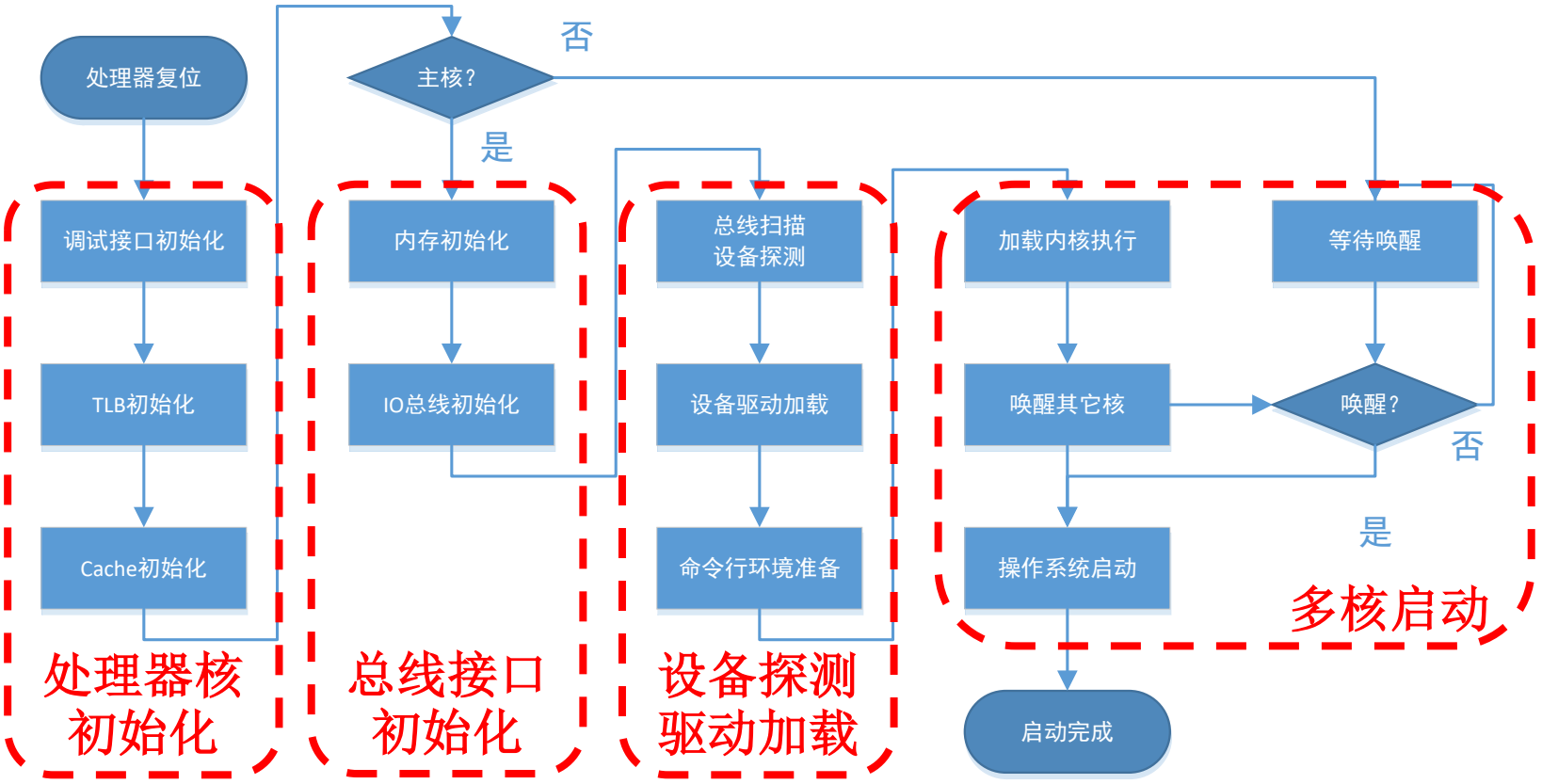
# 计算机系统启动过程分析

胡伟武

# 一句话要点

- 系统启动
  - 从复位到系统可用
- 初始化是什么
  - 将系统各种寄存器状态从不确定设置为确定，将一些模块状态从无序强制为有序的一个过程
- 什么东西需要初始化
  - CPU、内存、各类IO接口
- 怎么初始化
  - 按照从核内到核外，从片内到片外的次序进行

# 系统启动过程示意图


处理器复位
主核？
否
是
调试接口初始化
TLB初始化
Cache初始化
处理器核
初始化
内存初始化
IO总线初始化
总线接口
初始化
总线扫描
设备探测
设备驱动加载
命令行环境准备
设备探测
驱动加载
加载内核执行
唤醒其它核
操作系统启动
等待唤醒
唤醒？
否
是
多核启动
启动完成

# 提纲

- CPU、内存和IO
- 处理器核初始化
- 总线接口初始化
- 设备探测及驱动加载
- 多核启动过程

注意代码背后的结构

# CPU、内存和IO

- 冯诺依曼结构
  - 五部分组成：运算器、控制器、存储器、输入设备、输出设备
  - 物理上分成三大部分：CPU（运算器+控制器）、内存（DRAM）、IO（I+O）
- 三大部分关系
  - CPU/内存：CPULoad/Store访问内存，高带宽、低延迟
  - IO/内存：IO通过DMA访问内存，较高带宽，高延迟
  - CPU/IO：CPU通过PIO访问IO，低带宽、高延迟
- CPU与IO同步关系
  - 查询： CPU通过不断读取IO状态寄存器的内容获取设备控制器的状态
  - 中断：设备完成某个操作时，产生中断通知CPU，把CPU从查询IO中解放出来，提高CPU的利用率，一般与DMA配合使用

# IO寄存器寻址

- IO访问与存储访问的不同
  - 存储器是存储单元阵列，存储访问通过读写指令直接完成，对某单元的读写不会影响其它单元
  - IO设备都有专门的设备控制器，设备控制器向CPU提供一组IO寄存器，CPU通过读取IO寄存器获知IO控制器状态，通过写（有时候是读）IO寄存器来控制IO设备，CPU写入IO寄存器的数据，会被设备控制器解释成控制IO设备的命令
- IO寄存器寻址
  - 通过独立的IO指令寻址，如X86的IN和OUT指令
  - 内存映射统一寻址，如MIPS通过LW和SW指令的地址区分是内存访问还是IO访问

# CPU和IO设备间的同步

- CPU和IO控制器是两个不同的主体，两者协同完成IO访问，
- 查询方式
  - CPU通过不断读取IO状态寄存器的内容获取设备控制器的状态
  - 如打印机控制器包括状态寄存器和控制寄存器，CPU在打印一串数据时，先把数据写入数据寄存器，然后不断读取状态寄存器的值，当读出的“完成位”为“1”时，再把下一个数据写入数据寄存器
- 中断方式
  - 查询方式效率太低，改由设备完成某个操作时，产生中断通知CPU，把CPU从查询IO中解放出来，提高CPU的利用率
  - 如CPU写入打印机的数据寄存器后，转去执行别的操作，打印机打印完数据寄存器的数据后，通过中断通知CPU，CPU再查询状态寄存器

# 存储器和IO设备间的通信

- 存储器和IO设备之间需要大量的通信
  - 系统启动时，需要操作系统代码和数据从硬盘搬运到内存
  - 显示输出时，需要把显示数据从内存搬运到GPU的显存
  - 冯诺依曼结构本质上是以内存为中心的结构
- 存储器和IO设备间通信有两种方式
  - PIO（ Programming Input/Output）方式：CPU从IO设备/内存中把数据读到CPU内部寄存器中，再写入到内存/IO设备
  - DMA（Direct Memory Access）方式：在内存和外设之间开辟直接的数据传输通道，由DMA控制器控制数据在内存和外设之间直接、连续传输

# PIO传输方式

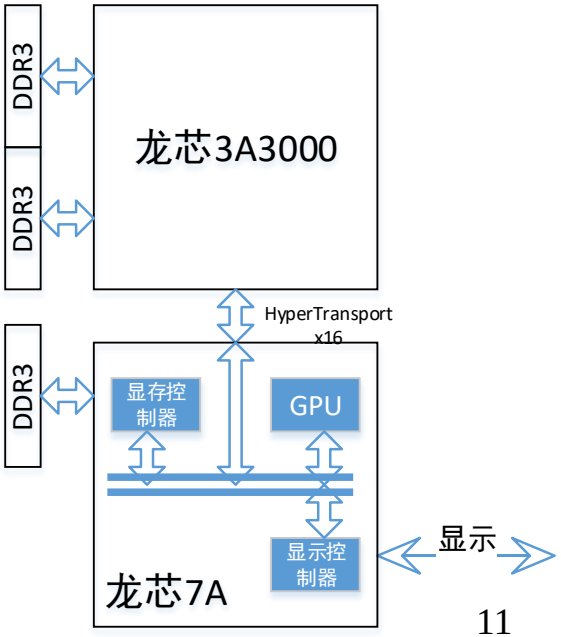
- IO设备和内存之间的通信通过CPU的寄存器中转
  - CPU和IO设备之间的同步可以是查询方式，也可以是中断方式
- 使用PIO传输方式的设备
  - 计算机中多数设备都可以通过PIO访问
  - 低速设备如键盘、鼠标只能通过PIO访问
- CPU访问IO设备都是Uncache访问
  - 提供Uncache Accelerate加速：把多个连续访问合并成一个大的
  - 对很多IO设备的性能至关重要，如CPU往GPU传数据时，部分数据通过PIO方式传输

# DMA数据传输

- 在存储器和外设之间开辟直接的数据传送通道，数据传送由专门的硬件（DMA控制器）来控制。
  - 处理器准备某内存区域为DMA区域
  - 处理器设置DMA控制器参数后可以去干别的事
  - DMA控制器进行数据传输
  - DMA控制器向处理器发出一个中断，通知处理器数据传送的结果
  - 处理器收到中断后安排下一块传输
- DMA要Cache和内存的一致性
  - DMA前CPU先刷Cache，现代处理器一般由硬件自动维护一致性
- 多数高速IO设备均采用DMA传输方式
  - 硬盘、网络

# CPU、GPU与DC间的数据传输

- 龙芯3A3000+龙芯7A桥片两片结构
  - GPU与DC（显示控制器）在桥片中集成
  - 专供GPU、DC使用的显存控制器在桥片中集成
- CPU、GPU与DC之间的几种数据传输方式
  - PIO方式
    - CPU读写GPU中的控制寄存器
    - CPU读写DC中的控制寄存器
    - CPU读写显存
  - DMA方式
    - GPU读写内存/显存
    - DC读内存/显存

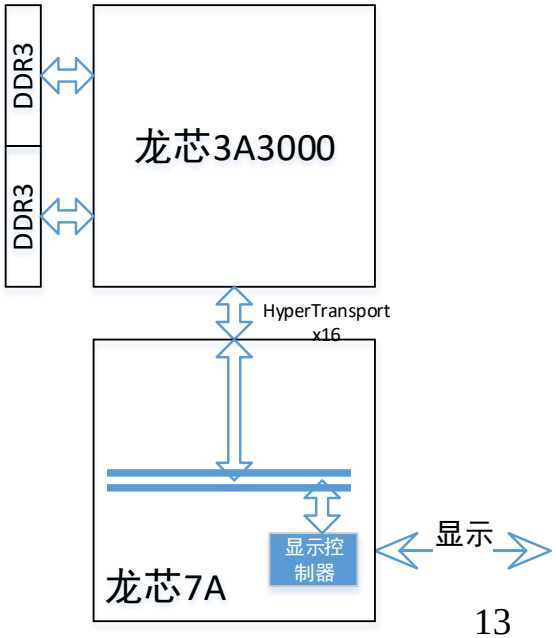


# GPU与DC间的DMA差异

- DC的DMA行为比较简单
  - DC的作用是将画面进行持续显示，无需进行复杂的计算
  - DC的DMA初始化主要是将显示分辨率，帧缓冲（framebuffer）指针配置好。然后开始周期性地DMA数据搬运。
  - CPU或GPU在这一过程中会周期性地向一个帧缓冲或多个帧缓冲（与DC的实现相关）进行填充，填充时机通常是由DC的中断决定
- GPU的DMA行为稍微复杂
  - 只有需要的时候，CPU才会调用GPU进行运算
  - GPU的初始化主要是描述符列表的指针，描述符是在内存里规定好的数据结构。每个描述符中又包含指向数据区域的指针
  - CPU在需要的时候，会将数据区域填好，修改相应的描述符，再通知GPU启动DMA。GPU通过描述符得到数据区域，进行相应计算。

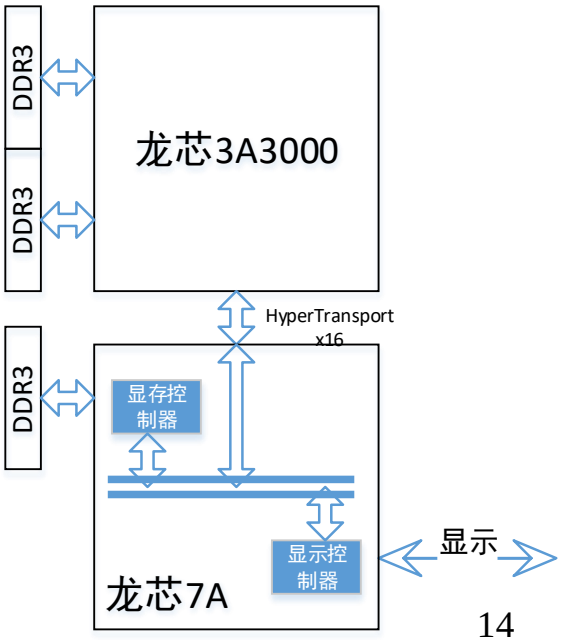
# 模式一：CPU与DC使用共享显存

- 不使用GPU
- 共享显存
  - 不使用桥片上的显存，而在内存中分配一个区域专供显示使用
  - 这个区域称之为framebuffer（帧缓存）
- 数据传输方式
  - CPU读写DC中的控制寄存器，启动DMA
    - PIO操作
  - CPU将需要显示的内容写入内存framebuffer
    - CPU正常内存操作
  - DC读内存framebuffer
    - DMA操作



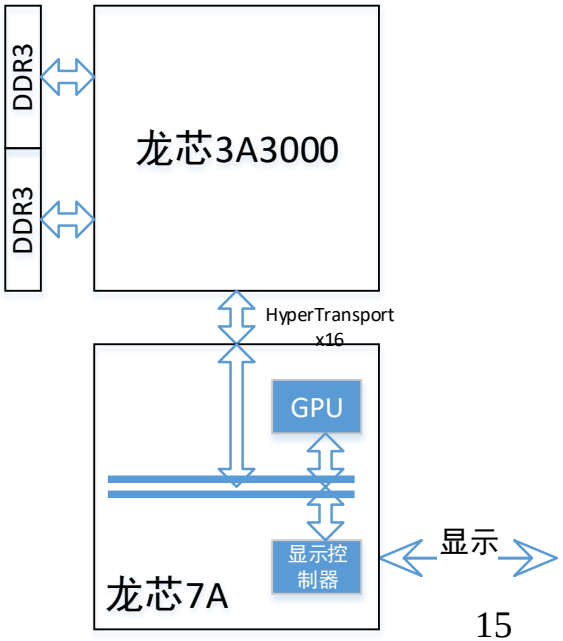
# 模式二：CPU与DC使用独立显存

- 不使用GPU
- 独立显存
  - 使用桥片上的显存
  - 这个区域称之为framebuffer（帧缓存）
- 数据传输方式
  - CPU读写DC中的控制寄存器，启动传输
    - PIO操作
  - CPU将需要显示的内容从内存读出，再写入独立显存上的framebuffer
    - PIO操作
  - DC读显存framebuffer
    - 桥片内的访存显存操作



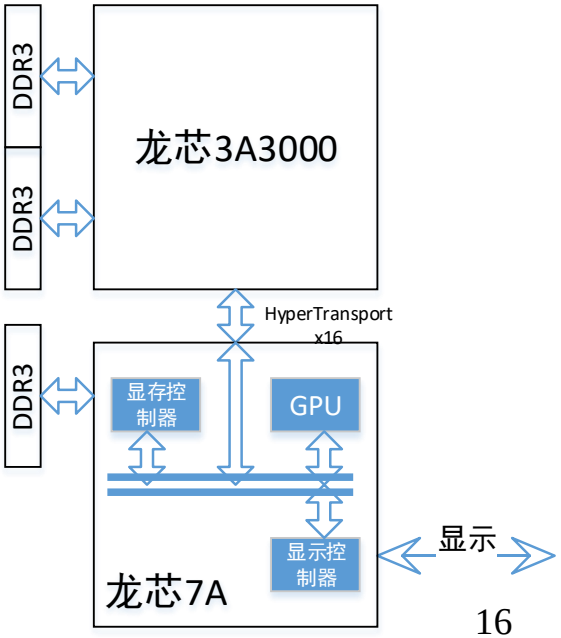
# 模式三：CPU、GPU/DC使用共享显存

- 数据传输方式
  - CPU读写DC中的控制寄存器，启动DMA
    - PIO操作
  - CPU在内存中分配GPU使用的空间，并将相关数据填入
  - CPU读写GPU中的控制寄存器，启动DMA
    - PIO操作
  - GPU读内存
    - DMA操作
  - GPU将计算结果写入内存framebuffer
    - DMA操作
  - DC读内存framebuffer
    - DMA操作



# 模式四：CPU、GPU/DC使用独立显存

- 数据传输方式
  - CPU读写DC中的控制寄存器，启动DMA
    - PIO操作
  - CPU在内存中分配GPU使用的空间，并将相关数据填入
  - CPU读写GPU中的控制寄存器，启动DMA
    - PIO操作
  - GPU读内存
    - DMA操作
  - GPU将计算结果写入显存framebuffer
    - 桥片内的访存显存操作
  - DC读内存framebuffer
    - 桥片内的访存显存操作



# 几种使用模式比较

- 无GPU的共享显存与独立显存
  - 对CPU来说，写内存的性能比写IO性能高得多
  - 在DC读内存（DMA）带宽能够保证的前提下，共享显存性能更好
- 有GPU的共享显存与独立显存
  - 独立显存的使用能够减少两次DMA操作，有效降低内存带宽需求
- 有GPU与无GPU的比较
  - GPU对于复杂的图形变换计算效率较高，尤其是处理3D图形时
  - CPU在处理某些简单图形时会更快，并且节省了DMA启动开销
- 实际使用中，即使有GPU，也会存在一些显示数据由CPU直接写入framebuffer的情况，所以PIO性能也会对系统性能造成一定影响

# IO中断控制器

- 中断：打断当前CPU的执行进程，转去执行某特定程序
  - 中断源发出中断信号到中断控制器
  - 中断控制器产生中断请求给CPU
  - CPU响应中断并读取中断类型码
  - CPU根据中断类型执行相应的中断服务程序
  - CPU从中断服务程序返回
- 中断信号的传递
  - 中断线直传和MSI（Message Signaled Interrupt）
- Intel 8259A的中断寄存器
  - 中断请求寄存器（IRR）：存放当前的中断请求
  - 中断在服务寄存器（ISR）：存放正在服务的中断请求
  - 中断屏蔽寄存器（IMR）：存放中断屏蔽位

# PIO与DMA

| 键盘输入（PIO） | 网络收包（DMA） |
| --- | --- |
| 敲击键盘 | 接收端看到网络包 |
| 键盘输入被记录在PS/2控制器内 | 网卡将收到的网络包写入内存中预先分配好的区域 |
| 中断CPU | 中断CPU |
| CPU查询中断源，发现键盘中断 | CPU查询中断源，发现网络中断 |
| CPU从南桥的PS/2控制器读键盘值 | CPU从内存中读网络包，初始化新的接收缓冲区供网卡使用 |
| CPU清中断 | CPU清中断 |

# 提纲

- **CPU、内存和IO**
- 处理器核初始化
- 总线接口初始化
- 设备探测及驱动加载
- 多核启动过程

注意代码背后的结构

# 处理器复位

- 从芯片引脚输入电平信号，将处理器内部的部分寄存器等状态置为预设值
  - 开始取指（MIPS处理器的第一条指令为0xBFC00000）
- 复位输入是一个硬件初始化动作，至少需要完成最基本的芯片状态初始化（保证第一条指令取指执行）
  - 控制寄存器状态初始化，如处于核心态
  - 核内程序计数器（PC）初始化为0xBFC00000
  - 片内对于0xBFCxxxxx段地址的路由通路初始化
  - 清空流水线有效位避免混乱
  - 为什么不初始化更多硬件（如内存、Cache、TLB、寄存器）？
- 复位后，CPU内部只有取指通路一丝光亮，其余漆黑一片

# 首条指令

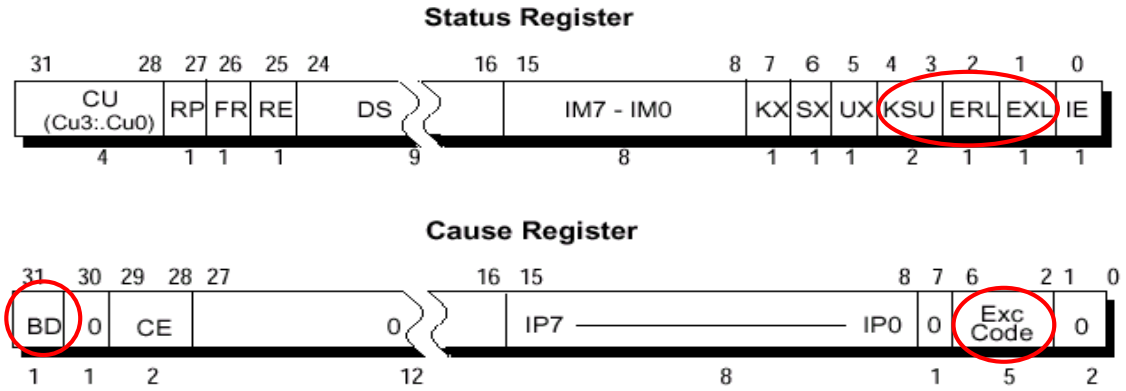
- 初始化CP0（0号协处理器，用于控制处理器行为）
  - 状态寄存器Status
  - 原因寄存器Cause
- 初始化软件约定使用的通用寄存器
  - 栈指针：BIOS用的堆栈
  - 全局指针：BIOS用的地址空间
  - 内核约定物理地址240MB-256MB之间的空间给BIOS用

```
mtc0    zero, COP_0_STATUS_REG
mtc0    zero, COP_0_CAUSE_REG
li      t0, SR_BOOT_EXC_VEC
mtc0    t0, COP_0_STATUS_REG
la      sp, stack
la      gp, _gp
```

```
mtc0    zero,c0_sr
mtc0    zero,c0_cause
lui     t0,0x40    //SR bit26
mtc0    t0,c0_sr
lui     sp,0x8f90  //249MB
addiu   sp,sp,-16384 //-0x4000
lui     gp,0x8f9a
addiu   gp,gp,-8192   //-0x2000
```

# Status和Cause控制寄存器

- 三种系统状态
  - User mode：EXL=0 and ERL=0 and KSU=10
  - Supervisor mode：EXL=0 and ERL=0 and KSU=01
  - Kernel mode：EXL=1 or ERL=1 or KSU=00
- 64位和32位模式
  - KX、SX、UX分别表示三种状态是64位还是32位模式



# 外部调试接口初始化

- 外部调试接口本身不是处理器核初始化的内容，但调试时需要尽早开始人机交互，方便软件调试
- 输出
  - 蜂鸣器
  - 数码管显示
- 输入输出
  - 串口控制器
- 显示器这样的设备相比之下是比较复杂的接口，在启动的后期才会使用

# 串口初始化

- 建立主机与被调试机之间的通信协议
  - 其中GS3_UART_BASE是串口控制器内部的寄存器
- 所谓驱动就是读写IO设备控制器
  - 注意读写IO设备与读写存储器不同

```
LEAF(initserial)
        li      a0, GS3_UART_BASE

        li      t1, 128
        sb      t1, 3(a0)

        li      t1, 0x12
        sb      t1, 0(a0)




        li      t1, 0x0
        sb      t1, 3(a0)

        li      t1, 0
        sb      t1, 1(a0)

        li      t1,71
        sb      t1,2(a0)
        jr      ra
        nop
END(initserial)
```

加载串口设备基地址

线路控制寄存器，写入10000000表示后续寄存器访问为分频寄存器

配置串口波特率分频，当串口控制器输入频率为33MHz时，将串口通讯速率设置在115200，分频的方式为
33,000,000 / 16 / 0x12 = 114583
由于串口通信有固定的格式，只要两端的速率保持在一定范围之内就可以保证传输的正确性

设置传输字符宽度为8，同时将后续寄存器访问设置为正常寄存器

不使用中断模式

# Cache初始化

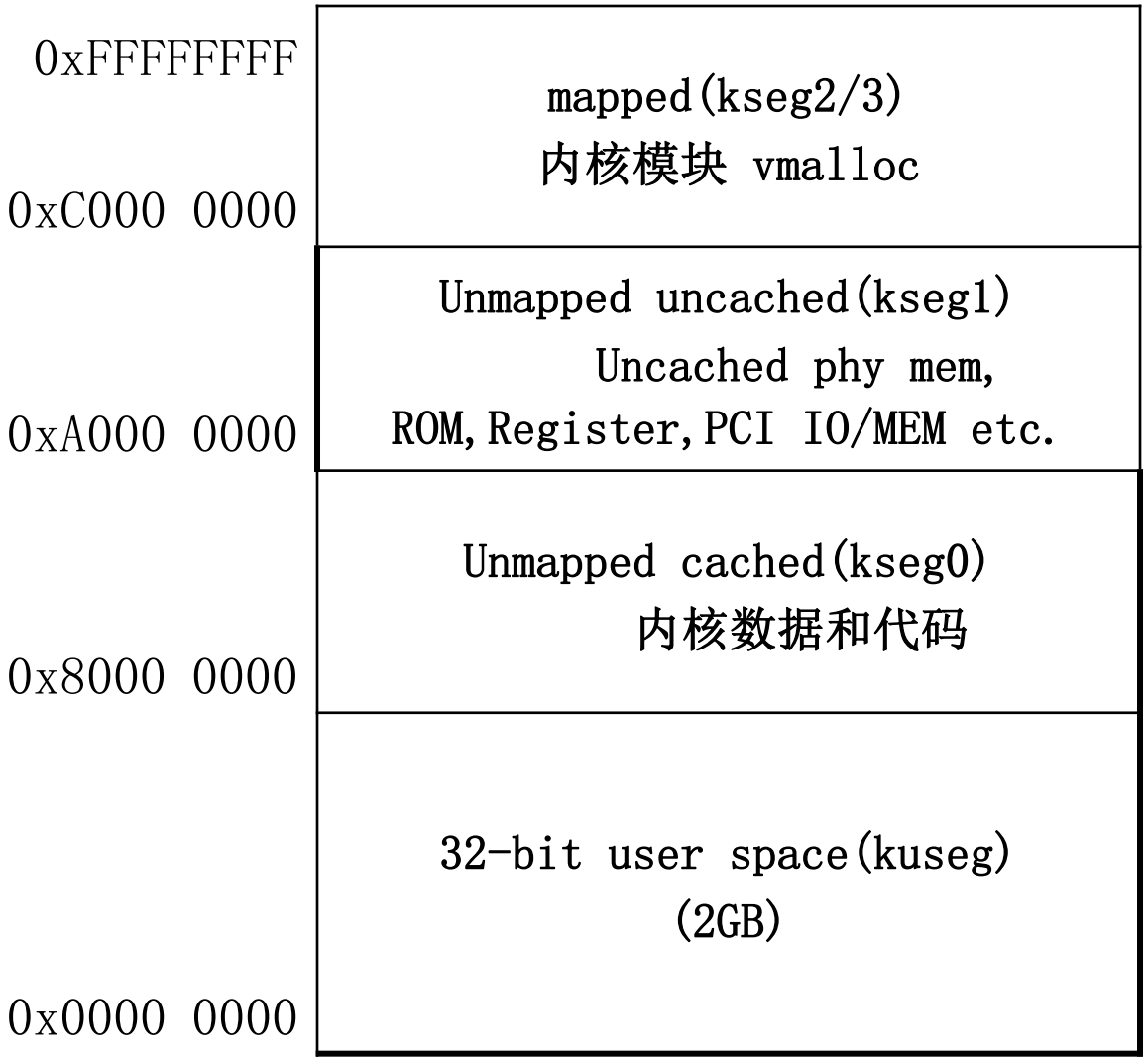
- 复位后BIOS刚开始启动时，处理器从非缓存空间开始执行
  - 上千拍完成一条指令的的取指和执行
  - Cache初始化后，跳转到Cache空间执行，全速流水
  - 尽早完成Cache初始化，使用Cache地址执行能够大大提升启动速度
- 从Cache来看，MIPS地址空间，分为非缓存与缓存
  - 0x80000000 – 0x8FFFFFFF：Cached/Unmapped内存空间
  - 0x9FC00000 – 0x9FCFFFFF：Cached/Unmapped取指空间
  - 0xB0000000 – 0xBFFFFFFF：Uncached/Unmapped IO空间
  - Cached/Unmapped取指空间和Uncached/Unmapped IO空间的物理地址是相同的，Cache失效会自动到IO空间去取指

# MIPS存储空间分段情况

- 32位模式

| 地址范围 | 容量 | 映射方式 | Cached | 访问权限 |
|---|---|---|---|---|
| 0xE0000000-0xFFFFFFF | 0.5GB | 查找TLB | Yes (TLB) | Kernel |
| 0xC0000000-0xDFFFFFFF | 0.5GB | 查找TLB | Yes (TLB) | Kernel, Supervisor |
| 0xA0000000-0xBFFFFFFF | 0.5GB | 地址-0xA0000000 | No | Kernel |
| 0x80000000-0x9FFFFFFF | 0.5GB | 地址-0x80000000 | Yes (Config) | Kernel |
| 0x00000000-0x7FFFFFFF | 2GB | 查找TLB | Yes (TLB) | Kernel, Supervisor, User |

# Linux/MIPS虚拟地址空间安排



# 一级Cache初始化代码

```
LEAF(godson2_cache_init)
    lui     a0, 0x8000
    li      a2, (1<<14)

    mtc0    $0, CP0_TAGHI
    mtc0    $0, CP0_TAGLO
    li      a1, 0x22
    addu    v0, $0, a0
    addu    v1, a0, a2
1:
    slt     a3, v0, v1
    beq     a3, $0, 2f
    nop
     mtc0    a1, CP0_ECC
    cache   Index_Store_Tag_D, 0x0(v0)
    cache   Index_Store_Tag_D, 0x1(v0)
    cache   Index_Store_Tag_D, 0x2(v0)
    cache   Index_Store_Tag_D, 0x3(v0)
    mtc0    zero, CP0_ECC
    cache   Index_Store_Tag_I, 0x0(v0)
    cache   Index_Store_Tag_I, 0x1(v0)
    cache   Index_Store_Tag_I, 0x2(v0)
    cache   Index_Store_Tag_I, 0x3(v0)
    b       1b
    daddiu   v0, v0, 0x20
2:
    jr      ra
    nop
END(godson2_cache_init)
```

基地址
64KB/4路，为Index的实际数量

对TAGHI进行初始化
对TAGLO进行初始化
A1寄存器用于存放ECC校验码，全0时，ECC位为0x22

根据V0和V1的值判断是否已经完成所有项的遍历

数据Cache采用ECC检验，初始化为0x22

对4路数据Cache分别进行写TAG操作

指令Cache采用奇偶检验，初始化为0

对4路指令Cache分别进行写TAG操作

# MIPS的Cache指令及寄存器

- Cache在功能上对用户程序时透明的，但对OS是可见的
  - 硬件一般不对Cache做初始化，刚上电时Cache内容是乱的
  - 需要软件把Cache初始化成任何访问都不命中
- MIPS的Cache相关寄存器
  - 包括TagHi和TagLo寄存器，用于读写Cache Tag
- MIPS的Cache指令，只能在核心态下使用
  - MIPS有一系列Cache指令用来访问Cache
  - 包括根据index读写Cache的Tag和数据等

# 开Cache执行

- Cache初始化仅仅是使Cache可用
  - 要使用Cache必须开Cache，并使用Cache地址操作
  - 还是从BIOS取指，但取指后的指令进Cache了
  - 注意0xA0000000段和0x80000000段的物理地址是相同的

```
## enable kseg0 cachablilty####
        mfc0    t6, CP0_CONFIG
        ori     t6, t6, 7
        xori    t6, t6, 4
        mtc0    t6, CP0_CONFIG

#jump to cached kseg0 address
        PRINTSTR("Jump to 9fc\r\n")
        lui     t0, 0xdfff
        ori     t0, t0, 0xffff
        bal     1f
        nop
1:
        and     ra, ra, t0
        addiu   ra, ra, 16
        jr      ra
        nop
2:
```

开Cache执行
使能0x80000000 – 0x9FFFFFFF的Cache功能

将程序地址跳转到0x9FC00000段执行
BIOS执行到该位置时，应该在Uncache的取指执行，所在的地址段为0xBFC00000 – 0xBFCFFFFF。

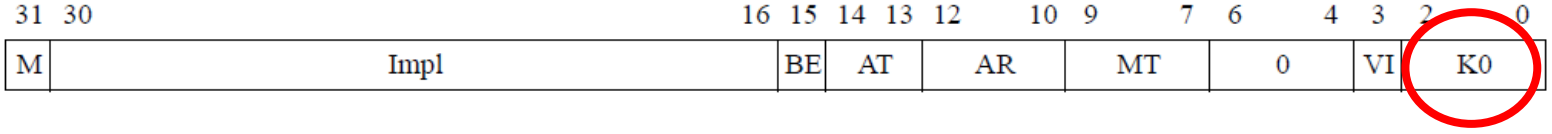
BAL指令将会将延迟槽后的指令地址填入RA寄存器，正常情况下为0xBFCxxxxx。

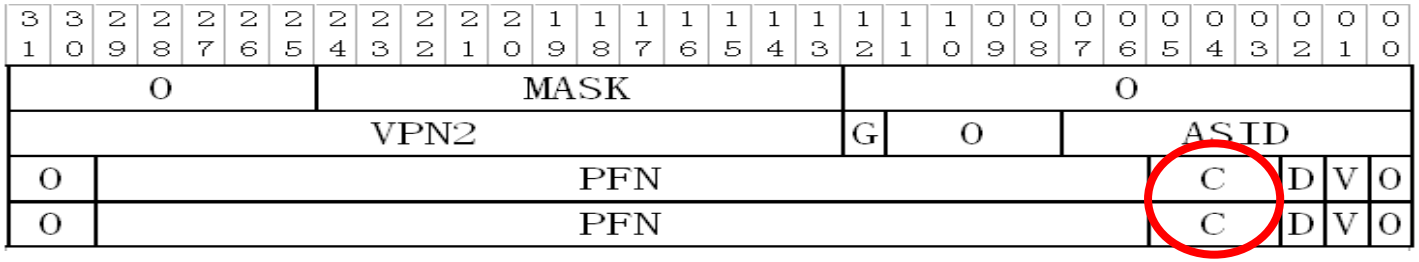
0xBFCx & 0xDFFF = 0x9FCx，地址变换为Cached
将记录的地址再加16，实际上是原指令后的第4条，也即标号2处的指令
跳转到该指令的Cache空间处

# Cache算法配置

- 三位决定Cache算法
  - 对于kseg0段（0x80000000段），由Config寄存器的k0域决定
  - 其它部分由TLB表项决定
  - 2、3分别表示uncached和cached，其余6个值自己定

**Figure 8-23 Config Register Format**



# TLB初始化

- TLB用于虚实地址转换
- 从TLB来看，MIPS地址空间，分为非映射与映射
  - 0x80000000 – 0x8FFFFFFF：Cached/Unmapped内存空间
  - 0x9FC00000 – 0x9FCFFFFF：Cached/Unmapped取指空间
  - 0xB0000000 – 0xBFFFFFFF：Uncached/Unmapped IO空间
  - 对于BIOS，基本使用非映射空间，无需TLB转换
- 对于大地址空间的使用，会借助TLB映射
  - 0xC0000000 -> 0x40000000：对应显存等大IO空间

# MIPS存储空间分段情况

- 32位模式

| 地址范围 | 容量 | 映射方式 | Cached | 访问权限 |
|---|---|---|---|---|
| 0xE0000000-0xFFFFFFFF | 0.5GB | 查找TLB | Yes (TLB) | Kernel |
| 0xC0000000-0xDFFFFFFF | 0.5GB | 查找TLB | Yes (TLB) | Kernel, Supervisor |
| 0xA0000000-0xBFFFFFFF | 0.5GB | 地址-0xA0000000 | No | Kernel |
| 0x80000000-0x9FFFFFFF | 0.5GB | 地址-0x80000000 | Yes (Config) | Kernel |
| 0x00000000-0x7FFFFFFF | 2GB | 查找TLB | Yes (TLB) | Kernel, Supervisor, User |

# TLB初始化代码

- 逐一清空每一个TLB项
  - 需要使用时再填入正确映射（通过TLB Refill例外）

```
LEAF(CPU_TLBClear)
        li      a3, 0                   循环控制变量
        li      a2, 64
        li      a2, PG_SIZE_4K          设置页大小
        MTC0    a2, COP_0_TLB_PG_MASK
1:
        MTC0    zero, COP_0_TLB_HI      对TLB_HI寄存器初始化，对应于VPN
        MTC0    zero, COP_0_TLB_LO0     对TLB_LO0寄存器初始化，对应于PFN0
        MTC0    zero, COP_0_TLB_LO1     对TLB_LO1寄存器初始化，对应于PFN1
        mtc0    a3, COP_0_TLB_INDEX     当前处理的TLB项索引号
        addiu   a3, 1
        tlbwi                           操作TLB表
        bne      a3, a2, 1b
        nop

        jr       ra
        nop
END(CPU_TLBClear)
```

# MIPS的TLB及相关控制寄存器

- 32位模式
- 全相联
- 32-64项

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | | | | | MASK | | | | | | | | | | 0 | | | | | | | | | |
| VPN2 | | | | | | | | | | | | | | | | | | | | G | 0 | | | ASID | | | | | | | |
| 0 | PFN | | | | | | | | | | | | | | | | | | | | | | | C | | | D | V | 0 | | |
| 0 | PFN | | | | | | | | | | | | | | | | | | | | | | | C | | | D | V | 0 | | |

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pagemask | 0 | | | | | | | | | | | | MASK | | | | | | | | | | 0 | | | | | | | | | |
| EntryHi | VPN2 | | | | | | | | | | | | | | | | | | | | | | 0 | ASID | | | | | | | |
| EntryLo0 | 0 | PFN | | | | | | | | | | | | | | | | | | | | | | | C | | | D | V | G | | |
| EntryLo1 | 0 | PFN | | | | | | | | | | | | | | | | | | | | | | | C | | | D | V | G | | |
| Index | P | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | Index | | |
| Random | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Random | | |
| Wired | 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Wired | | |
| EPC | EPC | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| BadVAddr | Bad Virtual Address | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Context | PTEBase | | | | | | | | | | | | | | | BadVPN2 | | | | | | | | | | | | | | 0 | | |

# 处理器核初始化后

- CPU内部一片光亮，除了串口和BIOS接口，多数“门窗”都没开

# 提纲

- CPU、内存和IO
- 处理器核初始化
- 总线接口初始化
- 设备探测及驱动加载
- 多核启动过程

# 内存接口初始化

- 到此为止，BIOS从 Flash读程序，写IO或控制寄存器器
  - 相比Flash设备，内存接口的访问性能大大提升
- 内存接口的初始化与核内部件的初始化的差别
  - Cache/TLB的初始化主要是将内容设置为无效
  - 内存接口的初始化针对接口控制
    - 通过内存的SPD（并非必需），获取内存大小，类型，频率，延迟等各种信息
    - 根据所得到的内存信息对控制器和内存进行设置
    - 可能还有对于时序配合的信号训练
    - 对内容并不关心（ECC内存除外，不初始化内容会导致ECC错）

# 内存控制器配置代码

- 将预先定义好的值写入内存控制器的相应寄存器中
- 内存控制器将自动对内存进行初始化配置
  - 主要设置延迟、匹配阻抗等
- 初始配置之后，会再对内存信号进行训练

```
ddr2_config:
    daddu   a2, a2, s0
    dli     t1, DDR_PARAM_NUM
    daddiu  v0, t8, 0x0
1:
    ld      a1, 0x0(a2)
    sd      a1, 0x0(v0)
    daddiu  t1, t1, -1
    daddiu  a2, a2, 0x8
    daddiu  v0, v0, 0x8
    bnez    t1, 1b
    nop
```

a2为调用该程序时传入的参数，与s0之和用于表示初始化参数在FLASH中的基地址
t1用于表示内存参数的个数
t8为调用该程序时传入的参数，用于表示内存控制器的寄存器基地址

初始化的过程就是从FLASH中取数再写入内存控制器中的寄存器的过程

# IO总线初始化

- 根据不同IO总线的需求进行针对性的初始化
- 通过初始化配置消除与具体实现相关的总线特性，保证软件兼容性
  - 信号定义，硬件实现各不相同：HyperTransport、PCIE等
  - 软件协议兼容PCI：配置空间、IO空间及Memory空间
- 龙芯3号中使用HyperTransport接口
  - 地址划分：确定配置访问、IO访问及Memory访问的地址
  - 设定DMA访问地址
  - 对总线频率、宽度进行重新设置

# 总线接口初始化后

- CPU内部一片光亮，“门窗”也已打开
- 但窗外还是一片漆黑

# 提纲

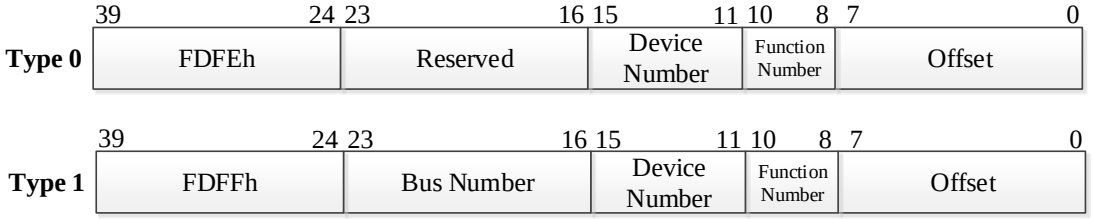
- 处理器核初始化
- 总线接口初始化
- 设备探测及驱动加载
- 多核启动过程

# PCI协议下的工作流程

- 上世纪九十年代提出，软件结构被PCIE和HT等继承
  - 配置空间、IO空间、Memory空间
- 通过配置空间探测总线设备
  - 配置空间大部分情况下是对设备的属性进行刻画
  - 在完成设备探测之后基本不再使用配置空间
- 对总线上的各个设备所需的地址空间进行分配
  - IO、Memory空间是真正使用设备功能时的地址空间
  - 对于IO和内存统一编址的CPU，两者区别不大
- 使用分配好的空间对各个设备进行控制
  - 主要是驱动程序加载
- 特点：
  - 总线设备发生变化时无需修改软件
  - 同一总线支持多个相同设备不会导致冲突

# 设备探测

- 使用配置空间：以HyperTransport的配置空间为例
  - 高位地址访问不同设备，FDFE和FDFF是HT协议规定的
  - 通过总线号、设备号及功能号可以遍历总线设备
  - Offset是设备内地址偏移，索引设备空间寄存器




  - 通过规定的寄存器得到设备的唯一标识、设备类型等
    - Device ID、Vendor ID
  - 通过基址寄存器（BAR）获取并对设备空间进行配置
    - IO设备需要使用的内存空间

# 设备配置空间寄存器分布

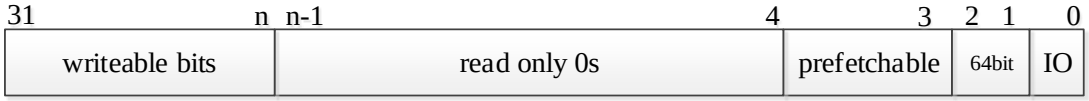
- 一般放在IO控制器上（如PCIE控制器）
  - 所有设备的空间分布都一致
  - Vender ID：Intel为0x8086，龙芯为0x0014
  - Device ID：为全0或全1表示没有设备

| 31 24 | 23 16 | 15 8 | 7 0 | |
|---|---|---|---|---|
| Device ID | | Vendor ID | | 00h |
| Status | | Command | | 04h |
| Class Code | | | Revision ID | 08h |
| BIST | Header Type | Latency Timer | Cache Line Size | 0Ch |
| Base Address Registers | | | | 10h<br>14h<br>18h<br>1Ch<br>20h<br>24h |
| Cardbus CIS Pointer | | | | 28h |
| Subsystem ID | | Subsystem Vendor ID | | 2Ch |
| Expansion ROM Base Address | | | | 30h |
| Reserved | | | Capabilities Pointer | 34h |
| Reserved | | | | 38h |
| Max_Lat | Min_Gnt | Interrupt Pin | Interrupt Line | 3Ch |

***Note:*** *Shaded registers contain minimum-required read-write bits. Other registers are read-only or contain only device-dependent bits.*

# 地址空间分配

- 基址寄存器（BAR）
  - 向该寄存器写入全1，读出再判断0的个数来获取空间需求
  - 再向该寄存器写入分配好的基地址供设备进行命中判断



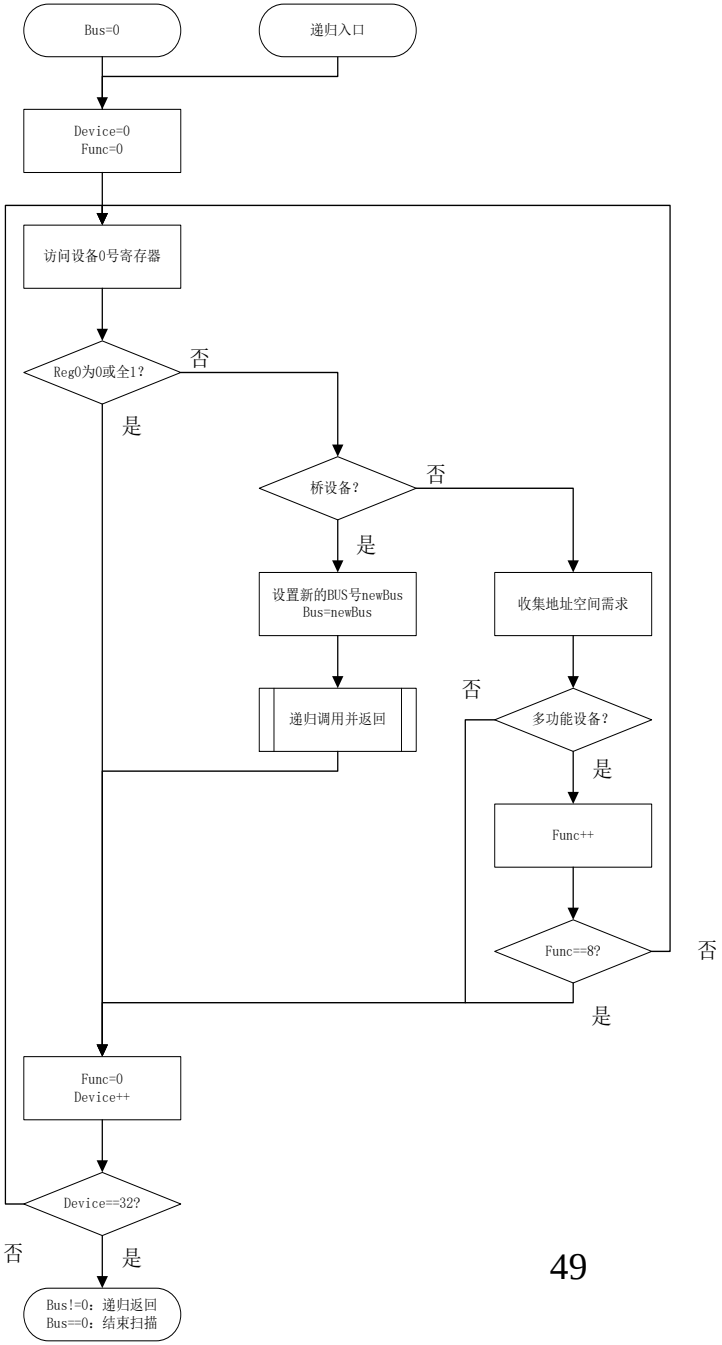

- 每个设备地址空间为2的幂次方对齐
- 先收集所有的设备空间信息
  - 按照大小排序
  - 根据顺序分配基地址，以减少空间碎片

# PCI设备的探测

1. 将初始总线号、初始设备号、初始功能号设为0；
2. 使用当前的总线号、设备号、功能号组成一个配置空间地址，这个地址的构成如图7.3所示，使用该地址，访问其0号寄存器，检查其设备识别号；
3. 如果读出全1或全0，表示无设备；
4. 如果该设备为有效设备，检查每个BAR所需空间大小，并收集相关信息；
5. 检测其是否为多功能设备，如果是则将功能号加1重复扫描，执行第2步；
6. 如果该设备为桥设备，则给该桥配置一个新的总线号，再使用该总线号，从设备号0、功能号0开始递归调用，执行第2步；
7. 如果设备号非31，则设备号加1，继续执行第2步；如果设备号为31，且总线号为0，表示扫描结束；如果总线号非0，则退回上一层递归调用；

# PCI设备探测过程

- 对多功能设备扫描FUNC
  - 可以有更多的地址空间BAR
- 对桥设备递归调用该过程
  - 如南桥上有很多设备
- 遍历整个总线获得所有设备信息
  - 同时对设备地址空间大小进行排序
- 再次遍历对每个设备空间进行配置
  - 把设备地址空间的信息传递给设备驱动程序，设备驱动程序使用配置好的设备地址空间



# 地址空间分配示例

- 设备内部的地址空间，CPU通过Uncache或IO指令访问
- 配置空间扫描后得到的设备及空间需求

| 设备号 | 名称 | BAR号 | 大小 |
|---|---|---|---|
| 1 | USB控制器 | 0 | 4KB |
| 2 | 显示控制器 | 0 | 128MB |
| | | 1 | 64KB |
| 3 | 网卡控制器 | 0 | 4KB |
| | | 1 | 16KB |

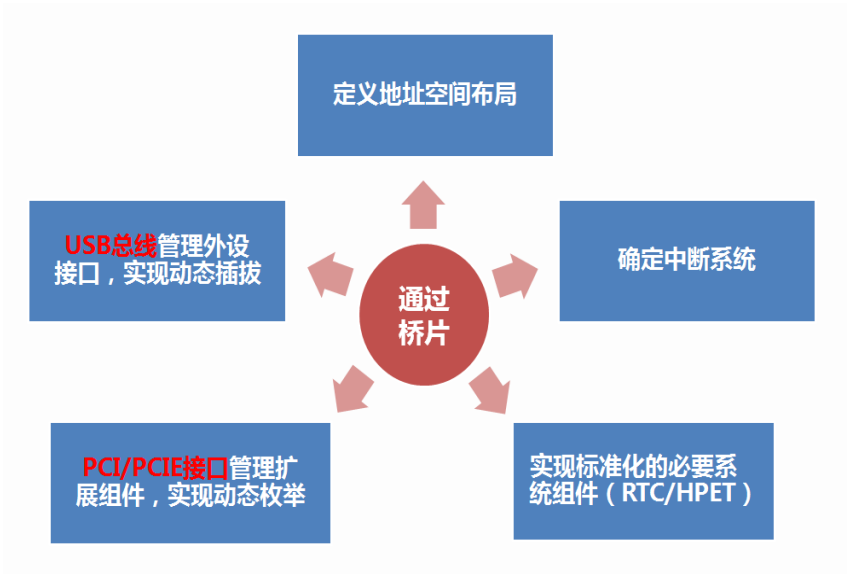
- 最终的地址空间分配

| 设备号 | 名称 | BAR号 | 大小 | 起始地址 | 结束地址 |
|---|---|---|---|---|---|
| 1 | USB控制器 | 0 | 4KB | 0x48015000 | 0x48015FFF |
| 2 | 显示控制器 | 0 | 128MB | 0x40000000 | 0x47FFFFFF |
| | | 1 | 64KB | 0x48000000 | 0x4800FFFF |
| 3 | 网卡控制器 | 0 | 4KB | 0x48014000 | 0x48014FFF |
| | | 1 | 16KB | 0x48010000 | 0x48013FFF |

# 实现OS二进制兼容统一系统架构

| | | | | | |
|---|---|---|---|---|---|
| 跨代兼容、增量式<br>创新的指令集 | 标准化的全局<br>地址空间布局 | 标准化的中断系统<br>(中断路由和编程模型) | 编程接口标准化的<br>多核与互联架构 | 可动态枚举的扩展组件<br>GPU/Raid等 | 动态插拔的外设接口<br>U盘/打印机 |

## 桥片确定系统架构




## BIOS规范系统架构

**UEFI固件标准的三个规范**

| 规范OS的加载过程 | 规范固件和操作系统的接口 | 规范固件的驱动开发 |
|---|---|---|
| • OS影响安装的介质和位置标准化<br>• UEFI提供启动时的标准服务，不再依赖于Int10 | • 通过ACPI传递系统配置，实现功耗管理等功能<br>• 标准的服务机制，提供启动和OS运行时的服务 | • CPU架构无关，便于设备厂商 |

**UEFI平台效果**

| 实现一套OS支持所有差异化主板系统 |
|---|
| 屏蔽硬件方案的差异，实现对统一的抽象层 |

# 提纲

- CPU、内存和IO
- 处理器核初始化
- 总线接口初始化
- 设备探测及驱动加载
- 多核启动过程

# 多核初始化

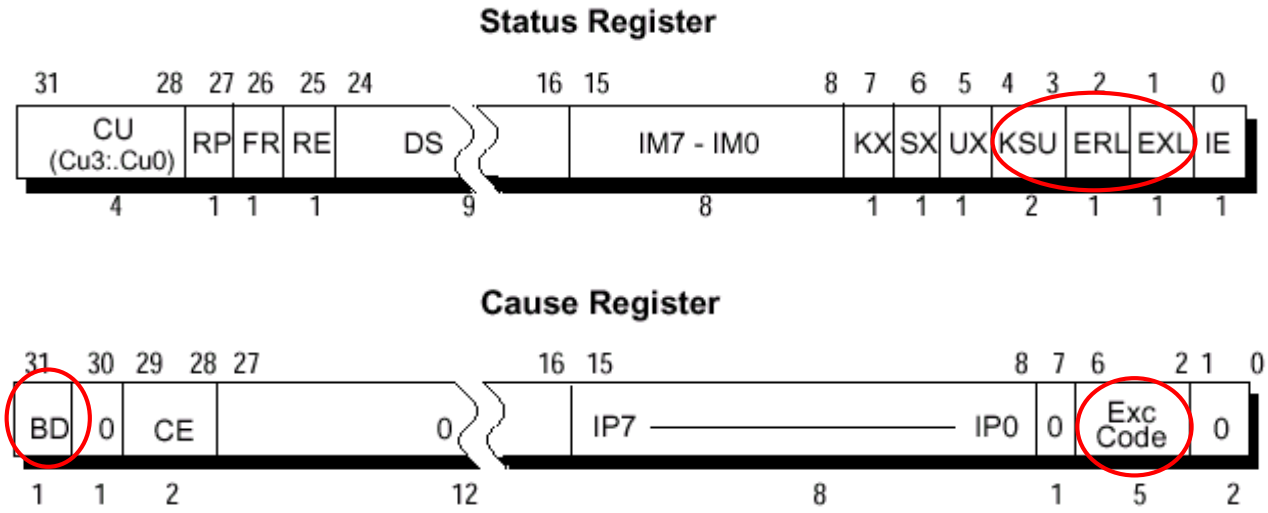
- 一个核执行主流程，其它核仅负责私有部件的初始化
  - 主核：核内私有部件、片内共享部件、总线接口
  - 从核：核内私有部件
- 通过核间通信机制进行同步
  - 串口初始化后，通知从核，开始打印输出
  - 共享Cache初始化后，通知从核，开Cache执行
  - 内存初始化后，通知从核，可以读写内存
- 可以通过并行化共享部件初始化的方法来加速启动
  - 每个核初始化一部分共享Cache
  - 每个核执行不同的初始化：Cache、内存、接口

# 核间通信机制

- 核间中断机制，以龙芯3A为例
  - 每个核有32个不同的中断可供软件使用
  - 可以约定为不同核产生的中断，或者对应不同事件
- 信箱寄存器（查询方式）
  - 多个寄存器供传递参数及特殊事件

| 名称 | 读写权限 | 描述 |
|---|---|---|
| IPI_Status | R | 32位状态寄存器，被置1且对应位使能情况下，处理器核中断线被置位 |
| IPI_Enable | RW | 32位使能寄存器，控制对应中断位是否有效 |
| IPI_Set | W | 32位置位寄存器，往对应的位写1，则对应的STATUS寄存器位被置1 |
| IPI_Clear | W | 32位清除寄存器，往对应的位写1，则对应的STATUS寄存器位被清0 |
| MailBox0 | RW | 缓存寄存器，供传递参数使用 |
| MailBox01 | RW | 缓存寄存器，供传递参数使用 |
| MailBox02 | RW | 缓存寄存器，供传递参数使用 |
| MailBox03 | RW | 缓存寄存器，供传递参数使用 |

# Status和Cause寄存器



# 多核唤醒

- 内核启动过程中，会将从核逐一加入系统管理
  - 这一过程称之为“唤醒”
- 唤醒中主从核的运行
  - 主核：将从核需要运行的程序指针、参数写入信箱寄存器。等待从核被唤醒
  - 从核：轮询信箱寄存器，发现非零值则跳转执行。执行完相关的程序后，通过信箱寄存器通知主核
  - 主核：收到信箱寄存器通知后，继续下一个核的唤醒或其它操作

# BIOS完成上述工作后

- 从硬盘从把内核取到内存（DMA或PIO），并跳转到内核地址，由内核负责后续计算机控制
  - BIOS要把基本的硬件参数通过BIOS与内核接口（如ACPI接口）传递给内核
  - Intel定义的UEFI 和ACPI接口是保证PC跨代兼容的重要原因
- 内核根据预设的（如启动X系统）或临时的（如打开PPT）用户需求，把相关程序取到内存执行
  - 内核通过进程方式调度CPU，通过虚拟存储管理管内存，通过设备驱动管IO

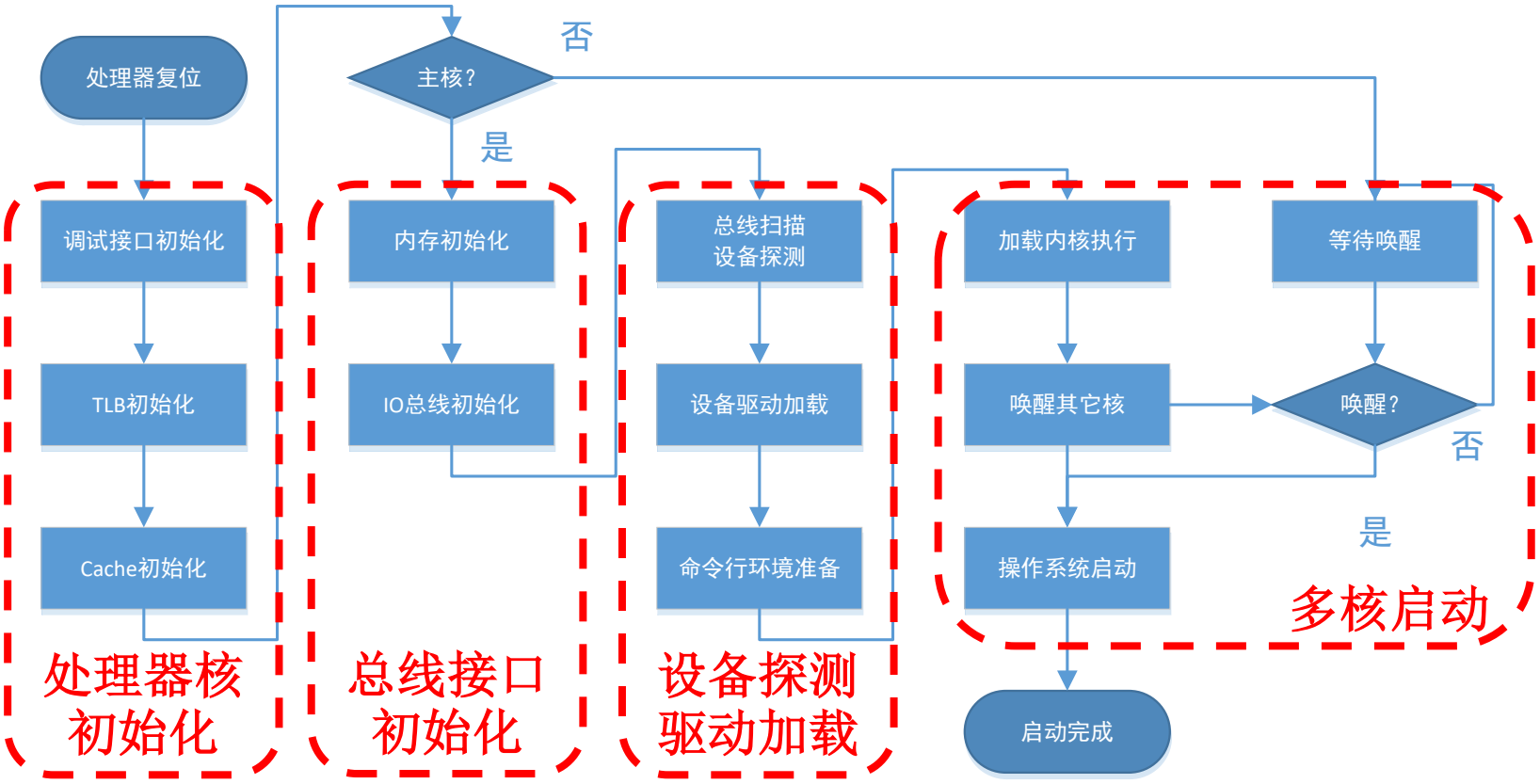
# 小结

- CPU刚上电时，硬件只对必要的状态进行复位
  - 除了从PC到BIOS接口有一丝光亮，其它都是漆黑一片
  - 从0xBFC00000取第一条指令时要绕过Cache和TLB
- 软件初始化必要的调试接口：指示灯、蜂鸣器、串口
  - 从SPI接口（最高33MHz）取指，每拍一位，32拍才取回一条指令，CPU要1000拍才执行一条指令；CPU内部有些朦胧的光亮
  - Load/Store指令访问IO接口的控制寄存器与访问内存有不同的含义
- 软件初始化CPU内部
  - 先初始化Cache，CPU可以高速运行；再打开TLB，访问地址空间更大
  - CPU内部一片光明，但外部还是漆黑一片
- 软件初始化内存控制器
  - 通过I2C总线读取内存条信息
  - 内存控制器参数太多，主要是从BIOS空间读取内存参数并写入内部控制寄存器
  - DDR4 3200每传输一位数据只有0.3ns，电信号只能在主板上行进4-5cm
  - 内存是CPU的后花园，这时CPU通往后花园的路已经打通

# 小结

- 软件初始化IO接口
  - PCI协议：上世纪九十年代提出，软件结构被PCIE和HT等继承
  - 配置空间：通过配置空间探测总线设备，配置空间大部分情况下是对设备的属性进行刻画；CPU通过对设备空间的扫描发现所有IO控制器；
  - IO空间、Memory空间：驱动程序真正使用设备功能时的地址空间；对于IO和内存统一编址的CPU，两者区别不大；
  - 总线设备发生变化时无需修改软件，同一总线支持多个相同设备不会导致冲突
  - Intel的厉害之处：PCI协议使得硬件能自动识别IO设备并加载驱动程序；UEFI协议使得最新的CPU可以运行十年前的操作系统；X86指令系统使得最新的CPU可以运行几十年前的应用程序
- 唤醒其它处理器核，进入多核状态
  - 多核之间通过信箱等方式通信
  - 信箱可以是内存单元，也可以是CPU内部寄存器
- 至此，CPU内部光明一片，对外四通八达

# 小 结


处理器复位
主核？
否
是
调试接口初始化
TLB初始化
Cache初始化
处理器核
初始化
内存初始化
IO总线初始化
总线接口
初始化
总线扫描
设备探测
设备驱动加载
命令行环境准备
设备探测
驱动加载
加载内核执行
唤醒其它核
操作系统启动
等待唤醒
唤醒？
否
是
多核启动
启动完成

# 思考题

- 请列出我国大陆境内提供CPU、GPU、内存颗粒和内存条、闪存芯片和SSD 盘、液晶屏的企业并进行简单评述。