
EECS 452 – Lecture 9

TLP – Thread-Level Parallelism

Instructor: Gokhan Memik

EECS Dept., Northwestern University

The lecture is adapted from slides by Iris Bahar (Brown), James Hoe (CMU), and John Shen (CMU -> Intel -> Nokia)

Overview of ILP



- When executing a program, how many *independent* operations can be performed in parallel
- How have we taken advantage of ILP so far?
 - Pipelining (including superpipelining)
 - overlap different stages from different instructions
 - limited by divisibility of an instruction and ILP
 - Superscalar
 - overlap processing of different instructions in all stages
 - limited by ILP
 - VLIW
 - limited by ILP
- What methods have we employed to increase ILP?
 - dynamic/static register renaming -> reduce WAW and WAR
 - dynamic/static instruction scheduling -> reduce RAW hazards
- Use predictions/speculation to optimistically break dependence
 - Covered: Branch prediction, memory dependence prediction
 - Not covered: value prediction

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 a lot of parallelism *Thread-Level Parallelism*
- Taking advantage of “thread-level” parallelism, (by concurrent execution), can improve the overall throughput of the processor
 - But NOT turn-around time of any single thread

Context-switch



- 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 spinwaiting 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 Switching (cont.)



- 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 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 occupies unique resources must be copied and saved to a special memory region belonging exclusively to the OS
 - e.g. program counter, reg file contents, control/status reg
- “Context” information that occupies 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
- *can take a few hundred cycles per switch, but the cost is amortize over the execution “quantum”*

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, control/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

Really Fast Context Switches

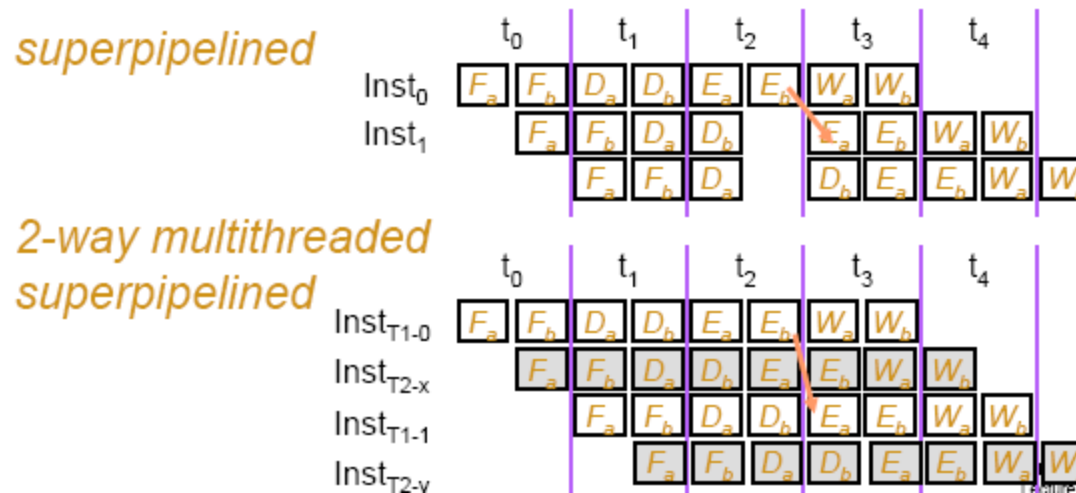


- When pipelined processor stalls due to RAW dependence between instructions, the execution stage is idling
- *Why not switch to another thread?*
- Not only do you need hardware contexts, switching between contexts must be instantaneous to have any advantage!!
- If this can be done,
 - don't need complicated 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*

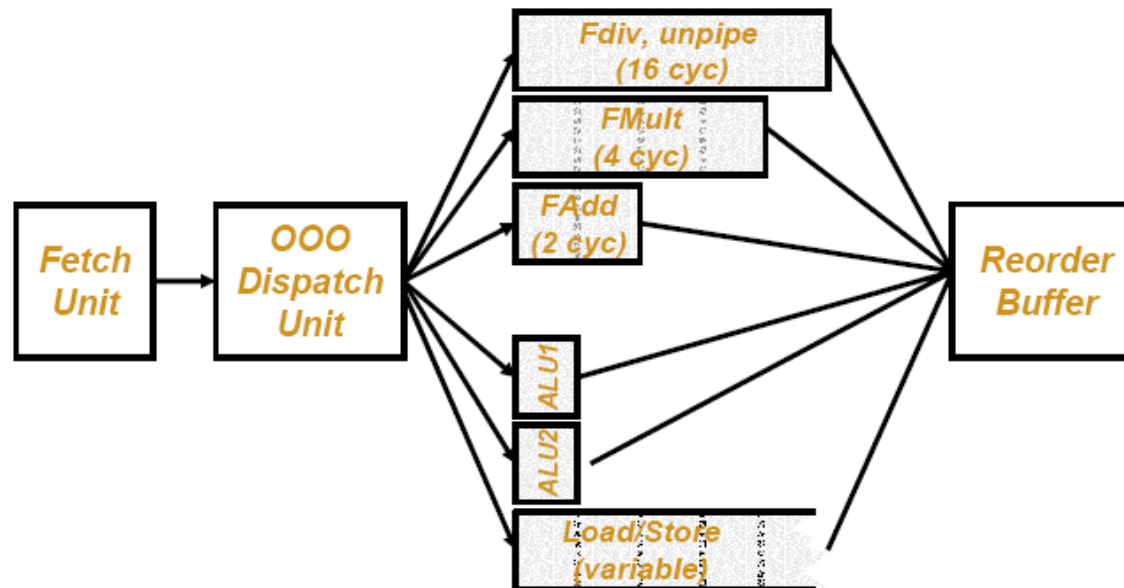
Fine-Grain Multithreading



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

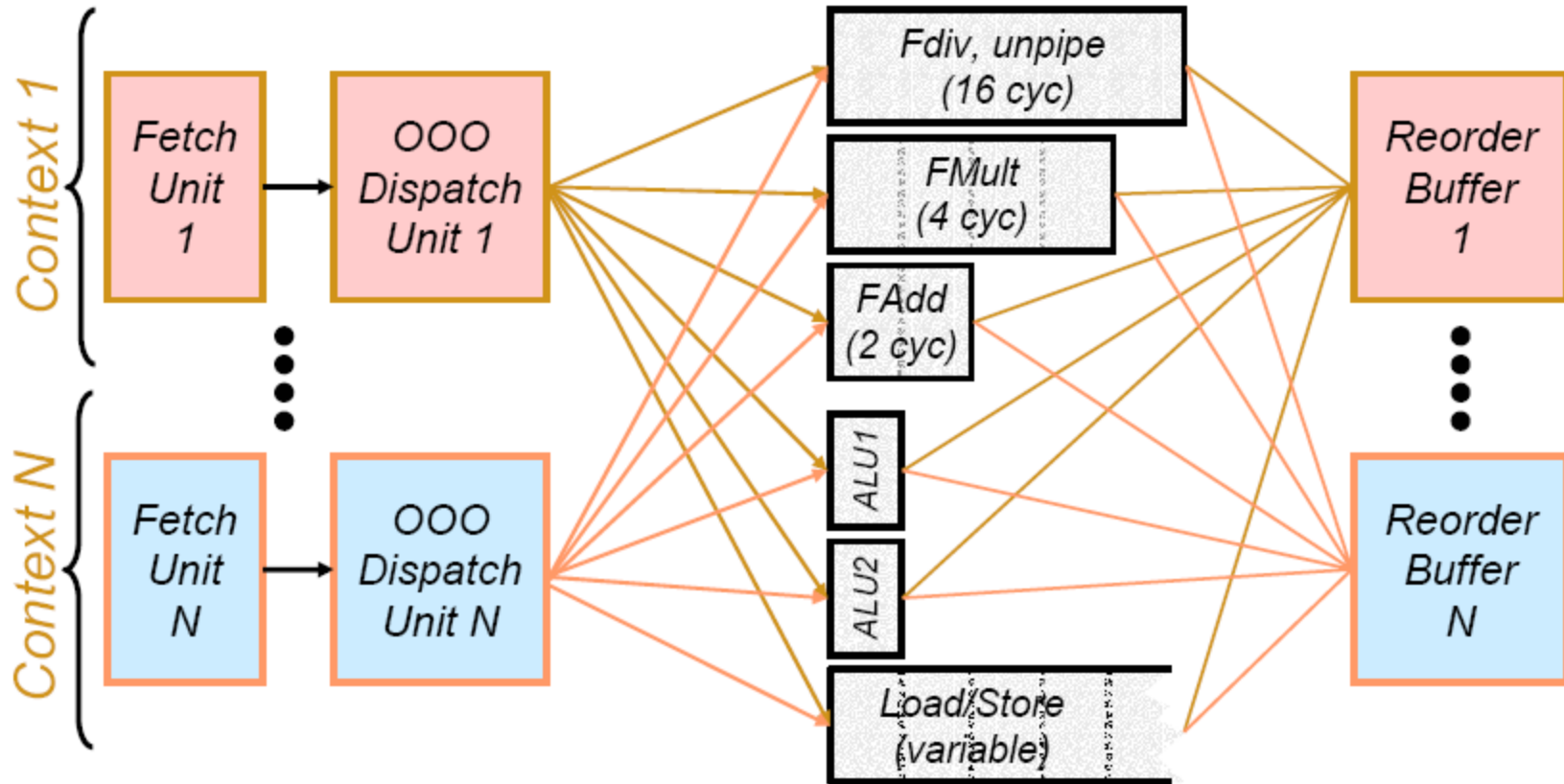


Really really fast context switch



- 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 achieve IPC of 2 to 3
- Some units must be idling in each cycle
 - Why not “switch” to another thread?

Simultaneous Multithreading



- Dynamic and flexible sharing of functional units between multiple threads
- *increases utilization* *increases throughput*

Digital/Compaq/HP 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

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 (i.e., unified reservation stations)
 - Caches, TLB, branch predictors
- The dynamic superscalar execution pipeline is more or less unchanged

SMT Issues and Summary



- Adding 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

MT Architectures

