18-640 Foundations of Computer Architecture

Lecture 12: "Multithreading Processors"

John Paul Shen
October 14, 2014

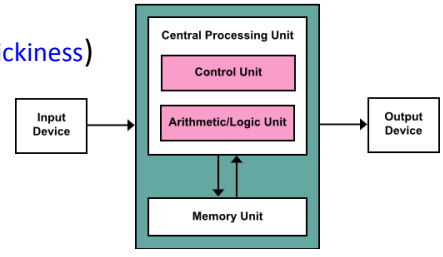
- Required Reading Assignment:
  - Chapter 11 of Shen and Lipasti.
- Recommended Reference:
  - "Multithreading Architecture" by Mario Nemirosky and Dean Tullsen, in Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2012.

Electrical & Computer ENGINEERING

18-640 Course Coverage: Processor Designs

Persistence of Von Neumann Model (Legacy SW Stickiness)

1. One CPU
2. Monolithic Memory
3. Sequential Execution Semantics



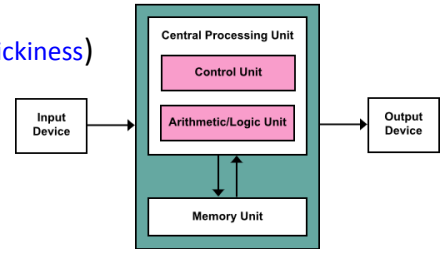
Evolution of Von Neumann Implementations:

- PP: Pipelined Processors (overlapped execution of in-order instructions)
- SSP: Superscalar Processors (out-of-order execution of multiple instructions
- MTP: Multithreaded Processors (concurrent execution of multiple threads)
- MCP: Multi-core Processors = CMP: Chip Multiprocessors (concurrent multi-threads)
- SMP: Symmetric Multiprocessors (concurrent multi-threads and multi-programs)

18-640 Course Coverage: Parallelism Exploited

Persistence of Von Neumann Model (Legacy SW Stickiness)

1. One CPU
2. Monolithic Memory
3. Sequential Execution Semantics



Parallelisms for **Performance** (→ for **Power** Reduction → for **Energy** Efficiency)

➢ **ILP**:	Basic Block	(exploit ILP in PP, SSP)
➢ **ILP**:	Loop Iteration	(exploit ILP in SSP, VLIW)
➢ **TLP**:	Task/Thread	(exploit TLP in MTP, MCP)
➢ **DLP**:	Data Set	(exploit DLP in VP/SIMD, GPU)
➢ **PLP**:	Process/Program	(exploit PLP in MCP, SMP)

18-640 Foundations of Computer Architecture

Lecture 12: "Multithreading Processors"

A. Thread-Level Parallelism
B. Multithreading Architecture
C. Coarse-Grain Multithreading
D. Fine-Grain Multithreading
E. Simultaneous Multithreading

Electrical & Computer ENGINEERING

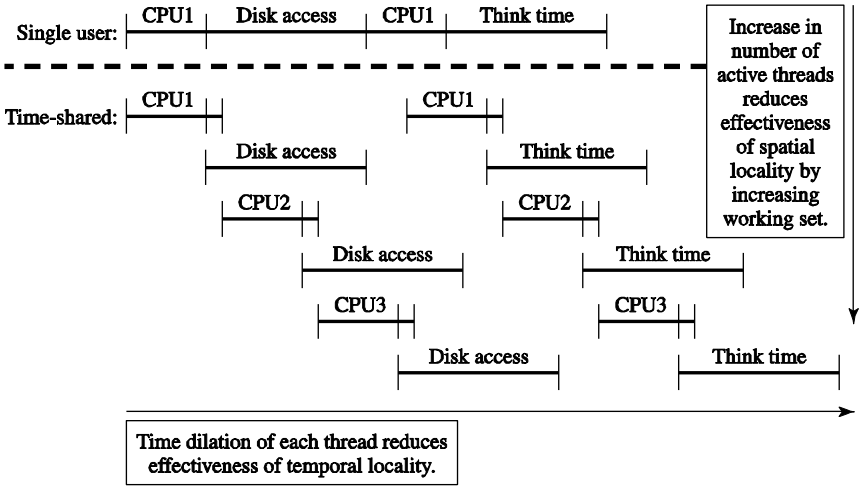
The Big Picture

- So far, we run single process, single thread
  - Extracting ILP from sequential instruction stream
- Single-thread performance can't scale indefinitely!
  - Limited ILP within each thread
  - Power consumption & complexity of superscalar cores
- We will now pursue Thread-Level Parallelism
  - To increase utilization and tolerate latency in single core
  - To exploit multiple cores (next lecture)

Thread-Level Parallelism

- Instruction-level parallelism
  - Reaps performance by finding independent work in a single thread
- Thread-level parallelism
  - Reaps performance by finding independent work across multiple threads
- Historically, requires explicitly parallel workloads
  - Originate from mainframe time-sharing workloads
  - Even then, CPU speed >> I/O speed
  - Had to overlap I/O latency with “something else” for the CPU to do
  - Hence, operating system would schedule other tasks/processes/threads that were “time-sharing” the CPU

Thread-Level Parallelism



- Reduces effectiveness of temporal and spatial locality

Thread-Level Parallelism

- Initially motivated by time-sharing of single CPU
  - OS, applications written to be multithreaded
- Quickly led to adoption of multiple CPUs in a single system
  - Enabled scalable product line from entry-level single-CPU systems to high-end multiple-CPU systems
  - Same applications, OS, run seamlessly
  - Adding CPUs increases throughput (performance)
- More recently:
  - Multiple threads per processor core
    - Coarse-grained multithreading (aka “switch-on-event”)
    - Fine-grained multithreading
    - Simultaneous multithreading
  - Multiple processor cores per die
    - Chip multiprocessors (CMP)
    - Chip multithreading (CMT)

Thread-Level Parallelism

- Parallelism limited by sharing
  - Amdahl's law:
    - Access to shared state must be serialized
    - Serial portion limits parallel speedup
  - Many important applications share (lots of) state
    - Relational databases (transaction processing): GBs of shared state
  - Even completely independent processes "share" virtualized hardware through O/S, hence must synchronize access
- Access to shared state/shared variables
  - Must occur in a predictable, repeatable manner
  - Otherwise, chaos results
- Architecture must provide primitives for serializing access to shared state

Reminder: Processes and Software Threads

- Process: an instance of a program executing in a system
  - OS supports concurrent execution of multiple processes
  - Each process has its own address space, set of registers, and PC
  - Two different processes can partially share their address spaces to communicate
- Thread: an independent control stream within a process
  - A process can have one or more threads
  - Private state: PC, registers (int, FP), stack, thread-local storage
  - Shared state: heap, address space (VM structures)
- A parallel program is one process but multiple threads

Reminder: Classic OS Context Switch

- OS context-switch
  - Timer interrupt stops a program mid-execution (precise)
  - OS saves the context of the stopped thread
    - PC, GPRs, and more
    - Shared state such as physical pages are not saved
  - OS restores the context of a previously stopped thread (all except PC)
  - OS uses a "return from exception" to jump to the restarting PC
    - The restored thread has no idea it was interrupted, removed, later restarted
  - Take a few hundred cycles per switch (why?)
    - Amortized over the execution "quantum"
- What latencies can you hide using OS context switching?
- How much faster would a user-level thread switch be?

Multithreaded Cores

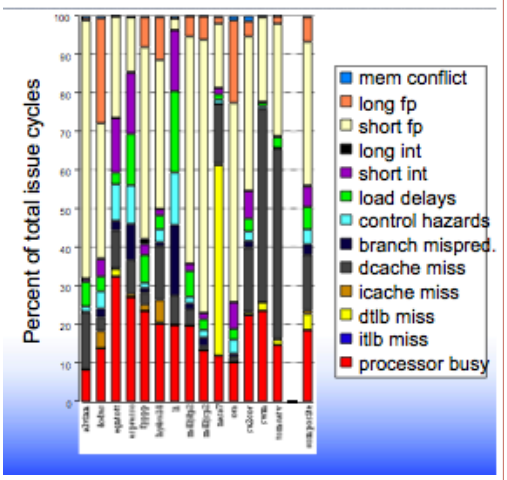
- Basic idea:
  - CPU resources are expensive and should not be idle
- 1960's: Virtual memory and multiprogramming
  - Virtual memory/multiprogramming invented to tolerate latency to secondary storage (disk/tape/etc.)
  - Processor-disk speed mismatch:
    - microseconds to tens of milliseconds (1:10000 or more)
  - OS context switch used to bring in other useful work while waiting for page fault or explicit read/write
  - Cost of context switch must be much less than I/O latency (easy)

Multithreaded Cores

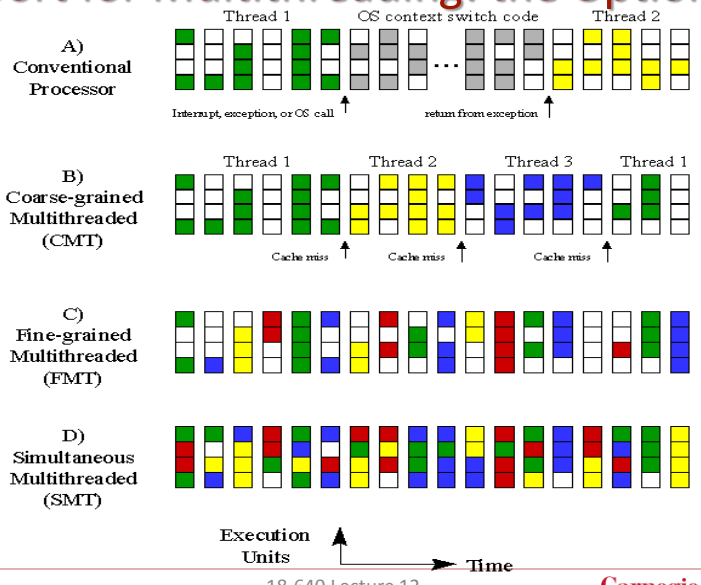
- 1990's: Memory wall and multithreading
  - Processor-DRAM speed mismatch:
    - nanosecond to fractions of a microsecond (1:500)
  - H/W task switch used to bring in other useful work while waiting for cache miss
  - Cost of context switch must be much less than cache miss latency
- Very attractive for applications with abundant thread-level parallelism
  - Commercial multi-user workloads

Multithreaded Processors

- Motivation: HW underutilized on stalls
  - Memory accesses (misses)
  - Data & control hazards
  - Synchronization & I/O
- Instead of reducing stalls, switch to running new thread for a while
  - Latency tolerance (vs avoidance)
  - Improves throughput & HW utilization
  - Does not improve single thread latency
- Need hardware support for fast context-switching

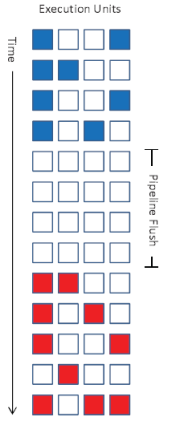


HW Support for Multithreading: the Options



Approaches to Multithreading

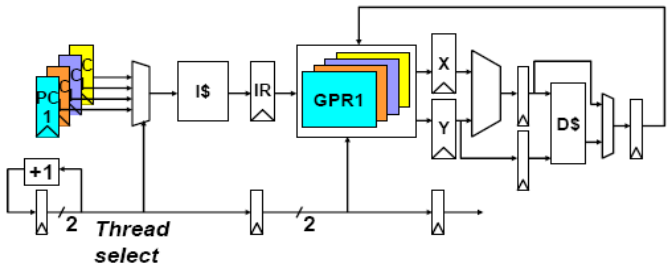
- Coarse-grain multithreading
  - Switch contexts on long-latency events (e.g. cache misses)
  - Need a handful of contexts (2-4) for most benefit
- Example: IBM RS64-IV (Northstar), 2 contexts
- Benefits:
  - Simple, improved throughput (~30%), low cost
  - Thread priorities mostly avoid single-thread slowdown
- Drawback:
  - Nondeterministic, conflicts in shared caches



Coarse Grain MT

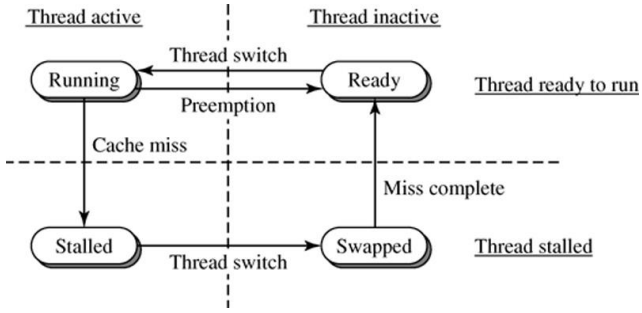
- Switch to another thread when processor runs into cache miss
  - In general, switch on high latency event
  - But not using SW because that would take longer than the cache miss
- HW support for context-switching
  - Replicate hardware context registers: PC, GPRs, cntrl/status, PT base ptr
    - Eliminates copying
  - Share some resources with tagging thread id
    - Cache, BTB and TLB match tags
- Hardware context switch takes only a few cycles
  - Flush the pipeline
    - Needed if you cannot tag instructions from different threads in the pipe
  - Choose the next ready thread as the currently active one
  - Start fetching instructions from this thread
- Where does the penalty come from? Can it be eliminated?

Simple Multithreaded Processor



- Multiple threads supported in hardware
  - With multiple hardware contexts
- Context switch events: with coarse-grain MT,
  - High latency event cache miss, I/O operations
  - Max cycle count for fairness

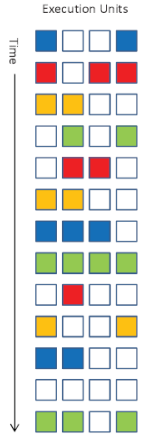
Coarse-Grain Multithreading: Thread States



- 4 potential states:
  - {Ready/not-ready} x {using HW context, swapped out}
- User-level runtime or OS can manage swapping
  - Based on readiness, fairness, or priorities
- HW switches between resident threads

Approaches to Multithreading

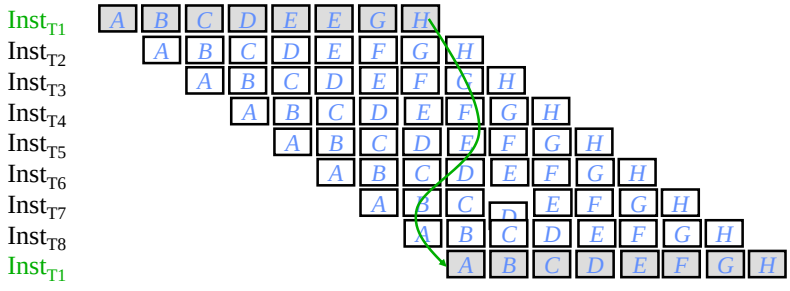
- Fine-grain multithreading
  - Switch contexts at fixed fine-grain interval (e.g. every cycle)
  - Need enough thread contexts to cover stalls
  - Example: Tera MTA, 128 contexts, no data caches
- Benefits:
  - Conceptually simple, high throughput, deterministic behavior
- Drawback:
  - Very poor single-thread performance



Fine Grain Multithreading

- Lead : when pipelined processor stalls due to RAW dependence, the execution stage is idling
- Why not switch to another thread on every cycle?
  - This will space apart dependent instructions
  - Switching must be instantaneous to have any advantage
- Solution: 1-cycle context switch
  - Multiple hardware contexts
  - Instruction buffer
    - To dispatch instruction from another thread at the next cycle
  - Multiple threads co-exist in the pipeline

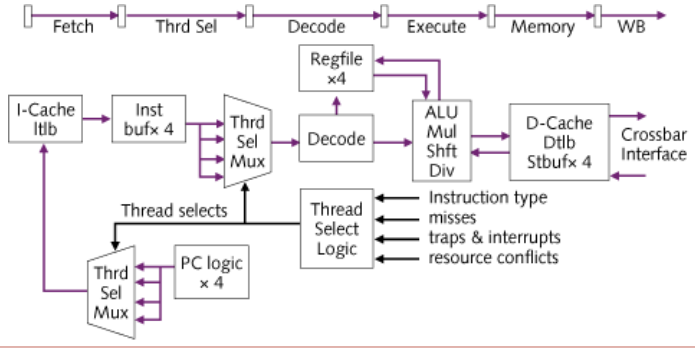
Example: Tera MTA

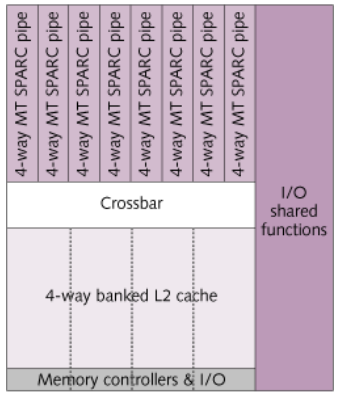


- Instruction latency hiding
  - Worst case instruction latency is 128 cycles (need 128 threads!!)
  - On every cycle, select a "ready" thread (i.e. last instruction has finished) from a pool of up to 128 threads
- Benefits : no forwarding logic, no interlock, and no cache
- Problem : single thread performance

Example: Sun UltraSparc T1

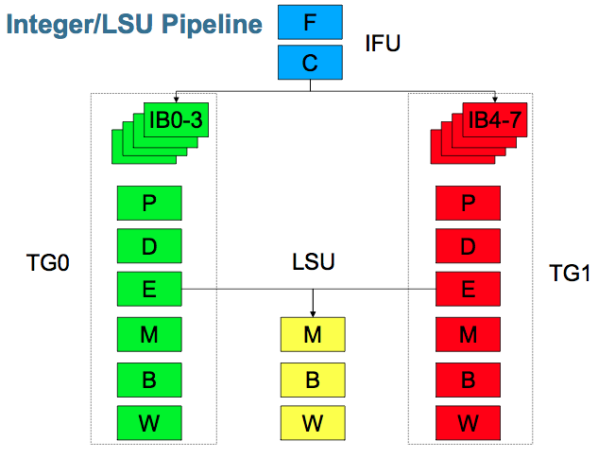
- A fine-grain multithreaded system
  - With multiple cores on a chip
- 4-threads per CPU, round-robin switch
  - Thread blocked on stalls, mul, div, loads, …





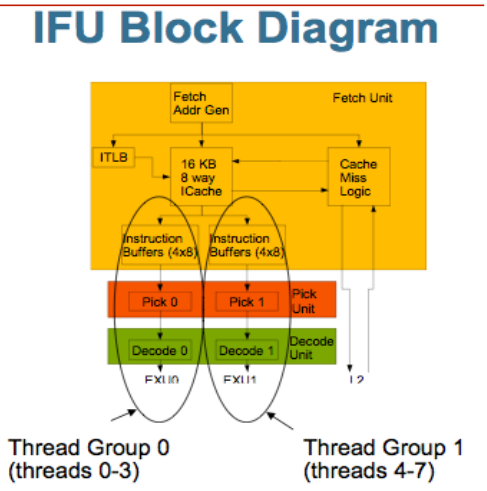
Example: Sun UltraSparc T2

- Similar to T1, but 8-way threaded
- 8-stage dual pipeline can execute 2 threads at the same time
  - Fetch, cache, pick, decode, execute, memory, bypass, writeback



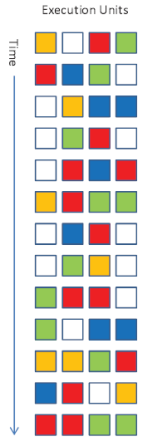
UltraSparc T2: Instruction Fetch Logic

- Fetch up to 4 instructions from cache
  - Priority to least-recently fetched ready thread
  - Predict not-taken (5 cycle penalty)
- Threads separated into two groups
  - 1 instruction issued per group
  - Priority to least-recently picked ready thread
  - Decode stage resolves LSU conflicts



Approaches to Multithreading

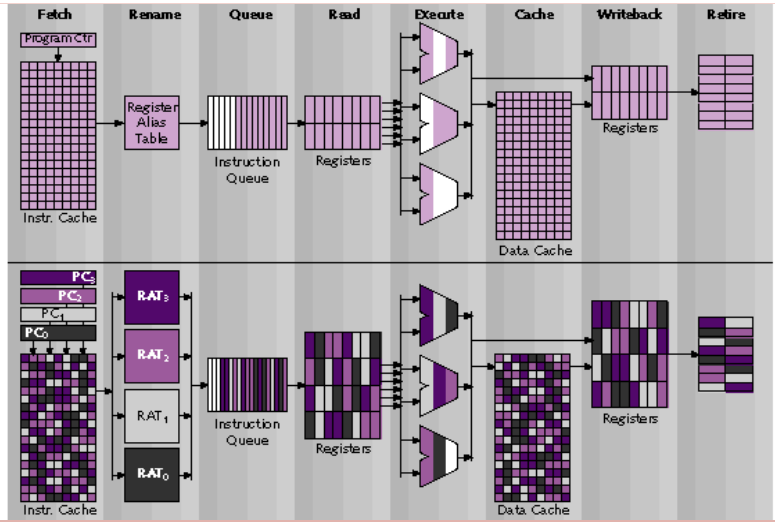
- Simultaneous Multithreading (SMT)
  - Multiple concurrent active threads (no notion of thread switching)
  - Need a handful of contexts for most benefit (2-8)
- Example:
  - Intel Pentium 4/Nehalem/Sandybridge,
  - IBM Power 5/6/7,
  - Alpha EV8/21464
- Benefits:
  - Natural fit for OOO superscalar
  - Improved throughput
  - Low incremental cost
- Drawbacks:
  - Additional complexity over OOO superscalar
  - Cache conflicts



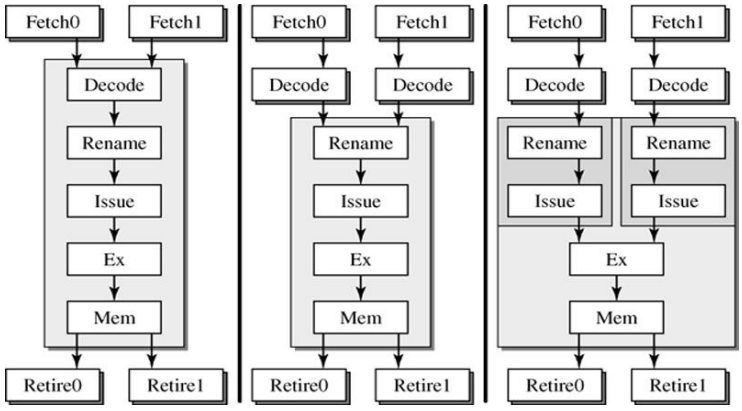
Simultaneous Multithreading or "HyperThreading"

- Lead: not all resources on all stages are busy in a superscalar core
- Why not execute multiple threads in the same pipeline stage
  - Problem : how can we support multiple active threads?
- Solution: variety of design options
  - What pipeline stages to share?
    - The rest of the stages are replicated
  - How to share resources?
    - Time sharing: one thread at a cycle
    - Spatial partition: dedicate a portion of resource to a thread

Regular OOO Vs. SMT OOO: the Alpha Approach



SMT Resource Sharing



- What are the tradeoffs here?
  - Complexity, resource utilization, interference

SMT Design Options (1)

- Fetch
  - How many threads to fetch from each cycle? If >1,
    - Multi-ported I-cache
    - I-fetch buffer replication
  - From which threads to fetch?
  - Branch predictor:
    - BTB is shared
    - BHR and RAS are replicated. Why?
- Decode
  - Share decoding logic
  - O(n^2) for dependence check
    - Spatial partition at the cost of single thread performance

SMT Design Options (2)

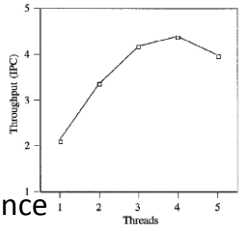
- Rename / RF
  - Need per-thread map tables
  - More architectural registers → Larger (A)RF
  - Sharing registers?
    - Multi-ported register file same as in non-SMT superscalar
- Issue
  - Share wakeup logic for arguments
  - O(n^2) for instruction selection
    - Spatial partition at the cost of single thread performance
- Execution
  - Very natural to share
  - Pipeline vs. dedicate multi-cycle ALUs

SMT Design Options (3)

- Memory
  - Multi-ported L1D?
  - Separate load/store queues?
    - Load forwarding/bypassing must be thread-aware
- Retire
  - Separate reorder buffers?
    - Per-thread instruction retirement

SMT Design Options: How Many Threads?

- As the number of HW threads increases:
  - Need larger register file
  - Resources statically partitioned/replicated → Lower utilization, lower single-thread performance
  - Shared resources → Better utilization, but interference and fairness issues
- More threads != higher aggregate throughput!
- Scheduling optimizations
  - Adapt number of running threads depending on interference
  - Co-schedule threads that have good interference behavior (symbiotic threads)



SMT Processors

- Alpha EV8: would be the 1st SMT CPU if not cancelled
  - 8-wide superscalar with support for 4-way SMT
    - SMT mode: like 4 CPUs with shared caches and TLBs
  - Replicated HW: PCs, registers (different maps, shared physical regs)
  - Shared: instruction queue, caches, TLBs, branch predictors, …
- Pentium 4 HT: 1st commercial SMT CPU (2 threads)
  - SMT threads share: caches, FUs, predictors (5% area increase)
  - Statically partitioned IW & ROB
  - Replicated RAS, 1st level global branch history table
    - Shared second-level branch history table, tagged with logical processor IDs
- IBM Power5, IBM Power 6: 2-way SMT
- Intel Nehalem (Core i7): 4-wide superscalar, 2-way SMT
- Intel Atom, AMD Bulldozer, …

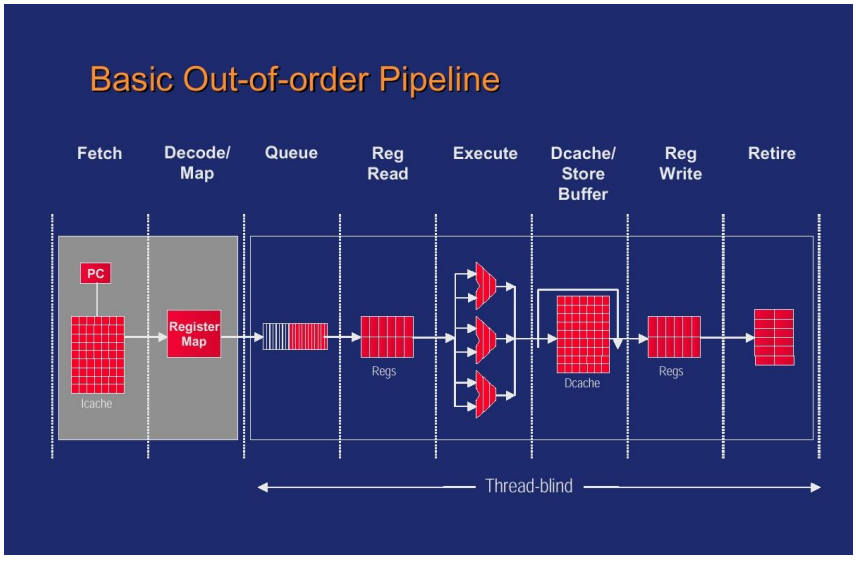
Fetch Policies for SMT

- Fetch polices try to identify the "best" thread to fetch from
- Policies
  - Least unresolved branches (BRCOUNT)
  - Least misses in-flight (MISSCOUNT)
  - Least instructions in decode/rename/issue stages (ICOUNT)
- Motivation for each policy?
- ICOUNT typically does best
  - Gives priority to more efficient threads
  - Avoids thread starvation

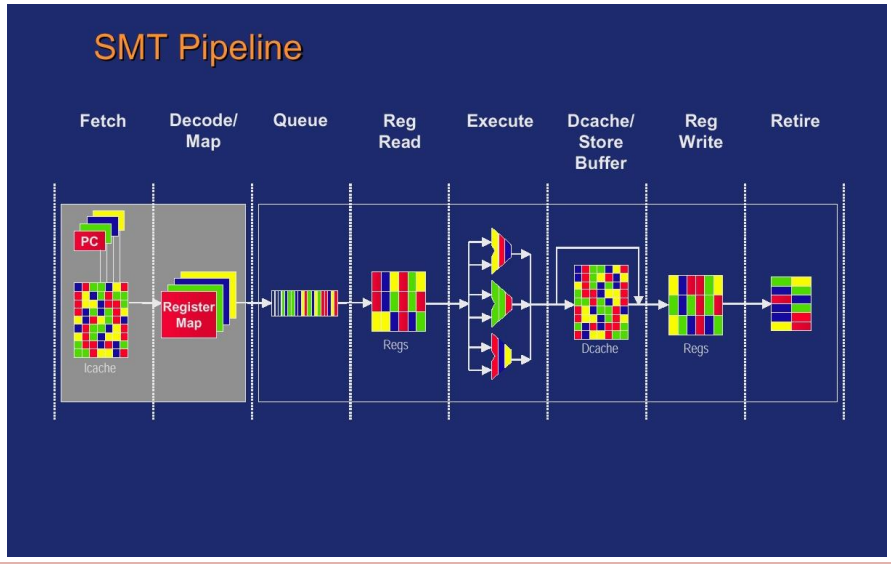
Long-Latency Stall Tolerance with SMT

- SMT does not necessarily work well for long-latency stalls
  - E.g. a thread misses to main memory (100-300 cycles)
  - CPU continues issuing instructions from stalled thread, eventually clogging IW
- Improved policies
  - Detect long-latency stall
  - Completely stop issuing instructions
  - Flush instructions from stalled thread (works quite well)

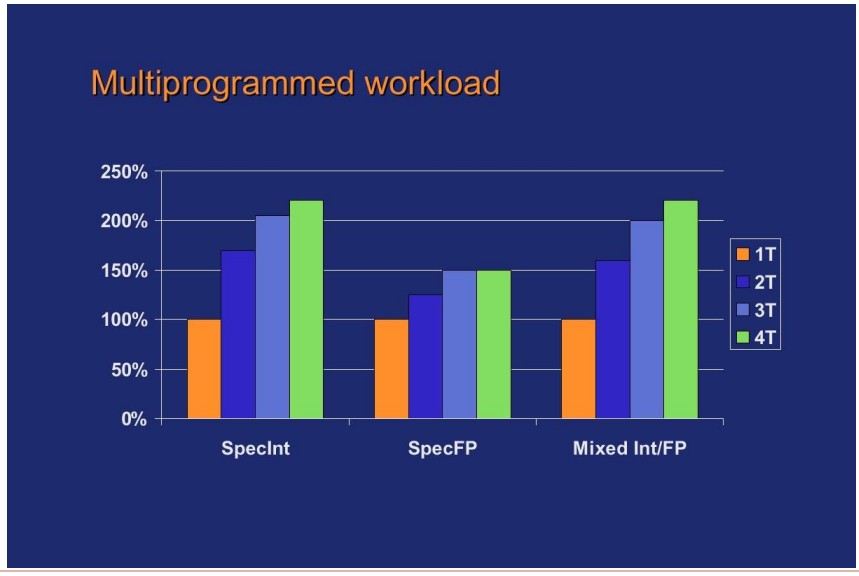
SMT Microarchitecture [Emer, PACT '01]



SMT Microarchitecture [Emer, PACT '01]



SMT Performance [Emer, PACT '01]



SMT Summary

- Goal: increase throughput
  - Not latency
- Utilize execution resources by sharing among multiple threads
- Usually some hybrid of fine-grained and SMT
  - Front-end is FG, core is SMT, back-end is FG
- Resource sharing
  - I$, D$, ALU, decode, rename, commit – shared
  - IQ, ROB, LQ, SQ – partitioned vs. shared

Summary

- Multithreaded processors implement hardware support to share pipeline across multiple threads
  - Stall tolerance
  - Better resource utilization
- 3 types of multithreading
  - Coarse grain (switch on long-latency stalls)
  - Fine-grain (switch every cycle)
  - Simultaneous (instrs from multiple threads each cycle)
- Design options in SMT processors

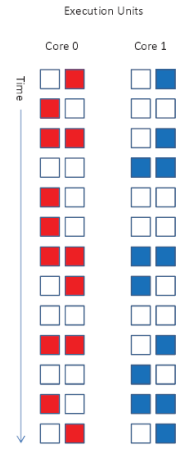
Approaches to Multithreading

Processor	Cores/chip	Multi-threaded?	Resources shared
IBM Power 4	2	No	L2/L3, system interface
IBM Power 7	8	Yes (4T)	Core, L2/L3, DRAM, system interface
Sun Ultrasparc	2	No	System interface
Sun Niagara	8	Yes (4T)	Everything
Intel Pentium D	2	Yes (2T)	Core, nothing else
Intel Core i7	4	Yes	L3, DRAM, system interface
AMD Opteron	2, 4, 6, 12	No	System interface (socket), L3

- Chip Multiprocessors (CMP)
- Becoming very popular

Approaches to Multithreading

- Chip Multithreading (CMT)
  - Similar to CMP
- Share something in the core:
  - Expensive resource, e.g. floating-point unit (FPU)
  - Also share L2, system interconnect (memory and I/O bus)
- Examples:
  - Sun Niagara, 8 cores per die, one FPU
  - AMD Bulldozer: one FP cluster for every two INT clusters
- Benefits:
  - Same as CMP
  - Further: amortize cost of expensive resource over multiple cores
- Drawbacks:
  - Shared resource may become bottleneck
  - 2nd generation (Niagara 2) does **not** share FPU



Multithreaded/Multicore Processors

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 con trol 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
CMT	Various core components (e.g. FPU), secondary cache, system interconnect	All contexts concurrently active; no switching
CMP	Secondary cache, system interconnect	All contexts concurrently active; no switching

- Many approaches for executing multiple threads on a single die
  - Mix-and-match: IBM Power7 CMP+SMT