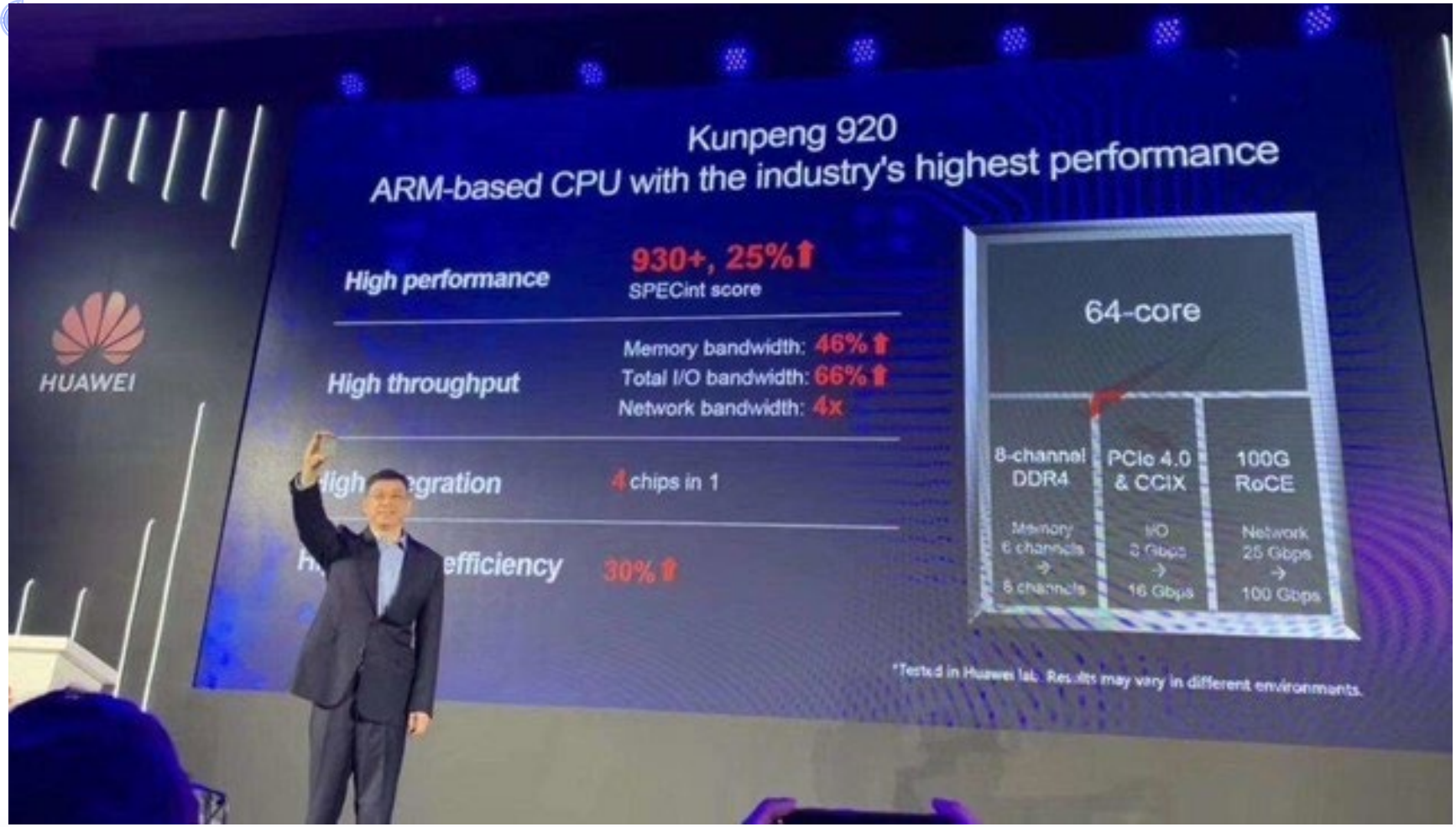
华为鲲鹏处理器简介

华为鲲鹏处理器


Kunpeng 920
ARM-based CPU with the industry's highest performance
HUAWEI
High performance
930+, 25%
SPECint score
High throughput
Memory bandwidth: 46%
Total I/O bandwidth: 66%
Network bandwidth: 4x
4 chips in 1
30%
64-core
8-channel DDR4
Memory
6 channels
8 channels
PCIe 4.0 & CCIX
I/O
16 Gbps
100G RoCE
Network
25 Gbps
100 Gbps

华为鲲鹏处理器

复杂指令集计算机

CISC—— Complex Instruction Set Computer

Inetl x86、AMD

精简指令集计算机

RISC—— Reduced Instruction Set Computer

ARM 进阶精简指令集机器 Advanced RISC Machine

华为鲲鹏处理器 —— ARM based CPU

华为鲲鹏处理器 ARM based CPU

- ARM： Advanced RISC Machines
- 1985年第一个ARM原型在英国剑桥诞生
- 公司 只设计芯片，设计了大量高性能、廉价、耗能低的RISC处理器
- 公司不生产芯片
- 它提供ARM技术知识产权（IP）核，将技术授权给世界上许多著名的半导体、软件和OEM厂商，并提供服务；
- ARM， 一个公司的名字
  - 一类微处理器的通称
  - 一种技术的名字

华为鲲鹏处理器 ARM based CPU

ARM9 微处理器系列

- 哈佛体系结构
- 5级整数流水线
- 支持32位 ARM 指令集 和16位 Thumb 指令集
- 全性能的MMU，支持数据Cache和指令Cache，具有更高的指令和数据处理能力
- 高性能和低功耗
- 支持Windows CE、Linux、Palm OS等多种主流嵌入式操作系统

冯·诺依曼结构 VS 哈佛体系结构

- 将数据和指令都存储在存储器中的计算机；
- 计算系统由一个中央处理单元（CPU）和一个存储器组成；
- 存储器拥有数据和指令，可以根据所给的地址对它进行读写；
- 程序指令和数据的宽度相同（一种数据总线）

  Intel 8086、ARM7、MIPS处理器等

- 为数据和程序提供了各自独立的存储器；
- 程序计数器只指向程序存储器而不指向数据存储器，很难在哈佛机上编写出一个自修改的程序；
- 指令和数据可以有不同的数据宽度；

  ARM9、ARM10 处理器等

ARM 与 X86 指令集的比较

- ARM指令集是固定大小，固定格式的指令编码。
  指令地址32bit 位宽，4字节对齐；
- X86是可变长度指令集体系结构。指令地址没有对齐要求。

- ARM是一种load-store架构
  数据处理指令不能直接对内存的内容进行操作，仅对寄存器操作。
  加载和存储指令只能在寄存器和内存之间传输数据。
  X86 的数据处理指令可以直接在内存以及寄存器上处理数据。
- X86 支持访问I/O地址空间的单独I/O指令
  X86 包括一部分指令可以直接对IO地址空间进行操作。
  IN、OUT指令直接对I/O 端口进行数据读写。
  ARM没有等效功能，而是假定所有外围设备都在标准4GB地址空间内映射到内存里的。

ARM 与 X86 指令集的比较


```
    int  x = 10;
C7 45 F8 0A 00 00 00 mov         dword ptr [x],0Ah
    int  y = 20;
C7 45 EC 14 00 00 00 mov         dword ptr [y],14h
    int  z;
    z = 2 * x + 3 * y;
6B 45 EC 03          imul        eax,dword ptr [y],3
8B 4D F8             mov         ecx,dword ptr [x]
8D 14 48             lea         edx,[eax+ecx*2]
89 55 E0             mov         dword ptr [z],edx
```

X86 变长指令；
数据处理指令可访问存储单元

```
int x=10;
0000004006a0 <main+8>: 40 01 80 52     mov     w0, #0xa
0000004006a4 <main+12>:        a0 1f 00 b9     str     w0, [x29, #28]

int y=20;
0000004006a8 <main+16>:        80 02 80 52     mov     w0, #0x14

0000004006ac <main+20>:        a0 1b 00 b9     str     w0, [x29, #24]

int z;
z=2*x+3*y;
0000004006b0 <main+24>:        a0 1f 40 b9     ldr     w0, [x29, #28]
0000004006b4 <main+28>:        02 78 1f 53     lsl     w2, w0, #1
0000004006b8 <main+32>:        a1 1b 40 b9     ldr     w1, [x29, #24]
0000004006bc <main+36>:        e0 03 01 2a     mov     w0, w1
0000004006c0 <main+40>:        00 78 1f 53     lsl     w0, w0, #1
0000004006c4 <main+44>:        00 00 01 0b     add     w0, w0, w1
0000004006c8 <main+48>:        40 00 00 0b     add     w0, w2, w0
0000004006cc <main+52>:        a0 17 00 b9     str     w0, [x29, #20]
```

ARM 等长指令；
数据处理仅对寄存器操作

ARM 与 X86 指令集的比较

无法在ARM指令中嵌入任意32位地址

- 所有的内存访问，都是基于存放在一个寄存器中的地址进行索引的，通过通用寄存器间接进行的。

X86指令中可嵌入任意32位地址

- 可以进行直接地址访问，指令集的可变长度。

ARM指令不能包含任意的32位常量

- 操作长度较长的立即数时，通过mov/movk 多次进行生成。

ARM 与 X86 指令集的比较

int u; // 全局变量

u=20;

```
00 01 00 90      adrp    x0, 0x420000 <
00 d0 00 91      add     x0, x0, #0x34
81 02 80 52      mov     w1, #0x14
01 00 00 b9      str     w1, [x0]
```

x0: 64位的通用寄存器

前两条语句得到 u的地址放到 x0中。

adrp

w1： x1的低32位

u=0x123456;

```
00 01 00 90      adrp    x0, 0x420000 <__libc_
00 d0 00 91      add     x0, x0, #0x34
c1 8a 86 52      mov     w1, #0x3456
41 02 a0 72      movk    w1, #0x12, lsl #16
01 00 00 b9      str     w1, [x0]
```

ARM 与 X86 指令集的比较

- X86 使用分段寻址模型

  所有 x86 内存/存储器访问都相对于段寄存器之一，因此必须首先设置这些访问。 较大的偏移量需要较大的指令才能对较大的常数进行编码。

- ARM没有分段寻址的概念，并且没有等效的分段寄存器。

  DS: Offset(base,index,scale) → offset(base)

ARM 64　31个64位的通用寄存器

- x0~x7：传递子程序的参数和返回值，使用时不需要保存，多余的参数用堆栈传递，64位的返回结果保存在x0中。
- x8：用于保存子程序的返回地址，使用时不需要保存。
- x9~x15：临时寄存器，也叫可变寄存器，子程序使用时不需要保存。
- x16~x17：子程序内部调用寄存器（IPx），使用时不需要保存，尽量不要使用。
- x18：平台寄存器，它的使用与平台相关，尽量不要使用。
- x19~x28：临时寄存器，子程序使用时必须保存。
- x29：帧指针寄存器（FP），用于连接栈帧，使用时必须保存。
- x30：链接寄存器（LR），用于保存子程序的返回地址。

ARM 64　　专用寄存器

- x31：堆栈指针寄存器（SP），用于指向每个函数栈顶。
- pc：指令指针寄存器

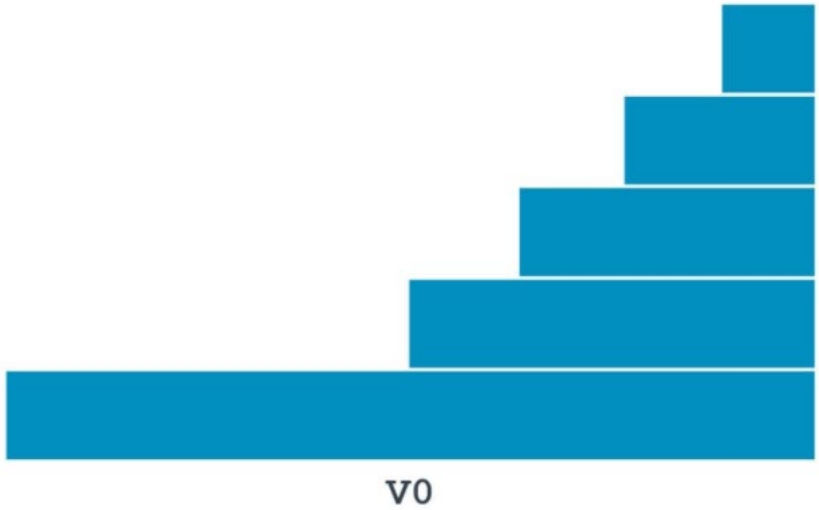
- w0~w30：　x0 ~x30 的低32位。

在gdb 中，通过 i reg 可以看到 寄存器的名称及相应的值

华为鲲鹏处理器 ARM based CPU

- V0~v31：128 位的 SIMD 寄存器，用于浮点数和向量运算。

- Arm64 与 arm32 是两套不同的指令集
- SIMD 指令集完全不同



B：字节 8位

H：16位

S ：32位

D ：64位

Q ：128位

Vn 的组成部分 Bn、Hn、Sn、Dn、Qn，可独立使用

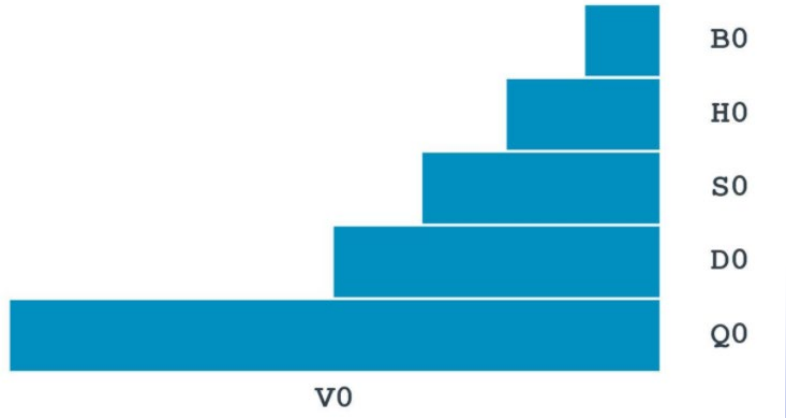
华为鲲鹏处理器 ARM based CPU

- V0~v31：128 位的 SIMD 寄存器，用于浮点数和向量运算。

当使用V形式时，寄存器被视为一个向量 （SIMD 指令集）
此时它被视为包含多个独立值，而不是单个值。

ADD V0.2D, V1.2D, V2.2D　　整数向量加

FADD V0.2D, V1.2D, V2.2D　　浮点数向量加



ARM 的 9 种寻址方式

- 立即寻址： 操作数是立即数，以“#”为前缀。
- 寄存器寻址： 操作数的值在寄存器中。
- 寄存器偏移寻址： 寄存器中的值进行移位操作。
- 寄存器间接寻址： 寄存器为操作数的地址指针。
- 基址寻址： 寄存器的值与偏移量相加，形成操作数的地址。
- 多寄存器寻址： 一次传送多个寄存器值，允许一条指令传送 16 个寄存器的任何子集或所有寄存器。
- 堆栈寻址。
- 块拷贝寻址： 将一块数据从存储器的某一位置拷贝到另一位置。
- 相对寻址： 基址寻址的一种变通，由程序计数器 PC 提供基准地址、以及偏移量，两者相加后得到有效地址。

华为鲲鹏处理器 ARM based CPU

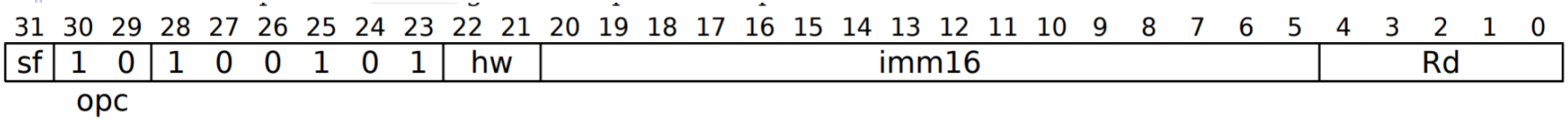

```
[root@localhost ~]# cat test.c
#include <stdio.h>
int main()
{
    int x=10;
    int y=20;
    int z;
    z=2*x+3*y;
     printf("%d \n",z);
    return 0;
}

[root@localhost ~]# gcc -g test.c -o mytest
[root@localhost ~]#
```

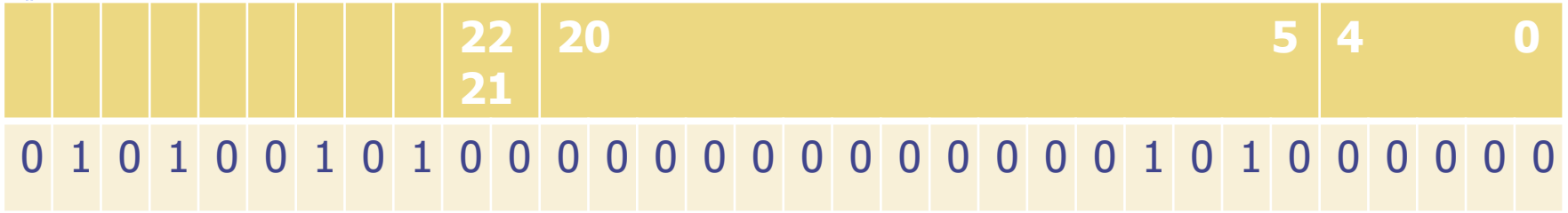
华为鲲鹏处理器 ARM based CPU

華中科技大學

```
(gdb) disass /rs
Dump of assembler code for function main:
test.c:
3       {
   0x0000000000400624 <+0>:     fd 7b be a9     stp     x29, x30, [sp, #-32]!
   0x0000000000400628 <+4>:     fd 03 00 91     mov     x29, sp

4           int x=10;
=> 0x000000000040062c <+8>:     40 01 80 52     mov     w0, #0xa
   0x0000000000400630 <+12>:    a0 1f 00 b9     str     w0, [x29, #28]

5           int y=20;
   0x0000000000400634 <+16>:    80 02 80 52     mov     w0, #0x14
   0x0000000000400638 <+20>:    a0 1b 00 b9     str     w0, [x29, #24]

6           int z;
7           z=2*x+3*y;
   0x000000000040063c <+24>:    a0 1f 40 b9     ldr     w0, [x29, #28]
   0x0000000000400640 <+28>:    02 78 1f 53     lsl     w2, w0, #1
   0x0000000000400644 <+32>:    a1 1b 40 b9     ldr     w1, [x29, #24]
   0x0000000000400648 <+36>:    e0 03 01 2a     mov     w0, w1
   0x000000000040064c <+40>:    00 78 1f 53     lsl     w0, w0, #1
--Type <RET> for more, q to quit, c to continue without paging--
```

MOV W0, #0xa 40 01 80 52





5 2 8 0 0 1 4 0

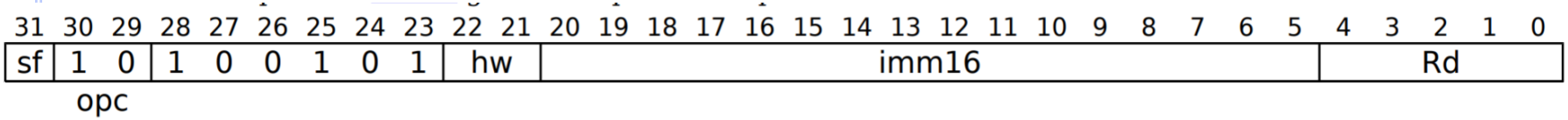
指令 MOV Rd, #imm 立即数imm → Rd

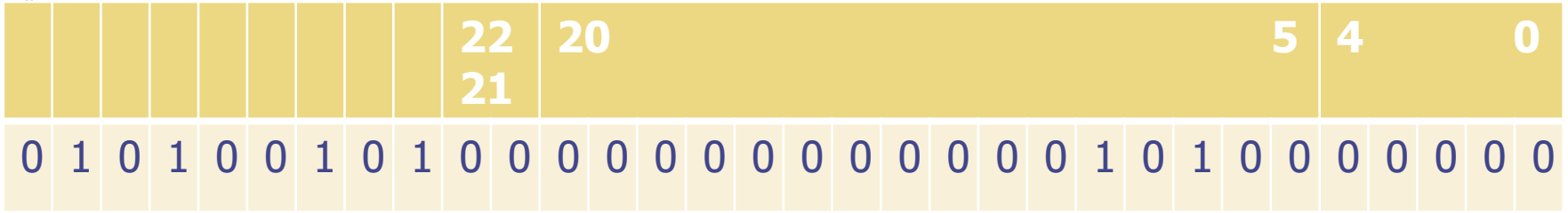
使用 32位寄存器， sf=0

16位的立即数 0xa

Rd 用 5位二进制编码，为 0，表明为 w0或x0, 由sf=0, 确定为 w0

MOV W0, #0x14　　80 02 80 52





5　　2　　8　　0　　0　　1　　8　　0

指令　MOV Rd，#imm　　立即数imm → Rd

使用 32位寄存器，sf=0

16位的立即数　0x14　　0…00010100

Rd 用 5位二进制编码，为 0，表明为 w0或x0, 由sf=0, 确定为 w0

str W0, [x29,#28] a0 1f 00 b9

									22 21	20								5	4				0
1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	1	0	0	0	0	0

b 9 0 0 1 f a 0

华为鲲鹏处理器 ARM based CPU


```
int  x = 10;
    mov  w0, #0xa
    str    w0, [x29, #28]
```

- 源操作数是立即数（立即寻址），以“#”为前缀，0x表示16进制
- W0 寄存器寻址，32位寄存器，x0的低32位
- 变量 x 的地址表达 [x29, #28]
- x29：帧指针寄存器（FP）
- &x = 0xfffffffffffffa0c = x29 + 0x1c (28)
- STR： 将左边寄存器的值 存放右边地址指定的单元中
  Store Register

```
(gdb) info reg x29
x29            0xffffffff9f0       281474976709104
(gdb) print &x
$2 = (int *) 0xffffffffa0c
```

31-28	27-25	24-21	20	19-16	15-12	11-0
Cond	001	opcode	s	Rn	Rd	Shifter_operand

- 源操作数是立即数（立即寻址），以“#”为前缀，0x表示16进制
- W0 寄存器寻址，32位寄存器，x0的低32位

华为鲲鹏处理器 ARM based CPU


```
int  y = 20;
    mov  w0, #0x14
    str    w0, [x29, #24]
    &x = x29 + 0x1c  (#28)
    &y = x29 + 0x18  (#24)
```

```
(gdb) print &x
$3 = (int *) 0xffffffffa0c
(gdb) print &y
$4 = (int *) 0xffffffffa08
(gdb) i reg x29
x29            0xffffffff9f0       281474976709104
```

华为鲲鹏处理器 ARM based CPU


```
z= 2*x + 3*y;
ldr w0, [x29, #28]      ; LoaD Register     x ->  w0
lsl  w2,w0, #1          ; Logical Shift Left   w0左移1位 -> w2
                        ;  w2 = 2*x
ldr  w1, [x29, #24]
mov w0, w1              ;  w0 = w1 =y;
lsl  w0, w0, #1         ;  w0 左移 1位 -> w0, 即  w0= 2*y
add w0, w0, w1          ;  w0 + w1 -> w0 ;  w0= 3*y
add w0, w2, w0          ;  w0 + w2 -> w0 ;  w0 = 2*x + 3*y
str   w0, [x29, #20]  ;  Store   Register  w0 -> z
```

```
        bl   label1
label2 :
```

bl 指令跳转到 label1 处执行，以后遇到第一个ret,就会返回到bl的下一条指令（label2处）继续执行。

函数调用时，一般会使用 该指令

```
B   跳转指令          //   相当于  jmp
BL 带返回的跳转指令   // 相当于 call
B.条件  用在cmp比较后，条件成立时跳转，否则不跳转。
    b.lt   label1   // 小于时转移，类似的有 eq; ne;gt;ge;lt;le
```

Adrp 指令是给寄存器赋值.

赋值的规则是:

先把pc寄存器里的数值先按照16进制表示,后3位清零,

再把adrp 右边的立即数,左移3位,也就是在末尾+3个0 .

然后让2个结果相加.

例： pc = 0x0000000104ff6754

把pc后3位清零 得到0x0000000104ff6000

adrp x8,1

右边的立即数是1,左移3位,得到 0x1000

0x0000000104ff6000+0x1000 = 0x0000000104ff7000

把0x0000000104ff7000 赋值给 x8

华为鲲鹏处理器 ARM based CPU

add x0, x1, #n ; (x1) +立即数 n → x0.

add x0, x1, x2 ; (x1) + (x2)→ x0.

add x0,x1, x2, lsl #n ; (x1) + ((x2) << n) → x0.

移位运算 与 加法的混合

add 寄存器 + 位扩展操作

add w0, w1, w2, sxtb

(w1) + ((w2) 的低8位扩展) → w0

sxtb 将一个寄存器的值取低八位进行 带符号扩展 为寄存器的位数; uxtb(低八位无符号扩展)、sxth（低十六位带符号扩展）、uxth（低十六位无符号扩展）等

多字节加载和存储指令ldp和stp

ldp x1, x2, [x0] ;

从 [x0] 处取一个值 , 送给 x1

从 [x0+8] 处取一个值 , 送给 x2

stp x1, x2, [x0] ;

(x1) 存储到 [x0] 处

(x2) 存储到 [x0+8] 处

华为鲲鹏处理器 ARM based CPU

多字节加载和存储指令ldp和stp

```
4           int a[10]={1,2,3,4,50,60,70,800,900,1000};
   0x0000000000400628 <+4>:     adrp    x0, 0x400000
   0x000000000040062c <+8>:     add     x1, x0, #0x790
   0x0000000000400630 <+12>:    add     x0, sp, #0x30
   0x0000000000400634 <+16>:    ldp     x2, x3, [x1]
   0x0000000000400638 <+20>:    stp     x2, x3, [x0]
   0x000000000040063c <+24>:    ldp     x2, x3, [x1, #16]
   0x0000000000400640 <+28>:    stp     x2, x3, [x0, #16]
   0x0000000000400644 <+32>:    ldr     x1, [x1, #32]
   0x0000000000400648 <+36>:    str     x1, [x0, #32]
```

一次加载、存储 两个 64位数，即 4个 int。

更多资源

ARM汇编语言官方手册（中文）

ARM模拟上机环境 QEMU

GCC、GDB等