高等计算机体系结构
第三讲: 单周期和多周期微体系结构

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

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，预习4.5-4.8, 流水线)
  - 附录 D
- 选读
  - Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.
  - Patt & Patel's *Introduction to Computing Systems: From Bits and Gates to C and Beyond* （计算机系统概论）
    - 附录C ： LC-3b ISA及微体系结构

2

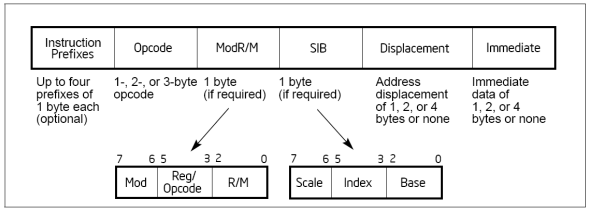
回顾: ISA Tradeoffs

- 复杂指令与简单指令: semantic gap
- 利用"翻译"的方法改变tradeoff策略
- 固定长度与可变长度, 统一与非统一译码
- 寄存器个数
- 执行指令之前把复杂指令翻译成"简单"指令有什么好处?
  - 硬件 (Intel, AMD)?
  - 软件 (Transmeta)?
- 哪一种 ISA 更容易扩展: 固定长度 还是 可变长度?
- 如何拥有可变长度、统一译码的 ISA?

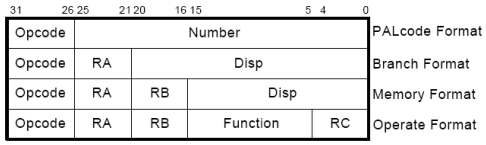
3

回顾: x86 vs. Alpha 指令格式

- x86:



- Alpha:



4

回顾：ISA折衷-寄存器个数

- 影响：
  - 译码寄存器地址需要的位数
  - 寄存器堆中保存的数值数
  - (对微架构)寄存器堆的大小、访问时间、功耗等

- 寄存器数量多：
  - \+ 编译器可以更好的分配和优化寄存器 → 更少的存储/恢复
  - -- 更长的指令
  - -- 更大的寄存器堆

回顾：ISA折衷—寻址方式

- 寻址方式说明如何获得指令的操作数
  - 寄存器
  - 立即数
  - 存储器（偏移量、寄存器间接、变址、绝对、存储器间接、自增、自减 …）

- 其它方式：
  - \+ 更好地帮助支持编程（数组、基于指针的访问）
  - -- 设计更困难
  - -- 编译器面临更多选择
    - 使编译器设计更复杂
    - *Wulf, "Compilers and Computer Architecture," IEEE Computer 1981*

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

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

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

9

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)
- …

10

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

11

ISA实现：微体系结构基础

12

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'

13

13

“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)

14

14

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

↓

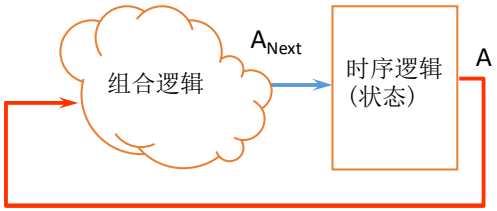
A’ = 时钟周期结束时的体系结构状态(程序员可见)

15

15

最基本的指令处理引擎

- 单周期机器

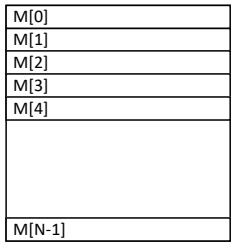


- 时钟周期长度由谁来决定?
- 组合逻辑中的关键路径由谁来决定?

16

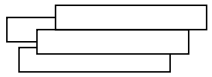
16

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

17

17

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

18

18

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

19

19

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

20

20

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

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

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

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

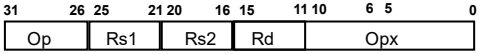
现在，我们有两个独立的自由度可以优化

单周期微体系结构——
近距离观察

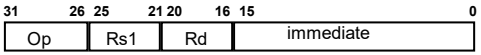
25

例子： MIPS

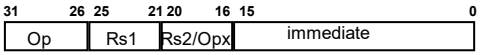
Register-Register



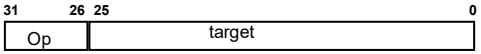
Register-Immediate



Branch



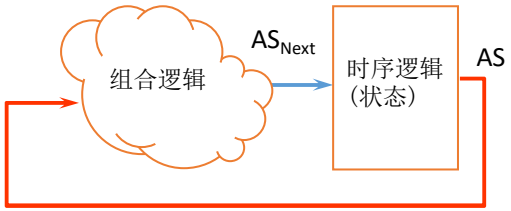
Jump / Call



Op Rd, Rs1, Rs2

26

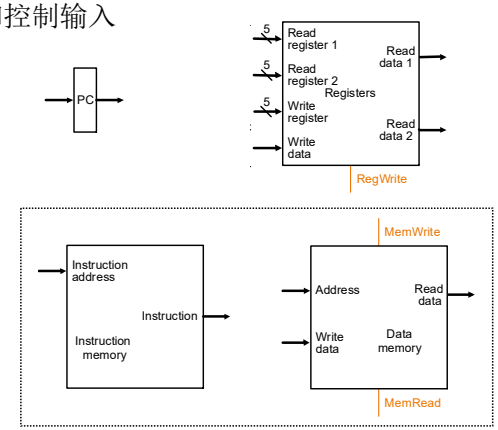
- 单周期的机器



27

从状态单元开始

- 数据和控制输入



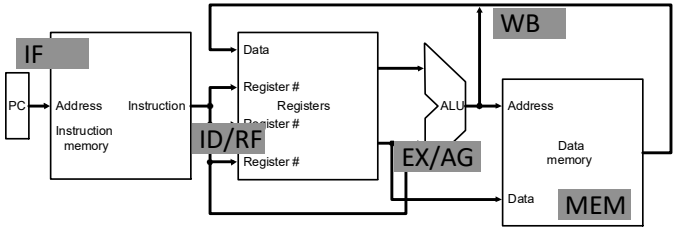
28

现在，我们假设

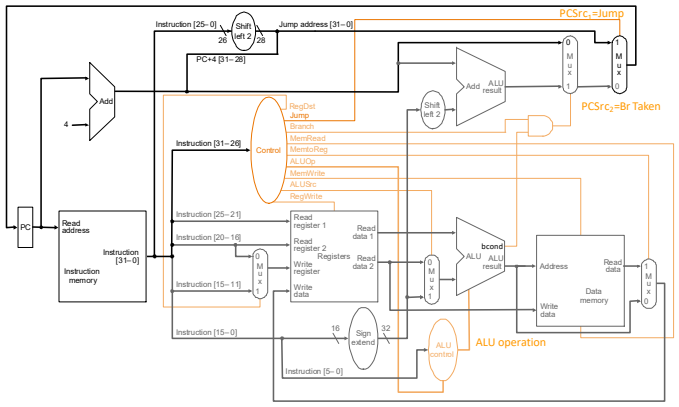
- “理想化”内存和寄存器堆
- 组合读
  - 读数据端口的输出是寄存器堆中内容和相应的读端口地址的组合函数
- 同步写
  - 写使能信号有效时，被选定的寄存器在时钟信号上升沿更新
    - 不会影响时钟沿之间的寄存器读输出
    - 会影响时钟沿上的寄存器读输出 （无所谓?）
- 单周期，同步存储
  - 内存读写需要确保数据准备好

指令处理

- 5个一般步骤 (Patterson & Hennessy’s Book)
  - 取指令 (IF)
  - 指令译码和取寄存器操作数 (ID/RF)
  - 执行/计算内存地址 (EX/AG)
  - 取内存操作数 (MEM)
  - 存储/写回结果 (WB)



完整的数据通路



忽略JAL, JR, JALR

单周期数据通路——
算术和逻辑指令

R类型ALU指令

- 汇编指令（例如，寄存器-寄存器带符号加法）

  ADD rd_{reg} rs_{reg} rt_{reg}

- 机器码

0	rs	rt	rd	0	ADD	R-type
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit	

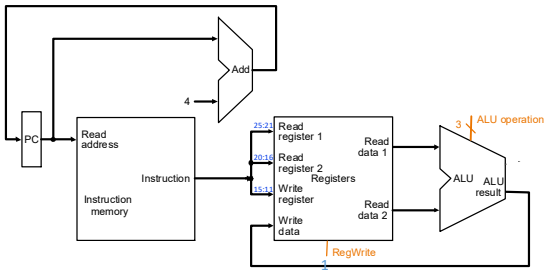
- 语义

```
if MEM[PC] == ADD rd rs rt
        GPR[rd] ← GPR[rs] + GPR[rt]
        PC ← PC + 4
```

33

33

R类型ALU指令数据通路




```
if MEM[PC] == ADD rd rs rt
        GPR[rd] ← GPR[rs] + GPR[rt]
        PC ← PC + 4
```

IF	ID	EX	MEM	WB

状态更新的组合逻辑

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

34

34

I类型ALU指令

- 汇编指令（例如，寄存器-立即数带符号加法）

  ADDI rt_{reg} rs_{reg} $immediate_{16}$

- 机器码

ADDI	rs	rt	immediate	I-type
6-bit	5-bit	5-bit	16-bit	

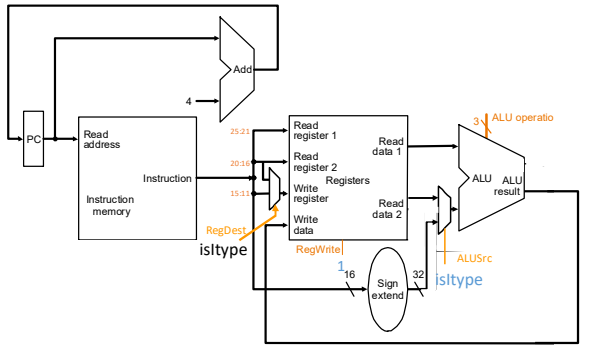
- 语义

```
if MEM[PC] == ADDI rt rs immediate
        GPR[rt] ← GPR[rs] + sign-extend (immediate)
        PC ← PC + 4
```

35

35

R类型和I类型ALU指令数据通路




```
if MEM[PC] == ADDI rt rs immediate
        GPR[rt] ← GPR[rs] + sign-extend (immediate)
        PC ← PC + 4
```

IF	ID	EX	MEM	WB

状态更新的组合逻辑

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

36

36

单周期数据通路——
数据移动类指令

37

37

Load

- 汇编指令（例如，load 4-byte的字）

  LW rt_{reg} $offset_{16}$ ($base_{reg}$)

- 机器码

LW	base	rt	offset	I-type
6-bit	5-bit	5-bit	16-bit	

- 语义

  if MEM[PC]==LW rt $offset_{16}$ (base)

  EA = sign-extend(offset) + GPR[base]

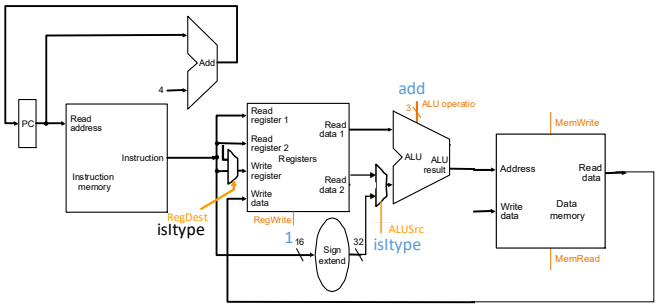
  GPR[rt] ← MEM[EA]

  PC ← PC + 4

38

38

LW 数据通路



if MEM[PC]==LW rt $offset_{16}$ (base)
EA = sign-extend(offset) + GPR[base]
GPR[rt] ← MEM[EA]
PC ← PC + 4

IF	ID	EX	MEM	WB

状态更新的组合逻辑

39

Store

- 汇编指令（例如，store 4-byte的字）

  SW rt_{reg} $offset_{16}$ ($base_{reg}$)

- 机器码

SW	base	rt	offset	I-type
6-bit	5-bit	5-bit	16-bit	

- 语义

  if MEM[PC]==SW rt $offset_{16}$ (base)

  EA = sign-extend(offset) + GPR[base]

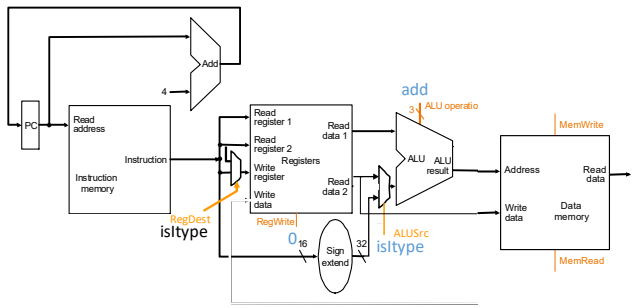
  MEM[EA] ← GPR[rt]

  PC ← PC + 4

40

40

SW 数据通路



if MEM[PC]==SW rt offset$_{16}$ (base)
EA = sign-extend(offset) + GPR[base]
MEM[EA] ← GPR[rt]
PC ← PC + 4

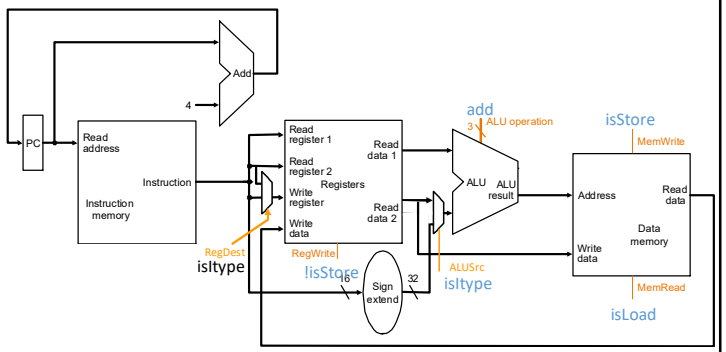
IF	ID	EX	MEM	WB

状态更新的组合逻辑

50

41

Load-Store 数据通路

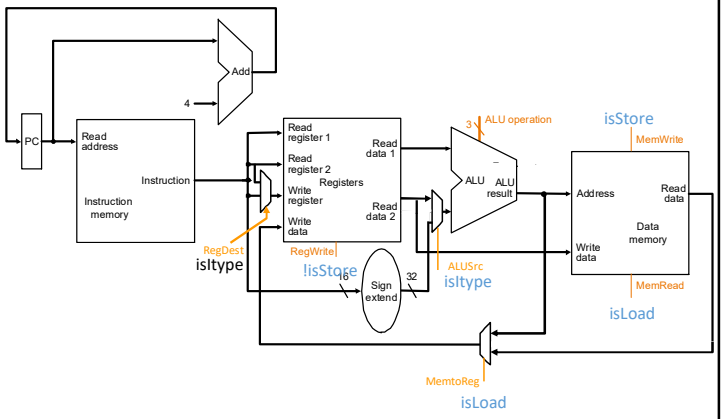


**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

42

42

不含控制流指令的数据通路



**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

43

43

单周期数据通路——*控制流指令*

44

44

无条件转跳指令

- 汇编指令

  J immediate$_{26}$

- 机器码

J	immediate	J-type
6-bit	26-bit	

- 语义

  if MEM[PC]==J immediate$_{26}$

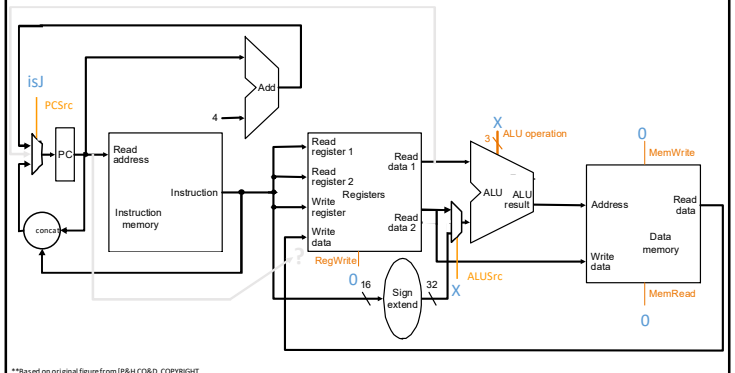
  target = { PC[31:28], immediate$_{26}$, 2'b00 }

  PC ← target

45

45

无条件转跳数据通路



**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

if MEM[PC]==J immediate26
PC = { PC[31:28], immediate26, 2'b00 }

JR, JAL, JALR?

46

条件分支指令

- 汇编指令（例如, branch if equal）

  BEQ rs$_{reg}$ rt$_{reg}$ immediate$_{16}$

- 机器码

BEQ	rs	rt	immediate	I-type
6-bit	5-bit	5-bit	16-bit	

- 语义（假设没有分支延迟槽）

  if MEM[PC]==BEQ rs rt immediate$_{16}$

  target = PC + 4 + sign-extend(immediate) x 4

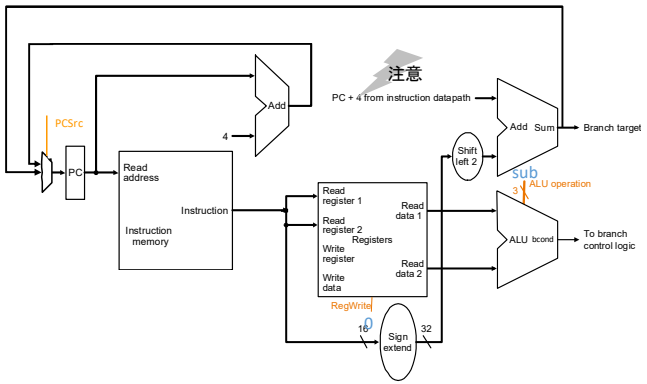
  if GPR[rs]==GPR[rt] then PC ← target

  else PC ← PC + 4

47

47

条件分支数据通路

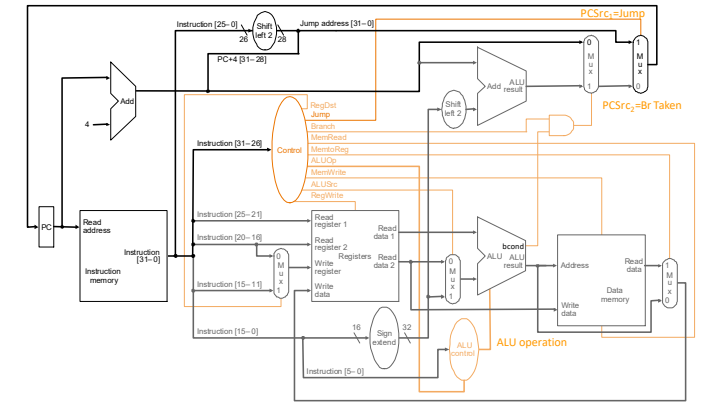


**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

48

48

数据通路整合



忽略 JAL, JR, JALR

49

49

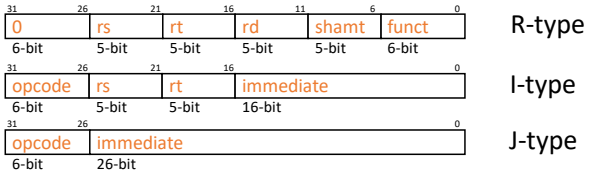
单周期控制逻辑

50

50

单周期硬连线控制

- Inst=MEM[PC]的组合函数



- 考虑
  - 所有All R-type 和 I-type ALU 指令
  - LW 和 SW
  - BEQ, BNE, BLEZ, BGTZ
  - J, JR, JAL, JALR

51

51

1-Bit 控制信号

	无效（=0）	有效（=1）	判断条件
RegDest	寄存器堆写入地址为rt, 即 inst[20:16]	寄存器堆写入地址为rd, 即 inst[15:11]	opcode==0
ALUSrc	ALU的第二个输入来自寄存器堆的第二个读出端口	ALU的第二个输入来自16位立即数的符号扩展	(opcode!=0) && (opcode!=BEQ) && (opcode!=BNE)
MemtoReg	ALU 的输出结果写入寄存器堆的写入端口	内存load出来的结果写入寄存器堆的写入端口	opcode==LW
RegWrite	寄存器堆写无效	寄存器堆写使能	(opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR))

52

52

1-Bit 控制信号

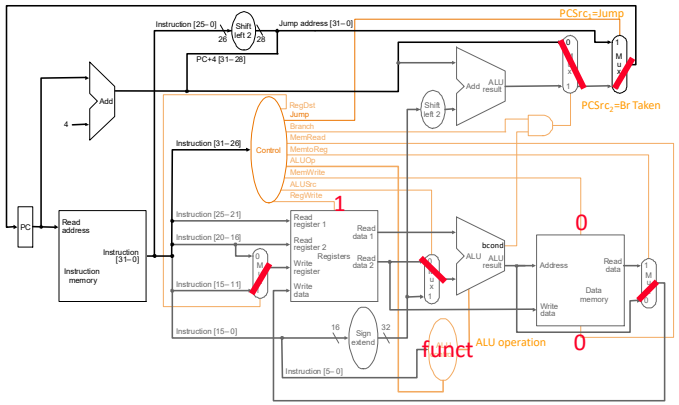
	无效（=0）	有效（=1）	判断条件
MemRead	内存读无效	内存读端口返回 load的值	opcode==LW
MemWrite	内存写无效	内存写使能	opcode==SW
$PCSrc_1$	由 $PCSrc_2$ 决定	下一个 PC 由26位立即数决定无条件转跳目标	(opcode==J) || (opcode==JAL)
$PCSrc_2$	PC = PC + 4	下一个 PC 由16位立即数决定分支转跳目标	(opcode==Bxx) && “bcond is satisfied”

ALU 控制信号

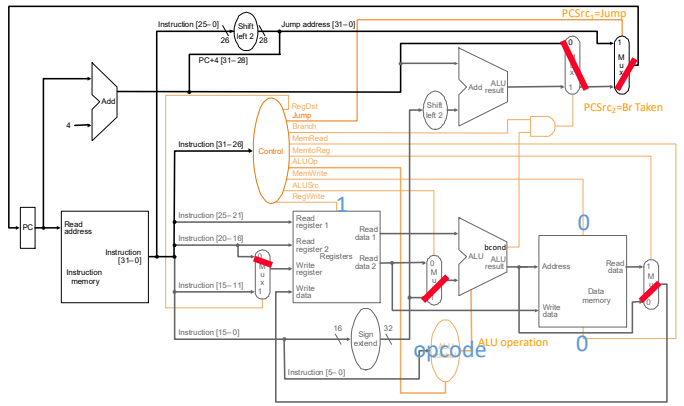
- case opcode
  - ‘0’ ⇒ 按照指令的 funct 字段决定执行的操作
  - ‘ALUi’ ⇒ 按照指令的 opcode 字段决定执行的操作
  - ‘LW’ ⇒ 加法
  - ‘SW’ ⇒ 加法
  - ‘Bxx’ ⇒ 由bcond决定操作
  - 其它 ⇒ 不用考虑

- 一些 ALU 操作的例子
  - ADD, SUB, AND, OR, XOR, NOR, 等等.
  - bcond on equal, not equal, LE zero, GT zero, 等等.

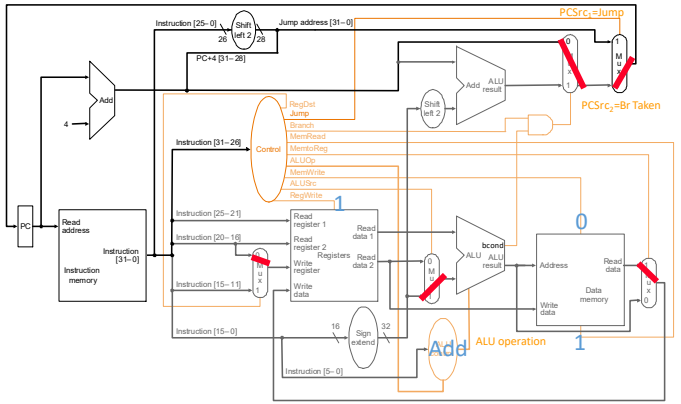
R-Type ALU指令的控制信号



I-Type ALU指令的控制信号



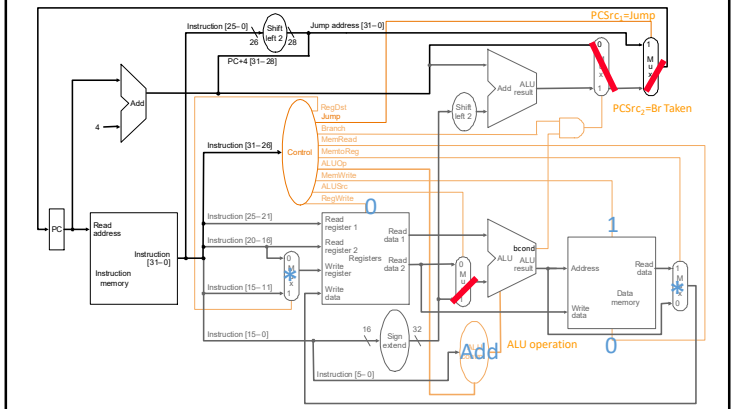
LW指令的控制信号



57

57

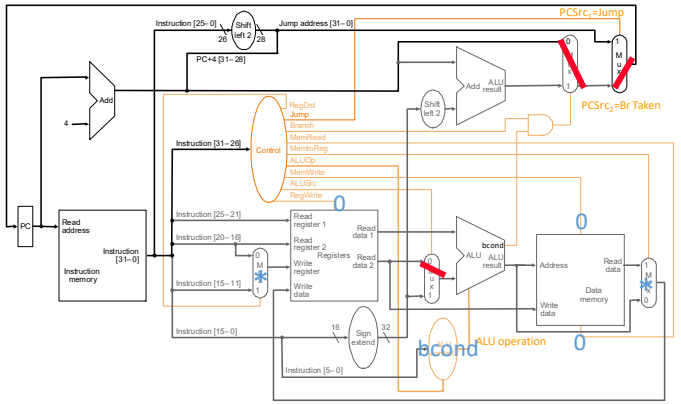
SW指令的控制信号



58

58

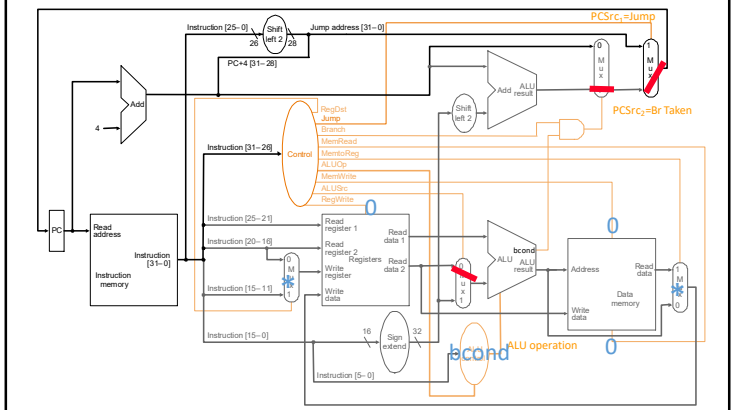
分支未发生的控制信号



59

59

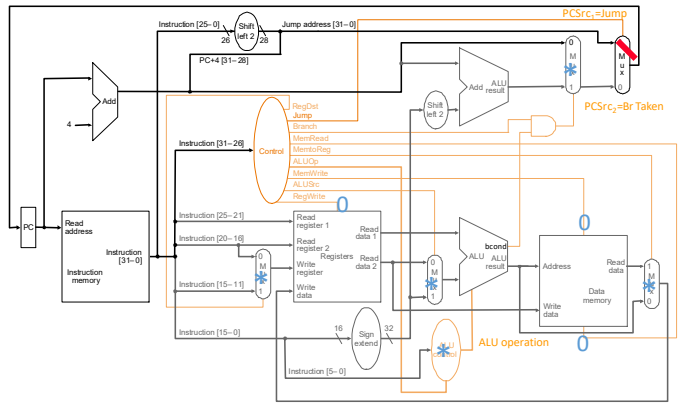
分支发生的控制信号



60

60

Jump



**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

61

61

椭圆形的“Control”圈圈里是什么?

- 组合逻辑→ 硬连线控制
  - 思路：基于指令用组合逻辑生成控制信号
- 时序逻辑→ 时序/微程序控制
  - 控制存储
  - 思路：用一个存储结构保存指令的控制信号

62

62

单周期微体系结构性能评价

63

63

单周期微体系结构

- 它是一个好的设计吗?
- 它什么时候会是一个好的设计?
- 什么时候不好?
- 我们如何才能设计一个更好的微体系结构?

64

64

单周期微体系结构：分析

- 每条指令执行占用1个时钟周期
  - CPI (Cycles per instruction) = 1
- 每条指令执行的时间受限于执行最慢的那条指令
  - 即使很多指令不需要执行那么长时间
- 微体系结构中的时钟周期长度由完成最慢的指令所需时间决定
  - 处理最慢指令的时间决定了关键路径的设计

65

65

最慢的指令流程是什么？

- 指令处理周期的全部6个阶段在*一个机器时钟周期*内完成

取指
译码
计算地址
取操作数
执行
存结果

1. 取指令（IF）
2. 指令译码和取寄存器操作数（ID/RF）
3. 执行/计算内存地址（EX/AG）
4. 取内存操作数(MEM)
5. 存储/写回结果（WB）

- 上面这些阶段对所有的指令来说都会花费同样的时间（时延）吗?

66

66

单周期数据通路分析

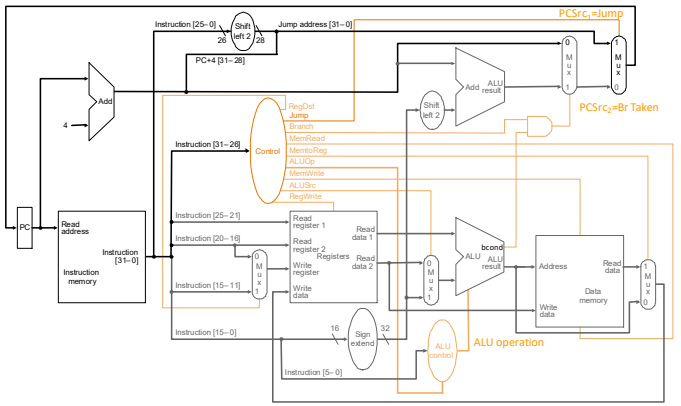
- 假设以下的部件时延
  - 内存单元(读或写)：200 ps
  - ALU和加法器：100 ps
  - 寄存器堆(读或写)：50 ps
  - 其它组合逻辑：0 ps

阶段	IF	ID	EX	MEM	WB	时延（关键路径）
来源	mem	RF	ALU	mem	RF	
R类型	200	50	100		50	400
I类型	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

67

67

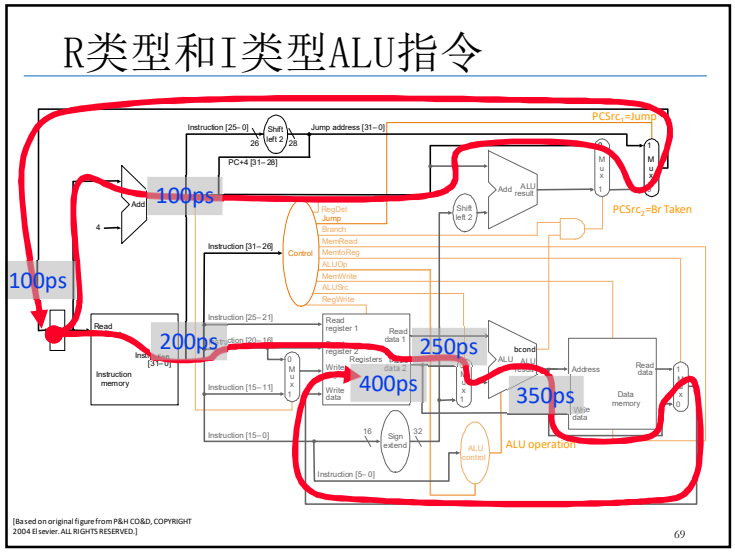
找到关键路径



68

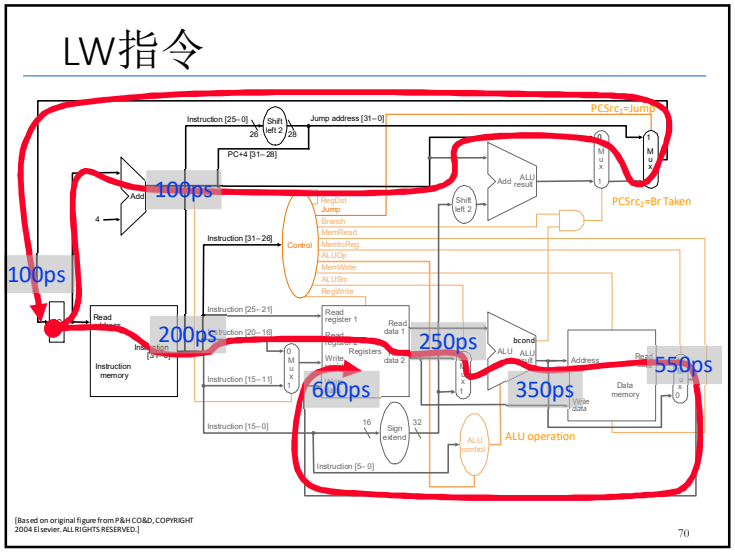
68

R类型和I类型ALU指令



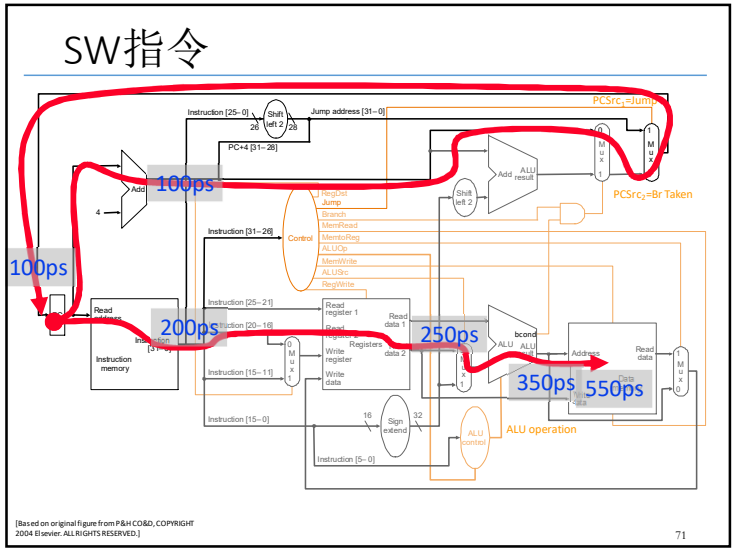
69

LW指令



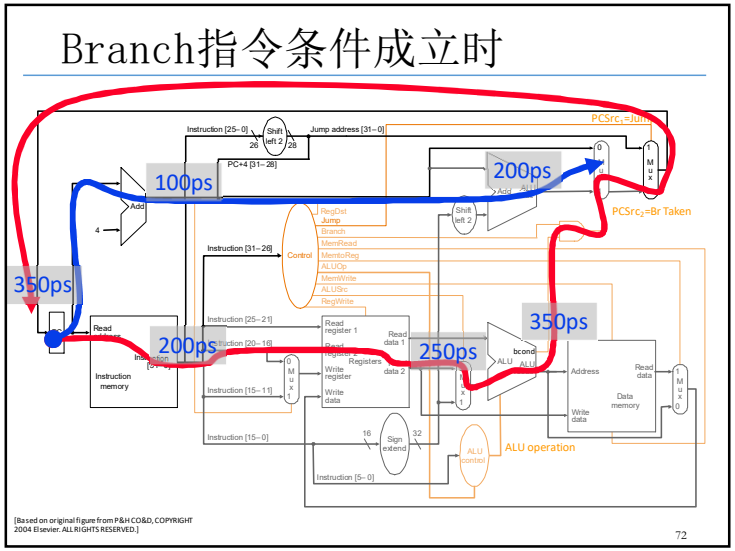
70

SW指令



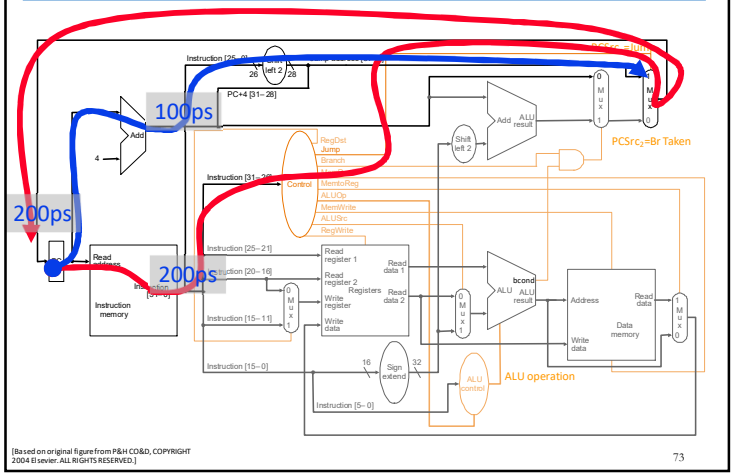
71

Branch指令条件成立时



72

Jump指令



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

73

73

控制逻辑?

- 控制逻辑如何影响关键路径?
  - 控制逻辑能否出现在关键路径上?
  - 控制存储的访问有时可能会花费很长时间

74

74

最慢的指令流程是什么?

- 存储器不是理想的
- 如果有时候访存要花费100ms怎么办?
- 让简单的寄存器加或者无条件转跳花上和访存操作一样的100ms+的时间有意义吗?
- 另外，如果处理一条指令需要不止一次访存该怎么办?
  - 什么指令需要?
  - 是否提供了多个内存端口?

75

75

单周期微架构:复杂性

- 人为因素
  - 所有指令都和最慢的指令一样慢
- 低效
  - 所有指令都和最慢的指令一样慢
  - 必须为所有指令提供最坏情况下的资源
  - 对于一条指令执行周期中在不同阶段会访问同一个资源的情况，必须为该资源提供“副本”
- 不一定是实现ISA的最简单方法
  - REP MOVS, INDEX, POLY等指令的单周期实现?
- 不容易优化/提升性能
  - 对通常情况(普通指令)做优化不起作用
  - 任何时候都要优化最坏的情况

76

76

微体系结构设计原则

- 关键路径设计
  - 找到时延最大的组合逻辑，尽可能的减小它的时延
- 基本（典型）设计
  - 在重要的地方花时间和资源
    - 提升机器设计目标要求的应有能力
  - 通常情况 vs. 特殊情况
- 平衡设计
  - 平衡流过硬件部件的指令/数据流
  - 平衡完成工作所需要的硬件
- *单周期微体系结构是如何遵循这些原则的？*

77

77

多周期微体系结构

78

78

多周期微体系结构

- 目标：使每一条指令的执行只（大致）花费它该花费的时间
- 思路
  - 时钟周期的决定独立于指令处理时间
  - 每条指令需要花费多少时钟周期
    - 一条指令执行过程中会有多次状态转换
    - 每条指令的状态变换是不同的

79

79

回顾：“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）

80

80

多周期微体系结构

AS = 指令执行之前程序员可见的体系结构状态

第1步：在一个时钟周期内处理一部分指令

第2步：在下一个时钟周期内处理一部分指令

…

AS' =指令执行之后程序员可见的体系结构状态

81

81

多周期设计的好处

- 关键路径设计
  - 可以独立地针对每条指令的最糟糕情况来优化关键路径
- 基本(典型)设计
  - 可以通过优化执行“重要”指令（占用大量执行时间）所需的状态数来达到需要的效果
- 平衡设计
  - 不需要提供比实际需求更多的资源或能力
    - 一条指令需要多次使用资源“X”并不意味着需要多个“X”
    - 使硬件更高效：一条指令可以多次***重用***硬件部件

82

82

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

有两个独立的自由度可以优化

83

83

CPI vs. 主频

- CPI vs. 时钟周期长度
- 互相矛盾
  - 对一条指令来说，减少一个就会增加另一个
  - 为什么?
- 多条指令并发处理可以使平均CPI被平摊/减小
  - 同一个时钟周期被用来处理多条指令
  - 例如：流水线，超标量等

84

84

多周期微体系结构
近距离观察

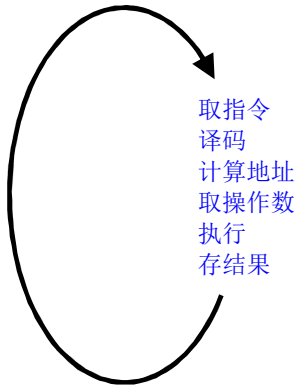
85

如何实现多周期?

- Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.
- 微码/微程序控制的概念
- 实现
  - 可以按照描述状态之间序列的有限状态机来实现"指令处理"的步骤，最终状态机回到"取指令"状态
  - 状态由控制信号推定
  - 下一个状态的控制信号由当前状态决定

86

指令执行周期



87

基本的多周期微体系结构

- 指令执行周期被划分为多个"状态"
  - 指令执行周期的每个阶段可以拥有多个状态
- 多周期微体系结构通过状态到状态的序列处理指令
  - 某个状态下机器的行为由该状态下的控制信号决定
- 整个处理器的行为可以被定义成一个有限状态机
- 在某个状态(时钟周期)中，控制信号控制
  - 数据通路如何处理数据
  - 如何为下一个时钟周期生成控制信号

88

微程序控制相关术语

- 与当前状态相关的控制信号
  - 微指令
- 从一个状态过渡到另一个状态的动作
  - 决定下一个状态以及下一个状态的微指令
  - 微序列（生成）
- 控制存储（器）为每一个可能的状态存储控制信号
  - 为整个有限状态机存储微指令
- 微序列（控制）器决定下一个时钟周期（下一个状态）将会用到的控制信号集合

89

89

在一个时钟周期里发生了什么?

- 对当前状态控制的控制信号(微指令)
  - 在数据通路中推进
  - 为下一个周期生成控制信号（微指令）
- 数据通路和微序列器并发操作
- 问题：为什么不在当前周期生成当前周期需要的控制信号?
  - 会使时钟周期延长
  - 为什么?

90

90

简单的LC-3b控制和数据通路

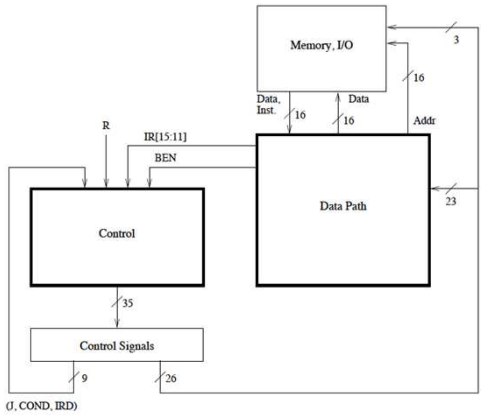


Figure C.1: Microarchitecture of the LC-3b, major components

91

91

什么决定了下一个周期的控制信号?

- 当前时钟周期发生了什么
  - 流入“Control”框的9根线
- 被执行的指令
  - 来自数据通路的IR[15:11]
- 不管分支条件是否满足，如果执行的指令是分支
  - 来自数据通路的BEN（1 bit）
- 不管访存操作是否在本周期完成，只要有访存操作在处理
  - 来自存储器的R（1 bit）

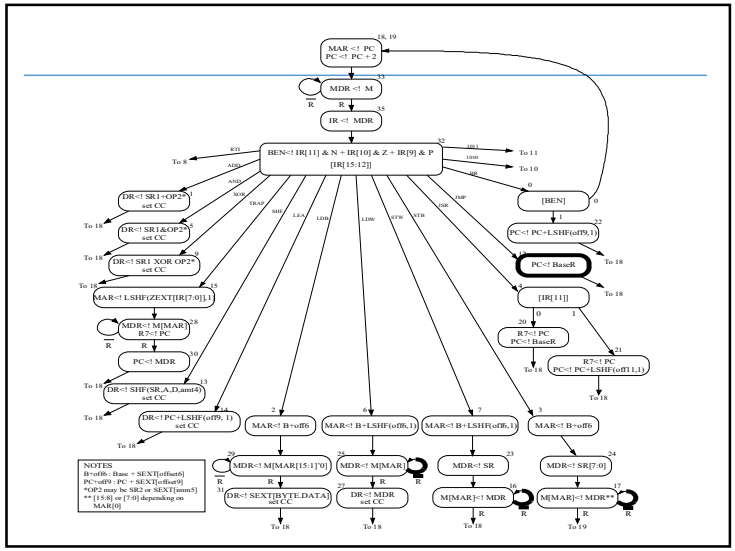
92

92

LC-3b多周期处理的状态机

- LC-3b微架构的行为由以下因素决定
  - 35个控制信号和7位由数据通路连入控制逻辑的信号
- 35个控制信号完整描述了控制结构的状态
- LC-3b的所有行为都可以描述为状态机——一个有向图
  - 结点(关联到每个状态)
  - 弧(代表一个状态到另一个状态的流)

93



95

LC-3b状态机

- Patt & Patel, 附录 C, 图 C.2
- 每个状态描述必须是唯一的
  - 通过*状态变量*
- LC-3b状态机有31个不同的状态
  - 由6个状态变量编码
- 例如
  - 状态18,19对应指令处理周期的开始
  - 取指阶段：状态18, 19 → 状态33 → 状态35
  - 译码阶段：状态32

94

关于LC-3b状态机的几个问题

- 最快的指令执行需要多少个时钟周期?
- 最慢的指令执行需要多少个时钟周期?
- 什么决定了时钟周期?
- 这是个摩尔型状态机还是米里型状态机?

96

简单的LC-3b控制和数据通路

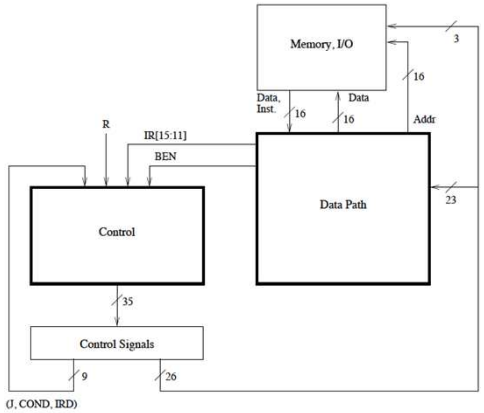


Figure C.1: Microarchitecture of the LC-3b, major components

97

97

LC-3b数据通路

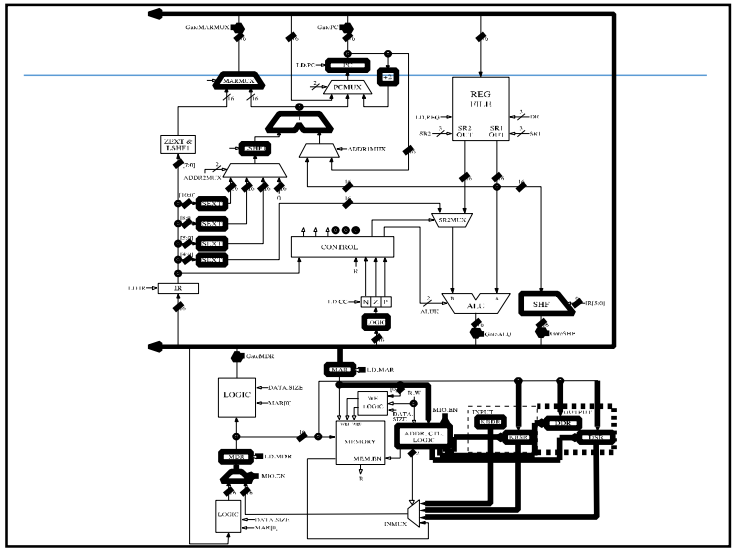
- Patt & Patel, 附录 C, 图 C.3

- 单总线数据通路设计
  - 任何时候只能有一个数值被放上总线(选通并使用总线)
  - 优点：硬件成本低，只有一条总线
  - 缺点：降低了并发性 - 如果指令需要因为两件不同的事使用总线两次，需要在不同的状态中完成

- 26个控制信号决定了一个时钟周期内数据通路上发生什么
  - Patt & Patel, 附录 C, 表 C.1

98

98



99

Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateSHF/1:	NO, YES	
PCMUX/2:	PC+2	;select pc+2
	BUS	;select value from bus
	ADDER	;select output of address adder
DRMUX/1:	11.9	;destination IR[11:9]
	R7	;destination R7
SR1MUX/1:	11.9	;source IR[11:9]
	8.6	;source IR[8:6]
ADDR1MUX/1:	PC, BaseR	
ADDR2MUX/2:	ZERO	;select the value zero
	offset6	;select SEXT[IR[5:0]]
	PCoffset9	;select SEXT[IR[8:0]]
	PCoffset11	;select SEXT[IR[10:0]]
MARMUX/1:	7.0	;select LSHF(ZEXT[IR[7:0]],1)
	ADDER	;select output of address adder
ALUK/2:	ADD, AND, XOR, PASSA	
MIO.EN/1:	NO, YES	
R.W/1:	RD, WR	
DATA.SIZE/1:	BYTE, WORD	
LSHF1/1:	NO, YES	

Table C.1: Data path control signals

100

关于LC-3b 数据通路的几个问题

- 在数据通路中是如何做到根据状态机实现取指令的?
- 选通和载入有什么不同?
- 这个设计是最节省硬件的吗?

101

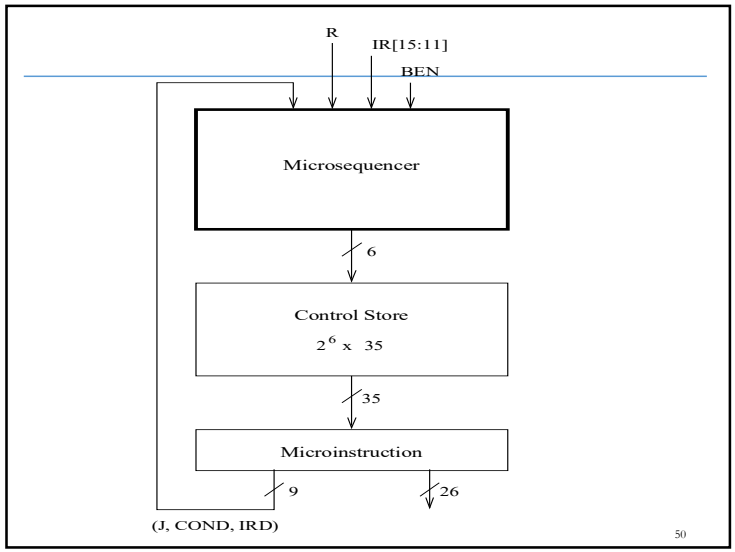
101

LC-3b 微程序设计控制结构

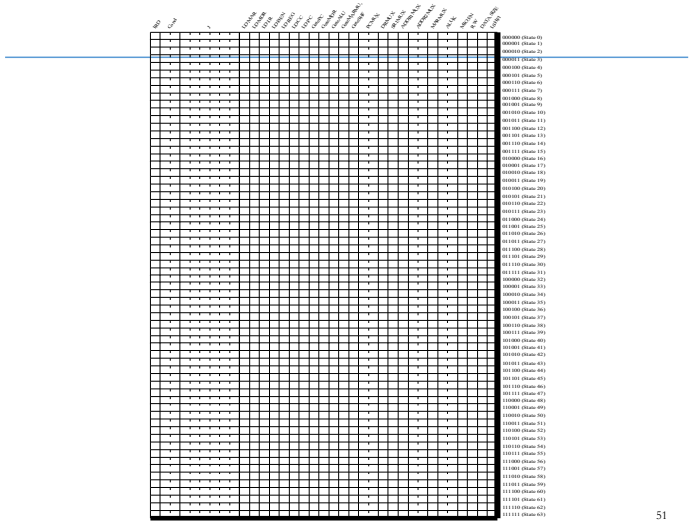
- Patt & Patel, 附录 C, 图 C.4
- 三个组件:
  - 微指令, 控制存储, 微序列（控制）器
- 微指令: 26个控制数据通路，9个决定下一个状态
- 每个微指令存储在控制存储（特殊的存储结构）的特定位置
- 特定位置: 对应微指令的状态地址
  - 每个状态对应一条微指令
- 微序列（控制）器决定下一条微指令的地址（下一个状态）

102

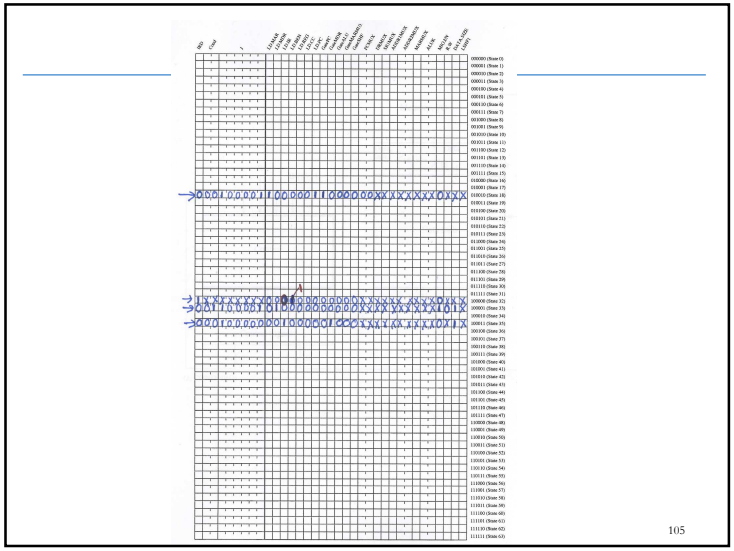
102



103

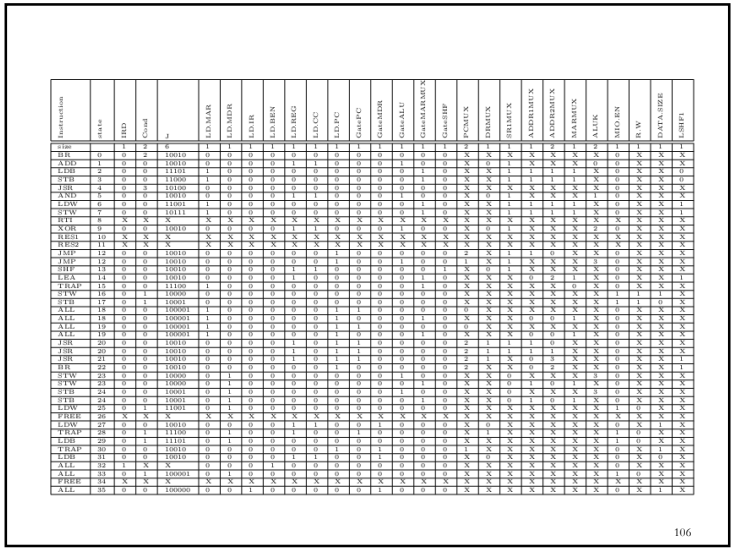


104



105

105



106

106

LC-3b 微序列（控制）器

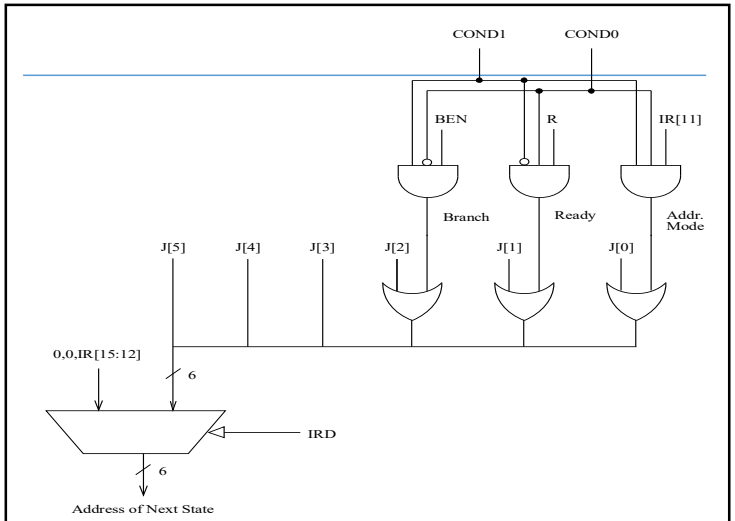
- Patt & Patel, 附录 C, 图 C.5
- 微序列器的目的是决定下一条微指令的地址(下一个状态)
- 下一个地址取决于 9 个控制信号

Signal Name	Signal Values	
J/6:		
COND/2:	$COND_0$	;Unconditional
	$COND_1$	;Memory Ready
	$COND_2$	;Branch
	$COND_3$	;Addressing Mode
IRD/1:	NO, YES	

Table C.2: Microsequencer control signals

107

107



108

108

关于微序列（控制）器的几个问题

- IRD信号什么时候生效?
- 如果一条非法指令被译码会发生什么?
- 条件(COND)位是用来做什么的?
- 延迟可变存储如何处理?
- 如何对状态编码?
  - 使状态变量数最少
  - 从16路分支开始
  - 然后根据COND位确定约束表和状态

109

109

关于控制存储的几个问题

■什么控制信号能够被存入控制存储?

■什么控制信号只能由硬连线逻辑生成?

❑ 什么信号必须在数据通路中处理才能得到?

110

110

延迟可变存储

- Ready信号(R)使得存储器读写能够正确的执行
  - 例如: 状态33向状态35转变由存储器准备好后生成的R信号控制
- 在单周期微体系结构中是如何做的?

111

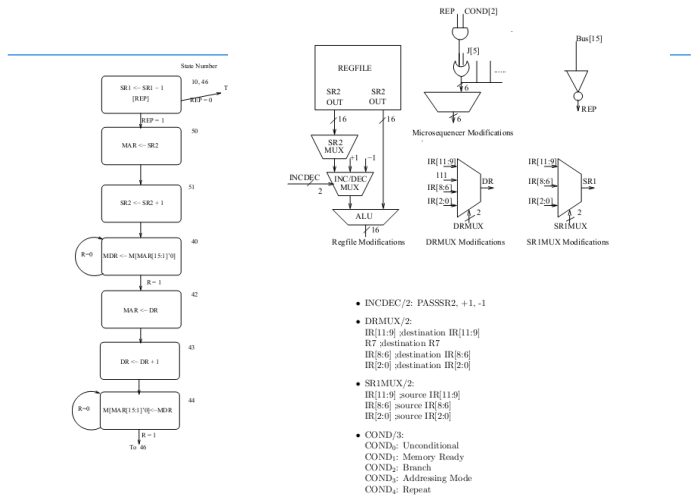
111

关于微序列器的高级问题

- 如果机器出现中断会发生什么?
- 如果指令产生异常会怎么样?
- 如何使用这种控制结构实现一条复杂指令?
  - 考虑 REP MOVS

112

112



113

抽象的力量

- 控制存储的微指令概念使得硬件设计者具有一种新的抽象：微程序设计
- 设计者可以将任何希望的操作翻译成微指令序列
- 设计者只需要提供
  - 实现目标操作所需的微指令序列
  - 具有正确驱动微指令序列能力的控制逻辑
  - 其它必须附加的数据通路控制信号(如果操作不能翻译成已有的控制信号)

114

114

其它：内存中的对齐矫正

- 访存对齐
- LC-3b 有字节 load 和 store 指令，可以不按照字边界移动数据
  - 对程序员/编译器很方便
- 硬件如何保证读写的正确性?
  - 状态 29 - LDB
  - 状态 24 和 17 - STB
  - 额外的逻辑处理未对齐的访问

115

115

其它：内存映射I/O

- 地址控制逻辑决定访存指令的地址是内存还是I/O设备
- 相应地驱动内存或I/O设备并且设置多路选择器
- 有些控制信号不能保存在控制存储中
  - 依靠地址

116

116

微程序控制的优点

- 通过控制数据通路（用序列器），可以用非常简单的数据通路实现强有力的计算
  - 高级ISA翻译成微码（微指令序列）
  - 微码使得用最简单的数据通路***仿真***ISA成为可能
  - 微指令可以被看作是用户不可见的ISA
- 使ISA很容易扩展
  - 可以通过改变微码支持新的指令
  - 可以通过简单微指令的序列来支持复杂的指令
- 如果可以把任意指令序列化，那么也能够把任意“程序”序列化成微程序序列
  - 在微码中需要一些新的状态（如：循环计数器）来序列化更复杂的程序

硬件升级

- 对微码升级/打补丁的能力（处理器发货之后）
  - 不用更换处理器就可以增加新的指令!
  - “修复”硬件实现的缺陷
- 例如
  - IBM 370 Model 145：微码存储在主存中，可以在重启之后升级
  - IBM System z：与 370/145类似
    - Heller and Farrell, “Millicode in an IBM zSeries processor,” IBM JR&D, May/Jul 2004.
  - B1700 微码可以在处理器运行时更新
    - 用户可微编程的机器!