

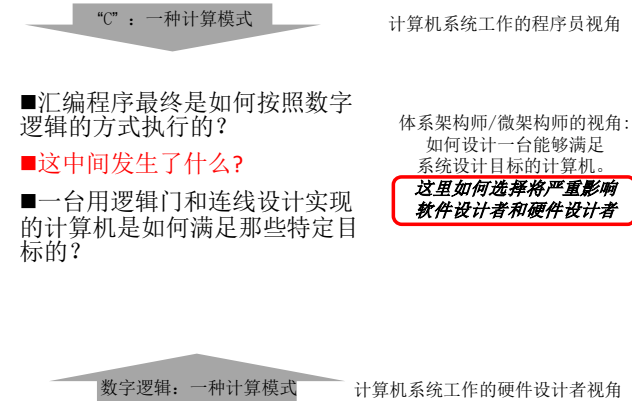
高等计算机体系结构

第一讲: 课程概论&计算机体系结构的基本概念

栾钟治
北京航空航天大学 计算机学院 中德联合软件研究所

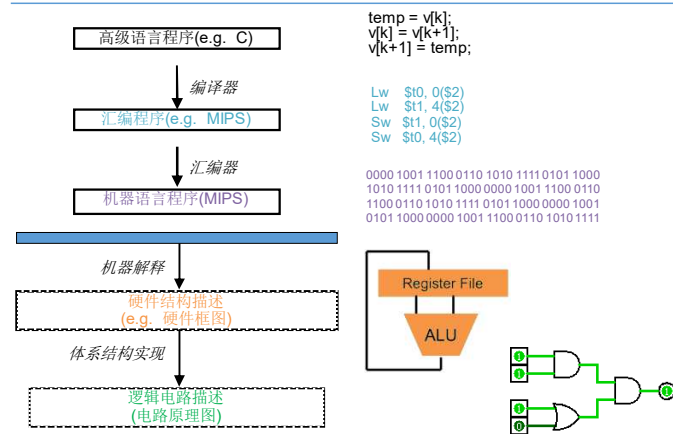
1

你们为什么而来



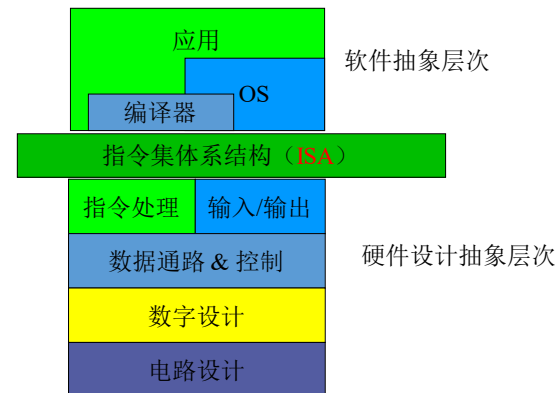
2

计算机系统的层次结构



3

计算机系统的层次结构



4

系统层次结构

"The purpose of computing is insight" (Richard Hamming)
我们在解决问题过程中获得洞察
如何确保可以用电子解决问题?



5

5

抽象的力量

• 分层描述系统产生了抽象

- 抽象: 上一层只需要知道跟下一层的接口, 而不需要知道下一层是如何实现的
- 例如, 高级语言的程序员并不需要“真的”了解什么是ISA以及计算机是如何执行指令的

• 抽象提高了生产效率

- 不需要考虑底层所做出的决策
- 例如, Java编程 vs. C编程 vs. 汇编语言编程 vs. 二进制代码 vs. 每一个时钟周期作用在每个晶体管上的电信号

- 那么, 为什么你想要知道到底下一层或者上一层发生了什么?

6

6

跨越抽象层次

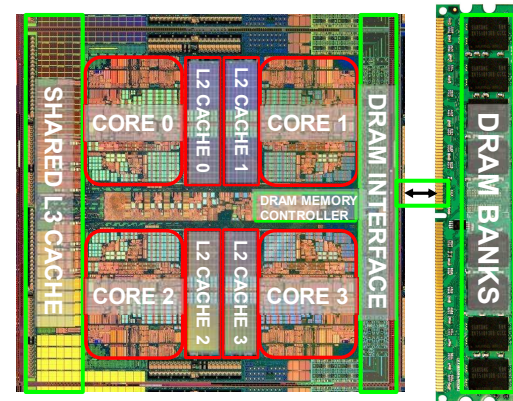
- 如果一切顺利, 不知道别的层次发生了什么也无关系要
- 但是, 如果
 - 你写的程序跑得很慢呢?
 - 你写的程序执行得不正确呢?
 - 你写的程序能耗太高呢?
- 但是, 如果
 - 在你设计的硬件上编程太困难呢?
 - 你设计的硬件效率太低因为你为软件提供的原语不够合理呢?
- 这门课的一个目标就是理解一个处理器是如何工作在软件层面之下以及硬件的决策是如何影响软件

7

7

一个例子

多核芯片

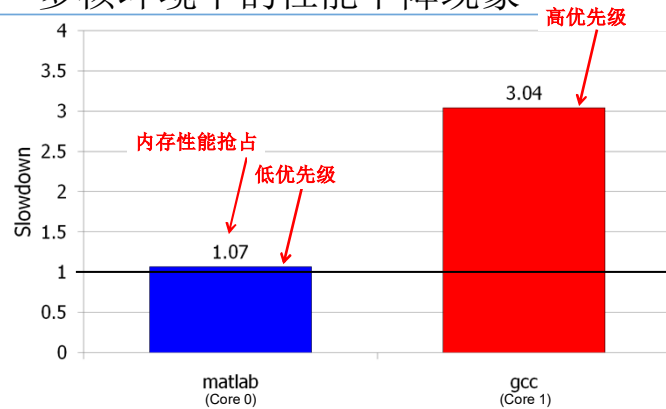


* AMD Barcelona

8

8

多核环境下的性能下降现象



Moscibroda and Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," USENIX Security 2007.

9

9

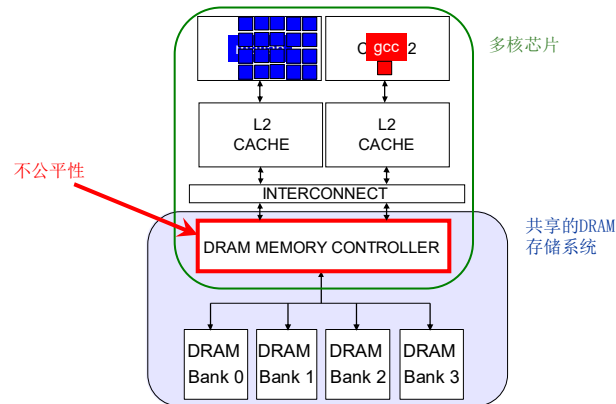
问题

- 如果你不知道处理器是如何执行程序的，你是否有可能找出性能下降差异的原因？
- 你能在不知道底层发生了什么的情况下解决问题吗？

10

10

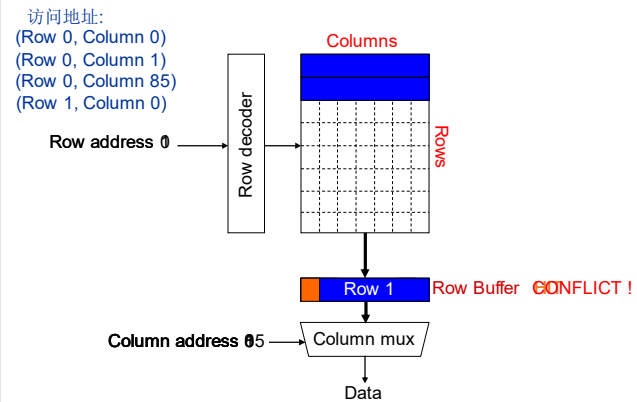
为什么性能下降会有这么大的差异？



11

11

DRAM Bank的操作



12

12

DRAM 控制器

- 行不命中时的内存访问时间远远大于行命中时的访问时间
- 现在的控制器得益于行缓冲 (row buffer)
- 通常采用的调度策略 (FR-FCFS) [Rixner 2000]*
 - (1) Row-hit first: 优先行缓冲命中
 - (2) Oldest-first: 然后先来先服务
- 这一调度策略的目的在于最大化DRAM的吞吐率

*Rixner et al., "Memory Access Scheduling," ISCA2000.

13

13

问题

- 多线程共享DRAM控制器
- DRAM 控制器设计用来最大化DRAM吞吐率
- DRAM调度策略是“线程不公平”的
 - Row-hit first: 导致对于高“行缓冲局部性”线程的“不公平优先”
 - Oldest-first: 导致对于内存密集型线程的“不公平优先”
- DRAM 控制器容易受到拒绝服务攻击
 - 可以写程序来利用这种不公平性

14

14

例子

```
// initialize large arrays A, B
for (j=0; j<N; j++) {
    index = j*linesize; streaming
    A[index] = B[index];
    ...
}
```

流

- 顺序的访存
- 极高的行缓冲局部性 (96% 命中率)
- 内存密集型

```
// initialize large arrays A, B
for (j=0; j<N; j++) {
    index = rand(); random
    A[index] = B[index];
    ...
}
```

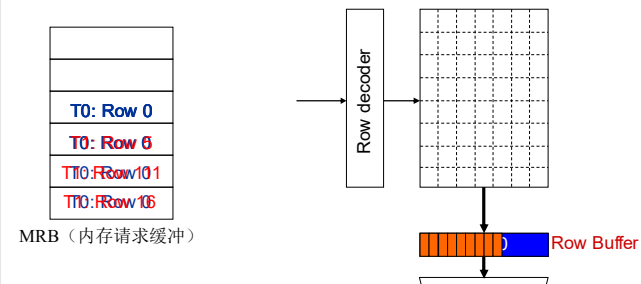
随机

- 随机访存
- 极低的行缓冲局部性 (3% 命中率)
- 同样是内存密集型

15

15

例子



16

16

现在我们知道底层会发生什么

■ 怎么解决这样的问题呢?

■ 在什么位置解决呢?

- ☐ 程序员?
- ☐ 系统软件?
- ☐ 编译器?
- ☐ 硬件 (内存控制器)?
- ☐ 硬件 (DRAM)?
- ☐ 电路?

■ 本课程的另外两个目标

- ☐ 批判性的思考
- ☐ 全局性的思考



17

17

简要回顾: 课程目标

- 理解处理器如何工作
- 实现一个简单的处理器
- 理解硬件层面的决策如何影响软件/程序员以及硬件设计本身
- 解决问题时的批判性思维
- 在不同层次之间的全局性思维
- 理解在设计时如何分析并且做出折衷 (tradeoff)

18

18

你们能学到什么

- 计算机体系结构: 通过硬件组件的设计、选择、互连以及软硬件接口的设计来创造计算系统的科学与艺术, 它使得创造出的计算系统能够满足功能、性能、能耗、成本以及其他特定的目标。

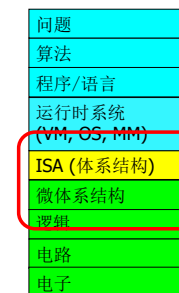
- 传统的定义: “体系结构这个术语描述了所观察到的系统属性, 也就是控制流的组织、逻辑设计性的结构和功能性的行为。”
Journal of R&D, 1964年4月



19

19

计算机体系结构在系统层次中的位置



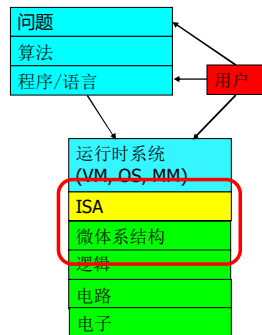
- 阅读: Patt, "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution," Proceedings of the IEEE 2001.

20

20

重新审视系统的层次结构

- 以用户为中心的视角: 为用户设计的计算机



- 整个层次结构都需要为用户需求来优化

21

21

你们能学到什么

- 设计软硬件接口和现代微处理器主要组件时的基本原则和方法
 - 关注一些新的研究和进展
 - 如何做出折衷
- 如何设计、实现和评价一个功能性的现代处理器
 - 通过实验
 - RTL级别的实现和更高层模拟相结合
 - 关注功能性(个别关注“如何可以更好”)
- 如何查找信息, 如何思考
- 如何更加努力的工作!

22

22

关于软件和硬件

- 本课程的分类大约是在“计算机硬件”之下
- 但是, 如果你能够同时掌握软件和硬件 (包括两者的接口)
 - 理解底层硬件能帮助你开发更好的软件
 - 理解软件的执行将使你更好的设计硬件
 - 两者都理解使你能够设计更好的计算系统
- 本课程涵盖软硬件接口以及微体系结构
 - 我们将关注设计时的折衷以及它是如何影响软件的

23

23

阅读材料布置

- Patt, [“Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution,”](#) Proceedings of the IEEE 2001.
- Patt & Patel's *Introduction to Computing Systems: From Bits and Gates to C and Beyond* (计算机系统概论) 第一章(基础)
- Patterson & Hennessy's *Computer Organization and Design: The Hardware/Software Interface* (计算机组成与设计: 软硬件接口) 第一章和第二章(概述, 抽象, ISA, MIPS)
- 其他相关材料
 - MIPS 手册

24

24

关于教材

- 没有要求
- 但是希望大家能够找一些书来阅读
 - David Patterson & John Hennessy, 《计算机组成与设计：软硬件接口》第4版
 - Yale Patt & Sanjay Patel, 《计算机系统概论》第2版
 - David Harris & Sarah Harris, 《数字设计和计算机体系结构》第2版
 -

25

25

为什么要学习计算机体系结构?

26

26

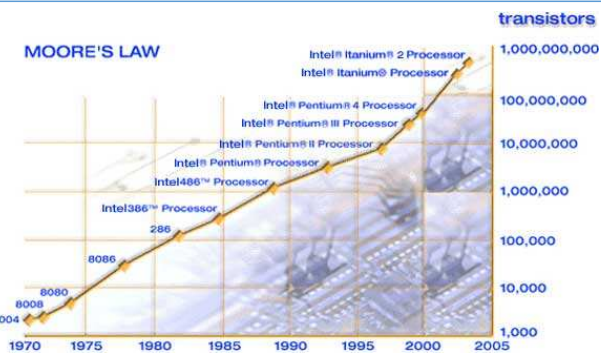
什么是计算机体系结构?

- 通过硬件组件的设计、选择、互连以及软硬件接口的设计来创造计算系统的科学与艺术，它使得创造出的计算系统能够满足功能、性能、能耗、成本以及其他特定的目标。
- 你们很快将会看到体系结构和微体系结构之间的区别

27

27

使能: Moore's Law

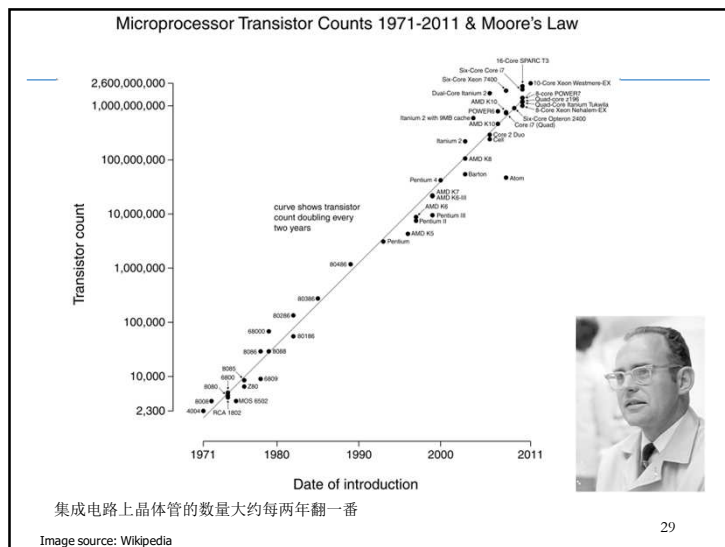


Moore, "Cramming more components onto integrated circuits,"
Electronics Magazine, 1965. 组件数量每隔一年翻一番

Image source: Intel

28

28



29

我们用这些晶体管来做什么

- 请大家在课下的阅读中思考...
- Patt, “Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution,” Proceedings of the IEEE 2001.

30

30

为什么要学习计算机体系结构?

- **更好的系统:** 使计算机更快、更便宜、更小、更可靠...
 - 通过利用底层技术的进步
- **更新的应用**
 - 三维可视化
 - 虚拟现实
 - 个人基因组
- **更好的解决问题**
 - 软件的创新与计算机体系结构的发展和改进紧密融合
 - 每年超过一半的硬件性能提升支撑了软件的创新
- **理解计算机为什么会这样工作**

31

31

今天的计算机体系结构(I)

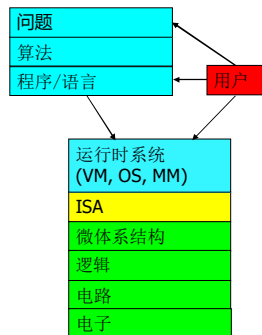
- 计算机工业进入一个大的转变时期: **多核-更多的核**
 - 很多潜在的不同的系统设计可能性
- **很多困难的问题** *驱动了* 这种转变同时也 *由于* 这些转变变得更困难
 - 功耗/能耗约束
 - 设计的复杂性 → 多核?
 - 技术扩展的困难 → 新的技术?
 - 存储墙(wall/gap)
 - 可靠性墙/问题(wall/issues)
 - 可编程性墙/问题(wall/problem)
- 这些问题都没有清晰、确定的答案

32

32

今天的计算机体系结构(II)

- 这些问题影响着计算系统的各个层次- 如果我们不改变我们设计系统的方式



33

33

今天的计算机体系结构(III)

- 如果你理解软件和硬件并且对它们作相应的改变, 你可能革新构建计算机的方法
- 你可以发明新的计算、通信和存储模式
- 推荐一本书: Kuhn, "The Structure of Scientific Revolutions" (1962)
 - 前范式阶段
 - 常规阶段
 - 反常阶段
 - 危机阶段
 - 革命阶段

34

34

... 但是, 首先 ...

- 让我们来理解基本的概念...
- 只有当你理解得足够好之后才有可能改变...
 - 尤其是那些过去和现在起主导地位的范式
 - 同时, 它们的优点和缺点 -- tradeoffs

35

35

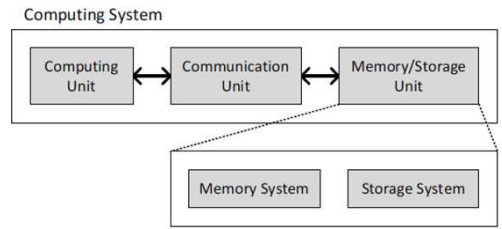
基本概念

36

36

什么是计算机?

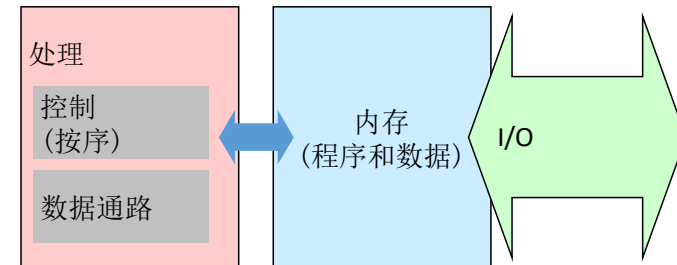
- 三个关键的要素
- 计算
- 通信
- 存储 (内存)



37

什么是计算机?

- 我们会讨论所有这三个要素



38

38

冯诺依曼结构/模型

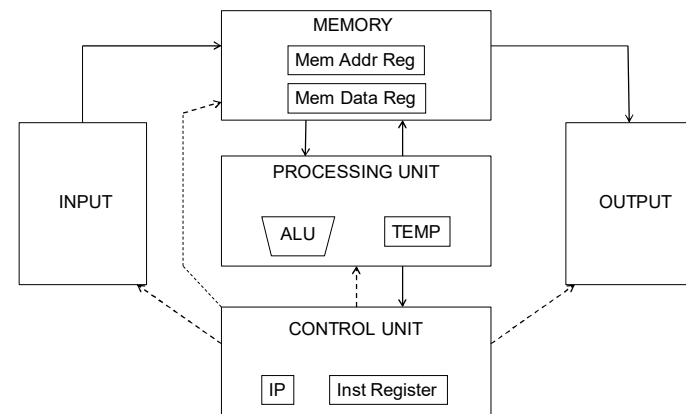
- 也叫 **存储程序计算机** (指令在内存中), 两个关键的属性:
- 存储程序
 - 指令存储在一个线性的存储阵列中
 - 内存统一的存储指令和数据
 - 依靠控制信号实现对存储的值的解释
- 顺序的指令处理
 - 一次处理一条指令 (取指、执行)
 - 程序计数器 (指令指针) 标识“当前”指令
 - 程序计数器按顺序推进, 除了控制转移指令
- 推荐阅读
 - Burks, Goldstein, von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," 1946.
 - Patt & Patel, 第四章, "The von Neumann Model"

什么时候一串数字会被解释成一条指令呢?

39

39

冯诺依曼结构/模型(计算机)



40

40

数据流模型(计算机)

- 冯诺依曼模型: 指令的获取和执行按照**控制流的顺序**
 - 由**指令指针**来指定
 - 顺序推进除非遇到明确的控制转移指令
- 数据流模型: 指令的获取和执行按照**数据流的顺序**
 - 当操作数准备好
 - 没有指令指针**
 - 指令的顺序依赖数据流来确定
 - 每条指令指定结果的接收者
 - 一条指令在获得所有操作数后就可以执行
 - 意味着多条指令可能同时执行
 - 本质上具备更高的并行性

41

41

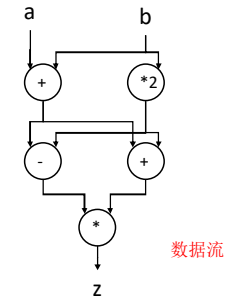
冯诺依曼vs 数据流

■考虑一个冯诺依曼结构下的程序

- 程序的顺序
- 存储的位置

```
v <= a + b;
w <= b * 2;
x <= v - w;
y <= v + w;
z <= x * y;
```

顺序的



数据流

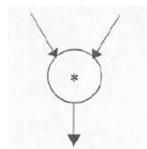
■如果你是程序员，你觉得哪一种模型更自然？

42

42

关于数据流

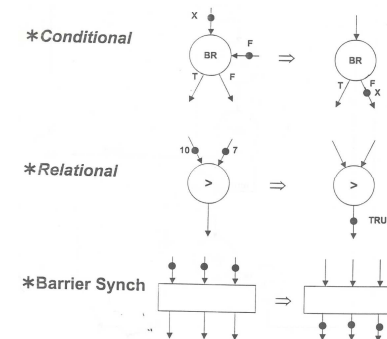
- 在数据流机中，程序由数据流节点构成
 - 数据流节点会在所有输入都准备好时发射(取指和执行)
 - 可以理解为当所有输入获得令牌
- 数据流节点和它的 ISA 表示



*	R	ARG1	R	ARG2	Dest. Of Result
---	---	------	---	------	-----------------

43

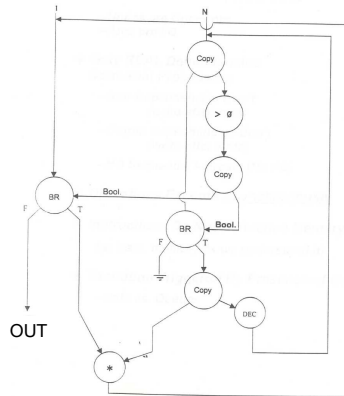
数据流节点



44

44

数据流程的例子



45

45

指令指针—ISA层面的折衷

■ 是否需要在ISA中设计指令指针?

□ 是: 控制驱动, 按顺序执行

■ 指令在IP指向它时被执行

■ IP 按顺序自动改变 (控制转移指令除外)

□ 否: 数据驱动, 并行执行

■ 当所有操作数就位执行指令 (数据流)

■ Tradeoff: 涉及到上层

□ 编程是否方便 (对大多数程序员)?

□ 编译是否方便?

□ 性能: 并行性如何?

□ 硬件复杂性如何?

46

46

ISA 和 微体系结构的折衷

• 在微体系结构层面需要作出类似的tradeoff

• ISA: 程序员视角看指令如何执行

- 程序员看到一个顺序的、控制流驱动的执行序

vs.

- 程序员看到一个数据流驱动的执行序

• 微体系结构: 底层实现如何执行指令

- 微体系结构可以按照任意的序来执行指令, 只要它能够按照ISA确定的语义将指令结果呈献给软件即可
 - 程序员应该看到的是ISA确定的序

47

47

让我们回到冯诺依曼结构

• 如果你想了解更多有关数据流...

- Dennis and Misunas, "A preliminary architecture for a basic data-flow processor," ISCA 1974.
- Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.

48

48

冯诺依曼结构/模型

- 今天所有主要的ISA都遵循冯诺依曼结构
 - x86, ARM, MIPS, SPARC, Alpha, POWER
- 在微体系结构层面,几乎所有的具体实现(微体系结构)都有很大的不同
 - 流水线执行: Intel 80486 微架构
 - 多指令并发: Intel Pentium 微架构
 - 乱序执行: Intel Pentium Pro 微架构
 - 指令和数据cache分离
- 但是,不管底层采用什么看上去与冯诺依曼模型不符的情况,都不会向上暴露给软件层面
 - ISA 和 微体系结构之间的区别

49

49

再来看什么是计算机体系结构?

- 现代的定义 (ISA+实现): 通过硬件组件的设计、选择、互连以及软硬件接口的设计来创造计算系统的科学与艺术, 它使得创造出的计算系统能够满足功能、性能、能耗、成本以及其他特定的目标。
- 传统的定义(只有ISA): “体系结构这个术语用来描述程序员所观察到的系统属性, 也就是那些不同于数据流和控制流的组织、逻辑设计以及物理实现的概念性的结构和功能性的行为。” Gene Amdahl, IBM Journal of R&D, April 1964

50

50

ISA vs. 微体系结构

- ISA
 - 约定软硬件之间的接口
 - 软件开发需要了解以便编写及调试系统或用户程序
- 微体系结构
 - 某种ISA的一个特定实现
 - 对软件不可见
- 微处理器
 - ISA, 微架构, 电路
 - “Architecture” = ISA + microarchitecture

问题
算法
程序
ISA
微体系结构
电路
电子

51

51

ISA vs. 微体系结构

- 哪些部分属于ISA 或者微体系结构?
 - 油门: “加速”的接口
 - 发动机内部: “加速”的具体实现
- 在满足ISA的规范前提下具体实现(微架构)可以是多种多样的
 - 加法指令vs. 加法器实现
 - 串行加法器、脉动进位加法器、超前进位加法器等都是微体系结构的一部分
 - x86 ISA 有很多种实现: 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core, ...
- 微体系结构通常比ISA演变的快
 - 只有有限的ISA (x86, ARM, SPARC, MIPS, Alpha) 但是有很多种微架构
 - Why?

52

52

ISA

- 指令
 - 操作码、寻址方式、数据类型
 - 指令类型和格式
 - 寄存器、状态码
- 存储（内存）
 - 地址空间、寻址能力、对齐
 - 虚存管理
- 调用, 中断/异常处理
- 访问控制, 优先级/特权
- I/O: 内存映射vs. 指令
- 任务/线程管理
- 功耗和温度管理
- 多线程支持, 多处理器支持

53

53

微体系结构

- ISA 在具体设计约束和目标之下的具体实现
- 任何在硬件上完成而没有暴露给软件的部分
 - 流水线
 - 指令按序或者乱序执行
 - 访存调度策略
 - 投机执行
 - 超标量处理(多指令发射)
 - 时钟门控
 - 高速缓存: 级数, 大小, 关联方式, 替换策略
 - 预取
 - 电压/频率调节
 - 差错修正

54

54

与两者都相关的属性

- 加法指令的操作码
- 通用寄存器的个数
- 寄存器堆的端口数
- 执行乘法指令需要几个周期
- 机器是否采用流水线指令执行
-
- 牢记
 - 微体系结构: ISA 在具体设计约束和目标之下的具体实现

55

55

设计要点（Design Point）

- 一组设计时需要考虑的重要问题
 - 将导致包括ISA和微架构方面的tradeoff
- 关注
 - 成本
 - 性能
 - 最大功耗限制
 - 能耗(电池寿命)
 - 可用性
 - 可靠性和正确性
 - 上市时间

问题
算法
程序
ISA
微体系结构
电路
电子

- 设计要点由“问题”空间(应用)或者面向的用户/市场决定

56

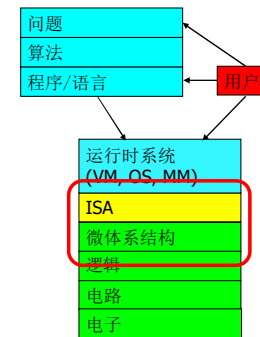
56

Tradeoff: 计算机体系结构的灵魂

- ISA层面的折衷
- 微体系结构层面的折衷
- 系统和任务层面的折衷
 - 如何分配软件和硬件应该承担的工作?
- 计算机体系结构是为满足设计点要求做出合适折衷的科学和艺术

57

57



58

58