18-600 Foundations of Computer Systems

Lecture 9: "Modern Superscalar Out-of-Order Processors"

John P. Shen & Gregory Kesden
September 27, 2017

Lecture #7 – Processor Architecture & Design
Lecture #8 – Pipelined Processor Design
Lecture #9 – Superscalar O3 Processor Design

➢ Required Reading Assignment:
- **Chapter 4 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.**

➢ Recommended Reading Assignment:
- ❖ **Chapter 5 of Shen and Lipasti (SnL).**

Electrical & Computer ENGINEERING

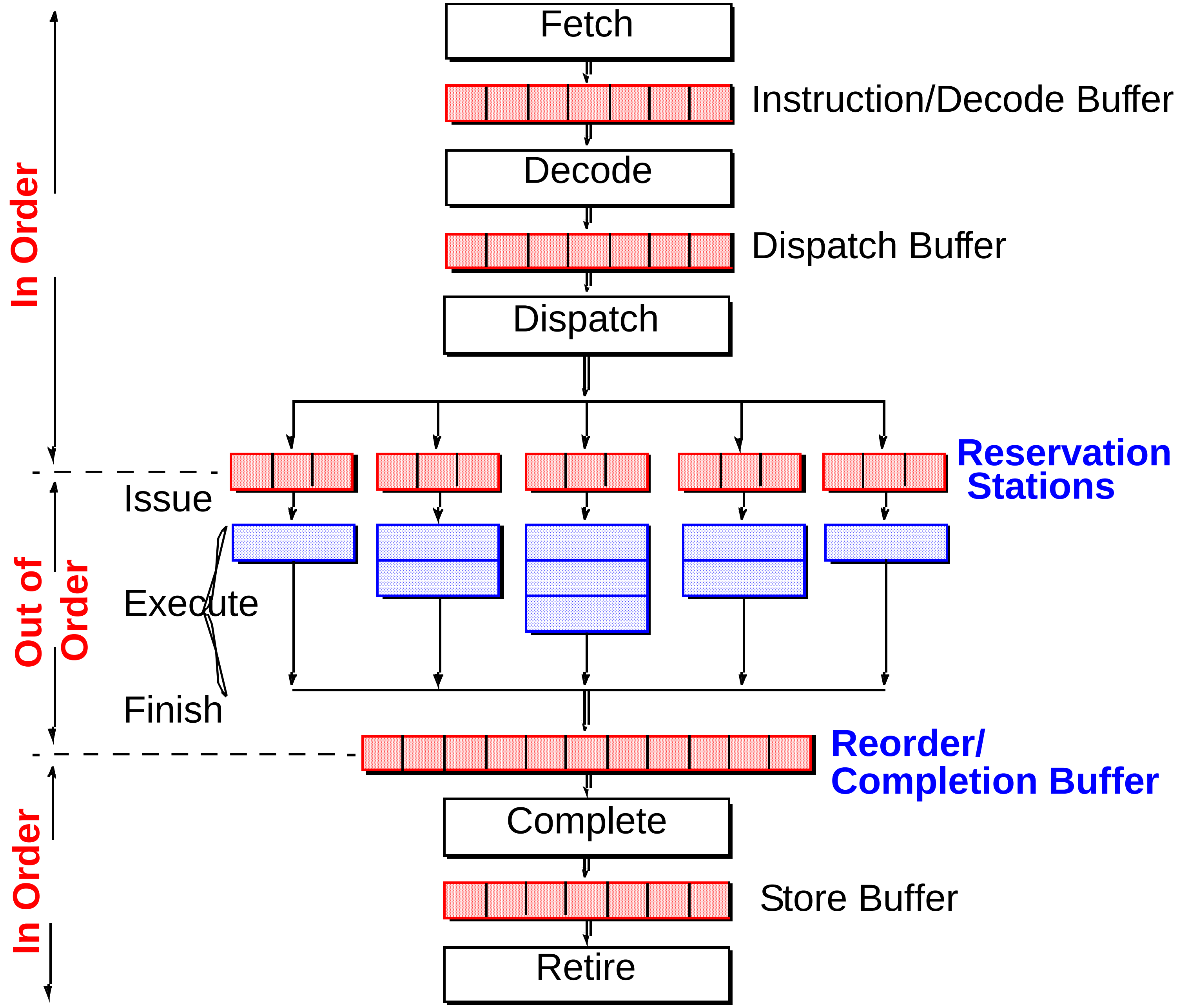
18-600 Foundations of Computer Systems

Lecture 9: "Modern Superscalar Out-of-Order Processors"

A. Instruction Flow Techniques
   a. Control Flow Prediction
   b. Dynamic Branch Prediction
B. Register Data Flow Techniques
   a. Resolving Anti and Output Dependencies
   b. Resolving True Dependencies
   c. Dynamic Out-of-Order Execution
C. Memory Data Flow Techniques
   a. Memory Data Dependencies
   b. Load Bypassing & Load Forwarding

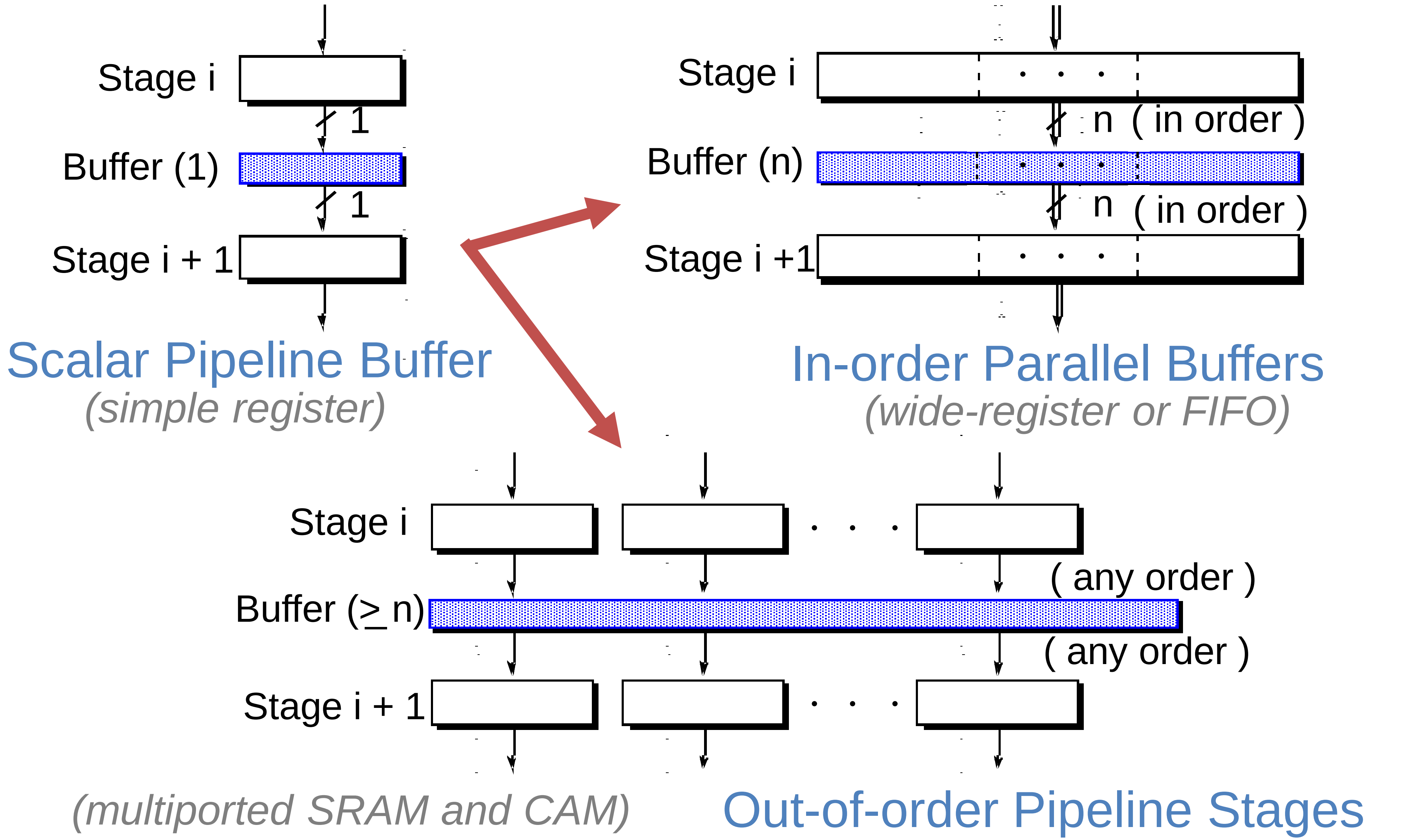
Electrical & Computer ENGINEERING

Modern Superscalar Processor Organization

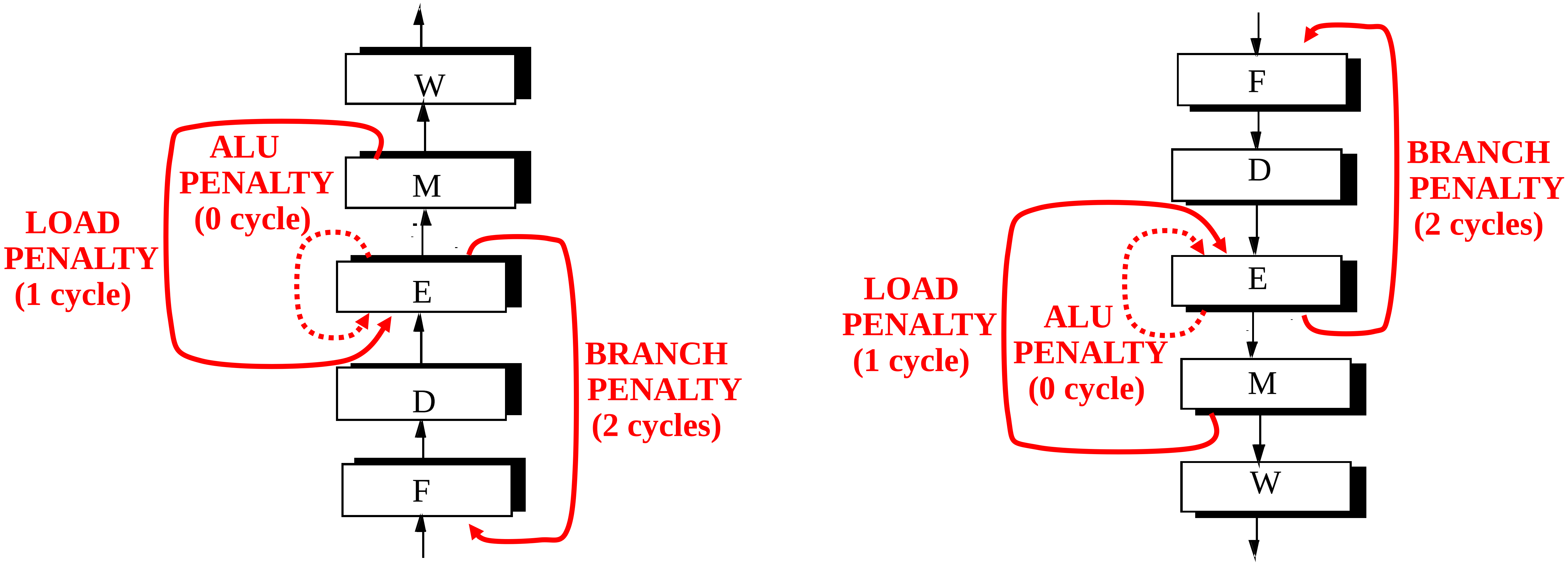


- Buffers provide decoupling
- In OOO designs they also facilitate (re-)ordering
- More details on specific buffers to follow

Designs of Inter-stage Buffers



3 Major Penalty Loops of Pipelined Processors

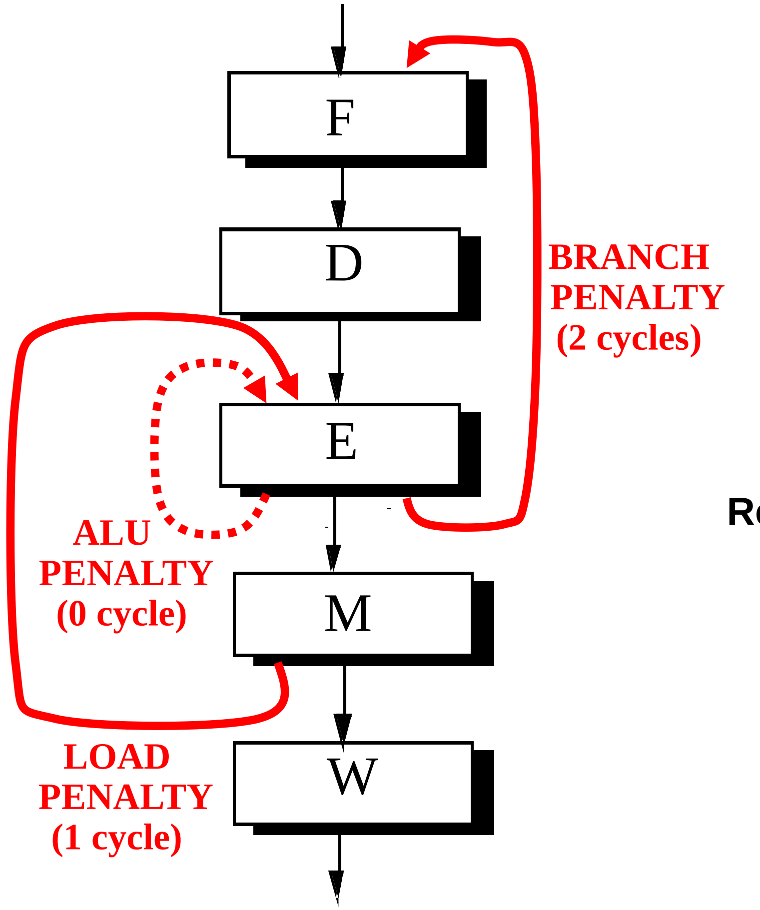


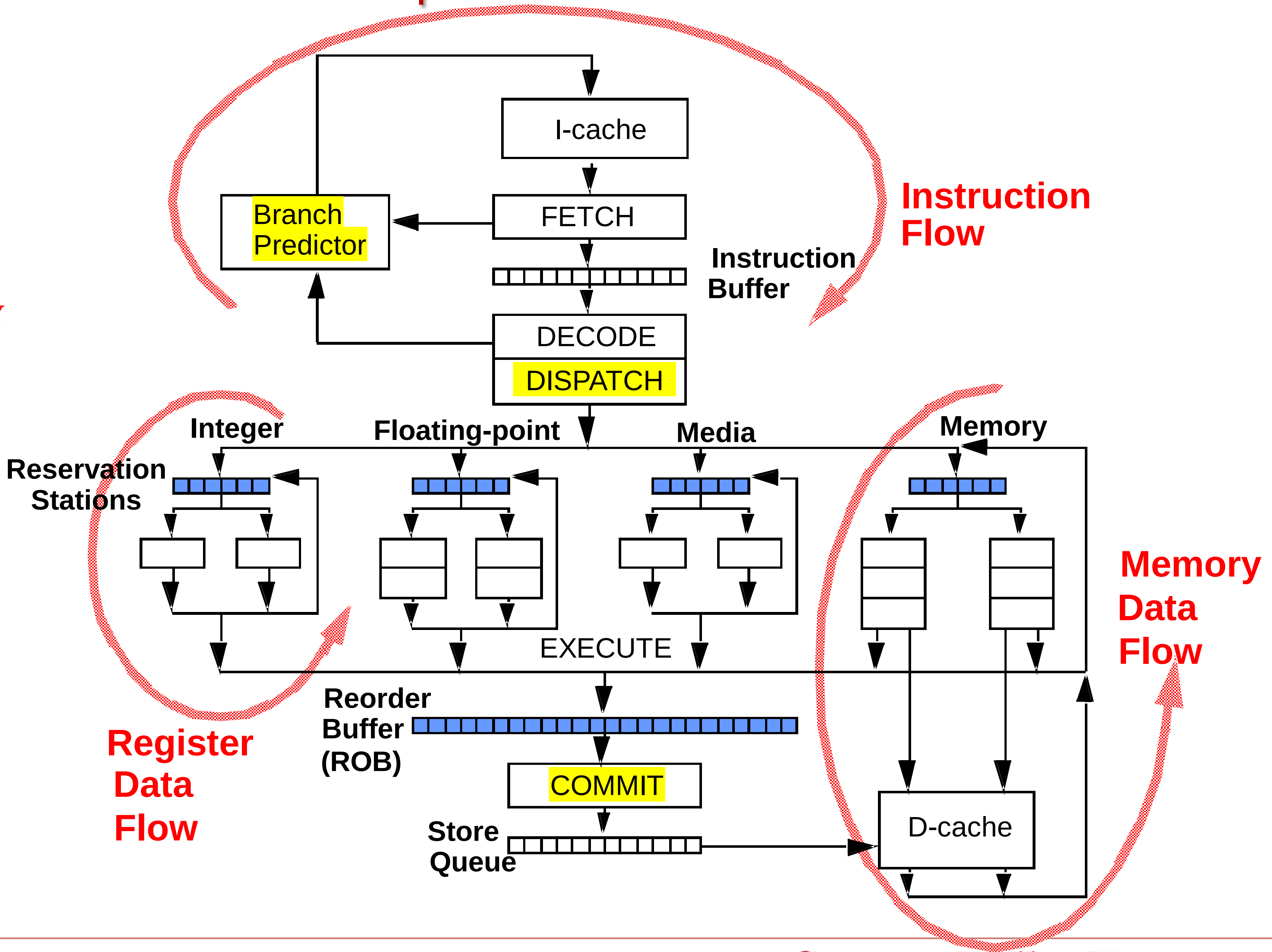
Performance Objective: Reduce CPI as close to 1 as possible.

Best Possible for Real Programs is as Low as CPI = 1.15.

CAN WE DO BETTER? … CAN WE ACHIEVE IPC > 1.0?

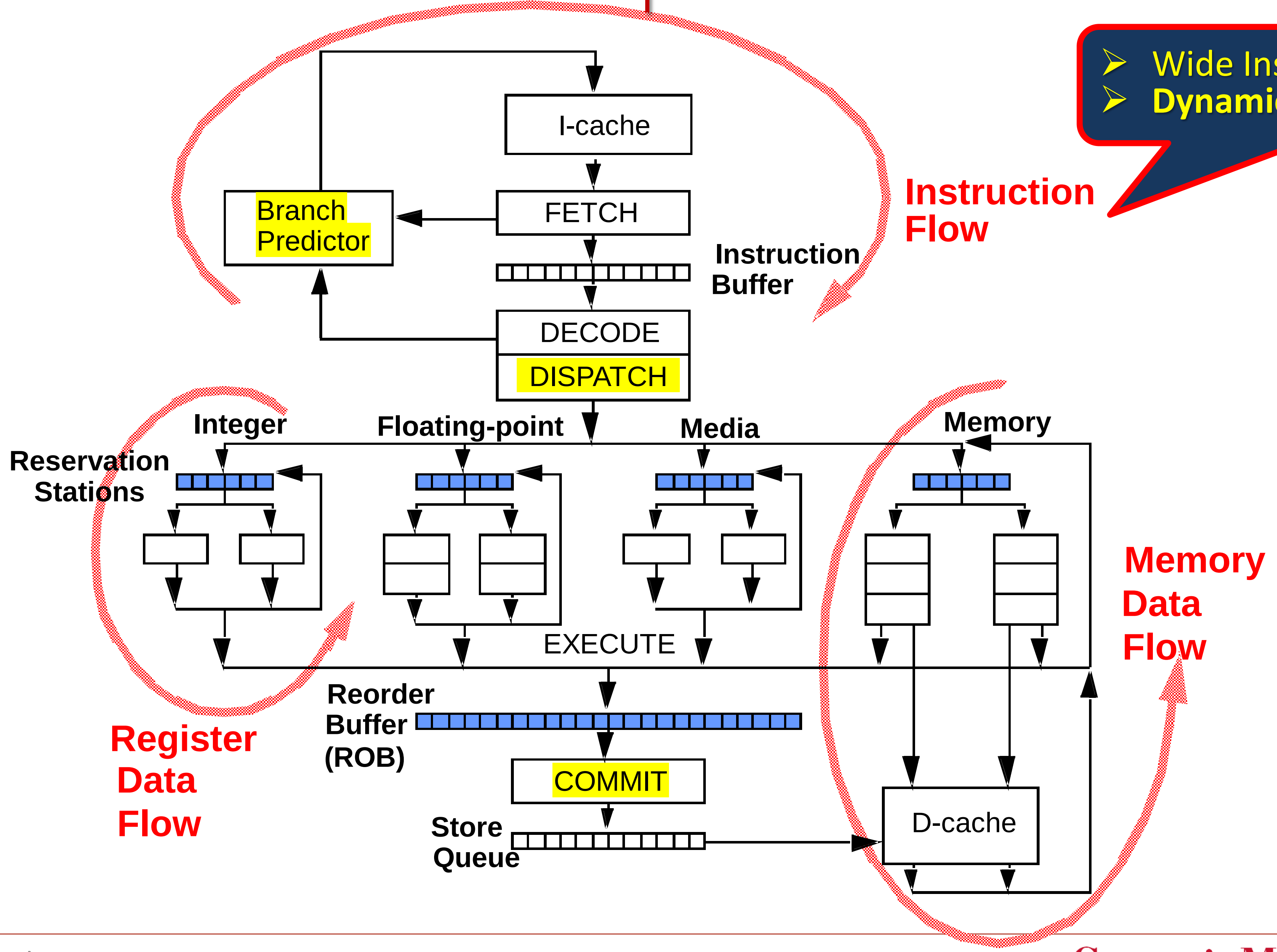
Three Impediments to Superscalar Performance





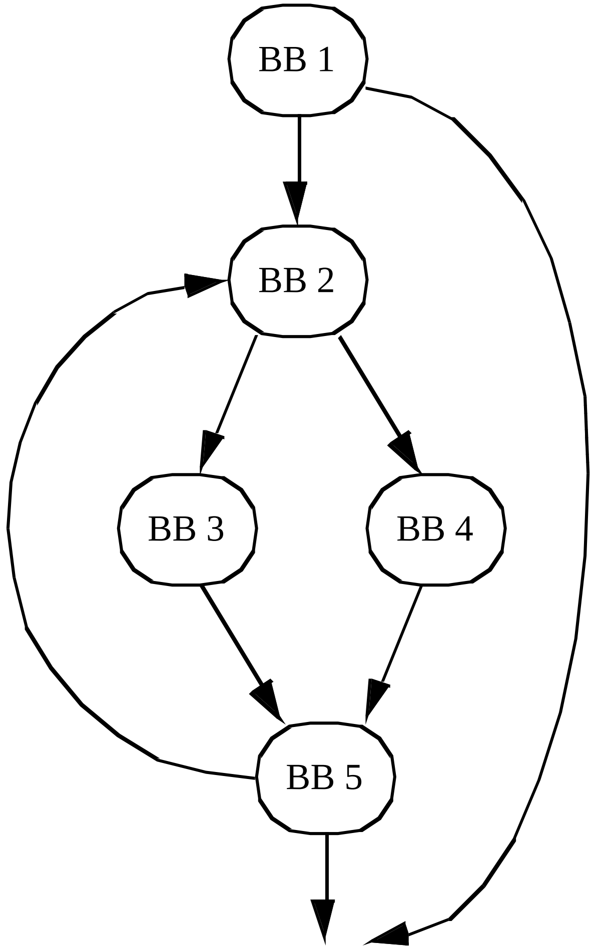
Three Flow Paths of Superscalar Processors

- Wide Instruction Fetching
- **Dynamic Branch Prediction**



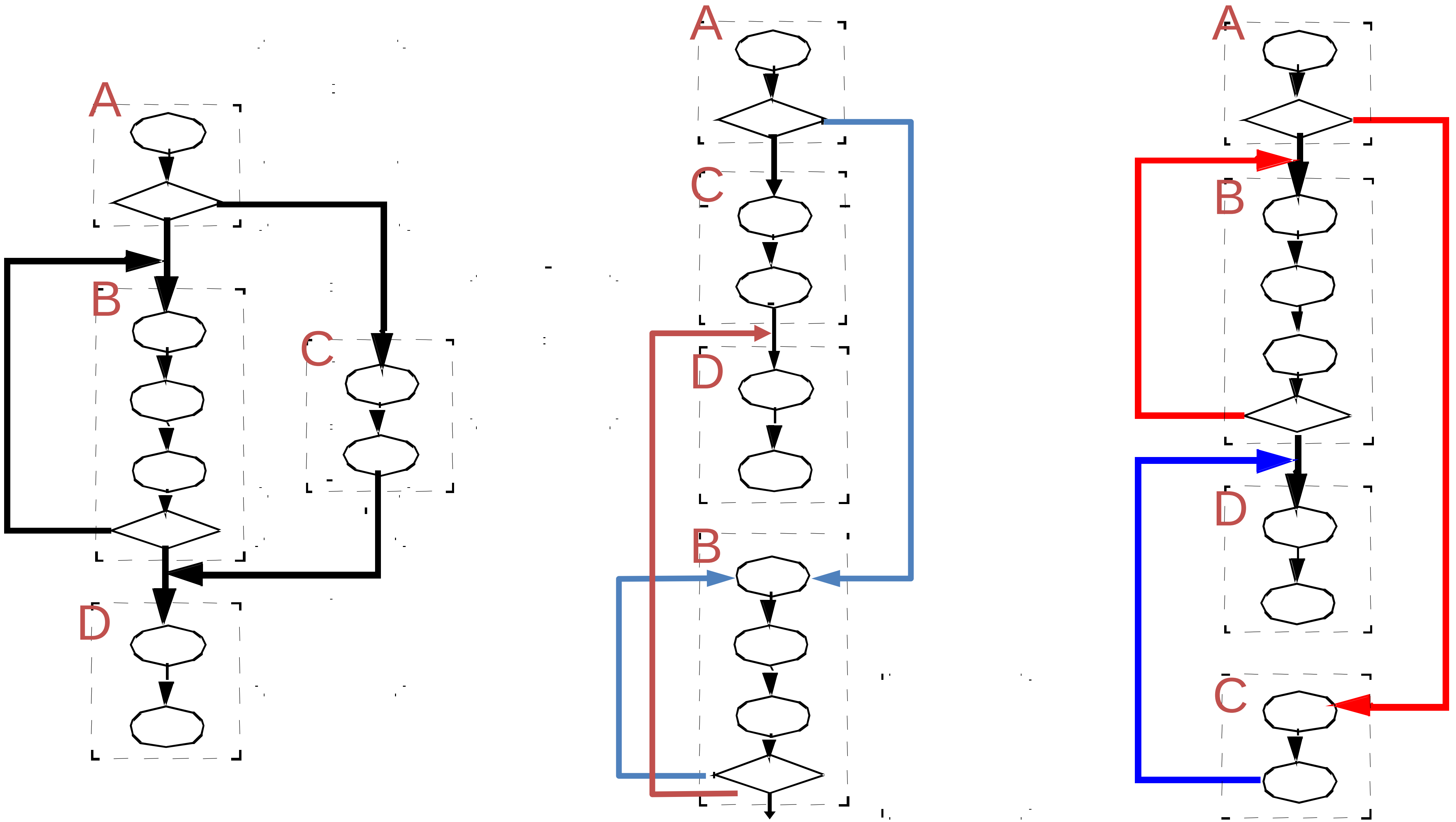
Control Flow Graph (CFG)

- Your program is actually a control flow graph
  - Shows possible paths of control flow through basic blocks
- Control Dependence
  - Node *X* is control dependent on Node *Y* if the computation in *Y* determines whether *X* executes




```
main:
        addi r2, r0, A      |
        addi r3, r0, B      |
        addi r4, r0, C      |   BB 1
        addi r5, r0, N      |
        add  r10,r0, r0     |
        bge  r10,r5, end    |
loop:
        lw   r20, 0(r2)     |
        lw   r21, 0(r3)     |   BB 2
        bge  r20,r21,T1     |
        sw   r21, 0(r4)     |   BB 3
        b    T2             |
T1:
        sw   r20, 0(r4)     |   BB 4
T2:
        addi r10,r10,1      |
        addi r2, r2, 4      |
        addi r3, r3, 4      |   BB 5
        addi r4, r4, 4      |
        blt  r10,r5, loop   |
end:
```

Mapping CFG to Linear Instruction Sequence



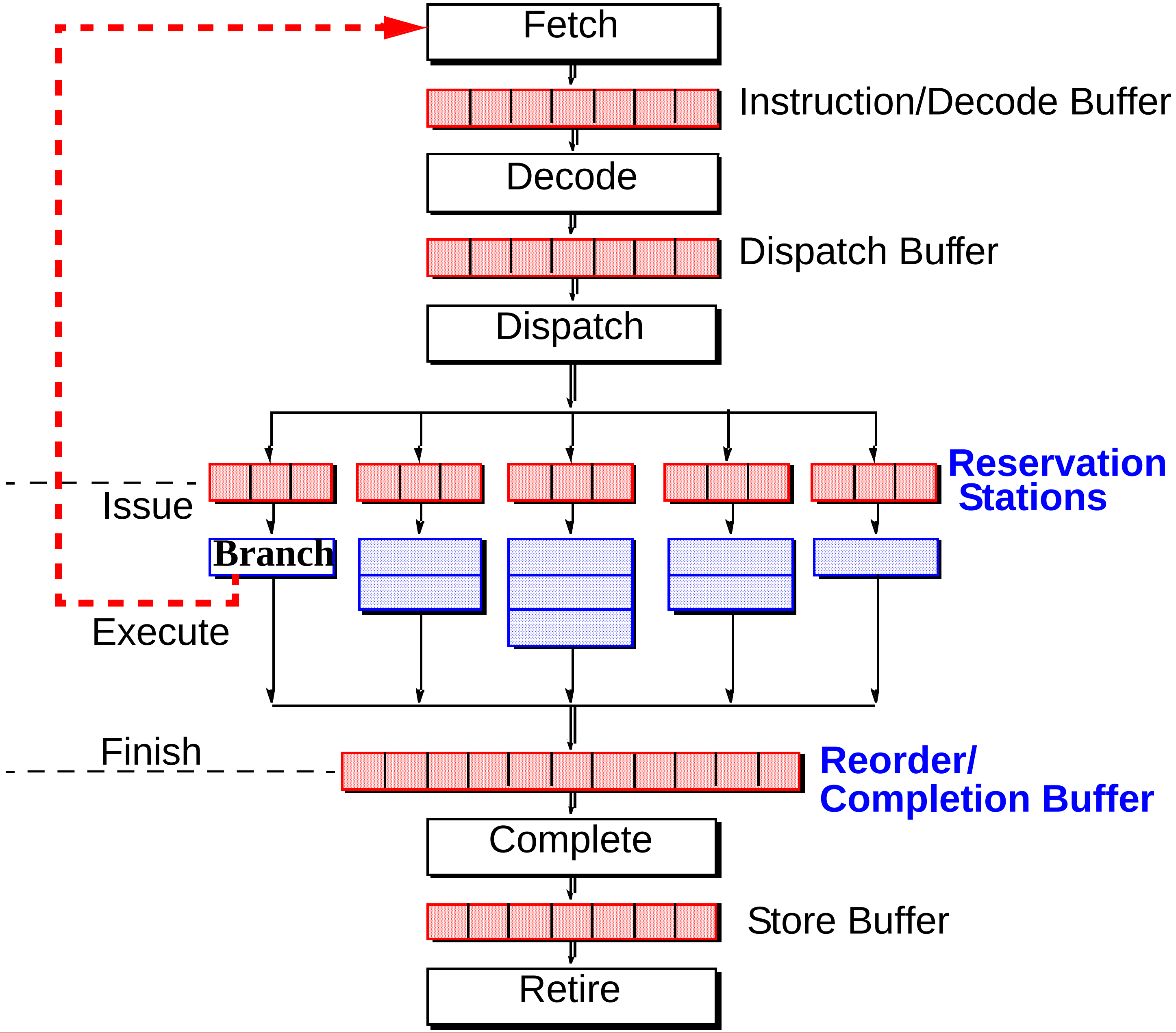
Branch Types and Implementation

- Types of Branches
  - Conditional or Unconditional?
  - Subroutine Call (aka Link), needs to save PC?
  - How is the branch target computed?
    - Static Target e.g. immediate, PC-relative
    - Dynamic targets e.g. register indirect
- Conditional Branch Architectures
  - Condition Code "*N-Z-C-V*" *e.g. PowerPC*
  - General Purpose Register *e.g. Alpha, MIPS*
  - Special Purposes register *e.g. Power's Loop Count*

What's So Bad About Branches?

- Robs instruction fetch bandwidth and ILP
  - Use up execution resources
  - Fragmentation of I-cache lines
  - Disruption of sequential control flow
    - Need to determine branch direction (conditional branches)
    - Need to determine branch target
- Example:
  - We have a N-way superscalar processor (N is large)
  - A branch every 5 instructions that takes 3 cycles to resolve
  - What is the effective fetch bandwidth?

Disruption of Sequential Control Flow



Riseman and Foster's Study

- 7 benchmark programs on CDC-3600
- Assume infinite machine:
  - Infinite memory and instruction stack, register file, fxn units

    Consider only true dependency at data-flow limit
- If bounded to single basic block, i.e. no bypassing of branches $\Rightarrow$ maximum speedup is **1.72**
- Suppose one can bypass conditional branches and jumps (i.e. assume the actual branch path is always known such that branches do not impede instruction execution)

Br. Bypassed:	*0*	*1*	*2*	*8*	*32*	*128*
Max Speedup:	***1.72***	***2.72***	***3.62***	***7.21***	***24.4***	***51.2***

Dynamic Branch Prediction Tasks

➢ Target Address Generation

- Access register
  - PC, GP register, Link register
- Perform calculation
  - +/- offset, auto incrementing/decrementing

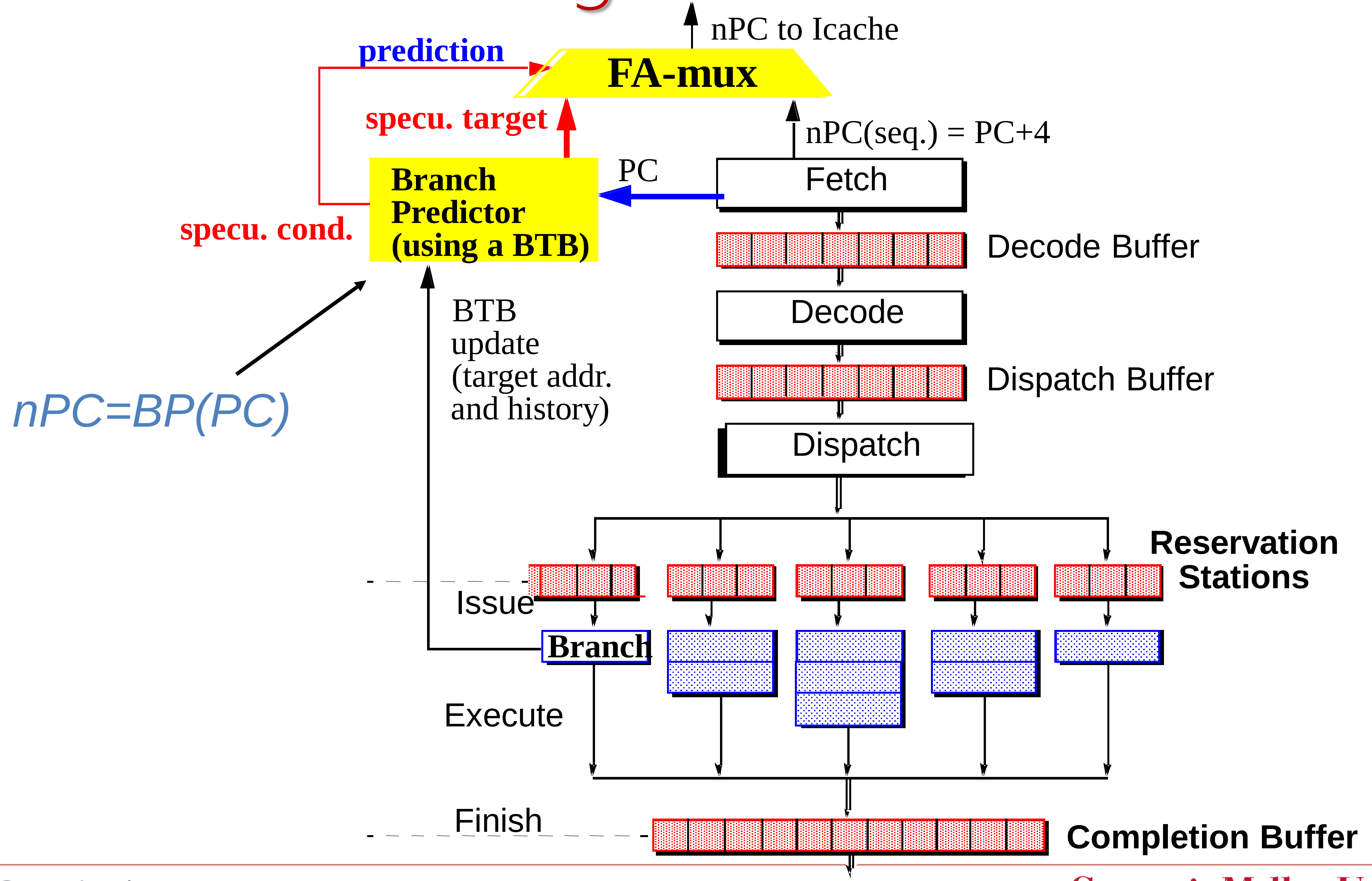
⇒ Target Speculation

➢ Condition Resolution

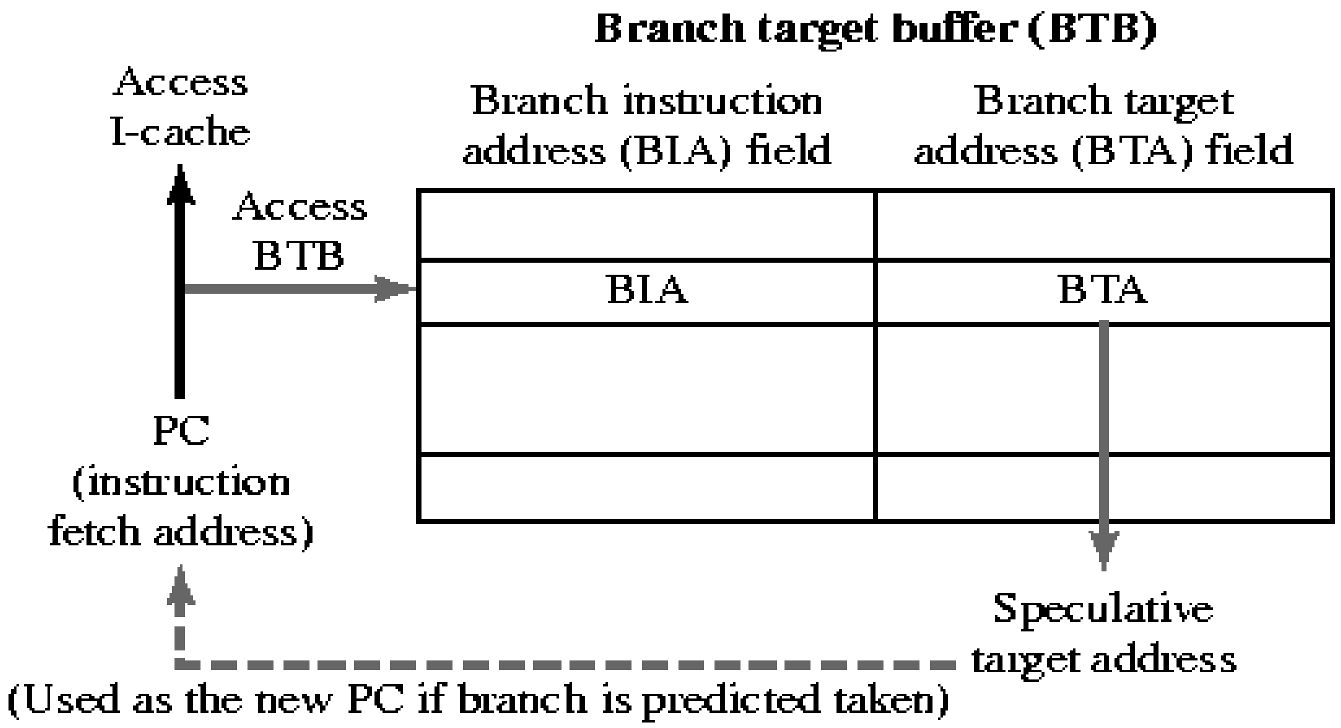
- Access register
  - Condition code register, data register, count register
- Perform calculation
  - Comparison of data register(s)

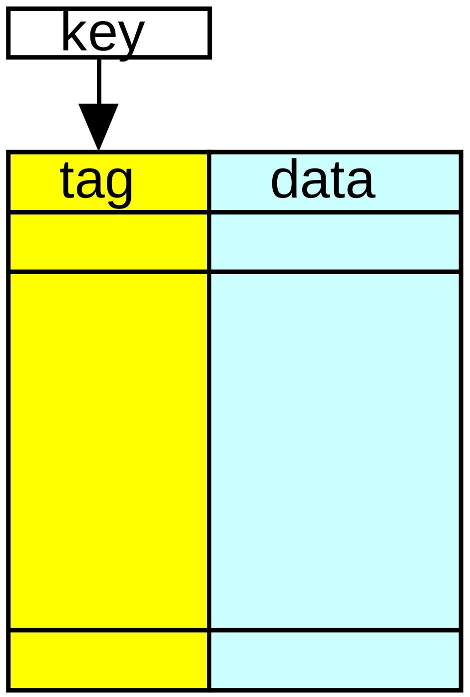
⇒ Condition Speculation

Dynamic Branch Target Prediction



Target Prediction: Branch Target Buffer (BTB)



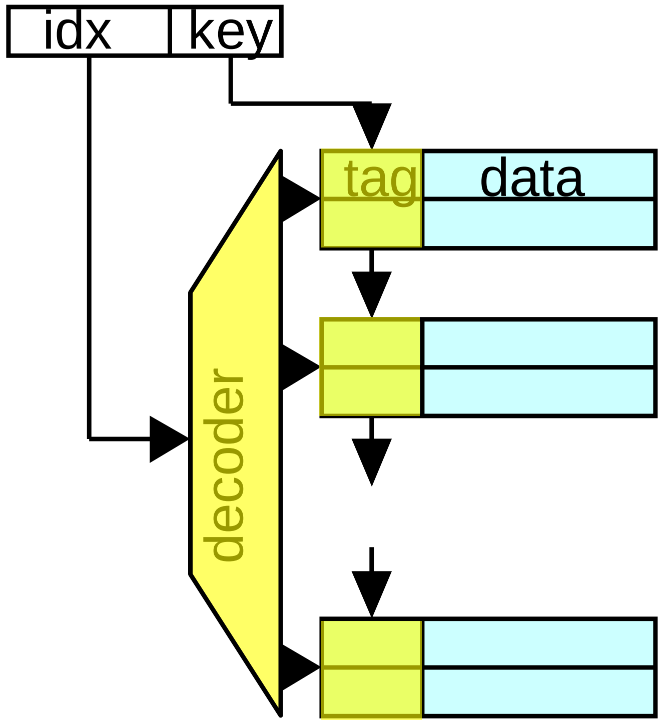


Associative Memory (CAM)

- A small "cache-like" memory in the instruction fetch stage
- Remembers previously executed branches, their addresses (PC), information to aid target prediction, and most recent target addresses
- I-fetch stage compares current PC against those in BTB to "guess" nPC
  - If matched then prediction is made else nPC=PC+N
  - If predict taken then nPC=target address in BTB else nPC=PC+N
- When branch is actually resolved, BTB is updated

More on BTB (aka BTAC)

- Typically a large associative structure
  - Pentium3: 512 entries, 4-way; Opteron: 2K entries, 4-way
- Entry format
  - Valid bit, address tag (PC), target address, fall-through BB address (length of BB), branch type info, branch direction prediction
- BTB provides both target and direction prediction
- Multi-cycle BTB access?
  - The case in many modern processors (2 cycle BTB)
  - Start BTB access along with I-cache in cycle 0
  - In cycle 1, fetch from PC+N (predict not-taken)
  - In cycle 2, use BTB output to verify
    - 1 cycle fetch bubble if branch was taken

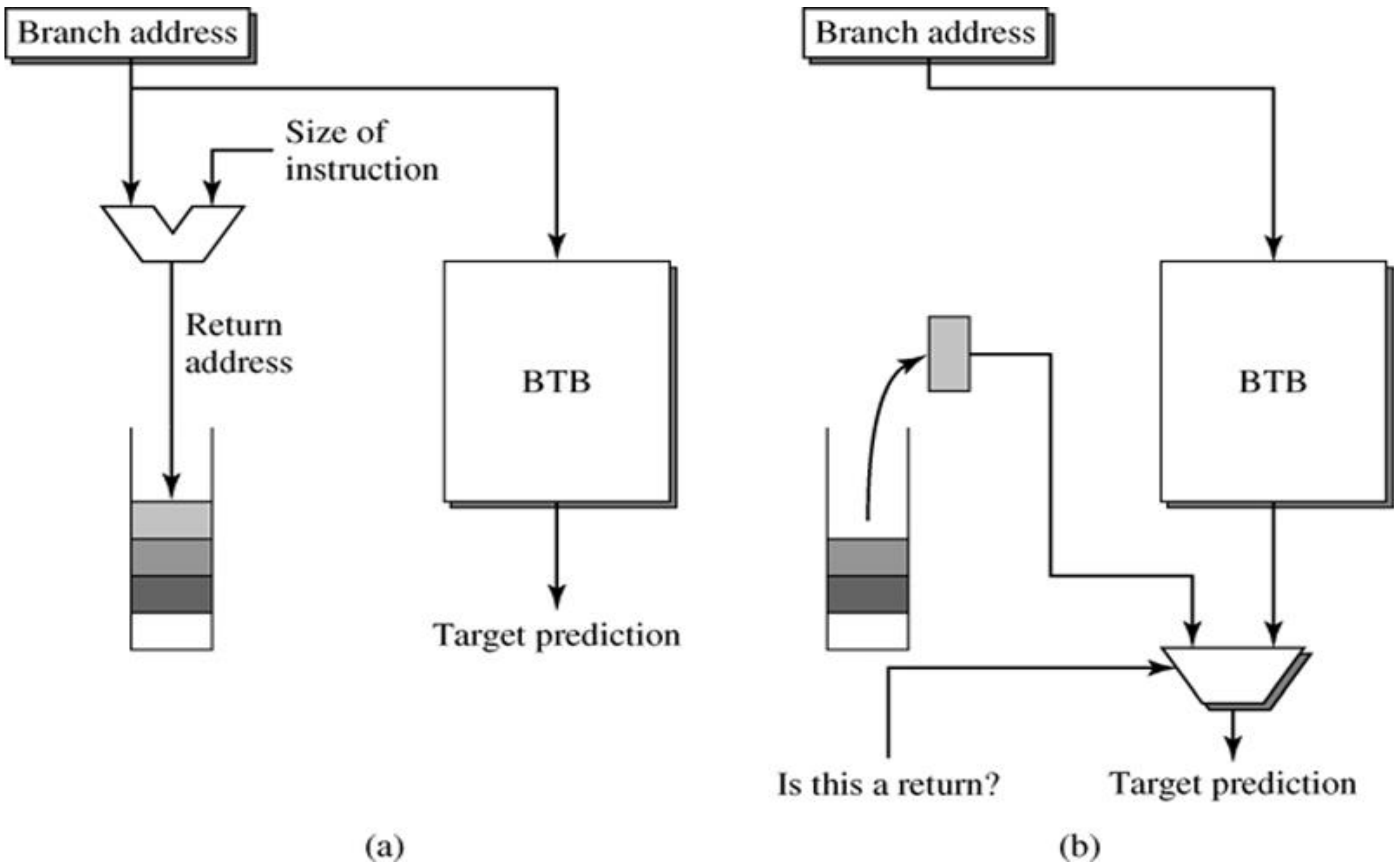


N-Way
Set-Associative Memory
k-bit index
$2^k \cdot N$ blocks

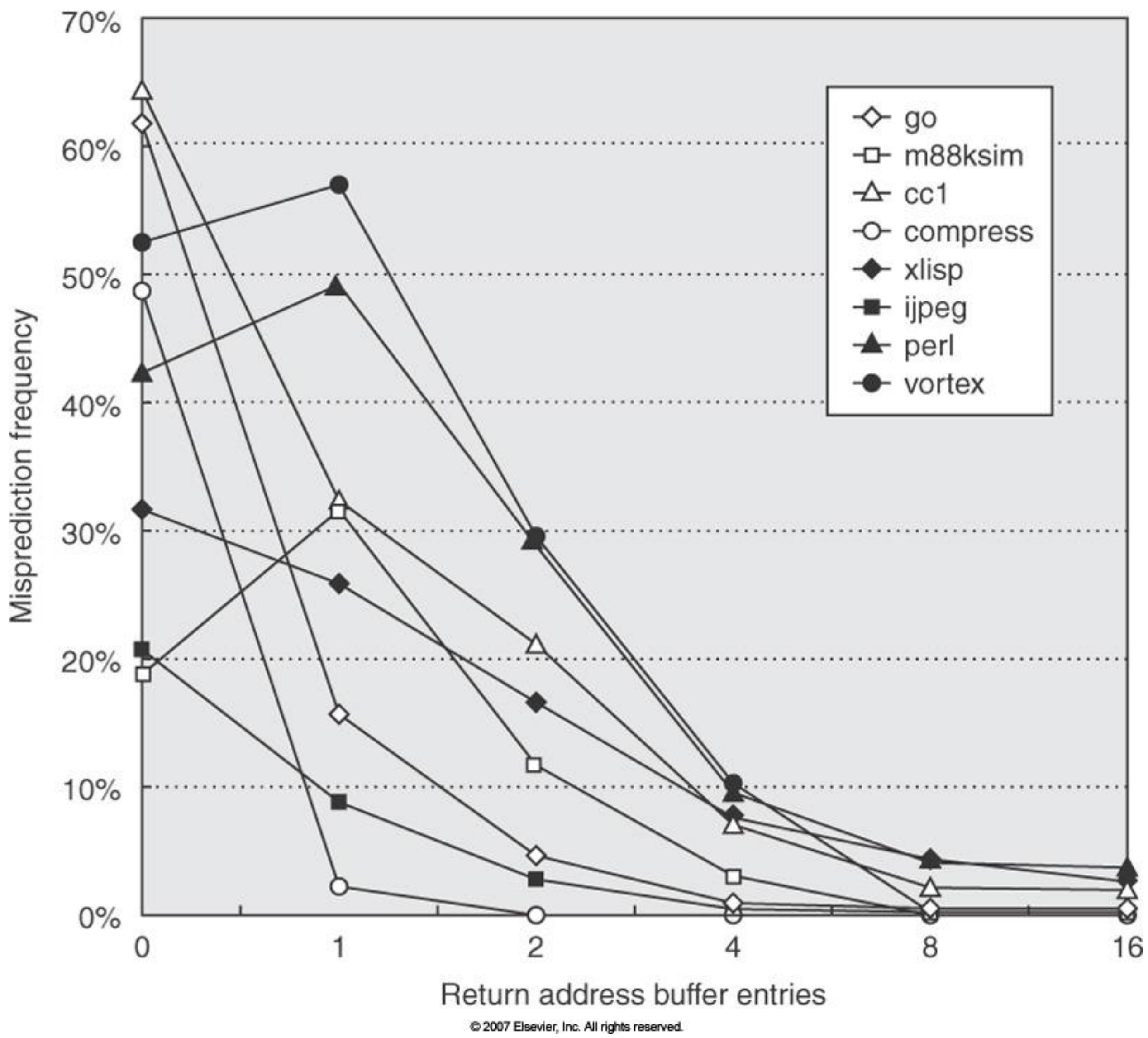
Branch Target Prediction for Function Returns

- In most languages, function calls are fully nested
  - If you call A() ⇒ B() ⇒ C() ⇒ D()
  - Your return targets are PCc ⇒ PCb ⇒ PCa ⇒ PCmain
- Return address stack (RAS)
  - A FILO structure for capturing function return addresses
  - Operation
    - On a function call retirement, push call PC into the stack
    - On a function return, use the top value in the stack & pop
  - A 16-entry RAS can predict returns almost perfectly
    - Most programs do not have such a deep call tree
  - Sources of RAS inaccuracies
    - Deep call statements (circular buffer overflow – will lose older calls)
    - Setjmp and longjmp C functions (irregular call semantics)

Return Address Stack (RAS) Operation



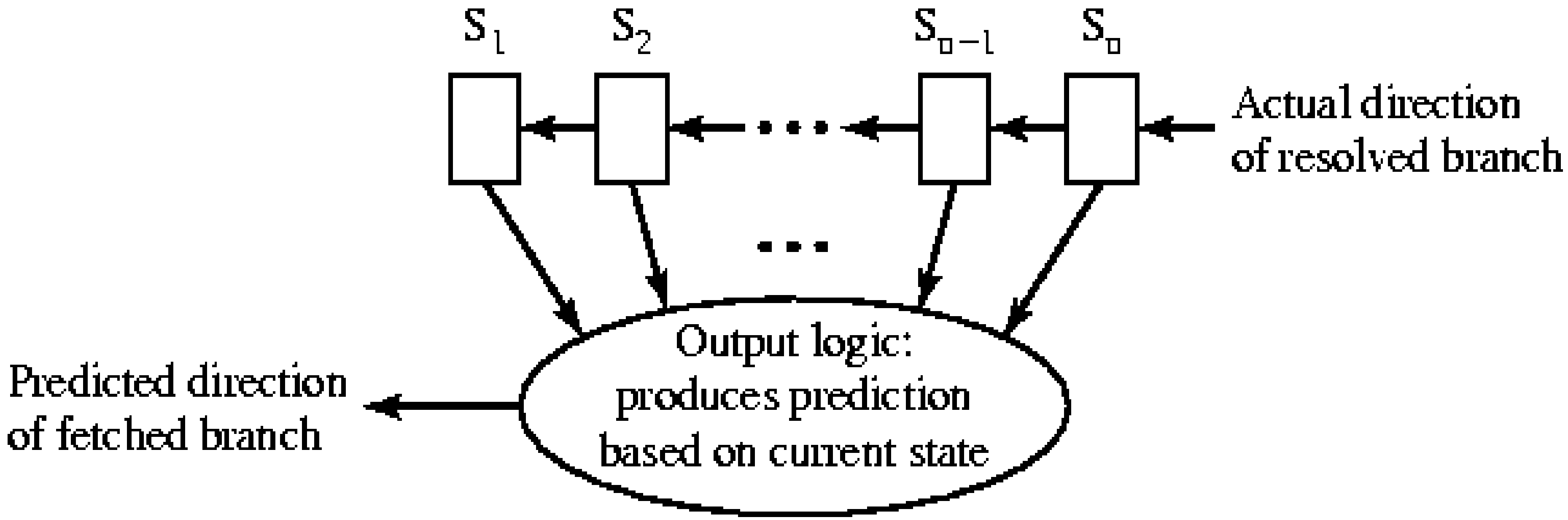
RAS Effectiveness & Size (SPEC CPU'95)



Branch Condition Prediction

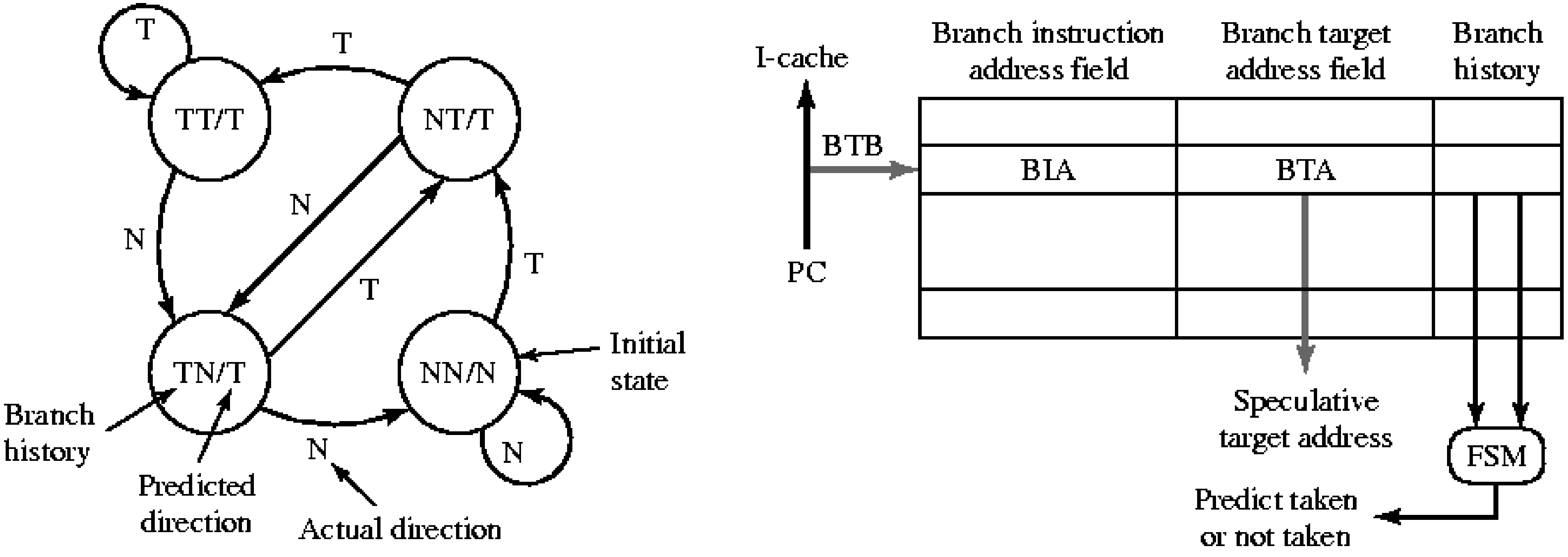
- Biased For Not Taken
  - Does not affect the instruction set architecture
  - Not effective in loops
- Software Prediction
  - Encode an extra bit in the branch instruction
    - Predict not taken: set bit to 0
    - Predict taken: set bit to 1
  - Bit set by compiler or user; can use profiling
  - Static prediction, same behavior every time
- Prediction Based on Branch Offsets
  - Positive offset: predict not taken
  - Negative offset: predict taken
- Prediction Based on History

History-Based Branch Direction Prediction



- Track history of previous directions of branches (T or NT)
- History can be local (per static branch) or global (all branches)
- Based on observed history bits (T or NT), a FSM makes a prediction of Taken or Not Taken
- Assumes that future branching behavior is predictable based on historical branching behavior

History-Based Branch Prediction



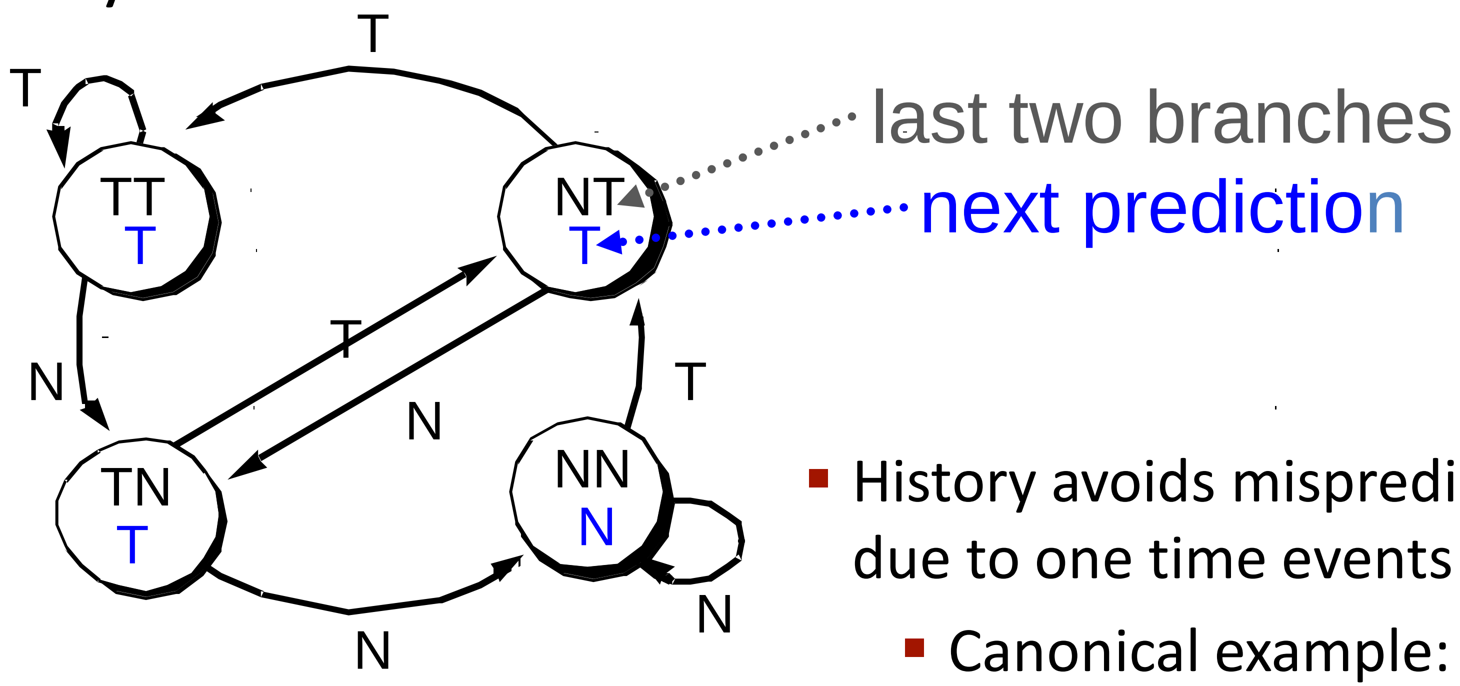
(a)

(b)

[James E Smith, CDC, 1981]

Example Prediction Algorithm

- Prediction accuracy approaches maximum with as few as 2 preceding branch occurrences used as history

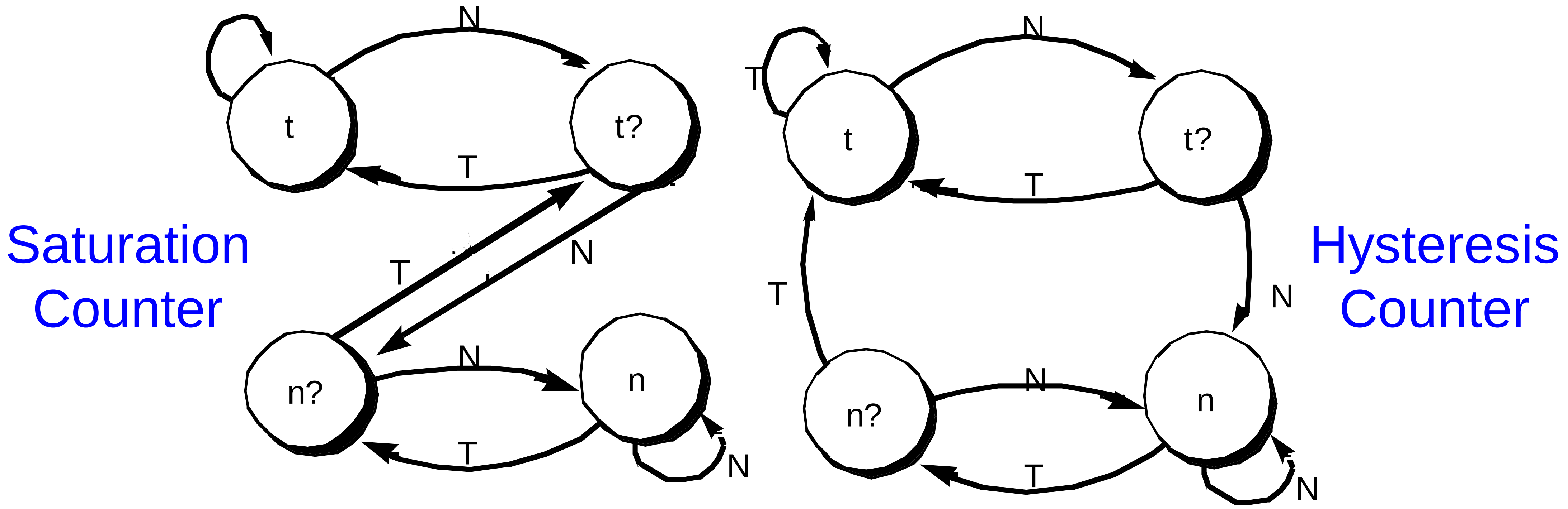


Results (%)

IBM1	IBM2	IBM3	IBM4	DEC	CDC
93.3	96.5	90.8	83.4	97.5	90.6

- History avoids mispredictions due to one time events
  - Canonical example: loop exit
- 2-bit FSM as good as n-bit FSM
- Saturating counter as good as any FSM

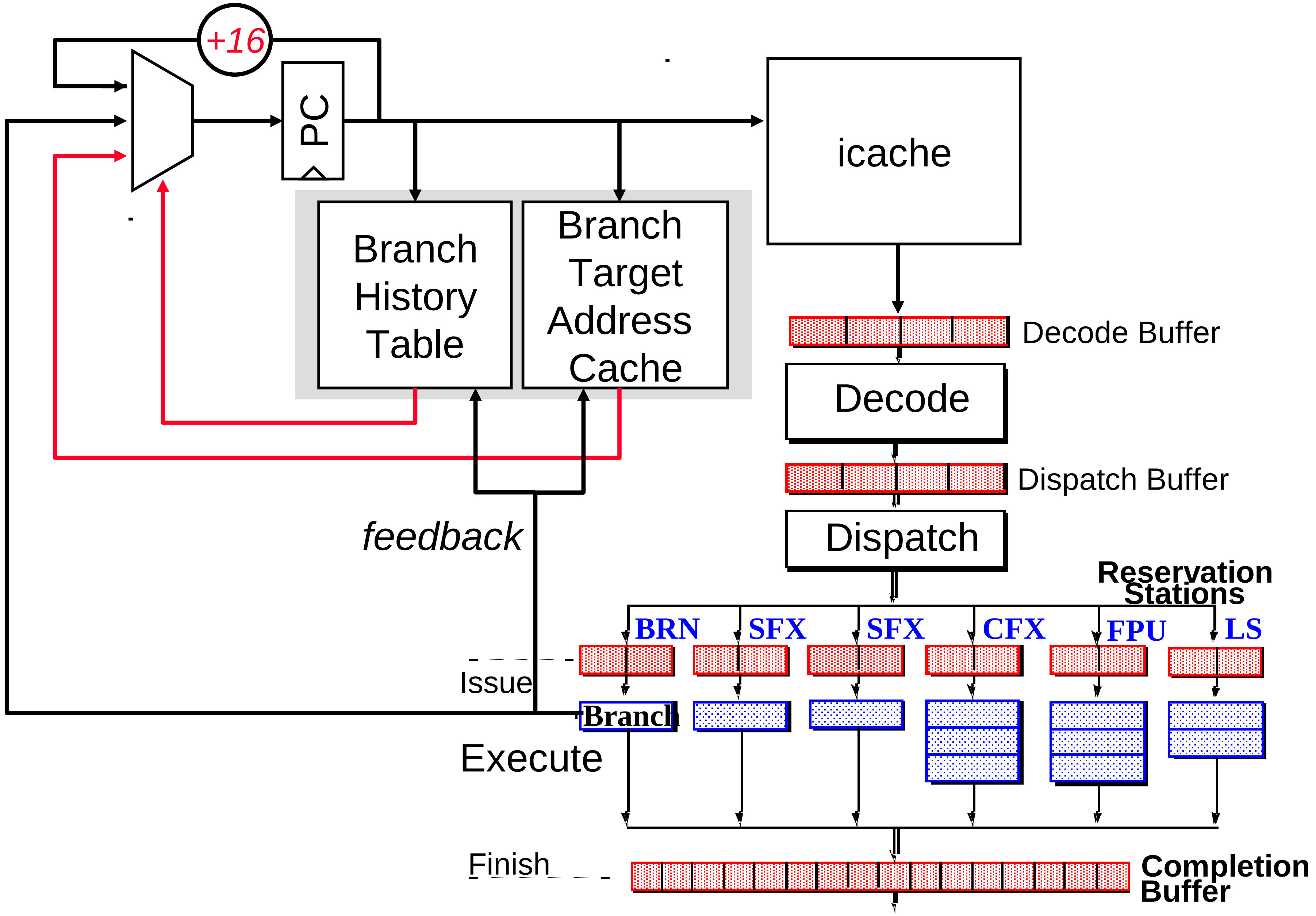
Other Prediction Algorithms



- Combining prediction accuracy with BTB hit rate (86.5% for 128 sets of 4 entries each), branch prediction can provide the net prediction accuracy of approximately 80%. This implies a 5-20% performance enhancement.

Dynamic Branch Prediction Based on History

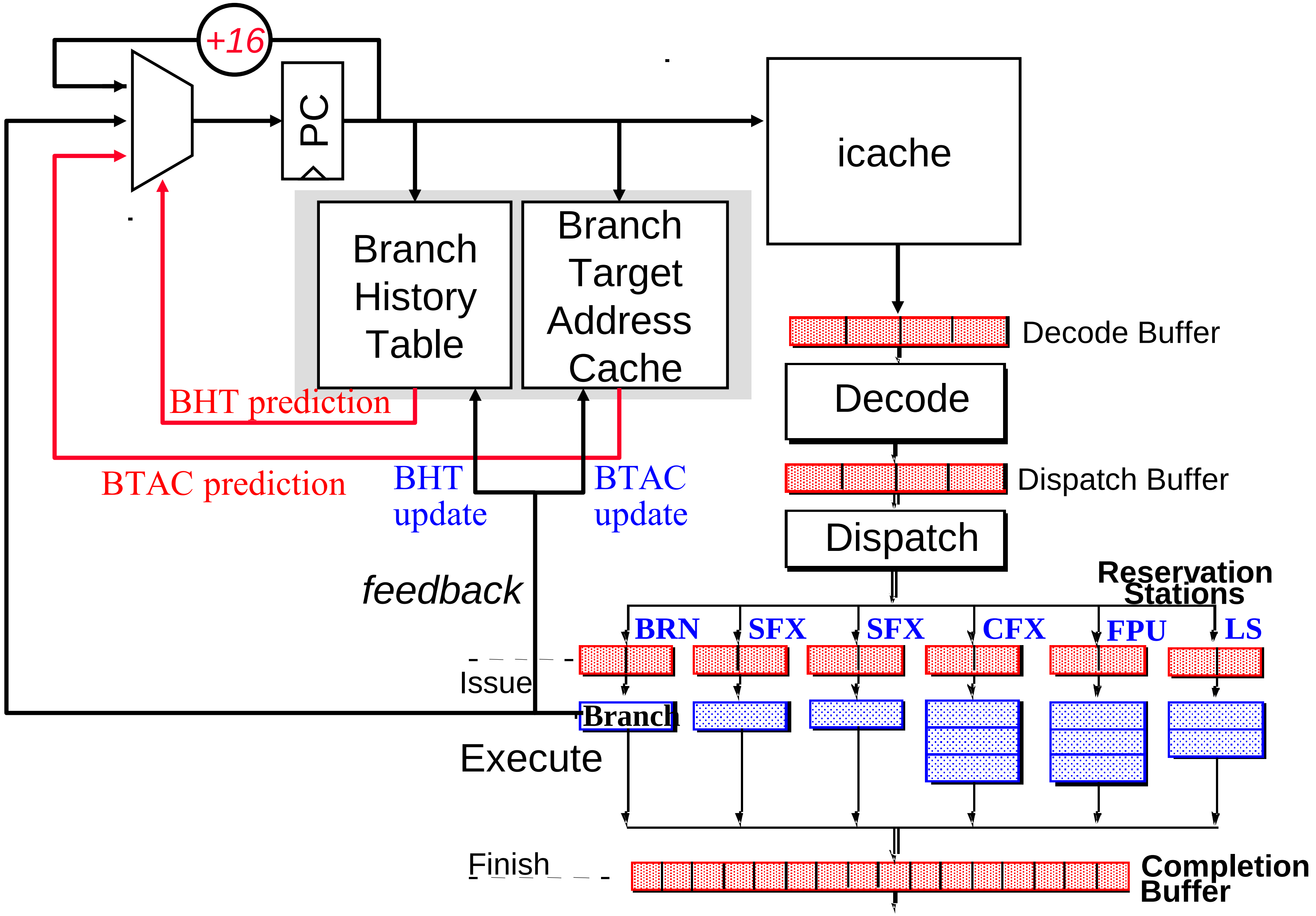
- Use HW tables to track history of direction/targets
  - nextPC = function(PC, history)
- Need to verify prediction
  - Branch still gets to execute



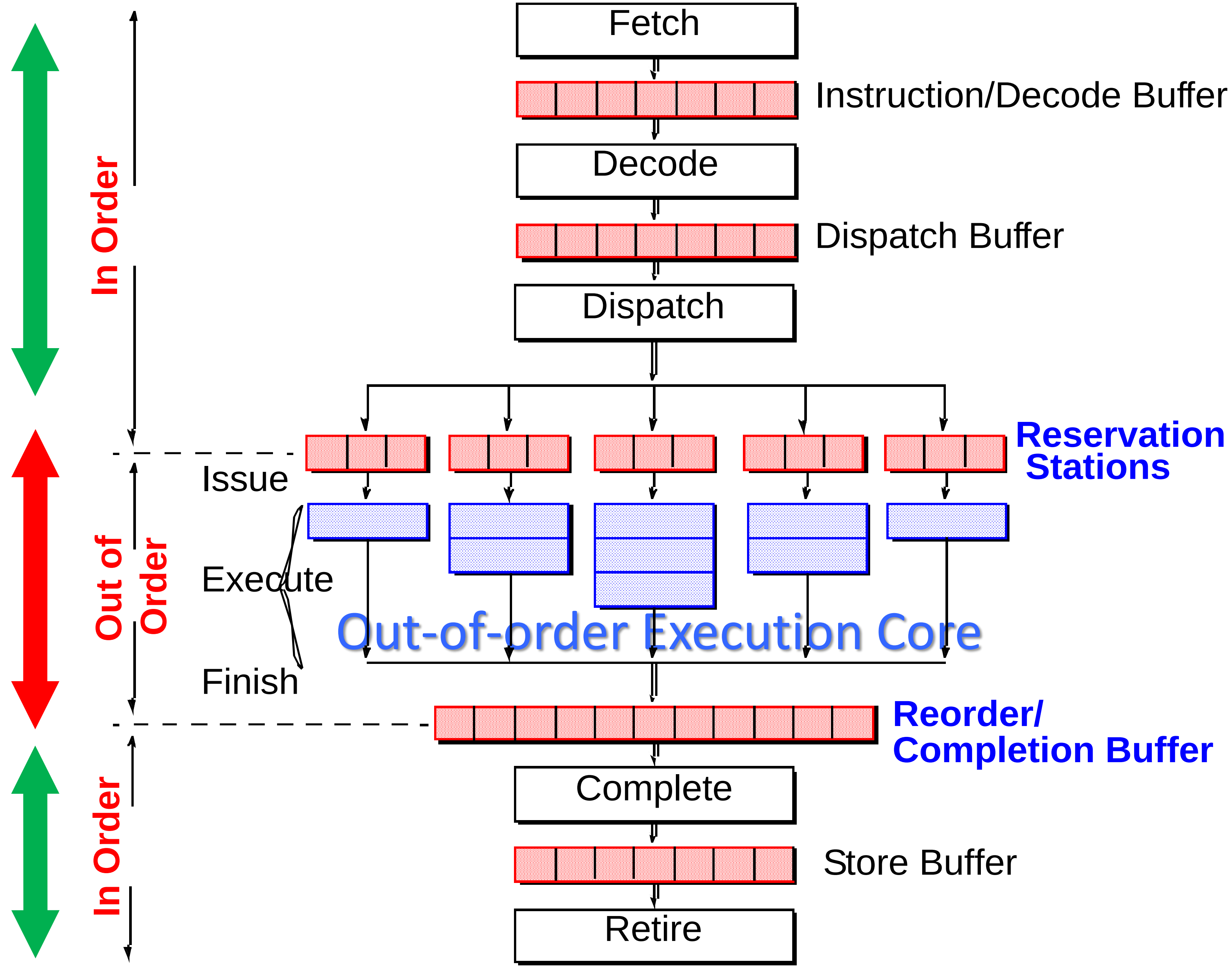
PowerPC 604 Branch Predictor: BHT & BTAC

BTAC:
- 64 entries
- Fully associative
- Hit → predict taken

BHT:
- 512 entries
- Direct mapped
- 2-bit saturating counter
- History based prediction
- Overrides BTAC prediction



Modern Superscalar Processor Organization



We have: fetched & decoded instructions

- In-order but speculative (branch prediction)

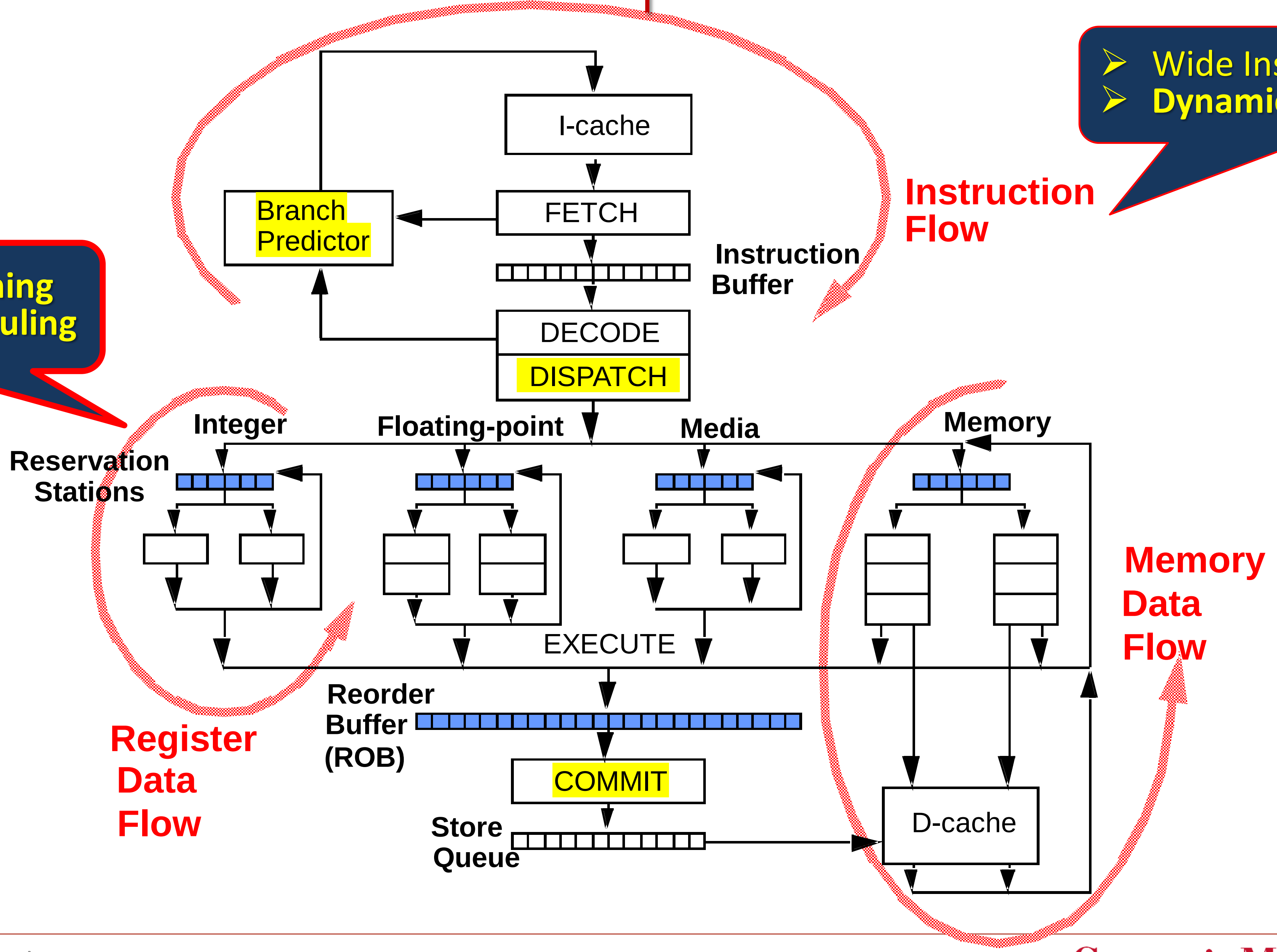
Register Renaming

- Eliminate WAR and WAW dependencies without stalling

Dynamic Scheduling

- Track & resolve true RAW dependencies
- Scheduling HW: Instruction window, reservation stations, common data bus, …

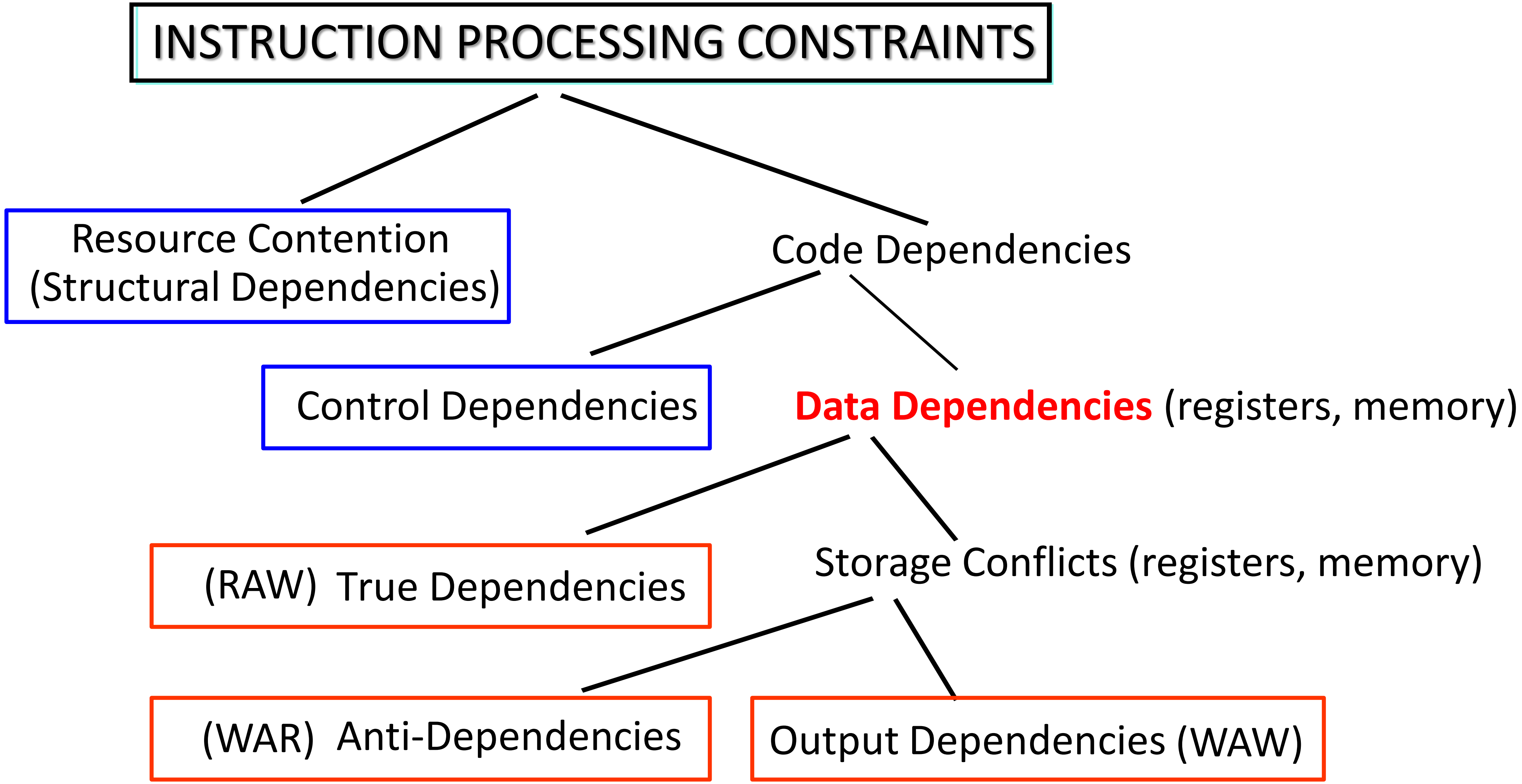
Three Flow Paths of Superscalar Processors



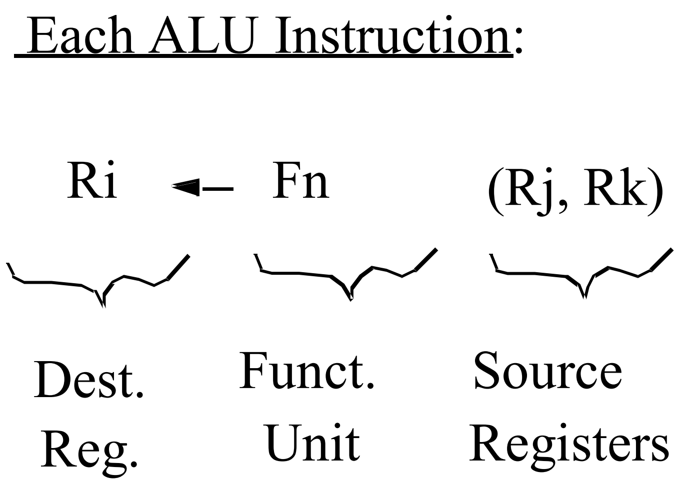
- Wide Instruction Fetching
- **Dynamic Branch Prediction**

- **Register Renaming**
- **Dynamic Scheduling**

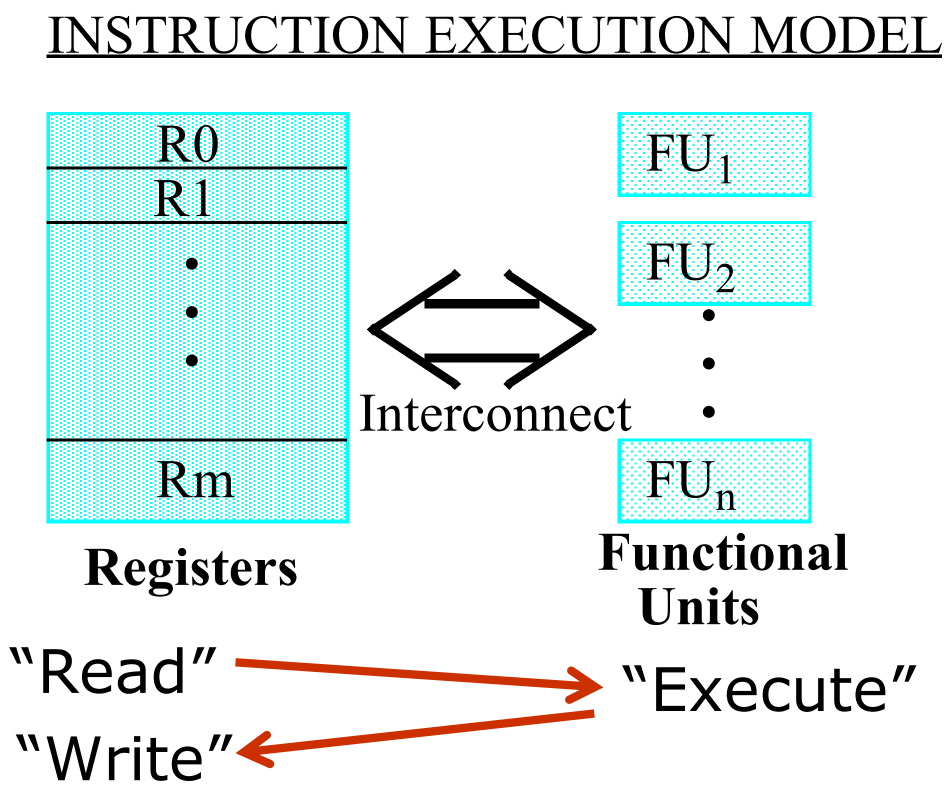
The Big Picture: Impediments Limiting ILP



Register Data Flow



"Register Transfer"



- For an instruction to execute:
  - Need availability of functional unit Fn (structural dependency)
  - Need availability of Rj and Rk (RAW: true data dependency)
  - Need availability of Ri (WAR and WAW: anti and output dependencies)

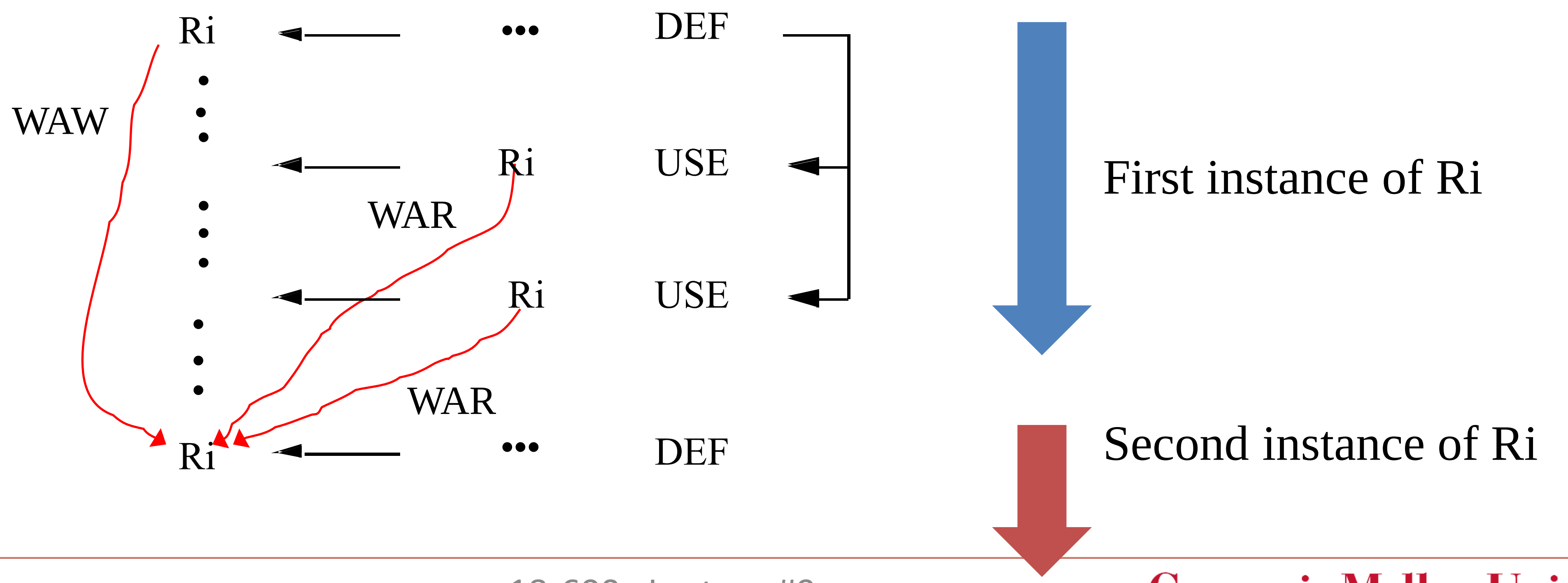
Causes of Register Storage Conflict

REGISTER RECYCLING

- MAXIMIZE USE OF REGISTERS
- MULTIPLE ASSIGNMENTS OF VALUES TO REGISTERS

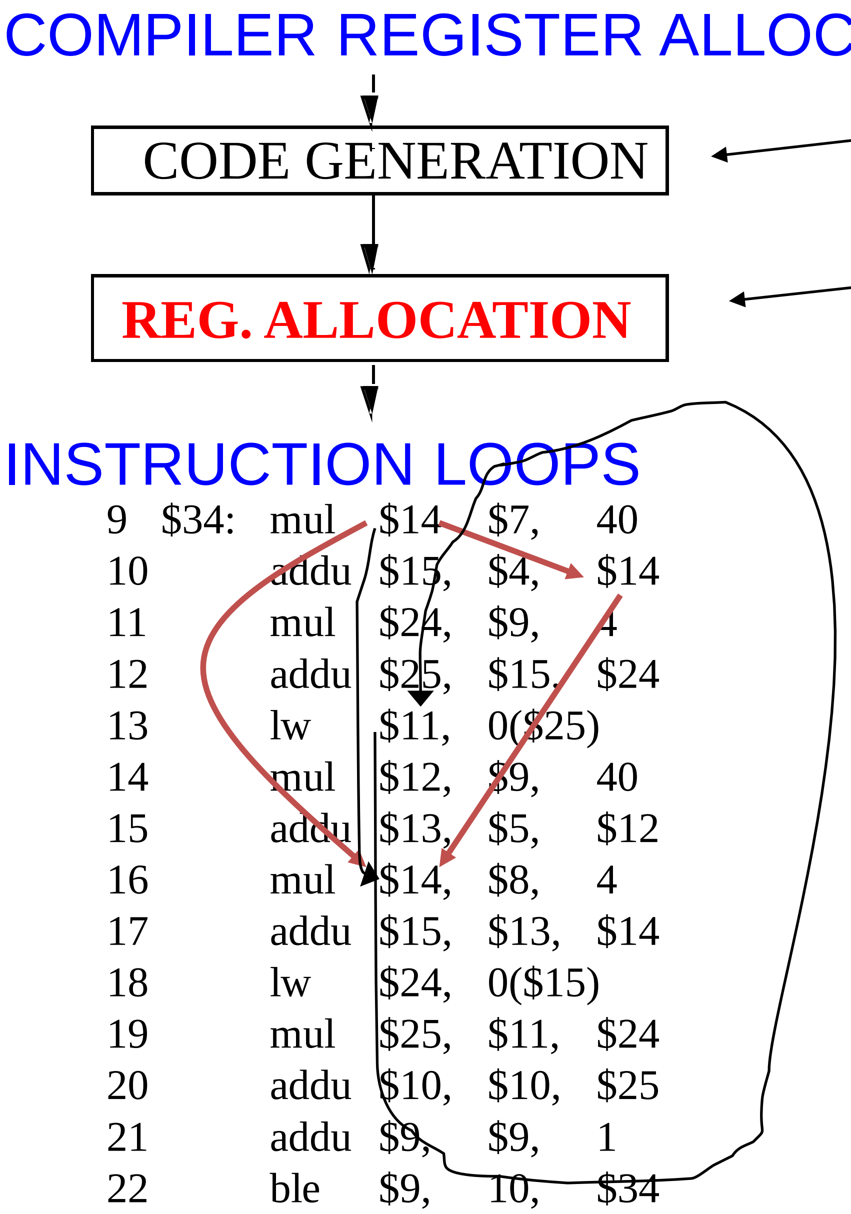
OUT OF ORDER ISSUING AND COMPLETION

- LOSE IMPLIED PRECEDENCE OF SEQUENTIAL CODE
- LOSE 1-1 CORRESPONDENCE BETWEEN VALUES AND REGISTERS



Reason for WAW and WAR: Register Recycling

- Intermediate code
  - Infinite number of symbolic registers
  - One used per value definition
- Register Allocation via graph coloring
  - Map symbolic registers to few architectural registers
  - Leads to register reuses



Single Assignment, Symbolic Reg.

Map Symbolic Reg. to Physical Reg.
Maximize Reuse of Reg.

"Spill code"
(if not enough registers)

For (k=1;k<= 10; k++)
t += a [i] [k] * b [k] [j] ;

- Dynamic register reuse
  - Reuse same set of registers in each iteration
  - Overlapped execution of multiple iterations

Register Renaming: The Idea

- Anti and output dependencies are false dependencies

$$r_3 \leftarrow r_1 \text{ op } r_2$$
$$r_5 \leftarrow r_3 \text{ op } r_4$$
$$r_3 \leftarrow r_6 \text{ op } r_7$$

- The dependency is on name/location rather than data
- Given unlimited number of registers, anti and output dependencies can always be eliminated

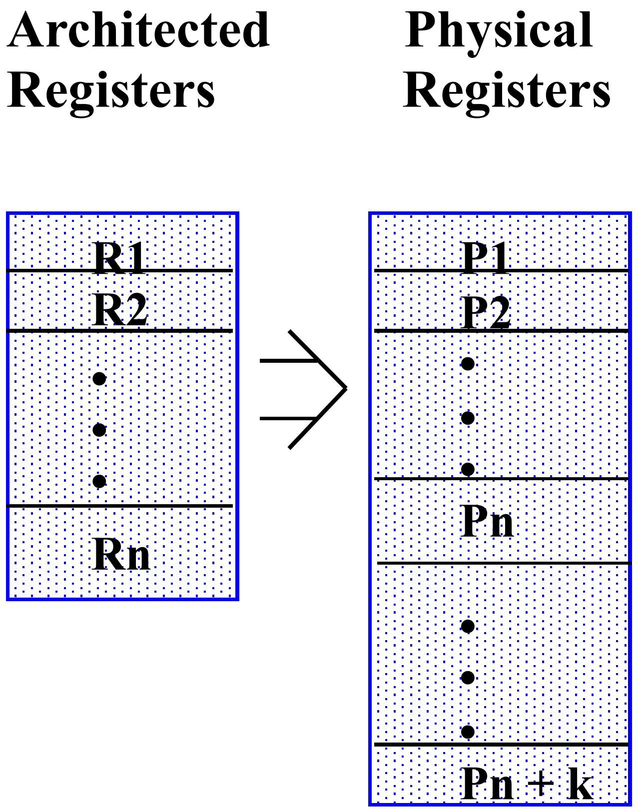
Original	Renamed
r1 ← r2 / r3	r1 ← r2 / r3
r4 ← r1 * r5	r4 ← r1 * r5
r1 ← r3 + r6	r8 ← r3 + r6
r3 ← r1 - r4	r9 ← r8 - r4

Register Renaming

Register Renaming Resolves:

Anti- Dependences

Output Dependences



Design of Redundant Registers

Number:

- **One**
- **Multiple**

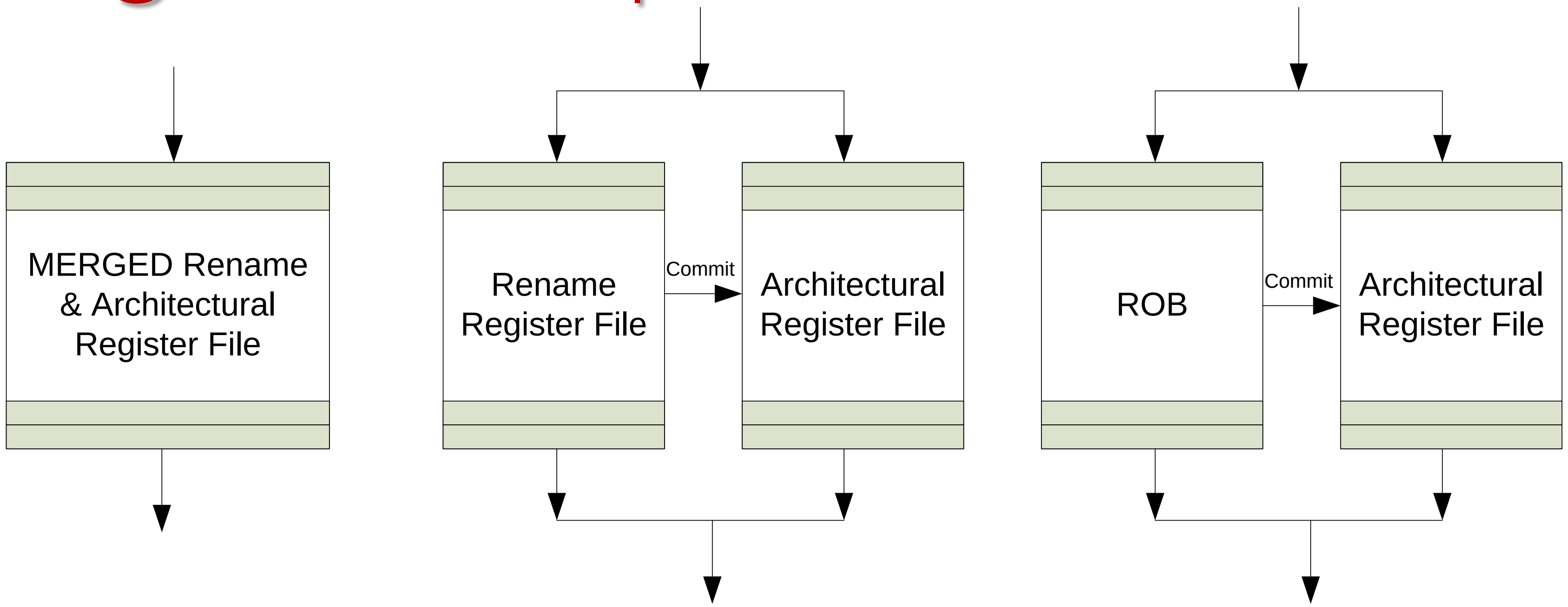
Allocation:

- **Fixed for Each Register**
- **Pooled for all Regsiters**

Location:

- **Attached to Register File (Centralized)**
- **Attached to functional units (Distributed)**

Renaming Buffer Options



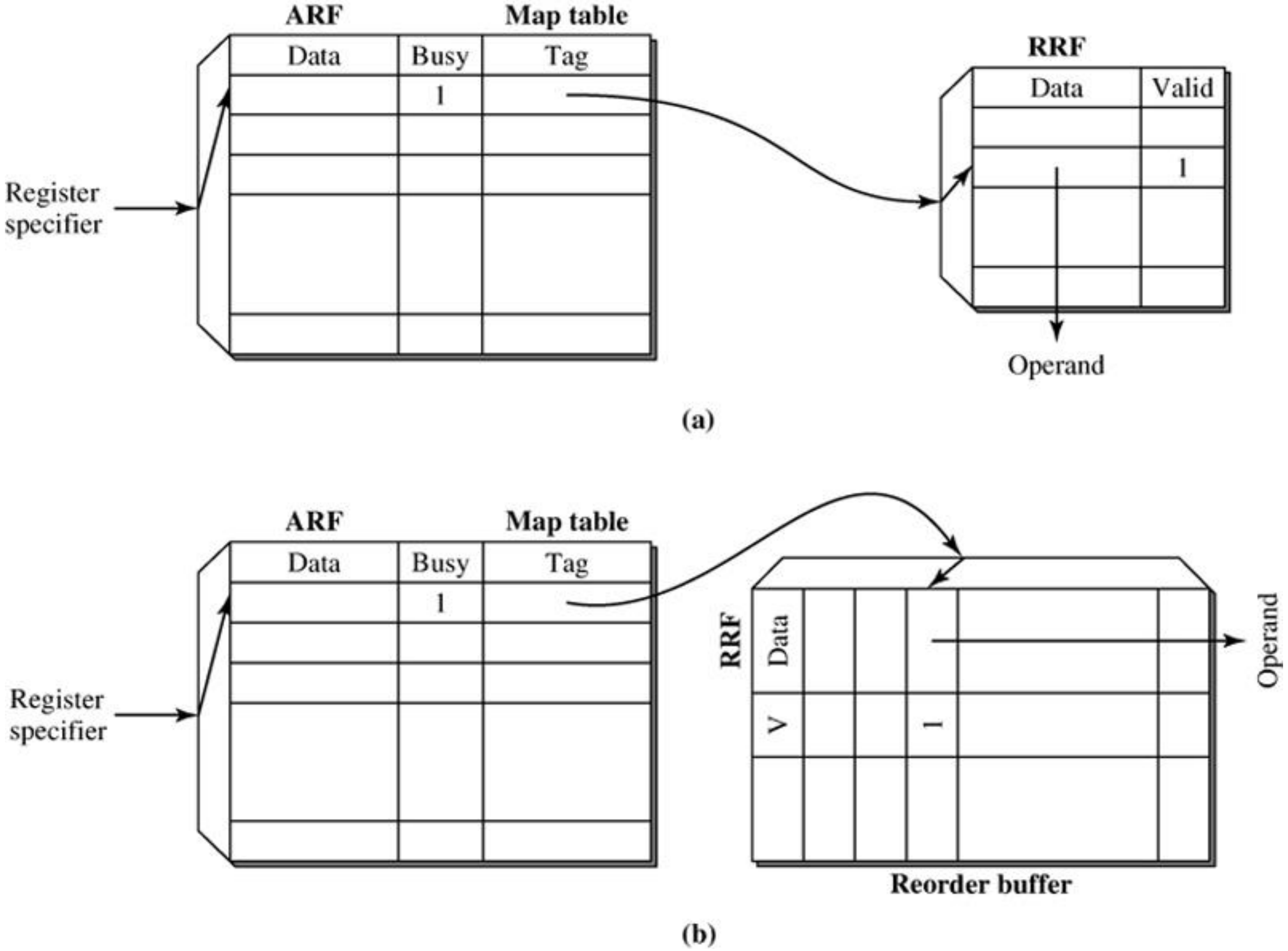
- Unified/merged register file – MIPS R10K, Alpha 21264
  - Registers change role architecture to renamed
- Rename register file (RRF) – PA 8500, PPC 620
  - Holds new values until they are committed to ARF (extra transfer)
- Renaming in the ROB – Pentium III
- Note: can have a single scheme or separate for integer/FP

Number of Rename Registers

- Naïve: as many as the number of pending instructions
  - Waiting to be scheduled + executing + waiting to commit
- Simplification
  - Do not need renaming for stores, branches, …
- Usual approach:
  - # scheduler entries ≤ # RRF entries ≤ # ROB entries

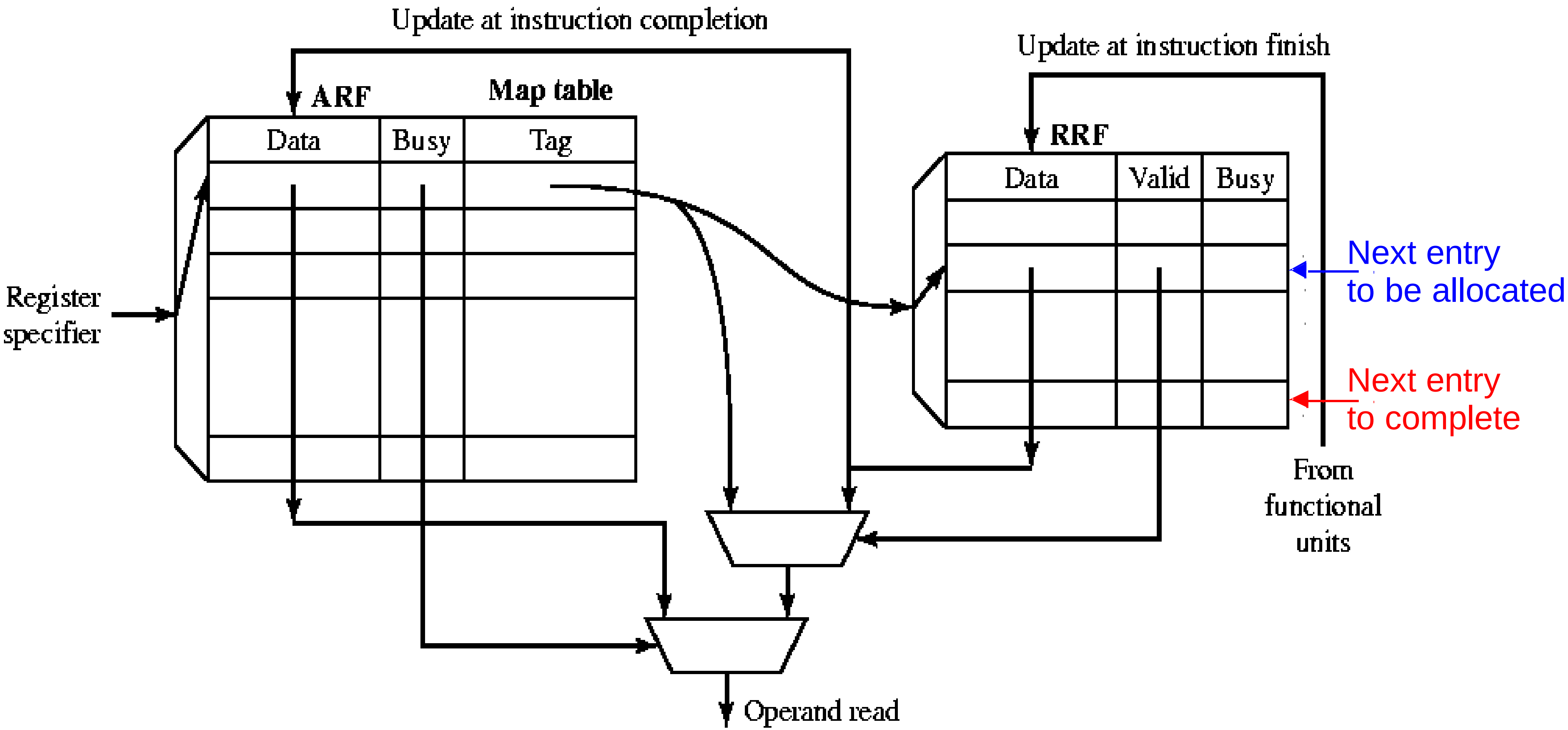
- Examples:
  - PPC 620: scheduler 15, RRF 16 (RRF), ROB 16
  - MIPS R12000: scheduler 48, RRF 64 (merged), ROB 48
  - Pentium III: scheduler 20, RRF 40 (in ROB), ROB 40

Integrating Map Table with the ARF

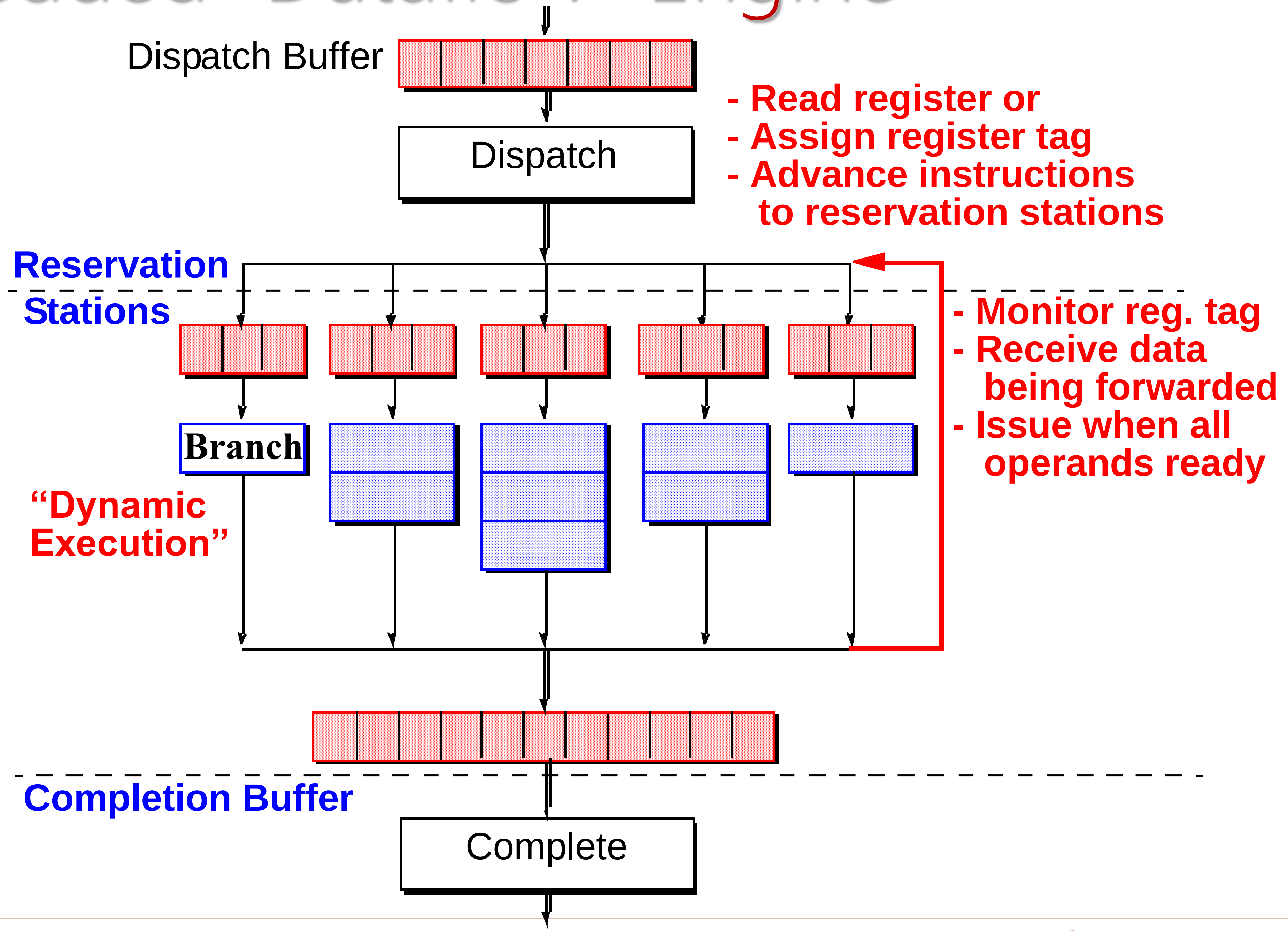


Register Renaming Tasks

- Source Read, Destination Allocate, Register Update



Embedded "Dataflow" Engine



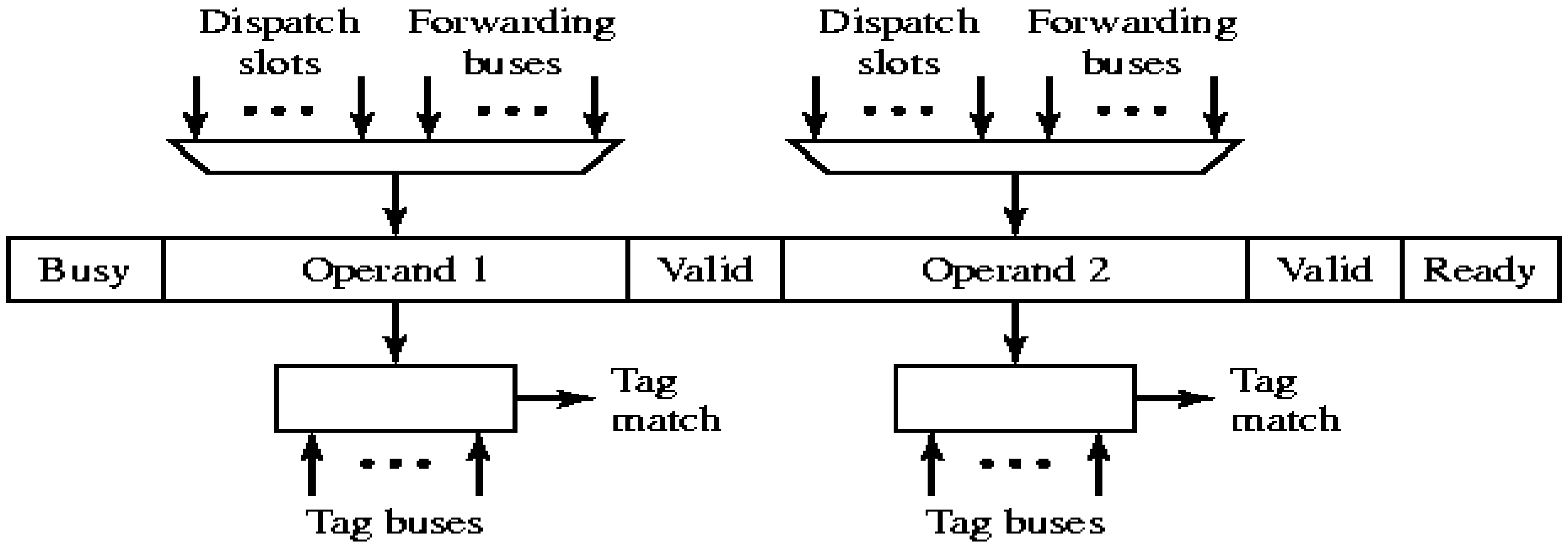
Steps in Dynamic OOO Execution (1)

- FETCH instruction (in-order, speculative)
  - I-cache access, predictions, insert in a fetch buffer
- DISPATCH (in-order, speculative)
  - Read operands from Register File (ARF) and/or Rename Register File (RRF)
    - RRF may return a ready value or a Tag for a physical location
  - Allocate new RRF entry (rename destination register) for destination
  - Allocate Reorder Buffer (ROB) entry
  - Advance instruction to appropriate entry in the scheduling hardware
    - Typical name for centralized: Issue Queue or Instruction Window
    - Typical name for distributed: Reservation Stations

Steps in Dynamic OOO Execution (2)

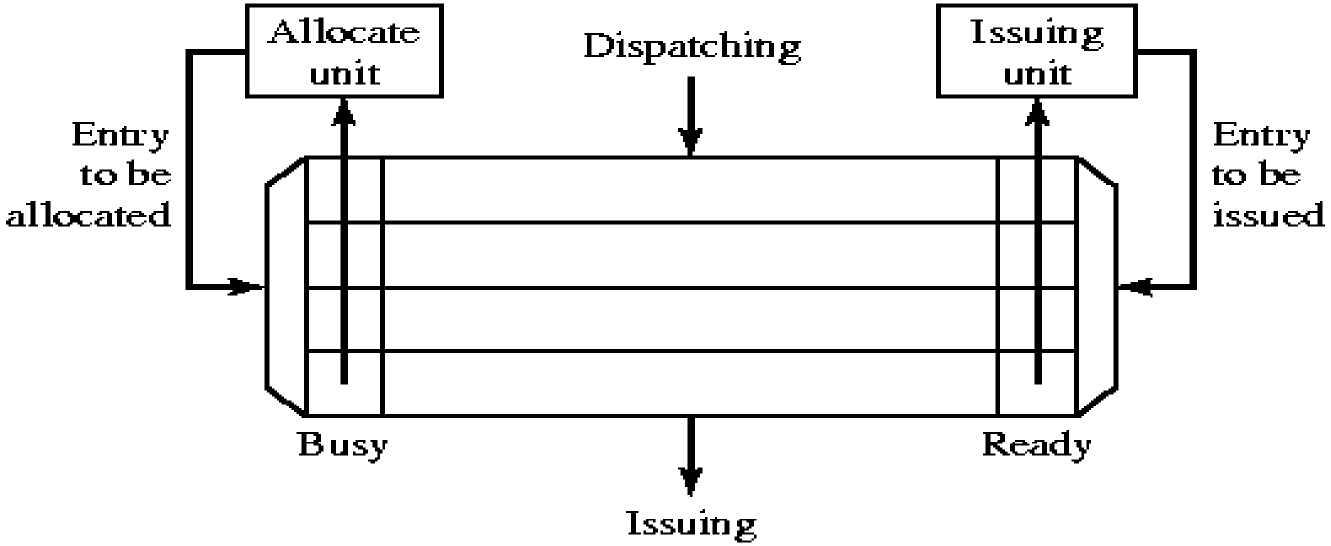
- ISSUE & EXECUTE (out-of-order, speculative)
  - Scheduler entry monitors result bus for rename register Tag(s) for pending operand(s)
    - Find out if source operand becomes ready; if Tag(s) match, latch in operand(s)
  - When all operands ready, instruction is ready to be issued into FU (wake-up)
  - Issue instruction into FU, deallocate scheduler entry, no further stalling in FU pipe
    - Issuing is subject to structural hazards and scheduling priorities (select)
  - When execution finishes, broadcast result to waiting scheduler entries and RRF entry
- COMMIT/RETIRE/GRADUATE (in-order, non-speculative)
  - When ready to commit result into "in-order" (architectural) state (head of the ROB):
    - Update architectural register from RRF entry, deallocate RRF entry, and if it is a store instruction, advance it to Store Buffer
    - Deallocate ROB entry and instruction is considered architecturally completed
    - Update predictors based on instruction result

Reservation Station Implementation



(a)

+ info for executing instruction (opcode, ROB entry, RRF entry…)



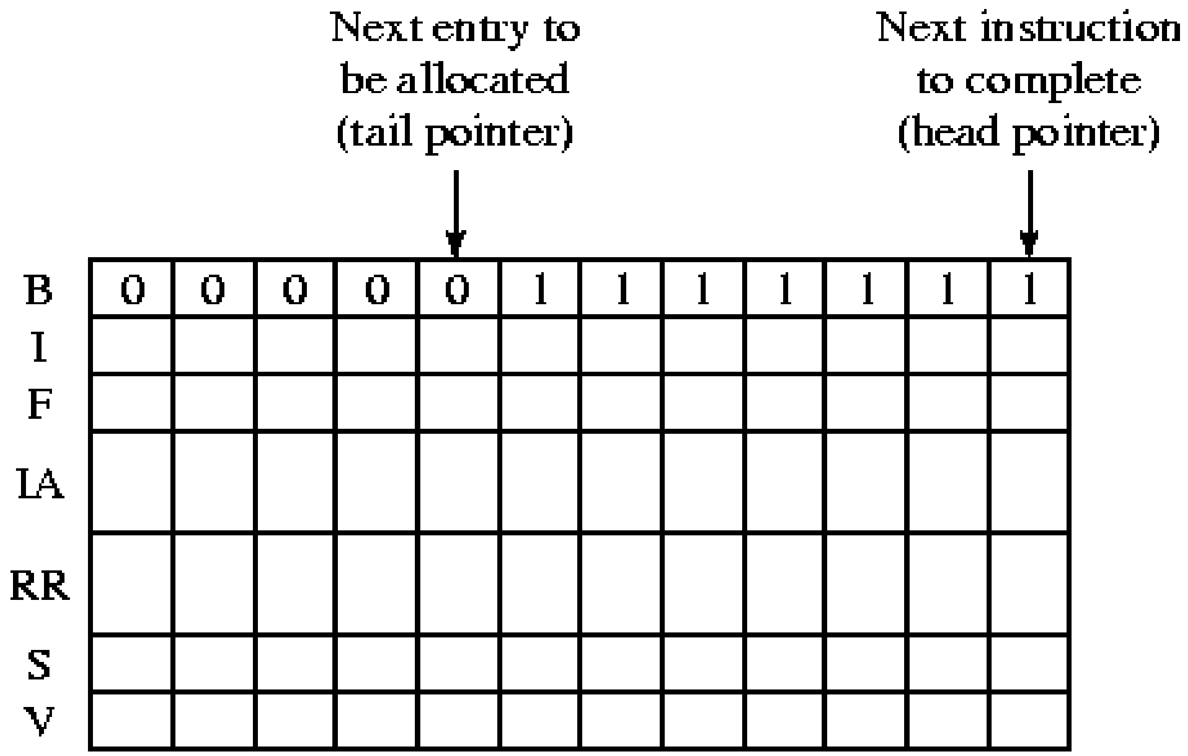
(b)

- Reservation Stations: distributed vs. centralized
  - Wakeup: benefit to partition across data types
  - Select: much easier with partitioned scheme
    - Select 1 of n/4 vs. 4 of n

Reorder Buffer Implementation

Busy	Issued	Finished	Instruction address	Rename register	Speculative	Valid

(a)

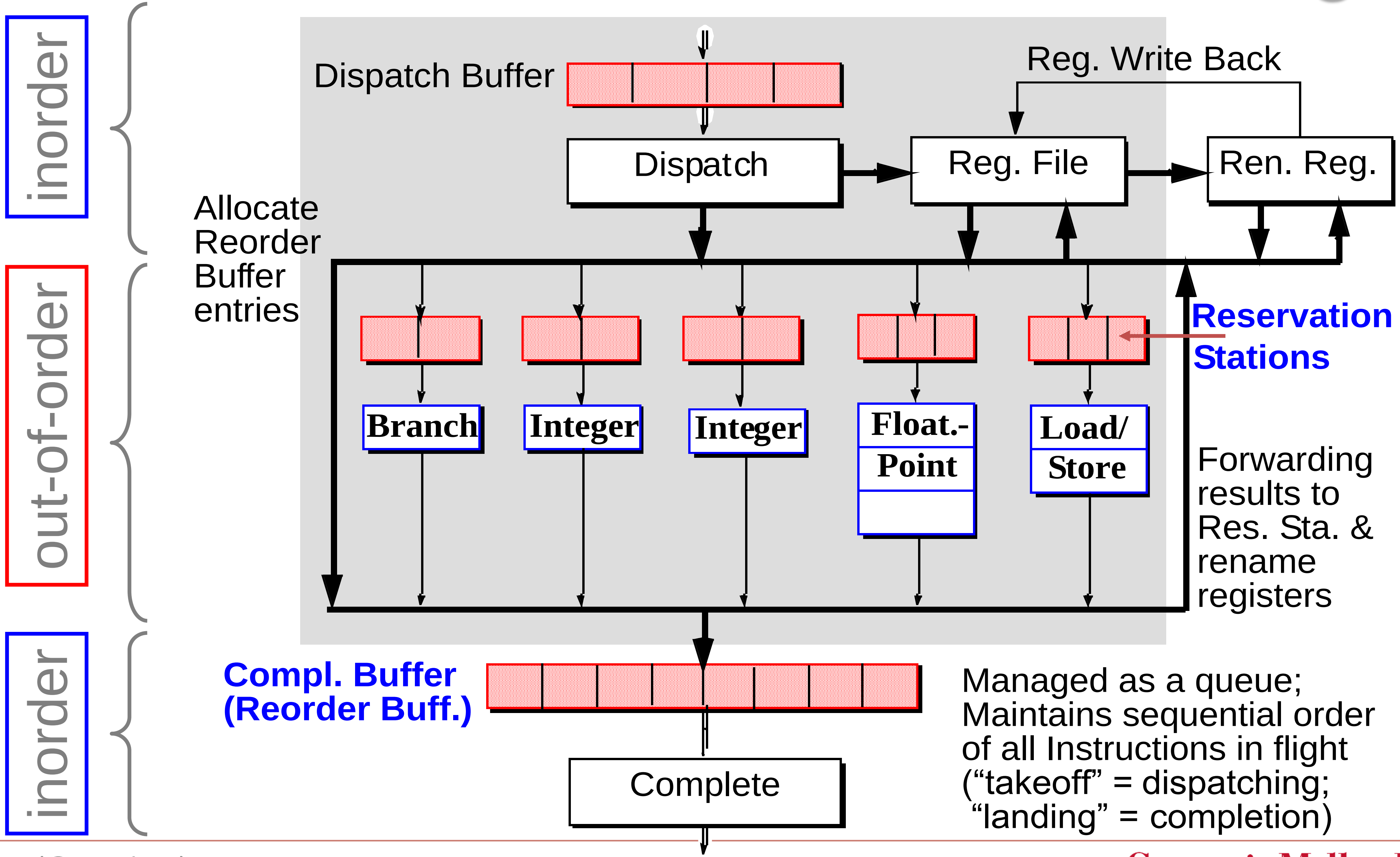


Reorder buffer

(b)

- Reorder Buffer
  - “Bookkeeping”
  - Can be instruction-grained, or block-grained (4-5 ops)

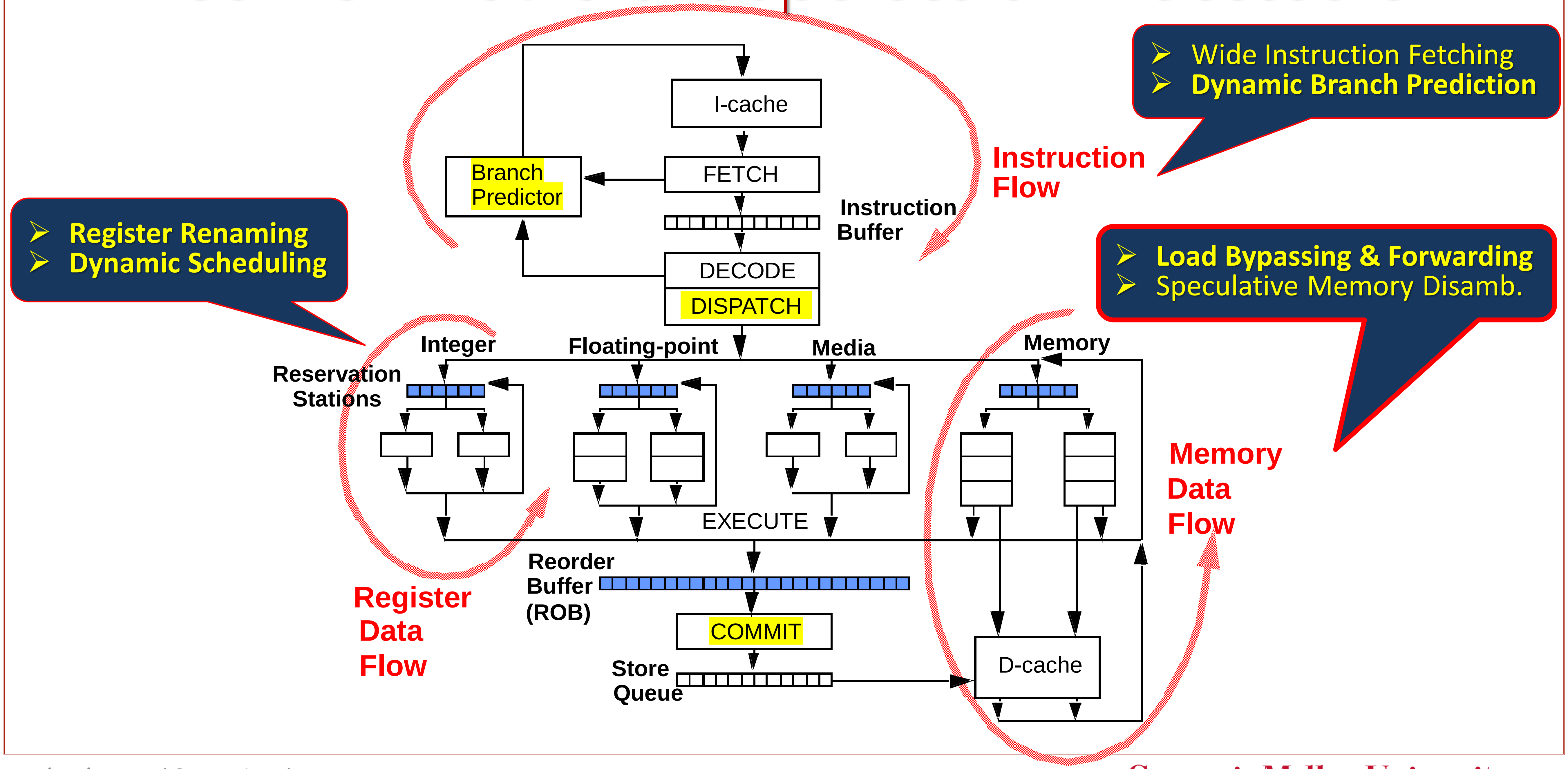
Elements of Modern Micro-Dataflow Engine



Dynamic Scheduling Implementation Cost

- To support N-way dispatch per cycle
  - Nx2 simultaneous lookups into the rename map (or associative search)
  - N simultaneous write ports into the IW and the ROB
- To support N-way issue per cycle (assuming read at issue)
  - 1 prioritized associative lookup of N entries
  - N read ports into the IW
  - Nx2 read ports into the RF
- To support N-way complete per cycle
  - N write ports into the RF and the ROB
  - Nx2 associative lookup and write in IW
- To support N-way retire per cycle
  - N read ports in the ROB
  - N ports into the RF (potentially)

Three Flow Paths of Superscalar Processors



Memory Operations and Memory Data Flow

- So far, we only considered register-register instructions
  - Add, sub, mul, branch, jump,
- Loads and Stores
  - Necessary because we don't have enough registers for everything
    - Memory allocated objects, register spill code
  - RISC ISAs: only loads and stores access memory
  - CISC ISAs: memory micro-ops are essentially RISC loads/stores
- Steps in load/store processing
  - Generate address (not fully encoded by instruction)
  - Translate address (virtual ⇒ physical) [due to virtual memory]
  - Execute memory access (actual load/store)

Memory Data Dependencies

- Besides branches, long memory latencies are one of the biggest performance challenges today.
- To preserve sequential (in-order) state in the data caches and external memory (so that recovery from exceptions is possible) stores are performed in order. This takes care of anti-dependences and output dependences to memory locations.
- However, loads can be issued out of order with respect to stores if the out-of-order loads check for data dependences with respect to previous, pending stores.

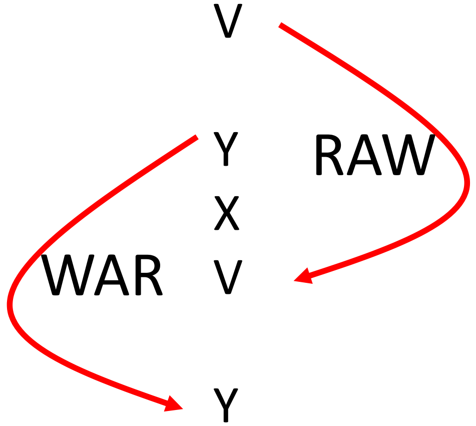
WAW	WAR	RAW
store X	load X	store X
:	:	:
store X	store X	load X

Memory Data Dependency Terminology

- "**Memory Aliasing**" = Two memory references involving the same memory location (collision of two memory addresses).
- "**Memory Disambiguation**" = Determine whether two memory references will alias or not (whether there is a dependence or not).
- **Memory Dependency Detection**:
  - Must compute effective addresses of both memory references
  - Effective addresses can depend on run-time data and other instructions
  - Comparison of addresses require much wider comparators

Example code:

(1)	STORE	V
(2)	ADD	
(3)	LOAD	Y
(4)	LOAD	X
(5)	LOAD	V
(6)	ADD	
(7)	STORE	Y

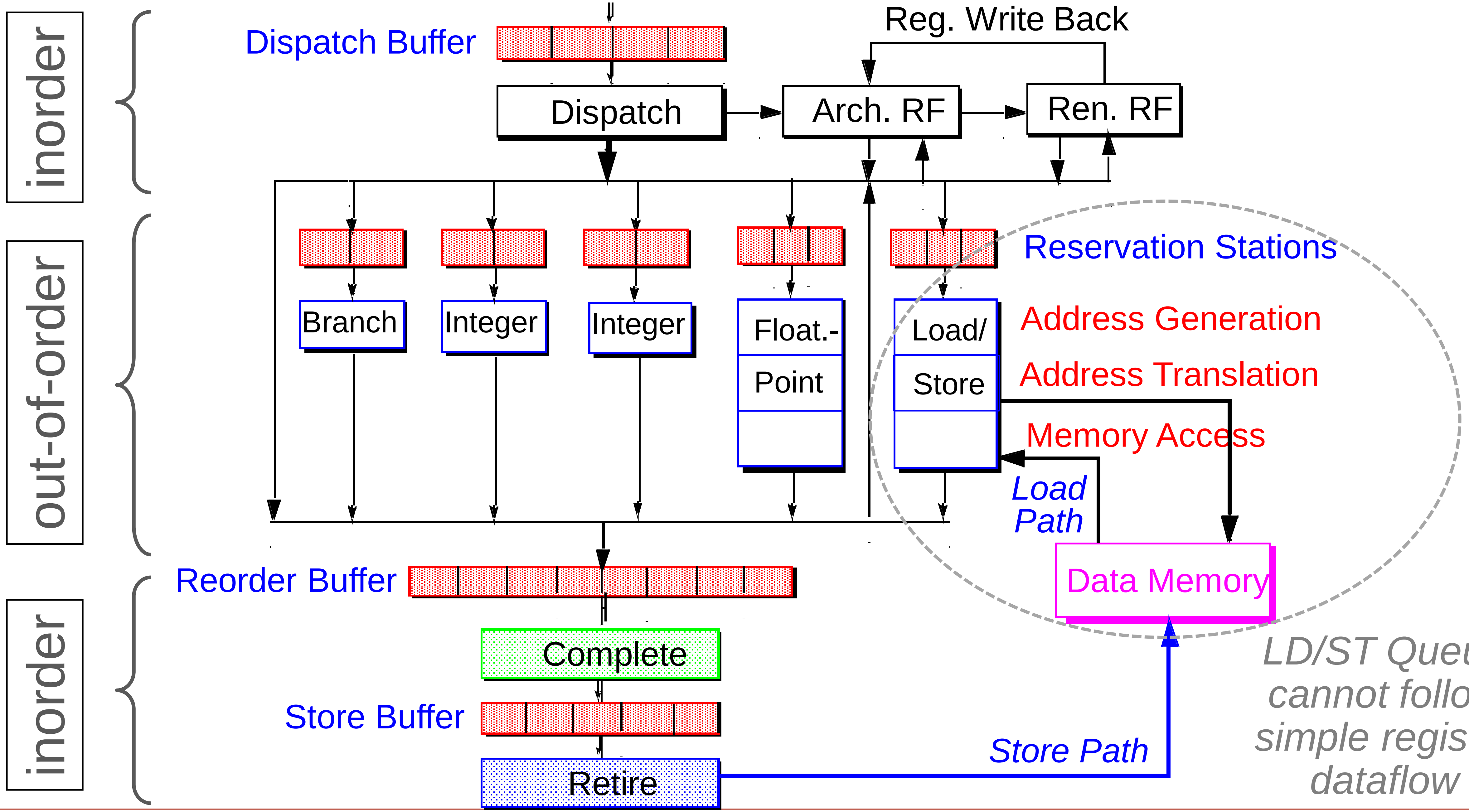


In-Order (Total Ordering) Load/store Processing

- Stores
  - Allocate store buffer entry at DISPATCH (in-order)
  - When register value available, issue and calculate address ("finished")
  - When all previous instructions retire, store considered completed
    - Store buffer split into "finished" and "completed" part through pointers
  - Completed stores go to memory in order

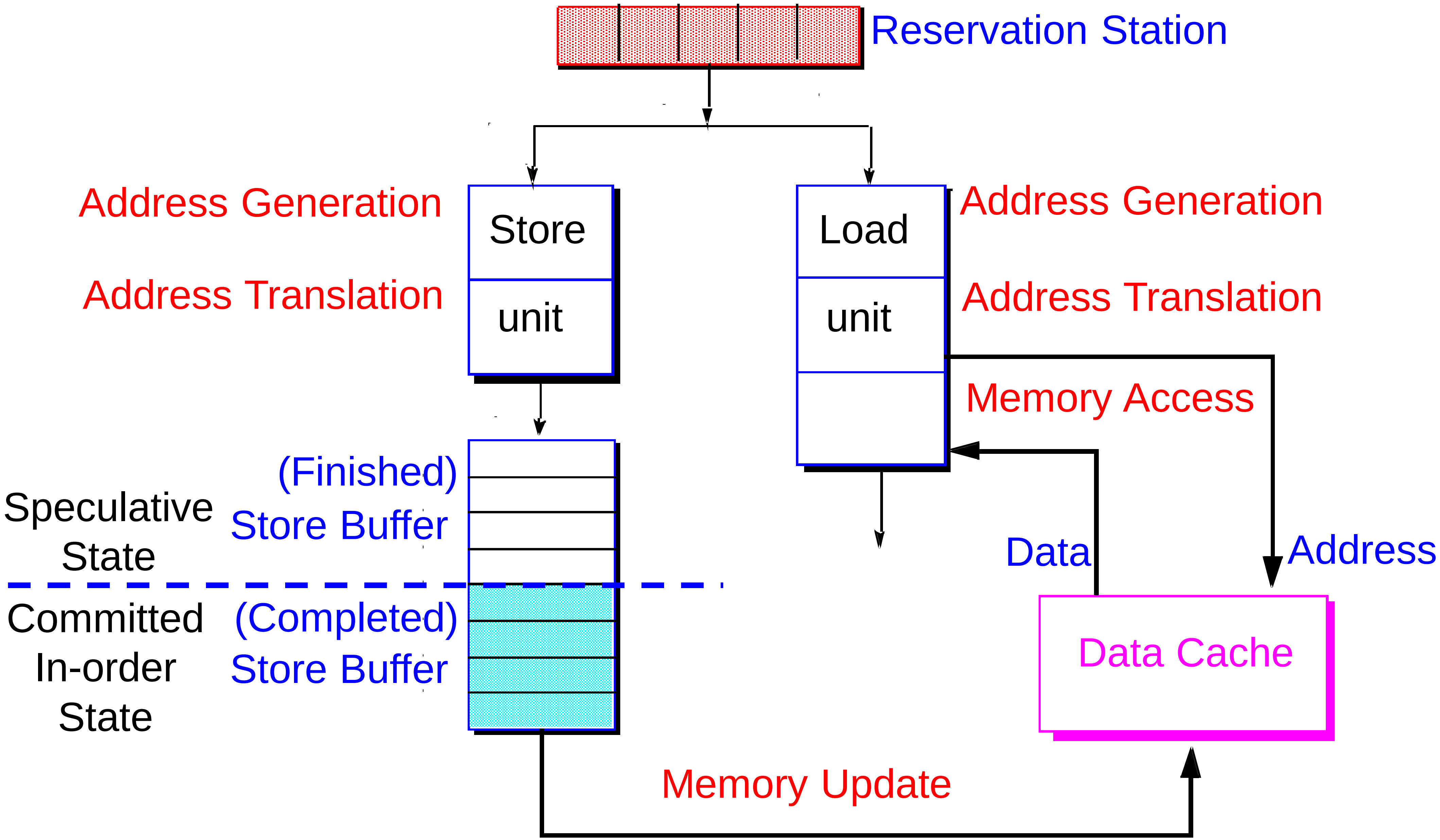
- Loads
  - Loads remember the store buffer entry of the last store before them
  - A load can issue when
    - Address register value is available AND
    - All older stores are considered "completed"

Processing of Load/Store Instructions

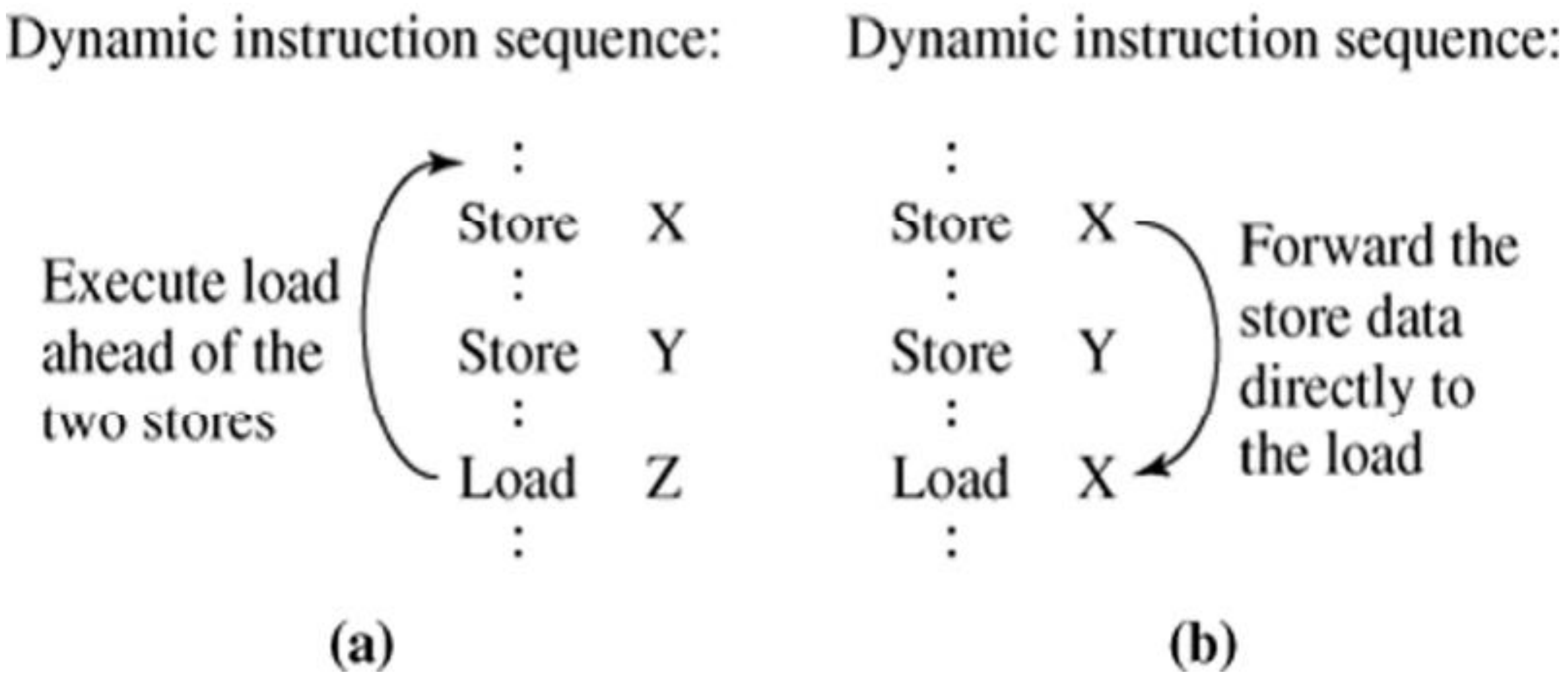


LD/ST Queue cannot follow simple register dataflow

Load/Store Units and Store Buffer



Load Bypassing & Load Forwarding: Motivation



Load Bypassing

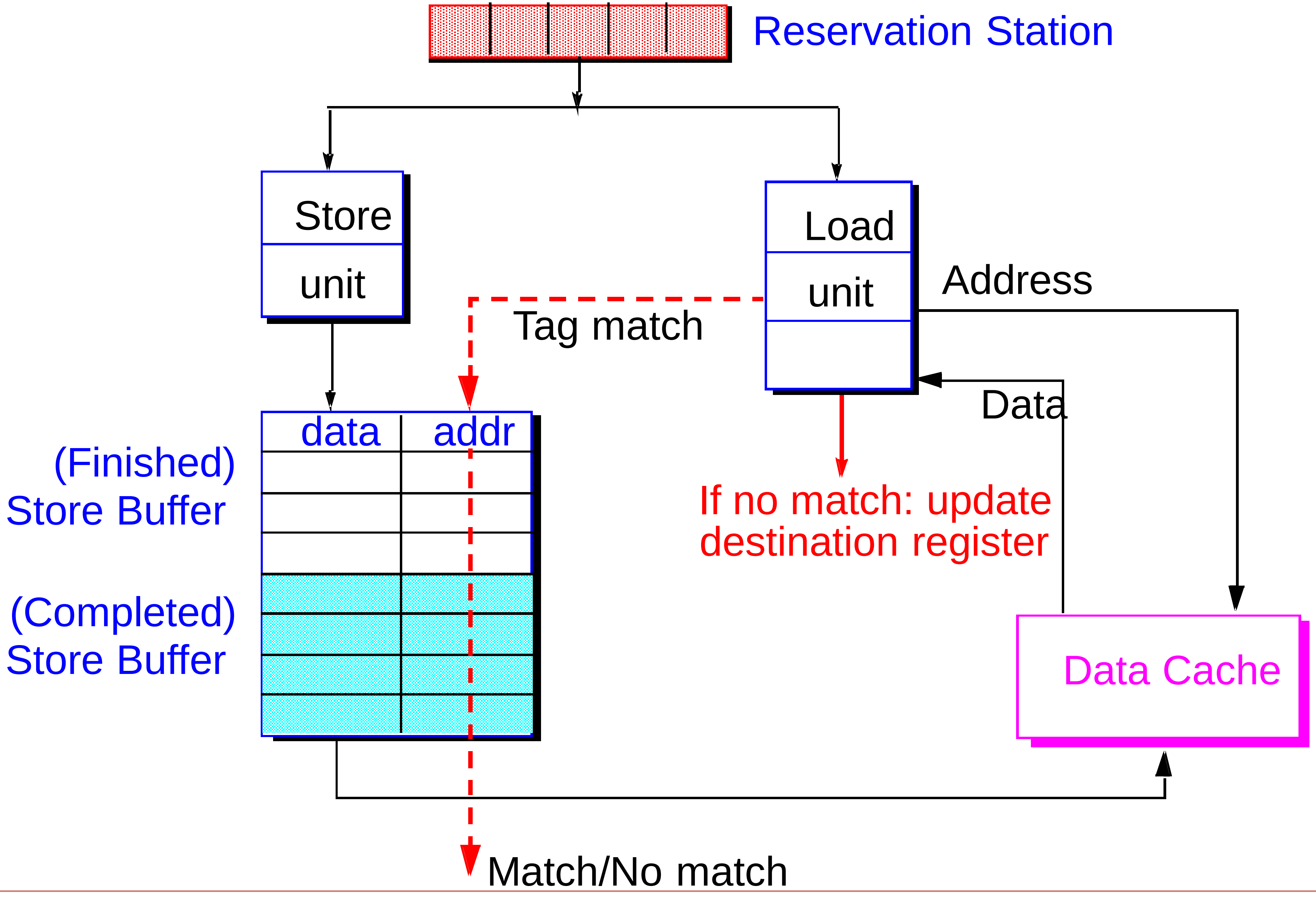
Load Forwarding

Load Bypassing

- Loads can be allowed to bypass older stores if no aliasing is found
  - Older stores' addresses must be computed before loads can be issued to allow checking for RAW load dependences. If dependence cannot be checked, e.g. store address cannot be determined, then all subsequent loads are held until address is valid (conservative).
- Alternatively a load can assume no aliasing and bypass older stores *speculatively*
  - Validation of no aliasing with previous stores must be done and mechanism for reversing the effect must be provided.
- Stores are kept in ROB until all previous instructions complete, and kept in the store buffer until gaining access to cache port.
  - At completion time, a store is moved to the Completed Store Buffer to wait for turn to access cache. Store buffer is "future file" for memory.

Store is consider completed. Latency beyond this point has little effect on the processor throughput.

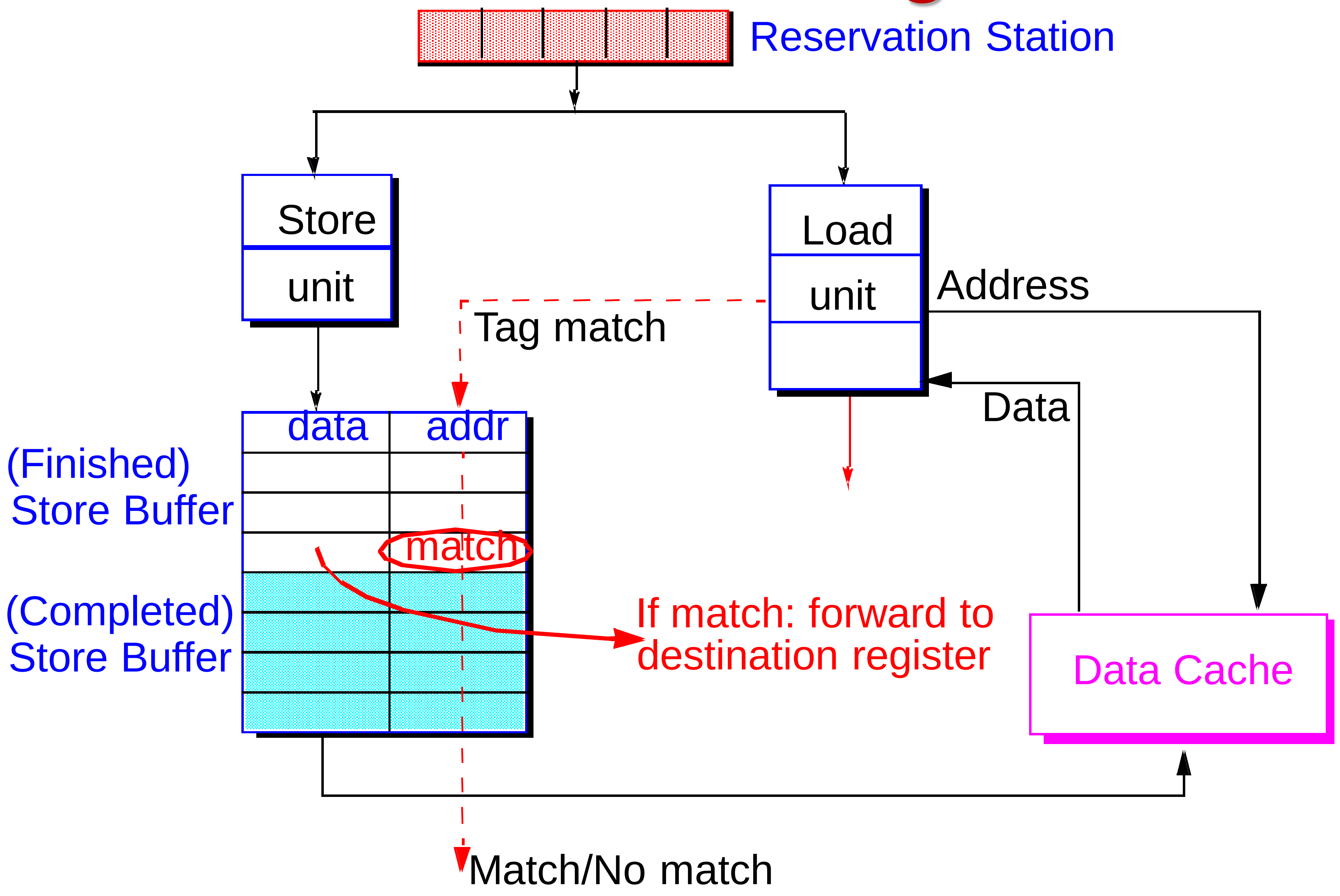
Illustration of Load Bypassing



Load Forwarding

- If a pending load is RAW dependent on an earlier store still in the store buffer, it need not wait till the store is issued to the data cache
- The load can be directly satisfied from the store buffer if both load and store addresses are valid and the data is available in the store buffer
- Since data is sourced directly from the store buffer, this avoids the latency (and power consumption) of accessing the data cache

Illustration of Load Forwarding

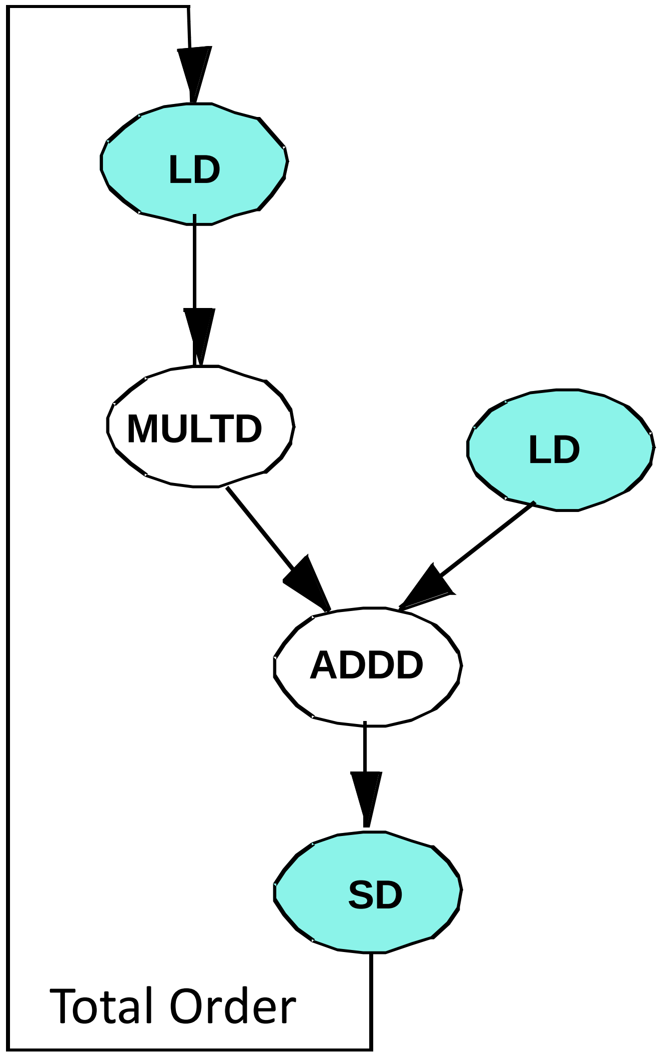


The "DAXPY" Example

Y(i) = A * X(i) + Y(i)

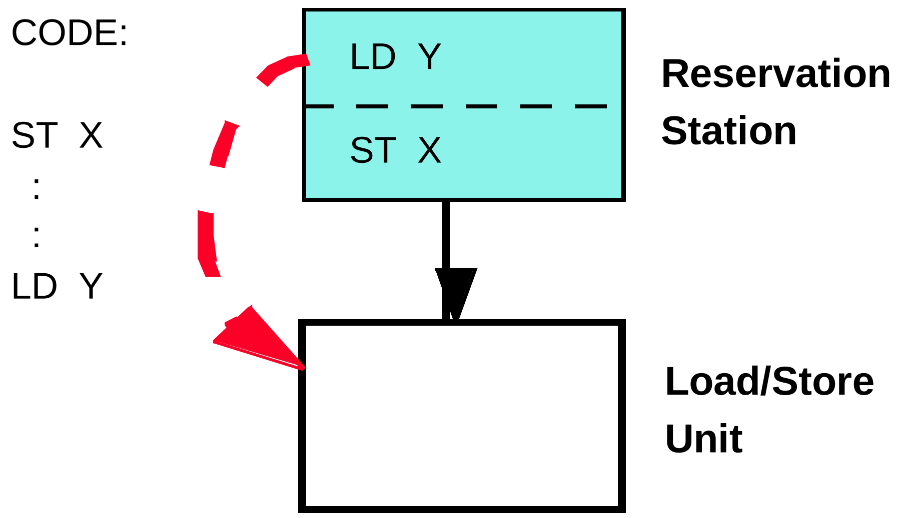
```
      LD      F0, a
      ADDI    R4, Rx, #512          ; last address

Loop:
      LD      F2, 0(Rx)             ; load X(i)
      MULTD   F2, F0, F2            ; A*X(i)
      LD      F4, 0(Ry)             ; load Y(i)
      ADDD    F4, F2, F4            ; A*X(i) + Y(i)
      SD      F4, 0(Ry)             ; store into Y(i)
      ADDI    Rx, Rx, #8            ; inc. index to X
      ADDI    Ry, Ry, #8            ; inc. index to Y
      SUB     R20, R4, Rx           ; compute bound
      BNZ     R20, loop             ; check if done
```

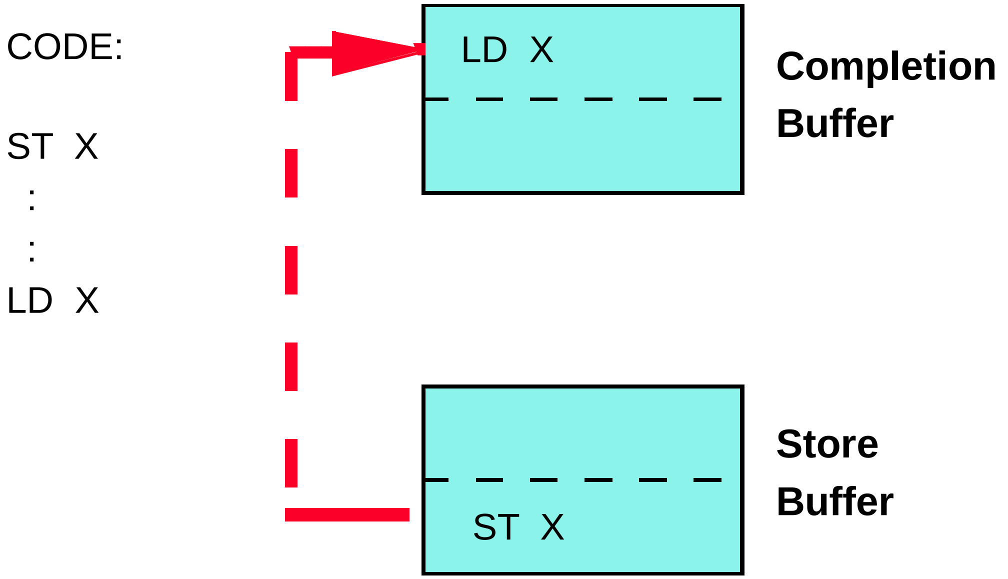


Performance Gains From Weak Ordering

Load Bypassing:



Load Forwarding:

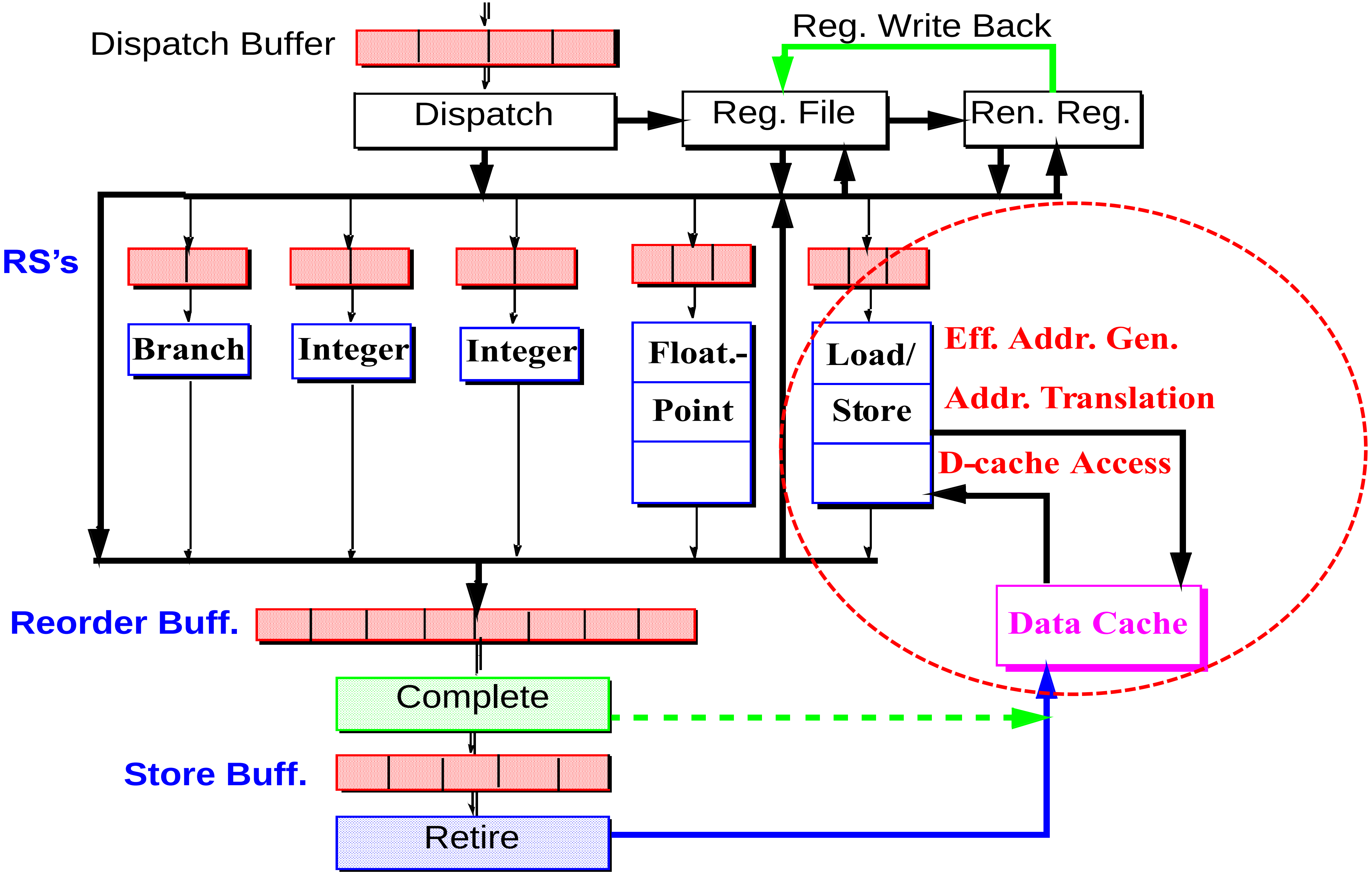


Performance gain:

Load bypassing: 11%-19% increase over total ordering

Load forwarding: 1%-4% increase over load bypassing

The Memory Bottleneck



Memory Bottleneck Techniques

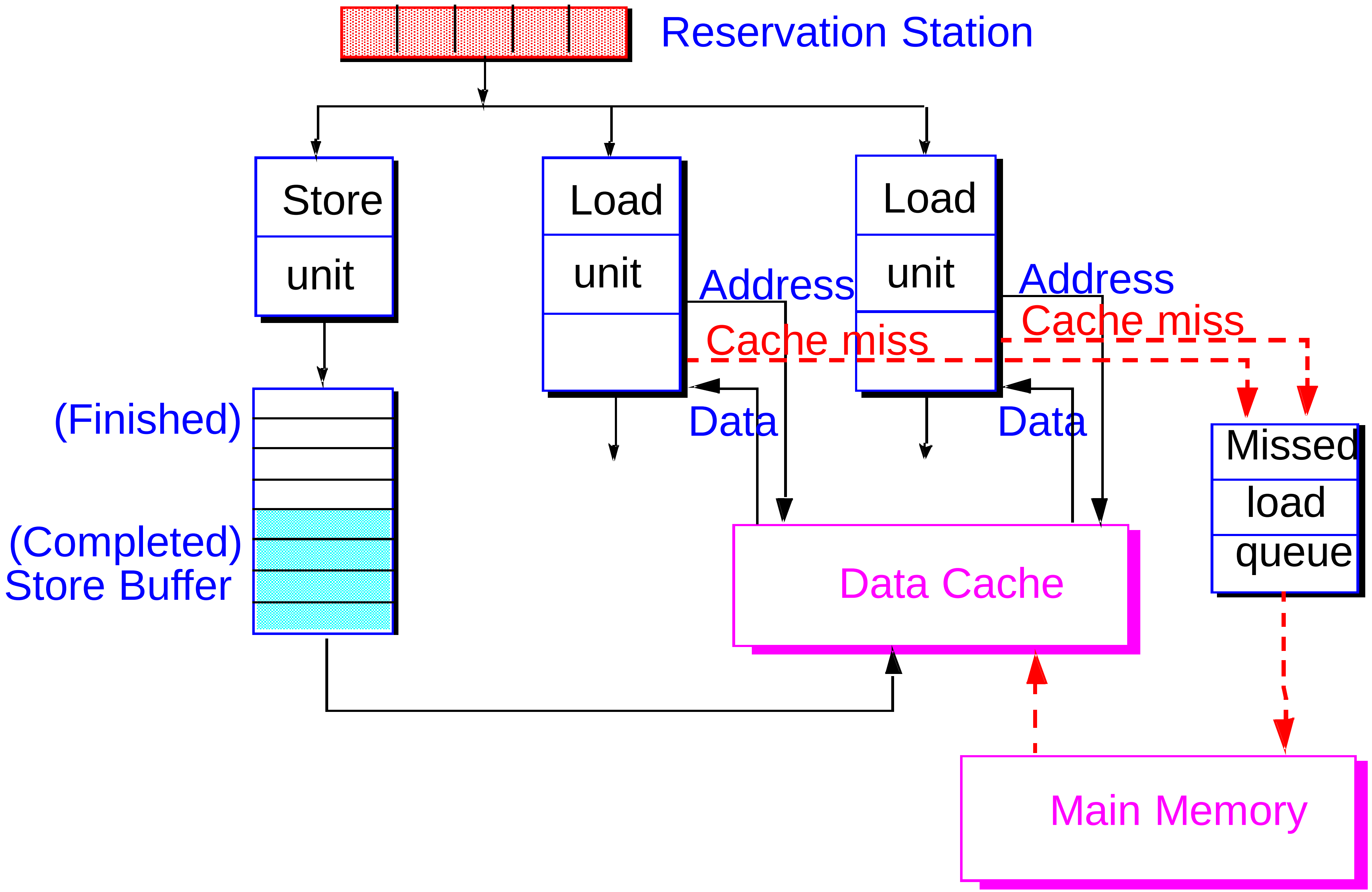
Dynamic Hardware (Microarchitecture):

- Use Multiple Load/Store Units (need multiported D-cache)
- Use More Advanced Caches (victim cache, stream buffer)
- Use Hardware Prefetching (need load history and stride detection)
- Use Non-blocking D-cache (need missed-load buffers/MSHRs)
- Large instruction window (memory-level parallelism)

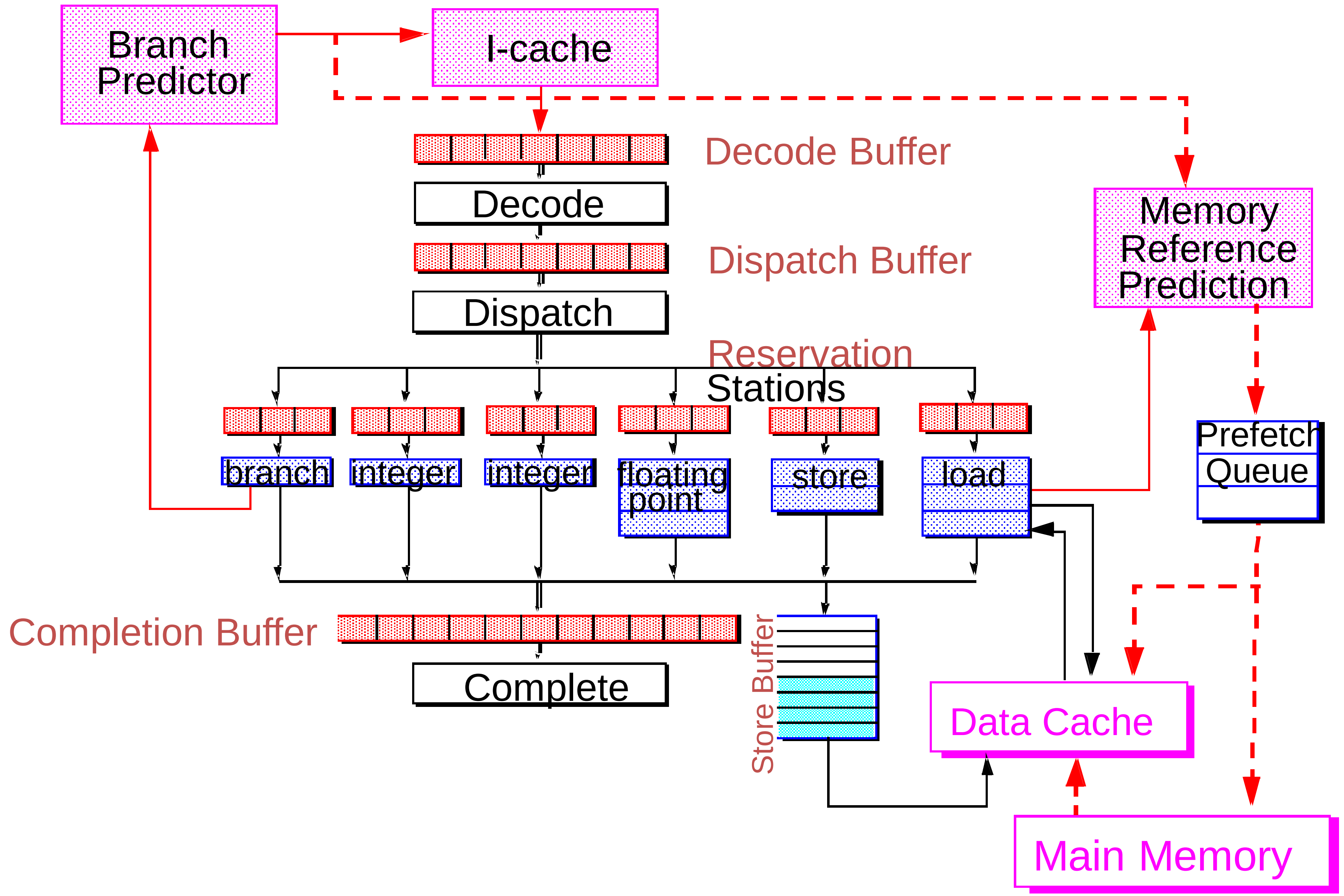
Static Software (Code Transformation):

- Insert Prefetch or Cache-Touch Instructions (mask miss penalty)
- Array Blocking Based on Cache Organization (minimize misses)
- Reduce Unnecessary Load/Store Instructions (redundant loads)
- Software Controlled Memory Hierarchy (expose it to above DSI)

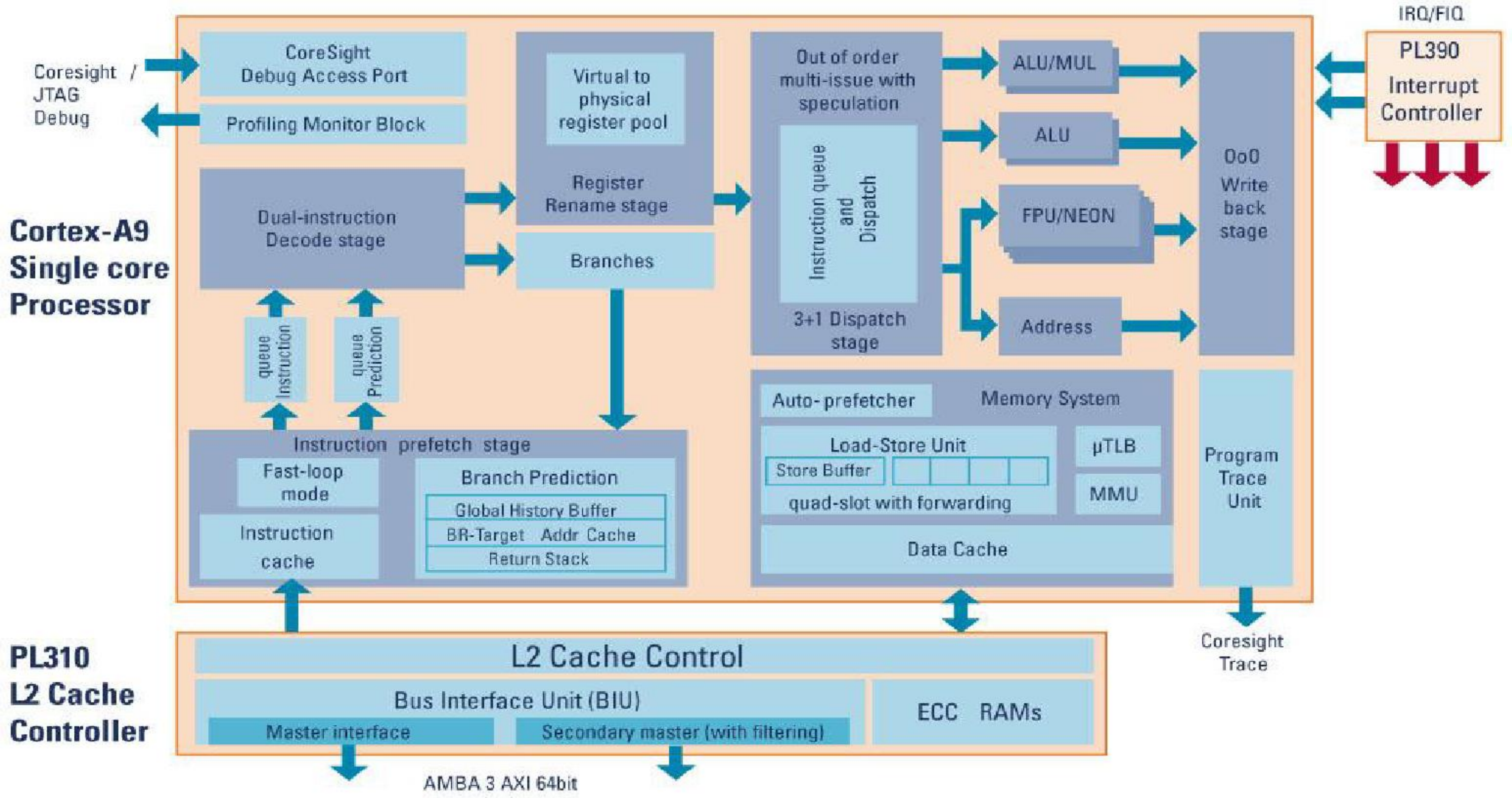
Dual-Ported Non-Blocking Cache



Prefetching Data Cache



Cortex-A9 Single Core Microarchitecture

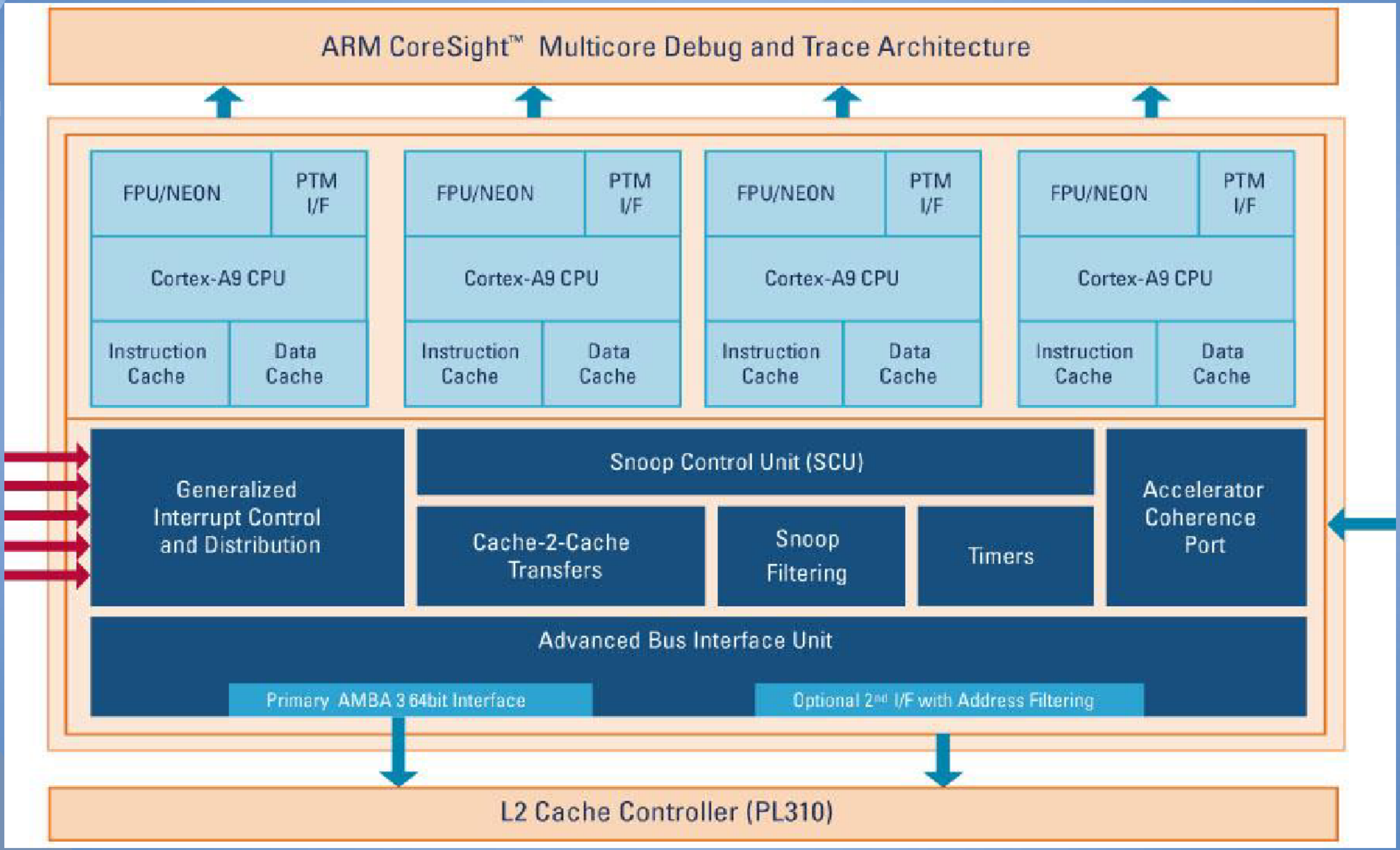


Cortex-A9 Microarchitecture Structure and the Single Core Interfaces

Introduction

- ARM Cortex-A9 is the 2nd generation of ARM MPCore technology series
- High performance
- Uses ARMv7-A ISA
- Used many embedded devices due to its ability to control different level of power consumption
  - essential for mobile devices

Cortex-A9 MultiCore Processor



Cortex-A9 Multicore Processor Configuration

Apple A5

- IPhone 4s, IPad2, IPad mini
- consists of a dual-core ARM Cortex-A9 MPCore CPU
- Max. CPU clock rate
  - 0.8GHz for IPhone 4s
  - 1GHz for IPad2, mini\
- L1 cache: 32 KB instruction + 32 KB data
- L2 cache: 1 MB

PlayStation Vita SoC

- Four Cortex-A9 processors
- Graphics: Quad-core PowerVR SGX543MP4+
- 2 GHz CPU clock rate
- Power: 2200 mAh, 3-5 hours
- 2.2 million units sold

Exynos 4

- Samsung Galaxy S III, Galaxy Note
- Quad-core ARM Cortex A-9
- CPU: 1.4-1.6 GHz
- over 10 million note sold
- over 50 million of S III sold

18-600 Foundations of Computer Systems

Lecture 10:
"The Memory Hierarchy"

John P. Shen & Gregory Kesden

October 2, 2017

Next Time …

- Required Reading Assignment:
  - **Chapter 6 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.**
- Recommended Reference:
  - **Chapter 3 of Shen and Lipasti (SnL).**

Electrical & Computer ENGINEERING