Introducing Multi-core Computing / Hyperthreading
Clock Frequency with Time

- Digital VAX-11/780: 5 MHz in 1978
- Sun-4 SPARC: 16.7 MHz in 1986
- MIPS M2000: 25 MHz in 1989
- Digital Alpha 21064: 150 MHz in 1992
- Digital Alpha 21164A: 500 MHz in 1996
- Intel Pentium III: 1000 MHz in 2000
- Intel Pentium4 Xeon: 3200 MHz in 2003
- Intel Nehalem Xeon: 3330 MHz in 2010

Clock rate (MHz)


1.5% / year
40% / year
15% / year

3/9/2017
Why multi-core/hyperthreading?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning
- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)
Multi-core Computing

Complex uniprocessor

L2 Cache

L2 Cache
Jumping to the Right side of the square law

- If $n$ is the number of transistors on a die and $k$ is the number of cores:
  - Area, power $= \mathcal{O}(k*n/k)$
  - Performance $= \mathcal{O}(k*\sqrt{n/k})$

- For example,
  - 1 EV6 $\Leftrightarrow$ 5 EV5 cores (area)
  - 1 EV6 $\Leftrightarrow$ 2.0-2.2 EV5 cores (throughput)
  - 5 EV5 cores $\geq 2$ EV6 cores (throughput)

Performance potentially doubled just by having multiple cores!

The main motivation for having multi-core architectures
Ancillary Advantages

- High performance/watt, performance/area
- Simpler cores: simpler/cheaper to design/verify
- Greater exploitation of parallelism
Motivation for Hardware multithreading (“hyperthreading“)

- Modern processors fail to utilize execution resources well.
- There is no single culprit.
- Attacking the problems one at a time (e.g., *specific* latency-tolerance solutions) always has limited effectiveness.
- However, a *general latency-tolerance solution* which can hide all sources of latency can have a large impact on performance.
Hardware Multithreading

Conventional Processor

Multithreaded Processor

CPU

PC regs

PC regs

PC regs

PC regs
Superscalar Execution

Time (clock cycles)

Issue Slots

[Diagram showing the relationship between time (clock cycles) and issue slots with some slots shaded blue and others white, indicating the execution process.]
Multithreading on Superscalar

Time (proc cycles)

Issue Slots

Thread 1
Thread 2
Context Switch
Superscalar Execution with Coarse-Grained Multithreading

Time (proc cycles)

Issue Slots

Thread 1
Thread 2
Thread 3
Thread 4
Context Switch
Superscalar Execution with Fine-Grain Multithreading
Simultaneous Multithreading

Time (proc cycles)

Issue Slots

Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
The Potential for SMT

![Graph comparing Simultaneous Multithreading, Fine-Grain Multithreading, and Conventional Superscalar]

- **Throughput (Instructions per Cycle)**
  - Simultaneous Multithreading
  - Fine-Grain Multithreading
  - Conventional Superscalar

- **Number of Threads**
  - 1, 2, 3, 4, 5, 6, 7, 8
Goals

Three primary goals for this SMT:

1. Minimize the architectural impact on conventional superscalar design.


3. Achieve significant throughput gains with many threads.
A Conventional Superscalar Architecture

- Fetch up to 8 instructions per cycle
- Out-of-order, speculative execution
- Issue 3 floating point, 6 integer instructions per cycle
An SMT Architecture

- Fetch up to 8 instructions per cycle
- Out-of-order, speculative execution
- Issue 3 floating point, 6 integer instructions per cycle
Real-World SMT

Intel – Hyperthreading
- IBM whitepaper: 20-50% performance benefit
Real-World SMT (2)

- AMD - “It’s all about the cores!”
  - “Our cores are real.” – January, 2010
    - “Hyperthreading is stupid. So is Intel.” - paraphrase
  - October 12, 2011: Bulldozer: 4 “modules”, 8 threads
More SMT

- Network Processors
- CMT processors (Oracle)
- Many Intel processors, etc.
Interconnection Network

- Bus
- Network
- pros/cons?
Programming Model

- Shared Memory -- every processor can name every address location
- Message Passing -- each processor can name only its local memory. Communication is through explicit messages (multicomputer).
- pros/cons?

- find the max of 100,000 integers on 10 processors.
The cache coherence problem

Initially processors 0 and 1 both read location x, initially containing the value 0, into their caches.

When processor 0 writes the value 1 to location x, the stale value 0 for location x is still in processor 1’s cache.
In invalidation-based protocols

- When processor 0 writes \( x = 1 \), the line containing \( x \) is invalidated from processor 1’s cache.
- The next time processor 1 reads location \( x \) it suffers a cache miss, and goes to memory to retrieve the latest copy of the cache line.
- Write-through caches: memory can supply the data.
- Write-back caches: processor 1 asks processor 0 for the latest copy of the cache line.
In update-based protocols

- When processor 0 writes $x = 1$, it sends the new copy of the datum directly to processor 1 and updates the line in processor 1’s cache with the new value.

In either case, subsequent reads by processor 1 now “see” the correct value of 1 for location $x$, and the system is said to be cache coherent.
Invalidation vs. Update

- Invalidation is bad when:
  - single producer and many consumers of data.

- Update is bad when:
  - multiple writes by one processor before data is read by another processor.
  - junk data accumulates in large caches.

- Overall, invalidation schemes are more popular as the default due to easier implementation.
Cache coherence protocols - *snoopy protocols*

- Each cache controller “snoops” all bus transactions
  - Transaction is relevant if it is for a block this cache contains
  - Take action to ensure coherence
    - Invalidate
    - Update
    - Supply value to requestor if Owner
  - Actions depend on the state of the block and the protocol
- Main memory controller also snoops on bus
  - If no cache is owner, then memory is owner
- Simultaneous operation of independent controllers
Processor and Bus Actions

- **Processor:**
  - Load
  - Store
  - Writeback

- **Bus**
  - GetShared (GETS): Get without intent to modify, data could come from memory or another cache
  - GetExclusive (GETX): Get with intent to modify, must invalidate all other caches’ copies
  - PutExclusive (PUTX): cache controller puts contents on bus and memory is updated

- **Definition:** cache-to-cache transfer occurs when another cache satisfies GETS or GETX request
Simple 2-State Invalidate Snooping Protocol

Notation: observed event / action taken
A 3-State Write-Back Invalidation Protocol

- 2-State Protocol
  - Simple hardware and protocol
  - Uses lots of bandwidth (every write goes on bus!)

- 3-State Protocol (MSI)
  - Modified
    - One cache exclusively has valid (modified) copy -> Owner
    - Memory is stale
  - Shared
    - $>=1$ cache and memory have valid copy (memory = owner)
  - Invalid (only memory has valid copy and memory is owner)

- Must invalidate all other copies before entering modified state
- Requires bus transaction (order and invalidate)
Used in Silicon Graphics machines
A 4-state protocol

- Consider writes to private variables
Illinois Protocol: State Diagram

- **I** (Initial State): Read miss from mem.
  - Proc. induced
  - Bus induced

- **M** (Miss State): Read/Write Hit
  - Read miss from cache
  - Bus read miss
  - Write hit

- **S** (Shared State): Read hit
  - Bus read miss
  - Write hit

- **E** (Exclusive State): Read hit
  - Bus read miss and bus read miss

States and transitions:
- **I** to **M**: Bus write miss
- **M** to **I**: Bus write miss
- **M** to **E**: Bus read miss
- **E** to **M**: Read miss from mem.
- **E** to **S**: Bus read miss
- **S** to **E**: Read miss from cache
- **S** to **M**: Write hit
Example: P2 reads A (A only in memory)

- Read miss from mem.
- Read hit
- Bus read miss
- Bus write miss
- Bus write miss
- Read hit and bus read miss
- Read miss from cache
- Hit
- Write hit
- Write hit
- Write hit
- Bus read miss
- Bus write miss
- Bus write miss
- Proc. induced
- Bus induced
Example: P3 reads A (A comes from P2)

Both P2 and P3 will have A in state S

Proc. induced

Bus induced

Read miss from mem.

Bus write miss

Write hit

Bus read miss

Read hit

Read miss from cache

Write hit

Write miss

Hit

I

M

S

E
Example: P4 writes A (A comes from P2)

P2 and P3 will have A in state I; P4 will be in state M.

Proc. induced

Bus induced

Read miss from mem.

P4 writes A (A comes from P2)

Read hit

Bus read miss

Read hit and bus read miss

Read miss from cache

Bus write miss

Write hit

Write hit

Hit

Bus write miss

Write miss

Bus read miss

P4 writes A (A comes from P2)
MOESI – motivation

• Consider cache 0 that wants to read from x:
• If cache 0 line is “Invalid” and cache 1 line is “Modified”, then:
  – cache 1 needs to write back the data to the main memory
• Afterwards, cache 0 changes x, cache 1 wants to read from x:
  – cache 0 invalidates, writes back...
• The line is ping-ponged between the caches
  – each time should be written-back to the main memory