# 计算机体系结构

## Topic II: ISA

# Fundamental Considerations in Designing an Instruction Set Architecture

# 指令集架构 (ISA)

- **软件子系统与硬件子系统的关键界面**
- **一组直接由硬件执行的指令，包括**
  - 程序员可见的机器状态
  - 程序员可见的指令集合(操作机器状态的指令)

- **应具备的特性**
  - 成本
  - 简洁性
  - 架构和具体实现分离：可持续多代，以保持向后（backward) 兼容
  - 可扩展空间：可用于不同应用领域（desktops, servers, embedded applications）
  - 易于编程/编译/链接：为高层软件的设计与开发提供方便的功能
  - 性能：方便低层硬件子系统高效实现
- **IBM 360 是第一个将ISA与其实现分离的系列机**
  - today you can buy AMD or Intel processors that run the x86-64 ISA
  - many cellphones use the ARM ISA with implementations from many different companies including TI, Qualcomm, Samsung, Marvell, etc

# 用户级ISA和特权级ISA

- **重要的系统界面（System Interface）**
  - ISA界面（Instruction Set Architecture）
  - ABI界面（Application Binary Interface）
- **ISA：用户级ISA+特权级ISA**
  - 用户级ISA 适用于操作系统和应用程序
  - 特权级ISA 适用于硬件资源的管理（操作系统）比如hardware thread. A key feature of RISC-V

# ISA的实现

- **ISA 通常设计时会考虑特定的微体系结构（实现）方式**
  - Accumulator ⇒ hardwired, unpipelined
  - CISC ⇒ microcoded （微程序）
  - RISC ⇒ hardwired, pipelined（硬布线、流水线）
  - VLIW ⇒ fixed-latency in-order parallel pipelines (固定延时、顺序执行、多条流水线并行）
  - JVM ⇒ software interpretation（软件解释）

- **ISA 理论上可以用任何微体系结构（实现）方式**
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Simics: Software-interpreted SPARC RISC machine
  - ARM Jazelle: A hardware JVM processor
  - RISC-V

# ISA 的演进

几个操作数？其存储位置

C=A+B

| Accumulator | Stack | Register-Memory | Register-Register |
|---|---|---|---|
| Load A | Push A | Load R1, A | Load R1, A |
| Add B | Push B | Add R1, B | Load R2, B |
| Store C | Add | Store R1, C | Add R3, R1, R2 |
| | Pop C | | Store R3, C |

# ISA必须specify哪些东西?

- **指令格式或编码方式。即如何编码?**
- **操作数和操作结果的存放位置**
  - 存放位置?
  - 多少个显式操作数?
  - 存储器操作数如何定位?
  - 哪些操作数可以或不可以放到存储器中?
  - 寻址方式
- **数据类型和大小**
- **支持哪些操作**
- **下一条指令地址**
  - jumps, conditions, branches
  - fetch-decode-execute is implicit!

# 有关ISA的若干问题

- **存储器寻址**
- 操作数的类型与大小
- 所支持的操作
- 控制转移类指令
- 指令格式

# Processor – Memory Interconnections

- Memory is viewed as a large, single-dimension array, with an address
- A memory address is an index into the array

# 存储器寻址

- **80年以来几乎所有机器的存储器都是按字节编址**
- **一个存储器地址可以访问：**
  - 一个字节、2个字节、4个字节、更多字节.....
- **不同体系结构对字word的定义是不同的**
  - 16位字（Intel X86）32位字（MIPS）
- **如何读32位字，两种方案**
  - 每次一个字节，四次完成；每次一个字，一次完成
- **问题：**

  （1）如何将字节地址映射到字地址 (尾端问题 endian)

  （2）一个字是否可以存放在任何字节边界上(对齐问题 alignment)

# Addressing Objects: Endianess and Alignment

- Big Endian: leftmost byte is word address
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- Little Endian: rightmost byte is word address
  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

# Example of Loading and Storing Bytes

- Given following code sequence and memory state (contents are given in hexidecimal), what is the state of the memory after executing the code?

```
add   $s3, $zero, $zero
lb    $t0, 1($s3)
sb    $t0, 6($s3)
```

| Memory | |
|---|---|
| 0 0 0 0 0 0 0 0 | 24 |
| 0 0 0 0 0 0 0 0 | 20 |
| 0 0 0 0 0 0 0 0 | 16 |
| 1 0 0 0 0 0 1 0 | 12 |
| 0 1 0 0 0 4 0 2 | 8 |
| F F F F F F F F | 4 |
| 0 0 9 0 1 2 A 0 | 0 |
| Data | Word Address (Decimal) |

- What value is left in $t0?
- What if the machine was little Endian?

# 尾端问题 Endian

- **little endian, big endian, 在一个字内部的字节顺序问题**

1. Little Endian byte ordering

   - ✧ Memory address X = address of least-significant byte (Intel x86)

2. Big Endian byte ordering

   - ✧ Memory address X = address of most-significant byte (SPARC)

- **如地址xxx00指定了一个字（int），存储器中从xxx00处连续存放ffff0000, 则有两种方式：**
  - Little endian 方式下xxx00位置是字的最低字节, 四个字节的内容分别为00 00 ff ff, Intel 80x86, DEC Vax, DEC Alpha (Windows NT)
  - Big endian 方式下xxx00位置是字的最高字节，四个字节的内容分别为ff ff 00 00, IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

# Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory

  memory: $2^{32}$ bytes = $2^{30}$ words

- Therefore, the memory address of a word must be a multiple of 4 (alignment restriction)

**Alignment restriction: requires that objects fall on address that is multiple of their size.**

# Recap: 对齐问题

- 对s字节的对象访问地址为A，如果A mod s =0 称为边界对齐。
- 边界对齐的原因是存储器本身读写的要求，存储器本身读写通常就是边界对齐的，对于不是边界对齐的对象的访问可能要导致存储器的两次访问，然后再拼接出所需要的数。（或发生异常）

| Address mod 8 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Byte | Aligned | Aligned | Aligned | Aligned | Aligned | Aligned | Aligned | Aligned |
| 2 Bytes | Aligned | | Aligned | | Aligned | | Aligned | |
| 2 Bytes | | Misaligned | | Misaligned | | Misaligned | | Misalign |
| 4 Bytes | Aligned | | | | Aligned | | | |
| 4 Bytes | | Misaligned | | | | Misaligned | | |
| 4 Bytes | | | Misaligned | | | | Misaligned | |
| 4 Bytes | | | | Misaligned | | | | Misalign |
| 8 Bytes | Aligned | | | | | | | |
| 8 Bytes | | Misaligned | | | | | | |

# Recap: 寻址方式

- **寻址方式：如何说明要访问的对象地址**
- **有效地址：由寻址方式说明的某一存储单元的实际存储器地址。有效地址 vs. 物理地址**

| Mode | Example | Meaning | When used |
|---|---|---|---|
| Register | Add R1, R2 | R1 ← R1 + R2 | Values in registers |
| Immediate | Add R1, 100 | R1 ← R1 + 100 | For constants |
| Register Indirect | Add R1, (R2) | R1 ← R1 + Mem(R2) | R2 contains address |
| Displacement | Add R1, (R2+16) | R1 ← R1 + Mem(R2+16) | Address local variables |
| Absolute | Add R1, (1000) | R1 ← R1 + Mem(1000) | Address static data |
| Indexed | Add R1, (R2+R3) | R1 ← R1 + Mem(R2+R3) | R2=base, R3=index |
| Scaled Index | Add R1, (R2+s*R3) | R1 ← R1 + Mem(R2 + s*R3) | s = scale factor = 2, 4, or 8 |
| Post-increment | Add R1, (R2)+ | R1 ← R1 + Mem(R2)<br>R2 ← R2 + s | Stepping through array<br>s = element size |
| Pre-decrement | Add R1, -(R2) | R2 ← R2 – s<br>R1 ← R1 + Mem(R2) | Stepping through array<br>s = element size |

# MIPS寻址方式图示

1. Register addressing

2. Base addressing

3. Immediate addressing

4. PC-relative addressing

5. Pseudo-direct addressing

# Naming Conventions for Registers

| | | |
|---|---|---|
| 0 | $zero | constant 0 |
| 1 | $at | reserved for assembler |
| 2 | $v0 | expression evaluation & |
| 3 | $v1 | function results |
| 4 | $a0 | arguments |
| 5 | $a1 | |
| 6 | $a2 | |
| 7 | $a3 | |
| 8 | $t0 | temporary: caller saves |
| . . . | | (callee can clobber) |
| 15 | $t7 | |
| 16 | $s0 | callee saves |
| . . . | | (caller can clobber) |
| 23 | $s7 | |
| 24 | $t8 | temporary (cont'd) |
| 25 | $t9 | |
| 26 | $k0 | reserved for OS kernel |
| 27 | $k1 | |
| 28 | $gp | pointer to global area |
| 29 | $sp | stack pointer |
| 30 | $fp | frame pointer |
| 31 | $ra | return address |

# MIPS寄存器

- **MIPS的寄存器**
  - 32个64位通用寄存器（GPRs）
    - R0，R1，…，R31
    - 也被称为整数寄存器
    - R0的值永远是0
  - 32个64位浮点数寄存器（FPRs）
    - F0，F1，…，F31
    - 用来存放32个单精度浮点数（32位），也可以用来存放32个双精度浮点数（64位）。
    - 存储单精度浮点数（32位）时，只用到FPR的一半，其另一半没用
  - 一些特殊寄存器
    - 它们可以与通用寄存器交换数据。
    - 例如，浮点状态寄存器用来保存有关浮点操作结果的信息。

# 各种寻址方式的使用情况? (忽略寄存器直接寻址)

**三个SPEC89程序在VAX结构上的测试结果：**

**立即寻址，偏移寻址使用较多**

# 偏移寻址

- **主要问题：偏移的范围（偏移量的大小）**

lw $1, ? ($2)

64bit

**Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)**

# 立即数寻址

**Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)**

# 立即数的大小

**The distribution of immediate values. About 20% were negative for CINT2000 and about 30% were negative for CFP2000. These measurements were taken on a Alpha, where the maximum immediate is 16 bits, for the spec cpu2000 programs. A similar measurement on the VAX, which supported 32-bit immediates, showed that about 20% to 25% of immediates were longer than 16 bits.**

# 寻址方式小结

- **重要的寻址方式:**
  - 偏移寻址方式, 立即数寻址方式, 寄存器间址方式
  - SPEC测试表明，使用频度达到 75%--99%
- **偏移字段的大小应该在 12 - 16 bits**
  - 可满足75%-99%的需求
- **立即数字段的大小应该在 8 -16 bits**
  - 可满足50%-80%的需求

# Recap: 有关ISA的若干问题

- 存储器寻址
- **操作数的类型与大小**
- 所支持的操作
- 控制转移类指令
- 指令格式

# 操作数的类型、表示和大小

- **操作数类型和操作数表示是软硬件的主要界面之一。**
- **操作数类型：是面向应用、面向软件系统所处理的各种数据类型。**
  - 整型、浮点型、字符、字符串、向量类型等
  - 类型由操作码确定或数据附加硬件解释的标记，一般采用由操作码确定
  - 数据附加硬件解释的标记，现在已经不采用
- **操作数的表示：操作数在机器中的表示，硬件结构能够识别，指令系统可以直接使用的表示格式**
  - 整型：原码、反码、补码 (2’s complement)
  - 浮点：IEEE 754标准
  - 十进制：BCD码/二进制十进制表示

# 常用操作数类型

- ASCII character = 1 byte (64-bit register can store  8 characters
- Unicode character or Short integer = 2 bytes = 16 bits  ( half word)
- Integer = 4 bytes = 32 bits (word size on many RISC Processors)
- Single-precision float = 4 bytes = 32 bits (word size)
- Long integer = 8 bytes = 64 bits (double word)
- Double-precision float = 8 bytes = 64 bits (double word)
- Extended-precision float = 10 bytes = 80 bits (Intel architecture)
- Quad-precision float = 16 bytes = 128 bits

# 操作数的大小

**基准测试的结论：（1）对单字、双字的数据访问具有较高的频率**
**（2）支持64位双字操作，更具有一般性**

# MIPS的数据表示

- **整数**
  - 字节（8位） 半字（16位）字（32位） 双字（64位）
- **浮点数**
  - 单精度浮点数（32位） 双精度浮点数（64位）

- **字节、半字或者字在装入32/64位寄存器时，用零扩展或者用符号位扩展来填充该寄存器的剩余部分。装入以后，对它们将按照32/64位整数的方式进行运算。**

# Recap: 有关ISA的若干问题

- 存储器寻址
- 操作数的类型与大小
- **所支持的操作**
- 控制转移类指令
- 指令格式

# 典型操作类型

| Operator type | Examples |
|---|---|
| Arithmetic and logical | Integer arithmetic and logical operations: add, subtract, and, or, multiple, divide |
| Data transfer | Loads-stores (move instructions on computers with memory addressing) |
| Control | Branch, jump, procedure call and return, traps |
| System | Operating system call, virtual memory management instructions |
| Floating point | Floating-point operations: add, multiply, divide, compare |
| Decimal | Decimal add, decimal multiply, decimal-to-character conversions |
| String | String move, string compare, string search |
| Graphics | Pixel and vertex operations, compression/decompression operations |

- **一般计算机都支持前三类所有的操作；**
- **不同计算机系统 对系统支持程度不同，但都支持基本的系统功能。**
- **对最后四类操作的支持程度差别也很大，有些机器不支持，有些机器还在此基础上做一些扩展，这些指令有时作为可选的指令。**

# Top 10 80x86 Instructions

- Rank instruction Integer Average Percent total executed

| Rank | instruction | Integer Average Percent total executed |
|---|---|---|
| 1 | load | 22% |
| 2 | conditional branch | 20% |
| 3 | compare | 16% |
| 4 | store | 12% |
| 5 | add | 8% |
| 6 | and | 6% |
| 7 | sub | 5% |
| 8 | move register-register | 4% |
| 9 | call | 1% |
| 10 | return | 1% |
| | Total | 96% |

- Simple instructions dominate instruction frequency

# ISA对操作类型的选择

- **需考虑的因素：速度、价格和灵活性**
- **基本要求：指令系统的完整性、规整性、高效率和兼容性**
  - 完整性设计：具备基本指令种类
  - 兼容性：系列机
  - 高效率：指令执行速度快、使用频度高
  - 规整性
    - 让所有运算部件都能对称、均匀的在所有数据存储单元之间进行操作。
    - 对所有数据存储单元都能同等对待，无论是操作数或运算结果都可以无约束地存放到任意数据存储单元中
  - 正交性
    - 数据类型独立于寻址方式
    - 寻址方式独立于所要完成的操作
- **当前对这一问题的处理有两种截然不同的方向**
  - CISC和RISC

# CISC vs RISC

- **ISA的功能设计**
  - 任务：确定硬件支持哪些操作
  - 方法：统计的方法
  - 两种类型：CISC和RISC
- **CISC（Complex Instruction Set Computer）**
  - 目标：强化指令功能，减少运行的指令条数，提高系统性能
  - 方法：面向目标程序的优化，面向高级语言和编译器的优化
- **RISC（Reduced Instruction Set Computer）**
  - 目标：通过简化指令系统，用高效的方法实现最常用的指令
  - 方法：充分发挥流水线的效率，降低（优化）CPI

# Recap: 有关ISA的若干问题

- 存储器寻址
- 操作数的类型与大小
- 所支持的操作
- **控制转移类指令**
- 指令格式

# 控制流类指令中的寻址方式

- **PC-relative 方式（相对寻址）**
  - 例如：条件转移
- **说明动态的转移地址方式(通过寄存器间接跳转）：**
  - 编译时不知道目标地址，程序执行时动态确定
    - Case or switch statements
    - Virtual function or methods
    - High-order functions or function pointers
    - Dynamically shared libraries
  - 转移地址放到某一寄存器中，通过寄存器间接跳转

# 转移目标地址与当前指令的距离

Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

建议：PC-relative 寻址，偏移地址至少8位

# 控制类指令

- 四种类型的控制流改变：
  - 条件分支( Conditional branch) 、跳转(Jump)、过程调用(Procedure calls)、过程返回(Procedure returns)

Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

# Recap：跳转指令

- **根据跳转(jump)指令确定目标地址的方式不同以及跳转时是否链接，可以把跳转指令分成4种。**
- **确定目标地址的方式**
  - 把指令中的26位偏移量左移2位后，替换程序计数器的低28位。
  - 间接跳转：由指令中指定的一个寄存器来给出转移目标地址。
- **跳转的两种类型**
  - 简单跳转：把目标地址送入程序计数器。
  - 跳转并链接：把目标地址送入程序计数器，把返回地址（即顺序下一条指令的地址）放入寄存器R31。

# 分支比较类型比较

Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

# Recap：分支指令（条件转移）

- **分支条件由指令确定。**
  - 例如：测试某个寄存器的值是否为零
- **提供一组比较指令，用于比较两个寄存器的值。**
  - 例如：“置小于”指令
- **有的分支指令可以直接判断寄存器内容是否为负，或者比较两个寄存器是否相等。**
- **分支的目标地址。**
  - 由16位带符号偏移量左移两位后和PC相加的结果来决定
- **一条浮点条件分支指令：通过测试浮点状态寄存器来决定是否进行分支。**

# Recap: 有关ISA的若干问题

- 存储器寻址
- 操作数的类型与大小
- 所支持的操作
- 控制转移类指令
- **指令格式**

# 指令编码

Variable:

Fixed:

Hybrid:

# 指令格式选择策略

- **如果代码规模最重要，那么使用变长指令格式**
- **如果性能至关重要，使用固定长度指令**
- **有些嵌入式CPU附加可选模式，由每一应用自主选择性能还是代码量**
  - 窄指令支持更少的操作、更短的地址和立即数字段、更少的寄存器以及双地址格式，而不是传统的RISC计算机的三地址格式
  - 例如：RISC-V 的 RV32IC， C表示压缩表示
  - ARM Thumb ， microMIPS
- **有些机器使用边执行边解压的方式**
  - 例如 IBM 的CodePack PowerPC

# MIPS的指令格式

- **寻址方式编码到操作码中**
- **所有指令都是32位的**
- **操作码占6位**
- **3种指令格式**
  - I类
  - R类
  - J类

# I类指令

- 包括所有的load和store指令、立即数指令，分支指令。
- 立即数字段为16位，用于提供立即数或偏移量。

# 具体的I类指令

- **load指令**
  - 访存有效地址：Regs[rs] + immediate
  - 从存储器取来的数据放入寄存器rt
- **store指令**
  - 访存有效地址：Regs[rs] + immediate
  - 要存入存储器的数据放在寄存器rt中
- **立即数指令**
  - Regs[rt] ← Regs[rs] op immediate
- **分支指令**
  - 转移目标地址：PC ← PC+4 + 4*immediate

# R类指令

- **包括ALU指令、专用寄存器读/写指令、寄存器跳转指令、move指令等**
- **ALU指令**
  - Regs[rd]← Regs[rs] funct Regs[rt]
  - func为具体的运算操作编码

| 6 | 5 | 5 | 5 | 5 | 6 |
|---|---|---|---|---|---|
| 操作码 | rs | rt | rd | sham | func |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

# J类指令

- 包括跳转指令、跳转并链接指令、自陷指令、异常返回指令
- 在这类指令中，指令字的低26位是偏移量，它与PC值相加形成跳转的地址

# Recap：MIPS的操作

- MIPS指令可以分为四大类
  - load和store（内存访问）、ALU操作、分支与跳转、浮点操作

# Recap：MIPS（64位）控制类指令

| 指令举例 | 指令名称 | 含义 |
|---|---|---|
| J name | 跳转 | $PC_{36..63} \leftarrow$ name<<2 |
| JAL name | 跳转并链接 | Regs[R31]←PC+4；$PC_{36..63}$←name<<2；<br>$((PC+4)-2^{27}) \leqslant name < ((PC+4)+2^{27})$ |
| JALR R3 | 寄存器跳转并链接 | Regs[R31]←PC+4；PC← Regs[R3] |
| JR R5 | 寄存器跳转 | PC← Regs[R5] |
| BEQZ R4，name | 等于零时分支 | if(Regs[R4]== 0) PC←name ；<br>$((PC+4)-2^{17}) \leqslant name < ((PC+4)+2^{17})$ |
| BNE R3，R4，name | 不相等时分支 | if(Regs[R3]!= Regs[R4]) PC←name<br>$((PC+4)-2^{17}) \leqslant name < ((PC+4)+2^{17})$ |
| MOVZ R1，R2，R3 | 等于零时移动 | if(Regs[R3]==0) Regs[R1]← Regs[R2] |

# 小结：指令集架构

- **ISA需考虑的问题**
  - Class of ISA
  - Memory addressing
  - Types and sizes of operands
  - Operations
  - Control flow instructions
  - Encoding an ISA
  - ……
- **ISA的类型**
  - 通用寄存器型占主导地位
- **寻址方式**
  - 重要的寻址方式: 偏移寻址方式, 立即数寻址方式, 寄存器间址方式
    - SPEC测试表明，使用频度达到 75%--99%
  - 偏移字段的大小应该在 12 - 16 bits, 可满足75%-99%的需求
  - 立即数字段的大小应该在 8 -16 bits, 可满足50%-80%的需求
- **操作数的类型和大小**
  - 对单字、双字的数据访问具有较高的频率
  - 支持64位双字操作，更具有一般性

# MIPS

- **MIPS是最典型的RISC 指令集架构**
  - Stanford， 1980年提出，主要受到IBM801 小型机的影响
  - 第一个商业实现是R2000（1986）
  - 最初的设计中，其整数指令集仅有**58条指令**，直接实现单发射(single-issued)、顺序流水线
  - 30年来，逐步增加到约**400条指令**。
- **主要特征：**
  - Load/Store型结构，专门的指令完成存储器与寄存器之间的传送
  - ALU类指令的操作数来源于寄存器或立即数（指令中的特定区域）
  - 降低了指令集和硬件的复杂性，依赖于优化编译技术，方便了简单流水线的实现

# MIPS Shortcomings

- **主要缺陷：**
  - 针对特定的微体系架构的实现方式（5级流水、单发射、顺序流水线）进行过度的优化设计
    - 延迟转移 (delayed branch)问题导致超标量等复杂流水线的实现难度，当无法有效填充延迟槽时会导致代码尺寸变大
    - MIPS-I中暴露出其他流水线冲突（load、乘除引起的冲突）采用简单的Interlocking 简单又高效，但为了保持兼容性，仍然保留了延迟转移

**Scheduling the Branch Delay Slot**

# MIPS Shortcomings

- **主要缺陷：**
  - ISA对位置无关的代码（position-independent code, PIC)支持不足。
    - 直接跳转没有提供PC相对寻址，需要通过间接跳转方式实现PIC，增加了代码尺寸，降低了性能
    - 2014年MIPS的修订，改进了PC-相对寻址(针对数据)，但仍然要多条指令才能完成

Please give an example instruction that is position dependent

# MIPS Shortcomings

- **主要缺陷：**
  - 16位位宽立即数消耗了大量编码空间，只有少量的编码空间可供扩展指令
    - 2014修订版，保存有1/64的编码空间供扩展
    - 架构师如果想采用压缩指令编码来降低代码空间，就不得不采用新的指令编码
  - 乘除指令使用了特殊的寄存器（HI,LO)，导致上下文切换内容、指令条数、代码尺寸增加，微架构实现复杂

- ISA假设浮点操作部件是一个独立的协处理器，使得单芯片实现无法最优
  - 例如，整型数与浮点数的转换结果写到浮点数寄存器，使用结果时，需要额外的mov指令，更糟糕的是浮点数寄存器文件与整型数寄存器文件之间的传输，需要有显式的延迟槽
- 在标准的ABI中，保留两个整型寄存器用于内核程序，减少了用户程序可用的寄存器数
- 使用特殊指令处理未对齐的load和store会消耗大量的操作码空间，并使除了最简单的实现之外的其他实现复杂化。
- 时钟速率/CPI 的权衡使得架构师省略了整数大小比较和分支指令。随着分支预测和静态CMOS逻辑的出现，这种权衡在今天已经不太合适了。

- 除了技术方面，MIPS是非开放的专属指令集，不能自由使用

# SPARC (Scalable Processor Architecture)

- **Sun Microsystems的专属指令集**
  - 可追溯到Berkeley RISC-I和RISC-II项目 (early 1980)
  - One of the most successful early commercial RISC
- **SPARC V8 （1990） 主要特征**
  - 用户级 整型ISA **90条指令**； 硬件支持IEEE 754-1985标准的浮点数(50条) ；特权级指令 **20条**
- **主要问题**
  - SPARC使用了寄存器窗口来加速函数调用
    - The number of registers is very limited. At function calls, the registers often need to be saved, thus very slow.
    - Registers windows enable a set of registers to be swamped during function calls.
    - 对于所有的实现来说，寄存器窗口都消耗很大的面积和功耗

# SPARC (Scalable Processor Architecture)

- **Sun Microsystems的专属指令集**
  - 可追溯到Berkeley RISC-I和RISC-II项目 (early 1980)
  - One of the most successful early commercial RISC
- **SPARC V8 （1990） 主要特征**
  - 用户级 整型ISA **90条指令**； 硬件支持IEEE 754-1985标准的浮点数(50条) ；特权级指令 **20条**
- **主要问题**
  - 分支使用条件码
  - For signed numbers, SPARC uses three of the three condition codes- the Z, N, and V bits - to regulate conditional branching
  - 这些条件码由于在一些指令之间创建了额外的依赖关系，增加了体系结构状态并使实现复杂化

```
mov 4, %l1          !move 4 into %l1
subcc %l1, 4, %g0   !subtract 4 from %l1
```

| Instruction | Description | Example |
|---|---|---|
| ba target | branch always | ba loop1 |
| bne target | branch not equal | bne loop1 |
| be target | branch equal | be loop1 |
| bg target | branch greater | bg loop1 |
| ble target | branch less than or equal | ble loop1 |
| bge target | branch greater than or equal | bge loop1 |
| bl target | branch less than | bl loop1 |
| bgu target | branch greater (unsigned ) | bgu loop1 |
| bleu target | branch less or equal ( unsigned ) | bleu loop1 |

# SPARC (Scalable Processor Architecture)

- **Sun Microsystems的专属指令集**
  - 可追溯到Berkeley RISC-I和RISC-II项目 (early 1980)
  - One of the most successful early commercial RISC
- **SPARC V8 （1990） 主要特征**
  - 用户级 整型ISA **90条指令**； 硬件支持IEEE 754-1985标准的浮点数(50条) ；特权级指令 **20条**
- **主要问题**
  - load和store**相邻寄存器对**的指令
    - Ldd （需要一对相邻寄存器存放双字，并且必须是偶数寄存器；高字移入偶数寄存器，低字移入奇数寄存器）
    - 对于简单的微体系结构很有吸引力，可以在很少增加硬件复杂性的情况下提高吞吐量。
    - 遗憾的是当使用寄存器重命名使实现复杂化，因为在寄存器文件中数据在物理上可能不再相邻
  - 浮点寄存器文件和整数寄存器文件之间的移动必须使用内存系统作为中介，限制了系统性能

# SPARC

- ISA通过体系结构公开的延迟陷阱队列支持非精确浮点异常，该队列向系统监控程序提供信息，以恢复此类异常上的处理器状态
- 唯一的原子内存操作是fetch-and-store，这对于实现许多无等待的数据结构是不够的

- **SPARC与其他80年代RISC结构的许多有缺陷的特性**
  - ISA设计面向单发射、顺序、五级流水线的微体系架构；
  - SPARC具有分支延迟插槽和许多显式的数据和控制冲突，这些冲突使代码生成复杂化，无助于更积极的实现；
  - 缺乏位置无关的寻址方式（相对寻址）
  - 由于SPARC缺乏足够的自由编码空间，因此不能方便地对其进行改进以支持压缩ISA扩展

# Alpha（DEC）

- **DEC公司的架构师在20世纪90年代初定义了他们的RISC ISA, Alpha**
  - 摒弃了当时非常吸引人的特性，如分支延迟、条件码、寄存器窗口等
  - 创建了64位寻址空间、设计简洁、实现简单、高性能的ISA
  - Alpha架构师仔细地将特权体系结构和硬件平台的大部分细节隔离在抽象接口(特权体系结构库)后面(PALcode)
- **主要问题：尽管如此，DEC对顺序微架构的Alpha进行了过度优化，并添加了一些不太适合现代实现的特性**
  - 为了追求高时钟频率，ISA的原始版本避免了8位和16位的加载和存储，实际上创建了一个字寻址的内存系统。为了在广泛使用这些操作的应用程序上性能，添加了特殊的未对齐的加载和存储指令以及一些整数指令，以加速重新组合过程。
  - 为了方便长延迟浮点指令的乱序完成，Alpha 有一个非精确的浮点陷阱模型。这个决定可能是可以单独接受的，但是ISA还定义了异常标记和默认值(如果需要的话)必须由软件例程提供。
  - Alpha缺少整数除法指令，建议使用软件牛顿迭代法实现，导致浮点除法速度高于整数除法

- 与它的前辈RISC一样，没有预先考虑可能的压缩指令集扩展，因此没有足够的操作码空间来进行更新
- ISA包含有条件的移动，这使得微架构与寄存器重命名复杂化
  - cmoveq R1, R2, R3 (if R1==0 R2 = R3)

  a = max (a, b) ➔ if (a>= b) {c = a}; else c=b;

- 使用商业Alpha ISAs的一个重要风险:它们可能会被摒弃。康柏在上世纪90年代末收购了摇摇欲坠的DEC后不久，他们选择逐步淘汰Alpha，转而采用英特尔的安腾架构。康柏将Alpha的知识产权出售给了英特尔，此后不久，惠普收购了康柏，并在2004年完成了Alpha的最终实现

# ARMv7 (Advanced RISC Machine)

- **32位 RISC ISA**
  - 目前世界上使用最广的体系结构。当我们权衡是否要设计自己的指令集时，ARMv7是一个自然的选择，大量的软件已经被移植到该ISA上，而且它在嵌入式和移动设备中无处不在。
  - 是一个封闭的标准，剪裁或扩充是不允许的，即使是微架构的创新也仅限于那些能够获得ARM所称的架构许可的人
  - ARMv7十分庞大复杂。整型类指令**600+条**
- **即使知识产权不是问题，它仍然存在一些技术缺陷**
  - 不支持64位地址， ISA缺乏硬件支持IEEE754-2008标准（ ARMv8 纠正了这些缺陷）
  - 特权体系结构的细节渗透到用户级体系结构的定义中

- ARMv7附带一个压缩ISA，具有固定宽度的16位指令，称为Thumb。
  - Thumb虽然提供了有竞争力的代码尺寸，但性能较差
  - Thumb-2 虽然提供了较高的性能，但32位的Thumb-2编码方式与基本的ISA编码方式不同,16位的Thumb-2的编码方式与基本的16位编码方式也不同。导致译码器需要理解三种编码格式，使得能耗、延迟以及设计成本增加
- ISA中包含了许多实现复杂的特性。
  - 程序计数器(PC)是可寻址寄存器之一，这意味着几乎任何指令都可以改变控制流。
  - 更糟糕的是，程序计数器的最低有效位反映ISA当前正在执行(ARM或Thumb)哪个ISA——简单的ADD指令可以更改ISA当前在处理器上执行的指令!
  - 分支使用条件码以及谓词指令进一步使高性能实现复杂化。

# ARMv8

- **2011年，ARM发布新的ISA ARMv8**
  - 64位地址; 扩展了整型寄存器组；32位指令。
  - 摒弃了ARMv7中实现复杂的一些特性
    - PC不再能直接访问；
    - 不再有谓词指令
    - 删除了load-multiple和store-multiple 指令
    - 指令编码归一化
- **主要问题**
  - 使用条件码 (比以前少了很多)
  - 存在许多特殊的寄存器 (zero, PC, sp, program status registers ,etc)

- 增加了一些缺陷，包括大量的subword-SIMD架构
- 指令集更加厚重：**1070条指令，53种格式，8种寻址方式。说明文档达到了5778页**
- 与其他大多数ISA一样，通常以暴露底层实现的方式将用户和特权架构紧密地结合在一起
- 此外，随着ARMv8的引入，ARM不再支持压缩指令编码

- 最后，和它的以前版本一样，ARMv8也是一个封闭的标准

# OpenRISC

- **OpenRISC项目是一个开放源码处理器设计项目**
  - 来源于Hennessy和Patterson的体系结构教科书。
  - 适用于教学、科研和工业界的实现。
- **主要问题** (Pease read https://riscv.org/2014/10/why-not-build-on-openrisc/)
  - OpenRISC项目主要是开源处理器设计项目，而不是开源的ISA 规格说明，ISA和实现是紧密耦合的
  - 固定的32位编码与16位立即数阻碍了压缩ISA扩展
  - 没有64位的支持
  - Delay slots
  - 不能支持压缩指令
  - 值得一提的是：2010年这两个问题都得到了解决:延迟插槽已经成为可选的，64位版本已经定义(但是，据我们所知，从未实现过)。最终，我们（UCB）认为最好从头开始，而不是相应地修改OpenRISC。

# 80x86

- 在过去的四十年里，英特尔的8086架构已经成为笔记本电脑、台式机和服务器市场上最流行的指令集。
  - 在嵌入式系统领域之外，几乎所有流行的软件都被移植到x86上，或者是为x86开发的
  - 它受欢迎的原因有很多：该架构在IBM PC诞生之初的偶然可用性; 英特尔专注于二进制兼容性； 它们积极的卓有成效的微结构实现; 以及他们的前沿制造技术
  - **指令集设计质量并不是它流行的原因之一。**
- 主要问题：
  - **1300条指令**，许多寻址方式，很多特殊寄存器，多种地址翻译方式，**从AMD K5微架构开始，所有的Intel支持乱序执行的微结构，都是动态地将x86指令翻译为内部的RISC-风格的指令集。**
  - ISA不利于虚拟化，因为一些特权指令在用户模式下会无声地失败，而不是被捕获。VMware的工程师们用复杂的动态二进制翻译软件解决了这一缺陷
  - ISA的指令长度为任意整数字节数，最多为15个字节，但是数量较少的短操作码已经被随意使用

- ISA有数量极少的寄存器组
- 大多数整数寄存器在ISA中执行特殊功能，这加剧了体系结构寄存器的不足
- 一些ISA特性，包括隐式条件代码和带有谓词的移动操作，在微架构中实现复杂

- 这些ISA决策对静态代码大小有深刻的影响。尽管存在所有这些缺陷，x86通常比RISC体系结构使用更少的动态指令完成相同的功能，因为x86指令可以编码多个基本操作。
- 最后，80x86是一个专有指令集

# ISA Summary

| | MIPS | SPARC | Alpha | ARMv7 | ARMv8 | OpenRISC | 80x86 |
|---|---|---|---|---|---|---|---|
| Free and Open | | √ | | | | √ | |
| 64-bit Address | √ | √ | √ | | √ | √ | √ |
| Compressed Instructions | √ | | | √ | | | Partial |
| Separate Privileged ISA | | | √ | | | | |
| Position-Indep. Code | Partial | | | √ | √ | | √ |
| IEEE 754-2008 | | | | | √ | | √ |
| Classically Virtualizable | √ | √ | √ | | √ | | |

# ISA的功能设计

CISC vs RISC

- 一般来讲，RISC vs CISC, 哪个需要更复杂的硬件支持？哪个需要更复杂的编译支持？为什么？
- 说出一条你最不喜欢的关于RISC的设计

- **ISA的功能设计**
  - 任务：确定硬件支持哪些操作
  - 方法：统计的方法
  - 两种类型：CISC和RISC
- **CISC（Complex Instruction Set Computer）**
  - 目标：强化指令功能，减少运行的指令条数，提高系统性能
  - 方法：面向目标程序的优化，面向高级语言和编译器的优化
- **RISC（Reduced Instruction Set Computer）**
  - 目标：通过简化指令系统，用高效的方法实现最常用的指令
  - 方法：充分发挥流水线的效率，降低（优化）CPI

# CISC计算机ISA的功能设计

- **目标：强化指令功能，减少指令条数，以提高系统性能**
- **基本优化方法**

**1. 面向目标程序的优化是提高计算机系统性能最直接方法**

- 优化目标
  - 缩短程序的长度
  - 缩短程序的执行时间
- 优化方法
  - 统计分析目标程序执行情况，找出使用频度高，执行时间长的指令或指令串
  - 优化使用频度高的指令
  - 用新的指令代替使用频度高的指令串

# 优化目标程序的主要途径（1/2)

1）增强运算型指令的功能

如sin(x), cos(x), SQRT(X)，甚至多项式计算

如用一条三地址指令完成

$P(X) = C(0)+C(1)X+C(2)X^2+C(3)X^3+\ldots..$

2) 增强数据传送类指令的功能

主要是指数据块传送指令

- R-R, R-M, M-M之间的数据块传送可有效的支持向量和矩阵运算，如IBM370
- R-Stack之间设置数据块传送指令，能够在程序调用和程序中断时，快速保存和恢复程序现场，如 VAX-11

# 优化目标程序的主要途径（2/2)

**3) 增强程序控制指令的功能**

在CISC中，一般均设置了多种程序控制指令，正常仅需要转移指令和子程序控制指令

**2. 面向高级语言和编译程序改进指令系统**

优化目标：主要是缩小HL-ML之间的差距

优化方法：

**1）增强面向HL和Compiler支持的指令功能**

- 统计分析源程序中各种语句的使用频度和执行时间
- 增强相关指令的功能，优化使用频度高、执行时间长的语句
- 增加专门指令，以缩短目标程序长度，减少目标程序执行时间，缩短编译时间

## FORTRAN语言和COBOL语言中各种主要语句的使用频度

| 语言 | 一元赋值 | 其他赋值 | IF | GOTO | I/O | DO | CALL | 其他 |
|---|---|---|---|---|---|---|---|---|
| FORTRAN | 31.0 | 15.0 | 11.5 | 10.5 | 6.5 | 4.5 | 6.0 | 15.0 |
| COBOL | 42.1 | 7.5 | 19.1 | 19.1 | 8.46 | 0.17 | 0.17 | 3.4 |

**观察结果：**

**（1）一元赋值在其中比例最大，增强数据传送类指令功能，缩短这类指令的执行时间是对高级语言非常有力的支持，**

**（2）其他赋值语句中，增1操作比例较大，许多机器都有专门的增1指令**

怎么算出来的？

**（3）条件转移和无条件转移占22%，38.2%,因此增强转移指令的功能，增加转移指令的种类是必要的**

## 2）高级语言计算机系统

缩小HL和ML的差别时，走到极端，就是把HL和ML合二为一，即所谓的高级语言计算机。在这种机器中，高级语言不需要经过编译，直接由机器硬件来执行。如LISP机，PROLOG机

## 3）支持操作系统的优化实现 - 些特权指令

任何一种计算机系统通常需要操作系统，而OS又必须用指令系统来实现，指令系统对OS的支持主要有

- 处理器工作状态和访问方式的转换
- 进程的管理和切换
- 存储管理和信息保护
- 进程同步和互斥，信号量的管理等

# RISC的定义和特点

- **RISC是一种计算机体系结构的设计思想，它不是一种产品。RISC是近代计算机体系结构发展史中的一个里程碑，直到现在，RISC没有一个确切的定义**
- **早期对RISC特点的描述**
  - 大多数指令在单周期内完成
  - 采用Load/Store结构
  - 硬布线控制逻辑
  - 减少指令和寻址方式的种类
  - 固定的指令格式
  - 注重代码的优化
- **从目前的发展看，RISC体系结构还应具有如下特点：**
  - 面向寄存器结构
  - 十分重视流水线的执行效率 - 尽量减少断流
  - 重视优化编译技术
- **减少指令平均执行周期数 CPI 是RISC思想的精华**

# 问题

RISC的指令系统精简了，CISC中的一条指令可能由一串指令才能完成，那么为什么RISC执行程序的速度比CISC还要快？

ExecuteTime = CPI X IC X T

| | IC | CPI | T |
|---|---|---|---|
| CISC | 1 | 2~15 | 33ns~5ns |
| RISC | 1.3~1.4 | 1.1~1.4 | 10ns~2ns |

IC： 实际统计结果，RISC的IC只比CISC 长30%~40%

CPI: CISC CPI一般在4~6之间，RISC 一般CPI =1，
Load/Store 为2

T: RISC采用硬布线逻辑，指令要完成的功能比较简单

# RISC为什么会减少CPI

- **硬件方面：**
  - 硬布线控制逻辑
  - 减少指令和寻址方式的种类
  - 使用固定格式
  - 采用Load/Store
  - 指令执行过程中设置多级流水线。
- **软件方面：十分强调优化编译的作用**

# 小结

- **ISA的功能设计：任务为确定硬件支持哪些操作。方法是统计的方法。存在CISC和RISC两种类型**
  - CISC（Complex Instruction Set Computer）
    - 目标：强化指令功能，减少指令的指令条数，以提高系统性能
    - 基本方法：面向目标程序的优化，面向高级语言和编译器的优化
  - RISC（Reduced Instruction Set Computer）
    - 目标：通过简化指令系统，用最高效的方法实现最常用的指令
    - 主要手段：充分发挥流水线的效率，降低（优化）CPI
- **控制转移类指令**
- **指令编码（指令格式）**

# ISA的演进

Single Accumulator (EDSAC 1950)

Accumulator + Index Registers
(Manchester Mark I, IBM 700 series 1953)

Separation of Programming Model
from Implementation

High-level Language Based
(B5000 1963)

Concept of a Family
(IBM 360 1964)

General Purpose Register Machines

Complex Instruction Sets，CISC
(Vax, Intel 8086 1977-80)

Load/Store Architecture
(CDC 6600, Cray 1 1963-76)

VLIW

Cydra-5 1983, Multiflow 1984,
TMSC6X 1996, Intel EPIC 1997

RISC
(Mips,Sparc, IBM RS6000, . . .1987+)

# RISC-V ISA

- UC Berkeley 设计的第5代RISC指令集
- 设计理念（指导思想）：通用的ISA
  - 能适应从最袖珍的嵌入式控制器，到最快的高性能计算机等各种规模的处理器。
  - 能兼容各种流行的软件栈和编程语言。
  - 适应所有实现技术，包括现场可编程门阵列（FPGA）、专用集成电路（ASIC）、全定制芯片，甚至未来的技术。
  - 对所有微体系结构实现方式都有效。例如：
    - 微编码或硬连线控制；顺序或乱序执行流水线；单发射或超标量等等。
  - 支持广泛的定制化，成为定制加速器的基础。随着摩尔定律的消退，加速器的重要性日益提高。
  - 基础的指令集架构是稳定的。不能像以前的专有指令集架构一样被弃用，例如AMD Am29000、Digital Alpha、Digital VAX、Hewlett Packard PA-RISC、Intel i860、Intel i960、Motorola 88000、以及Zilog Z8000。
  - 完全开源

# 技术目标

- **将ISA分成基础ISA和可选的扩展部分**
  - ISA的基础足够简单，可以用于教学和许多嵌入式处理器，包括定制加速器的控制单元。它足够完整，可以运行现代软件栈。
  - 扩展部分提高计算的性能，并支持多处理机并行
  - 支持32位和64位地址空间
- **方便定制ISA扩展**
  - 包括紧耦合功能单元和松耦合协处理器
- **支持变长指令集扩展**
  - 既为了提高代码密度，也为了扩展可能的自定义ISA扩展的空间
- **提供对现代标准的有效硬件支持**
- **用户级ISA和特权级ISA是正交的（相互独立，互不依赖）**
  - 在保持用户应用程序二进制接口(ABI)兼容性的同时，允许完全虚拟化，并允许在特权ISA中进行实验测试

# RISC-V ISA的特点

- **完全开源：**
  - 它属于一个开放的，非营利性质的RISC-V基金会。
  - 开源采用BSD协议（企业完全自由免费使用，允许企业添加自有指令而不必开放共享以实现差异化发展）
- **架构简单**
  - 没有针对某一种微体系结构实现方式做过度的架构设计
  - 新的指令集，没有向后（backward）兼容的包袱
  - 说明书的页数.....（图1.6）
- **模块化的指令集架构**
  - RV32I和RV64I是基础的ISA。可扩展增加其他特性的支持
  - 面向教育或科研，易于扩充或剪裁
  - 支持32位和64位地址空间
- **面向多核并行**

| ISA | Pages | Words | Hours to read | Weeks to read |
|---|---|---|---|---|
| RISC-V | 236 | 76,702 | 6 | 0.2 |
| ARM-32 | 2736 | 895,032 | 79 | 1.9 |
| x86-32 | 2198 | 2,186,259 | 182 | 4.5 |

图1.6：ISA手册的页数和字数来自[Waterman and Asanovi'c 2017a]，[Waterman and Asanovi'c 2017b]，[Intel Corporation 2016]，[ARM Ltd. 2014]。读完需要的时间按每分钟读200个单词，每周读40小时计算。 基于[Baumann 2017]的图1的一部分。

# RISC-V子集命名约定

**ISA base and extensions**

| Name | Description | Version | Status[a] |
|---|---|---|---|
| **Base** | | | |
| RV32I | Base Integer Instruction Set, 32-bit | 2.1 | Frozen |
| RV32E | Base Integer Instruction Set (embedded), 32-bit, 16 registers | 1.9 | Open |
| RV64I | Base Integer Instruction Set, 64-bit | 2.0 | Frozen |
| RV128I | Base Integer Instruction Set, 128-bit | 1.7 | Open |
| **Extension** | | | |
| M | Standard Extension for Integer Multiplication and Division | 2.0 | Frozen |
| A | Standard Extension for Atomic Instructions | 2.0 | Frozen |
| F | Standard Extension for Single-Precision Floating-Point | 2.0 | Frozen |
| D | Standard Extension for Double-Precision Floating-Point | 2.0 | Frozen |
| G | Shorthand for the base and above extensions | N/A | N/A |
| Q | Standard Extension for Quad-Precision Floating-Point | 2.0 | Frozen |
| L | Standard Extension for Decimal Floating-Point | 0.0 | Open |
| C | Standard Extension for Compressed Instructions | 2.0 | Frozen |
| B | Standard Extension for Bit Manipulation | 0.92 | Open |
| J | Standard Extension for Dynamically Translated Languages | 0.0 | Open |
| T | Standard Extension for Transactional Memory | 0.0 | Open |
| P | Standard Extension for Packed-SIMD Instructions | 0.1 | Open |
| V | Standard Extension for Vector Operations | 0.8 | Open |
| N | Standard Extension for User-Level Interrupts | 1.1 | Open |
| H | Standard Extension for Hypervisor | 0.4 | Open |

a. ^ Frozen parts are expected to have their final feature set and to receive only clarifications before being ratified.

# RV32 处理器状态

- 32x32-bit 整型数寄存器 (**x0-x31**)
  - **x0 总是** 0
  - 原则上其他寄存器可以通用
  - 不过software convention如右图所示

| Register name | Symbolic name | Description | Owner |
|---|---|---|---|
| **32 integer registers** | | | |
| x0 | Zero | Always zero | |
| x1 | ra | Return address | Caller |
| x2 | sp | Stack pointer | Callee |
| x3 | gp | Global pointer | |
| x4 | tp | Thread pointer | |
| x5 | t0 | Temporary / alternate return address | Caller |
| x6–7 | t1–2 | Temporary | Caller |
| x8 | s0/fp | Saved register / frame pointer | Callee |
| x9 | s1 | Saved register | Callee |
| x10–11 | a0–1 | Function argument / return value | Caller |
| x12–17 | a2–7 | Function argument | Caller |
| x18–27 | s2–11 | Saved register | Callee |
| x28–31 | t3–6 | Temporary | Caller |
| **32 floating-point extension registers** | | | |
| f0–7 | ft0–7 | Floating-point temporaries | Caller |
| f8–9 | fs0–1 | Floating-point saved registers | Callee |
| f10–11 | fa0–1 | Floating-point arguments/return values | Caller |
| f12–17 | fa2–7 | Floating-point arguments | Caller |
| f18–27 | fs2–11 | Floating-point saved registers | Callee |
| f28–31 | ft8–11 | Floating-point temporaries | Caller |

# RISC-V 指令编码

| | | | |
|---|---|---|---|
| | | xxxxxxxxxxxxxxaa | 16-bit (aa ≠ 11) |
| | xxxxxxxxxxxxxxxx | xxxxxxxxxxxbbb11 | 32-bit (bbb ≠ 111) |
| ···xxxx | xxxxxxxxxxxxxxxx | xxxxxxxxxx011111 | 48-bit |
| ···xxxx | xxxxxxxxxxxxxxxx | xxxxxxxxx0111111 | 64-bit |
| ···xxxx | xxxxxxxxxxxxxxxx | xnnnxxxxx1111111 | (80+16*nnn)-bit, nnn≠111 |
| ···xxxx | xxxxxxxxxxxxxxxx | x111xxxxx1111111 | Reserved for ≥192-bits |
| Byte Address: | base+4 | base+2 | base |

- 可支持变长指令
- 基础指令集 (RV32) 总是固定的32-bit指令（lowest two bits = $11_2$）
- 所有的条件转移和无条件转移的转移地址16-bit对齐

# RV32I 指令格式

Additional opcode bits/immediate

Reg. Source 2

Reg. Source 1

Destination Reg.

7-bit opcode field (but low 2 bits $=11_2$)

| 31 30 25 | 24 21 20 | 19 15 | 14 12 | 11 8 | 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | | opcode | R-type |
| imm[11:0] | | rs1 | funct3 | rd | | opcode | I-type |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | | opcode | S-type |
| imm[12] imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type |
| imm[31:12] | | | | rd | | opcode | U-type |
| imm[20] imm[10:1] | imm[11] | imm[19:12] | | rd | | opcode | J-type |

# RV32I Integer-Immediate (I)

| 31 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| I-immediate[11:0] | src | ADDI/SLTI[U] | dest | OP-IMM |
| I-immediate[11:0] | src | ANDI/ORI/XORI | dest | OP-IMM |

`ADDI` adds the sign-extended 12-bit immediate to register `rs1`. Arithmetic overflow is ignored and the result is simply the low `XLEN` bits of the result.

`ADDI rd, rs1, 0` is used to implement the `MV rd, rs1` assembler pseudo-instruction.

# RV32I Integer-Immediate (I)

| 31 – 25 | 24 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|---|
| imm[11:5] | imm[4:0] | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| 0000000 | shamt[4:0] | src | SLLI | dest | OP-IMM |
| 0000000 | shamt[4:0] | src | SRLI | dest | OP-IMM |
| 0100000 | shamt[4:0] | src | SRAI | dest | OP-IMM |

Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in `rs1`, and the `shift amount` is encoded in the lower 5 bits of the `I-immediate` field. The right shift type is encoded in bit 30.

`SLLI` is a logical left shift (zeros are shifted into the lower bits);
`SRLI` is a logical right shift (zeros are shifted into the upper bits);
`SRAI` is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

# RV32I Register-Register (R)

| 31 – 25 | 24 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| 0000000 | src2 | src1 | ADD/SLT/SLTU | dest | OP |
| 0000000 | src2 | src1 | AND/OR/XOR | dest | OP |
| 0000000 | src2 | src1 | SLL/SRL | dest | OP |
| 0100000 | src2 | src1 | SUB/SRA | dest | OP |

`ADD` performs the addition of `rs1` and `rs2`. `SUB` performs the subtraction of `rs2` from `rs1`. Overflows are ignored and the low `XLEN` bits of results are written to the destination `rd`.

`SLT` and `SLTU` perform signed and unsigned compares respectively, writing `1` to `rd` if `rs1 < rs2`, `0` otherwise.

`AND`, `OR`, and `XOR` perform bitwise logical operations.
`SLL`, `SRL`, and `SRA` perform logical left, logical right, and arithmetic right shifts on the value in register `rs1` by the shift amount held in the lower 5 bits of register `rs2`.

# RV32I Unconditional Jump (J)

| 31 | 30 21 | 20 | 19 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode |
| 1 | 10 | 1 | 8 | 5 | 7 |
| | offset[20:1] | | | dest | JAL |

Plain unconditional jumps (assembler pseudoinstruction J) are encoded as a `JAL` with `rd=x0`.

The jump and link (`JAL`) instruction uses the `J`-type format, where the `J`-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ±1 MiB range.

Position Independent!

# RV32I Indirect Jump (I)

| 31 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| offset[11:0] | base | 0 | dest | JALR |

The indirect jump instruction `JALR` (jump and link register) uses the `I`-type encoding. The target address is obtained by adding the sign-extended 12-bit `I-immediate` to the register `rs1`, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (`pc+4`) is written to register `rd`. Register `x0` can be used as the destination if the result is not required.

No architecturally visible delay slots!

# RV32I Conditional Branches (B)

| 31 | 30 25 | 24 20 | 19 15 | 14 12 | 11 8 | 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode |
| 1 | 6 | 5 | 5 | 3 | 4 | 1 | 7 |
| offset[12\|10:5] | | src2 | src1 | BEQ/BNE | offset[11\|4:1] | | BRANCH |
| offset[12\|10:5] | | src2 | src1 | BLT[U] | offset[11\|4:1] | | BRANCH |
| offset[12\|10:5] | | src2 | src1 | BGE[U] | offset[11\|4:1] | | BRANCH |

The 12-bit `B-immediate` encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ±4 KiB.

Branch instructions compare two registers. `BEQ` and `BNE` take the branch if registers `rs1` and `rs2` are equal or unequal respectively.

`BLT` and `BLTU` take the branch if `rs1` is less than `rs2`, using signed and unsigned comparison respectively.

`BGE` and `BGEU` take the branch if `rs1` is greater than or equal to `rs2`, using signed and unsigned comparison respectively.

# Integer Load/Store Instructions

| 31 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| offset[11:0] | base | width | dest | LOAD |

I-type

| 31 – 25 | 24 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|---|
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| offset[11:5] | src | base | width | offset[4:0] | STORE |

S-type

`Load` and `store` instructions transfer a value between the registers and memory. Loads are encoded in the `I`-type format and stores are `S`-type. The effective address is obtained by adding register `rs1` to the sign-extended 12-bit offset. Loads copy a value from memory to register `rd`. Stores copy the value in register `rs2` to memory.

# RV32I 带有指令布局，操作码，格式类型和名称的操作码映射

| 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 |

# RV32I 带有指令布局，操作码，格式类型和名称的操作码映射

| 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 |

# 微程序实现

# 思考

- 认真对照MIPS的每条指令与RV32I的指令。发现RV32I做出的一系列改动，并解释为什么做出这些改动。
- 如果卡在某条指令上面，立即联系助教或者老师

# RISC-V 指令执行阶段

- **Instruction Fetch**
- **Instruction Decode**
- **Register Fetch**
- **ALU Operations**
- **Optional Memory Operations**
- **Optional Register Writeback**
- **Calculate Next Instruction Address**

# 控制部分与数据通路

- 处理器设计可以分为datapath和Control设计两部分
  - datapath, 存储数据、算术逻辑运算单元
  - control, 控制数据通路上的一系列操作

- 早期的计算机设计者的最大挑战是控制逻辑的正确性
- Maurice Wilkes 提出了微程序设计的概念来设计处理器的控制逻辑（EDSAC-II, 1958）
- 当时的技术水平
  - Logic: Vacuum Tubes
  - Main Memory: Magnetic cores
  - Read-Only Memory: Diode matrix, punched metal cards, …
  - Cost： Logic > RAM > ROM
  - Speed: ROM > RAM

# 微程序控制RISC-V的单总线数据通路

微指令的寄存器传输级表示:

- MA:=PC means RegSel=PC; RegW=0; RegEn=1; MALd=1

# 微程序控制RISC-V的单总线数据通路

微指令的寄存器传输级表示:

- B:=Reg[rs2] means RegSel=rs2; RegW=0; RegEn=1; BLd=1

# 微程序控制RISC-V的单总线数据通路

微指令的寄存器传输级表示:

- Reg[rd]:=A+B means ALUop=Add; ALUEn=1; RegSel=rd; RegW=1

# 微程序控制 CPU

Next State
μPC
Microcode ROM
(holds fixed μcode instructions)
Decoder
Busy?
Opcode
Condition
Control Lines
Datapath
Address
Data
Main Memory
(holds user program written in macroinstructions, e.g., x86, RISC-V)

# Microcode示意 (1)

**Instruction Fetch:**

**MA,A:=PC**
**PC:=A+4**
***wait for memory***
**IR:=Mem**
***dispatch on opcode***

**ALU:**

**A:=Reg[rs1]**
**B:=Reg[rs2]**
**Reg[rd]:=ALUOp(A,B)**
***goto instruction fetch***

**ALUI:**

**A:=Reg[rs1]**
**B:=ImmI //Sign-extend 12b immediate**
**Reg[rd]:=ALUOp(A,B)**
***goto instruction fetch***

# Microcode 示意 (2)

**LW:**

**A:=Reg[rs1]**
**B:=ImmI  //Sign-extend 12b immediate**
**MA:=A+B**
***wait for memory***
**Reg[rd]:=Mem**
***goto instruction fetch***

**JAL:**

**Reg[rd]:=A  // Store return address**
**A:=A-4  // Recover original PC**
**B:=ImmJ  // Jump-style immediate**
**PC:=A+B**
***goto instruction fetch***

**Branch:**

**A:=Reg[rs1]**
**B:=Reg[rs2]**
**if (!ALUOp(A,B))** ***goto instruction fetch*** **//Not taken**
**A:=PC  //Microcode fall through if branch taken**
**A:=A-4**
**B:=ImmB// Branch-style immediate**
**PC:=A+B**
***goto instruction fetch***

# 采用 ROM 实现微程序控制

- **How many address bits?**

  |μaddress| = |μPC|+|opcode|+ 1 + 1
- **How many data bits?**

  |data| = |μPC|+|control signals| = |μPC| + 18
- **Total ROM size = $2^{|\mu address|}$x|data|**

# ROM 中的内容

| Address | | | | Data | |
|---|---|---|---|---|---|
| µPC | Opcode | Cond? | Busy? | Control Lines | Next µPC |
| fetch0 | X | X | X | MA,A:=PC | fetch1 |
| fetch1 | X | X | 1 | | fetch1 |
| fetch1 | X | X | 0 | IR:=Mem | fetch2 |
| fetch2 | ALU | X | X | PC:=A+4 | ALU0 |
| fetch2 | ALUI | X | X | PC:=A+4 | ALUI0 |
| fetch2 | LW | X | X | PC:=A+4 | LW0 |
| …. | | | | | |
| ALU0 | X | X | X | A:=Reg[rs1] | ALU1 |
| ALU1 | X | X | X | B:=Reg[rs2] | ALU2 |
| ALU2 | X | X | X | Reg[rd]:=ALUOp(A,B) | fetch0 |

# 单总线数据通路结构的微程序控制存储器大小

- 取指阶段有3个common steps
- 指令分为12 组
- 完成一条指令需要5条微指令（包括dispatch）
- 共计 3+12*5 = 63条微指令,因此 μPC需要6位

- 指令操作码（Opcode)5 位, 每条微指令~18个控制信号

- 微控制器的大小 = $2^{(6+5+2)}$x(6+18)=$2^{13}$x24 = ~25KiB!

# Reducing Control Store Size

- Reduce ROM height (#address bits)
  - Use external logic to combine input signals
  - Reduce #states by grouping opcodes
- Reduce ROM width (#data bits)
  - Restrict µPC encoding (next,dispatch,wait on memory,…)
  - Encode control signals (vertical µcoding, nanocoding)

# 单总线 RISC-V 微程序控制引擎

## Reducing Control Store Size

$|\mu address| = |\mu PC| + |opcode| + 1 + 1$

$|data| = |\mu PC| + |control\ signals|$

Total ROM size = $2^{|\mu address|} \times |data|$

μPC jump = next | spin | fetch | dispatch | ftrue | ffalse

# μPC Jump 类型

- ***next*** **: increments μPC**
- ***spin*** **: waits for memory**
- ***fetch*** **: jumps to start of instruction fetch**
- ***dispatch*** **: jumps to start of decoded opcode group**
- ***ftrue/ffalse*** **: jumps to fetch if Cond? true/false**

# 微程序控制存储器ROM中的内容

| Address | Data | |
|---|---|---|
| μPC | Control Lines | Next μPC |
| fetch0 | MA,A:=PC | next |
| fetch1 | IR:=Mem | spin |
| fetch2 | PC:=A+4 | dispatch |
| | | |
| ALU0 | A:=Reg[rs1] | next |
| ALU1 | B:=Reg[rs2] | next |
| ALU2 | Reg[rd]:=ALUOp(A,B) | fetch |
| | | |
| Branch0 | A:=Reg[rs1] | next |
| Branch1 | B:=Reg[rs2] | next |
| Branch2 | A:=PC | ffalse |
| Branch3 | A:=A-4 | next |
| Branch4 | B:=ImmB | next |
| Branch5 | PC:=A+B | fetch |

# 示例：实现一条复杂指令

**Memory-memory add: M[rd] = M[rs1] + M[rs2]**

| Address | Data | |
|---|---|---|
| μPC | Control Lines | Next μPC |
| MMA0 | MA:=Reg[rs1] | next |
| MMA1 | A:=Mem | spin |
| MMA2 | MA:=Reg[rs2] | next |
| MMA3 | B:=Mem | spin |
| MMA4 | MA:=Reg[rd] | next |
| MMA5 | Mem:=ALUOp(A,B) | spin |
| MMA6 | | fetch |

复杂指令的实现通常不需要修改数据通路，仅仅需要编写相应的微程序（可能会占用更多的控存）

采用硬布线控制器而不修改数据通路来实现这些指令是非常困难的

# Single-Bus Datapath for Microcoded RISC-V

Datapath unchanged for complex instructions!

# Horizontal vs Vertical µCode

- **水平微编码的水平型微指令**
  - 每条微指令有多个微操作并行
  - 每条宏指令（指令）具有较少的微指令
  - 稀疏的（微操作）编码 ⇒ 微指令较宽（含有较多的位数）
- **垂直微编码的垂直型微指令**
  - 典型的是每条微指令代表一个数据通路操作
    - 不同的数据通路分支是独立的微指令
  - 每条宏指令（指令）需要更多的微指令
  - 紧凑的（微指令）编码 ⇒ 微指令较窄（含有较少的位数）
- **Nanocoding**
  - 水平型微指令和垂直型微指令的结合

Reduce ROM width (#data bits)

# Acknowledgements


- **These slides contain material developed and copyright by:**
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- **MIT material derived from course 6.823**
- **UCB material derived from course CS252**
- **KFUPM material derived from course COE501、COE502**

# 计算机体系结构

## Topic III: Pipelining

# 流水线的基本概念

- **一个任务可以分解为 *k* 个子任务**
  - K 个子任务在 K 个不同阶段（使用不同的资源）运行
  - 每个子任务执行需要 1 个单位时间
  - 整个任务的执行时间为 K 倍单位时间
- **流水线执行模式是重叠执行模式**
  - K 个流水段并行执行 K 个不同任务
  - 每个单位时间进入 / 离开流水线一个任务

**Serial Execution**

One completion every *k* time units

**Pipelined Execution**

One completion every 1 time unit

# 同步流水线

- **流水段之间采用时钟控制的寄存器文件（clocked registers)**
- **时钟上升沿到达时…**
  - 所有寄存器同时保存前一流水段的结果
- **流水段是组合逻辑电路**
- **流水线设计中希望各段相对平衡**

# 流水线的性能

- **设 $\tau_i$ = time delay in stage Si**
- **时钟周期 $\tau$ = max( $\tau_i$ 为最长的流水段延迟**
- **时钟频率 f = 1/ $\tau$= 1/max( )$\tau_i$**
- **流水线可以在 k+n-1 个时钟周期内完成 n 个任务**
  - 完成第一个任务需要 k 个时钟周期
  - 其他 n-1 个任务需要 n-1 个时钟周期完成
- **K- 段流水线的理想加速比（相对于串行执行）**

$$S_k = \frac{\text{Serial execution in cycles}}{\text{Pipelined execution in cycles}} = \frac{nk}{k+n-1} \qquad S_k \rightarrow k \text{ for large } n$$

# 典型的 RISC　5 段流水线

- **5 个流水段，每段的延迟为 1 个 cycle**
- **IF: 取值阶段**
  - 选择地址：下一条指令地址、转移地址
- **ID: 译码阶段**
  - 确定控制信号 并从寄存器文件中读取寄存器值
- **EX: 执行**
  - Load 、 Store： 计算有效地址
  - Branch： 计算转移地址并确定转移方向
- **MEM: 存储器访问（仅 Load 和 Store)**
- **WB: 结果写回**

# 典型的 RISC 5 段流水线

*This version designed for regfiles/memories with synchronous reads and writes.*

# 流水线的可视化表示

- **多条指令执行多个时钟周期**
  - 指令按程序序从上到下排列
  - 图中展示了每一时钟周期资源的使用情况
  - 不同指令相邻阶段之间没有干扰

# Recap： 流水线技术要点

- **流水线技术要点**
  - 多个任务重叠（并发 / 并行）执行，但使用不同的资源
  - 流水线技术提高整个系统的吞吐率，不能缩短单个任务的执行时间
  - 其潜在的加速比＝流水线的级数
  - 影响流水线性能的因素
    - 流水线中的瓶颈——最慢的那一段
    - 流水段所需时间不均衡将降低加速比
    - 流水线存在装入时间和排空时间，使得加速比降低
  - 由于存在相关 (hazards) 问题，会导致流水线停顿
    - Hazards 问题：流水线的执行可能会导致对资源的访问冲突，或破坏对资源的访问顺序
- **流水线正常工作的基本条件**
  - 增加寄存器文件保存当前段传送到下一段的数据和控制信息
  - 需要更高的存储器带宽

# 指令流时序

- **时序图展示：**
  - 每个时钟周期指令所使用的流水段情况
- **指令流在采用 5 段流水线执行模式的执行情况**

# 单周期、多周期、流水线控制性能比较

## Single-Cycle Execution:

$T_{clock}$ = 350+250+350+350+250 = 1550 ps

CPI = 1, but long clock cycle

## Multi-Cycle Execution:

$T_{clock}$ = 350 ps

Average CPI = 5×0.2 + 4×0.1 + 4×0.4 + 3×0.3 = 3.9

## Pipelined Execution:

$T_{clock}$ = 350 ps = max(350, 250)

One instruction completes each cycle

Average CPI = 1

Ignore time to fill pipeline

# 流水线的相关（Hazards)

- **结构相关：流水线中一条指令可能需要另一条指令使用的资源**

- **数据和控制相关：一条指令可能依赖于先前的指令生成的内容**
  - 数据相关：依赖先前指令产生的结果（数据）值
  - 控制相关：依赖关系是如何确定下一条指令地址 (branches, exceptions)

- **处理相关的一般方法是插入 bubble，导致 CPI>1 (单发射理想 CPI)**

# Pipeline CPI Examples

*Measure from when first instruction finishes to when last instruction in sequence finishes.*

Time →

3 instructions finish in 3 cycles
CPI = 3/3 =1

3 instructions finish in 4 cycles
CPI = 4/3 = 1.33

3 instructions finish in 5cycles
CPI = 5/3 = 1.67

# 消减结构相关

- **当两条指令同时需要相同的硬件资源时，就会发生结构相关（冲突）**
  - 方法 1： 通过将新指令延迟到前一条指令执行完（释放资源后）执行
  - 方法 2： 增加新的资源
    - E.g.，如果两条指令同时需要操作存储器，可以通过增加到两个存储器操作端口来避免结构冲突
- **经典的 RISC 5- 段整型数流水线通过设计可以没有结构相关**
  - 很多 RISC 实现在多周期操作时存在结构相关
    - 例如多周期操作的 multipliers, dividers, floating-point units 等，由于没有多个寄存器文件写端口 导致 结构冲突

# 三种基本的数据相关

- **写后读相关 (Read After Write (RAW))**

```
I:  add x1,x2,x3
J:  sub x4,x1,x3
```

  – 由于实际的数据交换需求而引起的

- **读后写相关（Write After Read (WAR)**

```
I:  sub x4,x1,x3
J:  add x1,x2,x3
```

  – 编译器编写者称之为“anti-dependence”（反相关），是由于重复使用寄存器名“**x1**”引起的.

- **写后写相关（Write After Write (WAW)）**

```
I:  sub x1,x4,x3
J:  add x1,x2,x3
```

  – 编译器编写者称之为“output dependence”，也是由于重复使用寄存器名“**x1**”引起的.
  – 在后面的复杂的流水线中我们将会看到 WAR 和 WAW 相关

# 消减数据相关的三种策略

- **连锁机制（Interlock）**
  - 在 issue 阶段保持当前相关指令，等待相关解除
- **设置旁路定向路径（Bypass or Forwarding)**
  - 只要结果可用，通过旁路尽快传递数据
- **投机（Speculate）**
  - 猜测一个值继续，如果猜测错了再更正

# Interlocking Versus Bypassing

```
add x1, x3, x5
sub x2, x1, x4
```

Instruction interlocked in decode stage

Bypass around ALU with no bubbles

# Example Bypass Path

# Fully Bypassed Data Path

Fetch | Decode | EXecute | Memory | Writeback

PC
Instruction Cache
Inst. Register
Registers
Imm
B
A
ALU
Store
Data Cache

| | | | | |
|---|---|---|---|---|
| F | D | X | M | W |
| | F | D | X | M | W |
| | | F | D | X | M | W |
| | | | F | D | X | M | W |

```
lw $t0, 4($t1)
addi $t1, $t1
addi $t0, $t0, 1
```

*[ Assumes data written to registers in a W cycle is readable in parallel D cycle (dotted line). Extra write data register and bypass paths required if this is not possible. ]*

# 针对数据相关的值猜测执行

- **不等待产生结果的指令产生值，直接猜测一个值继续**

- **这种技术，仅在某些情况下可以使用：**
  - 分支预测
  - 堆栈指针更新
  - 存储器地址消除歧义（Memory address disambiguation）

# 采用软件方法避免数据相关

**Try producing fast code for**

**a = b + c;**

**d = e – f;**

**assuming a, b, c, d ,e, and f in memory.**

**Slow code:**

```
LW  Rb,b
LW  Rc,c
ADD     Ra,Rb,Rc
SW      a,Ra
LW  Re,e
LW  Rf,f
SUB     Rd,Re,Rf
SW  d,Rd
```

Fast code:

```
LW  Rb,b
LW  Rc,c
LW  Re,e
ADD     Ra,Rb,Rc
LW  Rf,f
SW      a,Ra
SUB     Rd,Re,Rf
SW d,Rd
```

# 流水线的可视化表示

- **多条指令执行多个时钟周期**
  - 指令按程序序从上到下排列
  - 图中展示了每一时钟周期资源的使用情况
  - 不同指令相邻阶段之间没有干扰

# 流水线的相关（Hazards)

- **结构相关：流水线中一条指令可能需要另一条指令使用的资源**

- **数据和控制相关：一条指令可能依赖于先前的指令生成的内容**
  - 数据相关：依赖先前指令产生的结果（数据）值
  - 控制相关：依赖关系是如何确定下一条指令地址 (branches, exceptions)

- **处理相关的一般方法是插入 bubble，导致 CPI>1 (单发射理想 CPI)**

# Control Hazards

## 如何计算下一条指令地址（next PC）

- **无条件直接转移**
  - Opcode, PC, and offset
- **基于基址寄存器的无条件转移**
  - Opcode, Register value, and offset
- **条件转移**
  - Opcode, Register (for condition), PC and offset
- **其他指令**
  - Opcode and PC ( and have to know it’s not one of above )

# Control flow information in pipeline

# RISC-V Unconditional PC-Relative Jumps

*[ Kill bit turns instruction into a bubble ]*

**F**etch **D**ecode E**X**ecute

# Pipelining for Unconditional PC-Relative Jumps

```
j target   ;等同于 jal x0,offset
           ;       pc +=sext(offset)

target: add x1, x2, x3
```

# Branch Delay Slots

- 早期的 RISC 机器的延迟槽技术—改变 ISA 语义，在分支 / 跳转后的延迟槽中指令总是在控制流发生变化之前执行：
  - 0x100 j target
  - 0x104 add x1, x2, x3 // Executed before target
  - …
  - 0x205 target: xori x1, x1, 7
- 软件必须用有用的工作填充延迟槽（delay slots），或者用显式的 NOP 指令填充延迟槽

# Post-1990 RISC ISAs 取消了延迟槽

- **性能问题**
  - 当延迟槽中填充了 NOPs 指令后，增加了 I-cache 的失效率
  - 即使延迟槽中只有一个 NOP， I-cache 失效导致机器等待
- **使先进的微体系架构复杂化**
  - 例如 4 发射 30 段流水线
- **好的分支预测技术减少了采用延迟槽技术的动力**

# RISC-V Conditional Branches

# Pipelining for Conditional Branches

# Pipelining for Jump Register

- **Target address obtained in execute stage**

`jr x1` （等同于 `jalr x0,0(x1); pc=0(x1)`

*bubble*

*bubble*

`target: add x5, x6, x7`

# 为什么在经典的五段流水线中指令不能在每个周期都被分发 (CPI>1)

- **采用全定向路径可能代价太高而无法实现**
  - 通常提供经常使用的定向路径
  - 一些不经常使用的定向路径可能会增加时钟周期的长度，从而抵消降低 CPI 的好处
- **Load 操作有两个时钟周期的延迟**
  - Load 指令后的指令不能马上使用 Load 的结果
  - MIPS-I ISA 定义了延迟槽，软件可见的流水线冲突（由编译器调度无关的指令或插入 NOP 指令避免冲突），MIPS-II 中取消了延迟槽语义（硬件上增加流水线 interlocks 机制）
    - MIPS:“Microprocessor without Interlocked Pipeline Stages”
- **Jumps/Conditional branches 可能会导致流水线断流（bubbles)**
  - 如果没有延迟槽，则 stall 后续的指令

带有软件可见的延迟槽有可能需要执行大量的由编译器插入的 NOP 指令

NOP 指令降低了 CPI，但是增加了程序中执行的指令条数

# 陷阱和中断

**术语说明**

- **异常（Exception）**：由程序执行过程中引起的异常内部事件
  - E.g., page fault, arithmetic underflow
- **陷阱（Trap）**：因异常情况而被迫将控制权移交给监控程序
  - Not all exceptions cause traps (c.f. IEEE 754 floating-point standard)
- **中断（Interrupt）**：运行程序之外的外部事件，导致控制权转移到监控程序
- **陷阱和中断通常由同样的流水线机制处理**

# 异常处理的发展历史

- **第一个带有陷阱（traps）的系统是 Univac-I （1951 年）**
  - 算术运算溢出有两种选择
    - 1. 在地址 0 处触发一个两条指令的修复例程的执行 或者
    - 2. 根据程序员的选择，使计算机停止
  - 在后来的 Univac 1103, 1955，增加了外部中断机制
    - 用于实时采集风洞数据
- **第一个带有 I/O 中断的系统是 DYSEAC （1954）**
  - 带有两个程序计数器，并且 I/O 信号引起这两个 PC 间的切换
  - 它也是第一个带有 DMA (direct memory access by I/O device) 的系统
  - 同时，也是第一台移动计算机（两台半挂牵引车，12 tons + 8 tons)

# DYSEAC, first mobile computer!

- Carried in two tractor trailers, 12 tons + 8 tons
- Built for US Army Signal Corps

*[Courtesy Mark Smotherman]*

# 异步中断

- **I/O 设备通过发出中断请求信号来请求处理**
- **当处理器决定响应该中断时**
  - 停止当前指令 (Ii) 的执行 ， 执行完当前指令前面的指令（执行完 Ii-1 ） （ 精确中断）
  - 将 Ii 指令的 PC 值保存到专门寄存器 (EPC) 中
  - 关中断并将控制转移到以监控模式运行的指定的中断处理程序

# Interrupts:
## 改变正常的控制流

需要由另一个（系统）程序处理的外部或内部事件。从程序的角度来看，事件通常是意外的或罕见的。

# Interrupt Handler

- **允许嵌套中断时，在开中断之前需要保存 EPC ⇒**
  - 需要执行指令将 EPC 存入 GPRs
  - 至少在保存 EPC 之前，需要一种方法来暂时屏蔽进一步的中断
- **需要读取记录中断原因的状态寄存器**
- **使用专门的间接跳转指令 ERET (return-from-environment) 返回**
  - 开中断
  - 将处理器恢复到用户模式
  - 恢复硬件状态和控制状态

# Synchronous Trap

- **同步陷阱是由特定指令执行的异常引起的**
- **通常，该指令无法完成，需要在处理异常之后重新启动**
  - 需要撤消一个或多个部分执行的指令的效果
- **在系统调用（陷阱）的情况下，该指令被认为已经完成**
  - 一种特殊的跳转指令，涉及到切换到特权模式的操作

# Exception Handling 5-Stage Pipeline

- 如何处理不同流水段的多个并发异常？
- 如何以及在哪里处理外部异步中断？

# Exception Handling 5-Stage Pipeline

# Exception Handling 5-Stage Pipeline

- 在流水线中将异常标志保留到提交阶段
- 针对某一给定的指令，早期流水阶段中的异常覆盖该指令的后续异常

- **在提交阶段并入异步中断请求（覆盖其他中断）**

- **如果提交时检测到异常：**
  - 更新异常原因及 EPC 寄存器
  - 终止所有流水段，
  - 将异常处理程序的地址送入 PC 寄存器，以跳转到处理程序中执行

# 异常的推测

- **预测机制**
  - 异常总是比较少的，所以简单地预测为没有异常 通常是大概率事件
- **检查预测机制**
  - 在指令执行的最后阶段进行异常检测
  - 采用专门的硬件用于检测各种类型的异常
- **恢复执行机制**
  - 仅在提交阶段改变机器状态，因此可以在发生异常后丢弃部分执行的指令
  - 刷新流水线后启动异常处理程序
- **定向路径机制允许后续的指令使用没有提交的指令结果**

# 流水线的性能分析

- **基本度量参数：吞吐率，加速比，效率**
- **吞吐率：在单位时间内流水线所完成的任务数量或输出结果的数量。**

$$TP = \frac{n}{T_K}$$

$n$：任务数

$T_k$：处理完成 $n$ 个任务所用的时间

# 流水线技术提高系统的任务吞吐率

## 1. 各段时间均相等的流水线

– 各段时间均相等的流水线时空图

# 吞吐率

- 流水线完成 n 个连续任务所需要的总时间（假设一条 k 段线性流水线）

  $Tk = k\Delta t + (n-1)\Delta t = (k + n-1)\Delta t$

- 流水线的实际吞吐率

$$TP = \frac{n}{(k+n-1)\Delta t}$$

- 最大吞吐率：流水线在连续流动达到稳定状态后所得到的吞吐率。

$$TP_{\max} = \lim_{n\to\infty} \frac{n}{(k+n-1)\Delta t} = \frac{1}{\Delta t}$$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S4 | | | | 1 | 2 | 3 | 4 | 5 | .. | .. | .. | n-1 | n |
| S3 | | | 1 | 2 | 3 | 4 | 5 | .. | .. | .. | n-1 | n | |
| S2 | | 1 | 2 | 3 | 4 | 5 | .. | .. | .. | n-1 | n | | |
| S1 | 1 | 2 | 3 | 4 | 5 | .. | .. | .. | n-1 | n | | | |

$$TP = \frac{n}{k+n-1} TP_{\max}$$

# TP 与 Tpmax 的关系

– 最大吞吐率与实际吞吐率的关系

$$TP = \frac{n}{k + n - 1} TP_{\max}$$

- 流水线的实际吞吐率小于最大吞吐率，它除了与每个段的时间有关外，还与流水线的段数 k 以及输入到流水线中的任务数 n 有关。
- 只有当 n>>k 时，才有 TP≈TPmax 。

# 流水线中的瓶颈——最慢的段

## 2. 各段时间不完全相等的流水线

– 各段时间不等的流水线及其时空图
  - 一条 4 段的流水线
  - S1， S3， S4 各段的时间： Δt
  - S2 的时间： 3Δt （瓶颈段）
– 流水线中这种时间最长的段称为流水线的瓶颈段

– 各段时间不等的流水线的实际吞吐率：

（ Δti 为第 i 段的时间，共有 k 个段）

$$TP = \frac{n}{\sum_{i=1}^{k} \Delta t_i + (n-1)\max(\Delta t_1, \Delta t_2, \cdots, \Delta t_k)}$$

– 流水线的最大吞吐率为

$$TP_{\max} = \frac{1}{\max(\Delta t_1, \Delta t_2, \cdots \Delta t_k)}$$

– 例如：一条 4 段的流水线中， S1 ， S2 ， S4 各段的时间都是 Δt ，唯有 S3 的时间是 3Δt

最大吞吐率为

$$TP_{\max} = \frac{1}{3\Delta t}$$

# 3. 解决流水线瓶颈问题的常用方法

– 细分瓶颈段

例如：对前面的 4 段流水线

把瓶颈段 $S_3$ 细分为 3 个子流水线段： $S_{3a}$， $S_{3b}$， $S_{3c}$

**改进后的流水线的吞吐率：**

$$TP_{\max} = \frac{1}{\Delta t}$$

– 重复设置瓶颈段

- 缺点：控制逻辑比较复杂，所需的硬件增加了。
- 例如：对前面的 4 段流水线
- 重复设置瓶颈段 S3： S3a， S3b， S3c

重复设置瓶颈段后的时空图

# 加速比

- **加速比：完成同样一批任务，不使用流水线所用的时间与使用流水线所用的时间之比**
  - 假设：不使用流水线（即顺序执行）所用的间为 Ts，使用流水线后所用的时间为 Tk，则该流水线的加速比为

$$S = \frac{T_s}{T_k}$$

## 1. 流水线各段时间相等（都是 Δt）

- k 段流水线完成 n 个连续任务所需要的时间为

$$T_k = (k + n-1)\Delta t$$

- 顺序执行 n 个任务所需要的时间

$$T_s = nk\Delta t$$

- 流水线的实际加速比为

$$S = \frac{nk}{k+n-1}$$

- 最大加速比

$$S_{max} = \lim_{n\to\infty} \frac{nk}{k+n-1} = k$$

- **当 $n>>k$ 时，$S \approx k$**

**思考：流水线的段数愈多愈好?**

## 2. 流水线的各段时间不完全相等时

– 一条 $k$ 段流水线完成 $n$ 个连续任务的实际加速比为

$$S=\frac{n\sum_{i=1}^{k}\Delta t_i}{\sum_{i=1}^{k}\Delta t_i+(n-1)\max(\Delta t_1,\Delta t_2,\cdots,\Delta t_k)}$$

# 效率

- **效率：流水线中的设备实际使用时间与整个运行时间的比值，即流水线设备的利用率 utilization 。**
  - 由于流水线有通过时间和排空时间，所以在连续完成 n 个任务的时间内，各段并不是满负荷地工作。
- **各段时间相等**
  - 各段的效率 ei 相同

$$e_1 = e_2 = \cdots = e_k = \frac{n\Delta t}{T_k} = \frac{n}{k+n-1}$$

– 整条流水线的效率为

$$E=\frac{e_1+e_2+\cdots+e_k}{k}=\frac{ke_1}{k}=\frac{kn\Delta t}{kT_k}$$

可以写成

$$E=\frac{n}{k+n-1}$$

最高效率为

$$E_{\max}=\lim_{n\to\infty}\frac{n}{k+n-1}=1$$

**当 *n*>>*k* 时， *E*≈1 。**

– 当流水线各段时间相等时，流水线的效率与吞吐率成正比。

E=TP△t

$$E=\frac{n}{k+n-1}$$

$$TP=\frac{n}{(k+n-1)\Delta t}$$

- **流水线的效率是流水线的实际加速比 S 与它的最大加速比 k 的比值。**

$$S=\frac{nk}{k+n-1}$$

$$E=\frac{S}{k}$$

**当 *E*=1 时， *S*=*k* ，实际加速比达到最大。**

- **从时空图上看，效率就是 n 个任务占用的时空面积和 k 个段总的时空面积之比。**

$$E = \frac{n\text{个任务实际占用的时空区}}{k\text{个段总的时空区}}$$

当各段时间不相等时

$$E = \frac{n \cdot \sum_{i=1}^{k} \Delta t_i}{k\left[\sum_{i=1}^{k} \Delta t_i + (n-1) \cdot \max(\Delta t_1, \Delta t_2, \cdots, \Delta t_k)\right]}$$

# Summary

- 实际吞吐率：假设 k 段，完成 n 个任务，单位时间所实际完成的任务数。
- 加速比：k 段流水线的速度与等功能的非流水线的速度之比。
- 效率：流水线的设备利用率。

# -Review: Pipelining

- **指令流水线通过指令重叠减小 CPI**
- **充分利用数据通路**
  - 当前指令执行时，启动下一条指令
  - 其性能受限于花费时间最长的段
  - 检测和消除相关
- **如何有利于流水线技术的应用**
  - 所有的指令都等长
  - 只有很少的指令格式
  - 只用 Load/Store 来进行存储器访问

# 流水线的加速比计算

$$CPI_{pipelined} = Ideal\ CPI + Average\ Stall\ cycles\ per\ Inst$$

$$Speedup = \frac{Ideal\ CPI \times Pipeline\ depth}{Ideal\ CPI + Pipeline\ stall\ CPI} \times \frac{Cycle\ Time_{unpipelined}}{Cycle\ Time_{pipelined}}$$

*For simple RISC pipeline, CPI = 1:*

$$Speedup = \frac{Pipeline\ depth}{1 + Pipeline\ stall\ CPI} \times \frac{Cycle\ Time_{unpipelined}}{Cycle\ Time_{pipelined}}$$

# 结构相关对性能的影响

- **例如： 如果每条指令平均访存 1.3 次，而每个时钟周期只能访存一次，那么**
  - 在其他资源 100 %利用的前提下，平均 CPI ≥ 1.3

# 例如： Dual-port vs. Single-port

- **机器 A: Dual ported memory (“Harvard Architecture”)**
- **机器 B: Single ported memory**
- **存在结构相关的机器 B 的时钟频率是机器 A 的时钟频率的 1.05 倍**
- **Ideal CPI = 1**
- **在机器 B 中 load 指令会引起结构相关，所执行的指令中 Loads 指令占 40%**

**Average instruction time = CPI * Clock cycle time**

**无结构相关的机器 A：**

Average Instruction time = Clock cycle time

**存在结构相关的机器 B：**

Average Instruction time = (1+0.4*1) * clock cycle time /1.05
= 1.3 * clock cycle time

# 解决控制相关的方法

- **#1: Stall 直到分支方向确定**
- **#2: 预测分支失败**
  - 直接执行后继指令
  - 如果分支实际情况为分支成功，则撤销流水线中的指令对流水线状态的更新
  - 要保证：分支结果出来之前不会改变处理机的状态，以便一旦猜错时，处理机能够回退到原先的状态。
- **#3: 预测分支成功**
  - 前提：先知道分支目标地址，后知道分支是否成功
- **#4: 延迟转移技术**

# 评估减少分支策略的效果

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

| *Scheduling penalty* | *Branch penalty* | *CPI unpipelined* | *speedup v. stall* | *speedup v. scheme* |
|---|---|---|---|---|
| Stall pipeline | 3 | 3.5 | 1.42 | |
| Predict taken | 1 | 4.4 | 1.14 | |
| Predict not taken | 1 | 4.1 | 1.09 | 1.29 |
| Delayed branch | 0.5 | 4.0 | 1.07 | 1.31 |

1.14 = 1 + 1*14%*100%

1.09 = 1+1*14%*65%

1.07 = 1+ 0.5*14%

Conditional & Unconditional = 14%, 65% change PC

# 多周期操作的处理

**对整型 pipeline 增加浮点数处理**

**RISC-V**

# 多周期操作的处理

- **问题**
  - 浮点操作在 1 ～ 2 个 cycles 完成是不现实的，一般要花费较长时间
  - 在 MIPS 中如何处理
- **现假设 FP 指令与整数指令采用相同的流水线，那么**
  - EX 段需要循环多次来完成 FP 操作，循环次数取决于操作类型
  - 有多个 FP 功能部件，如果发射出的指令导致结构或数据相关，需暂停

# 对 MIPS 的扩充

四个功能部件

- **Integer** 部件处理：**Loads, Store, Integer ALU** 操作和 **Branch**
- **FP/Integer** 乘法部件：处理浮点数和整数乘法
- **FP** 加法器：处理 **FP** 加，减和类型转换
- **FP/Integer** 除法部件：处理浮点数和整数除法

# 假设这些部件没有流水线化

**整数流水线和浮点流水线分开**

# Latency & Repeat Interval

- **延时 (Latency)**
  - If latency=x, then (x+1)th instruction can use the result
- **循环间隔（Repeat/Initiation interval）**
  - 发射相同类型的操作所需的间隔周期数
- **对于 EX 部件流水化的新的 MIPS**

| **Function Unit** | **Latency** | **Repeat Interval** |
|---|---|---|
| Integer ALU | **0** | **1** |
| Data Memory (Integer and FP loads(1 less for store latency)) | **1** | **1** |
| **FP Add** | **3** | **1** |
| **FP multiply** | **6** | **1** |
| **FP Divide (also integer divide and FP sqrt)** | **24** | **25** |

# 将部分执行部件流水化后的MIPS流水线

# 新的相关和定向问题

- **结构冲突增多**
  - 非流水的 Divide 部件，使得 EX 段增长 24 个 cycles
  - 在一个周期内可能有多个寄存器写操作
- **可能指令乱序完成（乱序到达 WB 段）有可能存在 WAW**
- **由于在 ID 段读，还不会有 WAR 相关**
- **乱序完成导致异常处理复杂**
- **由于指令的延迟加大导致 RAW 相关的 stall 数增多**
- **需要付出更多的代价来增加定向路径**

# 新的结构相关

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MULTD F0, F4, F6 | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB |
| ... | | IF | ID | EX | MEM | WB | | | | | |
| ... | | | IF | ID | EX | MEM | WB | | | | |
| ADDD F2, F4, F6 | | | | IF | ID | A1 | A2 | A3 | A4 | MEM | WB |
| ... | | | | | IF | ID | EX | MEM | WB | | |
| ... | | | | | | IF | ID | EX | MEM | WB | |
| LD F8, 0(R2) | | | | | | | IF | ID | EX | MEM | WB |

- **纵向检查指令所使用的资源**
  - 第 10 个 cycle，三条指令同时进入 MEM，但由于 MULTD 和 ADDD 在 MEM 段没有实际动作，这种情况没有关系
  - 第 11 个 cycle，三条指令同时进入 WB 段，存在结构相关

# 解决方法

- **Option 1**
  - 在 ID 段跟踪写端口的使用情况，以便能暂停该指令的发射
  - 一旦发现冲突，暂停当前指令的发射
- **Option 2**
  - 在进入 MEM 或 WB 段时，暂停冲突的指令，让有较长延时的指令先做．这里假设较长延时的指令，可能会更容易引起其他 RAW 相关，从而导致更多的 stalls

# 关于数据相关

| | Clock cycle number | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| L.D F4,0(R2) | IF | ID | EX | MEM | WB | | | | | | | | | | | | |
| MUL.D F0,F4,F6 | | IF | ID | stall | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB | | | | |
| ADD.D F2,F0,F8 | | | IF | stall | ID | stall | stall | stall | stall | stall | stall | A1 | A2 | A3 | A4 | MEM | WB |
| S.D F2,0(R2) | | | | | IF | stall | stall | stall | stall | stall | stall | ID | EX | stall | stall | stall | MEM |

S.D 多延迟一个 cycle， 以消解与 ADD.D 的冲突

# 新的冲突源

- **GPR 与 FPR 间的数据传送造成的数据相关**
  - MOVI2FP and MOVFP2I instructions
- **如果在 ID 段进行相关检测，指令发射前须做如下检测：**
  - 结构相关
    - 循环间隔检测
    - 确定寄存器写端口是否可用
  - RAW 相关
    - 列表所有待写的目的寄存器
    - 不发射以待写寄存器做为源寄存器的指令，插入 latency 个 stall
  - WAW 相关
    - 仍然使用上述待写寄存器列表
    - 不发射那些目的寄存器与待写寄存器列表中的指令有 WAW 冲突的指令，延迟 1 个 cycle 发射。

# 精确中断与长流水线

- **例如**
  - DIVF F0,F2,F4
  - ADDF F10,F10,F8
  - SUBF F12,F12,F14
- **ADDF 和 SUBF 都在 DIVF 前完成**
- **如果 DIVF 导致异常，会如何?**
  - 非精确中断
- **Ideas???**

# 精确中断与非精确中断

- **精确中断**
  - 如果流水线可以控制使得引起异常的指令前序指令都执行完，故障后的指令可以重新执行，则称该流水线支持精确中断
  - 按照指令的逻辑序处理异常
  - 理想情况，引起故障的指令没有改变机器的状态
  - 要正确的处理这类异常请求，必须保证故障指令不产生副作用
- **在有些机器上，浮点数异常**
  - 流水线段数多，在发现故障前，故障点后的指令就已经写了结果，在这种情况下，必须有办法处理。
  - 很多高性能计算机，Alpha 21164，MIPS R10000等支持精确中断，但精确模式要慢10+倍，一般用在代码调试时，很多系统要求精确中断模式，如IEEE FP标准处理程序，虚拟存储器等。
- **精确中断对整数流水线而言，不是太难实现**
  - 指令执行的中途改变机器的状态
  - 例如IA-32 的自动增量寻址模式

# MIPS 中的异常

- **IF**
  - page fault, misaligned address, memory protection violation
- **ID**
  - undefined or illegal opcode
- **EX**
  - arithmetic exception
- **MEM**
  - page fault, misaligned address, memory protection violation
- **WB**
  - none

# 处理中断 4 种可能的办法

- **方法 1： 忽略这种问题，当非精确处理**
  - 原来的 supercomputer 的方法
  - 但现代计算机对 IEEE 浮点标准的异常处理，虚拟存储的异常处理要求必须是精确中断。
- **方法 2： 缓存操作结果，直到早期发射的指令执行完。**
  - 当指令运行时间较长时， Buffer 区较大
  - Future file（Power PC620 MIPS R10000)
    - 缓存执行结果，按指令序确认
  - history file（CYBER 180/990)
    - 尽快确认
    - 缓存区存放原来的操作数，如果异常发生，回卷到合适的状态

# 第 3 & 4 种方法

- **方法 3： 以非精确方式处理，用软件来修正**
  - 为软件修正保存足够的状态
  - 让软件仿真尚未执行完的指令的执行
  - 例如
    - Instruction 1 – A 执行时间较长，引起中断的指令
    - Instruction 2, instruction 3, ….instruction n-1 未执行完的指令
    - Instruction n 已执行完的指令
    - 由于第 n 条指令已执行完，希望中断返回后从第 n+1 条指令开始执行，如果我们保存所有的流水线的 PC 值，那么软件可以仿真 Instruction1 到 Instruction n-1 的执行
- **方法 4： 暂停发射，直到确定先前的指令都可无异常的完成，再发射下面的指令。**
  - 在 EX 段的前期确认（MIPS 流水线在前三个周期中）
  - MIPS R2K to R4K 以及 Pentium 使用这种方法

# MIPS 流水线的性能

Stalls per FP operation for each major type of FP operation for the SPEC89 FP benchmarks

# 平均每条指令的 stall 数

The stalls occurring for the MIPS FP pipeline for five for the SPEC89 FP benchmarks.

# MIPS R4000 （1991）

- **实际的 64-bit 机器**
  - 主频 100MHz ~200MHz
  - 较深的流水线（级数较多）（有时也称为 superpipelining）

# MIPS R4000 的 8 级整数流水线

- IF– 取指阶段前半部分；选择 PC 值，初始化指令 cache 的访问
- IS– 取指阶段后半部分，主要完成访问指令 cache 的操作
- RF– 指令译码，寄存器读取，相关检测以及指令 cache 命中检测
- EX– 执行，包括：计算有效地址，进行 ALU 操作，计算分支目标地址和检测分支条件
- DF– 取数据，访问数据 cache 的前半部分
- DS– 访问数据 cache 的后半部分
- TC–tag 检测，确定数据 cache 是否命中
- WB–Load 操作和 R-R 操作的结果写回

# 需注意的问题

- **在使用定向技术的情况下，Load 延迟为 2 个 cycles**
  - Load 和与其相关的指令间必须有 2 条指令或两个 bubbles
  - 原因：load 的结果在 DS 结束时可用
- **分支延迟 3 个 cycles**
  - 分支与目标指令间需要 3 条指令或 3 个 bubbles
  - 原因：目标地址在 EX 段后才能知道
- **R4000 的流水线中，到 ALU 输入端有四个定向源**
  - EX/DF, DF/DS, DS/ TC, TC/WB

# 图示

**TWO Cycle Load Latency**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| IF | IS | RF | EX | DF | DS | TC | WB |
| | IF | IS | RF | EX | DF | DS | TC |
| | | IF | IS | RF | EX | DF | DS |
| | | | IF | IS | RF | EX | DF |
| | | | | IF | IS | RF | EX |
| | | | | | IF | IS | RF |
| | | | | | | IF | IS |
| | | | | | | | IF |

**THREE Cycle Branch Latency**
(conditions evaluated during EX phase)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| IF | IS | RF | EX | DF | DS | TC | WB |
| | IF | IS | RF | EX | DF | DS | TC |
| | | IF | IS | RF | EX | DF | DS |
| | | | IF | IS | RF | EX | DF |
| | | | | IF | IS | RF | EX |
| | | | | | IF | IS | RF |
| | | | | | | IF | IS |
| | | | | | | | IF |

**Delay slot plus two stalls**
**Branch likely cancels delay slot if not taken**

# ALU 输入端的定向源

R4000的流水线中ALU输入有四个定向源
EX/DF
DF/DS
DS/TC
TC/WB
IF
IS
RF
EX
DF
DS
TC
WB
指令存储器
Reg
ALU
数据存储器
Reg

# MIPS R4000 浮点数操作

- **3个功能部件组成： FP Adder, FP Multiplier, FP Divider**
- **在乘/除操作的最后一步要使用FP Adder**
- **FP操作需要2（negate）-112个（square root）cycles**

- **8 种类型的FP units:**
  - Stage Functional unit Description
  - A FP adder Mantissa ADD stage 尾数
  - D FP divider Divide pipeline stage
  - E FP multiplier Exception test stage
  - M FP multiplier First stage of multiplier
  - N FP multiplier Second stage of multiplier
  - R FP adder Rounding stage
  - S FP adder Operand shift stage
  - U Unpack FP numbers

Each Stage has one single copy

| 流水段 | 功能部件 | 描　　述 |
| --- | --- | --- |
| A | 浮点加法器 | 尾数加流水段 |
| D | 浮点除法器 | 除法流水段 |
| E | 浮点乘法器 | 例外测试段 |
| M | 浮点乘法器 | 乘法器第一个流水段 |
| N | 浮点乘法器 | 乘法器第二个流水段 |
| R | 浮点加法器 | 舍入段 |
| S | 浮点加法器 | 操作数移位段 |
| U | | 展开浮点数 |

# 双精度浮点数操作延迟及初始化间隔

| 浮点指令 | 初始化间隔 | 延迟 | 使用的流水段 |
|---|---|---|---|
| 加、减 | 4 | 3 | U,S+A,A+R,R+S |
| 乘 | 8 | 4 | U,E+M,M,M,M,N,N+A,R |
| 除 | 36 | 35 | U,A,R,$D^{28}$,D+A,D+R,D+A,D+R,A,R |
| 求平方根 | 112 | 111 | U,E,$(A+R)^{108}$,A,R |
| 取反 | 2 | 1 | U,S |
| 求绝对值 | 2 | 1 | U,S |
| 浮点比较 | 3 | 2 | U,A,R |

# MIPS FP 流水段

| *FP Instr* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *…* |
|---|---|---|---|---|---|---|---|---|---|
| Add, Subtract | U | S+A | A+R | R+S | | | | | |
| Multiply | U | E+M | M | M | M | N | N+A | R | |
| Divide | U | A | R | $D^{28}$ … | | | D+A , | D+R, D+R, D+A, D+R, A, R | |
| Square root | U | E | $(A+R)^{108}$ | … | | | AR | | |
| Negate | U | S | | | | | | | |
| Absolute value | U | S | | | | | | | |
| FP compare | U | A | R | | | | | | |

*Stages:*

| | |
|---|---|
| *M* | *First stage of multiplier* |
| *N* | *Second stage of multiplier* |
| *R* | *Rounding stage* |
| *S* | *Operand shift stage* |
| *U* | *Unpack FP numbers* |

| | |
|---|---|
| ***A*** | ***Mantissa ADD stage*** |
| ***D*** | ***Divide pipeline stage*** |
| ***E*** | ***Exception test stage*** |

| | | Clock cycle | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Operation | Issue/stall | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Multiply | Issue | U | E+M | M | M | M | N | N+A | R | | | | | |
| Add | Issue | | U | S+A | A+R | R+S | | | | | | | | |
| | Issue | | | U | S+A | A+R | R+S | | | | | | | |
| | Issue | | | | U | S+A | A+R | R+S | | | | | | |
| | Stall | | | | | U | S+A | A+R | R+S | | | | | |
| | Stall | | | | | | U | S+A | A+R | R+S | | | | |
| | Issue | | | | | | | U | S+A | A+R | R+S | | | |
| | Issue | | | | | | | | U | S+A | A+R | R+S | | |

| | | Clock cycle | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Operation** | **Issue/stall** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| Add | Issue | U | S+A | A+R | R+S | | | | | | | | | |
| Multiply | Issue | | U | M | M | M | M | N | N+A | R | | | | |
| | Issue | | | **E+M** U | M | M | M | M | N | N+A | R | | | |

| | | Clock cycle | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Operation** | **Issue/stall** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** | **35** | **36** |
| Divide | issued in cycle 0... | D | D | D | D | D | D+A | D+R | D+A | D+R | A | R | |
| Add | Issue | | U | S+A | A+R | R+S | | | | | | | |
| | Issue | | | U | S+A | A+R | R+S | | | | | | |
| | Stall | | | | U | S+A | A+R | R+S | | | | | |
| | Stall | | | | | U | S+A | A+R | R+S | | | | |
| | Stall | | | | | | U | S+A | A+R | R+S | | | |
| | Stall | | | | | | | U | S+A | A+R | R+S | | |
| | Stall | | | | | | | | U | S+A | A+R | R+S | |
| | Stall | | | | | | | | | U | S+A | A+R | R+S |
| | Issue | | | | | | | | | | U | S+A | A+R |
| | Issue | | | | | | | | | | | U | S+A |
| | Issue | | | | | | | | | | | | U |

| | | Clock cycle | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Operation | Issue/stall | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Add | Issue | U | S+A | A+R | R+S | | | | | | | | | |
| Divide | Stall | | U | A | R | D | D | D | D | D | D | D | D | D |
| | Issue | | | U | A | R | D | D | D | D | D | D | D | D |
| | Issue | | | | U | A | R | D | D | D | D | D | D | D |

# R4000 性能（1）

Figure A.48 The pipeline CPI for 10 of the SPEC92 benchmarks, assuming a perfect cache. The pipeline CPI varies from 1.2 to 2.8. The leftmost five programs are integer programs, and branch delays are the major CPI contributor for these. The rightmost five programs are FP, and FP result stalls are the major contributor for these. Figure A.49 shows the numbers used to construct this plot.

# R4000 性能（2）

| Benchmark | Pipe CPI | Load Stalls | Branch Stalls | FP Res. Stalls | FP Struc. Stalls |
|---|---|---|---|---|---|
| compress | 1.20 | 0.14 | 0.06 | 0.00 | 0.00 |
| eqntott | 1.88 | 0.27 | 0.61 | 0.00 | 0.00 |
| espresso | 1.42 | 0.07 | 0.35 | 0.00 | 0.00 |
| gcc | 1.56 | 0.13 | 0.43 | 0.00 | 0.00 |
| li | 1.64 | 0.18 | 0.46 | 0.00 | 0.00 |
| INTEGER AVERAGE | 1.54 | 0.16 | 0.39 | 0.00 | 0.00 |
| doduc | 2.84 | 0.01 | 0.22 | 1.39 | 0.22 |
| mkljdp2 | 2.66 | 0.01 | 0.31 | 1.20 | 0.15 |
| ear | 2.17 | 0.00 | 0.46 | 0.59 | 0.12 |
| hydro2d | 2.53 | 0.00 | 0.62 | 0.75 | 0.17 |
| su2cor | 2.18 | 0.02 | 0.07 | 0.84 | 0.26 |
| FP AVERAGE | 2.48 | 0.01 | 0.33 | 0.95 | 0.18 |
| OVERALL AVERAGE | 2.00 | 0.10 | 0.36 | 0.46 | 0.09 |

# 基本流水线小结

- **流水线提高的是指令带宽（吞吐率），而不是单条指令的执行速度**
- **相关限制了流水线性能的发挥**
  - 结构相关：需要更多的硬件资源
  - 数据相关：需要定向，编译器调度
  - 控制相关：尽早检测条件，计算目标地址，延迟转移，预测
- **增加流水线的级数会增加相关产生的可能性**
- **异常，浮点运算使得流水线控制更加复杂**
- **编译器可降低数据相关和控制相关的开销**
  - Load 延迟槽
  - Branch 延迟槽
  - Branch 预测

# Review

- **流水线技术并不能提高单个任务的执行效率，它可以提高整个系统的吞吐率**
  - 多个任务同时执行，但使用不同的资源
- **流水线性能分析：吞吐率、加速比、效率**
  - 流水线中的瓶颈——最慢的那一段
  - 其潜在的加速比＝流水线的级数
  - 流水段所需时间不均衡将降低加速比
  - 流水线存在装入时间和排空时间，使得加速比降低
- **由于存在相关问题，会导致流水线停顿**
  - 结构相关、数据相关和控制相关

# Acknowledgements


- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - David Patterson (UCB)
  - Chenxi Zhang (Tongji)
- **UCB material derived from course CS152、 CS252、 CS61C**
- **KFUPM material derived from course COE501、 COE502**

# 计算机体系结构

Topic IV: Memory Hierarchy

# 第4章　存储层次结构设计

- 存储层次结构
- Cache基本知识
- 基本的Cache优化方法
- 高级的Cache优化方法
- 存储器技术与优化
- 虚拟存储器 - 基本原理

# 存储层次结构

- **存储系统设计是计算机体系结构设计的关键问题之一**
  - 价格，容量，速度的权衡
- **用户对存储器的“容量，价格和速度”要求是相互矛盾的**
  - 速度越快，每位价格就高
  - 容量越大，每位价格就低
  - 容量越大，速度就越慢
  - 目前主存一般由DRAM构成
- **Microprocessor与Memory之间的性能差异越来越大**
  - CPU性能提高大约60%/year
  - DRAM 性能提高大约 9%/year

# Trends in DRAM

| Year Produced | Chip Size | DRAM Type | Bus Cycle | Latency New Request |
|---|---|---|---|---|
| 1980 | 64 Kbit | Asynchronous DRAM | | 250 ns |
| 1983 | 256 Kbit | Asynchronous DRAM | | 190 ns |
| 1986 | 1 Mbit | Asynchronous DRAM | | 140 ns |
| 1989 | 4 Mbit | Asynchronous DRAM | | 110 ns |
| 1992 | 16 Mbit | Asynchronous DRAM | | 90 ns |
| 1996 | 64 Mbit | SDRAM | 10 ns | 70 ns |
| 1998 | 128 Mbit | SDRAM | 7.5 ns | 60 ns |
| 2000 | 256 Mbit | DDR SDRAM | 6 ns | 55 ns |
| 2002 | 512 Mbit | DDR SDRAM | 5 ns | 55 ns |
| 2004 | 1 Gbit | DDR2 SDRAM | 3 ns | 50 ns |
| 2006 | 2 Gbit | DDR2 SDRAM | 2 ns | 45 ns |
| 2010 | 4 Gbit | DDR3 SDRAM | 1.5 ns | 37 ns |
| 2012 | 8 Gbit | DDR3 SDRAM | 1 ns | 31 ns |

# 微处理器与DRAM 的性能差异

## Processor-DRAM Memory Gap (latency)

# Microprocessor-DRAM性能差异

- **利用caches缓解微处理器与存储器性能上的差异**
- **Microprocessor-DRAM 性能差异**
  - time of a full cache miss in instructions executed
  - 1st Alpha : 340 ns/5.0 ns = 68 clks x 2 or 136 instructions
  - 2nd Alpha :266 ns/3.3 ns = 80 clks x 4 or 320 instructions
  - 3rd Alpha :180 ns/1.7 ns =108 clks x 6 or 648 instructions

# 存储系统的设计目标

*通过优化存储系统的组织来使得针对典型应用平均访存时间最短*

# 基本解决方法：多级层次结构

- **多级分层结构**

  - M1 速度最快，容量最小，每位价格最高
  - Mn速度最慢，容量最大，每位价格最低
- **并行**
- **存储系统接近M1的速度，容量和价格接近Mn**

# 计算机系统的多级存储层次

Server

| | | | | | |
|---|---|---|---|---|---|
| 300ps | 1ns | 3-10ns | 10-20ns | 50-100ns | 5-10ms |
| 1000B | 64KB | 256K | 2-4MB | 4-16GB | 4-16TB |

Personal Mobile Device

| | | | | |
|---|---|---|---|---|
| 500ps | 2ns | 10-20ns | 50-100ns | 25-50 μ s |
| 500B | 64KB | 256K | 256-512MB | 4-8GB |

# 典型的存储器访问模式

Address
n loop iterations
Instruction fetches
subroutine call
subroutine return
Stack accesses
argument access
vector access
Data accesses
scalar accesses
Time

# 存储层次工作原理： Locality!

- **应用程序局部性原理: 给用户**
  - 一个采用低成本技术达到的存储容量. （容量大，价格低）
  - 一个采用高速存储技术达到的访问速度. （速度快）
- **Temporal Locality (时间局部性):**
  - =>保持最近访问的数据项最接近微处理器
- **Spatial Locality (空间局部性):**
  - 以由地址连续的若干个字构成的块为单位，从低层复制到上一层

# 存储层次结构涉及的基本概念

- **Block**
  - Block （块）：不同层次的Block大小可能不同
  - 命中（hit）和命中率（hit rate）
  - 失效（miss）和失效率（miss rate）
- **镜像和一致性问题 (inclusion and consistency)**
  - 高层存储器是较低层存储器的一个镜像
  - 高层存储器内容的修改必须反映到低层存储器中
    - 数据一致性问题
- **寻址：不管如何组织，我们必须知道如何访问数据**
- **要求：不同层次上块大小可以不同**
  - 在L0 cache 可能以Double, Words, Halfwords, 或bytes
  - 在L1cache仅以cache line 或 slot为单位访问
  - 在更低层…..
  - 因此总是存在地址映射问题
  - 物理地址格式　Block Address + Block Offset

# 存储层次的性能参数(1/2)

- **假设采用二级存储：M1和M2**
  - M1和M2的容量（size)、价格(cost)、访问时间(time to access)分别为:
  - S1 、C1、TA1 S2、C2、TA2

# 存储层次的性能参数 (2/2)

- **存储层次的平均每位价格C**
  - C=(C1*S1+C2*S2)/(S1+S2)
- **命中(Hit): 访问的块在存储系统的较高层次上**
  - 若一组程序对存储器的访问，其中N1次在M1中找到所需数据，N2次在M2中找到数据 则
  - Hit Rate (命中率) : 存储器访问在较高层命中的比例 H= N1/(N1+N2)
  - Hit Time (命中时间) :访问较高层的时间，TA1
- **失效(Miss):访问的块不在存储系统的较高层次上**
  - Miss Rate (失效) = 1 - (Hit Rate) = 1 - H = N2/(N1+N2)
  - 当在M1中没有命中时：一般必须从M2中将所访问的数据所在块搬到M1中，然后CPU才能在M1中访问。
  - 设传送一个块的时间为TB,即不命中时的访问时间为：TA2+TB+TA1 = TA1+TM
  - TM 通常称为失效开销
- **平均访存时间:**
  - 平均访存时间 TA = H*TA1+(1-H)*(TA1+TM) = TA1+(1-H)*TM

# 常见的存储层次的组织

- **Registers <-> Memory**
  - 由编译器完成调度
- **cache <-> memory**
  - 由硬件完成调度
- **memory <-> disks**
  - 由硬件和操作系统（虚拟管理）
  - 由程序员完成调度

# Cache Memory

- **小而快（SRAM）的存储技术**
  - 存储正在访问的部分指令和数据
- **用于减少平均访存时间**
  - 通过保持最近访问的数据在处理器附近来挖掘时间局部性
  - 通过以块为单位在不同层次移动数据来挖掘空间局部性
- **主要目标：**
  - 提高访存速度
  - 降低存储系统成本

# Cache 无处不在

- **体系结构中，Cache无处不在**
- **寄存器：Cache on variables**
- **一、二级Cache： Cache on memory**
- **Memory：Cache on hard disks**
  - 存储最近执行的程序和数据
  - Hard disks可以视为主存的扩展（VM）
- **分支目标缓存(branch target buffer)及分支预测缓存(branch prediction buffer)**
  - 缓存分支目标及预测信息

# Cache基本知识

## Sample Memory Hierarchy

# Q1：映象(mapping)规则

- **当要把一个块从主存调入Cache时，如何放置问题**
- **三种方式**
  - 全相联方式（Fully Associative)：即所调入的块可以放在cache中的任何位置
  - 直接映象方式（Direct Mapped)：主存中每一块只能存放在cache中的唯一位置。 一般，主存块地址i 与cache中块地址j 的关系为：

    j = i mod (M)，M为cache中的块数
  - 组相联映象(Set Associative)：主存中每一块可以被放置在Cache中唯一的一个组(set)中的任意一个位置，组由若干块构成，若一组由n块构成，我们称N路组相联
    - 组间直接映象
    - 组内全相联
    - 若cache中有G组，则主存中的第i 块的组号K为 K = i mod (G)，

# Q1: Where can a block be placed in the upper level?

- **Block 12 placed in 8 block cache:**
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets

Fully associative: block 12 can go anywhere

Direct mapped: block 12 can go only into block 4 (12 mod 8)

Set associative: block 12 can go anywhere in set 0 (12 mod 4)

# Q1的讨论

- **N-Way组相联：如果每组由N个块构成，cache的块数为M，则cache的组数G为M/N**
- **不同相联度下的路数和组数**

| | 路数 | 组数 |
|---|---|---|
| **全相联** | **M** | **1** |
| **直接相联** | **1** | **M** |
| **其他组相联** | **1 < N <M** | **1 < G < M** |

  - 相联度越高，cache空间利用率就越高，块冲突概率就越小，失效率就越低
  - N值越大，失效率就越低，但Cache的实现就越复杂，代价越大
  - 现代大多数计算机都采用直接映象，两路或四路组相联。

# Q2(1/2): 查找方法

- **在CACHE中每一block都带有tag域（标记域），标记分为两类**
  - Address Tags：标记所访问的单元在哪一块中，这样物理地址就分为三部分： Address Tags ## Block index## block Offset
  - 全相联映象时，没有Block Index
  - 显然 Address tag越短，查找所需代价就越小
  - Status Tags：标记该块的状态，如Valid，Dirty等

# Q2（2/2)查找方法

- **原则：所有可能的标记并行查找，cache的速度至关重要，即并行查找**
- **并行查找的方法**
  - 用相联存储器实现，按内容检索
  - 用单体多字存储器和比较器实现
- **显然相联度 N越大，实现查找的机制就越复杂，代价就越高**
- **无论直接映象还是组相联，查找时，只需比较 tag，index无需参加比较**

# Tag和数据阵列并行访问的逻辑结构

**FIGURE 2.1:** High-level logical cache organization.

# Tag 和数据阵列并行访问的流水线模式

**FIGURE 2.2:** Parallel tag and data array access pipeline.

# Tag和数据阵列串行访问的逻辑结构

**FIGURE 2.3:** High-level logical cache organization with serial tag and data array access.

# Tag 和数据阵列串行访问的流水线模式

**FIGURE 2.4:** Serial tag and data array access pipeline.

# 直接映像Cache查找过程

Enable write access to data block if cache hit

Aligner shifts data within word according to byte offset

# 全相联Cache查找过程

# Example: 1 KB Direct Mapped Cache with 32 B Blocks

- **对于容量为 $2^N$ 字节的cache:**
  - 最高(32-N)位部分 为 Cache Tag
  - 最低M位为字节选择位 (Block Size = $2^M$)

# Example: Set Associative Cache

- **N-way set associative: 每一个cache索引对应N个cache entries**
  - 这N个cache项并行操作
- **Example: Two-way set associative cache**
  - Cache index 选择cache中的一组
  - 这一组中的两块对应的Tags与输入的地址同时比较
  - 根据比较结果选择数据

# Q3：替换算法

- **主存中块数一般比cache中的块多，可能出现该块所对应的一组或一个Cache块已全部被占用的情况，这时需强制腾出其中的某一块，以接纳新调入的块，替换哪一块，这是替换算法要解决的问题：**
  - 直接映象，因为只有一块，别无选择
  - 组相联和全相联有多种选择
- **替换方法**
  - 随机法（Random），随机选择一块替换
    - 优点：简单，易于实现
    - 缺点：没有考虑Cache块的使用历史，反映程序的局部性较差，失效率较高
  - FIFO - 选择最早调入的块
    - 优点：简单
    - 虽然利用了同一组中各块进入Cache的顺序，但还是反映程序局部性不够，因为最先进入的块，很可能是经常使用的块
  - 最近最少使用法（LRU）（Least Recently Used）
    - 优点：较好的利用了程序的局部性，失效率较低
    - 缺点：比较复杂，硬件实现较困难

# LRU和Random的比较（失效率）

| | Associativity | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Two-way | | | Four-way | | | Eight-way | | |
| Size | LRU | Random | FIFO | LRU | Random | FIFO | LRU | Random | FIFO |
| 16 KB | 114.1 | 117.3 | 115.5 | 111.7 | 115.1 | 113.3 | 109.0 | 111.8 | 110.4 |
| 64 KB | 103.4 | 104.3 | 103.9 | 102.4 | 102.3 | 103.1 | 99.7 | 100.5 | 100.3 |
| 256 KB | 92.2 | 92.1 | 92.5 | 92.1 | 92.1 | 92.5 | 92.1 | 92.1 | 92.5 |

Data Cache misses per 1000 instructions comparing LRU, Random, FIFO replacement for several sizes and associativities. These data were collected for a block size of 64 bytes for the Alpha architecture using 10 SPEC2000 benchmarks. Five are from SPECint2000(gap, gcc, gzip, mcf and perl) and five are from SPECfp2000(applu, art, equake, lucas and swim)

- **观察结果（失效率）**
  - 相联度高，失效率较低。
  - Cache容量较大，失效率较低。
  - LRU 在Cache容量较小时，失效率较低
  - 随着Cache容量的加大，Random的失效率在降低

# Q4: 写策略

- **程序对存储器读操作占26%， 写操作占9%**
  - 写所占的存储器访问比例 9/(100+26+9) 大约为7%
  - 占访问数据Cache的比例： 9/(26+9) 大约为25%
- **大概率事件优先原则 - 优化Cache的读操作**
- **Amdahl定律：不可忽视“写”的速度**
- **“写”的问题**
  - 读出标识，确认命中后，对Cache写 （串行操作）
  - Cache与主存内容的一致性问题
- **写策略就是要解决: 何时更新主存问题**

# 两种写策略

- **写直达法（Write through)**
  - 优点：易于实现，容易保持不同层次间的一致性
  - 缺点：速度较慢
- **写回法 (Write back)**
  - 优点：速度快，减少访存次数
  - 缺点：一致性问题
- **当发生写失效时的两种策略**
  - 按写分配法(Write allocate)：写失效时，先把所写单元所在块调入Cache，然后再进行写入，也称写时取（Fetch on Write)方法
  - 不按写分配法（no-write allocate)：写失效时，直接写入下一级存储器，而不将相应块调入Cache，也称绕写法（Write around)
  - 原则上以上两种方法都可以应用于写直达法和写回法，一般情况下
    - Write Back 用Write allocate
    - Write through 用no-write allocate

# Write-back, Write Allocate

# Write-Through, No Write Allocate

# Alpha AX 21064 Cache结构（数据Cache）

- **基本技术特性**
  - 容量 8KB，Block为32Bytes，　共256个Blocks，每个字为8个字节
  - 直接映象
  - 写直达法，写失效时，no-write allocate 方法
  - 写缓冲：4个blocks
- **21064物理地址34位**
  - 21位tag##8位index ##5位块内偏移　怎么算出来的？
- **Cache命中的步骤**
  - 读命中
  - 写命中
- **Cache失效**
  - Cache向CPU发暂停信号
  - 块传送，21064 Cache与下一级存储器之间数据通路16字节，传送全部32字节需要10个cycles

# Alpha AX 21064 Cache结构（数据Cache)

# Alpha 21264 Data Cache

- **基本技术特征**
  - Cache size: 64Kbytes
  - Block size: 64-bytes
  - Two-way 组相联
  - Write back, 写失效时, write allocate
- **21264 48-bits 虚拟地址，虚实映射为44-bits的物理地址，也支持43-bits虚拟地址，虚实映射为41-bits的物理地址**
  - 29位tags##9位index##6位 Block offset

怎么算出来的?

# Alpha 21264 Data Cache

FIGURE 5.7 The organization of the data cache in the Alpha 21264 microprocessor.

# Cache 性能分析

- **CPU time = (CPU execution clock cycles + Memory stall clock cycles) x clock cycle time**

- **Memory stall clock cycles = (Reads x Read miss rate x Read miss penalty + Writes x Write miss rate x Write miss penalty)**

- **Memory stall clock cycles = Memory accesses x Miss rate x Miss penalty**

- **Different measure: AMAT**

  **Average Memory Access time (AMAT) = Hit Time + (Miss Rate x Miss Penalty)**

- **Note:** ***memory hit time is included in execution cycles.***

# 性能分析举例

- **Suppose a processor executes at**
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- **Miss Behavior:**
  - 10% of memory operations get 50 cycle miss penalty
  - 1% of instructions get same miss penalty
- **CPI = ideal CPI + average stalls per instruction**

  **= 1.1(cycles/ins) +**
  **[ 0.30 (DataMops/ins)**
  **x 0.10 (miss/DataMop) x 50 (cycle/miss)] +**
  **[ 1 (InstMop/ins)**
  **x 0.01 (miss/InstMop) x 50 (cycle/miss)]**
  **= (1.1 + 1.5 + .5) cycle/ins = 3.1**
- **65% (2/3.1) of the time the processor is stalled waiting for memory!**
- **AMAT=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54**

# Example: Harvard Architecture

- **Unified vs Separate I&D (Harvard)**

- **Statistics (given in H&P):**
  - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
  - 32KB unified: Aggregate miss rate=1.99%
- **Which is better (ignore L2 cache)?**
  - Assume 33% data ops ⇒ 75% accesses from instructions (1.0/1.33)
  - hit time=1, miss time=50
  - Note that data hit has 1 stall for unified cache (only one port)

- **AMATHarvard=75%x(1+0.64%x50)+25%x(1+6.47%x50) = 2.05**
- **AMATUnified =75%x(1+1.99%x50)+25%x(1+1+1.99%x50)= 2.24**

| Size | Instruction cache | Data cache | Unified cache |
|---|---|---|---|
| 8 KB | 8.16 | 44.0 | 63.0 |
| 16 KB | 3.82 | 40.9 | 51.0 |
| 32 KB | 1.36 | 38.4 | 43.3 |
| 64 KB | 0.61 | 36.9 | 39.4 |
| 128 KB | 0.30 | 35.3 | 36.2 |
| 256 KB | 0.02 | 32.6 | 32.9 |

**FIGURE 5.8 Miss per 1000 instructions for instruction, data, and unified caches of different sizes.** The percentage of instruction references is about 78%. The data are for two-way associative caches with 64-byte blocks for the same computer and benchmarks as Figure 5.6.

- 以顺序执行的计算机 UltraSPARC III为例. 假设Cache失效开销为100 clock cycles，所有指令忽略存储器停顿需要1个cycle, Cache失效可以用两种方式给出

（1）假设平均失效率为2%，平均每条指令访存1.5次

（2）假设每1000条指令cache失效次数为30次

分别基于上述两种条件计算处理器的性能

- 结论：
- CPUtime = IC * (1 + 2%*1.5*100 ) * T = IC * 4 * T

(1) CPI越低，固定周期数的Cache失效开销的相对影响就越大

(2) 在计算CPI时，失效开销的单位是时钟周期数。因此，即使两台计算机的存储层次完全相同，时钟频率较高的CPU的失效开销会较大，其CPI中存储器停顿部分也就较大。

因此 Cache对于低CPI，高时钟频率的CPU来说更加重要

# 考虑不同组织结构的Cache对性能的影响:

**B-19例题：直接映像Cache 和两路组相联Cache，试问他们对CPU性能的影响？先求平均访存时间，然后再计算CPU性能。分析时请用以下假设：**

**（1）理想Cache(命中率为100%）情况下CPI 为1.0，时钟周期为0.35ns，平均每条指令访存1.4次**

**（2）两种Cache容量均为128KB，块大小都是64B**

**（3）采用组相联时，由于多路选择器的存在，时钟周期增加到原来的1.35倍**

**（4）两种结构的失效开销都是65ns (在实际应用中，应取整为整数个时钟周期)**

**（5）命中时间为1个cycle，128KB直接映像Cache的失效率为2.1%, 相同容量的两路组相联Cache 的失效率为1.9%**

# Summary of performance equations in this chapter

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}}\right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{L1}}{\text{Instruction}} \times \text{Hit time}_{L2} + \frac{\text{Misses}_{L2}}{\text{Instruction}} \times \text{Miss penalty}_{L2}$$

**FIGURE 5.9 Summary of performance equations in this chapter.** The first equation calculates with cache index size, but the rest help evaluates performance. The final two equations deal with multilevel caches, is explained early in the next section. They are included here to help make the figure a useful reference.

# 改进Cache 性能的方法

- **平均访存时间 = 命中时间 + 失效率×失效开销**
- **从上式可知，基本途径**
  - 降低失效率
  - 减少失效开销
  - 缩短命中时间

# 基本Cache优化方法

- **降低失效率**
  - 1、增加Cache块的大小
  - 2、增大Cache容量
  - 3、提高相联度
- **减少失效开销**
  - 4、多级Cache
  - 5、使读失效优先于写失效
- **缩短命中时间**
  - 6、避免在索引缓存期间进行地址转换

# 降低失效率

**Cache失效的原因 可分为三类 3C**

- **强制性失效 (Compulsory or cold)**
  - 第一次访问某一块，只能从下一级Load，也称为冷启动或首次访问失效
- **容量失效（Capacity)**
  - 如果程序执行时，所需块由于容量不足，不能全部调入Cache， 则当某些块被替换后，若又重新被访问，就会发生失效。
  - 可能会发生“抖动”现象
- **冲突失效（Conflict (collision))**
  - 组相联和直接相联的副作用
  - 若太多的块映像到同一组（块）中，则会出现该组中某个块被别的块替换（即使别的组或块有空闲位置），然后又被重新访问的情况，这就属于冲突失效

# 各种类型的失效率

Compulsory misses are independent of cache size

Very small for long-running programs

Capacity misses decrease as capacity increases

Conflict misses decrease as associativity increases

Data were collected using LRU replacement

# 从统计规律中得到的一些结果

- **相联度越高，冲突失效就越小**
- **强制性失效和容量失效不受相联度的影响**
- **强制性失效不受Cache容量的影响**
- **容量失效随着容量的增加而减少**
- **符合2:1Cache经验规则**
  - 即大小为N的直接映象Cache的失效率约等于大小为N/2的两路组相联的Cache失效率。

# 减少3C的方法

**从统计规律可知**

- **增大Cache容量**
  - 对冲突和容量失效的减少有利
- **增大块**
  - 减缓强制性失效
  - 可能会增加冲突失效（因为在容量不变的情况下，块的数目减少了）
- **通过预取可帮助减少强制性失效**
  - 必须小心不要把你需要的东西换出去
  - 需要预测比较准确（对数据较困难，对指令相对容易）

# 增加块的大小

Increased Conflict Misses
Reduced Compulsory Misses
Miss Rate
25%
20%
15%
10%
5%
0%
16
32
64
128
256
Block Size (bytes)
1K
4K
16K
64K
256K
64-byte blocks are common in L1 caches
128-byte blocks are common in L2 caches

# 块大小、容量对失效率的影响

- **Given: miss rates for different cache sizes & block sizes**

| Block Size | Cache = 4 KB | Cache = 16 KB | Cache = 64 KB | Cache = 256 KB |
|---|---|---|---|---|
| 16 bytes | 8.57% | 3.94% | 2.04% | 1.09% |
| 32 bytes | 7.24% | 2.87% | 1.35% | 0.70% |
| 64 bytes | 7.00% | 2.64% | 1.06% | 0.51% |
| 128 bytes | 7.78% | 2.77% | 1.02% | 0.49% |
| 256 bytes | 9.51% | 3.92% | 1.15% | 0.49% |

- **Memory latency = 80 cycles + 1 cycle per 8 bytes**
  - Latency of 16-byte block = 80 + 2 = 82 clock cycles
  - Latency of 32-byte block = 80 + 4 = 84 clock cycles
  - Latency of 256-byte block = 80 + 32 = 112 clock cycles
- **Which block has smallest AMAT for each cache size?**

# 块大小、容量对AMAT的影响

- **Solution: assume hit time = 1 clock cycle**
  - Regardless of block size and cache size
- **Cache Size = 4 KB, Block Size = 16 bytes**
  - AMAT = 1 + 8.57% × 82 = 8.027 clock cycles
- **Cache Size = 256 KB, Block Size = 256 bytes**
  - AMAT = 1 + 0.49% × 112 = 1.549 clock cycles

| Block Size | Cache = 4 KB | Cache = 16 KB | Cache = 64 KB | Cache = 256 KB |
|---|---|---|---|---|
| 16 bytes | AMAT = 8.027 | AMAT = 4.231 | AMAT = 2.673 | AMAT = 1.894 |
| 32 bytes | AMAT = **7.082** | AMAT = 3.411 | AMAT = 2.134 | AMAT = 1.588 |
| 64 bytes | AMAT = 7.160 | AMAT = **3.323** | AMAT = **1.933** | AMAT = **1.449** |
| 128 bytes | AMAT = 8.469 | AMAT = 3.659 | AMAT = 1.979 | AMAT = 1.470 |
| 256 bytes | AMAT = 11.65 | AMAT = 4.685 | AMAT = 2.288 | AMAT = 1.549 |

# 块大小、容量对AMAT的影响

| Block Size | Penalty | Cache Size 1K | Cache Size 4K | Cache Size 16K | Cache Size 64K | Cache Size 256K |
|---|---|---|---|---|---|---|
| 16 | 42 | 15.05% / 7.321 | 8.57% / 4.599 | 3.94% / 2.655 | 2.04% / 1.857 | 1.09% / 1.458 |
| 32 | 44 | 13.34% / 6.870 | 7.24% / 4.186 | 2.87% / 2.263 | 1.35% / 1.594 | 0.70% / 1.308 |
| 64 | 48 | 13.76% / 7.605 | 7.00% / 4.360 | 2.64% / 2.267 | 1.06% / 1.509 | 0.51% / 1.245 |
| 128 | 56 | 16.64%/ 10.318 | 7.78% / 5.357 | 2.77% / 2.551 | 1.02% / 1.571 | 0.49% / 1.274 |
| 256 | 72 | 22.01% /16.847 | 9.51% / 7.847 | 3.29% / 3.369 | 1.15% / 1.828 | 0.49% / 1.353 |

Miss Rate / Average Access Time (in cycles)

what's going on here?

- Remember
  - Avg-access-time= hit-time + miss-rate x miss-penalty

- **降低失效率最简单的方法是增加块大小；统计结果如图所示**
- **假定存储系统在延迟40个时钟周期后，每2个时钟周期能送出16个字节，即：经过42个时钟周期，它可提供16个字节；经过44个四周周期，可提供32个字节；依此类推。试根据图5-6列出的各种容量的Cache，在块大小分别为多少时，平均访存时间最小?**

# 块大小、容量的权衡

- **从统计数据可得到如下结论**
  - 对于给定Cache容量，块大小增加时，失效率开始是下降，但后来反而上升
  - Cache容量越大，使失效率达到最低的块大小就越大
- **分析**
  - 块大小增加，可使强制性失效减少（空间局部性原理）
  - 块大小增加，可使冲突失效增加（Cache中块数量减少）
  - 失效开销增大（上下层间移动，数据传输时间变大）
- **设计块大小的原则，不能仅看失效率**
  - 原因：平均访存时间 = 命中时间 + 失效率×失效开销

# 提高相联度

- **8路组相联在降低失效率方面的作用已经和全相联一样有效**
- **2:1Cache经验规则**
  - 容量为N的直接映象Cache失效率与容量为N/2的两路组相联Cache的失效率差不多相同
- **提高相联度，会增加命中时间**

| Size (KB) | 1-way | 2-way | 4-way | 8-way |
|---|---|---|---|---|
| 1 | 7.65 | 6.60 | 6.22 | 5.44 |
| 2 | 5.90 | 4.90 | 4.62 | 4.09 |
| 4 | 4.60 | 3.95 | 3.57 | 3.19 |
| 8 | 3.30 | 3.00 | 2.87 | 2.59 |
| 16 | 2.45 | 2.20 | 2.12 | 2.04 |
| 32 | 2.00 | 1.80 | 1.77 | 1.79 |
| 64 | 1.70 | 1.60 | 1.57 | 1.59 |
| 128 | 1.50 | 1.45 | 1.42 | 1.44 |

Average Memory Access Time

OOPS!

# Victim Cache(1/2)

- 在Cache和Memory之间增加一个小的全相联Cache

**FIGURE 5.13 Placement of victim cache in the memory hierarchy.** Although it reduces miss penalty, the victim cache is aimed at reducing the damage done by conflict misses, described in the next section. Jouppi [1990] found the four-entry victim cache could reduce the miss penalty for 20% to 95% of conflict misses.

# Victim Cache(2/2)

- **基本思想**
  - 通常Cache为直接映象时冲突失效率较大
  - Victim cache采用全相联 - 失效率较低
  - Victim cache存放由于（冲突）失效而被丢弃的那些块
  - 失效时，首先检查Victim cache是否有该块，如果有就将该块与Cache中相应块比较。
- **Jouppi (DEC SRC)发现，含1到5项的Victim cache对减少失效很有效，尤其是对于那些小型的直接映象数据Cache 。测试结果，项为4的Victim Cache能使4KB直接映象数据Cache冲突失效减少20%-90%**

# 减少失效开销

- **减少CPU与存储器间性能差异的重要手段**
  - 平均访存时间 = 命中时间 + 失效率×失效开销

- **基本手段：**
  - 4、多级Cache技术(Multilevel Caches)
  - 5、让读优先于写(Giving Priority to Read Misses over Writes)

# 采用多级Cache

- **一级cache保持较小容量**
  - 降低命中时间
  - 降低每次访问的能耗
- **增加二级cache**
  - 减少与存储器的gap
  - 减少存储器总线的负载
- **多级cache的优点**
  - 减少失效开销
  - 缩短平均访存时间（AMAT）
- **较大容量的L2 cache可以捕捉许多L1 cache的失效**
  - 降低全局失效率

# 多级包容性(multilevel inclusive)

- **L1 cache 的块总是存在于L2 cache中**
  - 浪费了L2 cache 空间，L2 还应当有存放其他块的空间
- **L1中miss, 但在L2中命中，则从L2拷贝相应的块到L1**
- **在L1和L2中均miss, 则从更低级拷贝相应的块到L1和L2**
- **对L1写操作导致将数据同时写到L1和L2**
- **Write-through 策略用于L1到L2**
- **Write-back 策略可用于L2 到更低级存储器，以降低存储总线的数据传输压力**
- **L2的替换动作（或无效)对L1可见**
  - 即L2的一块被替换出去，那么其在L1中对应的块也要被替换出去。

| |
|---|
| **0** |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |

- **L1： 1-way，容量：2块**
- **L2 4-way，容量：4块**
- **块大小相同**
- **访问的块序列10246**
- **替换策略：LRU**

| | |
|---|---|
| 0 | 0(0) |
| 1 | 0(1) |
| 2 | 1(2) |
| 3 | 1(3) |
| 4 | 2(4) |
| 5 | 2(5) |
| 6 | 3(6) |
| 7 | 3(7) |
| 8 | 4(8) |
| 9 | 4(9) |

- L1： 1-way，容量：2块
- L2 2-way，容量：2块
- L1和L2的块大小不同，L1的blockSize是L2的blockSize的1/2
- 替换策略：LRU
- 考察按L1的块划分的访问序列：

0148

# 多级不包容（Multilevel Exclusive）

- **L1 cache 中的块不会在L2 cache中，以避免浪费空间**
- **在L1中miss, 但在L2中命中，将导致Cache间块的互换**
- **在L1和L2中均miss, 将仅仅从更低层拷贝相应的块到L1**
- **L1的被替换的块移至L2**
  - L2 存储L1抛弃的块，以防后续L1还需要使用
- **L1到L2的写策略为 Write-Back**
- **L2到更低级cache的写策略为 Write-Back**
- **L1和L2的块大小可以相同也可以不同**
  - Pentium 4 had 64-byte blocks in L1 but 128-byte blocks in L2
  - Core i7 uses 64-byte blocks at all cache levels (simpler)

- L1： 1-way L2： 2-way
- 访问的块序列0123456 3
- 替换策略：LRU
- 块大小相同

# 多级cache的性能分析

- **局部失效率：该级Cache的失效次数 / 到达该级Cache的访存次数**
  - Miss rateL1 for L1 cache
  - Miss rateL2 for L2 cache
- **全局失效率：该级Cache的失效次数/ CPU发出的访存总次数**
  - Miss rateL1 for L1 cache
  - Miss rateL1 × Miss rateL2 for L2 cache
  - 全局失效率是度量L2 cache性能的更好方法
- **性能参数**
  - AMAT = Hit TimeL1+Miss rateL1×Miss penaltyL1
  - Miss penaltyL1 = HitTimeL2 + Miss rateL2×Miss penaltyL2
  - AMAT = Hit TimeL1+Miss rateL1× (Hit TimeL2+Miss rateL2×Miss penaltyL2)

# L1 cache 失效率

- **对于I-Cache和D-Cache分开的L1 Cache**

  Miss RateL1 = %inst × Miss RateI-Cache + %data × Miss RateD-Cache

  %inst = Percent of Instruction Accesses = 1 / (1 + %LS)

  %data = Percent of Data Accesses = %LS / (1 + %LS)

  %LS = Frequency of Load and Store instructions

- **每条指令的L1 失效次数:**

  Misses per InstructionL1 = Miss RateL1 × (1 + %LS)

  Misses per InstructionL1 = Miss RateI-Cache + %LS × Miss RateD-Cache

# 具有二级Cache的AMAT举例

- **Problem: 计算AMAT**
  - I-Cache 失效率 = 1%， D-Cache失效率 = 10%
  - L2 Cache失效率 = 40%
  - L1 命中时间 = 1 cycle ( I-Cache 和D-Cache相同)
  - L2 命中时间 = 8 cycles， L2 失效开销 = 100 cycles
  - Load + Store 指令频度 = 25%
- **Solution：**
  - 平均每条指令访存次数 = 1+25% = 1.25
  - 平均每条指令的失效次数 = 1%+25%× 10% = 0.035
  - L1的失效率= 0.035/1.25=0.028
  - L1的失效开销 = 8 + 0.4×100 = 48 cycles
  - AMAT = 1+0.028×48 = 2.344

# Memory Stall Cycles Per Instruction

- **Memory Stall Cycles per Instruction**

  = Memory Access per Instruction × Miss RateL1 × Miss PenaltyL1

  = (1 + %LS) × Miss RateL1 × Miss PenaltyL1

  = (1 + %LS) × Miss RateL1 × (Hit TimeL2 + Miss RateL2 × Miss PenaltyL2)

- **Memory Stall Cycles per Instruction**

  = Misses per InstructionL1 × Hit TimeL2 +
  Misses per InstructionL2 × Miss PenaltyL2

  Misses per InstructionL1 = (1 + %LS) × Miss RateL1

  Misses per InstructionL2 = (1 + %LS) × Miss RateL1 × Miss RateL2

# 两级Cache的性能

- **Problem: 程序运行产生1000个存储器访问**
  - I-Cache misses = 5, D-Cache misses = 35, L2 Cache misses = 8
  - L1 Hit = 1 cycle, L2 Hit = 8 cycles, L2 Miss penalty = 80 cycles
  - Load + Store frequency = 25%, CPIexecution = 1.1 (perfect cache)
  - 计算memory stall cycles per instruction 和有效的CPI
  - 如果没有L2 cache，有效的CPI是多少?
- **Solution：**
  - L1 Miss Rate = (5 + 35) / 1000 = 0.04 (or 4% per access)
  - L1 misses per Instruction =0.04 × (1 + 0.25) = 0.05
  - L2 misses per Instruction =(8 / 1000) × 1.25 = 0.01
  - Memory stall cycles per Instruction =0.05 × 8 + 0.01 × 80 = 1.2
  - CPIL1+L2 = 1.1 + 1.2 = 2.3, CPI/CPIexecution = 2.3/1.1 = 2.1x slower
  - CPIL1only =1.1 + 0.05 × 80 = 5.1 (worse)

# 不同大小的L2cache对应的执行时间

Copyright © 2012, Elsevier Inc. All rights reserved.

# Miss rates versus cache size for multilevel caches

# 两级Cache的一些研究结论

- **在L2比L1大得多得情况下，两级Cache全局失效率 和容量与第二级Cache相同的单级Cache的失效率接近**
- **局部失效率不是衡量第二级Cache的好指标**
  - 它是第一级Cache失效率的函数
  - 不能全面反映两级Cache体系的性能
- **第二级Cache设计需考虑的问题**
  - 容量：一般很大，可能没有容量失效，只有强制性失效和冲突失效
    - 相联度对第二级Cache的作用
    - Cache可以较大，以减少失效次数
  - 多级包容性问题：第一级Cache中的数据是否总是同时存在于第二级Cache中。
    - 如果L1和L2的块大小不同，增加了多级包容性实现的复杂性

# 多级Cache举例

**Given the data below, what is the impact of second-level cache associativity on its miss penalty?**

- Hit timeL2 for direct mapped = 10 clock cycles
- Two-way set associativity increases hit time by 0.1 clock cycles to 10.1clock cycles
- Local miss rateL2 for direct mapped = 25%
- Local miss rateL2 for two-way set associative = 20%
- Miss penaltyL2 = 100 clock cycles

- **结论：提高相联度，可减少第一级Cache的失效开销**
- **第二级Cache特点：容量大，高相联度，块较大，重点减少失效次数。**

# 让读优先于写图示

CPU
in out
Write Buffer
Tag
Data
write
buffer
DRAM
(or lower mem)

# 让读失效优先于写

- **由于读操作为大概率事件，需要读失效优先，以提高性能**
- **Write-Through Cache ->Write Buffer (写缓冲)，特别对写直达法更有效**
  - Write Buffer: CPU不必等待写操作完成，即将要写的数据和地址送到Write Buffer后，CPU继续作其他操作。
  - 写缓冲导致对存储器访问的复杂化
    - 在读失效时写缓冲中可能保存有所读单元的最新值，还没有写回
    - 例如，直接映射、写直达、512和1024映射到同一块。则
    - SW R3, 512(R0)
    - LW R1, 1024(R0)　　失效
    - LW R2, 512(R0)　　失效
  - 解决问题的方法
    - 推迟对读失效的处理，直到写缓冲器清空，导致新的问题——读失效开销增大。
    - 在读失效时，检查写缓冲的内容，如果没有冲突，而且存储器可访问，就可以继续处理读失效
  - 写回法时，也可以利用写缓冲器来提高性能
    - 把脏块放入缓冲区，然后读存储器，最后写存储器
- **Write-Back Cache ->Victim Buffer**
  - 被替换的脏块放到了victim buffer
  - 在脏块被写回前，需要处理读失效
  - 问题: victim buffer 可能含有该读失效要读取的块
    - Solution: 查找victim buffer，如果命中直接将该块调入Cache

# 缩短命中时间 Virtually indexed cache

- **Cache index用virtual地址还是physical地址?**
- **Virtual index 可以避免在cache indexing时的virtual to physical address translation, 所以命中时间短。**
- **How about tags? A pure virtual cache (with virtual index + virtual tag) seems faster. But what is its problem?**
- **With multiple processes and process switching, a pure virtual cache is not practical**
  - May need to purge the entire cache on a process switch
  - Or can attach a PID field
- **Usually Virtually index, physically tagged**

# 虚拟地址Cache VS. 物理地址Cache

**Figure B.16 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PID), and with process switches but without PIDs (purge).** PIDs increase the uniprocess absolute miss rate by 0.3% to 0.6% and save 0.6% to 4.3% over purging. Agarwal [1987] collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes. Note that the miss rate goes up from 128K to 256K. Such nonintuitive behavior can occur in caches because changing size changes the mapping of memory blocks onto cache blocks, which can change the conflict miss rate.

Copyright © 2011, Elsevier Inc. All rights Reserved.

虚拟地址转换与Cache定位 并行

Virtually indexed, Physically tagged L1

Physically indexed, Physically tagged L2

**Figure B.17 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access.** The page size is 16 KB. The TLB is two-way set associative with 256 entries. The L1 cache is a direct-mapped 16 KB, and the L2 cache is a four-way set associative with a total of 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits.

Copyright © 2011, Elsevier Inc. All rights Reserved.

# 高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

# 1、Small and simple first level caches

- **Small and simple first level caches**
  - 容量小，一般命中时间短，有可能做在片内
  - 另一方案，保持Tag在片内，块数据在片外，如DEC Alpha
  - 第一级Cache应选择容量小且结构简单的设计方案
- **Critical timing path:**
  1）定位组，确定tag的位置
  2）比较tags，
  3）选择正确的块
- **Direct-mapped caches can overlap tag compare and transmission of data**
  - 数据传输和tag 比较并行
- **Lower associativity reduces power because fewer cache lines are accessed**
  - 简单的Cache结构、可有效减少tag比较的次数，进而降低功耗

# L1 Size and Associativity

ty

# L1 Size and Associativity

vity

# 2、Way Prediction

- **为改进命中时间，预测被选中的路(way)**
  - 预测错误会导致更长的命中时间
  - 预测的准确性
    - 90%+ for two-way
    - 80%+ for four-way
    - I-cache比D-cache具有更好的准确性
  - 90年代中期第一次用于MIPS R10000
  - 用于ARM Cortex-A8
- **Way Prediction方法也可降低功耗：直接预测要访问的块**
  - 也称“路选择”“Way selection”
  - 可有效降低功耗，但一旦预测错误会有更长的命中时间

# 高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

# 3、Pipelining Cache

- **实现Cache访问的流水化**
  - 提高Cache的带宽，有利于采用高相联度的缓存
  - L1 cache的访问由多个时钟周期构成
    - Pentium: 1 cycle
    - Pentium Pro - Pentium III: 2 cycles
    - Pentium 4 - Core i7: 4 cycles
    - IBM Power7: 3 cycles
- **缺点：增加流水线的段数**
  - 增加了分支预测错误造成的额外开销 (I-cache)
  - 增加了Load指令与要使用其结果的指令间的latency
  - 增加了I-Cache和D-Cache的延时

**FIGURE 2.2:** Parallel tag and data array access pipeline.

**FIGURE 2.4:** Serial tag and data array access pipeline.

# 4、Nonblocking Caches

- **允许在Cache失效下继续命中**
  - 在Cache失效时，CPU无需stall
  - 主要用于乱序执行和多线程处理器
- **Hit under a Miss**
  - 减少有效的失效开销
  - 增加Cache的带宽
- **Hit under Multiple Misses**
  - 针对多个未解决的Cache失效
  - 可能会更多地减少有效的失效开销
  - 增加了Cache控制器的复杂性
  - 存储系统可以支持多个失效时的存储服务

# Nonblocking Cache Timeline

# Effectiveness of Non-Blocking Cache

**Reduces the miss penalty by not stalling the processor on a cache miss**

Copyright © 2012, Elsevier Inc. All rights reserved.

# Miss Status Holding Register (MSHR)

- **MSHR 包含正在等待处理的失效**
  - 相同的块可以包含多个未解决的Load/Store 失效
  - 可以有多个未解决的块地址
- **失效可以分为**
  - Primary: 第一次发起存取请求时的失效块
  - Secondary: 在后续过程中的失效
  - Structural Stall miss: MSHR 硬件资源耗尽

# NonBlocking Cache Operation

- **当Cache失效时，检查MSHR是否有匹配的块地址**
  - 如果有，为该地址分配新的load/store 表项
  - 如果没有，分配新的MSHR和load/store表项
  - 如果所有MSHR资源都分配完，则Stall（结构相关）
- **当从底层传输Cache块时**
  - 处理该块中的Load和Store指令引起的失效
  - Load：根据block offset从该块中装载数据到寄存器
  - Store：根据block offset将数据写入该块指定位置
  - 完成该块所有的失效的Load/Store后，释放MSHR中的对应表项

# Multi-Ported Cache

- **Dual-Ported Data Cache**
  - 两个地址端口（每个Cycle支持两条load/store 指令）
  - 在现代处理器中保持较高的指令吞吐率
- **True Multi-ported Cache Design**
  - 所有的控制和数据通路在Cache中是多份的
    - Address Decoder, way multiplexor, tag comparator, aligners
    - Tag Array, Data Array
  - 显著地增加了Cache的面积和访问时间
- **Multi-Banked Cache**
  - 将cache组织成多个banks
  - 每个bank是一个端口
  - 可以并行访问不同的Bank

# 5、Multibanked Caches

- **将Cache组织为多个独立的banks，以支持并行访问**
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2

**Figure 2.6 Four-way interleaved cache banks using block addressing.** Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

# 6、Critical Word First, Early Restart

- **关键字优先**
  - 首先请求CPU所需要的字
  - 请求字一到达就发给CPU，使其继续执行，同时从存储器中调入其他字
- **提前重启**
  - 请求字的顺序不变
  - 请求字一到达就发给CPU，使其继续执行，同时从存储器中调入其他字
- **通常在块比较大时，这些技术才有效**

# 7、Merging Write Buffer

- 在向写缓冲器写入地址和数据时，如果写缓冲器中存在被修改过的块，就检查其地址，看看本次写入数据的地址是否与写缓冲器内的某个有效块地址匹配，如果匹配，就把新数据与该块合并，称为“合并写”
- 可以缓解由于写缓冲满而造成的CPU停顿

| Write address | V | | V | | V | | V | |
|---|---|---|---|---|---|---|---|---|
| 100 | 1 | Mem[100] | 0 | | 0 | | 0 | |
| 108 | 1 | Mem[108] | 0 | | 0 | | 0 | |
| 116 | 1 | Mem[116] | 0 | | 0 | | 0 | |
| 124 | 1 | Mem[124] | 0 | | 0 | | 0 | |

No write merging

| Write address | V | | V | | V | | V | |
|---|---|---|---|---|---|---|---|---|
| 100 | 1 | Mem[100] | 1 | Mem[108] | 1 | Mem[116] | 1 | Mem[124] |
| | 0 | | 0 | | 0 | | 0 | |
| | 0 | | 0 | | 0 | | 0 | |
| | 0 | | 0 | | 0 | | 0 | |

With Write merging

# 高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

# 8、编译器优化

- **无需对硬件做任何改动，通过软件优化降低失效率**
- **研究从两方面展开：**
  - 减少指令失效
  - 减少数据失效
- **减少指令失效，重新组织程序（指令调度）而不影响程序的正确性**
  - 研究结果：通过使用profiling信息来判断指令组间可能发生的冲突，并将指令重新排序以减少失效。
  - 研究表明：
    - 容量为2KB，块大小为 4Bytes的直接映象Icache，通过使用指令调度可以使失效率降低50%。容量增大到 8KB，失效率可降低75%
    - 在有些情况下，当能够使某些指令不进入ICache时，可以得到最佳性能。即使不这样做，优化后（指令调度）的程序在直接映象Cache中的失效率也低于未优化程序在同样大小的8路组相联Cache中的失效率。
- **减少数据失效，主要通过优化来改善数据的空间局部性和时间局部性，基本方法为：**
  - 数据合并
  - 内外循环交换，循环融合
  - 分块

# 编译器优化方法举例之一：数组合并

```
// Original Code
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
for (i = 0; i < N; i++)
    d[i] = a[i] + b[i] * c[i];
```

Blocks are replaced in first loop then accessed in second

```
// After Loop Fusion
for (i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
    d[i] = a[i] + b[i] * c[i];
}
```

Revised version takes advantage of temporal locality

# 编译器优化方法举例之二：内外循环交换

```
/* Before */
for (j = 0; j < 100; j = j+1)
        for (i = 0; i < 5000; i =
  i+1)
                x[i][j] = 2 * x[i][j];
/* After */
for (i = 0; i < 5000; i = i+1)
        for (j = 0; j < 100; j =
  j+1)
                x[i][j] = 2 *
  x[i][j];
```

Revised version takes advantage of spatial locality

# 编译器优化方法举例之三：分块（1/2）

```
/* Before */
for (i = 0; i < N; i = i+1)
      for (j = 0; j < N; j = j+1)
      {     r = 0;
            for (k = 0; k < N; k = k + 1)
                  r = r + y[i][k]*z[k][j];
            x[i][j] = r;
      };
```

$N^3$次操作，产生的存储器访问次数为: $2N^3+N^2$

# 编译器优化方法举例之三：分块（2/2）

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
   for (kk = 0; kk < N; kk = kk+B)
         for (i = 0; i < N; i = i+1)
       for (j = jj; j < min(jj+B,N); j = j+1)
        {   r = 0;
                    for (k = kk; k < min(kk+B,N); k = k + 1)
                          r = r + y[i][k]*z[k][j];
                     x[i][j] = x[i][j] + r;
              };
```

# 9、Hardware Prefetching

- **CPU在执行当前块代码时，硬件预取下一块代码**
  - CPU可能马上就要执行这块代码，这样可以降低或消除Cache的访问失效
- **当块中有控制指令时，预取失效**
- **预取的指令可以放在Icache中，也可以放在其他地方（存取速度比Memory块的地方）**
- **AXP21064失效时，取2块指令块**
  - 目标块放在Icache，下一块放在ISB(指令流缓冲）中
  - 如果访问的块在ISB中，取消访存请求，直接从ISB中读，并发出对下一指令块的预取访存请求
- **研究结果：块大小为16字节，容量为4KB的直接映象Cache，1个块的指令流缓冲器，可以捕获15% - 25%的失效，4个块ISB可捕获50%的失效，16块ISB可捕获72%的失效**

# 硬件预取

- **预取数据**
  - 出发点：CPU访问一块数据，可能马上要访问下一块数据
  - Jouppi研究结果：块大小16字节，4KB直接映象Cache， 1Block DSB - 25%
  - 4Block DSB - 43%
  - Palacharla和Kessler 1994年研究
    - 一个具有两个64KB四路组相联(Icache, Dcache)的处理器来说，8Blocks流缓冲器能够捕获其50% - 70%的失效
- **举例：Alpha AXP21064采用指令预取技术，其实际失效率是多少？若不采用指令预取技术，Alpha AXP 21064的指令Cache必须为多大才能保持平均访存时间不变?**
  - 假设当指令不在指令Cache中，在预取缓冲区中找到时，需要多花1个时钟周期。
  - 假设预取命中率为25%，命中时间为1个时钟周期，失效开销为50个时钟周期
  - 8KB指令Cache的失效率为1.10%，16KB指令cache的失效率为0.64%
  - AMAT（预取）= 命中时间+失效率*预取命中率*1+失效率*（1-预取命中率）*失效开销
- **注意：预取是利用存储器的空闲带宽，而不是与正常的存储器操作竞争。**

# 硬件预取

Fetch two blocks on miss (include next sequential block)

Pentium 4 Pre-fetching

# 10、Compiler Prefetching（1/2）

- **在ISA中增加预取指令，让编译器控制预取**
- **预取的种类**
  - 寄存器预取 ： 把数据取到R中
  - Cache预取：只将数据取到Cache中，不放入寄存器
- **故障问题**
  - 两种类型的预取可以是故障性预取，也可以是非故障性预取
  - 所谓故障性预取指在预取时若出现虚地址故障，或违反保护权限，就会有异常发生
  - 非故障性预取，如导致异常就转化为空操作
- **只有在预取时，CPU可以继续执行的情况下，预取才有意义**
  - Cache在等待预取数据返回的同时，可以正常提供指令和数据，称为非阻塞Cache或非锁定Cache

# 由编译器控制预取 （2/2）

- **循环是预取优化的主要目标**
  - 失效开销较小时，Compiler简单的展开一两次，调度好预取与执行的重叠
  - 失效开销较大时，编译器将循环体展开多次，以便为较远的循环预取数据
  - 由于发出预取指令需要花费一条指令的开销，因此要避免不必要的预取
  - 重点放在那些可能导致失效的访问
- **举例1：P93（70） （Hennessy & Patterson 5th /中译本)**
- **举例2：P94（71） （Hennessy & Patterson 5th/中译本)**

# 举例1

**Example： For the code below, determine which accesses are likely to cause data cache misses. Next, insert prefetch instructions to reduce misses. Finally, calculate the number of prefetch instructions executed and the misses avoided by prefetching.**

**Let’s assume we have an 8 KB direct-mapped data cache with 16-byte blocks, and it is a write-back cache that does write allocate. The elements of a and b are 8 bytes long since they are double-precision floating-point arrays. There are 3 rows and 100 columns for a and 101 rows and 3 columns for b. Let’s also assume they are not in the cache at the start of the program.**

```
for (i = 0; i < 3; i = i+1)
    for (j = 0; j < 100; j = j+1)
        a[i][j] = b[j][0] * b[j+1][0];
```

```
for (j = 0; j < 100; j = j+1) {
        prefetch(b[j+7][0]);
        /* b(j,0) for 7 iterations later */
        prefetch(a[0][j+7]);
        /* a(0,j) for 7 iterations later */
        a[0][j] = b[j][0] * b[j+1][0];};

for (i = 1; i < 3; i = i+1)
        for (j = 0; j < 100; j = j+1) {
                prefetch(a[i][j+7]);
                /* a(i,j) for +7 iterations */
                a[i][j] = b[j][0] * b[j+1][0];}
```

# 举例2

Example :

Calculate the time saved in the example above.

(1) Ignore instruction cache misses and assume there are no conflict or capacity misses in the data cache.

(2) Assume that prefetches can overlap with each other and with cache misses, thereby transferring at the maximum memory bandwidth.

(3) Here are the key loop times ignoring cache misses: The original loop takes 7 clock cycles per iteration, the first prefetch loop takes 9 clock cycles per iteration, and the second prefetch loop takes 8 clock cycles per iteration (including the overhead of the outer for

# cache -review

- **基本优化方法**
  - 扩大Cache容量
  - 增大Cache块
  - 提高相联度
  - Victim Cache
  - 多级Cache
  - AMAT = Hit TimeL1+Miss rateL1×(Hit TimeL2+Miss rateL2×Miss penaltyL2)
  - 让读失效优先于写写失效
- **10 种高级优化方法**

# -review： Summary

| Technique | Hit time | Band-width | Miss penalty | Miss rate | Power consumption | Hardware cost/ complexity | Comment |
|---|---|---|---|---|---|---|---|
| Small and simple caches | + | | | – | + | 0 | Trivial; widely used |
| Way-predicting caches | + | | | | + | 1 | Used in Pentium 4 |
| Pipelined cache access | – | + | | | | 1 | Widely used |
| Nonblocking caches | | + | + | | | 3 | Widely used |
| Banked caches | | + | | | + | 1 | Used in L2 of both i7 and Cortex-A8 |
| Critical word first and early restart | | | + | | | 2 | Widely used |
| Merging write buffer | | | + | | | 1 | Widely used with write through |
| Compiler techniques to reduce cache misses | | | | + | | 0 | Software is a challenge, but many compilers handle common linear algebra calculations |
| Hardware prefetching of instructions and data | | | + | + | – | 2 instr., 3 data | Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware. |
| Compiler-controlled prefetching | | | + | + | | 3 | Needs nonblocking cache; possible instruction overhead; in many CPUs |

**Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity.** Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

# 存储器技术与优化

- **存储器的访问源**
  - 取指令、取操作数、写操作数和I/O
- **存储器性能指标**
  - 容量、速度和每位价格
  - 访问时间（Access Time）
  - 存储周期（Cycle Time）
- **种类：DRAM和SRAM**
  - Memory： DRAM， Cache： SRAM

# DRAM

- **DRAM是由单个信息位构成的阵列**
  - 通过行选择线和列选择线访问
  - 所有DRAM都由这些阵列构成
  - 不同的结构根据性能的需求选择的阵列数可能不同
- **所有DRAM的访问至少三个阶段**
  - Precharge, row access, column access
- **DRAM performance**
  - Latency
    - 处理器发出请求到所请求的第一组数据处理器输入引脚到达所需要的cycle数
  - Bandwidth
    - 第一组数据到达后，后续数据到达的速率

Figure 3.2: Timing Phases of a DRAM Access

# DRAM

- **DRAM**
  - 破坏性读：读后需要重新写回，必须要周期性的刷新
  - 每位1个 transistor
  - 地址线复用：
    - Lower half of address: column access strobe (CAS)
    - Upper half of address: row access strobe (RAS)
- **SRAM**
  - 每位6个transistors
  - 只需较低的功率来保持位状态

# DRAM Packaging (Laptops/Desktops/Servers)

- **DIMM (Dual Inline Memory Module) contains multiple chips with clock/control/address signals connected in parallel (sometimes need buffers to drive signals to all chips)**
- **Data pins work together to return wide word (e.g., 64-bit data bus using 16x4-bit parts)**

# Memory 优化

- Some optimizations:
  - Fast Page Mode Operation
    - Multiple accesses to same row
  - Synchronous DRAM
    - Added clock to DRAM interface
    - Burst mode with critical word first
  - Wider interfaces
  - Double data rate (DDR)
  - Multiple banks on each DRAM device

| Production year | Chip size | DRAM Type | Row access strobe (RAS) | | Column access strobe (CAS)/ data transfer time (ns) | Cycle time (ns) |
|---|---|---|---|---|---|---|
| | | | Slowest DRAM (ns) | Fastest DRAM (ns) | | |
| 1980 | 64K bit | DRAM | 180 | 150 | 75 | 250 |
| 1983 | 256K bit | DRAM | 150 | 120 | 50 | 220 |
| 1986 | 1M bit | DRAM | 120 | 100 | 25 | 190 |
| 1989 | 4M bit | DRAM | 100 | 80 | 20 | 165 |
| 1992 | 16M bit | DRAM | 80 | 60 | 15 | 120 |
| 1996 | 64M bit | SDRAM | 70 | 50 | 12 | 110 |
| 1998 | 128M bit | SDRAM | 70 | 50 | 10 | 100 |
| 2000 | 256M bit | DDR1 | 65 | 45 | 7 | 90 |
| 2002 | 512M bit | DDR1 | 60 | 40 | 5 | 80 |
| 2004 | 1G bit | DDR2 | 55 | 35 | 5 | 70 |
| 2006 | 2G bit | DDR2 | 50 | 30 | 2.5 | 60 |
| 2010 | 4G bit | DDR3 | 36 | 28 | 1 | 37 |
| 2012 | 8G bit | DDR3 | 30 | 24 | 0.5 | 31 |

**Figure 2.13 Times of fast and slow DRAMs vary with each generation.** (Cycle time is defined on page 95.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs. The introduction of various burst transfer modes in the mid-1990s and SDRAMs in the late 1990s has significantly complicated the calculation of access time for blocks of data; we discuss this later in this section when we talk about SDRAM access time and power. The DDR4 designs are due for introduction in mid- to late 2012. We discuss these various forms of DRAMs in the next few pages.

# Memory Optimizations

| Standard | Clock rate (MHz) | M transfers per second | DRAM name | MB/sec /DIMM | DIMM name |
|---|---|---|---|---|---|
| DDR | 133 | 266 | DDR266 | 2128 | PC2100 |
| DDR | 150 | 300 | DDR300 | 2400 | PC2400 |
| DDR | 200 | 400 | DDR400 | 3200 | PC3200 |
| DDR2 | 266 | 533 | DDR2-533 | 4264 | PC4300 |
| DDR2 | 333 | 667 | DDR2-667 | 5336 | PC5300 |
| DDR2 | 400 | 800 | DDR2-800 | 6400 | PC6400 |
| DDR3 | 533 | 1066 | DDR3-1066 | 8528 | PC8500 |
| DDR3 | 666 | 1333 | DDR3-1333 | 10,664 | PC10700 |
| DDR3 | 800 | 1600 | DDR3-1600 | 12,800 | PC12800 |
| DDR4 | 1066–1600 | 2133–3200 | DDR4-3200 | 17,056–25,600 | PC25600 |

**Figure 2.14 Clock rates, bandwidth, and names of DDR DRAMS and DIMMs in 2010.** Note the numerical relationship between the columns. The third column is twice the second, and the fourth uses the number from the third column in the name of the DRAM chip. The fifth column is eight times the third column, and a rounded version of this number is used in the name of the DIMM. Although not shown in this figure, DDRs also specify latency in clock cycles as four numbers, which are specified by the DDR standard. For example, DDR3-2000 CL 9 has latencies of 9-9-9-28. What does this mean? With a 1 ns clock (clock cycle is one-half the transfer rate), this indicate 9 ns for row to columns address (RAS time), 9 ns for column access to data (CAS time), and a minimum read time of 28 ns. Closing the row takes 9 ns for precharge but happens only when the reads from that row are finished. In burst mode, transfers occur on every clock on both edges, when the first RAS and CAS times have elapsed. Furthermore, the precharge in not needed until the entire row is read. DDR4 will be produced in 2012 and is expected to reach clock rates of 1600 MHz in 2014, when DDR5 is expected to take over. The exercises explore these details further.

- **DDR:**
  - DDR2: Lower power (2.5 V -> 1.8 V), Higher clock rates (266 MHz, 333 MHz, 400 MHz)
  - DDR3: 1.5 V, 800 MHz
  - DDR4: 1-1.2 V, 1600 MHz
- **GDDR5 is graphics memory based on DDR3**
- **Graphics memory:**
  - Achieve 2-5 X bandwidth per DRAM vs. DDR3
    - Wider interfaces (32 vs. 16 bit)
    - Higher clock rate

# 3D-Stacked DRAM and Processing in Memory (PIM)

Multiple DRAM chips are stacked within the same package as the processor, thus a shorter access latency

# Memory 功耗

Reducing power in SDRAMs:

- Lower voltage
- Low power mode (ignores clock, continues to refresh)

Figure 2.6 Power consumption for a DDR3 SDRAM operating under three conditions: low-power (shutdown) mode, typical system mode (DRAM is active 30% of the time for reads and 15% for writes), and fully active mode, where the DRAM is continuously reading or writing. Reads and writes assume bursts of eight transfers. These data are based on a Micron 1.5V 2GB DDR3-1066, although similar savings occur in DDR4 SDRAMs

# 三种存储器组织方式

a. One-word-wide memory organization

b. Wide memory organization

c. Interleaved memory organization

***Simple***:

CPU, Cache, Bus, Memory same width
(32 bits)

***Wide***:

**CPU/Mux 1 word; Mux/Cache, Bus, Memory N words (Alpha: 64 bits & 256 bits)**

***Interleaved***:

**CPU, Cache, Bus 1 word: Memory N Modules (4 Modules); example is *word interleaved***

# 提高主存性能的方法-增大存储器的宽度（并行访问存储器）

- **最简单直接的方法**
- **优点：简单、直接，可有效增加带宽**
- **缺点**
  - 增加了CPU与存储器之间的连接通路的宽度，实现代价提高
  - 主存容量扩充时，增量应该是存储器的宽度
  - 写操作问题（部分写操作）
- **冲突问题**
  - 取指令冲突，遇到程序转移时，一个存储周期中读出的n条指令中，后面的指令将无用
  - 读操作数冲突。一次同时读出的几个操作数，不一定都有用
  - 写操作冲突。这种并行访问，必须凑齐n个字之后一起写入。如果只写一个字，必须先把属于同一个存储字的数据读到数据寄存器中，然后在地址码的控制下修改其中一个字，最后一起写。
  - 读写冲突。当要读写的字在同一个存储字内时，无法并行操作。
- **冲突的原因**
  - 从存储器本身看，主要是地址寄存器和控制逻辑只有一套。

# 采用简单的多体交叉存储器

- **一套地址寄存器和控制逻辑**
- **存储器组织为多个体（Bank)**
- **存储体的宽度，通常为一个字，不需要改变总线的宽度**
- **目的：在总线宽度不变的情况下，完成多个字的并行读写**

- **存储模块中所包含的体数，为避免访问冲突，基本原则为：**
  - 体的数目 >= 访问体中一个字所需的时钟周期数
  - 例如：某一向量机的存储系统，CPU发出访存请求10个时钟周期后，CPU将从存储体0得到一个字，随后体0开始读该存储体的下一个字，而CPU依次从其余7个存储体中得到后继的7个字。在第18个周期，CPU 将需要存储体0提供下一个字，但该字要到第20个时钟周期才被读出，CPU只好等待。
- **缺陷：不能对单个体单独访问，对解决冲突没有帮助，逻辑上是一种宽存储器，对各个存储体的访问被安排在不同的时间段**

# Increasing Bandwidth - Interleaving

**Access Pattern without Interleaving:**

**Access Pattern with 4-way Interleaving:**

Write
bank0
bank1
bank2
bank3
0x00
mod 4 = 0

# 地址映射方法（m个存储体，每个存储体容量为n）

- **高位交叉编址：**相当于对存储单元矩阵按列优先的方式进行编址

  考虑处于第 i 行第 j 列的单元，其线性地址为：

  $A = j \times n + i$ // 其中：j =0~m-1 ；i = 0~n-1

  已知：一个单元的线性地址为A， 则：

  j = $\lfloor A/n \rfloor$ //A / n 取下限

  i = $A \bmod n$

  **取线性地址高位 $\log_2 m$位就是体号，其余低位部分就是体内地址**

- **低位交叉编址：**相当于对存储单元矩阵按行优先的方式进行编址

  考虑处于第 i 行第 j 列的单元，其线性地址为：

  $A = i \times m + j$ //其中：i = 0~n-1 ；j = 0~m-1

  已知：一个单元的线性地址为A， 则：

  i =$\lfloor A/m \rfloor$ j = A mod m

  **取线性地址低位 $\log_2 m$位就是体号，其余高位部分就是体内地址**

# 例题：

- **举例：假设某台机器的特性及其Cache的性能为：**
  - 块大小为1个字
  - 存储器总线宽度为1个字
  - Cache失效率为3%
  - 平均每条指令访存1.2次
  - Cache失效开销为32个时钟周期
  - 平均CPI（忽略Cache失效）为2
- **试问多体交叉和增加存储器宽度对提高性能各有何作用？**

- **假设：**
  - 送地址：4cycles
  - 每个字的访问时间（存取周期）为24cycles,
  - 传送一个字的数据需要4cycles

**如果当把Cache块大小变为2个字时，失效率降为2%；块大小变为4个字时，失效率降为1%。根据前面给出的访问时间，求在采用2路、4路多体交叉存取以及将存储器和总线宽度增加一倍时，性能分别提高多少?**

在改变前的机器中，Cache块大小为一个字，其CPI为：

$$2+(1.2\times3\%\times(4+24+4))=3.15$$

当将块大小增加为2个字时，在下面三种情况下的CPI分别为：

32位总线和存储器，不采用多体交叉：

$2+(1.2\times2\%\times2\times32)=3.54$

32位总线和存储器，采用多体交叉：

$2+(1.2\times2\%\times(4+24+2\times4))=2.86$

性能提高了10%

64位总线和存储器，不采用多体交叉：

$2+(1.2\times2\%\times1\times32)=2.77$

性能提高了14%

如果将块大小增加到4个字，则：

32位总线和存储器，不采用多体交叉：

$2+(1.2\times1\%\times4\times32)=3.54$

32位总线和存储器，采用多体交叉：

$2+(1.2\times1\%\times(4+24+4\times4))=2.53$

性能提高了25%

64位总线和存储器，不采用多体交叉：

$2+(1.2\times1\%\times2\times32)=2.77$

性能提高了14%

# Summary

| Technique | Hit time | Band-width | Miss penalty | Miss rate | Power consumption | Hardware cost/ complexity | Comment |
|---|---|---|---|---|---|---|---|
| Small and simple caches | + | | | – | + | 0 | Trivial; widely used |
| Way-predicting caches | + | | | | + | 1 | Used in Pentium 4 |
| Pipelined cache access | – | + | | | | 1 | Widely used |
| Nonblocking caches | | + | + | | | 3 | Widely used |
| Banked caches | | + | | | + | 1 | Used in L2 of both i7 and Cortex-A8 |
| Critical word first and early restart | | | + | | | 2 | Widely used |
| Merging write buffer | | | + | | | 1 | Widely used with write through |
| Compiler techniques to reduce cache misses | | | | + | | 0 | Software is a challenge, but many compilers handle common linear algebra calculations |
| Hardware prefetching of instructions and data | | | + | + | – | 2 instr., 3 data | Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware. |
| Compiler-controlled prefetching | | | + | + | | 3 | Needs nonblocking cache; possible instruction overhead; in many CPUs |

**Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity.** Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

# review

- **存储器组织**
  - 存储系统结构

**Figure 4.1: Memory System Architecture**

**Figure 1.5:** DIMMs, ranks, banks, and arrays. A system has potentially many DIMMs, each of which may contain one or more ranks. Each rank is a set of ganged DRAM devices, each of which has potentially many banks. Each bank has potentially many constituent arrays, depending on the parts data width.

**Figure 1.7:** The costs of accessing DRAM. Before a DRAM cell can be read or written, the device must first be placed into a state that allows read/write access to the cells. This is a two step process: the **precharge** command initializes the banks sense amplifiers (sets the voltage on the bitlines appropriately), and the **activate** command reads the row data from the specified storage capacitors into the sense amplifiers. Once the sense amplifiers have read the data, it can be read/written by the memory controller. Precharge and activate together take several tens of nanoseconds; reading or writing takes roughly a nanosecond. Timing values taken from a Micron 2Gb DDR3-1066 part.

**参考文献**


Davis, B. T. (2001). Modern dram architectures, University of Michigan: 221.

Jacob, B. (2009). The Memory System, Morgan & Claypool.

# review

## – 交叉存储器组织

**Figure 2.3: Interleaved Memory Banks**

Interleaved memory banks make use of individual control signals to share the address and data bus. In this way a device which is designed with a dedicated interface can share interface signals

# 虚拟存储器 - 基本原理

- **允许应用程序的大小，超过主存容量。目的是提高存储系统的容量**
- **帮助OS进行多进程管理**
  - 每个进程可以有自己的地址空间
  - 提供多个进程空间的保护
  - 可以将多个逻辑块映射到共享的物理存储器上
  - 静态重定位和动态重定位
    - 应用程序运行在虚地址空间
    - 虚实地址转换对用户是透明的
- **虚拟存储管理的是主存 - 辅助存储器这个层面上**
  - 失效：页失效或地址失效
  - 块：页或段

**FIGURE 5.31 The logical program in its contiguous virtual address space is shown on the left.** It consists of four pages A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

# Cache与VM的区别

- **目的不同**
  - Cache是为了提高访存速度
  - VM是为了提高存储容量
- **替换的控制者不同**
  - Cache失效由硬件处理
  - VM的页失效通常由OS处理
    - 一般页失效开销很大，因此替换算法非常重要
- **地址空间**
  - VM空间由CPU的地址尺寸确定
  - Cache的大小与CPU地址尺寸无关
- **下一级存储器**
  - Cache下一级是主存
  - VM下一级是磁盘，大多数磁盘含有文件系统，文件系统寻址与主存不同，它通常在I/O空间中，VM的下一级通常称为SWAP空间

# 虚拟存储器页式管理的典型参数与Cache的比较

| Parameter | First-level cache | Virtual memory |
|---|---|---|
| Block (page) size | 16–128 bytes | 4096–65,536 bytes |
| Hit time | 1–3 clock cycles | 100–200 clock cycles |
| Miss penalty | 8–200 clock cycles | 1,000,000–10,000,000 clock cycles |
| (access time) | (6–160 clock cycles) | (800,000–8,000,000 clock cycles) |
| (transfer time) | (2–40 clock cycles) | (200,000–2,000,000 clock cycles) |
| Miss rate | 0.1–10% | 0.00001–0.001% |
| Address mapping | 25–45 bit physical address to 14–20 bit cache address | 32–64 bit virtual address to 25–45 bit physical address |

**Figure C.19 Typical ranges of parameters for caches and virtual memory.** Virtual memory parameters represent increases of 10–1,000,000 times over cache parameters. Normally first-level caches contain at most 1 MB of data, while physical memory contains 256 MB to 1 TB.

- 从表中看　(与Cache参数相比)
  - 除了失效率较低，其他参数都比Cache大

# 页式管理和段式管理

FIGURE 5.33 Example of how paging and segmentation divide a program.

# 页式管理和段式管理

| Aspect | Page | Segment |
|---|---|---|
| Words/Address | One - contains page and offset | Two - possible large max-size hence need Seg and offset address words |
| Programmer visible | No | Sometimes yes |
| Replacement | Trivial - due to fixed size | Hard - need to find contiguous space ==> GC necessary or wasted memory |
| Memory Inefficiency | Internal fragmentation - wasted part of a page | External fragmentation - due to variable size blocks |
| Disk Efficiency | Yes - adjust page size to balance access and transfer time | Not always - segment size varies |

- VM可分为两类：页式和段式
  - 页式：每页大小固定
  - 段式：每段大小不等
  - 两者区别：

# VM的四个问题 (1/2)

- **映象规则**
  - 选择策略：低失效率和复杂的映象算法，还是简单的映射方法，高失效率
    - 由于失效开销很大，一般选择低失效率方法，即全相联映射
- **查找算法 - 用附加数据结构**
  - 固定页大小 - 用页表
    - VPN -> PPN
    - Tag标识该页是否在主存
  - 可变长段 - 段表
    - 段表中存放所有可能的段信息
    - 段号 ->段基址 再加段内偏移量
    - 可能存在许多小尺寸段
  - 页表
    - 页表中所含项数：一般为虚页的数量
    - 功能：VPN->PPN，方便页重新分配，有一位标识该页是否在内存

# VM的四个问题（2/2）

- **替换规则**
  - LRU是最好的
  - 但真正的LRU方法，硬件代价较大
  - 用硬件简化，通过OS来完成
    - 为了帮助OS寻找LRU页，每个页面设置一个 use bit
    - 当访问主存中一个页面时，其use bit置位
    - OS定期复位所有使用位，这样每次复位之前，使用位的值就反映了从上次复位到现在的这段时间中，哪些页曾被访问过。
    - 当有失效冲突时，由OS来决定哪些页将被换出去。
- **写策略**
  - 总是用写回法，因为访问硬盘速度很慢。

FIGURE 5.35 The mapping of a virtual address to a physical address via a page table.

# 页面大小的选择

- **页面选择较大的优点**
  - 减少了页表的大小
  - 如果局部性较好，可以提高命中率
- **页面选择较大的缺点**
  - 内存中的碎片较多，内存利用率低
  - 进程启动时间长
  - 失效开销加大

# Alpha VPN - >PPN

# TLB （Translation look-aside Buffer）

- **页表一般很大，存放在主存中。**
  - 导致每次访存可能要两次访问主存，一次读取页表项，一次读写数据
  - 解决办法：采用 TLB
- **TLB**
  - 存放近期经常使用的页表项，是整个页表的部分内容的副本。
  - 基本信息：

    VPN##PPN##Protection Field##use bit ## dirty bit
  - OS修改页表项时，需要刷新TLB，或保证TLB中没有该页表项的副本
  - TLB必须在片内
    - 速度至关重要
    - TLB过小，意义不大
    - TLB过大，代价较高
    - 相联度较高（容量小）

# TLB的典型参数

- block size - same as a page table entry - 1 or 2 words
- hit time - 1 cycle
- miss penalty - 10 to 30 cycles
- miss rate - .1% to 2%
- TLB size - 32 B to 8 KB

# 举例：Alpha 21064的TLB

# Summary of Virtual Memory and Caches

**Figure C.24 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access.** The page size is 8 KB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KB, and the L2 cache is a direct-mapped 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache is replication of pieces of this figure.

# Acknowledgements

- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - David Patterson (UCB)
  - Chenxi Zhang (Tongji)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**

# 计算机体系结构

Topic: Instruction level parallelism

# 第5章 指令级并行

- **指令集并行的基本概念及挑战**
- **软件方法挖掘指令集并行**
  - 基本块内的指令集并行
- **硬件方法挖掘指令集并行**
  - Scoreboard
  - Tomasulo
- **跨越基本块的指令集并行**
- **基于硬件的推测执行**
- **以多发射和静态调度来挖掘指令集并行**
- **以动态调度、多发射和推测执行来挖掘指令集并行**

# 系统结构的Flynn分类 (1966)

- **SISD: Single instruction stream, single data stream**
  - 单处理器模式
- **SIMD: Single instruction stream, multiple data streams**
  - 相同的指令作用在不同的数据
  - 可用来挖掘数据级并行(Data Level Parallelism)
  - 如：Vector processors, SIMD instructions, and Graphics processing units
- **MISD: Multiple instruction streams, single data stream**
  - No commercial implementation
- **MIMD: Multiple instruction streams, multiple data streams**
  - 通用性最强的一种结构，可用来挖掘线程级并行、数据级并行.....
  - 组织方式可以是松耦合方式也可以是紧耦合方式

# Levels of Parallelism

- **请求级并行**
  - 多个任务可被分配到多个计算机上并行执行
- **进程级并行**
  - 进程可被调度到不同的处理器上并行执行
- **线程级并行**
  - 任务被组织成多个线程，多个线程共享一个进程的地址空间
  - 每个线程有自己独立的程序计数器和寄存器文件
- **数据级并行**
  - 单线程（逻辑上）中并行处理多个数据 (SIMD/Vector execution)
  - 一个程序计数器, 多个执行部件
- **指令级并行**
  - 针对单一指令流，多个执行部件并行执行不同的指令

# Review: 基本流水线

- **流水线提高的是指令带宽（吞吐率），而不是单条指令的执行速度**
- **相关限制了流水线性能的发挥**
  - 结构相关：需要更多的硬件资源
  - 数据相关：需要定向，编译器调度
  - 控制相关：尽早检测条件，计算目标地址，延迟转移，预测
- **增加流水线的级数会增加相关产生的可能性**
- **异常，浮点运算使得流水线控制更加复杂**
- **编译器可降低数据相关和控制相关的开销**
  - Load 延迟槽
  - Branch 延迟槽
  - Branch预测

# 5.1 指令级并行的基本概念及挑战

- **指令级并行(ILP)： 流水线的平均CPI**
  - Pipeline CPI = Ideal Pipeline CPI + Struct Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls +……
  - 提高指令级并行的方法
    - 软件方法：指令流调度，循环展开 (loop unrolling)，软件流水线，trace scheduling
    - 硬件方法
- **软件方法：指令流调度-循环展开**
    - 指令调度，必须保证程序运行的结果不变
    - 偏移量的修改
    - 寄存器的重命名
    - 循环步长的调整

# 采用的基本技术

| Technique | Reduces | Section |
|---|---|---|
| Forwarding and bypassing | Potential data hazard stalls | A.2 |
| Delayed branches and simple branch scheduling | Control hazard stalls | A.2 |
| Basic dynamic scheduling (scoreboarding) | Data hazard stalls from true dependences | A.8 |
| Dynamic scheduling with renaming | Data hazard stalls and stalls from antidependences and output dependences | 3.2 |
| Dynamic branch prediction | Control stalls | 3.4 |
| Issuing multiple instructions per cycle | Ideal CPI | 3.6 |
| Speculation | Data hazard and control hazard stalls | 3.5 |
| Dynamic memory disambiguation | Data hazard stalls with memory | 3.2, 3.7 |
| Loop unrolling | Control hazard stalls | 4.1 |
| Basic compiler pipeline scheduling | Data hazard stalls | A.2, 4.1 |
| Compiler dependence analysis | Ideal CPI, data hazard stalls | 4.4 |
| Software pipelining, trace scheduling | Ideal CPI, data hazard stalls | 4,3 |
| Compiler speculation | Ideal CPI, data, control stalls | 4.4 |

# 本章遵循的指令延时

| 产生结果的指令 | 使用结果的指令 | 所需延时 |
|---|---|---|
| FP ALU op | Another FP ALU op | 3 |
| FP ALU op | Store double | 2 |
| Load double | FP ALU op | 1 |
| Load double | Store double | 0 |
| Integer op | Integer op | 0 |

# 5.2 基本块内的指令级并行

- **基本块的定义：**
  - 直线型代码，无分支；单入口
  - 程序由分支语句连接基本块构成
- **循环级并行**
  - for (i = 1; i <= 1000; i++)  x(i) = x(i) + s;
  - 计算x(i)时没有数据相关；可以并行产生1000个数据；
  - 问题：在生成代码时会有Branch指令 - 控制相关
  - 预测比较容易，但我们必须有预测方案
  - 向量处理机模型
    - load vectors x and y (up to some machine dependent max)
    - then do result-vec = xvec + yvec in a single instruction

- **Loop unrolling**

# 简单循环及其对应的汇编程序

```
for (i=1; i<=1000; i++)
        x(i) = x(i) + s;
```

| | | | |
|---|---|---|---|
| Loop: | LD | F0,0(R1) | ;F0=vector element |
| | ADDD | F4,F0,F2 | ;add scalar from F2 |
| | SD | 0(R1),F4 | ;store result |
| | SUBI | R1,R1,8 | ;decrement pointer 8B (DW) |
| | BNEZ | R1,Loop | ;branch R1!=zero |
| | NOP | | ;delayed branch slot |

# FP 循环中的相关

```
Loop:   LD      F0,0(R1)      ;F0=vector element
        ADDD    F4,F0,F2      ;add scalar from F2
        SD      0(R1),F4      ;store result
        SUBI    R1,R1,8       ;decrement pointer 8B (DW)
        BNEZ    R1,Loop       ;branch R1!=zero
        NOP                   ;delayed branch slot
```

| 产生结果的指令 | 使用结果的指令 | 所需延时 |
|---|---|---|
| FP ALU op | Another FP ALU op | 3 |
| FP ALU op | Store double | 2 |
| Load double | FP ALU op | 1 |
| Load double | Store double | 0 |
| Integer op | Integer op | 0 |

# FP 循环中的Stalls

```
1 Loop:   LD      F0,0(R1)      ;F0=vector element
2         stall
3         ADDD    F4,F0,F2      ;add scalar in F2
4         stall
5         stall
6         SD      0(R1),F4      ;store result
7         SUBI    R1,R1,8       ;decrement pointer 8B (DW)
8         stall                 ;
9         BNEZ    R1,Loop       ;branch R1!=zero
10        stall                 ;delayed branch slot
```

| *产生结果的指令* | *使用结果的指令* | *所需的延时* |
|---|---|---|
| FP ALU op | Another FP ALU op | 3 |
| FP ALU op | Store double | 2 |
| Load double | FP ALU op | 1 |
| Load double | Store double | 0 |
| Integer op | Integer op | 0 |

10 clocks: 是否可以通过调整代码顺序使stalls减到最小？

# FP 循环中的最少Stalls数

```
1 Loop:  LD      F0,0(R1)
2        SUBI    R1,R1,8
3        ADDD    F4,F0,F2
4        stall
5        BNEZ    R1,Loop ;delayed branch
6        SD      8(R1),F4  ;altered when move past SUBI
```

**Swap BNEZ and SD by changing address of SD**

**6 clocks: 通过循环展开4次是否可以提高性能?**

# 循环展开4次(straightforward way)

```
1 Loop:  LD      F0,0(R1)     stall
2        ADDD    F4,F0,F2     stall   stall
3        SD      0(R1),F4             ;drop SUBI & BNEZ
4        LD      F6,-8(R1)    stall
5        ADDD    F8,F6,F2     stall stall
6        SD      -8(R1),F8            ;drop SUBI & BNEZ
7        LD      F10,-16(R1)  stall
8        ADDD    F12,F10,F2   stall stall
9        SD      -16(R1),F12          ;drop SUBI & BNEZ
10       LD      F14,-24(R1)  stall
11       ADDD    F16,F14,F2   stall  stall
12       SD      -24(R1),F16
13       SUBI    R1,R1,#32    stall      ;alter to 4*8
14       BNEZ    R1,LOOP
15       NOP
```

Rewrite loop to minimize stalls?

**15 + 4 x (1+2) + 1 = 28 cycles, or 7 per iteration**

**Assumes R1 is multiple of 4**

# Stalls数最小的循环展开

```
1 Loop:  LD      F0,0(R1)
2        LD      F6,-8(R1)
3        LD      F10,-16(R1)
4        LD      F14,-24(R1)
5        ADDD    F4,F0,F2
6        ADDD    F8,F6,F2
7        ADDD    F12,F10,F2
8        ADDD    F16,F14,F2
9        SD      0(R1),F4
10       SD      -8(R1),F8
11       SUBI    R1,R1,#32
12       SD      16(R1),F12
13       BNEZ    R1,LOOP
14       SD      8(R1),F16        ; 8-32 = -24
```

- 代码移动后
  - SD移动到SUBI后，注意偏移量的修改
  - Loads移动到SD前，注意偏移量的修改

**14 clock cycles, or 3.5 per iteration**

# 循环展开示例小结

- 循环展开对循环间无关的程序是有效降低stalls的手段(对循环级并行）.
- 指令调度，必须保证程序运行的结果不变
- 注意循环展开中的Load和Store,不同次循环的Load 和Store 是相互独立的。需要分析对存储器的引用，保证他们没有引用同一地址.
- 不同次的循环，使用不同的寄存器
- 删除不必要的测试和分支后，需调整循环步长等控制循环的代码.
- 移动SD到SUBI和BNEZ后，需要调整SD中的偏移

# -Review

- **指令级并行(ILP)： 流水线的平均CPI**
  - Pipeline CPI = Ideal Pipeline CPI + Struct Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls +……
  - 提高指令级并行的方法
    - 软件方法：指令流调度，循环展开，软件流水线，trace scheduling
    - 硬件方法
- **软件方法：指令流调度-循环展开**
    - 指令调度，必须保证程序运行的结果不变
    - 偏移量的修改
    - 寄存器的重命名
    - 循环步长的调整

# 从编译器角度看代码移动（1/4)

- **编译器分析程序的相关性依赖于给定的流水线**
- **编译器进行指令调度来消除相关**
- **(True) 数据相关（Data dependencies）**
  - 对于指令i和j，如果指令 j使用指令i产生的结果，或指令 j 与指令 k相关, 并且指令 k 与指令 i有数据相关.
- **如果相关, 不能并行执行**

- **对于寄存器比较容易确定(fixed names)**
- **但对memory的引用，比较难确定:**
  - 100(R4) = 20(R6)?
  - 在不同次的循环中，20(R6) = 20(R6)?

# 下列程序哪里有数据相关?

```
1 Loop:   LD      F0,0(R1)
2         ADDD    F4,F0,F2
3         SUBI    R1,R1,8
4         BNEZ    R1,Loop    ;delayed branch
5         SD      8(R1),F4   ;altered when move past SUBI
```

# 从编译器角度看代码移动(2/4)

- **另一种相关称为名相关（name dependence）：两条指令使用同名参数(register or memory location)但不交换数据**
  - 反相关（Antidependence） (WAR if a hazard for HW)
    - Instruction j 所写的寄存器或存储单元，与 instruction i 所读的寄存器或存储单元相同，注意instruction i 是先执行

  - 输出相关(Output dependence) (WAW if a hazard for HW)
    - Instruction i 和instruction j 对同一寄存器或存储单元进行写操作，必须保证两条指令的写顺序

# 下列是否有名相关?

```
1 Loop: LD      F0,0(R1)
2       ADDD    F4,F0,F2
3       SD      0(R1),F4        ;drop SUBI & BNEZ
4       LD      F0,-8(R1)
5       ADDD    F4,F0,F2
6       SD      -8(R1),F4       ;drop SUBI & BNEZ
7       LD      F0,-16(R1)
8       ADDD    F4,F0,F2
9       SD      -16(R1),F4      ;drop SUBI & BNEZ
10      LD      F0,-24(R1)
11      ADDD    F4,F0,F2
12      SD      -24(R1),F4
13      SUBI    R1,R1,#32       ;alter to 4*8
14      BNEZ    R1,LOOP
15      NOP
```

**如何消除名相关?**

# 下列是否存在名相关?

```
1 Loop:LD     F0,0(R1)
2      ADDD   F4,F0,F2
3      SD     0(R1),F4        ;drop SUBI & BNEZ
4      LD     F6,-8(R1)
5      ADDD   F8,F6,F2
6      SD     -8(R1),F8       ;drop SUBI & BNEZ
7      LD     F10,-16(R1)
8      ADDD   F12,F10,F2
9      SD     -16(R1),F12     ;drop SUBI & BNEZ
10     LD     F14,-24(R1)
11     ADDD   F16,F14,F2
12     SD     -24(R1),F16
13     SUBI   R1,R1,#32       ;alter to 4*8
14     BNEZ   R1,LOOP
15     NOP
```

这种方法称为寄存器重命名（register renaming）

# 从编译器角度看代码移动（3/4)

- **访问存储单元时，很难判断名相关**
  - 100(R4) = 20(R6)?
  - 不同次的循环，20(R6) = 20(R6)?
- **我们给出的示例要求编译器知道假设R1不变，因此:**

$$0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$$

  - 因此loads和stores之间相互无关可以移动

# 从编译器角度看代码移动（4/4)

- **最后一种相关称为控制相关( control dependence)**
- **Example**
- **if p1 {S1;};**
- **if p2 {S2;};**
- **S1 依赖于P1的测试结果，S2依赖于P2的测试。**
- **处理控制相关的原则：**
  - 受分支指令控制的指令，不能移到控制指令之前，以免该指令的执行不在分支指令的控制范围.
  - 不受分支指令控制的指令，不能移到控制指令之后，以免该指令的执行受分支指令的控制.
- **减少控制相关可以提高指令的并行性**

# 下列程序段的控制相关

```
1 Loop:  LD     F0,0(R1)
2        ADDD   F4,F0,F2
3        SD     0(R1),F4
4        SUBI   R1,R1,8
5        BEQZ   R1,exit
6        LD     F0,0(R1)
7        ADDD   F4,F0,F2
8        SD     0(R1),F4
9        SUBI   R1,R1,8
10       BEQZ   R1,exit
```

```
11   LD     F0,0(R1)
12   ADDD   F4,F0,F2
13   SD     0(R1),F4
14   SUBI   R1,R1,8
15   BEQZ   R1,exit
....
```

# 循环间相关（1/4）

Example: 下列程序段存在哪些数据相关?
(A,B,C 指向不同的存储区且不存在覆盖区)

```
for (i=1; i<=100; i=i+1) {
        A[i+1] = A[i] + C[i];        /* S1 */
        B[i+1] = B[i] + A[i+1];      /* S2 */
}
```

Dependency Graph

1. S2使用由S1在同一循环计算出的 A[i+1].

2. S1 使用由S1在前一次循环中计算的值；S2也使用由S2在前一次循环中计算的值. 这种存在于循环间的相关，我们称为 “loop-carried dependence”这表示循环间存在相关，不能并行执行，它与我们前面的例子中循环间无关是有区别的

# 无回路的循环间相关（2/4)-

Example:A,B,C,D distinct & nonoverlapping

```
for (i=1; i<=100; i=i+1) {
        A[i] = A[i] + B[i];    /* S1 */
        B[i+1] = C[i] + D[i];} /* S2 */
```

Non-Circular Loop-Carried Dependence

Dependency Graph

1. S1和S2没有相关，S1和S2互换不会影响程序的正确性
2. 在第一次循环中，S1依赖于前一次循环的B[1].
3. S1依赖上一次循环的S2，但S2不依赖S1

# 循环间相关（3/4）-循环变换

OLD:

```
for (i=1; i<=100; i=i+1) {
        A[i] = A[i] + B[i];    /* S1 */
        B[i+1] = C[i] + D[i];} /* S2 */
```

NEW:

```
A[1] = A[1] + B[1];
        for     (i=1; i<=99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
        B[101] = C[100] + D[100];
```

# 循环间相关（4/4）-Dependence Distance

通常循环间相关呈现为递推关系

```
for (i=1; i<N; i++) A[i] = A[i-1] + B[i];
```

相关的距离可能大于1

```
for (i=4; i<N; i++) A[i] = A[i-4] + B[i];
```

可以通过循环展开增加循环内的并行性

```
for (i=4; i<N; i=i+4)
{    A[i]  = A[i-4] + B[i];
     A[i+1] = A[i-3] + B[i+1];
     A[i+2] = A[i-2] + B[i+2];
     A[i+3] = A[i-1] + B[i+3];
}
```

# 5.3 硬件方案: 指令级并行

- **为什么要使用硬件调度方案?**
  - 在编译时无法确定的相关，可以通过硬件调度来优化
  - 编译器简单
  - 代码在不同组织结构的机器上，同样可以有效的运行
- **基本思想: 允许 stall后的指令继续向前流动**

```
DIVD    F0,F2,F4
ADDD    F10,F0,F8
SUBD    F12,F8,F14
```

  - 允许乱序执行（out-of-order execution） => out-of-order completion

# 硬件方案之一: 记分牌

- 记分牌的基本概念示意图

# 记分牌技术要点(1/2)

- **Out-of-order execution 将ID 段分为:**
  - Issue—译码，检测结构相关
  - Read operands—等待到无数据相关时，读操作数
- **起源于1963年推出的CDC6600**
  - 4 FPU
  - 2 Memory Reference
  - 4 IU
- **集中相关检查，互锁机制解决相关**
- **CDC 6600: 顺序发射，乱序执行，乱序完成，CDC6600流水线没有采用定向技术，只实现非精确中断**
- **Load /store结构**
- **采用这种技术的微处理器企业**
  - MIPS，HP， IBM
  - Sun 公司的UltraSparc
  - DEC Alpha

# 记分牌技术要点(2/2)

- **Out-of-order completion => WAR, WAW hazards?**
- **WAR的一般解决方案**
  - 对操作排队
  - 仅在读操作数阶段读寄存器
- **对WAW而言, 检测到相关后，停止发射后一条指令，直到前一条指令完成**
- **要提高效率，需要有多条指令进入执行阶段=>必须有多个执行部件或执行部件是流水化的**
- **记分牌保存相关操作和状态**
- **记分牌用四段代替ID, EX, WB 三段**

# 带有记分牌控制的DLX

Note: this model could support both single or multi-issue

Exception is that one multiply will be issued per cycle

All depends on bus/trunk structure

# 记分牌控制的四阶段(1/2)

- **1.Issue—指令译码，检测结构相关**
  - 如果当前指令所使用的功能部件空闲，并且没有其他活动的指令使用相同的目的寄存器（WAW), 记分牌发射该指令到功能部件，并更新记分牌内部数据，如果有结构相关或WAW相关，则该指令的发射暂停，并且也不发射后继指令，直到相关解除.
- **2. Read operands—没有数据相关时，读操作数**
  - 如果先前已发射的正在运行的指令不对当前指令的源操作数寄存器进行写操作，或者一个正在工作的功能部件已经完成了对该寄存器的写操作，则该操作数有效。操作数有效时，记分牌控制功能部件读操作数，准备执行。
  - 记分牌在这一步动态地解决了RAW相关，指令可能会乱序执行。

# 记分牌控制的四阶段（2/2）

- **3.Execution—取到操作数后执行 (EX)**
  - 接收到操作数后，功能部件开始执行. 当计算出结果后，它通知记分牌，可以结束该条指令的执行.
- **4.Write result—finish execution (WB)**
  - 一旦记分牌得到功能部件执行完毕的信息后，记分牌检测WAR相关，如果没有WAR相关，就写结果，如果有WAR 相关，则暂停该条指令。
  - Example:

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F8,F8,F14
```

  - CDC 6600 scoreboard 将暂停 SUBD 直到ADDD 读取操作数后，才进入WR段处理。

# 记分牌的结构

1.Instruction status—记录正在执行的各条指令所处的状态步

2.Functional unit status—记录功能部件(FU)的状态。用9个域记录每个功能部件的9个参量：

Busy—指示该部件是否空闲

Op—该部件所完成的操作

Fi—其目的寄存器编号

Fj, Fk—源寄存器编号

Qj, Qk—产生源操作数Fj, Fk的功能部件

Rj, Rk—标识源操作数Fj, Fk是否就绪的标志

3.Register result status—如果存在功能部件对某一寄存器进行写操作，指示具体是哪个功能部件对该寄存器进行写操作。如果没有指令对该寄存器进行写操作，则该域 为Blank

# 记分牌流水线控制

| Instruction status | Wait until | Bookkeeping |
|---|---|---|
| **Issue** | Not busy (FU) and not result(D) | Busy(FU)← yes; Op(FU)← op; Fi(FU)← `D'; Fj(FU)← `S1'; Fk(FU)← `S2'; Qj← Result('S1'); Qk← Result(`S2'); Rj← not Qj; Rk← not Qk; Result('D')← FU; |
| **Read operands** | Rj and Rk | Rj← No; Rk← No |
| **Execution complete** | Functional unit done | |
| **Write result** | ∀f((Fj( f )≠Fi(FU) or Rj( f )=No) & (Fk( f ) ≠Fi(FU) or Rk( f )=No)) | ∀f(if Qj(f)=FU then Rj(f)← Yes); ∀f(if Qk(f)=FU then Rk(f)← Yes); Result(Fi(FU))← 0; Busy(FU)← No |

*

# Scoreboard Example

Instruction status

| Instruction | | j | k | Issue | Read operan | Executi comple | Write Result |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | | | | |
| LD | F2 | 45+ | R3 | | | | |
| MULTI | F0 | F2 | F4 | | | | |
| SUBD | F8 | F6 | F2 | | | | |
| DIVD | F10 | F0 | F6 | | | | |
| ADDD | F6 | F8 | F2 | | | | |

Functional unit status

| Time | Name | Busy | Op | dest Fi | S1 Fj | S2 Fk | FU for j Qj | FU for k Qk | Fj? Rj | Fk? Rk |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| | Mult1 | No | | | | | | | | |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| | Divide | No | | | | | | | | |

Register result status

| Clock | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
|---|---|---|---|---|---|---|---|---|---|
| FU | | | | | | | | | |

# Scoreboard Example: Cycle 1

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | | | |
| LD | F2 | 45+ | R3 | | | | |
| MULTD | F0 | F2 | F4 | | | | |
| SUBD | F8 | F6 | F2 | | | | |
| DIVD | F10 | F0 | F6 | | | | |
| ADDD | F6 | F8 | F2 | | | | |

- Issue LD!

*Functional unit status:*

| *Time Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj?* | *Fk?* |
|---|---|---|---|---|---|---|---|---|---|
| Integer | Yes | Load | F6 | | R2 | | | | Yes |
| Mult1 | No | | | | | | | | |
| Mult2 | No | | | | | | | | |
| Add | No | | | | | | | | |
| Divide | No | | | | | | | | |

*Register result status:*

Clock 1

| | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|
| *FU* | | | | Integer | | | | | |

# Scoreboard Example: Cycle 2

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | | |
| LD | F2 | 45+ | R3 | | | | |
| MULTD | F0 | F2 | F4 | | | | |
| SUBD | F8 | F6 | F2 | | | | |
| DIVD | F10 | F0 | F6 | | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | Yes | Load | F6 | | R2 | | | | Yes |
| | Mult1 | No | | | | | | | | |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| | Divide | No | | | | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **2** | *FU* | | | | Integer | | | | | |

- Issue 2nd LD?

# Scoreboard Example: Cycle 3

*Instruction status:*

| Instruction | | j | k | Issue | Read Oper | Exec Comp | Write Result |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | |
| LD | F2 | 45+ | R3 | | | | |
| MULTD | F0 | F2 | F4 | | | | |
| SUBD | F8 | F6 | F2 | | | | |
| DIVD | F10 | F0 | F6 | | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| Time | Name | Busy | Op | dest Fi | S1 Fj | S2 Fk | FU Qj | FU Qk | Fj? Rj | Fk? Rk |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | Yes | Load | F6 | | R2 | | | | No |
| | Mult1 | No | | | | | | | | |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| | Divide | No | | | | | | | | |

*Register result status:*

| Clock | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | FU | Integer | | | | | | | | |

- Issue MULT?

# Scoreboard Example: Cycle 4

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | | | | |
| MULTD | F0 | F2 | F4 | | | | |
| SUBD | F8 | F6 | F2 | | | | |
| DIVD | F10 | F0 | F6 | | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| | Mult1 | No | | | | | | | | |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| | Divide | No | | | | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **4** | *FU* | | | | Integer | | | | | |

# Scoreboard Example: Cycle 5

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | | | |
| MULTD | F0 | F2 | F4 | | | | |
| SUBD | F8 | F6 | F2 | | | | |
| DIVD | F10 | F0 | F6 | | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | Yes | Load | F2 | | R3 | | | | Yes |
| | Mult1 | No | | | | | | | | |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| | Divide | No | | | | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **5** | *FU* | | Integer | | | | | | | |

- Issue 2nd LD!

# Scoreboard Example: Cycle 6

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | | |
| MULTD | F0 | F2 | F4 | 6 | | | |
| SUBD | F8 | F6 | F2 | | | | |
| DIVD | F10 | F0 | F6 | | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | Yes | Load | F2 | | R3 | | | | Yes |
| | Mult1 | Yes | Mult | F0 | F2 | F4 | Integer | | No | Yes |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| | Divide | No | | | | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *…* | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **6** | *FU* | Mult1 | Integer | | | | | | | |

- **Issue Mult!**

# Scoreboard Example: Cycle 7

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | |
| MULTD | F0 | F2 | F4 | 6 | | | |
| SUBD | F8 | F6 | F2 | 7 | | | |
| DIVD | F10 | F0 | F6 | | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | Yes | Load | F2 | | R3 | | | | No |
| | Mult1 | Yes | Mult | F0 | F2 | F4 | Integer | | No | Yes |
| | Mult2 | No | | | | | | | | |
| | Add | Yes | Sub | F8 | F6 | F2 | | Integer | Yes | No |
| | Divide | No | | | | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *...* | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 | *FU* | Mult1 | Integer | | | Add | | | | |

- Issue SUBD! Read MULTD operands?

# Scoreboard Example: Cycle 8a (First half of clock cycle)

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | |
| MULTD | F0 | F2 | F4 | 6 | | | |
| SUBD | F8 | F6 | F2 | 7 | | | |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | Yes | Load | F2 | | R3 | | | | No |
| | Mult1 | Yes | Mult | F0 | F2 | F4 | Integer | | No | Yes |
| | Mult2 | No | | | | | | | | |
| | Add | Yes | Sub | F8 | F6 | F2 | | Integer | Yes | No |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **8** | *FU* | Mult1 | Integer | | | Add | Divide | | | |

- Issue DIVD

# Scoreboard Example: Cycle 8b (Second half of clock cycle)

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | | | |
| SUBD | F8 | F6 | F2 | 7 | | | |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| | Mult1 | Yes | Mult | F0 | F2 | F4 | | | Yes | Yes |
| | Mult2 | No | | | | | | | | |
| | Add | Yes | Sub | F8 | F6 | F2 | | | Yes | Yes |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **8** | *FU* | Mult1 | | | | Add | Divide | | | |

- Complete 2nd LD

# Scoreboard Example: Cycle 9

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | | |
| SUBD | F8 | F6 | F2 | 7 | 9 | | |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

Note Remaining

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 10 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | Yes | Yes |
| | Mult2 | No | | | | | | | | |
| 2 | Add | Yes | Sub | F8 | F6 | F2 | | | Yes | Yes |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **9** | *FU* | Mult1 | | | | Add | Divide | | | |

- Read operands for MULT & SUB! Issue ADDD?

# Scoreboard Example: Cycle 10

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | | |
| SUBD | F8 | F6 | F2 | 7 | 9 | | |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 9 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| | Mult2 | No | | | | | | | | |
| 1 | Add | Yes | Sub | F8 | F6 | F2 | | | No | No |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *...* | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **10** | *FU* | Mult1 | | | | Add | Divide | | | |

# Scoreboard Example: Cycle 11

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | | |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 8 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| | Mult2 | No | | | | | | | | |
| 0 | Add | Yes | Sub | F8 | F6 | F2 | | | No | No |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *...* | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **11** | *FU* | Mult1 | | | | Add | Divide | | | |

# Scoreboard Example: Cycle 12

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | | |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 7 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **12** | *FU* | Mult1 | | | | | Divide | | | |

- Read operands for DIVD?

# Scoreboard Example: Cycle 13

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | | |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | 13 | | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 6 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| | Mult2 | No | | | | | | | | |
| | Add | Yes | Add | F6 | F8 | F2 | | | Yes | Yes |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **13** | *FU* | Mult1 | | | Add | | Divide | | | |

- **Issue ADDD!**

# Scoreboard Example: Cycle 14

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | | |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | 13 | 14 | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 5 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| | Mult2 | No | | | | | | | | |
| 2 | Add | Yes | Add | F6 | F8 | F2 | | | Yes | Yes |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *…* | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **14** | *FU* | Mult1 | | | Add | | Divide | | | |

- Read Operands ADDD!

# Scoreboard Example: Cycle 15

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | | |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | 13 | 14 | | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 4 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| | Mult2 | No | | | | | | | | |
| 1 | Add | Yes | Add | F6 | F8 | F2 | | | No | No |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *...* | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **15** | *FU* | Mult1 | | | Add | | Divide | | | |

# Scoreboard Example: Cycle 16

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | | |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 3 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| | Mult2 | No | | | | | | | | |
| 0 | Add | Yes | Add | F6 | F8 | F2 | | | No | No |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *…* | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **16** | *FU* | Mult1 | | | Add | | Divide | | | |

- Completes ADDD!

# Scoreboard Example: Cycle 17

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | | |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | |

WAR Hazard!

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 2 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| | Mult2 | No | | | | | | | | |
| | Add | Yes | Add | F6 | F8 | F2 | | | No | No |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *…* | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **17** | *FU* | Mult1 | | | Add | | Divide | | | |

- Why not write result of ADD???

# Scoreboard Example: Cycle 18

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | | |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 1 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| | Mult2 | No | | | | | | | | |
| | Add | Yes | Add | F6 | F8 | F2 | | | No | No |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *...* | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **18** | *FU* | Mult1 | | | Add | | Divide | | | |

# Scoreboard Example: Cycle 19

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | 19 | |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| 0 | Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| | Mult2 | No | | | | | | | | |
| | Add | Yes | Add | F6 | F8 | F2 | | | No | No |
| | Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **19** | *FU* | Mult1 | | | Add | | Divide | | | |

- Completes MULTD!

# Scoreboard Example: Cycle 20

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | 19 | 20 |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | | | |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | |

*Functional unit status:*

| *Time Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|
| Integer | No | | | | | | | | |
| Mult1 | No | | | | | | | | |
| Mult2 | No | | | | | | | | |
| Add | Yes | Add | F6 | F8 | F2 | | | No | No |
| Divide | Yes | Div | F10 | F0 | F6 | | | Yes | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | *…* | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **20** | *FU* | | | | Add | | Divide | | | |

# Scoreboard Example: Cycle 21

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | 19 | 20 |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | 21 | | |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| | Mult1 | No | | | | | | | | |
| | Mult2 | No | | | | | | | | |
| | Add | Yes | Add | F6 | F8 | F2 | | | No | No |
| | Divide | Yes | Div | F10 | F0 | F6 | | | Yes | Yes |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **21** | *FU* | | | | Add | | Divide | | | |

- Read Operands DIVD; WAR Hazard is now gone…

# Scoreboard Example: Cycle 22

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | 19 | 20 |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | 21 | | |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | 22 |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| | Mult1 | No | | | | | | | | |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| 39 | Divide | Yes | Div | F10 | F0 | F6 | | | No | No |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **22** | *FU* | | | | | | Divide | | | |

- Write Result ADDD; DIVD in execution

**Continue…….**

# Scoreboard Example: Cycle 61

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | 19 | 20 |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | 21 | 61 | |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | 22 |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| | Mult1 | No | | | | | | | | |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| 0 | Divide | Yes | Div | F10 | F0 | F6 | | | No | No |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **61** | *FU* | | | | | | Divide | | | |

- Complete DIVD

# Scoreboard Example: Cycle 62

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | 19 | 20 |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | 21 | 61 | 62 |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | 22 |

*Functional unit status:*

| *Time* | *Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| | Mult1 | No | | | | | | | | |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| | Divide | No | | | | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **62** | *FU* | | | | | | | | | |

- Write Result DIVD

# Review: Scoreboard Example: Cycle 62

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 |
| MULTD | F0 | F2 | F4 | 6 | 9 | 19 | 20 |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 |
| DIVD | F10 | F0 | F6 | 8 | 21 | 61 | 62 |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | 22 |

*Functional unit status:*

| *Time Name* | *Busy* | *Op* | *dest Fi* | *S1 Fj* | *S2 Fk* | *FU Qj* | *FU Qk* | *Fj? Rj* | *Fk? Rk* |
|---|---|---|---|---|---|---|---|---|---|
| Integer | No | | | | | | | | |
| Mult1 | No | | | | | | | | |
| Mult2 | No | | | | | | | | |
| Add | No | | | | | | | | |
| Divide | No | | | | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **62** | *FU* | | | | | | | | | |

- In-order issue; out-of-order execute & commit

# 为什么顺序发射?

- **顺序发射使我们可以进行程序的数据流分析**
  - 我们可以知道某条指令的结果会流向哪些指令
  - 如果我们乱序发射，可能会混淆RAW和WAR相关
- **每一周期发射多条指令也使用该原则将会正确地工作:**
  - 需要多端口的 “rename table” ,以同时对一组指令所用的寄存器重命名
  - 需要在单周期内发射到多个RS中.
  - 寄存器文件需要有2x 个读端口和x个写端口.

# 记分牌控制的四阶段(1/2)

- **1.Issue—指令译码，检测结构相关**
  - 如果当前指令所使用的功能部件空闲，并且没有其他活动的指令使用相同的目的寄存器（WAW), 记分牌发射该指令到功能部件，并更新记分牌内部数据，如果有结构相关或WAW相关，则该指令的发射暂停，并且也不发射后继指令，直到相关解除.
- **2. Read operands—没有数据相关时，读操作数**
  - 如果先前已发射的正在运行的指令不对当前指令的源操作数寄存器进行写操作，或者一个正在工作的功能部件已经完成了对该寄存器的写操作，则该操作数有效。操作数有效时，记分牌控制功能部件读操作数，准备执行。
  - 记分牌在这一步动态地解决了RAW相关，指令可能会乱序执行。

# 记分牌控制的四阶段（2/2）

- **3.Execution—取到操作数后执行 (EX)**
  - 接收到操作数后，功能部件开始执行. 当计算出结果后，它通知记分牌，可以结束该条指令的执行.
- **4.Write result—finish execution (WB)**
  - 一旦记分牌得到功能部件执行完毕的信息后，记分牌检测WAR相关，如果没有WAR相关，就写结果，如果有WAR 相关，则暂停该条指令。
  - Example:

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F8,F8,F14
```

  - CDC 6600 scoreboard 将暂停 SUBD 直到ADDD 读取操作数后，才进入WR段处理。

# 记分牌的结构

1.Instruction status—记录正在执行的各条指令所处的状态步

2.Functional unit status—记录功能部件(FU)的状态。用9个域记录每个功能部件的9个参量：

Busy—指示该部件是否空闲

Op—该部件所完成的操作

Fi—其目的寄存器编号

Fj, Fk—源寄存器编号

Qj, Qk—产生源操作数Fj, Fk的功能部件

Rj, Rk—标识源操作数Fj, Fk是否就绪的标志

3.Register result status—如果存在功能部件对某一寄存器进行写操作，指示具体是哪个功能部件对该寄存器进行写操作。如果没有指令对该寄存器进行写操作，则该域 为Blank

# 记分牌流水线控制

| Instruction status | Wait until | Bookkeeping |
|---|---|---|
| **Issue** | Not busy (FU) and not result(D) | Busy(FU)← yes; Op(FU)← op; Fi(FU)← `D'; Fj(FU)← `S1'; Fk(FU)← `S2'; Qj← Result('S1'); Qk← Result(`S2'); Rj← not Qj; Rk← not Qk; Result('D')← FU; |
| **Read operands** | Rj and Rk | Rj← No; Rk← No |
| **Execution complete** | Functional unit done | |
| **Write result** | ∀f((Fj( f )≠Fi(FU) or Rj( f )=No) & (Fk( f ) ≠Fi(FU) or Rk( f )=No)) | ∀f(if Qj(f)=FU then Rj(f)← Yes); ∀f(if Qk(f)=FU then Rk(f)← Yes); Result(Fi(FU))← 0; Busy(FU)← No |

*

# Review

- **硬件方法挖掘ILP**
  - 编译阶段无法确定的相关性，在程序执行时，用硬件方法判定
  - 可以使得程序代码在其他机器上有效地执行
- **记分牌的主要思想：允许stall后的指令继续**
  - 乱序执行(out-of-order execution) => 乱序完成(out-of-order completion)
  - 发射前检测结构相关和WAW相关
  - 读操作数前检测RAW相关
  - 写结果前处理WAR相关

# CDC 6600 Scoreboard

**CDC 6600 scoreboard的主要缺陷:**

- **没有定向数据通路**
- **指令窗口较小，仅局限于基本块内的调度**
- **功能部件数较少**
  - 容易产生结构相关，
  - 其Load/store操作也是用IU部件完成的
- **结构冲突时不能发射**
- **WAR相关是通过等待解决的**
- **WAW相关时，不会进入IS阶段**

# 动态调度方案之二：Tomasulo Algorithm

- **该算法首次在 IBM 360/91上使用（ CDC6600推出三年后）**
- **目标: 在没有专用编译器的情况下，提高系统性能**
- **IBM 360 & CDC 6600 ISA的差别**
  - IBM360只有 2位寄存器描述符 vs. CDC 6600寄存器描述符3位
  - IBM360　4个FP 寄存器 vs. CDC 6600 8个
  - IBM 360 有memory-register 操作
- **Alpha 21264, HP 8000, MIPS 10000, Pentium II, PowerPC 604，…**

# Tomasulo Algorithm vs. Scoreboard

- **控制和缓存分布在各部件中 vs. 控制和缓存集中在记分牌**
  - FU 缓存称“reservation stations”; 保存待用操作数
- **指令中的寄存器在RS中用寄存器值或指向RS的指针代替（称为 register renaming）**
  - 避免 WAR, WAW hazards
  - RS多于寄存器，因此可以做更多编译器无法做的优化
- **传给FU的结果从RS来而不是从寄存器来**
  - FU的计算结果通过Common Data Bus 以广播方式发向所有功能部件
- **Load和Store部件也看作带有RS的功能部件**
- **可以跨越分支，允许FP操作队列中FP操作不仅仅局限于基本块**

# Tomasulo Organization

# Tomasulo Organization (cont.)

# Reservation Station 结构

Op: 部件所进行的操作

Vj, Vk: 源操作数的值。Store 缓冲区有Vk域，用于存放要写入存储器的值

A: 存放存储器地址。开始存立即数，计算出有效地址后，存放有效地址

Qj, Qk: 产生源操作数的RS

注：没有记分牌中的准备就绪标志， Qj, Qk=0 => ready

Store 缓存区中Qk表示产生结果的RS

Busy: 标识RS或FU是否空闲

Register result status—如果存在对寄存器的写操作，指示对该寄存器进行写操作的部件.

Qi: 保留站的编号

# Tomasulo 算法的三阶段

1. Issue—从FP操作队列中取指令

   如果RS空闲(no structural hazard), 则控制发射指令和操作数(renames registers). 消除WAR，WAW相关

2. Execution—operate on operands (EX)

   当两操作数就绪后，就可以执行
   如果没有准备好，则监测Common Data Bus 以获取结果。通过推迟指令执行避免RAW相关

3. Write result—finish execution (WB)

   将结果通过Common Data Bus传给所有等待该结果的部件;
   表示RS可用

- 通常的数据总线: data + destination (“go to” bus)
- Common data bus: data + source (“come from” bus)
  - 64 bits 数据线 + 4 bits 功能部件源地址（ FU source address）
  - 产生结果的部件如果与RS中等待的部件匹配，就进行写操作
  - 广播方式传送

# Tomasulo 算法流水线控制

rs, rt : 源寄存器名；rd:目的寄存器名
RS: 保留站数据结构；r:保留站编号
RegisterStat：寄存器结果状态表
Reg： 寄存器组

1. Issue

FP Operation:

Wait until： Station r empty

Action or bookkeeping:

1st 操作数

2nd 操作数

```
if(RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi}
 else {RS[r].Vj ←Reg[rs]; RS[r].Qj ←0 }
if(RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi}
else {RS[r].Vk ←Reg[rt]; RS[r].Qk←0 }
RS[r].Busy ← yes; RegisterStat[rd].Qi = r;
```

Load or Store:

Wait until: Buffer r empty

Action or bookkeeping:

基址寄存器

```
if(RegisterStat[rs].Qi≠0)
  {RS[r].Qj ← RegisterStat[rs].Qi}
else {RS[r].Vj ←Reg[rs]; RS[r].Qj ←0 }
RS[r].A ←imm;   RS[r].Busy ← yes;
```

Load only: RegisterStat[rt].Qi = r;

Store only:

需写入的
数据寄存器

```
if(RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi}
 else {RS[r].Vk ←Reg[rt]; RS[r].Qk←0 }
```

# 注意：Load操作在EXE阶段分两步

2、Execute

FP Operation

wait until: (RS[r].Qj=0) and (RS[r].Qk=0)

Action or bookkeeping:

computer result: Operands are in Vj and Vk

Load-store step1

wait until: RS[r].Qj =0 & r is head of load-store queue

Action or bookkeeping:

RS[r].A ← RS[r].Vj + RS[r].A;

Load step2

wait until: Load Step1 complete

Action or bookkeeping:

Read from Mem[RS[r].A]

3、Write result

FP Operation or Load

Wait until: Execution complete at r & CDB available

Action or bookkeeping

∀x (if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0})

∀ x (if(RS[x].Qj =r) {RS[x].Vj ←result; RS[x].Qj ←0});

∀ x (if(RS[x].Qk =r) {RS[x].Vk ←result; RS[x].Qk ←0});

RS[r].Busy ← no;

Store

wait until:

Execution complete at r & RS[r].Qk = 0

Action or bookkeeping

Mem[RS[r].A] ← RS[r].Vk;

RS[r].Busy ← no;

# Tomasulo Example

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | | | |
| LD | F2 | 45+ | R3 | | | |
| MULTD | F0 | F2 | F4 | | | |
| SUBD | F8 | F6 | F2 | | | |
| DIVD | F10 | F0 | F6 | | | |
| ADDD | F6 | F8 | F2 | | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | No | | | | | |
| | Mult2 | No | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | *FU* | | | | | | | | | |

# Tomasulo Example Cycle 1

*Instruction status:*

| Instruction | | j | k | Issue | Exec Comp | Write Result |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | | |
| LD | F2 | 45+ | R3 | | | |
| MULTD | F0 | F2 | F4 | | | |
| SUBD | F8 | F6 | F2 | | | |
| DIVD | F10 | F0 | F6 | | | |
| ADDD | F6 | F8 | F2 | | | |

| | Busy | Address |
|---|---|---|
| Load1 | Yes | 34+R2 |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| Time | Name | Busy | Op | S1 Vj | S2 Vk | RS Qj | RS Qk |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | No | | | | | |
| | Mult2 | No | | | | | |

*Register result status:*

| Clock | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | FU | | | | Load1 | | | | | |

# Tomasulo Example Cycle 2

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | | |
| LD | F2 | 45+ | R3 | 2 | | |
| MULTD | F0 | F2 | F4 | | | |
| SUBD | F8 | F6 | F2 | | | |
| DIVD | F10 | F0 | F6 | | | |
| ADDD | F6 | F8 | F2 | | | |

| | Busy | Address |
|---|---|---|
| Load1 | Yes | 34+R2 |
| Load2 | Yes | 45+R3 |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | No | | | | | |
| | Mult2 | No | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | *FU* | | Load2 | | Load1 | | | | | |

Note: Unlike 6600, can have multiple loads outstanding

# Tomasulo Example Cycle 3

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | |
| LD | F2 | 45+ | R3 | 2 | | |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | | | |
| DIVD | F10 | F0 | F6 | | | |
| ADDD | F6 | F8 | F2 | | | |

| | Busy | Address |
|---|---|---|
| Load1 | Yes | 34+R2 |
| Load2 | Yes | 45+R3 |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | Yes | MULTD | | R(F4) | Load2 | |
| | Mult2 | No | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | *FU* | Mult1 | Load2 | | Load1 | | | | | |

- Note: registers names are removed ("renamed") in Reservation Stations; MULT issued vs. scoreboard
- Load1 completing; what is waiting for Load1?

# Tomasulo Example Cycle 4

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | | |
| DIVD | F10 | F0 | F6 | | | |
| ADDD | F6 | F8 | F2 | | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | Yes | 45+R3 |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | Yes | SUBD | M(A1) | | | Load2 |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | Yes | MULTD | | R(F4) | Load2 | |
| | Mult2 | No | | | | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **4** | *FU* | Mult1 | Load2 | | M(A1) | Add1 | | | | |

- Load2 completing; what is waiting for Load2?

# Tomasulo Example Cycle 5

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | | |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| 2 | Add1 | Yes | SUBD | M(A1) | M(A2) | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| 10 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **5** | *FU* | Mult1 | M(A2) | | M(A1) | Add1 | Mult2 | | | |

# Tomasulo Example Cycle 6

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | | |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| 1 | Add1 | Yes | SUBD | M(A1) | M(A2) | | |
| | Add2 | Yes | ADDD | | M(A2) | Add1 | |
| | Add3 | No | | | | | |
| 9 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **6** | *FU* | Mult1 | M(A2) | | Add2 | Add1 | Mult2 | | | |

- Issue ADDD here vs. scoreboard?

# Tomasulo Example Cycle 7

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | 7 | |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| 0 | Add1 | Yes | SUBD | M(A1) | M(A2) | | |
| | Add2 | Yes | ADDD | | M(A2) | Add1 | |
| | Add3 | No | | | | | |
| 8 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 | *FU* | Mult1 | M(A2) | | Add2 | Add1 | Mult2 | | | |

- Add1 completing; what is waiting for it?

# Tomasulo Example Cycle 8

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| 2 | Add2 | Yes | ADDD | (M-M) | M(A2) | | |
| | Add3 | No | | | | | |
| 7 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **8** | *FU* | Mult1 | M(A2) | | Add2 | (M-M) | Mult2 | | | |

# Tomasulo Example Cycle 9

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| 1 | Add2 | Yes | ADDD | (M-M) | M(A2) | | |
| | Add3 | No | | | | | |
| 6 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **9** | *FU* | Mult1 | M(A2) | | Add2 | (M-M) | Mult2 | | | |

# Tomasulo Example Cycle 10

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | 10 | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| 0 | Add2 | Yes | ADDD | (M-M) | M(A2) | | |
| | Add3 | No | | | | | |
| 5 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **10** | *FU* | Mult1 | M(A2) | | Add2 | (M-M) | Mult2 | | | |

- Add2 completing; what is waiting for it?

# Tomasulo Example Cycle 11

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | 10 | 11 |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| 4 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **11** | *FU* | Mult1 | M(A2) | | (M-M+M | (M-M) | Mult2 | | | |

- Write result of ADDD here vs. scoreboard?
- All quick instructions complete in this cycle!

# Tomasulo Example Cycle 12

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | 10 | 11 |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| 3 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **12** | *FU* | Mult1 | M(A2) | | (M-M+M | (M-M) | Mult2 | | | |

# Tomasulo Example Cycle 13

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | 10 | 11 |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| 2 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **13** | *FU* | Mult1 | M(A2) | | (M-M+M | (M-M) | Mult2 | | | |

# Tomasulo Example Cycle 14

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | | |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | 10 | 11 |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| 1 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **14** | *FU* | Mult1 | M(A2) | | (M-M+M | (M-M) | Mult2 | | | |

# Tomasulo Example Cycle 15

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | 15 | |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | 10 | 11 |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| 0 | Mult1 | Yes | MULTD | M(A2) | R(F4) | | |
| | Mult2 | Yes | DIVD | | M(A1) | Mult1 | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **15** | *FU* | Mult1 | M(A2) | | (M-M+M | (M-M) | Mult2 | | | |

- Mult1 completing; what is waiting for it?

# Tomasulo Example Cycle 16

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | 15 | 16 |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | 10 | 11 |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | No | | | | | |
| 40 | Mult2 | Yes | DIVD | M*F4 | M(A1) | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **16** | *FU* | M*F4 | M(A2) | | (M-M+N | (M-M) | Mult2 | | | |

Faster than light computation
(skip a couple of cycles)

# Tomasulo Example Cycle 55

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | 15 | 16 |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | | |
| ADDD | F6 | F8 | F2 | 6 | 10 | 11 |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | No | | | | | |
| 1 | Mult2 | Yes | DIVD | M*F4 | M(A1) | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **55** | *FU* | M*F4 | M(A2) | | (M-M+M | (M-M) | Mult2 | | | |

# Tomasulo Example Cycle 56

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | 15 | 16 |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | 56 | |
| ADDD | F6 | F8 | F2 | 6 | 10 | 11 |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| *Time* | *Name* | *Busy* | *Op* | *S1 Vj* | *S2 Vk* | *RS Qj* | *RS Qk* |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | No | | | | | |
| 0 | Mult2 | Yes | DIVD | M*F4 | M(A1) | | |

*Register result status:*

| Clock | | *F0* | *F2* | *F4* | *F6* | *F8* | *F10* | *F12* | ... | *F30* |
|---|---|---|---|---|---|---|---|---|---|---|
| **56** | *FU* | M*F4 | M(A2) | | (M-M+N | (M-M) | Mult2 | | | |

- Mult2 is completing; what is waiting for it?

# Tomasulo Example Cycle 57

*Instruction status:*

| Instruction | | j | k | Issue | Exec Comp | Write Result |
|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 3 | 15 | 16 |
| SUBD | F8 | F6 | F2 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 5 | 56 | 57 |
| ADDD | F6 | F8 | F2 | 6 | 10 | 11 |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | No | |

*Reservation Stations:*

| Time | Name | Busy | Op | S1 Vj | S2 Vk | RS Qj | RS Qk |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | No | | | | | |
| 0 | Mult2 | Yes | DIVD | M*F4 | M(A1) | | |

*Register result status:*

| Clock | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 56 | FU | M*F4 | M(A2) | | (M-M+M | (M-M) | Mult2 | | | |

- Once again: In-order issue, out-of-order execution and completion.

# Compare to Scoreboard Cycle 62

*Instruction status:*

| Instruction | | *j* | *k* | *Issue* | *Read Oper* | *Exec Comp* | *Write Result* | *Issue* | *Exec Comp* | *Write Result* |
|---|---|---|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | 1 | 2 | 3 | 4 | 1 | 3 | 4 |
| LD | F2 | 45+ | R3 | 5 | 6 | 7 | 8 | 2 | 4 | 5 |
| MULTD | F0 | F2 | F4 | 6 | 9 | 19 | 20 | 3 | 15 | 16 |
| SUBD | F8 | F6 | F2 | 7 | 9 | 11 | 12 | 4 | 7 | 8 |
| DIVD | F10 | F0 | F6 | 8 | 21 | 61 | 62 | 5 | 56 | 57 |
| ADDD | F6 | F8 | F2 | 13 | 14 | 16 | 22 | 6 | 10 | 11 |

- 为什么scoreboard/6600所需时间较长?
  - 结构冲突
  - WAR,WAW冲突
  - 没有定向技术

# Reservation Station 结构

Op: 部件所进行的操作

Vj, Vk: 源操作数的值。Store 缓冲区有Vk域，用于存放要写入存储器的值

A: 存放存储器地址。开始存立即数，计算出有效地址后，存放有效地址

Qj, Qk: 产生源操作数的RS

注：没有记分牌中的准备就绪标志， Qj, Qk=0 => ready

Store 缓存区中Qk表示产生结果的RS

Busy: 标识RS或FU是否空闲

Register result status—如果存在对寄存器的写操作，指示对该寄存器进行写操作的部件.

Qi: 保留站的编号

# Tomasulo 算法的三阶段

1. Issue—从FP操作队列中取指令

   如果RS空闲(no structural hazard), 则控制发射指令和操作数(renames registers). 消除WAR，WAW相关

2. Execution—operate on operands (EX)

   当两操作数就绪后，就可以执行
   如果没有准备好，则监测Common Data Bus 以获取结果。通过推迟指令执行避免RAW相关

3. Write result—finish execution (WB)

   将结果通过Common Data Bus传给所有等待该结果的部件;
   表示RS可用

- 通常的数据总线: data + destination (“go to” bus)
- Common data bus: data + source (“come from” bus)
  - 64 bits 数据线 + 4 bits 功能部件源地址（ FU source address）
  - 产生结果的部件如果与RS中等待的部件匹配，就进行写操作
  - 广播方式传送

# Tomasulo 算法流水线控制

rs, rt : 源寄存器名；rd:目的寄存器名
RS: 保留站数据结构；r:保留站编号
RegisterStat：寄存器结果状态表
Reg： 寄存器组

1. Issue

FP Operation:

Wait until： Station r empty

Action or bookkeeping:

1st 操作数

if(RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi}

2nd 操作数

else {RS[r].Vj ←Reg[rs]; RS[r].Qj ←0 }

if(RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi}

else {RS[r].Vk ←Reg[rt]; RS[r].Qk←0 }

RS[r].Busy ← yes; RegisterStat[rd].Qi = r;

Load or Store:

Wait until: Buffer r empty

Action or bookkeeping:

基址寄存器

if(RegisterStat[rs].Qi≠0)

{RS[r].Qj ← RegisterStat[rs].Qi}

else {RS[r].Vj ←Reg[rs]; RS[r].Qj ←0 }

RS[r].A ←imm; RS[r].Busy ← yes;

Load only: RegisterStat[rt].Qi = r;

Store only:

需写入的
数据寄存器

if(RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi}

else {RS[r].Vk ←Reg[rt]; RS[r].Qk←0 }

# 注意：Load操作在EXE阶段分两步

2、Execute

FP Operation

wait until: (RS[r].Qj=0) and (RS[r].Qk=0)

Action or bookkeeping:

computer result: Operands are in Vj and Vk

Load-store step1

wait until: RS[r].Qj =0 & r is head of load-store queue

Action or bookkeeping:

RS[r].A ← RS[r].Vj + RS[r].A;

Load step2

wait until: Load Step1 complete

Action or bookkeeping:

Read from Mem[RS[r].A]

3、Write result

FP Operation or Load

Wait until: Execution complete at r & CDB available

Action or bookkeeping

∀x (if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0})

∀ x (if(RS[x].Qj =r) {RS[x].Vj ←result; RS[x].Qj ←0});

∀ x (if(RS[x].Qk =r) {RS[x].Vk ←result; RS[x].Qk ←0});

RS[r].Busy ← no;

Store

wait until:

Execution complete at r & RS[r].Qk = 0

Action or bookkeeping

Mem[RS[r].A] ← RS[r].Vk;

RS[r].Busy ← no;

# Tomasulo v. Scoreboard (IBM 360/91 v. CDC 6600)

| | |
|---|---|
| 流水化的功能部件 | 多个功能部件 |
| (6 load, 3 store, 3 +, 2 x/ ÷) | (1 load/store, 1 + , 2 x, 1 ÷) |
| 指令窗口大小: ~ 14 instructions | ~ 5 instructions |
| 有结构冲突时不发射 | 相同 |
| WAR: 用寄存器重命名避免 | stall 来避免 |
| WAW:用寄存器重命名避免 | 停止发射 |
| 从FU广播结果写寄存器方式 | |
| Control: RS集中式scoreboard | |

# -Tomasulo 算法的特点

- **控制和缓存分布在各部件中**
  - FU 缓存称“reservation stations”; 保存待用操作数
- **指令中的寄存器在RS中用寄存器值或指向RS的指针代替（称为 register renaming）**
  - 避免 WAR, WAW hazards
  - RS多于寄存器，因此可以做更多编译器无法做的优化
- **传给FU的结果从RS来而不是从寄存器来，FU的计算结果通过Common Data Bus 以广播方式发向所有功能部件**
- **Load和Store部件也看作带有RS的功能部件**
- **可以跨越分支，允许FP操作队列中FP操作不仅仅局限于基本块**

# Tomasulo 缺陷

- **复杂**
  - Large amount of hardware, high-speed buffers
- **要求高速CDB**
  - 性能受限于Common Data Bus
- **在IBM 360/91之后多年未用，90之后在multi-issue processors中广泛使用**

教材：Ch. 3.4-3.5

# Recap：Tomasulo 算法流水线控制

rs, rt：源寄存器名；rd:目的寄存器名
RS: 保留站数据结构；r:保留站编号
RegisterStat：寄存器结果状态表
Reg： 寄存器组

1. Issue

FP Operation:

Wait until： Station r empty

Action or bookkeeping:

1st 操作数

if(RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi}

2nd 操作数

else {RS[r].Vj ←Reg[rs]; RS[r].Qj ←0 }

if(RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi}

else {RS[r].Vk ←Reg[rt]; RS[r].Qk←0 }

RS[r].Busy ← yes; RegisterStat[rd].Qi = r;

Load or Store:

Wait until: Buffer r empty

Action or bookkeeping:

基址寄存器

if(RegisterStat[rs].Qi≠0)

{RS[r].Qj ← RegisterStat[rs].Qi}

else {RS[r].Vj ←Reg[rs]; RS[r].Qj ←0 }

RS[r].A ←imm; RS[r].Busy ← yes;

Load only: RegisterStat[rt].Qi = r;

Store only:

需写入的
数据寄存器

if(RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi}

else {RS[r].Vk ←Reg[rt]; RS[r].Qk←0 }

# 注意：Load操作在EXE阶段分两步

2、Execute

FP Operation

wait until: (RS[r].Qj=0) and (RS[r].Qk=0)

both操作数ready

Action or bookkeeping:

computer result: Operands are in Vj and Vk

Load-store step1

基址寄存器ready，load排队第一个

wait until: RS[r].Qj =0 & r is head of load-store queue

RS[r].A;

计算存储器地址

Load step2Action or bookkeeping:

RS[r].A ← RS[r].Vj +

wait until: Load Step1 complete

Action or bookkeeping:

Read from Mem[RS[r].A]

读取存储器

3、Write result

FP Operation or Load

Wait until: Execution complete at r & CDB available

Action or bookkeeping

写回寄存器

∀x (if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0})

∀ x (if(RS[x].Qj =r) {RS[x].Vj ←result; RS[x].Qj ←0});

Forward到等待的第一个操作数

∀ x (if(RS[x].Qk =r) {RS[x].Vk ←result; RS[x].Qk ←0});

RS[r].Busy ← no;

Forward到等待的第二个操作数

Store

wait until:

Execution complete at r & RS[r].Qk = 0

Action or bookkeeping

存储寄存器ready

Mem[RS[r].A] ← RS[r].Vk;

写存储器

RS[r].Busy ← no;

# Tomasulo Loop Example

| | | |
|---|---|---|
| **Loop:** | **LD** | **F0, 0(R1)** |
| | MULTD | F4, F0, F2 |
| | SD | F4, 0(R1) |
| | SUBI | R1,R1,#8 |
| | BNEZ | R1 Loop |

- **设Multiply执行阶段4 clocks**
- **第一次load 需8 clocks (cache miss), 第2次以后假设命中(hit)**
- **为清楚起见，下面我们也列出SUBI, BNEZ的时钟周期**

# Loop Example

| Instruction Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| 1 | LD | F0 | 0 | R1 | | | | Load1 | No | | |
| 1 | MULTD | F4 | F0 | F2 | | | | Load2 | No | | |
| 1 | SD | F4 | 0 | R1 | | | | Load3 | No | | |
| 2 | LD | F0 | 0 | R1 | | | | Store1 | No | | |
| 2 | MULTD | F4 | F0 | F2 | | | | Store2 | No | | |
| 2 | SD | F4 | 0 | R1 | | | | Store3 | No | | |

| Reservation Station: | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | No | | | | | | SUBI | R1 | R1 | #8 |
| | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

| Register Result Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| 0 | 80 | FU | | | | | | | | | |

# Loop Example Cycle 1

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR |
|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | | |
| 1 | MULTD | F4 | F0 | F2 | | | |
| 1 | SD | F4 | 0 | R1 | | | |
| 2 | LD | F0 | 0 | R1 | | | |
| 2 | MULTD | F4 | F0 | F2 | | | |
| 2 | SD | F4 | 0 | R1 | | | |

| | Busy | Addr | Fu |
|---|---|---|---|
| Load1 | Yes | 80 | |
| Load2 | No | | |
| Load3 | No | | |
| Store1 | No | | |
| Store2 | No | | |
| Store3 | No | | |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | No | | | | | |
| | Mult2 | No | | | | | |

| Code | | | |
|---|---|---|---|
| LD | F0 | 0 | R1 |
| MULTD | F4 | F0 | F2 |
| SD | F4 | 0 | R1 |
| SUBI | R1 | R1 | #8 |
| BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 80 | FU | Load1 | | | | | | | | |

# Loop Example Cycle 2

| Instruction Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| 1 | LD | F0 | 0 | R1 | 1 | 2~ | | Load1 | Yes | 80 | |
| 1 | MULTD | F4 | F0 | F2 | 2 | | | Load2 | No | | |
| 1 | SD | F4 | 0 | R1 | | | | Load3 | No | | |
| 2 | LD | F0 | 0 | R1 | | | | Store1 | No | | |
| 2 | MULTD | F4 | F0 | F2 | | | | Store2 | No | | |
| 2 | SD | F4 | 0 | R1 | | | | Store3 | No | | |

| Reservation Station: | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | YES | Multd | | R(F2) | Load1 | | SUBI | R1 | R1 | #8 |
| | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

| Register Result Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| 2 | 80 | FU | Load1 | | Mult1 | | | | | | |

# Loop Example Cycle 3

## Instruction Status

| ITER | Inst. | i | j | k | Issue | Exec | WR |
|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~ | |
| 1 | MULTD | F4 | F0 | F2 | 2 | | |
| 1 | SD | F4 | 0 | R1 | 3 | | |
| 2 | LD | F0 | 0 | R1 | | | |
| 2 | MULTD | F4 | F0 | F2 | | | |
| 2 | SD | F4 | 0 | R1 | | | |

| | Busy | Addr | Fu |
|---|---|---|---|
| Load1 | Yes | 80 | |
| Load2 | No | | |
| Load3 | No | | |
| Store1 | YES | 80 | Mult1 |
| Store2 | No | | |
| Store3 | No | | |

## Reservation Station:

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | YES | Multd | | R(F2) | Load1 | | SUBI | R1 | R1 | #8 |
| | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

## Register Result Status

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 80 | FU | Load1 | | Mult1 | | | | | | |

# What does this mean physically?

# Loop Example Cycle 4

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~ | | Load1 | Yes | 80 | |
| 1 | MULTD | F4 | F0 | F2 | 2 | | | Load2 | No | | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | No | | |
| 2 | LD | F0 | 0 | R1 | | | | Store1 | YES | 80 | Mult1 |
| 2 | MULTD | F4 | F0 | F2 | | | | Store2 | No | | |
| 2 | SD | F4 | 0 | R1 | | | | Store3 | No | | |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | YES | Multd | | R(F2) | Load1 | | SUBI | R1 | R1 | #8 |
| | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 80 | FU | Load1 | | Mult1 | | | | | | |

- Dispatching SUBI Instruction

# Loop Example Cycle 5

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~ | | Load1 | Yes | 80 | |
| 1 | MULTD | F4 | F0 | F2 | 2 | | | Load2 | No | | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | No | | |
| 2 | LD | F0 | 0 | R1 | | | | Store1 | YES | 80 | Mult1 |
| 2 | MULTD | F4 | F0 | F2 | | | | Store2 | No | | |
| 2 | SD | F4 | 0 | R1 | | | | Store3 | No | | |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | YES | Multd | | R(F2) | Load1 | | SUBI | R1 | R1 | #8 |
| | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 80 | FU | Load1 | | Mult1 | | | | | | |

- And, BNEZ instruction

# Loop Example Cycle 6

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~ | | Load1 | Yes | 80 | |
| 1 | MULTD | F4 | F0 | F2 | 2 | | | Load2 | Yes | 72 | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | No | | |
| 2 | LD | F0 | 0 | R1 | 6 | | | Store1 | YES | 80 | Mult1 |
| 2 | MULTD | F4 | F0 | F2 | | | | Store2 | No | | |
| 2 | SD | F4 | 0 | R1 | | | | Store3 | No | | |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | YES | Multd | | R(F2) | Load1 | | SUBI | R1 | R1 | #8 |
| | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 72 | FU | Load2 | | Mult1 | | | | | | |

- 注意：F0 不是从80地址处装载的值; 如何消除WAW?

# Loop Example Cycle 7

| Instruction Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| 1 | LD | F0 | 0 | R1 | 1 | 2~ | | Load1 | Yes | 80 | |
| 1 | MULTD | F4 | F0 | F2 | 2 | | | Load2 | Yes | 72 | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | No | | |
| 2 | LD | F0 | 0 | R1 | 6 | | | Store1 | YES | 80 | Mult1 |
| 2 | MULTD | F4 | F0 | F2 | 7 | | | Store2 | No | | |
| 2 | SD | F4 | 0 | R1 | | | | Store3 | No | | |

| Reservation Station: | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | Yes | Multd | | R(F2) | Load1 | | SUBI | R1 | R1 | #8 |
| | Mult2 | Yes | Multd | | R(F2) | Load2 | | BNEZ | R1 | Loop | |

| Register Result Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| 7 | 72 | FU | Load2 | | Mult2 | | | | | | |

- 对寄存器文件的操作都是第2次循环的指令。如何消除WAW?

# Loop Example Cycle 8

| Instruction Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| 1 | LD | F0 | 0 | R1 | 1 | 2~ | | Load1 | Yes | 80 | |
| 1 | MULTD | F4 | F0 | F2 | 2 | | | Load2 | Yes | 72 | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | No | | |
| 2 | LD | F0 | 0 | R1 | 6 | | | Store1 | YES | 80 | Mult1 |
| 2 | MULTD | F4 | F0 | F2 | 7 | | | Store2 | Yes | 72 | Mult2 |
| 2 | SD | F4 | 0 | R1 | 8 | | | Store3 | No | | |

| Reservation Station: | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | Yes | Multd | | R(F2) | Load1 | | SUBI | R1 | R1 | #8 |
| | Mult2 | Yes | Multd | | R(F2) | Load2 | | BNEZ | R1 | Loop | |

| Register Result Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| 8 | 72 | FU | Load2 | | Mult2 | | | | | | |

- 第1次循环与第2次循环重叠执行

# What does this mean physically?

# Loop Example Cycle 9

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~9 | | Load1 | Yes | 80 | |
| 1 | MULTD | F4 | F0 | F2 | 2 | | | Load2 | Yes | 72 | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | No | | |
| 2 | LD | F0 | 0 | R1 | 6 | | | Store1 | YES | 80 | Mult1 |
| 2 | MULTD | F4 | F0 | F2 | 7 | | | Store2 | Yes | 72 | Mult2 |
| 2 | SD | F4 | 0 | R1 | 8 | | | Store3 | No | | |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | Yes | Multd | | R(F2) | Load1 | | SUBI | R1 | R1 | #8 |
| | Mult2 | Yes | Multd | | R(F2) | Load2 | | BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 72 | FU | Load2 | | Mult2 | | | | | | |

- Dispatching SUBI，Load1执行完毕

**Load2 是否可以进入执行阶段？**
**Store2 是否可以进入执行阶段？**

# Loop Example Cycle 10

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| 1 | MULTD | F4 | F0 | F2 | 2 | | | Load2 | Yes | 72 | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | No | | |
| 2 | LD | F0 | 0 | R1 | 6 | | | Store1 | YES | 80 | Mult1 |
| 2 | MULTD | F4 | F0 | F2 | 7 | | | Store2 | Yes | 72 | Mult2 |
| 2 | SD | F4 | 0 | R1 | 8 | | | Store3 | No | | |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| 4 | Mult1 | Yes | Multd | M[80] | R(F2) | | | SUBI | R1 | R1 | #8 |
| | Mult2 | Yes | Multd | | R(F2) | Load2 | | BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 64 | FU | Load2 | | Mult2 | | | | | | |

- Dispatching BNEZ，Load1写结果

# Loop Example Cycle 11

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| 1 | MULTD | F4 | F0 | F2 | 2 | 11~ | | Load2 | Yes | 72 | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | Yes | 64 | |
| 2 | LD | F0 | 0 | R1 | 6 | 11 | | Store1 | YES | 80 | Mult1 |
| 2 | MULTD | F4 | F0 | F2 | 7 | | | Store2 | Yes | 72 | Mult2 |
| 2 | SD | F4 | 0 | R1 | 8 | | | Store3 | No | | |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| 3 | Mult1 | Yes | Multd | M[80] | R(F2) | | | SUBI | R1 | R1 | #8 |
| | Mult2 | Yes | Multd | | R(F2) | Load2 | | BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 64 | FU | Load3 | | Mult2 | | | | | | |

- Load3发射，F0 由第3次循环的Load装载地址为64单元的内容；解除WAW冲突

# Loop Example Cycle 12

| Instruction Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| 1 | MULTD | F4 | F0 | F2 | 2 | 11~ | | Load2 | No | | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | Yes | 64 | |
| 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | YES | 80 | Mult1 |
| 2 | MULTD | F4 | F0 | F2 | 7 | | | Store2 | Yes | 72 | Mult2 |
| 2 | SD | F4 | 0 | R1 | 8 | | | Store3 | No | | |

| Reservation Station: | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| 2 | Mult1 | Yes | Multd | M[80] | R(F2) | | | SUBI | R1 | R1 | #8 |
| 4 | Mult2 | Yes | Multd | M[72] | R(F2) | | | BNEZ | R1 | Loop | |

| Register Result Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| 12 | 64 | FU | Load3 | | Mult2 | | | | | | |

- Load2写结果
- Why not issue third multiply?

# Loop Example Cycle 13

| Instruction Status | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| | 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| | 1 | MULTD | F4 | F0 | F2 | 2 | 11~ | | Load2 | No | | |
| | 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | Yes | 64 | |
| | 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | YES | 80 | Mult1 |
| | 2 | MULTD | F4 | F0 | F2 | 7 | 13~ | | Store2 | Yes | 72 | Mult2 |
| | 2 | SD | F4 | 0 | R1 | 8 | | | Store3 | No | | |
| **Reservation Station:** | | | | | | | | | | | | |
| | Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | 1 | Mult1 | Yes | Multd | M[80] | R(F2) | | | SUBI | R1 | R1 | #8 |
| | 3 | Mult2 | Yes | Multd | M[72] | R(F2) | | | BNEZ | R1 | Loop | |
| | | | | | | | | | | | | |
| **Register Result Status** | | | | | | | | | | | | |
| | Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| | 13 | 64 | FU | Load3 | | Mult2 | | | | | | |

# Loop Example Cycle 14

## Instruction Status

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | | Load2 | No | | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | Yes | 64 | |
| 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | YES | 80 | Mult1 |
| 2 | MULTD | F4 | F0 | F2 | 7 | 13~ | | Store2 | Yes | 72 | Mult2 |
| 2 | SD | F4 | 0 | R1 | 8 | | | Store3 | No | | |

## Reservation Station:

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| 0 | Mult1 | Yes | Multd | M[80] | R(F2) | | | SUBI | R1 | R1 | #8 |
| 2 | Mult2 | Yes | Multd | M[72] | R(F2) | | | BNEZ | R1 | Loop | |

## Register Result Status

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 14 | 64 | FU | Load3 | | Mult2 | | | | | | |

Mult1执行完毕

# Loop Example Cycle 15

**Instruction Status**

| | ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| | 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| | 1 | SD | F4 | 0 | R1 | 3 | 4 | | Load3 | Yes | 64 | |
| | 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | Yes | 80 | [80]*R2 |
| | 2 | MULTD | F4 | F0 | F2 | 7 | 13~ | | Store2 | Yes | 72 | Mult2 |
| | 2 | SD | F4 | 0 | R1 | 8 | | | Store3 | No | | |

**Reservation Station:**

| | Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | | Mult1 | No | | | | | | SUBI | R1 | R1 | #8 |
| | 1 | Mult2 | Yes | Multd | M[72] | R(F2) | | | BNEZ | R1 | Loop | |

**Register Result Status**

| | Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 15 | 64 | FU | Load3 | | Mult2 | | | | | | |

- Mult1写结果

# Loop Example Cycle 16

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | 16 | Load3 | Yes | 64 | |
| 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | No | | |
| 2 | MULTD | F4 | F0 | F2 | 7 | 13~16 | | Store2 | Yes | 72 | Mult2 |
| 2 | SD | F4 | 0 | R1 | 8 | | | Store3 | No | | |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | Yes | Multd | | R(F2) | Load3 | | SUBI | R1 | R1 | #8 |
| 0 | Mult2 | Yes | Multd | M[72] | R(F2) | | | BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 64 | FU | Load3 | | Mult1 | | | | | | |

- Mult2执行完毕，SD1写结果，发射Mult3

# Loop Example Cycle 17

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | 16 | Load3 | Yes | 64 | |
| 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | No | | |
| 2 | MULTD | F4 | F0 | F2 | 7 | 13~16 | 17 | Store2 | Yes | 72 | [72]*R2 |
| 2 | SD | F4 | 0 | R1 | 8 | 17 | | Store3 | Yes | 64 | Mult1 |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | Yes | Multd | | R(F2) | Load3 | | SUBI | R1 | R1 | #8 |
| | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 17 | 64 | FU | Load3 | | Mult1 | | | | | | |

- Mult2写结果，SD2进入执行阶段，发射SD3

# Loop Example Cycle 18

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | 16 | Load3 | Yes | 64 | |
| 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | No | | |
| 2 | MULTD | F4 | F0 | F2 | 7 | 13~16 | 17 | Store2 | No | | |
| 2 | SD | F4 | 0 | R1 | 8 | 17 | 18 | Store3 | Yes | 64 | Mult1 |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | Mult1 | Yes | Multd | | R(F2) | Load3 | | SUBI | R1 | R1 | #8 |
| | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 18 | 64 | FU | Load3 | | Mult1 | | | | | | |

- SD2进入写结果阶段，LD3进入执行阶段，SD3进入执行阶段 （假设地址计算部件多个）

# Loop Example Cycle 19

**Instruction Status**

| | ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| | 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| | 1 | SD | F4 | 0 | R1 | 3 | 4 | 16 | Load3 | No | | |
| | 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | No | | |
| | 2 | MULTD | F4 | F0 | F2 | 7 | 13~16 | 17 | Store2 | No | | |
| | 2 | SD | F4 | 0 | R1 | 8 | 17 | 18 | Store3 | Yes | 64 | Mult1 |

**Reservation Station:**

| | Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | 4 | Mult1 | Yes | Multd | M[64] | R(F2) | | | SUBI | R1 | R1 | #8 |
| | | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

**Register Result Status**

| | Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 19 | 64 | FU | | | Mult1 | | | | | | |

- LD3写结果

# Loop Example Cycle 20

| Instruction Status | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| | 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| | 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| | 1 | SD | F4 | 0 | R1 | 3 | 4 | 16 | Load3 | No | | |
| | 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | No | | |
| | 2 | MULTD | F4 | F0 | F2 | 7 | 13~16 | 17 | Store2 | No | | |
| | 2 | SD | F4 | 0 | R1 | 8 | 17 | 18 | Store3 | Yes | 64 | Mult1 |

| Reservation Station: | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | 3 | Mult1 | Yes | Multd | M[64] | R(F2) | | | SUBI | R1 | R1 | #8 |
| | | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

| Register Result Status | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| | 20 | 64 | FU | | | Mult1 | | | | | | |

- Mult3 20~23

# Loop Example Cycle 21

| Instruction Status | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| | 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| | 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| | 1 | SD | F4 | 0 | R1 | 3 | 4 | 16 | Load3 | No | | |
| | 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | No | | |
| | 2 | MULTD | F4 | F0 | F2 | 7 | 13~16 | 17 | Store2 | No | | |
| | 2 | SD | F4 | 0 | R1 | 8 | 17 | 18 | Store3 | Yes | 64 | Mult1 |

| Reservation Station: | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | 2 | Mult1 | Yes | Multd | M[64] | R(F2) | | | SUBI | R1 | R1 | #8 |
| | | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

| Register Result Status | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| | 21 | 64 | FU | Load3 | | Mult1 | | | | | | |

- Mult3 20~23

# Loop Example Cycle 22

| Instruction Status | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| | 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| | 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| | 1 | SD | F4 | 0 | R1 | 3 | 4 | 16 | Load3 | No | | |
| | 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | No | | |
| | 2 | MULTD | F4 | F0 | F2 | 7 | 13~16 | 17 | Store2 | No | | |
| | 2 | SD | F4 | 0 | R1 | 8 | 17 | 18 | Store3 | Yes | 64 | Mult1 |

| Reservation Station: | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | 1 | Mult1 | Yes | Multd | M[64] | R(F2) | | | SUBI | R1 | R1 | #8 |
| | | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

| Register Result Status | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| | 22 | 64 | FU | Load3 | | Mult1 | | | | | | |

- Mult3 20~23

# Loop Example Cycle 23

**Instruction Status**

| ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| 1 | SD | F4 | 0 | R1 | 3 | 4 | 16 | Load3 | No | | |
| 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | No | | |
| 2 | MULTD | F4 | F0 | F2 | 7 | 13~16 | 17 | Store2 | No | | |
| 2 | SD | F4 | 0 | R1 | 8 | 17 | 18 | Store3 | Yes | 64 | Mult1 |

**Reservation Station:**

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| 0 | Mult1 | Yes | Multd | M[64] | R(F2) | | | SUBI | R1 | R1 | #8 |
| | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

**Register Result Status**

| Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 64 | FU | Load3 | | Mult1 | | | | | | |

- Mult3 20~23,Mult3执行完毕

# Loop Example Cycle 24

| Instruction Status | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| | 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| | 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| | 1 | SD | F4 | 0 | R1 | 3 | 4 | 16 | Load3 | No | | |
| | 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | No | | |
| | 2 | MULTD | F4 | F0 | F2 | 7 | 13~16 | 17 | Store2 | No | | |
| | 2 | SD | F4 | 0 | R1 | 8 | 17 | 18 | Store3 | Yes | 64 | [64]*R2 |

| Reservation Station: | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | | Mult1 | No | | | | | | SUBI | R1 | R1 | #8 |
| | | Mult2 | No | | | | | | BNEZ | R1 | Loop | |

| Register Result Status | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| | 24 | 64 | FU | | | [64]*R2 | | | | | | |

- Mult3 20~23,Mult3写结果

# Loop Example Cycle 25

| Instruction Status | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ITER | Inst. | i | j | k | Issue | Exec | WR | | Busy | Addr | Fu |
| | 1 | LD | F0 | 0 | R1 | 1 | 2~9 | 10 | Load1 | No | | |
| | 1 | MULTD | F4 | F0 | F2 | 2 | 11~14 | 15 | Load2 | No | | |
| | 1 | SD | F4 | 0 | R1 | 3 | 4 | 16 | Load3 | No | | |
| | 2 | LD | F0 | 0 | R1 | 6 | 11 | 12 | Store1 | No | | |
| | 2 | MULTD | F4 | F0 | F2 | 7 | 13~16 | 17 | Store2 | No | | |
| | 2 | SD | F4 | 0 | R1 | 8 | 17 | 18 | Store3 | No | | |
| **Reservation Station:** | | | | | | | | | | | | |
| | Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Code | | | |
| | | Add1 | No | | | | | | LD | F0 | 0 | R1 |
| | | Add2 | No | | | | | | MULTD | F4 | F0 | F2 |
| | | Add3 | No | | | | | | SD | F4 | 0 | R1 |
| | | Mult1 | No | | | | | | SUBI | R1 | R1 | #8 |
| | | Mult2 | No | | | | | | BNEZ | R1 | Loop | |
| **Register Result Status** | | | | | | | | | | | | |
| | Clock | R1 | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ..... | F30 |
| | 25 | 64 | FU | Load3 | | Mult2 | | | | | | |

- SD3写结果

# review

- **Tomasulo Algorithm 三阶段**

  1. **Issue—从FP操作队列中取指令**
     - 如果RS空闲(no structural hazard), 则控制发射指令和操作数(renames registers).
  2. **Execution—operate on operands (EX)**
     - 当两操作数就绪后，就可以执行
       如果没有准备好，则监测Common Data Bus 以获取结果
  3. **Write result—finish execution (WB)**
     - 将结果通过Common Data Bus传给所有等待该结果的部件;
       表示RS可用

- **基本数据结构**

  1. **Instruction Status**
  2. **Reservation Station**
  3. **Register Result Status**

# 04/17-review

- **Reservations stations: 寄存器重命名，缓冲源操作数**
  - 避免寄存器成为瓶颈
  - 避免了Scoreboard中无法解决的 WAR, WAW hazards
  - 允许硬件做循环展开
- **不限于基本块(IU先行，解决控制相关)**
- **贡献**
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- **360/91 后 Pentium II; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264使用这种技术**

# 04/17- review：Tomasulo算法实现循环覆盖执行？

- **寄存器重命名技术**
  - 不同的循环使用不同的物理寄存器 (dynamic loop unrolling).
  - 将代码中的静态寄存器名修改为动态寄存器指针 “pointers”
  - 有效地增加了寄存器文件的大小
- **关键: 整数部件必须先行，以便能发射多个循环中的操作**

# ?

- **动态硬件方案可以用硬件进行循环展开**
- **如何处理精确中断?**
  - Out-of-order execution -> out-of-order completion!
- **如何处理分支?**
  - 我们可以用硬件做循环展开必须可以解决分支指令问题

# 关于异常处理???

- **乱序完成加大了实现精确中断的难度**
  - 在前面指令还没有完成时，寄存器文件中可能会有后面指令的运行结果.
  - 如果这些前面的指令执行时有中断产生，怎么办？
  - 例如：DIVD F10, F0, F2
    - SUBD F4, F6, F8
    - ADDD F12, F14, F16
- **需要“rollback” 寄存器文件到原来的状态:**
  - 精确中断的含义是其返回地址为：
    - 该地址之前的所有指令都已完成
    - 其后的指令还都没有完成
- **实现精确中断的技术：顺序完成（或提交）**
  - 即提交指令完成的顺序必须与指令发射的顺序相同

# 进行循环重叠执行需要尽快解决分支问题!

- **在循环展开的例子中，我们假设整数部件可以快速解决分支问题，以便进行循环重叠执行!**

```
Loop:  LD      F0   0    R1
       MULTD   F4   F0   F2
       SD      F4   0    R1
       SUBI    R1   R1   #8
       BNEZ    R1   Loop
```

- **如果分支依赖于multd,怎么办??**
  - 需要能预测分支方向
  - 如果分支成功，我们就可以重叠执行循环
- **对于superscalar机器这一问题更加突出**

# 控制相关的动态解决技术

- **控制相关：**
  - 由条件转移或程序中断引起的相关，也称全局相关。
  - 控制相关对流水线的吞吐率和效率影响相对于数据相关要大得多
    - 条件指令在一般程序中所占的比例相当大
    - 中断虽然在程序中所占的比例不大，但中断发生在程序中的哪条指令，发生在一条指令执行过程中的哪个功能段都是不确定的
- **处理条件转移和中断引起的控制相关的关键问题：**
  - 要确保流水线能够正常工作
  - 减少因断流引起的吞吐率和效率的下降

# 分支对性能的影响

- **假设在一条有K段的流水线中，在最后一段才能确定目标地址**

- **当分支方向预测错误时**
  - 流水线中有多个功能段要浪费
  - 可能造成程序执行结果发生错误
  - 因此当程序沿着错误方向运行后，作废这些程序时，一定不能破坏通用寄存器和主存储器的内容。

# 条件转移指令对流水线性能的影响

- **假设对于一条有K段的流水线，由于条件分支的影响，在最坏情况下，每次条件转移将造成k-1个时钟周期的断流。假设条件分支在一般程序中所占的比例为p, 采用分支预测失败策略，条件成功的概率为q。试分析分支对流水线的影响。**
- **结论：条件转移指令对流水线的影响很大，必须采取相关措施来减少这种影响。**
- **预测可以是静态预测“Static” (at compile time) 或动态预测 “Dynamic” (at runtime)**
  - 例如：一个循环供循环10次，它将分支成功9次，1次不成功。
  - 动态分支预测 vs. 静态分支预测，哪个好？

# 分支预测

- **分支预测对提高性能是非常重要的**
  - 分支预测在哪个阶段完成？
  - 预测器设计的核心问题是什么？
  - 预测器的基本结构及输入输出？
- **预测器的分类**
  - 基于BHT表的预测器:
    - Basic 2-bit predictor:
    - Correlating predictor:
      - Multiple 2-bit predictors for each branch
      - One for each possible combination of outcomes of preceding n branches
    - Local predictor:
      - Multiple 2-bit predictors for each branch
      - One for each possible combination of outcomes for the last n occurrences of this branch
    - Tournament predictor: Combine correlating predictor with local predictor
  - 基于BTB的分支预测器

CH. 3.3

# Instruction Fetch Unit

# 预测器的基本结构及输入输出

- **根据outcomes来更新状态信息**
- **使用FSM(finite state machine)来完成决策**
  - 根据Branch History和PC来选择状态
  - 由状态决定输出

# Dynamic Branch Prediction

- **动态分支预测：预测分支的方向在程序运行时刻动态确定**
- **需解决的关键问题是：**
  - 如何记录转移历史信息
  - 如何根据所记录的转移历史信息，预测转移的方向
- **主要方法**
  - 基于BPB(Branch Prediction Buffer)或BHT(Branch History Table)
    - 1-bit BHT和2-bit BHT
    - Correlating Branch Predictors
    - Tournament Predictors: Adaptively Combining Local and Global Predictors
  - High Performance Instruction Delivery
    - BTB
    - Integrated Instruction Fetch Units
    - Return Address Predictors
- **Performance = *f*(accuracy, cost of misprediction)**
  - Misprediction □ Flush Reorder Buffer

# 1-bit BHT

- **Branch History Table:**
  - 分支指令的PC的低位索引
  - 该表记录上一次转移是否成功
  - 不做地址检查
  - 1-bit BHT

- **问题: 在一个循环中, 1-bit BHT 将导致2次分支预测错误**
  - 假设一循环次数为10次的简单程序段
  - 最后一次循环
    - 前面为预测成功，最后一次需要退出循环
  - 首次循环
    - 前面为预测为失败，这次实际上为成功

# 2-bit BHT

- **解决办法: 2位记录分支历史**
- **Blue: stop, not taken**
- **Green: go, taken**

**FIGURE 3.8 Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks.** The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the FP programs (average of 4%). Even omitting the FP kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch prediction study done using the IBM Power architecture and optimized code for that system. See Pan et al. [1992].

**FIGURE 3.9 Prediction accuracy of a 4096-entry two-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks.**

# BHT Accuracy

- **分支预测错误的原因:**
  - 预测错误
  - 由于使用PC的低位查找BHT表，可能得到错误的分支历史记录
- **BHT表的大小问题**
  - 4096 项的表分支预测错误的比例为1%（nasa7, tomcatv) to 18% (eqntott)，spice at 9% and gcc at 12%
  - 再增加项数，对提高预测准确率几乎没有效果 (in Alpha 21164)

# Correlating Branch Predicator

- 例如：

```
if (aa==2)   aa=0;
if (bb==2)  bb=0;
if (aa!=bb) {
```

- 翻译为MIPS

```
      SUBI R3,R1,#2
      BNEZ R3,L1          ; branch b1 (aa!=2)
      ADDI  R1,R0,R0      ;aa=0
L1:   SUBI R3,R2,#2
      BNEZ R3,L2          ;branch b2(bb!=2)
      ADDI R2,R0,R0       ; bb=0
L2:   SUBI R3,R1,R2       ;R3=aa-bb
      BEQZ R3,L3          ;branch b3 (aa==bb)
```

- 观察结果：

b3 与分支b2 和b1相关。
如果b1和b2都分支失败，则b3一定成功。

# Correlating Branches

- **Correlating predictors 或 两级预测器：**
  - 分支预测器根据其他分支的行为来进行预测
- **工作原理：**
  - 根据一个简单的例子来看其基本原理

翻译为MIPS

```
 if (d==0)d=1;
 if (d==1) d=0;

      BNEZ R1,L1        ;branch b1(d!=0)
      ADDI R1,R0,#1  ;d==0, so d=1
L1:  ADDI R3,R1,#-1
      BNEZ R3,L2        ;branch  b2(d!=1)
          ...
L2:
```

# 两级预测器基本工作原理

- 假设d的初始值序列为0，1，2
- b1 如果分支失败，b2一定也分支失败。
- 前面的两位标准的预测方案就没法利用这一点，而两级预测方案就可以。

```
 if (d==0)d=1;

  if (d==1) d=0;
```

翻译为DLX

```
      BNEZ R1,L1      ;branch b1(d!=0)
      ADDI R1,R0,#1   ;d==0, so d=1
L1:   ADDI R3,R1,#-1
      BNEZ R3,L2      ;branch b2(d!=1)
```

| Initial value of d | d==0? | b1 | Value of d before b2 | d==1? | b2 |
|---|---|---|---|---|---|
| 0 | yes | not taken | 1 | yes | not taken |
| 1 | no | taken | 1 | yes | not taken |
| 2 | no | taken | 2 | no | taken |

FIGURE 3.10 Possible execution sequences for a code fragment.

- 假设d的初始值在2和0之间切换。
- 用1-bit预测器，初始设置为预测失败，T表示预测成功，NT表示预测失败。
- 结论：这样的序列每次预测都错，预测错误率100%

```
      BNEZ R1,L1          ;branch b1(d!=0)
      ADDI R1,R0,#1       ;d==0, so d=1
L1:   ADDI R3,R1,#-1
      BNEZ R3,L2          ;branch b2(d!=1)
```

| d=? | b1 prediction | b1 action | New b1 prediction | b2 prediction | b2 action | New b2 prediction |
|---|---|---|---|---|---|---|
| 2 | NT | T | T | NT | T | T |
| 0 | T | NT | NT | T | NT | NT |
| 2 | NT | T | T | NT | T | T |
| 0 | T | NT | NT | T | NT | NT |

FIGURE 3.11 Behavior of a one-bit predictor initialized to not taken. T stands for taken, NT for not taken.

# Correlating Branches

- 基本思想：记为（1，1）
  - 用1位作为correlation位。记录最近一次执行的分支
  - 每个分支都有两个相互独立的预测位：一个预测位假设最近一次执行的分支失败时的预测位，另一个预测位是假设最近一次执行的分支成功时的预测位。
- 最近一次执行的分支与要预测的分支可能不是同一条指令

| Prediction bits | Prediction if last branch not taken | Prediction if last branch taken |
|---|---|---|
| NT/NT | not taken | not taken |
| NT/T | not taken | taken |
| T/NT | taken | not taken |
| T/T | taken | taken |

FIGURE 3.12 Combinations and meaning of the taken/not taken prediction bits. T stands for taken, NT for not taken.

- Correlating 预测器的预测和执行情况
- 显然只有在第一次d=2时，预测错误，其他都预测正确
- 记为（1，1）预测器，即根据最近一次分支的行为来选择一对1-bit预测器中的一个。
- 更一般的表示为（m, n)，即根据最近的m个分支，从$2^m$个分支预测器中选择预测器，每个预测器的位数为n

```
      BNEZ R1,L1          ;branch b1(d!=0)
      ADDI R1,R0,#1       ;d==0, so d=1
L1:   ADDI R3,R1,#-1
      BNEZ R3,L2          ;branch b2(d!=1)
```

| d=? | b1 prediction | b1 action | New b1 prediction | b2 prediction | b2 action | New b2 prediction |
|---|---|---|---|---|---|---|
| 2 | **NT**/NT | T | T/NT | NT/**NT** | T | NT/T |
| 0 | T/**NT** | NT | T/NT | **NT**/T | NT | NT/T |
| 2 | **T**/NT | T | T/NT | NT/**T** | T | NT/T |
| 0 | T/**NT** | NT | T/NT | **NT**/T | NT | NT/T |

FIGURE 3.13 The action of the one-bit predictor with one bit of correlation, initialized to not taken/not taken. T stands for taken, NT for not taken. The prediction used is shown in bold.

# Correlating Branches

- (2,2) predictor: 2-bit global, 2-bit local

# Gshare predictor

**Figure 3.4** A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

**FIGURE 3.15 Comparison of two-bit predictors.** A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating two-bit predictor with unlimited entries and a two-bit predictor with two bits of global history and a total of 1024 entries.

# Accuracy of Different Schemes

# Branch Prediction

- **Basic 2-bit predictor:**
- **关联预测器(n,2):**
  - 每个分支有多个 2-bit 预测器
  - 根据最近n次分支的执行情况从$2^n$中选择预测器
- **两级局部预测器(Local predictor):**
  - 每个分支有多个2-bit 预测器
  - 根据该分支的最近n次分支的执行情况从$2^n$中选择预测器
- **竞赛预测器(Tournament predictor):**
  - 结合关联预测器和两级局部预测器

# 竞赛预测器

- **全局预测器**
  - 使用最近n次分支跳转情况来索引，即全局预测器入口数：$2^n$每个入口是一个标准的2位预测器
- **局部预测器：设计为两层。**
  - 一个局部历史记录,使用指令地址的低m位进行索引，每个入口k位，分别对应这个入口最近的k次分支，即最近k次分支的 跳转情况
  - 从局部历史记录选择出的入口对一个$2^K$的入口表进行索引，这些入口由2位计数器构成，以提供本地预测。
- **选择器：**
  - 使用分支局部地址的低m位分支局部地址索引，每个索引得到一个两位计数器，用来选择使用局部预测器还是使用全局预测器的预测结果。
  - 在设计时默认使用局部预测器，当两个预测器都正确或都不正确时，不改变计数器；当全局预测器正确而局部预测器预测错误时，计数器加1，否则减1。

# Alpha 21264

Alpha 21264 竞赛预测器

# 选择器状态转移图

0/0, 1/0, 1/1
0/0, 1/1
0/1
1/0
选择局部预测器预测结果
0/1
1/0
1/0
0/1
选择全局预测器预测结果
0/0, 0/1, 1/1
0/0, 1/1

# Branch Prediction Performance

Branch predictor performance

# review

- **基于BHT表的预测器:**
  - Basic 2-bit predictor:
  - Correlating predictor:
    - 每个分支对应多个m-bit预测器
    - 最近n次的分支转移的每一种情况分别对应其中一个预测器
  - Local predictor:
    - 每个分支对应多个m-bit预测器
    - 该分支最近n次分支转移的每一种情况分别对应其中一个预测器
  - Tournament predictor:
    - 从多种预测器的预测结果中选择合适的预测结果。
    - 例如：Combine correlating predictor with local predictor
- **基于BTB的分支预测器**

# Simple dynamic prediction: Branch Target Buffer (BTB)

- **分支指令的地址作为BTB的索引，以得到分支预测地址**
  - 必须检测分支指令的地址是否匹配，以免用错误的分支地址
  - 从表中得到预测地址
  - 分支方向确定后，更新预测的PC

**Figure 2.23** The steps involved in handling an instruction with a branch-target buffer.

# Instruction Fetch Unit

# 预测器的基本结构及输入输出

- **使用FSM来完成决策**
  - 根据Branch History和PC来选择状态
  - 由状态决定输出
- **根据实际结果修改状态**

# 硬件支持推断执行以及精确中断

- **需要硬件缓存没有提交的指令结果: reorder buffer (ROB)**
  - 3 个域: 指令类型,目的地址, 值
  - Reorder buffer 可以作为操作数源 => 就像有更多的寄存器（与RS类似）
  - 当指令执行阶段完成后，用ROB的编号代替RS中的值
  - 增加指令提交阶段
  - ROB提供执行完成阶段和提交阶段的操作数
  - 一旦操作数提交，结果就写入寄存器
  - 在预测失败时，容易恢复推断执行的指令，或发生异常时，容易恢复状态

# 支持推断执行的 Tomasulo 算法的四阶段

- **1. Issue—get instruction from FP Op Queue**
  - 如果RS和ROB有空闲单元就发射指令。如果寄存器或ROB中源操作数可用，就将其发送到RS，目的地址的ROB编号也发送给RS
- **2. Execution—operate on operands (EX)**
  - 当操作数就绪后，开始执行。如果没有就绪，监测CDB，检查RAW相关
- **3. Write result—finish execution (WB)**
  - 将运算结果通过CDB传送给所有等待结果的FU以及ROB单元，标识RS可用
- **4. Commit—update register with reorder result**
  - 按ROB表中顺序，如果结果已有，就更新寄存器（或存储器），并将该指令从ROB表中删除
  - 预测失败或有中断时，刷新ROB
  - P191 Figure 3.14 (英文版), P141 Figure 3-9 (中文版)

# Issue

| Status | Wait until | Action or bookkeeping |
|---|---|---|
| Issue all instructions | Reservation station (r) and ROB (b) both available | `if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/`<br>`{h ← RegisterStat[rs].Reorder;`<br>`if (ROB[h].Ready)/* Instr completed already */`<br>`{RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;}`<br>`else {RS[r].Qj ← h;} /* wait for instruction */`<br>`} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;};`<br>`RS[r].Busy ← yes; RS[r].Dest ← b;`<br>`ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no;` |
| FP operations and stores | | `if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/`<br>`{h ← RegisterStat[rt].Reorder;`<br>`if (ROB[h].Ready)/* Instr completed already */`<br>`{RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;}`<br>`else {RS[r].Qk ← h;} /* wait for instruction */`<br>`} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};` |
| FP operations | | `RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes;`<br>`ROB[b].Dest ← rd;` |
| Loads | | `RS[r].A ← imm; RegisterStat[rt].Reorder ← b;`<br>`RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;` |
| Stores | | `RS[r].A ← imm;` |

h：ROB中与当前指令相关的指令对应的ROB编号；
b：当前指令对应的ROB编号

# Execute

| | | |
|---|---|---|
| Execute FP op | (RS[r].Qj == 0) and (RS[r].Qk == 0) | Compute results—operands are in Vj and Vk |
| Load step 1 | (RS[r].Qj == 0) and there are no stores earlier in the queue | `RS[r].A ← RS[r].Vj + RS[r].A;` |
| Load step 2 | Load step 1 done and all stores earlier in ROB have different address | Read from Mem[RS[r].A] |
| Store | (RS[r].Qj == 0) and store at queue head | `ROB[h].Address ← RS[r].Vj + RS[r].A;` |

# Write result & Commit

| | | |
|---|---|---|
| Write result all but store | Execution done at *r* and CDB available | b ← RS[r].Dest; RS[r].Busy ← no;<br>∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0});<br>∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0});<br>ROB[b].Value ← result; ROB[b].Ready ← yes; |
| Store | Execution done at *r* and (RS[r].Qk == 0) | ROB[h].Value ← RS[r].Vk; |
| Commit | Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes | d ← ROB[h].Dest; /* register dest, if exists */<br>if (ROB[h].Instruction==Branch)<br>{if (branch is mispredicted)<br>{clear ROB[h], RegisterStat; fetch branch dest;};}<br>else if (ROB[h].Instruction==Store)<br>{Mem[ROB[h].Destination] ← ROB[h].Value;}<br>else /* put the result in the register destination */<br>{Regs[d] ← ROB[h].Value;};<br>ROB[h].Busy ← no; /* free up ROB entry */<br>/* free up dest register if no one else writing it */<br>if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;}; |

# 例如：

```
LD      F6, 34(R2)
LD      F2, 45(R3)
MULT    F0, F2, F4
SUBD    F8, F6, F2
DIVD    F10, F0, F6
ADDD    F6, F8, F2
```

**假设：执行阶段的周期数**

**LD： 1 cycles  MULT: 10 cycles**

**SUBD/ADDD: 2cycles  DIVD: 40 cycles**

# Tomasulo With Reorder Buffer-Cycle 0

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 0 | Mult2 | No | | | | | | |

Reservation Station

LD F6, 34(R2)
LD F2, 45(R3)
MULT F0, F2, F4
SUBD F8, F6, F2
DIVD F10, F0, F6
ADDD F6, F8, F2

| Entry | Busy | Instruction | State | Destination | Value |
|---|---|---|---|---|---|
| 1 | | | | | |
| 2 | | | | | |
| 3 | | | | | |
| 4 | | | | | |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |
| 8 | | | | | |
| 9 | | | | | |
| 10 | | | | | |

| | Busy | Address |
|---|---|---|
| Load1 | | |
| Load2 | | |
| Load3 | | |

Reorder Buffer

| Cycle | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Reorder# | | | | | | | | | |
| | Busy | No | No | No | No | No | No | No | | No |

# Tomasulo With Reorder Buffer-Cycle 1

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 0 | Mult2 | No | | | | | | |

Reservation Station

LD F6, 34(R2)
LD F2, 45(R3)
MULT F0, F2, F4
SUBD F8, F6, F2
DIVD F10, F0, F6
ADDD F6, F8, F2

| | Entry | Busy | Instruction | State | Destination | Value |
|---|---|---|---|---|---|---|
| Head | 1 | Yes | LD F6,34(R2) | Issue | F6 | |
| | 2 | | | | | |
| | 3 | | | | | |
| | 4 | | | | | |
| | 5 | | | | | |
| | 6 | | | | | |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

| | Busy | Address |
|---|---|---|
| Load1 | Yes | 34+Regs[R2] |
| Load2 | | |
| Load3 | | |

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Reorder# | | | | #1 | | | | | |
| | Busy | No | No | No | Yes | No | No | No | | No |

# Tomasulo With Reorder Buffer-Cycle 2

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 0 | Mult2 | No | | | | | | |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | Yes | 34+Regs[R2] |
| Load2 | Yes | 45+Regs[R3] |
| Load3 | | |

LD F6, 34(R2)
LD F2, 45(R3)
MULT F0, F2, F4
SUBD F8, F6, F2
DIVD F10, F0, F6
ADDD F6, F8, F2

| | Entry | Busy | Instruction | State | Destination | Value |
|---|---|---|---|---|---|---|
| Head | 1 | Yes | LD F6,34(R2) | Ex1 | F6 | |
| tail | 2 | Yes | LD F2,45(R3) | Issue | F2 | |
| | 3 | | | | | |
| | 4 | | | | | |
| | 5 | | | | | |
| | 6 | | | | | |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

| Cycle | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | Reorder# | | #2 | | #1 | | | | | |
| | Busy | No | Yes | No | Yes | No | No | No | | No |

# Tomasulo With Reorder Buffer-Cycle 3

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | Yes | Mult | | Regs[F4] | #2 | | #3 |
| 0 | Mult2 | No | | | | | | |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | Yes | 45+Regs[R3] |
| Load3 | | |

LD F6, 34(R2)
LD F2, 45(R3)
MULT F0, F2, F4
SUBD F8, F6, F2
DIVD F10, F0, F6
ADDD F6, F8, F2

| | Entry | Busy | Instruction | State | Destination | Value |
|---|---|---|---|---|---|---|
| Head | 1 | Yes | LD F6,34(R2) | Write | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Ex1 | F2 | |
| tail | 3 | Yes | MULT F0,F2,F4 | Issue | F0 | |
| | 4 | | | | | |
| | 5 | | | | | |
| | 6 | | | | | |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | Reorder# | #3 | #2 | | #1 | | | | | |
| | Busy | Yes | Yes | No | Yes | No | No | No | | No |

# Tomasulo With Reorder Buffer-Cycle 4

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 2 | Add1 | Yes | SUB | Regs[F6] | Mem[45+regs[R3]] | | #2 | #4 |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 10 | Mult1 | Yes | Mult | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | No | | | | | | |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| Head | 2 | Yes | LD F2,45(R3) | Write | F2 | Mem[load2] |
| | 3 | Yes | MULT F0,F2,F4 | Issue | F0 | |
| tail | 4 | Yes | SUBD F8,F6,F2 | Issue | F8 | |
| | 5 | | | | | |
| | 6 | | | | | |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

LD F6, 34(R2)
LD F2, 45(R3)
MULT F0, F2, F4
SUBD F8, F6, F2
DIVD F10, F0, F6
ADDD F6, F8, F2

| Cycle | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 4 | Reorder# | #3 | #2 | | | #4 | | | | |
| | Busy | Yes | Yes | No | No | Yes | No | No | | No |

# Tomasulo With Reorder Buffer-Cycle 5

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 1 | Add1 | Yes | SUB | Regs[F6] | Mem[45+regs[R3]] | | | #4 |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 9 | Mult1 | Yes | Mult | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | Yes | DIV | | Regs[F6] | #3 | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

LD F6, 34(R2)
LD F2, 45(R3)
MULT F0, F2, F4
SUBD F8, F6, F2
DIVD F10, F0, F6
ADDD F6, F8, F2

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Ex1 | F0 | |
| | 4 | Yes | SUBD F8,F6,F2 | Ex1 | F8 | |
| Tail | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| | 6 | | | | | |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

| Cycle | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | Reorder# | #3 | #2 | | | #4 | #5 | | | |
| | Busy | Yes | No | No | No | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 6

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | Yes | SUB | Regs[F6] | Mem[45+regs[R3]] | | | #4 |
| 0 | Add2 | Yes | ADD | | Regs[F2] | #4 | | #6 |
| 0 | Add3 | No | | | | | | |
| 8 | Mult1 | Yes | MULT | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | Yes | DIV | | Regs[F6] | #3 | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Ex2 | F0 | |
| | 4 | Yes | SUBD F8,F6,F2 | Ex2 | F8 | |
| | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Issue | F6 | |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 6 | Reorder# | #3 | | | #6 | #4 | #5 | | | |
| | Busy | Yes | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 7

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 2 | Add2 | Yes | ADD | #4 | Regs[F2] | | | #6 |
| 0 | Add3 | No | | | | | | |
| 7 | Mult1 | Yes | MULT | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | Yes | DIV | | Regs[F6] | #3 | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Ex3 | F0 | |
| | 4 | Yes | SUBD F8,F6,F2 | Write | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Issue | F6 | |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 | Reorder# | #3 | | | #6 | #4 | #5 | | | |
| | Busy | Yes | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 8

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 1 | Add2 | Yes | ADD | #4 | Regs[F2] | | | #6 |
| 0 | Add3 | No | | | | | | |
| 6 | Mult1 | Yes | MULT | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | Yes | DIV | | Regs[F6] | #3 | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Ex4 | F0 | |
| | 4 | Yes | SUBD F8,F6,F2 | Write | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Ex1 | F6 | |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

| Cycle | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 8 | Reorder# | #3 | | | #6 | #4 | #5 | | | |
| | Busy | Yes | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 9

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | Yes | ADD | #4 | Regs[F2] | | | #6 |
| 0 | Add3 | No | | | | | | |
| 5 | Mult1 | Yes | MULT | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | Yes | DIV | | Regs[F6] | #3 | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Ex5 | F0 | |
| | 4 | Yes | SUBD F8,F6,F2 | Write | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Ex2 | F6 | |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | Reorder# | #3 | | | #6 | #4 | #5 | | | |
| | Busy | Yes | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 10

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 4 | Mult1 | Yes | MULT | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | Yes | DIV | | Regs[F6] | #3 | | #5 |

Reservation Station

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Ex6 | F0 | |
| | 4 | Yes | SUBD F8,F6,F2 | Write | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10 | Reorder# | #3 | | | #6 | #4 | #5 | | | |
| | Busy | Yes | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 11

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 3 | Mult1 | Yes | MULT | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | Yes | DIV | | Regs[F6] | #3 | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Ex7 | F0 | |
| | 4 | Yes | SUBD F8,F6,F2 | Write | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 11 | Reorder# | #3 | | | #6 | #4 | #5 | | | |
| | Busy | Yes | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 12

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 2 | Mult1 | Yes | MULT | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | Yes | DIV | | Regs[F6] | #3 | | #5 |

Reservation Station

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Ex8 | F0 | |
| | 4 | Yes | SUBD F8,F6,F2 | Write | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 12 | Reorder# | #3 | | | #6 | #4 | #5 | | | |
| | Busy | Yes | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 13

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 1 | Mult1 | Yes | MULT | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | Yes | DIV | | Regs[F6] | #3 | | #5 |

Reservation Station

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Ex9 | F0 | |
| | 4 | Yes | SUBD F8,F6,F2 | Write | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

Reorder Buffer

| Cycle | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 13 | Reorder# | #3 | | | #6 | #4 | #5 | | | |
| | Busy | Yes | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 14

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | Yes | MULT | Mem[45+Regs[R3]] | Regs[F4] | | | #3 |
| 0 | Mult2 | Yes | DIV | | Regs[F6] | #3 | | #5 |

Reservation Station

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Ex10 | F0 | |
| | 4 | Yes | SUBD F8,F6,F2 | Write | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 14 | Reorder# | #3 | | | #6 | #4 | #5 | | | |
| | Busy | Yes | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 15

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 40 | Mult2 | Yes | DIV | #2*Regs[F4] | Regs[F6] | | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| Head | 3 | Yes | MULT F0,F2,F4 | Write | F0 | #2*F4 |
| | 4 | Yes | SUBD F8,F6,F2 | Write | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Issue | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 15 | Reorder# | #3 | | | #6 | #4 | #5 | | | |
| | Busy | Yes | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 16

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 39 | Mult2 | Yes | DIV | #2*Regs[F4] | Regs[F6] | | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| | 3 | Yes | MULT F0,F2,F4 | Commit | F0 | #2*F4 |
| Head | 4 | Yes | SUBD F8,F6,F2 | Write | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Ex1 | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

| Cycle | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 16 | Reorder# | | | | #6 | #4 | #5 | | | |
| | Busy | No | No | No | Yes | Yes | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 17

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 38 | Mult2 | Yes | DIV | #2*Regs[F4] | Regs[F6] | | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| | 3 | Yes | MULT F0,F2,F4 | Commit | F0 | #2*F4 |
| | 4 | Yes | SUBD F8,F6,F2 | Commit | F8 | F6-#2 |
| Head | 5 | Yes | DIVD F10,F0,F6 | Ex2 | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 17 | Reorder# | | | | #6 | | #5 | | | |
| | Busy | No | No | No | Yes | No | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 18

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 37 | Mult2 | Yes | DIV | #2*Regs[F4] | Regs[F6] | | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| | 3 | Yes | MULT F0,F2,F4 | Commit | F0 | #2*F4 |
| | 4 | Yes | SUBD F8,F6,F2 | Commit | F8 | F6-#2 |
| Head | 5 | Yes | DIVD F10,F0,F6 | Ex3 | F10 | |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 18 | Reorder# | | | | #6 | | #5 | | | |
| | Busy | No | No | No | Yes | No | Yes | No | | No |

Continue……37 Cycles

# Tomasulo With Reorder Buffer-Cycle 55

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 0 | Mult2 | Yes | DIV | #2*Regs[F4] | Regs[F6] | | | #5 |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| | 3 | Yes | MULT F0,F2,F4 | Commit | F0 | #2*F4 |
| | 4 | Yes | SUBD F8,F6,F2 | Commit | F8 | F6-#2 |
| Head | 5 | Yes | DIVD F10,F0,F6 | Ex40 | F10 | #3/F6 |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 54 | Reorder# | | | | #6 | | #5 | | | |
| | Busy | No | No | No | Yes | No | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 56

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 0 | Mult2 | No | | | | | | |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| | 3 | Yes | MULT F0,F2,F4 | Commit | F0 | #2*F4 |
| | 4 | Yes | SUBD F8,F6,F2 | Commit | F8 | F6-#2 |
| Head | 5 | Yes | DIVD F10,F0,F6 | Write | F10 | #3/F6 |
| Tail | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 56 | Reorder# | | | | #6 | | #5 | | | |
| | Busy | No | No | No | Yes | No | Yes | No | | No |

# Tomasulo With Reorder Buffer-Cycle 57

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 0 | Mult2 | No | | | | | | |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| | 3 | Yes | MULT F0,F2,F4 | Commit | F0 | #2*F4 |
| | 4 | Yes | SUBD F8,F6,F2 | Commit | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Commit | F10 | #3/F6 |
| Head | 6 | Yes | ADDD F6,F8,F2 | Write | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 57 | Reorder# | | | | #6 | | | | | |
| | Busy | No | No | No | Yes | No | No | No | | No |

# Tomasulo With Reorder Buffer-Cycle 58

| Time | Name | Busy | Op | Vj | Vk | Qj | Qk | Dest |
|---|---|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | | | |
| 0 | Add2 | No | | | | | | |
| 0 | Add3 | No | | | | | | |
| 0 | Mult1 | No | | | | | | |
| 0 | Mult2 | No | | | | | | |

Reservation Station

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

| | Entry | Busy | Instruction | State | Dest. | Value |
|---|---|---|---|---|---|---|
| | 1 | Yes | LD F6,34(R2) | Commit | F6 | Mem[load1] |
| | 2 | Yes | LD F2,45(R3) | Commit | F2 | Mem[load2] |
| | 3 | Yes | MULT F0,F2,F4 | Commit | F0 | #2*F4 |
| | 4 | Yes | SUBD F8,F6,F2 | Commit | F8 | F6-#2 |
| | 5 | Yes | DIVD F10,F0,F6 | Commit | F10 | #3/F6 |
| Head | 6 | Yes | ADDD F6,F8,F2 | Commit | F6 | #4+F2 |
| | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |
| | 10 | | | | | |

Reorder Buffer

Cycle

| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | …… | F30 |
|---|---|---|---|---|---|---|---|---|---|---|
| 58 | Reorder# | | | | | | | | | |
| | Busy | No | No | No | No | No | No | No | | No |

# Tomasulo With Reorder Buffer-Summary

| Instruction | Issue | Exec Comp | WriteBack | Commit |
|---|---|---|---|---|
| LD F6,34(R2) | 1 | 2 | 3 | 4 |
| LD F2,45(R3) | 2 | 3 | 4 | 5 |
| MULT F0,F2,F4 | 3 | 5~14 | 15 | 16 |
| SUBD F8,F6,F2 | 4 | 5~6 | 7 | 17 |
| DIVD F10,F0,F6 | 5 | 16~55 | 56 | 57 |
| ADDD F6,F8,F2 | 6 | 8~9 | 10 | 58 |

In-order Issue/Commit　　Out-of-Order Execution/WriteBack

是不是可以进一步优化？

# Example of Speculative State of Reorder Buffer

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | Add1 | No | | | | |
| 0 | Add2 | No | | | | |
| 0 | Add3 | No | | | | |
| 0 | Mult1 | No | MULT | Mem[0+Regs[R1]] | Regs[F2] | #2 |
| 0 | Mult2 | No | MULT | Mem[0+Regs[R1]] | Regs[F2] | #7 |

Reservation Stations

| | Busy | Address |
|---|---|---|
| Load1 | No | |
| Load2 | No | |
| Load3 | | |

First loop

Second loop

| Entry | Busy | Instruction | State | Destination | Value |
|---|---|---|---|---|---|
| 1 | No | LD F0, 0(R1) | commit | F0 | Mem[0+R1] |
| 2 | No | MULT F4, F0, F2 | commit | F4 | F0 x F2 |
| 3 | Yes | SD 0(R1), F4 | write | 0+Reg[R1] | #2 |
| 4 | Yes | SUBI R1, R1, 8 | write | R1 | R1 - 8 |
| 5 | Yes | BNEZ R1, Loop | write | | |
| 6 | Yes | LD F0, 0(R1) | write | F0 | Mem[#4] |
| 7 | Yes | MULT F4, F0, F2 | write | F4 | #6 X F2 |
| 8 | Yes | SD 0(R1), F4 | write | 0+Regs[R1] | #7 |
| 9 | Yes | SUBI R1, R1, 8 | write | R1 | #4 - 8 |
| 10 | Yes | BNEZ R1, Loop | write | | |

Reorder Buffer

| | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
|---|---|---|---|---|---|---|---|---|---|
| Reorder # | 6 | | 7 | | | | | | |
| Busy | yes | no | yes | no | no | no | no | | no |

Multiply has just reached commit, so other instructions can start committing

# 使用ROB保持机器的精确状态

- **ROB维持了机器的精确状态，允许投机执行**
  - 直到确认无异常 然后进入提交阶段
  - 直到确定分支预测正确进入提交阶段
  - 如果有异常或预测错误
    - 刷新ROB、RS和寄存器结果状态表
- **存储器操作使用类似的方法**
  - Memory Ordering Buffer（MOB）
    - Store操作的结果先存放到MOB中，然后提交阶段按存储操作的程序序提交

# Memory Disambiguation：处理对存储器引用的数据相关

- **Question: 给定一个指令序列，store，load 这两个操作是否有关？即下列代码是否有相关问题?**

  **Eg: st 0(R2),R5**
  **ld R6,0(R3)**

- **我们是否可以较早启动ld?**
  - Store的地址可能会延迟很长时间才能得到.
  - 我们也许想在同一个周期开始这两个操作的执行.
- **两种方法:**
  - No Speculation: 不进行load操作，直到我们确信地址 0(R2) □ 0(R3)
  - Speculation: 我们可以假设他们相关还是不相关 (called "dependence speculation")，如果推测错误通过ROB来修正
- **参考书：Gonzalez, A., et al. (2011). "Processor Microarchitecture: An Implementation Perspective." Synthesis Lectures on Computer Architecture #12，Morgan & Claypool Publishers**

# Memory Disambiguation

TABLE 6.1: Memory disambiguation schemes.

| NAME | SPECULATIVE | DESCRIPTION |
|---|---|---|
| Total Ordering | No | All memory accesses are processed in order. |
| Partial Ordering | No | All stores are processed in order, but loads execute out of order as long as all previous stores have computed their address. |
| Load Ordering<br>Store Ordering | No | Execution between loads and stores is out of order, but all loads execute in order among them, and all stores execute in order among them. |
| Store Ordering | Yes | Stores execute in order, but loads execute completely out of order. |

- **非投机方式的基本原则：当前存储器指令之前的store指令计算存储器地址后，才能执行当前的存储器操作**

# 04/24-Tomasulo小结 #1/3

- **Reservations stations: 寄存器重命名，缓冲源操作数**
  - 避免寄存器成为瓶颈
  - 避免了Scoreboard中无法解决的 WAR, WAW hazards
  - 允许硬件做循环展开
  - 不限于基本块(IU先行，解决控制相关)
- **Reorder Buffer:**
  - 提供了撤销指令运行的机制
  - 指令以发射序存放在ROB中
  - 指令顺序提交
- **分支预测对提高性能是非常重要的**
  - 推断执行: 在控制相关还没有解决情况下，就开始执行
  - 推断执行利用了ROB撤销指令执行的机制
    - 处理预测错误时，撤销 推测执行的指令
  - 基于BHT的分支预测技术
  - 基于BTB的分支预测技术

# 04/24-Tomasulo小结 #2/3

- **贡献**
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- **360/91 后 Pentium II; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264使用这种技术**
- **不足之处：**
  - Too many value copy operations
    - Register File →RS→ROB→Register File
  - Too many muxes/busses (CDB)
    - Values are from everywhere to everywhere else!
  - Reservation Stations mix values(data) and tags (control)
    - Slow down max clock frequency

# 04/24-Tomasulo小结 #3/3

- **存储器访问的冲突消解**
  - 非投机方式的冲突消解
    - Total Ordering
    - Partial Ordering
      - Load指令前的store指令已经完成了地址计算，有可能乱序执行存储器load操作
    - Load Ordering, Store Ordering
      - Load指令前的存储器访问指令已经完成了地址计算，load队头的load操作有可能在store指令之前执行访存操作。
  - 投机方式的执行
    - Store Ordering
    - 假设Load操作与之前未计算出有效地址的store操作无关。
- **问题：给出四种访问方式挖掘并行性的能力排序。**

# Memory Disambiguation

TABLE 6.1: Memory disambiguation schemes.

| NAME | SPECULATIVE | DESCRIPTION |
|---|---|---|
| Total Ordering | No | All memory accesses are processed in order. |
| Partial Ordering | No | All stores are processed in order, but loads execute out of order as long as all previous stores have computed their address. |
| Load Ordering<br>Store Ordering | No | Execution between loads and stores is out of order, but all loads execute in order among them, and all stores execute in order among them. |
| Store Ordering | Yes | Stores execute in order, but loads execute completely out of order. |

- **非投机方式的基本原则：当前存储器指令之前的store指令计算存储器地址后，才能执行当前的存储器操作**

# Load Ordering. Store Ordering

**FIGURE 6.7:** Schematics of the AMD K6 pipeline to implement a load-ordering store-ordering memory disambiguation policy.

Load queue: 以程序序存储load指令，按照FIFO方式流出

Address generation: 生成存储器访问有效地址

Store queue: 以程序序存储store指令，按照FIFO方式流出

Store buffer: 以程序序保留store操作（有效地址，值），一直到其成为最早的store操作，才实际更新存储器

# Store 操作

**FIGURE 6.7:** Schematics of the AMD K6 pipeline to implement a load-ordering store-ordering memory disambiguation policy.

- **处于Store队列对头，并且计算地址的操作数准备好，就向前流动，在Address Generation阶段计算地址**
- **若要存储的数据准备好，则读该数据，若未准备好，则流水线停顿**
- **Disambiguation阶段：StoreBuffer中按程序序，执行最先的Store操作**

# Load 操作

**FIGURE 6.7:** Schematics of the AMD K6 pipeline to implement a load-ordering store-ordering memory disambiguation policy.

- **处于Load队列对头，并且计算地址的操作数准备好，就向前流动；在Address Generation阶段计算地址**
- **Disambiguation阶段：Load操作可以继续前行需同时满足的三个条件**
  - 在**StoreBuffer中**比该Load操作早的Store的地址与该Load地址不同
  - 如果正在进行地址计算的Store操作比该Load操作早，进行部分地址比较时，部分地址不同
  - 在StoreQueue中不存在比该Load早的Store操作。

# Partial Ordering （MIPS R10000）

**FIGURE 6.8:** Schematics of the MIPS R10000 pipeline to implement the partial ordering memory disambiguation policy.

**Load/store queue: 长度为16的队列，存储load/store指令，直到其操作数准备好。**

**Indetermination matrix: 16x16 矩阵（half），每行每列代表队列中的存储器操作。当有存储器操作进入队列时，对应列置位。当存储器指令计算有效地址时，对应列复位。计算有效地址的条件之一，该操作对应行中非对角线元素为0**

**FIGURE 6.9:** Example of a 6-entry indetermination matrix.

**FIGURE 6.10:** Example of a 6-entry dependency matrix.

- **Dependency matrix: 16x16 矩阵，每行每列对应load/store队列的存储器操作。Load操作如果依赖于前面的store，则将该load操作的行中对应该store的列置位。Store操作更新存储器时，将复位该store操作对应的列。只有当load操作对应的行全0时，方可执行load操作**
- **Address generation: 计算存储器操作的有效地址**
- **Address queue:保存访问cache的loads/store操作的地址。如果是load操作，除了保存地址，还需要比较所有在该load操作之前的store的地址，如果匹配，则对dependency matrix对应位置位。**

# Speculative Memory Disambiguation

**FIGURE 6.11:** Schematics of the Alpha 21264 pipeline to implement a speculative memory disambiguation policy.

Load/Store Queue: 保存存储器操作的队列,直到可以计算有效地址

Load Queue: 该队列以程序序存储load操作的存储器物理地址。该队列有32项

Store Queue: 存储store操作的存储器物理地址以及要存储的值。该对列32项

Wait Table: 具有1024 项，每项1位的表，由存储器指令虚拟地址索引. 当我们检测到一个load操作超越了与它相关的store操作时，该load对应项置1，取指部件读取该信息，将不会再提前发射出去。该表每隔16384周期复位一次，否则该表可能所有项均为1

# Memory Disambiguation:

# 如何使CPI < 1？ (1/2)

- **前面所述的各种技术主要通过减少数据相关和控制相关，使得CPI = 1（CPI接近1）**
- **是否能够使CPI < 1? 多发射处理器**
- **两种基本方法: Superscalar 、VLIW**
- **Superscalar:**
  - 每个时钟周期所发射的指令数不定（1 - 8条）
  - 由编译器或硬件完成调度
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000
  - 该方法对目前通用计算是最成功的方法
- **Instructions Per Clock (IPC) vs. CPI**

# 如何使 CPI < 1？ (2/2)

- **(Very) Long Instruction Words (V)LIW:**
  - 每个时钟周期流出的指令数（操作）固定 (4-16)
  - 由编译器调度，实际上由多个单操作指令构成一个超长指令
  - 目前比较成功的应用于DSP，多媒体应用
    - DSA （Domain Specific Architecture）
  - 1999/2000 HP和Intel达成协议共同研究VLIW
  - Intel Architecture-64 (Merced/A-64) 64-bit address
  - Style: “Explicitly Parallel Instruction Computer (EPIC)”

# Machine Parallelism

**Figure 4.2**
Machine Parallelism: (a) No Parallelism (Nonpipelined); (b) Temporal Parallelism (Pipelined); (c) Spatial Parallelism (Multiple Units); (d) Combined Temporal and Spatial Parallelism.

**Figure 4.16**
The Necessity of Instruction Dispatching in a Superscalar Pipeline.

# 简单的超标量流水线

**Figure 4.3**
A Parallel Pipeline of Width $s = 3$.

**Figure 4.4**
(a) The Five-Stage i486 Scalar Pipeline;
(b) The Five-Stage Pentium Parallel Pipeline of Width $s = 2$.

# 具有多种执行部件（流水化）

**Figure 4.5**
A Diversified Parallel Pipeline with Four Execution Pipes.

**Figure 4.9**
A Dynamic Pipeline of Width $s = 3$.

**Figure 4.20**
A Dynamic Pipeline with Reservation Station and Reorder Buffer.

# 超标量流水线中的分支预测

**Figure 5.10**
Branch Prediction in the PowerPC 604 Superscalar Microprocessor.

# 用于多发射处理器的五种主要方法

| Common name | Issue structure | Hazard detection | Scheduling | Distinguishing characteristic | Examples |
|---|---|---|---|---|---|
| Superscalar (static) | Dynamic | Hardware | Static | In-order execution | Mostly in the embedded space: MIPS and ARM, including the Cortex-A53 |
| Superscalar (dynamic) | Dynamic | Hardware | Dynamic | Some out-of-order execution, but no speculation | None at the present |
| Superscalar (speculative) | Dynamic | Hardware | Dynamic with speculation | Out-of-order execution with speculation | Intel Core i3, i5, i7; AMD Phenom; IBM Power 7 |
| VLIW/LIW | Static | Primarily software | Static | All hazards determined and indicated by compiler (often implicitly) | Most examples are in signal processing, such as the TI C6x |
| EPIC | Primarily static | Primarily software | Mostly static | All hazards determined and indicated explicitly by the compiler | Itanium |

**Figure 3.19 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them.** This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix H focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.

# Superscalar DLX

- **Superscalar DLX: 每个时钟周期发射2条指令，1 条FP指令和一条其他指令**
  - 每个时钟周期取64位; 左边为Int , 右边为FP
  - 只有第一条指令发射了，才能发射第二条
  - 需要更多的寄存器端口，因为如果两条指令中第一条指令是对FP的load操作（通过整数部件完成），另一条指令为浮点操作指令，则都会有对浮点寄存器文件的操作
- **原来1 cycle load 延时在Superscalar中扩展为3条指令**

| Type | Pipe Stages | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Int. instruction | IF | ID | EX | MEM | WB | | | |
| Fp. Instruction | IF | ID | EX | MEM | WB | | | |
| Int. instruction | | IF | ID | EX | MEM | WB | | |
| Fp. Instruction | | IF | ID | EX | MEM | WB | | |
| Int. Instruction | | | IF | ID | EX | MEM | WB | |
| Fp. Instruction | | | IF | ID | EX | MEM | WB | |

# Review: 具有最小stalls数的循环展开优化

```
1 Loop:     LD      F0,0(R1)
2           LD      F6,-8(R1)
3           LD      F10,-16(R1)
4           LD      F14,-24(R1)
5           ADDD    F4,F0,F2
6           ADDD    F8,F6,F2
7           ADDD    F12,F10,F2
8           ADDD    F16,F14,F2
9           SD      0(R1),F4
10          SD      -8(R1),F8
11          SUBI    R1,R1,#32
12          SD      16(R1),F12
13          BNEZ    R1,LOOP
14          SD      8(R1),F16       ; 8-32 = -24
```

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

**14 clock cycles, or 3.5 per iteration**

# 采用Superscalar技术的循环展开

| *Integer instruction* | *FP instruction* | *Clock cycle* |
|---|---|---|
| Loop: LD F0,0(R1) | | 1 |
| LD F6,-8(R1) | | 2 |
| LD F10,-16(R1) | ADDD F4,F0,F2 | 3 |
| LD F14,-24(R1) | ADDD F8,F6,F2 | 4 |
| LD F18,-32(R1) | ADDD F12,F10,F2 | 5 |
| SD 0(R1),F4 | ADDD F16,F14,F2 | 6 |
| SD -8(R1),F8 | ADDD F20,F18,F2 | 7 |
| SD -16(R1),F12 | | 8 |
| SUBI R1,R1,#40 | | 9 |
| SD +16(R1),F16 | | 10 |
| BNEZ R1,LOOP | | 11 |
| SD +8(R1),F20 | | 12 |

- 循环展开5次以消除延时 (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration (1.5X)

# 多发射的问题

- **如果Integer和FP操作很容易区分组合，那么对这类程序在下列条件满足的情况下理想CPI= 0.5：**
  - 程序中50% 为FP 操作
  - 没有任何相关
- **如果在同一时刻发射的指令越多，译码和发射就越困难**
  - 即使是同一时刻发射2条 =>需检查2个操作码，6个寄存器描述符 ，检查是发射1条还是2条指令。
- **VLIW**
  - 指令字较长可以容纳较多的操作
  - 根据定义,VLIW中的所有操作是由编译时刻组合的，并且是相互无关的，也就是说：可以并行执行
  - 例如 2 个整数操作，2个浮点操作，2个存储器引用，1个分支指令
    - 每一个操作用16 到 24 位 表示 => 共7*16 = 112 bits 到 7*24 = 168 bits wide
  - 需要用编译技术调度来解决分支问题

# 基于VLIW的循环展开

| Memory reference 1 | Memory reference 2 | FP operation 1 | FP op. 2 | Int. op/ branch | Clock |
|---|---|---|---|---|---|
| LD F0,0(R1) | LD F6,-8(R1) | | | | 1 |
| LD F10,-16(R1) | LD F14,-24(R1) | | | | 2 |
| LD F18,-32(R1) | LD F22,-40(R1) | ADDD F4,F0,F2 | ADDD F8,F6,F2 | | 3 |
| LD F26,-48(R1) | | ADDD F12,F10,F2 | ADDD F16,F14,F2 | | 4 |
| | | ADDD F20,F18,F2 | ADDD F24,F22,F2 | | 5 |
| SD 0(R1),F4 | SD -8(R1),F8 | ADDD F28,F26,F2 | | | 6 |
| SD -16(R1),F12 | SD -24(R1),F16 | | | SUBI R1,R1,#48 | 7 |
| SD 16(R1),F20 | SD 8(R1),F24 | | | | 8 |
| SD -0(R1),F28 | | | | BNEZ R1,LOOP | 9 |

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

注: 在VLIW中，一条超长指令有更多的读写寄存器操作(15 vs. 6 in SS)

# Trace Scheduling

- **消除分支的一种策略**
- **两步：**
  - Trace Selection
    - 搜索可能最长的直线型代码（由一组基本块构成）（通过静态预测或profile技术）(trace)
  - Trace Compaction
    - 将trace中的指令拼装为若干条VLIW 指令
    - 需要一些保存环境的代码，以防预测错误
- **由编译器撤销预测错误造成的后果（恢复寄存器的原值)**

# HW推断执行(Tomasulo) vs. SW (VLIW) 推断执行

- **HW 确定地址冲突**
- **HW 分支预测较好，预测准确率较高**
- **HW 可支持精确中断模型**
- **HW 不必执行保存环境和恢复环境的指令**
- **SW 推断执行比HW设计简单的多**

# Superscalar vs. VLIW

- **Superscalar**
  - 代码量较小
  - 二进制兼容性好
- **VLIW**
  - 译码、发射指令的硬件设计简单
  - 更多的寄存器，一般使用多个寄存器文件而不是多端口寄存器文件

# Superscalar 的动态调度（1/2）

- **静态调度的缺陷：**
  - 有相关就停止发射
  - 基于原来Superscalar的代码生成器所生成的代码可能在新的Superscalar上运行效率较差，代码与superscalar的结构有关

# Superscalar 的动态调度（2/2）

- **用Tomasulo如何发射两条指令并保持指令序**
  - 假设有1 浮点操作，1个整数操作
  - Tomasulo控制器一个控制整型操作的发射，一个控制浮点型操作的发射
- **如果每个周期发射两条不同的指令，比较容易保持指令序（整型类操作序，浮点类操作序）**
- **现在只有FP的Loads操作可能会引起整型操作发射和浮点操作发射的相关**
- **存储器引用问题：**
  - 将load的保留站组织成队列方式，操作数必须按指令序读取
  - Load操作时检测Store队列中Store的地址以防止RAW冲突
  - Store操作时检测Load队列的地址，以防止WAR相关
  - Store操作按指令序进行，防止WAW相关

# Example

Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

```
Loop:  LD R2,0(R1)          ; R2=array element
       DADDIU R2,R2,#1 ; increment R2
       SD R2,0(R1)          ;store result
       DADDIU R1,R1,#8 ;increment pointer
       BNE R2,R3,LOOP ;branch if not last element
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.

# Performance of Dynamic SS

| Iteration number | Instructions | Issues at clock cycle number | Executes at clock cycle number | Memory access at clock cycle number | Write CDB at clock cycle number | Comment |
|---|---|---|---|---|---|---|
| 1 | LD R2,0(R1) | 1 | 2 | 3 | 4 | First issue |
| 1 | DADDIU R2,R2,#1 | 1 | 5 | | 6 | Wait for LW |
| 1 | SD R2,0(R1) | 2 | 3 | 7 | | Wait for DADDIU |
| 1 | DADDIU R1,R1,#8 | 2 | 3 | | 4 | Execute directly |
| 1 | BNE R2,R3,LOOP | 3 | 7 | | | Wait for DADDIU |
| 2 | LD R2,0(R1) | 4 | 8 | 9 | 10 | Wait for BNE |
| 2 | DADDIU R2,R2,#1 | 4 | 11 | | 12 | Wait for LW |
| 2 | SD R2,0(R1) | 5 | 9 | 13 | | Wait for DADDIU |
| 2 | DADDIU R1,R1,#8 | 5 | 8 | | 9 | Wait for BNE |
| 2 | BNE R2,R3,LOOP | 6 | 13 | | | Wait for DADDIU |
| 3 | LD R2,0(R1) | 7 | 14 | 15 | 16 | Wait for BNE |
| 3 | DADDIU R2,R2,#1 | 7 | 17 | | 18 | Wait for LW |
| 3 | SD R2,0(R1) | 8 | 15 | 19 | | Wait for DADDIU |
| 3 | DADDIU R1,R1,#8 | 8 | 14 | | 15 | Wait for BNE |
| 3 | BNE R2,R3,LOOP | 9 | 19 | | | Wait for DADDIU |

**Figure 2.20 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation.** Note that the LD following the BNE cannot start execution earlier because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch-condition evaluation allow multiple instructions to execute in the same cycle. Figure 2.21 shows this example with speculation,

| Iteration number | Instructions | | Issues at clock number | Executes at clock number | Read access at clock number | Write CDB at clock number | Commits at clock number | Comment |
|---|---|---|---|---|---|---|---|---|
| 1 | LD | R2,0(R1) | 1 | 2 | 3 | 4 | 5 | First issue |
| 1 | DADDIU | R2,R2,#1 | 1 | 5 | | 6 | 7 | Wait for LW |
| 1 | SD | R2,0(R1) | 2 | 3 | | | 7 | Wait for DADDIU |
| 1 | DADDIU | R1,R1,#8 | 2 | 3 | | 4 | 8 | Commit in order |
| 1 | BNE | R2,R3,LOOP | 3 | 7 | | | 8 | Wait for DADDIU |
| 2 | LD | R2,0(R1) | 4 | 5 | 6 | 7 | 9 | No execute delay |
| 2 | DADDIU | R2,R2,#1 | 4 | 8 | | 9 | 10 | Wait for LW |
| 2 | SD | R2,0(R1) | 5 | 6 | | | 10 | Wait for DADDIU |
| 2 | DADDIU | R1,R1,#8 | 5 | 6 | | 7 | 11 | Commit in order |
| 2 | BNE | R2,R3,LOOP | 6 | 10 | | | 11 | Wait for DADDIU |
| 3 | LD | R2,0(R1) | 7 | 8 | 9 | 10 | 12 | Earliest possible |
| 3 | DADDIU | R2,R2,#1 | 7 | 11 | | 12 | 13 | Wait for LW |
| 3 | SD | R2,0(R1) | 8 | 9 | | | 13 | Wait for DADDIU |
| 3 | DADDIU | R1,R1,#8 | 8 | 9 | | 10 | 14 | Executes earlier |
| 3 | BNE | R2,R3,LOOP | 9 | 13 | | | 14 | Wait for DADDIU |

**Figure 2.21 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation.** Note that the LD following the BNE can start execution early because it is speculative.

# 04/29-review

- **存储器访问的冲突消解**
  - Total Ordering
  - Partial Ordering
  - Load Ordering, Store Ordering
  - Store Ordering

- **Superscalar and VLIW: CPI < 1 (IPC > 1)**
  - Dynamic issue vs. Static issue
  - 同一时刻发射更多的指令 => 导致更大的冲突开销

# Memory Disambiguation

TABLE 6.1: Memory disambiguation schemes.

| NAME | SPECULATIVE | DESCRIPTION |
|---|---|---|
| Total Ordering | No | All memory accesses are processed in order. |
| Partial Ordering | No | All stores are processed in order, but loads execute out of order as long as all previous stores have computed their address. |
| Load Ordering<br>Store Ordering | No | Execution between loads and stores is out of order, but all loads execute in order among them, and all stores execute in order among them. |
| Store Ordering | Yes | Stores execute in order, but loads execute completely out of order. |

- **非投机方式的基本原则：当前存储器指令之前的store指令计算存储器地址后，才能执行当前的存储器操作**

# 多发射处理器受到的限制（1/2）

- **程序内在的ILP的限制**
  - 如果每5条指令中有1条相关指令：如何保持5-路VLIW 并行?
  - 部件的操作延时：许多操作需要调度，使部件延时加大
- **多指令流出的处理器需要大量的硬件资源**
  - 需要多个功能部件来使得多个操作并行(Easy)
  - 需要更大的指令访问带宽(Easy)
  - 需要增加寄存器文件的端口数（以及通信带宽）(Hard)
  - 增加存储器的端口数（带宽）(Harder)

# 多发射处理器受到的限制（2/2）

- **一些由Superscalar或VLIW的实现带来的特殊问题**
  - Superscalar的译码、发射问题
    - 到底能发射多少条指令?
  - VLIW 代码量问题：循环展开 + VLIW中无用的区域
  - VLIW 互锁 => 1 个相关导致所有指令停顿
  - VLIW 的二进制兼容问题

# For most apps, most execution units lie idle in an OoO superscalar

**For an 8-way superscalar.**

Sources of all unused issue cycles in an 8-issue superscalar processor. Processor busy represents the utilized issue slots; all others represent wasted issue slots.

From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism", ISCA 1995.

# Superscalar Machine Efficiency

*Completely idle cycle (vertical waste)*

*Partially filled cycle, i.e., IPC < 4 (horizontal waste)*

# Multithreading

- **背景：从单线程程序挖掘指令集并行越来越困难**
- **许多工作任务可以使用线程级并行来完成**
  - 线程级并行来源于多道程序设计
  - 线程级并行的基础是多线程应用，即一个任务可以用多个线程并行来加速
- **多线程应用可以用线程级并行来提高单个处理器的利用率**
  - 针对单个处理器：多个线程以重叠方式共享单个处理器的功能单元

# Pipeline Hazards

- 每条指令与前一条指令存在RAW相关

- 如何处理相关?
  - 使用interlock机制(slow)
  - 或定向路径

# Multithreading

- **如何保证流水线中指令间无数据依赖关系？**
- **一种办法：在相同的流水线中交叉执行来自不同线程的指令**

*Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe*

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5
T1: LW r5, 12(r1)

A conventional processor compared with a multithreaded processor.

# CDC 6600 Peripheral Processors (Cray, 1964)

- **容忍延时：**
  - 当一个线程遇到长延时操作时，处理器可以从另一个线程执行有用的操作
- **CDC 6600 I/O处理器**
  - 第一个支持多线程模式的硬件系统
  - 10个I/O线程交替执行
  - 10 个“virtual” I/O 处理器
  - 流水线控制采用固定交叉方式
  - 流水线每段100ns
  - 每个虚拟处理器每隔1000ns执行一条指令
  - 累加器型ISA有利于保存处理器状态

# CDC 6600 Architecture

FIGURE 2

J. E. Thornton， Design of a Computer_ The Control Data 6600 (1970, Scott, Foresman and Company)

MAJOR CYCLE：1000 ns
Storage 读和写
MINOR CYCLE： 100ns
ALU运算, 以及传输一个字的时间 （字长12bit）

例如：Load (d)； A <- mem[d]
Add X；

FIGURE 83 Peripheral processors.

# Simple Multithreaded Pipeline

- 必须传递线程选择信号以保证各流水段读写的正确性
- 从软件（包括OS）的角度看 好像存在多个CPU（针对每个线程，CPU似乎运行的慢一些)

# Multithreading Costs

- **每个线程需要拥有自己的用户态信息（user state）：包括PC、GPRs**

- **需要自己的系统态信息（system state）**
  - 虚拟存储的页表基地址寄存器（Virtual-memory page-table-base register）
  - 异常处理寄存器（Exception-handling registers）

- **其他开销:**
  - 需要处理由于线程竞争导致的Cache/TLB冲突 或需要更大的cache/TLB 容量
  - 更多的OS调度开销

# Thread Scheduling Policies

- **固定交叉模式 (CDC 6600 PPUs, 1964)**
  - 针对N个线程，每个线程每隔N个周期执行一条指令
  - 如果流水线的某一时隙(slot)其对应线程未就绪，插入pipeline bubble

- **软件控制的交叉模式 (TI ASC PPUs, 1971)**
  - OS 为N个线程分配流水线的S个pipeline slots
  - 硬件针对S个slots采用固定交叉模式执行相应的线程

- **硬件控制的线程调度 (HEP, 1982)**
  - 硬件跟踪哪些线程处于ready状态
  - 根据优先级方案选择线程执行

# Superscalar Machine Efficiency

*Completely idle cycle*
*(vertical waste)*

*Partially filled cycle, i.e., IPC < 4*
*(horizontal waste)*

问题：如何充分利用资源，提高并行度？

-- multithreading

# Chip Multiprocessing (CMP)

- **分成多个处理器后的效果？**
  - 同一时钟周期可以运行不同线程的指令
  - 单个处理器核同一时钟周期无法执行不同线程的指令
  - 从单个核看，没有减少水平浪费和垂直浪费。
  - 由于发射宽度在核间进行了静态分配，导致水平和垂直方向浪费减少
- **例如4核2发射的CMP**
  - 发射宽度为8，即同时可以发射8条指令
  - 当1个线程stall，那么垂直浪费最多为2条指令

# Vertical(Coarse-Grain) Multithreading

- **粗粒度多线程方式**
  - 当线程运行时存在较长时间延时时切换到另一线程
    - 例如：Cache失效时
    - 等待同步结束
  - 同一线程指令间的较短延时不切换
- **如果基于粗粒度的时钟周期交叉运行模式，结果怎样？**
  - 仍然存在垂直方向的浪费
  - 减少水平方向的浪费？

**A *Coarse-Grain Multithreaded* architecture has the ability to switch between threads with only a short hardware context switch latency < 10 cycles**

# Coarse-Grain Multithreading

**Figure 3.1:** The MIPS R3000 Pipeline. This pipeline does not support multithreading.

**Figure 3.2:** The MIPS R3000 Pipeline with support for Coarse-Grain Multithreading.

# CGMT Context swap

**Figure 3.3:** The operation of a CGMT context swap in our reference design.

# Fine-Grain Multithreading

- **细粒度多线程**
  - 多个线程的指令交叉执行
- **如果基于细粒度的时钟周期交叉运行模式，结果怎样？**
  - 减少甚至消除垂直方向的浪费
  - 仍然存在水平方向的浪费

# Fine-Grain Multithreading

**Figure 4.1:** The MIPS R3000 Pipeline with support for Fine-Grain Multithreading. Although our target design would support 8 threads to maximize throughput, we show a 4-thread implementation for simplicity.

# Ideal Superscalar Multithreading [Tullsen, Eggers, Levy, UW, 1995]

- **采用多线程交叉模式使用多个issue slots**
- **Simultaneous Multithreading (SMT)**

# Simultaneous Multithreading (SMT) for OoO Superscalars

- **“vertical”多线程：即某一时段每条流水线上运行一个线程**
- **SMT 使用OoOSuperscalar细粒度控制技术在相同时钟周期运行多个线程的指令，以更好的利用系统资源**
  - Alpha AXP 21464
  - Intel Pentium 4， Intel Nehalem i7
  - IBM Power5

# O-o-O Simultaneous Multithreading
[Tullsen, Eggers, Emer, Levy, Stamm, Lo, DEC/UW, 1996]

- 增加多上下文切换以及取指引擎可以从多个线程取指令，并可同时发射
- 使用OoO（Out-of-Order) superscalar处理器的发射宽度，从发射队列中选择指令发射，这些指令可来源于多个线程
- OoO 指令窗口已经具备从多个线程调度指令的绝大多数电路
- 任何单线程程序可以使用整个系统资源

**Figure 5.1:** The pipeline of the MIPS R10000 processor, as seen by a load instruction.

**Figure 5.2:** The MIPS R10000 pipeline with support added for simultaneous multithreading.

# SMT adaptation to parallelism type

For regions with high thread level parallelism (TLP) entire machine width is shared by all threads

For regions with low thread level parallelism (TLP) entire machine width is available for instruction level parallelism (ILP)

# Performance

**Figure 2.7:** The performance of fine-grain and simultaneous multithreading models on a wide superscalar processor, as the number of threads increases. Reprinted from Tullsen et al. [1995].

# Summary: Multithreaded Categories

Various approaches to resource sharing and context switching

| MT Approach | Resources Shared between Threads | Context Switch Mechanism |
|---|---|---|
| None | Everything | Explicit operating system context switch |
| Fine-grained | Everything but register file and control logic/state | Switch every cycle |
| Coarse-grained | Everything but I-fetch buffers, register file, and control logic/state | Switch on pipeline stall |
| SMT | Everything but instruction fetch buffers, return address stack, architected register file, control logic/state, reorder buffer, store queue, etc. | All contexts concurrently active; no switching |
| CMP | Secondary cache, system interconnect | All contexts concurrently active; no switching |

John Paul Shen，Mikko H. Lipasti；Modern Processor Design：Fundamentals of Superscalar Processors ； 2013, Waveland Press

Four possible alternatives are: chip multiprocessing (a), which statically partitions execution bandwidth; fine-grained multiprocessing (b), which executes a different thread in alternate cycles; coarse-grained multithreading (c), which switches threads to tolerate long-latency events; and simultaneous multithreading (d), which intermingles instructions from multiple threads. The CMP and FGMT approaches partition execution resources statically, either with a spatial partition by assigning a fixed number of resources to each processor, or with a temporal partition that time-multiplexes multiple threads onto the same set of resources. The CGMT and SMT approaches allow dynamic partitioning, with either a per-cycle temporal partition in the CGMT approach, or a per-functional unit partition in the SMT approach. The greatest flexibility and highest resource utilization and instruction throughput are achieved by the SMT approach.

**Figure 11.11**

Running Multiple Threads on One Chip.

*Source:* Tullsen et al., 1996.

# Acknowledgements


- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - John Hennessy (Standford)and David Patterson (UCB)
  - Chenxi Zhang (Tongji)
  - Muhamed Mudawar (KFUPM)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**

# Denelcor HEP (Burton Smith, 1982)

第一个在主处理器中使用硬件多线程技术的商用机器

- 120 threads per processor
- 10 MHz clock rate
- Up to 8 processors
- precursor to Tera MTA (Multithreaded Architecture)

# Tera MTA (1990-)

- Up to 256 processors
- Up to 128 active threads per processor
- Processors and memory modules populate a sparse 3D torus interconnection fabric
- Flat, shared main memory
  - No data cache
  - Sustains one main memory access per cycle per processor
- GaAs logic in prototype, 1KW/processor @ 260MHz
  - Second version CMOS, MTA-2, 50W/processor
  - New version, XMT, fits into AMD Opteron socket, runs at 500MHz

# MTA Pipeline

- Every cycle, one VLIW instruction from one active thread is launched into pipeline
- Instruction pipeline is 21 cycles long
- Memory operations incur ~150 cycles of latency

Assuming a single thread issues one instruction every 21 cycles, and clock rate is 260 MHz…

*What is single-thread performance?*

Effective single-thread issue rate is 260/21 = 12.4 MIPS

# Coarse-Grain Multithreading

- **Tera MTA 适用于大规模数据集并且数据本地性较低的应用**
  - 没有数据Cache
  - 需要大量的并行线程以隐藏存储器访问延迟
- **其他数据本地性较好的应用**
  - 如果Cache命中，流水线上的停顿较少
  - 仅需要添加较少的线程来隐藏偶尔Cache失效导致的延迟
  - Cache失效时，将该线程换出
- **当某一线程的执行存在长延迟操作时，选择另一线程执行，并切换线程上下文**

# MIT Alewife (1990)

- Modified SPARC chips
  - register windows hold different thread contexts
- Up to four threads per node
- Thread switch on local cache miss

# IBM PowerPC RS64-IV (2000)

- Commercial coarse-grain multithreading CPU
- Based on PowerPC with quad-issue in-order five-stage pipeline
- Each physical CPU supports two virtual CPUs
- On L2 cache miss, pipeline is flushed and execution switches to second thread
  - short pipeline minimizes flush penalty (4 cycles), small compared to memory access latency
  - flush pipeline to simplify exception handling

# Oracle/Sun Niagara processors

- **Target is datacenters running web servers and databases, with many concurrent requests**
- **Provide multiple simple cores each with multiple hardware threads, reduced energy/operation though much lower single thread performance**

- **Niagara-1 [2004], 8 cores, 4 threads/core**
- **Niagara-2 [2007], 8 cores, 8 threads/core**
- **Niagara-3 [2009], 16 cores, 8 threads/core**
- **T4 [2011], 8 cores, 8 threads/core**
- **T5 [2012], 16 cores, 8 threads/core**

# Oracle/Sun Niagara-3, “Rainbow Falls” 2009

# IBM Power 4

**Single-threaded predecessor to Power 5. 8 execution units in out-of-order engine, each may issue an instruction each cycle.**

# Power 4

# Power 5

2 commits (architected register sets)

2 fetch (PC),

2 initial decodes

# Power 5 data flow …

Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck

# Changes in Power 5 to support SMT

- **Increased associativity of L1 instruction cache and the instruction address translation buffers**
- **Added per-thread load and store queues**
- **Increased size of the L2 (1.92 vs. 1.44 MB) and L3 caches**
- **Added separate instruction prefetch and buffering per thread**
- **Increased the number of virtual registers from 152 to 240**
- **Increased the size of several issue queues**
- **The Power5 core is about 24% larger than the Power4 core because of the addition of SMT support**

# Pentium-4 Hyperthreading (2002)

- **First commercial SMT design (2-way SMT)**
  - Hyperthreading == SMT
- **Logical processors share nearly all resources of the physical processor**
  - Caches, execution units, branch predictors
- **Die area overhead of hyperthreading ~ 5%**
- **When one logical processor is stalled, the other can make progress**
  - No logical processor can use all entries in queues when two threads are active
- **Processor running only one active software thread runs at approximately same speed with or without hyperthreading**
- **Hyperthreading dropped on OoO P6 based followons to Pentium-4 (Pentium-M, Core Duo, Core 2 Duo), until revived with Nehalem generation machines in 2008.**
- **Intel Atom (in-order x86 core) has two-way vertical multithreading**

# Initial Performance of SMT

- **Pentium 4 Extreme SMT yields 1.01 speedup for SPECint_rate benchmark and 1.07 for SPECfp_rate**
  - Pentium 4 is dual threaded SMT
  - SPECRate requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark
- **Running on Pentium 4 each of 26 SPEC benchmarks paired with every other (262 runs) speed-ups from 0.90 to 1.58; average was 1.20**
- **Power 5, 8-processor server 1.23 faster for SPECint_rate with SMT, 1.16 faster for SPECfp_rate**
- **Power 5 running 2 copies of each app speedup between 0.89 and 1.41**
  - Most gained some
  - Fl.Pt. apps had most cache conflicts and least gains

# Upper Limit to ILP: Ideal Machine

# More Realistic HW: Branch Impact

# More Realistic HW: Register Impact (rename regs)

# More Realistic HW: Alias Impact

# Realistic HW for '9X: Window Impact

# 计算机体系结构

Topic: Data Level Parallelism

https://blog.csdn.net/bpssy/article/details/16965377

# Review: Multithreaded Categories

# Data-Level Parallelism in Vector, SIMD, and GPU Architectures

- **数据级并行的研究动机**
  - 传统指令级并行技术的问题
  - SIMD结构的优势
  - 数据级并行的种类
- **向量体系结构**
  - 向量处理模型
  - 起源-超级计算机
  - 基本特性及结构
  - 性能评估及优化
- **面向多媒体应用的SIMD指令集扩展**
- **GPU**
  - GPU简介
  - GPU的编程模型
  - GPU的存储系统

# 动机：传统指令级并行技术的问题

- **提高性能的传统方法（挖掘ILP）的主要缺陷:**
  - 程序内在的并行性
  - 提高流水线的时钟频率: 提高时钟频率，有时导致CPI随着增加 (branches, other hazards)
  - 指令预取和译码: 有时在每个时钟周期很难预取和译码多条指令
  - 提高Cache命中率 : 在有些计算量较大的应用中（科学计算）需要大量的数据，其局部性较差，有些程序处理的是连续的媒体流(multimedia),其局部性也较差。

# 动机：DLP的兴起

- 应用需求和技术发展推动着体系结构的发展
- 图形、机器视觉、语音识别、机器学习等新的应用均需要大量的数值计算，其算法通常具有数据并行特征
- SIMD-based 结构 (vector-SIMD, subword-SIMD, SIMT/GPUs) 是执行这些算法的最有效途径

# 系统结构的Flynn分类 (1966)

- **SISD: Single instruction stream, single data stream**
  - 单处理器模式
- **SIMD: Single instruction stream, multiple data streams**
  - 相同的指令作用在不同的数据
  - 可用来挖掘数据级并行(Data Level Parallelism)
  - 如：Vector processors, SIMD instructions, and Graphics processing units
- **MISD: Multiple instruction streams, single data stream**
  - No commercial implementation
- **MIMD: Multiple instruction streams, multiple data streams**
  - 通用性最强的一种结构，可用来挖掘线程级并行、数据级并行.....
  - 组织方式可以是松耦合方式也可以是紧耦合方式

# 动机：SIMD结构的优势

- **SIMD 结构可有效地挖掘数据级并行:**
  - 基于矩阵运算的科学计算
  - 图像和声音处理

- **SIMD比MIMD更节能**
  - 针对每组数据操作仅需要取指一次
  - SIMD对PMD( personal mobile devices)更具吸引力

- **SIMD 允许程序员继续以串行模式思维**

# SIMD 结构的种类

- **向量体系结构**
- **多媒体SIMD指令集 扩展**
- **Graphics Processor Units (GPUs)**

- **For x86 processors:**
  - 每年增加2cores/chip
  - SIMD 宽度每4年翻一番
  - SIMD潜在加速比是MIMD的2倍

**Figure 4.1 Potential speedup via parallelism from MIMD, SIMD, and both MIMD and SIMD over time for x86 computers.** This figure assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD will double every four years.

# 向量处理模型

- **向量处理机具有更高层次的操作，一条向量指令可以处理N个或N对操作数（处理对象是向量）**

# 起源：Supercomputers

- **Supercomputer的定义:**
  - 对于给定任务而言世界上最快的机器
  - 任何造价超过3千万美元的机器
  - 计算能力达到每秒万亿次的机器
- **由Seymour Cray设计的机器**
- **CDC6600 (Cray, 1964) 被认为是第一台超级计算机**

# CDC 6600 *Seymour Cray, 1963*

- **A fast pipelined machine with 60-bit words**
  - 128 Kword main memory capacity, 32 banks
- **Ten functional units (parallel, unpipelined)**
  - Floating Point: adder, 2 multipliers, divider
  - Integer: adder, 2 incrementers, …
- **Hardwired control (no microcoding)**
- ***Scoreboard* for dynamic scheduling of instructions**
- **Ten Peripheral Processors for Input/Output**
  - a fast multi-threaded 12-bit integer ALU
- **Very fast clock, 10 MHz (FP add in 4 clocks)**
- **>400,000 transistors, 750 sq. ft., 5 tons, 150 kW, novel freon-based technology for cooling**
- **Fastest machine in world for 5 years (until 7600)**
  - over 100 sold ($7-10M each)

# IBM Memo on CDC6600

**Thomas Watson Jr., IBM CEO, August 1963:**

"Last week, Control Data … announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers… Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer."

**To which Cray replied: "It seems like Mr. Watson has answered his own question."**

# Supercomputer Applications

- **典型应用领域**
  - 军事研究领域（核武器研制、密码学）
  - 科学研究
  - 天气预报
  - 石油勘探
  - 工业设计 (car crash simulation)
  - 生物信息学
  - 密码学
- **均涉及大量的数据集处理**
- **70-80年代Supercomputer = Vector Machine**

# 向量处理机的基本特性

- **基本思想：两个向量的对应分量进行运算，产生一个结果向量。**
- **简单的一条向量指令包含了多个操作=> fewer instruction fetches**
- **每一结果独立于前面的结果**
  - 长流水线，编译器保证操作间没有相关性
  - 硬件仅需检测两条向量指令间的相关性
  - 较高的时钟频率
- **向量指令以已知的模式访问存储器**
  - 可有效发挥多体交叉存储器的优势
  - 可通过重叠减少存储器操作的延时 64 elements
  - 不需要数据Cache! (仅使用指令cache)
- **在流水线控制中减少了控制相关**

# 向量处理机的基本结构

- ***memory-memory vector processors:*** **所有的向量操作是存储器到存储器**
- ***vector-register processors:*** **除了load 和store操作外，所有的操作是向量寄存器与向量寄存器间的操作**
  - 向量机的Load/Store结构
  - 1980年以后的所有的向量处理机都是这种结构: Cray, Convex, Fujitsu, Hitachi, NEC
  - 我们也主要针对这种结构

# Vector Memory-Memory versus Vector Register Machines

- **存储器-存储器型向量机所有指令操作的操作数来源于存储器**
- **第一台向量机 CDC Star-100 ( '73) and TI ASC ( '71), 是存储器-存储器型机器**
- **Cray-1 (' 76) 是第一台寄存器型向量机**

Example Source Code

```
for (i=0; i<N; i++)
{
  C[i] = A[i] + B[i];
  D[i] = A[i] - B[i];
}
```

Vector Memory-Memory Code

```
ADDV C, A, B
SUBV D, A, B
```

Vector Register Code

```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

# Vector Memory-Memory vs. Vector Register Machines

- **存储器-存储器型向量机 (VMMA) 需要更高的存储器带宽**
  - All operands must be read in and out of memory
- **VMMA结构使得多个向量操作重叠执行较困难**
  - Must check dependencies on memory addresses
- **VMMA启动时间更长**
  - CDC Star-100 在向量元素小于100时，标量代码的性能高于向量化代码
- **CDC Cray-1后续的机器 (Cyber-205, ETA-10) 都是寄存器型向量机**

# Vector Supercomputers

- **Cray-1的变体（1976）：**
- **Scalar Unit：Load/Store Architecture**
- **Vector Extension**
  - Vector Registers
  - Vector Instructions
- **Implementation**
  - 硬布线逻辑控制
  - 高效流水化的功能部件
  - 多体交叉存储系统
  - 无Data Cache
  - 不支持 Virtual Memory

# Vector Instruction Set Advantages

- **格式紧凑**
  - 一条指令包含N个操作
- **表达能力强，一条指令能告诉硬件:**
  - N个操作之间无相关性访
  - 使用同样的功能部件
  - 访问不相交的寄存器
  - 与前面的操作以相同模式访问寄存器
  - 问存储器中的连续块 (unit-stride load/store)
  - 以已知的模式访问存储器 (strided load/store)
- **可扩展性好**
  - 可以在多个并行的流水线上运行同样的代码 (lanes)

# Vector Instructions

| | Instr. | Operands | Operation Comment |
|---|---|---|---|
| ADDV | V1,V2,V3 | V1=V2+V3 | vector + vector |
| ADDSV | V1,F0,V2 | V1=F0+V2 | scalar + vector |
| MULTV | V1,V2,V3 | V1=V2xV3 | vector x vector |
| MULSV | V1,F0,V2 | V1=F0xV2 | scalar x vector |
| LV | V1,R1 | V1=M[R1..R1+63] | load, stride=1 |
| LVWS | V1,R1,R2 | V1=M[R1..R1+63*R2] | load, stride=R2 |
| LVI | V1,R1,V2 | V1=M[R1+V2i,i=0..63] | indir.("gather") |
| CeqV | VM,V1,V2 | VMASKi = (V1i=V2i)? | comp. setmask |
| MOV | VLR,R1 | Vec. Len. Reg. = R1 | set vector length |
| MOV | VM,R1 | Vec. Mask = R1 | set vector mask |

# 向量处理机的基本组成单元

- ***Vector Register*: 固定长度的一块区域，存放单个向量**
  - 至少2个读端口和一个写端口（一般最少16个读端口，8个写端口）
  - 典型的有8-32 向量寄存器，每个寄存器存放64到128个64位元素
- ***Vector Functional Units (FUs)*: 全流水化的，每一个clock启动一个新的操作**
  - 一般4到8个FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical，shift；可能有些重复设置的部件
- ***Vector Load-Store Units (LSUs)*: 全流水化地load 或store一个向量，可能会配置多个LSU部件**
- ***Scalar registers*: 存放单个元素用于标量处理或存储地址**
- **用交叉开关连接(Cross-bar) FUs , LSUs, registers**

Main memory
Vector load-store
FP add/subtract
FP multiply
FP divide
Integer
Logical
Vector registers
Scalar registers

# Vector Arithmetic Execution

- **使用较深的流水线(=> fast clock) 执行向量元素的操作**
- **由于向量元素相互独立，简化了深度流水线的控制 (=> no hazards!)**

*Six stage multiply pipeline*

V3 <- v1 * v2

# Vector Unit Structure

# Vector Instruction Execution

ADDV C,A,B

*使用一条流水化的功能部件执行*

*使用4条流水化的功能部件执行*

# Interleaved Vector Memory System

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- *Bank busy time*: Time before bank ready to accept next request

# T0 Vector Microprocessor (UCB/ICSI, 1995)

*Vector register elements striped over lanes*

*Lane*

# Vector Instruction Parallelism

- **多条向量指令可重叠执行(链接技术)**
  - 例如：每个向量 32 个元素，8 lanes（车道）

Complete 24 operations/cycle while issuing 1 short instruction/cycle

# Vector Execution Time

- ***Time*** **= f(vector length, data dependencies, struct. hazards)**
- ***Initiation rate*****: 功能部件消耗向量元素的速率**
- ***Convoy:*** **可在同一时钟周期开始执行的指令集合 (no structural or data hazards)**
- ***Chime:*** **执行一个*convoy*所花费的大致时间（approx. time）**
- ***m convoys take m chimes*****;**
  - 如果每个向量长度为*n*, 那么*m*个*convoys* 所花费的时间是*m个chimes*
  - 每个*chime*所花费的时间是*n个clocks*，该程序所花费的总时间大约为*m x n clock cycles* (忽略额外开销; 当向量长度较长时这种近似是合理的）

```
1:  LV     V1,Rx      ;load vector X
2:  MULV   V2,F0,V1   ;vector-scalar mult.
    LV     V3,Ry      ;load vector Y
3:  ADDV   V4,V2,V3   ;add
4:  SV     Ry,V4      ;store the result
```

**4 convoys, 1 lane, VL=64**
**=> 4 x 64 = 256 clocks**
**(or 4 clocks per result)**

# Vector Startup

- **向量启动时间由两部分构成**
  - 功能部件延时：一个操作通过功能部件的时间
  - 截止时间或恢复时间（dead time or recovery time ）：运行下一条向量指令的间隔时间

# VMIPS Start-up Time

**Start-up time: FU 部件流水线的深度**

| Operation | Start-up penalty (from CRAY-1) |
|---|---|
| Vector load/store | 12 |
| Vector multiply | 7 |
| Vector add | 6 |

Assume convoys don't overlap; vector length = n

| *Convoy* | *Start* | *1st result* | *last result* | |
|---|---|---|---|---|
| 1. LV | 0 | 12 | 11+n | (12+n-1) |
| 2. MULV, LV | 12+n | 12+n+7 | 18+2n | *Multiply startup* |
| | 12+n | 12+n+12 | 23+2n | *Load start-up* |
| 3. ADDV | 24+2n | 24+2n+6 | 29+3n | *Wait convoy 2* |
| 4. SV | 30+3n | 30+3n+12 | 41+4n | *Wait convoy 3* |

# Vector Length

- 当向量的长度不是64时（假设向量寄存器的长度是64）怎么办？
- vector-length register (VLR) 控制特定向量操作的长度, 包括向量的load/store. (当然一次操作的向量的长度不能 > 向量寄存器的长度) 例如：

```
      do 10    i = 1, n
10    Y(i) = a * X(i) + Y(i)
```

n的值只有在运行时才能知道

n > Max. Vector Length (MVL)怎么办？

# Strip Mining（分段开采）

- 假设Vector Length > Max. Vector Length (MVL)?
- Strip mining: 产生新的代码，使得每个向量操作的元素数 ≤ MVL
- 第一次循环做最小片(n mod MVL), 以后按VL = MVL操作

```
low = 1
VL = (n mod MVL)  /*find the odd size piece*/
do 1  j = 0, (n / MVL)  /*outer loop*/
    do 10 i = low, low+VL-1  /*runs for length VL*/
            Y(i) = a*X(i) + Y(i)  /*main operation*/
10  continue
    low = low+VL  /*start of next vector*/
    VL = MVL  /*reset the length to max*/
1   continue
```

# Strip Mining的向量执行时间计算

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

试计算A=B×s，其中A,B为长度为200的向量（每个向量元素占8个字节），s是一个标量。向量寄存器长度为64。各功能部件的启动时间如前所述，求总的执行时间，($T_{loop}$ = 15)

```
        ADDI R2,R0,#1600      ;total # bytes in vector
        ADD R2,R2,Ra          ;address of the end of A vector
        ADDI R1,R0,#8         ;loads length of 1st segment
        MOVI2S  VLR,R1        ;load vector length in VLR
        ADDI R1,R0,#64        ;length in bytes of 1st segment
        ADDI R3,R0,#64        ;vector length of other segments
Loop:   LV  V1,Rb             ;load B
        MULSV V2,V1,Fs        ;vector * scalar
        SV Ra,V2              ;store A
        ADD Ra,Ra,R1          ;address of next segment of A
        ADD  Rb,Rb,R1         ;address of next segment of B
        ADDI R1,R0,#512       ;load byte offset next segment
        MOVI2S VLR,R3         ;set length to 64 elements
        SUB R4,R2,Ra          ;at the end of A?
        BNEZ R4,Loop          ;if not, go back
```

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

Tstart = 12 + 7 + 12 = 31

T200 = 660+4*31 = 784

每一元素的执行时间 = 784/200 = 3.9

**Figure G.9** The total execution time per element and the total overhead time per element versus the vector length for the example on page G-19. For short vectors the total start-up time is more than one-half of the total time, while for long vectors it reduces to about one-third of the total time. The sudden jumps occur when the vector length crosses a multiple of 64, forcing another iteration of the strip-mining code and execution of a set of vector instructions. These operations increase $T_n$ by $T_{loop} + T_{start}$.

# Common Vector Metrics

- ***$R_\infty$*: 当向量长度为无穷大时的向量流水线的最大性能。常在评价峰值性能时使用，单位为MFLOPS**
  - 实际问题是向量长度不会无穷大，start-up的开销还是比较大的
  - $R_n$ 表示向量长度为n时的向量流水线的性能
- ***$N_{1/2}$*: 达到R∞ 一半的值所需的向量长度，是评价向量流水线start-up 时间对性能的影响。**
- ***$N_V$*：向量流水线方式的工作速度优于标量串行方式工作时所需的向量长度临界值。**
  - 该参数既衡量建立时间，也衡量标量、向量速度比对性能的影响

# Example Vector Machines

| Machine | Year | Clock(MHZ) | Regs | Elements | Fus | LSUs |
|---|---|---|---|---|---|---|
| Cray 1 | 1976 | 80 | 8 | 64 | 6 | 1 |
| Cray XMP | 1983 | 120 | 8 | 64 | 8 | 2L, 1S |
| Cray YMP | 1988 | 166 | 8 | 64 | 8 | 2L, 1S |
| Cray C-90 | 1991 | 240 | 8 | 128 | 8 | 4 |
| Cray T-90 | 1996 | 455 | 8 | 128 | 8 | 4 |
| Conv. C-1 | 1984 | 10 | 8 | 128 | 4 | 1 |
| Conv. C-4 | 1994 | 133 | 16 | 128 | 3 | 1 |
| Fuj. VP200 | 1982 | 133 | 8-256 | 32-1024 | 3 | 2 |
| Fuj. VP300 | 1996 | 100 | 8-256 | 32-1024 | 3 | 2 |
| NEC SX/2 | 1984 | 160 | 8+8K | 256+var | 16 | 8 |
| NEC SX/3 | 1995 | 400 | 8+8K | 256+var | 16 | 8 |

# A Modern Vector Super: NEC SX-9 (2008)

- 65nm CMOS technology
- Vector unit (3.2 GHz)
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 64-bit functional units: 2 multiply, 2 add, 1 divide/sqrt, 1 logical, 1 mask unit
  - 8 lanes (32+ FLOPS/cycle, 100+ GFLOPS peak per CPU)
  - 1 load or store unit (8 x 8-byte accesses/cycle)
- Scalar unit (1.6 GHz)
  - 4-way superscalar with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache

- Memory system provides 256GB/s DRAM bandwidth per CPU
- Up to 16 CPUs and up to 1TB DRAM form shared-memory *node*
  - total of 4TB/s bandwidth to shared DRAM memory
- Up to 512 nodes connected via 128GB/s network links (message passing between nodes)

Picture from NEC article "A hardware overview of SX-6 and SX-7 supercomputer"

# Vector Linpack Performance (MFLOPS)

**Matrix Inverse (gaussian elimination)**

| Machine | Year | Clock(Mhz) | 100x100 | 1kx1k | Peak(Procs) |
|---|---|---|---|---|---|
| Cray 1 | 1976 | 80 | 12 | 110 | 160(1) |
| Cray XMP | 1983 | 120 | 121 | 218 | 940(4) |
| Cray YMP | 1988 | 166 | 150 | 307 | 2,667(8) |
| Cray C-90 | 1991 | 240 | 387 | 902 | 15,238(16) |
| Cray T-90 | 1996 | 455 | 705 | 1603 | 57,600(32) |
| Conv. C-1 | 1984 | 10 | 3 | -- | 20(1) |
| Conv. C-4 | 1994 | 136 | 160 | 2531 | 3,240(4) |
| Fuj. VP200 | 1982 | 133 | 18 | 422 | 533(1) |
| NEC SX/2 | 1984 | 166 | 43 | 885 | 1,300(1) |
| NEC SX/3 | 1995 | 400 | 368 | 2757 | 25,600(4) |

# Interleaved Vector Memory System

- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency
  - *Bank busy time*: Time before bank ready to accept next request
  - *If stride = 1 & consecutive elements interleaved across banks & number of banks >= bank latency, then can sustain 1 element/cycle throughput*

**Example(AppF F-15)** Suppose we want to fetch a vector of 64 elements starting at byte address 136,and a memory access takes 6 clocks. Bank busy time is 2 clocks. How many memory banks must we have to support one fetch per clock cycle? With what addresses are the banks accessed? When will the various elements arrive at the CPU?

| | Bank | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Cycle no. | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | | 136 | | | | | | |
| 1 | | busy | 144 | | | | | |
| 2 | | busy | busy | 152 | | | | |
| 3 | | busy | busy | busy | 160 | | | |
| 4 | | busy | busy | busy | busy | 168 | | |
| 5 | | busy | busy | busy | busy | busy | 176 | |
| 6 | | | busy | busy | busy | busy | busy | 184 |
| 7 | 192 | | | busy | busy | busy | busy | busy |
| 8 | busy | 200 | | | busy | busy | busy | busy |
| 9 | busy | busy | 208 | | | busy | busy | busy |
| 10 | busy | busy | busy | 216 | | | busy | busy |
| 11 | busy | busy | busy | busy | 224 | | | busy |
| 12 | busy | busy | busy | busy | busy | 232 | | |
| 13 | | busy | busy | busy | busy | busy | 240 | |
| 14 | | | busy | busy | busy | busy | busy | 248 |
| 15 | 256 | | | busy | busy | busy | busy | busy |
| 16 | busy | 264 | | | busy | busy | busy | busy |

**Figure F.7 Memory addresses (in bytes) by bank number and time slot at which access begins.** Each memory bank latches the element address at the start of an access and is then busy for 6 clock cycles before returning a value to the CPU. Note that the CPU cannot keep all eight banks busy all the time because it is limited to supplying one new address and receiving one data item each cycle.

# Vector Stride

- 假设处理顺序相邻的元素在存储器中不顺序存储。例如

```
  do 10 i = 1,100
    do 10 j = 1,100
       A(i,j) = 0.0
       do 10 k = 1,100
10        A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

- B 或 C 的两次访问不会相邻 (相隔800 bytes)
- *stride*: 向量中相邻元素间的距离
  => LVWS (load vector with stride) instruction
- Strides => 会导致体冲突
  (e.g., stride = 32 and 16 banks)

# Memory operations

- **Load/store 操作成组地在寄存器和存储器之间移动数据**
- **三类寻址方式**
  - Unit stride（单步长)
    - Fastest
  - Non-unit (constant) stride（常数步长)
  - Indexed (gather-scatter) (间接寻址)
    - 等价于寄存器间接寻址方式
    - 对稀疏矩阵有效
    - 用于向量化操作的指令增多

# DAXPY (Y = a × X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector

```
LD      F0,a          ;load scalar a
LV      V1,Rx         ;load vector X
MULTS   V2,F0,V1      ;vector-scalar mult.
LV      V3,Ry         ;load vector Y
ADDV    V4,V2,V3      ;add
SV      Ry,V4         ;store the result
```

```
      LD      F0,a
      ADDI    R4,Rx,#512 ;last address to load
loop: LD      F2, 0(Rx)  ;load X(i)
      MULTD   F2,F0,F2   ;a*X(i)
      LD      F4, 0(Ry)  ;load Y(i)
      ADDD    F4,F2, F4  ;a*X(i) + Y(i)
      SD      F4,0(Ry)   ;store into Y(i)
      ADDI    Rx,Rx,#8   ;increment index to X
      ADDI    Ry,Ry,#8   ;increment index to Y
      SUB     R20,R4,Rx  ;compute bound
      BNZ     R20,loop   ;check if done
```

578 (2+9*64) vs.
321 (1+5*64) ops (1.8X)

578 (2+9*64) vs.
6 instructions (96X)

64 operation vectors + no loop overhead

also 64X fewer pipeline hazards

# Vector Opt#1: Vector Chaining

- 寄存器定向路径的向量机版本
- 首次在Cray-1上使用

```
LV    v1
MULV  v3,v1,v2
ADDV  v5,v3, v4
```

# Vector Chaining Advantage

- 不采用链接技术，必须处理完前一条指令的最后一个元素，才能启动下一条相关的指令

- 采用链接技术，前一条指令的第一个结果出来后，就可以启动下一条相关指令的执行

# Vector Instruction Parallelism

- **多条向量指令可重叠执行(链接技术)**
  - 例如：每个向量 32 个元素，8 lanes（车道）

Complete 24 operations/cycle while issuing 1 short instruction/cycle

# Vector Opt #2: Conditional Execution

- **Suppose:**

```
   do 100 i = 1, 64
        if (A(i) .ne. 0) then
            A(i) = A(i) - B(i)
        endif
100 continue
```

- ***vector-mask control*** **使用长度为MVL的布尔向量控制向量指令的执行**
- ***当vector-mask register*** **使能时，向量指令操作仅对 vector-mask register中对应位为1的分量起作用**

# Masked Vector Instructions

## Simple Implementation

- execute all N operations, turn off result writeback according to mask

## Density-Time Implementation

- scan mask vector and only execute elements with non-zero masks

```
LV   V1,Ra            ;  load vector A into V1
LV V2,Rb              ;  load vector B
L.D F0,#0             ; load FP zero into F0
SNEVS.D V1,F0         ;sets VM(i) to 1 if V1(i)!=F0
SUBV.D V1,V1,V2       ;subtract under vector mask
CVM                   ;set the vector mask to all 1s
SV Ra,V1              ;store the result in A
```

- **使用vector-mask寄存器的缺陷**
  - 简单实现时，条件不满足时向量指令仍然需要花费时间
  - 有些向量处理器带条件的向量执行仅控制向目标寄存器的写操作，可能会有除法错。

# Vector Opt #3: Sparse Matrices

- **Suppose:**

```
        do   100 i = 1,n
100     A(K(i)) = A(K(i)) + C(M(i))
```

- ***gather*** (LVI) **operation 使用*index vector*中给出的偏移再加基址来读取 => a nonsparse vector in a vector register**
- **这些元素以密集的方式操作完成后，再使用同样的index vector存储到稀疏矩阵的对应位置**
- **这些操作编译时可能无法完成。主要原因：编译器无法预知Ki以及是否有数据相关**
- **使用CVI 设置步长（ index 0, 1xm, 2xm, ..., 63xm ）**

# Sparse Matrix Example

- Cache (1993) vs. Vector (1988)

| | IBM RS6000 | Cray YMP |
|---|---|---|
| Clock | 72 MHz | 167 MHz |
| Cache | 256 KB | 0.25 KB |
| Linpack | 140 MFLOPS | 160 (1.1) |
| Sparse Matrix (Cholesky Blocked ) | 17 MFLOPS | 125 (7.3) |

- Cache: 1 address per cache block (32B to 64B)
- Vector: 1 address per element (4B)

# Automatic Code Vectorization

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

*Scalar Sequential Code*

*Vectorized Code*

向量化是指在编译期间对操作重定序⇒ 需要进行大量的循环相关分析

# Vector/SIMD Processing Summary

- **Vector/SIMD 机器适合挖掘数据级并行**
  - 同样的操作作用于不同的数据元素
  - 向量内的元素操作独立，可有效提高性能，简化设计
- **性能的提升受限于代码的向量化**
  - 标量操作限制了向量机的性能
  - Amdahl's Law
- **很多ISA包含SIMD操作指令**
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

# Array vs. Vector Processors

**Array processor：**又称为并行处理机、SIMD处理器。其核心是一个由**多个处理单元**构成的**阵列**，用**单一的控制部件**来控制多个处理单元对**各自的数据**进行**相同的运算**和操作。

# SIMD Array Processing vs. VLIW

- VLIW: 多个独立的操作由编译器封装在一起

# SIMD Array Processing vs. VLIW

- **Array processor: 单个操作作用在多个不同的数据元素上**

# Multimedia Extensions (aka SIMD extensions)

- **在已有的ISA中添加一些向量长度很短的向量操作指令**
- **将已有的 64-bit 寄存器拆分为 2x32b or 4x16b or 8x8b**
  - 1957年，Lincoln Labs TX-2 将36bit datapath 拆分为 2x18b or 4x9b
  - 新的设计具有较宽的寄存器
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b for Intel AVX (Advanced Vector Extensions)
- **单条指令可实现寄存器中所有向量元素的操作**

# Multimedia Extensions (aka SIMD extensions)

# Intel Pentium MMX Operations

- **idea: 一条指令操作同时作用于不同的数据元**
  - 全阵列处理
  - 用于多媒体操作

Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

- No VLEN register
- Opcode determines data type:
  - 8 8-bit bytes
  - 4 16-bit words
  - 2 32-bit doublewords
  - 1 64-bit quadword
- Stride always equal to 1.

# MMX Example: Image Overlaying (I)

Figure 8. Chroma keying: image overlay using a background color.

PCMPEQB MM1, MM3

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| MM1 | Blue | Blue | Blue | Blue | Blue | Blue | Blue | Blue |
| MM3 | X7!=blue | X6!=blue | X5=blue | X4=blue | X3!=blue | X2!=blue | X1=blue | X0=blue |
| MM1 | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF |

Bitmask

Figure 9. Generating the selection bit mask.

# MMX Example: Image Overlaying (II)

Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
Movq      mm3, mem1    /* Load eight pixels from
                          woman's image
Movq      mm4, mem2    /* Load eight pixels from the
                          blossom image
Pcmpeqb   mm1, mm3
Pand      mm4, mm1
Pandn     mm1, mm3
Por       mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.

# Multimedia Extensions versus Vectors

- **受限的指令集:**
  - 无向量长度控制
  - Load/store操作无 常数步长寻址和 scatter/gather操作
  - loads 操作必须64/128-bit 边界对齐
- **受限的向量寄存器长度:**
  - 需要超标量发射以保持multiply/add/load 部件忙
  - 通过循环展开隐藏延迟增加了寄存器读写压力
- **在微处理器设计中向全向量化发展**
  - 更好地支持非对齐存储器访问
  - 支持双精度浮点数操作 (64-bit floating-point)
  - Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)

# -Review

- **向量处理机性能评估**
  - 向量指令流执行时间: Convey, Chimes, Start-up time
  - 其他指标： R∞ , N1/2 , NV
- **向量机的存储器访问**
  - 存储器组织：独立存储体、多体交叉方式
  - Stride : 固定步长（1 or 常数）, 非固定步长（index）
- **基于向量机模型的优化**
  - 链接技术
  - 有条件执行
  - 稀疏矩阵的操作
- **多媒体扩展指令**
  - 扩展的指令类型较少
  - 向量寄存器长度较短
- **GPU**

# Recap: Vector/SIMD Processing Summary

- **Vector/SIMD 机器适合挖据规整的数据级并行**
  - 同样的操作作用在许多数据元素上
  - 提高性能、设计简单（向量内的操作相互独立）
- **性能的提升受限于代码的向量化**
  - 标量操作限制着向量机的性能
- **很多已有的ISA扩展了一些SIMD操作**
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

# Summary：向量体系结构

- **向量处理机基本概念**
  - 基本思想：两个向量的对应分量进行运算，产生一个结果向量
- **向量处理机基本特征**
  - VSIW-一条指令包含多个操作
  - 单条向量指令内所包含的操作相互独立
  - 以已知模式访问存储器-多体交叉存储系统
  - 控制相关少
- **向量处理机基本结构**
  - **向量指令并行执行**
  - **向量运算部件的执行方式-流水线方式**
  - **向量部件结构-多“道”结构-多条运算流水线**
- **向量处理机性能评估**
  - **向量指令流执行时间:** Convey, Chimes, Start-up time
  - **其他指标：** $R_\infty$，$N_{1/2}$，$N_V$
- **向量处理机性能优化**
  - 链接技术
  - 条件执行
  - 稀疏矩阵

# Acknowledgements


- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - John Hennessy (Standford)and David Patterson (UCB)
  - Chenxi Zhang (Tongji)
  - Muhamed Mudawar (KFUPM)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**

# 计算机体系结构

GPGPU

# Review

- **单发射Tumasulo + 推断执行，superscalar, VLIW, Fined-grained MT (FGMT), coarse-grained MT, CMP, SMT, vector**
  - 哪个的概念和实现还不清楚、需要重新过一遍？

# Review

- **向量机的存储器访问**
  - 存储器组织：独立存储体、多体交叉方式
  - Stride : 固定步长（1 or 常数）, 非固定步长（index）
- **基于向量机模型的优化**
  - 链接技术
  - 有条件执行
  - 稀疏矩阵的操作

- **多媒体扩展指令**
  - 扩展的指令类型较少
  - 向量寄存器长度较短
- **GPU**

# GPU

- GPU的起源
- GPU基本的硬件结构
- GPU编程模型
- GPU的存储层次
- GPU分支处理（发散与汇聚）

# Graphics Processing Units (GPUs)

- **早期的GPU是指带有高性能浮点运算部件、可高效生成3D图形的具有固定功能的专用设备 (mid-late 1990s)**
  - 让PC机具有类似工作站的图形功能
  - **用户可以配置图形处理流水线，但不是真正的对其编程**
- **2001-2005，GPU加入了越来越多的可编程性**
  - 例如新的语言 Cg可用来编写一些小的程序处理图形的顶点或像素，是Windows DirectX的变体
  - 大规模并行（针对每帧上百万顶点或像素）但非常受限于编程模型
- **有些用户注意到通过将输入和输出数据映射为图像，并对顶点或像素渲染计算 可进行通用计算**
  - 因为不得不使用图形流水线模型，这对完成通用计算来说是个非常难用的编程模型

# What GPUs were originally designed to do: 3D rendering

Input: description of a scene:
3D surface geometry (e.g., triangle mesh)
surface materials, lights, camera, etc.

Image credit: Henrik Wann Jensen

Output: image of the scene

**Simple definition of rendering task: Generating a photorealistic image from 2D/3D models. Computing how each triangle in 3D mesh contributes to appearance of each pixel in the image?**

# Tip: how to explain a system

- Step 1: describe the things (key entities) that are manipulated
  - The nouns

# Real-time graphics primitives (entities)

**Represent surface as a 3D triangle mesh**

**Vertices**

**Primitives**
**(e.g., triangles, points, lines)**

# Real-time graphics primitives (entities)

Vertices

Primitives
(e.g., triangles, points, lines)

Fragments

Pixels (in an image)

# Tip: how to explain a system

- Step 1: describe the things (key entities) that are manipulated
  - The nouns
- Step 2: describe operations the system performs on the entities
  - The verbs

# Graphics pipeline architecture

Performs operations on vertices, triangles, fragments, and pixels

# Fragment processing simulates reflection of light off of real-world materials

Example materials:

# Early Graphics Programming

- **Graphics programming APIs provided application programmer mechanisms to set parameters of lights and materials**
  - `glLight(light_id, parameter_id, parameter_value)`
    - **Examples of light parameters: ambient/diffuse/specular color, position, direction**
  - `glMaterial(face, parameter_id, parameter_value)`
    - **Examples of material parameters: surface color, shininess**

# Graphics Shading Language

- Allow application to specify materials and lights programmatically!
  - Support large diversity in materials
  - Support large diversity in lighting conditions
- Programmer provides mini-programs ("shaders") that defines pipeline logic for certain stages
  - Pipeline maps shader function onto all elements of input stream

# Example Fragment Shader

**Run once per fragment (per pixel covered by a triangle)**

**HLSL language shader program:**
**defines behavior of fragment processing stage**

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

read-only global variables

per-fragment inputs

"fragment shader"
(a.k.a kernel function mapped onto input fragment stream)

per-fragment output: surface color at pixel

**myTex is a texture map**

myTex =

# Shader Result

Image contains output of diffuseShader for each pixel covered by surface
(Pixels covered by multiple surfaces contain output from surface closest to camera)

# Observation circa 2001-2003

**These GPUs are very fast processors for performing the same computation (shaders) on collections of data (streams of vertices, fragments, pixels).**

**Wait a minute! That sounds a lot like data parallelism to me! I remember data parallelism from exotic supercomputers in the 90s.**

# Observation circa 2001-2003

**Set OpenGL output image size to be output array size (e.g., 512 x 512)**

**Render 2 triangles that exactly cover screen**

**(one shader computation per pixel = one shader computation output image element)**

**We now can use the GPU like a data-parallel programming system.**

**Fragment shader function is mapped over 512 x 512 element collection.**

# “GPGPU” 2002-2003

Coupled Map Lattice Simulation [Harris 02]

Sparse Matrix Solvers [Bolz 03]

Ray Tracing on Programmable Graphics Hardware [Purcell 02]

# Brook stream programming language (2004)

- Stanford research project
- Abstract GPU as data-parallel processor

```
kernel void scale(float amount, float a<>, out float b<>)
{
   b = amount * a;
}

// note: omitting stream element initialization
float scale_amount;
float input_stream<1000>;
float output_stream<1000>;

// map kernel onto streams
scale(scale_amount, input_stream, output_stream);
```

- Brook compiler turned generic stream program into OpenGL commands such as drawTriangles()

# General-Purpose GPUs (GP-GPUs)

- **2006年, Nvidia 的 GeForce 8800 GPU 支持一种新的编程语言: CUDA**
  - “Compute Unified Device Architecture”
  - 随后工业界推出OpenCL，与CUDA具有相同的ideas, 但独立于供应商
- **Idea: 针对通用计算，发挥GPU计算的高性能和存储器的高带宽来加速一些通用计算中的核心（Kernels）**
- **一种协处理器模型（GPU作为附加设备）：Host CPU发射数据并行的kernels 到GP-GPU上运行**
- **我们仅讨论Nvidia CUDA样式的简化版本，仅考虑GPU的计算核部分，不涉及图形加速部分**

# NVIDIA Tesla architecture (2007)

## (GeForce 8xxx series)

**First alternative, non-graphics-specific ("compute mode") software interface to GPU**

### Multi-core CPU architecture

CPU presents itself to OS as a multi-processor system

ISA provides instructions for managing execution context (program counter, VM mappings, etc.) on a per core basis

### Pre-2007 GPU architecture

GPU presents following interface** to system software (driver):

Set screen size
Set shader program for pipeline
DrawTriangles

(** interface also included many other commands for configuring graphics pipeline)

### Post-2007 "compute mode" GPU architecture

GPU presents a new data-parallel interface to system software (driver):

Set kernel program
Launch(kernel, N)

# Using CPU+GPU Architecture

- CPU+GPU异构多核系统
  - 针对每个任务选择合适的处理器和存储器
- 通用CPU 适合执行一些串行的线程
  - 串行执行快
  - 带有cache，访问存储器延时低
- GPU 适合执行大量并行线程
  - 可扩放的并行执行
  - 高带宽的并行存取

强控制、弱计算

弱控制、强计算

(a) System with discrete GPU

(b) Integrated CPU and GPU

Figure 1.1: GPU computing systems include CPUs.

# GPU: a multithreaded coprocessor

SP: scalar processor
'CUDA core'

Executes one thread

**SM**
streaming multiprocessor
32xSP (or 16, 48 or more)

<u>Fast</u> local '**shared memory**'
(shared between SPs)
16 KiB (or 64 KiB)

# GPUs are SIMD Engines Underneath

Figure 3.1: Microarchitecture of a generic GPGPU core.

- **指令流水线类似于SIMD的流水线。**
  - 不是用SIMD指令编程
  - 基于一般的指令，应用由一组线程构成。
- **两个概念**
  - Programming Model (Software) vs Execution Model (Hardware)
- **编程模型指程序员如何描述应用（从程序员角度看到的机器模型）**
  - 例如, 顺序模型 (von Neumann), 数据并行(SIMD), 数据流模型、多线程模型 (MIMD, SPMD), …
- **执行模型指硬件底层如何执行代码**
  - 例如, 乱序执行、向量机、数据流处理机、多处理机、多线程处理机等
- **执行模型与编程模型可以差别很大**
  - 例如., 顺序模型可以在乱序执行的处理器上执行。
  - SPMD 模型可以用SIMD处理器实现 (a GPU)

# How Can You Exploit Parallelism Here?

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

*Scalar Sequential Code*

Let's examine three programming options to exploit instruction-level parallelism present in this sequential code:

1. Sequential (SISD)
2. Data-Parallel (SIMD)
3. Multithreaded (MIMD/SPMD)

# Prog. Model 1: Sequential (SISD)

*Scalar Sequential Code*

- **Can be executed on a Pipelined processor**
- **Out-of-order execution processor**
  - Independent instructions executed when ready
  - Different iterations are present in the instruction window and can execute in parallel in multiple functional units
  - In other words, the loop is dynamically unrolled by the hardware
- **Superscalar or VLIW processor**
  - Can fetch and execute multiple instructions per cycle

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

# Prog. Model 2: Data Parallel (SIMD)

*Scalar Sequential Code*

***Vector Instruction***

*Vectorized Code*

VLD A → V1

VLD B → V2

VADD V1 + V2 → V3

VST V3 → C

Realization: Each iteration is independent

Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Best executed by a SIMD processor (vector, array)

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

# Prog. Model 3: Multithreaded

*Scalar Sequential Code*

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

....

Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

Can be executed on a MIMD machine

# Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

....

This particular model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine
Single Instruction Multiple Thread

Can be executed on a SIMD machine

# SPMD

- **Single procedure/program, multiple data**
  - 它是一种编程模型而不是计算机组织
- **每个处理单元执行同样的过程，处理不同的数据**
  - 这些过程可以在程序中的某个点上同步，例如 barriers

- **多条指令流执行相同的程序**
  - 每个程序/过程
    - 操作不同的数据
    - 运行时可以执行不同的控制流路径
  - 许多科学计算应用以这种方式编程，运行在MIMD硬件结构上 (multiprocessors)
  - 现代 GPUs 以这种类似的方式编程，运行在SIMD硬件上

# A GPU is a SIMD (SIMT) Machine

- **GPU不是用SIMD指令编程**
- **使用线程 (SPMD 编程模型)**
  - 每个线程执行同样的代码，但操作不同的数据元素
  - 每个线程有自己的上下文(即可以独立地启动/执行等）
- **一组执行相同指令的线程由硬件动态组织成warp**
  - 一个warp是由硬件形成的SIMD操作

# Threads and Blocks

- 一个线程对应一个数据元素
- 大量的线程组织成很多线程块
- 许多线程块组成一个网格

- GPU 由硬件对线程进行管理
  - Thread Block Scheduler
  - SIMD Thread Scheduler

| Nvidia CUDA | OpenCL | Hennessy & Patterson[7] |
|---|---|---|
| Thread | Work-item | Sequence of SIMD Lane operations |
| Warp | Wavefront | Thread of SIMD Instructions |
| Block | Workgroup | Body of vectorized loop |
| Grid | NDRange | Vectorized loop |

Figure 6 Grid of Thread Blocks

# Programmer’s View of Execution

创建足够的线程块以适应输入向量(Nvidia 中将由多个线程块构成的、在GPU上运行的代码称为*Grid*，*Grid可以是2维的*)

blockDim = 256 (programmer can choose)

Conditional **(i<n)** turns off unused threads in last block

处理向量长度为8192的程序组织：

Grid 包含16个线程块（512个元素/Block）

Block包含16个SIMD 线程

每个SIMD线城处理32个向量元素

**Figure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long.** Each thread of SIMD instructions calculates 32 elements per instruction, and in this example, each Thread Block contains 16 threads of SIMD instructions and the Grid contains 16 Thread Blocks. The hardware Thread Block Scheduler assigns Thread Blocks to multithreaded SIMD Processors, and the hardware Thread Scheduler picks which thread of SIMD instructions to run each clock cycle within a SIMD Processor. Only SIMD Threads in the same Thread Block can communicate via local memory. (The maximum number of SIMD Threads that can execute simultaneously per Thread Block is 32 for Pascal GPUs.)

# Hardware Execution Model

- **GPU 由多个并行核构成，每个核是一个多线程SIMD处理器（包含多个车道（Lanes））**
- **CPU 发送整个 “grid” 到GPU，由GPU将这些线程块分发到多个核上（每个线程块在一个核上运行）**
  - GPU上核的数量对程序员而言是透明的

# Simplified CUDA Programming Model

- **计算由大量的相互独立的线程(*CUDA threads* or *microthreads*) 完成，这些线程组合成线程块（*thread blocks*）**

```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{  for (int i=0; i<n; i++)
       y[i] = a*x[i] + y[i]; }

// CUDA version.
__host__   // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__   // Piece run on GP-GPU.
void daxpy(int n, double a, double*x, double*y)
{  int i = blockIdx.x*blockDim.x + threadId.x;
   if (i<n) y[i]=a*x[i]+y[i]; }
```

# NVIDIA Instruction Set Arch.

- ISA 是硬件指令集的抽象
  - “Parallel Thread Execution (PTX)”
  - 使用虚拟寄存器
  - 用软件将其翻译成机器码
  - Example:

```
shl.s32  R8, blockIdx, 9       ; Thread Block ID * Block size (512 or 2^9)
add.s32  R8, R8, threadIdx  ; R8 = i = my CUDA thread ID
ld.global.f64      RD0, [X+R8]   ; RD0 = X[i]
ld.global.f64      RD2, [Y+R8]   ; RD2 = Y[i]
mul.f64   RD0, RD0, RD4      ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64   RD0, RD0, RD2      ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64    [Y+R8], RD0  ; Y[i] = sum (X[i]*a + Y[i])
```

# CUDA kernel maps to Grid of Blocks

```
kernel_func<<<nblk, nthread>>>(param, … );
```

# Thread blocks execute on a SM
# Thread instructions execute on a SP

**float myVar;** **__shared__ float shVar;** **__device__ float glVar;**

# CUDA： “Single Instruction, Multiple Thread”

- **GPUs 使用 SIMT模型, 每个CUDA线程的标量指令流汇聚在一起在硬件上以SIMD方式执行 (Nvidia groups 32 CUDA threads into a warp)**

**Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor.** It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

# SPMD on SIMT Machine

Warp: A set of threads that execute the same instruction (i.e., at the same PC)

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

This particular model is also called:

SPMD: Single Program Multiple Data

A GPU executes it using the SIMT model:
Single Instruction Multiple Thread

# SIMD vs. SIMT Execution Model

- **SIMD: 一条指令流（一串顺序的SIMD指令），每条指令对应多个数据输入（向量指令）**
- **SIMT: 多个指令流（标量指令）构成线程，这些线程动态构成warp。一个Warp处理多个数据元素**

- **SIMT 主要优点：**
  - 可以独立地处理线程,即每个线程可以在任何标量流水线上单独执行（ MIMD 处理模式）
  - 可以将线程组织成warp，即可以将执行相同指令流的线程构成warp，形成SIMD 处理模式, 以充分发挥SIMD处理的优势

# Multithreading of Warps

- 设一个warp由 32 threads构成
- 如果有32K次循环 → 1K 个warps
- 这些Warps可以在同一条流水线上交替执行
  - Fine grained multithreading of warps

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

# Warps and Warp-Level FGMT

- **Warp: 执行相同指令（操作不同的数据元素）的一组线程**
- **同一Warp中的所有线程执行相同代码**
- **Warp: The threads that run lengthwise in a woven fabric …**

# Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle

Slide credit: Krste Asanovic

# Latency Hiding via Warp-Level FGMT

- **Warp: 一组执行相同指令的线程（作用于不同的数据元素）**
- **Fine-grained multithreading**
  - 流水线上每次执行每个线程的一条指令 (No interlocking)
  - 通过warp的交叉执行来隐藏延时
- **所有线程的寄存器值保存在寄存器文件中**
- **FGMT 可以容忍长延时**
  - Millions of pixels

# Warp-based SIMD vs. Traditional SIMD

- **传统的SIMD仅包含一个线程**
  - Lock step: 一条向量指令执行完，然后启动下一条向量指令
  - 编程模型为SIMD (no extra threads) → SW 需要知道向量长度
  - ISA 包含vector/SIMD指令

- **Warp-based SIMD 由多个标量线程构成，以SIMD方式执行(即所有的线程执行相同的指令)**
  - 不需要lock step
  - 每个线程可以单独对待(即可以映射到不同的warp中），编程模型不是SIMD
    - SW 不必知道向量长度
    - 多个线程可以动态构成warp
  - ISA是标量ISA → 可以动态形成逻辑上的向量指令
  - 是一种在SIMD硬件上实现的SPMD编程模型

# Warp Execution (Recall the Slide)

32-thread warp executing ADD A[tid],B[tid] → C[tid]

*Execution using one pipelined functional unit*

*Execution using four pipelined functional units*

Slide credit: Krste Asanovic

# SIMD Execution Unit Structure

Slide credit: Krste Asanovic

# GPU Memory Hierarchy

constant memory

# GPU Global Memory

Global memory is the “main” memory of the GPU. It has global scope and lifetime of the allocating program (or until cudaFree is called).

Global memory is similar to the heap in a C program.

Global memory is separate hardware from the GPU core (containing SM’s,
caches, etc). The vast majority of memory on GPU is global memory. If data doesn’t fit into global memory, you are going to process it in chunks that do fit in globalmemory.

GPUs have .5 - 24GB of global memory, with most now having ~2GB. Global memory latency is ~300ns on Kepler and ~600ns on Fermi

# GPU Shared Memory

- Very fast memory located in the SM
- Same hardware as L1 cache, ~5ns of latency
- Maximum size of 48KB, but user configurable
- Scope of shared memory is the block

# Registers

- Fastest “memory” possible, about 10x faster than shared Memory

- Most stack variables declared in kernels are stored in registers (example: float x)

- Statically indexed arrays stored on the stack are sometimes put in registers

# Local Memory

- Local memory is everything on the stack that can't fit in registers. The scope of local memory is just the thread.

- Local memory is stored in global memory (much slower than registers!)

# GPU L1 Cache

- Fermi - caches local & global memory (local memory's scope is local to the thread and it resides off-chip)
- Kepler, Maxwell - only caches local memory
- same hardware as shared memory
- configurable size (16, 32, 48KB)
- each SM has its own L1 cache

# GPU L2 Cache

- caches all global & local memory accesses
- ~1MB in size
- shared by all SM's

# Constant Memory

- Used for constants that cannot be compiled into program
- Constants must be set from host before running kernel
- Constant memory is global memory with a special cache
- 64KB for user, 64KB for compiler (kernel arguments are passed through constant memory)

# Constant Cache

- 8KB cache on each SM specially designed to broadcast a single memory address to all threads in a warp (called static indexing)
- Can also load any statically indexed data
- through constant cache using “load uniform” (LDU) instruction

# A few terms

- SM, SP, lane
- Grid, block, warp, thread
- SIMT, SIMD, SPMD, FGMT

flynn

programming

Nvidia

# GPU Memory Hierarchy

SIMD

SIMT

Ld $r0, 4 $(r1)

① CPU vs GPU

## NVIDIA Fermi Memory Hierarchy

② why GPU is faster?

memory addr

# SIMT Memory Access

SIMT

- 不同线程的相同指令使用线程id来存取不同的数据元素

Let's assume N=16, 4 threads per warp → 4 warps

Slide credit: Hyesoon Kim

# Global Memory Access

Each thread issues memory accesses to data types of varying sizes, perhaps as small as 1 byte entities

Given an address to load or store, memory returns/updates "segments" of either 32 bytes, 64 bytes or 128 bytes

Maximizing bandwidth:

- Operate on an entire 128 byte segment for each memory transfer

# Understanding Global Memory Access

Start with memory request by smallest numbered thread.

Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)

Find other active threads requesting addresses within that segment and coalesce

SIMT

Reduce transaction size if possible

Access memory and mark threads as "inactive"

Repeat until all threads in half-warp are serviced
(Includes Tesla and GTX platforms)

# Thread Block Scheduling

- Each TB allocated to an SM; Multiple TBs in a SM
- Threads in a thread block may need to cooperate
  - Load/Store from memory together
  - Share results (through shared memory)
  - Synchronize with each other
- Thread Blocks can execute in any order

# Resident Thread Blocks

- No. of thread blocks allocated to SM is limited by
  - Maximum Thread blocks
  - Maximum number of threads
  - Registers (Registers per thread * Threads per Block * # Blks)
  - Shared Memory (Shared Memory per Block * # Blks)
- Warps of resident threadblocks are context switched to hide (long) latency

# SM Warp Scheduling

- Each SM launches Warps of Threads warp level MT
  - 2 levels of parallelism
  - Shared instruction fetch per 32 threads (warps)
- SM hardware implements zero overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

# SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction cache
  - into any instruction buffer slot
- Issue one "ready-to-go" warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand scoreboarding used to prevent hazards
- Issue selection based on round robin or age of warp
- SM broadcasts the same instruction to 32 threads of a Warp

# Scoreboarding

- How to determine if a thread is ready to execute?
- A scoreboard is a table in hardware that tracks
  - instructions being fetched, issued, executed
  - resources (functional units and operands) they need
  - which instructions modify which registers
- Old concept from CDC 6600 (1960s)

# Scoreboarding Example

Consider three separate instruction streams: warp1, warp3 and warp8

| Warp | Current Instruction | Instruction State |
|---|---|---|
| Warp 1 | 42 | Computing |
| Warp 3 | 95 | Computing |
| Warp 8 | 11 | Operands ready to go |
| ... | | |

Schedule at time k

# Scoreboarding Example

Consider three separate instruction streams: warp1, warp3 and warp8

| Warp | Current Instruction | Instruction State |
|---|---|---|
| Warp 1 | 42 | Ready to write result |
| Warp 3 | 95 | Computing |
| Warp 8 | 11 | Computing |
| … | | |

Schedule at time k+1

# Threads Can Take Different Paths in Warp-based SIMD

- 每个线程可以包含控制流指令
- 这些线程可以执行不同的控制流路径

# Control Flow Problem in GPUs/SIMT

- GPU 控制逻辑使用 SIMD流水线 以节省资源
  - 这些标量线程构成 warp

- 当一个WARP中的线程分支到不同的执行路径时，产生分支发散（Branch divergence）

**与向量处理机模型的条件执行类似**
**（Vector Mask and Masked Vector Operations?）**

# Conditional Branching

- **与向量结构类似, GPU 使用内部的屏蔽字(masks)**
- **还使用了（SIMT-Stack）**
  - 分支同步堆栈
    - 保存分支的路径地址
    - 保存该路径的SIMD lane 屏蔽字(mask)
      - 即指示哪些车道可以提交结果
  - 指令标记(instruction markers）
    - 管理何时分支（divergence）到多个执行路径，何时路径汇合(converge)
- **PTX层**
  - CUDA线程的控制流由PTX分支指令(branch、call、return and exit）控制 以及
  - 由程序员指定的每个线程车道的1-bit谓词寄存器
- **GPU硬件指令层,控制流包括：**
  - 分支指令(branch,jump call return)
  - 特殊的指令用于管理分支同步栈
  - GPU硬件为每个SIMD thread 提供堆栈 保存分支的路径
  - GPU硬件指令带有控制每个线程车道的1-bit谓词寄存器

# Branch divergence

- **硬件跟踪各μthreads转移的方向（判定哪些是成功的转移，哪些是失败的转移）**
- **如果所有线程所走的路径相同，那么可以保持这种 SIMD 执行模式**
- **如果各线程选择的方向不一致，那么创建一个屏蔽（mask）向量来指示各线程的转移方向（成功、失败）**
- **继续执行分支失败的路径，将分支成功的路径压入硬件堆栈（分支同步堆栈），待后续执行**
- **SIMD 车道何时执行分支同步堆栈中的路径?**
  - 通过执行pop操作，弹出执行路径以及屏蔽字，执行该转移路径
  - SIMD lane完成整个分支路径执行后再执行下一条指令 称为converge(汇聚）
  - 对于相同长度的路径，IF-THEN-ELSE 操作 的 效率平均为50%

# Example

```
if (X[i] != 0)
        X[i] = X[i] – Y[i];
else X[i] = Z[i];
```

```
        ld.global.f64  RD0, [X+R8]  ; RD0 = X[i]
        setp.neq.s32  P1, RD0, #0   ; P1 is predicate register 1
        @!P1, bra  ELSE1, *Push     ; Push old mask, set new mask bits
                                    ; if P1 false, go to ELSE1
        ld.global.f64 RD2, [Y+R8]   ; RD2 = Y[i]
        sub.f64 RD0, RD0, RD2       ; Difference in RD0
        st.global.f64 [X+R8], RD0   ; X[i] = RD0
        @P1, bra  ENDIF1, *Comp     ; complement mask bits
                                    ; if P1 true, go to ENDIF1
ELSE1:  ld.global.f64 RD0, [Z+R8]   ; RD0 = Z[i]
        st.global.f64 [X+R8], RD0   ; X[i] = RD0
ENDIF1: <next instruction>, *Pop    ; pop to restore old mask
```

then

# Branch Divergence Handling

```
A;
if (some condition) {
    B;
} else {
    C;
}
D;
```

One per warp

**Control Flow Stack**

| | Next PC | Recv PC | Active Mask |
|---|---|---|---|
| TOS → | D | -- | 1111 |
| | B | D | 1110 |
| | C | D | 0001 |

**Execution Sequence**

# SIMT stack operation

```
do {
  t1 = tid*N;          // A
  t2 = t1 + i;
  t3 = data1[t2];
  t4 = 0;
  if( t3 != t4 ) {
    t5 = data2[t2]; // B
    if( t5 != t4 ) {
      x += 1;          // C
    } else {
      y += 2;          // D
    }
  } else {
    z += 3;            // F
  }
  i++;                 // G
} while( i < N );
```

Example CUDA C source code for illustrating SIMT stack operation

```
A:    mul.lo.u32      t1, tid, N;
      add.u32         t2, t1, i;
      ld.global.u32   t3, [t2];
      mov.u32         t4, 0;
      setp.eq.u32     p1, t3, t4;
@p1   bra             F;
B:    ld.global.u32   t5, [t2];
      setp.eq.u32     p2, t5, t4;
@p2   bra             D;
C:    add.u32         x, x, 1;
      bra             E;
D:    add.u32         y, y, 2;
E:    bra             G;
F:    add.u32         z, z, 3;
G:    add.u32         i, i, 1;
      setp.le.u32     p3, i, N;
@p3   bra             A;
```

Example PTX assembly code for illustrating SIMT stack operation.

Figure 3.4: Example of SIMT stack operation (based on Figure 5 from Fung et al. [2007]).

# Remember: Each Thread Is Independent

- **SIMT 主要优点：**
  - 可以独立地处理线程,即每个线程可以在任何标量流水线上单独执行（ MIMD 处理模式）
  - 可以将线程组织成warp，即可以将执行相同指令流的线程构成warp，形成SIMD 处理模式, 以充分发挥SIMD处理的优势

- **如果有许多线程，对具有相同PC值的线程可以将它们动态组织到一个warp中**
- **这样可以减少“分支发散” →提高SIMD 利用率**
  - SIMD 利用率: 执行有用操作的SIMD lanes的比例 (即, 执行活动线程的比例）

# Dynamic Warp Formation/Merging

- **Idea:** 分支发散之后，动态合并执行相同指令的线程
  - 从那些等待的warp中形成新的warp
  - 足够多的线程分支到每个路径，有可能创建完整的新warp

# Dynamic Warp Formation/Merging

- Idea:
  - 分支发散之后，动态合并执行相同指令的线程

- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

# Dynamic Warp Formation Example

# Hardware Constraints Limit Flexibility of Warp Grouping

# When You Group Threads Dynamically

- **存储器访问如何处理？**

- **固定模式的存储器访问相对简单，当动态构成warp时，使得访问模式具有随机性，使得问题变得复杂。→ 降低存储器访问的局部性**
  - → 导致存储器带宽利用率的下降

# What About Memory Divergence?

- 现代 GPUs 包括高速缓存，减少对存储器的访问
- Ideally: 一个warp中的所有线程的存储器访问都命中 (互相没有冲突）
- Problem: 一个Warp中有些命中，有些失效
- Problem: 一个线程的stall导致整个warp停顿

- 需要有相关技术来解决存储器发散访问问题

# NVIDIA GeForce GTX 285

- **NVIDIA-speak:**
  - 240 stream processors
  - "SIMT execution"
  -

- **Generic speak:**
  - 30 cores
  - 8 SIMD functional units per core

# NVIDIA GeForce GTX 285 "core"

= SIMD functional unit, control shared across 8 units

= multiply-add

= multiply

= instruction stream decode

= execution context storage

# NVIDIA GeForce GTX 285 “core”

- **Groups of 32 threads share instruction stream (each group is a Warp)**
- **Up to 32 warps are simultaneously interleaved**
- **Up to 1024 thread contexts can be stored**

# NVIDIA GeForce GTX 285

- **30 cores on the GTX 285: 30,720 threads**

# GP100 GPU

- 56 SMs
- 64 Lanes/SM

Or

- 3584 Stream Processors

**Figure 4.20 Block diagram of the multithreaded SIMD Processor of a Pascal GPU.** Each of the 64 SIMD Lanes (cores) has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The 64 SIMD Lanes interact with 32 double-precision ALUs (DP units) that perform 64-bit floating-point arithmetic, 16 load-store units (LD/STs), and 16 special function units (SFUs) that calculate functions such as square roots, reciprocals, sines, and cosines.

# Summary- 向量处理机 vs. GPU

- **不同层次相近的术语比较**

| Type | Descriptive name | Closest old term outside of GPUs | Official CUDA/NVIDIA GPU term | Short explanation |
|---|---|---|---|---|
| Program abstractions | Vectorizable Loop | Vectorizable Loop | Grid | A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel |
| | Body of Vectorized Loop | Body of a (Strip-Mined) Vectorized Loop | Thread Block | A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via local memory |
| | Sequence of SIMD Lane Operations | One iteration of a Scalar Loop | CUDA Thread | A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register |
| Machine object | A Thread of SIMD Instructions | Thread of Vector Instructions | Warp | A traditional thread, but it only contains SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask |
| | SIMD Instruction | Vector Instruction | PTX Instruction | A single SIMD instruction executed across SIMD Lanes |

# Summary-向量处理机 vs. GPU

| Type | Descriptive name | Closest old term outside of GPUs | Official CUDA/NVIDIA GPU term | Short explanation |
|---|---|---|---|---|
| Processing hardware | Multithreaded SIMD Processor | (Multithreaded) Vector Processor | Streaming Multiprocessor | A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors |
| | Thread Block Scheduler | Scalar Processor | Giga Thread Engine | Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors |
| | SIMD Thread Scheduler | Thread Scheduler in a Multithreaded CPU | Warp Scheduler | Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution |
| | SIMD Lane | Vector Lane | Thread Processor | A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask |
| Memory hardware | GPU Memory | Main Memory | Global Memory | DRAM memory accessible by all multithreaded SIMD Processors in a GPU |
| | Private Memory | Stack or Thread Local Storage (OS) | Local Memory | Portion of DRAM memory private to each SIMD Lane |
| | Local Memory | Local Memory | Shared Memory | Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors |
| | SIMD Lane Registers | Vector Lane Registers | Thread Processor Registers | Registers in a single SIMD Lane allocated across a full Thread Block (body of vectorized loop) |

**Figure 4.12 Quick guide to GPU terms used in this chapter.** We use the first column for hardware terms. Four groups cluster these 11 terms. From top to bottom: program abstractions, machine objects, processing hardware, and memory hardware. Figure 4.21 on page 312 associates vector terms with the closest terms here, and Figure 4.24 on page 317 and Figure 4.25 on page 318 reveal the official CUDA/NVIDIA and AMD terms and definitions along with the terms used by OpenCL.

# Acknowledgements


- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - John Hennessy (Standford)and David Patterson (UCB)
  - Chenxi Zhang (Tongji)
  - Muhamed Mudawar (KFUPM)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**
- **CMU Introduction to Computer Architecture**

**http://www.ece.cmu.edu/~ece447/**

# 计算机体系结构

## Cache Consistency

# 单个处理器存储器操作的序

- **操作顺序由 von Neumann 模型约定**
- **顺序串行执行**
  - 硬件执行 load 和 store 操作以程序序顺序执行
- **乱序执行不改变程序语义**
  - 硬件以程序序报告 load 和 store 操作的结果
- **优点 1) 在执行时机器状态是确定的。 2) 程序的不同次运行机器状态是一致的，有利于程序调试**
- **缺点：维护这种序的额外开销，降低了性能，增加了复杂性，降低了可扩放性**

# MIMD 处理器中的存储器操作序

- **每个处理器的存储器操作以顺序序执行对应于运行在该处理器上的一个线程（假设每个处理器符合 von Neumann 模型）**
- **多个处理器并发地执行存储器操作**
- **存储器如何看来自所有处理器的存储器操作?**
  - 即不同处理器发出的存储器操作，在共享存储器端看到的应该是什么序?

**FIGURE 3.1:** A Sequentially Consistent Execution of Table 3.1's Program.

# 序的重要性

- **易于调试**
  - 若程序的每次执行都是相同的序有利于程序的调试
- **正确性**
  - 从不同处理器看到的存储器操作序可能会不同
- **性能和代价权衡**
  - 强制符合严格 “sequential ordering” 使得硬件设计人员实现性能增强技术变得十分复杂（例如 ., OoO 执行 , caches)
- **<p ： 程序序 (program order)**

  **<m： 存储器操作序 (memory order)**

# 顺序同一性的存储器模型

“ A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

*Leslie Lamport*

Sequential Consistency =

多个进程之间的存储器操作可以任意交叉

每个进程的存储器操作按照程序序

# 顺序同一性的充分条件

- **多个进程可以交织执行，但顺序同一性模型没有定义具体的交织方式，满足每个进程程序序的总体执行序可能会很多。因此有下列定义：**
  - 顺序同一性的执行：如果程序的一次执行产生的结果与前面定义的任意一种可能的总体序产生的结果一致，那么程序的这次执行就称为是顺序同一的。
  - 顺序同一性的系统：如果在一个系统上的任何可能的执行都是顺序同一的，那么这个系统就是顺序同一的

# 顺序同一性的充分条件

- 每个进程按照程序执行序发出存储操作
- 发出写操作后，进程要等待写的完成，才能发出它的下一个操作
- 发出读操作后，进程不仅要等待读的完成，还要等待产生所读数据的那个写操作完成，才能发出它的下个操作。即：如果该写操作对这个处理器来说完成了，那么这个处理器应该等待该写操作对所有处理器都完成了。
- 第三个条件保证了写操作的原子性。即读操作必须等待逻辑上先前的写操作变得全局可见

| TABLE 3.1: Should r2 Always be Set to NEW? | | |
|---|---|---|
| **Core C1** | **Core C2** | **Comments** |
| S1: Store data = NEW;<br>S2: Store flag = SET; | L1: Load r1 = flag;<br>B1: if (r1 ≠ SET) goto L1;<br>L2: Load r2 = data; | /* Initially, data = 0 & flag ≠ SET */<br>/* L1 & B1 may repeat many times */ |

**FIGURE 3.1:** A Sequentially Consistent Execution of Table 3.1's Program.

**(1) 所有 core 执行的 Load/Store 满足程序序**

/* Load -> Load */

If L(a) <p L(b) => L(a) <m L(b)

/*Load -> Store */

If L(a) <p S(b) => L(a) <m L(b)

/* Store ->Store */

If S(a) <p S(b) => S(a) <m S(b)

/* Store -> Load */

If S(a) <p L(b) => S(a) <m L(b)

**(2) 对同一存储单元的 Load 操作的值来源于最近一次写操作 (global memory order)**

Value of L(a) = Value of $Max_{<m}$\{S(a) <m L(a)\},

$Max_{<m}$ 表示最近的 memory order

# Sequential Consistency

- **SC 约束了所有的存储器操作的序：**
  - Write → Read
  - Write → Write
  - Read → Read
  - Read → Write
- **是有关并行程序执行的简单模型**
- **但是，直觉上在单处理器上的合理的存储器操作的重排序会违反 SC 模型**
- **现代微处理器设计中一直都在应用重排序操作来获得性能提升 (write buffers, overlapped writes, non-blocking reads…).**
- **Question: 如何协调性能提升与 SC 的约束?**

# Issues in Implementing Sequential Consistency

现代计算机系统实现 SC 的两个问题

- *Out-of-order execution capability*

  Load(a); Load(b) *yes*
  Load(a); Store(b) *yes if* a ≠ b
  Store(a); Load(b) *yes if* a ≠ b
  Store(a); Store(b) *yes if* a ≠ b

- *Caches. Write buffer*

  Cache 使得某一处理器的 store 操作不能被另一处理器即时看到

***No common commercial architecture has a sequentially consistent memory model ！！！***

# Relaxed Consistency Models

- **Rules:**
  - X → Y :Operation X must complete before operation Y is done

Sequential consistency requires (SC) :

R → W, R → R, W → R, W → W

Relax W → R (TSO)

"Total store ordering"

(X86)

Relax W → W (PSO)

"Partial store order"

Figure 2.24: Relationship among models according to the "stricter" relation.

Relax R → W and R → R

"Weak ordering" and "release consistency"

Relax R → R , R → W , W-R, W → W (RMO)

"Release Memory Ordering"

Maintains the program order to access the same location:

W →R, W → W

# Simple categorization of relaxed models

| Relaxation | W → R Order | W → W Order | R → RW Order | Read Others' Write Early | Read Own Write Early | Safety net |
|---|---|---|---|---|---|---|
| SC [16] | | | | | √ | |
| IBM 370 [14] | √ | | | | | serialization instructions |
| TSO [20] | √ | | | | √ | RMW |
| PC [13, 12] | √ | | | √ | √ | RMW |
| PSO [20] | √ | √ | | | √ | RMW, STBAR |
| WO [5] | √ | √ | √ | | √ | synchronization |
| RCsc [13, 12] | √ | √ | √ | | √ | release, acquire, nsync, RMW |
| RCpc [13, 12] | √ | √ | √ | √ | √ | release, acquire, nsync, RMW |
| Alpha [19] | √ | √ | √ | | √ | MB, WMB |
| RMO [21] | √ | √ | √ | | √ | various MEMBAR's |
| PowerPC [17, 4] | √ | √ | √ | √ | √ | SYNC |

Figure 8: Simple categorization of relaxed models. A √ indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The "Read Own Write Early" relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The "Read Others' Write Early" relaxation is possible and detectable with complex implementations of RCsc.

| TABLE 4.1: Can Both r1 and r2 be Set to 0? | | |
|---|---|---|
| **Core C1** | **Core C2** | **Comments** |
| S1: x = NEW;<br>L1: r1 = y; | S2: y = NEW;<br>L2: r2 = x; | /* Initially, x = 0 & y = 0*/ |

TABLE 4.1: Can Both r1 and r2 be Set to 0?

| Core C1 | Core C2 | Comments |
|---|---|---|
| S1: x = NEW;<br>L1: r1 = y; | S2: y = NEW;<br>L2: r2 = x; | /* Initially, x = 0 & y = 0*/ |

(c) TSO & SC Execution 3

(d) TSO Execution, but NOT an SC Execution

RE 4.2: Four Alternative TSO Executions of Table 4.1's Program.

| TABLE 4.3: Can r1 or r3 be Set to 0? | | |
|---|---|---|
| **Core C1** | **Core C2** | **Comments** |
| S1: x = NEW;<br>L1: r1 = x;<br>L2: r2 = y; | S2: y = NEW;<br>L3: r3 = y;<br>L4: r4 = x; | /* Initially, x = 0 & y = 0*/<br><br>/* Assume r2 = 0 & r4 = 0 */ |

- **当 (r2, r4) = (0, 0) 时，（r1, r3) 一定为 (0, 0）？**

**FIGURE 4.3:** A TSO Execution of Table 4-3's Program (with "bypassing").

# Memory Fences

*Instructions to sequentialize memory accesses*

实现弱同一性或放松的存储器模型的处理器（允许针对不同地址的 loads 和 stores 操作乱序）需要提供存储器栅栏指令来强制对某些存储器操作串行化

*Examples of processors with relaxed memory models:*

Sparc V8 (TSO,PSO): Membar

Sparc V9 (RMO):

Membar #LoadLoad, Membar #LoadStore

Membar #StoreLoad, Membar #StoreStore

PowerPC (WO): Sync, EIEIO

ARM: DMB (Data Memory Barrier)

X86/64: mfence (Global Memory Barrier)

*存储器栅栏是一种代价比较大的操作，仅仅在需要时，对存储器操作串行化*

Table 1: Summary of Memory Ordering

| | Loads Reordered After Loads? | Loads Reordered After Stores? | Stores Reordered After Stores? | Stores Reordered After Loads? | Atomic Instructions Reordered With Loads? | Atomic Instructions Reordered With Stores? | Dependent Loads Reordered? | Incoherent Instruction Cache/Pipeline? |
|---|---|---|---|---|---|---|---|---|
| Alpha | Y | Y | Y | Y | Y | Y | Y | Y |
| AMD64 | | | | Y | | | | |
| IA64 | Y | Y | Y | Y | Y | Y | | Y |
| (PA-RISC) | Y | Y | Y | Y | | | | |
| PA-RISC CPUs | | | | | | | | |
| POWER™ | Y | Y | Y | Y | Y | Y | | Y |
| (SPARC RMO) | Y | Y | Y | Y | Y | Y | | Y |
| (SPARC PSO) | | | Y | Y | | Y | | Y |
| SPARC TSO | | | | Y | | | | Y |
| x86 | | | | Y | | | | Y |
| (x86 OOStore) | Y | Y | Y | Y | | | | Y |
| zSeries® | | | | Y | | | | Y |

# Synchronization

系统中只要存在并发进程，即使是单核系统都需要同步操作

总体上存在两类同步操作问题：

*生产者 - 消费者问题：一个* 消费者进程必须等待生产者进程产生数据

*互斥问题（Mutual Exclusion）：* 保证在一个给定的时间内只有一个进程使用共享资源（临界区）

# A Producer-Consumer Example

Producer posting Item x:

Load $R_{tail}$, (tail)

Store ($R_{tail}$), x

$R_{tail}=R_{tail}+1$

Store (tail), $R_{tail}$

Consumer:

Load $R_{head}$, (head)

spin: Load $R_{tail}$, (tail)

if $R_{head}==R_{tail}$ goto spin

Load R, ($R_{head}$)

$R_{head}=R_{head}+1$

Store (head), $R_{head}$

process(R)

*Problems?*

假设指令都是顺序执行的

# A Producer-Consumer Example *continued*

Producer posting Item x:

Load $R_{tail}$, (tail)

1 Store ($R_{tail}$), x

$R_{tail}$=$R_{tail}$+1

2 Store (tail), $R_{tail}$

Consumer:

Load $R_{head}$, (head)

spin: Load $R_{tail}$, (tail) 3

if $R_{head}$==$R_{tail}$ goto spin

Load R, ($R_{head}$) 4

$R_{head}$=$R_{head}$+1

Store (head), $R_{head}$

process(R)

*Can the tail pointer get updated before the item x is stored?*

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:

2, 3, 4, 1

4, 1, 2, 3

# Using Memory Fences

Producer posting Item x:

Load $R_{tail}$, (tail)

Store ($R_{tail}$), x

$Membar_{SS}$

$R_{tail}=R_{tail}+1$

Store (tail), $R_{tail}$

*ensures that tail ptr is not updated before x has been stored*

Consumer:

Load $R_{head}$, (head)

spin: Load $R_{tail}$, (tail)

if $R_{head}==R_{tail}$ goto spin

$Membar_{LL}$

Load R, ($R_{head}$)

$R_{head}=R_{head}+1$

Store (head), $R_{head}$

process(R)

*ensures that R is not loaded before x has been stored*

# Multiple Consumer Example

Producer posting Item x:

Load $R_{tail}$, (tail)
Store ($R_{tail}$), x
$R_{tail}=R_{tail}+1$
Store (tail), $R_{tail}$

*Critical section:*
*Needs to be executed atomically by one consumer*

Consumer:

Load $R_{head}$, (head)
spin: Load $R_{tail}$, (tail)
if $R_{head}==R_{tail}$ goto spin
Load R, ($R_{head}$)
$R_{head}=R_{head}+1$
Store (head), $R_{head}$
process(R)

*What is wrong with this code?*

# Mutual Exclusion Using Load/Store

基于两个共享变量 c1 和 c2 的同步协议。初始状态 c1 和 c2 均为 0

*Process 1*

```
      ...
     c1=1;
L:  if c2=1 then go to L
      < critical section>
     c1=0;
```

*Process 2*

```
      ...
     c2=1;
L:  if c1=1 then go to L
      < critical section>
     c2=0;
```

What is wrong? *Deadlock!*

# Mutual Exclusion: second attempt

*Process 1*

```
    ...
L:  c1=1;
    if c2=1 then
    { c1=0; go to L}
      < critical section>
    c1=0
```

*Process 2*

```
    ...
L:  c2=1;
    if c1=1 then
    { c2=0; go to L}
      < critical section>
    c2=0
```

- 为避免死锁，我们让一进程等待时放弃 reservation （预订）
  - Process 1 sets c1 to 0.
- 死锁显然是没有了，但有可能会发生 活锁 *(livelock)* 现象
  - C1 = 1, C2=1, Read C2, Read C1, C1 =0, C2 = 0, C1=1, C2=1, C1=0, C2=0
- 可能还会出现某个进程始终无法进入临界区 ⊠⊠ *starvation*
  - *例如： C1=1， C2= 1， C1=0， Read C1 进入临界区， P1 和P2 竞争 P2 始终胜出*

# A Protocol for Mutual Exclusion

*T. Dekker, 1966*

基于 3 个共享变量 c1, c2 和 turn 的互斥协议，初始状态三个变量均为 0.

*Process 1*

```
      ...
      c1=1;
      turn = 1;
L: if c2=1 & turn=1
            then go to L
        < critical section>
      c1=0;
```

*Process 2*

```
      ...
      c2=1;
      turn = 2;
L: if c1=1 & turn=2
            then go to L
        < critical section>
      c2=0;
```

- turn = *i* *保证仅仅进程 i 等待*
- 变量 c1 和 c2 保证 n 个进程互斥地访问临界区，该算法由 *Dijkstra 给出，相当精巧*

# Locks or Semaphores

*E. W. Dijkstra, 1965*

信号量（A *semaphore*） 是一非负整数，具有如下操作：

P(s): *if s>0, decrement s by 1, otherwise wait*

V(s): *increment s by 1 and wake up one of the waiting processes*

P(s) 和 V(s) 必须是原子操作，i.e.，不能被中断，不能由多个处理器交叉访问 s

```
Process i
    P(s)
        <critical section>
    V(s)
```

*s 的初始值设置为可访问临界区的最大进程数*

# Atomic Operations

- **顺序一致性并不保证操作的原子性**
- **原子性：存储器操作使用原子性操作，操作序列一次全部完成。例如 exchange （交换操作）。**

exchange(r,M): 互换寄存器 r 与存储单元 M 的内容

```
r0 = 1;
do exchange(r0,S) while (r0 != 0);
                                  //S is memory location
  //enter critical section
  .....
  //exit critical section
  S = 0;
```

# Implementation of Semaphores

在顺序同一性模型中，信号量 (mutual exclusion) 可以用常规的 Load 和 Store 指令实现，但是互斥的协议很难设计。一种简单的解决方案是提供：

*atomic read-modify-write instructions*

Examples: *m is a memory location, R is a register*

```
Test&Set (m), R:
    R ← M[m];
    if R==0 then
    M[m] ← 1;
```

```
Fetch&Add (m), Rv, R:
    R ← M[m];
    M[m] ← R + Rv;
```

```
Swap (m), R:
    Rt ← M[m];
    M[m] ← R;
    R ← Rt;
```

# Multiple Consumers Example

*using the Test&Set Instruction*

```
P:    Test&Set (mutex),R_temp
      if (R_temp!=0) goto P
      Load R_head, (head)
spin:    Load R_tail, (tail)
      if R_head==R_tail goto spin
      Load R, (R_head)
      R_head=R_head+1
      Store (head), R_head
V:    Store (mutex),0
      process(R)
```

*Critical Section*

其他原子的 read-modify-write 指令（Swap, Fetch&Add，etc.）也能实现 P(s) 和 V(s) 操作

# Nonblocking Synchronization

```
Compare&Swap(m), R_t, R_s:
  if (R_t==M[m])
      then      M[m]=R_s;
                R_s=R_t ;
                status ← success;
      else status ← fail;
```

status is an *implicit argument*

```
try:     Load R_head, (head)
spin:    Load R_tail, (tail)
     if R_head==R_tail goto spin
     Load R, (R_head)
     R_newhead = R_head+1
     Compare&Swap(head), R_head, R_newhead
     if (status==fail) goto try
     process(R)
```

# Performance of Locks

Blocking atomic read-modify-write instructions

*e.g., Test&Set, Fetch&Add, Swap*

vs

Non-blocking atomic read-modify-write instructions

*e.g., Compare&Swap,*

*Load-reserve/Store-conditional*

vs

Protocols based on ordinary Loads and Stores

*Performance depends on several interacting factors:*

degree of contention,

caches,

out-of-order execution of Loads and Stores

*later ...*

# 本课程主要内容

**关键字：**

**并行（并发）、评估、优化**

# 第 1 章

- **重点关注： 与性能度量、功耗（能耗）有关的问题**

- **性能度量**
  - 响应时间 (response time)
  - 吞吐率 (Throughput)
- **CPU 执行时间 = IC × CPI × T**
  - CPI （ Cycles per Instruction)
- **MIPS = Millions of Instructions Per Second**
- **Latency versus Bandwidth**
  - Latency 指单个任务的执行时间， Bandwidth 指单位时间完成的任务量（rate）
  - Latency 的提升滞后于带宽的提升（在过去的 30 年）
- **Amdahl’s Law 用来度量加速比（speedup)**
  - 性能提升受限于任务中可加速部分所占的比例
  - 应用于多处理器系统的基本假设：在给定的问题规模下，研究随着处理器数目的增加性能的变化
- **Benchmarks：指一组用于测试的程序**
  - 比较计算机系统的性能
  - SPEC benchmark： 针对一组应用综合性能值采用 SPEC ratios 的几何平均

# Power & Energy

- **Dynamic Energy ∝**

  **Capacitive Load × Voltage$^2$**

  – 从 0-1-0 或 1-0-1 逻辑跃迁的脉冲能量

  – Capacitive Load = 输出晶体管和导线的电容负载

  – 20 年来晶体管供电电压已经从 5V 降到 1V

- **Dynamic Power ∝**

**Capacitive Load × Voltage$^2$ × Frequency Switched**

# 第 2 章 指令集架构

- **重点关注： ISA 设计的基本方法、 RISC**

- **ISA 需考虑的问题：**
  - Class of ISA; Memory addressing; Types and sizes of operands ; Operations ; Control flow instructions ; Encoding an ISA
  - ……
- **ISA 的类型**
  - 通用寄存器型占主导地位
- **寻址方式**
  - 重要的寻址方式：偏移寻址方式，立即数寻址方式，寄存器间址方式
    - SPEC 测试表明，使用频度达到 75%--99%
  - 偏移字段的大小应该在 12 - 16 bits，可满足 75%-99% 的需求
  - 立即数字段的大小应该在 8 -16 bits，可满足 50%-80% 的需求
- **操作数的类型和大小**
  - 对单字、双字的数据访问具有较高的频率
  - 支持 64 位双字操作，更具有一般性

- **ISA 的功能设计：任务为确定硬件支持哪些操作。方法是统计的方法。存在 CISC 和 RISC 两种类型**
  - CISC （Complex Instruction Set Computer）
    - 目标：强化指令功能，减少指令的指令条数，以提高系统性能
    - 基本方法：面向目标程序的优化，面向高级语言和编译器的优化
  - RISC （Reduced Instruction Set Computer）
    - 目标：通过简化指令系统，用最高效的方法实现最常用的指令
    - 主要手段：充分发挥流水线的效率，降低（优化） CPI
- **控制转移类指令**
- **指令编码（指令格式）**
- **MIPS ISA**
- **RISC-V ISA**

# 第 3 章

- 流水线性能评估和优化技术

- 实际吞吐率：假设 k 段，完成 n 个任务，单位时间所实际完成的任务数。
- 加速比：k 段流水线的速度与等功能的非流水线的速度之比。
- 效率：流水线的设备利用率。

$$TP_{\max} = \frac{1}{\max\{\Delta t_i\}}$$

$$TP = \frac{n}{\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j}$$

$$S = \frac{n \cdot \sum_{i=1}^{k} \Delta t_i}{\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j}$$

$$E = \frac{n \cdot \sum_{i=1}^{k} \Delta t_j}{k \cdot [\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j]}$$

$$E = TP \cdot \frac{\sum_{i=1}^{k} \Delta t_i}{k}$$

- **指令流水线通过指令重叠减小 CPI**
- **充分利用数据通路**
  - 当前指令执行时，启动下一条指令
  - 其性能受限于花费时间最长的段
  - 检测和消除相关
- **如何有利于流水线技术的应用**
  - 所有的指令都等长
  - 只有很少的指令格式
  - 只用 Load/Store 来进行存储器访问

# 流水线的加速比计算

$$CPI_{pipelined} = Ideal\ CPI + Average\ Stall\ cycles\ per\ Inst$$

$$Speedup = \frac{Ideal\ CPI \times Pipeline\ depth}{Ideal\ CPI + Pipeline\ stall\ CPI} \times \frac{Cycle\ Time_{unpipelined}}{Cycle\ Time_{pipelined}}$$

*For simple RISC pipeline, CPI = 1:*

$$Speedup = \frac{Pipeline\ depth}{1 + Pipeline\ stall\ CPI} \times \frac{Cycle\ Time_{unpipelined}}{Cycle\ Time_{pipelined}}$$

- **影响流水线性能**
  - 结构相关、数据相关
  - 控制相关、异常
- **异常处理**
  - 种类与分类
  - 精确与非精确中断
- **支持浮点数操作的MIPS流水线**
  - Latency & Repeat Interval
  - 问题：结构相关（增多）；数据相关、控制相关引起的stall增多；有新的冲突源产生；定向路径增多；异常处理复杂
  - MIPS R4000 8级流水线
    - 存储器操作分阶段 – load延迟为2个cycles
    - Branch操作在EX段确定分支方向 - 3个cycles的延迟
    - 多个定向源： EX/DF， DF/DS， DS/TC， TC/WB
  - MIPS R4000的浮点数操作

# 第 4 章

- **存储层次、 Cache 性能评估和优化、主存组织、虚拟存储（TLB）**
- **Cache 基本原理 4Q**
- **Cache 性能分析**

CPU time = (CPU execution clock cycles + Memory stall clock cycles) x clock cycle time

Memory stall clock cycles =
(Reads x Read miss rate x Read miss penalty + Writes x Write miss rate x Write miss penalty)

Memory stall clock cycles =
Memory accesses x Miss rate x Miss penalty

Different measure: AMAT
Average Memory Access time (AMAT) =
Hit Time + (Miss Rate x Miss Penalty)

# Cache 优化

| Technique | Hit time | Band-width | Miss penalty | Miss rate | Power consumption | Hardware cost/ complexity | Comment |
|---|---|---|---|---|---|---|---|
| Small and simple caches | + | | | – | + | 0 | Trivial; widely used |
| Way-predicting caches | + | | | | + | 1 | Used in Pentium 4 |
| Pipelined cache access | – | + | | | | 1 | Widely used |
| Nonblocking caches | | + | + | | | 3 | Widely used |
| Banked caches | | + | | | + | 1 | Used in L2 of both i7 and Cortex-A8 |
| Critical word first and early restart | | | + | | | 2 | Widely used |
| Merging write buffer | | | + | | | 1 | Widely used with write through |
| Compiler techniques to reduce cache misses | | | | + | | 0 | Software is a challenge, but many compilers handle common linear algebra calculations |
| Hardware prefetching of instructions and data | | | + | + | – | 2 instr., 3 data | Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware. |
| Compiler-controlled prefetching | | | + | + | | 3 | Needs nonblocking cache; possible instruction overhead; in many CPUs |

**Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity.** Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

# 第 5 章

- **提高指令级并行的硬件方法、软件方法**

- **指令级并行 (ILP)：流水线的平均 CPI**
  - Pipeline CPI = Ideal Pipeline CPI + Struct Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls +……
  - 提高指令级并行的方法
    - 软件方法：指令流调度，循环展开，软件流水线，trace scheduling
    - 硬件方法
- **软件方法：指令流调度 - 循环展开**
- **硬件方法：两种指令流动态调度方法**
  - 如何处理精确中断？
  - Out-of-order execution -> out-of-order completion!
  - 如何处理分支？
- **动态分支预测**
  - 常见的方法有哪些？评估

- **存储器访问的冲突消解**
  - Total Ordering
  - Partial Ordering
  - Load Ordering, Store Ordering
  - Store Ordering

- **Superscalar and VLIW: CPI < 1 (IPC > 1)**
  - Dynamic issue vs. Static issue
  - 同一时刻发射更多的指令 => 导致更大的冲突开销

# Memory Disambiguation

TABLE 6.1: Memory disambiguation schemes.

| NAME | SPECULATIVE | DESCRIPTION |
|---|---|---|
| Total Ordering | No | All memory accesses are processed in order. |
| Partial Ordering | No | All stores are processed in order, but loads execute out of order as long as all previous stores have computed their address. |
| Load Ordering<br>Store Ordering | No | Execution between loads and stores is out of order, but all loads execute in order among them, and all stores execute in order among them. |
| Store Ordering | Yes | Stores execute in order, but loads execute completely out of order. |

- **非投机方式的基本原则：当前存储器指令之前的 store 指令计算存储器地址后，才能执行当前的存储器操作**

# Multithreaded Categories

# 第 6 章

- **向量处理机模型、G P U**

- **向量处理机基本概念**
  - 基本思想：两个向量的对应分量进行运算，产生一个结果向量
- **向量处理机基本特征**
  - VSIW-一条指令包含多个操作
  - 单条向量指令内所包含的操作相互独立
  - 以已知模式访问存储器-多体交叉存储系统
  - 控制相关少
- **向量处理机基本结构**
  - 向量指令并行执行
  - 向量运算部件的执行方式-流水线方式
  - 向量部件结构-多“道”结构-多条运算流水线
- **向量处理机性能评估**
  - 向量指令流执行时间：Convey, Chimes, Start-up time
  - 其他指标： $R_\infty$ , $N_{1/2}$ , $N_V$

- **两个问题**
  - 向量处理机中的存储器访问
  - 向量处理机中的优化技术

- **向量机的存储器访问**
  - 存储器组织：独立存储体、多体交叉方式
  - Stride：固定步长（1 or 常数），非固定步长（index）
- **分段开采技术**
- **基于向量机模型的优化**
  - 链接技术
  - 有条件执行
  - 稀疏矩阵的操作

# GPU

- **GPU： 多线程协处理器**
- **GPU 编程模型： SPMD (Single Program Multiple Data)**
  - 使用线程（SPMD 编程模型），不是用 SIMD 指令编程
    - 每个线程执行同样的代码，但操作不同的数据元素
    - 每个线程有自己的上下文（即可以独立地启动/执行等）
  - 计算由大量的相互独立的线程 (CUDA threads or microthreads) 完成，这些线程组合成线程块（thread blocks）
- **GPU 执行模型： SIMT (Single Instruction Multiple Thread)**
  - 一组执行相同指令的线程由硬件动态组织成 warp
  - 一个 warp 是由硬件形成的 SIMD 操作
- **GPU 存储器组织**
  - Local Memory， Shared Memory, Global Memory
- **GPU 分支处理（发散与汇聚）**

# 第 7 章

- **重点关注： Coherence 、 Consistency**

# Acknowledgements


- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - David Patterson (UCB)
  - Chenxi Zhang (Tongji)
- **UCB material derived from course CS152、 CS252、 CS61C**
- **KFUPM material derived from course COE501、 COE502**