Drastic 激烈的

innovation创新

dissipation耗散

ILP：指令级并行

TLP：任务级并行

DLP：数据级并行

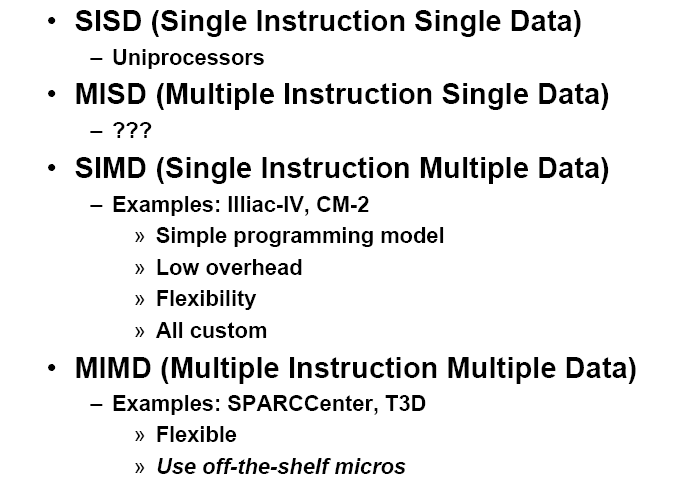
Peripheral device：外围设备

MISD：实现应用不存在（视图破解单个编码语言的不同解码程序）

SISD：单处理机系统

SIMD：并行处理器，异步执行同一条指令（相同指令不同数据流）

MIMD：多处理机系统，指令任务的并行

Dispatcher：调度程序

Minicomputer => servers using microprocessors 小型机 -- 使用微处理器的服务器

Mainframe => multiprocessors consisting of microprocessors 主机 -- 使用处理器集合的多处理机

Supercomputer => multiprocessor collections 超级计算机 -- 多处理器的集合

Irrelevant：无关紧要的

A rule of thumb经验法则

Cost decrease rate ~ density（密度） increase rate

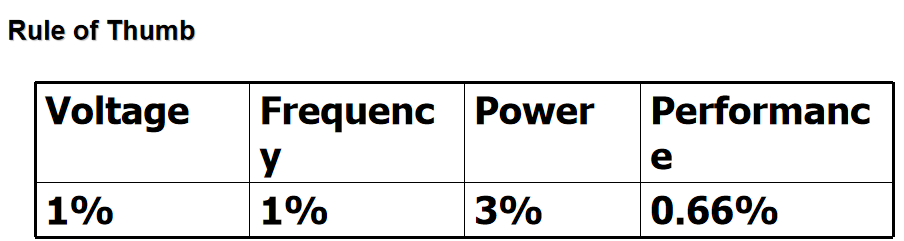
Bandwidth grow rate ~ improvement in latency

Power dynamic（功率） = ½ \*Capacitive load（电容负载） \* Voltage^2 \* Frequency switched

Energy dynamic（能量） = Capacitive load \* Voltage^2

（关闭时功耗 -- 漏电qaq）Power static = current static（静态电流？） \* Voltage

功率优化 --- 经验公式

一块大的core还是多块小的core

Power ~ area（功耗正比面积）

Single thread performance ~ area\*\*.5（每一块core性能正比其面积开根号）

Dependability 是比 reliability更为宽泛的概念；

Reliability

MTTF: Mean Time To Failure

MTTR: Mean Time To Repair

FIT : Failure In Time = 1/MTTF

MTBF: Mean Time Between Failure = MTTF+MTTR

Availability

Module Availability= MTTF / (MTTF + MTTR)

性能指标Performance Metrics

Response Time (or execution time)：Time between start and end of an event

Throughput ：Total amount of work done in a given time

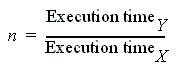
Response time的统计存有问题（多程序运行、等待用户输入开销），因而使用CPU time更为准确。

User CPU time - time spent in user mode

System CPU time - time spent in the operating system (OS)

MIPS - Millions of Instructions per Second（with the same instruction set）

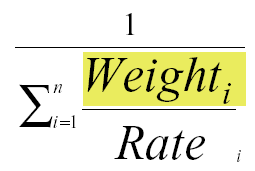
Performance = 1 / Execution time

Processor X is **n times fast** than Y is

Total Execution Time：

Time 记录时：

Rate（速率）记录时（分子部分应该为n）：

重温：

Take advantage of parallelism

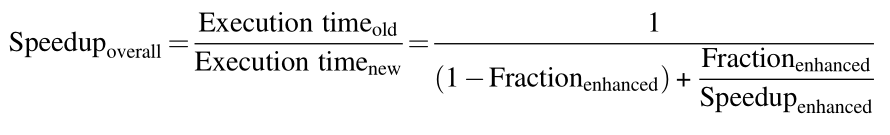
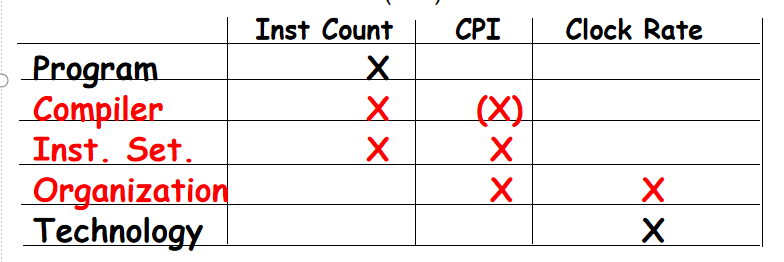
Principle of Locality

Focus on the common case

Amdahl's Law

Increasing the clock rate would not affect memory access time（特别注意，内存访问时间本身不变，因为clock rate变化而发生变化的是clock cycle）





CPU Performance Equation（CPU性能方程）

FP 浮点数运算

FPSQR 浮点数平方根运算

overlapped 重叠

Individual 单个的

流水线的hazard

Structural hazards：硬件上的冲突

Memory与data寄存器的冲突访问

double bump：前半时钟周期读寄存器，后半时钟周期写寄存器

Data hazards：数据使用上的冲突

Forwarding、bypass、short-circuiting是一个东西

可以使用forwarding以及交换instruction顺序来解决data hazard

Control hazards：PC跳转上的冲突

Control hazard比data hazard会引发更多的stall，且其不可以被软件解决

Control hazard一定会带来一个时钟周期的stall --- 称作branch delay slot，可以通过交换instruction顺序来填充这一slot（from before，from target，from fall-through）

Stall、bubble、pipeline bubble同义

考虑到stall的pipeline的加速比计算公式：

chap3_2-6newLatency---the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.

Initiation interval---the number of cycles that must elapse（使用） between instructions issue to the same unit.

Initial interval往往是latency + 1

RAW( Read after write) true dependence

WAW(Write after write) output dependence

WAR( Write after read) anti-denpendence

当EXE阶段计算将使用多余一个时钟周期（往往对映于浮点数运算，不同计算模块可以并行）：

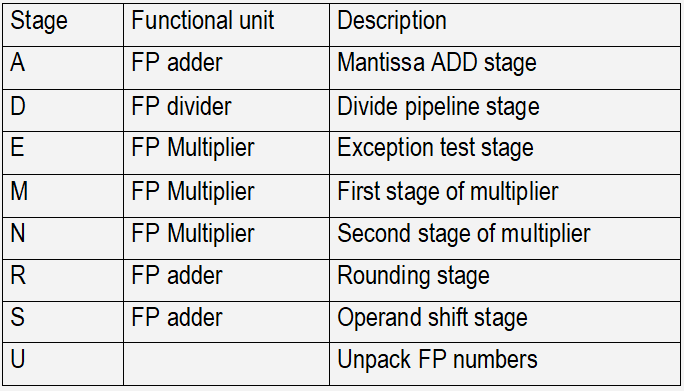
会产生WAW hazard，可以使用Stall an instruction直到先前的hazard instruction执行到MEM阶段，或直接Cancel the WB phase of the earlier instruction来解决。

ID阶段需要进行一些额外的控制手段：

Check for RAW hazards，只有Its source registers are no longer listed as destinations，此条instruction才能正常执行。

Check for structural hazards：divide unit and Register write port，ID阶段直接控制多个instruction不在同一clock cycle执行write back操作。

Check for WAW hazards：check不同计算功能路径上相同的destination register WB冲突问题



Temporal Locality

Spatial Locality

Write allocate：在write miss的时候，将block拿到cache中再进行修改

No write allocate（write round）： write miss时直接修改内存中的数据

Block offset：块偏移可以得到的字节



Uniformly 一致的，公平的

Unified cache ：data cache和instruction cache融合为一体 ---- Split I & D cache

Miss rate的分母是memory reference

注意local miss rate和global miss rate的区别（除以total access的是global miss rate）



注意这个概念，Memory references per instruction: 1.5，只需要考虑这个memory reference即可

Compulsory also called cold start misses or first reference misses.

Coherence一致性

2:1 rule of thumb a direct-mapped cache of size N has the same miss rate as a 2-way set-associative cache of size N/2.

Non blocking hit在浮点数运算中更具有优势（相较于associative），虽然本不改变hit time

加速virtual转换，使用TLB；使用virtual cache --- 并行命中与转换操作；更小更简单的cache；不同进程间区别，不只是简单的flash cache；

内存

Quantitative：定量的

主存也称为：inter-memory(EMS memory)

主存：DRAM（need refresh），Cache：SRAM（不需要refresh）.

RAM（Random Access Memory）是一种需要持续供电才能将数据保存在其中的存储器。

Performance of main memory

Latency: harder to reduce latency

Bandwidth: easier to improve bandwidth with new organizations. （Bytes/clock cycles）

Quadruples：成倍增加

Simultaneously：同时进行

Widening memory好但复杂；simple Interleaved Memory 较好且简单

Number of banks ≥ Number of clock cycles to access word in bank （banks -- interleave状态下区块的个数）

Access time ----- time between when a read is requested and when the desired word arrives,

Cycle time ----- minimum time between requests to memory.

virtual memory

memory is contiguous

memory space is larger than physical memory, Provides illusion（错觉） of very large memory

Virtual memory的两个格式：

Pages---- fixed-size blocks,

segments---- variable-size blocks

减小页表大小：页表转换为hash的格式，容器数量为物理页数量 --- inverted page table

降低虚拟地址转换时间：Computers use a cache dedicated（专用的） to these address translations, called a **translation look-aside buffer**, or simply translation buffer.

Instruction-Level Parallelism (ILP)

Basic Block: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit

Exploit：开发，利用

RAW true dependence 写后读

WAR anti-dependence 读后写

Name dependence 同名寄存器使用

WAW output dependence 写后写

Control dependence

2 properties critical to program correctness

Exception behavior：跳转的结果与顺序（？）

Data flow：actual flow of data values among instructions that produce results and those that consume them

减少stall的方法

Rewrite loop（unroll loop）

重排instruction

Dynamic Scheduling

Allow instructions behind stall to proceed 需要决定完成时间与顺序

可以跨pipeline高速运行

重新划分pipeline运行阶段：

IF阶段；将ID阶段分为Issue（decode instruction，check结构hazard）和Read operands阶段（check没有data hazard后，read operand）[对映于IS和RO阶段]；（合并）EXE与MEM阶段为EX阶段；WB阶段

ISSUE阶段，functional unit空闲，目标寄存器没有被预定使用（同样作为目标寄存器），没有WAW hazard与结构冲突

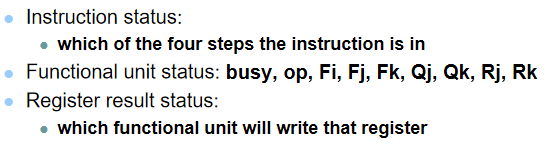
Read Operand 动态解决RAW hazard，之前的指令都未将operand（源寄存器）作为目标寄存器

EX阶段，结束后修改scoreboard信息，释放functional unit

WB阶段，check WAR hazard，必要时暂停流水线（其他instruction源寄存器与目标寄存器是否冲突）

Scoreboard

顺序解析instruction，乱序计算，使instruction尽可能早的计算

Three data structure

Instruction status 正在执行什么指令（数字）的什么阶段

Function unit status， functional unit是否使用，执行指令是什么，操作数来源以及是否准备就绪及源操作数状态（Rj,Rk记录是否准备就绪，Qj,Qk记录寄存器数据来源functional unit）

Register result status 寄存器结果由谁提供

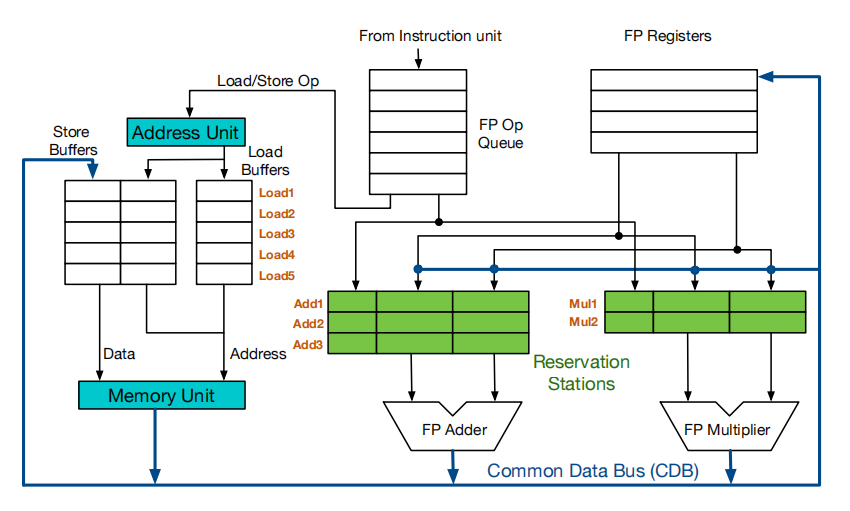
Scoreboard缺陷

If we can't find independent instructions to execute, scoreboard helps very little.

Issue queue可以关注的instruction数量也是有限的（window的大小有限，一般不能跨越branch）

WAR and WAW hazards limit the scoreboard more than RAW hazards（对于WAW，WAR而言只能使用等待策略）

Tomasulo’s Algorithm

Tomasulo’s algorithm是一种计算机硬件架构的算法，硬件设计示意图如下：

FP OP Queue表示浮点指令队列，指令在此等待发射

青绿色模块是加法单元和乘法单元的保留站（保留站可以保留已经发射的指令信息和缓冲下来的数据） --- reservation station（有多个相同的functional unit处理单元）

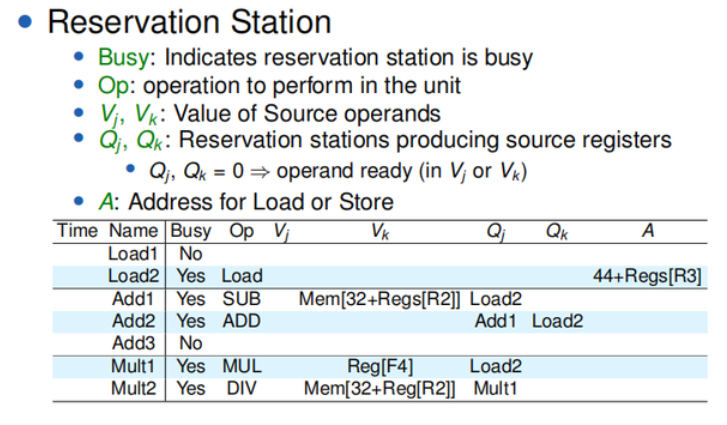
蓝绿色的Address Unit是地址计算单元，在此算法中存储指令在执行前会先计算好存储的地址，Memory Unit是存储单元

CDB（common data bus）是数据广播总线，它可以直达寄存器堆（用以更新通用寄存器）、加法乘法存储单元的保留站（输送保留站中指令需要的数据）

Tomasulo’s algorithm使用的记录信息

还有store buffer结构

Reservation station

其中Vj，Vk直接把能读取到的数据拷贝至保留站中（每个保留站只对应一个functional unit，数据直接读取相当于使用了寄存器重命名），Qj,Qk记录未准备就绪的寄存器数据来源，A是存储指令的目标地址（Issue阶段已计算获得），time记录functional unit需要的时钟周期数，time为0时对映execution completion

另外，Tomasulo’s algorithm同样要记录寄存器结果来源

算法执行流程

发射：指令按程序中的顺序一条接一条被发射到保留站。判断能否发射的唯一标准是指令对应通路的保留站是否有空余位置。周期结束时会更新保留站和寄存器结果状态表，如果指令有可以读取的数据，就会立刻拷贝到保留站中；寄存器结果状态表中总是存有最新的值，即如果后序指令的目的寄存器和前序指令的目的寄存器重合，那就只保留后序指令的写信息。

执行：指令通过拷贝数据和监听CDB获得源数据（阻塞直到所有源数据均可获得），然后开始执行，执行可能是多周期的，在执行过程中不改变处理器状态。

写回：指令在写回阶段**通过CDB总线将数据直通到寄存器堆和各个保留站**；周期结束时，根据寄存器结果状态表来更新寄存器堆，并且清除保留站和寄存器结果状态表的信息。

Common data bus：data + source（come from）

64 bits of data + 4 bits of Functional Unit（source address）

Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue

Pending 挂起的，待定的

Simultaneously 同时的

优点

Reservation stations and CDB（重命名寄存器，没有register个数的bottleneck）

消除WAW、WAR的stall

可以实现循环间并行（硬件的loop unrolling）

No limit to basic block

缺陷

实现复杂

CDB传输速率要求较高，performance limited by common data bus（每次只能有一个 functional unit计算完成）

非精确中断（中断来临时，因乱序执行无法判断确切指令执行位置）

Get precise interrupt point的方法

Undo table mappings

一些解决方案

Dynamic branch prediction

Speculation 在control dependences are resolved之前执行后续指令

Dynamic scheduling决定基本块之间的调度

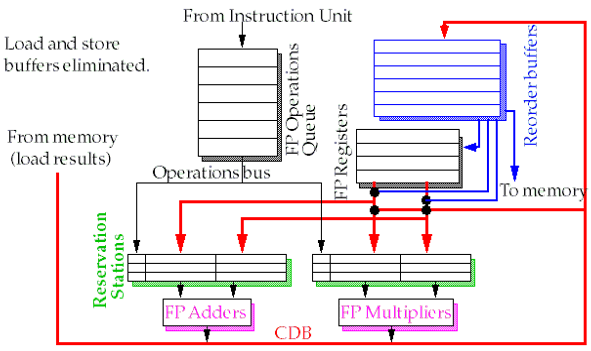
投机的key idea

将instruction完成与data 的register写回分离

投机计算允许，但instruction提交受限（除非确认为非投机行为）

乱序计算允许，但只能顺序提交

添加reorder buffer（ROB）结构（完成的instruction在此存储，并不直接将结果写入register file中）；因为添加了该结构，可以取消store buffer的存在（都是对register file的处理）



ROB含有

Instruction （type）（主要）

Destination field（主要）

Value field（主要）

Ready field

Exception vector

用ROB记录位次代替rename的register

From memory信息单独成表

ISSUE阶段的发射需要额外判断ROB是否有slot空闲，且需要相应修改ROB destination（结果）处暂时不写

Write back阶段将结果写回ROB中而不修改register files

（新增）Commit阶段，ROB中对映结果写入register file中并清除ROB相应记录

对于ROB底部的instruction

若为non-branch instruction，写回register并remove ROB中记录，对于store指令，修改memory

For mispredicted branch，ROB all flushed and从正确的instruction位置开始重新运行

For right-predicted branch按正常instruction结束即可

使用ROB后，need buffer to keep track of all outstanding stores to memory

需要记录store写回memory的值，因为顺序提交可能延迟写回，但乱序执行需要提前使用该数据，否则memory在新一轮load时会发生冲突；每一条load指令则需要记录store buffer中对映于其的记录起始位置



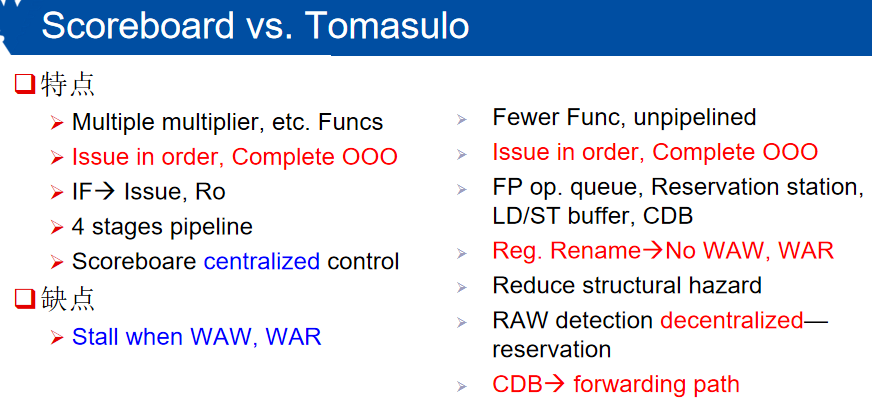
Register rename的key insight

Allocate a new physical destination register for every instruction that writes（类似于static single assignment SSA --- compiler功能）

Rename相当于维护了一个translation table，进行physical register mapping，当register written时，replace every entry with new data

优点：allow data to be fetched from single register file，解耦调度与rename

两个算法对比



Rename的支持

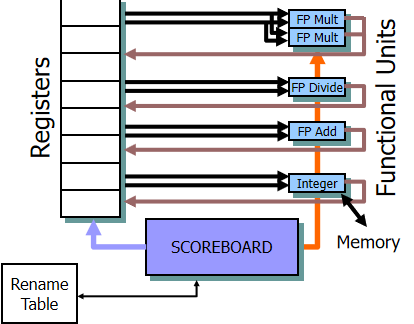
Physical register file（维护free-list，需要取用，null则stall）

Rapid access to table of translations

维护free register file可以不需要reservation stations

表格记录当前寄存器rename对映，记录free-list，记录正在执行instruction对rename map table的修改（便于checkpoint/interrupt触发后的undo恢复）

Scoreboard与rename table的联合使用

在issue阶段allocate new physical register for result（no free register时stall）

Control hazard

Branch frequency

Prediction accuracy

Misprediction penalty

1 bit branch-prediction buffer

使用branch history table 使用一位进行branch的预测，记录上一次branch是否发生

2 bit branch-prediction buffer

只有连续的两次预测错误，才会改变预测结果

Hysteresis：滞后

第一类：

T

T

NT

Predict Taken

Predict Not

Taken

Predict Taken

Predict Not

Taken

T

NT

T

NT

NT

第二类：

T

T

NT

Predict Taken

Surely

Predict Not

Taken

Predict Taken

Predict Not

Taken surely

T

NT

T

NT

NT

可以由2bit predictor扩展至n bit predictor，根据数值大小进行预测判断（大数值对映branch taken），每次taken branch + 1，每次not taken branch - 1即可

Predict depend on previous behavior and other branches

Correlating predictor

（n,m）表示m bit预测器，预测器向前看n步，共有2^n个预测器

同样使用2bit，如果预测错误，仍然需要通过修改进行补偿

第一个bit用于prediction if previous is not taken

第二个bit用于prediction if previous is taken

Mispredict cause

Wrong guess for the branch（错误猜测）

Got branch history of wrong branch when index the table.（获取到错误的预测分支）

Tournament predictors

使用两个predictor，一个based on global information，一个based on local information；使用一个selector在global与local预测器间作选择

Global predictor （12,2）

Local predictor分两层设计

Top level （local history table）

记录1024个条目的最近10次branch信息

Next level （10,3）

3 bit saturating counter（3bit饱和计数器）---> 一半一判断

Branch target buffer（BTB）

不仅预测分支是否跳转，还获取分支预测PC地址相应的 address(instruction)

一般predicted PC只会记录预测需要跳转的具体指令

CAM content addressable memory 内容寻址内存

AM associate memory 关联内存

Return address stack 使用small buffer预测return address

Prediction for direct jump

CPI为1仍不满足，希望使用multiple issue processor（多处理器）来更一步降低CPI

Vector processing（使用vector计算loop）

Superscalar（超标量）：varying number of instructions/cycle

允许执行多条指令，每个时钟周期处理的指令数可以变化

(Very) Long Instruction Words（VLIW）

采用固定数量的指令（每个执行周期处理的指令数量是确定的，是一个固定的值），指令调度由编译器完成

Static Superscalar（静态超标量）:

在编译时确定指令的执行方式，编译器负责静态调度，将多条指令同时发射到执行单元，以提高并行执行的效率。

Dynamic Superscalar（动态超标量）:

其中在运行时决定指令的执行方式。与静态超标量不同，动态超标量使用硬件中的动态调度逻辑，可以根据运行时的情况决定在某个时钟周期内执行多少条指令。

Speculative Superscalar（推测性超标量）:

采用预测执行的技术，以便在预测正确的情况下提前执行指令，从而提高指令级并行性。

VLIW/LIW (Very Long Instruction Word/Long Instruction Word):

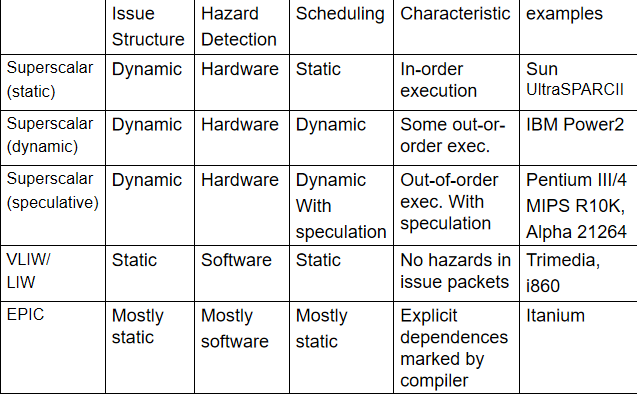
VLIW 和 LIW 都是指令级并行（ILP）的处理器架构。在这种架构中，一条指令可以包含多个操作，这些操作在同一时钟周期内同时执行，以充分利用硬件并行性。

VLIW 通常由编译器静态调度，而 LIW 则可能由硬件或编译器进行动态调度。

EPIC (Explicitly Parallel Instruction Computer) / IA-64:

采用VLIW设计，强调明确支持指令级并行性。IA-64使用了一种称为EPIC架构的指令集，旨在通过硬件和编译器的协同工作来实现高级别的并行执行。

Comparison：



Superscalar可以选择Statically / Dynamically scheduled（转移语句不能并发）

Statically scheduled往往对映 in-order execution（可灵活选择发射指令的数量）

Dynamically scheduled则对映 out-of-order execution

当使用静态调度时，all pipeline hazard checked at issue time

Issue packet，可能在一个clock cycle中发射的instruction group

以每clock cycle两个时钟周期为例，一条为FP instruction，一条为anything

在64bit fetch时，FP instruction往往在右

第二条指令的发生受到第一条指令的限制

浮点寄存器更多的写入以及输出端口

Challenges

1. issue（同时发生）要求clock cycle time ---> O(n2-n)

用于issue check（是否可以发射/是否存在冲突）

Need multiple buses / need multiple forwarding paths

Rename same register multiple times in one cycle

当使用动态调度时 --- 运行时调度

Overcome the issue restriction

Pipeline：run one step in half a clock cycle

Widen issue logic：Build the logic necessary to handle two instructions at once

Integer function unit常常成为bottle neck

Separate integer function units

ALU计算

Branch condition计算

Effective address计算

预测可加速；ld / sd类型指令执行memory地址计算和存/取memory可以separate考虑

Software approaches

Loop unrolling循环展开

无法确定确切的n，设定k --> n mod k进行相应的unrolled loop操作

交换指令顺序，弥补寄存器的改变量

可以时重命名寄存器

分析内存地址的使用不冲突

可能导致（缺陷）：

growth of code size

Potential shortfall in registers（展开的过多可能导致寄存器的数量不足）

static branch prediction（静态分支预测）--- 编译器时调度

Static multiple issue（VLIW）--- 发发射长度固定，没有并行指令时浪费

explicit coding for multiple operations，每个 VLIW 指令一次性执行多个操作

Need compiling technique that schedules across several branches

编译器负责生成VLIW指令，确保每个操作都是独立的，因而处理器不需要在运行时检测和解决数据依赖性

VLIW的problem

Code size increase

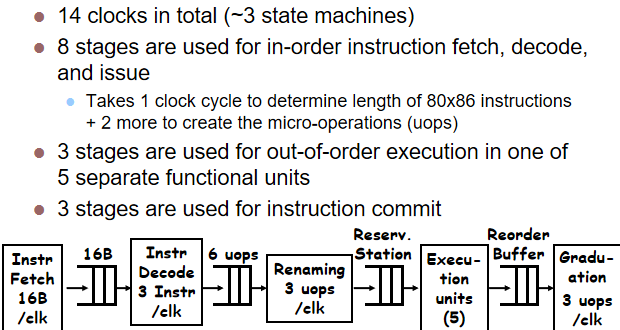
A stall in any function unit may cause the entire processor to stall

Advanced Compiler Support for Exploiting ILP

Detecting and Enhancing Loop-level Parallelism

Global Code Scheduling

P6 pipeline



Multiple thread

使用thread来提升并行度

Each thread has unique “PC + register + stack”

因而需要准备PC（copy）以及register（copy），其余计算单元只有一份

Fixed interleave

固定的发射率，每N个cycle轮到一次instruction执行

Software-controlled interleave

根据计算需求，合理分配S slot，并不完全平均发射instruction

Hardware-controlled thread scheduling

根据硬件优先级进行轮换执行（只有cache miss、运算delay时才发射其他thread instruction）

Tera MTA （细粒度（fine-grained） --- 每个周期进行切换）

拥有256个processor，每个processor拥有128个active threads，no data cache（结合后续分析）

因为128个active thread，所以需要大量的register（copy），branch-target register（用于预测的寄存器），general-purpose register，stream status word register（流状态寄存器）

使用short VLIW（一般包含三条instruction）

One memory operation

One arithmetic operation

One arithmetic or branch operation

含有look ahead机制，用于表明与当前发射的instruction是否存有冲突（dependence）

常常处理low locality的data set，因而data cache的使用意义不大，每次基本需要从memory中读取数据，因而发生hit miss时，可以有大量的thread用于切换（hide large memory latency）。

其他application

IBM Power RS64-IV（粗粒度 --- 几个周期进行一次切换 --- coarse-grained）

可以借助locality，因而data cache有效

Each physical CPU support two virtual CPU，在L2 cache miss时，pipeline flushed and execution switch to second thread

超标量机器（super scalar machine efficiency）

Completely idle cycle --- vertical waste（空issue，miss或等待计算结果）

Partially filled cycle --- horizontal waste（每次issue没有占满width）

Second thread可以消除 vertical waste

措施：

可以将width一分为二，分别给一个thread使用，从而一定程度降低vertical waste和horizontal waste，但是但thread的最大运行速率下降（要求：Any single thread can utilize whole machine）

Idle情况



硬件支持，CPU数量以及Register file数量（计算硬件的计算阶段也可拆分）

Trade-off between fine-grained implementation performance（细粒度实现性能） and single-thread performance.

Ensuring that Cache and TLB conflicts（不同thread间冲突） do not cause performance degradation（衰弱）

Pentium-4 hyperthreading（超线程）

SMT：Simultaneous Multithreading

Without超线程运行时，需要最小限度的削弱单线程运行speed

Trace cache：instruction cache

Two logical processors arbitrates（仲裁，判断） access to TC every clock cycle

If one logical processor stalled, then other logical processor can use the full bandwidth of TC

Front-end 根据

Back-end：out-of-order 的计算与instruction的commit

需要考虑register rename（two register alias tables -- RAT）、instruction scheduling、execution unit、retirement（计算结果的写回）、allocate（分配buffer及硬件资源）

执行时分为MIAQ（memory instruction address queue）与GIAQ（general instruction address queue）

拥有五个调度器调度不同类型的微操作，MIAQ和GIAQ都以最快的速度向五个调度器发送微操作（micro-ops allocate、rename后分别进入MIAQ、GIAQ后续被发生进入调度器，再进行执行计算）

Re-order buffer（ROB）也被partition，use half entries

Retire（写回）操作在两个logical processor间交替操作

不同logical processor的请求不同且独立，没有优先级的设置，均为先到先得

Memory subsystem

DTLB（Dual Translation Lookaside Buffer）--- 双重转译查找缓冲器

包含两个独立的缓存（对映memory操作与一般运算操作），用于介导虚拟地址到物理地址的转换

三级cache（logical processors share all entries in three level cache）

CMP（chip multiprocessing） n cores per processor（一个处理上多个核独立运行）

HMT（hardware multi-threading）n threads per core

CMT（chip multi-threading）n\*m threads per processor（感觉是CMP与HMT的结合）

Increase core die area by 20%，improve performance by 50 - 100%

T1 architecture

Priority given to least-recently-executed thread（最近最少执行最高优先级）

Cache miss、trap、load、branch等都会导致thread not-ready-to-run

1. cache：1 tag 2 valid-bit，每个周期取两条指令

MUL/DIV功能单元只能被单独使用

Request priority：instruction miss > instruction date miss > stores, {fpu, strm, interrupt}

只支持最多悬挂64个中断

Data level parallelism

Vector processor

GPU

SIMD可以利用显著的数据级并行

Each result independent of previous result

Few instruction fetches

hardware does not have to check for data hazards

Vector instructions that access memory have a known access pattern（有已知的访问内存模式/规律）

no (data) caches required

Type

Memory-to-memory vector processor：all vector operations are memory to memory

Vector-register processor：只有register上的数据才可进行运算

Vector memory-to-memory architecture

需要更大的main memory bandwidth

Need check dependencies on memory address

C1 feature

No data caches

No virtual memory

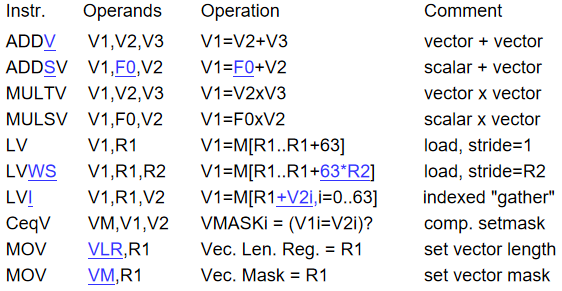
Vector register：at least 2 read and 1 write ports

Vector functional unit：fully pipeline，start new operation every clock

Vector load-store units：Multiple elements fetched/stored per cycle

Interleaved memory system（交叉存储系统，多个单字存储体构成）

Basic vector instruction

Length：用于控制vector进行计算的长度

Mask：用于选择vector进行计算的分量

Support for various combinations of data widths in memory（各种数据类型的组合）

Addressing type

Unit stride（单位步幅）：每次访问都移动到下一个相邻的内存位置。

Non-unit stride：当访问内存时，步幅不是单位步幅，即每次访问移动到内存中的不相邻位置。

Indexed（Gather-Scatter）：使用索引数组（index array）来访问非连续内存位置，其中索引数组存储了要访问的元素的位置

Maximum vector length（MVL）：需要参与计算的vector最大个数

suppose application vector length > MVL，需要进行strip mining（整除与取余）

Optimization

vector chaining

在vector计算中也使用forwarding（一个vector**分量**计算完成后就向后传递）

Conditional execution

Vector分量间的比较直接得到分量间是否需要进行计算的结果

Simple implementation：计算所有的vector分量，选择实际需要进行计算的分量结果

Density-time implementation：only execute elements with non-zero masks

Mask也可以用于compress（压缩）与expand（扩展）

Sparse matrices（稀疏矩阵）便于循环矢量化

Multi-lane implementation 多通道实施

使用pipelined datapath、functional unit（多个计算单元，full连续计算）

编译器视角（compiler perspectives）

Inner loop vectorization（内环矢量化）

计算2个64 bit间的操作

Outer loop vectorization（外环矢量化）

计算64个2个1 bit间的操作

需要独立、大量的blank支持不同的access

Vector advantage

High performance

Scalable：adding hardware resources can get higher performance

Compact：instruction简化

Vector pitfalls

Concentrating on peak performance and ignoring start-up overhead（专注于峰值性能可能忽略启动的开销）

Good processor vector performance without providing good memory bandwidth

GPU

CPU是host，GPU是设备

Thread is associated with each data element

Threads are organized into blocks

Blocks are organized into a grid

Grid - thread block - SMID block

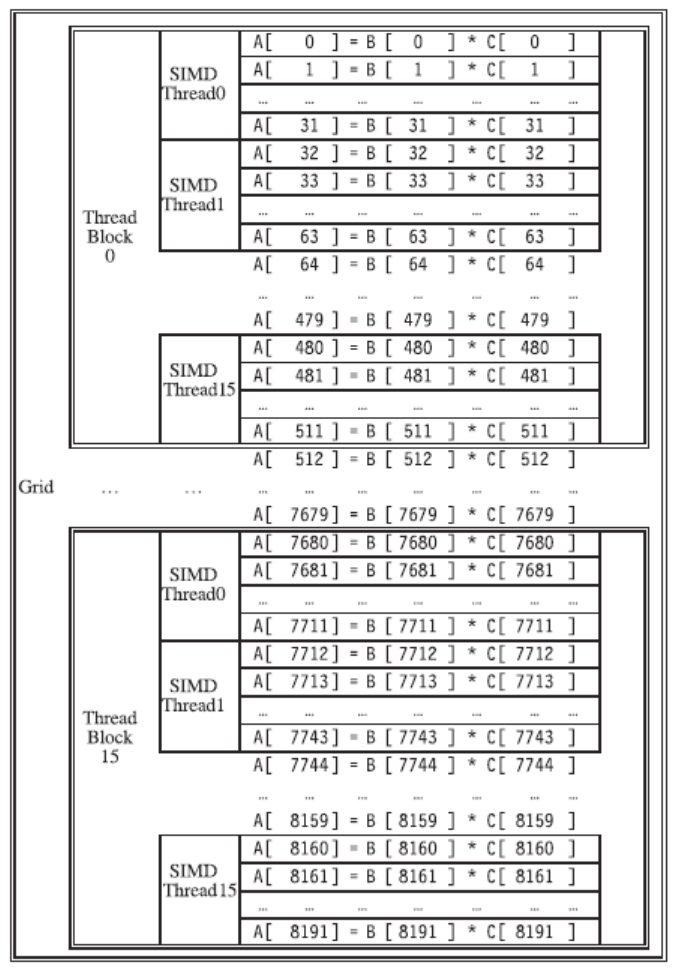
GPU运行的example：

使用mask进行结果的有效提交，（条件判断要执行的block执行计算，不需要执行的block等待，共同进入下一阶段计算）

Each SIMD Lane has private section of off-chip DRAM

Each multithreaded SIMD processor also has local memory

Memory shared by SIMD processors is GPU Memory



Loop-level parallelism（loop级别的并行）

循环展开 --- 查看循环间依赖关系（loop-carried dependence）

Find dependencies

GCD test：If a dependency exists, GCD(c, a) must evenly divide (d-b)

定义a x j + b = c x k + d

Reduction可以更大限度的提升计算性能

Pitfalls

Concentrating on peak performance in vector architectures and ignoring start-up overhead

Increasing vector performance without comparable increases in scalar performance

You can get good vector performance without providing memory bandwidth

On GPUs, just add more threads if you don’t have enough memory performance

Cache 一致性

A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.

Flynn’s taxonomy of parallel machines（富林并行机分配法）

SISD：single instruction single data (uniprocessors)

MISD：multiple instruction single data (not used much)

SIMD：multiple instruction single data（指令相同数据不同）

MIMD：multiple instruction multiple data（自己指令自己数据）

在多处理器架构中，以存储器为中心有两种实现方案

Shared类型（使用同一个大memory）

内存访问时间一致

共享内存，直接通过内存进行进程间通信

Distributed类型（不同processor使用不同笑memory类型）

内存访问时间差别较大

消息传递，通过消息进行进程间通信

Centralized shared memory ：SMP（symmetric multiple processor） / UMA（unified memory access）

数据位置与存取latency无关

仅支持小型多处理器（少量处理器）

Simpler programming model、more complex hardware



Distributed memory

Distributed shared memory：DSM / NUMA

数据存放位置很关键

Local memory往往有较高的存取速度

会有更多的communication latency，software实现也更加复杂

Simple hardware、difficult programming model

应用于需要使用大量处理器的架构

逻辑地址统一，但是物理分布式存储



Multiple computer

地址空间由多个private地址空间构成，processor之间不能相互访问

NOW（network of workstation）由局域网连接nodes（不同PC）构成

Memory shared model

一致性与保护问题需要解决

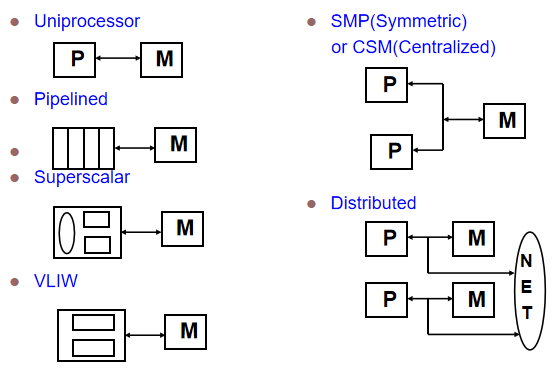
扩展性较差

Message passing model

Send、receive、synch等操作

Larger NUMA 往往由 smaller UMA组合得到

总结图



只增加处理器的个数，也会受到阿姆达尔定律的限制

Remote reference往往会带来很大的latency

尽量发挥多处理器性能

使用cache并解决cache coherence问题

在software层面尽可能local的使用data

通过切换任务、重叠通信消息，隐藏部分latency

Cache一致性的解决

Shared-memory architecture

Limited processor node（只能支持一定数量的处理器）

Large cache（有更大的memory bandwidth）、更小的取用latency

DMA（direct memory access）可能导致cache的一致性出问题（cache与memory中数据不一致）

Write back、write through（考虑同一个memory不同cache的情况，仅写cache会与其他cache以及memory产生冲突，write through会与其他cache产生冲突）

写入的时机也很关键

处理策略：

Snooping solution

Works well with bus

在small scale machines上效果较好

在cache中添加了一cache controller模块，snooping address tag（cache不仅记录块数据，还记录块数据对映的共享状态）

两种子策略：

Write invalidate protocol

Single writer，写入冲突的cache需要将相应数据位设置为invalid

Write broadcast protocol（typically write through）

将最新的写入结果写到所有含此数据的cache与memory中

Write serialization：总线需要序列化请求

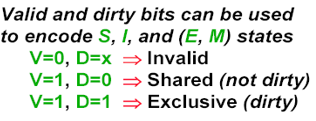
使用状态来协调不同cache、bus与memory之间的data传递策略

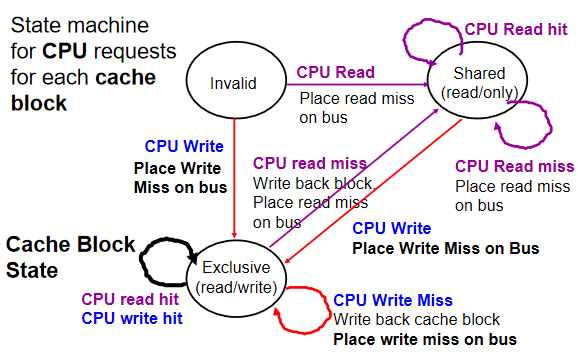
一般而言借助valid和dirty位实现

Invalid状态需要将对映的新写入（同名）数据置为无效

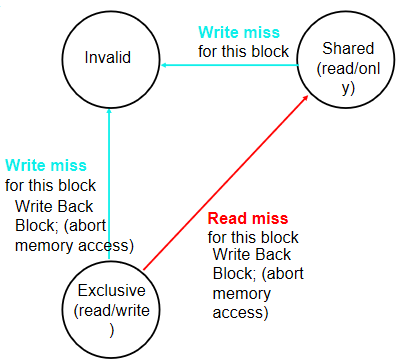
Shared状态可以将数据共享

Exclusive状态可以写入新数据



Cache根据CPU操作的状态变化表

根据从总线接收到的信息，cache状态可以发生如下转变

状态的转换关系需要理解清楚，CPU read/write表示当前cache的操作结果与状态转变；read/write miss表示其他cache发布在总线上的信号

Directory-based schemes

一般用于distributed memory，有更好的延展性

需要使用directory 追踪memory中每一个block的state状况

States of every block（三种状态，shared、uncached、exclusive）

Shared：至少一个processor使用此块数据

Uncached：cache中数据无有效使用

Exclusive：仅有一个processor占有data块（owner -- 修改）

其他哪个processor占有对映数据块的副本

Block是否dirty

Directory size = entry number \* entry size

Directory protocol中所有message都会收到响应

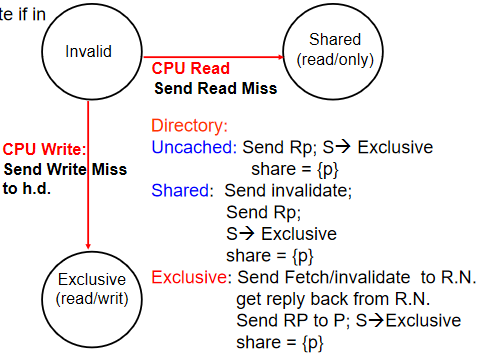
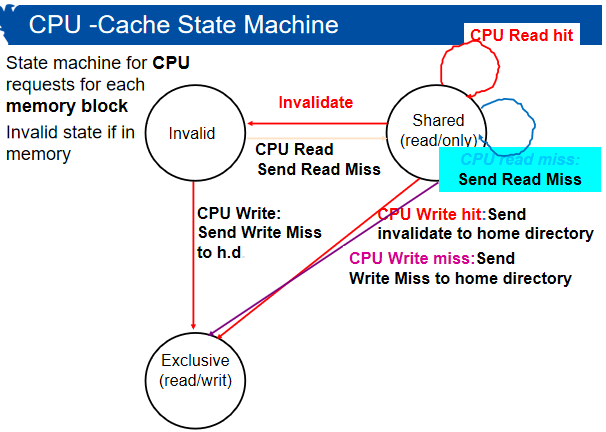
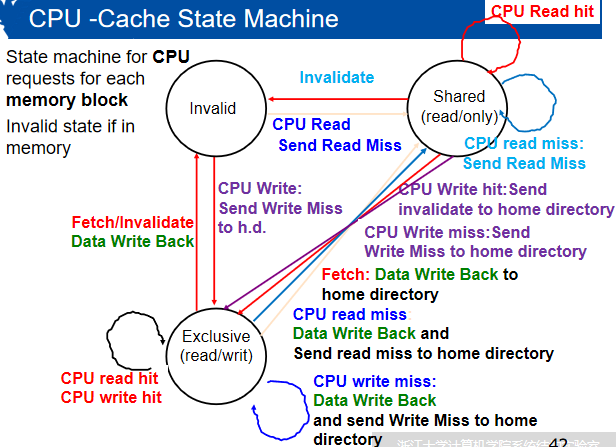
其中有三类node：

Home node：请求数据块的memory存放processor

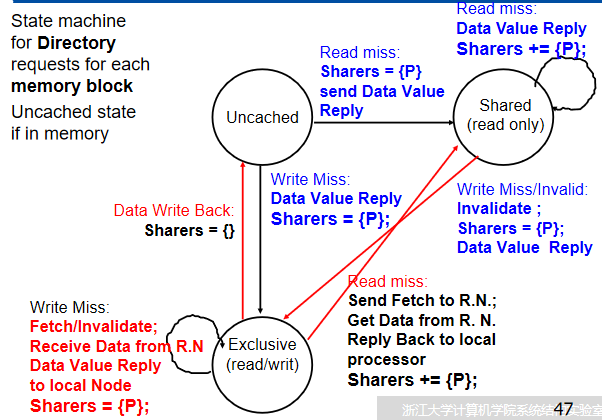
Local node：请求提出processor

Remote node：用于数据块copy / owner的processor

状态转变表



Directory的状态变换

对于owner而言

Read miss ---> fetch

Write miss -->fetch / invalidate

Invalidate 发送给 share（ 有write介入时 ）

Write/read miss都会收到home产生的一个reply的原有数据

在实现directory-based coherence时也会存在一些问题，我们需要作一些简化

Point-to-point in-order delivery

Unlimited buffering（按顺序记录所有的request并逐一处理）

在信号竞争时，需要使用额外的信号发送，表明新一轮的实际操作对象

Write miss时的data reply也是一种机制

Message deliver within finite time

可能产生死锁，因而需要 duplicate coherence controller for each block（需要额外的数据结构去存储所有cache的state信息）

因为操作的原子性，需要互相等待释放资源以完成上一轮产生的事件

解决方法：

1.让resource总是可以被释放而获得的

将request与reply网络分离

每一个request都需要一个reply

Controller可以直接使用NAK信号拒绝request，但不能使用NAK拒绝reply

任何request接收到NAK后可以重试

1. directory controller must be reentrant（可重入的）

在上一个数据块操作未完成时，允许下一request数据块进入

Fetch无效时control state需要保持状态

Owner node可以直接为request传递数据

对NAK的处理：

将original request装入NAK中，可以使NAK具有鉴别性

Buffer不仅可以保存reply也能存储request相关信息

同步（synchronization）

Atomic exchange

交换寄存器与某一内存地址（充当lock）的数据内容

一般而言，1表示synchronization variable locked

0表示free的synchronization variable

c7b0a1e3f511da993898842fe2f53c9

将R1 + 0地址的数值与寄存器R2的数值进行交换

Test and set：tests a value and sets it if the value passes the test

3cd5a8d612f9893bff6d4c0cc02d3c0Fetch and increment：it returns the value of a memory location and atomically increments it

Test and set与exchange往往会造成大量的性能消耗 --- 不断轮询对shared data的read与write

Load linked and store conditional

Load linked获得initial value，若期间initial value未被其余processor改变，则store conditional （本身会将某个数值存储到内存中的对映地址）返回1，否则返回0；

Spin locks（自旋锁）: processor continuously tries to acquire, spinning around a loop trying to get the lock，注意只是描述对lock的获得与否进行不断问询

每次只能一个processor对lock进行获取，而lock每次仅能位于一个processor的cache中，因而会spin locks会导致大量的read miss与write miss.

Barrier：障碍，一种同步机制，直至所有processor均到达barrier时，才能release all the processes to go forward.

主要的performance提升方式为减少latency时间

增加轮询间的delay，使轮询的发生不那么频繁

也可以使用queuing lock，只有processor自己的turn才尝试去获得锁，且每个processor对同一个锁使用不同的address，或者Transfer the lock from one process to another explicitly.

Queuing lock

显式的将锁从一个processor传递给另一个processor

Performance：n + 1 + 2\*(n - 1) [n次first miss，一次释放，后续n-1个processor的获取与释放]

一般的performance （2n + 1）的求和[一次read miss，一次write miss，一次release]

Fetch & increment 使用3n [n read miss for count，n read miss for release，n fetch&increment]

Memory coherence（内存一致性）

每个processor中是in order计算的，但在不同processor中是interleave（可交错）计算

使用synchronization operations进行shared data 的ordered operation

Relaxed consistency model：设定read和write在一定架构中是可以乱序执行的，但每个隔间作为一个同步点，隔间限制processor（类似于lock的机制），但processor可以既乱序又交叉执行

SC（完全顺序）：read write正常顺序一一执行

TSO（total store order）：在隔间中，允许不相关的read write乱序执行（只有read和write之间）--- IBM

PSO（partial store order）：不相关的先后write也可以乱序执行 --- sparc

WO（weak ordering）：均可乱序（隔间外的read与write），没有限制（隔间为同步点） --- PowerPC

RC（release consistency）：自由度更高，read write操作开始结束不受acquire与release的影响 --- Alpha，MIPS