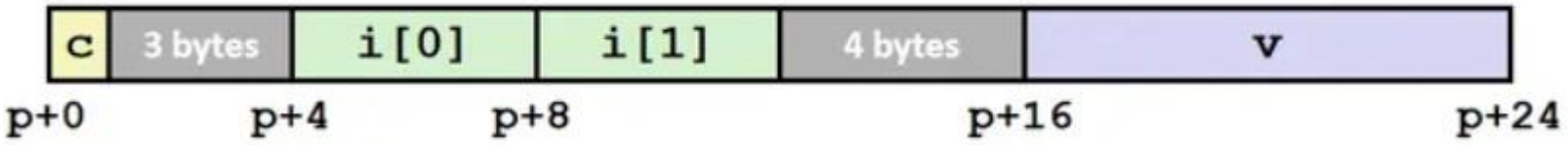
对齐原则是任何K字节的基本对象的地址必须是K的倍数。

数据对齐

对于包含结构的代码，编译器可能在字段的分配中插入空隙，以保证每个结构元素都满足它的对齐要求。



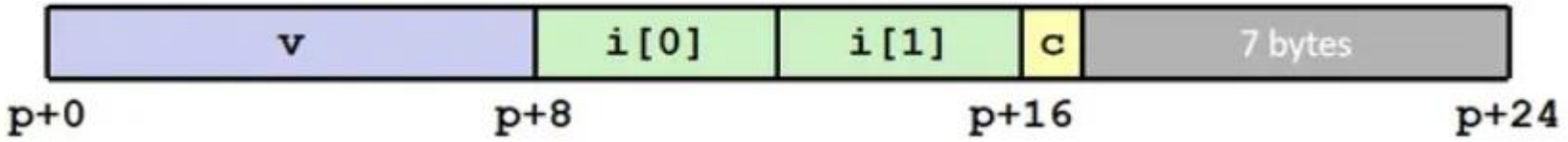

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



数据对齐

同时，结构体整体也有对齐要求，要求与其中最大的数据类型一致。

编译器结构的末尾可能需要一些填充来满足结构体整体的对齐要求。




```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```

意义：假设定义了struct S2 d[4]，在S2末尾的填充保证了d中每个元素的对齐要求。

数据对齐

9.下列结构体的总大小字节数为 A，重新排列优化后的最小字节数是 B，则 A-B=
（　　）

```
struct {
        char *a;   short b;   double c; short d;
        float e;   char  f;   long    g; int    h;
}rec;
```

A. 12　　B. 15

C. 16　　D. 19

数据对齐的最优问题中，当所有的数据元素长度都是2的幂时，一种有效的策略是按照大小的降序（或升序）排列结构的元素。

数据对齐

```
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;
}
```

```
long call_proc()
{
    long  x1 = 1; int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

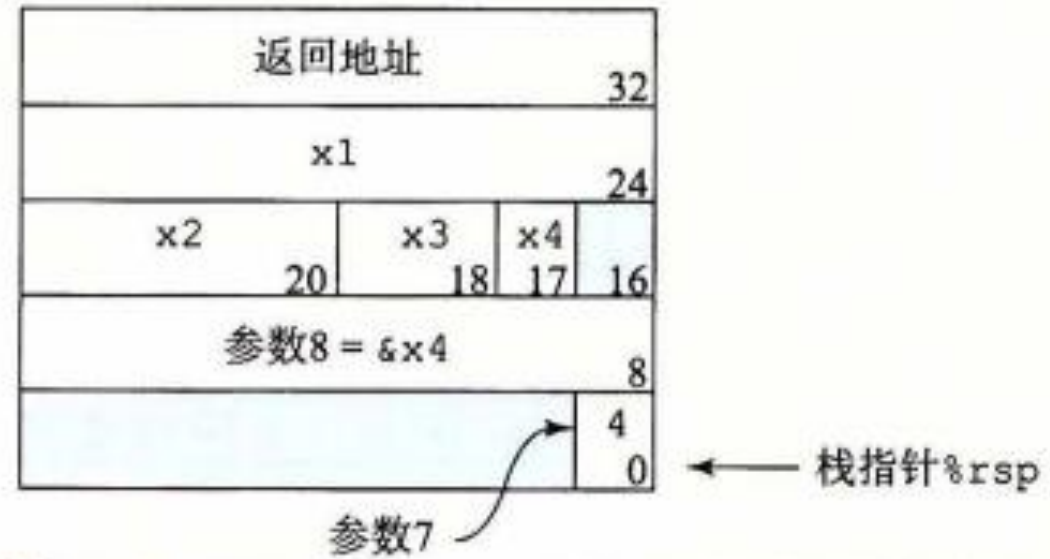


图 3-33 函数 call_proc 的栈帧。该栈帧包含局部变量和两个要传递给函数 proc 的参数

联合： union

联合提供了一种方式，能够规避C语言的类型系统，允许以多种类型来引用一个对象。联合声明的语法和结构的语法一样。

一个联合的总的大小等于它最大字段的大小。

联合的应用情况：

1、一个数据结构中的两个不同字段的使用是互斥的。

2、访问不同数据类型的位模式。

联合的应用情况

假如我们想实现一个二叉树的数据结构，每个叶子节点都有两个double类型的数据值，而每个内部节点都有指向两个孩子节点的指针。

```
struct node_s {
    struct node_s *left;
    struct node_s *right;
    double data[2];
};
```

```
typedef enum { N_LEAF, N_INTERNAL } nodetype_t;

struct node_t {
    nodetype_t type;
    union {
        struct {
            struct node_t *left;
            struct node_t *right;
        } internal;
        double data[2];
    } info;
};
```

联合的应用情况

假设我们想将一个double类型的值d转换为unsigned long类型的值u

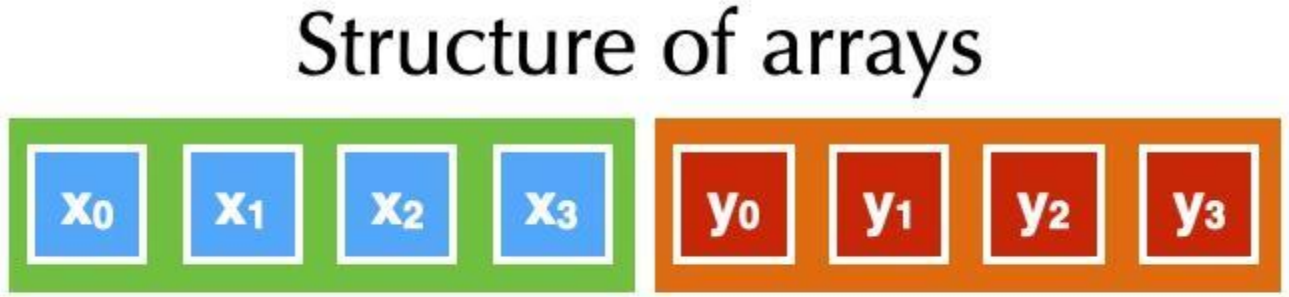
```
unsigned long u = (unsigned long) d;
```

值u会是d的整数表示。u的位表示基本会与d的很不一样。

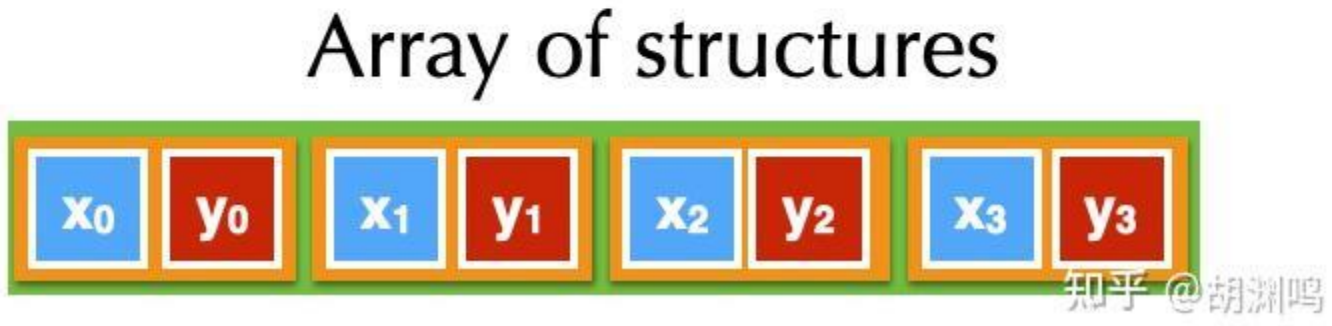
```
unsigned long double2bits(double d) {
    union {
        double d;
        unsigned long u;
    } temp;
    temp.d = d;
    return temp.u;
};
```

u具有和d一样的位表示。u的数值与d的数值基本没有任何关系。

structure of arrays VS array of structures
数组的结构体（SOA） 结构体的数组（AOS）




```
// Structure of arrays
struct Particles {
    float x[1000];
    float y[1000];
    float z[1000];
    float w[1000];
};
```




```
// Array of structures
struct Particle {float x, y, z, w};
Particle particles[1000];
```

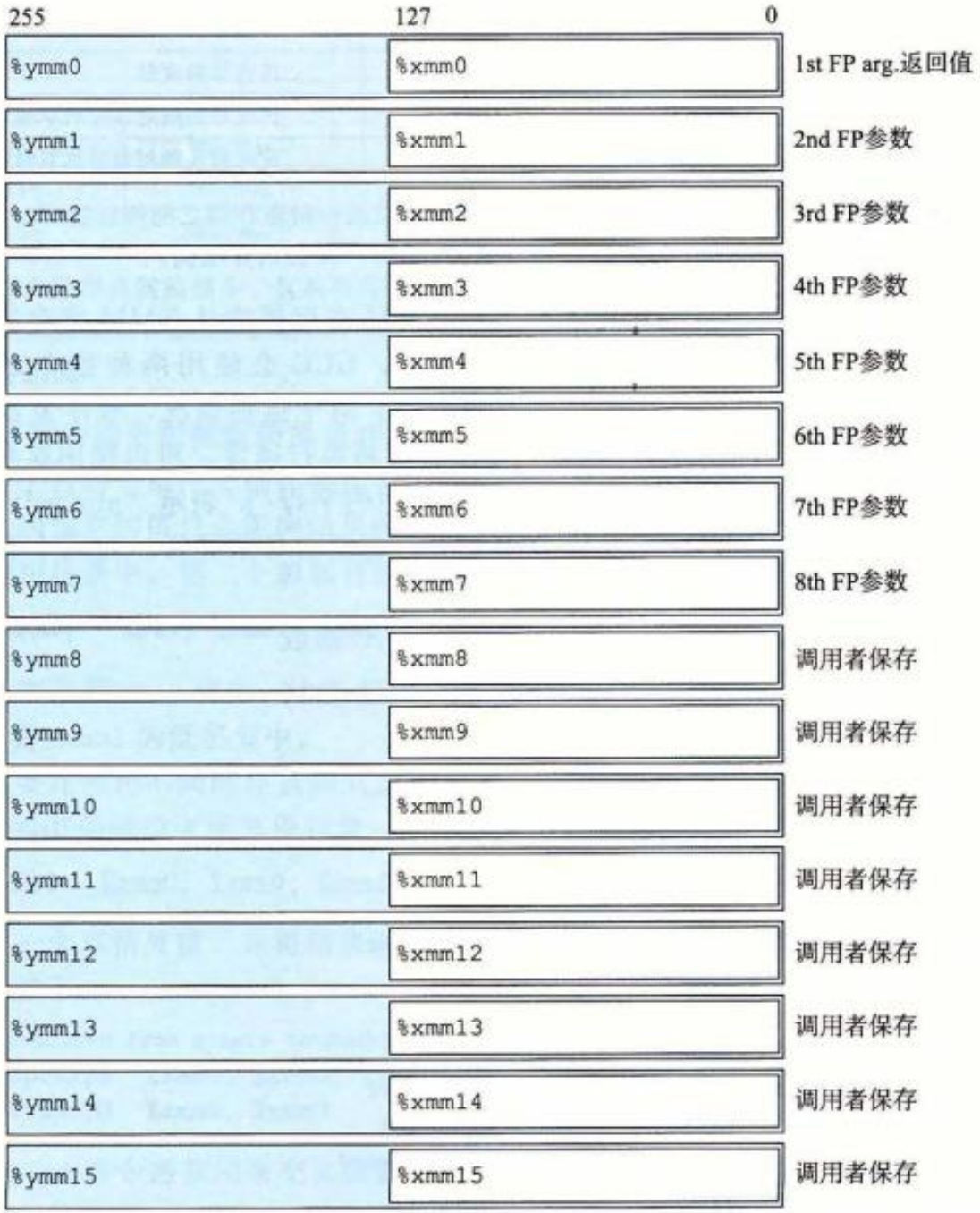
图源：[优化数据排布，让你的程序加速 4 倍！ - 知乎 (zhihu.com)]

浮点数

课本的讲述基于AVX，即AVX的第二个版本。

AVX浮点体系结构允许数据存储在16个YMM寄存器中，它们的名字为%ymm0~%ymm15.每个YMM寄存器都是32字节，当对标量数据操作时，这些寄存器只保存浮点数，而且只使用低32位（float）或64位（double）。汇编代码用寄存器的SSE XMM寄存器名字%xmm0~%xmm15来引用它们，每个XMM寄存器都是对应的YMM寄存器的低128位。

浮点数


255
127
0
%ymm0
%xmm0
1st FP arg.返回值
%ymm1
%xmm1
2nd FP参数
%ymm2
%xmm2
3rd FP参数
%ymm3
%xmm3
4th FP参数
%ymm4
%xmm4
5th FP参数
%ymm5
%xmm5
6th FP参数
%ymm6
%xmm6
7th FP参数
%ymm7
%xmm7
8th FP参数
%ymm8
%xmm8
调用者保存
%ymm9
%xmm9
调用者保存
%ymm10
%xmm10
调用者保存
%ymm11
%xmm11
调用者保存
%ymm12
%xmm12
调用者保存
%ymm13
%xmm13
调用者保存
%ymm14
%xmm14
调用者保存
%ymm15
%xmm15
调用者保存

浮点转换指令

指令	源	目的	描述
vmovss	M_{32}	X	传送单精度数
vmovss	X	M_{32}	传送单精度数
vmovsd	M_{64}	X	传送双精度数
vmovsd	X	M_{64}	传送双精度数
vmovaps	X	X	传送对齐的封装好的单精度数
vmovapd	X	X	传送对齐的封装好的双精度数

图 3-46　浮点传送指令。这些操作在内存和寄存器之间以及一对寄存器之间传送值（X：XMM寄存器（例如%xmm3）；M_{32}：32 位内存范围；M_{64}：64 位内存范围）

无论数据对齐与否，这些指令都能正确执行。不过代码优化规则建议32位内存数据满足4字节对齐，64位数据满足8字节对齐。

浮点传送指令

指令	源	目的	描述
vcvttss2si	X/M_{32}	R_{32}	用截断的方法把单精度数转换成整数
vcvttsd2si	X/M_{64}	R_{32}	用截断的方法把双精度数转换成整数
vcvttss2siq	X/M_{32}	R_{64}	用截断的方法把单精度数转换成四字整数
vcvttsd2siq	X/M_{64}	R_{64}	用截断的方法把双精度数转换成四字整数

图 3-47 双操作数浮点转换指令。这些操作将浮点数转换成整数（X：XMM 寄存器（例如%xmm3）；R_{32}：32 位通用寄存器（例如%eax）；R_{64}：64 位通用寄存器（例如%rax）；M_{32}：32 位内存范围；M_{64}：64 位内存范围）

指令	源 1	源 2	目的	描述
vcvtsi2ss	M_{32}/R_{32}	X	X	把整数转换成单精度数
vcvtsi2sd	M_{32}/R_{32}	X	X	把整数转换成双精度数
vcvtsi2ssq	M_{64}/R_{64}	X	X	把四字整数转换成单精度数
vcvtsi2sdq	M_{64}/R_{64}	X	X	把四字整数转换成双精度数

图 3-48 三操作数浮点转换指令。这些操作将第一个源的数据类型转换成目的的数据类型。第二个源值对结果的低位字节没有影响（X：XMM 寄存器（例如%xmm3）；M_{32}：32 位内存范围；M_{64}：64 位内存范围）

浮点运算操作

单精度	双精度	效果	描述
vaddss	vaddsd	$D \leftarrow S_2 + S_1$	浮点数加
vsubss	vsubsd	$D \leftarrow S_2 - S_1$	浮点数减
vmulss	vmulsd	$D \leftarrow S_2 \times S_1$	浮点数乘
vdivss	vdivsd	$D \leftarrow S_2 / S_1$	浮点数除
vmaxss	vmaxsd	$D \leftarrow \max(S_2, S_1)$	浮点数最大值
vminss	vminsd	$D \leftarrow \min(S_2, S_1)$	浮点数最小值
sqrtss	sqrtsd	$D \leftarrow \sqrt{S_1}$	浮点数平方根

图 3-49　标量浮点算术运算。这些指令有一个或两个源操作数和一个目的操作数

浮点位级操作

单精度	双精度	效果	描述
vxorps	vorpd	$D \leftarrow S_2 \hat{} S_1$	位级异或(EXCLUSIVE−OR)
vandps	andpd	$D \leftarrow S_2 \& S_1$	位级与(AND)

图 3-50　对封装数据的位级操作(这些指令对一个 XMM 寄存器中的所有 128 位进行布尔操作)

浮点比较操作

指令	基于	描述
ucomiss S_1，S_2	S_2-S_1	比较单精度值
ucomisd S_1，S_2	S_2-S_1	比较双精度值

条件码的设置条件如下：

顺序 S_2：S_1	CF	ZF	PF
无序的	1	1	1
$S_2<S_1$	1	0	0
$S_2=S_1$	0	1	0
$S_2>S_1$	0	0	0

Thanks for watching !