Instruction Set Principles

Basic Concepts of Computing

计算机的基本组成

- 计算：包括控制与 datapath
- 存储：数据以及指令
- 通信：组件间以及与外界的通信

Von-Neumann

冯诺依曼架构由五个部分组成：控制，运算，存储，输入，输出

冯诺依曼架构有两个主要属性

- stored program：指令存储在内存中，机器不区分指令和数据
- sequential instruction processing：每次处理一条指令，pc 除了控制转移以外顺序向前

事实上除了冯诺依曼架构还有数据流模型

Data flow

数据流模型中指令按照**数据流**的顺序处理执行，数据流顺序是根据指令的依赖决定的（指令需要的结果从哪条指令而来）

只要能够执行，可以同时执行多条指令，对比冯诺依曼架构（按照**控制流**顺序执行）

数据流可以用 petri net 建模，一条指令能够执行（fire）当且仅当其依赖的输入全都就绪

Tradeoff：是否真的需要一个 instruction pointer?

为什么需要数据流模型

- 微架构可以使用数据流的执行顺序，同时向程序员展示一个顺序执行的结果
- 深度学习的神经网络需要数据流模型

事实上所有主流 ISA 均采用冯诺依曼架构，而底层的微架构根据实现各有不同

ISA vs. Microarchitecture

ISA 是软硬件的接口，对软件可见

微架构是一个 ISA 的具体实现，对软件不可见

"Architecture" = ISA + Microarchitecture

ISA

ISA 包括多种内容

- 指令
  - 操作码，寻址模式，数据类型
  - 指令类型和格式
  - 寄存器，状态码
- 内存
  - 地址空间，寻址，对齐
  - 虚拟内存管理
- 调用，中断和异常处理
- 控制访问，优先级
- IO：内存映射或是 IO 指令
- 任务调度
- 能耗管理
- 多线程/多处理器支持

Microarchitecture

实现 ISA

- 流水线
- 按序/乱序执行
- 内存访问策略
- 预测执行
- 超标量
- 时钟周期
- cache
- 预取
- 电压/频率的调整
- 纠错

ISA Tradeoffs

tradeoff 是体系结构的灵魂

- ISA 级别
- 微架构级别
- 系统与任务级别

Instruction Format

是软硬件接口的基本元素，由两个基本要素组成

- opcode：做什么
- operands：谁来做

可以是定长的（MIPS）或是变长的（x86）

定长的编码一般可以使用一致的解码，如 opcode 总是在指令的同一位置

变长的编码（如x86）通常有多种前缀后缀

tradeoff：

- 定长易于解码，且可以同时解码多条指令
- 定长难以扩展 ISA
- 变长可以压缩代码（Huffman）
- 变长需要更复杂的解码逻辑，且难以同时解码多条指令

根据操作数的不同可以分为多种

- 0-address: stack machine (op, push A, pop A)
- 1-address: accumulator machine (op ACC, ld A, st A)
- 2-address: 2-operand machine (op S, D; one is both source and dest)
- 3-address: 3-operand machine (op, S1, S2, D)

tradeoff：更长的指令还是更多的指令数

以 stack machine 为例，优缺点分别为

- 优点：指令尺寸小（不需要 operand），过程调用容易（所有参数都在栈中）
- 缺点：不能使用前缀方式解决的计算难以实现

Data Type

数据及在其上定义的操作

ISA 可以支持多种数据类型，如整数，浮点数，字符，二进制串，BCD 码等，甚至如双链表，队列，位向量，栈等复杂结构

tradeoff：更简单的数据类型或是更高层的数据类型（ISA 与编程语言之间的 semantic gap）

更复杂的数据结构可以简化编译，但是会增加硬件设计的难度

Complex vs. Simple instruction

复杂的指令或简单的指令

如 x86 中的 REP 前缀，可以让一个指令重复执行

复杂指令的 tradeoff

- 代码体积小，提高内存利用，编译器更简单
- 编译器更难进行细粒度优化，硬件复杂度提升

随之产生了 RISC 和 CISC（以及一些 semantic gap 更大或更小的架构）

也可以选择和编程语言 semantic gap 更小的 ISA，然后进一步将 ISA 的指令翻译为更接近微架构的微码

Translation：可以将一个 ISA 翻译到另一个 ISA：用户呈现的 ISA 和实现的 ISA，x86 就采用了这种方式

Visible/Invisible States

程序员可见的状态：内存，寄存器，PC

程序员不可见的状态：cache，流水线寄存器

内存的组织包括了

- 内存空间：有多少唯一的可寻址的位置
- 存放方式：绝大多数按字节存放（由此产生了大端序和小端序的问题）
- 虚拟内存

寄存器包括

- 寄存器数量
- 寄存器大小
- 种类（通用 or 专用）

寄存器是利用了程序的局部性（空间 and 时间）

寄存器的进化经过了：累加器——累加器+地址寄存器——通用寄存器（GPRs）的过程

寄存器数量的 tradeoff

- 数量多代表编码的比特数多
- 数量多可以存储更多临时量，利于寄存器分配
- 数量影响访问时间，能耗

Load/Store vs. register/memory

load/store 架构是指指令的 operand 只能是寄存器

register/memory 架构是指指令的 operand 可以是寄存器或内存地址

> 寻址方式可以有很多种
>
> - 绝对寻址：直接以立即数为地址
> - 寄存器：以寄存器内容为地址
> - 寄存器 + 偏移
> - 寄存器 + 下标
> - 间接内存：以寄存器内容的地址所存储的内容为地址
> - 寄存器寻址，但是每次寻址后自增/自减

寻址模式的 tradeoff

- 更多寻址模式可以更好地将高层指令映射到机器
- 有些地址用不同模式表示可以显著减小编码体积
- 编译器工作更复杂

> ISA 正交性：所有寻址模式可以用于所有指令

IO device

与 IO 设备的交互也有多种方式

- 内存映射：将一个内存区域映射到 IO 设备，使用内存读写命令
- 直接读写：使用专门的 IO 读写指令

Other ISA elements

如访问控制，异常/中断处理，虚拟内存，条件赋值，对齐，精确中断，cache 一致性……