Lecture 15:

Embedded Processors
Multithreaded Processors

Department of Electrical Engineering
Stanford University

`http://eeclass.stanford.edu/ee382a`

Announcements

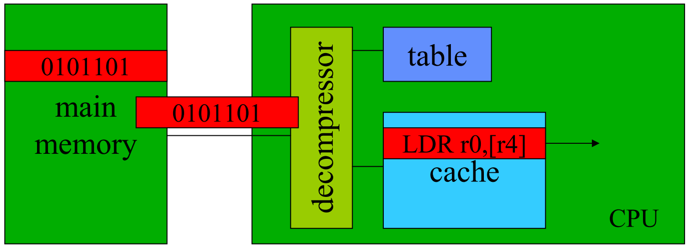
- TBD

Architectural Implications for Embedded Processors

- Simpler, predictable processor cores
  - In-order pipelines, VLIW, vectors, …
  - Limited use of prediction, out-of-order, …
- Simpler memory hierarchies
  - Software managed scratchpads instead of caches
    - Program explicitly manages subset of data close to processor
  - DMA transfers
  - Shallow cache hierarchies
- Support for fast interrupt processing
  - See EE282: multiple interrupt vectors, multithreading, …
- Note: most of these are good for power consumption as well…

Code Size Optimizations

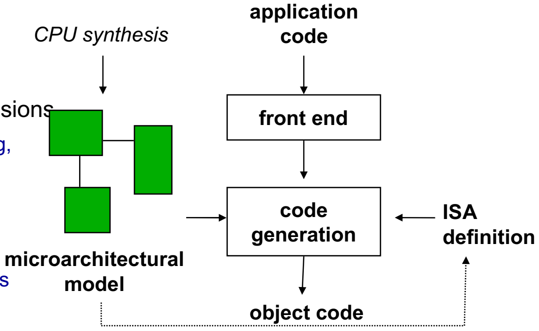
- Use multiple width instructions sets (e.g. MIPS-16, Thumb-2)
  - 32-bit for performance critical, 16-bit for the rest
- On the fly memory compression
  - Using dictionary based compression or other techniques



Performance Optimizations: ISA Extensions

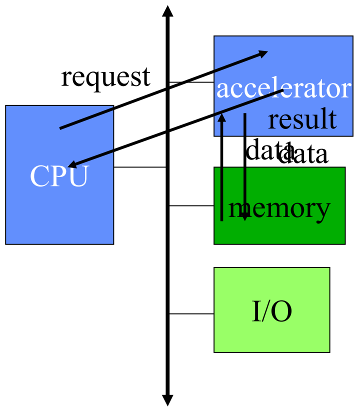
- Add app-specific instructions to CPU
  - E.g. special bit permutation
  - E.g. special MAC instruction
- Benefits
  - Better cost/performance
  - Better power/performance
- Several domain-specific extensions
  - Image processing, networking,
  - Video processing, Java, …
- Can be automated
  - Given a CPU/ISA template
  - Analyze apps for common ops
  - Define as new instructions
  - Implement in CPU/compiler

```
for (i=0; i<N; i++)
   c[i] = foo(a[i],b[i]);
```



Performance Optimizations: Custom Coprocessors and Accelerators

- Coprocessors: extend ISA for specific domain
  - E.g. FPU, vector/SIMD
  - Tightly integrated, fed instructions on each cycle
- Accelerators: custom or domain-specific functionality
  - E.g. encryption engine, TPC offloading engine, GPU, ME engine, …
  - Typically as a device on a bus, accessed through registers
    - But may be able to DMA in/out of its registers
  - Accelerators often implement a standard protocol
- Why accelerators
  - Better cost/performance
  - Better real-time performance (particularly for I/O)
  - May consume less energy
  - May not be able to do all the work on even the largest single CPU



Power Optimizations (see EE282)

- Simpler architectures
  - Processor and memory hierarchy
- Aggressive clock gating and use of low-power/idle modes
- Bus encoding or compression
- Reducing power supply (big wins as $P=CV^2f$)
  - Dynamic adjustment of Vdd and clock frequency
  - Save energy by reducing performance to minimum required
- Parallelism and pipelining
  - Allow for same performance with lower clock frequency
  - Enable aggressive clock frequency and power supply gating

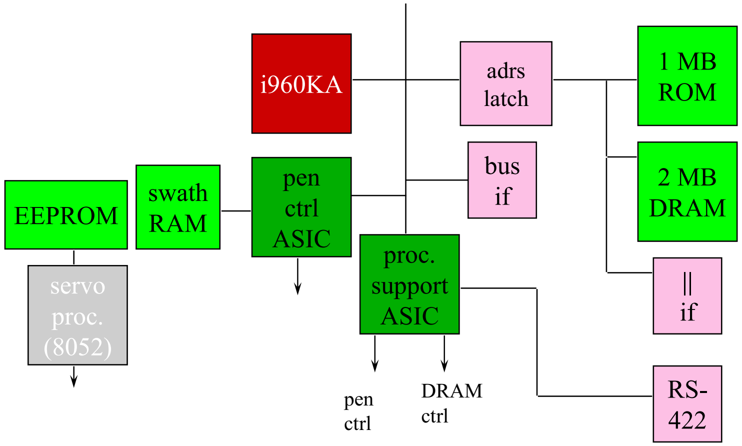
Example: Vector Power Consumption

- Can trade-off parallelism for power
  - Power = C *Vdd2 *F
  - If we double the lanes, peak performance doubles
  - Halving F restores peak performance but also allows halving the Vdd
  - Powernew = (2C)*(Vdd/2)2*(f/2) = Power/4
- Simpler logic for large number of operations/cycle
  - Replicated control for all lanes
  - No multiple issue or dynamic execution logic
- Simpler to gate clocks
  - Each vector instruction explicitly describes all the resources it needs for a number of cycles
  - Conditional execution leads to further savings

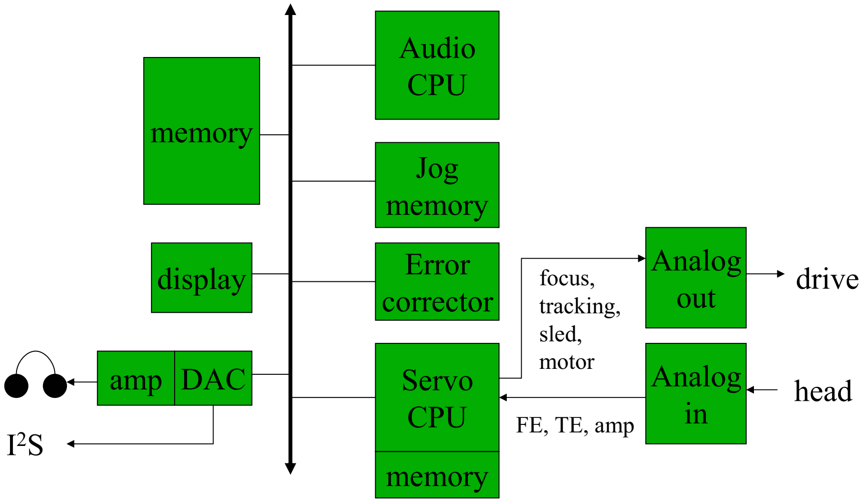
Embedded Software Development

- Still a lot of it done in assembly
  - Reasons: real-time constraints, limited performance, cost sensitive
    - If 1 month of assembly coding saves $1 per device, it may be worth it
- Large market of tools for embedded software development
  - Libraries, code development frameworks, analysis frameworks,…
  - System partitioning tools, …
  - Moving towards high-level frameworks (e.g. matlab or simulink)
- Operating systems
  - Simpler and typically customizable
  - Fast interrupt processing, real-time/priority scheduling, power management
  - Main players: VxWorks, GreenHill, QNX, RT-Linux, Windows-CE, PalmOS,

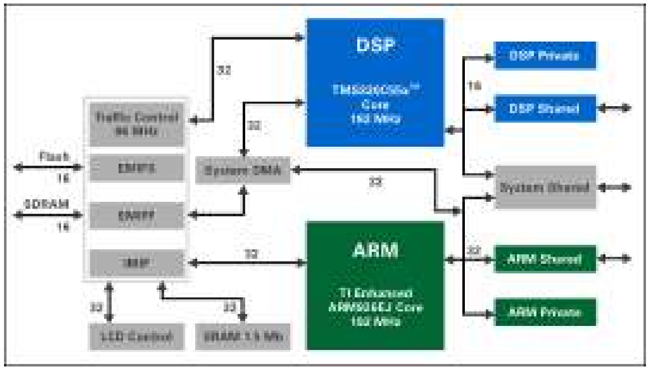
Example: HP DesignJet architecture



Example: CD/MP3 player

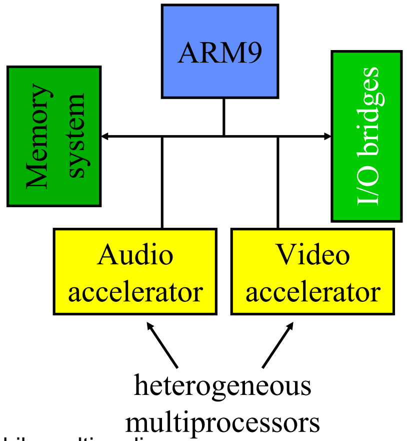


Example: TI OMAP



- Targets communications, multimedia.
- Multiprocessor with DSP, RISC.

Example: ST Nomadik



ARM9

Memory system

I/O bridges

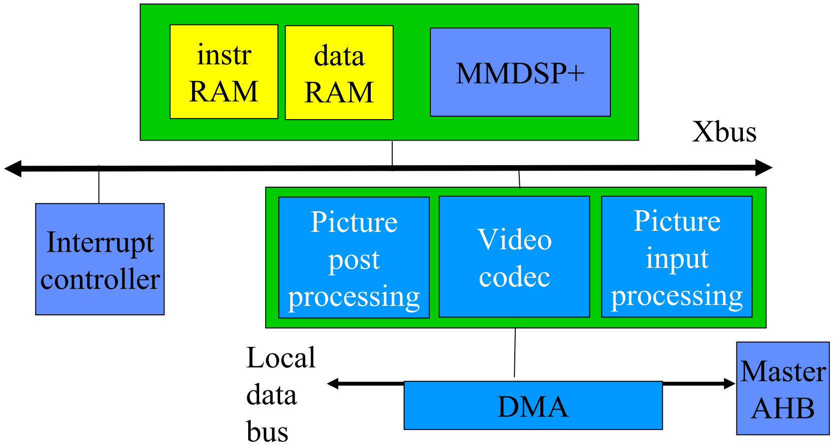
Audio accelerator

Video accelerator

heterogeneous multiprocessors

- Targets mobile multimedia
- A multiprocessor-of-multiprocessors.

Example Nomadik video accelerator



instr RAM

data RAM

MMDSP+

Xbus

Interrupt controller

Picture post processing

Video codec

Picture input processing

Local data bus

DMA

Master AHB

Multithreaded Processors

Department of Electrical Engineering
Stanford University

`http://eeclass.stanford.edu/ee382a`

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 ⇒ reduce WAW and WAR
  - Dynamic/static instruction scheduling ⇒ reduce RAW hazards
  - Use predictions to optimistically break dependence

Thread-Level Parallelism

- Most computers actually executes several "programs" at the same time
  - A.k.a. processes, threads of control, etc

Time Multiplexing

- The instructions from these different threads have lots of parallelism
  - No dependences across programs
- Taking advantage of "thread-level" parallelism, i.e. by concurrent execution, can improve the overall throughput of the processor
  - But not execution latency of any one thread

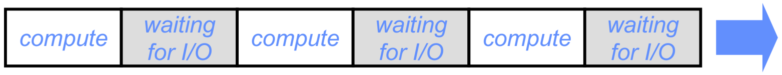
Basic Assumption: the processor has idle resources when running only one thread at a time

The Cheap Cousin of Multithreading: Multiprocessing

- Time-multiplex multiprocessing on uniprocessors started back in 1962
- Even concurrent execution by time-multiplexing improves throughput

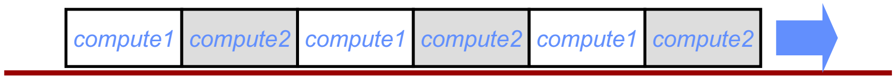
How?

  - 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



compute | waiting for I/O | compute | waiting for I/O | compute | waiting for I/O

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



compute1 | compute2 | compute1 | compute2 | compute1 | compute2

What is a Thread or Context or Process

- Register state
  - PC
  - General purpose registers (integer, FP, …)
- Memory state
  - Control and exception handling registers
  - Page-table base register
  - Private page-table

- What about the state in caches (L1, L2, TLBs)?

Classic Context Switch

- The process
  - Timer interrupt stops a program mid-execution (precise)
  - OS saves away the context of the stopped thread
    - State that occupies unique resources must be copied and saved to a special memory region belonging exclusively to the OS
    - State that occupies commodity resources just needs to be hidden from the other threads (e.g. pages in physical memory)
  - 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

⇒ 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

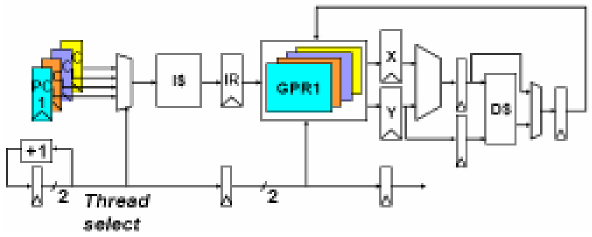
    eliminates copying

  - 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

Simple Multithreaded Processor

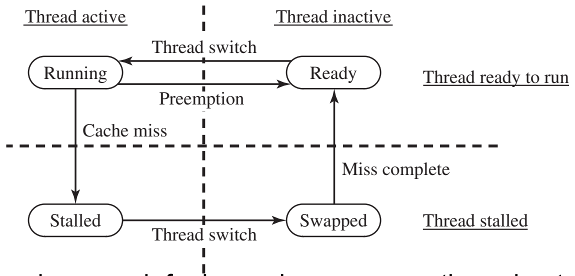


- Some number of threads supported in hardware
  - Switch thread on a cache miss or other high latency event
    - Examples? Trade-offs?
- What happens if all HW threads are blocked?

Example: MIT's Sparcle Processor

- Based SUN SPARC II processors
  - Provided HW contexts for 4 threads, one is reserved for the interrupts
  - Hijacked SPARC II's register windowing mechanism to support fast switching between 4 sets of 32 GPRs
  - Switches context in 4 cycles
    - Why would it take >1 cycle to switch?
- 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

Coarse-grain Multithreading Explained



Thread active

Thread inactive

Thread switch

Running

Ready

Thread ready to run

Preemption

Cache miss

Miss complete

Stalled

Swapped

Thread stalled

Thread switch

- Low-overhead approach for improving processor throughput
  - Also known as "switch-on-event"
- Long history: Denelcor HEP
- Commercialized in IBM Northstar, Pulsar
- Now in many MT processors

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?

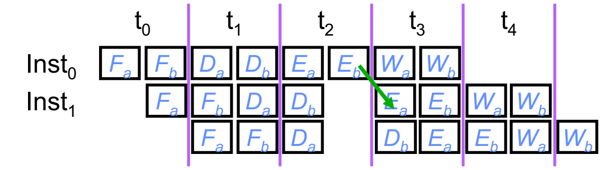
- Not only do you need hardware contexts, switching between contexts must be instantaneous to have any advantage!!
  - What is the cost of 1-cycle context switching?
- If this can be done,
  - Don't need complicated forwarding logic to avoid stalls
  - RAW dependence and long latency operations (multiply, cache misses) do not cause throughput performance loss

Multithreading is a "latency hiding" technique

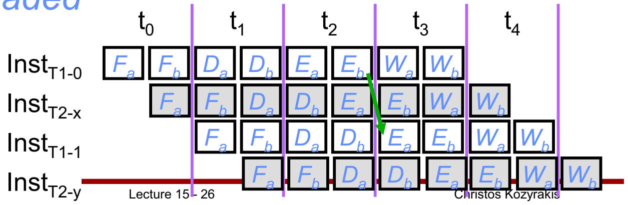
Fine-grain Multithreading

- 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

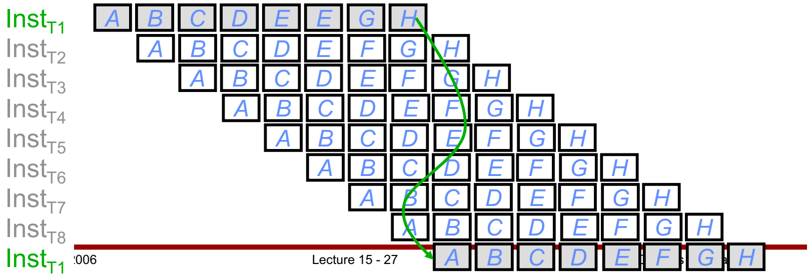

t_0
t_1
t_2
t_3
t_4
$Inst_0$
$Inst_1$
F_a F_b D_a D_b E_a E_b W_a W_b
F_a F_b D_a D_b E_a E_b W_a W_b
F_a F_b D_a D_b E_a E_b W_a W_b

2-way multithreaded superpipelined

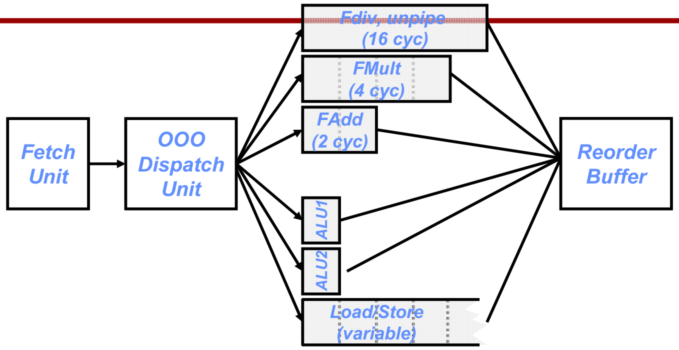

t_0
t_1
t_2
t_3
t_4
$Inst_{T1-0}$
$Inst_{T2-x}$
$Inst_{T1-1}$
$Inst_{T2-y}$
F_a F_b D_a D_b E_a E_b W_a W_b
F_a F_b D_a D_b E_a E_b W_a W_b
F_a F_b D_a D_b E_a E_b W_a W_b
F_a F_b D_a D_b E_a E_b W_a W_b

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



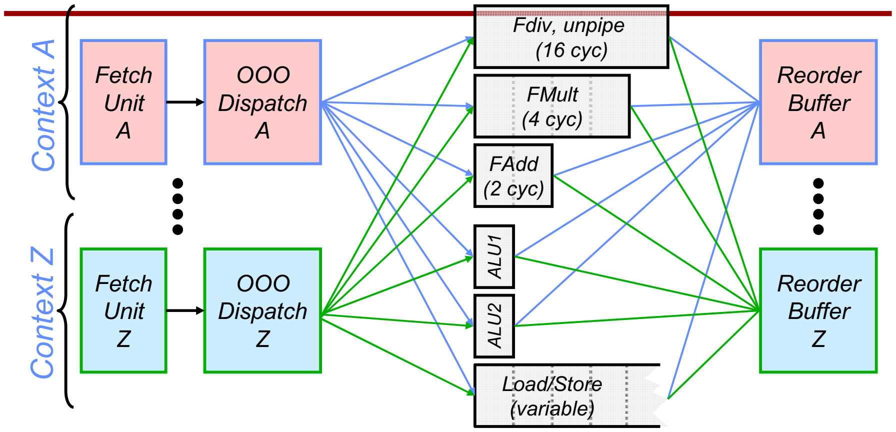
Really Really Fast Context Switches



- 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?

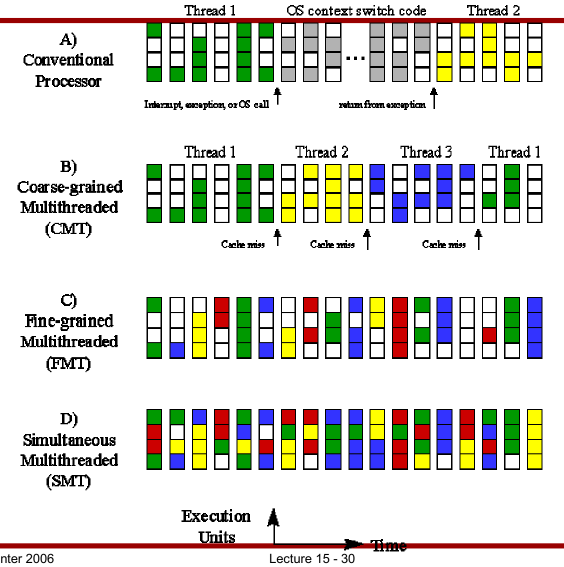
Simultaneous Multi-Threading *[Eggers, et al.]*



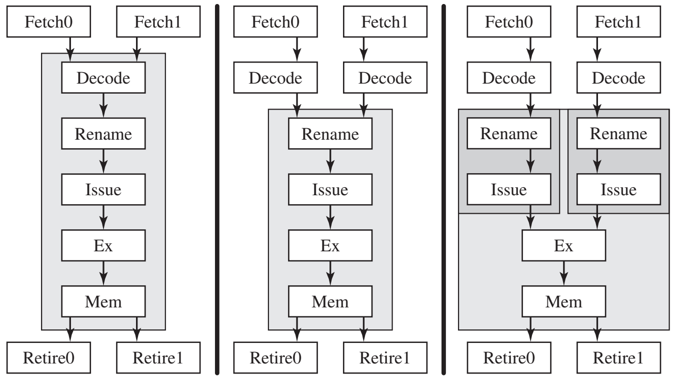
- Dynamic and flexible sharing of functional units between multiple threads

  ⇒ increases utilization ⇒ increases throughtput

Multithreading Options


Thread 1
OS context switch code
Thread 2
A) Conventional Processor
Interrupt, exception, or OS call
return from exception
Thread 1
Thread 2
Thread 3
Thread 1
B) Coarse-grained Multithreaded (CMT)
Cache miss
Cache miss
Cache miss
C) Fine-grained Multithreaded (FMT)
D) Simultaneous Multithreaded (SMT)
Execution Units
Time

SMT Resource Sharing

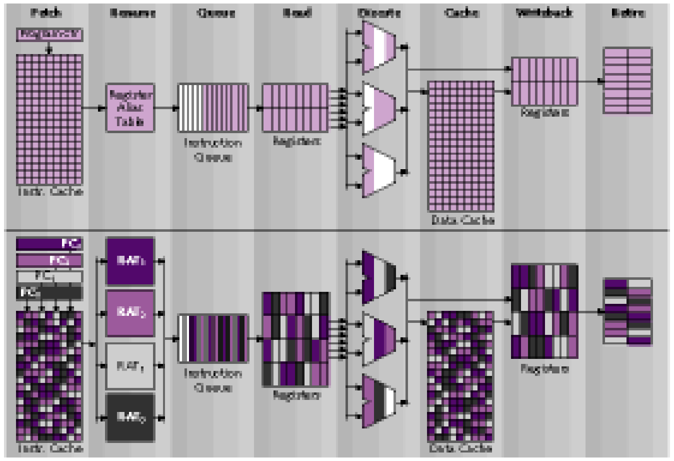


- What are the trade-offs here?

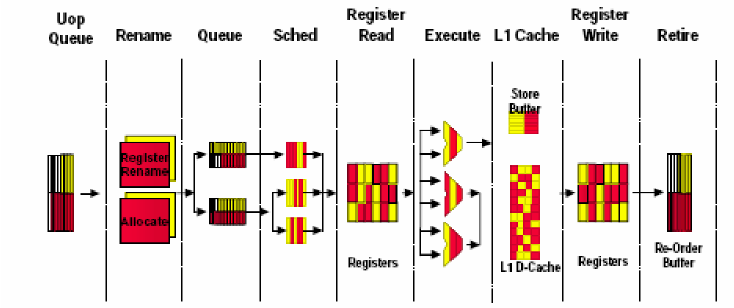
SMT Processors

- 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, …
- Pentium4 HT: 1st commercial SMT CPU (2 threads)
  - Logical CPUs share: caches, FUs, predictors (5% area increase)
  - Separate: RAS, 1st level global branch history table
    - Shared second-level branch history table, tagged with logical processor IDs
    - Why?
  - No logical CPUs can use all entries in queues when 2 threads active
- IBM Power5, Opterons, …

Regular OOO Vs. SMT OOO: the Alpha Approach



Intel Pentium 4 with HT



Fetch Policies for SMT

- Icount policy: fetch from thread with the least instructions in flight
  - Keep hardware counts
- Why does Icount work?
  - Priority to fastest moving threads
  - Avoids thread starvation
- Optimizations
  - Recognize when a thread is busy waiting and reduce it's priority
  - Adapt number of threads running depending on interference
    - ILP per thread
    - Cache misses etc

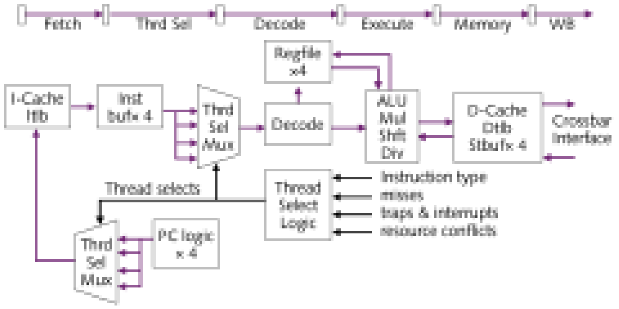
Other SMT Issues

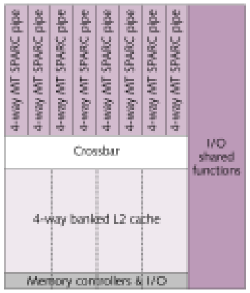
- Adding a SMT to superscalar
  - Single-thread performance is slight worse due to overhead (longer pipeline, longer combinational delays)
  - Over-utilization of shared resources
    - contention for instruction and data memory bandwidth
    - interferences in caches, TLB and BTBs

    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

Sun's UltraSparc T1 or Niagara

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





Explicitly Multithreaded 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 control 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
CMP Next lecture…	Secondary cache, system interconnect	All contexts concurrently active; no switching

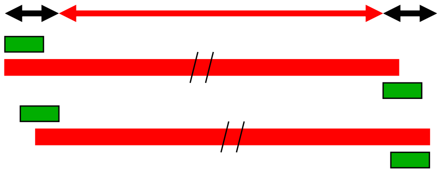
So far, it's all been about throughput with multiple programs

Can MT help with a single program?

Out-of-Order & Increasing Memory Latency

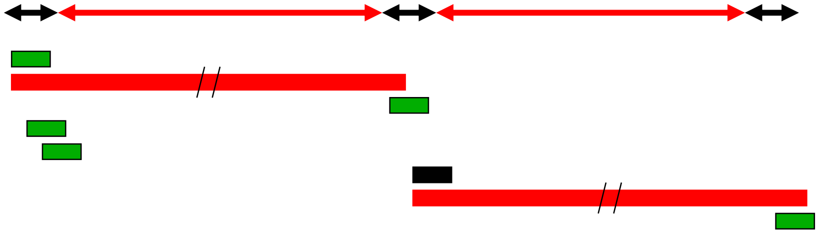
- Good Case

Compute & Memory Phases



- Bad Case

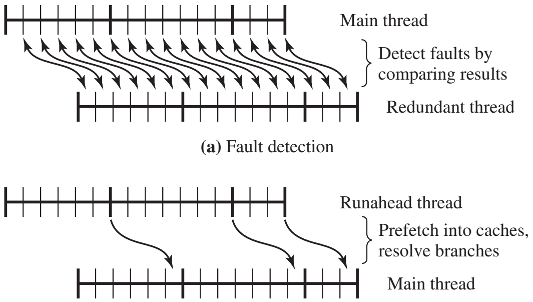
Compute & Memory Phases



Slipstream or Run-ahead 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!!

Illustration


Main thread
Detect faults by comparing results
Redundant thread
Runahead thread
Prefetch into caches, resolve branches
Main thread

(a) Fault detection

(b) Pre-execution

- Why execute the same thread twice?
  - Detect faults
  - Better performance
    - Prefetch, resolve branches

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

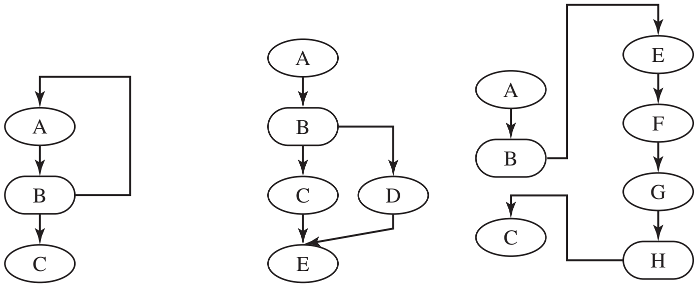
    e.g. start several loop iterations concurrently as different threads, if data dependence is detected, redo the work

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

Implicitly Multithreaded Processors

- Goal: speed up execution of a single thread
- Implicitly break program up into multiple smaller threads, execute them in parallel
- Parallelize loop iterations across multiple processing units
- Usually, exploit control independence in some fashion
- Many challenges:
  - Maintain data dependences (RAW, WAR, WAW) for registers
  - Maintain precise state for exception handling
  - Maintain memory dependences (RAW/WAR/WAW)
  - Maintain memory consistency model
    - Not really addressed in any of the literature
- Active area of research
  - Only a subset is covered here, in a superficial manner

Sources of Control Independence



(a) Loop-closing **(a)** Control-flow convergence **(a)** Call/return

Implicit Multithreading Proposals

	Multiscalar	Disjoint Eager Execution (DEE)	Dynamic Multi-threading (DMT)	Thread-level Speculation (TLS)
Control Flow Attribute Exploited	Control Independence	Control independence Cumulative branch misprediction	Control independence	Control independence
Source of implicit threads	Loop bodies Control-flow joins	Loop bodies Control-flow joins Cumulative branch mispredictions	Loop exits Subroutine returns	Loop bodies
Thread creation mechanism	Software/compiler	Implicit hardware	Implicit Hardware	Software/compiler
Thread creation and sequencing	Program order	Out of program order	Out of program order	Program order
Thread execution	Distributed processing elements	Shared processing elements	Shared multi-threaded processing elements	Separate CPUs
Register data dependences	Software with hardware speculation support	Hardware; no speculation	Hardware; data dependence prediction and speculation	Disallowed; compiler must avoid
Memory data dependences	Hardware-supported speculation	Hardware	Hardware; prediction and speculation	Dependence speculation; checked with simple extension to MESI coherence