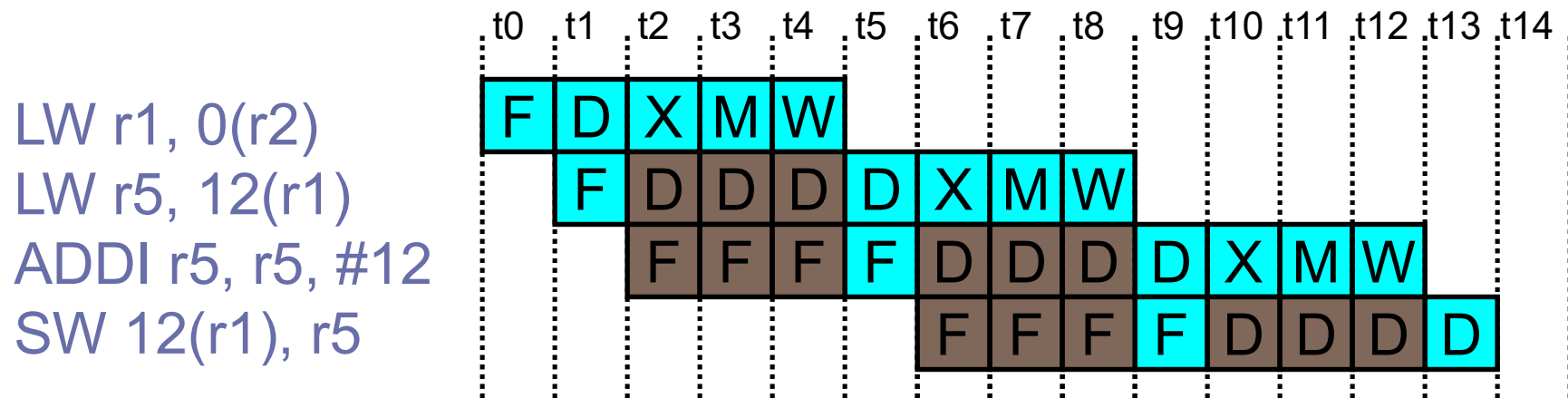


# Multithreading Architectures

*Joel Emer*  
Computer Science & Artificial Intelligence Lab  
M.I.T.

# Pipeline Hazards



- Each instruction may depend on the previous one

What can be done to cope with this?

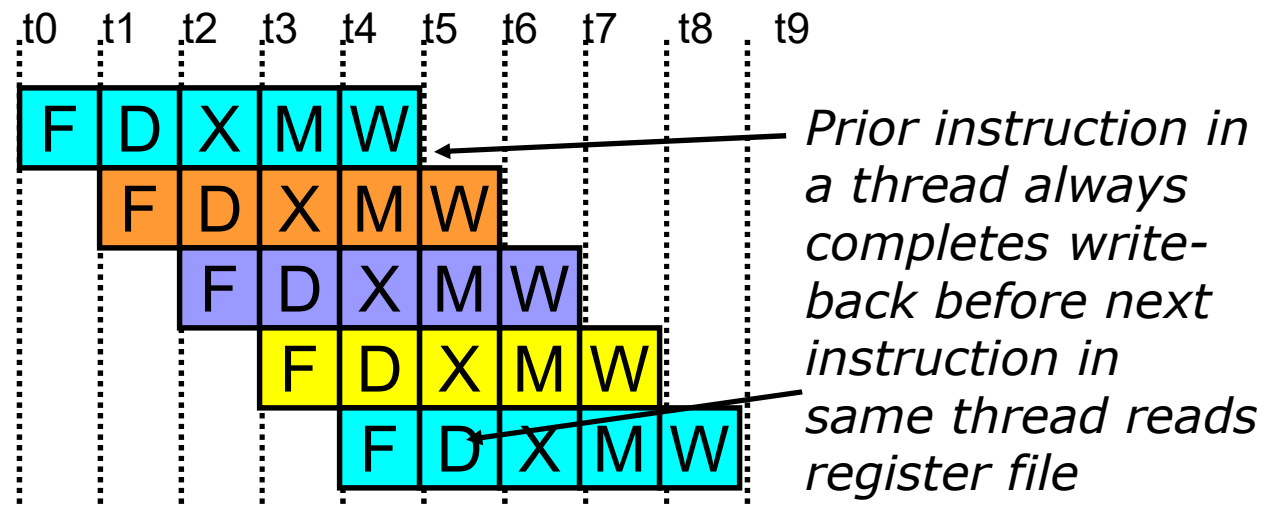
- Even bypassing, speculation and finding something else to do (via O-O-O) does not eliminate all delays

# Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

*Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe*

T1: LW r1, 0(r2)  
T2: ADD r7, r1, r4  
T3: XORI r5, r4, #12  
T4: SW 0(r7), r5  
T1: LW r5, 12(r1)



# CDC 6600 Peripheral Processors (Cray, 1964)

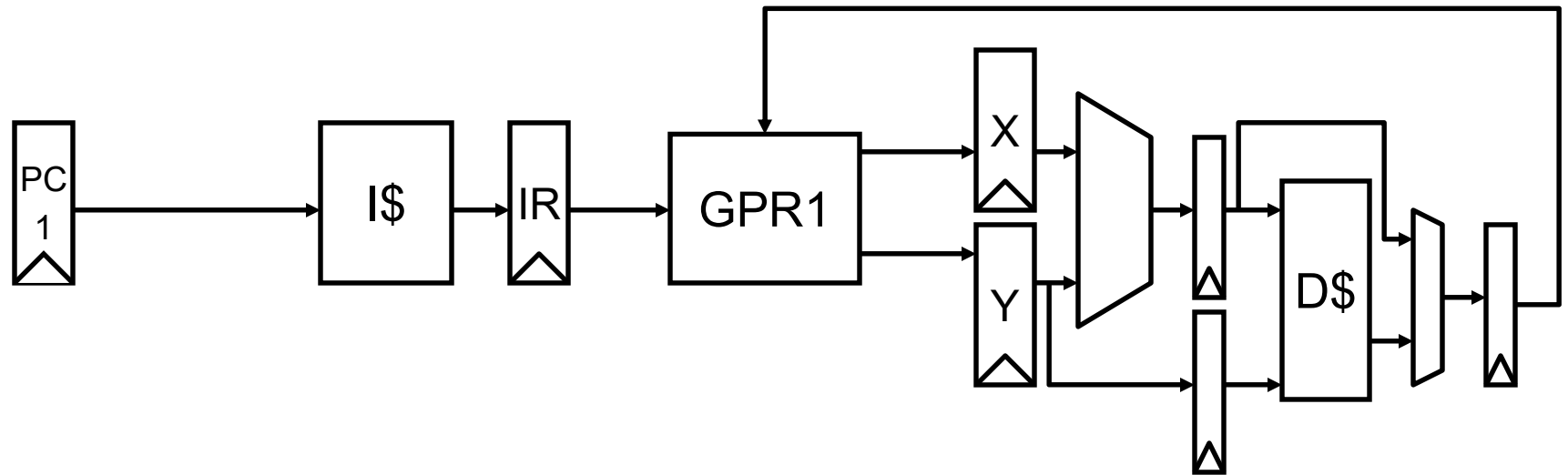
---



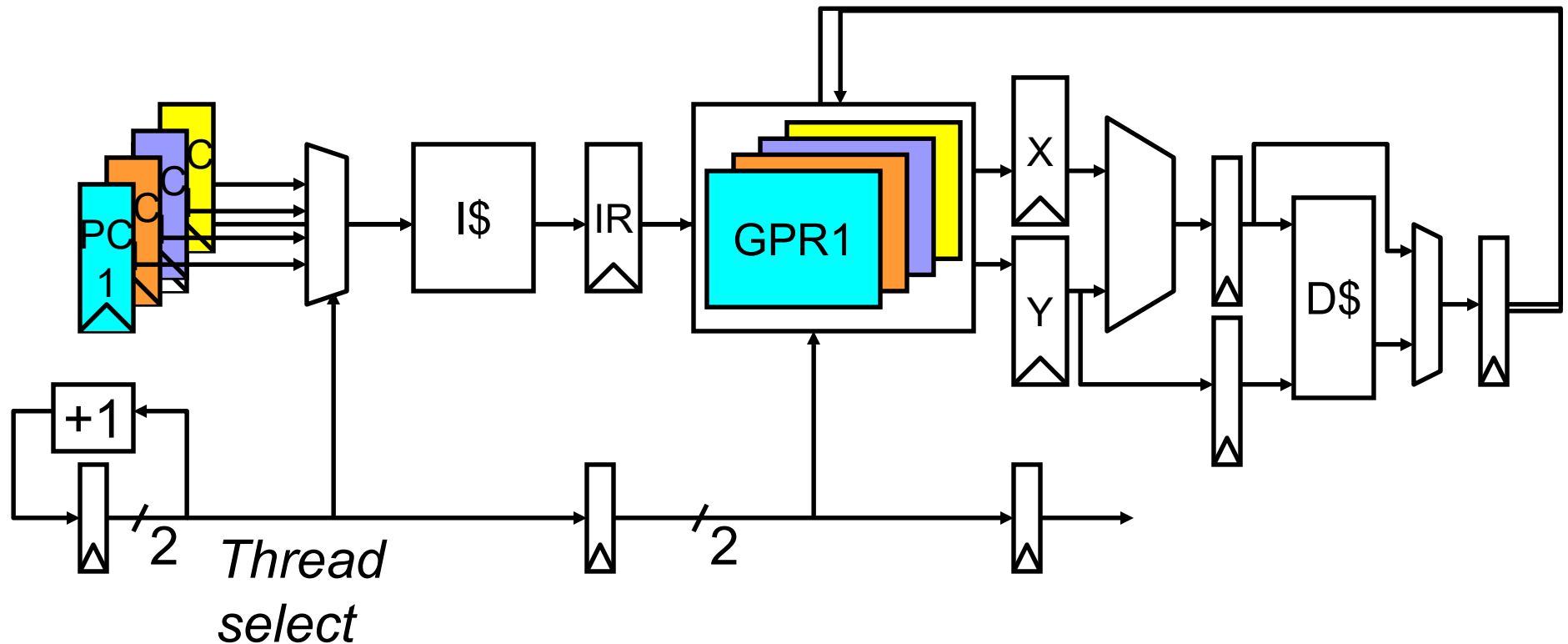
- First commercial multithreaded hardware
- 10 “virtual” I/O processors
- Fixed interleave on simple pipeline
- Pipeline has 100ns cycle time
- Each virtual processor executes one instruction every 1000ns

# Simple Single-threaded Pipeline

---



# Simple Multi-threaded Pipeline



Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage


# Multithreading Costs

---

- Each thread needs its own user architectural state
  - PC
  - GPRs (CDC6600 PPU – accumulator-based architecture)
- Also, needs its own system architectural state
  - Virtual memory page table base register
  - Exception handling registers
- *Other costs?*
- Appears to software (including OS) as multiple, albeit slower, CPUs

# Thread Scheduling Policies

---

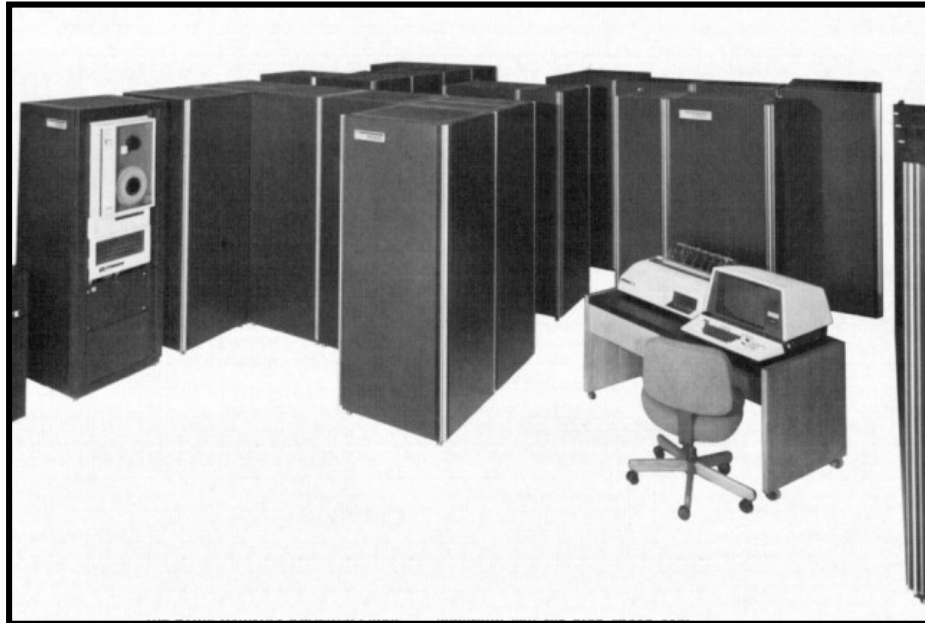
- Fixed interleave (*CDC 6600 PPUs, 1965*)
    - Each of  $N$  threads executes one instruction every  $N$  cycles
    - If thread not ready to go in its slot, insert pipeline bubble
  - Software-controlled interleave (*TI ASC PPUs, 1971*)
    - OS allocates  $S$  pipeline slots among  $N$  threads
    - Hardware performs fixed interleave over  $S$  slots, executing whichever thread is in that slot
- 
- Hardware-controlled thread scheduling (*HEP, 1982*)
    - Hardware keeps track of which threads are ready to go
    - Picks next thread to execute based on hardware priority scheme



# Denelcor HEP

(Burton Smith, 1982)

---



First commercial machine to use hardware threading in main CPU

- 120 threads per processor
- 10 MHz clock rate
- Up to 8 processors
- Precursor to Tera MTA (Multithreaded Architecture)

# Tera MTA

(Burton Smith, 1990-97)

---



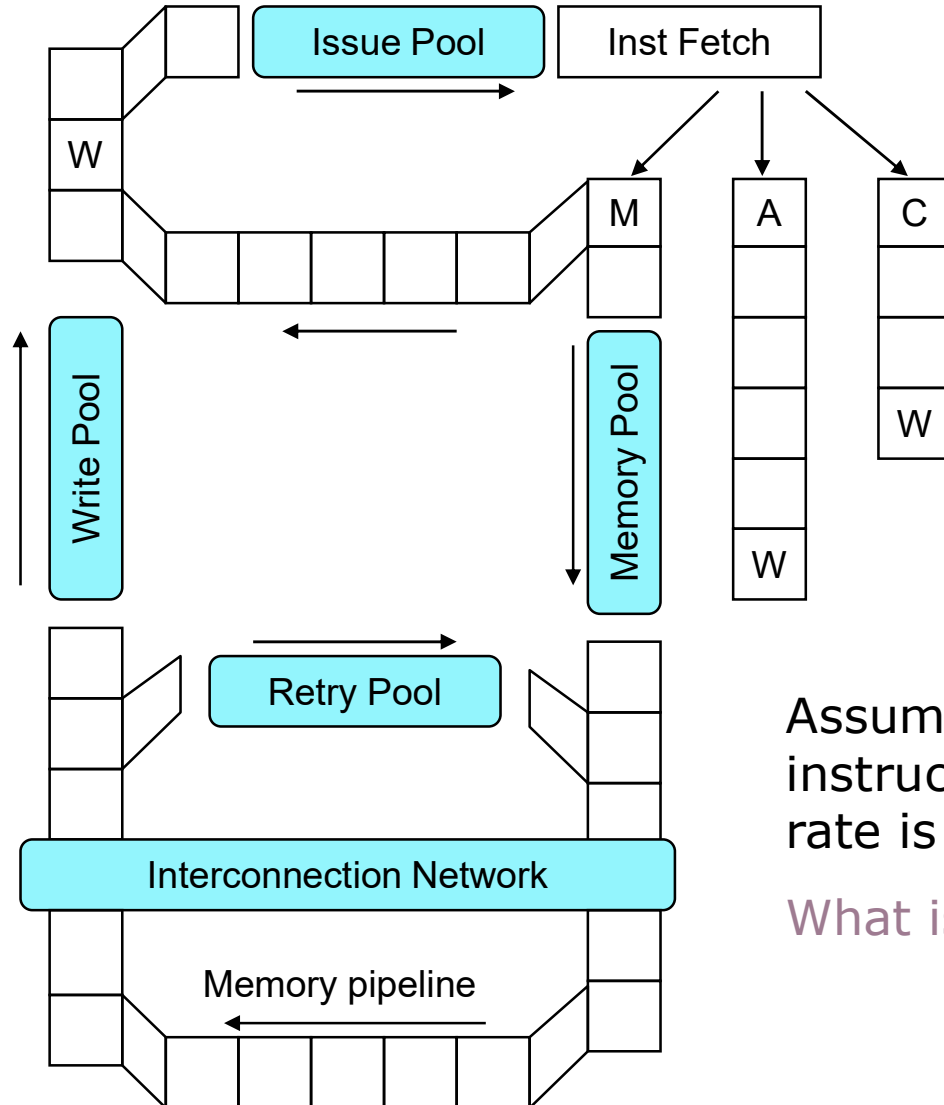
- Up to 256 processors
- Up to 128 active threads per processor
- Processors and memory modules populate a sparse 3D torus interconnection fabric
- Flat, shared main memory
  - No data cache
  - Sustains one main memory access per cycle per processor
- GaAs logic in prototype, 1KW/processor @ 260MHz
  - CMOS version, MTA-2, 50W/processor

# MTA Architecture

---

- Each processor supports 128 active hardware threads
  - $1 \times 128 = 128$  stream status word (SSW) registers,
  - $8 \times 128 = 1024$  branch-target registers,
  - $32 \times 128 = 4096$  general-purpose registers
- Three operations packed into 64-bit instruction (short VLIW)
  - One memory operation,
  - One arithmetic operation, plus
  - One arithmetic or branch operation
- Thread creation and termination instructions
- Explicit 3-bit “lookahead” field in instruction gives number of subsequent instructions (0-7) that are independent of this one
  - Allows fewer threads to fill machine pipeline
  - Used for variable-sized branch delay slots

# MTA Pipeline



- Every cycle, one instruction from one active thread is launched into pipeline
- Instruction pipeline is 21 cycles long
- Memory operations incur  $\sim 150$  cycles of latency

Assuming a single thread issues one instruction every 21 cycles, and clock rate is 260 MHz...

What is single thread performance?

# Multithreading Design Choices

---

Tera MTA designed for supercomputing applications with large data sets and low locality

- No data cache
- Many parallel threads needed to hide large memory latency

Other applications are more cache friendly

- Few pipeline bubbles when cache getting hits
- Just add a few threads to hide occasional cache miss latencies
- Swap threads on cache misses -> Coarse-grained Multithreading

# Multithreading Design Choices

---

- Fine-grained multithreading
  - Context switch among threads every cycle
- Coarse-grained multithreading
  - Context switch among threads every few cycles, e.g., on:
    - Function unit data hazard,
    - L1 miss,
    - L2 miss...
- Why choose one style over another?
- Choice depends on
  - Context-switch overhead
  - Number of threads supported (due to per-thread state)
  - Expected application-level parallelism...

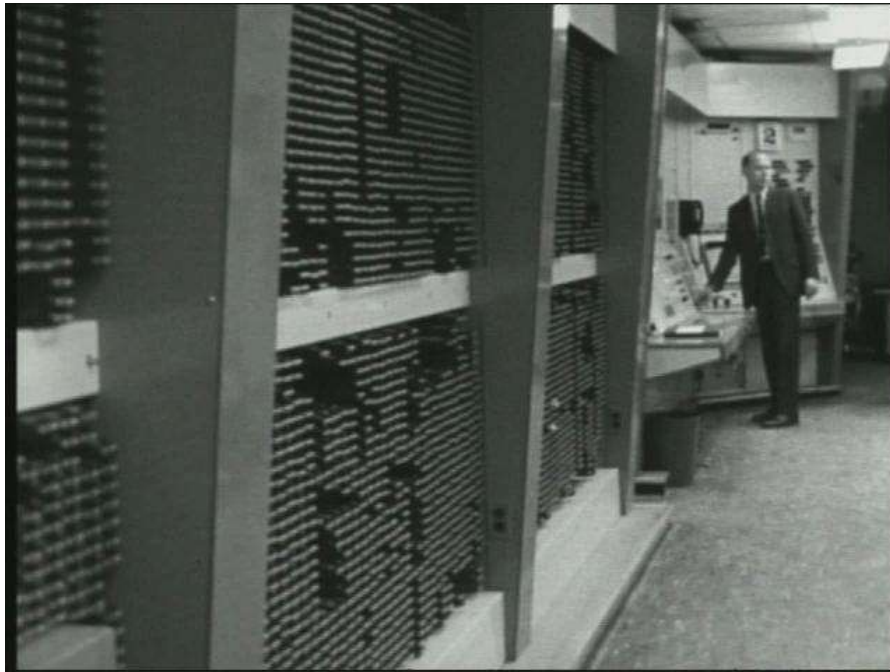
# TX-2: Multi-sequence computer

(Wes Clark, Lincoln Labs, 1956)

---

32 Instruction sequences (threads) with

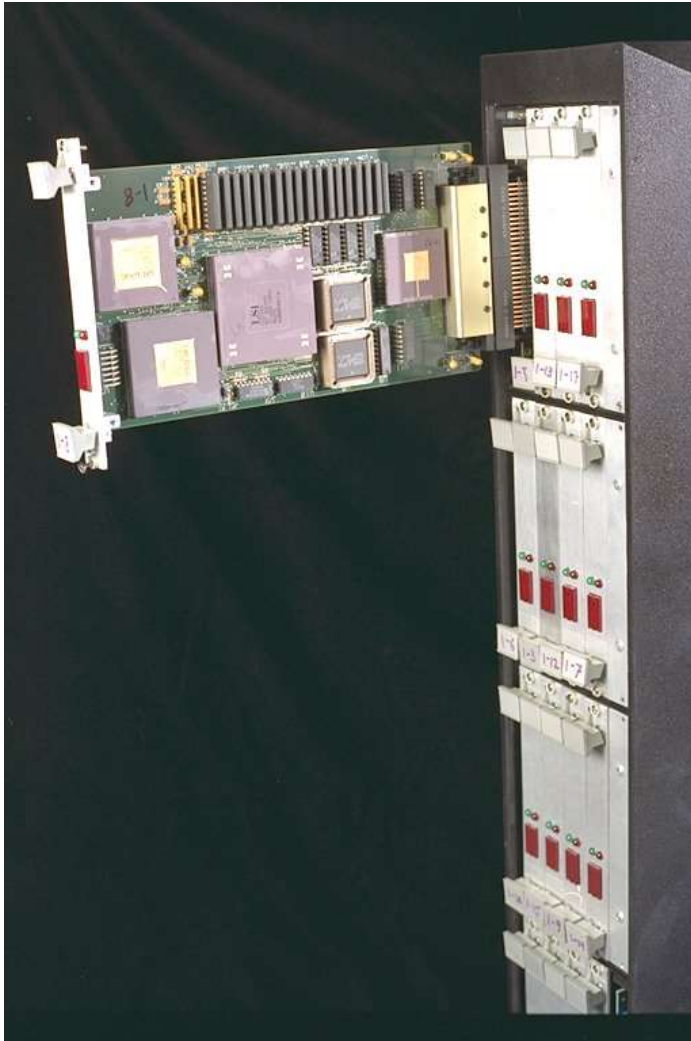
- a fixed priority order among the threads, and
- executes many instructions in a thread - switches mediated by:
  - Instruction “break”/“dismiss” bits
  - Attention request from I/O



- Start-Over
- In-out alarms
- Arithmetic alarms (overflows, etc.)
- Magnetic tape units (multiple)
- High-speed printer
- Analog-to-digital converter
- Paper tape readers (multiple)
- Light pen
- Display (multiple)
- Memory Test Computer
- TX-O
- Digital-to-analog converter
- Paper tape punch
- Flexowriters (multiple)
- \*Main sequences (three)

# MIT Alewife (1990)

---



- Modified SPARC chips
  - Register windows hold different thread contexts
- Up to four threads per node
- Thread switch on local cache miss



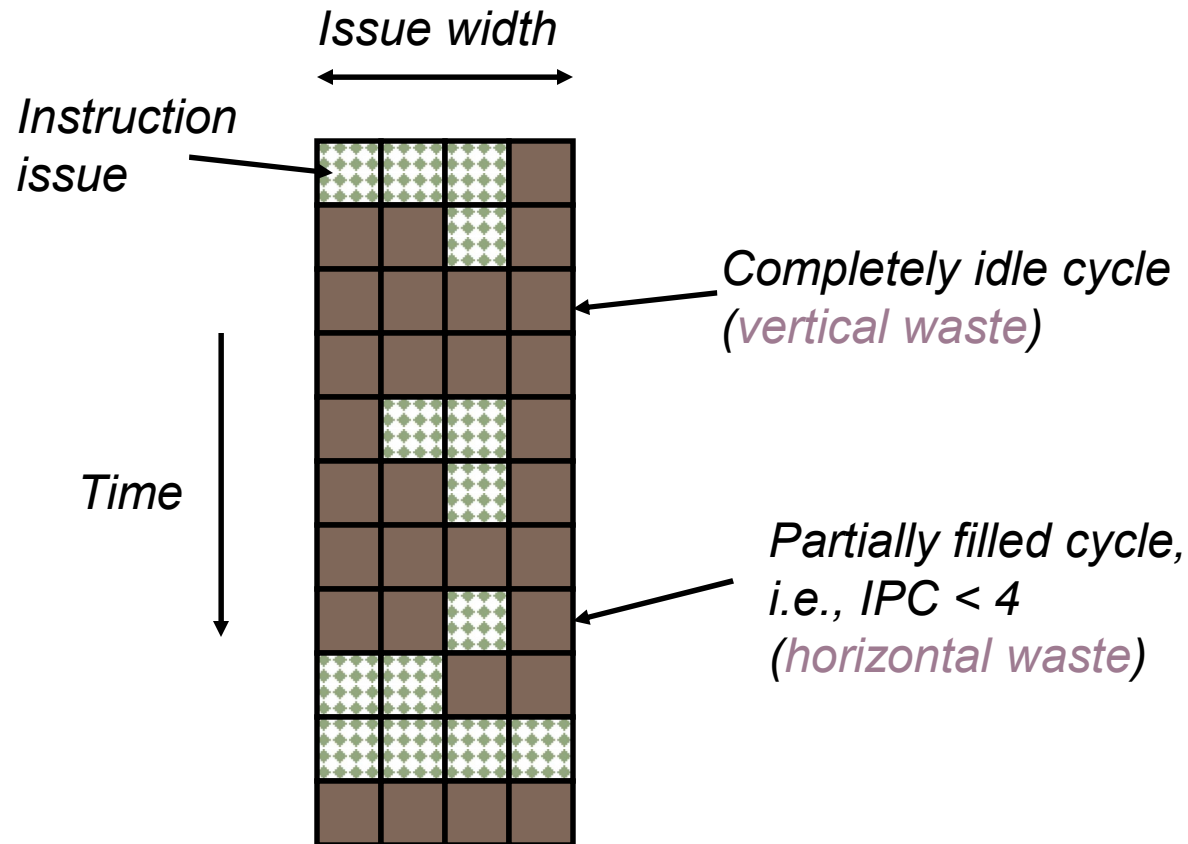
# IBM PowerPC RS64-IV (2000)

---

- Commercial coarse-grain multithreading CPU
- Based on PowerPC with quad-issue in-order five-stage pipeline
- Each physical CPU supports two virtual CPUs
- On L2 cache miss, pipeline is flushed and execution switches to second thread
  - Short pipeline minimizes flush penalty (4 cycles), small compared to memory access latency
  - Flush pipeline to simplify exception handling

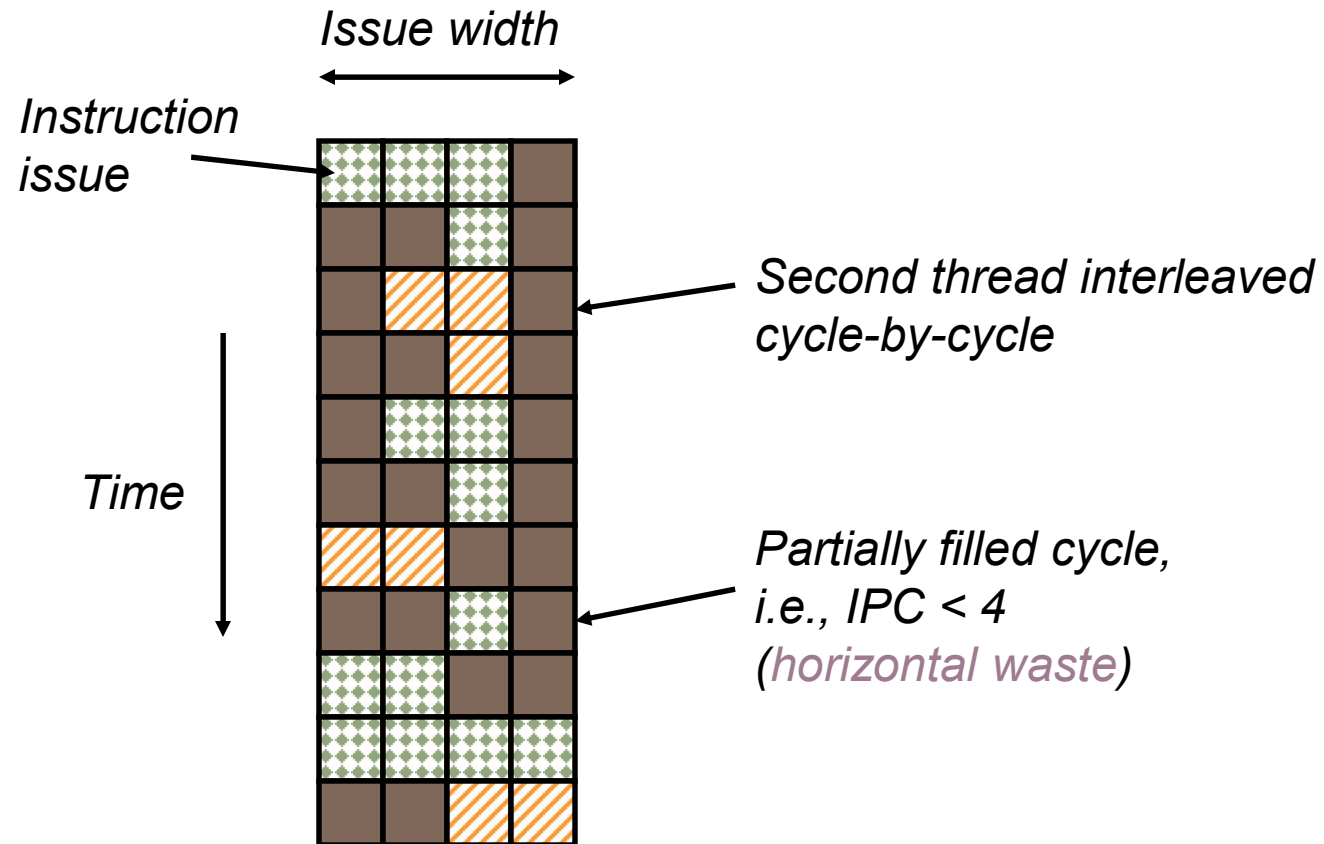
# Superscalar Machine Efficiency

---



- *Why horizontal waste?*
- *Why vertical waste?*

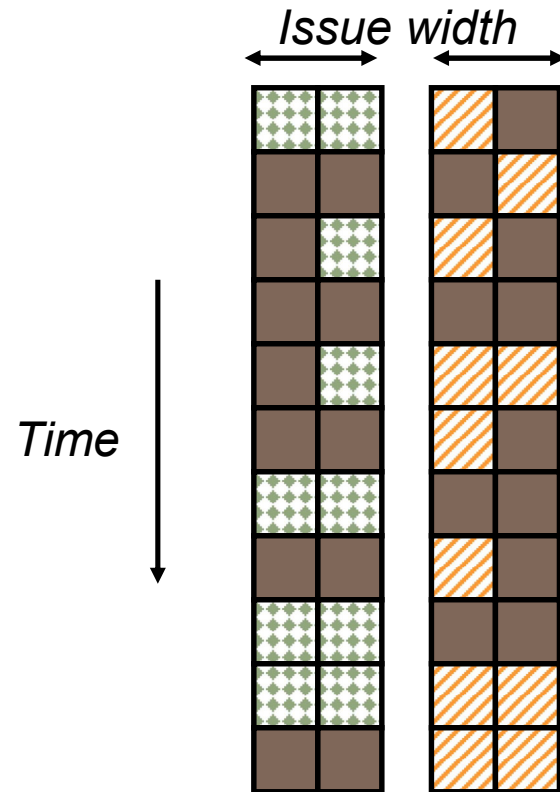
# Vertical Multithreading



- What is the effect of cycle-by-cycle interleaving?

# Chip Multiprocessing

---

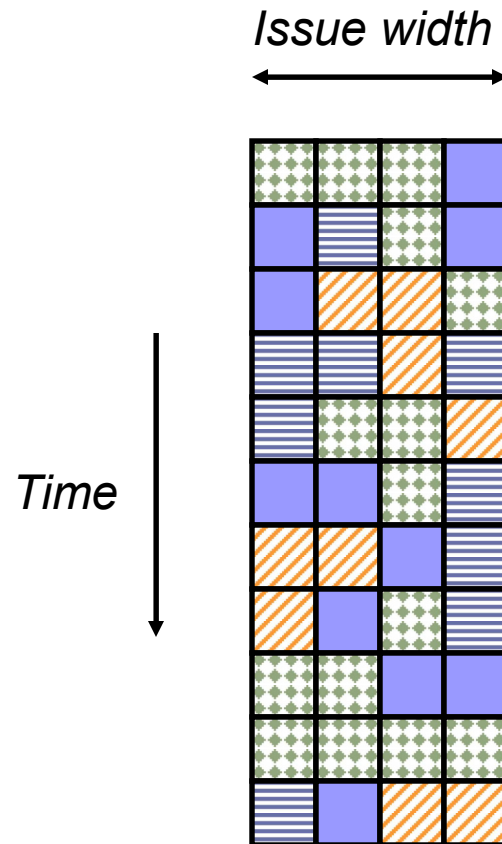


- What is the effect of splitting into multiple processors?

# Ideal Superscalar Multithreading

[Tullsen, Eggers, Levy, UW, 1995]

---



- Interleave multiple threads to multiple issue slots with no restrictions

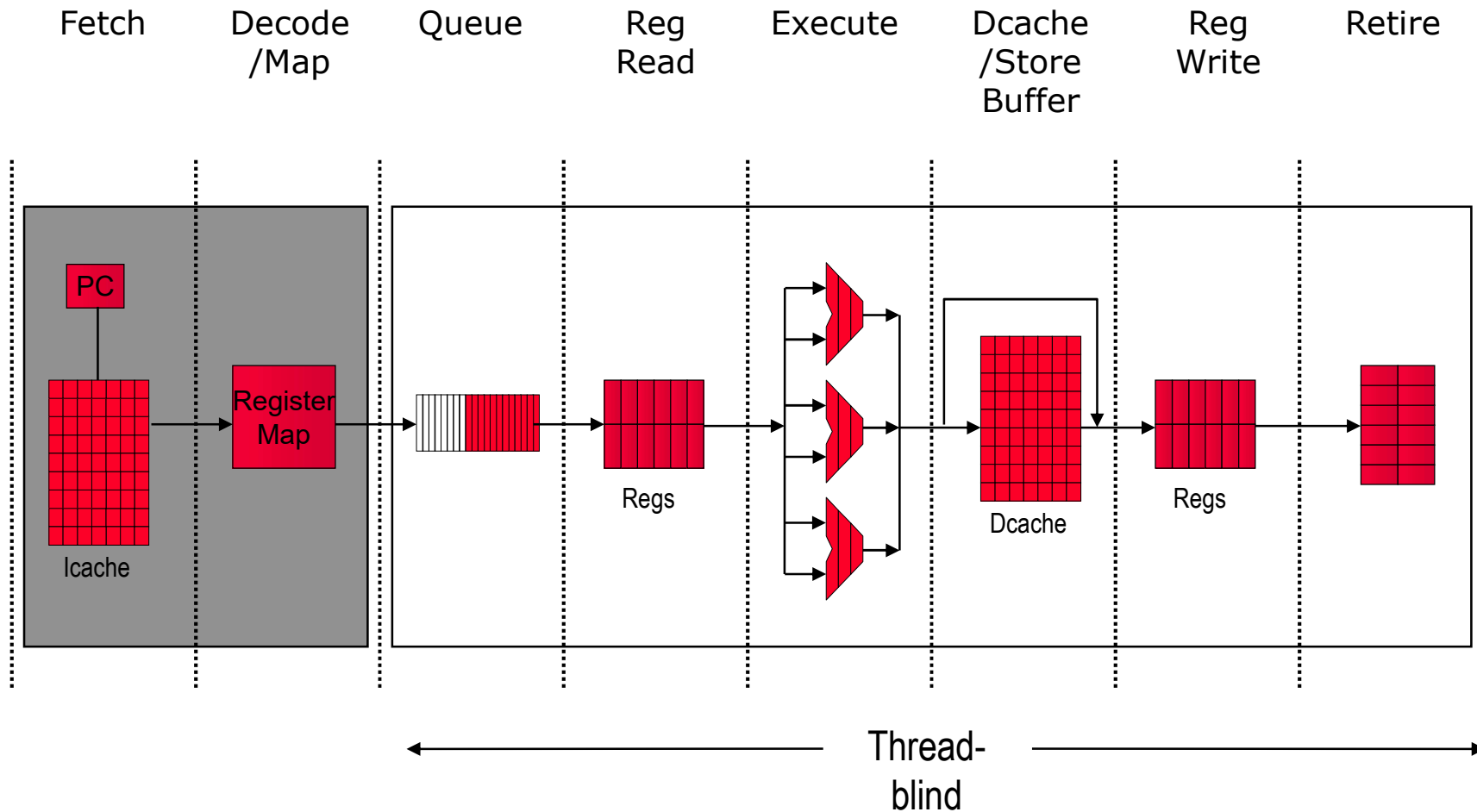
# O-o-O Simultaneous Multithreading

[Tullsen, Eggers, Emer, Levy, Stamm, Lo, DEC/UW, 1996]

---

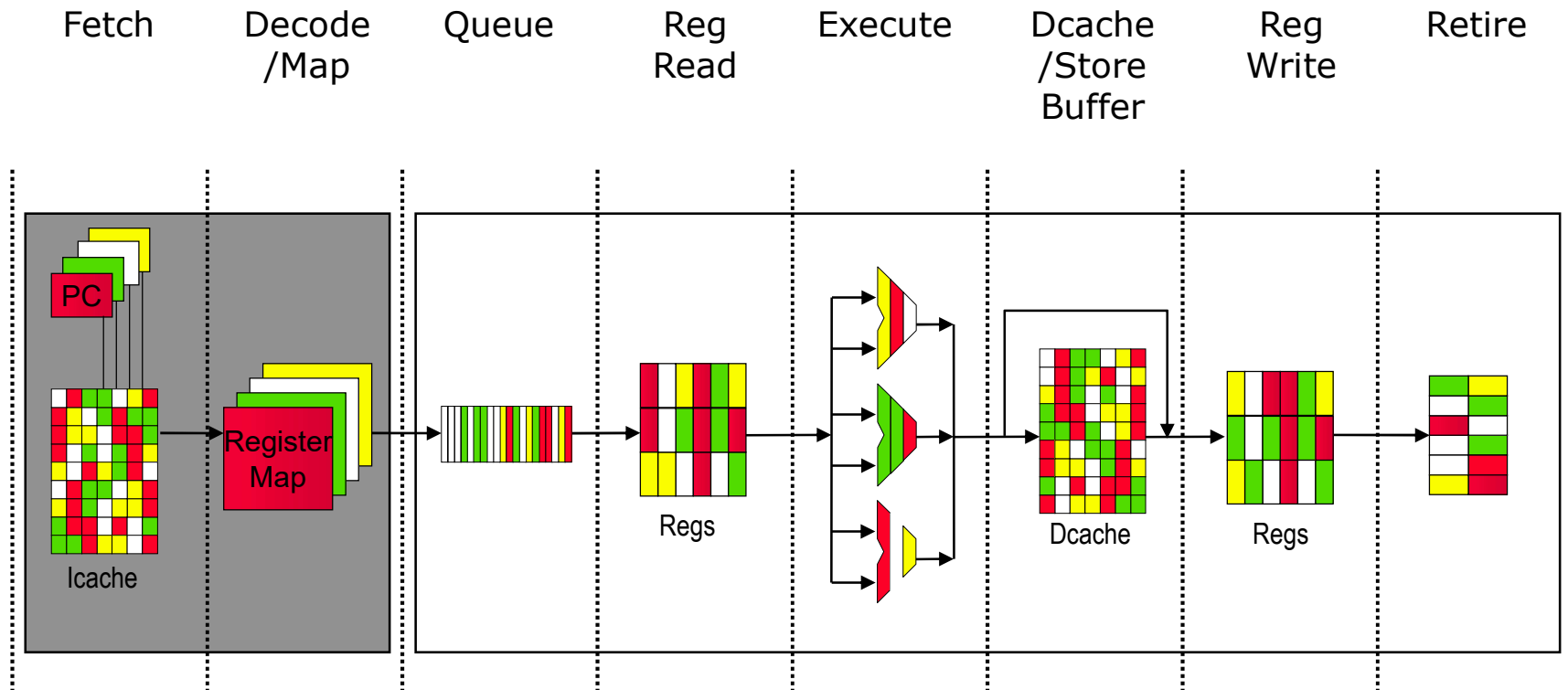
- Add multiple contexts and fetch support and allow instructions fetched from different threads to issue simultaneously
- Utilize wide out-of-order superscalar processor issue queue to find instructions to issue from multiple threads
- Key insight: OOO instruction window already has most of the circuitry required to schedule from multiple threads
- Any single thread can utilize whole machine

# Basic Out-of-order Pipeline



[ EV8 – Microprocessor Forum, Oct 1999]

# SMT Pipeline



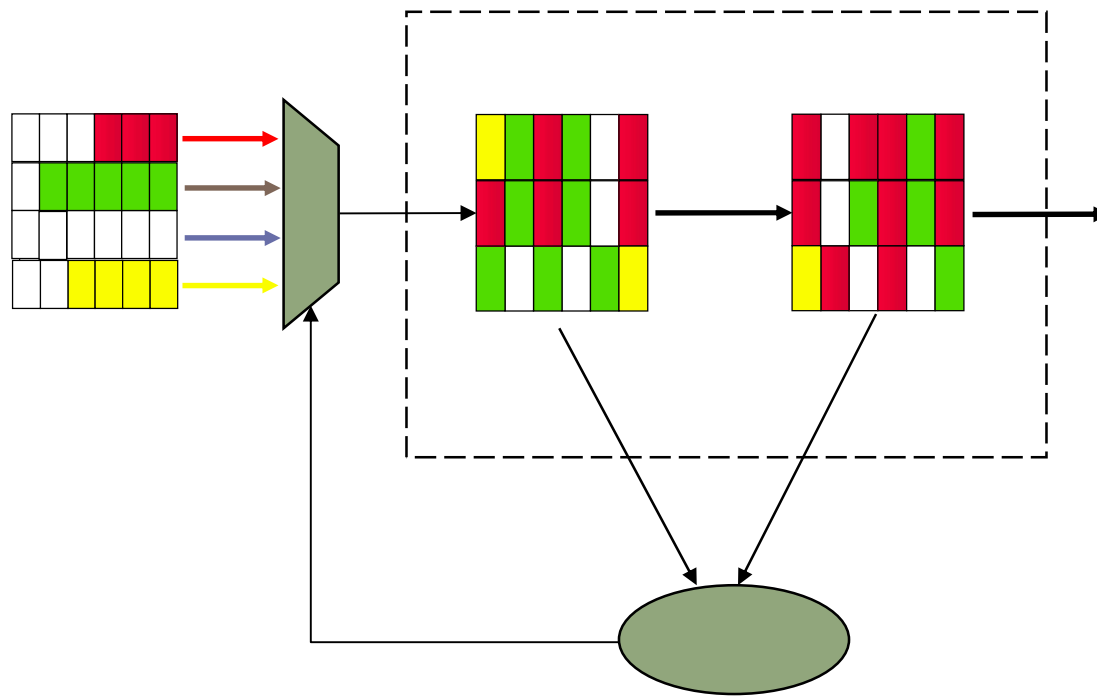
[ EV8 – Microprocessor Forum, Oct 1999]



# Icount Choosing Policy

---

Fetch from thread with the least instructions in flight.



*Why does this enhance throughput?*

# Why Does Icount Make Sense?

---

$$T = \frac{N}{L}$$

Assuming latency (L) is unchanged with the addition of threading.  
For each thread i with original throughput  $T_i$  (and 4 threads):

$$T_i / 4 = \frac{N / 4}{L}$$

# SMT Fetch Policies (Locks)

---

- Problem:  
Spin looping thread consumes resources
- Solution:  
Provide quiescing operation that allows a thread to sleep until a memory location changes

loop:

```
ARM r1, 0(r2)
BEQ r1, got_it
QUIESCE
BR loop
```

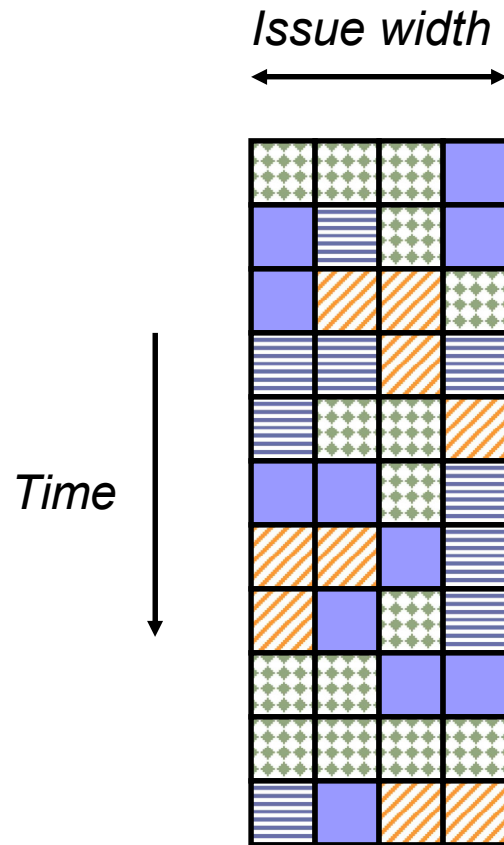
got\_it:

Load and start  
watching 0(r2)

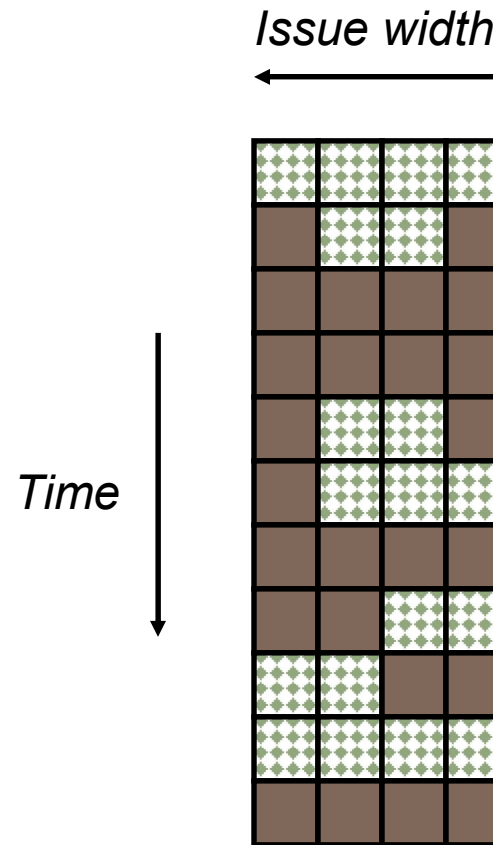
Inhibit scheduling of  
thread until activity  
observed on 0(r2)

# Adaptation to parallelism type

For regions with high thread level parallelism (TLP) entire machine width is shared by all threads



For regions with low thread level parallelism (TLP) entire machine width is available for instruction level parallelism (ILP)



# Pentium-4 Hyperthreading (2002)

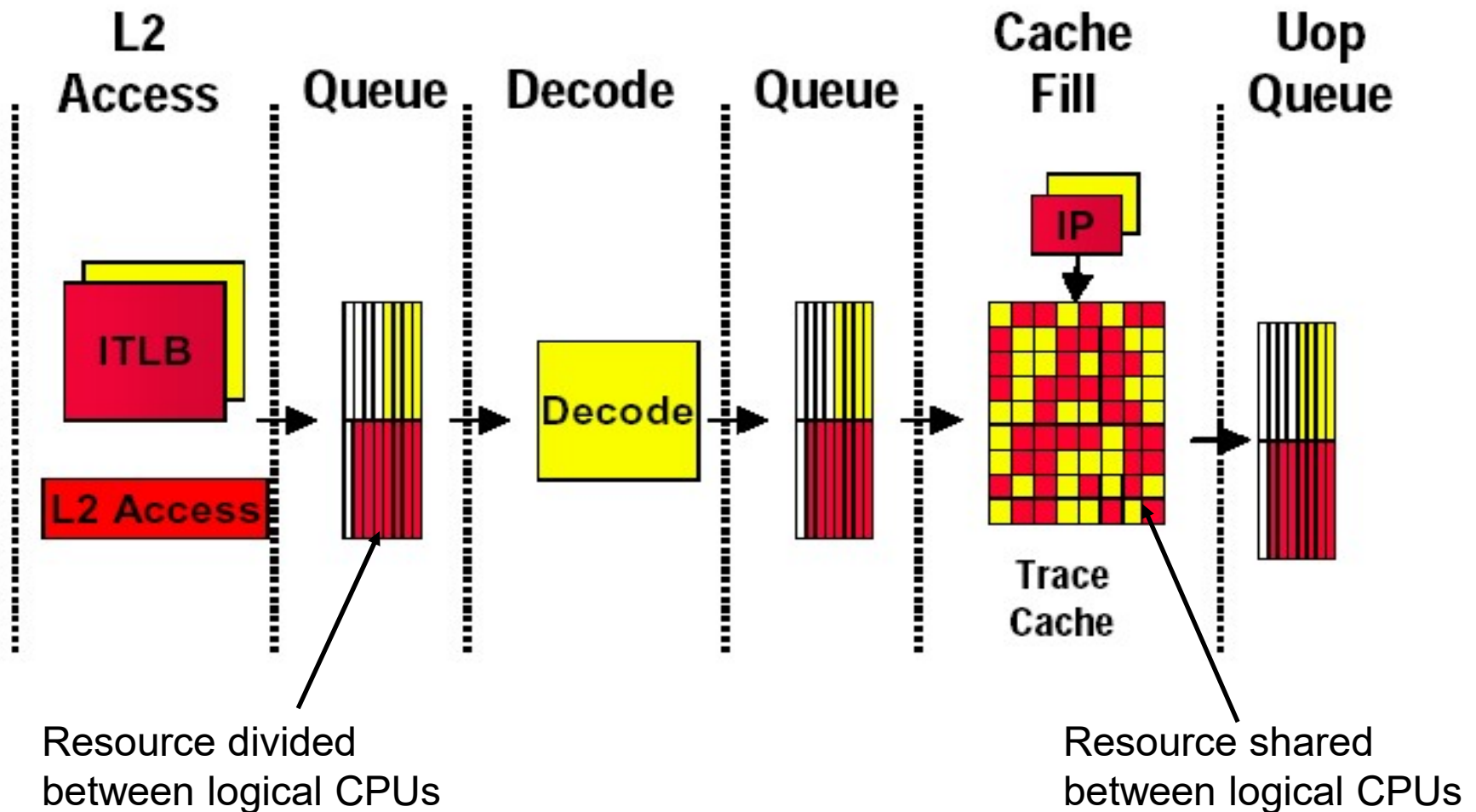
---

- First commercial SMT design (2-way SMT)
  - Hyperthreading == SMT
- Logical processors share nearly all resources of the physical processor
  - Caches, execution units, branch predictors
- Die area overhead of hyperthreading  $\sim 5\%$
- When one logical processor is stalled, the other can make progress
  - No logical processor can use all entries in queues when two threads are active
- Processor running only one active software thread runs at approximately same speed with or without hyperthreading

# Pentium-4 Hyperthreading

## *Front End*

---



[ Intel Technology Journal, Q1 2002 ]

# Pentium-4 Branch Predictor

---

- Separate return address stacks per thread

*Why?*

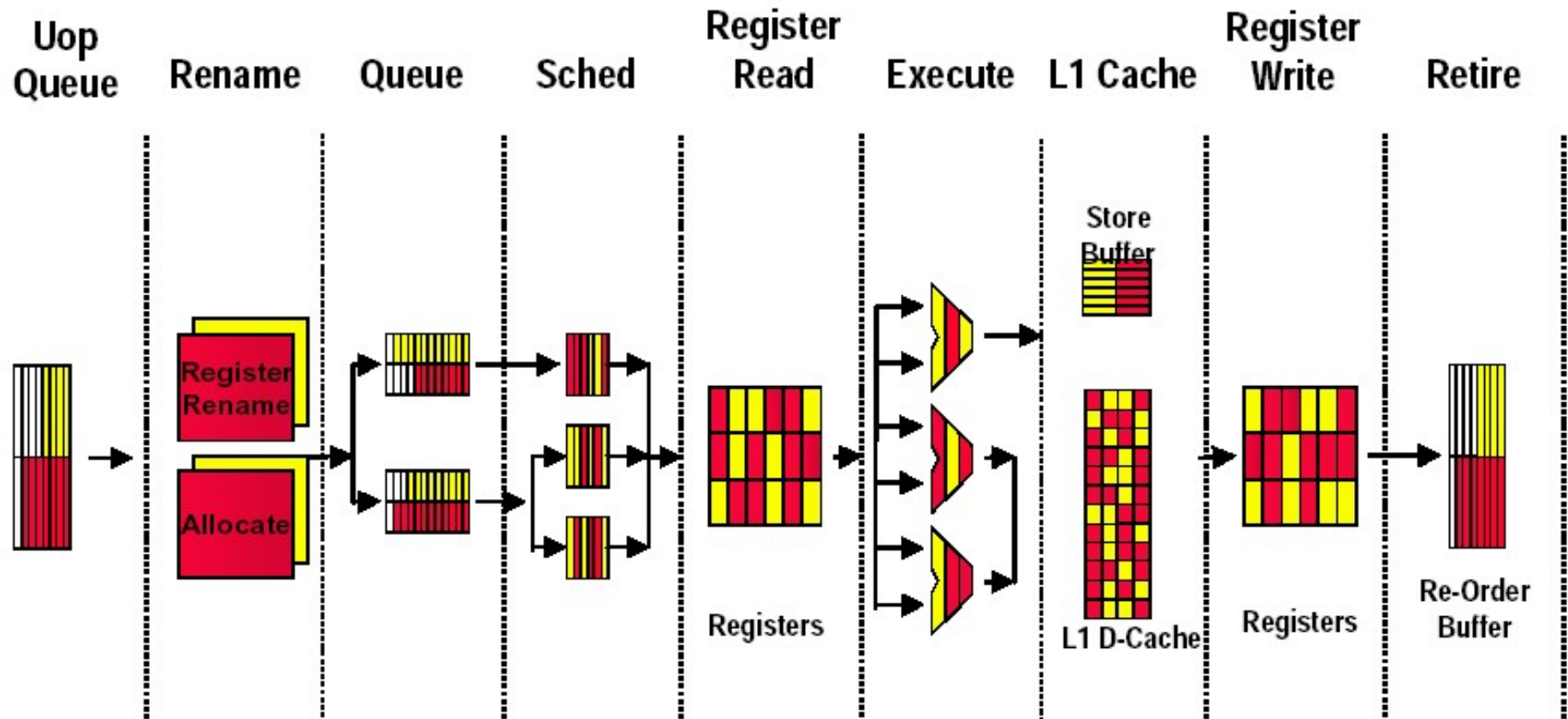
- Separate first-level global branch history table

*Why?*

- Shared second-level branch history table, tagged with logical processor IDs

# Pentium-4 Hyperthreading *Execution Pipeline*

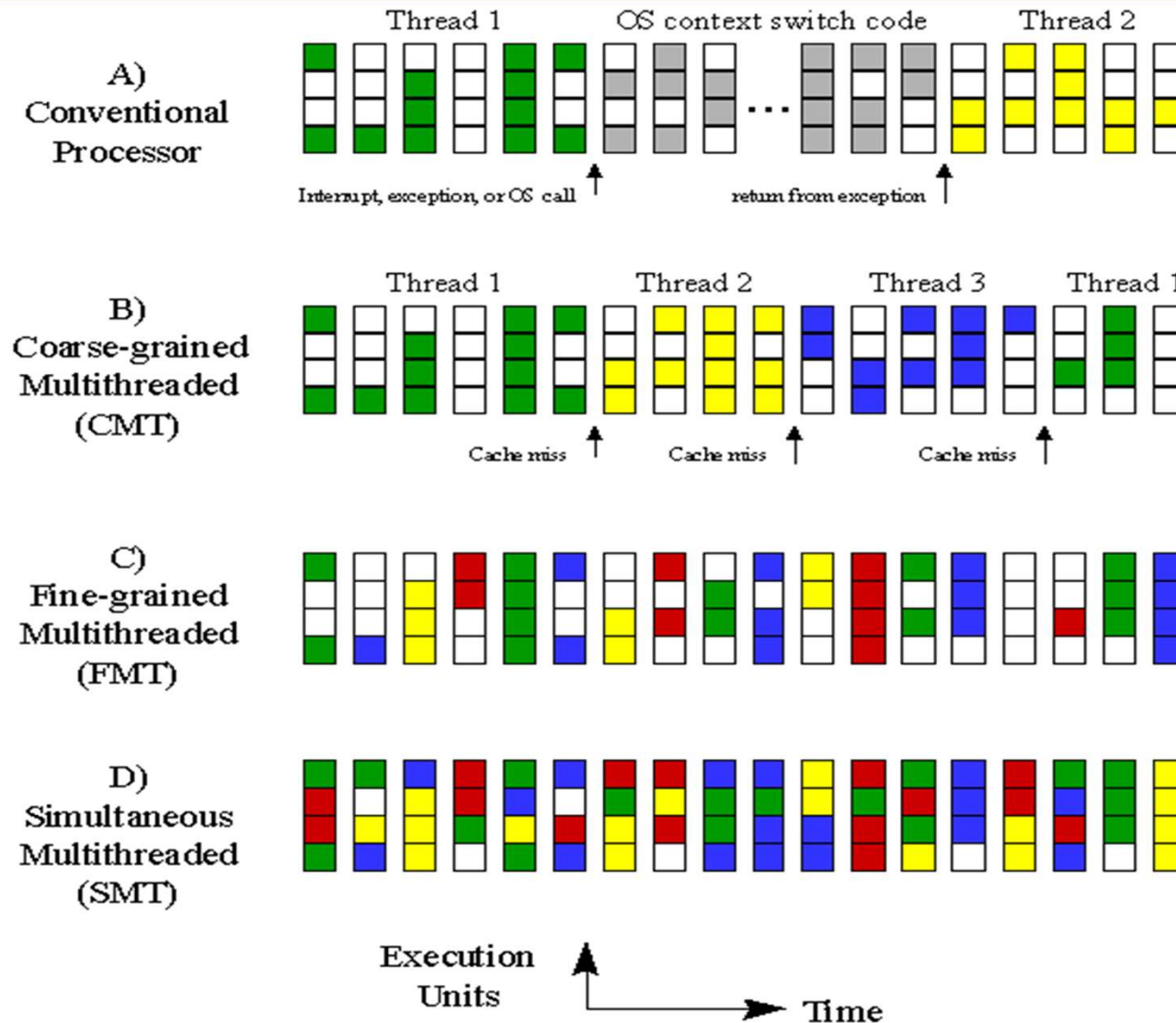
---



[ Intel Technology Journal, Q1 2002 ]



# Summary: Multithreading Styles



*Thank you!*