中国科学技术大学
University of Science and Technology of China

计算机组成原理
--课程信息

卢建良
lujl@ustc.edu.cn
2022年春季学期

提纲

- 教辅团队
- 课程资源
- 参考教材
- 教学安排
- 成绩考核
- 学习建议
- 课程导论

教辅团队

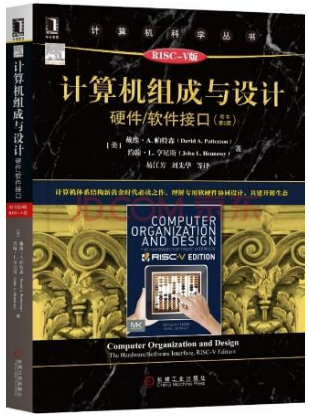
- 理论课
  - 教师：1人
    - 卢建良　　lujl@ustc.edu.cn 电三楼411室
  - 助教：6人
    - 覃云集　　qinyunji_21@mail.ustc.edu.cn
    - 郑振东　　zzd1411@mail.ustc.edu.cn
    - 何旭　　hexuustc@mail.ustc.edu.cn
    - 梁峻滔　　ljt990113@mail.ustc.edu.cn
    - 郭记　　guoji@mail.ustc.edu.cn
    - 李雨航　　hangge9468@mail.ustc.edu.cn
- 实验课
  - 教师：4人（由张俊霞老师负责）
  - 助教：组成原理课程全体理论课助教

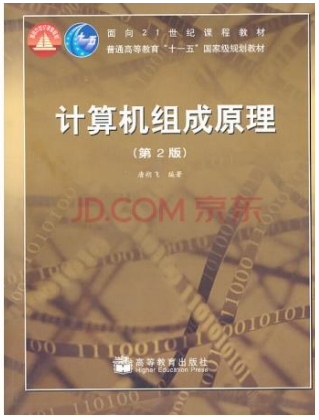
课程资源

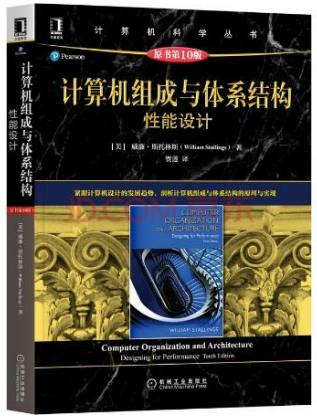
- 上课时间地点
  - 1~15周，周三（6,7）下午14:00~15:35, 3C103
  - 1~15周，周五（6,7）下午14:00~15:35, 3C103
- 课程QQ群：
  - 993174927
  - QQ群是课程信息发布及讨论的主要途径
- 课程主页：
  - BB系统
  - 网站：待建设
- 其它资源：
  - VLAB平台：vlab.ustc.edu.cn
  - 网络资源：bilibili、网易公开课、慕课等
  - 李曦、王超老师的历年课程资源是本课程的重要参考

教材及参考书

- 主要教材
  - 《Computer Organization and Design：The HW-SW I/F》RISC-V EDITION，简称COD：RISC-V版







- 辅助教材
  - 《计算机组成原理》，第2版（第3版），唐朔飞，2008
- 参考书
  - 数字设计和计算机体系结构，第2版
  - 计算机组成与体系结构：性能设计，简称COA，最新第10版

教材主要内容比较

COD：RISC-V

1. 计算机概念与工艺
2. 指令
3. 算逻运算
4. RISC处理器
   - 单周期、多周期（COD3）、流水线
   - 异常：CP0
   - 微程序：COD5（C.9）\COD3
5. 存储器层次结构
   - SRAM、DRAM、FLASH、DISK
   - Cache、Cache控制器
   - 可靠性问题（海明码）
   - 虚存
6. 并行处理器
   - 硬件多线程
   - 多核

- COD5：“I/O不单独成章”！
- 总线?

- COA9，10：以CISC为主
  - 第一部分 概论
    - 性能
  - 第二部分 计算机系统
    - 总线，内存，Cache，外存，I/O
    - *OS（虚存）（唐本无）*
  - 第三部分 CPU
    - 算逻运算、ISA，CPU结构
    - 介绍RISC、流水线概念
  - 第四部分 CU
    - A模型处理器
    - 多周期：硬连线实现，微程序实现
  - *第五部分 并行（唐本无）*
- 强调系统完整性：总线/内存/IO
- *考研：唐朔飞or白中英*

COD5（CPU、存储器、Cache、异常、MMU）+唐（总线、I/O、中断、DMA）

教学安排

- 基本遵从教材（COD：RISC-V）的内容及顺序

- ch0 课程介绍及背景知识介绍 (教材 附录A)
- ch1 计算机抽象及相关技术 (教材 第一章)
- ch2 指令：计算机的语言 (教材 第二章)
- ch3 计算机的算数运算 (教材 第三章)
- ch4 处理器 (教材 第四章)
- ch5 大而快：层次化存储 (教材 第五章)
- ch6 中断与异常 (RISC-V手册)
- ch7 总线及外设 (唐本 第三、五章)
- ch8 并行处理器：从客户端到云 (教材 第六章)

成绩考核

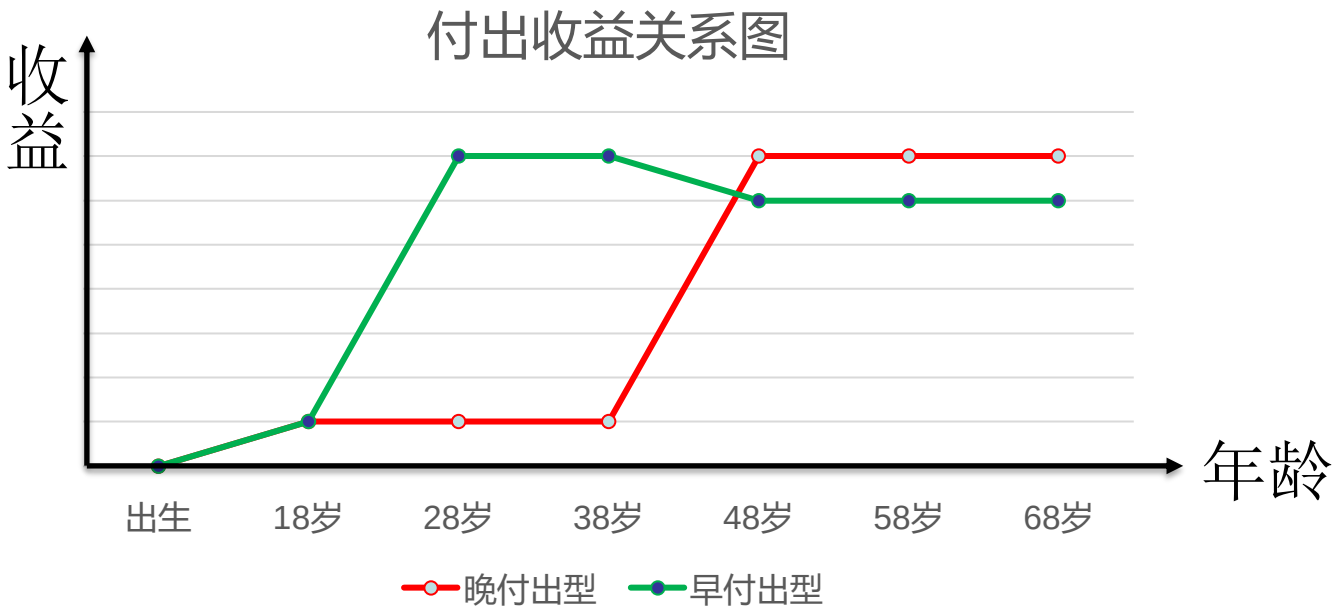
- 理论课：70%（发现抄袭，取消双方当次成绩）
  - 期末考试：60%（闭卷）
  - 课后作业：30%（不能补交）（面临挂科的同学除外）
  - 练习及实验：10%
- 实验课：30%（以下为2021年度实验安排）
  - 实验一 运算器及其应用 (1周)
  - 实验二 寄存器堆与存储器及其应用 (1周)
  - 实验三 汇编应用程序设计 (1周)
  - 实验四 多周期CPU设计 (3周)
  - 实验五 流水线CPU设计 (3周)
  - 实验六 综合设计 (2周)

学习建议

- 建立信心——基础
  - 大学所学都是非常成熟的课程，已有大量学生顺利完成了相关课程的学习
  - 要相信自己的智商
- 端正态度——方向
  - 世上无难事只怕有心人
  - 在学习上多花时间、多花精力
- 掌握方法——效率
  - 主动学习与被动学习的重要区别：课前预习！！！
  - 积极创建良好的学习环境，远离诱惑比遏制诱惑更容易
  - 充分利用网络资源：B站上面啥都有

学习建议-续

- “朝三暮四”新解
  - 狙公赋芧，曰：“朝三而暮四。”众狙皆怒。曰：“然则朝四而暮三。”众狙皆悦。名实未亏而喜怒为用，亦因是也。—《庄子·齐物论》
  - 【翻译】养猴人给猴子分橡子，说：“早上分给三升，晚上分给四升”。猴子们听了非常愤怒。养猴人便改口说：“那么就早上四升晚上三升吧。”猴子们听了都高兴起来。名义和实际都没有亏损，喜与怒却各为所用而有了变化，也就是因为这样的道理。
  - 从统计意义上看：晚付出型（朝三暮四）与早付出型（朝四暮三）所产生的历史总收益（曲线面积积分）有巨大差别



中国科学技术大学
University of Science and Technology of China

计算机组成原理
--导论

卢建良
lujl@ustc.edu.cn
2022年春季学期

投票(匿名) 最多可选5项

设置

“计算机组成原理”是什么意思?

A 计算机的组成 && 计算机的原理

B 计算机的（组成 && 原理）

C （计算机的组成）的原理

D 计算机的（组成的原理）

E 其它

提交

什么是计算机组成原理

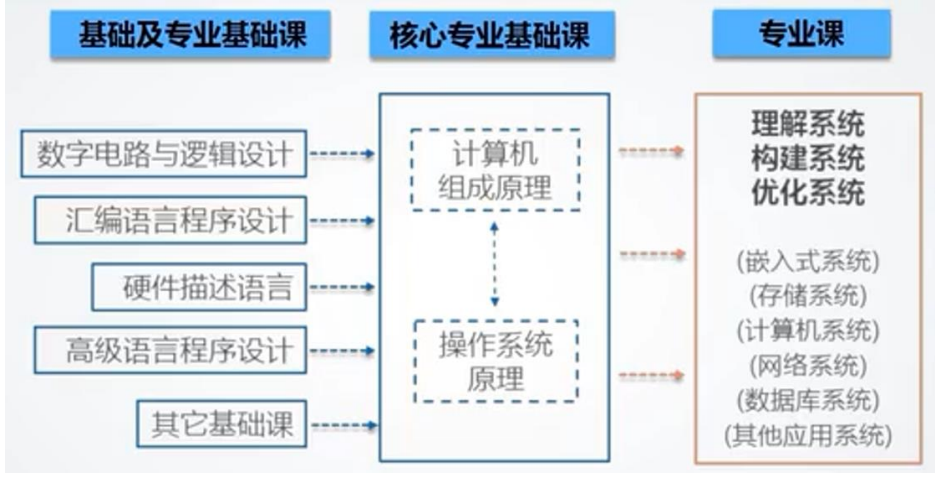
- 基本定义（baidu）
  - 在大学阶段开设的一门课程
- 基本性质（baidu）

计算机组成原理在各个计算机应用中都有应用，大学中也有不少大学开设，使用的教科书也有些差别， 但是都讲述了相近的知识。

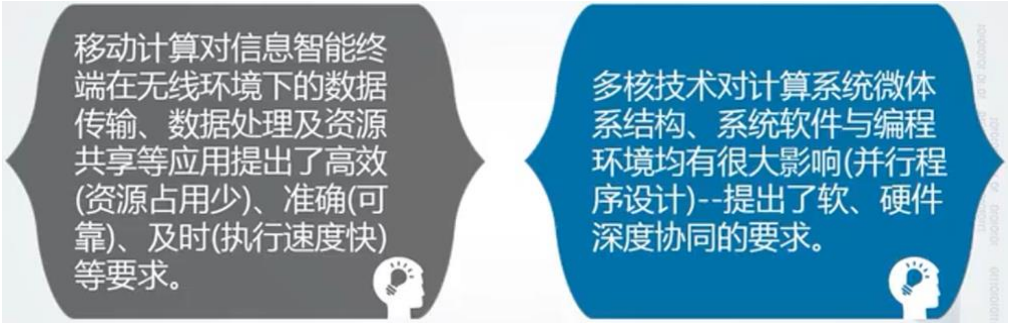
  - 计算机简史
  - 冯 诺依曼计算机组成及相关思想
  - 总线相关知识
  - 存储器相关知识
  - 处理器相关知识
  - 数据计算，原反补移码
  - 指令
  - IO操作

为什么学习计算机组成原理

- 计算机专业核心基础课，在课程体系中起到承上启下的作用
  - 介绍运算器、控制器、存储器的结构、工作原理、设计方法及互连构成整机的有关技术



- 后PC时代技术发展的迫切需求
  - 专业地位更加凸显
  - 懂软件的硬件工程师
  - 懂硬件的软件工程师
  - 精通软硬件的系统架构师



- 信息产业发展的迫切需求

- 服务于国家战略的迫切需求

引自：网上慕课

如何学好计算机组成原理

- 构造观、系统观、工程观的学习视角和学习方法
  - 构造观
  - 系统观
  - 工程观
- 多实践（实验课）
  - 利用仿真软件（Logisim、RARS等）进行设计或仿真
  - 在FPGA平台上完成部件及系统的设计、仿真及验证
- 多练习、多交流、多思考
  - 完成课上、课下练习
  - 完成实验课程内容
  - 多与老师、助教、同学进行交流
  - 多思考软硬件协同设计等相关问题

如何学好计算机组成原理

- 自学

课程目标

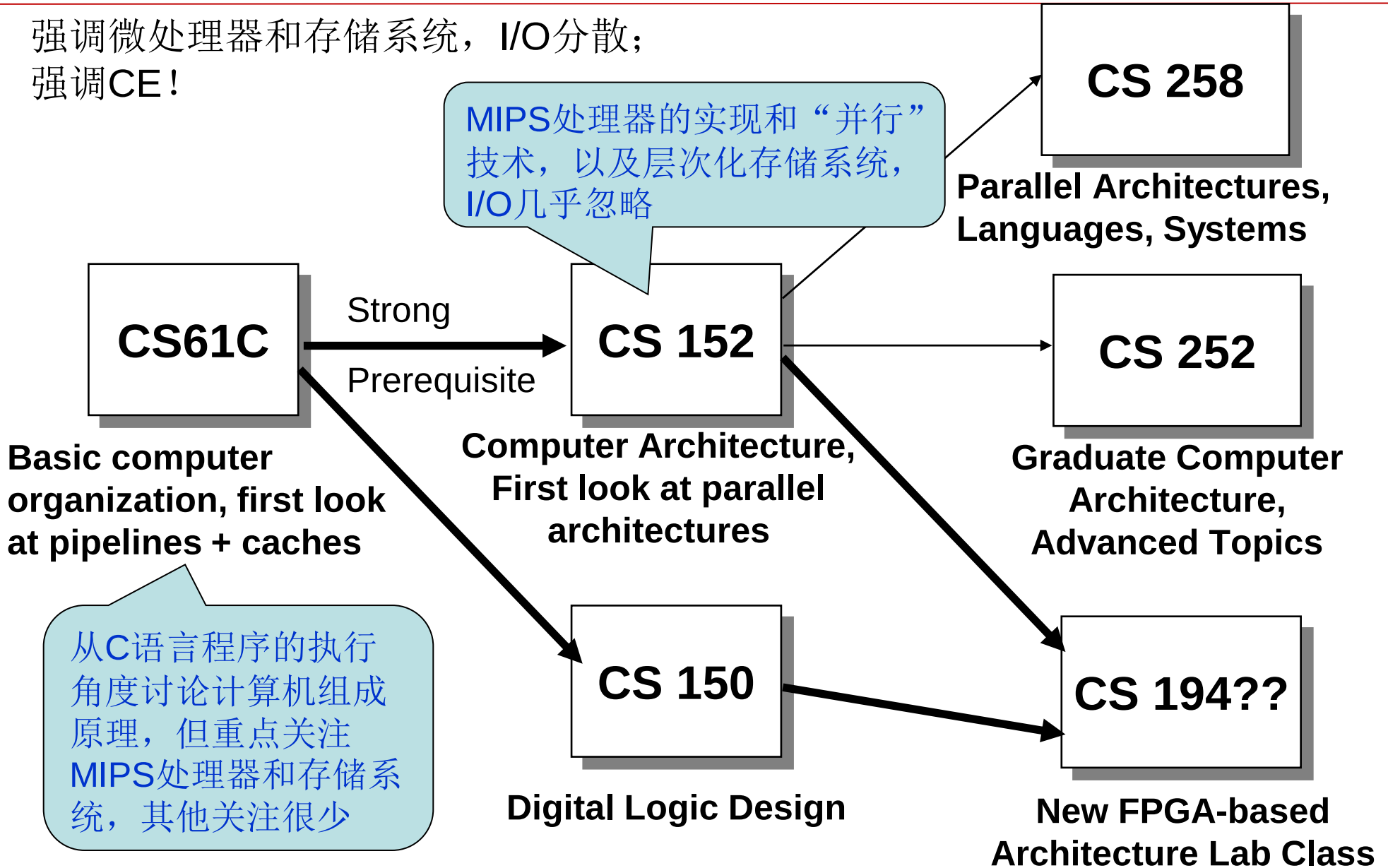
- 深刻理解现代数字计算机系统的工作原理，及软硬件设计折中
- 引导学生经历完整的设计过程，解决工程设计面临的实际问题
- 讨论计算机体系结构的变化历史，预测计算机设计的发展趋势

掌握计算机组成及工作原理

具备设计计算机原型系统的能力

UC伯克利相关课程

强调微处理器和存储系统，I/O分散；
强调CE！



USTC相关课程

- 模拟与数字电路：大二上
- 数字电路实验（VerilogHDL）：大二上
- 计算机组成原理（本课程）：大二下
  - 对计算机系统的基本组成结构和工作机制有比较透彻的理解
  - 重点讨论“单处理器”计算机系统
  - CPU：侧重COD的RISC-V模型（RISC）
  - 一般不涉及具体机型
- 微型计算机原理：大三
  - 突出应用，详细讲述X86微处理器编程结构、汇编语言、接口技术和应用编程方法
- 计算机体系结构：大三下
  - 计算机系统的设计优化技术和性能定量分析方法
  - 多处理器/多核，并行

- 本课程的目标
  - 深入理解计算机系统的硬件组成、工作原理和软硬件I/F
  - 深入理解处理器的内部结构和工作原理
  - 深入理解各个功能部件的系统级和RTL（寄存器传输级）设计过程

相关课程资源

- 美国UC Berkeley大学“Machine Structure”2012年课程网站：
  http://inst.eecs.berkeley.edu/~cs61c/sp12/
- 美国UC Berkeley大学“Components and Design Techniques for Digital System”2012年课程网站：http://inst.eecs.berkeley.edu/~cs150/sp12/
- 美国UC Berkeley大学“Computer Architecture and Engineering”2012课程网站：
  http://inst.eecs.berkeley.edu/~cs152/sp12/
- 美国Stanford大学“Computer Organization and Systems” 2012年课程网站：
  https://courseware.stanford.edu/pg/courses/281000/cs107-spring-2012
- 美国Stanford大学“Digital Systems II”课程网站：
  http://www.stanford.edu/class/ee108b/
- 美国Stanford大学“Digital Systems II”课程网站：
  http://www.stanford.edu/class/cs110/
- 美国Carnegie Mellon 大学“Introduction to Computer Architecture”课程网站：
  http://www.cs.cmu.edu/~213/
- 美国Carnegie Mellon 大学“Introduction to Computer Architecture”课程网站：
  http://www.ece.cmu.edu/~ece447/
- 美国Univ. Illinois at Urbana-Champaign“Computer Architecture II”课程网站：
  http://www.cs.uiuc.edu/class/sp11/cs232/
- 美国麻省理工学院(MIT)“Computation Structures”课程网站：
  http://6004.csail.mit.edu

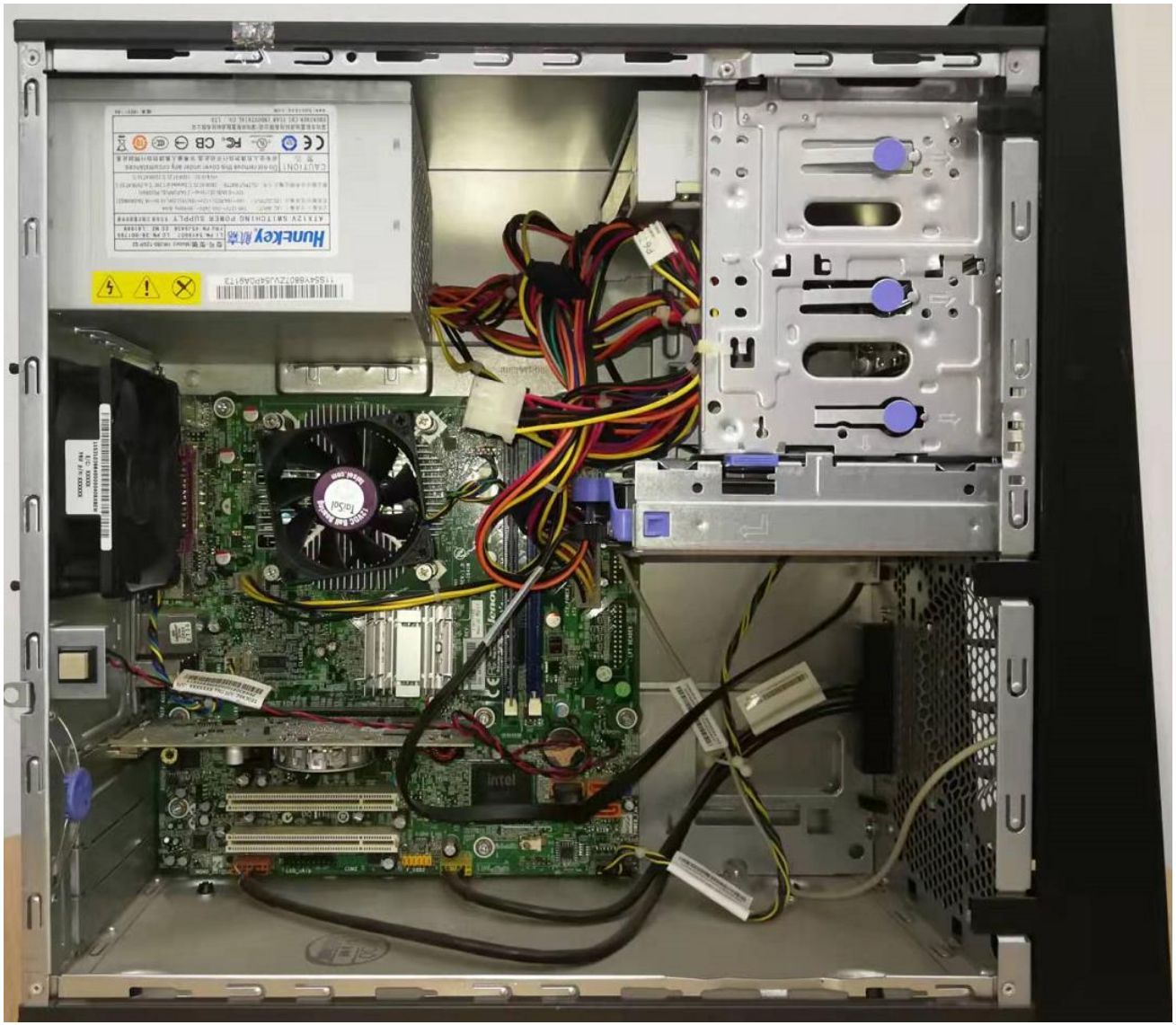
引自：李曦老师课件

请思考以下问题

- 程序是如何转换成机器语言的?
- 计算机硬件如何执行机器语言?
- 计算机硬件与软件的接口是什么?
- 哪些因素会影响到程序的性能? 如何提高?
- 硬件设计者如何提高计算机性能?
- 什么是并行处理? 如何实现?

现场拆机实验

现场拆机实验



现场拆机实验

现场拆机实验

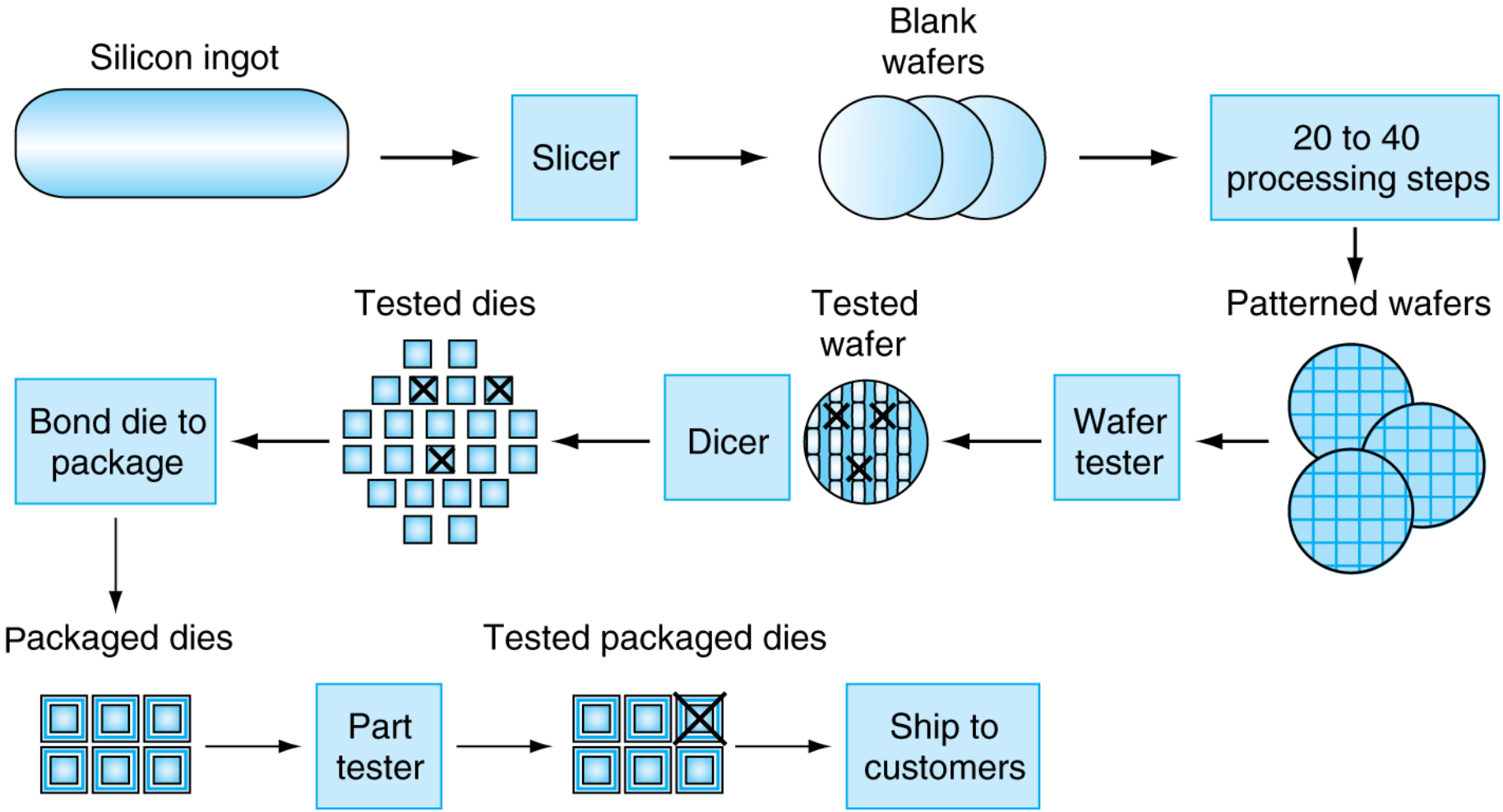
- https://www.bilibili.com/video/BV1EX4y1F7nM?from=search&seid=6151589271298041935&spm_id_from=333.337.0.0
  - 输入、输出设备
  - 存储器：硬盘、内存、缓存
  - 处理器：运算器、控制器
  - 互联设备
  - 电源
  - 其它

PCB + 芯片 + 其它

PCB制作

- 确定设计方案
- 绘制原理图
- 绘制PCB
- 加工PCB，采购元器件
- 焊接PCB
- 调试
- https://www.bilibili.com/video/BV1AA411P7oS/?spm_id_from=autoNext
- 演示: https://lceda.cn/

芯片制作流程


Silicon ingot
Slicer
Blank wafers
20 to 40 processing steps
Patterned wafers
Wafer tester
Tested wafer
Dicer
Tested dies
Bond die to package
Packaged dies
Part tester
Tested packaged dies
Ship to customers

从沙子到芯片

- Intel
  - https://www.bilibili.com/video/BV1Rt411A7bV?from=search&seid=17847863675708071328&spm_id_from=333.337.0.0
- 中芯国际
  - https://www.bilibili.com/video/BV1bE411C76e?from=search&seid=17847863675708071328&spm_id_from=333.337.0.0
- 详细介绍
  - https://www.bilibili.com/video/BV1hL411473Y/?spm_id_from=333.788.recommend_more_video.0

中国科学技术大学
University of Science and Technology of China

计算机组成原理
--数字电路背景知识

卢建良 lujl@ustc.edu.cn

2022年春季学期

提纲

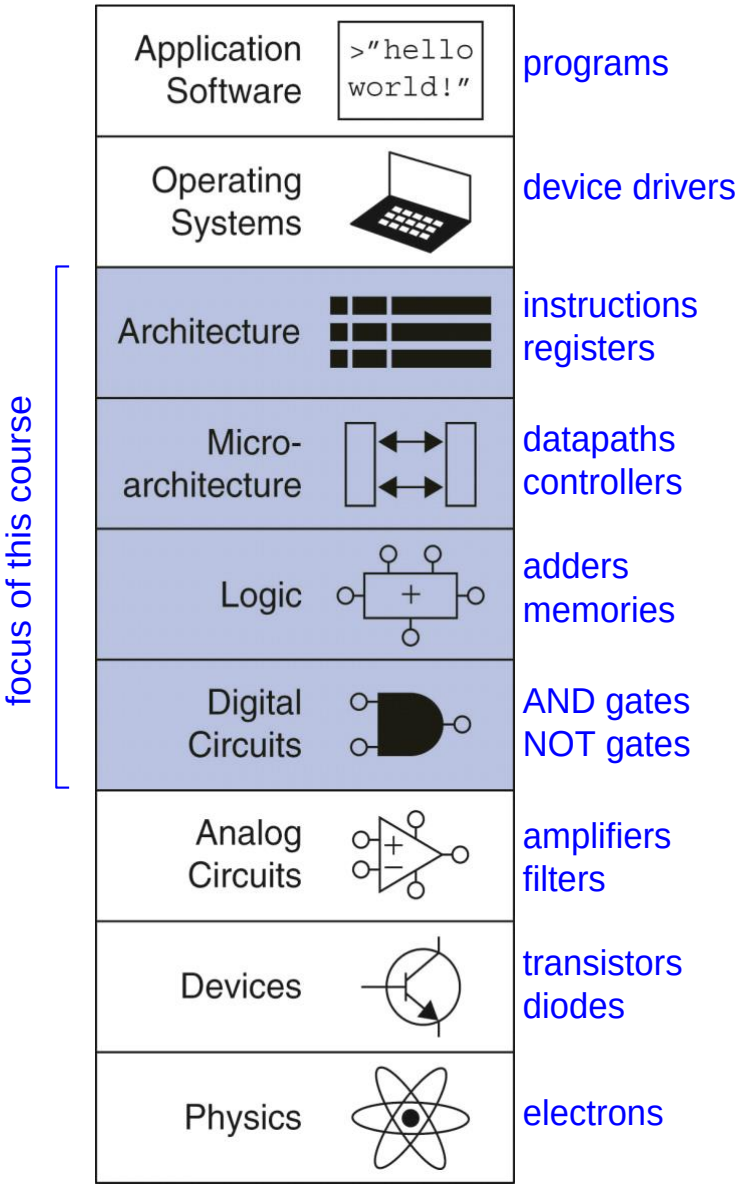
- 1. 引言
- 2. 门、真值表、逻辑方程
- 3. 组合逻辑电路
  - 译码器、多选器、两级逻辑（PLA）、ROM、无关项、逻辑单元阵列
- 4. 硬件描述语言
  - 数据类型和操作、Verilog代码结构、复杂组合逻辑的表示
- 5. 构建基本算数逻辑单元（ALU）
  - 1位ALU、64位ALU、修改64位ALU以适应RISC-V、用Verilog定义RISC-V ALU
- 6. 快速加法：超前进位
- 7. 时钟
- 8. 存储元件：触发器、锁存器和寄存器
  - 触发器和锁存器、寄存器堆、使用Verilog描述时序逻辑

提纲-续

- 9. 存储元件：SRAM和DRAM
  - SRAM、DRAM、错误修正
- 10. 有限状态机（FSM）
- 11. 定时方法
  - 电平敏感的时钟控制、异步输入和同步器
- 12. 现场可编程设备
- 13. 本章小结

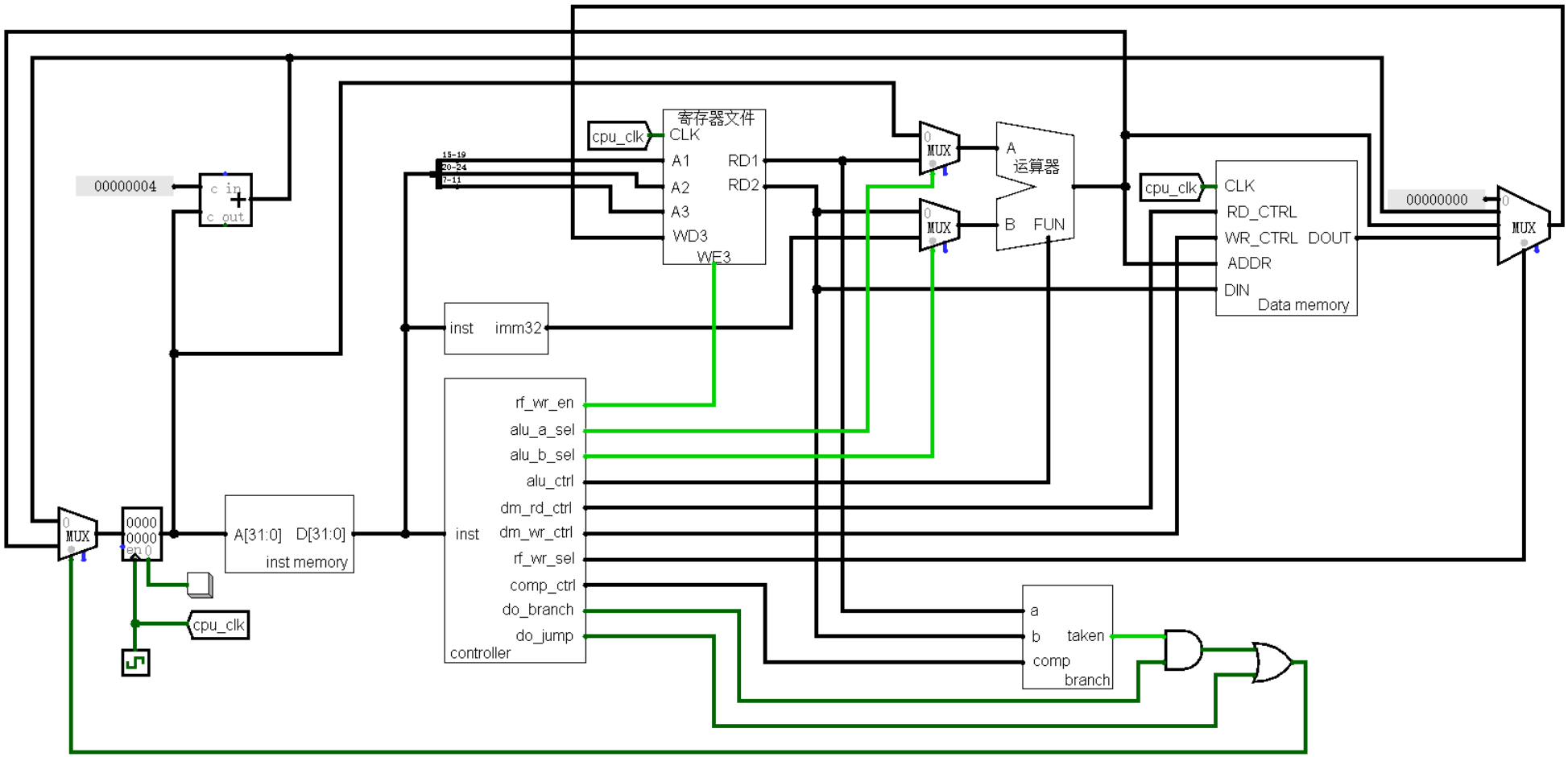
A.1 引言

- 本章简要讨论逻辑设计的基础知识
- 为理解本课程内容提供必要的背景知识储备
- 在适当的地方配有Verilog代码片段
- 完整的Verilog教程网站：http://staff.ustc.edu.cn/~han/CS152CD/Content/Tutorials/Verilog/VOL/main.htm
- Verilog在线测评网站
  - https://hdlbits.01xz.net/wiki/Step_one
  - https://verilogoj.ustc.edu.cn/oj/



A.1 引言

- 复习数字电路知识是为设计RV32I CPU做准备

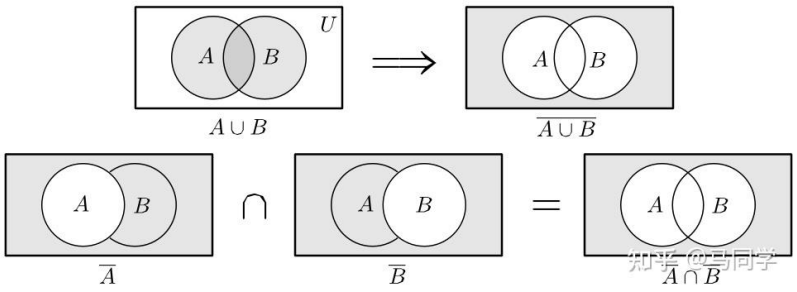


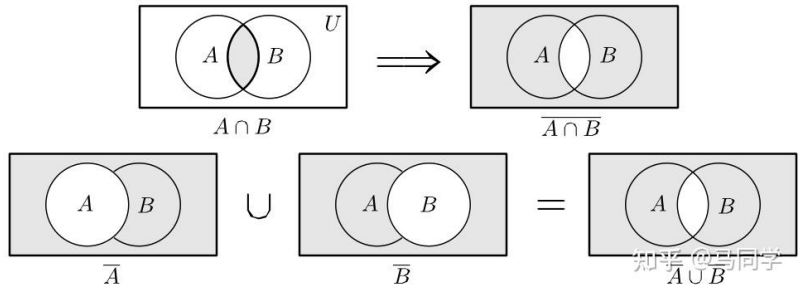
A.2 门、真值表和逻辑方程

- 数字电路是现代计算机的内部核心
- 数字电路有两个稳定的电平状态：高电平 vs 低电平
  - 1 vs 0、 真 vs 假、 有效 vs 无效、正 vs 负、高 vs 低、 阳 vs 阴
  - 隐患：电路处于不稳定的电平状态时会如何?
- 组合逻辑电路 vs 时序逻辑电路
  - 电路中是否含有存储单元
  - 输出是否与之前的状态有关
- 真值表
  - 输入逻辑变量所有取值的组合与其对应的输出逻辑函数值构成的表格
  - 缺点：表项增长太快、不容易理解
  - 改进：有时采用仅列举非零输出表项的简化真值表
- 布尔代数
  - 或操作： A + B
  - 与操作： A · B
  - 非操作： /A

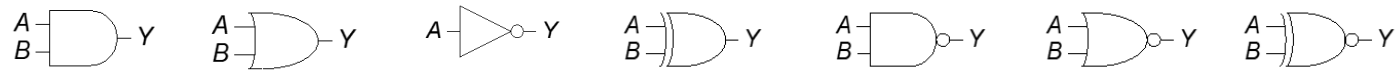
A.2 门、真值表和逻辑方程-续

- 布尔代数-续
  - 布尔代数中的基本定律：恒等定律、0/1定律、互补律、交换律、结合律、分配律
  - 德摩根定律 $\overline{A \cup B} = \overline{A} \cap \overline{B}$ $\overline{A \cap B} = \overline{A} \cup \overline{B}$

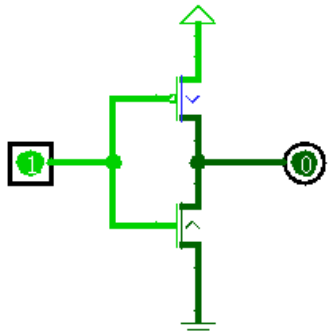




- 门（gate）：实现基本逻辑函数的单元，与、或、非、与非等
  - 逻辑门



  - 逻辑门电路：具有逻辑门功能的电路单元
  - 可以使用CMOS工艺实现逻辑门电路
  - 在Logisim软件中进行行为仿真
  - 万能门电路：或非门、与非门
    - 任何逻辑电路都可以只使用该类型构建

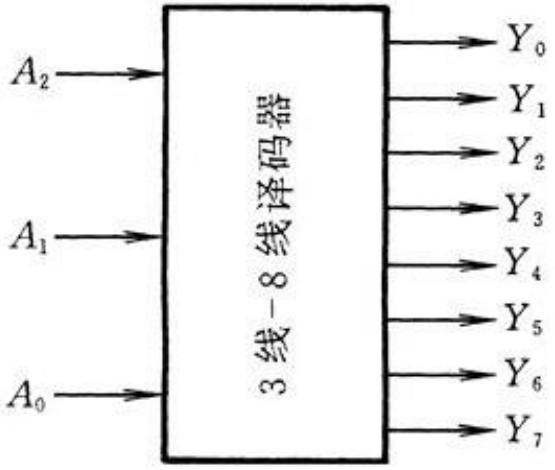


$Y = \overline{A}$

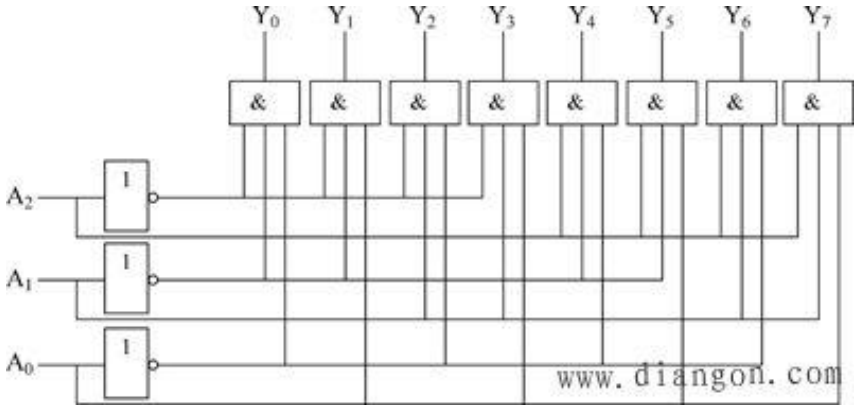
A	Y
0	1
1	0

A.3 组合逻辑电路

- 译码器（decoder）
  - 具有n位输入和2^n个输出的逻辑单元
  - 每种输入组合仅对应一个有效输出
  - 最常见的为3-8译码器
  - 用途：
    - 用来生成不同时使用的多个功能模块的片选或使能信号
    - 用来实现IO接口的扩展，如数码管
  - 与其相对应的是编码器，如何实现？



输入			输出							
A_2	A_1	A_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



A.3 组合逻辑电路

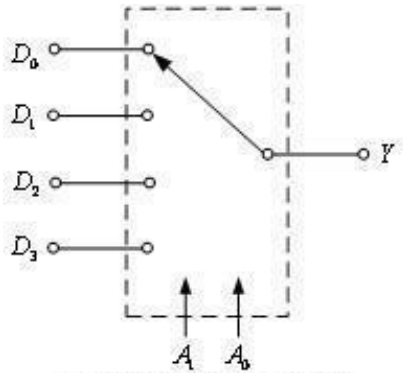
- 练习：
- 用Verilog实现3-8译码器
- 用Verilog实现8-3编码器

A.3 组合逻辑电路

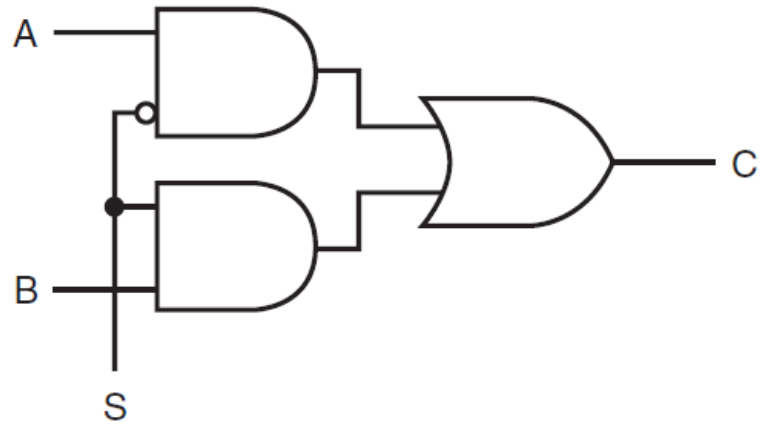
- 多路选择器（Multiplexor/Mux）
  - 多路选择器是数据选择器的别称。在多路数据传送过程中，能够根据需要将其中任意一路选出来的电路，叫做数据选择器，也称多路选择器或多路开关
  - 逻辑函数：

    C = （A · /S） + （B · S）
  - Verilog实现

```
module mux2(A,B,S,C);
    input A,B,S;
    output C;
    assign C = S ? B : A ;
endmodule
```

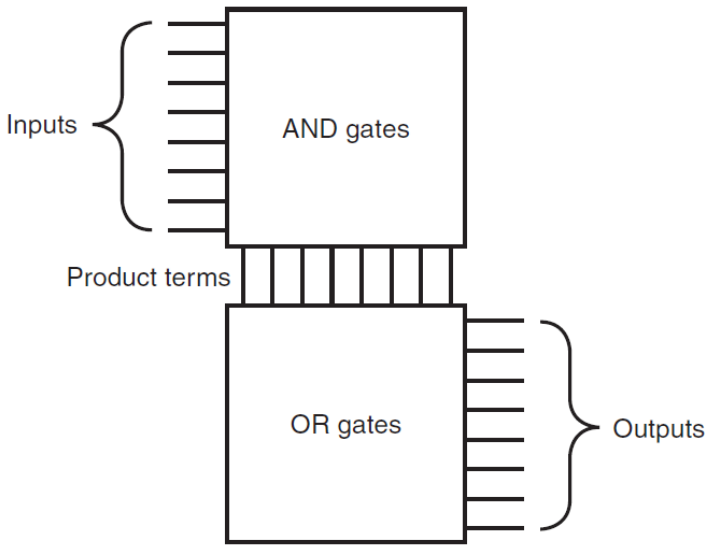


4选1数据选择器原理示意



A.3 组合逻辑电路

- 两级逻辑、PLA
  - 任何逻辑都可以写成“或-与式”或者“与-或式”的表示形式
  - 或-与式（ product of sums ） $E = \overline{(\bar{A} + \bar{B} + C) \cdot (\bar{A} + \bar{C} + B) \cdot (\bar{B} + C + A)}$
  - 与-或式（sum of products） $E = (A \cdot B \cdot \bar{C}) + (A \cdot C \cdot \bar{B}) + (B \cdot C \cdot \bar{A})$
  - 与-或式对应于可编程逻辑阵列（PLA）的常用结构化实现
  - PLA： **P**rogrammable **L**ogic **A**rray

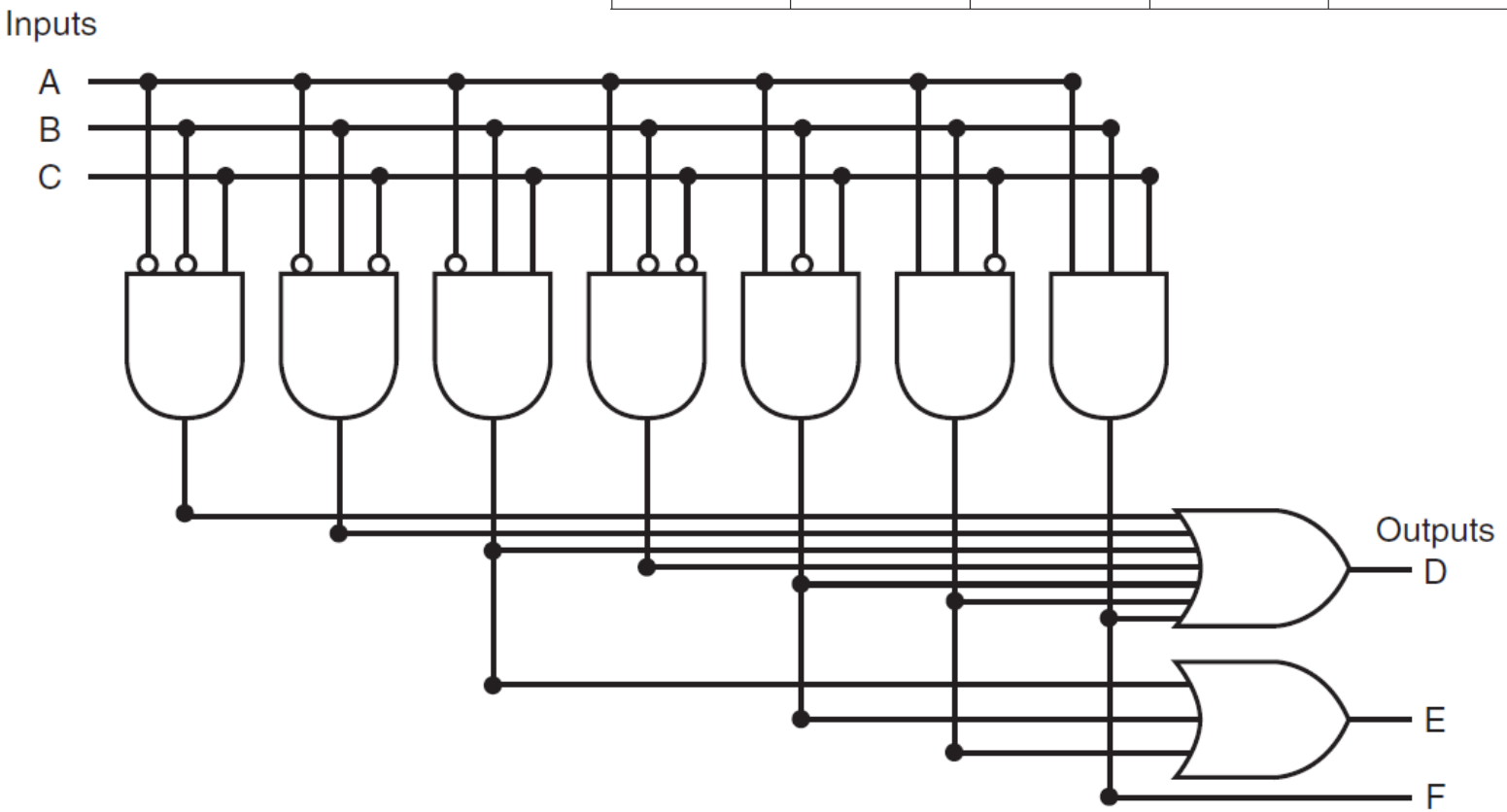


Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

A.3 组合逻辑电路

- 两级逻辑、PLA
  - 例：用PLA实现真值表所描述的电路

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1



A.3 组合逻辑电路

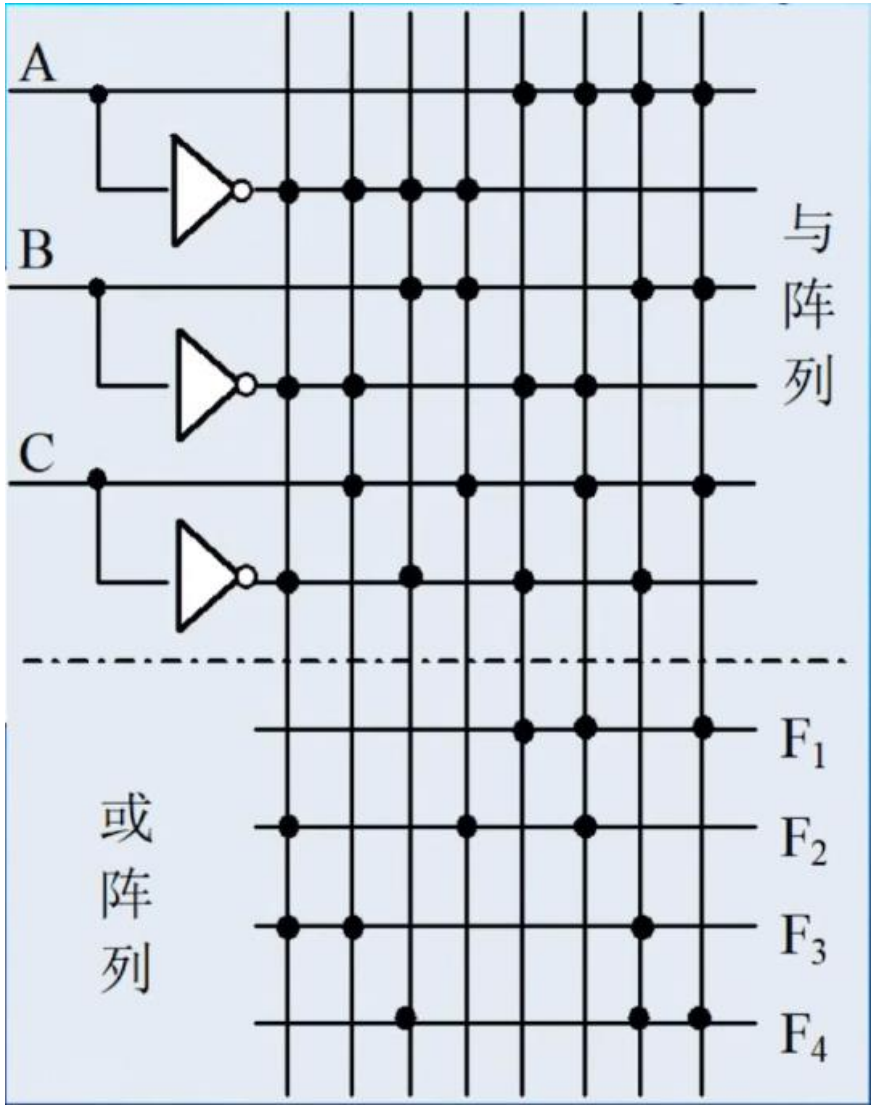
- 两级逻辑、PLA
  - 例：用PLA器件实现下列逻辑函数

$$F_1 = A\bar{B} + AC$$

$$F_2 = \bar{A}BC + A\bar{B}C + \bar{A}\bar{B}\bar{C}$$

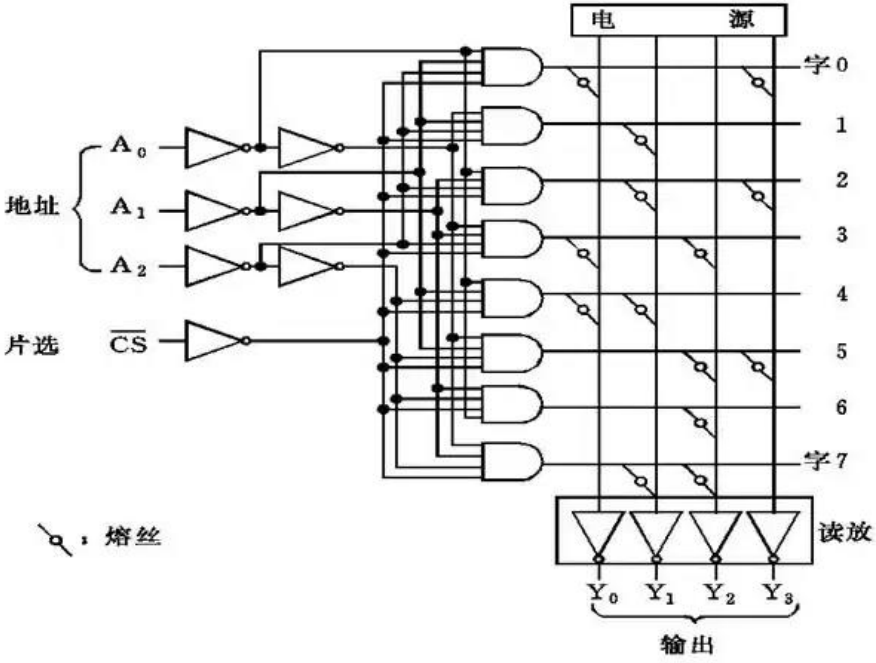
$$F_3 = \bar{A}\bar{B} + AB\bar{C}$$

$$F_4 = \bar{A}B\bar{C} + AB$$

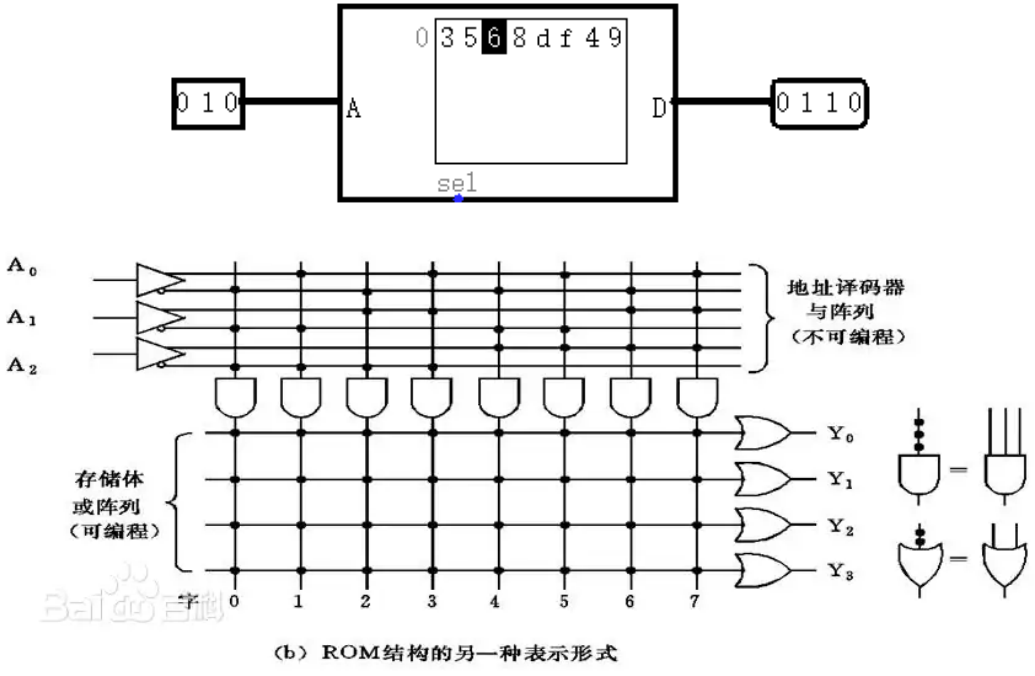


A.3 组合逻辑电路

- ROM
  - Read-Only Memory，只读存储器
  - 一种可以长期保存信息的存储器，具有断电后信息仍可继续保存的特点，在正常工作时只可读取数据，而不能写入数据
  - 是用于实现组合逻辑函数结构化逻辑形式的另一方式



（a）熔丝型8×4ROM原理图



（b）ROM结构的另一种表示形式

A.3 组合逻辑电路

- 无关项
  - 分为输入无关项和输出无关项
  - 无关项对逻辑函数的优化至关重要
  - 例题：

Consider a logic function with inputs A, B, and C defined as follows:

- If A or C is true, then output D is true, whatever the value of B.
- If A or B is true, then output E is true, whatever the value of C.
- Output F is true if exactly one of the inputs is true, although we don't care about the value of F, whenever D and E are both true.

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

A.3 组合逻辑电路

- 无关项
  - 输出无关项化简

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

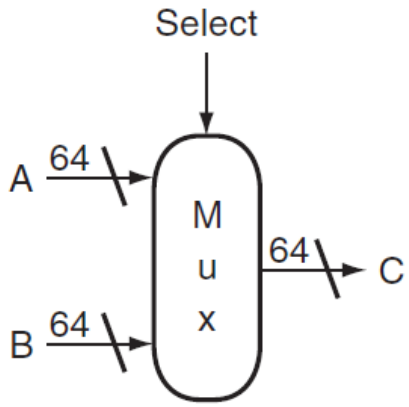
  - 输入无关项化简

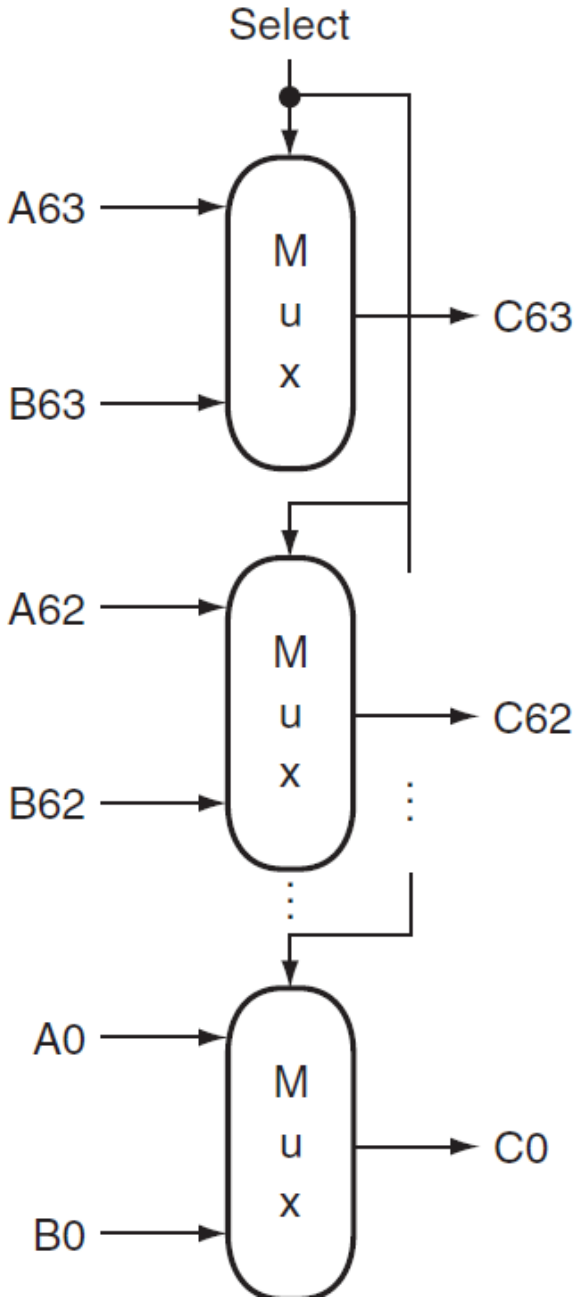
Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

A.3 组合逻辑电路

- 逻辑单元阵列，总线
  - 数据处理时经常需要对整个数据字（32bit or 64bit）进行处理，因此需要构建逻辑单元阵列，又称总线，bus
  - PS:总线也用于指示具有多信号源和多设备共享的线路集合
  - Verilog实现：

```
module mux64(
input [63:0] A,B,
input S,
output [63:0] C);
    assign C = S ? B : A;
endmodule
```





单选题 1分

设置

Parity is a function in which the output depends on the number of 1s in the input. For an even parity function, the output is 1 if the input has an even number of ones. Suppose a ROM is used to implement an even parity function with a 4-bit input. Which of A, B, C, or D represents the contents of the ROM?

Address	A	B	C	D
0	0	1	0	1
1	0	1	1	0
2	0	1	0	1
3	0	1	1	0
4	0	1	0	1
5	0	1	1	0
6	0	1	0	1
7	0	1	1	0
8	1	0	0	1
9	1	0	1	0
10	1	0	0	1
11	1	0	1	0
12	1	0	0	1
13	1	0	1	0
14	1	0	0	1
15	1	0	1	0

A

B

C

D

提交

A.4 使用硬件描述语言

- 硬件描述语言
- Verilog中的数据类型和操作
  - 两种基本数据类型：wire 、 reg
  - 定义数据向量：reg [31:0] x; wire [63:0] y;
  - 定义数据组：reg [31:0] regfile[0:31];
  - reg、wire信号可能的取值有：0、1、X、Z
  - 常量值可以指定为2,8,10,16进制数
    - `4'b0100` specifies a 4-bit binary constant with the value 4, as does `4'd4`.
    - `-8'h4` specifies an 8-bit constant with the value −4 (in two's complement representation)

Values can also be concatenated by placing them within { } separated by commas. The notation `{x{bitfield}}` replicates `bitfield` `x` times. For example:

- `{32{2'b01}}` creates a 64-bit value with the pattern 0101 … 01.
- `{A[31:16],B[15:0]}` creates a value whose upper 16 bits come from `A` and whose lower 16 bits come from `B`.

A.4 使用硬件描述语言

Check Yourself

Which of the following define exactly the same value?

1. `8'bimoooo`
2. `8'hF0`
3. `8'd240`
4. `{{4{1'b1}},{4{1'b0}}}`
5. `{4'b1,4'b0)`

A.4 使用硬件描述语言

- Verilog程序的结构
  - `initial` constructs, which can initialize `reg` variables
  - Continuous assignments, which define only combinational logic
  - `always` constructs, which can define either sequential or combinational logic
  - Instances of other modules, which are used to implement the module being defined
- Verilog复杂组合逻辑的表示
  - assign

```
module half_adder (A,B,Sum,Carry);
   input A,B; //two 1-bit inputs
   output Sum, Carry; //two 1-bit outputs
   assign Sum = A ^ B; //sum is A xor B
   assign Carry = A & B; //Carry is A and B
endmodule
```

  - always
    - 阻塞赋值：用于组合逻辑，用“=”表示
    - 非阻塞赋值：用于时序逻辑，用“<=”表示

A.4 使用硬件描述语言

- Verilog复杂组合逻辑的表示
  - always

```
module RISCVALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [63:0] A,B;
   output reg [63:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

A.4 使用硬件描述语言

Check Yourself Assuming all values are initially zero, what are the values of A and B after executing this Verilog code inside an `always` block?

```
C = 1;
A <= C;
B = C;
```

阻塞赋值示例：

```
always@(posedge clk)
begin
        a = 1;
        b = a;
        c  = b;
end
```

不建议使用

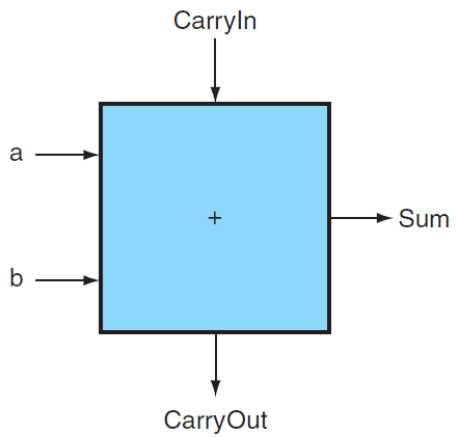
非阻塞赋值示例：

```
always@(posedge clk)
begin
        a <= 1;
        b <= a;
        c  <= b;
end
```

建议使用

A.5 构建基本算数逻辑单元

- 算数逻辑单元：Arithmetic Logical Unit，简称ALU、运算器
  - RV32I的数据位宽为32bit，因此需要构建32bit位宽的ALU
  - 首先构建1bit位宽ALU
- 1位ALU
  - 支持与、或、加法三种逻辑运算



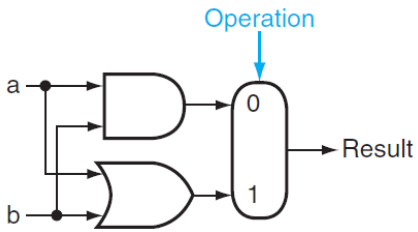
Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

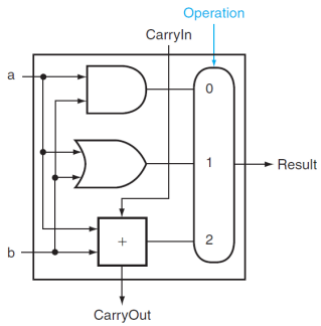
$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

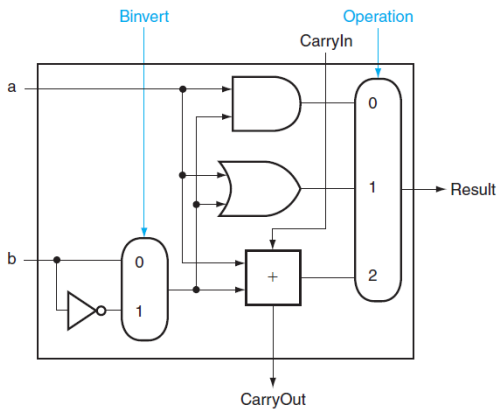
A.5 构建基本算数逻辑单元

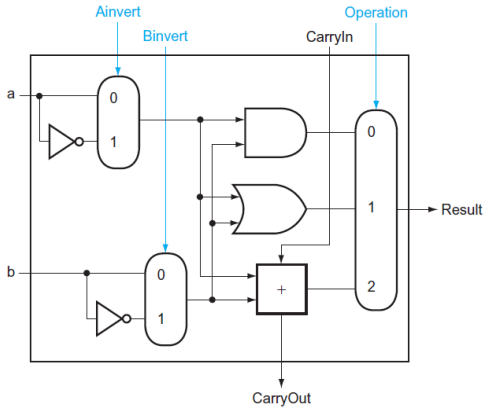
- 1位ALU
  - 支持与、或、加法三种逻辑运算

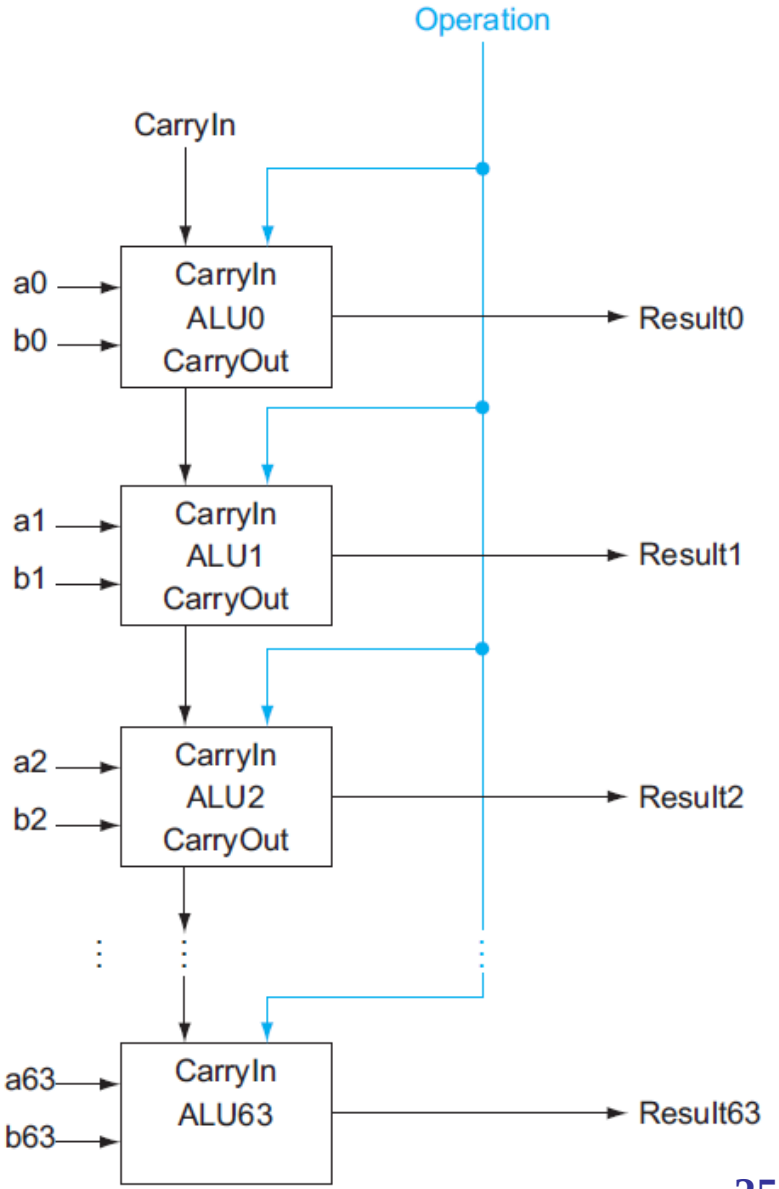




- 64位ALU
  - 将64个1bit ALU依次级联
  - 增加减法操作、或非操作

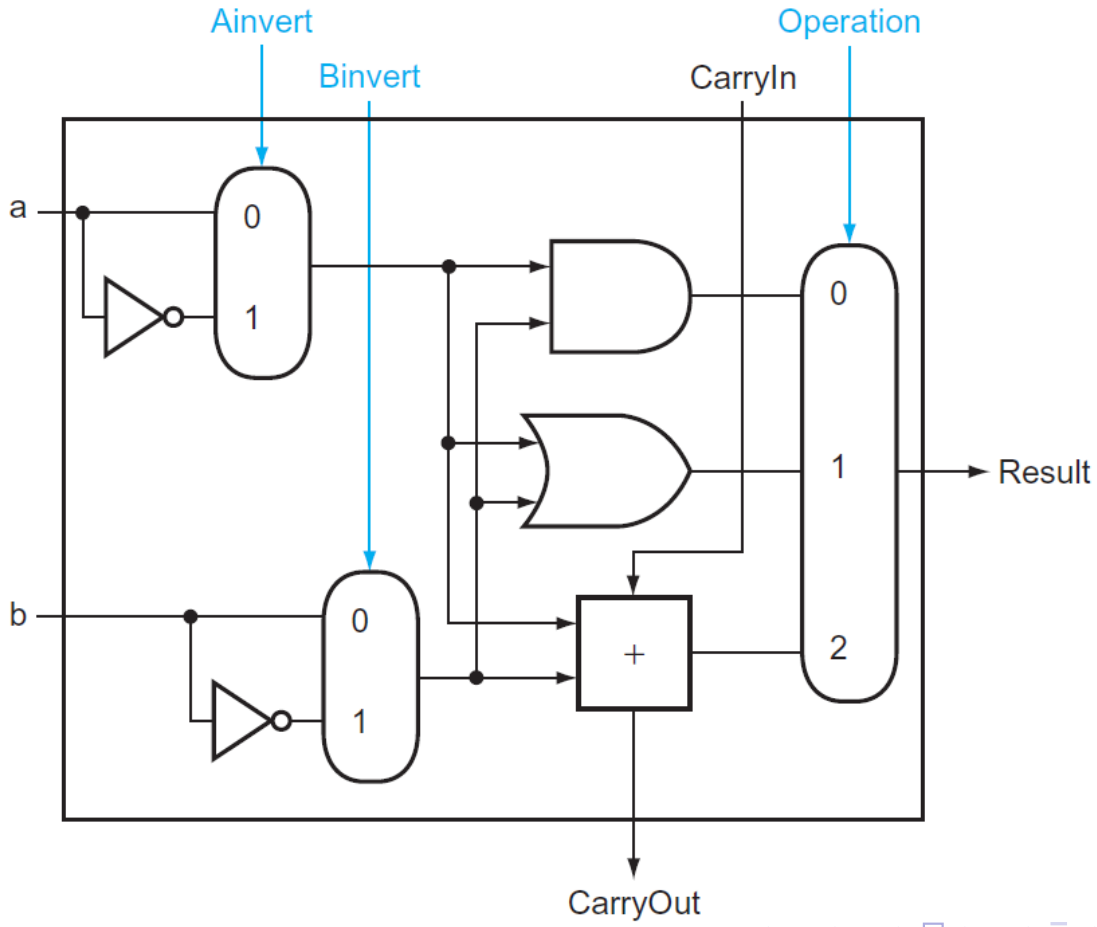






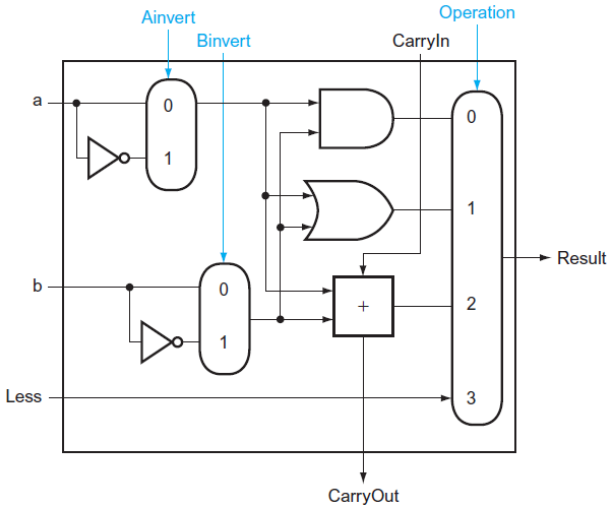
A.5 构建基本算数逻辑单元

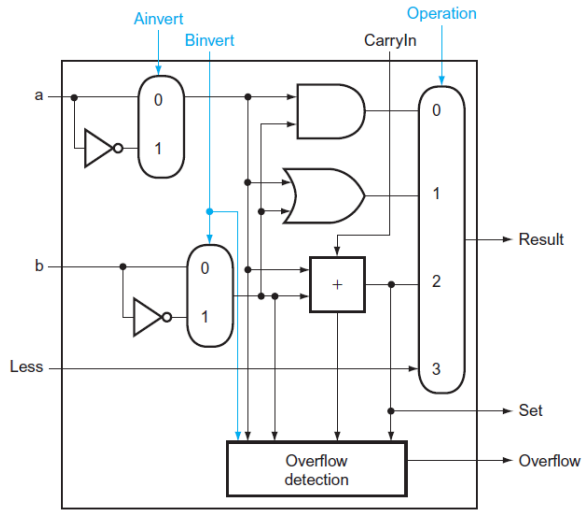
- 请思考：
- 上述64bit ALU已支持加法、减法、与、或、非、或非、与非等操作，还需要支持哪些逻辑操作？

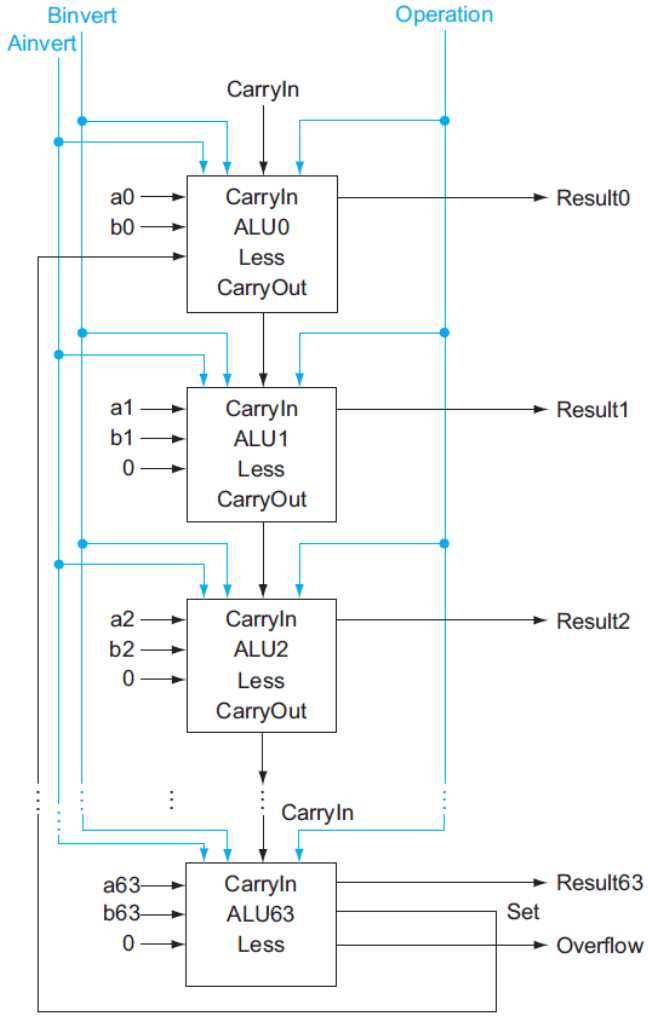


A.5 构建基本算数逻辑单元

- 修改ALU以适应RISC-V
  - 增加小于置位指令（slt，set less than）
  - 如果a<b，则输出1，否则输出0

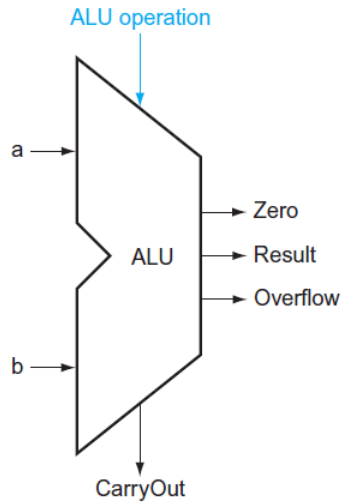




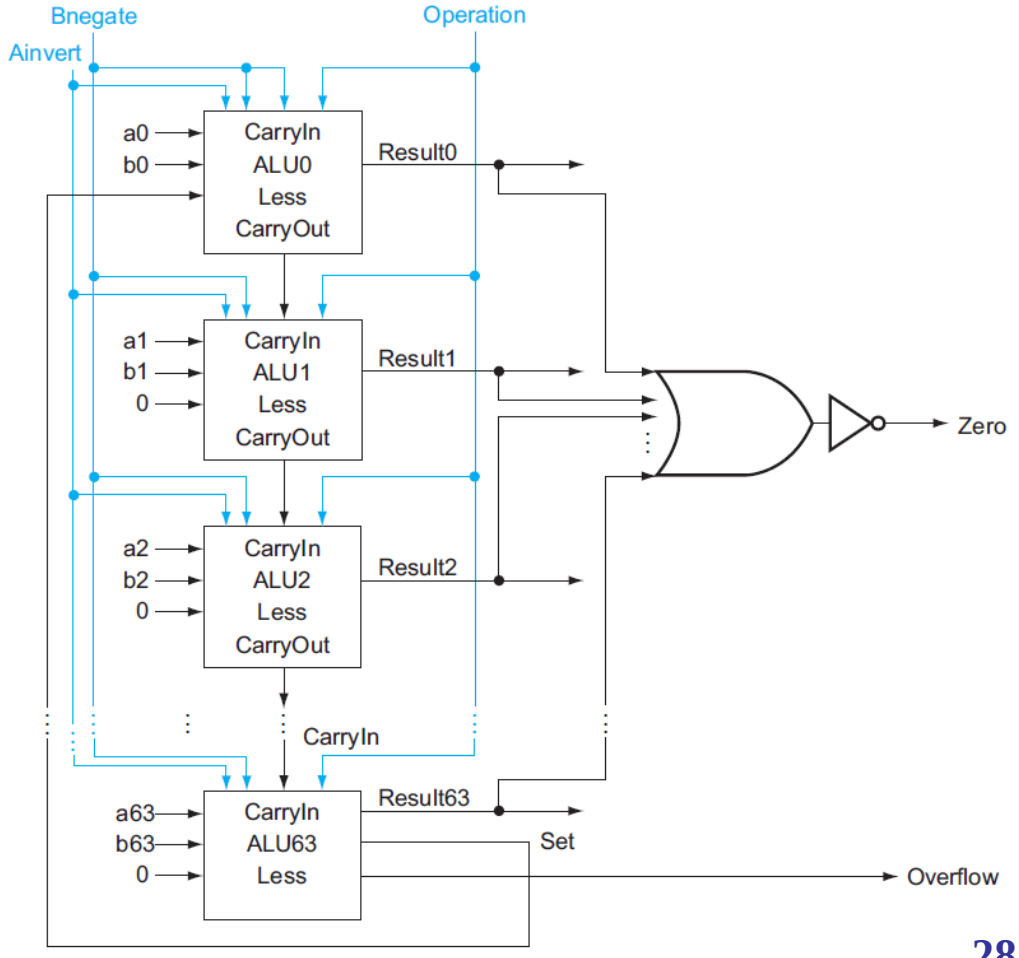


A.5 构建基本算数逻辑单元

- 修改ALU以适应RISC-V
  - 增加相等判断逻辑（beq指令，相等则跳转）



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR



A.5 构建基本算数逻辑单元

- 用Verilog定义RISC-V ALU
  - 纯组合逻辑电路
  - RISCVALU：算数逻辑单元
  - ALUControl：控制信号生成

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
   input [1:0] ALUOp;
   input [5:0] FuncCode;
   output [3:0] reg ALUCtl;
   always case (FuncCode)
   32: ALUOp<=2; // add
   34: ALUOp<=6; // subtract
   36: ALUOP<=0; // and
   37: ALUOp<=1; // or
   39: ALUOp<=12; // nor
   42: ALUOp<=7; // slt
   default: ALUOp<=15; // should not happen
   endcase
endmodule
```

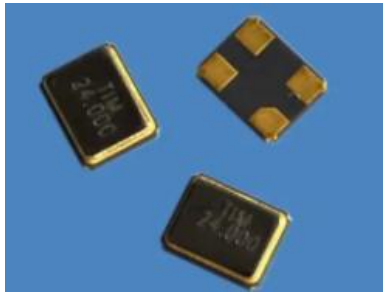
```
module RISCVALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [63:0] A,B;
   output reg [63:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
    end
endmodule
```

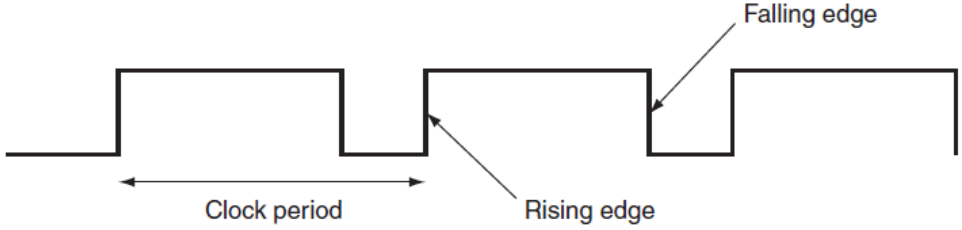
A.6 快速加法：超前进位

- 自学，略

A.7 时钟

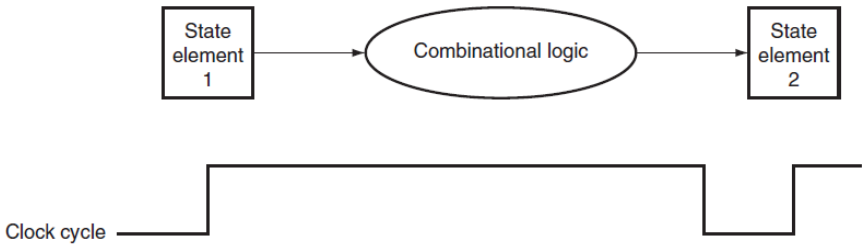
- 定义：是一个具有固定周期的不停翻转的信号
- A clock is simply a free-running signal with a fixed cycle time
- 与时钟相关的参数：周期、频率、占空比、驱动能力、抖动、偏移等
- 一般由专门的器件来生成：有源晶振、无源晶振、锁相环等
- 晶振介绍：
  - https://www.bilibili.com/video/BV1yS4y1k7EE?from=search&seid=10604632539372921070&spm_id_from=333.337.0.0





A.7 时钟

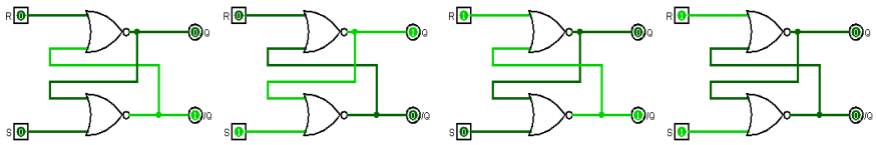
- 时序逻辑电路需要通过时钟信号控制何时更新存储元件的状态
  - 电平触发：锁存器
  - 边沿触发：触发器
- 同步系统：在时钟边沿同步更新存储元件状态的电路系统
  - 组合逻辑：（是否是同步系统？）
  - 时序逻辑：锁存器电路（？）、触发器电路（？）
- 边沿触发电路的优点
  - 同步，时序可控
  - 可以实现反馈（有什么用？）



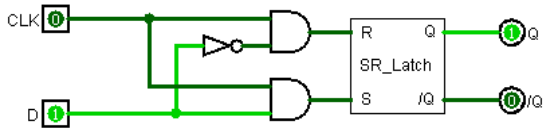
A.8 存储元件：触发器、锁存器和寄存器

- 基本逻辑门 → RS锁存器→ D锁存器→ D触发器→ 寄存器

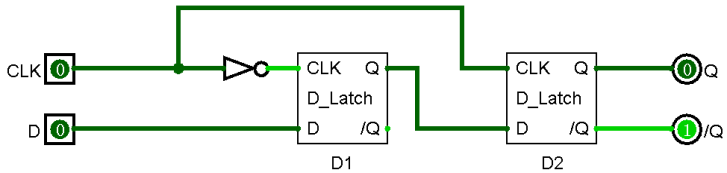
- RS锁存器



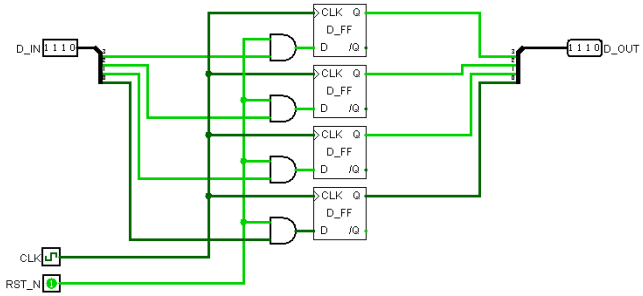
- D锁存器



- D触发器



- 寄存器

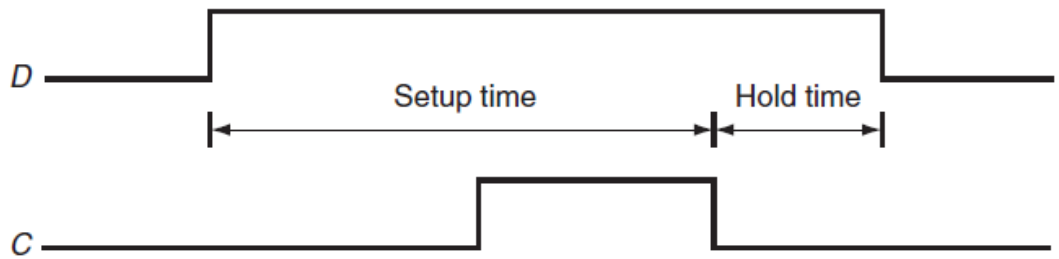


A.8 存储元件：触发器、锁存器和寄存器

- Verilog实现

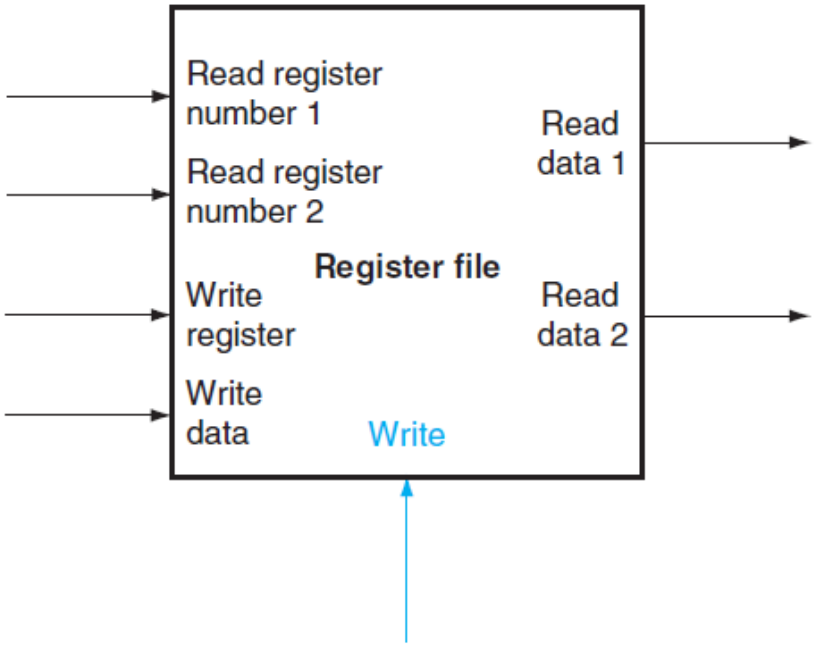
```
module DFF(clock,D,Q,Qbar);
    input clock, D;
    output reg Q;
    output Qbar;
    assign Qbar= ~ Q;
    always @(posedge clock)
       Q=D;
endmodule
```

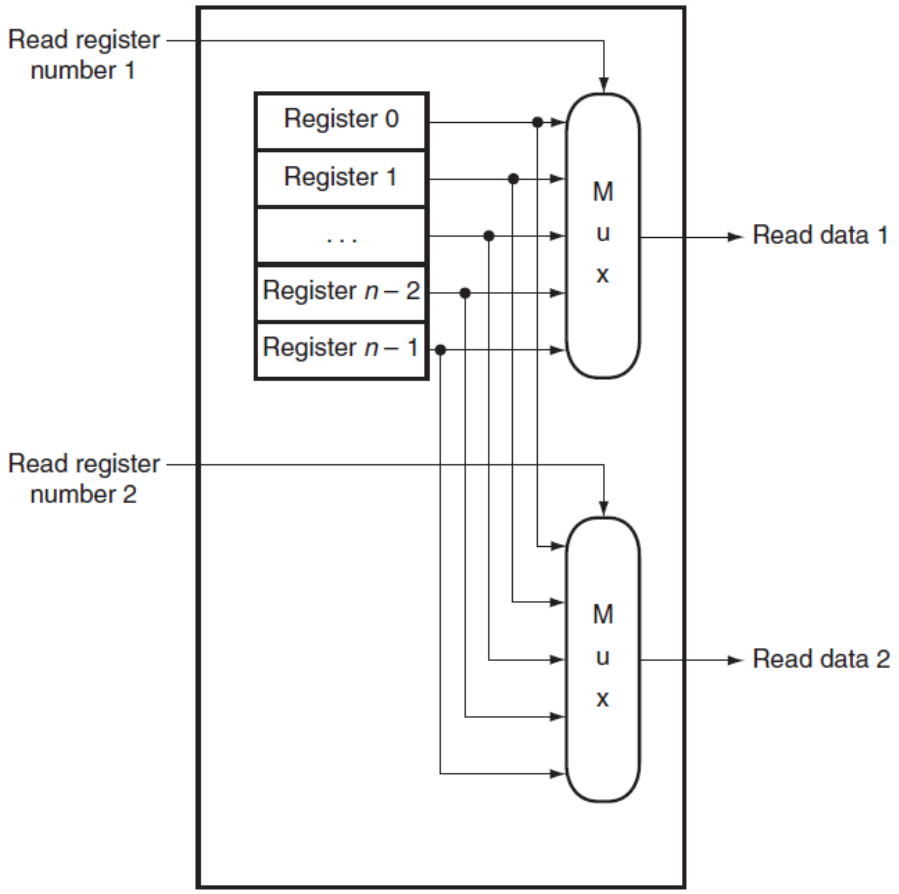
- 建立时间和保持时间



A.8 存储元件：触发器、锁存器和寄存器

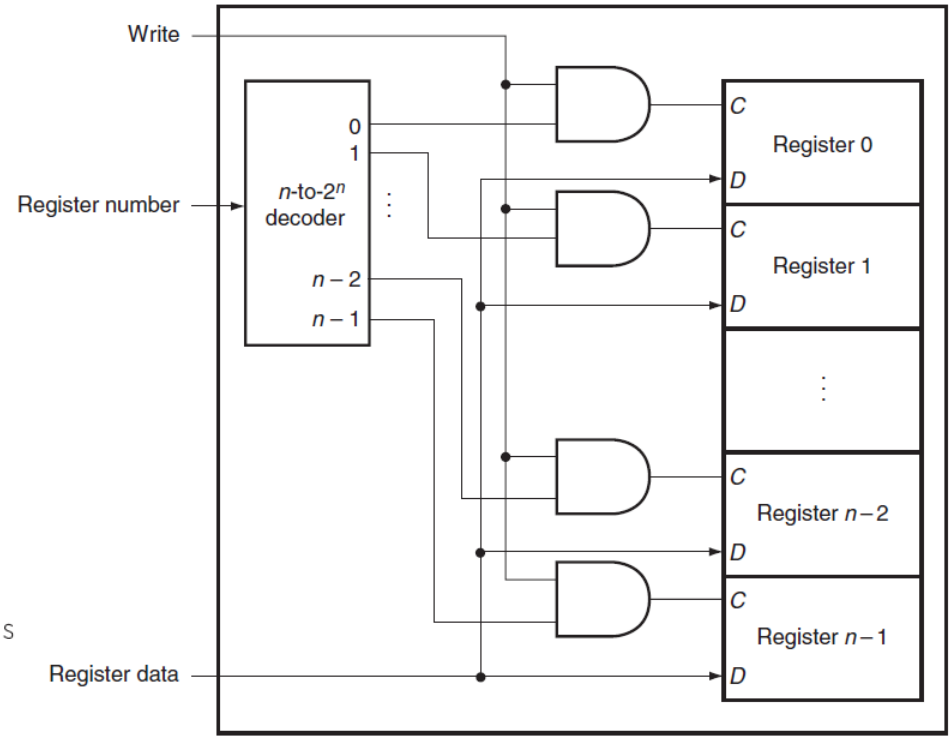
- 寄存器文件（Register File）
  - 在CPU数据通路中至关重要的结构
  - 由一组寄存器组成，可以通过寄存器编号进行读写操作
  - 一般有两组读端口和一组写端口





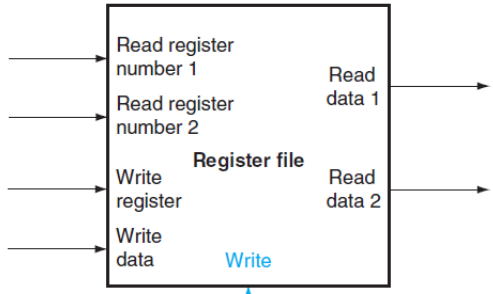
A.8 存储元件：触发器、锁存器和寄存器

- 寄存器文件
  - 写接口结构图



- Verilog实现

```
module registerfile (Read1,Read2,WriteReg,WriteData,RegWrite,
Data1,Data2,clock);

   input [5:0] Read1,Read2,WriteReg; // the register numbers
to read or write
   input [63:0] WriteData; // data to write
   input RegWrite, // the write control
     clock; // the clock to trigger write
   output [63:0] Data1, Data2; // the register values read
   reg [63:0] RF [31:0]; // 32 registers each 32 bits long

   assign Data1 = RF[Read1];
   assign Data2 = RF[Read2];

   always begin
      // write the register with new value if Regwrite is
high
      @(posedge clock) if (RegWrite) RF[WriteReg] <=
WriteData;
   end
endmodule
```



A.8 存储元件：触发器、锁存器和寄存器

Check Yourself

In the Verilog for the register file in Figure A.8.11, the output ports corresponding to the registers being read are assigned using a continuous assignment, but the register being written is assigned in an `always` block. Which of the following is the reason?

a. There is no special reason. It was simply convenient.

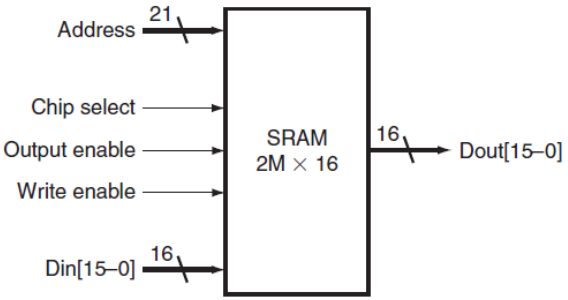
b. Because Data1 and Data2 are output ports and WriteData is an input port.

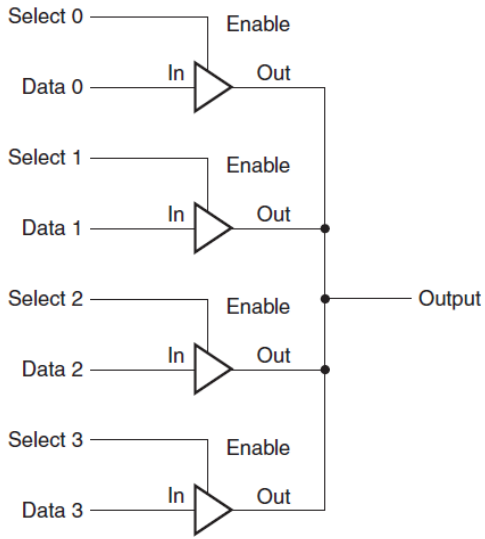
c. Because reading is a combinational event, while writing is a sequential event.

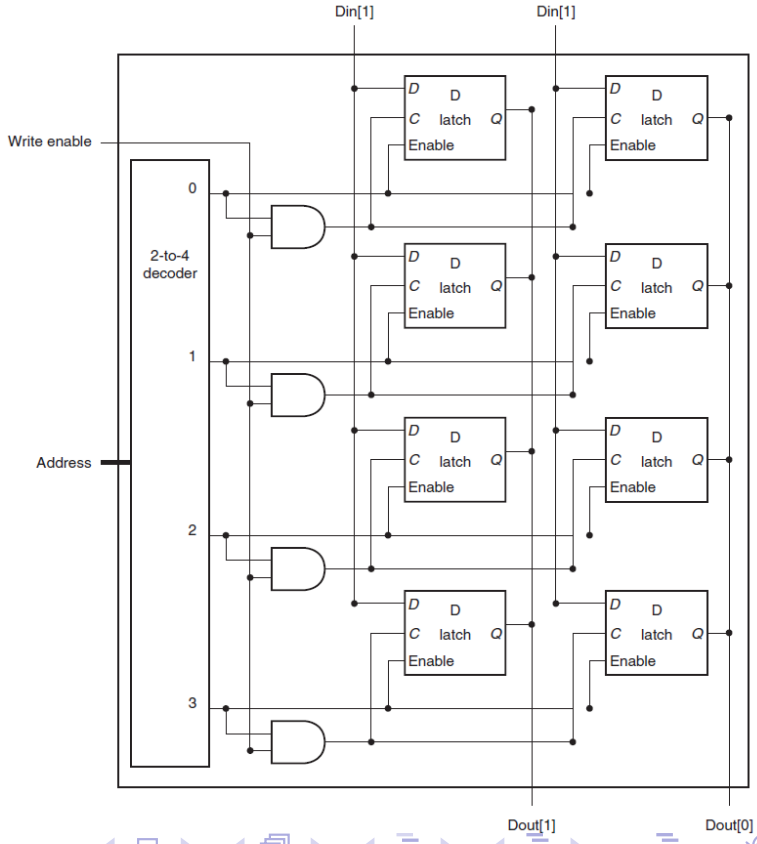
```
module registerfile (Read1,Read2,WriteReg,WriteData,RegWrite,
Data1,Data2,clock);

   input [5:0] Read1,Read2,WriteReg; // the register numbers
to read or write
   input [63:0] WriteData; // data to write
   input RegWrite, // the write control
     clock; // the clock to trigger write
   output [63:0] Data1, Data2; // the register values read
   reg [63:0] RF [31:0]; // 32 registers each 32 bits long

   assign Data1 = RF[Read1];
   assign Data2 = RF[Read2];

   always begin
      // write the register with new value if Regwrite is
high
      @(posedge clock) if (RegWrite) RF[WriteReg] <=
WriteData;
   end
endmodule
```

A.9 存储元件：SRAM和DRAM

- 存储元件
  - 寄存器和寄存器文件的缺点：结构复杂、成本高、集成度低
  - 大容量数据存储需要使用SRAM（静态）、DRAM（动态）
- SRAM：static random access memories
  - 巨型多路选择器不具备可行性
  - SRAM使用共享输出线的方式实现

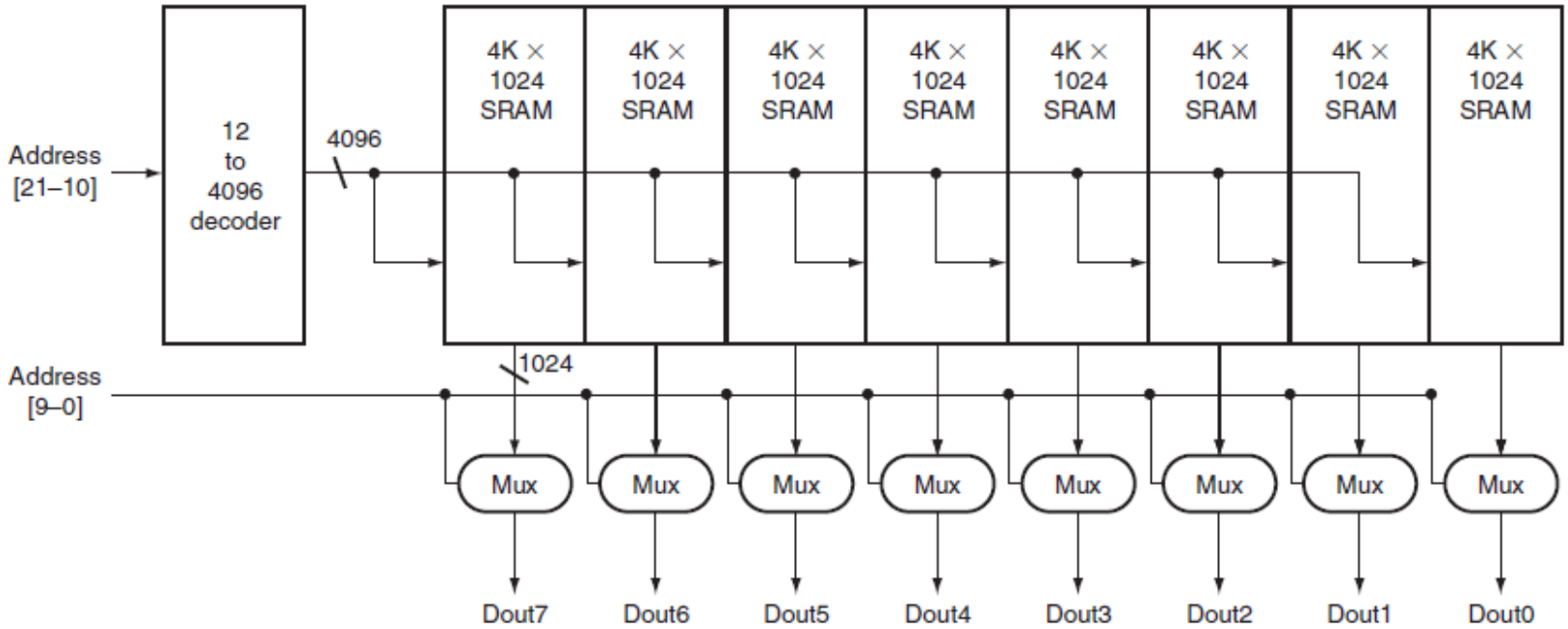






A.9 存储元件：SRAM和DRAM

- SRAM：static random access memories
  - 前面例子的设计中消除了巨型选择器的需求
  - 仍然需要大型译码器和大量字线
  - 改进：使用二级译码装置

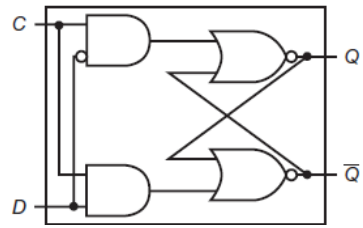


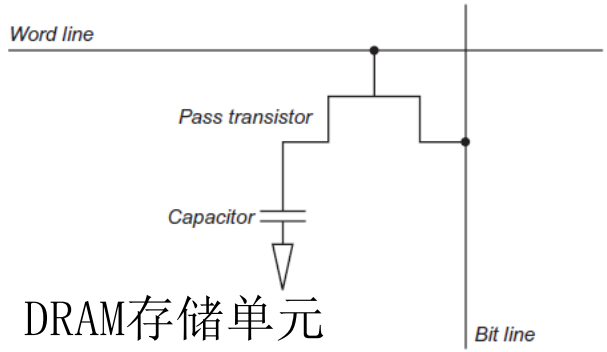
  - 同步SRAM：SSRAM，簇发传输，由时钟信号控制

A.9 存储元件：SRAM和DRAM

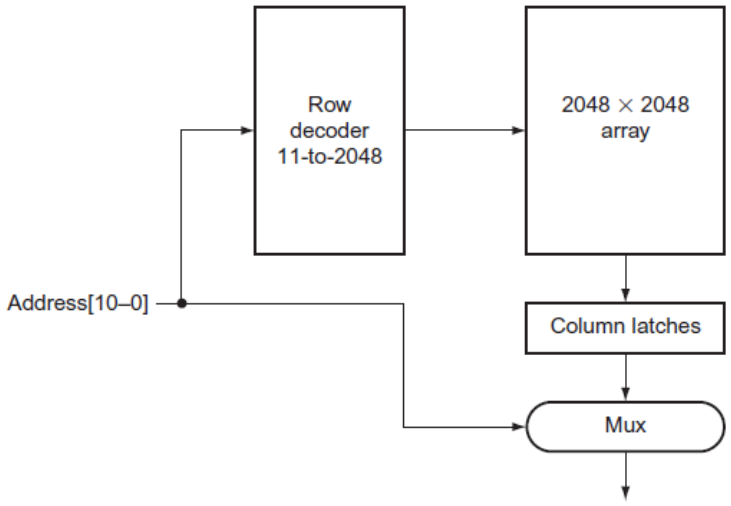
- DRAM：Dynamic Random Access Memory
  - 中文全程：动态随机访问存储器
  - SRAM数据保存：双稳态电路（4~6个晶体管/bit）
  - DRAM数据保存：电容(1个晶体管/bit)
  - 优点：结构简单、成本更低，集成度更高
  - 缺点：电容漏电，需要定时刷新（动态）
- SDRAM
  - 同步动态随机访问存储器
  - Synchronous DRAM
- DDR SDRAM
  - 双倍数据速率SDRAM
  - Double Data Rate SDRAM

SRAM存储单元





DRAM存储单元



A.9 存储元件：SRAM和DRAM

- 最新主流DDR SDRAM相关参数
  - 外形规格：双列直插式存储模块 Dual-Inline-Memory-Modules，简称 DIMM
  - 频率：时钟1600MHz，数据3200MHz
  - 容量：128G
  - 价格：约30元/GB（消费级）

现代海力士（SK hynix） DDR4 ECC RDIMM REG 工作站 服务器内存条 LMKJ 128G DDR4 3200 REG 服务器内存

海力士原厂DDR4-RDIMM-REG服务器内存条，不支持笔记本台式机！可兼容（联想-IBM-戴尔-惠普-浪潮-华为-思科-华3等服务器）

京东价 ¥ 7999.00 降价通知

优惠券 满100减5

累计评价 12

京东超市 金士顿（Kingston）128GB USB3.2 Gen 1 U盘 DTX 时尚设计 轻巧便携

【金士顿装机盛典】晒单赢50元E卡，会员专享百元券包，更有Kingston.Fury定制滑板等你来赢！请戳~查看>

甜蜜礼 心意之选，爱耀出色

京东价 ¥ 74.90 降价通知

¥ 69.90 粉丝价 关注店铺，即享粉丝价

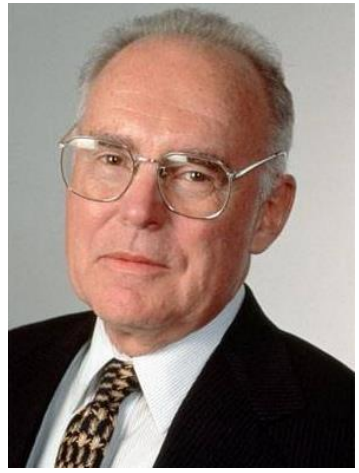
A.9 存储元件：SRAM和DRAM

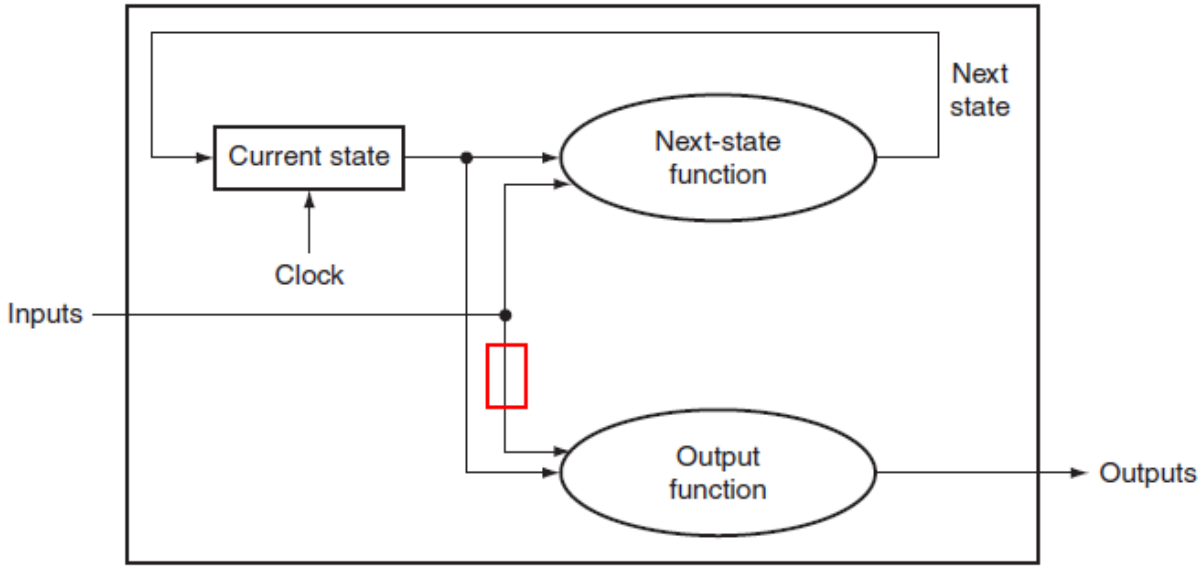
- 错误修正
  - 大容量存储器中存在数据损坏可能性
  - 使用校验码来进行检测或修正
  - 奇偶校验（奇校验、偶校验）：简单，只能检测奇数位的错误
  - ECC：error correction codes，又称汉明码/海明码/Hamming code
    - 根据校验位的数值，可以判定是否出错，以及出错位

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

A.10 有限状态机

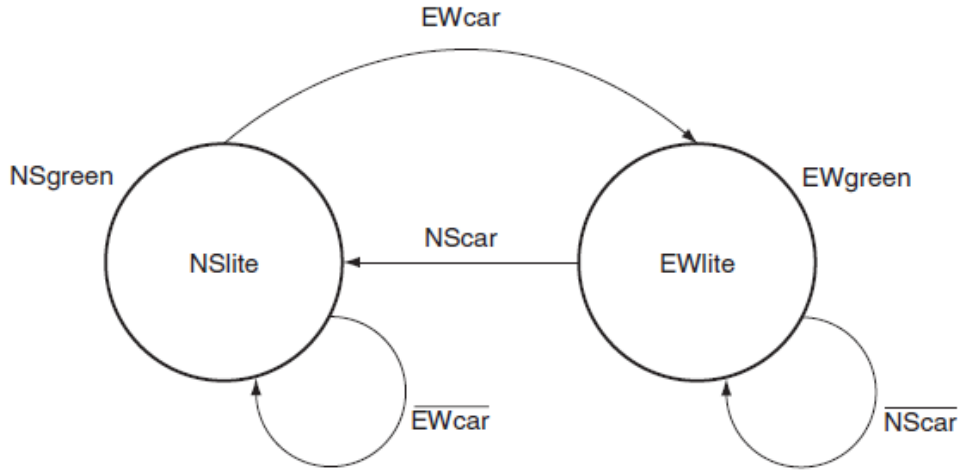
- 有限状态机：FSM，**F**inite-**S**tate **M**achines
- 一种时序逻辑函数，包括一组输入、输出、将当前状态和输出映射到新状态的下一状态函数，以及将当前状态和输入映射到一组有效输出的输出函数
  - 摩尔型：输出依赖于当前状态，时序更好
  - 米莉型：输出依赖于当前状态和当前输入，状态更少
- 是实现时序逻辑控制的重要手段





A.10 有限状态机

- 例：交通灯
  - 输入信号：
    - EWcar：东西方向有车
    - NScar：南北方向有车
  - 输出信号
    - EWlite：东西方向灯，1为绿灯，0为红灯
    - NSlite：南北方向灯，1为绿灯，0为红灯
  - 两个状态：
    - EWgreen：东西方向通行
    - NSgreen：南北方向通行
  - 要求：
    - 每30秒更新一次状态：跳转或保持
  - 建议：FSM采用三段式写法
    - 参考VerilogOJ平台第56题




```
module TrafficLite (EWCar,NSCar,EWLite,NSLite,clock);

   input EWCar, NSCar,clock;
output EWLite,NSLite;

reg state;

initial state=0;  //set initial state

//following two assignments set the output, which is based
only on the state variable
assign NSLite = ~ state; //NSLite on if state = 0;
assign EWLite = state; //EWLite on if state = 1

always @(posedge clock) // all state updates on a positive
clock edge
   case (state)
      0: state = EWCar; //change state only if EWCar

      1: state = ~ NSCar; // change state only if NSCar

   endcase
endmodule
```

A.11 定时方法

- 略

A.12 现场可编程设备

- 略

A.13 本章小结

- 略

计算机组成原理
CH1_计算机抽象及相关技术

卢建良
lujl@ustc.edu.cn
2022年春季学期

提纲

- 引言
- 计算机体系结构中的8个伟大思想
- 程序表象之下
- 箱盖后的硬件
- 处理器和存储制造技术
- 性能
- 功耗墙
- 沧海巨变：从单处理器向多处理器转变
- 实例：测评Intel Core i7
- 谬误与陷阱
- 本章小结

1.1 引言

- 图灵：战争英雄，同性恋、计算机科学之父
  - 艾伦·麦席森·图灵（英语：Alan Mathison Turing，1912年6月23日—1954年6月7日），英国数学家、逻辑学家，被称为计算机科学之父，人工智能之父。1931年图灵进入剑桥大学国王学院，毕业后到美国普林斯顿大学攻读博士学位，第二次世界大战爆发后回到剑桥，后曾协助军方破解德国的著名密码系统Enigma，帮助盟军取得了二战的胜利。
  - 1952年，英国政府对图灵的同性恋取向定罪，随后图灵接受化学阉割（雌激素注射）。1954年6月7日，图灵吃下含有氰化物的**苹果**中毒身亡，享年41岁。2013年12月24日，在英国司法大臣克里斯·格雷灵的要求下，英国女王伊丽莎白二世向图灵颁发了皇家赦免。
  - 图灵对于人工智能的发展有诸多贡献，提出了一种用于判定机器是否具有智能的试验方法，即图灵试验，每年都有试验的比赛。此外，图灵提出的著名的图灵机模型为现代计算机的逻辑工作方式奠定了基础。
  - https://www.bilibili.com/video/BV1tx411V7yQ/?spm_id_from=333.788.recommend_more_video.3

1.1 引言

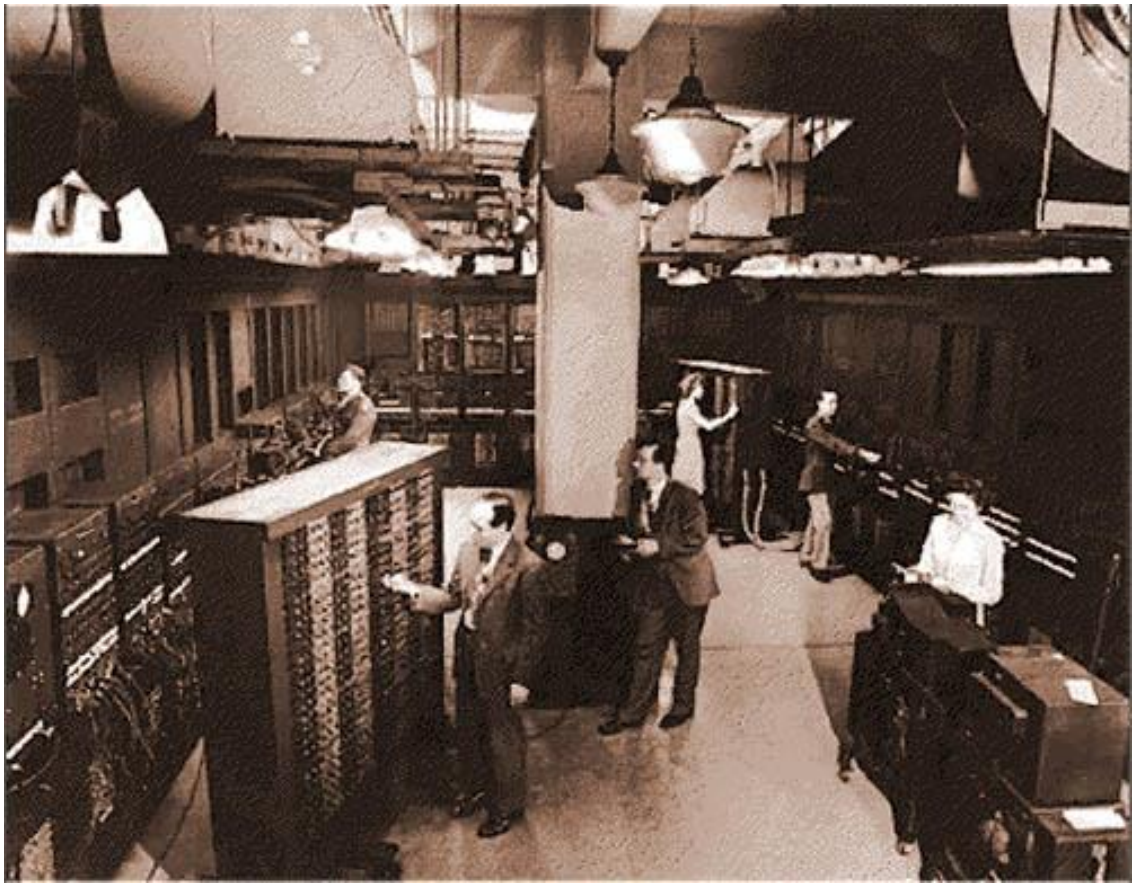
- 图灵机
  - 图灵机（Turing Machine）是图灵在1936年发表的 "On Computable Numbers, with an Application to the Entscheidungsproblem"（《论可计算数及其在判定性问题上的应用》）中提出的**数学模型**。既然是数学模型，它就并非一个实体概念，而是架空的一个想法。在文章中图灵描述了它是什么，并且证明了，**只要图灵机可以被实现，就可以用来解决任何可计算问题**
- 图灵完备
  - 图灵完备性（Turing Completeness）是针对一套数据操作规则而言的概念。数据操作规则可以是一门编程语言，也可以是计算机里具体实现了的指令集。当这套规则可以实现图灵机模型里的全部功能时，就称它具有图灵完备性。直白一点说，图灵完备性就是我给你一工具箱的东西，包括无限内存、if/else 控制流、while 循环……那么你现在图灵完备了吗?
- https://blog.csdn.net/a493823882/article/details/109149332
- https://www.bilibili.com/video/BV1br4y1N762/?spm_id_from=333.788.recommend_more_video.5

1.1 引言

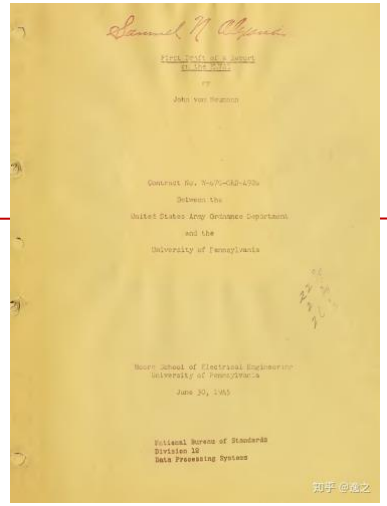
- ABC ,1942年，第一台电子计算机
  - Atanasoff–Berry computer
  - 阿塔纳索夫-贝里计算机
  - 爱荷华州立大学
  - 用于求解线性方程组
  - 最多支持29个方程
  - 非图灵完备

1.1 引言

- ENIAC ,1946年，第一台通用电子计算机
  - Electronic Numerical Integrator and Computer
  - 电子数字积分器和计算机
  - 宾夕法尼亚大学莫尔学院
  - 成本：100万美元
  - 功耗：150kw/h
  - 占地：170m^2
  - 运算速度：5000次/秒
  - 不可编程
  - 十进制并行计算
  - 无程序存储功能



1.1 引言

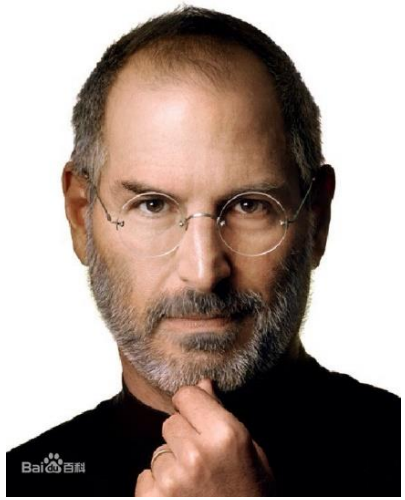


- EDVAC,1944~1952,
  - Electronic Discrete Variable Automatic Computer
  - 电子离散变量自动计算机
  - 1MHz
  - 二进制
  - 串行
  - 存储程序
  - 冯·诺依曼架构
  - EDVAC报告书的第一份草案

- ✓ 机器内部使用二进制表示数据；
- ✓ 像存储数据一样存储程序；
- ✓ 计算机由运算器、控制器、存储器、输入模块和输出模块5部分组成

1.1 引言

- iphone13 pro max,2021
  - 主频：3GHz
  - 尺寸：16cm*8cm*0.8cm
  - 功耗：1~4W
  - 成本：2000美元
  - 性能：每秒15.8万亿次计算
  - 集成度：150亿晶体管



Apple iPhone 13 Pro Max (A2644) 1TB 远峰蓝色 支持移动联通电信5G 双卡双待手机

自适应高刷新率，画面更流畅、响应更灵敏，电影效果模式随手拍大片！选购[快充套装]加99元得20W快充头！更多

京 东 价 ¥ 12999.00 降价通知

促 销 赠品 × 1 × 1 × 1

¥9799.00

Apple iPhone 13 Pro Max (A2644) 256GB 远峰蓝色 支持移动联通电信5G 双卡双待

¥11399.00

Apple iPhone 13 Pro Max (A2644) 512GB 远峰蓝色 支持移动联通电信5G 双卡双待

¥219.00

京东超市 闪迪(SanDisk)256GB USB3.0 U盘 CZ73酷铄 银色 读速150MB/s 金属外

1.1 引言

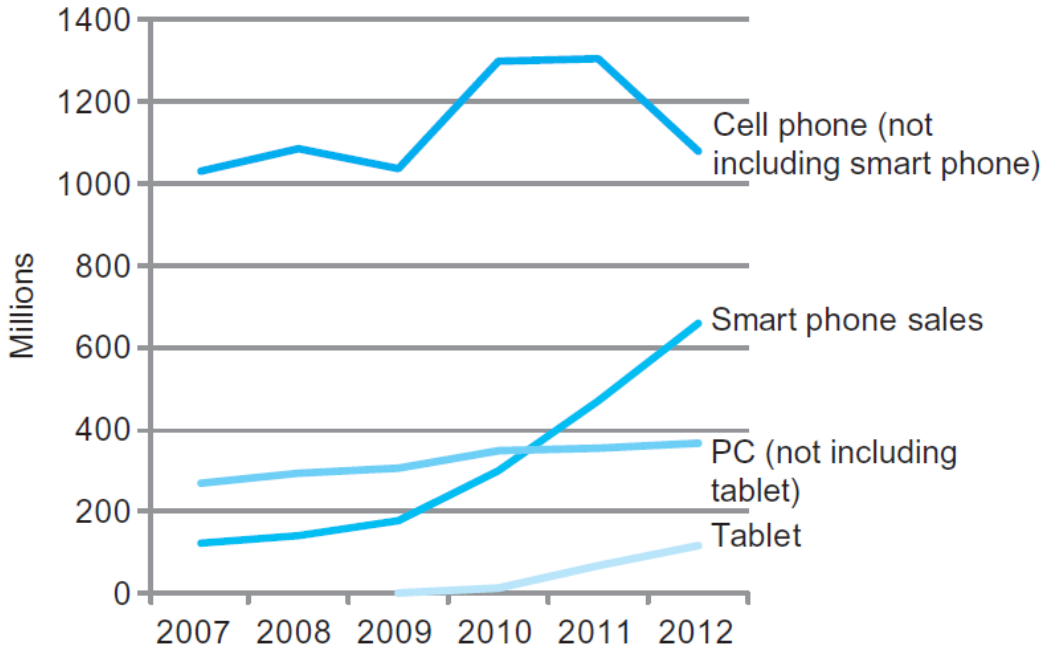
- 常用规格术语
  - 二进制 vs 十进制

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

FIGURE 1.1 The 2^X vs. 10^Y bytes ambiguity was resolved by adding a binary notation for all the common size terms. In the last column we note how much larger the binary term is than its corresponding decimal term, which is compounded as we head down the chart. These prefixes work for bits as well as bytes, so *gigabit* (Gb) is 10^9 bits while *gibibits* (Gib) is 2^{30} bits.

1.1 引言

- 传统计算机分类
  - 个人计算机（PC）
  - 服务器
  - 嵌入式计算机
- 后PC时代
  - 个人移动设备
  - 云计算
    - 软件即服务 SaaS
    - Software as a Service



1.1 引言

- 您从本课程中学到什么
  - How programs are translated into the machine language
    - And how the hardware executes them
  - The hardware/software interface
  - What determines program performance
    - And how it can be improved
  - How hardware designers improve performance
  - What is parallel processing

1.1 引言

Understanding Performance

Hardware or software component	How this component affects performance	Where is this topic covered?
Algorithm	Determines both the number of source-level statements and the number of I/O operations executed	Other books!
Programming language, compiler, and architecture	Determines the number of computer instructions for each source-level statement	Chapters 2 and 3
Processor and memory system	Determines how fast instructions can be executed	Chapters 4, 5, and 6
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed	Chapters 4, 5, and 6

1.1 引言

Check Yourself

Check Yourself sections are designed to help readers assess whether they comprehend the major concepts introduced in a chapter and understand the implications of those concepts. Some *Check Yourself* questions have simple answers; others are for discussion among a group. Answers to the specific questions can be found at the end of the chapter. *Check Yourself* questions appear only at the end of a section, making it easy to skip them if you are sure you understand the material.

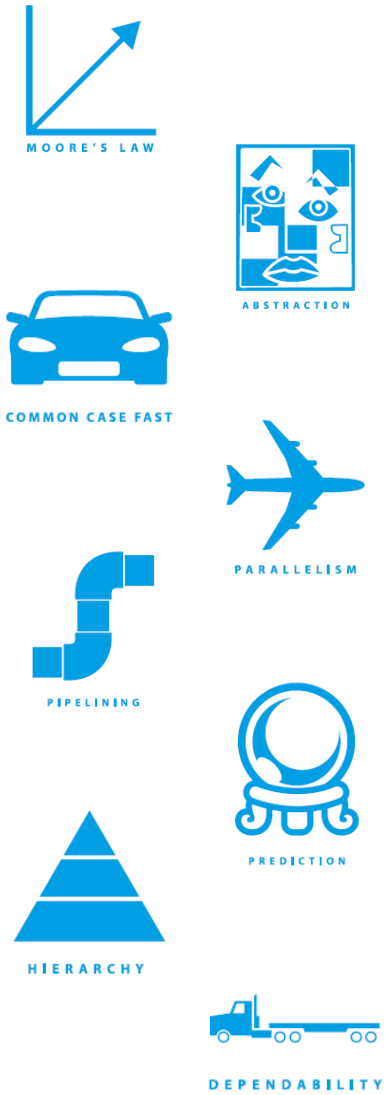
1. The number of embedded processors sold every year greatly outnumbers the number of PC and even post-PC processors. Can you confirm or deny this insight based on your own experience? Try to count the number of embedded processors in your home. How does it compare with the number of conventional computers in your home?

2. As mentioned earlier, both the software and hardware affect the performance of a program. Can you think of examples where each of the following is the right place to look for a performance bottleneck?

   - The algorithm chosen
   - The programming language or compiler
   - The operating system
   - The processor
   - The I/O system and devices

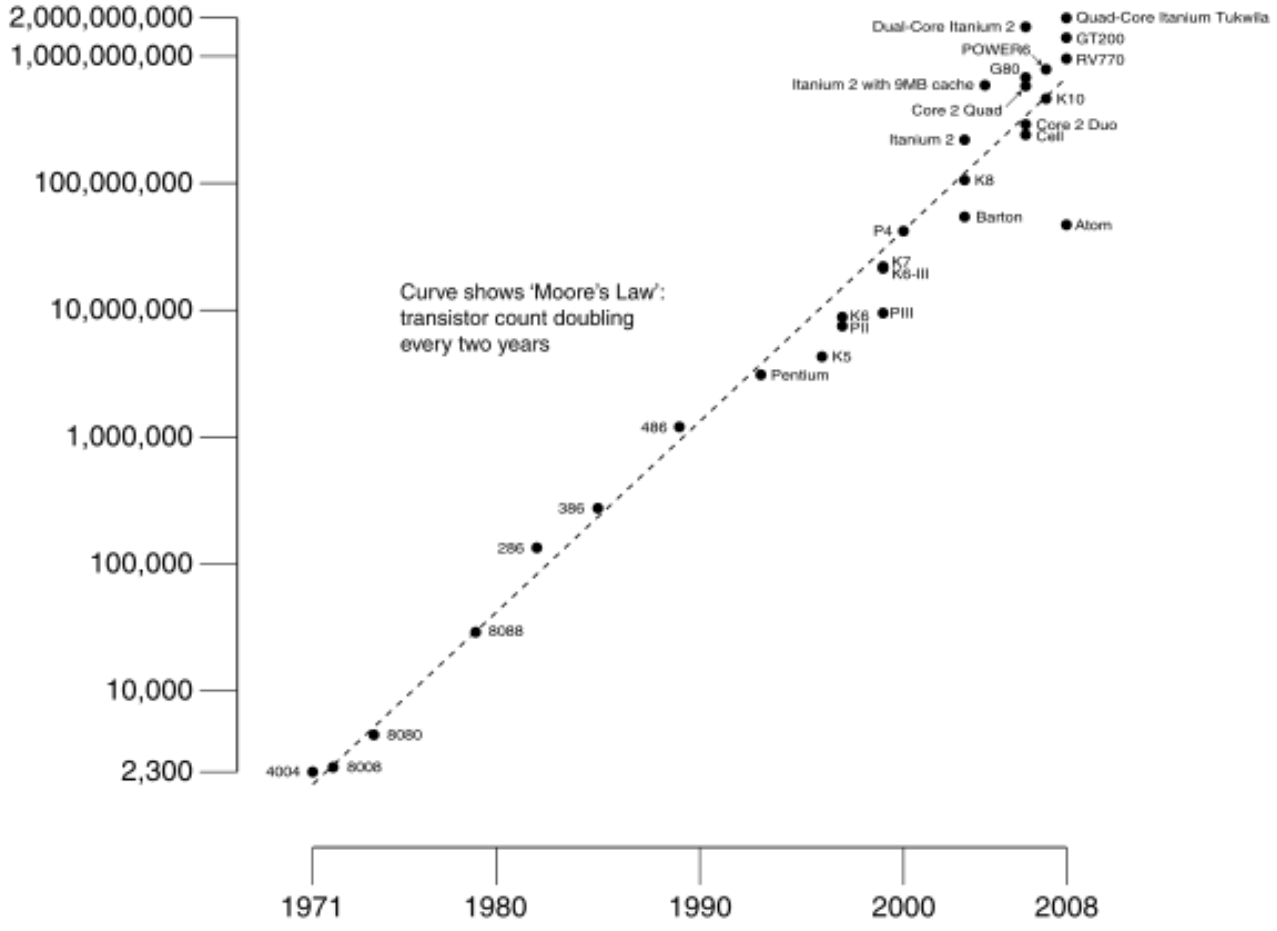
1.2 计算机体系结构中的8个伟大思想

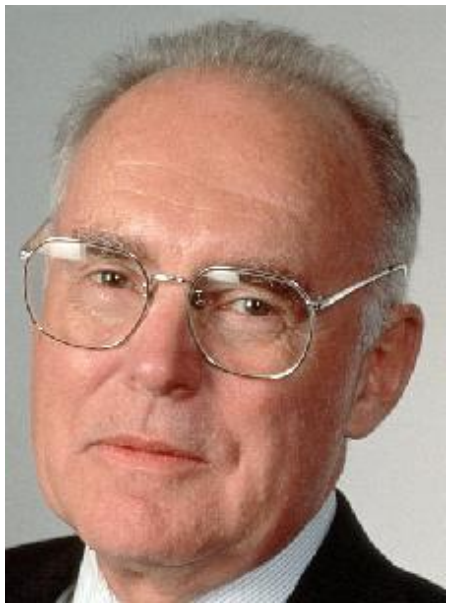
- Design for ***Moore’s Law***
- Use ***abstraction*** to simplify design
- Make the ***common case fast***
- Performance *via* ***parallelism***
- Performance *via* ***pipelining***
- Performance *via* ***prediction***
- ***Hierarchy*** of memories
- ***Dependability*** *via* redundancy



1.2 计算机体系结构中的8个伟大思想

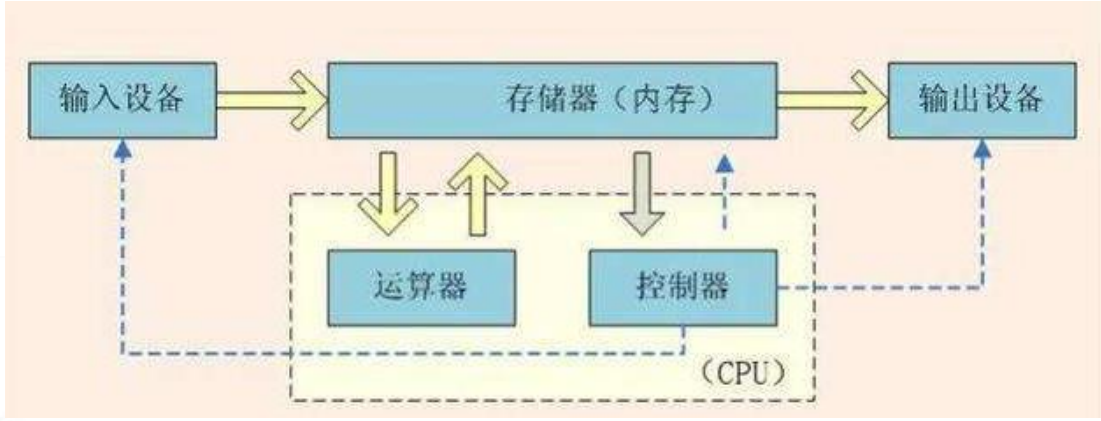
- 1 面向摩尔定律
  - 英特尔创始人之一戈登·摩尔的经验之谈，其核心内容为：集成电路上可以容纳的晶体管数目在大约每经过18个月便会增加一倍

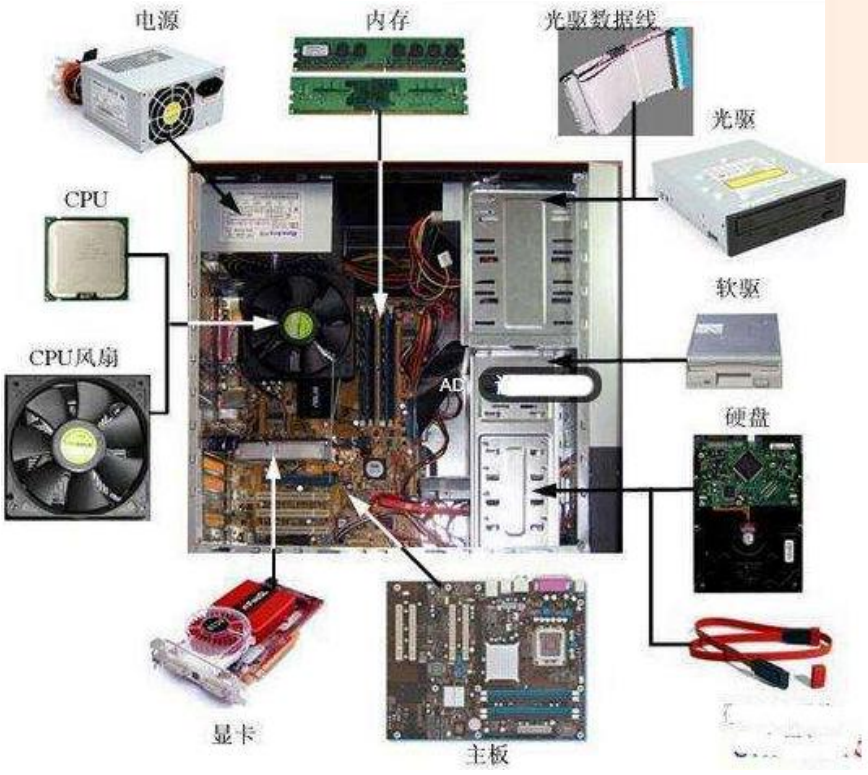




1.2 计算机体系结构中的8个伟大思想

- 2 使用抽象简化设计
  - 隐藏低层次细节，为高层次提供更简单的模型





1.2 计算机体系结构中的8个伟大思想

- 3 加速经常性事件
  - 通过实验及测量确定经常性事件

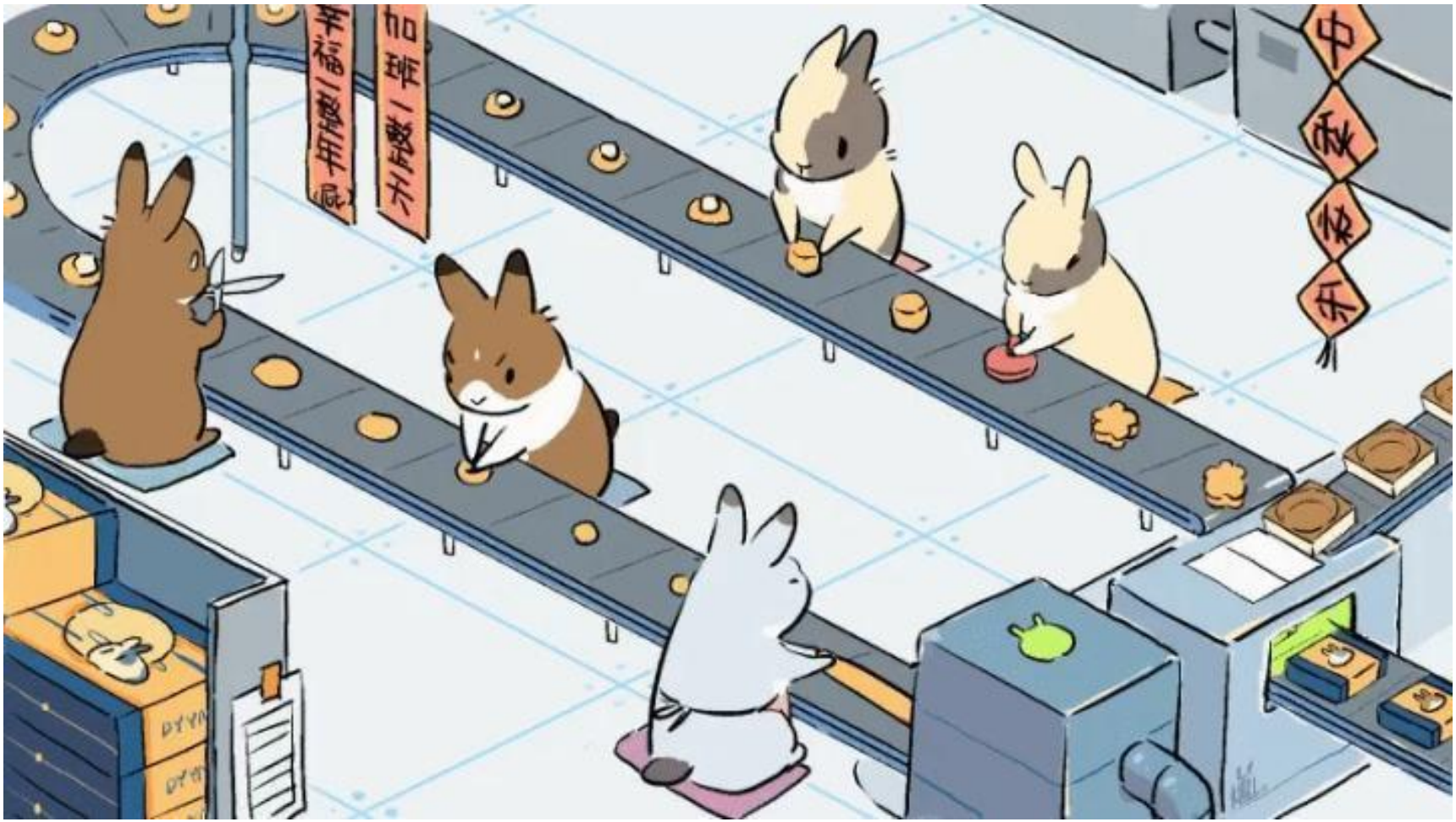
  - 宰我问：“三年之丧，期已久矣！君子三年不为礼，礼必坏；三年不为乐，乐必崩。旧谷既没，新谷既升，钻燧改火，期可已矣。”子曰：“食夫稻，衣夫锦，于女安乎？”曰：“安！”“女安则为之！夫君子之居丧，食旨不甘，闻乐不乐，居处不安，故不为也。今女安，则为之！”
  - 宰我出，子曰：“予之不仁也！子生三年，然后免于父母之怀。夫三年之丧，天下之通丧也，予也有三年之爱于其父母乎！”

1.2 计算机体系结构中的8个伟大思想

- 4 通过并行提高性能

1.2 计算机体系结构中的8个伟大思想

- 5 通过流水线提高性能



1.2 计算机体系结构中的8个伟大思想

- 6 通过预测提高性能

1.2 计算机体系结构中的8个伟大思想

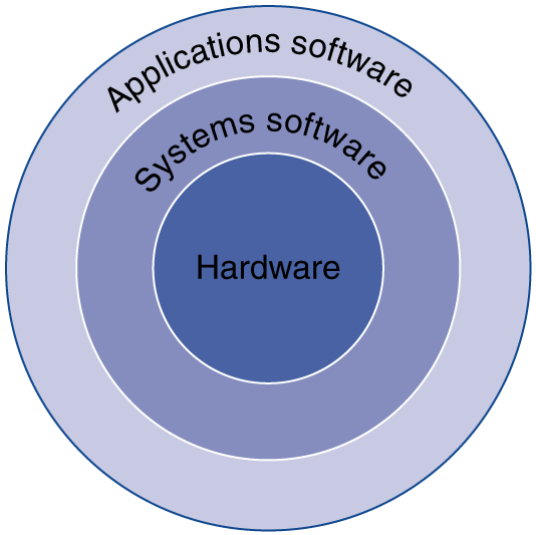
- 7 存储层次

1.2 计算机体系结构中的8个伟大思想

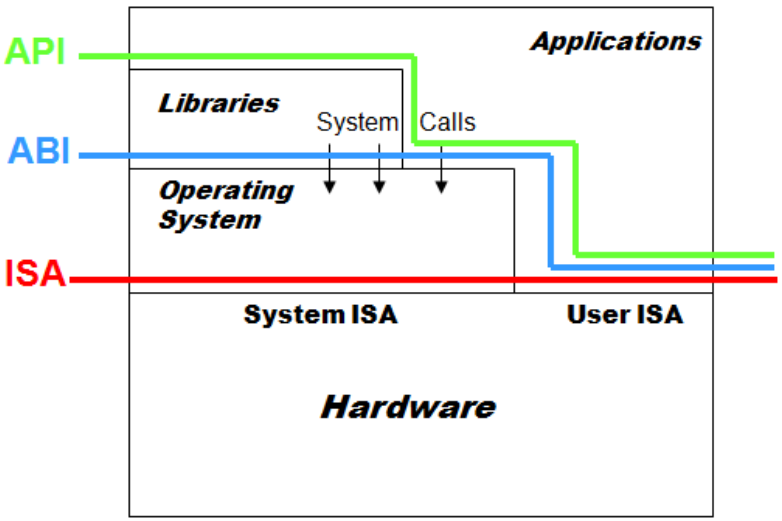
- 8 通过冗余提高可靠性

1.3 程序表象之下

- 应用软件
- 系统软件
- 硬件



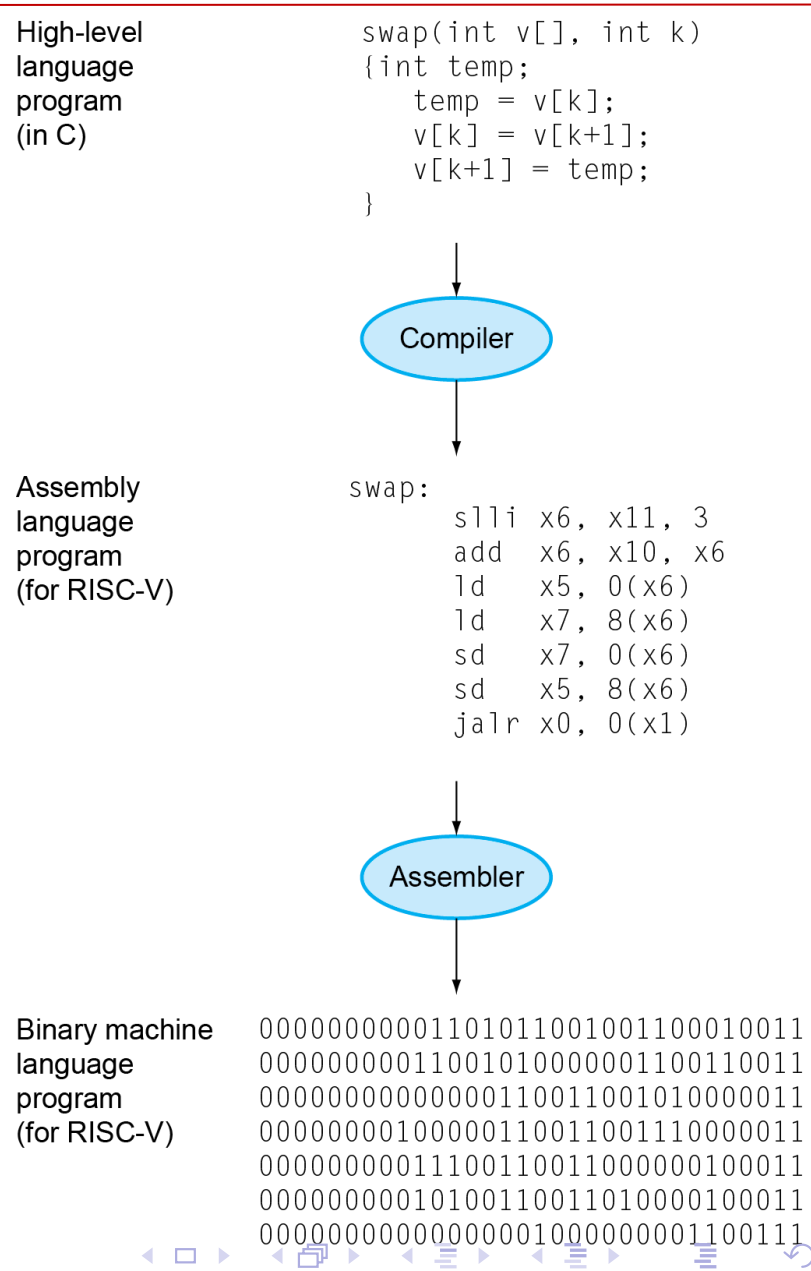
- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers



- API – application programming interface
- ABI – application binary interface
- ISA – instruction set architecture

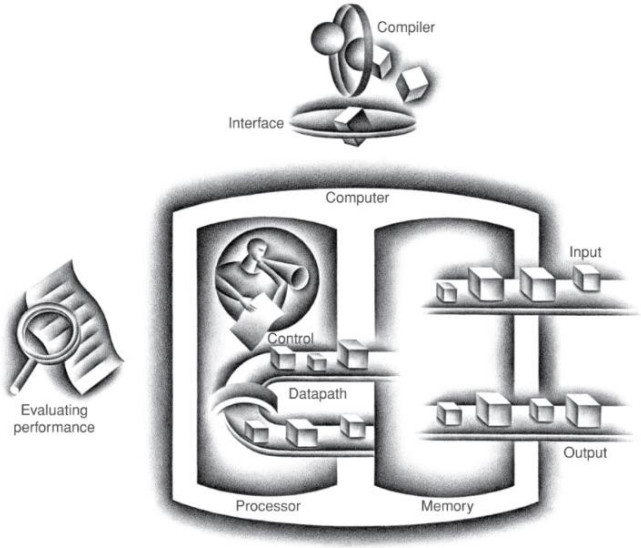
1.3 程序表象之下

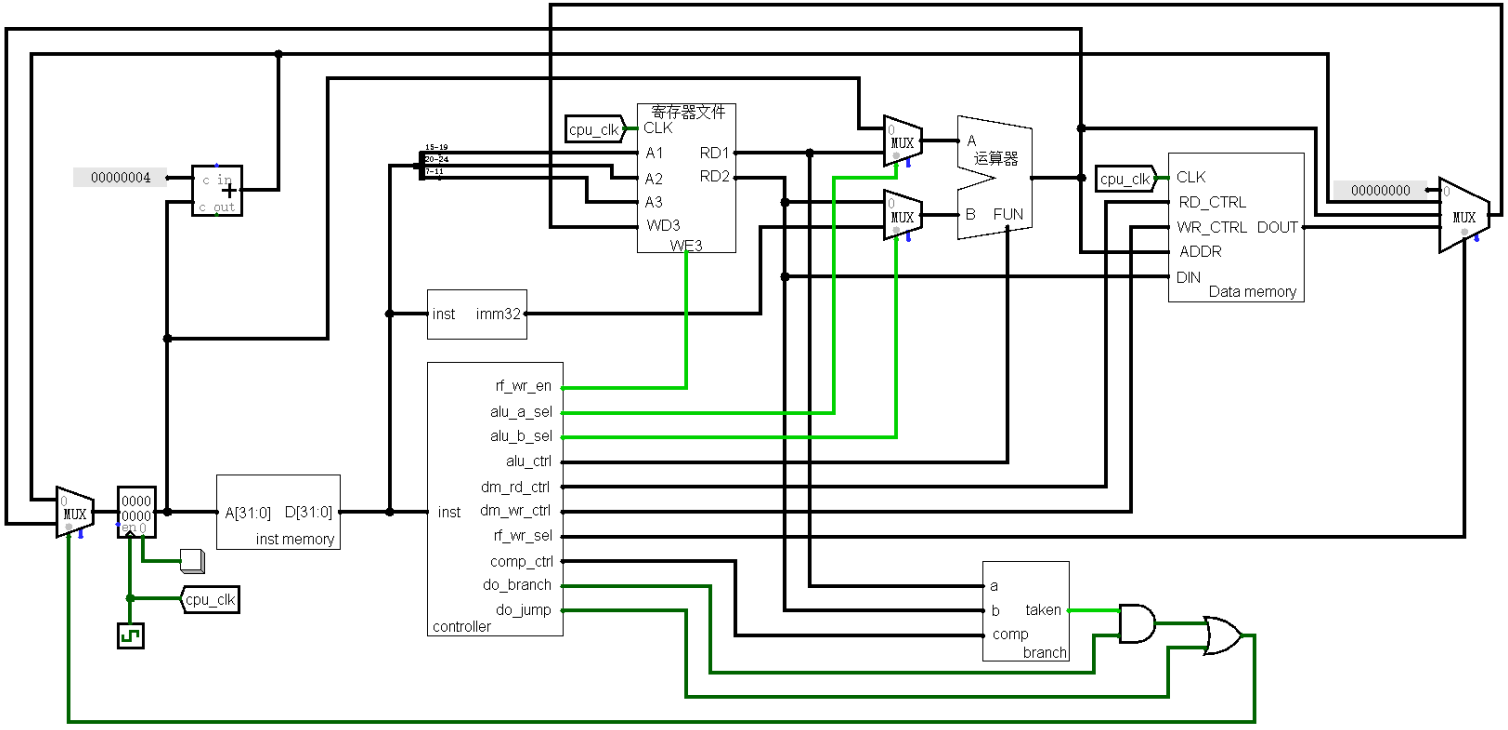
- 从高级语言到硬件语言
  - High-level language
    - Level of abstraction closer to problem domain
    - Provides for productivity and portability
  - Assembly language
    - Textual representation of instructions
  - Hardware representation
    - Binary digits (bits)
    - Encoded instructions and data



1.4 箱盖后的硬件

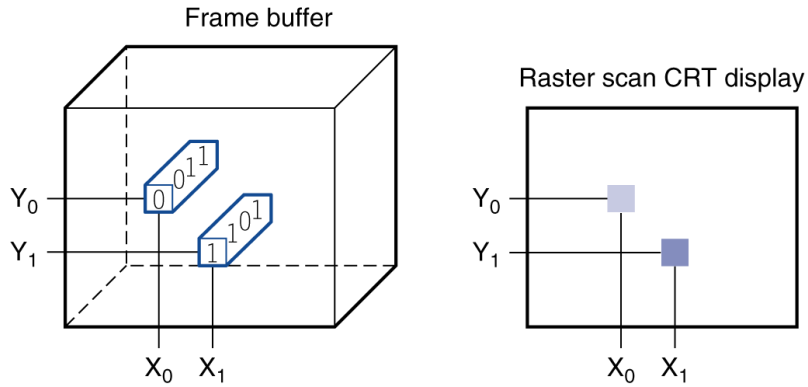
- 输入设备：鼠标、键盘
- 输出设备：显示器
- 存储器：硬盘、内存
- 控制器：处理器的一部分
- 运算器：处理器的另一部分





1.4 箱盖后的硬件

- 显示器
  - LCD：Liquid Crystal Display
  - 显示器的工作原理，2’43”
  - https://www.bilibili.com/video/BV12s411a76M?from=search&seid=12997120343234421829&spm_id_from=333.337.0.0
  - LCD屏幕的工作原理是怎样的，7’20”
  - https://www.bilibili.com/video/BV1Sb411W7Zf?from=search&seid=12997120343234421829&spm_id_from=333.337.0.0
  - 用高速摄像和微距镜头探索电视工作原理，11’38”
  - https://www.bilibili.com/video/BV11W411H7Bo/?spm_id_from=autoNext

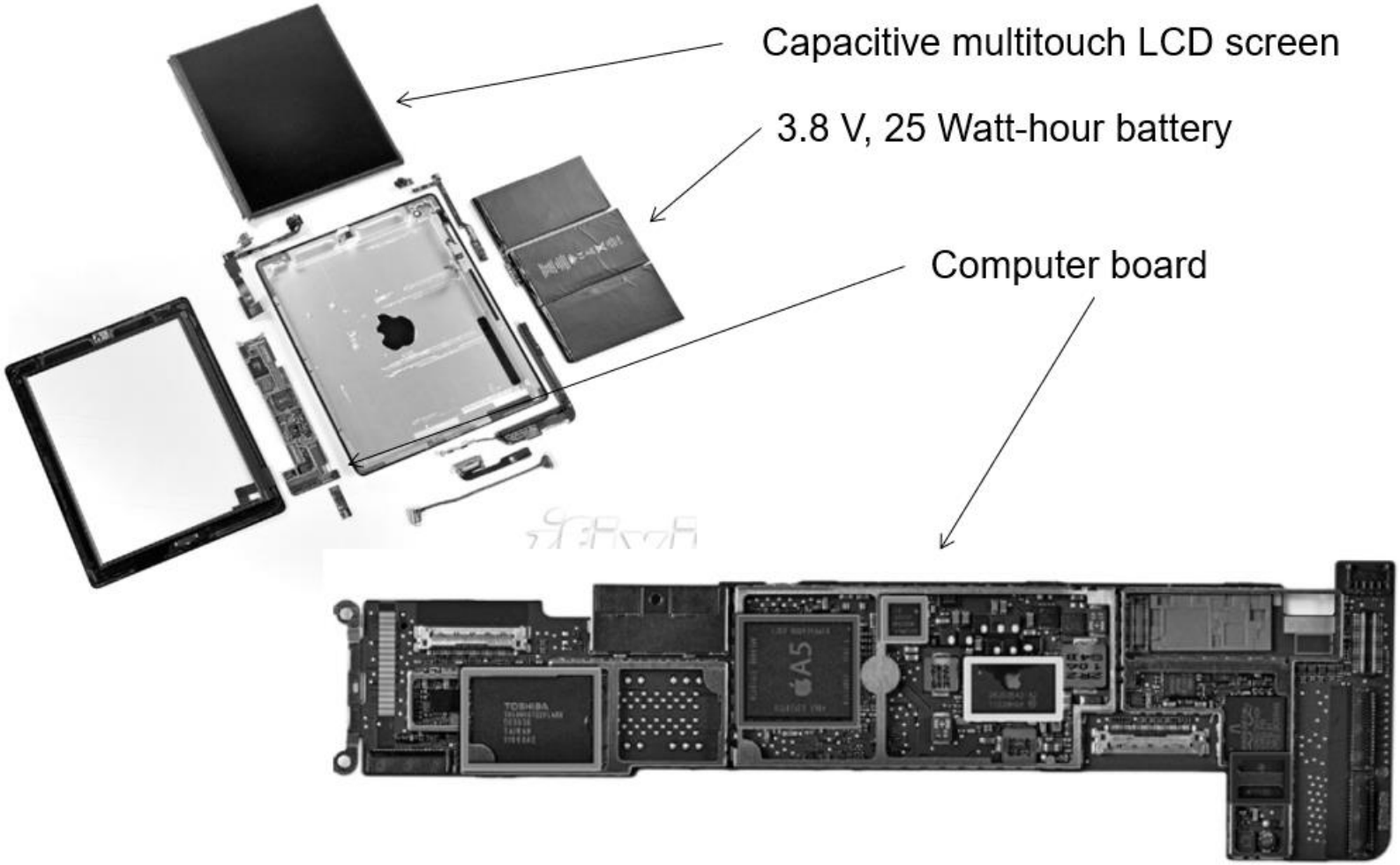


1.4 箱盖后的硬件

- 触摸屏
  - 触摸屏原理 手机电容式触摸屏显示器是如何工作的,8’ 36”
  - https://www.bilibili.com/video/BV1ub4y1n77d?from=search&seid=25256951038393402 89&spm_id_from=333.337.0.0
- 多点触控
  - 触摸屏多点触控原理，5’ 57”
  - https://www.bilibili.com/video/BV14T4y127A1/?spm_id_from=333.788.recommend_more_video.-1

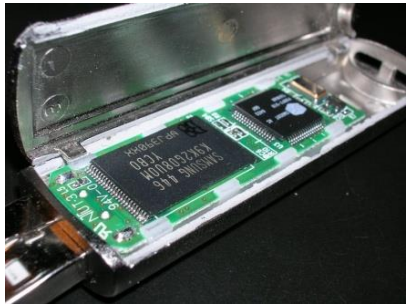
1.4 箱盖后的硬件

- 打开机箱（机壳）



1.4 箱盖后的硬件

- 数据安全
  - 易失性存储器：掉电后数据丢失
    - 寄存器、缓存、内存等
  - 非易失性存储器：掉电后数据不丢失
    - 硬盘、Flash、光盘等



1.4 箱盖后的硬件

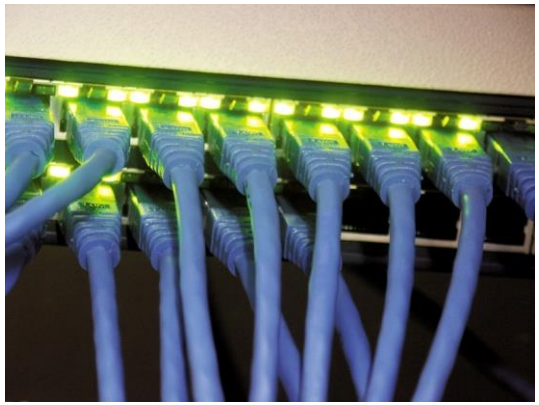
Check Yourself

- Semiconductor DRAM memory, flash memory, and disk storage differ significantly. For each technology, list its volatility, approximate relative access time, and approximate relative cost compared to DRAM.

1.4 箱盖后的硬件

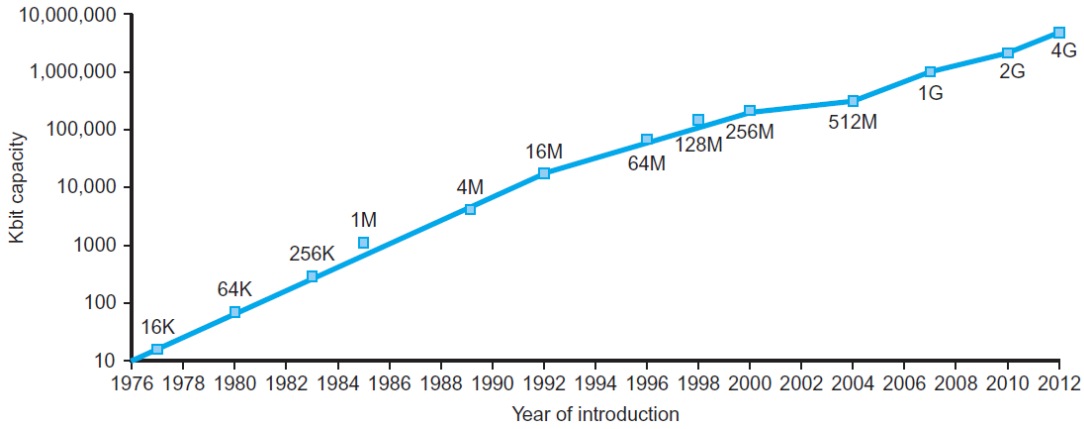
- 网络
  - Communication, resource sharing, nonlocal access
  - Local area network (LAN): Ethernet
  - Wide area network (WAN): the Internet
  - Wireless network: WiFi, Bluetooth

- 带宽
  - 永远不要忽略一辆载满磁带的在高速公路上飞驰的卡车的带宽
  - https://www.zhihu.com/question/20548494/answer/989693899?utm_source=qq



1.5 处理器和存储制造技术

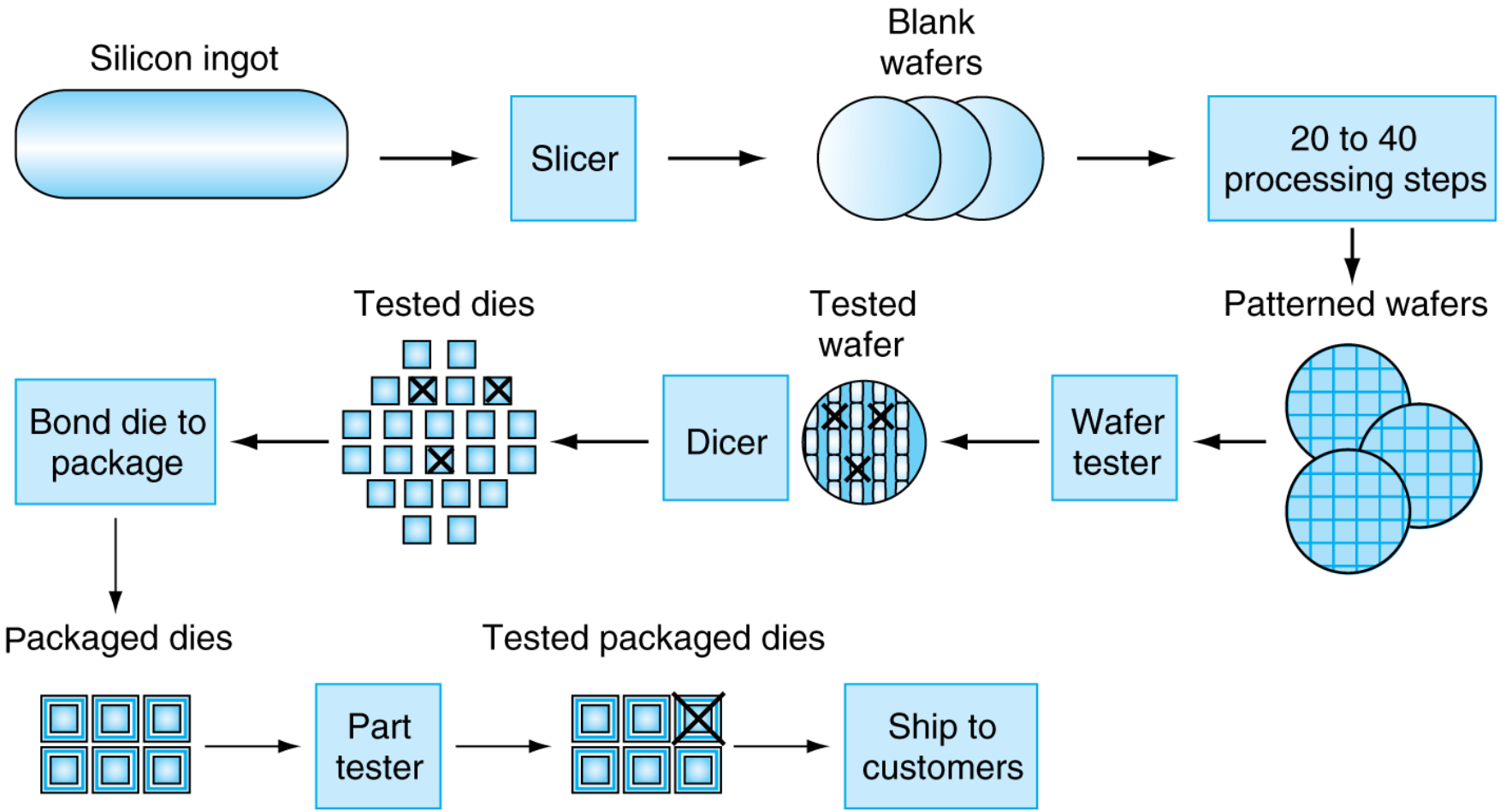
- 硅：Silicon，化学符号Si
- 是一种半导体
- 通过掺杂，可以转变为
  - 导体
  - 绝缘体
  - 开关（可控的导通与截止）



Year	Technology	Relative performance/cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit (IC)	900
1995	Very large scale IC (VLSI)	2,400,000
2013	Ultra large scale IC	250,000,000,000

1.5 处理器和存储制造技术

- ingot → wafer → die → chip
- 良率：yield，合格芯片数占总芯片数的百分比



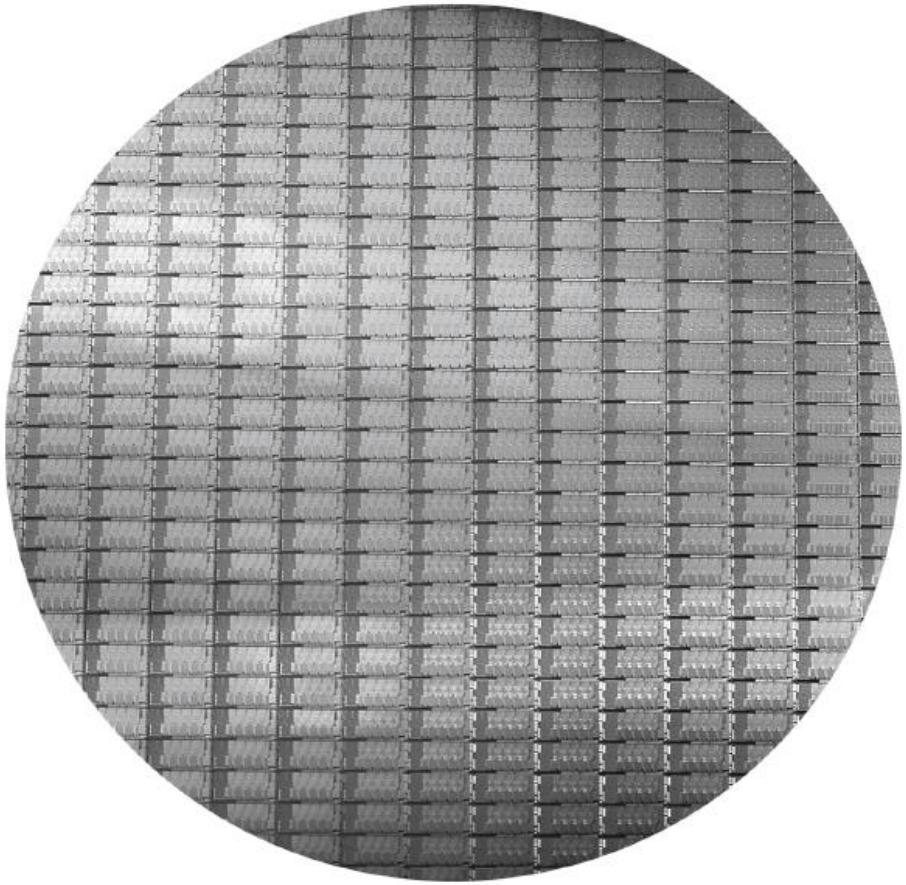
1.5 处理器和存储制造技术

- 碳基 vs 硅基

	ⅠA 1																0 18
1	1 H 氢 1.008	ⅡA 2											ⅢA 13	ⅣA 14	ⅤA 15	ⅥA 16	ⅦA 17	2 He 氦 4.003
2	3 Li 锂 6.941	4 Be 铍 9.012											5 B 硼 10.81	6 C 碳 12.01	7 N 氮 14.01	8 O 氧 16.00	9 F 氟 19.00	10 Ne 氖 20.18
3	11 Na 钠 22.99	12 Mg 镁 24	ⅢB 3	ⅣB 4	ⅤB 5	ⅥB 6	ⅦB 7	Ⅷ 8	Ⅷ 9	Ⅷ 10	ⅠB 11	ⅡB 12	13 Al 铝 26.98	14 Si 硅 28.09	15 P 磷 30.97	16 S 硫 32.06	17 Cl 氯 35.45	18 Ar 氩 39.95
4	19 K 钾 39.10	20 Ca 钙 40,08	21 Sc 钪 44.96	22 Ti 钛 47.87	23 V 钒 50.94	24 Cr 铬 52.00	25 Mn 锰 54.94	26 Fe 铁 55.85	27 Co 钴 58.93	28 Ni 镍 58.69	29 Cu 铜 63.55	30 Zn 锌 65.41	31 Ga 镓 69.72	32 Ge 锗 72.64	33 As 砷 74.92	34 Se 硒 78.96	35 Br 溴 79.90	36 Kr 氪 83.80
5	37 Rb 铷 85.47	38 Sr 锶 87.62	39 Y 钇 88.91	40 Zr 锆 91.2	41 Nb 铌 92.91	42 Mo 钼 95.94	43 Tc 锝 98	44 Ru 钌 101.1	45 Rh 铑 102.9	46 Pd 钯 106.4	47 Ag 银 107.9	48 Cd 镉 112.4	49 In 铟 114.8	50 Sn 锡 118.7	51 Sb 锑 121.8	52 Te 碲 127.6	53 I 碘 126.9	54 Xe 氙 131.3
6	55 Cs 铯 132.9	56 Ba 钡 137.3	57-71 La-Lu 镧系	72 Hf 铪 178.	73 Ta 钽 180.9	74 W 钨 183.8	75 Re 铼 186.2	76 Os 锇 190.2	77 Ir 铱 192.2	78 Pt 铂 195.1	79 Au 金 197.0	80 Hg 汞 200.6	81 Tl 铊 204.4	82 Pb 铅 207.2	83 Bi 铋 209.0	84 Po 钋 209	85 At 砹 210	86 Rn 氡 222

1.5 处理器和存储制造技术

- Intel Core i7晶圆
  - 300mm晶圆，280chip，32nm工艺
  - chip尺寸：20.7 * 10.5mm



1.5 处理器和存储制造技术

- 集成电路的成本
  - 与面积和缺陷率并不呈线性关系
- 公式1：直接导出
- 公式2：近似
- 公式3：经验公式

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$$

$$\text{Dies per wafer} \approx \text{Wafer area} / \text{Die area}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

1.5 处理器和存储制造技术

Check Yourself A key factor in determining the cost of an integrated circuit is volume. Which of the following are reasons why a chip made in high volume should cost less?

1. With high volumes, the manufacturing process can be tuned to a particular design, increasing the yield.
2. It is less work to design a high-volume part than a low-volume part.
3. The masks used to make the chip are expensive, so the cost per chip is lower for higher volumes.
4. Engineering development costs are high and largely independent of volume; thus, the development cost per die is lower with high-volume parts.
5. High-volume parts usually have smaller die sizes than low-volume parts and therefore, have higher yield per wafer.

1.6 性能

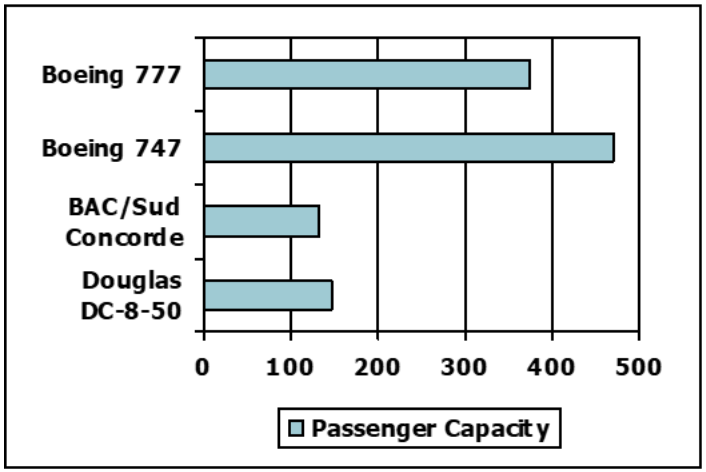
- 什么是性能，如何比较?
  - 以飞机为例，从载客量、航程、航速、乘客吞吐率等方面比较

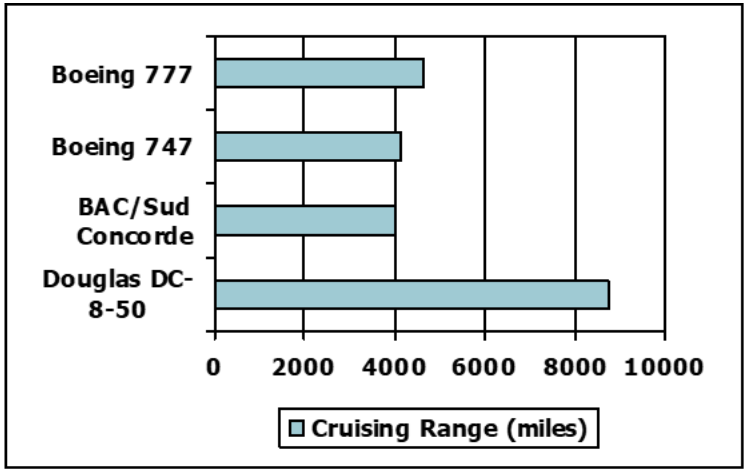
Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers × m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

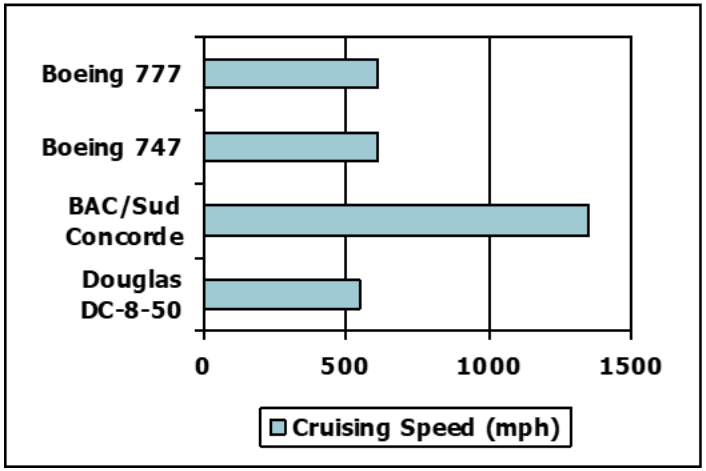
FIGURE 1.14 The capacity, range, and speed for a number of commercial airplanes. The last column shows the rate at which the airplane transports passengers, which is the capacity times the cruising speed (ignoring range and takeoff and landing times).

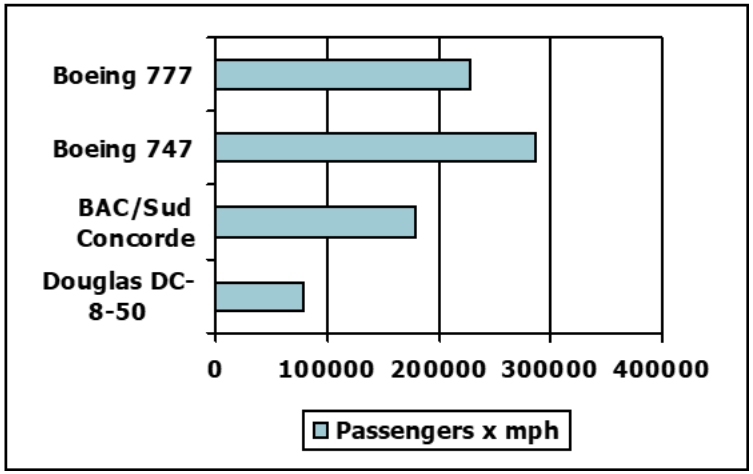
1.6 性能

- 飞机性能比较


Boeing 777
Boeing 747
BAC/Sud Concorde
Douglas DC-8-50
0
100
200
300
400
500
Passenger Capacity


Boeing 777
Boeing 747
BAC/Sud Concorde
Douglas DC-8-50
0
2000
4000
6000
8000
10000
Cruising Range (miles)


Boeing 777
Boeing 747
BAC/Sud Concorde
Douglas DC-8-50
0
500
1000
1500
Cruising Speed (mph)


Boeing 777
Boeing 747
BAC/Sud Concorde
Douglas DC-8-50
0
100000
200000
300000
400000
Passengers x mph

1.6 性能

- 响应时间
  - How long it takes to do a task
- 吞吐率（带宽）
  - Total work done per unit time
- 影响响应时间和吞吐率的因素
  - Replacing the processor with a faster version?
  - Adding more processors?

- 个人电脑：更关注响应时间
- 服务器：更关注吞吐率

1.6 性能

- 性能 = 1/执行时间

- X的执行速度是Y的n倍
  - 性能$_X$ / 性能$_Y$ = 执行时间$_Y$ / 执行时间$_X$ = n
- 示例：程序运行时间
  - 计算机A运行某程序，需要10秒
  - 计算机B运行同一程序，需要15秒
  - 性能$_A$ / 性能$_B$ = 执行时间$_A$ / 执行时间$_B$ = 15s/10s = 1.5

  - 计算机A的性能是B的1.5倍

1.6 性能

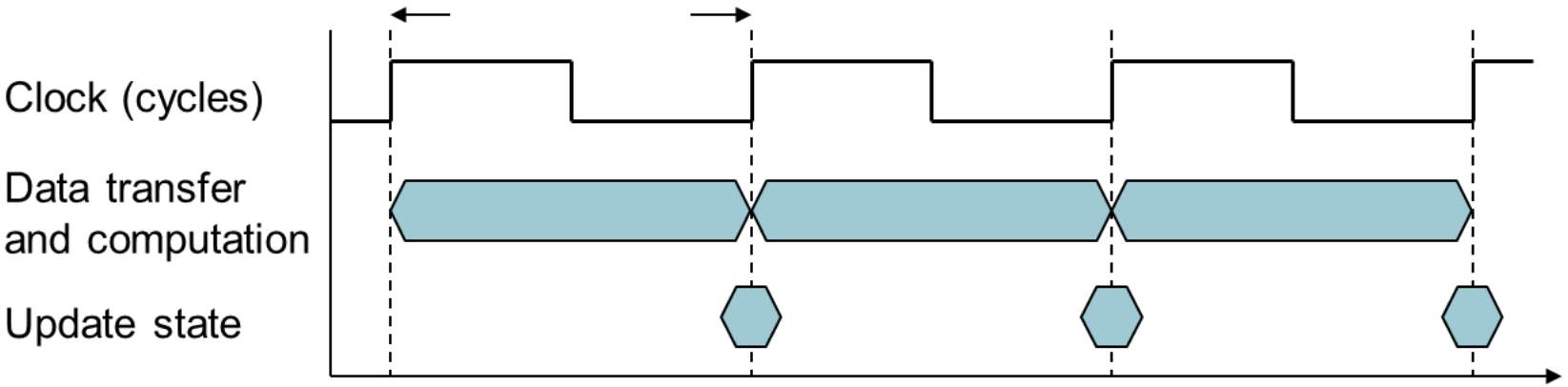
- 性能的度量：时间

- 运行时间：Elapsed time
  - 又称挂钟时间、响应时间
- CPU时间(或CPU执行时间)
  - 执行某一任务，在CPU上所花费的时间
  - 不包括IO时间、运行其它程序的时间
  - CPU时间 = 用户CPU时间 + 系统CPU时间
  - 用户CPU时间：程序本身所花费的CPU时间
  - 系统CPU时间：为执行程序而花费在操作系统上的时间

- 不同程序受CPU性能和系统性能的影响不同

1.6 性能

- CPU时钟/ CPU clock
  - 时钟周期，如：10ns
  - 时钟频率，如：4GHz



1.6 性能

- 影响CPU时间的相关参数
  - 减少时钟数
  - 增加时钟频率

  - 硬件设计者需在时钟频率和时钟周期数之间权衡

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

1.6 性能

- CPU时间举例
- 某程序在时钟频率为2GHz的计算机A上运行需要10秒，现尝试设计计算机B，将运行时间缩短为6秒，但由于频率的提高，时钟周期数变为计算机A的1.2倍，试计算B的时钟频率

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

1.6 性能

- CPI：Clock Pre Instruction
- 程序指令数（Instruction Count）的影响因素
  - 程序、ISA、编译器
- 指令的平均执行周期
  - 由CPU的硬件结构决定
  - 如果不同指令有不同的CPI，则需要考虑平均CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

1.6 性能

- 基于权重的平均CPI

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instructio n Count}_i)$$

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

1.6 性能

- CPI举例：
- 计算机 A：Cycle Time = 250ps, CPI = 2.0
- 计算机 B：Cycle Time = 500ps, CPI = 1.2
- 相同的ISA
- 用I表示总指令数，哪台计算机更快？快多少？

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$ ← A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$ ← …by this much

1.6 性能

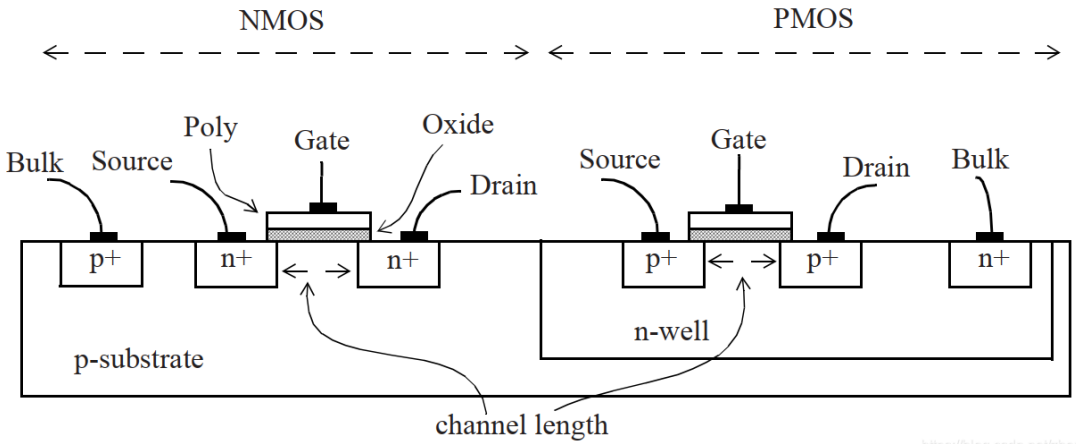
- CPI相关例题，请参考教材

$$\text{CPU Time} = \frac{\text{Instructio ns}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructio n}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

1.7 功耗墙

- CMOS工艺集成电路的功耗
  - 静态功耗：漏电流，与工艺相关



  - 动态功耗：与负载、电压、频率相关

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30　　5V → 1V　　×1000

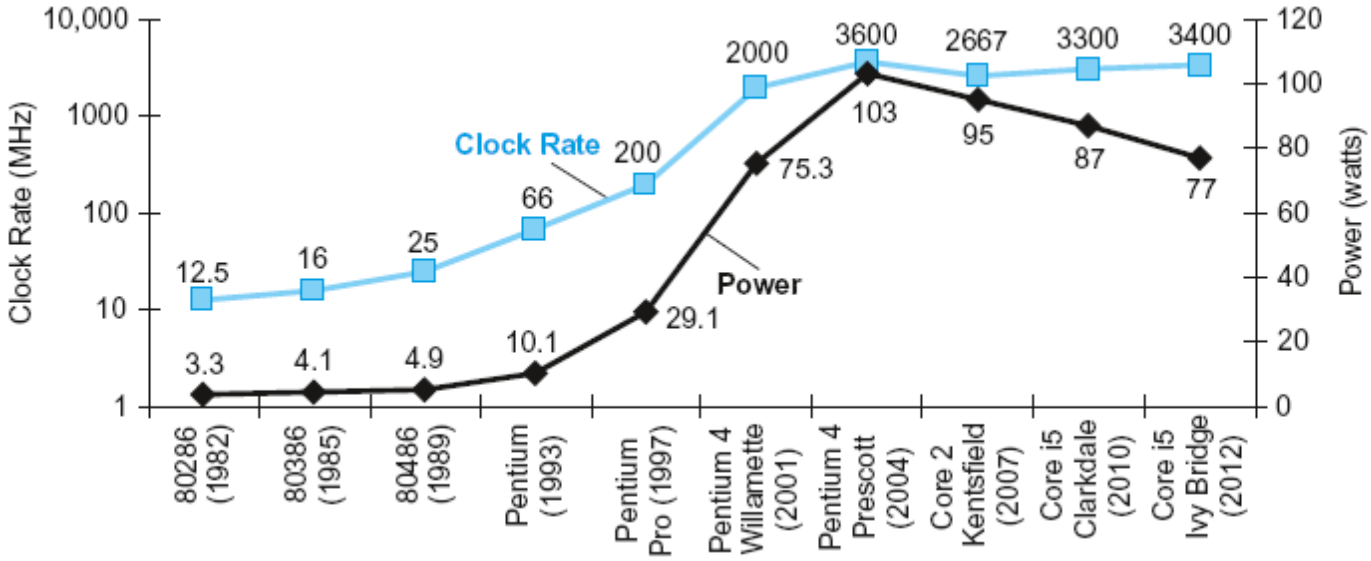
1.7 功耗墙

- 如何降低功耗?
- 假设一个新的CPU
  - 负载电容降为原来的85%
  - 电压和频率也都降为原来的85%

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

- 难题
  - 无法继续降低电压，电压降低将导致漏电流增大
  - 缺乏有效的降温手段
- 如何进一步提高计算机性能?

1.7 功耗墙



谁最早提出多核处理器？谁制造了第一个多核处理

我来答 分享 举报

1个回答 #热议# 医生收受红包、抢着给领导买单构成受贿罪吗?

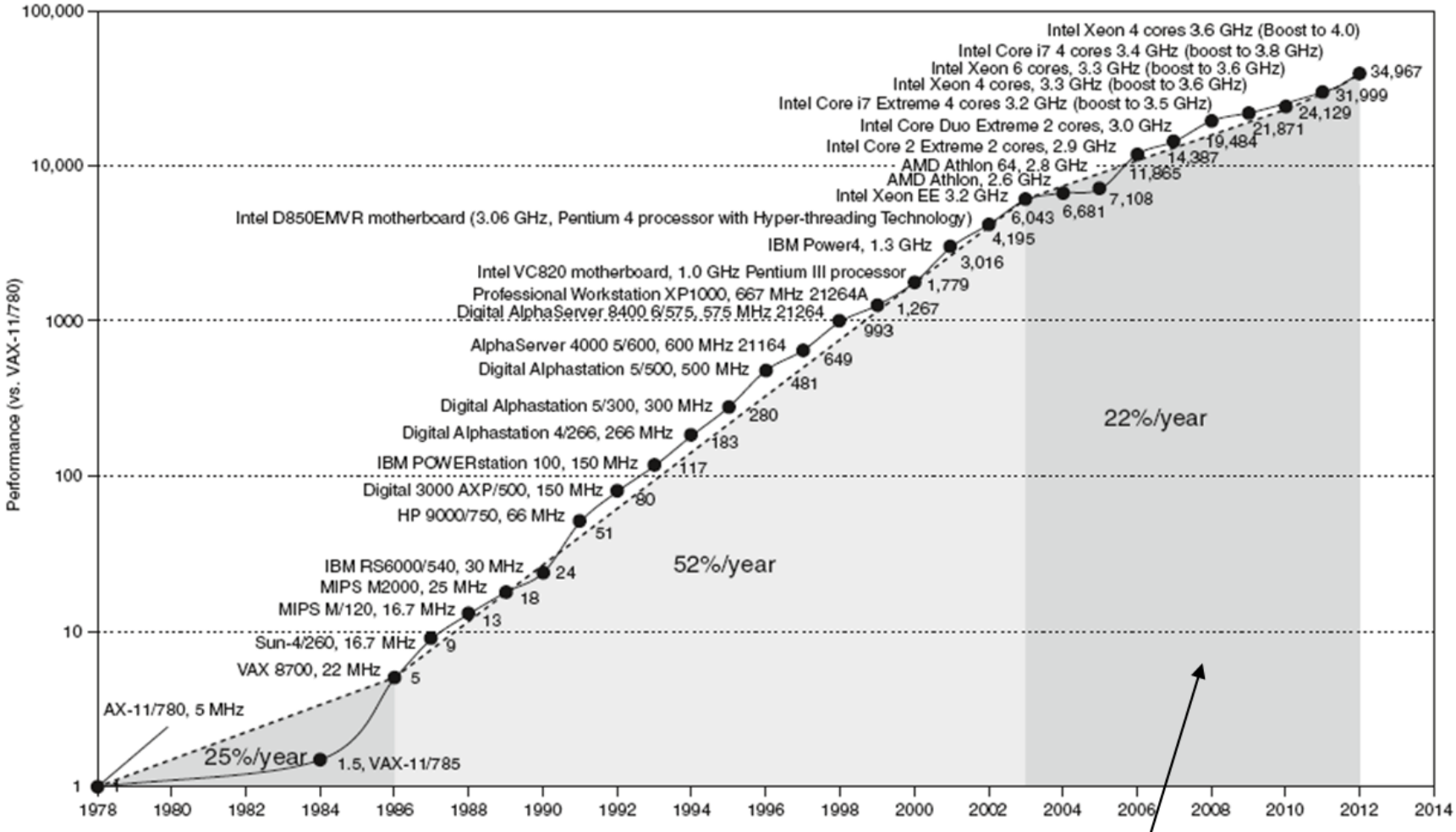
绫波的老公t

2014-07-10 · 超过72用户采纳过TA的回答

关注

AMD最早提出多核，2004提出的，结束了误入歧途的高频之争。 第一款AMD双核应该是速龙X2 3600+，intel第一款应该是奔腾D820(我的第一款电脑就是这款功耗很大的双核）。 如果严格来说，ARM11，9处理器其实就是可以多路并联的CPU，比如诺基亚N96手机就采用了ARM9双核处理器，不过由于CPU太老了，性能一般，那个时候手机是不关注CPU的。真正把手机带到双核的ARM A9的NVIDIA tegra 2处理器。

1.8 沧海巨变：从单处理器到多处理器



Constrained by power, instruction-level parallelism, memory latency

1.8 沧海巨变：从单处理器到多处理器

- 多处理器
  - 每个芯片有多个处理器
- 需要显式并行编程
  - 与指令级并行性比较
    - 硬件一次执行多条指令
    - 对程序员隐藏
- 难点
  - 高性能编程
  - 负载平衡
  - 优化通信和同步

1.9 实例：测评Intel Core i7

- 略

1.10 谬误与陷阱

- 谬误1：低利用率的计算机具有更低功耗
- 谬误2：面向性能的设计和面向能效的设计具有不相关的目标
- 陷阱1：在改进计算机的某个方面时，期望总性能的提高与改进大小成正比
- 陷阱2：用性能公式的一个子集去度量性能

- Amdahl定律：

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

- MIPS：每秒百万条指令数

$$MIPS = \frac{Instruction\ count}{Execution\ time \times 10^6}$$

$$= \frac{Instruction\ count}{\frac{Instruction\ count \times CPI}{Clock\ rate} \times 10^6} = \frac{Clock\ rate}{CPI \times 10^6}$$

1.11 本章小结

- 8个伟大思想
- ISA
- 性能评价：以真实程序的执行时间为尺度
- CPI
- MIPS
- CPU时间 = 用户CPU时间 + 系统CPU时间
- CPU时间 = 指令数 * CPI * 时钟周期长度
- Amdahl定律

计算机组成原理
CH2_指令：计算机的语言

卢建良
lujl@ustc.edu.cn
2022年春季学期

提纲

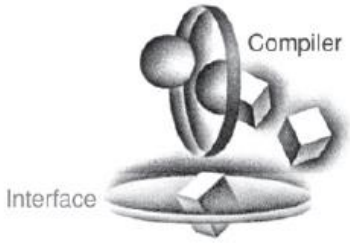
- 引言
- 计算机硬件的操作
- 计算机硬件的操作数
- 有符号数与无符号数
- 计算机中的指令表示
- 逻辑操作
- 用于决策的指令
- 计算机硬件对过程的支持
- 人机交互
- 对大立即数的RISC-V编址和寻址
- 指令与并行性：同步
- 翻译并启动程序

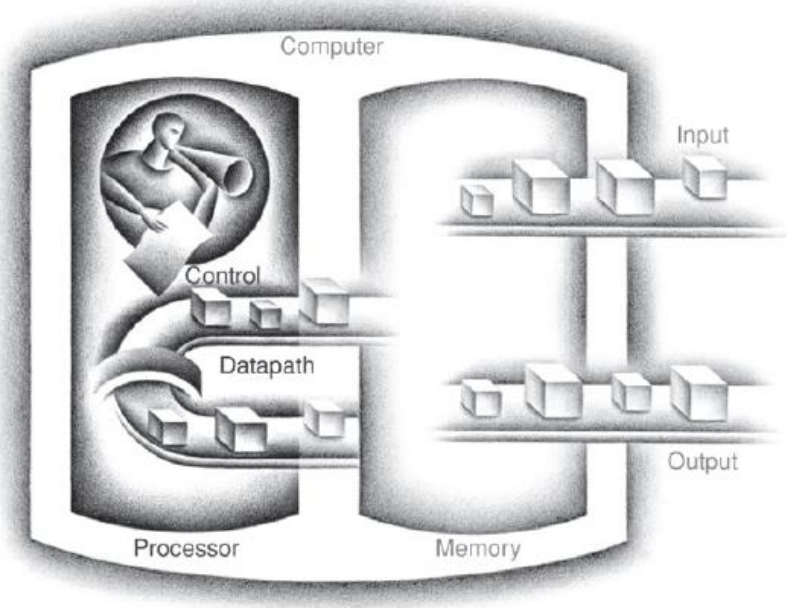
提纲

- 以C排序程序为例的汇总整理
- 数字与指针
- 高级专题：编译C语言和解释JAVA语言
- 实例：MIPS指令
- 实例：x86指令
- 实例：RISC-V指令系统的剩余部分
- 谬误与陷阱
- 本章小结

引言

- 指令：Instruction，什么是指令
- 指令集：Instruction Set
- 指令集架构：Instruction-Set Architecture, ISA
  - 指令集
  - 指令集编码
  - 基本数据类型
- 常见的指令集
  - x86:intel,amd
  - mips:龙芯
  - arm：苹果，华为
  - risc-v：COD
    - 开源
    - 模块化
    - 可扩展
    - 更成熟





2.1 引言

- RV32I
  - 算术
  - 访存
  - 逻辑
  - 移位
  - 分支
  - 跳转
  - 其它

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	x5 = x6 + x7	Three register operands; add
	Subtract	`sub x5, x6, x7`	x5 = x6 - x7	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	x5 = x6 + 20	Used to add constants
Data transfer	Load doubleword	`ld x5, 40(x6)`	x5 = Memory[x6 + 40]	Doubleword from memory to register
	Store doubleword	`sd x5, 40(x6)`	Memory[x6 + 40] = x5	Doubleword from register to memory
	Load word	`lw x5, 40(x6)`	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	x5 = Memory[x6 + 40]	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	x5 = x6 << x7	Shift left by register
	Shift right logical	`srl x5, x6, x7`	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	x5 = x6 >> 3	Arithmetic shift right by immediate

2.1 引言

Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

- 指令集详细介绍请参考相关文档，riscv.org
- RV64的操作数（64位），RV32为32位

RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

2.1 引言

- RV32I指令集汇总
  - 共47条
  - 本课程主要学习前37条

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I cssrrsi
csr			zimm	111	rd	1110011	I csrrci

2.2 计算机硬件的操作

- 加法/减法
  - 人的语言：A加B等于多少（用C表示）？
  - 高级语言：C = A + B
  - 汇编语言：add C, A, B //A+B,结果存入C中
  - 机器语言：有约定含义的二进制数字：010101.....0101010
  - 所有的算术操作都采用这种格式
    - 减法如何实现？多个数的加法如何实现?
- 设计原则1：简单源于规整
  - 规则性使实现更简单
  - 简单性可以以更低的成本实现更高的性能

2.2 计算机硬件的操作

- 举例: `f = (g + h) - (i + j);`

- RISC-V汇编代码

```
add t0, g, h    // temp t0 = g + h
add t1, i, j    // temp t1 = i + j
sub f, t0, t1   // f = t0 - t1
```

- 上述汇编代码是否能真正运行? 为什么

2.3 计算机硬件的操作数

- 位，bit：最小单位
- 字节，Byte：8bit，编址的最小单位
- 半字，Halfword，16bit，2个字节，注意不是任意组合
- 字，word，32bit，RV32的基本操作单元
- 双字，double word，64bit，RV64的基本单元
- 寄存器
  - 算术运算操作数的主要来源
  - 有32个，32bit（RV32）或者64bit（RV64）
  - 表示：x0~x31
  - 为什么限制为32个?
- 设计原则2：更少则更快
  - 主存数量：4G内存有2^{32}个字节

2.3 计算机硬件的操作数

- 寄存约定
  - x0: the constant value 0
  - x1: return address
  - x2: stack pointer
  - x3: global pointer
  - x4: thread pointer
  - x5 – x7, x28 – x31: temporaries
  - x8: frame pointer
  - x9, x18 – x27: saved registers
  - x10 – x11: function arguments/results
  - x12 – x17: function arguments

2.3 计算机硬件的操作数

- 举例:
- C代码 `f = (g + h) - (i + j);`
  - `f, …, j in x19, x20, …, x23`

- RISC-V汇编代码

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

- 上述汇编代码是否能真正运行？为什么

2.3 计算机硬件的操作数

- 存储器操作数
  - 用于存储复合数据
    - 数组、结构体、动态数据
  - 在算术运算中的作用
    - 加载：Load，将值从内存加载到寄存器
    - 存储：Store，将结果从寄存器存储到内存
  - 以字节为单位进行编址（历史原因）
  - 大小端问题
    - Big Endian
    - Little Endian
    - RISC-V采用小端模式
  - 对齐问题
    - 一个字占用4个字节，是否必须为4的倍数?
    - 不同于有些架构，RISC-V不要求必须对齐

2.3 计算机硬件的操作数

- 示例:

- C代码

```
A[12] = h + A[8];
```

  - h存放在x21
  - A的基地址存放在x22

- 编译后的RISC-V汇编代码

```
ld          x9, 64(x22)
add         x9, x21, x9
sd          x9, 96(x22)
```

  - RV64的基本单元为双字，占用8个字节
  - 8号元素的偏移地址为64

2.3 计算机硬件的操作数

- 寄存器与存储器的比较
- 寄存器访问速度比存储器快
- 存储器中的数据操作需要使用Load、Store操作
  - 用到更多的指令
- 寄存器可以同时操作三个数据
- 编译器必须尽可能多地使用变量寄存器
  - 仅将不太常用的变量存放到内存中
  - 寄存器优化很重要!

2.3 计算机硬件的操作数

- 常数（立即数）操作数
  - 指令中指定的常量数据
  - `addi x22, x22, 4`
- 加速经常性事件
  - 小常数很常见
  - 立即操作数避免加载指令

2.4 有符号数与无符号数

- 计算机中所有的信息都由二进制数位表示
- 二进制数位：又称位，以2为基数表示，作为信息的基本单元
- 给定一个n位（n-bit）的数字

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- MSB，most significant bit，最高有效位
- LSB，least significant bit，最低有效位
- 示例：
  - $0000\ 0000 \ldots 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- 范围：$0 \sim 2^n-1$
  - 64bit数据范围
  - 0 to +18,446,774,073,709,551,615

2.4 有符号数与无符号数

- 如何表示负数?
- 原码表示：用单独的一个bit来表示正负
  - 符号位放在什么位置?
  - 在算术运算中，如何设置符号位?
  - 0有两种表示方式
- 二进制补码表示
  - 易于进行符号判定：MSB
  - 扩展方便
  - 运算方便
  - 一一对应

2.4 有符号数与无符号数

- 二进制补码
- 给定一个n-bit的数字

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- 范围：-2^{n-1} to $+2^{n-1} - 1$

- 示例：
  - 1111 1111 … 1111 11002
    = –1×231 + 1×230 + … + 1×22 +0×21 +0×20
    = –2,147,483,648 + 2,147,483,644 = –410

- 64bit数据范围
  - –9,223,372,036,854,775,808
    to 9,223,372,036,854,775,807

2.4 有符号数与无符号数

- 有符号数取反
  - 按位取反再加1
  - 按位取反：1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- 示例：从2的二进制求-2的二进制补码表示
  - +2 = 0000 0000 ... 0010two
  - –2 = 1111 1111 ... 1101two + 1
    = 1111 1111 ... 1110two

2.4 有符号数与无符号数

- 符号位扩展
  - 将一个有符号数用更多的bit位表示出来
  - 将MSB复制并填充到数据的左侧
- 示例：8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- In RISC-V instruction set
  - lb: sign-extend loaded byte
  - lbu: zero-extend loaded byte

Check Yourself

What is the decimal value of this 64-bit two's complement number?

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111000_{two}$$

1) -4_{ten}
2) -8_{ten}
3) -16_{ten}
4) $18,446,744,073,709,551,609_{ten}$

2.5 计算机中的指令表示

- 程序员以汇编语言的方式识别和使用指令
- 计算机中的指令是以二进制数值进行存储的
  - 机器码， machine code

- RISC-V指令
  - 32bit表示，简单源于规整
  - 划分为多个字段
  - 用字段来表示操作类型、寄存器编号等信息
- 示例：
  - add x9, x20, x21
  - 十进制
  - 二进制

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

  - 十六进制的引入

2.5 计算机中的指令表示

- 二进制编写冗长、阅读困难
- 十进制转换麻烦
- 16进制表示收到普遍欢迎

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- 示例：
  - 二进制：1110 1100 1010 1000 0110 0100 0010 0000
  - 十六进制：eca8 6420
  - 十进制：计算繁琐，且存在符号问题

2.5 计算机中的指令表示

- 指令格式，RISC-V指令都是32bit的
  - R-type ： add
  - I-type ： addi，lw/ld
  - S-type ： sw
  - B-type
  - J-type
  - U-type

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands; add
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands; subtract
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
Data transfer	Load doubleword	ld x5, 40(x6)	x5 = Memory[x6 + 40]	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	Memory[x6 + 40] = x5	Doubleword from register to memory
	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register

2.5 计算机中的指令表示

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- R-type：寄存器类型，6个指令字段
  - opcode：操作码字段
  - rd：目的寄存器字段
  - funct3：附加的操作码字段
  - rs1：第一个源操作数寄存器字段
  - rs2：第二个源操作数寄存器字段
  - funct7：附加的操作码字段

- 思考：
  - 操作码字段为什么要分开
  - 为什么是3个寄存器字段
  - 寄存器字段为什么是5bit

2.5 计算机中的指令表示

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- 汇编语言：`add x9,x20,x21`

0	21	20	0	9	51

0000000	10101	10100	000	01001	0110011

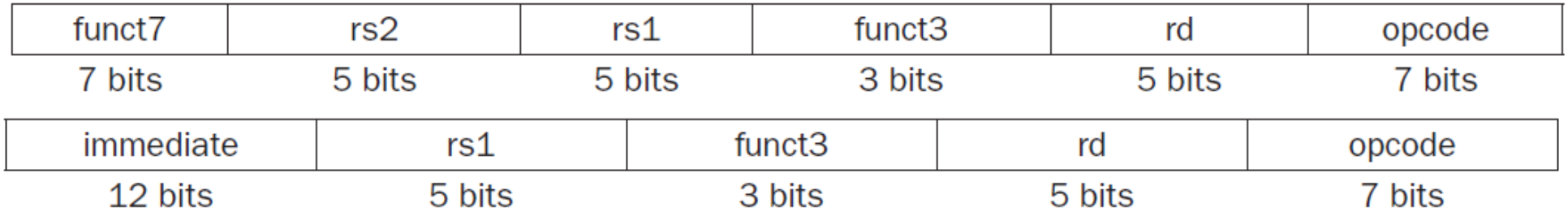
- 二进制表示：
  - $0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two}$
- 十六进制表示：
  - $015A04B3_{16}$

2.5 计算机中的指令表示

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

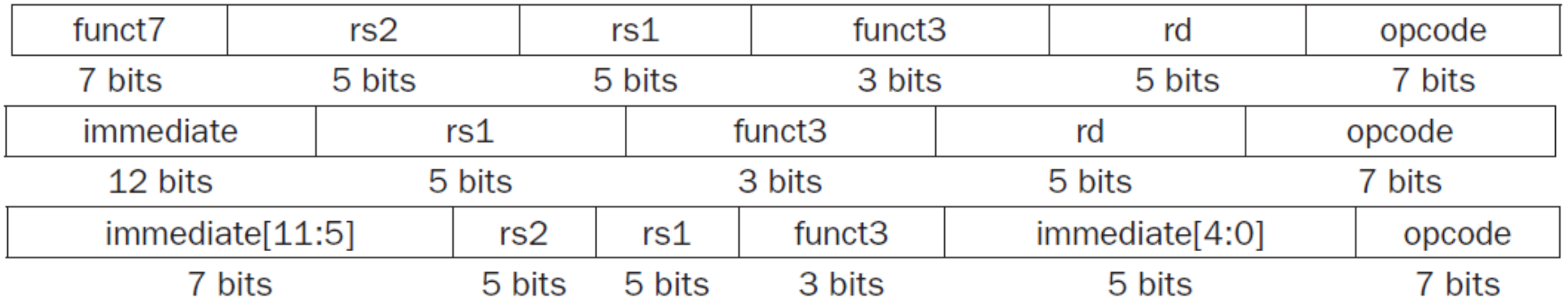
- 加载指令，
  - ld x9, 64(x22)
  - 将内存中地址为[x22]+64的数据加载到x9寄存器
- 无法用R-type指令格式实现，如何解决矛盾?
  - 方案1：增加新的指令字段
  - 方案2：增加新的指令格式
- 设计原则3：优秀的设计需要折中

2.5 计算机中的指令表示



- 立即数运算和加载指令（I-type）
  - lw x9, 64(x22)
  - addi x9,x22,64
- 与R-type相比
  - 多了立即数字段
  - 少了一个rs字段
  - 操作码字段也可以简化
- funct7（7bit）+ rs2（5bit） → immediate字段（12bit）

2.5 计算机中的指令表示



- 存储指令（S-type）
  - sw x9, 64(x22)
  - 将x9寄存器中的数据存储到内存中地址为[x22]+64的位置
- 指令分析
  - 两个源寄存器字段
  - 一个立即数字段
  - 操作码字段
- 保持rs1，rs2，opcode字段位置不变

2.5 计算机中的指令表示

- 三种类型的指令格式总结
  - R-type
  - I-type
  - S-type

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (add immediate)	001111101000		00010	000	00001	0010011	addi x1, x2, 1000
ld (load doubleword)	001111101000		00010	011	00001	0000011	ld x1, 1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sd (store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1, 1000(x2)

2.5 计算机中的指令表示

- 已涉及指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I cssrrsi
csr			zimm	111	rd	1110011	I csrrci

Translating RISC-V Assembly Language into Machine Language

EXAMPLE

We can now take an example all the way from what the programmer writes to what the computer executes. If `x10` has the base of the array `A` and `x21` corresponds to `h`, the assignment statement

```
A[30] = h + A[30] + 1;
```

is compiled into

```
ld   x9, 240(x10)    // Temporary reg x9 gets A[30]
add  x9, x21, x9     // Temporary reg x9 gets h+A[30]
addi x9, x9, 1       // Temporary reg x9 gets h+A[30]+1
sd   x9, 240(x10)    // Stores h+A[30]+1 back into A[30]
```

What is the RISC-V machine language code for these three instructions?

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	00	00000	1110011	I ecall
000000000000		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I cssrrsi
csr		zimm	111	rd	1110011	I csrrci

2.5 计算机中的指令表示

- 答案

```
ld   x9, 240(x10)    // Temporary reg x9 gets A[30]
add  x9, x21, x9     // Temporary reg x9 gets h+A[30]
addi x9, x9, 1       // Temporary reg x9 gets h+A[30]+1
sd   x9, 240(x10)    // Stores h+A[30]+1 back into A[30]
```

immediate	rs1	funct3	rd	opcode
000011110000	01010	011	01001	0000011

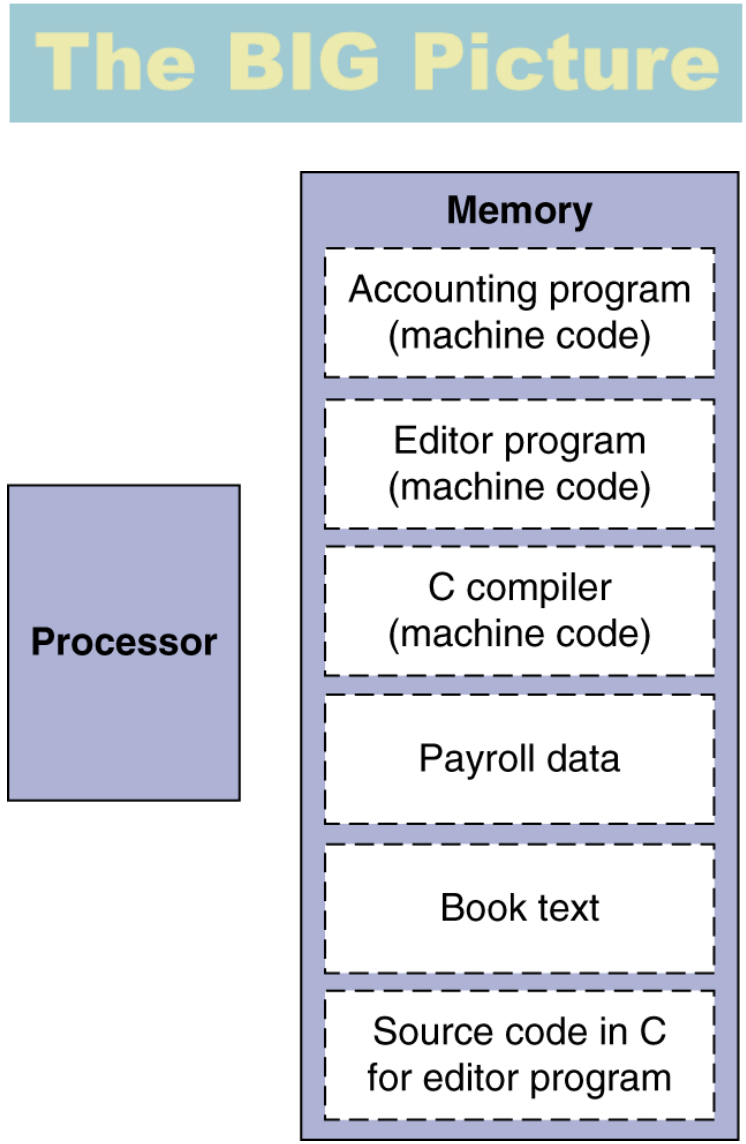
funct7	rs2	rs1	funct3	rd	opcode
0000000	01001	10101	000	01001	0110011

immediate	rs1	funct3	rd	opcode
000000000001	01001	000	01001	0010011

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
0000111	01001	01010	011	10000	0100011

2.5 计算机中的指令表示

- 当代计算机构建基于两个关键原则
  - 指令由数字形式表示
  - 指令和数据一样保存在存储器中进行读写
- 引出程序存储的概念
- 二进制兼容性
  - 程序以二进制形式发布
  - 促使指令集生态的形成
    - x86、ARM



2.5 计算机中的指令表示

自我测试

What RISC-V instruction does this represent? Choose from one of the four options below.

Check Yourself

funct7	rs2	rs1	funct3	rd	opcode
32	9	10	000	11	51

1. `sub x9, x10, x11`
2. `add x11, x9, x10`
3. `sub x11, x10, x9`
4. `sub x11, x9, x10`

2.6 逻辑操作

- 最初的计算机只对整字进行操作
- 实践表明对字内的单个或多个位进行操作非常必要

- 常见的逻辑操作

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

  - Useful for extracting and inserting groups of bits in a word

2.6 逻辑操作

- 移位操作
  - 将一个寄存器中的数据，向左/右移动一定位数
  - 填充问题：空出来的位，填0，还是填1?
  - 如何指定移动的bit数：立即数，还是寄存器?
- 示例：
  - slli x11,x19,4　　//reg x11 = reg x19 << 4bit
  - 是否兼容于已有的指令格式

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (add immediate)	001111101000		00010	000	00001	0010011	addi x1, x2, 1000
ld (load doubleword)	001111101000		00010	011	00001	0000011	ld x1, 1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sd (store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1, 1000(x2)

2.6 逻辑操作

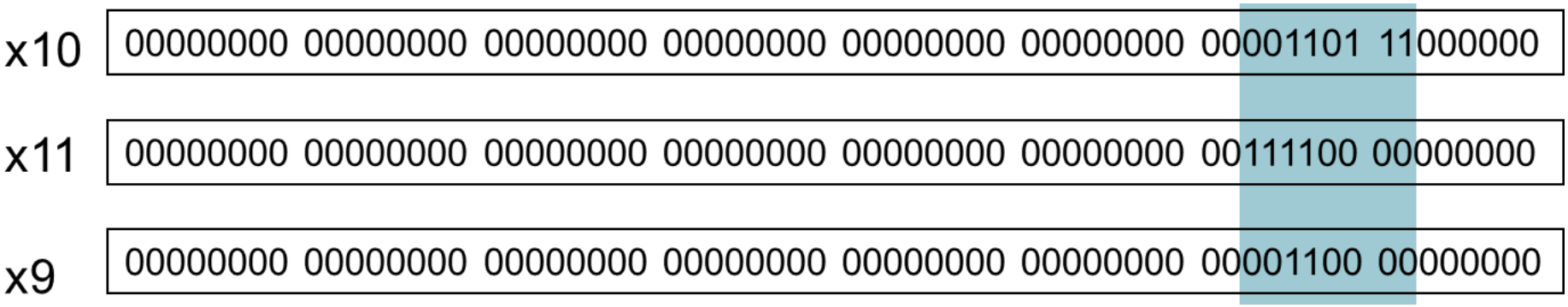
- 立即数逻辑左移：slli
  - 向左移位，低位补0
  - 左移i位，表示乘以2^i
- 立即数逻辑右移：srli
  - 向右移位，高位填0
  - 右移i位，表示除以2^i，仅对无符号
- 立即数算术右移：srai
  - 向右移位，高位复制符号位
- I-type

0	immediate	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

I-type Instructions	immediate	rs1	funct3	rd	opcode	Example
addi (add immediate)	001111101000	00010	000	00001	0010011	addi x1, x2, 1000
ld (load doubleword)	001111101000	00010	011	00001	0000011	ld x1, 1000 (x2)

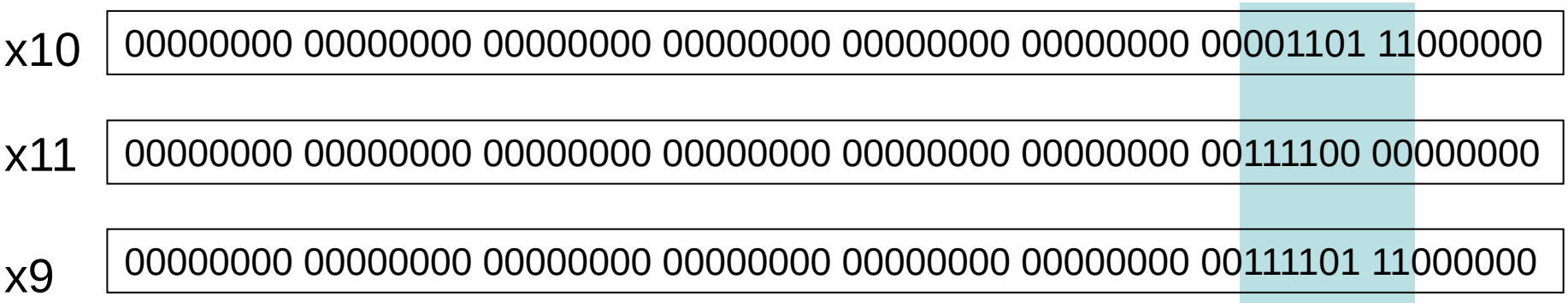
2.6 逻辑操作

- 与操作
  - and x9, x10,x11 //reg[x9] = reg[x10] & reg[x11]
  - 常用于将rd寄存器中的某些bit置为0
  - 常见用法：掩码操作



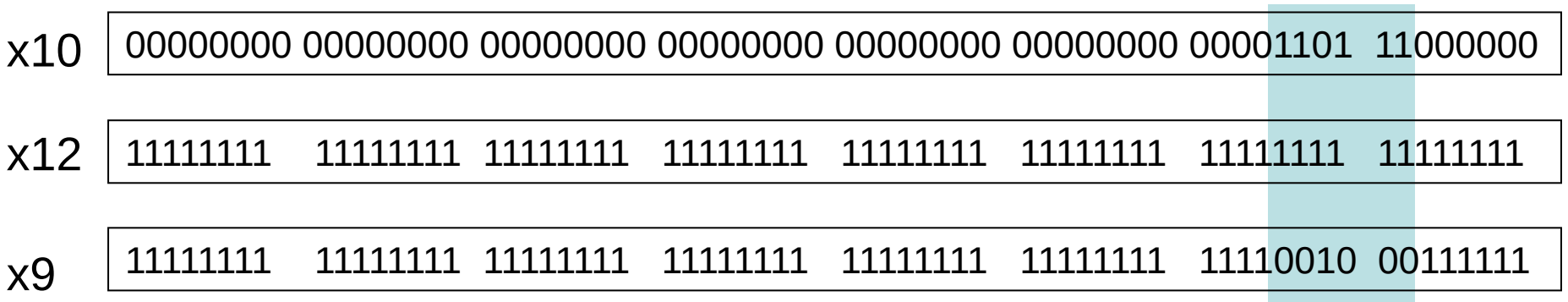
2.6 逻辑操作

- 或操作
  - or x9, x10,x11　　//reg[x9] = reg[x10] | reg[x11]
  - 常用于将rd寄存器中的某些bit置为0
  - 常见用法：掩码操作



2.6 逻辑操作

- 异或操作
  - xor x9, x10,x11　//reg[x9] = reg[x10] ^ reg[x11]
  - 异或可以同来实现取反操作（NOT）
  - RISC-V中没有专门的取反（NOT）指令



2.6 逻辑操作

- 总结
  - 3条移位逻辑操作指令
  - 3条逻辑运算指令
  - 上述指令都有I-type和R-type两种形式
- I-type
  - slli
  - srli
  - srai
- R-type
  - sll
  - srl
  - sra
- R-type
  - and
  - or
  - xor
- I-type
  - andi
  - ori
  - xori

2.6 逻辑操作

- 已涉及的指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I csrrsi
csr			zimm	111	rd	1110011	I csrrci

2.6 逻辑操作

Check Yourself

Which operations can isolate a field in a doubleword?

1. AND
2. A shift left followed by a shift right

2.7 用于决策的指令

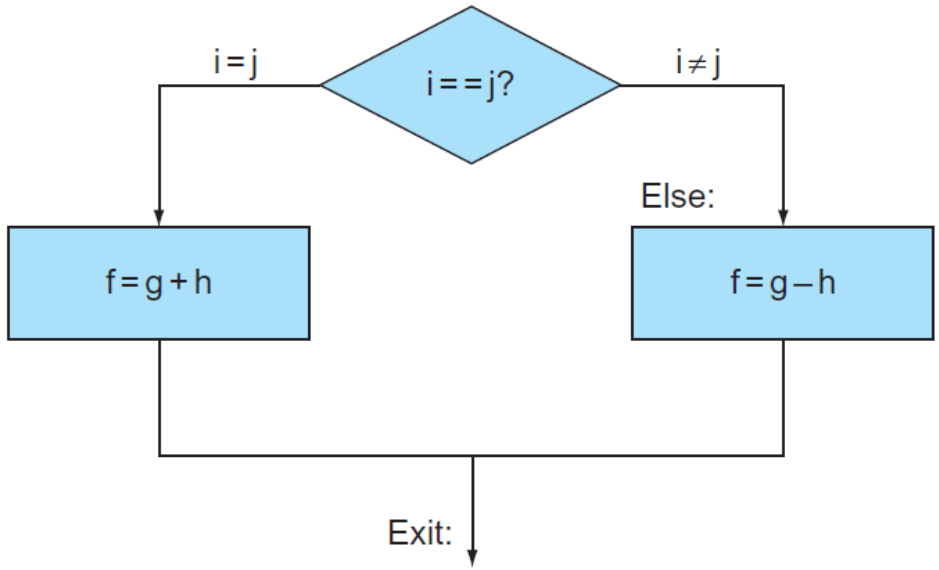
- 计算机与简单计算器的区别
  - 是否具备决策能力
  - 根据输入数据和计算结果的值，执行不同的指令
- C语言相关关键字
  - if...else
  - case/switch
- RISC-V汇编语言
  - beq rs1, rs2, L1
  - bne rs1, rs2, L1
  - beq、bne称为条件分支指令
- 条件分支指令
  - 一条指令，先检测一个值，然后根据检测结果，允许后续控制流转移到程序中的一个新地址

2.7 用于决策的指令

- if...else
  - if (i == j) f = g + h; else f = g – h;
  - f,g,h,i,j均为变量，分别存放在x19~x23寄存器中
- RISC-V汇编代码

```
        bne x22, x23, Else
        add x19, x20, x21
        beq x0,x0,Exit // unconditional
Else:   sub x19, x20, x21
Exit:   …
```

- 标签的实际地址由汇编器计算
- 对比
  - C语言：条件成立则执行
  - 汇编：条件成立则跳转



2.7 用于决策的指令

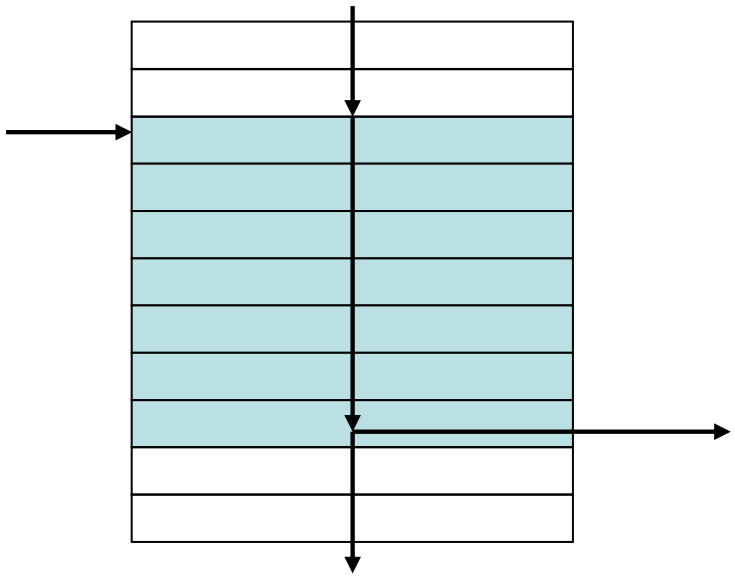
- 循环
  - while (save[i] == k) i += 1;
  - i in x22, k in x24, address of save in x25
- RISC-V汇编代码

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit
       addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

2.7 用于决策的指令

- 基本块
  - 一个没有分支的指令序列（结尾处可以有），同时没有分支目标或分支标签（起始处可以有）
  - No embedded branches (except at end)
  - No branch targets (except at beginning)

  - 编译器的基础工作之一就是将程序划分为基本块
  - 高级的处理器可以对基本块进行加速
    - how?



2.7 用于决策的指令

- 更多的决策指令

- blt rs1, rs2, L1
  - if (rs1 < rs2) branch to instruction labeled L1

- bge rs1, rs2, L1
  - if (rs1 >= rs2) branch to instruction labeled L1

- 示例
  - if (a > b) a += 1;
  - a in x22, b in x23

```
    bge  x23, x22, Exit       // branch if b >= a
    addi x22, x22, 1
Exit:
```

2.7 用于决策的指令

- 有符号比较
  - eg：blt, bge
- 无符号比较
  - eg： bltu, bgeu
- 示例
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23 // signed`
    - –1 < +1
  - `x22 > x23 // unsigned`
    - +4,294,967,295 > +1

2.7 用于决策的指令

- case/switch语句
  - 简单实现方式：if... else if ... else if ...
  - 更有效的实现方式：分支地址表
  - 包含代码中标签对应地址的一个数组
- 使用间接跳转指令实现：jalr
  - eg：中断向量表

2.7 用于决策的指令

- MIPS指令集架构
  - 先比较
  - 再根据比较结果进行分支判断
  - 优点：数据通路更简单
  - 缺点：指令条数增加
- ARM指令集架构
  - 条件码/标志位方式
  - 缺点：流水线依赖关系增加，难以优化
- RISC-V指令集架构
  - 最佳方案

2.7 用于决策的指令

- 已涉及的指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	00	00000	1110011	I ecall
000000000000		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I cssrrsi
csr		zimm	111	rd	1110011	I csrrci

2.7 用于决策的指令

Check Yourself

I. C has many statements for decisions and loops, while RISC-V has few. Which of the following does or does not explain this imbalance? Why?

1. More decision statements make code easier to read and understand.
2. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
3. More decision statements mean fewer lines of code, which generally reduces coding time.
4. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

II. Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||), while RISC-V doesn't?

1. Logical operations AND and ORR implement & and |, while conditional branches implement && and ||.
2. The previous statement has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.
3. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.

2.8 计算机硬件对过程的支持

- 过程：一个根据给定参数，执行特定任务的已存储的子程序
- 有助于提高可理解性和可重用性
- 帮助程序员一次只关注于任务的一个部分
- 执行过程时的六个步骤：
  - 将参数放到可以访问到的位置（x10~x17）
  - 将控制转交给过程，同时记住当前位置：jal x1,foo
  - 获取过程所需的存储资源
  - 执行所需的任务
  - 将结果放到调用程序可以访问到的位置
  - 将控制返回到初始点（x1）

2.8 计算机硬件对过程的支持

- 调用者，caller
  - 启动过程，并提供必要参数值的程序
- 被调用者，callee
  - 根据调用者提供的参数，执行一系列已存储的指令的过程，然后将控制权返还给调用者
- 程序计数器，program counter，PC
  - 一个包含程序中正在执行指令的地址的寄存器

2.8 计算机硬件对过程的支持

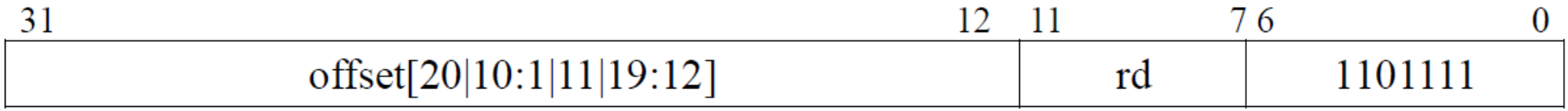
- 思考
  - 对于过程调用，是否可以用前面所讲的B-type指令实现?
  - eg：beq x0, x0, ProcedureAddress

2.8 计算机硬件对过程的支持

- 过程调用相关指令
  - jal：jump and link
  - jalr：jump and link register
- jal：一般用于过程调用
  - `jal x1, ProcedureLabel`
  - Address of following instruction put in x1
  - Jumps to target address
- jalr：一般用于过程调用结束后的返回
  - `jalr x0, 0(x1)`
  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - eg：case/switch语句

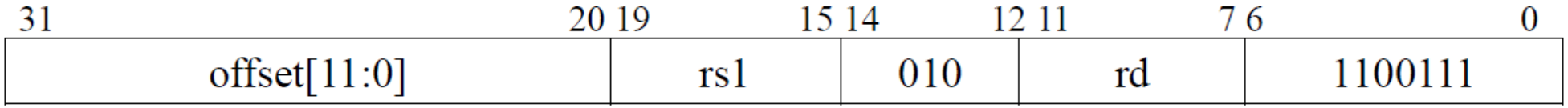
2.8 计算机硬件对过程的支持

- JAL
  - 汇编语法：jal rd，offset
  - 行为：x[rd] = pc + 4; pc += sext(offset)
  - 描述，该指令完成两件事情
    - 把下一条指令的地址（RV32为pc+4，RV64为pc+8）保存到rd中
    - 把pc设置为当前值加上符号扩展的offset
    - RISC-V中约定默认使用x1作为rd存放返回地址
  - 指令格式（J-type，RV32I中唯一的一条）
    - 操作码
    - 目的寄存器，rd
    - 偏移量offset（立即数）
    - 立即数共20位，offset[20:1]，没有bit0，且顺序打乱



2.8 计算机硬件对过程的支持

- JALR
  - 汇编语法：jalr rd，offset（rs1）
  - 行为：rd=pc+4; pc=(x[rs1]+sext(offset))&~1
  - 描述，该指令完成两件事情
    - 把下一条指令的地址（RV32为pc+4，RV64为pc+8）保存到rd中
    - 把pc设置为x[rs1]加上符号扩展的offset，最低位设置为0
    - RISC-V中约定默认使用x1作为rd存放返回地址
  - 指令格式（I-type）
    - 操作码
    - 目的寄存器
    - 源操作数寄存器
    - 立即数



2.8 计算机硬件对过程的支持

- 涉及指令汇总

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

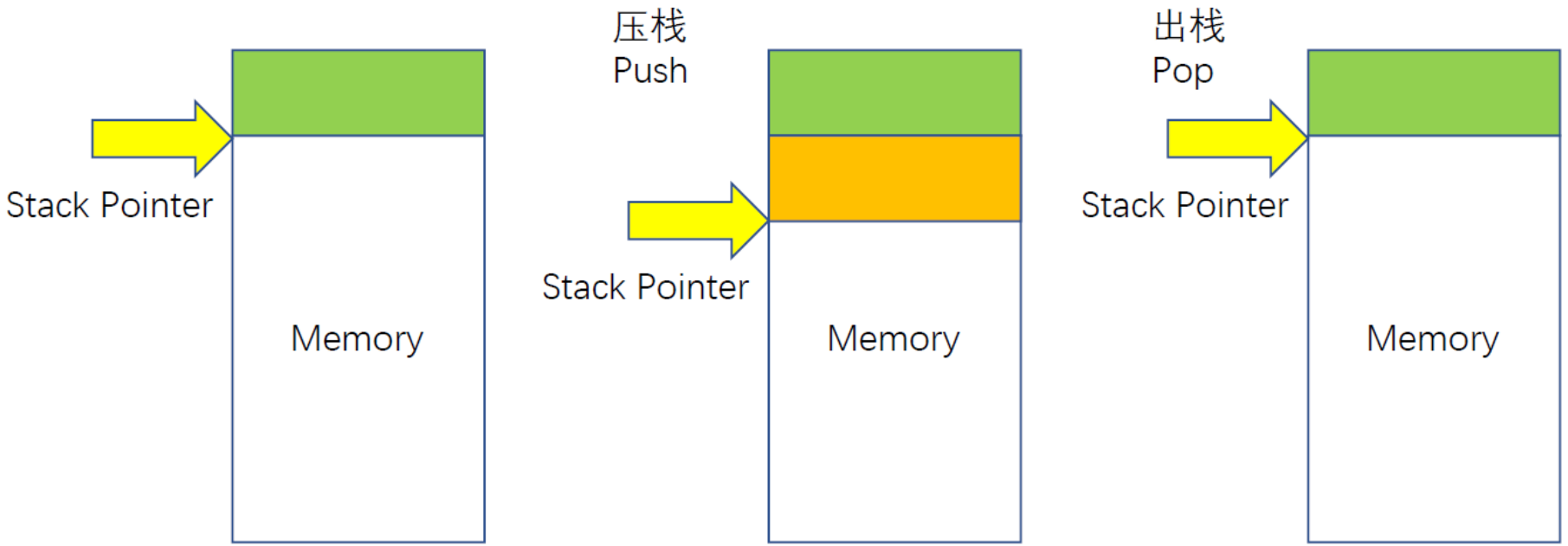
31 25	24 20	19 15	14 12	11 7	6 0	
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I cssrrsi
csr			zimm	111	rd	1110011	I csrrci

2.8 计算机硬件对过程的支持

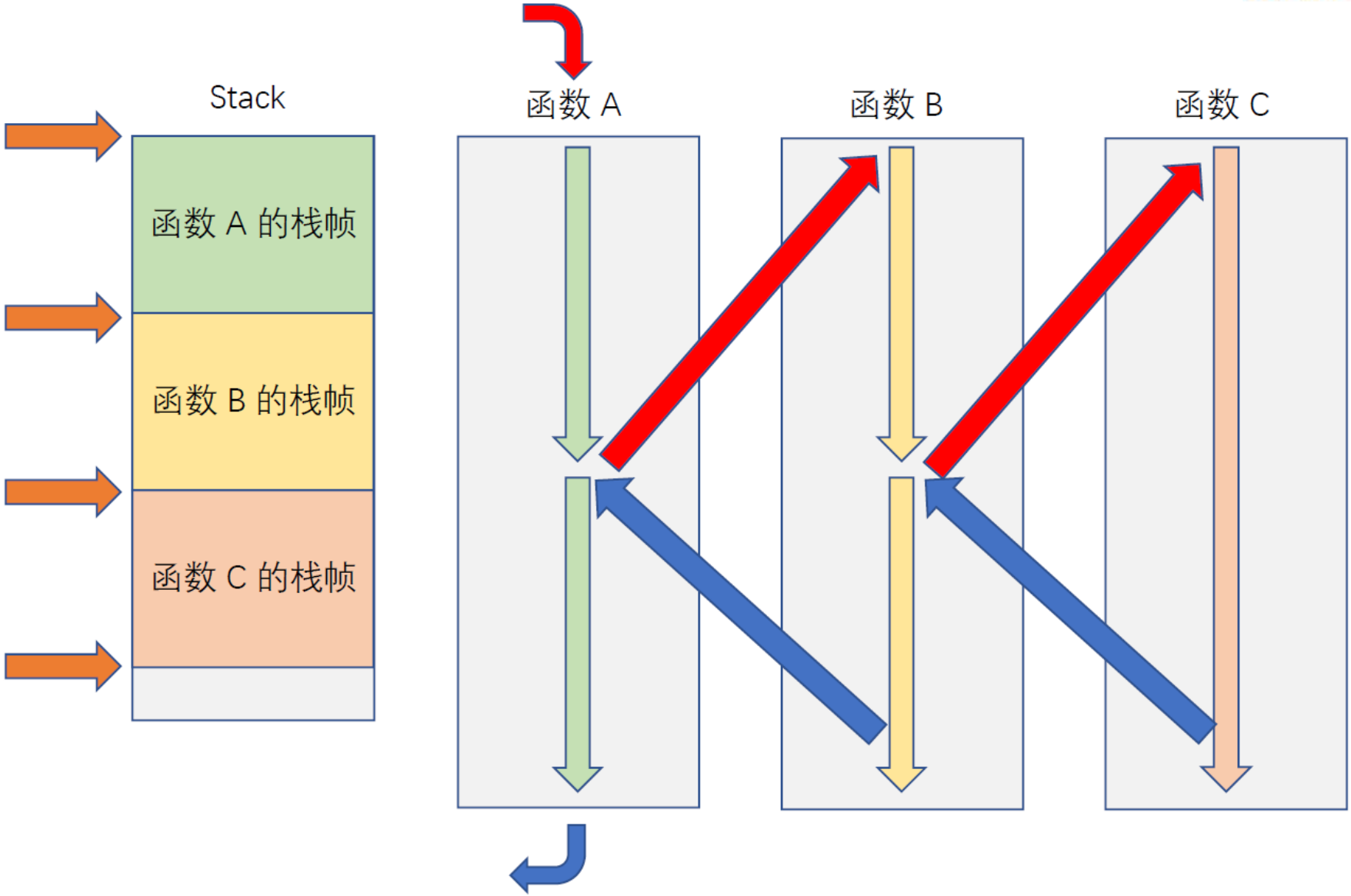
- 使用更多的寄存器
  - 过程调用会遇到参数传递的问题
  - 传递的参数是否有数量限制？参数多了如何处理?
  - 子过程是否可以毫无顾忌的使用所有寄存器?
  - 过程返回后如何恢复现场?
- 解决思路：充分利用大容量的主存
  - 参数传递采用传送地址的方式
  - 将寄存器换出到存储器中
- 解决方案：栈，stack
  - 栈是一种被组织成后进先出队列并用于寄存器换出的数据结构
  - 栈指针：stack pointer，也称sp，存放栈中最新分配的地址
  - RISC-V中约定使用寄存器文件中的x2存放sp

2.8 计算机硬件对过程的支持

- 栈的相关操作
  - 入栈/压栈，push，向栈中添加元素
  - 出栈/弹栈，pop，从栈中移除元素



2.8 计算机硬件对过程的支持



2.8 计算机硬件对过程的支持

- 示例:
- C code:

```
long long int leaf_example (
    long long int g, long long int h,
    long long int i, long long int j) {
  long long int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Need to save x5, x6, x20 on stack

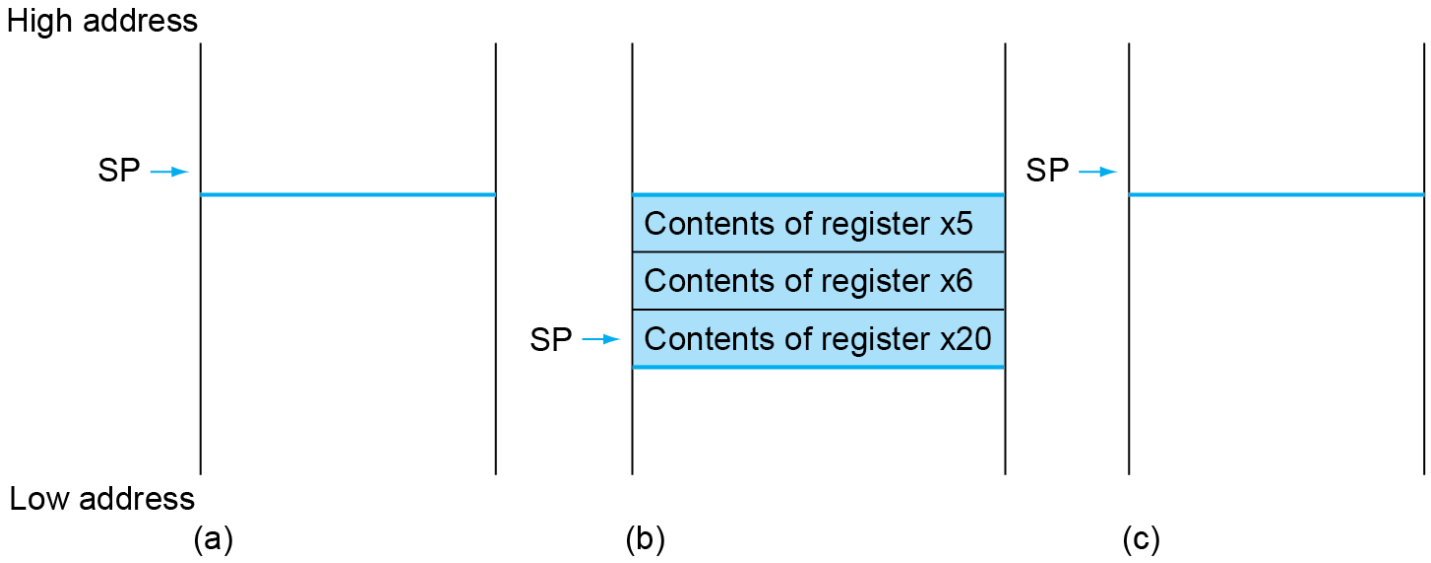
2.8 计算机硬件对过程的支持

- RISC-V 汇编代码:

```
leaf_example:
   addi sp,sp,-24
   sd   x5,16(sp)      //save x5, x6, x20 on stack
   sd   x6,8(sp)
   sd   x20,0(sp)
   add  x5,x10,x11     //x5 = g + h
   add  x6,x12,x1      //x6 = i + j
   sub  x20,x5,x6      //f = x5 - x6
   addi x10,x20,0      //copy f to return register
   ld   x20,0(sp)      //resore x5,x6,x20 from stack
   ld   x6,8(sp)
   ld   x5,16(sp)
   addi sp,sp,24
   jalr x0,0(x1)       //return to caller
```

2.8 计算机硬件对过程的支持

- 上图：栈操作示例
- 下图：过程调用中，并非所有的信息都需要保存



Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

2.8 计算机硬件对过程的支持

- 寄存器使用约定
  - x5 – x7, x28 – x31：临时寄存器， temporary registers，在调用过程中不被被调用者（callee）保存
  - x8 – x9, x18 – x27：保存寄存器， saved registers，在调用过程中必须被保存，一旦被调用者（callee）使用，由被调用者保存并恢复

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

2.8 计算机硬件对过程的支持

■ 函数调用过程中有关寄存器的编程约定

寄存器名	ABI 名 （编程用名）	用途约定	谁负责在函数调用过程中维护这些寄存器
x0	zero	读取时总为 0， 写入时不起任何效果	N/A
x1	ra	存放函数返回值（return address）	Caller
x2	sp	存放栈指针（stack pointer）	Callee
x5~x7, x28~x31	t0~t2, t3~t6	**临时（temporaries）寄存器**，Callee 可能会使用这些寄存器，所以 Callee 不保证这些寄存器中的值在函数调用过程中保持不变，这意味着对于 Caller 来说，如果需要的话，Caller 需要自己在调用 Callee 之前保存临时寄存器中的值。	Caller
x8, x9, x18~x27	s0, s1, s2~s11	**保存（saved）寄存器**，Callee 需要保证这些寄存器的值在函数返回后仍然维持函数调用之前的原值，所以一旦 Callee 在自己的函数中会用到这些寄存器则需要在栈中备份并在退出函数时进行恢复。	Callee
x10 , x11	a0 , a1	**参数（argument）寄存器**，用于在函数调用过程中保存第一个和第二个参数，以及在函数返回时传递返回值。	Caller
x12 ~ x17	a2 ~ a7	**参数（argument）寄存器**，如果函数调用时需要传递更多的参数，则可以用这些寄存器，但注意用于传递参数的寄存器最多只有 8 个（a0 ~ a7），如果还有更多的参数则要利用栈。	Caller

2.8 计算机硬件对过程的支持

- 叶子过程，leaf procedures
  - 不调用其它过程的过程，称为叶子过程
- 嵌套过程， Nested Procedures
  - 递归，Recursive：直接或者间接调用自己的过程
- 嵌套过程中Caller做的事
  - 调用前，保存返回地址
  - 调用前，保存所有的参数和返回后需要用到的临时变量
  - 调用后，从栈中恢复现场

- 递归调用演示视频?

https://www.bilibili.com/video/BV1sV41167zL?from=search&seid=14222955537136045147&spm_id_from=333.337.0.0

例子：

你和你朋友，约一个地点，在那里见面。 现在呢，他已经到了，你距离你们约定的地点还差100步。

那么，最后这100步，你肯定是需要一步一步走过去的。

那么咱们如何用递归的方式来实现最后走这100步呢。

下面看下代码：

```
function walk(step) {
    if (step == 0) {
        console.log(`已经到达了目的地`)
    } else {
        step--
        console.log(`走了一步，还剩下${step}步`)
        walk(step)
    }
}
walk(100)
```

2.8 计算机硬件对过程的支持

- 递归调用示例
- C代码

```
long long int fact (long long int n)
 {
   if (n < 1) return f;
   else return n * fact(n - 1);
 }
```

  - Argument n in x10
  - Result in x10

2.8 计算机硬件对过程的支持

- RISC-V汇编代码

```
long long int fact (long long int n)
 {
   if (n < 1) return f;
   else return n * fact(n - 1);
 }
```

```
fact:
    addi sp,sp,-16      Save return address and n on stack
    sd   x1,8(sp)
    sd   x10,0(sp)
    addi x5,x10,-1      x5 = n - 1
    bge  x5,x0,L1       if n >= 1, go to L1
    addi x10,x0,1       Else, set return value to 1
    addi sp,sp,16       Pop stack, don't bother restoring values
    jalr x0,0(x1)       Return
L1: addi x10,x10,-1     n = n - 1
    jal  x1,fact        call fact(n-1)
    addi x6,x10,0       move result of fact(n - 1) to x6
    ld   x10,0(sp)      Restore caller's n
    ld   x1,8(sp)       Restore caller's return address
    addi sp,sp,16       Pop stack
    mul  x10,x10,x6     return n * fact(n-1)
    jalr x0,0(x1)       return
```

2.8 计算机硬件对过程的支持

- 一些递归过程可以使用迭代实现
  - 可以降低开销，提高性能
  - 示例，C代码

```
long long int sum (long long int n, long long int acc) {
     if (n > 0)
          return sum(n - 1, acc + n);
     else
          return acc;
}
```

  - 汇编代码

```
sum: ble x10, x0, sum_exit   // go to sum_exit if n <= 0
     add x11, x11, x10       // add n to acc
     addi x10, x10, -1       // subtract 1 from n
     jal x0, sum             // jump to sum
sum_exit:
     addi x12, x11, 0        // return value acc
     jalr x0, 0(x1)          // return to caller
```

2.8 计算机硬件对过程的支持

- 在栈中为新数据分配空间
  - 过程帧：也称活动记录。栈中包含过程保存的寄存器和局部变量的段
  - 过程帧的地址由帧指针（fp）确定，RISC-V约定x8存放fp
  - 在过程中，fp比sp更加稳定，方便做为基地址
  - 在前面的示例中，仅在进入和退出过程中才调整sp，但实际上并不绝对

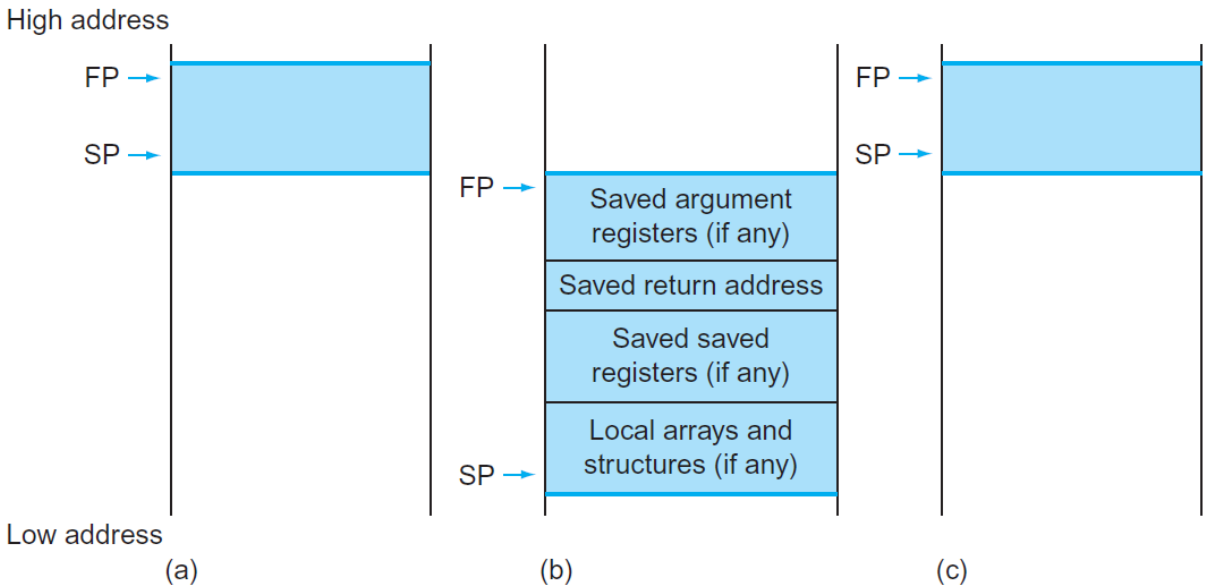
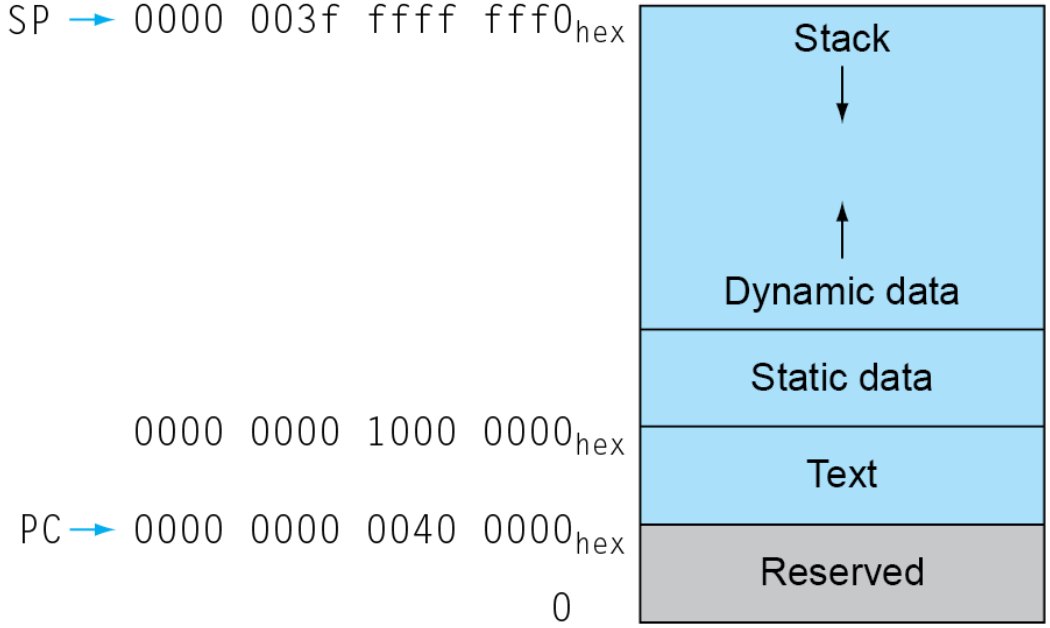


FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call.

2.8 计算机硬件对过程的支持

- 内存布局
  - 保留区域
  - 代码段
  - 静态数据/全局变量
    - static
    - constant arrays and strings
    - x3/gp为基址进行寻址的数据
  - 动态数据（堆，heap）
    - C：malloc
    - java：new
  - 栈
    - 在高级语言中，自动保存



- 堆与栈相向而长，以达到对内存的高效利用

2.8 计算机硬件对过程的支持

Check Yourself

Which of the following statements about C and Java is generally true?

1. C programmers manage data explicitly, while it's automatic in Java.
2. C leads to more pointer bugs and memory leak bugs than does Java.

2.9 人机交互

- ASCII码：
  - American Standard Code for Information Interchange
  - 其它编码：Unicode、UTF-8、UTF-16等
  - https://baike.baidu.com/item/ASCII

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

2.9 人机交互

1,000,000,000	
HEX	3B9A CA00
DEC	1,000,000,000
OCT	7 346 545 000
BIN	0011 1011 1001 1010 1100 1010 0000 0000

- 示例，用ASCII码表示10亿
  - 二进制存储：0011 1011 1001 1010 1100 1010 0000 0000，32bit
  - ASCII格式存储：31,30,30,30,30,30,30,30,30,30,10个字节
    - 以ASCII格式存储16进制的10亿呢?

ASCII versus Binary Numbers

EXAMPLE

We could represent numbers as strings of ASCII digits instead of as integers. How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?

ANSWER

One billion is 1,000,000,000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be $(10 \times 8)/32$ or 2.5. Beyond the expansion in storage, the hardware to add, subtract, multiply, and divide such decimal numbers is difficult and would consume more energy. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal computer is bizarre.

2.9 人机交互

- 字符存储方案如何选择？ 8bit/字符
- ASCII码以字节为单位进行存储
  - 充分利用存储空间
  - 需要指令能对字节进行操作，lb、sb等相关指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

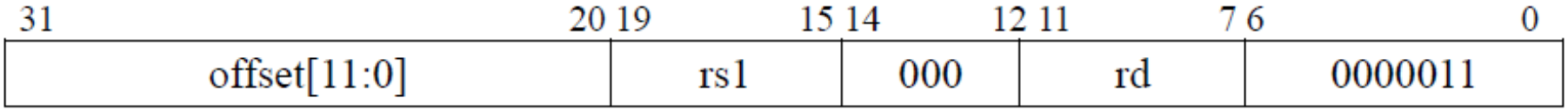
31 25	24 20	19 15	14 12	11 7	6 0	
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I csrrsi
csr			zimm	111	rd	1110011	I csrrci

2.9 人机交互

lb rd, offset(rs1)　　x[rd] = sext(M[x[rs1] + sext(offset)][7:0])

字节加载 *(Load Byte)*. I-type, RV32I and RV64I.

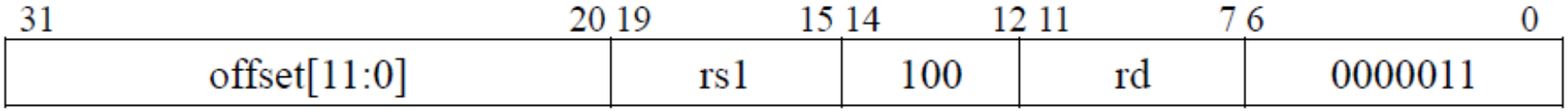
从地址 x[*rs1*] + *sign-extend*(*offset*)读取一个字节，经符号位扩展后写入 x[*rd*]。



lbu rd, offset(rs1)　　x[rd] = M[x[rs1] + sext(offset)][7:0]

无符号字节加载 *(Load Byte, Unsigned)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取一个字节，经零扩展后写入 x[*rd*]。

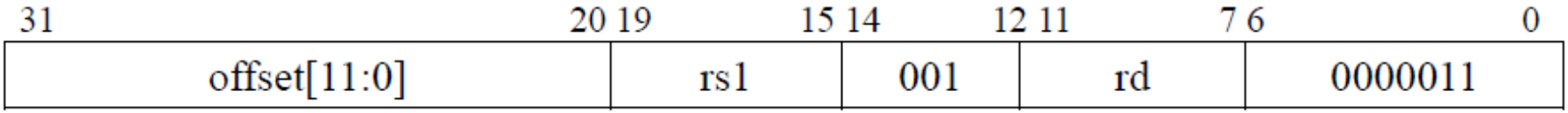


2.9 人机交互

lh rd, offset(rs1)　　x[rd] = sext(M[x[rs1] + sext(offset)][15:0])

半字加载 *(Load Halfword)*. I-type, RV32I and RV64I.

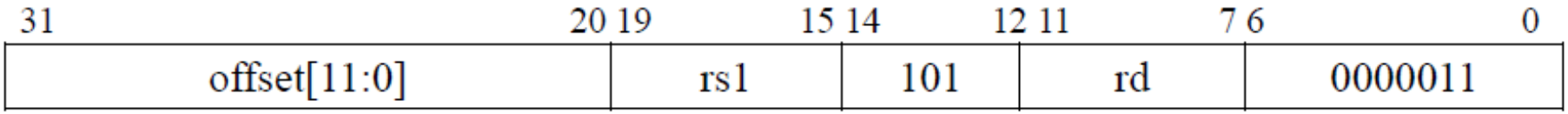
从地址 x[*rs1*] + *sign-extend*(*offset*)读取两个字节，经符号位扩展后写入 x[*rd*]。



lhu rd, offset(rs1)　　x[rd] = M[x[rs1] + sext(offset)][15:0]

无符号半字加载 *(Load Halfword, Unsigned)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取两个字节，经零扩展后写入 x[*rd*]。



2.9 人机交互

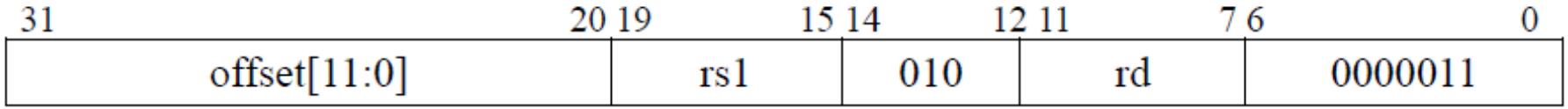
- RV32I中没有跟lw对应的lwu指令

lw rd, offset(rs1)　　x[rd] = sext(M[x[rs1] + sext(offset)][31:0])

字加载 *(Load Word)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取四个字节，写入 x[*rd*]。对于 RV64I，结果要进行符号位扩展。

压缩形式：**c.lwsp** rd, offset; **c.lw** rd, offset(rs1)

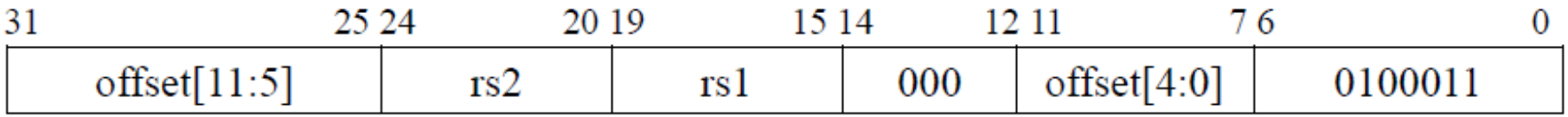


2.9 人机交互

sb rs2, offset(rs1)　　M[x[rs1] + sext(offset) = x[rs2][7:0]

存字节(Store Byte). S-type, RV32I and RV64I.

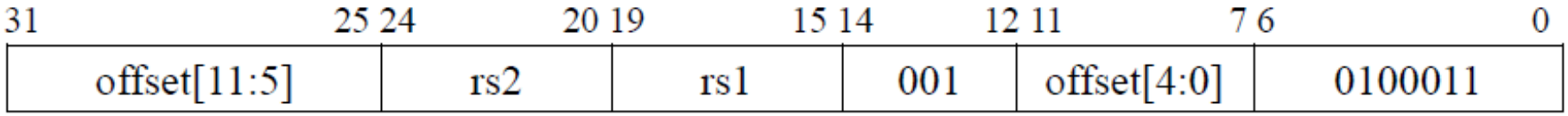
将 x[*rs2*]的低位字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。



sh rs2, offset(rs1)　　M[x[rs1] + sext(offset) = x[rs2][15:0]

存半字(Store Halfword). S-type, RV32I and RV64I.

将 x[*rs2*]的低位 2 个字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。



2.9 人机交互

- RV32I的访存相关指令
- 加载字节、半字、字：符号扩展
  - lb rd, offset(rs1)
  - lh rd, offset(rs1)
  - lw rd, offset(rs1)
- 加载字节、半字、字：零扩展
  - lbu rd, offset(rs1)
  - lhu rd, offset(rs1)
- 存储字节、半字、字: Store rightmost 8/16/32 bits
  - sb rs2, offset(rs1)
  - sh rs2, offset(rs1)
  - sw rs2, offset(rs1)

2.9 人机交互

- RV32I未涉及指令汇总

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	00	00000	1110011	I ecall
000000000000		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I csrrsi
csr		zimm	111	rd	1110011	I csrrci

2.9 人机交互

- 字符串存储及操作
  - printf(“Hello World !\n”);
- 存储方案选择
  - 字符串的第一个位置保留，用于表示字符串长度 ×
  - 附加带有字符串长度的变量 ×
  - 用一个特殊字符表示字符串的结束 √
    - 使用ASCII值为0的符号表示，null

2.9 人机交互

- 示例，字符串拷贝程序

```
void strcpy (char x[], char y[]){
        size_t i;
        i = 0;
        while ((x[i] = y[i]) !=    '\0'   ) /* copy & test byte */
        i += 1;
}
```

2.9 人机交互

- RISC-V汇编代码

```
strcpy:
     addi sp,sp,-8        // adjust stack for 1 doubleword
     sd   x19,0(sp)       // push x19
     add  x19,x0,x0       // i=0
L1:  add  x5,x19,x10      // x5 = addr of y[i]
     lbu  x6,0(x5)        // x6 = y[i]
     add  x7,x19,x10      // x7 = addr of x[i]
     sb   x6,0(x7)        // x[i] = y[i]
     beq  x6,x0,L2        // if y[i] == 0 then exit
     addi x19,x19,1       // i = i + 1
     jal  x0,L1           // next iteration of loop
L2:  ld   x19,0(sp)       // restore saved x19
     addi sp,sp,8         // pop 1 doubleword from stack
     jalr x0,0(x1)        // and return
```

2.9 人机交互

Check Yourself

I. Which of the following statements about characters and strings in C and Java is true?

1. A string in C takes about half the memory as the same string in Java.
2. Strings are just an informal name for single-dimension arrays of characters in C and Java.
3. Strings in C and Java use null (0) to mark the end of a string.
4. Operations on strings, like length, are faster in C than in Java.

II. Which type of variable that can contain $1{,}000{,}000{,}000_{ten}$ takes the most memory space?

1. `long long int` in C
2. `string` in C
3. `string` in Java

2.10 对大立即数的编址和寻址

- 如何向寄存器中写入一个32bit的数据?
  - load、store指令
  - addi + slli指令组合

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I csrrsi
csr			zimm	111	rd	1110011	I csrrci

2.10 对大立即数的编址和寻址

- 前面涉及的指令可操作12bit的立即数
- JAL指令可生成20bit立即数，但只能影响PC
  - pc += sext(offset)

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I csrrsi
csr			zimm	111	rd	1110011	I csrrci

2.10 对大立即数的编址和寻址

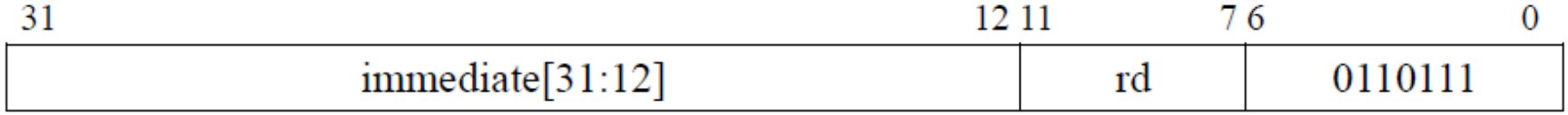
- LUI指令，load upper immediate，取立即数高位

lui rd, immediate　　x[rd] = sext(immediate[31:12] << 12)

高位立即数加载 *(Load Upper Immediate)*. U-type, RV32I and RV64I.

将符号位扩展的 20 位立即数 *immediate* 左移 12 位，并将低 12 位置零，写入 x[*rd*]中。

压缩形式：**c.lui** rd, imm



- lui x19, 976 // 0x003D0

0000 0000 0011 1101 0000	0000 0000 0000

- addi x19,x19,1280 //0x500

0000 0000 0011 1101 0000	0101 0000 0000

  - addi指令使用符号扩展，因此立即数的bit11表示的是符号位
  - 对于bit11为1的立即数加载，需要考虑符号位的问题
- li x19, 0x003D0500
  - 伪指令,汇编器会将该指令拆成上述两条真实的汇编指令

2.10 对大立即数的编址和寻址

- 分支中的寻址
- bne x10, x11, 2000 #2000（十进制）= 0111 1101 0000
  - 该指令中，12bit的立即数采用特殊的组织方式
  - 数据通路设计得以简化，组装更加复杂
  - 寻址范围：PC-4096 ~ PC+4094(偶数地址)
  - Q：请根据bne指令格式写出上述指令的十六进制数值

bne rs1, rs2, offset　　if (rs1 ≠ rs2) pc += sext(offset)

不相等时分支 *(Branch if Not Equal)*. B-type, RV32I and RV64I.

若寄存器 x[*rs1*]和寄存器 x[*rs2*]的值不相等，把 *pc* 的值设为当前值加上符号位扩展的偏移 *offset*。

压缩形式：**c.bnez** rs1, offset

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	001	offset[4:1|11]	1100011

2.10 对大立即数的编址和寻址

- 分支中的寻址-续
- JAL x0, 2000 #2000（十进制=0000 00000111 1101 0000）
  - 该指令中，20bit立即数采用特殊的组织方式
  - 寻址范围：PC ± 2^{20}
- Q：请根据jal指令格式写出上述指令的十六进制数值

jal rd, offset　　x[rd] = pc+4; pc += sext(offset)

跳转并链接 *(Jump and Link)*. J-type, RV32I and RV64I.

把下一条指令的地址(*pc*+4)，然后把*pc*设置为当前值加上符号位扩展的*offset*。*rd*默认为x1。

压缩形式：**c.j** offset；**c.jal** offset

31 — 12	11 — 7	6 — 0
offset[20|10:1|11|19:12]	rd	1101111

2.10 对大立即数的编址和寻址

- PC相对寻址
  - 一种寻址方式，地址为PC和指令中的常量之和
  - eg：B-type、J-type
  - B-type：条件分支，$\pm 2^{12}$个字节，±4KByte
  - J-type：无条件分支，$\pm 2^{20}$个字节，±1MByte
- 长距离跳转
  - 场景：过程/函数调用
  - 通过使用双指令序列来实现
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

```
lui x19, 976
jalr x0, 1280(x19)
```

2.10 对大立即数的编址和寻址

EXAMPLE

Showing Branch Offset in Machine Language

The *while* loop on page 94 was compiled into this RISC-V assembler code:

```
Loop:slli x10, x22, 3     // Temp reg x10 = i * 8
     add  x10, x10, x25   // x10 = address of save[i]
     ld   x9, 0(x10)      // Temp reg x9 = save[i]
     bne  x9, x24, Exit   // go to Exit if save[i] != k
     addi x22, x22, 1     // i = i + 1
     beq  x0, x0, Loop    // go to Loop
Exit:
```

ANSWER

If we assume we place the loop starting at location 80000 in memory, what is the RISC-V machine code for this loop?

Address	Instruction					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

2.10 对大立即数的编址和寻址

EXAMPLE

Branching Far Away

Given a branch on register x10 being equal to zero,

```
beq    x10, x0, L1
```

ANSWER

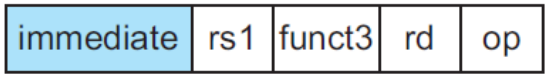
replace it by a pair of instructions that offers a much greater branching distance. These instructions replace the short-address conditional branch:

```
      bne      x10, x0, L2
      jal      x0, L1
L2:
```

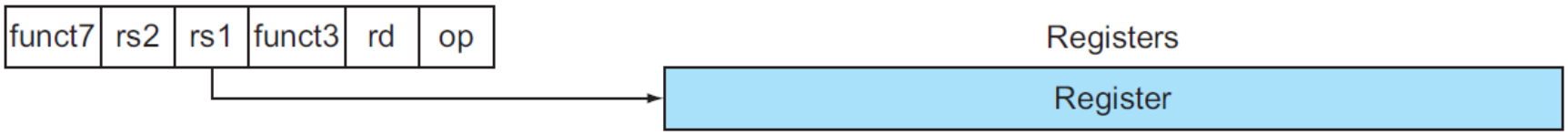
2.10 对大立即数的编址和寻址

- RISC-V寻址模式总结

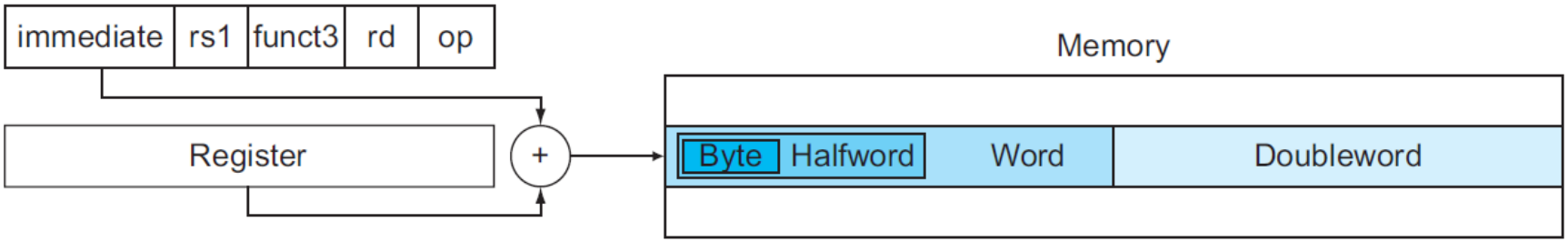
1. Immediate addressing



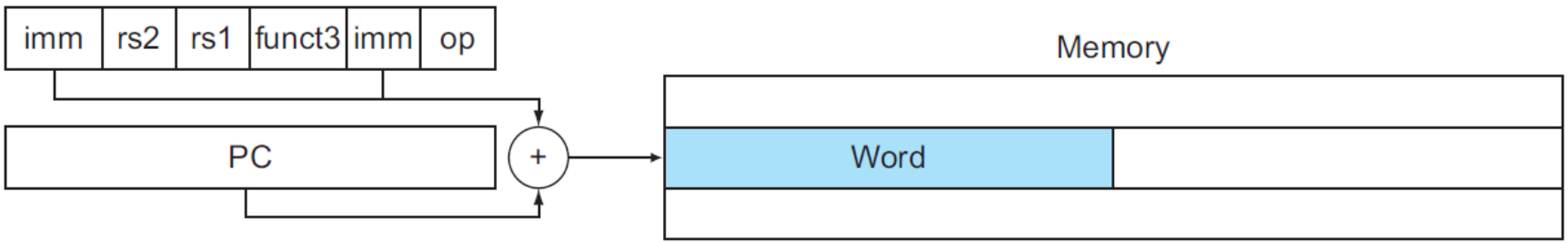
2. Register addressing



3. Base addressing



4. PC-relative addressing



2.10 对大立即数的编址和寻址

RISC-V指令格式汇总

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

2.10 对大立即数的编址和寻址

练习

Decoding Machine Code

What is the assembly language statement corresponding to this machine instruction?

00578833_{hex}

The first step is converting hexadecimal to binary:

0000 0000 0101 0111 1000 1000 0011 0011

To know how to interpret the bits, we need to determine the instruction format, and to do that we first need to determine the opcode. The opcode is the rightmost 7 bits, or 0110011. Searching Figure 2.20 for this value, we see that the opcode corresponds to the R-type arithmetic instructions. Thus, we can parse the binary format into fields listed in Figure 2.21:

funct7	rs2	rs1	funct3	rd	opcode
0000000	00101	01111	000	10000	0110011

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	ld	0000011	011	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	lwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

2.10 对大立即数的编址和寻址

Check Yourself

I. What is the range of byte addresses for conditional branches in RISC-V (K = 1024)?

1. Addresses between 0 and 4K – 1
2. Addresses between 0 and 8K – 1
3. Addresses up to about 2K before the branch to about 2K after
4. Addresses up to about 4K before the branch to about 4K after

II. What is the range of byte addresses for the jump-and-link instruction in RISC-V (M = 1024K)?

1. Addresses between 0 and 512K – 1
2. Addresses between 0 and 1M – 1
3. Addresses up to about 512K before the branch to about 512K after
4. Addresses up to about 1M before the branch to about 1M after

2.11 指令与并行性：同步

- 数据竞争：data race
  - 如果来自两个不同的线程的访存请求访问同一个位置，至少有一个是写，且连续出现，那么这两次存储访问形成了数据竞争
  - 解决办法：通过加锁（lock）和解锁（unlock）同步操作创建只有单个处理器（或进程）可以操作的区域（也称互斥区）
  - 在硬件上，通过原子交换原语来实现
- RISC-V相关指令
  - 保留加载双字指令：lr.d
  - 条件存储双字指令：sc.d

```
again:lr.d x10, (x20)          // load-reserved
      sc.d x11, x23, (x20)     // store-conditional
      bne  x11, x0, again      // branch if store fails
      addi x23, x10, 0         // put loaded value in x23
```

  - 在lr.d和sc.d指令中间加入其它指令构成锁

```
      addi x12, x0, 1       // copy locked value
again: lr.d  x10, (x20)     // load-reserved to read lock
      bne   x10, x0, again  // check if it is 0 yet
      sc.d  x11, x12, (x20) // attempt to store new value
      bne   x11, x0, again  // branch if store fails
```

We release the lock just using a regular store to write 0 into the location:

```
sd x0, 0(x20)     // free lock by writing 0
```

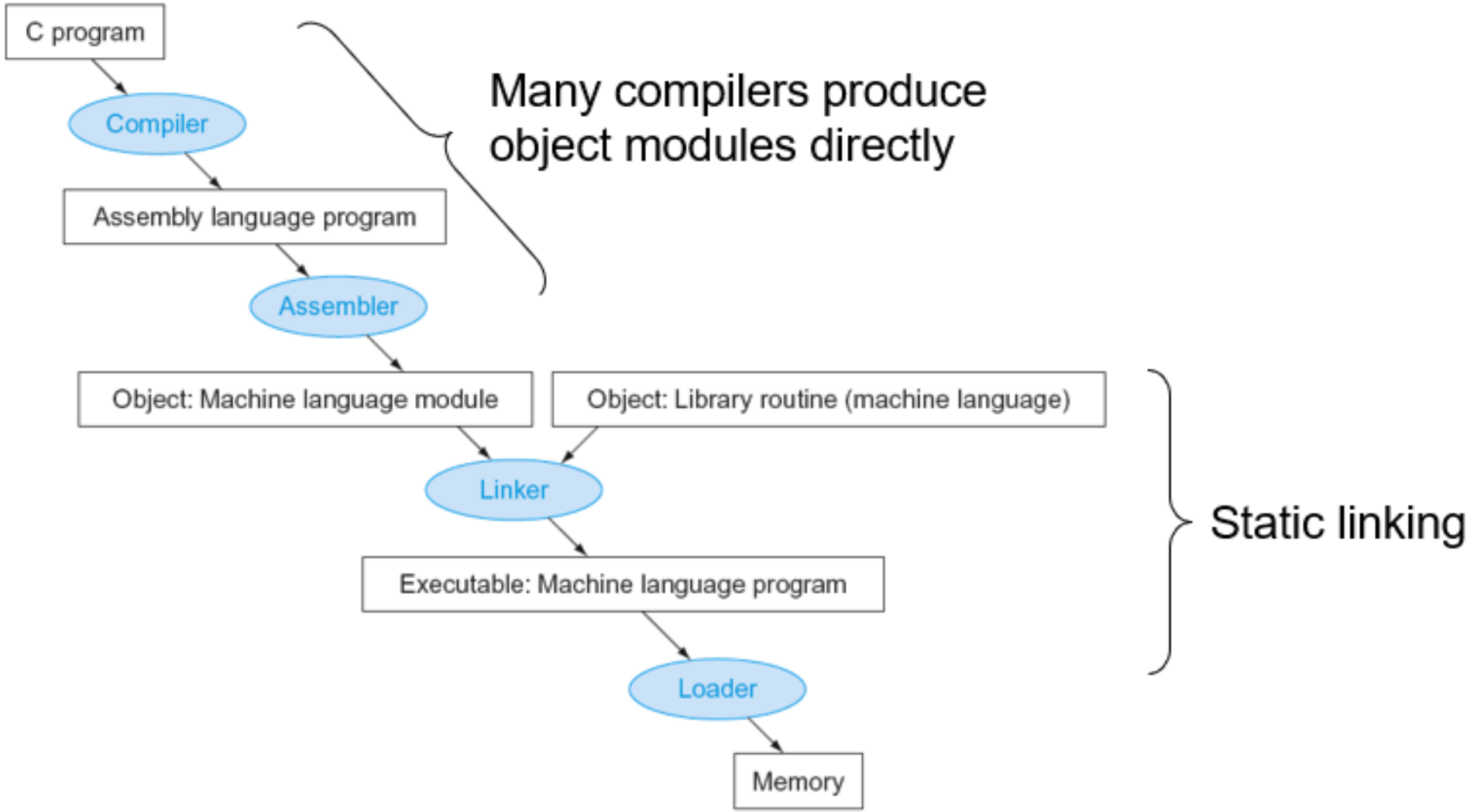
2.11 指令与并行性：同步

Check Yourself

When do you use primitives like load-reserved and store-conditional?

1. When cooperating threads of a parallel program need to synchronize to get proper behavior for reading and writing shared data.
2. When cooperating processes on a uniprocessor need to synchronize for reading and writing shared data.

2.12 翻译并启动程序


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

2.12 翻译并启动程序

- 编译器
  - 将C语言转换成汇编语言
  - 汇编语言：一种能被翻译为二进制机器语言的符号语言
  - 高级语言 VS 汇编语言
    - 代码行数更少
    - 编程效率更高
    - 可读性更强
    - 可移植性更高
- 汇编器
  - 将汇编语言程序转换为机器语言模块（目标文件，object file）
  - 可以处理机器指令的常见变体——伪指令
    - 伪指令：硬件不需要实现这些指令，使用伪指令可以简化程序转化和编程

```
li x9, 123          ⟶   addi x9, x0, 123
mv x10, x11         ⟶   addi x10, x11, 0
and x9, x10, 15     ⟶   andi x9, x10, 15
```

2.12 翻译并启动程序

- UNIX系统的目标文件通常包含6个部分
  - 目标文件头
  - 代码段
  - 静态数据段
  - 重定位信息
  - 符号表
  - 调试信息

2.12 翻译并启动程序

- 链接器
  - 程序进行少量修改后进行完整的重新编译是一种资源浪费
  - 独立编译和汇编每个过程
  - 链接器将各个独立的目标模块合并
  - 定义：也叫链接编辑器，是一个系统程序，将独立汇编的机器语言程序组合起来，并解析所有未定义的标签，最终生成可执行文件
- 链接器的工作步骤
  - 将代码和数据模块按符号特征放入内存
  - 决定数据和指令标签的地址
  - 修正内部和外部引用
- 可执行文件
  - 一种具有目标文件格式的功能程序，不包含未解析的引用
  - 可以包含符号表和调试信息，剥离的可执行文件不包含这些信息，可以包括用于加载器的重定位信息

2.12 翻译并启动程序

- 例题：目标文件的链接

2.12 翻译并启动程序

- 加载器
  - 是一个系统程序，将存储在硬盘上的目标程序放到内存中以准备执行
- 加载器执行流程
  - 读取可执行文件首部，以确定文本段和数据段的大小
  - 为正文和数据创建足够大的地址空间
  - 将可执行文件中的指令和数据复制到内存中
  - 将主程序的参数（如果有）复制到栈顶
  - 初始化处理器寄存器，并将栈指针指向第一个空闲位置
  - 跳转到启动例程，将参数复制到参数寄存器中，并调用程序的主例程。当主例程返回时，启动例程通过exit系统调用终止程序

2.12 翻译并启动程序

- 静态链接库的优点
  - 速度快，易理解
- 静态链接库的缺点
  - 库例程成为可执行代码的一部分，不利于更新、升级
  - 会加载程序用到的所有库的所有例程，因此会占用较大空间
    - eg：linux系统，RISC-V标准C库大小为1.5MB
- 动态链接库，dynamically linked libraries，DLL
  - 定义：在执行期间链接到程序的库例程
  - 要求程序代码可重新定位
  - 避免静态链接库引用导致的镜像膨胀
  - 自动获取新的库版本

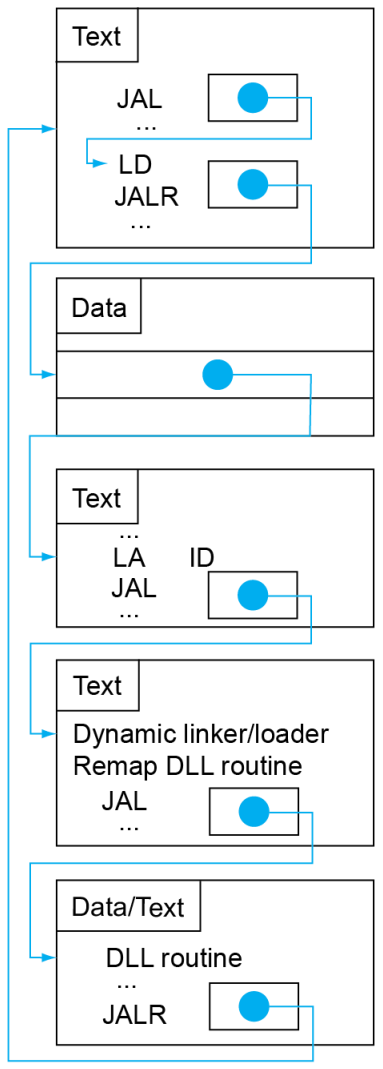
2.12 翻译并启动程序

Indirection table

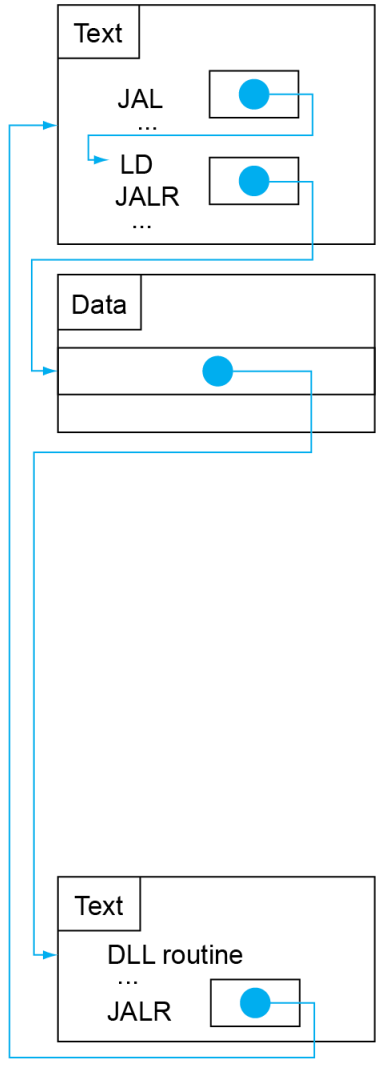
Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code



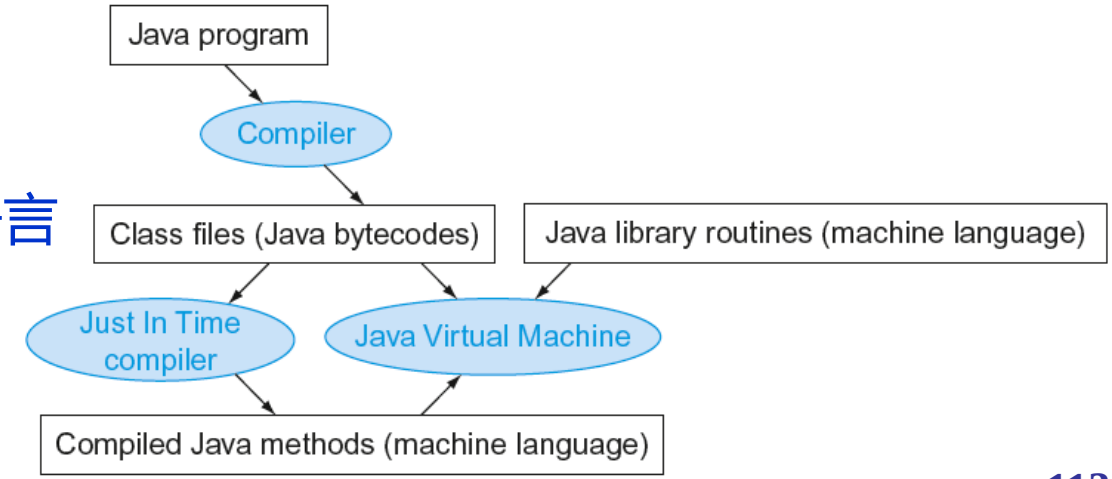
(a) First call to DLL routine



(b) Subsequent calls to DLL routine

2.12 翻译并启动程序

- 启动JAVA程序
  - sun公司开发的一门面向对象的编程语言
  - 不是编译成目标计算机的汇编语言
  - 而是编译成易于解释的指令：JAVA字节码
- JAVA字节码
  - 为解释JAVA程序而设计的指令系统只的指令
- JAVA虚拟机：JVM
  - 解释JAVA字节码的程序
- JIT：即时编译器
  - 在运行时将已解释过的代码片段翻译为宿主机上的机器语言



2.12 翻译并启动程序

Check Yourself

Which of the advantages of an interpreter over a translator was the most important for the designers of Java?

1. Ease of writing an interpreter
2. Better error messages
3. Smaller object code
4. Machine independence

2.13 以C排序程序为例的汇总整理

- C →→ 汇编
  - 为程序中的变量分配寄存器
  - 为过程体生成汇编代码
  - 保存过程调用期间的寄存器
- swap过程
  - 参数v、k使用x10、x11保存
  - 变量temp使用x5保存

```
void swap(long long int v[], size_t k)
{
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

  - 将v[k]的地址存入x6
  - 将v[k]的数据存入x5
  - 将v[k+1]的数据存入x7
  - 将x7的数据存入v[k]
  - 将x5的数据存入v[k+1]
  - 返回

```
swap:
  slli   x6, x11, 3   // reg x6 = k * 8
  add    x6, x10, x6  // reg x6 = v + (k * 8)
  ld     x5, 0(x6)    // reg x5 (temp) = v[k]
  ld     x7, 8(x6)    // reg x7 = v[k + 1]
  sd     x7, 0(x6)    // v[k] = reg x7
  sd     x5, 8(x6)    // v[k+1] = reg x5 (temp)
  jalr   x0, 0(x1)    // return to calling routine
```

2.13 以C排序程序为例的汇总整理

- sort过程

```
void sort (long long int v[], size_t int n)
{
    size_t i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v,j);
        }
    }
}
```

Saving registers

```
sort:   addi sp, sp, -40        # make room on stack for 5 registers
        sd x1, 32(sp)           # save return address on stack
        sd x22, 24(sp)          # save x22 on stack
        sd x21, 16(sp)          # save x21 on stack
        sd x20, 8(sp)           # save x20 on stack
        sd x19, 0(sp)           # save x19 on stack
```

Procedure body

Move parameters	`mv x21, x10` # copy parameter x10 into x21 `mv x22, x11` # copy parameter x11 into x22
Outer loop	`li x19, 0` # i = 0 `for1tst:bge x19, x22, exit1` # go to exit1 if i >= n
Inner loop	`addi x20, x19, -1` # j = i - 1 `for2tst:blt x20, x0, exit2` # go to exit2 if j < 0 `slli x5, x20, 3` # x5 = j * 8 `add x5, x21, x5` # x5 = v + (j * 8) `ld x6, 0(x5)` # x6 = v[j] `ld x7, 8(x5)` # x7 = v[j + 1] `ble x6, x7, exit2` # go to exit2 if x6 < x7
Pass parameters and call	`mv x10, x21` # first swap parameter is v `mv x11, x20` # second swap parameter is j `jal x1, swap` # call swap
Inner loop	`addi x20, x20, -1` j for2tst `j for2tst` # go to for2tst
Outer loop	`exit2: addi x19, x19, 1` # i += 1 `j for1tst` # go to for1tst

2.13 以C排序程序为例的汇总整理

- sort过程
  - 入栈操作
  - for循环展开
  - 过程调用
  - v in x10, n in x11, i in x19, j in x20

```
void sort (long long int v[], size_t int n)
{
    size_t i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v,j);
        }
    }
}
```

Saving registers

```
sort:   addi sp, sp, -40        # make room on stack for 5 registers
        sd x1, 32(sp)           # save return address on stack
        sd x22, 24(sp)          # save x22 on stack
        sd x21, 16(sp)          # save x21 on stack
        sd x20, 8(sp)           # save x20 on stack
        sd x19, 0(sp)           # save x19 on stack
```

Procedure body

Move parameters	`mv x21, x10` # copy parameter x10 into x21 `mv x22, x11` # copy parameter x11 into x22
Outer loop	`li x19, 0` # i = 0 `for1tst:bge x19, x22, exit1` # go to exit1 if i >= n
Inner loop	`addi x20, x19, -1` # j = i - 1 `for2tst:blt x20, x0, exit2` # go to exit2 if j < 0 `slli x5, x20, 3` # x5 = j * 8 `add x5, x21, x5` # x5 = v + (j * 8) `ld x6, 0(x5)` # x6 = v[j] `ld x7, 8(x5)` # x7 = v[j + 1] `ble x6, x7, exit2` # go to exit2 if x6 < x7
Pass parameters and call	`mv x10, x21` # first swap parameter is v `mv x11, x20` # second swap parameter is j `jal x1, swap` # call swap
Inner loop	`addi x20, x20, -1` j for2tst `j for2tst` # go to for2tst
Outer loop	`exit2: addi x19, x19, 1` # i += 1 `j for1tst` # go to for1tst

Restoring registers

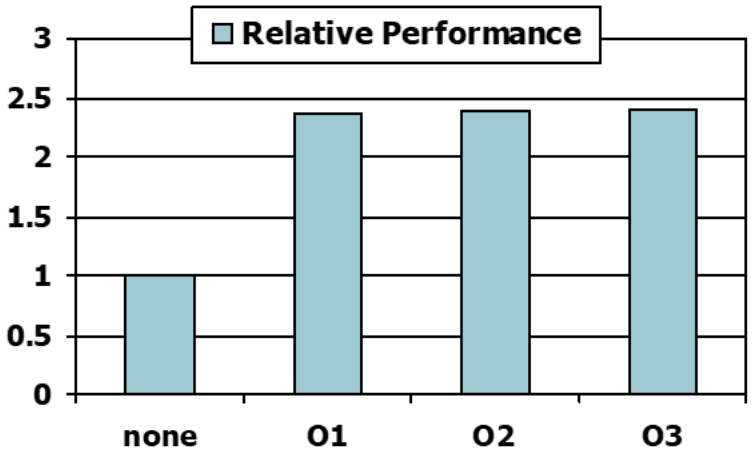
```
exit1:  ld x19, 0(sp)           # restore x19 from stack
        ld x20, 8(sp)           # restore x20 from stack
        ld x21, 16(sp)          # restore x21 from stack
        ld x22, 24(sp)          # restore x22 from stack
        ld x1, 32(sp)           # restore return address from stack
        addi sp, sp, 40         # restore stack pointer
```

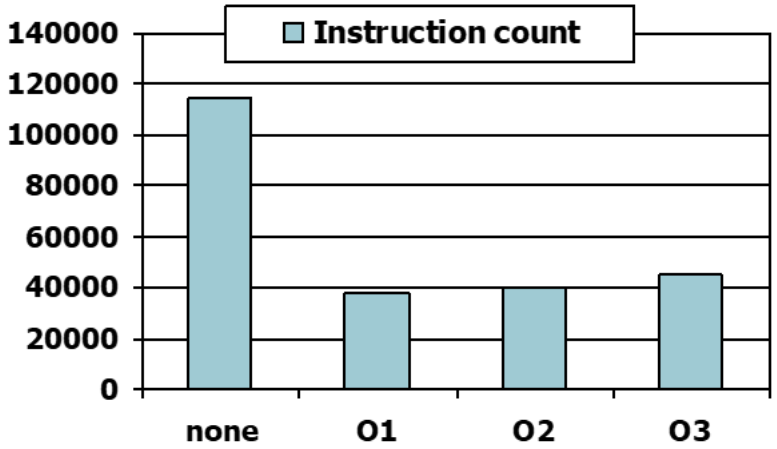
Procedure return

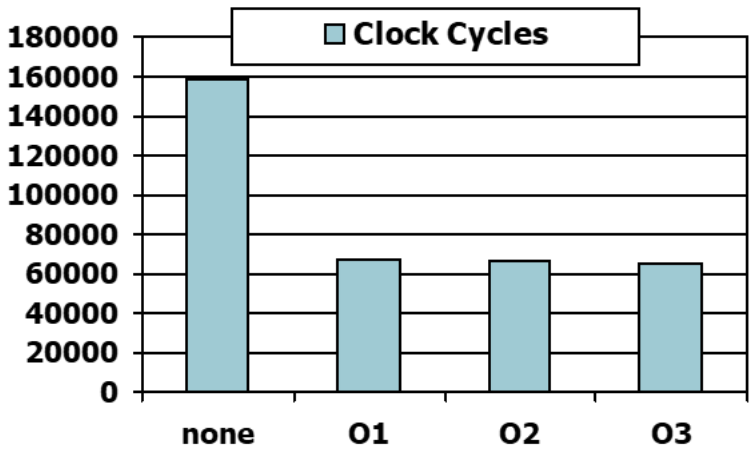
```
        jalr x0, 0(x1)          # return to calling routine
```

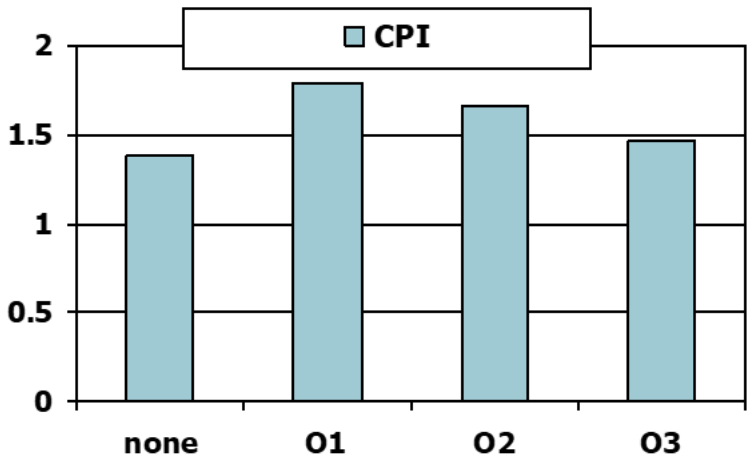
2.13 以C排序程序为例的汇总整理

Compiled with gcc for Pentium 4 under Linux

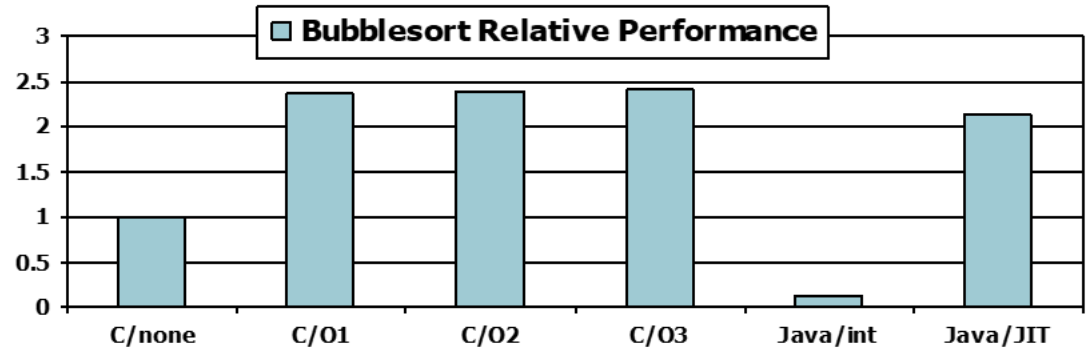


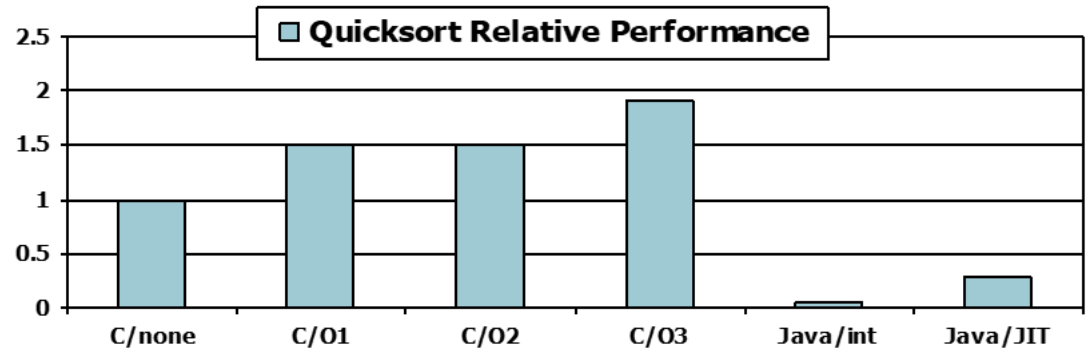


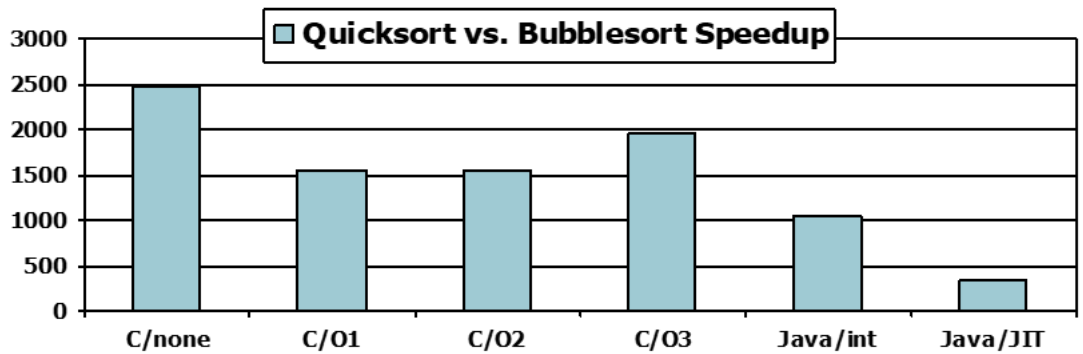




2.13 以C排序程序为例的汇总整理







2.13 以C排序程序为例的汇总整理

- 指令条数和CPI都不是独立评估程序性能的好的指标
- 编译优化对算法的具体实现相当敏感
- JAVA及时编译器（JIT）比JVM解释执行快很多
  - 有时可以接近经过优化的C代码的性能
- 愚蠢的算法无可救药

2.13 以C排序程序为例的汇总整理

- 指令条数和CPI都不是独立评估程序性能的好的指标
- 编译优化对算法的具体实现相当敏感
- JAVA及时编译器（JIT）比JVM解释执行快很多
  - 有时可以接近经过优化的C代码的性能
- 愚蠢的算法无可救药

2.14 数组与指针

- 数组索引
  - 索引编号乘以元素大小
  - 加上数组基地址
- 指针直接对应于内存地址
  - 可以避免索引的复杂性

2.14 数组与指针

- 示例：将一个数组全部清零

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
   li   x5,0        // i = 0
loop1:
   slli x6,x5,3     // x6 = i * 8
   add  x7,x10,x6   // x7 = address
                    // of array[i]
   sd   x0,0(x7)    // array[i] = 0
   addi x5,x5,1     // i = i + 1
   blt  x5,x11,loop1  // if (i<size)
                      // go to loop1
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
   mv x5,x10        // p = address
                    // of array[0]
   slli x6,x11,3    // x6 = size * 8
   add x7,x10,x6    // x7 = address
                    // of array[size]
loop2:
   sd x0,0(x5)      // Memory[p] = 0
   addi x5,x5,8     // p = p + 8
   bltu x5,x7,loop2
                    // if (p<&array[size])
                    // go to loop2
```

2.14 数组与指针

- 数组与指针的比较
  - 数组版本在循环内必须具有“乘”和“加”操作，以计算地址
  - 数组版本要求shift操作位于循环内
    - i累加
    - 根据元素大小进行移位：i左移
    - 与基地址相加，得到元素地址
  - 编译器优化可以为数组产生与指针版本同样好的效果

2.14 数组与指针

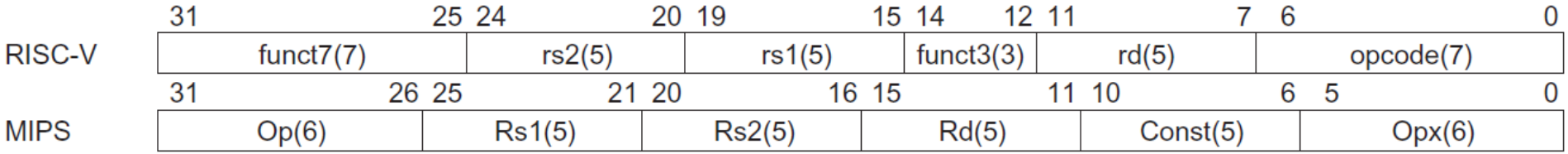
- 数组与指针的比较
  - 数组版本在循环内必须具有“乘”和“加”操作，以计算地址
  - 数组版本要求shift操作位于循环内
    - i累加
    - 根据元素大小进行移位：i左移
    - 与基地址相加，得到元素地址
  - 编译器优化可以为数组产生与指针版本同样好的效果

2.15 编译C语言和解释JAVA语言

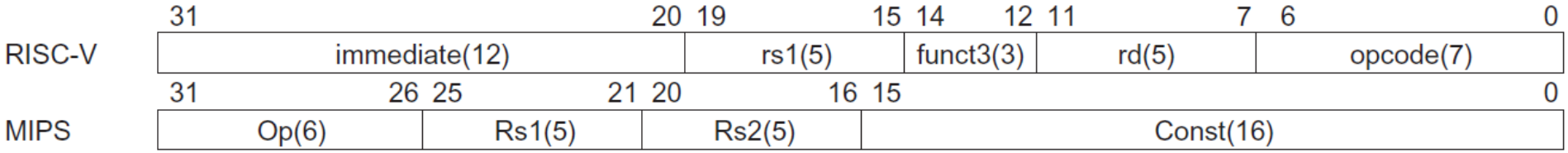
- 略

2.16 实例：MIPS指令

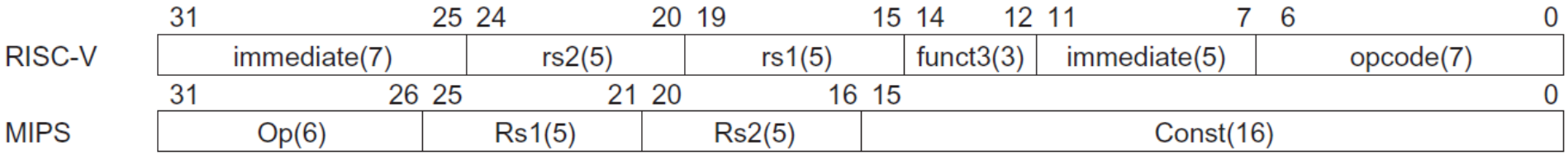
Register-register



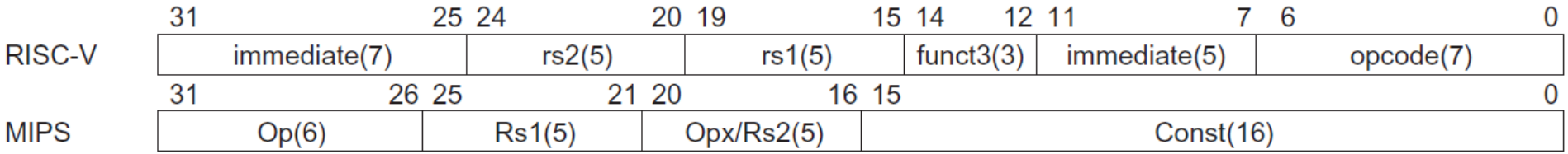
Load



Store



Branch



2.16 实例：MIPS指令

- RISC-V与MIPS的相同
  - 所有指令都是32bit位宽
  - 有32个通用寄存器，其中reg0硬连线为0
  - 通过Load、Store指令访问内存
  - 没有同时加载或存储多个寄存器的指令
  - 具有等于零跳转和不等于零跳转的分支指令
  - 指令系统的寻址模式适用于所有字长
- RISC-V与MIPS的不同
  - 除相等或不等外的条件分支： <, <=, >, >=
  - RISC-V仅提供分支指令来比较两个寄存器
  - MIPS的比较指令可将寄存器置0或置1，RISC-V: blt, bge, bltu, bgeu
  - MIPS的小于比较指令存在有符号和无符号两种：slt、sltu
    - Then use beq, bne to complete the branch
  - 完整的MIPS指令系统远大于RISC-V基础指令集（RV32I：47条）

2.17 实例：x86指令

- Evolution with backward compatibility
- 8080 (1974): 8-bit microprocessor
  - Accumulator, plus 3 index-register pairs
- 8086 (1978): 16-bit extension to 8080
  - Complex instruction set computer(CISC)
- 8087 (1980): floating-point coprocessor
  - Adds FP instructions and register stack
- 80286 (1982): 24-bit addresses, MMU
  - Segmented memory mapping and protection
- 80386 (1985): 32-bit extension (now IA-32)
  - Additional addressing modes and operations
  - Paged memory mapping as well as segments

2.17 实例：x86指令

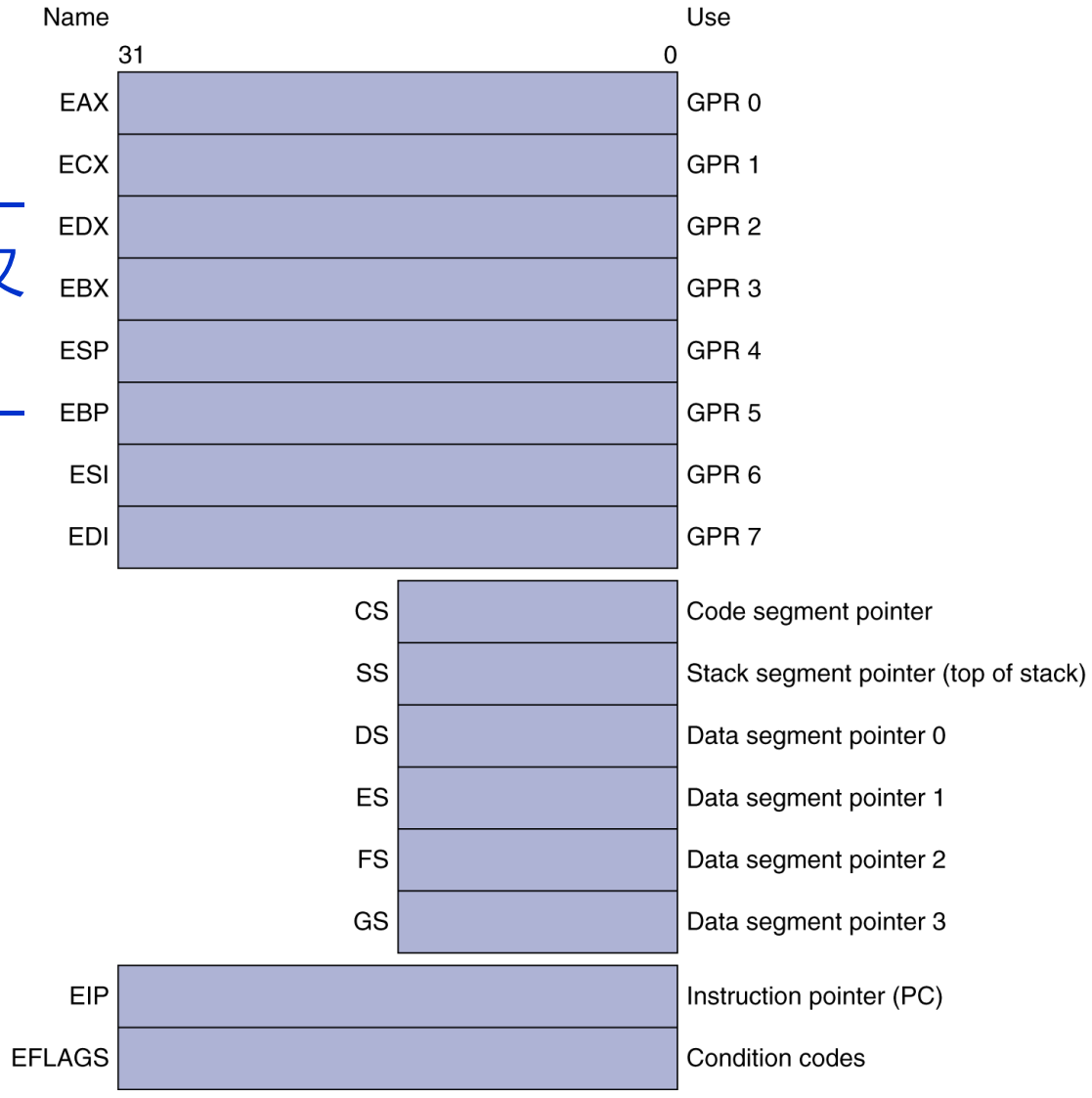
- Further evolution…
- i486 (1989): pipelined, on-chip caches and FPU
  - Compatible competitors: AMD, Cyrix, …
- Pentium (1993): superscalar, 64-bit datapath
  - Later versions added MMX (Multi-Media eXtension) instructions
  - The infamous FDIV bug
- Pentium Pro (1995), Pentium II (1997)
  - New microarchitecture (see Colwell, The Pentium Chronicles)
- Pentium III (1999)
  - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
  - New microarchitecture
  - Added SSE2 instructions

2.17 实例：x86指令

- And further…
- AMD64 (2003): extended architecture to 64 bits
- EM64T – Extended Memory 64 Technology (2004)
  - AMD64 adopted by Intel (with refinements)
  - Added SSE3 instructions
- Intel Core (2006)
  - Added SSE4 instructions, virtual machine support
- AMD64 (announced 2007): SSE5 instructions
  - Intel declined to follow, instead…
- Advanced Vector Extension (announced 2008)
  - Longer SSE registers, more instructions
- If Intel did not extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

2.17 实例：x86指令

- x86的寄存器
  - 8个通用寄存器
  - 算术和逻辑运算中必须有一个操作数即为源操作数，又是目的操作数
  - 任何一条指令都有可能有一个操作数在存储器中



2.17 实例：x86指令

- 算术、逻辑和数据传输指令允许的操作数组合情况
  - 立即数可以为8bit、16bit、32bit
  - 寄存器可以是14个主要寄存器（8个32bit，6个16bit）中的任意一个
  - 唯一缺少的是存储器-存储器模式

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

2.17 实例：x86指令

x86 32bit寻址模式汇总

Mode	Description	Register restrictions	RISC-V equivalent
Register indirect	Address is in a register.	Not ESP or EBP	`ld x10, 0(x11)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP	`ld x10, 40(x11)`

Base plus scaled index	The address is Base + (2^{Scale} × Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`slli x12, x12, 3`
			`add x11, x11, x12`
			`ld x10, 0(x11)`

Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} × Index) + Displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`slli x12, x12, 3`
			`add x11, x11, x12`
			`ld x10, 40(x11)`

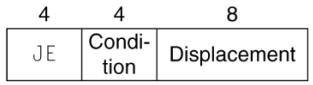
- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = R_{base} + 2^{scale} × R_{index} (scale = 0, 1, 2, or 3)
  - Address = R_{base} + 2^{scale} × R_{index} + displacement

2.17 实例：x86指令

- 指令长度不统一
  - 一个操作数：1byte~15byte
- 后缀字节指定寻址模式
- 前缀字节修改操作
  - 操作数长度、重复、锁定

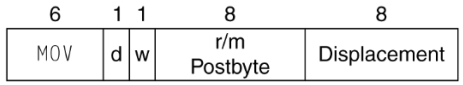
Instruction	Function
je name	if equal(condition code) {EIP=name}; EIP-128 <= name < EIP+128
jmp name	EIP=name
call name	SP=SP-4; M[SP]=EIP+5; EIP=name;
movw EBX,[EDI+45]	EBX=M[EDI+45]
push ESI	SP=SP-4; M[SP]=ESI
pop EDI	EDI=M[SP]; SP=SP+4
add EAX,#6765	EAX= EAX+6765
test EDX,#42	Set condition code (flags) with EDX and 42
movsl	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

a. JE EIP + displacement

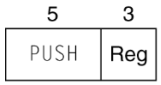


b. CALL

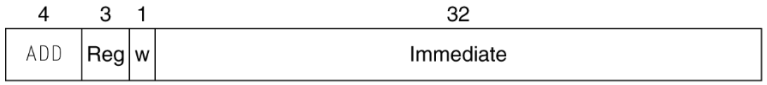
c. MOV EBX, [EDI + 45]



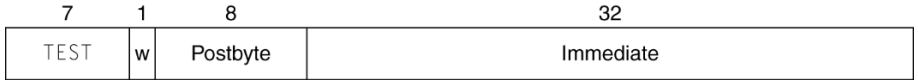
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



2.17 实例：x86指令

- 复杂的指令集使实现变得困难
- 硬件将指令转换为更简单的微操作
  - 简单指令：1-1
  - 复杂指令：1–多个
- 类似RISC的微引擎
- 市场份额使这在经济上可行
- 性能与RISC相当
  - 编译器避免复杂的指令

2.18 RISC-V指令系统的剩余部分

Instruction	Name	Format	Description
Add upper immediate to PC	auipc	U	Add 20-bit upper immediate to PC; write sum to register
Set if less than	slt	R	Compare registers; write Boolean result to register
Set if less than, unsigned	sltu	R	Compare registers; write Boolean result to register
Set if less than, immediate	slti	I	Compare registers; write Boolean result to register
Set if less than immediate, unsigned	sltiu	I	Compare registers; write Boolean result to register
Add word	addw	R	Add 32-bit numbers
Subtract word	subw	R	Subtract 32-bit numbers
Add word immediate	addiw	I	Add constant to 32-bit number
Shift left logical word	sllw	R	Shift 32-bit number left by register
Shift right logical word	srlw	R	Shift 32-bit number right by register
Shift right arithmetic word	sraw	R	Shift 32-bit number right arithmetically by register
Shift left logical word immedate	slliw	I	Shift 32-bit number left by immediate
Shift right logical word immediate	srliw	I	Shift 32-bit number right by immediate
Shift right arithmetic word immediate	sraiw	I	Shift 32-bit number right arithmetically by immediate

2.18 RISC-V指令系统的剩余部分

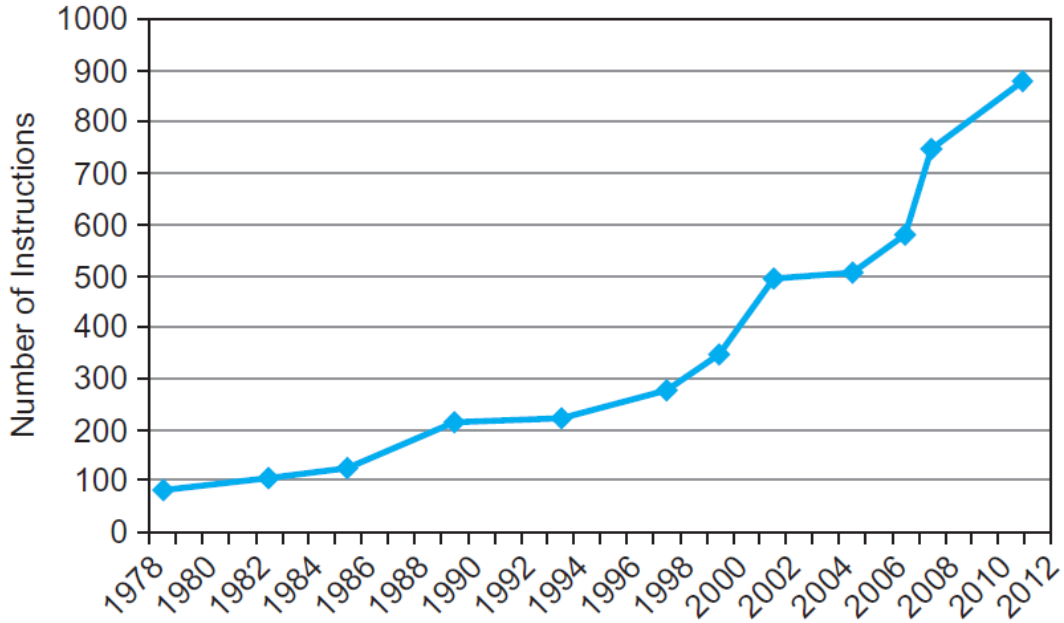
- RISC-V指令集架构采用模块化划分
  - RV64有：184条指令+13条系统指令（第五章）
  - 基本体系结构 + 扩展体系结构
  - I：基本体系结构 RV32I/RV64I
  - M：整数乘法、除法
  - A：原子操作
  - F：单精度浮点（第三章）
  - D：双精度浮点（第三章）
  - C：压缩指令

RISC-V Base and Extensions

Mnemonic	Description	Insn. Count
I	Base architecture	51
M	Integer multiply/divide	13
A	Atomic operations	22
F	Single-precision floating point	30
D	Double-precision floating point	32
C	Compressed instructions	36

2.19 谬误与陷阱

- 谬误1：强大的指令意味着更高的性能
- 谬误2：用汇编语言编程以获得最高性能
- 谬误3：商用计算机二进制兼容的重要性意味着成功的指令系统无须改变



- 陷阱1：在字节寻址的机器中，连续的字或双字地址相差不为1
- 陷阱2：在变量的定义过程外，不要将指针指向该变量

2.20 本章小结

- 存储程序计算机的两个原理
  - 指令的使用与数据没有区别
  - 两者都使用可变存储器
- 设计原则
  - 简单源于规整
  - 更少则更快
  - 优秀的设计需要适当的折中
- 加速经常性事件
- 软硬件接口层次
  - 编译器、汇编器、硬件
- RISC-V是典型的精简指令集架构

2.20 本章小结

RISC-V Instructions	Name	Format
Add	add	R
Subtract	sub	R
Add immediate	addi	I
Load doubleword	ld	I
Store doubleword	sd	S
Load word	lw	I
Load word, unsigned	lwu	I
Store word	sw	S
Load halfword	lh	I
Load halfword, unsigned	lhu	I
Store halfword	sh	S
Load byte	lb	I
Load byte, unsigned	lbu	I
Store byte	sb	S
Load reserved	lr.d	R
Store conditional	sc.d	R
Load upper immediate	lui	U
And	and	R
Inclusive or	or	R
Exclusive or	xor	R
And immediate	andi	I
Inclusive or immediate	ori	I
Exclusive or immediate	xori	I

RISC-V Instructions	Name	Format
Shift left logical	sll	R
Shift right logical	srl	R
Shift right arithmetic	sra	R
Shift left logical immediate	slli	I
Shift right logical immediate	srli	I
Shift right arithmetic immediate	srai	I
Branch if equal	beq	SB
Branch if not equal	bne	SB
Branch if less than	blt	SB
Branch if greater or equal	bge	SB
Branch if less, unsigned	bltu	SB
Branch if greatr/eq, unsigned	bgeu	SB
Jump and link	jal	UJ
Jump and link register	jalr	I

Pseudo RISC-V	Name	Real Instruction
Move	mv	addi
Load immediate	li	addi
Jump	j	jal
Load address	la	lui+addi

计算机组成原理
CH3_计算机的算术运算

卢建良
lujl@ustc.edu.cn
2022年春季学期

提纲

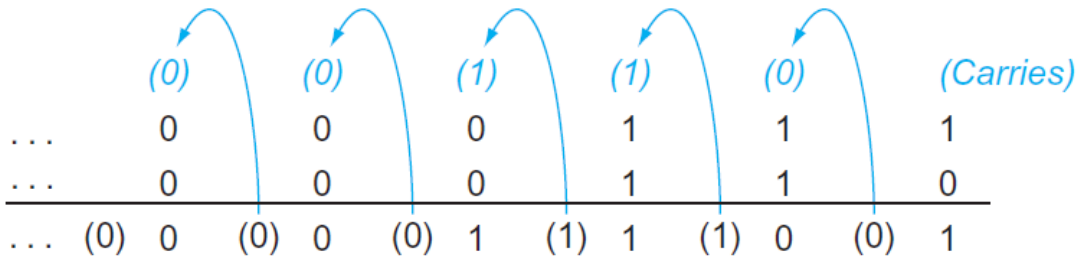
3.1 引言

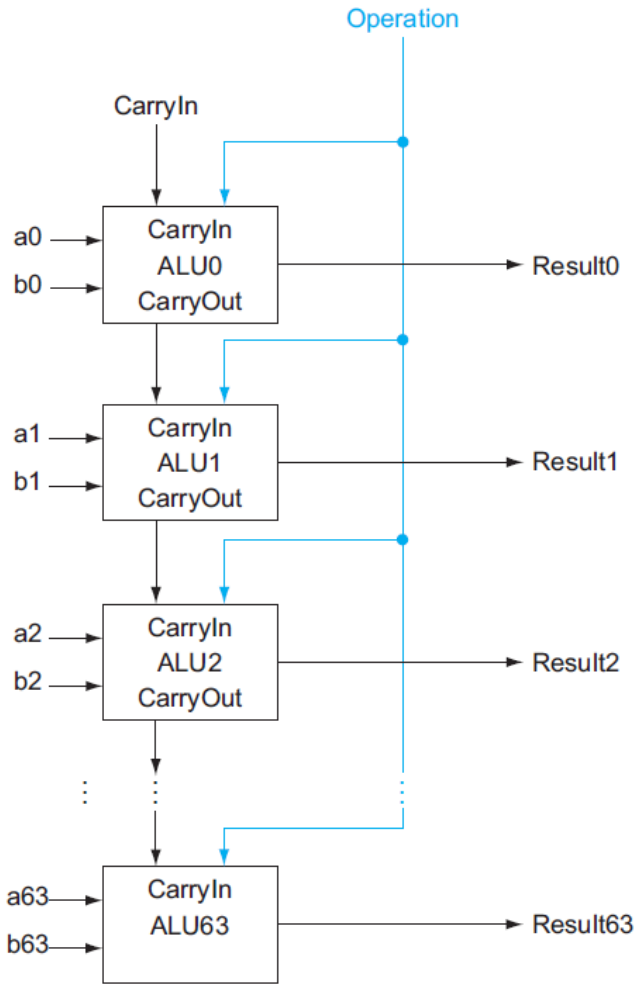
- 在32bit处理器中，可以表示2^{32}个整数
  - 无符号数：0 ~ 2^{32}-1
  - 有符号数：- 2^{31} ~ 2^{31}-1
- 其它数字该如何表示?
  - 如何表示小数和其它实数?
  - 如果运算产生了大到无法表示的数，该如何处理?
  - 硬件如何实现乘法、除法运算?
- 本章内容
  - 实数的表示
  - 算术的算法
  - 硬件结构

3.2 加法和减法

- 示例：7 + 6 = 13
  - 硬件实现可参考右图（附录A.5.2）

$$
\begin{array}{ll}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\
+ & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001101_{two} = 13_{ten}
\end{array}
$$





  - 当结果超出表示范围时，会发生溢出
  - 正数与负数相加，不会溢出
  - 两个正数相加，如果结果符号位为1，则溢出
  - 两个负数相加，如果结果符号位为0，则溢出

3.2 加法和减法

- 示例： 7 – 6 = 1
  - 方式1：通过正常的减法算式来计算

$$
\begin{array}{ll}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\
- & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}
\end{array}
$$

  - 方式2：用二进制补码来表示-6，计算7 + （-6）

$$
\begin{array}{ll}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\
+ & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111010_{two} = -6_{ten} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}
\end{array}
$$

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

3.2 加法和减法

- 图像和多媒体处理操作经常涉及8bit、16bit的数据
  - 使用64bit的加法器，同时将进位链进行适当分割
  - 实现8*8bit、4*16bit、2*32bit的向量操作
  - SIMD： single-instruction, multiple-data

- 饱和操作
  - 当计算溢出时，结果设置为最大正数或最小负数
  - eg：收音机音量旋钮

3.2 加法和减法

Check Yourself

Some programming languages allow two's complement integer arithmetic on variables declared byte and half, whereas RISC-V only has integer arithmetic operations on full words. As we recall from Chapter 2, RISC-V does have data transfer operations for bytes and halfwords. What RISC-V instructions should be generated for byte and halfword arithmetic operations?

1. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mul`, `div`, using `and` to mask result to 8 or 16 bits after each operation; then store using `sb`, `sh`.
2. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mul`, `div`; then store using `sb`, `sh`.

3.3 乘法

- RV32M指令集
  - 定义了可选的RV32M，它定义了整数乘法除法指令，总共8条指令

RV32M

multiply

multiply high { unsigned / signed unsigned }

{ divide / remainder } { unsigned }

31 25	24 20	19 15	14 12	11 7	6 0	
0000001	rs2	rs1	000	rd	0110011	R mul
0000001	rs2	rs1	001	rd	0110011	R mulh
0000001	rs2	rs1	010	rd	0110011	R mulhsu
0000001	rs2	rs1	011	rd	0110011	R mulhu
0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

- 示例：8 * 9 = 72（0x48）

Address	Code	Basic	
0x00000000	0x00800293	addi x5, x0, 8	1: li t0, 8
0x00000004	0x00900313	addi x6, x0, 9	2: li t1, 9
0x00000008	0x02628533	mul x10, x5, x6	3: mul a0, t0, t1

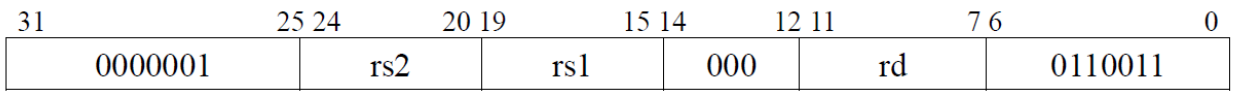
3.3 乘法

- 带符号的乘法
  - 将被乘数和乘数转换为正数进行计算，最后进行符号转换
- RISC-V中的乘法（RV64M）
  - mul，乘
  - mulh，乘法取高位
  - mulhsu，有符号/无符号乘法取高位

mul rd, rs1, rs2 $x[rd] = x[rs1] \times x[rs2]$

乘*(Multiply)*. R-type, RV32M and RV64M.

把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，乘积写入 x[*rd*]。忽略算术溢出。

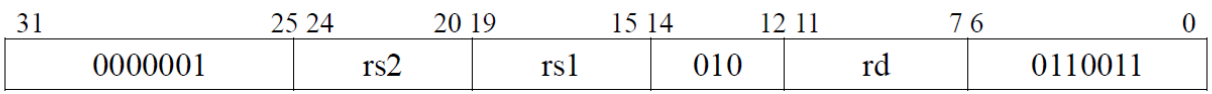


31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	000	rd	0110011

mulhsu rd, rs1, rs2 $x[rd] = (x[rs1]_s \times_u x[rs2]) \gg_s XLEN$

高位有符号−无符号乘*(Multiply High Signed-Unsigned)*. R-type, RV32M and RV64M.

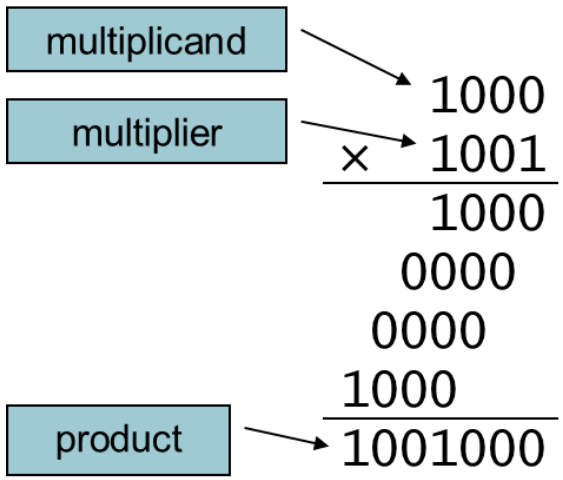
把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，x[*rs1*]为 2 的补码，x[*rs2*]为无符号数，将乘积的高位写入 x[*rd*]。



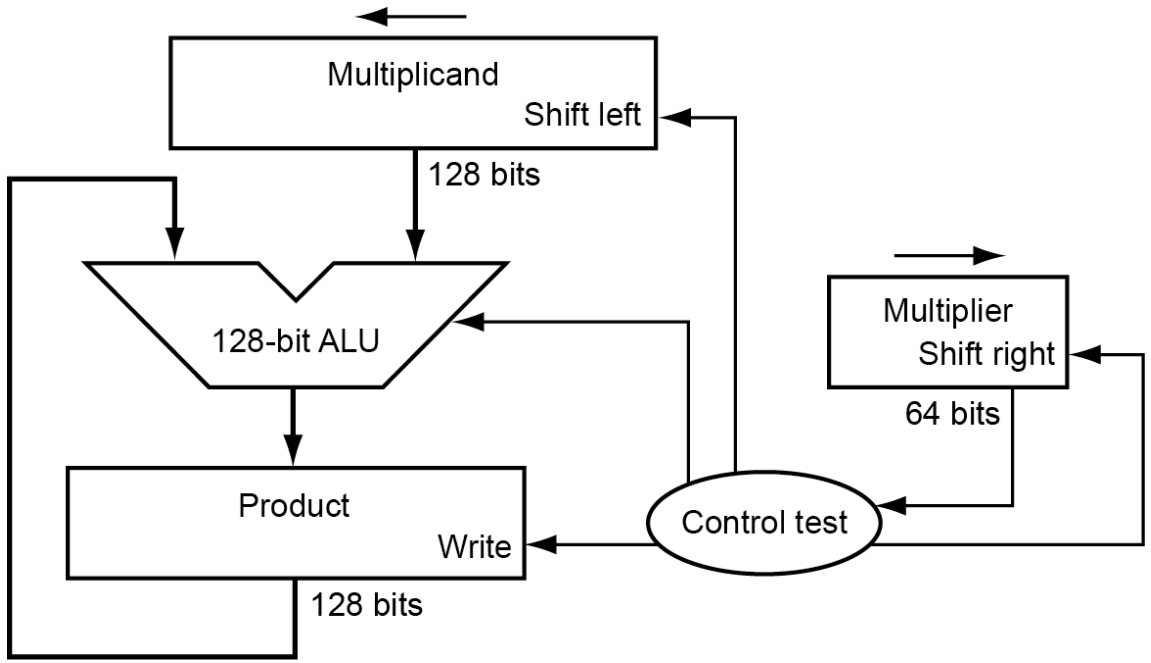
31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	010	rd	0110011

3.3 乘法

- 被乘数：multiplicand，乘法算式的第一个操作数
- 乘数：multiplier，乘法算式的第二个操作数
- 积：product，乘法算式的结果

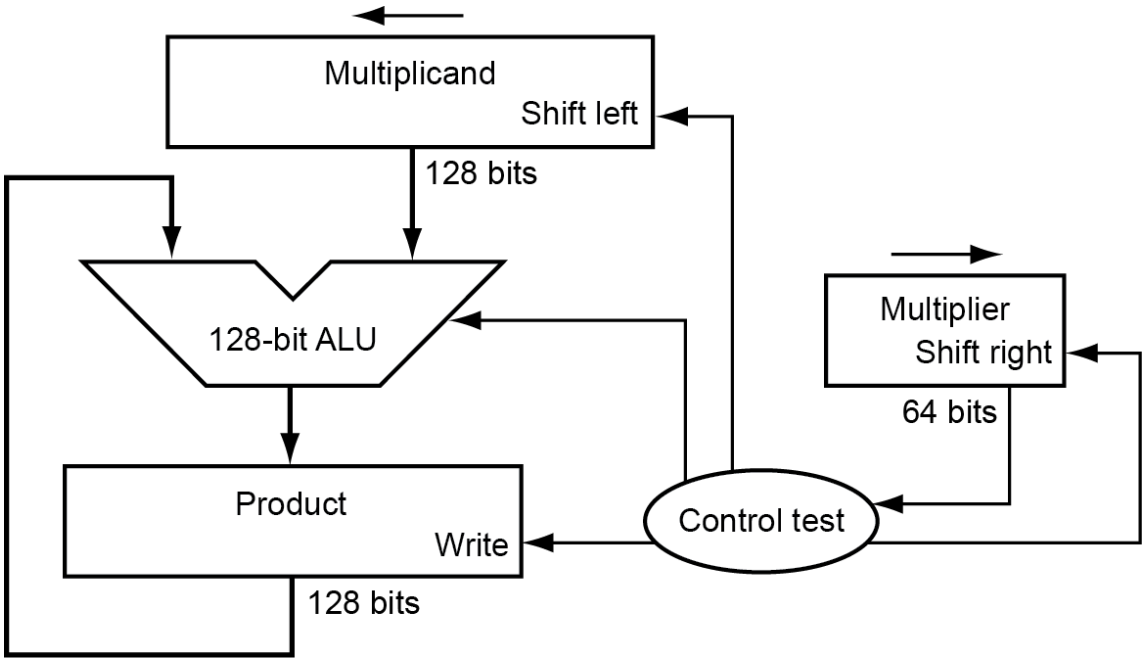


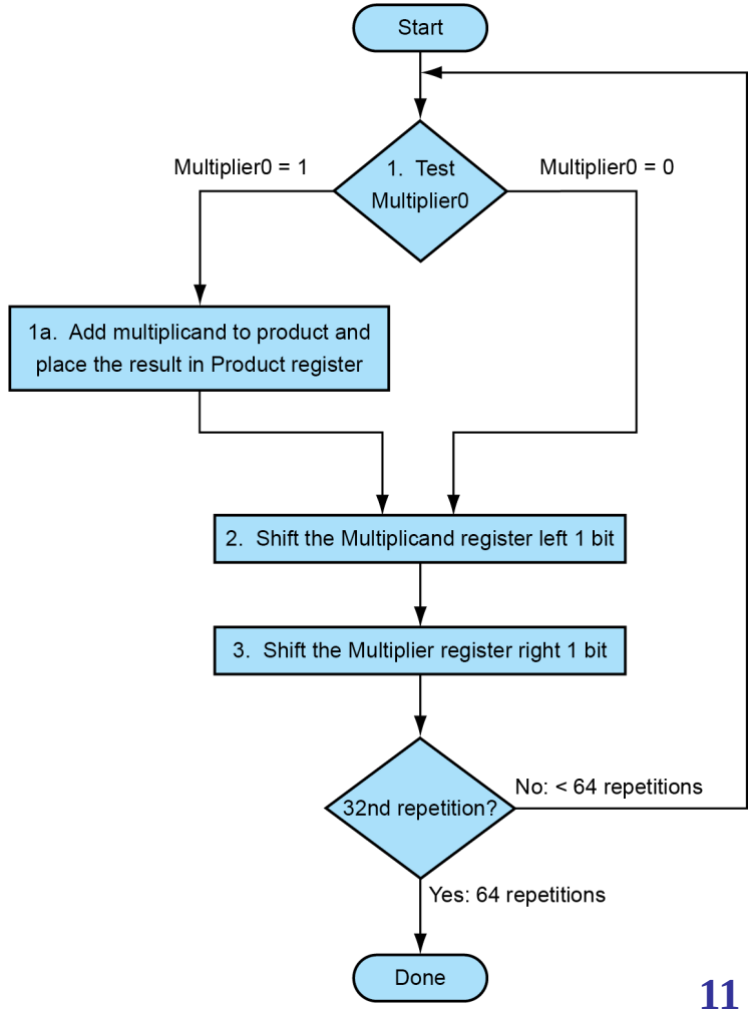
Length of product is the sum of operand lengths



3.3 乘法

- 乘法器硬件结构及算法实现，耗时约200个时钟周期
  - 1. 检测乘数最低位
    - 为1，将被乘数加到积上，并存入寄存器
    - 为0，则跳过
  - 2. 将被乘数寄存器左移一位
  - 3. 将乘数寄存器右移一位





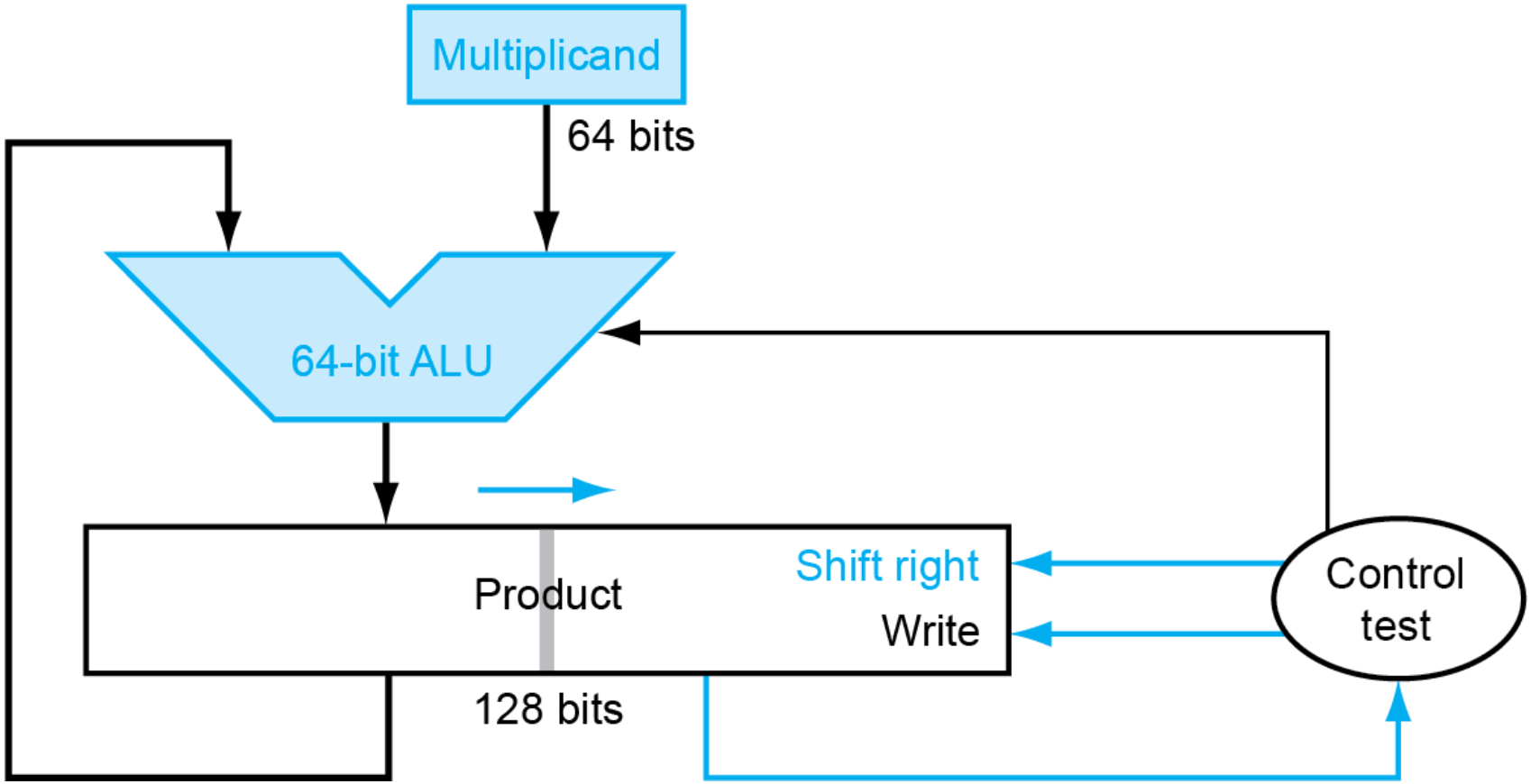
3.3 乘法

- 示例：2 * 3 (或者0010 * 0011)

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0011	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

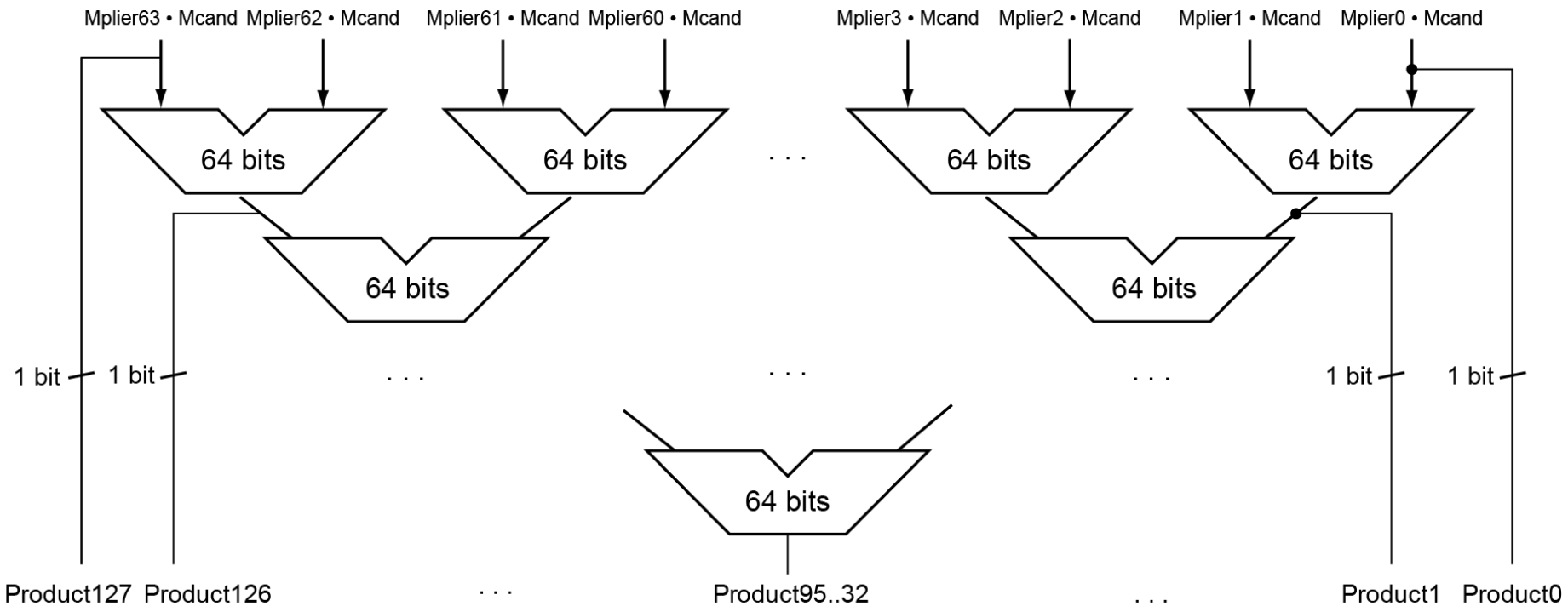
3.3 乘法

- 经过优化的乘法器，耗时约64个时钟周期



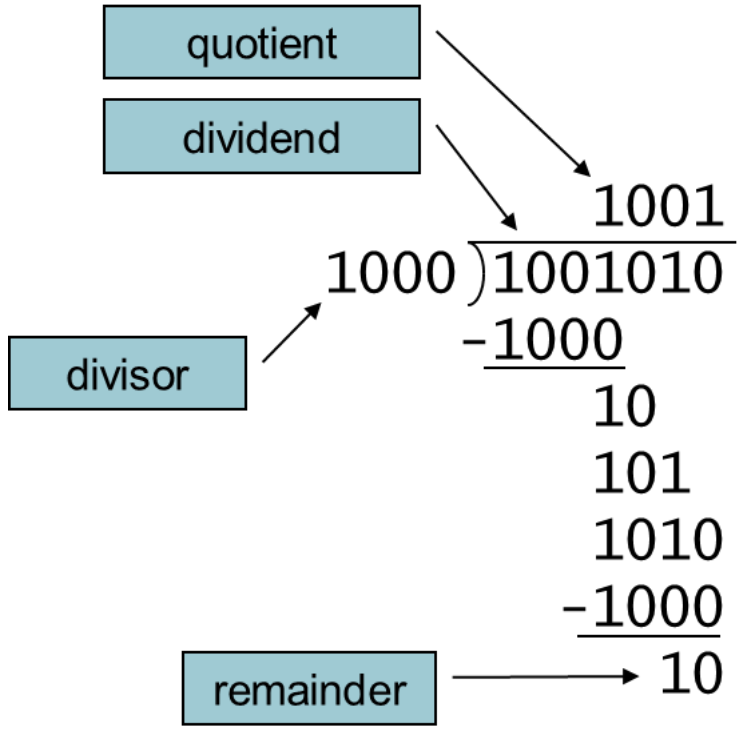
3.3 乘法

- 快速乘法，耗时约6个时钟周期
  - 使用多个加法器
  - 运算速度快于做6次加法
  - 可以通过流水线技术，同时支持多个乘法



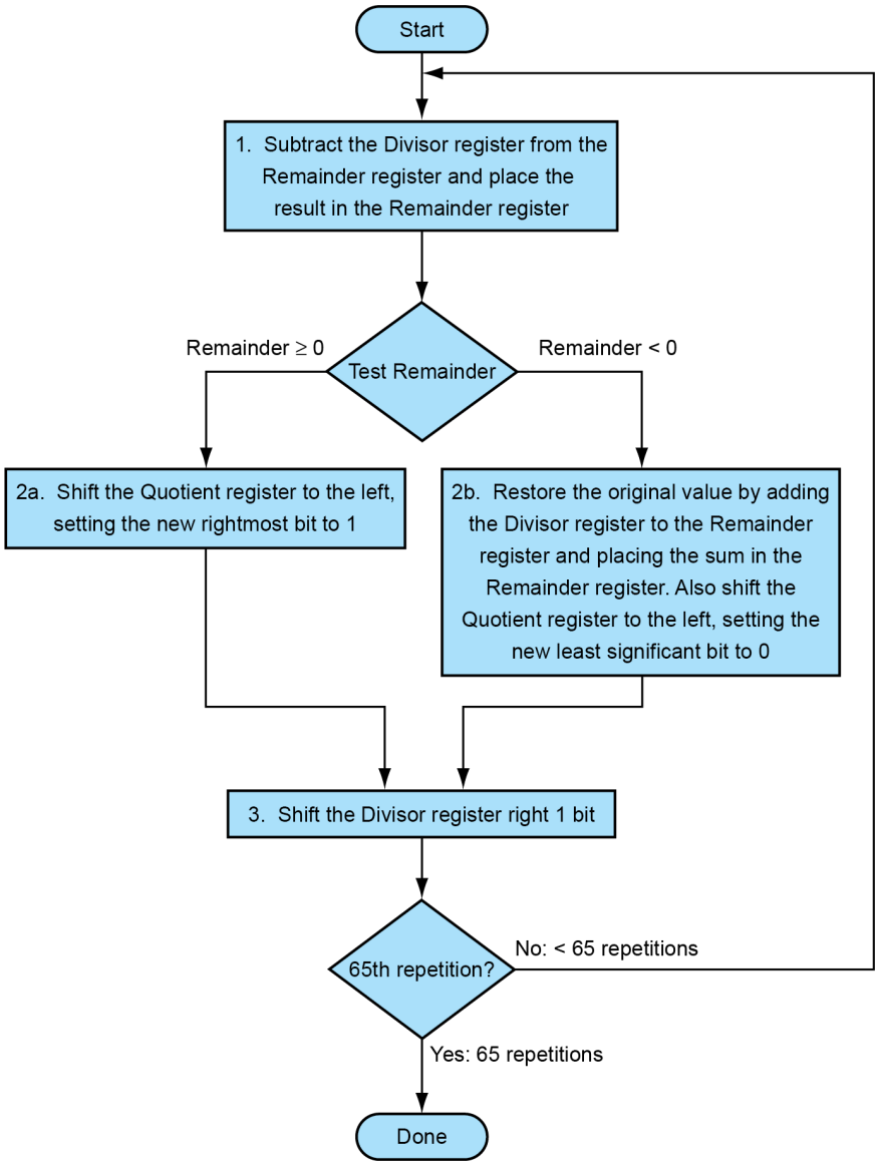
3.4 除法

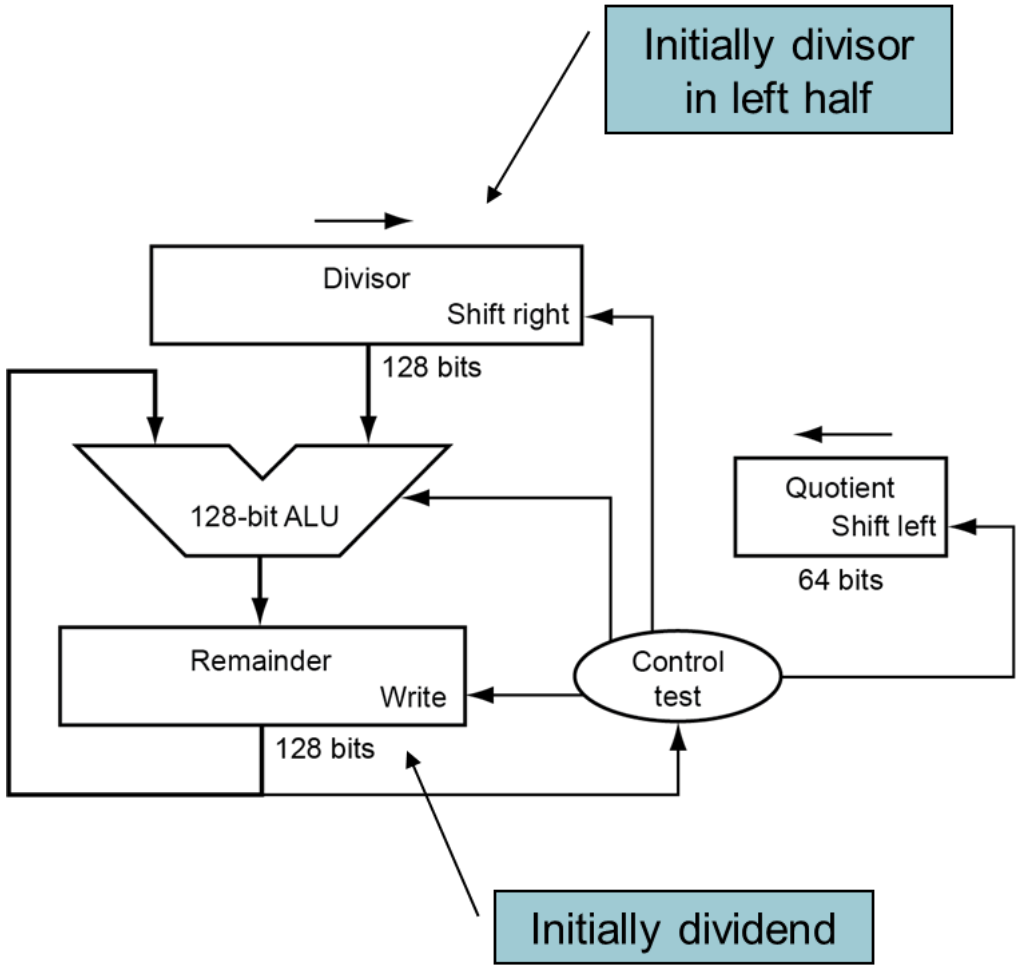
- 被除数：dividend
- 除数：divisor
- 商：quotient
- 余数：remainder
- 例：74 ÷ 8 = 9 余 2



- 说明：
  - 检查除数不为0
  - 在RV64中，两个源操作数和两个结果均为64位
  - 如除数和被除数都是正数，那商和余数也都是非负的（可以为0）
  - 对于有符号数的除法
    - 使用绝对值进行计算
    - 根据需要调整商和余数的符号

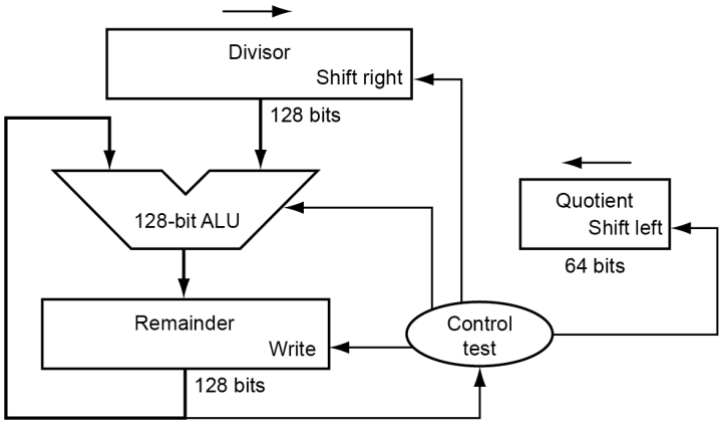
3.4 除法

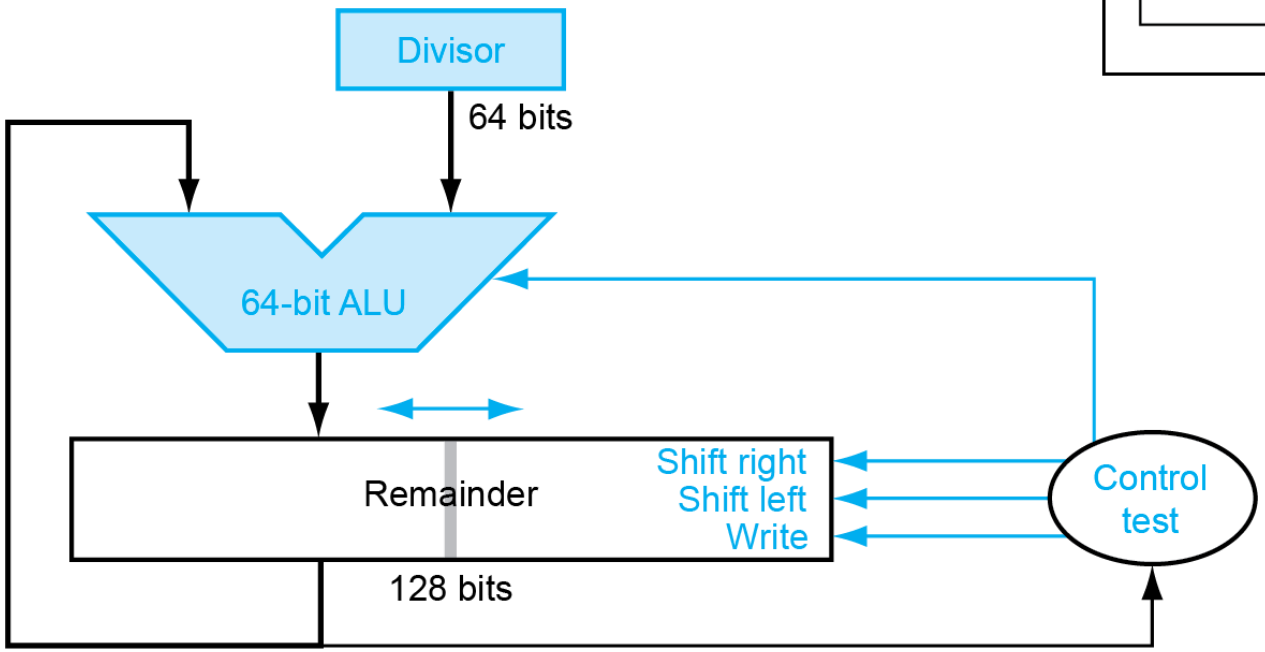

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
65th repetition?
No: < 65 repetitions
Yes: 65 repetitions
Done


Initially divisor in left half
Divisor
Shift right
128 bits
128-bit ALU
Quotient
Shift left
64 bits
Remainder
Write
Control test
128 bits
Initially dividend

3.4 除法

- 改进电路
  - 除数寄存器和ALU由128位降为64位
  - 取消商寄存器，与余数寄存器合用
  - 余数寄存器由128位改为129位





- 快速除法
  - 可自行调研，如：SRT除法技术

3.5 浮点运算

- RV32F：单精度浮点指令
- RV32D：双精度浮点指令
- RISC-V遵从IEEE 754-2008浮点标准

- RV32FD使用单独的32个F寄存器
- RV32F使用F寄存器的低32位

Floating-Point Computation

float { add | subtract | multiply | divide | square root | minimum | maximum } { .single | .double }

float { – negative } multiply { add | subtract } { .single | .double }

float move to .single from .x register

float move to .x register from .single

Comparison

compare float { equals | less than | less than or equals } { .single | .double }

Load and Store

float { load | store } { word | doubleword }

Conversion

float convert to { .single | .double } from .word { – unsigned }

float convert to .word { – unsigned } from { .single | .double }

float convert to .single from .double

float convert to .double from .single

Other instructions

float sign injection { – negative | exclusive or } { .single | .double }

float classify { .single | .double }

63 32	31 0	
	f0 / ft0	FP Temporary
	f1 / ft1	FP Temporary
	f2 / ft2	FP Temporary
	f3 / ft3	FP Temporary
	f4 / ft4	FP Temporary
	f5 / ft5	FP Temporary
	f6 / ft6	FP Temporary
	f7 / ft7	FP Temporary
	f8 / fs0	FP Saved register
	f9 / fs1	FP Saved register
	f10 / fa0	FP Function argument, return value
	f11 / fa1	FP Function argument, return value
	f12 / fa2	FP Function argument
	f13 / fa3	FP Function argument
	f14 / fa4	FP Function argument
	f15 / fa5	FP Function argument
	f16 / fa6	FP Function argument
	f17 / fa7	FP Function argument
	f18 / fs2	FP Saved register
	f19 / fs3	FP Saved register
	f20 / fs4	FP Saved register
	f21 / fs5	FP Saved register
	f22 / fs6	FP Saved register
	f23 / fs7	FP Saved register
	f24 / fs8	FP Saved register
	f25 / fs9	FP Saved register
	f26 / fs10	FP Saved register
	f27 / fs11	FP Saved register
	f28 / ft8	FP Temporary
	f29 / ft9	FP Temporary
	f30 / ft10	FP Temporary
	f31 / ft11	FP Temporary
32	32	

3.5 浮点运算

- RV32F包含26条指令（左图）
- RV32D包含26条指令（右图）

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	010	rd	000011	I flw
imm[11:5]		rs2	rs1	010	imm[4:0]	010011	S fsw
rs3	00	rs2	rs1	rm	rd	100001	R4
rs3	00	rs2	rs1	rm	rd	100011	R4
rs3	00	rs2	rs1	rm	rd	100101	R4
rs3	00	rs2	rs1	rm	rd	100111	R4
0000000		rs2	rs1	rm	rd	101001	R fadd.s
0000100		rs2	rs1	rm	rd	101001	R fsub.s
0001000		rs2	rs1	rm	rd	101001	R fmul.s
0001100		rs2	rs1	rm	rd	101001	R fdiv.s
0001100		00000	rs1	rm	rd	101001	R fsqrt.s
0010000		rs2	rs1	000	rd	101001	R fsgnj.s
0010000		rs2	rs1	001	rd	101001	R fsgnjn.s
0010000		rs2	rs1	010	rd	101001	R fsgnjx.s
0010100		rs2	rs1	000	rd	101001	R fmin.s
0010100		rs2	rs1	001	rd	101001	R fmax.s
1100000		00000	rs1	rm	rd	101001	R fcvt.w.s
1100000		00001	rs1	rm	rd	101001	R
1110000		00000	rs1	000	rd	101001	R fmv.x.w
1010000		rs2	rs1	010	rd	101001	R feq.s
1010000		rs2	rs1	001	rd	101001	R flt.s
1010000		rs2	rs1	000	rd	101001	R fle.s
1110000		00000	rs1	001	rd	101001	R fclass.s
1101000		00000	rs1	rm	rd	101001	R fcvt.s.w
1101000		00001	rs1	rm	rd	101001	R
1111000		00000	rs1	000	rd	101001	R fmv.w.x

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	011	rd	000011	I fld
imm[11:5]		rs2	rs1	011	imm[4:0]	010011	S fsd
rs3	01	rs2	rs1	rm	rd	100001	R4
rs3	01	rs2	rs1	rm	rd	100011	R4
rs3	01	rs2	rs1	rm	rd	100101	R4
rs3	01	rs2	rs1	rm	rd	100111	R4
0000001		rs2	rs1	rm	rd	101001	R fadd.d
0000101		rs2	rs1	rm	rd	101001	R fsub.d
0001001		rs2	rs1	rm	rd	101001	R fmul.d
0001101		rs2	rs1	rm	rd	101001	R fdiv.d
0001101		00000	rs1	rm	rd	101001	R fsqrt.d
0010001		rs2	rs1	000	rd	101001	R fsgnj.d
0010001		rs2	rs1	001	rd	101001	R fsgnjn.d
0010001		rs2	rs1	010	rd	101001	R fsgnjx.d
0010101		rs2	rs1	000	rd	101001	R fmin.d
0010101		rs2	rs1	001	rd	101001	R fmax.d
0100000		00001	rs1	rm	rd	101001	R fcvt.s.d
0100001		00000	rs1	rm	rd	101001	R fcvt.d.s
1010001		Rs2	rs1	010	rd	101001	R feq.d
1010001		rs2	rs1	001	rd	101001	R flt.d
1010001		rs2	rs1	000	rd	101001	R fle.d
1110001		00000	rs1	001	rd	101001	R fclass.d
1100001		00000	rs1	rm	rd	101001	R fcvt.w.d
1100001		00001	rs1	rm	rd	101001	R
1101001		00000	rs1	rm	rd	101001	R fmv.d.w
1101001		00001	rs1	rm	rd	101001	R

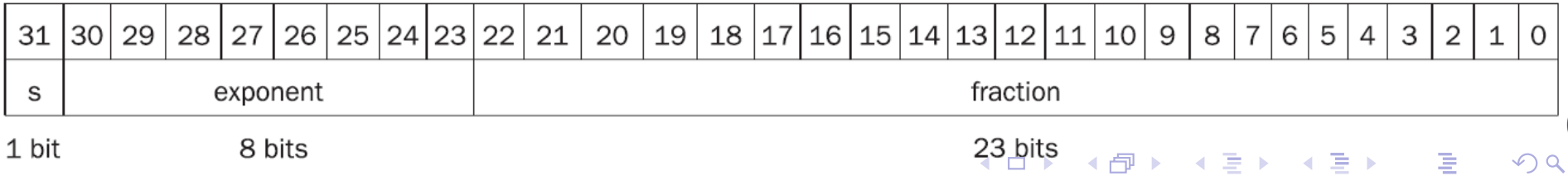
3.5 浮点运算

- 以科学计数法的形式表示二进制实数(IEEE 754标准)
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
  - 对应于C语言中的float（单精度）和double（双精度）类型

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

single: 8 bits　　single: 23 bits
double: 11 bits　　double: 52 bits

  - S：符号位，0为正，1为负
  - F：尾数，单精度23位，双精度52位，无符号
  - E：指数，单精度8位，双精度11位，无符号
  - Bias：偏移值，单精度为127，双精度为1023



3.5 浮点运算

- 浮点表示实例：-0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - 符号位：S = 1
  - 尾数：Fraction = $1000\ldots00_2$
  - 指数：Exponent = –1 + Bias
    - 单精度：–1 + 127 = 126 = 01111110_2
    - 双精度：–1 + 1023 = 1022 = 01111111110_2

- -0.75单精度表示：10111111010000...00
- -0.75双精度表示：10111111111010000...00

- 浮点加法：略
- 浮点乘法：略

3.6 并行性与计算机算术：子字并行

- 略

3.7 实例：x86中的SIMD扩展和高级向扩展

- 略

3. 8. 加速：子字并行和矩阵乘法

- 略

3. 9. 谬误与陷阱

- 略

3. 10. 本章小结

- 略

中国科学技术大学
University of Science and Technology of China

计算机组成原理
CH4_处理器

卢建良
lujl@ustc.edu.cn
2022年春季学期

提纲

- 4.1 引言
- 4.2 逻辑设计的一般方法
- 4.3 建立数据通路
- 4.4 一个简单的实现方案
- 4.4a Verilog实现
- 4.4b 多周期设计
- 4.5 流水线概述
- 4.6 流水线数据通路和控制
- 4.7 数据冒险：前递与停顿
- 4.8 控制冒险
- 4.9 例外

提纲

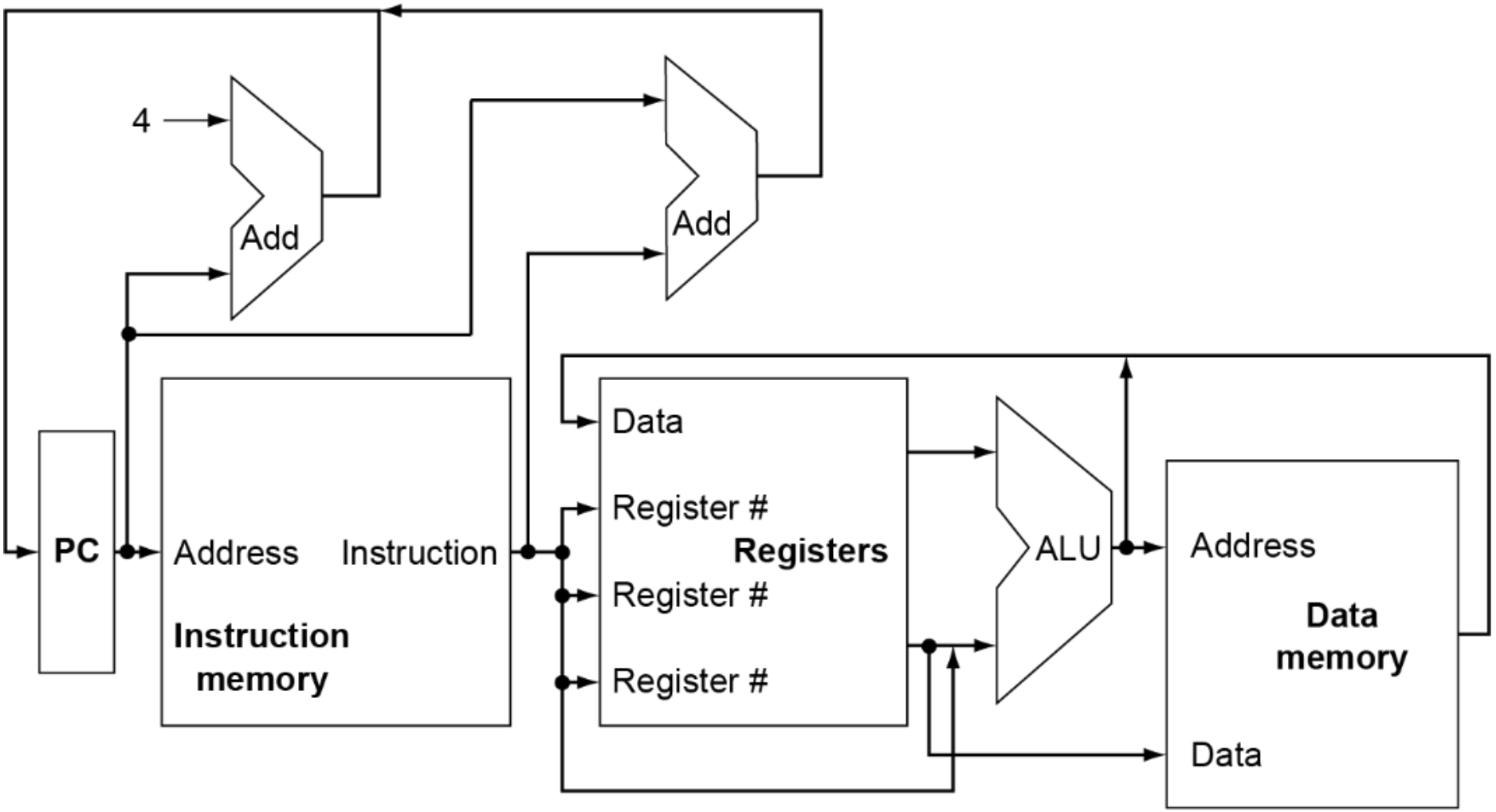
- 4.10 指令间的并行性
- 4.11 实例：ARM cortex-A53和Intel Core i7流水线结构
- 4.12 加速：指令级并行和矩阵乘法
- 4.13 高级专题
- 4.14 谬误与陷阱
- 4.15 本章小结

4.1 引言

- 本教材实现实现RISC-V的一个核心子集
  - 存储器访问指令：ld和sd
  - 算术逻辑指令：add、sub、and、or
  - 条件分支指令：beq
- 指令执行过程
  - 1. 以PC（程序计数器）为地址，从指令存储器中获取指令
  - 2. 解析指令中的寄存器编号，从寄存器文件中读取数据
  - 3. 根据指令的不同，执行不同的操作
    - 通过ALU进行运算
      - 算术逻辑指令：运算结果
      - 访存指令：Memory地址
      - 分支指令：比较结果
    - 访存指令读写存储器/算术运算指令写寄存器文件/分支指令确定目标地址
  - 4. 更新PC值：PC+4或者目标跳转地址

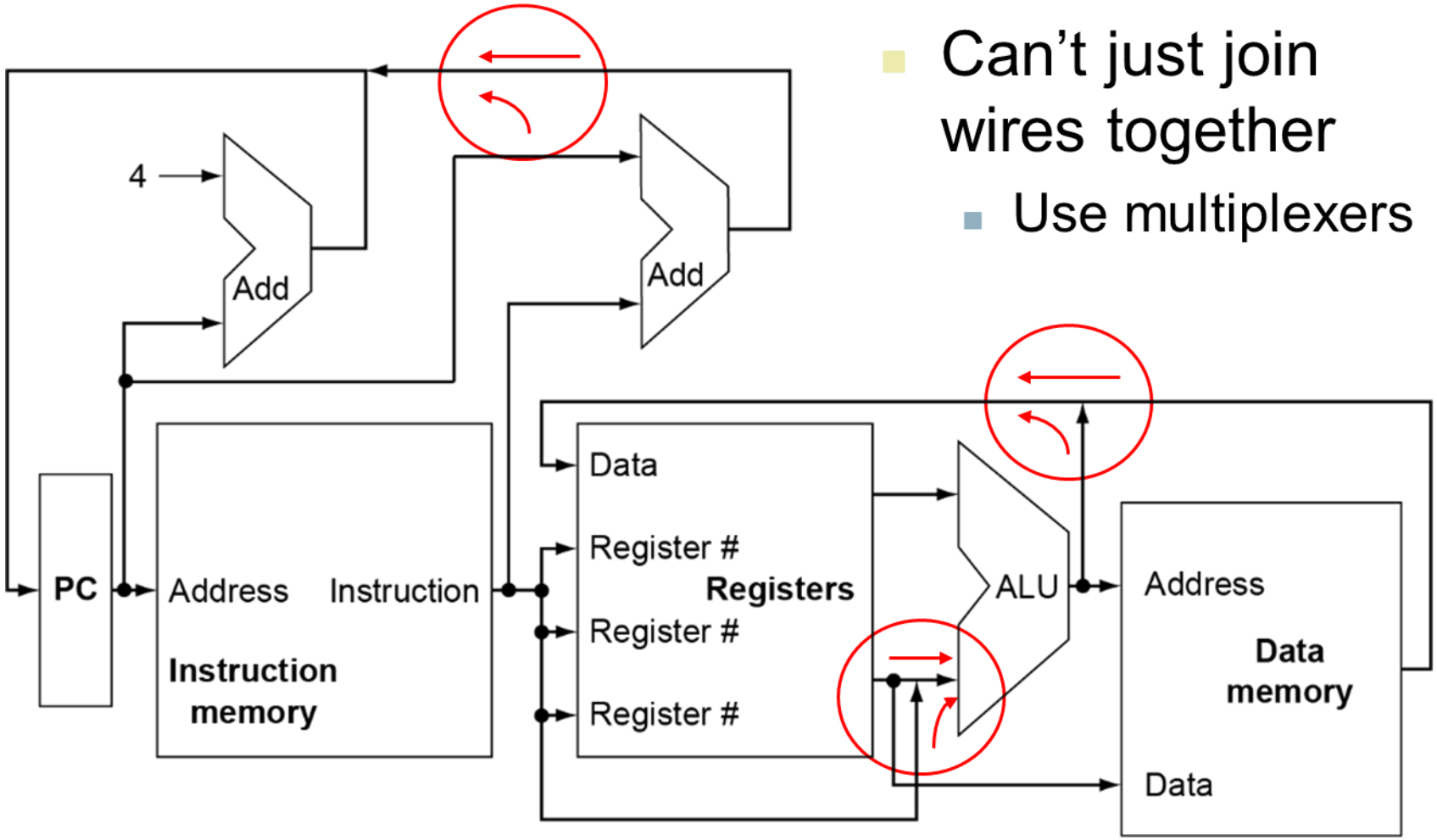
4.1 引言

- 数据通路示意图



4.1 引言

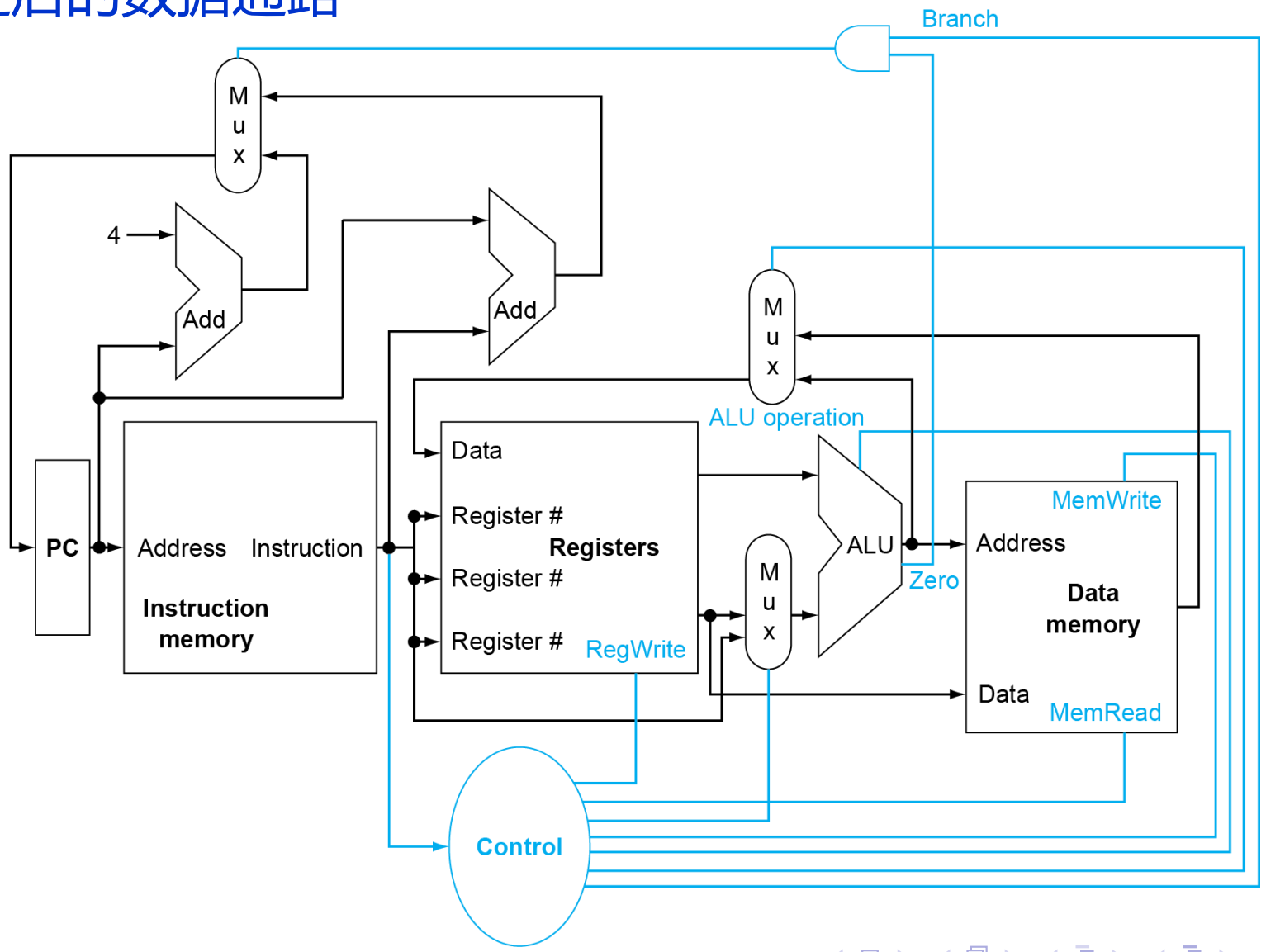
- 添加Mux以实现控制



- Can't just join wires together
  - Use multiplexers

4.1 引言

- 改进后的数据通路

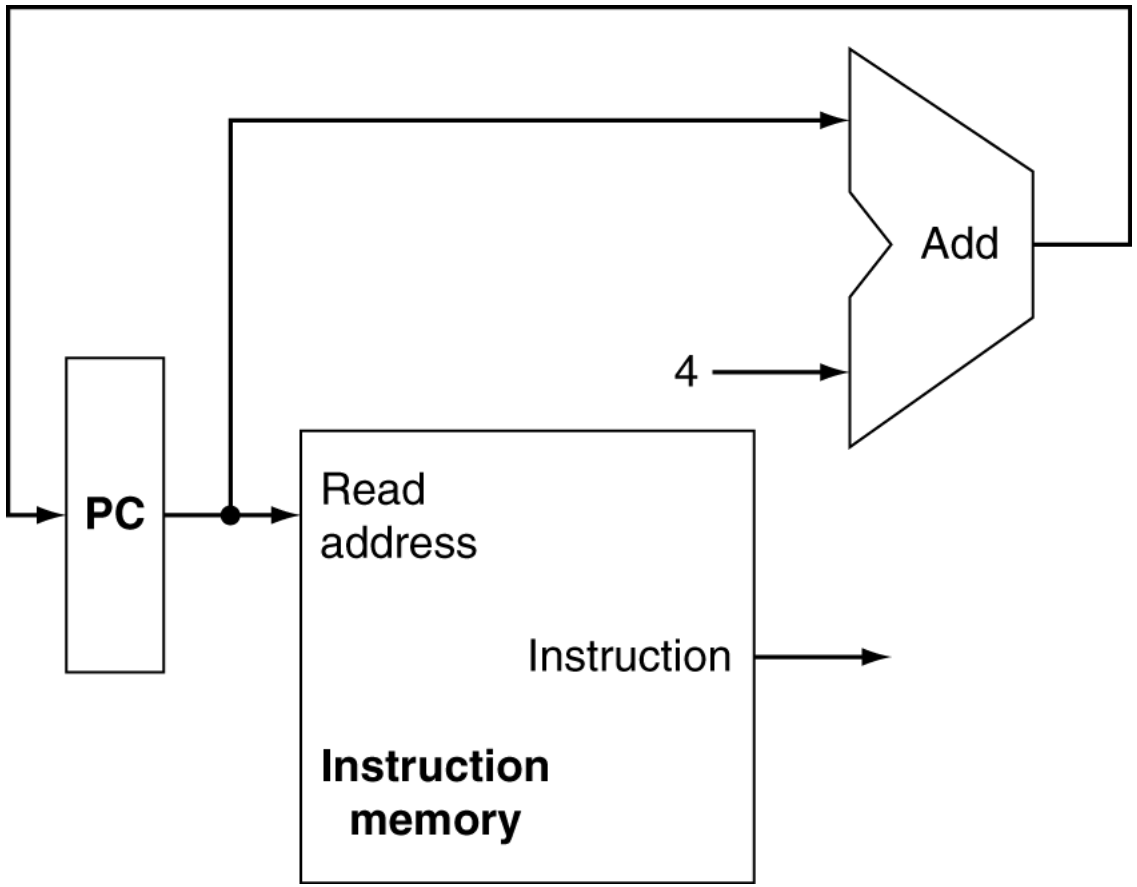


4.2 逻辑设计的一般方法

- 略

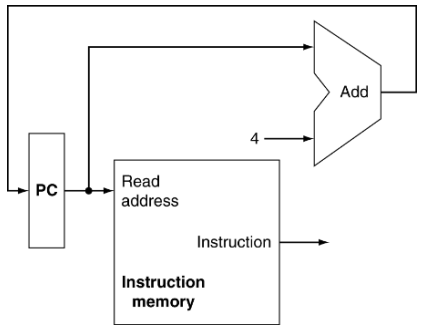
4.3 建立数据通路

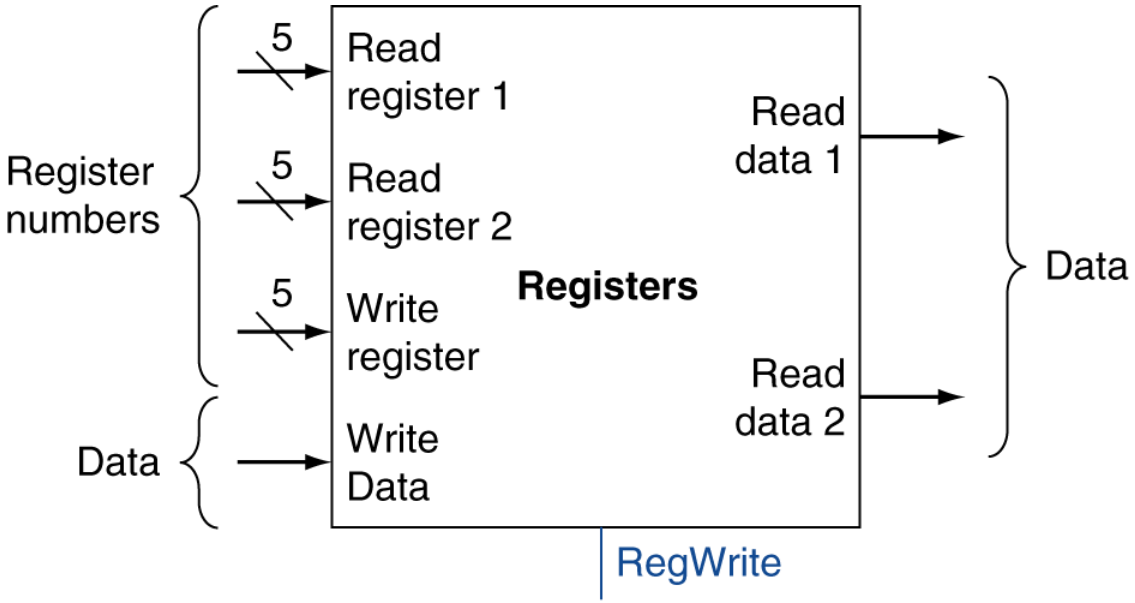
- 取指相关电路



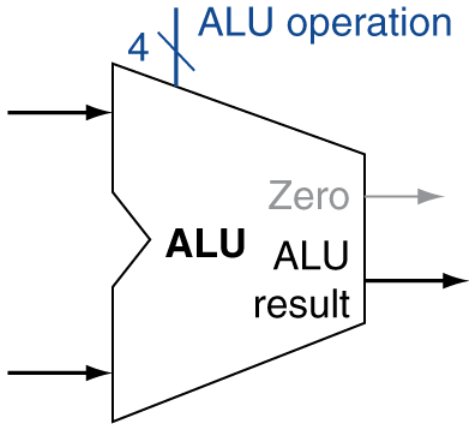
4.3 建立数据通路

- R-type指令
  - 从寄存器文件中读取两个寄存器
  - 通过ALU进行算术、逻辑运算
  - 将结果写入寄存器文件
  - eg：add x5, x6, x7





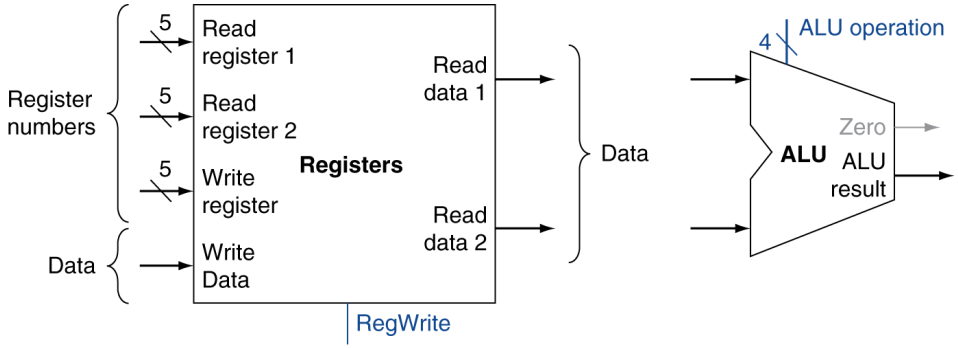
a. Registers



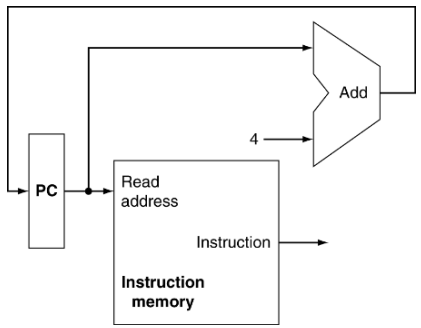
b. ALU

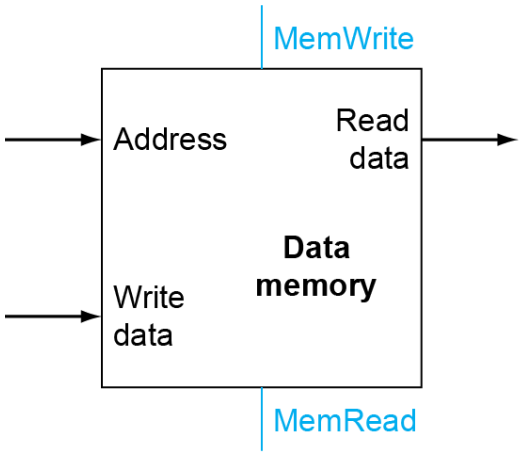
4.3 建立数据通路

- Load/Store指令
  - eg：lw　x5, 0(x6)
  - eg：sw　x5, 0(x6)
  - 从寄存器文件中读取操作数
  - 使用12bit偏移量计算目标地址
  - 执行访存操作
    - Load：从存储器中读取数据，写入寄存器文件
    - Store：将从寄存器文件读取的数据写入存储器

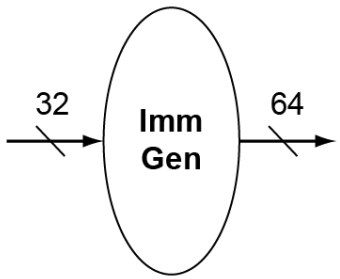


a. Registers　　b. ALU





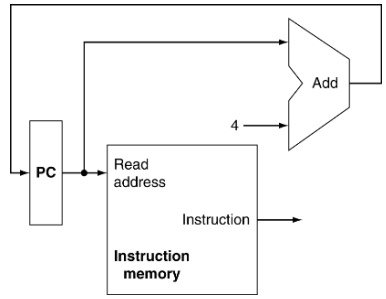
a. Data memory unit

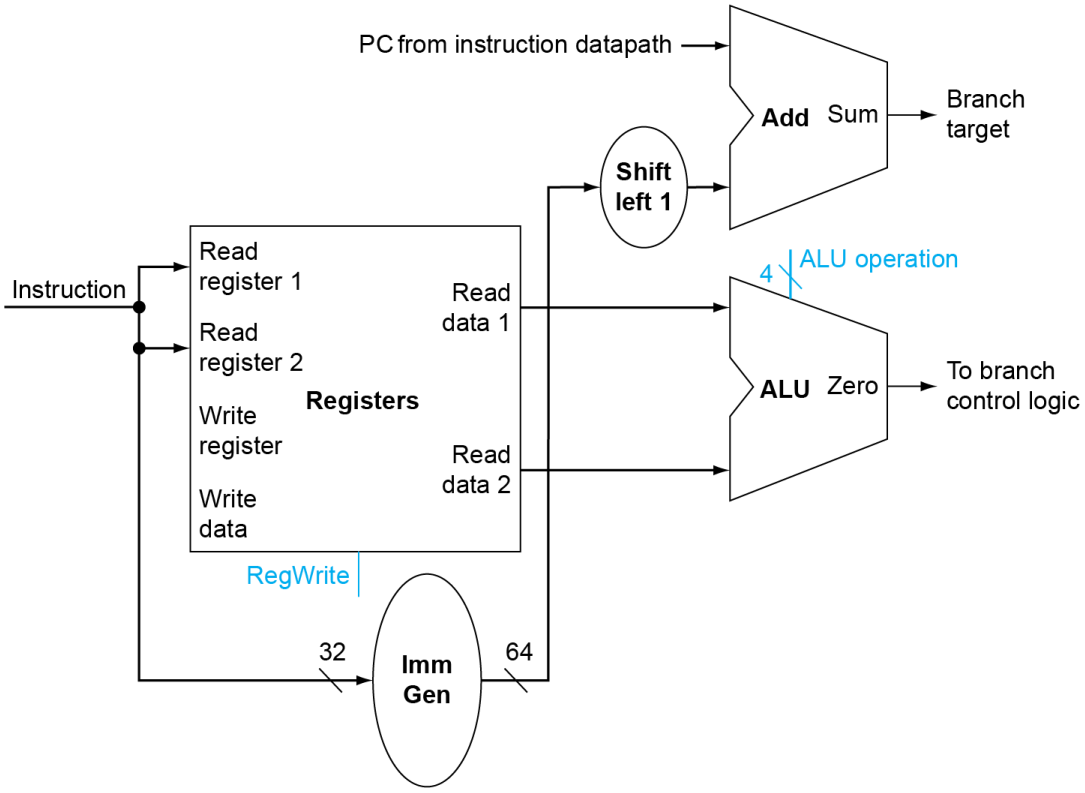


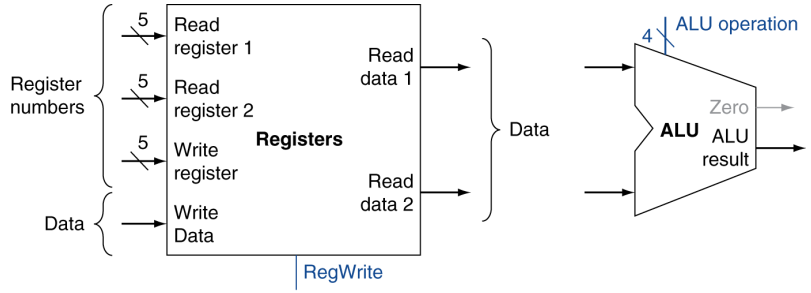
b. Immediate generation unit

4.3 建立数据通路

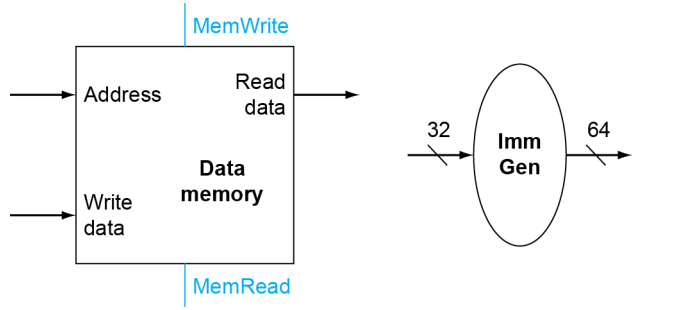
- 分支指令
  - eg： beq x5, x6, Lable
  - 从寄存器文件中读取两个寄存器
  - 使用ALU进行比较操作，使用ALU、减法并检查Zero输出
  - 计算目标地址（由什么部件来计算？）







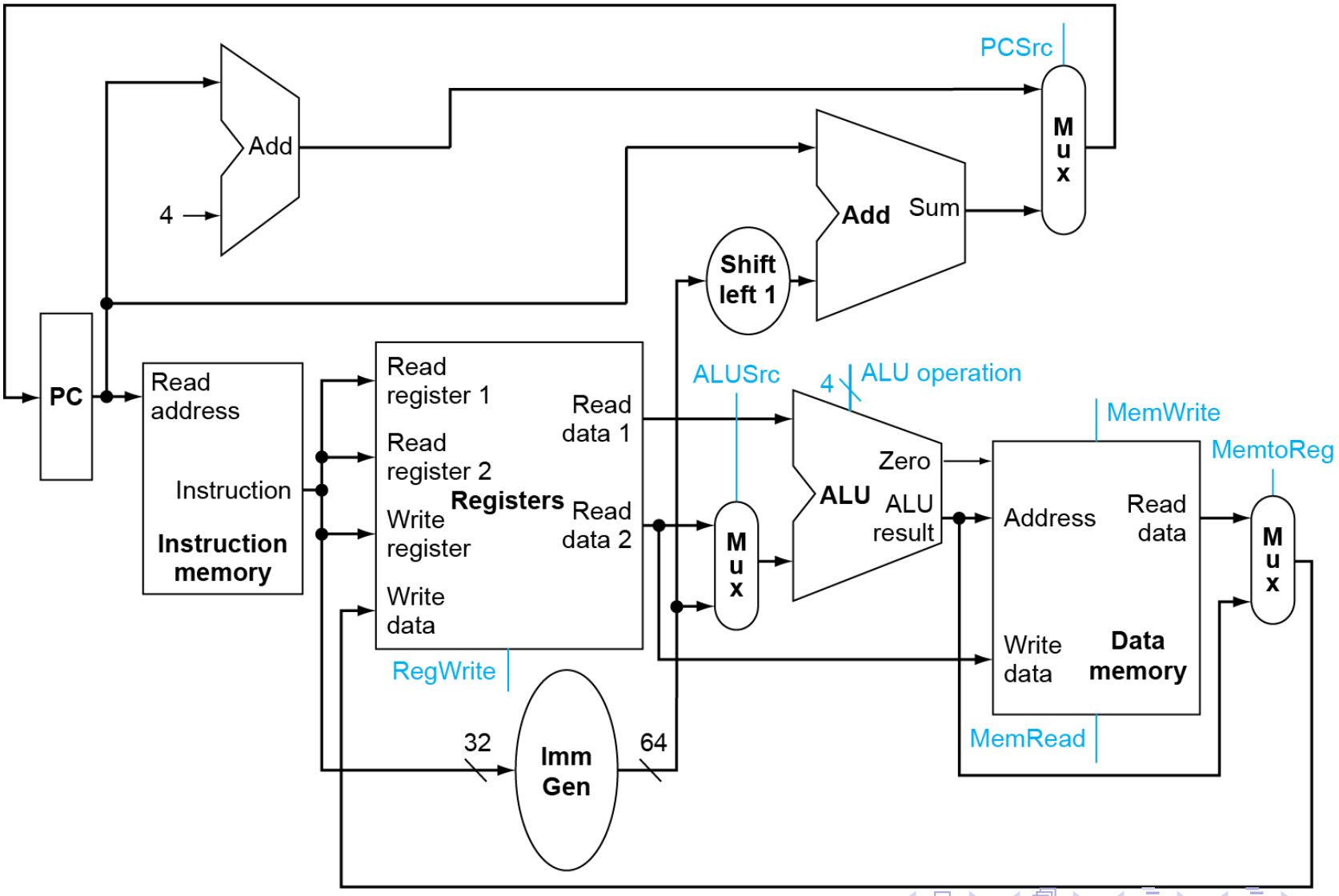
a. Registers　　b. ALU



a. Data memory unit　　b. Immediate generation unit

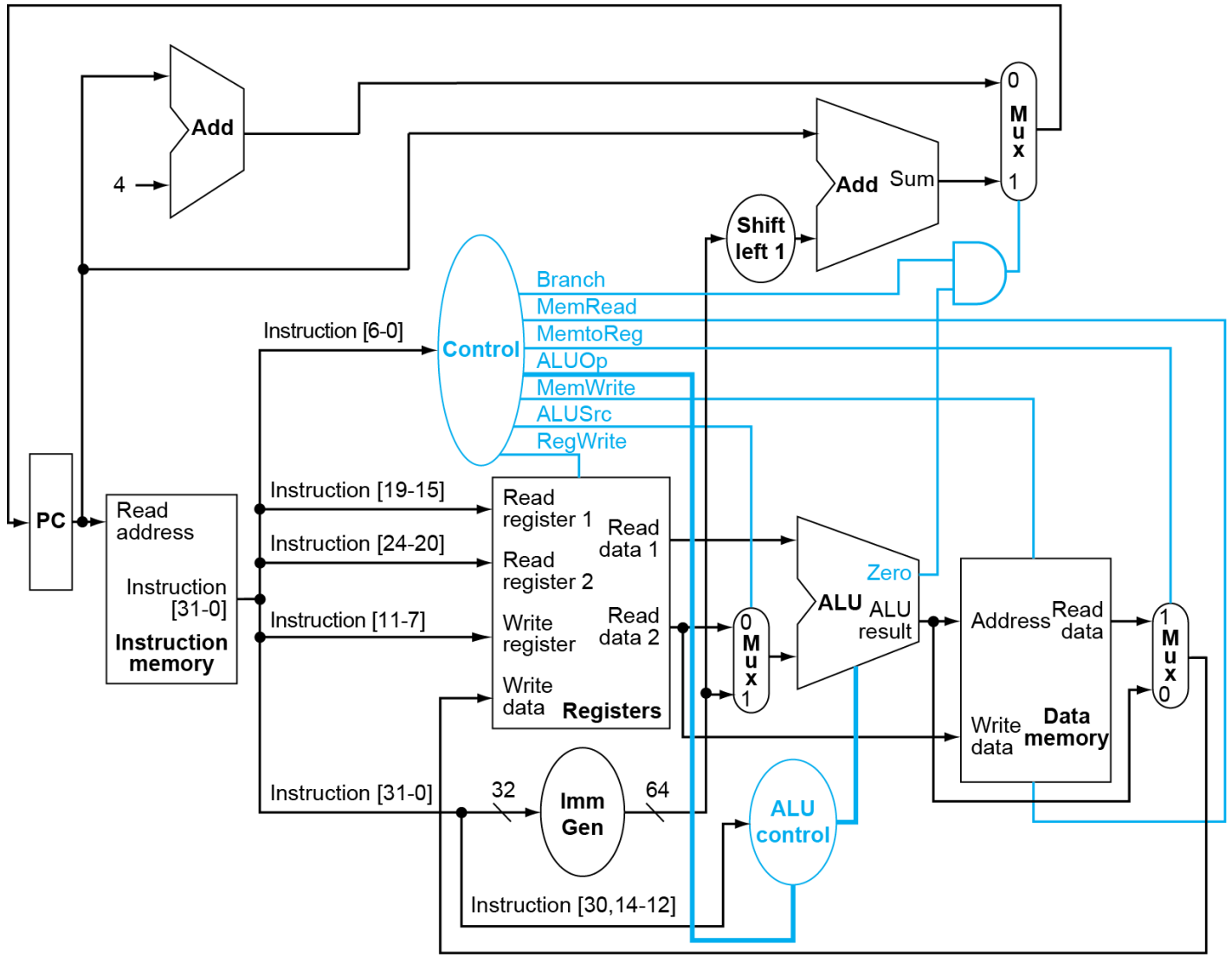
4.3 建立数据通路

- 完整的数据通路



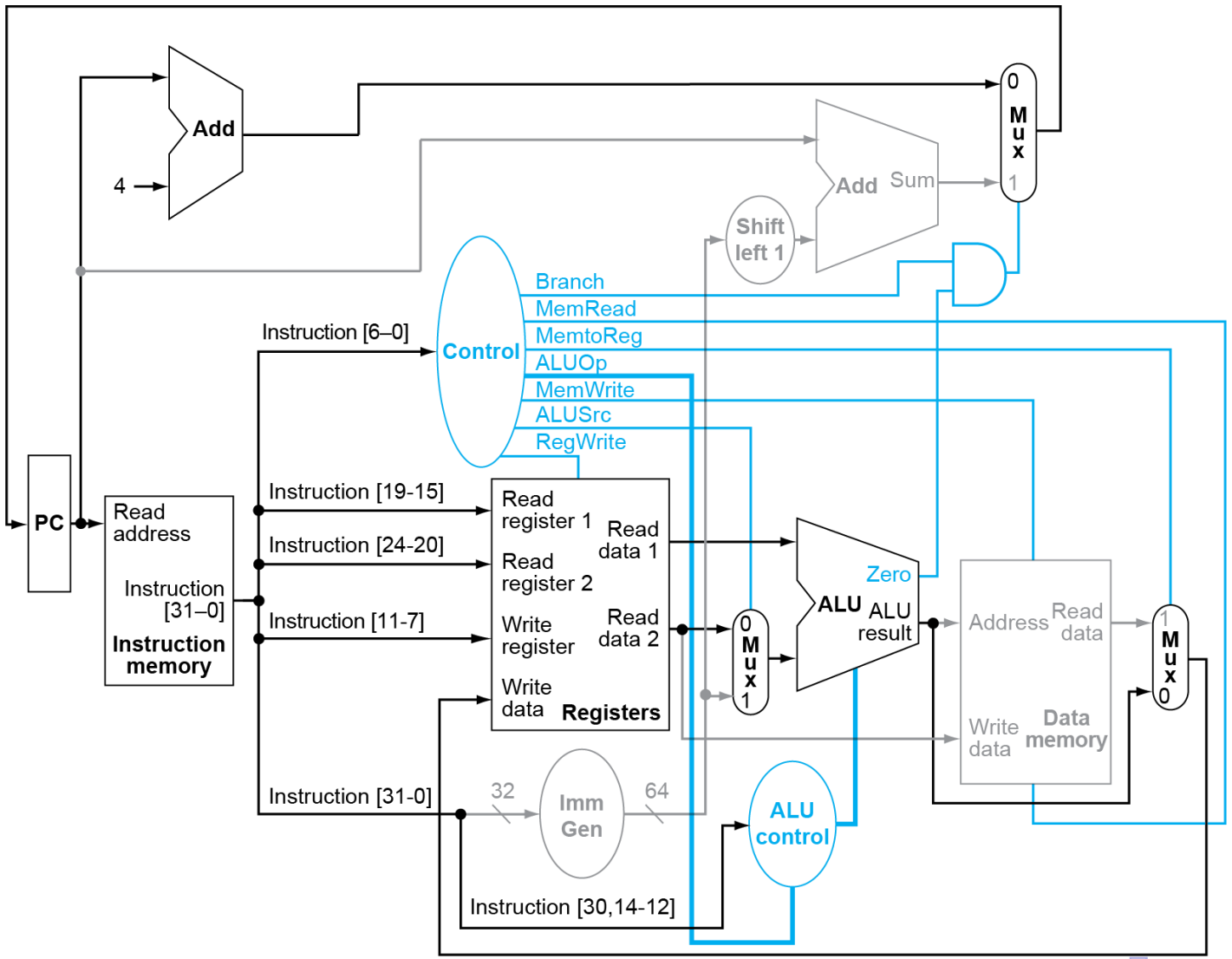
4.4 一个简单的实现方案

- 带控制器的数据通路



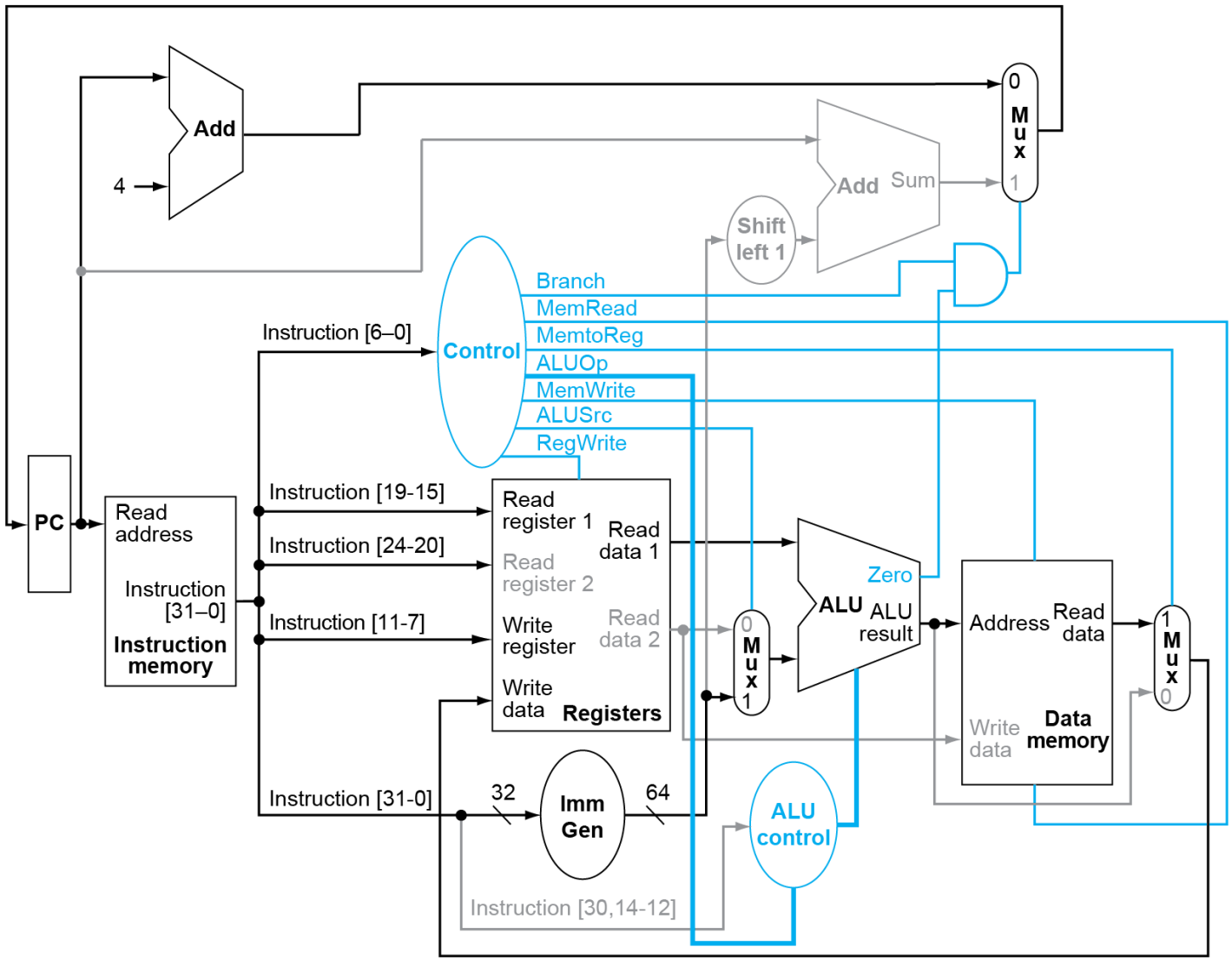
4.4 一个简单的实现方案

- R-type数据通路



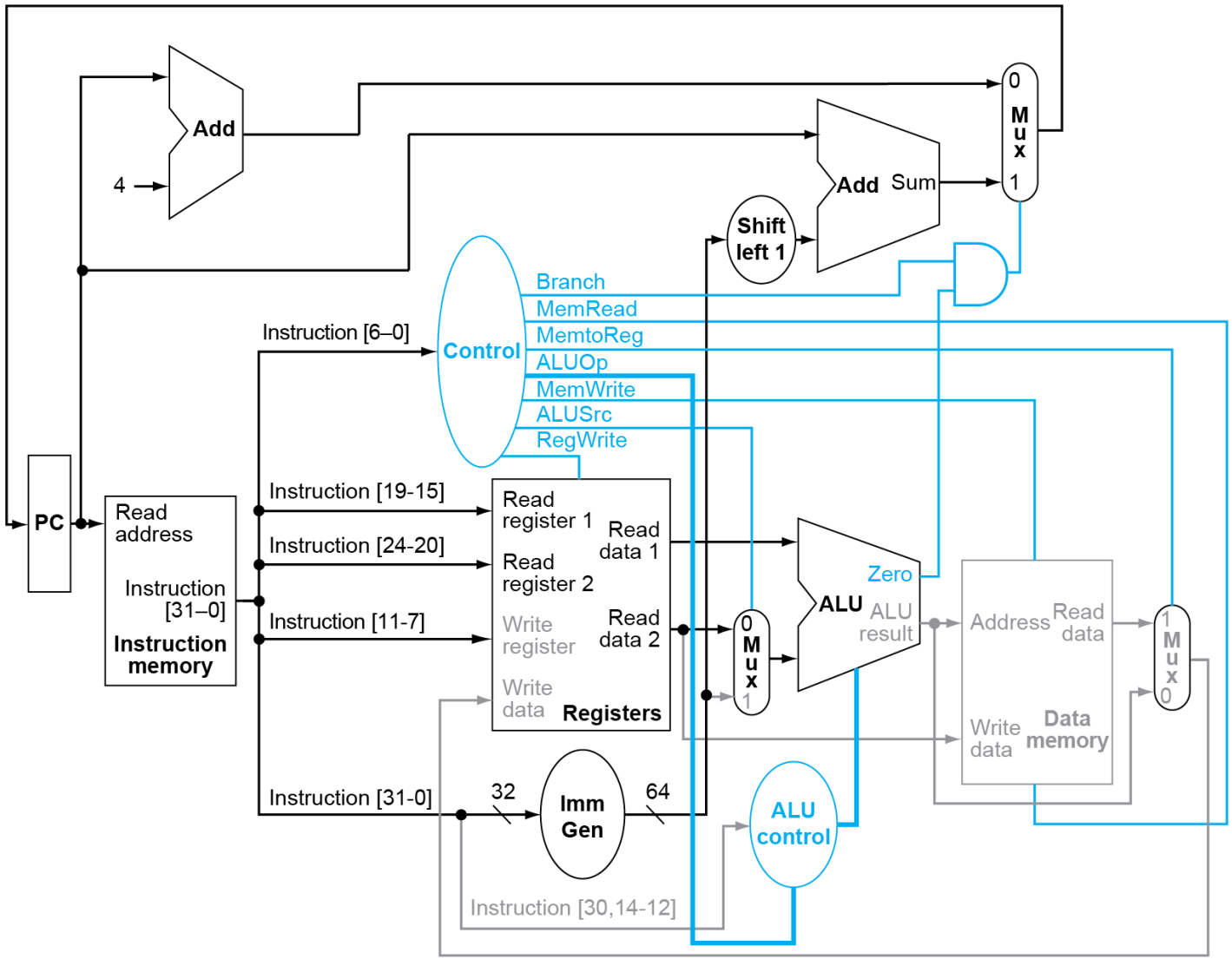
4.4 一个简单的实现方案

- Load指令



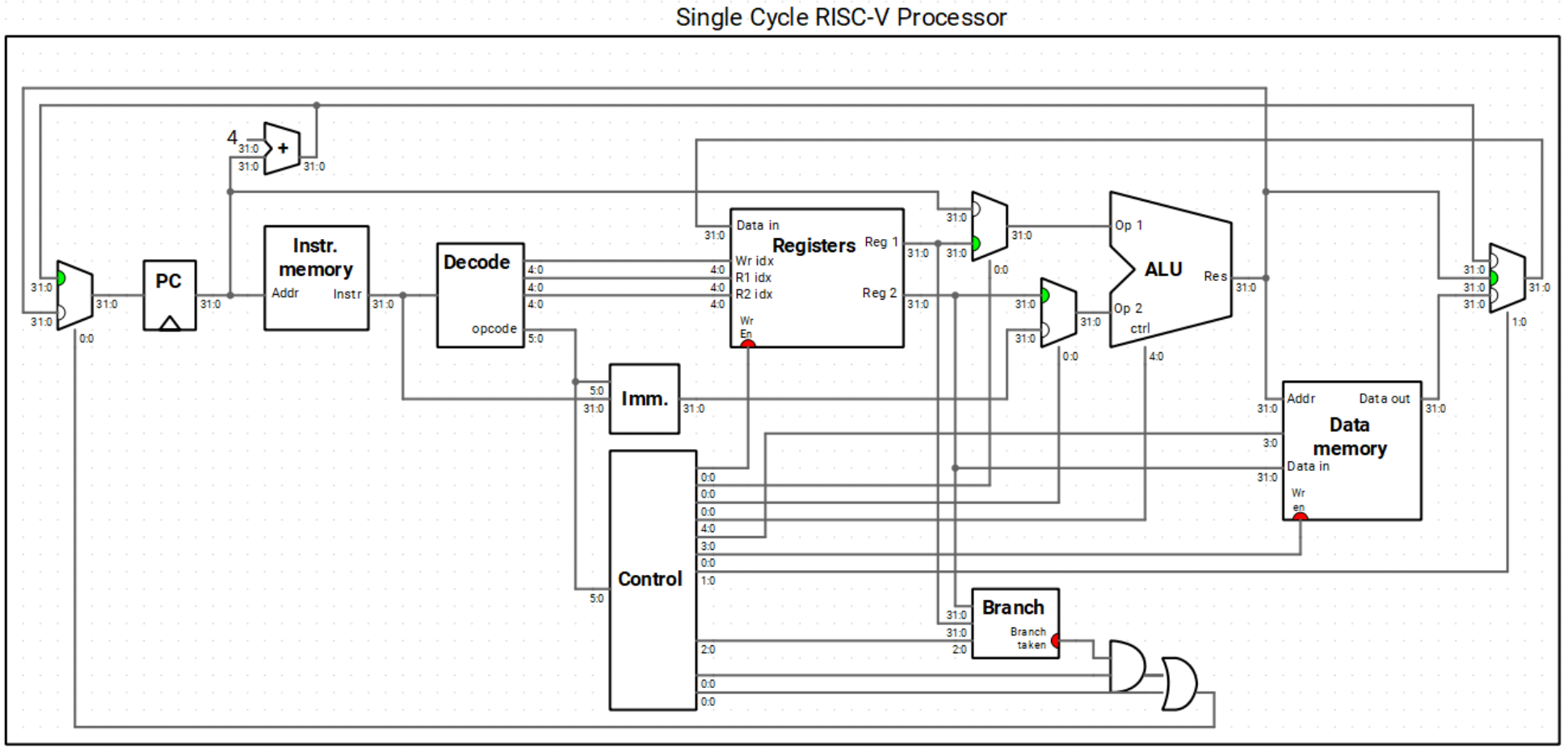
4.4 一个简单的实现方案

- beq指令



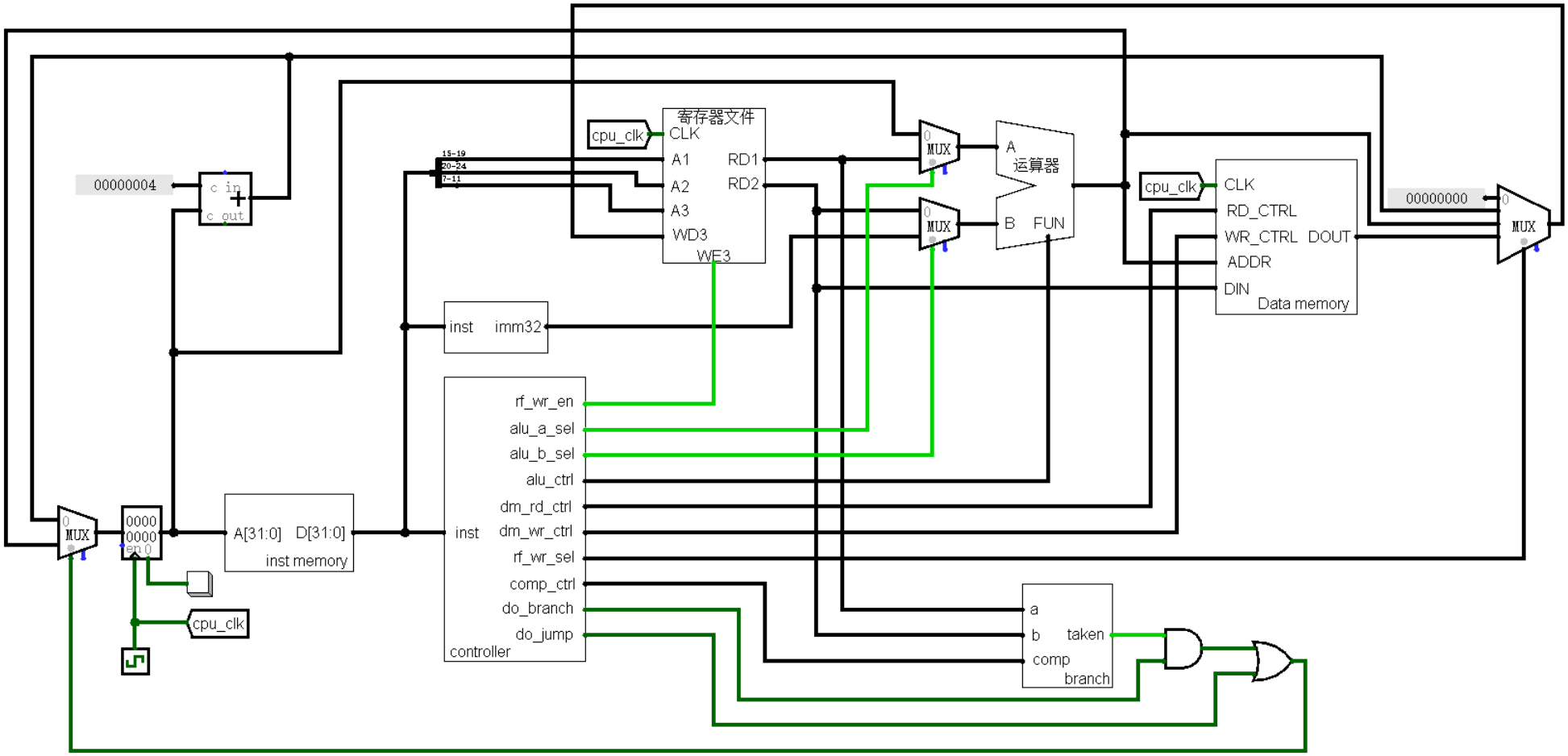
4.4 一个简单的实现方案

- Ripes模拟器中的单周期数据通路
  - 最大区别：条件分支指令中两个操作数的比较不由ALU实现



4.4 一个简单的实现方案

- 基于Logisim的RV32I 37条指令的单周期实现
  - 特点：电路可编辑，可查看各模块的内部结构



4.4a Verilog实现

- 顶层模块

```
1
2   module core(
3   input clk,
4   input rst
5   );
6   wire        clk;
7   wire        rst;
8   wire        branch;
9   wire [31:0] pc;
10  wire [31:0] pc_plus4;
11  wire    [31:0]  inst;
12  wire    [31:0]  imm_out;
13  wire            rf_wr_en;
14  wire            alu_a_sel;
15  wire            alu_b_sel;
16  wire    [3:0]   alu_ctrl;
17  wire    [2:0]   dm_rd_ctrl;
18  wire    [1:0]   dm_wr_ctrl;
19  wire    [1:0]   rf_wr_sel;
20  wire    [2:0]   comp_ctrl;
21  wire            do_branch;
22  wire            do_jump;
23
24  reg     [31:0]  rf_wd3;
25  wire    [31:0]  rf_rd1,rf_rd2;
26  wire    [31:0]  alu_a,alu_b,alu_out;
27  wire    [31:0]  dm_dout;
```

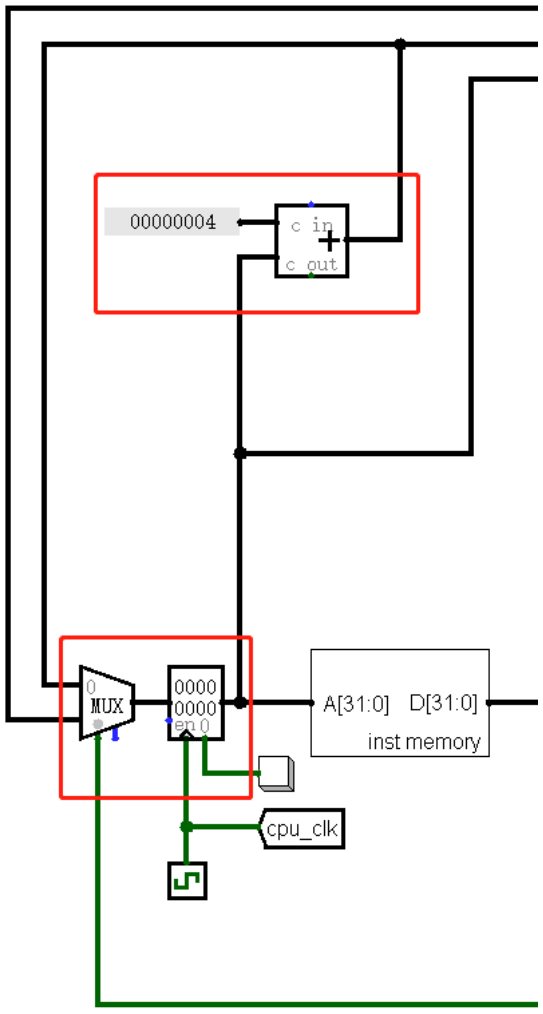
```
29  pc_calc     pc_calc(
30  .clk        (clk),
31  .rst        (rst),
32  .branch     (branch),
33  .alu_out    (alu_out),
34  .pc         (pc),
35  .pc_plus4   (pc_plus4)
36  );
37
38  mem_cache   mem_cache(
39  .clk        (clk),
40  .im_addr    (pc),
41  .im_dout    (inst),
42  .dm_rd_ctrl (dm_rd_ctrl),
43  .dm_wr_ctrl (dm_wr_ctrl),
44  .dm_addr    (alu_out),
45  .dm_din     (rf_rd2),
46  .dm_dout    (dm_dout)
47  );
48
49  imm         imm(
50  .inst       (inst),
51  .imm_out    (imm_out)
52  );
53
54  controller  controller(
55  .inst       (inst ),
56  .rf_wr_en   (rf_wr_en ),
57  .alu_a_sel  (alu_a_sel    ),
58  .alu_b_sel  (alu_b_sel    ),
59  .alu_ctrl   (alu_ctrl     ),
60  .dm_rd_ctrl (dm_rd_ctrl   ),
61  .dm_wr_ctrl (dm_wr_ctrl   ),
62  .rf_wr_sel  (rf_wr_sel    ),
63  .comp_ctrl  (comp_ctrl    ),
64  .do_branch  (do_branch    ),
65  .do_jump    (do_jump      )
66  );
```

```
68  always@(*)
69  begin
70      case(rf_wr_sel)
71      2'b00:  rf_wd3 = 32'h0;
72      2'b01:  rf_wd3 = pc_plus4;
73      2'b10:  rf_wd3 = alu_out;
74      2'b11:  rf_wd3 = dm_dout;
75      default:rf_wd3 = 32'h0;
76      endcase
77  end
78
79  reg_file    reg_file(
80  .clk        (clk),
81  .a1         (inst[19:15]),
82  .a2         (inst[24:20]),
83  .a3         (inst[11:7]),
84  .wd3        (rf_wd3),
85  .we3        (rf_wr_en),
86  .rd1        (rf_rd1),
87  .rd2        (rf_rd2)
88  );
89
90  assign      alu_a = alu_a_sel ? rf_rd1 : pc ;
91  assign      alu_b = alu_b_sel ? imm_out : rf_rd2 ;
92
93  alu         alu(
94  .a          (alu_a),
95  .b          (alu_b),
96  .alu_ctrl   (alu_ctrl),
97  .alu_out    (alu_out)
98  );
99
100 brah          brah(
101 .a          (rf_rd1),
102 .b          (rf_rd2),
103 .comp_ctrl  (comp_ctrl),
104 .do_branch  (do_branch),
105 .do_jump    (do_jump),
106 .branch     (branch)
107 );
108
109 endmodule
```

4.4a Verilog实现

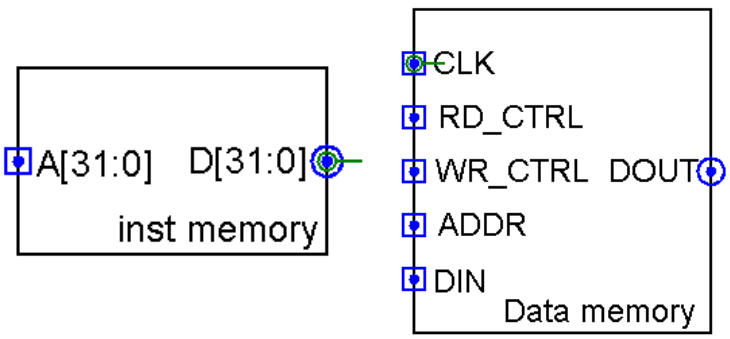
- pc模块
  - 电路图与Verilog代码并不完全对应
    - eg：PC复位值不相同

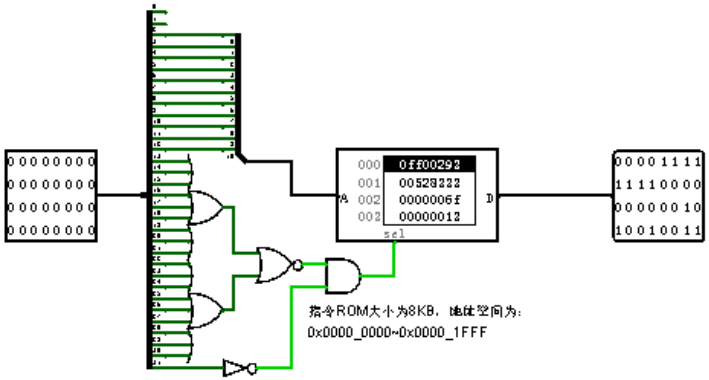
```
1  module pc_calc(
2  input                clk,
3  input                rst,
4  input                branch,
5  input        [31:0] alu_out,
6  output reg   [31:0] pc,
7  output       [31:0] pc_plus4);
8
9  always@(posedge clk or posedge rst)
10 begin
11     if(rst)
12         pc <= 32'h8000_0000;
13     else if(branch)
14         pc <= alu_out;
15     else
16         pc <= pc_plus4;
17 end
18
19 assign pc_plus4 = pc + 32'h4;
20
21 endmodule
```

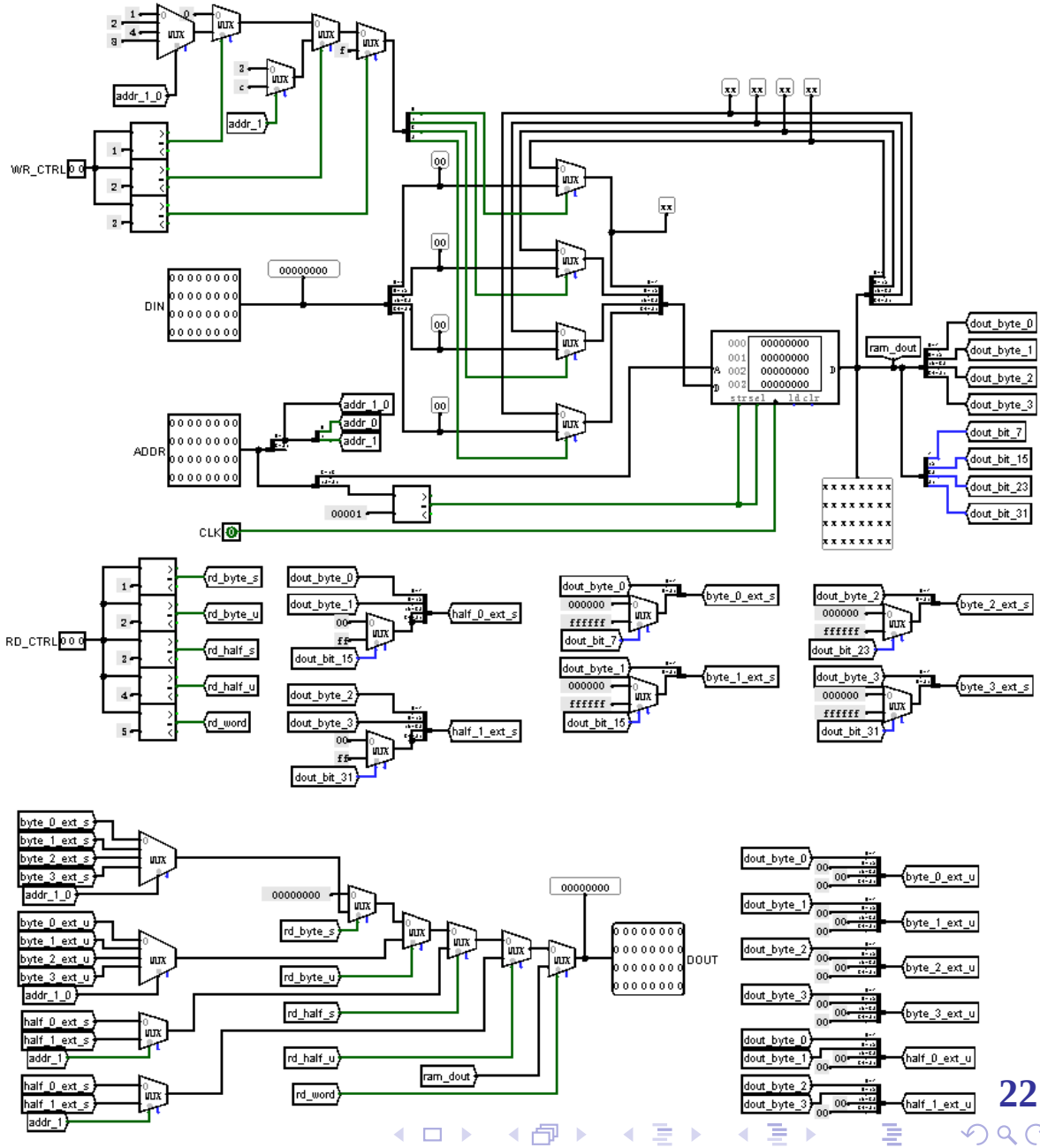


4.4a Verilog实现

- 存储器模块
  - 数据存储器
  - 指令存储器



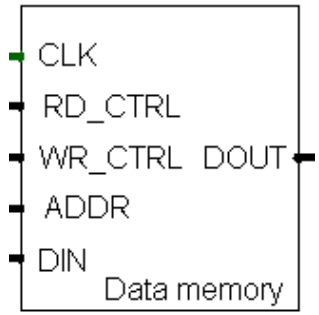




4.4a Verilog

- 存储器模块
  - 数据存储器
  - 指令存储器

```
module mem_cache(
input           clk,
input   [31:0]  im_addr,
output  [31:0]  im_dout,
input   [2:0]   dm_rd_ctrl,
input   [1:0]   dm_wr_ctrl,
input   [31:0]  dm_addr,
input   [31:0]  dm_din,
output reg  [31:0]  dm_dout
);
reg     [3:0]   byte_en;
reg     [31:0]  mem[0:4095];
reg     [31:0]  mem_out;
integer i;
initial
begin
    for(i=0;i<4095;i=i+1) mem[i] = 0;
end
initial
begin
    $readmemh("D:/code/my_rv32i_v1/inst.dat",mem);
end

assign im_dout = (im_addr[31]&(~(|im_addr[30:14]))) ? mem[im_addr[13:2]]:32'h0;

always@(*)
begin
    case(dm_addr[1:0])
    2'b00:  mem_out = mem[dm_addr[13:2]][31:0];
    2'b01:  mem_out = {8'h0,mem[dm_addr[13:2]][31:8]};
    2'b10:  mem_out = {16'h0,mem[dm_addr[13:2]][31:16]};
    2'b11:  mem_out = {24'h0,mem[dm_addr[13:2]][31:24]};
    endcase
end
always@(*)
begin
    case(dm_rd_ctrl)
    3'h5:   dm_dout = mem_out;                              //is_lw
    3'h4:   dm_dout = {16'h0,mem_out[15:0]};                //is_lhu
    3'h3:   dm_dout = {{16{mem_out[15]}},mem_out[15:0]};    //is_lh
    3'h2:   dm_dout = {24'h0,mem_out[7:0]};                 //is_lbu
    3'h1:   dm_dout = {{24{mem_out[7]}},mem_out[7:0]};      //is_lb
    3'h0:   dm_dout = 32'h0;                                //other
    default:dm_dout = 32'h0;
    endcase
end
```

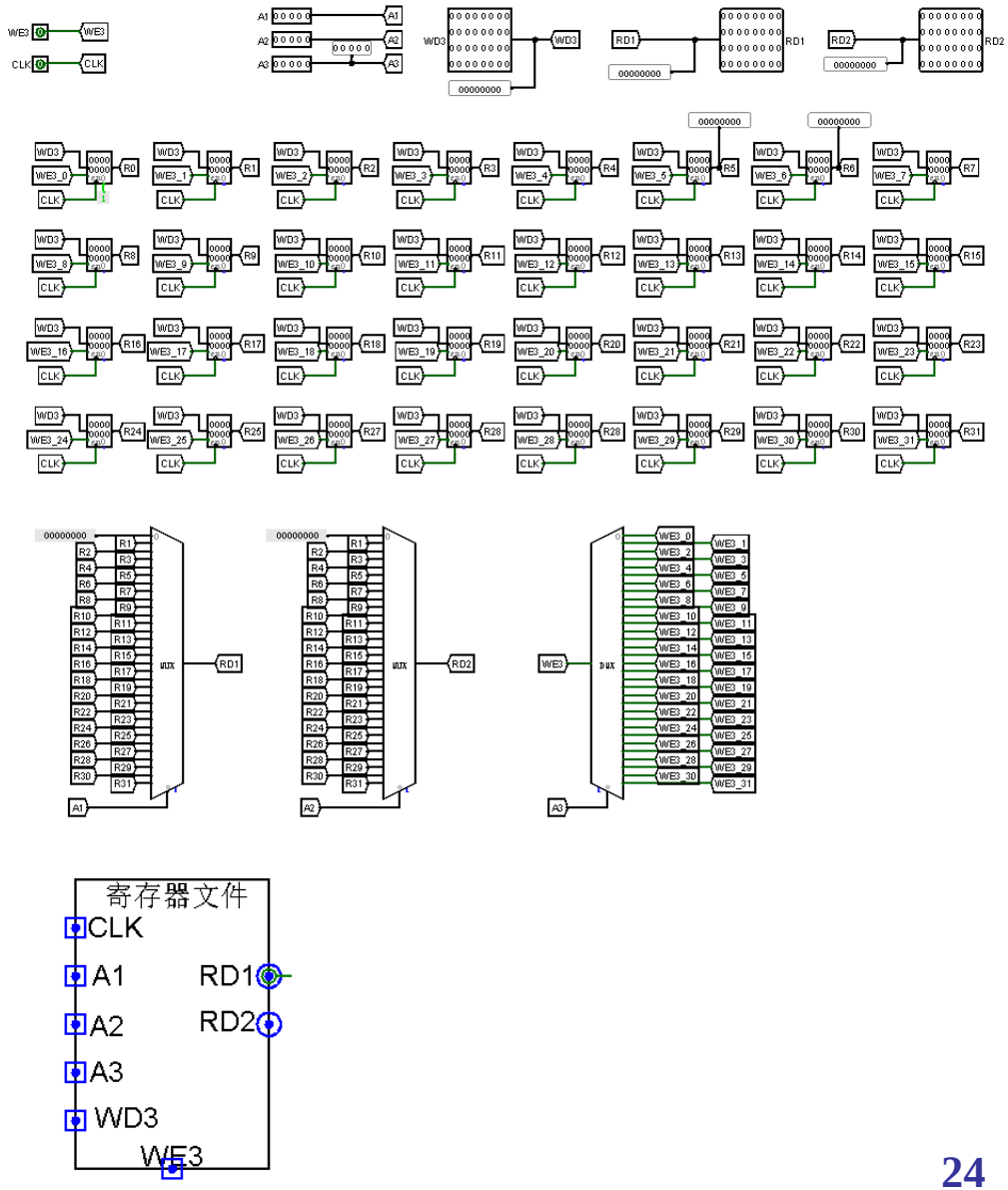



```
always@(*)
begin
    if(dm_wr_ctrl == 2'b11)
        byte_en = 4'b1111;
    else if(dm_wr_ctrl == 2'b10)
    begin
        if(dm_addr[1] == 1'b1)
            byte_en = 4'b1100;
        else
            byte_en = 4'b0011;
    end
    else if(dm_wr_ctrl == 2'b01)
    begin
        case(dm_addr[1:0])
        2'b00:  byte_en = 4'b0001;
        2'b01:  byte_en = 4'b0010;
        2'b10:  byte_en = 4'b0100;
        2'b11:  byte_en = 4'b1000;
        endcase
    end
    else
        byte_en = 4'b0000;
end

always@(posedge clk)
begin
    if((byte_en != 1'b0)&&(dm_addr[30:12]==19'b0))
    begin
        if(byte_en == 4'b1111)      mem[dm_addr[13:2]] <= dm_din;
        else if(byte_en == 4'b0011) mem[dm_addr[13:2]] <= {mem[dm_addr[13:2]][31:16],dm_din[15:0]};
        else if(byte_en == 4'b1100) mem[dm_addr[13:2]] <= {dm_din[15:0],mem[dm_addr[13:2]][15:0]};
        else if(byte_en == 4'b0001) mem[dm_addr[13:2]] <= {mem[dm_addr[13:2]][31:8],dm_din[7:0]};
        else if(byte_en == 4'b0010) mem[dm_addr[13:2]] <= {mem[dm_addr[13:2]][31:16],dm_din[7:0],mem[dm_addr[13:2]][7:0]};
        else if(byte_en == 4'b0100) mem[dm_addr[13:2]] <= {mem[dm_addr[13:2]][31:24],dm_din[7:0],mem[dm_addr[13:2]][15:0]};
        else if(byte_en == 4'b1000) mem[dm_addr[13:2]] <= {dm_din[7:0],mem[dm_addr[13:2]][23:0]};
    end
end
endmodule
```

4.4a Verilog实现

寄存器文件模块

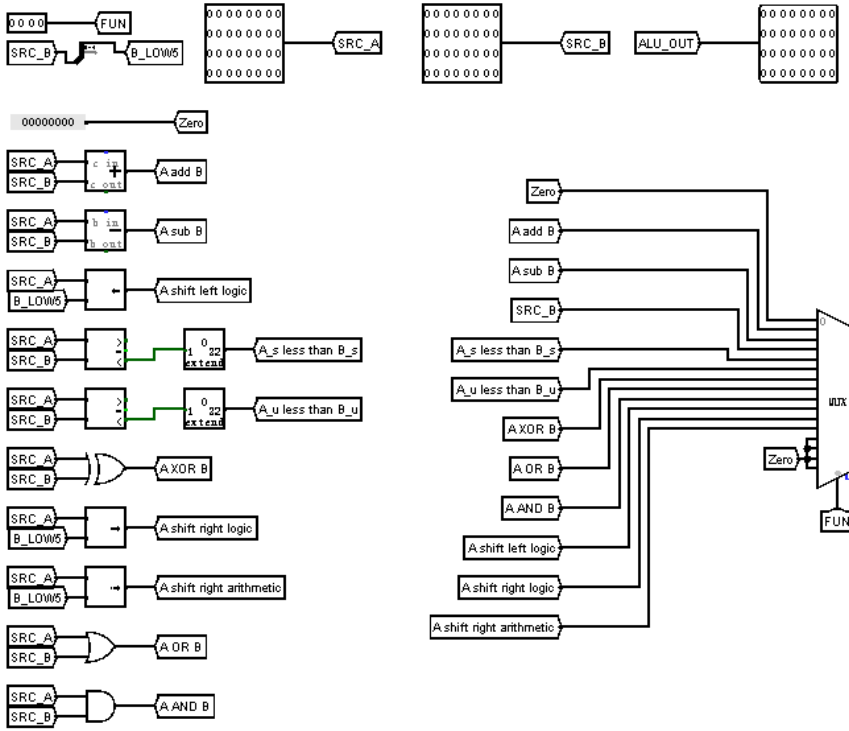
```
1  module reg_file(
2  input          clk,
3  input  [4:0]   a1,a2,a3,
4  input  [31:0]  wd3,
5  input          we3,
6  output [31:0]  rd1,rd2
7  );
8  reg [31:0] reg_file[0:31];
9  integer i;
10 initial
11 begin
12     for(i=0;i<32;i=i+1) reg_file[i] = 0;
13 end
14 always@(posedge clk)
15 begin
16     if((we3)&&(|a3))//写有效，且地址不为0
17         reg_file[a3] <= wd3;
18 end
19
20 assign rd1 = a1 ? reg_file[a1] : 32'h0;
21 assign rd2 = a2 ? reg_file[a2] : 32'h0;
22
23 endmodule
```

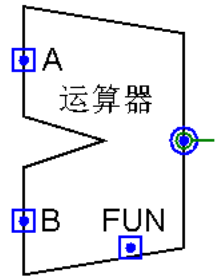


4.4a Verilog实现

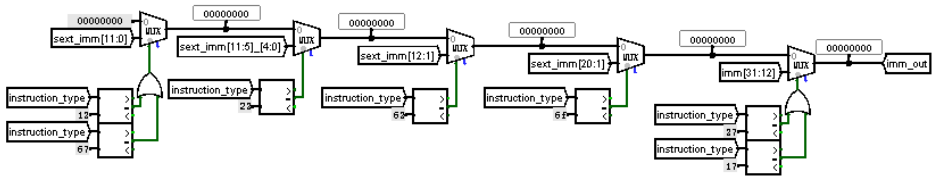
- ALU模块

```
1  module alu(
2  input [31:0] a,b,
3  input [3:0]  alu_ctrl,
4  output reg [31:0] alu_out
5  );
6
7  wire signed [31:0] signed_a;
8  wire signed [31:0] signed_b;
9
10 assign signed_a = a;
11 assign signed_b = b;
12
13 always@(*)
14 begin
15     case(alu_ctrl)
16     4'h0:    alu_out = 32'h0;
17     4'h1:    alu_out = a + b;
18     4'h2:    alu_out = a - b;
19     4'h3:    alu_out = b;
20     4'h4:    alu_out = (signed_a < signed_b) ? 32'b1 : 32'b0;
21     4'h5:    alu_out = (a < b) ? 32'b1 : 32'b0;
22     4'h6:    alu_out = a ^ b;
23     4'h7:    alu_out = a | b;
24     4'h8:    alu_out = a & b;
25     4'h9:    alu_out = a << b[4:0];
26     4'ha:    alu_out = a >> b[4:0];
27     //4'hb:    alu_out = {{b[4:0]{a[31]}},a>>b[4:0]};
28     4'hb:    alu_out = ({{31{a[31]}},1'b0} << (~b[4:0])) | (a >> b[4:0]) ;
29     default:alu_out = 32'h0;
30     endcase
31 end
32
33 endmodule
```

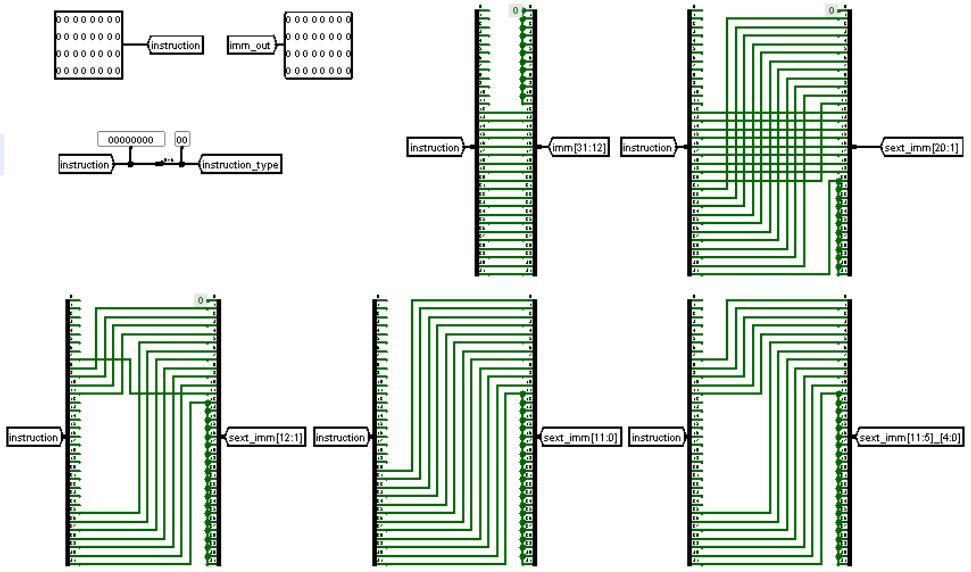




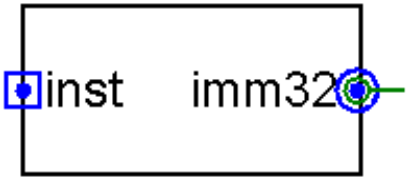
4.4a Verilog实现



■ 立即数扩展模块

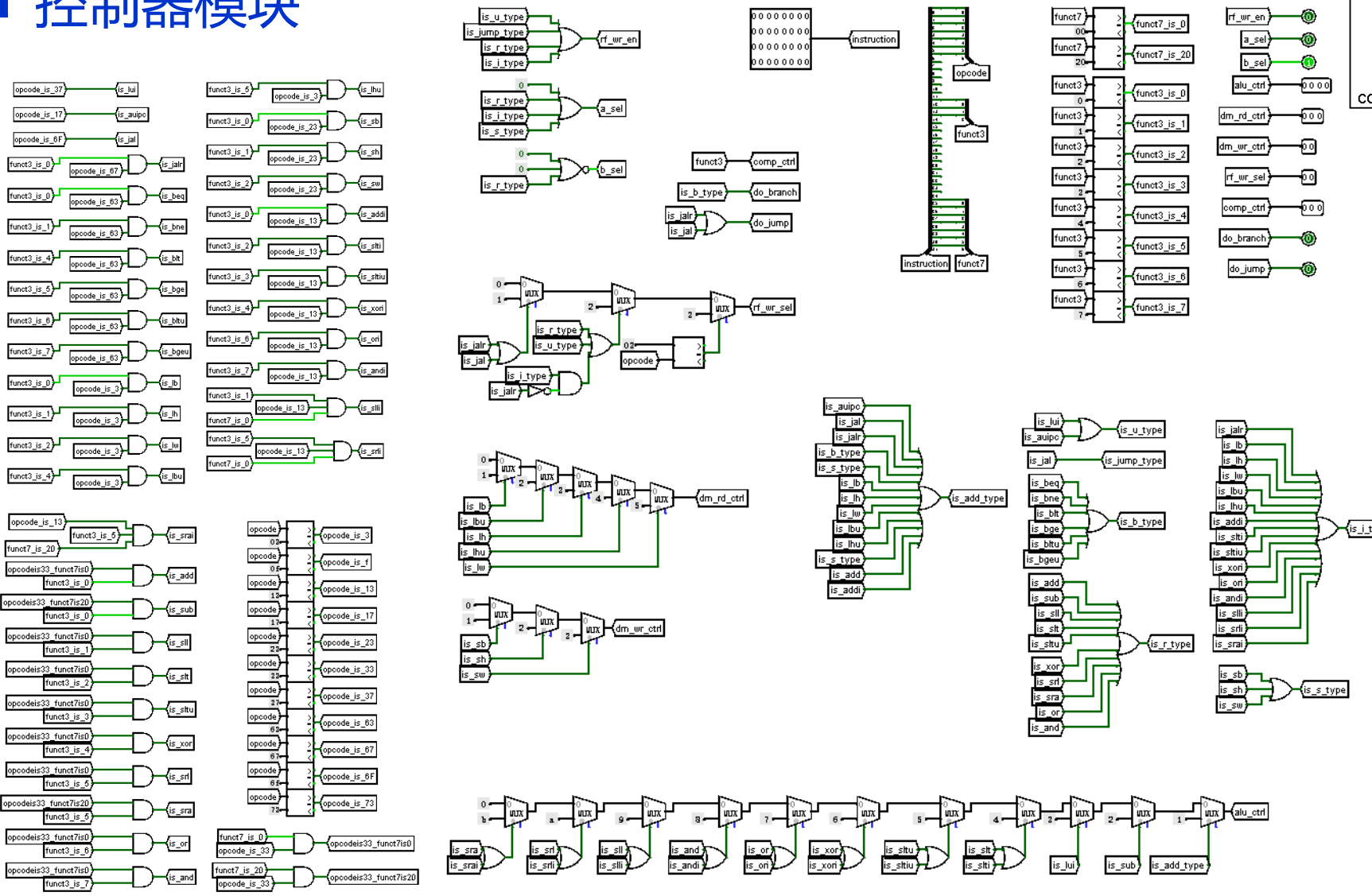


```
module imm(
input        [31:0] inst,
output reg   [31:0] imm_out
);
wire     [6:0] inst_type;

assign  inst_type = inst[6:0];

always@(*)
begin
    case(inst_type)
    7'h03: imm_out = {{21{inst[31]}},inst[30:20]};//i_type:lb、lh、lw、lbu、lhu
    7'h13: imm_out = {{21{inst[31]}},inst[30:20]};//i_type_logic and i_type_shift
    7'h17: imm_out = {inst[31:12],12'h0};//auipc
    7'h37: imm_out = {inst[31:12],12'h0};//lui
    7'h6f: imm_out = {{13{inst[31]}},inst[19:12],inst[20],inst[30:21],1'b0};//jal
    7'h63: imm_out = {{20{inst[31]}},inst[7],inst[30:25],inst[11:8],1'b0};//b_type
    7'h23: imm_out = {{21{inst[31]}},inst[30:25],inst[11:7]};//s_type
    7'h67: imm_out = {{21{inst[31]}},inst[30:20]};//jalr
    default: imm_out = 32'h0;
    endcase
end

endmodule
```



4.4a Verilog实现

- 控制器模块



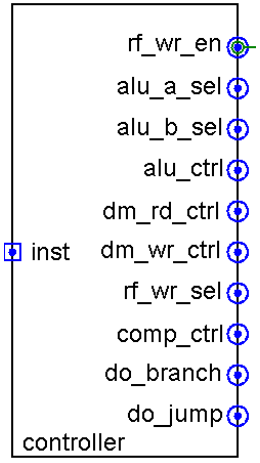
4.4a Verilog实现

```
wire    is_lui;
wire    is_auipc;
wire    is_jal;
wire    is_jalr;
wire    is_beq;
wire    is_bne;
wire    is_blt;
wire    is_bge;
wire    is_bltu;
wire    is_bgeu;
wire    is_lb;
wire    is_lh;
wire    is_lw;
wire    is_lbu;
wire    is_lhu;
wire    is_sb;
wire    is_sh;
wire    is_sw;
wire    is_addi;
wire    is_slti;
wire    is_sltiu;
wire    is_xori;
wire    is_ori;
wire    is_andi;
wire    is_slli;
wire    is_srli;
wire    is_srai;
wire    is_add;
wire    is_sub;
wire    is_sll;
wire    is_slt;
wire    is_sltu;
wire    is_xor;
wire    is_srl;
wire    is_sra;
wire    is_or;
wire    is_and;

wire    is_add_type;
wire    is_u_type;
wire    is_jump_type;
wire    is_b_type;
wire    is_r_type;
wire    is_i_type;
wire    is_s_type;
```

```
assign  opcode  = inst[6:0];
assign  funct7  = inst[31:25];
assign  funct3  = inst[14:12];

assign  is_lui  = (opcode == 7'h37) ;
assign  is_auipc= (opcode == 7'h17) ;
assign  is_jal  = (opcode == 7'h6F) ;
assign  is_jalr = (opcode == 7'h67) && (funct3 ==3'h0) ;
assign  is_beq  = (opcode == 7'h63) && (funct3 ==3'h0) ;
assign  is_bne  = (opcode == 7'h63) && (funct3 ==3'h1) ;
assign  is_blt  = (opcode == 7'h63) && (funct3 ==3'h4) ;
assign  is_bge  = (opcode == 7'h63) && (funct3 ==3'h5) ;
assign  is_bltu = (opcode == 7'h63) && (funct3 ==3'h6) ;
assign  is_bgeu = (opcode == 7'h63) && (funct3 ==3'h7) ;
assign  is_lb   = (opcode == 7'h03) && (funct3 ==3'h0) ;
assign  is_lh   = (opcode == 7'h03) && (funct3 ==3'h1) ;
assign  is_lw   = (opcode == 7'h03) && (funct3 ==3'h2) ;
assign  is_lbu  = (opcode == 7'h03) && (funct3 ==3'h4) ;
assign  is_lhu  = (opcode == 7'h03) && (funct3 ==3'h5) ;
assign  is_sb   = (opcode == 7'h23) && (funct3 ==3'h0) ;
assign  is_sh   = (opcode == 7'h23) && (funct3 ==3'h1) ;
assign  is_sw   = (opcode == 7'h23) && (funct3 ==3'h2) ;
assign  is_addi = (opcode == 7'h13) && (funct3 ==3'h0) ;
assign  is_slti = (opcode == 7'h13) && (funct3 ==3'h2) ;
assign  is_sltiu= (opcode == 7'h13) && (funct3 ==3'h3) ;
assign  is_xori = (opcode == 7'h13) && (funct3 ==3'h4) ;
assign  is_ori  = (opcode == 7'h13) && (funct3 ==3'h6) ;
assign  is_andi = (opcode == 7'h13) && (funct3 ==3'h7) ;
assign  is_slli = (opcode == 7'h13) && (funct3 ==3'h1) && (funct7 == 7'h00);
assign  is_srli = (opcode == 7'h13) && (funct3 ==3'h5) && (funct7 == 7'h00);
assign  is_srai = (opcode == 7'h13) && (funct3 ==3'h5) && (funct7 == 7'h20);
assign  is_add  = (opcode == 7'h33) && (funct3 ==3'h0) && (funct7 == 7'h00);
assign  is_sub  = (opcode == 7'h33) && (funct3 ==3'h0) && (funct7 == 7'h20);
assign  is_sll  = (opcode == 7'h33) && (funct3 ==3'h1) && (funct7 == 7'h00);
assign  is_slt  = (opcode == 7'h33) && (funct3 ==3'h2) && (funct7 == 7'h00);
assign  is_sltu = (opcode == 7'h33) && (funct3 ==3'h3) && (funct7 == 7'h00);
assign  is_xor  = (opcode == 7'h33) && (funct3 ==3'h4) && (funct7 == 7'h00);
assign  is_srl  = (opcode == 7'h33) && (funct3 ==3'h5) && (funct7 == 7'h00);
assign  is_sra  = (opcode == 7'h33) && (funct3 ==3'h5) && (funct7 == 7'h20);
assign  is_or   = (opcode == 7'h33) && (funct3 ==3'h6) && (funct7 == 7'h00);
assign  is_and  = (opcode == 7'h33) && (funct3 ==3'h7) && (funct7 == 7'h00);
```

■ 控制器模块

```
module controller(
input      [31:0]  inst,
output             rf_wr_en,
output             alu_a_sel,
output             alu_b_sel,
output reg  [3:0]  alu_ctrl,
output reg  [2:0]  dm_rd_ctrl,
output reg  [1:0]  dm_wr_ctrl,
output reg  [1:0]  rf_wr_sel,
output      [2:0]  comp_ctrl,
output             do_branch,
output             do_jump
);

wire    [6:0]   opcode;
wire    [2:0]   funct3;
wire    [6:0]   funct7;
```



4.4a Verilog实现

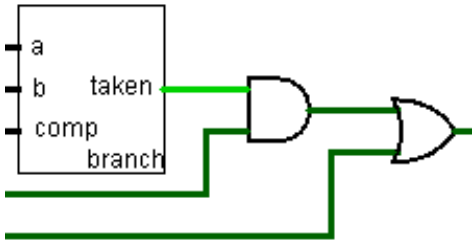
- 控制器模块-续

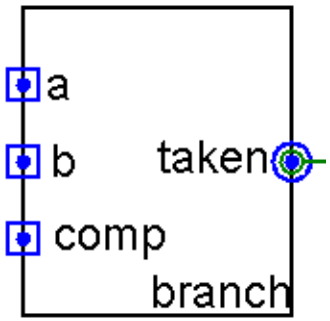
```
107  assign  is_add_type = is_auipc | is_jal | is_jalr | is_b_type | is_s_type
108                      | is_lb | is_lh | is_lw | is_lbu | is_lhu | is_s_type
109                      | is_add | is_addi ;
110  assign  is_u_type   = is_lui | is_auipc ;
111  assign  is_jump_type= is_jal ;
112  assign  is_b_type   = is_beq | is_bne | is_blt | is_bge | is_bltu | is_bgeu ;
113  assign  is_r_type   = is_add | is_sub | is_sll | is_slt | is_sltu | is_xor
114                      | is_srl | is_sra | is_or | is_and ;
115  assign  is_i_type   = is_jalr | is_lb | is_lh | is_lw | is_lbu | is_lhu
116                      | is_addi | is_slti | is_sltiu | is_xori | is_ori | is_andi
117                      | is_slli | is_srli | is_srai ;
118  assign  is_s_type   = is_sb | is_sh | is_sw ;
119
120
121  always@(*)
122  begin
123      if(is_add_type)             alu_ctrl = 4'h1;
124      else if(is_sub)             alu_ctrl = 4'h2;
125      else if(is_lui)             alu_ctrl = 4'h3;
126      else if(is_slt | is_slti)   alu_ctrl = 4'h4;
127      else if(is_sltu | is_sltiu) alu_ctrl = 4'h5;
128      else if(is_xor | is_xori)   alu_ctrl = 4'h6;
129      else if(is_or | is_ori)     alu_ctrl = 4'h7;
130      else if(is_and | is_andi)   alu_ctrl = 4'h8;
131      else if(is_sll | is_slli)   alu_ctrl = 4'h9;
132      else if(is_srl | is_srli)   alu_ctrl = 4'ha;
133      else if(is_sra | is_srai)   alu_ctrl = 4'hb;
134      else                        alu_ctrl = 4'h0;
135  end
136
137  assign rf_wr_en     = is_u_type | is_jump_type | is_r_type | is_i_type ;
138  assign alu_a_sel    = is_r_type | is_i_type | is_s_type;
139  assign alu_b_sel    = ~ is_r_type ;
140  //[2:0]dm_rd_ctrl,
141  always@(*)
142  begin
143      if(is_lw)       dm_rd_ctrl = 3'h5;
144      else if(is_lhu) dm_rd_ctrl = 3'h4;
145      else if(is_lh)  dm_rd_ctrl = 3'h3;
146      else if(is_lbu) dm_rd_ctrl = 3'h2;
147      else if(is_lb)  dm_rd_ctrl = 3'h1;
148      else            dm_rd_ctrl = 3'h0;
149  end
```

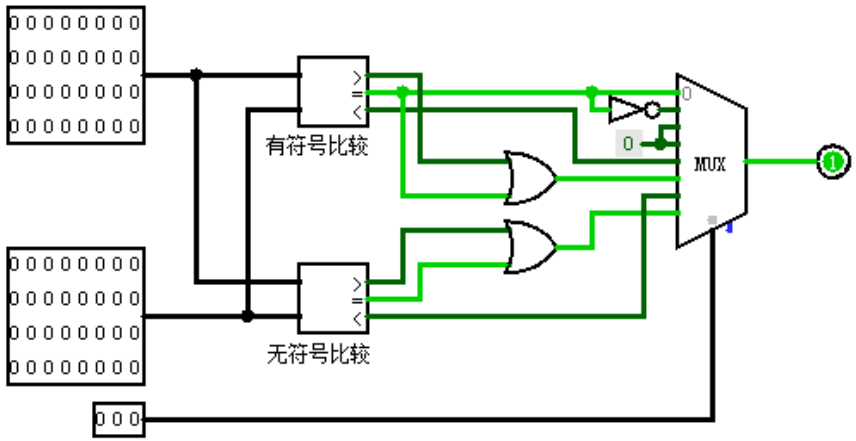
```
150  //[1:0]dm_wr_ctrl,
151  always@(*)
152  begin
153      if(is_sw)       dm_wr_ctrl = 2'h3;
154      else if(is_sh)  dm_wr_ctrl = 2'h2;
155      else if(is_sb)  dm_wr_ctrl = 2'h1;
156      else            dm_wr_ctrl = 2'h0;
157  end
158  //[1:0]rf_wr_sel,
159  always@(*)
160  begin
161      if(opcode == 7'h3)          rf_wr_sel = 2'h3;
162      else if(((~is_jalr)&is_i_type) | is_u_type | is_r_type) rf_wr_sel = 2'h2;
163      else if(is_jalr | is_jal)   rf_wr_sel = 2'h1;
164      else                        rf_wr_sel = 2'h0;
165  end
166
167  assign comp_ctrl    = funct3;
168  assign do_branch    = is_b_type;
169  assign do_jump      = is_jal | is_jalr ;
170
171  endmodule
```

4.4a Verilog实现

- 分支模块



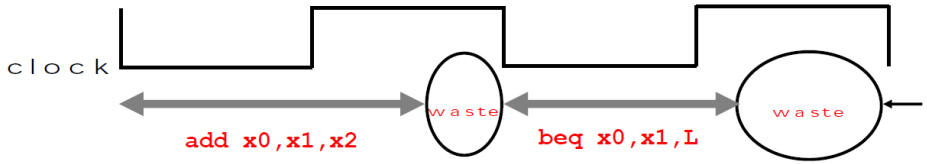




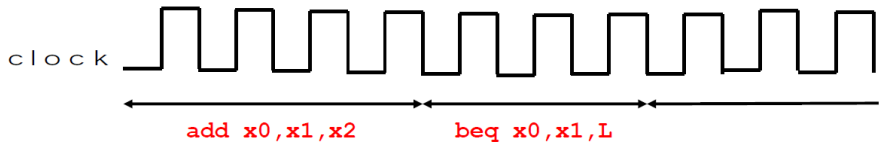

```
1  module brah(          //branch模块
2  input [31:0]     a,
3  input [31:0]     b,
4  input [2:0]      comp_ctrl,
5  input            do_branch,
6  input            do_jump,
7  output           branch
8  );
9  wire signed      [31:0] signed_a;
10 wire signed      [31:0] signed_b;
11 wire unsigned    [31:0] unsigned_a;
12 wire unsigned    [31:0] unsigned_b;
13 reg                     taken;
14
15 assign signed_a = a;
16 assign signed_b = b;
17 assign unsigned_a = a;
18 assign unsigned_b = b;
19
20 always@(*)
21 begin
22     case(comp_ctrl)
23      3'h0:  taken = (signed_a == signed_b);
24      3'h1:  taken = ~(signed_a == signed_b);
25      3'h2:  taken = 1'b0;
26      3'h3:  taken = 1'b0;
27      3'h4:  taken = (signed_a < signed_b);
28      3'h5:  taken = (signed_a >= signed_b);
29      3'h6:  taken = (unsigned_a < unsigned_b);
30      3'h7:  taken = (unsigned_a >= unsigned_b);
31      default: taken = 1'b0;
32      endcase
33 end
34
35 assign branch = (taken && do_branch) || do_jump;
36
37 endmodule
```

4.4b 多周期设计

- 单周期CPU设计的缺陷
  - 指令存储器和数据存储器必须分开
  - 所有操作必须在一个时钟周期内完成
    - 电路频率受延时最大的指令限制
    - 使用DRAM时，会受到极大限制

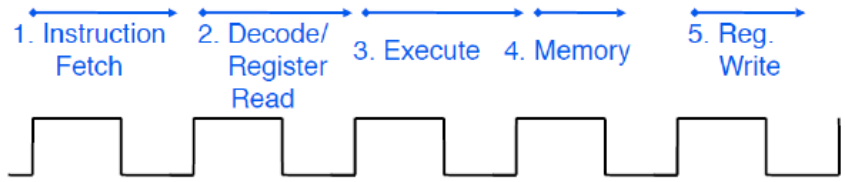


单周期



多周期

- 改进方法
  - 多周期设计方案
  - 根据指令执行所使用的功能部件，将执行过程划分成多个阶段，每个阶段一个机器周期
  - 每条指令需要占用多个机器周期，指令结束后可立即执行后面的指令
  - 通过提高时钟频率，减少指令的浪费时间，可提高性能
  - 指令数据存储器可以合并
  - 时钟周期确定标准
    - 每个周期的工作尽量平衡
    - 假设每个周期可以完成：1次内存访问、一次寄存器访问、一次ALU操作

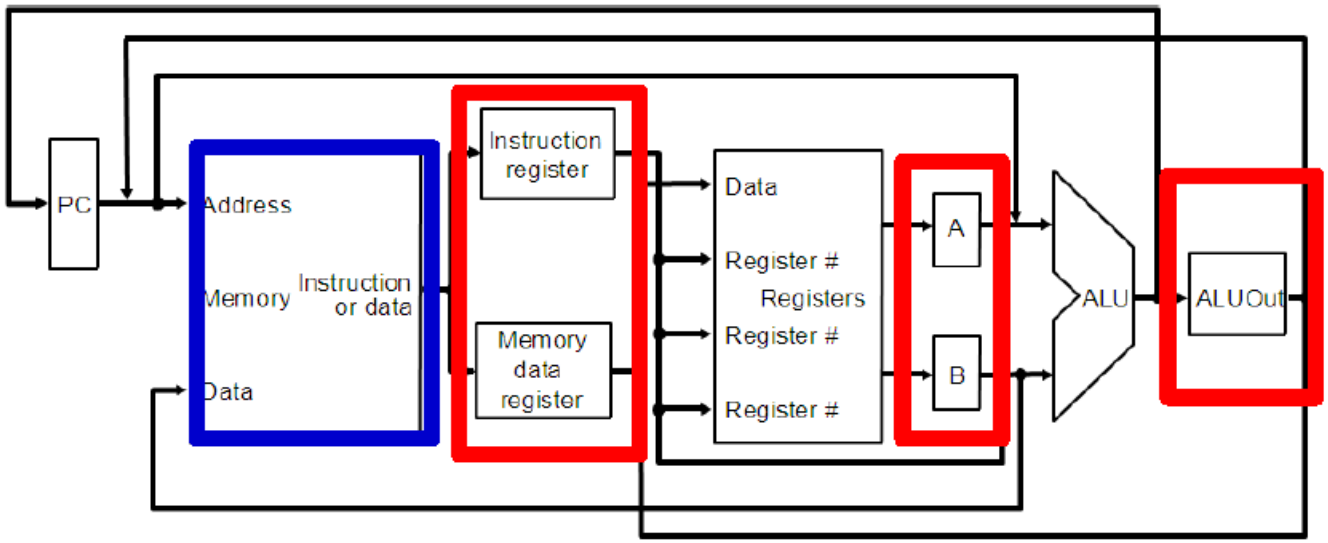


4.4b 多周期设计

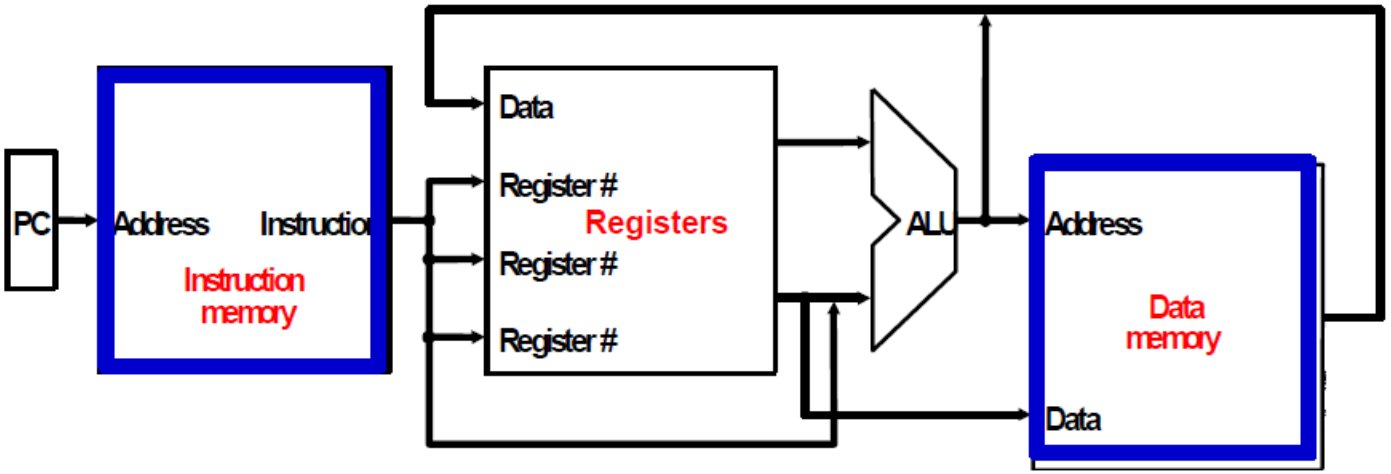
- 阶段划分，每条指令最多分为5个阶段
  - 取指
  - 译码：计算beq目标地址
  - 执行：R-type指令执行、访存地址计算、分支完成阶段
  - 访存：lw读、store和R-type指令完成阶段
  - 写回：lw完成阶段
- 注意
  - 定长机器周期：机器周期 = 时钟周期
  - 不定长机器周期：机器周期 ≠ 时钟周期

4.4b 多周期设计

- 单周期和多周期数据通路区别

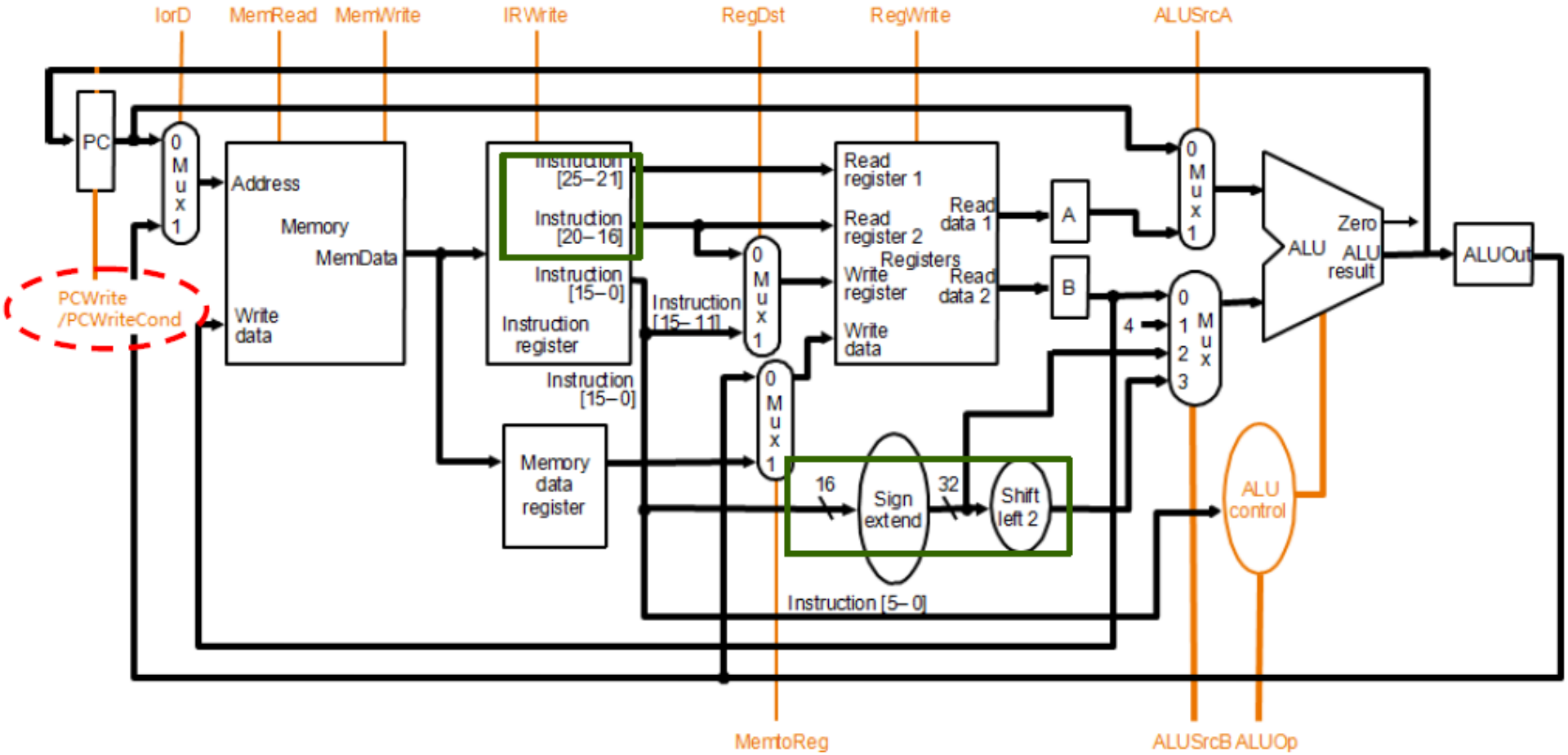


ALU、寄存器、存储器、控制信号



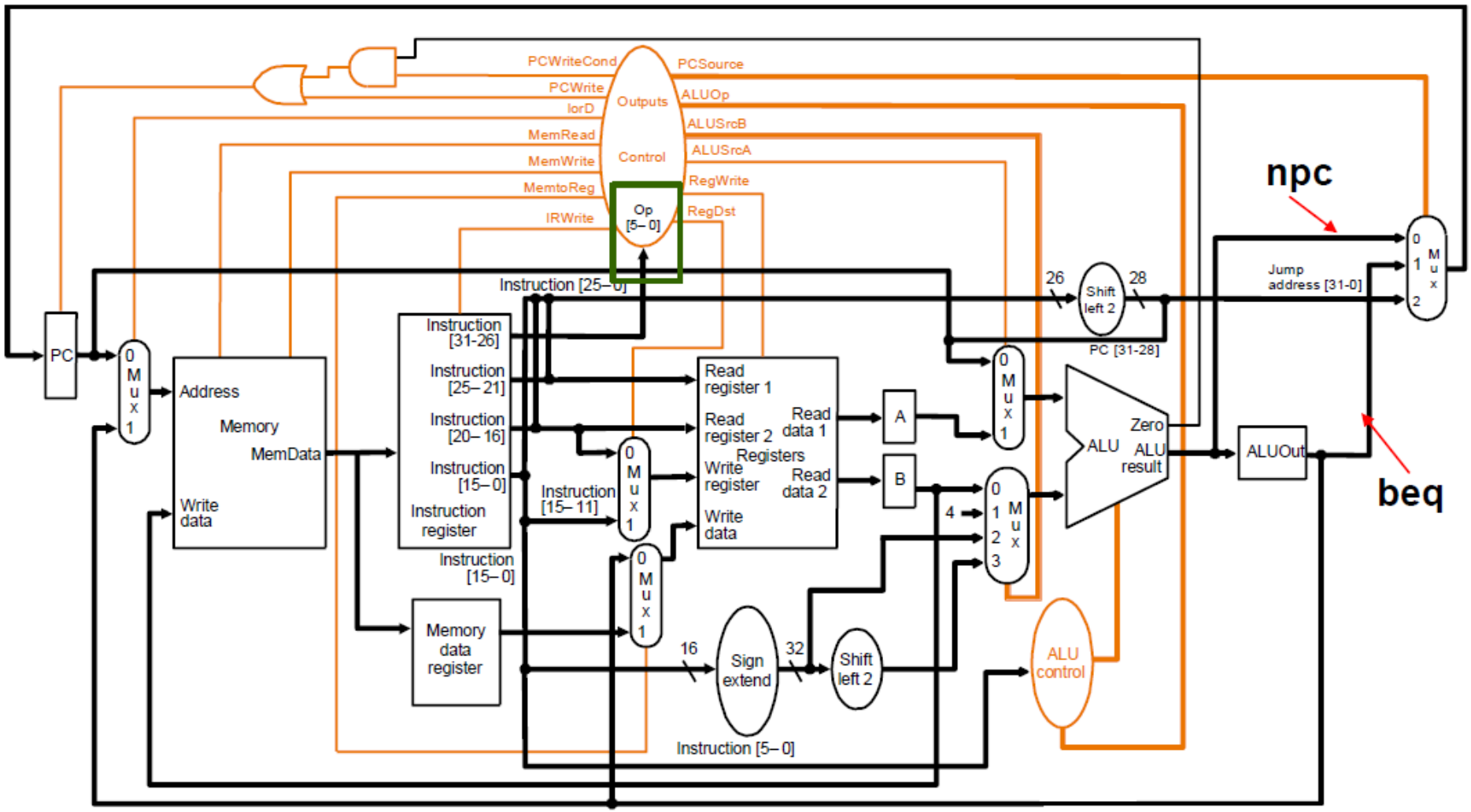
4.4b 多周期设计

- 多周期数据通路（供参考）



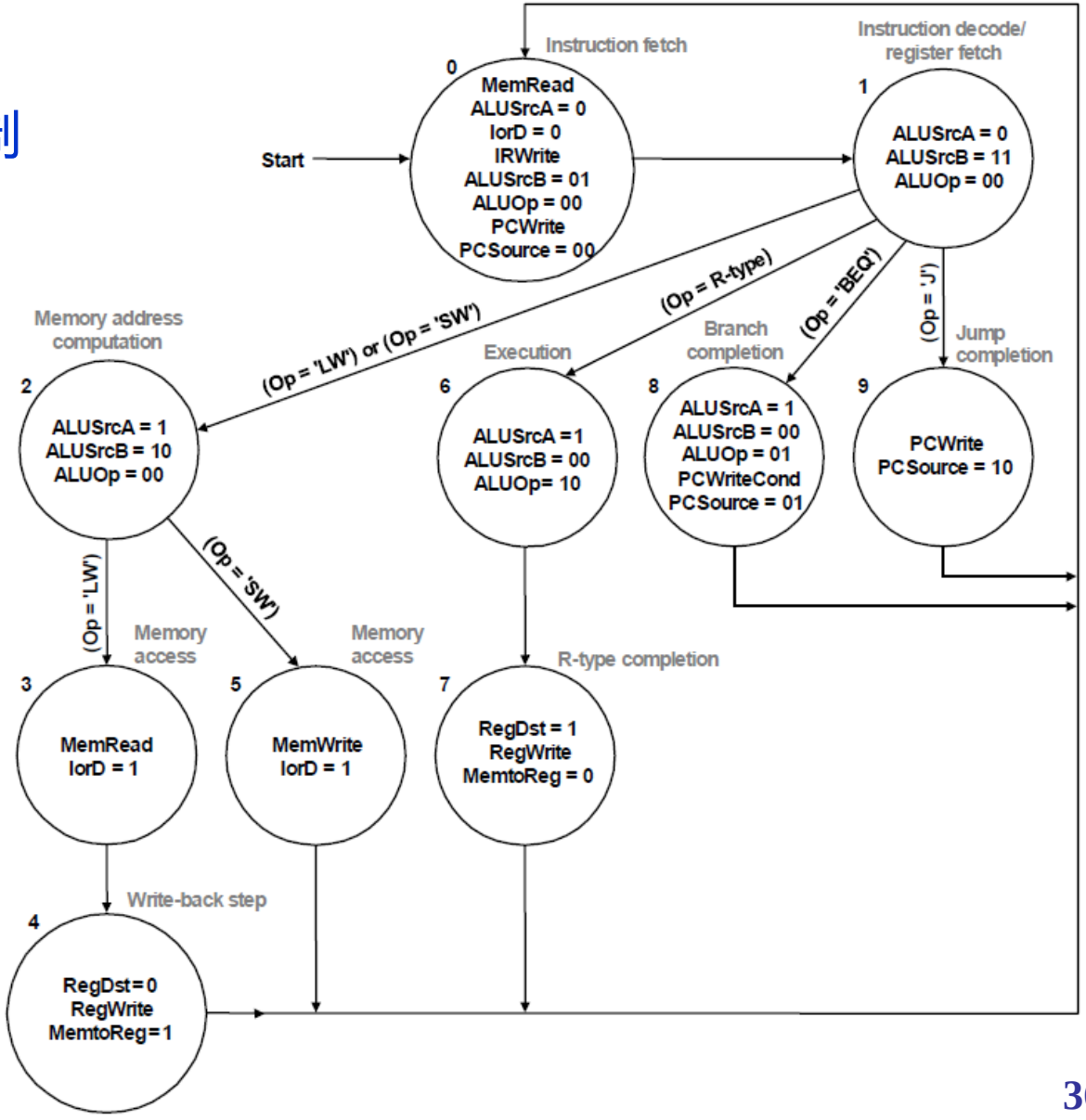
4.4b 多周期设计

- 多周期数据通路：控制器（供参考）



4.4b 多周期设计

- 控制器实现（供参考）
  - 通过有限状态机实现控制
  - J-type：3个状态
  - B-type：3个状态
  - R-type：4个状态
  - SW：4个状态
  - LW：5个状态
  - 其它指令
    - 自行分析

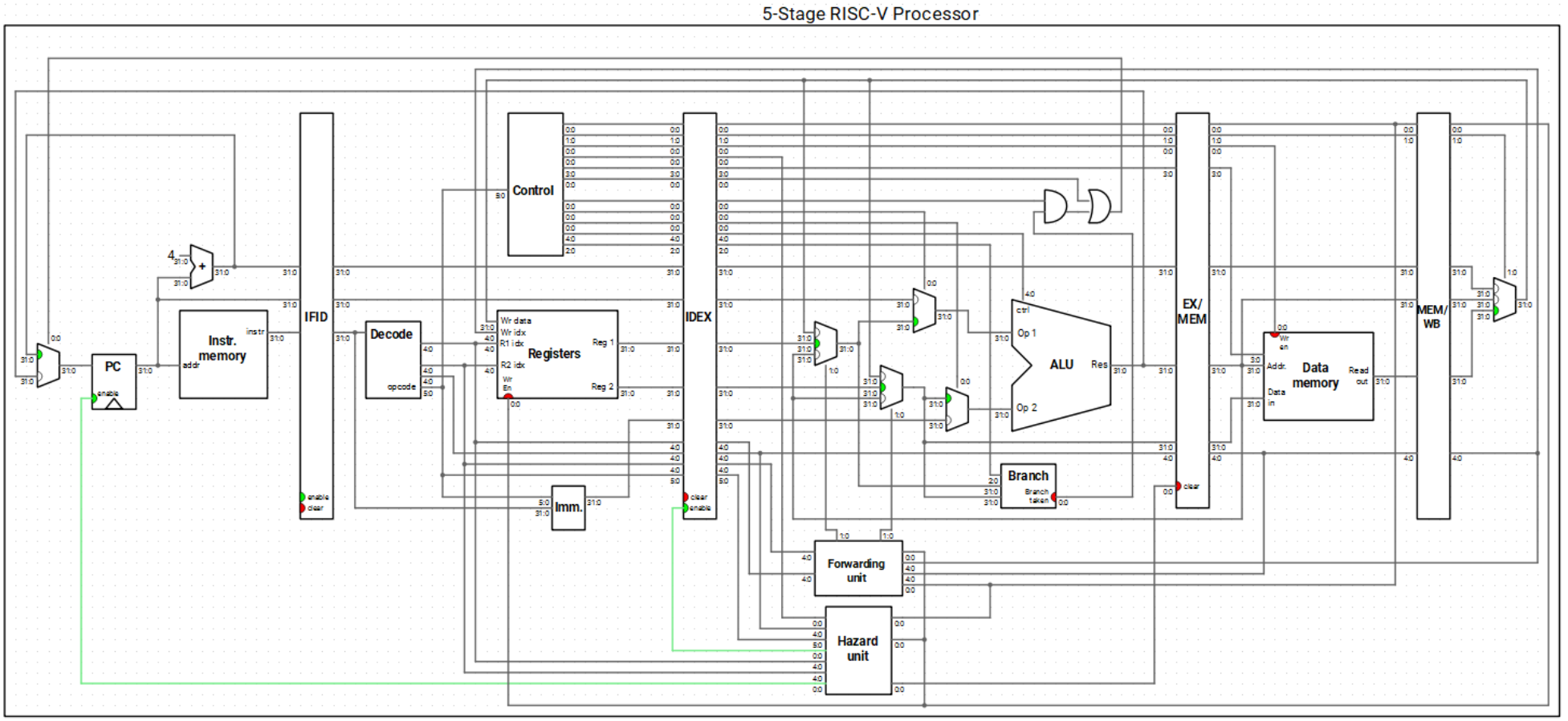


4.4b 多周期设计

- 基于Logisim的RV32I 多周期实现（待完成）
- 基于Verilog的RV32I 多周期实现（待完成）

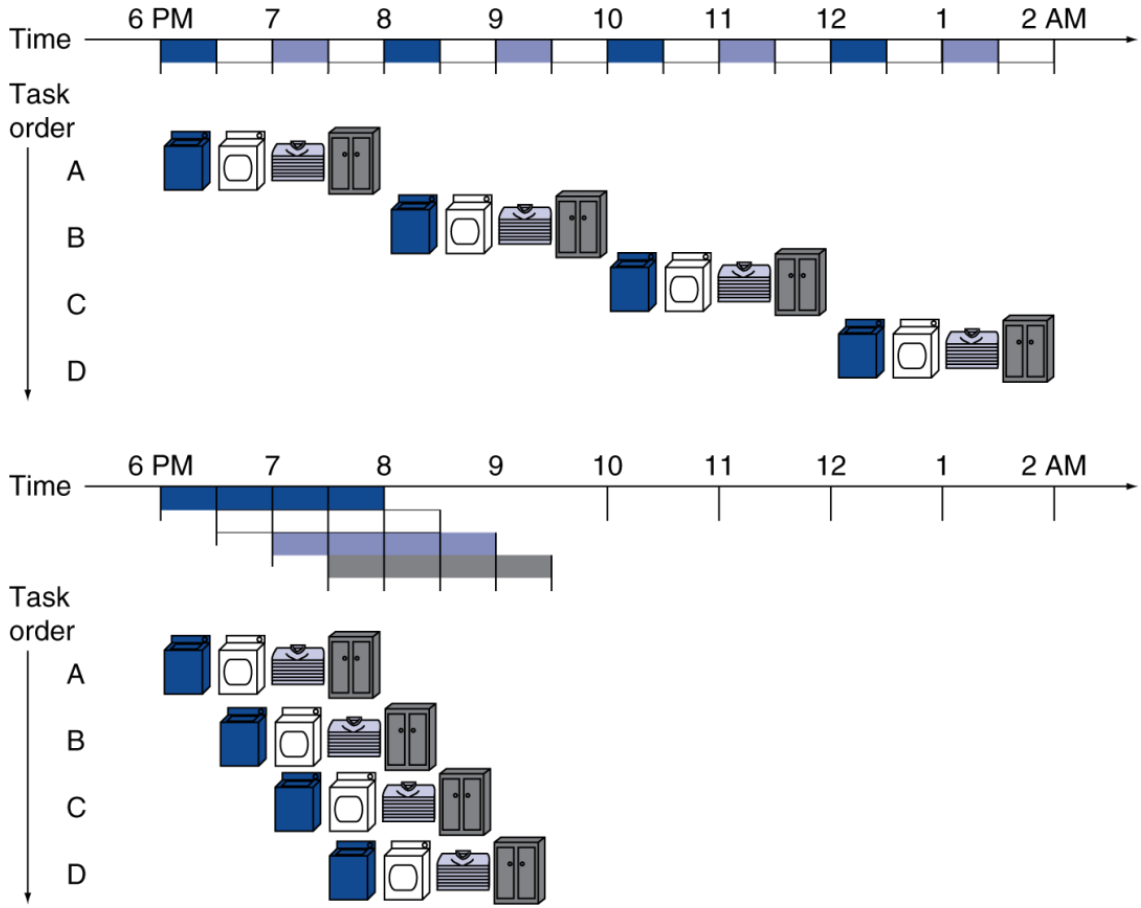
4.5 流水线概述

- 学习目标
  - 完成完整的RISC-V五级流水线CPU设计



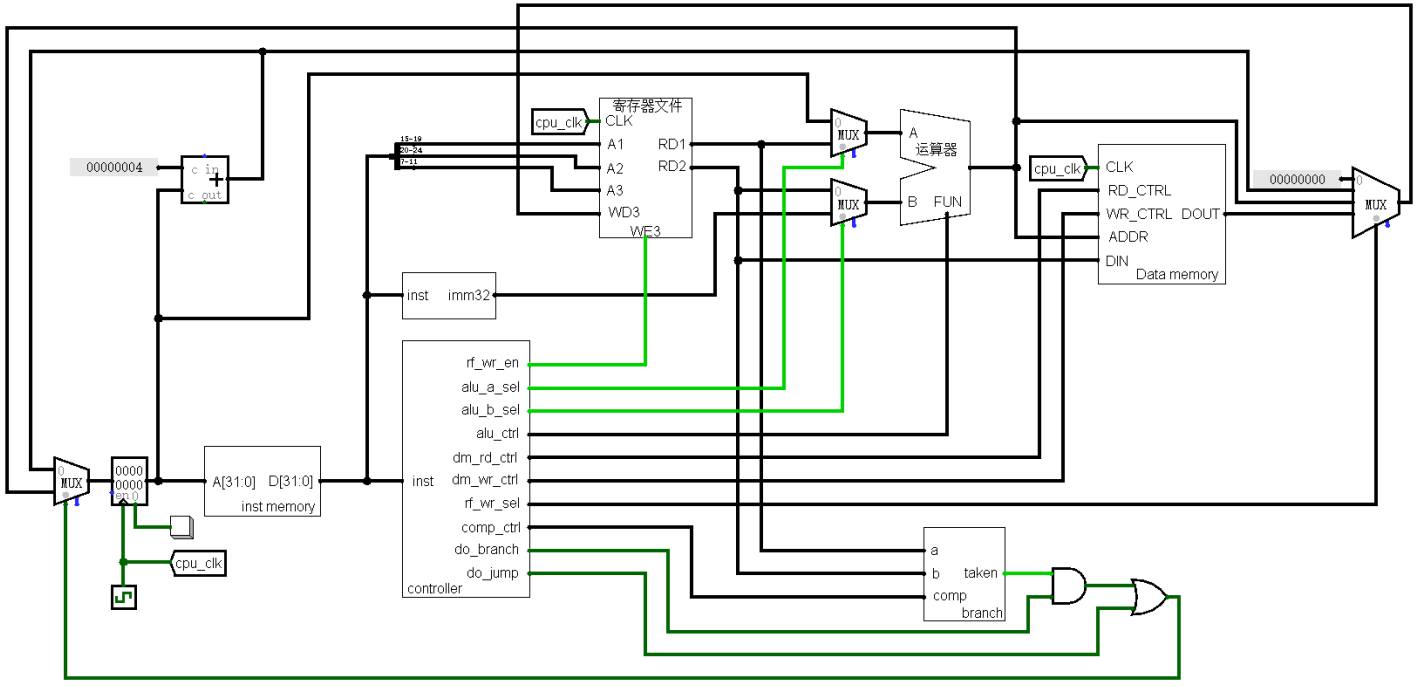
4.5 流水线概述

- 流水线：Pipeline
  - 一种实现多条指令重叠执行的技术，与生产流水线类似
- 非流水线 vs 流水线
  - 4个负载：
    - 16/7=2.3倍加速比
  - 大量负载：
    - 约4倍加速比
- 流水线特点
  - 没有缩短单个任务的时间
  - 提高了整体的吞吐率
  - 各部件高负荷运作
  - 不能跳过中间步骤



4.5 流水线概述

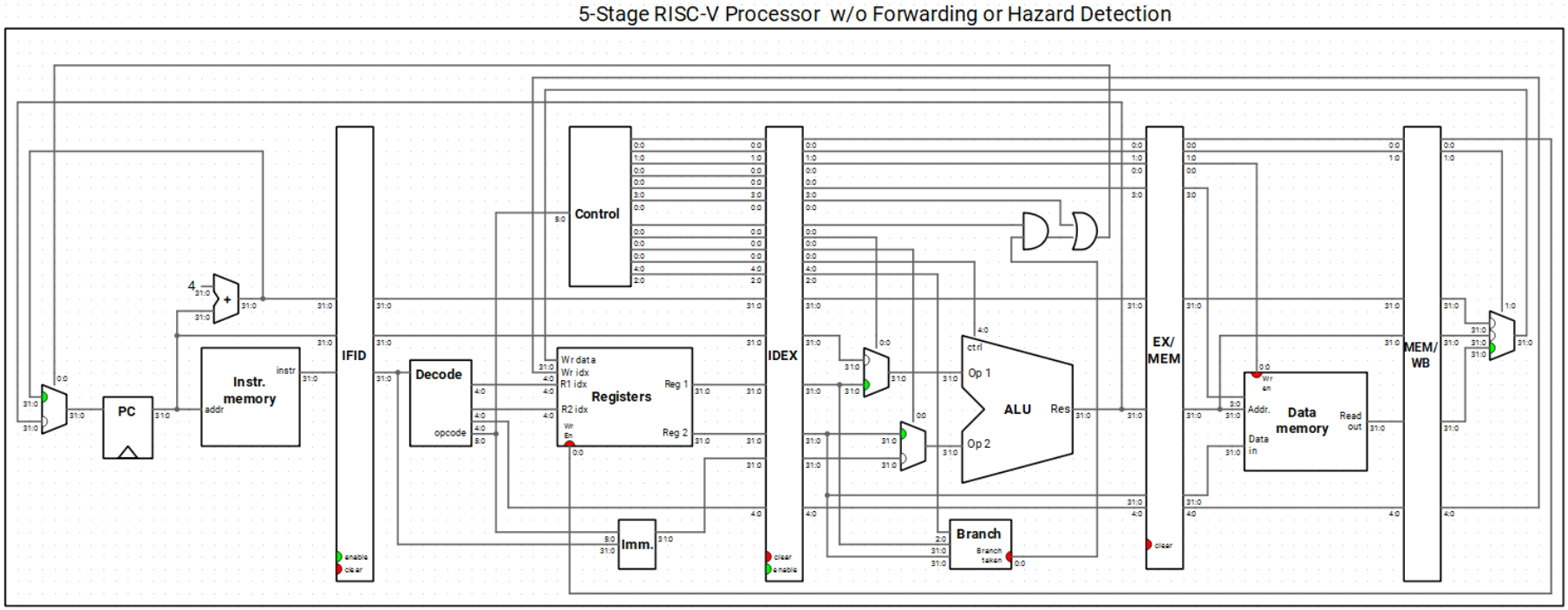
- RISC-V流水线，5个步骤
  - IF（取指）：Instruction Fetch from memory
  - ID（译码）： Instruction decode & register read
  - EX（执行）： Execute operation or calculate address
  - MEM（访存）： Access memory operand
  - WB（回写）： Write result back to register



4.5 流水线概述

- 理想流水线示例

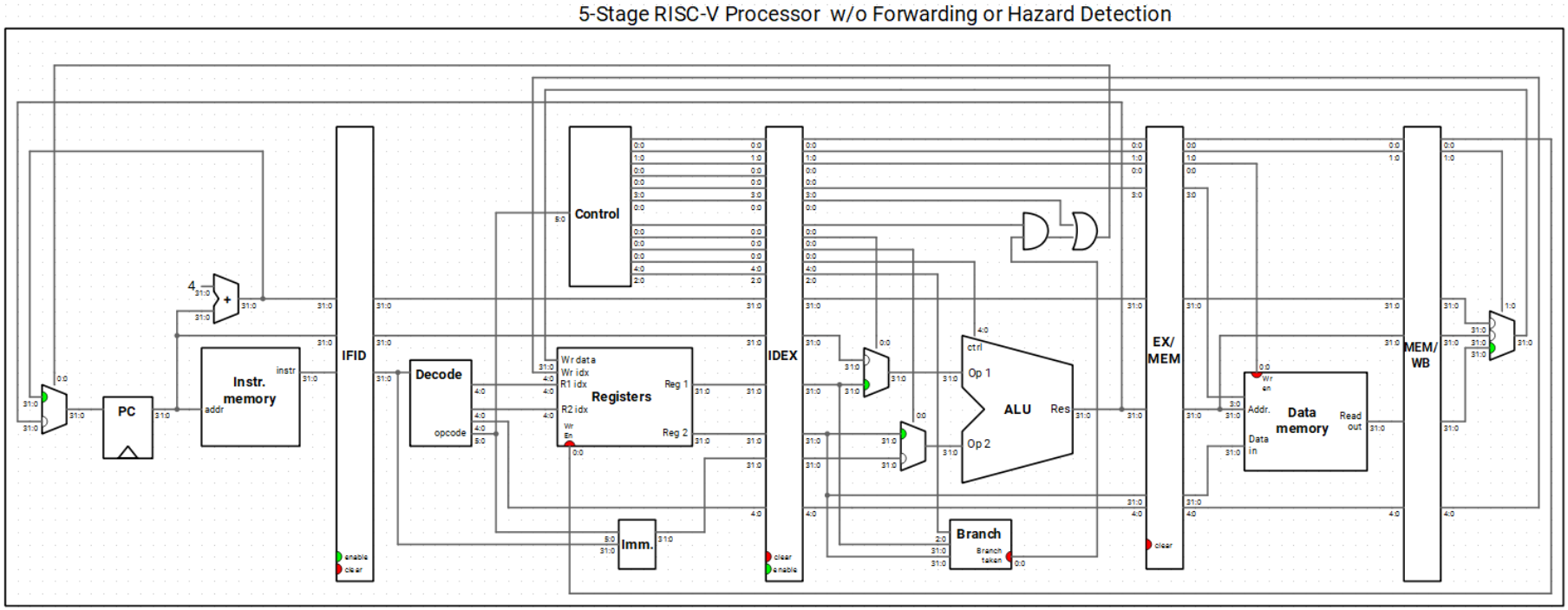
```
ADDI    X1, X0, 1
ADDI    X2, X0, 2
ADDI    X3, X0, 3
ADD     X4, X1, 4
ADD     X5, X1, X2
```



4.5 流水线概述

- 数据冒险示例_1

```
ADDI    X1, X0, 1
ADDI    X2, X0, 2
ADDI    X3, X1, 2
ADDI    X4, X3, 1
ADDI    X5, X0, 4
```



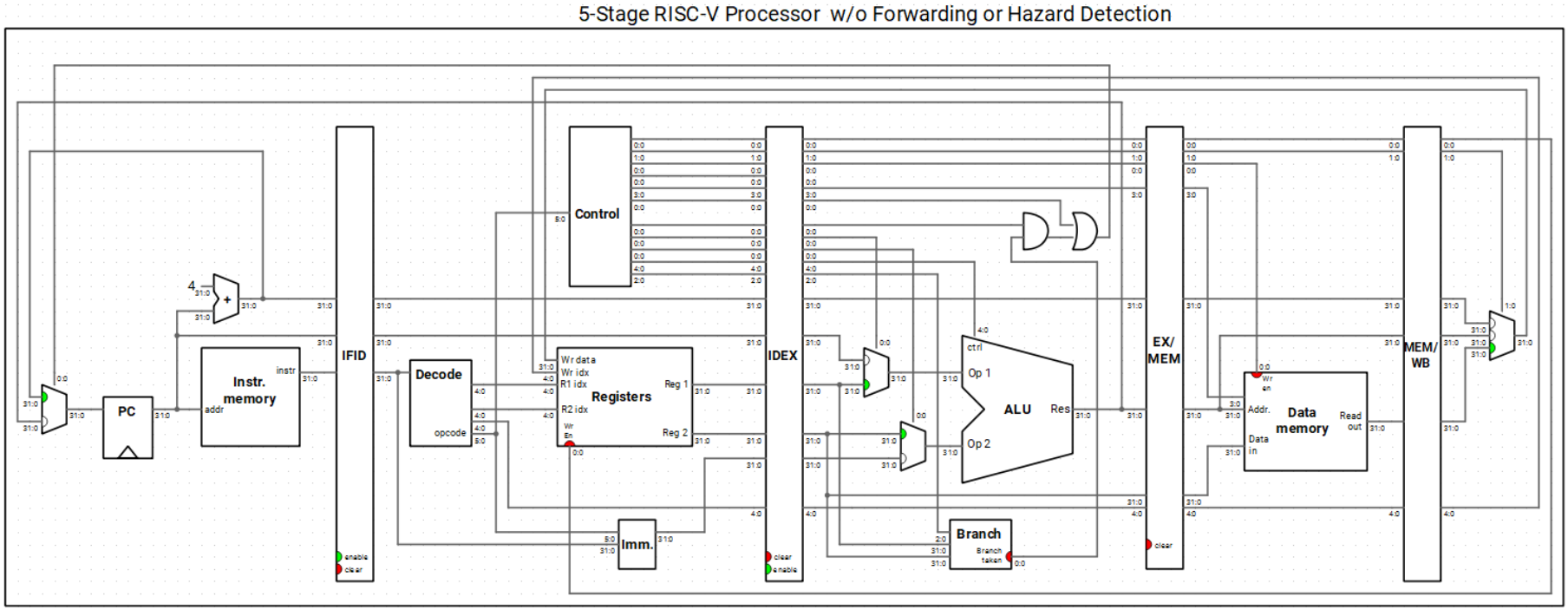
4.5 流水线概述

- 数据冒险示例_2

```
ADDI    X1, X0, 1
LW      X2, 4 (X0)
ADD     X3, X1, X2
ADDI    X4, X0, 4
ADDI    X5, X0, 5
```

数据存储器

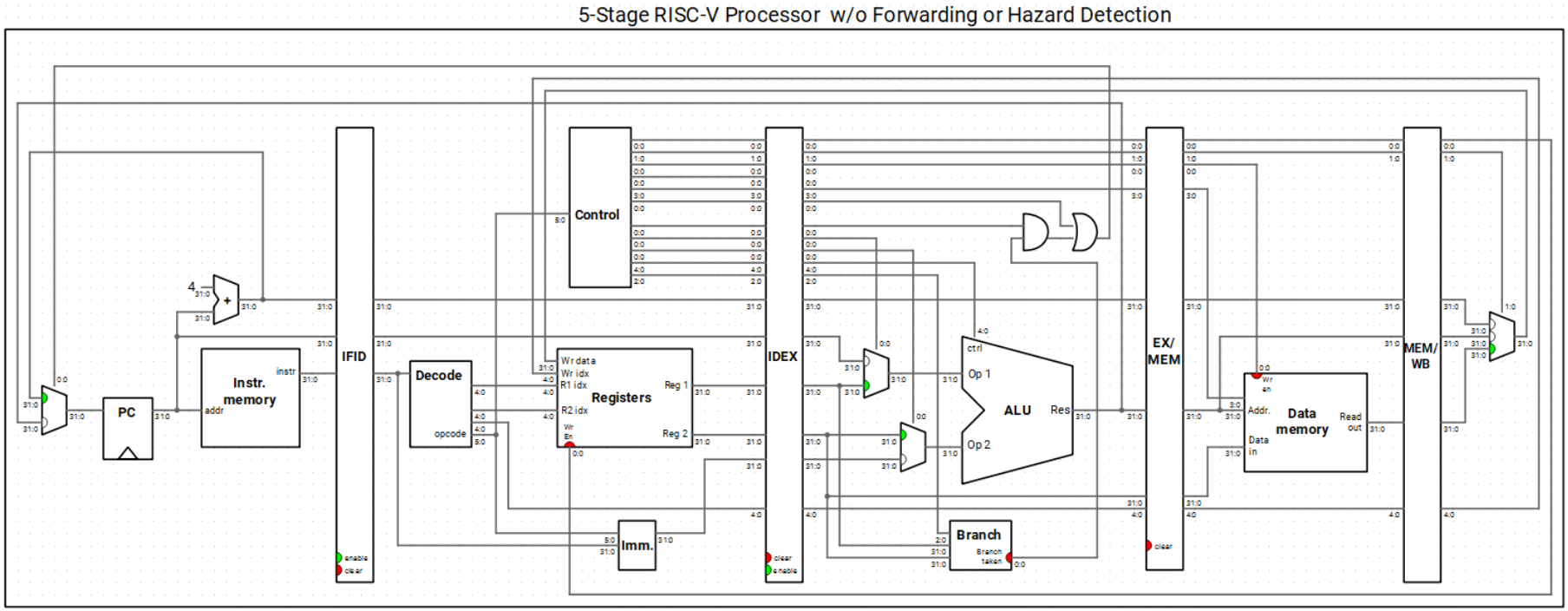
地址	数据
0	0x0000_0001
4	0x0000_0002
8	0x0000_0003



4.5 流水线概述

- 控制冒险

```
ADDI    X1，X0，1
BEQ     X2，X0，8
ADDI    X3，X0，3
ADDI    X4，X0，4
ADD     X5，X1，X2
```



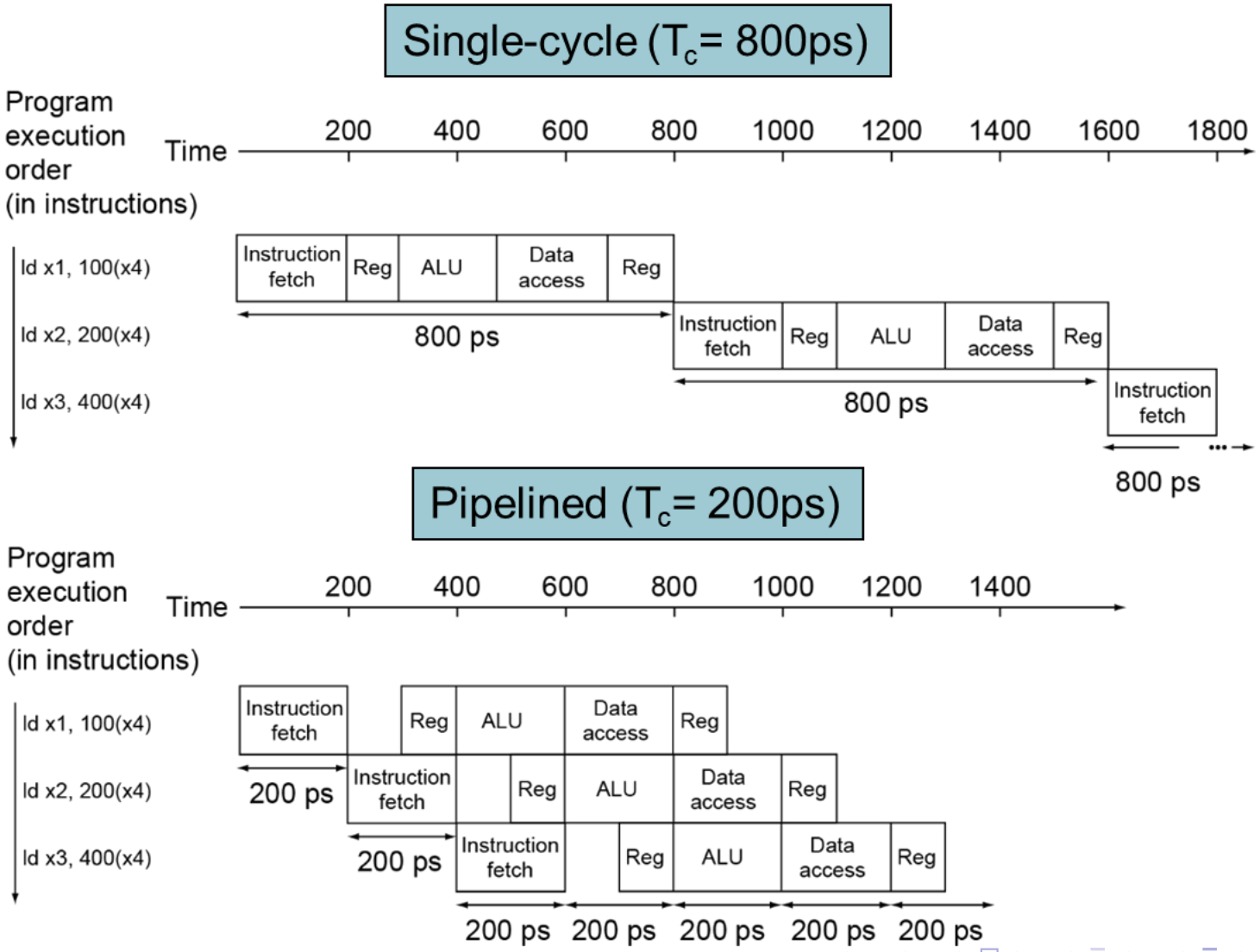
4.5 流水线概述

- 实例：单周期实现与流水线性能，作如下假设：
  - 寄存器读/写耗时100ps
  - 其它阶段耗时200ps
  - 考虑指令：ld、sd、R-type（add、sub、and、or）、beq
  - 每条指令的耗时情况如下所示：

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

4.5 流水线概述

- 实例：单周期实现与流水线性能-续



4.5 流水线概述

- 流水线加速情况
- 理想情况：
  - 流水线各阶段完全均衡（耗时相等）
  - 指令执行时间$_{流水线}$ = 指令执行时间$_{非流水线}$ /流水线级数
- 实际情况：
  - 各阶段不完全均衡
  - 加速效果低于理想情况

4.5 流水线概述

- RISC-V：面向流水线的指令系统设计
  - 所有RISC-V指令长度相等，RV64和RV32指令长度均为32bit
    - 取指和译码操作更容易
  - 指令格式种类少，格式规整，源寄存器、目的寄存器位置固定
    - 译码和读寄存器操作可以在一个阶段内完成
  - 存储器访问只通过Load、Store指令实现
    - 在执行（EX）阶段进行存储器地址的计算
    - 在访存（MEM）阶段进行存储器访问

4.5 流水线概述

- 冲突/冒险：Hazard
  - 在下一个时钟周期无法执行下一条指令的情况，称为冒险
  - 分为三类
    - 结构冒险
    - 数据冒险
    - 控制冒险

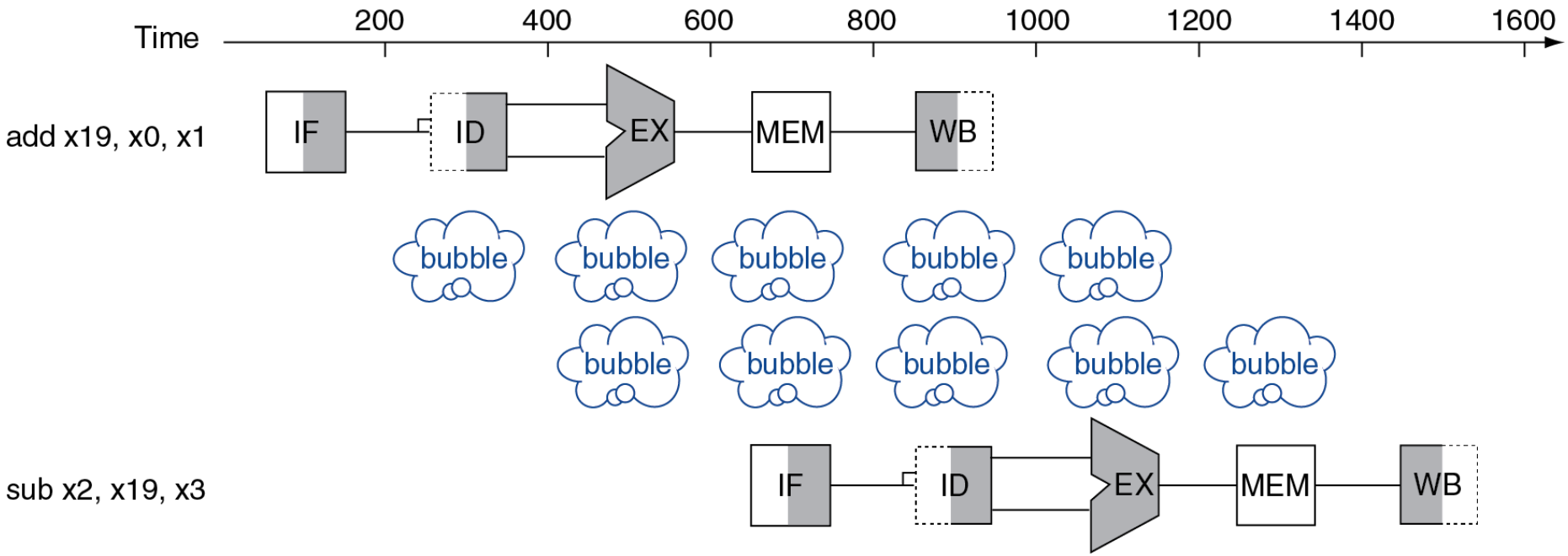
  - 结构冒险：Structure hazard
    - 因缺乏硬件支持，而导致指令不能在预定的时钟周期内执行的情况
  - 数据冒险：Data hazard
    - 因无法提供指令执行所需数据而导致指令不能在预期的时钟周期内执行
  - 控制冒险：Control hazard
    - 也称分支冒险，由于取到的指令并不是所需的，或者指令地址的流向不是流水线所预期的，导致正确的指令无法在正确的时钟周期内执行

4.5 流水线概述

- 结构冒险
  - eg：指令和数据使用同一个存储器，Load/Store指令访问Mem时便会与取指操作相冲突，造成流水线发生结构冒险
  - 解决办法：RISC-V是面向流水线设计的，采用指令存储器和数据存储器相分离的设计，因此不会发生结构冒险
  - 结论：RISC-V采用哈佛体系结构，不用考虑结构冒险

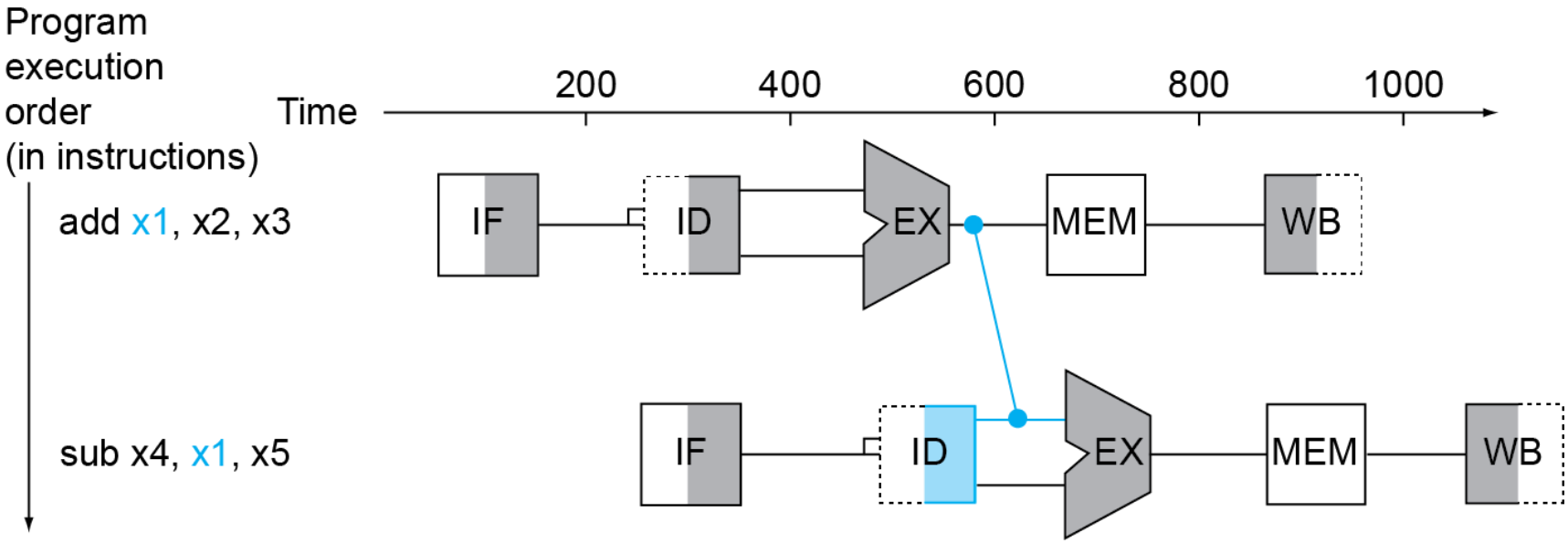
4.5 流水线概述

- 数据冒险
  - add x19, x0, x1
  - sub x2, x19, x3



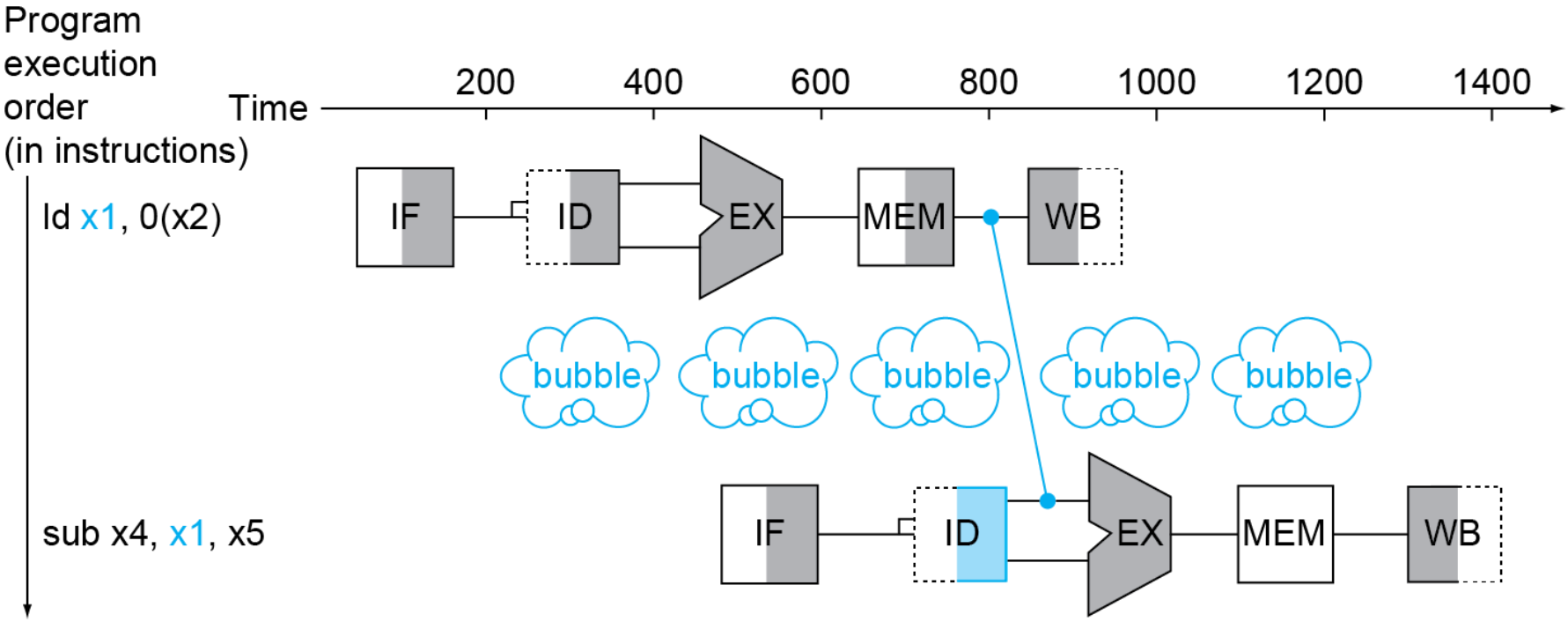
4.5 流水线概述

- 数据冒险解决办法：前递（forwarding）或旁路（bypass）
- 前递（forwarding）或旁路（bypass）
  - 一种解决数据冒险的方法，提前从内部缓冲中取到数据，而不是等到数据到达程序员可见的寄存器或存储器再去使用
  - 需要在数据通路中增加额外连接电路
  - add x1, x2, x3
  - sub x4, x1, x5



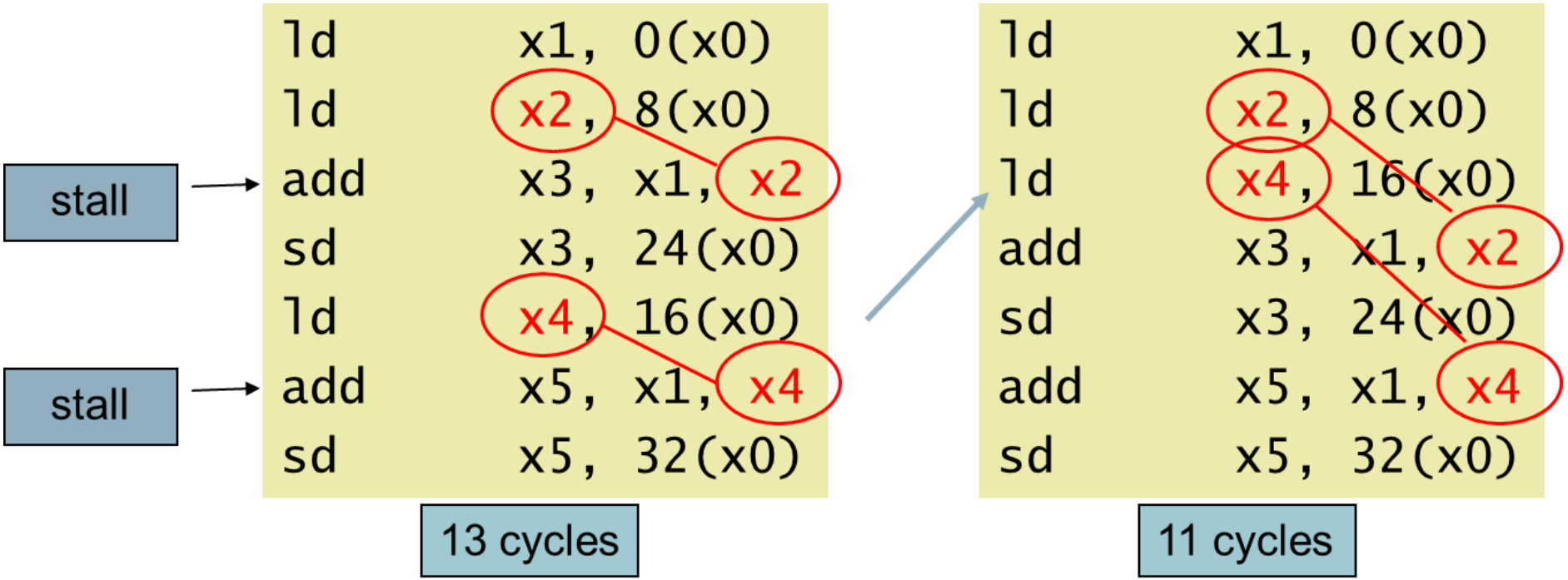
4.5 流水线概述

- 数据冒险解决办法：流水线停顿（pipeline stall），俗称气泡（bubble）
  - 前递效果很好，但不能解决所有的数据冒险
  - 载入-使用型数据冒险（load-use data hazard）不能及时将数据前递给相应部件，需要将流水线停顿，以等待数据



4.5 流水线概述

- 数据冒险解决办法：代码重新排序
  - 避免在Load指令后立即使用其结果
  - 利用编译器，在不影响软件功能的前提下，调整指令顺序
  - eg：C代码 a = b + e; c = b + f;



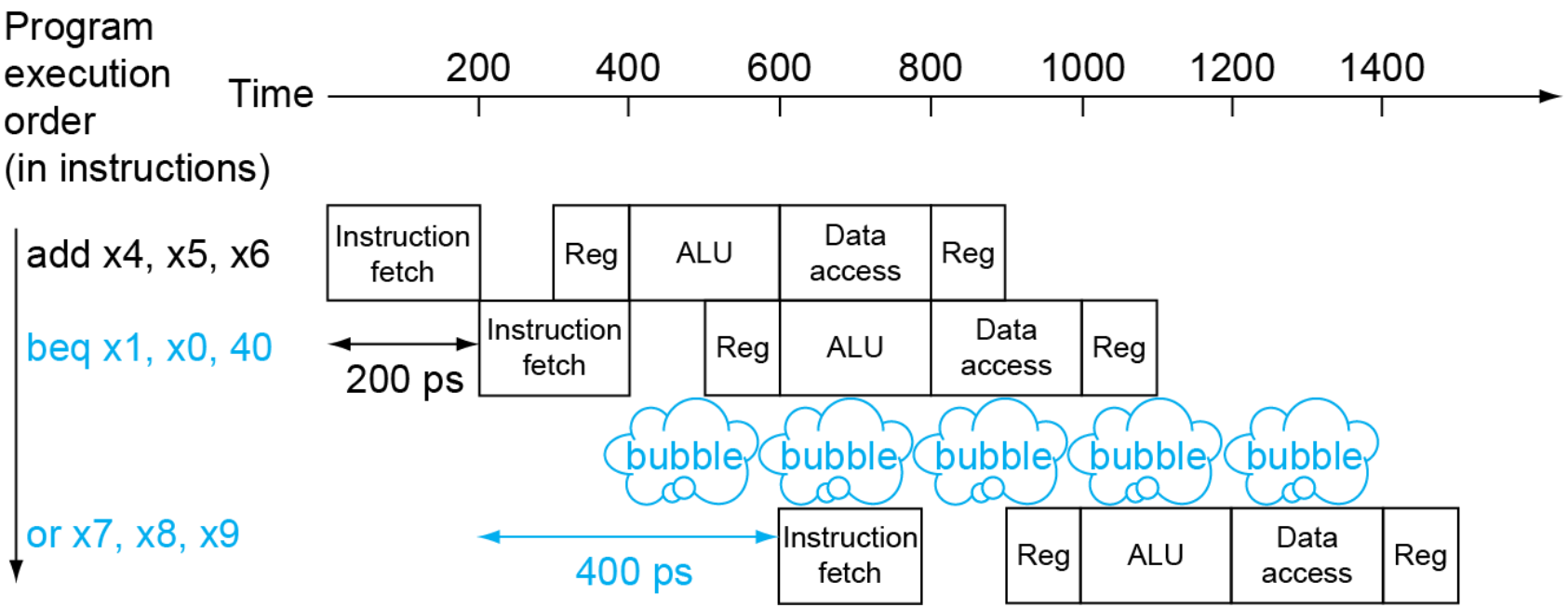
4.5 流水线概述

- 控制冒险，也称分支冒险
- 分支指令的结果决定指令执行流向
  - 下一条执行的指令取决于分支指令的结果
  - 流水线无法一直取到下一条要执行的正确指令
    - 取指时，分支指令还处于译码阶段
- 在RISC-V流水线中，为解决控制冒险
  - 提前比较寄存器，并计算目标地址
  - 添加硬件，在译码阶段完成

  - 方案1：阻塞
  - 方案2：预测

4.5 流水线概述

- 阻塞流水线
  - 等待，直至分支指令计算出下一条要执行的指令
  - 优点：有效，实现简单
  - 缺点：影响性能



4.5 流水线概述

- 分支预测
  - 一种解决分支冒险的方法。它预测分支的结果并沿预测方向执行，而不是等分支结果确定后才开始执行
  - 较长的流水线通常无法在较早的阶段（如ID阶段）确定分支方向
  - 每条分支指令都阻塞，将导致计算机性能严重降低
  - 解决方法：预测分支方向
    - 仅预测失败时导致阻塞

- 在RISC-V流水线CPU设计中（仅供参考）
  - 可以约定始终预测分支不发生
  - 可以直接取分支指令的下一条指令进入流水线

4.5 流水线概述

- 几种实际的分支预测方案

- 静态分支预测
  - 基于始终不变的分支预测行为
  - eg：循环、if条件语句
  - 预测backward分支发生
  - 预测forward分支不发生

- 动态分支预测
  - 硬件测量分支发生的实际情况
    - eg：记录最近发生的分支情况
  - 假定后续行为会延续历史记录

4.5 流水线概述

- 总结

- 流水线通过增加指令吞吐量提高性能
  - 并行执行多条指令
  - 每条指令具有相同的延时
- 冒险分类
  - 结构冒险
  - 数据冒险
  - 控制冒险
- 指令集设计会影响到流水线实现的复杂性
- 流水线对程序员不可见

4.5 流水线概述

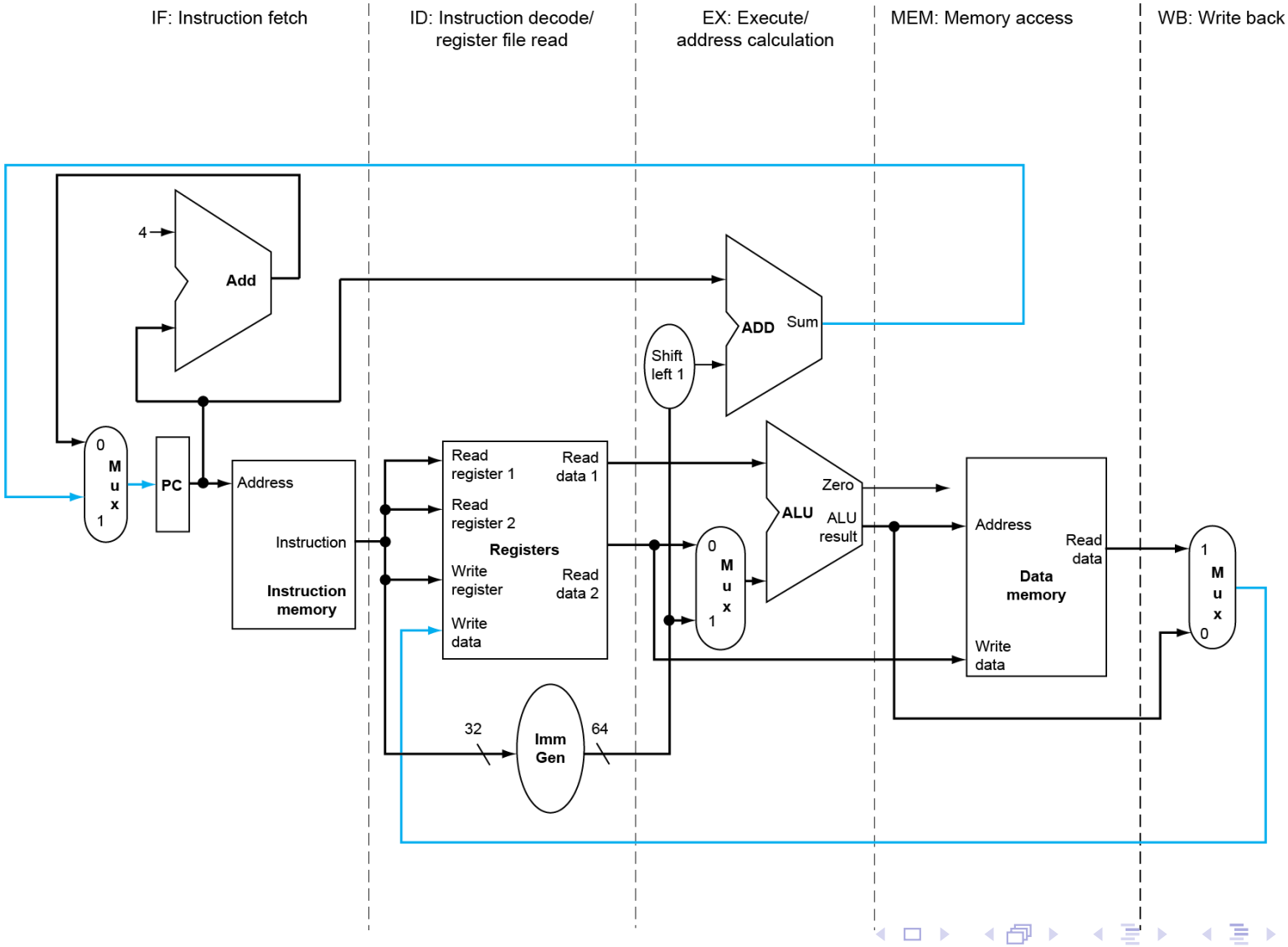
- 思考题：
- 与单周期CPU设计相比，多周期设计和流水线设计是如何提高CPU性能的

自我检测 对于下面的每个代码序列，说明它是否必须停顿，或者只使用前递就可以避免停顿，或者既不需要停顿也不需要前递就可以执行。

序列 1	序列2	序列3
ld x10, 0(x10)	add x11, x10, x10	addi x11, x10, 1
add x11, x10, x10	addi x12, x10, 5	addi x12, x10, 2
	addi x14, x11, 5	addi x13, x10, 3
		addi x14, x10, 4
		addi x15, x10, 5

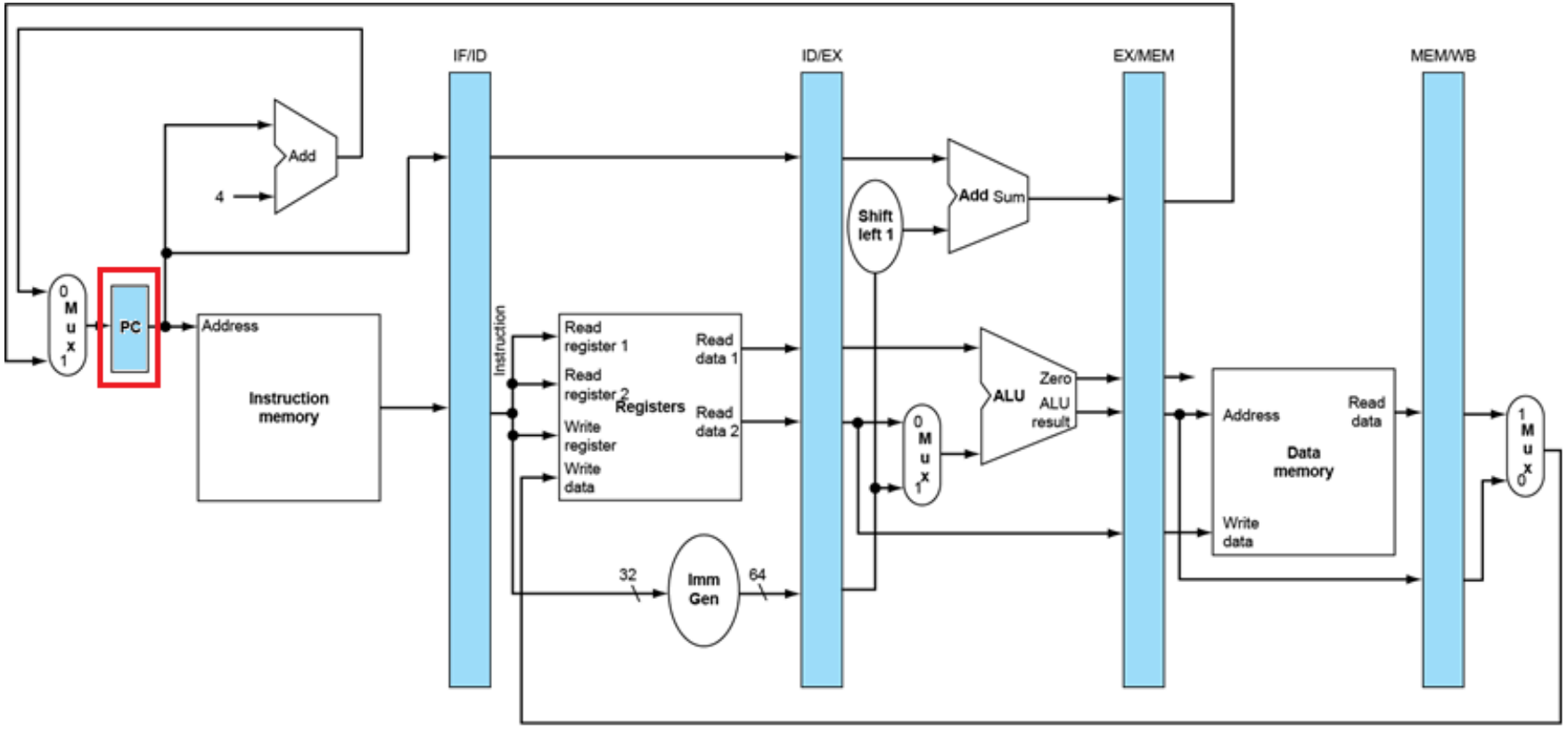
4.6 流水线数据通路和控制

- 从右向左的信号将导致冒险



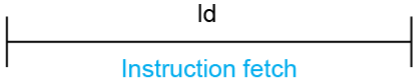
4.6 流水线数据通路和控制

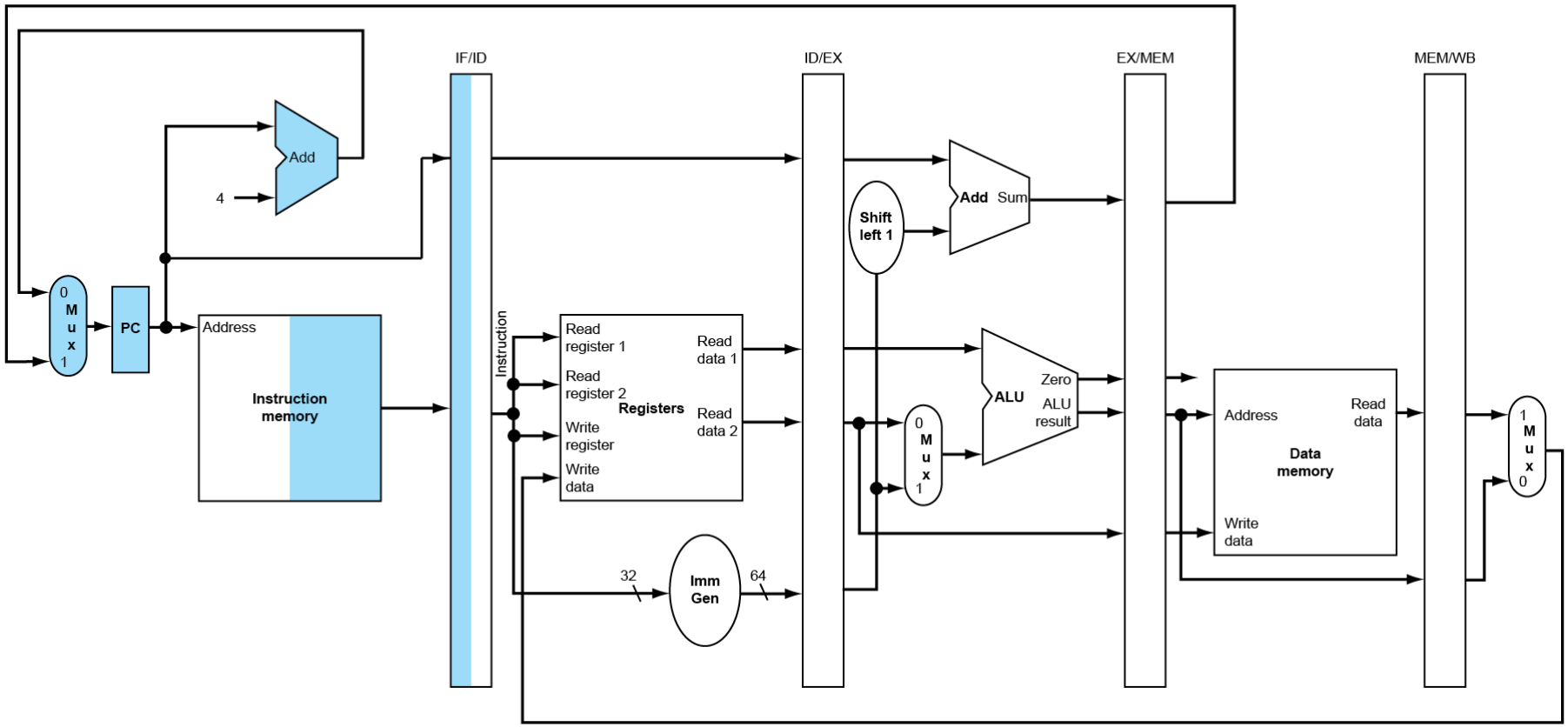
- 流水线寄存器
  - 在不同的阶段之间需要插入寄存器，用来保存前一阶段的数据信息
  - 五级流水线 vs 五级寄存器
  - 说明：PC寄存器也可以认为是一级寄存器（教材中没有标蓝）



4.6 流水线数据通路和控制

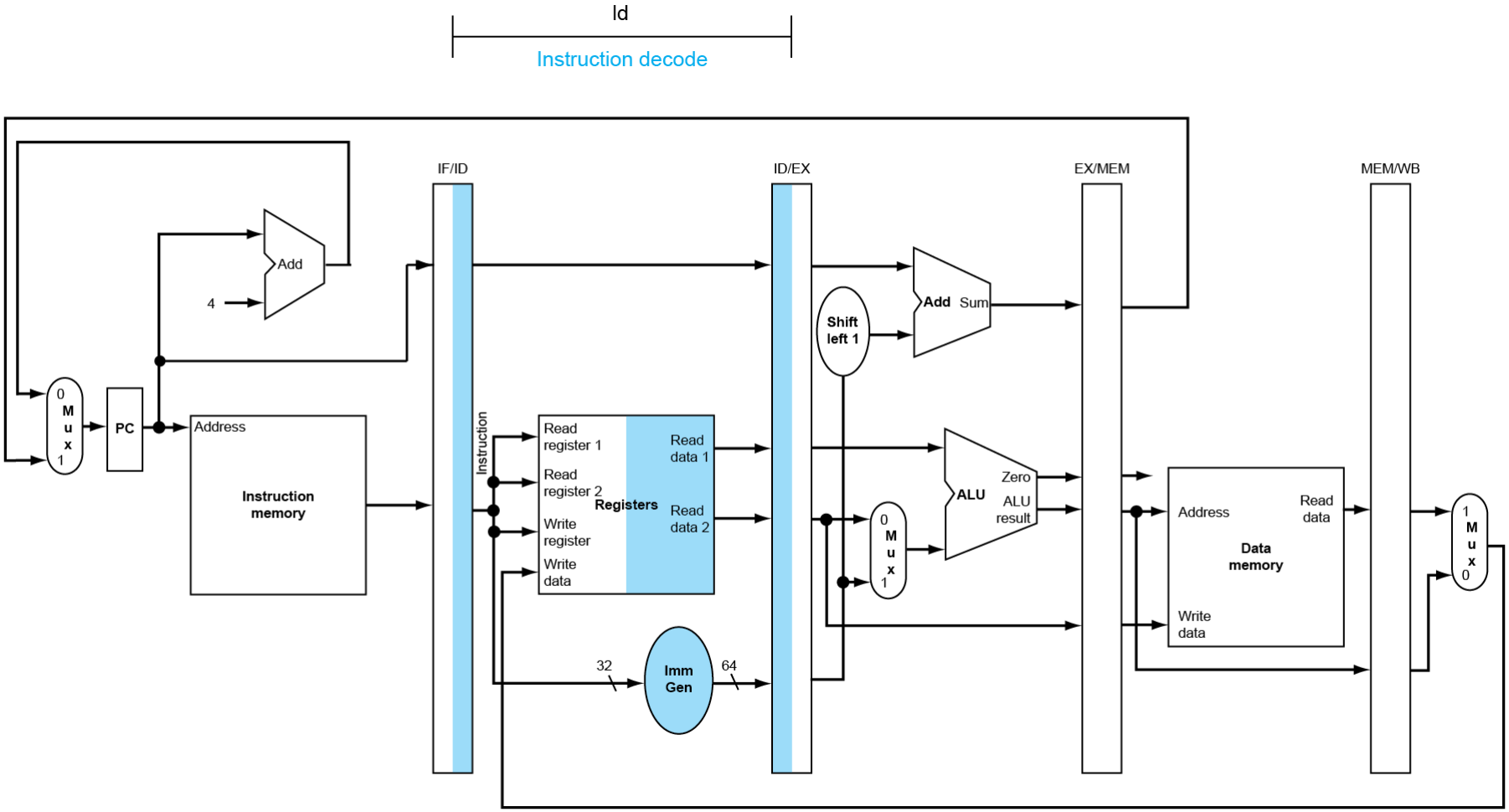
- 流水线操作
  - Load、Store指令，IF阶段





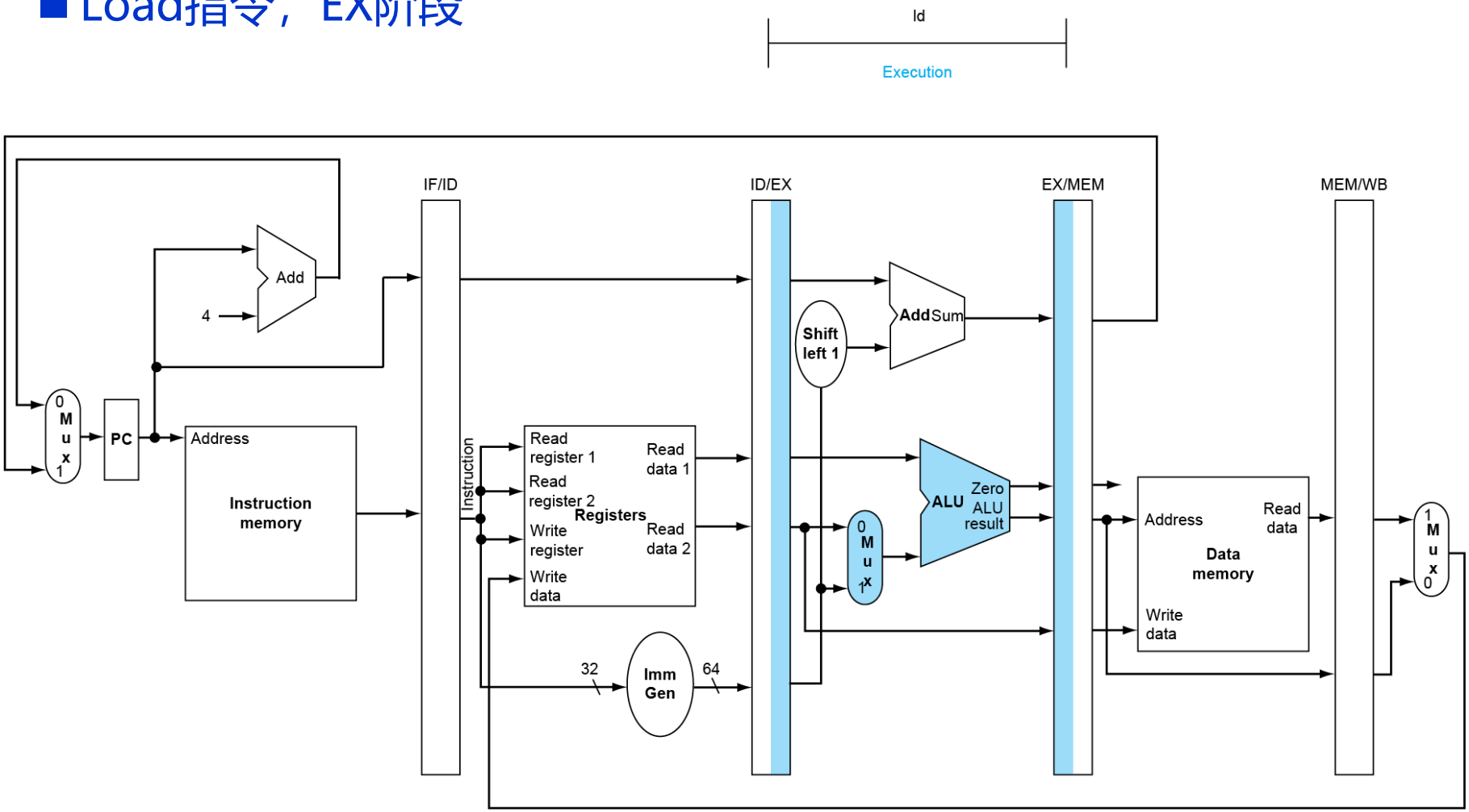
4.6 流水线数据通路和控制

- 流水线操作
  - Load、Store指令，ID阶段



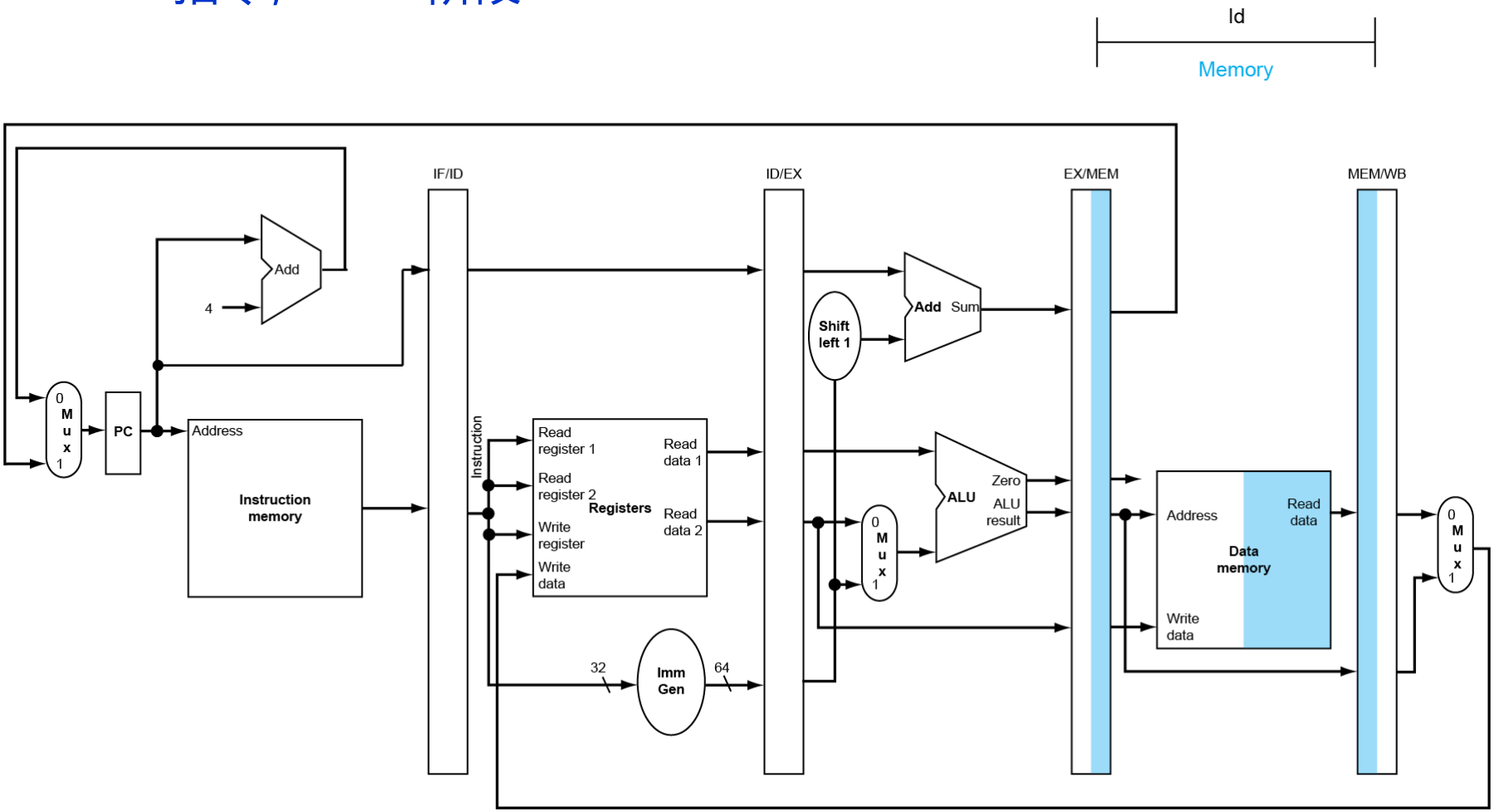
4.6 流水线数据通路和控制

- 流水线操作
  - Load指令，EX阶段



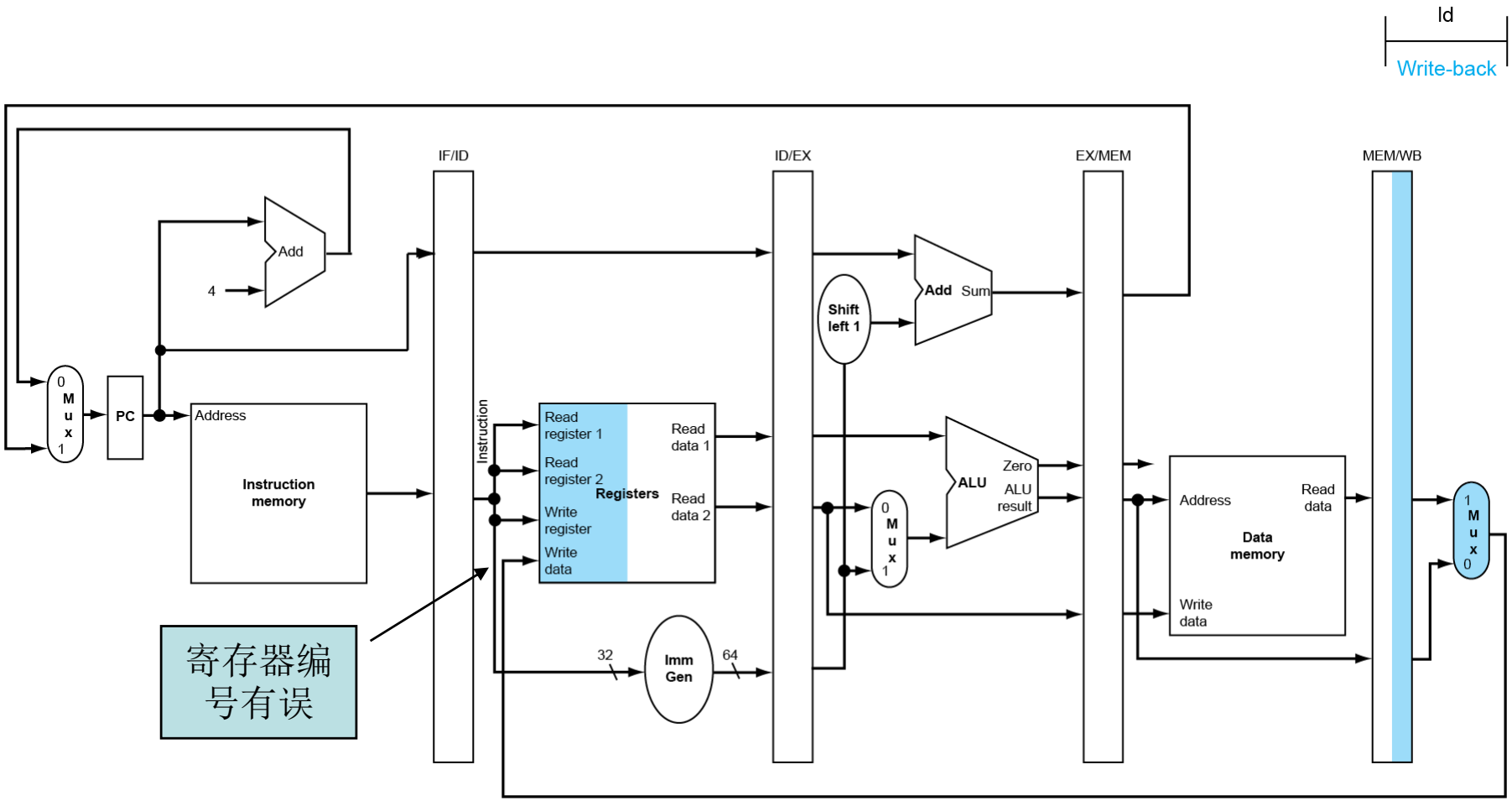
4.6 流水线数据通路和控制

- 流水线操作
  - Load指令，MEM阶段



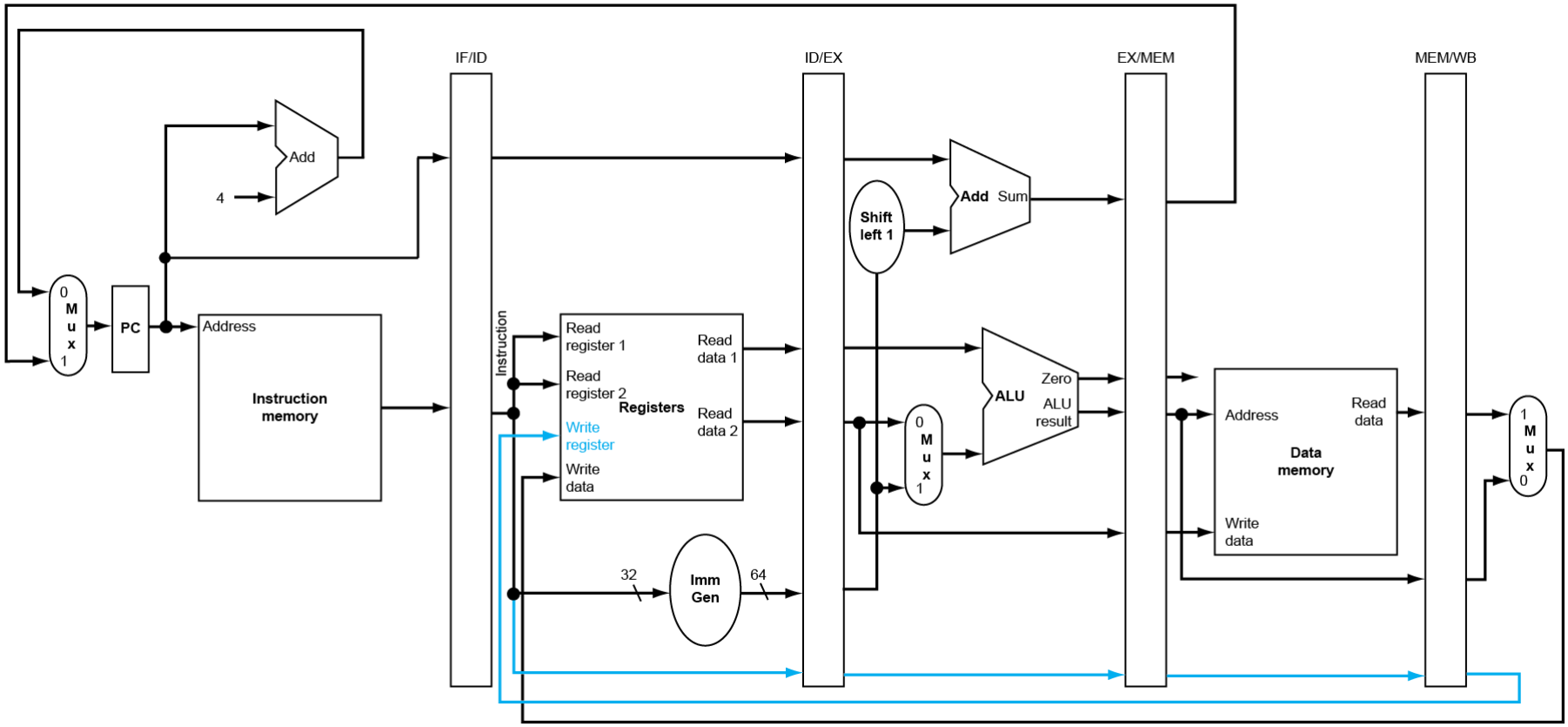
4.6 流水线数据通路和控制

- 流水线操作
  - Load指令，WB阶段



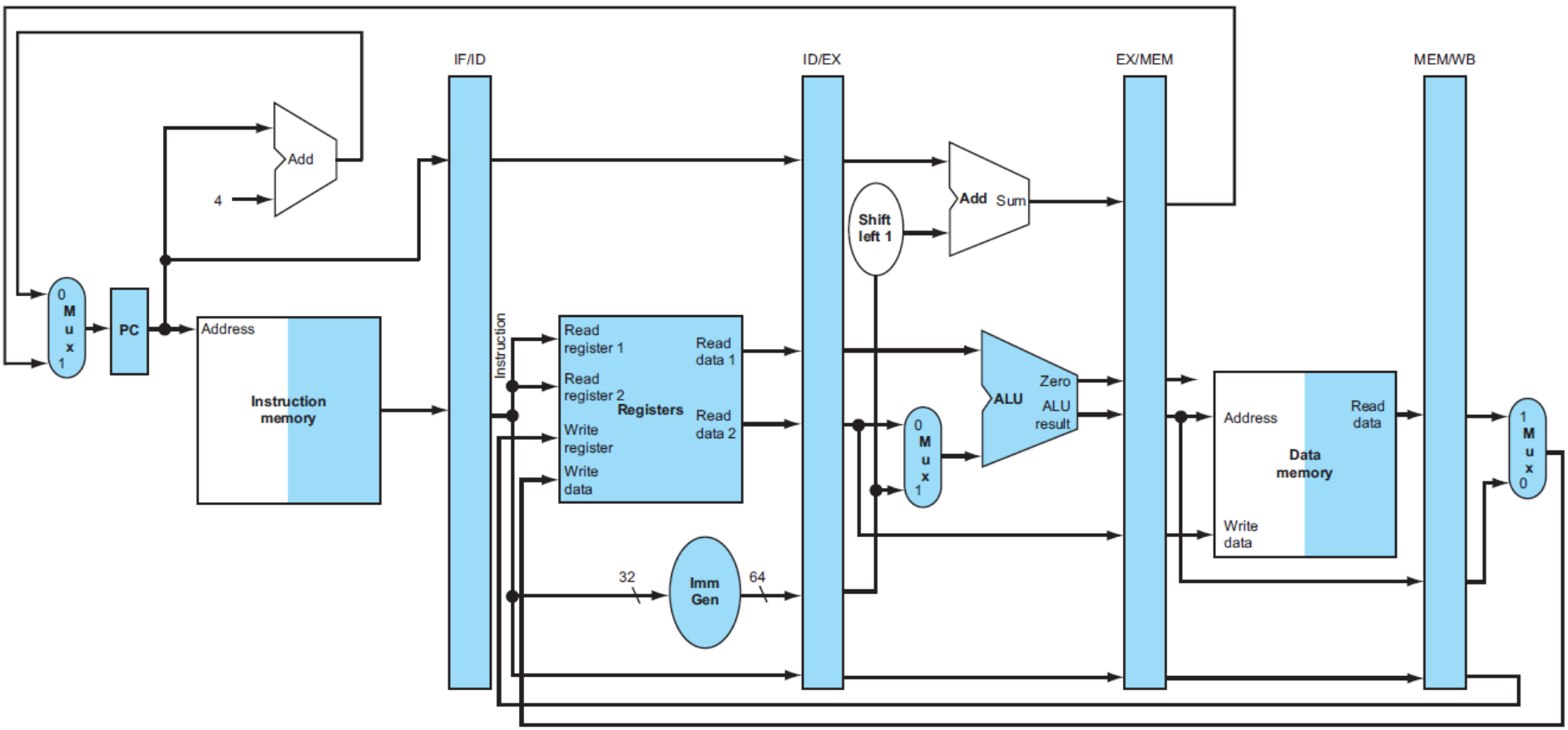
4.6 流水线数据通路和控制

- 流水线操作
  - Load指令，WB阶段_改进



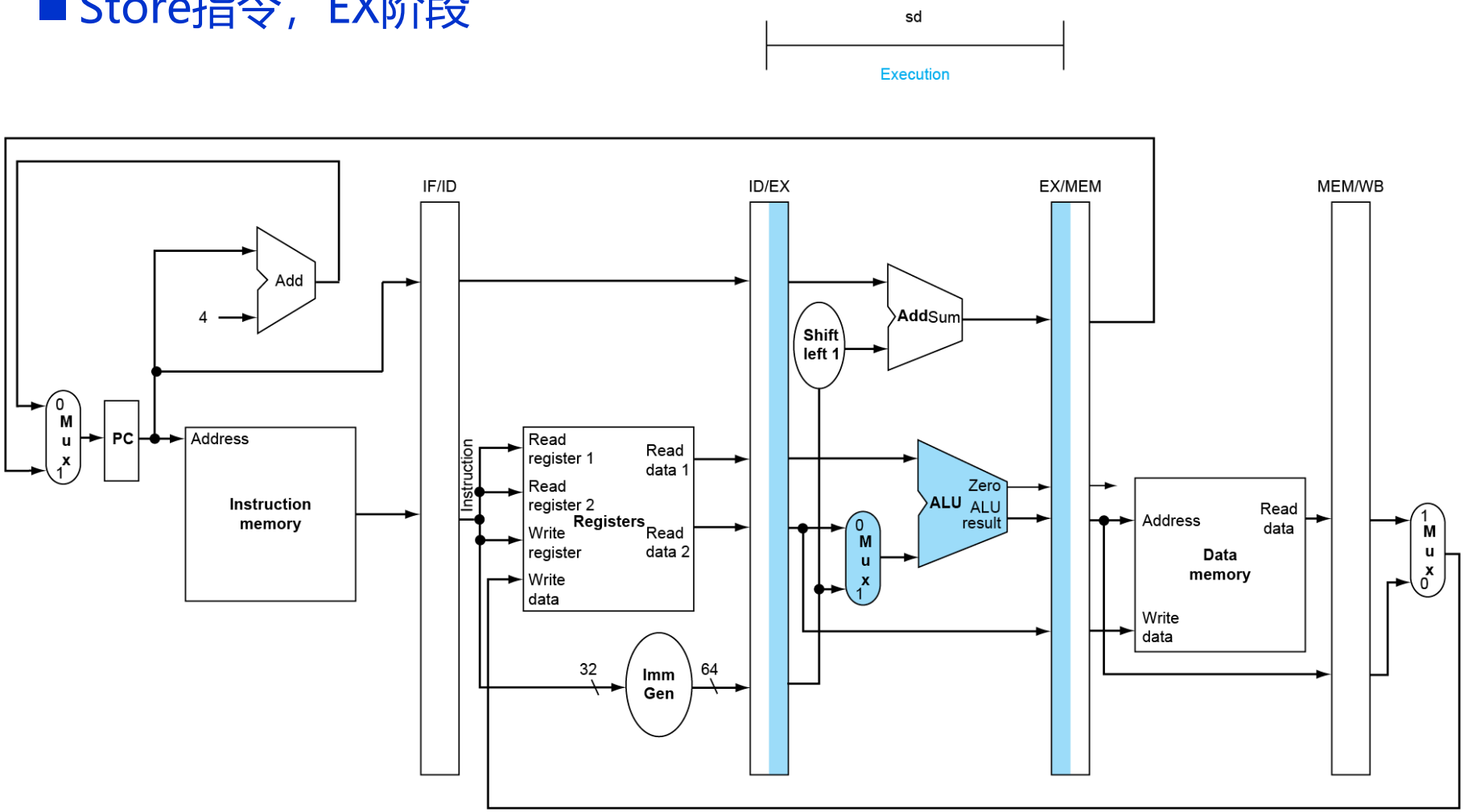
4.6 流水线数据通路和控制

- 流水线操作
  - Load指令，全部阶段



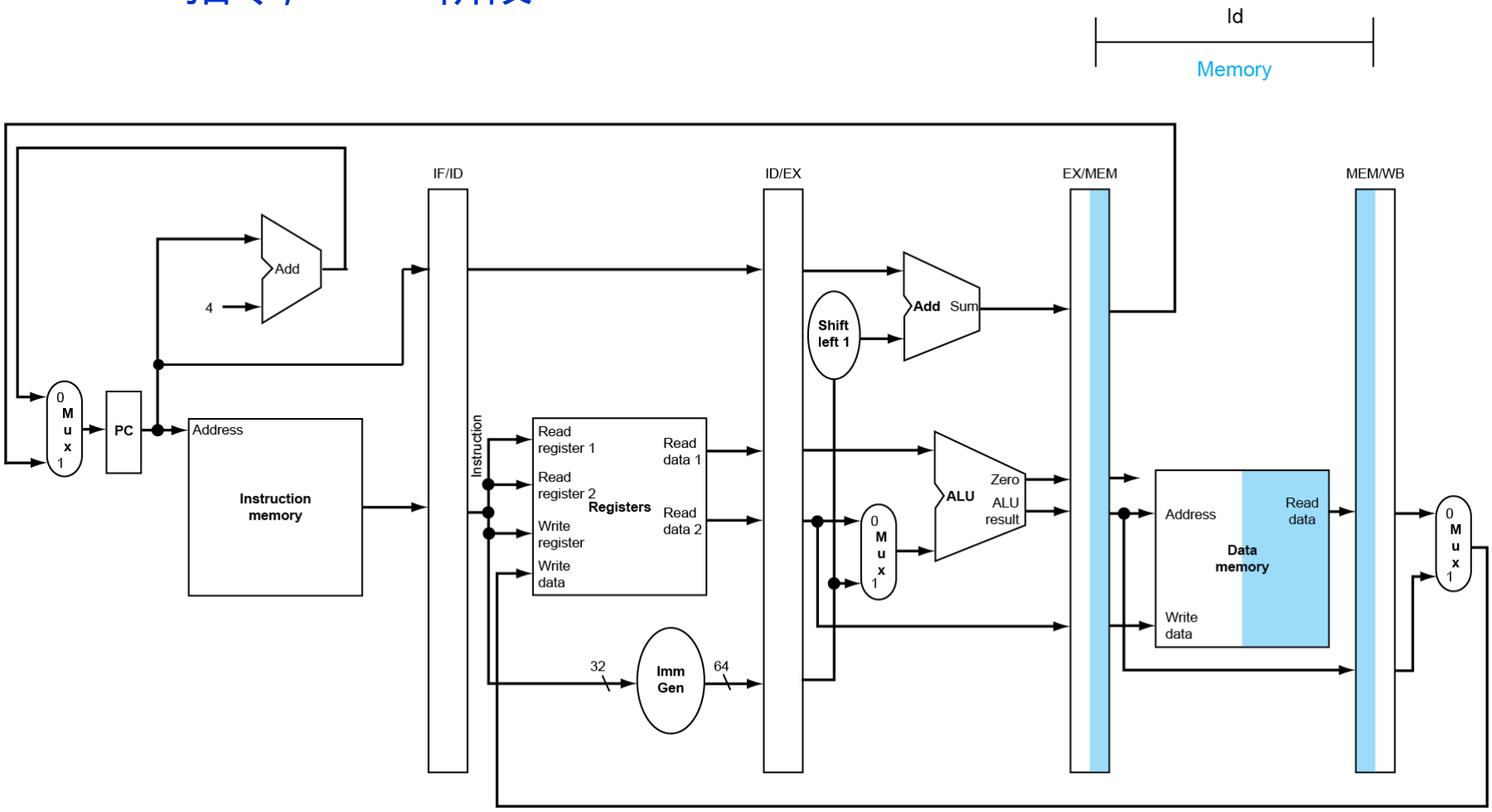
4.6 流水线数据通路和控制

- 流水线操作
  - Store指令，EX阶段



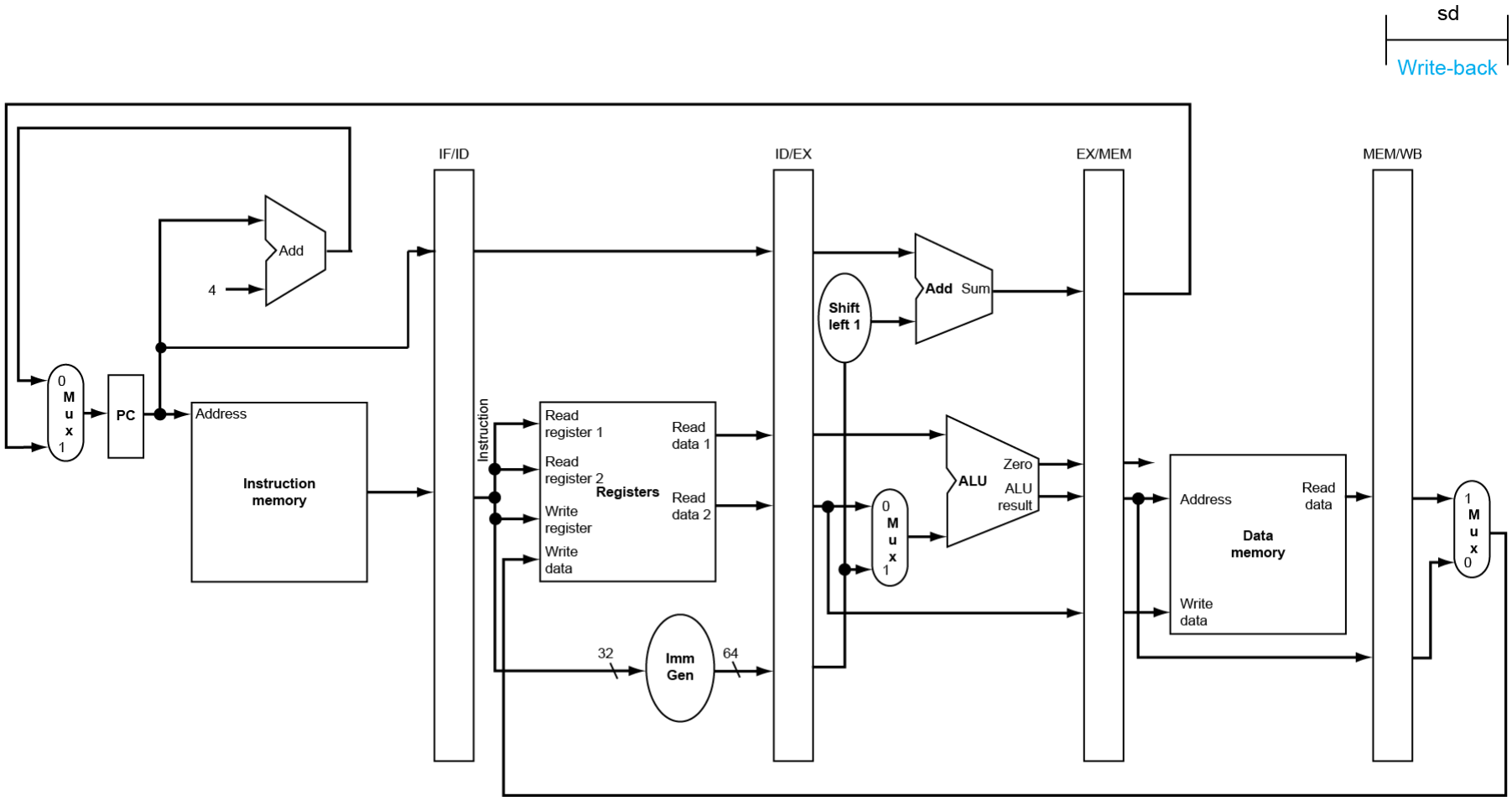
4.6 流水线数据通路和控制

- 流水线操作
  - Store指令，MEM阶段



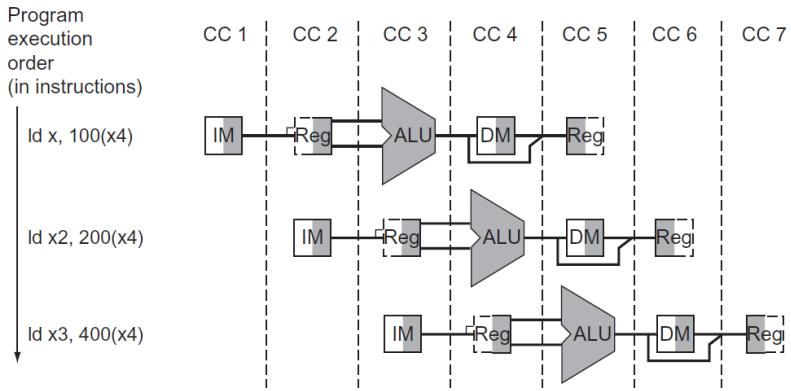
4.6 流水线数据通路和控制

- 流水线操作
  - Store指令，WB阶段

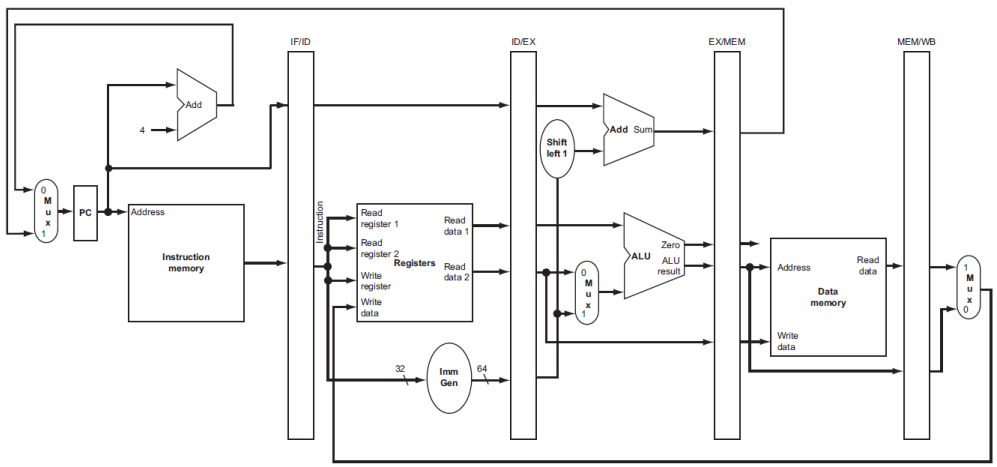


4.6 流水线数据通路和控制

- 流水线的图形化表示
  - 多时钟周期流水线图



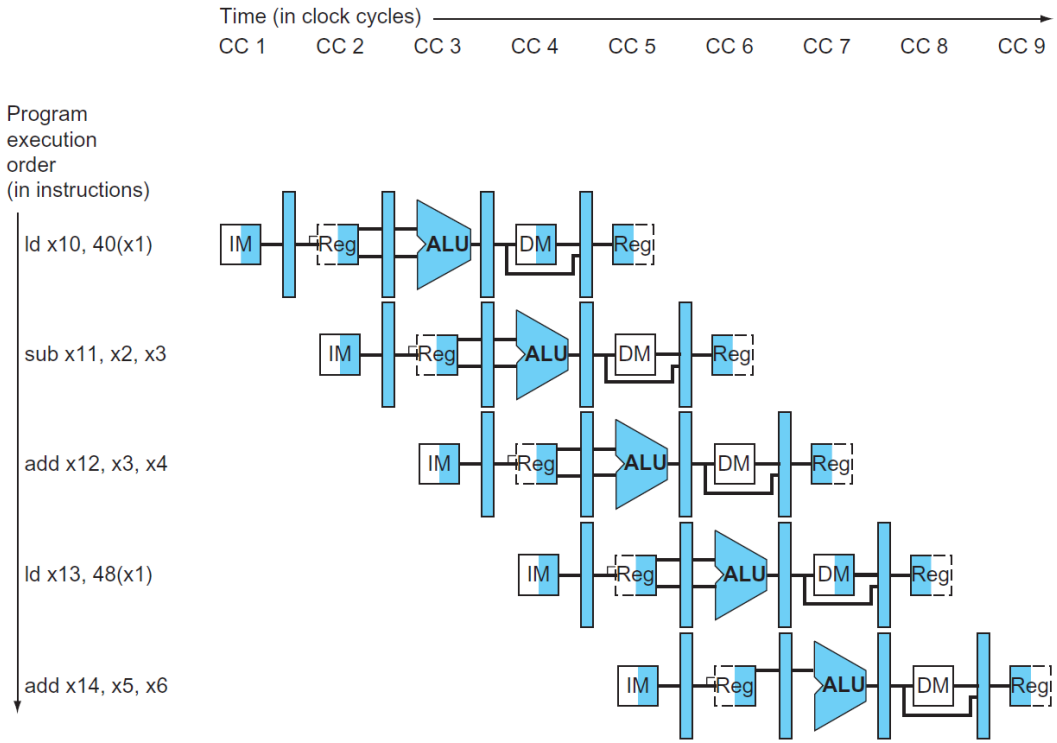
  - 单时钟周期流水线图



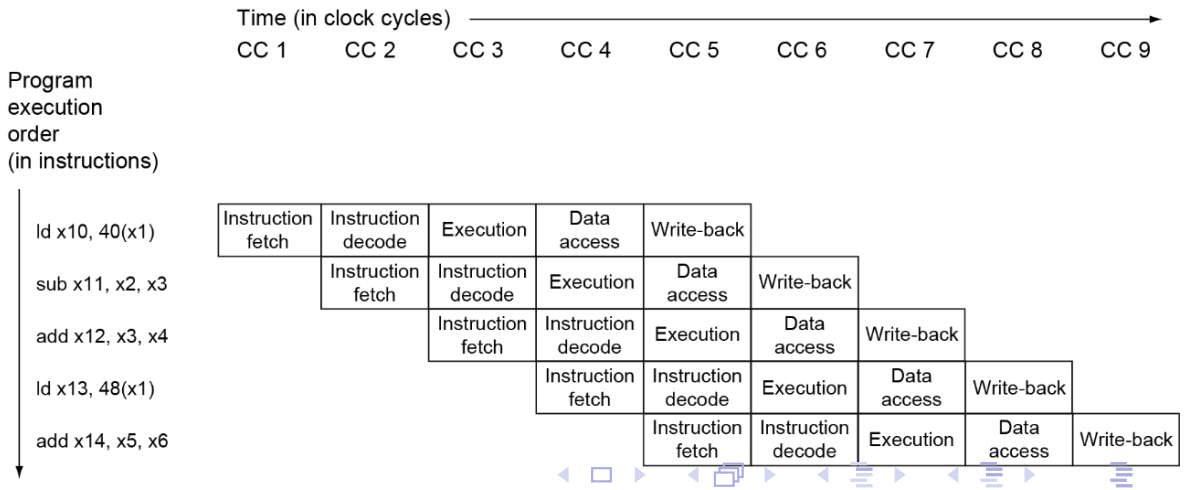
4.6 流水线数据通路和控制

- 流水线的图形化表示
  - 多周期流水线图

```
ld   x10,  40(x1)
sub x11,  x2, x3
add x12,  x3, x4
ld    x13,  48(x1)
add x14,  x5, x6
```



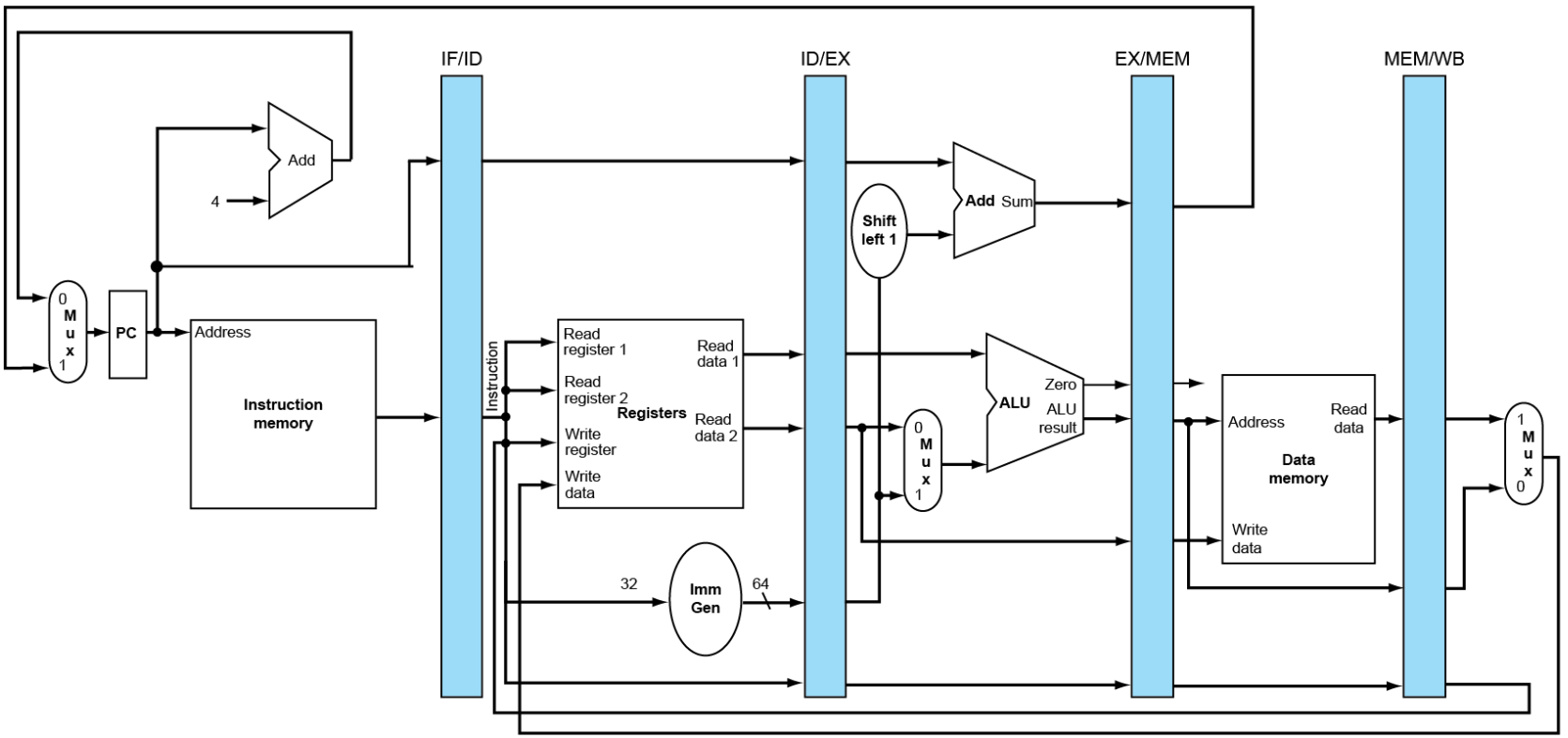
- 传统画法



4.6 流水线数据通路和控制

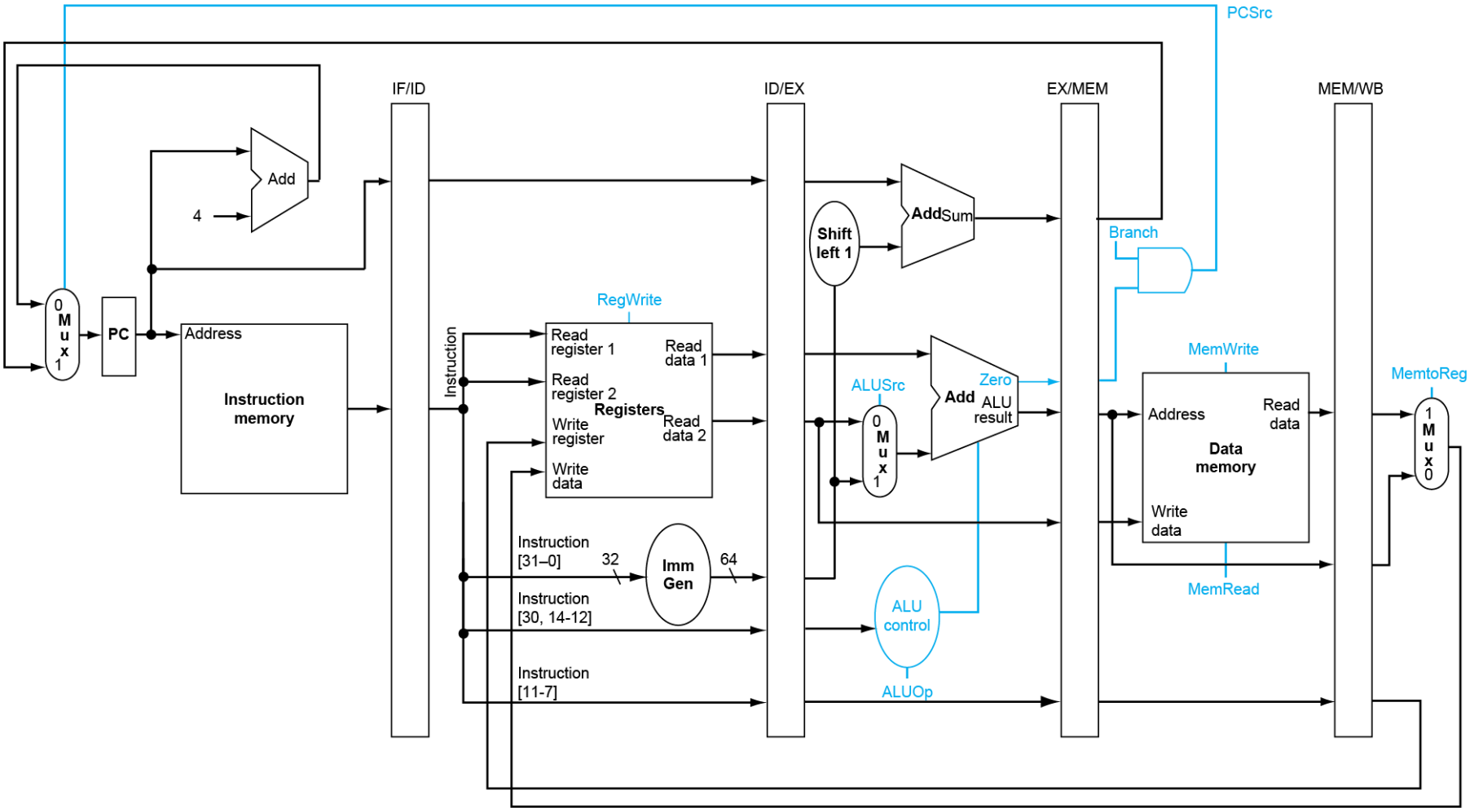
- 单时钟周期流水线图
  - 第五个时钟周期

add x14, x5, x6	ld x13, 48(x1)	add x12, x3, x4	sub x11, x2, x3	ld x10, 40(x1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



4.6 流水线数据通路和控制

- 流水线控制_简单版本



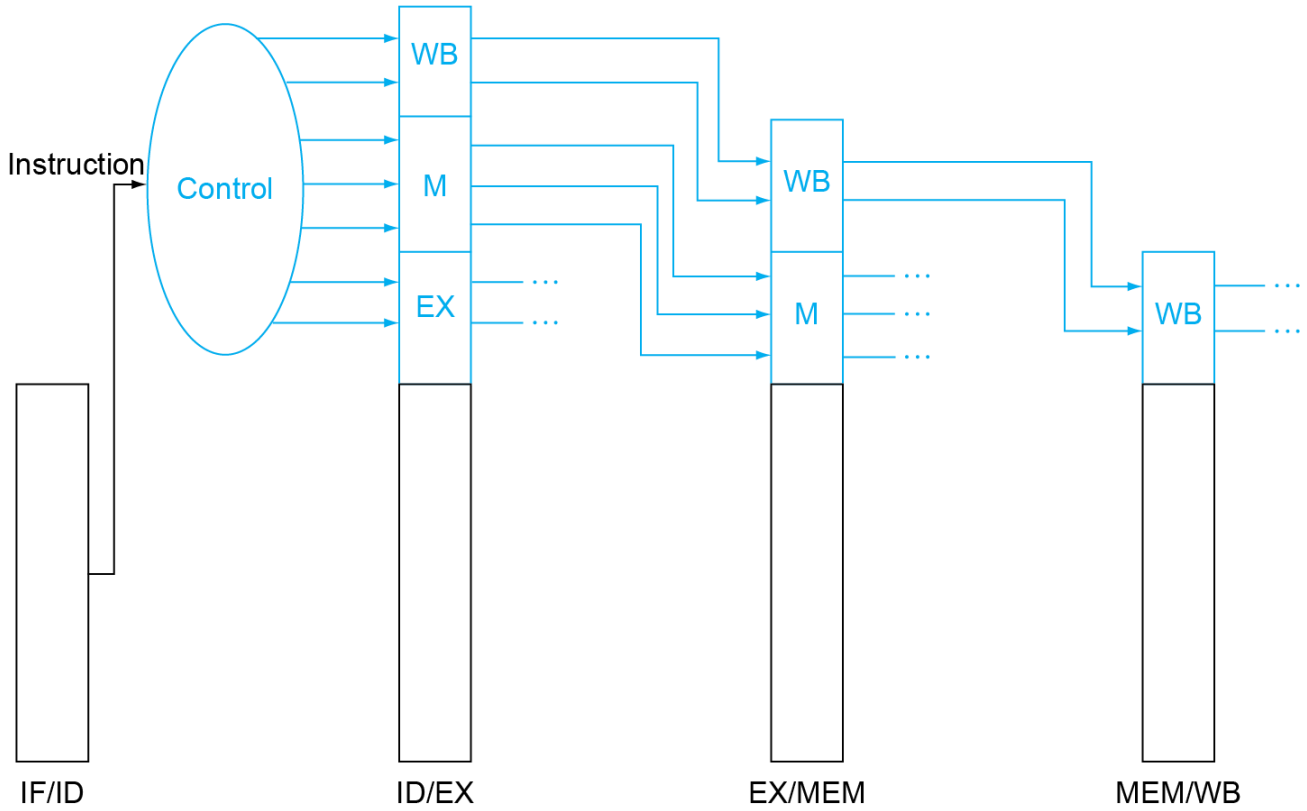
4.6 流水线数据通路和控制

- 流水线控制_简单版本
  - 取指：没有需要特别关注的控制信号
  - 指令译码/读寄存器堆：没有需要特别关注的控制信号
  - 执行/地址计算：ALUOp、ALUSrc、算出分支目标地址
  - 存储器访问：Branch、MemRead、MemWrite、产生分支控制信号
  - 写回：MemtoReg、RegWrite

指令	执行/地址计算阶段控制线		存储器访问阶段控制线			写回阶段控制线	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

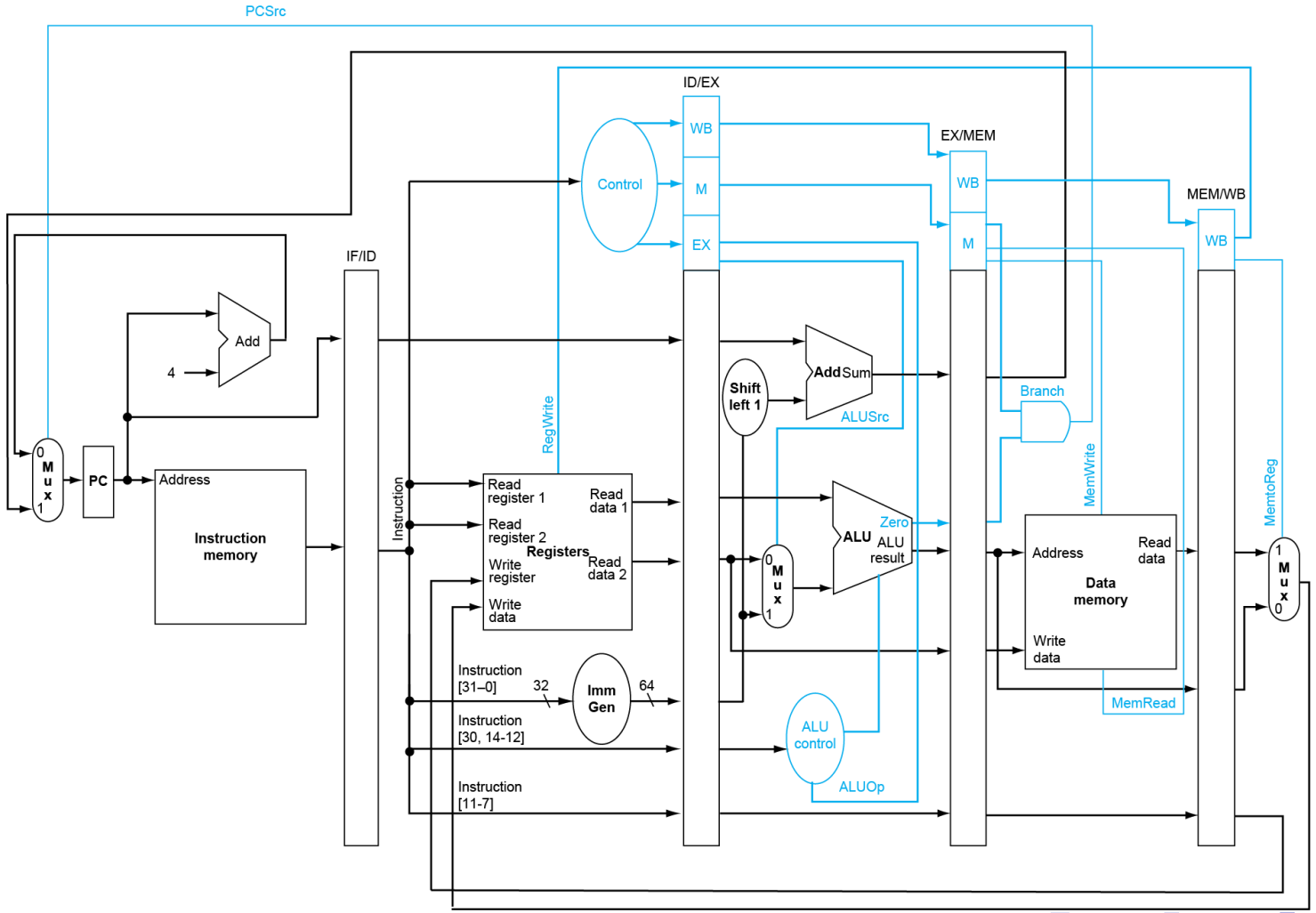
4.6 流水线数据通路和控制

- 流水线控制_简单版本
  - 控制信号由指令译码产生
  - 通过流水线的各级寄存器进行传递



4.6 流水线数据通路和控制

- 流水线控制_简单版本
  - 带控制器的完整数据通路



4.7 数据冒险：前递与停顿

- 数据冒险会导致流水线执行受阻
- eg:

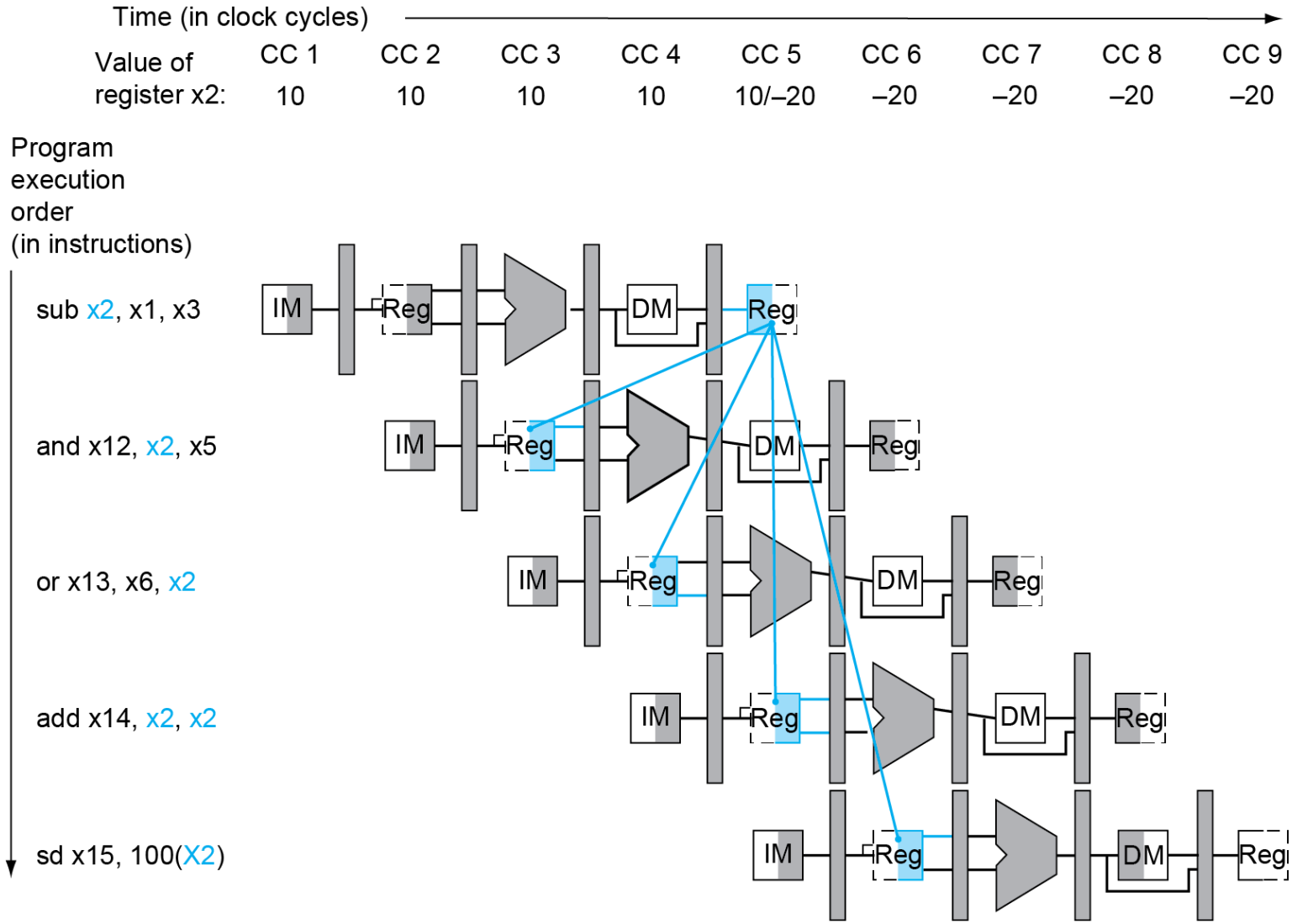
```
sub   x2, x1,x3
and   x12,x2,x5
or    x13,x6,x2
add   x14,x2,x2
sd    x15,100(x2)
```

```
nop
addi x1, x0,1
addi x2, x1,1
nop
nop
add  x3, x1,x2
```

- 可以采用前递技术处理数据冒险
  - Q1：如何检测数据冒险
  - Q2：如何实现数据前递（Forward）

4.7 数据冒险：前递与停顿

- 依赖和前递



4.7 数据冒险：前递与停顿

- 检测数据冒险
  - 将寄存器编码沿流水线逐级传递
  - 寄存器命名规则，分为两部分：流水线寄存器名称.寄存器字段名称
    - eg: ID/EX.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register
  - EX阶段ALU操作所用的寄存器文件编号来自ID/EX
    - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- 以下条件成立时，会发生数据冒险
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

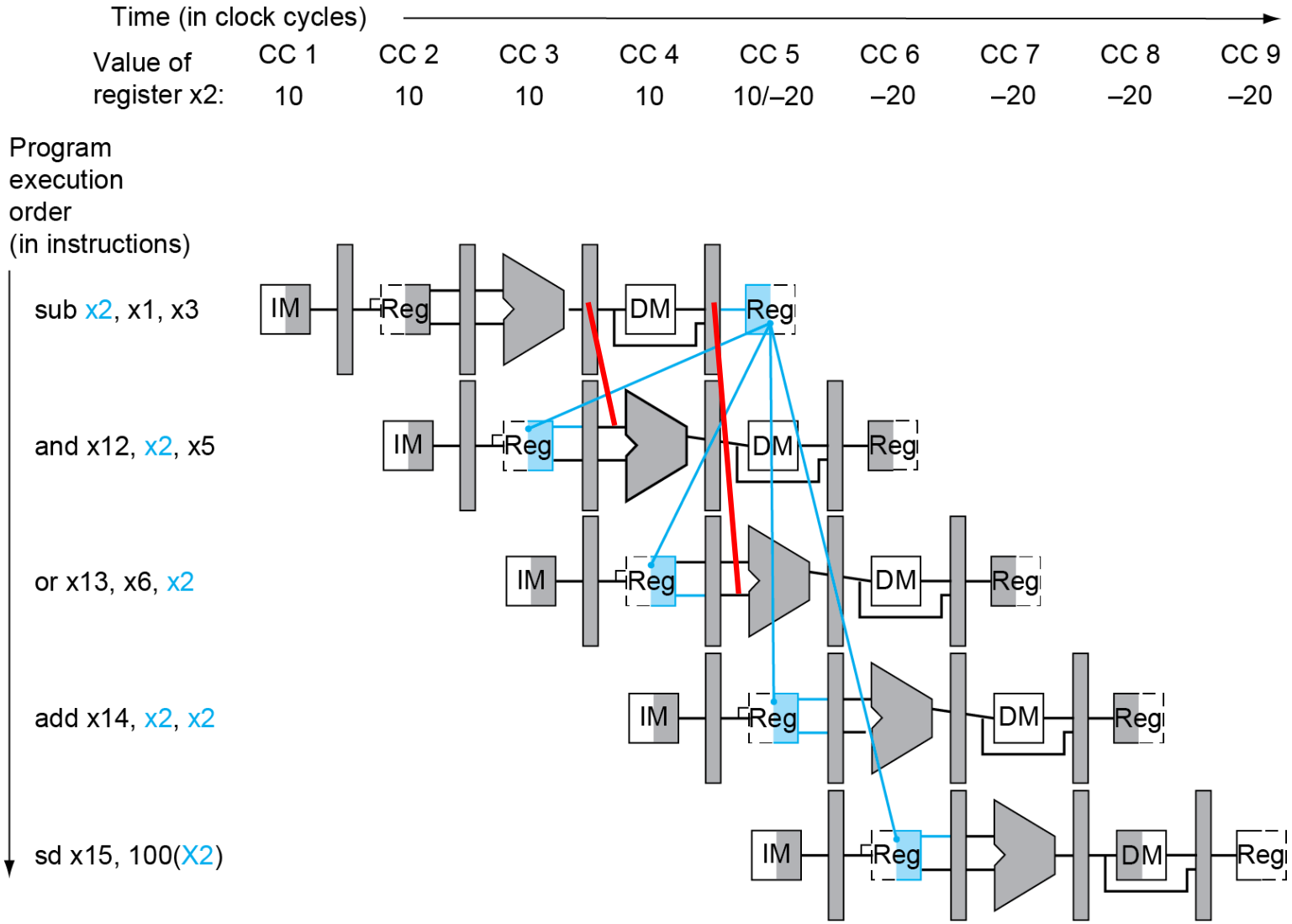
Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

- 同时要求
  - 前递指令有写寄存器操作：EX/MEM.RegWrite, MEM/WB.RegWrite
  - 目标寄存器编号不为0

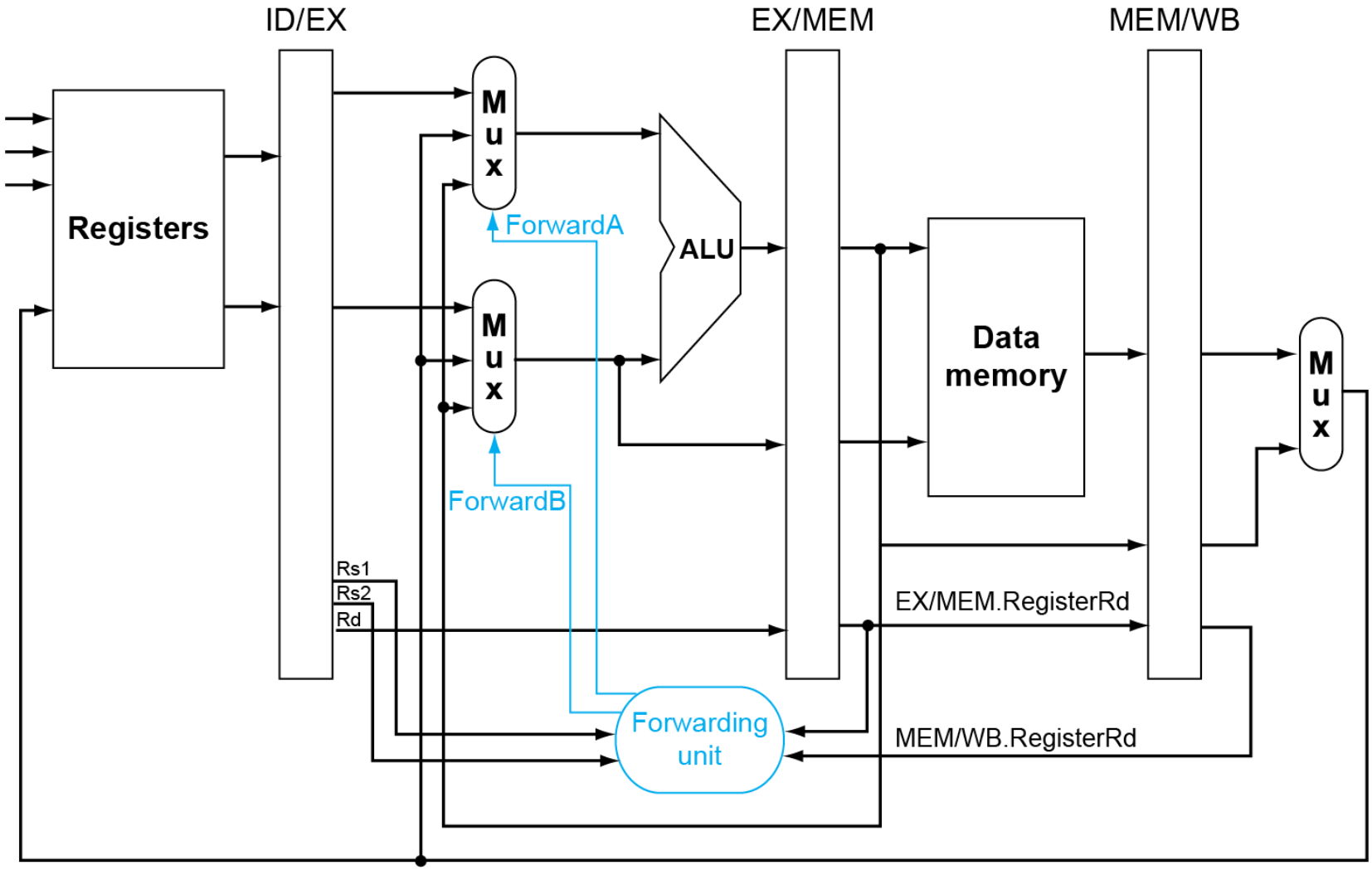
4.7 数据冒险：前递与停顿

- 依赖和前递



4.7 数据冒险：前递与停顿

- 前递相关电路



4.7 数据冒险：前递与停顿

- 前递条件总结
  - EX冒险

```
if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10

if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

  - MEM冒险

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

多选器控制	源	解释
ForwardA = 00	ID/EX	ALU的第一个操作数来自寄存器堆
ForwardA = 10	EX/MEM	ALU的第一个操作数来自上一个ALU计算结果的前递
ForwardA = 01	MEM/WB	ALU的第一个操作数来自数据存储器或者更早的ALU计算结果的前递
ForwardB = 00	ID/EX	ALU的第二个操作数来自寄存器堆
ForwardB = 10	EX/MEM	ALU的第二个操作数来自上一个ALU计算结果的前递
ForwardB = 01	MEM/WB	ALU的第二个操作数来自数据存储器或者更早的ALU计算结果的前递

4.7 数据冒险：前递与停顿

- 思考：
  - WB阶段会不会存在冒险？
  - 在一个时钟周期内对某一寄存器同时进行读写操作会有什么结果？

4.7 数据冒险：前递与停顿

- 一种复杂的数据冒险：双重冒险
  - WB阶段寄存器结果、MEM阶段寄存器结果、EX（ALU）阶段源操作数之间均存在数据冒险
  - eg：

```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

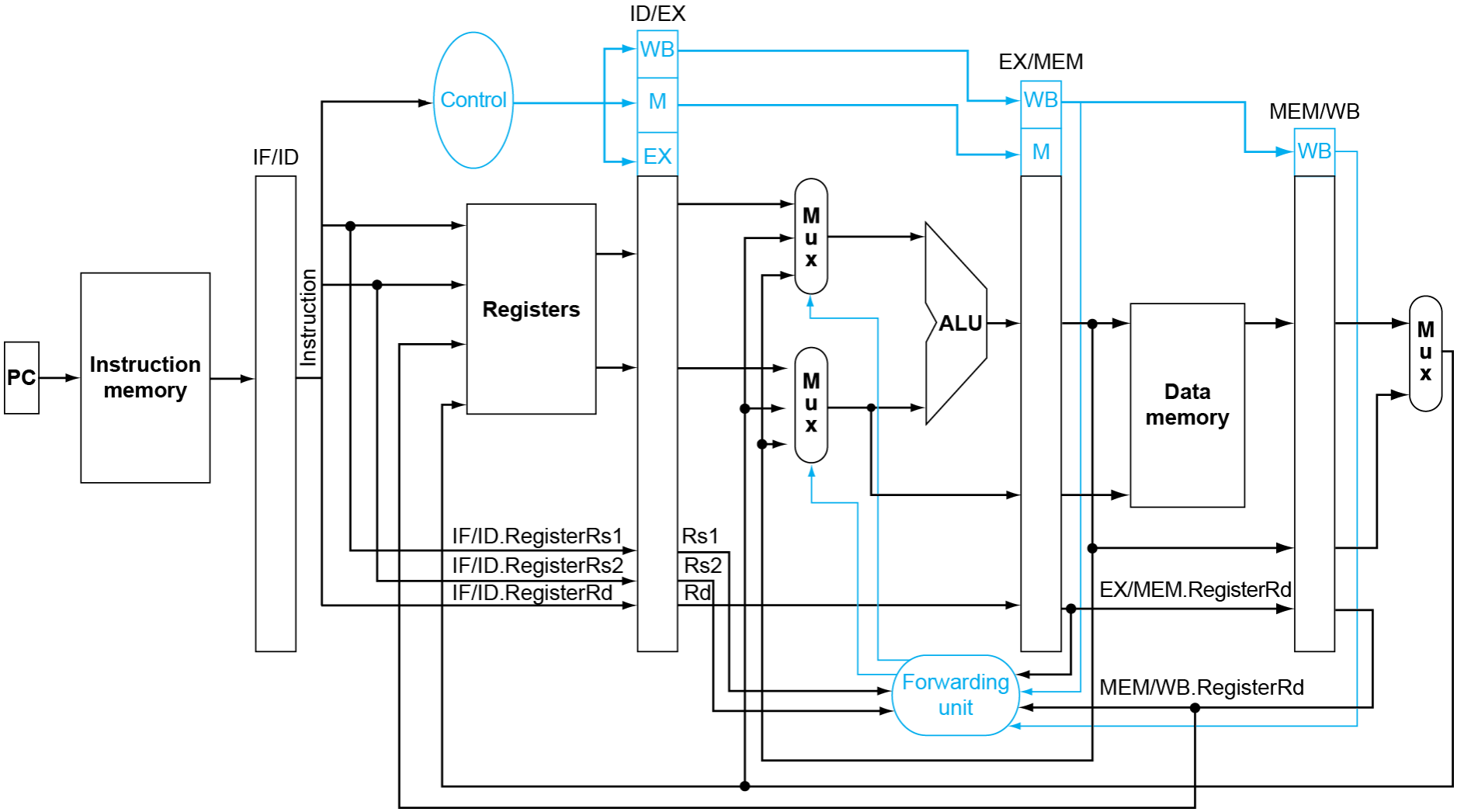
  - 处理方法：使用最新的结果
    - 使用EX/MEM寄存器内的字段
  - MEM/WB阶段的冒险
    - 当EX/MEM阶段的冒险条件为假时才可能有效

4.7 数据冒险：前递与停顿

- MEM冒险控制逻辑修正
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

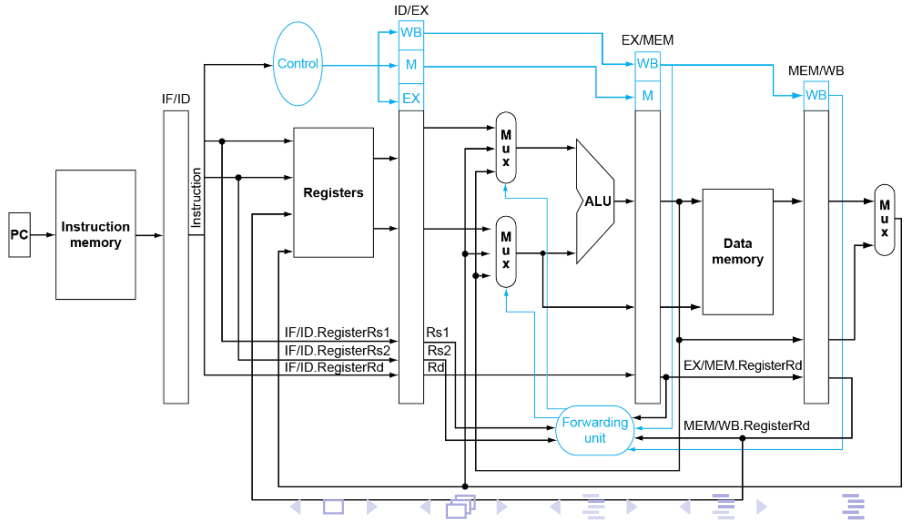
4.7 数据冒险：前递与停顿

- 带有前递功能的数据通路



4.7 数据冒险：前递与停顿

- 加载指令相关的数据冒险（Load-Use Hazard）检测
  - 在ID阶段对指令进行译码时检测
  - ALU运算相关寄存器编号在ID阶段来自
    - IF/ID.RegisterRs1
    - IF/ID.RegisterRs2
- Load-Use冒险发生条件
  - ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRs1))
- Load-Use冒险解决办法
  - 阻塞流水线
  - 插入气泡（bubble）

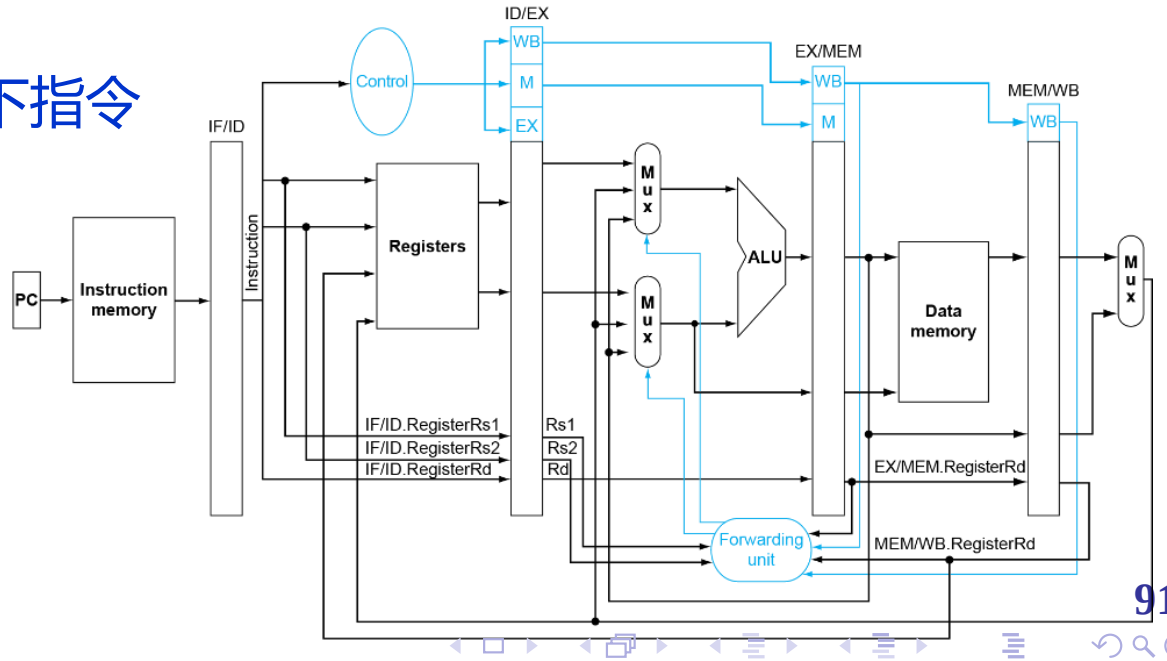


4.7 数据冒险：前递与停顿

- 如何阻塞流水线
  - 强制将ID/EX register中与控制相关的信号设置为0
    - 相关指令在EX、MEM、WB阶段变为无操作（no-operation）
  - PC和IF/ID register暂停更新
    - 再次对原指令（前一周期的指令）进行译码操作
    - 再次对原指令的后续指令进行取指操作
    - 一个周期的阻塞允许Load指令在MEM阶段完成数据加载

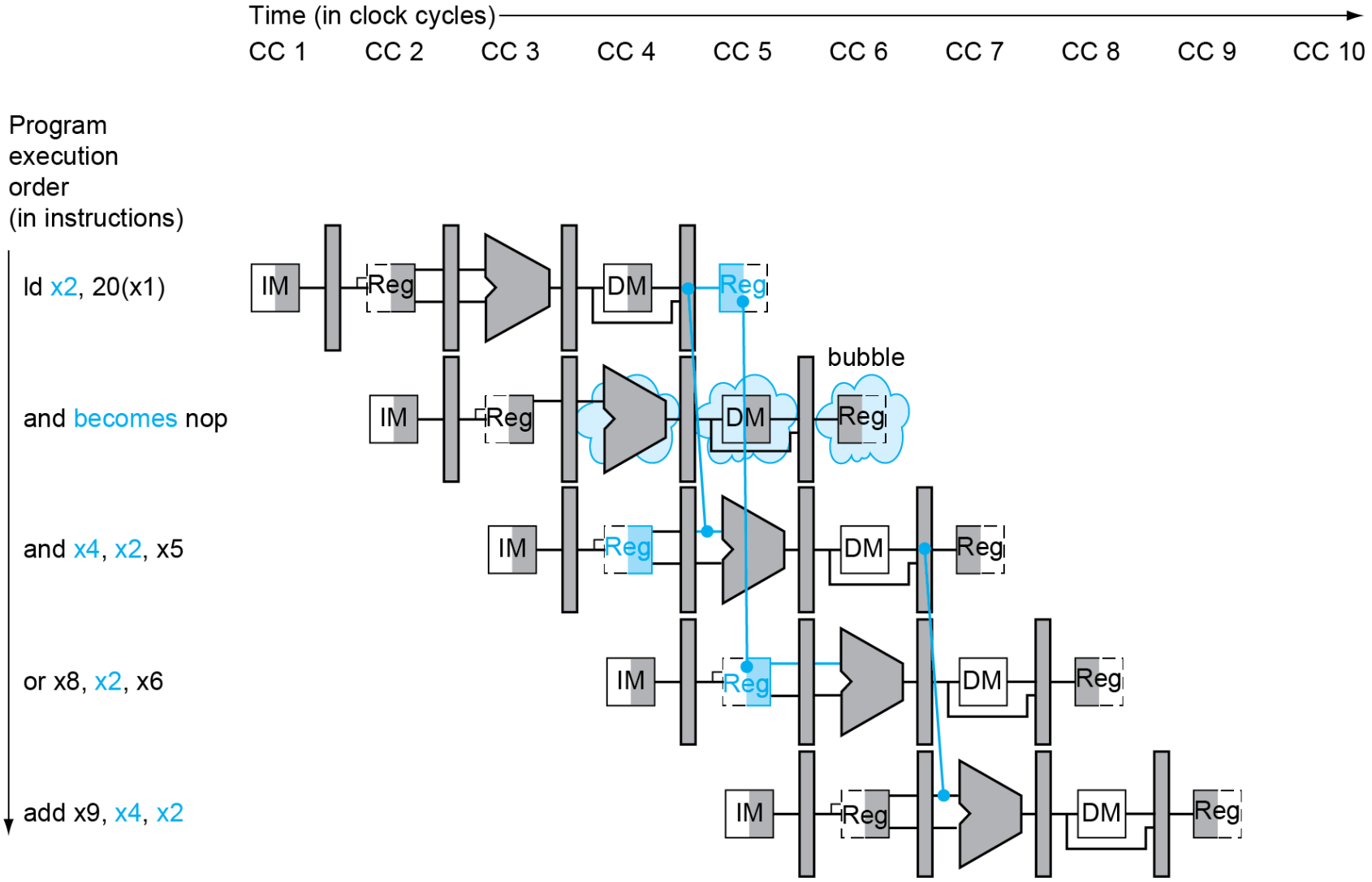
- 示例：在ripes中运行以下指令

```
nop
lw x1,0(x0)
addi x2, x1,1
nop
```

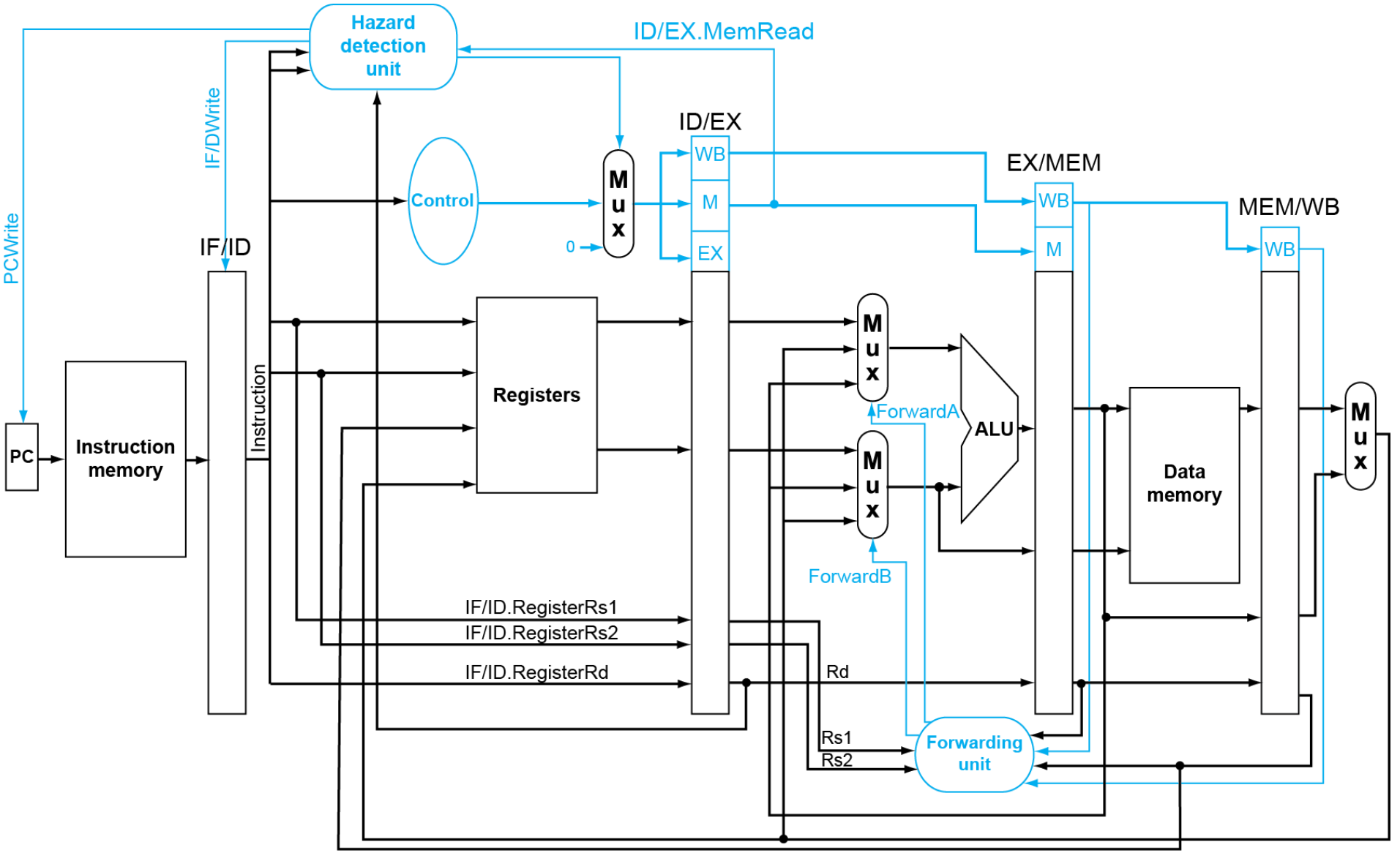


4.7 数据冒险：前递与停顿

- Load-Use Hazard阻塞流水线



4.7 数据冒险：前递与停顿



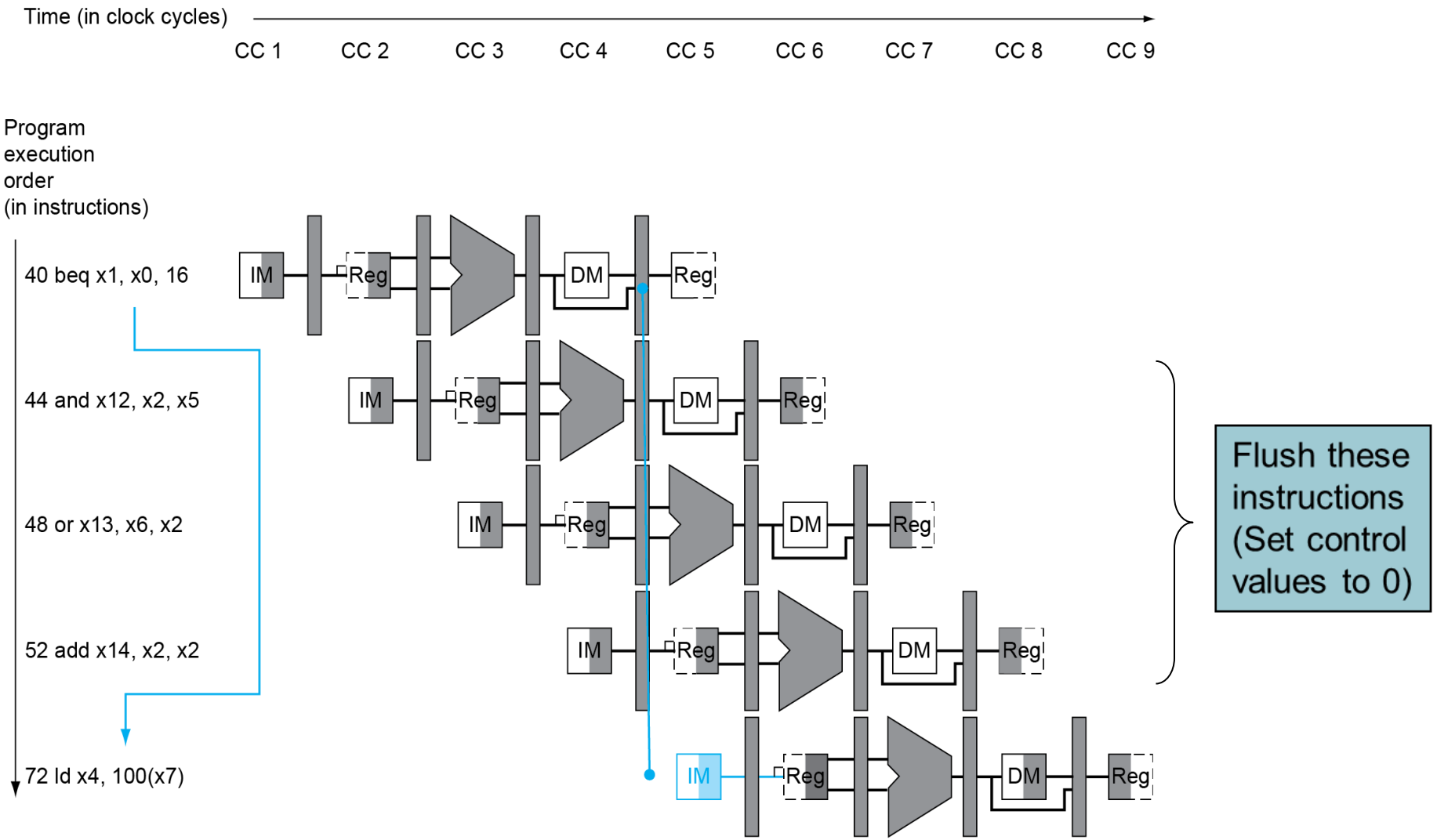
4.7 数据冒险：前递与停顿

- 阻塞对性能的影响
  - 阻塞流水线会降低性能
    - 为使流水线产生正确的结果，阻塞是必须的
  - 编译器可以对代码顺序进行重排，以避免冒险和阻塞
    - 要求编译器开发者或软件工程师对流水线结构有深入理解

- 数据冒险总结
  - R/I-type指令的数据相关问题，可以通过前递技术解决
    - 不会对CPU性能造成损失
  - Load-Use类型的数据相关问题，可通过阻塞+前递技术解决
    - 会造成一个周期的性能损失

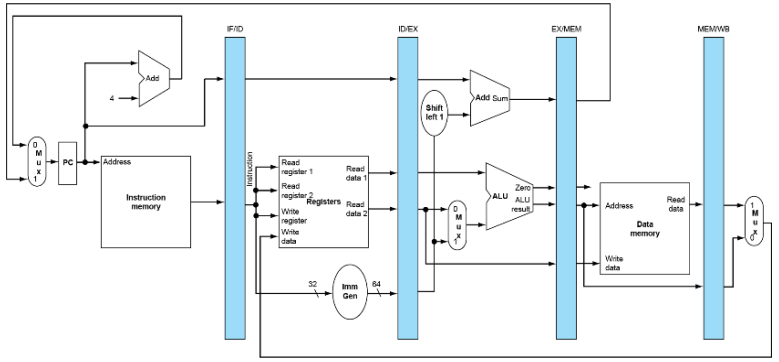
4.8 控制冒险

- 分支指令在MEM阶段产生确定性的结果（存疑，EXE阶段？）



4.8 控制冒险

- 假设分支不发生（指令顺序执行）
- 降低分支指令延时
  - 将分支指令判定相关硬件移到ID阶段实现
    - 寄存器比较
    - 目标地址计算



- 示例：控制冒险（分支发生）

```
36:   sub   x10, x4, x8
40:   beq   x1,  x3, 16   // PC-relative branch
                          // to 40+16*2=72
44:   and   x12, x2, x5
48:   or    x13, x2, x6
52:   add   x14, x4, x2
56:   sub   x15, x6, x7
      ...
72:   ld    x4, 50(x7)
```

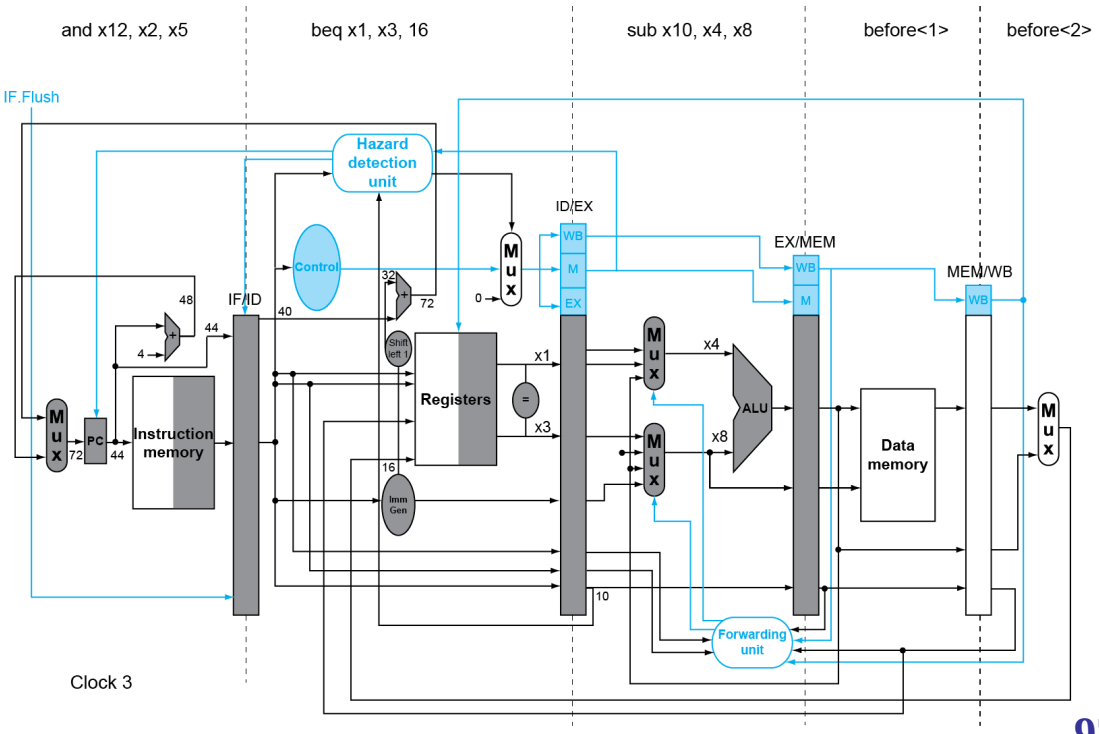
ripes示例程序

```
nop
beq x0,x0, L1
addi x1,x0,1
nop
L1:
addi x2, x1, 1
```

4.8 控制冒险

- 降低分支指令延时
  - 将分支指令判定相关硬件移到ID阶段实现
    - 寄存器比较
    - 目标地址计算
- 示例：控制冒险（分支发生）

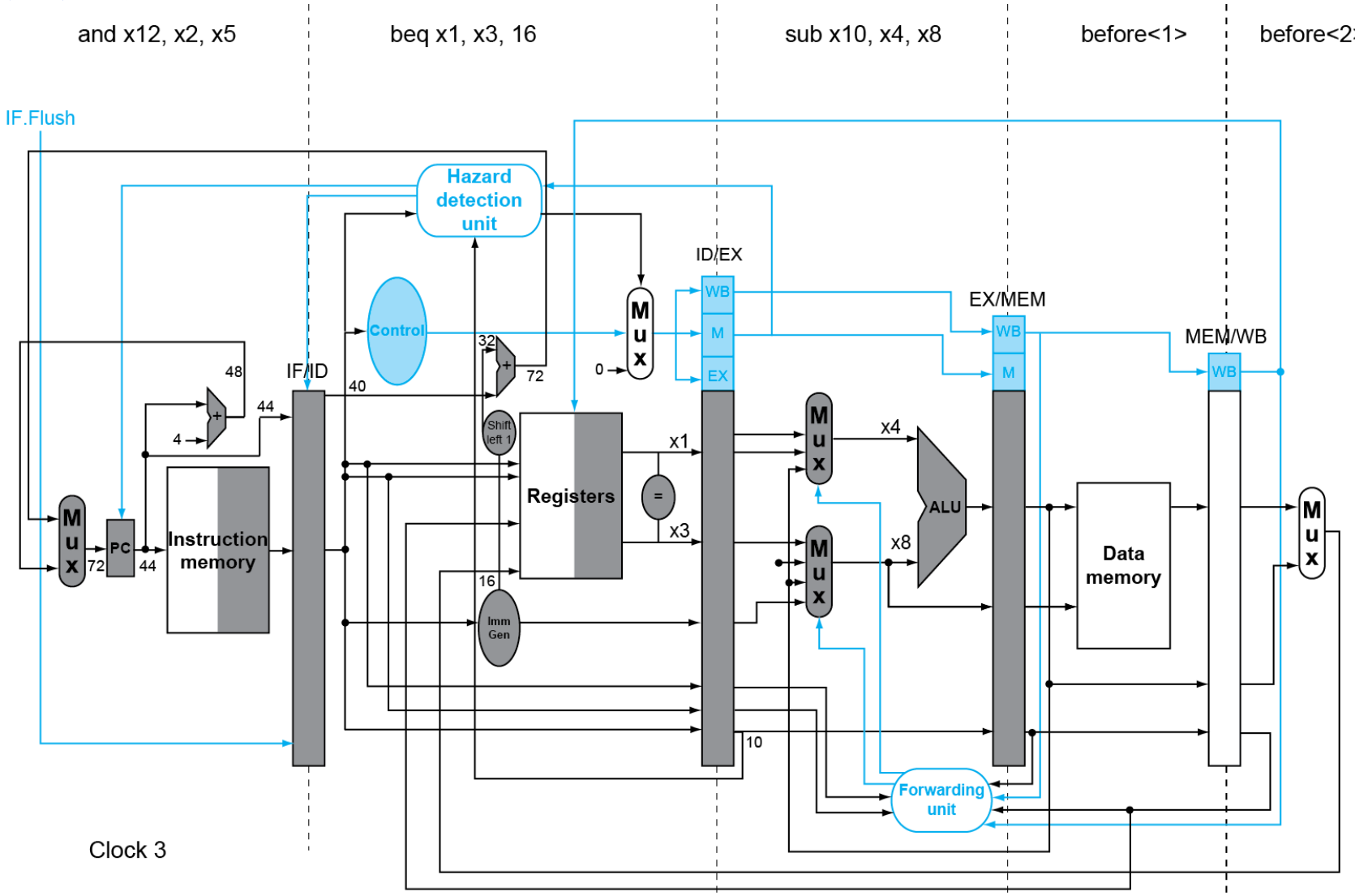
```
36:   sub   x10, x4, x8
40:   beq   x1,  x3, 16
44:   and   x12, x2, x5
      ...
72:   ld    x4, 50(x7)
```



4.8 控制冒险

```
36:   sub   x10, x4, x8
40:   beq   x1,  x3, 16
44:   and   x12, x2, x5
      ...
72:   ld    x4, 50(x7)
```

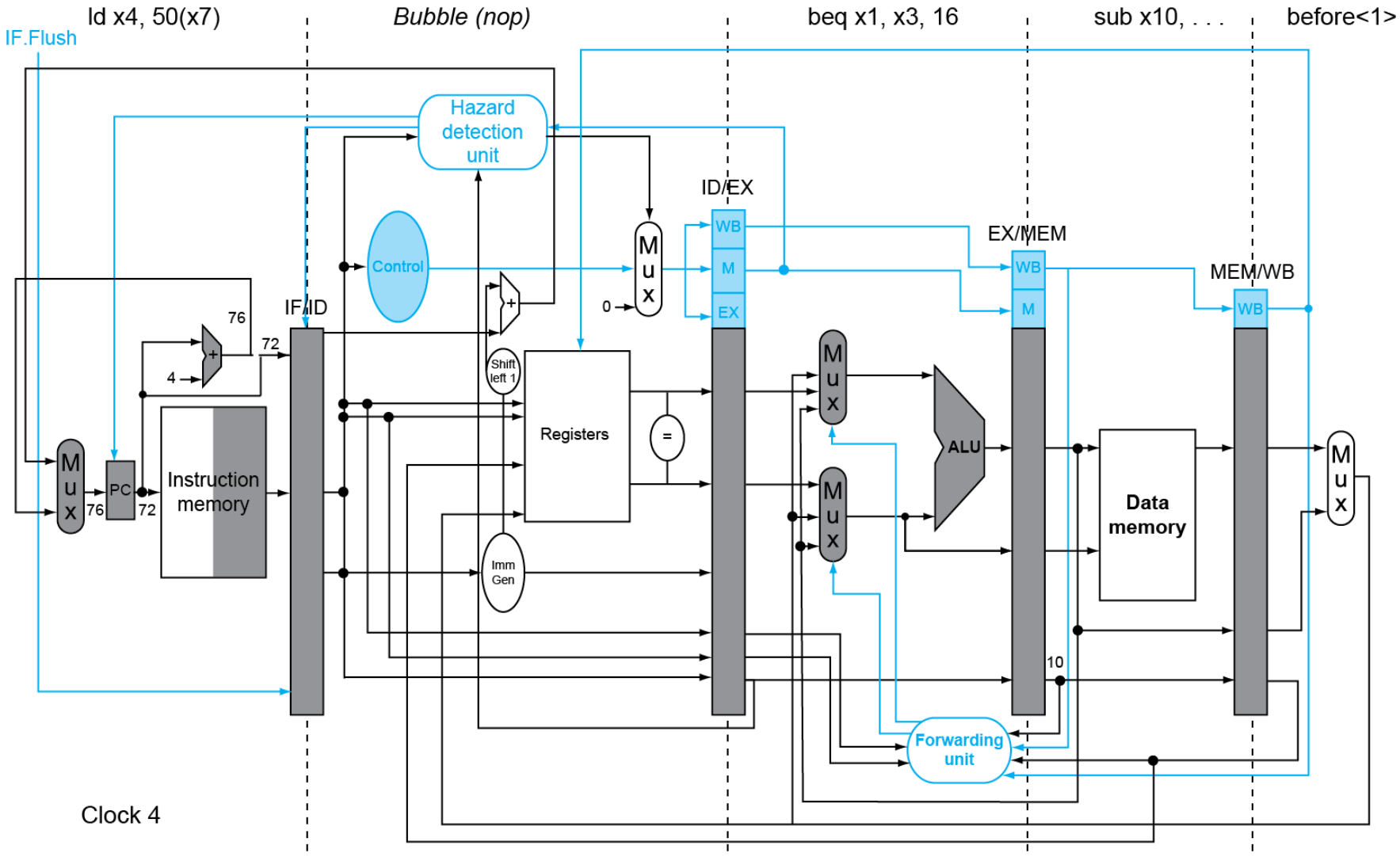
- 分支发生



4.8 控制冒险

```
36:   sub   x10, x4, x8
40:   beq   x1,  x3, 16
44:   and   x12, x2, x5
      ...
72:   ld    x4, 50(x7)
```

- 分支发生

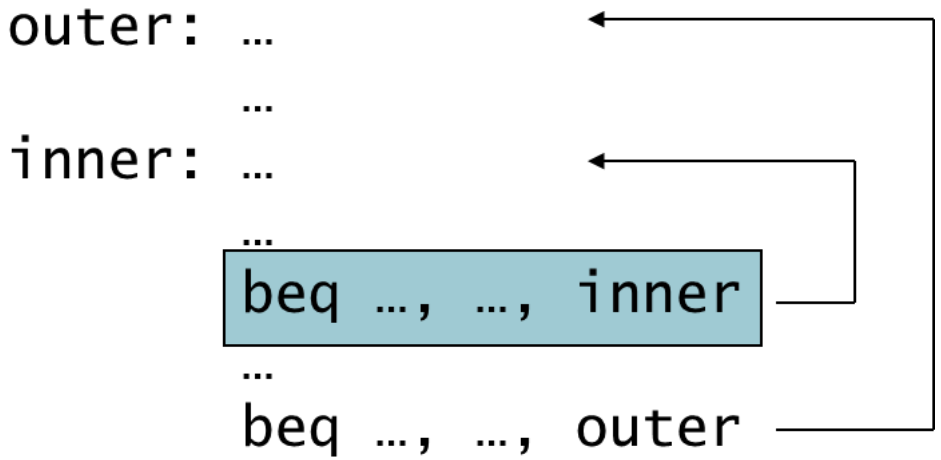


4.8 控制冒险

- branch penalty
  - 分支预测失败时，因冲刷流水线而造成的性能损失
  - 流水线级数越多，损失越大
- 分支预测
  - 静态分支预测、动态分支预测
- 动态分支预测
  - 分支预测缓存
  - 使用近期执行过的分支指令地址（部分地址）作为索引
  - 存储该分支的执行结果：发生或未发生（1bit）
  - 执行分支指令的流程
    - 检查分支预测缓存（如果缓存中没有怎么办？）
    - 根据缓存结果，从目标地址（分支目标或下一条）取指
    - 如果预测失败，冲刷流水线，并修改缓存中的预测结果

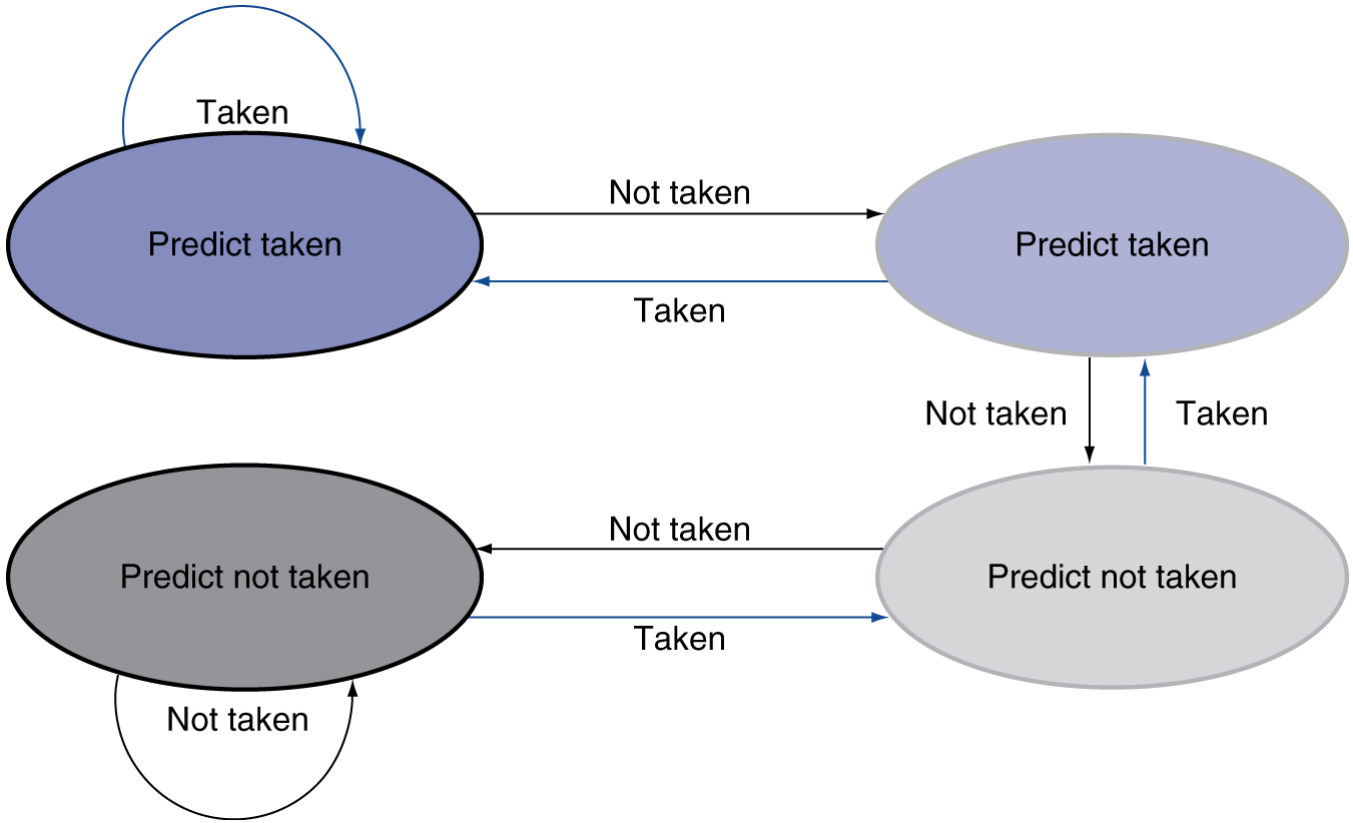
4.8 控制冒险

- 1bit分支预测器的缺点
- 对于多层循环，将导致连续的预测失败
- 以下图为例
  - 内层循环在最后一次迭代时，预测判定失败
  - 当再次进入内层循环，第一次迭代时预测判定也会失败



4.8 控制冒险

- 2bit分支预测
  - 只有连续两次预测失败时才会更改预测结果

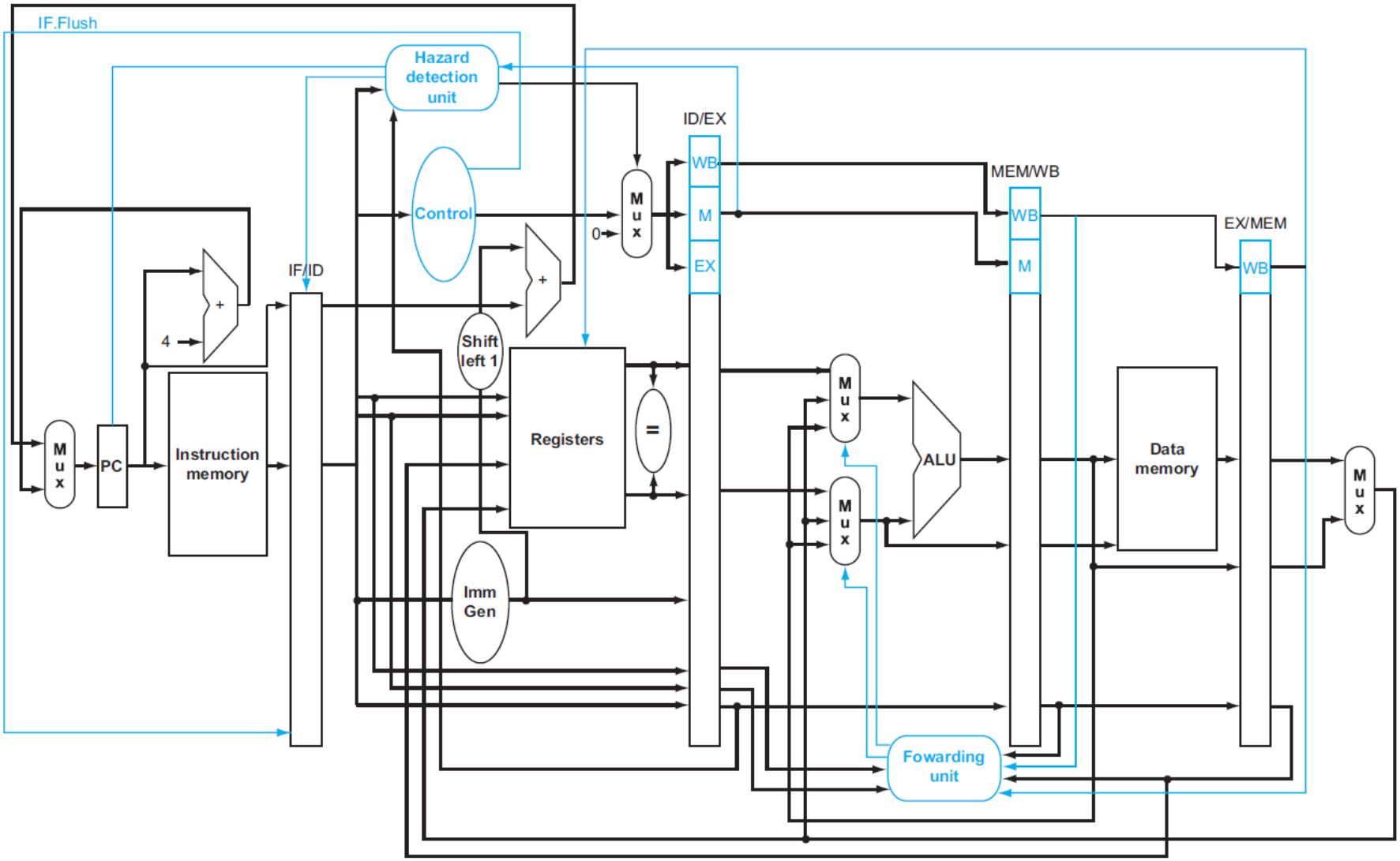


4.8 控制冒险

- 计算分支地址
  - 即便采用分支预测器，仍然需要计算目标地址
  - 发生分支跳转时，需要一个时钟周期的代价来计算分支目标地址
- 采用分支目标缓存器，可解决上述延时问题
  - 将分支目标地址存放在Cache中
  - 使用取指阶段的PC作为索引
  - 满足如下条件时可立即获取分支目标地址
    - 命中
    - 分支预测器预测分支发生

4.8 控制冒险

- 流水线总结



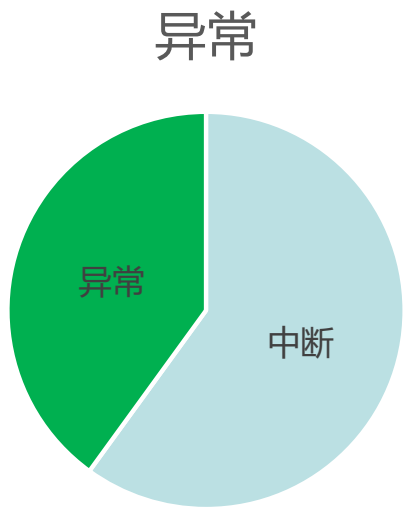
4.8 控制冒险

自我检测 考虑三个分支预测机制：预测分支不发生、预测分支发生以及动态预测。假定它们在预测正确时代价为零，在预测错误时代价为两个时钟周期。假定动态预测的平均预测准确率为90%。对于下面这些分支来说，哪一种预测机制是最佳选择？

1. 分支发生概率为 5% 的条件分支
2. 分支发生概率为 95% 的条件分支
3. 分支发生概率为 70% 的条件分支

4.9 异常

- 异常/例外（exception） 和中断（interrupt）
  - 有时也可统称为异常
- 非预期的突发事件发生时，需要改变程序的控制流
  - 不同的ISA具有不同的实现方式
- 异常
  - 发生在CPU内部的非预期事件
  - eg：未定义指令、系统调用等
- 中断
  - 来自外部的非预期事件
  - 键鼠操作、U盘插入等
- 很难在不影响性能的前提下处理异常及中断



4.9 异常

- 异常处理
  - 保存异常发生时的PC值（保存在什么地方？）
    - RISC-V：Supervisor Exception Program Counter (SEPC)
  - 保存异常发生的原因
    - RISC-V：Supervisor Exception Cause Register (SCAUSE)
  - 跳转到异常处理函数
    - eg：目标地址位于$0000\ 0000\ 1C09\ 0000_{hex}$

4.9 异常

- 异常处理的一种替代方案
  - 向量式中断
    - 处理函数的入口地址与cause寄存器有关
  - 处理函数入口地址由基址寄存器加上异常代码（cause寄存器作为偏移）得出

例外类型	例外向量地址，即偏移，与中断向量表基地址相加
未定义指令	$00\ 0100\ 0000_2$
系统错误（硬件故障）	$01\ 1000\ 0000_2$

4.9 异常

- 处理流程
  - 读取异常原因，跳转到相应入口地址
  - 确定后续动作
  - 对于可恢复的异常
    - 完成对异常的正确处理
    - 利用SEPC寄存器返回原程序继续执行
  - 对于不可恢复异常
    - 中止程序
    - 利用SEPC、SCAUSE等寄存器报告错误信息

4.9 异常

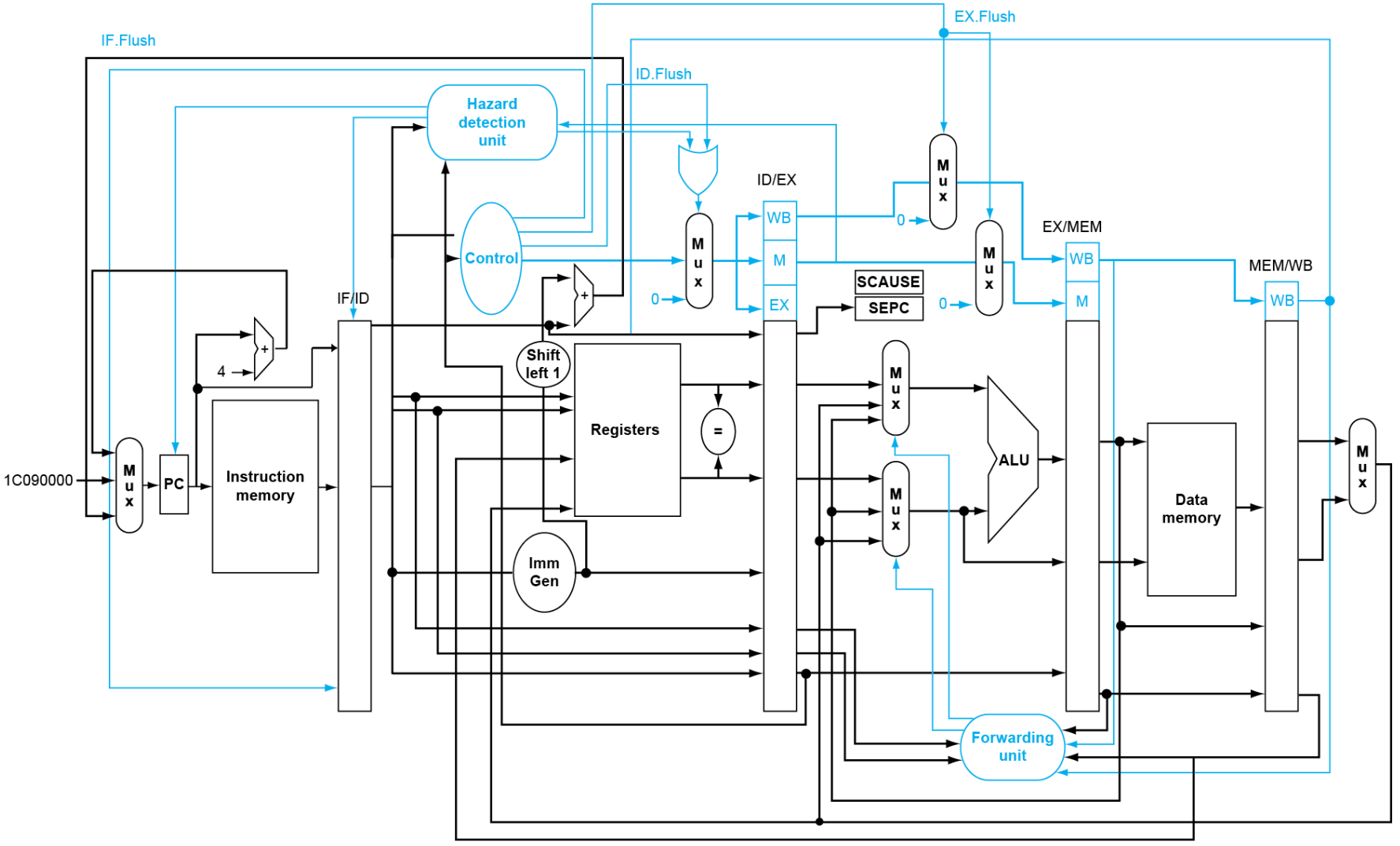
- 流水线中实现的异常

- 异常会导致控制冒险
  - Eg：add指令在EX阶段产生硬件故障
    - add x1, x2, x1
  - 阻止相关寄存器或存储器被修改
  - 完成该指令之前的指令
  - 冲刷已进入流水线的后续指令
  - 设置SEPC和SCAUSE寄存器
  - 将控制权转移到异常处理函数

- 与分支跳转指令预测失败时的流程相似
  - 两者复用了大部分的硬件电路

4.9 异常

- 支持异常的流水线数据通路



4.9 异常

- 异常属性

- 可重启（重新执行）的异常
  - 流水线可冲刷掉该条指令
  - 异常处理函数执行完毕后，重新回到触发异常的指令，继续执行
    - 指令重新进入流水线
- 返回地址保存在SEPC寄存器中
  - Identifies causing instruction

4.9 异常

- 异常示例：add指令发生异常
  - 疑问：add指令能发生什么异常？？？
- 以下代码中add指令发生异常

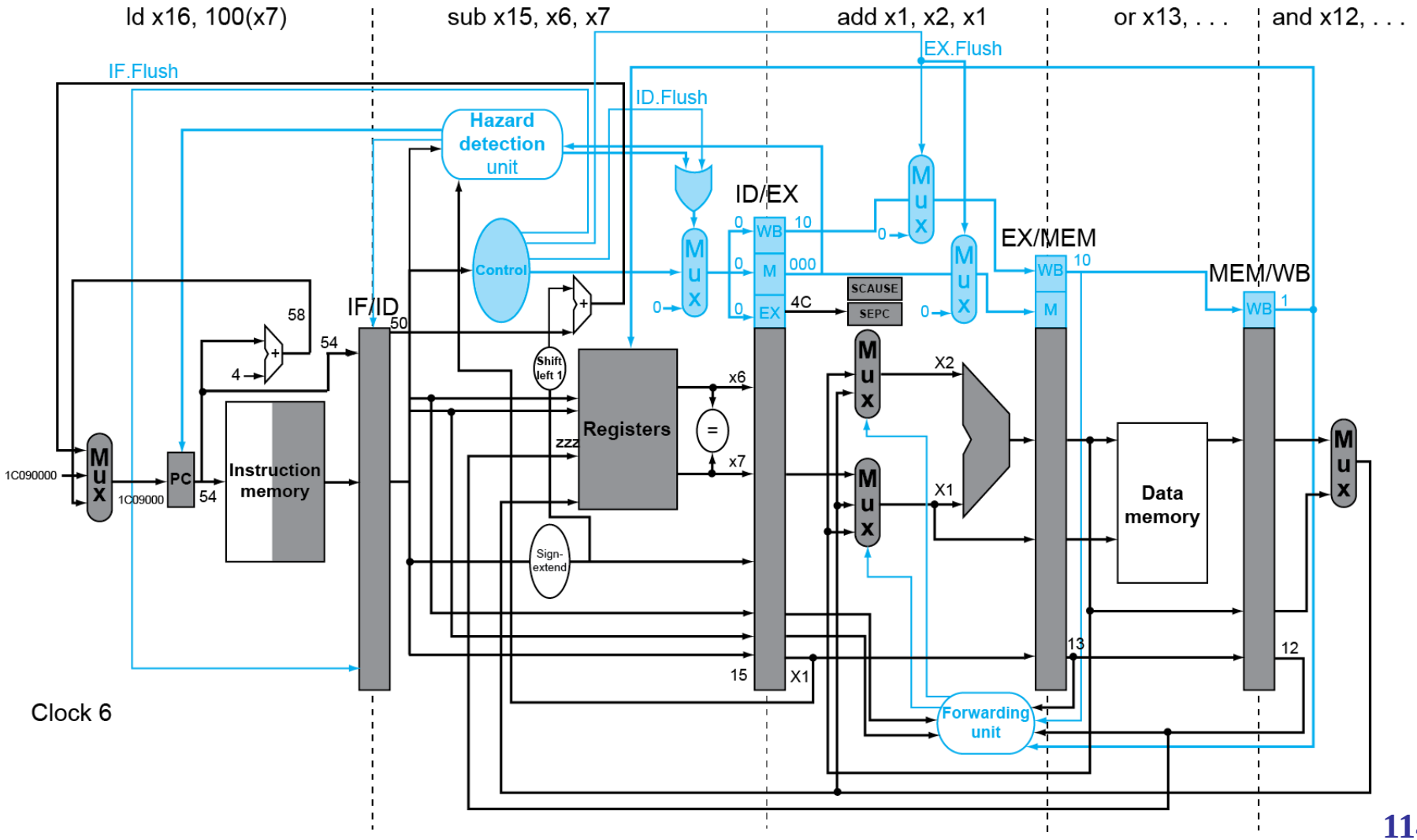
```
40          sub  x11, x2, x4
44          and  x12, x2, x5
48          or    x13, x2, x6
4c          add  x1,  x2, x1
50          sub  x15, x6, x7
54          ld    x16, 100(x7)
...
```

- Handler

```
1C090000    sd    x26, 1000(x10)
1C090004    sd    x27, 1008(x10)
...
```

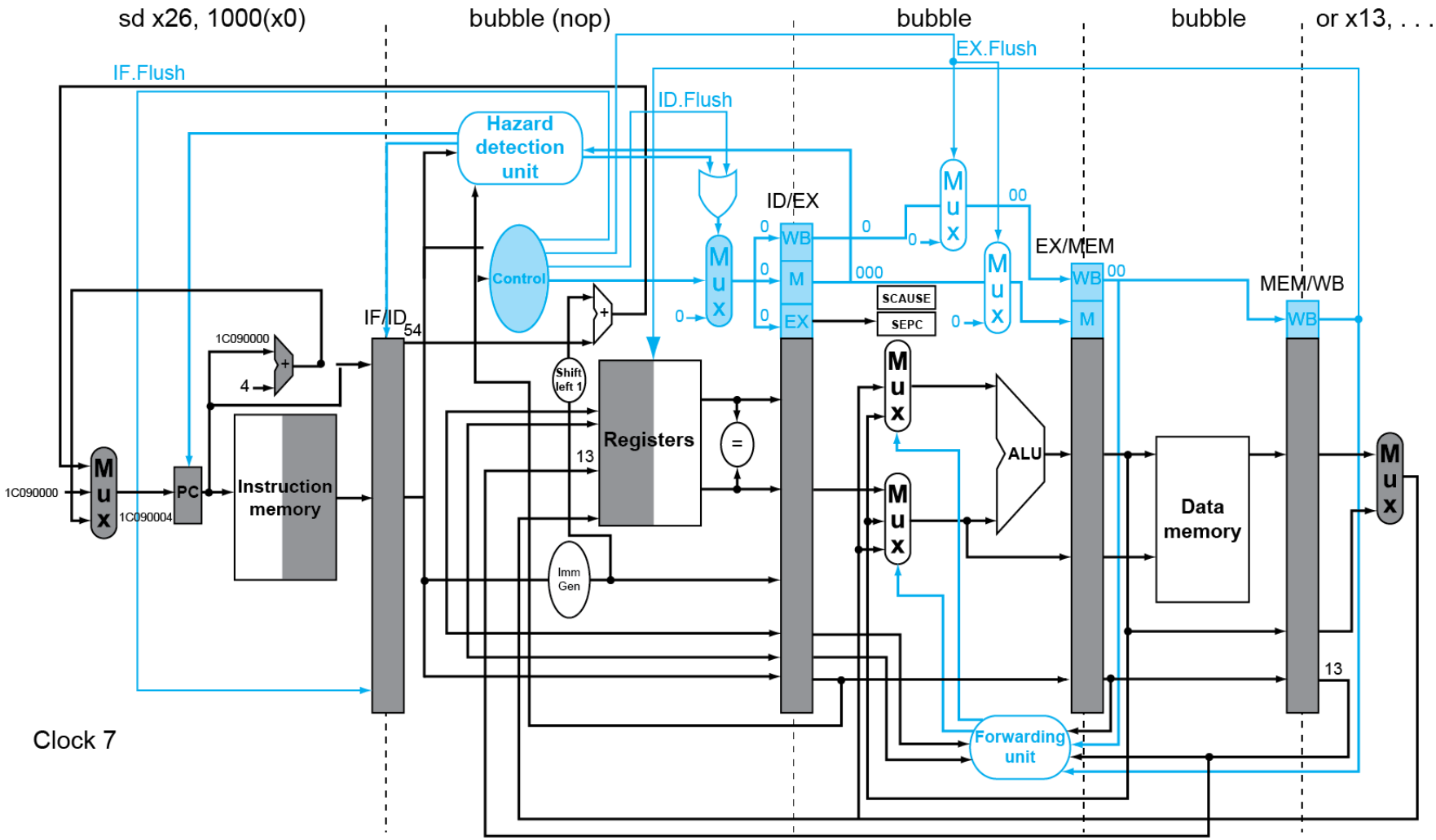
4.9 异常

- 异常示例：add指令发生异常



4.9 异常

- 异常示例：add指令发生异常-续



4.9 异常

- 多个异常同时发生
  - 流水线中同一时刻有多条指令在执行
  - 因此有可能会同时产生多个异常
- 如何处理?
  - 简单方案：处理最早一条指令引发的异常
    - 冲刷流水线
    - 要求处理器支持精确异常
    - 什么是精确异常?
- 精确异常与非精确异常
  - 异常发生时，SEPC寄存器保存的是否是引发异常的指令地址
  - 在复杂流水线中
    - 多发射：一个周期会发射多条指令
    - 乱序执行：处理器会根据情况调整指令执行的先后次序
    - 上述特性使得实现精确异常非常困难

4.9 异常

- 非精确异常
  - 阻塞流水线，并保存状态（包括引发异常的原因）
  - 异常处理程序的工作
    - 判定哪条指令引发了异常
    - 判定哪些指令需要完成、哪些指令需要被冲刷
    - 必要时可能需要手动确认
  - 优缺点
    - 优点：简化了硬件
    - 缺点：异常处理函数更加复杂
  - 不适用于复杂的多发射、乱序执行流水线
- RISC-V支持精确异常

4.10 指令间的并行性

- ILP：指令级并行
- 流水线可以并行执行多条指令
- 增加指令级并行性（ILP）的途径
  - 增加流水线级数
    - 每级流水线承担更少的工作 → 更短的时钟周期，更高的时钟频率
  - 多发射技术：一个周期内可以发射多条指令
    - 复制流水线各阶段的相关部件 → 多个流水线
    - 每个时钟周期可以开始多条指令
    - CPI < 1, 引入IPC概念（Instructions Per Cycle）
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - 各种依赖会降低实际的性能
- 发射槽
  - 指令发射时所处的位置，可类比为起跑位置

4.10 指令间的并行性

- 多发射
  - 静态多发射、动态多发射
- 静态多发射
  - 定义：多发射的一种实现方法，由编译器完成发射相关判断
  - 编译器将相关指令组织成一组
  - 编译器将相关指令打包放入“发射槽”
  - 编译器检测并处理各种冒险
- 动态多发射
  - 定义：多发射的一种实现方法，在动态执行过程中，由硬件完成发射相关判断
  - 在每个周期，CPU检查指令流，并选择可发射指令
  - 编译器可协助参与指令重排序
  - CPU在运行时依靠硬件处理各种冒险

4.10 指令间的并行性

- 推测
  - 定义：编译器或处理器“猜测”指令的行为，以尽早消除掉该指令与其他指令之间的依赖关系
  - 尽早开始执行操作
  - 检查“猜测”是否正确
    - 如果正确，完成操作
    - 如果不正确，回溯并完成正确的操作
  - Eg：推测分支不发生，分支后的指令可以提早执行
  - Eg：store + load指令，推测地址不同，load指令可以提早执行

  - 推测技术的缺点
    - 推测错误时，恢复机制非常复杂，实现非常困难
    - 推测可能引入不必要的异常
    - 推测错误时，将对处理器产生较大影响

4.10 指令间的并行性

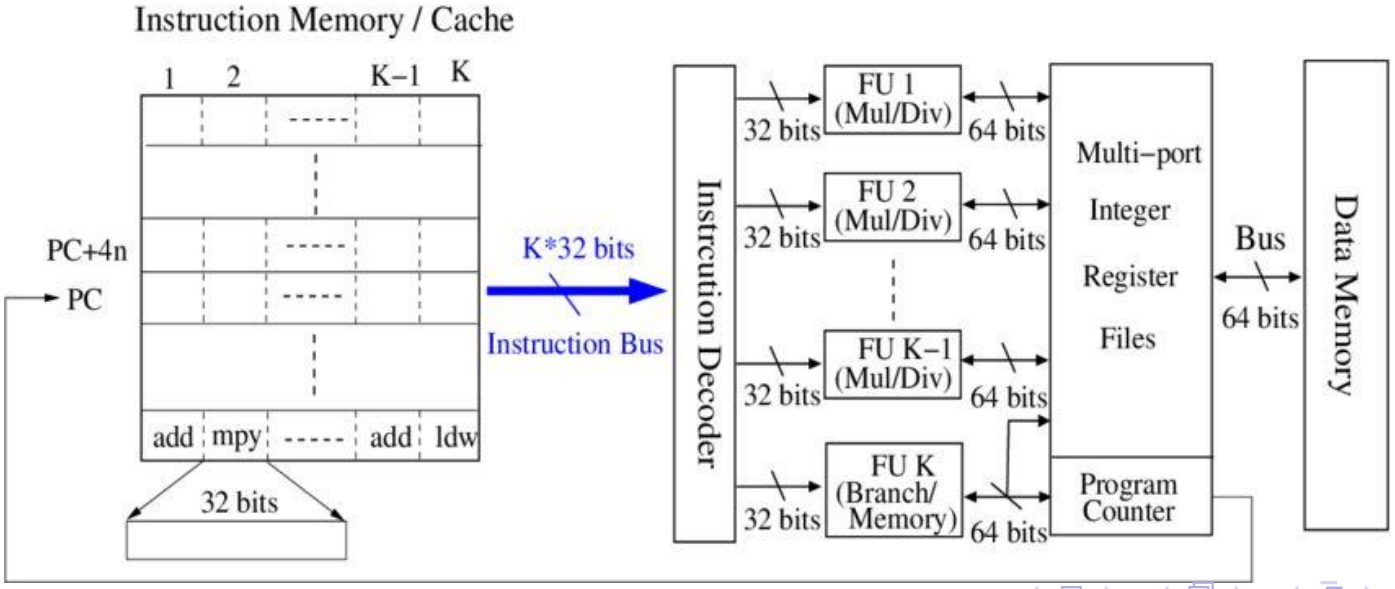
- 编译器推测 vs 硬件推测
  - 静态推测 vs 动态推测
- 编译器可以对指令序列进行重排
  - 可以加入“定位”指令，以帮助从错误推测中进行恢复
- 硬件可以提前执行指令
  - 将结果进行缓存
  - 如推测成功，指令被执行到，则将结果提交
  - 如推测失败，指令没有被执行，则清除相关缓存

4.10 指令间的并行性

- 推测和异常
- 如果推测执行的指令出现异常怎么办?
  - 如：推测性加载前没有进行检查的指针（可能为空指针）
- 静态推测中的异常处理
  - 添加特定支持来避免，延迟响应直到确认推测正确
- 动态推测中的异常处理
  - 缓冲异常，直到指令完成（可能不需要完成）
    - 如指令需要完成，则处理相应异常
    - 如指令不需要完成，则清除异常

4.10 指令间的并行性

- 静态多发射
  - 编译器将指令分组为“发射包”
    - 可以在一个周期内发出的一组指令
    - 由所需的流水线硬件资源决定
  - 将发射包视为非常长的指令
    - 指定多个并发操作——超长指令字（VLIW）
  - 超长指令字：一种类型的指令系统体系结构，支持在单条指令中使用不同的编码位来定义多个可同时被发射的独立操作



4.10 指令间的并行性

- 静态多发射的调度

- 编译器必须消除部分或所有冒险
  - 将指令重新排序到发射包中
  - 发射包内的指令相互之间没有依赖关系
  - 不同的发射包之间可能存在一些依赖关系
    - 不同的ISA各不相同
    - 编译器一定知道
  - 如有必要，用nop填充

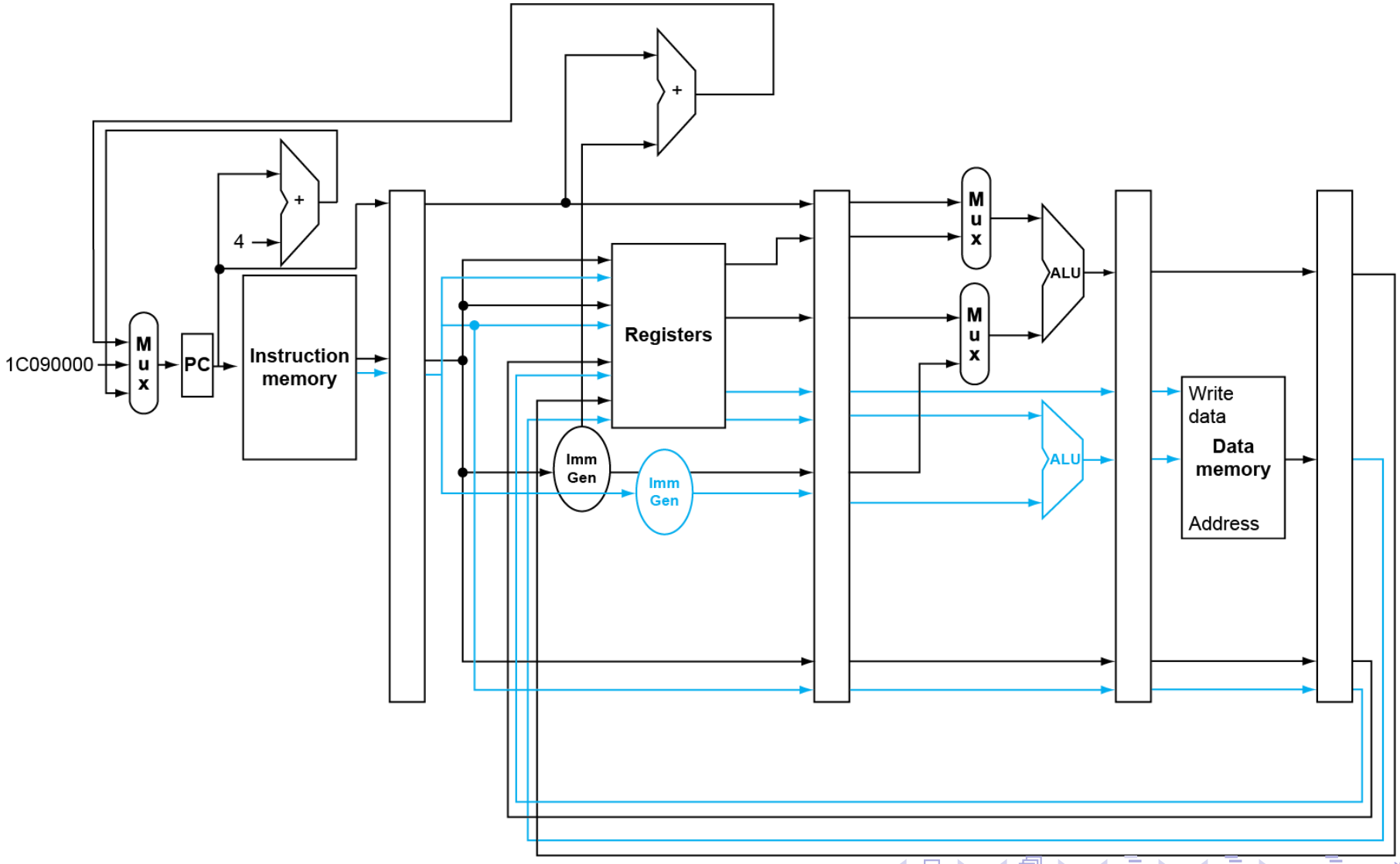
4.10 指令间的并行性

- RISC-V中的静态双发射
- 双发射指令包
  - 一条ALU/branch指令
  - 一条Load/Store指令
  - 64bit对齐（一条指令32bit）
    - ALU/branch + Load/Store
    - 必要时可插入nop指令

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

4.10 指令间的并行性

- RISC-V静态双发射数据通路



4.10 指令间的并行性

- RISC-V双发射设计中的冒险
  - 更多并行执行的指令
  - 从而导致更多的冒险
- 数据冒险
  - 单发射设计中可通过前递避免流水线阻塞
  - 双发射设计中呢?
- 加载/存储指令在同一发射包中，因此不能直接使用ALU结果
  - add x10, x0, x1
  - ld   x2, 0(x10)
  - 分成两个包，实际上等同于一次阻塞
- 加载指令的使用延迟
  - 加载指令有一个周期的使用延迟，但现在有两条指令
- 需要更激进的调度策略

4.10 指令间的并行性

- 调度示例： A[i] = A[i] + C

```
Loop: ld   x31,0(x20)     // x31=array element
      add  x31,x31,x21    // add scalar in x21
      sd   x31,0(x20)     // store result
      addi x20,x20,-8     // decrement pointer
      blt  x22,x20,Loop   // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
Loop:	nop	ld x31,0(x20)	1
	addi x20,x20,-8	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sd x31,8(x20)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

4.10 指令间的并行性

- 循环展开（loop unrolling）
  - 一种针对数组访问循环体的提高程序性能的技术，将循环体展开多遍，对不同循环体内的指令进行统一调度。
- 复制循环体以挖掘更多并行性
  - 减少环路控制开销
  - 使用更多的寄存器，代码量增加
- 每次复制使用不同的寄存器
  - 称为“寄存器重命名”
  - 避免循环携带的“反依赖”
    - 存储，然后加载相同的寄存器
    - 又名“名字依赖”
  - 重新使用寄存器名

4.10 指令间的并行性

- 循环展开示例

	ALU/branch	Load/store	cycle
Loop:	addi x20,x20,-32	ld x28, 0(x20)	1
	nop	ld x29, 24(x20)	2
	add x28,x28,x21	ld x30, 16(x20)	3
	add x29,x29,x21	ld x31, 8(x20)	4
	add x30,x30,x21	sd x28, 32(x20)	5
	add x31,x31,x21	sd x29, 24(x20)	6
	nop	sd x30, 16(x20)	7
	blt x22,x20,Loop	sd x31, 8(x20)	8

- IPC = 14/8 = 1.75
  - 接近2， 但是要以寄存器和代码大小为代价

4.10 指令间的并行性

- 动态多发射处理器
  - 也称超标量处理器
- 超标量
  - 一种高级流水线技术，指处理器能够在动态执行时选择指令，并在一个周期内执行一条以上的指令
- 由CPU决定一个周期内发射的指令数：0,1,2
  - 避免结构冒险和数据冒险
- 避免了编译器调度的需要
  - 尽管它可能仍然有帮助
  - 代码语义由CPU保证

4.10 指令间的并行性

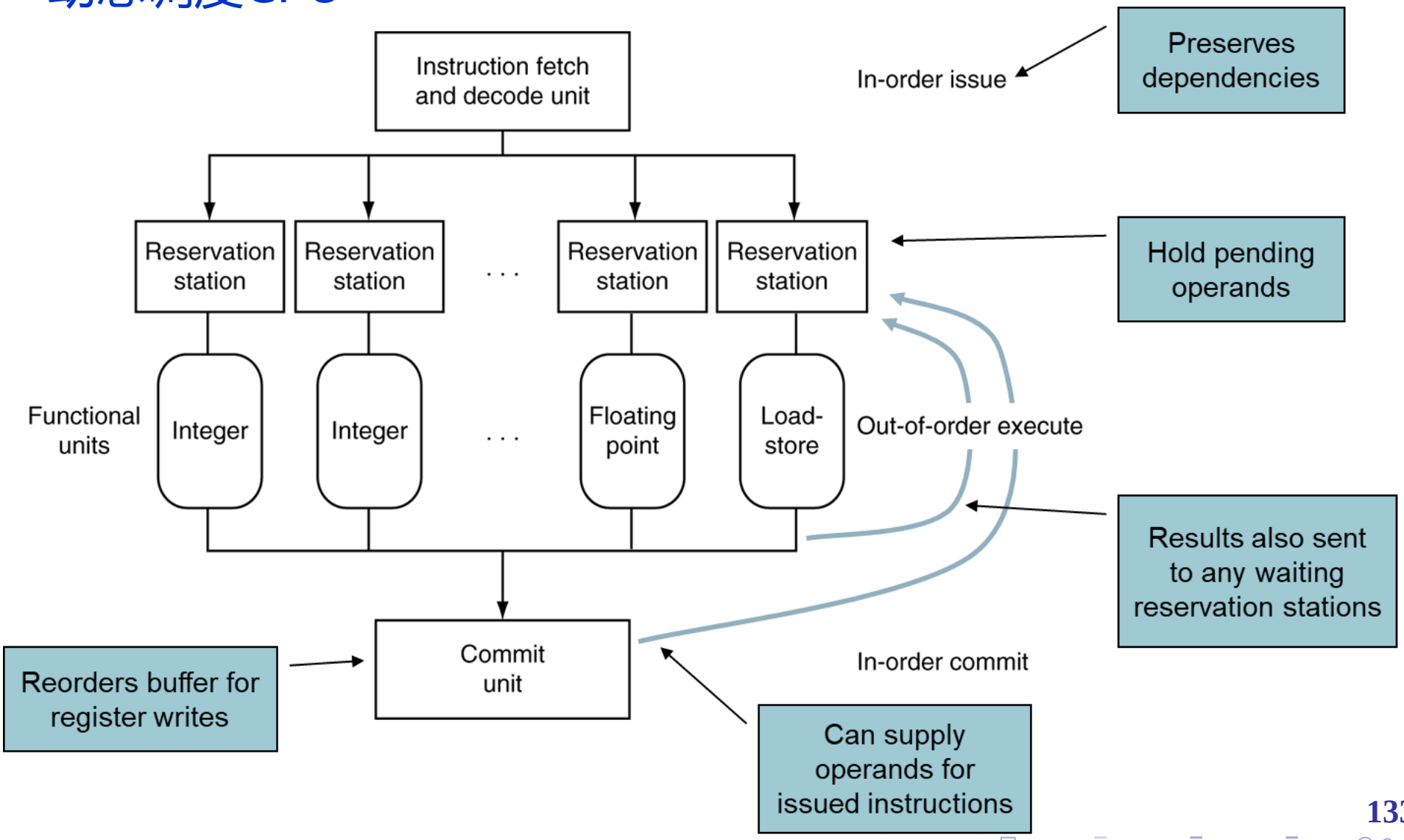
- 动态流水线调度
  - 指一种为避免停顿流水线，对指令执行顺序进行重排的硬件技术
  - 乱序执行、顺序提交
- 允许CPU无序执行指令以避免暂停
  - 但结果按顺序提交到寄存器
- 示例

```
ld   x31,20(x21)
add  x1,x31,x2
sub  x23,x23,x3
andi x5,x23,20
```

  - 在add等待ld时可以启动sub

4.10 指令间的并行性

- 动态调度CPU



4.10 指令间的并行性

- 寄存器重命名
  - 编译器或硬件对寄存器进行重新命名，消除指令序列中的反相关
- 保留站和重排序缓冲两者共同提供了一种寄存器重命名的实现方式
  - 保留站：功能部件前的缓冲区，用来存放指令的操作和所需操作数
  - 重排序缓冲：动态调度处理器中用来保存指令执行结果的缓冲区。一旦指令确认将被提交，将会把缓冲区中的结果写入内存或者寄存器中
- 关于向保留站发射指令
  - 如果操作数在寄存器文件或重排序缓冲区中可用
    - 复制到保留站
    - 寄存器文件中的内容可以被覆盖掉
  - 如果操作数还不可用
    - 由一个功能单元提供给保留站
    - 旁路掉寄存器文件

4.10 指令间的并行性

- 推测

- 预测分支，并按照预测结果持续发射指令
  - 在分支结果确定之前不进行提交操作
- 加载预测
  - 避免加载及缓存失效所造成的延迟
    - 预测有效地址
    - 预测加载数值
    - 在完成存储操作之前执行加载操作
    - 将Store操作的数值直接旁路到加载单元
  - 在猜测确认之前不要提交加载

4.10 指令间的并行性

- 为什么要进行动态调度
- 为什么不让编译器调度代码呢?
- 并非所有的阻塞都是可以预测的
  - 例如，缓存未命中
- 分支指令附近的指令调度存在困难
  - 分支结果是动态确定的
- ISA的不同实现具有不同的延迟和冒险

4.10 指令间的并行性

- 多发射能工作吗?
  - 是的，但没有我们想要的那么多
- 程序的各种依赖和相关会限制ILP
- 有些依赖关系很难消除
  - 例如，指针别名
- 有些并行性很难揭示
  - 指令发射期间限制窗口大小
- 内存延迟和有限的带宽
  - 流水线很难保持全负荷运作
- 如果做得好，投机可能会有所帮助

4.10 指令间的并行性

- 能效/功耗

- 动态调度和推测的复杂性需要消耗能量
- 多个更简单的内核可能更好

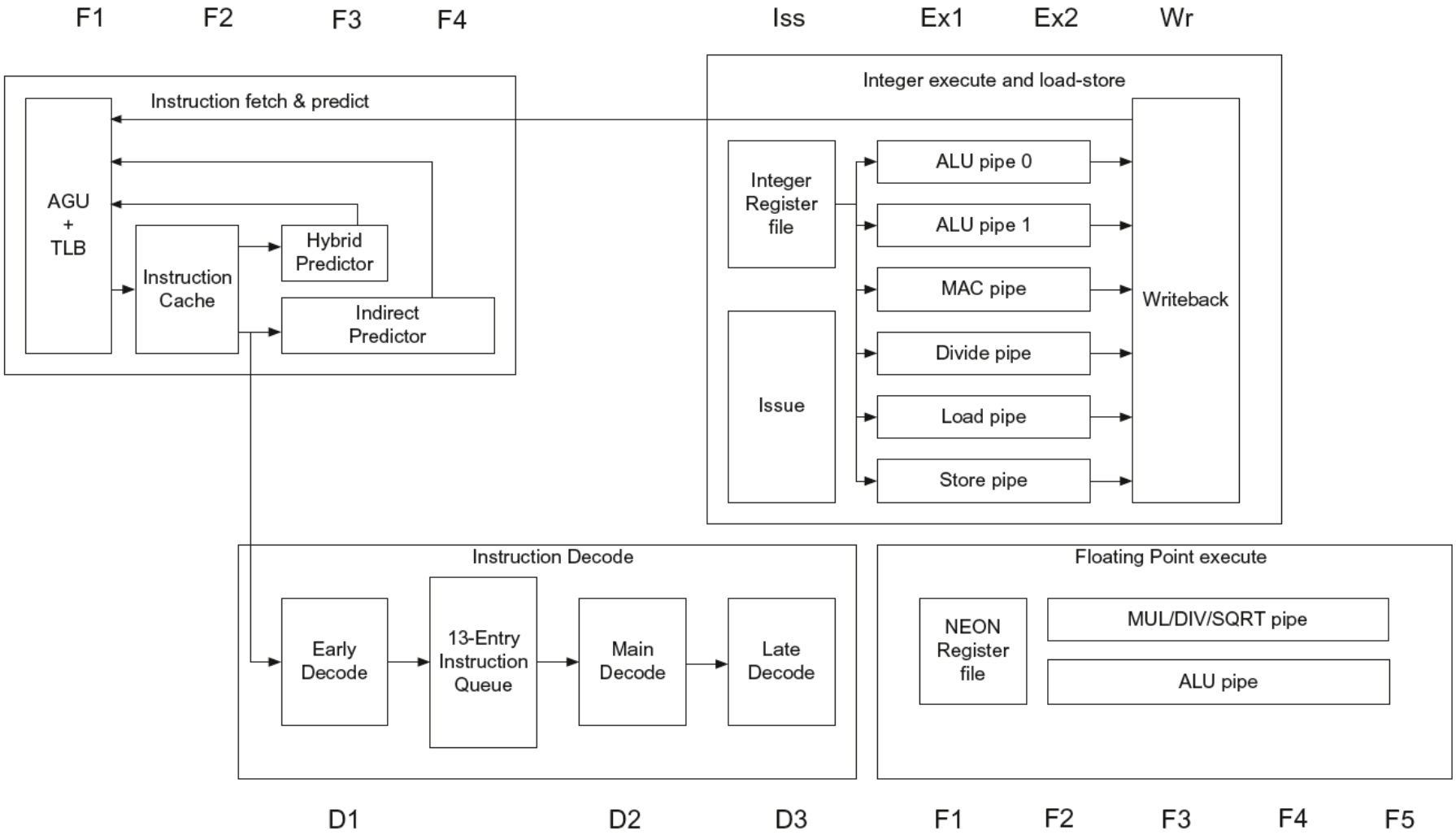
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

4.11 实例：Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

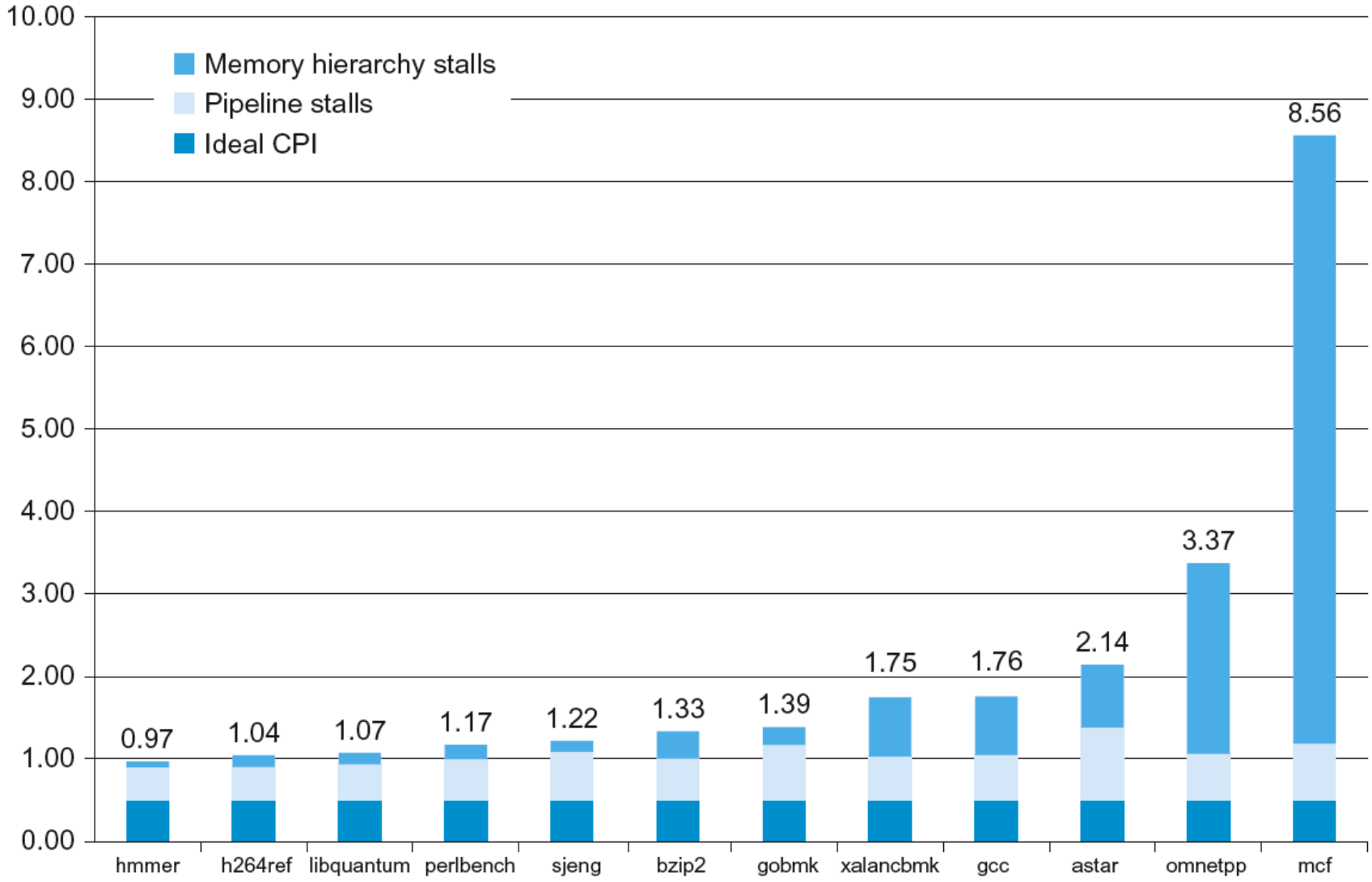
4.11 实例：Cortex A53 and Intel i7

- ARM Cortex-A53 流水线



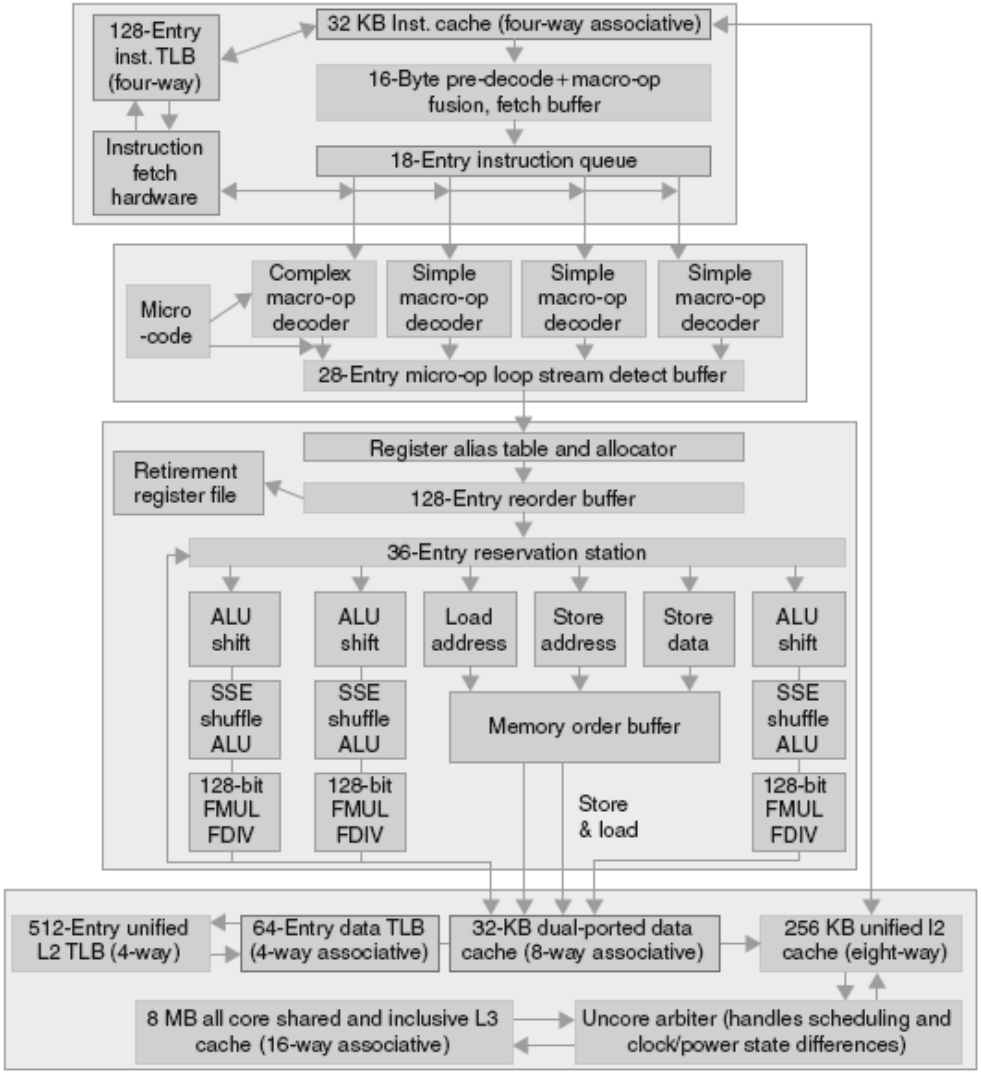
4.11 实例：Cortex A53 and Intel i7

- ARM Cortex-A53 性能



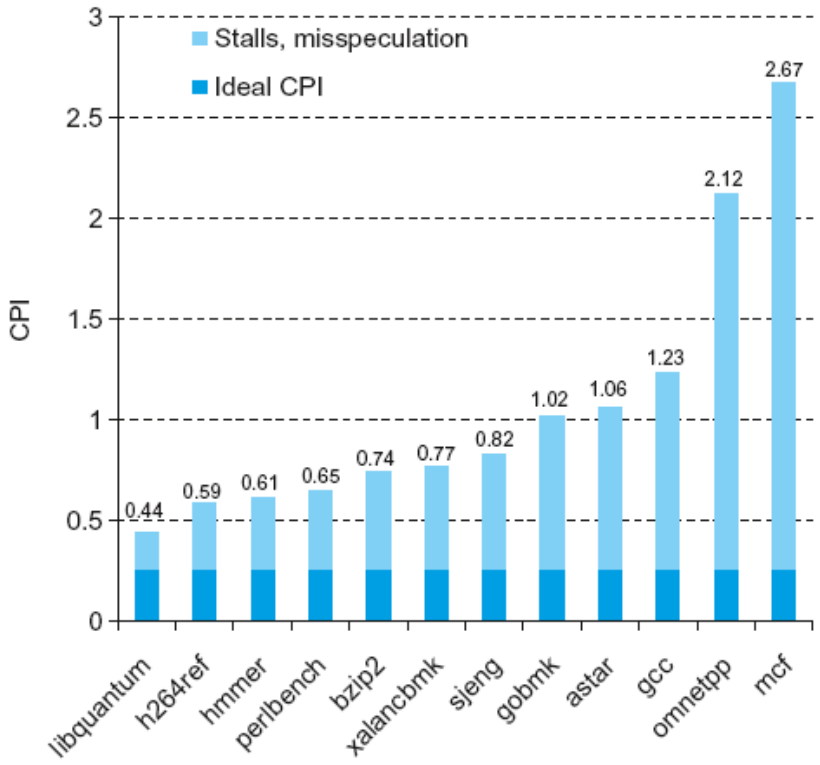
4.11 实例：Cortex A53 and Intel i7

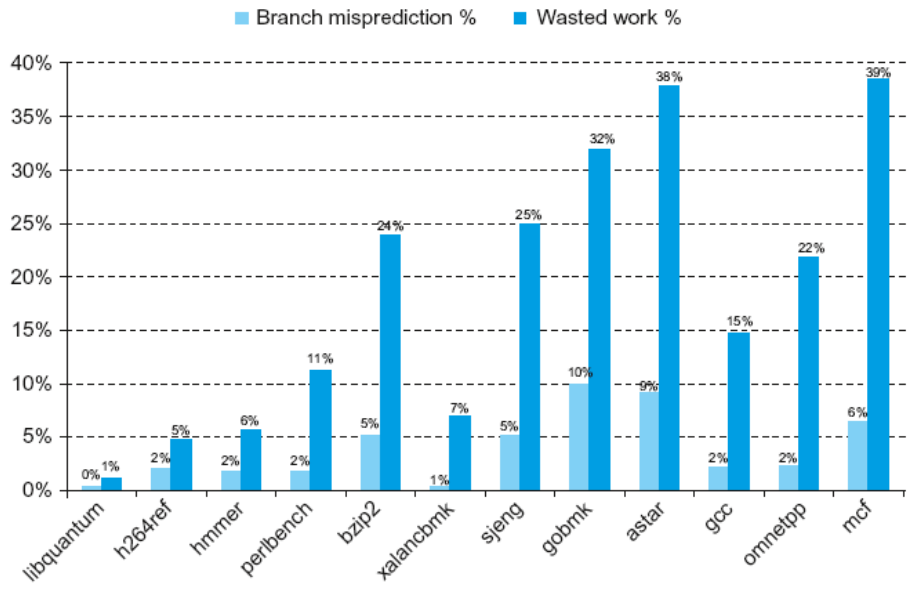
- Core i7 流水线



4.11 实例：Cortex A53 and Intel i7

- Core i7 性能





4.12 加速：指令级并行和矩阵乘法

- 矩阵乘法
- C代码展开

```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6  for ( int i = 0; i < n; i+=UNROLL*4 )
7   for ( int j = 0; j < n; j++ ) {
8    __m256d c[4];
9    for ( int x = 0; x < UNROLL; x++ )
10    c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12   for( int k = 0; k < n; k++ )
13   {
14    __m256d b = _mm256_broadcast_sd(B+k+j*n);
15    for (int x = 0; x < UNROLL; x++)
16    c[x] = _mm256_add_pd(c[x],
17                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18   }
19
20    for ( int x = 0; x < UNROLL; x++ )
21     _mm256_store_pd(C+i+x*4+j*n, c[x]);
22  }
23 }
```

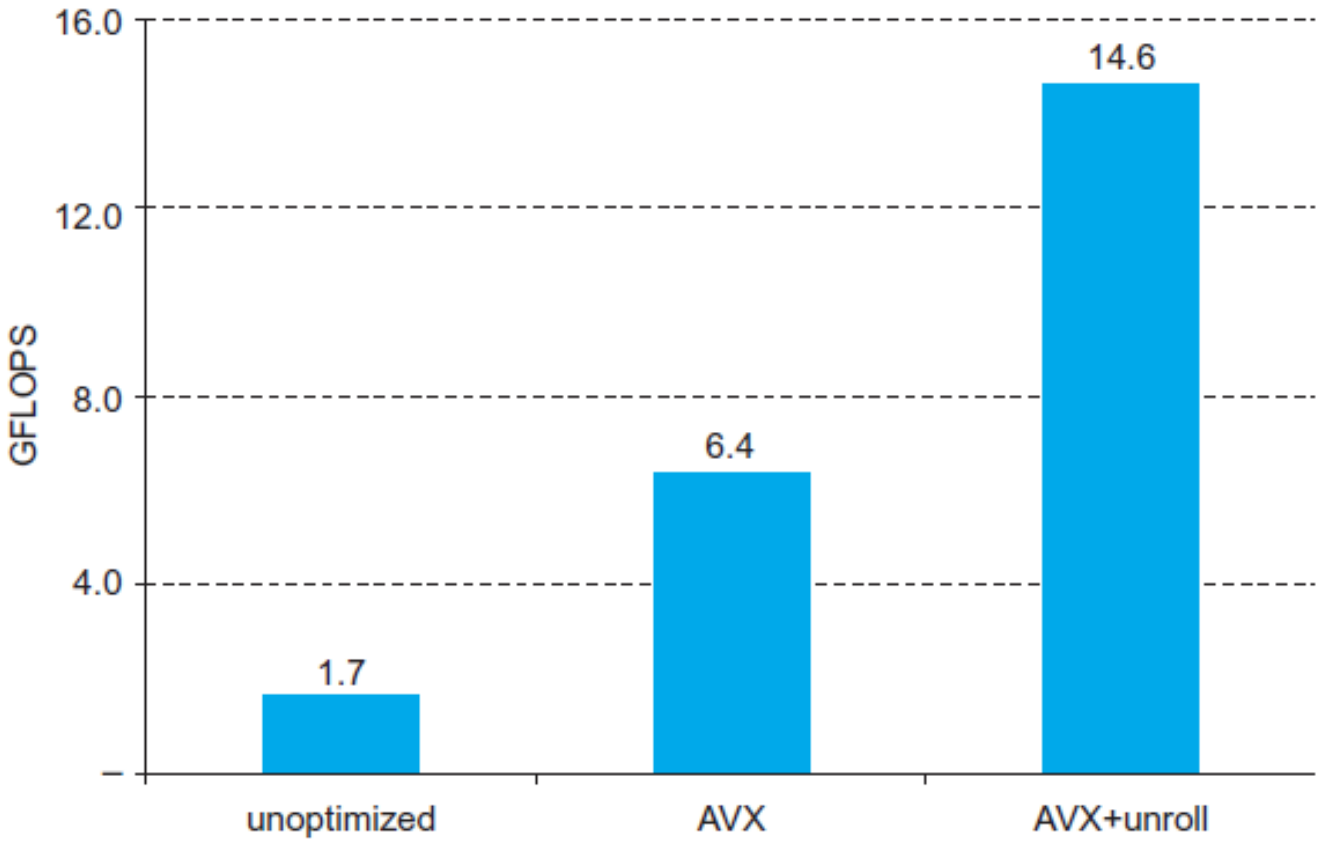
4.12 加速：指令级并行和矩阵乘法

- 矩阵乘法
- 汇编代码

```
1 vmovapd (%r11),%ymm4                  # Load 4 elements of C into %ymm4
2 mov %rbx,%rax                         # register %rax = %rbx
3 xor %ecx,%ecx                         # register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3              # Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2              # Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1              # Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0       # Make 4 copies of B element
8 add $0x8,%rcx # register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5             # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4             # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5        # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3             # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5        # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0        # Parallel mul %ymm1,4 A elements
15 add %r8,%rax                         # register %rax = %rax + %r8
16 cmp %r10,%rcx                        # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2             # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1             # Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>                  # jump if not %r8 != %rax
20 add $0x1,%esi                        # register % esi = % esi + 1
21 vmovapd %ymm4,(%r11)                 # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)             # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)             # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)             # Store %ymm1 into 4 C elements
```

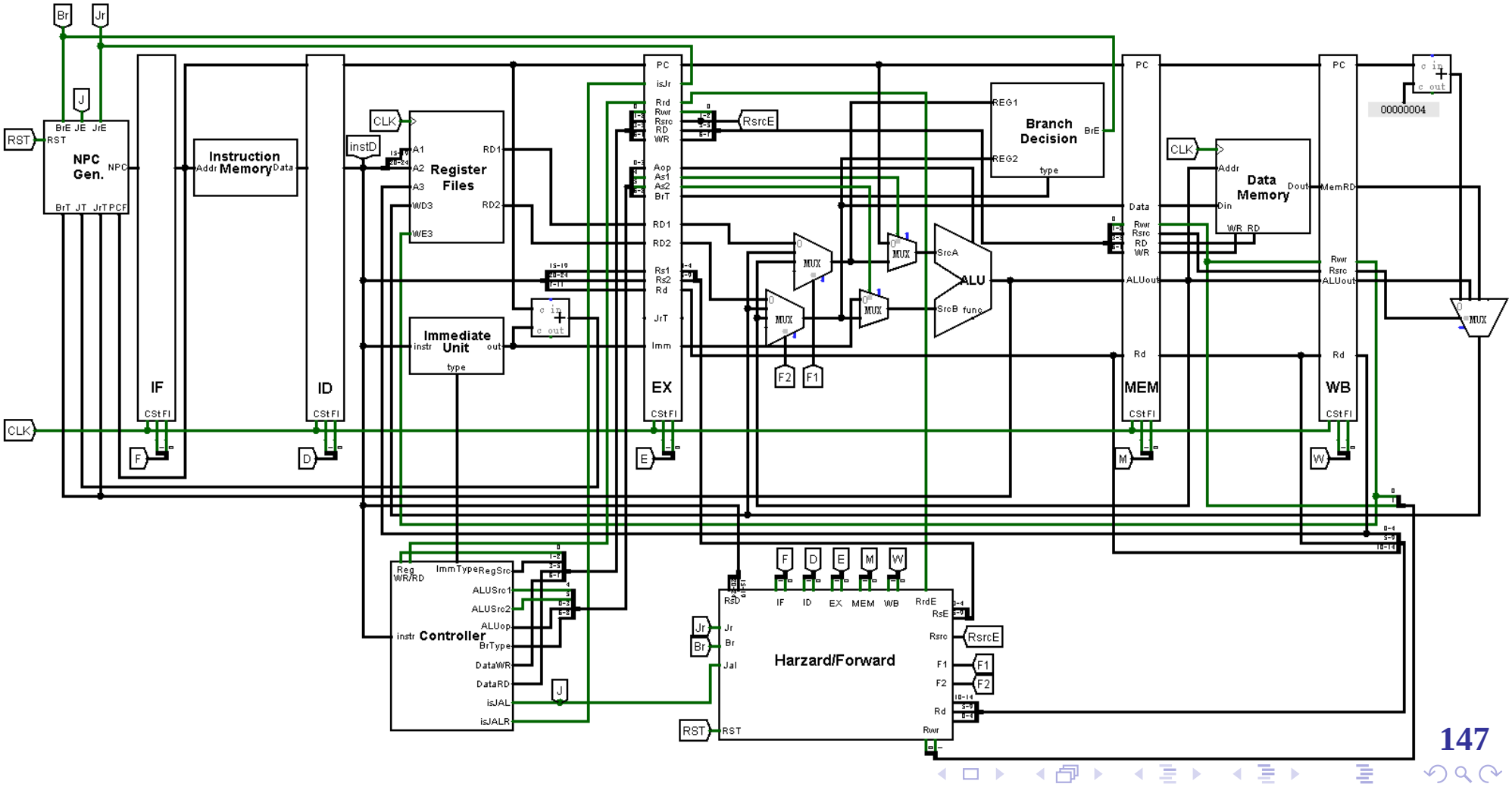
4.12 加速：指令级并行和矩阵乘法

- 性能影响
  - AVX：Intel Advanced Vector Extensions，子字并行
  - unroll：循环展开



4.13 高级专题

- 流水线CPU的Verilog实现（略）
- 流水线CPU的Logisim实现



4.14 谬误与陷阱

- 谬误1：流水线是简单的
- 谬误2：对于流水线等结构设计，可以与工艺无关

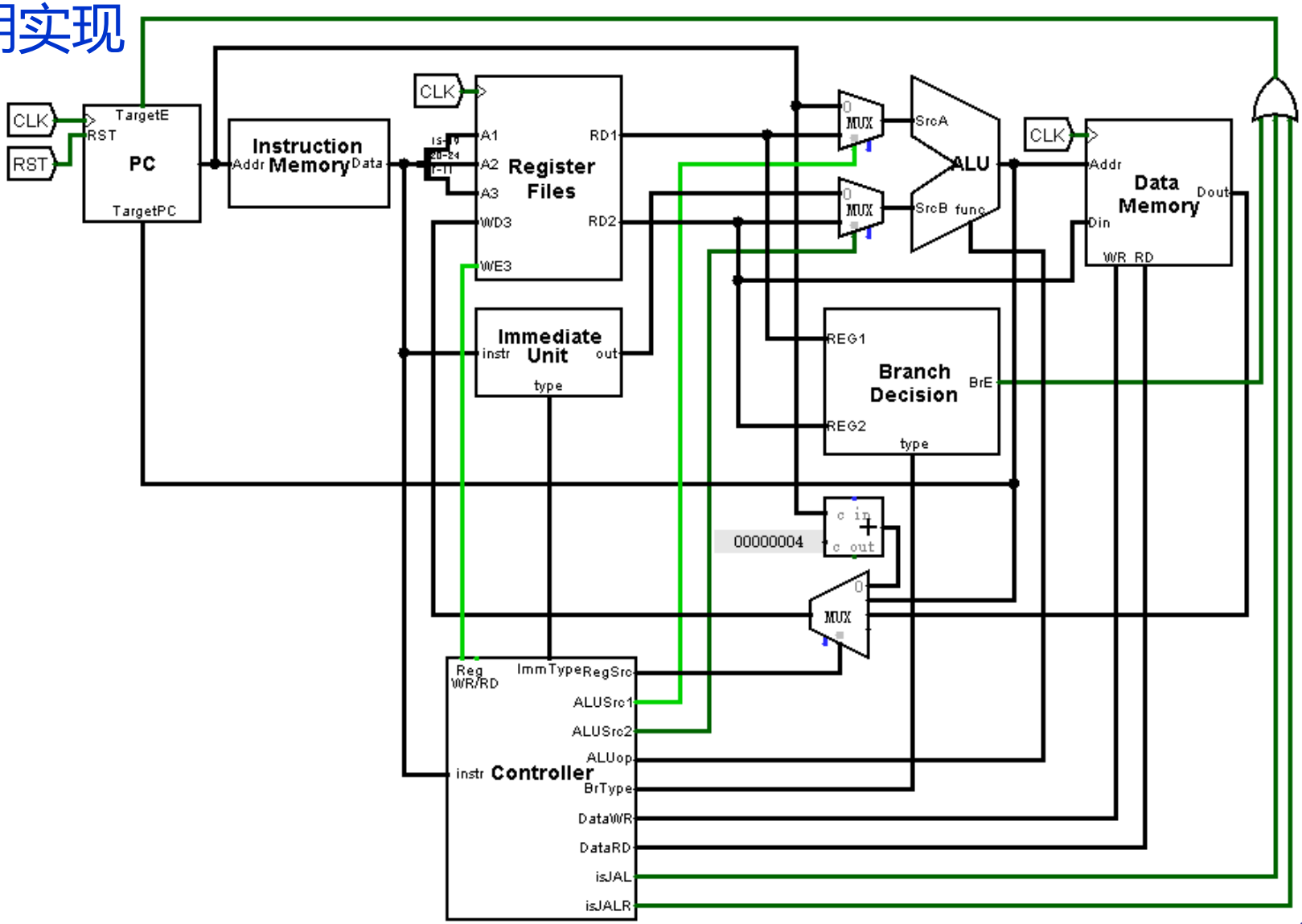
- 陷阱：缺乏对指令系统设计的考虑反过来会影响流水线的实现

4.15 本章小结

- ISA影响数据通路和控制器的设计
- 数据通路和控制器影响ISA设计
- 流水线使用并行性提高指令吞吐量
  - 每秒完成更多指令
  - 每条指令的延迟没有减少
- 冒险
  - 结构
  - 数据
  - 控制
- 多发射和动态调度（ILP）
  - 依赖性限制了并行性的挖掘
  - 复杂性导致功耗墙

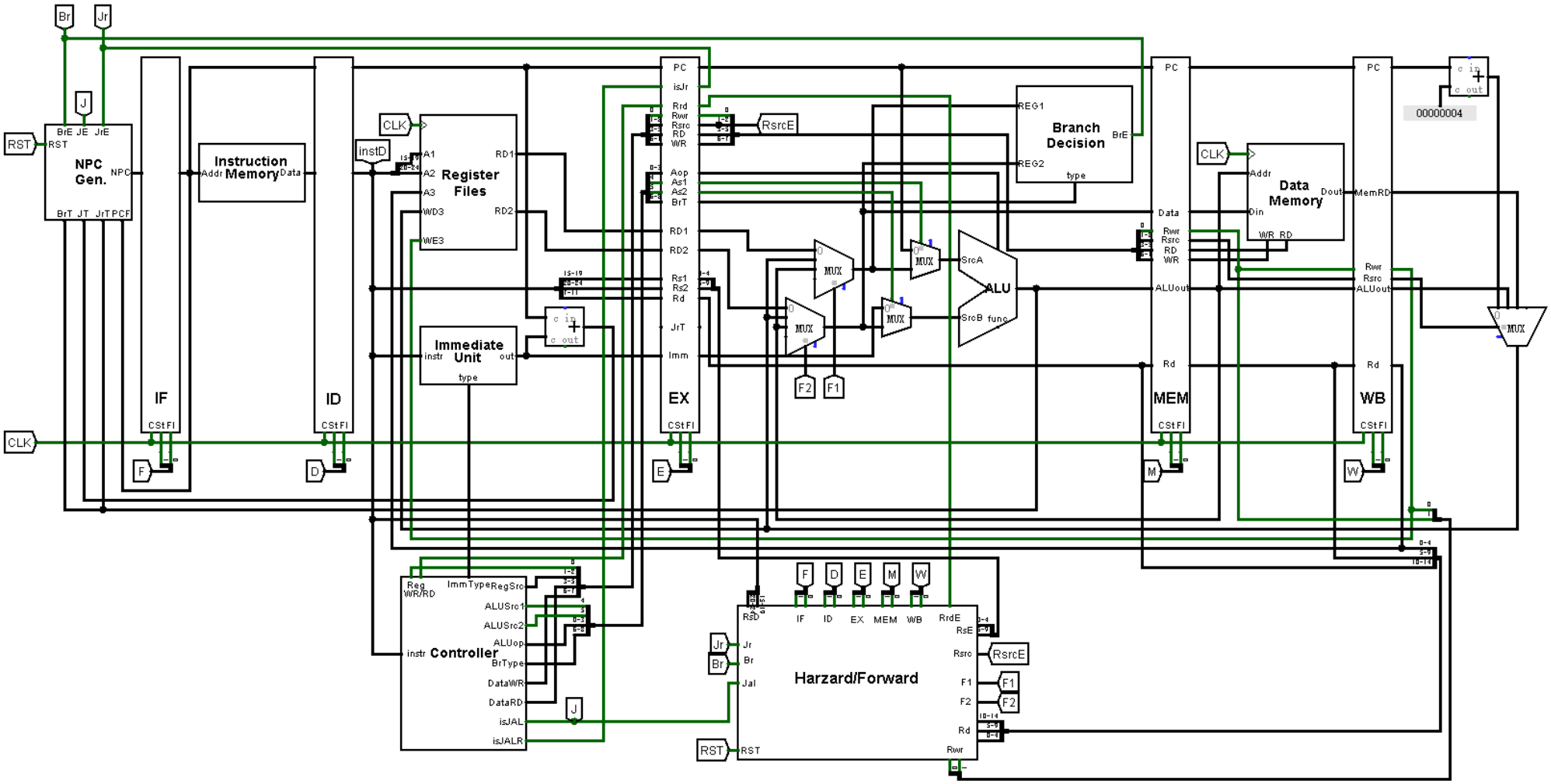
4.15 本章小结

- 单周期实现



4.15 本章小结

- 流水线实现



计算机组成原理
CH5_层次化存储

卢建良
lujl@ustc.edu.cn
2022年春季学期

提纲

- 5.1 引言
- 5.2 存储技术
- 5.3 cache基础
- 5.4 cache的性能评估和改进
- 5.5 可靠的存储器层次
- 5.6 虚拟机
- 5.7 虚拟存储
- 5.8 存储层次结构的一般框架
- 5.9 使用有限状态机控制简单的cache
- 5.10 并行和存储层次结构：cache一致性

提纲

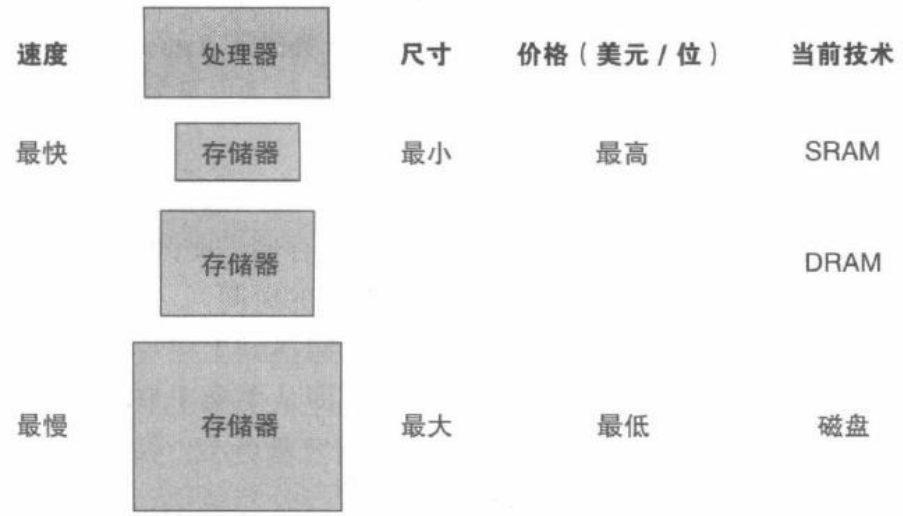
- 5.11 并行与存储层次结构：RAID
- 5.12 高级专题：实现缓存控制器
- 5.13 实例：Cortex-A53和Core i7的存储层次结构
- 5.14 实例：RISC-V系统的其它部分和特殊指令
- 5.15 加速：cache分块和矩阵乘法
- 5.16 谬误与陷阱
- 5.17 本章小结

5.1 引言

- 局部性原理
  - 程序在任何时候都会访问其地址空间中较小的一小部分
  - 时间局部性、空间局部性
  - 类比：图书馆中借书的例子
- 时间局部性
  - 如果某个数据项被访问，那么在不久的将来可能再次被访问
  - 指令：循环指令
  - 数据：循环指令相关的变量
- 空间局部性
  - 如果某个数据项被访问，与它地址相邻的数据项可能很快也将被访问
  - 指令：顺序执行的指令、循环指令
  - 数据：数组

5.1 引言

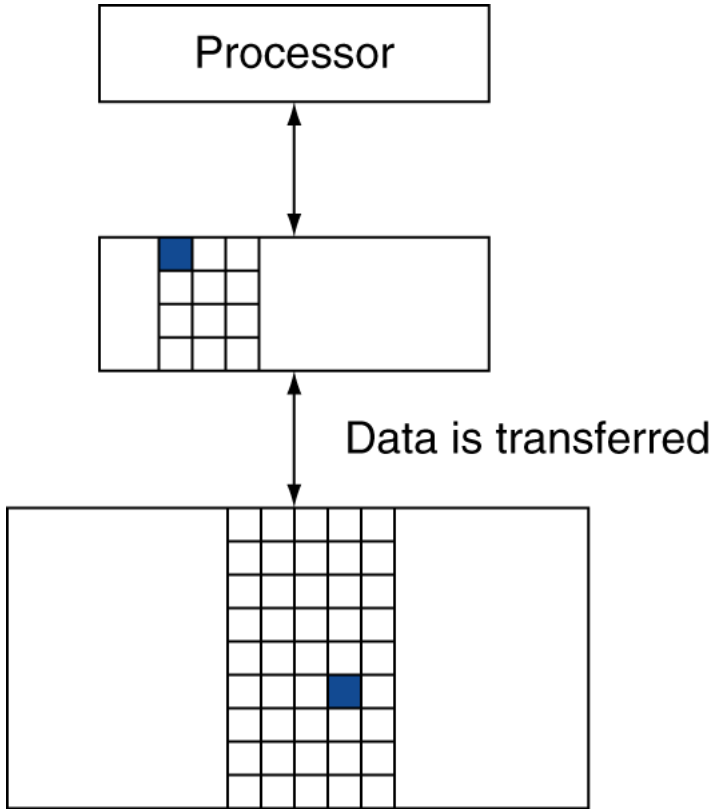
- 局部性原理的应用
  - 层次化的存储器
    - 多级存储采用的结构，与处理器距离越远，存储的容量越大，访问速度越慢
  - 所有信息存放在硬盘上
  - 将近期访问（及其临近）的数据项从硬盘拷贝到DRAM
    - 从硬盘加载到主存/内存
    - 如：操作系统程序、系统服务程序、正在运行的用户程序等
  - 将更加近期访问的数据项从DRAM拷贝到一个更小的SRAM
    - 从内存/主存拷贝到缓存/cache
    - cache离CPU更近
    - 速度更快，但容量更小
    - cache也可能有多个层次



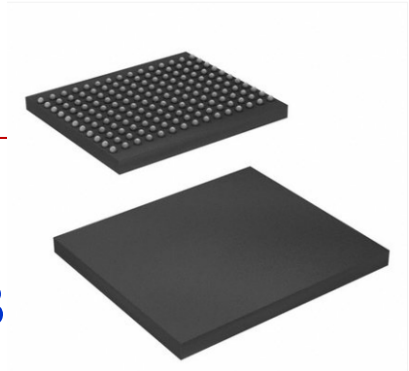
5.1 引言

- 存储器层次结构

- 块/行（block/line）
  - 在相邻层次结构间信息交换的最小单元
  - 也是缓存中存储信息的最小单位
  - 一般会包含多个字：16~256字节
- 命中（hit）
  - 待访问的数据在上层存储单元中
  - 命中率：命中次数/访问次数
- 缺失（miss）
  - 待访问的数据不在上层存储单元中
  - 缺失率：缺失次数/访问次数 = 1-命中率
  - 数据缺失时会从下一层次获取数据

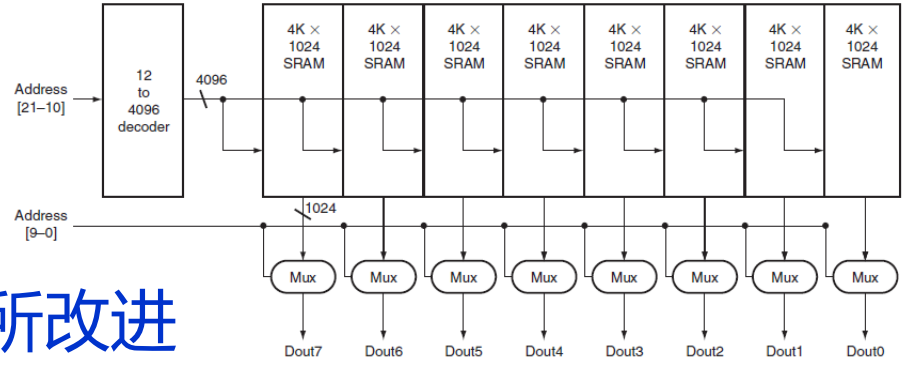


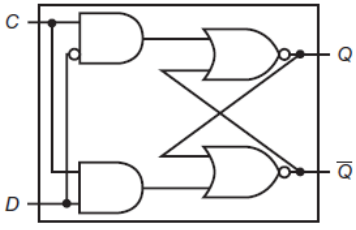
5.2 存储技术

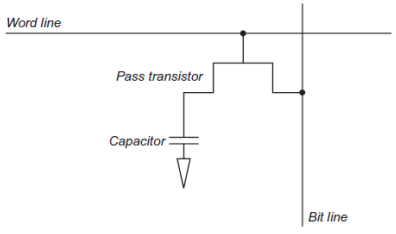


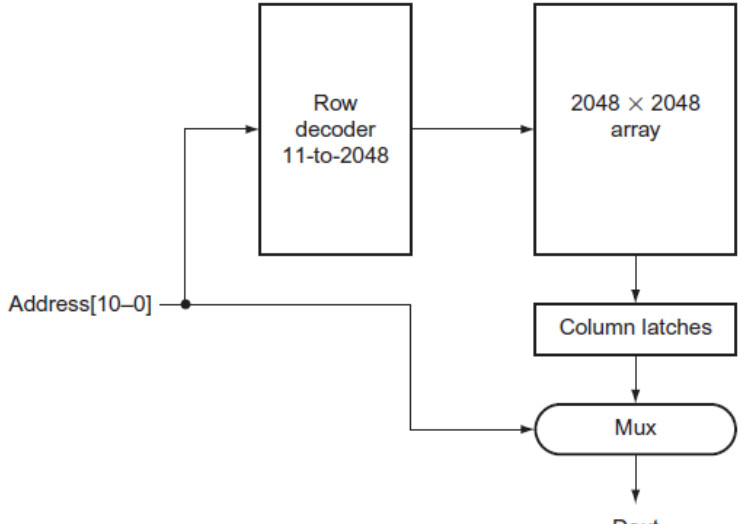
CY7C1360C-166BZC [IC SRAM 9MBIT 166MHZ 165FBGA]
全新原装正品 快速交货速度 批量优价详询
价格 ¥126.10
运费 广东深圳 至 合肥 快递: 23.00
月销量 0 送天猫积分 12
数量 1 件 库存紧张
花呗分期 登录后确认是否享有该服务 什么是花呗分期
¥42.99x3期 (含手续费) ¥21.95x6期 (含手续费) ¥11.28x12期 (含手续费)
立即购买 加入购物车

- 静态RAM（SRAM）
  - 0.5ns~2.5ns， $2000 – $5000/GB
- 动态RAM（DRAM）
  - 50ns~70ns， $20 – $75/GB
- 磁盘
  - 5ms~20ms， $0.20 – $2/GB
- 随着工艺的进步，上述参数会有所改进
- 理想的存储器
  - 访问时间与SRAM相当
  - 容量和每GB成本与磁盘相当



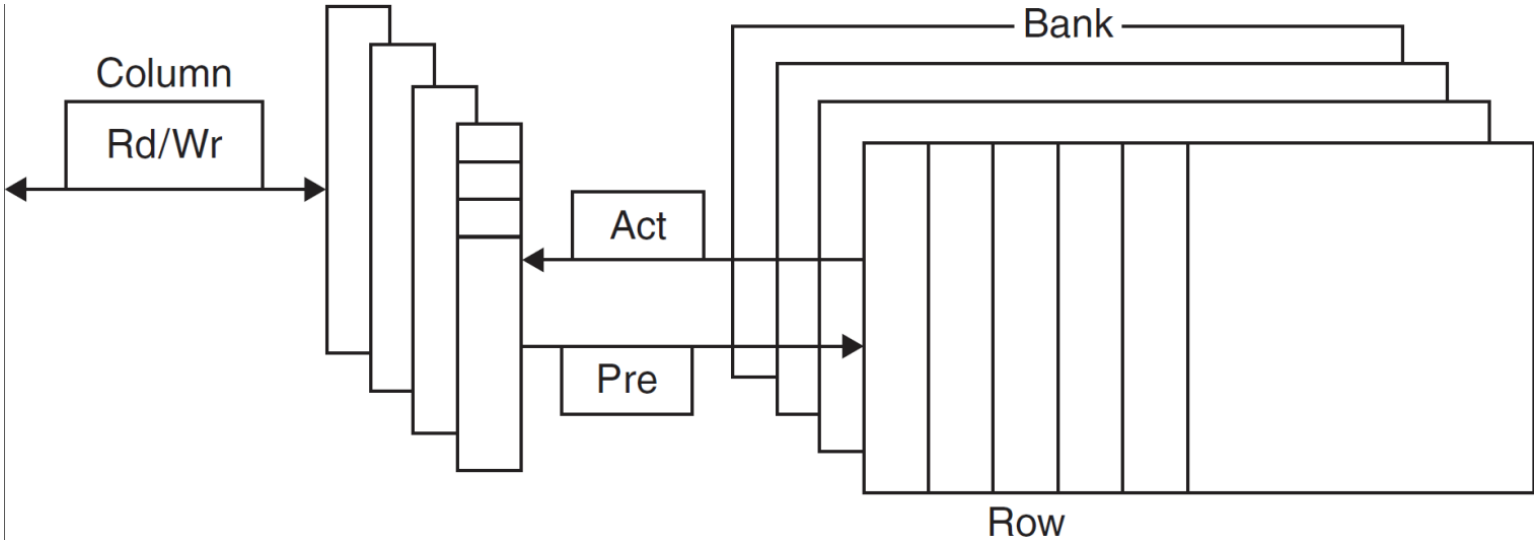






5.2 存储技术

- DRAM技术
  - 以电容充放电的方式存放数据
  - 每bit数据单元使用一个晶体管控制
  - 必须进行周期性的刷新
    - 刷新操作：读取数据内容，然后重新写回
    - 刷新时，以DRAM的“行”作为操作单元



5.2 存储技术

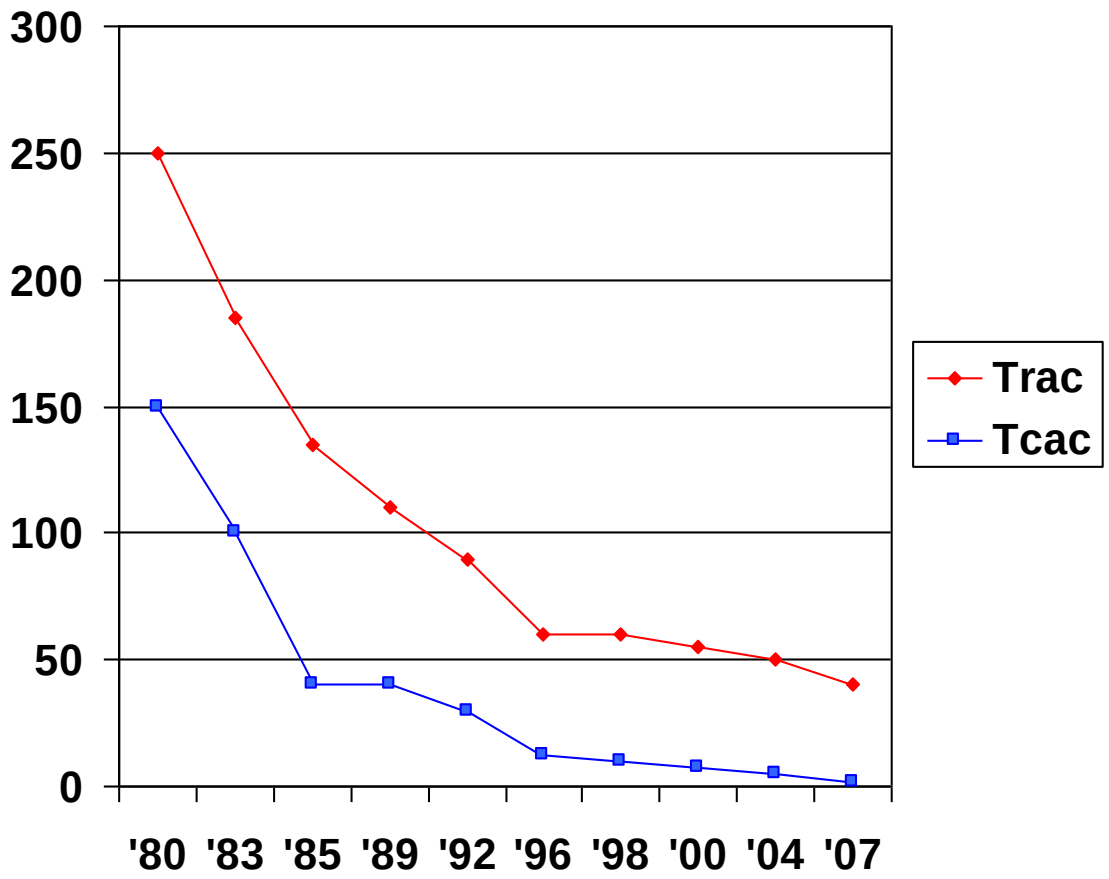
- DRAM的组织结构

- DRAM中的位被组织成矩形阵列
  - DRAM访问整行
  - 突发模式：连续提供一行中的所有字，减少延迟
- 双倍数据速率（DDR）DRAM
  - 在上升和下降时钟边缘传输
- 四倍数据速率（QDR）DRAM
  - 独立的DDR输入和输出

5.2 存储技术

- DRAM技术演化

Year	Capacity	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50



5.2 存储技术

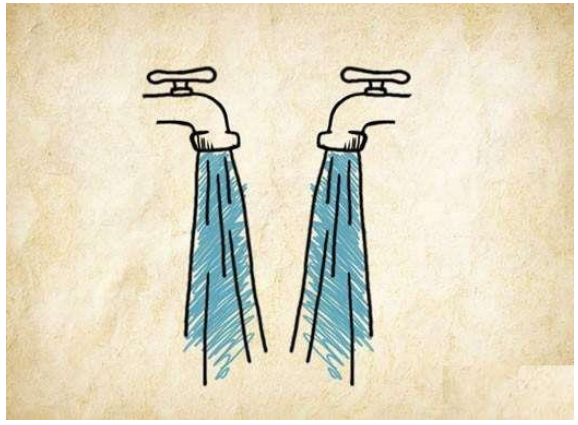
- 影响DRAM性能的相关因素

- 行缓冲区（Row buffer）
  - 允许同时读取和刷新多个字
- 同步动态随机存储器
  - 允许以突发方式连续访问，无需发送每个地址
  - 提高带宽
- DRAM banking
  - 允许同时访问多个DRAM
  - 提高带宽

5.2 存储技术

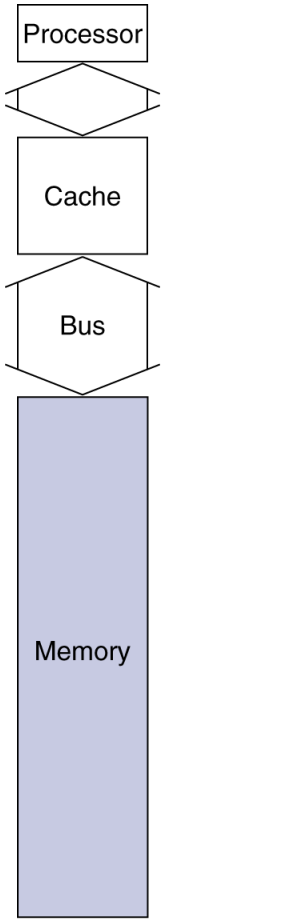
- 从DRAM取数据 vs 从水龙头取水

- DRAM有访问延时，打开水龙头需要耗时
- CPU消耗速度大于DRAM带宽，水的消耗速率大于排水量
- 数据存在局部性（重复利用），水可重复利用
- 提高效率的途径
  - 增加带宽
  - 增加重复使用的效率

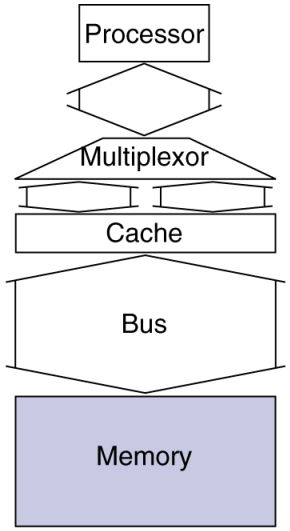


5.2 存储技术

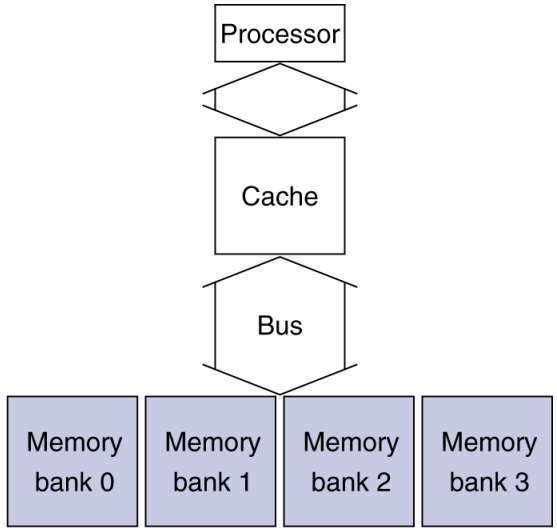
- 增加存储器带宽



a. One-word-wide memory organization



b. Wider memory organization



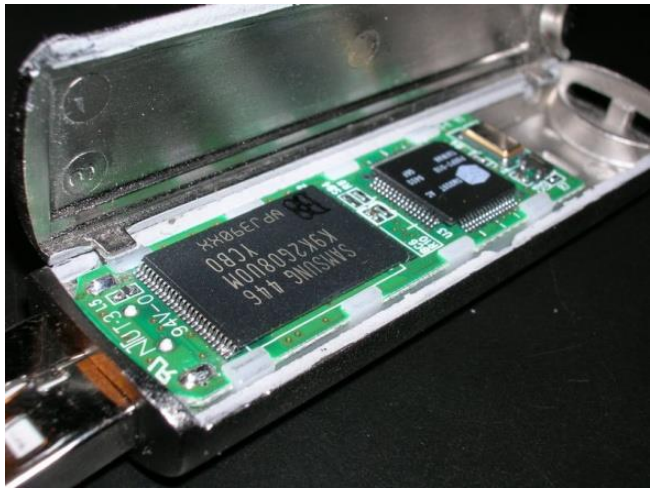
c. Interleaved memory organization

- 将数据位宽扩展到4个字位宽
  - 失效惩罚：1+15+1 = 17个总线时钟周期
  - 带宽：16byte/17 cycles = 0.94 Byte/cycle
- 4个bank的交叉地址访问
  - 失效惩罚： 1 + 15 + 4×1 = 20个总线时钟周期
  - 带宽： 16 bytes / 20 cycles = 0.8 B/cycle

5.2 存储技术

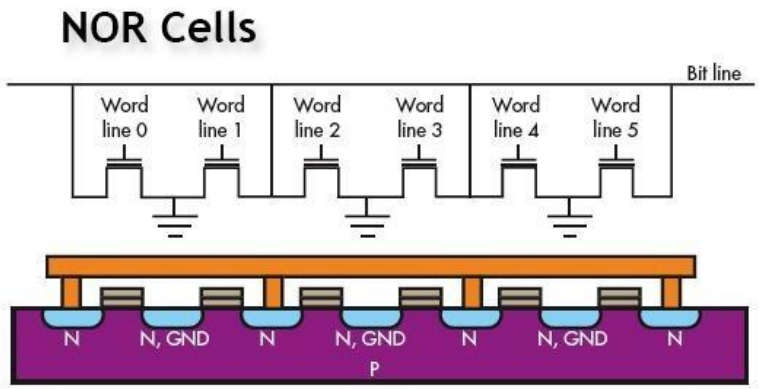
- 闪存(Flash)
  - 闪存是一种新型的EEPROM（电可擦除只读存储器）

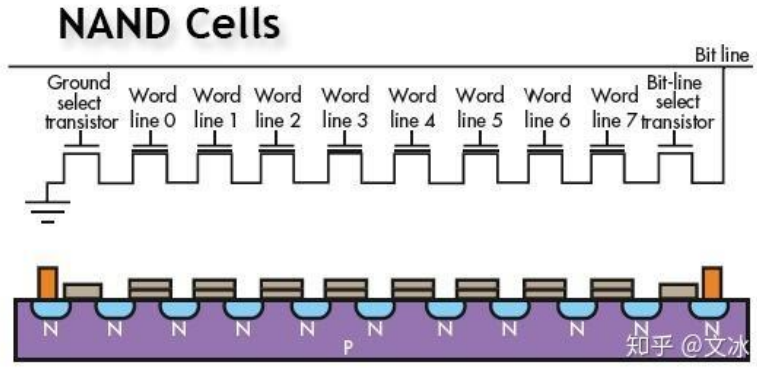
- 闪存是一种非易失性的半导体存储器件
- 比磁盘快100~1000倍
- 体积更小、功耗更低、性能更稳定
- 价格介于磁盘与DRAM之间



5.2 存储技术

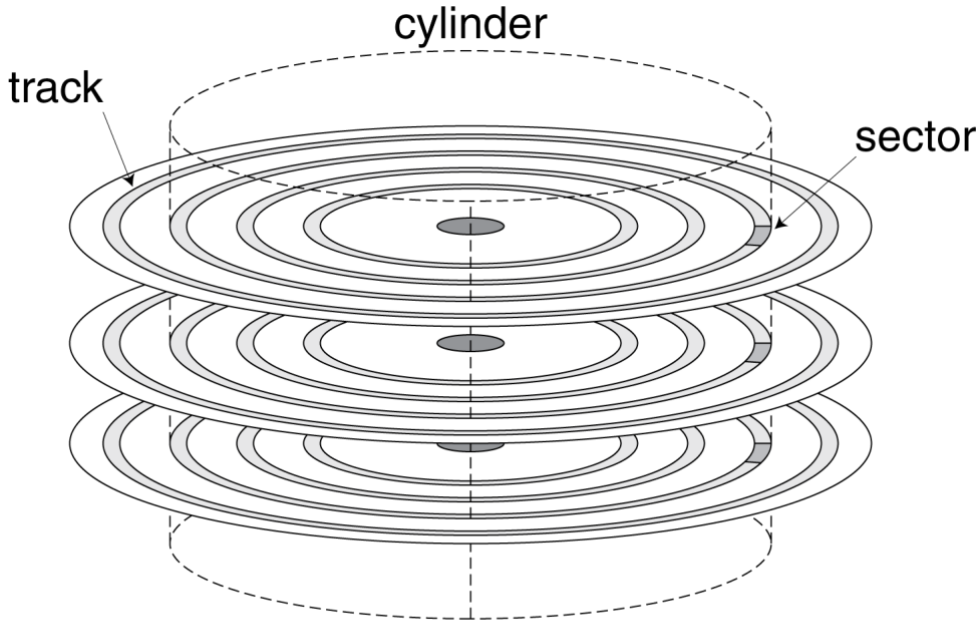
- 闪存种类
  - NOR flash、NAND flash
- NOR flash：bit单元类似NOR门
  - 随机读/写访问
  - 用于嵌入式系统中的指令存储器
- NAND flash：bit单元类似NAND门
  - 更密集（位/面积），但一次阻塞访问
  - 每GB更便宜
  - 用于USB密钥、媒体存储...
- 经过1000次访问后，闪存位会磨损
  - ~~不适合直接替代RAM或磁盘~~（现在已可以）
  - 磨损均衡：将数据重新映射到较少使用的块
  - 可以类比快递柜的使用原理





5.2 存储技术

- 磁盘存储
  - 非易失性，旋转磁存储器



5.2 存储技术

- 磁盘扇区和访问
- 每个磁盘扇区记录的信息
  - 扇区ID
  - 数据（512字节，建议4096字节）
  - 纠错码（ECC）
    - 用于隐藏缺陷和记录错误
  - 同步字段和间隙
- 影响访问磁盘扇区数据的因素
  - 等待其他访问时的排队延迟
  - 寻道：移动头部
  - 旋转潜伏期
  - 数据传输
  - 控制器开销

5.2 存储技术

- 磁盘访问示例
  - 512B扇区，15000rpm，4ms平均寻道时间，100MB/s传输速率，0.2ms控制器开销，空闲磁盘
- 平均读取时间
  - 4ms寻道时间
  - +½/（15000/60）=2ms旋转延迟
  - +512/100MB/s=0.005ms传输时间
  - +0.2ms控制器延迟
  - =6.2ms
- 如果实际平均寻道时间为1ms
  - 平均读取时间=3.2ms

5.2 存储技术

- 磁盘性能问题
- 制造商引用平均寻道时间
  - 基于所有可能的追求
  - 局部性和操作系统调度导致实际平均寻道时间缩短
- 智能磁盘控制器在磁盘上分配物理扇区
  - 向主机提供逻辑扇区接口
  - SCSI、ATA、SATA
- 磁盘驱动器包括缓存
  - 预取扇区以预期访问
  - 避免寻道和旋转延迟

5.3 Cache基础

- Cache实验
  - 存储器：512Byte
  - Cache：128Byte
  - 块大小：16Byte
  - 右侧为地址序列

CPU

- 9′ b0_0000_0000
- 9′ b0_0000_0100
- 9′ b0_0000_1000
- 9′ b0_0000_1100
- 9′ b0_0001_0000
- 9′ b0_0001_0100
- 9′ b0_0001_1000
- 9′ b0_1000_0100
- 9′ b0_1000_1000
- 9′ b1_0000_0000

编号	地址	有效	Tag	块（word3~word0）
0				
1				
2				
3				
4				
5				
6				
7				

编号	地址	Word 3	Word 2	Word 1	Word 0
0	0_0000_0000	3	2	1	0
1	0_0001_0000	7	6	5	4
2	0_0010_0000	11	10	9	8
3	0_0011_0000	15	14	13	12
4	0_0100_0000	19	18	17	16
5	0_0101_0000	23	22	21	20
6	0_0110_0000	27	26	25	24
7	0_0111_0000	31	30	29	28
8	0_1000_0000	35	34	33	32
9	0_1001_0000	39	38	37	36
10	0_1010_0000	...	...	...	...
11	0_1011_0000	...	...	...	...
12	0_1100_0000	...	...	...	...
13	0_1101_0000	...	...	...	...
14	0_1110_0000	...	...	...	...
15	0_1111_0000	...	...	...	...
16	1_0000_0000	...	...	...	...
17	1_0001_0000	...	...	...	...
18	1_0010_0000	...	...	...	...
19	1_0011_0000	...	...	...	...
20	1_0100_0000	...	...	...	...
21	1_0101_0000	...	...	...	...
22	1_0110_0000	...	...	...	...
...	...	...	...	...	...

5.3 Cache基础

- Cache实验
  - 存储器：512Byte
  - Cache：128Byte
  - 块大小：16Byte
  - 右侧为指令序列
    - 9′ b0_0000_0000
    - 9′ b0_0001_0000
    - 9′ b0_0010_0000
    - 9′ b0_0011_0000
    - 9′ b0_0100_0000
    - 9′ b0_0101_0000
    - 9′ b0_1000_0100

CPU

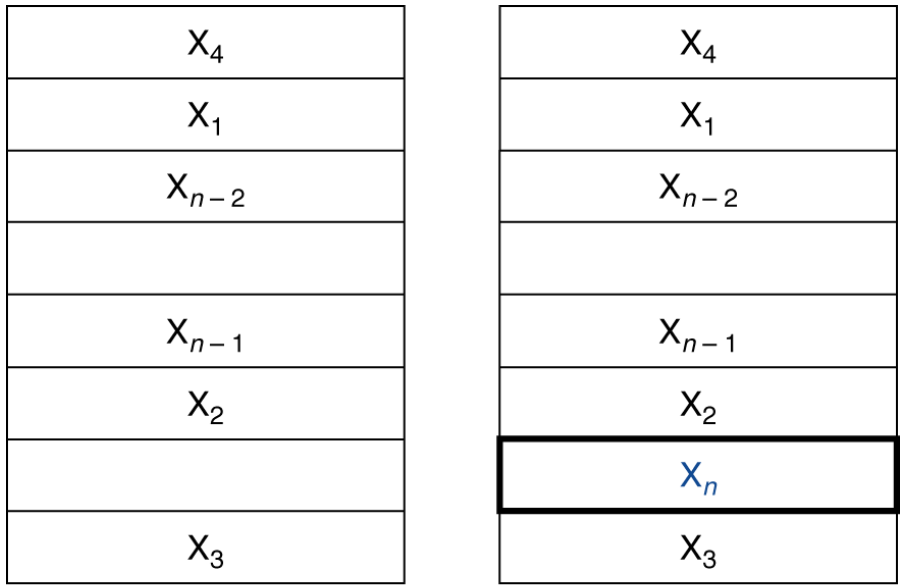
编号	地址	有效	tag	块1	有效	tag	块0
0							
1							
2							
3							

编号	地址	Word 3	Word 2	Word 1	Word 0
0	0_0000_0000	3	2	1	0
1	0_0001_0000	7	6	5	4
2	0_0010_0000	11	10	9	8
3	0_0011_0000	15	14	13	12
4	0_0100_0000	19	18	17	16
5	0_0101_0000	23	22	21	20
6	0_0110_0000	27	26	25	24
7	0_0111_0000	31	30	29	28
8	0_1000_0000	35	34	33	32
9	0_1001_0000	39	38	37	36
10	0_1010_0000	...	...	...	...
11	0_1011_0000	...	...	...	...
12	0_1100_0000	...	...	...	...
13	0_1101_0000	...	...	...	...
14	0_1110_0000	...	...	...	...
15	0_1111_0000	...	...	...	...
16	1_0000_0000	...	...	...	...
17	1_0001_0000	...	...	...	...
18	1_0010_0000	...	...	...	...
19	1_0011_0000	...	...	...	...
20	1_0100_0000	...	...	...	...
21	1_0101_0000	...	...	...	...
22	1_0110_0000	...	...	...	...
...	...	...	...	...	...

5.3 Cache基础

- Cache存储器
  - 一个隐藏或者存储信息的安全场所
  - 在存储器层次结构中离CPU最近的一层

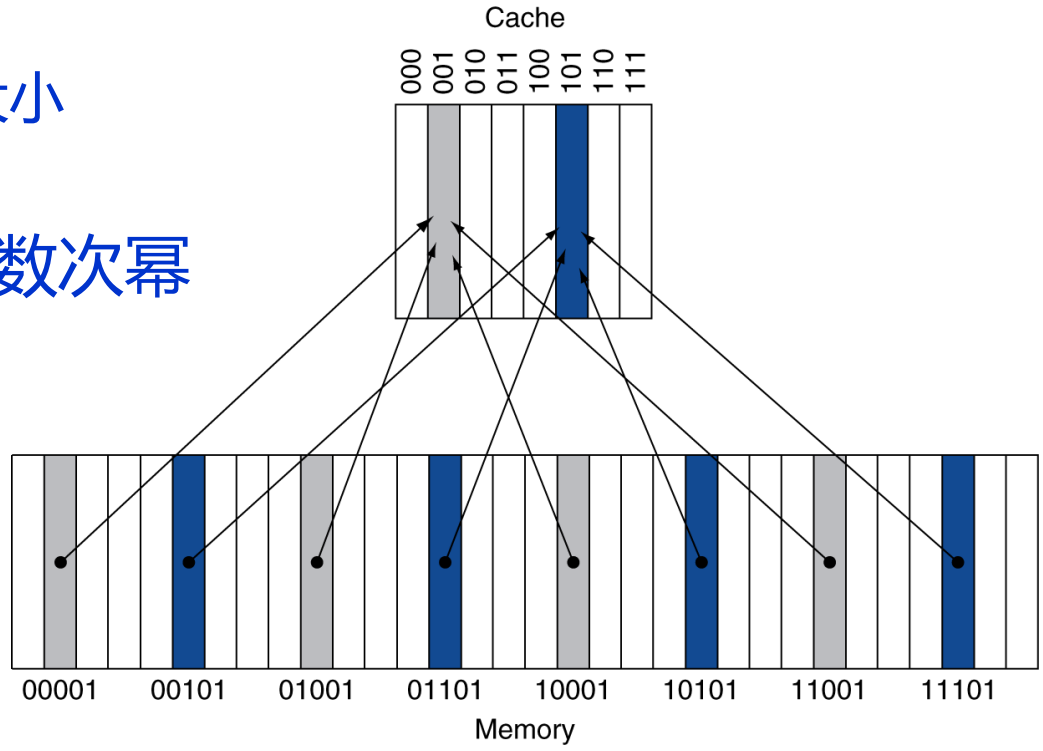
- 示例：给定访问序列$X_1, \ldots, X_{n-1}, X_n$
  - 如何知道访问的数据是否存在
  - 应该到哪里获取数据?



a. Before the reference to X_n　　b. After the reference to X_n

5.3 Cache基础

- 直接映射Cache
  - 一种cache结构，其中每个存储地址都映射到cache中的确定位置
- 映射位置由地址决定
- 直接映射，只有一个选项
  - 块地址 % Cache中的块数量
  - 块地址：内存物理地址 / 块大小
- 块（block）大小是2的整数次幂
- 块数量是2的整数次幂
- 低位地址用于在块内寻址



5.3 Cache基础

- 标签和有效位
- 我们如何知道哪个特定块存储在缓存位置?
  - 存储块地址和数据
  - 实际上，只需要地址的高位字段（高多少位?）
  - 该字段称为标签（Tag）
- 如果某个位置没有数据怎么办?
  - 有效位：1=存在，0=不存在
  - 初始值为0

5.3 Cache基础

- Cache示例
  - 8个block，每个block含1个字，直接映射
  - 初始状态

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

5.3 Cache基础

- Cache示例
  - 8个block，每个block含1个字，直接映射

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

5.3 Cache基础

- Cache示例
  - 8个block，每个block含1个字，直接映射

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

5.3 Cache基础

- Cache示例

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

5.3 Cache基础

- Cache示例

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

5.3 Cache基础

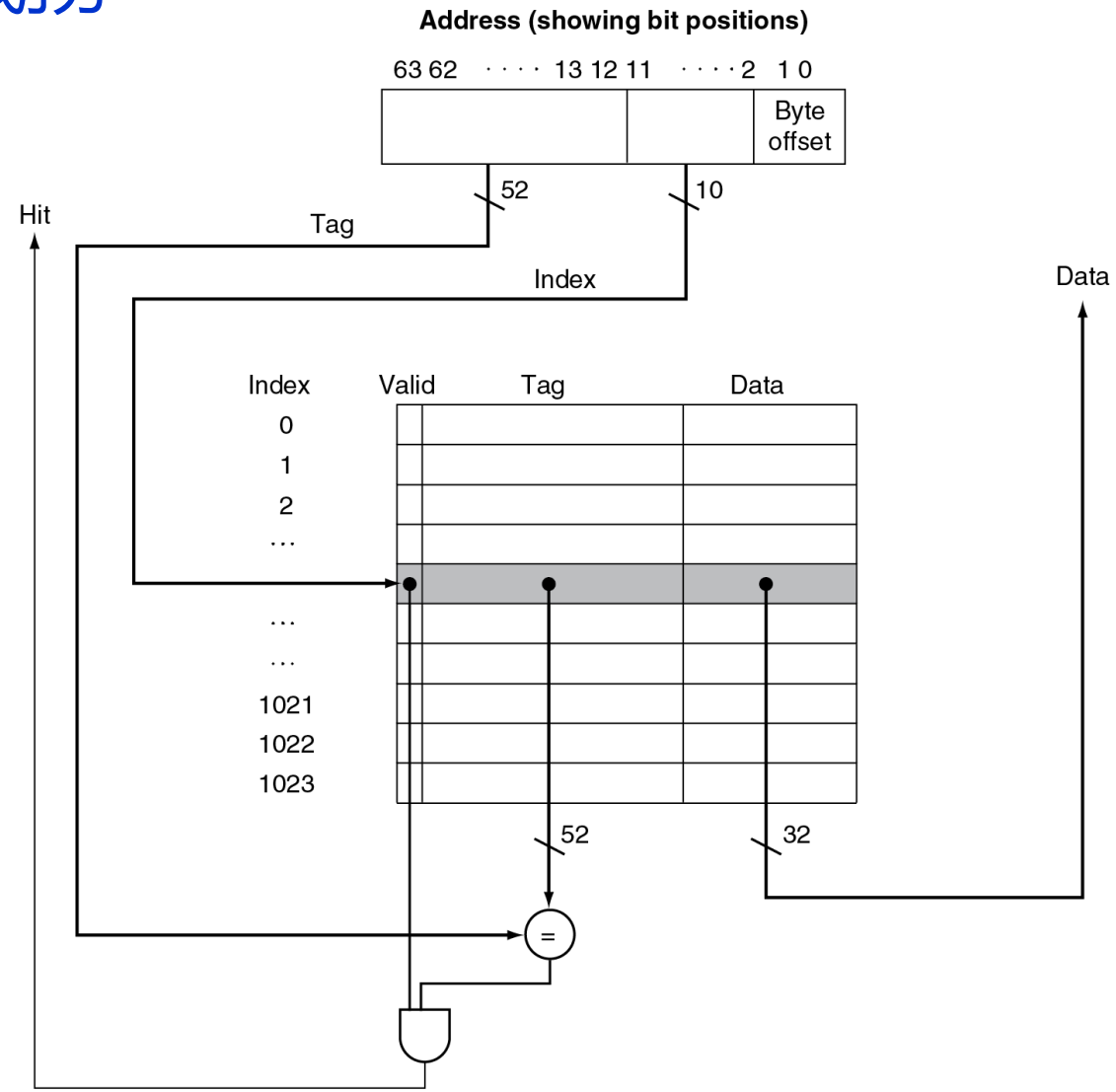
- Cache示例

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

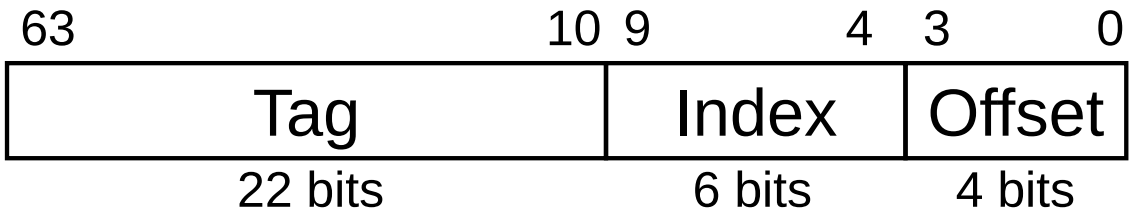
5.3 Cache基础

- 地址字段划分



5.3 Cache基础

- 示例：增加块（block）大小
  - 64个块，16字节/块
    - 地址1200对应于哪个块?
  - [主存的]块地址 = 物理地址 / 块大小　1200/16 = 75
  - [缓存的]块地址 = 主存块地址 % 缓存块数量　75 % 64 = 11

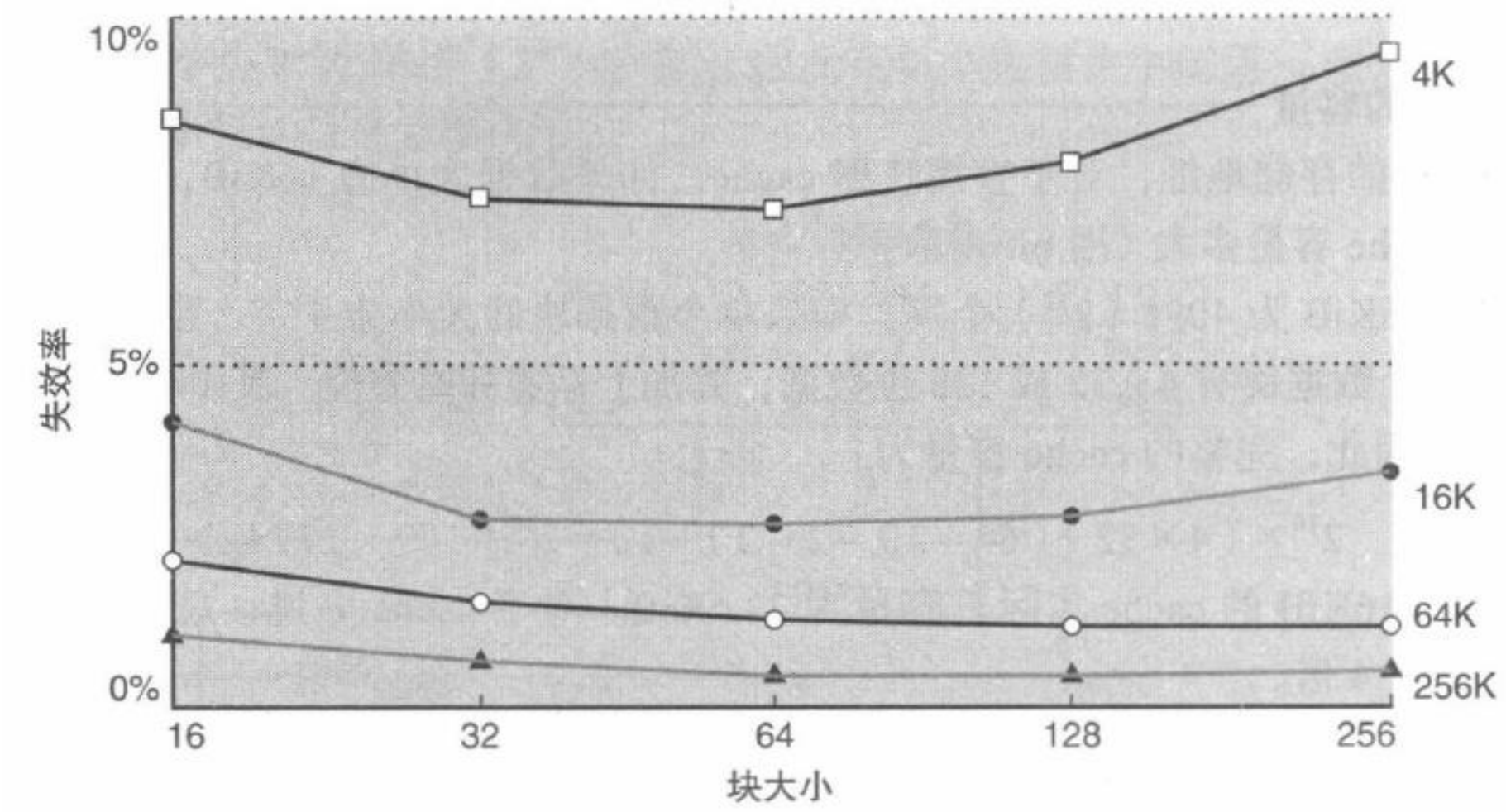


5.3 Cache基础

- 块大小设置的相关考虑
  - 由于空间局部性原理，增加block大小可以减少失效率
  - 对于固定大小的缓存，增加block大小，将导致block数量减少
  - block数量减少，将导致更多竞争，从而增加失效率
- 失效惩罚的相关考虑
  - block越大，数据失效所造成的延时越大
  - 较大的延时将会抵消掉降低失效率所带来的收益
- 对于大数据块所造成延时的改进办法
  - 提早重启技术
    - 只要所需数据返回来就继续执行，无须等数据块中的所有数据都完成传输
  - 关键字先行技术
    - 重新组织存储，让数据块中所需的数据首先从内存传输到cache中
  - 会带来新的问题

5.3 Cache基础

- 块大小与失效率间的关系



5.3 Cache基础

- 缓存失效（Cache Miss）

- Cache命中时，CPU可正常执行
- Cache失效时
  - 阻塞CPU流水线
  - 从下一级存储器（如主存）获取一个数据块
  - 对于指令cache失效
    - 重新启动取指操作
  - 对于数据cache失效
    - 完成数据访问

5.3 Cache基础

- 在数据写入命中时，可以只更新缓存中的块
  - 但这样一来，缓存和内存就会不一致
- 写直达（write-through）
  - 又称写穿透，一种写策略，写操作总是同时更新cache和下一级存储，保证两者之间的数据一致
  - 写直达：更新缓存的同时也更新内存
- 但这会让写操作耗时更长
  - eg：如果基本CPI=1，10%的指令是存储，写入内存需要100个周期
    - 有效CPI=1+0.1×100=11
- 解决方案：写入缓冲区
  - 使用缓冲区保存需要写入内存的数据
  - CPU立即继续
    - 仅当写入缓冲区已满时写入暂停，阻塞CPU

5.3 Cache基础

- 写返回（Write-Back）
  - 一种写策略，处理写操作时，只更新cache中对应数据块的数值，当该数据块被替换时，再将更新后的数据块写入下一级存储
- 写返回策略：在数据写入命中时，只需更新缓存中的块
  - 跟踪每个数据块是否是脏块
  - 什么是脏块？...
- 当更换脏块时
  - 写回存储器
  - 可以使用写缓冲区，将要替换的块存在到缓冲区中

5.3 Cache基础

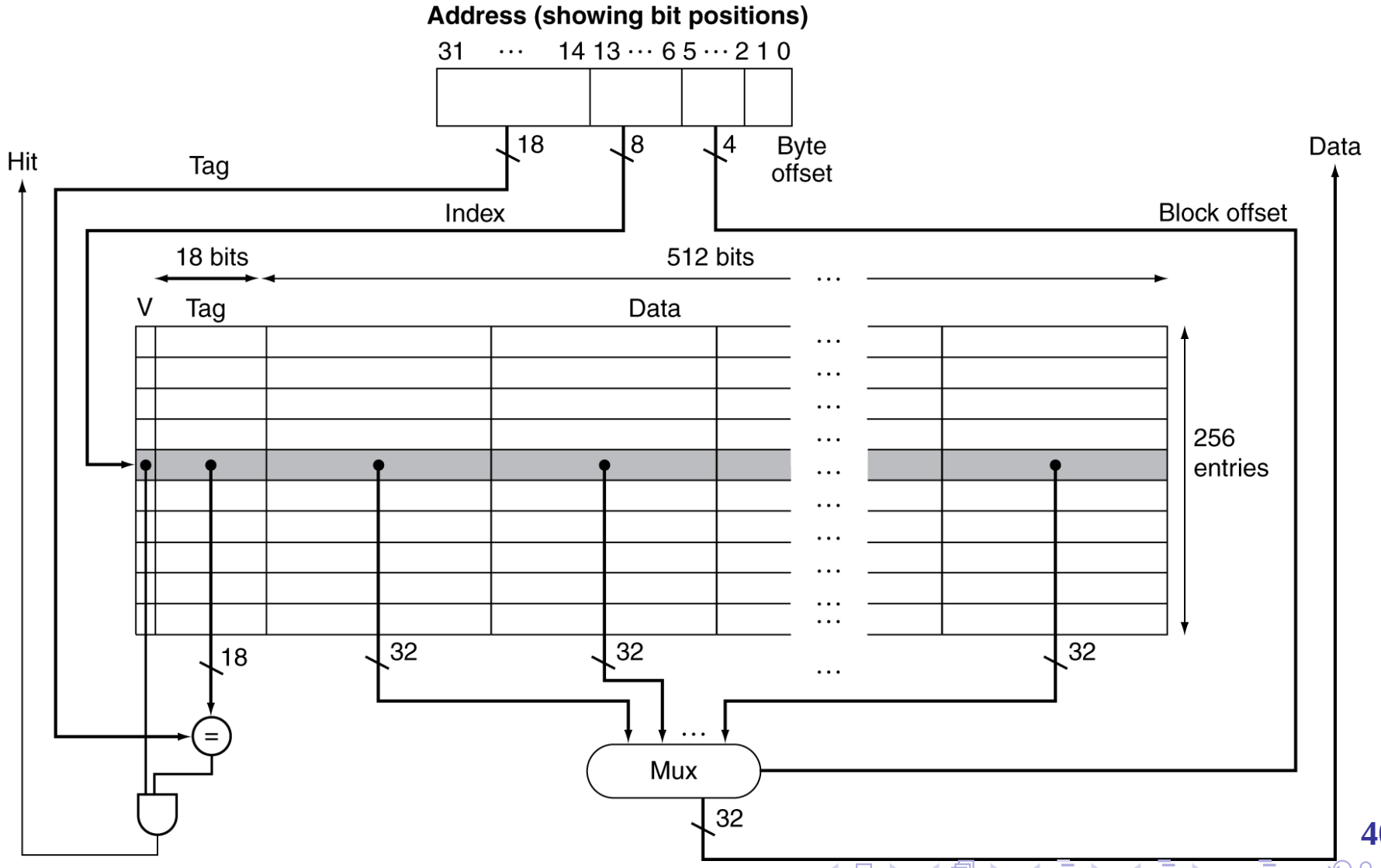
- 写分配（write allocate）
  - 如果发生写失效（write miss），该怎么办?
  - 写穿透策略中，如果要写入的块不在cache中，则在cache中为其分配一个数据块，称为写分配
  - 与其相对应的是写不分配策略（no write allocate），多用于内存的初始化操作
- 对于写直达策略
  - 选项1——未命中时再分配：先读取块
  - 选项2——直接写：不要读取数据块
    - 因为程序通常会在读取前写入整个块（例如初始化）
- 对于写返回策略
  - 通常会读取整个数据块

5.3 Cache基础

- 实例：Intrinsity FastMATH处理器
- 嵌入式MIPS处理器
  - 12级流水线
  - 指令和数据访问均占用一个时钟周期
- 分离的缓存：指令缓存和数据缓存
  - 每个缓存16KB：256个块×16个字/块
  - D-cache：直写或回写
- SPEC CPU2000测评程序失效率
  - 什么是SPEC CPU2000?
    - The Standard Performance Evaluation Corporation (SPEC)
  - I-cache:0.4%
  - D-cache:11.4%
  - 加权平均：3.2%
  - I-cache和D-cache哪种失效率对CPU性能影响更大？为什么？

5.3 Cache基础

- 实例：Intrinsity FastMATH处理器



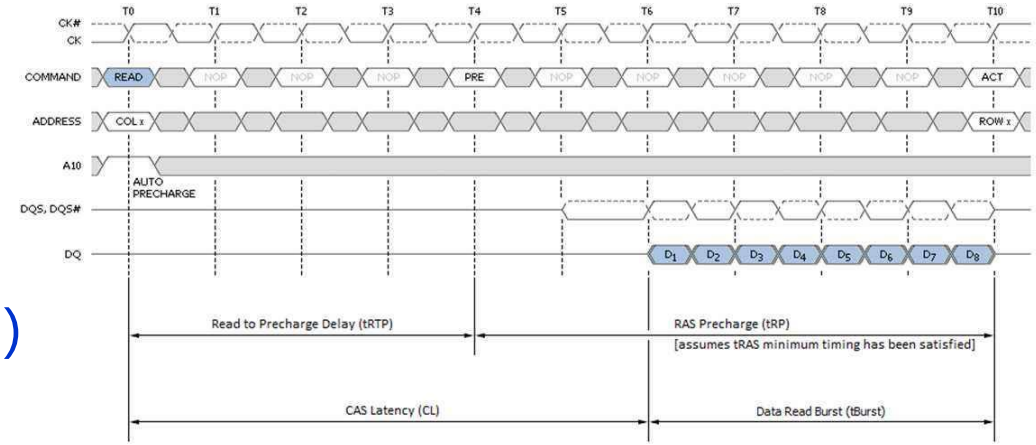
5.3 Cache基础

自我检测 存储系统的速度影响了设计者对于cache数据块容量的判断。针对cache设计者，下面哪一条准则通常是有效的？

1. 存储延迟越短，cache的数据块容量越小。
2. 存储延迟越短，cache的数据块容量越大。
3. 存储带宽越高，cache的数据块容量越小。
4. 存储带宽越高，cache的数据块容量越大。

5.4 Cache的性能评估和改进

- 支持缓存的主存



- 使用DRAM作为主存
  - 固定的数据位宽（例如：32bit）
  - 使用确定位宽的总线进行连接
    - 总线的时钟频率一般都比CPU频率低
- Cache数据块读取示例
  - 发送读地址：1个总线周期
  - DRAM访问：15个总线周期
  - 数据传输：1个总线周期/字
- 对于4word的数据块，1word位宽的DRAM
  - 失效惩罚：1 + 4 * 15 + 4 * 1 = 65个总线周期
  - 带宽：16byte/65总线周期 = 0.25byte/总线周期

5.4 Cache的性能评估和改进

- 计算Cache性能
- CPU时间的组成部分
  - 程序执行周期
    - 包括缓存命中时间
  - 内存暂停周期
    - 主要是缓存未命中
- 通过简化假设：

$$\text{Memory stall cycles}$$

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructio ns}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instructio n}} \times \text{Miss penalty}$$

5.4 Cache的性能评估和改进

- Cache性能示例
  - I-cache未命中率=2%
  - D-cache未命中率=4%
  - 未命中惩罚=100个时钟周期
  - 基本CPI（理想缓存）=2
  - 加载和存储占指令的36%
- 每个指令的未命中周期
  - I-cache:0.02×100=2
  - D-cache:0.36×0.04×100=1.44
- 实际CPI=2+2+1.44=5.44
  - 理想的CPU速度是实际速度的5.44/2=2.72倍

5.4 Cache的性能评估和改进

- 平均访问时间

- 命中时间对性能也很重要
- 平均内存访问时间（AMAT）
  - AMAT=命中时间+未命中率×未命中惩罚
- 实例
  - CPU具有1ns的时钟（1GHz），命中时间=1个周期，未命中惩罚=20个周期，I-cache未命中率=5%
  - AMAT=1+0.05×20=2ns
    - 每个指令2个周期

5.4 Cache的性能评估和改进

- 性能总结

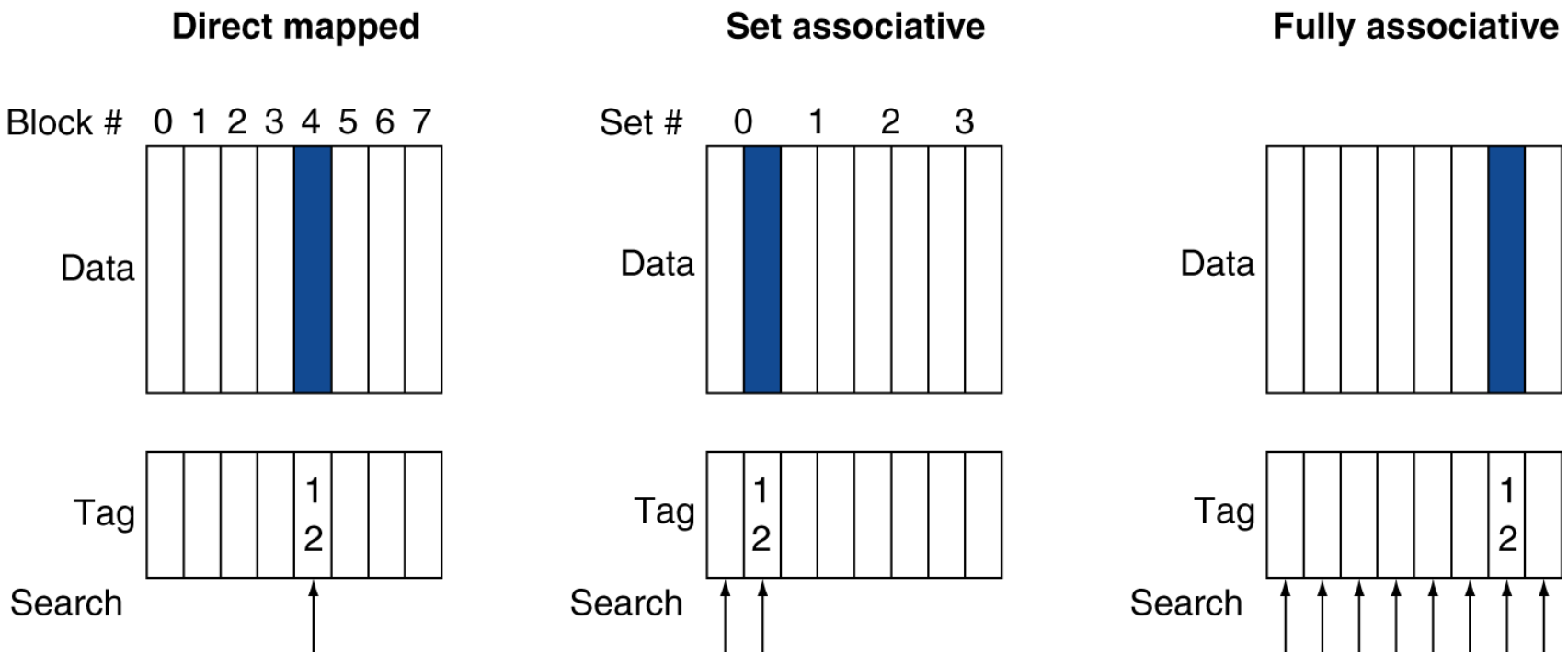
- 当CPU性能提高时
  - 失效惩罚所带来的影响变得更加显著
- 降低基础CPI
  - 花在内存阻塞上的时间比例更大
- 提高时钟频率
  - 内存阻塞会导致占用更多的CPU周期
- 在评估系统性能时不能忽略缓存行为

5.4 Cache的性能评估和改进

- 相连Cache
  - 全相连 Fully associative
  - N路组相连 *n*-way set associative
- 全相联
  - 允许给定的块进入任何缓存项
  - 要求立即搜索所有条目
  - 需要对每个条目使用比较器进行比较（昂贵）
- N路组相连
  - 每组包含n个条目
  - 块号决定了存入cache中的哪个集合
  - （块号）模（#缓存中的集合）
  - 只需搜索给定集合中的所有条目
  - n个比较器（更便宜）

5.4 Cache的性能评估和改进

- 相联Cache示例
  - 直接映射
  - 组相联
  - 全相联



5.4 Cache的性能评估和改进

- 相联Cache的结构
- 以包含8个条目的Cache为例

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

5.4 Cache的性能评估和改进

- 相连示例

- 比较4块缓存
  - 直接映射，2路组相联，全相联
- 块访问顺序：0,8,0,6,8
- 直接映射

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

5.4 Cache的性能评估和改进

- 2路组相联

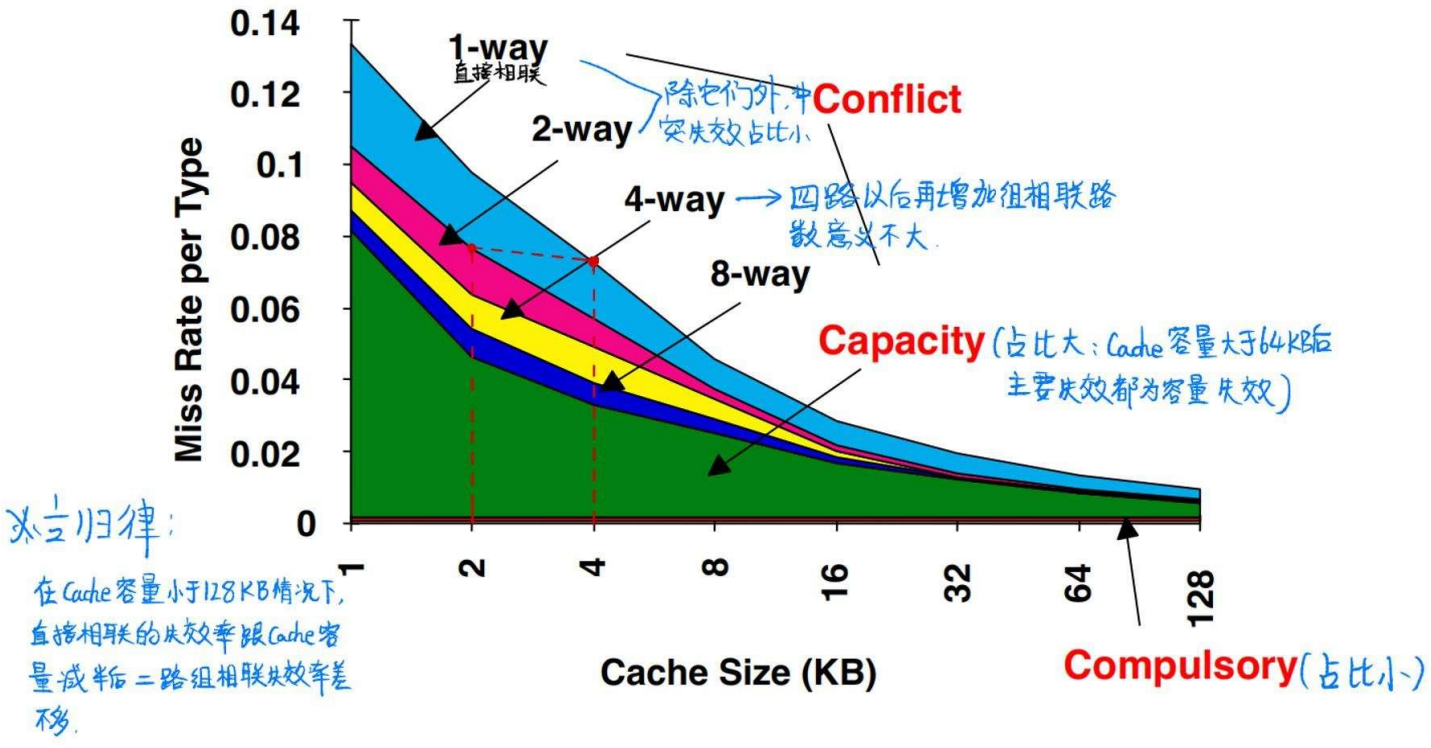
Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- 全相联

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

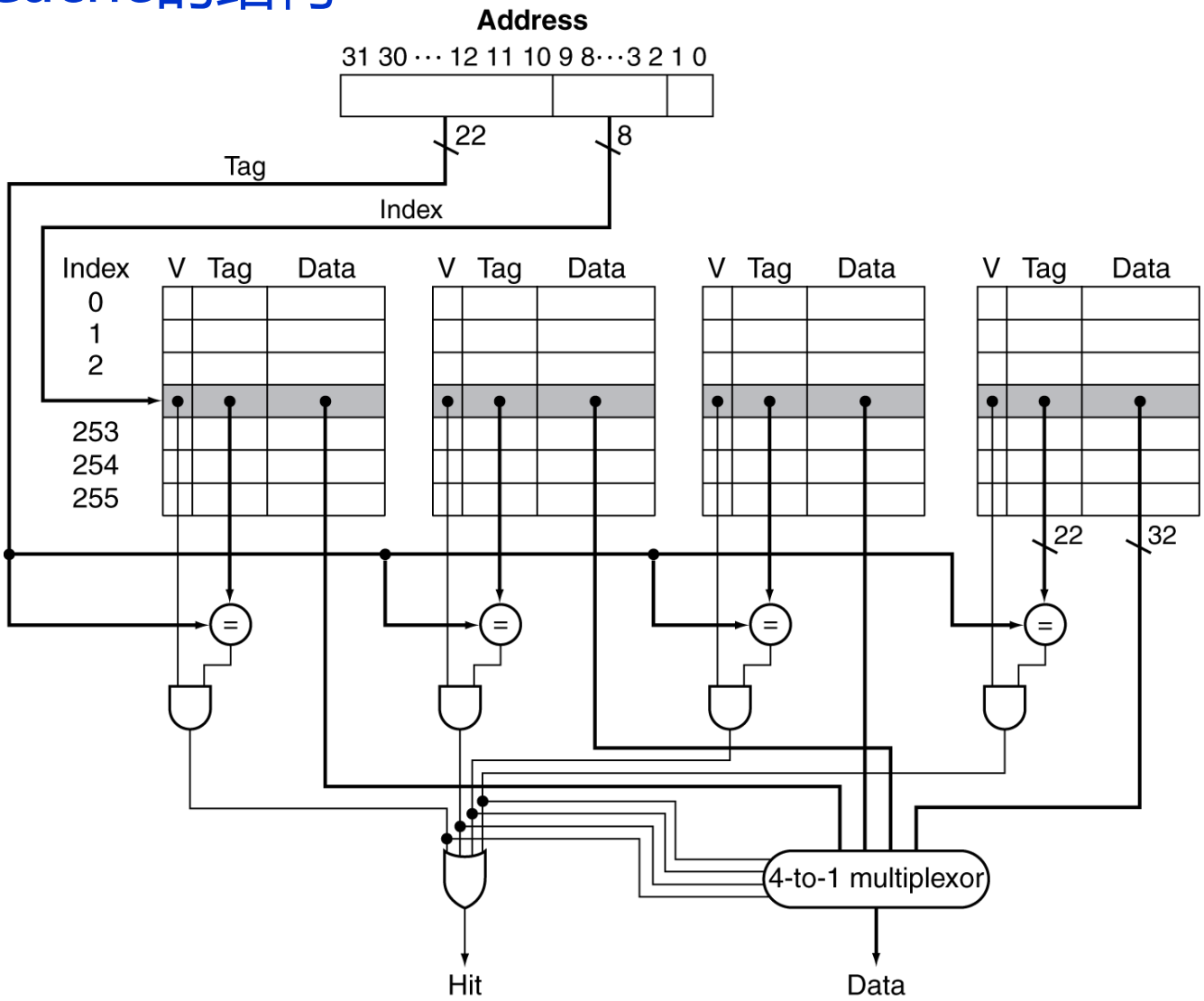
5.4 Cache的性能评估和改进

- 如何确定相联程度
- 增加关联性可以降低失效率
  - 但回报却在递减
- 系统模拟：64KB数据缓存、16字/块、SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%
  - 右图仅供参考



5.4 Cache的性能评估和改进

- 组相联Cache的结构



5.4 Cache的性能评估和改进

- 替换策略

- 直接映射：别无选择
- 组相联
  - 如果有无效条目，请选择无效条目
  - 否则，请在集合中的条目中进行选择
- 最近最少使用（LRU）
  - 选择最长时间未使用的一个
    - 2路简单，4路可控，8路及以上非常困难
- 随机的
  - 在高关联性方面提供与LRU大致相同的性能

5.4 Cache的性能评估和改进

- 多级缓存

- 主缓存（L1 cache）连接到CPU
  - 虽小，但速度很快
- L1 cache miss时访问L2 cache
  - 比L1更大、速度更慢，但仍然很快
- 主存储器服务L-2缓存未命中
- 一些高端系统包括L-3缓存

5.4 Cache的性能评估和改进

- 多级缓存示例
  - CPU基本CPI=1，时钟频率=4GHz（0.25ns/clock）
  - 未命中率/指令=2%
  - 主存储器访问时间=100ns
- 只有主缓存（L-1）
  - 未命中惩罚=100ns/0.25ns=400个时钟周期
  - 有效CPI=1+0.02×400=9
- 现在添加L-2缓存
  - 访问时间=5ns（20个clock）
  - 主存储器的全局未命中率=0.5%
- L-1失效，L-2命中
  - 惩罚=5ns/0.25ns=20个时钟周期
- L-1失效，L-2失效
  - 额外惩罚=400个时钟周期
- CPI=1+0.02×20+0.005×400=3.4
- 性能比=9/3.4=2.6

5.4 Cache的性能评估和改进

- 对于多级缓存的考虑

- 主缓存（L-1）
  - 关注最短的命中时间
- L-2缓存
  - 重点关注低未命中率，以避免主内存访问
  - 命中时间对整体影响较小
- 结果
  - L-1缓存通常比单个缓存小
  - L-1区块大小小于L-2区块大小

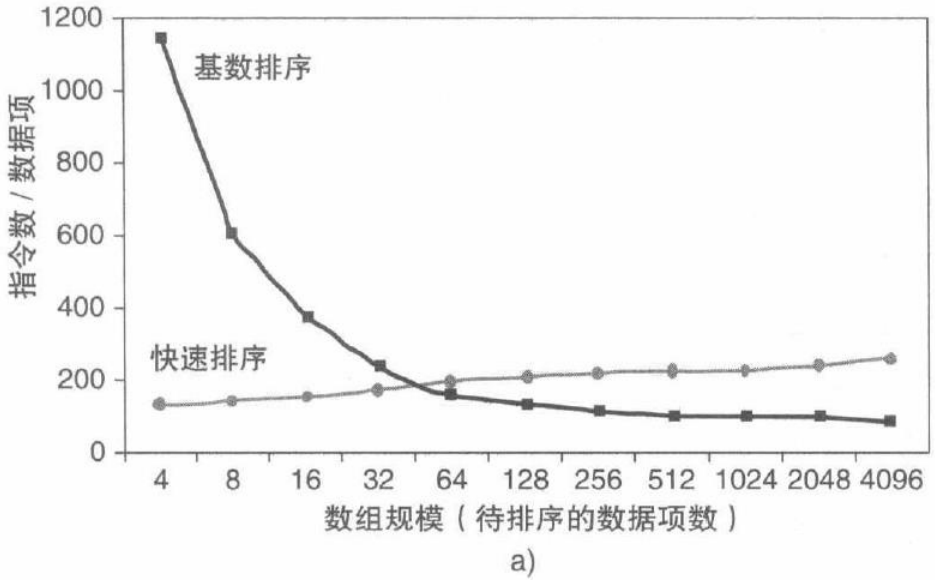
5.4 Cache的性能评估和改进

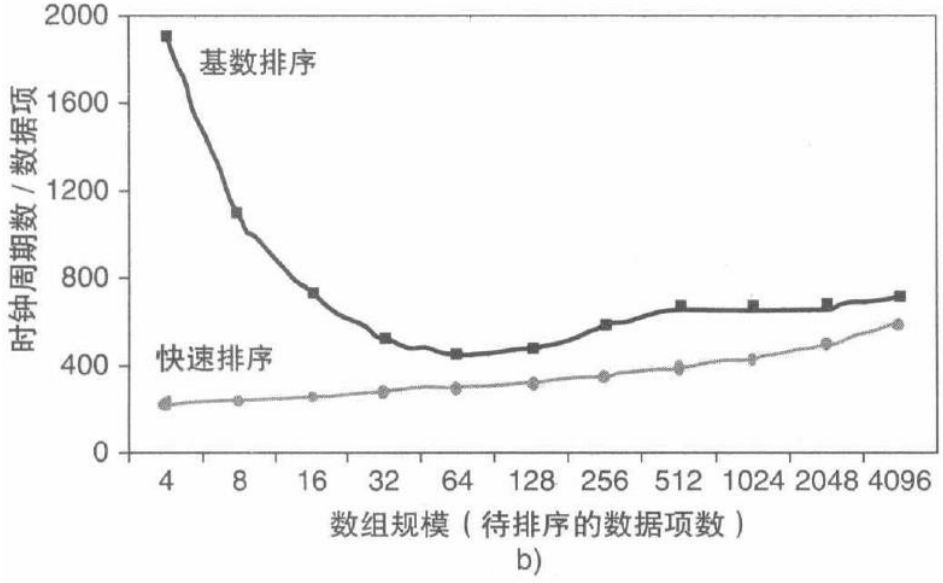
- 高级CPU中对于Cache失效的处理

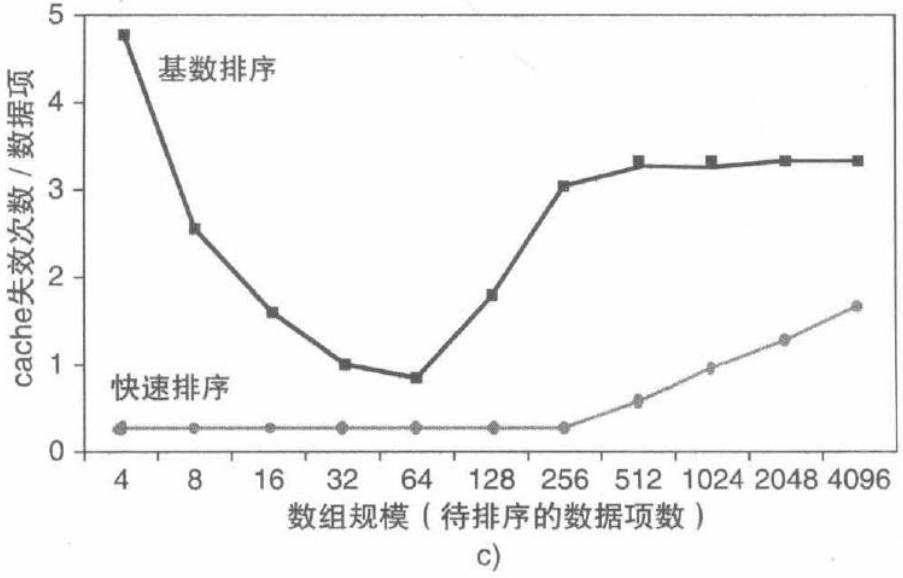
- 支持乱序执行的CPU可以在cache miss时继续执行其它指令
  - 将相关的访存指令保持在load/store单元
  - 与访存相关的指令继续在保留站等待
    - 不相关指令可继续执行
- 未命中的影响取决于程序数据流
  - 更难分析
  - 使用系统模拟

5.4 Cache的性能评估和改进

- 对软件的影响
- 未命中取决于内存访问模式
  - 算法行为
  - 内存访问的编译器优化







5.4 Cache的性能评估和改进

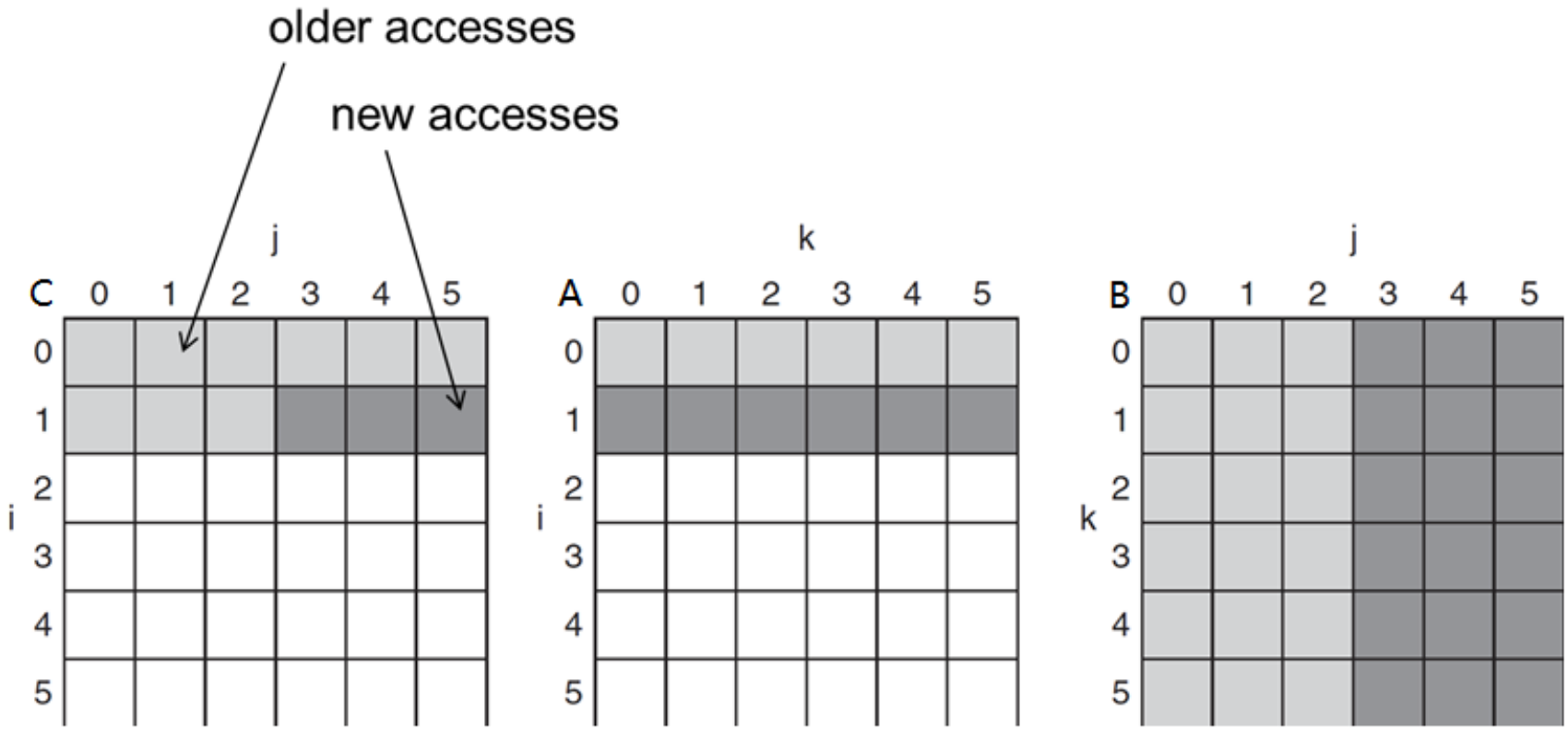
- 通过分块进行软件优化

- 目标：在数据被替换之前最大限度地访问数据
- 考虑DGEMM的内部循环：

```
for (int j = 0; j < n; ++j)
  {
    double cij = C[i+j*n];
    for( int k = 0; k < n; k++ )
      cij += A[i+k*n] * B[k+j*n];
    C[i+j*n] = cij;
  }
```

5.4 Cache的性能评估和改进

- DGEMM访存模式
  - 数组C、A、B的缓存模式



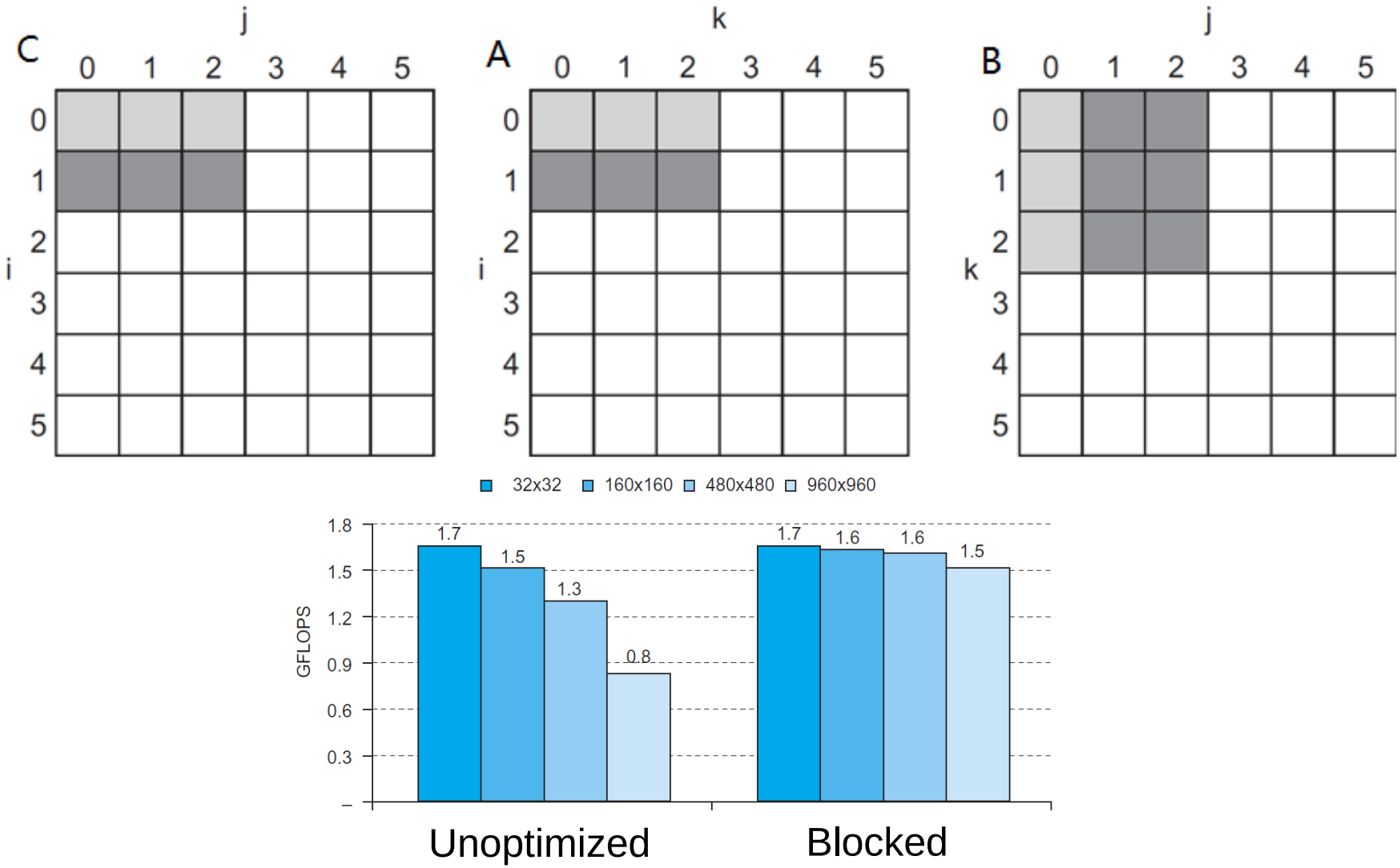
5.4 Cache的性能评估和改进

- DGEMM的cache分块版本

```
1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5  for (int i = si; i < si+BLOCKSIZE; ++i)
6   for (int j = sj; j < sj+BLOCKSIZE; ++j)
7   {
8    double cij = C[i+j*n];/* cij = C[i][j] */
9    for( int k = sk; k < sk+BLOCKSIZE; k++ )
10    cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11   C[i+j*n] = cij;/* C[i][j] = cij */
12  }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17   for ( int si = 0; si < n; si += BLOCKSIZE )
18    for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19     do_block(n, si, sj, sk, A, B, C);
20 }
```

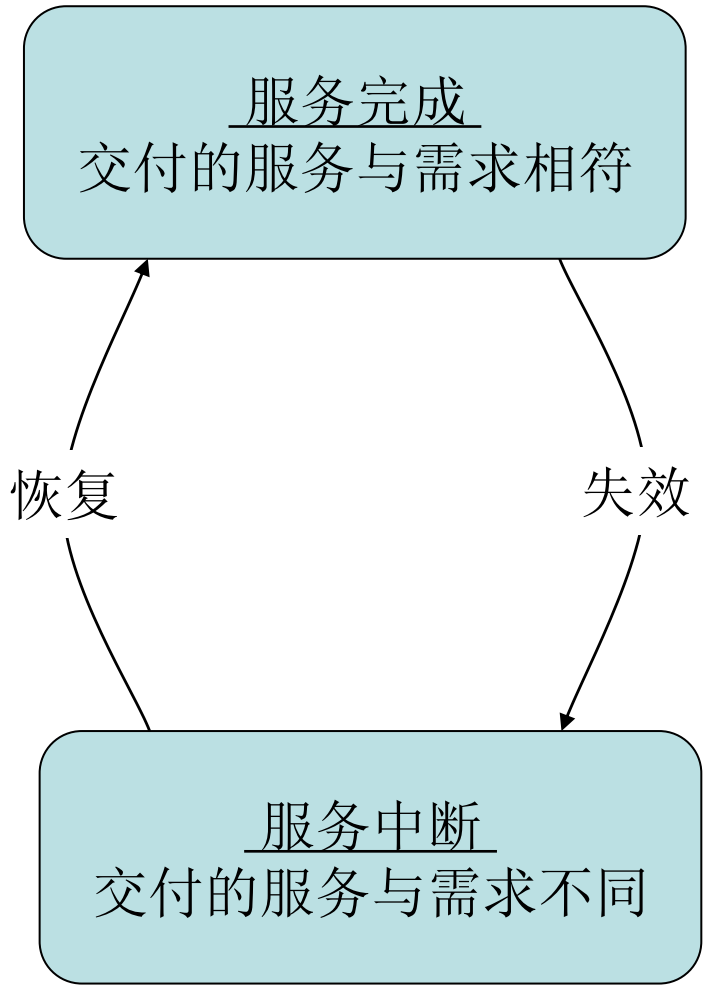
5.4 Cache的性能评估和改进

- DGEMM的cache分块访存模式



5.5 可靠的存储器层次

- 可靠性
  - 是一个系统能够持续提供用户需求的服务的度量，即从参考时刻到失效的时间间隔
  - 度量参数为平均无故障时间，MTTF，Mean Time To Failures



5.5 可靠的存储器层次

- 可靠性度量

- 可靠性：平均失效时间（MTTF）
- 服务中断：平均维修时间（MTTR）
- 平均失效间隔
  - MTBF=MTTF+MTTR
- 可用性=MTTF/（MTTF+MTTR）
- 提高可用性
  - 提高MTTF：故障避免、容错、故障预测
  - 减少MTTR：改进诊断和维修工具和流程

5.5 可靠的存储器层次

- 汉明码
  - SEC码：Single Error Correcting Code
  - DED码：Double Error Detecting Code
- 汉明距离
  - 两个位模式之间不同的位数
- 最小距离等于2时可以提供单位错误检测
  - 例如奇偶校验码
- 最小距离等于3时提供单错误校正，和2位错误检测

5.5 可靠的存储器层次

- 汉明SEC编码

- 为了计算汉明编码
  - 从bit1开始，从左向右进行编号
  - 所有2的整数次幂的位置用来放置校验位
  - 每个校验位对特定的数据位进行校验

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded date bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverate	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

5.5 可靠的存储器层次

- 汉明SEC解码
  - 所有校验位所构成的数据的数值代表了出错的bit位
  - 使用编码过程中的编号
  - 例如：
    - 校验值=0000，表明没有出错
    - 校验值=1010，表明bit10发生了翻转

5.5 可靠的存储器层次

- SEC/DEC编码
- 为整个字添加额外的奇偶校验位-p_n
- 将汉明距离设为4
- 解码：
  - 设H=SEC校验位数
    - H偶数，p_n偶数，无错误
    - H奇数、pn奇数、可纠正的单bit错误
    - H偶数，pn奇数，pn位错误
    - 出现H奇数、pn偶数、双错误
- 注：ECC DRAM使用SEC/DEC，64位数据，8位校验

5.6 虚拟机

- 主机模拟访客操作系统和机器资源
  - 改进了对多个访客账户的隔离
  - 避免安全性和可靠性问题
  - 目标在于共享资源
- 虚拟化对性能有一定影响
  - 适用于现代高性能计算机
- 例子
  - IBM VM/370（1970年代的技术！）
  - VMWare
  - 微软虚拟机

5.6 虚拟机

- 虚拟机监视器（VMM）

- 将虚拟资源映射到物理资源
  - 内存、I/O设备、CPU
- 访客代码以用户模式在本机上运行
  - 根据特权指令和对受保护资源的访问设置VMM陷阱
- 访客操作系统可能不同于主机操作系统
- VMM处理真实的I/O设备
  - 为访客模拟通用虚拟I/O设备

5.6 虚拟机

- 示例：定时器虚拟化

- 在本地机器中，定时中断发生时
  - 操作系统暂停当前进程，处理中断，选择并恢复下一个进程
- 使用虚拟机监视器
  - VMM挂起当前VM，处理中断，选择并恢复下一个VM
- 如果虚拟机需要定时器中断
  - VMM模拟虚拟计时器
  - 当物理计时器中断发生时，模拟VM的中断

5.6 虚拟机

- ISA支持

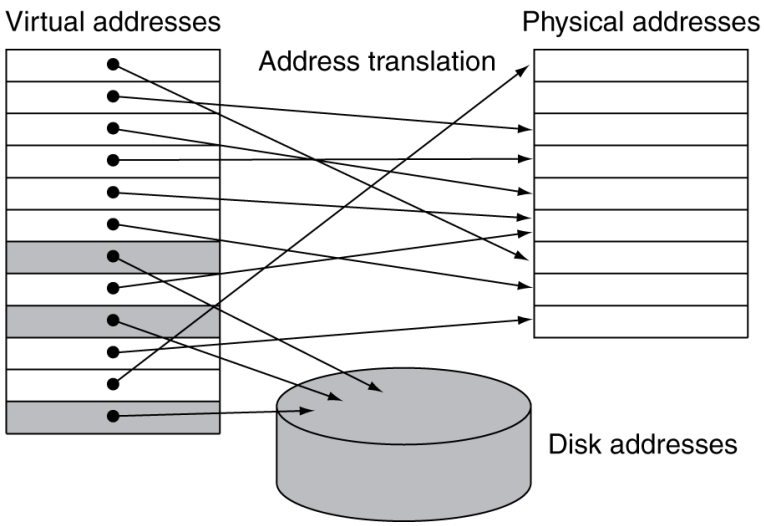
- 支持用户模式和系统模式
- 特权指令仅在系统模式下可用
  - 如果在用户模式下执行，将陷入到系统模式
- 所有物理资源只能使用特权指令访问
  - 包括页表、中断控制、I/O寄存器
- 虚拟化支持的复兴
  - 当前的ISA（如x86）正在调整

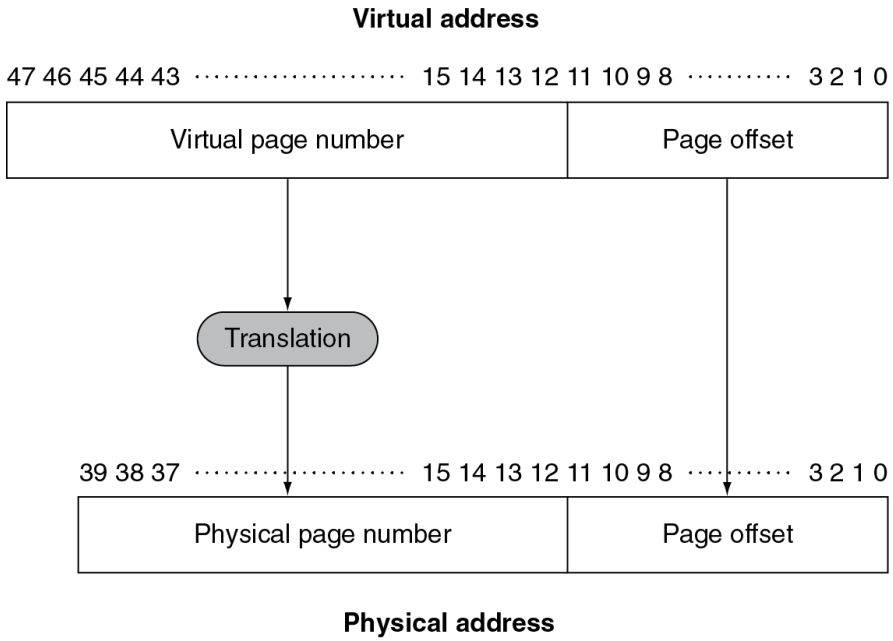
5.7 虚拟存储

- 虚拟存储
  - 一种将主存看作辅助存储的cache的技术
  - 由CPU硬件和操作系统（OS）共同管理
  - VM中的“块”称为页
  - VM中将“未命中”称为缺页失效
- 程序共享主存
  - 每个程序都有一个私有的虚拟地址空间，其中包含其常用的代码和数据
  - 不受其他程序的影响
- CPU和操作系统将虚拟地址转换为物理地址
  - 虚拟地址：虚拟空间的地址，当访问内存时，需要通过地址映射转换为物理地址
  - 物理地址：主存的地址
  - 地址转换：也称地址映射，将虚拟地址转换为物理地址的过程

5.7 虚拟存储

- 地址转换
  - 页大小固定，一般为4K





5.7 虚拟存储

- 缺页失效代价

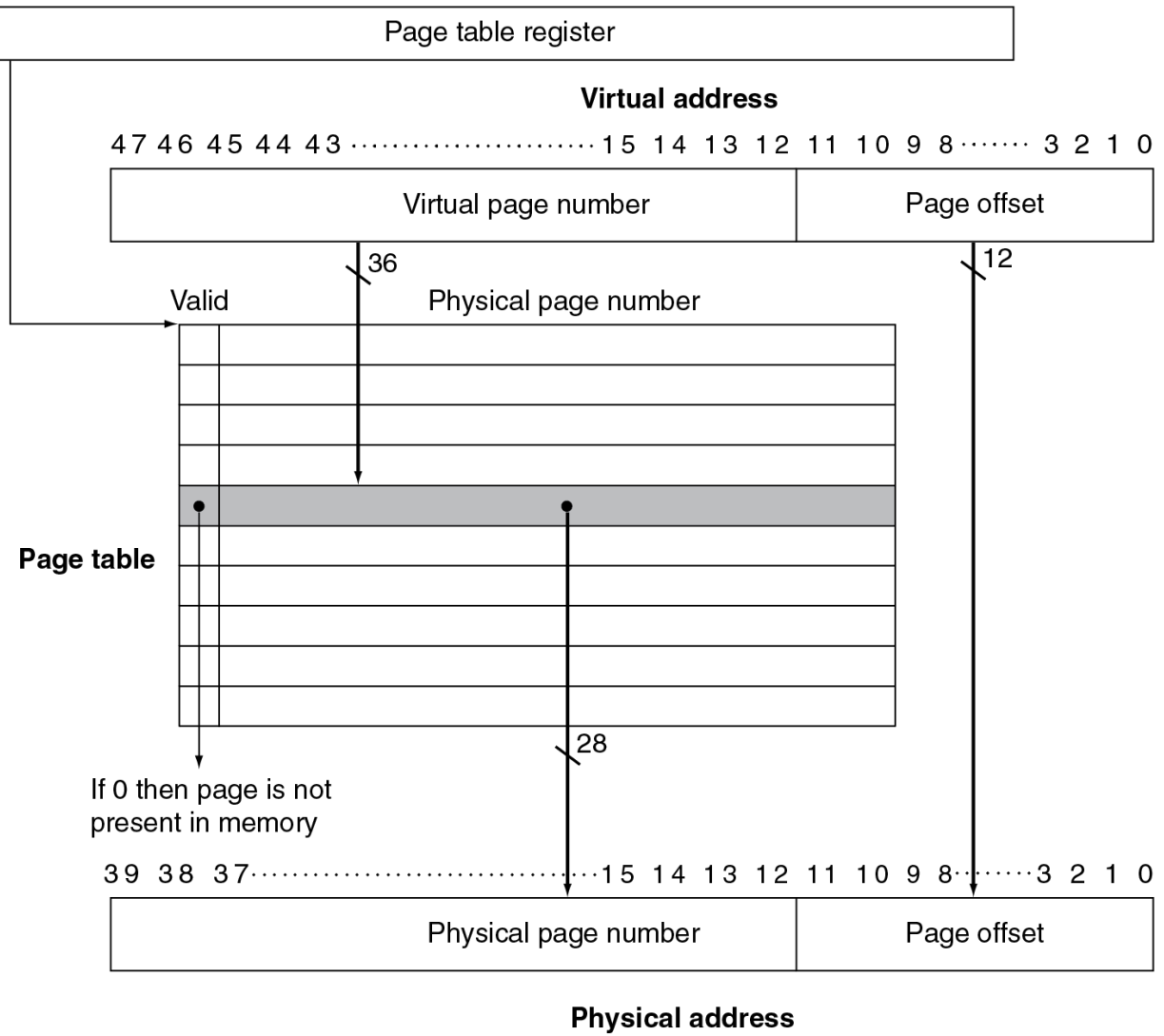
- 缺页失效时，必须从磁盘中提取页面
  - （对于机械硬盘来说）需要数百万个时钟周期
  - 由操作系统代码处理
- 尽量减少缺页失效的比率
  - 全相联的映射方式
  - 智能替换算法

5.7 虚拟存储

- 页表
  - 在虚拟存储系统中，保存着虚拟地址和物理地址之间转换关系的表
  - 页表保存在内存中，通常使用虚拟页号来索引
  - 如果这个页在内存中，页表中的对应项包含该页对应的物理页号
- 页表用来存储映射信息
  - 按虚拟页码索引的页表项数组
  - CPU中的页表寄存器指向物理内存中的页表
- 如果页在内存中
  - PTE存储物理页码
    - page table entry页表条目
  - 加上其他状态位（引用、脏等）
- 如果页不在内存中
  - 页表条目可以指磁盘上交换空间中的位置

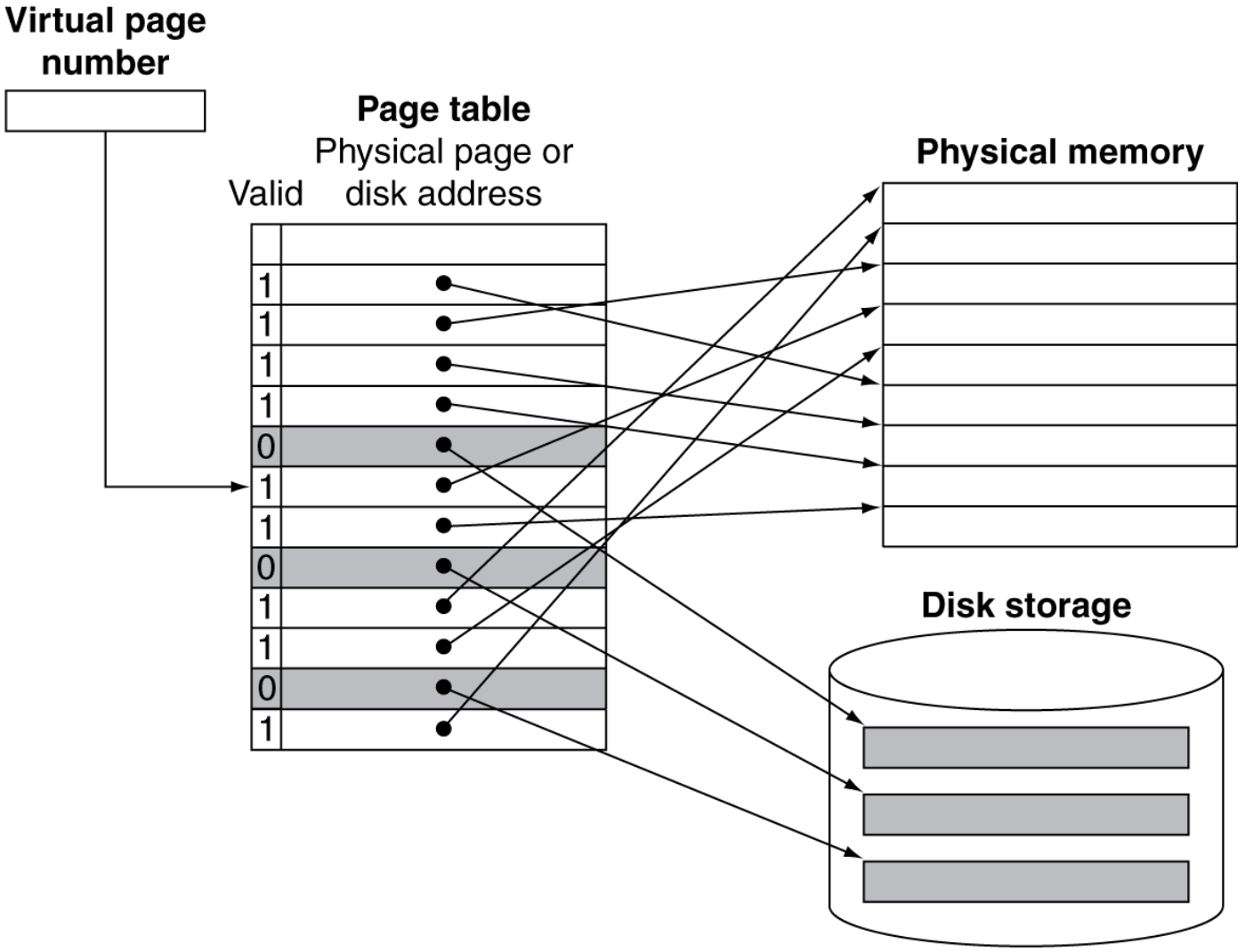
5.7 虚拟存储

- 使用页表进行地址转换



5.7 虚拟存储

- 将页表映射到磁盘



5.7 虚拟存储

- 替换和写操作

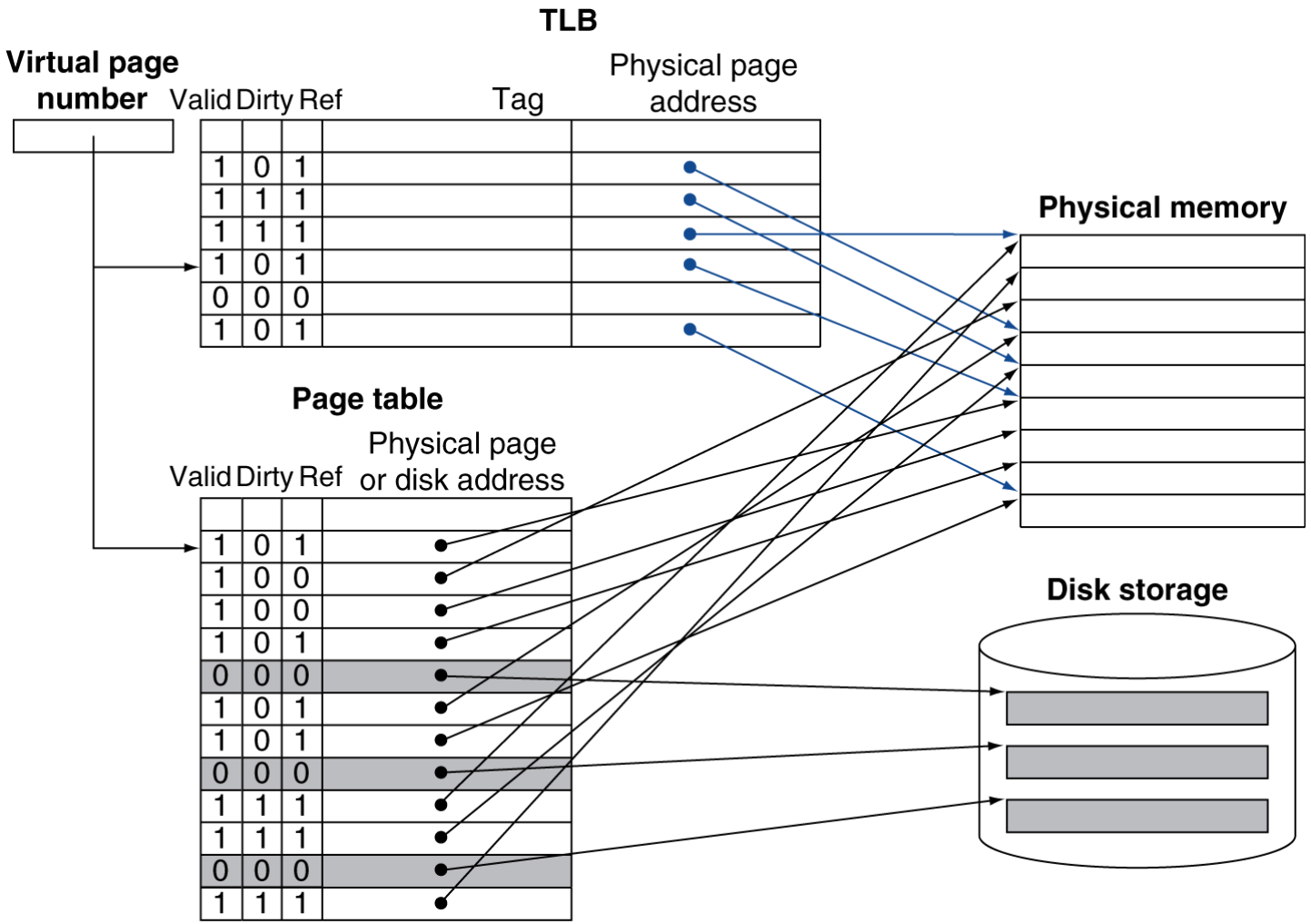
- 要降低缺页失效比率，需要选择最近最少使用（LRU）的页进行替换
  - 访问页时，PTE中的参考位（也称为使用位）设置为1
  - 操作系统定期将其清除为0
  - 最近未使用指的是引用位为0的页面
- 磁盘写入需要数百万个周期
  - 因此需要对整块进行操作，而不是每次访问都进行定位
  - 写直达（或称写穿透）是不现实的
  - 需要使用回写策略
  - 页内发生写操作时，设置PTE中的脏位为1

5.7 虚拟存储

- 使用快表（TLB）进行快速转换
  - TLB： Translation-Lookaside Buffer，用于记录最近使用地址的映射信息的cache，从而避免每次都要访问页表
  - 为什么需要引入快表?
- 地址转换需要额外的内存访问
  - 第一次访问PTE，获取物理地址
  - 第二次才是实际的内存访问
- 但对页表的访问具有良好的局部性
  - 因此，在CPU中使用快速缓存PTE
  - 典型：16-512次PTE，命中时耗时0.5-1个周期，未命中时耗时10-100个周期，未命中率为0.01%-1%
  - 未命中时可以通过硬件或软件处理

5.7 虚拟存储

- 使用快表（TLB）进行快速转换



5.7 虚拟存储

- TLB miss
  - 页在内存中
  - 页不在内存中
- 如果页面在内存中
  - 从内存中加载PTE，然后重试
  - 可以用硬件处理
    - 对于更复杂的页表结构，可能会变得复杂
  - 或者通过软件处理
    - 使用优化的处理程序，引发特殊异常
- 如果页面不在内存中（缺页失效）
  - 操作系统负责获取页面和更新页面表
  - 然后重新启动缺失指令

5.7 虚拟存储

- TLB miss 处理函数

- TLB miss表明
  - 页面存在，但PTE不在TLB中
  - 页面不存在
- 必须在重新目标寄存器之前识别TLB未命中
  - 引发异常
- 处理程序将PTE从内存复制到TLB
  - 然后重新启动指令
  - 如果页面不存在，则会出现缺页失效异常

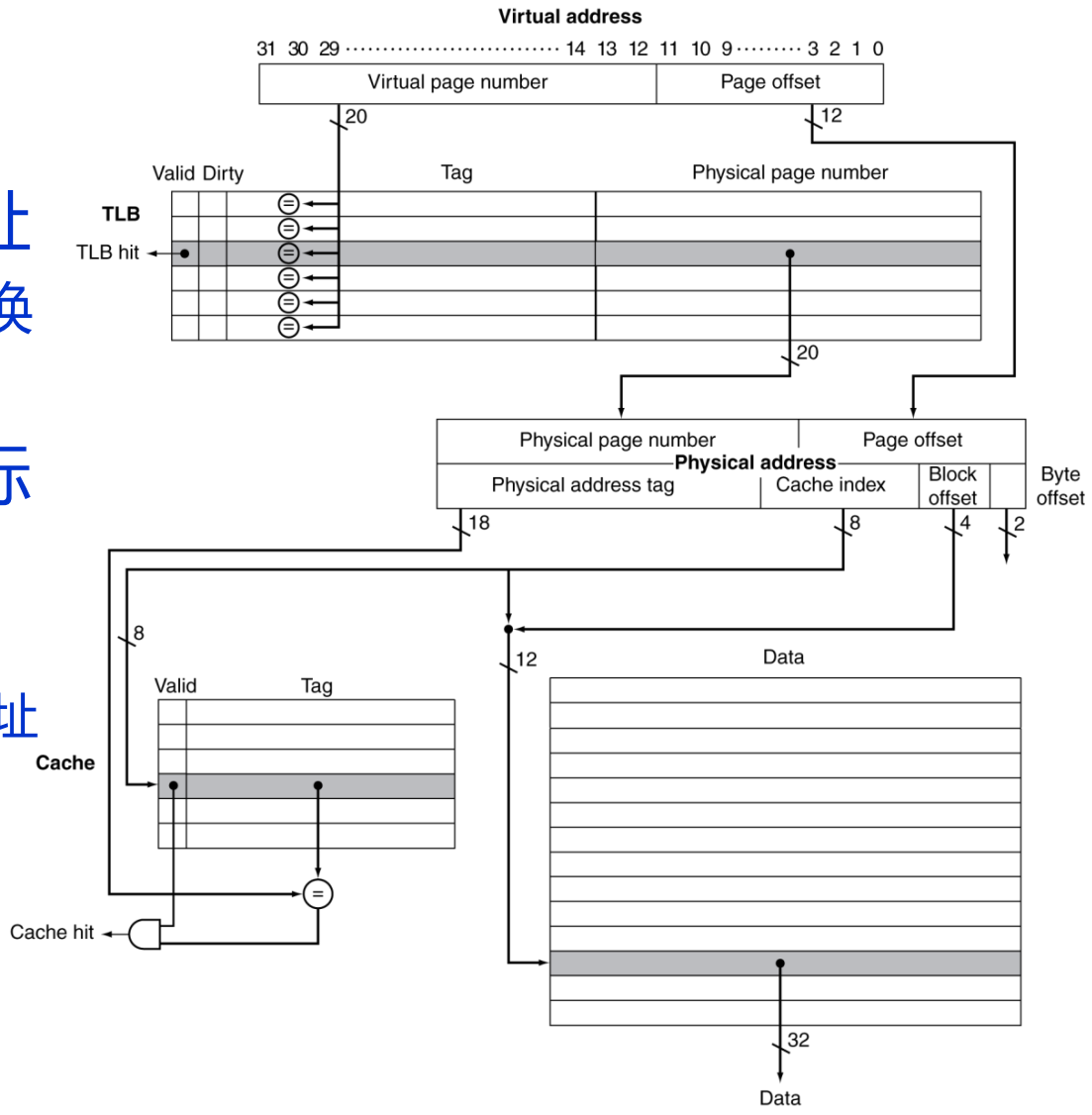
5.7 虚拟存储

- TLB miss 处理函数

- 使用缺失的虚拟地址查找PTE
- 在磁盘上找到页面
- 选择要替换的页面
  - 如果脏了，首先写入磁盘
- 将页面读入内存并更新页面表
- 使进程再次运行
  - 从缺失指令重新启动

5.7 虚拟存储

- TLB 和 Cache 交互
- 如果缓存标记使用物理地址
  - 需要在缓存查找之前进行转换
- 备选方案：使用虚拟地址标签
  - 别名引起的并发症
    - 共享物理地址的不同虚拟地址



5.7 虚拟存储

- 虚拟存储中的保护
- 不同的任务可以共享部分虚拟地址空间
  - 但需要防止错误访问
  - 需要操作系统协助
- 对操作系统保护的硬件支持
  - 特权管理模式（又称内核模式）
  - 特权指令
  - 页面表格和其他状态信息只能在管理器模式下访问
  - 系统调用异常（例如RISC-V中的ecall）
- 管理模式
  - 也称内核模式，是一种运行操作系统进程的模式
- 系统调用
  - 将控制权从用户模式转换到管理模式的特殊指令，触发进程中的一个例外机制

5.8 存储器层次结构的一般框架

- 通用原则适用于内存层次结构的所有级别
  - 基于缓存的概念
- 在层次结构中的每个级别都需要考虑如下问题
  - 块放置
  - 找到一个block
  - 未命中时的替换策略
  - 写策略

5.8 存储器层次结构的一般框架

- 块放置
  - 主要取决于相联性
- 直接映射（1路组相联）
  - 每个数据块只有一个对应位置可以放置
- N路组相联
  - 每个数据块有N个对应位置可以放置
  - N个为一组（set）
- 全相联
  - 数据块可放置在任意位置

- 增加相联度可以提高命中率
  - 相应的，会增加复杂性、成本和访问时间

5.8 存储器层次结构的一般框架

- 硬件缓存
  - 一般使用N路组相联方式
  - 减少比较器，从而降低成本和复杂度
- 虚拟存储
  - 一般采用全相联方式
  - 条目少，因此全相联方式可行
  - 可以提高命中率

相联方式	定位方式	标签比较器数量
直接映射	索引	1
N路组相联	索引组,查找组中元素	N
全相联	查找所有的cache表项	等于所有条目的数量
	独立的查找表	0

5.8 存储器层次结构的一般框架

- 替换策略

- 在未命中时选择要替换的条目
  - 最近最少使用（LRU）
    - 复杂且昂贵的硬件可实现高关联性
    - 一般实现的是LRU的近似形式
  - 随机选择
    - 性能接近LRU，更容易实施

- 对于虚拟存储
  - 失效代价很高
  - 发生频率较低
  - 经常使用软件实现近似的LRU算法

5.8 存储器层次结构的一般框架

- 写策略

- 写直达策略
  - 更新上层和下层
  - 简化了替换流程，但可能需要写缓冲区
- 写返回策略
  - 仅更新上层
  - 替换块时更新较低级别
  - 需要保存更多的状态

- 对于虚拟存储
  - 考虑到磁盘写入延迟，只有回写是可行的

5.8 存储器层次结构的一般框架

- 失效分类
  - 3C模型：将所有的cache失效都归为三种类型的cache模型，分别为强制失效（Compulsory misses）、容量失效（Capacity misses）、冲突失效（Conflict misses）
  - 强制失效，也称为冷启动失效
    - 对没有在cache中出现过的块进行第一次访问是产生的失效
  - 容量失效
    - 由于cache在全相联时都不可能容纳所有请求的块而导致的失效
    - 例如，一个被替换掉的块又被再次访问
  - 冲突失效，也称为碰撞失效
    - 在组相连或者直接映射cache中，很多块为了竞争同一个组导致的失效，这种失效在使用相同大小的全相联cache中是不存在的。

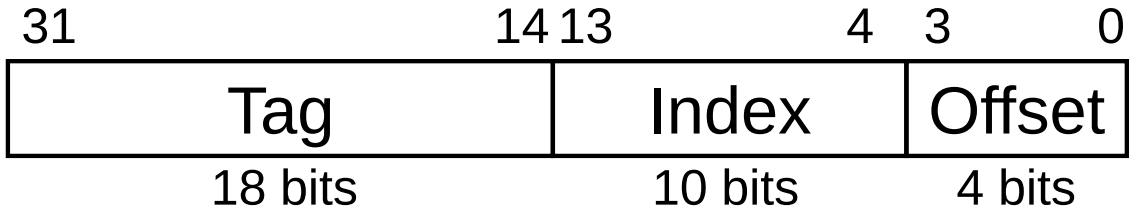
5.8 存储器层次结构的一般框架

- cache设计中的权衡

设计变化	对失效率的影响	可能对性能产生的负面影响
增加cache容量	降低失效率	可能延长访问时间
增加相联度	由于减少了冲突失效，降低了失效率	可能延长访问时间
增加块容量	由于空间局部性，对很宽范围内变化的块大小，降低了失效率	增加失效损失，块太大还会增大失效率

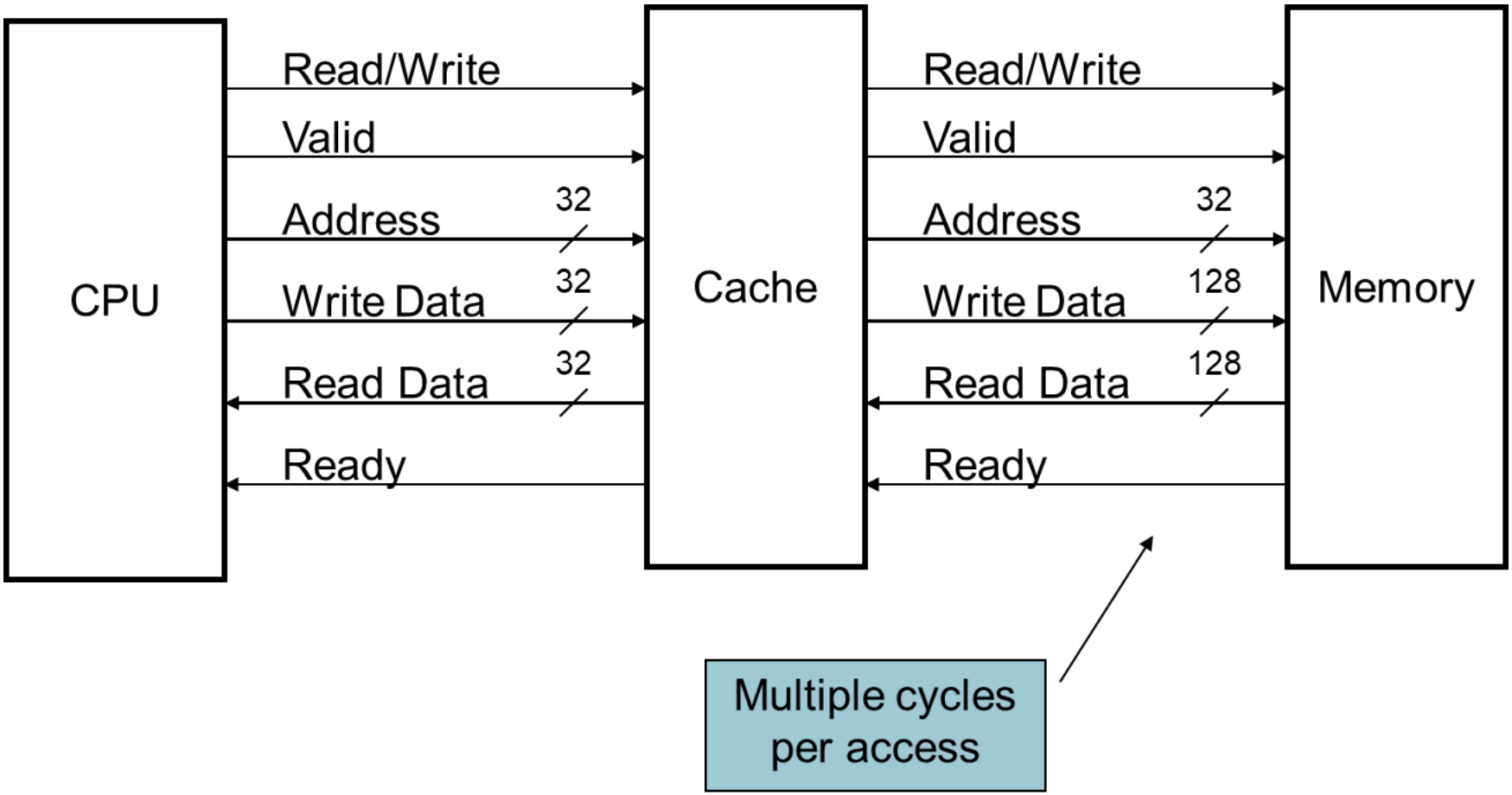
5.9 使用有限状态机控制简单的cache

- 缓存特性
  - 直接映射、写回、写分配
  - 块大小：4个字（16字节）
  - 缓存大小：16 KB（1024个块）
  - 32位字节地址
  - 每个块有效位和脏位各1bit
  - 阻塞缓存
    - CPU等待访问完成
- 计算相关字段位宽
  - cache块索引：10bit
  - 块内偏移：4bit
  - 标签位宽：32-10-4 = 18bit



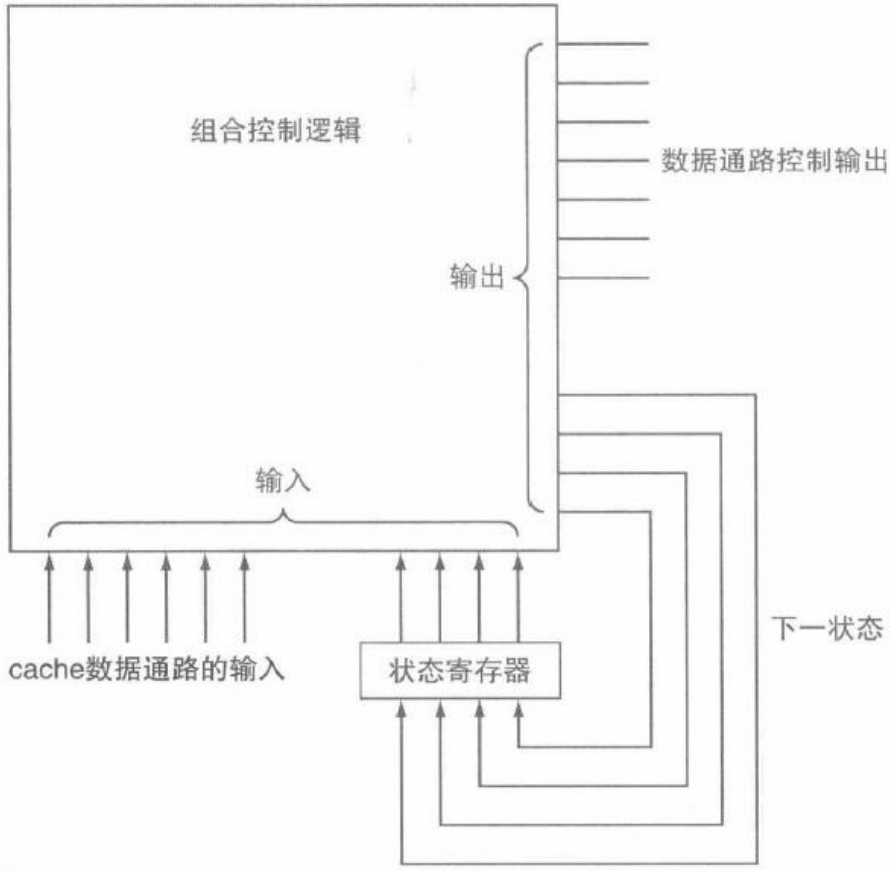
5.9 使用有限状态机控制简单的cache

- 接口信号



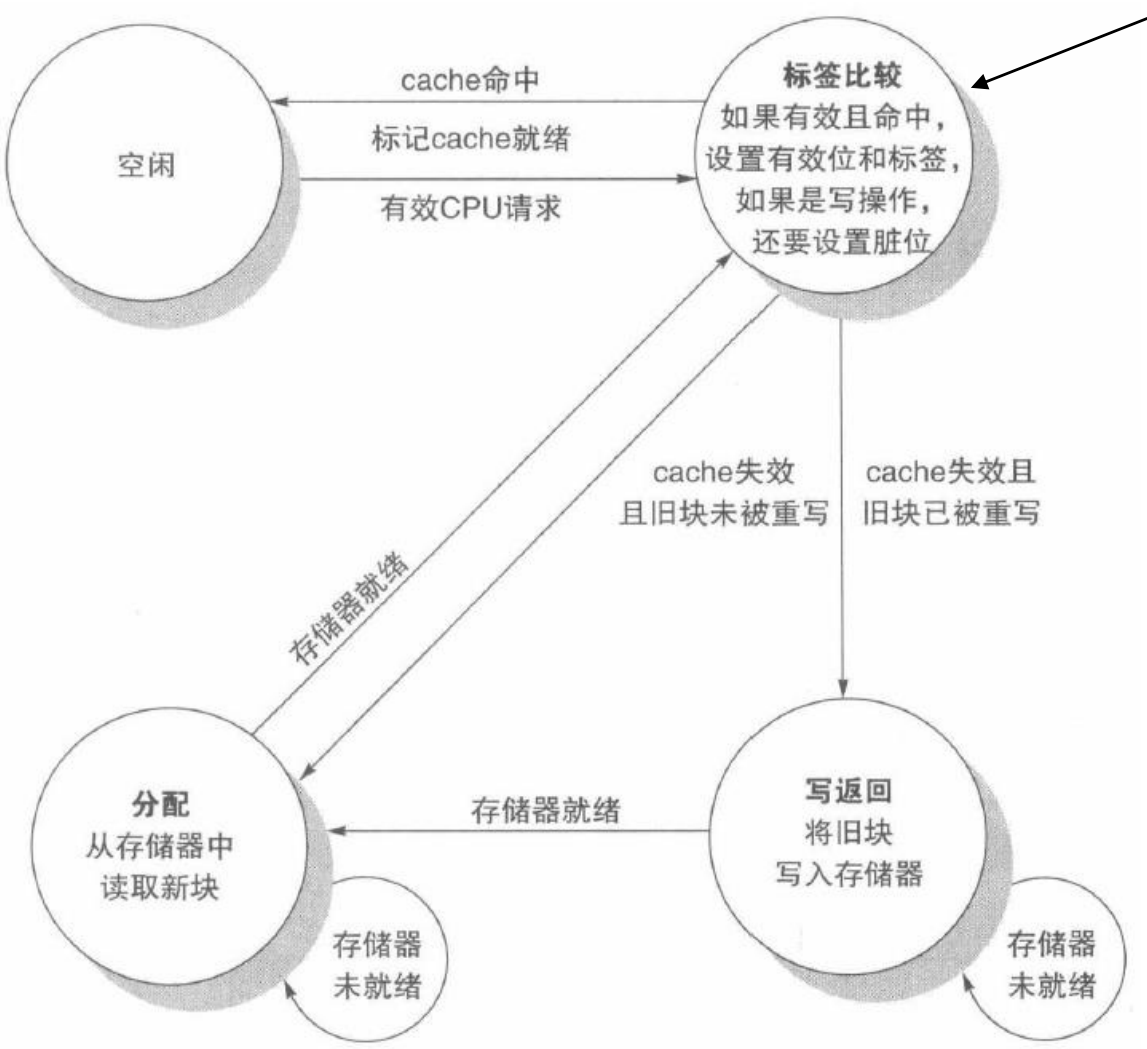
5.9 使用有限状态机控制简单的cache

- 使用FSM对控制步骤进行排序
- 状态集，每个时钟边缘上的转换
  - 状态值是二进制编码的
  - 存储在寄存器中的当前状态
  - 下一状态
    = fn（当前状态，当前输入）
- 控制输出信号
  = fo（当前状态）



5.9 使用有限状态机控制简单的cache

- cache控制器有限状态机



可以划分为不同的状态以减少时钟周期时间

5.10 并行和存储层次结构：cache一致性

- 缓存一致性问题
- 假设两个CPU核共享一个物理地址空间
  - 采用写直达策略

时间	事件	CPU A cache内容	CPU B cache内容	内存位置X的内容
0				0
1	CPU A读X	0		0
2	CPU B读X	0	0	0
3	CPU A向X写入1	1	0	1

5.10 并行和存储层次结构：cache一致性

- 一致性定义
- 非正式地：读取返回最近写入的值
- 正式地：
  - P写X；P读取X（无中间写入）→ 读取时返回写入的值
  - P1写X；P2读X（足够晚） → 读取时返回写入的值
    - 例如在前面示例的步骤3之后，CPU B读取X
  - P1写X，P2写X　所有处理器都以相同的顺序看到写入操作
    - 最终读取到相同的X最终值

5.10 并行和存储层次结构：cache一致性

- cache一致性方案

- 由多处理器中的缓存执行的操作，以确保一致性
  - 将数据迁移到本地缓存
    - 减少共享内存的带宽
  - 读取共享数据的复制
    - 减少访问争用
- 监听协议
  - 每个缓存都监视总线读/写
- 基于目录的协议
  - 目录中块的缓存和内存记录共享状态

5.10 并行和存储层次结构：cache一致性

- 写无效协议
- 缓存在写入块时以独占方式访问该块
  - 在总线上广播失效消息
  - 另一个缓存中的后续读取未命中
    - 拥有缓存提供了更新的价值

处理器行为	总线行为	CPU A cache内容	CPU B cache内容	存储器位置X的内容
				0
CPU A读X	X在cache中失效	0		0
CPU B读X	X在cache中失效	0	0	0
CPU A向X写入1	令X无效	1		0
CPU B读X	X在cache中失效	1	1	1

5.10 并行和存储层次结构：cache一致性

- 内存一致性
- 其他处理器何时可以看到写操作
  - “Seen”表示读取返回写入的值
  - 不可能是瞬间的
- 假设
  - 只有当所有处理器都看到写入时，写入才会完成
  - 处理器不会对其他访问的写入进行重新排序
- 结果
  - P写X，然后写Y ⇒ 所有看到新Y的处理器也会看到新X
  - 处理器可以对读取进行重新排序，但不能对写入进行重新排序

5.11 并行与存储层次结构：RAID

- 略

5.12 高级专题：实现缓存控制器

- 略

5.13 实例

- 多级片上cache

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	16-way set associative	8-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

5.13 实例

- 两级TLB结构

Characteristic	ARM Cortex-A53	Intel Core i7
Virtual address	48 bits	48 bits
Physical address	40 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both micro TLBs are fully associative, with 10 entries, round robin replacement 64-entry, four-way set-associative TLBs TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, seven per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

5.13 实例

- Cortex-A53和Core i7对多发射的支持

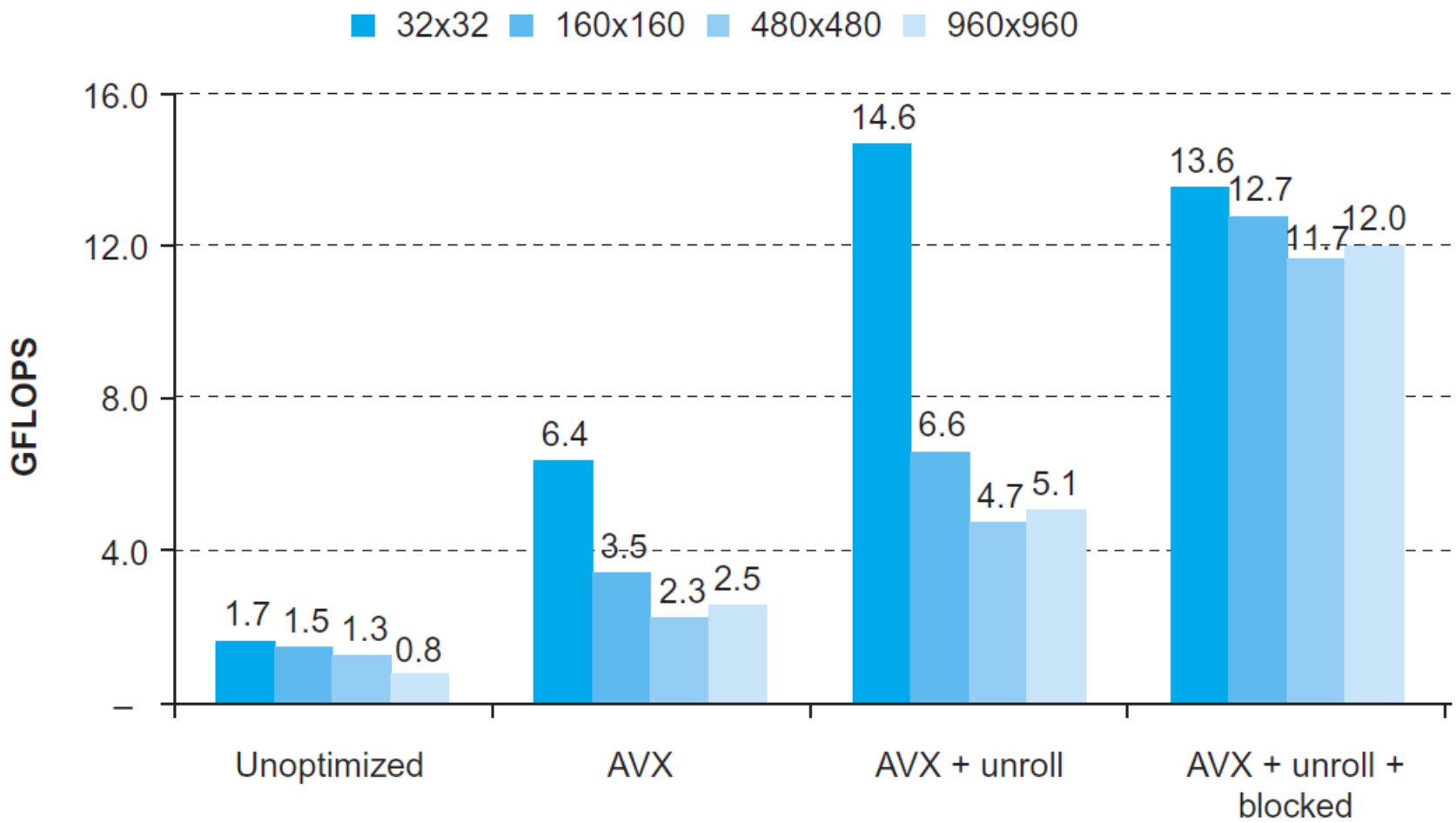
- 两者都有多银行缓存，在不存在银行冲突的情况下，允许每个周期进行多次访问
- 其他优化
  - 关键字优先：首先返回请求的数据
  - 非阻塞缓存
    - 缺失下命中：缺失期间有其它缓存命中
    - 缺失下缺失：允许发生多个未完成的缓存缺失
  - 数据预取

5.14 实例： RISC-V系统的其它部分和特殊指令

Type	Mnemonic	Name
Mem ordering	fence.i	Instruction fence
	fence	Fence
	sfence.vm	Address translation fence
CSR access	csrrwi	CSR read/write immediate
	csrrsi	CSR read/set immediate
	csrrci	CSR read/clear immediate
	csrrw	CSR read/write
	csrrs	CSR read/set
	csrrc	CSR read/clear
System	ecall	Environment call
	ebreak	Environment breakpoint
	sret	Supervisor exception return
	wfi	Wait for interrupt

5.15 加速：cache分块和矩阵乘法

- DGEMM
  - 结合子字并行子字并行、循环展开、缓存阻塞技术等优化技术



5.16 谬误与陷阱

- 字节编址 vs字编址
  - 示例：32byte直接映射的cache，4byte/block
    - 字节地址36，映射到block1
    - 字地址36，映射到block4
- 编写或生成代码时，忽略存储系统的影响
  - 示例：对数组行进行迭代 vs 对列进行迭代
  - 大范围跳转导致较差的局部性

5.16 谬误与陷阱

- 在具有共享二级或三级缓存的多处理器中
  - 关联性小于核心会导致冲突未命中
  - 更多核心　需要增加关联性
- 使用AMAT评估无序处理器的性能
  - 忽略非阻塞访问的影响
  - 相反，通过模拟来评估性能

5.16 谬误与陷阱

- 使用段扩展地址范围
  - 例如，英特尔80286
  - 但一个细分市场并不总是足够大
  - 使地址算法变得复杂
- 在非虚拟化设计的ISA上实现VMM
  - 例如，访问硬件资源的非特权指令
  - 要么扩展ISA，要么要求来宾操作系统不要使用有问题的指令

5.17 本章小结

- 快的记忆很小，大的记忆很慢
  - 我们真的想要快速、大容量的记忆
  - 缓存带来了这种错觉
- 局部性原则
  - 程序经常使用一小部分内存空间
- 内存层次结构
  - 一级缓存 ↔ 二级缓存 ↔ ... ↔ DRAM存储器 ↔ 磁盘
- 内存系统设计对多处理器至关重要

计算机组成原理
CH6_并行处理器：从客户端到云

卢建良
lujl@ustc.edu.cn
2022年春季学期

提纲

- 6.1 引言
- 6.2 创建并行处理程序的难点
- 6.3 SISD、MIMD、SIMD、SPMD、SPMD和向量机
- 6.4 硬件多线程
- 6.5 多核及其他共享内存多处理器
- 6.6 GPU简介
- 6.7 集群、仓储级计算机和其它消息传递多处理器
- 6.8 多处理器网络拓扑简介
- 6.9 与外界通信：集群网络

提纲

- 6.10 多处理器测试基准和性能模型
- 6.11 实例：评测Intel Core i7和NVIDIA Tesla GPU
- 6.12 加速：多处理器和矩阵乘法
- 6.13 谬误与陷阱
- 6.14 本章小结

6.1 引言

- 目标：连接多台计算机以获得更高的性能
  - 多处理器
  - 可扩展性、可用性和能效
- 任务级/进程级并行性
  - 通过同时运行独立的多个程序来使用多处理器
  - 独立工作的任务，高吞吐量就代表着高性能
- 并行处理程序
  - 同时在多个处理器上运行的单个程序
- 多核微处理器
  - 多处理器芯片（核心）

6.1 引言

- 实例：硬件和软件
- 硬件
  - 串行：例如奔腾4
  - 并行：例如四核至强e5345
- 软件
  - 顺序：例如，矩阵乘法
  - 并发：例如，操作系统
- 顺序/并发软件可以在串行/并行硬件上运行
  - 挑战：有效利用并行硬件

		软件	
		顺序	并发
硬件	串行	在Intel Pentium 4上运行的使用MATLAB编写的矩阵乘法程序	在Intel Pentium 4上运行的Windows Vista的操作系统程序
	并行	在Intel Core i7上运行的使用MATLAB编写的矩阵乘法程序	在Intel Core i7上运行的Windows Vista的操作系统程序

6.1 引言

- 已经涉及到的内容总结

- §2.11：并行性及其说明
  - 同步
- §3.6：并行性和计算机算法
  - 子字并行
- §4.10：并行性和高级指令级并行性
- §5.10：并行性和内存层次结构
  - 缓存一致性

自我检测 判断题：要从多处理器中获得收益，应用程序必须是并发的。

6.2 创建并行处理程序的难点

- 并行程序
  - 能够并行（在多个处理器上）运行的单个程序
- 并行程序的主要困难在于软件
  - 软件重写
  - 提高性能和效率
  - 否则，只需使用更快的单处理器，因为它更容易！
- 困难点
  - 分割
  - 协作
  - 通信开销
  - 示例：8个记者协作编写一个故事

6.2 创建并行处理程序的难点

- Amdahl定律

$$\text{加速比} = \frac{\text{改进前的执行时间}}{\text{改进前的执行时间} - \text{受优化影响的执行时间} + \frac{\text{受优化影响的执行时间}}{\text{优化量}}}$$

$$\text{加速比} = \frac{1}{1 - \text{受优化影响的执行时间比例} + \frac{\text{受优化影响的执行时间比例}}{\text{优化量}}}$$

- 顺序部分可以限制加速
- 示例：100个处理器，如何获取90倍的加速比?

$$T_{new} = T_{parallelizable}/100 + T_{sequential}$$

$$\text{Speedup} = \frac{1}{(1 - F_{parallelizable}) + F_{parallelizable}/100} = 90$$

结论: $F_{parallelizable} = 0.999$

- 要求顺序部分在原始时间的占比小于0.1%

6.2 创建并行处理程序的难点

- 比例缩放示例
- 工作量：10个标量求和，以及10×10矩阵求和
  - 单处理器实现
  - 10个处理器加速
  - 100个处理器加速
  - 假设可以在处理器之间实现负载均衡
- 单处理器
  - 时间=（10+100）×t_{add}
- 10个处理器：
  - 时间=10 × t_{add} + 100/10 × t_{add}=20 × t_{add}
  - 加速比=110/20=5.5（潜力的55%）
- 100个处理器
  - 时间=10×t_{add}+100/100×t_{add}=11×t_{add}
  - 加速比=110/11=10（潜力的10%）

6.2 创建并行处理程序的难点

- 比例缩放示例
  - 假设负载均衡
  - 矩阵大小变为100×100
- 单处理器：时间=（10+10000）×t_{add}
- 10个处理器
  - 时间=10×t_{add}+10000/10×t_{add}=1010×t_{add}
  - 加速比=10010/1010=9.9（99%的潜力）
- 100个处理器
  - 时间=10×t_{add}+10000/100×t_{add}=110×t_{add}
  - 加速比=10010/110=91（91%的潜力）

6.2 创建并行处理程序的难点

- 比例缩放分类
  - 强比例缩放
  - 弱比例缩放
- 强比例缩放：在不增加问题规模的情况下在多处理器上获得的加速比
  - 如前面的例子
- 弱比例缩放：在问题规模与处理器数量成比例增加的情况下在多处理器上获得的加速比
  - 10个处理器，10×10矩阵
    - 时间=$20\times t_{add}$
  - 100个处理器，32×32矩阵
    - 时间=$10\times t_{add}+1000/100\times t_{add}=20\times t_{add}$
  - 在本例中，性能保持不变

6.2 创建并行处理程序的难点

- 程序或操作系统如何在多核处理器或多个处理器上运行

```
    .text
_start:
    # park harts with id != 0
    csrr    t0, mhartid     # read current hart id
    mv  tp, t0          # keep CPU's hartid in its tp for later usage.
    bnez    t0, park        # if we're not on the hart 0
                    # we park the hart
    # Setup stacks, the stack grows from bottom to top, so we put the
    # stack pointer to the very end of the stack range.
    slli    t0, t0, 10      # shift left the hart id by 1024
    la  sp, stacks + STACK_SIZE # set the initial stack pointer
                    # to the end of the first stack space
    add sp, sp, t0      # move the current hart stack pointer
                    # to its place in the stack space

    j   start_kernel        # hart 0 jump to c

park:
    wfi
    j   park

stacks:
    .skip   STACK_SIZE * MAXNUM_CPU # allocate space for all the harts stacks

    .end                # End of file
```

6.3 SISD、MIMD、SIMD、SPMD、SPMD和向量机

- 指令流和数据流
  - SISD
  - SIMD
  - MISD
  - MIMD

		数据流	
		单数据流	多数据流
指令流	单指令流	SISD：Intel Pentium 4	SIMD：x86的SSE指令
	多指令流	MISD：目前为止还没有实例	MIMD：Intel Core i7

- SPMD
  - 单程序多数据
  - 一种传统的MIMD编程模型，该模型中单个程序运行在所有处理器上
  - MIMD计算机上的并行程序
  - 不同处理器根据条件代码执行不同的代码段

6.3 SISD、MIMD、SIMD、SPMD、SPMD和向量机

- 向量机
  - SIMD
- 高度流水线功能单元
- 向量寄存器与存储单元之间的数据流传递
  - 从内存收集数据到寄存器
  - 运算结果从寄存器存储到内存
- 示例：RISC-V的向量扩展
  - v0到v31:32×64元素寄存器（64位元素）
  - 向量指令
    - fld.v, fsd.v：加载/存储向量
    - fadd.d.v：添加双精度向量
    - fadd.d.vs：向双精度向量的每个元素添加标量
- 显著减少取指所占用的带宽

6.3 SISD、MIMD、SIMD、SPMD、SPMD和向量机

- 示例：DAXPY（Y = a × X + Y）
  - Conventional RISC-V code:

```
        fld     f0,a(x3)      // load scalar a
        addi    x5,x19,512    // end of array X
loop:   fld     f1,0(x19)     // load x[i]
        fmul.d  f1,f1,f0      // a * x[i]
        fld     f2,0(x20)     // load y[i]
        fadd.d  f2,f2,f1      // a * x[i] + y[i]
        fsd     f2,0(x20)     // store y[i]
        addi    x19,x19,8     // increment index to x
        addi    x20,x20,8     // increment index to y
        bltu    x19,x5,loop   // repeat if not done
```

  Vector RISC-V code:

```
        fld       f0,a(x3)    // load scalar a
        fld.v     v0,0(x19)   // load vector x
        fmul.d.vs v0,v0,f0    // vector-scalar multiply
        fld.v     v1,0(x20)   // load vector y
        fadd.d.v  v1,v1,v0    // vector-vector add
        fsd.v     v1,0(x20)   // store vector y
```

6.3 SISD、MIMD、SIMD、SPMD、SPMD和向量机

- 向量 vs 标量

- 向量体系结构和编译器
  - 简化数据并行编程
  - 不存在循环携带依赖项的明确声明
    - 减少了硬件的检查
  - 常规访问模式受益于交织和突发内存
  - 通过避免循环来避免控制冒险
- 比多媒体扩展（如MMX、SSE）更通用
  - 能够更好地匹配编译器技术

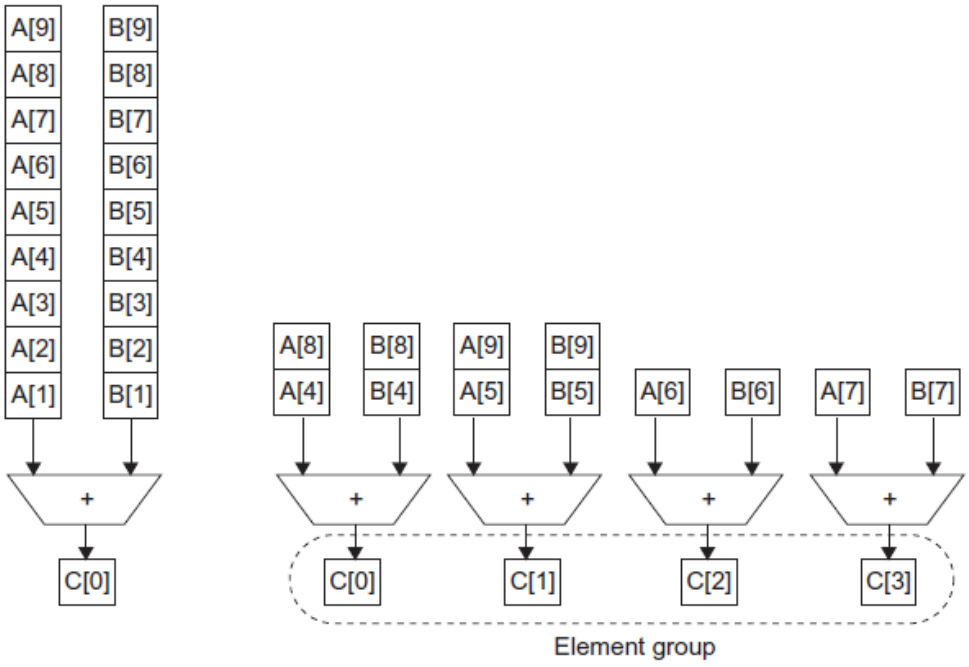
6.3 SISD、MIMD、SIMD、SPMD、SPMD和向量机

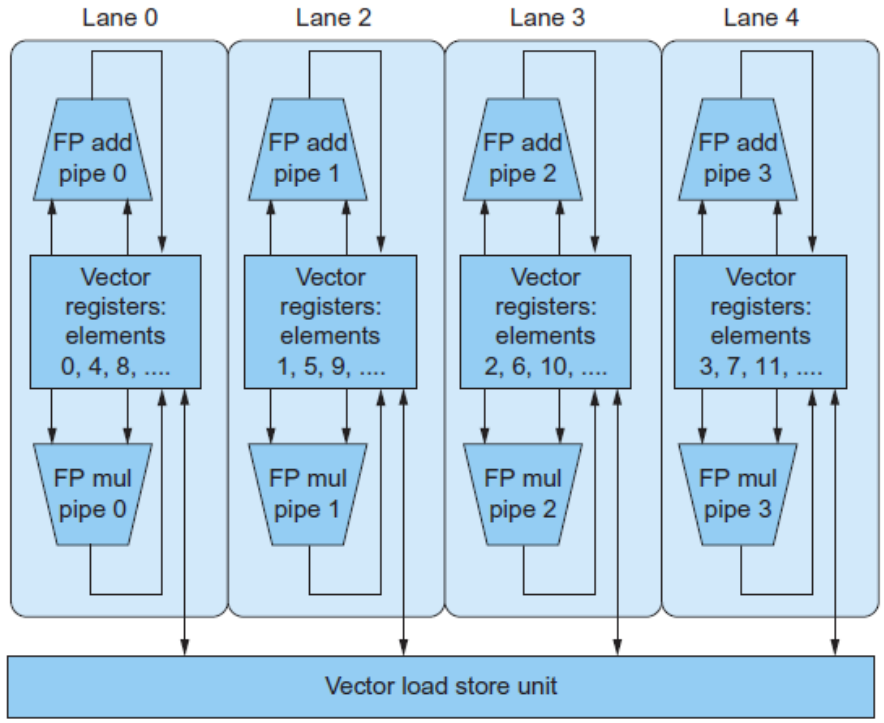
- SIMD，单指令多数据

- 对向量的数据元素进行操作
  - 例如x86中的MMX和SSE指令
    - 128位宽寄存器中的多个数据元素
- 所有处理器同时执行同一条指令
  - 每个都有不同的数据地址等。
- 简化同步
- 精简指令控制相关的硬件
- 最适用于高度数据并行的应用程序

6.3 SISD、MIMD、SIMD、SPMD、SPMD和向量机

- 向量（矢量）与多媒体扩展（MMX）
  - 矢量指令有可变的矢量宽度，多媒体扩展有固定的宽度
  - 矢量指令支持快速访问，而多媒体扩展不支持
  - 矢量单元可以是流水线和阵列功能单元的组合：





6.4 硬件多线程

- 多线程 Multithreading

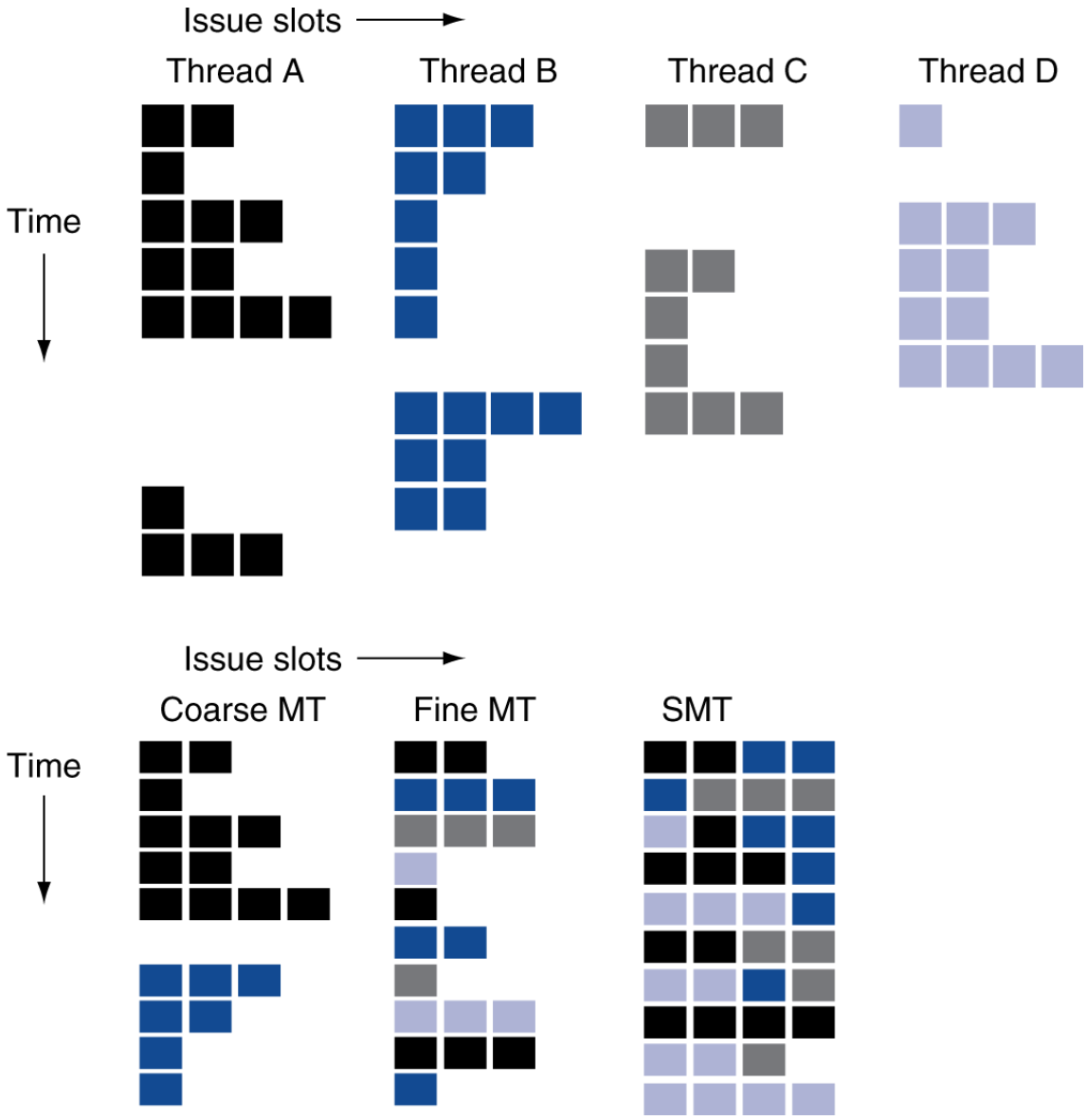
- 并行执行多个线程
  - 复制寄存器、PC等。
  - 线程之间的快速切换
- 细粒度多线程
  - 在每个周期后切换线程
  - 交织指令执行
  - 如果一个线程暂停，其他线程将被执行
- 粗粒度多线程
  - 仅打开长暂停（例如二级缓存未命中）
  - 简化硬件，但不隐藏短暂停（例如，数据冒险）

6.4 硬件多线程

- 同时多线程
  - SMT，Simultaneous Multithreading
- 在多问题动态调度处理器中
  - 调度来自多个线程的指令
  - 当功能单元可用时，执行来自各独立线程的指令
  - 在线程中，依赖项通过调度和寄存器重命名来处理
- 示例：英特尔奔腾4 HT
  - 两个线程：复制寄存器、共享功能单元和缓存

6.4 硬件多线程

- 多线程示例



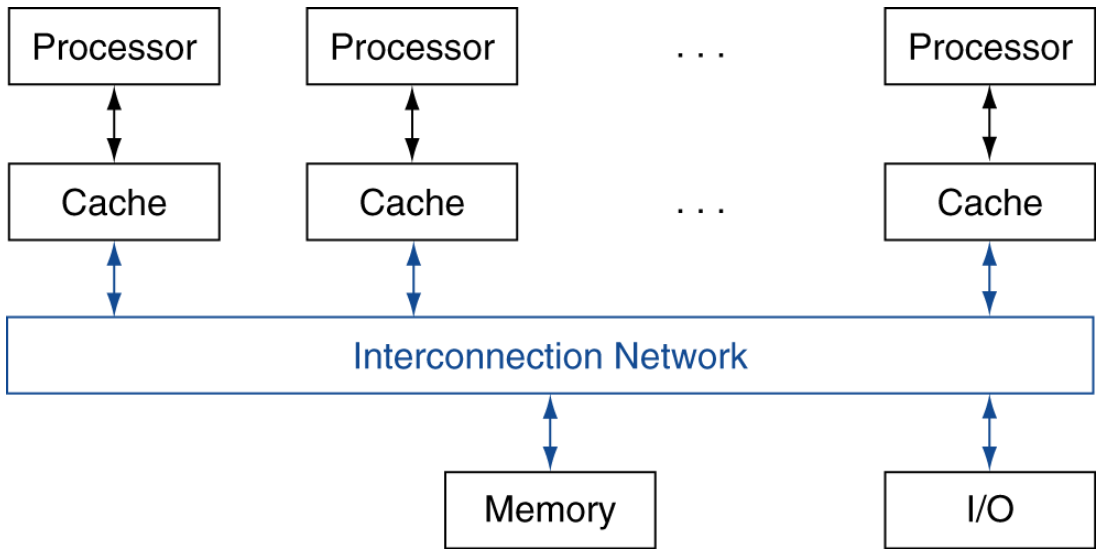
6.4 硬件多线程

- 多线程的未来
  - 还会存在吗？以什么形式？

- 功耗考量 ⇒ 简化微体系结构
  - 更简单的多线程形式
- 容忍缓存未命中延迟
  - 线程切换可能是最有效的
- 多个简单内核可以更有效地共享资源

6.5 多核及其他共享内存多处理器

- 共享内存
- 共享内存多处理器
  - SMP，shared memory multiprocessor
  - 硬件为所有处理器提供单一物理地址空间
  - 使用锁同步共享变量
  - 内存访问时间是否跟发起请求的具体处理器有关?
    - UMA（uniform）与NUMA（nonuniform）



6.5 多核及其他共享内存多处理器

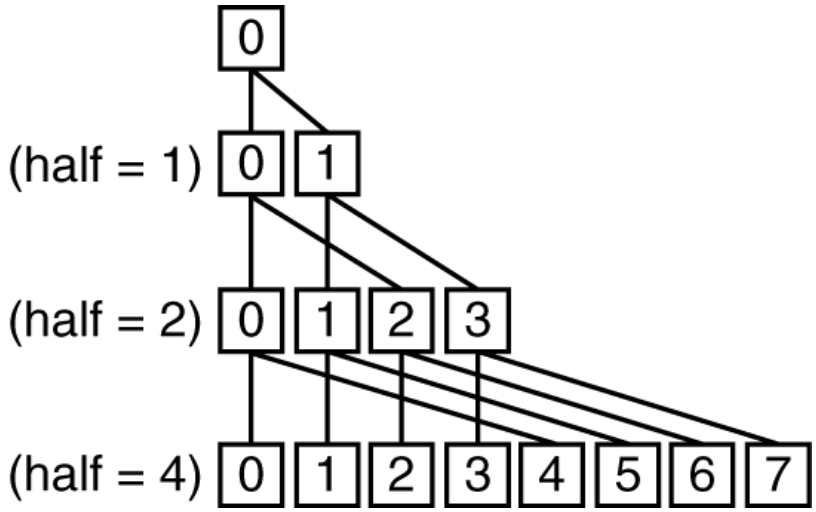
- 示例：规约求和
- 64处理器UMA上的64000个数字之和
  - 每个处理器的ID为：0≤ Pn≤ 63
  - 每个处理器有1000个分区
  - 每个处理器上的初始总和

```
sum[Pn] = 0;
  for (i = 1000*Pn;
       i < 1000*(Pn+1); i += 1)
    sum[Pn] += A[i];
```

- 现在需要加上这些部分和
  - 规约：分而治之
  - 一半的处理器实现两两相加的运算，四分之一的处理器实现...
  - 需要在规约的各步骤之间同步

6.5 多核及其他共享内存多处理器

- 示例：规约求和-续




```
half = 64;
do
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] += sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1);
```

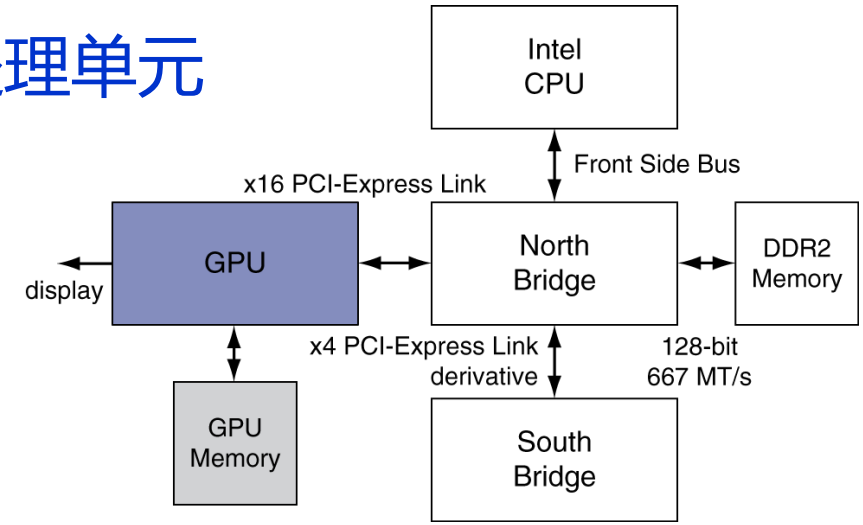
6.6 GPU简介

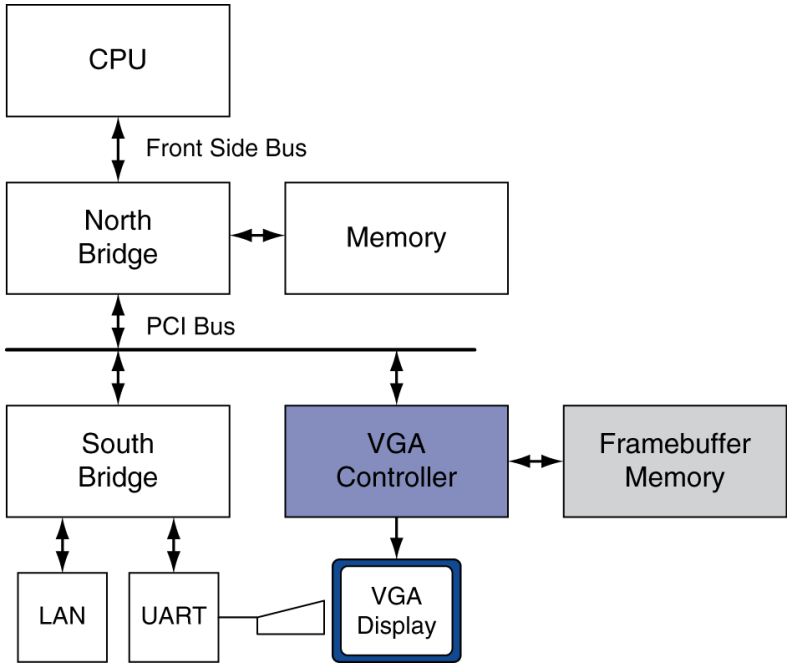
- GPU历史

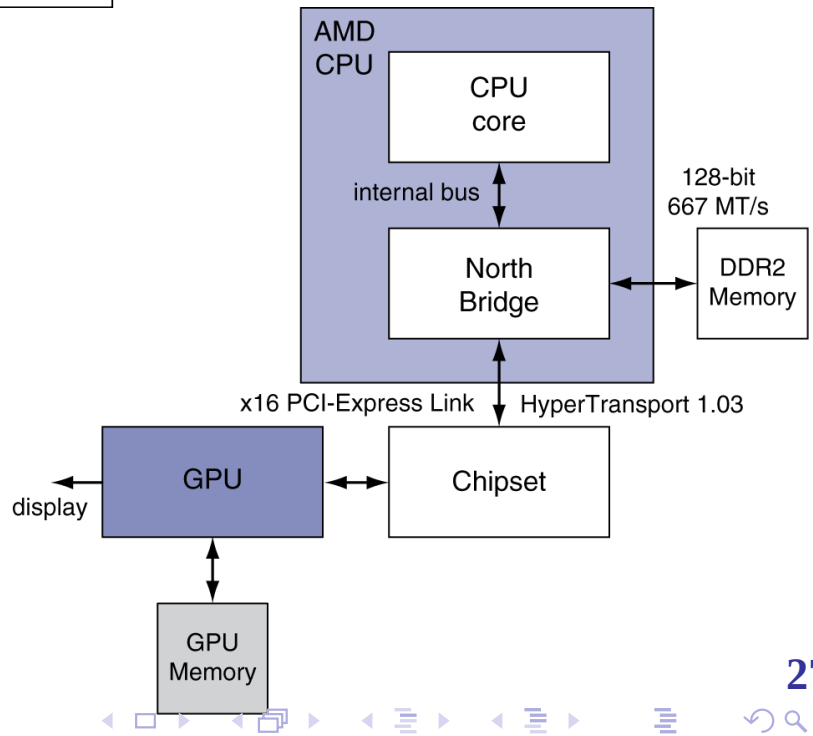
- 早期作为视频卡出现
  - 具有视频输出地址生成功能的帧缓冲存储器
- 三维图形处理
  - 最初是高端计算机（如SGI）
  - 摩尔定律　成本更低，密度更高
  - 用于PC和游戏机的3D图形卡
- 图形处理单元
  - 面向3D图形任务的处理器
  - 顶点/像素处理、着色、纹理映射、光栅化

6.6 GPU简介

- 系统中的图像处理单元





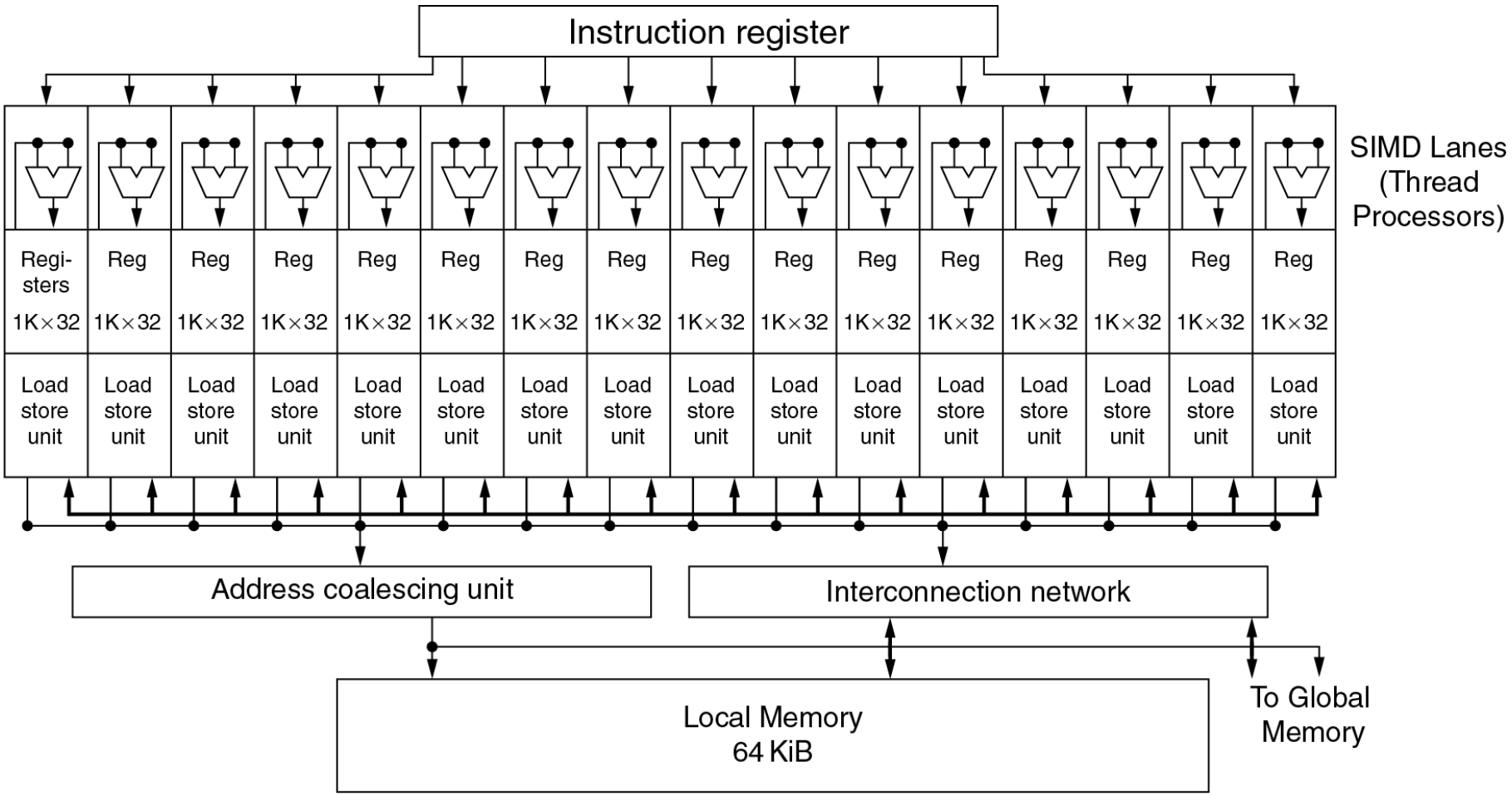


6.6 GPU简介

- GPU体系结构
- 处理是高度并行的
  - GPU是高度多线程的
  - 使用线程切换隐藏内存延迟
    - 减少对多级缓存的依赖
  - 图形内存位宽更宽、带宽更高
- 通用GPU的发展趋势
  - 异构CPU/GPU系统
  - CPU用于顺序代码，GPU用于并行代码
- 编程语言/API
  - DirectX，OpenGL
  - C for Graphics（Cg），High Level Shader Language（HLSL）
  - Compute Unified Device Architecture （CUDA）

6.6 GPU简介

- 示例：NVIDIA Fermi
- 多个SIMD处理器，每个如图所示：



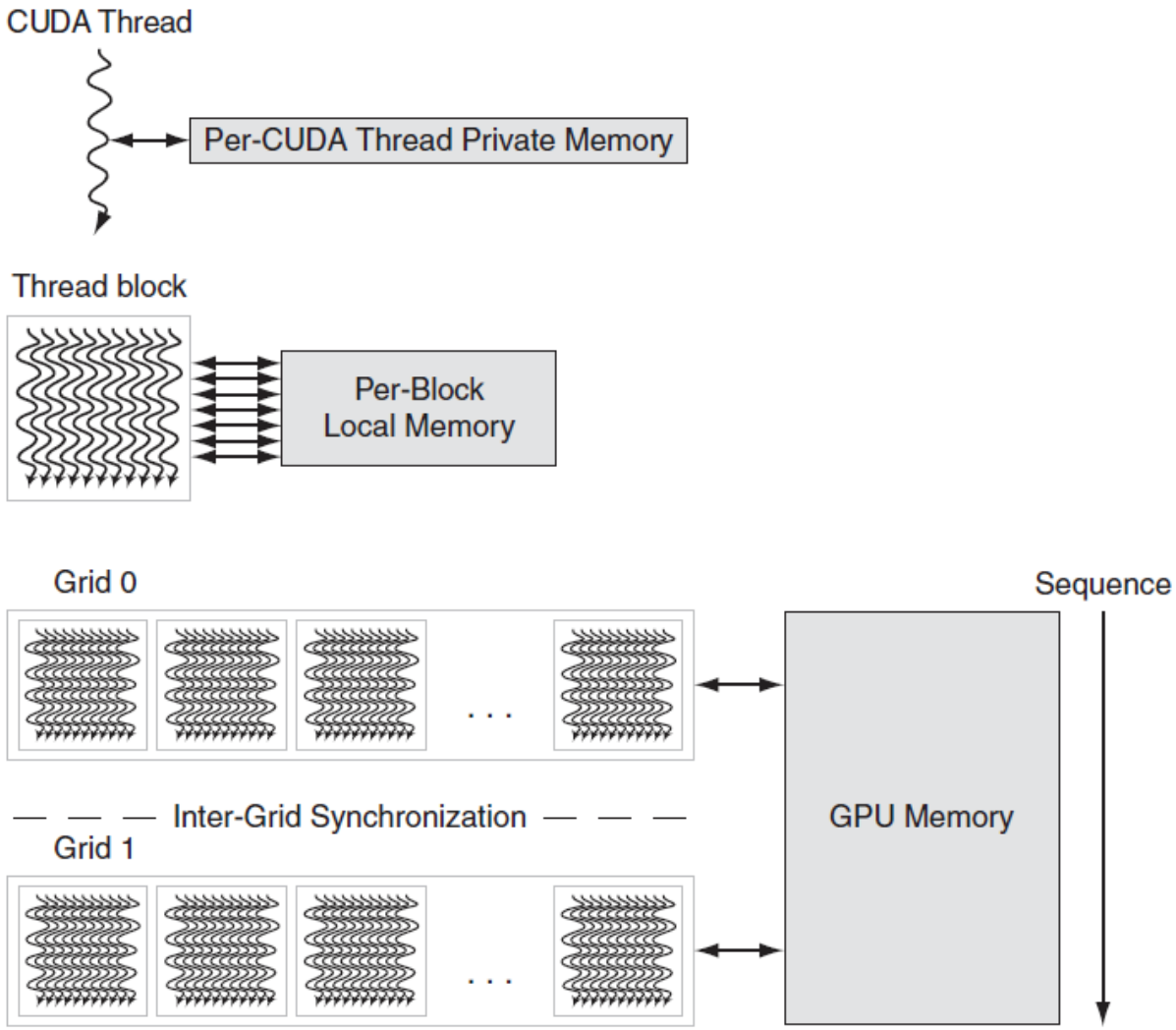
6.6 GPU简介

- 示例：NVIDIA Fermi

- SIMD处理器：16条SIMD通道
- SIMD指令
  - 处理32个元素宽的SIMD指令
  - 需要在16宽处理器上动态调度2个周期
- 32K x 32位寄存器跨通道分布
  - 每个SIMD线程最多可以使用64个向量寄存器
  - 每个向量寄存器有32个元素
  - 每个元素32bit

6.6 GPU简介

- GPU存储结构



6.6 GPU简介

- GPU分类
- 不适合SIMD/MIMD模型
  - 线程中的条件执行允许出现MIMD的假象
    - 但是性能下降了
    - 需要谨慎地编写通用代码

	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**Tesla Multiprocessor**

6.6 GPU简介

- 透视GPU

特点	SIMD扩展的多核计算机	GPU
SIMD处理器数量	4~8	8~16
SIMD通道/处理器数量	2~4	8~16
支持SIMD线程的多线程硬件数量	2~4	16~32
最大cache容量	8 MiB	0.75 MiB
存储器地址大小	64位	64位
主存容量	8~256GiB	4~6GiB
页级别的内存保护	是	是
动态页面调度	是	否
cache一致性	是	否

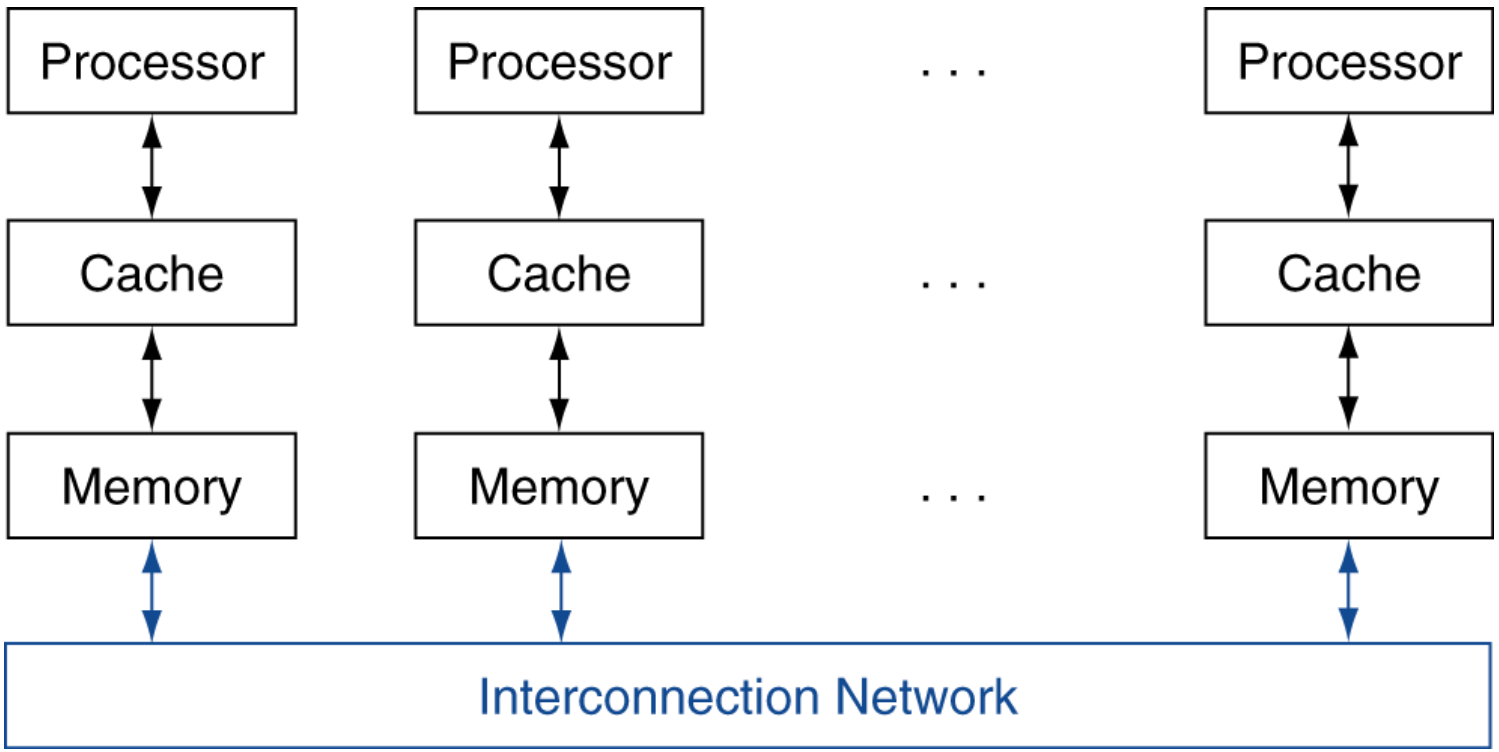
6.6 GPU简介

- GPU术语

类型	更具描述性的术语	最接近的传统术语	NVIDIA GPU 官方术语	教科书定义
程序抽象	可向量化循环	可向量化循环	网格	一个可向量化的循环，在 GPU 上执行，由一个或多个可并行执行的线程块（向量化的循环体）组成
	向量化的循环体	向量化的循环（切分）体	线程块	一个在多线程 SIMD 处理器上执行的向量化的循环，由一个或多个 SIMD 指令线程组成。它们之间可以通过局部存储器进行通信
	SIMD 通道操作序列	标量循环的一次迭代	CUDA 线程	SIMD 指令线程的垂直切割，对应于由一个 SIMD 通道执行的一个元素。根据屏蔽寄存器和预测寄存器存储结果
机器对象	一个 SIMD 指令的线程	一个向量指令的线程	Warp	一个传统线程，但是包含执行在多线程 SIMD 处理器上的 SIMD 指令。根据每个元素的屏蔽寄存器存储结果
	SIMD 指令	向量指令	PTX 指令	一条横跨在多个 SIMD 通道间执行的单个 SIMD 指令
处理硬件	多线程 SIMD 处理器	（多线程的）向量处理器	流多处理器	多线程 SIMD 处理器执行 SIMD 指令线程，与其他 SIMD 处理器相互独立
	线程块调度器	标量处理器	千兆线程引擎	将多个线程块（向量化的循环体）分配到多线程 SIMD 处理器上
	SIMD 线程调度器	多线程 CPU 中的线程调度器	Warp 调度器	当 SIMD 指令线程准备好执行时调度并发射的硬件单元，包括一个追踪 SIMD 线程执行的记分板
	SIMD 通道	向量通道	线程处理器	一个 SIMD 通道，执行单个元素上的 SIMD 指令线程的操作。根据屏蔽寄存器存储结果
存储器硬件	GPU 存储器	主存	全局存储器	可以被一个 GPU 上的所有多线程 SIMD 处理器访问的 DRAM 存储器
	局部存储器	局部存储器	共享存储器	一个多线程 SIMD 处理器上的快速的局部 SRAM，不可被其他 SIMD 处理器访问
	SIMD 通道寄存器	向量通道寄存器	线程处理器寄存器	在整个线程块（向量化的循环体）中分配的一个 SIMD 通道中的寄存器

6.7 集群、仓储级计算机和其它消息传递多处理器

- 消息传递
  - 每个处理器都有专用的物理地址空间
  - 硬件在处理器之间发送/接收消息



6.7 集群、仓储级计算机和其它消息传递多处理器

- 松散耦合集群
- 独立计算机网络
  - 每个都有专用内存和操作系统
  - 使用I/O系统连接
    - 例如，以太网/交换机、互联网
- 适用于具有独立任务的应用程序
  - 网络服务器、数据库、模拟...
- 高可用性、可扩展性和经济性
- 问题
  - 管理成本（首选虚拟机）
  - 低互连带宽
  - SMP上的处理器/内存带宽

6.7 集群、仓储级计算机和其它消息传递多处理器

- 规约求和（again）
- 在64个处理器上对64000个元素求和
- 首先向每个处理器分发1000个数字
  - 这个函数做部分和
  - 总和=0；for（i=0；i<1000；i+=1）
    sum+=AN[i]；
- 规约求和
  - 一半的处理器发送，另一半接收并求和
  - 四分之一的处理器发送、四分之一的处理器接收、求和
  - …

6.7 集群、仓储级计算机和其它消息传递多处理器

- 规约求和（again）
- 假设已经实现了send() 和 receive() 操作

```
limit = 64; half = 64;/* 64 processors */
do
  half = (half+1)/2; /* send vs. receive
                        dividing line */
  if (Pn >= half && Pn < limit)
    send(Pn - half, sum);
  if (Pn < (limit/2))
    sum += receive();
  limit = half; /* upper limit of senders */
while (half > 1); /* exit with final sum */
```

- 发送/接收还需要提供同步功能
- 假设发送/接收与求和运算花费的时间相当

6.7 集群、仓储级计算机和其它消息传递多处理器

- 网格计算

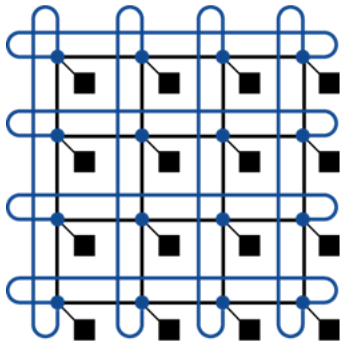
- 通过长途网络互连的独立计算机
  - 例如，互联网连接
  - 工作单位外包，将计算结果发回
- 可以利用PC上的空闲时间
  - 例如：SETI@home项目，世界社区网格

6.8 多处理器网络拓扑简介

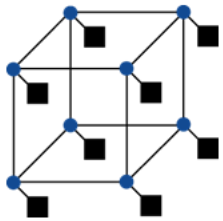
- 互连网络
- 网络拓扑
  - 处理器、交换机和链路的安排

Bus

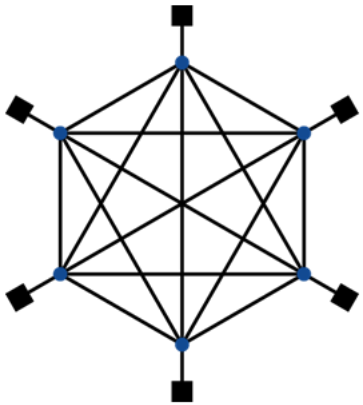
Ring



2D Mesh



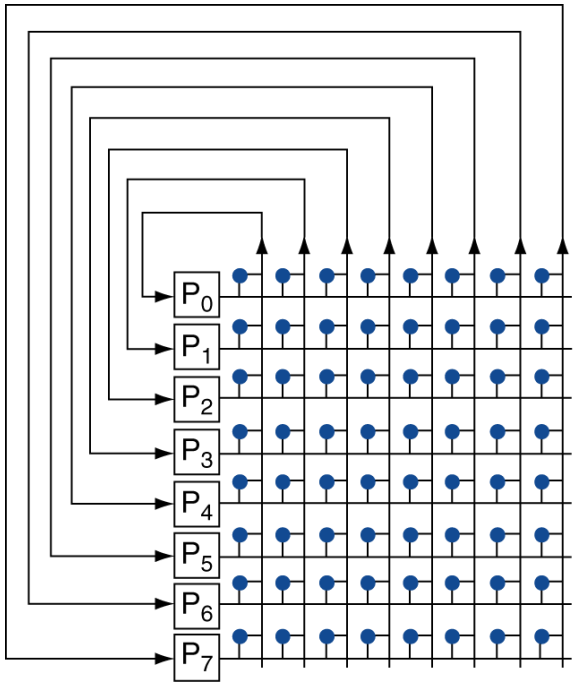
N-cube (N = 3)



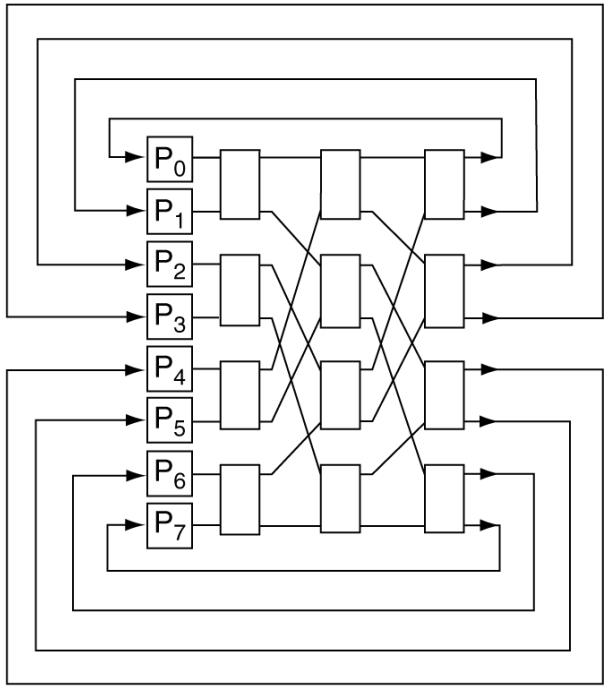
Fully connected

6.8 多处理器网络拓扑简介

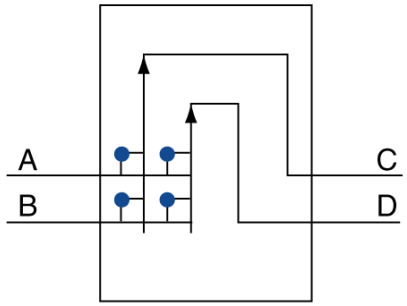
- 多级网络
  - 交叉开关网络
    - 性能最好
    - 成本最高
  - Omega网络
    - 有限制，会冲突
      - P1~P4
      - P0~P6
    - 成本低
  - Omega网络开关盒



a. Crossbar



b. Omega network



c. Omega network switch box

6.8 多处理器网络拓扑简介

- 网络特征

- 性能
  - 每条消息的延迟（空载的网络）
  - 吞吐量
    - 链路带宽
    - 网络总带宽
    - 二分法带宽
  - 拥堵延误（取决于负载情况）
- 造价/成本
- 功耗
- 硅的可布线性（工艺）

6.10 多处理器测试基准和性能模型

- 并行基准测试程序

- Linpack：矩阵线性代数
- SPECrate：SPEC CPU程序的并行运行
  - 作业级并行性
- SPLASH：斯坦福共享内存并行应用程序
  - 内核和应用程序的混合，强大的可扩展性
- NAS（NASA高级超级计算）套件
  - 计算流体动力学内核
- PARSEC（普林斯顿共享内存计算机应用程序存储库）套件
  - 使用Pthreads和OpenMP的多线程应用程序

6.10 多处理器测试基准和性能模型

- 代码 或 应用程序
- 传统基准测试程序
  - 固定代码和数据集
- 并行编程正在发展
  - 算法、编程语言和工具应该是系统的一部分吗?
  - 比较系统，前提是它们实现了给定的应用程序
  - 例如，Linpack、Berkeley设计模式
- 将促进并行方法的创新

基准测试程序	可扩展性	可编程性	描述
Linpack	弱	是	稠密矩阵线性代数 [Dongarra，1979]
SPECrate	弱	否	独立任务并行化 [Henning，2007]
面向共享存储器的斯坦福并行应用SPLASH2 [Woo et al., 1995]	强（虽然提供了两种问题规模）	否	复杂1D FFT 模块化LU分解 模块化稀疏丘拉斯基分解 整数基数排序 巴尔内斯小屋 适应性快速多极算法 海洋仿真 光线跟踪 声音渲染器 空间数据结构的水仿真 非空间数据结构的水仿真
NAS并行基准测试程序 [Bailey et al., 1991]	弱	是（只能是C或Fortran）	EP：令人尴尬的并行内核 MG：简化的多重网格计算 CG：面向共轭梯度方法的非结构化网格 FT：使用FFT的3D偏微积分方程 IS：大型整数排序
PARSEC 基准测试程序集 [Bienia et al., 2008]	弱	否	Blackscholes：使用毕苏期权定价模式的期权定价 Bodytrack：人体跟踪 Canneal：使用cache感知的模拟退火进行布线优化 Dedup：采用数据去重的下一代压缩 Facesim：人脸运动仿真 Ferret：内容相似性搜索服务器 Fluidanimate：SPH方法的流体动力学动画 Freqmine：常见物品集合的数据挖掘 Streamcluster：输入流的在线分类 Swaptions：交换组合的定价 Vips：图像处理 X264：H.264视频编码
伯克利设计数据集 [Asanovic et al., 2006]	强或弱	是	有限状态自动机 组合逻辑 图的遍历 结构化网格 稠密矩阵 稀疏矩阵 波谱法 动态程序设计 N体问题 MapReduce 反向跟踪/分支与边界 图模型推导 非结构化网格

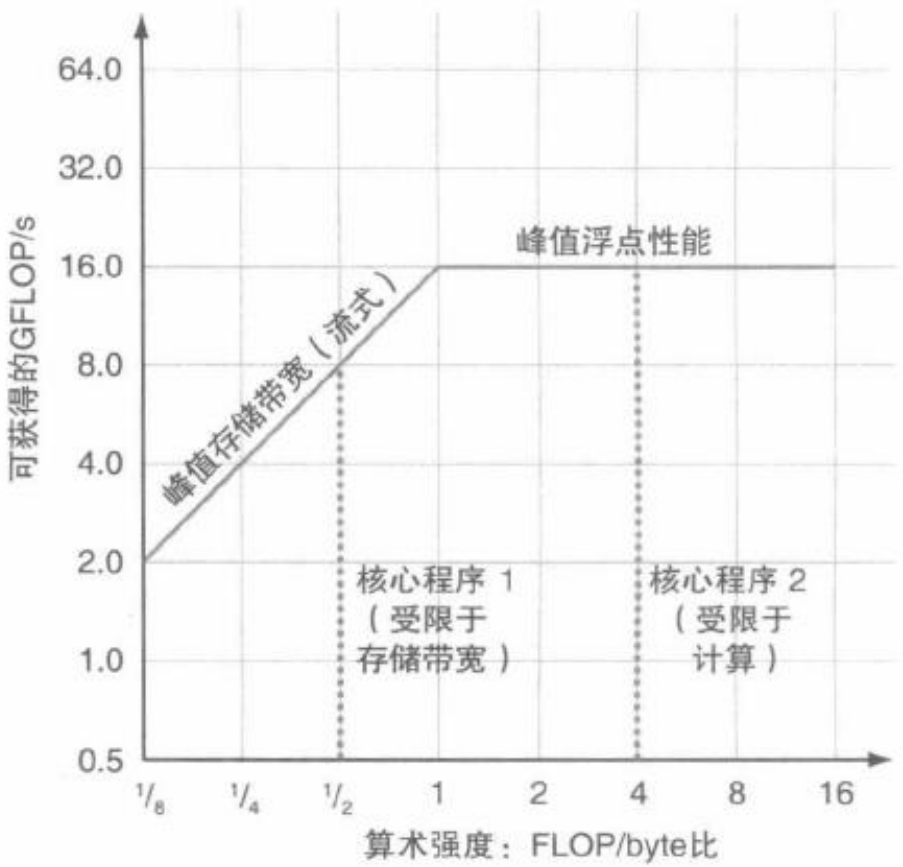
6.10 多处理器测试基准和性能模型

- 性能建模

- 假设感兴趣的性能指标是可实现的GFLOPs/sec
  - 使用伯克利设计模式的计算内核进行测量
- 核的算术强度
  - 一个程序中浮点操作数量与访问内存字节数量的比值
  - 完成的FLOPs次数/访存主存的字节数
- 对于给定的计算机，以下参数是确定的
  - 峰值GFLOPS（来自数据手册）
  - 峰值内存字节/秒（使用流基准）

6.10 多处理器测试基准和性能模型

- Roofline模型



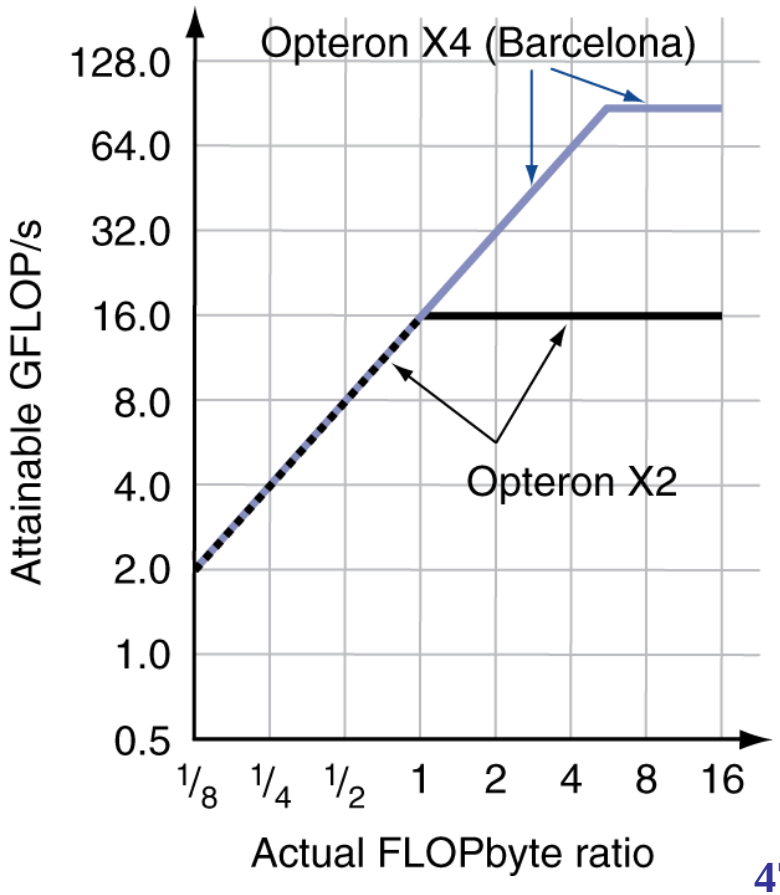
Attainable GPLOPs/sec
= Max (Peak Memory BW × Arithmetic Intensity, Peak FP Performance)

6.10 多处理器测试基准和性能模型

- 系统PK

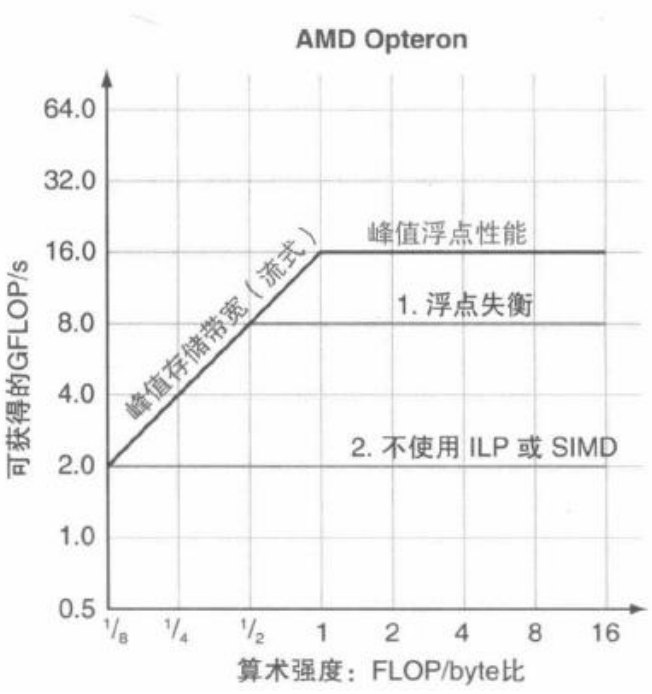
- 示例：Opteron X2与Opteron X4（AMD）
  - 2核vs4核
  - 2×FP性能/核
  - 2.2GHz vs.2.3GHz
  - 相同的存储系统

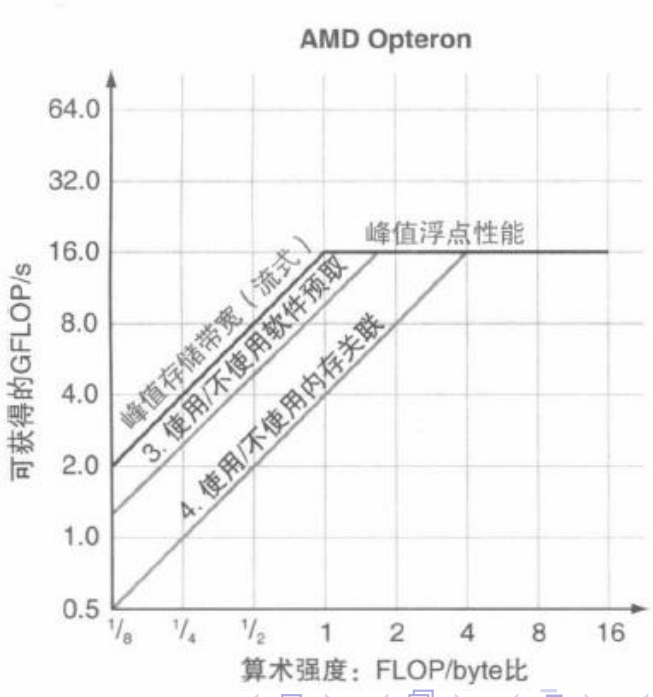
- 在X4上获得比X2更高的性能
  - 需要高算术强度
  - 或者工作组必须适合X4的2MB L-3缓存



6.10 多处理器测试基准和性能模型

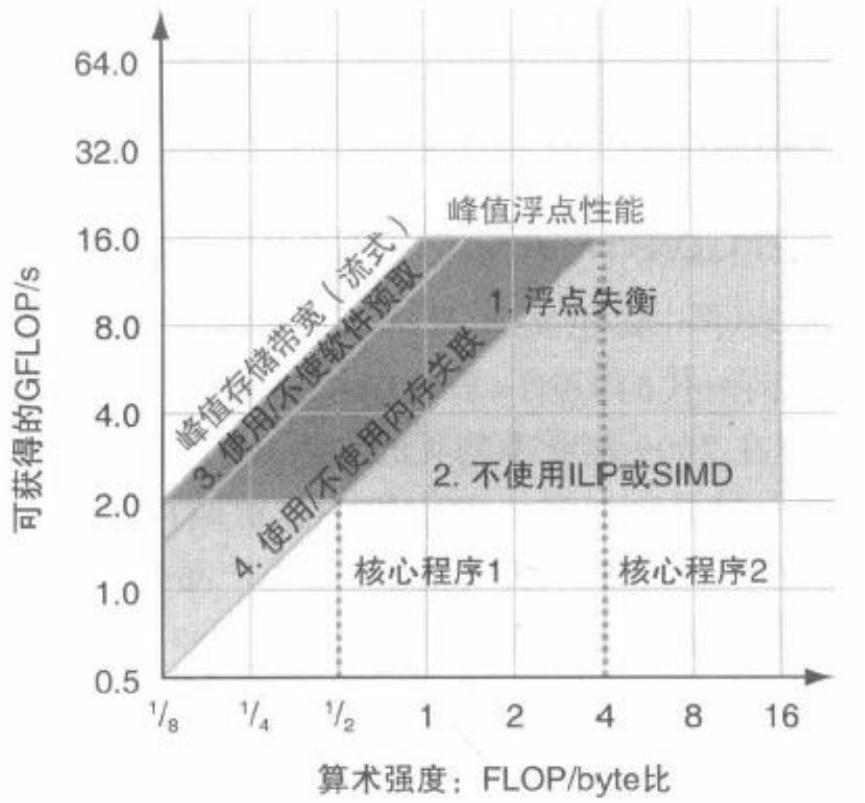
- 优化性能
  - 均衡分配乘法加法操作
  - 改进超标量ILP和SIMD指令的使用
- 优化内存使用
  - 软件预取：避免负载暂停
  - 内存关联：避免非本地数据访问





6.10 多处理器测试基准和性能模型

- 优化性能-续
- 优化的选择取决于代码的算术强度
- 算术强度并不总是固定的
  - 可能会随着问题的大小而变化
  - 缓存减少了内存访问
    - 增加算术强度



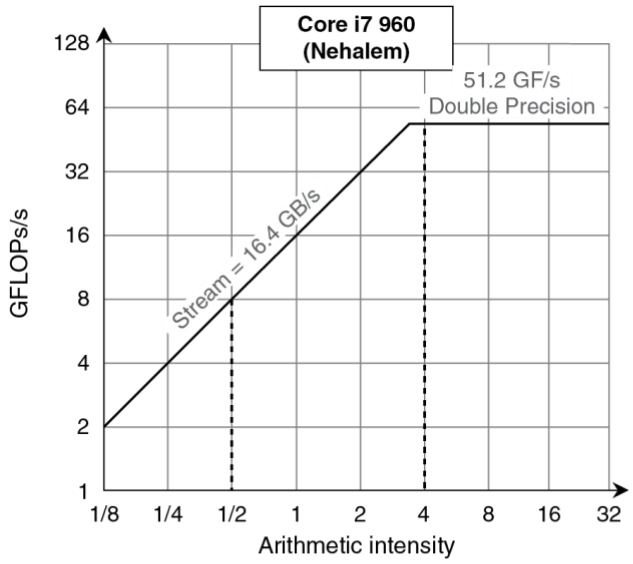
6.11 实例：评测Intel Core i7和NVIDIA Tesla GPU

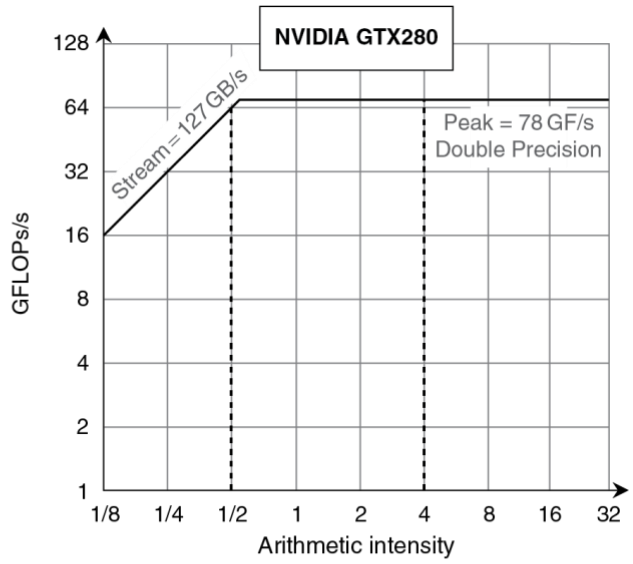
- i7-960 vs. NVIDIA Tesla 280/480

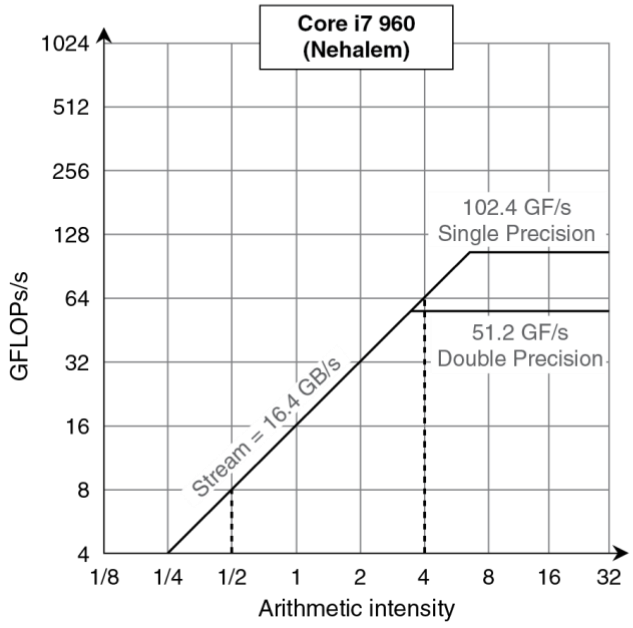
	Core i7-960	GTX 280	GTX 480	比值280/i7	比值480/i7
处理单元的数量（核或流式微处理器数）	4	30	15	7.5	3.8
时钟频率（GHz）	3.2	1.3	1.4	0.41	0.44
晶片尺寸	263	576	520	2.2	2.0
工艺	Intel 45nm	TSMC 65nm	TSMC 40nm	1.6	1.0
功耗（芯片，不是模组）	130	130	167	1.0	1.3
晶体管数量	700 M	1400 M	3030 M	2.0	4.4
内存带宽（GByte/s）	32	141	177	4.4	5.5
单精度SIMD宽度	4	8	32	2.0	8.0
双精度SIMD宽度	2	1	16	0.5	8.0
单精度向量运算峰值性能（GFLOP/s）	26	117	63	4.6	2.5
单精度SIMD运算峰值性能（GFLOP/s）	102	311～933	515 或 1344	3.0～9.1	6.6～13.1
（SP 1中加法器或乘法器）	不支持	(311)	(515)	(3.0)	(6.6)
（SP 1中融合乘加部件）	不支持	(622)	(1344)	(6.1)	(13.1)
（双发射SP包括融合的乘加部件和乘法部件）	不支持	(933)	不支持	(9.1)	—
双精度SIMD 运算峰值性能（GFLOP/s）	51	78	515	1.5	10.1

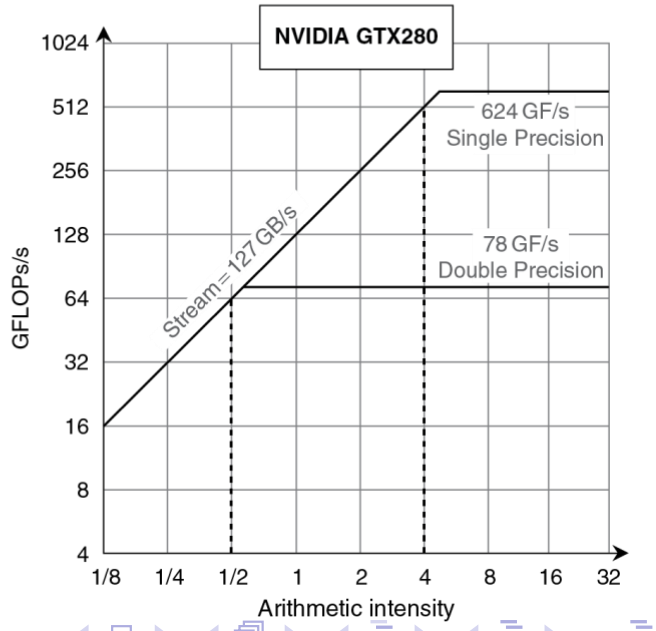
6.11 实例：评测Intel Core i7和NVIDIA Tesla GPU

- Rooflines图









6.11 实例：评测Intel Core i7和NVIDIA Tesla GPU

- 基准测试程序

核心程序	单位	Core i7-960	GTX 280	GTX 280 i7-960
SGEMM	GFLOP/s	94	364	3.9
MC	十亿条路径/秒	0.8	1.4	1.8
Conv	百万像素/秒	1250	3500	2.8
FFT	GFLOP/s	71.4	213	3.0
SAXPY	GByte/秒	16.8	88.8	5.3
LBM	百万次查找/秒	85	426	5.0
Solv	帧/秒	103	52	0.5
SpMV	GFLOP/s	4.9	9.1	1.9
GJK	帧/秒	67	1020	15.2
Sort	百万个元素/秒	250	198	0.8
RC	帧/秒	5	8.1	1.6
Search	百万次查询/秒	50	90	1.8
Hist	百万像素/秒	1517	2583	1.7
Bilat	百万像素/秒	83	475	5.7

6.11 实例：评测Intel Core i7和NVIDIA Tesla GPU

- 性能总结

- GPU（480）拥有4.4倍的内存带宽
  - 内存受限内核（程序）会有更大的性能提升
- GPU的吞吐量，单精度达13.1倍，双精度达2.5倍
  - 得益于计算绑定内核（程序）
- CPU缓存可以防止内核（程序）受到内存限制
  - GPU则不然
- GPU提供“分散-聚集”功能，这有助于处理具有“跳跃式”数据结构的内核
- GPU上缺乏同步和内存一致性支持限制了某些内核（程序）的性能

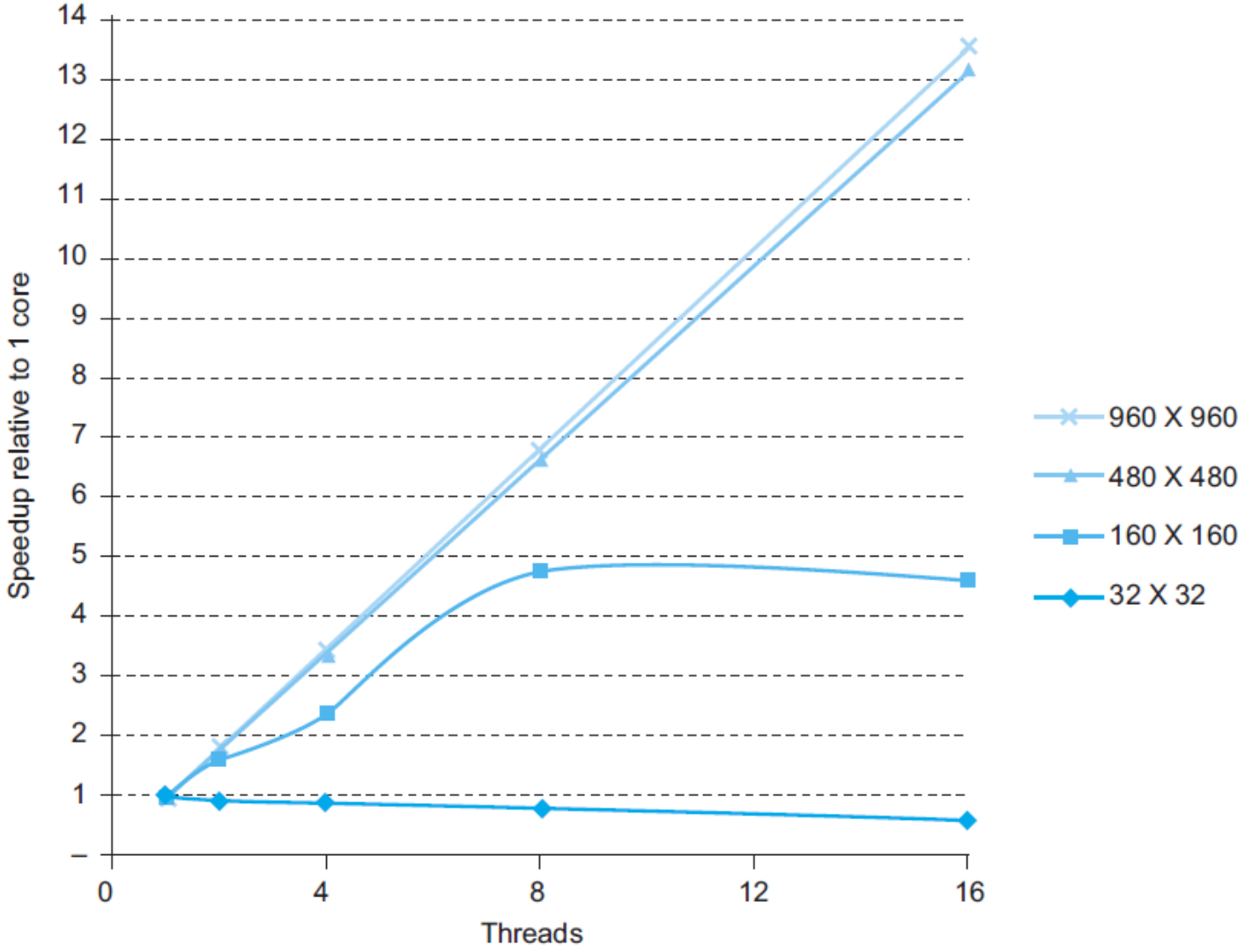
6.12 加速：多处理器和矩阵乘法

- 多线程DGEMM
- 使用OpenMP

```
void dgemm (int n, double* A, double* B, double* C)
{
#pragma omp parallel for
 for ( int sj = 0; sj < n; sj += BLOCKSIZE )
  for ( int si = 0; si < n; si += BLOCKSIZE )
   for ( int sk = 0; sk < n; sk += BLOCKSIZE )
    do_block(n, si, sj, sk, A, B, C);
}
```

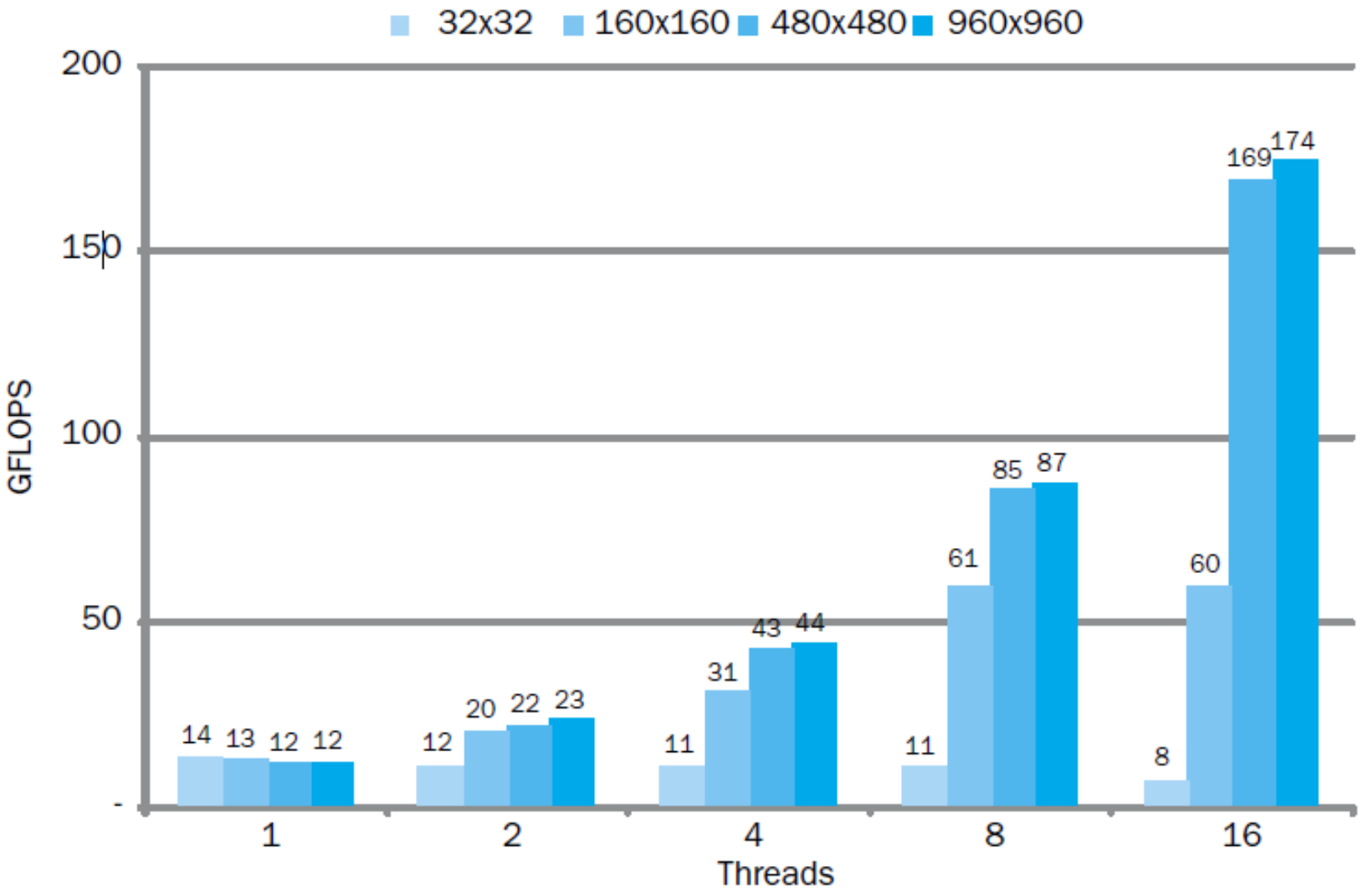
6.12 加速：多处理器和矩阵乘法

- 多线程DGEMM



6.12 加速：多处理器和矩阵乘法

- 多线程DGEMM



6.13 谬误与陷阱

- 谬误

- 阿姆达尔定律不适用于并行计算机
  - 因为我们可以实现线性加速
  - 但仅适用于伸缩性较弱的应用程序
- 峰值性能可代表实际性能
  - 营销人员喜欢这种方法!
  - 但将Xeon与示例中的其他产品进行比较
  - 需要注意瓶颈

6.13 谬误与陷阱

- 陷阱

- 开发软件时不考虑利用和优化多处理器体系结构
  - 示例：对共享复合资源使用单个锁
    - 序列化访问，即使它们可以并行完成
    - 使用更细粒度的锁定

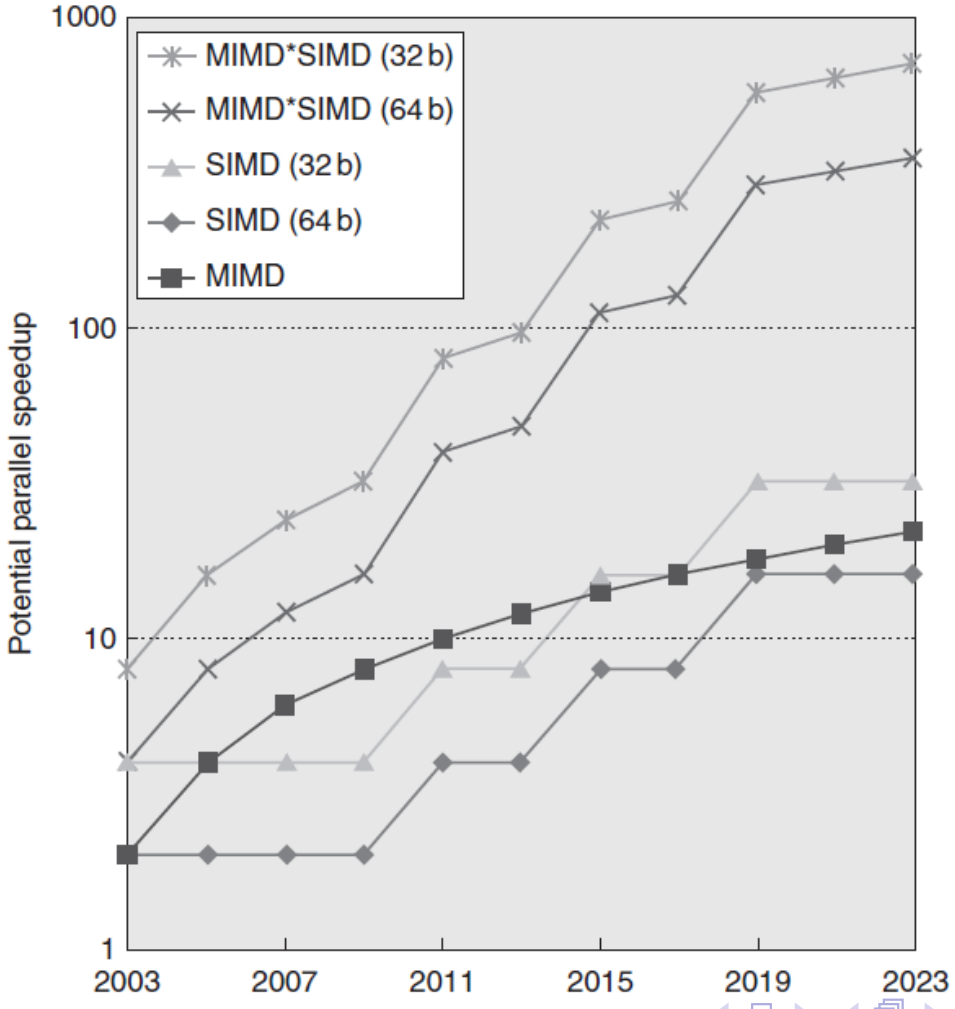
6.14 本章小结

- 结束语

- 目标：通过使用多个处理器提高性能
- 困难
  - 开发并行软件
  - 设计合适的架构
- SaaS的重要性与日俱增，集群是一个很好的匹配
- 移动和WSC的每美元性能和每焦耳性能

6.14 本章小结

- 结束语-续
- SIMD和矢量运算与多媒体应用程序相匹配，易于编程



中国科学技术大学
University of Science and Technology of China

计算机组成原理
CH7_中断与异常

卢建良
lujl@ustc.edu.cn
2022年春季学期

提纲

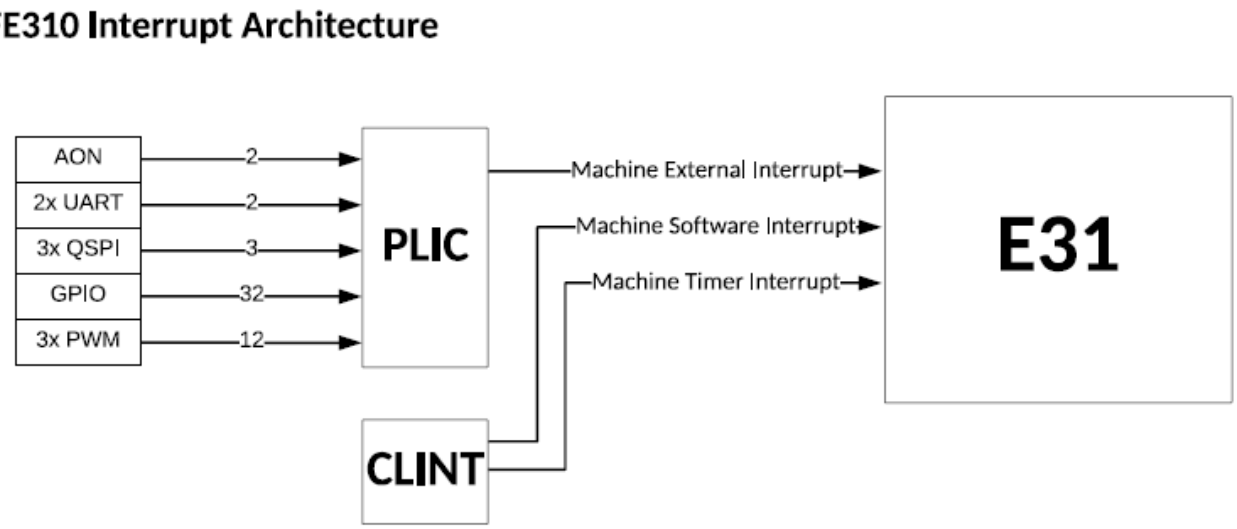
- 中断异常相关理论知识
  - 主要参考唐朔飞第三版教材，Ch5、Ch8
- MIPS、ARM、RISC-V异常处理相关实现
- RISC-V异常相关指令及寄存器
- RISC-V异常处理实例代码

中断系统

- 中断和异常本身不是指令，却是处理器架构中非常重要的一环
- 中断（唐书 P192）
  - 计算机在执行程序的过程中，当出现异常情况或特殊请求时，计算机停止现行程序的运行，转向对这些异常情况或特殊请求的处理，处理结束后，再返回到现行程序的间断处，继续执行原程序，这就是“中断”。
- 中断的作用
  - 异常：响应软硬件错误或者故障
  - I/O：响应外设的中断
    - 系统与环境交互，实时响应外部事件
  - 并发：提高计算机的整机效率
    - 单核多任务并发
    - 多核多任务并发
  - 服务：用户程序与OS间交互，Trap（陷阱）
    - 一种提供系统服务和保护的机制

中断系统

- 中断管理
  - 中断服务程序 ISR， Interrupt Service Routines
- RISC-V中，中断与异常区别
  - 异常是由处理器内部时间或程序执行中的事件引起的（如硬件故障、程序故障），或者执行特定的系统服务指令（如ecall），是内部因素引起的。
  - 中断往往是由外部因素引起的，RISC-V定义了4类中断
    - 外部中断，如串口中断
    - 计时器中断
    - 软件中断
    - 调试中断
- RISC-V中断架构

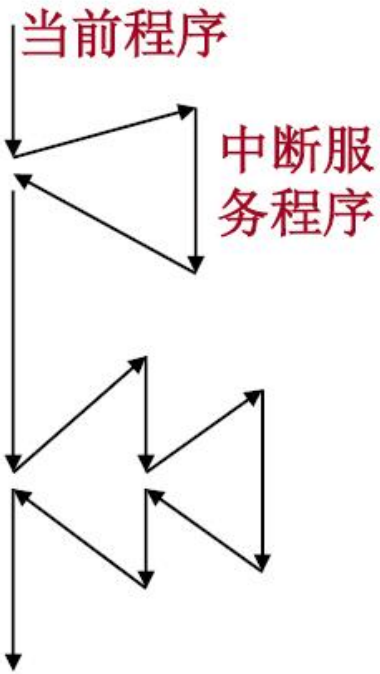


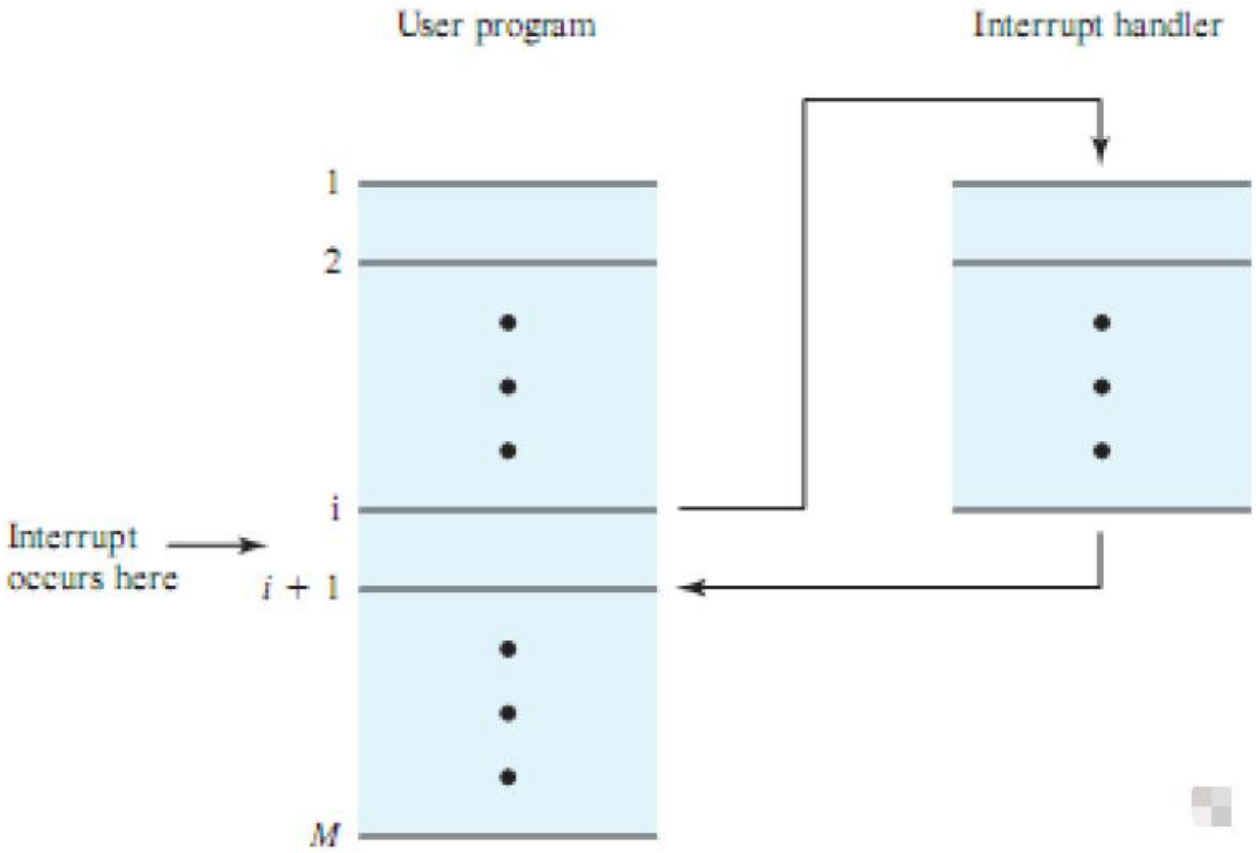
中断系统

- X86系统的Interrupt
  - 外部中断：硬中断
    - 可屏蔽中断
    - 不可屏蔽中断
  - 内部中断：软中断
    - 程序异常
    - 系统调用：INT x
    - 指令断点
- RISC系统（MIPS）的Exceptions
  - 外部事件：I/O中断
  - 异常：软、硬件故障
  - 陷阱：syscall、断点break，自陷TEQ

中断的行为

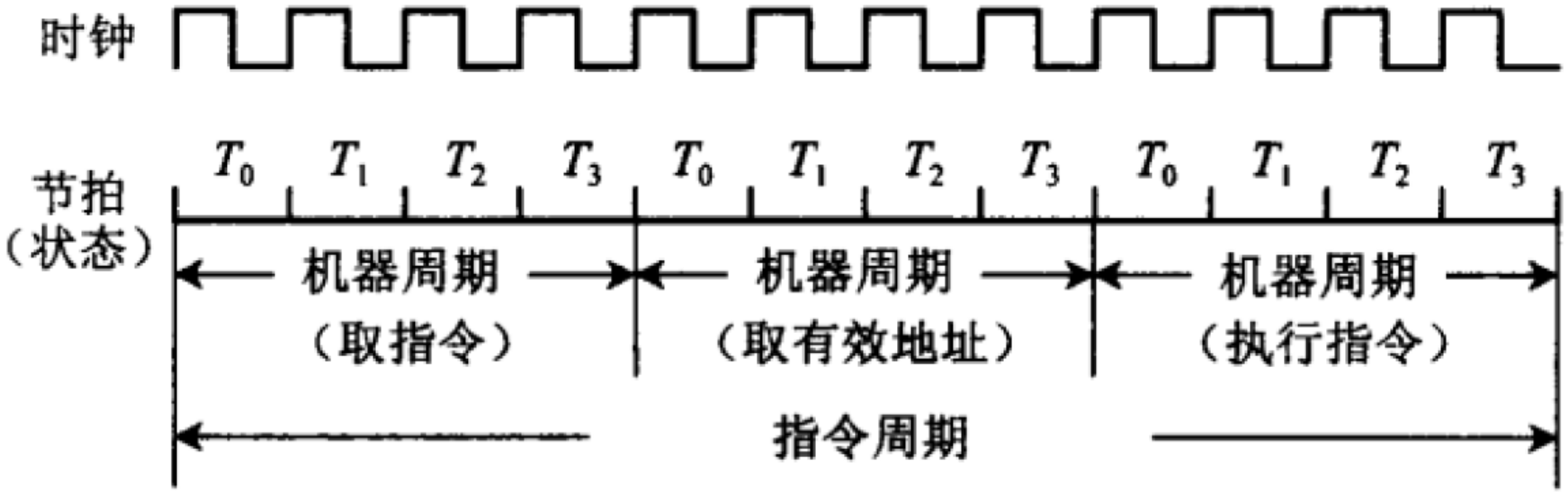
- 过程调用
- 对CPU控制的转移（当前程序=>中断服务程序）
- 对计算资源的并发使用





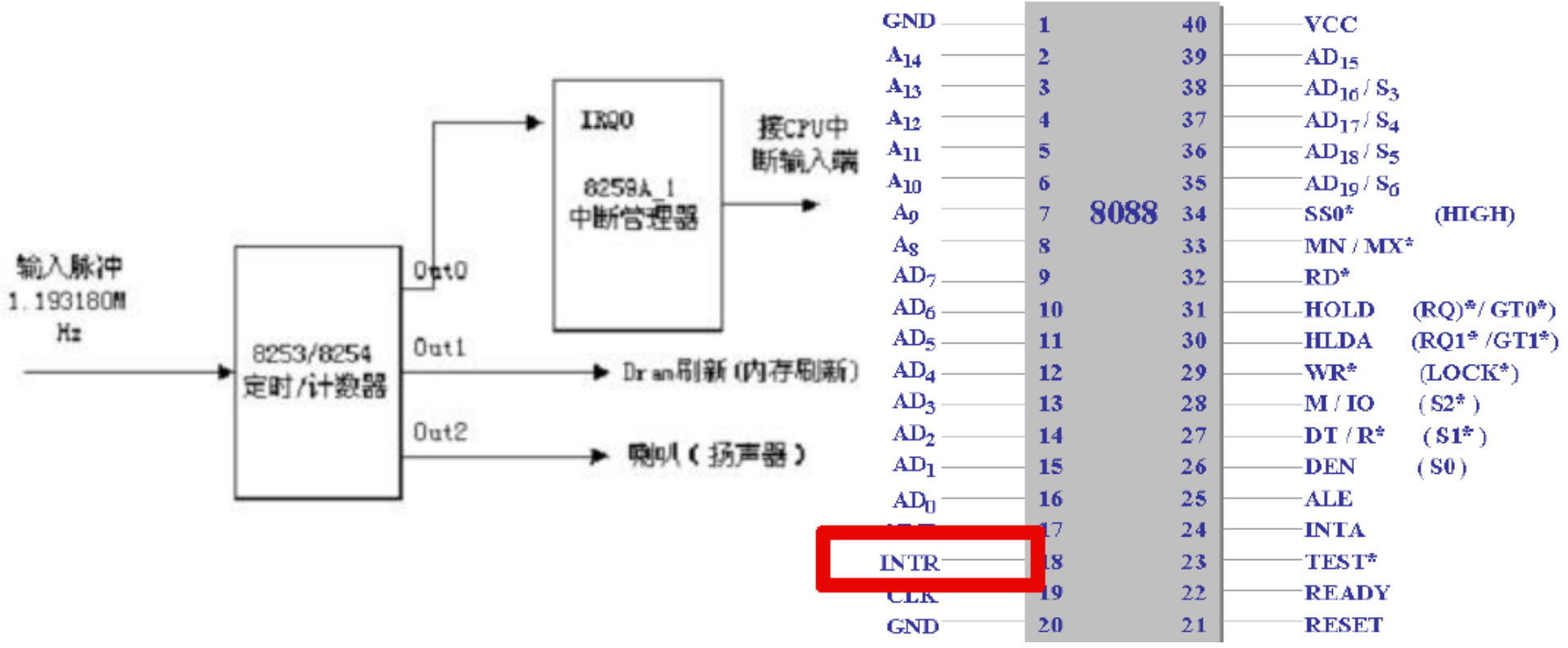
中断、异常发生的时机

- I/O中断（Interrupts），随时发生，延迟响应
- 异常（Exceptions），随时发生，随时处理
- 陷阱（Traps），专用指令，特殊处理



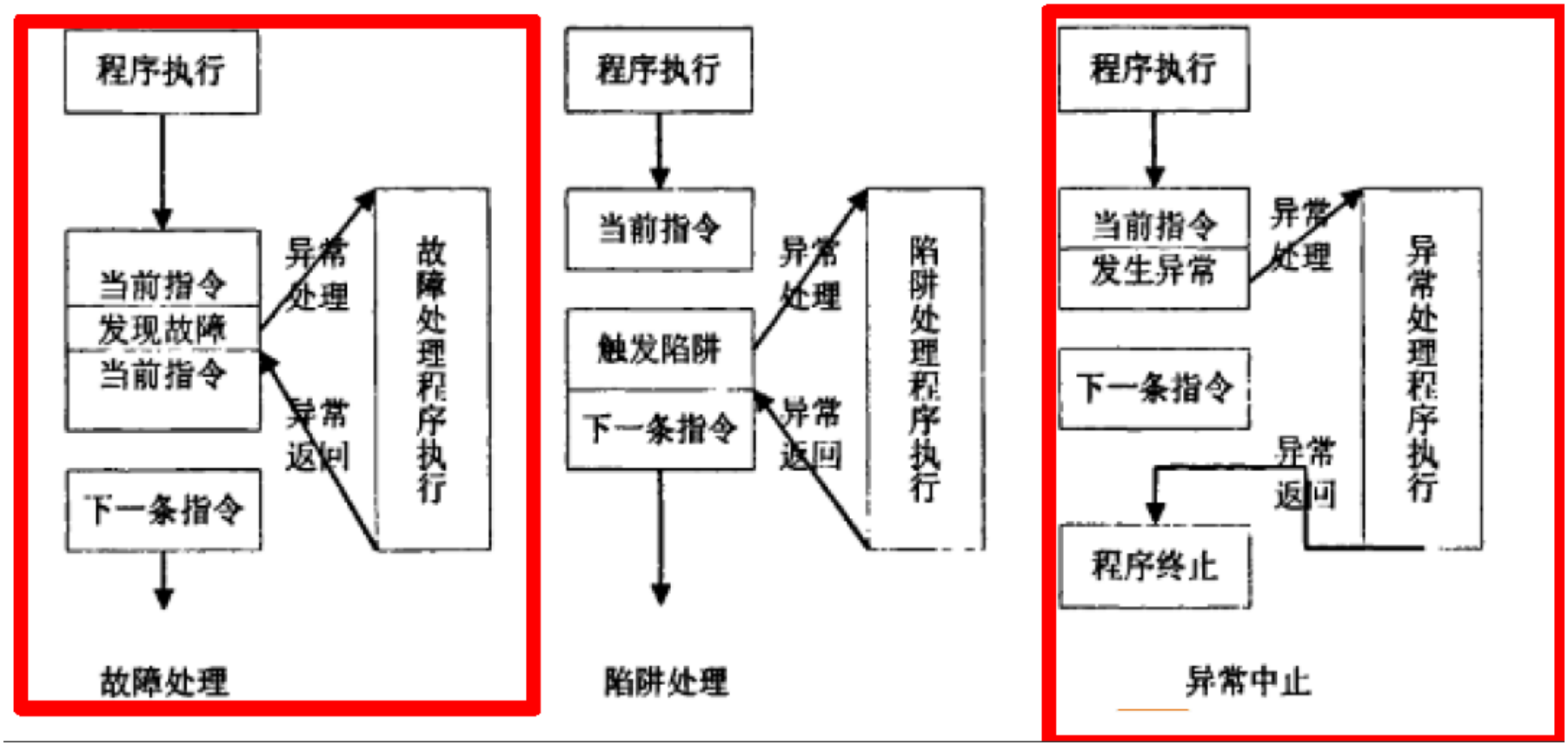
例：时钟中断

- 由可编程定时/计数器产生的INT
  - 维持系统时间（大约每10ms更新实时时钟）
  - 时钟中断维持系统时间、促使环境切换，以保证所有进程共享CPU

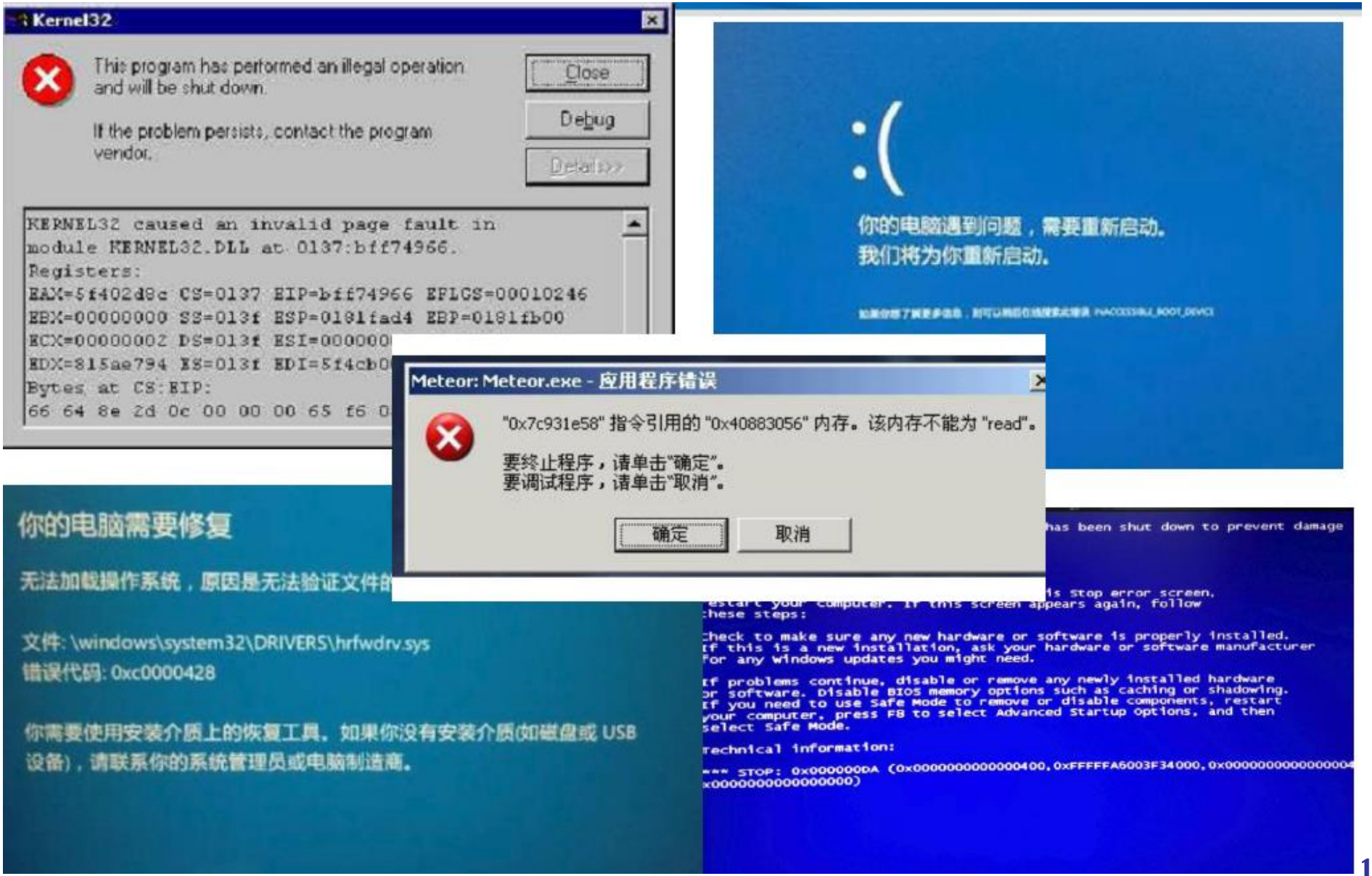


例：异常处理

- 服务：将控制转移给OS的异常处理程序
  - 可恢复异常（下图中的“故障处理”，OS处理后返回）
  - 不可恢复异常（下图中的“异常中止”）

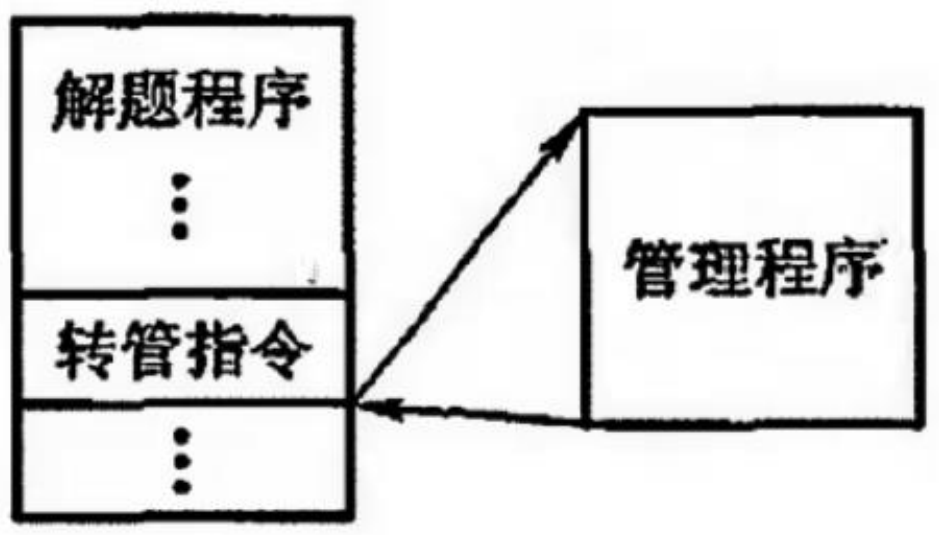


例：不可恢复异常：异常中止



引起中断的因素

- 人为设置的中断
  - eg：转管指令
- 程序异常
  - 溢出
  - 无效指令
  - 除零
- 硬件故障
- I/O设备发起的中断请求
- 外部事件
  - eg：用键盘中断现行程序



中断系统需要解决的问题

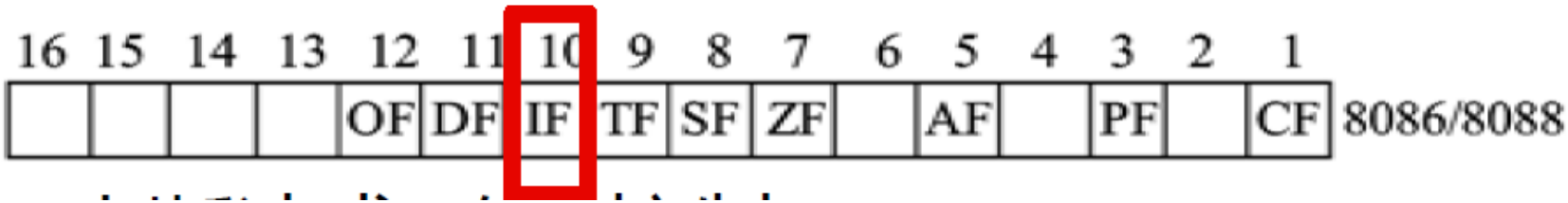
- 1. 各中断源如何向CPU提出请求?
- 2. 各中断源同时提出请求怎么办?
- 3. CPU什么条件、什么时间、以什么方式响应中断?
- 4. 如何保护现场？如何恢复现场？如何返回?
- 5. 如何寻找入口地址?
- 6. 处理中断的过程中，又出现新的中断怎么办?

- 解决方案：硬件 + 软件

中断机制组成

- CPU中断禁止/允许：PSW寄存器中的IF字段（MIPS）

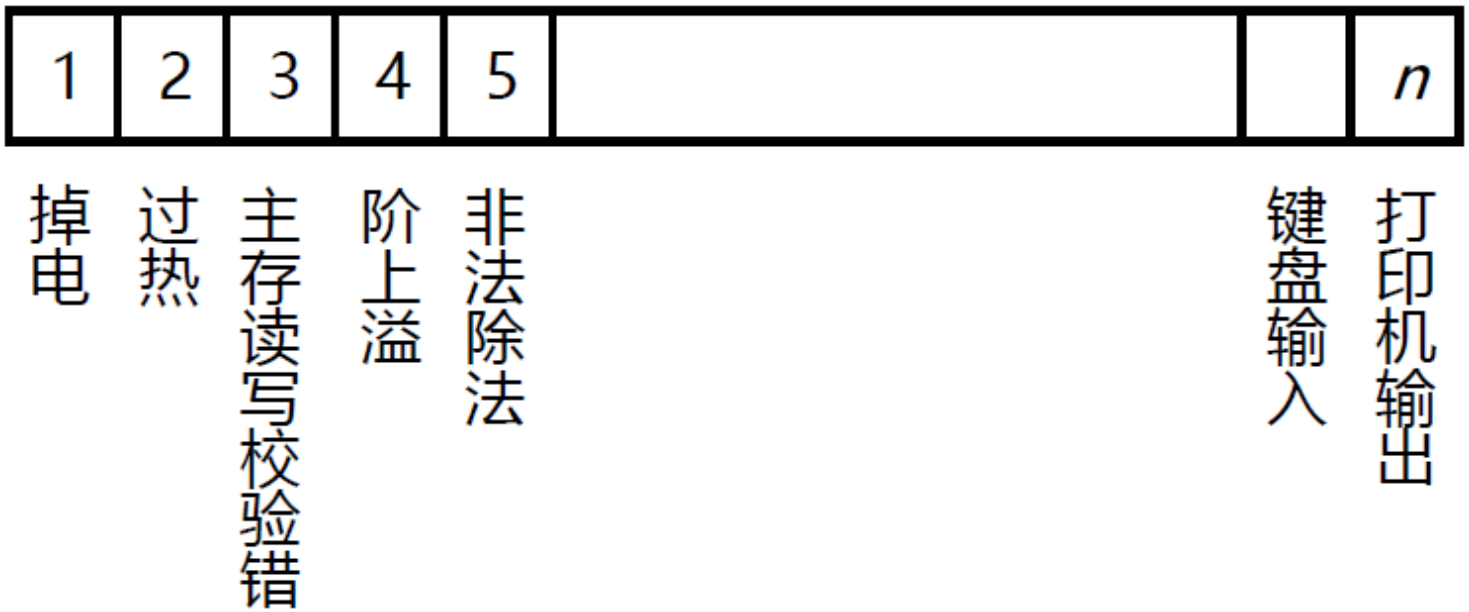
PSW即程序状态字（程序状态寄存器），Program Status Word。



- CPU中断请求/响应控制：INTR、INTA
- 中断响应/返回：中断隐指令
- 断点/现场保存：MEM（stack）
- 中断服务
  - 中断源识别/判定优先级：中断控制器
  - ISR入口：向量方式、非向量方式

1. 各中断源如何向CPU提出请求?

- 中断请求标记 INTR
  - 一个请求源对应一个INTR中断请求标记位
  - 多个INTR组成中断请求标记寄存器

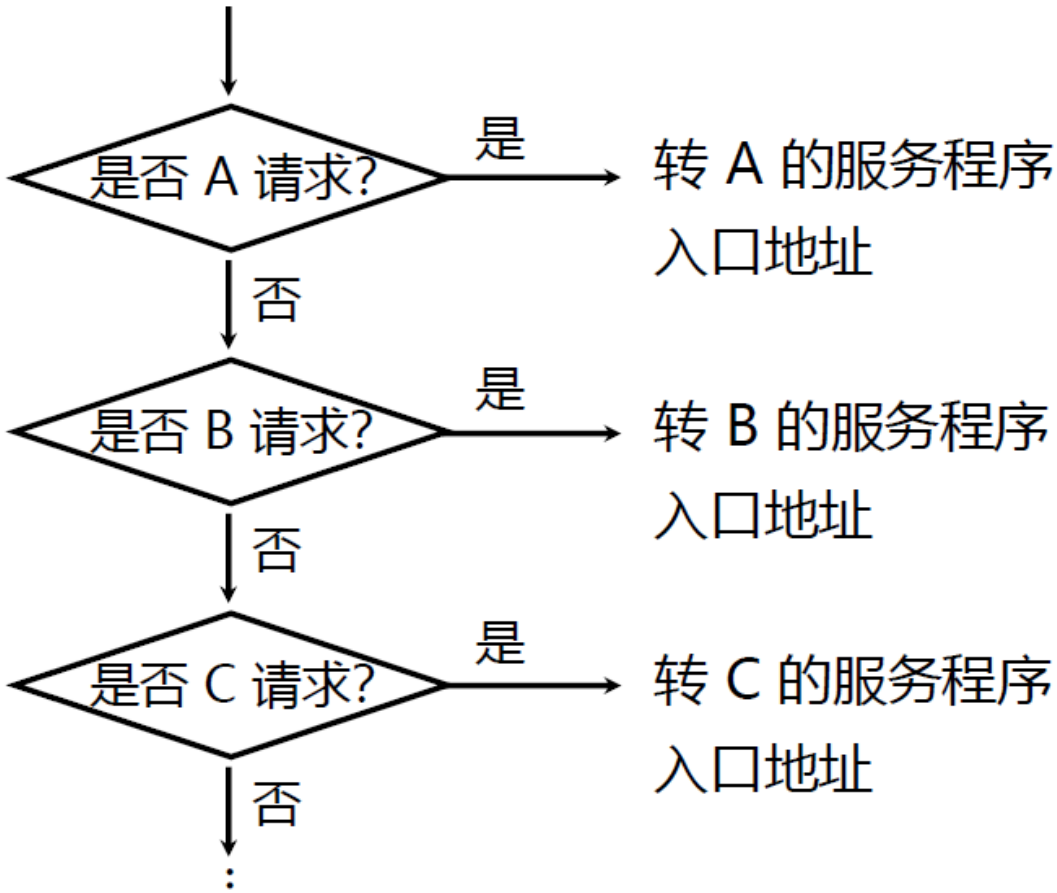


  - INTR可以分散在各个中断源的接口电路中
  - INTR也可以集中在CPU的中断系统内

2. 各中断源同时提出请求怎么办?

- 中断判定优先级逻辑
  - 软件实现（程序查询）

A、B、C 优先级按 降序 排列



2. 各中断源同时提出请求怎么办?

- 中断判定优先级逻辑
  - 硬件实现（排队器）
    - 分散在各个中断源的接口电路中的链式排队器

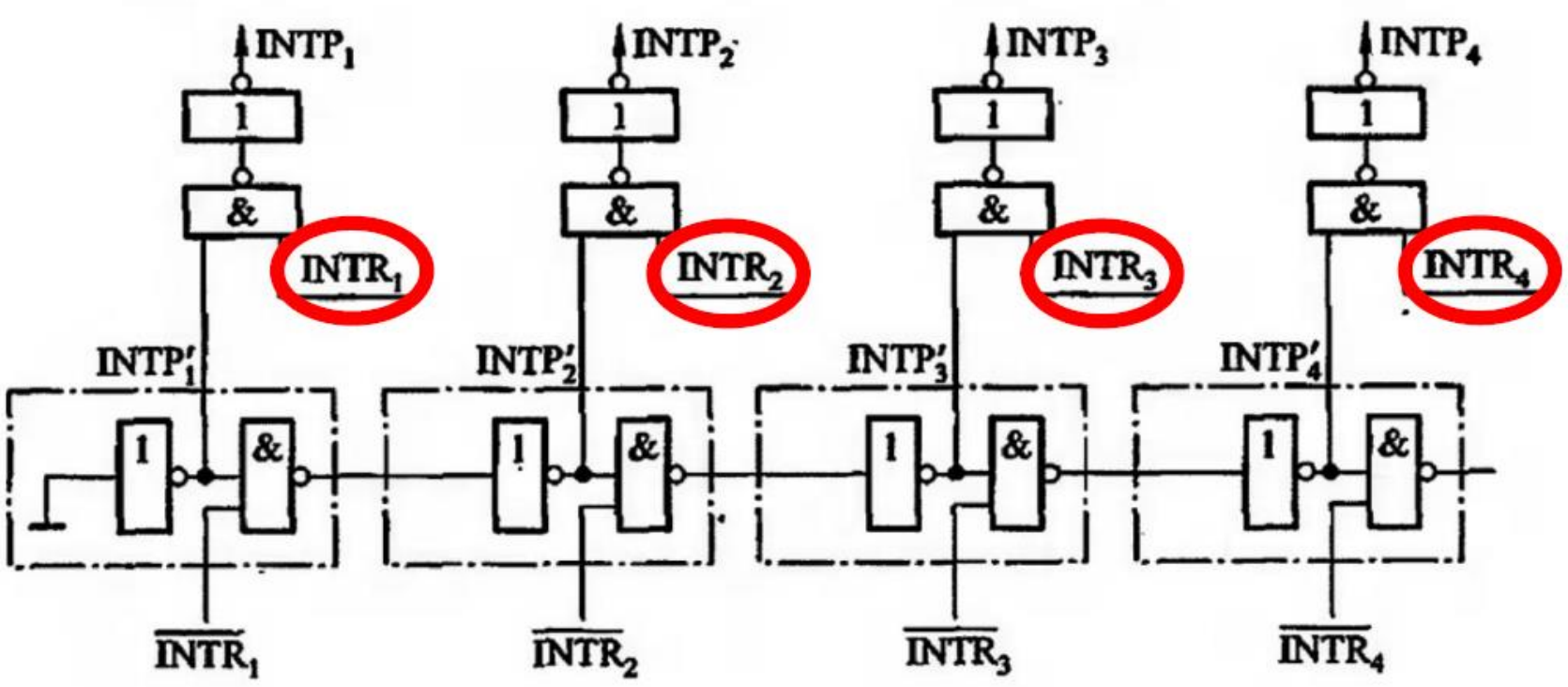
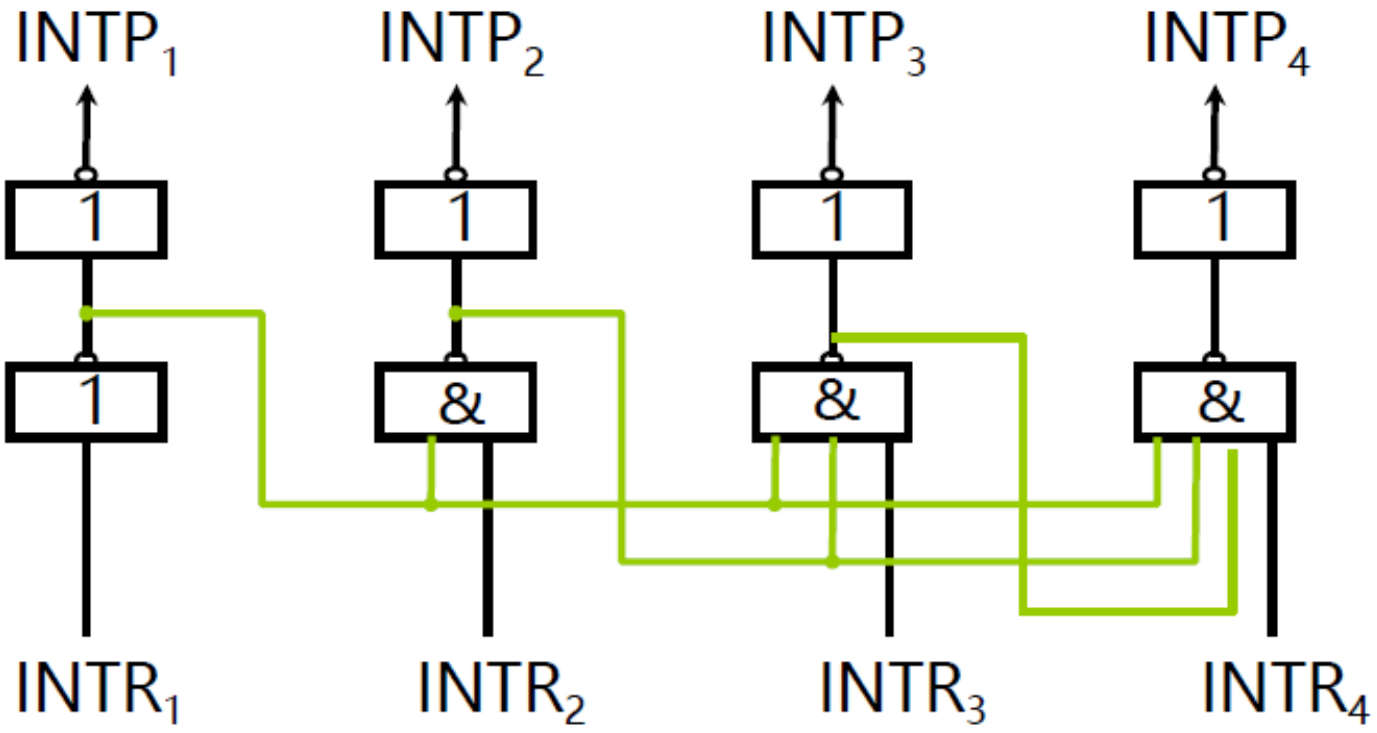


图 5.38　链式排队器

2. 各中断源同时提出请求怎么办?

- 中断判定优先级逻辑
  - 硬件实现（排队器）
    - 集中在CPU内的链式排队器



$INTR_1$、$INTR_2$、$INTR_3$、$INTR_4$ 优先级按降序排列

3. CPU什么条件、什么时间、以什么方式响应中断?

- 中断响应
  - 响应中断的条件
    - CPU允许中断触发器 EINT = 1
  - 响应中断的时间
    - 指令执行周期结束时刻，由CPU发查询信号

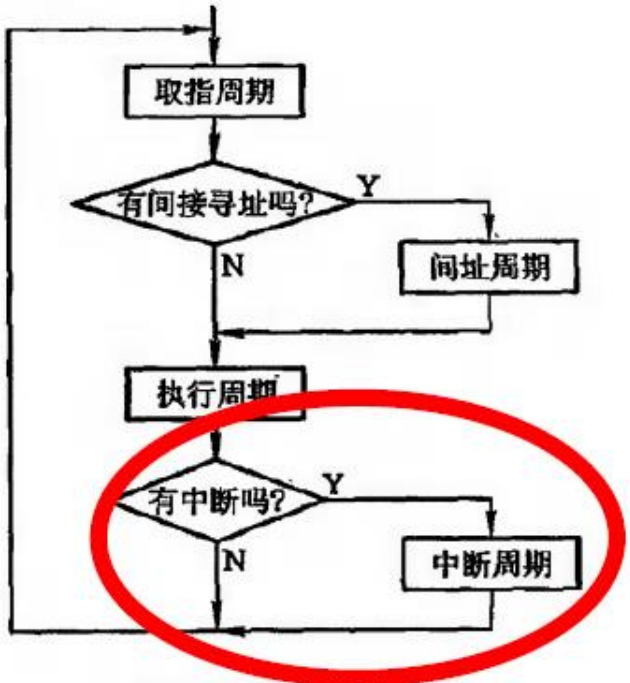


图 8.8 指令周期流程

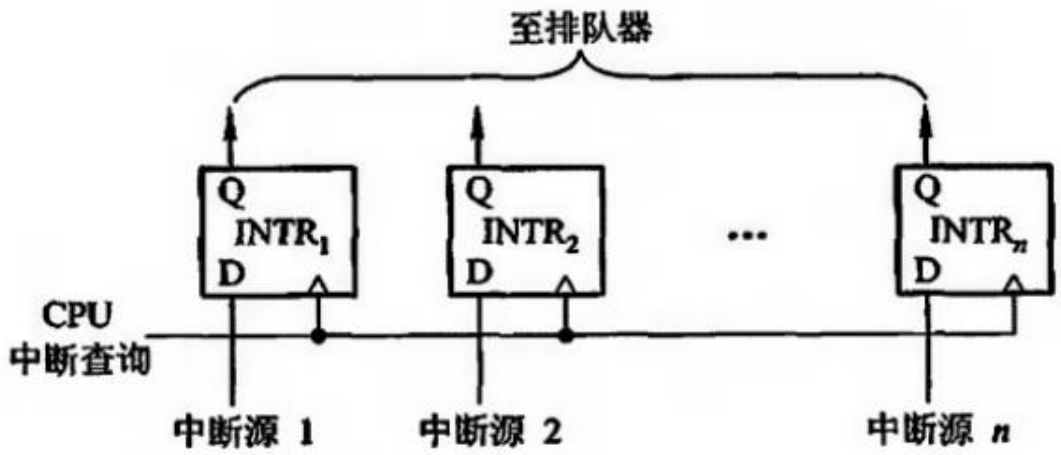


图 8.29 CPU 在统一时间发中断查询信号

3. CPU什么条件、什么时间、以什么方式响应中断?

- 中断响应-中断隐指令
  - CPU响应中断之后，经过某些操作，转去执行中断服务程序。这些操作是由 硬件直接实现的，把它称为中断隐指令
  - 中断周期完成的主要操作，通过中断隐指令完成
  - 1. 保护程序断点
    - 断点存于特定地址（如0号地址）内，或者在堆栈内
  - 2. 寻找服务程序入口地址
    - 中断识别程序入口地址M -> PC（软件查询法）
    - 向量地址 -> PC（硬件向量法）
  - 3. 硬件关中断
    - EINT 允许中断寄存器，写0
    - INT 中断标记寄存器，写1

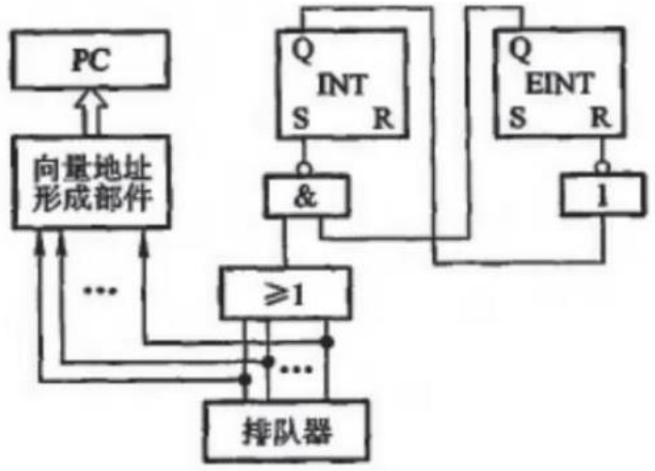
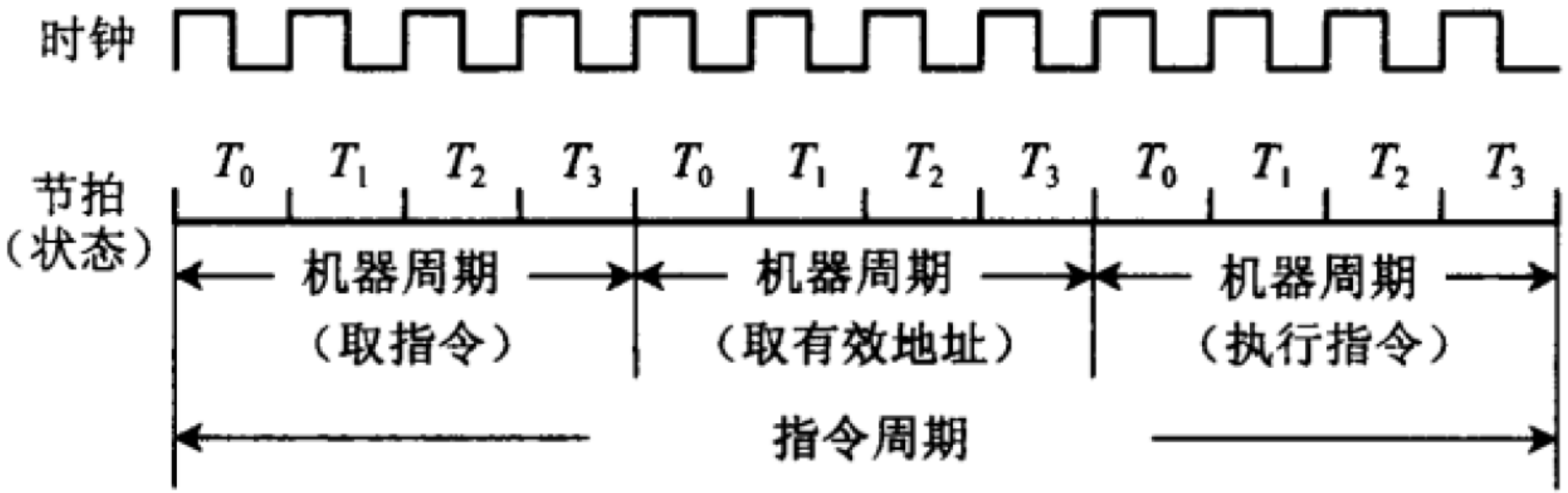


图 8.30 硬件关中断示意图

3. CPU什么条件、什么时间、以什么方式响应中断?

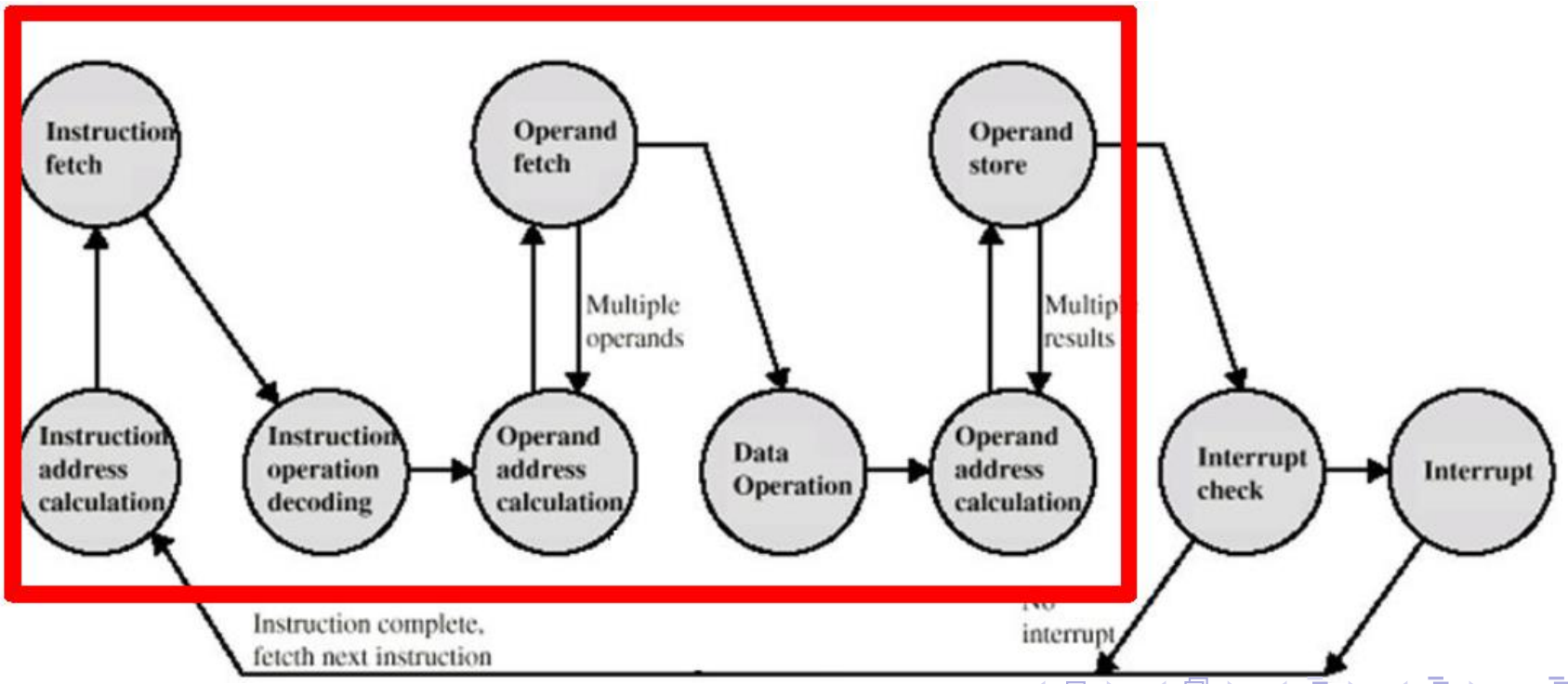
- 中断响应的时机与条件

- 中断发生与CPU响应时间
  - 外部中断（IO）：异步（延迟响应），指令周期结束
  - 内部中断（陷阱和异常）：同步（马上响应）



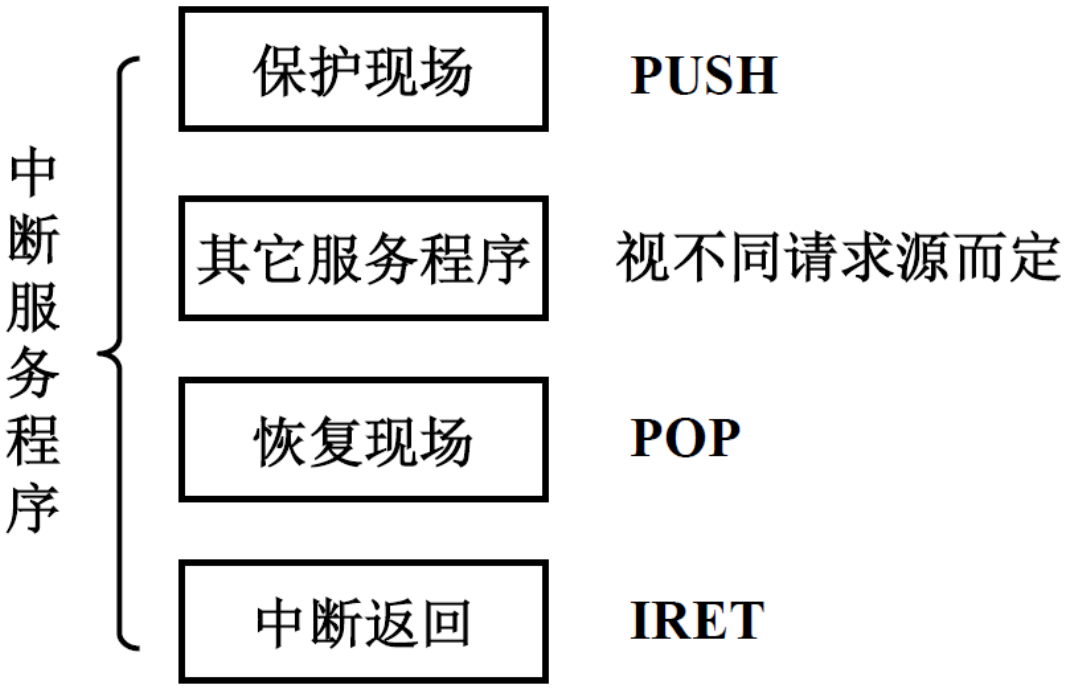
3. CPU什么条件、什么时间、以什么方式响应中断?

- 中断周期
  - 指令周期结束
  - CPU与中断控制器通信，响应外部事件
    - 判断是否有中断请求
    - 识别中断源，获得中断向量
  - 相关操作：保存PC和PSW，关中断，转ISR



4. 如何保护及恢复现场？如何返回？

- 1. 保护现场
  - 保存断点：由中断隐指令完成（由硬件完成）
  - 寄存器内容：由中断服务程序ISR完成（由软件完成）
- 2. 恢复现场

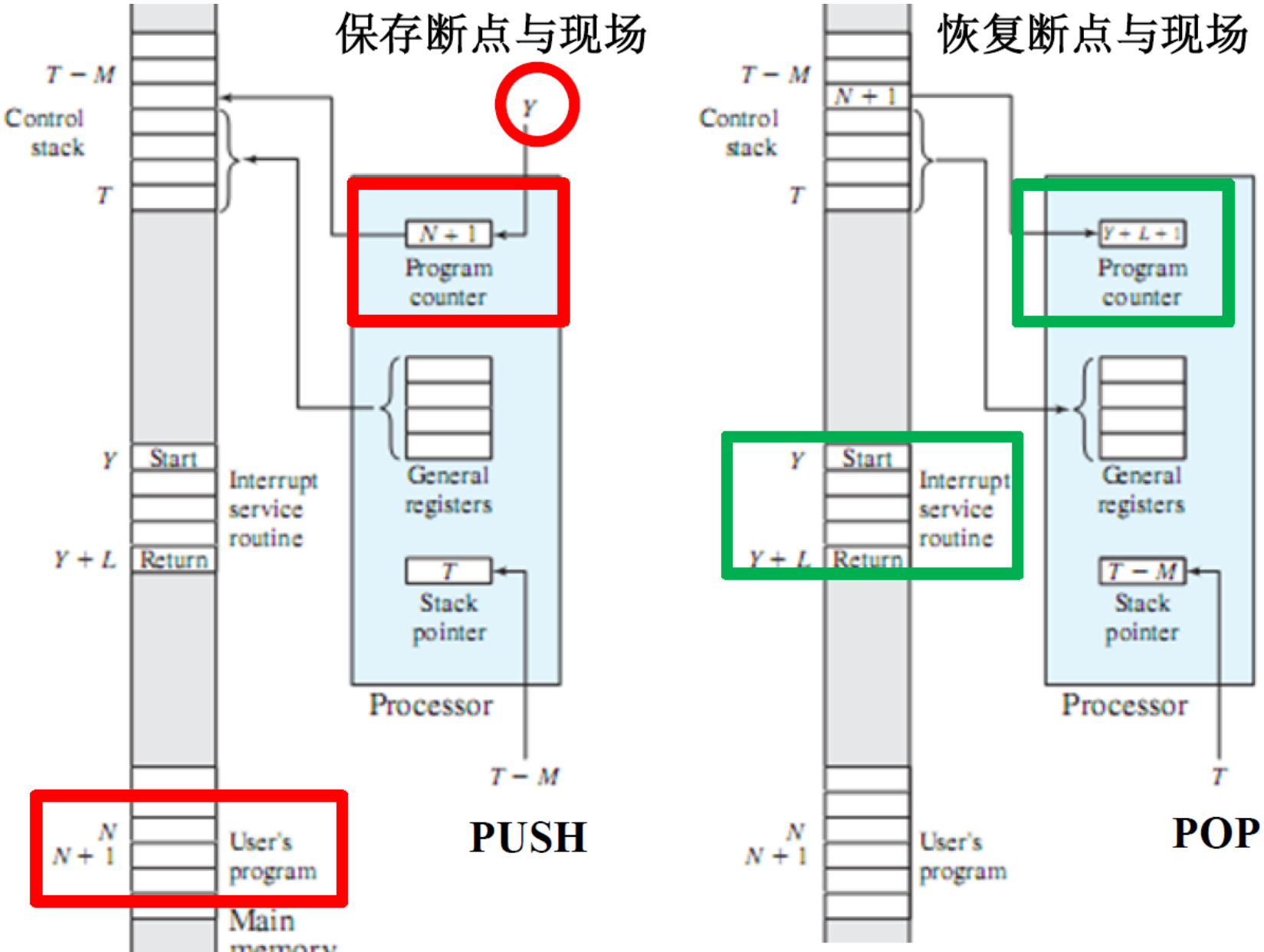



```
show proc near
 push ds
    push es
    push ax
    push cx
    push dx
    push bx
   push sp
    push bp
    push si
    push di
    sti
   mov dl,41h
    mov ah,02h
    int 21h
exit :
  cli
   pop di
    pop si
    pop bp
    pop sp
    pop bx
    pop dx
    pop cx
    pop ax
    pop es
    pop ds
      iret
show endp
```

中断服务程序ISR示例

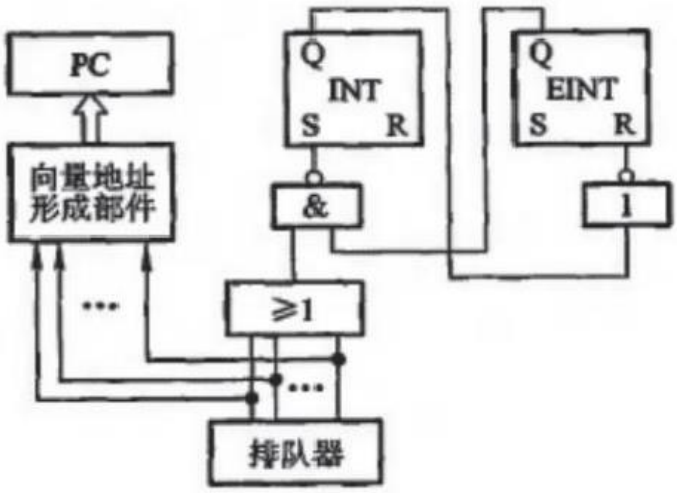
汇编用法参见 IBM PC的BIOS及DOS功能调用、微机原理及接口技术

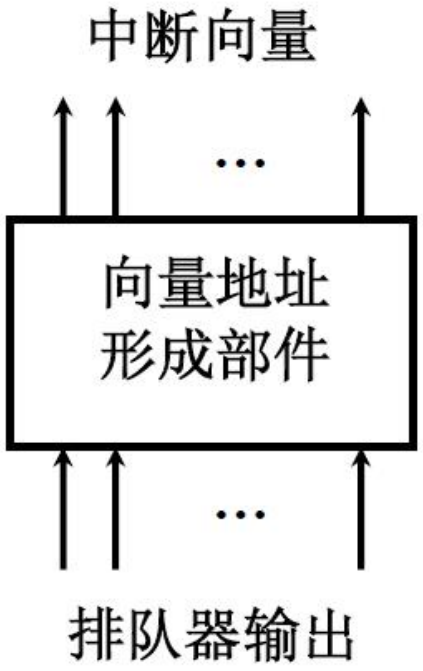
4. 如何保护及恢复现场？如何返回？



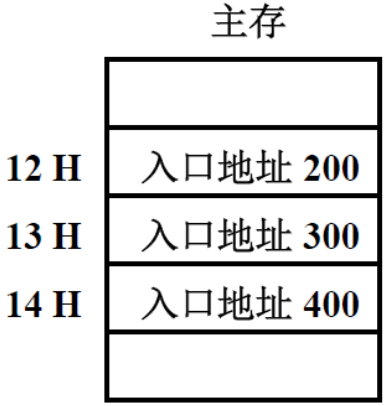
5. 如何寻找入口地址?

- 1. 硬件向量法





- 2. 软件查询法

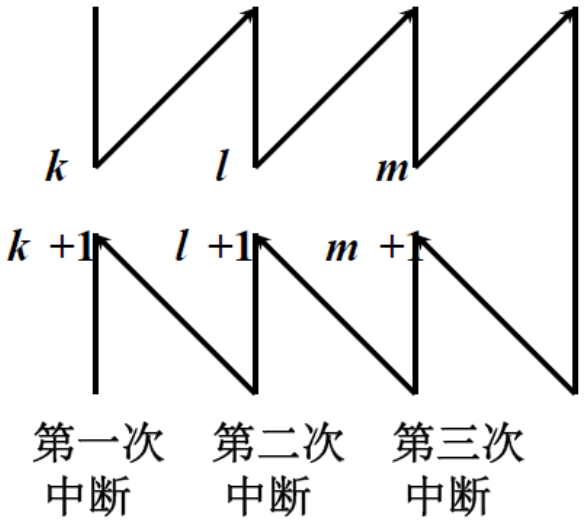


向量地址 12H、13H、14H

入口地址 200、 300、 400

6. 处理中断的过程中，又出现新的中断怎么办?

- 中断屏蔽技术
  - 主要用于多重中断的处理
- 多重中断（中断嵌套）
  - 当CPU正在执行某个中断服务程序时，另一个中断源又提出了新的中断请求，而CPU又响应了这个新的请求，暂时停止正在运行的服务程序，转去执行新的中断服务程序，这称为多重中断，又称中断嵌套。



程序断点 $k+1$，$l+1$，$m+1$

6. 处理中断的过程中，又出现新的中断怎么办？

- 实现多重中断的条件
  - 1. 提前设置开中断

中断隐指令响应，存断点之后关中断
中断服务结束之前开中断

中断隐指令响应，存断点之后关中断
中断服务开始之后开中断

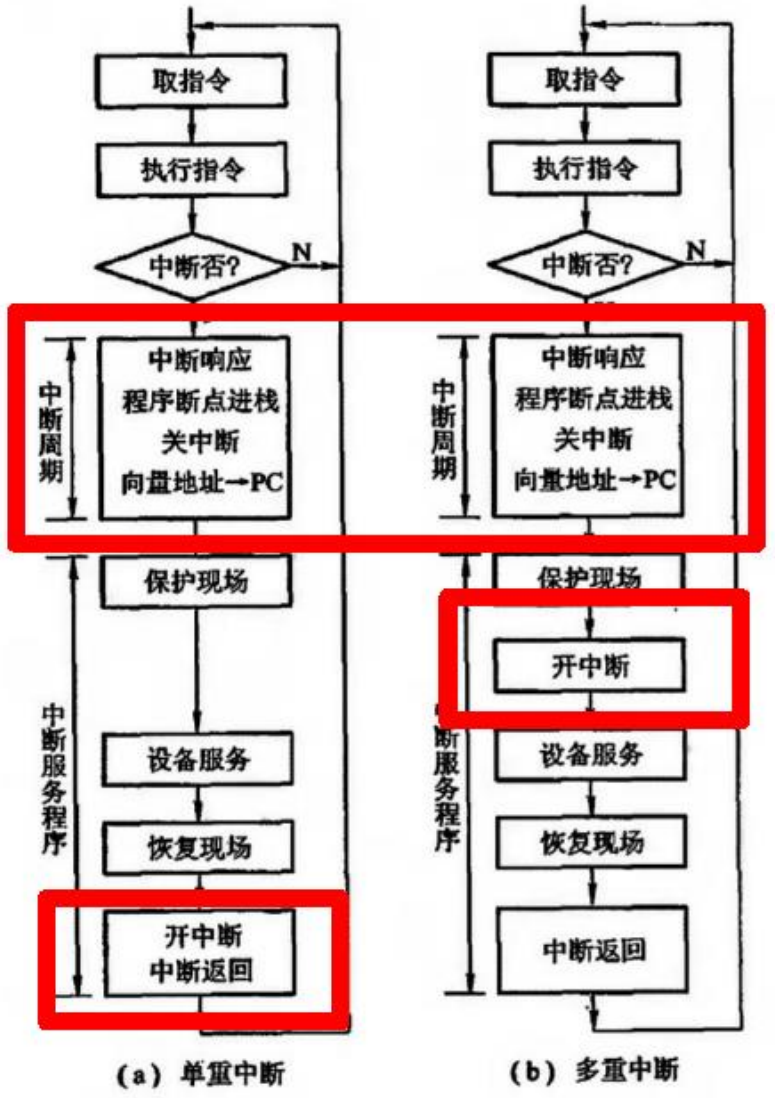
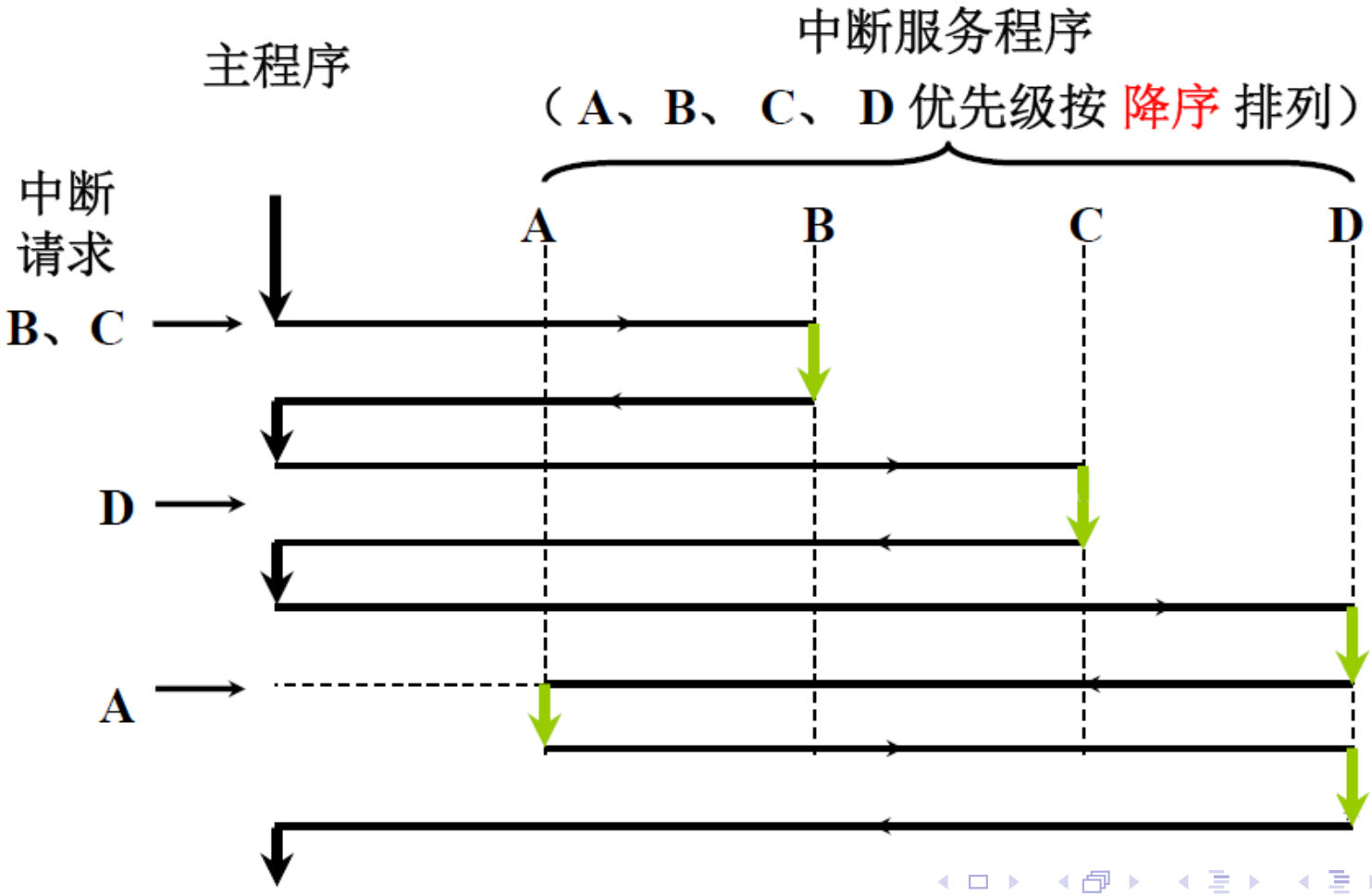


图 5.43 单重中断和多重中断服务程序流程

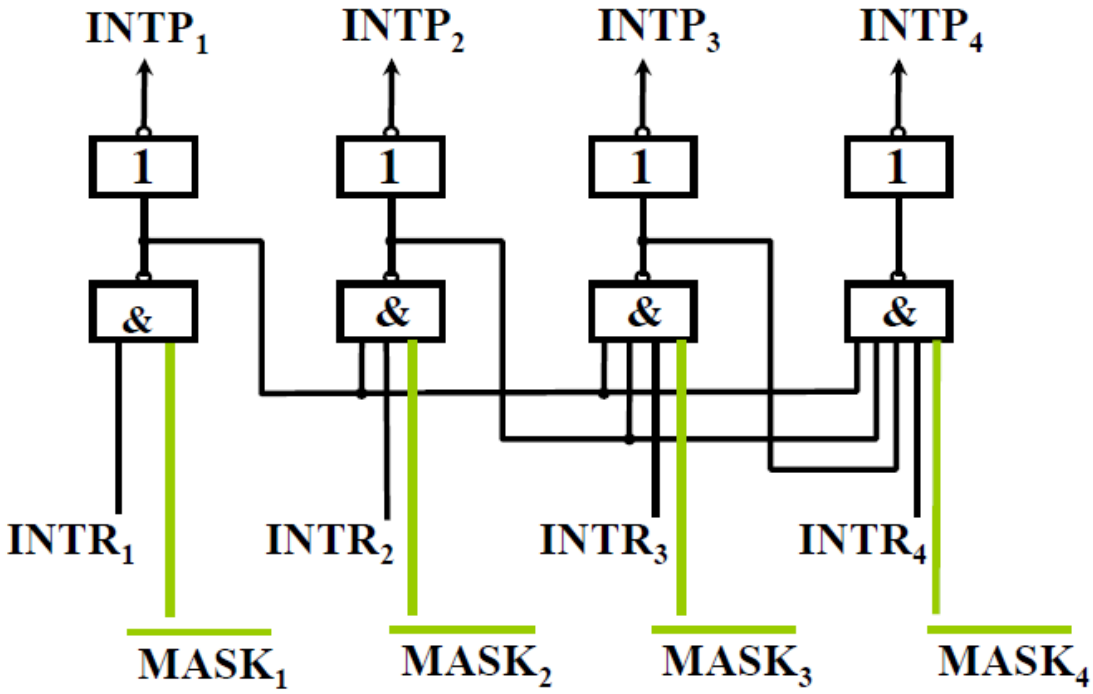
6. 处理中断的过程中，又出现新的中断怎么办?

- 实现多重中断的条件
  - 1. 提前设置开中断
  - 2. 优先级别高的中断源有权中断优先级别低的中断源



6. 处理中断的过程中，又出现新的中断怎么办?

- 屏蔽技术
  - 1. 屏蔽触发器
    - 实现动态优先级的设定



MASK = 0（未被屏蔽）　　$MASK_i = 1$（被屏蔽）

INTR 被置“1”　　$INTP_i = 0$（不能被排队选中）

6. 处理中断的过程中，又出现新的中断怎么办?

- 屏蔽技术
  - 2. 屏蔽字

16个中断源 1，2，3 … 16 按 降序 排列，每个对应16位屏蔽字

在ISR中设置屏蔽字，屏蔽对应中断源

优先级	屏 蔽 字
1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2	0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
3	0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
4	0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
5	0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
6	0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
⋮	⋮
15	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
16	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

6. 处理中断的过程中，又出现新的中断怎么办?

- 屏蔽技术
  - 3. 新屏蔽字的设置-ISR

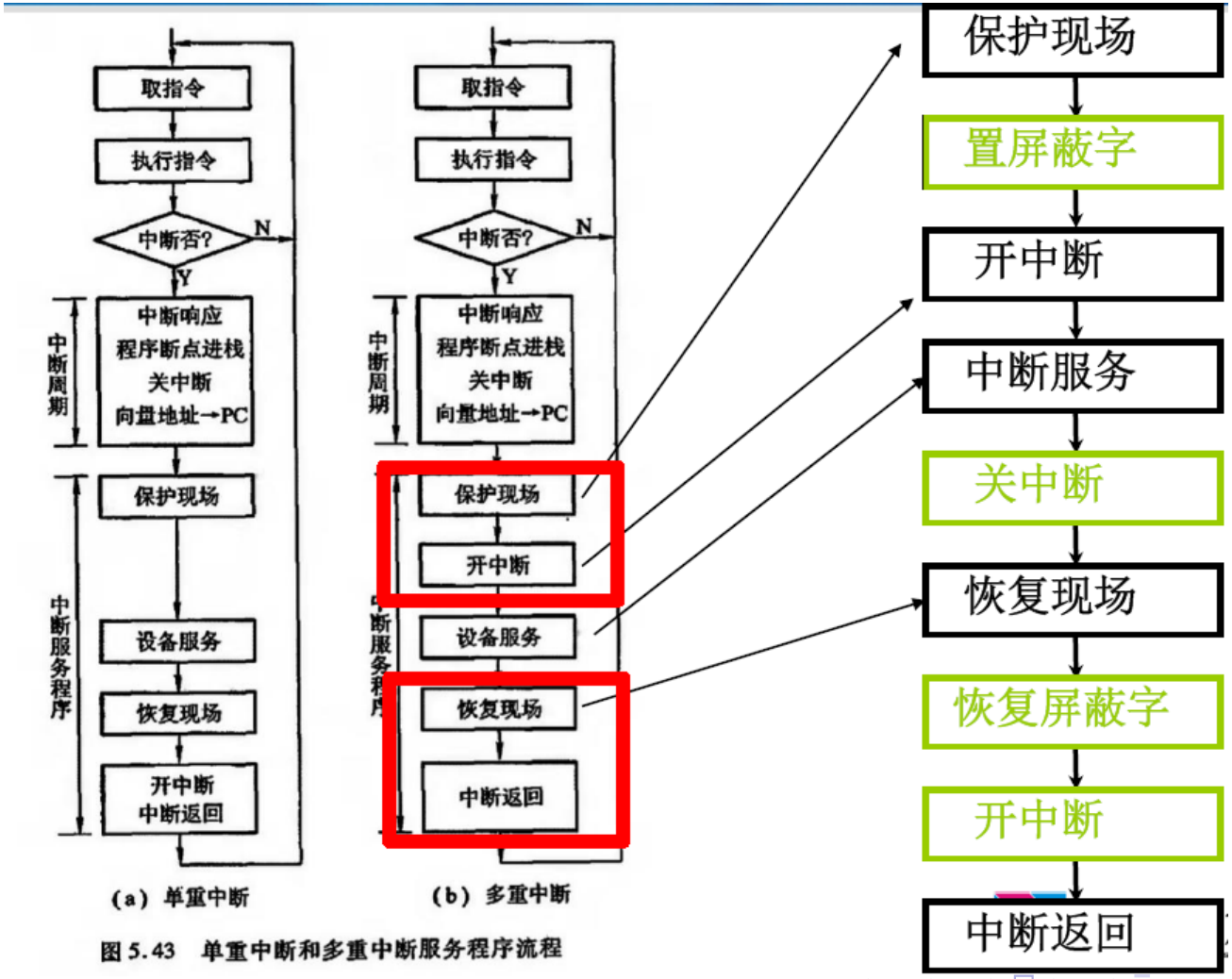


图 5.43 单重中断和多重中断服务程序流程

6. 处理中断的过程中，又出现新的中断怎么办?

- 屏蔽技术
  - 4. 屏蔽技术可以改变处理优先等级

响应优先级　　不可改变　为何不改变响应优先级?

处理优先级　　可改变（通过重新设置屏蔽字）

中断源	原屏蔽字	新屏蔽字
A	1 1 1 1	1 1 1 1
B	0 1 1 1	0 1 0 0
C	0 0 1 1	0 1 1 0
D	0 0 0 1	0 1 1 1

响应优先级 A→B→C→D 降序排列

处理优先级 A→D→C→B 降序排列

中断系统已解决的问题

- 1. 各中断源如何向CPU提出请求?
- 2. 各中断源同时提出请求怎么办?
- 3. CPU什么条件、什么时间、以什么方式响应中断?
- 4. 如何保护现场？如何恢复现场？如何返回?
- 5. 如何寻找入口地址?
- 6. 处理中断的过程中，又出现新的中断怎么办?

MIPS中的异常

0: Interrupt，中断；
1: TLB Modified，试图修改TLB中映射为只读的内存地址；
2: TLB Miss Load，试图读取一个没有在TLB中映射到物理地址的虚拟地址；
3: TLB Miss Store，试图向一个没有在TLB中映射到物理地址的虚拟地址存入数据；
4: Address Error Load，试图从一个非对齐的地址读取信息；
5: Address Error Store，试图向一个非对齐的地址写入信息；
6: Instruction Bus Error，一般是指令Cache出错；
7: Data Bus Error，一般是数据Cache出错；
8: Syscall，由syscall指令产生。操作系统下，通用的由用户态进入内核态的方法。
9: Break Point，由break指令产生。最常见的bp指令，是由编译器产生的，在除法运算时插入一个break point指令，以达到在除0时抛出错误信息的目的。因此，如果在定位问题时发现了一个Break Point异常，且它的异常分代码为07，应当考虑是出现了除0的情形；
10: RI，保留指令。在CPU执行到一条没有定义的指令时，进入此异常；
11: Co-processor Unavailable，协处理器不可用。这个异常是由于试图对不存在的协处理器进行操作引起的。特别的，在没有浮点协处理器的处理器上执行这条命令，会导致这个异常。随之，操作系统会调用模拟浮点的lib库，来实现软件的浮点运算；
12: **Overflow**，算术溢出。只有带符号的运算会引起这个异常；
13: Trap，这个异常来源于trap指令。和syscall指令类似地，trap指令也会引起一个异常，但trap指令可以附带一些条件，这样可以用于调试程序用。
14: VCEI，指令高速缓存中的虚地址一致性错误。
15: Float Point Exception，浮点异常；
16: Co-processor 2 Exception，协处理器2的异常；
17~22，留作扩展；
23: Watch，内存断点异常。当设定了WatchLo/WatchHi两个寄存器时起作用。当load/store的虚拟地址和WatchLo/WatchHi中匹配时，会引发这样一个异常
24~30，留作扩展；

MIPS中的异常分类

- 外部事件
  - IO中断或读总线出错等 0,6,7
- 操作系统调用
  - 缺页或越界等 8,9,13
- 算术溢出 12
- 非法指令 10
- 其它硬件错误
  - 存储等、协处理器、浮点

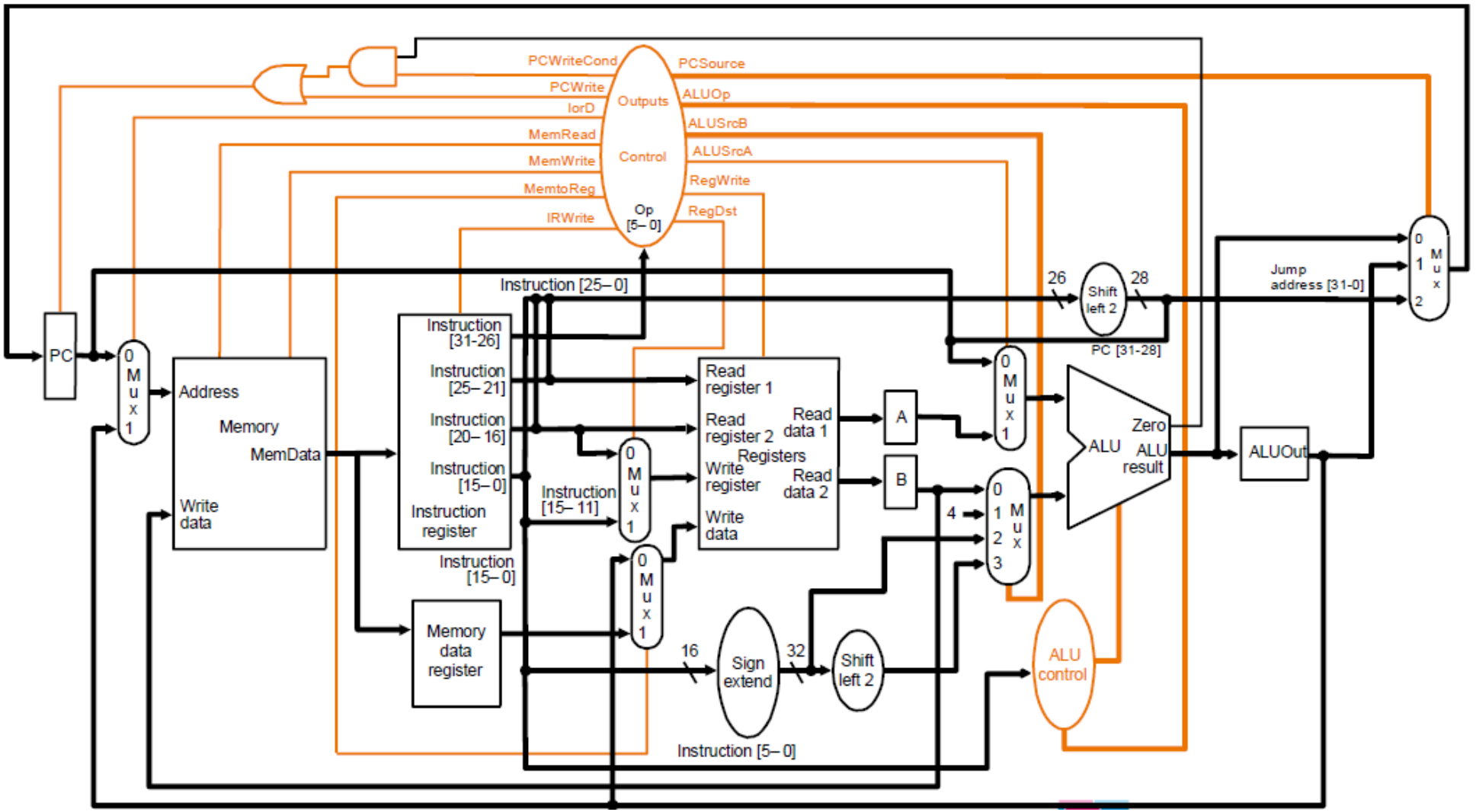
Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

MIPS的异常处理

- 讨论两种异常作为示例：非法指令和算术溢出
- 异常处理的主要工作
  - ✓断点保存：将异常执行指令的地址保存在EPC寄存器中
  - ✓异常识别：根据状态寄存器cause中的异常原因分别处理异常
    - 非法指令：转到特定服务程序
    - 溢出：对溢出进行响应
  - ✓跳转异常服务程序（PC）：停止程序的执行并报告错误等
    - 未定义指令异常处理在$8000\ 0000_{hex}$
    - 算数溢出异常处理$8000\ 0180_{hex}$
- 服务程序的入口

Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

数据通路

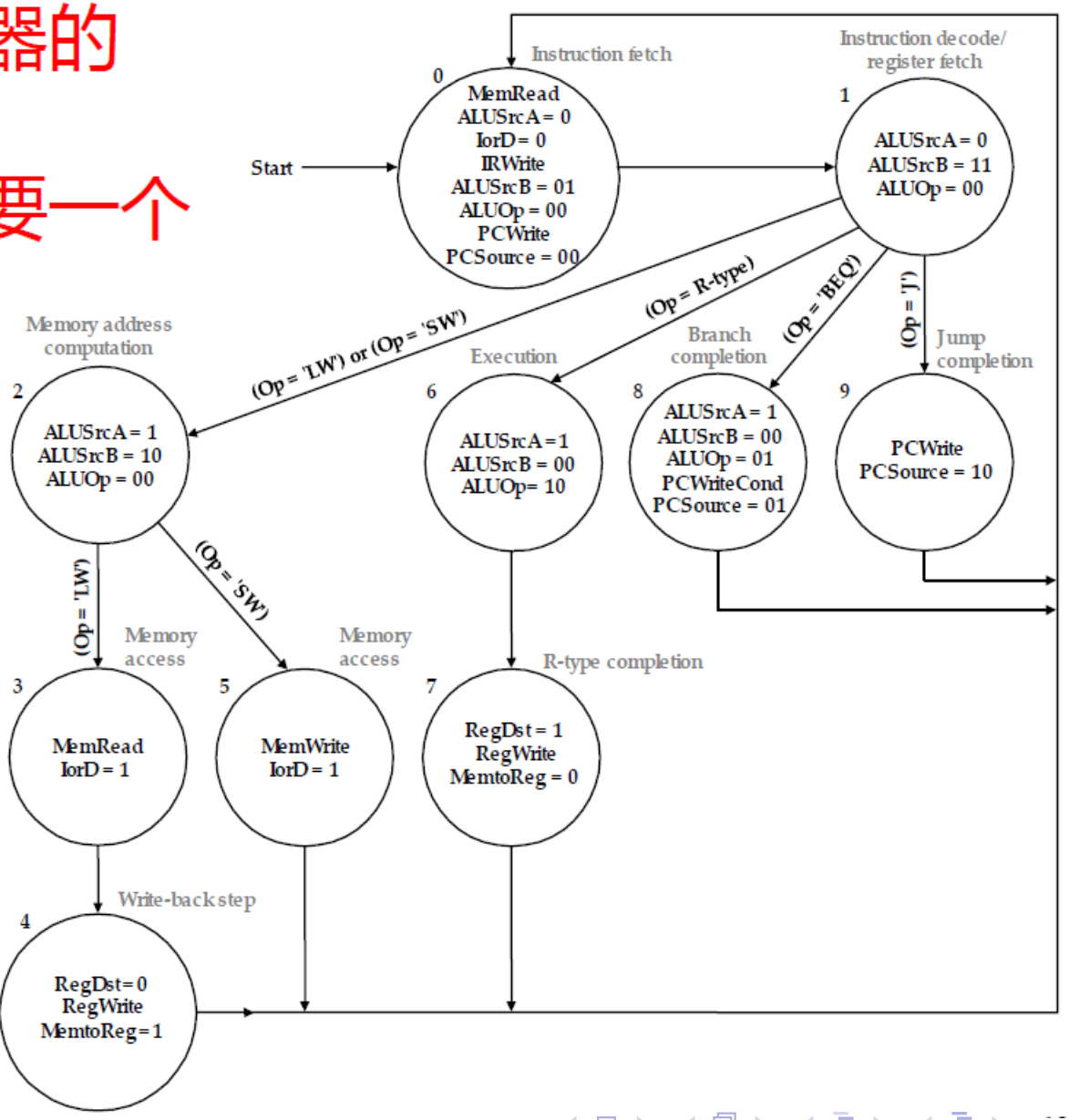

PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op [5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0 M u x 1
Address
Memory
MemData
Write data
Instruction [31-26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–0]
Instruction register
Instruction [15–0]
Instruction [15–11]
Memory data register
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
A
B
4
16
Sign extend
32
Shift left 2
ALU
Zero
ALU result
ALUOut
ALU control
Instruction [5–0]
26
28
Shift left 2
PC [31-28]
Jump address [31-0]
0 1 2 3 M u x

多周期实现

Step	R-Type	lw/sw	beq/bne	j
IF	IR = Mem[PC] PC = PC + 4			
ID	A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALUOut = PC + (SE(IR[15-0]) << 2)			
EX	ALUOut = A op B	ALUOut = A +SE(IR[15-0])	If (A==B) then PC = ALUOut	PC = PC[31-28] || (IR[25-0]<<2
MEM	Reg[IR[15-11]] = ALUOut	MDR=Mem[ALUOut] Mem[ALUOut] = B		
WB		Reg[IR[20-16]] = MDR		

多周期FSM

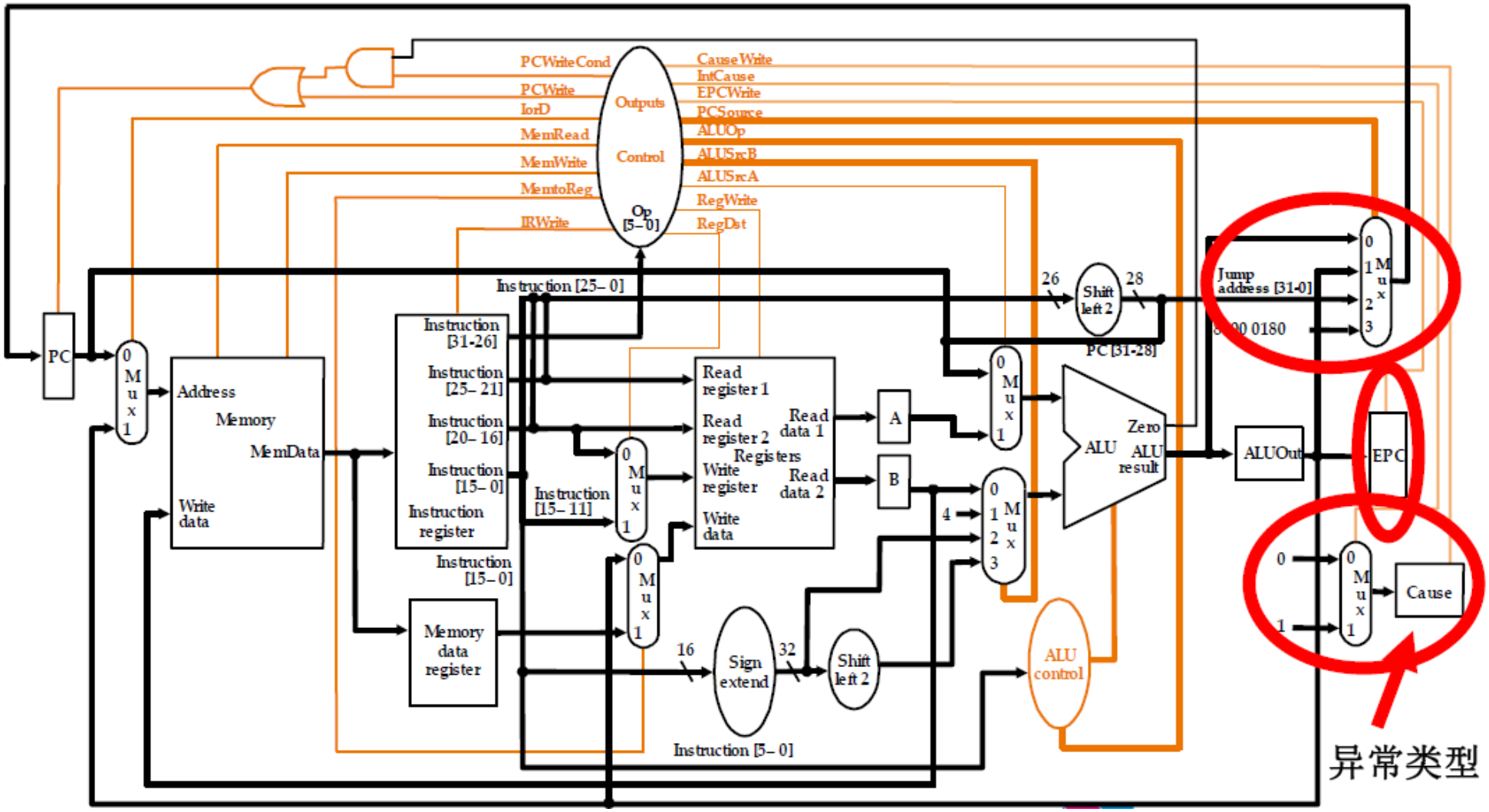
多周期控制器的MooreFSM，每个状态需要一个时钟周期。



多周期支持异常

- 完成功能（非法指令、算术溢出）
  - ✓PC赋值(根据异常原因跳转到异常处理程序)
  - ✓出错PC保存（PC-4保存到寄存器）
  - ✓出错原因保存（异常原因保存到寄存器）
- 数据通路
  - ✓PC、EPC、Cause
- 控制信号
  - ✓PC写（多路选择器）、EPC计算写入、Cause写入
- RTL代码
  - ✓非法指令实现、算术溢出实现
- 状态机
  - ✓加入几个状态，哪里加入?

多周期模式的异常处理


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
CauseWrite
IntCause
EPCWrite
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Instruction [25– 0]
26
Shift
left 2
28
PC [31-28]
Jump
address [31-0]
0 0180
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[31-26]
Instruction
[25– 21]
Instruction
[20– 16]
Instruction
[15– 0]
Instruction
register
Instruction
[15– 11]
Instruction
[15– 0]
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
Zero
ALU
ALU
result
ALUOut
EPC
Memory
data
register
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5– 0]
Cause
异常类型

异常

Step	R-Type	lw/sw	beq/bne	j	Exception
IF	IR = Mem[PC] PC = PC + 4				
ID	A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALUOut = PC + (SE(IR[15-0]) << 2)				
EX	ALUOut = A op B	ALUOut = A +SE(IR[15-0])	If (A==B) then PC = ALUOut	PC = PC[31-28] || (IR[25-0]<<2	
MEM	Reg[IR[15-11]] = ALUOut	MDR=Mem[ALUOut] Mem[ALUOut] = B			
WB		Reg[IR[20-16]] = MDR			

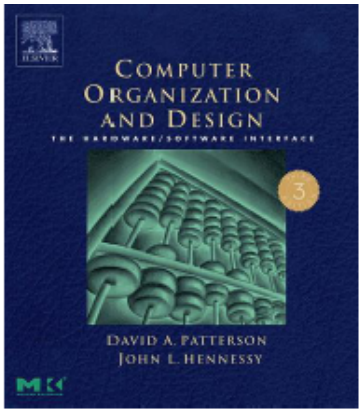
异常指令

PC=0X80000000
EPC=PC-4
CAUSE=0

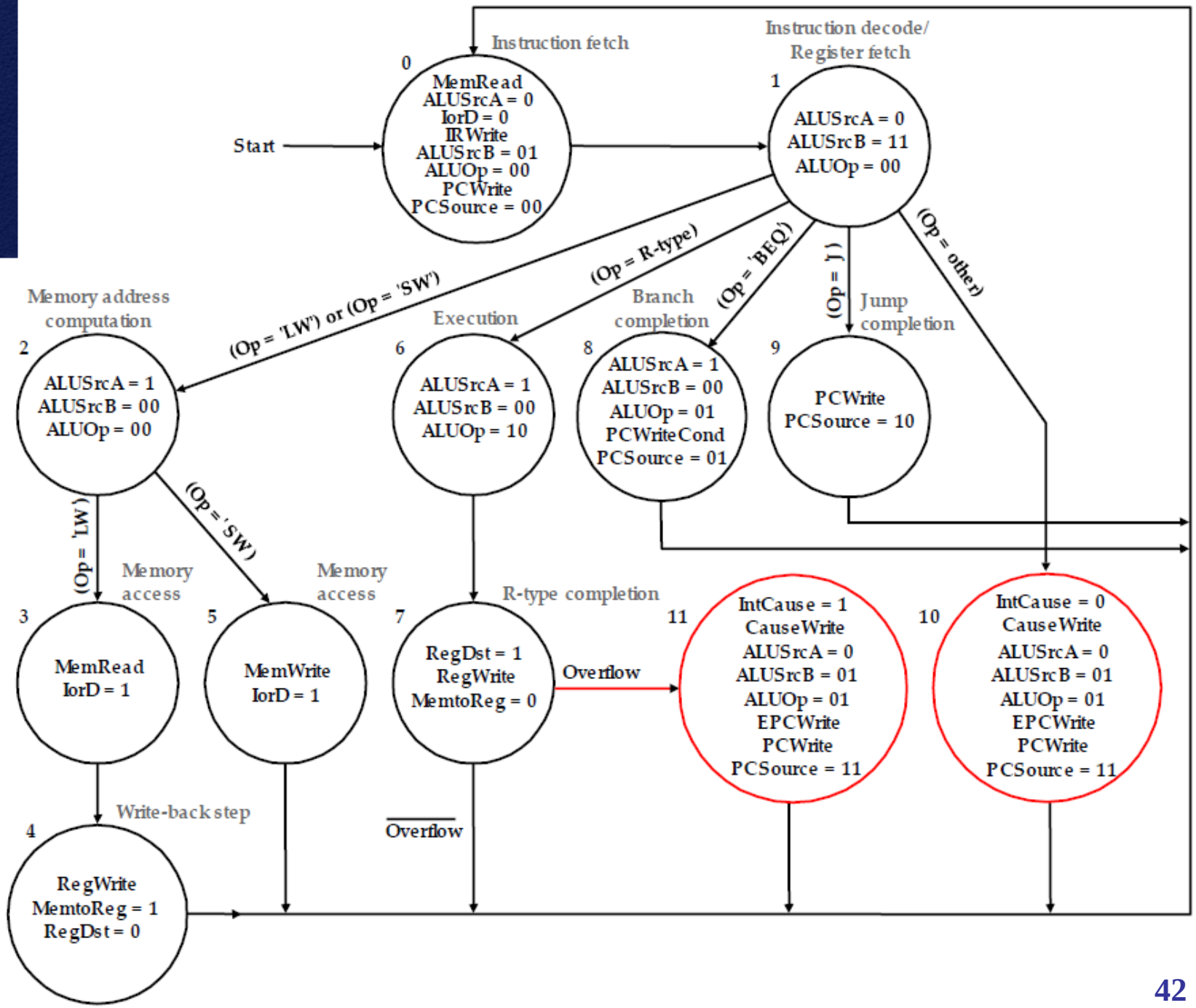
算术溢出
PC=0X80000180
EPC=PC-4
CAUSE=1

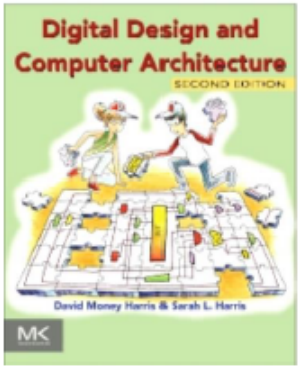
两种异常的处理

问：算术溢出应该在哪个周期？

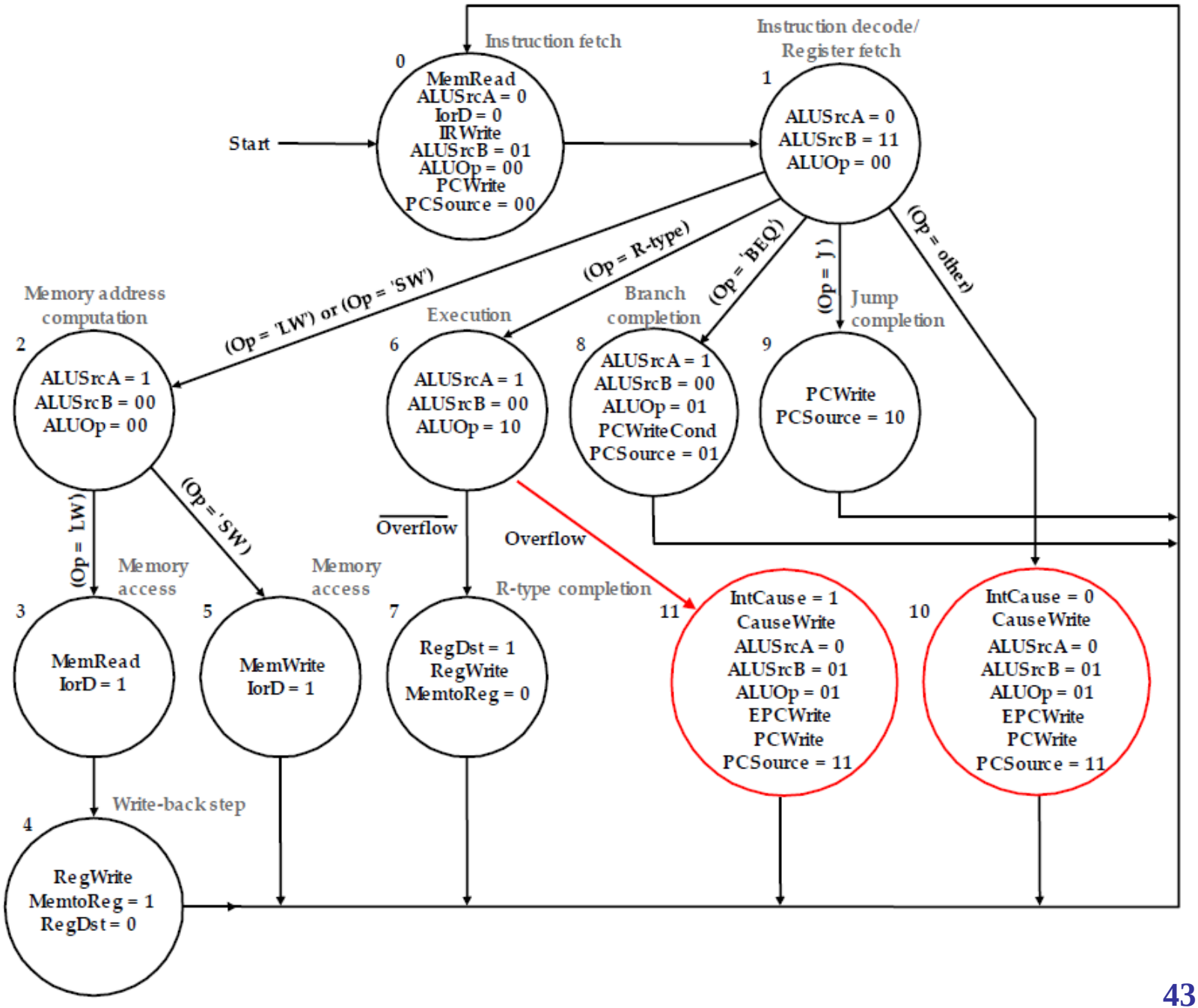

COMPUTER
ORGANIZATION
AND DESIGN
THE HARDWARE/SOFTWARE INTERFACE
DAVID A. PATTERSON
JOHN L. HENNESSY

异常处理控制

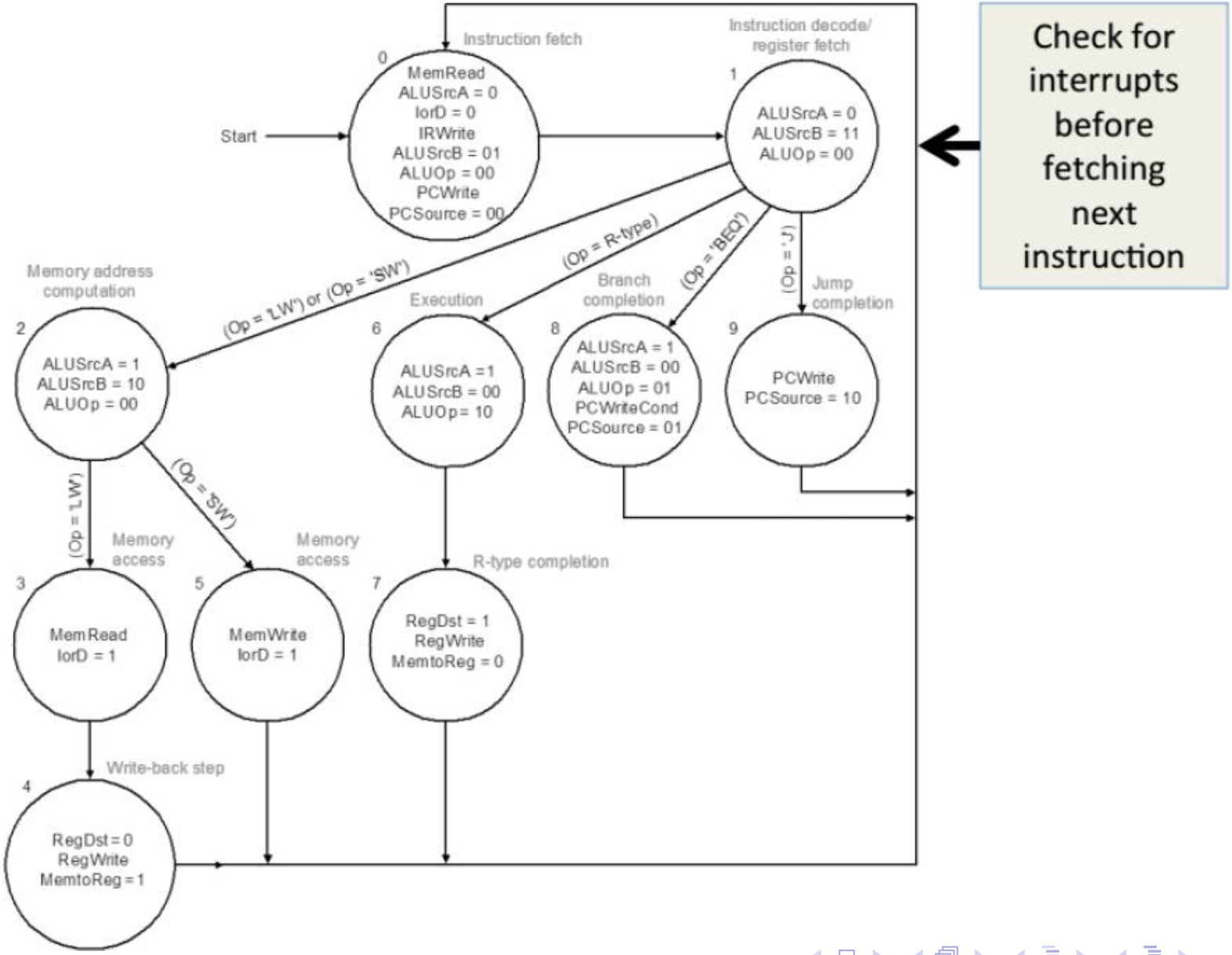

Instruction fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
Instruction decode/
Register fetch
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
(Op = other)
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
3
MemRead
IorD = 1
Memory
access
5
MemWrite
IorD = 1
R-type completion
7
RegDst = 1
RegWrite
MemtoReg = 0
Overflow
11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
Write-back step
4
RegWrite
MemtoReg = 1
RegDst = 0
Overflow


Digital Design and
Computer Architecture
SECOND EDITION
David Money Harris & Sarah L. Harris
MK

异常处理控制

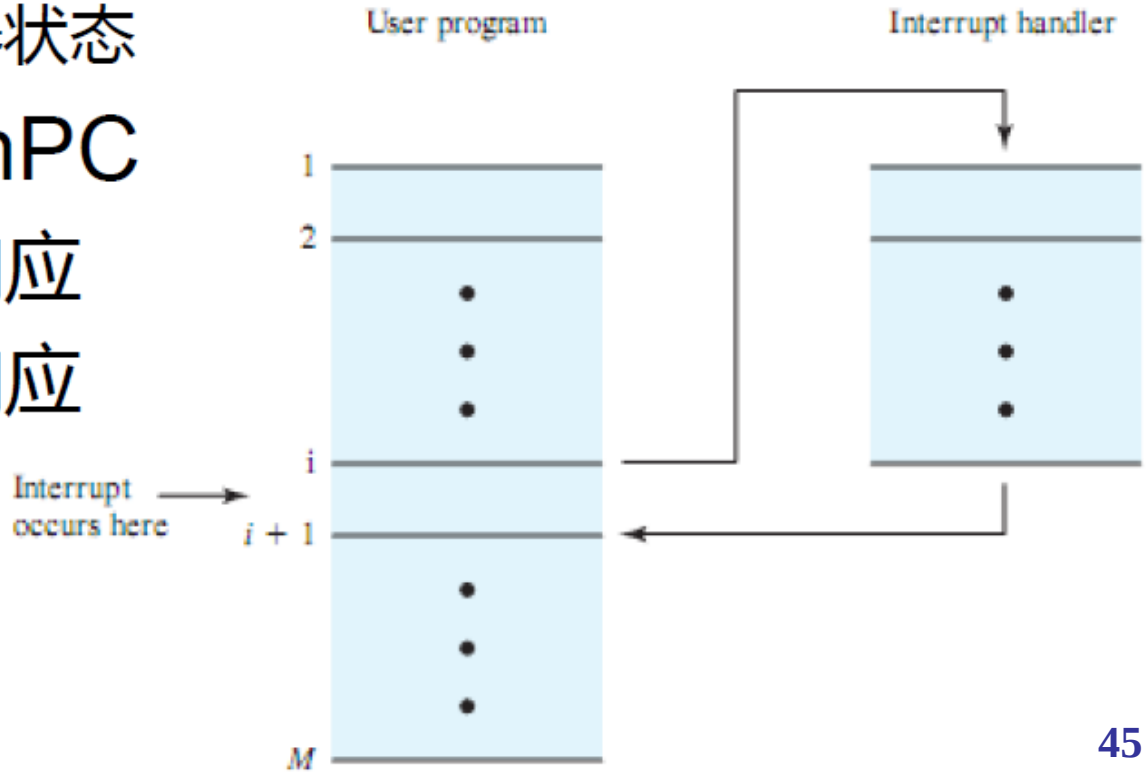

Start
Instruction fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Instruction decode/
Register fetch
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
(Op = other)
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Overflow
Overflow
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
Write-back step
4
RegWrite
MemtoReg = 1
RegDst = 0

多周期的其它中断控制

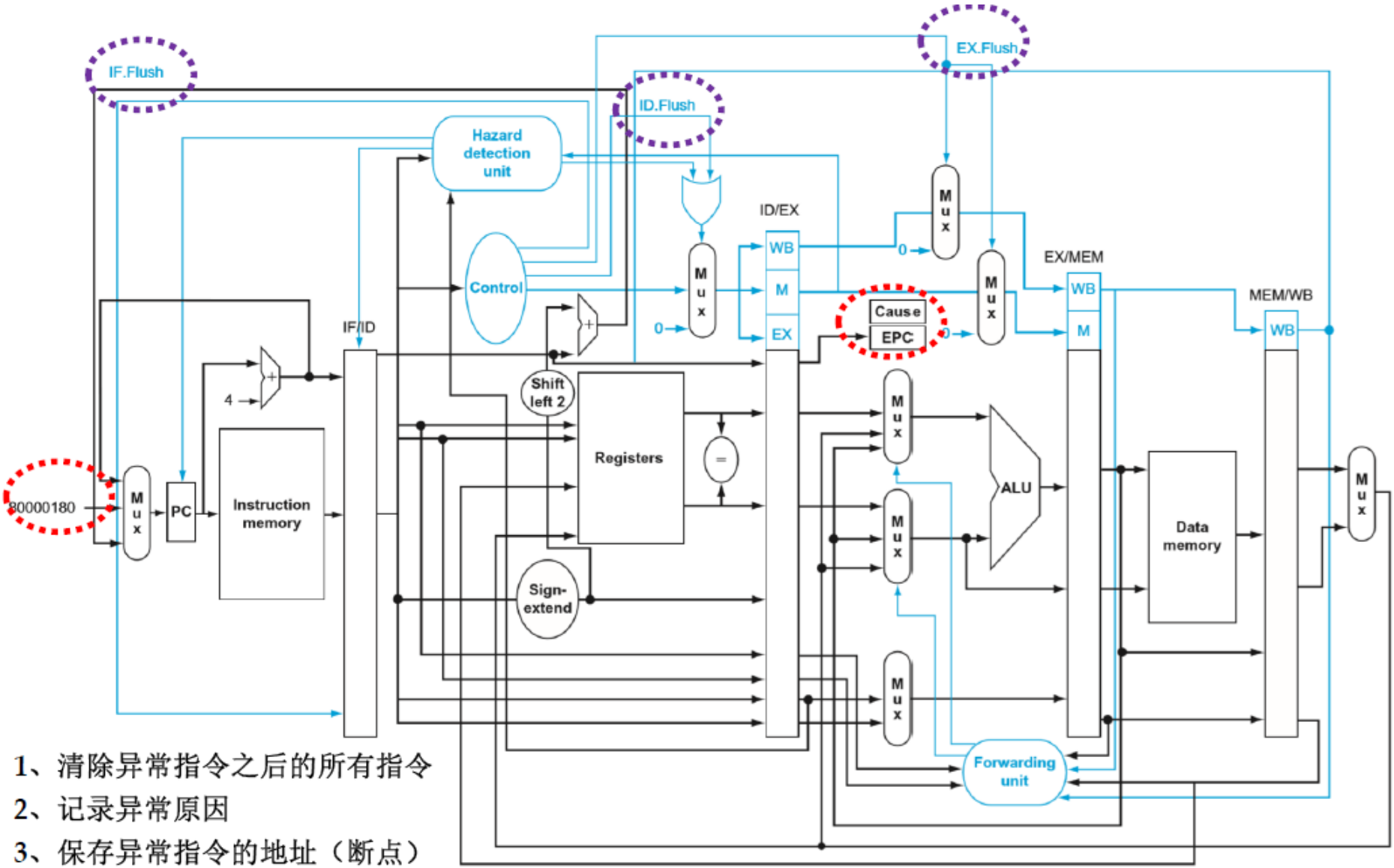

Instruction fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
Instruction decode/
register fetch
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
3
MemRead
IorD = 1
Memory
access
5
MemWrite
IorD = 1
R-type completion
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst = 0
RegWrite
MemtoReg = 1
Check for interrupts before fetching next instruction

中断和异常操作语义特点（非流水线）

- 顺序语义
  - ✓之前的指令都已执行完成
    - 已经提交其状态
  - ✓之后的指令还没有启动
    - 没有改变任何机器状态
- 断点精确：= PC/nPC
  - ✓外部中断：异步响应
  - ✓指令异常：同步响应
- 现场简明
  - ✓Cause
  - ✓EPC



例子：在MIPS流水线中实现算术溢出异常



1、清除异常指令之后的所有指令

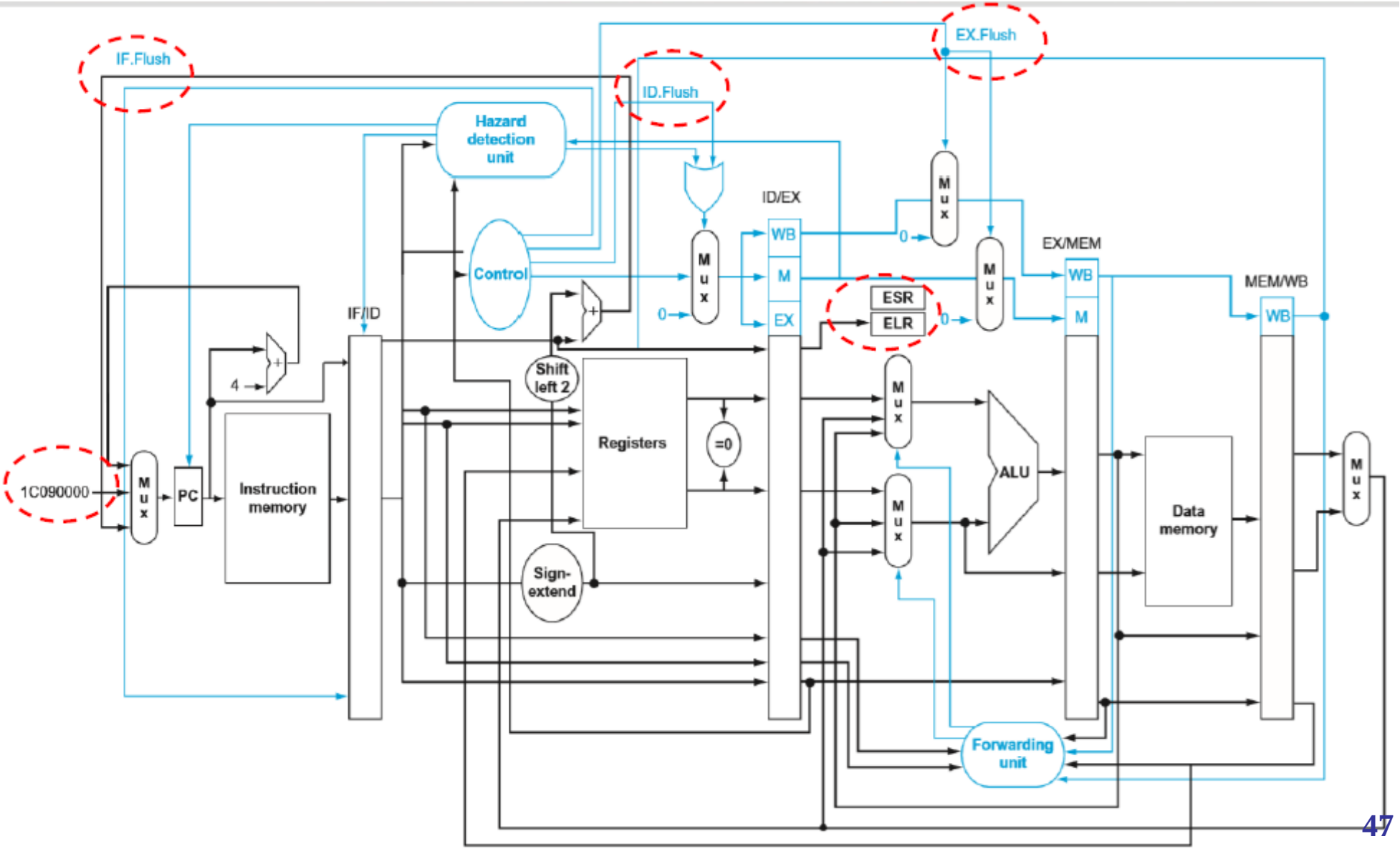
2、记录异常原因

3、保存异常指令的地址（断点）

4、转到服务程序运行

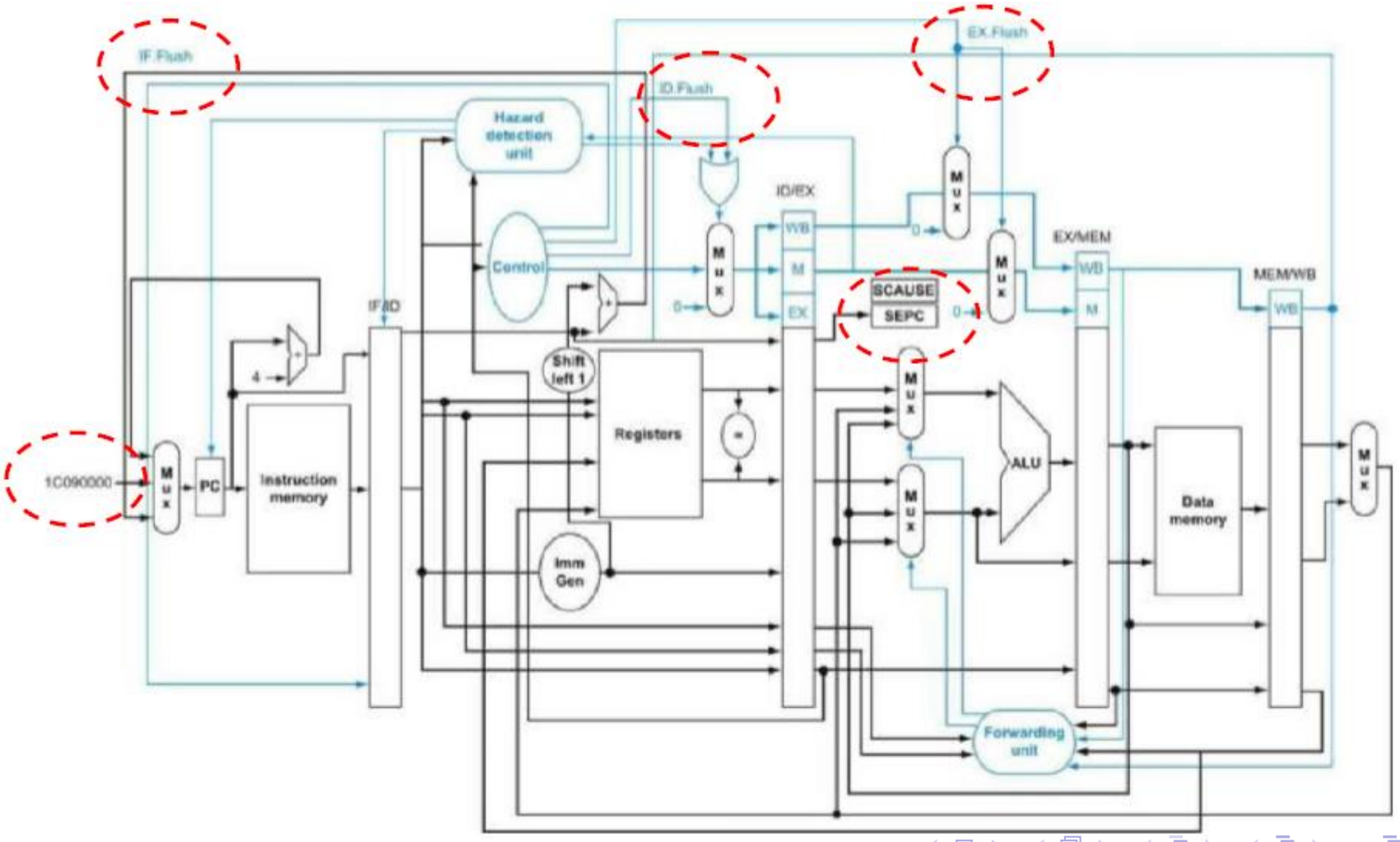
在ARM中实现

Exception type	Exception vector address to be added to a Vector Table Base Register
Unknown Reason	$00\ 0000_{two}$
Floating-point arithmetic exception	$10\ 1100_{two}$
System Error (hardware malfunction)	$10\ 1111_{two}$



在RISC-V中实现

Exception type	Exception vector address to be added to a Vector Table Base Register
Undefined instruction	$00\ 0100\ 0000_{two}$
System Error (hardware malfunction)	$01\ 1000\ 0000_{two}$



RISC-V异常的种类

- External events
  - ✓ 中断，读总线错
- Memory translation exceptions
  - ✓ 缺页，越界
- Other unusual program conditions for the kernel to fix
  - ✓ 须内核处理的不常见程序状态
- Program or hardware-detected errors
  - ✓ 非法指令、溢出、对齐等
- Data integrity problems（Checksum etc.）
  - ✓ 校验错
- System Calls and traps

Type of event	From where?	RISC-V terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Either

RISC-V处理异常的方法

- 保存导致异常（或被中断）的指令的PC值
  - ✓ RISC-V使用SEPC（Supervisor Exception Program Counter，管理员异常程序计数器）
- 保存问题的表征
  - ✓ RISC-V使用SCAUSE（Supervisor Exception Cause Register，管理员异常原因寄存器）
  - ✓ 64位，但多数位没有用到：异常编码字段：2为未定义的操作码，12为硬件故障，…
- 跳转到处理程序
  - ✓ 假定程序位于0000 0000 1C09 0000hex

Type of event	From where?	RISC-V terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Either

RISC-V异常/中断的5种属性

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

RISC-V处理异常的方法

- 向量化中断
  - ✓ 处理程序的地址由中断原因决定
- 向量表基地址寄存器加上异常向量地址：
  - ✓ 未定义的操作码： 00 0100 0000two
  - ✓ 硬件故障： 01 1000 0000two

Exception type	Exception vector address to be added to a Vector Table Base Register
Undefined instruction	$00\ 0100\ 0000_{two}$
System Error (hardware malfunction)	$01\ 1000\ 0000_{two}$

- 处理程序的功能为
  - ✓ 处理中断，
  - ✓ 或是跳转到实际的处理程序

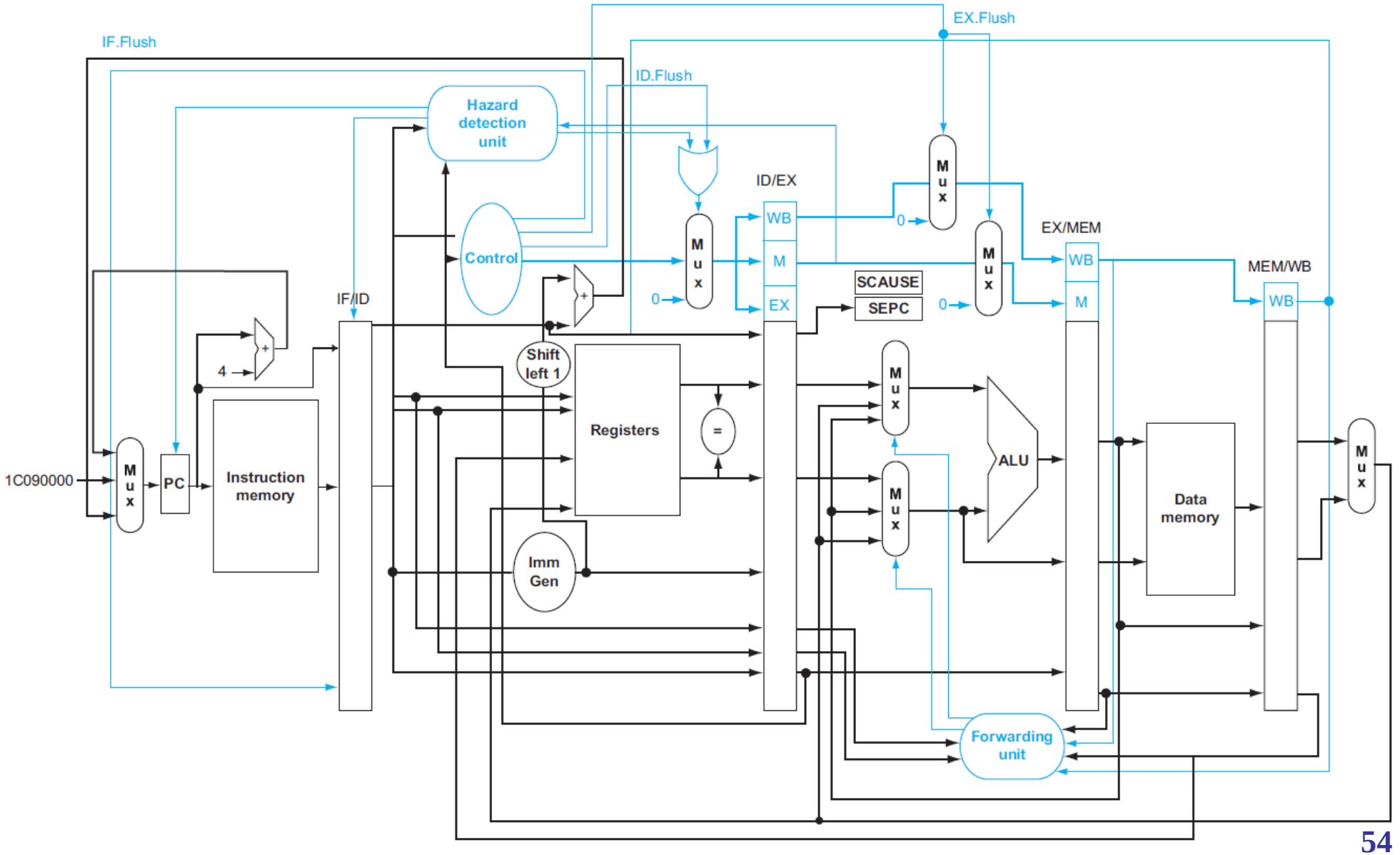
RISC-V流水线中的异常

- 另一种形式的控制冒险
- 考虑add在EX级发生的故障

```
add x1, x2, x1
```

  - ✓防止x1被错误赋值
  - ✓完成先前的指令
  - ✓清除add及之后的指令
  - ✓设置SEPC和SCAUSE寄存器的值
  - ✓把控制转移到处理程序
- 与误预测的分支类似
  - ✓用到的硬件多数相同

支持异常处理的RISC-V流水线



例子

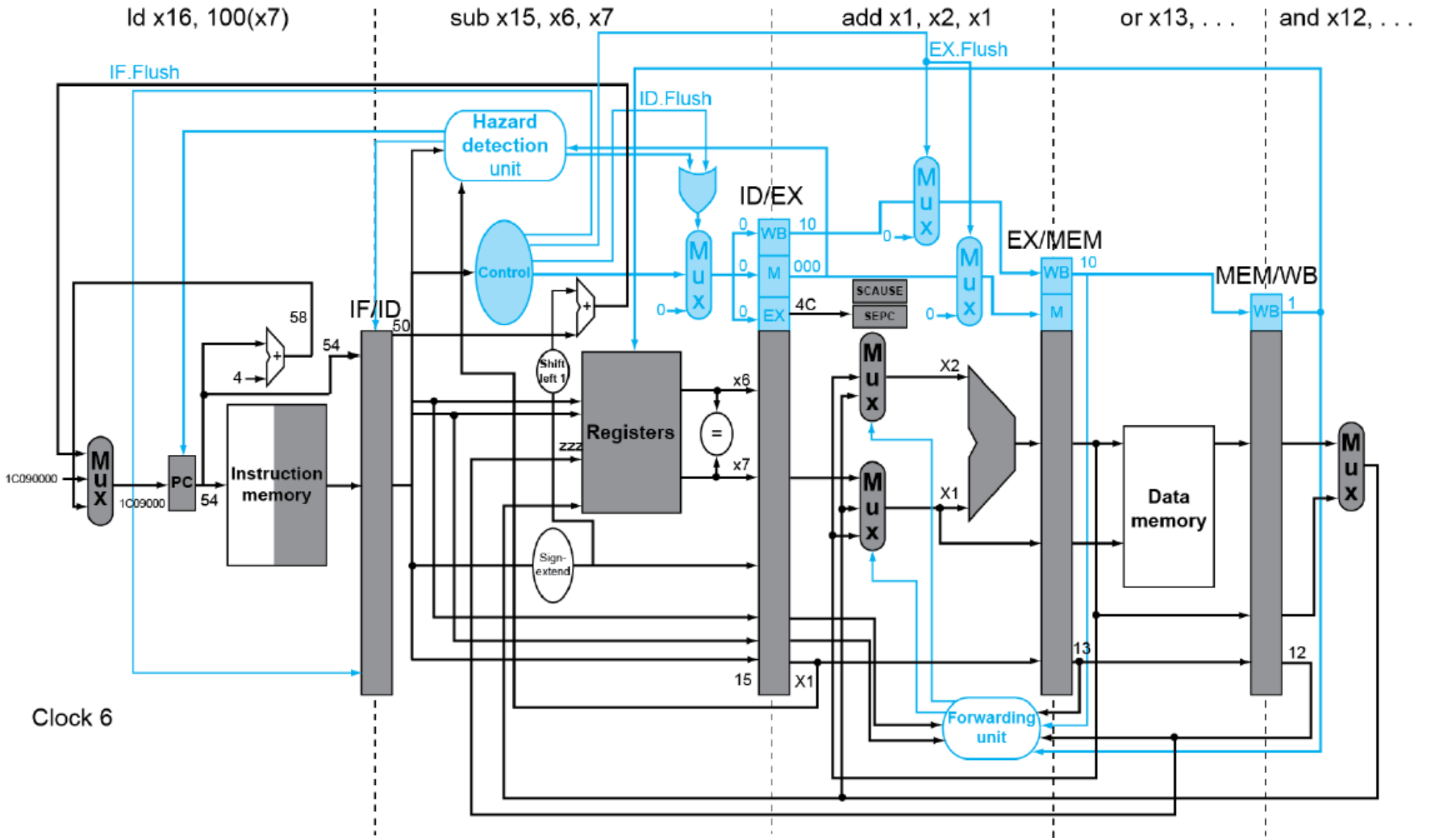
- Exception on add in

```
40    sub   x11, x2, x4
44    and   x12, x2, x5
48    orr   x13, x2, x6
4c    add   x1,  x2, x1
50    sub   x15, x6, x7
54    ld    x16, 100(x7)
…
```

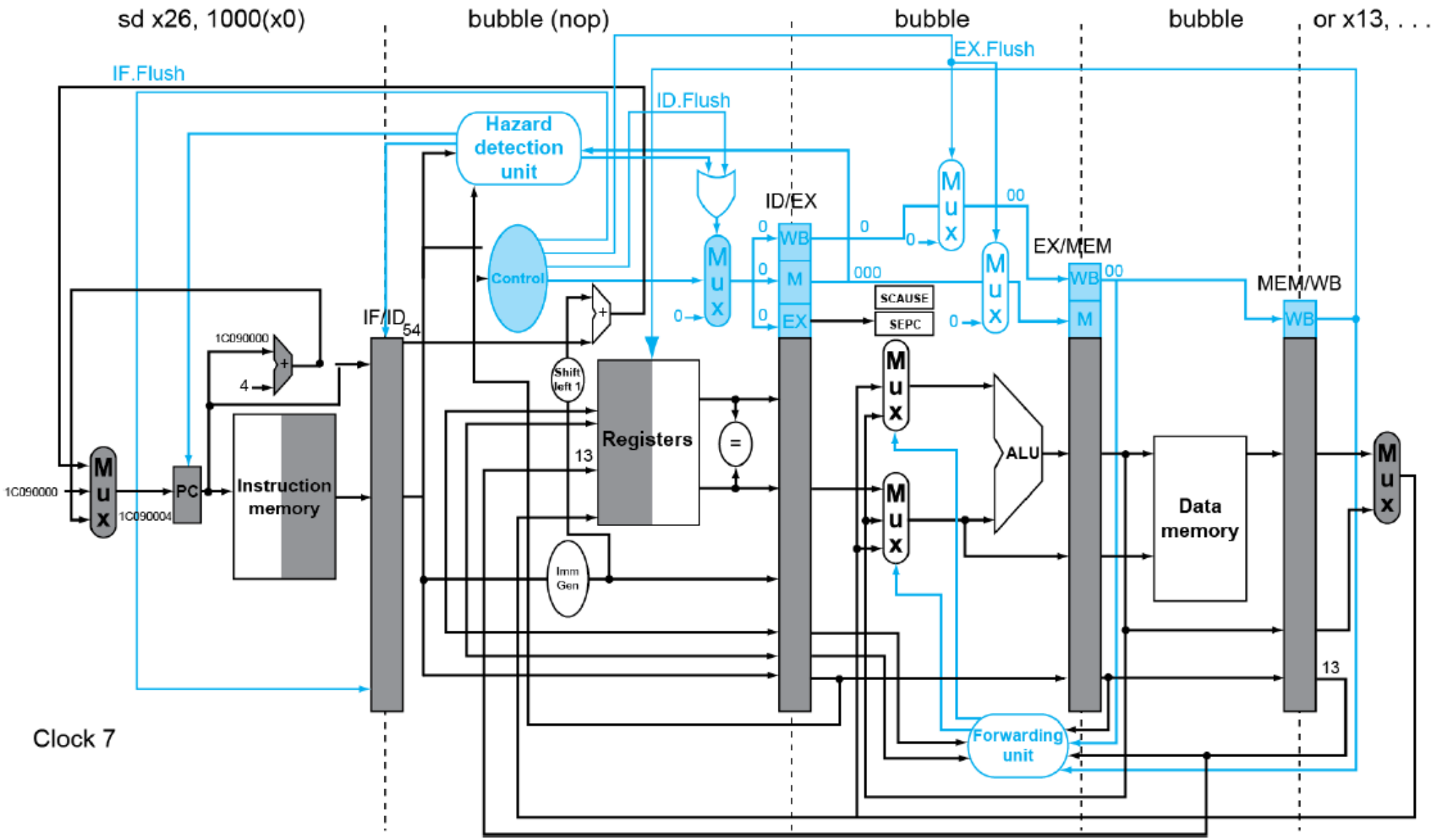
- Handler

```
1c090000  sd   x26, 1000(x10)
1c090004   sd   x27, 1008(x10)
…
```

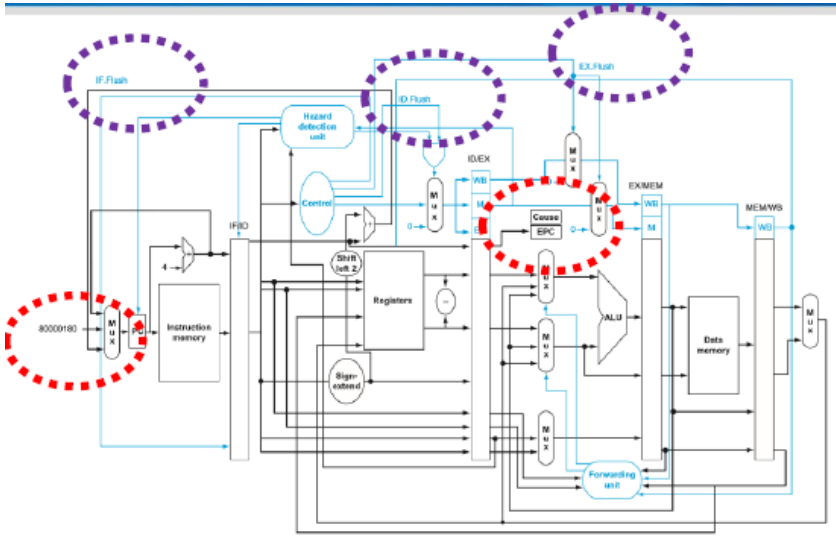
例子

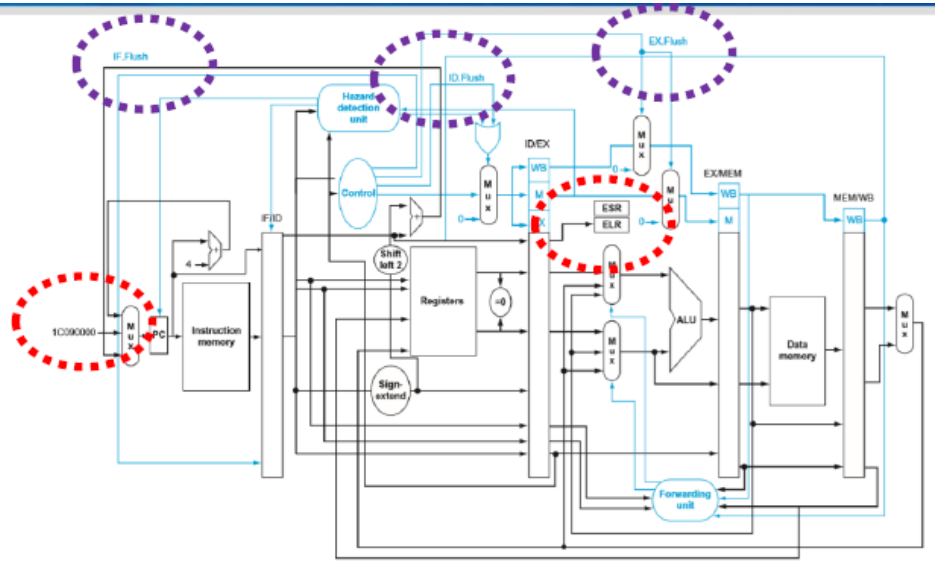

ld x16, 100(x7)
sub x15, x6, x7
add x1, x2, x1
or x13, . . .
and x12, . . .
IF.Flush
EX.Flush
ID.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
000
10
1
4C
SCAUSE
SEPC
Shift left 1
Registers
Sign-extend
Instruction memory
Data memory
Forwarding unit
PC
1C090000
1C09000
zzz
x6
x7
X1
X2
15
13
12
58
54
50
4
Clock 6

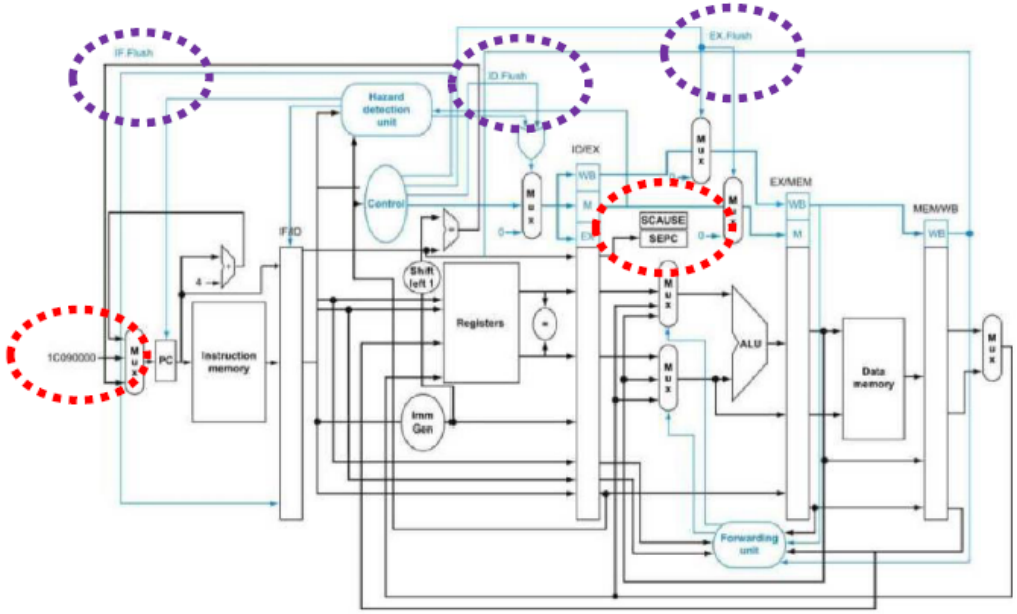
例子


sd x26, 1000(x0)
bubble (nop)
bubble
bubble
or x13, . . .
IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
SCAUSE
SEPC
IF/ID
1C090000
4
PC
1C090004
Instruction memory
Shift left 1
Registers
13
Imm Gen
=
ALU
Data memory
Forwarding unit
Mux
Clock 7

总结：MIPS-ARM-RISCV



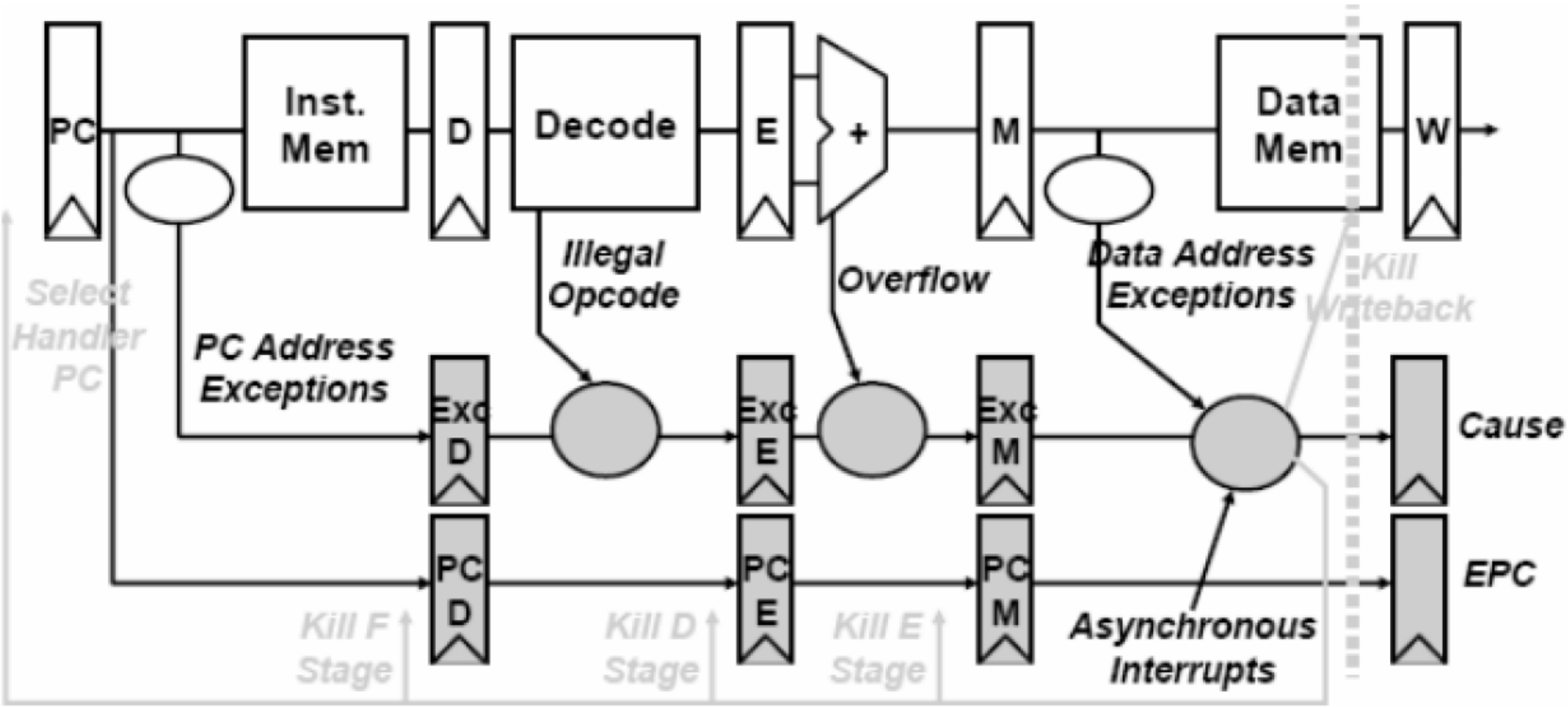




1、清除异常指令之后的所有指令
2、记录异常原因
3、保存异常指令的地址（断点）
4、转到服务程序运行

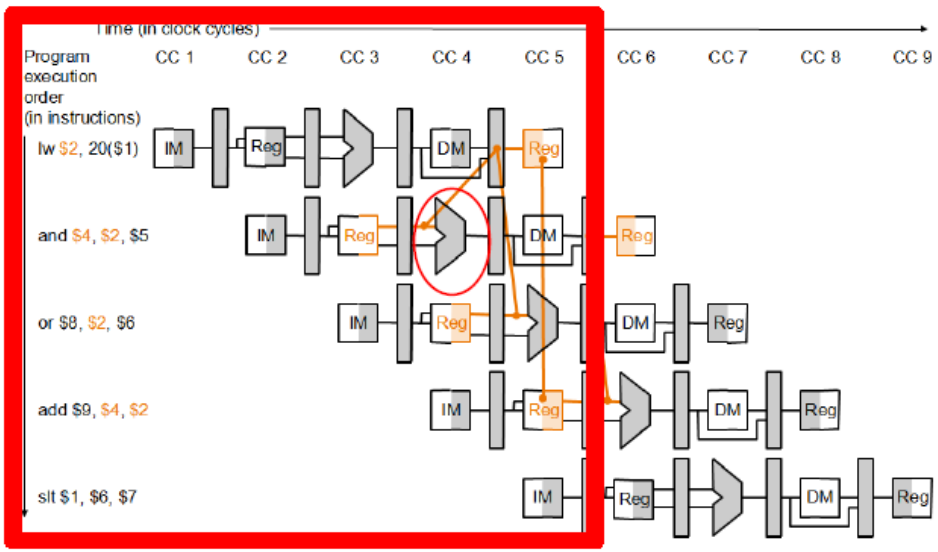
流水线中的异常与中断

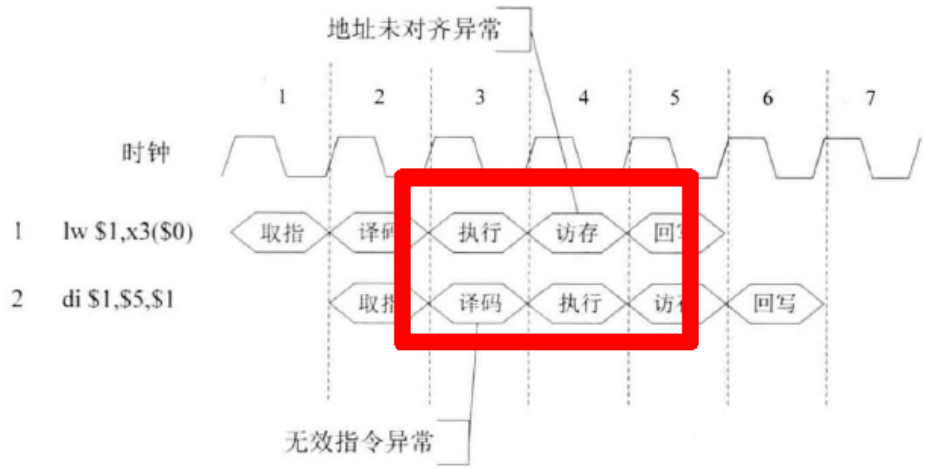
- 多个流水段，多条指令，多种异常并发
  - ✓异常：访存-地址错（缺页越界），译码-非法指令，ALU溢出
  - ✓中断



流水线异常处理的挑战

- “顺序执行”只是一种逻辑关系：断点和状态难以确定
  - ✓后续指令在产生异常指令完成之前改变了系统的部分状态
    - 如 ld 指令在MEM段产生异常，而其后的R-type指令设置了0标志
  - ✓异常发生顺序与指令执行顺序不一定相同
    - 如di指令在ID段CC3发生异常，但LW指令在MEM段CC4异常
  - ✓转移指令和分支预测给异常处理带来了麻烦

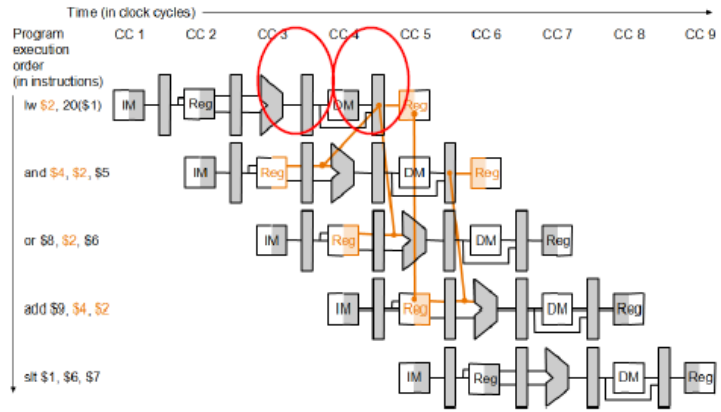




解决方案：非精确处理、精确处理

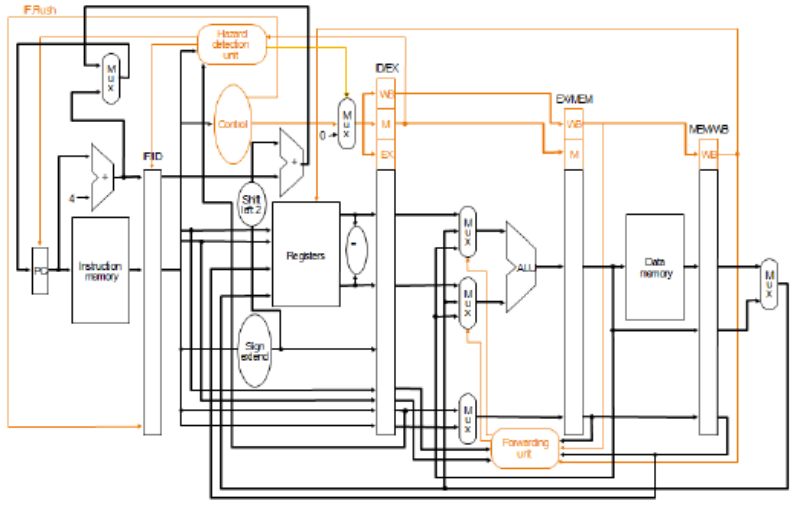
思路1：非精确异常

- 非精确方案1：允许已进入流水线中的指令执行完再转去执行中断处理
  - ✓断点：无论在第i条指令的哪一流水段上发生异常，都不再允许后继指令进入流水线，断点为最后进入流水线的那条指令的地址(非精确)
    - 非精确（可能不是“当前指令”）
    - 可变（不同段发生异常，EPC增量不同）
- 优点：硬件比较简单



思路1：非精确异常

- 非精确方案2：将异常指令的后续指令排空，COD
- 处理：将异常视为一种控制相关，工作如下：
  - ✓暂停指令流中导致异常的指令
  - ✓执行完异常指令之前的所有指令
  - ✓清除异常指令之后的所有指令
  - ✓记录异常原因
  - ✓保存断点
  - ✓转异常处理程序
- 不允许后续指令继续执行



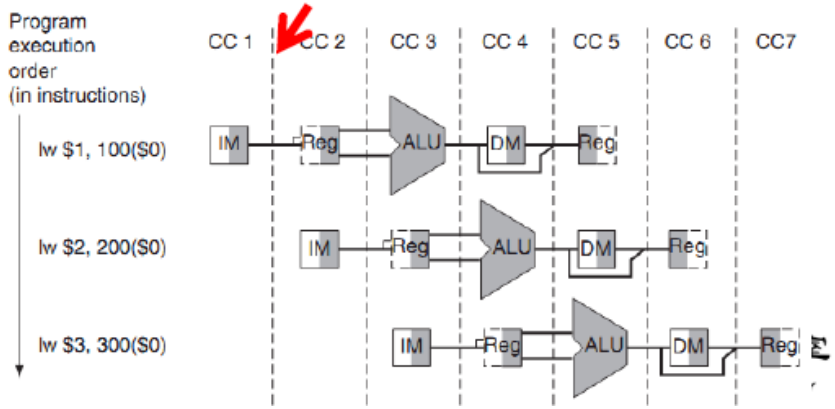
非精确异常存在的问题

◘缺点

- ✓ 异常响应时间较长
- ✓ 如果等进入流水线的指令执行结束，可能会导致程序出错
  - i: ADD R1, R2; (R1)+(R2) ->R1，如果此时溢出？
  - i+1: MUL R3, R1; (R3)×(R1) ->R3，无效执行！
  - （此处假设有forwarding）
- ✓ 程序调试不便：
  - 程序员在第 i 条指令设置断点，但程序不能准确中断在所设置的断点处。

◘如何非精确处理？

◘如何处理多个异常？

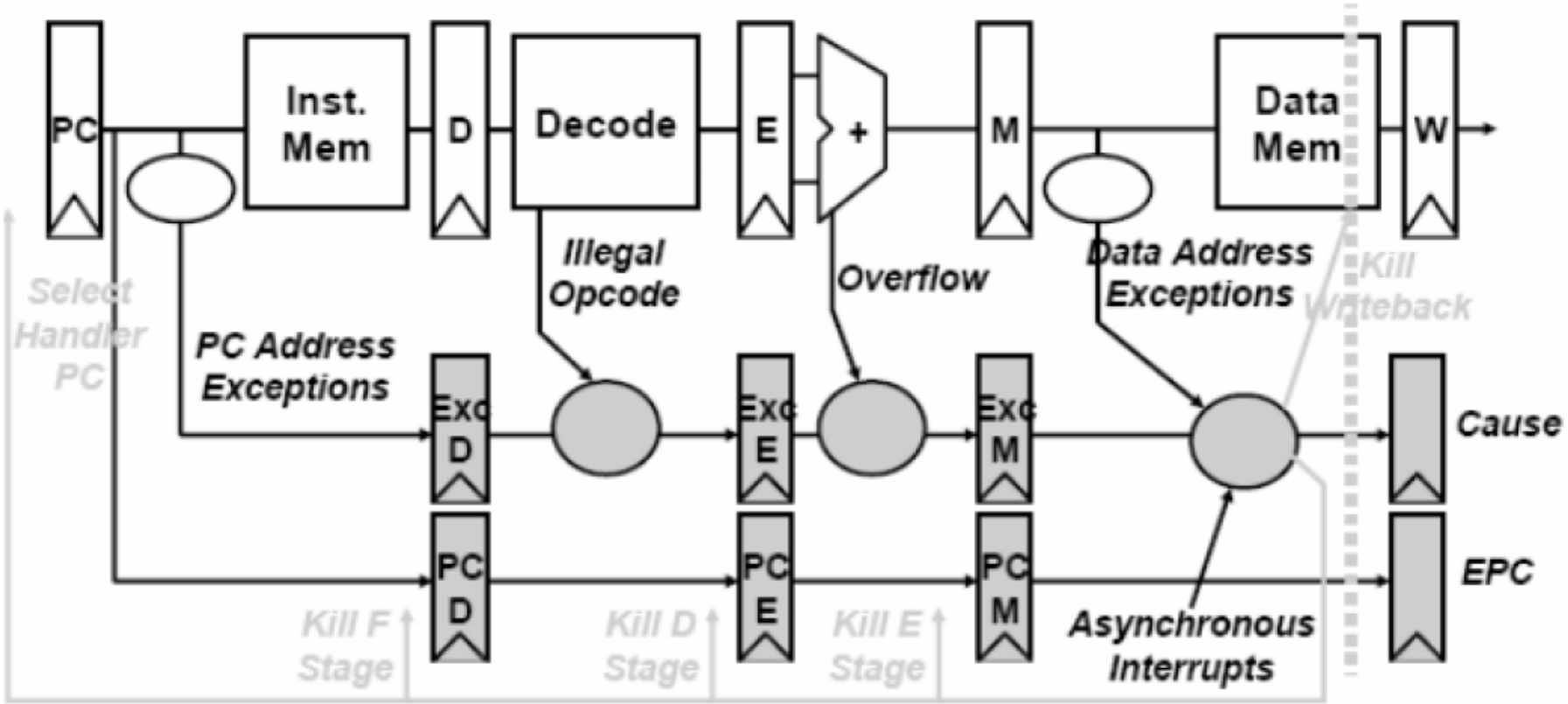


思路2：精确异常

- 实现“精确”开销大
  - ✓要保证异常指令“可重启”
    - 安全停止流水线，并完整保存当前状态
    - 需要大量后援寄存器保存流水线中各指令的现场
      - 包括RegFile、PSW、流水段寄存器（含各段的控制寄存器）！
- 例如：采用提交点技术实现“精确”异常
  - ✓提交点：M段
    - 多个异常：先发生的异常并不立即处理，只是被标记
      - EXCn寄存器：保存异常类型
      - 流水线中最深的指令引起的异常最优先

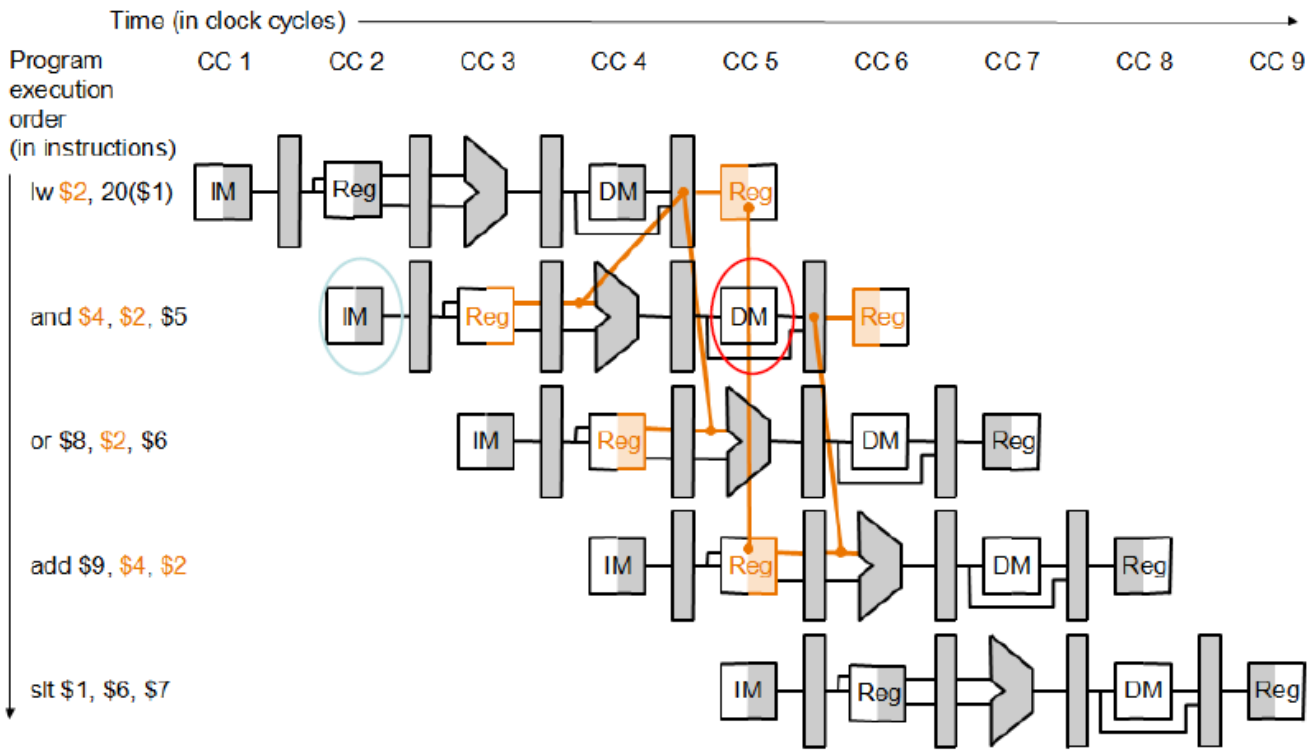
各阶段产生的异常及处理（MIPS）

- 保持流水线的异常标记直到提交点（M段）
- 如果提交点有异常，则更新cause和EPC，清除所有流水段，新PC值到fetch段
- 早期流水段的异常抑制后来的异常（flush IF/ID/EX）
- 提交点处引入中断处理



在MEM阶段处理异常的原因

- 例：如果and出现取指缺页错，直到MEM才处理
  - ✓前一条指令已执行完成（保证）
  - ✓后续指令被flush（*kill*），保证没有指令完成写回
  - ✓异常返回重新执行and



异常响应的顺序

MIPS优先级从高到低

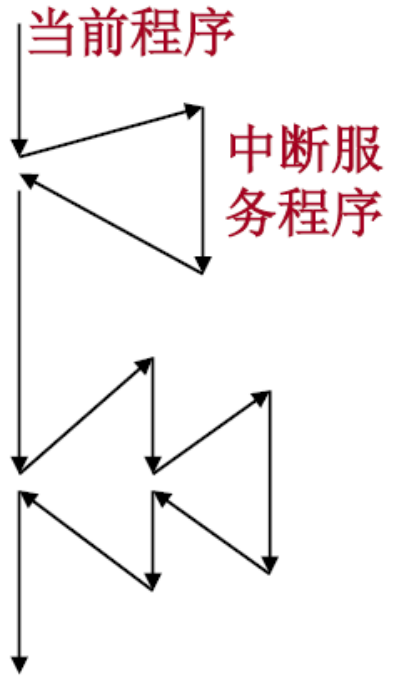
1. Reset (highest priority)
2. Soft Reset
3. Nonmaskable Interrupt (NMI)
4. Address error --Instruction fetch
5. TLB refill--Instruction fetch
6. TLB invalid--Instruction fetch
7. Cache error --Instruction fetch
8. Bus error --Instruction fetch
9. Watch - Instruction Fetch
10. Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unus-able, or Floating-Point Exception Address error--Data access
11. TLB refill --Data access
12. TLB invalid --Data access
13. TLB modified--Data write
14. Cache error --Data access
15. Watch - Data access
16. Virtual Coherency - Data access
17. Bus error -- Data access
18. Interrupt (lowest priority)

Arm优先级从高到低：

(1) Reset (highest priority)
(2) Data abort
(3) FIQ（快速中断）
(4) IRQ
(5) Prefetch abort
(6) 未定义指令, Software interrupt

总结

- 中断的概念
  - ✓ 暂停当前程序的执行，转而执行其他程序，在它们执行完成后再恢复被中断程序的执行。
  - ✓ 中断源：外部中断、内部中断
    - 中断源识别：查询法，向量法
    - 中断判优
    - 中断服务程序（ISR）：入口地址
  - ✓ 中断响应
    - 断点（nPC）：系统关键状态，隐指令完成
    - 现场（regs，PSW）：中断服务相关
    - 中断返回
  - ✓ 中断嵌套（屏蔽字）



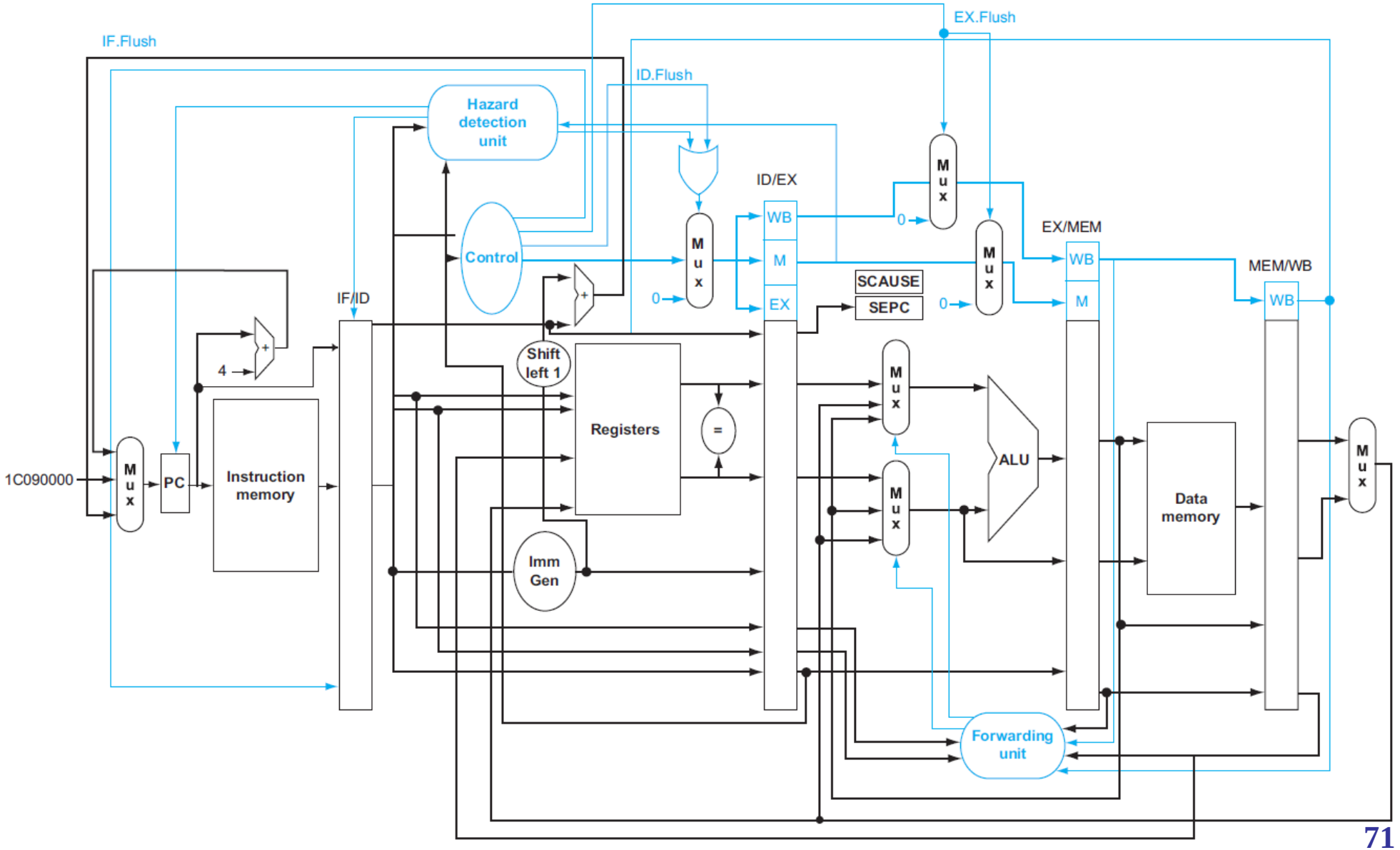
总结

- 多周期实现异常（以MIPS为例）
  - ✓ 主要过程：判断异常，保存现场，跳转执行
  - ✓ 数据通路
  - ✓ RTL代码
  - ✓ 状态机
- 流水线实现异常（MIPS、RISC-V）
  - ✓ 非精确实现：判断异常，保存现场，清空指令，跳转执行
  - ✓ 精确实现：判断异常，暂存错误，到点提交，保存现场，清空指令，跳转执行
- 多周期、精确、非精确实现的区别

思考

- 中断周期要完成哪些微操作?
- 多周期状态机中，出现溢出的指令是否将错误结果写回?
- 多周期中状态机中，如何响应中断?
- 指令顺序执行，中断“精确”；指令流水执行，中断“精确”或“非精确”可选
- 精确中断，为何提交点是M段?
- EPC和cause应该在哪个段？异常检测电路?
- mips异常返回指令eret如何实现?
- 异常与中断同时发生，优先级?
- （分支）延迟槽中的指令发生异常，EPC = ?
- 比较中断、异常、过程调用
  - ✓ 请求时间、响应时间，断点与现场，返回点，同步异步，中断周期、系统状态?

实例：RISC-V异常处理



RISC-V的工作模式

- RISC-V定义了3中工作模式（也称特权模式）
  - 机器模式（Machine Mode）
  - 监督模式（Supervisor Mode）
  - 用户模式（User Mode）
  - 通过mstatus寄存器控制

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

- M模式下的处理器（或称hart，hardware thread）对内存、IO和一些对启动和配置系统来说必要的底层功能有着完全的使用权
- M模式是所有标准RISC-V处理器都必须实现的权限模式
- 一般来说，简单的RISC-V微控制器只支持M模式

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

RISC-V控制状态寄存器-CSR

- 标准的RISC-V ISA采用12bit对CSR进行编码
  - 高4位用来表明不同特权模式下对该寄存器的读写权限
  - bit[9:8]表明访问该寄存器的最低特权模式
  - bit[11:10]表明该寄存器的读写属性
    - read/write
    - read-only

CSR Address [11:10]	[9:8]	[7:4]	Hex	Use and Accessibility
User CSRs				
00	00	XXXX	0x000-0x0FF	Standard read/write
01	00	XXXX	0x400-0x4FF	Standard read/write
10	00	XXXX	0x800-0x8FF	Custom read/write
11	00	0XXX	0xC00-0xC7F	Standard read-only
11	00	10XX	0xC80-0xCBF	Standard read-only
11	00	11XX	0xCC0-0xCFF	Custom read-only
Supervisor CSRs				
00	01	XXXX	0x100-0x1FF	Standard read/write
01	01	0XXX	0x500-0x57F	Standard read/write
01	01	10XX	0x580-0x5BF	Standard read/write
01	01	11XX	0x5C0-0x5FF	Custom read/write
10	01	0XXX	0x900-0x97F	Standard read/write
10	01	10XX	0x980-0x9BF	Standard read/write
10	01	11XX	0x9C0-0x9FF	Custom read/write
11	01	0XXX	0xD00-0xD7F	Standard read-only
11	01	10XX	0xD80-0xDBF	Standard read-only
11	01	11XX	0xDC0-0xDFF	Custom read-only
Hypervisor CSRs				
00	10	XXXX	0x200-0x2FF	Standard read/write
01	10	0XXX	0x600-0x67F	Standard read/write
01	10	10XX	0x680-0x6BF	Standard read/write
01	10	11XX	0x6C0-0x6FF	Custom read/write
10	10	0XXX	0xA00-0xA7F	Standard read/write
10	10	10XX	0xA80-0xABF	Standard read/write
10	10	11XX	0xAC0-0xAFF	Custom read/write
11	10	0XXX	0xE00-0xE7F	Standard read-only
11	10	10XX	0xE80-0xEBF	Standard read-only
11	10	11XX	0xEC0-0xEFF	Custom read-only
Machine CSRs				
00	11	XXXX	0x300-0x3FF	Standard read/write
01	11	0XXX	0x700-0x77F	Standard read/write
01	11	100X	0x780-0x79F	Standard read/write
01	11	1010	0x7A0-0x7AF	Standard read/write debug CSRs
01	11	1011	0x7B0-0x7BF	Debug-mode-only CSRs
01	11	11XX	0x7C0-0x7FF	Custom read/write
10	11	0XXX	0xB00-0xB7F	Standard read/write
10	11	10XX	0xB80-0xBBF	Standard read/write
10	11	11XX	0xBC0-0xBFF	Custom read/write
11	11	0XXX	0xF00-0xF7F	Standard read-only
11	11	10XX	0xF80-0xFBF	Standard read-only
11	11	11XX	0xFC0-0xFFF	Custom read-only

RISC-V控制状态寄存器-CSR

- 用户模式控制状态寄存器列表

Number	Privilege	Name	Description
		User Trap Setup	
0x000	URW	ustatus	User status register.
0x004	URW	uie	User interrupt-enable register.
0x005	URW	utvec	User trap handler base address.
		User Trap Handling	
0x040	URW	uscratch	Scratch register for user trap handlers.
0x041	URW	uepc	User exception program counter.
0x042	URW	ucause	User trap cause.
0x043	URW	utval	User bad address or instruction.
0x044	URW	uip	User interrupt pending.
		User Floating-Point CSRs	
0x001	URW	fflags	Floating-Point Accrued Exceptions.
0x002	URW	frm	Floating-Point Dynamic Rounding Mode.
0x003	URW	fcsr	Floating-Point Control and Status Register (frm + fflags).
		User Counter/Timers	
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC01	URO	time	Timer for RDTIME instruction.
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC03	URO	hpmcounter3	Performance-monitoring counter.
0xC04	URO	hpmcounter4	Performance-monitoring counter.
		⋮	
0xC1F	URO	hpmcounter31	Performance-monitoring counter.
0xC80	URO	cycleh	Upper 32 bits of cycle, RV32I only.
0xC81	URO	timeh	Upper 32 bits of time, RV32I only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32I only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3, RV32I only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4, RV32I only.
		⋮	
0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter31, RV32I only.

RISC-V控制状态寄存器-CSR

- 监督模式控制状态寄存器列表

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x102	SRW	sedeleg	Supervisor exception delegation register.
0x103	SRW	sideleg	Supervisor interrupt delegation register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Trap Handling			
0x140	SRW	sscratch	Scratch register for supervisor trap handlers.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor bad address or instruction.
0x144	SRW	sip	Supervisor interrupt pending.
Supervisor Protection and Translation			
0x180	SRW	satp	Supervisor address translation and protection.

Table 2.3: Currently allocated RISC-V supervisor-level CSR addresses.

RISC-V控制状态寄存器-CSR

- 机器模式控制状态寄存器列表

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.
Machine Memory Protection			
0x3A0	MRW	pmpcfg0	Physical memory protection configuration.
0x3A1	MRW	pmpcfg1	Physical memory protection configuration, RV32 only.
0x3A2	MRW	pmpcfg2	Physical memory protection configuration.
0x3A3	MRW	pmpcfg3	Physical memory protection configuration, RV32 only.
0x3B0	MRW	pmpaddr0	Physical memory protection address register.
0x3B1	MRW	pmpaddr1	Physical memory protection address register.
		⋮	
0x3BF	MRW	pmpaddr15	Physical memory protection address register.

RISC-V异常和中断

- M模式最主要的特性是拦截和处理异常的能力
- RISC-V中的异常分为两类
  - 同步异常
  - 中断
- RISC-V支持精确异常
  - 保证异常之前的所有指令都完整地执行了
  - 后续的指令都没有开始执行（或等同于没有执行）
- M模式下同步异常分类
  - 访问错误异常 当物理内存的地址不支持访问类型时发生（例如尝试写入 ROM）。
  - 断点异常 在执行 ebreak 指令，或者地址或数据与调试触发器匹配时发生。
  - 环境调用异常 在执行 ecall 指令时发生。
  - 非法指令异常 在译码阶段发现无效操作码时发生。
  - 非对齐地址异常 在有效地址不能被访问大小整除时发生，例如地址为 0x12 的 amoadd.w。

RISC-V异常和中断

Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

RISC-V异常处理相关寄存器

八个控制状态寄存器（CSR）是机器模式下异常处理的必要部分：

- mtvec（Machine Trap Vector）它保存发生异常时处理器需要跳转到的地址。
- mepc（Machine Exception PC）它指向发生异常的指令。
- mcause（Machine Exception Cause）它指示发生异常的种类。
- mie（Machine Interrupt Enable）它指出处理器目前能处理和必须忽略的中断。
- mip（Machine Interrupt Pending）它列出目前正准备处理的中断。
- mtval（Machine Trap Value）它保存了陷入（trap）的附加信息：地址例外中出错的地址、发生非法指令例外的指令本身，对于其他异常，它的值为 0。
- mscratch（Machine Scratch）它暂时存放一个字大小的数据。
- mstatus（Machine Status）它保存全局中断使能，以及许多其他的状态。

XLEN-1	XLEN-2 … 23	22	21	20	19	18	17
SD	0	TSR	TW	TVM	MXR	SUM	MPRV
1	XLEN-24	1	1	1	1	1	1

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS	FS	MPP	0	SPP	MPIE	0	SPIE	UPIE	MIE	0	SIE	UIE
2	2	2	2	1	1	1	1	1	1	1	1	1

mstatus控制状态寄存器，简单情况下，只有MIE、MPIE有效

RISC-V 异常相关指令

- RV32I相关指令及特权指令

0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	00	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I cssrrsi
csr			zimm	111	rd	1110011	I csrrci

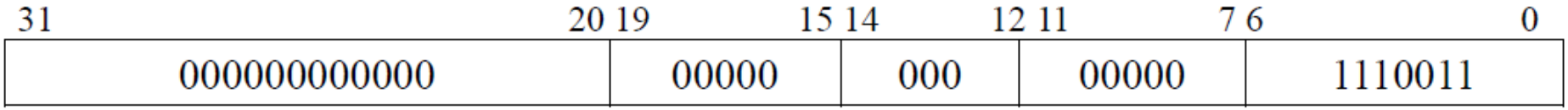
31 27	26 25	24 20	19 15	14 12	11 7	6 0	
0001000		00010	00000	000	00000	1110011	R sret
0011000		00010	00000	000	00000	1110011	R mret
0001000		00101	00000	000	00000	1110011	R wfi
0001001		rs2	rs1	000	00000	1110011	R sfence.vma

异常指令详解

ecall

环境调用 *(Environment Call)*. I-type, RV32I and RV64I.

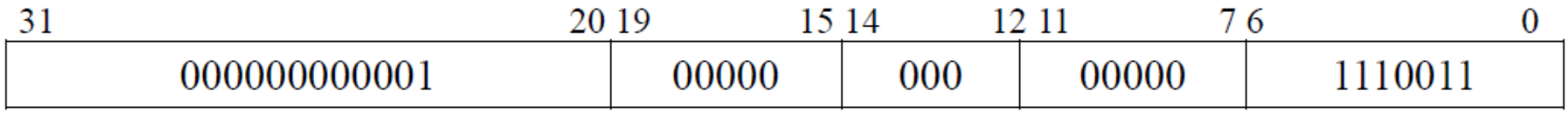
通过引发环境调用异常来请求执行环境。



ebreak

环境断点 *(Environment Breakpoint)*. I-type, RV32I and RV64I.

通过抛出断点异常的方式请求调试器。



CSR指令详解

csrrc rd, csr, rs1　　t = CSRs[csr]; CSRs[csr] = t &~x[rs1]; x[rd] = t

读后清除控制状态寄存器 *(Control and Status Register Read and Clear)*. I-type, RV32I and RV64I.

记控制状态寄存器 *csr* 中的值为 *t*。把 *t* 和寄存器 x[*rs1*]按位与的结果写入 *csr*，再把 *t* 写入 x[*rd*]。

31 20	19 15	14 12	11 7	6 0
csr	rs1	011	rd	1110011

csrrci rd, csr, zimm[4:0]　　t = CSRs[csr]; CSRs[csr] = t &~zimm; x[rd] = t

立即数读后清除控制状态寄存器 *(Control and Status Register Read and Clear Immediate)*. I-type, RV32I and RV64I.

记控制状态寄存器 *csr* 中的值为 *t*。把 *t* 和五位的零扩展的立即数 *zimm* 按位与的结果写入 *csr*，再把 *t* 写入 x[*rd*]（*csr* 寄存器的第 5 位及更高位不变）。

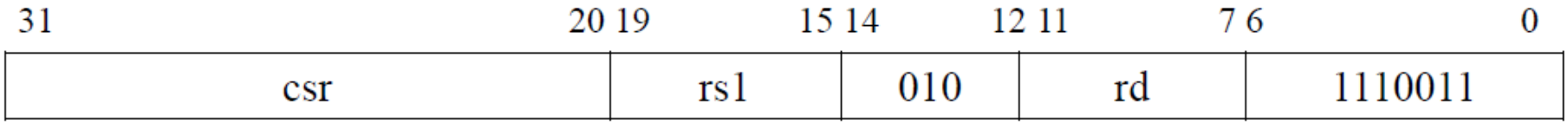
31 20	19 15	14 12	11 7	6 0
csr	zimm[4:0]	111	rd	1110011

CSR指令详解

csrrs rd, csr, rs1　　t = CSRs[csr]; CSRs[csr] = t | x[rs1]; x[rd] = t

读后置位控制状态寄存器 *(Control and Status Register Read and Set)*. I-type, RV32I and RV64I.

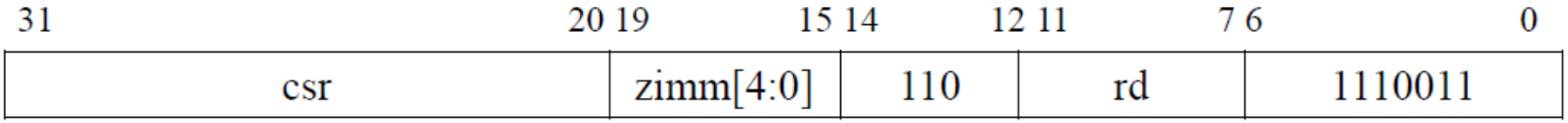
记控制状态寄存器 *csr* 中的值为 *t*。把 *t* 和寄存器 x[*rs1*]按位或的结果写入 *csr*，再把 *t* 写入 x[*rd*]。



csrrci rd, csr, zimm[4:0]　　t = CSRs[csr]; CSRs[csr] = t | zimm; x[rd] = t

立即数读后设置控制状态寄存器 *(Control and Status Register Read and Set Immediate)*. I-type, RV32I and RV64I.

记控制状态寄存器 *csr* 中的值为 *t*。把 *t* 和五位的零扩展的立即数 *zimm* 按位或的结果写入 *csr*，再把 *t* 写入 x[*rd*]（*csr* 寄存器的第 5 位及更高位不变）。

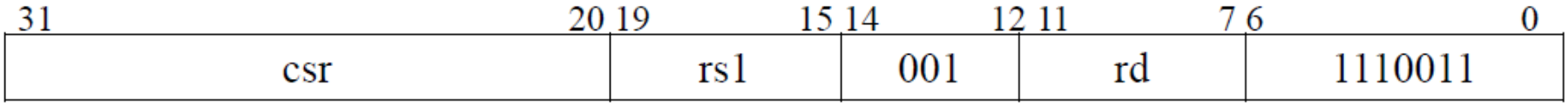


CSR指令详解

csrrw rd, csr, zimm[4:0]　　t = CSRs[csr]; CSRs[csr] = x[rs1]; x[rd] = t

读后写控制状态寄存器 *(Control and Status Register Read and Write)*. I-type, RV32I and RV64I.

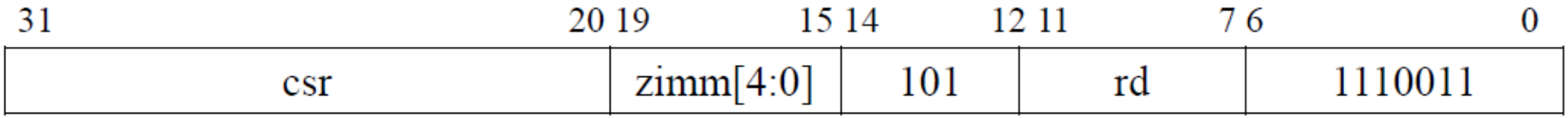
记控制状态寄存器 *csr* 中的值为 *t*。把寄存器 x[*rs1*]的值写入 *csr*，再把 *t* 写入 x[*rd*]。



csrrwi rd, csr, zimm[4:0]　　x[rd] = CSRs[csr]; CSRs[csr] = zimm

立即数读后写控制状态寄存器 *(Control and Status Register Read and Write Immediate)*. I-type, RV32I and RV64I.

把控制状态寄存器 *csr* 中的值拷贝到 x[*rd*]中，再把五位的零扩展的立即数 *zimm* 的值写入 *csr*。



特权指令详解

sret

ExceptionReturn(Supervisor)

管理员模式例外返回*(Supervisor-mode Exception Return)*. R-type, RV32I and RV64I 特权指令。从管理员模式的例外处理程序中返回，设置 *pc* 为 CSRs[spec]，权限模式为 CSRs[sstatus].SPP，CSRs[sstatus].SIE 为 CSRs[sstatus].SPIE，CSRs[sstatus].SPIE 为 1，CSRs[sstatus].spp 为 0。

31 25	24 20	19 15	14 12	11 7	6 0
0001000	00010	00000	000	00000	1110011

mret

ExceptionReturn(Machine)

机器模式异常返回*(Machine-mode Exception Return)*. R-type, RV32I and RV64I 特权架构从机器模式异常处理程序返回。将 *pc* 设置为 CSRs[mepc]，将特权级设置成 CSRs[mstatus].MPP, CSRs[mstatus].MIE 置成 CSRs[mstatus].MPIE，并且将 CSRs[mstatus].MPIE 为 1;并且，如果支持用户模式，则将 CSR [mstatus].MPP 设置为 0。

31 25	24 20	19 15	14 12	11 7	6 0
0011000	00010	00000	000	00000	1110011

特权指令详解

wfi

while (noInterruptPending) idle

等待中断*(Wait for Interrupt)*. R-type, RV32I and RV64I 特权指令。

如果没有待处理的中断，则使处理器处于空闲状态。

31 25	24 20	19 15	14 12	11 7	6 0
0001000	00101	00000	000	00000	1110011

sfence.vma rs1, rs2

Fence(Store, AddressTranslation)

虚拟内存屏障*(Fence Virtual Memory)*. R-type, RV32I and RV64I 特权指令。

根据后续的虚拟地址翻译对之前的页表存入进行排序。当 *rs2*=0 时，所有地址空间的翻译都会受到影响；否则，仅对 x[*rs2*]标识的地址空间的翻译进行排序。当 *rs1*=0 时，对所选地址空间中的所有虚拟地址的翻译进行排序；否则，仅对其中包含虚拟地址 x[*rs1*]的页面地址翻译进行排序。

31 25	24 20	19 15	14 12	11 7	6 0
0001001	rs2	rs1	000	00000	1110011

控制状态寄存器详解

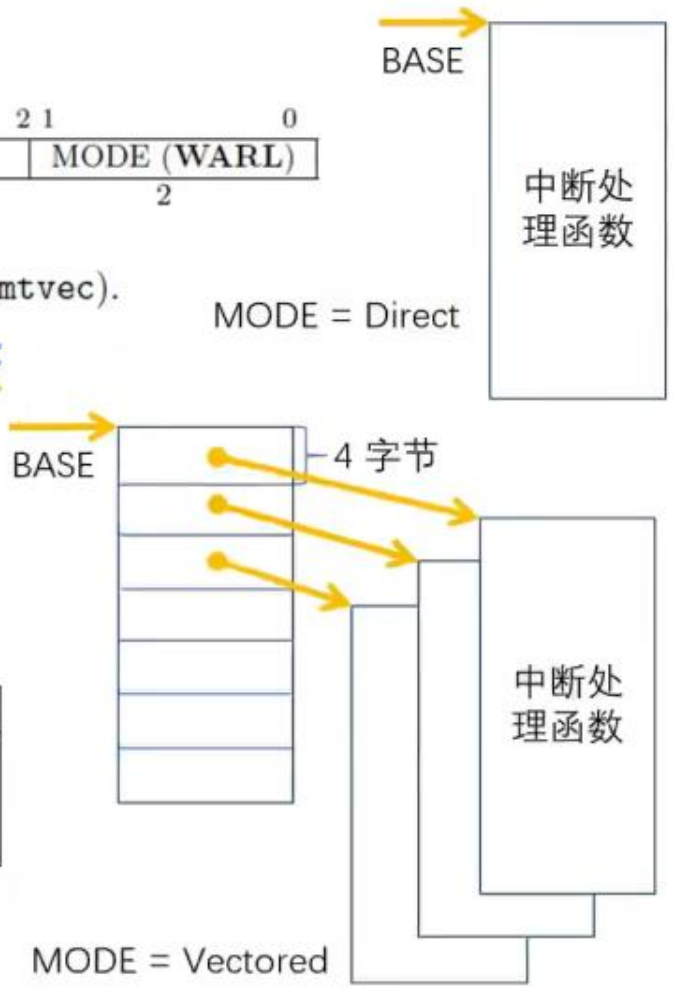
mtvec (Machine Trap-Vector Base-Address)

【参考 2】Figure 3.8: Machine trap-vector base-address register (mtvec).

- BASE：trap 入口函数的基地址，必须保证四字节对齐
- MODE：进一步用于控制入口函数的地址配置方式：
  - Direct：所有的 exception 和 interrupt 发生后 PC 都跳转到 BASE 指定的地址处。
  - Vectored：exception 处理方式同 Direct；但 interrupt 的入口地址以数组方式排列。

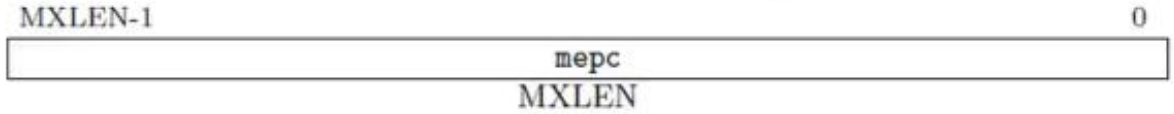
Value	Name	Description
0	Direct	All exceptions set pc to BASE.
1	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
≥2	—	*Reserved*

【参考 2】Table 3.5: Encoding of mtvec MODE field.



控制状态寄存器详解

mepc（Machine Exception Program Counter）



【参考 2】Figure 3.21: Machine exception program counter register.

- 当 trap 发生时，pc 会被替换为 mtvec 设定的地址，同时 hart 会设置 mepc 为当前指令或者下一条指令的地址，当我们需要退出 trap 时可以调用特殊的 mret 指令，该指令会将 mepc 中的值恢复到 pc 中（实现返回的效果）。
- 在处理 trap 的程序中我们可以修改 mepc 的值达到改变 mret 返回地址的目的。

控制状态寄存器详解

mcause（Machine Cause）

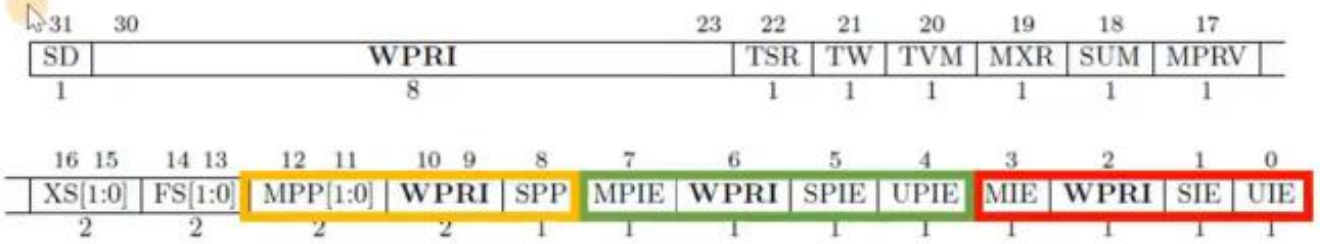
【参考 2】Figure 3.22: Machine Cause register mcause.

- 当 trap 发生时，hart 会设置该寄存器通知我们 trap 发生的原因。
- 最高位 Interrupt 为 1 时标识了当前 trap 为 interrupt，否则是 exception。
- 剩余的 Exception Code 用于标识具体的 interrupt 或者 exception 的种类。

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

控制状态寄存器详解

mstatus（Machine Status）



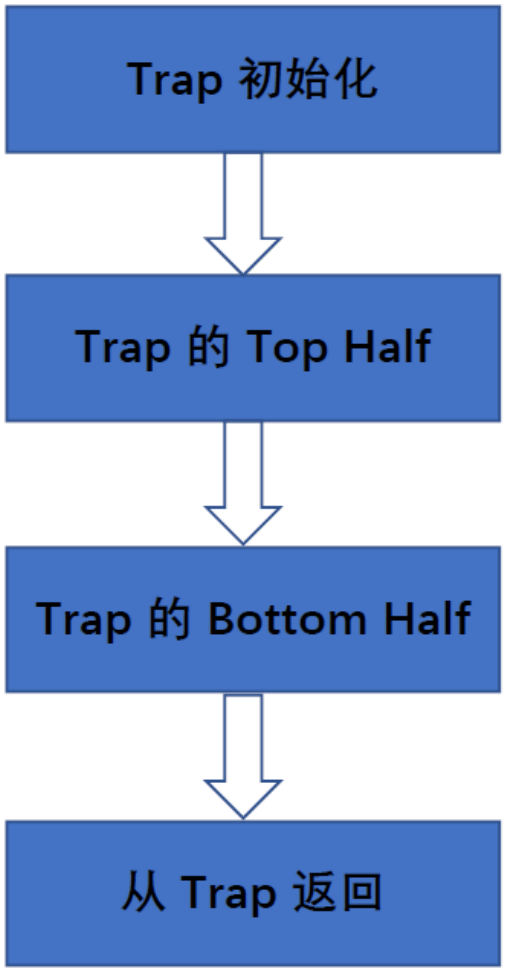
【参考 2】Figure 3.6: Machine-mode status register (mstatus) for RV32.

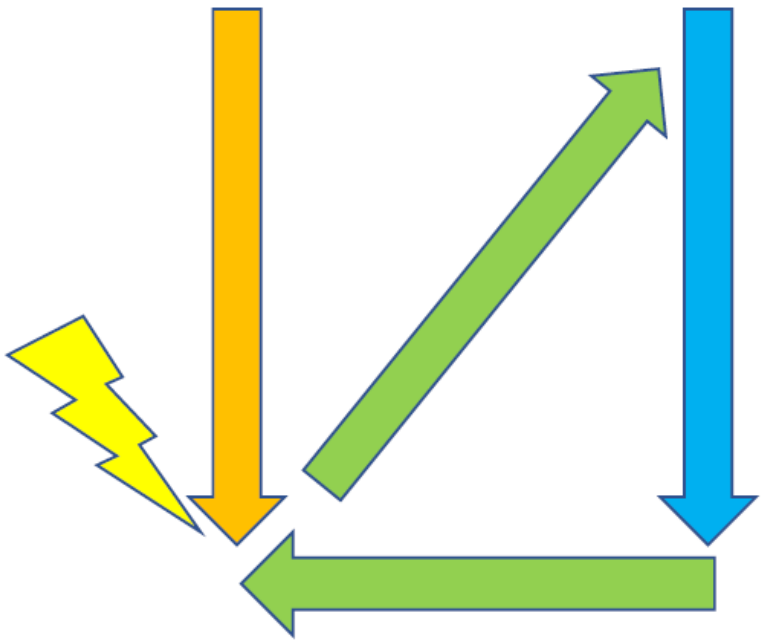
- xIE（x=M/S/U）：分别用于打开（1）或者关闭（0） M/S/U 模式下的全局中断。当 trap 发生时， hart 会自动将 xIE 设置为 0。
- xPIE（x=M/S/U）:当 trap 发生时用于保存 trap 发生之前的 xIE 值。
- xPP（x=M/S）:当 trap 发生时用于保存 trap 发生之前的权限级别值。注意没有 UPP。
- 其他标志位涉及内存访问权限、虚拟内存控制等，暂不考虑。

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

RISC-V异常处理流程

- 在RISC-V中，异常也称为陷入(Trap)

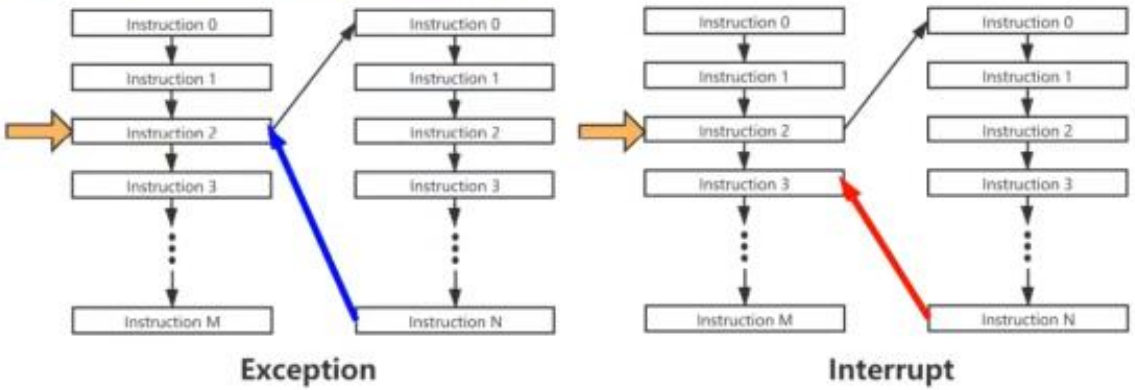




RISC-V异常处理流程：硬件部分，Top Half

Trap 发生，Hart 自动执行如下状态转换

- 把 mstatus 的 MIE 值复制到 MPIE 中，清除 mstatus 中的 MIE 标志位，效果是中断被禁止。
- 设置 mepc，同时 PC 被设置为 mtvec。（需要注意的是，对于 exception， mepc 指向导致异常的指令；对于 interrupt，它指向被中断的指令的下一条指令的位置。）



- 根据 trap 的种类设置 mcause，并根据需要为 mtval 设置附加信息。
- 将 trap 发生之前的权限模式保存在 mstatus 的 MPP 域中，再把 hart 权限模式更改为 M（也就是说无论在任何 Level 下触发 trap，hart 首先切换到 Machine 模式）。

RISC-V异常处理流程：软件部分 Bottom Half

- 保存（save）当前控制流的上下文信息（利用 mscratch）
- 调用 C 语言的 trap handler
- 从 trap handler 函数返回，mepc 的值有可能需要调整
- 恢复（restore）上下文的信息
- 执行 MRET 指令返回到 trap 之前的状态。

```
trap_vector:
        # save context(registers).
        csrrw   t6, mscratch, t6        # swap t6 and mscratch
        reg_save t6

        # Save the actual t6 register, which we swapped into
        # mscratch
        mv      t5, t6          # t5 points to the context of current task
        csrr    t6, mscratch    # read t6 back from mscratch
        sw      t6, 120(t5)     # save t6 with t5 as base

        # Restore the context pointer into mscratch
        csrw    mscratch, t5

        # call the C trap handler in trap.c
        csrr    a0, mepc
        csrr    a1, mcause
        call    trap_handler

        # trap_handler will return the return address via a0.
        csrw    mepc, a0

        # restore context(registers).
        csrr    t6, mscratch
        reg_restore t6

        # return to whatever we were doing before trap.
        mret
```

RISC-V异常处理代码详解

```
#include "platform.h"

    # size of each hart's stack is 1024 bytes
    .equ    STACK_SIZE, 1024

    .global _start

    .text
_start:
    # park harts with id != 0
    csrr    t0, mhartid     # read current hart id
    mv  tp, t0          # keep CPU's hartid in its tp for later us
    bnez    t0, park        # if we're not on the hart 0
                    # we park the hart

    # Set all bytes in the BSS section to zero.
    la  a0, _bss_start
    la  a1, _bss_end
    bgeu    a0, a1, 2f
1:
    sw  zero, (a0)
    addi    a0, a0, 4
    bltu    a0, a1, 1b
2:
    # Setup stacks, the stack grows from bottom to top, so we put
    # stack pointer to the very end of the stack range.
    slli    t0, t0, 10      # shift left the hart id by 1024
    la  sp, stacks + STACK_SIZE # set the initial stack pointer
                    # to the end of the first stack space
    add sp, sp, t0      # move the current hart stack pointer
                    # to its place in the stack space

    j   start_kernel        # hart 0 jump to c

park:
    wfi
    j   park

stacks:
    .skip   STACK_SIZE * MAXNUM_CPU # allocate space for all the harts stacks

    .end                # End of file
```

```
#include "os.h"

/*
 * Following functions SHOULD be called ONLY ONE time here,
 * so just declared here ONCE and NOT included in file os.h.
 */
extern void uart_init(void);
extern void page_init(void);
extern void sched_init(void);
extern void schedule(void);
extern void os_main(void);
extern void trap_init(void);

void start_kernel(void)
{
    uart_init();
    uart_puts("Hello, RVOS!\n");

    page_init();

    trap_init();

    sched_init();

    os_main();

    schedule();

    uart_puts("Would not go here!\n");
    while (1) {}; // stop here!
}
```

RISC-V异常处理代码详解

```c
#include "os.h"

extern void trap_vector(void);

void trap_init()
{
    /*
     * set the trap-vector base-address for machine-mode
     */
    w_mtvec((reg_t)trap_vector);
}

reg_t trap_handler(reg_t epc, reg_t cause)
{
    reg_t return_pc = epc;
    reg_t cause_code = cause & 0xfff;

    if (cause & 0x80000000) {
        /* Asynchronous trap - interrupt */
        switch (cause_code) {
        case 3:
            uart_puts("software interruption!\n");
            break;
        case 7:
            uart_puts("timer interruption!\n");
            break;
        case 11:
            uart_puts("external interruption!\n");
            break;
        default:
            uart_puts("unknown async exception!\n");
            break;
        }
    } else {
        /* Synchronous trap - exception */
        printf("Sync exceptions!, code = %d\n", cause_code);
        panic("OOPS! What can I do!");
        //return_pc += 4;
    }

    return return_pc;
}
```

```asm
# interrupts and exceptions while in mach
.globl trap_vector
# the trap vector base address must alway
.align 4
trap_vector:
    # save context(registers).
    csrrw   t6, mscratch, t6    # swap t6
    reg_save t6

    # Save the actual t6 register, which
    # mscratch
    mv  t5, t6      # t5 points to the co
    csrr    t6, mscratch    # read t6 bac
    sw  t6, 120(t5) # save t6 with t5 as

    # Restore the context pointer into ms
    csrw    mscratch, t5

    # call the C trap handler in trap.c
    csrr    a0, mepc
    csrr    a1, mcause
    call    trap_handler

    # trap_handler will return the return
    csrw    mepc, a0

    # restore context(registers).
    csrr    t6, mscratch
    reg_restore t6

    # return to whatever we were doing be
    mret
```

RISC-V异常处理代码详解

```c
void start_kernel(void)
{
    uart_init();
    uart_puts("Hello, RVOS!\n");

    page_init();

    trap_init();

    sched_init();

    os_main();

    schedule();

    uart_puts("Would not go here!\n");
    while (1) {}; // stop here!
}
```

```c
void sched_init()
{
    w_mscratch(0);
}
```

```c
void task_yield()
{
    schedule();
}
```

```c
/* NOTICE: DON'T LOOP INFINITELY IN main() */
void os_main(void)
{
    task_create(user_task0);
    task_create(user_task1);
}
```

```c
void schedule()
{
    if (_top <= 0) {
        panic("Num of task should be greater than zero!");
        return;
    }

    _current = (_current + 1) % _top;
    struct context *next = &(ctx_tasks[_current]);
    switch_to(next);
}
```

```c
int task_create(void (*start_routin)(void))
{
    if (_top < MAX_TASKS) {
        ctx_tasks[_top].sp = (reg_t) &task_stack[_top][STACK_SIZE - 1];
        ctx_tasks[_top].ra = (reg_t) start_routin;
        _top++;
        return 0;
    } else {
        return -1;
    }
}
```

```c
void user_task0(void)
{
    uart_puts("Task 0: Created!\n");
    while (1) {
        uart_puts("Task 0: Running...\n");

        trap_test();

        task_delay(DELAY);
        task_yield();
    }
}

void user_task1(void)
{
    uart_puts("Task 1: Created!\n");
    while (1) {
        uart_puts("Task 1: Running...\n");
        task_delay(DELAY);
        task_yield();
    }
}

/* NOTICE: DON'T LOOP INFINITELY IN main() */
void os_main(void)
{
    task_create(user_task0);
    task_create(user_task1);
}
```

RISC-V异常处理代码详解

- 异常测试
  - 向非法地址写入数据，code = 7

```
44  void trap_test()
45  {
46      /*
47       * Synchronous exception code = 7
48       * Store/AMO access fault
49       */
50      *(int *)0x00000000 = 100;
51
52      /*
53       * Synchronous exception code = 5
54       * Load access fault
55       */
56      //int a = *(int *)0x00000000;
57
58      uart_puts("Yeah! I'm return back from trap!\n");
59  }
```

```
Hello, RVOS!
HEAP_START = 800070d4, HEAP_SIZE = 07ff8f2c, num of pages = 32752
TEXT:   0x80000000 -> 0x8000337c
RODATA: 0x8000337c -> 0x8000365a
DATA:   0x80004000 -> 0x80004004
BSS:    0x80004004 -> 0x800070d4
HEAP:   0x80010000 -> 0x88000000
Task 0: Created!
Task 0: Running...
Sync exceptions!, code = 7
panic: OOPS! What can I do!
```

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

参考资料

- 辅助教材：唐书

- staff.ustc.edu.cn/~llxx
- staff.ustc.edu.cn/~cswang
- B站：循序渐进，学习开发一个RISC-V上的操作系统
- RISC-V指令手册，卷I，非特权ISA，中文版
- RISC-V指令手册，卷II，特权架构，英文版

- 对于上述作者，一并感谢！

中国科学技术大学
University of Science and Technology of China

计算机组成原理
CH8_总线与外设

卢建良
lujl@ustc.edu.cn
2022年春季学期

提纲

- 总线系统
  - 总线概述
  - 总线结构
  - 总线仲裁
  - 总线通信
- 外设与IO
  - IO系统概述
  - IO接口
  - IO信息交换方式
  - IO设备
- RISC-V架构典型SOC
  - FE310-G000

总线的基本概念

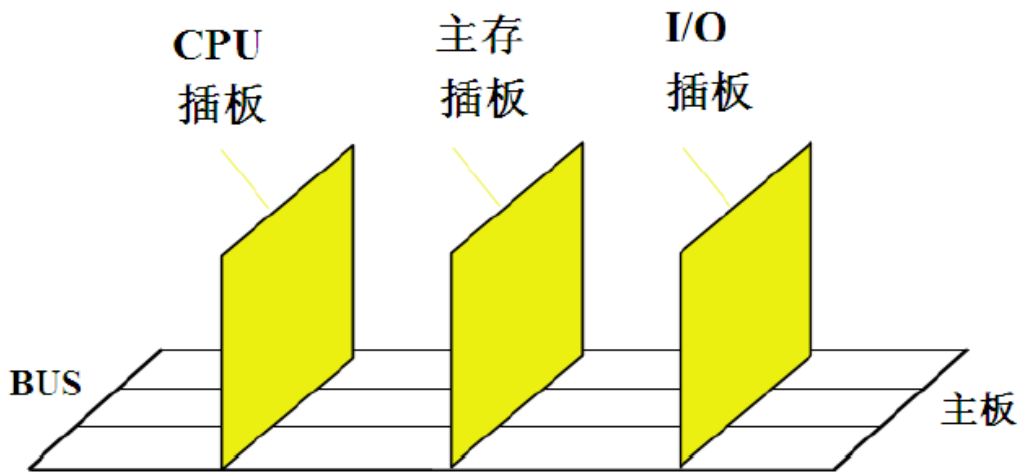
- 构成计算机系统的互连结构，是连接系统中多个部件的信息传输线
- 实现计算机各个部件地址、数据和控制信息的交换，并在争用资源的基础上进行工作
  - ✓ 某一时刻，只允许有一个部件向总线发送信息，多个部件可以同时从总线上接收相同信息
- 总线的信息传送
  - ✓ 由许多传输线或通路组成，每条线可以传输一位二进制代码

总线的分类

- 按数据传送方式划分
  - ✓ 并行总线（又可按数据宽度细分）
  - ✓ 串行总线
- 按总线的使用范围划分
  - ✓ 计算机总线
  - ✓ 测控总线
  - ✓ 网络通信总线等
- 按时钟同步/异步划分
  - ✓ 总线上的数据与时钟同步工作的总线称为同步总线
  - ✓ 与时钟不同步工作的总线称为异步总线
- 单机系统中，按连接部件不同划分
  - ✓ 内部总线：连接CPU内部各寄存器及运算部件
  - ✓ 系统总线：连接CPU同计算机系统的其他高速功能部件，以及中低速I/O设备（I/O总线）

总线的分类

- 系统总线
  - ✓ CPU、主存、I/O设备各大部件之间的信息传输线，也叫板级总线



  - ✓ 系统总线按传输信息不同
    - 数据总线：用于传输各功能部件之间的数据信息，双向传输总线
    - 地址总线：用来指出数据总线上的数据在主存单元或I/O设备的地址，单向传输总线，由CPU发出
    - 控制总线：用来发出各种控制信号，单向传输线

总线的分类

- 常见的控制信号
  - ✓时钟：用来同步各种操作
  - ✓复位：初始化所有部件
  - ✓总线请求：表示某部件需获得总线使用权
  - ✓总线允许：表示需要获得总线使用权的部件已被允许
  - ✓中断请求：表示某部件提出中断请求
  - ✓中断响应：表示中断请求已经被接收
  - ✓存储器写：将数据总线的数据写至存储器的指定地址单元
  - ✓存储器读：将指定存储单元中的数据读到数据总线上
  - ✓I/O读：从指定的I/O端口将数据读到数据总线上
  - ✓I/O写：将数据总线上的数据输出到指定的I/O端口

总线的特性和性能指标

总线的特性

✓ 物理特性

- 总线的物理连接方式，包括总线的根数、插头、插座形状、引脚线个数及排列方式等

✓ 功能特性

- 描述总线中每一根线的功能，如地址总线、数据总线、控制总线

✓ 电气特性

- 定义每一根线上信号的传递方向及有效电平范围
- 送入CPU的信号称为输入信号，CPU发出的信号称为输出信号

✓ 时间特性

- 定义每根线在什么时间有效，即规定总线各信号的有效时序关系

总线的特性和性能指标

- 总线的性能指标
  - ✓总线宽度：通常指数据总线的位数
  - ✓总线频率：1/传输一次数据时间
  - ✓总线带宽：总线的数据传输速率，即单位时间内总线传输数据的位数
    - 通常用每秒传输信息的字节数来衡量
  - ✓总线复用：一条信号线上分时传送多种信号
  - ✓其他指标：如负载能力、电源电压、总线宽度扩展等

几种传统的微型计算机总线性能

名称	ISA (PC-AT)	EISA	STD	VESA (VL-BUS)	MCA	PCI
适用机型	80286,386, 486 系列机	386,486,586 IBM 系列机	Z-80,V20, V40 IBM-PC 系列机	i486，PC-AT 兼容机	IBM 个人机与工作站	P5 个人机，PowerPC, Alpha 工作站
最大传输率	15MB/s	33MB/s	2MB/s	266MB/s	40MB/s	133MB/s
总线宽度	16 位	32 位	8 位	32 位	32 位	32 位
总线工作频率	8MHz	8.33MHz	2MHz	66MHz	10MHz	0~33MHz
同步方式	同步			异步	同步	
仲裁方式	集中	集中	集中	集中		
地址宽度	24	32	20			32/64
负载能力	8	6	无限制	6	无限制	3
信号线数		143		90	109	49
64 位扩展	不可	无规定	不可	可	可	可
并发工作				可		可
引脚使用	非多路复用	非多路复用	非多路复用	非多路复用		多路复用

总线性能指标例题

- 例 （1）某总线在一个总线周期中并行传送4个字节数据，假设一个总线周期等于一个总线时钟周期，总线时钟频率为33MHz，总线带宽多少？

  （2）如果一个总线周期中并行传送64位数据，总线时钟频率升为66MHz，总线带宽多少？

解：

（1）设总线带宽用D_r表示，总线时钟周期为T = 1/f，一个总线周期传送的数据量用D表示，则有

$$D_r = D/T = D \times f = 4B \times 33MHz = 132MB/s$$

（2）D = 64b = 8B

$$D_r = D/T = D \times f = 8B \times 66MHz = 528MB/s$$

总线标准

- 总线标准
  - ✓ 指系统与各功能模块、模块与模块之间的一个互连的标准规范
  - ✓ 基于总线标准连接的两个模块，只需根据标准的要求完成自身一方接口功能要求，无须了解对方接口与总线的连接要求
    - 按总线标准设计的接口被视为通用接口，有利于计算机接口软硬件设计
- ISA总线**Industry Standard Architecture**
  - ✓ IBM为采用全16位CPU而推出的，又称AT总线
  - ✓ 使用独立的总线时钟，使得CPU时钟频率可以比总线高
  - ✓ 不支持总线仲裁，不能支持多台主设备系统
  - ✓ 所有数据传送必须通过CPU或DMA（直接存储访问）接口来管理
  - ✓ 总线时钟8MHz，最大传输率16MBps，数据线16位，地址线24位

总线标准

EISA总线 Extended ISA

- ✓ 在ISA基础上扩展开放的总线标准，与ISA完全兼容
- ✓ 从CPU分离出总线控制权，支持多个总线主控器和突发方式的传输
- ✓ 总线时钟频率8MHz，最大传输率可达33MBps
- ✓ 数据总线32位，地址总线32位，扩展DMA访问范围达2^{32}

PCI总线 Peripheral Component Interconnect

- ✓ 外围部件互连总线，Intel于1991年首推
  - 独立于CPU时钟，采用33MHz和66MHz的总线时钟
  - 数据线32位，可扩展到64位；传输速率从132MBps到528MBps
  - 支持**突发工作方式（Burst Mode）**

指若被传送的数据在主存中连续存放，则在访问此组数据时，只需给出第一个数据地址，占用一个时钟周期，其后每个数据的传送各占一个时钟周期，而不必每次给出各个数据的地址。

总线标准

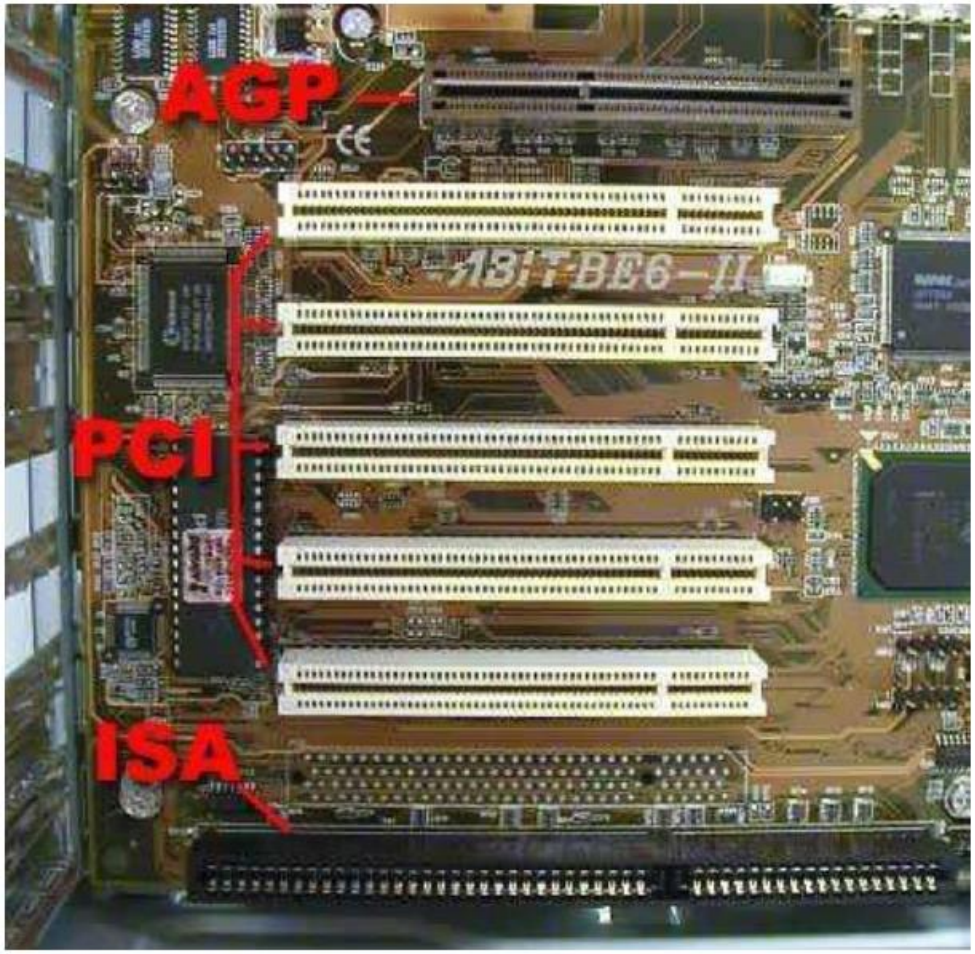
PCI总线的特点

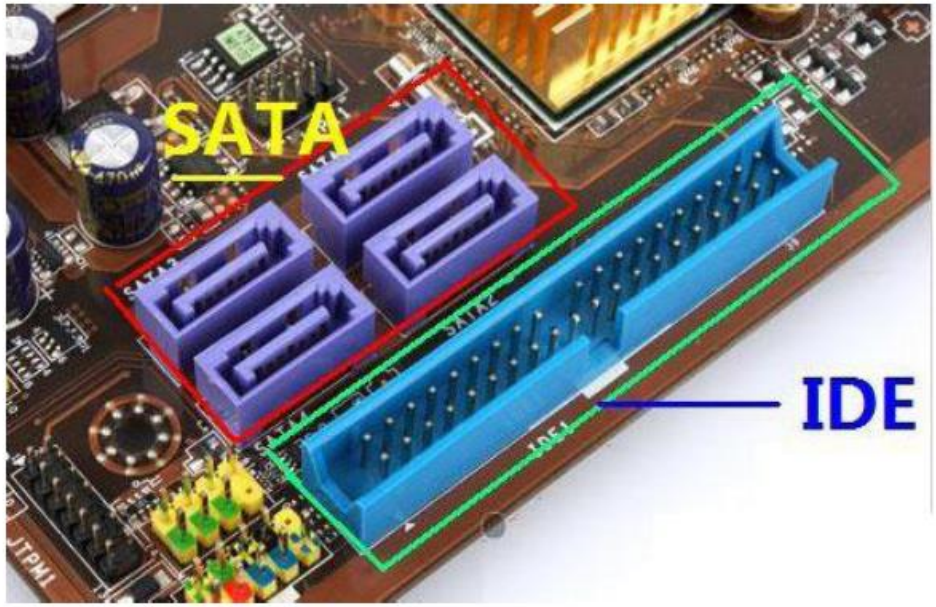
- ✓ 良好的兼容性
  - PCI总线部件和插件接口独立于处理器，支持所有目前和未来不同结构的处理器
  - 与ISA/EISA总线兼容，可转换
- ✓ 支持即插即用
  - 配有存放设备具体信息的寄存器
- ✓ 支持多主设备能力
- ✓ 具有与处理器和存储子系统完全并行操作的能力
  - PCI总线可视为CPU和外设之间的中间层
- ✓ 提供数据和地址校验功能
- ✓ 支持两种电压标准：5V和3.3V
- ✓ 采用多路复用技术

总线标准

- AGP总线**Accelerated Graphics Port—加速图形接口**
  - ✓ 加速图形端口总线，显示卡专用的局部总线
  - ✓ 采用点对点通道方式，以66.7MHz的频率直接与主存联系
  - ✓ 最大传输率从266MBps、533MBps到2.1GBps
- STD：STD总线于1987年被IEEE列为标准（IEEE961标准）
  - ✓ 主要用于以微处理器为中心的工业控制领域。
  - ✓ 数据总线8位，最大传输率2MB/S。
- SCSI：Small Computer System Interface—小型计算机系统接口
  - ✓ 主要用于光驱、音频设备、扫描仪、打印机以及像硬盘驱动器这样的大容量存储设备等的连接，是一种直接连接外设的并行I/O总线。
- USB总线
  - ✓ 通用串行总线标准
  - ✓ 基于通用连接技术，实现外设的简单快速连接
  - ✓ 真正的即插即用特征：不断电安装和拆卸
  - ✓ 可链式连接127个外设到同一系统，标准USB电缆3m，通过链式连接可达30m
  - ✓ 数据传输率有1.5Mbps、12Mbps和480Mbps
  - ✓ 4芯连接电缆，2条用于信号连接，2条用于电源和接地

总线标准





典型总线标准的比较

总线标准	数据线	总线时钟	带宽
ISA	16	8 MHz（独立）	16 MBps
EISA	32	8 MHz（独立）	33 MBps
VESA (VL-BUS)	32	33 MHz（CPU）	133 MBps
PCI	32 64	33 MHz（独立） 66 MHz（独立）	132 MBps 528 MBps
AGP	32	66.7 MHz（独立） 133 MHz（独立）	266 MBps 533 MBps
RS-232	串行通信 总线标准	数据终端设备（计算机）和数据通信设备（调制解调器）之间的标准接口	
USB	串行接口 总线标准	普通无屏蔽双绞线 带屏蔽双绞线 最高	1.5 Mbps (USB1.0) 12 Mbps (USB1.0) 480 Mbps (USB2.0)

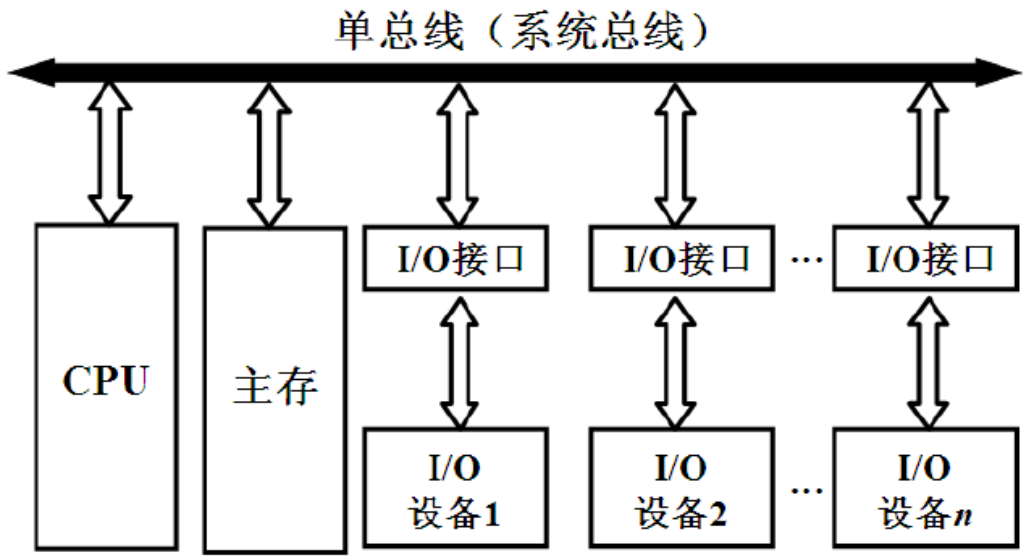
单总线结构

◘单总线结构

✓使用单一系统总线来连接CPU、主存和I/O设备

◘特点：

- ✓要求连接到总线上的部件必须**高速运行**完成操作，迅速放弃总线控制权
- ✓CPU发出的地址，不仅加至主存，也同时加至总线上的所有外设
- ✓对IO设备的操作与主存操作一样,可以指定地址
- ✓易于扩展成多CPU系统

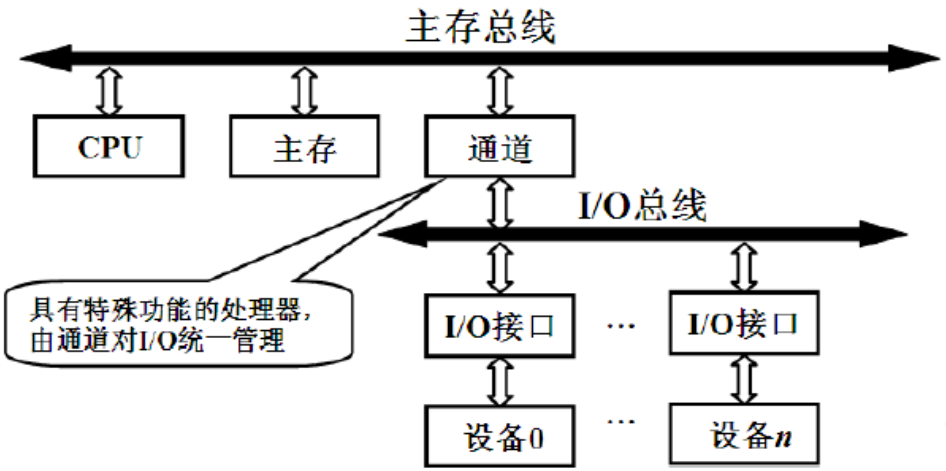


多总线结构

多总线结构

- ✓在CPU、主存、I/O之间互联采用多条总线

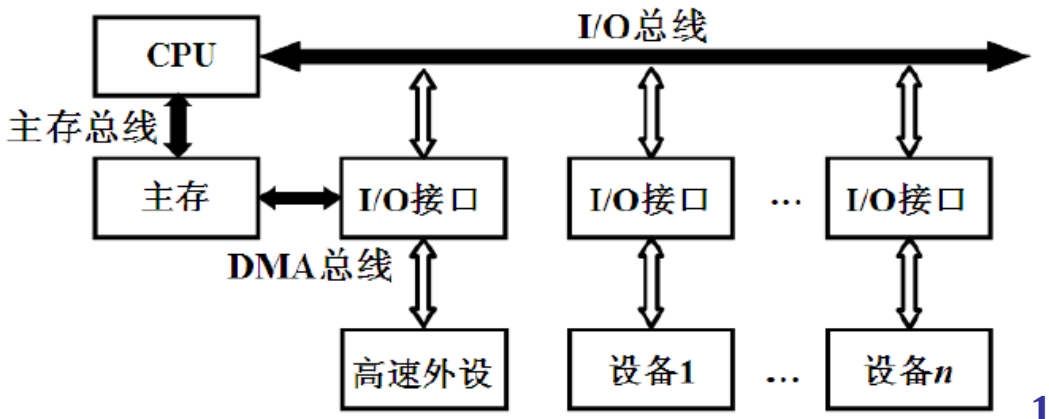
双总线结构



特点：将速度较低的I/O设备从单总线上分离，形成主存总线与I/O总线分开的结构

多用于大、中型计算机系统

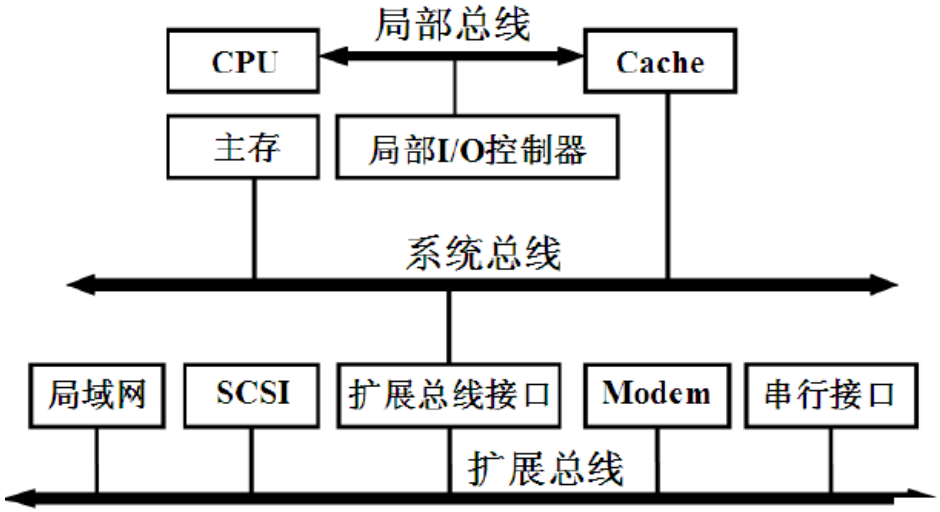
三总线结构



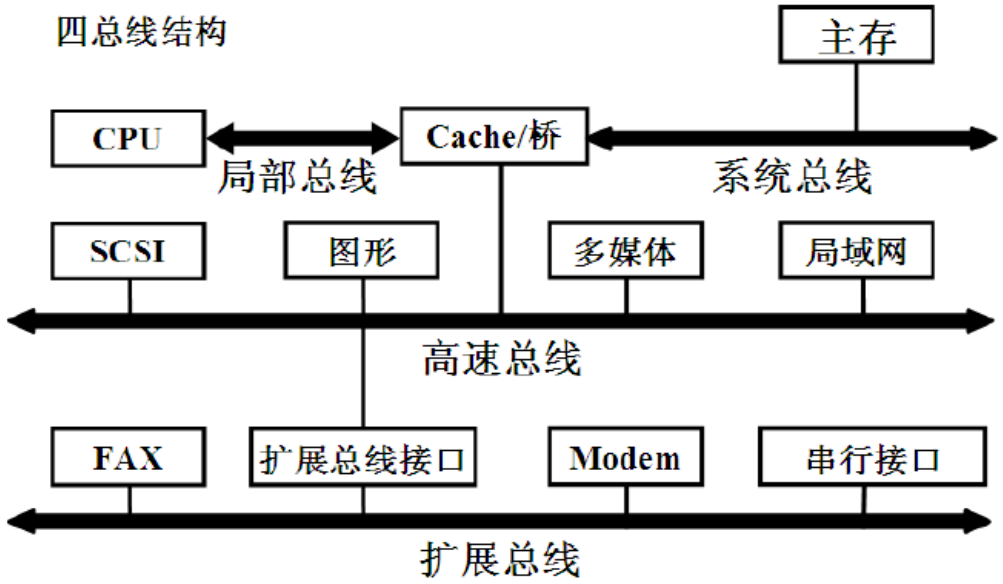
特点：在双总线的基础上，进一步地将I/O设备按速率不同进行分类，形成多总线结构

多总线结构

三总线结构的又一形式

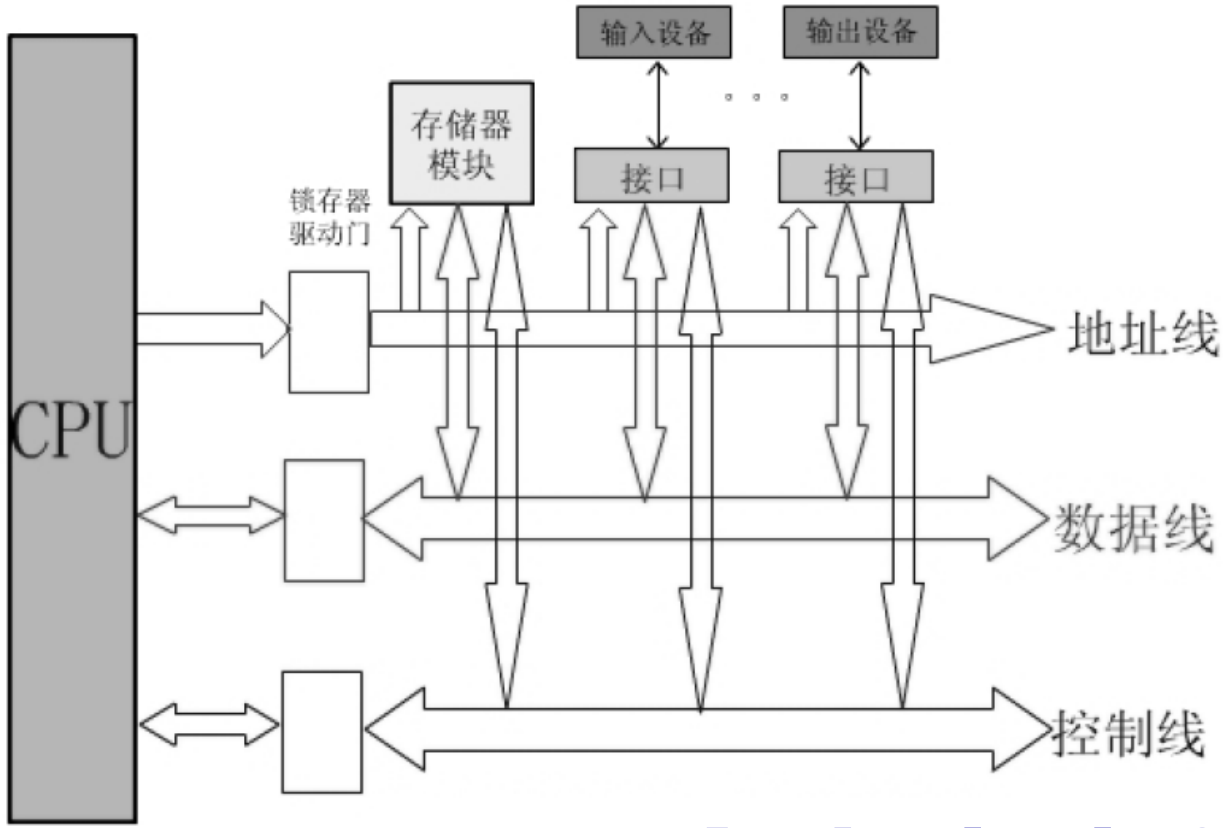


四总线结构



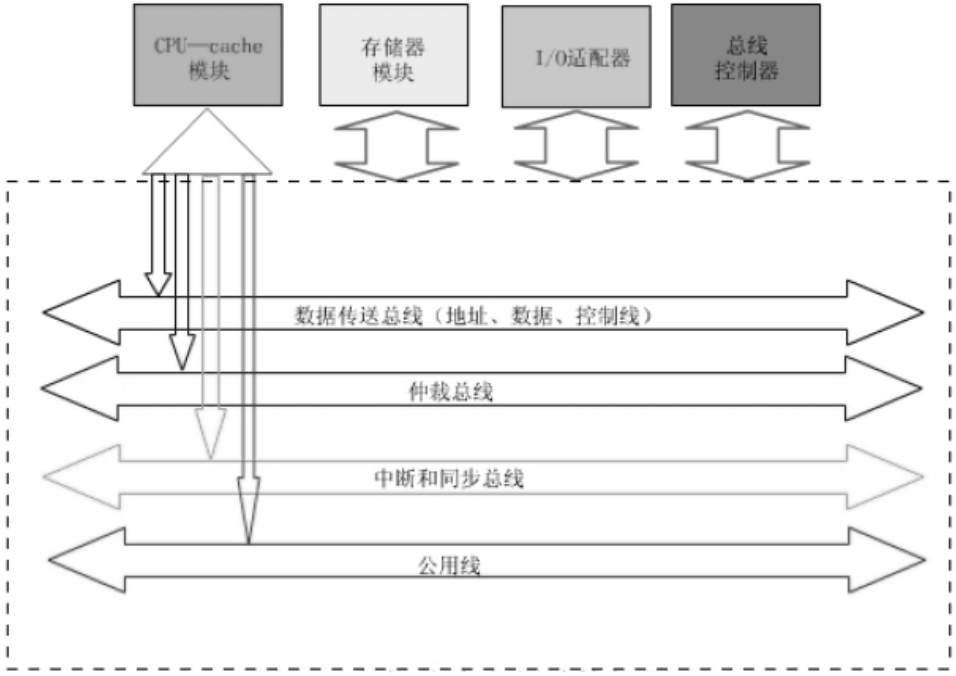
总线内部结构

- 早期总线的内部结构
  - ✓处理器芯片引脚的延伸，是处理器与I/O设备适配器的通道
  - ✓不足之处：
    - 总线结构与CPU密切相关，通用性差



总线内部结构

- 现行总线内部结构
  - ✓ 标准总线，与结构、CPU等无关
  - ✓ CPU连同其Cache一起作为一个模块与总线相连
  - ✓ 四部分组成：
    - 数据传送总线：数据、地址、控制
    - 仲裁总线：包括总线请求线和总线授权线
    - 中断和同步总线：用于处理带优先级的中断操作，包括中断请求线和中断响应线
    - 公用线：包括时钟线、电源线、地线、复位线及加电/断电的时序信号线等



总线仲裁（总线判优）

- 设备的主从状态
  - ✓ 连接到总线上的设备有主动和被动两种形态
  - ✓ 主设备持续占用总线的时间成为总线占用期
  - ✓ 主动方—对总线具有控制功能，可以启动一个总线周期，如CPU
    被动方—只能响应主动方的请求，如存储器
  - ✓ 每次总线操作，只能有一个主动方占用总线控制权，但可以同时有一个或多个被动方
- 总线仲裁
  - ✓ 对多个主设备提出的总线占用请求进行仲裁
  - ✓ 采用优先级或公平策略
  - ✓ 根据总线仲裁电路位置不同，分为集中式仲裁和分布式仲裁

集中式仲裁

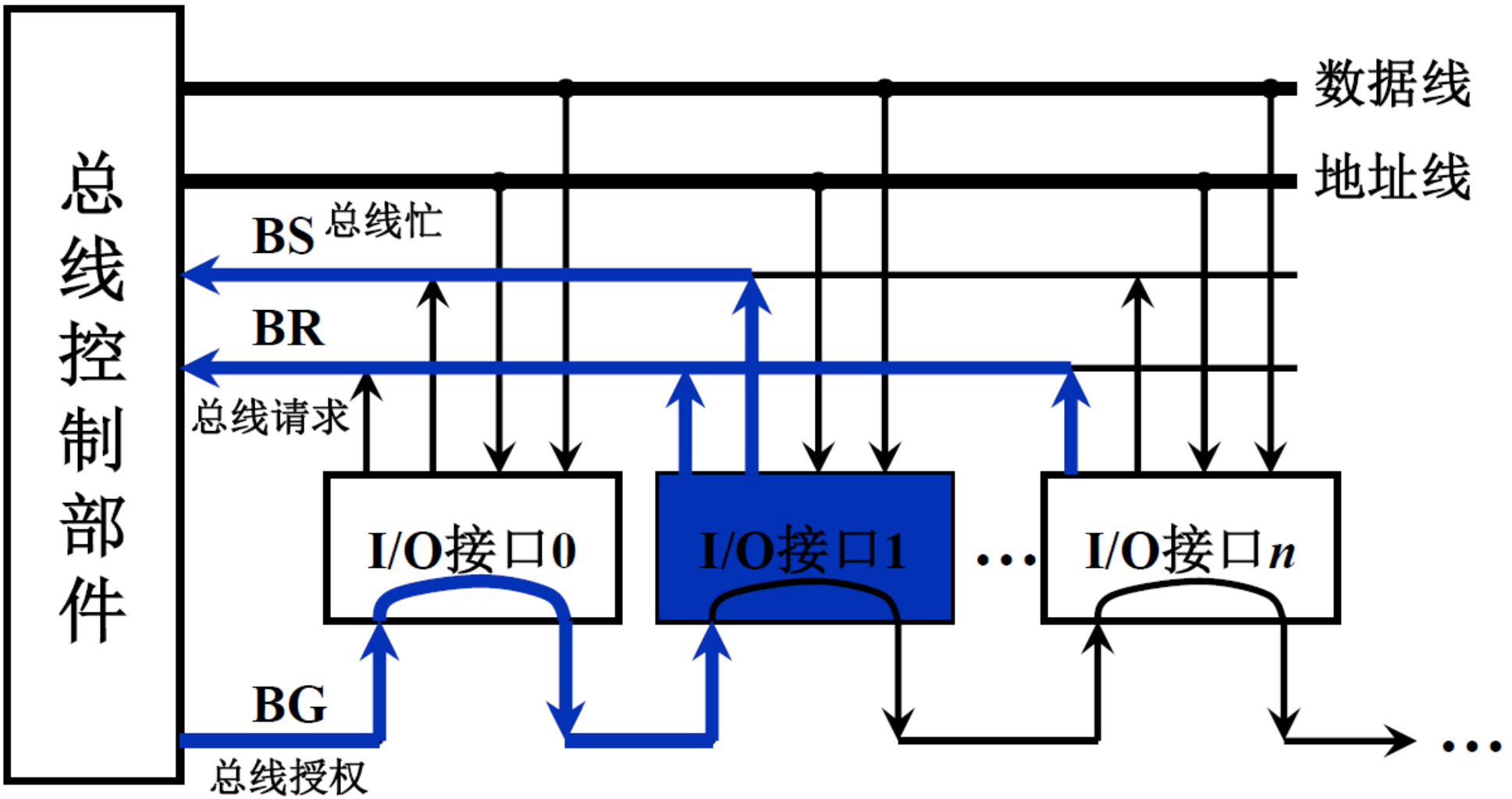
- 集中式仲裁
  - ✓控制逻辑集中在一处（如CPU中的总线仲裁器）
  - ✓每个设备模块有两条线连到总线仲裁器
    - 一条送往仲裁器的总线请求信号线BR
    - 一条仲裁器送出的总线授权信号线BG
  - ✓三种常见的集中式仲裁方式：
    - **链式查询**
    - **计数定时**
    - **独立请求**

链式查询方式

链式查询方式

- ✓总线授权信号线BG串行地从一个I/O接口传送到下一个I/O接口
  - 若BG到达的接口无总线请求，则继续往下查询；
  - 若BG到达的接口有总线请求，则不再往下查询，当前接口获得总线使用权，建立总线忙BS信号
- ✓优先级仲裁——离总线仲裁器最近的设备具有最高的优先级
- ✓优缺点
  - 优点：硬件连线简单，且易于扩充
  - 缺点：对电路故障敏感，优先级低的设备很难获得请求

链式查询方式

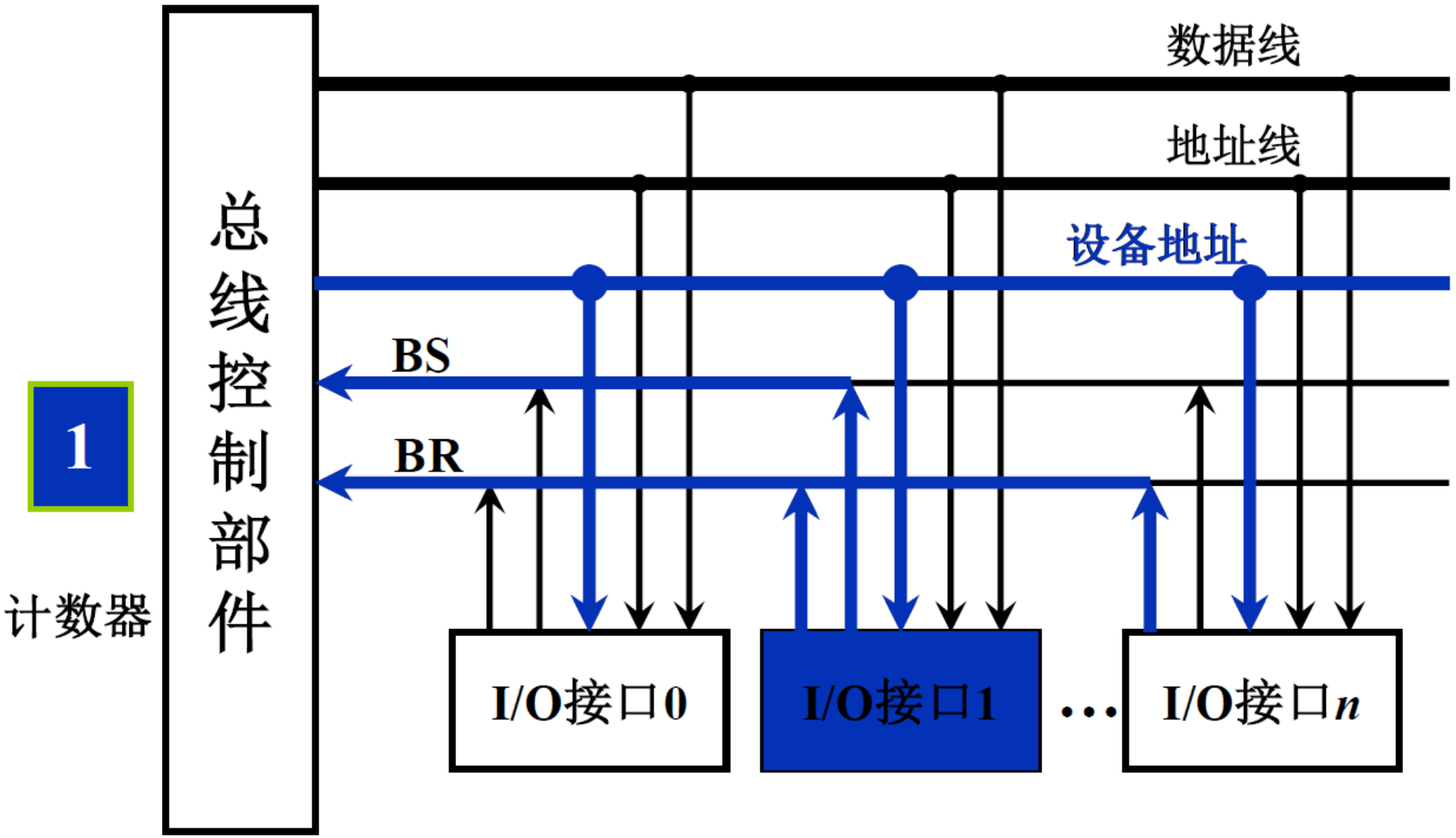


计数器定时查询方式

计数器定时查询方式

- ✓ 查询过程
  - 设备要使用总线时，通过BR线发出总线请求
  - 总线仲裁器接到请求信号后，在总线当前未被使用的情况下开始计数，并将计数值通过设备地址线发给各设备
  - 各设备接口将自身的设备地址与计数值进行比较，若一致，则该设备获得总线使用权，置BS线为“1”，此时中止计数查询
- ✓ 每次计数可以从“0”开始，也可以从上一次的中止值开始
  - 若从“0”开始，各设备的优先级顺序固定
  - 若从中止值开始，为一种循环方法，各设备的优先级相等
- ✓ 特点：
  - 计数器初始值可以由程序设置，因而设备优先级次序可以改变
  - 对电路故障不敏感，但增加了控制线数，控制较复杂

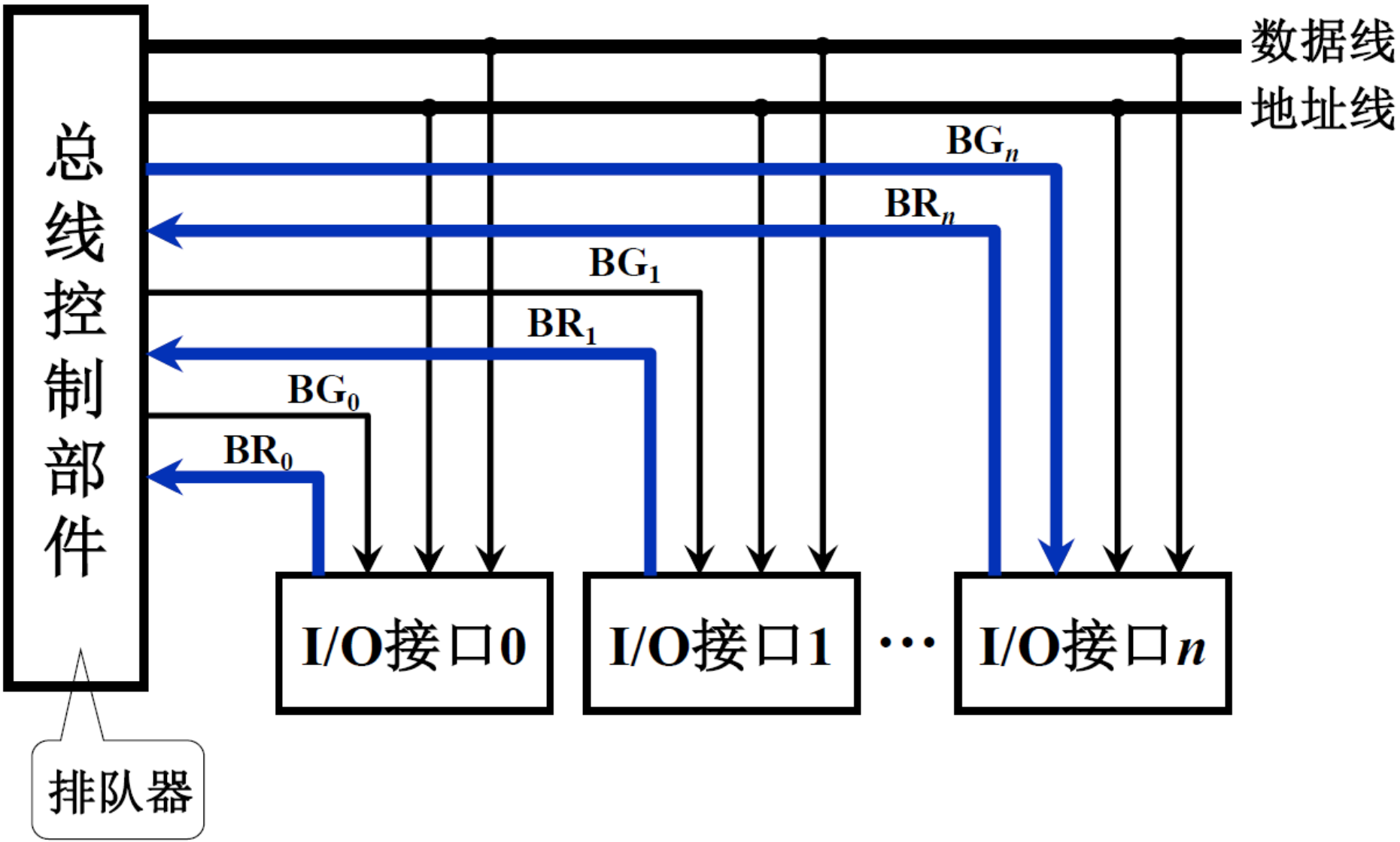
计数器定时查询方式



独立请求查询方式

- 独立请求查询方式
  - ✓每个设备都有一对总线请求线和总线授权线
    - 设备要使用总线时，发出该设备的请求信号
  - ✓总线仲裁器有一个排队电路，根据一定的优先次序决定首先响应哪个设备的请求
  - ✓优缺点
    - 优点——响应时间快，对优先次序的控制十分灵活
    - 缺点——控制线数量多，控制更复杂
  - ✓当代总线标准普遍采用的集中仲裁方式

独立请求查询方式



分布式仲裁

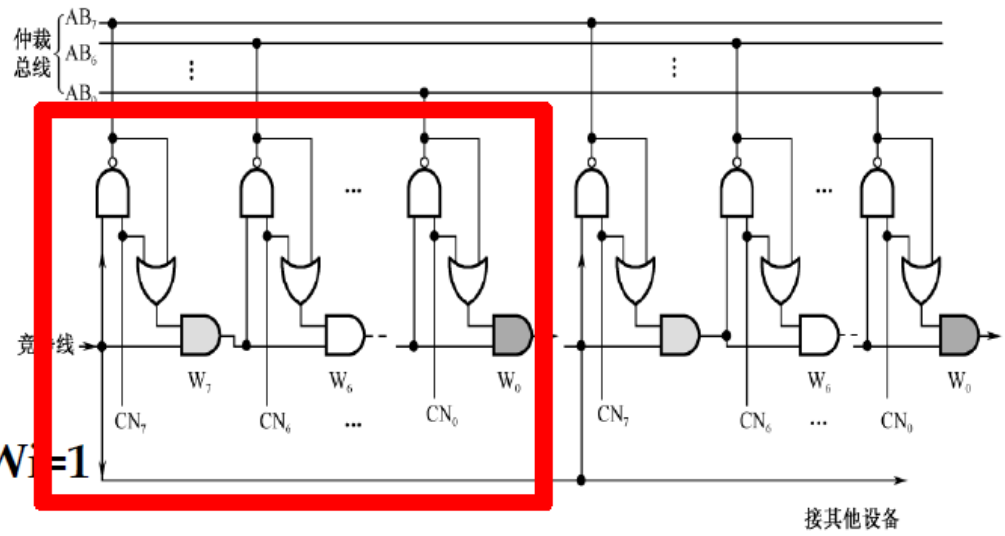
- 分布式仲裁不需要中央仲裁器，有三种常见的仲裁方式：
  - ✓自举分布式仲裁（每个设备独立地决定自己是否是最高优先级请求者。在总线裁决期间，每个设备将有关请求线上的信号合成后取回分析，根据这些请求信号确定自己能否拥有总线控制权）
  - ✓冲突检测分布式仲裁（每个设备独立地请求总线，多个同时使用总线的设备会发生冲突，冲突被检测到，按照某种策略在冲突的各方选择一个设备。（CSMA/CD带冲突检测的载波侦听多路访问）
  - ✓并行竞争分布式仲裁

并行竞争分布式仲裁

- 并行竞争分布式仲裁
  - ✓ 每个主设备具有专属的仲裁号和仲裁器
  - ✓ 第一个设备将自己的仲裁号写入仲裁总线
  - ✓ 仲裁过程
    - 当它们有总线请求时，把它们唯一的仲裁号发送到共享的仲裁总线上
    - 每个仲裁器将仲裁总线上得到的号与自己的号进行比较
    - 如果仲裁总线上的号大，则它的总线请求不予响应，并撤消它的仲裁号
    - 最后，获胜者的仲裁号保留在仲裁总线上。
  - ✓ 基于优先级策略的仲裁方式

Cni为设备的仲裁号，为1则请求竞争
Abi=0说明总线此位目前有请求

如Cni=0，Abi=0，则竞争失败，Wi=0
如Cni=0，Abi=1，则此位无竞争，Wi=1
如CNi=1，Abi=1，则竞争成功，Wi=1
如CNi=1，Abi=0，则继续竞争下一位，Wi=1



并行竞争分布式仲裁示例

- 两个设备同时要求使用总线，仲裁号分别是00000101和00001010；最终留在仲裁线上的号为00001010。

裁决号1		裁决号2		裁决线电平	裁决线逻辑
cn	AB	cn	AB		
0	高	0	高	高	0
0	高	0	高	高	0
0	高	0	高	高	0
0	高	0	高	高	0
0	高	1	低	低	1
1	高	0	高	高	0
0	高	1	低	低	1
1	高	0	高	高	0

总线操作与总线周期

- 读/写操作
  - ✓读操作：由从设备到主设备的数据传送
    地址-命令-数据
  - ✓写操作：由主设备到从设备的数据传送
    地址-数据-命令
- 块传送操作，猝发式传送（Burst）
  - ✓只需给出块起始地址，然后对固定块长度的数据一个接一个地读出或写入

总线操作与总线周期

- 写后读、读修改写操作
  - ✓两种组合操作：先写后读 / 先读后写
  - ✓只给出地址一次，读写为同一目标地址
  - ✓用途：
    - 先写后读一般用于校验目的；
    - 先读后写多用于多道程序系统中对共享存储资源的保护
- 广播/广集操作
  - ✓一个主设备对多个从设备的写操作，称为广播
  - ✓多个从设备对一个主设备的读操作，称为广集
    - 将选定的多个从设备的数据在总线上进行与/或操作
    - 可用于检测多个中断源

总线操作与总线周期

总线周期

- ✓通常指完成一次总线操作的时间
- ✓一般可以分为4个阶段
  - 申请分配阶段

    主设备提出总线使用申请，总线仲裁机构决定下一个传输周期的总线使用权归属
  - 寻址阶段

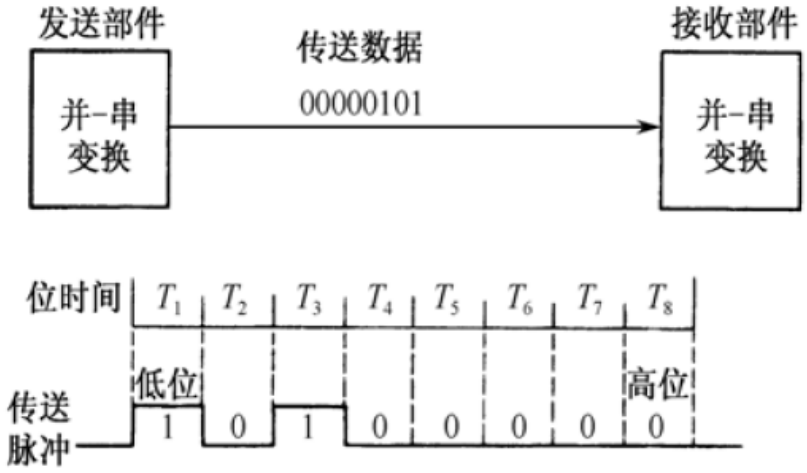
    获得总线使用权的主设备发送本次要访问的从设备的地址及有关命令，启动参与本次传送的从设备
  - 传送阶段

    主设备与从设备进行数据交换
  - 结束阶段

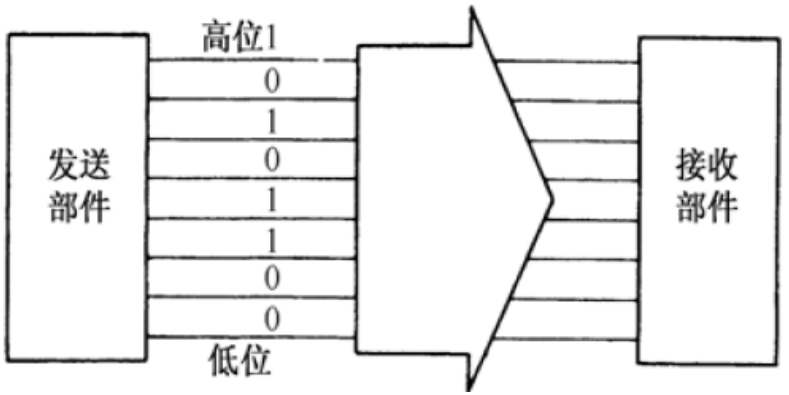
    主设备相关信息从总线上撤除，让出总线使用权

串行/并行传送与复用

- 传送方式
  - ✓ 串行传送
    - 只有一条传输线，采用脉冲传送
    - 位时间：每个二进制位在传输线上占用的时间长度，由同步脉冲体现
  - ✓ 并行传送
    - 使用多条传输线，同时传输多位二进制信息，采用电位传送
    - 串-并转换与并-串转换
- 分时复用
  - ✓ 总线复用，如既传数据，又传地址
  - ✓ 共享总线部件，分时使用



(a) 串行传送

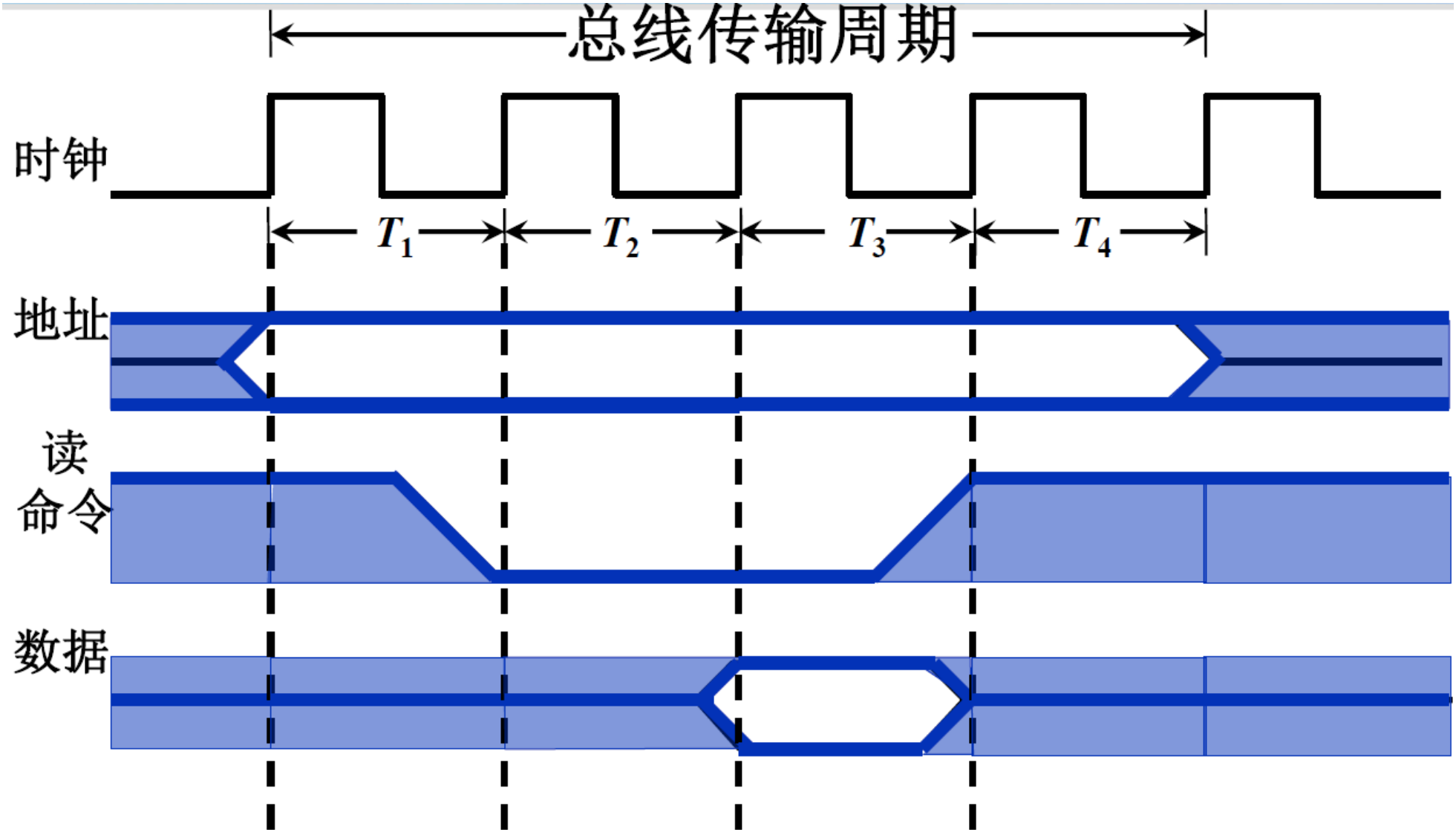


总线通信方式

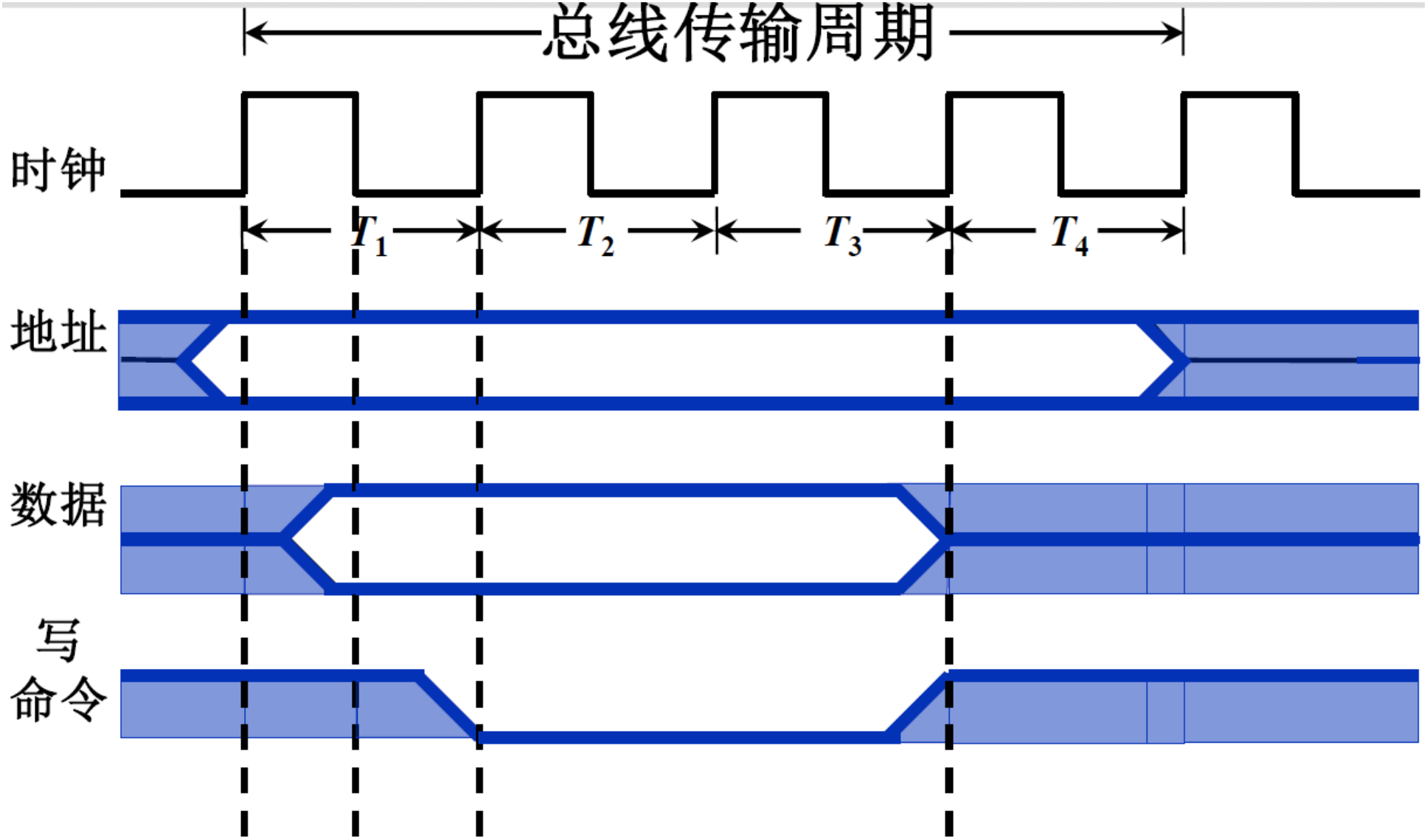
- 总线通信控制
  - ✓ 主要解决通信双方如何获知传输开始和传输结束，以及通信双方如何协调如何配合
  - ✓ 四种方式：同步通信、异步通信、半同步通信、分离式通信
- 同步通信
  - ✓ 通信双方由统一的时钟标准控制数据传送
  - ✓ 时钟标准的形成
    - 通常由CPU总线控制部件发出，发送给总线上的所有设备部件
    - 也可以由各个设备部件各自的时序发生器发出，但必须由总线控制部件发出的时钟信号对它们进行同步
  - ✓ 优点：规定明确、统一，模块间的配合简单一致

    缺点：1）强制同步，必须在限定的时间内完成规定操作；2）需按最慢速度部件来设计公共时钟，影响总线效率，缺乏灵活性
  - ✓ 一般用于总线长度较短、各部件存取时间较一致的场合

总线通信方式：同步数据输入

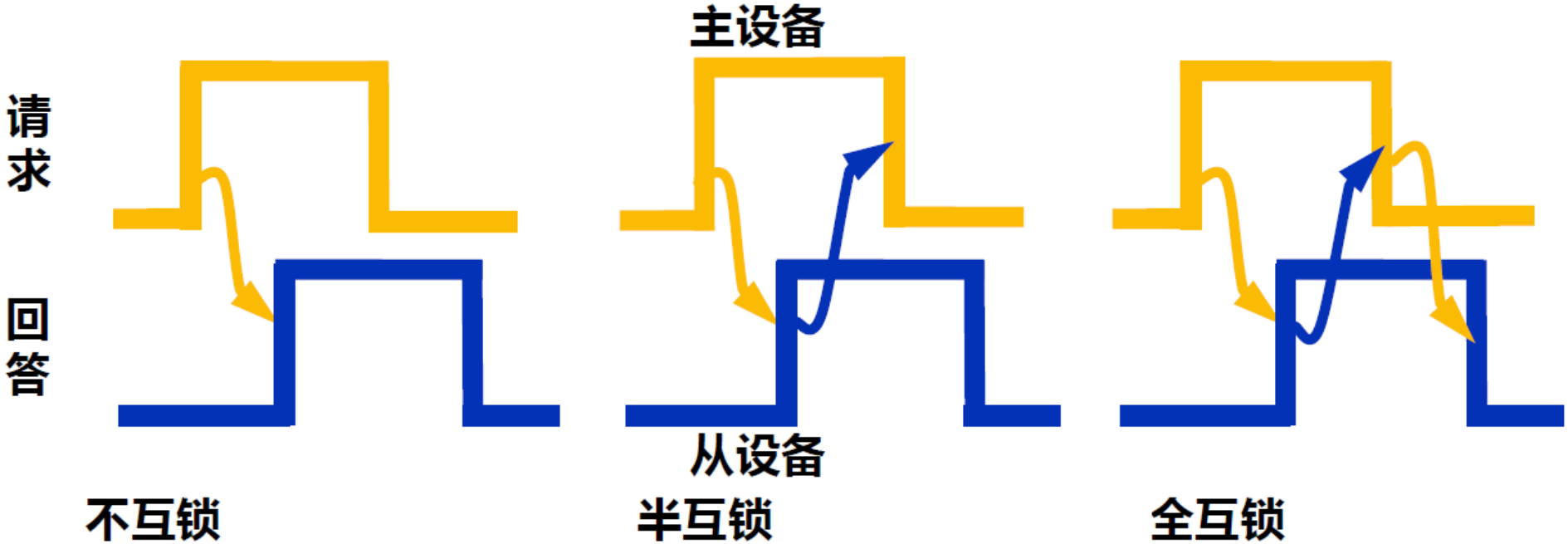


总线通信方式：同步数据输出



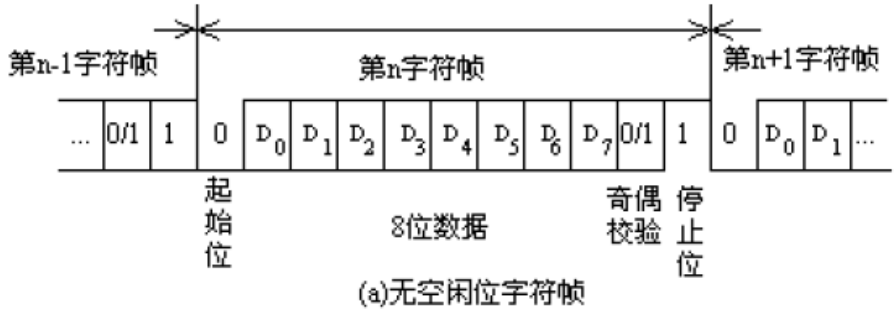
总线通信方式：异步通信

- 异步通信
  - ✓ 没有公共的时钟标准，不要求所有部件严格统一操作时间，允许各部件速度不一致
  - ✓ 采用应答方式（握手方式）
    - 需在主、从设备间增加两条应答线
  - ✓ 异步通信应答方式：不互锁（访存）、半互锁（访问共享存储器）和全互锁（网络通信）



不同数据传输率的异步串行通信

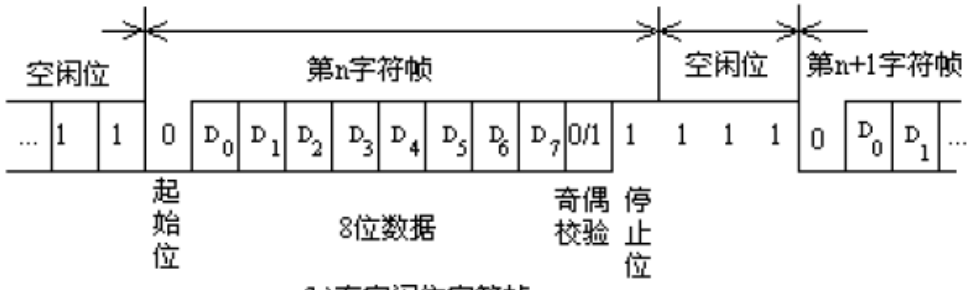
异步串行通信字符格式中包含起始位、终止位、校验位等若干附加位，若只考虑有效数据位，可用比特率来衡量数据传输速率



(a)无空闲位字符帧

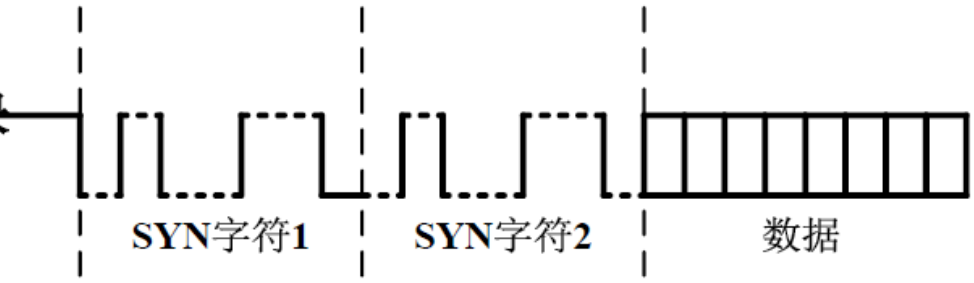
比特率——单位时间内传送的二进制有效数据的位数，单位为bps

波特率——单位时间内传送的二进制数据的位数，单位为bps



(b)有空闲位字符帧

为提高速度，可以去掉附加位，采用同步传送。同步传送中，数据块开始处要用同步字符SYN来指明。

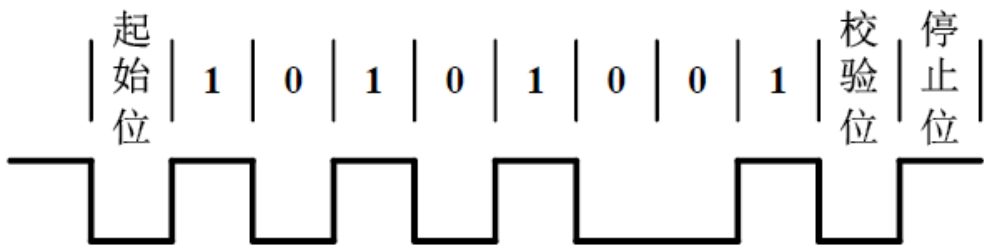


同步/异步传送例题

- 例 画图说明异步串行传送方式发送十六进制数据95H。要求字符格式为：1位起始位、8位数据位、1位偶校验位、1位终止位

  解：95H = 1001 0101B

  异步串行传送在起始位后传输数据位的最低位，数据位的最高位之后传输校验位，最后终止位。95H的偶校验位为0，波形图如下：



- **例 在异步串行传输系统中，字符格式为1位起始位、8位数据位、1位奇校验位和1位终止位。假设波特率为1200bps，求相应的比特率**

  解：根据题中的字符格式，有效数据位为8位，而传送一个字符需1+8+1+1=11位

  所以，比特率为

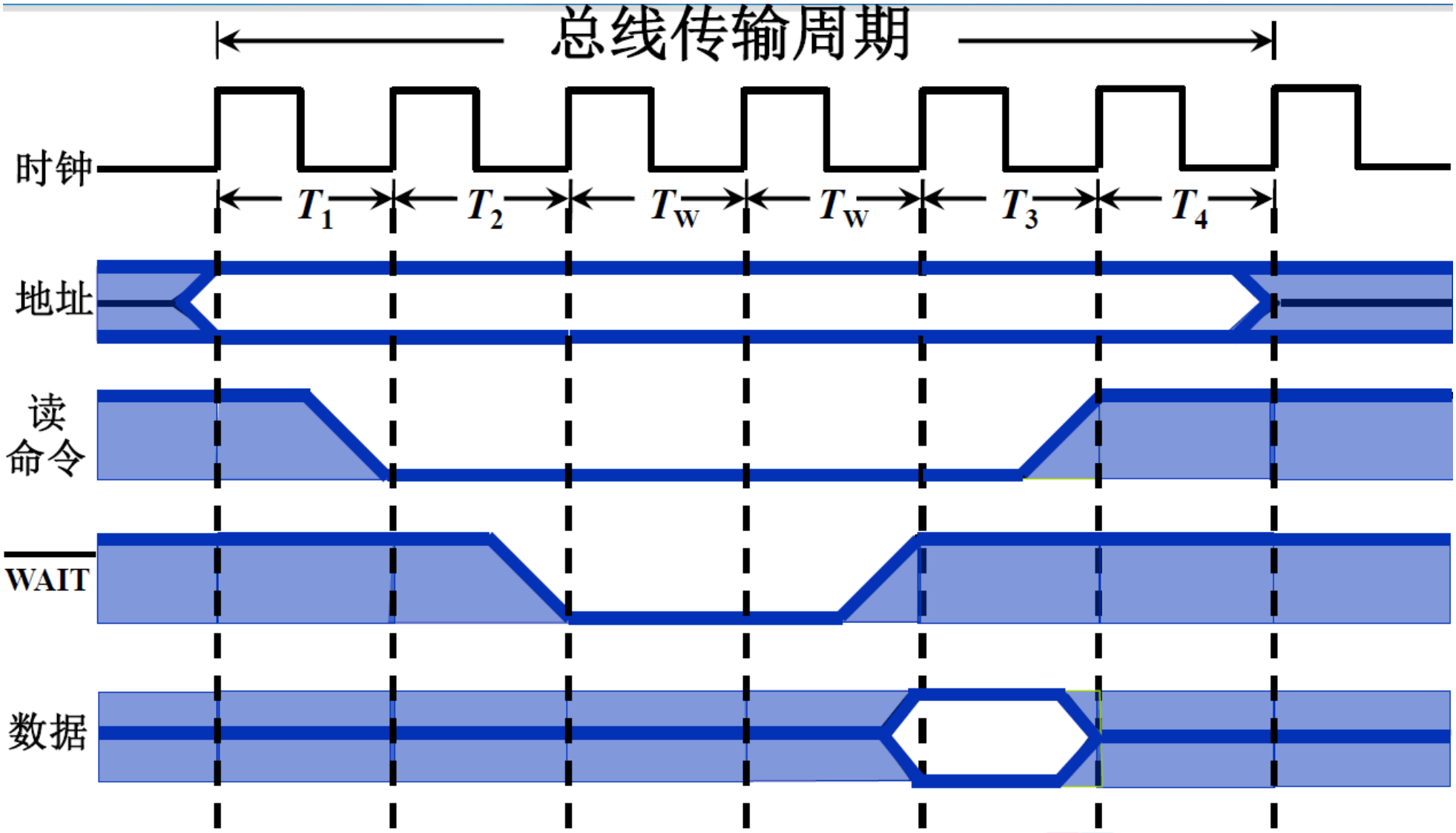
$$1200 \times (8 / 11) = 872.72\text{bps}$$

总线通信方式：半同步通信

- 半同步通信
  - ✓保留同步通信的基本特点
    - 所有地址、命令、数据信号的发出，都严格参照系统时钟沿开始
  - ✓结合异步通信方式，允许设备部件以不同速度工作
    - 增设一条“等待”响应信号线，采用插入时钟（等待）周期的措施来协调通信双方的配合问题
  - ✓优点：控制方式比异步通信简单；各模块由统一时钟控制同步工作，可靠性较高

    缺点：等待时间不确定导致工作效率低
  - ✓适用于工作速度差异较大的各类设备组成的简单系统

半同步通信数据输入过程



总线通信方式

- 上述三种通信方式的特点
  - ✓总线传输周期从主设备发出地址和读写命令开始，直到数据传输结束
  - ✓传输周期，系统总线由具有总线使用权的主设备和它选中的从设备占据
  - ✓总线传输周期时间主要花费在
    - 主设备通过总线向从设备发送地址和命令
    - 从设备按照命令准备数据
    - 从设备通过总线向主设备提供数据

总线通信方式

- 分离式通信方式
  - ✓ 充分挖掘系统总线的潜力，提高系统性能
  - ✓ 基本思想：将一个总线周期分为两个子周期
    - 第一个子周期，主设备获得总线使用权后向相关从设备发送地址和命令等信息，然后放弃总线使用权
    - 第二个子周期，从模块准备好数据，然后申请总线使用权，向相应的主设备发送要求的数据信息
- 分离式通信特点
  - ✓ 两个子周期都只有单方向的信息流，每个设备其实都成了主设备
  - ✓ 各个设备都有权申请总线使用权
  - ✓ 采用同步方式通信，不等对方回答
  - ✓ 各模块准备数据时，不占用总线
  - ✓ 总线被占用时都在有效工作，不存在空闲等待时间
  - ✓ 总线在多个主、从设备间交叉重叠并行式传送
- 控制比较复杂，在普通微型计算机系统中很少采用，多见于大型计算机系统

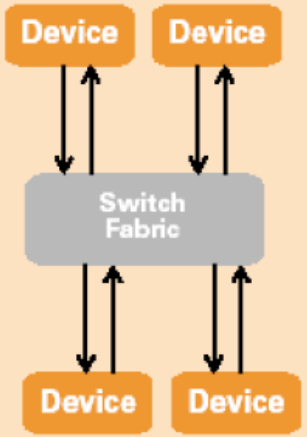
总线系统及其发展趋势

System Performance

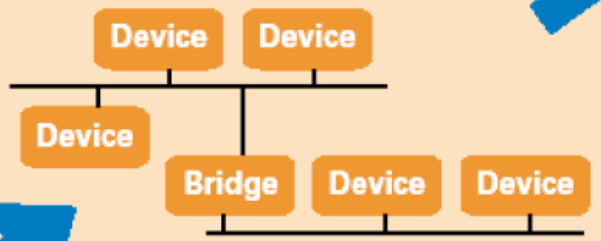
目前正在研发和已经使用的新型总线：

RapidIO：1Gbps ~ 60Gbps
Infiniband：100Gbps
3GIO： 2.5GB/s, Intel/AMD
FutureIO：10GB/s， IBM,Compaq,HP
Serial AGP， Serial ATA， Intel
ARM AMBA-AHB-APB-AXI
Intel QPI
IBM PLB OPB
PCI-E

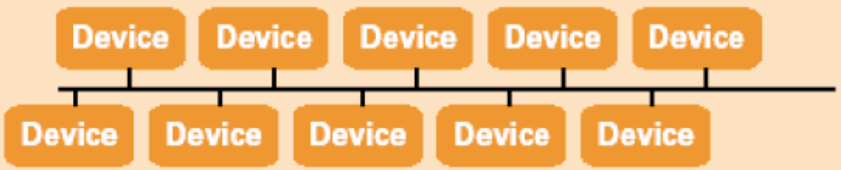
- Packet Switched
- Point-to-Point
- Low Pin Count



Very High Frequency



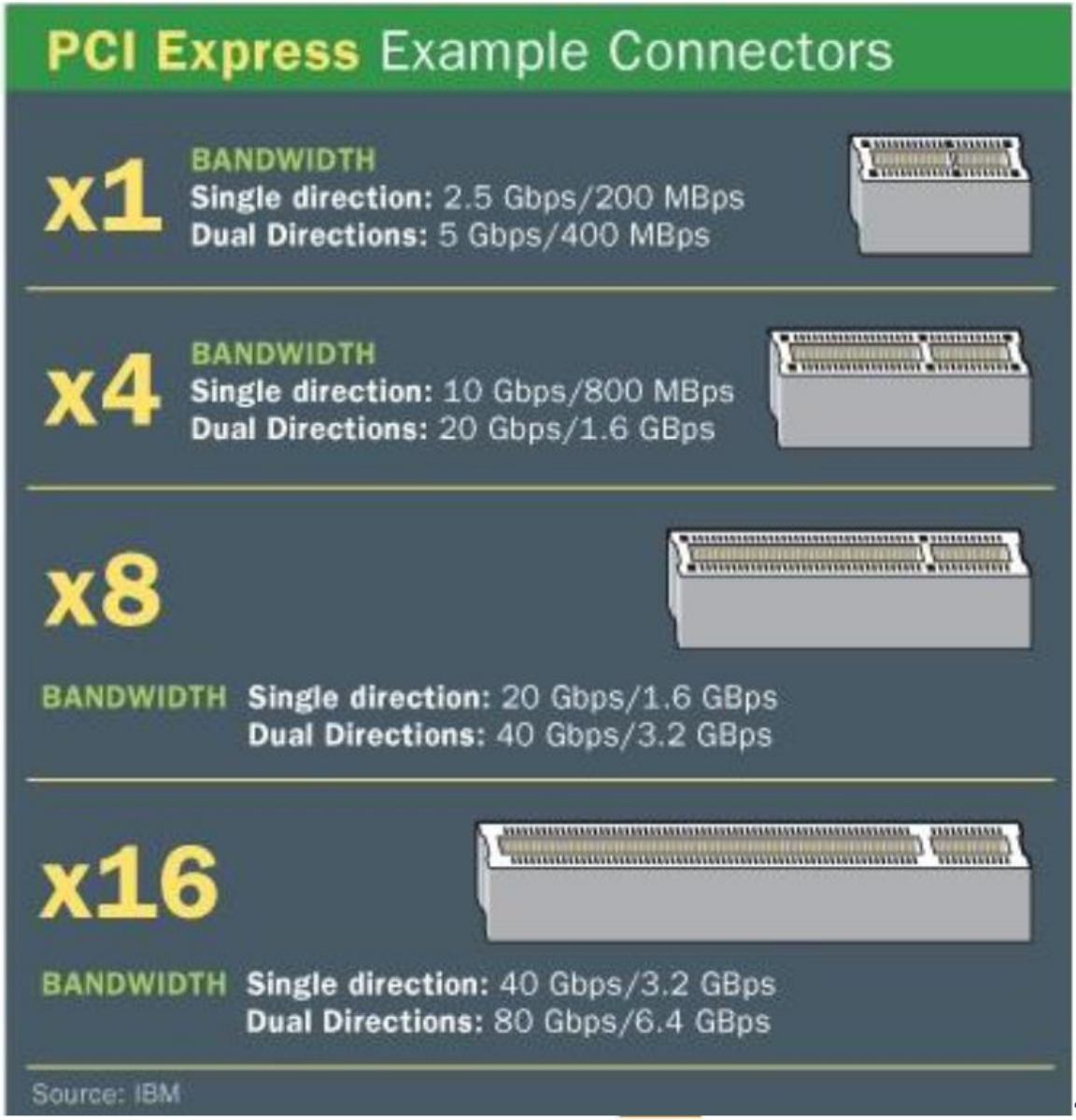
Bridged Hierarchy
Broadcast
< 133MHz
Examples: PCI, PCI-X



Single Segment
Broadcast
< 33MHz
Examples: CompactPCI, VME

PCIe

英文全名	Peripheral Component Interconnect Express
中文全名	快捷外设互联标准
发明日期	2004年
发明者	Intel

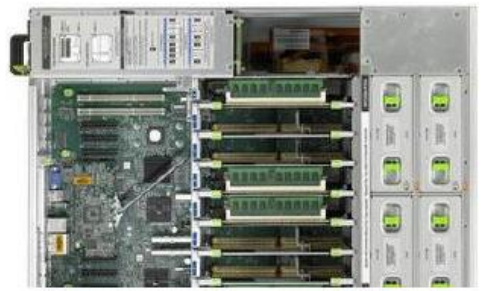


刀片服务器

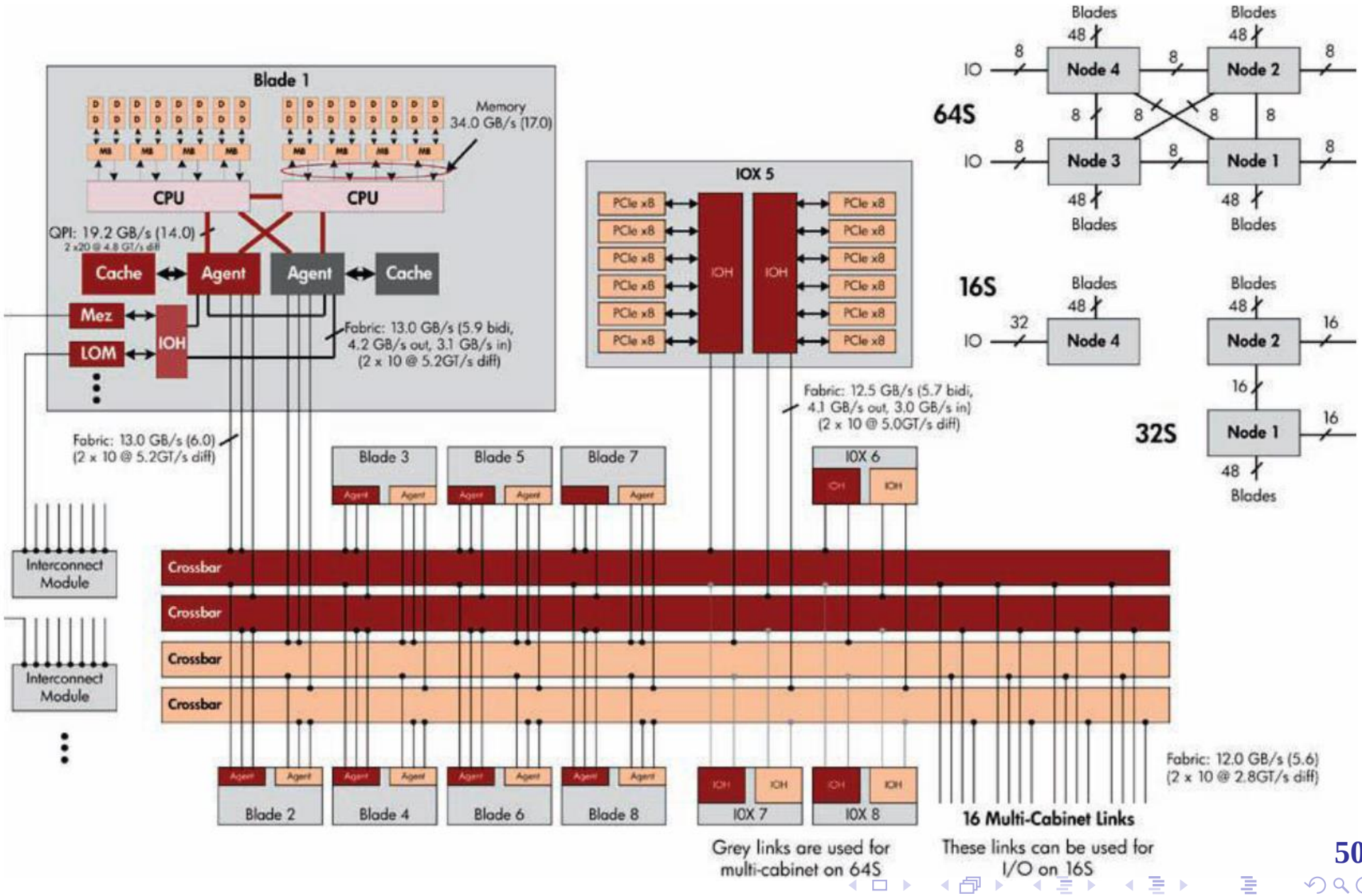
- 刀片服务器
  - ✓ 每一块“刀片”是一块系统主板，配置了CPU、内存、磁盘和网卡等设备
  - ✓ 每一个主板运行自己的系统，服务于指定的不同用户群
  - ✓ 可以通过刀片服务器中集成的交换“背板”形成星形连接网络。
  - ✓ 专门为特殊应用行业和**高密度**计算环境设计





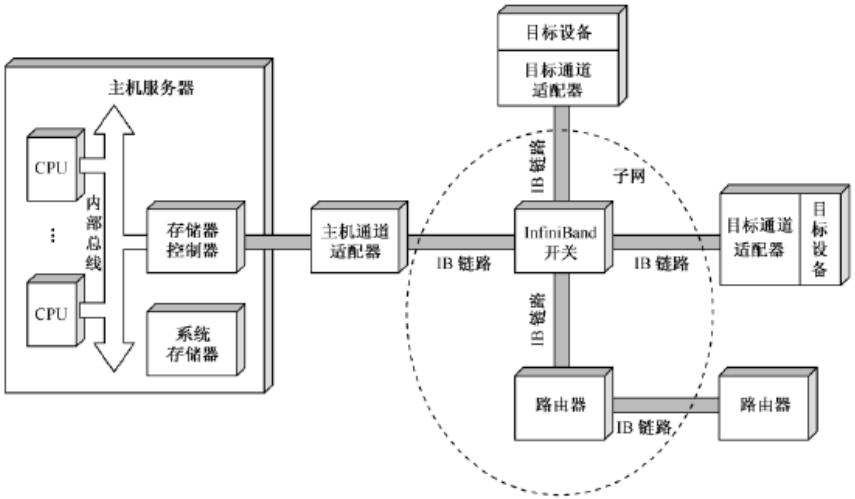


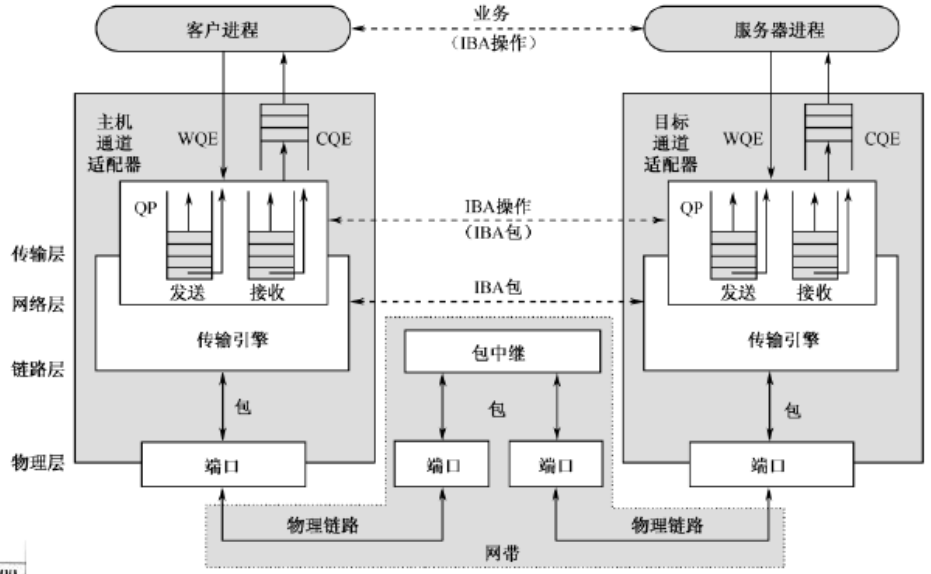

基于交叉开关Crossbar的互联



InfiniBand

- 基于开关和链路组成的中央开关网带，可连接多达64000个服务器、存储系统和网络设备

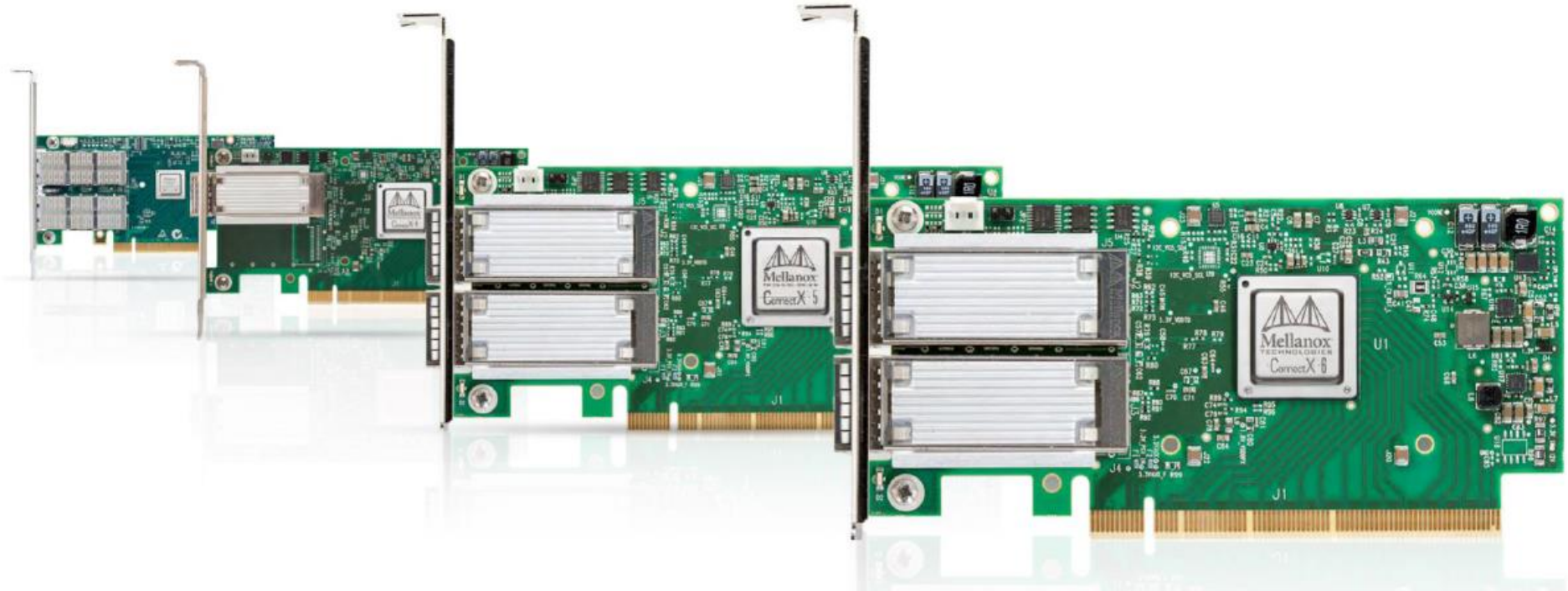




	型号	网卡和网线	1		¥153,500
1-1	网卡和网线	Mellanox 双口100Gb/S MCX556A	1	9,800	
		Mellanox 连接线IB EDR 100Gb/S,3Meter	1	1,700	
		Mellanox SWITCH-IB 36口交换机	1	142,000	
		总金额(TOTAL):	##########	元整	¥153,500

36个端口交换机带宽
100Gbps*36*2=7.2Tbps

InfiniBand


Mellanox
ConnectX-5
Mellanox
TECHNOLOGIES
ConnectX-6
U1
J1
J4
J5


LinkX

小结

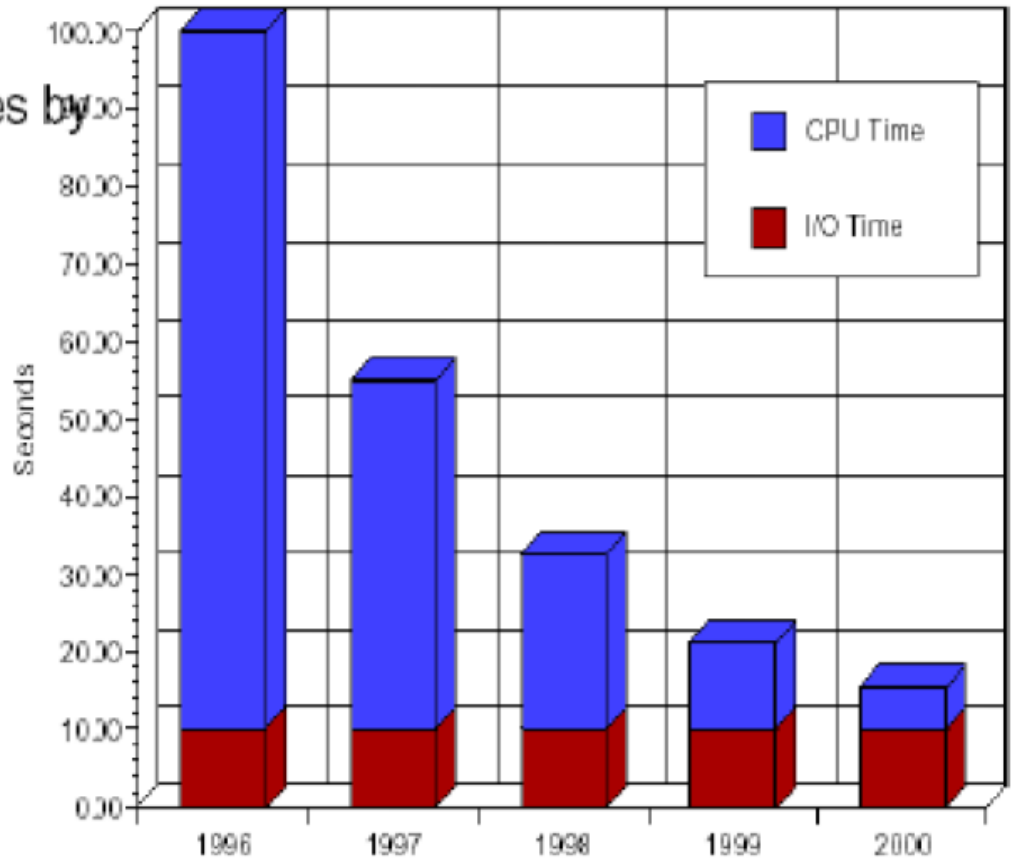
- 总线分类、特性与性能指标，拓扑结构
- 总线仲裁
- 总线通信
  - ✓传输过程
  - ✓控制过程
- 总线数据传输
  - ✓串并行方式，编码方式
  - ✓数据传输模式
- 总线周期、总线带宽

IO系统

1. I/O系统概述
   - 1.1 I/O系统的组成
   - 1.2 I/O系统的发展概况
   - 1.3 I/O系统与主机的联系
2. I/O接口
   - 2.1 I/O接口概述
   - 2.2 I/O接口的功能与组成
   - 2.3 I/O接口类型
3. I/O信息交换方式
   - 3.1 程序查询方式
   - 3.2 程序中断方式
   - 3.3 DMA方式
   - 3.4 通道方式
4. I/O设备
   - 4.1 IO设备概述
   - 4.2 常见的输入和输出设备

IO逐渐称为性能瓶颈

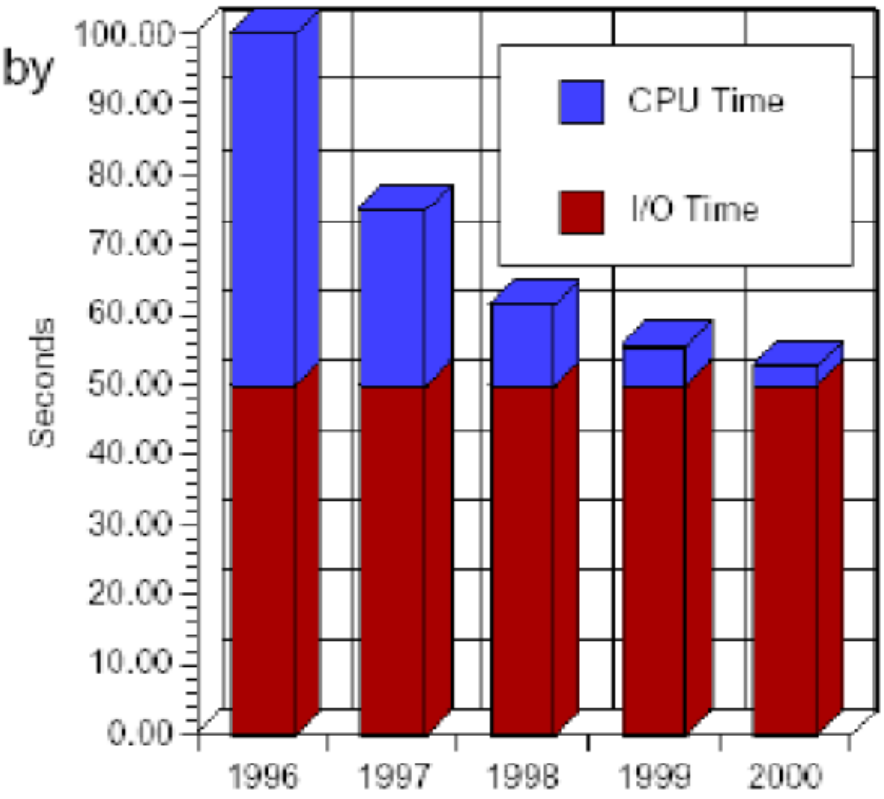
- 1996 - 1997
  - CPU performance improves by
    - N = 400/200 = 2
  - program performance improves by
    - N = 100/55 = 1.81
- 1997 - 1998
  - CPU performance - factor of 2
  - program performance
    - N = 55/32.5 = 1.7
- 1998 - 1999
  - CPU performance - factor of 2
  - program performance
    - N = 32.5 / 21.25 = 1.53
- 1999 - 2000
  - CPU Performance - factor of 2
  - program performance
    - N = 21.25 / 15.6 = 1.36



IO密集型应用

Performance for Web Surfing

- Assume 50 seconds CPU & 50 seconds I/O
- 1996 - 1997
  - CPU performance improves by
    - N = 400/200 = 2
  - program performance improves by
    - N = 100/75 = 1.33
- 1997 - 1998
  - CPU performance - factor of 2
  - program performance
    - N = 75/62.5= 1.2
- 1998 - 1999
  - CPU performance -f actor of 2
  - program performance
    - N = 62.5/56.5 = 1.11



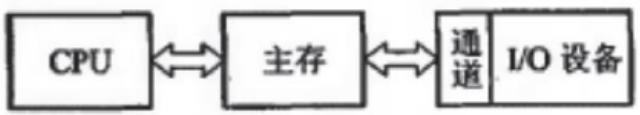
IO系统的组成

- I/O软件
  - ✓ 主要任务
    - 将用户程序或数据输入主机
    - 将运算结果输出给用户
    - 实现输入输出系统与主机工作的协调等
  - ✓ I/O指令
    - 机器指令的一类，反映CPU与I/O设备交换信息的各种特点（IN/OUT）
    - 一般格式

操作码	命令码	设备码

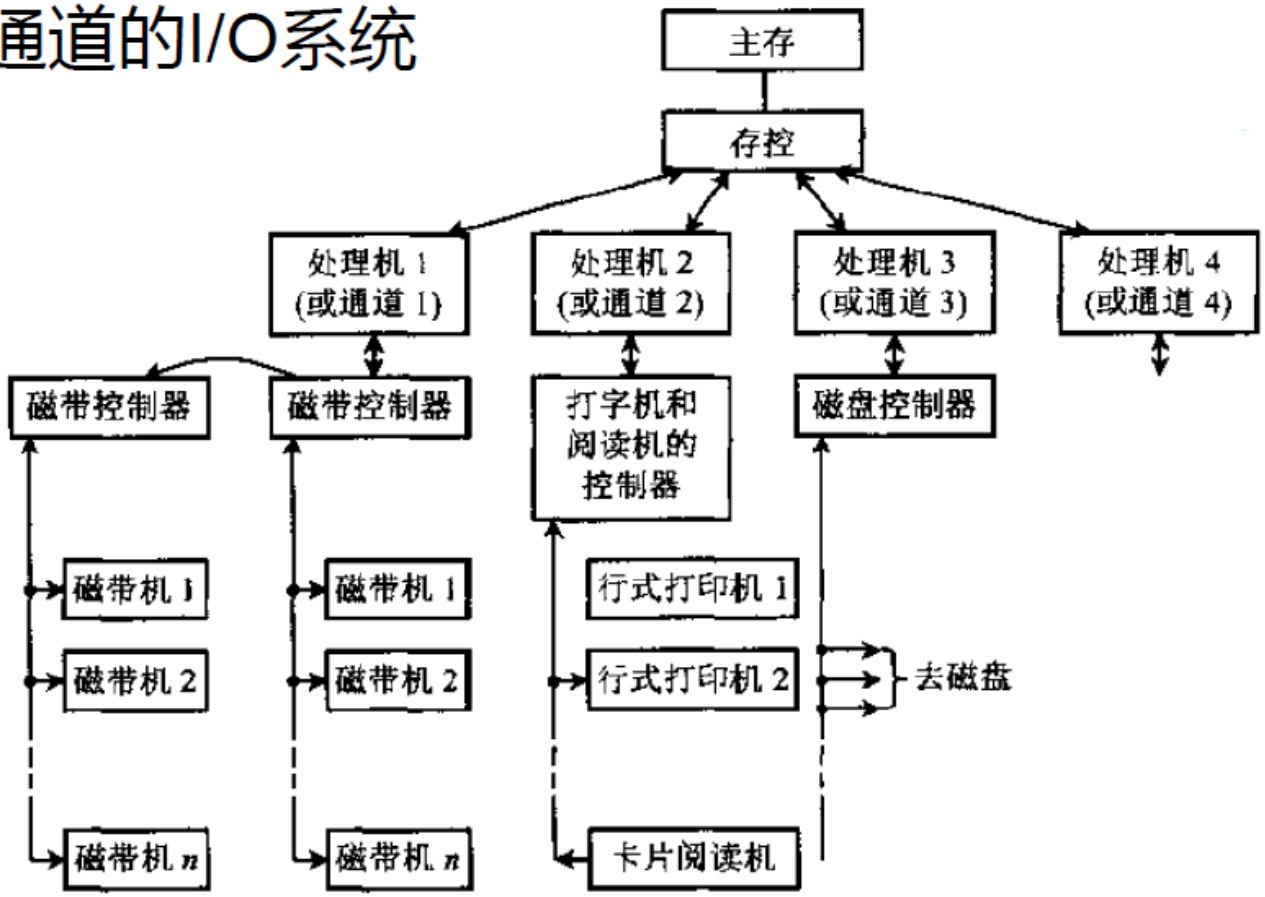
命令码：体现I/O设备的具体操作
设备码：相当于设备地址，用于在多台I/O设备中进行选择

  - ✓ 通道指令(通道控制字 Channel Control Word)
    - 对具有通道的I/O系统专门设置的指令，由通道执行，不属于CPU指令集
    - CPU执行了相应I/O指令后，将由通道指令来接管I/O设备的管理
    - 位数一般较长，一般用于指明数据首地址、传送字数及设备码、命令码等



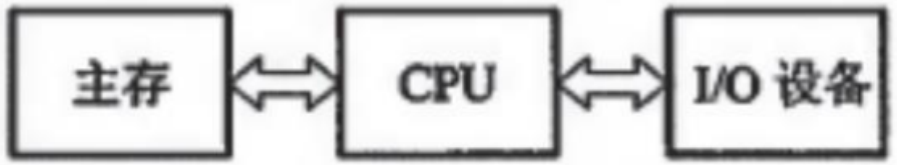
IO系统的组成

- I/O硬件（2）
  - ✓带接口的I/O系统，一般包括接口模块及I/O设备两部分
    - 接口电路一般包含数据传送通路、控制信号通路及相应逻辑电路
  - ✓具有通道的I/O系统

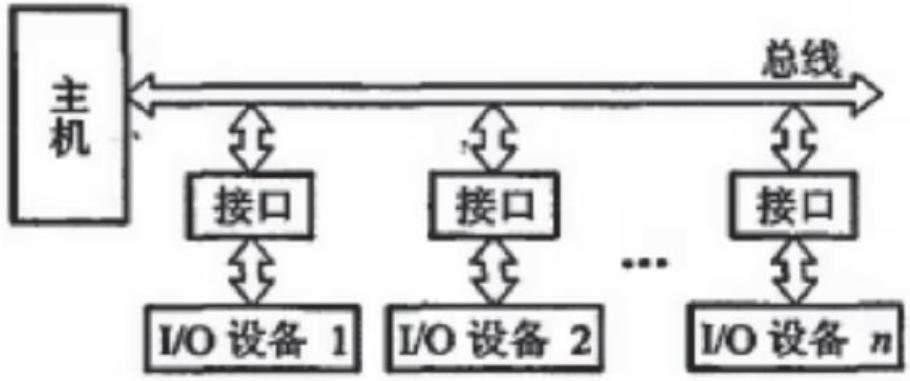


IO系统的发展概况

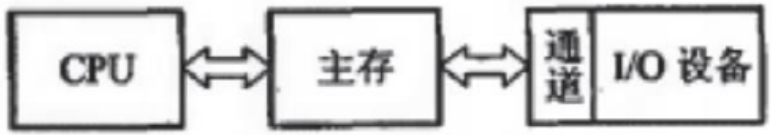
- 早期阶段
  - ✓ I/O设备种类少，与主存交换信息都必须通过CPU
  - ✓ 当时I/O设备的特点
    - 每个I/O设备都必须由一套独立的逻辑电路与CPU相连，分散连接
    - 输入输出过程穿插在CPU执行过程进行，CPU与I/O设备串行工作



- 接口模块和DMA（Direct Memory Access）阶段
  - ✓ I/O设备通过接口模块与主机连接，系统采用总线结构
  - ✓ 接口提供缓冲和数据转换功能，并支持中断请求处理，I/O设备与CPU可以并行工作
  - ✓ DMA：I/O设备与主存之间使用一条直接的数据通路



IO系统的发展概况



- 具有通道结构的阶段
  - ✓ 在I/O设备繁多、数据传送频繁的情况下，DMA方式同样带来硬件成本增加、控制复杂化和占用CPU时间等问题
  - ✓ 通道
    - 用来负责管理I/O设备以及实现主存与I/O设备之间交换信息的部件，可以认为是具有特殊功能的处理器
    - 专门的通道指令，构成独立地输入输出程序
    - 依赖CPU的I/O指令启动、停止或改变工作状态，是从属于CPU的专用处理部件
- 具有I/O处理机的阶段
  - ✓ 外围处理机，独立于主机工作
  - ✓ 可完成I/O通道要完成的I/O控制，以及**数据处理、转换、检错纠错**等操作
  - ✓ 与CPU工作的并行性更高

IO设备与主机的联系

I/O设备编址方式

✓统一编址

- 将I/O设备地址看做存储器地址的一部分

  如在64K地址的存储空间中，划出8K地址作为I/O设备地址，凡对这8K地址范围的访问，就是对I/O设备的访问

- I/O指令与访存指令类似

✓独立编址

- I/O地址和存储器地址分开，使用专门的I/O指令访问I/O设备

✓优缺点

- 统一编址：占用存储空间，减少主存容量，但无须专用I/O指令
- 独立编址：不占用主存容量，但需设置专用的I/O指令

IO设备与主机的联系

- 传送方式
  - ✓并行传送 V.S. 串行传送
- 联络方式
  - ✓用于I/O设备与主机相互了解彼此工作状态
  - ✓按I/O设备的工作速度，分为三种
    - 立即响应方式—对工作速度缓慢的I/O设备,无控制信号
    - 异步应答信号方式—I/O设备与CPU工作速度不匹配的情况
    - 同步联络方式—I/O设备与CPU工作速度完全同步

IO接口概述

I/O接口概述

- ✓ 接口：两个系统或两个部件之间的交接部分
  - 可以是两种硬设备之间的连接电路
  - 也可以是两个软件之间的共同逻辑边界
- ✓ I/O接口
  - 也叫**适配器**，指主机与I/O设备间设置的硬件电路及其相应的软件控制
- ✓ 为什么需要I/O接口
  - CPU可能连接多个不同设备号的I/O设备，可以通过接口实现设备的选择
  - 利用接口实现I/O设备与CPU的数据缓冲，减缓两者的速度差
  - 利用接口实现数据的串-并转换、电平转换
  - 通过接口传送控制命令
  - 通过接口监视设备工作状态并保存，供CPU查询使用

IO接口概述

21 文件传输服务器(控制连接)(FTP)
23 运程终端服务器(TELNET)
25 简单邮件传输服务器(SMTP)
80 万维网服务器(HTTP)

- 接口与端口
  - ✓端口：指接口电路中的一些**寄存器**
    - 这些寄存器用于存放数据、控制命令、状态信息等
  - ✓接口与端口的关系
    - **若干端口加上相应的控制逻辑组成接口**
  - ✓CPU对I/O接口（或I/O设备）的信息读写，实际上都是对端口的操作

IO接口的功能与组成

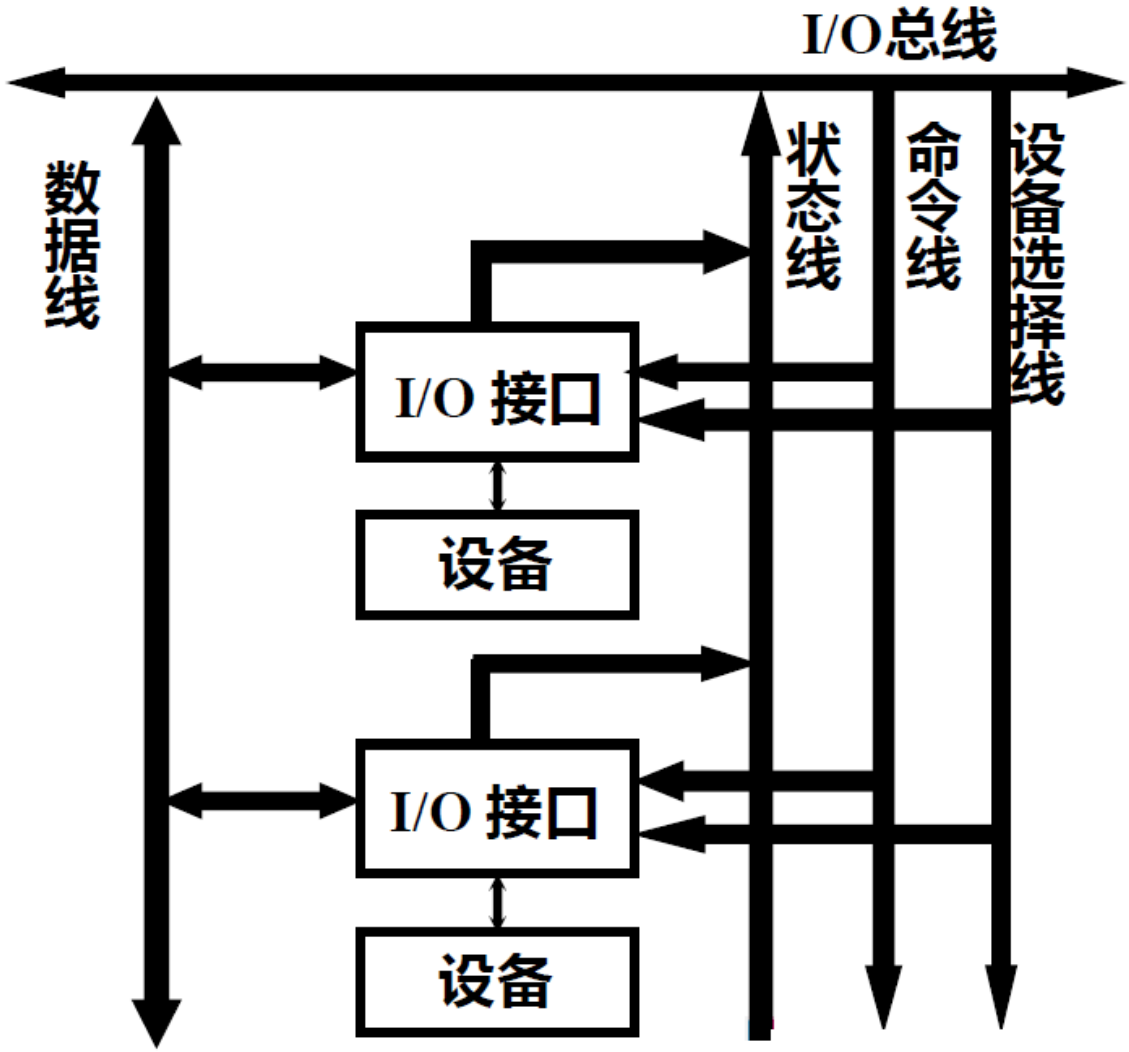
- 总线连接方式的I/O接口电路

(1) 设备选择线

(2) 数据线

(3) 命令线

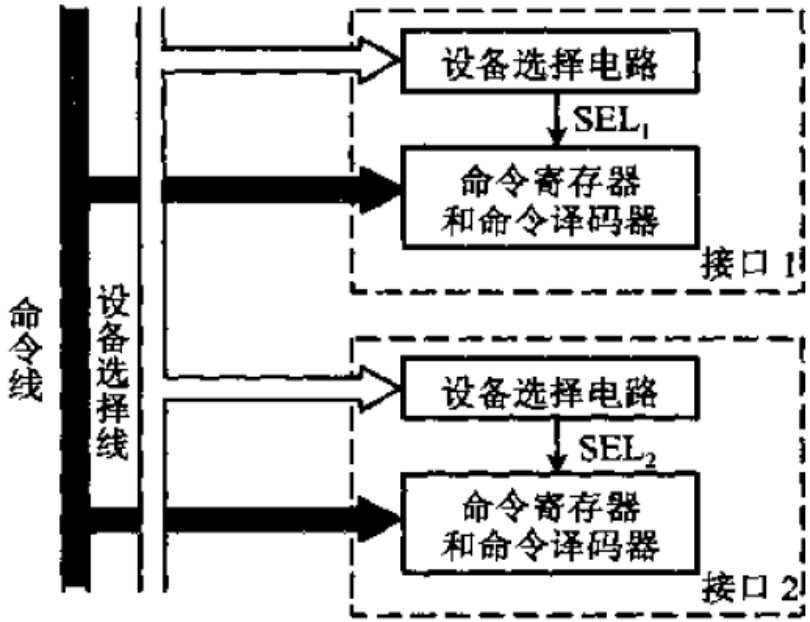
(4) 状态线



IO接口的功能与组成

- I/O接口的功能
  - ✓选址功能
    - 利用接口的设备选择电路实现
  - ✓传送命令的功能
    - 存放命令的命令寄存器、命令译码器等
  - ✓传送数据的功能
    - 设置数据缓冲寄存器，与数据线相连
  - ✓反映I/O设备状态的功能
    - 设置相关的状态触发器

      如中断请求触发器、中断屏蔽触发器
      工作标志触发器等
    - 完成触发器D，工作触发器B



IO接口的功能与组成

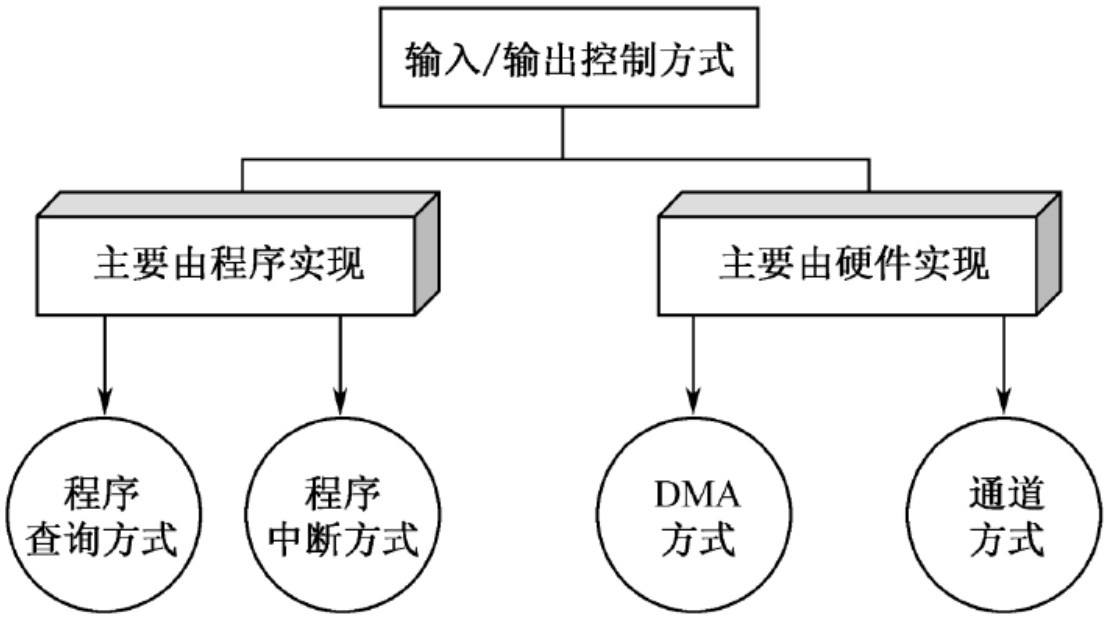
I/O接口的基本组成

IO接口类型

- 按数据传送方式分类
  - ✓ 并行接口：如Intel 8255
  - ✓ 串行接口：如Intel 8251
- 按功能选择的灵活性分类
  - ✓ 可编程接口：接口功能及操作方式可通过程序来改变或选择，如Intel 8255/8251
  - ✓ 不可编程接口：只能通过硬连线逻辑来实现不同功能，如Intel 8212
- 按通用性分类
  - ✓ 通用接口：可以供多种I/O设备使用
  - ✓ 专用接口：专门为某类外设或某种用途而设计
- 按数据传送的控制方式分类
  - ✓ 程序型接口：用于连接速度较慢的I/O设备，采用程序中断方式
  - ✓ DMA接口：用于连接高速I/O设备

信息交换方式

- 信息交换方式
  - ✓程序查询方式、程序中断方式、DMA方式、通道方式

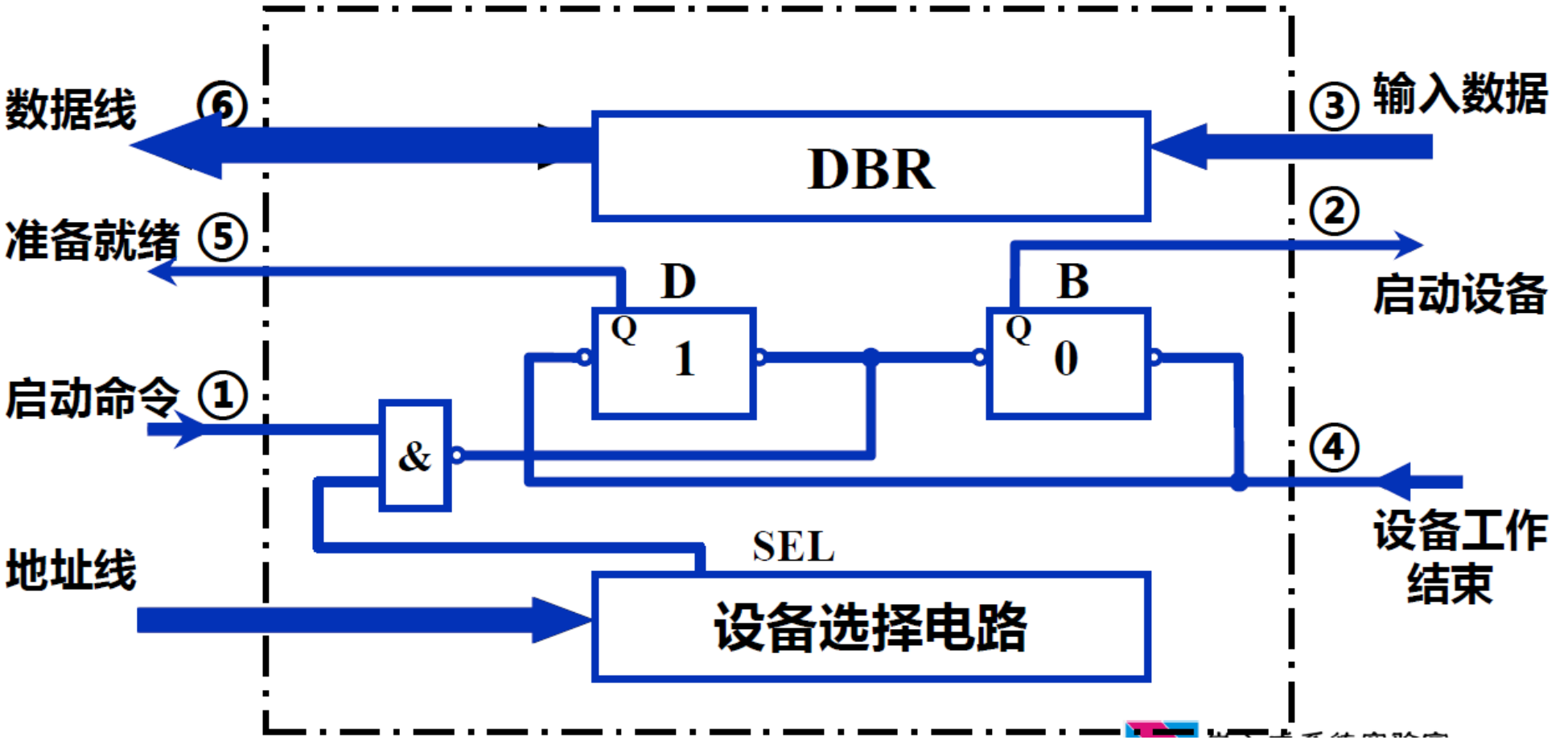


程序查询方式

- 程序查询方式的特点
  - ✓ 最简单的输入输出方式
  - ✓ 数据在CPU和外围设备之间的传送完全靠计算机程序控制
    - 在CPU的主动控制下进行
    - 有I/O操作时，CPU暂停主程序，转去执行设备I/O的服务程序
  - ✓ 优缺点
    - 优点：CPU与外设的操作能够同步，硬件结构简单
    - 缺点：CPU循环查询，浪费CPU周期和资源
  - ✓ 一般用于单片机或数字信号处理DSP中
- 查询通过做I/O指令完成
  - ✓ 对I/O接口的某些控制寄存器置“0”或“1”，用于控制设备进行相关工作
  - ✓ 测试设备的某些状态，如“忙”、“就绪”等，以便决定下一步操作
  - ✓ 传送数据

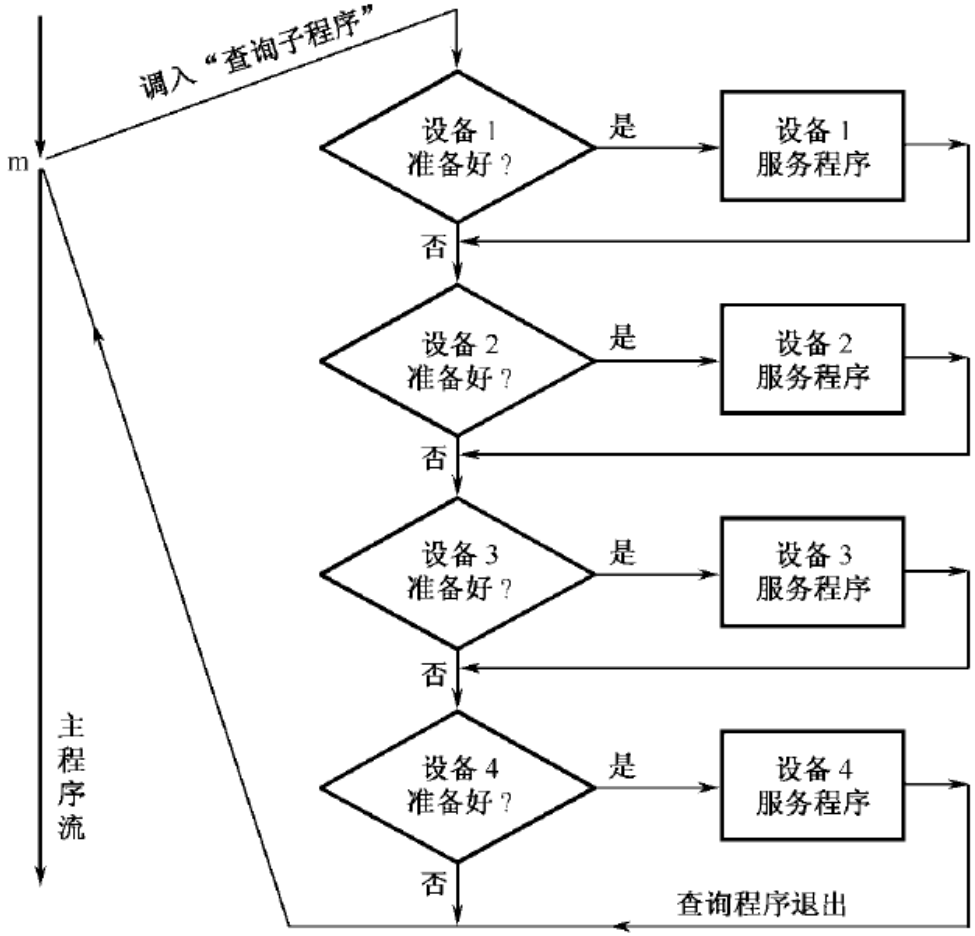
程序查询方式

- 程序查询方式的接口
  - ✓设备选择电路、数据缓冲寄存器、设备状态标志



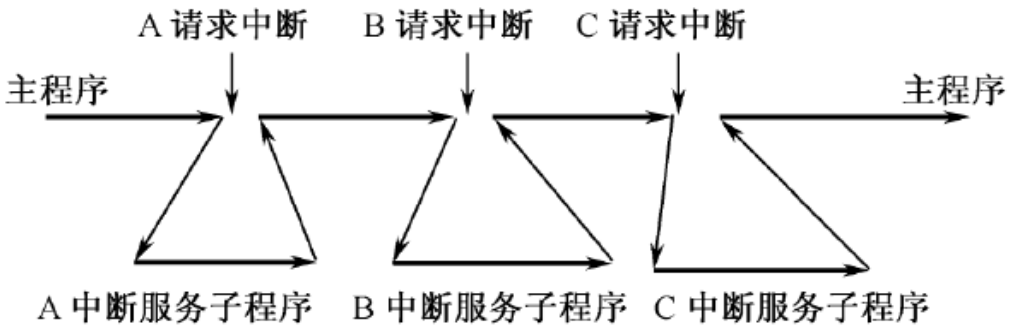
程序查询方式

程序查询方式的过程



程序中断方式

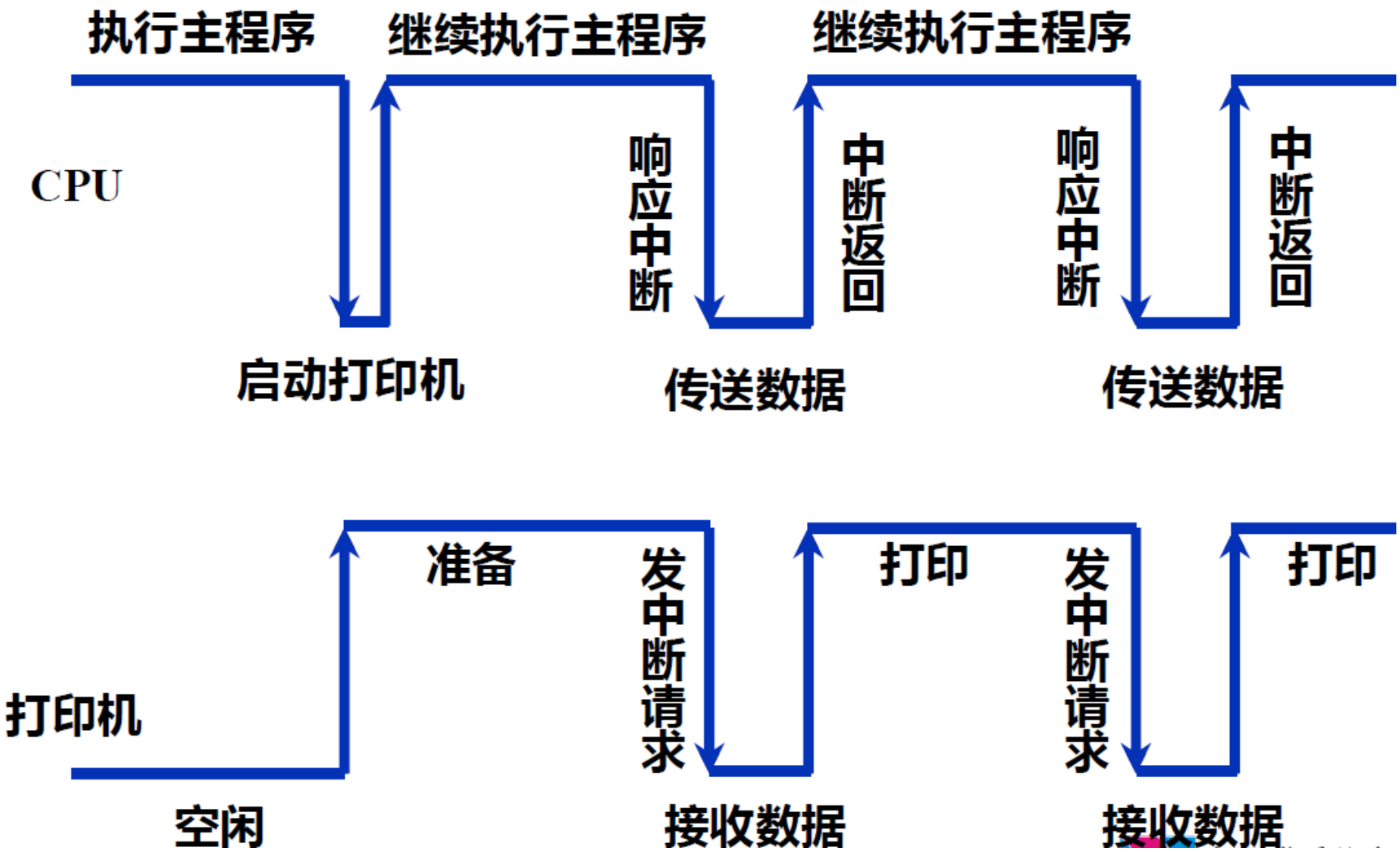
- 中断的概念
  - ✓ 指CPU暂时中止现行程序，转去处理随机发生的紧急事件，处理完后自动返回原程序的功能和技术
  - ✓ 原理与调用子程序类似，不过中断请求是由外设发出的



- 中断系统
  - ✓ 计算机实现中断功能的软硬件总称
  - ✓ 一般在CPU中设置中断机构，在外设接口中设置中断控制器，在软件上设置相应的中断服务程序

IO中断示例

以打印机为例　　CPU 与打印机并行工作



DMA方式

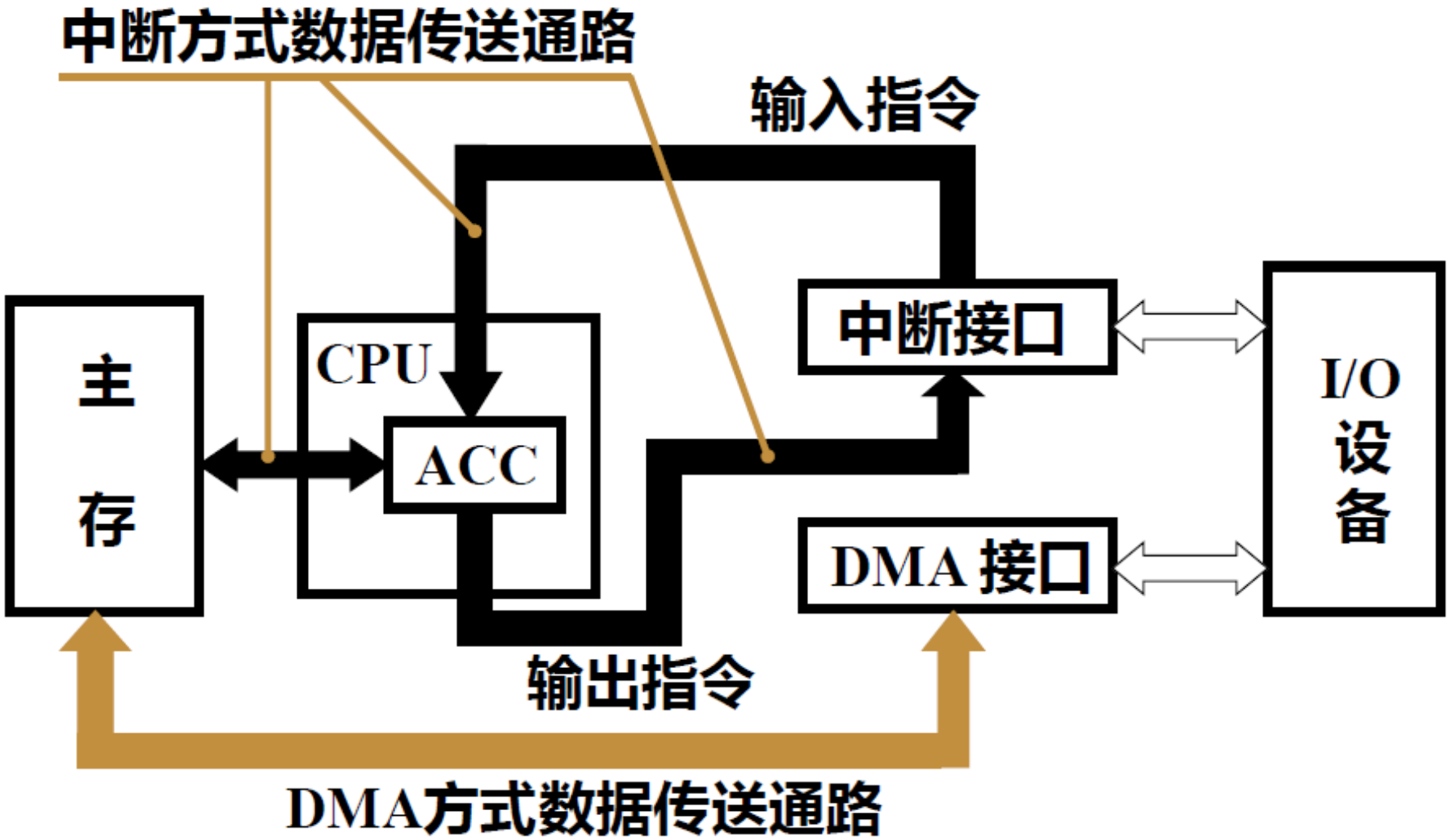
DMA

- ✓直接内存访问，一种完全由硬件执行I/O交换的工作方式
- ✓主存与I/O设备间高速交换批量数据，传送速度快
- ✓基本思想
  - 硬件DMA控制器从CPU完全接管对总线的控制，数据交换不经过CPU，直接在主存和I/O设备之间进行

工作过程描述

- 在主存中要开辟连续地址的专用缓冲器，用来提供或接收传送的数据。在数据传送前和结束后CPU要通过程序或中断方式对缓冲器和DMA控制器进行预处理和后处理。
- 由DMA控制器给出当前正在传送的数据的主存地址，并统计传送数据的个数以确定一组数据的传送是否已结束。

DMA和程序中断两种方式的数据通路



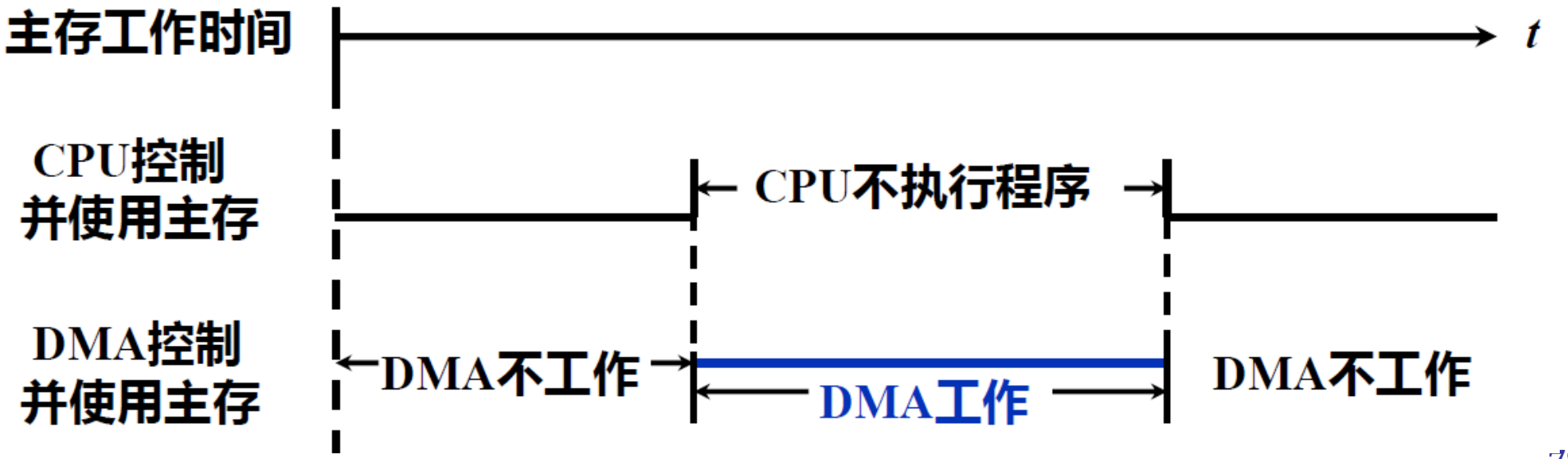
DMA方式

- DMA控制器
  - ✓通过大规模集成电路硬件实现
- DMA的基本操作
  - ✓从外设发出DMA请求
  - ✓CPU响应请求，DMA控制器接管总线控制
  - ✓DMA控制器对内存寻址，决定数据传送的内存单元地址及数据传送个数的计数，并执行数据传送操作
  - ✓向CPU报告DMA操作结束

在DMA方式中，数据传送前后的准备和处理工作，由CPU上的管理程序负责，DMA控制器仅负责数据传送工作

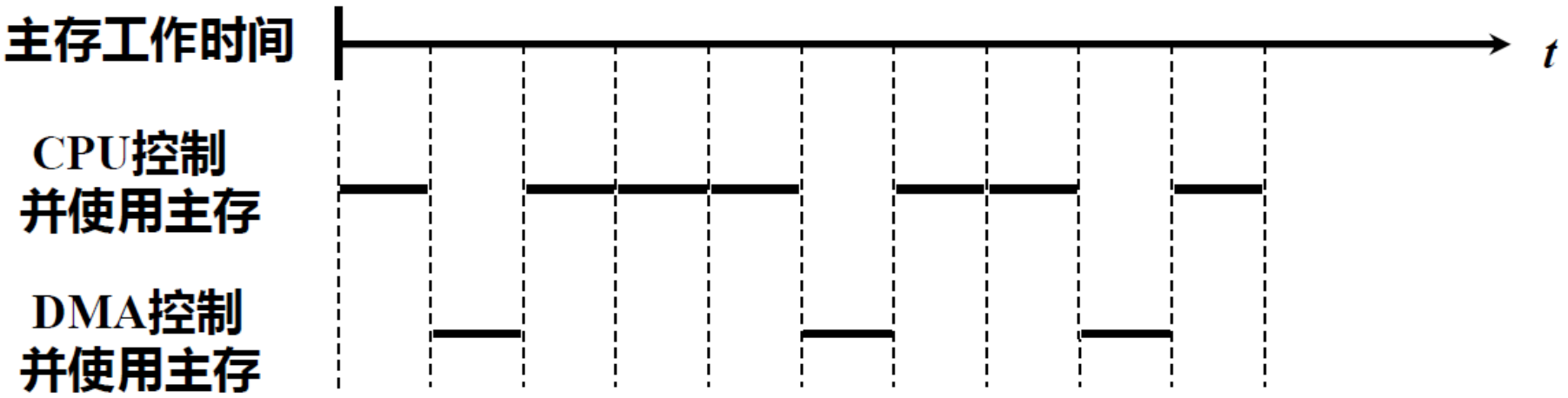
DMA与主存交换数据的三种方式

- 停止CPU访问内存
  - ✓CPU处于不工作状态或保持状态，未能发挥CPU对主存的利用率
  - ✓控制简单，适用于数据传输率很高的设备进行成组传输



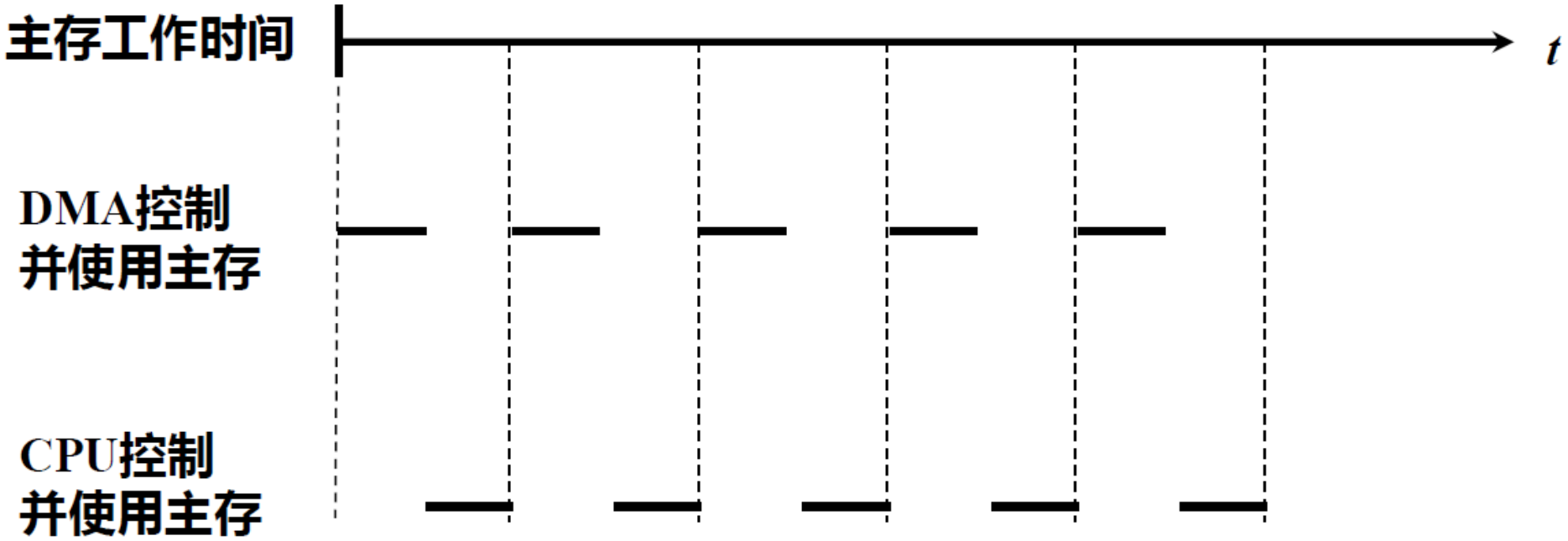
DMA与主存交换数据的三种方式

- 周期挪用（周期窃取）
  - ✓ DMA控制器与主存间传送一个数据时，占用（窃取）一个或多个CPU周期。即CPU暂停工作一个周期，然后继续执行程序
  - ✓ I/O设备通过DMA访问主存有三种可能情况
    - CPU此时不访存——无冲突（执行复杂ALU指令）
    - CPU正在访存——待访问结束后，让出总线
    - CPU和DMA同时请求访存——DMA优先
  - ✓ 较常采用的方法：异步、需申请总线访问，比较适合于I/O读写周期大于主存周期的情况



DMA与主存交换数据的三种方式

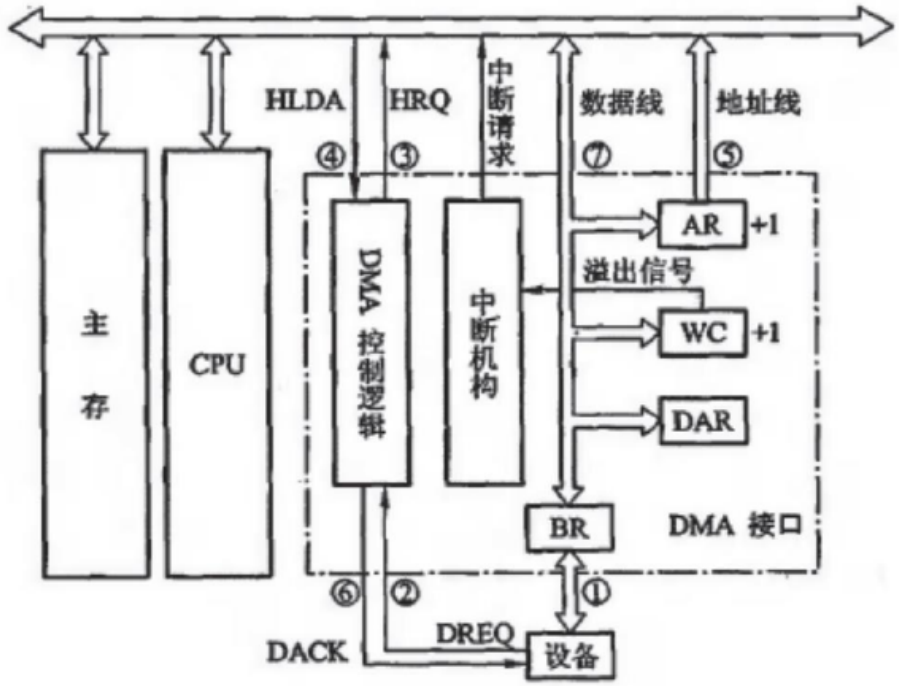
- DMA与CPU交替访存
  - ✓ CPU周期分为两部分，一部分专用于CPU访存，另一部分专用于DMA访存
  - ✓ 不需要申请和归还总线使用权，总线控制权的转移速度快，DMA效率高，控制复杂



DMA方式

- DMA控制器基本组成
  - ✓ 内存地址寄存器(计数器)AR：用于存放内存中要交换的数据的地址，DMA传送时，每交换一次数据，寄存器值加“1”
  - ✓ 字计数器WC：用于记录传送数据块长度
  - ✓ 数据缓冲寄存器BR：用于暂存每次传送的数据
  - ✓ 设备地址寄存器DAR： IO设备码或辅存寻址信息
  - ✓ 控制/状态逻辑：负责管理DMA传送过程，用于修改相关信息和状态

中断机构：字计数器溢出时，表示一组数据传送完毕，触发中断



DMA方式

- DMA数据传送过程
  - ✓ 分为三个阶段：预处理、数据传送、后处理
  - ✓ 预处理阶段
    - CPU通过I/O指令给DMA控制器预置初值，取状态和设置传送需要的有关参数

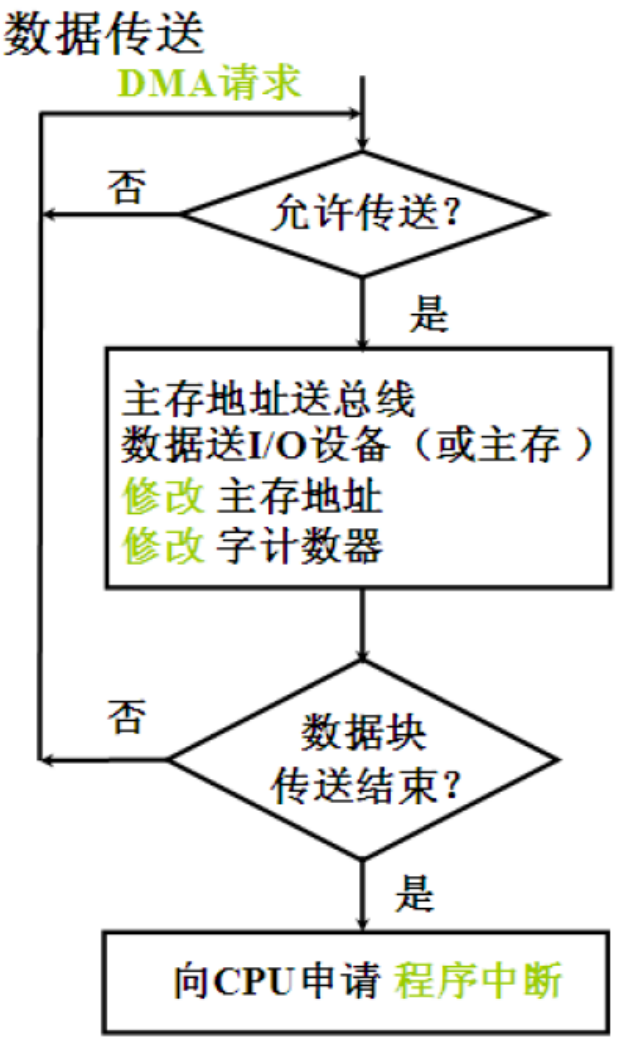
    1）通知DMA控制器传送方向

    2）设备地址写入DMA设备地址寄存器

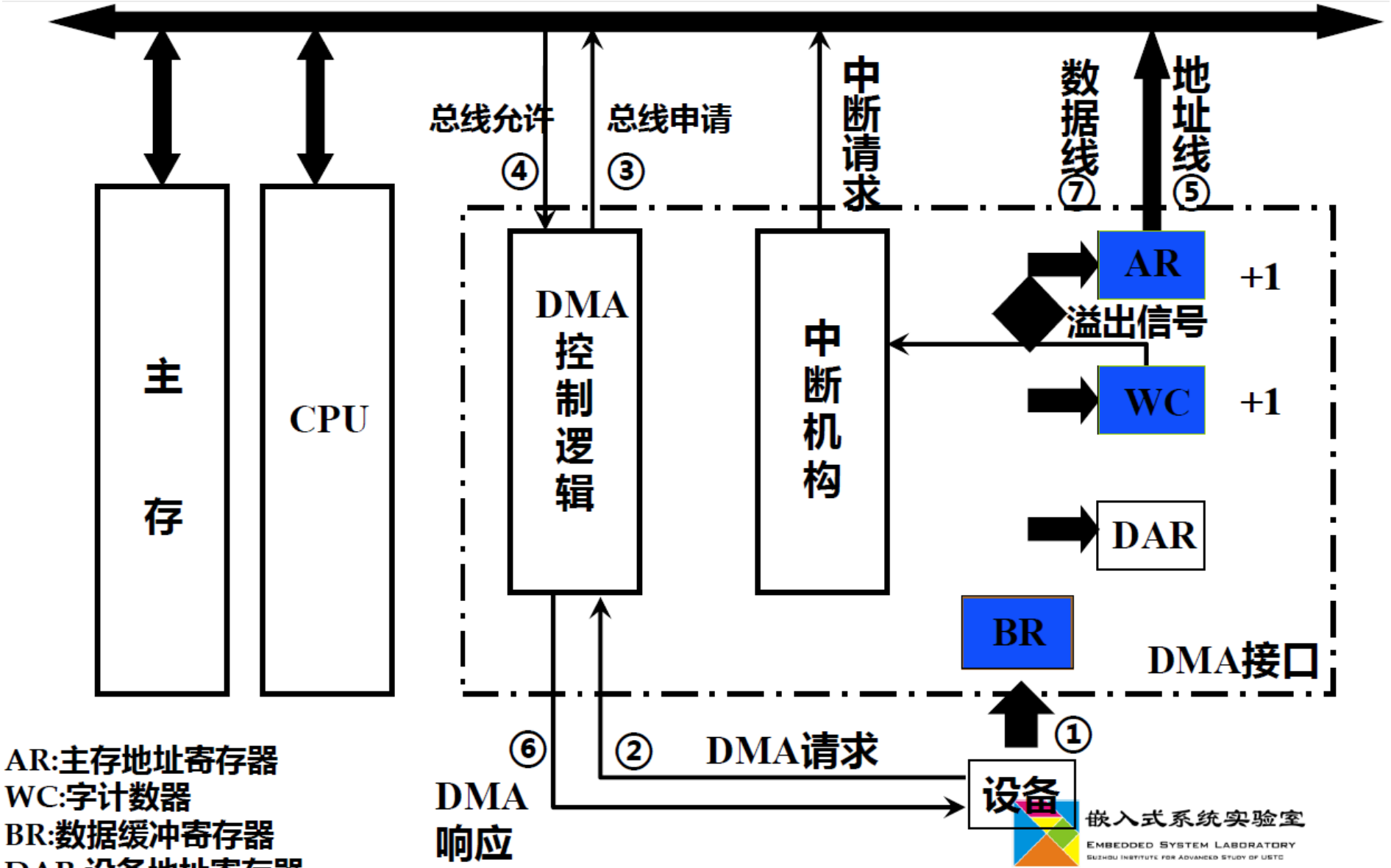
    3）主存地址写入内存地址寄存器

    4）传送字数写入字计数器
  - ✓ 数据传送阶段
    - 以数据块为基本单位
    - 通过循环来实现
  - ✓ 后处理阶段
    - 校验送入主存的数据是否正确
    - 是否继续用DMA

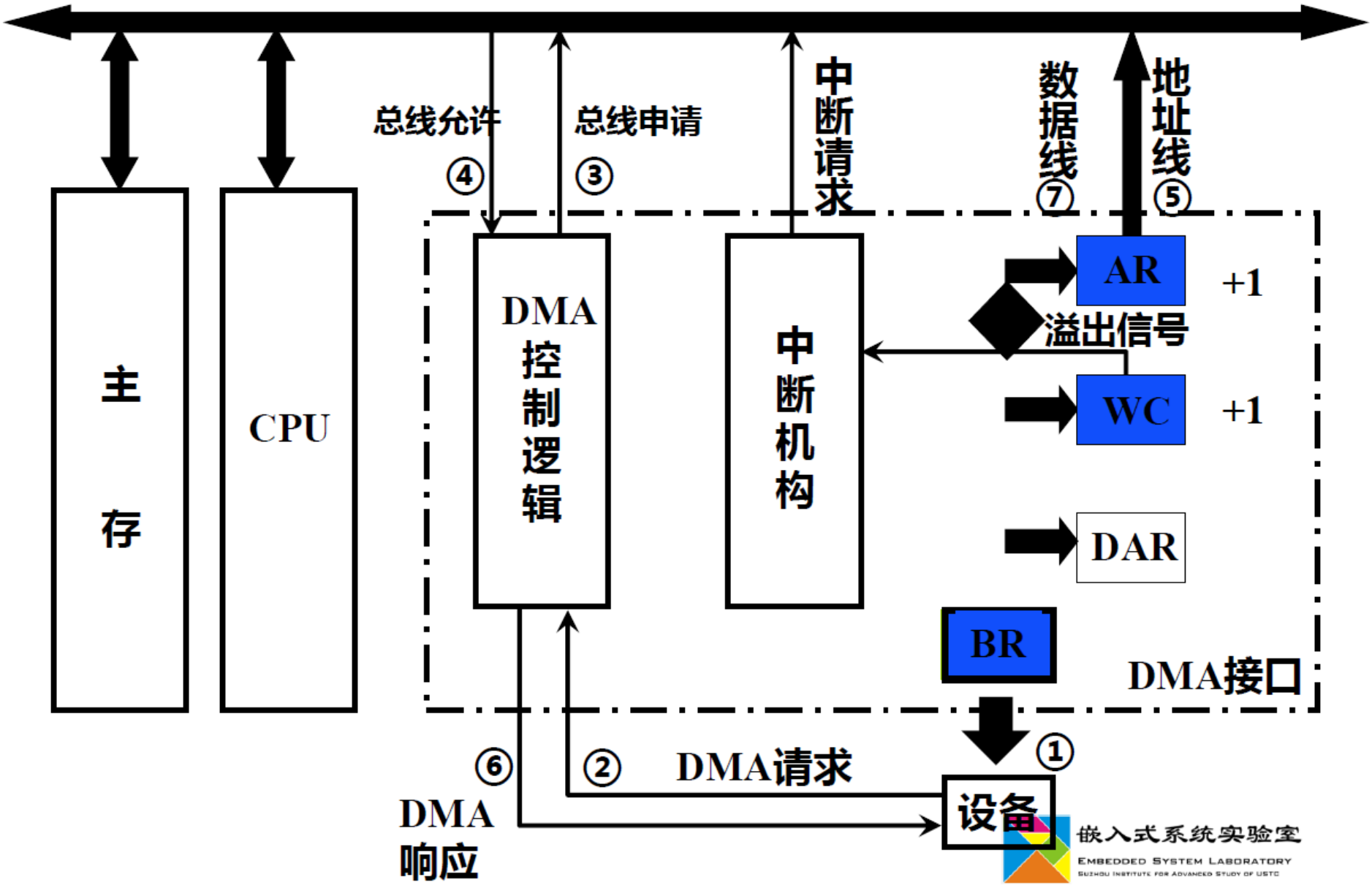
    由中断服务程序完成



输入数据传送过程

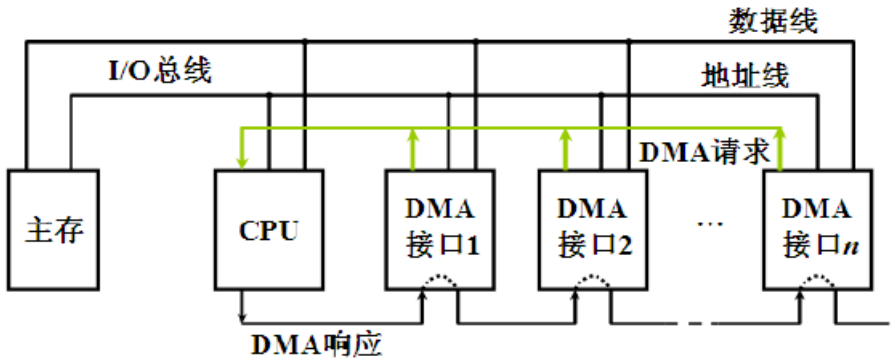

总线允许
④
总线申请
③
中断请求
数据线⑦
地址线⑤
主存
CPU
DMA控制逻辑
中断机构
AR
+1
溢出信号
WC
+1
DAR
BR
DMA接口
⑥
DMA响应
②
DMA请求
①
设备
AR:主存地址寄存器
WC:字计数器
BR:数据缓冲寄存器
DAR:设备地址寄存器

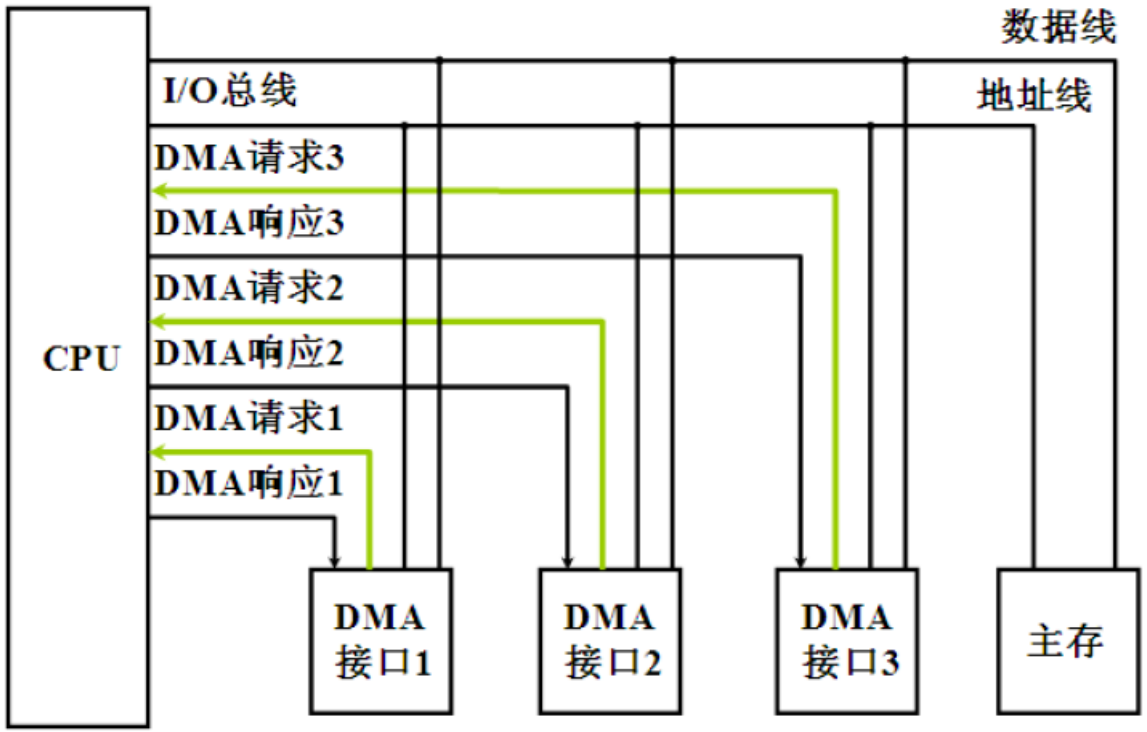
输出数据传送过程


主存
CPU
总线允许
④
总线申请
③
中断请求
数据线
⑦
地址线
⑤
DMA控制逻辑
中断机构
AR
+1
溢出信号
WC
+1
DAR
BR
DMA接口
①
⑥
②
DMA请求
DMA响应
设备

DMA方式

- DMA接口与系统的连接方式
  - ✓公用DMA请求方式
  - ✓独立DMA请求方式

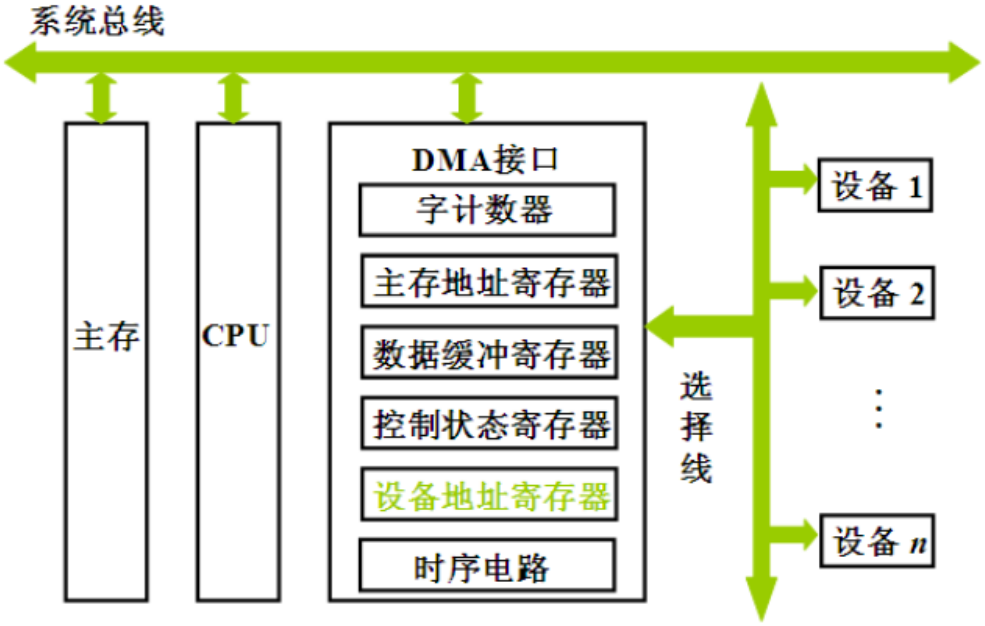


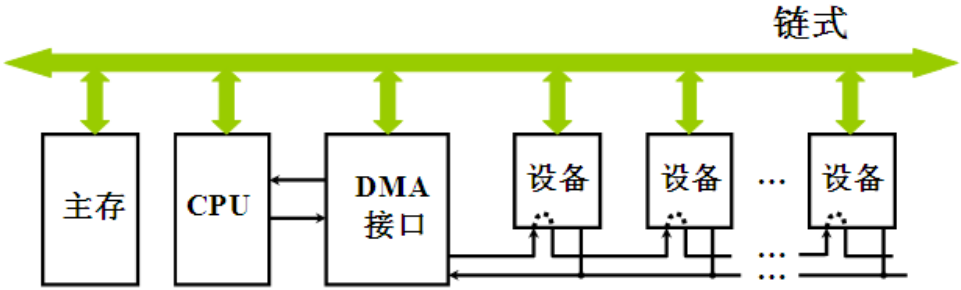


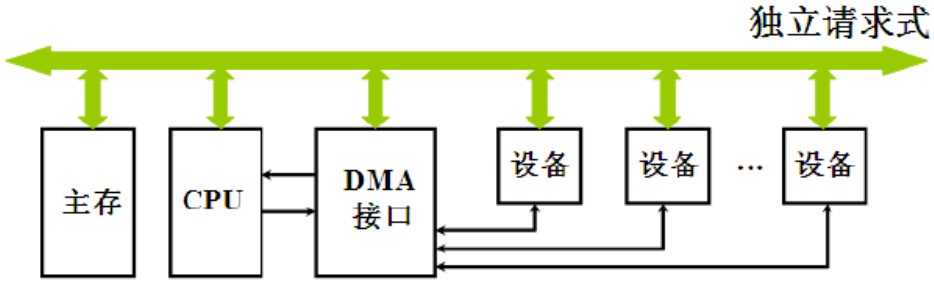
DMA方式

DMA控制器的类型

- ✓选择型
  - 在物理上连接多个设备，在逻辑上只允许连接一个设备
- ✓多路型
  - 在物理上连接多个设备，在逻辑上允许连接多个设备同时工作



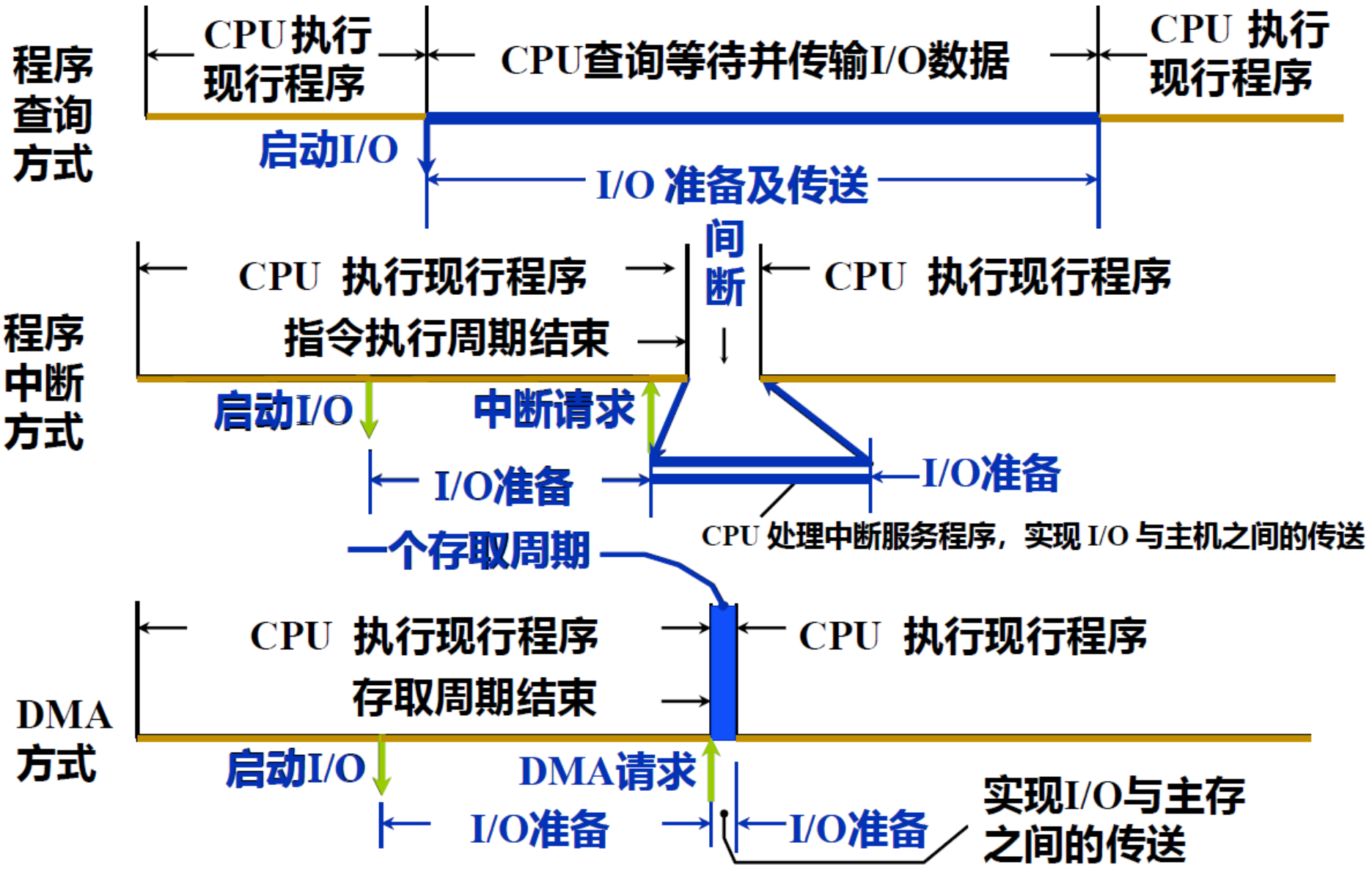




DMA方式与中断方式的比较

	中断方式	DMA 方式
(1) 数据传送	程序	硬件
(2) 响应时间	指令执行结束	存取周期结束
(3) 处理异常情况	能	不能
(4) 中断请求用途	传送数据	溢出、后处理
(5) 优先级	低	高

三种方式的CPU效率比较



通道方式

通道的基本概念

- ✓ 通道：计算机系统中代替CPU管理控制外设的独立部件，是一种能执行有限I/O指令集合——通道命令的I/O处理部件
- ✓ 在通道控制方式中，一个主机可以连接几个通道。每个通道又可连接多台I/O设备
  - 这些设备可具有不同速度，可以是不同种类。
  - 这种输入输出系统增强了主机与通道操作的并行能力以及各通道之间、同一通道的各设备之间的并行操作能力。
  - 为用户提供了增减外围设备的灵活性
- ✓ 采用通道方式组织输入输出系统，多使用主机—通道—设备控制器—I/O设备四级连接方式。
- ✓ 在CPU启动通道后，通道自动地去内存取出通道指令并执行。直到数据交换过程结束向CPU发出中断请求，进行通道结束处理工作

通道方式

通道的功能

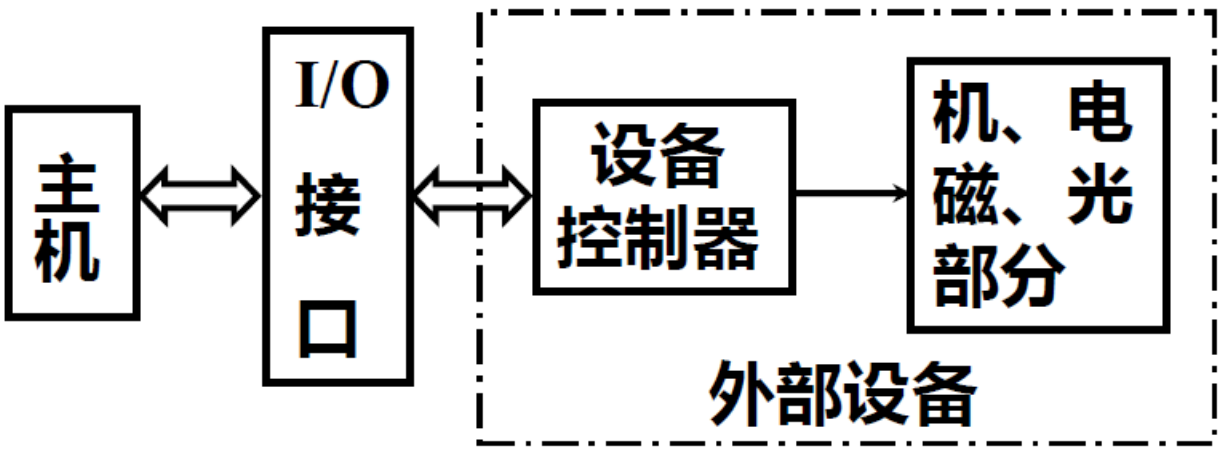
- ✓ 执行通道指令，组织外设和内存进行数据传输，按I/O指令要求启动外设，向CPU报告中断等
- ✓ 具体5项任务
  - 接受CPU的I/O指令，按指令要求与指定的外围设备进行通信
  - 从内存选取属于该通道程序的通道指令，经译码后向设备控制器和设备发送各种命令
  - 组织外设和内存之间进行数据传送，并根据需要提供数据缓存的空间，以及提供数据存入内存的地址和传送的数据量
  - 从外围设备得到设备的状态信息，形成并保存通道本身的状态信息，根据要求将这些状态信息送到内存的指定单元，供CPU使用
  - 将外围设备的中断请求和通道本身的中断请求，按次序及时报告CPU

通道方式

- 通道的发展
  - ✓IO处理机（IOP）
    - 不是一台独立的计算机，而是计算机系统中的一个部件
    - 可以和CPU并行工作，提供高速的DMA处理能力，实现数据的高速传送
    - 有些IOP还提供数据的变换、搜索和字装配/分拆能力。如8位和16位微机中使用的Intel 8089 I/O处理器就是这种通道型I/O处理器

IO设备

一、概述



外部设备大致分三类

1. 人机交互设备　　键盘、鼠标、打印机、显示器
2. 计算机信息存储设备　　磁盘、光盘、磁带
3. 通信设备　　调制解调器等

IO设备

二、输入设备

1. 键盘

按键

判断哪个键按下

将此键翻译成 ASCII 码 （编码键盘法）

2. 鼠标

机械式　　金属球 电位器

光电式　　光电转换器

3. 触摸屏

IO设备

三、输出设备

1. 显示器

(1) 字符显示　　字符发生器

(2) 图形图像显示

2. 打印机

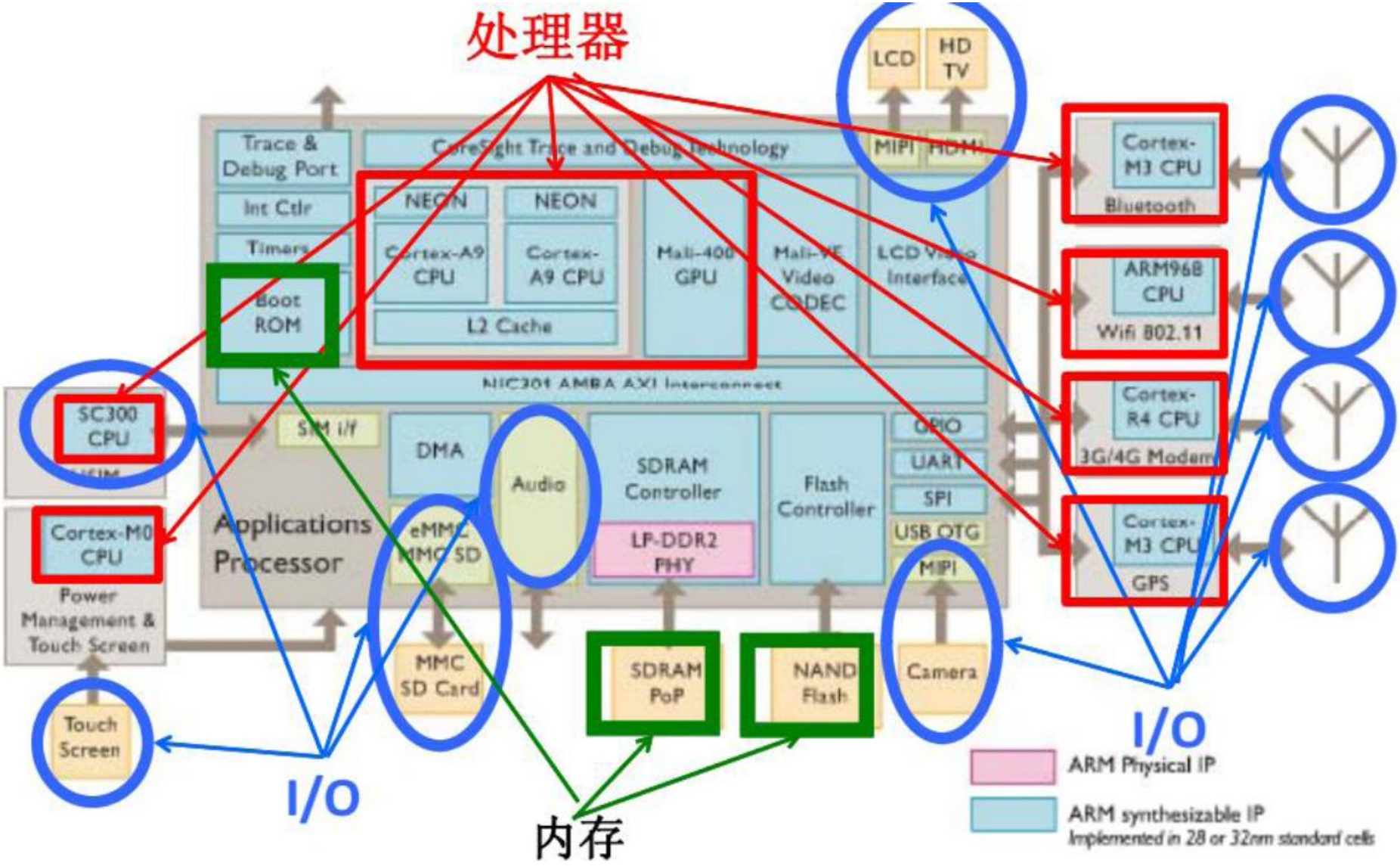
(1) 击打式　　点阵式（逐字、逐行）

(2) 非击打式　　激光（逐页）　喷墨（逐字）

四、其他

模拟/数字（数字/模拟）转换器　　汉字处理

iPhone


处理器
LCD
HD TV
MIPI
HDMI
Trace & Debug Port
CoreSight Trace and Debug Technology
Int Ctlr
Timers
Boot ROM
NEON
NEON
Cortex-A9 CPU
Cortex-A9 CPU
L2 Cache
Mali-400 GPU
Mali-VE Video CODEC
LCD Video Interface
NIC301 AMBA AXI Interconnect
SC300 CPU
SIM I/F
DMA
Audio
SDRAM Controller
LP-DDR2 PHY
Flash Controller
GPIO
UART
SPI
USB OTG
MIPI
Applications Processor
eMMC MMC SD
Cortex-M0 CPU
Power Management & Touch Screen
Touch Screen
MMC SD Card
SDRAM PoP
NAND Flash
Camera
Cortex-M3 CPU
Bluetooth
ARM968 CPU
Wifi 802.11
Cortex-R4 CPU
3G/4G Modem
Cortex-M3 CPU
GPS
I/O
I/O
内存
ARM Physical IP
ARM synthesizable IP
Implemented in 28 or 32nm standard cells

辅助存储

辅存概述

- ✓又叫外部存储器，简称外存
- ✓辅存特点
  - 容量大、速度慢、价格低、可脱机保存信息等
  - 不直接与CPU交换信息
- ✓辅存种类
  - 硬盘、软盘、磁带、光盘、U盘、闪存等

RAID技术

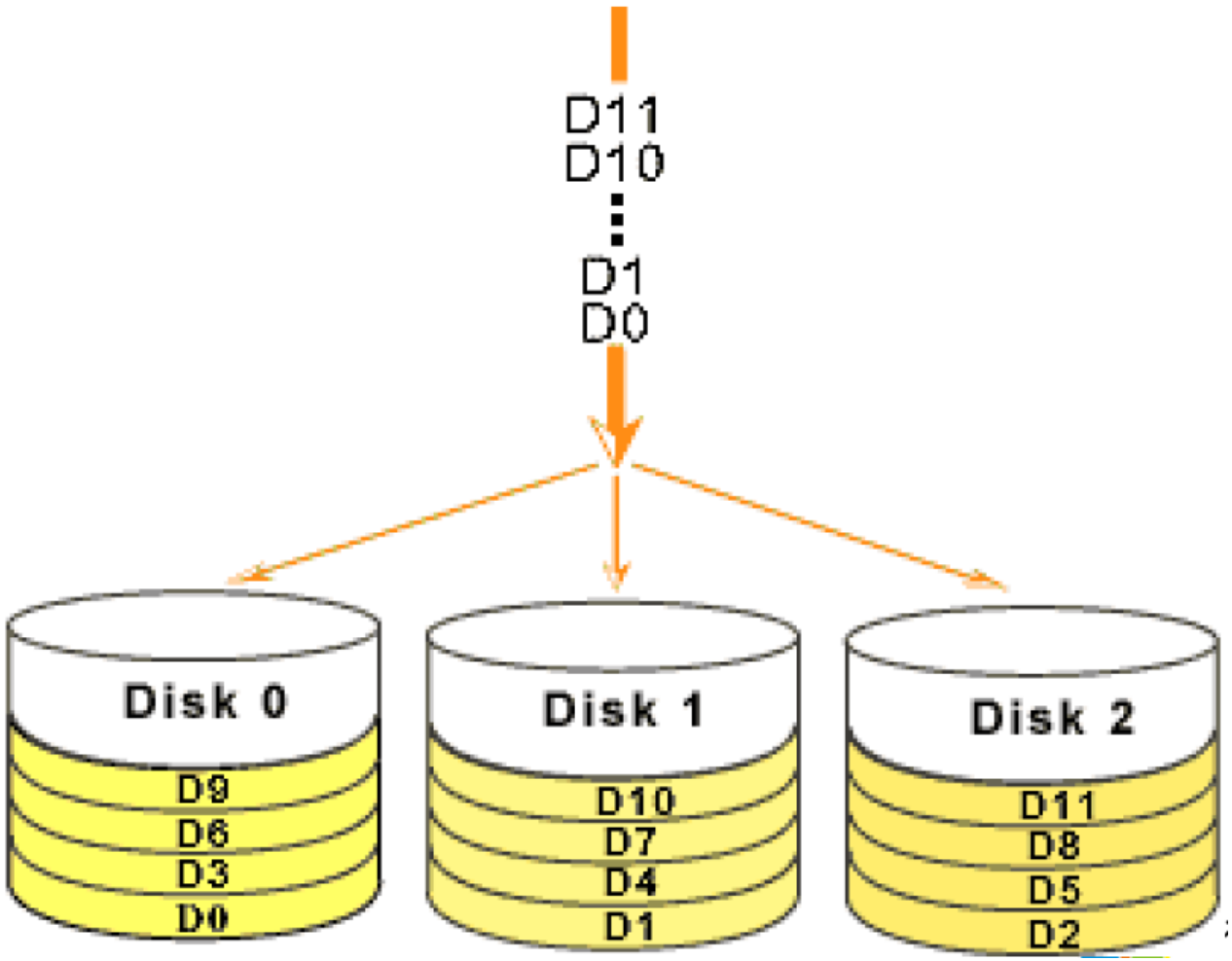
- 1987年，Patterson等@UCB
  - ✓将多只容量较小的、相对廉价的硬盘组合，使其性能超过一只昂贵的大硬盘
- Redundant Array of Independent Disk
  - ✓支持自动检测故障硬盘；
  - ✓支持重建硬盘坏轨信息；
  - ✓支持不须停机的硬盘备援(Hot Spare)
  - ✓支持不须停机的硬盘替换(Hot Swap)
  - ✓支持扩充硬盘容量

RAID级别

- RAID0：无差错控制的区组
- RAID1：镜象结构
- RAID2：带海明码校验
- RAID3：带奇偶校验码的并行传送
- RAID4：带奇偶校验码的独立磁盘结构
- RAID5：分布式奇偶校验的独立磁盘结构
- RAID6：带有两种分布存储的奇偶校验码的独立磁盘结构
  - ✓ 对RAID5的扩展
- RAID7：优化的高速数据传送磁盘结构
  - ✓ 采用并行和Cache技术

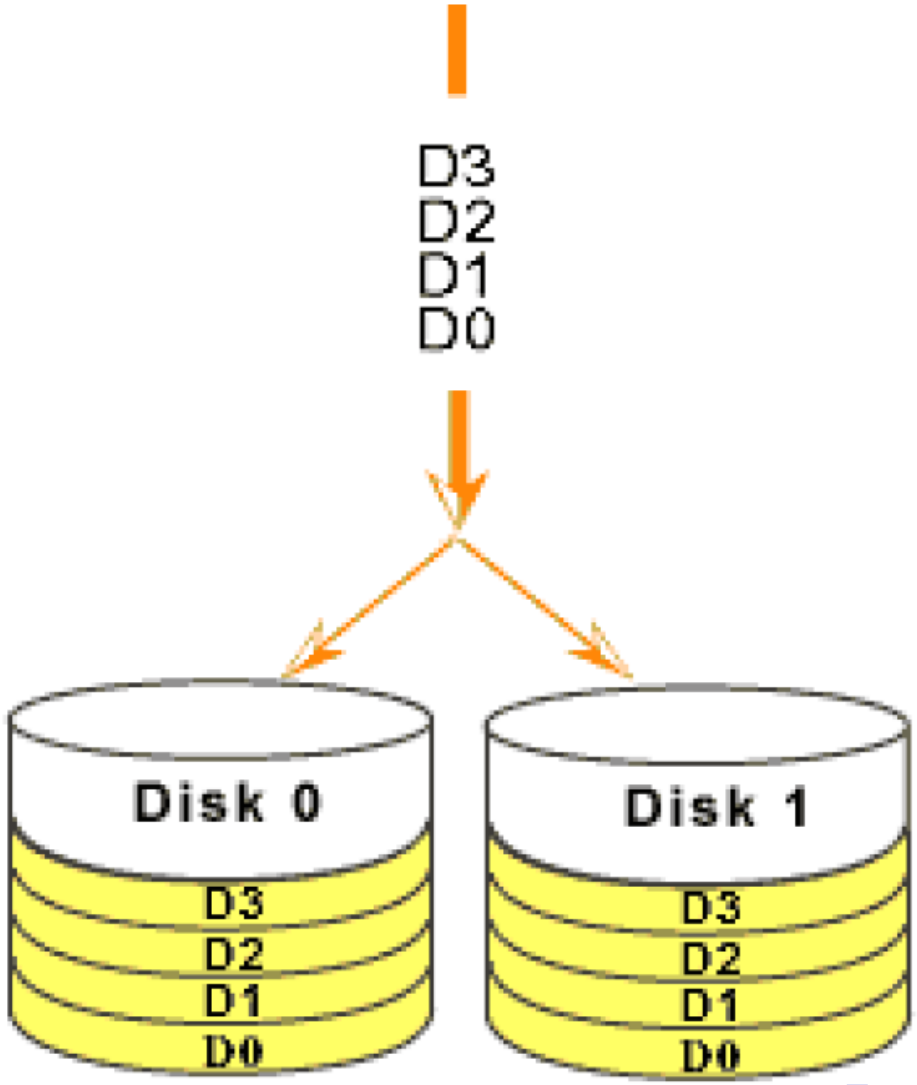
RAID 0（无差错控制的区组）

- 目的：利用多体并行提高存储性能



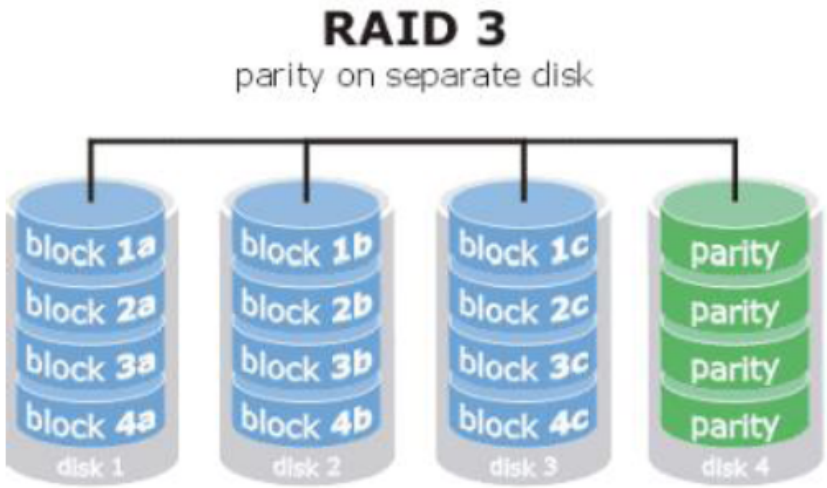
RAID 1（别名：镜像）

- 目标：保证数据的可用性和可修复性



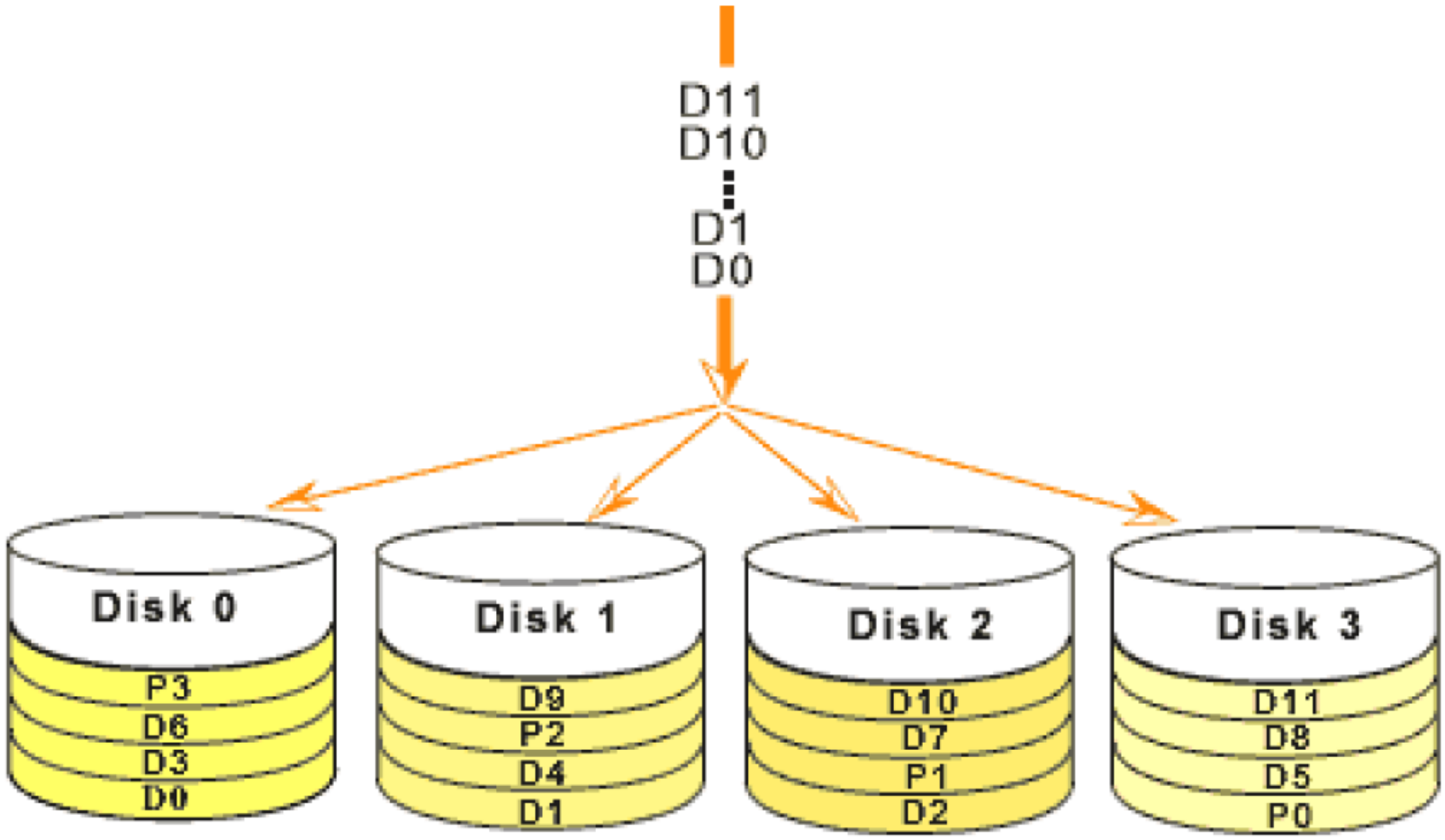
RAID 3

- RAID3：带奇偶校验码的并行传送（检错）
  - ✓将数据条块化分布于不同的硬盘上
    - 条块单位为位或字节。
    - RAID4 （少用）：按数据块访问数据
    - RAID2（少用）：带海明码校验
  - ✓必须要有三个以上的驱动器
  - ✓校验码在写入数据时产生，保存在另一个磁盘上。
  - ✓根据奇偶校验数据可以恢复



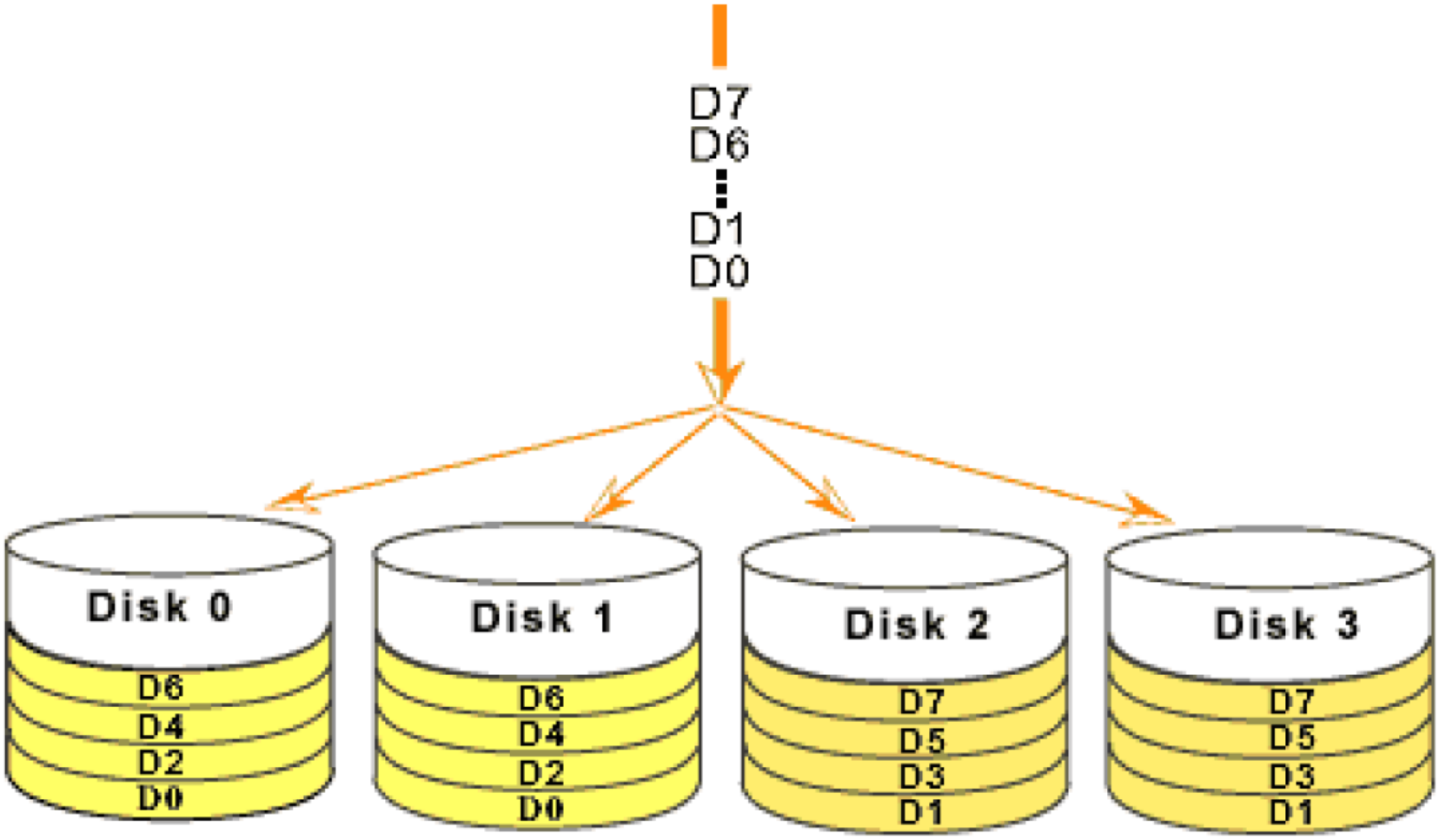
RAID 5

- 分布式奇偶校验的独立磁盘结构
  - ✓奇偶校验码存在于所有磁盘上



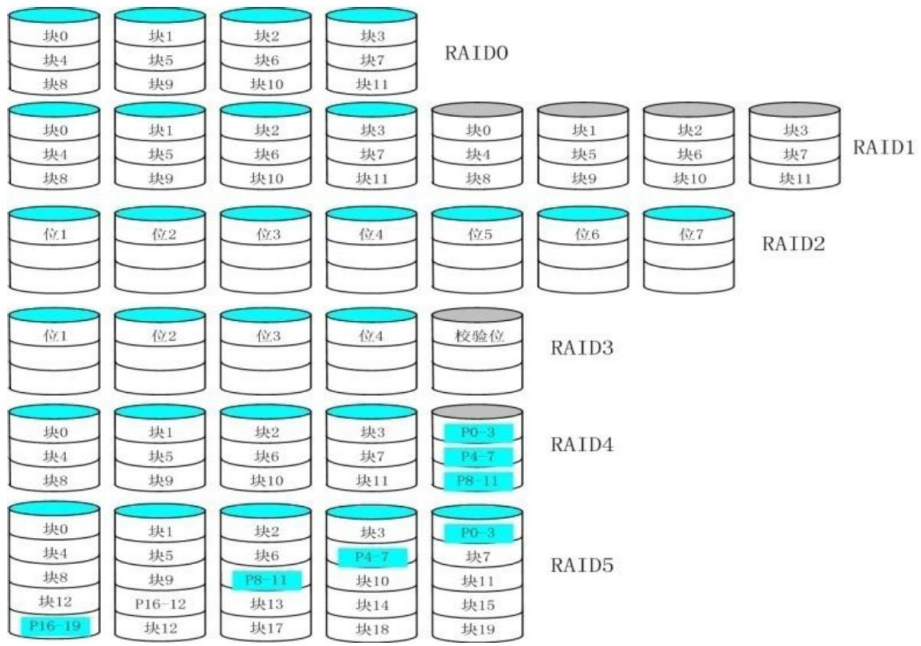
RAID 10 = RAID 0 + RAID 1

- 多体并行（RAID0）+镜像(RAID1)



RAID选择

RAID级别	RAID-0	RAID-1	RAID-3	RAID-5	RAID-10
别名	条带	镜像	专用奇偶位条带	分布奇偶位条带	镜象阵列条带
容错性	没有	有	有	有	有
冗余类型	没有	复制	奇偶校验	奇偶校验	复制
热备盘选项	没有	有	有	有	有
读性能	高	低	高	高	中间
随机写性能	高	低	最低	低	中间
连续写性能	高	低	低	低	中间
需要的磁盘数	一个或多个	只需2个或2xN个	三个或更多	三个或更多	只需4个或4xN个
可用容量	总的磁盘的容量	只能用磁盘容量的50%	(n-1)/n的磁盘容量。其中n为磁盘数	(n-1)/n的总磁盘容量。其中n为磁盘数	磁盘容量的50%
典型应用	无故障的迅速读写，要求安全性不高，如图形工作站等	随机数据写入，要求安全性高，如服务器、数据库存储领域	连续数据传输，要求安全性高，如视频编辑，大型数据库等	随机数据传输要求安全性高，如金融，数据库，存储等	要求数据量大，安全性高，如银行，金融等领域

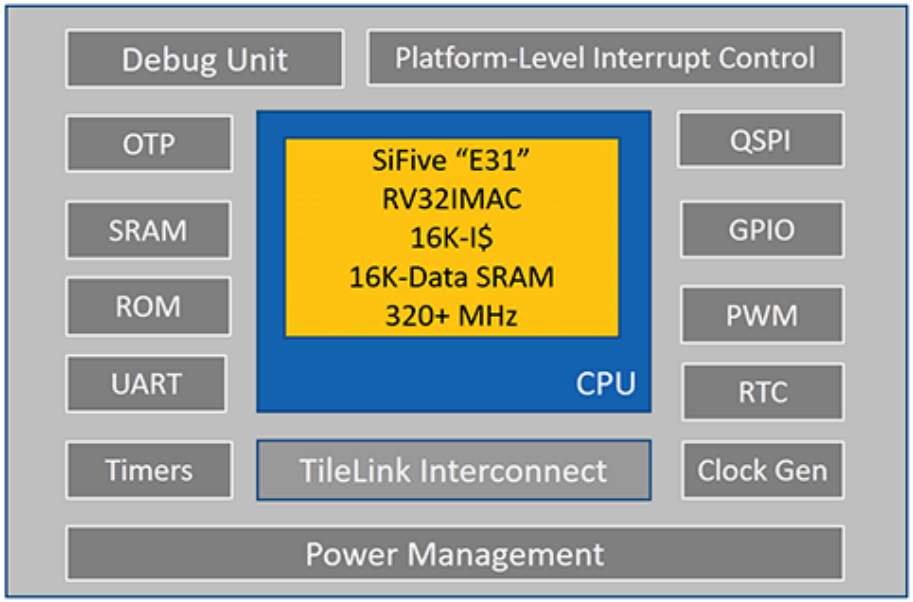


小结

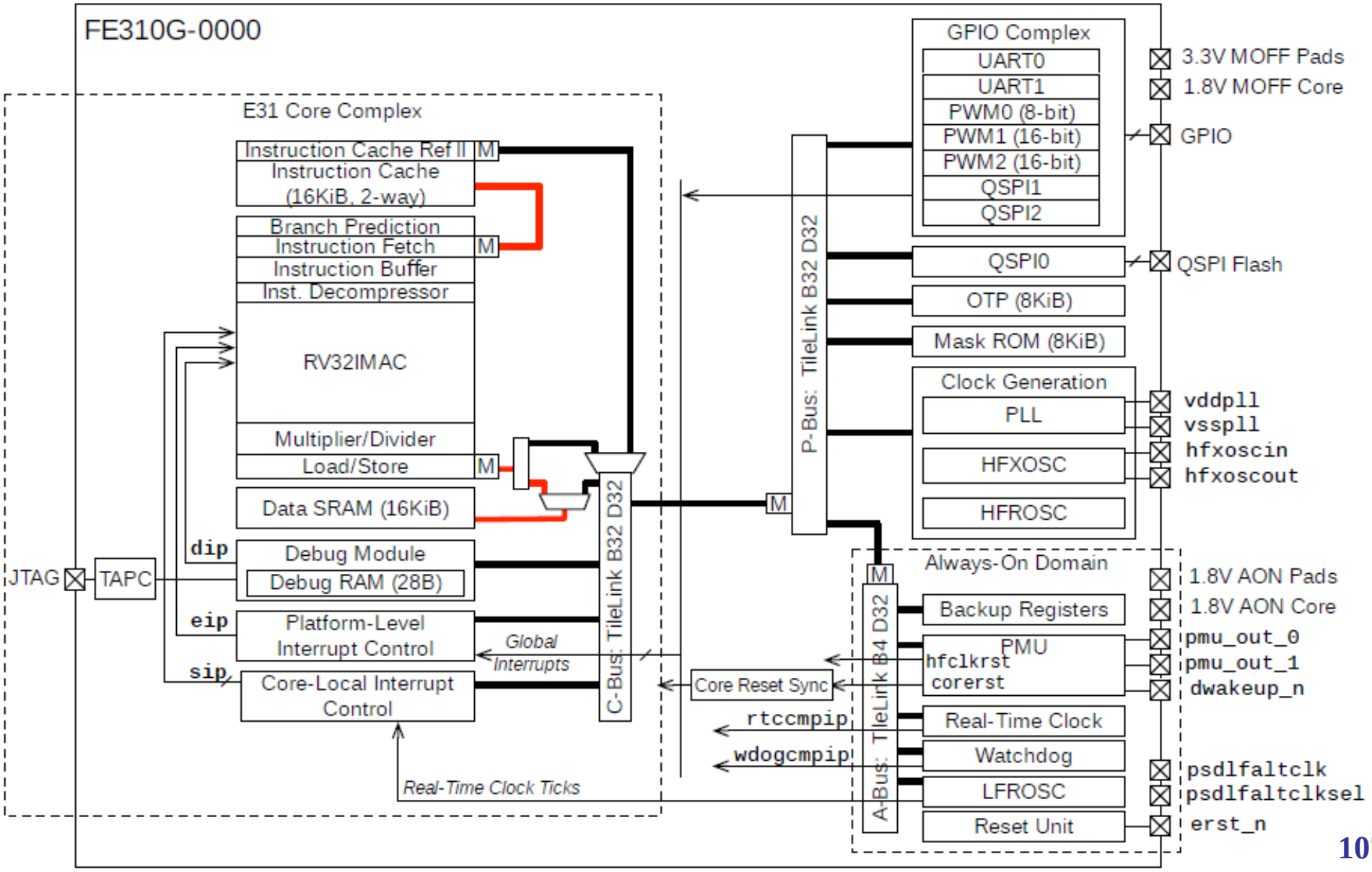
- 内容
  - ✓ I/O系统组成
  - ✓ I/O接口的基本工作机制
    - 信息交换方式
  - ✓ 辅助存储器
    - RAID技术

RVSC-V SOC：FE310-G000

- Freedom E310是SiFive公司产品
  - The Freedom E310 (FE310) is the first member of the Freedom Everywhere family of customizable SoCs
  - Designed for microcontroller, embedded, IoT, and wearable applications
  - The FE310 features SiFive’ s E31 RISC-V Core（RV32IMAC）
    - 单发射顺序执行流水线
  - Running at 320+ MHz



FE310顶层框图


FE310G-0000
E31 Core Complex
Instruction Cache Ref II
M
Instruction Cache
(16KiB, 2-way)
Branch Prediction
Instruction Fetch
M
Instruction Buffer
Inst. Decompressor
RV32IMAC
Multiplier/Divider
Load/Store
M
Data SRAM (16KiB)
dip
Debug Module
Debug RAM (28B)
eip
Platform-Level
Interrupt Control
sip
Core-Local Interrupt
Control
JTAG
TAPC
C-Bus: TileLink B32 D32
Global
Interrupts
Real-Time Clock Ticks
Core Reset Sync
M
P-Bus: TileLink B32 D32
GPIO Complex
UART0
UART1
PWM0 (8-bit)
PWM1 (16-bit)
PWM2 (16-bit)
QSPI1
QSPI2
QSPI0
OTP (8KiB)
Mask ROM (8KiB)
Clock Generation
PLL
HFXOSC
HFROSC
3.3V MOFF Pads
1.8V MOFF Core
GPIO
QSPI Flash
vddpll
vsspll
hfxoscin
hfxoscout
Always-On Domain
M
A-Bus: TileLink B4 D32
Backup Registers
PMU
hfclkrst
corerst
Real-Time Clock
Watchdog
LFROSC
Reset Unit
rtccmpip
wdogcmpip
1.8V AON Pads
1.8V AON Core
pmu_out_0
pmu_out_1
dwakeup_n
psdlfaltclk
psdlfaltclksel
erst_n

FE310特性汇总

Feature	Description	Available in QFN48
RISC-V Core	1× E31 RISC-V cores with machine mode only, 16 KiB 2-way L1 I-cache, and 16 KiB data tightly integrated memory (DTIM).	✔
Interrupts	Software and timer interrupts, 51 peripheral interrupts connected to the PLIC with 7 levels of priority.	✔
UART 0	Universal Asynchronous/Synchronous Transmitters for serial communication.	✔
UART 1	Universal Asynchronous/Synchronous Transmitters for serial communication.	
QSPI 0 Control	Serial Peripheral Interface. QSPI 0 Control has 1 chip select signal.	✔
QSPI 1	Serial Peripheral Interface. QSPI 1 has 4 chip select signals.	✔ (3 CS lines) (2 DQ lines)
QSPI 2	Serial Peripheral Interface. QSPI 2 has 1 chip select signal.	
PWM 0	8-bit Pulse-width modulator with 4 comparators.	✔
PWM 1	16-bit Pulse-width modulator with 4 comparators.	✔
PWM 2	16-bit Pulse-width modulator with 4 comparators.	✔
GPIO	32 General Purpose I/O pins.	✔ (19 pins)
Always On Domain	Supports low-power operation and wakeup.	✔

E31 Core特性

- 单发射顺序执行
- 5级流水线

This chapter describes the 32-bit E31 RISC-V processor core used in the FE310-G000. The E31 processor core comprises an instruction memory system, an instruction fetch unit, an execution pipeline, a data memory system, and support for global, software, and timer interrupts.

Feature	Description
ISA	RV32IMAC.
Instruction Cache	16 KiB 2-way instruction cache.
Data Tightly Integrated Memory	16 KiB DTIM.
Modes	The E31 supports the following modes: Machine

FE310地址空间

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_00FF		Reserved	Debug Address Space
0x0000_0100	0x0000_0FFF	RWXCA	Debug	
0x0000_1000	0x0000_1FFF	R XC	Mask ROM (4 KiB)	On-Chip Non Volatile Memory
0x0000_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region (8 KiB)	
0x0002_2000	0x01FF_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	On-Chip Peripherals
0x0201_0000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_7FFF	RW A	AON	
0x1000_8000	0x1000_FFFF	RW A	PRCI	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	
0x1001_3000	0x1001_3FFF	RW A	UART 0	

FE310地址空间-续

Base	Top	Attr.	Description	Notes
0x1001_4000	0x1001_4FFF	RW A	QSPI 0 Control	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	QSPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	QSPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XCA	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWXCA	DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	

FE310中断架构

- 中断使能位（mstatus.MIE）被清零，所有中断都不被响应
- 中断使能位有效时，响应高优先级的中断
- 进入中断时：

When an interrupt occurs:

- The value of `mstatus.MIE` is copied into `mcause.MPIE`, and then `mstatus.MIE` is cleared, effectively disabling interrupts.
- The privilege mode prior to the interrupt is encoded in `mstatus.MPP`.
- The current `pc` is copied into the `mepc` register, and then `pc` is set to the value specified by `mtvec` as defined by the `mtvec.MODE` described

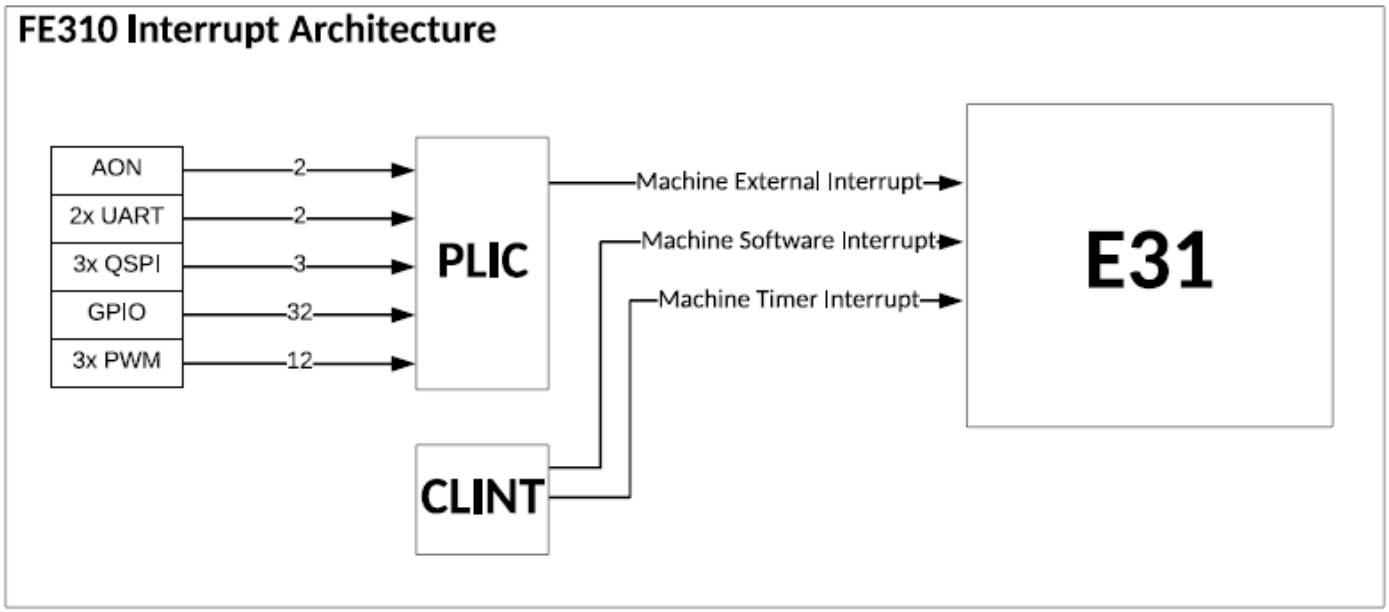
- 中断返回时：

When an MRET instruction is executed, the following occurs:

- The privilege mode is set to the value encoded in `mstatus.MPP`.
- The global interrupt enable, `mstatus.MIE`, is set to the value of `mcause.MPIE`.
- The `pc` is set to the value of `mepc`.

FE310中断架构

- 中断优先级
- 全局外部中断的优先级由PLIC确定
- 中断优先级按降序排列如下
  - Machine external interrupts
  - Machine software interrupts
  - Machine timer interrupts



FE310 GPIO

Table 46: *GPIO Instance*

Instance Number	Address	ngpio
0	0x10012000	32

General Purpose Input/Output

Offset	Name	Description
0x00	input_val	Pin value
0x04	input_en	Pin input enable*
0x08	output_en	Pin output enable*
0x0C	output_val	Output value
0x10	pue	Internal pull-up enable*
0x14	ds	Pin drive strength
0x18	rise_ie	Rise interrupt enable
0x1C	rise_ip	Rise interrupt pending
0x20	fall_ie	Fall interrupt enable
0x24	fall_ip	Fall interrupt pending
0x28	high_ie	High interrupt enable
0x2C	high_ip	High interrupt pending
0x30	low_ie	Low interrupt enable
0x34	low_ip	Low interrupt pending
0x38	iof_en	I/O function enable
0x3C	iof_sel	I/O function select
0x40	out_xor	Output XOR (invert)
0x44	passthru_high_ie	Pass-through active-high interrupt enable
0x48	passthru_low_ie	Pass-through active-low interrupt enable

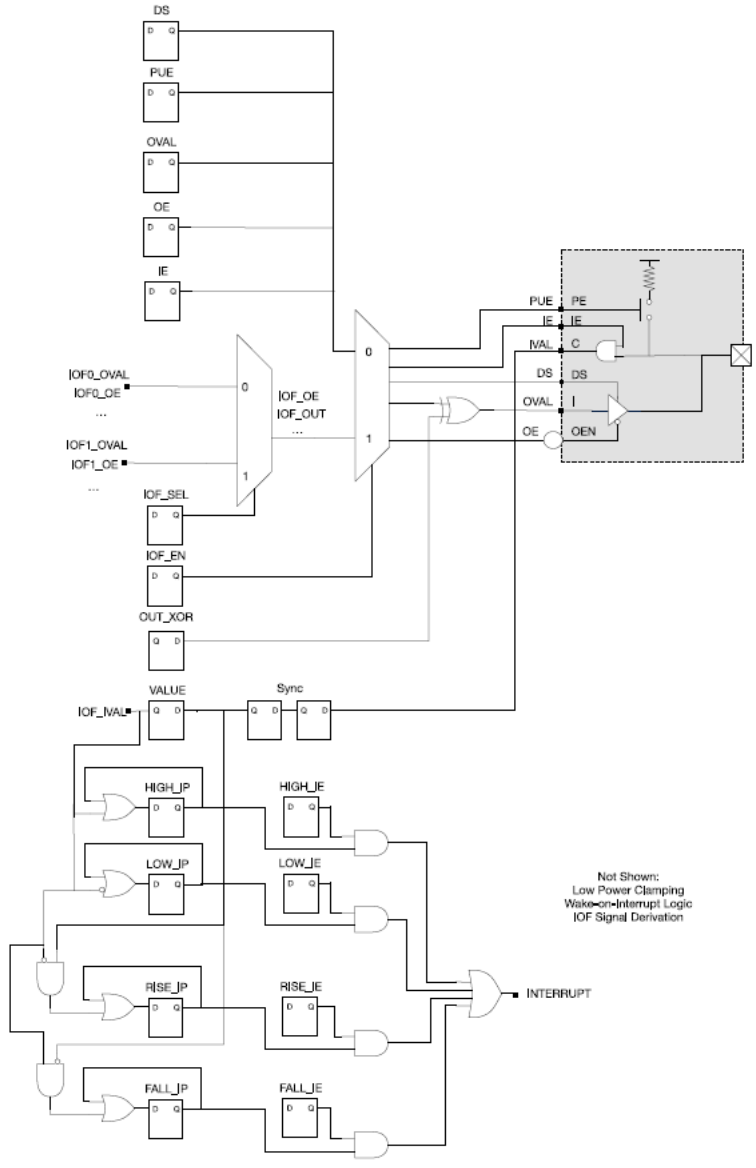


Figure 9: *Structure of a single GPIO Pin with Control Registers. This structure is repeated for each pin.*

FE310 UART

Instance Number	Address	div_width	div_init	TX FIFO Depth	RX FIFO Depth
0	0x10013000	16	3	8	8
1	0x10023000	16	3	8	8

- FE310内包含两个UART控制器
- UART地址空间分布，如右图
- UART接收控制相关（略）
- UART发射控制相关
  - txen为发射使能位
  - nstop指定停止位位数：1位 or 2位
  - txcnt用来指定阈值，TX FIFO内的数据超过阈值则触发中断
  - 其余位保留

Offset	Name	Description
0x00	txdata	Transmit data register
0x04	rxdata	Receive data register
0x08	txctrl	Transmit control register
0x0C	rxctrl	Receive control register
0x10	ie	UART interrupt enable
0x14	ip	UART interrupt pending
0x18	div	Baud rate divisor

Transmit Data Register (txdata)				
Register Offset	0x0			
Bits	Field Name	Attr.	Rst.	Description
[7:0]	data	RW	X	Transmit data
[30:8]	Reserved			
31	full	RO	X	Transmit FIFO full

Transmit Control Register (txctrl)				
Register Offset	0x8			
Bits	Field Name	Attr.	Rst.	Description
0	txen	RW	0x0	Transmit enable
1	nstop	RW	0x0	Number of stop bits
[15:2]	Reserved			
[18:16]	txcnt	RW	0x0	Transmit watermark level
[31:19]	Reserved			

FE310 UART

- UART使用流程
  - UART初始化
    - 配置相关寄存器
  - 接收数据
    - 收到中断信号
    - 进入中断处理程序
    - 从rx fifo读取数据，直至读空
  - 发送数据
    - 向tx fifo写入数据
    - tx使能，发送数据
    - 数据发送完成后触发中断
- UART也可以不使用中断，而采用软件查询的方式

UART Interrupt Enable Register (ie)				
Register Offset	0x10			
Bits	Field Name	Attr.	Rst.	Description
0	txwm	RW	0x0	Transmit watermark interrupt enable
1	rxwm	RW	0x0	Receive watermark interrupt enable

UART Interrupt Pending Register (ip)				
Register Offset	0x14			
Bits	Field Name	Attr.	Rst.	Description
0	txwm	RO	X	Transmit watermark interrupt pending
1	rxwm	RO	X	Receive watermark interrupt pending

Baud Rate Divisor Register (div)				
Register Offset	0x18			
Bits	Field Name	Attr.	Rst.	Description
[15:0]	div	RW	X	Baud rate divisor. div_width bits wide, and the reset value is div_init.
[31:16]	Reserved			

tlclk (MHz)	Target Baud (Hz)	Divisor	Actual Baud (Hz)	Error (%)
200	31250	6400	31250	0
200	115200	1736	115207	0.0064
200	250000	800	250000	0
200	1843200	109	1834862	0.45
384	31250	12288	31250	0
384	115200	3333	115211	0.01
384	250000	1536	250000	0
384	1843200	208	1846153	0.16

FE310其它外设

- SPI
- PWM
- RTC
- WDT
- PMU
- 可通过数据手册了解原理及使用方法

资料来源

- 辅助教材：唐书

- staff.ustc.edu.cn/~llxx
- staff.ustc.edu.cn/~cswang
- SiFive FE310-G000 Manual

- 对于上述作者，一并感谢！