计算机体系结构
第四讲: 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
2021-03-19

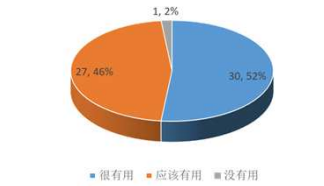
1

作业 0

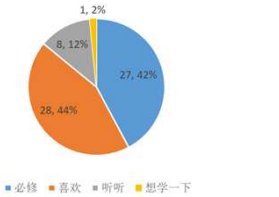
- 03-12日截止!
- 统计了58份提交的作业

2

对体系结构的认识



体系结构重要性

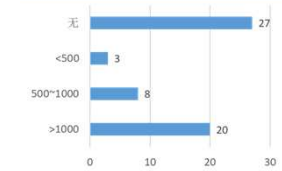


选课动机

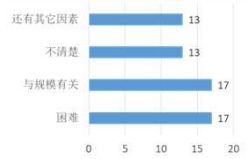
3

学习基础

知识基础：计组/HDL/CPU



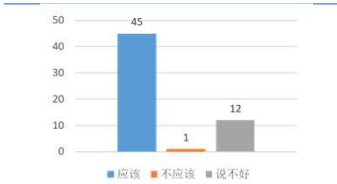
开发基础：HDL开发规模



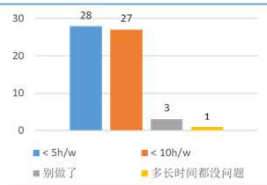
预测的CPU开发难度

4

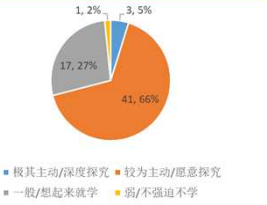
课程学习



实验内容：是否有CPU实验



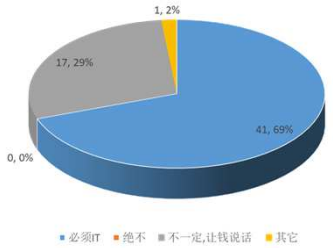
实验强度：每周投入时间



学习能力

职业规划

- 未来规划：各有不同☺，而且
  - 有人说想当老师了☺☺☺☺



提醒：接下来两周的作业

- 作业 1
  - 已发布，3月26日提交，课程网站
  - MIPS 、ISA 基本概念，基本的性能分析评价
- 作业 2
  - 3月26日发布…

如何对论文做评价

- 1：简要的总结（摘要）
  - 论文试图解决什么问题?
  - 论文的核心/关键思路（创新点）是什么?
  - 论文在当时的主要贡献是什么?
  - 你从中领悟到的最重要的东西是什么?
- 2：论文的优点（最重要的几点）
  - 论文对相关问题解决的好吗?
- 3：论文的弱点（最重要的几点）
  - 这一部分是应该仔细思考的，每一篇论文都有弱点。这不意味着这篇论文不好，它意味着还有改进的空间，在未来的工作中能够完成这种改进。
- 4：你能做得更好吗？列出你的想法/思路。
- 5：从论文中你学到了什么，喜欢哪一部分，不喜欢哪一部分，为什么？

- 短小精干（可以半页纸甚至更短）
- 严肃认真
- 始终思考有没有更好的解决问题的方法

提醒：实验 1

- 已发布，4月16日提交
  - 用Logisim设计1个7指令单周期MIPS CPU
- 学习 MIPS ISA
  - 课程网站上传了部分相关材料

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

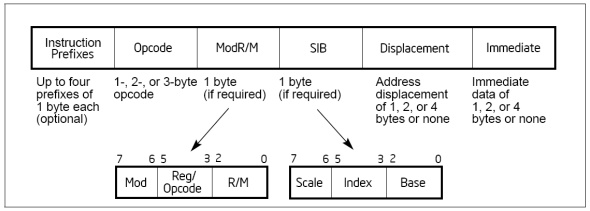
回顾：ISA Tradeoffs

- 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
- 寻址方式

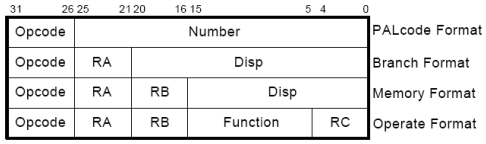
  Wulf, "Compilers and Computer Architecture," IEEE Computer 1981
- 执行指令之前把复杂指令翻译成"简单"指令有什么好处?
  - 硬件 (Intel, AMD)?
  - 软件 (Transmeta)?
- 哪一种 ISA 更容易扩展：固定长度 还是 可变长度?
- 如何拥有可变长度、统一译码的 ISA?

回顾：x86 vs. Alpha 指令格式

- x86:



- Alpha:



回顾：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)
- …

13

13

回顾微体系结构：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'

14

14

回顾微体系结构：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

↓

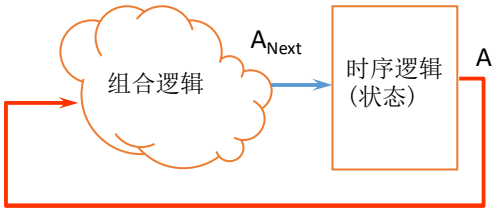
A' = 时钟周期结束时的体系结构状态(程序员可见)

15

15

回顾微体系结构：最基本的指令处理引擎

- 单周期机器

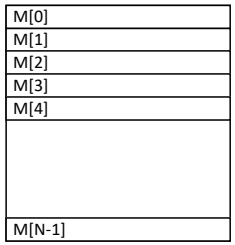


- 时钟周期长度由谁来决定?
- 组合逻辑中的关键路径由谁来决定?

16

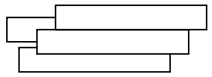
16

回顾微体系结构：程序员可见的(体系结构)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

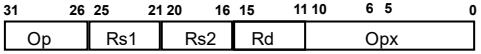
现在，我们有两个独立的自由度可以优化

单周期微体系结构——
近距离观察

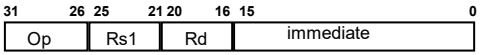
25

例子： MIPS

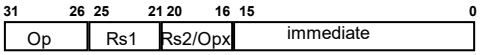
Register-Register



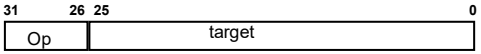
Register-Immediate



Branch



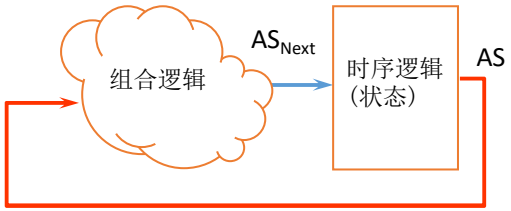
Jump / Call



Op Rd, Rs1, Rs2

26

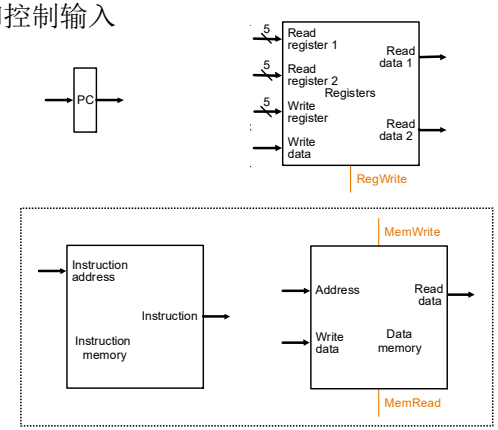
- 单周期的机器



27

从状态单元开始

- 数据和控制输入



28

现在，我们假设

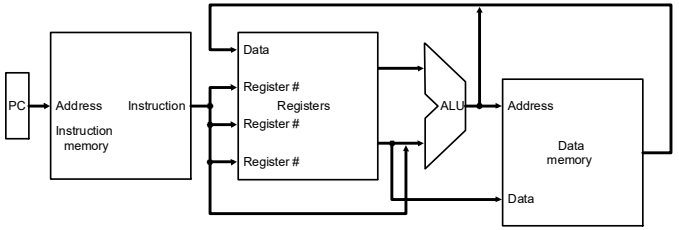
- “理想化”内存和寄存器堆
- 组合读
  - 读数据端口的输出是寄存器堆中内容和相应的读端口地址的组合函数
- 同步写
  - 写使能信号有效时，被选定的寄存器在时钟信号上升沿更新
    - 不会影响时钟沿之间的寄存器读输出
    - 会影响时钟沿上的寄存器读输出 （无所谓?）
- 单周期，同步存储
  - 内存读写需要确保数据准备好

29

29

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)

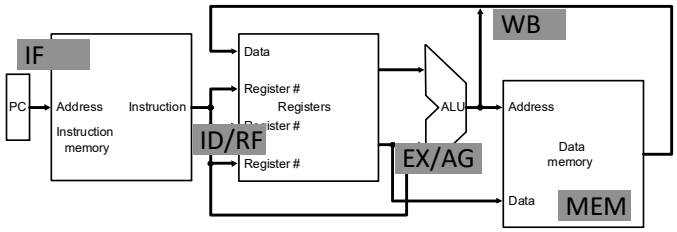


30

30

指令处理

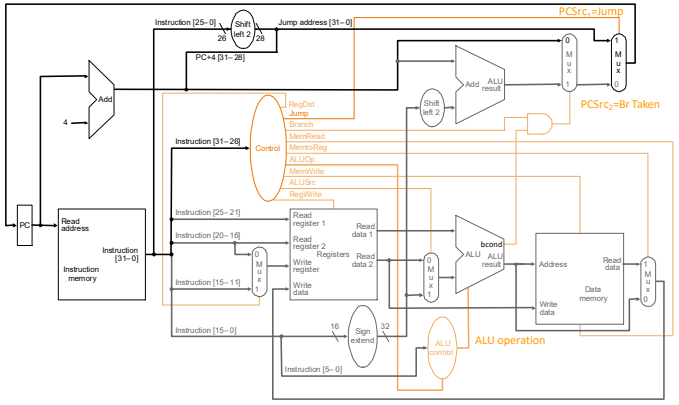
- 5个一般步骤 (Patterson & Hennessy’s Book)
  - 取指令 (IF)
  - 指令译码和取寄存器操作数 (ID/RF)
  - 执行/计算内存地址 (EX/AG)
  - 取内存操作数 (MEM)
  - 存储/写回结果 (WB)



31

31

完整的数据通路



忽略JAL, JR, JALR

32

32

单周期数据通路——
算术和逻辑指令

33

33

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	

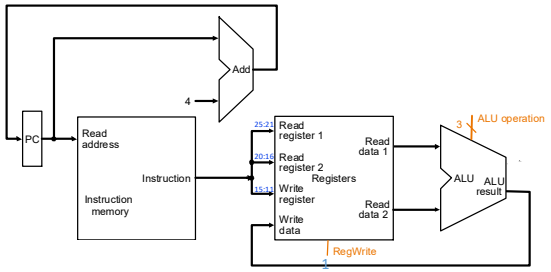
- 语义

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

34

34

R类型ALU指令数据通路



IF	ID	EX	MEM	WB

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

状态更新的组合逻辑

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

35

35

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

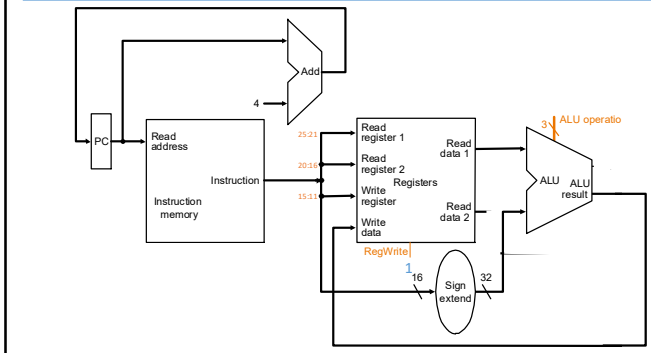
  GPR[rt] ← GPR[rs] + sign-extend (immediate)

  PC ← PC + 4

36

36

R类型和I类型ALU指令数据通路



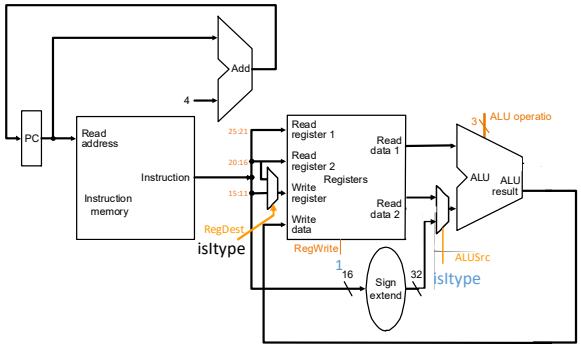
IF | ID | EX | MEM | WB

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

状态更新的组合逻辑

37

R类型和I类型ALU指令数据通路



IF | ID | EX | MEM | WB

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

状态更新的组合逻辑

38

单周期数据通路——
数据移动类指令

39

39

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
6-bit	5-bit	5-bit	16-bit

I-type

- 语义

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

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

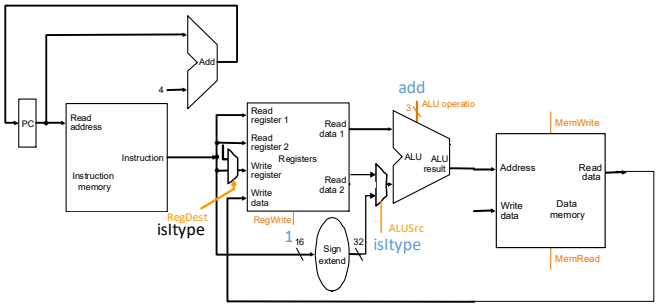
  GPR[rt] ← MEM[EA]

  PC ← PC + 4

40

40

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

41

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	

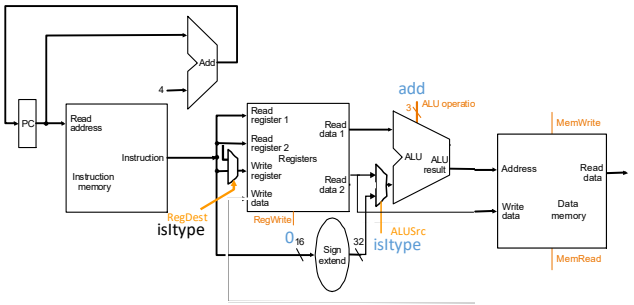
- 语义

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

42

42

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

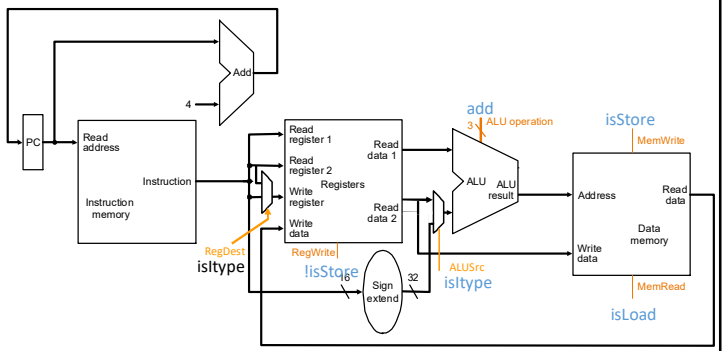
IF	ID	EX	MEM	WB

状态更新的组合逻辑

50

43

Load-Store 数据通路

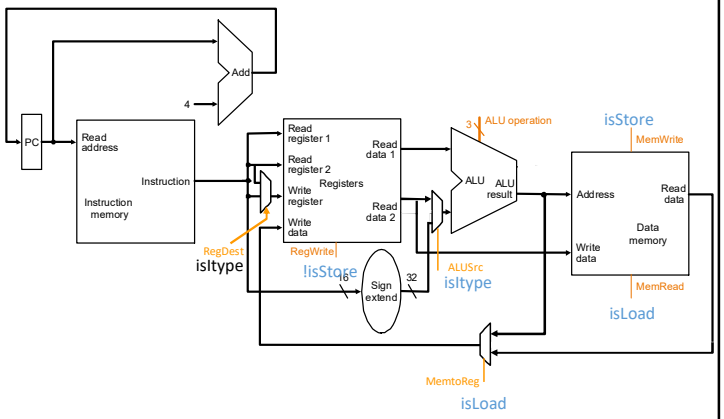


**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

44

44

不含控制流指令的数据通路



45

单周期数据通路——*控制流指令*

46

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

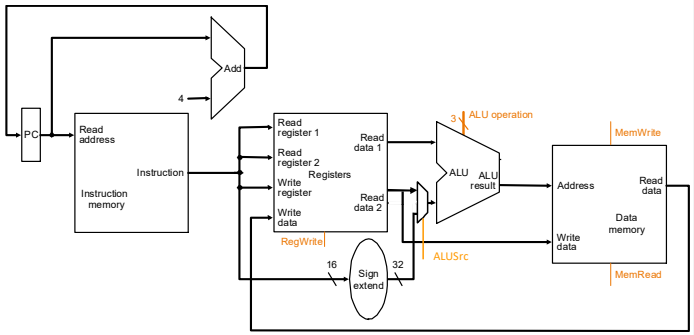
  if MEM[PC]==J immediate$_{26}$

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

  PC ← target

47

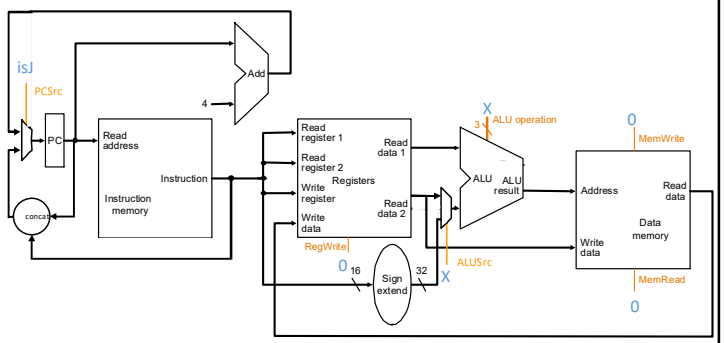
无条件转跳数据通路



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

48

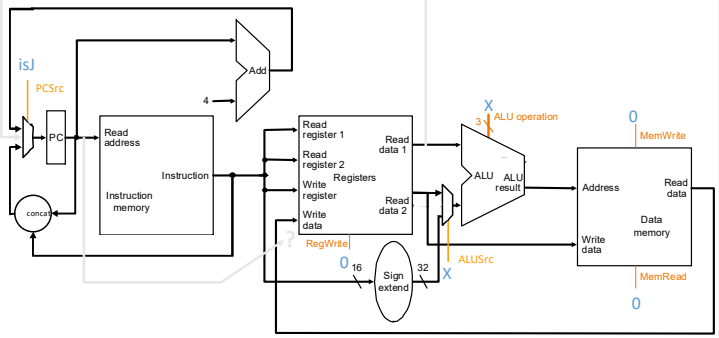
无条件转跳数据通路



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

49

无条件转跳数据通路



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

JR, JAL, JALR?

50

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
6-bit	5-bit	5-bit	16-bit

I-type

- 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

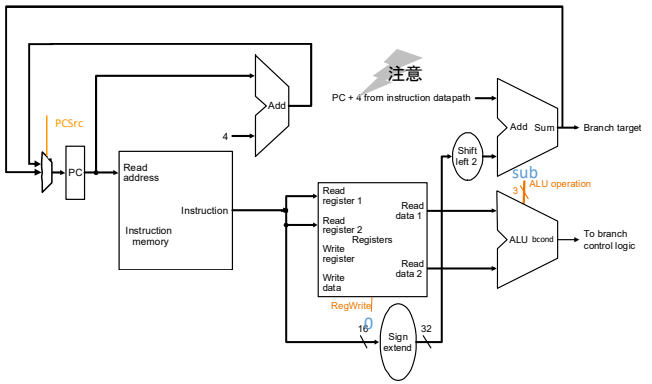
  if GPR[rs]==GPR[rt] then PC ← target

  else PC ← PC + 4

51

51

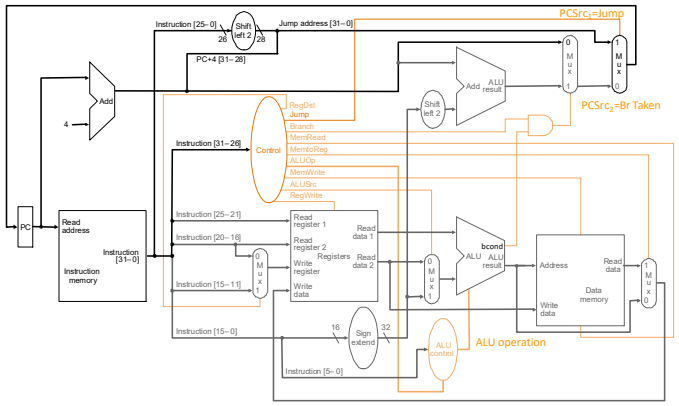
条件分支数据通路



52

52

数据通路整合



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

忽略 JAL, JR, JALR

53

53

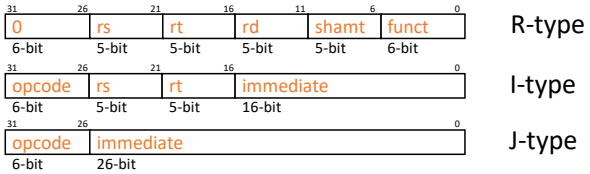
单周期控制逻辑

54

54

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

55

55

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))

56

56

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”

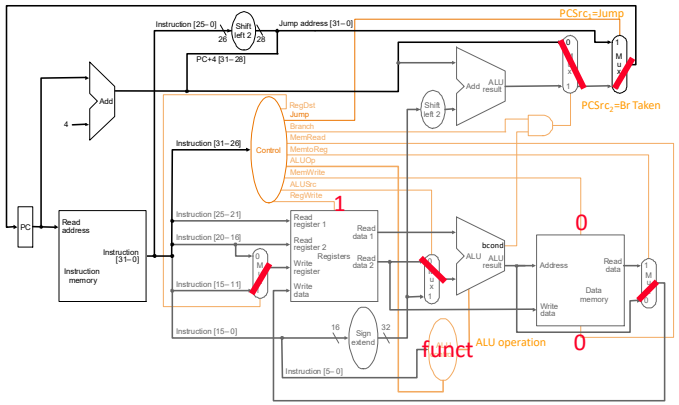
57

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.

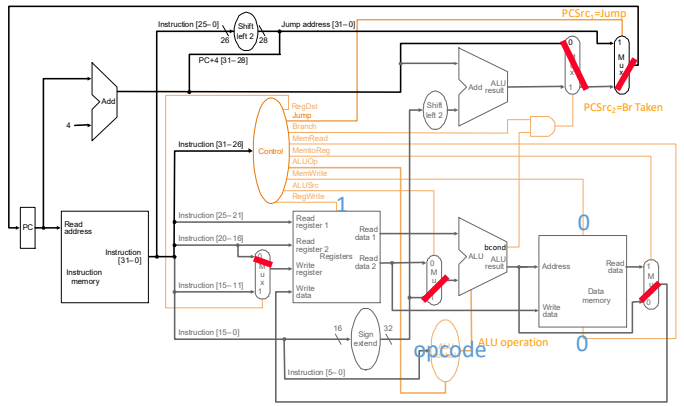
58

R-Type ALU指令的控制信号



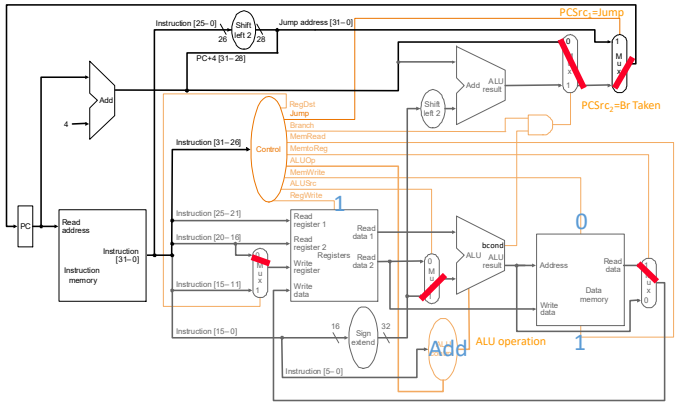
59

I-Type ALU指令的控制信号



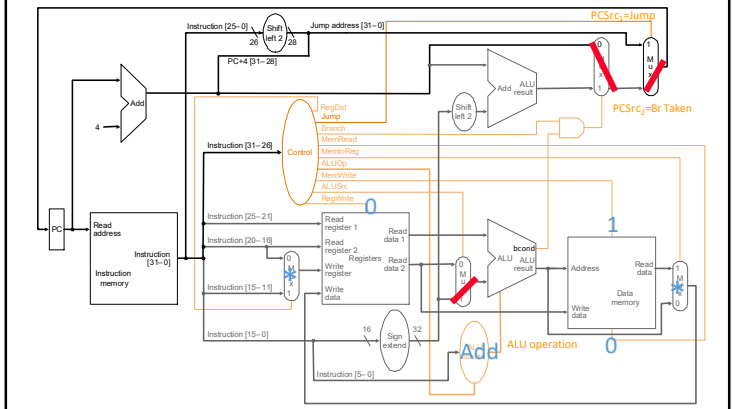
60

LW指令的控制信号



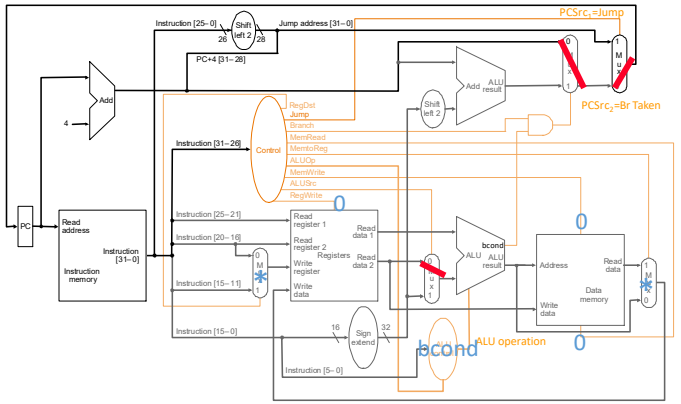
61

SW指令的控制信号



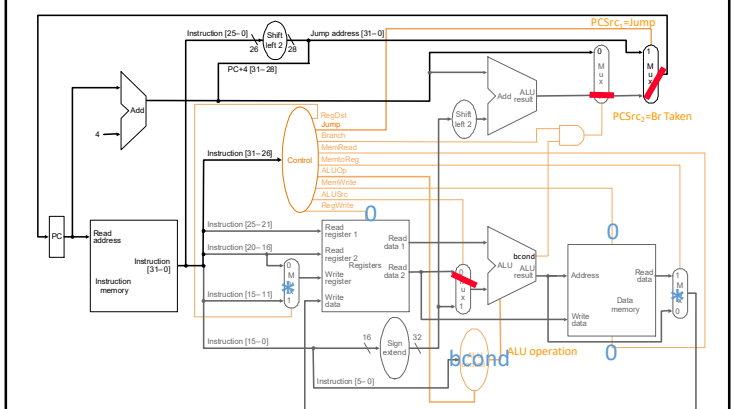
62

分支未发生的控制信号



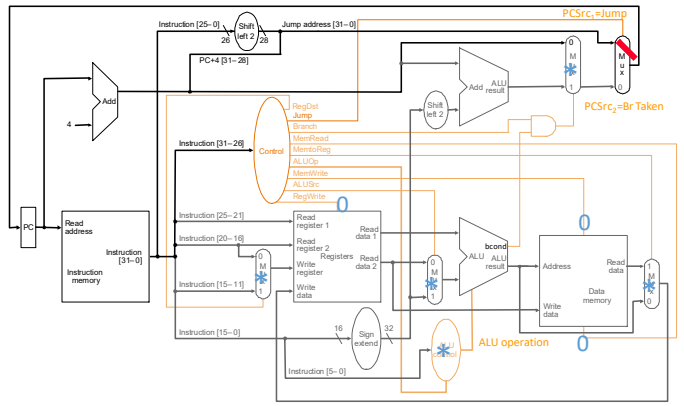
63

分支发生的控制信号



64

Jump



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

65

65

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

66

66

单周期微体系结构性能评价

67

67

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?

68

68

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

69

69

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?

70

70

单周期数据通路分析

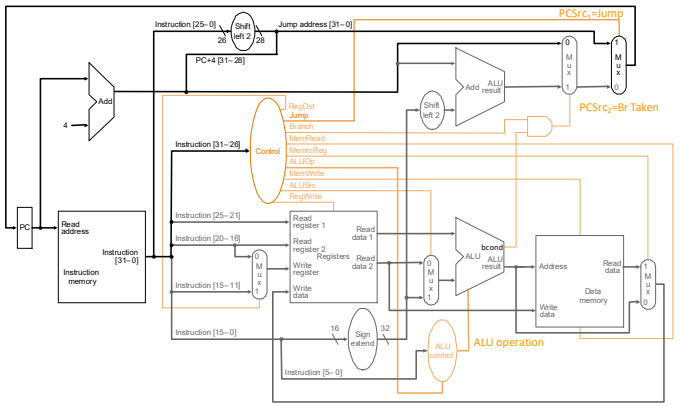
- 假设以下的部件时延
  - 内存单元(读或写)：200 ps
  - ALU和加法器：100 ps
  - 寄存器堆(读或写)：50 ps
  - 其它组合逻辑：0 ps

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

71

71

找到关键路径

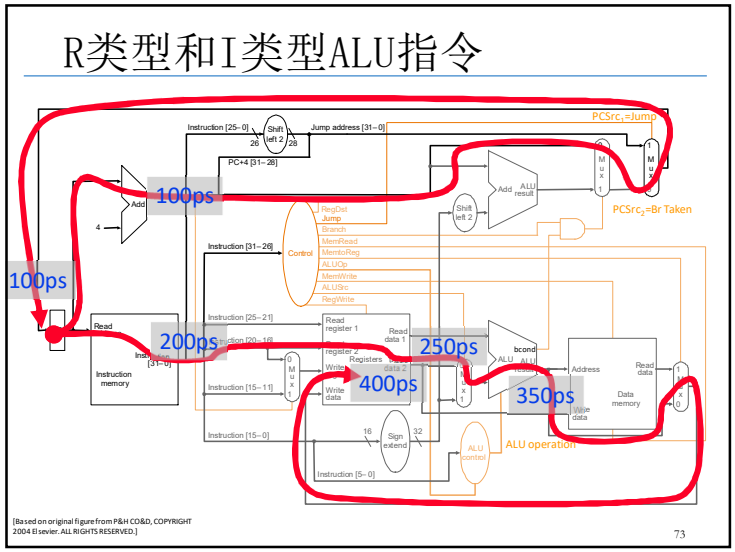


[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

72

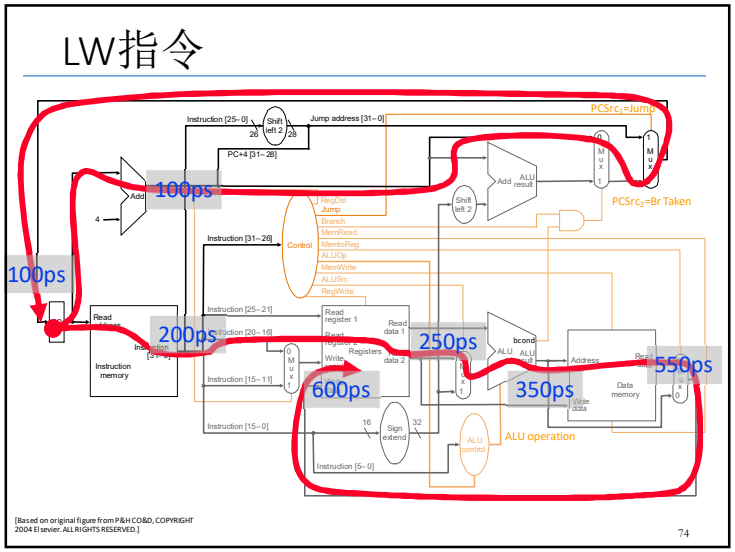
72

R类型和I类型ALU指令



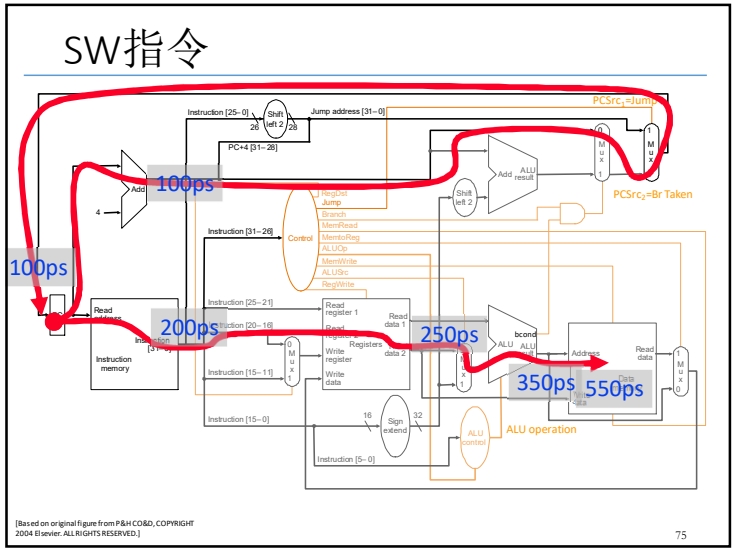
73

LW指令



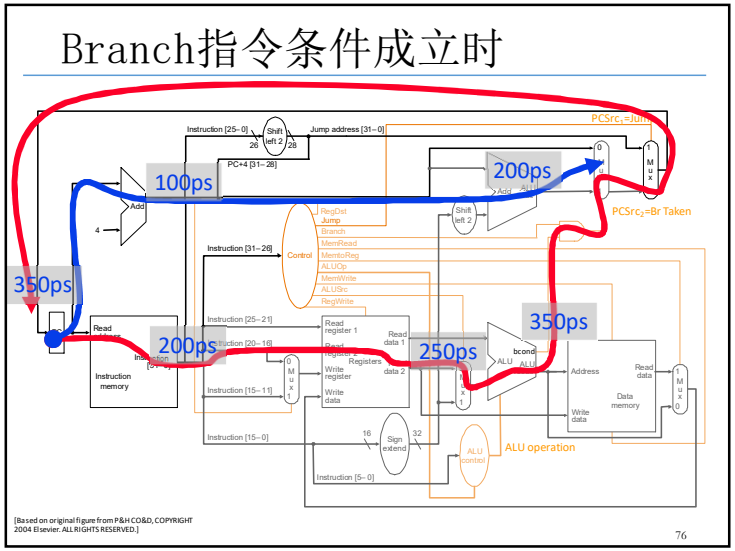
74

SW指令



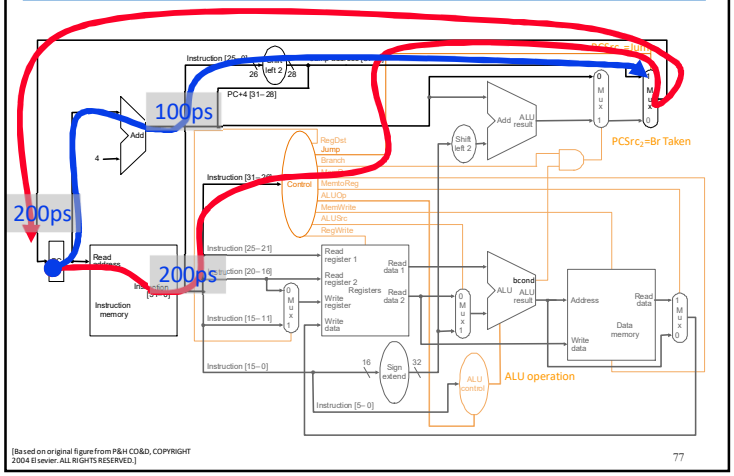
75

Branch指令条件成立时



76

Jump指令



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

77

77

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

78

78

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?

79

79