北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Computer Organization and Architecture

Chapter 13

Reduced Instruction Set Computers

School of Computer Science（National Pilot Software Engineering School）
AO XIONG (熊翱)
xiongao@bupt.edu.cn

作业- 1

12.1（a）若在一个8位的计算机上完成的最后操作是两个操作数2和3，如下标志位是什么值?

进位　符号　零　偶校验　上溢　半进位

（b）若两个操作数是-1（2的补码）和+1，标志位是什么值?

作业- 1

进位标志　符号位　零　偶校验　溢出位　半进位

（a）　2和 3相加

```
    0000 0010
+   0000 0011
-------------
    0000 0101
```

进位标志：0

符号位：0

零：0

偶校验：1

溢出位：0

半进位：0

（b）　-1和 1相加

```
    1111 1111
+   0000 0001
-------------
  1 0000 0000
```

进位标志：1

符号位：0

零：1

偶校验：1

溢出位：0

半进位：1

作业- 2

12.9 某时钟速率为2. 5GHz的流水式处理器执行一个有150万条指令的程序。流水线有5段并以每时钟周期1条的速率发射指令。不考虑分支指令和乱序(out-of-sequence)执行所带来的性能损失。

(a)同样执行这个程序该处理器比非流水式处理器加速了多少?

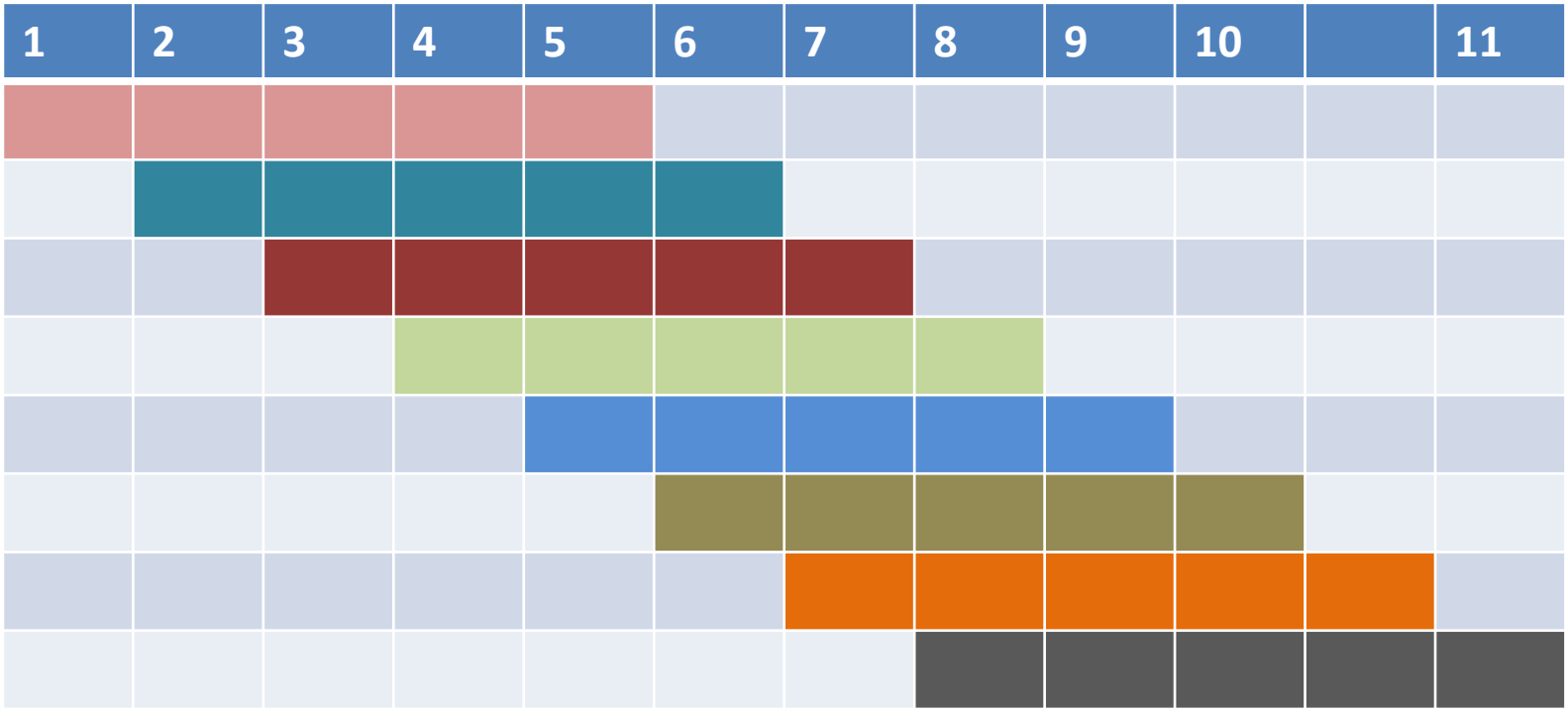
(b)此流水式处理器的吞吐率是多少(以MIPS为单位)?

作业- 2

(a) 因为指令数足够多，采用5段流水，我们认为加速了5倍。

(b) 每个时钟周期结束有一条指令被完成，吞吐率为

$$\frac{1}{1/2.5G} = 2500MIPS$$



作业- 3

12.10.某时钟频率为2.5GHz的非流水式处理器，其平均CPI (每指令周期数)是4。此处理器的升级版本引入了5段流水。然而，由于如锁存延迟这样的流水线内部延迟，使新版处理器的时钟频率必须降低到2GHz。

(a)对典型程序，新版处理器所实现的加速比是多少?

(b)新、旧两版处理器的MIPS速率是多少?

作业- 3

(a) 执行指令数为n，指令流水的段数k=5

假设两个处理器的主频相同，那么不采用流水线的处理器，总执行的指令段数为5n，花费的时间为5n个时间单位，每个时间单位为执行1个阶段的时间。

采用流水线的处理器，执行n个指令需要花费的总时间为n+5-1个时间单位。时间单位和上面的相同。

所以如果主频相同，加速比=5n/(n-4)

但是因为主频降低了，所以加速比=(2/2.5)*5n/（n-4）≈4

作业- 3

(b)

1. 旧版处理器的MIPS速率为$\frac{2.5G}{4}$ =625MIPS

2. 新版处理器，当指令足够多时，每一个时钟周期结束将有一个指令被完成，因此MIPS速率为2000MIPS

问题

(a) 加速比为4

(b) 旧版的MIPS为625，新版的MIPS为2000。新版为旧版的3.2倍。

Why?

1. 指令CPI为4，主频为2.5G。执行一个指令需要4*(1/2.5G)=1.6ns
2. 该指令分为5阶段流水线，每阶段流水线需要0.32ns
3. 由于锁存等原因，主频由2.5G降到2G，理论上速度降低到原来的4/5
4. 所以理论上，每阶段流水线需要0.4ns。这样MIPS就是2.5G
5. 实际上，2G的主频，每个时钟周期是0.5ns
6. 流水线执行过程中，每个阶段需要1个时钟周期。所以实际MIPS为2000
7. 这个过程属于内部碎片。使用统一时钟导致时间浪费

Preface

We have learned:

- Overview
  - Basic Concepts and Computer Evolution 基本概念和计算机发展历史
  - Performance Issues 性能问题
- The computer system
  - Top level view of computer function and interconnection 计算机功能和互联结构顶层视图
  - Cache Memory cache存储器
  - Internal Memory 内部存储器
  - External Memory 外部存储器
  - Input& Output 输入输出
  - Operating System Support 操作系统支持

Preface

We have learned:

- Arithmetic and Logic 算术与逻辑
  - Computer arithmetic 计算机算术
- The central processing unit 中央处理器
  - Instruction sets: characteristics and function 指令集的特征和功能
  - Instruction Sets: Addressing Modes and Formats 指令集的寻址模式和格式
  - Processor Structure and Function 处理器结构和功能
    - Processor Organization 处理器组织
    - Register Organization 寄存器组织
    - Instruction Cycle 指令周期
    - Instruction Pipelining 指令流水线
    - The X86 Processor Family x86处理器
    - The ARM Processor ARM处理器

Review

- 流水线中的有哪几种冒险?
- 数据冒险包括哪几类?
- 有哪些方法解决控制冒险?

Review

- 流水线中的有哪几种冒险?
  - 资源冒险，数据冒险，控制冒险
- 数据冒险包括哪几类?
  - 真相关（写后读），反相关（读后写），输出相关（写后写）
- 有哪些方法解决控制冒险?
  - 多指令流，预取分支，环形缓存，分支预测，延迟分支

Preface

We will focus the following contents today:

- Reduced Instruction Set Computer 精简指令集计算机
  - What are the problem of CISC computer? CISC计算机存在什么问题?
  - Why develop RISC computer? 为什么要发展RISC?
  - What are the main advantages of RISC computer? 有什么优点?
  - Pipeline technology of RISC RISC的流水线

Outline

- Register and instruction architecture 寄存器和指令集
- Reduced Instruction Set Architecture 精简指令集架构
- The Use of a Large Register File 大寄存器组的使用
- Compiler-Based Register Optimization 基于编译器的寄存器优化
- RISC Pipelining RISC的流水线
- RISC Versus CISC Controversy RISC和CISC的争议

Major advanced in computers – 1

- The family concept 系列机概念
  - IBM System/360 1964，DEC PDP-8
  - Separates architecture from implementation 架构和实现分离
- Microprogrammed control unit 微程序控制器
  - Idea by Wilkes 1951，Produced by IBM S/360 1964
  - Easier controller design and implementation 控制器的设计和实现更容易
- Cache
  - IBM S/360 model 85 1969
  - Greatly improves the performance of computer 极大提升了计算机的性能

Major advanced in computers – 2

- Microprocessors 微处理器
  - Intel 4004 1971
  - Reduced the size of the computer 减小了计算机的体积
- Pipelining 流水线
  - Introducing parallelism into instruction execution将并行性引入到指令执行的过程中
  - Greatly improves instruction throughput 大大提高了指令的吞吐量
- Multiple processors 多处理器
  - Multiple processors combine to form a new architecture 多个处理器组合在一起形成全新的架构
  - Further improve the processing capacity of the computer 进一步提高计算机的处理能力

Major advanced in computers – 3

- RISC 精简指令计算机
  - Major challenges to traditional CPU 对传统CPU的重大挑战
  - It has been widely used 曾经得到了广泛的应用
  - Learn from RISC and be widely used in different fields 和RISC相互借鉴，在不同的领域得到广泛的应用
- Solid State RAM 固态RAM
  - Access speed is much faster than mechanical hard disk 访问速度比机械硬盘快很多
  - Greatly improves the computer’s I/O performance 大大提高了计算机的I/O性能

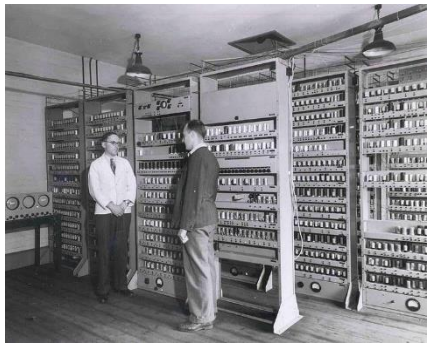
Instruction Set Architecture（ISA） 指令集

- The complete collection of instructions that are understood by a CPU CPU 能理解的完整指令集合
- The interface provided by computer hardware to the upper layer 计算机硬件向上层提供的接口
- It specifies what operations the computer performs, the address space of the operands, and the type of operands 规定了计算机执行什么操作，操作数地址空间以及操作数类型
- Instruction set is the key to computer hardware design 指令集是计算机硬件设计的关键
- Closely related to registers 和寄存器关系密切

Register and ISAs 寄存器和指令集1

- Accumulators 累加器
  - Early stored-program computers had ***one*** register!

    最早的计算机只有1个寄存器
  - Very inconvenient to use 使用很不方便
  - Requires a memory-based operand-addressing mode in instruction 指令中需要基于存储器的操作数寻址模式
    - Example Instructions: add 200
    - Add the accumulator to the word in memory at address 200 将内存200的内容加到累加器
    - Place the sum back in the accumulator 写回累加器



EDSAC in 1949

Intel 8008 in 1972 was an accumulator

Register and ISAs 寄存器和指令集2

- Next step, more registers… 之后，更多的寄存器
  - Dedicated registers 专用寄存器
    - separate accumulators for multiply or divide instructions 乘法或除法指令的单独累加器
    - op-of-stack pointer 堆栈顶部指针
  - Extended Accumulator 扩展累加器
    - Increase bits of register 扩展累加器的位数
    - Increase the number of registers 增加寄存器的数量
- More flexible 更加灵活的指令

Intel 8086
“extended accumulator”
Processor for IBM PCs

Register and ISAs 寄存器和指令集3

- Next step, more registers… 下一步，更多的寄存器
  - General-purpose registers 通用寄存器
    - Registers can be used for any purpose E.g. MIPS, ARM, x86 多种用途的寄存器，例如MIPS，ARM，x86
- Instruction architecture 指令架构
  - *Register-memory* architectures 寄存器-存储器架构
    - One operand may be in memory (i.e. 80386 processors) 一个操作数可能在内存中，例如386
  - *Register-register* architectures (load-store) 寄存器-寄存器架构
    - All operands ***must*** be in registers E.g. MIPS, ARM 所有操作数都在寄存器中，例如MIPS，ARM

Registers of computer -1 计算机的可用寄存器1

Computer	Number of General Purpose Registers	Architectural Style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-Store	1963
IBM 360	18	Register-Memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-Memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-Memory, Memory-Memory	1977

早期的计算机，大部分的寄存器都很少，只有1个累加器

Registers of computer -2 计算机的可用寄存器2

Computer	Number of General Purpose Registers	Architectural Style	Year
Intel 8086	1	Extended Accumulator	1978
Motorola 6800	16	Register-Memory	1980
Intel 80386	8	Register-Memory	1985
ARM	16	Load-Store	1985
MIPS	32	Load-Store	1985
HP PA-RISC	32	Load-Store	1986
SPARC	32	Load-Store	1987
PowerPC	32	Load-Store	1992
DEC Alpha	32	Load-Store	1992
HP/Intel IA-64	128	Load-Store	2001
AMD64 (EMT64)	16	Register-Memory	2003

RISC架构的处理器的寄存器数量要求较多，因为它的所有操作数都要在寄存器中

Comparison of processors 处理器的比较

	Complex Instruction Set (CISC) Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40–520	32	32	40–520	32
Control memory size (Kbits)	420	480	246	—	—	—	—	—
Cache size (KBytes)	64	64	8	32	128	16–32	32	64

- 指令数量、长度、寻址方式等方面，CISC明显高于RISC
- RISC和超标量的通用寄存器数量比CISC要多
- RISC和超标量一般采用硬布线控制，所以没有配置控制存储器

Why registers are so important?

Register has the following characteristics:

- On the same chip with ALU and controller, with the shortest distance 与ALU、控制器在同一个芯片上，距离最短
- Connect directly to ALU 直接和ALU连接
- Fastest storage unit 速度最快的存储部件
- High cost, small quantity 成本高，数量少
- Short addressing bits 寻址位数短

Influence of registers 寄存器的影响1

- Development history of registers is closely related to the development of instruction sets 寄存器的发展历史，和指令集的发展紧密相连
  - Early CPU only has one register, and memory based operand addressing must be used 早期的CPU 只有1个寄存器，必须要使用存储器操作数寻址
  - Later，special registers are set to improve the execution efficiency of instructions 之后设置专用寄存器，可以提高指令执行效率
  - General registers are set to provide flexibility for user programs, avoid frequent access to memory, and improve efficiency 设置通用寄存器，给用户程序提供了灵活性，避免了频繁访问存储器，提高效率
  - Multiple registers also make instruction sets and addressing more flexible 多个寄存器也使得指令集和寻址方式更加灵活

Influence of registers 寄存器的影响2

- The full use of registers is an important part of instruction set design 对寄存器的充分利用，是指令集设计中的重要内容
  - CPU can directly access registers CPU能直接访问寄存器
  - Little 数量少
  - Fast 速度快
  - Saving instruction bits 节省指令位数
- Methods for optimizing register usage 优化寄存器使用的方法
  - Configure more registers in the CPU CPU 中配置更多的寄存器
  - Compiler based optimization 基于编译器的优化

Summary

- Instruction set is the key to computer hardware design 指令集是计算机硬件设计的关键
- Registers and instruction sets are closely related 寄存器和指令集关系密切
- The configuration and use of registers are one of the core elements of RISC and CISC design 寄存器的配置和寄存器的使用，是RISC和CISC设计的核心之一

Outline

- Register and instruction architecture 寄存器和指令集
- Reduced Instruction Set Architecture 精简指令集架构
- The Use of a Large Register File 大寄存器组的使用
- Compiler-Based Register Optimization 基于编译器的寄存器优化
- RISC Pipelining RISC的流水线
- RISC Versus CISC Controversy RISC和CISC的争议

Why CISC? -1

- CISC--Complex Instruction Set Computer 复杂指令集计算机
- With the development of computer technology, an instruction set design concept is proposed 随着计算机技术的发展而提出的一种指令集设计理念
- Hardware is getting cheaper and software is getting more complex 硬件越来越便宜，而软件越来越复杂
  - Machine language or assembly language cannot meet the needs of programmers 机器语言或汇编语言不能适应程序员的需要
  - More powerful and complex high-level languages meet the needs of complex programming 功能更强大、更复杂的高级语言满足了复杂编程的需要

Why CISC? -2

- The gap between high-level programming languages and instruction sets is growing 高级编程语言和指令集之间的差距越来越大
  - Need more complex compilers to translate high-level languages 需要更复杂的编译器来翻译高级语言
  - A high-level language statement requires multiple instructions to complete, low execution efficiency 一个高级语言语句需要多个指令来完成，执行效率低
- CISC proposed by instruction set designer 指令集设计人员提出CISC
  - Provides more types of instructions, and even uses a single instruction to implement complex high-level language statements 提供更多类型的指令，甚至用单个指令来实现复杂的高级语言语句
  - Provide more addressing modes to meet the needs of high-level languages for various addressing 提供更多的寻址模式，适应高级语言对各种寻址的需要

Characteristics of CISC CISC的特点

- There are many kinds of instruction opcodes 指令操作码种类多
  - The opcode of X86 is 1~2 bytes X86的操作码为1~2个字节
- Variable instruction length 指令长度可变
  - The instruction length of X86 is 1~16 bytes X86的指令长度为1~16字节
- Various addressing modes 寻址模式多种多样
  - X86 has 9 addressing modes, such as base address proportional index band offset X86有9种寻址模式，典型的如基址比例变址带偏移量

Ideal effect of CISC CISC的理想效果1

- Compilers are easy to write 编译器容易编写
  - The instruction set provides many types of instructions 指令集提供的指令类型多
  - The compiler can use the most appropriate instructions to translate statements in high-level languages. Less instructions after compilation and less space. 编译器可以使用最合适的指令来翻译高级语言的语句。编译后的指令条数少，占用空间小
  - The instruction set provides rich addressing modes 指令集提供丰富的寻址模式
  - Meet the requirements of high-level language for flexible addressing mode 满足高级语言对灵活寻址方式的要求

Ideal effect of CISC CISC的理想效果2

- Improve the execution efficiency of high-level language statements 提高高级语言语句的执行效率
  - It used to require multiple instructions to complete statements in a high-level language, but now it can be completed with one instruction, and some high-level language statements can be implemented at the hardware level 原来需要多个指令来完成一个高级语言的语句，现在可以用一个指令就可以完成，在硬件级实现某些高级语言的语句
  - Instruction level addressing mode to help implement complex instructions 指令级的寻址模式，帮助实现复杂的指令

是否真能达到这个效果呢？？？

Instruction execution characteristics 指令执行的特征

- Operations performed 操作类型
  - Functions that can be completed by CPU and interaction with memory CPU以及与存储器的相互交互，能够完成的功能
- Operands used 操作数
  - Type and frequency of operands，which determine the organization and addressing mode of the storage system 操作数的类型、使用的频度，确定了存储系统组织架构和寻址方式
- Execution sequencing 执行顺序
  - Control function and pipeline organization 控制功能和流水线组织

Operations 操作类型

- It is best to optimize the most used and time-consuming statements 对使用最多、耗时最长的语句进行优化，效果最好
- Assignments 赋值语句
  - Movement of data 数据的移动
- Condition （Loop and IF ）条件语句（循环和判断）
  - Sequence control 顺序控制
- Procedure call-return 过程的调用与返回
  - Very time consuming 非常耗时

Weighted proportion of Operations 操作的加权比例

	Dynamic Occurrence		Machine-Instruction Weighted		Memory-Reference Weighted	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%	14%	15%
LOOP	5%	3%	42%	32%	33%	26%
CALL	15%	12%	31%	33%	44%	45%
IF	29%	43%	11%	21%	7%	13%
GOTO	—	3%	—	—	—	—
OTHER	6%	1%	3%	1%	2%	1%

- 第2、3列是原始的出现频度。4、5列是将高级语言编译后产生的机器指令数后加权得到的比例，6、7列是考虑指令访问存储器次数的比例
- 过程调用和返回是典型的高级语言程序中最耗时的操作
- 循环语句、条件语句和赋值语句也占很大的比重

Operands 操作数

- Mainly local scalar variables 主要是局部标量变量
- Optimisation direction should concentrate on storage and accessing local variables 优化方向应集中于本地变量的存储和访问

	Pascal	C	Average
Integer Constant	16%	23%	20%
Scalar Variable	58%	53%	55%
Array/Structure	26%	24%	25%

Procedure calls 过程调用

- Very time consuming 非常耗时
  - Depends on number of parameters passed 依赖于参数传递的数量
  - Depends on level of nesting 依赖于嵌套级别
- Characteristics 特点
  - Number of parameter is mostly less than 6 传递参数数量绝大部分小于6个
  - Most variables are local 大部分变量是局部的
- It further explains that operand access is highly localized 进一步说明操作数访问是高度局部化的

Summary

- Assignment statement 赋值语句
  - high proportion, valuable to improve efficiency 占比高，提高效率有价值
  - Need to access cache or storage 需要访问cache或存储器
  - Use register access to reduce memory access and improve efficiency 采用寄存器访问的方式，减少存储器访问，提高效率
- Condition and procedure calls 条件转移和过程调用
  - time consuming, high proportion, valuable to improve efficiency 耗时长，比例较高，提高效率有价值
  - Influence the execution of the pipeline 影响流水线的执行
  - Design a better pipeline to reduce the impact of transfer statements on the water line 设计更好的流水线，降低转移语句对流水线的影响

单纯依靠提供接近于高级语言的指令并不一定能提高典型语句的执行效率！！

Why CISC? -1 为什么用CISC -1?

- Ideal 1 of CISC: Compilers are easy to write CISC理想1：编译器简单，容易写
- Implementation method：complex instruction 实现方法：复杂指令
- Compiler simplification? 编译器更简单?
  - Because of strict requirements of instruction design，compiler needs high-level language strictly meet the instruction 指令设计要求严格，编译器需要高级语言必须符合指令的要求
  - The compiler needs to optimize machine instructions to reduce the length of generated machine code and meet the requirements of pipeline operation. 编译器需要进行机器指令的优化，以减少生成后的机器代码的长度，并且满足流水线运行的要求
  - This is also difficult to achieve for complex instructions 对于复杂指令很难达到优化效果

Why CISC? -2 为什么用CISC -2?

- Ideal 2 of CISC: smaller programs CISC理想2：程序更短
- Implementation method：mores instruction 实现方法：更多指令
- Smaller programs? 更短的程序?
  - Program looks using less memory 看起来占用更少的内存
  - Memory is now cheap 内存便宜
  - May not occupy less bits, just look shorter in symbolic form 只是符号上显得短，实际占用内存并不少
  - More instructions require longer op-codes 更多的指令占用长的操作码
  - CISC has no fewer machine instructions than RISC CISC的机器指令数量并不会比RISC的指令少

Why CISC? -3 为什么用CISC -3?

- Ideal 3 of CISC: high efficiency CISC理想3：效率高
- Implementation method：more instructions and more addressing mode 实现方法：更多指令，更多的寻址模式
- Faster programs? 更快的程序?
  - Compiler bias towards use of simpler instructions 编译器偏向于使用简单指令
  - CISC need more complex control unit CISC需要更复杂的控制单元
  - Microprogram control store larger 使用的微程序控制机制需要更大的存储
  - Simple instructions take longer to execute 简单指令执行花的时间更长

Are you sure CISC is the ideal solution? 你还能确定CISC是理想的方案吗?

Conclusion 结论

- The goal CISC hopes to achieve is actually contradictory to the way CISC realizes it CISC希望达到的目标和CISC的实现方式矛盾
- Target：improve operational efficiency by optimizing the most frequently used and time-consuming functions 目标：通过优化最常用和最耗时的功能来提高运行效率
- Following methods may be better choices 下述方法可能更好
  - More registers to reduce memory access 更多的寄存器减少存储器的访问
  - Careful design of pipeline to improve the efficiency of the pipeline 精心设计的流水线来提高流水线的效率
  - Careful designed simple instruction set to improve the efficiency of instruction execution 精心设计的简单指令集来提高指令执行的效率

That is RISC!

- RISC：Reduced Instruction Set Computer 精简指令集计算机
- Key features 主要特征
  - Large number of general purpose registers 大量通用寄存器
  - Compiler technology to optimize register use 编译器技术优化寄存器使用
  - Limited and simple instruction set 有限且简单的指令集
  - Emphasis on optimising the instruction pipeline 强调指令流水线的优化

Main contributors of RISC -1 RISC的主要贡献人员1

- John Cocke：IBM
  - Aware that more streamlined instruction set design can help reduce the difficulty and cost of hardware development, and it is also conducive to the compiler's code optimization 意识到更加精简的指令集设计有助于减少硬件开发难度和成本，同时也有利于编译器进行代码优化
  - Idea: Possible to make a very small and very fast core 思想：可以制作一个非常小和非常快的核心
  - Influences: Known as “the father of RISC Architecture”. Turing Award Recipient and National Medal of Science. 影响力：RISC之父，图灵奖和国家科学奖

Main contributors of RISC -2 RISC的主要贡献人员2

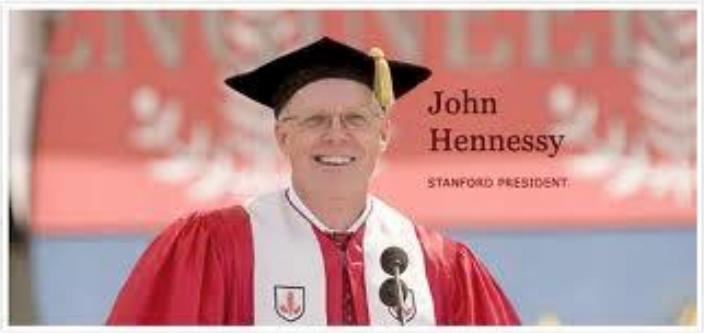
- Dave Patterson：UC Berkeley
  - Recognize the weakness of CISC and start new 认清CISC的弱点，着手进行新的设计
  - RISC-I processor: use ½ transistors to get 3x faster RISC1处理器：使用1/2的晶体管，得到了3倍速度
  - Influences: Sun SPARC from his achievements 影响力：SUN SPARC芯片来自于他的成果

Main contributors of RISC -3 RISC的主要贡献人员3

- John L. Hennessy：Stanford
  - Greatly promoted the development of RISC architecture 极大地推动了RISC架构的发展
  - Basic thought：Simple pipelining, keep full 简单流水线，保持充满状态
  - Influences: In 1984, MIPS (Microprocessor without interlocked pipelined stages) was founded 影响力：1984创立了MIPS公司

和David Patterson一起荣获

2017年ACM图灵奖



Typical features of RISC RISC的典型特征1

- Simplified instruction set 简化的指令集
  - Standardized, fixed length instruction format (ARM instructions are all 32-bit) 采用规范化的、定长的指令格式（ARM的指令均为32位）
  - Limited operation types, only 8-bit operation code 有限的操作类型，操作码只有8位
  - Fetching and decoding instruction become easier 取指、解码变得更容易
  - One instruction per machine cycle 每个机器周期一个指令
  - Hardwired design (no microcode) 硬布线设计，没有微代码

Typical features of RISC RISC的典型特征2

- Use registers whenever possible 尽量使用寄存器
  - Except for load/save instructions, other instructions are for register operations 除了加载/保存指令，其他指令都是针对寄存器操作
  - Data operations can only be performed in registers 数据运算只能在寄存器中进行
  - Memory access has only three addressing modes 内存访问只有三种寻址模式

Typical features of RISC RISC的典型特征3

- Better pipeline design 更好的流水线设计
  - Instruction pipeline is carefully designed to better meet the impact of conditional branches and procedure calls on the flow pipeline 精心设计指令流水线，更好地满足条件分支和过程调用对流水线的影响
  - Each instruction is conditionally executed, which can reduce branches 每个指令都是条件执行，这样可以减少分支

Influence of RISC concept RISC概念的影响1

- RISC's design concept gave birth to a series of new computer architectures RISC的设计理念催生了一系列新的计算机架构
  - Successful RISC machines include DEC Alpha, MIPS, SUN SPARC, etc 比较成功的RISC机器包括：DEC Alpha，MIPS，SUN SPARC等
  - RISC dominated the server market around the 1990s 20世纪90年代前后，RISC在服务器市场占据统治地位
  - Intel also designed i860 series RISC processors while introducing CISC product line Intel在推出CISC产品线的同时，也推出了i860系列RISC处理器
  - Simplified pipeline design on RISC instruction set is becoming more and more attractive RISC指令集上的简化流水线设计越来越具有吸引力

Influence of RISC concept RISC概念的影响2

- RISC and CISC learn from each other RISC和CISC相互借鉴
  - Intel P6 adopts "**Out-of-order issue/out-of-order**" technology, and realizes the conversion of CISC instruction set to RISC instructions in the decoding stage Intel Pentium 6 采用"乱序发射乱序完成"机制，实现了CISC指令集在解码阶段上向RISC类指令的转化
  - Make up for the disadvantage of CISC pipeline implementation 弥补了CISC流水线实现上的劣势
  - RISC is also learning from CISC, and both sides are learning from each other RISC也在向CISC取经，双方都在相互取长补短

Influence of RISC concept RISC概念的影响3

- Current situation 当前情况
  - With the development of architecture and microelectronics technology, the so-called disadvantage of CISC in structure is gradually reduced 随着体系结构和微电子技术的进步，CISC在结构上的所谓劣势逐渐缩小
  - RISC's theory of superiority has gradually died down, and RISC camp has been losing ground RISC的优越论也逐渐偃旗息鼓，RISC阵营节节败退
  - Intel's server CPU accounts for 95% of the market share Intel的服务器CPU占据了95%的市场份额
  - Focus on the design and implementation of micro structures and physics, and explore the possibilities buried in operating systems, compilers and upper applications 当前的焦点：关注微结构与物理设计实现，并发掘操作系统、编译器与上层应用当中埋藏的（提升性能的）可能性

Summary -1 小结1

- The number of available registers greatly influenced the instruction set architecture (ISA) 可用寄存器的数量极大地影响了指令集体系结构（ISA）
- Complex Instruction Set Computers were very complex CISC确实很复杂
  - Necessary to reduce the number of instructions required to fit a program into memory 减少将程序装入内存所需的指令数
  - However, also greatly increased the complexity of the ISA as well 然而，这也大大增加了ISA的复杂性

Summary -2 小结2

- CISC was necessary
  - The processor of X86 architecture dominates the server and desktop market X86架构的处理器在服务器和桌面市场占据了统治地位
  - X86 draws many advantages from RISC X86从RISC中吸取了很多优点
- RISC is still widely concerned and applied RISC仍在得到广泛的关注和应用
  - ARM occupies a major share of the embedded market ARM占据了嵌入式市场的主要份额
  - Mobile phones, tablets PC and various sensors in daily life mostly adopt ARM architecture 日常生活中的手机，平板电脑，各种传感器，大多采用ARM架构
  - ARM borrows a bit from both RISC and CISC ARM从RISC和CISC中借用了很多特性

Outline

- Register and instruction architecture 寄存器和指令集
- Reduced Instruction Set Architecture 精简指令集架构
- The Use of a Large Register File 大寄存器组的使用
- Compiler-Based Register Optimization 基于编译器的寄存器优化
- RISC Pipelining RISC的流水线
- RISC Versus CISC Controversy RISC和CISC的争议

The use of register 寄存器的使用

- Target：Keep frequently accessed operands in registers目标：将频繁访问的操作数保留在寄存器中
- Software solution 软件方法
  - Require compiler to allocate registers 编译器分配寄存器
  - Allocate based on most used variables in a given time 基于最多使用原则分配寄存器
  - Requires sophisticated program analysis 要求精细的程序分析
- Hardware solution 硬件方法
  - Have more registers 更多的寄存器
  - Thus more variables will be in registers 更多的变量在寄存器中

How and problem?

- Limited number of registers requires reasonable use, and the locality principle provides the possibility 有限的寄存器数量要求合理的使用，局部性原理提供了可能性
  - Store local scalar variables in registers 寄存器中保存局部标量变量
  - Reduces memory access 减少内存访问
- Every procedure (function) call changes locality 每个过程或函数调用改变了局部性
  - Parameters must be passed 参数需要传递
  - Results must be returned 结果需要返回
  - Variables from calling programs must be restored 调用程序的变量需要恢复

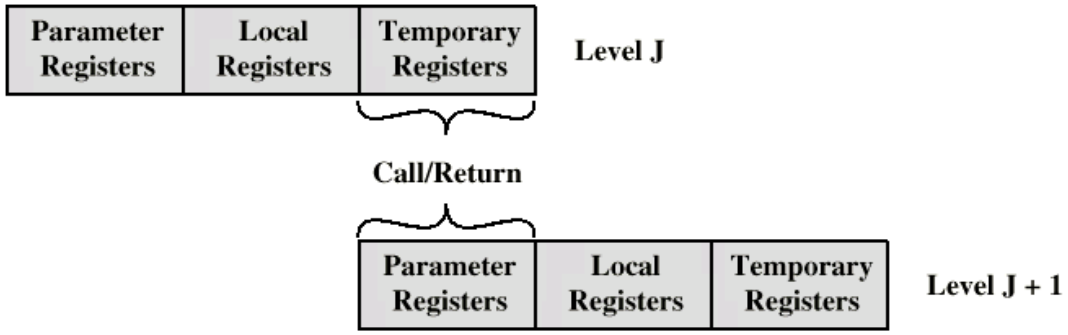
Register windows – 1 寄存器窗口1

- Characteristics of procedure call 过程调用的特点
  - Only few parameters 只有少量的参数
  - Limited range of depth of call 有限的调用深度
- Divide the available registers into several small registers set 将可用寄存器划分为若干个小的寄存器组
  - Calls switch to a different set of registers 调用的时候切换到不同的寄存器组
  - Returns switch back to a previously used set of registers 返回的时候，切换回之前使用的寄存器组
- Set of registers called register windows 寄存器组称为寄存器窗口

Register windows – 2 寄存器窗口2

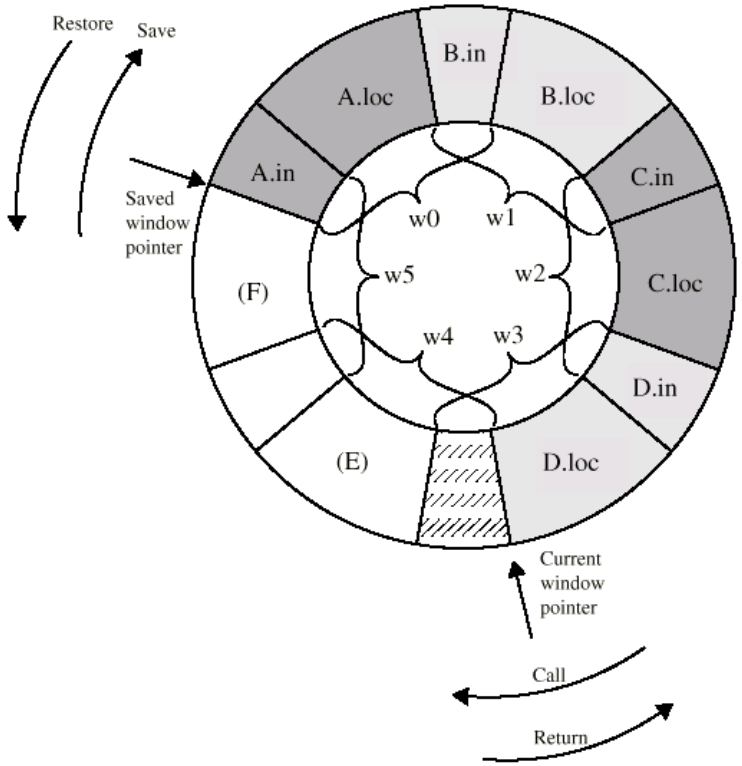
- Three areas within a register set 每个寄存器组包含三个区域
  - Parameter registers 参数寄存器
  - Local registers 局部变量寄存器
  - Temporary registers 临时变量寄存器
- Temporary registers from one set overlap parameter registers from the next 临时变量寄存器和下一级过程的参数寄存器重叠
  - This allows parameter passing without moving data 允许不需要移动数据，就可以完成参数传递
- At any time, only one register window is visible 任何时刻，只有一个寄存器窗口可见

Overlapping register windows 寄存器窗口重叠



- 本级的临时变量寄存器和下一级的参数寄存器在物理上是同一个，在传递参数时，不需要移动数据
- 程序中过程的调用和返回的数量不确定，所以寄存器窗口应该足够多，以保证所有的过程调用都能用到
- 由于寄存器的数量有限，只能保证少数最近的过程能够使用寄存器。更早的过程调用还是需要保存到存储器中。当嵌套深度减少的时候，再将数据从存储器恢复到寄存器中
- 这种方式称为环形缓冲窗口

Circular buffer diagram 环形缓冲图



CWP，current window point，当前窗口指针

SWP，saved window pointer，最近保存窗口指针。

- 寄存器窗口以一种部分重叠的形式形成一个环形。当环形寄存器窗口都充满了后，再有过程调用，把最早的寄存器窗口保存到存储器
- 调用时，移动当前窗口指针以显示当前活动的寄存器窗口
- 如果所有窗口都在使用中，将生成一个中断，并将最早的窗口（调用嵌套中最远的窗口）保存到内存中
- 保存的窗口指针标识最近保存在内存中的窗口
- 当过程返回的时候，CWP会回退一个。当CWP回退到和SWP一样的时候，就会引起一个中断，导致保存到存储器中的寄存器窗口恢复
- 嵌套层数不会太深，所以一般不会保存到存储器中

Global variables 全局变量

- Allocated by the compiler to memory 由编译器将全局变量分配到内存中
  - Inefficient for frequently accessed variables 对于频繁访问的变量这样不够
  - Frequent access to memory, low efficiency 频繁访问存储器，效率较低
- A set of registers for global variables 有一组寄存器给全局变量
  - compiler determines which global variables can be placed in global registers 编译器决定哪些全局变量可以放到全局寄存器中
  - Replacement is also determined by the compiler 替换也是由编译器来决定

Registers v cache -1 寄存器和cache的比较1

- Cache:
  - Inserting cache between processor and memory can solve the problem of speed difference 在处理器和存储器之间插入高速缓存，解决速度差异问题
- Register:
  - organized in the form of windows, which is similar to a small fast buffer. It stores a subset of all variables that may be used many times 组织成窗口的形式，作用跟一个小的快速的缓冲器很类似， 存放了可能多次使用的所有变量的子集

Registers v cache -2 寄存器和cache的比较2

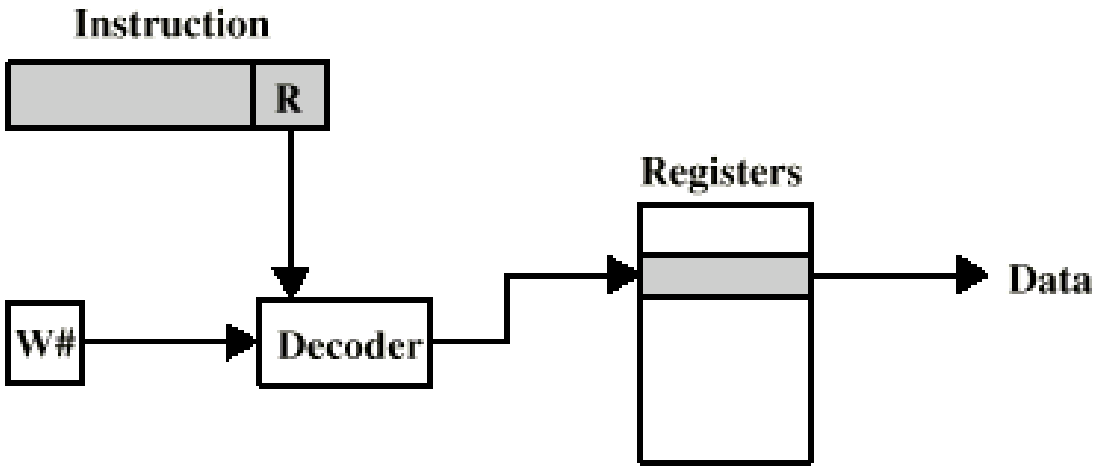
Large Register File	Cache
All local scalars	Recently used local scalars
Individual variables	Blocks of memory
Compiler assigned global variables	Recently used global variables
Save/restore based on procedure nesting depth	Save/restore based on cache replacement algorithm
Register addressing	Memory addressing

- 寄存器组中保存的是所有局部标量变量。cache保存的是最近使用过的标量变量
- 寄存器组中保存的是个别的变量。cache中保存的是内存中的一个块
- 寄存器组方案中，需要编译器来决定全局变量的保存。cache中则是根据最近使用原则进行管理
- 寄存器的数据保存或者恢复，依赖的是过程调用嵌套的深度。cache根据替换算法进行替换
- 寄存器组采用的是寄存器寻址。cache采用的是内存寻址

Registers v cache -3 寄存器和cache的比较3

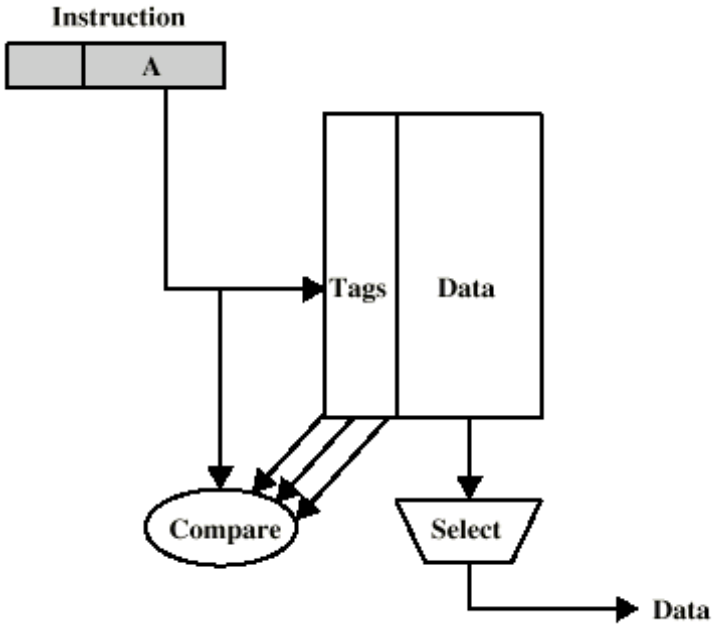
- The register saves time because all local scalar variables are retained 寄存器文件节省了时间，因为保留了所有局部标量变量
- Not efficient use of space, because not all procedures will need the full window space allocated to them 没有有效地利用空间，因为并非所有过程都需要分配给它们的完整窗口空间
- The cache may make more efficient use of space because it stores necessary data dynamically Cache可以更有效地利用空间，因为它动态存储必要的数据

Register access 寄存器访问



- 要访问基于窗口的寄存器组中的一个标量变量，需要给出窗口号和一个寄存器号
- 通过一个相对简单的译码器，就可以得到对应的寄存器，读出这个数据

Cache access cache访问



- 对于cache访问，需要生成一个完整的地址，操作的复杂度和寻址方式有关
- 进行对比，看数据是否命中
- 如果命中，就可以读取数据
- 如果没有命中，那么就需要先替换cache行，然后才能得到数据

Summary

- Although RISC uses a large number of registers, the number is still limited, and effective utilization of these registers is the key to design 尽管RISC采用了较多的寄存器，但是数量仍然有限，有效利用这些寄存器是设计的关键
- The principle of program locality provides the possibility for effective utilization of registers, where commonly used scalars are placed in registers 程序局部性原理为寄存器的有效利用提供了可能，常用的标量放在寄存器中
- For procedure calls, register windows can be used 对于过程调用，可以采用寄存器窗口的方式
- Global variables can be saved in memory for use, or several registers can be allocated specifically for global variables 全局变量可以将其保存在内存中进行使用，也可以分配若干个寄存器给全局变量专用

Outline

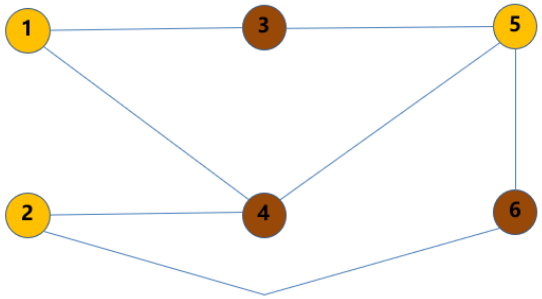
- Register and instruction architecture 寄存器和指令集
- Reduced Instruction Set Architecture 精简指令集架构
- The Use of a Large Register File 大寄存器组的使用
- Compiler-Based Register Optimization 基于编译器的寄存器优化
- RISC Pipelining RISC的流水线
- RISC Versus CISC Controversy RISC和CISC的争议

How?

- HLL programs have no explicit references to registers 高级语言不会显式引用寄存器
- Optimizing use is up to compiler 优化由编译器负责
  - Assign symbolic or virtual register to each candidate variable 为每个候选变量分配符号寄存器或虚拟寄存器
  - Map symbolic registers to real registers 映射符号寄存器到实际寄存器
  - Symbolic registers that do not overlap can share real registers 不重叠的符号寄存器可以共享实体寄存器
  - If you run out of real registers，some variables use memory 如果没有实体寄存器，变量只能用内存
- The essence is to judge which data needs to be put in the register at any time 本质是判断任意时刻，哪些数据需要放到寄存器中

Graph coloring 图着色问题

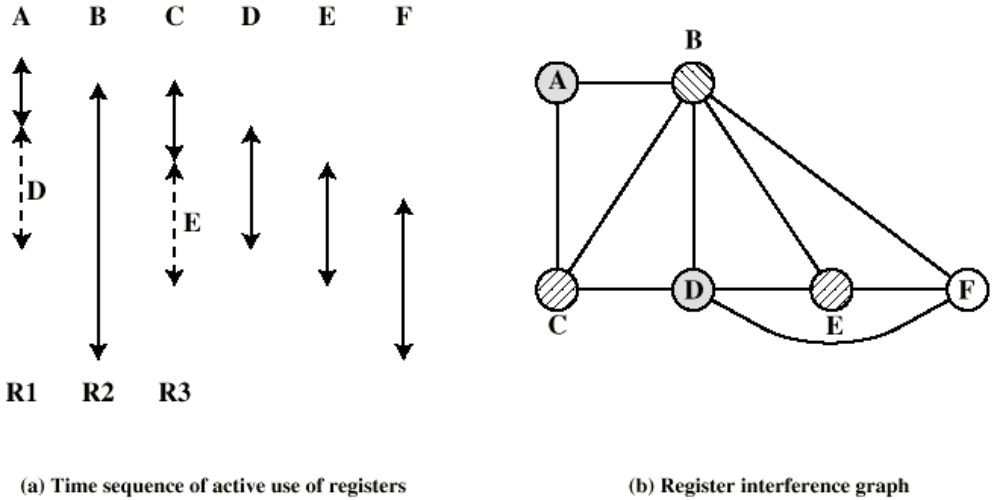
- Symbol registers is more than register 符号寄存器比寄存器数量多
- determine which symbol registers can use the actual registers 需要确定哪些符号寄存器可以使用实际的寄存器
- Using Graph Coloring of topology 采用图着色的拓扑学方法
  - Given a graph of nodes and edges 给定一个包含节点和边的图
  - Assign a colour to each node 给每个节点分配一个颜色
  - Adjacent nodes have different colours 相邻节点用不同的颜色
  - Use minimum number of colours 最少颜色数量



Graph coloring 图着色问题

- Nodes are symbolic registers 节点就是符号寄存器
- Two registers that are live in the same program fragment are joined by an edge 两个寄存器存在于同一个程序段，用一个边来连接
- Try to colour the graph with *n* colours, where *n* is the number of real registers 用n种颜色去着色，n是实体寄存器的数量
- Nodes that can not be coloured are placed in memory 不能被着色的节点需要放到内存中

Graph colouring approach 图着色方法



(a) Time sequence of active use of registers

(b) Register interference graph

- 构造无向图：A和BC在时间上有重叠，A和BC有连线；B和所有的节点都有时间重叠，B和所有的节点都有连线。C和ABD有时间上的重叠，C和ABD有连线
- 从A开始，给A赋一个灰色，B和C必须要和A不一样，并且B和C也不能一样，给B附一个顺斜杠，C赋一个反斜杠。D节点和BC相连，和A不相连，D点可以用灰色。E节点和BD相连，E可以用C的颜色。F和BED相连，而BDE分别是正斜杠、灰色和反斜杠，F必须要用到第四个颜色。
- 如果物理寄存器只有3个的话，那么F就需要保存到存储器中了，通过加载和保存来处理

Large Register vs Compiler 大寄存器vs 编译器

- When the number of registers is small, the effect will be better by optimizing the registers 寄存器数目较少的时候，通过将寄存器进行优化，达到的效果将会更好
- When the number of registers is large, the effect of register optimization will not be very good 寄存器数量较多时，采用寄存器优化的效果不会很好
- Optimization of registers is mainly for the case of a small number of registers 寄存器的优化，主要是针对寄存器数量较少的情况

Summary

- The compiler is responsible for the use of registers 编译器负责寄存器的使用
- Assign a variable to a virtual register and map it to the actual register 将变量分配一个虚拟寄存器，然后和实际的寄存器进行映射
- Mapping using graph coloring algorithms in graph theory 映射采用图论中的图着色算法
- When the number of registers is small, using register optimization has a better effect 寄存器数量较少时，采用寄存器优化的效果较好

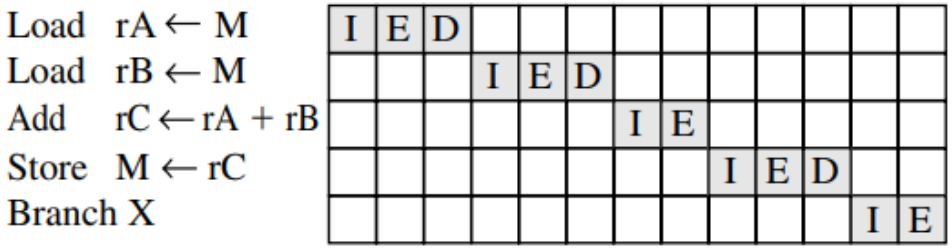
Outline

- Register and instruction architecture 寄存器和指令集
- Reduced Instruction Set Architecture 精简指令集架构
- The Use of a Large Register File 大寄存器组的使用
- Compiler-Based Register Optimization 基于编译器的寄存器优化
- RISC Pipelining RISC的流水线
- RISC Versus CISC Controversy RISC和CISC的争议

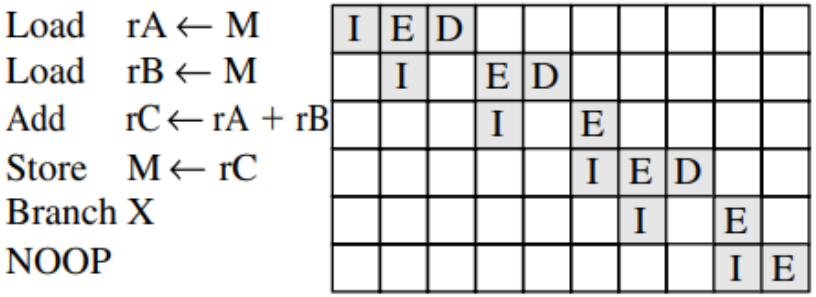
RISC pipelining RISC流水线

- Most instructions are register to register 大部分指令是寄存器到寄存器
- Two phases of execution 两阶段执行
  - I: Instruction fetch I：取指
  - E: Execute E：执行
    - ALU operation with register input and output ALU操作寄存器输入输出
- For load and store 对于加载和保存
  - I: Instruction fetch I：取指
  - E: Execute: Calculate memory address E：执行，计算存储器地址
  - D: Register to memory or memory to register operation D：存储，寄存器到内存或内存到寄存器的操作

Effects of pipelining -1 流水线的效率1



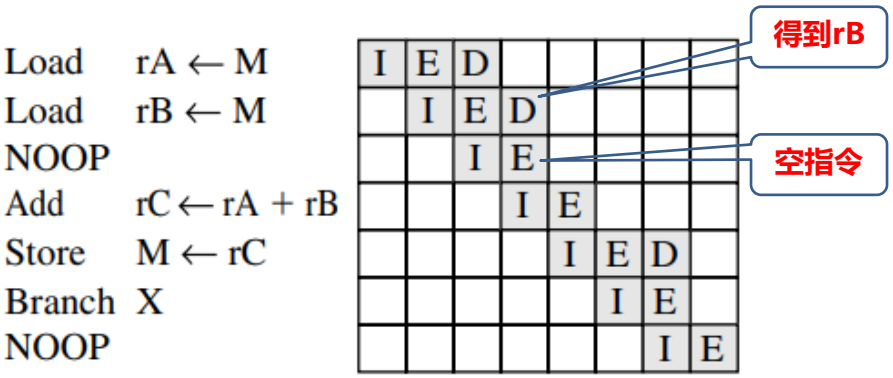
(a) Sequential execution



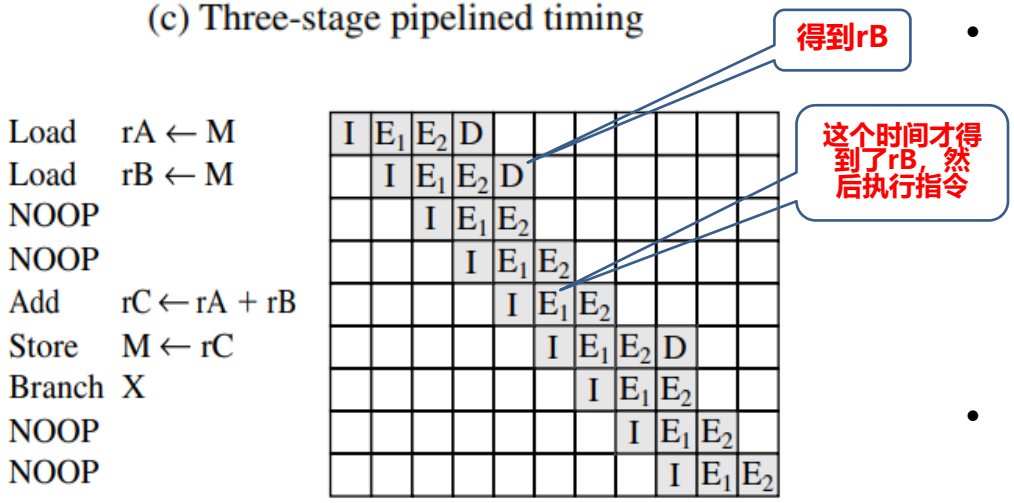
(b) Two-stage pipelined timing

- 图a是没有采用流水线技术，完全按照顺序来执行
- 图b是采用两阶段流水线的执行情况。由于同时只能有1个存储器访问，取指和存储会冲突，导致取指会延后一个时钟周期
- 存储器的访问限制导致了时钟周期的浪费

Effects of pipelining -2 流水线的效率2



(c) Three-stage pipelined timing



(d) Four-stage pipelined timing

- 如果存储支持2个访问，可以用三阶段流水线
- 指令相关性：Add rC<—rA+rB，指令需要的操作数为rA和rB。而第二条指令要到第四个时钟周期才能得到rB。所以要插入一个空指令NOOP
- 指令执行阶段，通常涉及到寄存器的读和ALU的操作，把E阶段进一步分为E1和E2。其中E1阶段完成寄存器的读，而E2阶段则完成ALU操作和寄存器的写操作
- 使用四阶段流水线来提高效率。但同样需要考虑相关性问题

Optimization of Pipelining 流水线的优化

- Dependency of data and branch will disrupt the pipeline and affect the efficiency 数据相关性以及分支指令，破坏流水线，影响效率
- Two methods：Delayed branch, Loop Unrolling 两种方法：延迟分支、循环展开
- Delayed branch 延迟分支
  - Branch instruction affects only the instructions that follow it 分支指令只对它后面的指令有影响
  - This following instruction is the delay slot 后面的指令就是延迟槽
  - Arrange a useful instruction to replace the NOOP instruction 安排一个有用的指令代替NOOP指令

RISC – delayed branch (1) 延迟分支1

- Calculate result of branch before unusable instructions pre-fetched 在不需要的指令预取之前计算分支的影响
  - Instructions that are not affected by branches are immediately followed by branch 在分支指令之后紧跟不受分支影响的指令
  - Keeps pipeline full while fetching new instruction stream 获取新指令时保持流水线充满状态
- Not as good for superscalar 对超标量没有效果
  - Multiple instructions need to execute in delay slot 多个延迟槽
  - Instruction dependence problems 指令相关性问题
  - Often use branch prediction 经常使用分支预测

RISC – delayed branch (2) 延迟分支2

- Problem: How do you find instructions to fill the delay slots?

  问题：如何找到指令去填充延迟槽

  - Branch must be independent of delay slot instructions 分支需要和延迟槽执行独立
  - Unconditional branch: Easier to find instructions to fill the delay slot 无条件分支：容易找到
  - Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot 条件分支：条件计算不能依赖于分支槽的指令→不好填充延迟槽

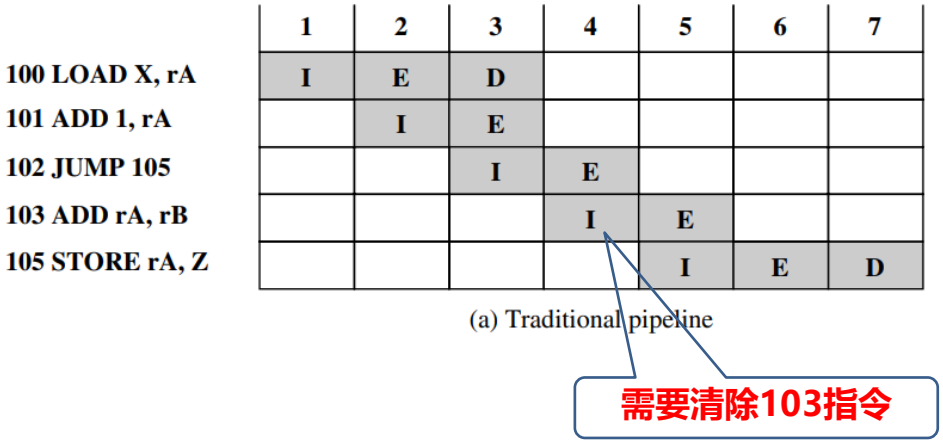
Normal and delayed branch 正常和延迟分支

Address	Normal Branch	Normalized Pipeline	Optimized Delayed Branch
100	LOAD X, rA	LOAD X, rA	LOAD X, rA
101	ADD 1, rA	ADD 1, rA	JUMP 105
102	JUMP 105	JUMP 106	ADD 1, rA
103	ADD rA, rB	NOOP	ADD rA, rB
104	SUB rC, rB	ADD rA, rB	SUB rC, rB
105	STORE rA, Z	SUB rC, rB	STORE rA, Z
106		STORE rA, Z	

- Normal Branch这一列是正常的指令流程
- Normalized pipeline 是规范化之后的指令流，不需要清空流水线而增加了NOOP指令
- Optimized Delayed Branch是优化的延迟分支的指令流

Normal branch pipeline 正常流水线

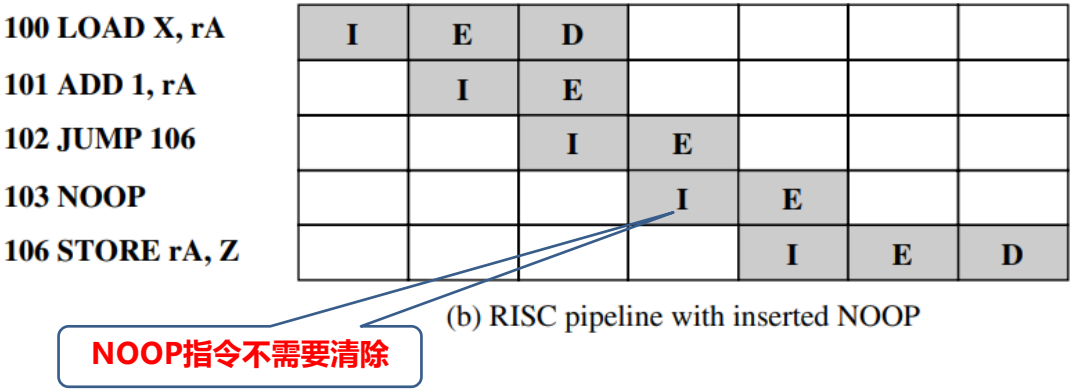
Address	Normal Branch
100	LOAD X, rA
101	ADD 1, rA
102	JUMP 105
103	ADD rA, rB
104	SUB rC, rB
105	STORE rA, Z
106	



- 102为跳转指令
- 102执行跳转的时候，103已经完成了取指，此时需要把103清除掉，重新取指105
- 清空流水线需要专门的电路，增加了电路复杂度

Normalized branch pipeline 规范化流水线

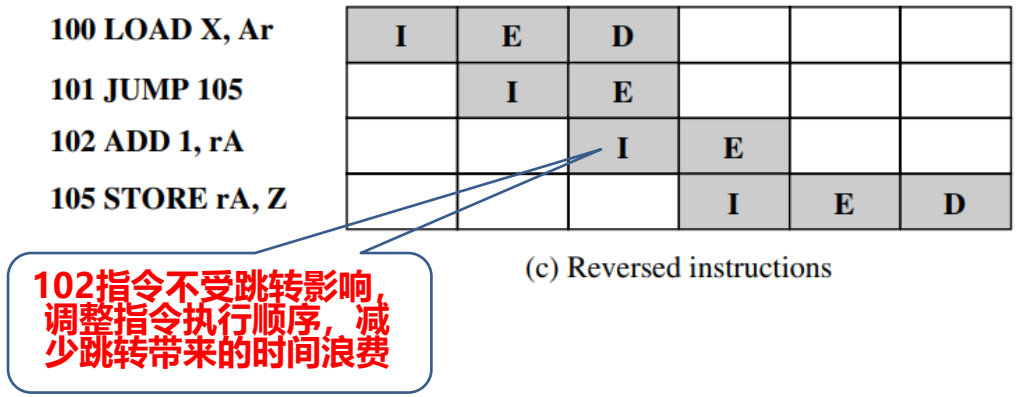
Address	Normalized Pipeline
100	LOADX, rA
101	ADD 1, rA
102	JUMP 106
103	NOOP
104	ADD rA, rB
105	SUB rC, rB
106	STORE rA, Z



- 在102之后增加了一个NOOP指令
- 102跳转的时候，直接跳转到106即可，因为103指令没有做任何操作，所以不需要清空流水线
- 存在一个时钟周期的浪费，但是省去了清空流水线的工作

Optimized Delayed Branch 优化的延迟分支

Address	Optimized Delayed Branch
100	LOAD X, rA
101	JUMP 105
102	ADD 1, rA
103	ADD rA, rB
104	SUB rC, rB
105	STORE rA, Z
106	



(c) Reversed instructions

- 编译器发现，JUMP之前的指令，ADD 1，rA，JUMP没有直接的关联，如果把它和JUMP进行对调，这样就JUMP后面不需要插入NOOP指令
- JUMP指令之后，不需要清空流水线，继续执行完ADD 1，rA指令
- 通过指令顺序的调整，节约了时间

Comparison of three pipelines 三种方法的比较

Time →

	1	2	3	4	5	6	7
100 LOAD X, rA	I	E	D				
101 ADD 1, rA		I	E				
102 JUMP 105			I	E			
103 ADD rA, rB				I	E		
105 STORE rA, Z					I	E	D

(a) Traditional pipeline

100 LOAD X, rA	I	E	D				
101 ADD 1, rA		I	E				
102 JUMP 106			I	E			
103 NOOP				I	E		
106 STORE rA, Z					I	E	D

(b) RISC pipeline with inserted NOOP

100 LOAD X, Ar	I	E	D			
101 JUMP 105		I	E			
102 ADD 1, rA			I	E		
105 STORE rA, Z				I	E	D

(c) Reversed instructions

- 从时间上来看，第二种方法并没有节约时间，省去了清空流水线的操作
- 而采用延迟分支之后，节省了一个时钟周期

Advantage 优点

- Keeps the pipeline full with useful instructions in a simple way assuming 以一种简单的方式让流水线充满有用的指令
  - Number of delay slots = number of instructions to keep the pipeline full before the branch resolves 延迟插槽数=分支解析前保持管道已满的指令数
  - All delay slots can be filled with useful instructions 所有延迟槽都可以填充有用的指令

Disadvantage 缺点

- Not easy to fill the delay slots (even with a 2-stage pipeline) 不容易填充延迟槽（即使是两阶段流水线）
  - Number of delay slots increases with pipeline depth, superscalar execution width 延迟槽的数量随着流水线深度、超标量执行宽度的增加而增加
  - Number of delay slots should be variable with variable latency operations 延迟插槽的数量应随延迟操作的变化而变化

Another method-Loop Unrolling 循环展开

- Replicate body of loop a number of times 多次复制循环体
  - Iterate loop fewer times 循环迭代次数更少
  - Reduces loop overhead 减少循环开销
  - Increases instruction parallelism 提高指令并行性
- During the execution of the loop body, because of the locality principle, some data can be used for many times, which can reduce the number of times to access the 循环体执行中，因为局部性原理，有些数据可以多次利用，这样可以减少访问存储器的次数

Example

```
do i=2, n-1
    a[i] = a[i] + a[i-1] * a[i+1]
end do
```

Becomes

```
do i=2, n-2, 2
    a[i] = a[i] + a[i-1] * a[i+1]
    a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2,2) = i) then
    a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

- 原始的循环是要做一个一维数组的变化。循环体中有一个语句需要执行
- 把这个循环体进行拆解，变成了2个语句，一个循环体执行了2个迭代操作
- 2个指令可以并行进行。原来每次循环需要访问3次存储器，现在相当于2次循环只需要访问4次存储器，降低了存储器访问的次数

Summary

- Optimized pipeline design is the core concept of RISC 优化的流水线设计是RISC的核心理念
- Most instructions are register to register, except for load and save instructions 大部分指令都是在寄存器-寄存器操作，除了加载和保存指令
- Regular instructions and simple pipeline design 指令规整，流水线设计简单
- For dependency of data and branch, using delayed branching or loop unrolling to improve pipeline efficiency 对于数据相关性或分支指令，采用延迟分支或循环展开提高流水线效率

Outline

- Register and instruction architecture 寄存器和指令集
- Reduced Instruction Set Architecture 精简指令集架构
- The Use of a Large Register File 大寄存器组的使用
- Compiler-Based Register Optimization 基于编译器的寄存器优化
- RISC Pipelining RISC的流水线
- RISC Versus CISC Controversy RISC和CISC的争议

RISC v CISC

- Not clear cut 没有明确的界限
  - RISC designs may benefit from the inclusion of some CISC features and that RISC设计可能会包含一些CISC功能
  - CISC designs may benefit from the inclusion of some RISC features CISC设计可能受益于一些RISC功能
- Many designs borrow from both philosophies 许多设计借鉴了这两种理念
  - PowerPC are no longer “pure” RISC PowerPC不再是“纯”RISC
  - Pentium II and later Pentium models do incorporate some RISC characteristics 奔腾II和后期奔腾模型确实包含了一些RISC特性

Characteristics of Some Processors 一些处理器的特征

Processor	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic	Max number of memory operands	Unaligned addressing allowed	Max Number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
AMD29000	1	4	1	no	no	1	no	1	8	3[a]
MIPS R2000	1	4	1	no	no	1	no	1	5	4
SPARC	1	4	2	no	no	1	no	1	5	4
MC88000	1	4	3	no	no	1	no	1	5	4
HP PA	1	4	10[a]	no	no	1	no	1	5	4
IBM RT/PC	2[a]	4	1	no	no	1	no	1	4[a]	3[a]
IBM RS/6000	1	4	4	no	no	1	yes	1	5	5
Intel i860	1	4	4	no	no	1	no	1	5	4
IBM 3090	4	8	2[b]	no[b]	yes	2	yes	4	4	2
Intel 80486	12	12	15	no[b]	yes	2	yes	4	3	3
NSC 32016	21	21	23	yes	yes	2	yes	4	3	3
MC68040	11	22	44	yes	yes	2	yes	8	4	3
VAX	56	56	22	yes	yes	6	yes	24	4	0
Clipper	4[a]	8[a]	9[a]	no	no	1	0	2	4[a]	3[a]
Intel 80960	2[a]	8[a]	9[a]	no	no	1	yes[a]	—	5	3[a]

A: RISC that does not conform to this characteristic
B: CISC that does not conform to this characteristic.

Controversy -1 论战1

- Quantitative 定量
  - compare program sizes and execution speeds 比较程序的大小和执行速度
- Qualitative 定性
  - examine issues of high level language support and use of VLSI real estate 考察对高级语言的支持程度以及VLSI资源的优化使用

Controversy -2 论战2

- Problems 问题
  - No pair of RISC and CISC that are directly comparable 没有一对RISC和CISC能够直接比较
  - No definitive set of test programs 没有确定的测试程序集
  - Difficult to separate hardware effects from complier effects 难以区分硬件效果和编译器效果
  - Most comparisons done on "toy" rather than production machines 大多数比较是在"玩具"（试验机）而不是生产机器上进行
  - Most commercial devices are a mixture 大多数商用设备都是混合的

Controversy -3 论战3

- Integrated with each other 相互融合
  - Intel x86 series：integrates the characteristics of RISC，such as increasing the number of registers and emphasizing instruction pipeline technology Intel x86系列，融合了RISC的特点增加寄存器数量，强调指令流水线技术
  - ARM series：Become more and more complex ARM系列芯片，也变得越来越复杂

Key Terms

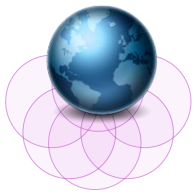
Complex instruction set computer (CISC)	delayed load	Reduced instruction set computer (RISC)	Register window
delayed branch	High-Level language (HLL)	Register file	

Summary and Question

- 小结
  - 对指令集的发展进行了介绍，并对CISC的问题进行了剖析
  - RISC架构和特点
  - RISC流水线以及存在的问题和解决方法
- 问题
  - 简述CISC的设计理念，以及设计的效果
  - RISC设计的核心思想
  - 延迟分支的概念

Assignments

- Review Questions
  - 15.1~ 15.5



Thank You!