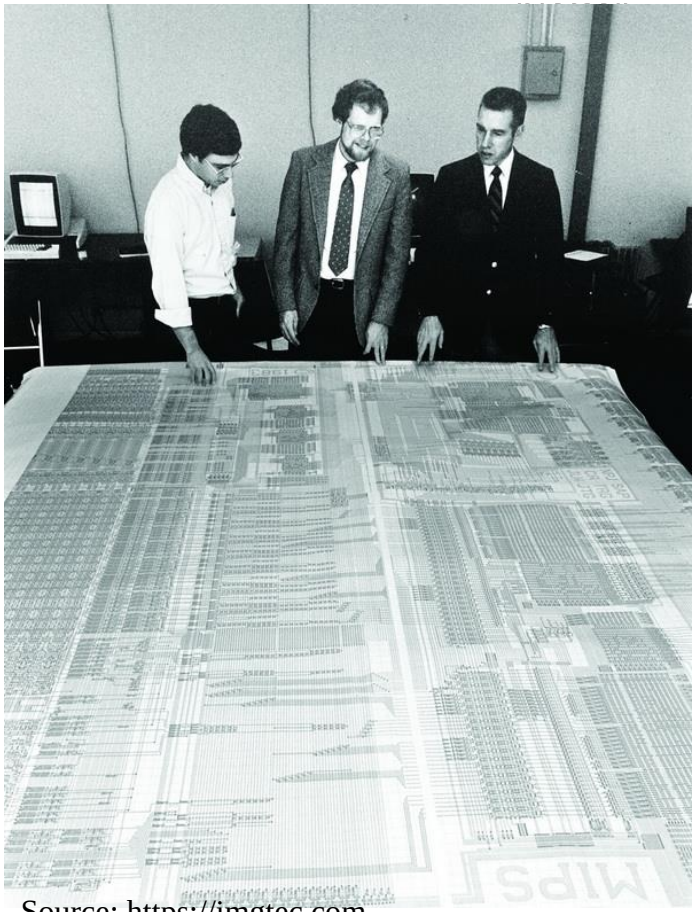
Day 1: Introduction
Course: Superscalar Architecture

12th International ACACES Summer School
10-16 July 2016, Fiuggi, Italy

Lecture notes based in part on slides created by John Shen and Ilhyun Kim

CPU, circa 1986

Stage	Phase	Function performed
IF	φ_1	Translate virtual instr. addr. using TLB
	φ_2	Access I-cache
RD	φ_1	Return instruction from I-cache, check tags & parity
	φ_2	Read RF; if branch, generate target
ALU	φ_1	Start ALU op; if branch, check condition
	φ_2	Finish ALU op; if ld/st, translate addr
MEM	φ_1	Access D-cache
	φ_2	Return data from D-cache, check tags & parity
WB	φ_1	Write RF
	φ_2	



Source: https://imgtec.com

- MIPS R2000, ~“most elegant pipeline ever devised” J. Larus
- Enablers: RISC ISA, pipelining, on-chip cache memory

Iron Law

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) (CPI) (cycle time)

Architecture --> Implementation --> Realization

Compiler Designer **Processor Designer** **Chip Designer**

Limitations of Scalar Pipelines

- Scalar upper bound on throughput
  - IPC <= 1 or CPI >= 1
- Rigid pipeline stall policy
  - One stalled instruction stalls entire pipeline
- Limited hardware parallelism
  - Only temporal (across pipeline stages)

Superscalar Proposal

- Fetch/execute multiple instructions per cycle
- Decouple stages so stalls don't propagate
- Exploit instruction-level parallelism (ILP)

Limits on Instruction Level Parallelism (ILP)

Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86 (Flynn's bottleneck)
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7 (Jouppi disagreed)
Kuck et al. [1972]	8
Riseman and Foster [1972]	51 (no control dependences)
Nicolau and Fisher [1984]	90 (Fisher's optimism)

High-IPC Processor Evolution

Desktop/Workstation Market

Scalar RISC Pipeline	2-4 Issue In-order	Limited Out-of-Order	Large ROB Out-of-Order
1980s: MIPS SPARC Intel 486	Early 1990s: IBM RIOS-I Intel Pentium	Mid 1990s: PowerPC 604 Intel P6	2000s: DEC Alpha 21264 IBM Power4/5 AMD K8

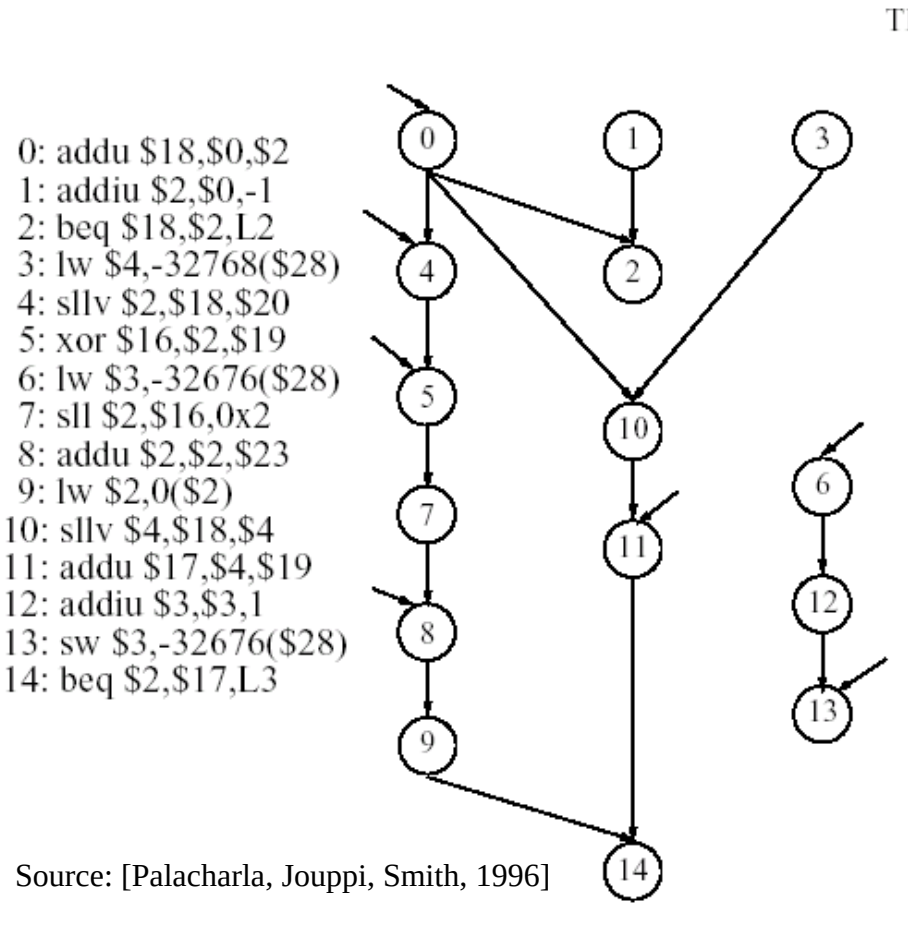
1985 – 2005: 20 years, 100x frequency

Mobile Market

Scalar RISC Pipeline	2-4 Issue In-order	Limited Out-of-Order	Large ROB Out-of-Order
2002: ARM11	2005: Cortex A8	2009: Cortex A9	2011: Cortex A15

2002 – 2011: 10 years, 10x frequency

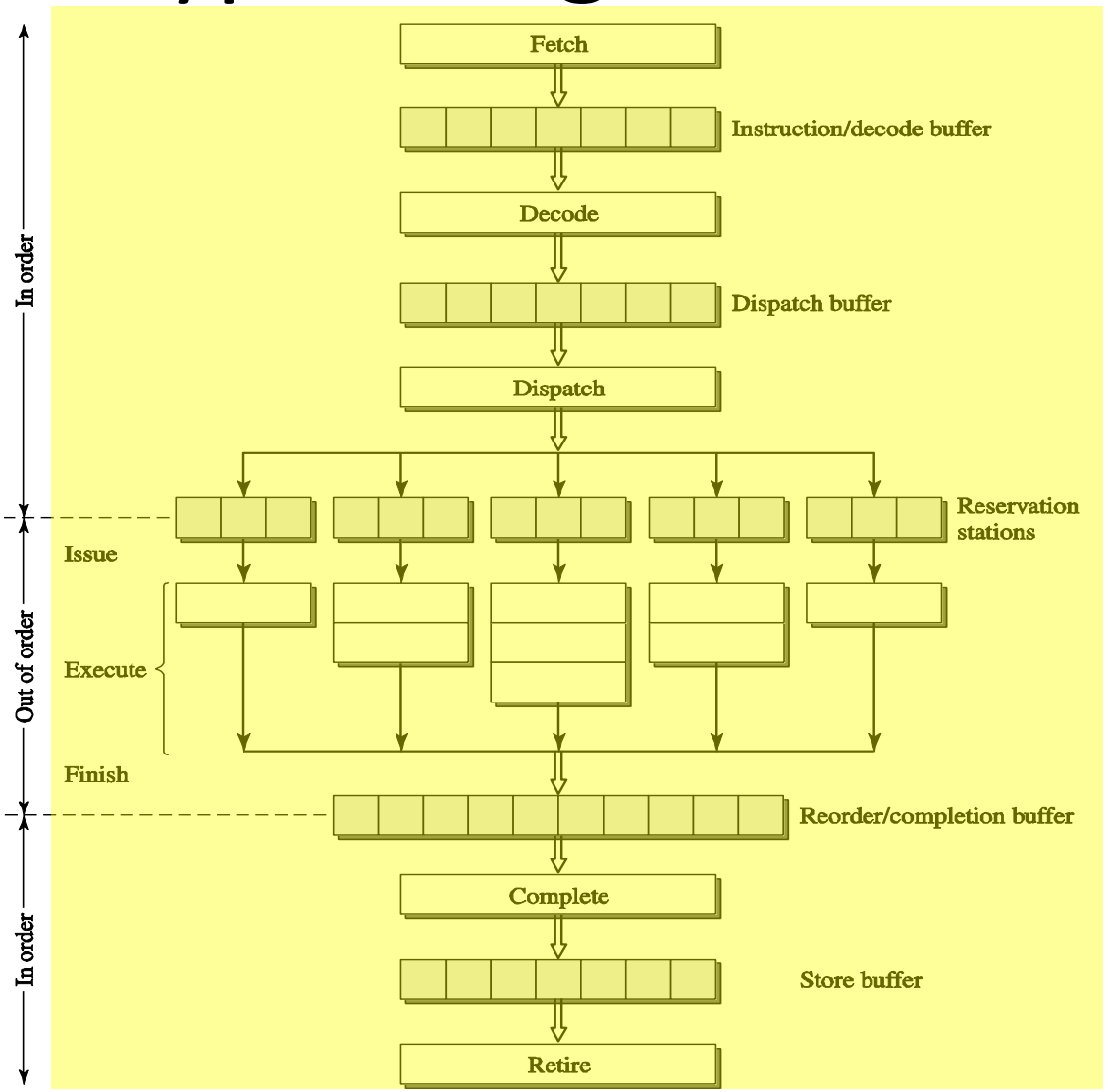
What Does a High-IPC CPU Do?



Source: [Palacharla, Jouppi, Smith, 1996]

1. Fetch and decode
2. Construct data dependence graph (DDG)
3. Evaluate DDG
4. Commit changes to program state

A Typical High-IPC Processor



1. Fetch and Decode

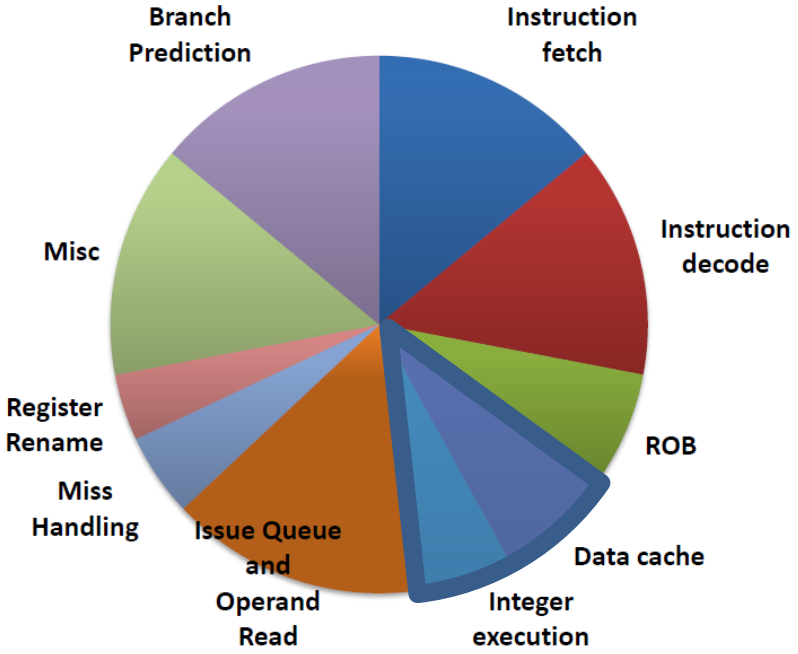
2. Construct DDG

3. Evaluate DDG

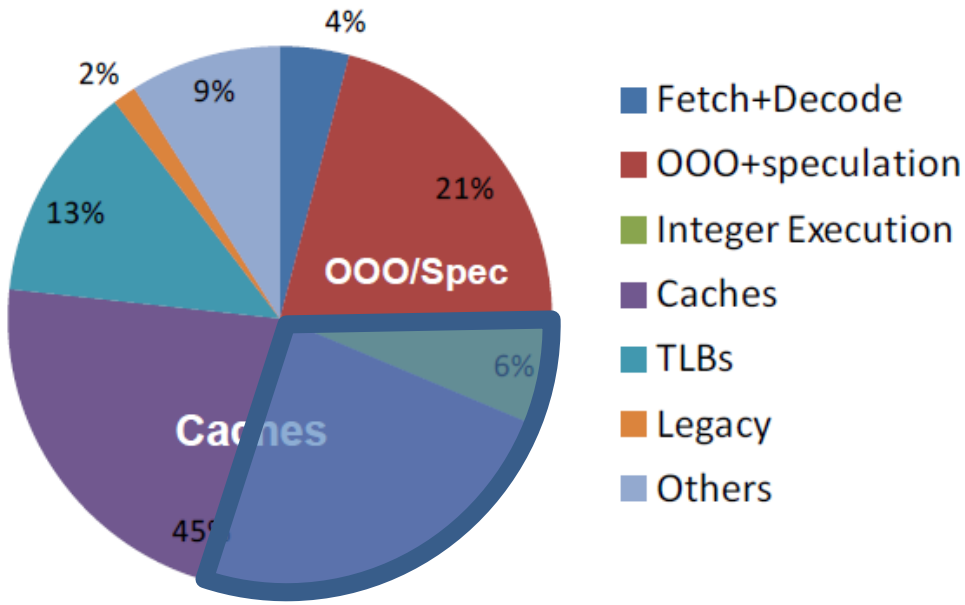
4. Commit results

Power Consumption

ARM Cortex A15 [Source: NVIDIA]



Core i7 [Source: Intel]

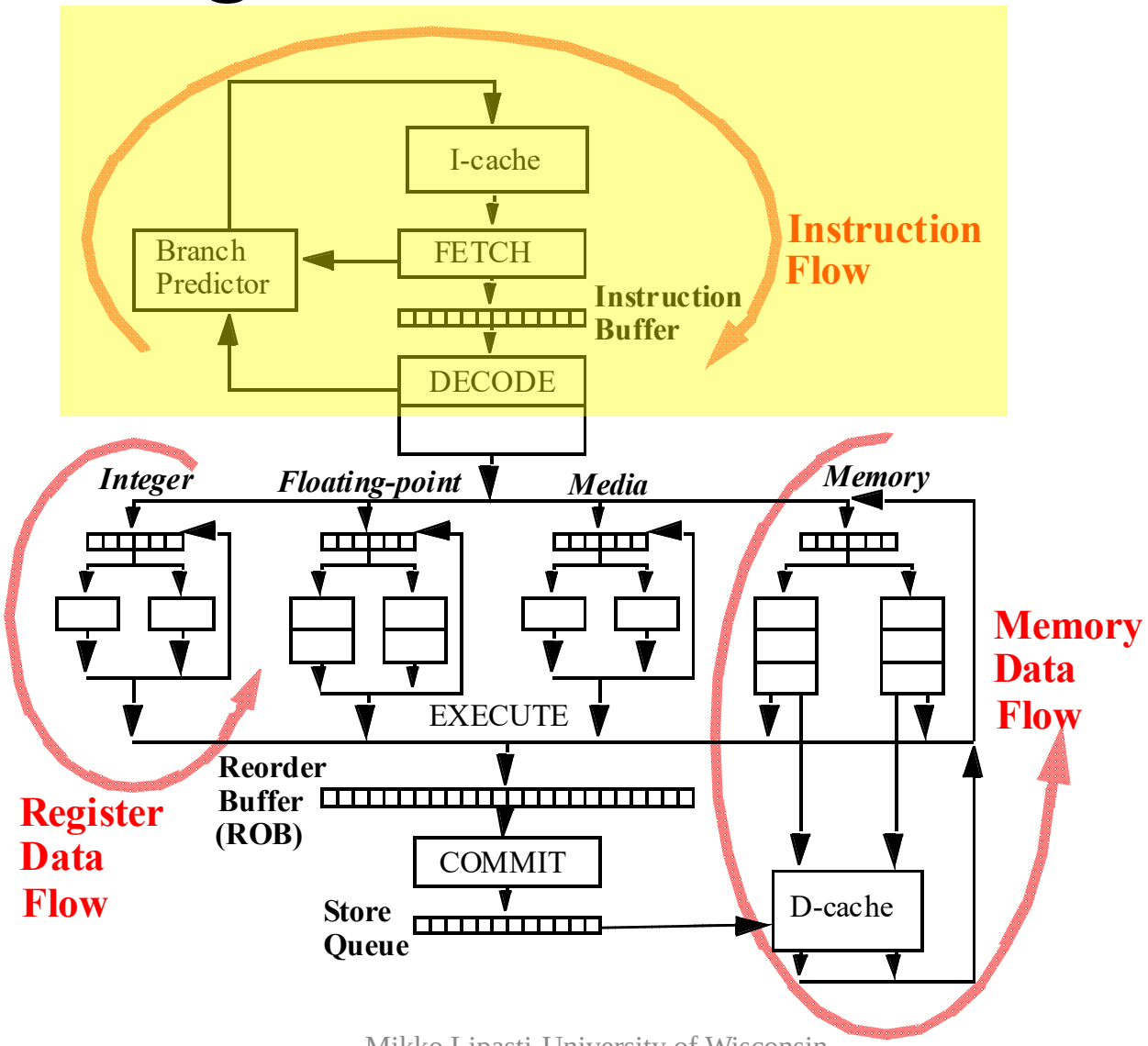


- Actual computation overwhelmed by overhead of aggressive execution pipeline

Lecture Outline

- Evolution of High-IPC Processors
- Main challenges
  - Instruction Flow
  - Register Data Flow
  - Memory Data Flow

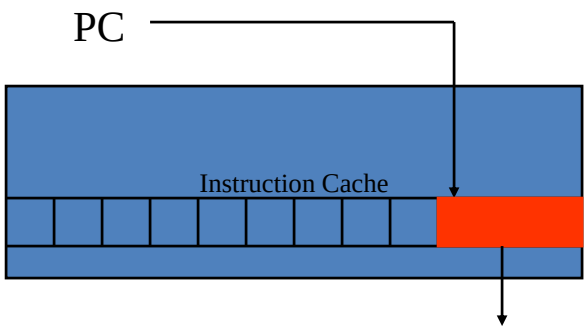
High-IPC Processor



Instruction Flow

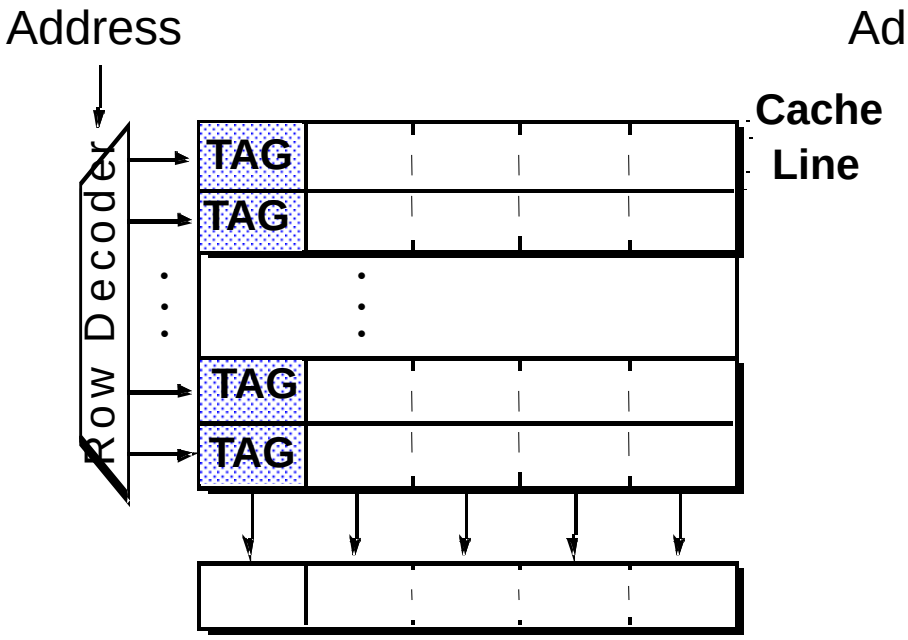
Objective: Fetch multiple instructions per cycle

- Challenges:
  - Branches: unpredictable
  - Branch targets misaligned
  - Instruction cache misses
- Solutions
  - Prediction and speculation
  - High-bandwidth fetch logic
  - Nonblocking cache and prefetching

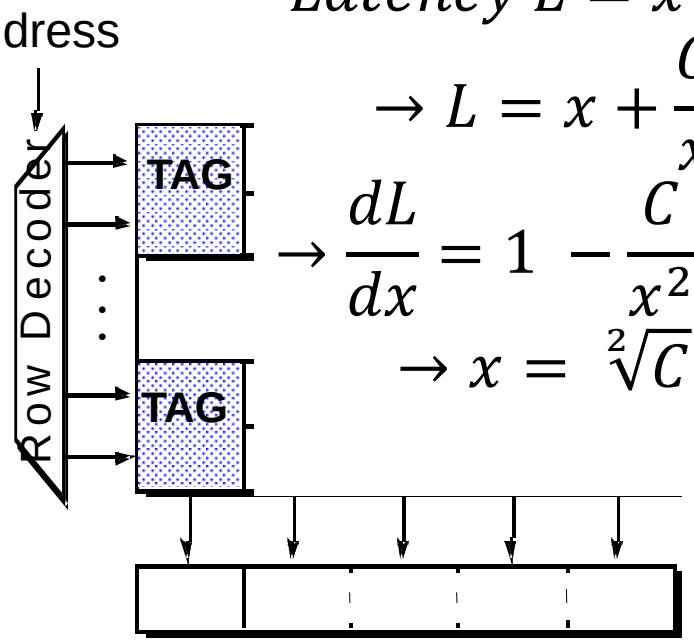


only 3 instructions fetched

I-Cache Organization



1 cache line = 1 physical row



1 cache line = 2 physical rows

$$Capacity\ \mathrm{C} = \mathrm{x} \times \mathrm{y}$$

$$Latency\ L = x + y$$

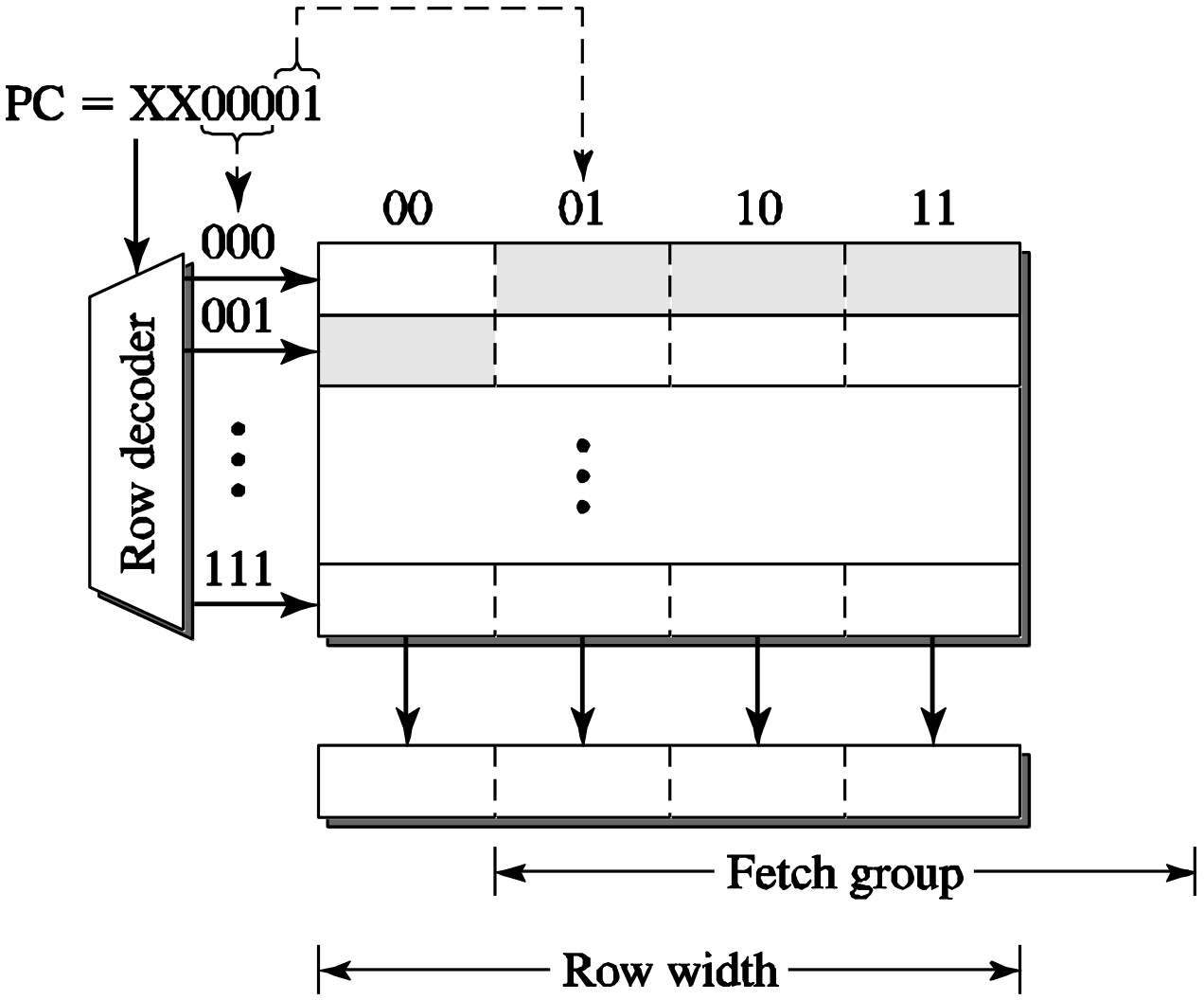
$$\rightarrow L = x + \frac{C}{x}$$

$$\rightarrow \frac{dL}{dx} = 1 - \frac{C}{x^2} = 0$$

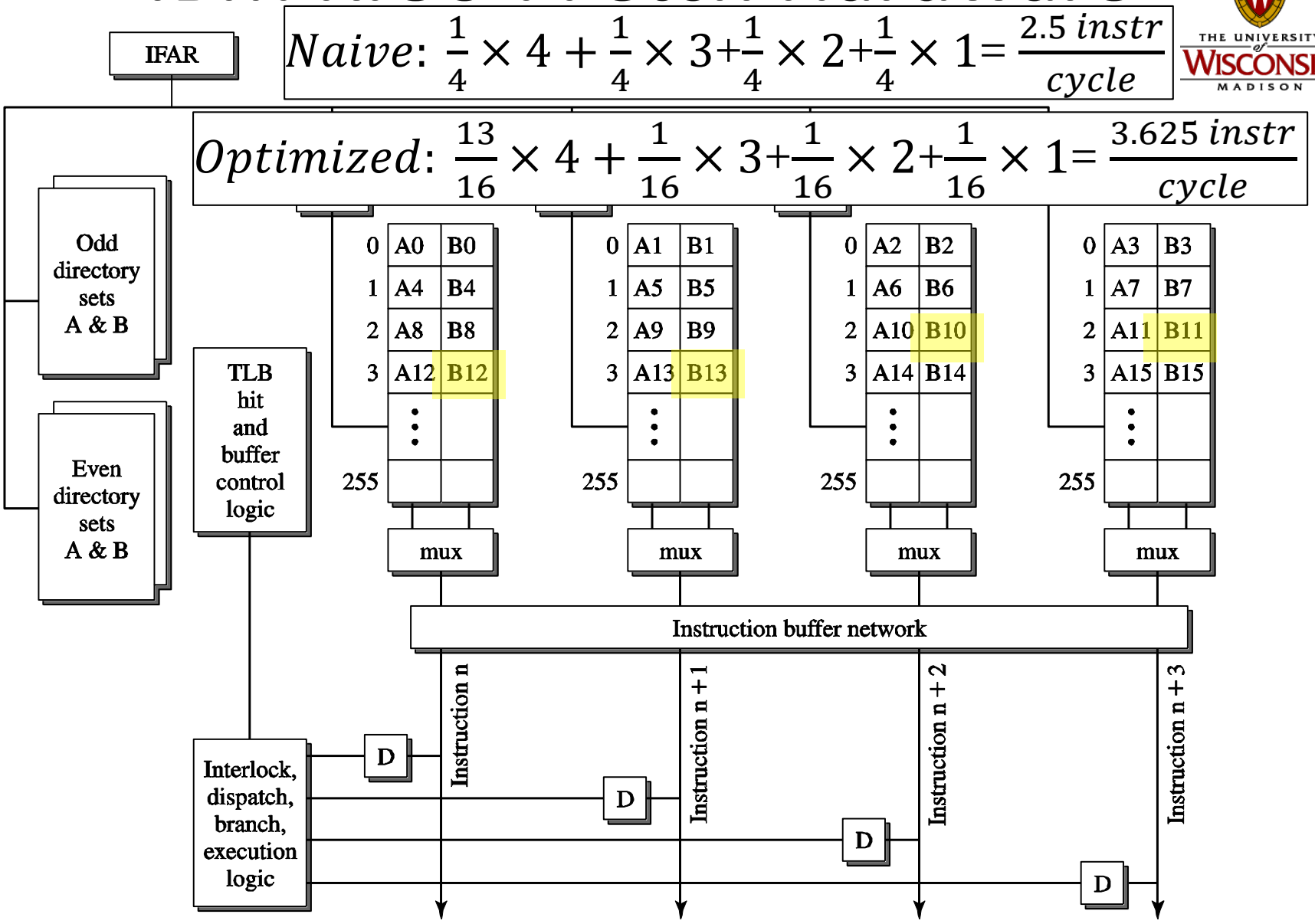
$$\rightarrow x = \sqrt[2]{C}$$

SRAM arrays need to be square to minimize delay

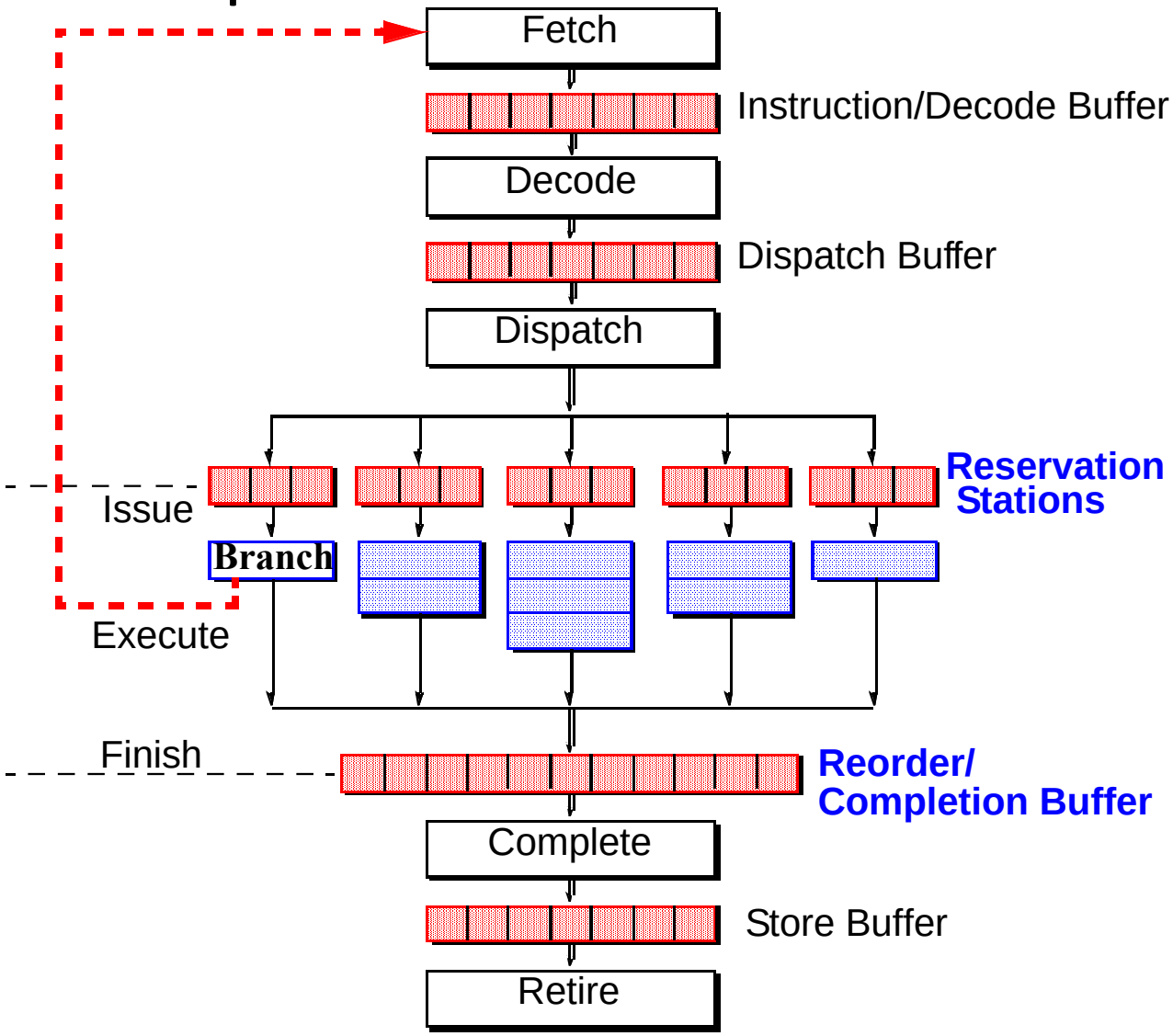
Fetch Alignment



IBM RIOS-I Fetch Hardware



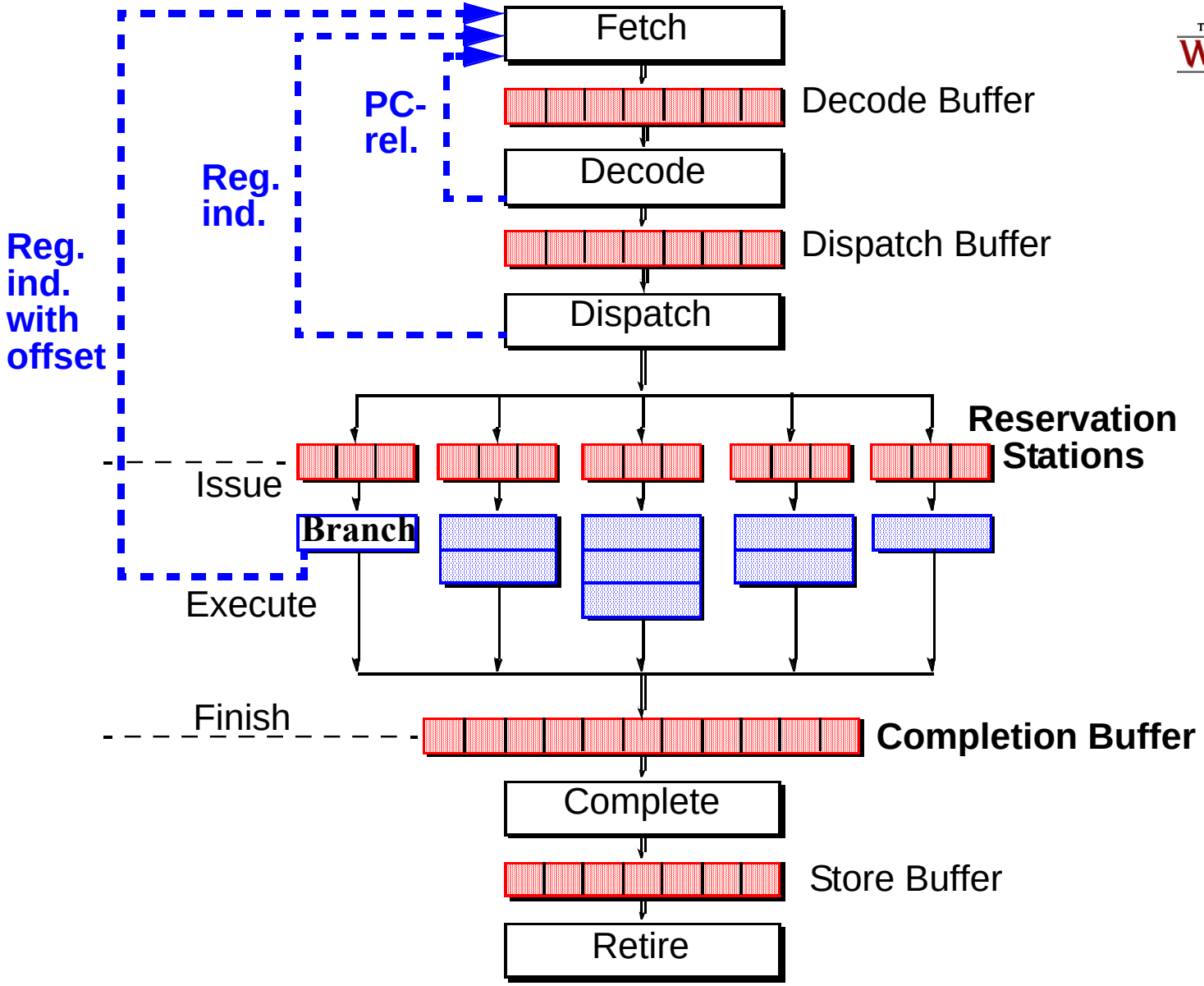
Disruption of Instruction Flow



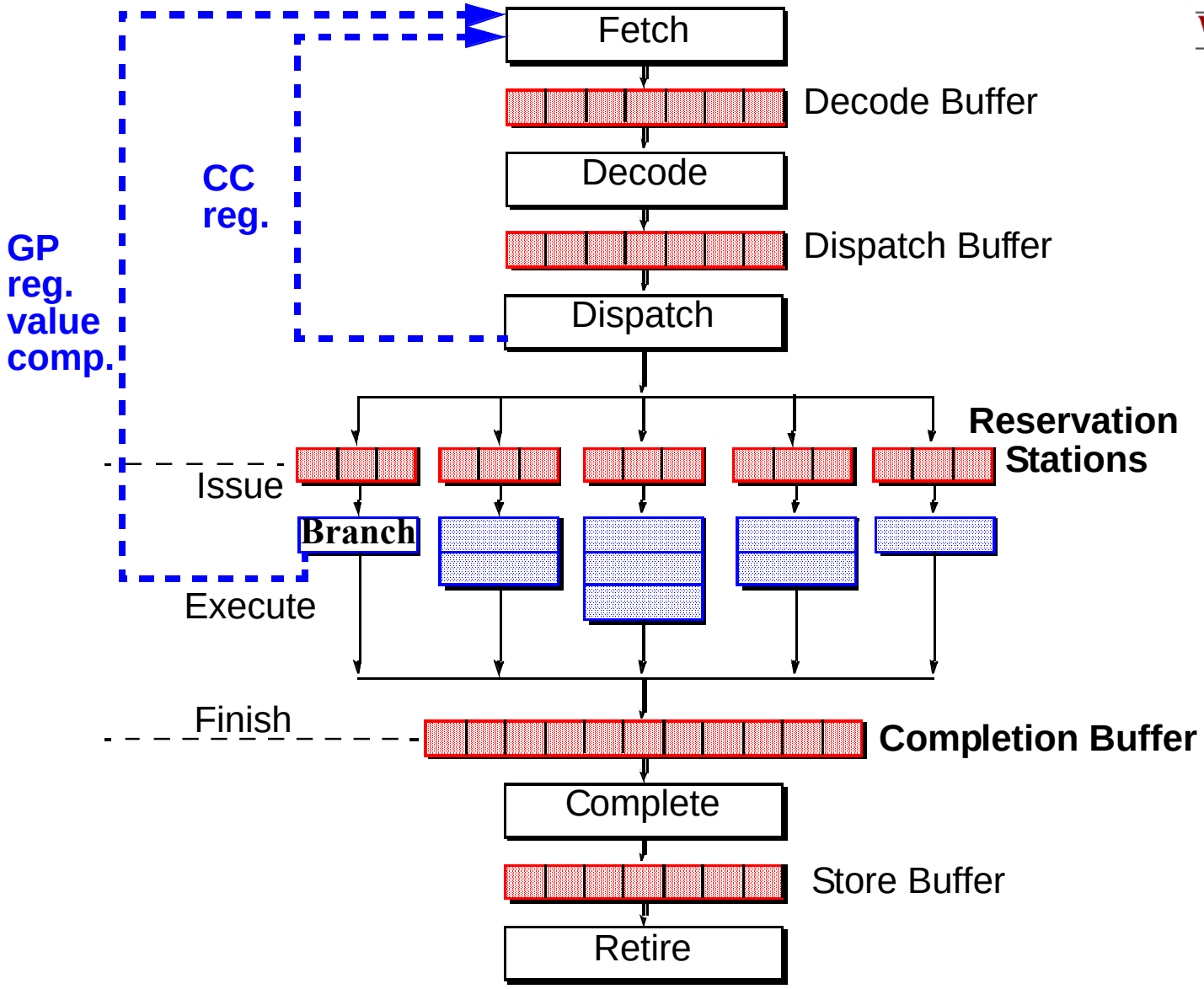
Branch Prediction

- Target address generation → Target speculation
  - Access register:
    - PC, General purpose register, Link register
  - Perform calculation:
    - +/- offset, autoincrement
- Condition resolution → Condition speculation
  - Access register:
    - Condition code register, General purpose register
  - Perform calculation:
    - Comparison of data register(s)

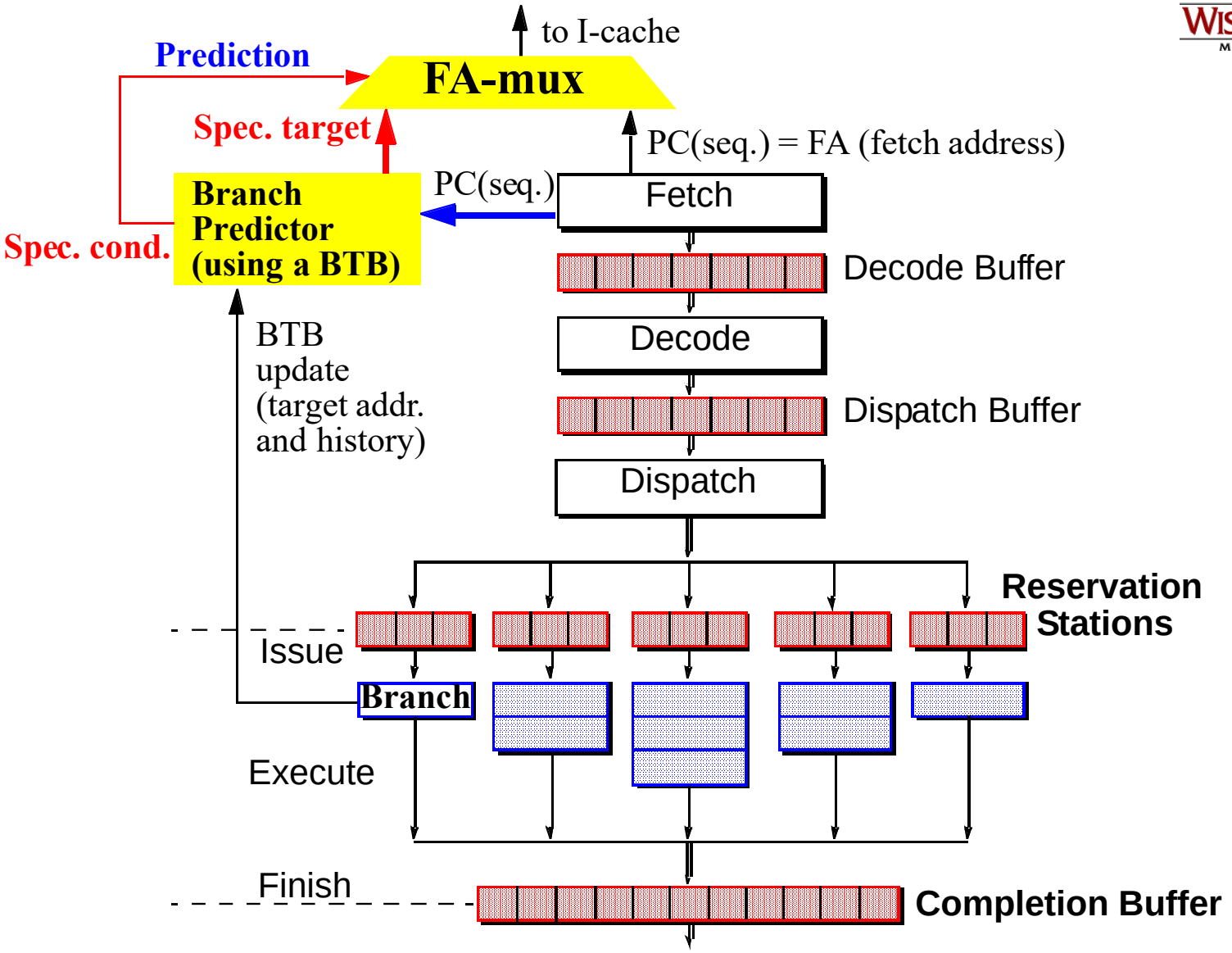
Target Address Generation



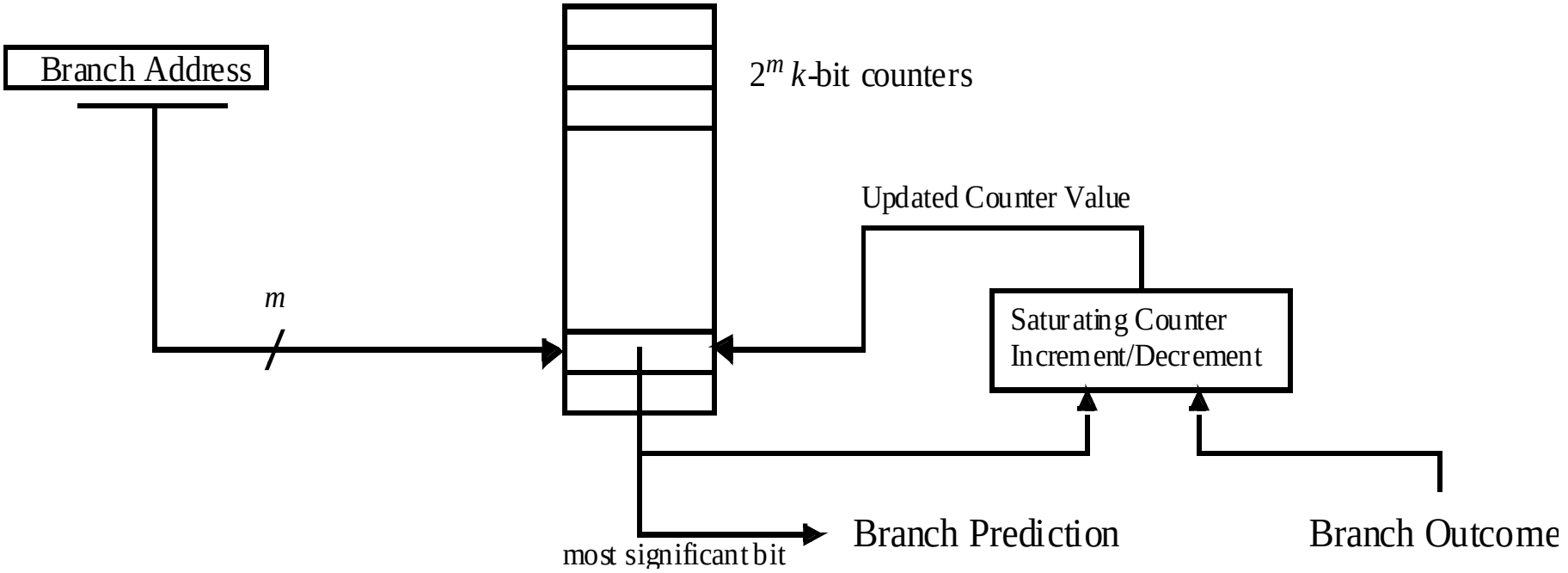
Branch Condition Resolution



Branch Instruction Speculation

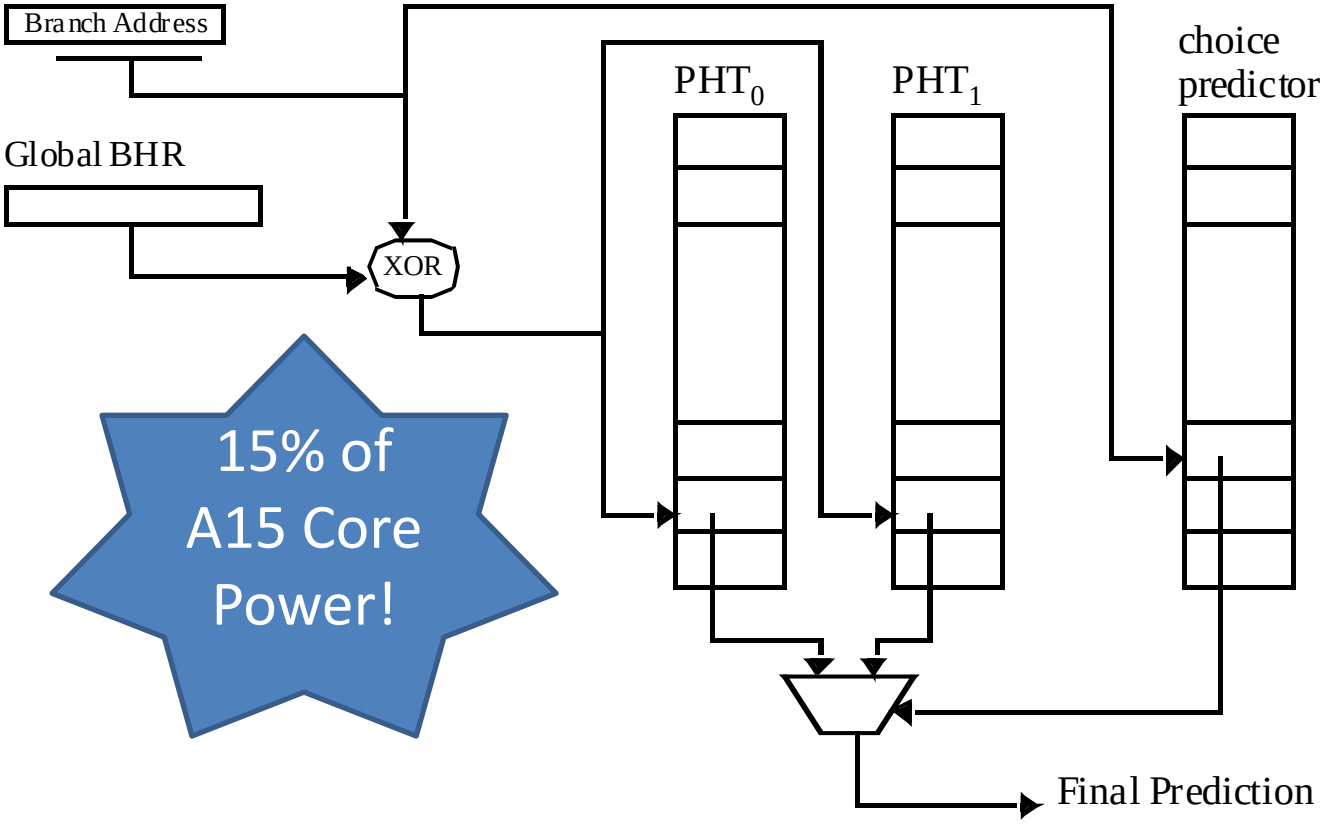


Hardware Smith Predictor



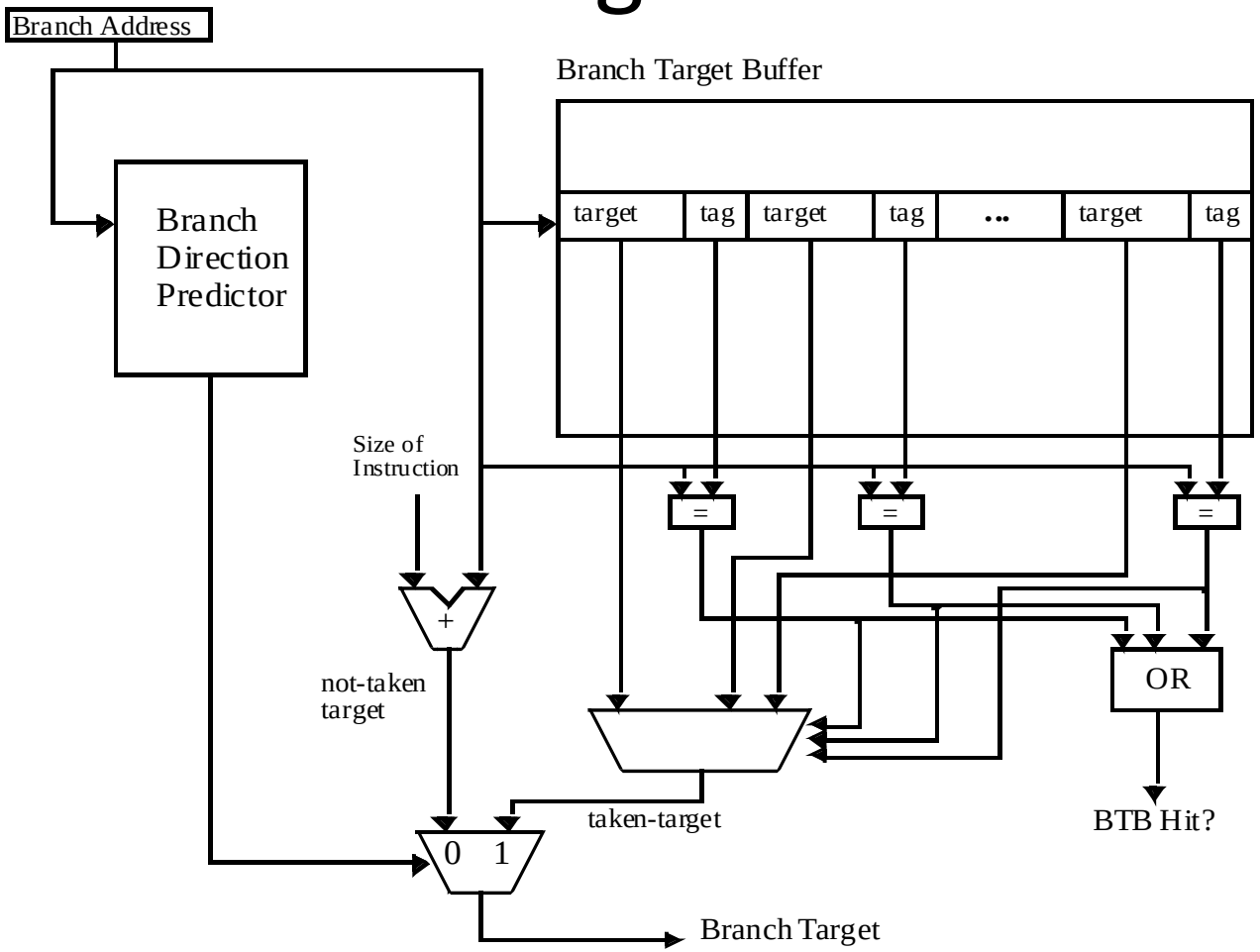
- Jim E. Smith. A Study of Branch Prediction Strategies. International Symposium on Computer Architecture, pages 135-148, May 1981
- Widely employed: Intel Pentium, PowerPC 604, MIPS R10000, etc.

Cortex A15: Bi-Mode Predictor



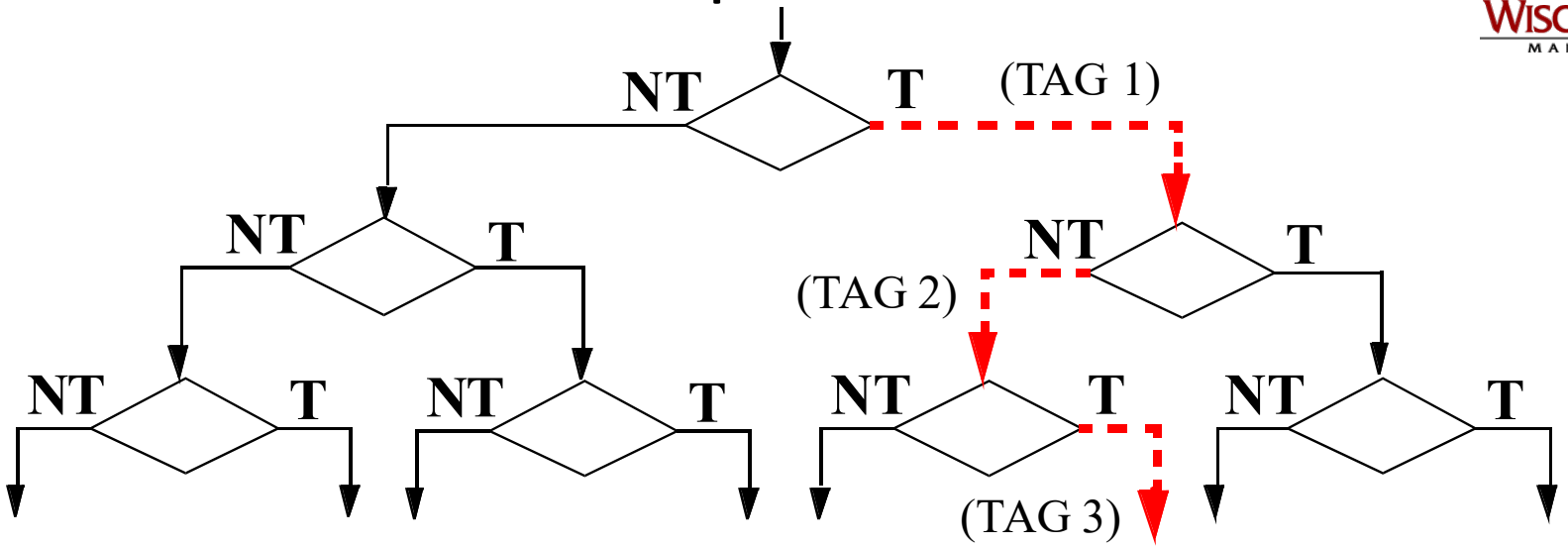
- PHT partitioned into T/NT halves
  - Selector chooses source
- Reduces negative interference, since most entries in PHT_0 tend towards NT, and most entries in PHT_1 tend towards T

Branch Target Prediction



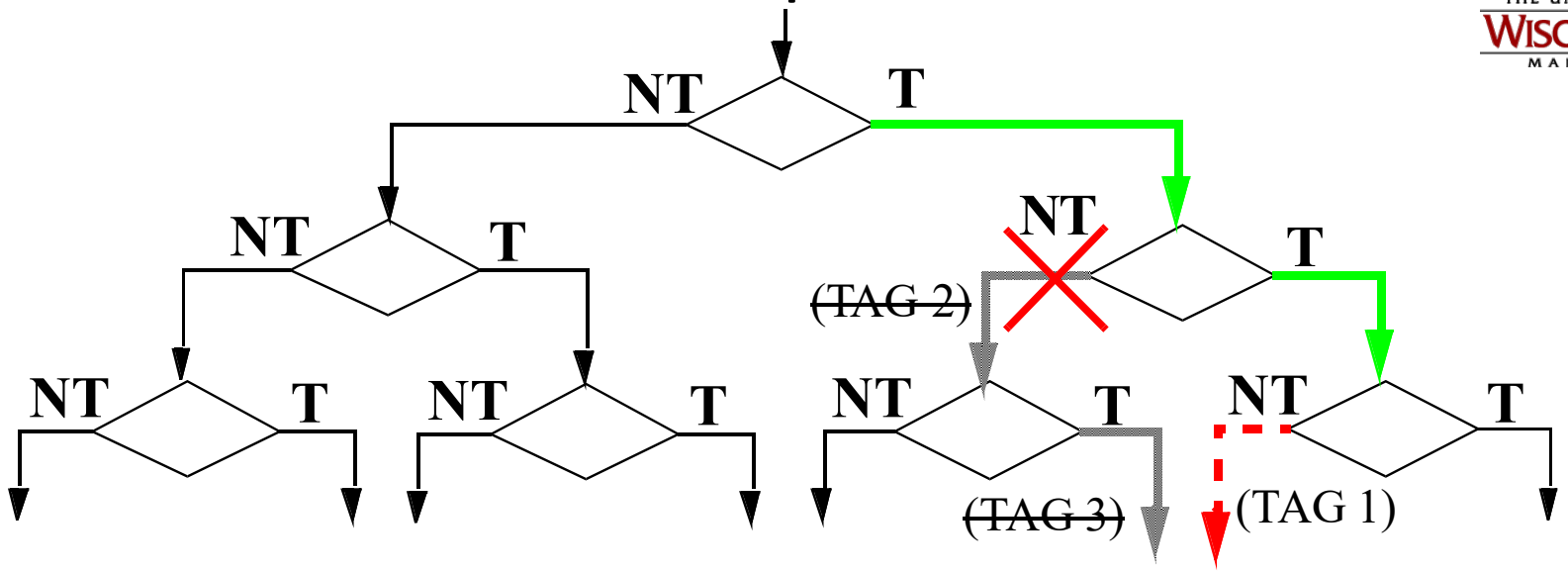
- Does not work well for function/procedure returns
- Does not work well for virtual functions, switch statements

Branch Speculation



- Leading Speculation
  - Done during the Fetch stage
  - Based on potential branch instruction(s) in the current fetch group
- Trailing Confirmation
  - Done during the Branch Execute stage
  - Based on the next Branch instruction to finish execution

Branch Speculation



- Start new correct path
  - Must remember the alternate (non-predicted) path
- Eliminate incorrect path
  - Must ensure that the mis-speculated instructions produce no side effects

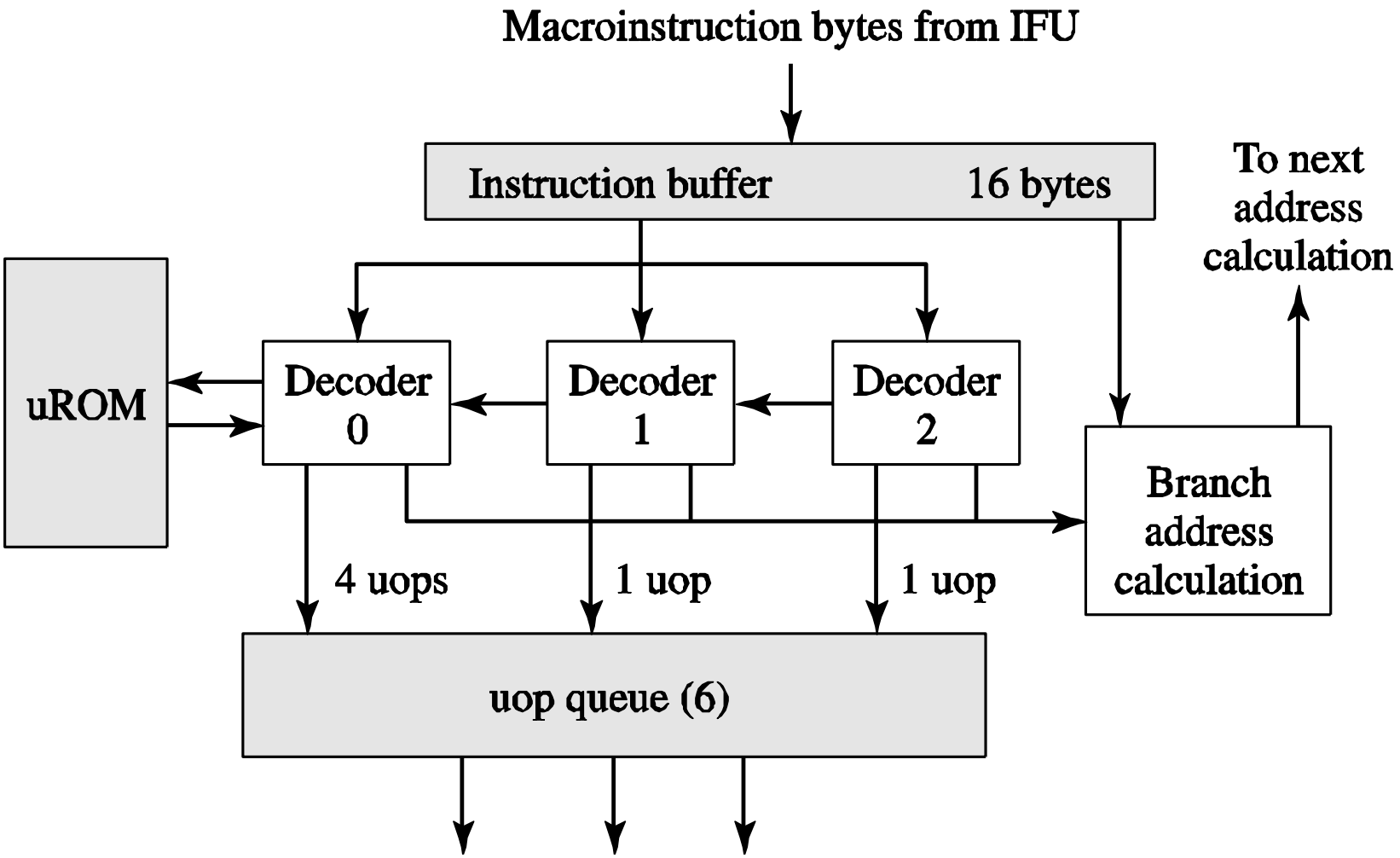
Mis-speculation Recovery

- Start new correct path
  1. Update PC with computed branch target (if predicted NT)
  2. Update PC with sequential instruction address (if predicted T)
  3. Can begin speculation again at next branch
- Eliminate incorrect path
  1. Use tag(s) to deallocate resources occupied by speculative instructions
  2. Invalidate all instructions in the decode and dispatch buffers, as well as those in reservation stations

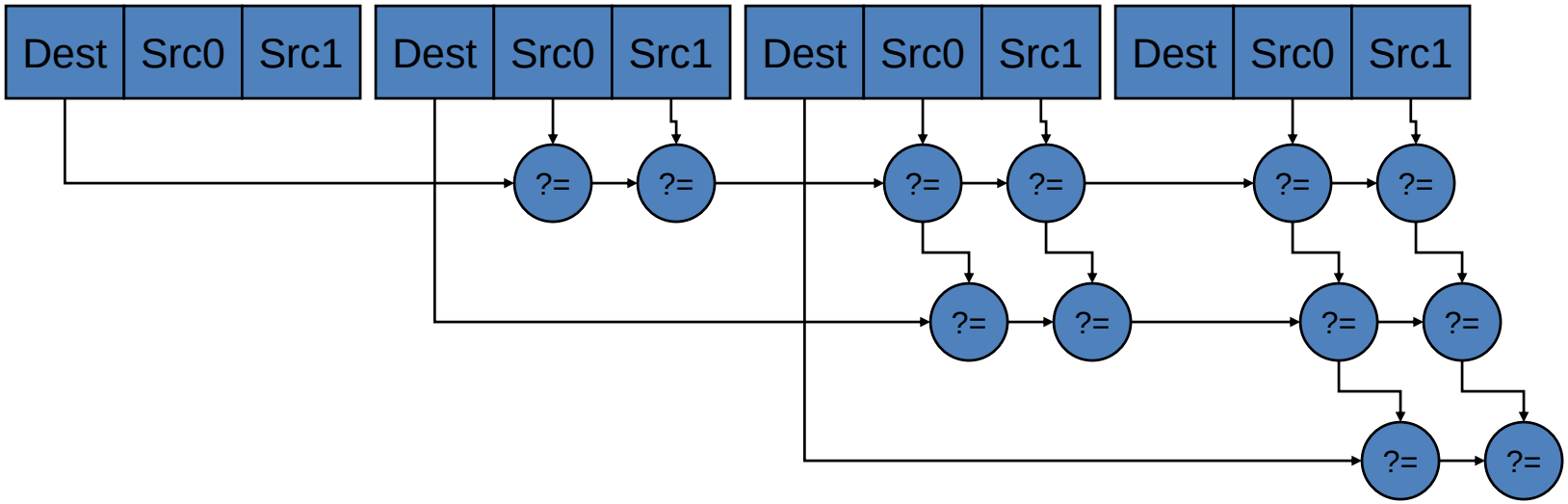
Parallel Decode

- Primary Tasks
  - Identify individual instructions (!)
  - Determine instruction types
  - Determine dependences between instructions
- Two important factors
  - Instruction set architecture
  - Pipeline width

Intel P6 Fetch/Decode



Dependence Checking

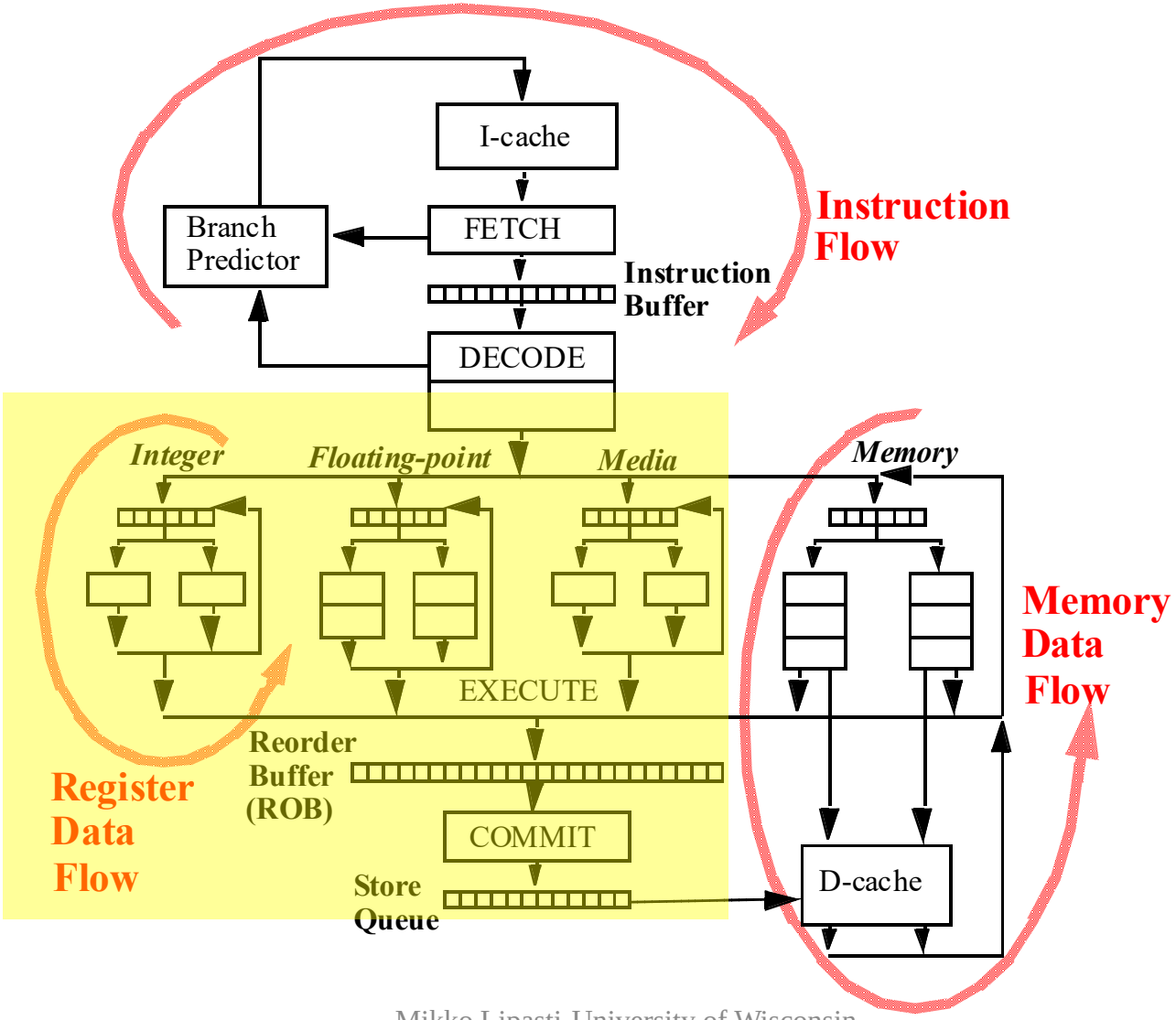


- Trailing instructions in fetch group
  - Check for dependence on leading instructions

Summary: Instruction Flow

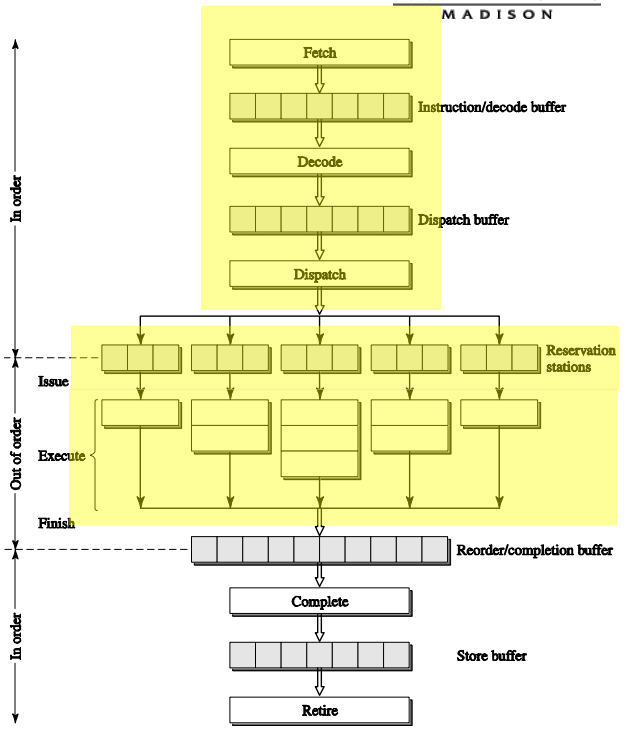
- Fetch group alignment
- Target address generation
  - Branch target buffer
- Branch condition prediction
- Speculative execution
  - Tagging/tracking instructions
  - Recovering from mispredicted branches
- Decoding in parallel

High-IPC Processor



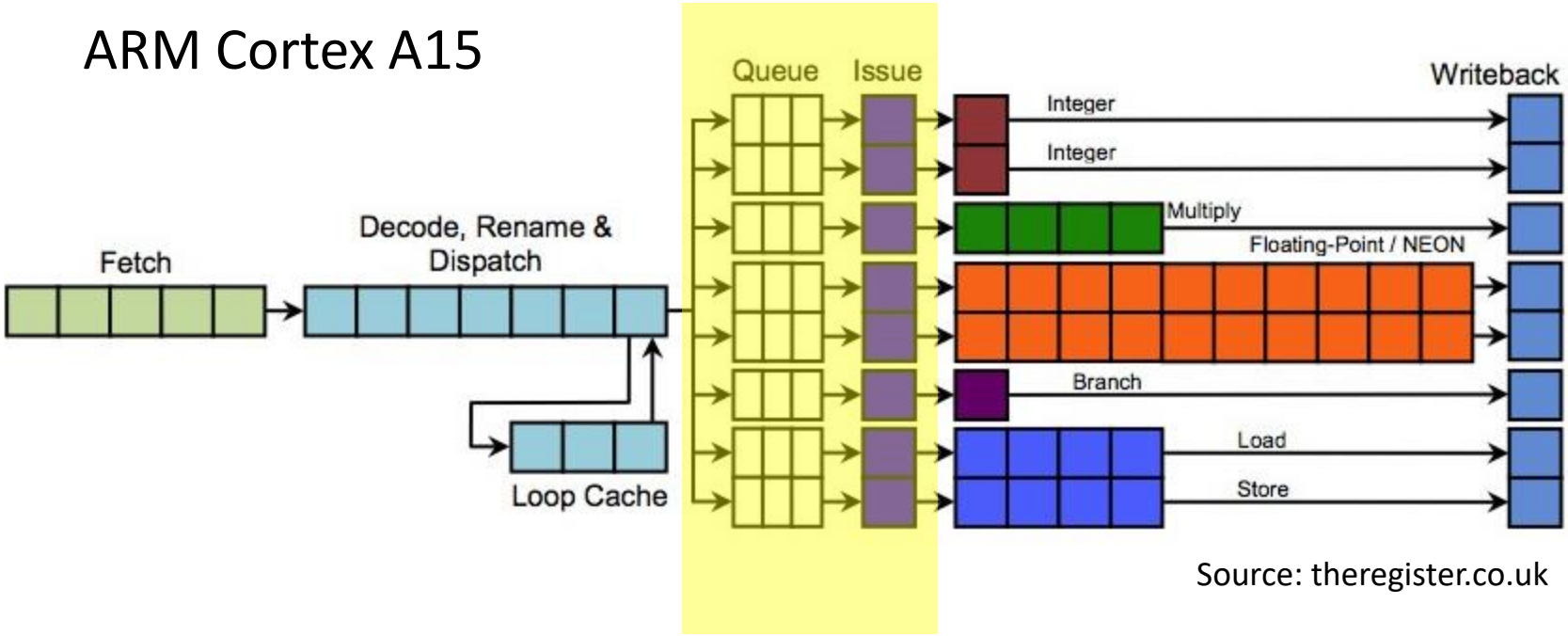
Register Data Flow

- Parallel pipelines
  - Centralized instruction fetch
  - Centralized instruction decode
- Diversified execution pipelines
  - Distributed instruction execution
- Data dependence linking
  - Register renaming to resolve true/false dependences
  - Issue logic to support out-of-order issue
  - Reorder buffer to maintain precise state



Issue Queues and Execution Lanes

ARM Cortex A15



Source: theregister.co.uk

Program Data Dependences

- True dependence (RAW)
  - j cannot execute until i produces its result

$$D(i) \cap R(j) \neq \phi$$

- Anti-dependence (WAR)
  - j cannot write its result until i has read its sources

$$R(i) \cap D(j) \neq \phi$$

- Output dependence (WAW)
  - j cannot write its result until i has written its result

$$D(i) \cap D(j) \neq \phi$$

Register Data Dependences

- Program data dependences cause hazards
  - True dependences (RAW)
  - Antidependences (WAR)
  - Output dependences (WAW)
- When are registers read and written?
  - Out of program order to extract maximum ILP
  - Hence, any and all of these can occur

- Solution to all three: register renaming

Register Renaming: WAR/WAW

- Widely employed (Core i7, Cortex A15, …)
- Resolving WAR/WAW:
  - Each register write gets unique "rename register"
  - Writes are committed in program order at Writeback
  - WAR and WAW are not an issue
    - All updates to "architected state" delayed till writeback
    - Writeback stage always later than read stage
  - Reorder Buffer (ROB) enforces in-order writeback

Add R3 <= …	P32 <= …
Sub R4 <= …	P33 <= …
And R3 <= …	P35 <= …

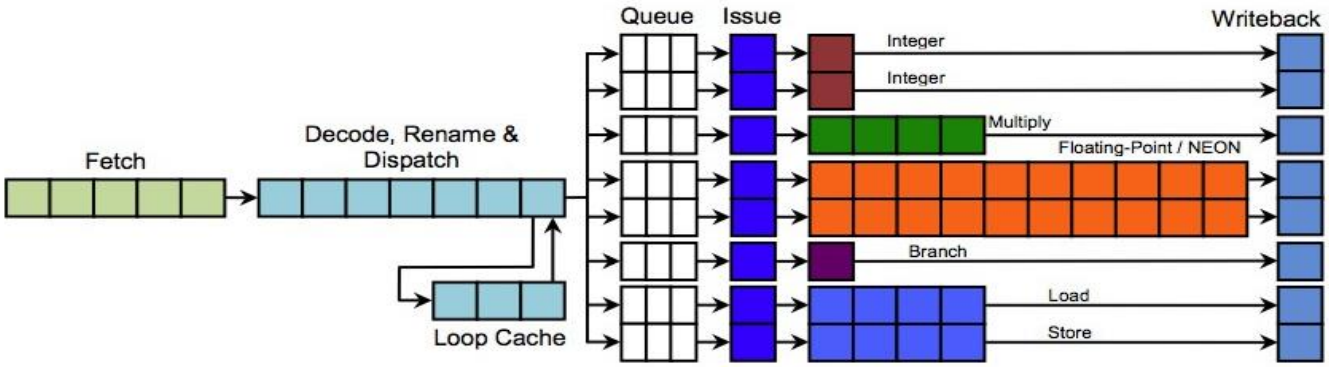
Register Renaming: RAW

- In order, at dispatch:
  - Source registers checked to see if "in flight"
    - Register map table keeps track of this
    - If not in flight, can be read from the register file
    - If in flight, look up "rename register" tag (IOU)
  - Then, allocate new register for register write

Add R3 <= R2 + R1	P32 <= P2 + P1
Sub R4 <= R3 + R1	P33 <= P32 - P1
And R3 <= R4 & R2	P35 <= P33 & P2

Register Renaming: RAW

- Advance instruction to instruction queue
  - Wait for rename register tag to trigger issue
- Issue queue/reservation station enables out-of-order issue
  - Newer instructions can bypass stalled instructions

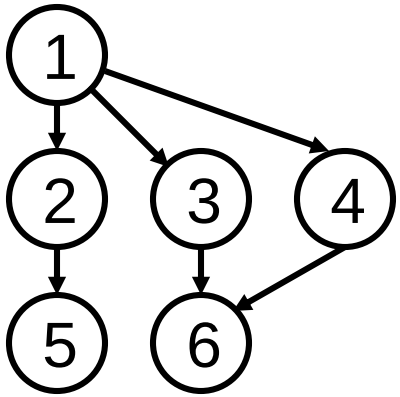


Source: theregister.co.uk

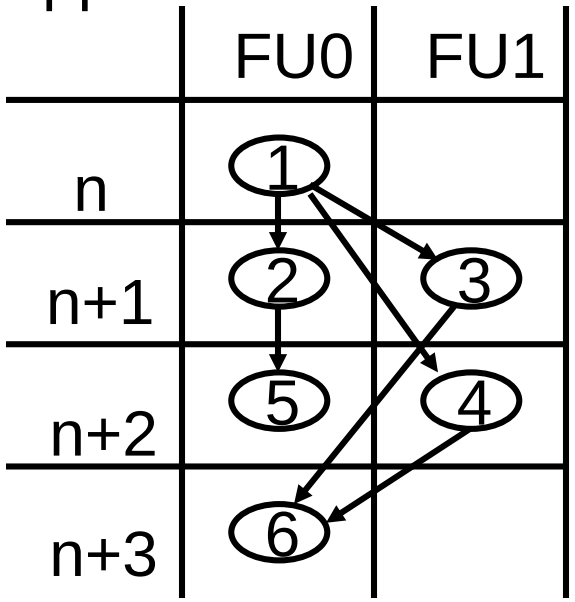
Instruction scheduling

- A process of mapping a series of instructions into execution resources
  - Decides when and where an instruction is executed

- Data dependence graph



- Mapped to two FUs

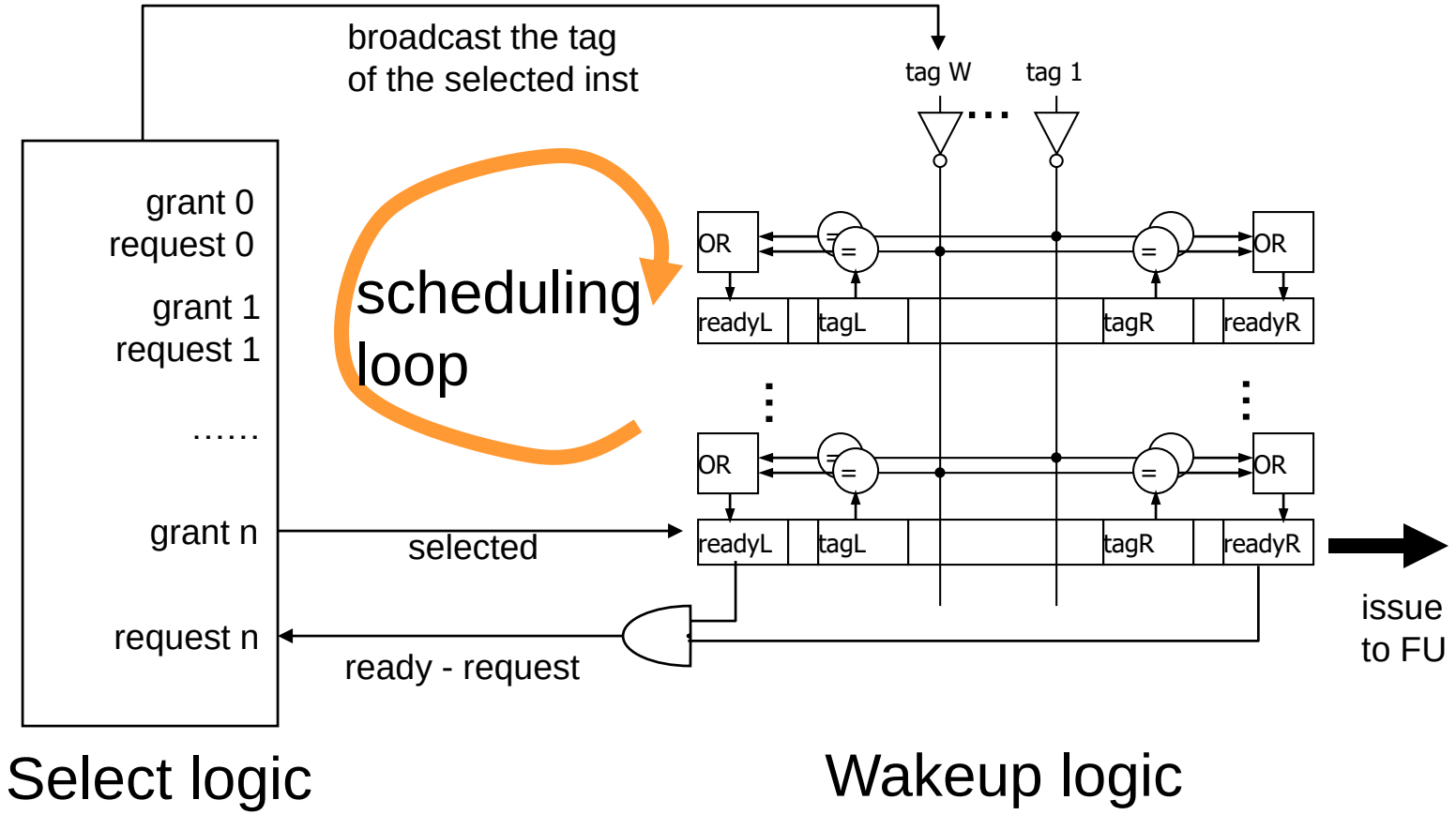


Instruction scheduling

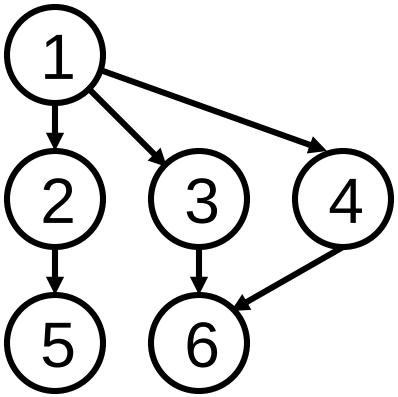
- A set of wakeup and select operations
  - Wakeup
    - Broadcasts the tags of parent instructions selected
    - Dependent instruction gets matching tags, determines if source operands are ready
    - Resolves true data dependences
  - Select
    - Picks instructions to issue among a pool of ready instructions
    - Resolves resource conflicts
      - Issue bandwidth
      - Limited number of functional units / memory ports

Scheduling loop

- Basic wakeup and select operations

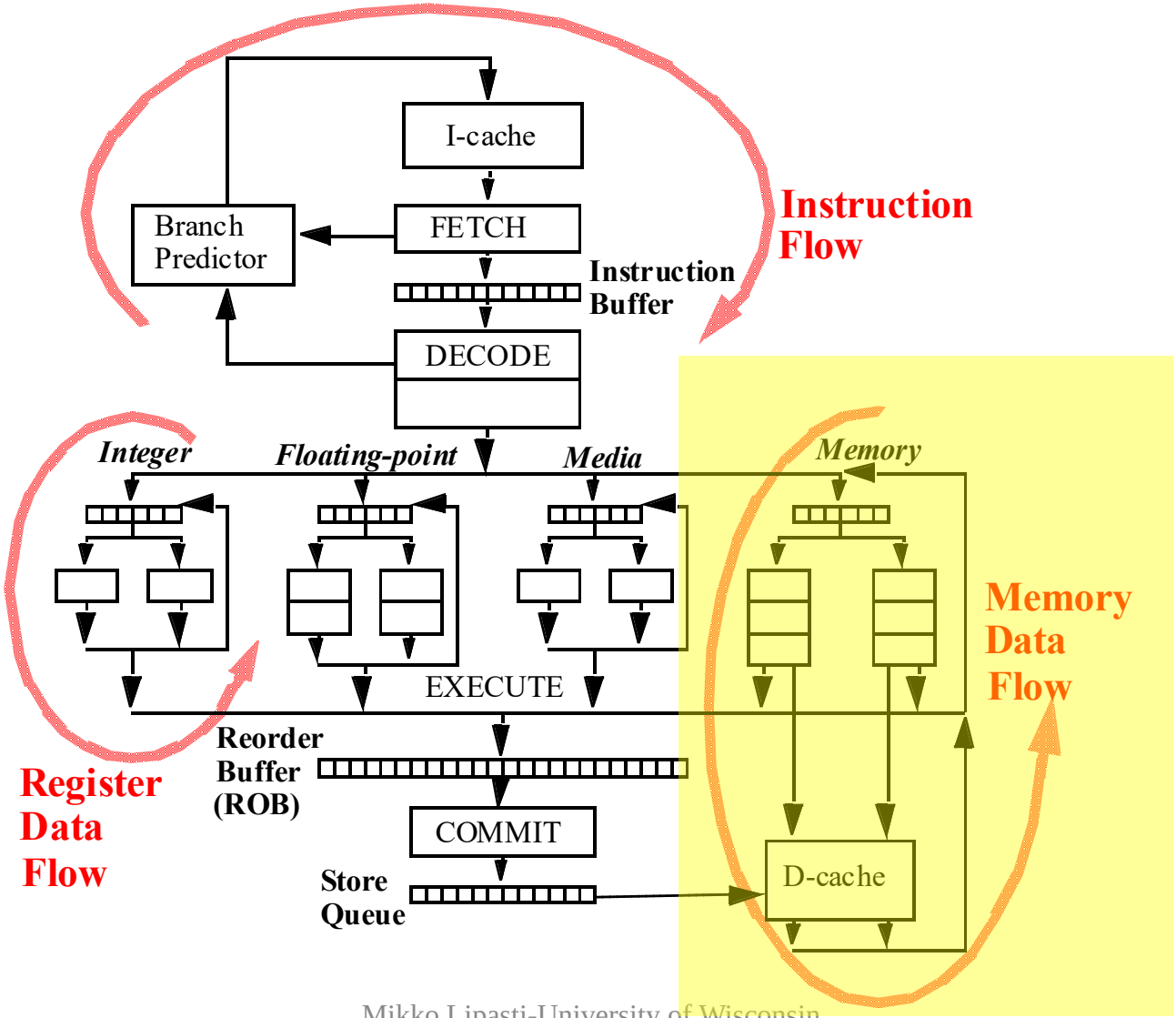


Wakeup and Select



	FU0	FU1	Ready inst to issue	Wakeup / select
n	1		1	Select 1 Wakeup 2,3,4
n+1	2	3	2, 3, 4	Select 2, 3 Wakeup 5, 6
n+2	5	4	4, 5	Select 4, 5 Wakeup 6
n+3	6		6	Select 6

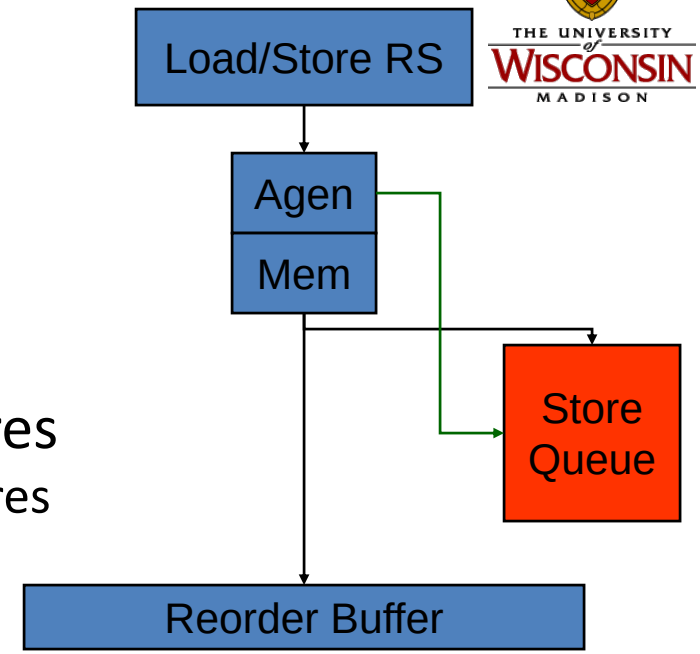
High-IPC Processor



Memory Data Flow

- Resolve WAR/WAW/RAW memory dependences
  - MEM stage can occur out of order
- Provide high bandwidth to memory hierarchy
  - Non-blocking caches

Memory Data Dependences



- WAR/WAW: stores commit in order
  - Hazards not possible.
- RAW: loads must check pending stores
  - Store queue keeps track of pending stores
  - Loads check against these addresses
  - Similar to register bypass logic
  - Comparators are 64 bits wide
  - Must consider position (age) of loads and stores

- Major source of complexity in modern designs
  - Store queue lookup is position-based
  - What if store address is not yet known?

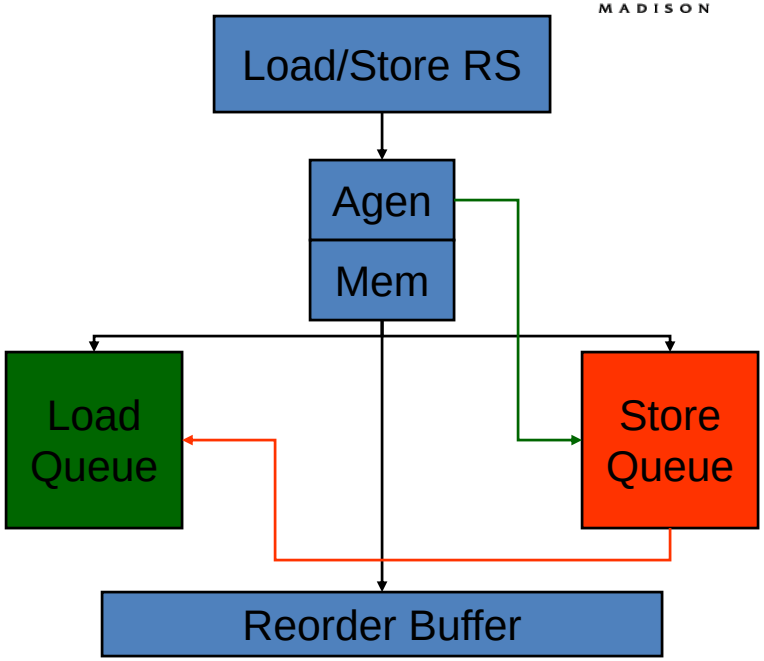
Optimizing Load/Store Disambiguation

- Non-speculative load/store disambiguation
  1. Loads wait for addresses of all prior stores
  2. Full address comparison
  3. Bypass if no match, forward if match
- (1) can limit performance:

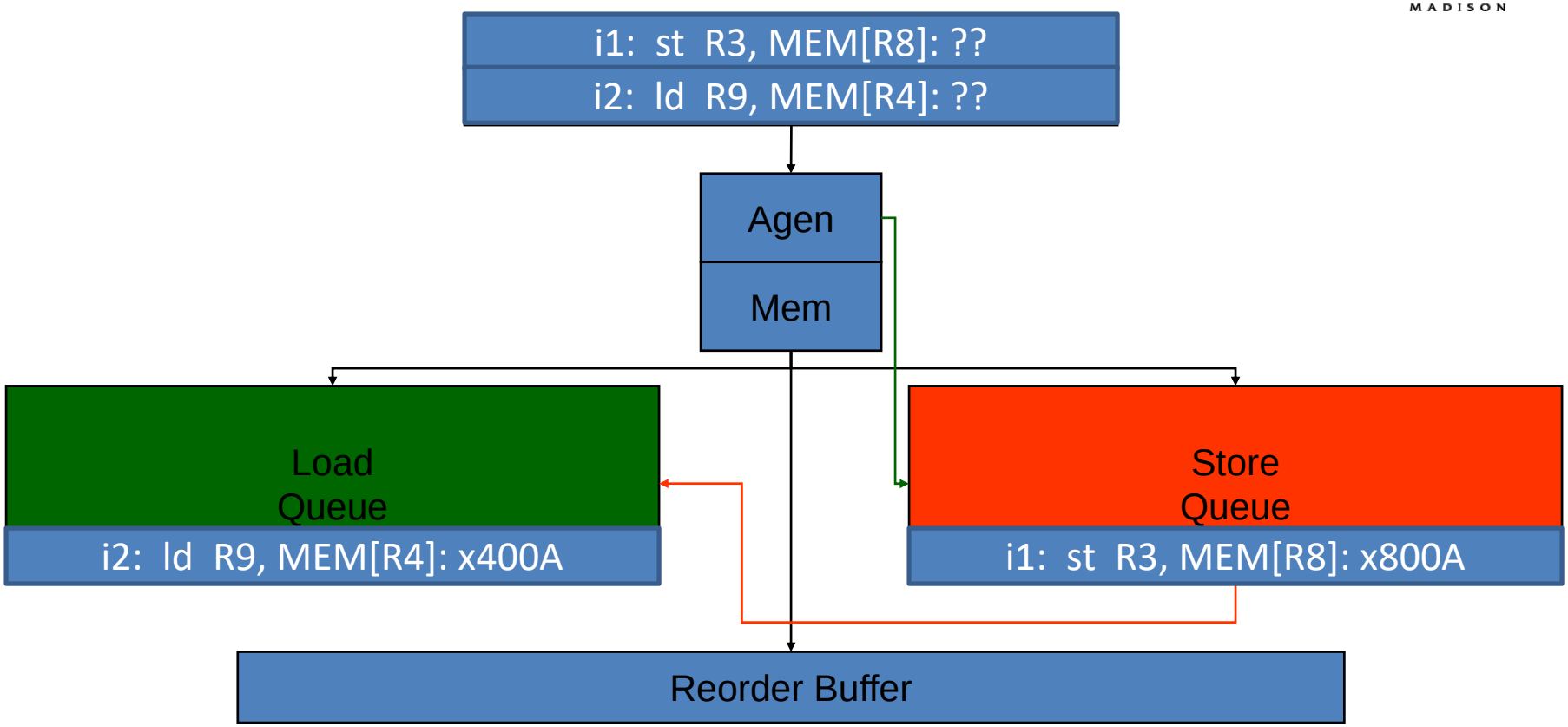
load r5,MEM[r3] ← cache miss
store r7, MEM[r5] ← RAW for agen, stalled
…
load r8, MEM[r9] ← independent load stalled

Speculative Disambiguation

- What if aliases are rare?
  1. Loads don't wait for addresses of all prior stores
  2. Full address comparison of stores that are ready
  3. Bypass if no match, forward if match
  4. Check all store addresses when they commit
     - No matching loads – speculation was correct
     - Matching unbypassed load – incorrect speculation
  5. Replay starting from incorrect load

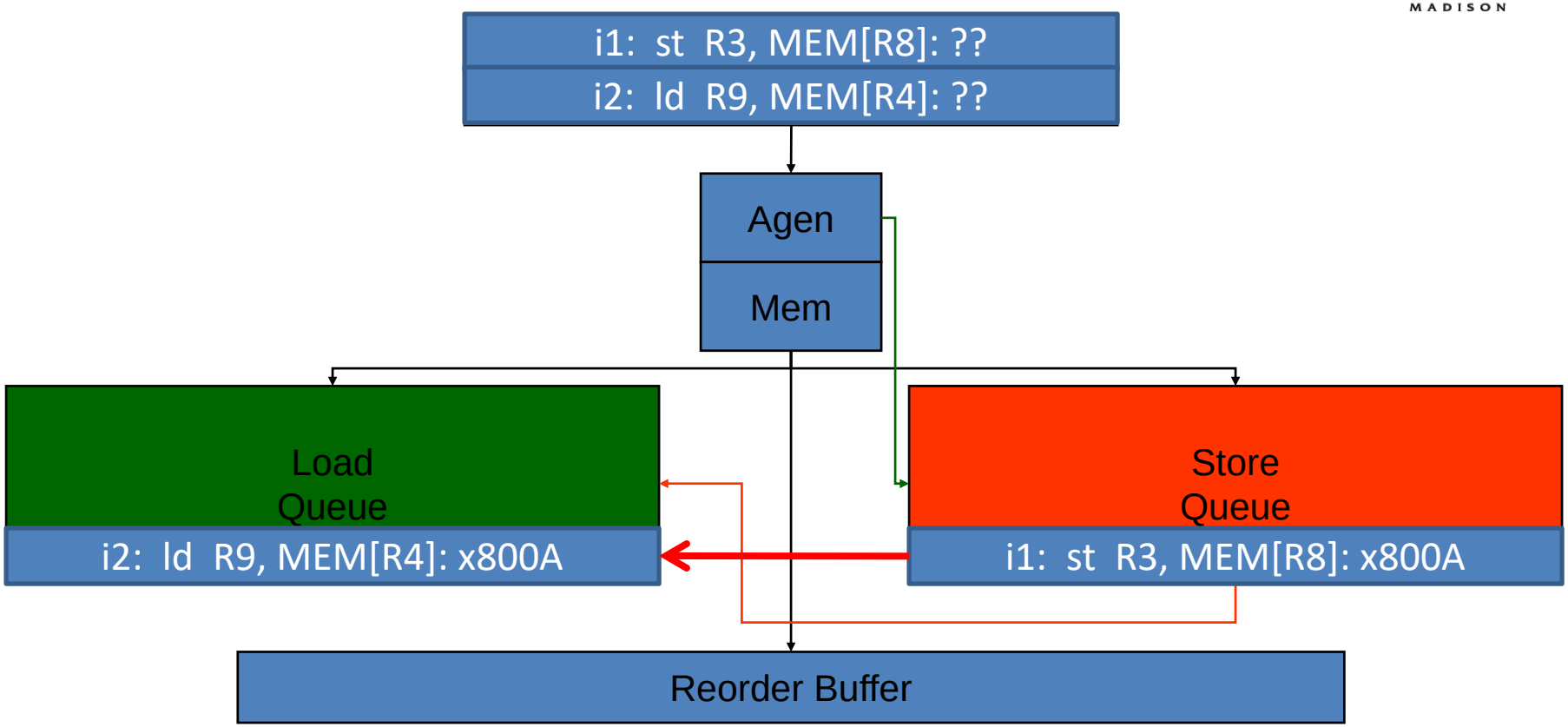


Speculative Disambiguation: Load Bypass



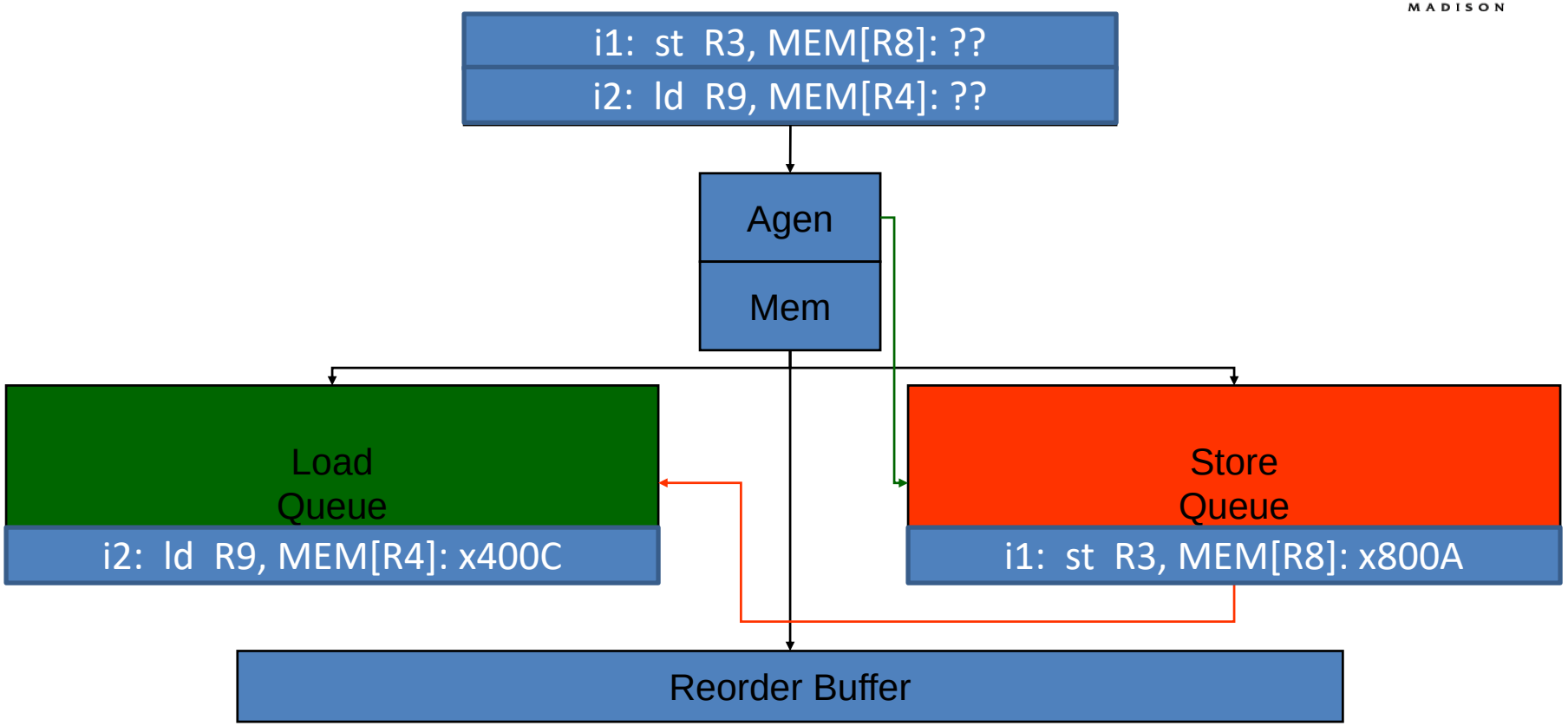
- i1 and i2 issue in program order
- i2 checks store queue (no match)

Speculative Disambiguation: Load Forward



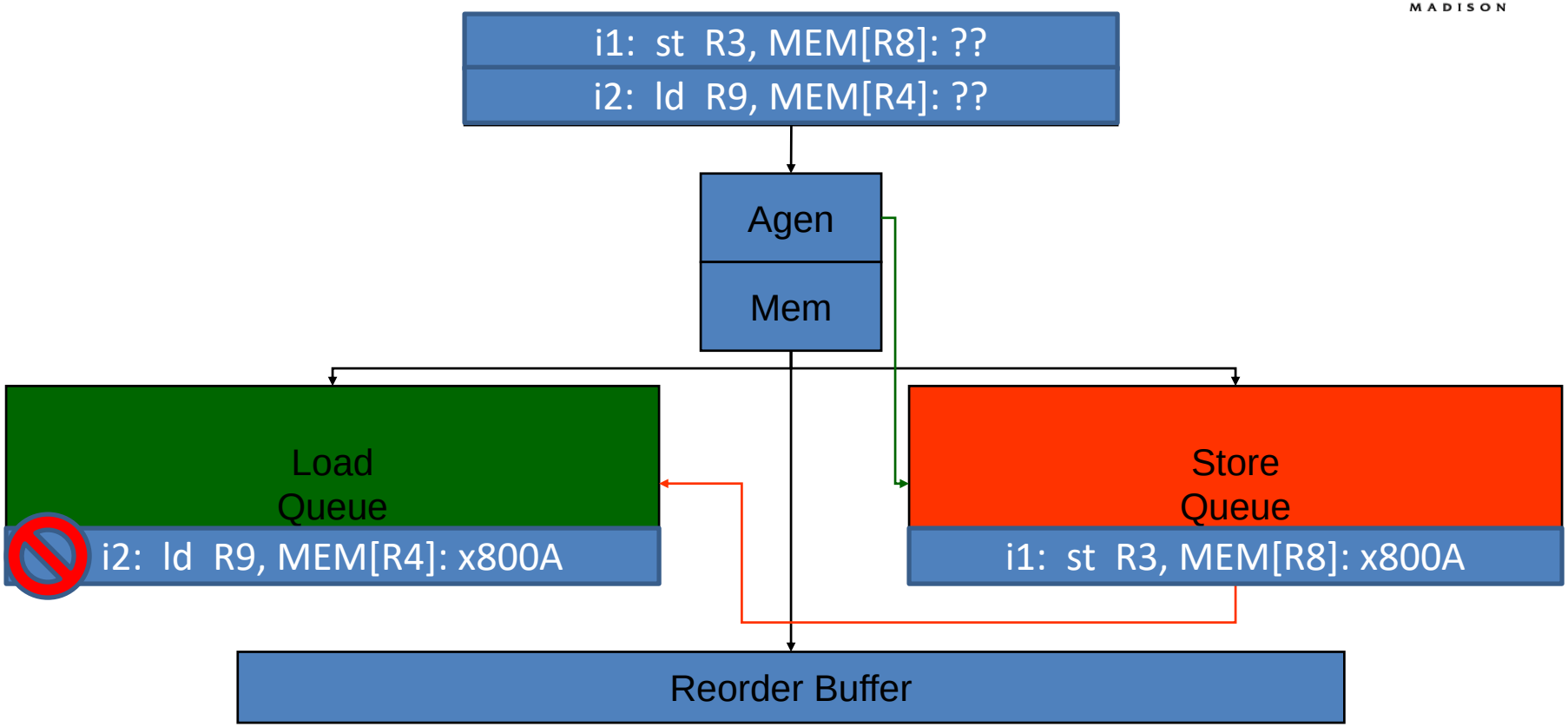
- i1 and i2 issue in program order
- i2 checks store queue (match=>forward)

Speculative Disambiguation: Safe Speculation



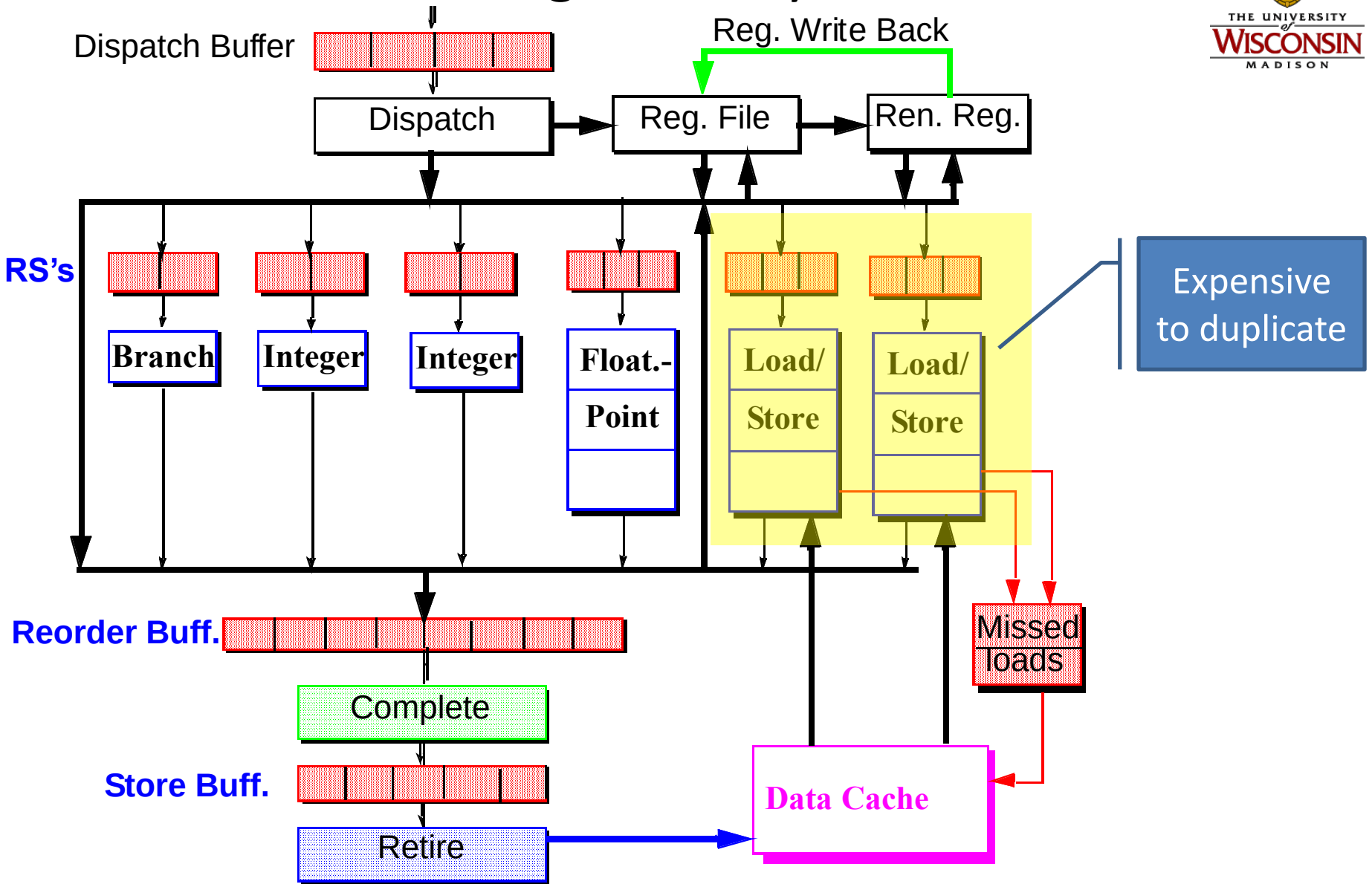
- i1 and i2 issue out of program order
- i1 checks load queue at commit (no match)

Speculative Disambiguation: Violation



- i1 and i2 issue out of program order
- i1 checks load queue at commit (match)
  - i2 marked for replay

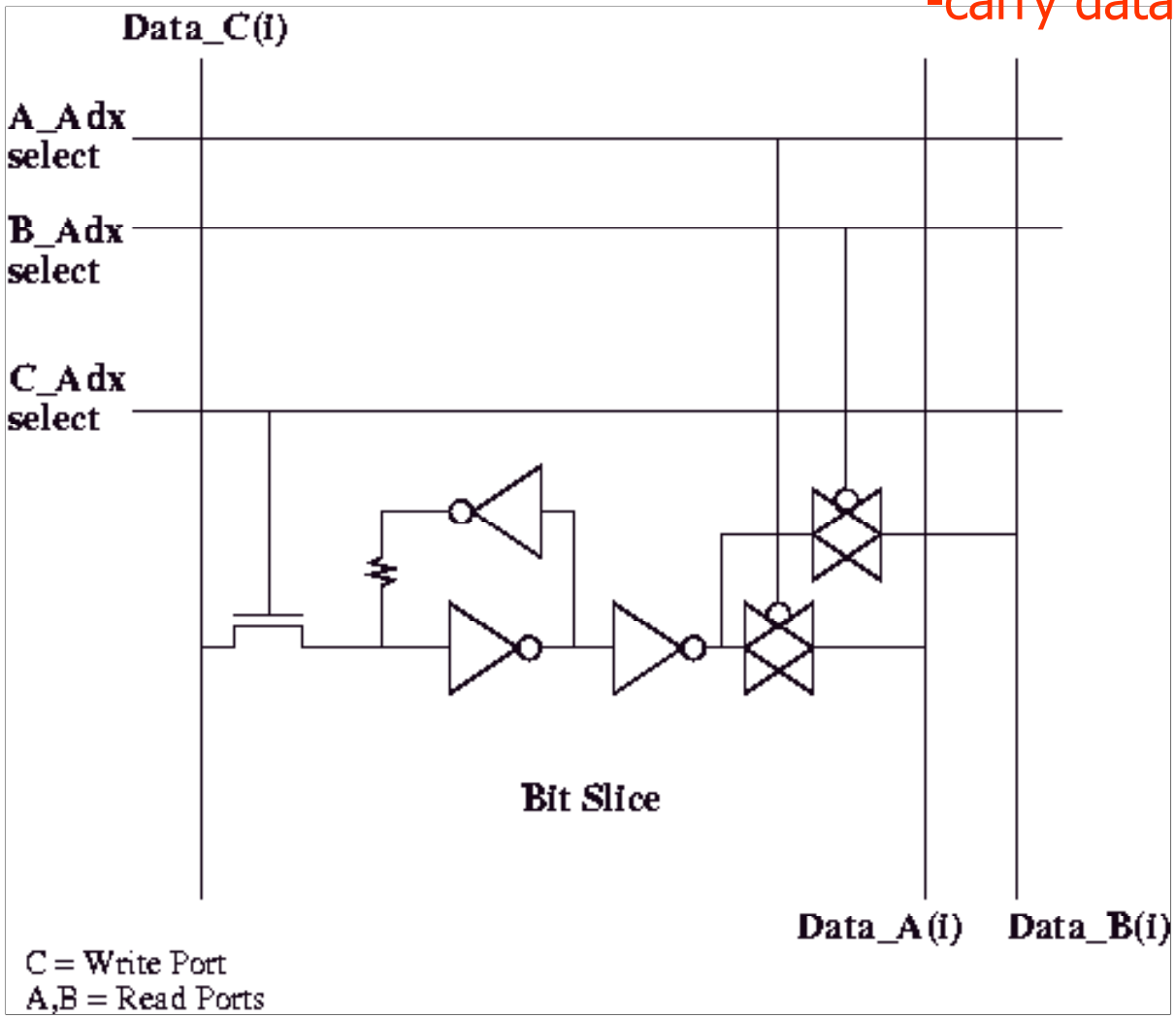
Increasing Memory Bandwidth



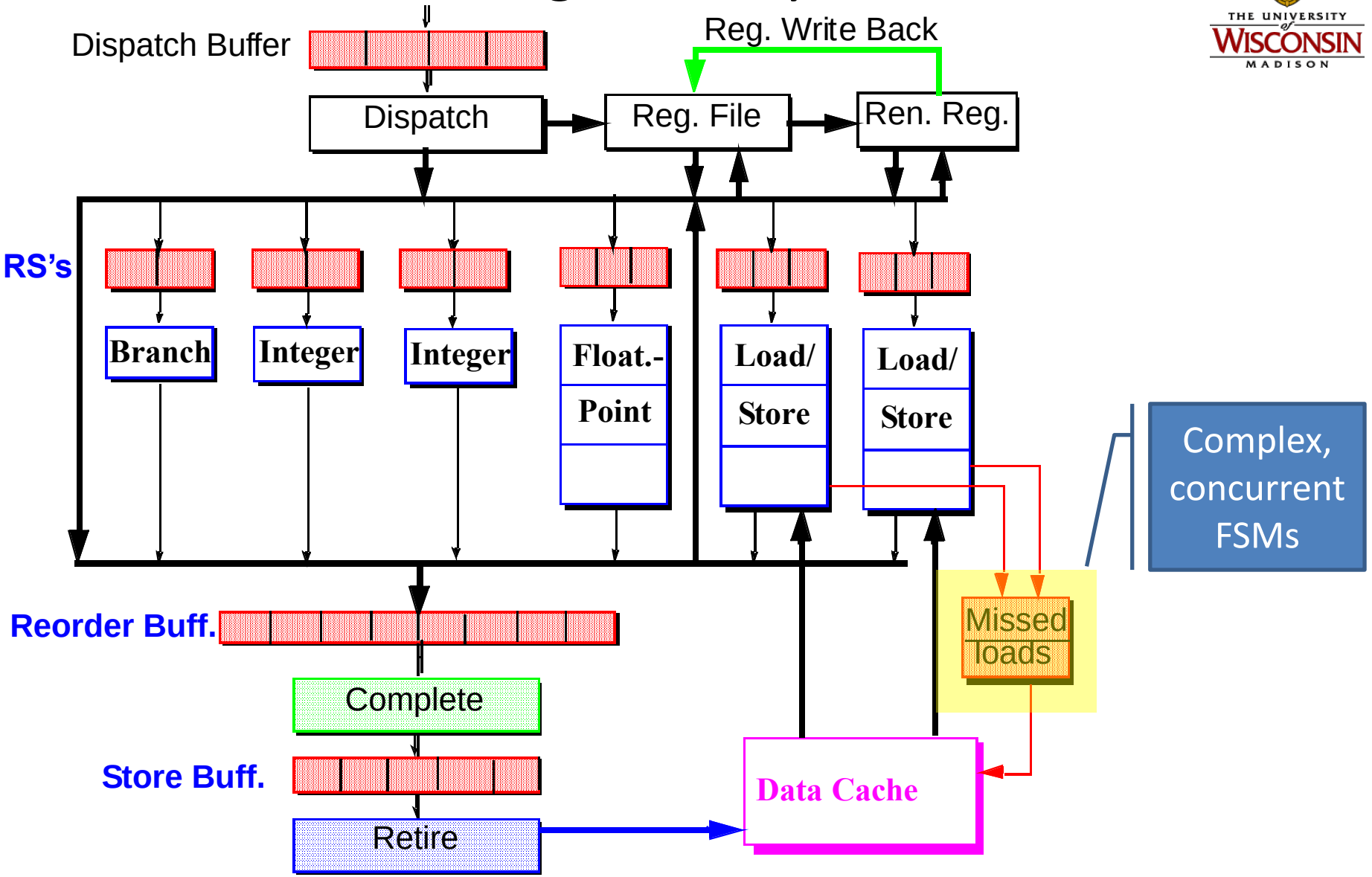
True Multiporting of SRAM

"Bit" Lines
-carry data in/out

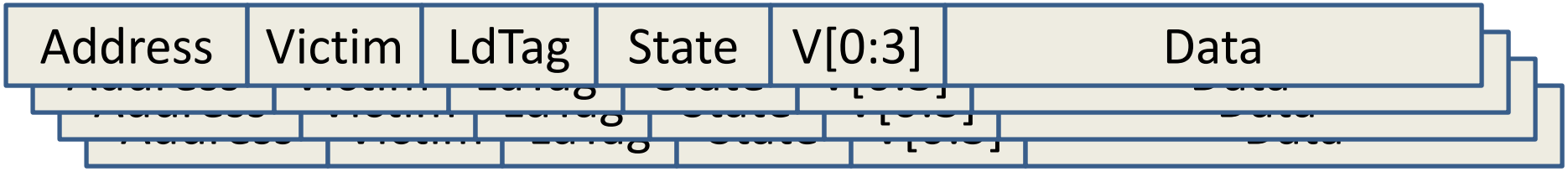
"Word" Lines
-select a row



Increasing Memory Bandwidth

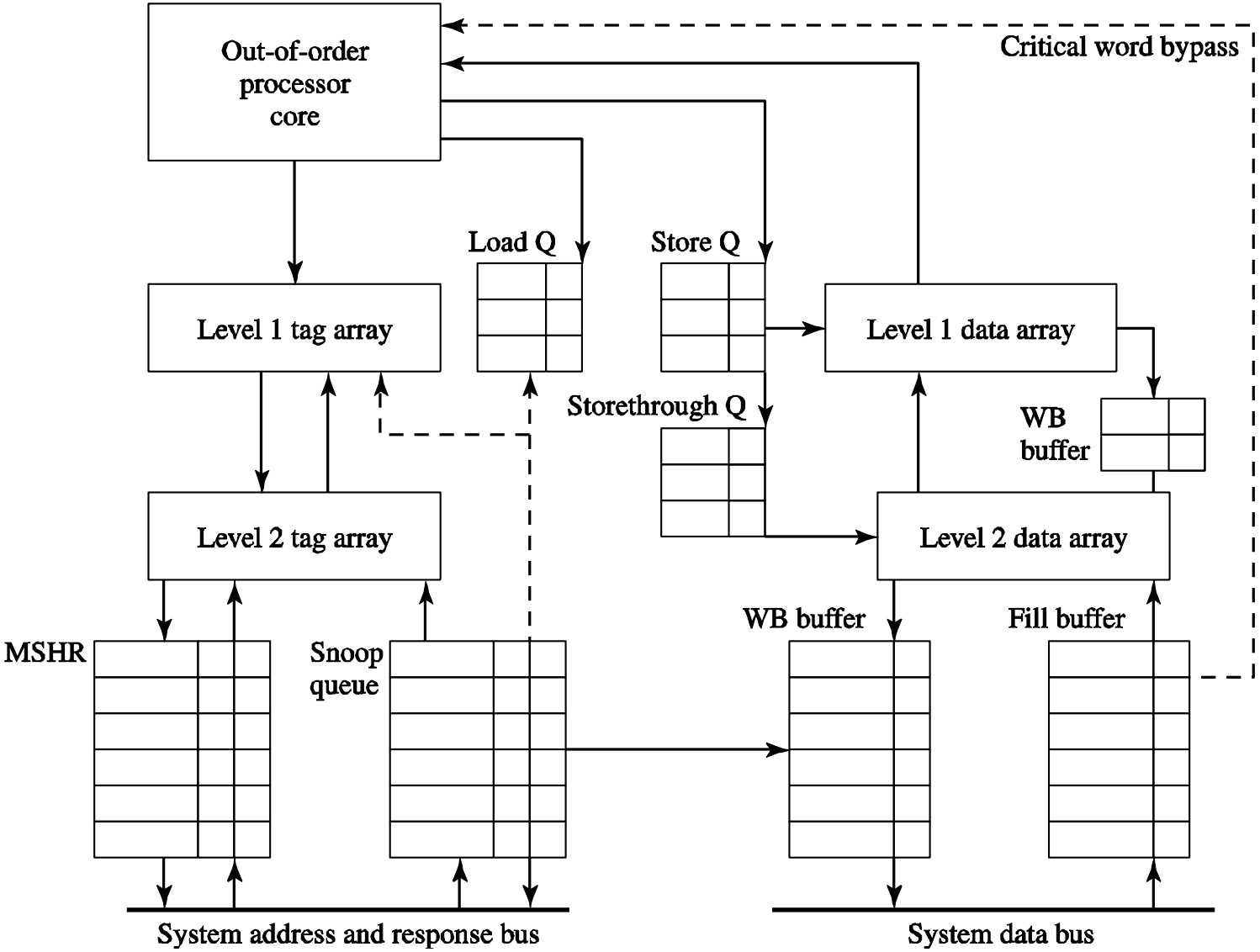


Miss Status Handling Register



- Each MSHR entry keeps track of:
  - Address: miss address
  - Victim: set/way to replace
  - LdTag: which load (s) to wake up
  - State: coherence state, fill status
  - V[0:3]: subline valid bits
  - Data: block data to be filled into cache

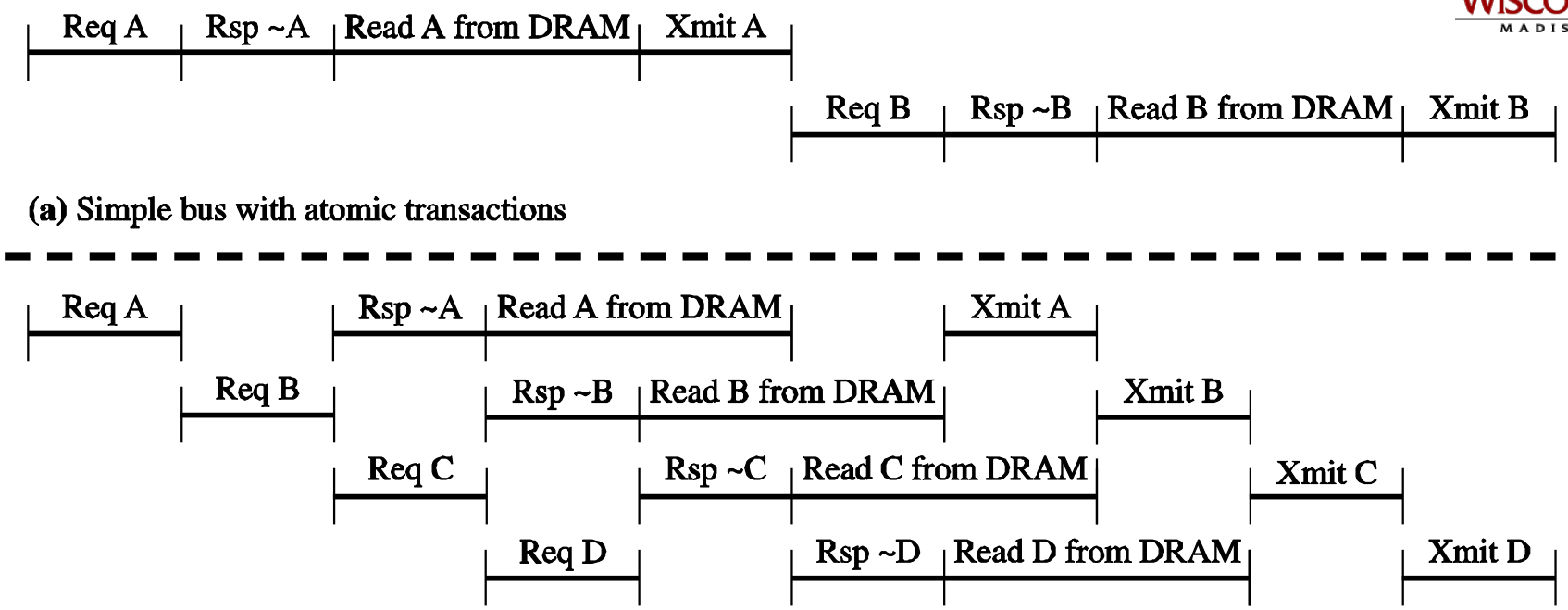
Coherent Memory Interface



Coherent Memory Interface

- Load Queue
  - Tracks inflight loads for aliasing, coherence
- Store Queue
  - Defers stores until commit, tracks aliasing
- Storethrough Queue or Write Buffer or Store Buffer
  - Defers stores, coalesces writes, must handle RAW
- MSHR
  - Tracks outstanding misses, enables *lockup-free caches* [Kroft ISCA 91]
- Snoop Queue
  - Buffers, tracks incoming requests from coherent I/O, other processors
- Fill Buffer
  - Works with MSHR to hold incoming partial lines
- Writeback Buffer
  - Defers writeback of evicted line (demand miss handled first)

Split Transaction Bus



- "Packet switched" vs. "circuit switched"
- Release bus after request issued
- Allow multiple concurrent requests to overlap memory latency
- Complicates control, arbitration, and coherence protocol
  - *Transient* states for pending blocks (e.g. "req. issued but not completed")

Memory Consistency

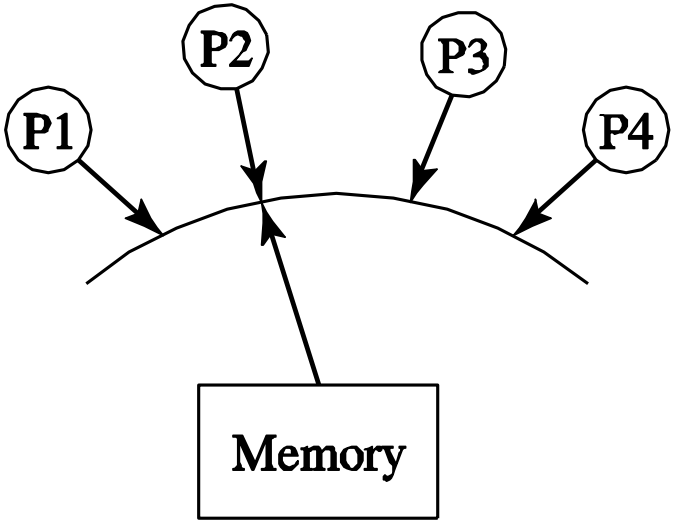
```
Reorder
load
before
store
```

```
Proc0

st A=1
if (load B==0) {
   ...critical section
}
```

```
Proc1

st B=1
if (load A==0) {
   ...critical section
}
```

- How are memory references from different processors interleaved?
- If this is not well-specified, synchronization becomes difficult or even impossible
  - ISA must specify consistency model
- Common example using Dekker's algorithm for synchronization
  - If load reordered ahead of store (as we assume for a baseline OOO CPU)
  - Both Proc0 and Proc1 enter critical section, since both observe that other's lock variable (A/B) is not set
- If consistency model allows loads to execute ahead of stores, Dekker's algorithm no longer works
  - Common ISAs allow this: IA-32, PowerPC, SPARC, Alpha

Sequential Consistency [Lamport 1979]

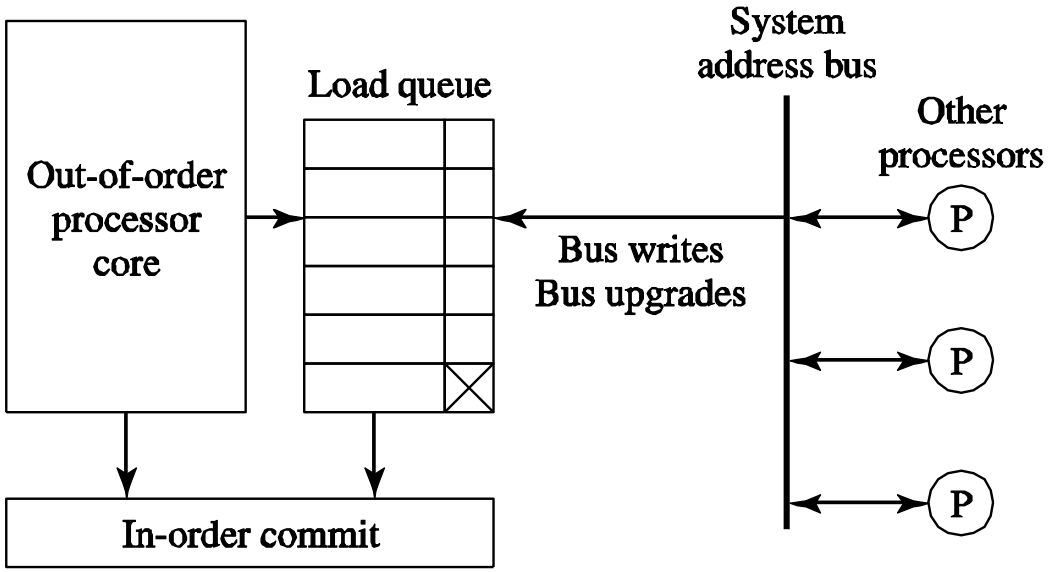


- Processors treated as if they are interleaved processes on a single time-shared CPU
- All references must fit into a total global order or interleaving that does not violate any CPU's program order
  - Otherwise sequential consistency not maintained
- Now Dekker's algorithm will work
- Appears to preclude any OOO memory references
  - Hence precludes any real benefit from OOO CPUs

High-Performance Sequential Consistency

- Coherent caches isolate CPUs if no sharing is occurring
  - Absence of coherence activity means CPU is free to reorder references
- Still have to order references with respect to misses and other coherence activity (snoops)
- Key: use speculation
  - Reorder references speculatively
  - Track which addresses were touched speculatively
  - Force replay (in order execution) of such references that collide with coherence activity (snoops)

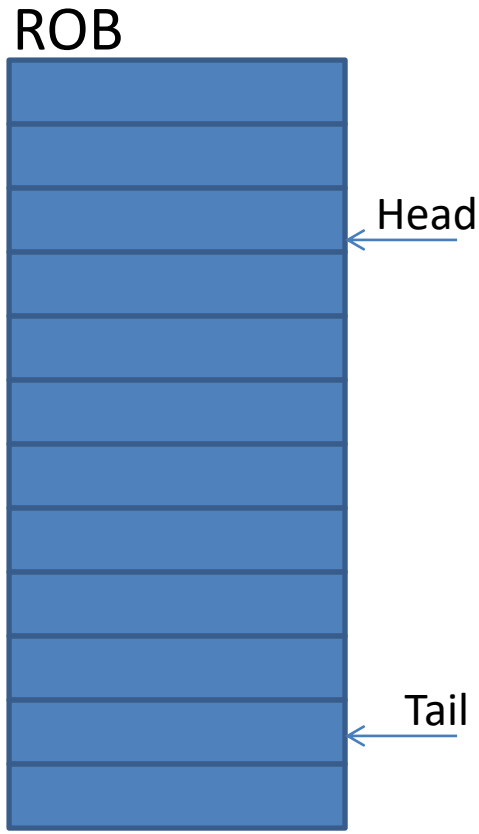
High-Performance Sequential Consistency



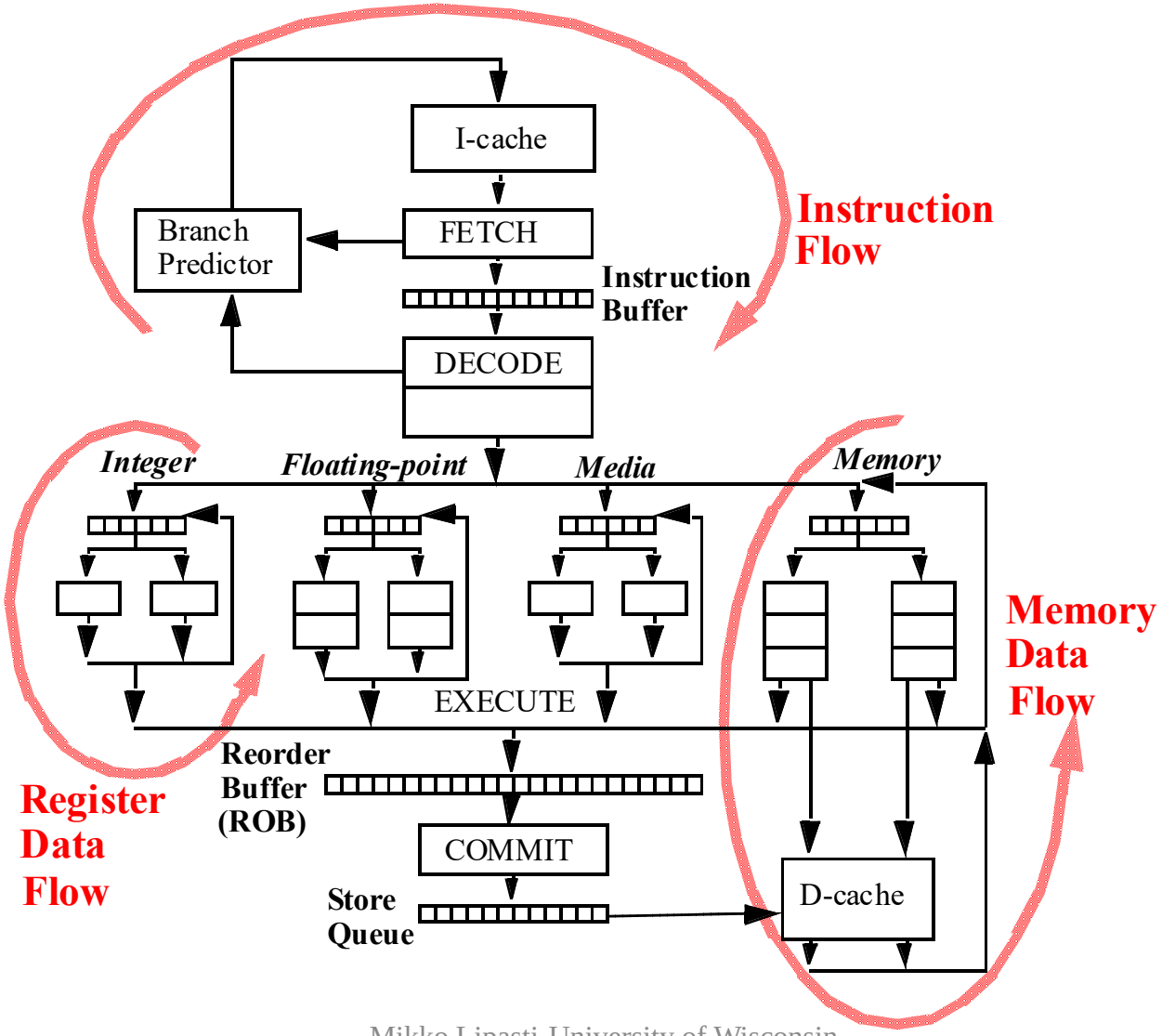
- Load queue records all speculative loads
- Bus writes/upgrades are checked against LQ
- Any matching load gets marked for replay
- At commit, loads are checked and replayed if necessary
  - Results in machine flush, since load-dependent ops must also replay
- Practically, conflicts are rare, so expensive flush is OK

Maintaining Precise State

- Out-of-order execution
  - ALU instructions
  - Load/store instructions
- In-order completion/retirement
  - Precise exceptions
- Solutions
  - Reorder buffer retires instructions in order
  - Store queue retires stores in order
  - Exceptions can be handled at any instruction boundary by reconstructing state out of ROB/SQ

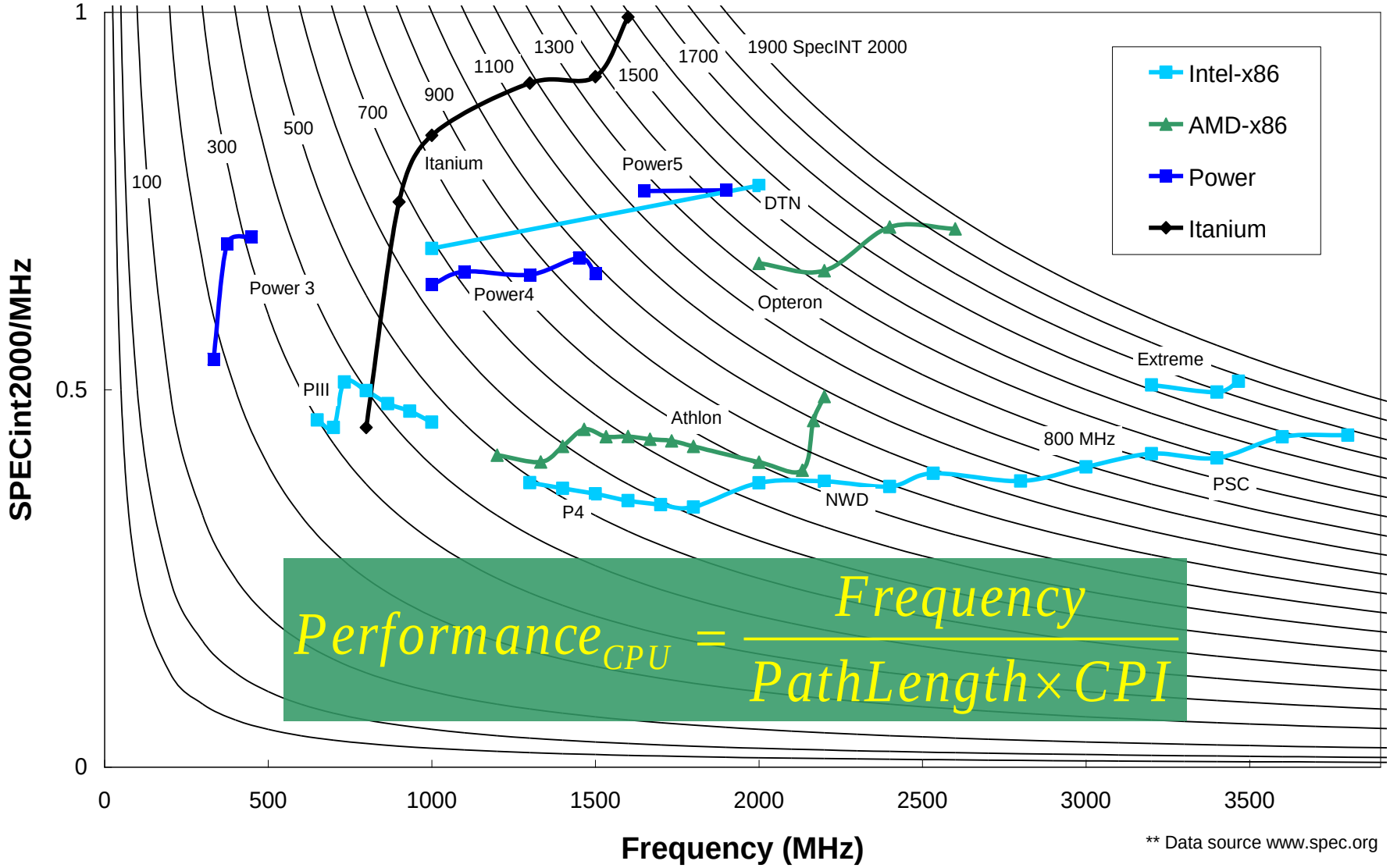


Summary: A High-IPC Processor



[John DeVale & Bryan Black, 2005]

Landscape of Microprocessor Families



** Data source www.spec.org

Review of 752

- ✓ Iron law
- ✓ Superscalar challenges
  - ✓ Instruction flow
  - ✓ Register data flow
  - ✓ Memory Dataflow
- ✓ Modern memory interface
- • What was not covered
  - – Memory hierarchy (review later)
  - – Virtual memory
  - – Power & reliability
  - – Many implementation/design details
  - – Etc.
  - – Multithreading (coming up later)