CSE 530
Fundamentals of Computer Architecture
Spring 2021

Multithreading (Hyperthreading)

John (Jack) Sampson (cse.psu.edu/~sampson)

Course material on Canvas

[Adapted in part from Mary Jane Irwin, V. Narayanan, Amir Roth, Milo Martin, and others]

Performance and utilization

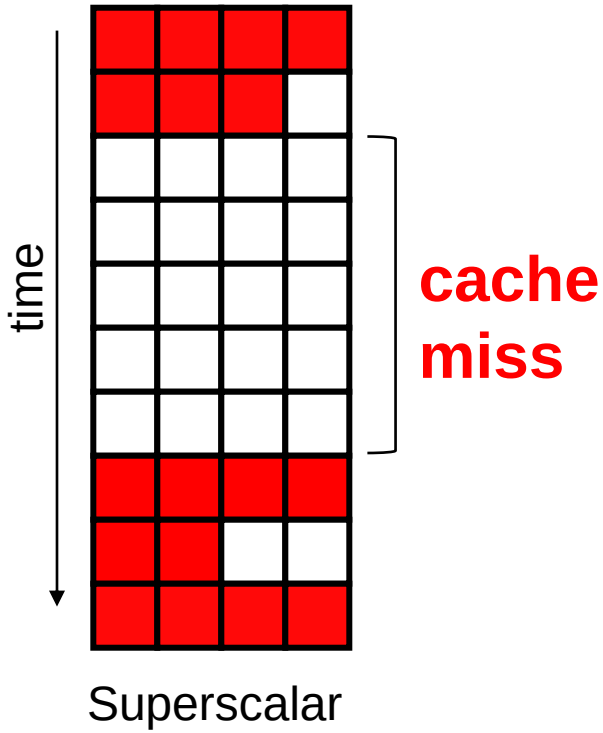
- Performance (IPC) important
- Utilization (actual IPC / peak IPC) important too

- Even moderate superscalars (e.g., 4-way) not fully utilized
  - Average sustained IPC: 1.5–2 $\rightarrow$ < 50% utilization
    - Mispredicted branches
    - Cache misses, especially L2/L3
    - Data dependences

- **Multi-threading (MT)**
  - Improve *utilization* by multiplexing multiple (process) threads on single CPU
  - One thread cannot fully utilize CPU? Maybe 2, 4 (or 100) can

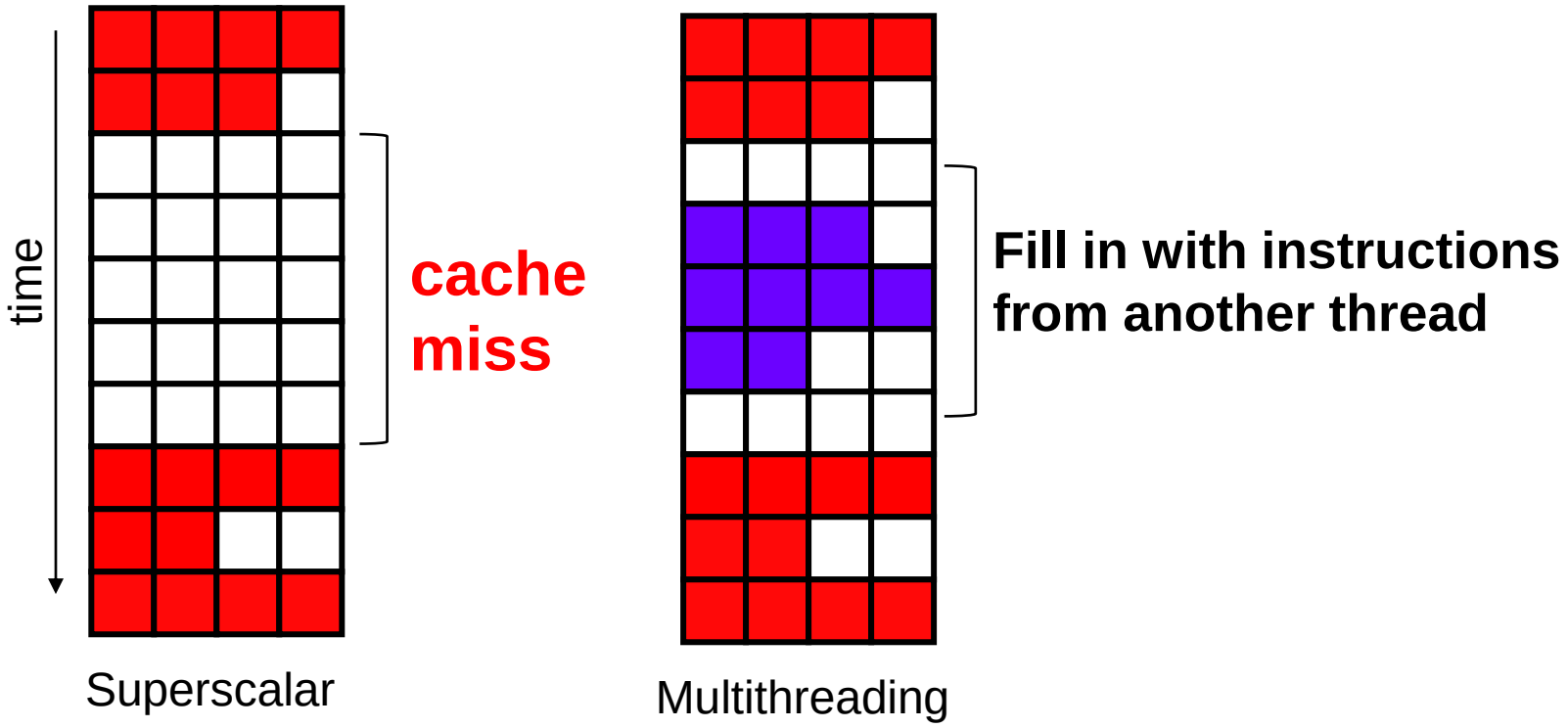
Superscalar under-utilization

- Time evolution of issue slot
  - 4-issue processor



Simple multithreading

- Time evolution of issue slot
  - 4-issue processor



- Where does it find a thread?
  - Depends on both SW and HW
  - Multiple models exist (many ~same as multicore/shared memory)

Latency vs throughput

- **MT trades (single-thread) latency for throughput**
  - – Sharing processor degrades latency of individual threads
  - + But improves aggregate latency of both threads
  - + Improves utilization
- Example
  - Thread A: individual latency=10s, latency with thread B=15s
  - Thread B: individual latency=20s, latency with thread A=25s
  - Sequential latency (first A then B or vice versa): 30s
  - Parallel latency (A and B simultaneously): 25s
  - – MT slows each thread by 5s
  - + But improves total latency by 5s
- Different workloads have different parallelism
  - SpecFP has lots of **ILP**; i.e., can make use of an 8-wide machine
  - Server workloads have **TLP** (Thread Level Parallelism), i.e., have multiple threads that can run in parallel

MT implementations: Similarities

- How do multiple threads share a single processor?
  - Different sharing mechanisms for different kinds of structures
  - Depend on what kind of state the structure stores

- **No state**: ALUs
  - Dynamically shared
- **Persistent hard state (aka "context")**: PC, registers
  - Must be replicated
- **Persistent soft state**: TLBs, caches, bpred (BTB), IQ
  - Dynamically partitioned (like on a multi-programmed uni-processor)
    - TLBs need thread ids, caches/bpred tables don't
  - Exception: **ordered "soft" state** (BHR, RAS) is replicated
- **Transient state**: pipeline latches, ROB, LSQ
  - Partitioned … somehow

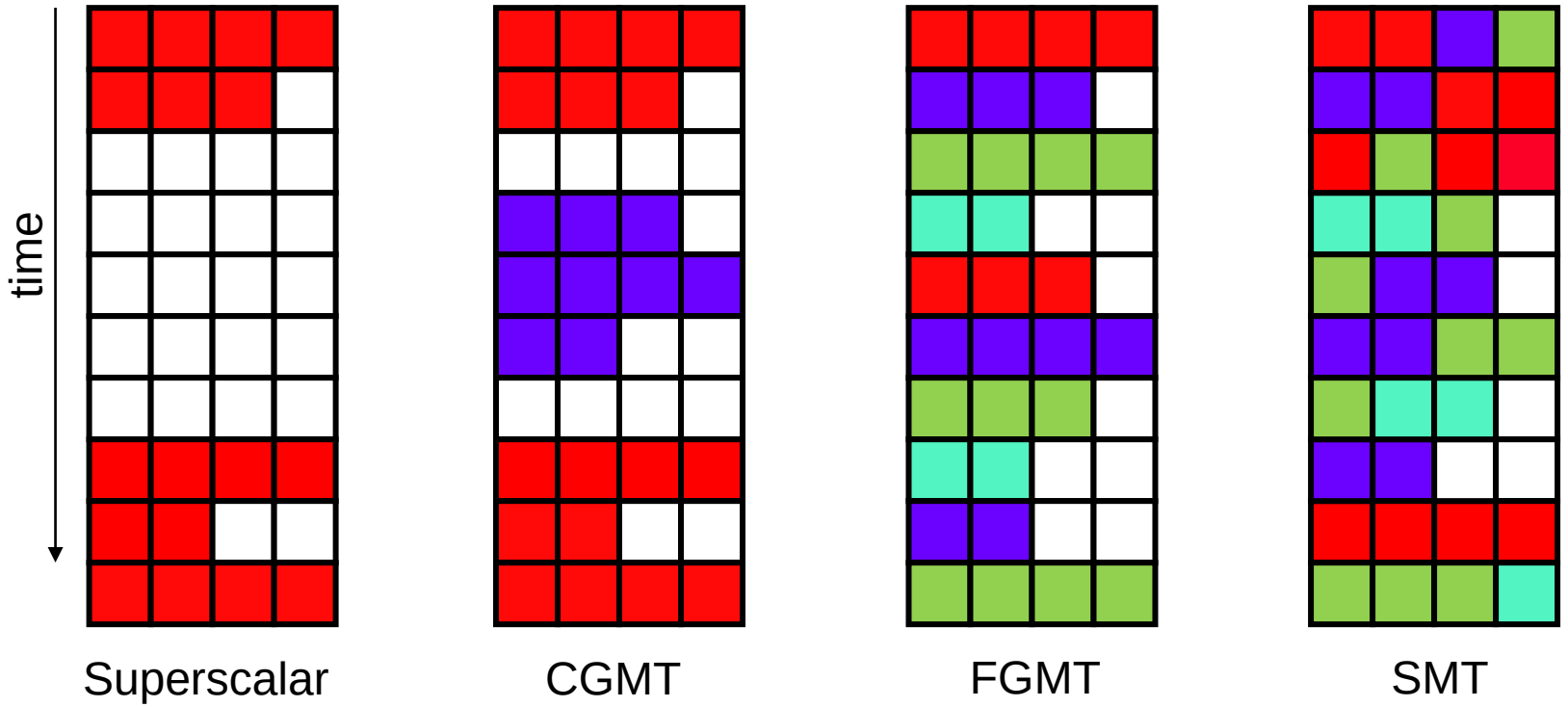
MT implementations: Differences

- Main question: **thread scheduling policy**
  - When to switch from one thread to another?
- Related question: **pipeline partitioning**
  - How exactly do threads share the pipeline itself?

- Choice depends on
  - What kind of latencies (specifically, length of) you want to tolerate
  - How much single thread performance you are willing to sacrifice
- Three design choices
  - Coarse-grain multithreading (**CGMT**)
  - Fine-grain multithreading (**FGMT**)
  - Simultaneous multithreading (**SMT**)

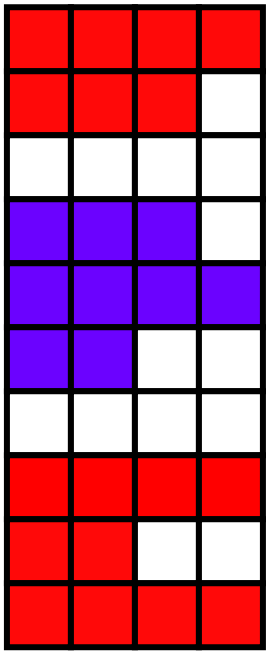
The standard multithreading pictures

- Time evolution of issue slots
  - Color = thread



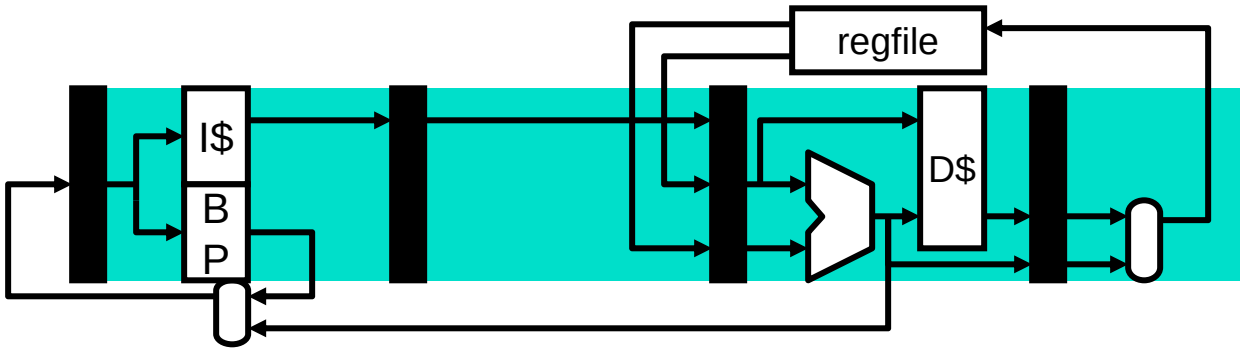
Coarse-Grain MultiThreading (CGMT)

- + Sacrifices very little single thread performance (of one thread)
- – Tolerates only long latencies (e.g., misses to main memory)
- Thread scheduling policy
  - Designate a “preferred” thread (e.g., thread A)
  - Switch to thread B on thread A L2 miss
  - Switch back to A when A L2 miss returns
- Pipeline partitioning
  - None, flush on switch
  - – So can’t tolerate latencies shorter than twice pipeline depth
    - Need short in-order pipeline for good performance
- Example: IBM Northstar/Pulsar

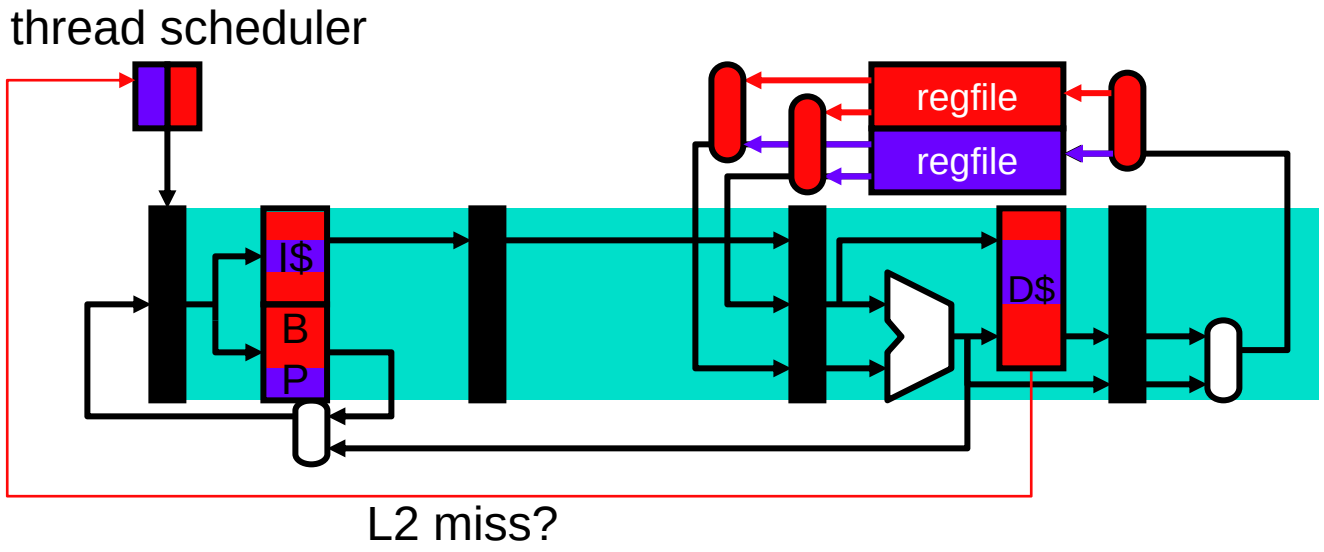


CGMT

Coarse-Grain MultiThreaded architecture

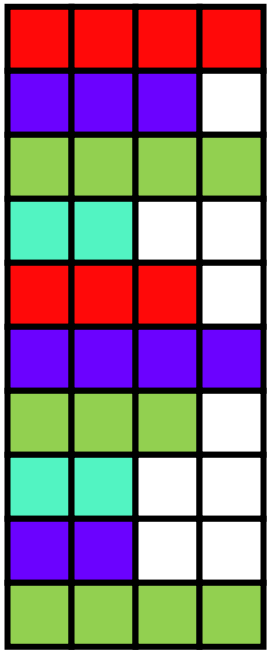


- CGMT



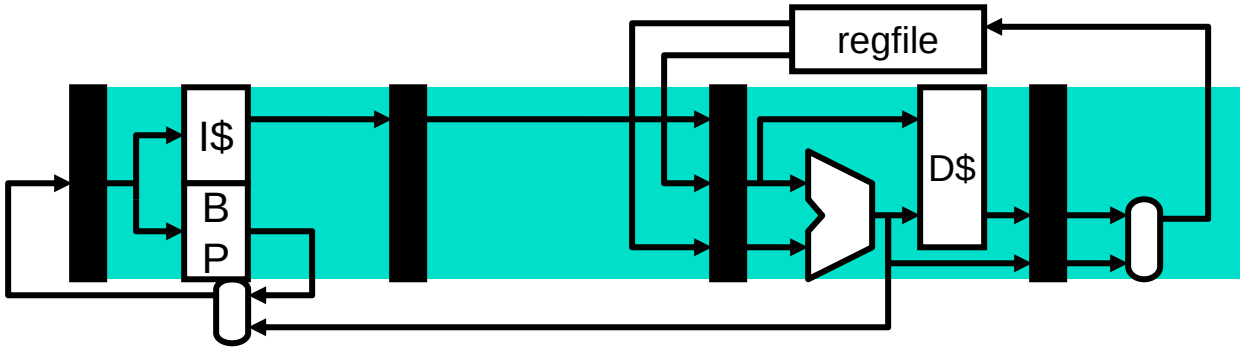
Fine-Grain MultiThreading (FGMT)

- – Sacrifices significant single thread performance
- \+ Tolerates latencies (e.g., L2 misses, mispredicted branches, etc.)
- Thread scheduling policy
  - Switch threads every cycle (round-robin, skipping stalled threads), L2 miss or no
- Pipeline partitioning
  - Dynamic, no flushing
    - Length of pipeline doesn't matter so much
- – Need a lot of threads
- Extreme example: Denelcor HEP
  - So many threads (100+), it didn't even need caches
  - Failed commercially
- Sun's UltraSPARC, but not so popular today for CPUs
  - Many threads → many register files

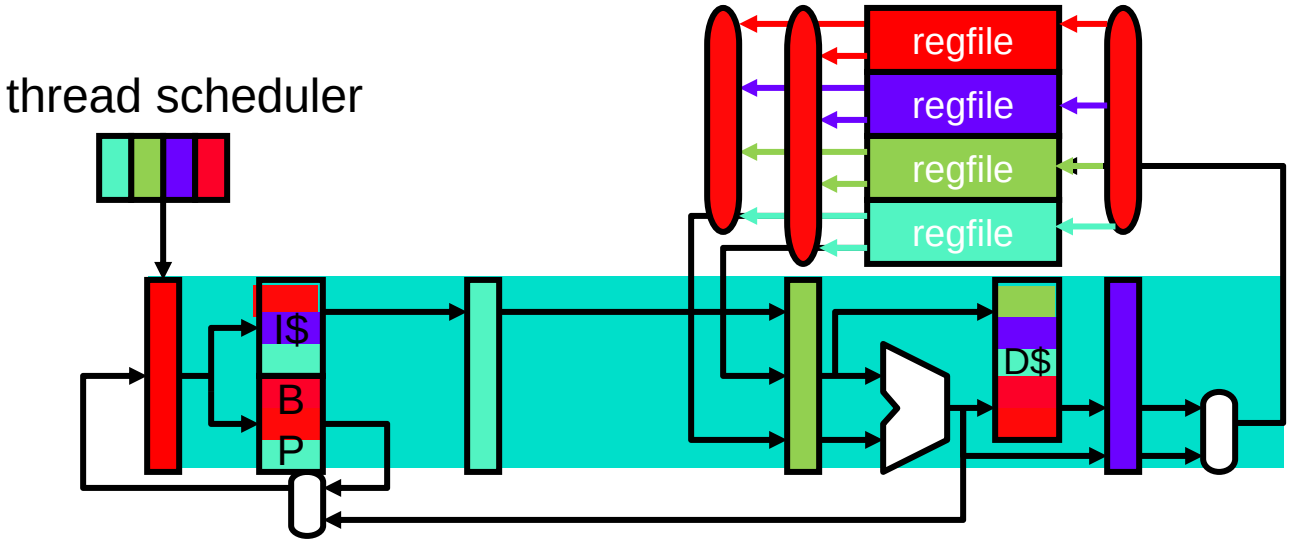


FGMT

Fine-Grain MultiThreaded architecture

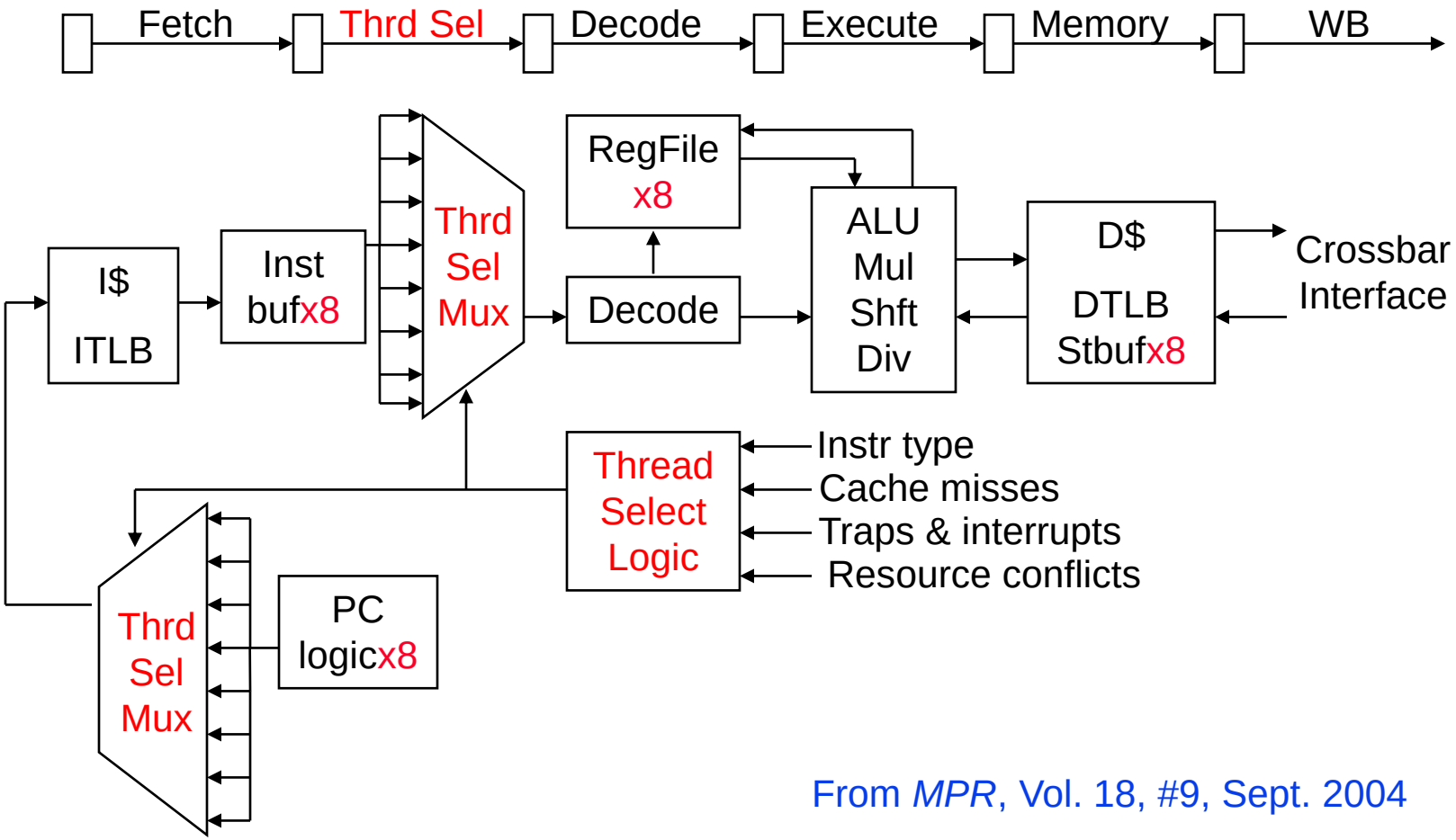


- FGMT
  - **Multiple threads in pipeline at once** = (many) more threads



Niagara's FGMT integer pipeline

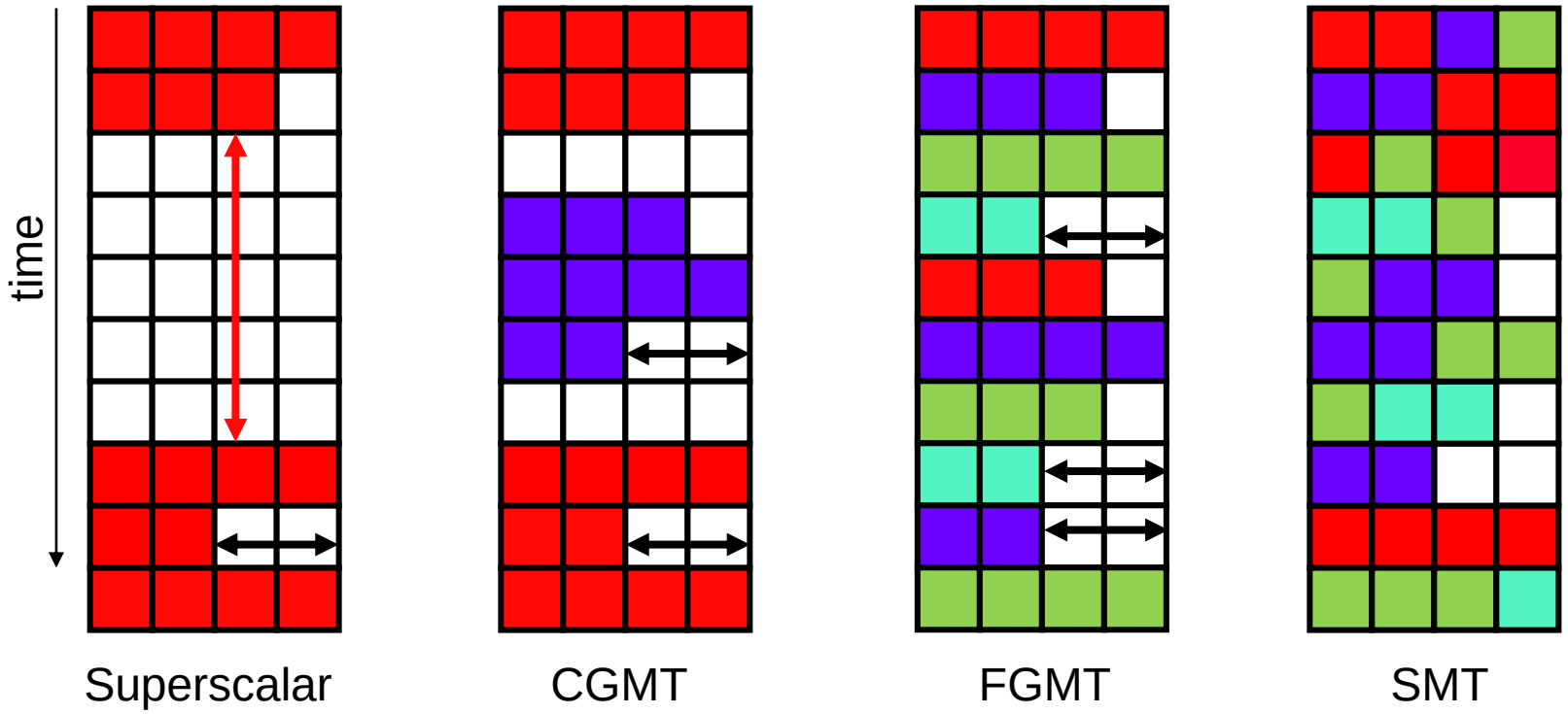
- Cores are simple (single-issue, 6 stage, no branch prediction), small, and power-efficient



From *MPR*, Vol. 18, #9, Sept. 2004

Vertical and horizontal under-utilization

- CGMT and FGMT reduce **vertical under-utilization**
  - Loss of all slots in an issue cycle
- Do not help with **horizontal under-utilization**
  - Loss of some slots in an issue cycle (in a superscalar processor)



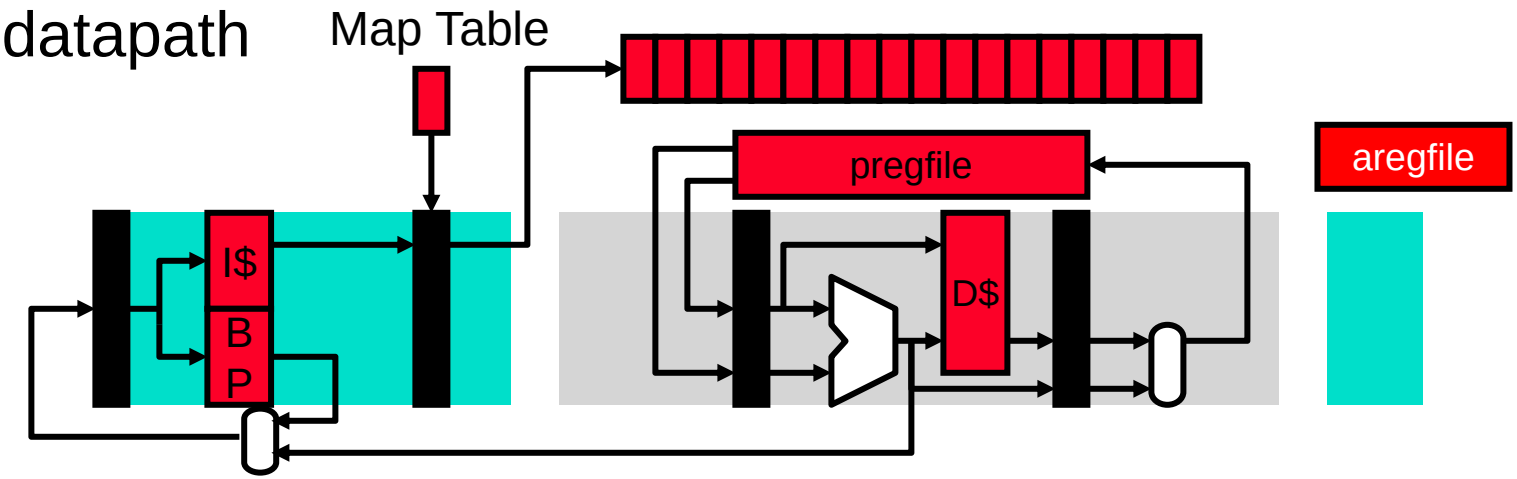
Simultaneous MultiThreading (SMT)

- What can issue instr's from multiple threads in one cycle?
  - Same thing that issues instr's from multiple parts of same program…
  - …out-of-order execution

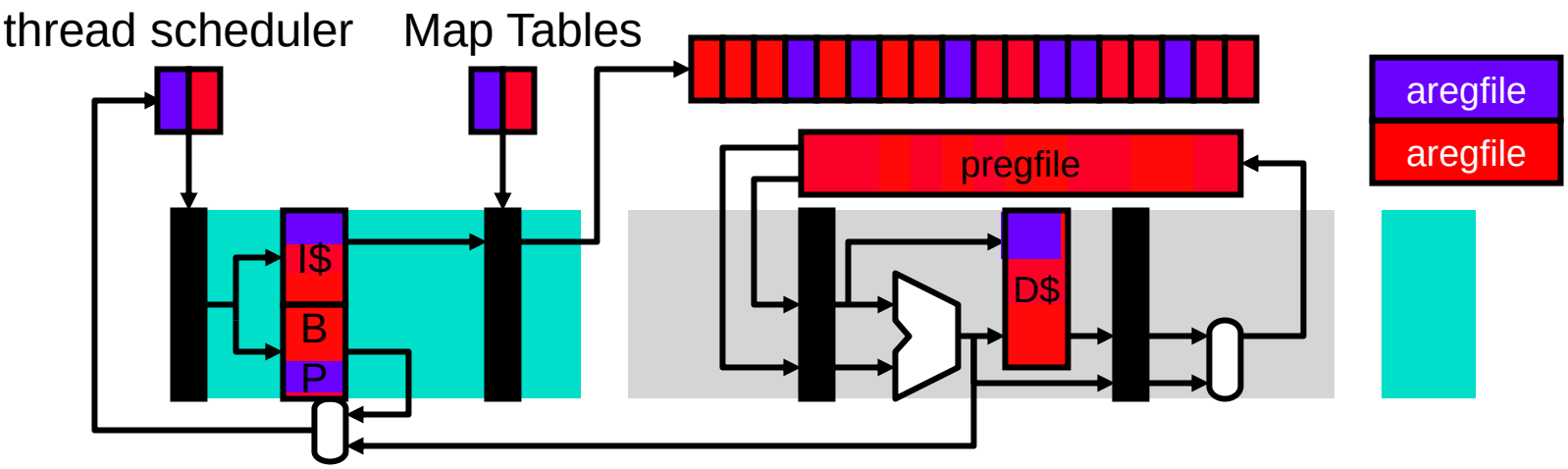
- **Simultaneous multithreading (SMT)**: OoO + FGMT
  - AKA (by Intel's ™-happy marketers) **"hyper-threading"**
  - Observation: once instr's are renamed, scheduler doesn't care which thread they come from (well, for non-loads at least)
  - Some examples
    - IBM Power5: 4-way, 2 threads; IBM Power7: 4-way?, 4 threads
    - Intel Pentium4: 3-way, 2 threads; Intel Core Nehalem: 4-way, 2 threads
    - AMD Bulldozer: 4-way, 2 threads
    - Alpha 21464: 8-way issue, 4 threads (canceled)
    - Notice a pattern? #threads (T) * 2 = #issue width (N)

Simultaneous MultiThreaded architecture

- OoO datapath

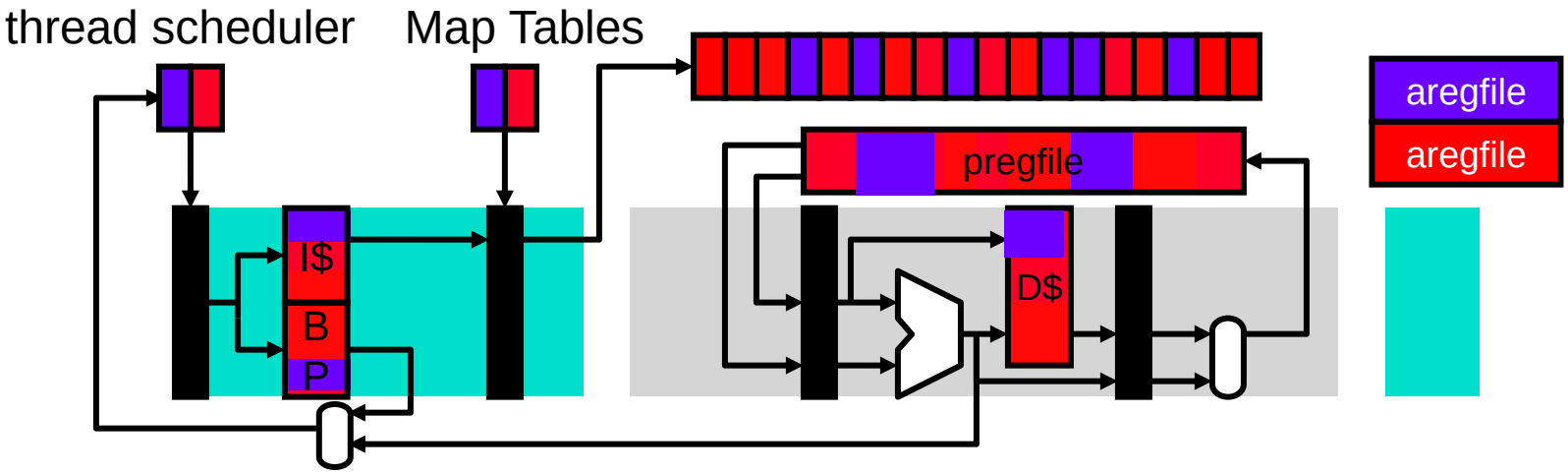


- SMT
  - Replicate Map Table, share (larger) physical register file



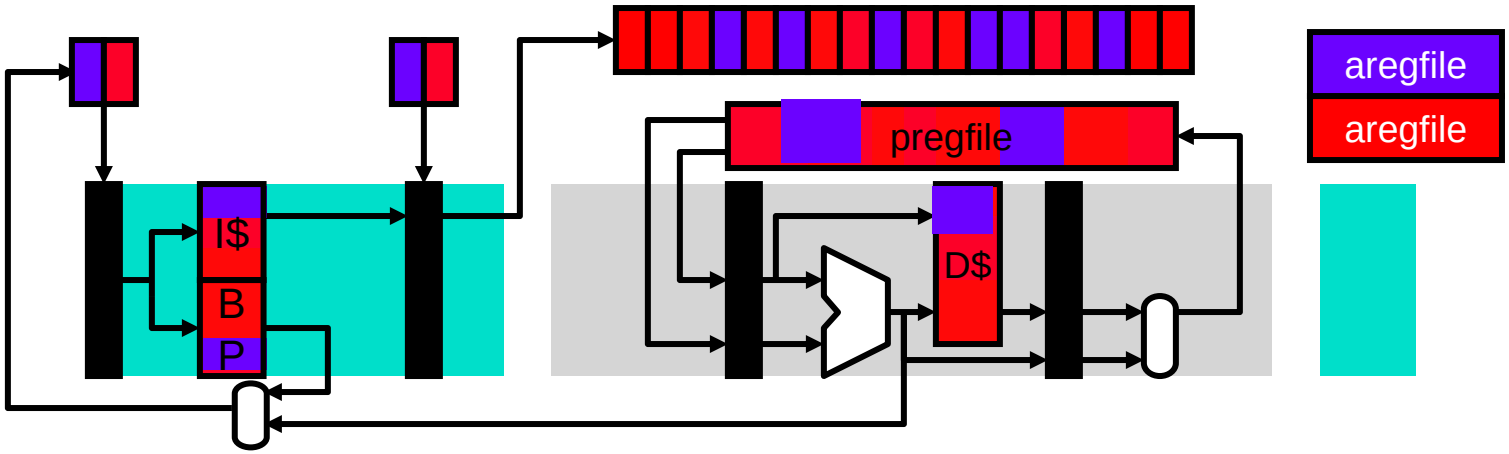
SMT resource partitioning

- PRegFile and IQ entries can be shared at fine-grain
  - Physically unordered and so fine-grain sharing is possible
- How are physically ordered structures (e.g., ROB, LSQ) shared?
  - Fine-grain sharing (below) would entangle commit (and squash)
  - Allowing threads to commit independently is important



Static/dynamic ROB & LSQ partitioning

- **Static partitioning**
  - T equal-sized contiguous partitions
  - ± No starvation, sub-optimal utilization (fragmentation)
- **Dynamic partitioning**
  - P > T partitions, available partitions assigned on need basis
  - ± Better utilization, possible starvation
  - ICOUNT: fetch policy prefers thread with fewest in-flight instr's
- Couple both with larger ROBs/LSQs



Sharing soft state

- Caches are shared naturally…
  - Physically-tagged: address translation distinguishes different threads
- … but TLBs need explicit thread IDs to be shared
  - Virtually-tagged: entries of different threads indistinguishable
  - Thread IDs are only a few bits: enough to identify on-chip contexts

Sharing soft state

- Thread IDs make sense on BTB (branch target buffer)
  - BTB entries are already large, a few extra bits / entry won't matter
  - Different thread's target prediction → automatic mis-prediction
- … but not on a BHT (branch history table)
  - BHT entries are small, a few extra bits / entry is huge overhead
  - Different thread's direction prediction → mis-prediction not automatic
- Ordered soft-state should be replicated
  - Ex: Branch History Register (BHR), Return Address Stack (RAS)
  - Otherwise it becomes meaningless… Fortunately, it is typically small

Other multithreading issues

- Cache interference
  - General concern for all MT variants
  - Can the working sets of multiple threads fit in the caches?
  - Shared memory SPMD threads help here
    - \+ Same instr's → share I$
    - \+ Shared data → less D$ contention
    - \- MT is good for workloads with shared instr/data
  - To keep miss rates low, SMT might need a larger L2 (which is OK)
    - \- Out-of-order tolerates L1 misses

- Large physical register file (and Map Table)
  - physical registers = (**#threads** * #arch-regs) + #in-flight instr's
  - Map Table entries = (**#threads** * #arch-regs)

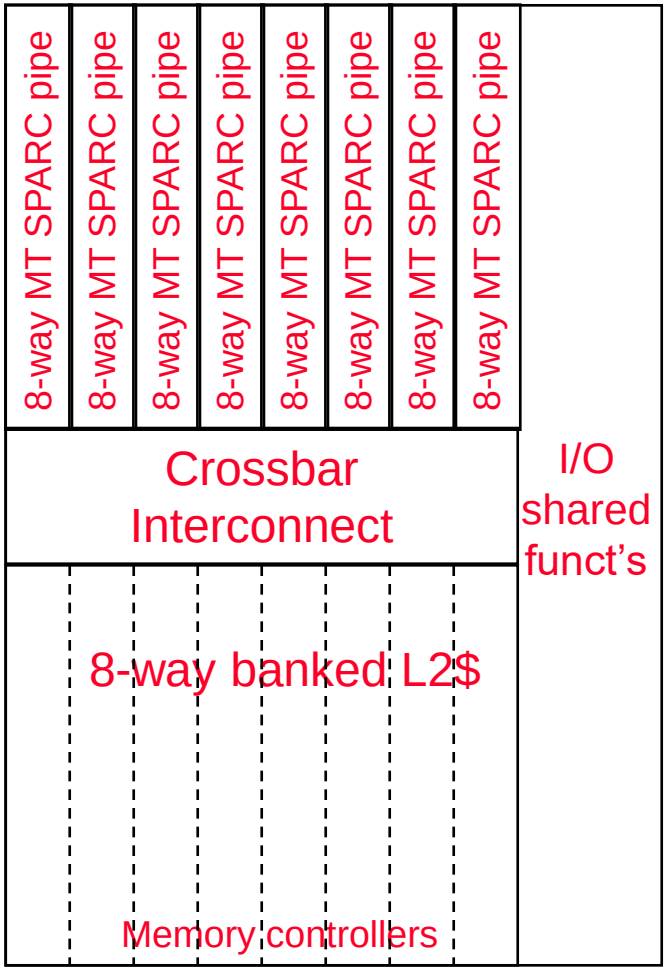
Multithreading vs multicore

- If you wanted to run multiple threads would you build a
  - A multicore: multiple separate pipelines?
  - A multithreaded processor: a single larger pipeline?
- **Both will get you throughput on multiple threads**
  - A multicore core will be simpler, possibly faster clock
  - SMT will get you better performance (IPC) on a single thread
    - SMT is basically an ILP engine that converts TLP to ILP
    - Multicore is mainly a TLP (thread-level parallelism) engine
- **Do both**
  - Power8 – 12 cores, each with 8-way SMT = 96 threads!
  - Sun's Niagara (UltraSPARC)
    - 8 processors, each with 4-threads (non-SMT threading)
    - 1Ghz clock, in-order, short pipeline (6 stages or so)
    - Designed for power-efficient "throughput computing"

FGMT Example: Sun's Niagara (UltraSPARC T2)

- Eight FGMT single-issue, in-order cores

	Niagara 2
Data width	64-b
Clock rate	1.4 GHz
Cache (I/D/L2)	16K/8K/4M
Issue rate	1 issue
Pipe stages	6 stages
TLB entries	64I/64D
Memory BW	60+ GB/s
Transistors	??? million
Power (max)	<95 W



Aside: Speculative multithreading (SpMT)

- **Speculative multithreading**
  - Use multiple threads/processors for **single-thread performance**
  - Speculatively parallelize sequential loops, that might not be parallel
    - Processing elements (called PE) arranged in logical ring
    - Compiler or hardware assigns iterations to consecutive PEs
    - Hardware tracks logical order to detect mis-parallelization
  - Techniques for doing this on non-loop code too
    - Detect reconvergence points (function calls, conditional code)
  - Effectively chains ROBs of different processors into one big ROB
    - Global commit "head" travels from one PE to the next
    - Mis-parallelization flushes one PEs, but ***not all*** PEs
  - Also known as split-window or "Multiscalar"
  - Not commercially available yet…
    - But it is the "biggest idea" from academia not yet adopted

Aside: Multithreading for reliability

- Can multithreading help with reliability?
  - Design bugs/manufacturing defects? No
  - Gradual defects, e.g., thermal wear? No
  - Transient errors? Yes

- **Staggered redundant multithreading (SRT)**
  - Run two copies of program at a slight stagger
  - Compare results, difference? Flush both copies and restart
  - Significant performance overhead

Multithreading summary

- Key considerations:
  - Latency vs. throughput
  - Partitioning different processor resources
  - Three multithreading variants
    - Coarse-grain: no single-thread degradation, but long latencies only
    - Fine-grain: other end of the trade-off
    - Simultaneous: fine-grain with out-of-order
  - Multithreading vs. chip multiprocessing
    - How best to allocate finite resources?
    - Can do both!

- SMT as a parallelism transformer:
  - Turns TLP into ILP
  - Ergo, requires an aggressive ILP engine to get the best value

Next reading assignment – Paper #F9

- Read “Exploiting Choice” Dean Tullsen, et.al., ISCA, 1995. A < 30 second elevator summary of the paper

- Favorite one liner/quote from paper
- And answer questions
  - Q1: What structures are dynamically shared among all the threads? Which, if any, of the structures are not dynamically shared (that is, are replicated or statically partitioned among the threads)?
  - Q2: What is “IQ Clogging”? What does this paper propose to address this issue?