

# ARM64硬件结构与系统接口

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

<https://ipads.se.sjtu.edu.cn>

# 版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

# 回顾：操作系统考虑的一些问题

- hello 这个可执行文件存储在什么位置？是如何存储的？
- hello 这个可执行文件是如何加载到 CPU 中运行？
- hello 这个可执行文件是如何将"Hello World!"这行字输出到屏幕？
- 两个hello 程序同时运行的过程中如何在一个 CPU 中运行？
- ...

**操作系统需要：1、服务应用；2、管理应用**

# 回顾：操作系统的功能：管理

- **如何卡死一个OS?**

- 例: rogue-1.c 可以fork出无数的进程

- **如何解决这个问题？**

- 资源配额 : cgroup/Linux
  - 虚拟化 : 虚拟机
  - 万能方法 : 重启机器
  - 制度约束 : AppStore的程序预审准入机制

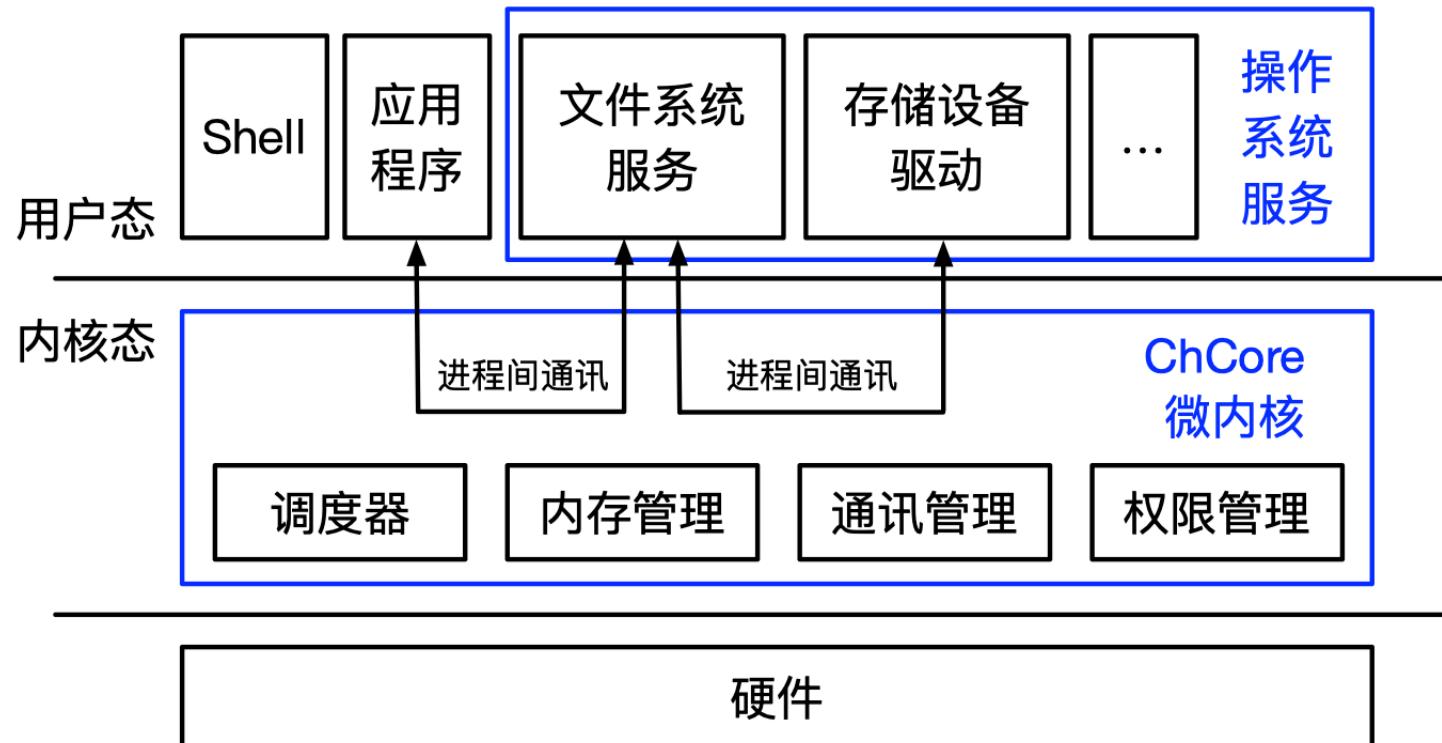
Rogue-2.c

```
int main () {  
    while (1){  
        fork(); }  
}
```

# 回顾：为什么操作系统比较难/有意思？

- 深入事情本质：直接管理硬件细节
  - 好处：实现资源的高效利用，从根本上解决问题，做一个高效程序员（降维）
  - 挑战：需要理解与处理硬件细节，硬件甚至可能出错
  - “把复杂留给自己、把简单留给用户”
    - 对比：用户态写一个Hello World和在操作系统内核中输出一个Hello World (Lab1)
- 锻炼系统架构能力
  - 将复杂问题进行抽象与化简
  - 最好的体现了M.A.L.H原则的使用
  - 计算机科学中30%的原则是从操作系统中来的 (13/41, greatprinciples.org)
- 各种问题的交互与相互影响
  - `fd = open(); ... ; fork();`

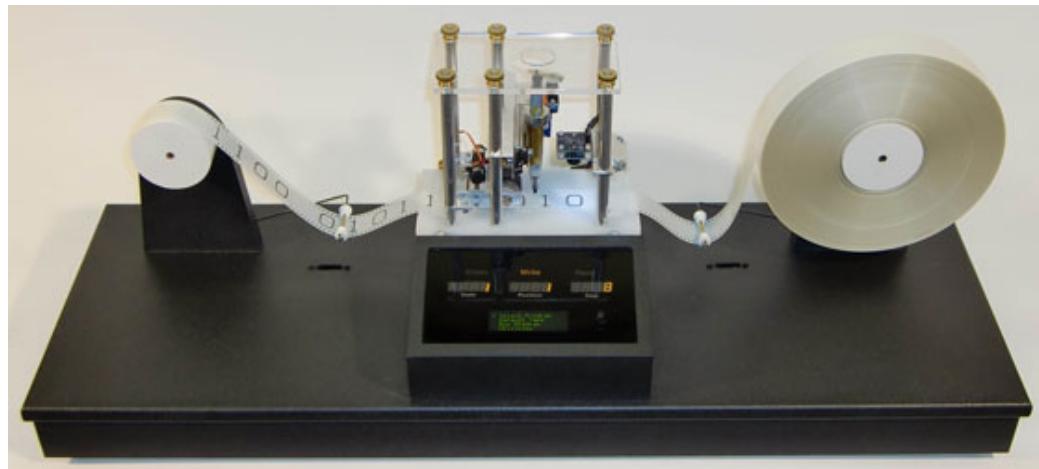
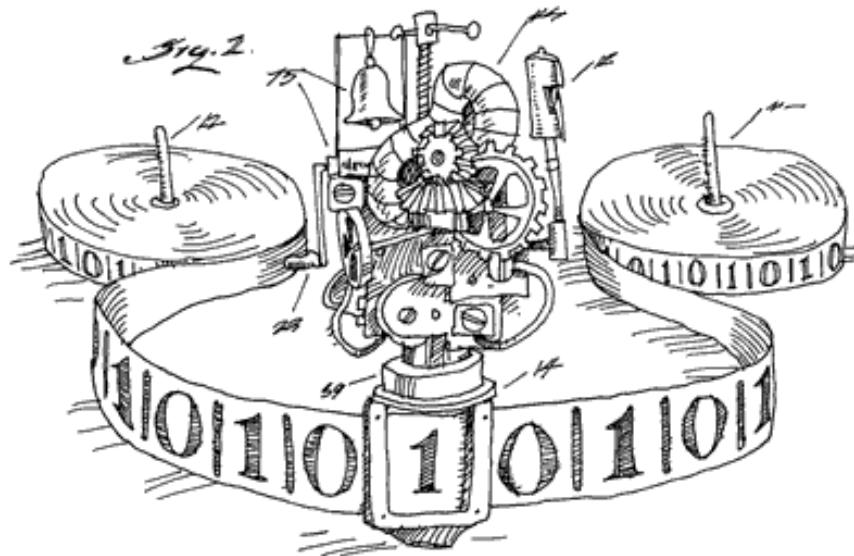
# 回顾 : ChCore : 微内核架构操作系统



# 图灵机

计算  $3+2$ : 0000001110110000

		0000001110110000
1	0	0000001110110000
2	0	0000001110110000
3	0	0000001110110000
4	0	0000001110110000
5	0	0000001110110000
6	0	0000001110110000
7	0	0000001110110000
8	1	0000001110110000
9	1	0000001110110000
10	1	0000001110110000
11	10	0000001111110000
12	10	0000001111110000
13	10	0000001111110000
14	11	0000001111110000
15	0	00000011111100000



# 冯诺依曼架构



- **输入/输出:** 和设备之间交互数据
- **CPU:** 包括处理单元和控制单元
- **存储器:** 内存

# 冯诺依曼架构

Main memory

CPU

```
for (;;) {  
    next instruction  
}
```

instruction  
instruction  
instruction  
...  
data  
data  
data



- CPU : 解析指令
- 内存 : 存储指令和数据

# 思考：冯诺依曼架构有什么局限呢？

- 1. CPU与内存的交互引起的内存墙问题
- 2. 数据与指令不区分，指令等数据或数据等指令
- 2.串行顺序处理，缺乏数据并行能力

# 大纲

- 为什么选择ARM
- 硬件体系结构
- 操作系统启动过程
- 硬件模拟



# 为什么选择ARM

# ARM: 智能手机的模式指令集



# ARM：正在走向服务器



# ARM发展

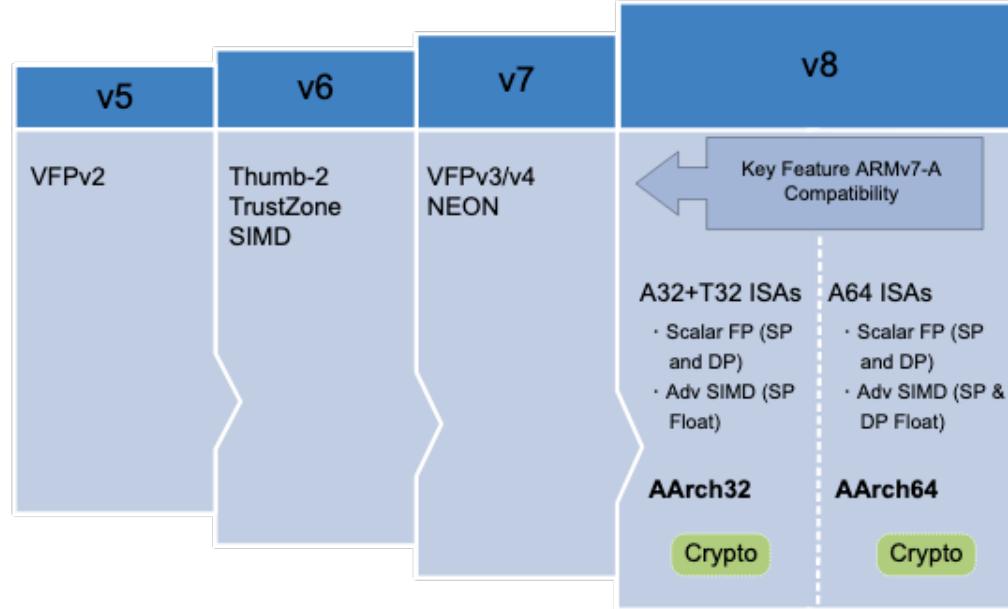
- ARMv4及之前
  - ARM 32位指令集
- ARMv4T、ARMv5TE
  - 加入Thumb 16位指令集
- ARMv6
  - 多核、SIMD、TrustZone®
  - Thumb-2 32位指令集

# ARM发展

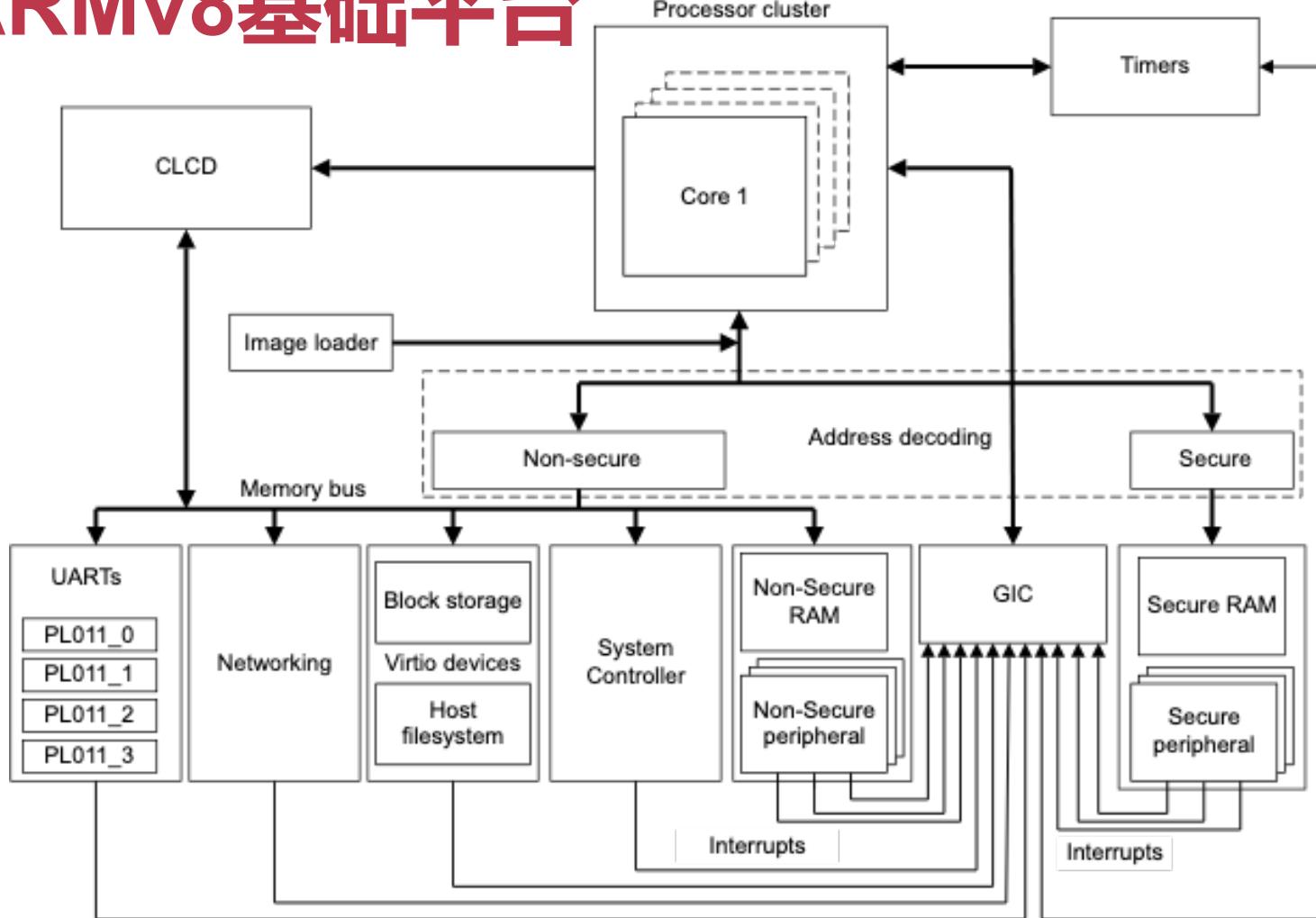
- ARMv7
  - 增强SIMD拓展——NEON
  - 全面支持平台操作系统，如Linux
  - 可预测实时高性能

# ARMv8

- 扩大物理寻址
  - 4GB以外的物理地址
- 64位虚拟地址
- 自动事件信号
  - 低功耗、高性能的自旋锁
- 硬件加速加密
- 新的异常模型



# ARMv8基础平台



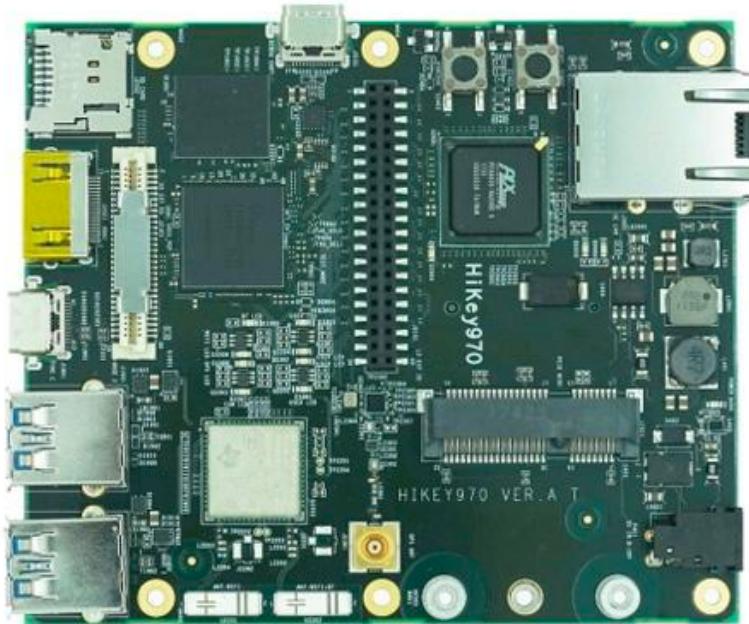
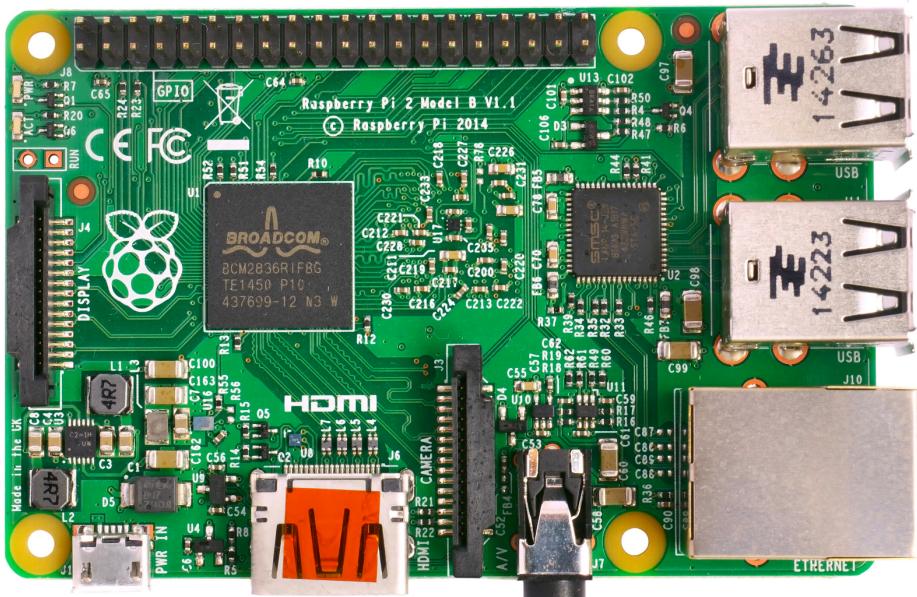


# AARCH64体系结构

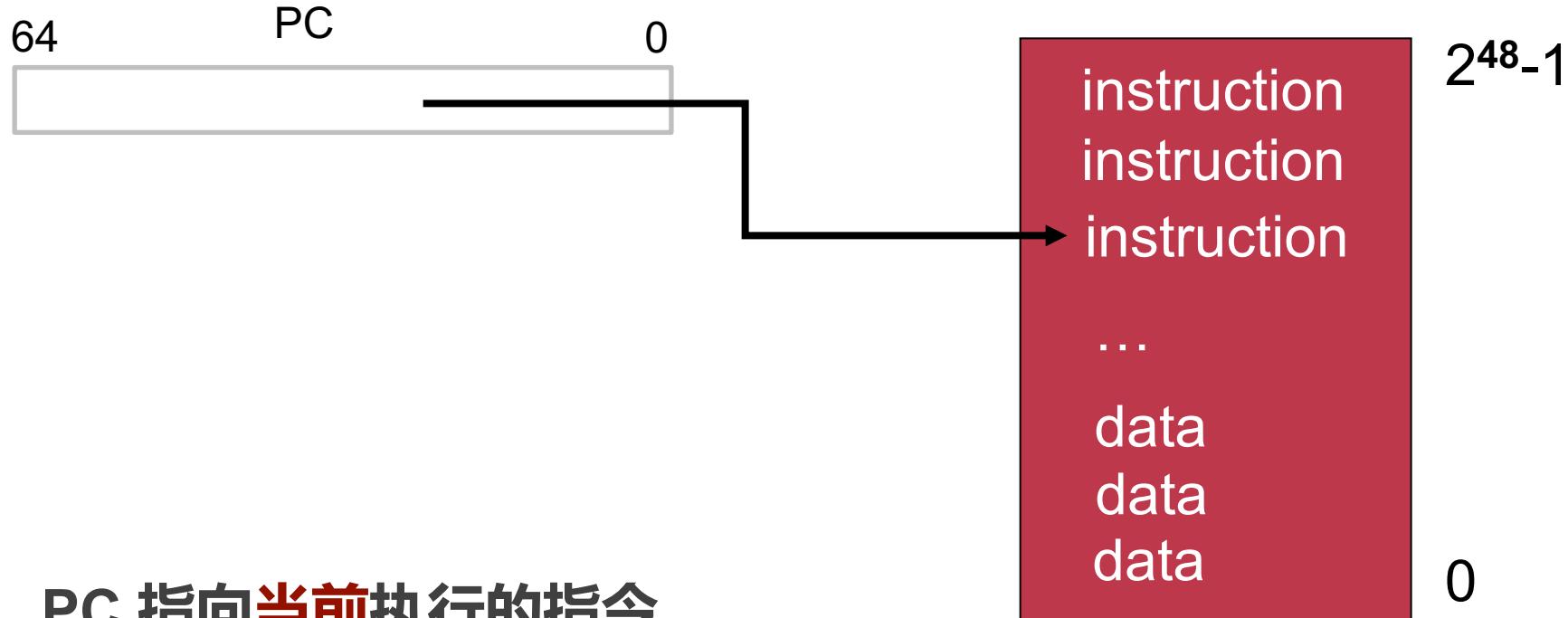
# ARMv8支持的执行模式

- Aarch64
  - 支持A64指令集
- Aarch32
  - 支持A32、T32、T16指令集

# ARM开发板



# Aarch64实现



- PC 指向**当前执行的指令**
- 指令长度相同 (RISC, 32bit)
- PC 会被跳转指令修改 : B, BL, BX, BLX

# 寄存器 ARM

- 31个64位通用寄存器
  - X0-X30
- 1个PC寄存器
- 4个栈寄存器（切换时保存SP）
  - SP\_EL0, SP\_EL1, SP\_EL2, SP\_EL3
- 3个异常链接寄存器（保存异常的返回地址）
  - ELR\_EL1, ELR\_EL2, ELR\_EL3
- 3个程序状态寄存器（切换时保存PSTATE）
  - SPSR\_EL1, SPSR\_EL2, SPSR\_EL3

# 寄存器 ARM vs X86-64

X86-64

- **31个64位通用寄存器**
    - X0-X30
  - **1个PC寄存器**
  - **4个栈寄存器（切换时保存SP）**
    - SP\_EL0, SP\_EL1, SP\_EL2, SP\_EL3
  - **3个异常链接寄存器（保存异常的返回地址）**
    - ELR\_EL1, ELR\_EL2, ELR\_EL3
  - **3个程序状态寄存器（切换时保存PSTATE）**
    - SPSR\_EL1, SPSR\_EL2, SPSR\_EL3
- 思考：X86架构中，切换特权级时rsp是如何保存，以及如何恢复的？
- |           |
|-----------|
| 16个通用寄存器  |
| 1个%rip寄存器 |
| 1个%rsp寄存器 |
| 返回地址压栈    |
| EFLAGS    |

# 指令集

- RISC
  - 固定长度指令格式
  - 更多的通用寄存器
  - Load/store 结构
  - 简化寻址方式

```
0000000000080000 <_start>:  
 80000: d53800a8      mrs    x8, mpidr_el1  
 80004: d2b82009      mov    x9, #0xc1000000  
002688  
 80008: 8a290108      bic    x8, x8, x9  
 8000c: b4000068      cbz   x8, 80018 <primary>  
  
0000000000080010 <secondary_hang>:  
 80010: 94000000      bl     80010 <secondary_hang>  
  
0000000000080014 <proc_hang>:  
 80014: 94000000      bl     80014 <proc_hang>  
  
0000000000080018 <primary>:  
 80018: 94001bfa      bl     87000 <arm64_elX_to_el1>  
 8001c: 1003a8a0      adr    x0, 87530 <boot_cpu_stack>  
 80020: 91400400      add    x0, x0, #0x1, lsl #12  
 80024: 9100001f      mov    sp, x0  
 80028: 94001ca6      bl     872c0 <init_c>  
 8002c: 17fffffa      b      80014 <proc_hang>  
 80030: 14000007      b      8004c <__start_kernel_veneer+0>
```

# RISC vs CISC

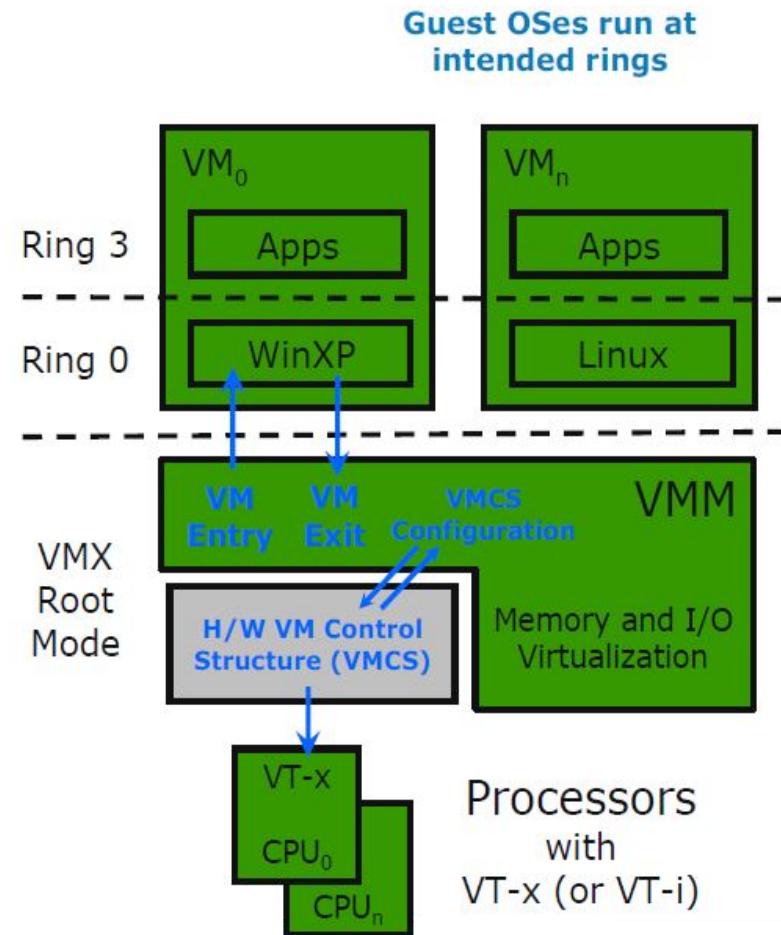
	RISC (Aarch64)	CISC (x86-64)
指令长度	定长	变长
寻址模式	寻址方式单一	多种寻址方式
内存操作	load/store	mov
实现	微码	增加通用寄存器数量
指令复杂度	简单	复杂
汇编复杂度	复杂	简单
中断响应	快	慢
功耗	低	高
处理器结构	简单	复杂

# 讨论

- RISC和CISC指令集的优劣

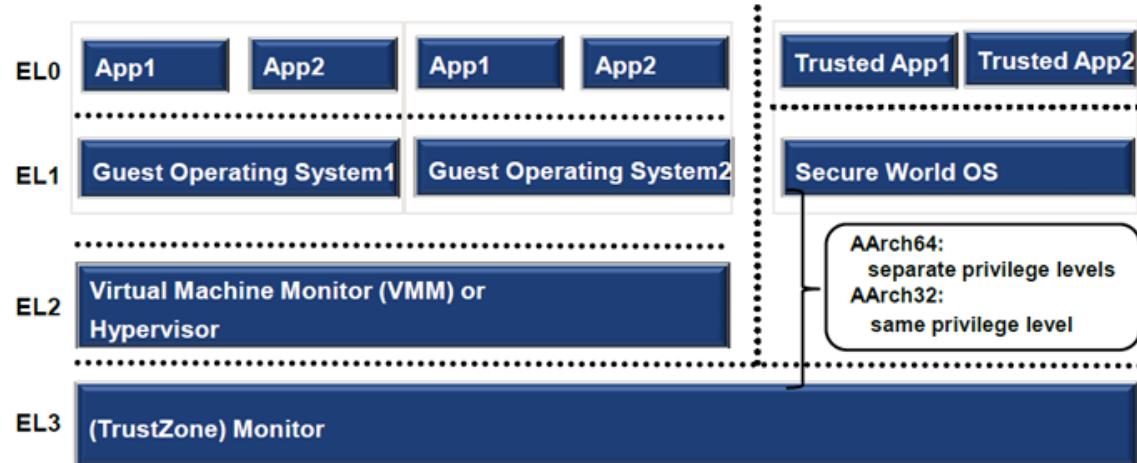
# 特权级/X86-64

- **Non-root :**
  - Ring 0 : Guest app
  - Ring 3 : Guest os
- **Root :**
  - Ring 0 : App
  - Ring 3 : Hypervisor



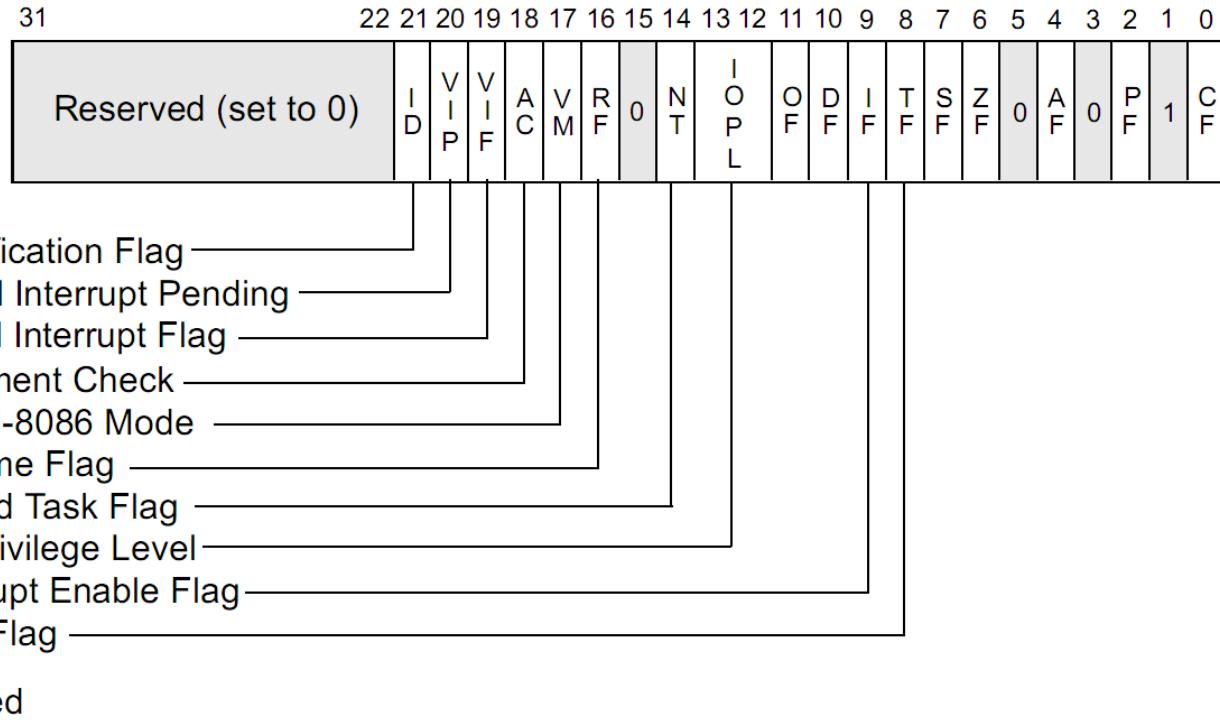
# 特权级/ARM (Exception Level)

- EL0: 用户态程序
- EL1: 内核
- EL2: hypervisor
- EL3: monitor



# 系统状态寄存器 X86-64

- System Flags in the EFLAGS

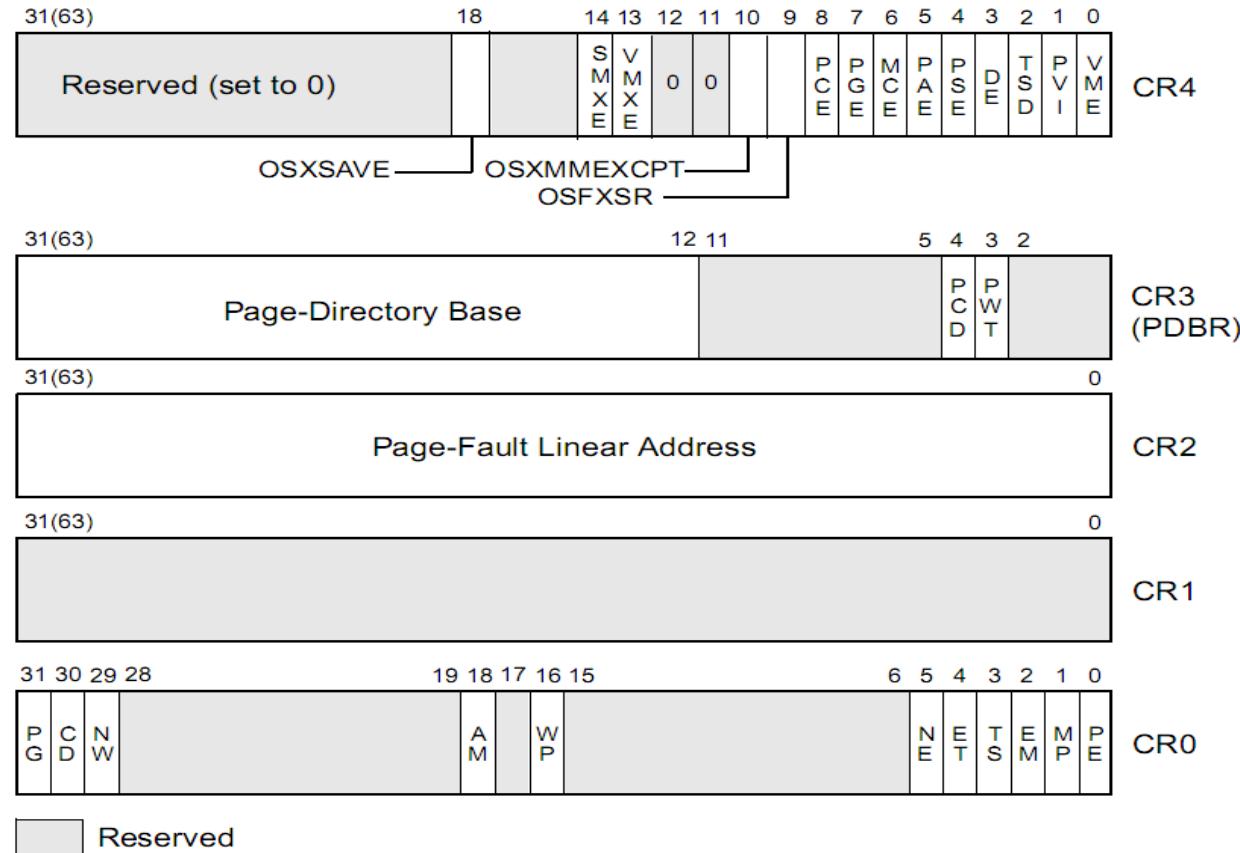


# 系统状态寄存器 ARM

- 抽象进程状态信息 ( PSTATE )
  - 条件标记 (Condition flags)
  - 执行状态 (Execution state controls)
  - 异常掩码 (Exception mask bits)
  - 访问，时钟控制 (Access control bits)

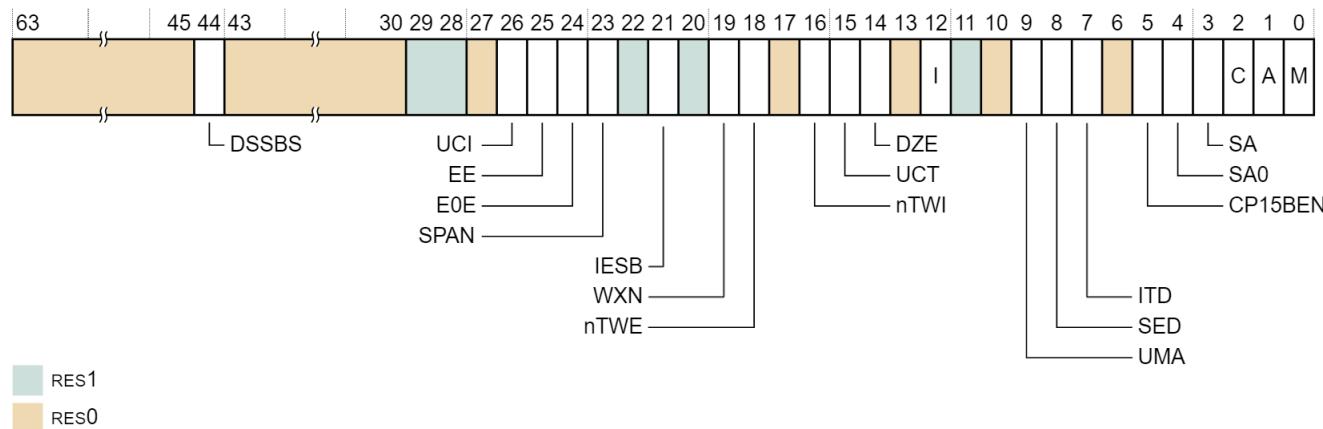
Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F
CurrentEL	EL
SPSel	SP
PAN	PAN
UAO	UAO
DIT	DIT
SSBS	SSBS

# 系统控制寄存器 X86-64



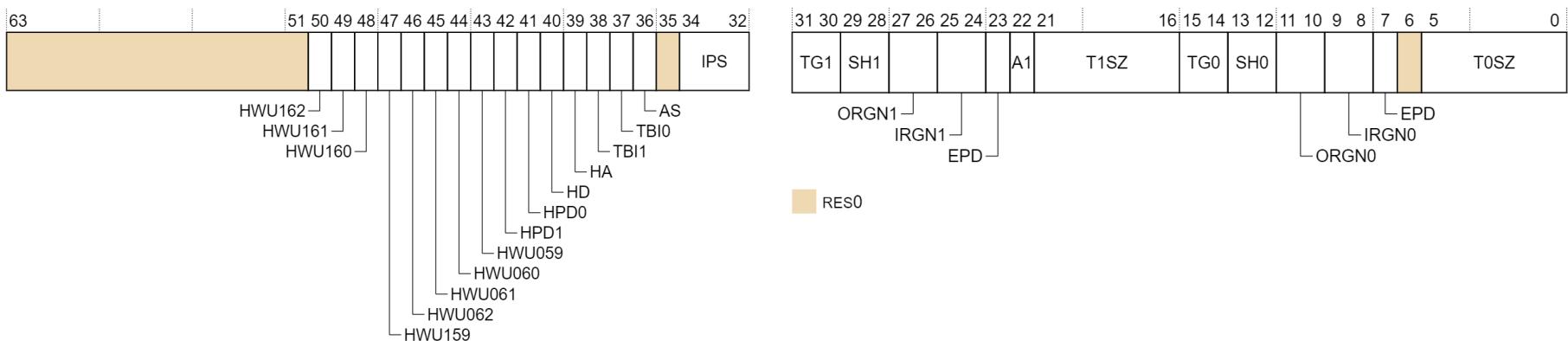
# 系统控制寄存器 (System Control Register)

- 对于系统的顶层控制
  - 大小端、使用MMU、检查Tag、内存系统



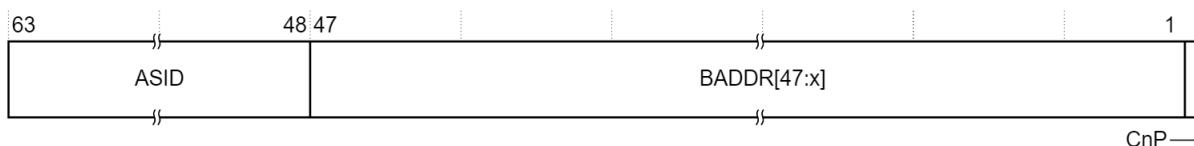
# 内存系统相关寄存器

- Translation Control Register (TCR)



- Translation Table Base Register (TTBR)

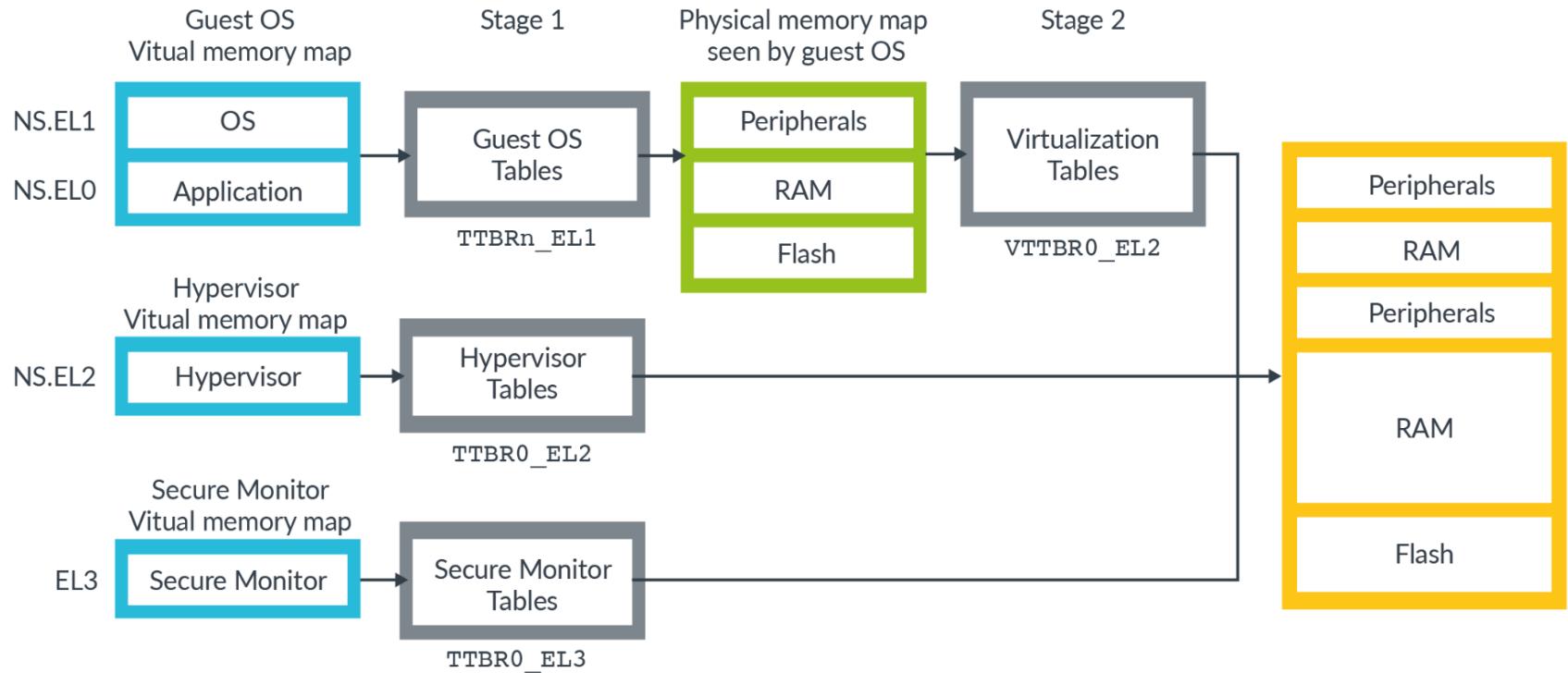
X86-64:  
CR3



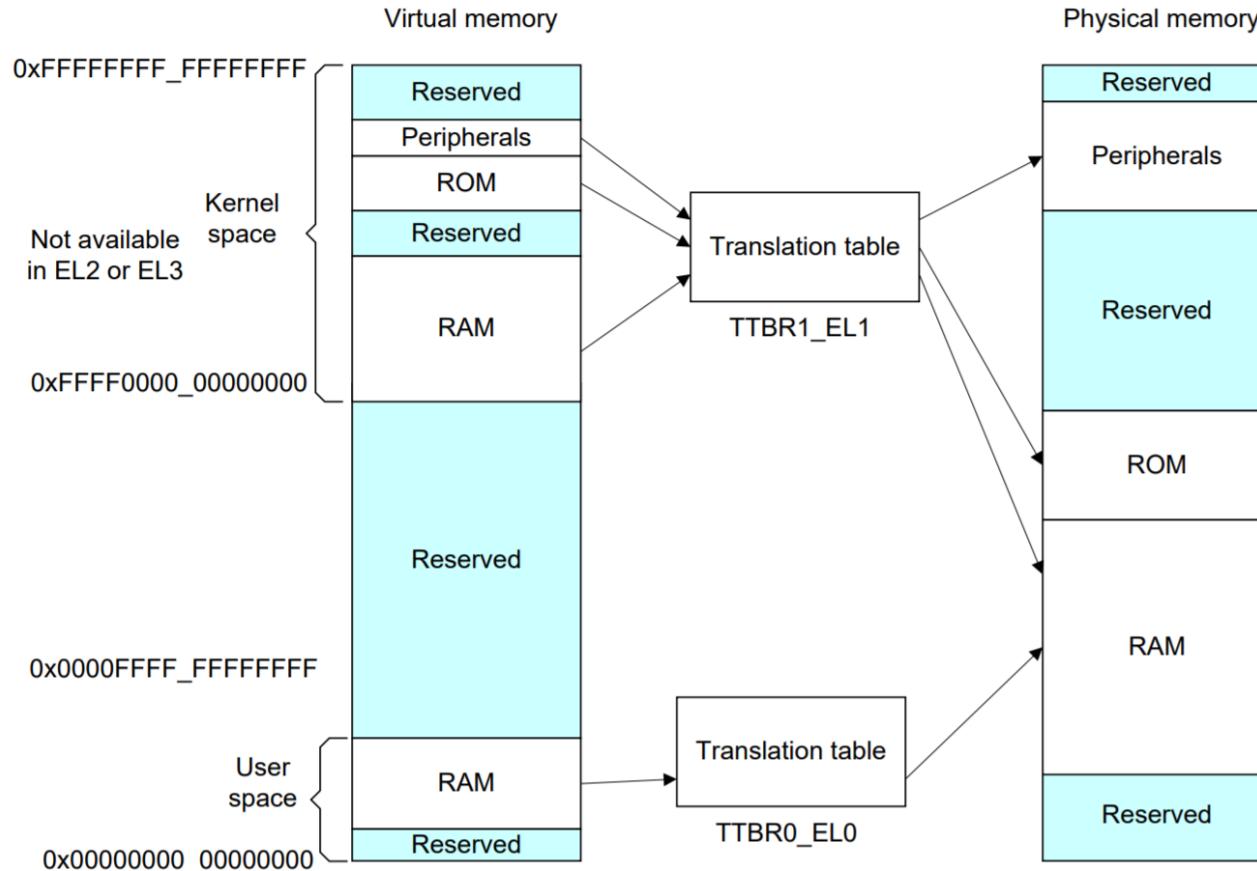
## 讨论二

- Aarch64的TTBR0支持( $0 \sim 2^{48}-1$ )的地址映射，TTBR1支持( $2^{48} \sim 2^{64}$ )的地址映射，这样的硬件设计与x86-64中的CR3相比较，能够如何协助到操作系统的设计？

# 地址翻译



# 内存空间



# ARM 输入/输出

```
unsigned int early_uart_recv(void)
{
    while (1) {
        if (early_uart_lsr() & 0x01)
            break;
    }
    return early_get32(AUX_MU_IO_REG) & 0xFF;
}
```

```
void early_uart_send(unsigned int c)
{
    while (1) {
        if (early_uart_lsr() & 0x20)
            break;
    }
    early_put32(AUX_MU_IO_REG, c);
}
```

```
BEGIN_FUNC(early_get32)
    ldr w0,[x0]
    ret
END_FUNC(early_get32)
```

- **MMIO: 复用ldr和str指令**
  - 映射到物理内存的特殊地址段

```
BEGIN_FUNC(early_put32)
    str w1,[x0]
    ret
END_FUNC(early_put32)
```

# MMIO与PIO

- **MMIO (Memory-mapped IO)**
  - 将设备映射到连续的物理内存中，使用相同的指令
  - 如，Raspi3映射到0x3F200000
  - 行为与内存不完全一样，读写会有副作用
- **PIO (Port IO)**
  - IO设备具有独立的地址空间
  - 使用特殊的指令（如x86中的in/out指令）



以ChCore为例

# 操作系统启动过程

# 设备上电后.....

- OS如何启动
  - 不同ARM设备启动OS的方式不同，例
  - Raspi 3: CPU进行初始化后从SD卡上加载
  - Hikey970: 通过BL链和UEFI加载
- OS启动时进行硬件初始化工作，并开启页表
- 进入内核

# ChCore Bootloader

- **Bootloader和kernel放在同一个ELF文件中**
  - Bootloader位于.init段，并通过链接器设置入口
  - Kernel位于.text段

```
set_property(
    TARGET kernel.img
    APPEND_STRING
    PROPERTY
        LINK_FLAGS
        "-T ${CMAKE_CURRENT_BINARY_DIR}/${link_script} -e _start"
)
```

# ChCore Bootloader

- **Bootloader和kernel放在同一个ELF文件中**
  - Bootloader位于.init段，并通过链接器设置入口
  - Kernel位于.text段
- **主CPU启动，其他次CPU等待**

```
BEGIN_FUNC(_start)
    mrs x8, mpidr_el1
    mov x9, #0xc1000000
    bic x8, x8, x9
    cbz x8, primary
```

# 层级切换

- 其他EL(>=1)到EL1

primary:

/\* Turn to el1 from other exception levels. \*/

bl arm64\_elX\_to\_el1

.Lno\_gic\_sr:

// Set EL1 to 64bit.

mov x9, HCR\_EL2\_RW  
msr hcr\_el2, x9

// Set the return address and exception level.

adr x9, .Lttarget  
msr elr\_el2, x9  
mov x9, #(SPSR\_ELX\_DAIF | SPSR\_ELX\_EL1H)  
msr spsr\_el2, x9

isb  
eret

.Lttarget:

ret

# 准备函数栈和异常向量

- 准备C函数栈
  - 设置SP寄存器
- 设置异常向量
  - 为了debug

```
primary:  
    /* Turn to el1 from other exception levels. */  
    bl  arm64_elX_to_el1  
  
    /* Prepare stack pointer and jump to C. */  
    adr  x0, boot_cpu_stack  
    add  x0, x0, #0x1000  
    mov  sp, x0  
  
    /* Set up boot exceptions vector */  
    adr  x0, boot_vector  
    msr  vbar_el1, x0  
  
    bl  init_c  
  
    /* Should never be here */  
    b   proc_hang  
END_FUNC(_start)
```

# 初始化UART

- 根据UART协议进行内存空间初始化
  - 映射到IO的内存空间

```
void early_uart_init(void)
{
    unsigned int ra;

    early_put32(AUX_ENABLES, 1);
    early_put32(AUX_MU_IER_REG, 0);
    early_put32(AUX_MU_CNTL_REG, 0);
    early_put32(AUX_MU_LCR_REG, 3);
    early_put32(AUX_MU_MCR_REG, 0);
    early_put32(AUX_MU_IER_REG, 0);
    early_put32(AUX_MU_IIR_REG, 0xC6);
    early_put32(AUX_MU_BAUD_REG, 270);
    ra = early_get32(GPFSEL1);
    ra &= ~(7 << 12); //gpio14
    ra |= 2 << 12; //alt5
    ra &= ~(7 << 15); //gpio15
    ra |= 2 << 15; //alt5
    early_put32(GPFSEL1, ra);
    early_put32(GPPUD, 0);
    delay(150);
    early_put32(GPPUDCLK0, (1 << 14) | (1 << 15));
    delay(150);
    early_put32(GPPUDCLK0, 0);
    early_put32(AUX_MU_CNTL_REG, 3);
}
```

# 初始化页表并开启MMU

- 初始话页表并开启MMU
  - 将kernel代码映射到低地址段（和物理地址相同）  
和高地址段两份

思考：为何需要将kernel代码同时映射到低地址段和高地址段两份？

```
/* Setup page tables */
adrp    x8, _boot_pgd_down
msr    ttbr0_el1, x8
adrp    x8, _boot_pgd_up
msr    ttbr1_el1, x8
isb

/* invalidate all TLB entries for EL1 */
tli    vmalldis
dsb    ish
isb

enable_mmumctlr_el1 , x8
```

# 进入kernel

- 跳转到kernel的main函数

```
BEGIN_FUNC(start_kernel)
/*
 * Code in bootloader specified only the primary
 * cpu with MPIDR = 0 can be boot here. So we directly
 * set the TPIDR_EL1 to 0, which represent the logical
 * cpuid in the kernel
 */
mov    x3, #0
msr    TPIDR_EL1, x3

ldr    x2, =kernel_stack
add    x2, x2, KERNEL_STACK_SIZE
mov    sp, x2
bl     main
END_FUNC(start_kernel)
```

# Kernel

- 真正进入ChCore
- 开启OS的各种服务
  - 后续课程会介绍

```
void main(void *addr)
{
    u32 ret = 0;

    /* Init uart */
    uart_init();
    kinfo("[ChCore] uart init finished\n");

    /* Init exception vector */
    arch_interrupt_init();
    kinfo("[ChCore] interrupt init finished\n");

    /* Init mm */
    mm_init(NULL);
    kinfo("[ChCore] mm init finished\n");

    /* Init big kernel lock */
    ret = lock_init(&big_kernel_lock);
    kinfo("[ChCore] lock init finished\n");
    BUG_ON(ret != 0);

    /* Init scheduler with specified policy */
    sched_init(&rr);
    // sched_init(&pbrr);
    kinfo("[ChCore] sched init finished\n");

    /* Create initial thread here, which use the `init.bin` */
    create_root_thread("/init.bin");
    kinfo("[ChCore] root thread init finished\n");

    /* Other cores are busy looping on the addr, wake up those cores */
    enable_smp_cores(addr);
    kinfo("[ChCore] boot multicore finished\n");

    /* Enable PMU by setting PMCR_EL0 register */
    pmu_init();
    kinfo("[ChCore] pmu init finished\n");
```

# X86-64的启动过程

- 为了向前兼容，内核启动中存在兼容性工作
- 通过段寄存器，进行模式转换
  - 实模式(16-bit)
  - 保护模式(32-bit)
  - IA32E (64-bit)
- 由于实模式的内存限制(64KB)，再将内核代码按段拷贝到内存中
- 进入内核



# 硬件模拟

# 使用模拟器进行开发

- ARM模拟器

- 仅用软件完全实现硬件行为
- 如同跑在真正的ARM硬件上

像跑在host OS上的普通程序

ChCore

ARM模拟器

Host OS

ARM

# 模拟CPU

```
switch (opcode) {
case 0x10: /* ADD, SUB */
    if (u) {
        OPCODE_ADD(tcg_rd, tcg_rn, tcg_rm);
        tcg_gen_add_i64(tcg_rd, tcg_rn, tcg_rm);
    }
    break;
case 0x11: /* SQADD */
    if (u) {
        gen_helper_neon_qadd_u64(tcg_rd, cpu_env, tcg_rn, tcg_rm);
    } else {
        gen_helper_neon_qadd_s64(tcg_rd, cpu_env, tcg_rn, tcg_rm);
    }
    break;
case 0x5: /* SQSUB */
    if (u) {
        gen_helper_neon_qsub_u64(tcg_rd, cpu_env, tcg_rn, tcg_rm);
    } else {
        gen_helper_neon_qsub_s64(tcg_rd, cpu_env, tcg_rn, tcg_rm);
    }
    break;
}
```

# 模拟寄存器

```
typedef struct CPUARMState {
    /* Regs for current mode. */
    uint32_t regs[16];

    /* 32/64 switch only happens when taking and returning from
     * exceptions so the overlap semantics are taken care of then
     * instead of having a complicated union.
     */
    /* Regs for A64 mode. */
    uint64_t xregs[32];
    uint64_t pc;
```

# 模拟内存

```
AbstractMemory::access(PacketPtr pkt)
{
    assert(pkt->getAddrRange().isSubset(range));
    uint8_t *host_addr = toHostAddr(pkt->getAddr());
    if (pkt->isRead()) {
        assert(!pkt->isWrite());
        if (pkt->isLLSC()) {
            assert(!pkt->fromCache());
            // if the packet is not coming from a cache then we have
            // LL/SC tracking here
            MEM_READ
            read(pkt);
        }
        if (pmemAddr) {
            pkt->setData(host_addr);
        }
        TRACE_PACKET(pkt->req->isInstFetch() ? "IFetch" : "Read");
        stats.numReads[pkt->req->masterId()]++;
        stats.bytesRead[pkt->req->masterId()] += pkt->getSize();
        if (pkt->req->isInstFetch())
            stats.bytesInstRead[pkt->req->masterId()] += pkt->getSize();
    } else if (pkt->isInvalidate() || pkt->isClean()) {
        assert(!pkt->isWrite());
        // in a fastmem system invalidating and/or cleaning packets
        // can be seen due to cache maintenance requests
        // no need to do anything
    }
}
```

# 模拟设备

- 硬盘 : host OS的文件
- VGA显示 : host OS的显示窗口
- 键盘 : host OS的键盘API
- 时钟芯片 : host OS的时钟
- .....

# 为何需要模拟器？

- 测试与调试
- 提高资源利用率
- 正如IBM的虚拟化
  - IBM's M44/44X , 1960s

# 下次课内容

- 中断、异常与系统调用