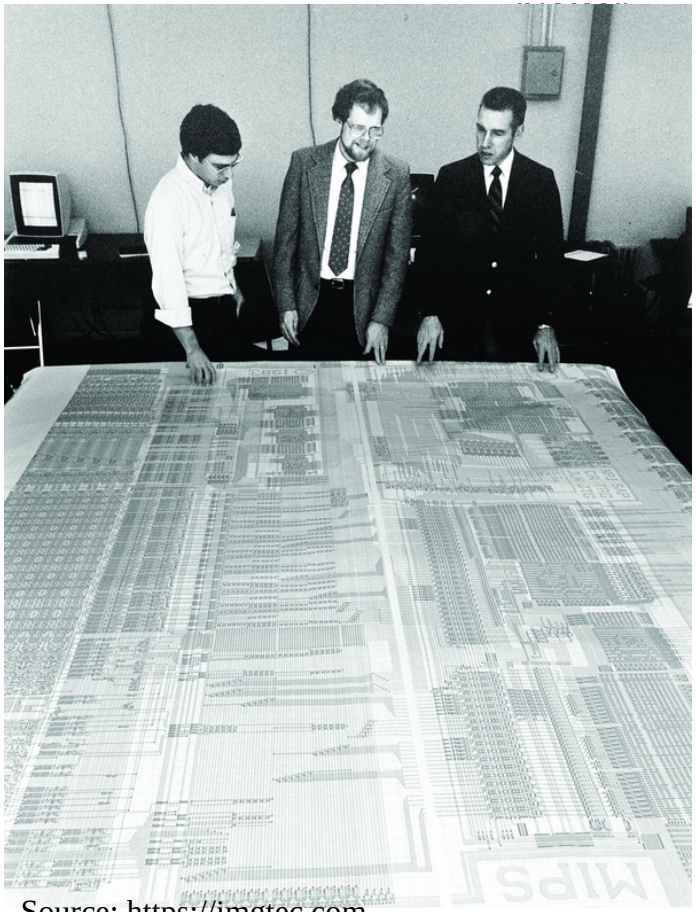
Superscalar Organization

ECE/CS 752 Fall 2017

Prof. Mikko H. Lipasti
University of Wisconsin-Madison

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

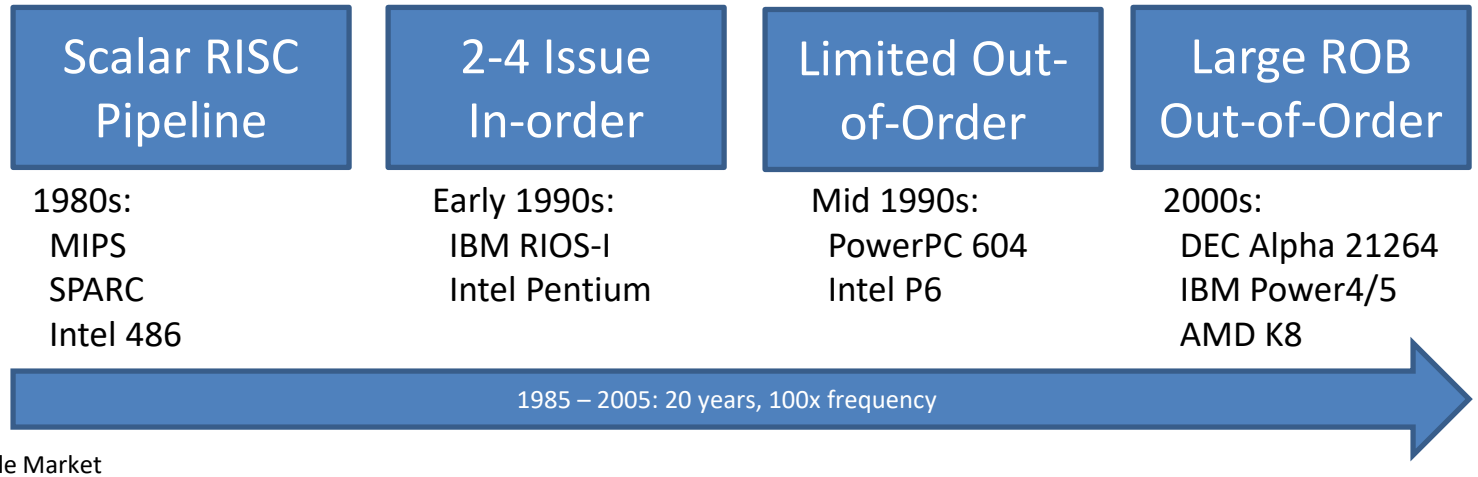
- Fetch/execute multiple instructions per cycle
- Decouple stages so stalls don't propagate
- Exploit instruction-level parallelism (ILP)

Limits on Instruction Level Parallelism (ILP)

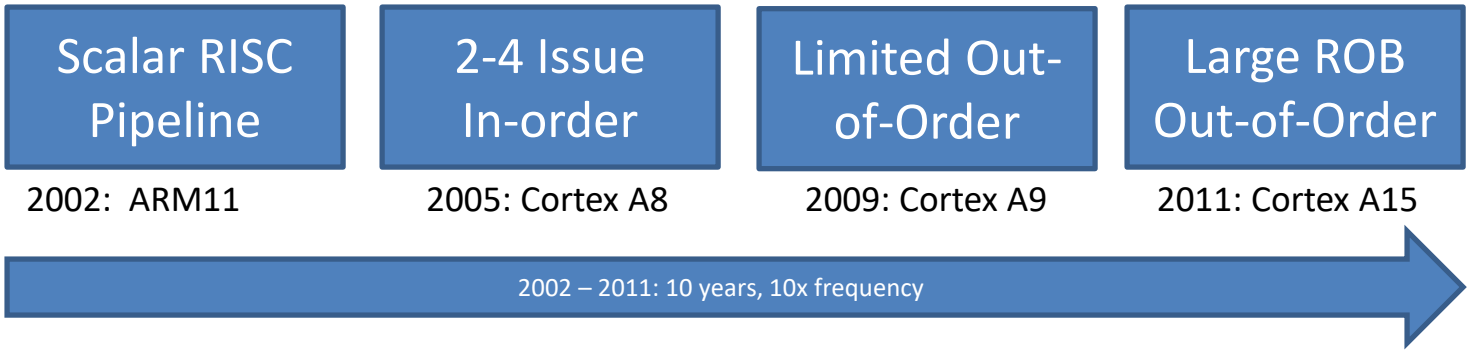
Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86 (Flynn's bottleneck)
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin 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

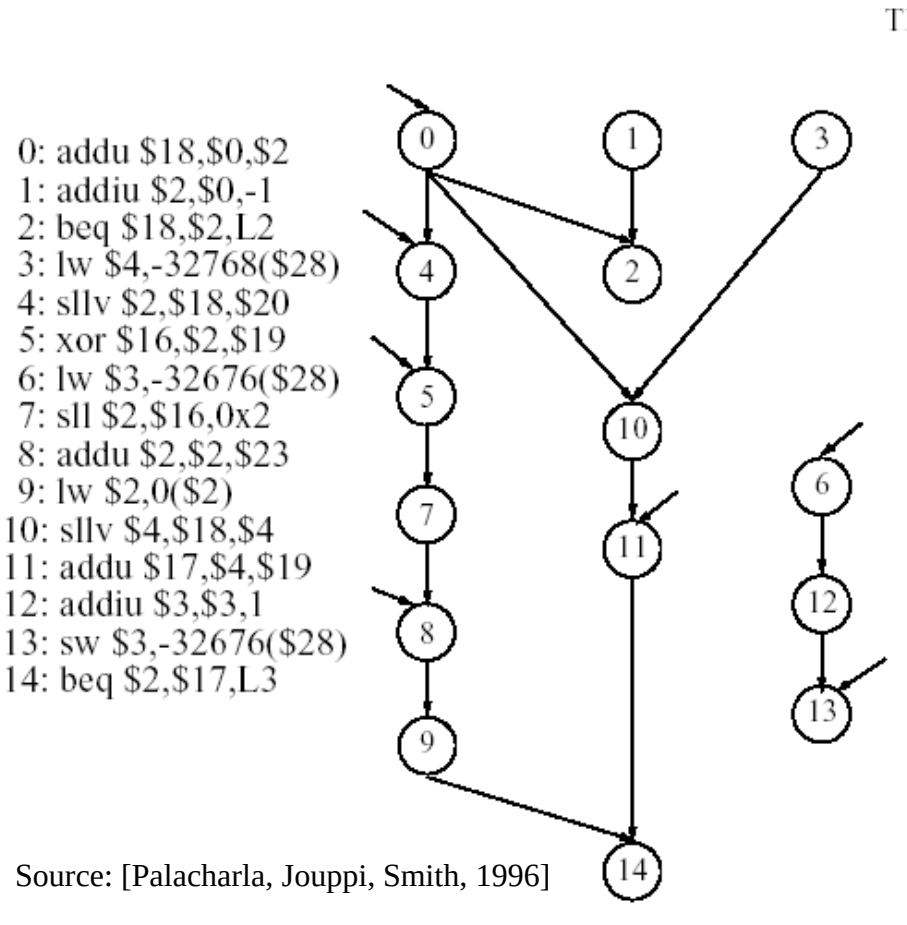
Desktop/Workstation Market



Mobile Market



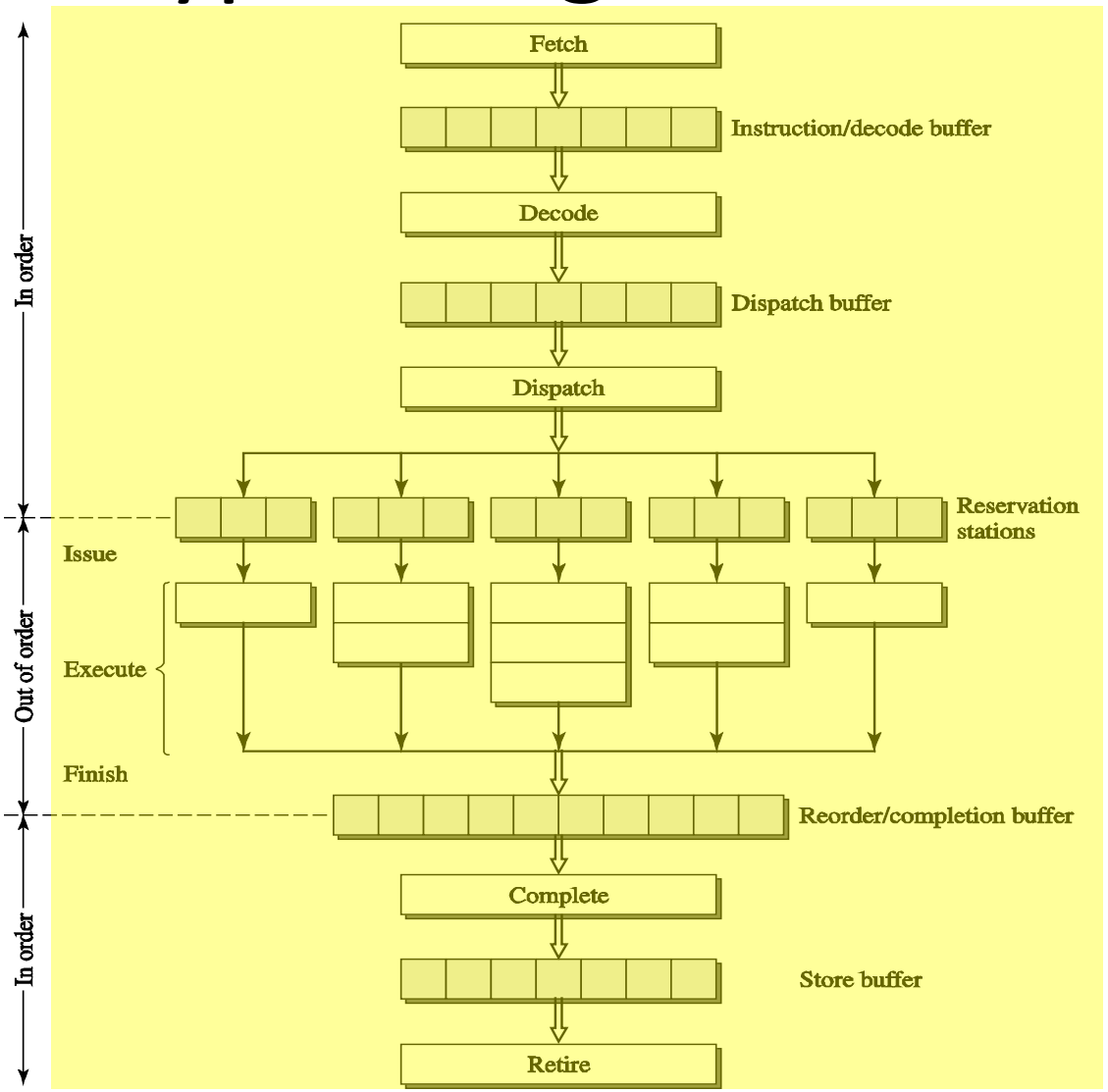
What Does a High-IPC CPU Do?



Source: [Palacharla, Jouppi, Smith, 1996]

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

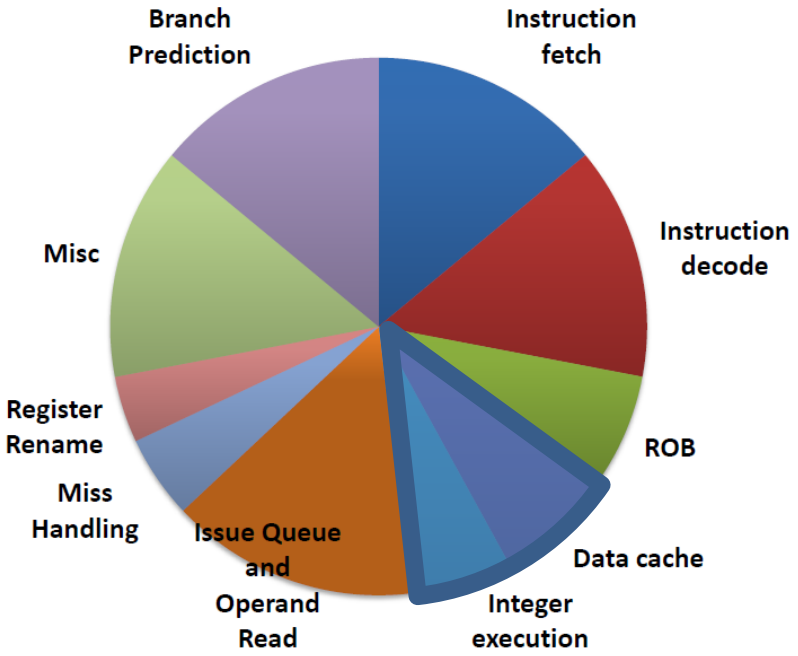
A Typical High-IPC Processor



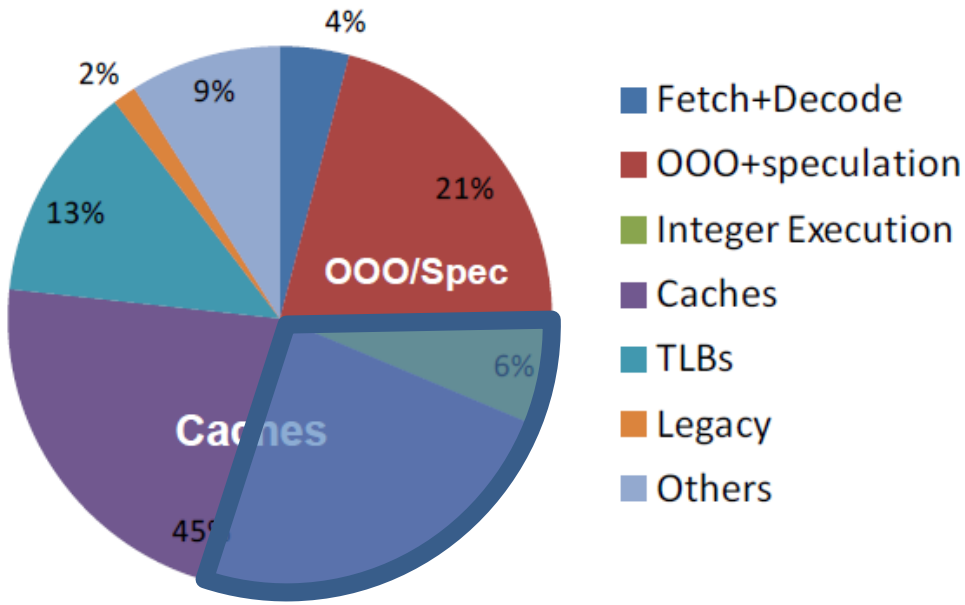
1. Fetch and Decode
2. Construct DDG
3. Evaluate DDG
4. Commit results

Power Consumption

ARM Cortex A15 [Source: NVIDIA]



Core i7 [Source: Intel]

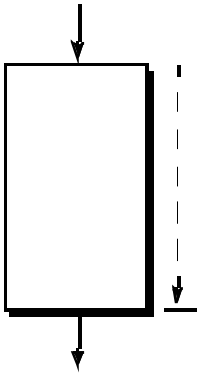


- Actual computation overwhelmed by overhead of aggressive execution pipeline

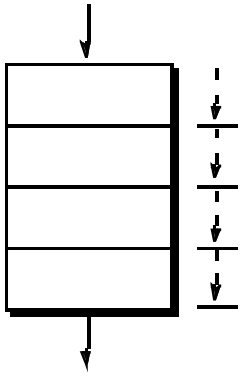
Limitations of Scalar Pipelines

- Scalar upper bound on throughput
  - IPC <= 1 or CPI >= 1
- Inefficient unified pipeline
  - Long latency for each instruction
- Rigid pipeline stall policy
  - One stalled instruction stalls all newer instructions

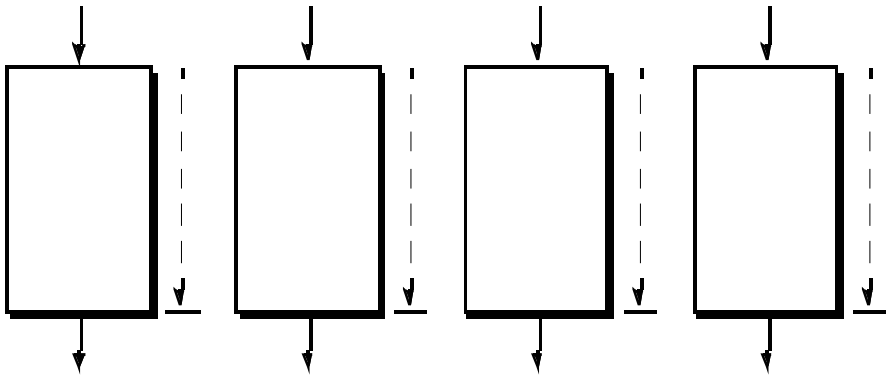
Parallel Pipelines



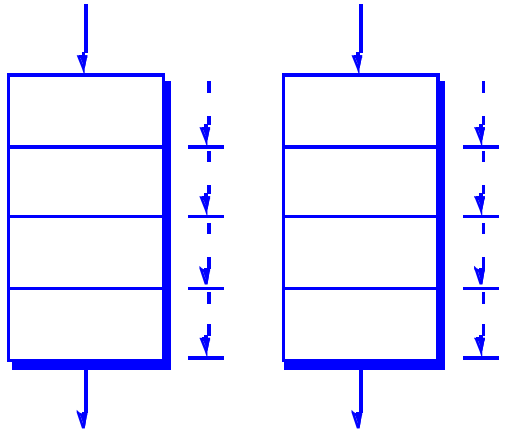
(a) No Parallelism



(b) Temporal Parallelism

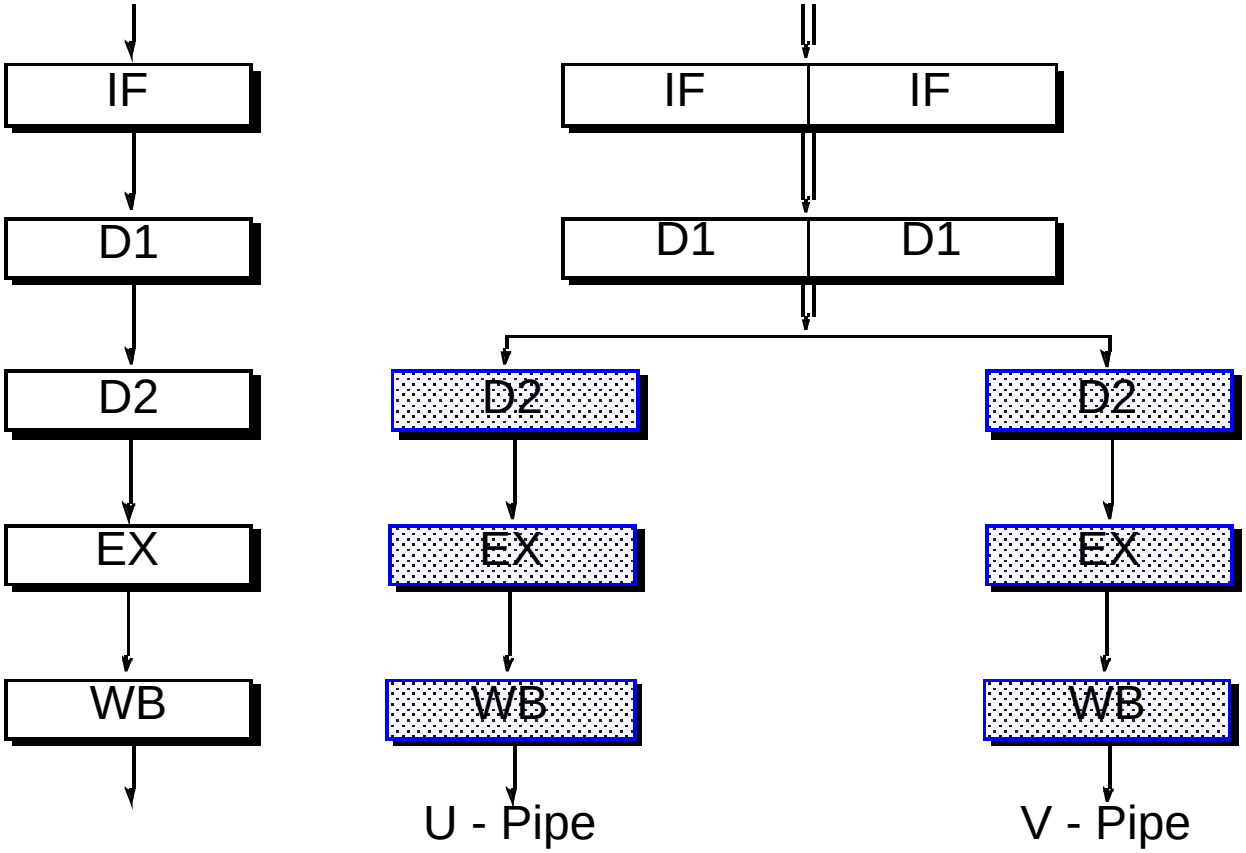


(c) Spatial Parallelism

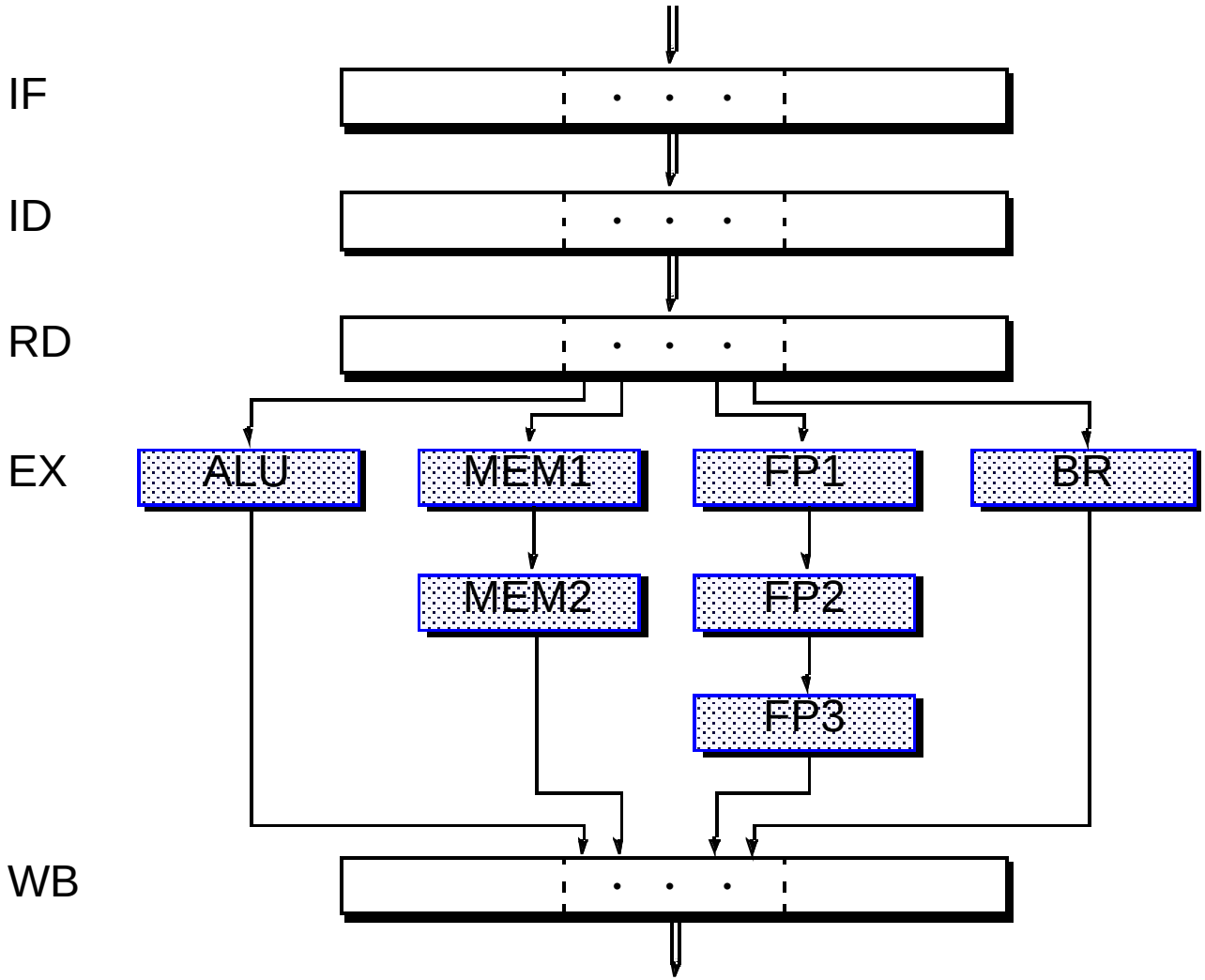


(d) Parallel Pipeline

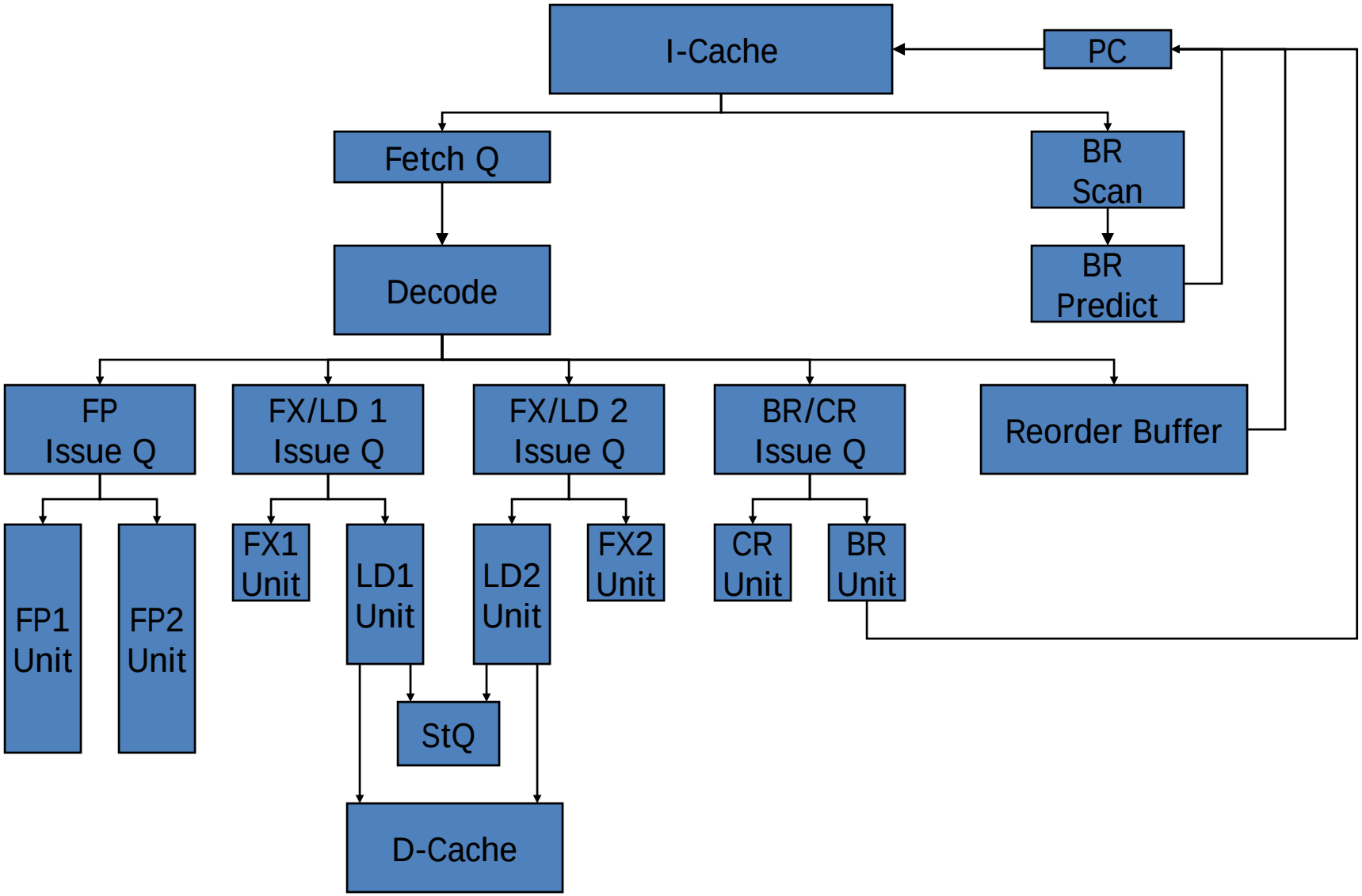
Intel Pentium Parallel Pipeline



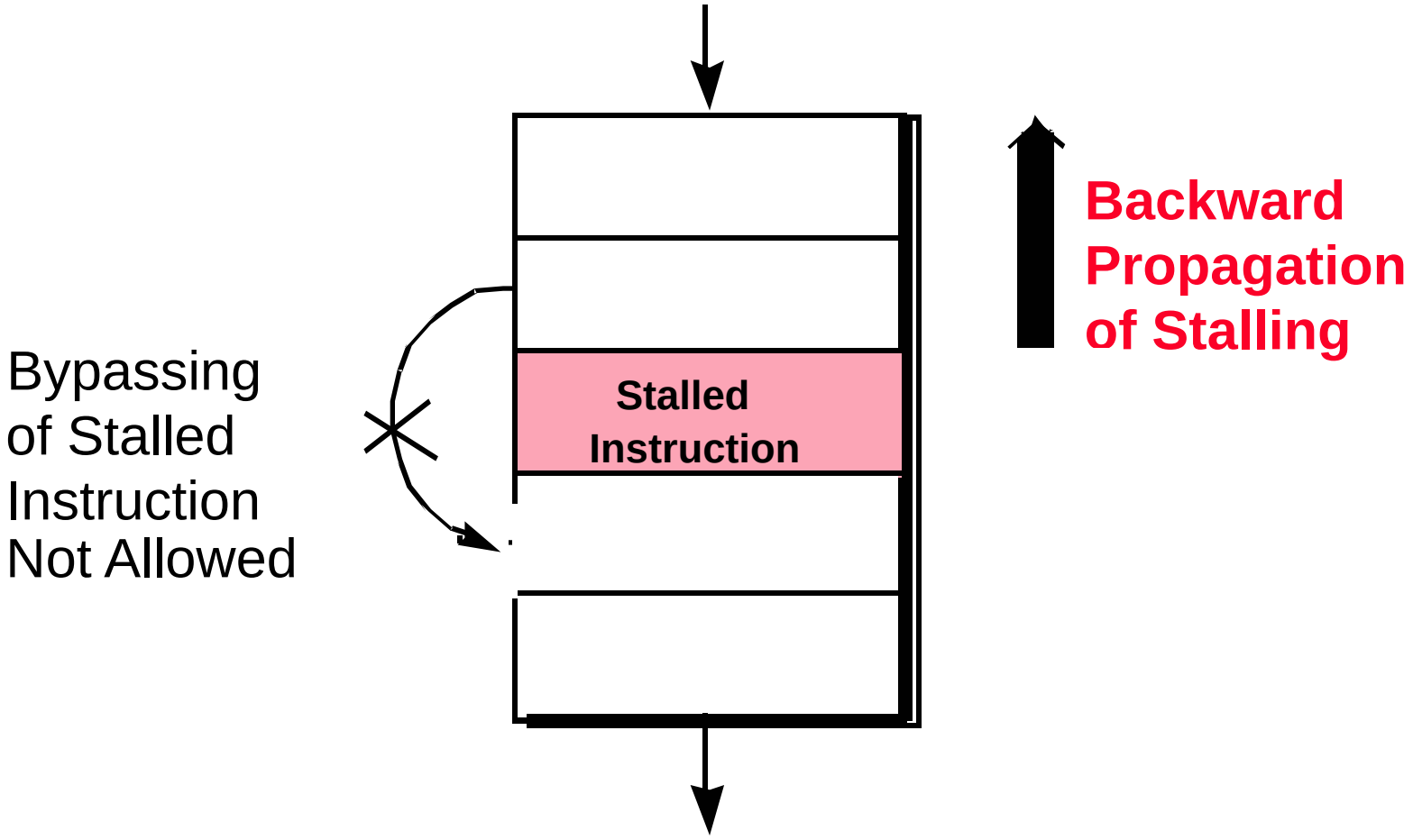
Diversified Pipelines



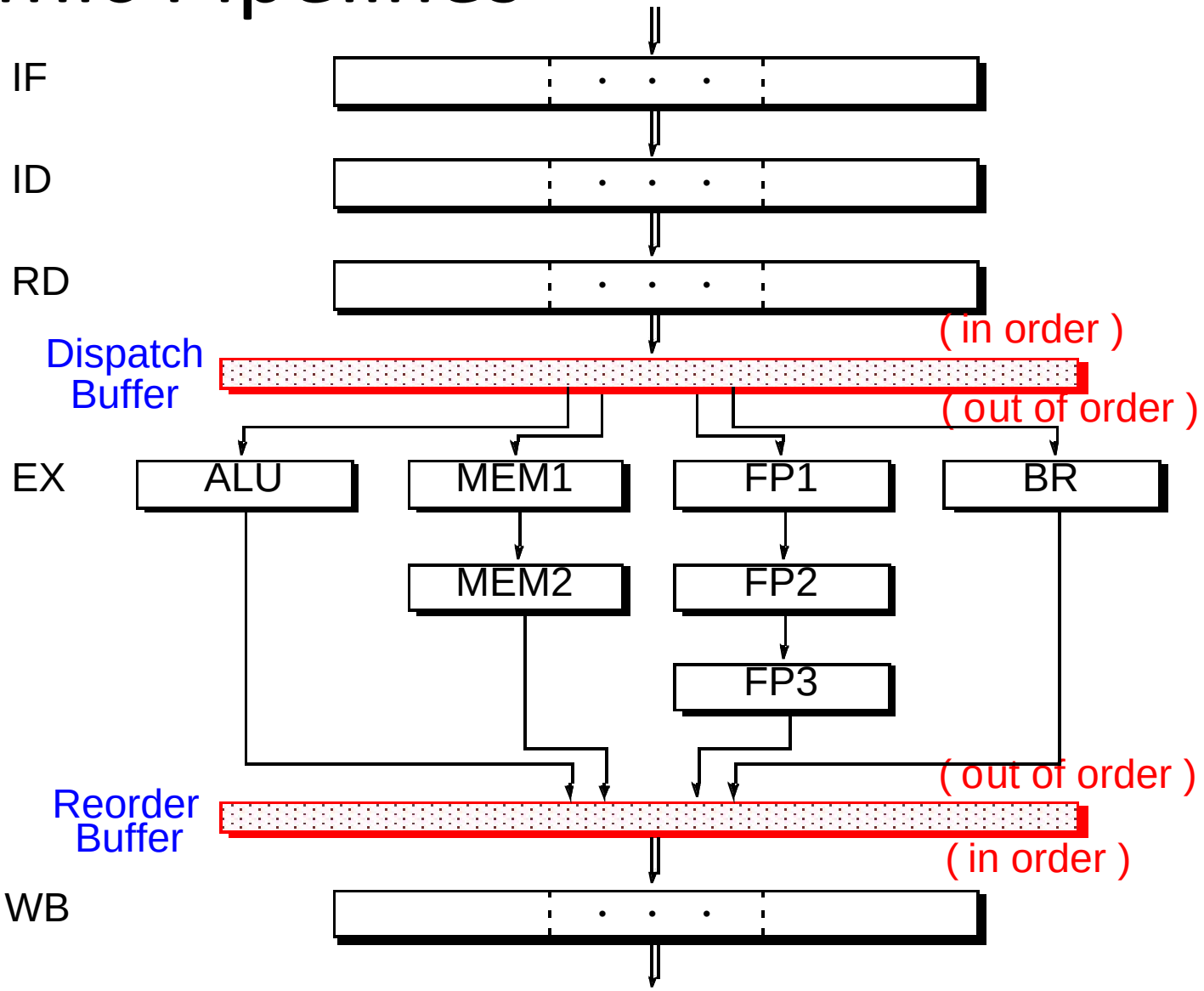
Power4 Diversified Pipelines



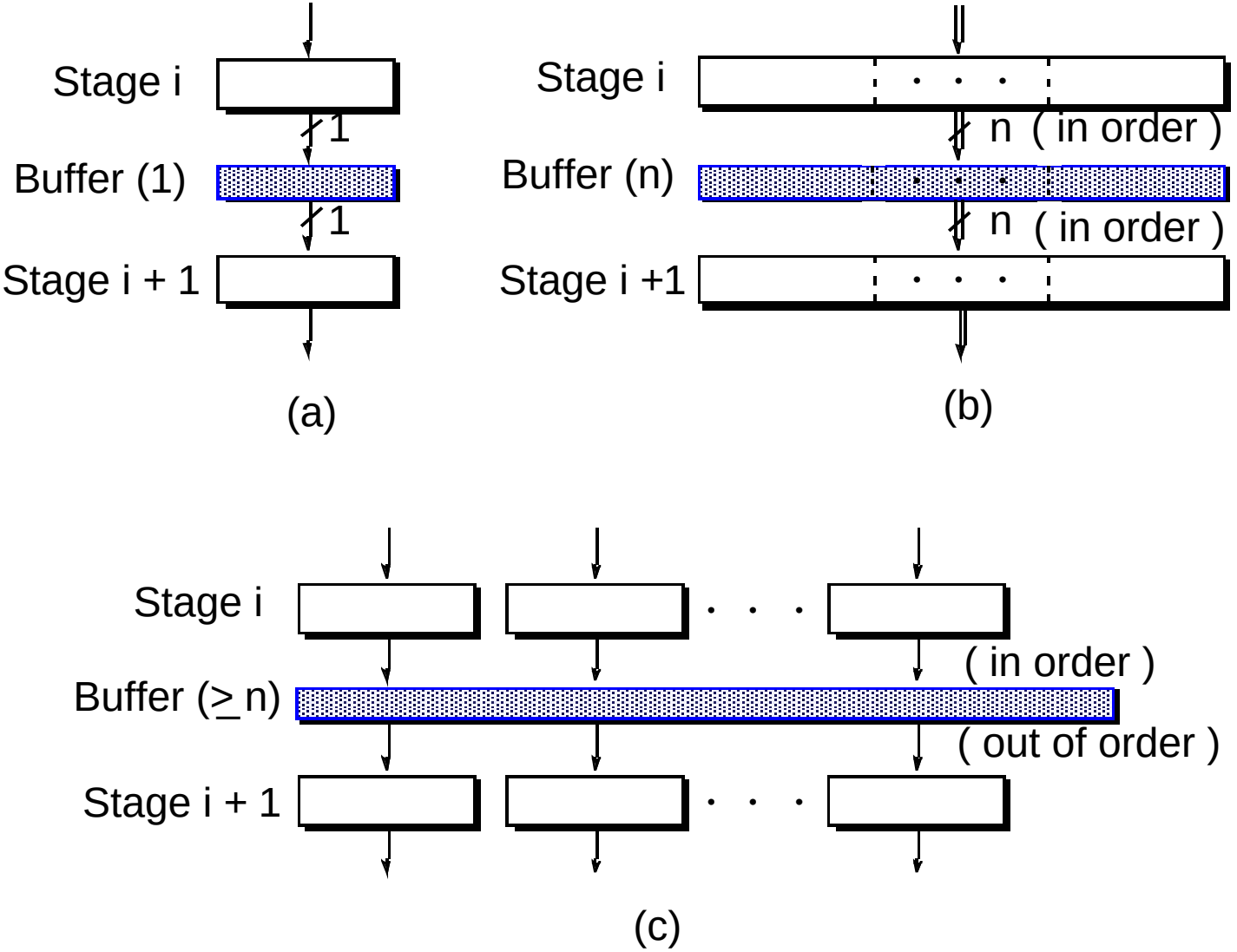
Rigid Pipeline Stall Policy



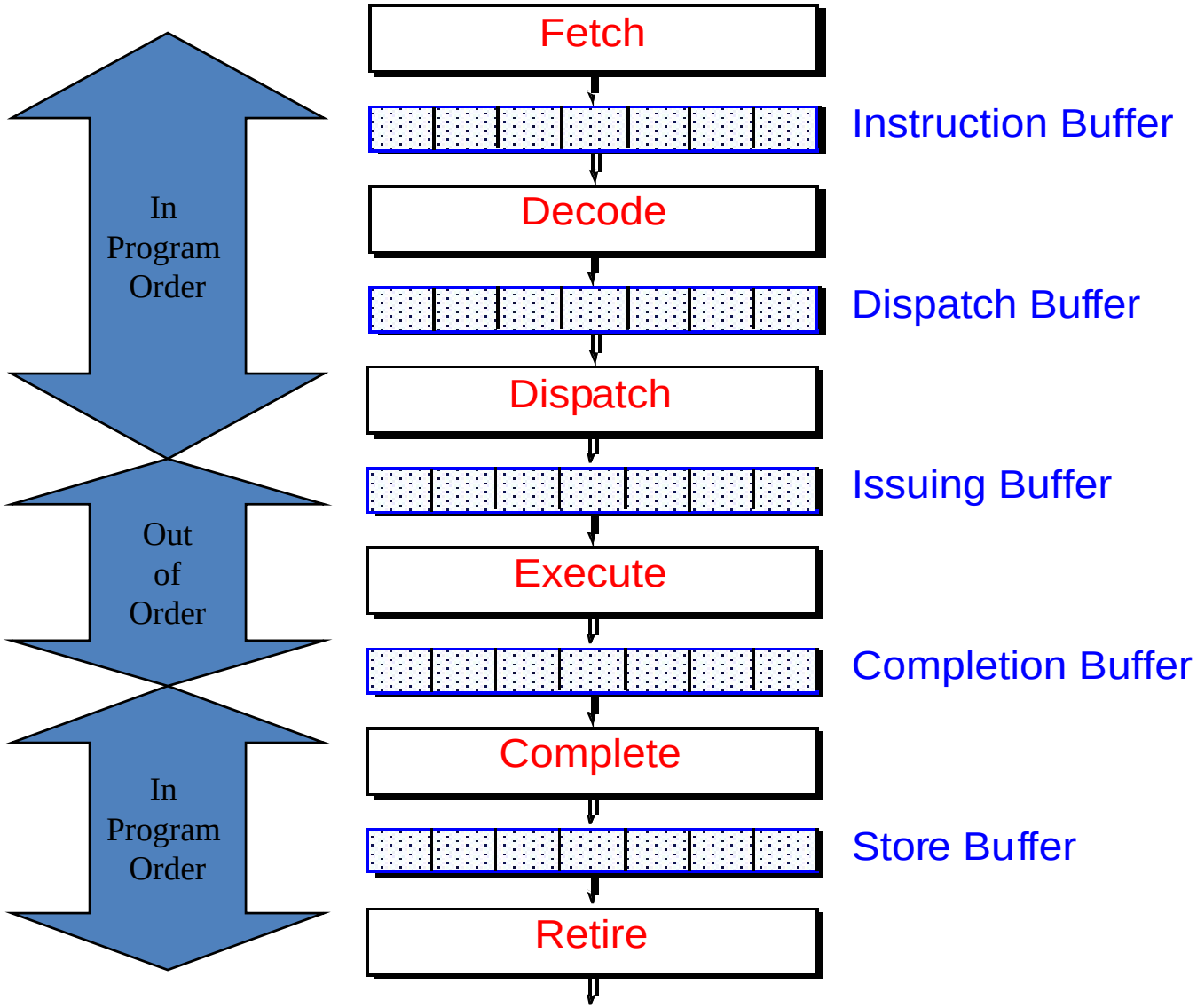
Dynamic Pipelines



Interstage Buffers



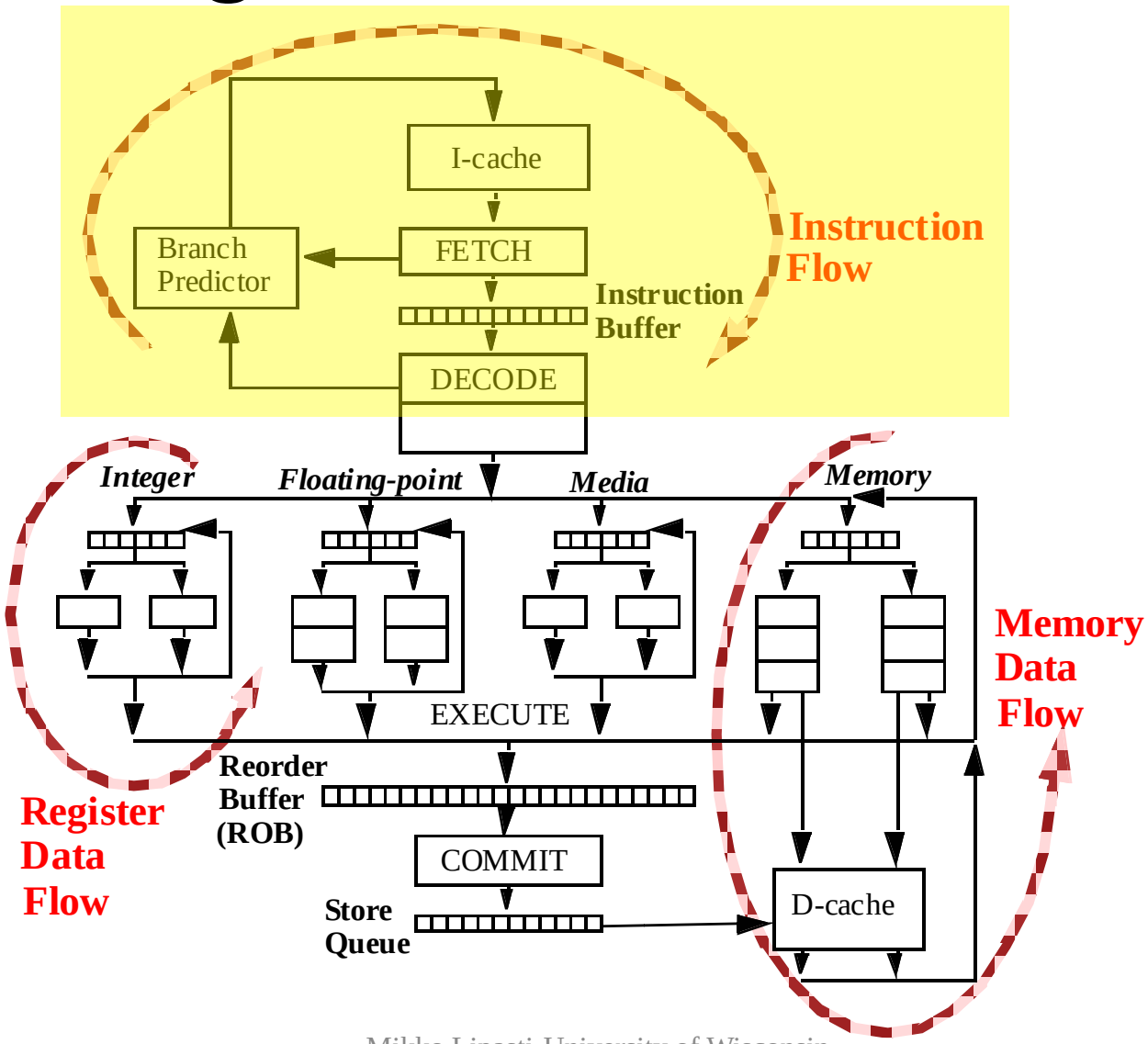
Superscalar Pipeline Stages



Limitations of Scalar Pipelines

- Scalar upper bound on throughput
  - IPC <= 1 or CPI >= 1
  - Solution: wide (superscalar) pipeline
- Inefficient unified pipeline
  - Long latency for each instruction
  - Solution: diversified, specialized pipelines
- Rigid pipeline stall policy
  - One stalled instruction stalls all newer instructions
  - Solution: Out-of-order execution, distributed execution pipelines

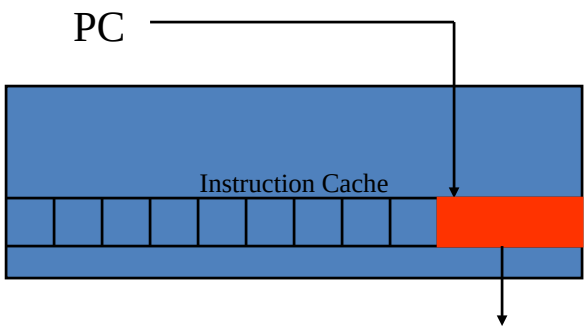
High-IPC Processor



Instruction Flow

Objective: Fetch multiple instructions per cycle

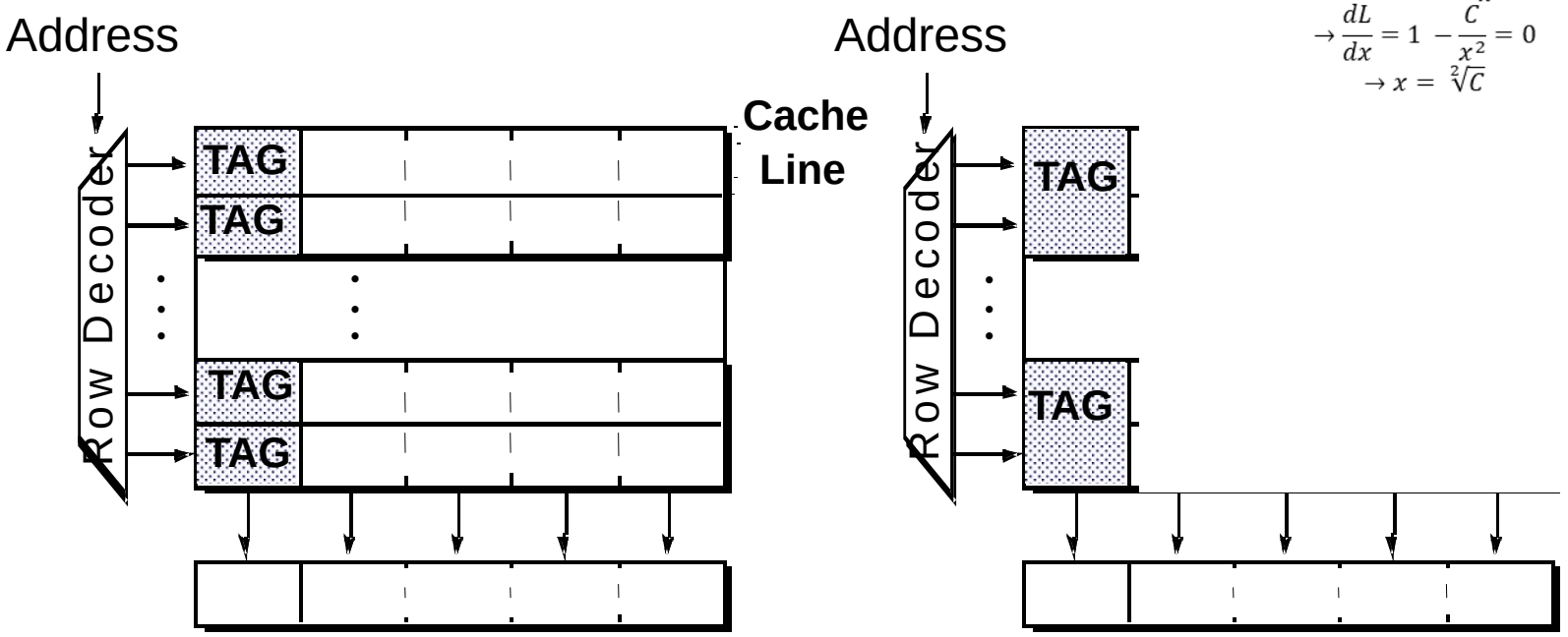
- Challenges:
  - Branches: unpredictable
  - Branch targets misaligned
  - Instruction cache misses
- Solutions
  - Prediction and speculation
  - High-bandwidth fetch logic
  - Nonblocking cache and prefetching



only 3 instructions fetched

I-Cache Organization

$$Capacity\ C = x \times y$$
$$Latency\ L = x + y$$
$$\rightarrow L = x + \frac{C}{x}$$
$$\rightarrow \frac{dL}{dx} = 1 - \frac{C}{x^2} = 0$$
$$\rightarrow x = \sqrt[2]{C}$$

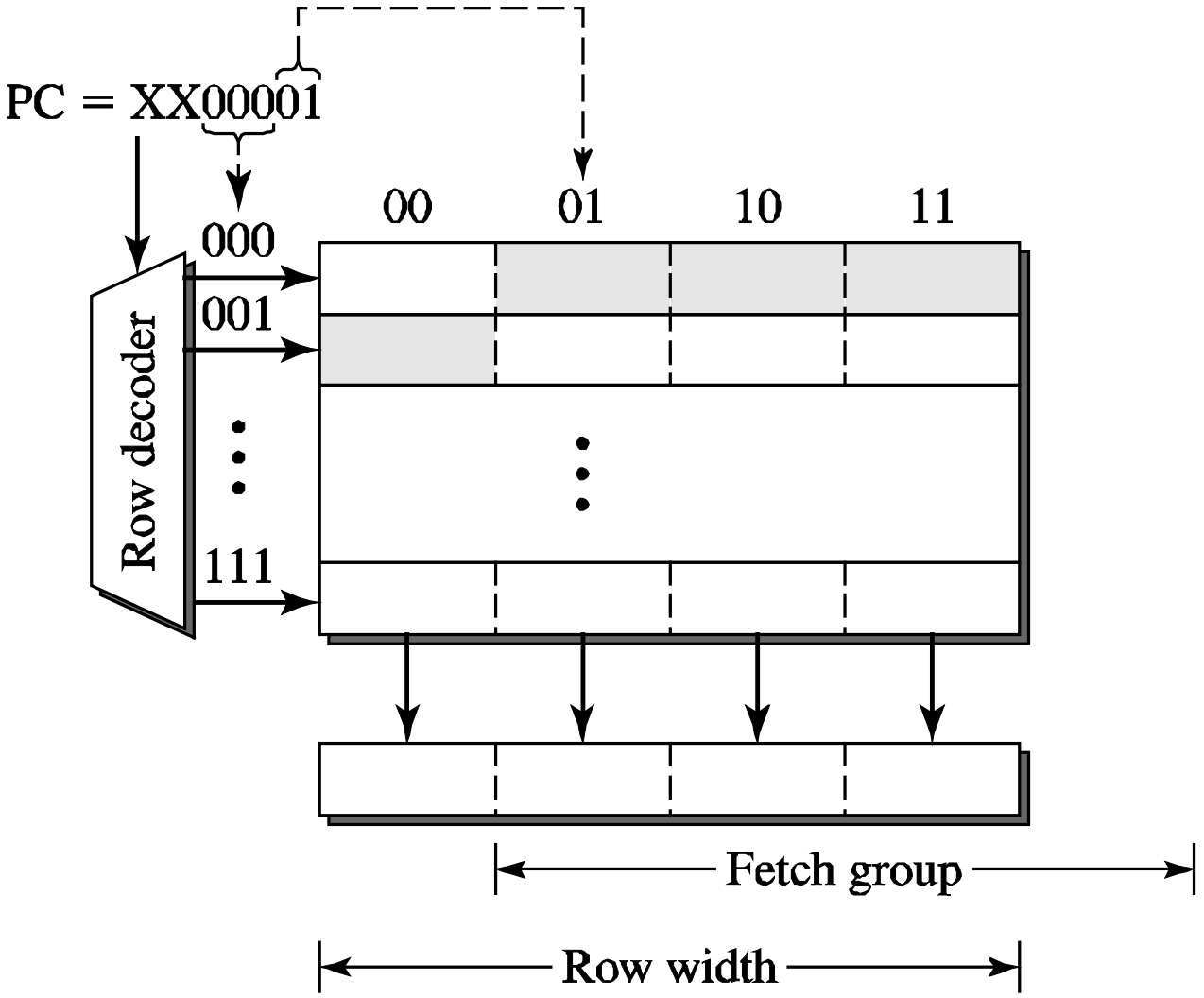


1 cache line = 1 physical row

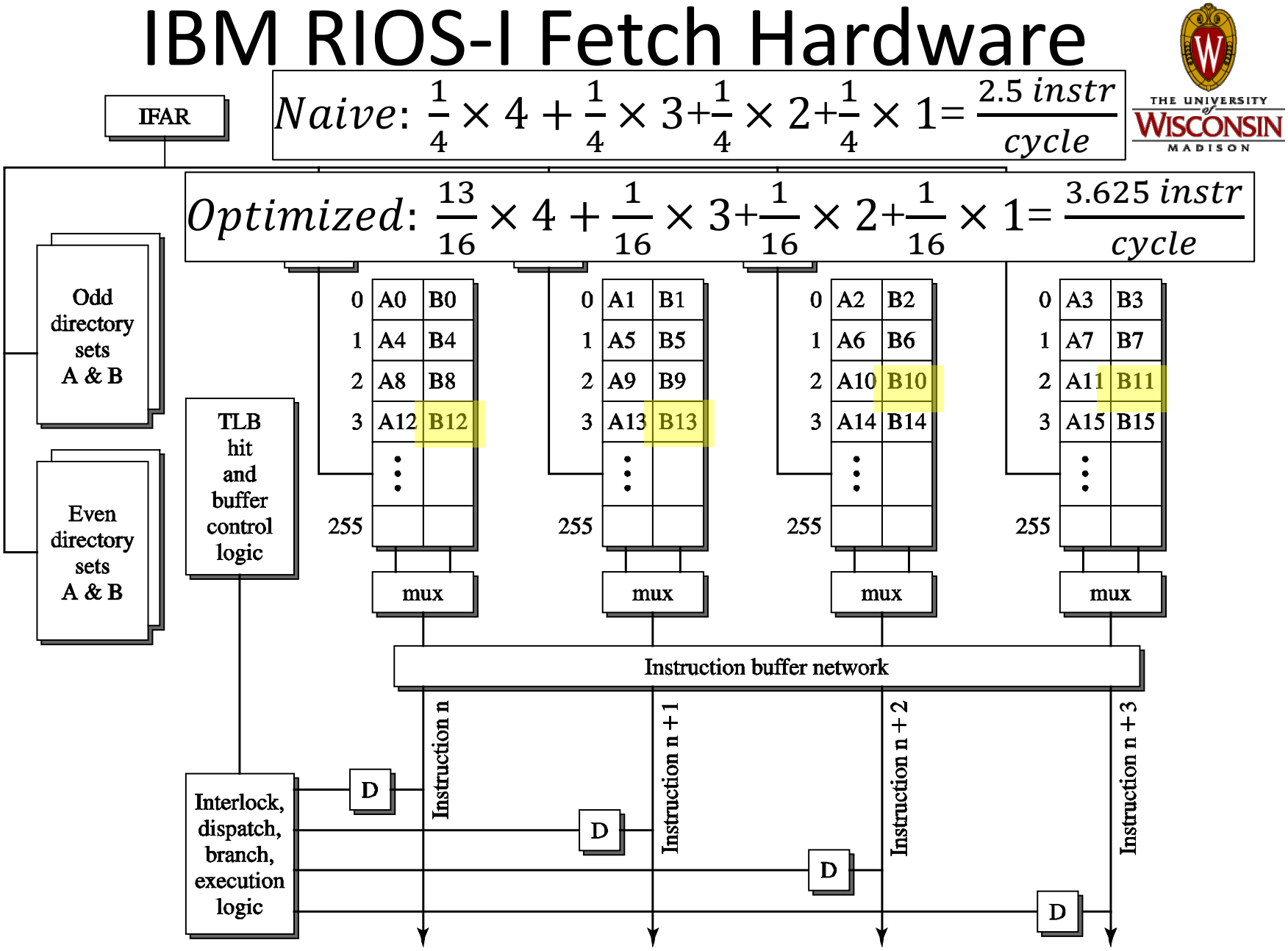
1 cache line = 2 physical rows

SRAM arrays need to be square to minimize delay

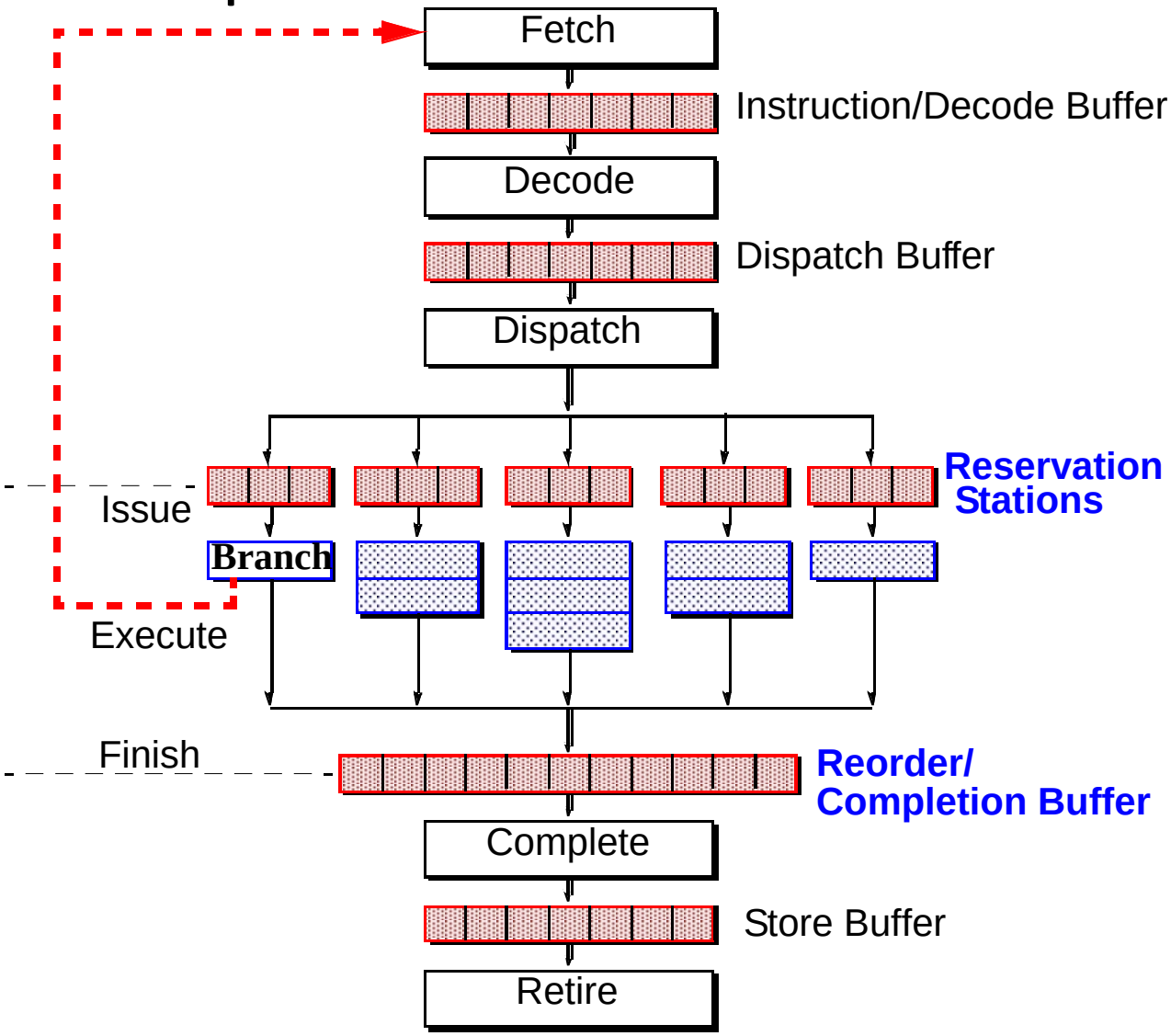
Fetch Alignment



IBM RIOS-I Fetch Hardware



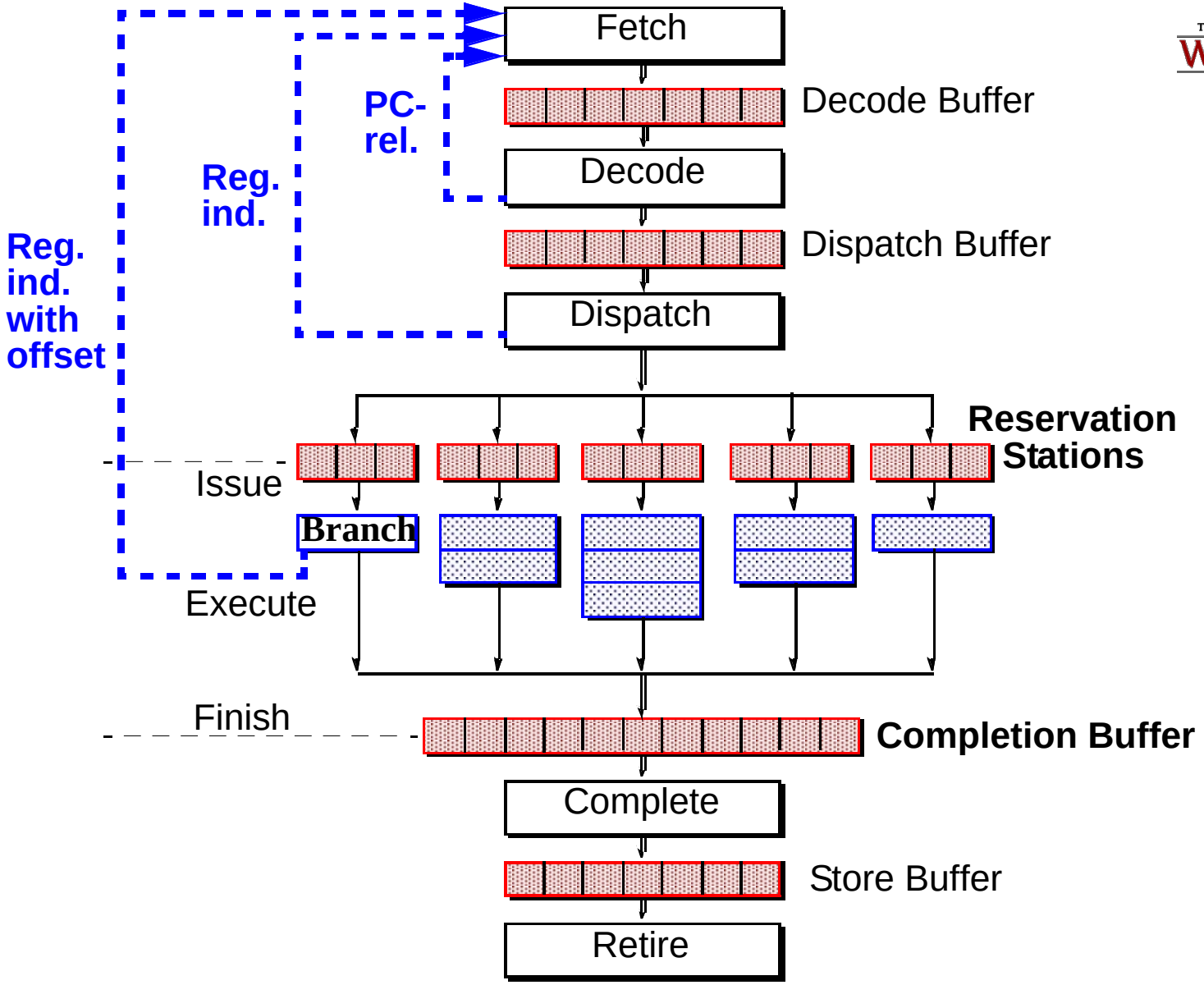
Disruption of Instruction Flow



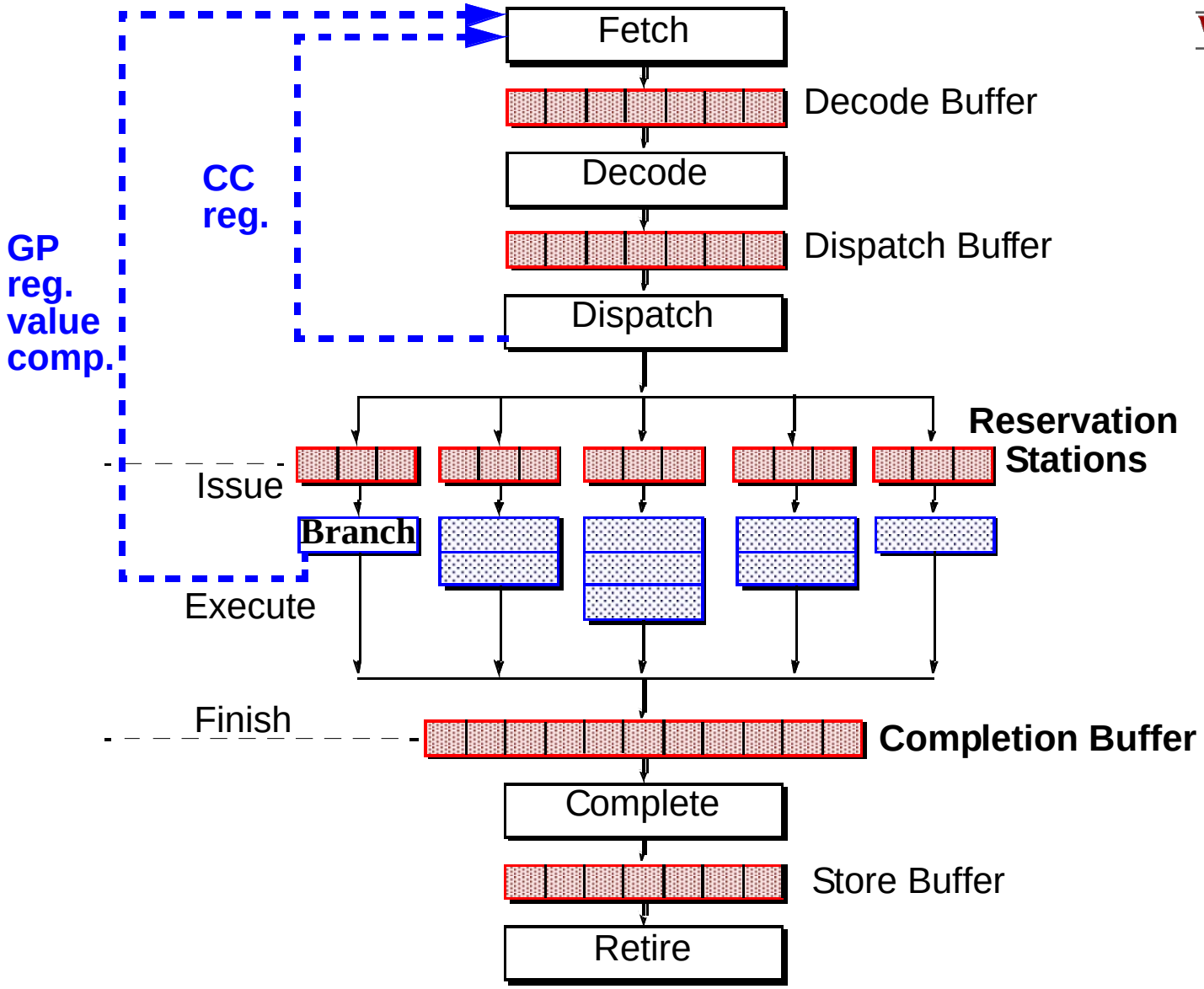
Branch Prediction

- 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)

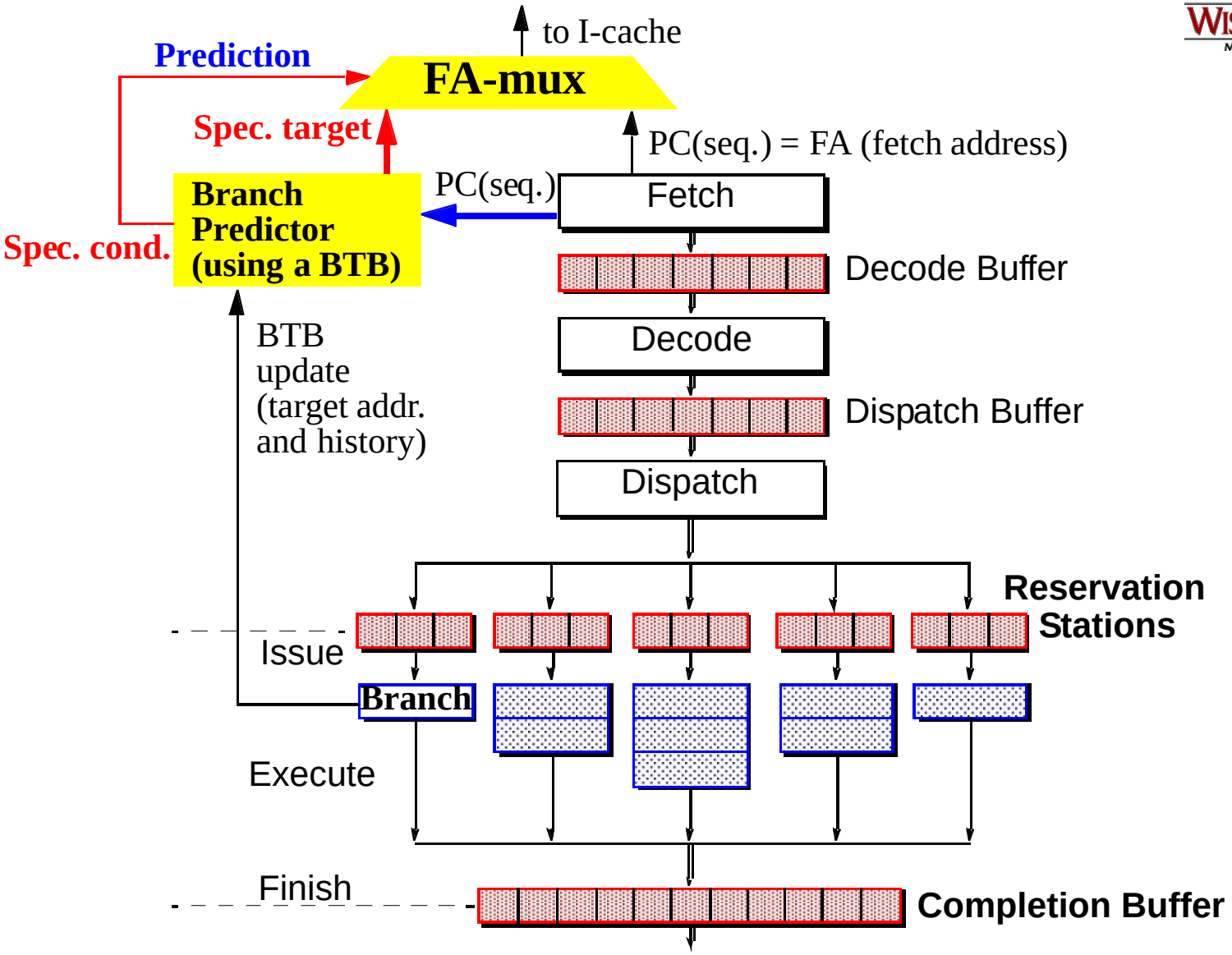
Target Address Generation



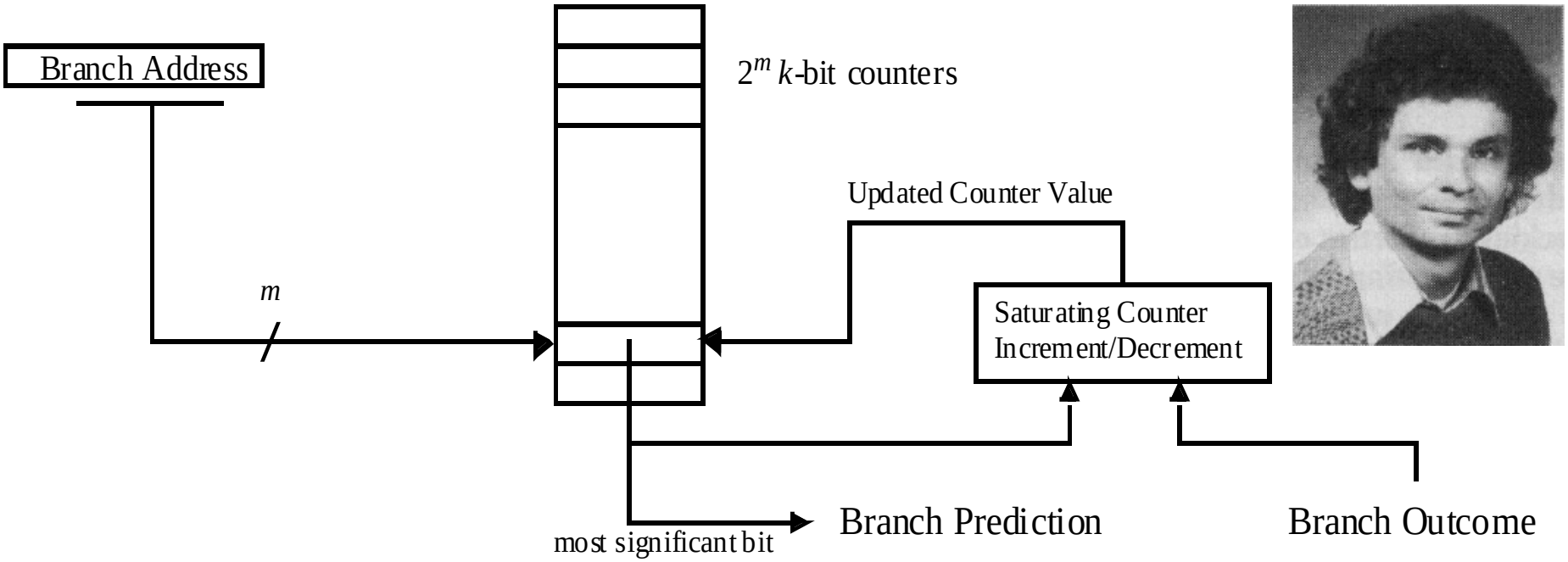
Branch Condition Resolution



Branch Instruction Speculation

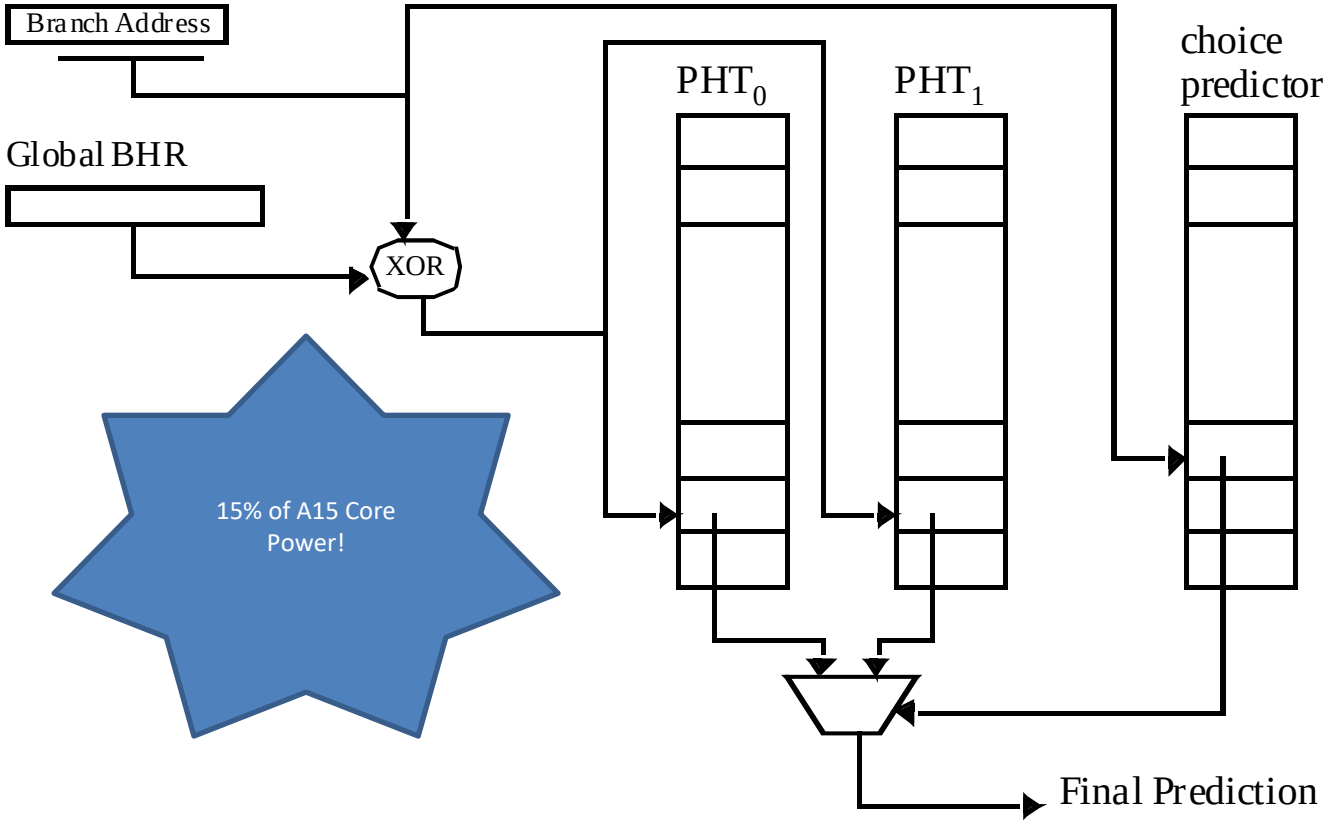


Hardware Smith Predictor



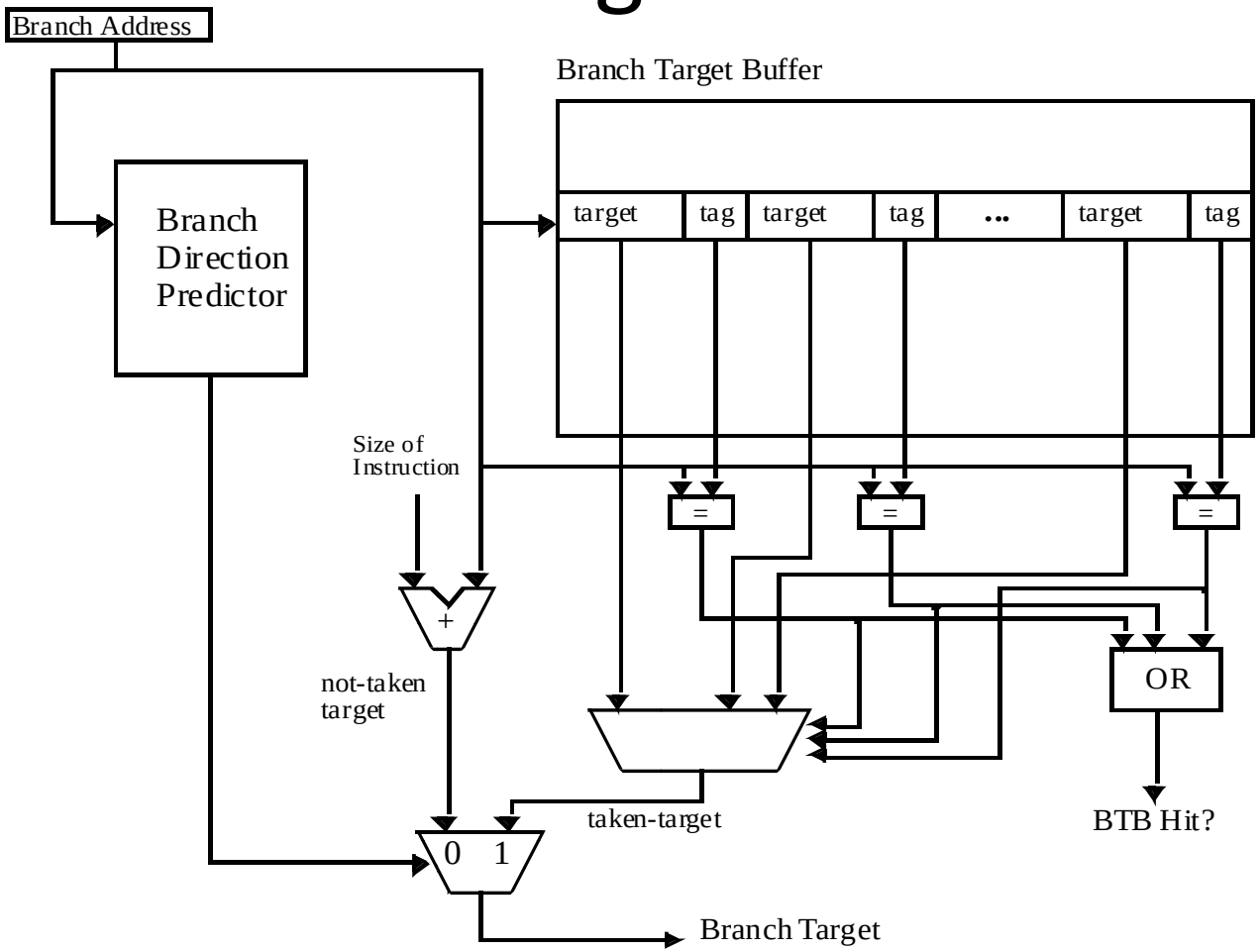
- Jim E. Smith. A Study of Branch Prediction Strategies. International Symposium on Computer Architecture, pages 135-148, May 1981
- Widely employed: Intel Pentium, PowerPC 604, MIPS R10000, etc.

Cortex A15: Bi-Mode Predictor



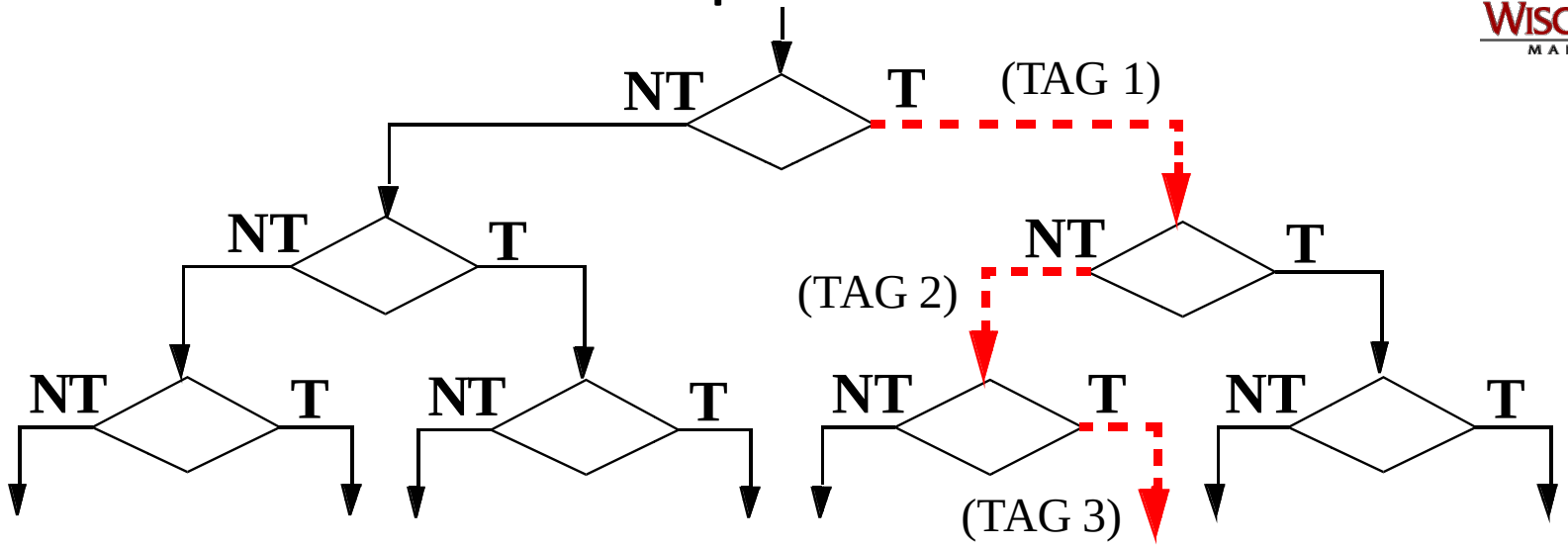
- PHT partitioned into T/NT halves
  - Selector chooses source
- Reduces negative interference, since most entries in PHT_0 tend towards NT, and most entries in PHT_1 tend towards T

Branch Target Prediction



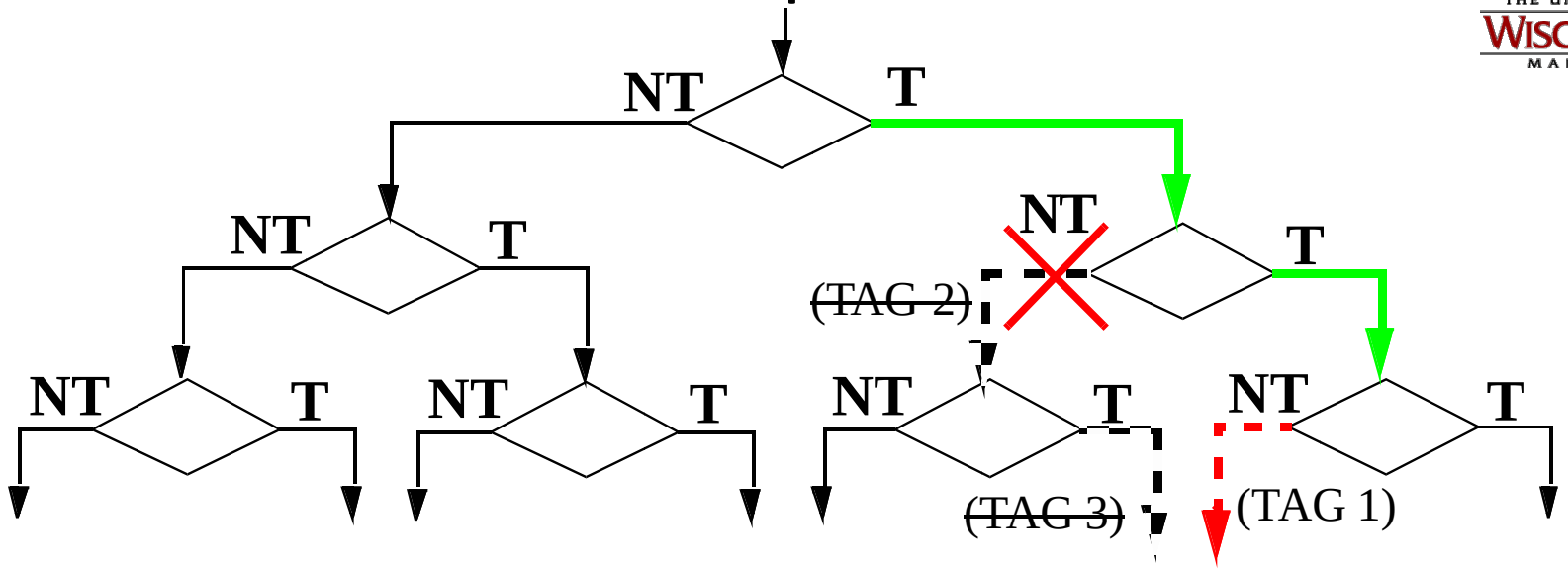
- Does not work well for function/procedure returns
- Does not work well for virtual functions, switch statements

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

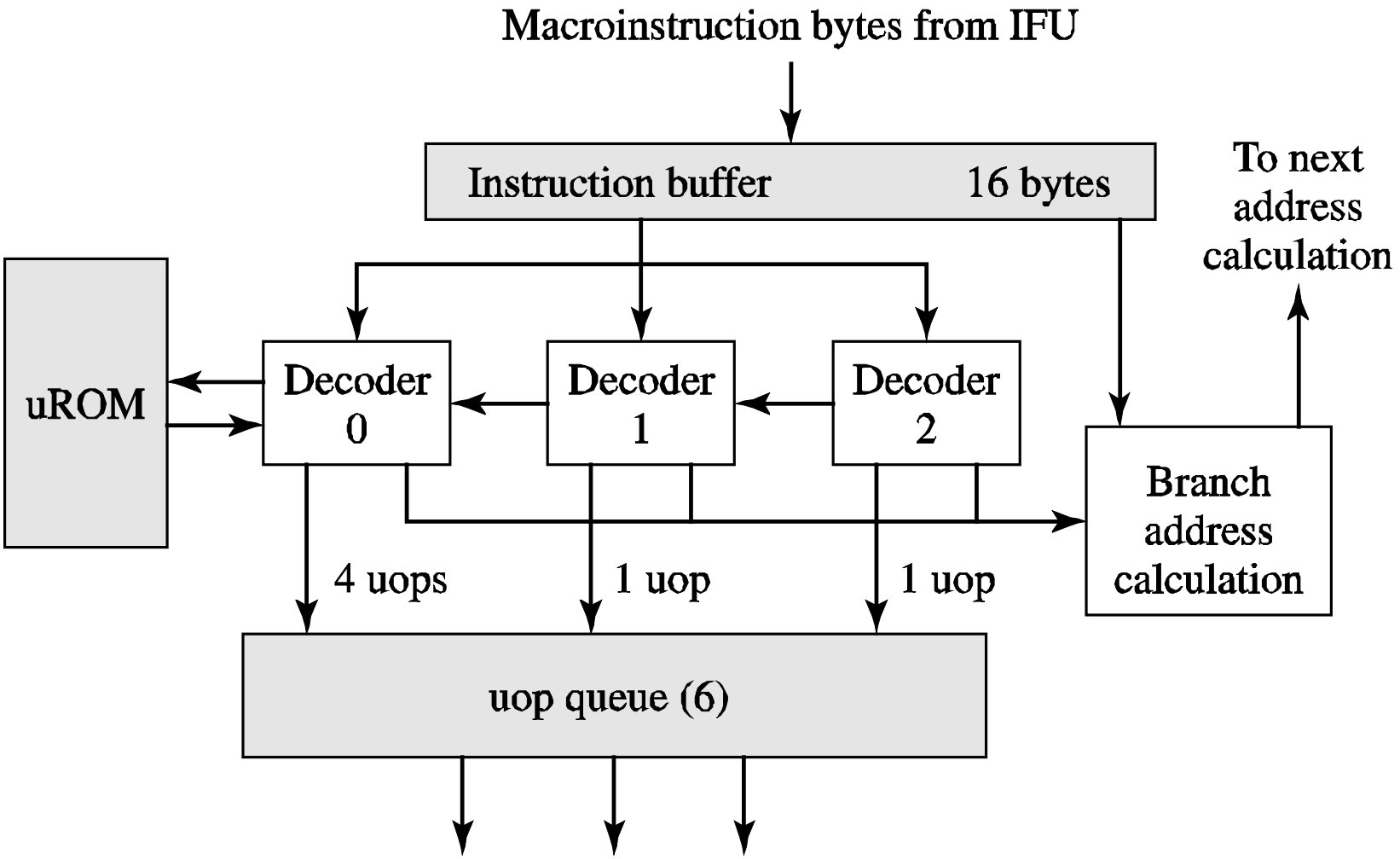
Mis-speculation Recovery

- 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

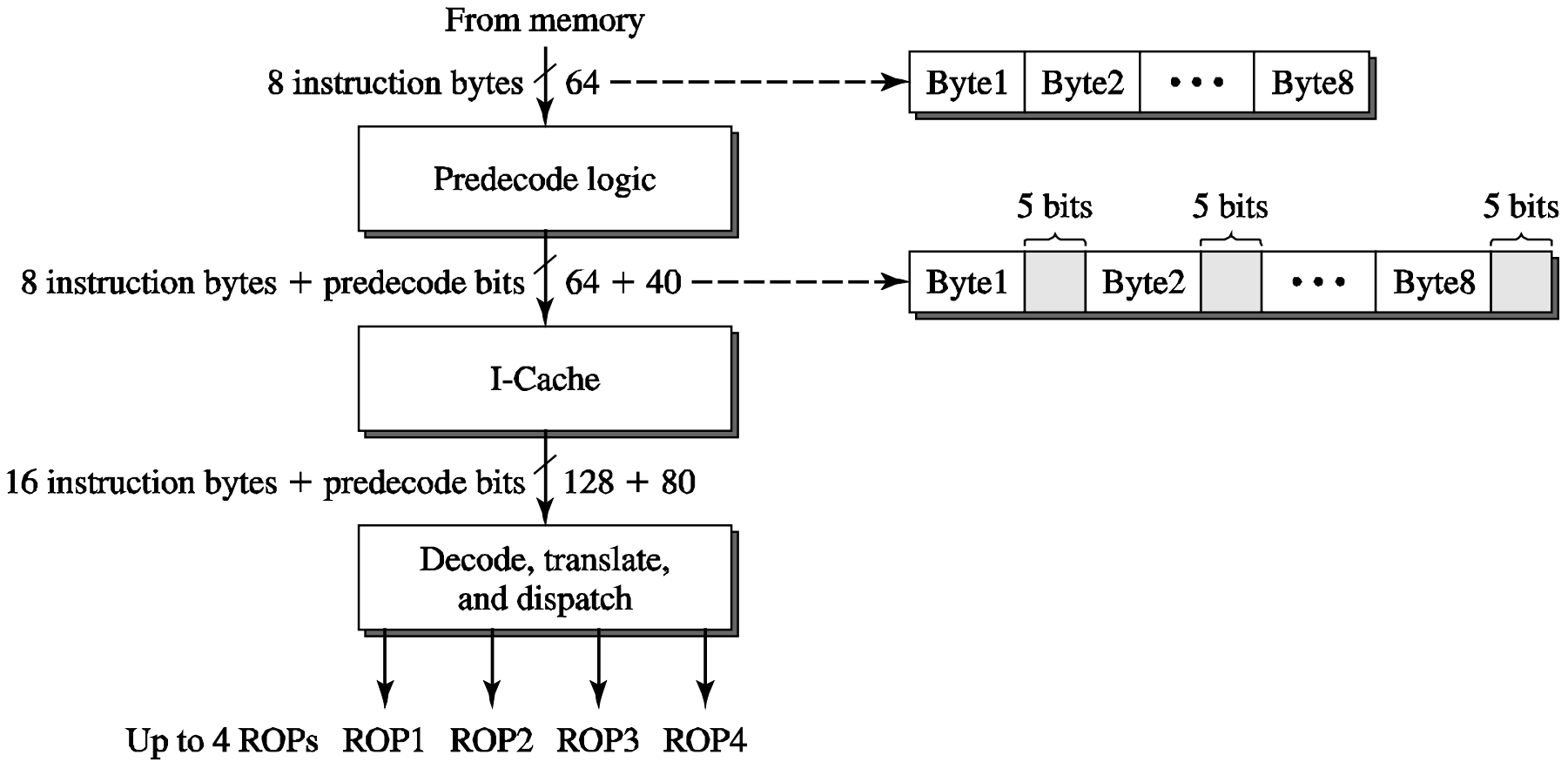
Parallel Decode

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

Intel P6 Fetch/Decode

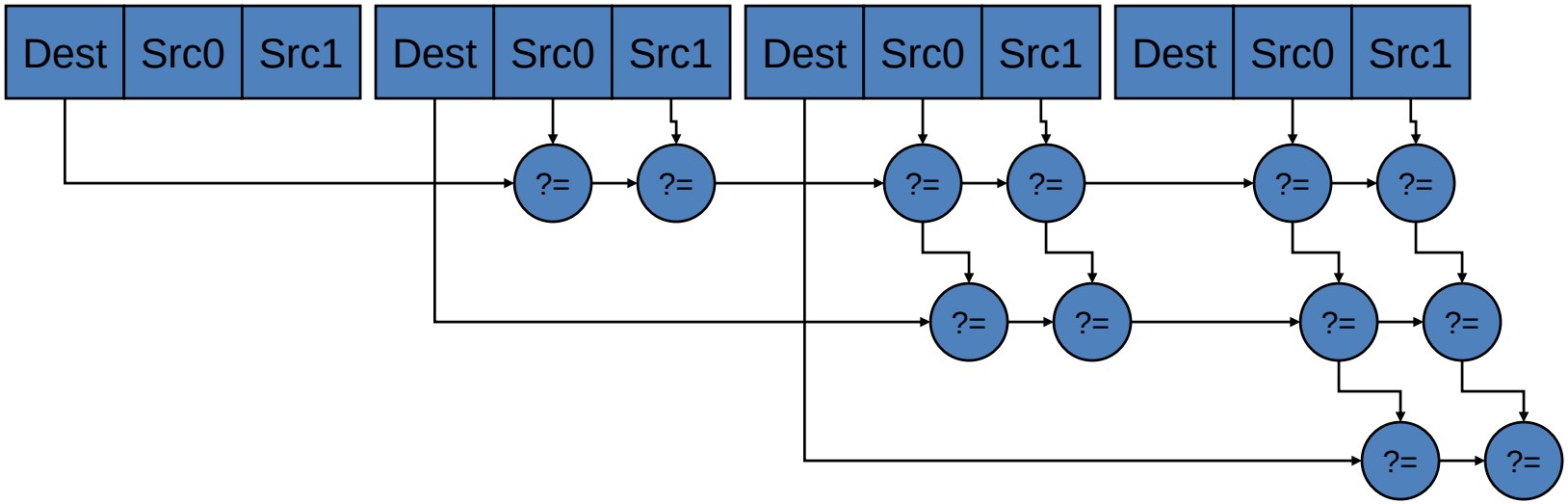


Predecoding in the AMD K5



- Now commonly employed in loop buffers, decoded instruction caches (uop caches)

Dependence Checking

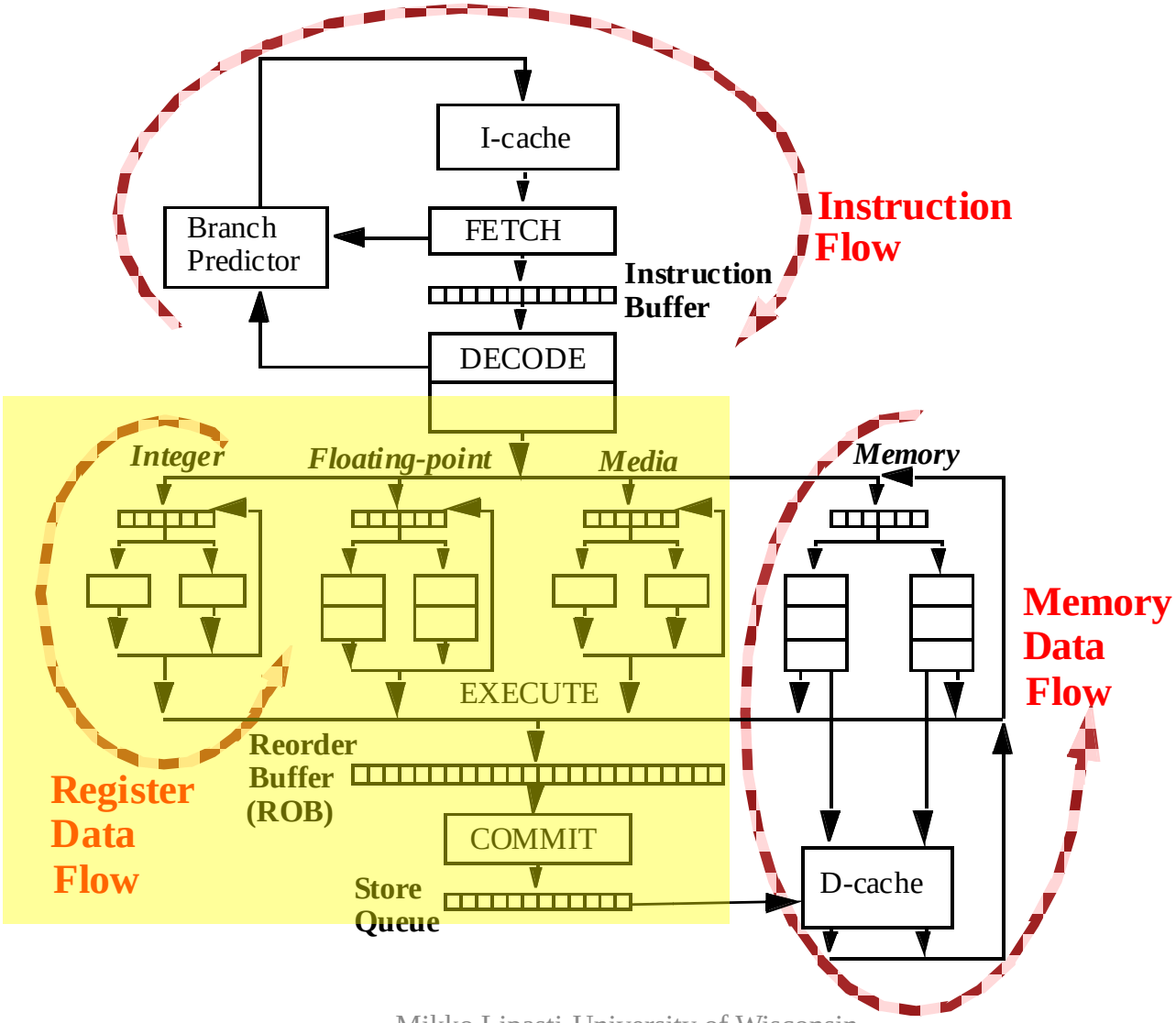


- Trailing instructions in fetch group
  - Check for dependence on leading instructions

Summary: Instruction Flow

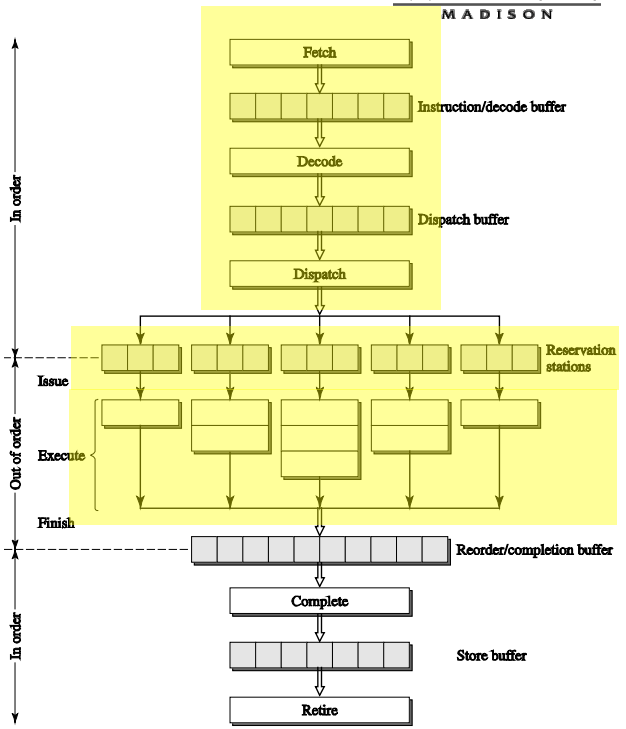
- Fetch group alignment
- Target address generation
  - Branch target buffer
- Branch condition prediction
- Speculative execution
  - Tagging/tracking instructions
  - Recovering from mispredicted branches
- Decoding in parallel

High-IPC Processor

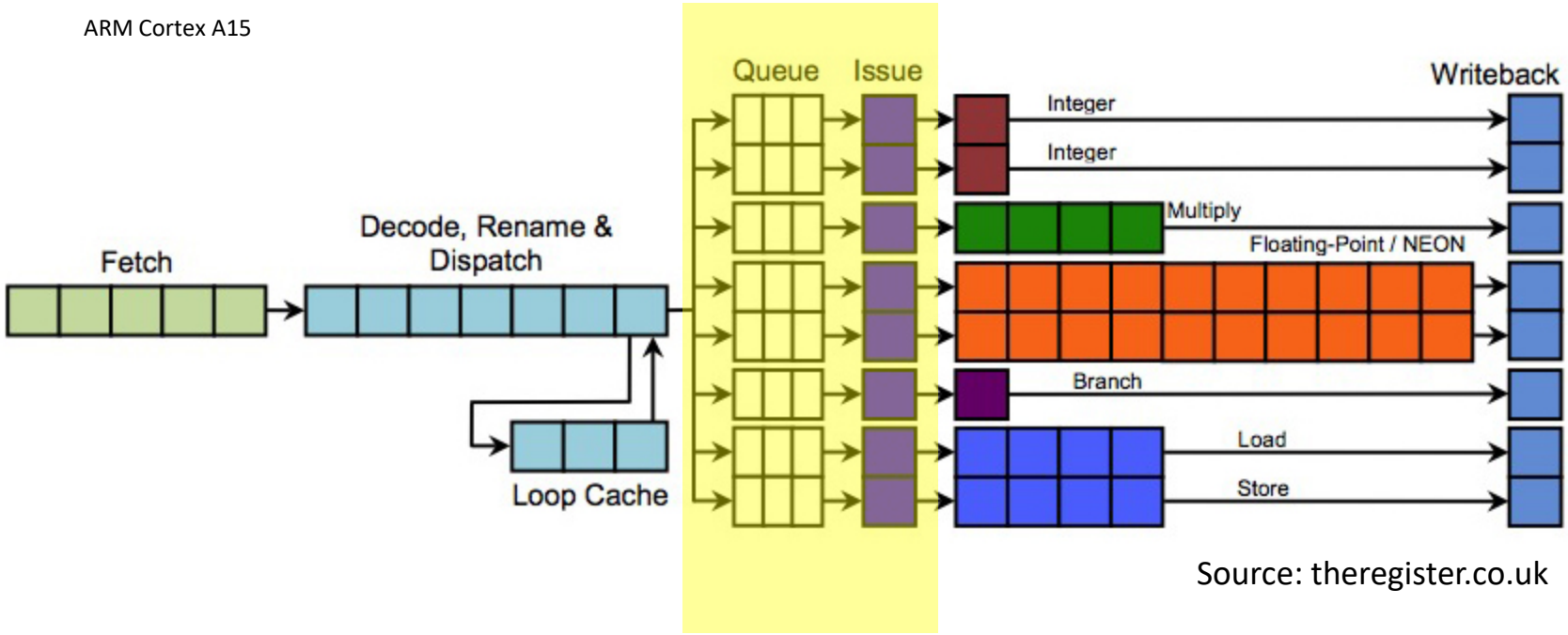


Register Data Flow

- 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

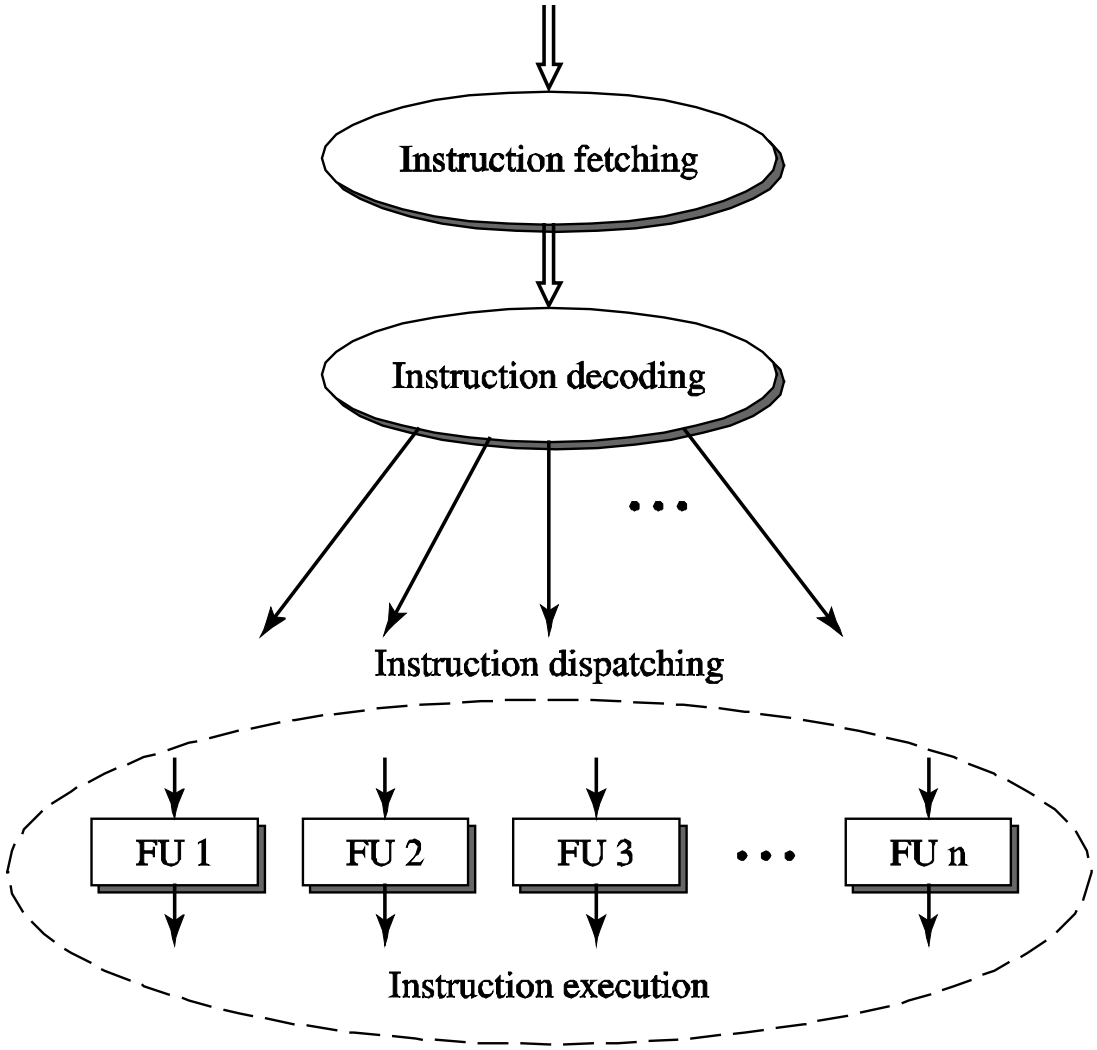


Issue Queues and Execution Lanes

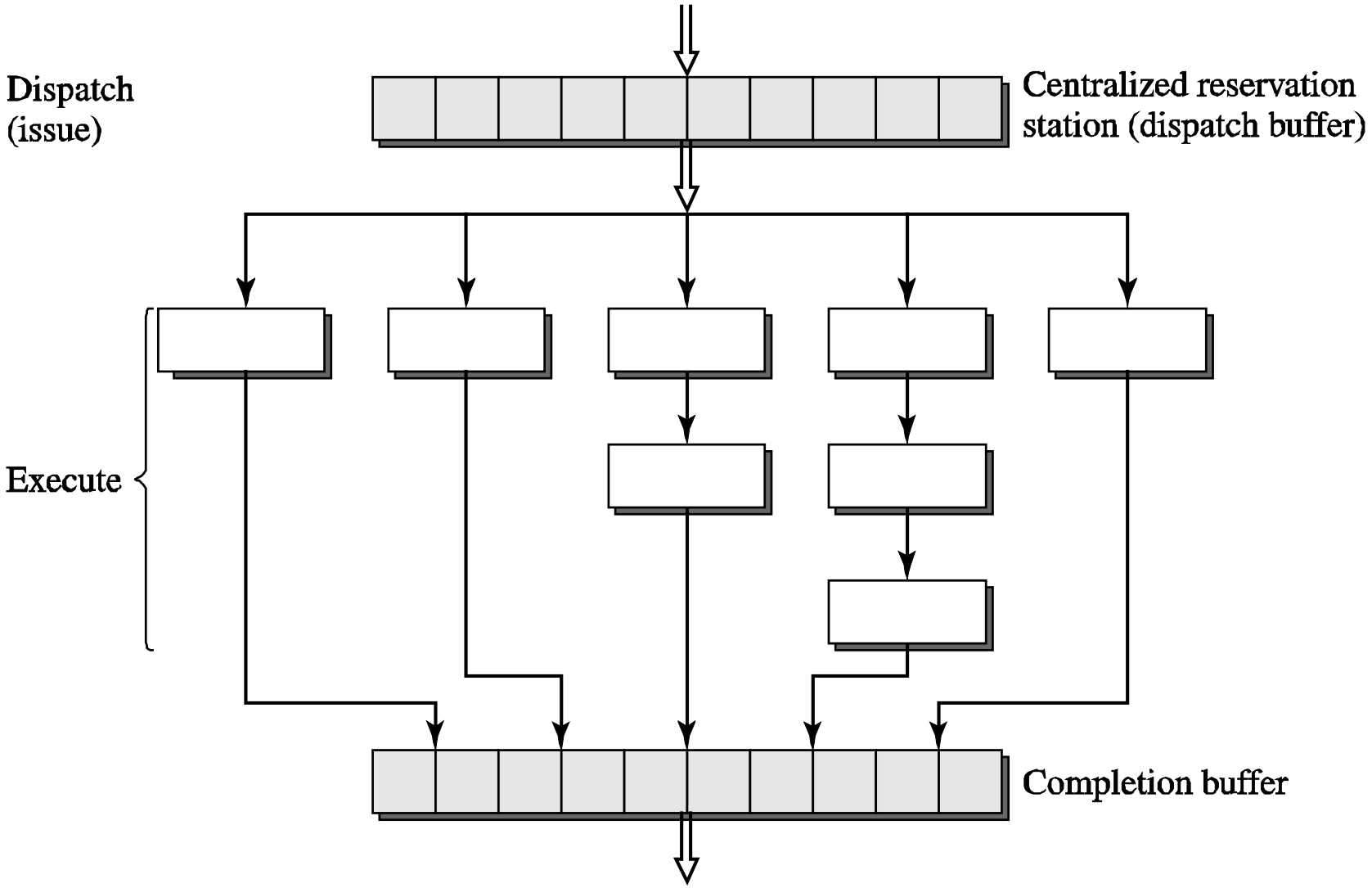


Source: theregister.co.uk

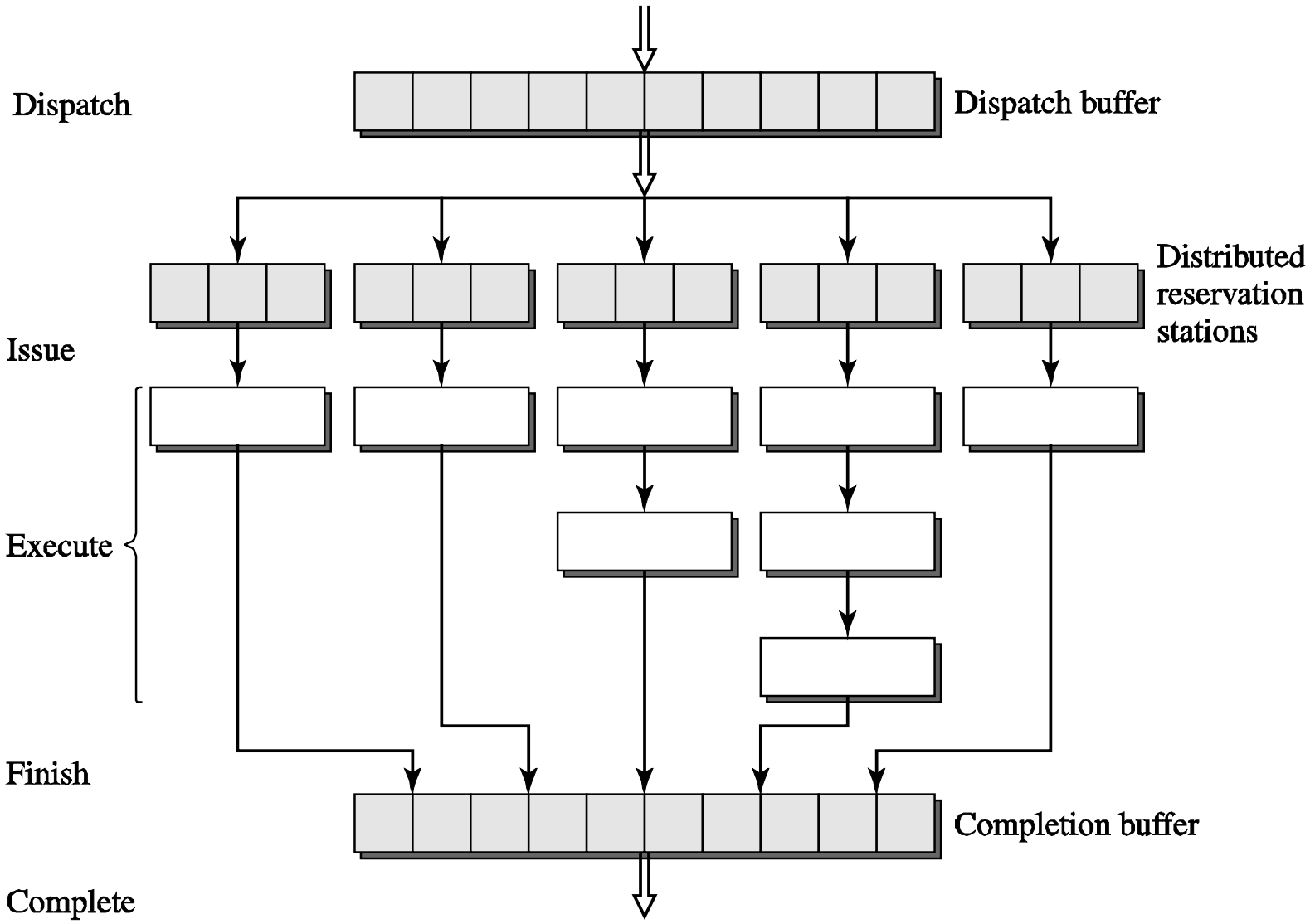
Necessity of Instruction Dispatch



Centralized Reservation Station



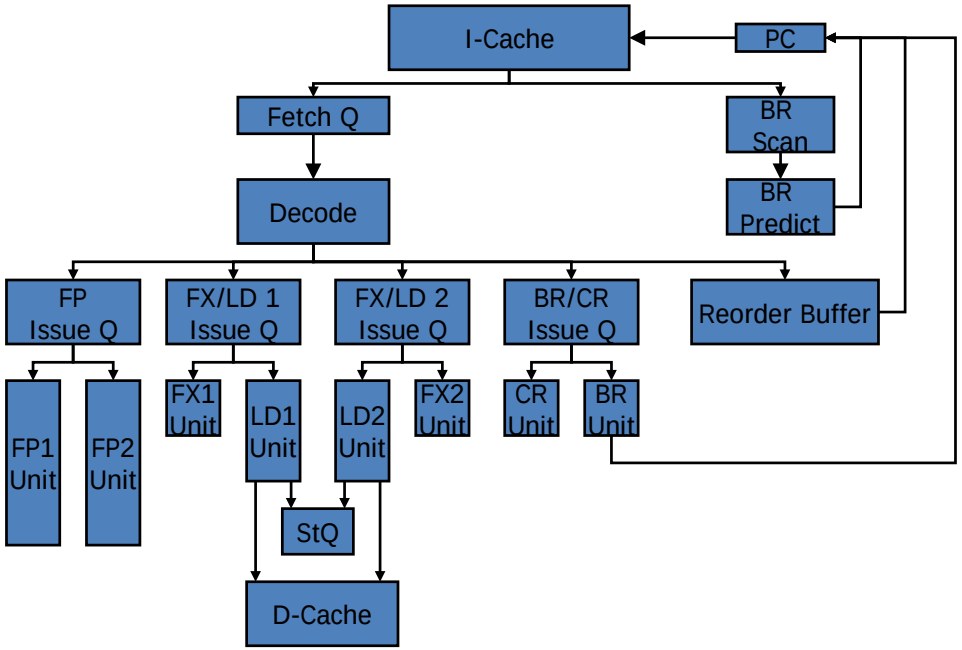
Distributed Reservation Station



Issues in Instruction Execution

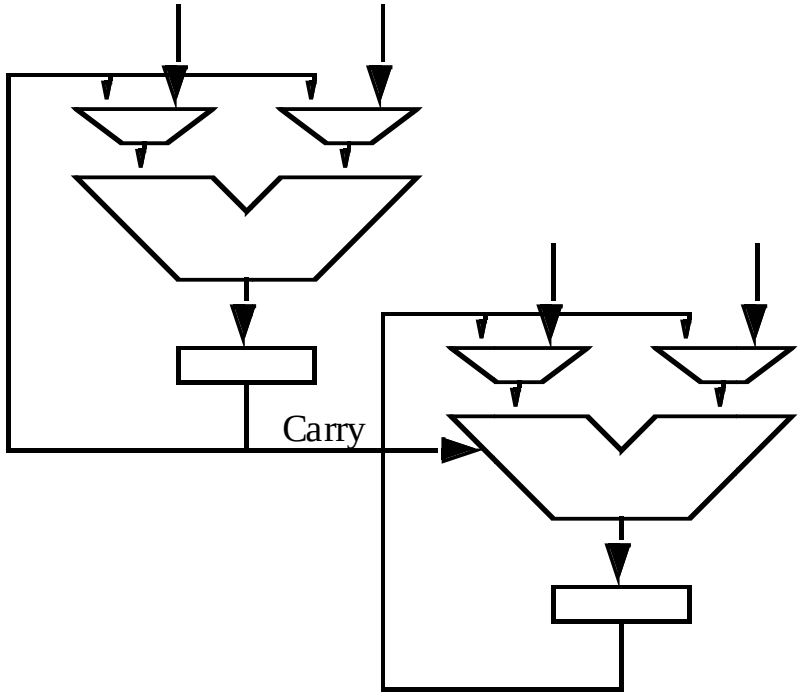
- Current trends
  - More parallelism ← bypassing very challenging
  - Deeper pipelines
  - More diversity
- Functional unit types
  - Integer => short vector
  - Floating point => vector (longer and longer)
  - Load/store ← most difficult to make parallel
  - Branch
  - Specialized units (media)
    - Very wide datapaths (256 bits/register or more)

Bypass Networks



- $O(n^2)$ interconnect from/to FU inputs and outputs
- Associative tag-match to find operands
- Solutions (hurt IPC, help cycle time)
  - Use RF only (IBM Power4) with no bypass network
  - Decompose into clusters (Alpha 21264)

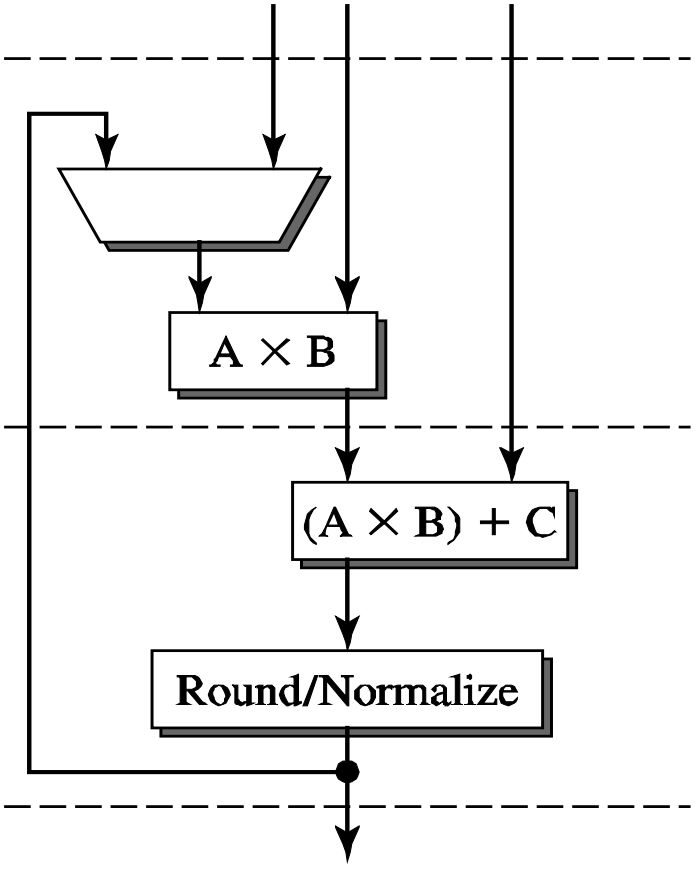
Specialized units



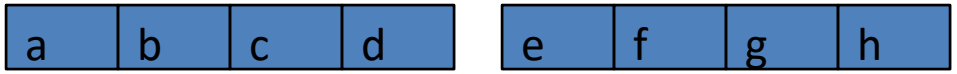
- Intel Pentium 4 staggered adders
  - *Fireball*
- Run at 2x clock frequency
- Two 16-bit bitslices
- Dependent ops execute on half-cycle boundaries
- Full result not available until full cycle later

Specialized units

- FP multiply-accumulate
  R = (A x B) + C
- Doubles FLOP/instruction
- Lose RISC instruction format symmetry:
  - 3 source operands
- Widely used



Media Data Types



- Subword parallel vector extensions
  - Media data (pixels, quantized datum) often 1-2 bytes
  - Several operands packed in single 32/64b register

    {a,b,c,d} and {e,f,g,h} stored in two 32b registers
  - Vector instructions operate on 4/8 operands in parallel
  - New instructions, e.g. sum of abs. differences (SAD)

    me = |a – e| + |b – f| + |c – g| + |d – h|
- Substantial throughput improvement
  - Usually requires hand-coding of critical loops
  - Shuffle ops (gather/scatter of vector elements)

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

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

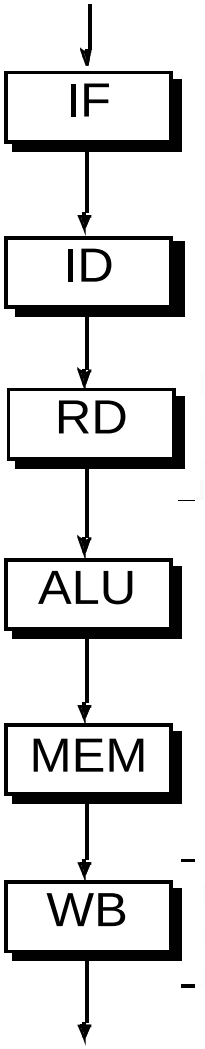
- Output dependence (WAW)
  - j cannot write its result until i has written its result

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

Pipeline Hazards

- Necessary conditions:
  - WAR: write stage earlier than read stage
    - Is this possible in IF-RD-EX-MEM-WB ?
  - WAW: write stage earlier than write stage
    - Is this possible in IF-RD-EX-MEM-WB ?
  - RAW: read stage earlier than write stage
    - Is this possible in IF-RD-EX-MEM-WB?
- If conditions not met, no need to resolve
- Check for both register and memory

Pipeline Hazard Analysis



- Memory hazards
  - WAR: Yes/No?
  - WAW: Yes/No?
  - RAW: Yes/No?
- Register hazards
  - WAR: Yes/No?
  - WAW: Yes/No?
  - RAW: Yes/No?

WAR: write stage earlier than read?
WAW: write stage earlier than write?
RAW: read stage earlier than write?

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!
  - 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 <= …

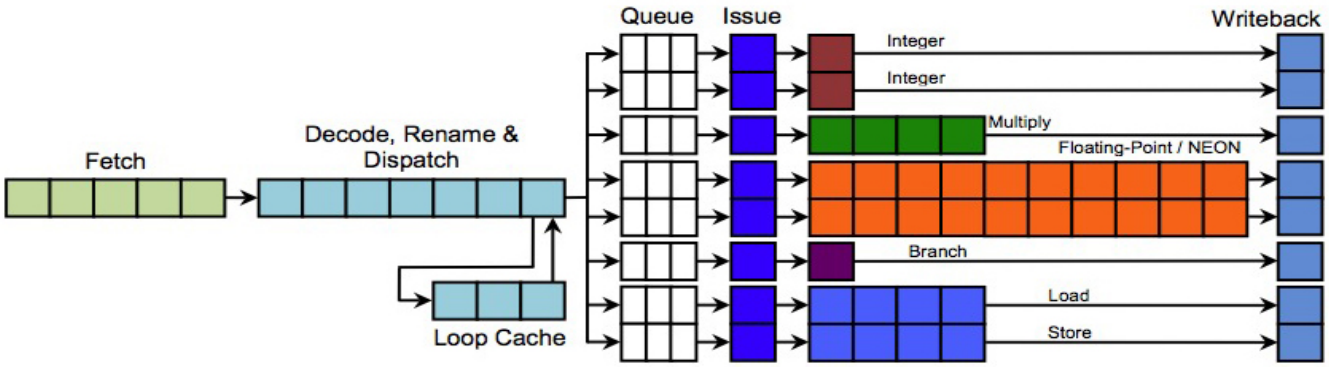
Register Renaming: RAW

- 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

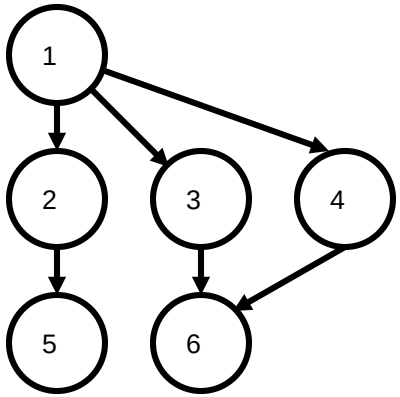


Source: theregister.co.uk

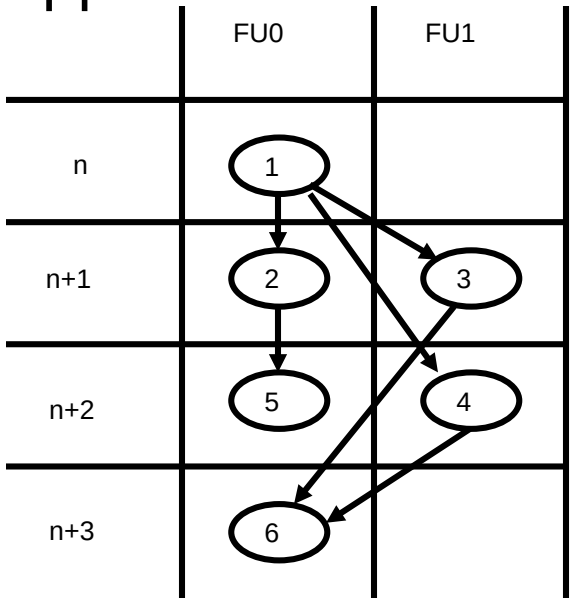
Instruction scheduling

- 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

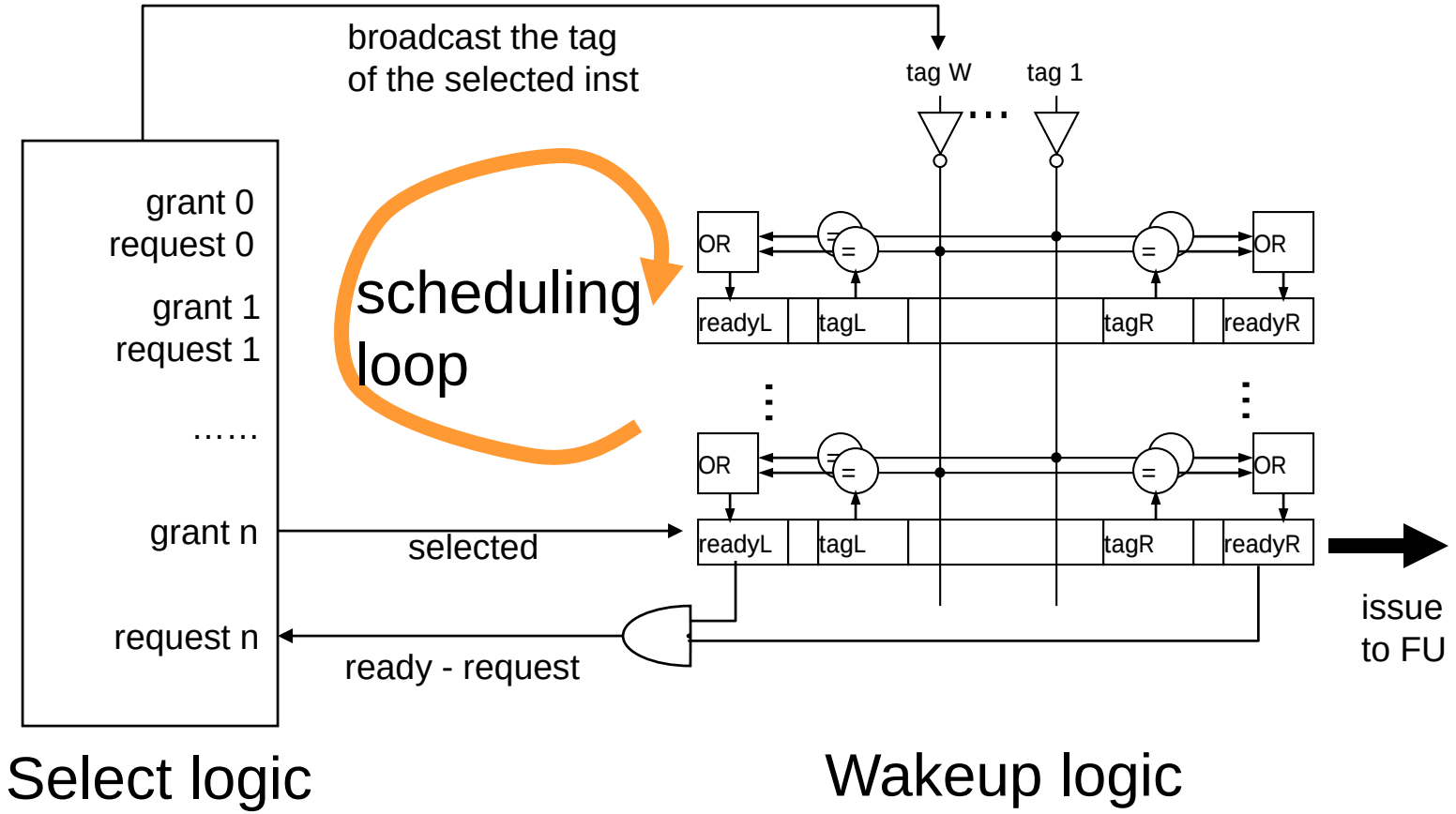


Instruction scheduling

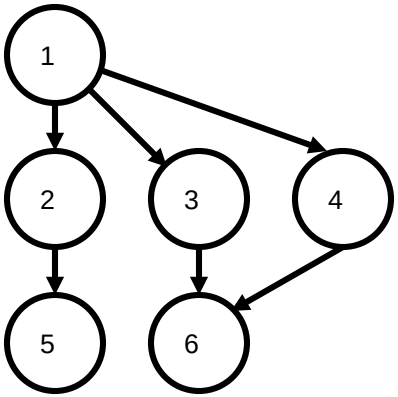
- A set of wakeup and select operations
  - Wakeup
    - Broadcasts the tags of parent instructions selected
    - Dependent instruction gets matching tags, determines if source operands are ready
    - Resolves true data dependences
  - Select
    - Picks instructions to issue among a pool of ready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

Scheduling loop

- Basic wakeup and select operations

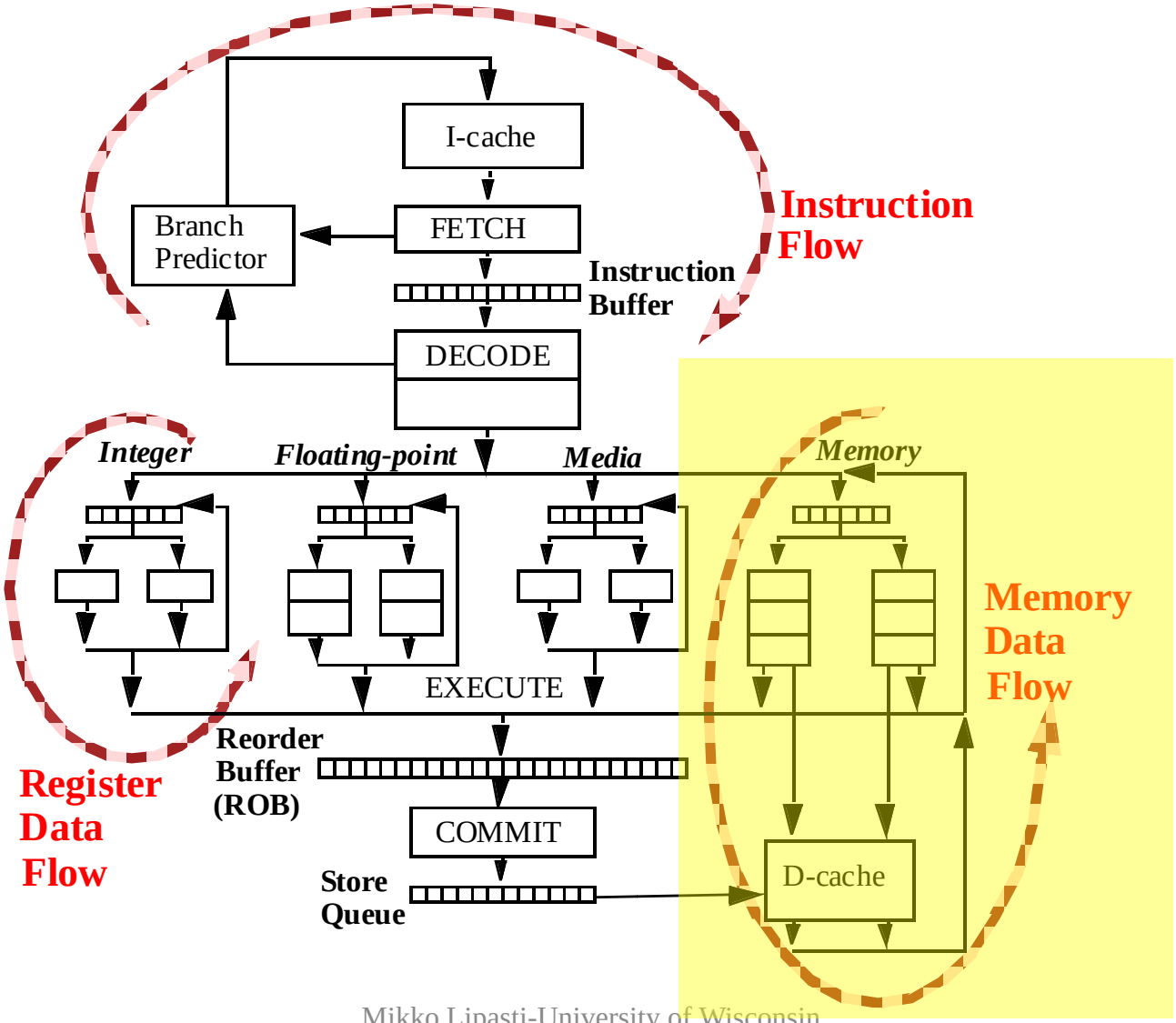


Wakeup and Select



	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

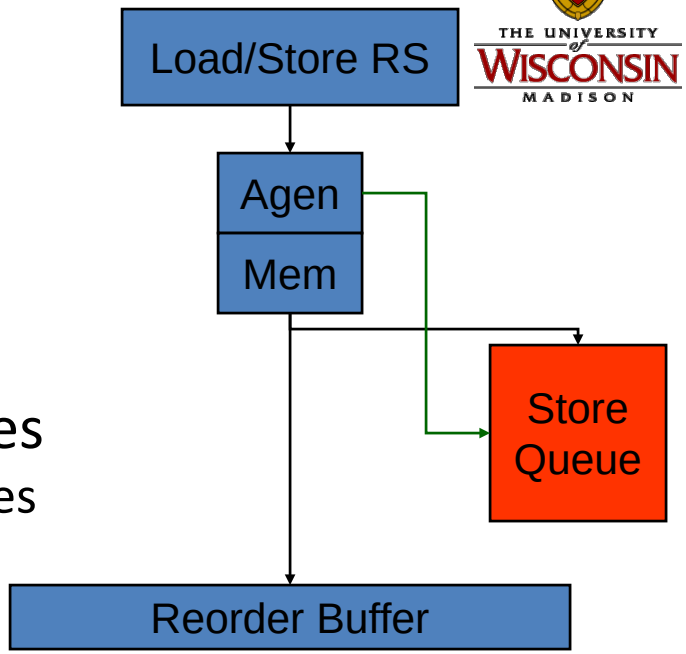
High-IPC Processor



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

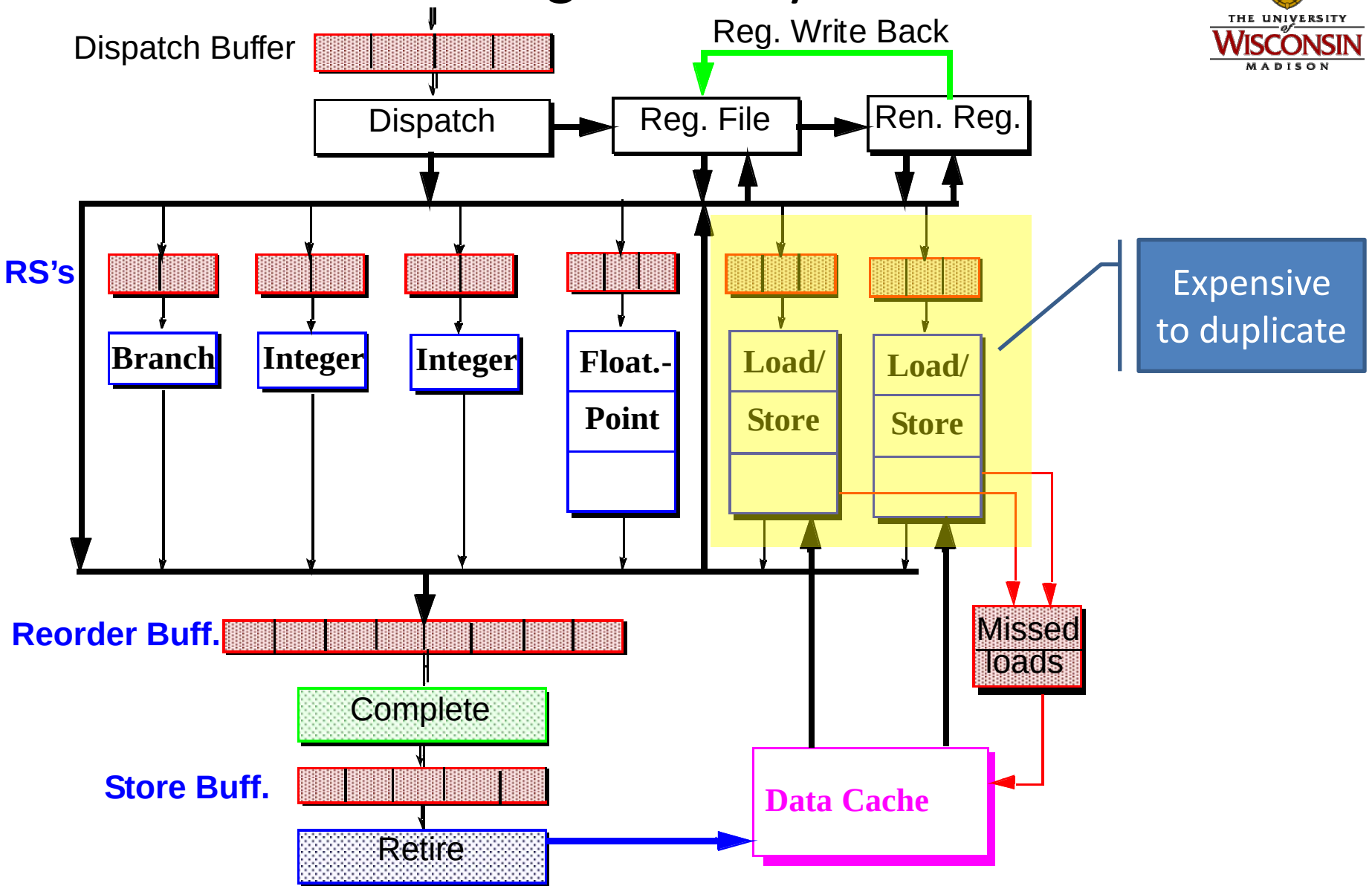
Memory Data Dependences



- 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?

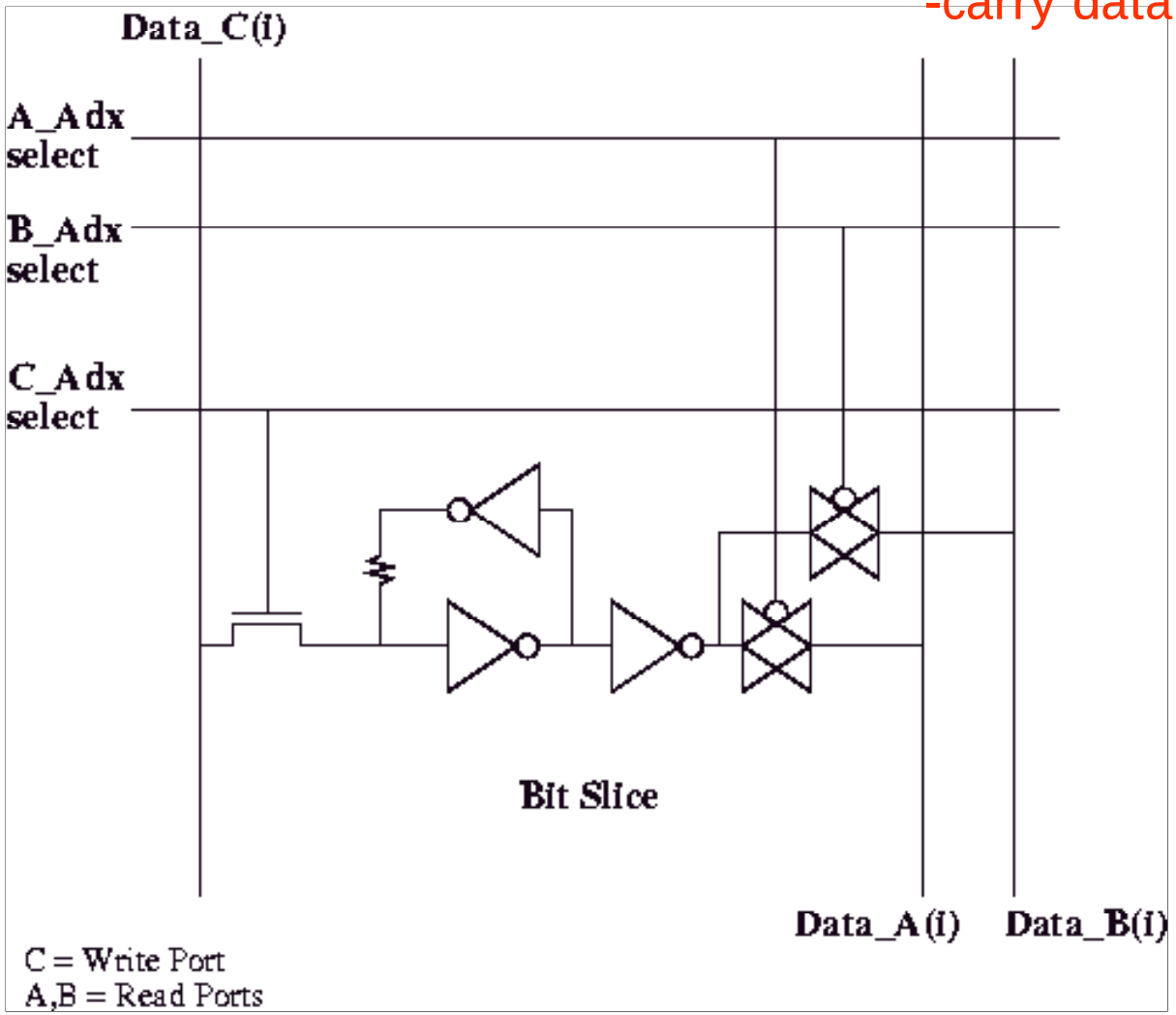
Increasing Memory Bandwidth



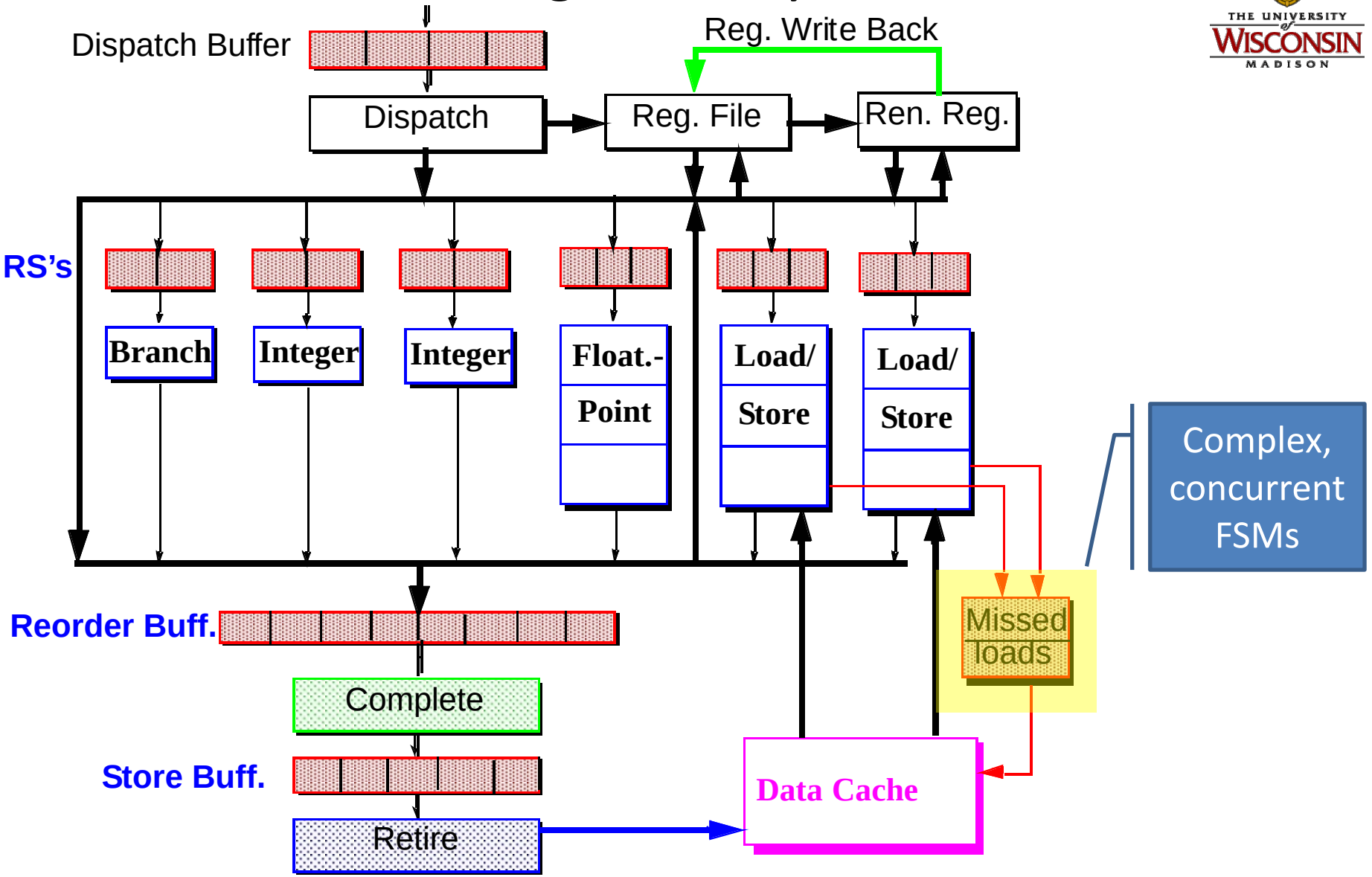
True Multiporting of SRAM

"Bit" Lines
-carry data in/out

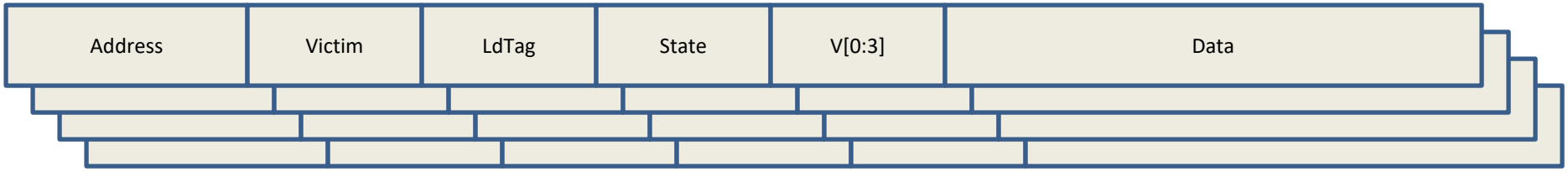
"Word" Lines
-select a row



Increasing Memory Bandwidth



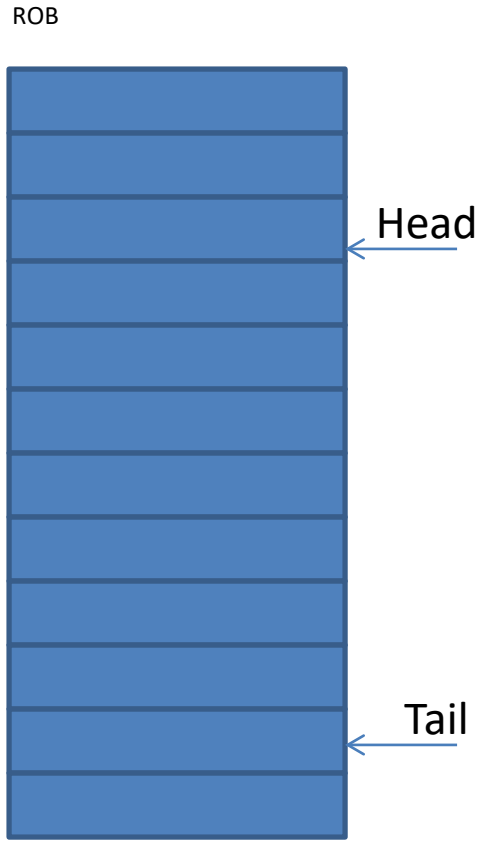
Miss Status Handling Register



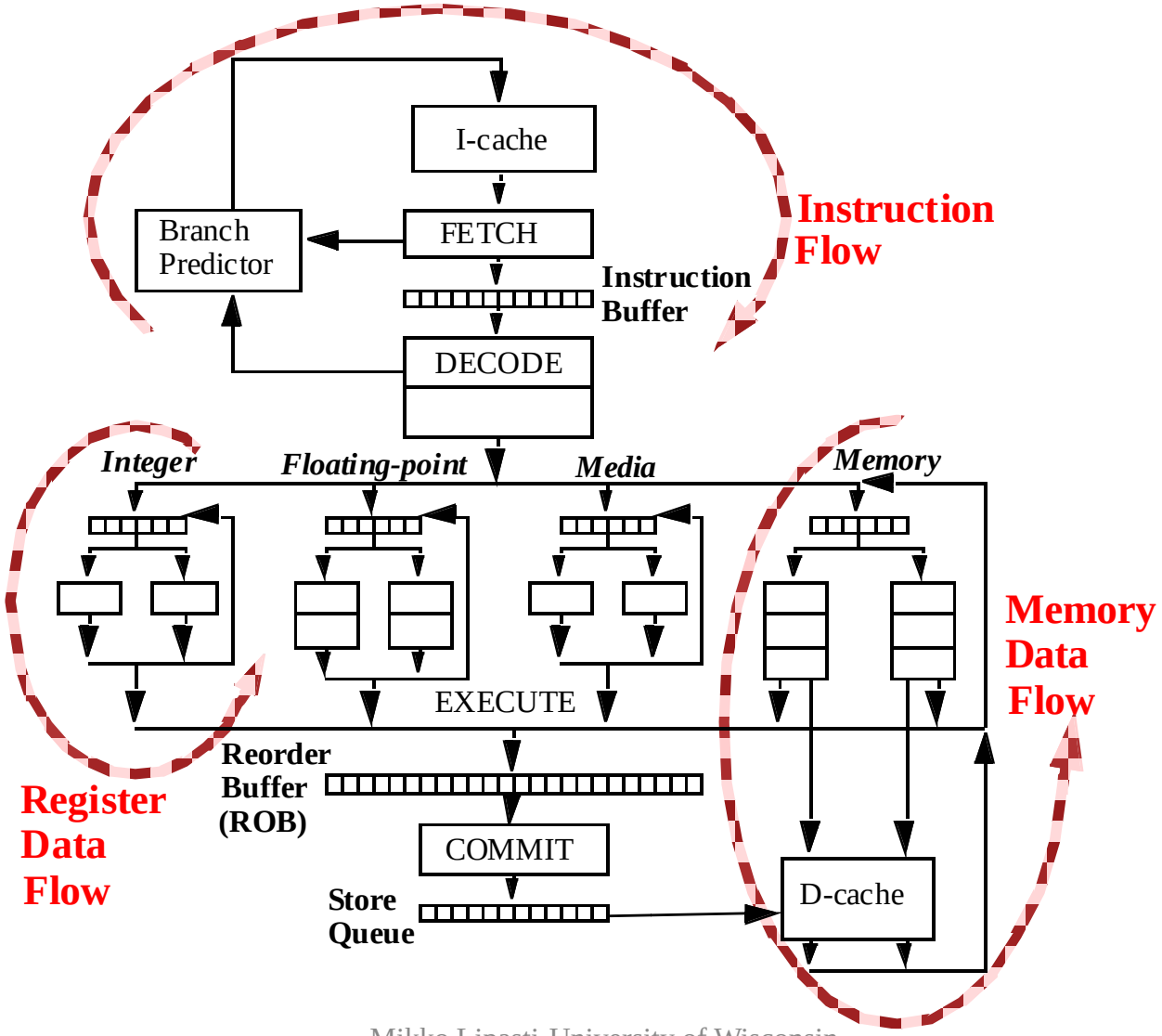
- Each MSHR entry keeps track of:
  - Address: miss address
  - Victim: set/way to replace
  - LdTag: which load (s) to wake up
  - State: coherence state, fill status
  - V[0:3]: subline valid bits
  - Data: block data to be filled into cache

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



Superscalar Overview

- Instruction flow
  - Branches, jumps, calls: predict target, direction
  - Fetch alignment
  - Instruction cache misses
- Register data flow
  - Register renaming: RAW/WAR/WAW
  - Instruction scheduling: wakeup & select
- Memory data flow
  - In-order stores: WAR/WAW
  - Store queue: RAW
  - Data cache misses