# 18-747 Lecture 21: Multithreading Processors

James C. Hoe
Dept of ECE, CMU
November 14, 2001

*Reading Assignments:* *paper below*

*Announcements:* Project 3 Short Proposal due Monday November 19

*Handouts:* *"Simultaneous Multithreading: A Platform for Next-Generation Processors",* Eggers, et al., IEEE Micro

# Remaining Lectures

- 11/19 L22: Binary Translation and Optimization
- 11/26 L23: SMP Cache Coherence

---

- 11/28 L24: Exam Review and Course at a Glance
  Guest Lecture: Low Power Processor Design
  *by Prof. D. Marculescu*
- 12/3 Exam 2
- 12/5 Recitation by Aaron
- 12/10 Class Presentations

# Instruction-Level Parallelism

- When executing a program, how many "independent" operations can be performed in parallel
- How to take advantage of ILP
  - Pipelining (including superpipelining)
    - overlap different stages from different instructions
    - limited by divisibility of an instruction and ILP
  - Superscalar (including VLIW)
    - overlap processing of different instructions in all stages
    - limited by ILP
- How to increase ILP
  - dynamic/static register renaming $\Rightarrow$ reduce WAW and WAR
  - dynamic/static instruction scheduling $\Rightarrow$ reduce RAW hazards
  - use predictions to optimistically break dependence

# Thread-Level Parallelism

- The average processor actually executes several "programs" (a.k.a. processes, threads of control, etc) at the same time *Time Multiplexing*
- The instructions from these different threads have lots of parallelism
- Taking advantage of "thread-level" parallelism, i.e. by concurrent execution, can improve the overall throughput of the processor (but not turn-around time of any one thread)

*Basic Assumption: the processor has idle resources when running only one thread at a time*

# Multiprocessing

- Time-multiplex multiprocessing on uniprocessors started back in 1962
- Even concurrent execution by time-multiplexing improves throughput *How?*
  - a single thread would effectively idle the processor when spin-waiting for I/O to complete, e.g. disk, keyboard, mouse, etc.
  - can spin for thousands to millions of cycles at a time


  - a thread should just go to "sleep" when waiting on I/O and let other threads use the processor, *a.k.a. context switch*

# Context Switch

- A "context" is all of the processor (plus machine) states associated with a particular process
  - *programmer visible states:* program counter, register file contents, memory contents
  - *and some invisible states:* control and status reg, page table base pointers, page tables

  *What about cache (virtual vs. physical), BTB and TLB entries?*
- Classic Context Switching
  - timer interrupt stops a program mid-execution (precise)
  - OS saves away the context of the stopped thread
  - 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 restored and restarted*

# Saving and Restoring Context

- Saving
  - "Context" information that occupy unique resources must be copied and saved to a special memory region belonging exclusively to the OS
    - e.g. program counter, reg file contents, cntrl/status reg
  - "Context" information the occupy commodity resources just needs to be hidden from the other threads
    - e.g. active memory pages can be left in physical memory but page translations must be removed (but remembered)
- Restoring is the opposite of saving
- The act of saving and restoring is performed by the OS in software

  *$\Rightarrow$ can take a few hundred cycles per switch, but the cost is amortize over the execution "quantum"*

*(If you want the full story, take a real OS course!)*

# Fast Context Switches

- A processor becomes idle when a thread runs into a cache miss

  *Why not switch to another thread?*

- Cache miss lasts only tens of cycles, but it costs OS at least 64 cycles just to save and restore the 32 GPRs
- Solution: fast context switch in hardware
  - replicate hardware context registers: PC, GPRs, cntrl/status, PT base ptr — *eliminates copying*
  - allow multiple context to share some resources, i.e. include process ID as cache, BTB and TLB match tags

    *eliminates cold starts*
  - hardware context switch takes only a few cycles
    - set the PID register to the next process ID
    - select the corresponding set of hardware context registers to be active

# Example: MIT's Sparcle Processor

- Based SUN SPARC II processors
  - provided hardware contexts for 4 threads, one is reserved for the interrupt handlers
  - hijacked SPARC II's register windowing mechanism to support fast switching between 4 sets of 32 GPRs
  - switches context in 4 cycles
- Used in a cache-coherent distributed shared memory machine
  - On a cache miss to remote memory (takes hundreds of cycles to satisfy), the processor automatically switches to a different user thread
  - The network interface can interrupt the processor to wake up the message handler thread to handle communication

# Really Fast Context Switches

- When pipelined processor stalls due to RAW dependence between instructions, the execution stage is idling

*Why not switch to another thread?*

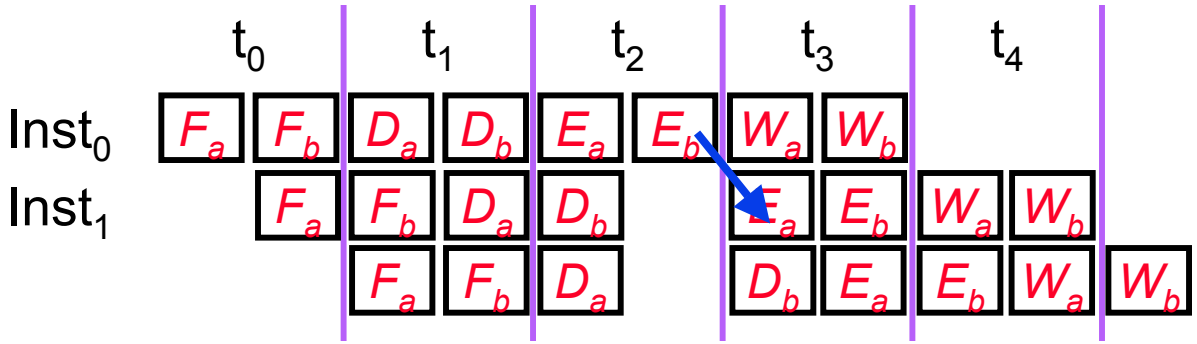
- Not only do you need hardware contexts, switching between contexts must be instantaneous to have any advantage!!
- If this can be done,
  - don't need complicated forwarding logic to avoid stalls
  - RAW dependence and long latency operations (multiply, cache misses) do not cause throughput performance loss

*Multithreading is a "latency hiding" technique*

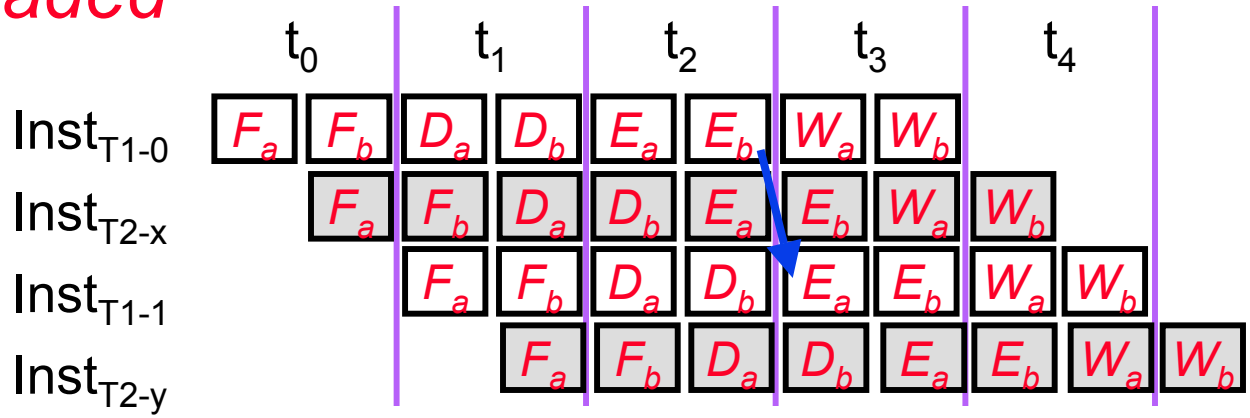
# Fine-grain Multithreading

- Suppose instruction processing can be divided into several stages, but some stages has very long latency
  - run the pipeline at the speed of the slowest stage, or
  - superpipeline the longer stages, but then back-to-back dependencies cannot be forwarded

*superpipelined*

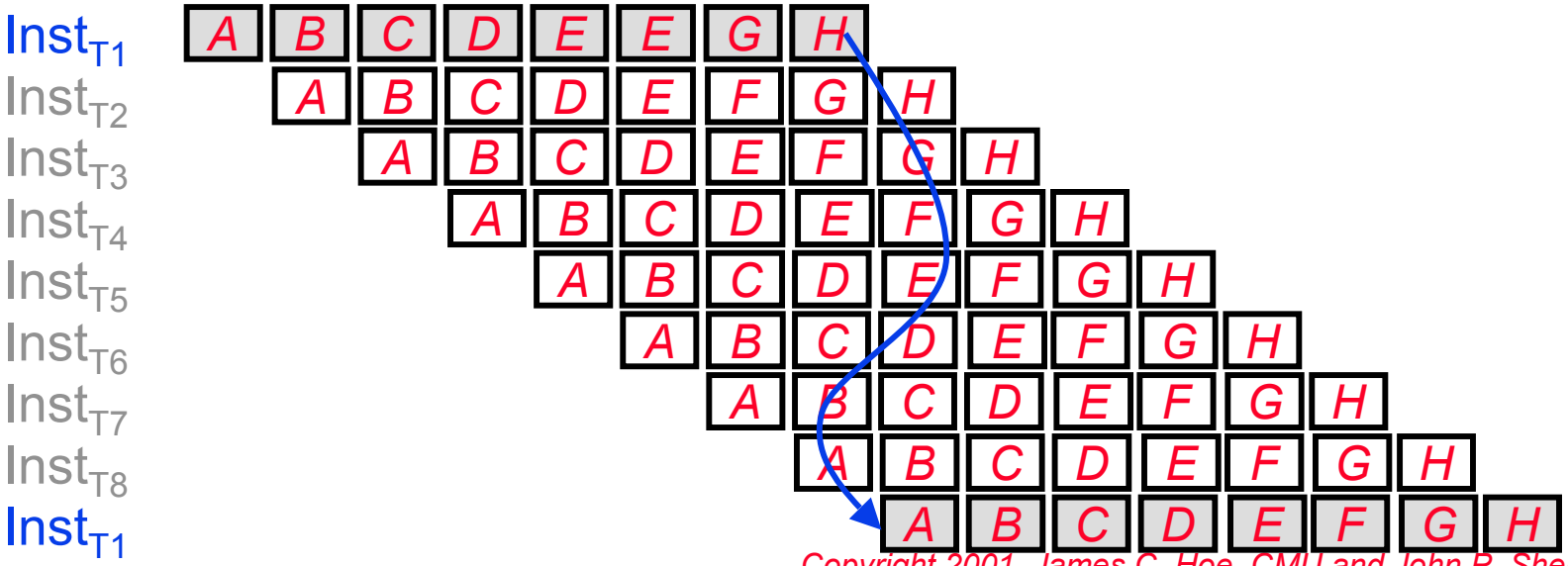



*2-way multithreaded superpipelined*

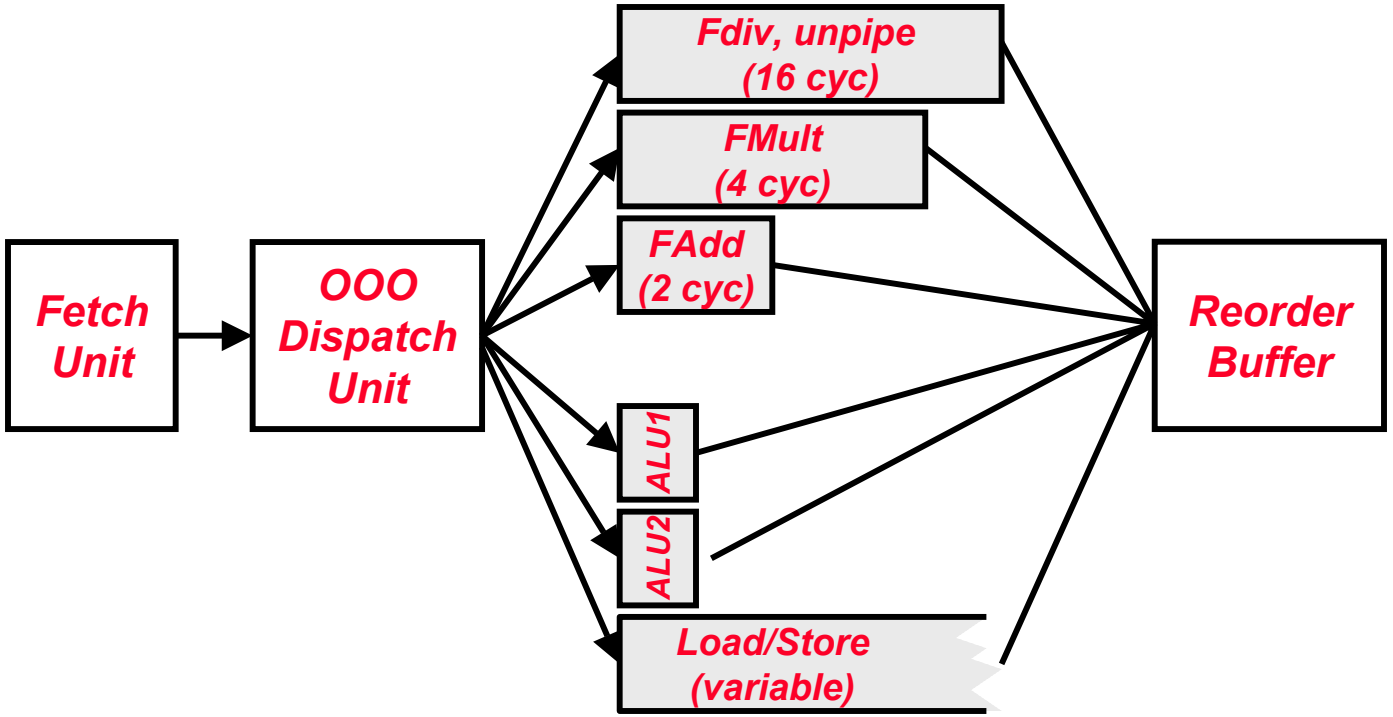


# Examples: Instruction Latency Hiding

- Using the previous scheme, MIT Monsoon pipeline cycles through 8 statically scheduled threads to hide its 8-cycle (pipelined) memory access latency
- HEP and Tera MTA *[B. Smith]*:
  - on every cycle, dynamically selects a "ready" thread (i.e. last instruction has finished) from a pool of upto 128 threads
  - worst case instruction latency is 128 cycles *(may need 128 threads!!)*
  - a thread can be waken early (i.e. before the last instruction finishes) using software hints to indicate no data dependence



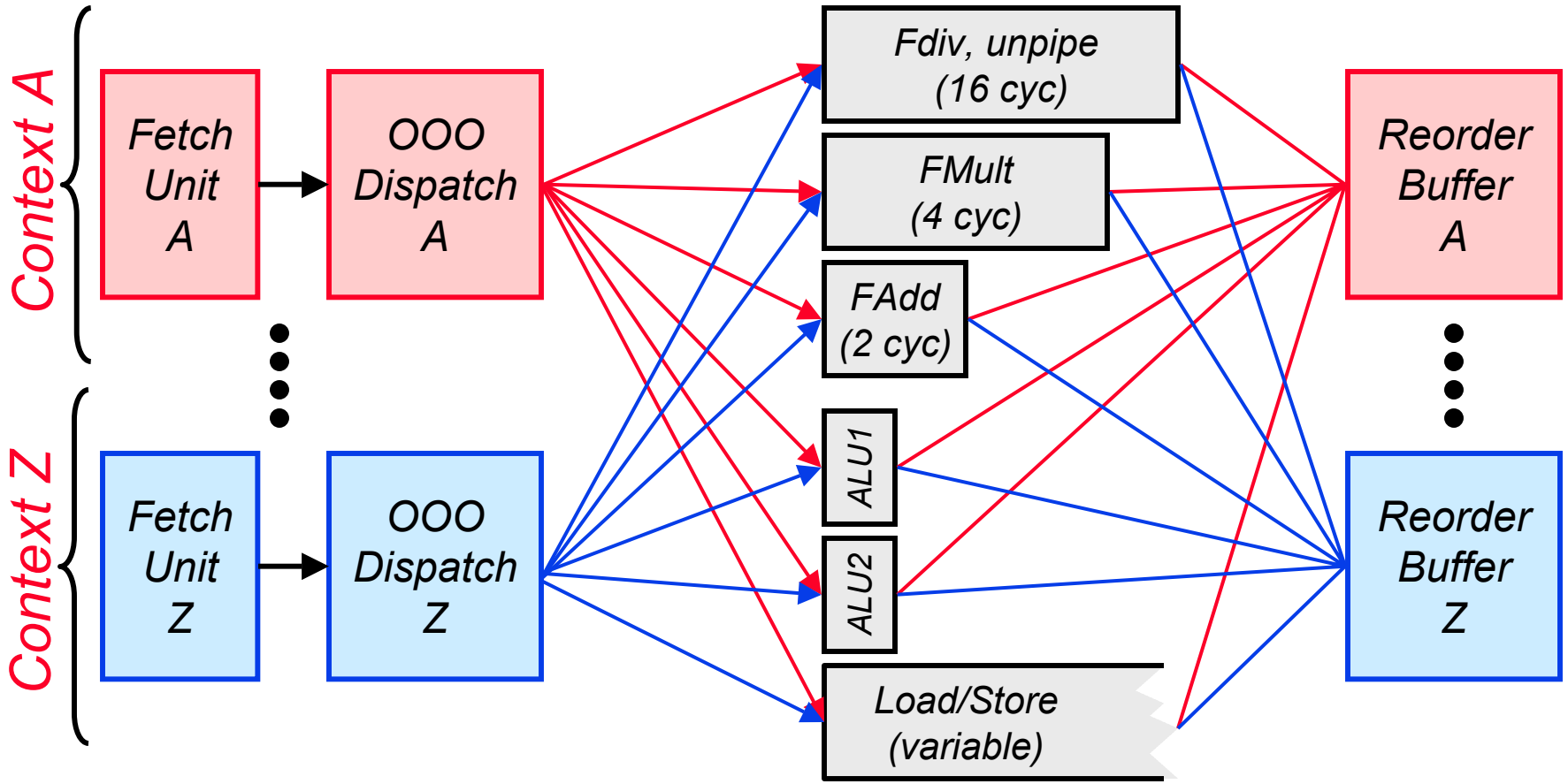
# Really Really Fast Context Switches




- Superscalar processor datapath must be over-resourced
  - has more functional units than ILP because the units are not universal
  - current 4 to 8 way designs only achieves IPC of 2 to 3
- Some units must be idling in each cycle

*Why not switch to another thread?*

# Simultaneous Multi-Threading *[Eggers, et al.]*




- Dynamic and flexible sharing of functional units between multiple threads

$\Rightarrow$ *increases utilization* $\Rightarrow$ *increases throughput*

# Compaq Alpha EV8

- Technology
  - 1.2 ~ 2.0 GHz
  - 250 million transistors (mostly in the caches)
  - 0.125um CMOS with copper
  - 1.2V Vdd
  - 1100 signal pins (flip chip)

    *probably about that many power and ground pins*
- Architecture
  - 8-wide superscalar with support for 4-way SMT

    *supports both ILP and thread-level parallelism*
  - On-chip router and directory support for building glueless 512-way ccNUMA SMP

*[Joel Emer's Microprocessor Forum]*

# EV8: Superscalar to SMT

- In SMT mode, it is as if there are 4 processors on a chip that shares their caches and TLB
- Replicated hardware contexts
  - program counter
  - architected registers *(actually just the renaming table since architected registers and rename registers come from the same physical pool)*
- Shared resources
  - rename register pool (larger than needed by 1 thread)
  - instruction queue
  - caches
  - TLB
  - branch predictors
- The dynamic superscalar execution pipeline is more or less unchanged

# SMT Issues

- Adding a SMT to superscalar
  - Single-thread performance is slight worse due to overhead (longer pipeline, longer combinational delays)
  - Over-utilization of shared resources
    - contention for instruction and data memory bandwidth
    - interferences in caches, TLB and BTBs

    *But remember multithreading can hide some of the penalties. For a given design point, SMT should be more efficient than superscalar if thread-level parallelism is available*
- High-degree SMT faces similar scalability problems as superscalars
  - needs numerous I-cache and d-cache ports
  - needs numerous register file read and write ports
  - the dynamic renaming and reordering logic is not simpler

# Speculative Multithreading

- SMT can justify wider-than-ILP datapath
- But, datapath is only fully utilized by multiple threads
- How to make single-thread program run faster?

  *Think about predication*

- What to do with spare resources?
  - execute both sides of hard-to-predictable branches
  - send another thread to scout ahead to warm up caches & BTB
  - speculatively execute future work

    *e.g. start several loop iterations concurrently as different threads, if data dependence is detected, redo the work*

    *Must have ways to contain the effects of incorrect speculations!!*
  - run a dynamic compiler/optimizer on the side

# Slipstream Processors

- Execute a single-threaded application redundantly on a "modified" 2-way SMT, with one thread slightly ahead
  - an advanced stream (A-stream) followed by a redundant stream (R-stream)

  *"The two redundant programs combined run faster than either can alone" [Rotenberg]*
- How is this possible?
  - A-stream is highly speculative
    - can use all kinds of branch and value predictions
    - doesn't go back to check or correct misprediction
    - even selectively skip some instructions

      *e.g. some instructions compute branch decisions, why execute them if I am going to predict the branch anyways*
  - A-stream should run faster, but its results can't be trusted
  - R-stream is executed normally, but it still runs faster because caches and TLB would have been warmed by the A-stream!!