ARM64和X86差异点

版本：v1.0

HUAWEI

前言

- 本文的重点是讲述ARM64和X86架构的差异点
- 前6个都是跨架构移植中需要注意的差异点
- 后3个差异点重点讲述寄存器集合、状态寄存器、指令集。

目录

char数据类型的符号 — signed

- **ARM64 和x86的char变量默认符号不一致**
  - ARM64向x86看齐，通过编译选项改为signed。
  - Makefile中通过“-fsinged-char”指定ARM64下的char为符号数，
  - 内核态的所有ko编译时继承makefile选项，都得到保证，用户态需要在一个统一的地方添加“-fsinged-char”编译选项，以免模块遗漏。
- 参考：http://3ms.huawei.com/hi/blog/18647_1331799.html?h=h

gcc\cpu	x86	armv8	mips32	e500
gcc 3.9.2	Signed	Unsigned	Signed	Unsigned

字节对齐

- **ARM64和x86在字节对齐上有些不一致**
  - ARM64支持非对齐访问，需要硬件使能，另外特殊指令要求字节对齐，如“ldaxr”要求4或者8字节对齐。
  - 使用此类指令的spinlock，原子操作和互斥量等需要保证入参的地址对齐，一些结构体的强制对齐要修改为默认对齐。如果锁结构成员在结构体中，在不影响兼容性的情况下，可以调整成员变量的偏移位置，保证锁结构按照4字节或者8字节对齐。
- 参考：http://3ms.huawei.com/hi/blog/18647_1745643.html?h=h

嵌入式汇编

- **ARM64和X86的嵌入式汇编语法基本一致**

  ARM64的汇编语言与x86完全不同，需要推翻重写。

  模板： __asm__ __volatile__ (汇编语句: 输出: 输入: 破坏描述部分)

  1、共四个部分：汇编语句模板，输出部分，输入部分，破坏描述部分

  2、各部分使用“:”格开

  3、汇编语句模板必不可少，其他三部分可选，如果使用了后面的部分，而前面部分为空，也需要用":"格开，相应部分内容为空

  4、举例： __asm__ __volatile__ (“" str %1, [%0]\n” :

  : "r" (&lock->lock), "r" (0)

  : "cc");


ARM GCC内联汇编手册


GCC嵌入式汇编培训材料

SIMD/NEON – kernel_fpu_begin/end

- **ARM64和x86都实现了SIMD，(单指令多数据)**
  - ARM64的SIMD和X86的sse和mmx实现上有差异，但是都共用浮点寄存器，内核态编程时需要考虑这些寄存器的现场保护。
  - 在浮点或者SIMD寄存器使用前调用kernel_fpu_begin，使用后，调用kernel_fpu_end。

加速引擎 -- RDE

- **ARM64和x86在加速引擎实现不一致**
  - ARM64使用一个单独的硬件计算单元，来计算DIF和RAID。DIF的CRC-16和RAID的P、PQ运算是计算密集型业务，可以使用专用的硬件逻辑加速。为了使用加速引擎的计算能力，CPU需要调用RDE驱动与RDE加速引擎通信。

- **ARM64和x86调用模式**
  - ARM64提供同步和异步两种模式，来使用RDE加速引擎驱动API接口。由于Hi1610有16个核，而加速引擎只有一个。在RDE加速引擎负载比较高的情况下，同步模式由于等待时延比较大而不可控。ARM64采用异步模式，而X86采用同步模式，两者差距比较大的地方。

延时

- **ARM64和x86在core频率和总线频率上不一致**
  - ARM64和x86的单周期指令执行时间并不同，定时器的参考时钟也不同。当前x86的HZ定义为250，也就是每个jiffer为4ms；arm的HZ定义为100，也就是每个jiffer为10ms；

寄存器集合 (1)

Low 32-bits	ARM Register	Conventional use	X86 Register	Low 32-bits	Low 16-bits	Low 8-bits
W0	X0	Return value, callee-owned	%rax	%eax	%ax	%al
W0	X0	1st argument, callee-owned	%rdi	%edi	%di	%dl
W1	X1	2nd argument, callee-owned	%rsi	%esi	%si	%sl
W2	X2	3rd argument, callee-owned	%rdx	%edx	%dx	%dl
W3	X3	4th argument, callee-owned	%rcx	%ecx	%cx	%cl
W4	X4	5th argument, callee-owned	%r8	%r8d	%r8w	%r8b
W5	X5	6th argument, callee-owned	%r9	%r9d	%r9w	%r9b
W6	X6	7th argument, callee-owned				
W7	X7	8th argument, callee-owned				
W16	X16	Scratch/temporary, callee-owned	%r10	%r10d	%r10w	%r10b
W17	X17	Scratch/temporary, callee-owned	%r11	%r11d	%r11w	%r11b
WSP	SP	Stack pointer, caller-owned	%rsp	%esp	%sp	%spl
W19	X19	Local variable, caller-owned	%rbx	%ebx	%bx	%bl
W20	X20	Local variable, caller-owned	%rbp	%ebp	%bp	%bpl
W21	X21	Local variable, caller-owned	%r12	%r12d	%r12w	%r12b
W22	X22	Local variable, caller-owned	%r13	%r13d	%r13w	%r13b
W23	X23	Local variable, caller-owned	%r14	%r14d	%r14w	%r14b
W24	X24	Local variable, caller-owned	%r15	%r15d	%r15w	%r15b

寄存器集合 (2)

Low 32-bits	ARM Register	Conventional use	X86 Register
W25	X25	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
W26	X26	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
W27	X27	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
W28	X28	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
	PC(不可直接访问)	Instruction pointer	%rip（可用于地址访问）
	%CPSR	Status/condition code bits	%eflags
W9...W18	X9...X18	Temporary registers (临时寄存器，子函数内可使用)	
	LR(r30)	The Link Register.	
	FP(r29)	The Frame Pointer	
WZR	XZR	Zero-register(read only)	
		Code Segment	CS
		Default Data Segment	DS
		Stack Segment	SS
		Extra Data Segment	ES
		Additional Data Segment	FS
		Additional Data Segment	GS

状态寄存器

AArch64	Flag Name	含义	X86
CF	Carry Flag	进位/借位Carry。如果算术指令的结果的最高有效位产生了进位/借位，则置1；否则置0。这个标志位指示了无符号整数计算的结果是否溢出	CF
	Parity Flag	奇偶校验位Parity。如果指令结果的最低字节含有偶数个比特1，则置1；否则置0。	PF
	Auxiliary Flag	辅助进位/借位Auxiliary Carry。如果算术运算在第3比特位（最低比特位是0）产生了进位/借位，则置1；否则置0。这个标志主要用于二进制编码的十进制数BCD算术运算。	AF
ZF	Zero Flag	零标志位Zero。如果指令结果为0，则置1；否则置0	ZF
NF(negative)	Sign Flag	符号位Sign。设置为指令结果的最高有效比特位，即有符号整数的符号比特位。0表示结果是正数或者是0；1表示负数。	SF
VF	Overflow Flag	溢出位Overflow。如果整数运算结果超出了目标操作数可容纳的值域（正数过大，负数过小，不包括符号位），则置1；否则置0（即结果可信）。这个标志指示了有符号整数算术运算的结果是否溢出。	OF
	DF Direction Flag		

指令集 (1)

- 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体系结构是可变长度指令集体系结构。指令地址没有对齐要求。
  - 主要差异列举如下：
    - ARM是一种load-store架构：在RISC体系结构中很常见，这意味着数据处理指令不能直接对内存的内容进行操作，它们仅对寄存器进行操作。 反过来，加载和存储指令只能在寄存器和内存之间传输数据。 相比之下，IA-32指令集的数据处理指令可以直接在内存以及寄存器上处理数据。
    - IA-32支持访问I/O地址空间的单独I/O指令：X86的指令集包括一部分指令可以直接对IO地址空间进行操作。例如IN或者OUT，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。

指令集 (2)

- 无法在ARM指令中嵌入任意32位地址
  - 因为ARM指令是固定32bit宽度的，所以不可能直接在arm指令中编码进32bit的地址。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。换句话说，所有ARM的内存访问都是通过通用寄存器间接进行的。由于指令集的可变长度性质，在IA-32中可以在指令中嵌入32位地址。X86可以进行直接地址访问。
- ARM指令不能包含任意的32位常量（ ARM不能直接生成32bit宽度的立即数 ）。
- 同上的原因。ARM操作长度较长的立即数时，需要通过mov/movk 多次进行生成。
- IA-32使用分段寻址模型
  - 所有IA-32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)

指令集 (3)

- ARM指令通常都有3个操作数；x86 指令存在隐含操作数（ cpuid mul ）
  - 大多数ARM数据处理指令采用三个操作数，所有这些操作数可以是寄存器，其中之一可以是常量。 相反，IA-32指令通常具有只有两个操作数，从而使它们本质上具有破坏性（ 即，其中一个操作数被结果覆盖 ）。 这使ARM在指令级别上更加灵活。
  - 例如，可以将两个寄存器加在一起并将结果放在第三个寄存器中使用一条指令。相同的操作在IA-32上需要两条指令。

IA-32	ARM
`mov cx, ax    ; ca = ax` `add cx, bx    ; cx = cx + bx`	`add r0, r1, r2    ; r0 = r1 + r2`

指令集 (4)

- 许多复杂的IA-32指令都没有与ARM直接对应的指令
  - 因为x86的芯片电路相对设计复杂，可以支持较多复杂操作的指令。例如BCD转换和操作，点积，三角函数和对数函数等。
- ARM子例程通常不使用stack
  - ARM BL/B/B.cond 指令（用于子例程调用），是将返回地址防止在链接寄存器LR, 而不是把地址放到栈里。同样，子例程返回之后的返回地址，是从LR里返回，而不是存储在栈上的。
  - X86 call 和Ret 指令，将返回地址放在堆栈上并分别取出。此外，提供了ENTER和LEAVE指令，以在进入和离开过程时创建和释放固定格式的堆栈帧。ARM没有与此等效的方法，因为从堆栈指针中添加固定值或从中减去固定值非常简单。

指令集 (5)

- IA-32在被零除错误时产生异常
  - ARM UDIV和SDIV指令不会检测被零除的情况，并且始终返回零结果。(编译器高级别优化如-O2可能会检测到除零的情况，才生成brk指令，运行时报错)
- ARM不提供数组绑定检查指令
  - 在IA-32中，BOUND指令用于在执行潜在的无效访问之前针对数组边界检查内存地址。 ARM指令集无等效指令。

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and organization for a fully connected, intelligent world.

HUAWEI