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
2019-03-13

1

提醒：作业

- 作业 0
  - 今天是提交截止日
- 作业 1
  - 今天发布，3月27日截止提交，通过课程网站
  - MIPS 和 ISA 的基本概念，基本的性能评价

2

提醒：实验 1

- 用Logisim设计1个7指令单周期MIPS CPU
  - 掌握处理器的基本原理、结构、构造方法
  - 今天发布，预计完成截止时间为4月8日
  - 本科是北航计算机专业的同学免做，只需要在提交的实验报告中写明本科学号和当时的计组实验成绩即可
- 需要提前启动，因为你们要学习很多东西

3

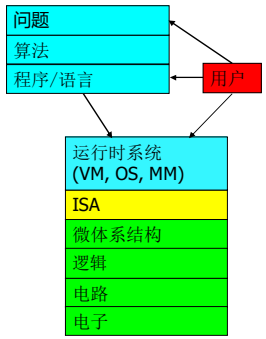
本讲相关阅读材料

- 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

4

上一讲回顾

- 抽象的力量
  - 为什么要跨越抽象层次
- 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。**
- 今天的计算机体系结构



5

5

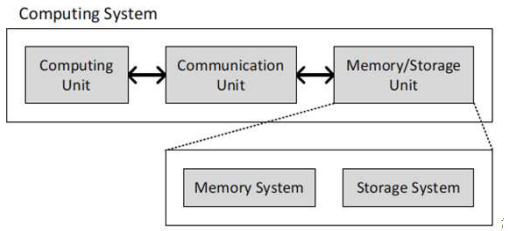
基本概念

6

6

什么是计算机?

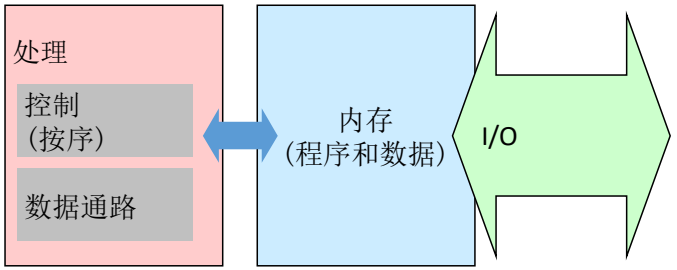
- 三个关键的要素
- 计算
- 通信
- 存储 (内存)



7

什么是计算机?

- 我们会讨论所有这三个要素



8

8

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

什么时候一串数字会被解释成一条指令呢?

9

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

什么时候一串数字会被解释成一条指令呢?

- 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

10

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

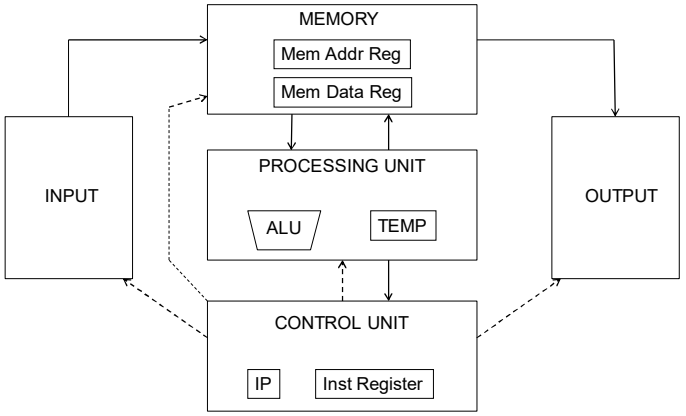
什么时候一串数字会被解释成一条指令呢?

- 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”

11

冯诺依曼结构/模型(计算机)



12

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

13

13

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

14

14

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

15

15

冯诺依曼vs 数据流

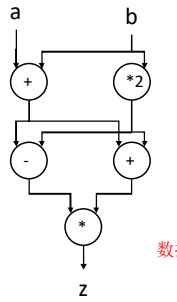
■考虑一个冯诺依曼结构下的程序

❑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

16

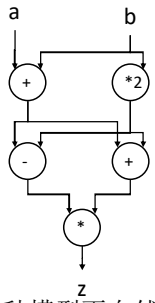
16

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

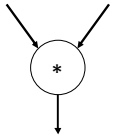
■如果你是程序员，你觉得哪一种模型更自然?

17

17

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

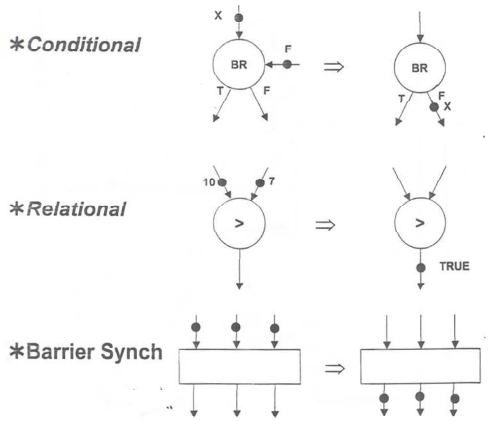


*	R	ARG1	R	ARG2	Dest. Of Result

18

18

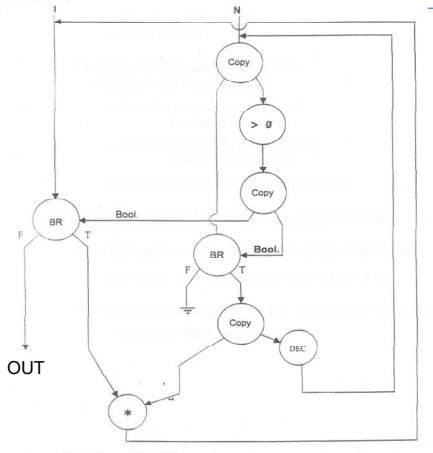
数据流节点



19

19

数据流程序的例子



20

20

指令指针—ISA层面的折衷

■是否需要在ISA中设计指令指针?

21

21

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)

22

22

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?

23

23

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

24

24

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

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

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.

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

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

29

29

ISA vs. 微体系结构

- ISA
  - 约定软硬件之间的接口
  - 软件开发者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

30

30

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, …

31

31

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
  - 只有几种有限的 ISA (x86, ARM, SPARC, MIPS, Alpha) 但是有很多种微架构
  - *Why?*

32

32

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

33

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

34

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
- ……

35

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

36

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

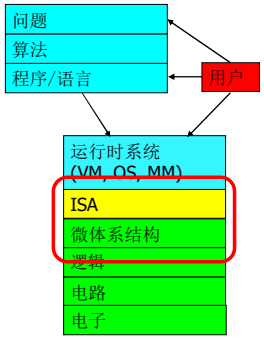
- 设计要点由“问题”空间 (应用)或者面向的用户/市场决定

37

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*

38

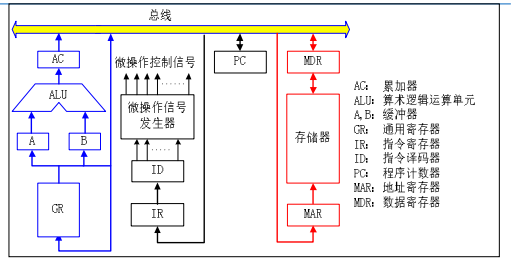


39

ISA 的设计原则和折衷

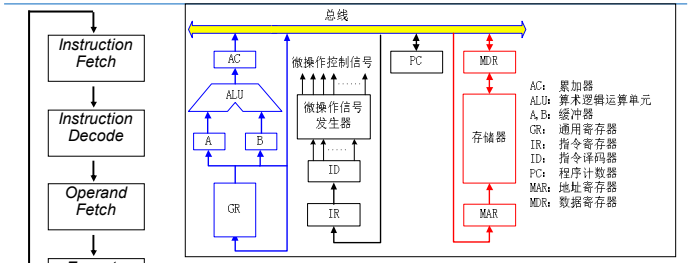
40

指令的执行过程



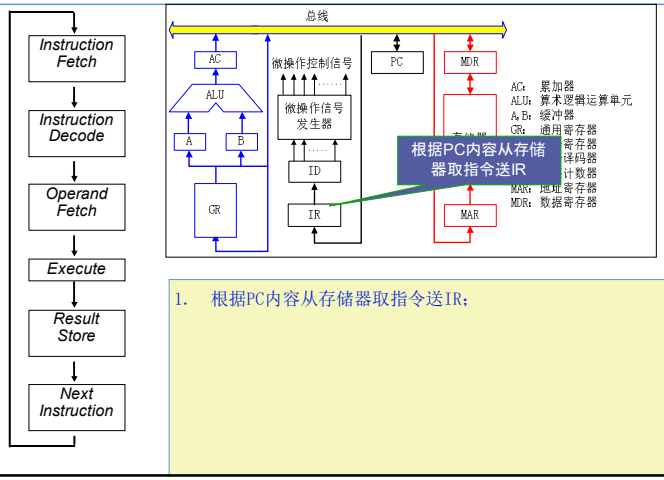
41

指令的执行过程



42

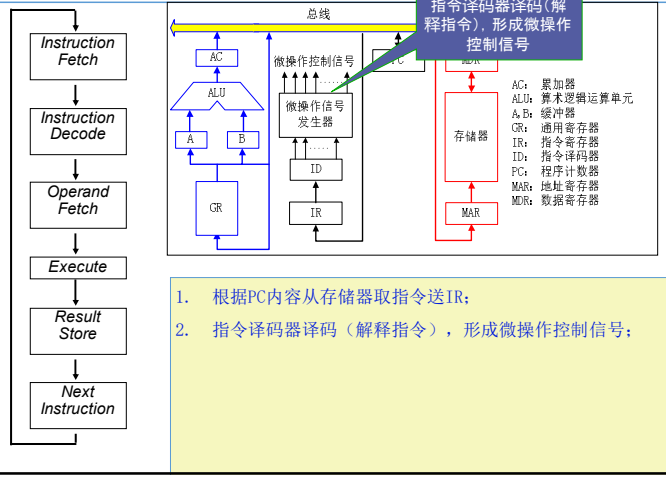
指令的执行过程



1. 根据PC内容从存储器取指令送IR；

43

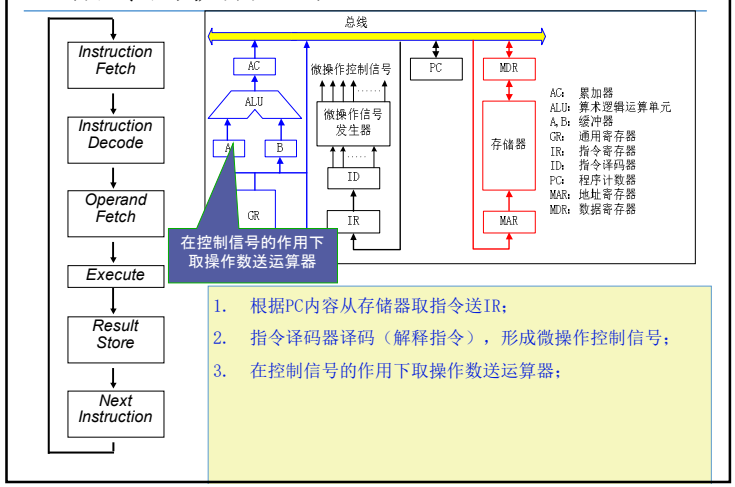
指令的执行过程



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；

44

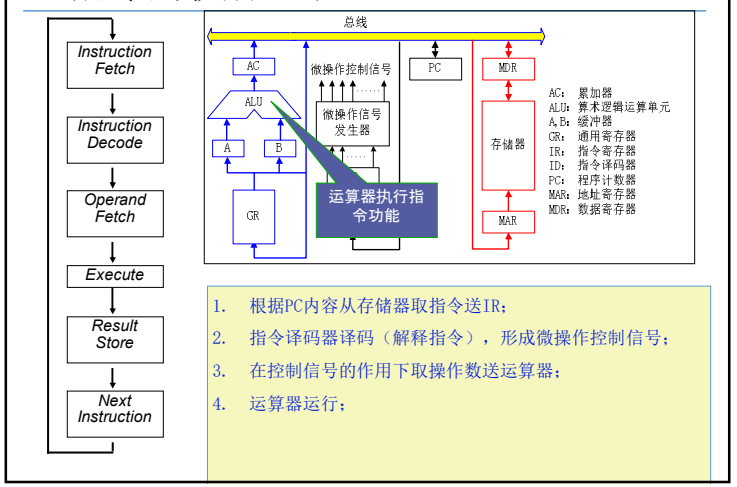
指令的执行过程



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；

45

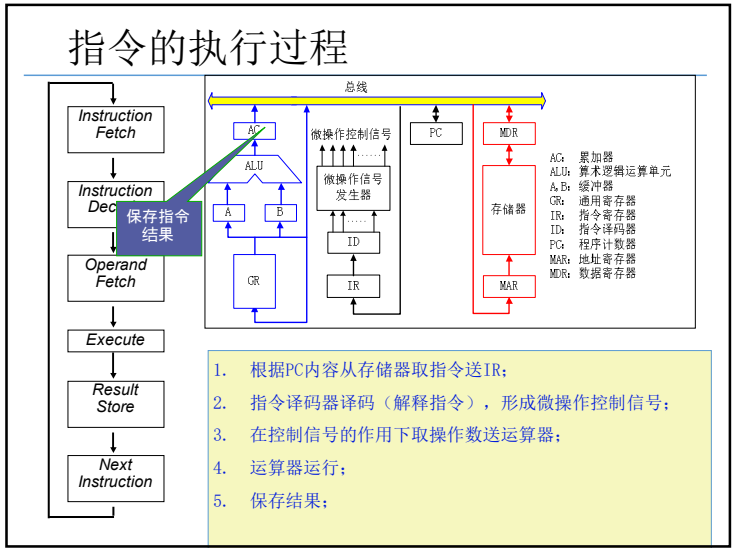
指令的执行过程



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；

46

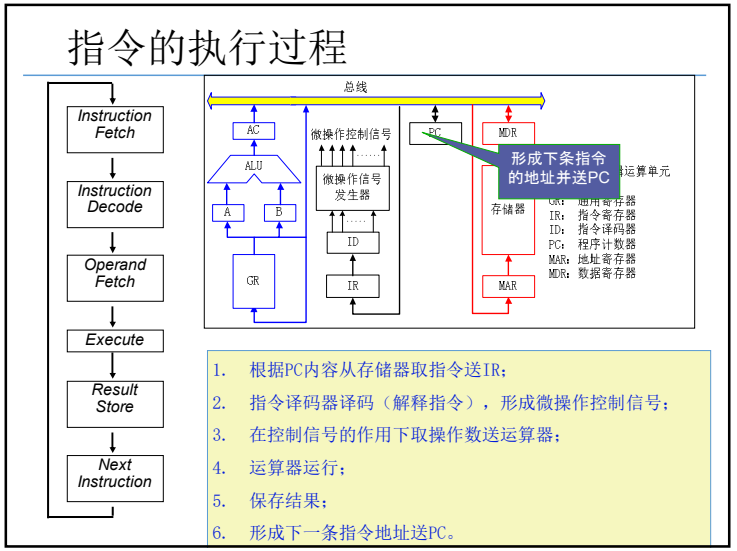
指令的执行过程



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；

47

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。

48

Example

Y=ax²+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

49

Example

Y=ax²+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 ←ax²+bx
SUB c	0216H	AC ←ax² + 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

50

Example

AC ???　开始PC 00H

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)

内存	地址
0112H	00H
0318H	02H
0014H	04H
0318H	06H
0216H	08H
0410H	0AH
	0CH
	0EH
结果y	10H
值a	12H
值b	14H
值c	16H
值x	18H

51

Example

AC a　PC 02H

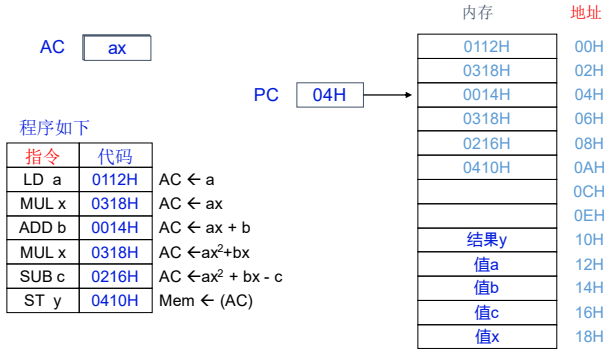
程序如下

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)

内存	地址
0112H	00H
0318H	02H
0014H	04H
0318H	06H
0216H	08H
0410H	0AH
	0CH
	0EH
结果y	10H
值a	12H
值b	14H
值c	16H
值x	18H

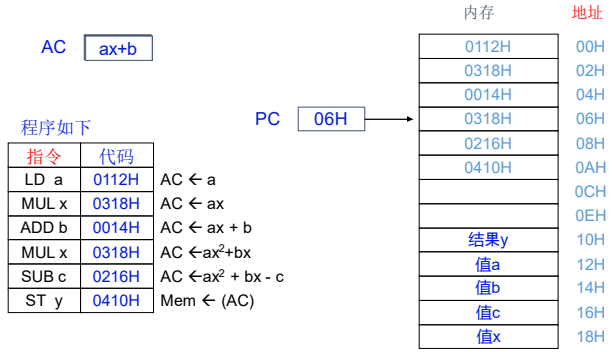
52

Example



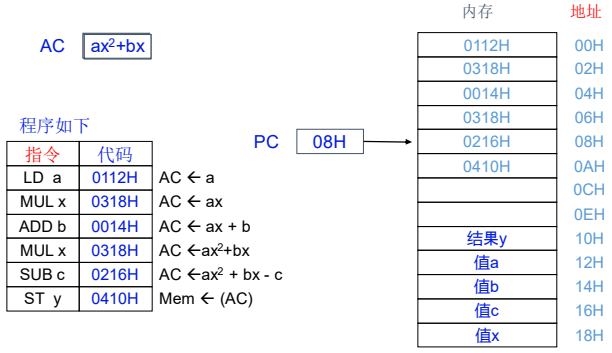
53

Example



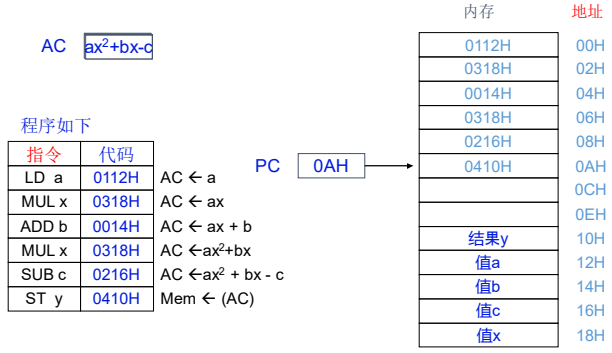
54

Example



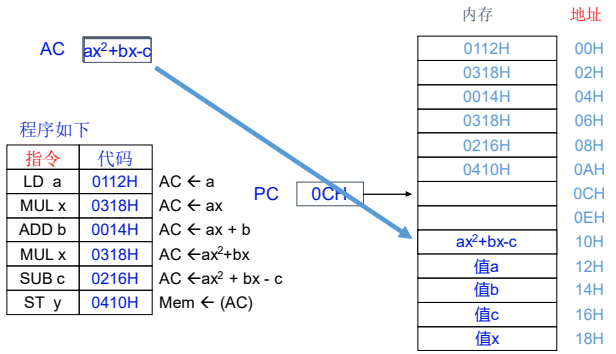
55

Example



56

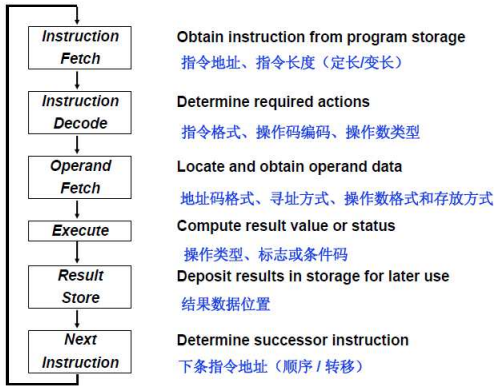
Example



57

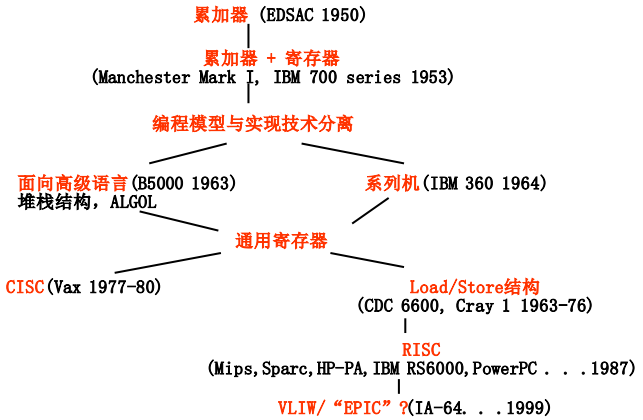
指令系统

- 从指令执行周期看指令涉及的内容



58

指令系统的演变



59

各种不同的指令集体系结构（ISA）

- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine CM-1
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPARC, ARM

60

60

各种不同的指令集体系结构（ISA）

- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connecti
- VLIW ISAs: Multiflow, Cydror
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPAR

61

61

各种不同的指令集体系结构（ISA）

- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRA
- VLIW ISAs: Mul
- PowerPC, POW
- RISC ISAs: Alpha

62

62

各种不同的指令集体系结构（ISA）

- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine CM-1
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPARC, ARM

- 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

63

63

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

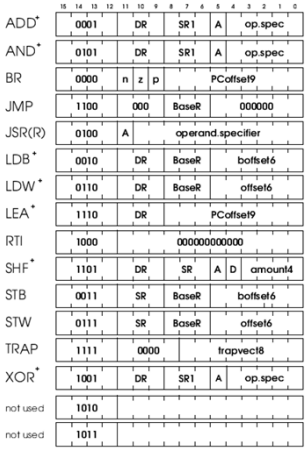
-

31 26	25 21	20 16	15 5	4 0	
Opcode	Number				PALcode Format
Opcode	RA	Disp			Branch Format
Opcode	RA	RB	Disp		Memory Format
Opcode	RA	RB	Function	RC	Operate Format

64

64

指令集

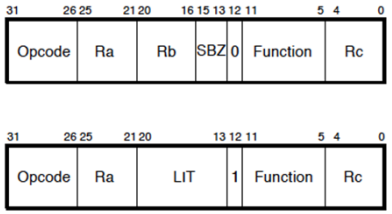


- LC-3b ISA
  - Patt & Patel
- "bit steering"
  - 指令中的某一位决定指令中其它位的含义
- 为什么要有"not used"的指令?

65

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:

66

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

67

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

68

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

69

69

例子: Stack Machine (II)

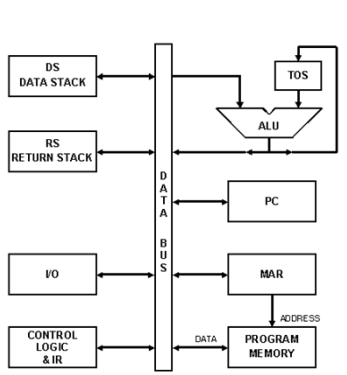


Figure 3.1 -- The canonical stack machine.

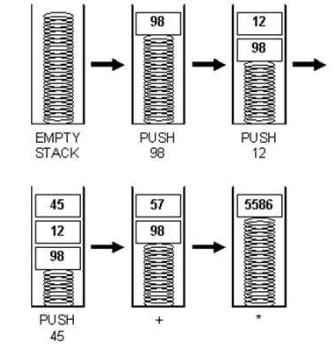


Figure 3.2 -- An example stack machine.

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

70

70

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?

71

71

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

72

72

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?

73

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

74

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？

75

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

76

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

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

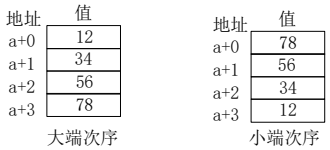
ISA的要素（III）

- 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

81

81

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?

82

82

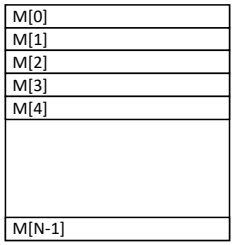
ISA的要素（IV）

- 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

83

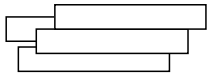
83

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

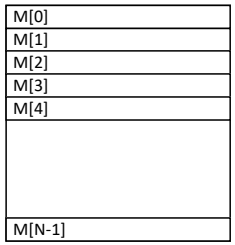
程序计数器

当前指令的内存地址

84

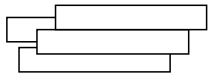
84

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

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

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）

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