高等计算机体系结构
第二讲: ISA设计和折衷的基本概念、原则和实现基础

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

本讲相关阅读材料

- Patterson & Hennessy's *Computer Organization and Design: The Hardware/Software Interface*（计算机组成与设计：软硬件接口）第四章 (重点阅读 4.1-4.4)
- Patt & Patel's *Introduction to Computing Systems: From Bits and Gates to C and Beyond*（计算机系统概论）, 第四章, "The von Neumann Model"
- 其他论文
  - 课程网站

2

2

上一讲回顾

- 冯诺依曼结构的两个关键的特征?
- 数据流模型的ISA可能带给程序员什么样的困难?
- 如何即能保留数据流的优点又能解决这些困难?

3

3

回顾：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

4

4

回顾: ISA 和 微体系结构的折衷

- 在ISA层面需要做出是数据流还是控制流的抉择，在微体系结构层面也要做出类似的折衷
- 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

5

5

回顾：与ISA和微架构都相关的属性？

- 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
- ……

微体系结构: ISA 在具体设计约束和目标之下的具体实现

6

6

回顾：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

- 设计要点由“问题”空间 (应用)或者面向的用户/市场决定

7

7

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*

8

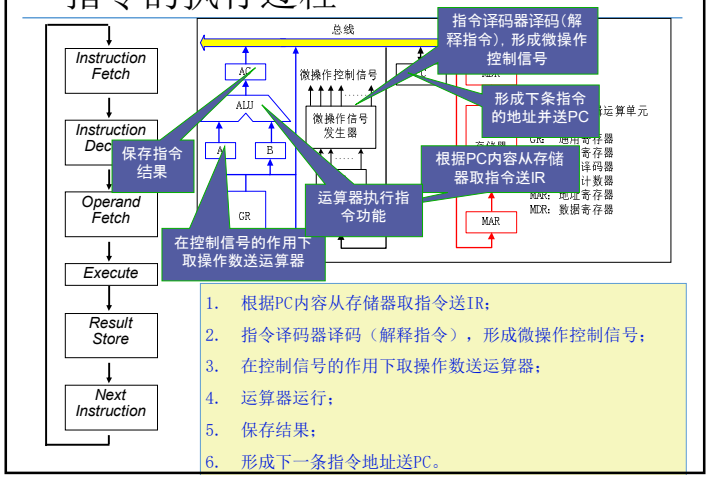
8

ISA 的设计原则和折衷

9

9

指令的执行过程



1. 根据PC内容从存储器取指令送IR；
2. 指令译码器译码（解释指令），形成微操作控制信号；
3. 在控制信号的作用下取操作数送运算器；
4. 运算器运行；
5. 保存结果；
6. 形成下一条指令地址送PC。

10

Example

$Y=ax^2+bx-c$ 假定a, b, c, x均为已知数，且已存放在内存，求y。

假定指令系统：16位指令系统

Opcode	Address
8	8

指令	操作码	说明
ADD	00H	AC ← (AC)+Mem(Add)
LD	01H	AC ← Mem(Add)
SUB	02H	AC ← (AC) — Mem(Add)
MUL	03H	AC ← (AC)×Mem(Add)
ST	04H	Mem(Add) ← (AC)

内存	地址
	00H
	02H
	04H
	06H
	08H
	0AH
	0CH
	0EH
结果y将存放在此	10H
值a	12H
值b	14H
值c	16H
值x	18H

11

Example

$Y=ax^2+bx-c$ 假定a, b, c, x均为已知数，且存放在内存中，求y。

指令	操作码	说明
ADD	00H	AC ← (AC)+Mem(Add)
LD	01H	AC ← Mem(Add)
SUB	02H	AC ← (AC) — Mem(Add)
MUL	03H	AC ← (AC)×Mem(Add)
ST	04H	Mem(Add) ← (AC)

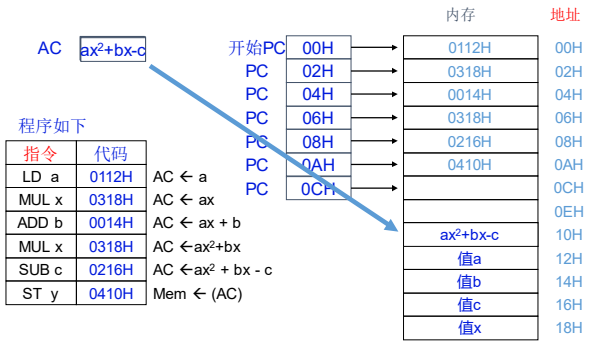
程序如下

指令	代码	
LD a	0112H	AC ← a
MUL x	0318H	AC ← ax
ADD b	0014H	AC ← ax + b
MUL x	0318H	AC ←ax^2+bx
SUB c	0216H	AC ←ax^2 + bx - c
ST y	0410H	Mem ← (AC)

内存	地址
	00H
	02H
	04H
	06H
	08H
	0AH
	0CH
	0EH
结果y将存放在此	10H
值a	12H
值b	14H
值c	16H
值x	18H

12

Example



程序如下

指令	代码	
LD a	0112H	AC ← a
MUL x	0318H	AC ← ax
ADD b	0014H	AC ← ax + b
MUL x	0318H	AC ←ax²+bx
SUB c	0216H	AC ←ax² + bx - c
ST y	0410H	Mem ← (AC)

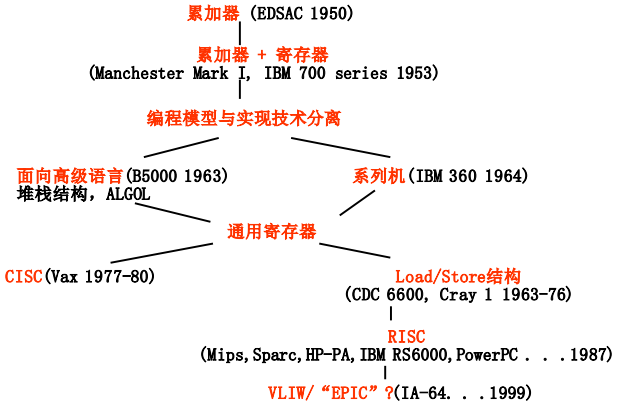
13

指令系统

- 从指令执行周期看指令涉及的内容

14

指令系统的演变



15

各种不同的指令集体系结构（ISA）

- x86
- PDP-x: Prog
- VAX
- IBM
- CD
- SIM
- VL
- Po
- RIS

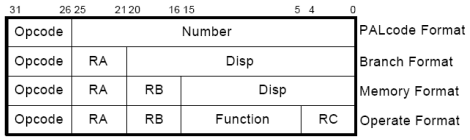
- 有什么本质的不同?
  - 例如，如何描述指令，指令的功能
  - 例如，指令有多复杂

16

16

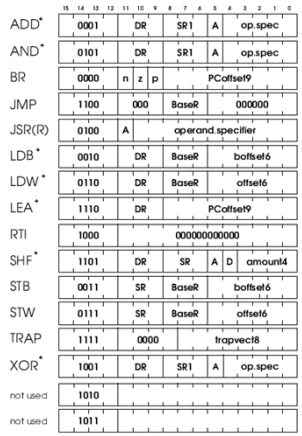
指令

- 软硬件接口的基本要素
- 指令的基本构成
  - 操作码（opcode）：做什么
  - 操作数（operand）：谁去做
-



17

指令集

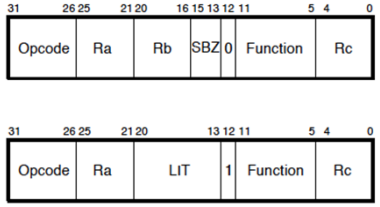


- LC-3b ISA
  - Patt & Patel
- “bit steering”
  - 指令中的某一位决定指令中其它位的含义
- 为什么要有“not used”的指令?

18

Alpha 指令集中的Bit Steering

Figure 3–4: Operate Instruction Format



If bit <12> of the instruction is 0, the Rb field specifies a source register operand.

If bit <12> of the instruction is 1, an 8-bit zero-extended literal constant is formed by bits <20:13> of the instruction. The literal is interpreted as a positive integer between 0 and 255 and is zero-extended to 64 bits. Symbolically, the integer Rbv operand is formed as follows:

19

ISA的要素（1）

- 指令的执行序
  - 控制流 vs. 数据流
  - Tradeoffs?
- 指令处理的风格
  - 操作数的个数及操作动作的定义
  - 0, 1, 2, 3 地址的机器
    - 0-地址：栈（push A, pop A, op）
    - 1-地址：累加器（ld A, st A, op A）
    - 2-地址：双操作数（其中一个操作数即是源又是目的）
    - 3-地址：3操作数（源和目的分离）
  - Tradeoffs?
    - 更大的操作指令数 vs. 更多的可执行操作
    - 代码大小vs. 执行时间 vs. 片上存储空间

20

例子: Stack Machine

+ 指令短小(指令不带操作数)
  - 简单的逻辑
  - 紧凑的代码

+ 过程调用很高效：所有的参数都在栈里
  - 不需要额外的时钟周期去做参数传递

-- 对计算模式有要求
  - 不能同时对多个值进行操作
  - 不灵活

21

21

例子: Stack Machine (II)

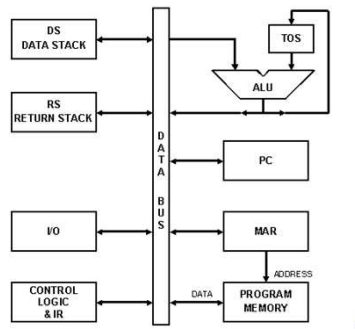


Figure 3.1 -- The canonical stack machine.

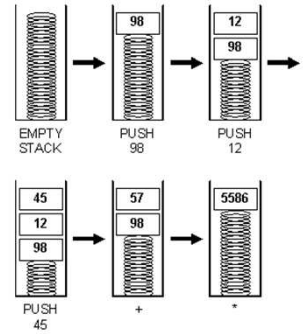


Figure 3.2 -- An example stack machine.

Koopman, "Stack Computers: The New Wave," 1989.

22

22

其它例子

- PDP-11: 双地址机器
  - PDP-11的 ADD: 4-bit 操作码, 2个 6-bit 操作数说明符
  - 用有限的位数指定一条指令
  - 缺点：其中一个源操作数总是会被指令执行的结果所覆盖
    - *如何确保能够保留源操作数的旧值?*

- X86: 3地址机器 (memory/memory)
- Alpha: 3地址机器 (load/store)
- MIPS?

23

23

ISA的要素（II）

- 指令
  - 操作码
  - 操作数说明符（包含寻址方式）
    - 如何获得操作数

为什么会有不同的寻址方式?

- 数据类型
  - 指令要操作的信息的表示方式
  - 整型，浮点数，字符，二进制，十进制，BCD
  - 双向链表，队列，串，位向量，栈

24

24

数据类型的Tradeoffs

- ISA中采用更多或者更高层的数据类型有什么好处？有什么缺点？
- 编译器/程序员 vs. 微体系结构
- 语义鸿沟（semantic gap）
  - 数据类型与语义或者指令的复杂性之间是紧密耦合的
- 例如：早期的 RISC vs. Intel 432
  - 早期的 RISC：只有整型
  - Intel 432：对象数据类型，能力很强大

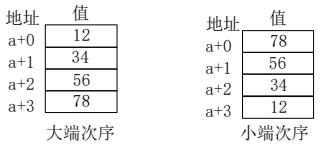
ISA的要素（III）

- 内存的组织
  - 地址空间：内存中有多少可以唯一标识的位置
  - 可寻址性：每个可唯一标识的位置能够存储多少数据
    - 字节寻址：大多数的ISA，8位的字符
    - 位寻址：Burroughs 1700
    - 64位寻址：一些超级计算机
    - 32位寻址：第一台 Alpha
    - 需要思考的是
      - 如何在字节寻址的结构下完成2个32位数的加法?
      - 如何在32位寻址的结构下完成2个8位数的加法?
      - 大端寻址和小端寻址? MSB 在高字节还是低字节
  - 对虚拟内存的支持

操作数的存储方式

- 大端（big-endian）次序：最高有效字节存储在地址最小位置
- 小端（little-endian）次序：最低有效字节存储在地址最小位置

例： Int a; //0x12345678



相关资料阅读

- 希望深度挖掘的同学可以阅读
  - Wilner, “Design of the Burroughs 1700,” AFIPS 1972
  - Levy, “The Intel iAPX 432,” 1981

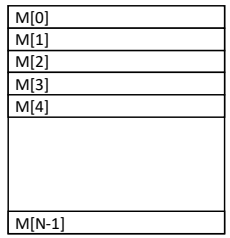
ISA的要素（IV）

- 寄存器
  - 多少个寄存器
  - 每个寄存器的大小
- 为什么用寄存器?
  - 因为程序展现了一种特性叫做数据局部性
  - 最近产生或者访问的值可能会被多次使用（时间局部性）
    - 存在寄存器中减少了访存次数

29

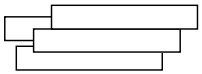
29

程序员可见(体系结构相关)的状态



内存

用地址索引的存储位置的阵列



寄存器

- 在ISA中会给寄存器命名（相当于地址）
- 通用和专用寄存器

程序计数器

当前指令的内存地址

指令和程序指定如何转换程序员可见的状态值

30

30

程序员不可见的状态

- 微体系结构的状态
  - 程序员不能直接访问
- 比如 cache 的状态
- 比如 流水线寄存器

31

31

寄存器结构的演变

- 累加器
  - 加法机时代遗留下来的
- 累加器+地址寄存器
  - 需要寄存器间接寻址
  - 最初的地址寄存器是专用的，只能用来加载一个地址用来间接寻址
  - 最终能够支持地址的运算
- 通用寄存器(GPR)
  - 所有的寄存器都能用来做任何事
  - 一开始只有几个，发展到 32 (RISC架构中常用)个 ，再到 128 个（ Intel IA-64）

32

32

指令的类别

- 操作类指令
  - 数据处理：算术和逻辑操作
  - 取操作数，计算结果，存储结果
  - 隐式的顺序控制流
- 数据移动类指令
  - 在内存、寄存器、I/O 设备之间移动数据
  - 隐式的顺序控制流
- 控制类指令
  - 改变指令执行的顺序

33

33

ISA的要素（V）

- Load/store vs. memory/memory 架构
  - Load/store 架构：操作类的指令只操作寄存器
    - 例如： MIPS, ARM 以及大多数 RISC ISA
  - Memory/memory 架构：操作类的指令可以操作内存
    - 例如： x86, VAX 以及大多数 CISC ISA

34

34

ISA的要素（VI）

- 形式地址与有效地址
  - 形式地址：指令中直接给出的地址编码。
  - 有效地址：根据形式地址和寻址方式计算出来的操作数在内存单元中的地址。
  - 寻址方式：根据形式地址计算到操作数的有效地址的方式（算法）
- 寻址方式 指明如何获得操作数
  - 绝对寻址方式 LW rt, 10000
    用立即数作为地址
  - 寄存器间接寻址 LW rt, (r_{base})
    用寄存器中的值作为地址
  - 基址寻址 LW rt, offset(r_{base})
    用基址寄存器中的内容加偏移量作为地址
  - 变址寻址 LW rt, (r_{base}, r_{index})
    用变址寄存器加偏移量构成地址
  - 内存间接寻址 LW rt (((r_{base}))
    使用寄存器值指向的内存单元中的值作为地址
  - 自动增/减寻址 LW Rt, (r_{base})
    使用寄存器值作为地址，但是每次将该寄存器值增加或减少一个量

35

35

采用不同的寻址方式有什么好处?

- 程序员和微架构做折衷的又一个例子
- 采用多种寻址方式的好处：
  - 具备更好的向高层映射的能力：换一种模式可能会是更好的表达方式，能够有效的减少指令数量和代码的尺寸
    - 数组的访问（自增模式）
    - 间接寻址（指针相关的结构）
    - 稀疏矩阵的访问
- 缺点：
  - 编译器要做更多的工作
  - 微架构要做更多的工作

36

36

ISA 的正交性

- 正交的 ISA:
  - 所有寻址方式对所有指令都有效
  - 例如: VAX
    - (~13 种寻址方式) x (>300 种操作码) x (整型 和浮点处理)
- 有什么好处?
- 有什么不好?

37

37

ISA的要素（VII）

- 与 I/O 设备的接口
  - 内存映射的I/O
    - 内存的一个区域映射到 I/O 设备
    - I/O 操作会通过对这一内存区域的存取来完成
  - 特殊的 I/O 指令
    - x86 架构采用专门的 IN 和 OUT 指令来处理I/O
  - 折衷?
    - 哪一种方法更通用?

38

38

ISA的要素（VIII）

- 特权模式
  - 用户 vs 超级管理员
  - 谁可以执行什么指令?
- 异常和中断处理
  - 当一条指令出了问题之后会发生什么?
  - 当一个外部设备请求处理器之后会发生什么?
  - 向量中断 vs. 非向量中断 (早期的 MIPS)
- 虚拟内存
  - 每一个程序都貌似拥有整个内存空间，而且是超过物理内存容量的
- 访问控制

39

39

复杂的指令 vs. 简单的指令

- 复杂的指令: 一条指令要做很多事，比如做很多操作
  - 插入一个双向链表
  - 计算 FFT
  - 串的复制
- 简单指令: 一条指令只做很少的事，是可以用来构建复杂操作的原语
  - Add
  - XOR
  - Multiply

40

40

复杂的指令 vs. 简单的指令

- 复杂指令的好处
  - + 编码的密度大→ 代码尺寸小 → 更好的内存利用率，节省片外带宽，更好的cache命中率(指令更自然的成组)
  - + 更简单的编译器：不需要优化太多的小指令
- 复杂指令的缺点
  - – 更大的功能块→ 编译器优化的余地小（很难做细粒度的优化）
  - – 更复杂的硬件→ 从高层实现到底层控制信号和优化的转换需要硬件完成

41

41

ISA的折衷：“语义鸿沟”

- ISA的位置 ？ 语义鸿沟
  - 靠近高级语言 → 语义鸿沟小，指令复杂
  - 靠近硬件控制信号 →语义鸿沟大，指令简单
- RISC vs. CISC
  - RISC：精简指令集计算机
  - CISC：复杂指令集计算机
    - FFT， 快排 ，浮点处理指令?
    - VAX 的INDEX 指令 （带边界检查的数组访问）

42

42

ISA的折衷：语义鸿沟

- 简单的编译器，复杂的硬件 vs. 复杂的编译器，简单的硬件
- 向后兼容性带来的问题
- 性能?
  - 优化的机会
  - 指令的尺寸，代码的尺寸

43

43

一个例子：X86（小语义鸿沟）的串操作

- 操作串的指令
  - 将一个任意长度的串移动到另一个位置
  - 比较两个串
- 通过在ISA中设计重复执行一条指令的能力来实现
  - 使用REP“前缀”
- 例如：REP MOVS 指令
  - 只占2个字节：REP 前缀字节和 MOVS 操作码字节
  - 隐含的源和目的寄存器(ESI，EDI)指向两个串
  - 隐含的计数寄存器 (ECX) 指明了串的长度

44

44

X86的串操作

REP MOVS (DEST SRC)

```
IF AddressSize = 16
    THEN
        Use CX for CountReg;
    ELSE IF AddressSize = 64 and REX.W used
        THEN Use RCX for CountReg; FI;
    ELSE
        Use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg ← (CountReg - 1);
        IF CountReg = 0
            THEN exit WHILE loop; FI;
        IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
        or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
            THEN exit WHILE loop; FI;
    OD;
```

```
DEST ← SRC;
IF (Byte move)
    THEN IF DF = 0
        THEN
            (R(E)SI ← (R(E)SI + 1;
            (R(E)DI ← (R(E)DI + 1;
        ELSE
            (R(E)SI ← (R(E)SI - 1;
            (R(E)DI ← (R(E)DI - 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (R(E)SI ← (R(E)SI + 2;
            (R(E)DI ← (R(E)DI + 2;
            FI;
        ELSE
            (R(E)SI ← (R(E)SI - 2;
            (R(E)DI ← (R(E)DI - 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (R(E)SI ← (R(E)SI + 4;
            (R(E)DI ← (R(E)DI + 4;
            FI;
        ELSE
            (R(E)SI ← (R(E)SI - 4;
            (R(E)DI ← (R(E)DI - 4;
        FI;
    ELSE IF (Quadword move)
        THEN IF DF = 0
            (R(E)SI ← (R(E)SI + 8;
            (R(E)DI ← (R(E)DI + 8;
            FI;
        ELSE
            (R(E)SI ← (R(E)SI - 8;
            (R(E)DI ← (R(E)DI - 8;
        FI;
FI;
```

MIPS当中需要多少条指令才能完成?

45

语义相关的折衷

- CISC vs. RISC
  - 复杂指令集计算机→ 复杂的指令
    - 最初的动机是代码生成的“不够好”
  - 精简指令集计算机→ 简单的指令
    - John Cocke, 1970年代中期, IBM 801
      - 目标：更好的编译器控制和优化
- RISC 的动机
  - 存储器停顿 (memory stall)
  - 简化硬件→ 更低的成本，更高的频率
  - 编译器可以更好的优化代码
    - 发掘细粒度并行减少存储器停顿

46

ISA位置的限制-鸿沟大小的极限

- 极大语义鸿沟
  - 每条指令指定一整套的控制信号
  - 编译器生成控制信号
  - 微码 (John Cocke, circa 1970s)
  - 优化编译器
- 极小语义鸿沟
  - ISA 几乎就是高级语言
  - LISP机

47

ISA的演进

- 今天的ISA已经可以反映几乎所有系统需要关注的方面
- 例如:
  - 有限的片上和片外存储器容量
  - 有限的编译器优化技术
  - 有限的存储带宽
  - 重要应用的特殊需求 (例如, MMX)
- 通过ISA转换(在硬件和软件中)的方法，使得不管什么样的ISA设计都可以采用类似的底层实现

48

ISA转换的效果

- 可以通过从一个ISA向另一个ISA的转换来改变语义鸿沟的折衷

- 例如
  - Intel 和 AMD 的 x86 实现在硬件层面将x86指令转换成程序员不可见的微操作(简单的指令)
  - Transmeta的 x86 实现在软件层面将 x86 指令转换成 VLIW 指令 (代码变形软件)

49

49

ISA的折衷：指令长度

- 固定长度：所有指令长度是一样的
  - \+ 硬件对单条指令译码更容易
  - \+ 同时对多条指令译码更容易
  - -- 会浪费指令中的某些位 (为什么不好?)
  - -- 不容易扩展 (如何增加新的指令?)
- 可变长度：指令长度不同 (由操作码和子操作码决定)
  - \+ 紧凑的编码 (为什么是优点?)
    Intel 432: 哈夫曼编码
  - -- 一条指令的译码需要更多的逻辑
  - -- 同时对多条指令译码很难

- Tradeoffs
  - 代码的尺寸 (内存空间，带宽，时延) vs. 硬件复杂性
  - ISA 的扩展性和描述能力
  - 性能? 紧凑的代码 vs.不完美的译码

50

50

ISA的折衷：统一的译码

- 统一译码：每条指令的相同位代表的意思相同
  - 操作码总是在同样的位置
  - 操作数说明符，立即数等也总在同样的位置 …
  - 为大多数“RISC”ISA所采用: Alpha, MIPS, SPARC
  - \+ 更容易译码，更简单的硬件
  - \+ 支持并行
  - -- 严格的指令格式 (更少的指令?) ，浪费空间

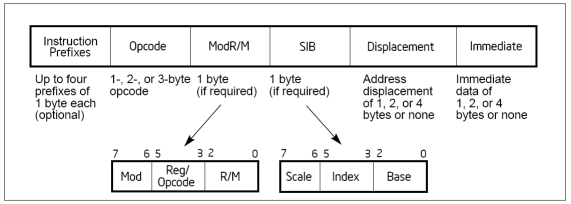
- 非统一译码
  - 例如，x86的操作码可能从第1-第7个字节
  - \+ 更紧凑也更强大的指令格式
  - --更复杂的译码逻辑

51

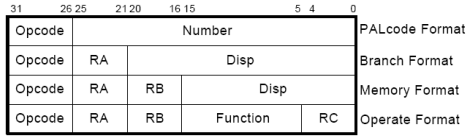
51

x86 vs. Alpha 指令格式

- x86:



- Alpha:

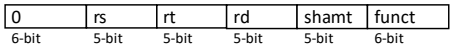


52

52

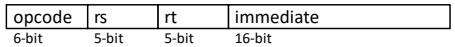
MIPS 指令格式

- R-type, 3 寄存器操作数



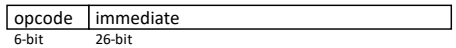
R-type

- I-type, 2 寄存器操作数和 16-bit 立即操作数



I-type

- J-type, 26-bit 立即操作数



J-type

- 简单的译码
  - 32位指令字长，不管是什么类型的指令
  - 必须做4字节对齐（PC的最低两位必须是00）
  - 硬件非常容易提取指令格式和字段

53

53

关于长度和统一

- 统一译码通常伴随着固定长度
- 对于可变长度的ISA，那些相同长度的指令也可以统一译码
  - 不同长度的指令很难作统一译码

54

54

ISA的折衷：寄存器个数

- 影响：
  - 用于译码寄存器地址的位数
  - 在快速的存储介质（寄存器堆）中保存的值的个数
  - （对微架构的影响）寄存器堆的大小、访问时间、功耗等
- 较多的寄存器数目：
  - \+ 编译器可以更好的分配和优化寄存器 → 更少的保存/恢复操作
  - -- 更大的指令尺寸
  - -- 更大的寄存器堆

55

55

RISC vs. CISC

- RISC
  - 指令简单
  - 固定长度
  - 统一译码
  - 寻址方式少
- CISC
  - 指令复杂
  - 可变长度
  - 非统一译码
  - 寻址方式多

56

56

RISC基本设计思想

- CPI：平均时钟周期数
- 目标：减小CPI
  - CPU时间=（IC × CPI）/时钟频率
- 方法1：保留最常用指令
  - 去掉复杂、使用频度不高的指令
- 方法2：采用Load/Store结构
  - 大大减少指令格式，统一了存储器访问方式
- 方法3：采用硬接线控制代替微程序控制

59

57

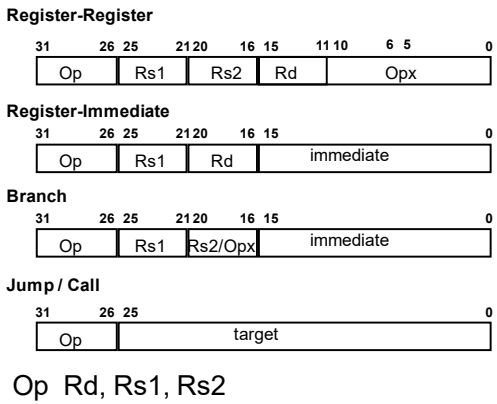
RISC结构的特点

- CPI接近于1
  - 大多数指令单周期完成
- Load/Store指令结构
- 寻址方式少，指令格式少且规整，指令长度统一(比如32bit)，
  - 便于提高流水线效率
- 便于编译优化
- 硬接线控制器

60

58

例子： MIPS



Op Rd, Rs1, Rs2

61

59

MIPS的典型特点

- 32位固定格式指令(3种格式)
- 32个32位GPR
- 3地址、寄存器－寄存器算术指令
- load/store单一寻址模式：
  - 基地址+偏移
  - 无间接寻址
- 简单的分支指令

62

60

高性能RISC处理器

- SUN公司的SPARC
- MIPS公司的SGI:MIPS
- HP公司的PA-RISC,
- IBM, Motorola公司的PowerPC
- DEC公司的Alpha

63

61

其它有关ISA的折衷

- 有 vs. 无状态码
- VLIW vs. 单指令
- 精确 vs. 非精确异常
- 有vs. 无虚拟存储
- 对齐 vs. 非对齐访问
- 硬件互锁 vs. 软件保证的互锁
- 软件 vs. 硬件管理的页失效处理
- Cache 一致性 (硬件 vs. 软件)
- …

62

62

程序员vs.(微)体系结构

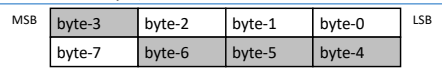
- 很多ISA的特性是设计用来帮助程序员的
- 但是会使硬件设计者的工作更复杂

- 虚拟存储
  - vs. overlay 编程
  - 程序员应该关心代码块大小是否与物理内存匹配吗?
- 寻址方式
- 对齐的内存访问
  - 编译器/程序员需要对齐数据

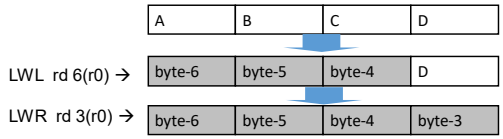
63

63

MIPS: 对齐访问



- LW/SW 对齐约束: 4-byte 字-对齐
  - 只设计了在一个字边界内取内存字节的功能
  - 不支持将未对齐的字节放入寄存器
- 对不常见的情况提供独立的操作码



- LWL/LWR 执行速度更慢
- LWL 和 LWR 仍然在一个字边界内取数据

64

64

X86:对齐访问

- LD/ST 指令自动对齐跨越字边界的数据
- 程序员/编译器不需要顾及数据存储的位置（不管字对齐还是未对齐）

4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

65

65

X86:对齐访问

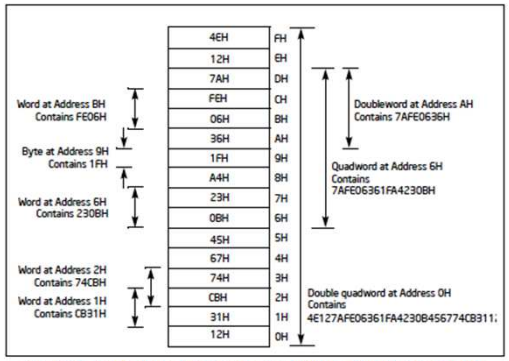


Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

66

66

对齐vs. 非对齐访问

- 不限制对齐的优点
- 不限制对齐的缺点
- 练习：填写上面的空白…

67

67

ISA实现：微体系结构基础

68

68

机器如何处理指令?

- 处理指令是什么意思?
- 冯诺依曼模型/结构

A = 指令执行之前程序员可见的体系结构状态

↓

处理指令

↓

A'= 指令执行之后程序员可见的体系结构状态

- 处理指令：根据ISA的指令规范将A变换成A'

69

69

“处理指令”的步骤

- ISA 抽象地说明给定一条指令和A，A’应该是什么
  - 定义一个抽象的有限状态机
    - 状态 = 程序员可见的状态
    - 次态逻辑 = 指令执行的规范
  - 从 ISA 的视角，指令执行的过程中A和A’之间没有“中间状态”
    - 每条指令对应一个状态转换
- 微体系结构实现 A 向 A’的转换
  - 有很多种实现方式的选择
  - 我们可以加入程序员不可见的状态来优化指令执行的速度：每条指令有多个状态转换
    - 选择 1：A → A’（在一个时钟周期内完成 A 到 A’的转换）
    - 选择 2：A → A+MS1 → A+MS2 → A+MS3 → A’（使用多个时钟周期完成 A 到 A’的转换）

70

70

最基本的指令处理引擎

- 每条指令花费一个时钟周期来执行
- 只用组合逻辑来实现指令的执行
  - *没有中间的、程序员不可见的状态更新*

A = 时钟周期开始时的体系结构状态（程序员可见）

↓

在一个时钟周期内处理指令

↓

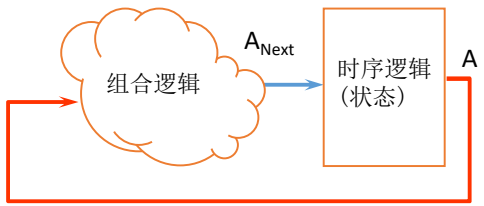
A’ = 时钟周期结束时的体系结构状态(程序员可见)

71

71

最基本的指令处理引擎

- 单周期机器

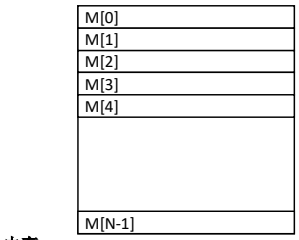


- 时钟周期长度由谁来决定?
- 组合逻辑中的关键路径由谁来决定?

72

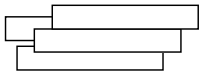
72

程序员可见(体系结构)的状态



内存

用地址索引的存储位置的阵列



寄存器

- 在ISA中会给寄存器命名
  (相当于地址)
- 通用和专用寄存器

程序计数器

当前指令的内存地址

指令和程序指定如何转换程序员可见的状态值

73

单周期 vs. 多周期

- 单周期的机器
  - 每条指令执行需要一个时钟周期
  - 所有状态的更新在指令执行结束的时刻完成
  - 劣势：最慢的指令决定时钟周期的长度 → 时钟周期时间长
- 多周期的机器
  - 指令处理分到多个周期/阶段中完成
  - 指令执行过程中可以更新状态
  - 但是体系结构状态的更新只能在指令执行结束的时刻完成
  - 与单周期相比的“优势”：最慢的“阶段”决定时钟周期长度
    - 单周期和多周期在微体系结构层面都遵从冯诺依曼结构

74

指令处理“周期”

- 指令在“控制单元”的指示下一步一步地处理
- 指令周期：指令处理的步骤序列
- 从根本上说，指令处理大约分为6个阶段：
  - 取指令
  - 译码
  - 计算地址
  - 取操作数
  - 执行
  - 存结果
- 不是所有的指令都需要所有6个阶段

75

指令处理“周期” vs. 机器时钟周期

- 单周期的机器：
  - 指令处理周期的所有阶段都在*一个机器时钟周期*中完成
- 多周期的机器：
  - 指令处理周期的所有阶段可以在*多个机器时钟周期*中完成
  - 实际上，每个阶段都可以在多个时钟周期中完成

76

观察指令处理的另一个视角

- 指令将数据(AS)转换成数据（AS'）
- 由功能单元完成转换
  - "操作"数据的单元
- 需要有人告诉这些单元对数据做什么操作
- 一个指令处理的引擎由两部分组件构成
  - 数据通路：由处理和转换数据信号的硬件部件组成
    - 操作数据的功能单元
    - 存储数据的存储单元（比如寄存器）
    - 使数据流能够流入功能单元和寄存器的硬件结构（比如连线和多路选择器）
  - 控制逻辑：由决定控制信号的硬件部件组成，这些控制信号决定了数据通路上的部件会如何操作数据

77

77

单周期vs. 多周期:控制&数据

- 单周期的机器:
  - 数据信号操作的同时产生控制信号（在同一个时钟周期内起作用）
  - 与一条指令相关的所有事情都发生在一个时钟周期内
- 多周期的机器:
  - 下一个周期需要的控制信号可以在前一个周期就产生
  - 数据通路上的延迟可以和控制处理的延迟重叠

78

78

数据通路和控制逻辑的设计方法很多

- 有很多方法可以用来设计数据通路和控制逻辑
- 单周期，多周期，流水线等
- 单总线vs. 多总线数据通路
- 硬连线/组合逻辑vs. 微码/微程序控制
  - 由组合逻辑电路产生控制信号
  - 在存储器结构中存储控制信号
- 控制信号和结构依赖于数据通路的设计

79

79

初步的性能分析

- 指令执行时间
  - {CPI} x {clock cycle time}
- 程序执行时间
  - 所有指令的[{CPI} x {clock cycle time}]之和
  - {指令数} x {平均 CPI} x {clock cycle time}
- 单周期微体系结构的性能
  - CPI = 1
  - Clock cycle time 长
- 多周期微体系结构的性能
  - CPI = 每条指令不同
    - 平均 CPI → 希望能很小
  - Clock cycle time 短

现在，我们有两个独立的自由度可以优化

80

80