# 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
2020-03-27

1

## 提醒：作业

- 作业 1
  - 今天截止
  - 课程网站提交
- 作业 2
  - 今晚发布，4月10日上课前
  - 单周期与多周期微体系结构
- 作业 3
  - 4月10日发布…
  - 流水线

系统性能



2

## 提醒：实验 1

- 4月10日截止
  - 用Logisim设计1个7指令单周期MIPS CPU
- 学习MIPS ISA



3

## 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）
  - 附录 D
  - 第四章 (4.5-4.8, , 4.9-4.11)
- 选读
  - Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.
  - Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - 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



4

# 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*

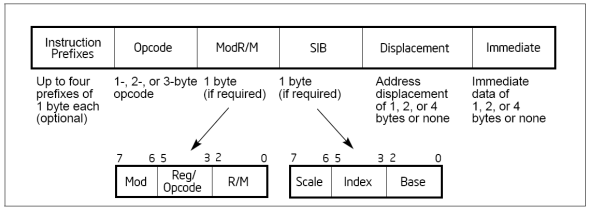
- 执行指令之前把复杂指令翻译成“简单”指令有什么好处?
  - 硬件 (Intel, AMD)?
  - 软件 (Transmeta)?
- 哪一种 ISA 更容易扩展：固定长度 还是 可变长度?
- 如何拥有可变长度、统一译码的 ISA?



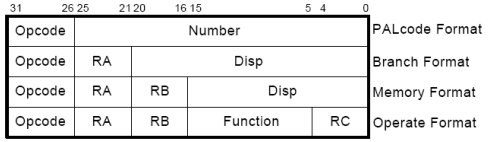


# 回顾：x86 vs. Alpha 指令格式

- 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)
- …





# 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'

## 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

↓

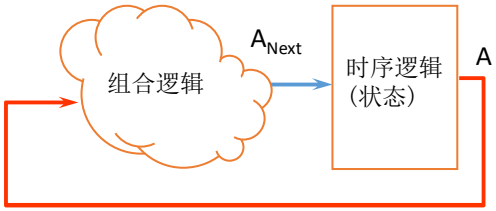
A' = 时钟周期结束时的体系结构状态(程序员可见)

9

9

## 回顾微体系结构：最基本的指令处理引擎

- 单周期机器

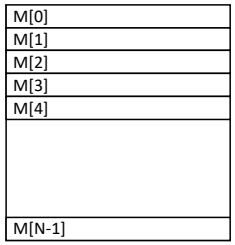



- 时钟周期长度由谁来决定?
- 组合逻辑中的关键路径由谁来决定?

10

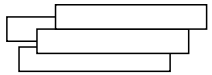
10

## 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

11

11

## 回顾微体系结构：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

- 指令处理“周期” vs. 机器时钟周期

12

12

## 回顾微体系结构：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
- 有很多方法可以用来设计数据通路和控制逻辑
- 控制信号和结构依赖于数据通路的设计



13

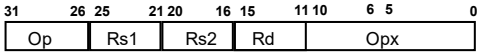
## 单周期微体系结构——*近距离观察*



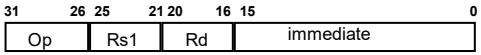
14

## 例子： MIPS

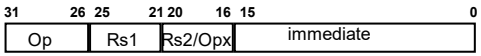
**Register-Register**



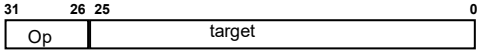

**Register-Immediate**




**Branch**




**Jump / Call**

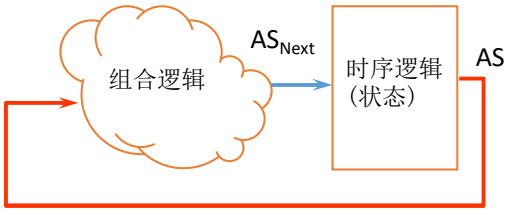



Op Rd, Rs1, Rs2



15

- 单周期的机器

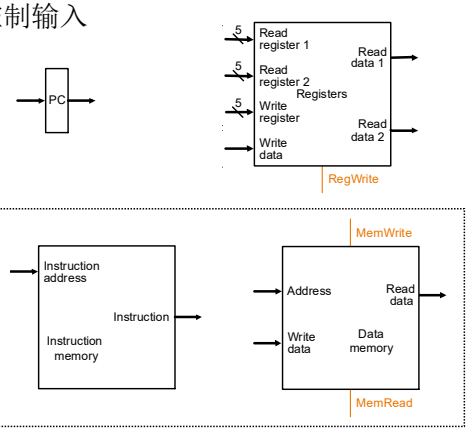





16

# 从状态单元开始

- 数据和控制输入






17

17

# 现在，我们假设

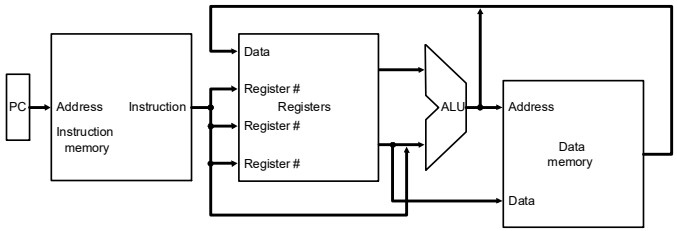
- “理想化”内存和寄存器堆
- 组合读
  - 读数据端口的输出是寄存器堆中内容和相应的读端口地址的组合函数
- 同步写
  - 写使能信号有效时，被选定的寄存器在时钟信号上升沿更新
    - 不会影响时钟沿之间的寄存器读输出
    - 会影响时钟沿上的寄存器读输出 （无所谓?）
- 单周期，同步存储
  - 内存读写需要确保数据准备好

18

18

# 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)



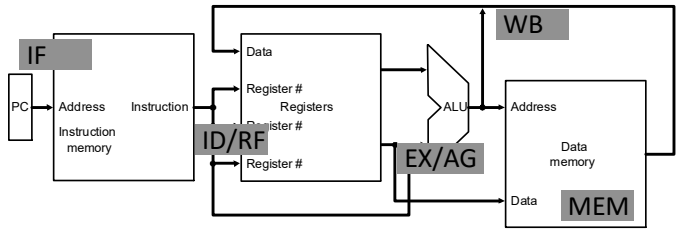



19

19

# 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)

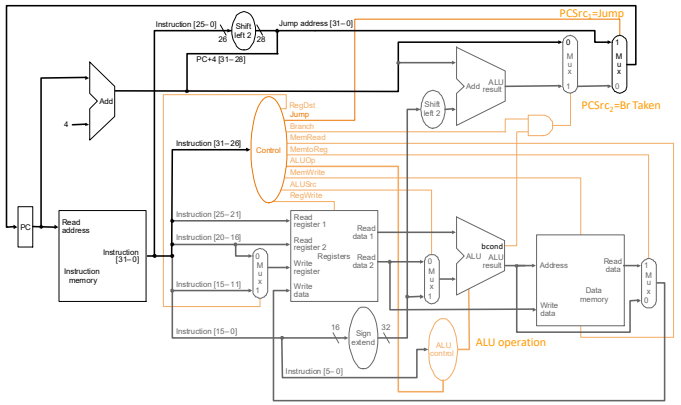





20

20

## 完整的数据通路




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

忽略JAL, JR, JALR



21

## 单周期数据通路——*算术和逻辑指令*



22

## 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 | |

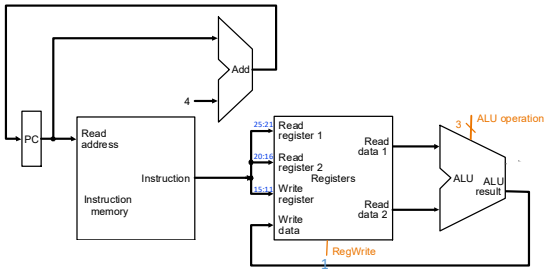
- 语义

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



23

## 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.]



24

# 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

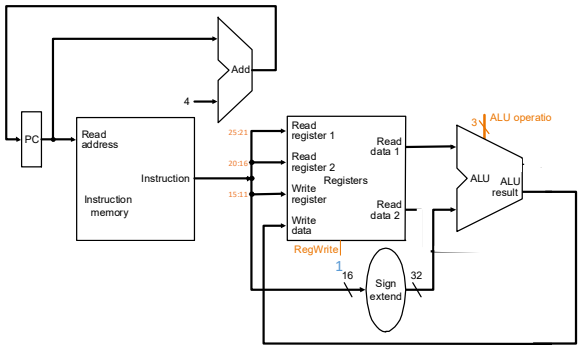
  GPR[rt] ← GPR[rs] + sign-extend (immediate)

  PC ← PC + 4



25

# R类型和I类型ALU指令数据通路




IF | ID | EX | MEM | WB

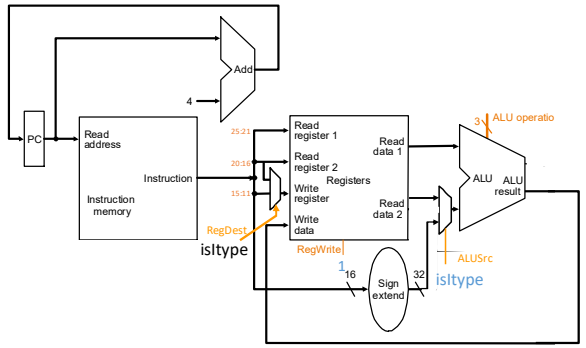
if MEM[PC] == ADDI rt rs immediate
GPR[rt] ← GPR[rs] + sign-extend (immediate)
PC ← PC + 4

状态更新的组合逻辑



26

# 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



27

# 单周期数据通路——

*数据移动类指令*



28

## 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]

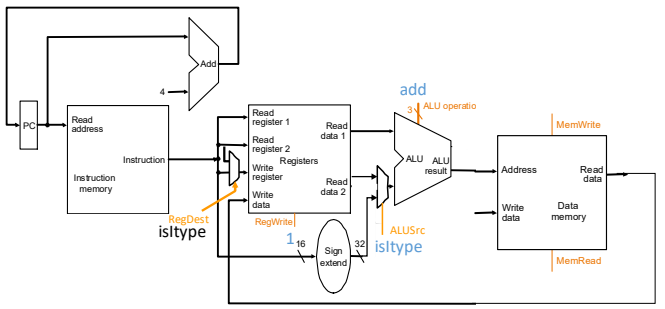
  GPR[rt] ← MEM[ EA ]

  PC ← PC + 4

29

29

## 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

30

## 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]

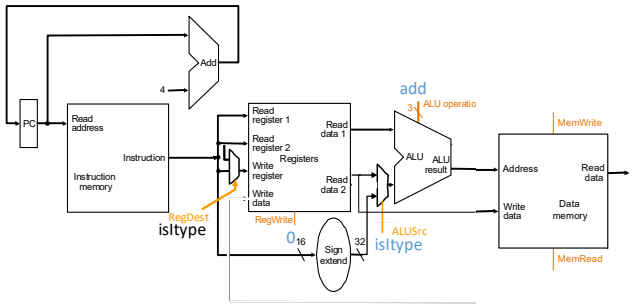
  MEM[ EA ] ← GPR[rt]

  PC ← PC + 4

31

31

## SW 数据通路




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

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

MEM[ EA ] ← GPR[rt]

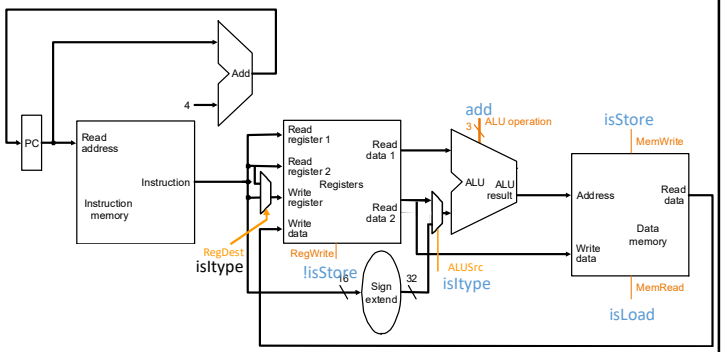
PC ← PC + 4

| IF | ID | EX | MEM | WB |
|---|---|---|---|---|

状态更新的组合逻辑

32

# Load-Store 数据通路

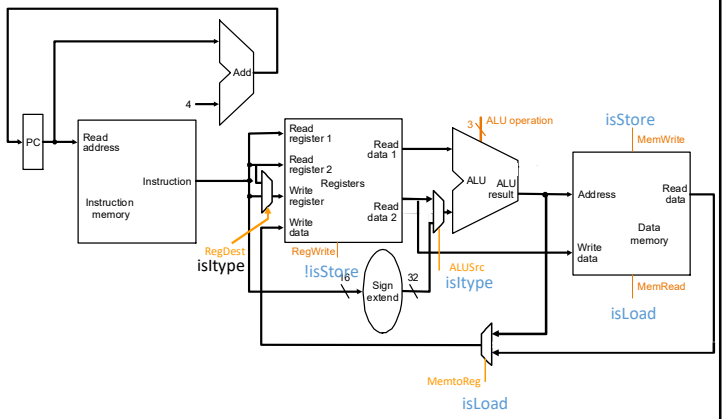







33

# 不含控制流指令的数据通路








34

# 单周期数据通路——*控制流指令*



35

# 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$_{26}$

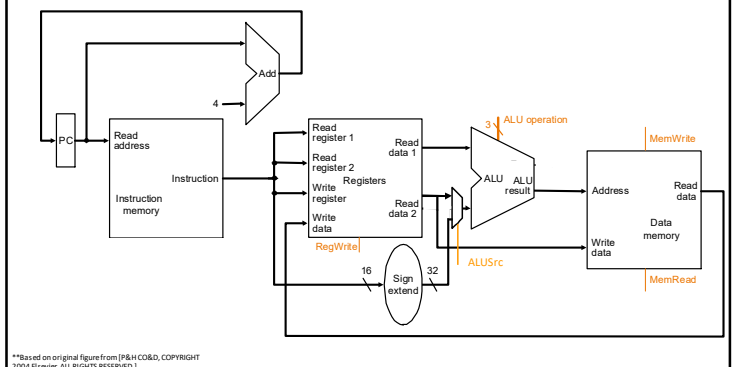
  target = { PC[31:28], immediate$_{26}$, 2'b00 }

  PC ← target



36

# 无条件转跳数据通路

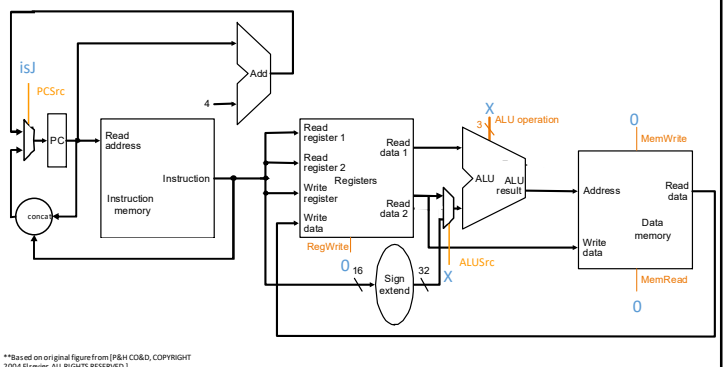





if MEM[PC]==J immediate26
PC = { PC[31:28], immediate26, 2'b00 }

37

# 无条件转跳数据通路

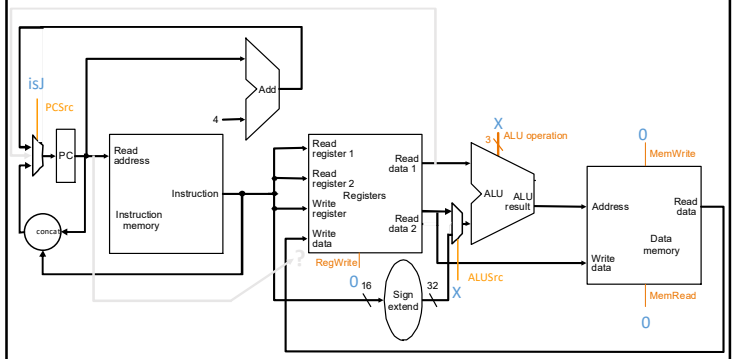





if MEM[PC]==J immediate26
PC = { PC[31:28], immediate26, 2'b00 }

38

# 无条件转跳数据通路






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

JR, JAL, JALR?

39

# 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 | |

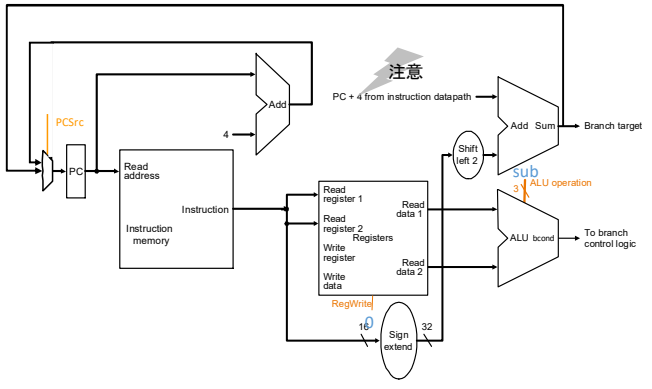
- 语义（假设没有分支延迟槽）

  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

40

40

## 条件分支数据通路

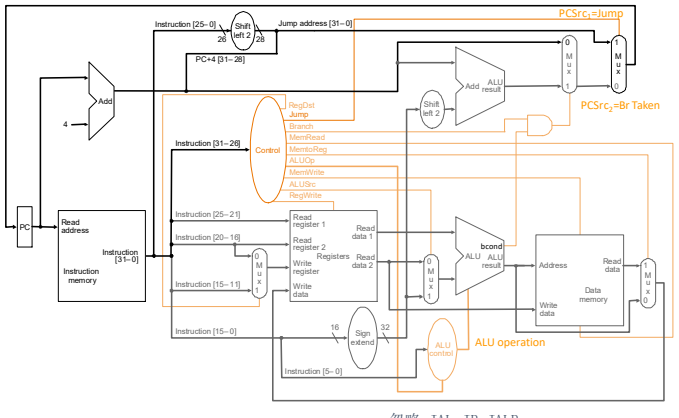



**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

41

41

## 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

42

42

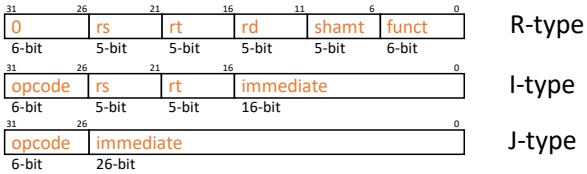
## *单周期控制逻辑*

43

43

## 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

44

44

# 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)) |

45

45

# 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” |

46

46

# ALU 控制信号

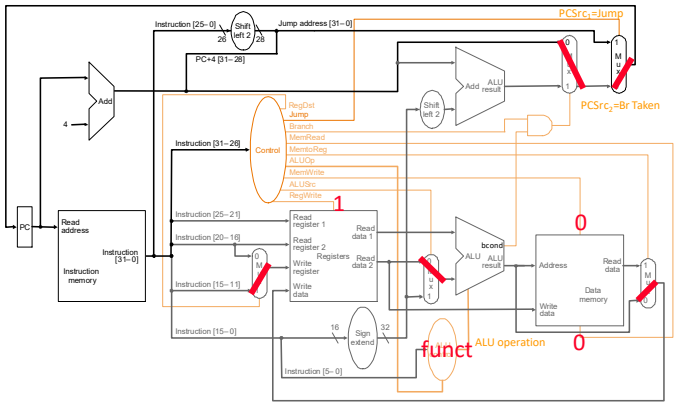
- case opcode
  - ‘0’ ⇒ 按照指令的 funct 字段决定执行的操作
  - ‘ALUi’ ⇒ 按照指令的 opcode 字段决定执行的操作
  - ‘LW’ ⇒ 加法
  - ‘SW’ ⇒ 加法
  - ‘Bxx’ ⇒ 由bcond决定操作
  - 其它 ⇒ 不用考虑

- 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.

47

47

# R-Type ALU指令的控制信号

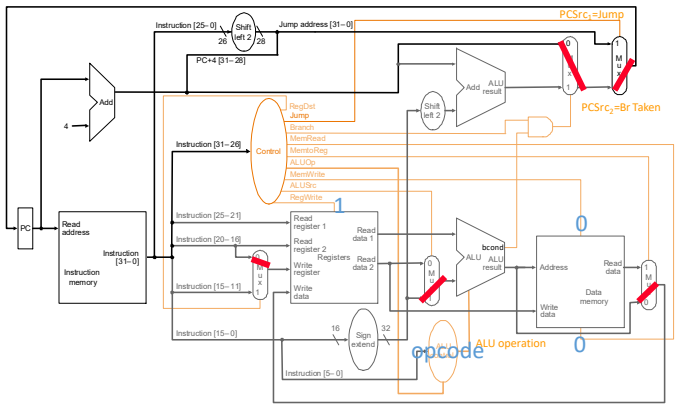



**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

48

48

# I-Type ALU指令的控制信号

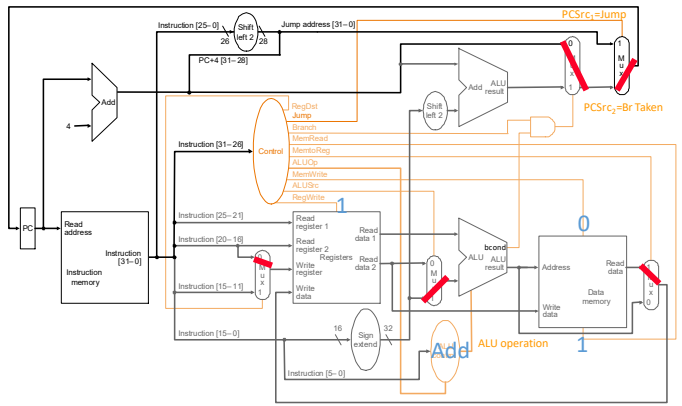





49

49

# LW指令的控制信号

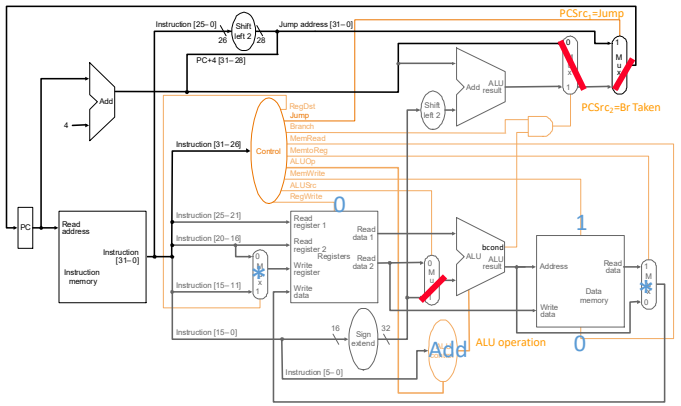





50

50

# SW指令的控制信号

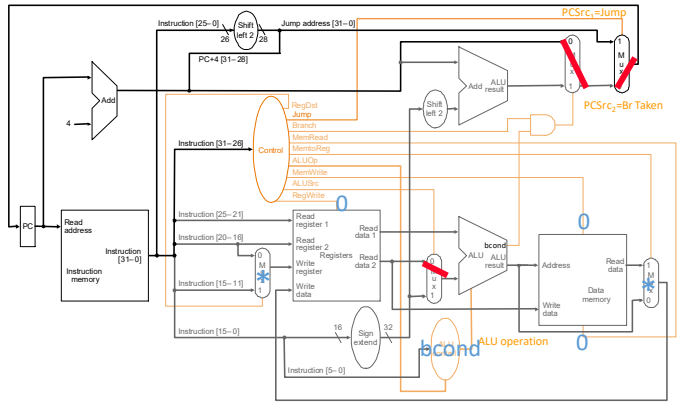





51

51

# 分支未发生的控制信号

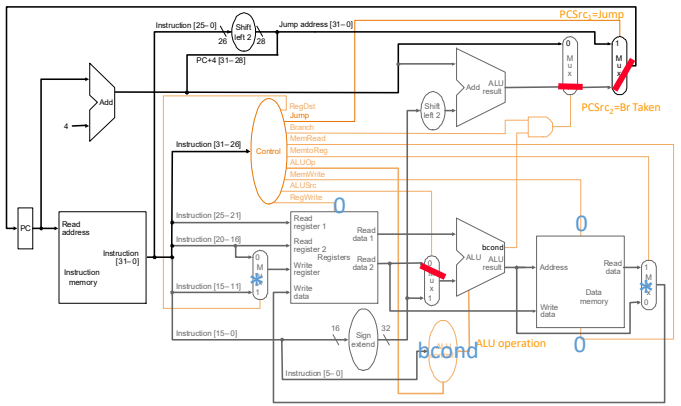





52

52

# 分支发生的控制信号

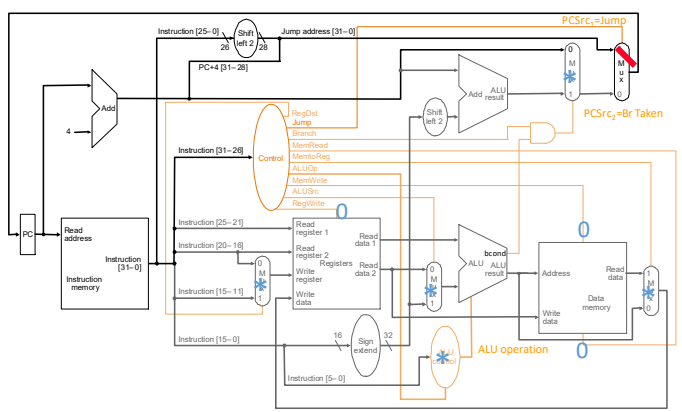





53

53

# Jump






54

54

# 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

55

55

# 单周期微体系结构性能评价

56

56

# 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?

57

57

# 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

58

58

# 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?

59

59

# 单周期数据通路分析

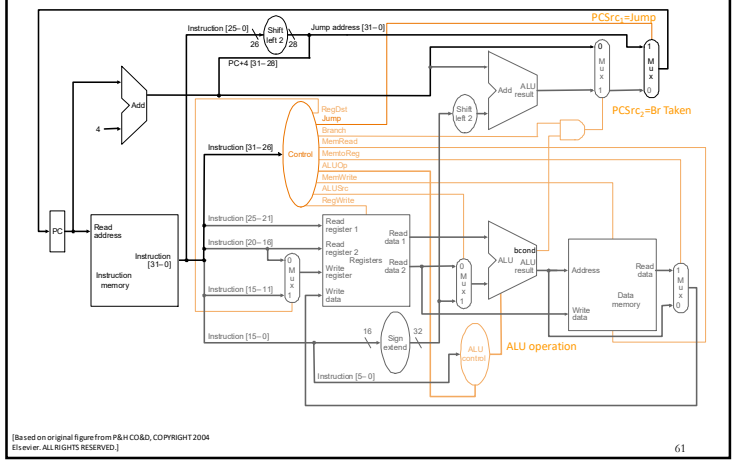
- 假设以下的部件时延
  - 内存单元(读或写): 200 ps
  - ALU和加法器: 100 ps
  - 寄存器堆(读或写): 50 ps
  - 其它组合逻辑: 0 ps

| 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 |

60

60

# 找到关键路径

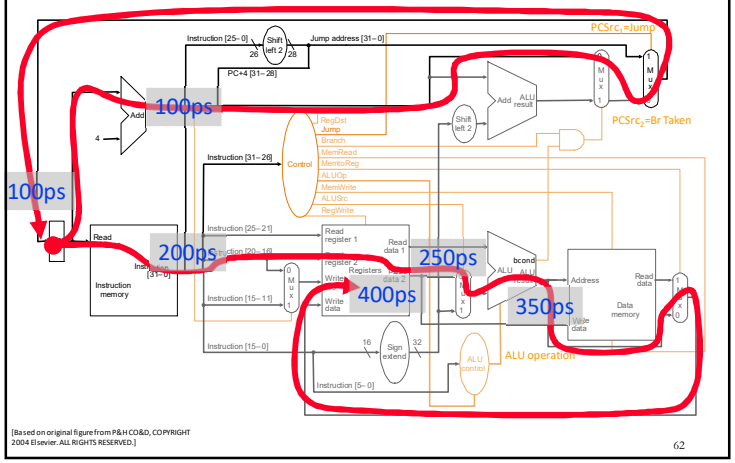



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

61

61

# R类型和I类型ALU指令

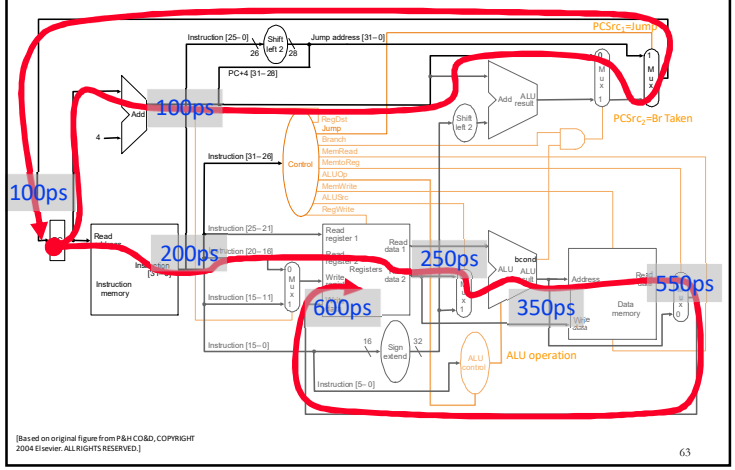



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

62

62

# LW指令

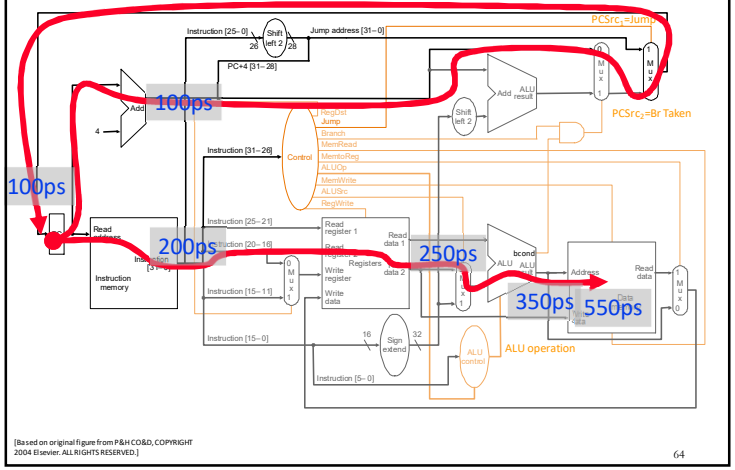



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

63

63

# SW指令

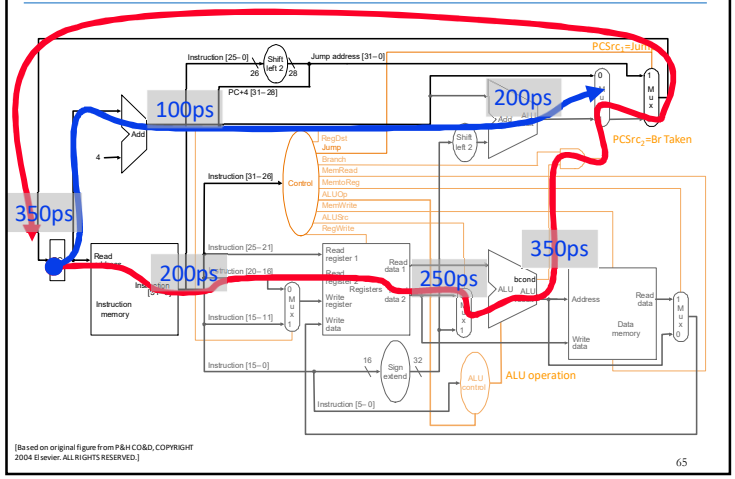



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

64

64

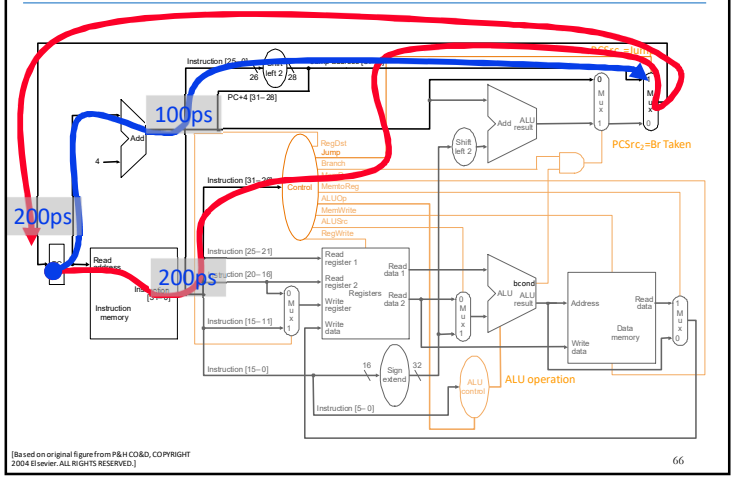
# Branch指令条件成立时




65

65

# Jump指令




66

66

# 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

67

67

# 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?

68

68

# 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





# 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?*





# *多周期微体系结构*





# 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

## 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)

73

73

## 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

74

74

## 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

75

75

## 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**

76

76

## 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

77

77