

期末复习

期末复习

第0章 序

第1章 引言

- 1.1 计算机体系统结构的研究内容
- 1.2 衡量计算机的指标
 - 1.2.1 计算机性能
 - 1.2.2 计算机价格
 - 1.2.3 计算机功耗
- 1.3 计算机体系统结构的发展
 - 1.3.1 摩尔定律和工艺发展
 - 1.3.2 计算机应用和体系结构
 - 1.3.3 计算机体系统结构发展
- 1.4 体系结构设计的基本原则
 - 1.4.1 平衡性
 - 1.4.2 局部性
 - 1.4.3 并行性
 - 1.4.4 虚拟化

第2章 指令系统结构

- 2.1 指令系统简介
- 2.2 指令系统设计原则
- 2.3 指令系统发展历程
 - 2.3.1 指令内容演变
 - 2.3.2 存储管理演变
 - 2.3.3 运行级别演变
- 2.4 指令系统组成
 - 2.4.1 地址空间
 - 2.4.2 操作数
 - 2.4.3 指令操作和编码
- 2.5 RISC指令系统比较
- 2.6 C语言的机器表示

第3章 特权指令系统

- 3.1 特权指令系统简介
- 3.2 异常与中断
 - 3.2.1 异常分类
 - 3.2.2 异常处理流程
 - 3.2.3 中断
- 3.3 存储管理

第4章 软硬件协同

- 4.1 应用程序二进制接口
 - 4.1.1 寄存器约定
 - 4.1.2 函数调用约定
 - 4.1.3 进程虚拟地址空间
 - 4.1.4 栈帧布局
- 4.2 六种常见的上下文切换场景
- 4.3 同步与通讯

第5章 计算机组成原理和结构

- 5.1 冯·诺依曼结构
- 5.2 计算机的组成部件
 - 5.2.1 运算器
 - 5.2.2 控制器

- 3.2.3 存储器
- 5.3 计算机系统硬件结构演进
- 5.4 处理器和IO通信
 - 5.4.1 IO寄存器寻址
 - 5.4.2 CPU和IO设备间的同步
 - 5.4.3 存储器和IO设备间的通信 (CPU和IO的通信)
- 第6章 计算机总线接口技术
 - 6.1 总线概述
 - 6.2 总线分类
 - 6.3 片上总线
 - 6.4 内存总线
 - 6.4.1 机械层
 - 6.4.2 电气层
 - 6.4.3 协议层
 - 6.4.4 内存控制器
 - 6.5 系统总线 (to be finished)
 - 6.6 IO总线 (to be finished)
- 第7章 计算机系统启动过程分析
 - 7.1 处理器核初始化
 - 7.2 总线接口初始化
 - 7.3 设备探测及驱动加载
 - 7.4 多核启动
- 第8章 运算器设计
 - 8.1 二进制与逻辑电路
 - 8.1.1 数的表示
 - 8.1.2 MOS晶体管工作原理
 - 8.2 简单运算器设计
 - 8.2.1 一位全加器
 - 8.2.2 行波进位加法器
 - 8.2.3 先行进位加法器 (重要)
 - 8.2.4 补码减法算法
 - 8.2.5 比较器与移位器
 - 8.3 定点补码乘法器
 - 8.3.1 补码乘法器
 - 8.3.2 Booth乘法器
 - 8.3.4 华莱士树 (to be continued)
- 第9章 指令流水线
 - 9.1 时空图
 - 9.2 异常和流水线
 - 9.3 指令调度技术
 - 9.4 提高流水线效率的技术
 - 9.4.1 动态调度
 - 9.4.2 多发射
 - 9.4.3 转移猜测技术
 - 9.4.4 高速缓存
- 第10章 并行编程基础
 - 10.1 程序的并行行为
 - 10.2 并行编程模型
 - 10.3 典型并行编程环境
 - 10.3.1 SIMD
 - 10.3.2 Posix Thread (pthread)
 - 10.3.3 OpenMP
 - 10.3.4 MPI标准
- 第11章 多核处理结构
 - 11.1 多核处理器发展的演化

11.2 多核处理器的访存结构 (to be continued, ESI协议)

11.3 多核处理器的互连结构

11.4 多处理器的同步机制

11.5 典型多核结构

第12章 计算机性能分析

12.1 计算机性能分析和评价

12.2 性能测试程序集

12.3 性能分析方法

第0章 序

什么是计算机体系结构？**描述计算机各组成部分及其相互关系的一组规则和方法，是程序员所看到的计算机属性**

计算机体系结构主要研究内容：**指令系统结构（ISA）和计算机组织结构（CO）**

计算机组织结构=硬件系统结构+CPU内部微结构？

表现方式：结构框图、高级语言、硬件描述语言+EDA

第1章 引言

信息产业的主要技术平台都是以**CPU**和**OS**为核心搭建

1.1 计算机体系结构的研究内容

API: 应用程序编程接口

MCU: 微控制器

HPC: 高性能计算机

通用计算机系统层次结构: 四个层次三个界面

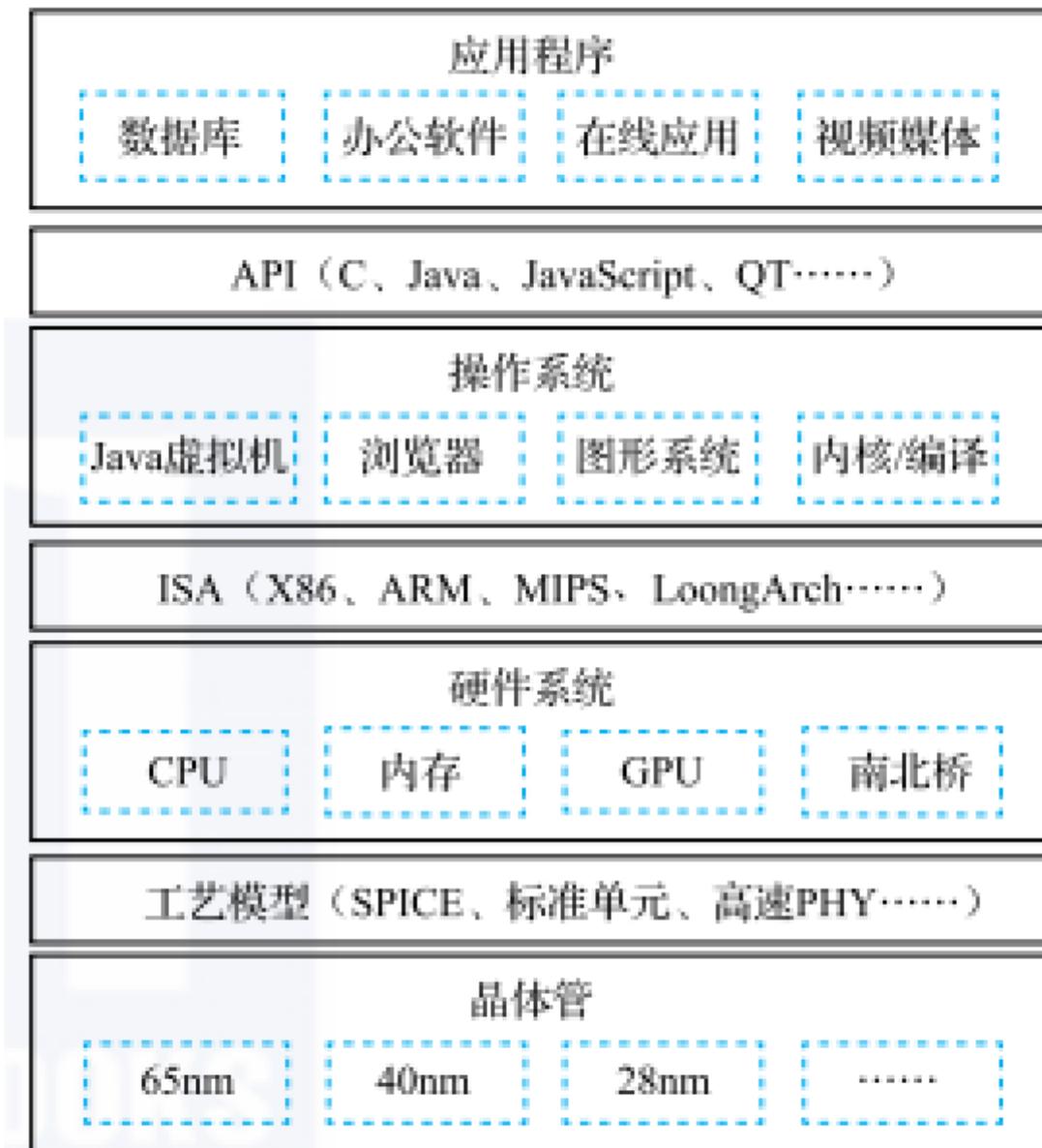


图 1.1 通用计算机系统的结构层次

SPICE: Simulation Program with Integrated Circuit Emphasis

此外，在API和ISA之间还有一层**ABI** (Application Binary Interface)，是应用程序访问计算机硬件和操作系统服务的接口，由**计算机用户态指令**和**操作系统的系统调用**组成。

冯诺依曼结构基本思想（主要特点）：储存程序+指令驱动 —— 计算机体体系结构基础

1.2 衡量计算机的指标

主要指标：性能、价格、功耗

1.2.1 计算机性能

性能的本质定义：完成一个任务需要的时间

时间取决于指令数、每条指令拍数和每拍时间。指令数取决于算法、编译器和指令功能；每条指令拍数与编译器、指令功能和CPU微结构相关；每拍时间与结构、电路、工艺等因素有关。

MIPS: 每秒百万条指令, **MFLOPS**: 每秒百万次浮点运算, **CPI**: 每周期指令数, **IPC**: 每指令周期数
(受处理器微结构影响)

主频宏观上取决于微结构设计,微观上取决于工艺和电路设计。

1.2.2 计算机价格

发展趋势:追求性能(Performance per Second)到追求性能价格比(Performance per Dollar)。

1.2.3 计算机功耗

近几年来, **性能功耗比(Performance per Watt)** 成为重要指标值之一

反相器由一个PMOS和一个NMOS组成,其功耗可以分成三类:**开关功耗**,**短路功耗**和**漏电功耗**。开关功耗来自于给电容充放电,短路功耗来自于P管和N管短路,漏电功耗来自于MOS管不能严格关闭时产生的漏电。

优化芯片功耗一般从两个角度入手:**动态功耗**和**静态功耗**

升级工艺可以有效降低动态功耗(降低电容和电压)

1.3 计算机体系结构的发展

摩尔定律、计算机应用、计算机体系结构三者的关系

以精简指令集(RISC)兴起为标志,**指令系统结构(ISA)**成为计算机体系结构研究重点。

工艺技术的发展和应用需求的提高是计算机体系结构发展的主要动力

2010年以前,计算机工业的发展是工艺驱动为主,应用驱动为辅;未来计算机应用对体系结构的影响将超过工艺技术

1.3.1 摩尔定律和工艺发展

摩尔定律是一个主管规律。每隔18个月工艺提升一代,即同硅面积中晶体管数目提高一倍。(现在放缓到2~3年或更长时间)

从历史上看,工艺技术和体系结构的关系经历了三个阶段:

- 晶体管不够用
- 集成度提升(更多、更快、更省电)
- 晶体管越来越多,但晶体管变得“复杂、不快、不省电、不便宜”

1.3.2 计算机应用和体系结构

1.3.3 计算机体系结构发展

面临复杂度、主频、功耗和带宽的障碍

摩尔定律、计算机应用、计算机体系结构的关系:

半导体工艺(摩尔定律)和计算机体系结构互为动力

应用需求时体系结构发展的长远动力

1.4 体系结构设计的基本原则

平衡性、局部性、并行性、虚拟化

1.4.1 平衡性

木桶效应

计算性能和访存带宽平衡的经验原则：峰值浮点运算速度和峰值访存带宽1:1

Amdahl定律：(e.g. 串并行优化)

1.4.2 局部性

利用事件局部性优化（有的频繁发生，有的不怎么发生）。e.g. 应当加速经常发生的事件

利用访存局部性优化（包括时间局部性和空间局部性）

1.4.3 并行性

三个层次：指令并行、数据并行、任务并行

1.4.4 虚拟化

本质：硬件和用户间的桥梁

第2章 指令系统结构

2.1 指令系统简介

软硬件界面：

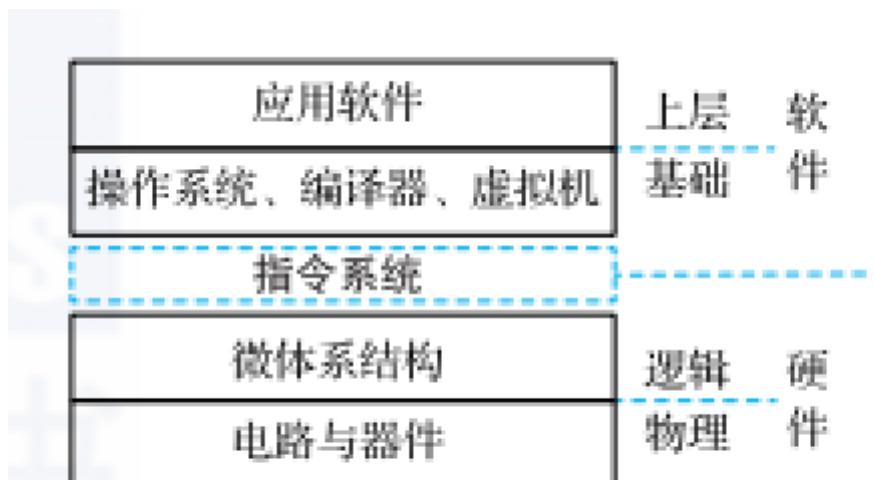


图 2.1 计算机系统的层次

包括：对指令功能、运行时环境（如储存管理、运行级别控制）等内容的定义，设计软硬件交互的各个方面

2.2 指令系统设计原则

兼容性、通用性、高效性、安全性

影响指令系统的因素：

工艺技术、计算机体系结构、操作系统、编译技术、应用程序

2.3 指令系统发展历程

2.3.1 指令内容演变

依据指令长度的不同，可以分为CISC（复杂指令系统），RISC（精简指令系统）和VLIW（超长指令字）。

2.3.2 存储管理演变

存储管理的演变经历了连续实地址、段式、页式虚拟存储管理等阶段

连续实地址：个程序所需的内存空间必须连续存放且不和其他程序冲突

段式：将内存分为多个段和节，将地址组织为相对于段地址的偏移

页式虚拟存储：将各进程划分成若干长度相同的页，虚拟地址和物理地址的对应组织成页表

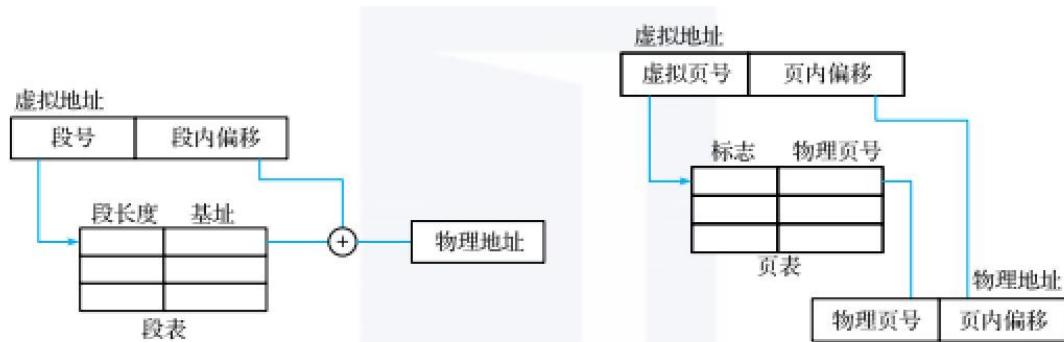


图 2.3 段式存储管理的地址转换过程

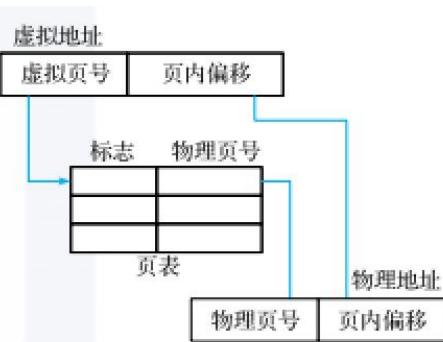


图 2.4 页式存储管理的地址转换过程

段页式则结合了两者的特点（有点类似多级页表？）

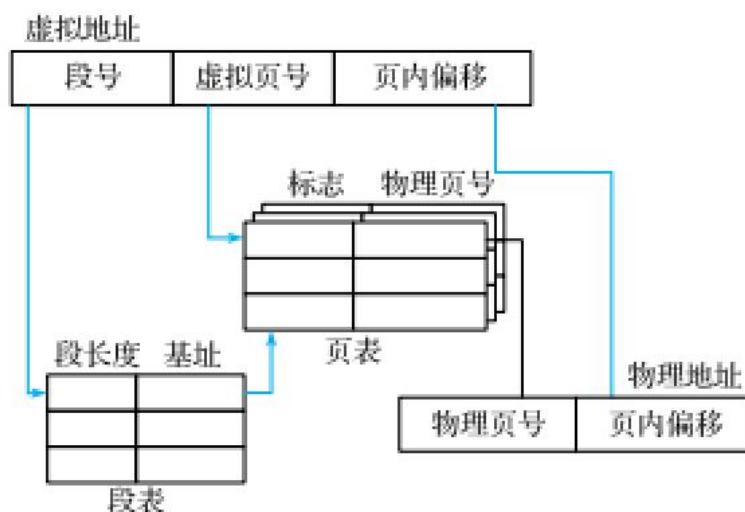


图 2.5 段页式存储管理的地址转换过程

2.3.3 运行级别演变

不同运行级别下，程序的运行资源不同

2.4 指令系统组成

指令系统由若干条指令和其操作对象组成。每条指令描述一个操作，主要包括操作码和操作数。操作码指示功能，操作数指示对象

2.4.1 地址空间

处理器可访问的地址空间包括寄存器空间和系统内存空间。

寄存器空间：通用寄存器、专用寄存器（e.g. 浮点?））、控制寄存器

系统内存空间：包括IO空间和内存空间

根据指令使用数据的方式，指令系统可分为堆栈型、累加器型和寄存器型（包括R-R和R-M）

2.4.2 操作数

尾端：最高有效字节的地址较小的是大尾端，反之为小尾端

以数0x123456为例，

大尾端：

数据	0x12	0x34	0x56
地址	0	4	8

小尾端

数据	0x56	0x34	0x12
地址	0	4	8

寻址方式：指令中如何表示要访问的内存地址

表 2.4 常用寻址方式介绍

寻址方式	格式	含义
寄存器寻址 (Register)	ADD R1, R2	$\text{regs}[R1] = \text{regs}[R1] + \text{regs}[R2]$
立即数寻址 (Immediate)	ADD R1, #2	$\text{regs}[R1] = \text{regs}[R1] + 2$
偏移量寻址 (Displacement)	ADD R1, 100(R2)	$\text{regs}[R1] = \text{regs}[R1] + \text{mem}[100 + \text{regs}[R2]]$
寄存器间接寻址 (Reg. Indirect)	ADD R1, (R2)	$\text{regs}[R1] = \text{regs}[R1] + \text{mem}[\text{regs}[R2]]$
变址寻址 (Indexed)	ADD R1, (R2+R3)	$\text{regs}[R1] = \text{regs}[R1] + \text{mem}[\text{regs}[R2] + \text{regs}[R3]]$
绝对寻址 (Absolute)	ADD R1, (100)	$\text{regs}[R1] = \text{regs}[R1] + \text{mem}[100]$
存储器间接寻址 (Mem. Indirect)	ADD R1, @ (R2)	$\text{regs}[R1] = \text{regs}[R1] + \text{mem}[\text{mem}[\text{regs}[R2]]]$
自增量寻址 (Autoincrement)	ADD R1, (R2)+	$\text{regs}[R1] = \text{regs}[R1] + \text{mem}[\text{regs}[R2]], \text{regs}[R2] = \text{regs}[R2] + d$
自减量寻址 (Autodecrement)	ADD R1, -(R2)	$\text{regs}[R2] = \text{regs}[R2] - d, \text{regs}[R1] = \text{regs}[R1] + \text{mem}[\text{regs}[R2]]$
比例变址寻址 (Scaled)	ADD R1, 100(R2)(R3)	$\text{regs}[R1] = \text{regs}[R1] + \text{mem}[100 + \text{regs}[R2] * \text{regs}[R3]]$

2.4.3 指令操作和编码

从功能上，指令可以分为四类**运算指令、访存指令、转移指令和系统管理指令**

2.5 RISC指令系统比较

2.6 C语言的机器表示

过程调用（传参，栈帧分配，栈帧释放，执行返回）

流程控制

空间布局（不同变量映射到地址空间的不同段）

第3章 特权指令系统

3.1 特权指令系统简介

用户态子集外的操作系统专用的特权态。

(1) 运行模式定义和切换

(2) 虚拟存储管理

(3) 异常与中断处理

(4) 控制状态寄存器

3.2 异常与中断

3.2.1 异常分类

从来源，异常可分为6种：

- 外部事件：CPU核外事件
- 指令执行错误：INE，除零，ALE，ADEF，ADEM等
- 数据完整性
- 地址转换：page fault
- 系统调用和陷入
- 需要软件修正的运算

3.2.2 异常处理流程

6步：**A 异常处理准备（例如记录异常的指令PC，调整权限），B 确定异常来源，C 保存执行状态，D 执行异常处理，E 恢复执行状态，F 返回正常执行流**

精确异常：异常指令前的指令均已执行完，之后的指令均未执行

异常嵌套：加入优先级

3.2.3 中断

中断传递机制：

(1) **中断线**：直接将中断源连接到处理器。

(2) **消息中断 (MSI)**：以数据的形式在总线上传递（包？）

向量化中断：

LA中，13个中断视为64~76号异常

中断优先级：

正常运行时处于最低优先级，更高优先级中断发生时可以抢断低优先级中断处理过程

中断使能控制的原子修改

3.3 存储管理

虚拟存储原理：

- 多进程环境下统一的编程空间
- 多进程环境下的保护和共享
- 支持大于实际物理地址的编程空间

存储管理部件**MMU**支持虚实地址转换

TLB (Translation Lookaside Buffer)：转换后援缓冲器/页表缓存/快表

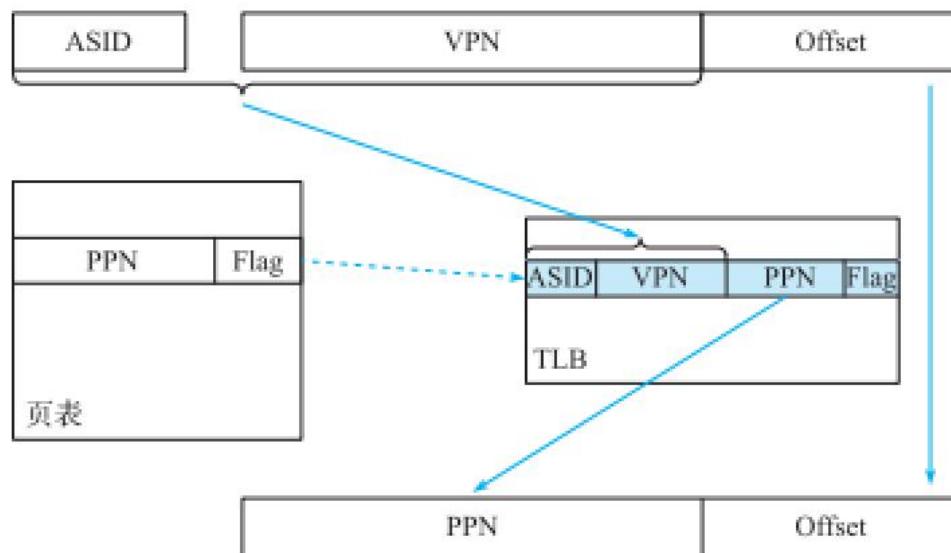


图 3.3 包含 TLB 的地址转换过程

LoongArch中的地址空间与翻译：

直接映射、窗口映射、TLB映射

TLB：一项TLB管两页

VPPN	PS	G	ASID	E
PPN0	RPLV0	PLV0	MAT0	NX0 NR0 D0 V0
PPN1	RPLV1	PLV1	MAT1	NX1 NR1 D1 V1

图 3.4 LoongArch64 指令系统中 TLB 表项结构

加速多级页表遍历：

LDIR rd, rj, level: 装载页表目录基址

LDPTE rj, seq: 装载页表表项

例子：

```
int *arr = (int *)malloc(0x1000);
for (int i = 0; i < 1024; i++)
    arr[i] = 0
```

分析异常次数
(Refill & Store invalidation)

TLB性能分析和一些优化：

防止缓冲区溢出攻击

硬件性能优化

软件性能优化

第4章 软硬件协同

指令系统除了指令和硬件资源的定义，还包含资源的使用方式约定。与二进制程序相关的约定为**ABI**（应用程序二进制接口）

4.1 应用程序二进制接口

ABI定义了应用程序二进制代码中数据结构和函数模块的格式与访问方式。硬件上并不强制要求这些内容。

ABI规范什么？

- 数据表示与对齐
- 寄存器使用
- 函数调用
- 栈帧布局
- 目标文件和可执行文件格式
- ...

4.1.1 寄存器约定

例如寄存器编号、助记符、使用约定

例如LA的寄存器约定：

表 4.2 LoongArch 整数通用寄存器约定

寄存器编号	助记符	使用约定
0	zero	总是为 0
1	ra	子程序返回地址
2	tp	Thread Pointer, 指向线程私有存储区
3	sp	栈指针
4-11	a0-a7	子程序的前八个参数
4-5	v0-v1	v0/v1 是 a0/a1 的别名，用于表示返回值
12-20	t0-t8	不需保存的暂存器
21	Reserved	暂时保留不用
22	fp	Frame Pointer, 栈帧指针
23-31	s0-s8	寄存器变量，子程序使用需要保存和恢复

4.1.2 函数调用约定

LP64

- 定点参数由a0~a7传递，超过8个用栈传递
- 浮点参数通过浮点寄存器传递，8个
- 结构体视为双字序列，用寄存器传递；超过两个双字时在栈上用指针传递
- 超过两个双字的返回值，用a0传指针，值在栈上
- 栈16字节对齐

4.1.3 进程虚拟地址空间

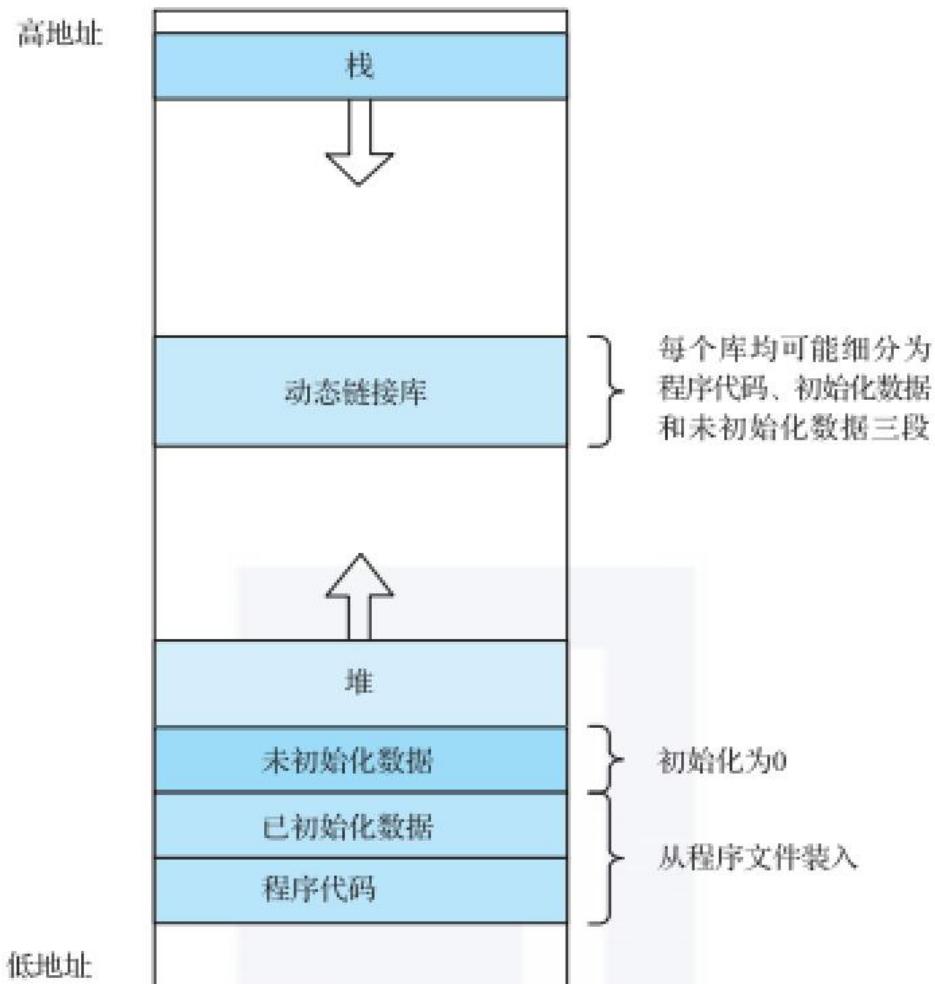


图 4.5 C 程序的典型虚拟内存布局

4.1.4 栈帧布局

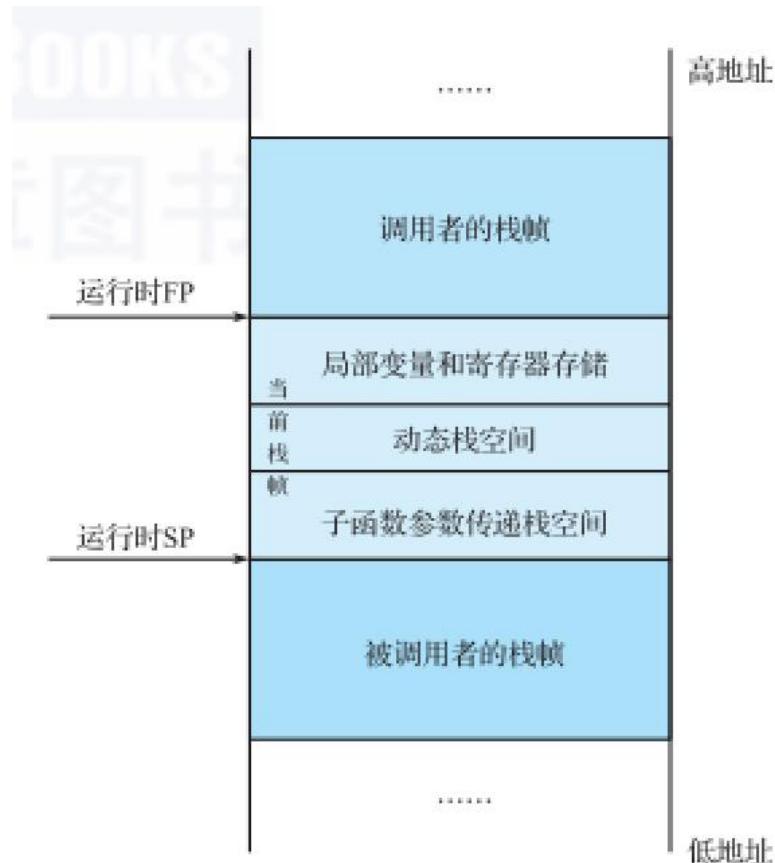


图 4.7 使用帧指针寄存器的栈帧布局

4.2 六种常见的上下文切换场景

- 函数调用：用户主动发起的上下文改变，遵循ABI的约定
- 异常和中断（被动触发）：保存和恢复所有可能被改变的用户可见上下文
- 系统调用（主动调用）
- 进程切换：保存不能多进程共享的处理器状态
- 线程切换
- 虚拟机切换

表 4.5 六种上下文切换场景

场景	上下文切换时保存和恢复的内容
函数调用	部分寄存器（包括栈帧相关的\$sp 和 \$fp）、返回地址
中断和异常	（通常情况）全部定点寄存器、异常现场信息、异常相关信息
系统调用	部分定点寄存器（包括栈帧相关寄存器）、异常现场信息
进程	全部用户态寄存器、页表基址等控制寄存器、当前 PC 等相关信息
线程	全部用户态寄存器、TLS、当前 PC 等相关信息
虚拟机	虚拟 CPU 状态（寄存器、必要的特权资源等）

4.3 同步与通讯

现代操作系统的关键特性：多任务

并发访问可能来自于：多处理器，中断引起的并发，多线程并发

用LL/SC指令实现锁机制

LL指令：load，设置LL bit，检测访问的物理地址是否被修改或者可能被修改，检测到则将LL bit清除

SC指令：带条件存储，当LL bit为0时，SC不会完成存储操作，而是把保存值的源寄存器清零

原子加:

```
atomic_inc:  
    ll.w    $t0, $a0, 0  
    addi.w   $t0, $t0, 1  
    sc.w    $t0, $a0, 0  
    beqz    $t0, atomic_inc  
    add.w    $a0, $t0, $zero      # return $a0  
    jr      $ra
```

test&set:

```
la.local    $a0, lock  
test_and_set:  
    ll.w      $v0, $a0, 0  
    li       $t0, 0x1  
    sc.w    $t0, $a0, 0  
    beqz    $t0, test_and_set
```

self spin:

```
la.local    $a0, lock  
selfspin:  
    ll.w      $t0, $a0, 0  
    bnez    $t0, selfspin  
    li       $t0, 0x1  
    sc.w    $t1, $a0, 0  
    beqz    $t1, selfspin  
<critical section>  
    st.w      $zero, lock
```

第5章 计算机组成原理和结构

5.1 冯·诺依曼结构

本质特征：存储程序和指令驱动

主要特点：

(1) **五部分**：存储器、运算器、控制器、输入设备、输出设备

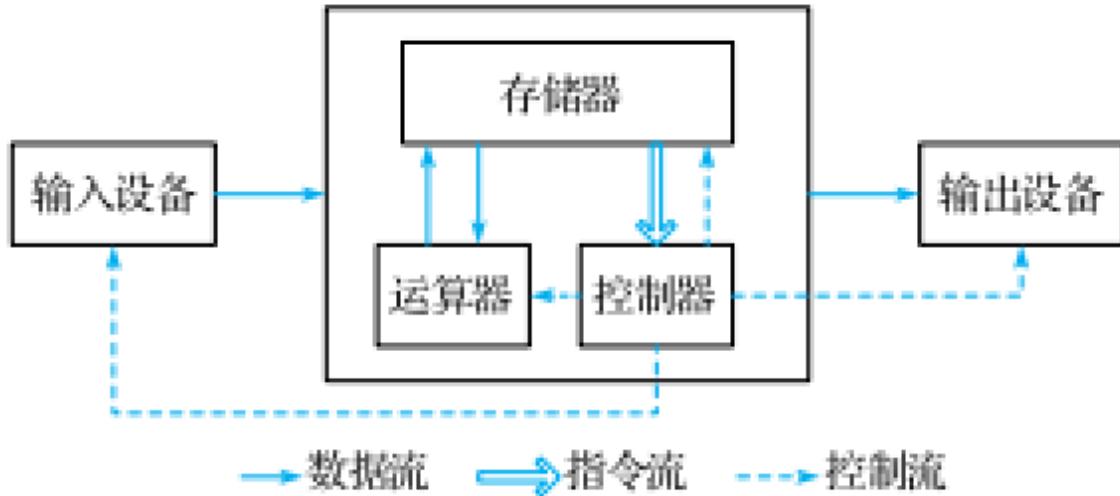


图 5.1 冯·诺依曼计算机体系结构

- (2) 存储器按地址访问，线性编址
- (3) 存储程序，即数据和指令不加区分地存储在同一个存储器
- (4) 控制器通过执行指令发出控制信号
- (5) 运算器为中心->改进为以存储器为中心

5.2 计算机的组成部件

物理上分为三大部分：CPU（运算器、控制器、Cache），内存，IO设备

三大部分之间的通信和同步对性能至关重要

5.2.1 运算器

执行运算指令的部件（算术和逻辑）

5.2.2 控制器

控制指令流和每条指令的执行

程序计数器 (PC) 存放当前指令执行的地址，指令寄存器 (IR) 存放当前正在执行的指令

通过控制器设计提升处理器性能的技术主要包括：

- 指令流水线
- 乱序
- 超标量（多车道）
- 转移猜测
- 预取以及高速缓存

(1) 指令流水线：堆叠，通过提升单位时间内执行的指令数来提升性能

(2) 乱序执行：

三类相关：数据相关 (RAW, WAW, WAR)，控制相关，结构相关

保留站/发射队列：等待操作数准备完毕

重排序缓冲 (ROB)：恢复顺序

重命名寄存器：消除假性的相关

(3) 超标量：允许流水线在每一阶段同时处理多条指令，结构复杂度和发射宽度平方成正比

(4) 转移猜测：依据历史行为，在取指或译码阶段预测方向

e.g. 转移历史表BHT：

用PC低位索引，不进行地址比较检查

每项1位记录同一项上次转移是否成功（例如：`for (i = 0; i < 10; i++)` 的转移模式为
(1111111110)）

两位的BHT表：只有连续两次猜错才会改变猜测方向

(5) 预取和高速缓存：略

3.2.3 存储器

又称**主存储器或内存**，一般用DRAM实现。CPU可以直接访问，IO设备也频繁交换数据

存储系统按速度可分为三个层次：**Cache、主存储器、辅助存储器**

Cache一般采用比DRAM速度快但容量小的SRAM实现。

现代计算机中还有少量ROM用来存放引导程序和BIOS

存储介质：SRAM（易失、快、贵），DRAM（易失、慢、便宜）、闪存（非易失、快、贵）、磁性存储介质（非易失、慢、便宜）

存储层次：

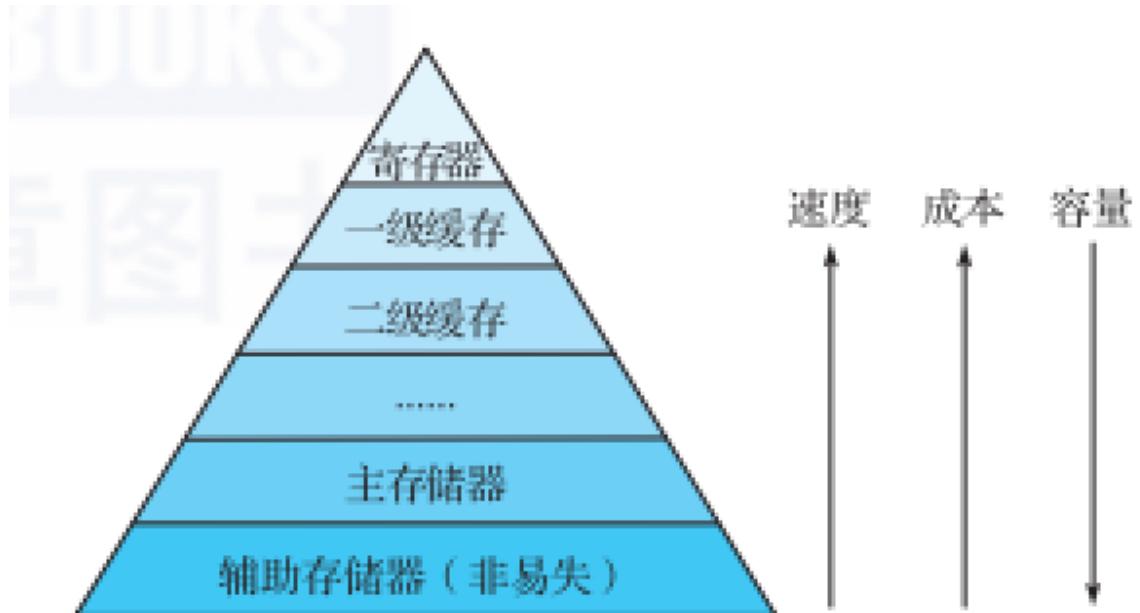


图 5.2 存储层次

局部性原理：时间局部性、空间局部性

访问速度：

- 寄存器：一拍多个

- L1: 1~4拍
- L2: 10~20拍
- L3: 40~60拍
- 内存: 100~200拍

高速缓存:

访存延迟成为以存储器为中心的冯诺依曼结构的主要瓶颈

AMAT: Average Memory Access Time, $= \text{HitTime} + \text{MissRate} * \text{MissPenalty}$
(MissPenalty指查Cache发现缺失后访存的时间)

内存:

CPU性能的决定性因素

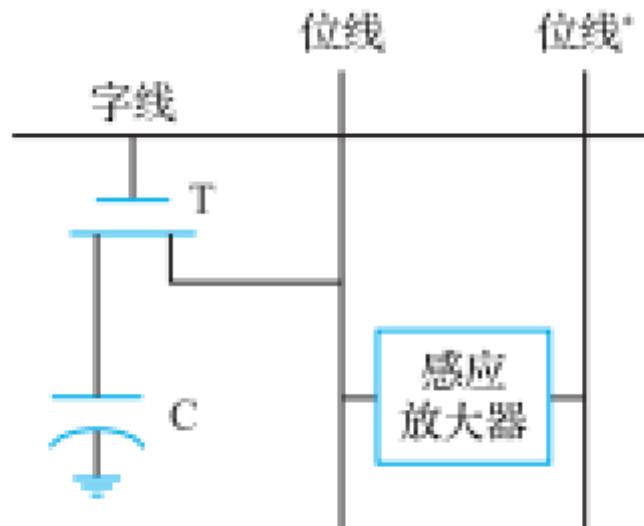


图 5.3 DRAM 的单元读写原理

电容决定逻辑值，字线根据地址译码连接同一字的不同位，位线是读写的数据连接不同字的同一位

读写过程:

- 行激活: Bank和Row地址译码
- 读写数据
- 预充: 关闭字/位线, 行缓冲的数据写回存储阵列

预充的策略:

关行 Close Page: 每次读写后自动预充

开行 Open Page: 地址切换时预充

提升访存性能的方法:

- 利用行缓冲的局部性
- 利用Bank级并行性 (多个Bank可以流水并行执行)
- 内存控制器调度

IO设备：

DMA: Directly Memory Access

处理器和IO设备同步的方式：查询和中断

GPU：显存和内存合并时，为协处理器，不属于IO设备

5.3 计算机系统硬件结构演进

四片结构：CPU+芯片组，CPU+GPU+南桥+北桥，内存在北桥上，北桥是系统连接的枢纽，南桥接低速IO

三片结构：GPU被集成到北桥（CPU+北桥+南桥），内存控制器集成到CPU（CPU+弱北桥+南桥）

两片结构：GPU进一步集成到CPU（CPU+南桥），或者MMU集成到CPU，GPU集成到北桥

单片结构：SoC

5.4 处理器和IO通信

CPU——内存：CPU通过Load/Store访问，高带宽低延迟

IO——内存：IO通过DMA访问，较高带宽高延迟

CPU——IO：CPU通过PIO访问IO，低带宽高延迟

5.4.1 IO寄存器寻址

读取IO特定的寄存器获取设备状态，写入值以驱动设备

寻址方式：独立的IO指令，或者内存映射统一寻址（使用Load/Store）

5.4.2 CPU和IO设备间的同步

查询：CPU不断读取IO的状态寄存器

中断：设备完成后以中断的方式通知CPU

5.4.3 存储器和IO设备间的通信（CPU和IO的通信）

PIO (Programming Input/Output) : CPU主导，CPU把数据读入到CPU内部的寄存器，再写到指定位置

DMA (Direct Memory Access) : 在内存和外设间开辟直接的数据传输通道

第6章 计算机总线接口技术

6.1 总线概述

总线是什么：连接计算机各部件，用于数据传输的通信系统

本质作用是进行数据交换

总线规范具有以下层次：

机械层、电气层、协议层、架构层

6.2 总线分类

按数据传输方向：单向、双向（半双工、全双工）

按数据组织方式：并行（一起传要求高，提频困难，PCI、HT），串行（字节按位传，PCIe、SATA）
串行取代并行：速率高、引脚少

按数据握手方式：无/Valid-Ready/Credit（令牌）

按连接方式：共享信号（PCI）、点对点（PCIe）

按时钟实现方式：全局时钟（e.g. PCI），源同步（时钟随数据一起发送，如HT；时钟嵌到数据中，如PCIe）

按总线实现位置：片上总线（CPU内）、内存总线（e.g. DDR3）、系统总线、IO总线

总线举例：

6.3 片上总线

连接处理器核、内存控制器、IP核，支撑基本的读写操作

与片外相比，片上总线引线资源丰富，全局时钟相对容易实现，不需要复杂的物理层转换，不使用三态信号

性能相关因素：频率、位宽、带宽利用率（数据传输时间占总时间的均值）

AXI总线架构：

五个通道

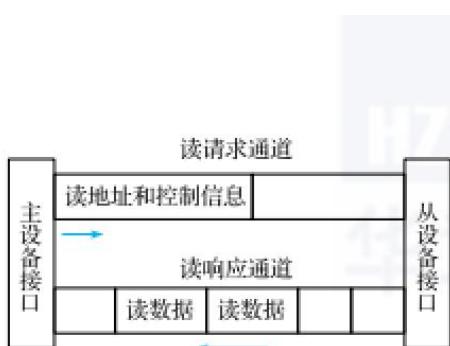


图 6.2 读事务架构



图 6.3 写事务架构

总线事务（transaction）：一次完整的读写过程

传输（transfer）：传输一个周期的数据，在valid和ready同时为高的周期完成一次传输

读事务: AR读请求 (读地址) 通道, R读返回 (读响应) 通道, 读请求握手, 读返回握手, RLast

写事务: AW写请求 (写地址) 通道, W写数据通道, B写响应通道, 写请求握手, 写数据 (WLast) , 写响应

关键信号:

ID: 支持读写乱序, 每个事务都有ID标签, 同ID的事务必须按序, 不同的可以乱序

Burst/Len/Size: 突发传输类型 (FIXED地址不变、INCR地址递增、WRAP地址递增回绕) 、突发传输长度、突发传输单位

WSTRB: 写掩码 (拉高有效)

RESP: 读/写响应状态

ARLOCK/AWLOCK: 原子性操作 (00普通, 01独占, 10加锁)

ARPROT/AWPROT: 访问保护 ([0]普通/特权, [1]安全/非安全, [2]数据/指令)

ARCACHE/AWCACHE: 缓存控制 ([0]可写缓冲, 未到达目的即回响应; [1]可缓存, 读预取, 写合并; [2]读分配, 缺失则分配缓存块; [3]写分配, 同[2])

AXI互连

AML: 单主多从

ASL: 单从多主

共享式 (一堆Master —— ASL —— AML —— 堆从) 连线少, 交换式 (M——AML——ASL——S) 带宽高

AHS: 高性能系统总线

ASB: 第一代AMBA总线

APB: 本地二级总线

性能相关因素:

- 频率
- 位宽
- 带宽利用率 (传输时间占比)

6.4 内存总线

内存总线规范由**JEDEC**制定, 包含**机械层、电气层和协议层**

内存组织:

- Row, Column
- Bank (并联拓展数据位宽)
- Rank (并联扩展存储容量, 通过片选区分)

6.4.1 机械层

信号分类：

- CKp/Ckn，内存系统工作的参考时钟以及地址命令信号的采样时钟
- 地址命令信号：BA, ADDR, RAS#, CAS#, WE#, CKE, CS, ODT（上升沿有效，与时钟沿错开以正确采样）
- 数据以及数据采样：DQSp/DQSn, DQ, DM

接口信号：

表 6.2 双面 DDR3 UDIMM 内存条的接口信号列表

引脚名称	描述	引脚名称	描述
A0-A15	SDRAM 地址线	SCL	EEPROM I2C 总线时钟
BA0-BA3	SDRAM Bank 地址	SDA	EEPROM I2C 总线数据线
RAS#	SDRAM 行地址选通	SA0-SA2	EEPROM I2C 从设备地址
CAS#	SDRAM 列地址选通	V _{DD}	SDRAM Core 电源
WE#	SDRAM 写使能	V _{DDQ}	SDRAM IO 输出电源
S0#-S1#	SDRAM 片选信号	V _{refDQ}	SDRAM IO 参考电源
CKE0- CKE1	SDRAM 时钟使能信号	V _{refCA}	SDRAM 命令地址参考电源
ODT0- ODT1	SDRAM 终端匹配电阻控制信号	V _{ss}	电源地信号
DQ0- DQ63	DIMM 内存数据线	V _{DQSPD}	EEPROM 电源
CB0- CB7	DIMM ECC 数据线	NC	空闲引脚
DQS0- DQS8	SDRAM 数据选通线（差分对的正沿）	TEST	测试引脚
DQS0- DQS8	SDRAM 数据选通线（差分对的负沿）	RESET#	复位引脚
DM0- DM8	SDRAM 数据 Mask 线	EVENT#	温度传感器引脚（可选）
CK0- CK8	SDRAM 时钟信号线（差分对的正沿）	V _{tt}	SDRAM IO 终端匹配电阻电源
CK0- CK8	SDRAM 时钟信号线（差分对的负沿）	RSVD	保留

6.4.2 电气层

内存电压、电平标准、信号斜率、时钟抖动范围

6.4.3 协议层

包含上电的时序、状态转换

读操作：

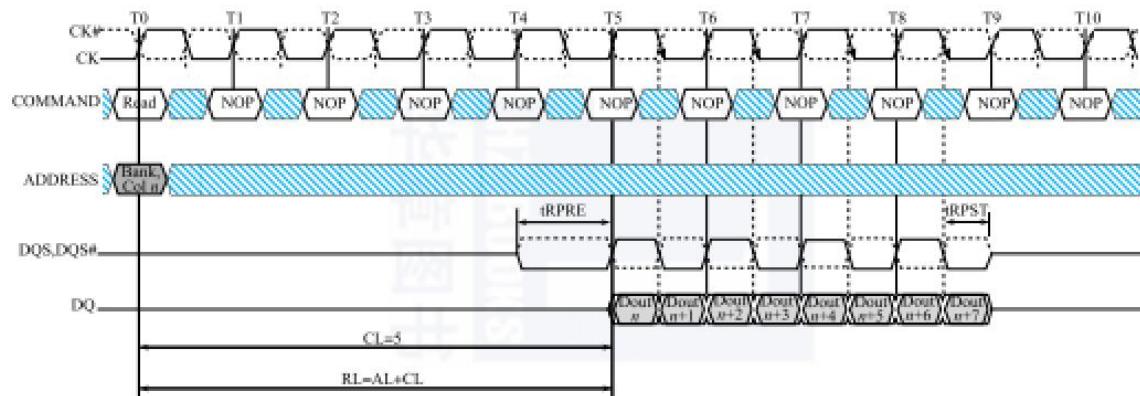


图 6.13 DDR3 SDRAM 读时序

假设当前行已激活，T0时发出读命令，T5时开始从DQ返回数据；由于是双沿采样，共返回8次数据，同时DQS也被驱动，与DQ同步。同时DQS有 preamble，供内存控制器过滤有效时钟

写操作:

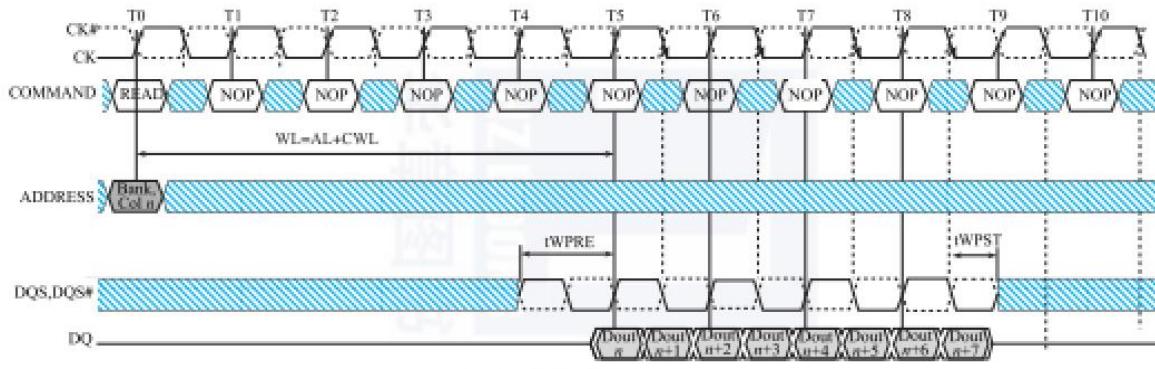


图 6.14 DDR3 SDRAM 写时序

类似，略；需要注意此时DQS与DQ是中心对齐

6.4.4 内存控制器

地址译码（物理地址->SDRAM地址[片选、bank、行、列]）

命令调度、时序控制、物理接口phy

6.5 系统总线 (to be finished)

处理器与桥片的连接、处理器间的连接

6.6 IO总线 (to be finished)

PCI总线信号：

- 地址&数据
- 控制
- 仲裁
- 时钟复位
- 中断（非强制）

PCI -> PCIe：改用串行差分线；引入网络报文，在出错时重试；引入虚通道

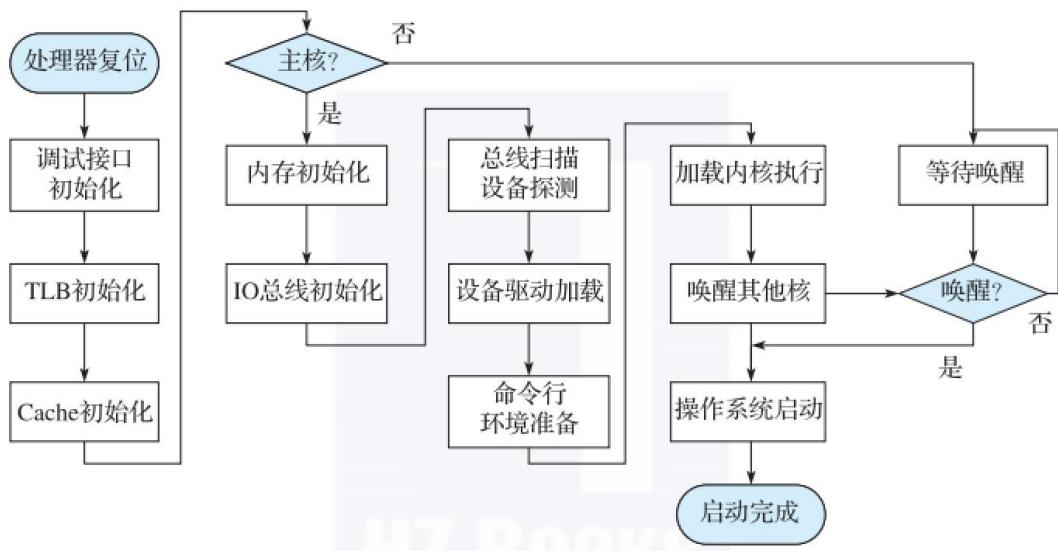
第7章 计算机系统启动过程分析

系统启动：从复位到系统可用

什么是初始化？寄存器从不确定到确定，模块状态从无序到有序

什么需要初始化？CPU、内存、IO接口

怎么初始化？核内到核外，片内到片外



7.1 处理器核初始化

处理器复位

将处理器内部部分寄存器状态设置成预设值

复位后只有取指通路是可用的

首条指令：初始化控制寄存器、设置软件的通用寄存器（BIOS用的）

外部调试接口初始化

蜂鸣器、数码管、串口控制器

TLB初始化

对于BIOS，基本使用非映射空间，无需TLB

过程：清空每个TLB项

Cache初始化

复位后BIOS刚启动时，处理器从非缓存空间开始执行

上千拍完成一条指令的取指和执行

处理器核初始化之后，除了串口和BIOS接口，多数门窗都还没开

7.2 总线接口初始化

内存总线接口初始化

内存接口的初始化与核内部件的初始化差别大，前者主要对接口控制做初始化，后者的初始化只需将内容无效

内存接口初始化内容：

- 获取大小、类型、频率、延迟等信息
- 根据信息对控制器和内存进行配置（把值填到相应的寄存器中）
- 可能还有信号训练
- 不关心内容！（除了ECC内存）

IO总线初始化

根据不同的IO总线需求进行针对性的初始化

7.3 设备探测及驱动加载

PCI协议下，IO的系统空间可分为**配置空间**、**IO空间**、**Memory空间**。

通过配置空间探测总线设备，IO、Memory空间才是真正使用设备功能时的地址空间。对于IO和Memory同一编址的CPU，区别不大

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也不会冲突

设备探测：使用配置空间，以HT的配置空间为例

Type 0	39 FDFEh	24 23 保留	16 15 设备号	11 10 功能号	8 7 偏移	0
Type 1	39 FDFFh	24 23 总线号	16 15 设备号	11 10 功能号	8 7 偏移	0

图 7.3 HyperTransport 总线配置访问的两种类型

HT配置访问为两种类型。只需要在总线号、设备号、功能号的位置进行枚举，就能遍历整个总线

地址空间分配：

基址寄存器（BAR），在设备的配置空间中，每个设备有6组

31 可写位	$n \ n-1$ 只读0位	4 可预取标识	3 2 1 0 64位 IO 标识
-----------	-------------------	------------	----------------------------

图 7.5 BAR 的寄存器定义

用于配置IO设备需要使用的内存空间

PCI设备的探测和驱动加载是一个递归的过程，大致算法如下：

- 将初始总线号、设备号、功能号设为0
- 访问当前总线号、设备号、功能号访问一个配置空间地址，检查该**配置空间**中的设备号
- 全1或全0表示无设备
- 若有效，检查每个BAR所需的空间
- 若为多功能设备，枚举每个功能
- 若为桥设备，给桥配置一个新的总线号，再使用新总线号、设备号0、功能号0开始探测
- 若设备号为31且总线号为0，表示扫描结束；若总线号非0，返回上一层递归调用

7.4 多核启动

一个核执行主流程，其他核仅负责私有化部件的初始化

通过核间通信机制进行同步

多核唤醒：

内核启动时，会将从核逐一加入系统管理

(将从核需要运行的程序指针、参数放入信箱寄存器；从核取出这些东西执行)

在BIOS完成上述工作后，将Kernel从硬盘搬运到内存展开，并跳转到内核入口，后续由内核负责

第8章 运算器设计

8.1 二进制与逻辑电路

8.1.1 数的表示

定点数表示：

原码：“符号—数值”，例如 $+19 = (0,0010011)$ ， $-19 = (1,0010011)$ ，存在一个+0一个-0，

表示范围：

$$[-2^{n-1} + 1, 2^{n-1} - 1]$$

补码：

$$[X]_{\text{补}} = 2^n + X \pmod{2^n}, -2^{n-1} \leq X < 2^{n-1}$$

正数补码等于原码，负数补码等于原码按位取反加1

表示范围：

$$[-2^{n-1}, 2^{n-1} - 1]$$

溢出：超过表示范围，或者正+正得负，负+负得正

浮点数表示：

IEEE754

31	30	23	22	0
符号	阶码	尾数		
1	8	23		

a) 32位单精度格式

63	62	52	51	0
符号	阶码	尾数		
1	11	52		

b) 64位双精度格式

尾数用原码表示，规格化定义为 $1.\text{xxxx}$ ；阶码用译码表示，单静常量为127，双精常量为1023

一些特殊规定

表 8.1 IEEE 754 浮点数格式

	单精度				双精度			
	符号	阶码	尾数	值	符号	阶码	尾数	值
正无穷	0	255	0	∞	0	2047	0	∞
负无穷	1	255	0	$-\infty$	1	2047	0	$-\infty$
非数 (NaN)	0 或 1	255	$\neq 0$	NaN	0 或 1	2047	$\neq 0$	NaN
规格化非 0 正数	0	$0 < e < 255$	f	$1.f \times 2^{e-127}$	0	$0 < e < 2047$	f	$1.f \times 2^{e-1023}$
规格化非 0 负数	1	$0 < e < 255$	f	$-1.f \times 2^{e-127}$	1	$0 < e < 2047$	f	$-1.f \times 2^{e-1023}$
非规格化非 0 正数	0	0	$f \neq 0$	$0.f \times 2^{-126}$	0	0	$f \neq 0$	$0.f \times 2^{-1022}$
非规格化非 0 负数	1	0	$f \neq 0$	$-0.f \times 2^{-126}$	1	0	$f \neq 0$	$-0.f \times 2^{-1022}$
正 0	0	0	0	0	0	0	0	0
负 0	1	0	0	-0	1	0	0	-0

8.1.2 MOS晶体管工作原理

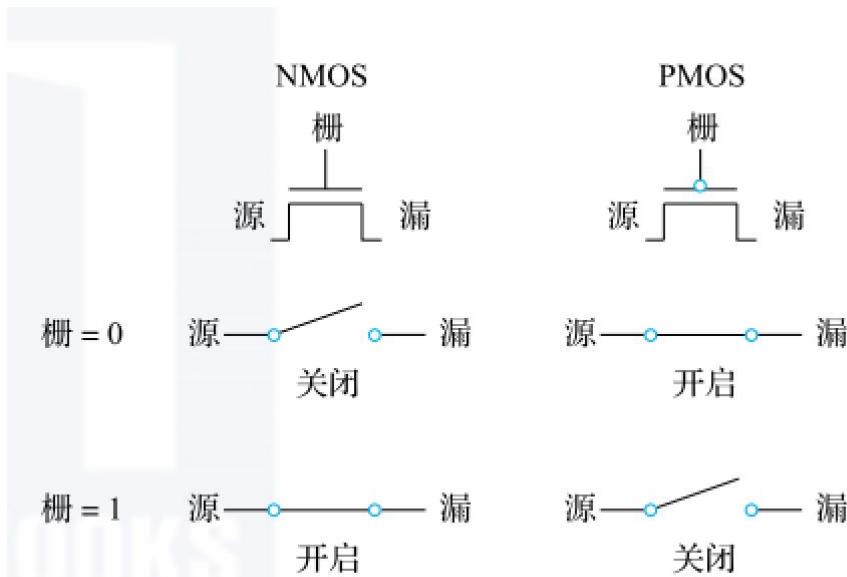
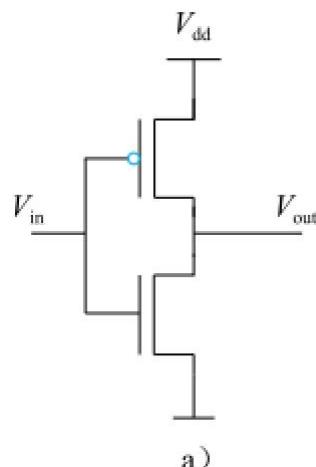


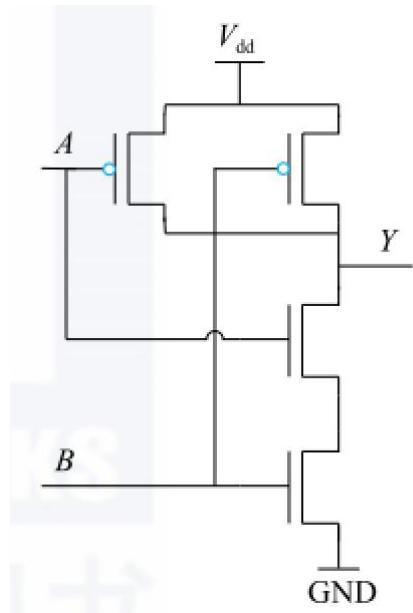
图 8.5 MOS 晶体管开关行为

CMOS逻辑电路

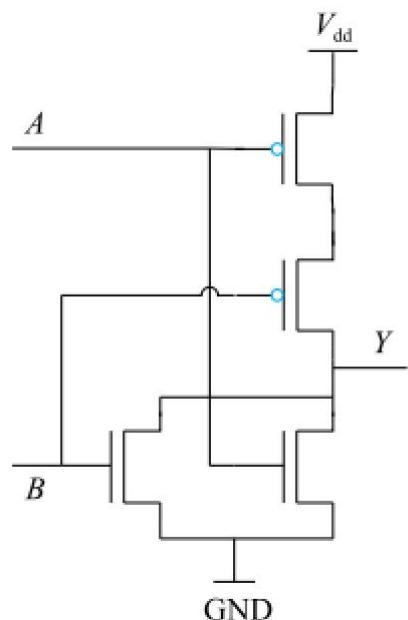
非门



与非门



或非门



传输门、D触发器

8.2 简单运算器设计

8.2.1 一位全加器

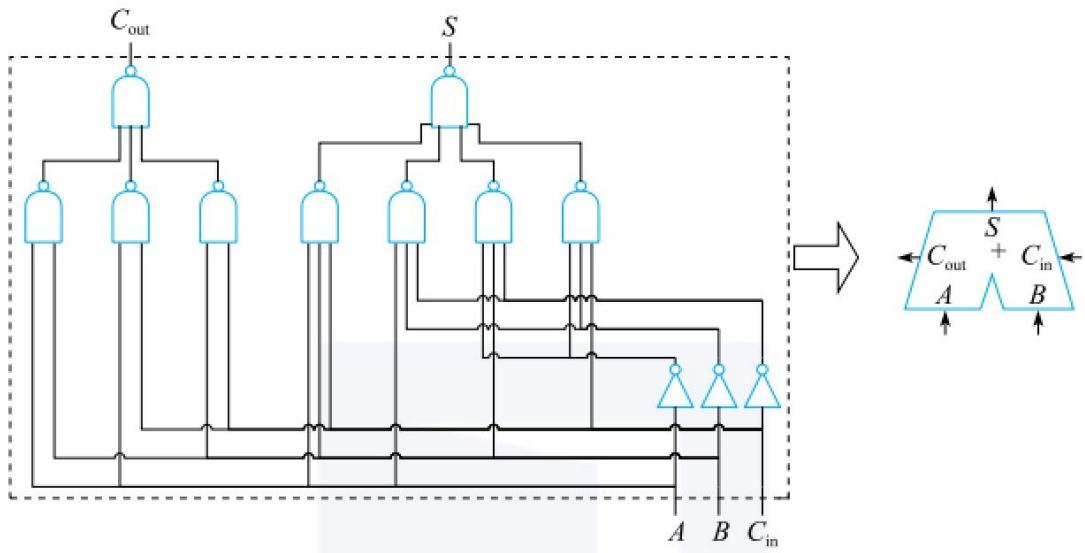


图 8.17 一位全加器逻辑电路图与示意图

延迟：

从A、B和Cin：到S三级，到Cout两级

8.2.2 行波进位加法器

把全加器串联

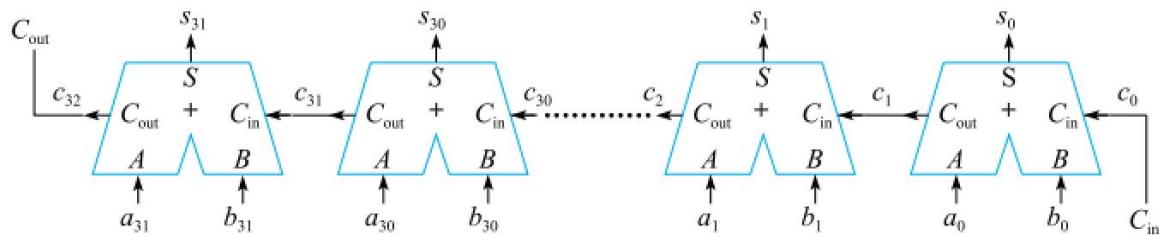


图 8.18 32 位行波进位加法器

从 $A_0 B_0 C_{in}$ 到 C_{31} : $31 \times 2 = 62$ 级

从 C_{31} 到 S_{31} : 3级

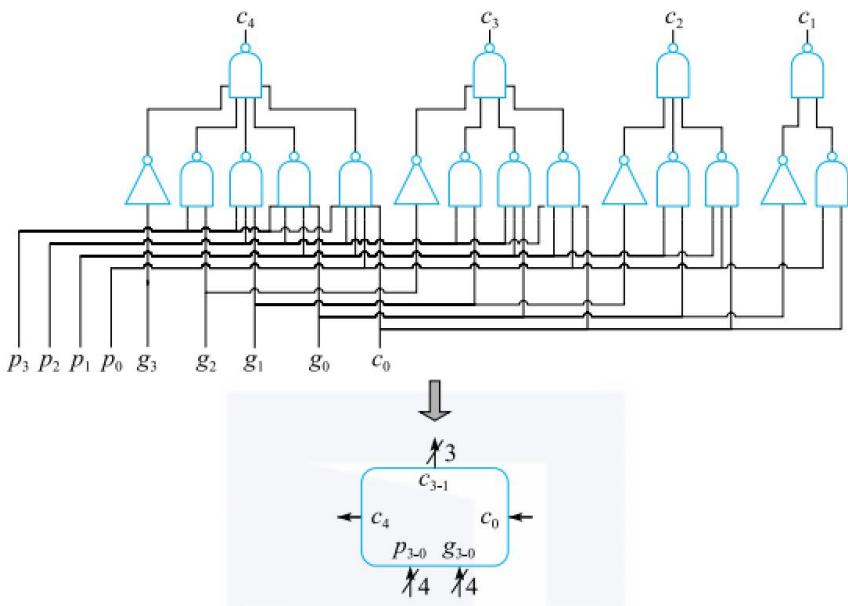
最大延迟65级

64位时，最大延迟为129级

8.2.3 先行进位加法器（重要）

首先看进位的生成逻辑

并行进位：



pg信号1级，进位2级

块内并行，块间串行：

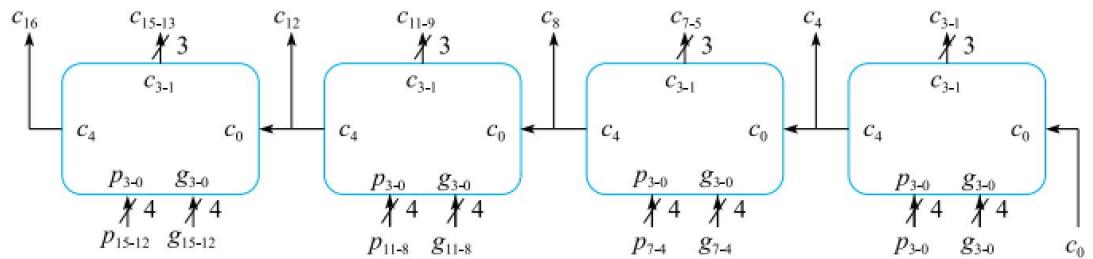
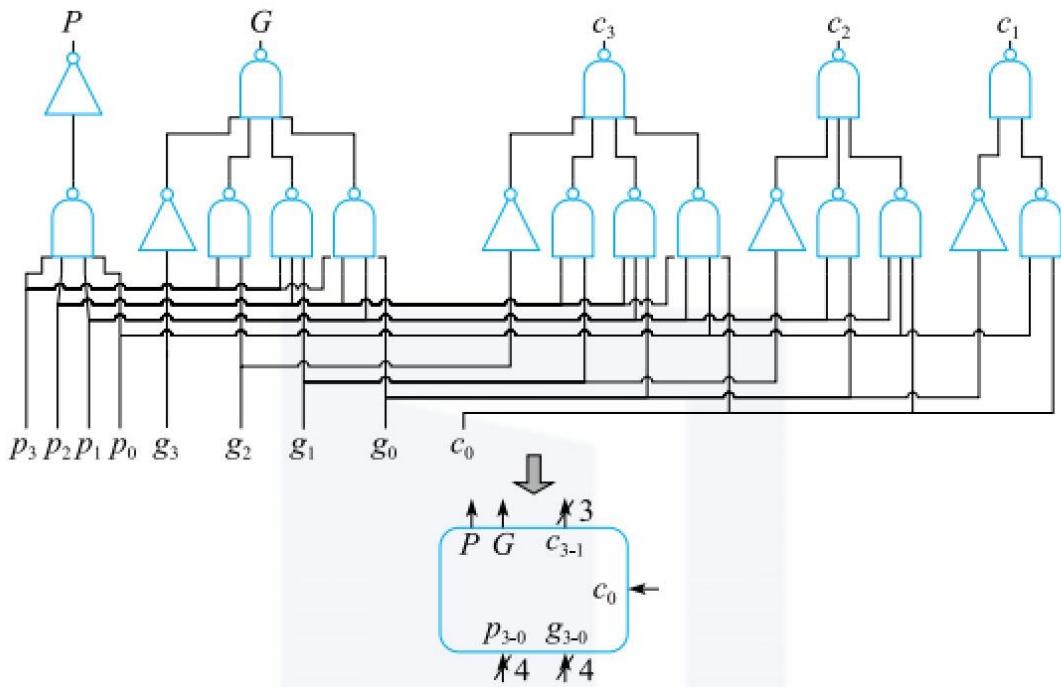


图 8.20 块内并行、块间串行的 16 位先行进位加法器的进位逻辑

pg1级， c_0 到 c_{16} 8级

块内并行，块间并行：

类似块内进位，定义块间的PG信号



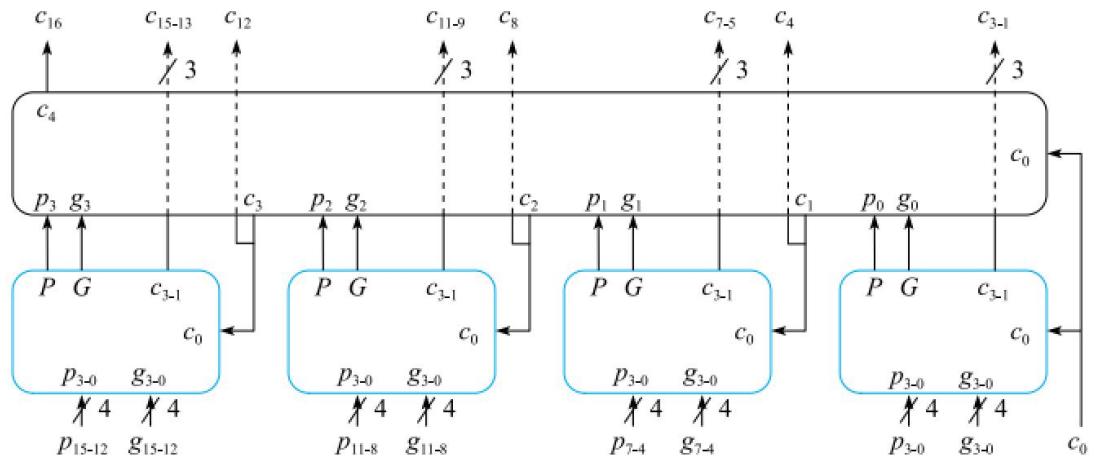


图 8.22 块内并行且块间并行的 16 位先行进位逻辑

假设 pg 信号级数为 0

- PG 信号生成：2 级
- c_1 到 c_3 ：2 级
- 最长： $c_0 - c_3 - c_{15-13}$, 共 6 级

8.2.4 补码减法算法

$$[A]_{\text{补}} - [B]_{\text{补}} = [A - B]_{\text{补}} = [A]_{\text{补}} + [-B]_{\text{补}}$$

其中 $[-B]_{\text{补}}$ 可以将 $[B]_{\text{补}}$ 按位取反再末位加 1

8.2.5 比较器与移位器

比较器：分析 $A - B$ 的结果，需要注意溢出情况

移位器：多路选择器

8.3 定点补码乘法器

8.3.1 补码乘法器

$$\begin{aligned} [X \times Y]_{\text{补}} &= -[X]_{\text{补}} \times (y_7 \times 2^7) + [X]_{\text{补}} \times (y_6 \times 2^6) + \cdots + [X]_{\text{补}} \times (y_0 \times 2^0) \\ &= [X]_{\text{补}} \times (-y_7 \times 2^7 + y_6 \times 2^6 + \cdots + y_0 \times 2^0) \end{aligned}$$

$\begin{array}{r} 1010 \times 1001 (-6 \times -7) \\ \hline 1010 \\ \times 1001 \\ \hline +11111010 \\ +0000000 \\ +000000 \\ -11010 \\ \hline \underline{00101010} (42) \end{array}$	$\begin{array}{r} 1010 \times 0101 (-6 \times 5) \\ \hline 1010 \\ \times 0101 \\ \hline +11111010 \\ +0000000 \\ +111010 \\ -00000 \\ \hline \underline{111100010} (-30) \end{array}$
---	--

图 8.27 补码乘法计算示例

注意做符号拓展，最后一项是减法

8.3.2 Booth乘法器

Booth一位乘

$$\begin{aligned}
 & (-y_7 \times 2^7 + y_6 \times 2^6 + \cdots + y_1 \times 2^1 + y_0 \times 2^0) \\
 &= (-y_7 \times 2^7 + (y_6 \times 2^7 - y_6 \times 2^6)) + (y_5 \times 2^6 - y_5 \times 2^5) + \cdots \\
 &\quad + (y_1 \times 2^2 - y_1 \times 2^1) + (y_0 \times 2^1 - y_0 \times 2^0) + (0 \times 2^0) \\
 &= (y_6 - y_7) \times 2^7 + (y_5 - y_6) \times 2^6 + \cdots + (y_0 - y_1) \times 2^1 + (y_{-1} - y_0) \times 2^0
 \end{aligned}$$

关键: $y_{i-1} - y_i$

表 8.5 Booth 一位乘运算规则

y_i	y_{i-1}	操作
0	0	不需要加 (+0)
0	1	补码加 $X(+[X]_{\text{补}})$
1	0	补码减 $X(-[X]_{\text{补}})$
1	1	不需要加 (+0)

$\begin{array}{r} 1010 \times 1001 (-6 \times -7) \\ \hline 1010 \\ \times 1001 \\ \hline -11111010 (10) \\ +1111010 (01) \\ +0000000 (00) \\ -11010 (10) \\ \hline +00000110 \\ +1111010 \\ +0000000 \\ +00110 \\ \hline \underline{100101010} (42) \end{array}$	$\begin{array}{r} 1010 \times 0101 (-6 \times 5) \\ \hline 1010 \\ \times 0101 \\ \hline -11111010 (10) \\ +1111010 (01) \\ -111010 (10) \\ +11010 (01) \\ \hline +00000110 \\ +1111010 \\ +000110 \\ +11010 \\ \hline \underline{111100010} (-30) \end{array}$
---	---

图 8.29 Booth 一位乘示例

需要注意，在算法开始的时候，要隐含的在乘数最右侧补一个 y_{-1} 的值

Booth两位乘：

关键： $y_{i-1} + y_i - 2y_{i+1}$

表 8.6 Booth 两位乘运算规则

y_{i+1}	y_i	y_{i-1}	操作
0	0	0	不需要加 (+0)
0	0	1	补码加 $X (+[X]_b)$
0	1	0	补码加 $X (+[X]_b)$
0	1	1	补码加 $2X (+[X]_b \text{ 左移})$
1	0	0	补码减 $2X (-[X]_b \text{ 左移})$
1	0	1	补码减 $X (-[X]_b)$
1	1	0	补码减 $X (-[X]_b)$
1	1	1	不需要加 (+0)

$$\begin{array}{r} 1010 \times 1001 (-6 \times -7) \\ \begin{array}{r} 1010 \\ \times \quad 1001 \\ \hline +11111010 \quad (010) \\ -11010 \quad (100) \\ \hline +11111010 \\ +00110 \\ \hline 100101010 \quad (42) \end{array} \end{array}$$

$$\begin{array}{r} 1010 \times 0101 (-6 \times 5) \\ \begin{array}{r} 1010 \\ \times \quad 0101 \\ \hline +11111010 \quad (010) \\ +111010 \quad (010) \\ \hline +11111010 \\ +111010 \\ \hline 111100010 \quad (-30) \end{array} \end{array}$$

图 8.30 Booth 两位乘示例

同样的，算法开始时在右侧补一个0

(结构 to be continued)

8.3.4 华莱士树 (to be continued)

基本思想：全加器能将三个数相加变成两个数相加

连线规则：

- 除了最高层的全加器，其他全加器的进位都要作为进位输出（到左侧）
- 接收等于进位输出位数的进位输入
- 若本层全加器不够接所有的进位输入，则传递到下一层接（不能往下接！）

第9章 指令流水线

9.1 时空图

多周期时空图：

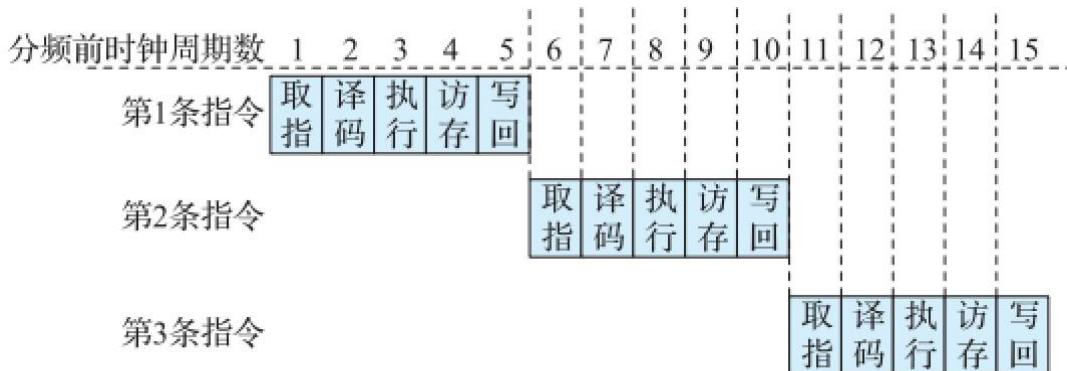
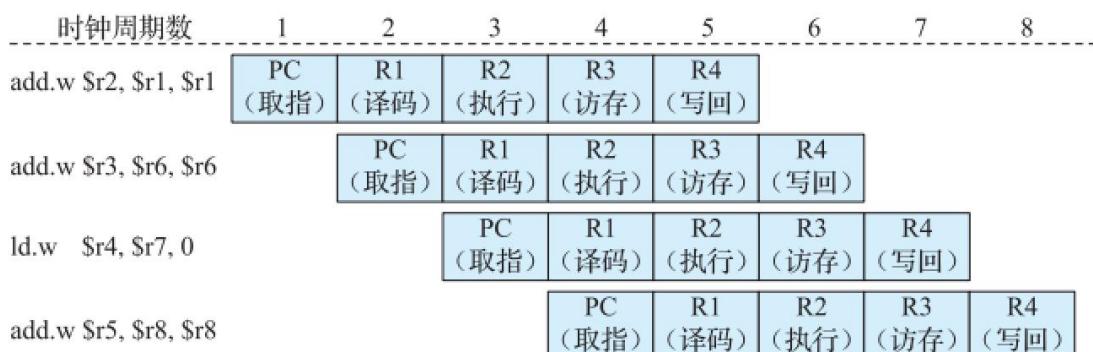


图 9.5 多周期处理器的流水线时空图

多周期处理器可以提高频率，但是每条指令执行的时间不能降低

简单的流水线时空图：



如果出现指令相关：

阻塞解决数据相关：



图 9.10 用阻塞解决数据相关的流水线时空图

前递解决数据相关（无法解决Load-Use型）：

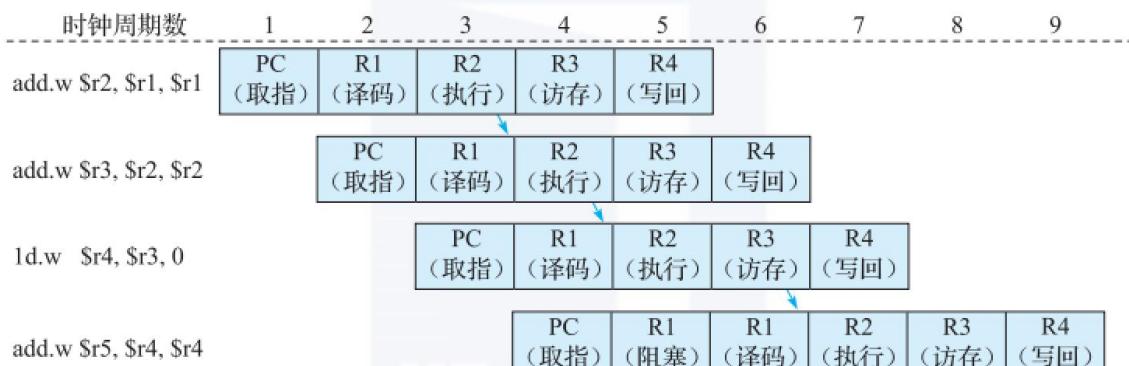
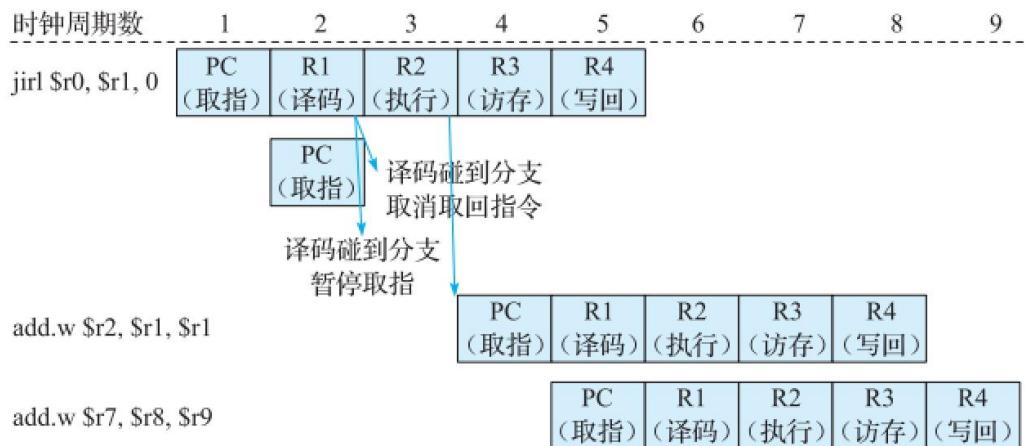
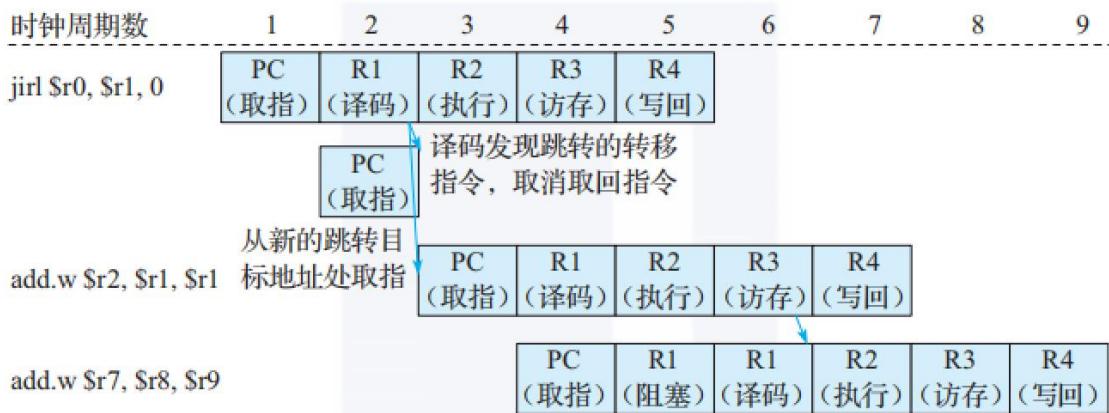


图 9.12 加入前递的数据相关时空图

控制相关 (假设在执行级判断) :



控制相关（假设在译码级判断）：



这里有一个假设，开始译码时，取指操作已经发出且不可取消

结构相关：多条指令同时使用同一个功能部件

9.2 异常和流水线

简单处理：

- 任何一级出现异常时，打上标记，到WB级再处理
 - 打上标记后不会进行任何动作
 - 外部中断作为取指异常（实验课作为译码异常？）

9.3 指令调度技术

编译器静态调度：使相关的指令距离尽可能远

循环展开

寄存器重命名消除假性的数据冲突

改变指令次序

增加发射宽度

9.4 提高流水线效率的技术

9.4.1 动态调度

基本思想：把相关的解决延后

把译码分成发射和读操作数两个阶段

基本做法：进入有序，执行可以乱序，结果（提交）必须有序

保留站：检查结构、数据的可用性，把使有序的指令能乱序执行

ROB：把乱序的结果恢复成有序提交

寄存器重命名：消除WAW、WAR冲突

9.4.2 多发射

让处理器每级流水线都可以同时处理多条指令

不仅要判断流水级间的相关性，还要判断流水级内的相关性

设计开销平方增加

9.4.3 转移猜测技术

基本原理：

猜测依据是当前的PC和性质（是否为转移指令）以及历史情况（局部和全局）

需要预测转移方向（是否跳转）和转移目标地址

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 10; j++) {
        ...
    }
```

1位BHT：

跳转方向	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0
BHT表值	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
猜测方向	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
猜测正确	0	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	0

2位BHT：

跳转方向	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0
BHT表值	00	01	10	11	11	11	11	11	11	10	11	11	11	11	11	11	11	11	11
猜测方向	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
猜测正确	0	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0

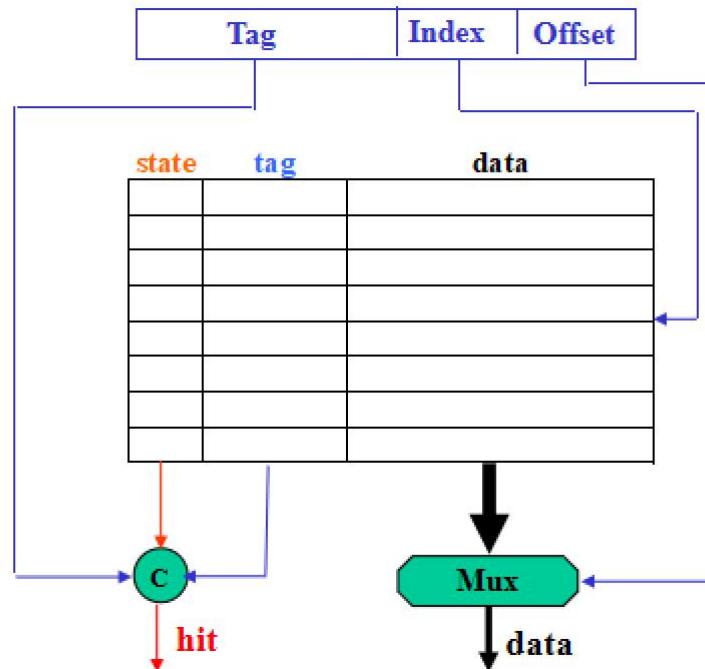
9.4.4 高速缓存

主存的子集，没有程序上的意义（采用相同的地址访问Cache和主存）

结构特点：同时存储数据和地址，需要考虑数据不在Cache中的情况（Miss）

关于Cache块的位置：

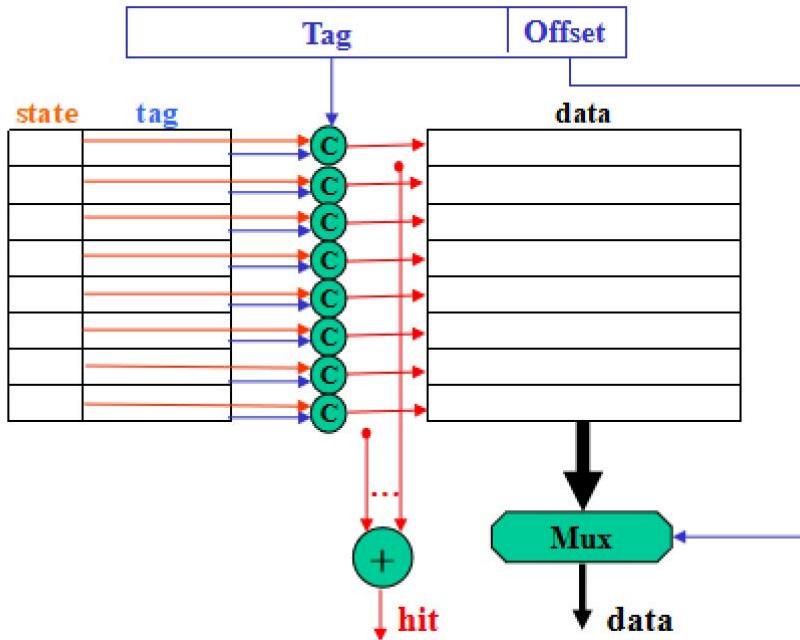
直接相连：内存中相同特征的块（比如模某个数后相等，一般是Cache支持的块数）映射到Cache中相同的一块



tag用来比对，index用来索引是Cache中的哪一块，offset用来指示数据在块中的位置

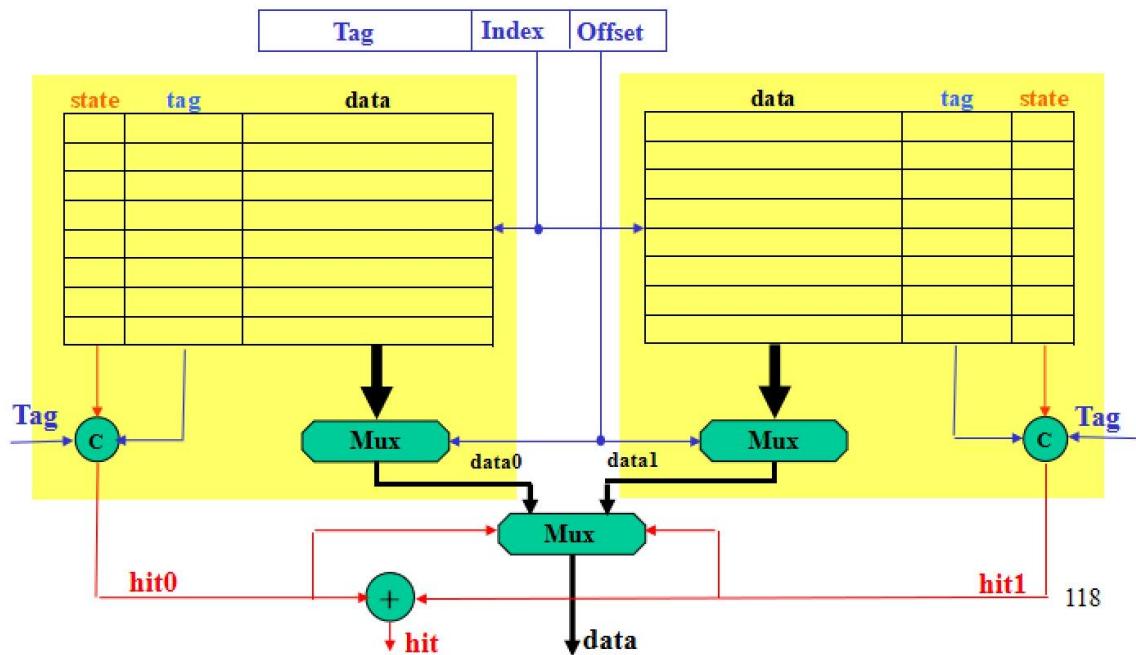
因此 $\text{len}(\text{index}) = \log_2(\text{Cache Item Num})$, $\text{len}(\text{offset}) = \log_2(\text{data block size})$

全相连：内存中的块可以映射到Cache中的任何一块



只有offset用来指示数据在块中的位置

组相连：首先将Cache分成若干组，组相连要求内存中特征相同的块可以映射到任意组中的同一位置。例如某内存块可以映射到第一组第一块，则它也可以映射到第二组的第一块。



检查每个组的第index项，查看是否有命中tag项的。

因此index用来指示组内的编号， offset用来指示块内的编号

Cache替换算法：

对于直接相连不存在替换算法的问题

常见的替换算法：随机替换、LRU (Least Recent Use) 、FIFO

Cache写命中的策略：

写穿 (Write Through) : 写Cache同时写到内存

写回 (Write Back) : 写Cache并且置dirty位，换出时若脏了再写回内存

一般来说L1到L2常用写穿， L2到内存用写回

Cache写失效策略： (写的时候Cache不命中)

写分配 (Write Allocate) : 先把失效块独到Cache，再在Cache中写

写不分配 (Write Non-allocate) : 写Cache失效时则直接写下一级存储

Cache性能分析: $AMAT = HitTime + MissRate \times MissPenalty$

优化：

- 降低失效率
- 降低失效延迟
- 降低命中延迟
- 提高Cache访问并行性

第10章 并行编程基础

10.1 程序的并行行为

指令级并行：不存在相关的指令可以并行执行（动态调度、多发射）

数据级并行：对集合或者数组中的元素同时执行相同的操作（如向量功能）

任务级并行：将不同的任务（进程或线程）分布到不同的处理单元上执行

10.2 并行编程模型

单任务数据并行 (SIMD)

多任务并行：共享存储

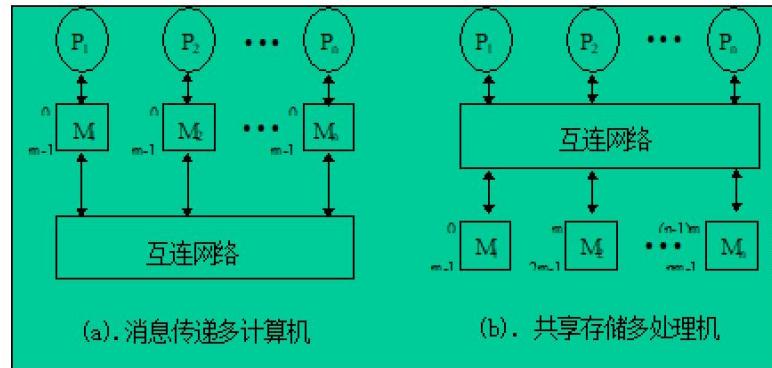
通过读写共享变量互相通信

如pthreads、OpenMP

多任务并行：消息传递

通过网络传递消息实现通信

如MPI、PVM



- 多地址空间
 - 消息传递通讯
 - 编程困难、程序移植困难
 - 通用性差
 - 可伸缩性好
- 单地址空间
 - 共享存储通讯
 - 编程容易、程序易移植
 - 通用性强
 - 可伸缩性一般

10.3 典型并行编程环境

10.3.1 SIMD

一条SIMD指令可以对一组数据进行相同的计算

10.3.2 Posix Thread (pthread)

POSIX: 共享存储编程标准

主要包含线程管理、线程调度、同步等原语

线程级并行

10.3.3 OpenMP

制导指令、运行库和环境变量

GCC支持 (#pragma制导!)

线程级并行

e.g. 矩阵乘法

```
#include <stdio.h>
#include <omp.h>
#define n 1000
double A[n][n], B[n][n], C[n][n];
int main()
{
    int i,j,k;
    //初始化矩阵A和矩阵B
    for(i=0;i<n;i++)
        for(j=0;j<n;j++) {
            A[i][j] = 1.0;
            B[i][j] = 2.0;
        }
    #pragma omp parallel for
    for(i=0;i<n;i++)
        for(j=0;j<n;j++) {
            C[i][j] = 0.0;
            for(k=0;k<n;k++)
                C[i][j] += A[i][k]*B[k][j];
        }
}
```

```

B[i][j] = 1.0; }

//并行计算矩阵C
#pragma omp parallel for shared(A,B,C) private(i,j,k)
for(i=0;i<n;i++)
    for(j=0;j<n;j++){
        C[i][j] = 0;
        for(k=0;k<n;k++)
            C[i][j]+=A[i][k]*B[k][j];}
    Return 0;
}

```

10.3.4 MPI标准

定义了一组消息传递函数库的编程接口标准

进程级并行

e.g. 积分求pi

```

#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
    int num_steps=1000000;
    double x,pi,step,sum,sumallprocs;
    int i,start, end,temp;
    int num_procs; //组中的进程数量,
    int ID; //进程编号,范围为0到num_procs-1
    MPI_Status status;

//Initialize the MPI environment
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&ID);
    MPI_Comm_size(MPI_COMM_WORLD,num_procs);

//任务划分并计算
    step = 1.0/num_steps;
    start = ID *(num_steps/num_procs) ;
    if (ID == num_procs-1)
        end = num_steps;
    else
        end = start + num_steps/num_procs;
    for(i=start; i<end;i++) {
        x=(i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Reduce(&sum,&sumallprocs,1,MPI_DOUBLE,MPI_SUM,0, MPI_COMM_WORLD);
    if(ID==0) {
        pi = sumallprocs*step;
        printf("pi %1f\n", pi);
    }
}

```

```

    MPI_Finalize();
    return 0;
}

```

矩阵乘法：

```

#include <stdio.h>
#include "mpi.h"
#define n 1000
int main(int argc, char **argv) {
    double *A,*B,*C;
    int i,j,k, ID,num_procs,line;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,
                  &ID); //获取当前进程号
    MPI_Comm_size(MPI_COMM_WORLD,
                  &num_procs); //获取进程数目
    //分配数据空间
    A = (double *)malloc(sizeof(double)*n*n);
    B = (double *)malloc(sizeof(double)*n*n);
    C = (double *)malloc(sizeof(double)*n*n);
    line = n/num_procs;//按进程数来划分数据
    if(ID==0){ //节点0, 主进程
        for(i=0;i<n;i++) //初始化数组
            for(j=0;j<n;j++){
                A[i*n+j] = 1.0;B[i*n+j] = 1;
        }
        //将矩阵A、B的相应数据发送给从进程
        for(i=1;i<num_procs;i++) {
            MPI_Send(B,n*n, MPI_DOUBLE,
                     i,0,MPI_COMM_WORLD);
            MPI_Send(A+(i-1)*line*n,line*n,
                     MPI_DOUBLE,i,1,MPI_COMM_WORLD);
        }
    }

    //接收从进程计算结果
    for(i=1;i<num_procs;i++)
        MPI_Recv(C+(i-1)*line*n,line*n,
                  MPI_DOUBLE,i,2,MPI_COMM_WORLD,&status);
    //计算剩下的数据
    for(i=(num_procs-1)*line;i<n;i++)
        for(j=0;j<n;j++) {
            C[i*n+j]=0;
            for(k=0;k<n;k++)
                C[i*n+j]+=A[i*n+k]*B[k*n+j];
        }
    } else { //其他进程接收数据, 计算结果, 发送给主进程
        MPI_Recv(B,n*n,MPI_DOUBLE,0,0,
                 MPI_COMM_WORLD,&status);
        MPI_Recv(A+(ID-1)*line*n,line*n,
                  MPI_DOUBLE,0,1,MPI_COMM_WORLD,&status);
        for(i=(ID-1)*line;i<ID*line;i++)
            for(j=0;j<n;j++) {
                C[i*n+j]=0;
                for(k=0;k<n;k++)
                    C[i*n+j]+=A[i*n+k]*B[k*n+j];
            }
    }
}

```

```

    }
    MPI_Send(C+(num_procs-1)*line*n, line*n,
             MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
}
MPI_Finalize();
Return 0;
} //main

```

第11章 多核处理结构

11.1 多核处理器发展的演化

动力：工艺、功耗墙、并行结构发展

分类：通用/专用，同构/异构，多核/众核 (>64)

11.2 多核处理器的访存结构 (to be continued, ESI协议)

片上Cache结构：

私有：处理器核包含私有Cache

片内共享：多个处理器核共享Cache (LLC)

片间共享：多片共享Cache

典型结构：片内共享LLC，片间共享内存

共享LLC结构： (LLC: Last Level Cache)

UCA (Uniform Cache Access) 结构：集中式共享，核通过crossbar连接最后一级Cache

NUCA (Non-uniform Cache Access) 结构：分布式共享，核拥有本地L2，可以通过片上互联访问其他核的L2

存储一致性模型：结构设计者和程序员之间的一种约定。对多处理器的访存次序做限制

- 顺序一致性：在多处理机环境的一个并行结果等于同一程序在单处理机下的结果
- 处理器一致性：允许Store之后的Load越过Store执行
- 弱存储一致性：把同步和访存分开，只在同步点维护一致性。冲突访问必须用同步操作保护
- 释放一致性：把同步进一步分成Acquire和Release，冲突访问必须用 (Rel, Acq) 对隔开

Cache一致性问题：保持数据在Cache以及主存中多个备份的一致性

——Cache一致性协议：把新写的值传播到其他处理器核的机制

如何传播？**Write-Invalidate & Write-Update**

- Write-Invalidate：当一个处理机更新某些单位时，通过某种机制使其他备份无效
- Write-Update：当一个处理机更新某些单位时，把内容传播给其他拥有备份的处理机

向何处传播？目录 & 倾听

- 倾听协议：其他处理机听广播，听到跟自己相关则进行相应的修改。适用于总线结构的SMP系统
- 目录协议：每个储存行有一个目录项，记录拥有该行副本的处理机；当发生修改时，根据目录项传播数据

ESI协议

实现一致性的方式。Cache行的三种一致性状态：

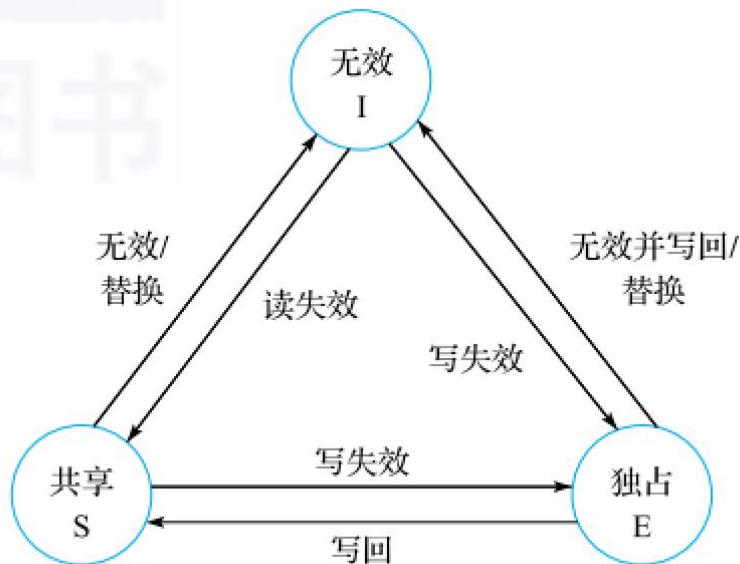


图 11.4 三状态 Cache 一致性协议状态转换图

MESI在ESI的基础上增加Modify态，用来表示当前Cache行被某处理器核独占且被修改。

- E：独占，表示对应的Cache行被当前核独占，当前核可以随意读写，其他核要用则必须请求该核放弃这行Cache
- S：共享，多个共享，但是只能读取，不能写入
- I：当前行无效

11.3 多核处理器的互连结构

片上总线

交叉开关

片上网络

拓扑结构：Ring、Mesh

路由算法（数据包怎么走？）：MESH中的寻路（维序路由：走到头；自适应路由：选择负载轻的方向）

路由器：将数据包从输入端口转发到输出端口（选择路径）

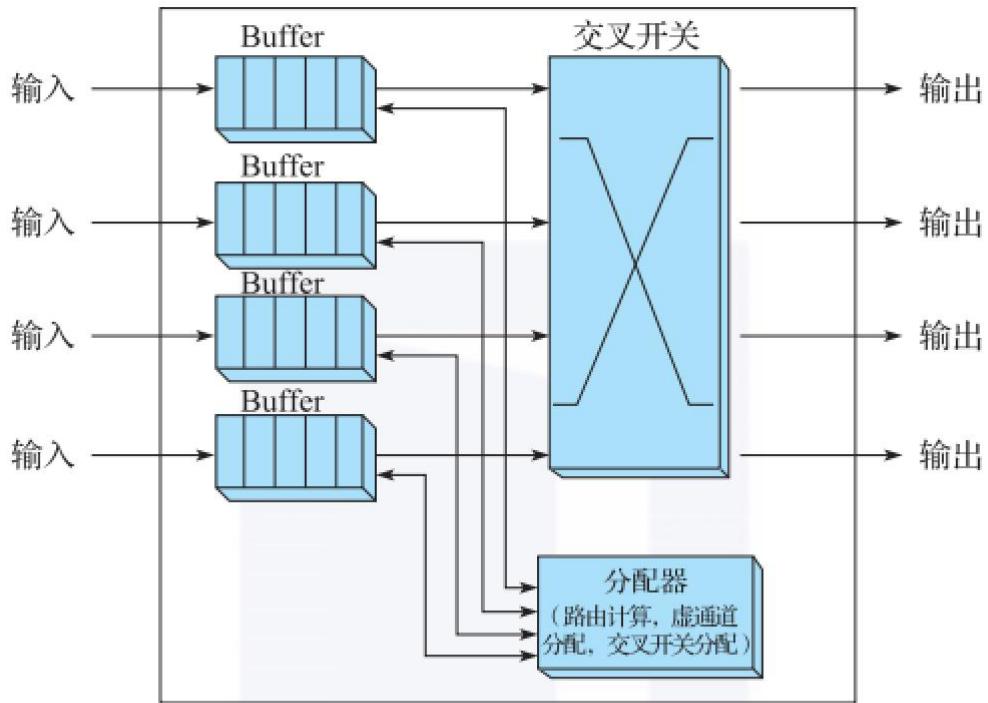


图 11.10 路由器结构图

流量控制：信道（channel）和缓冲区（buffers）。基于信号量的流控方法：每个节点对应的相邻节点都有一个计数器负责记录其buffer的使用情况，用发送Credit方式完成同步

11.4 多处理器的同步机制

三种常见的同步机制：**锁、barrier、事务内存**

常见的实现方式：**读改写指令、LL/SC指令、用户级软件例程（routine）**

读改写：

`test_and_set` (取完写)、`swap`、`compare_and_swap` (相等则交换)、`fetch_and_op` (取完操作再写回)

LL/SC：

LL: Load Linked, 取数目置LL bit为1

SC: Store Conditional, LL bit为1才能成功写回, 否则把源寄存器清零

事务内存：

事务：访问共享变量的代码区域声明为一个事务。事务中的指令要么执行要么不执行（原子性），任何时刻内存处于一致的状态（一致性），事务不能看见其他未提交的事务涉及的对象的状态（隔离性）

实现方式：软件（库或语言）、硬件

11.5 典型多核结构

第12章 计算机性能分析

12.1 计算机性能分析和评价

性能本质定义：完成一个任务的时间

影响性能的因素：应用、算法、编译系统、指令系统、微结构、主频

执行时间/响应时间

归一化执行时间（与一台标准机器相比）

CPI：每条指令的时钟周期数

MIPS：每秒钟执行百万条指令

MFLOPS：每秒执行百万条浮点指令

TPS：每秒中执行的事务

FPS：每秒的帧率

MBPS：衡量带宽，每秒多少兆字节

MHz：衡量主频，兆赫兹

并行系统的评价指标：

可扩展性（规模扩大，性能随之增长）

加速比（串转并的效益）

Amdahl定律

12.2 性能测试程序集

黑盒

Coremark, LMBench, Stream; SPEC CPU, EEMBC; Splash2....

12.3 性能分析方法

白盒

理论建模、模拟器、性能测量（性能计数器）

常见模拟器：SimOS、SimpleScalar...

性能分析工具：perf、oprofile