Contents

I Introduction

- Program
- OS
- CA
- organization
- digital logic

从单核到多核, 了解多核系统工作方式.

II Fundamentals-Basics

1. RISC Architecture

Reduced Instruction Set Computer
Present two critical perf techniques:

1) instruction-level parallelism (pipelining and multiple instruction issue)
2) caching

1.1 Pipelining

Divide instruction execution into stages
Overlap execution of multi-instruction (CPI>1)

1.2 Multiple Instruction Issue

Deploy multiple datapaths
Complete more than one instruction per clock cycle (CPI may <1)

1.3 Caching

faster temporary storage

1.4 Dennard Scaling

Power density is constant for a given area of silicon even as you increase the number of transistors because of smaller dimensions of each transistor
失效了

1.5 Moore's Law

The nubmer of transistors per chip would double every year (every two years later).

1.6 Multi-Core Processor

Multiple efficient processors. From instruction-level parallelism to data-level and thread-level parallelism.

1.7 Amdahl's Law

Make common case fast

$$\begin{aligned} 1 &= \text{Fraction}_{\text{enhanced}} + \text{Fraction}_{\text{last}} \\ \text{Speedup}_{\text{overall}} &= \frac{\text{Execution tmie}_{\text{old}}}{\text{Execution time}_{\text{new}}} \\ &= \frac{1}{\text{Fraction}_{\text{last}} + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} \end{aligned}$$

2. 5 Classes of Computer

1) PMD: Personal Mobile Device
2) Desktop Computing
3) Servers
4) Clusters/WSCs
5) Embedded Computers

3. Parallelism

application parallelism and hardware parallelism

3.1 Application Parallelism

- DLP: Data-Level Parallelism
- TLP: Task-Level Parallelism

3.2 Hardware Parallelism

four ways:

- ILP: Instruction-Level Parallelism
  exploit data-level parallelism
  - pipelining
    divide a task to steps;
    simultaneously run different steps of different tasks
  - speculative execution
    do some work in advance;
    prevent delay when the work is needed
- Vector Architectures, GPUs, and Multimedia Instruction Sets(MM)
  exploit data-level parallelism;
  apply a single instruction to a collection of data in parallel;
- TLP: Thread-Level Parallelism
  exploits either DLP or TLP,
  in a tightly coupled hardware model that allows for interaction among parallel threads;
- RLP: Request-Level Parallelism P65
  exploits parallelism among largely decoupled tasks specified by the programmer or the OS

3.3 Classes of Parallel Architectures

Single/**M**ultiple **I**nstruction/**D**ata stream

- SISD: Exploit instruction-level parallelism
- SIMD: The same instruction is executed by multiple processors using different data streams. Exploits data-level parallelism.
- MISD: hard to exploit data-level parallelism
- MIMD: Each processor fetches its own instructions and operates on its own data. Exploits task-level parallelism

4. Instruction Set Architecture (ISA)

7 Dimensions of ISA:

- Class of ISA
- Memory addressing
- Addressing modes:
- Types and sizes of operands
- Operations
  - Data Transfer
  - Arithmetic/Logical
  - Control
  - Floating point
- Control flow instructions
- Encoding an ISA

remember RISC-V Register Naming

III Fundamentals-Performance

1. Trend

energy workload
daynamic Energy
daynamic Power
static Power
Dies per Wafer
Cost of Die
Die Yield
Cost of Integrated Circuit
Price
Transient/permanent faults become more commonplace

2. Dependability

SLA: service level agreements
SLO: service level objectives
System states: up or down
Service states

- Failure: 仅当 actual behavior 偏离 actual behavior 时发生.
- Error
- Fault

2.1 Measures of Dependability

Module reliability: A measure of continuous service accomplishment (or of the time to failure) from a reference initial instant.

- MTTF: mean time to failure
- MTTR: mean time to repair
- MTBF: mean time between failures
- FIT: failures per billion hours

MTBF = MTTF + MTTR

Module availability=

2.2 *Dependability via Redundancy*

3. Disk Arrays - RAID

4. Measure Performance

IV Memory Hierarchy Design Basics

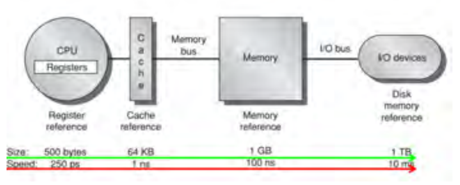


Figure IV.1: Memory Hierarchy

1. Technology & Optimizations

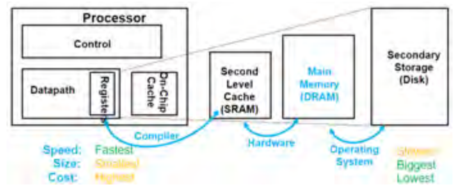


Figure IV.2: Main Memory

Performance measures

- Latency: the time to retrieve the first word of the block
  - access time: 从请求到数据到达的时间
  - cycle time: 从一个 access 开始到另一个 access 开始相距的时间
- Bandwidth: the time to retrieve the rest of this block

1.1 *SRAM for cache*

Static Random Access Memory(SRAM)

- Direct Mapped Cache
- Set Associative Cache
- Fully Associative Cache

信息 bit 为需要根据具体情况确定.

e.g 16KB direct mapped cache 256byte blocks. Assume 1words=8bits

use $\log_2 256 = 8$-bit offset, $16kB/256B = 64$ lines, $\log_2(64) = 6$-bit line index.

1.2 *DRAM for main memory*

- Dynamic Random Access Memory(DRAM)
- Single transistor per bit
- Reading destroys the information
- Refresh periodically
- cycle time > access time

DRAM Access Example

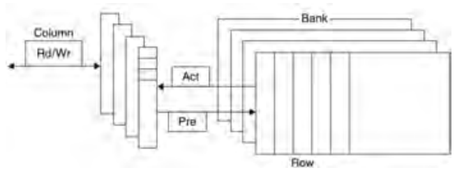


Figure IV.3: DRAM Organization

1.3 DRAM Improvement

- Timing signals
- Leverage spatial locality
- Clock signal
- SDRAM
- Wider DRAM
- DDR: double data rate
- Multiple Banks
- Reducing power consumption in SDRAMs

2. Cache Performance

Average Memory Access Time = Hit Time + Miss Rate x Miss Penalty

Six Basic Cache Optimizations:

1) larger block size
2) bigger caches
3) higher associativity
4) multilevel caches
5) giving priority to read misses over writes
6) avoiding address translation during indexing of the cache

Root Causes of Miss Rates:

- 冷启动/首次未命中
  Compulsory: cold-start/first-reference misses
- Capacity: cache size limit
  occur upon the cache is full
- Conflict: collision misses
  occur without the cache being fully occupied

2.1 Larger Block Size

- Reduce compulsory misses
  Leverage spatial locality
- Increase conflict/capacity misses
  Fewer blocks in the cache
  Larger blocks increase miss penalty

2.2 Larger Cache

- Reduce capacity misses
- Increase hit time, cost, and power

2.3 Higher Associativity

- Reduce conflict misses
- Increase hit time

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

2.4 Multilevel Cache

Reduce miss penalty

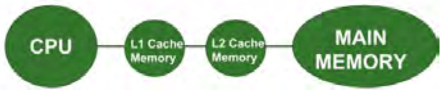


Figure IV.4: Multilevel Cache

Two-level cache: Add another level of cache between the original cache and memory.

- L1: small enough 来匹配处理器的 clock time
- L2: large enough 来存储要去主存的访问, 减少 miss penalty

Local miss rate and Global miss rate
Multilevel inclusion and Multilevel exclusion

2.5 Prioritize Read Misses Over Writes

Reduce miss penalty
读写操作分开对待. check the contents of write buffer, let the read miss continue if no conflicts with write buffer & memory system is available

2.6 Avoid Address Translation During Indexing Cache

Cache addressing

- virtual address – virtual cache
- physical address – physical cache

Processor/program – virtual address
Processor → address translation → Cache
Virtually indexed(TLB), physically tagged
并行执行虚拟地址的转换与地址查找, 提高缓存命中的时间.
e.g.

V Memory Hierarchy Design Advances

1. Ten Advanced Cache Optimizations

Goal: average memory access time down.
Metrics to reduce/optimize:

- hit time
- miss rate
- miss penalty
- cache bandwidth
- power consumption

1.1 Small and Simple First-Level Caches

Small size: support a fast clock cycle and reduce power
Lower associativity: reduce both hit tme and power

1.2 Way Prediction

Reduce conflict misses and hit time.

block predictor bits are added to each block to predict the way/block within the set of the next cache access. Used in a cache with more than one block in a set.

1.3 Pipelined Access and Multibanked Caches

Increase cache bandwidth. with Higher latency and Greater penalty.

Divide cache into independent banks that support simultaneous accesses.

Sequential interleaving spread the addresses of blocks sequentially across the banks.

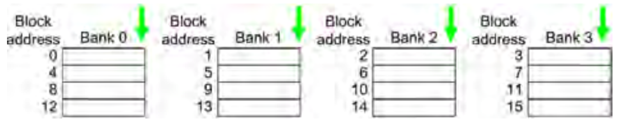


Figure V.1: Multibanked Caches

1.4 Nonblocking Caches

Increase cache bandwidth.

Nonblocking/lockup-free cache leverage out-of-order execution, allows data cache to continue to supply cache hits during a miss.

1.5 Critical Word First and Early Restart

Reduce miss penalty. Motivation: processor normally needs just one word of the block at a time.

Critical word first: request the missed word first from the memory and send it to the processor as soon as it arrives;

Early restart: fetch the words in normal order, as soon as the requested word arrives send it to the processor

1.6 Merging Write Buffer

Reduce miss penalty

Write merging merges four entries (with sequential addresses) into a single buffer entry

1.7 Compiler Optimization

Reduce miss rates, w/o hw changes

Tech 1 Loop interchange: exchange the nesting of the loops to make the code access the data in the order in which they are stored

Tech 2 Blocking: compute using submatrices after; maximize accesses to loaded data before they are replaced

1.8 Hardware Prefetching

Reduce miss penalty/rate

Prefetch items before the processor requests them, into the cache or external buffer

Instruction prefetch

1.9 Compiler Prefetching

Reduce miss penalty/rate

Compiler to insert prefetch instructions to request data before the processor needs it

- Register prefetch
- Cache prefetch

1.10 HBM

Use HBM as L4 cache(垂直空间)

2. Virtual Memory and Virtual Machine

考试不考, 摸了.

VI ILP — Static Parallelism

1. Instruction-Level Parallelism(ILP)

1.1 Basic Pipeline

An implementation technique whereby multiple instructions are overlapped in execution

1.2 Data Hazard & Forwarding

The hazard detection by comparing the destination and sources of adjacent instructions

2. floating-point pipeline

2.1 Multicycle FP Operation

allow for a longer latency for op, i.e. take >1 cc for EXE. Two changes over integer pipeline:

1) repeat EX
2) use multiple FP functional units. e.g. FP adder, FP divider

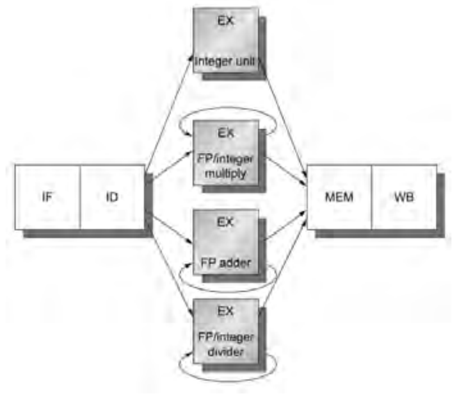


Figure VI.1: FP Pipeline

EX is not pipelined

2.2 Latency & Ini/Repeat Interval

- Latency: the number of intervening cycles between an instruction that produces a result and an instruction that uses the result
- Initiation/Repeat Interval: the number of cycles that must elapse between issuing two operations of a given type

2.3 Generalized FP Pipeline

EX is pipelined (except for FP divider). with Additional pipeline registers 但因为每条指令长度不一样, 会有乱序的问题.

Table VI.1: Latency & Ini/Repeat Interval

Functional unit	Latency	Initiation interval
Integer ALU	0	1
FP/integer loads	1	1
FP add	3	1
FP/integer multiply	6	1
FP/integer divide	24	25

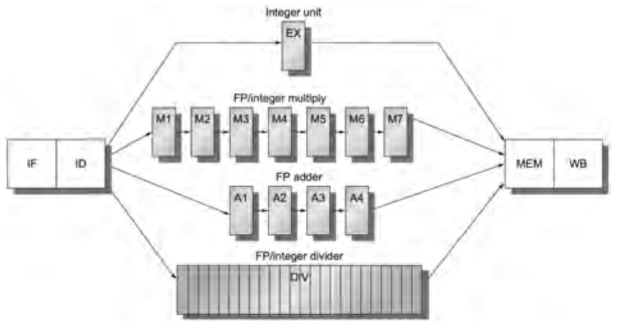


Figure VI.2: Generalized FP Pipeline

3. FP pipeline hazards

3.1 Structural Hazard

Instructions have varying running times, maybe > 1 register write in a cycle.

Solution: Interlock Detection.

3.2 WAW Hazard

Write after write (WAW) hazard: Instructions no longer reach WB in order.

Solution:

1) delay issuing
2) zero write control

3.3 RAW Hazard

Longer latency of operations, more frequent stalls for read after write (RAW) hazards

3.4 Hazard: Exceptions

Instructions may complete in a different order than they were issued -exceptions

3.5 Forwarding

Generalized with more sources EX/MEM, A4/MEM, M7/MEM, D/MEM, MEM/WB -> source registers of an FP instruction.

3.6 Out-of-Order Completion

How to deal with out-of-order:

1) ignore the problem
2) buffer the results of an operation until all the operations issued earlier complete
3) track what operations were in the pipeline and their PCs for trap-handling
4) issue an instruction only if it is certain that all previous instructions will complete without exception

4. ILP Exploitation

- Compiler-based static parallelism
- Hardware-based dynamic parallelism

5. Static Scheduling

- Pipeline Scheduling
- Loop Unrolling

5.1 Pipeline Scheduling

e.g.

```
for(i=999;i>=0;i-=1) x[i]=x[i]+s;
```

```
Loop:
    fld    f0, 0(x1)
    fadd.d f4, f0, f2
    fsd    f4, 0(x1)
    addi   x1, x1, -8
    bne    x1, x2, Loop
```

8 clock cycles per loop

```
Loop:
    fld    f0, 0(x1)
    addi   x1, x1, -8
    fadd.d f4, f0, f2
    fsd    f4, 8(x1)
    bne    x1, x2, Loop
```

7 clock cycles per loop

loop overhead

5.2 Loop Unrolling

Replicate loop body multiple times given the same amount of overhead instructions

```
Loop:
    fld    f0, 0(x1)
    fadd.d f4, f0, f2
    fsd    f4, 0(x1) //drop addi & bne
    fld    f6, -8(x1)
    fadd.d f8, f6, f2
    fsd    f8, -8(x1) //drop addi & bne
    fld    f0, -16(x1)
    fadd.d f12, f0, f2
    fsd    f12, -16(x1) //drop addi & bne
    fld    f14, -24(x1)
    fadd.d f16, f14, f2
    fsd    f16, -24(x1)
    addi   x1, x1, -32
    bne    x1, x2, Loop
```

26-CC per four ops

5.3 Loop Unrolling & Scheduling

```
Loop:
    fld    f0, 0(x1)
    fld    f6, -8(x1)
    fld    f0, -16(x1)
    fld    f14, -24(x1)
    fadd.d f4, f0, f2
    fadd.d f8, f6, f2
    fadd.d f12, f0, f2
    fadd.d f16, f14, f2
    fsd    f4, 0(x1)
    fsd    f8, -8(x1)
    fsd    f12, -16(x1)
    fsd    f16, -24(x1)
    addi   x1, x1, -32
    bne    x1, x2, Loop
```

14-CC per four ops

6. Handle Branches

Focus on a frequent critical path as if the branch were gone

6.1 Trace Scheduling

Trace: a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions

- Trace Selection: find the frequent path
- Trace Compaction: compress on-path instructions into fewer wider instructions

- trace exit

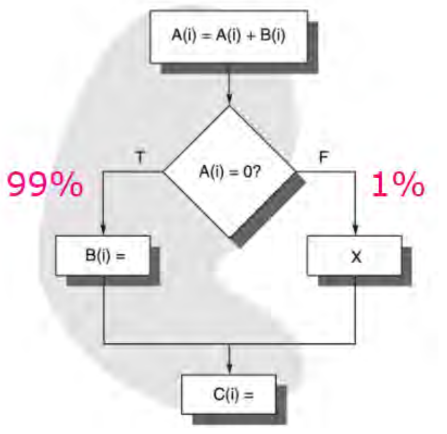


Figure VI.3: Frequent Critical Path

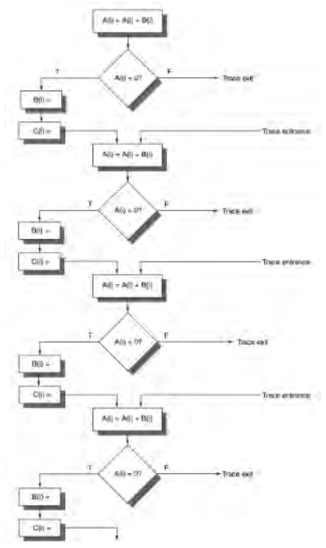


Figure VI.4: unrolled trace

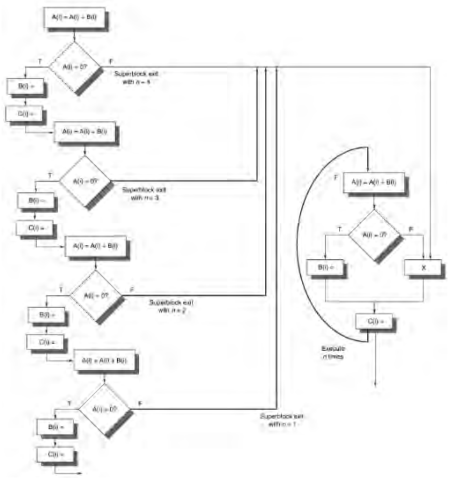


Figure VI.5: Superblock

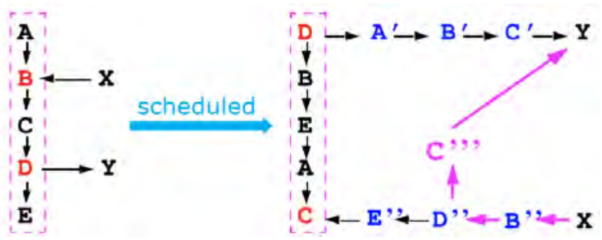


Figure VI.6: Fixup Code/Compensation Code

- trace entrance

too many entrances and exits amidst the trace

Superblock single entrance & multiple exits

Fixup Code copy instructions to ensure correctness of all paths

6.2 Branch Hazard

a branch may or may mot change PC to other values other than PC+4.

If the branch is untaken, the stall is unnecessary.

Solutions: 4 simple compile time schemes

1) Freeze or flush the pipeline: hold or delete any instructions after the branch till the branch dst is known
2) Predicted-untaken / not-taken: simply treat every branch as untaken
3) Predicted-taken: simply treat every branch as taken
4) Delayed branch: delay the branch execution after the next instruction

VII ILP — Dynamic Parallelism

1. Static Branch Prediction

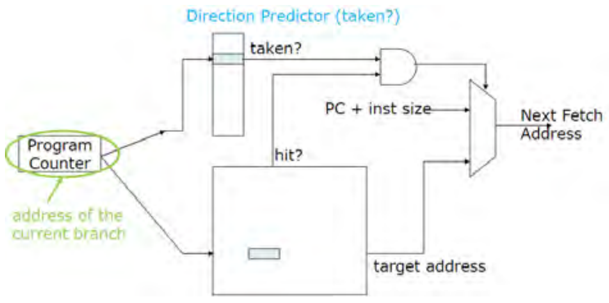


Figure VII.1: Static Branch Prediction

使用历史数据结合现在的指令进行判断, 会得到下一个预测的地址.

2. Dynamic Branch Prediction

- Follow branch history of taken branch
- Use branch prediction buffer (or BHT: branch history table) indexed by lower portion of branch address and marked with a bit to track taken/untaken status

2.1 1,2,N-bit Predictor

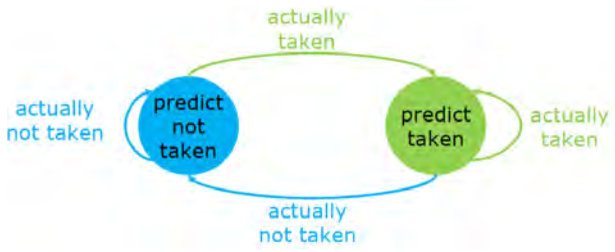


Figure VII.2: 1-bit Predictor

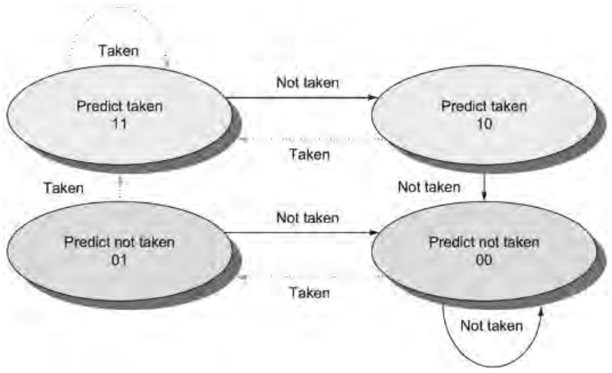


Figure VII.3: 2-bit Predictor

N-bit Predictor: assign an N-bit predictor per branch, range from 0 to $2^N - 1$. Counter threshold: $\frac{2^N-1}{2}$.

- greater: taken
- smaller: untaken

2.2 Local Predictor

A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed). 即使用最近的历史进行预测.

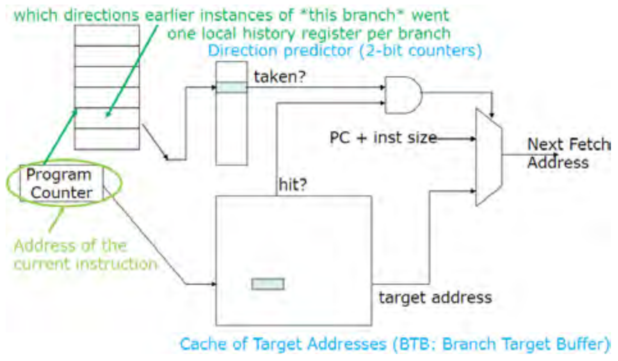

which directions earlier instances of *this branch* went
one local history register per branch
Direction predictor (2-bit counters)
taken?
PC + inst size
Next Fetch Address
Program Counter
hit?
Address of the current instruction
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

Figure VII.4: Local Predictor

3. Correlate Branch Prediction

3.1 Correlating/Two-level Predictor

use the behavior of other branches to make a prediction.

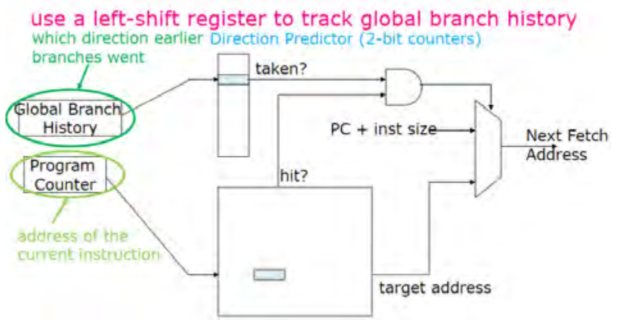

use a left-shift register to track global branch history
which direction earlier branches went
Direction Predictor (2-bit counters)
taken?
Global Branch History
PC + inst size
Next Fetch Address
Program Counter
hit?
address of the current instruction
target address

Figure VII.5: Two-level Predictor

(m, n) predictor:

- use the behavior of the last m branches to choose from 2^m branch predictors, each predictor is an n-bit counter.
- storage cost per branch given an individual pattern history table (PHT, $2^m \times n$ bits)

3.2 *Hybrid/Alloyed Predictor*

Use more than one type of predictor, Use more than one type of predictor.

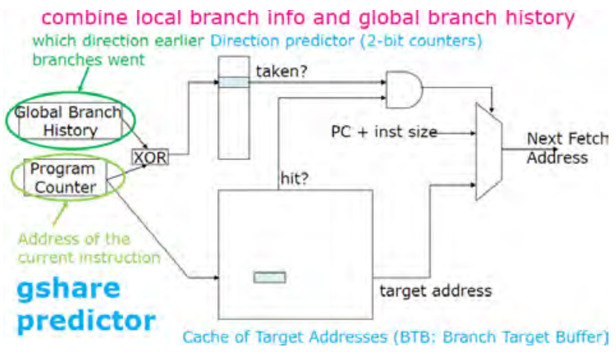

combine local branch info and global branch history
which direction earlier branches went
Direction predictor (2-bit counters)
taken?
Global Branch History
Program Counter
XOR
PC + inst size
Next Fetch Address
hit?
Address of the current instruction
gshare predictor
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

Figure VII.6: Hybrid/Alloyed Predictor

3.3 *Tournament Predictor*

- Global predictor + Local predictor
- Choose either with a selector

3.4 *Tagged Hybrid Predictor*

Use a series of global predictors indexed with different length histories

4. Dynamic Scheduling

hardware reorders the instruction execution to reduce the stalls while maintaining data flow and exception behavior

- Enable out-of-order execution
- Split ID into two stages:
  - Issue: decode instructions, check for structural hazards
  - Read operands: wait until no data hazards, then read operands

Scheduling policy: upon stalling instruction, issue other instructions if they do not depend on any active or stalled instruction.

How to schedule: use a centralized hazard detection and resolution unit

4.1 *Scoreboarding*

Scoreboard controls the instruction progression from one step to the next by communicating with functional units

Three parts:

- Instruction status:

  Indicate which of the four steps the instruction is in.

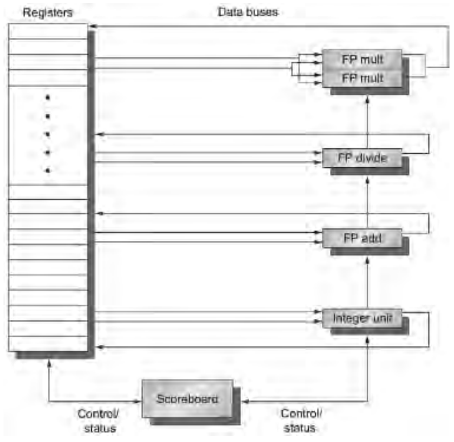
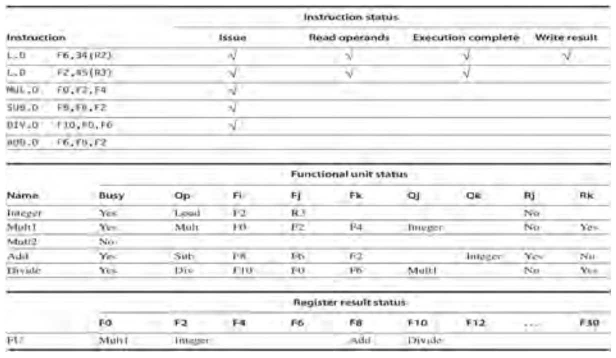


Figure VII.7: Scoreboarding



Instruction		Issue	Read operands	Execution complete	Write result
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	
MUL.D	F0,F2,F4	√			
SUB.D	F8,F6,F2	√			
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2				

Functional unit status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	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

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Integer			Add	Divide			

Figure VII.8: Scoreboarding Example

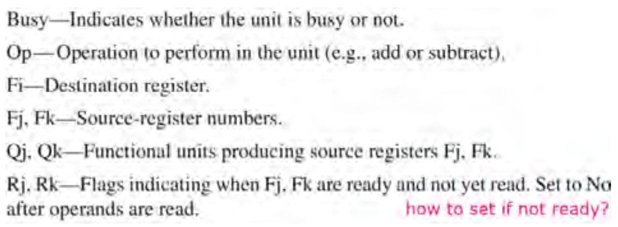
Four steps: issue, read operands, execution, write back

Omit memory access because the out-of-order focuses more on FP operations

- Functional unit status

  Indicate the state of the functional unit

  Nine fields per functional unit (FU)



Busy—Indicates whether the unit is busy or not.

Op—Operation to perform in the unit (e.g., add or subtract).

Fi—Destination register.

Fj, Fk—Source-register numbers.

Qj, Qk—Functional units producing source registers Fj, Fk.

Rj, Rk—Flags indicating when Fj, Fk are ready and not yet read. Set to No after operands are read. how to set if not ready?

Figure VII.9: Functional Unit Status

- Register result status

  Indicate which functional unit will write each register, if an active instruction has the register as its destination.

Set to blank whenever no pending instructions will wirte that register

其监控全局信息, 以支持乱序执行. 但当结构过大布线就十分困难, 有单点失效的问题, 有性能瓶颈.

4.2 Register Renaming

Elimination of stalls for WAW and WAR hazards through Register Renaming. Rename related destination registers.

```
fdiv.d f0, f2, f4          fdiv.d f0, f2, f4
fadd.d f6, f0, f8          fadd.d S, f0, f8
fsd    f6, 0(x1)      ->   fsd    S, 0(x1)
fsub.d f8, f10, f14        fsub.d T, f10, f14
fmul.d f6, f10, f8         fmul.d f6, f10, T
```

- RAW(Read after Write) Hazard: no rename
- WAR(Write after Read) Hazard: rename latter/destination
- WAW(Write after Write) Hazard: rename former

after rename destination, rename later source. 如此, 便支持了更大范围的乱序执行.

4.3 Tomasulo's Algorithm

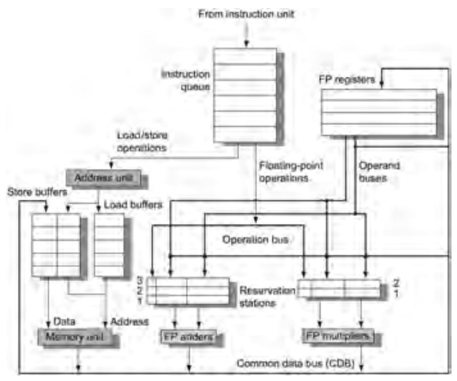


Figure VII.10: Tomasulo's Algorithm

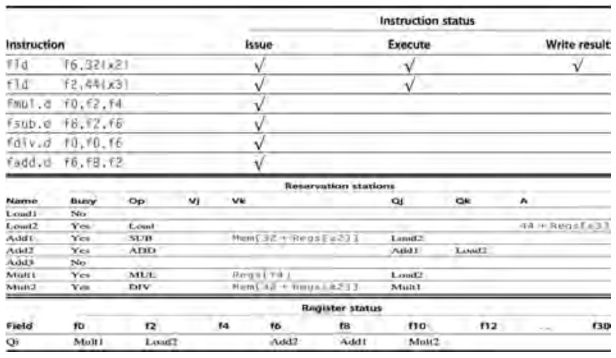
- 用 Reservation stations 存储并等待相关 functional unit 的信息. 对于一条指令, 需要等待其 source register 都 ready, 且 functional unit 完成当前的操作.
- 用 Common data bus (CDB) 链接 functional unit 输出的信息. 需要链接到 register, forwarding, memory.

Two major advantages over scoreboard:

1) Distribution of hazard detection logic across load/store buffers and reservation stations
2) Elimination of stalls for WAW and WAR hazards through Register Renaming

4.4 Hardware Speculation

out-of-order execution & in-order commit and issue



Instruction		Issue	Execute	Write result
fld	f6,32(x2)	√	√	√
fld	f2,44(x3)	√	√	
fmul.d	f0,f2,f4	√		
fsub.d	f8,f2,f6	√		
fdiv.d	f0,f0,f6	√		
fadd.d	f6,f8,f2	√		

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[x3]
Add1	Yes	SUB		Mem[32 + Regs[x2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[f4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[x2]]	Mult1		

Register status

Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Figure VII.11: Tomasulo Example

- Instruction Commit
- Reorder Buffer (ROB)

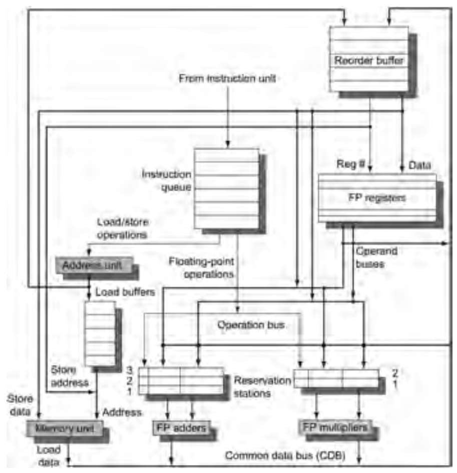


Figure VII.12: Hardware Speculation

Reorder buffer 顺序与 instructions 的顺序一致. 若 branch 预测错误, flush Reorder buffer 的相关内容.

5. Hardware Speculation

Table VII.1: Example

inst	operand	issue	exe	write	commit
div	x2, x3, x4	1	2	42	43
mul	x1, x5, x6	2	3	13	44
add	x3, x7, x8	3	4	5	45
mul	x1, x1, x3	14	15	25	46
sub	x4, x1, x5	15	26	27	47
add	x1, x4, x2	16	43	44	48

Example:

- free reservation station on result broadcast (not instruction dispatch);
- issue, capture, dispatch in same cycle;
- 1-cycle add, 10-cycle mul, 40-cycle div;

exe 是开始执行, write 是执行完 broadcast 的周期, 执行完是在 write-1 的周期. 对于此题, reservation station 只有在 broadcast 后 free, 所以 issue 要推迟.

VIII ILP – Exploitation

Ideal IPC & more Data path >1

1. Multiple Issue

2. VLIW

2.1 VLIW Scheduling

3. issue speculation

Further Example x2

4. Branch-Target Buffer

5. Return Address Stack

IX DLP Architecture

1. Data-Level Parallelism (DLP)

- Vector Architecture
- Multimedia SIMD
- GPU

2. Vector Architecture

1) Grab sets of data elements scattered about memory.
2) Place them in large sequential reg files.
3) Place them in large sequential reg files.
4) Disperse the results back into memory

A single instruction works on vector of data, which results in dozens of register-register operations on independent data elements.

2.1 RV64V

2.2 RV64V Vector Instructions

2.3 Dynamic Register Typing

Omitted in instruction encoding. Associated a data type and data size with each vector register.

Enable programs to disable unused vector registers.

2.4 DAXPY Example

Double precision a × X + Y

- X and Y have 32 elements
- x5 and x6 hold the starting addresses of X and Y, respectively

Code 1: RV64V code

```
    fld f0.a            #
    addi x28, x5, #256  # Last address to load
Loop:                   #
    fld f1, 0(x5)       #
    fmul.d f1, f1, f0   #
    fld f2, 0(x6)       #
    fadd.d f2, f2, f1   #
    fsd f2, 0(x6)       #
    addi x5, x5, #8     #
    addi x6, x6, #8     #
    bne x28, x5 Loop    #
```

Code 2: RV64V code: double-precision

```
vsetdcfg 4*FP64     # Enable 4 DP FP vregs
fld f0, a           #
vld v0, x5          #
vmul v1, v0, f0     #
vld v2, x6          #
vadd v3, v1, v2     #
vst v3, x6          #
vdisable            # Disable vector regs
```

no loop-carried dependences

Chaining:

- forwarding of element-dependent operations, in that dependent operations are chainedtogether
- scalar and vector registers generate interim results which can be used immediately, without additional memory references

Flexible Chaining: allow a vector instruction to chain to essentially any other active vector instruction

2.5 Vector Execution Time

For a sequence of vector operations

- Convoy

  the set of vector instructions that could potentially execute together

  co-convoy instructions must not contain any structural hazards

  RAW is allowed in the same convoy
- Chime:

  the unit of time taken to execute the convoy

Cross-convoy instructions are serialized

e.g.

3. optimize vector architecture

3.1 Multiple Lanes

use multiple functional units to improve vector performance; process several elements per clock cycle

vector register memory is divided across the lanes

3.2 Vector-Length Registers

Vector length is often unknown at compile time

Vector-length register (vl) controls the length of any vector operation, including a vector load or store

The value in vl cannot be greater than the maximum vector length (mvl)

e.g.

4. handle IF in loops

4.1 Predicate Registers

to handle IF in loops

Vector-mask control: use predicate registers to hold the mask and essentially provide conditional execution of each element operation

4.2 Memory Banks

4.3 Stride

4.4 Gather-Scatter

X DLP Exploitation

1. Loop-Level Parallelism

- in-iteration dependence
- cross-iteration/Loop-carried dependences
- No circular loop-carried dependence: self dependence, mutual dependence;

e.g.

2. find dependences

2.1 Index Affinity

2.2 Dependence Conditions

2.3 GCD Test

GCD (Greatest Common Divisor) Test: a, b, c and d are all constants, i.e. ai+b and ci+d. If a loop-carried dependence exists,

$$(d - b)\%gcd(c, a) = 0$$

3. Dependency Types

4. Dependency Elimination

XI TLP Coherence

1. Thread-Level Parallelism

TLP is identified by software or programmer and threads consist of (parallel) instructions

MIMD: multiple instruction streams and multiple data streams.

multiprocessors: computers consisting of tightly coupled processors

Exploiting TLP:

- Parallel processing
- Request-level parallelism

2. Multiprocessor Architecture

According to memory organization and interconnect strategy Two classes:

- symmetric/centralized shared-memory multiprocessors (SMP)
- distributed shared-memory multiprocessors (DMP)

2.1 Centralized Shared Memory

Share a single centralized memory; All processors have uniform latency from memory uniform memory access (UMA) multiprocessors

2.2 Distributed Shared Memory

Distributing memory among the nodes increases bandwidth & reduces local-memory latency.

NUMA: nonuniform memory access. access time depends on data word location in memory.

2.3 Hurdles of Parallel Processing

Limited parallelism in programs makes it difficult to achieve good speedups in any parallel processor (Amdahl's law)

Relatively high cost of communications involves the large latency of remote access in a parallel processor

Improve Parallel Processing:

- insufficient parallelism
- long-latency remote communication

3. Centralized Shared Memory

3.1 Cache Coherence Problem

- Global state defined by main memory
- Local state defined by the individual caches, private to each processor core

A memory system is Coherent if any read of a data item returns the most recently written value of that data item.

Two critical aspects:

- coherence: defines what values can be returned by a read
- consistency: determines when a written value will be returned by a read

3.2 Coherence Property

A memory is coherent if

1) A read by processor P to location X that follows a write by P to X (RAW), with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

2) A read by a processor to location X that follows a write by another processor to X (RAW) returns the written value if the read and the write are sufficiently separated in time and no other writes to X occur between the two accesses

3) Write serialization: two writes (WAW) to the same location by any two processors are seen in the same order by all processors

3.3 Consistency

Memory consistency protocol.

e.g, a write of X on one processor precedes a read of X on another processor by a very small time, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point

3.4 Cache Coherence Protocols

- Directory based: the sharing status of a particular block of physical memory is kept in one location, called directory
- Snooping: every cache that has a copy of the data from a block of physical memory could track the sharing status of the block

3.5 MSI protocol

Snooping Coherence Protocols:

- Write invalidate protocol: invalidate other copies on a write
- Write update/broadcast protocol

Implement Write Invalidate Protocol: MSI protocol: three block states

- Invalid
- Shared: indicates that the block in the private cache is potentially shared
- Modified: indicates that the block has been updated in the private cache; implies that the block is exclusive

MSI Extensions:
MESI

- exclusive: indicates when a cache block is resident only in a single cache but is clean (after initial read)

MOESI

- owned: indicates that the associated block is owned by that cache and out-of-date in memory

MESIF

- forward: designates which sharing processor (among ones with the same shared-state data block) should respond to a request

3.6 Coherence Miss

True sharing miss:

1) first write by a processor to a shared cache block causes an invalidation to establish ownership of that block;
2) another processor reads a modified word in that cache block;

False sharing miss: a single valid bit per cache block; occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into

4. Distributed Shared Memory

A directory is added to each node; Each directory tracks the caches that share the memory addresses of the portion of memory in the node;

need not broadcast on every cache miss as in snooping-based coherence protocol

4.1 Directory-based Cache Coherence Protocol

Common cache states

- Shared: one or more nodes have the block cached, and the value in memory is up to date (as well as in all the caches)
- Uncached: no node has a copy of the cache block
- Modified: exactly one node has a copy of the cache block, and it has written the block, so the memory copy is out of date