虚拟地址位宽：

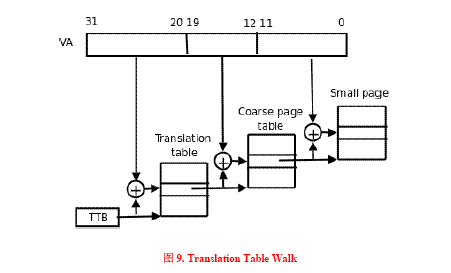
Kernel/arch/arm64/configs/xxxx\_defconfig文件， CONFIG\_ARM64\_VA\_BITS\_xx=y 决定。Sharkl3定义为39bit。

TTB(Translation Table Base Register)

TLB(Translation Lookaside Buffer)，减少translation 延迟和power consumption。TLB hit可以减少一次translation table walk。

ARM920T为例：

 VA到PA的映射过程



首先将CPU内核发送过来的32位VA[31:0]分成三段，前两段VA[31:20]和VA[19:12]作为两次查表的索引，第三段VA[11:0]作为页内的偏移，查表的步骤如下：

⑴从协处理器CP15的寄存器2(TTB寄存器，translation table base register)中取出保存在其中的第一级页表(translation table)的基地址，这个基地址指的是PA，也就是说页表是直接按照这个地址保存在物理内存中的。

⑵以TTB中的内容为基地址，以VA[31:20]为索引值在一级页表中查找出一项(2^12=4096项)，这个页表项(也称为一个描述符，descriptor)保存着第二级页表(coarse page table)的基地址，这同样是物理地址，也就是说第二级页表也是直接按这个地址存储在物理内存中的。

⑶以VA[19:12]为索引值在第二级页表中查出一项(2^8=256)，这个表项中就保存着物理页面的基地址，我们知道虚拟内存管理是以页为单位的，一个虚拟内存的页映射到一个物理内存的页框，从这里就可以得到印证，因为查表是以页为单位来查的。

⑷有了物理页面的基地址之后，加上VA[11:0]这个偏移量(2^12=4KB)就可以取出相应地址上的数据了。

这个过程称为Translation Table Walk，Walk这个词用得非常形象。从TTB走到一级页表，又走到二级页表，又走到物理页面，一次寻址其实是三次访问物理内存。注意这个“走”的过程完全是硬件做的，每次CPU寻址时MMU就自动完成以上四步，不需要编写指令指示MMU去做，前提是操作系统要维护页表项的正确性，每次分配内存时填写相应的页表项，每次释放内存时清除相应的页表项，在必要的时候分配或释放整个页表。

ARM64 提供多种Memory Layout。

Virtual address bit39：4KB gapes+3 levels.

Virtual address bit42：64KB gapes+2 levels.

Virtual address bit48：4KB gapes+4 levels or 64KB gapes+3 levels.

AArch64 Linux memory layout with 4KB pages + 3 levels:

Start End Size Use

-----------------------------------------------------------------------

0000000000000000 0000007fffffffff 512GB user

ffffff8000000000 ffffffffffffffff 512GB kernel

AArch64 Linux memory layout with 4KB pages + 4 levels:

Start End Size Use

-----------------------------------------------------------------------

0000000000000000 0000ffffffffffff 256TB user

ffff000000000000 ffffffffffffffff 256TB kernel

AArch64 Linux memory layout with 64KB pages + 2 levels:

Start End Size Use

-----------------------------------------------------------------------

0000000000000000 000003ffffffffff 4TB user

fffffc0000000000 ffffffffffffffff 4TB kernel

AArch64 Linux memory layout with 64KB pages + 3 levels:

Start End Size Use

-----------------------------------------------------------------------

0000000000000000 0000ffffffffffff 256TB user

ffff000000000000 ffffffffffffffff 256TB kernel

For details of the virtual kernel memory layout please see the kernel

booting log.

Translation table lookup with 4KB pages:

+--------+--------+--------+--------+--------+--------+--------+--------+

|63 56|55 48|47 40|39 32|31 24|23 16|15 8|7 0|

+--------+--------+--------+--------+--------+--------+--------+--------+

| | | | | |

| | | | | v

| | | | | [11:0] in-page offset

| | | | +-> [20:12] L3 index

| | | +-----------> [29:21] L2 index

| | +---------------------> [38:30] L1 index

| +-------------------------------> [47:39] L0 index

+-------------------------------------------------> [63] TTBR0/1

Translation table lookup with 64KB pages:

+--------+--------+--------+--------+--------+--------+--------+--------+

|63 56|55 48|47 40|39 32|31 24|23 16|15 8|7 0|

+--------+--------+--------+--------+--------+--------+--------+--------+

| | | | |

| | | | v

| | | | [15:0] in-page offset

| | | +----------> [28:16] L3 index

| | +--------------------------> [41:29] L2 index

| +-------------------------------> [47:42] L1 index

+-------------------------------------------------> [63] TTBR0/1

以虚拟地址位宽48bit，page size 4KB为例：

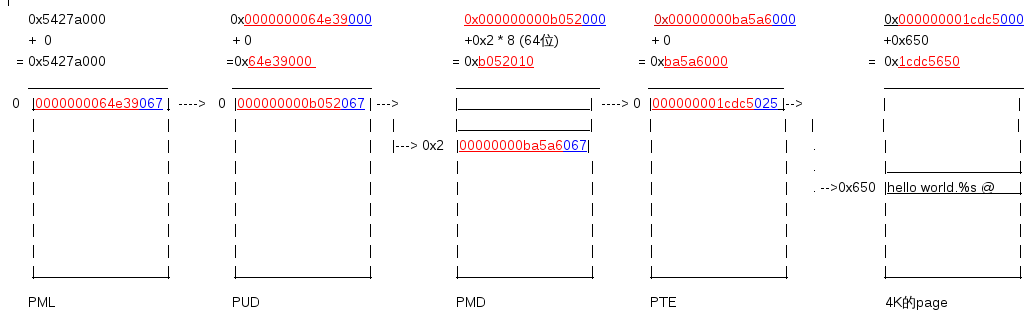
假如我们的线性地址（虚拟地址）为0x400650（例子中的地址确实是400650），得到物理地址的过程如下：

**每级页表（目录）索引值:**

PML中的目录项：  (0x400650>>（9+ 9 + 9 + 12）) & 0x1ff  = 0  
PUD中的目录项： (0x400650>>（9 + 9 + 12）) & 0x1ff  = 0  
PMD中的目录项： (0x400650>>（9 + 12）) & 0x1ff  = 0x2  
PTE中的目表项： (0x400650>>（12）) & 0x1ff  = 0

**在页中的偏移量(4K的页)**：

(0x400650>>（0）) & 0xfff  = 0x650



用crash工具查看translation table walk过程：

**crash>** ps |grep hello  
  24458  24204   0  ffff8801573b09e0  IN   0.0    4152    632  hello  
**crash>** set 24458  
    PID: 24458  
COMMAND: "hello"  
   TASK: ffff8801573b09e0  [THREAD\_INFO: ffff880090794000]  
    CPU: 0  
  STATE: TASK\_INTERRUPTIBLE    
**crash>** px ((struct task\_struct \*)0xffff8801573b09e0)->mm->pgd  
$1 = (pgd\_t \*) 0xffff88005427a000  
**crash>** px (0xffff88005427a000 - 0xffff880000000000)  
$2 = 0x5427a000                                           # 相当CR3中的内容， PML的首地址  
**crash>** px $2 + 0x0                                        #  PML的第0项  
$3 = 0x5427a000  
**crash>** rd -p 0x5427a000                             #  
        5427a000:  0000000064e39067                    g..d....  
**crash>** pte 0000000064e39067  
  PTE     PHYSICAL  FLAGS  
64e39067  64e39000  (PRESENT|RW|USER|ACCESSED|DIRTY)         # 64e39000 PUD的首地址  
**crash>** px (0x400650>>30) & 0x1ff  
$4 = 0x0

**crash>** px 0x64e39000 + 0x0                                                      #  PUD 的第0项  
$5 = 0x64e39000  
**crash>** rd -p 0x64e39000  
        64e39000:  000000000b052067                    g ......  
**crash>** pte 000000000b052067  
  PTE    PHYSICAL  FLAGS  
b052067   b052000  (PRESENT|RW|USER|ACCESSED|DIRTY)       # b052000 PMD的首地址  
**crash>** px (0x400650>>21) & 0x1ff  
$6 = 0x2  
**crash>** px 0xb052000 + 0x2 \* 8                                                  #  PMD 的第2项  
$7 = 0xb052010  
**crash>** rd -p 0xb052010  
         b052010:  00000000ba5a6067                    g`Z.....  
**crash>** pte 00000000ba5a6067  
  PTE     PHYSICAL  FLAGS  
ba5a6067  ba5a6000  (PRESENT|RW|USER|ACCESSED|DIRTY)    # ba5a6000 PTE的首地址  
**crash>** px (0x400650>>12) & 0x1ff   
$8 = 0x0  
**crash>** px 0xba5a6000 + 0x0                                  #  PTE 的第0项  
$9 = 0xba5a6000  
**crash>** rd -p 0xba5a6000  
        ba5a6000:  000000001cdc5025                    %P......  
**crash>** pte 000000001cdc5025  
  PTE     PHYSICAL  FLAGS  
1cdc5025  1cdc5000  (PRESENT|USER|ACCESSED)            # 进程， 线性页对应的 真正的物理页框地址  
**crash>** px 0x400650 & 0xfff  
$10 = 0x650  
**crash>** px 0x1cdc5000 + 0x650                                            # “hello world.%s” 所在 的物理地址。  
$11 = 0x1cdc5650  
**crash>** rd -p 0x1cdc5650 2  
        1cdc5650:  6f77206f6c6c6568 4020732500646c72   hello world.%s @  
**crash>** vtop 0x400650        # 以上步骤可以一步搞定。  
VIRTUAL     PHYSICAL          
400650      1cdc5650          
  
   PML: 5427a000 => 64e39067  
   PUD: 64e39000 => b052067  
   PMD: b052010 => ba5a6067  
   PTE: ba5a6000 => 1cdc5025  
  PAGE: 1cdc5000

PTE     PHYSICAL  FLAGS  
1cdc5025  1cdc5000  (PRESENT|USER|ACCESSED)  
  
      VMA           START       END     FLAGS FILE  
ffff8800541f2228     400000     401000 8000875 /home/shhfeng/work/workdir/ccode/hello  
  
      PAGE        PHYSICAL      MAPPING       INDEX CNT FLAGS  
ffffea0000737140  1cdc5000 ffff88003fc40200        0  2 3ffff800020068

Speadtrum MMU 设计:

共一级页表，page size 4KB。映射的虚拟地址访问空间访问64M。Page table size 16k。MMU Iram size 16k\*(34-12)(物理地址位宽-页位宽)。

软件配置Iram中的数据，或操作DMA配置Iram中数据。

TLB 设计：根据理论推算，连续访问越多，TLB可以越少。根据仿真找到合适的TLB size，伪随机访问。

Cortex-A7\_MPCore MMU design:

TLB page-walk mechanism 支持page descriptors缓存在L1 data cache。