THE UNIVERSITY
of
WISCONSIN
MADISON

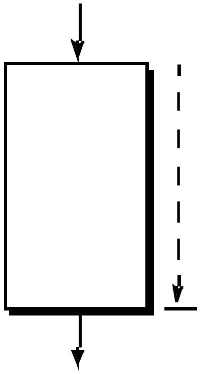
# ECE/CS 552: Introduction to Superscalar Processors

Lecture notes based in part on slides created by Mark Hill, David Wood, Guri Sohi, John Shen and Jim Smith

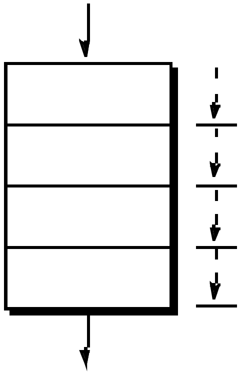
# Limitations of Scalar Pipelines

- 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

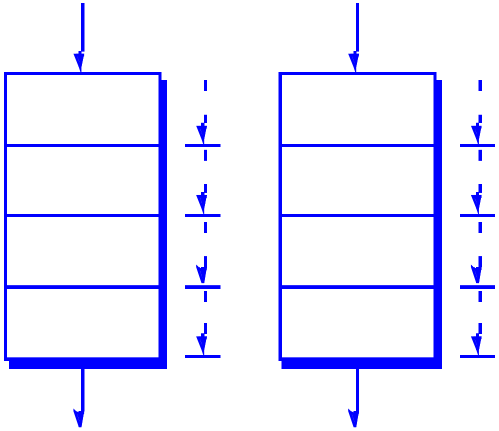
# Parallel Pipelines



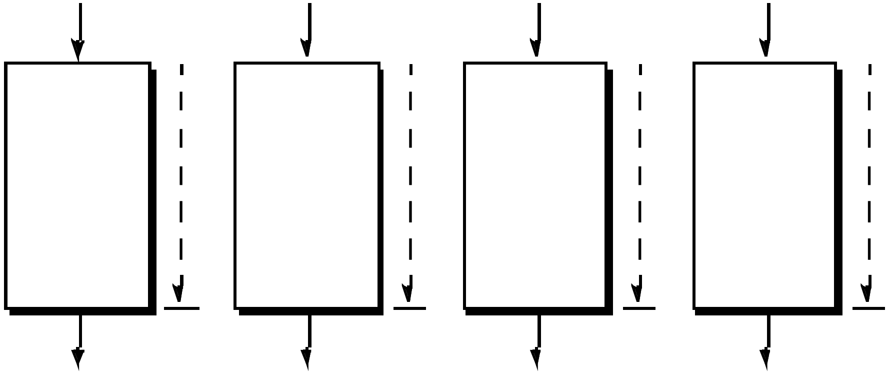
(a) No Parallelism



(b) Temporal Parallelism

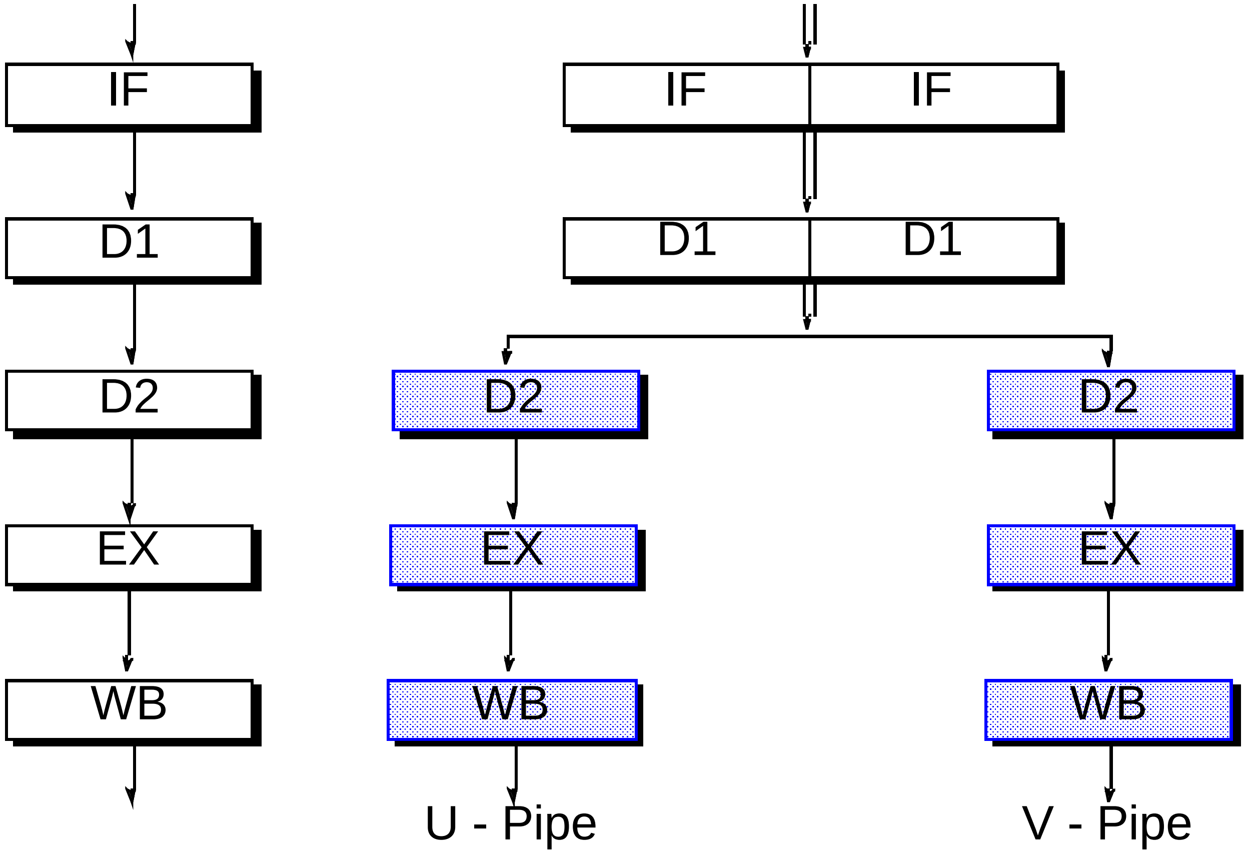


(d) Parallel Pipeline

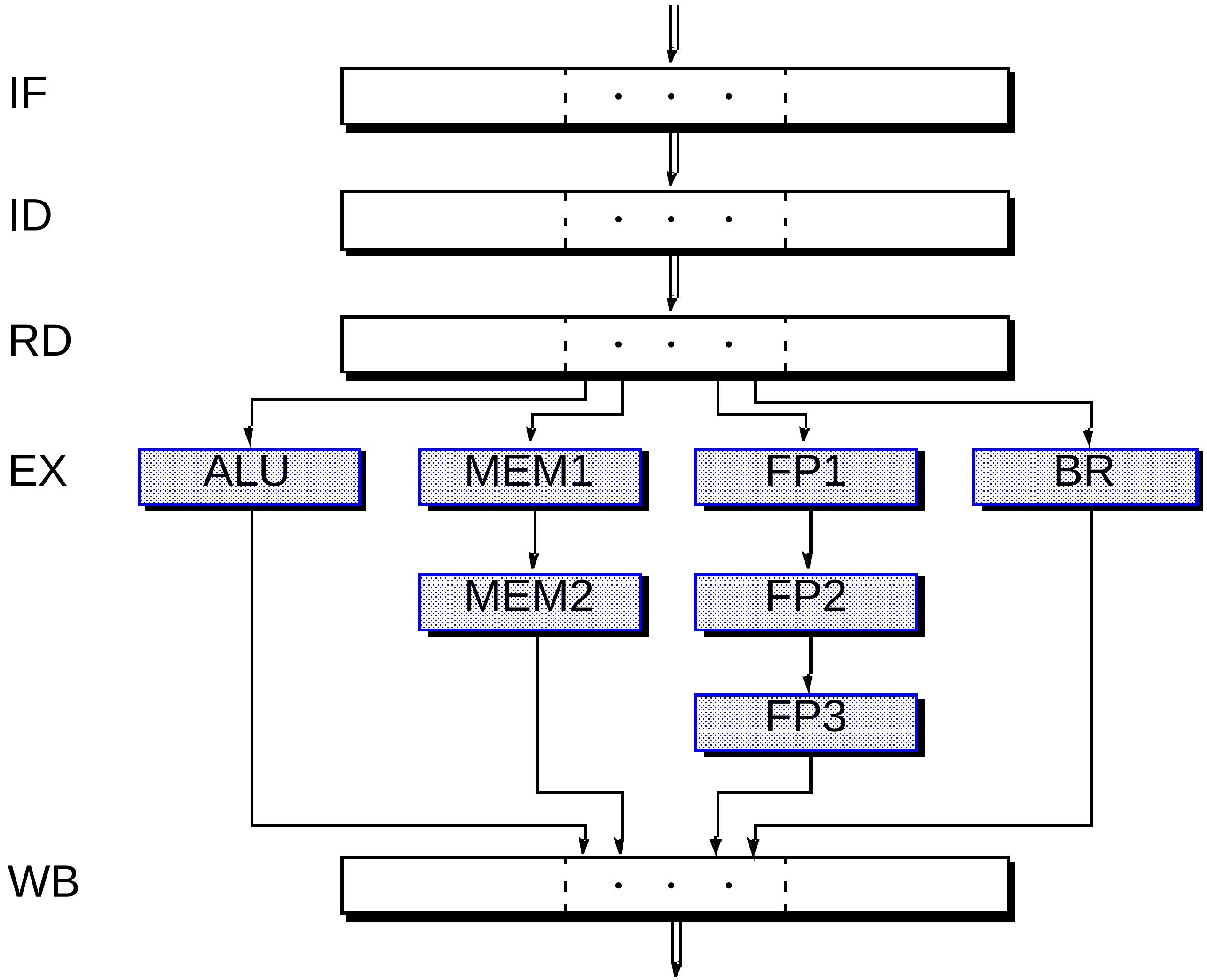


(c) Spatial Parallelism

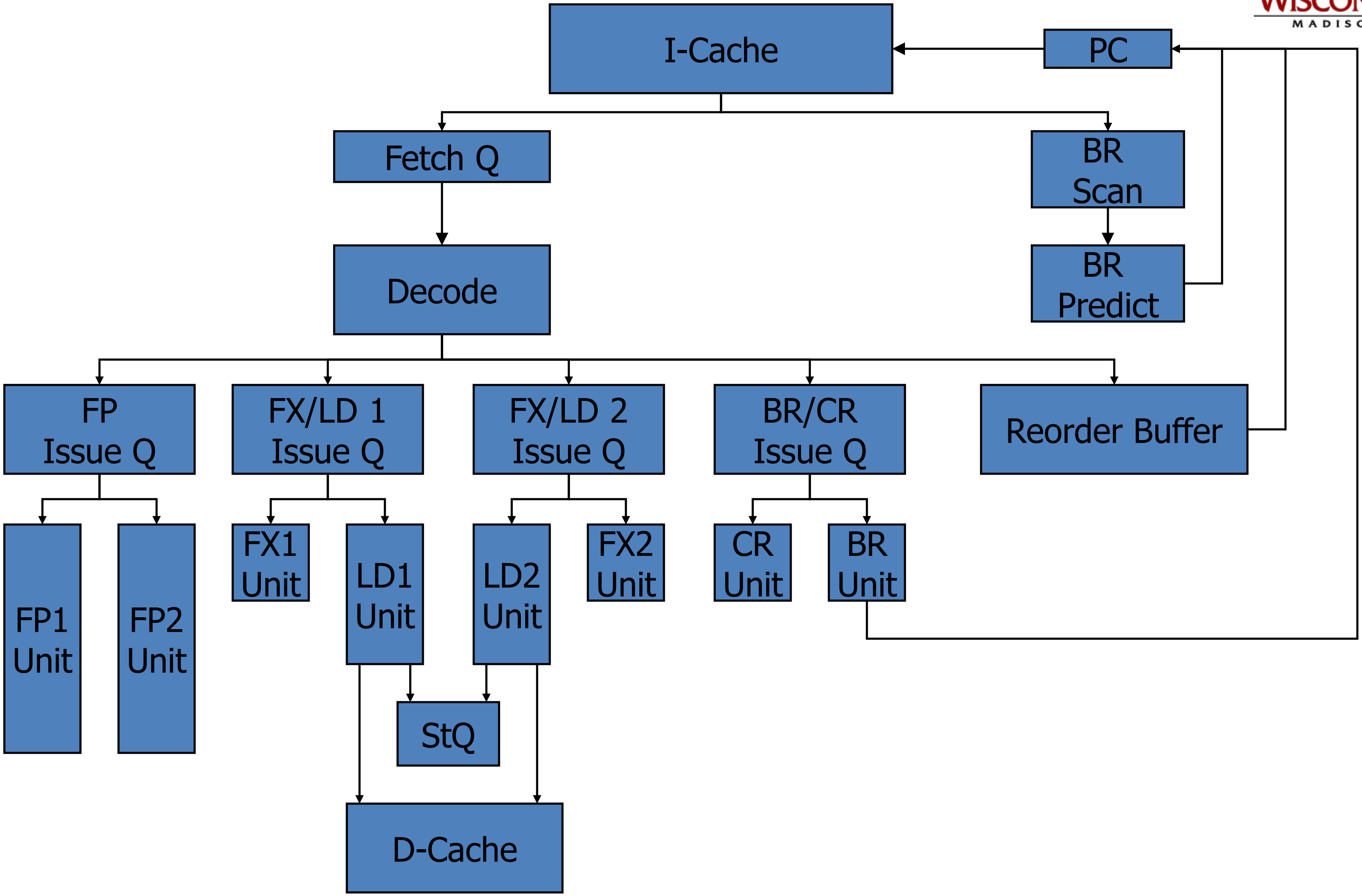
# Intel Pentium Parallel Pipeline



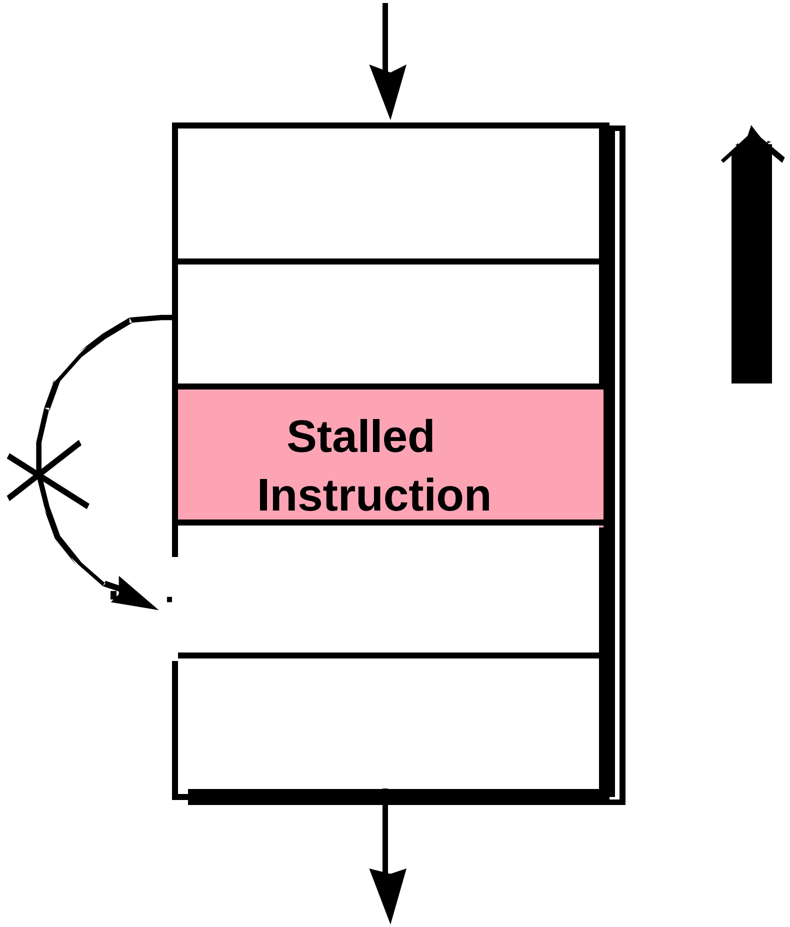
# Diversified Pipelines



# IBM Power4 Diversified Pipelines



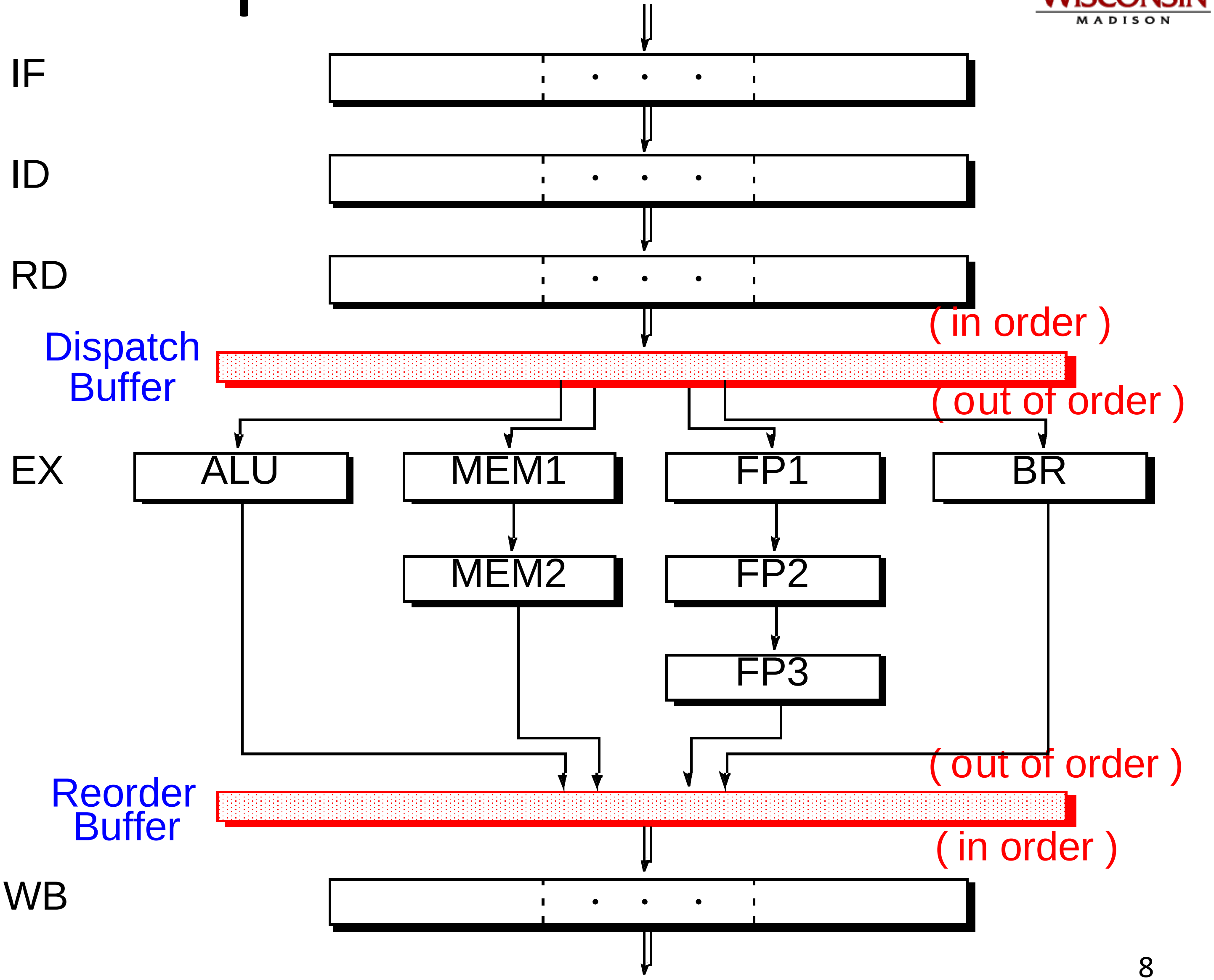
# Rigid Pipeline Stall Policy



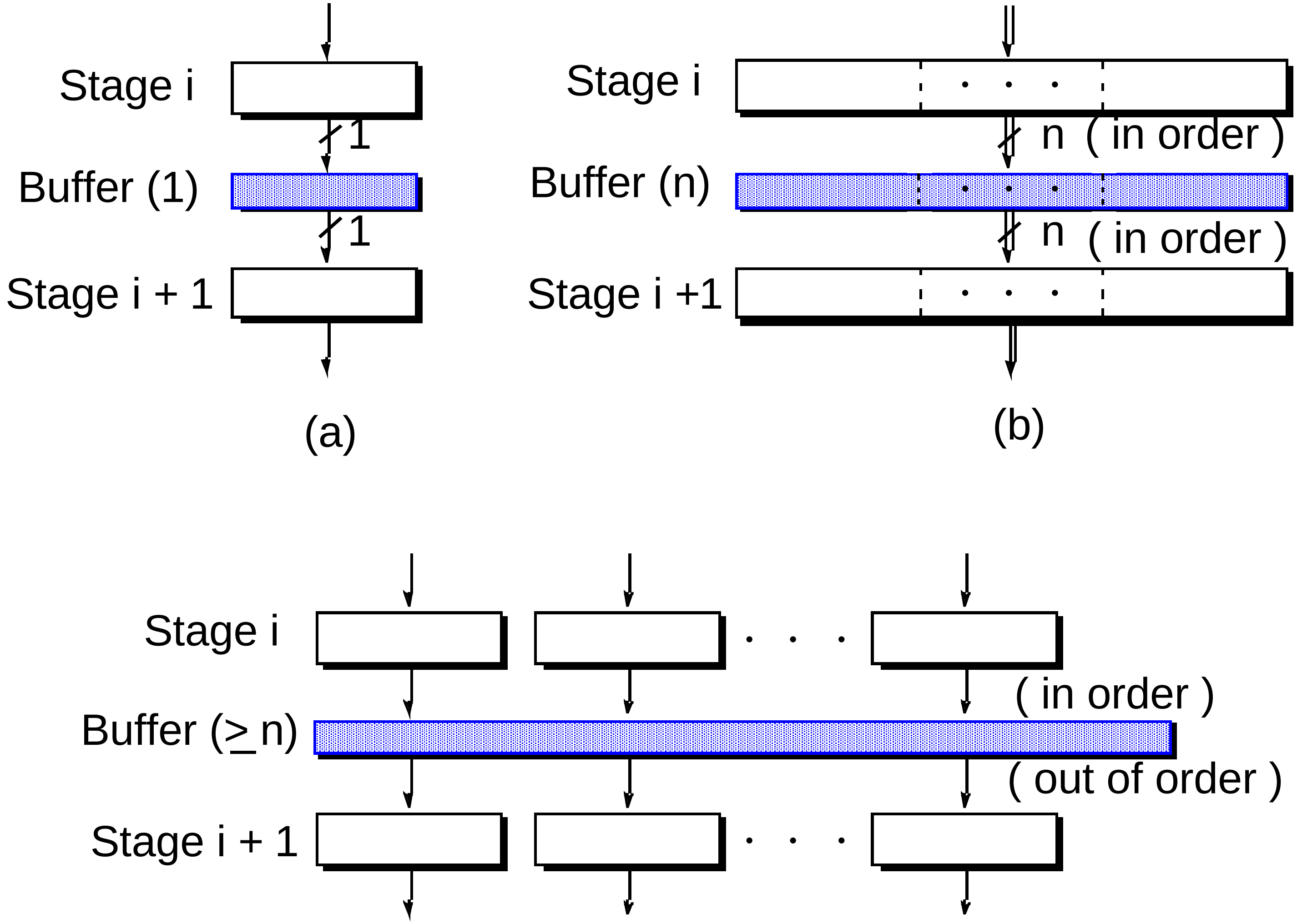
Bypassing of Stalled Instruction Not Allowed

Backward Propagation of Stalling

# Dynamic Pipelines



# Interstage Buffers

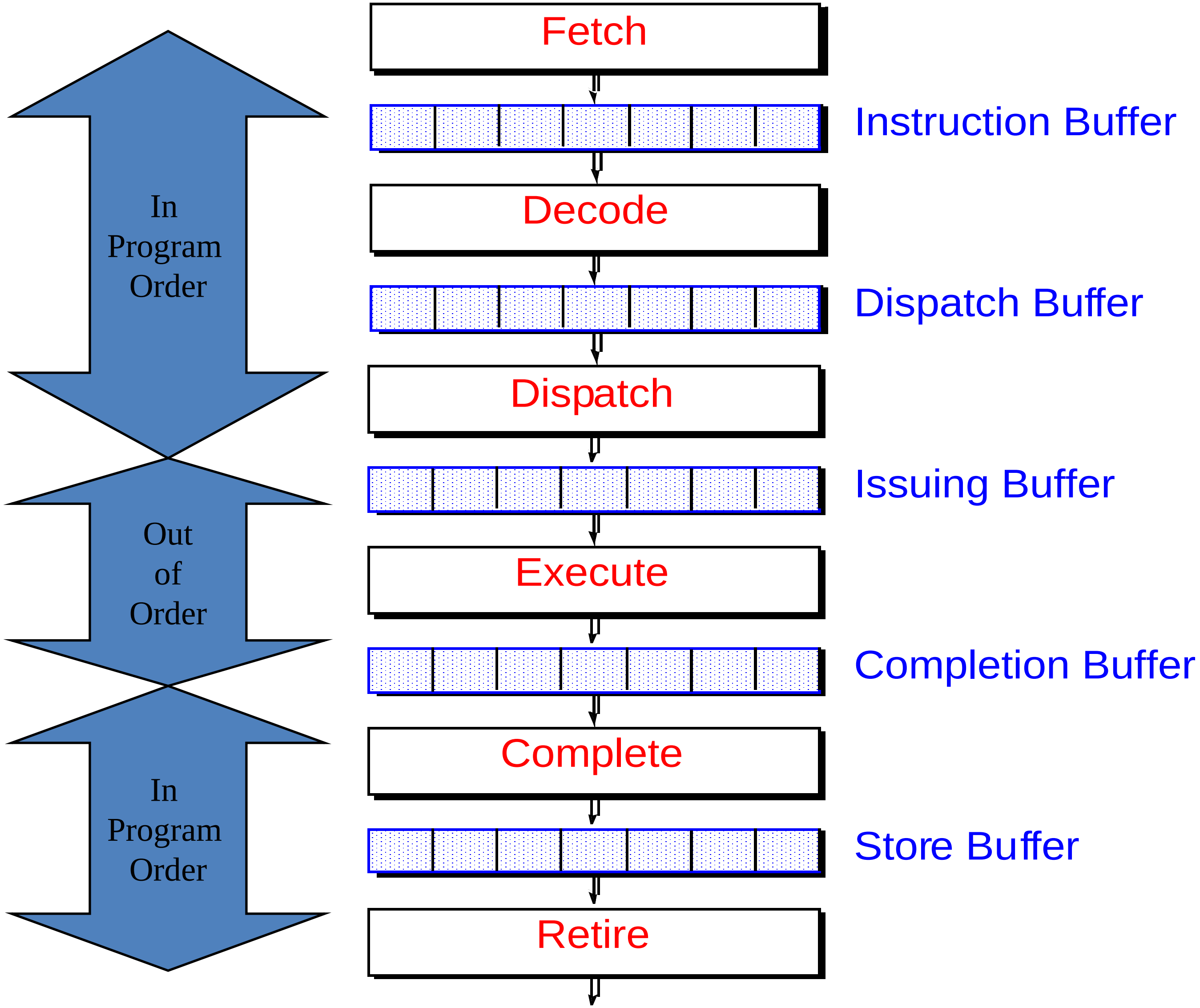


(a)

(b)

(c)

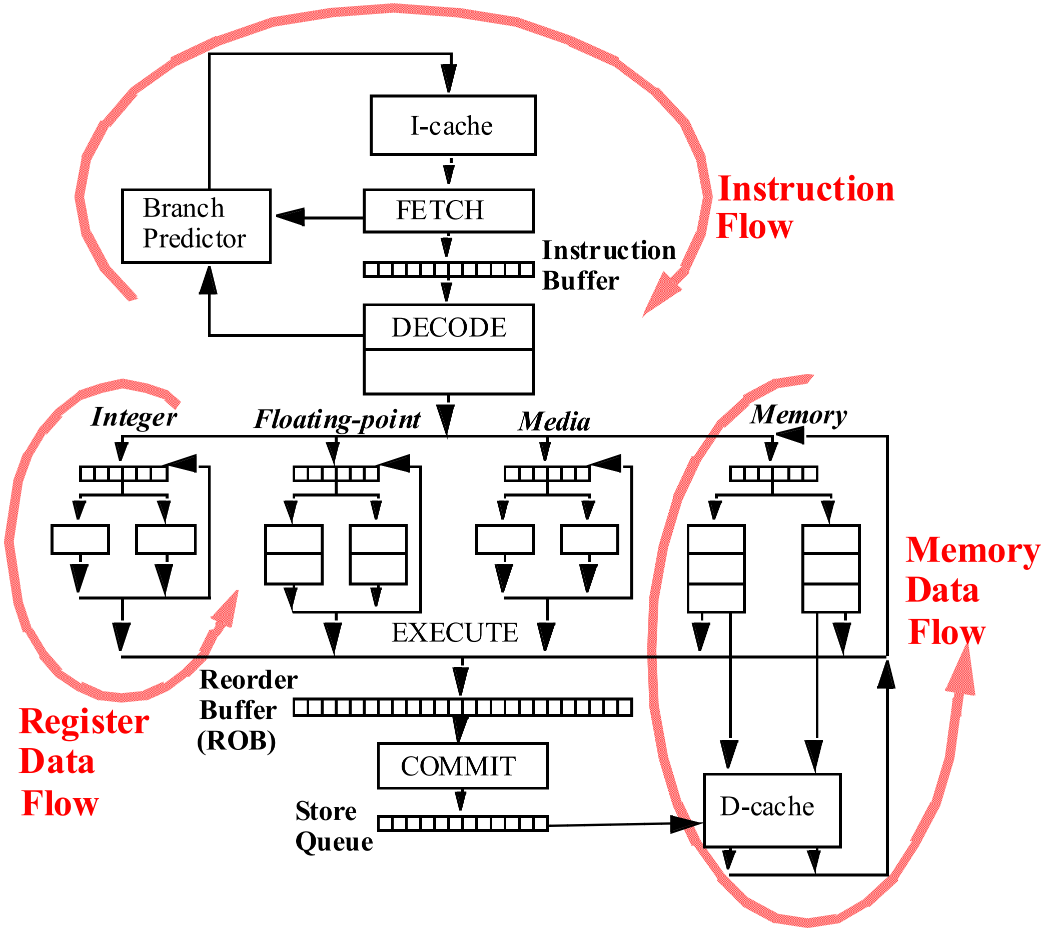
# Superscalar Pipeline Stages



# 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

# Impediments to High IPC



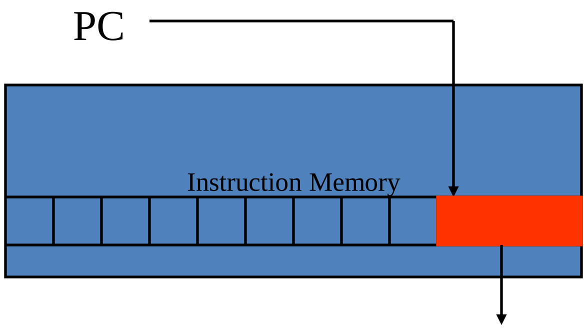
# Superscalar Pipeline Design

- Instruction Fetching Issues
- Instruction Decoding Issues
- Instruction Dispatching Issues
- Instruction Execution Issues
- Instruction Completion & Retiring Issues

# Instruction Fetch

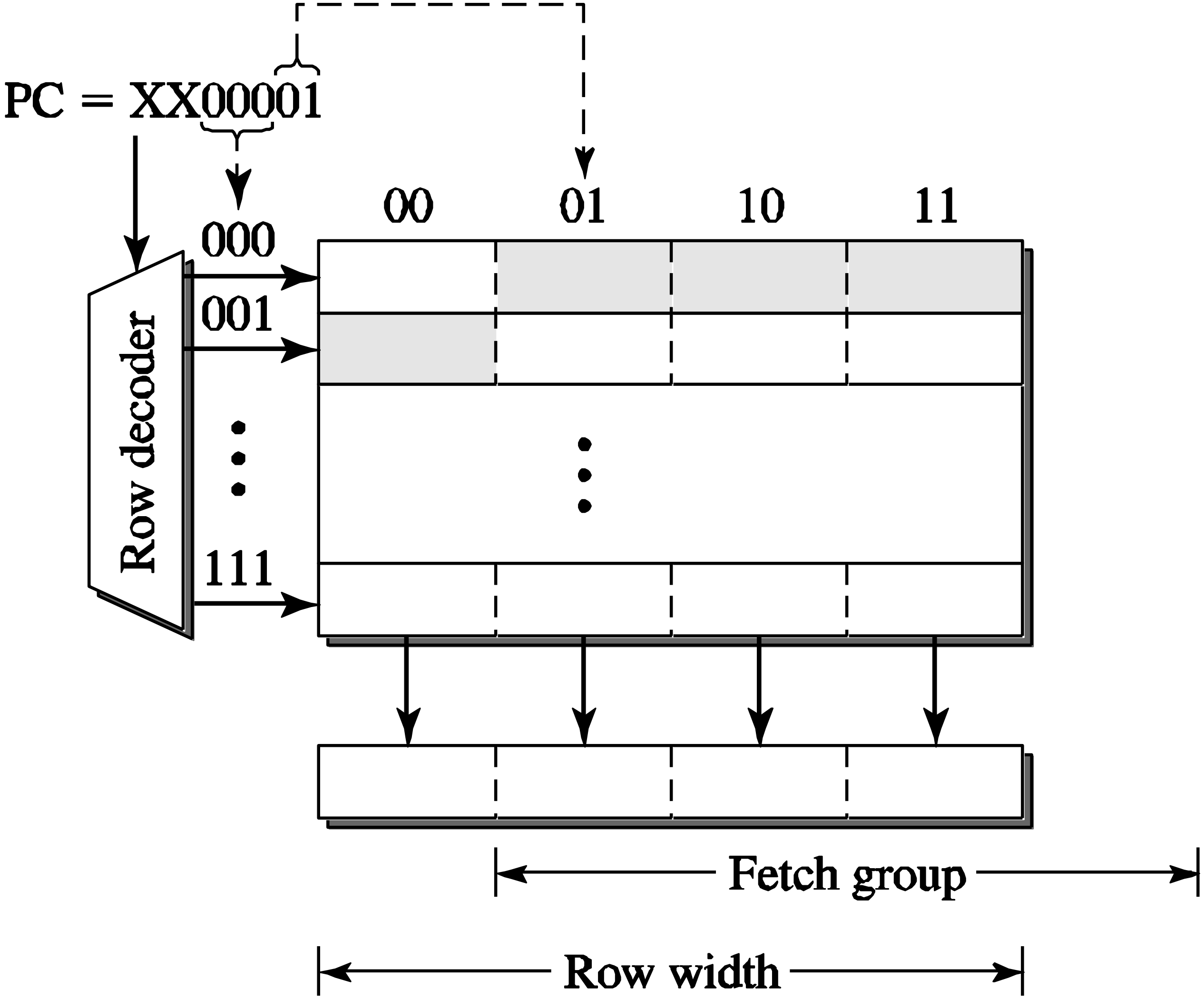
Objective: Fetch multiple instructions per cycle

- Challenges:
  - Branches: control dependences
  - Branch target misalignment
  - Instruction cache misses
- Solutions
  - Alignment hardware
  - Prediction/speculation



3 instructions fetched

# Fetch Alignment



# Branches – MIPS

6 Types of Branches

Jump (uncond, no save PC, imm)

Jump and link (uncond, save PC, imm)

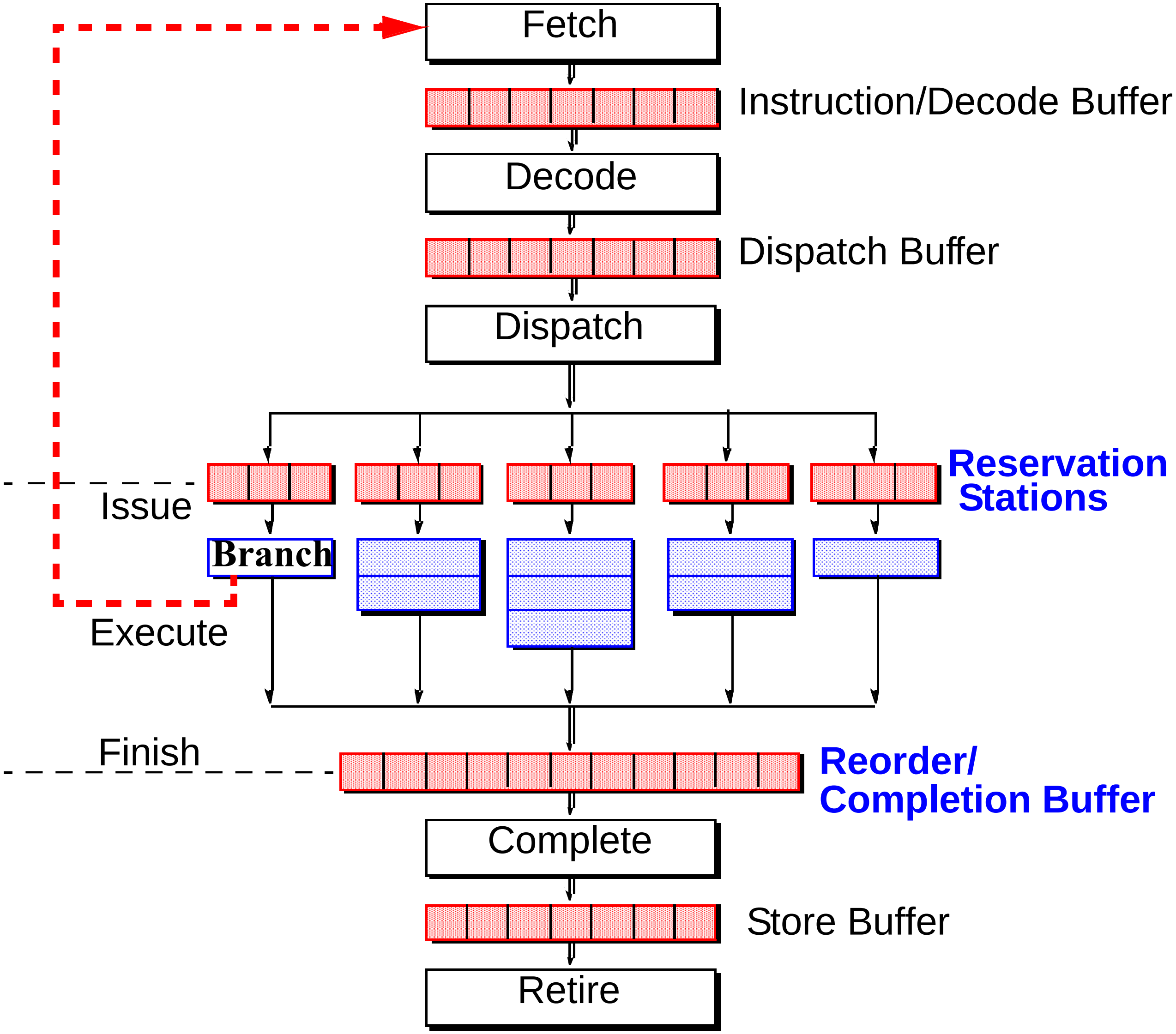
Jump register (uncond, no save PC, register)

Jump and link register (uncond, save PC, register)

Branch (conditional, no save PC, PC+imm)

Branch and link (conditional, save PC, PC+imm)

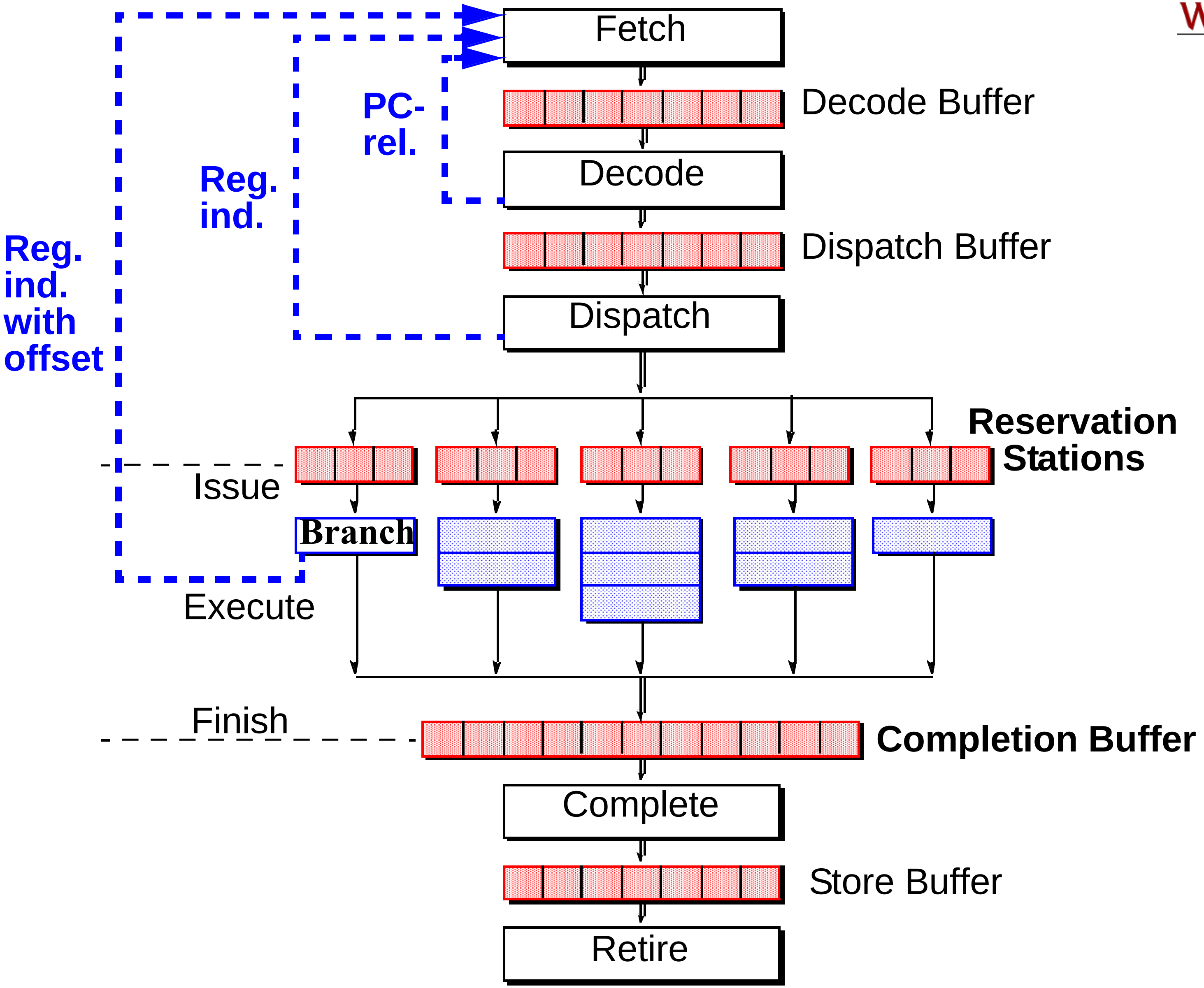
# Disruption of Sequential Control Flow



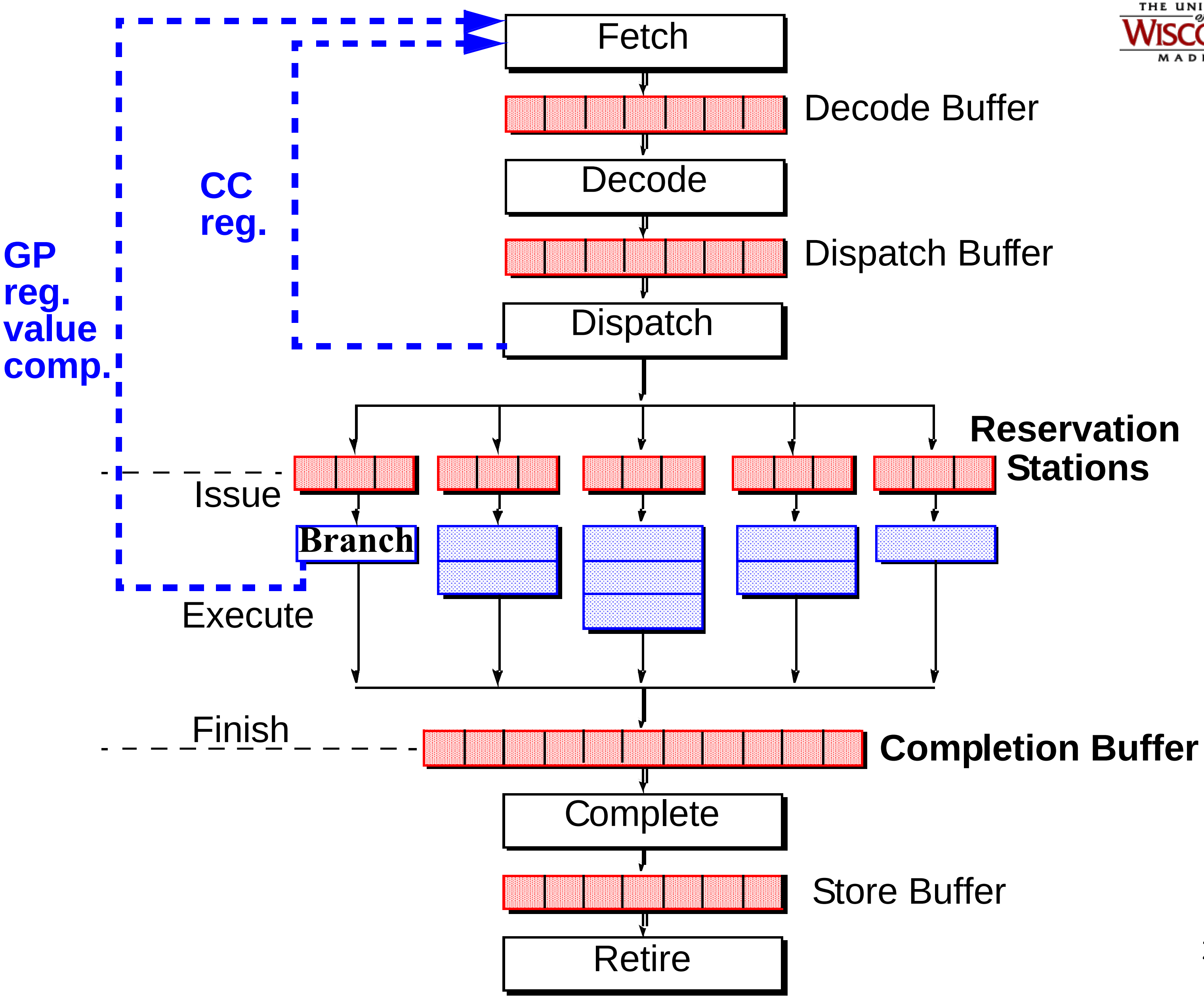
# Branch Prediction

- 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)

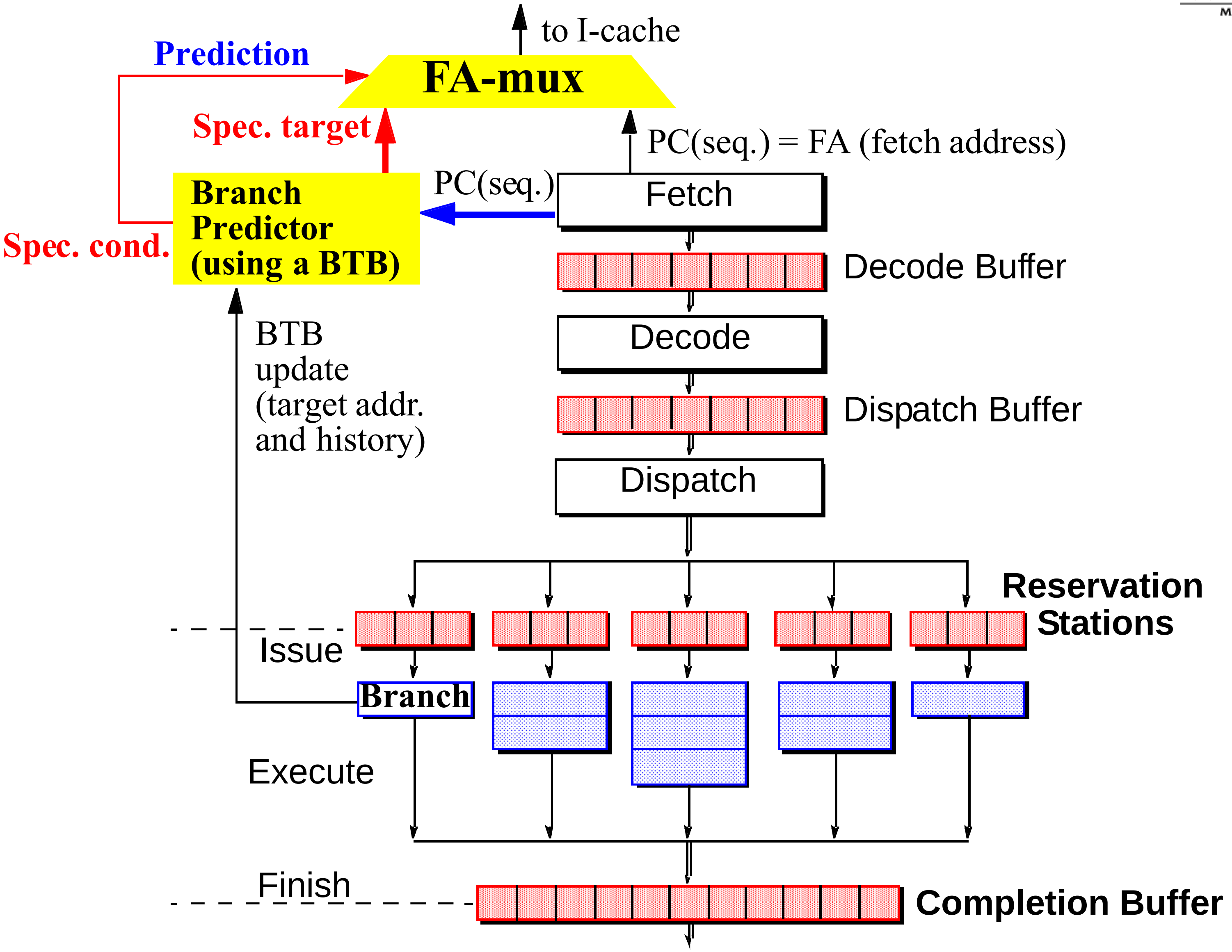
# Target Address Generation



# Condition Resolution



# Branch Instruction Speculation



# Static Branch Prediction

- Single-direction
  - Always not-taken: Intel i486
- Backwards Taken/Forward Not Taken
  - Loop-closing branches have negative offset
  - Used as backup in Pentium Pro, II, III, 4

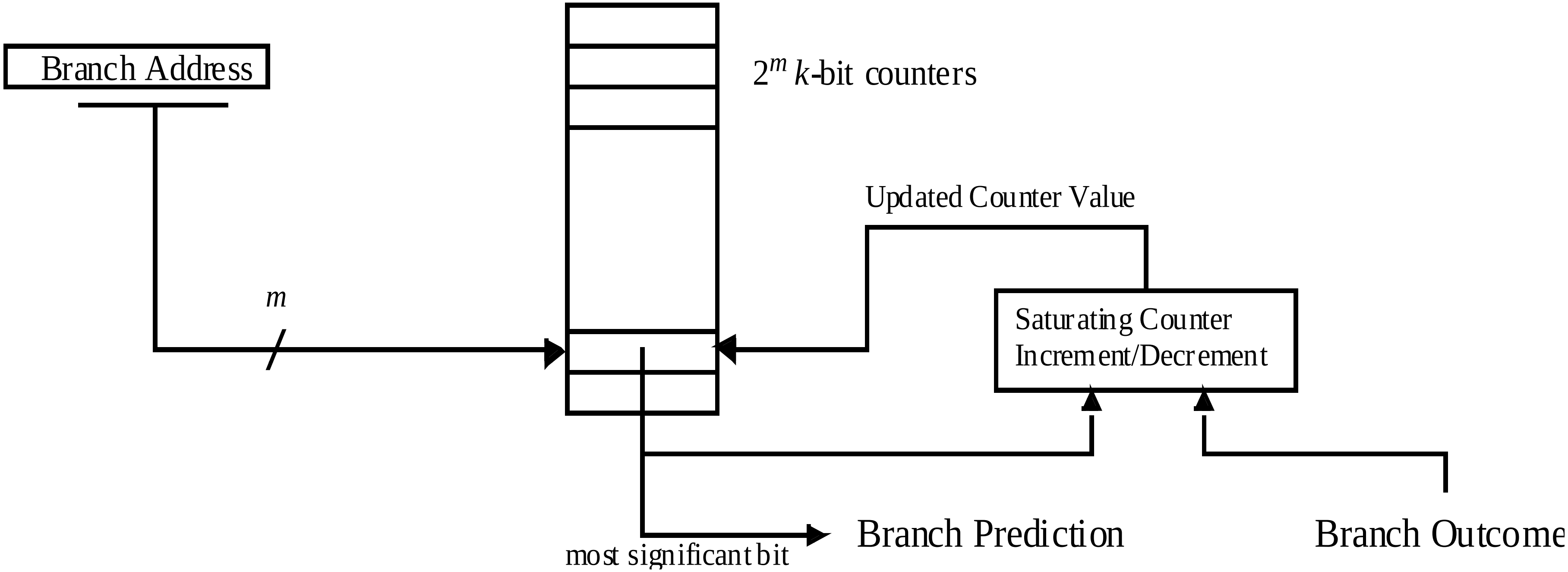
# Static Branch Prediction

- Profile-based
  1. Instrument program binary
  2. Run with representative (?) input set
  3. Recompile program
     a. Annotate branches with hint bits, or
     b. Restructure code to match predict not-taken
- Performance: 75-80% accuracy
  – Much higher for “easy” cases

# Dynamic Branch Prediction

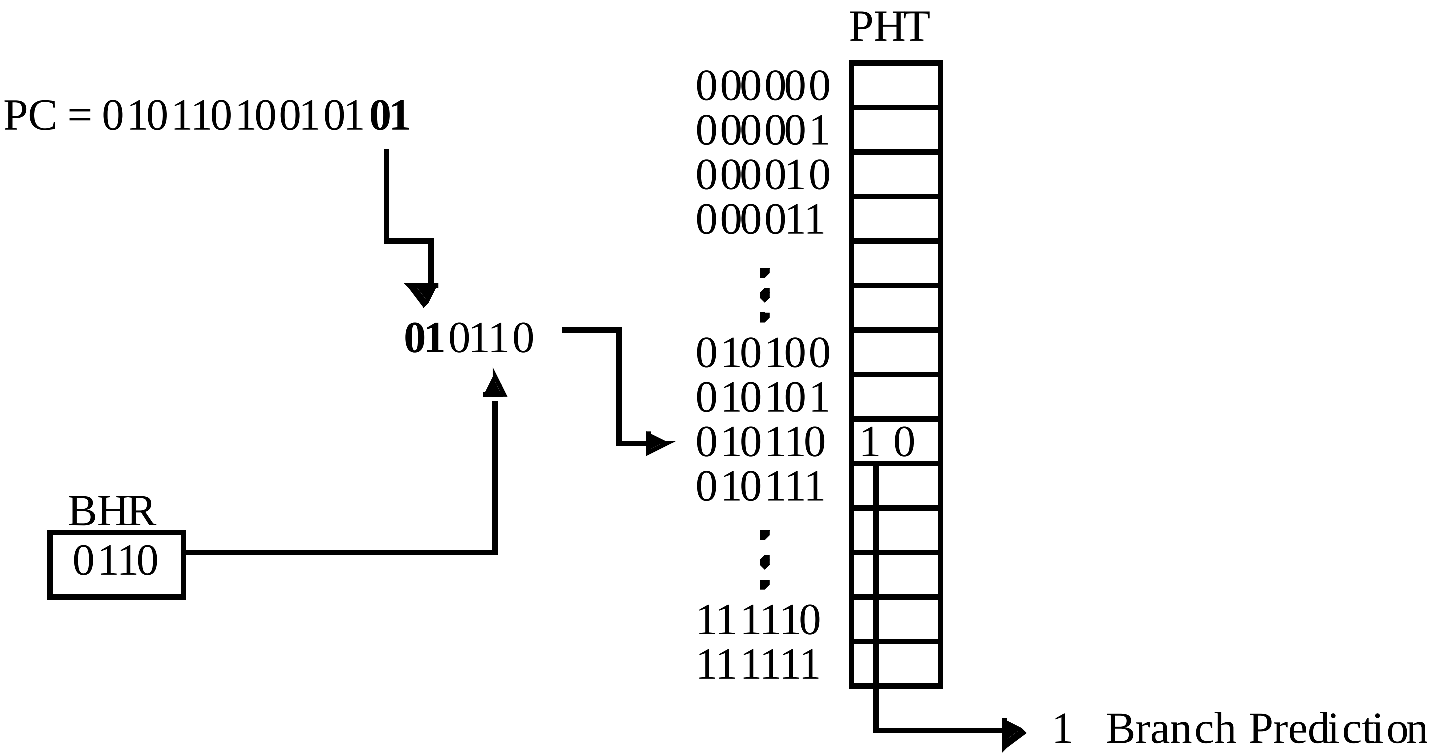
- Main advantages:
  - – Learn branch behavior autonomously
    - No compiler analysis, heuristics, or profiling
  - – Adapt to changing branch behavior
    - Program phase changes branch behavior
- First proposed in 1980
  - – US Patent #4,370,711, Branch predictor using random access memory, James. E. Smith
- Continually refined since then

# Smith Predictor Hardware



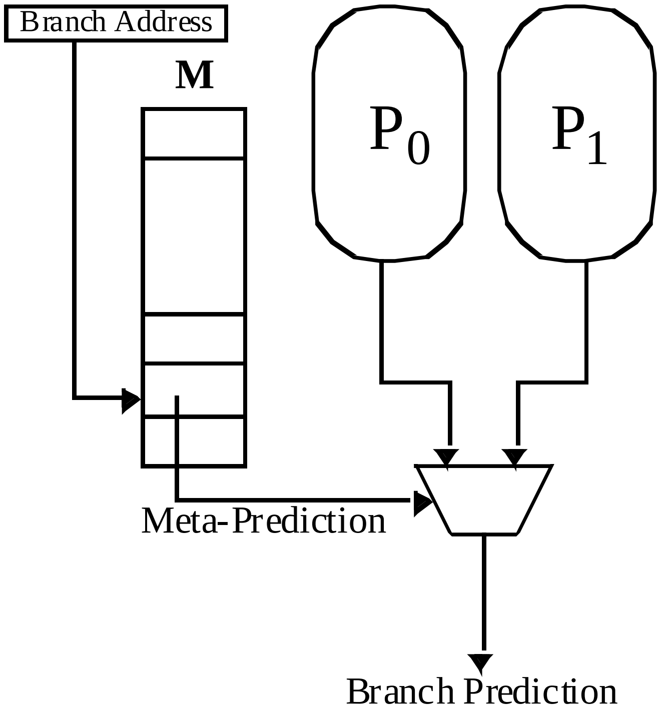
- Jim E. Smith.  A Study of Branch Prediction Strategies.  International Symposium on Computer Architecture, pages 135-148, May 1981
- Widely employed: Intel Pentium, PowerPC 604, PowerPC 620, etc.

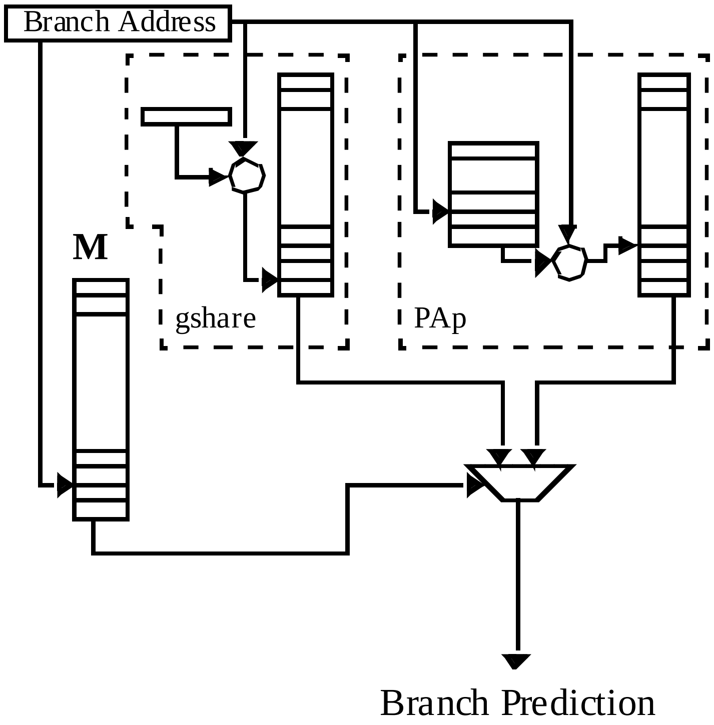
# Two-level Branch Prediction



- BHR adds *global* branch history
  - Provides more context
  - Can differentiate multiple instances of the same static branch
  - Can correlate behavior across multiple static branches

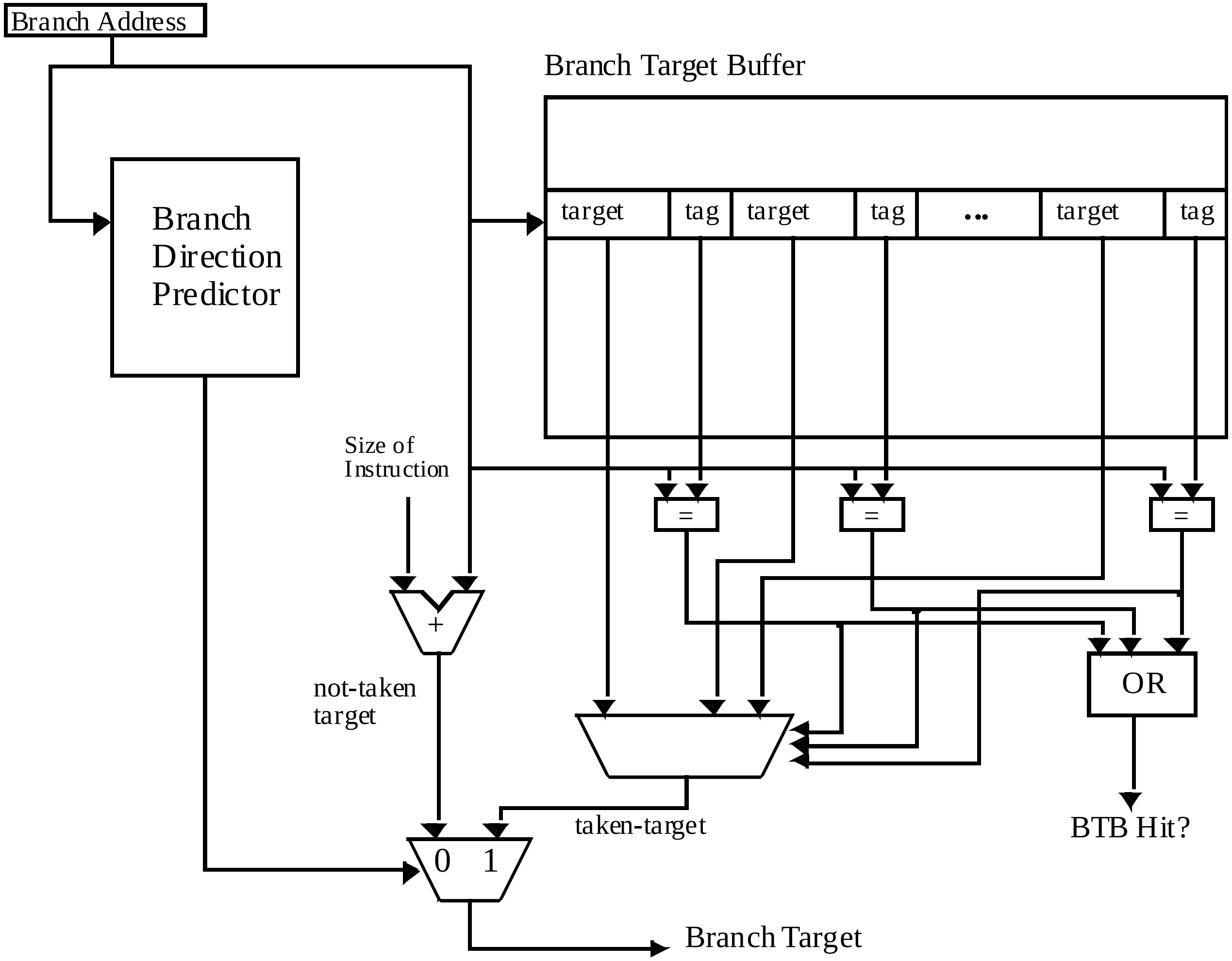
# Combining or Hybrid Predictors





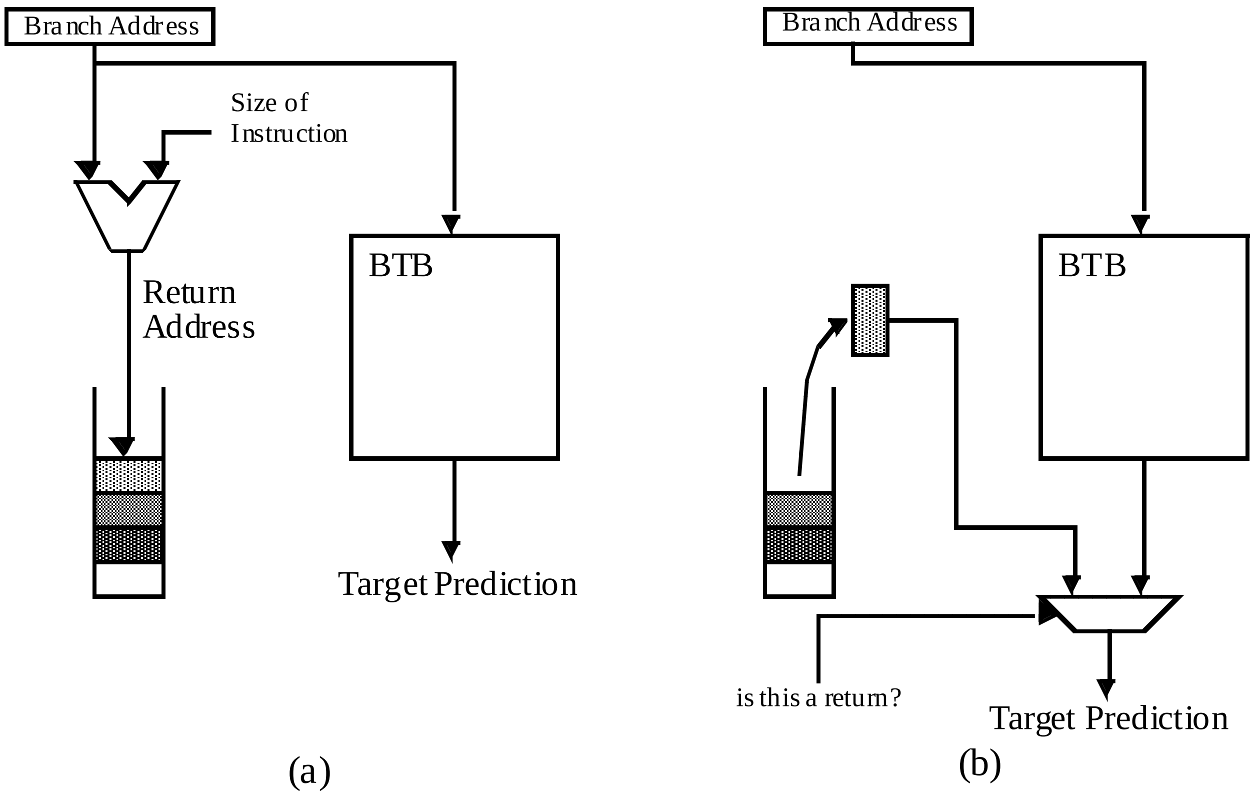
- Select “best” history
- Reduce interference w/partial updates
- Scott McFarling. Combining Branch Predictors. TN-36, Digital Equipment Corporation Western Research Laboratory, June 1993.

# Branch Target Prediction



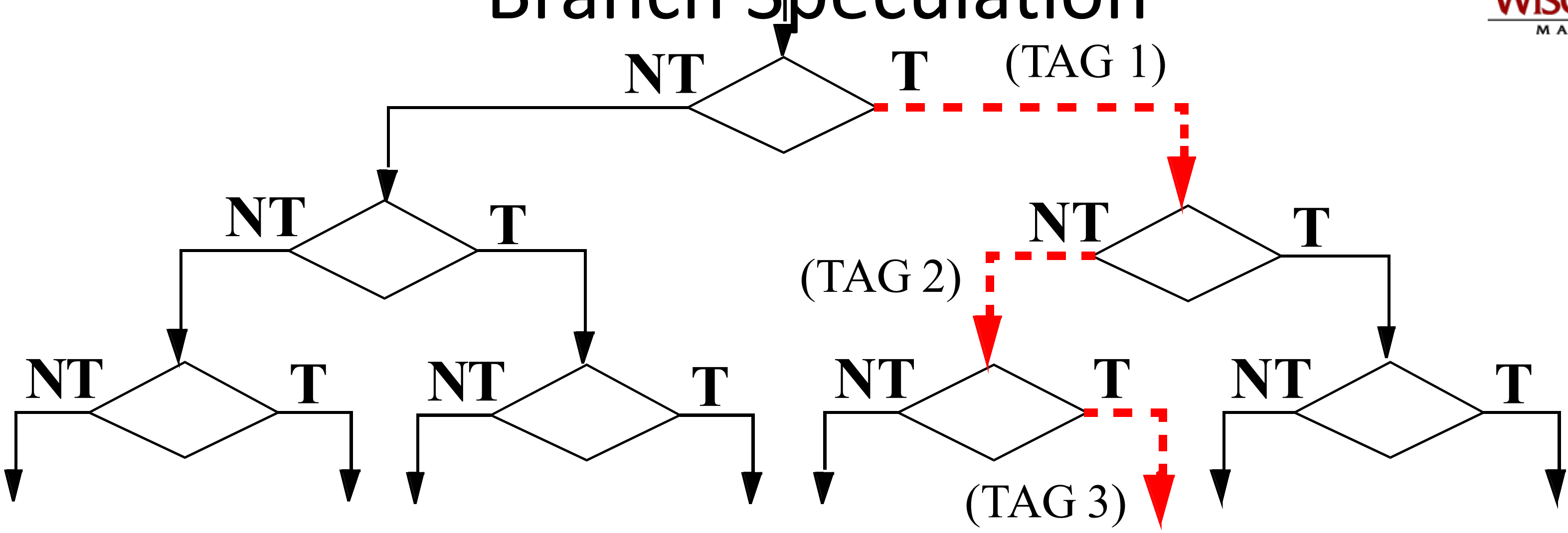
- Partial tags sufficient in BTB

# Return Address Stack



- For each call/return pair:
  - Call: push return address onto hardware stack
  - Return: pop return address from hardware stack

# Branch Speculation

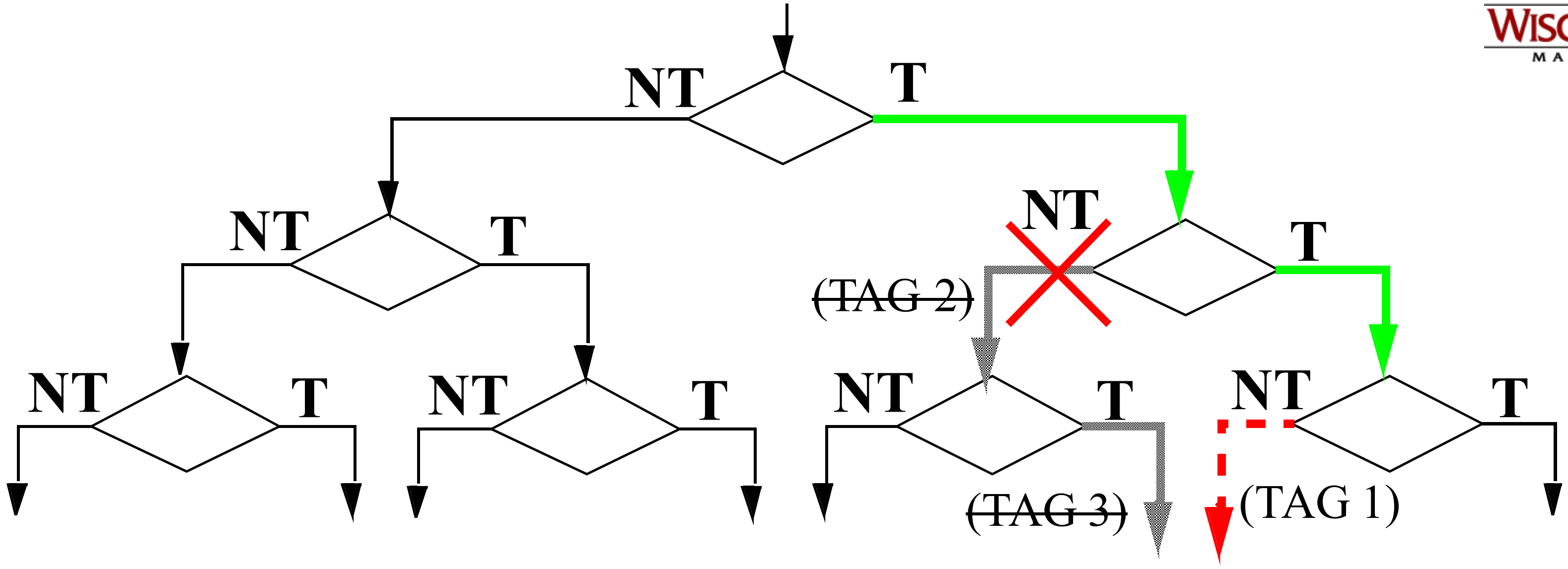


- Leading Speculation
  - Typically done during the Fetch stage
  - Based on potential branch instruction(s) in the current fetch group
- Trailing Confirmation
  - Typically done during the Branch Execute stage
  - Based on the next Branch instruction to finish execution

# Branch Speculation

- Leading Speculation
  1. Tag speculative instructions
  2. Advance branch and following instructions
  3. Buffer addresses of speculated branch instructions
- Trailing Confirmation
  1. When branch resolves, remove/deallocate speculation tag
  2. Permit completion of branch and following instructions

# 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

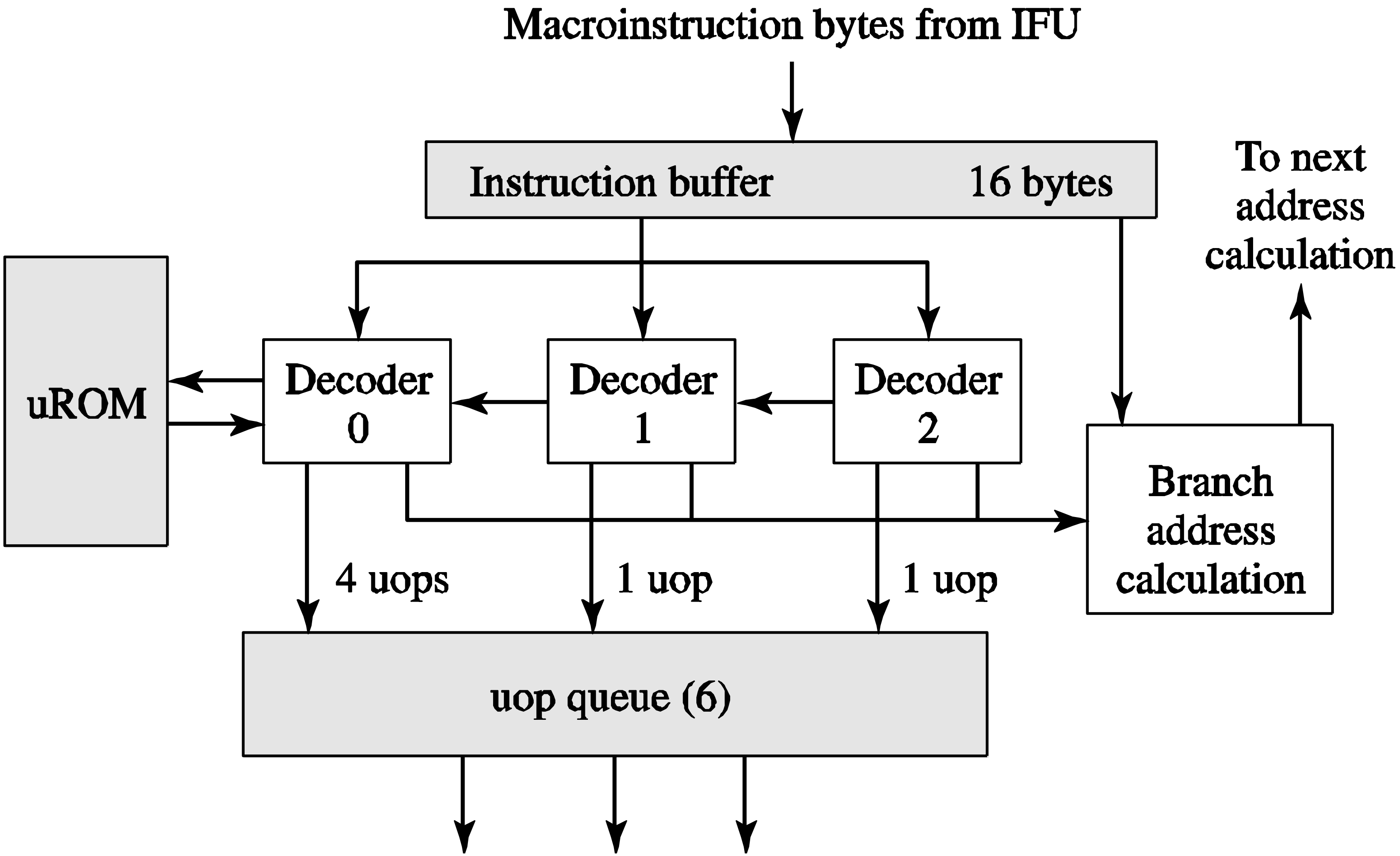
# Summary: Instruction Fetch

- Fetch group alignment
- Target address generation
  - Branch target buffer
  - Return address stack
- Target condition generation
  - Static prediction
  - Dynamic prediction
- Speculative execution
  - Tagging/tracking instructions
  - Recovering from mispredicted branches

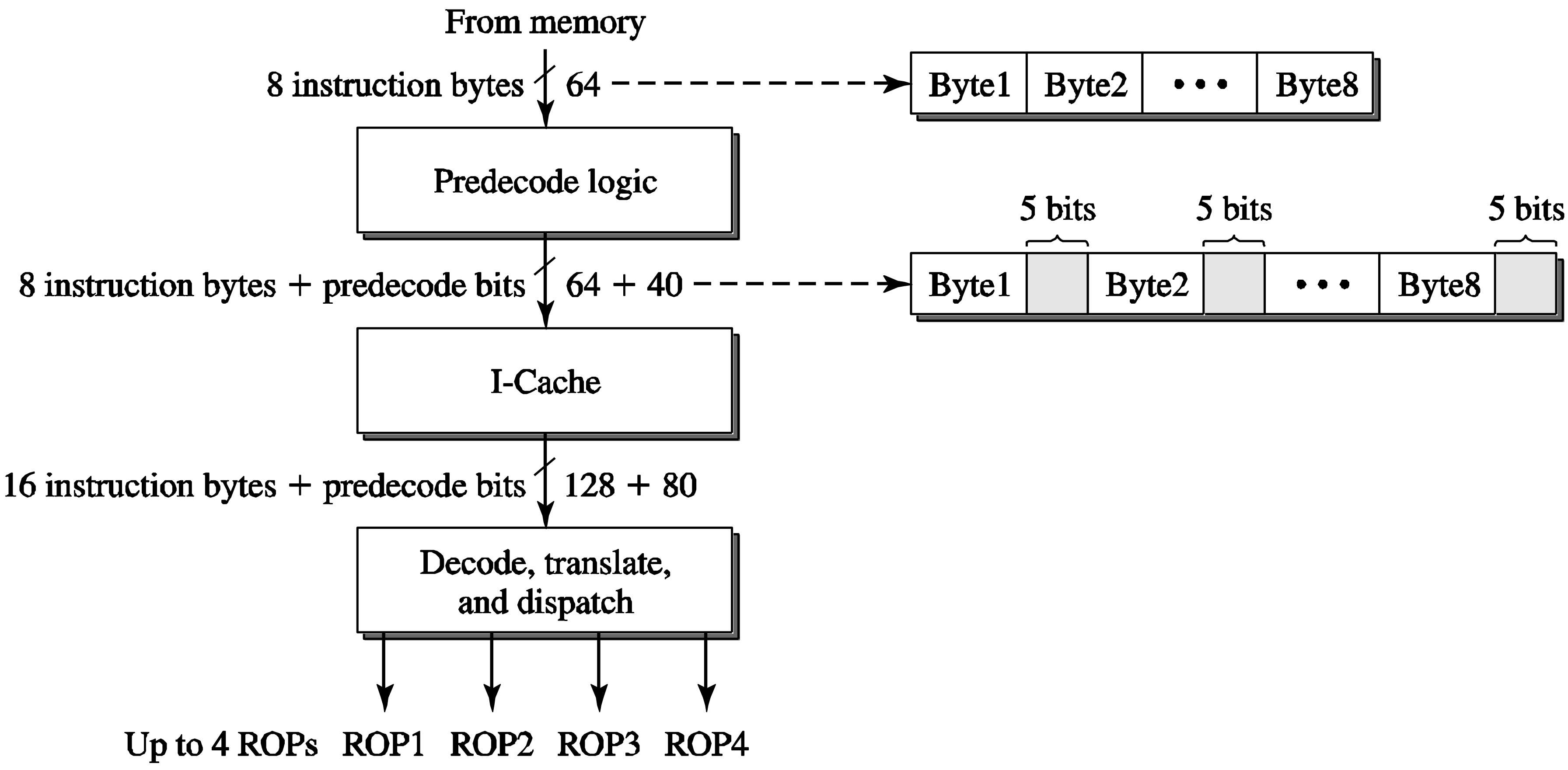
# Issues in Decoding

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

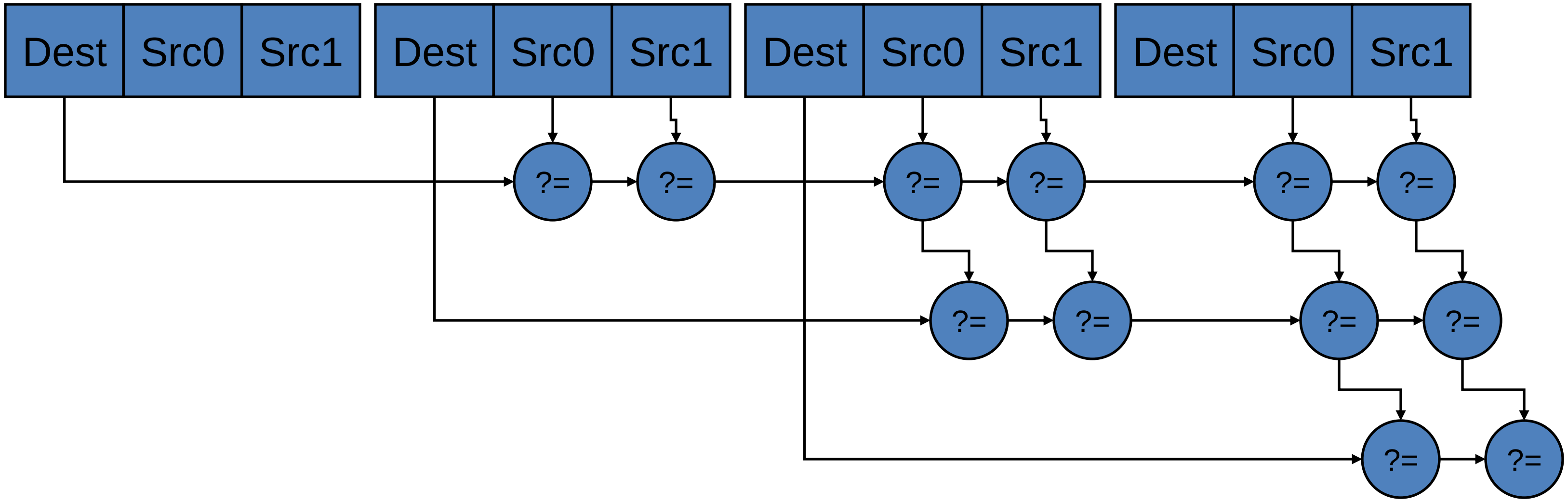
# Pentium Pro Fetch/Decode



# Predecoding in the AMD K5



# Dependence Checking

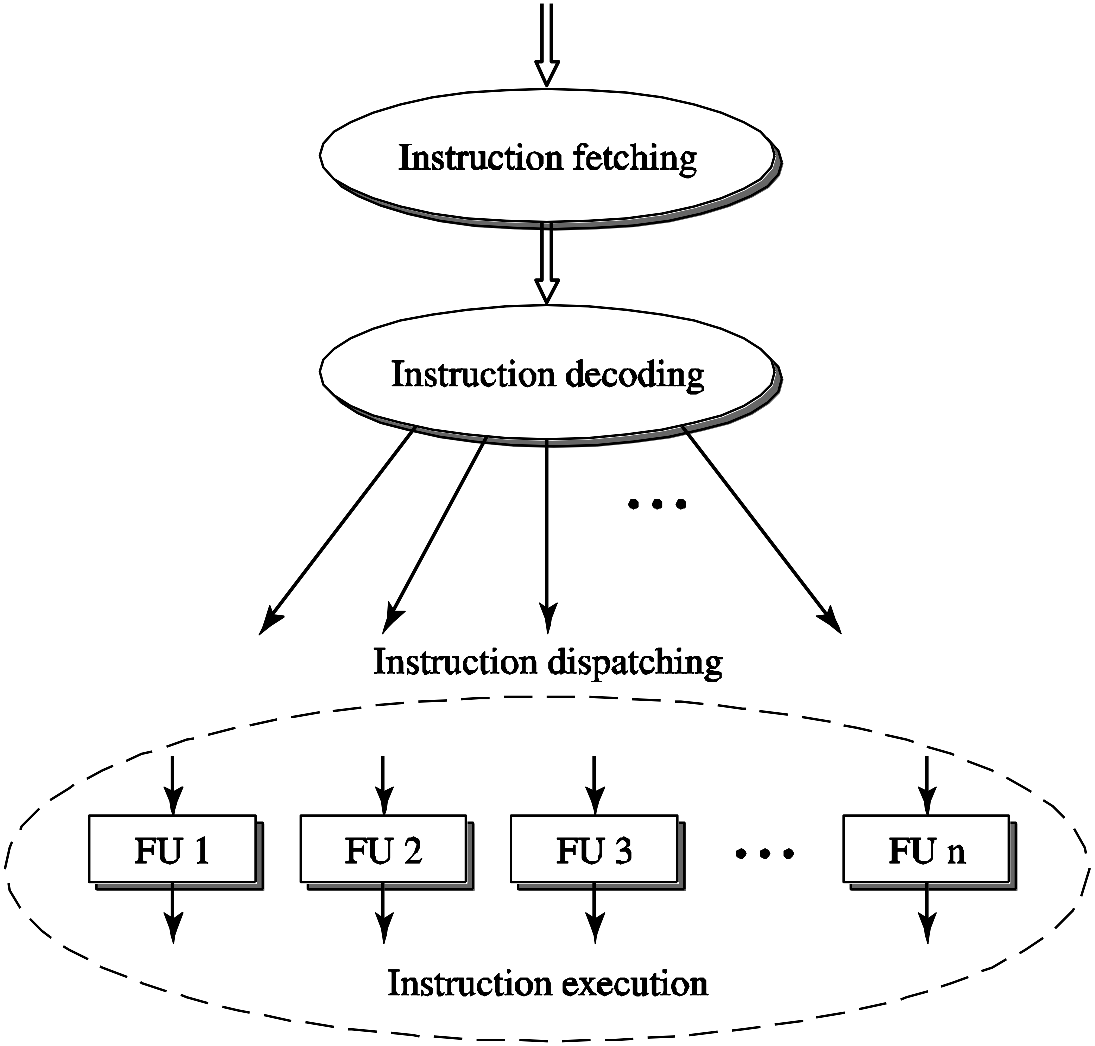


- Trailing instructions in fetch group
  - Check for dependence on leading instructions

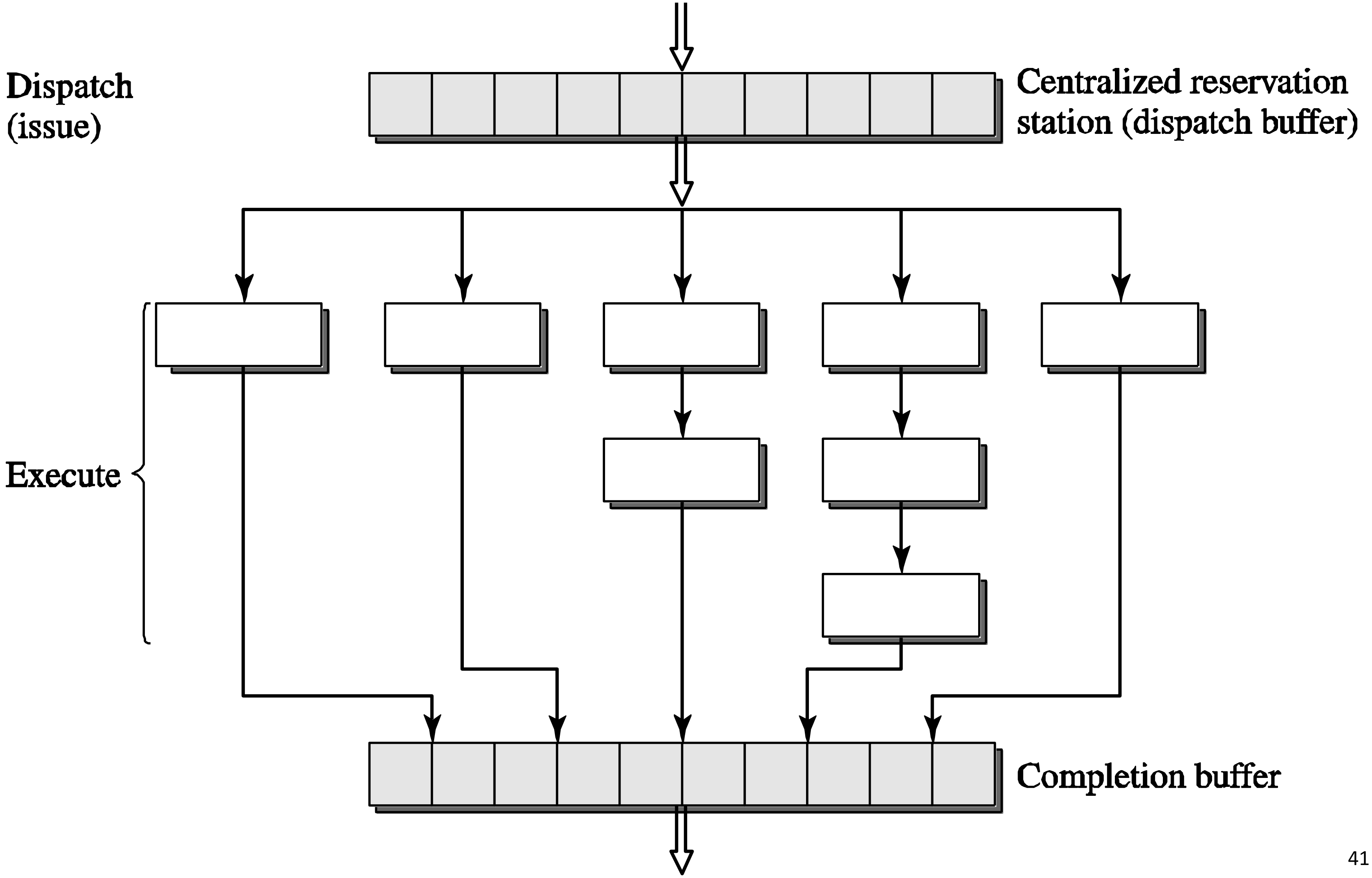
# Instruction Dispatch and Issue

- Parallel pipeline
  - Centralized instruction fetch
  - Centralized instruction decode
- Diversified pipeline
  - Distributed instruction execution

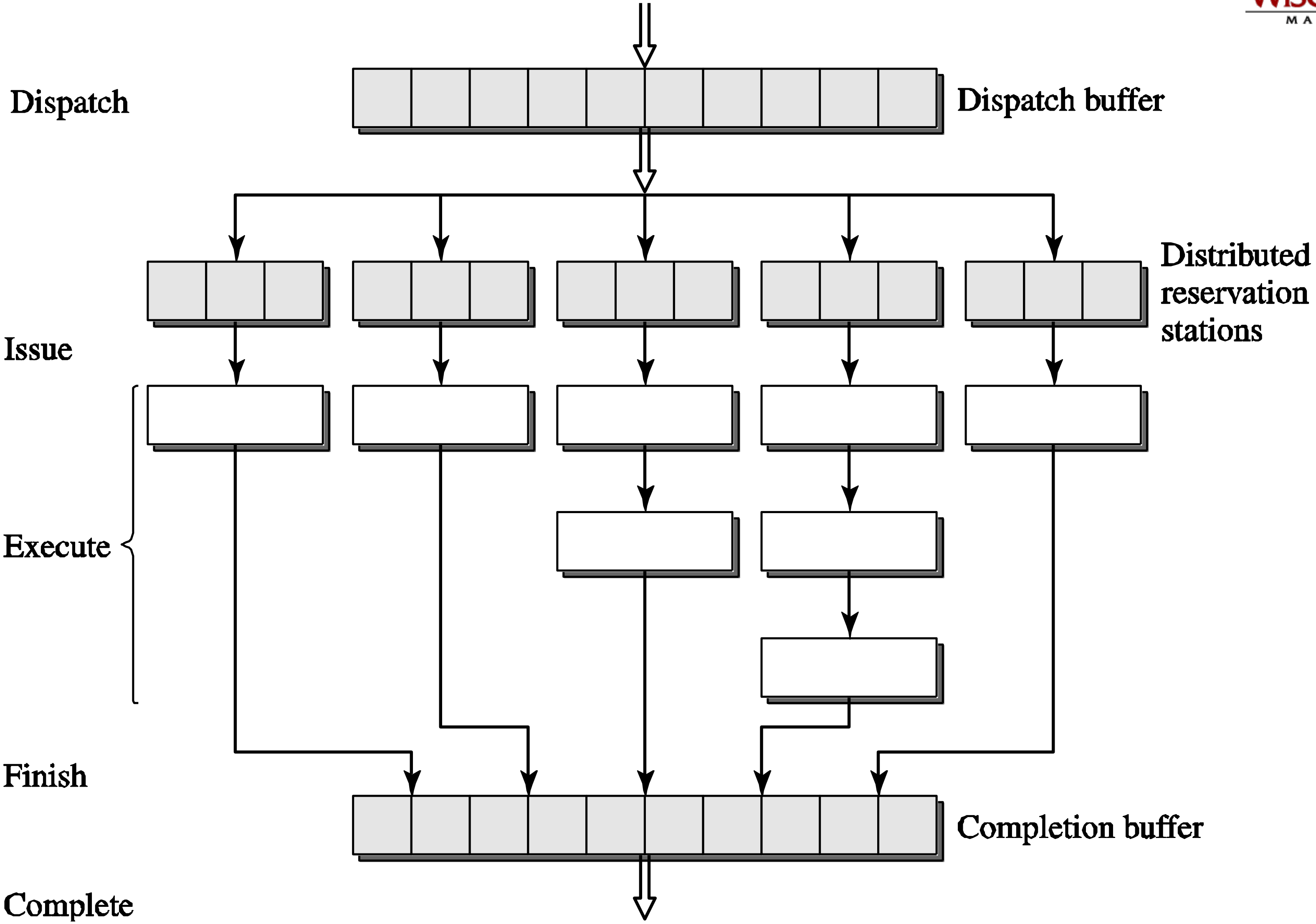
# Necessity of Instruction Dispatch



# Centralized Reservation Station



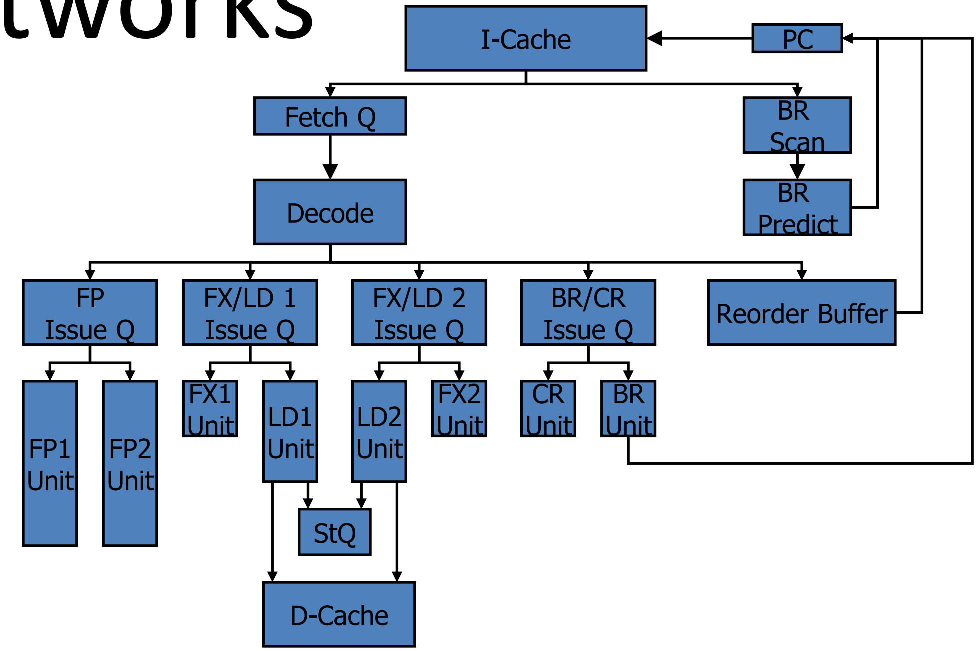
# Distributed Reservation Station



# Issues in Instruction Execution

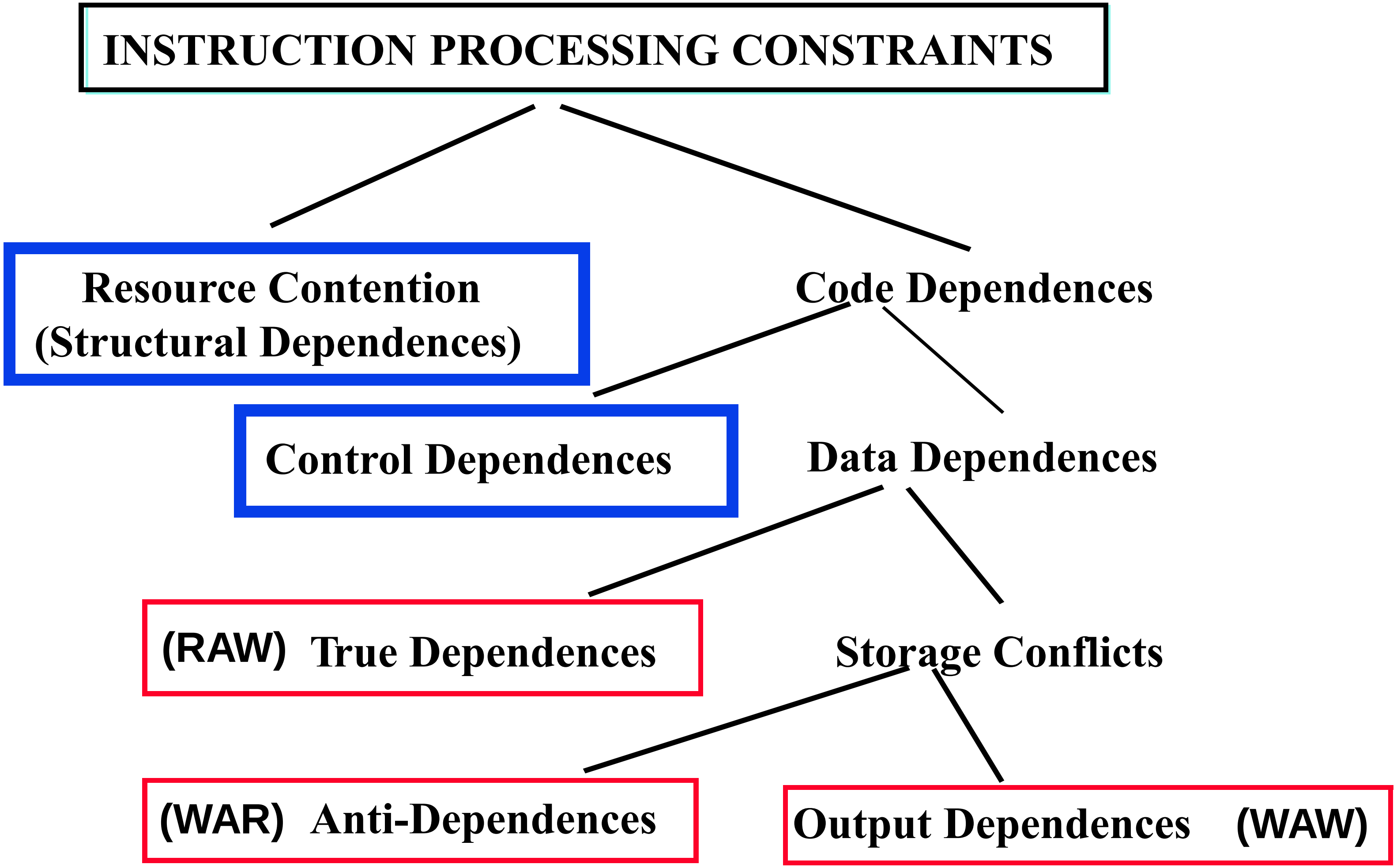
- Parallel execution units
  - Bypassing is a real challenge
- Resolving register data dependences
  - Want out-of-order instruction execution
- Resolving memory data dependences
  - Want loads to issue as soon as possible
- Maintaining precise exceptions
  - Required by the ISA

# 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)

# The Big Picture


INSTRUCTION PROCESSING CONSTRAINTS

Resource Contention (Structural Dependences)

Code Dependences

Control Dependences

Data Dependences

(RAW) True Dependences

Storage Conflicts

(WAR) Anti-Dependences

Output Dependences (WAW)

# 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/all of these can occur
- Solution to all three: register renaming

# Register Renaming: WAR/WAW

- Widely employed (Core i7, Athlon/Phenom, …)
- 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

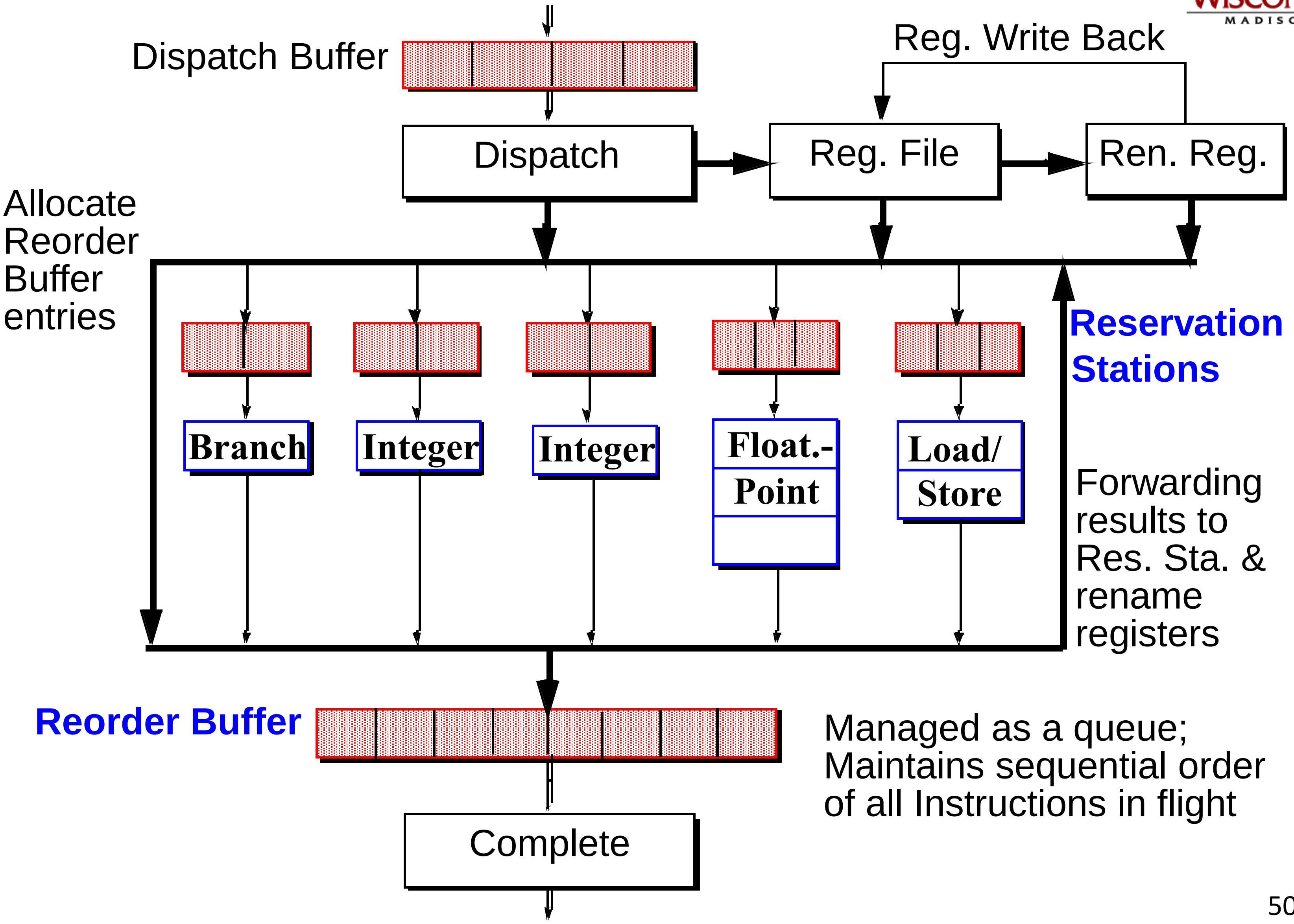
- In order, at dispatch:
  - – Source registers checked to see if “in flight”
    - Register map table keeps track of this
    - If not in flight, can be read from the register file
    - If in flight, look up “rename register” tag (IOU)
  - – Then, allocate new register for register write

```
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 reservation station
  - Wait for rename register tag to trigger issue
- Reservation station enables out-of-order issue
  - Newer instructions can bypass stalled instructions

# "Dataflow Engine" for Dynamic Execution

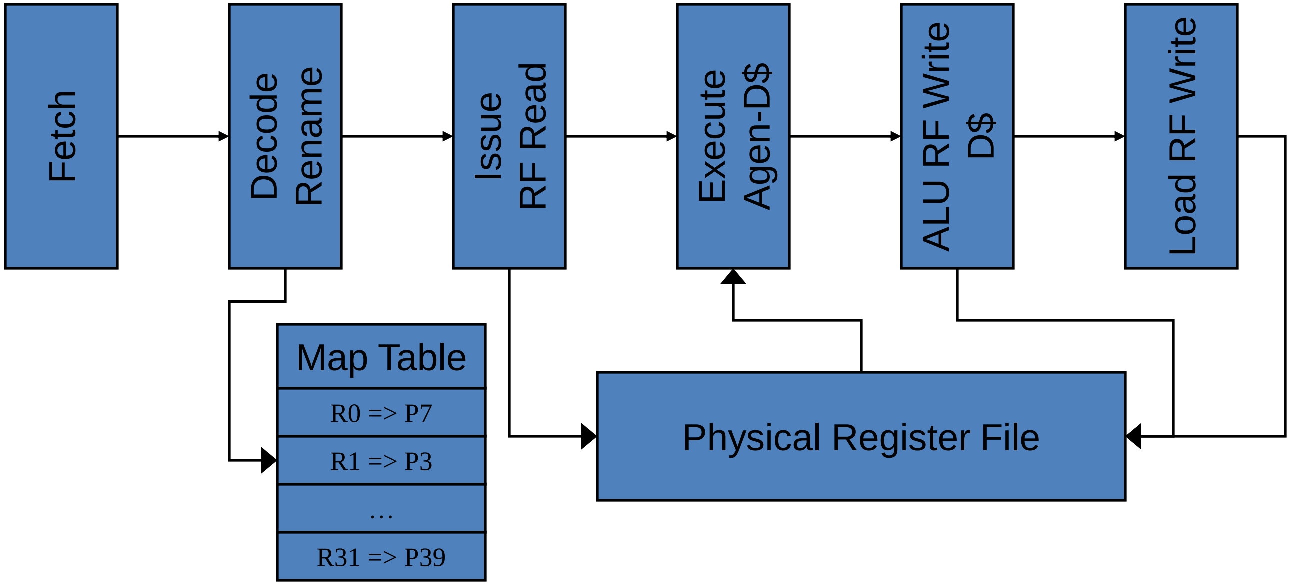


Managed as a queue;
Maintains sequential order
of all Instructions in flight

# Instruction Processing Steps

- **DISPATCH:**
  - **Read operands from Register File (RF) and/or Rename Buffers (RRB)**
  - **Rename destination register and allocate RRF entry**
  - **Allocate Reorder Buffer (ROB) entry**
  - **Advance instruction to appropriate Reservation Station (RS)**
- **EXECUTE:**
  - **RS entry monitors bus for register Tag(s) to latch in pending operand(s)**
  - **When all operands ready, issue instruction into Functional Unit (FU) and deallocate RS entry (no further stalling in execution pipe)**
  - **When execution finishes, broadcast result to waiting RS entries, RRB entry, and ROB entry**
- **COMPLETE:**
  - **Update architected register from RRB entry, deallocate RRB entry, and if it is a store instruction, advance it to Store Buffer**
  - **Deallocate ROB entry and instruction is considered architecturally completed**

# Physical Register File



- Used in MIPS R10000, Pentium 4, AMD Bulldozer
- All registers in one place
  - Always accessed right before EX stage
  - No copying to real register file at commit

# Managing Physical Registers

| Map Table |
|---|
| R0 => P7 |
| R1 => P3 |
| … |
| R31 => P39 |

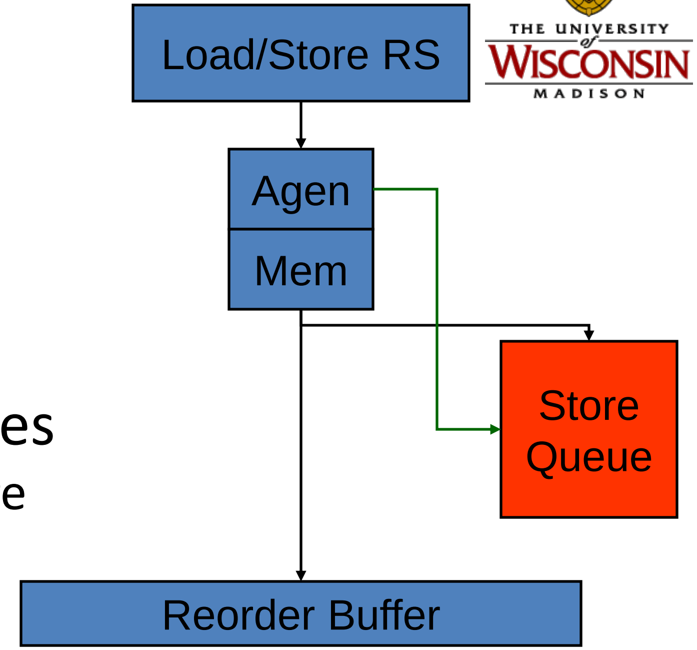
| | |
|---|---|
| Add R3 <= R2 + R1 | P32 <= P2 + P1 |
| Sub R4 <= R3 + R1 | P33 <= P32 + P1 |
| … | |
| … | |
| And R3 <= R4 & R2 | P35 <= P33 + P2 |

Release P32 (previous R3) when this instruction completes execution

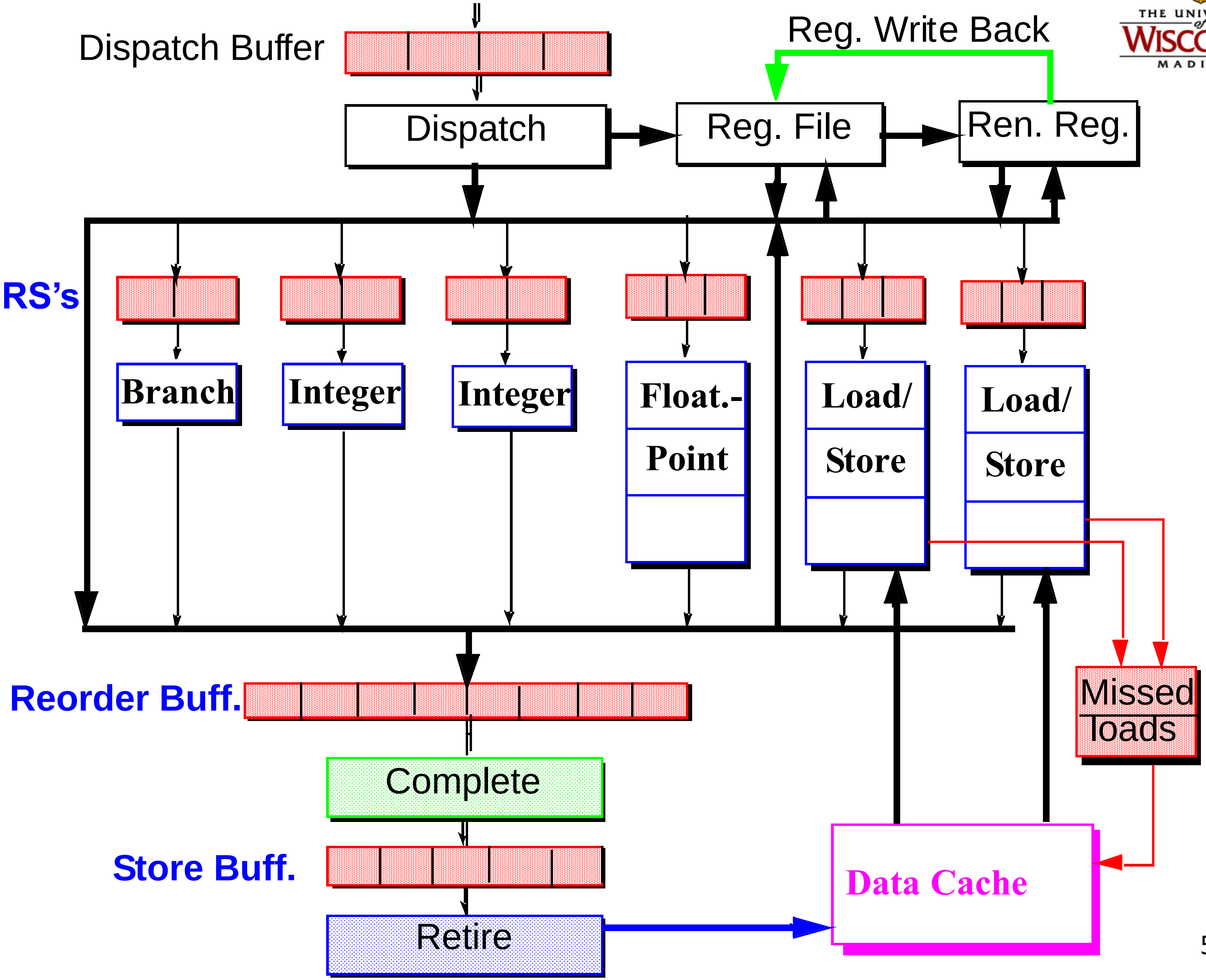
- What to do when all physical registers are in use?
  - Must release them somehow to avoid stalling
  - Maintain *free list* of "unused" physical registers
- Release when no more uses are possible
  - Sufficient: next write commits

# Memory Data Dependences

- WAR/WAW: stores commit in order
  - Hazards not possible. Why?
- RAW: loads must check pending stores
  - Store queue keeps track of pending store addresses
  - Loads check against these addresses
  - Similar to register bypass logic
  - Comparators are 32 or 64 bits wide (address size)
- Major source of complexity in modern designs
  - Store queue lookup is position-based
  - What if store address is not yet known?



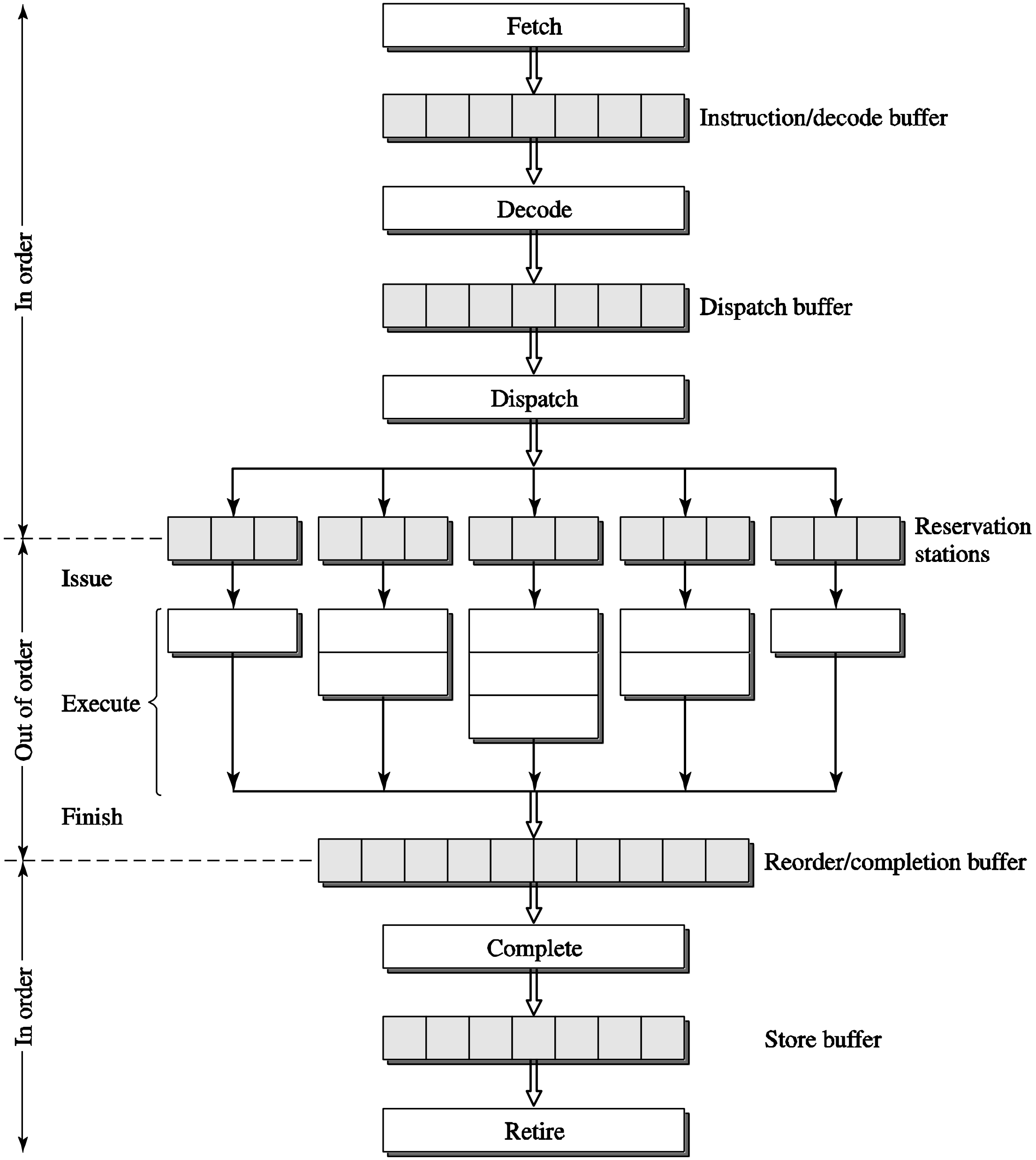
# Increasing Memory Bandwidth



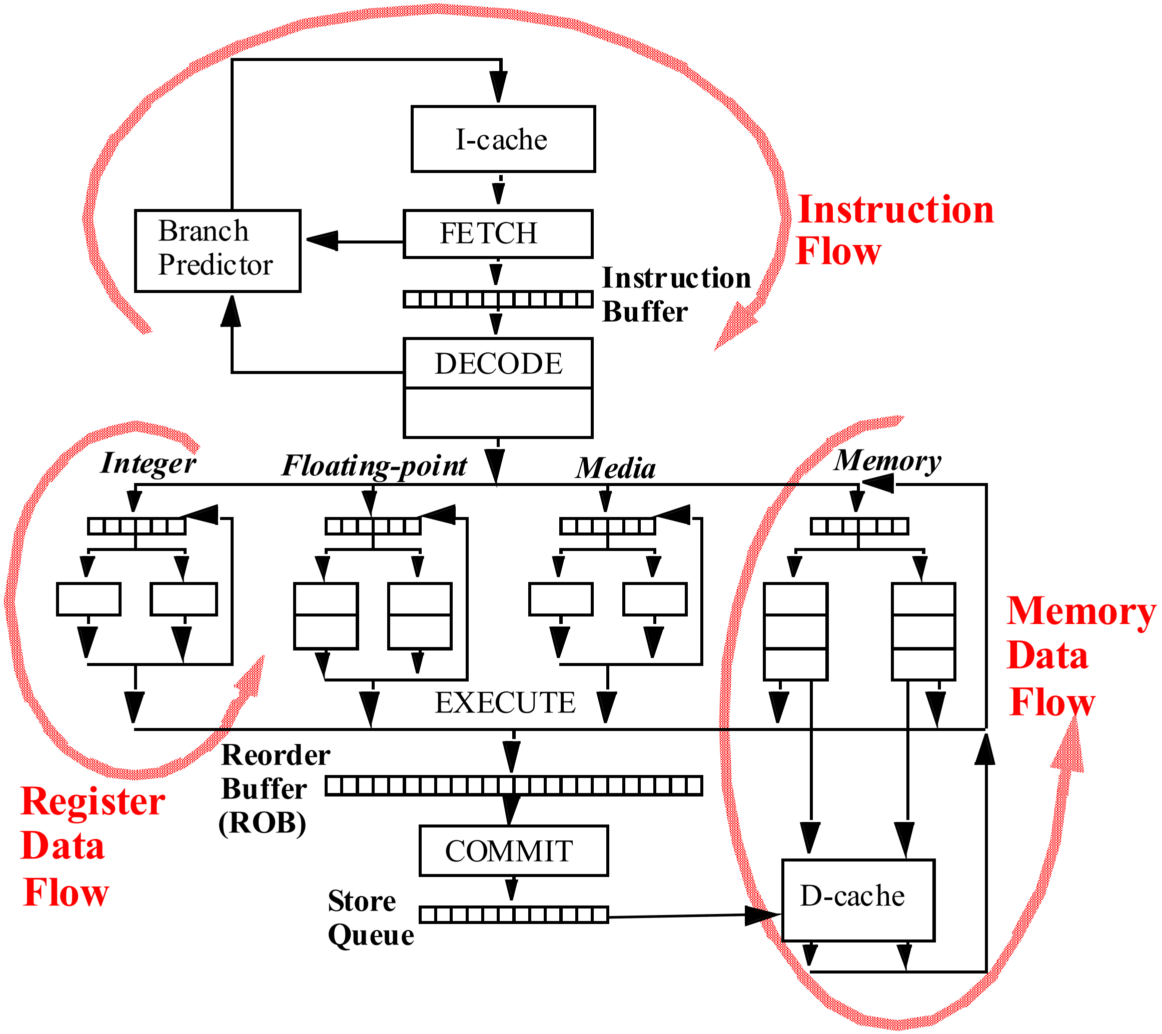
# Issues in Completion/Retirement

- 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

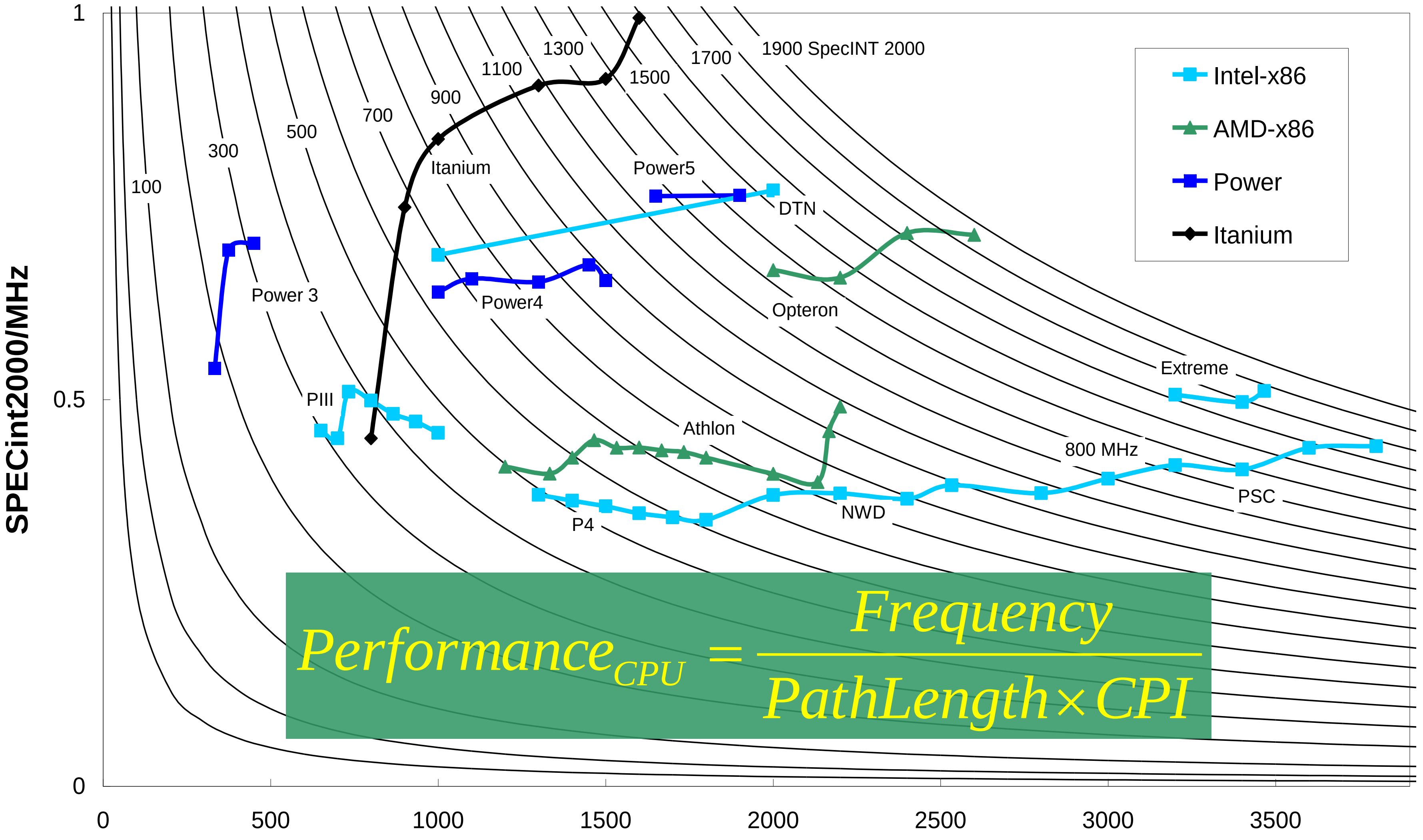
# A Dynamic Superscalar Processor



# Superscalar Summary



# Landscape of Microprocessor Families



$$Performance_{CPU} = \frac{Frequency}{PathLength \times CPI}$$

[John DeVale & Bryan Black, 2005]

** Data source www.spec.org

# Superscalar Summary

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