18-600 Foundations of Computer Systems

Lecture 11: "Modern Superscalar Out-of-Order Processors"

John P. Shen & Zhiyi Yu
October 5, 2016

- 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

Carnegie Mellon University

18-600 Foundations of Computer Systems

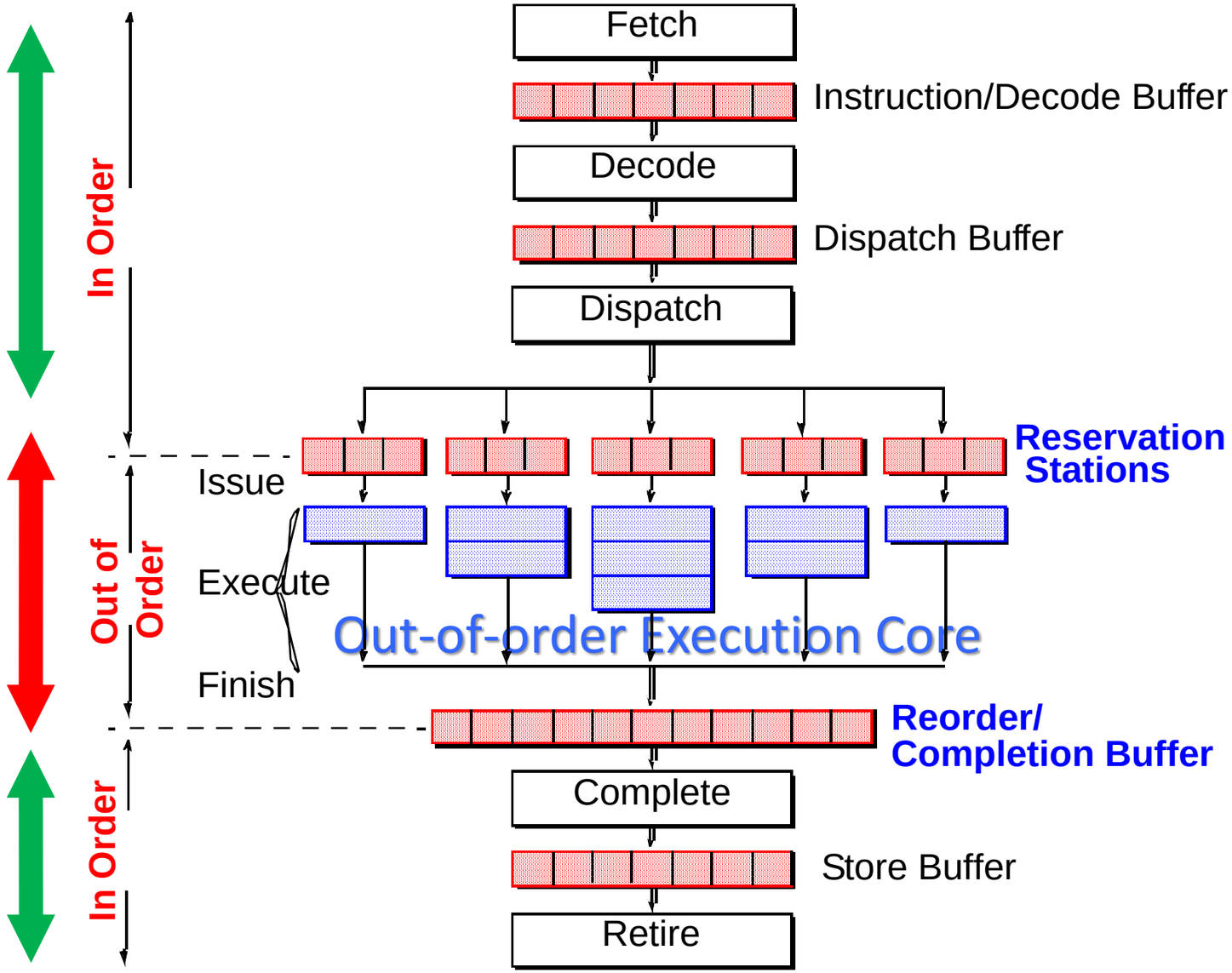
Lecture 11: "Modern Superscalar Out-of-Order Processors"

A. Register Data Flow Techniques
   a. Resolving Anti and Output Dependencies
   b. Resolving True Dependencies
   c. Dynamic Out-of-Order Execution

B. Memory Data Flow Techniques
   a. Memory Data Dependencies
   b. Load Bypassing & Load Forwarding
   c. Speculative Disambiguation

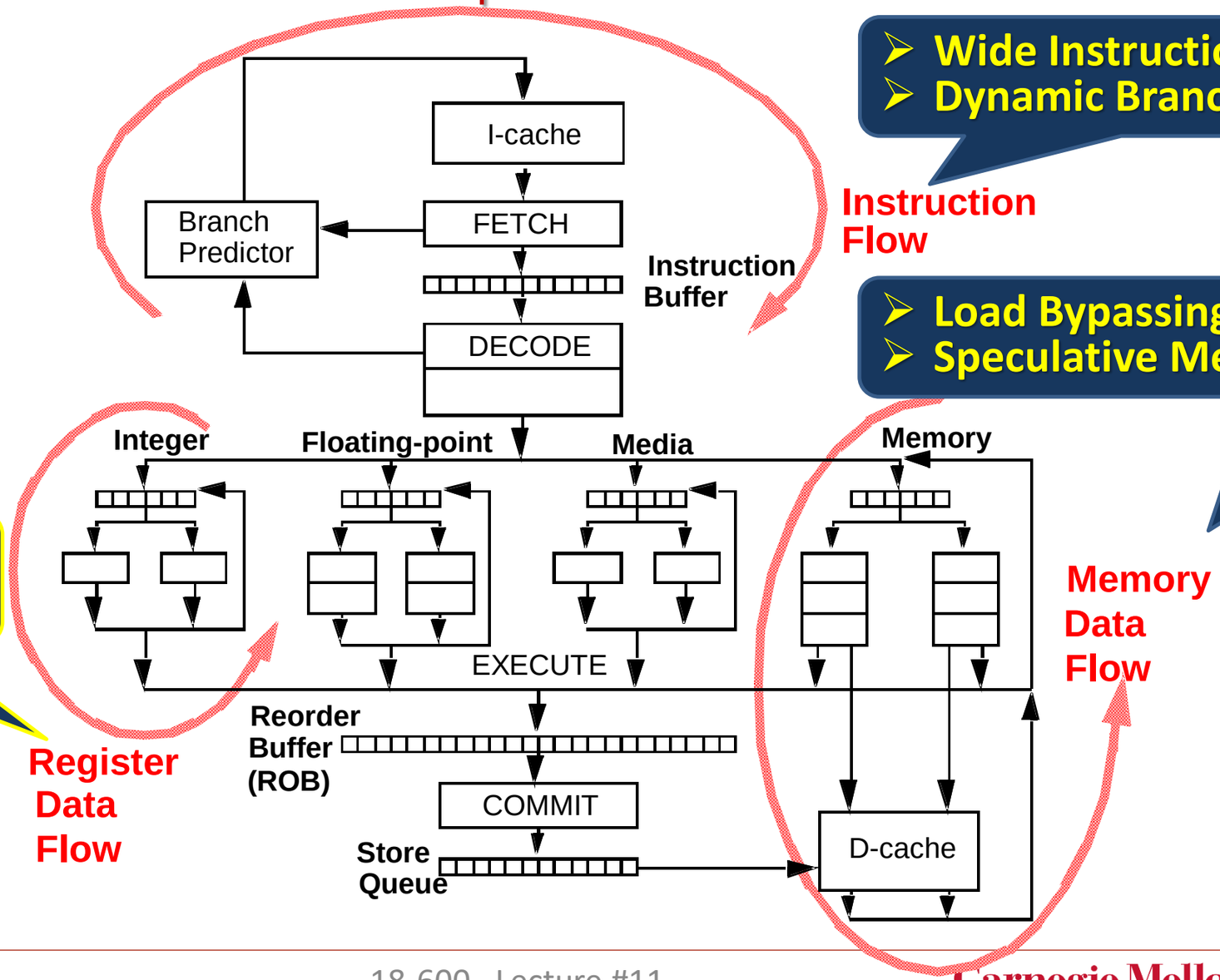
Electrical & Computer ENGINEERING

A 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

Instruction Flow

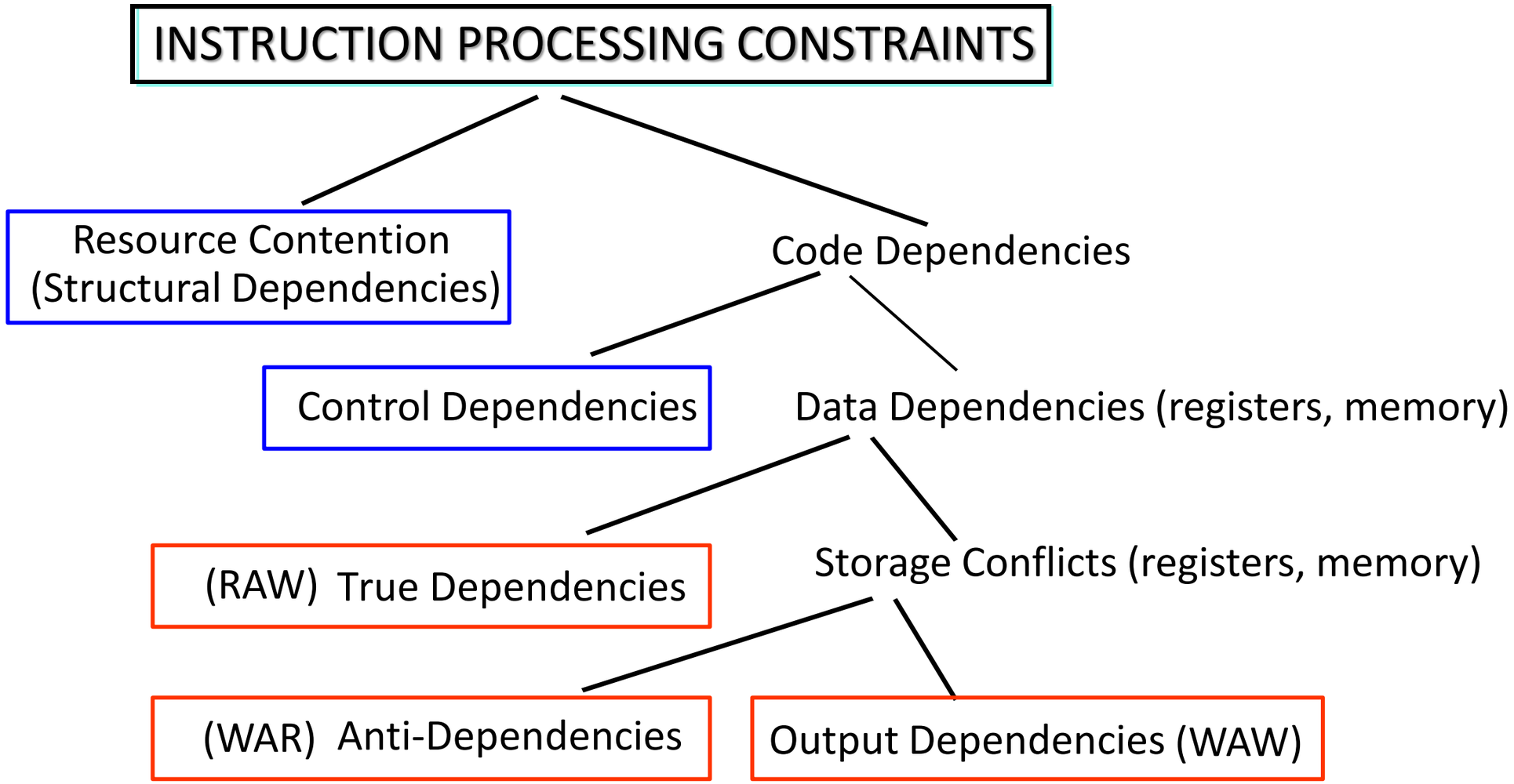
- Load Bypassing & Forwarding
- Speculative Memory Disamb.

Memory Data Flow

- Register Renaming
- Dynamic Scheduling

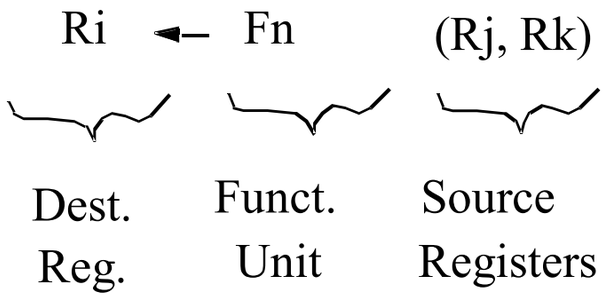
Register Data Flow

The Big Picture: Impediments Limiting ILP



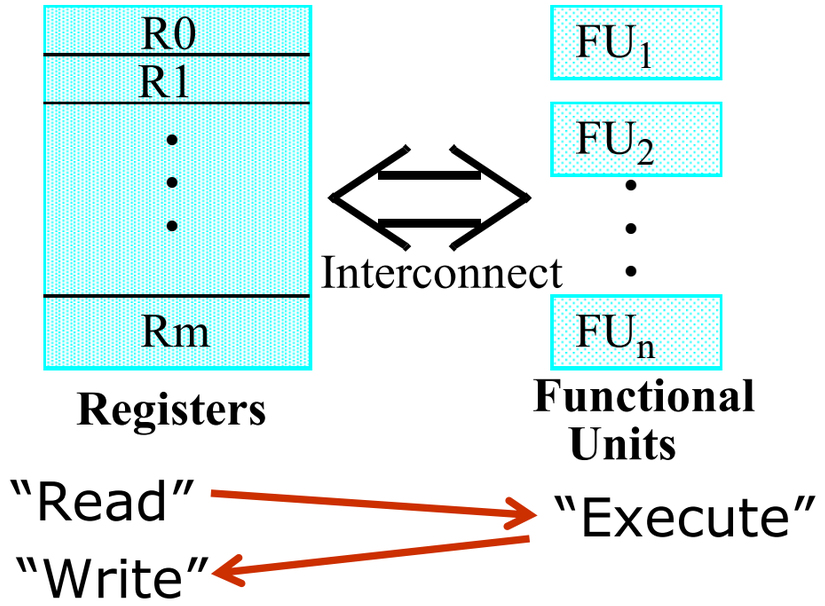
Register Data Flow

Each ALU Instruction:



"Register Transfer"

INSTRUCTION EXECUTION MODEL



- 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

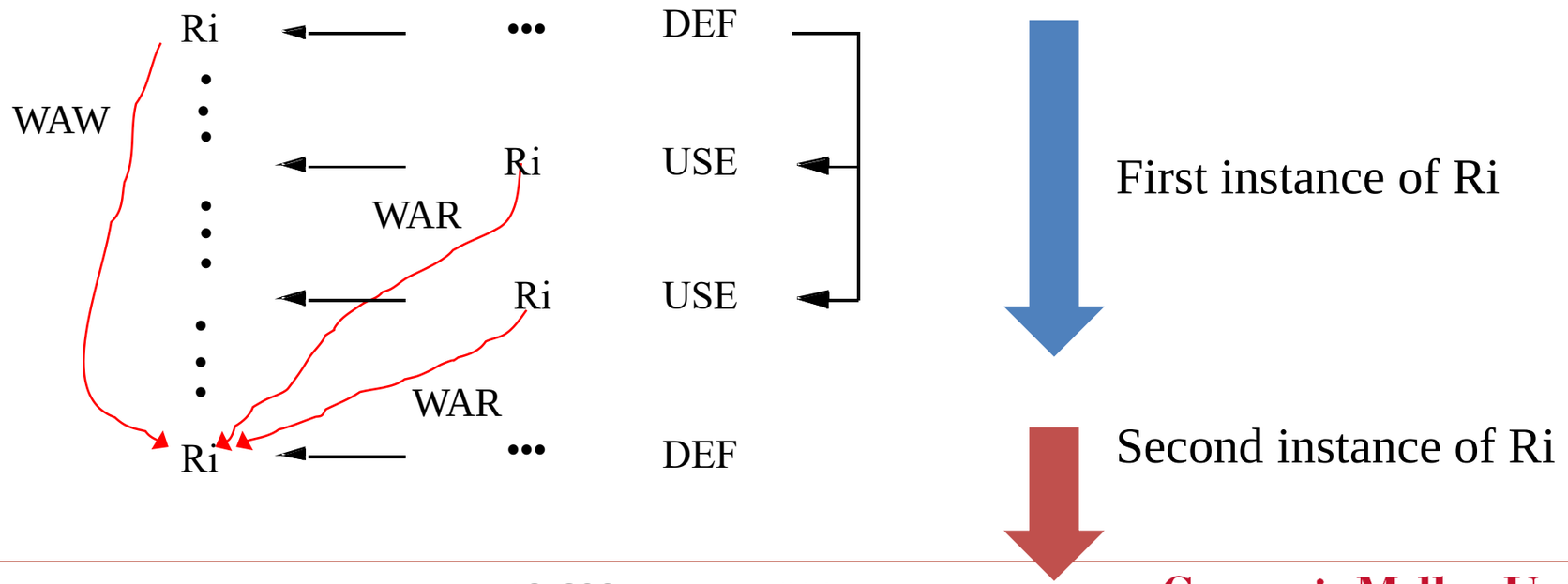
MAXIMIZE USE OF REGISTERS

MULTIPLE ASSIGNMENTS OF VALUES TO REGISTERS

OUT OF ORDER ISSUING AND COMPLETION

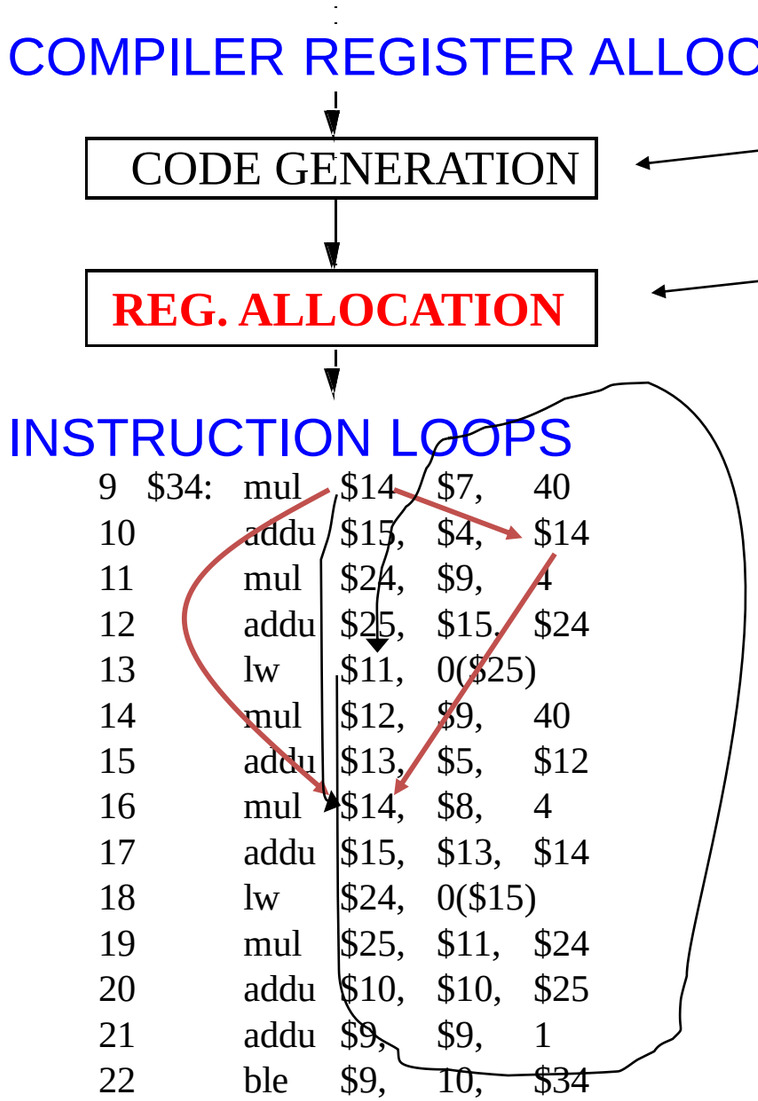
LOSE IMPLIED PRECEDENCE OF SEQUENTIAL CODE

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.

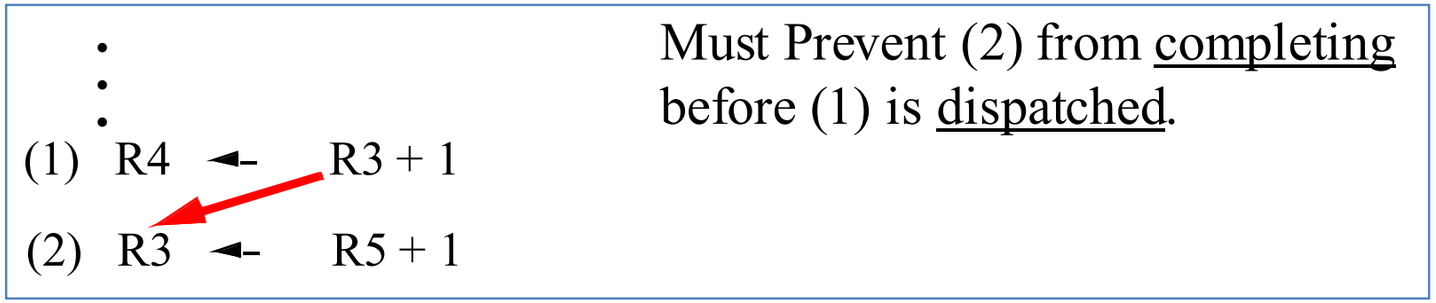
Map Symbolic Reg. to Physical Reg.
Maximize Reuse of Reg.

"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

Resolving Anti-Dependencies



STALL DISPATCHING

DELAY DISPATCHING OF (2)

REQUIRE RECHECKING AND REACCESSING

WAR only

COPY OPERAND

COPY NOT-YET-USED OPERAND TO PREVENT BEING OVERWRITTEN

MUST USE TAG IF ACTUAL OPERAND NOT-YET-AVAILABLE

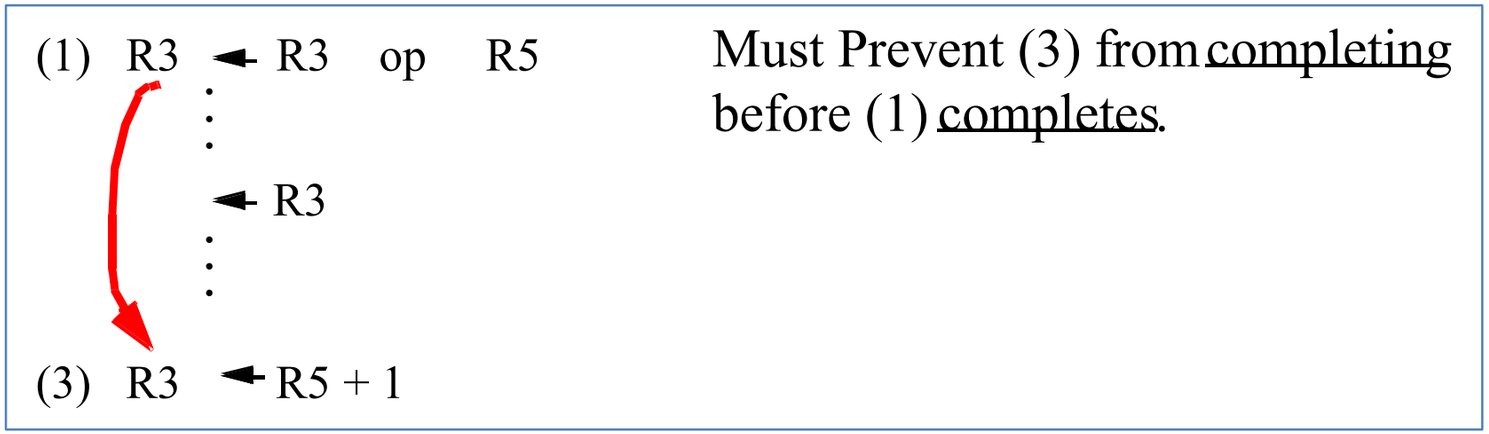
WAR and WAW

RENAME REGISTER

HARDWARE ALLOCATION

```
R3   ← …
     ← R3
R3'  ← …
     ← R3'
```

Resolving Output Dependencies



Must Prevent (3) from completing before (1) completes.

STALL DISPATCHING/ISSUING

DENOTE OUTPUT DEPENDENCY

HOLD DISPATCHING UNTIL RESOLUTION OF DEPENDENCY

ALLOW DECODING OF SUBSEQUENT INSTRUCTIONS

RENAME REGISTER

HARDWARE ALLOCATION

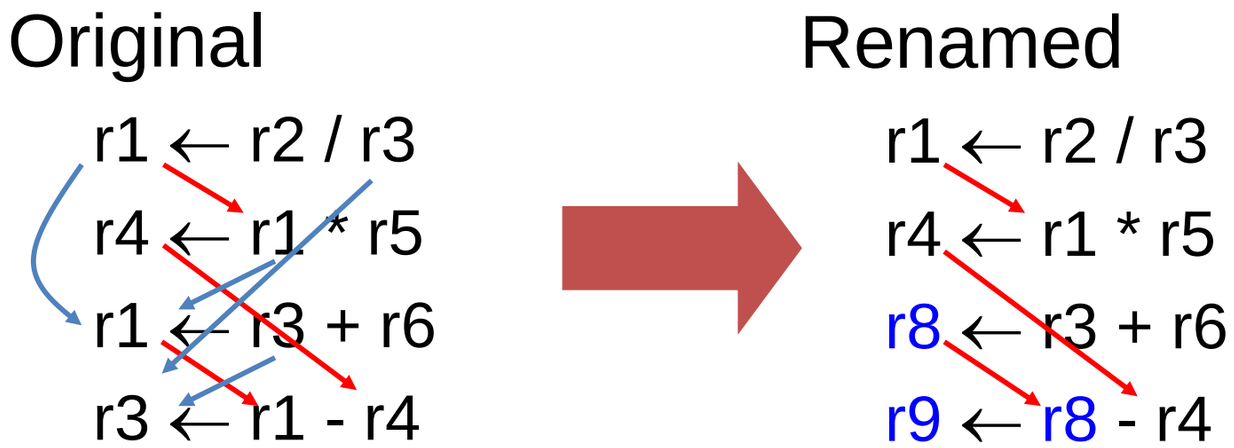
```
R3   ← …
     ← R3
R3'  ← …
     ← R3'
```

Register Renaming: The Idea

- Anti and output dependencies are false dependencies

$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4
$r_3 \leftarrow r_6$ op r_7

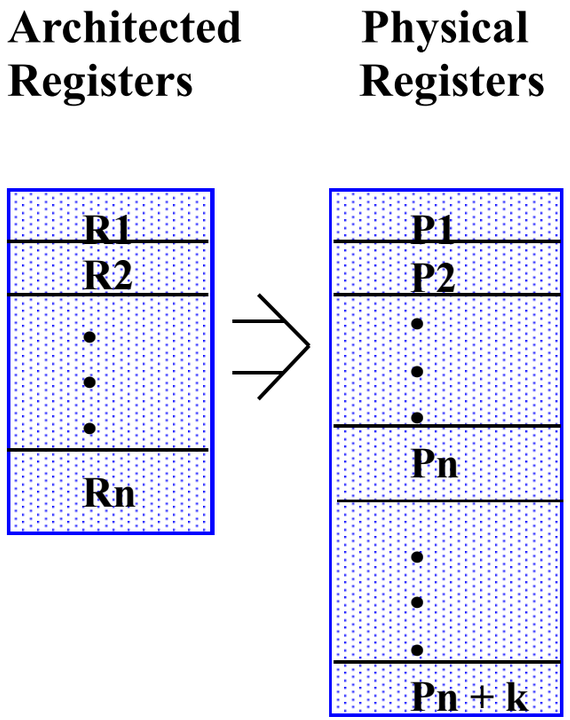
- The dependency is on name/location rather than data
- Given unlimited number of registers, anti and output dependencies can always be eliminated



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

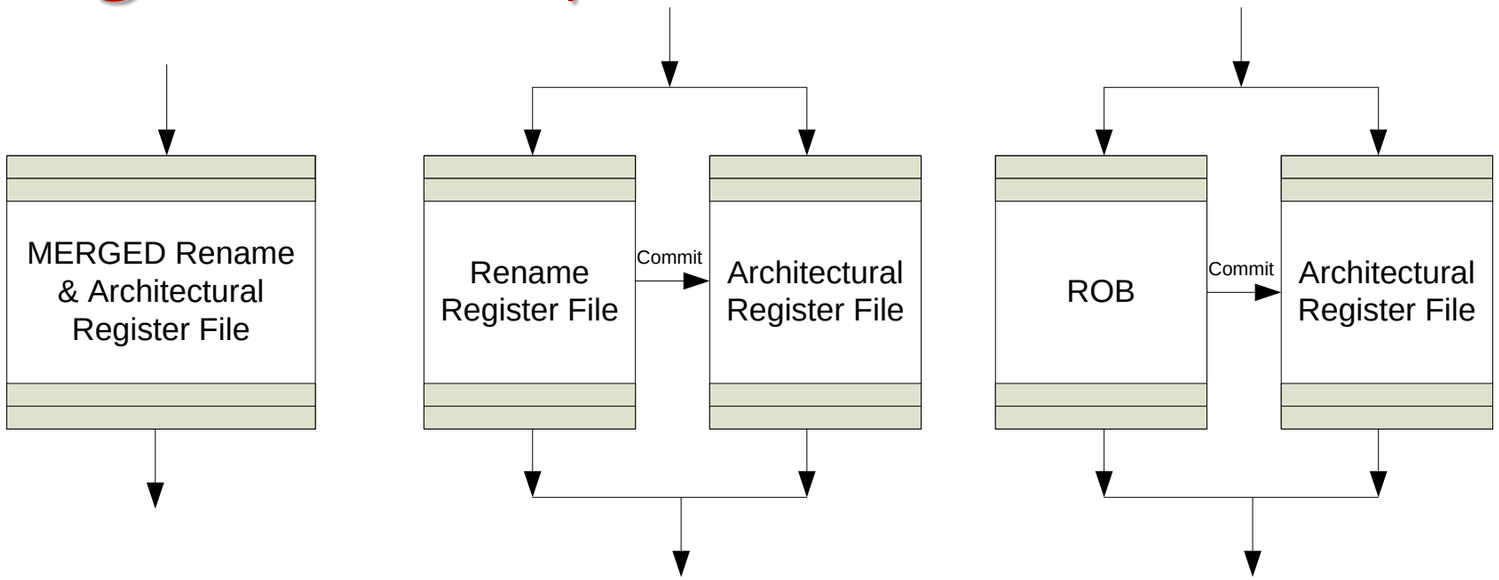
Location:

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

Register Renaming Implementation

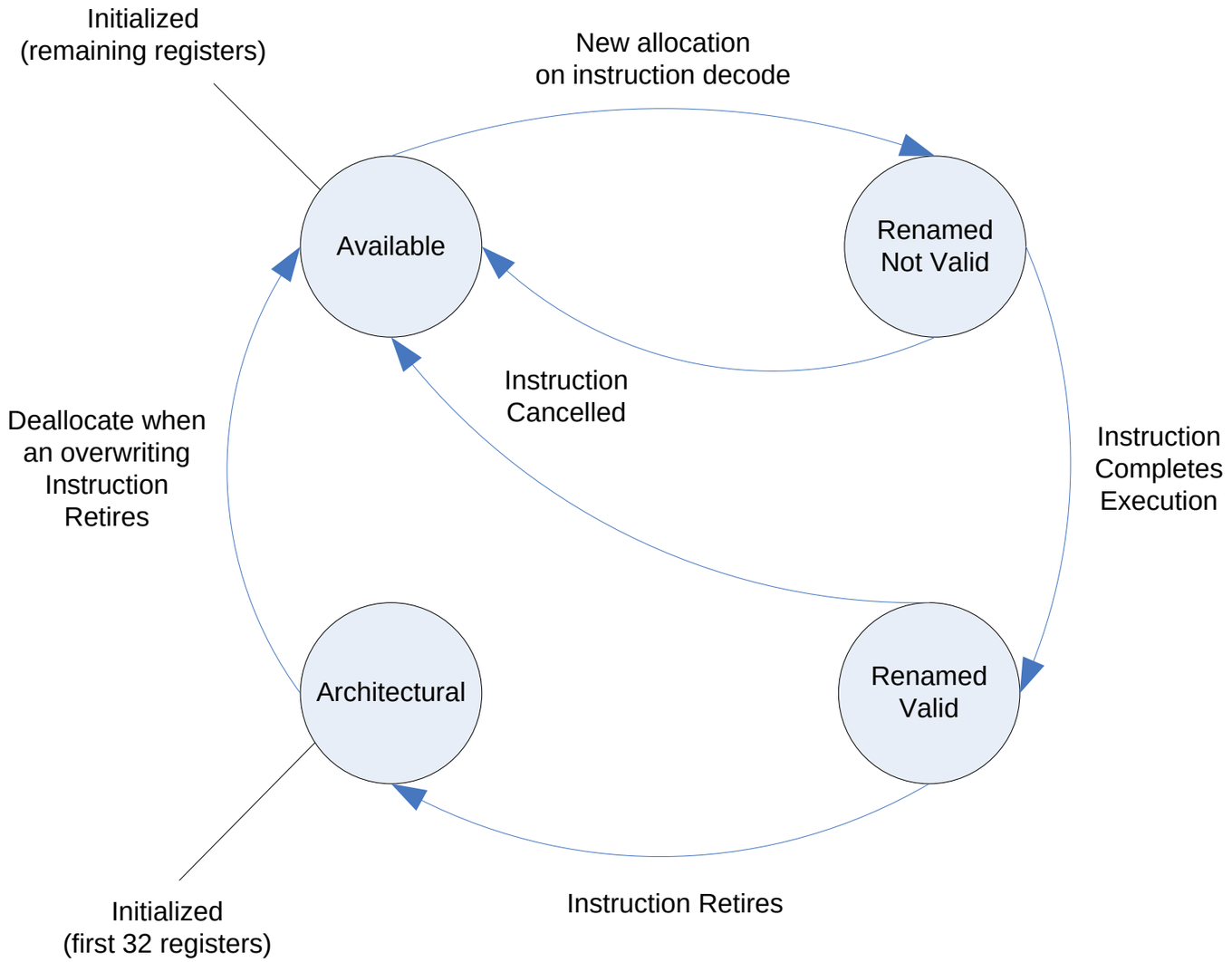
- Renaming:
  - Map small set of architecture registers to a large set of physical registers
  - New mapping for architectural register when it is assigned a new value
- Renaming buffer organization (how are registers stored)
  - Unified RF, split RF, renaming in the ROB
  - RF = register file
- Number of renaming registers
- Number of read/write ports
- Register mapping (how do I find the register I am looking for)
  - Allocation, de-allocation, and tracking

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

Unified Register File: Physical Register FSM



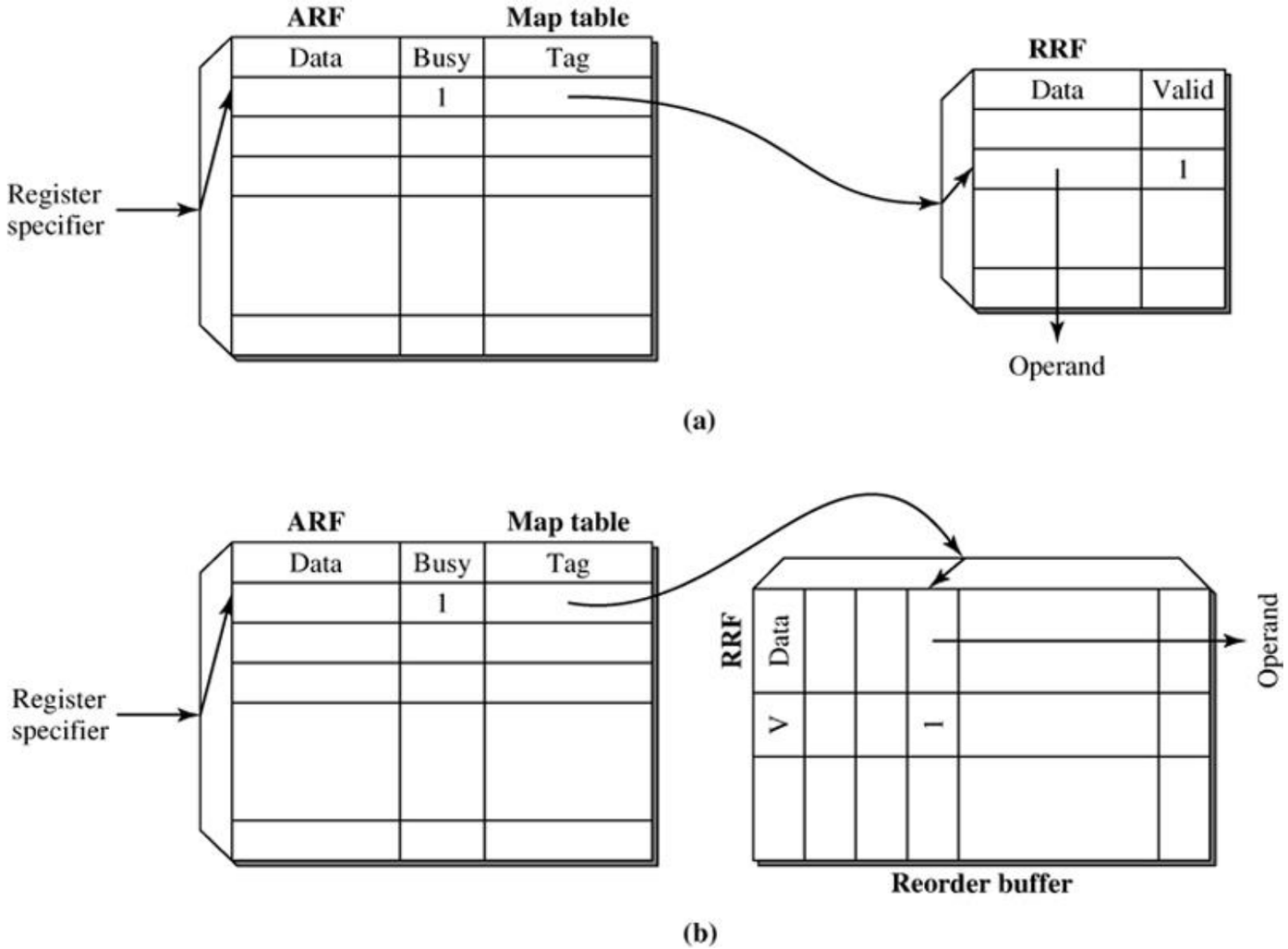
Number of Rename Registers

- 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

Register File Ports

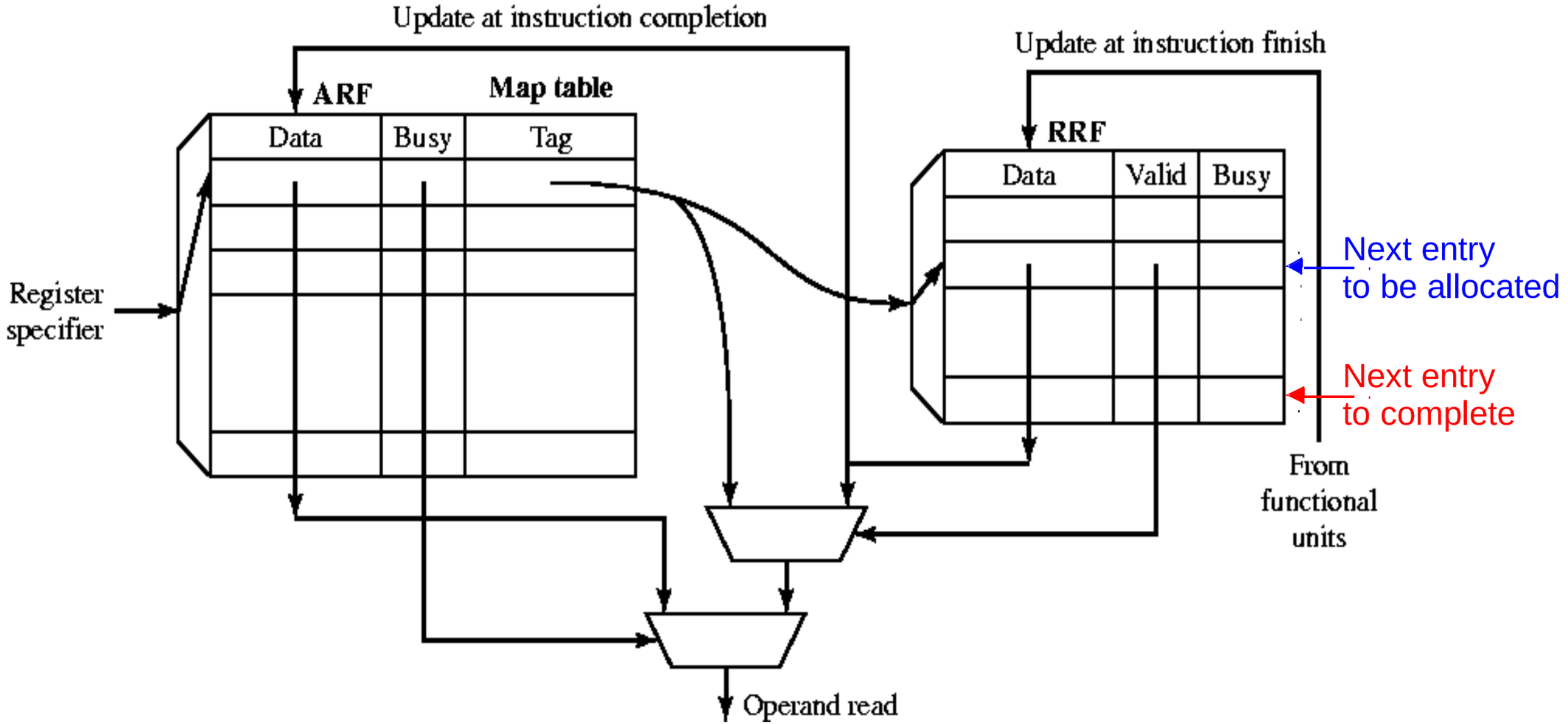
- Read: if operands read as instructions enter scheduler
  - Max # ports = 2 * # instructions dispatched
- Read: if operands read as instruction leave scheduler
  - Max #ports = 2* # instructions issued
- Write: # of FUs or # of instructions committing
  - Depends on unified vs separate rename registers
- Notes:
  - Can implement less ports and have structural hazards
    - Need control logic for port assignment & hazard handling
  - When using separate RRF and ARF, need ports for the final transfer
  - Alternatives to increasing ports: duplicated RF or banked RF
    - What are the issues?

Integrating Map Table with the ARF



Register Renaming Tasks

- Source Read, Destination Allocate, Register Update



Register Renaming in the IBM RS/6000

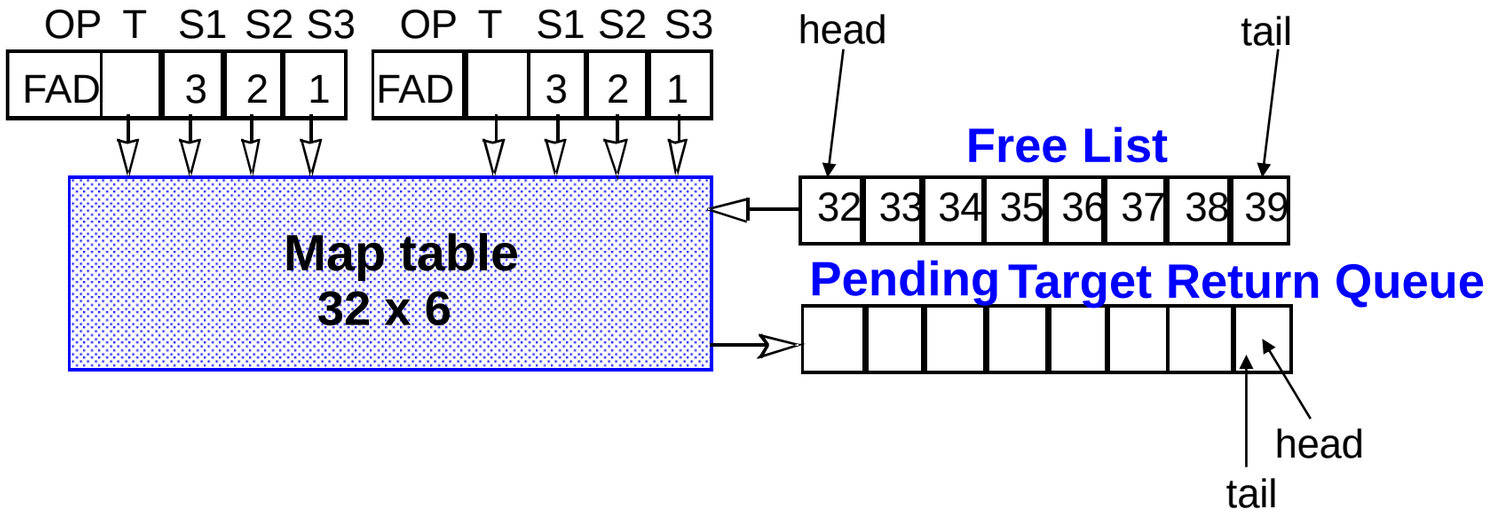
Incoming FPU instructions pass through a renaming table prior to decode

Physical register names only within the FPU!!

32 architectural registers ⇒ 40 physical registers

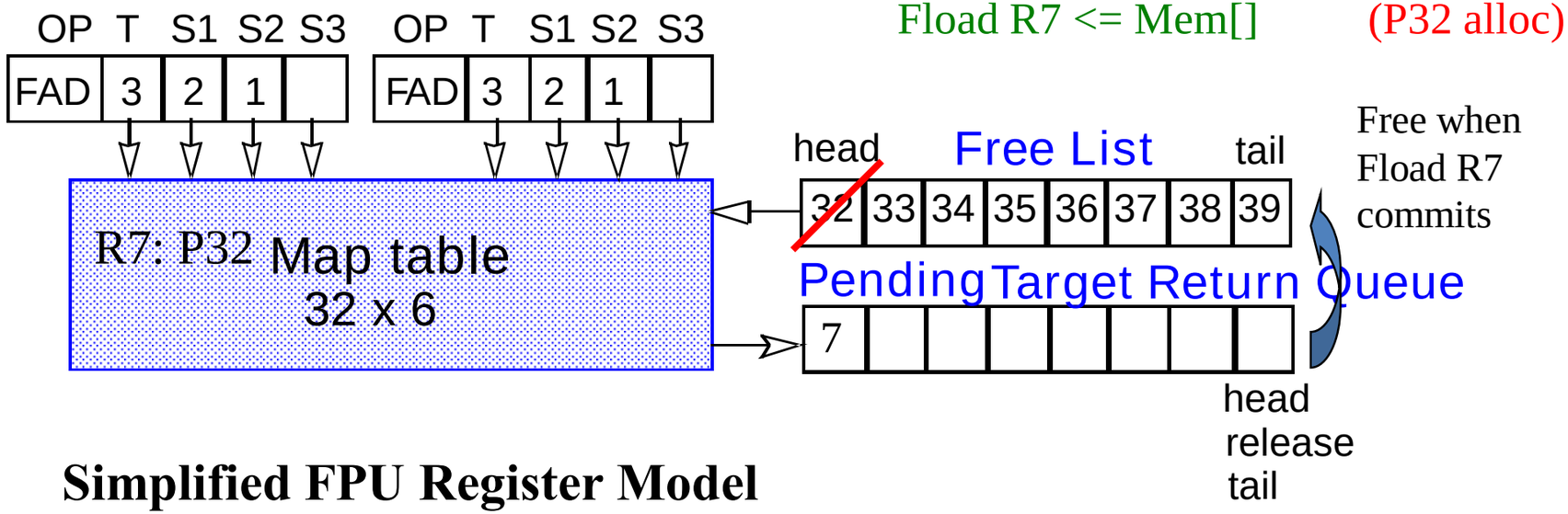
Complex control logic maintains active register mapping

FPU Register Renaming



Register Renaming in the IBM RS/6000 FPU

Fload R7 <= Mem[] (P32 freed)
…
<= R7 (actual last use) (P32)
…
Fload R7 <= Mem[] (P32 alloc)



Simplified FPU Register Model

Incoming FPU instructions pass through a renaming table prior to decode

The 32 architectural registers are remapped to 40 physical registers

- Physical register names are used within the FPU
- Complex control logic maintains active register mapping

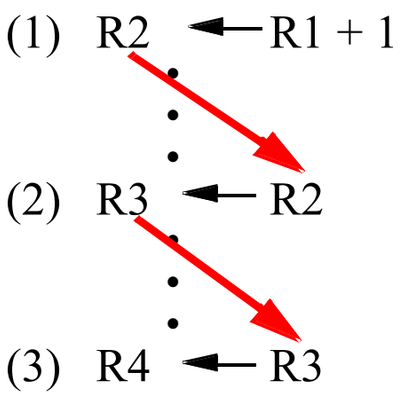
Renaming Difficulties: Wide Instruction Issue

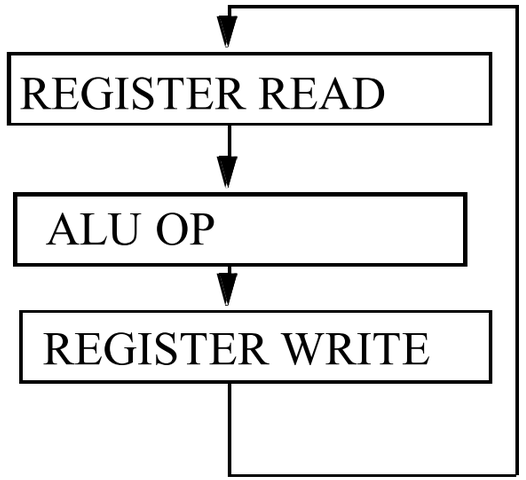
- Need many ports in RFs and mapping tables
- Instruction dependences during dispatching/issuing/committing
  - Must handle dependences across instructions
  - E.g. add R1←R2+R3; sub R6←R1+R5
  - Implementation: use comparators, multiplexors, counters
    - Comparators: discover RAW dependences
    - Multiplexors: generate right physical address (old or new allocation)
    - Counters: determine number of physical registers allocated

Renaming Difficulties: Mispredictions & Exceptions

- On exception/misprediction, register mapping must be precise
- Separate RRF: consider all RRF entries free
- ROB renaming: consider all ROB entries free
- Unified RF: restore precise mapping
  - Single map: traverse ROB to undo mapping (history file approach)
    - ROB must remember old mapping…
  - Two maps: architectural and future register map
    - On exception, copy architectural map into future map…
  - Checkpointing: keep regular check points of map, restore when needed
    - When do we make a checkpoint? On every instruction? On every branch?
    - What are the trade-offs?
    - We'll revisit this approach later on…

Resolving True Data Dependencies





STALL DISPATCHING

ADVANCE INSTRUCTIONS

"DYNAMIC EXECUTION"

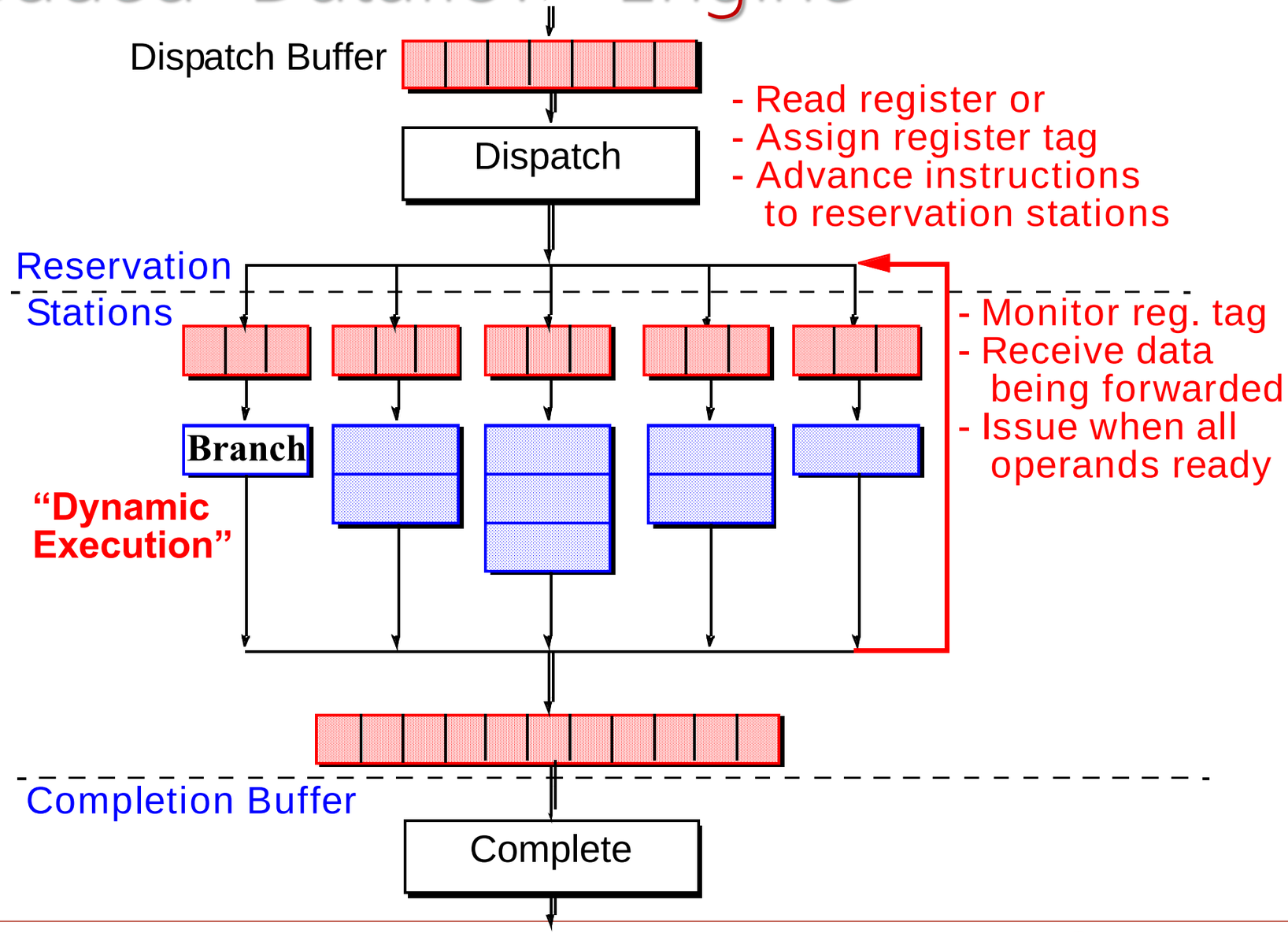
Reservation Station + Complex Forwarding

Out-of-order (OoO) Execution

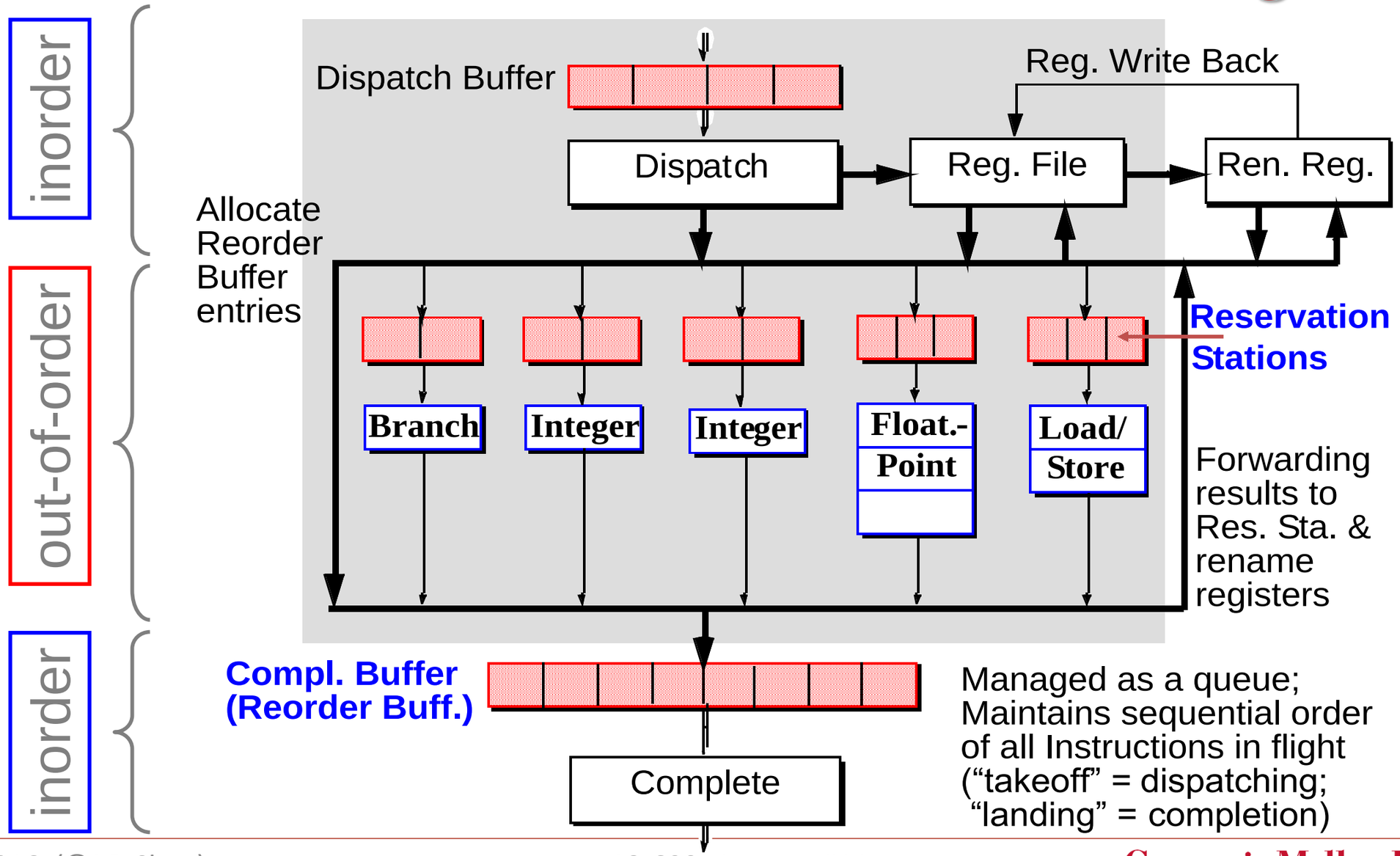
Try to Approach the "Data-Flow Limit"

1) Read register(s), get "IOU" if not ready
2) Advance to reservation station
3) Wait for "IOU" to show up
4) Execute

Embedded "Dataflow" Engine



Elements of Modern Micro-Dataflow Engine



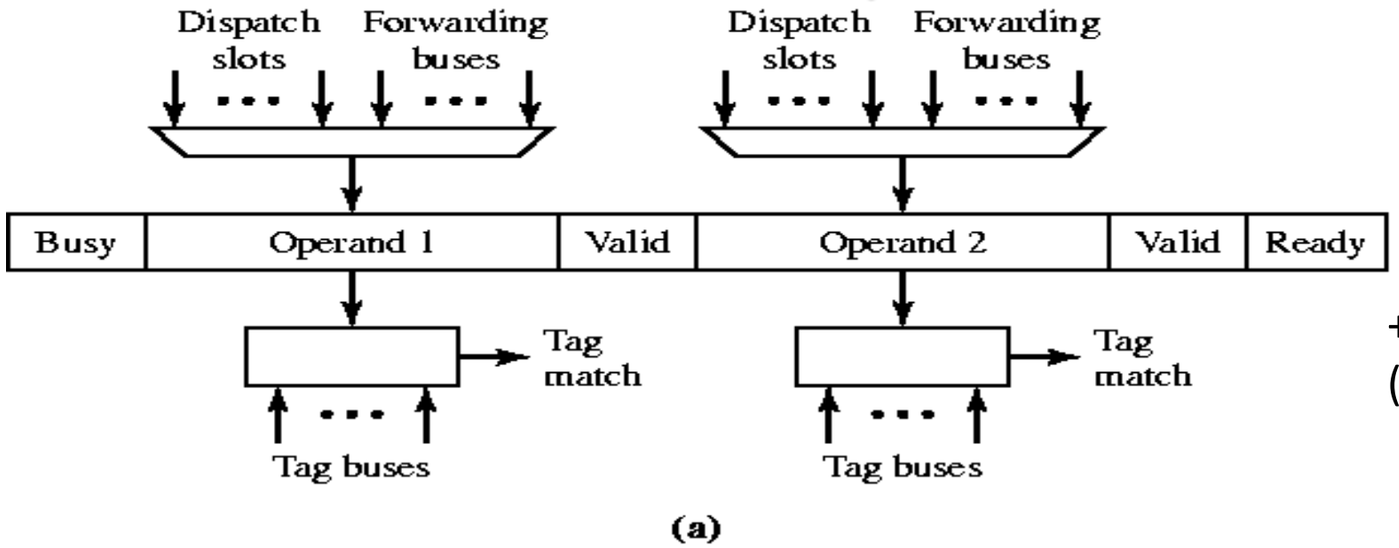
Steps in Dynamic OOO Execution (1)

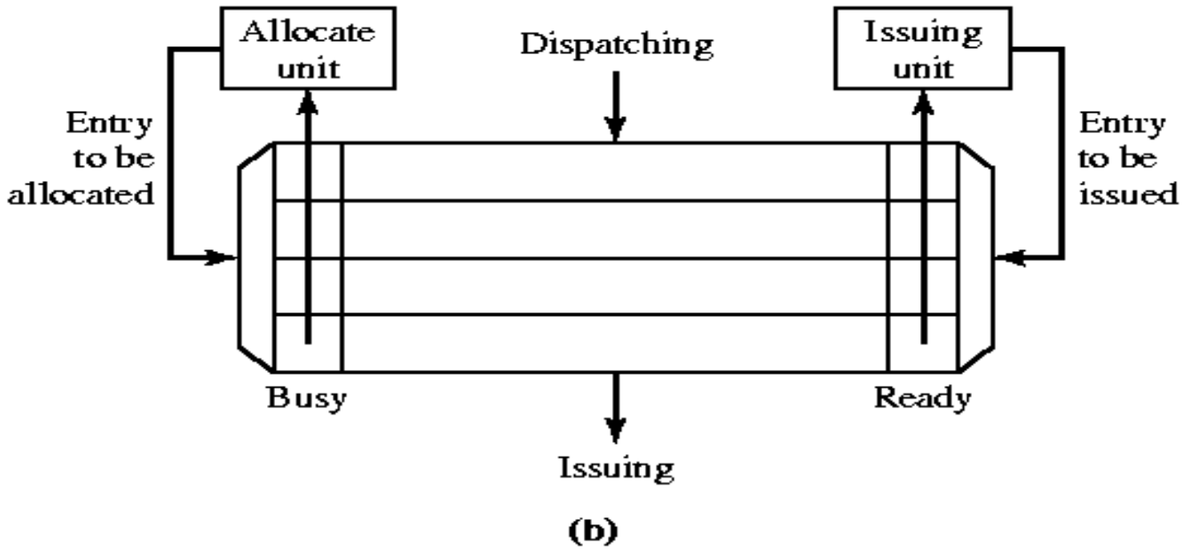
- FETCH instruction (in-order, speculative)
  - I-cache access, predictions, insert in a fetch buffer
- DISPATCH (in-order, speculative)
  - Read operands from Register File (ARF) and/or Rename Register File (RRF)
    - RRF may return a ready value or a Tag for a physical location
  - Allocate new RRF 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





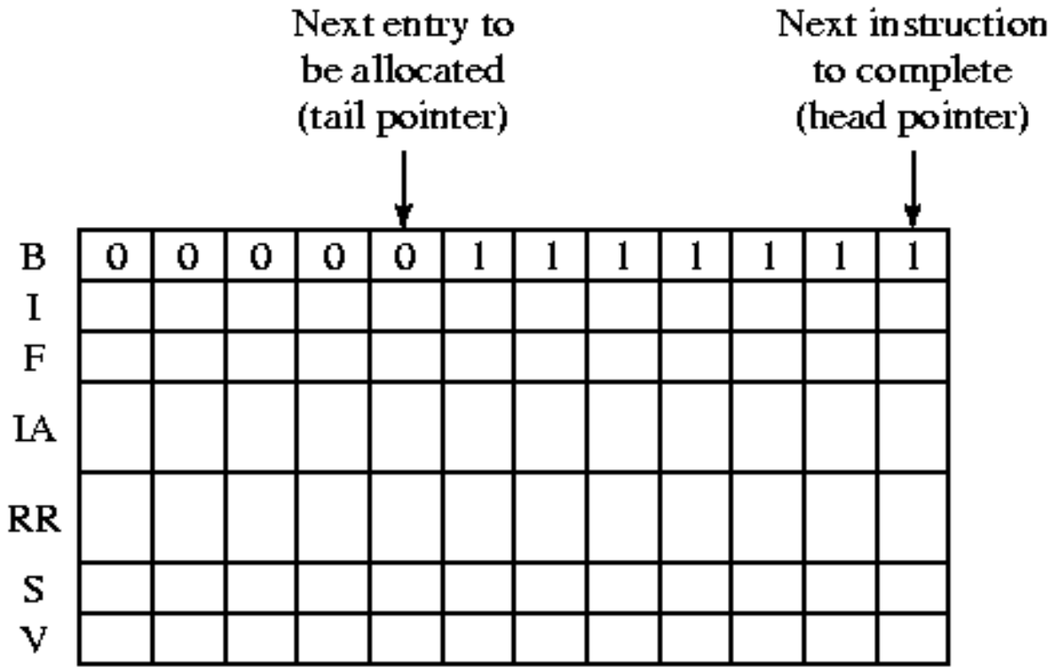
+ info for executing instruction (opcode, ROB entry, RRF entry…)

- 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)



(b)

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

Priority/Select Logic Issues

- Selection is easier if priority depends on instruction location
  - Older instructions are at the bottom of window and receive priority
- This creates an issue of compacting/collapsing
  - As instructions depart, compress remaining towards the bottom
  - Younger instructions will be inserted towards the top (lower priority)
- Compacting the window can be quite complex
  - Its complexity can affect performance (clock frequency)
  - Often implemented in some restricted form
    - E.g. split window into multiple groups, allow compaction of groups
    - Trade-off between window utilization and compaction simplicity

Wake-up Latency

- Assume a result becomes available in cycle i
  - When can you start executing an instruction that is waiting for it?
- Ideal solution: in cycle i+1
  - Back to back executing, just like with 5-stage pipeline
  - Requirement: the following have to work in one cycle
    - Distribute result tag to the window and detect that instruction becomes ready
    - Select instruction for execution and forward its info/operands to FU
  - May stress clock cycle in wide processors
- Alternative: split wake-up and select in separate cycles
  - Simpler hardware, faster clock cycle
  - Lower IPC (dependencies cost one extra cycle)

Result Forwarding

- Common data bus: used to broadcast results of FUs
- Broadcast destinations
  - ARF or RRF or ROB, depending on the renaming scheme
  - Instruction window
    - May need result or tag for the result
- Number of CDBs
  - Best case, 1 per functional unit
  - Can have less, but now we may have structural hazard
- Notes:
  - CDBs can be slow as the routing goes across large chip area
  - Broadcast tag early

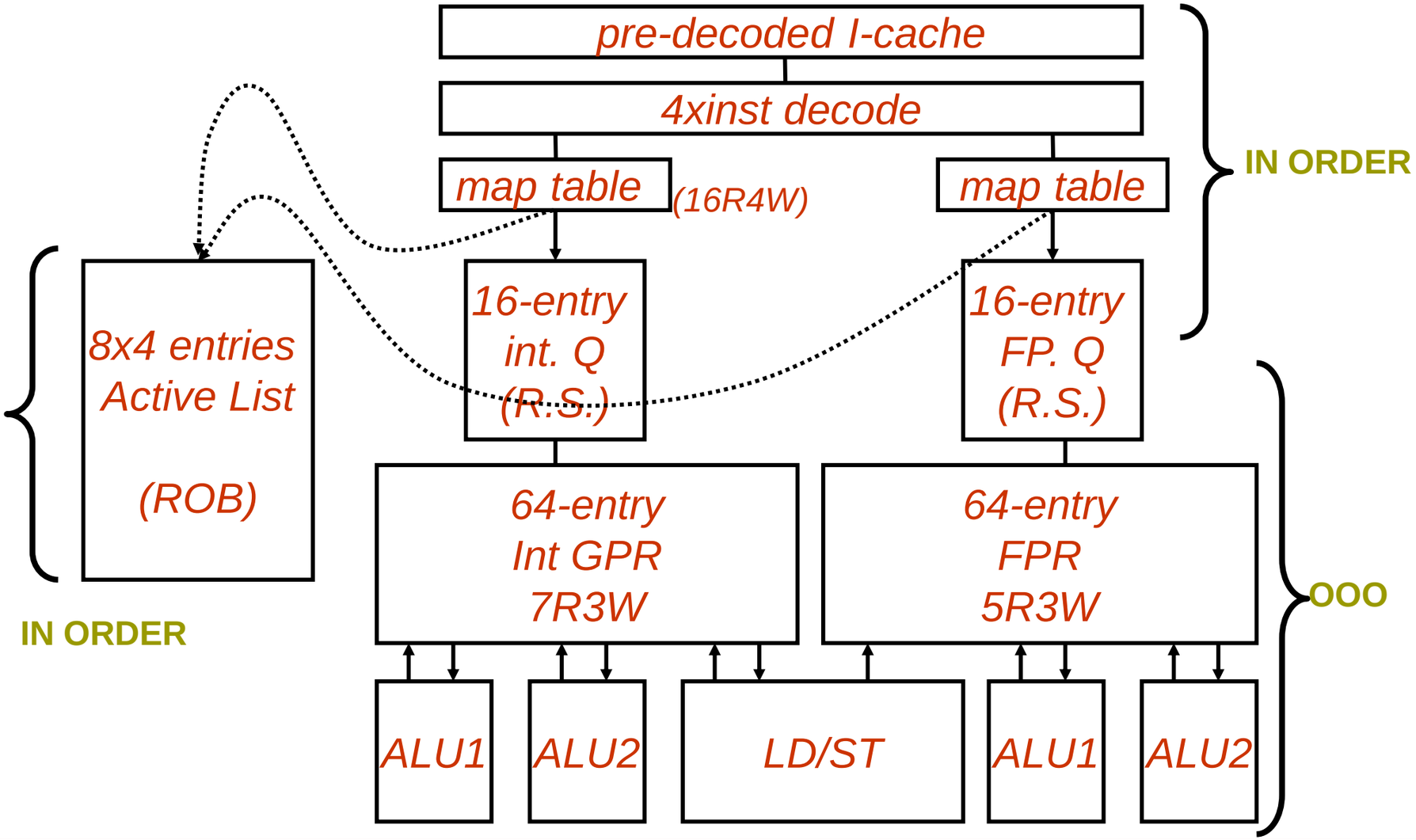
Dynamic Scheduling Implementation Cost

- To support N-way dispatch into IW 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)

Example: MIPS R10000 circa 1996

- 4-way superscalar
  - fetch, decode, dispatch, and complete
- 5 execution pipelines
  - 2 Int, FP Add, FP Mult, Ld/St
- Micro-dataflow instruction scheduling
  - 16+16 instruction window
- Register renaming + memory renaming
  - 64 physical integer registers to hold 33 logical registers + renamed registers
- Speculative execution pass 4 unresolved branches
- Precise Interrupts

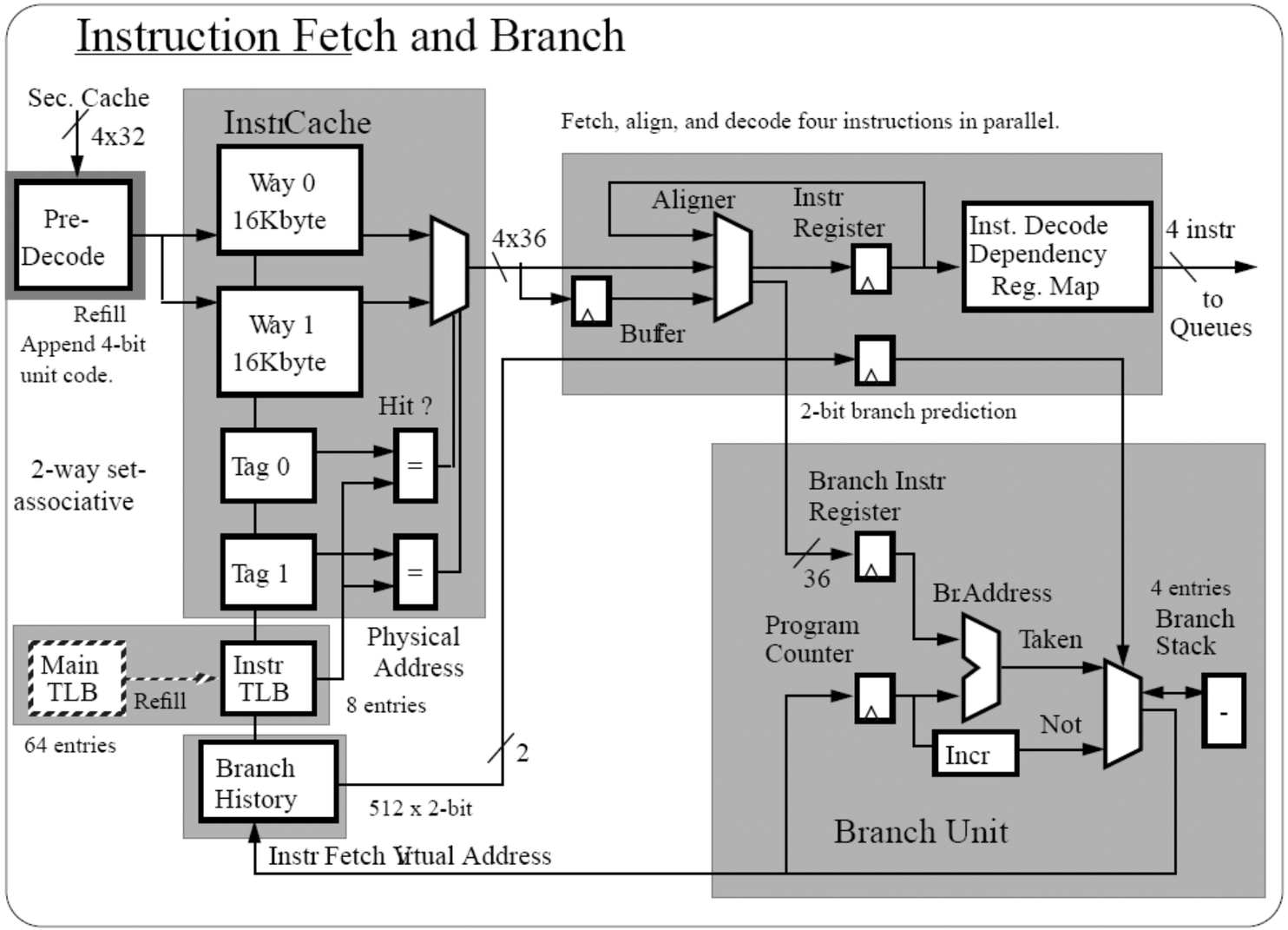
Dynamic Scheduling in the OOO MIPS R10000



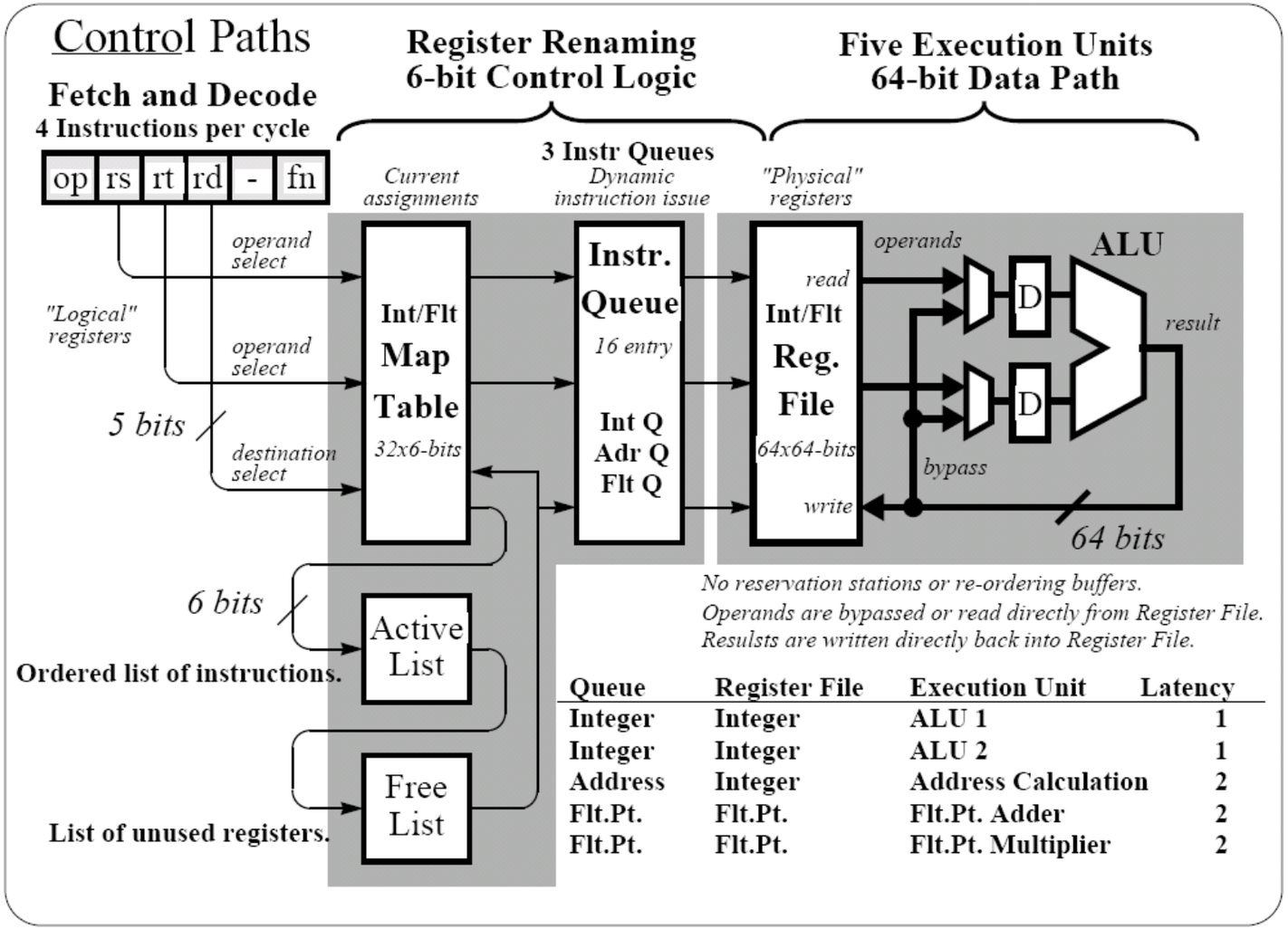
Design Choices

- Register Renaming
  - Map table lookup + dependency check on simultaneous dispatches
  - Unified physical register file
  - 4-deep branch stack to backup the map table on branch predictions
  - Sequential (4-at-a-time) back-tracking to recover from exceptions
- Instruction Queues
  - Separate 16-entry floating point and integer instruction queues
  - Prioritized, dataflow-ordered scheduling
- Reorder Buffer
  - One per outstanding instruction, FIFO ordered
  - Stores PC, logical destination number, old physical destination number
    Why not current physical destination number?

R10000 Instruction Fetch and Branch

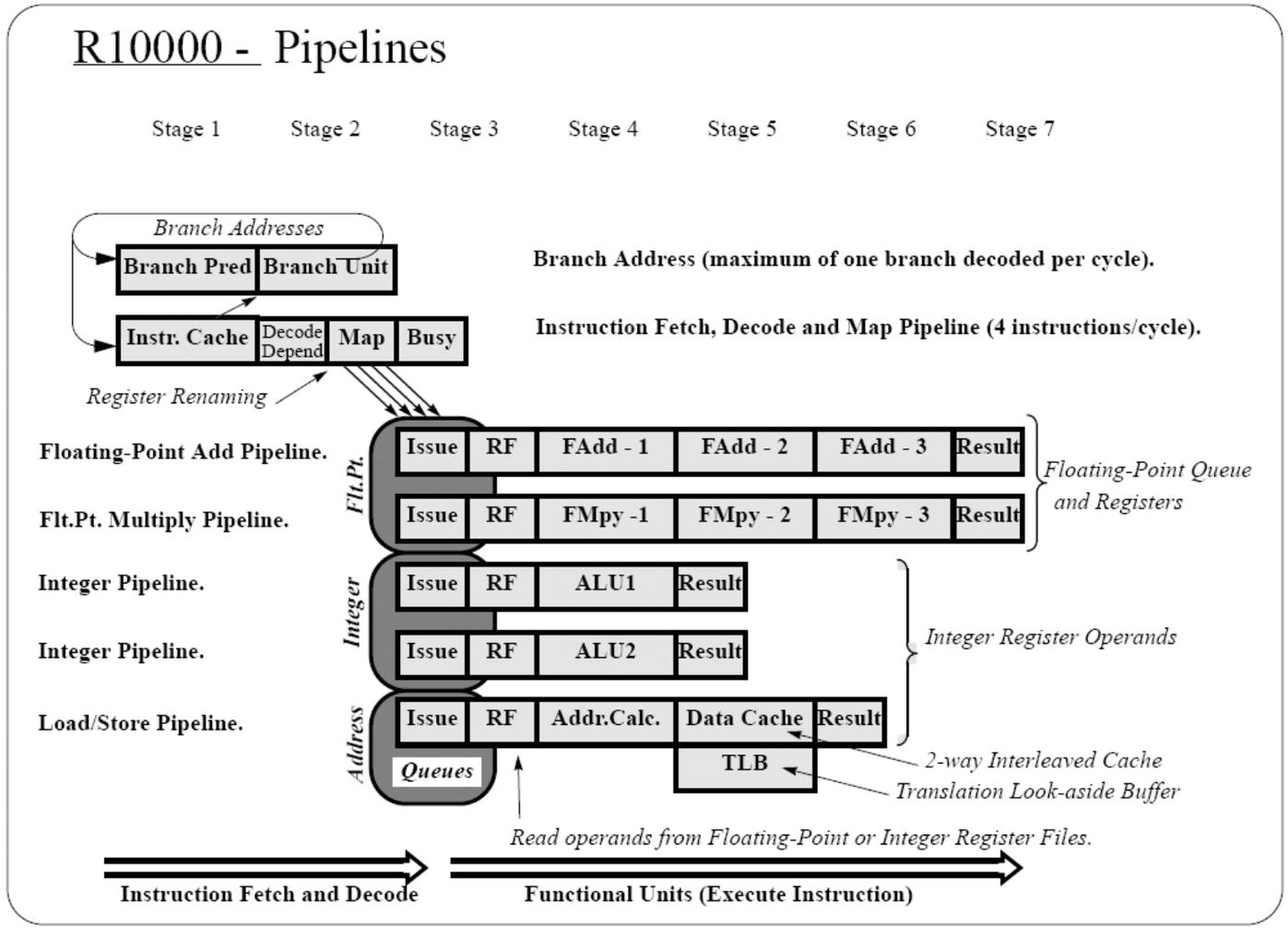


R10000 Register Renaming



Queue	Register File	Execution Unit	Latency
Integer	Integer	ALU 1	1
Integer	Integer	ALU 2	1
Address	Integer	Address Calculation	2
Flt.Pt.	Flt.Pt.	Flt.Pt. Adder	2
Flt.Pt.	Flt.Pt.	Flt.Pt. Multiplier	2

R10000 Pipelines

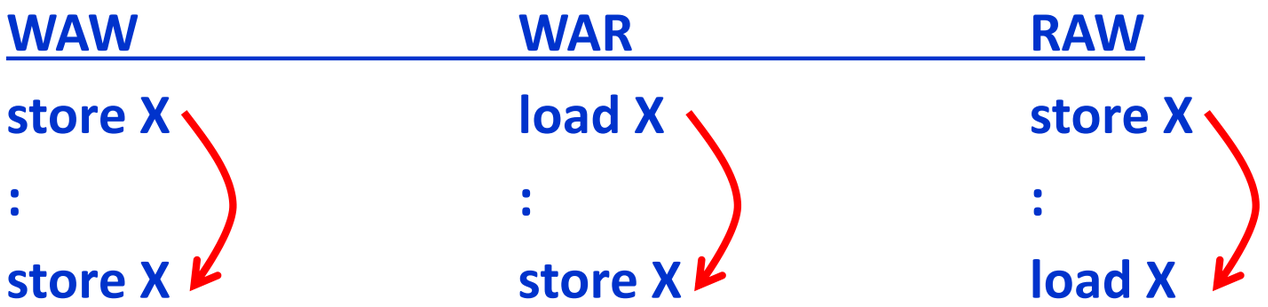


Memory Operations and Data Flow

- 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)
  - 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.

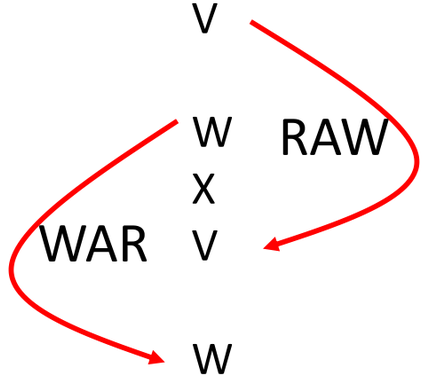


Memory Data Dependency Terminology

- "**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    W
(4)   LOAD    X
(5)   LOAD    V
(6)   ADD
(7)   STORE   W
```



Total Ordering of Loads and Stores

- Keep all loads and stores totally in order with respect to each other.
- However, loads and stores can execute out of order with respect to other types of instructions (while obeying register data dependences).
- Consequently, stores are held for all previous instructions, and loads are held for stores.
  - I.e. stores performed at commit point
  - Sufficient to prevent wrong branch path stores since all prior branches now resolved

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"

Dynamic Reordering of Memory Operations

- Storing to memory irrevocably changes the in-order machine state, therefore a Store instruction is only executed when it is the oldest unfinished instruction

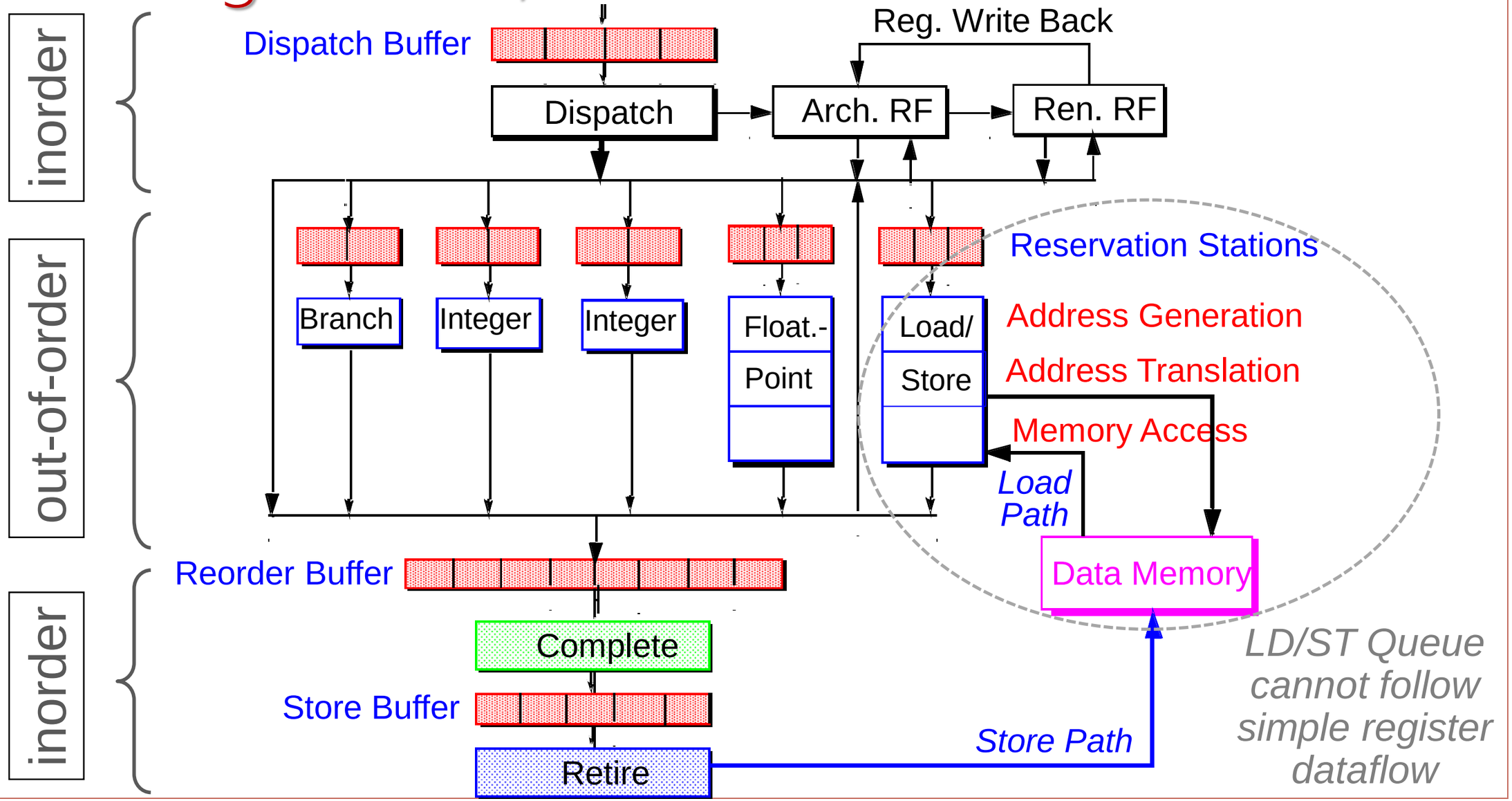
  No memory WAW or WAR

- When to start a load instruction (on a uniprocessor)?
  - No more older store instructions in RS

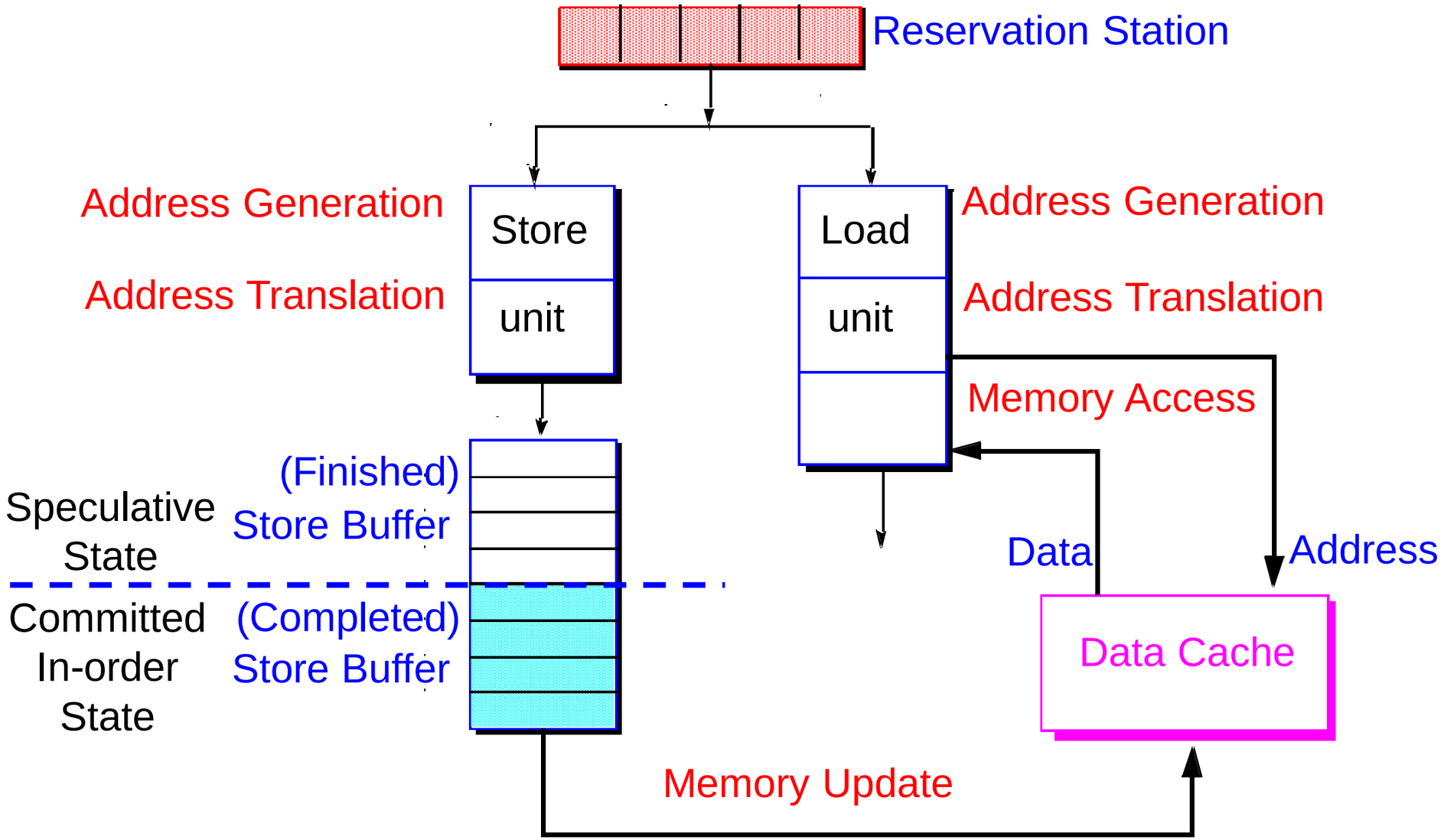
    or
  - Must know the addresses *(VA or PA??)* of all older stores

    or
  - Load speculatively and just *reload* if RAW hazard

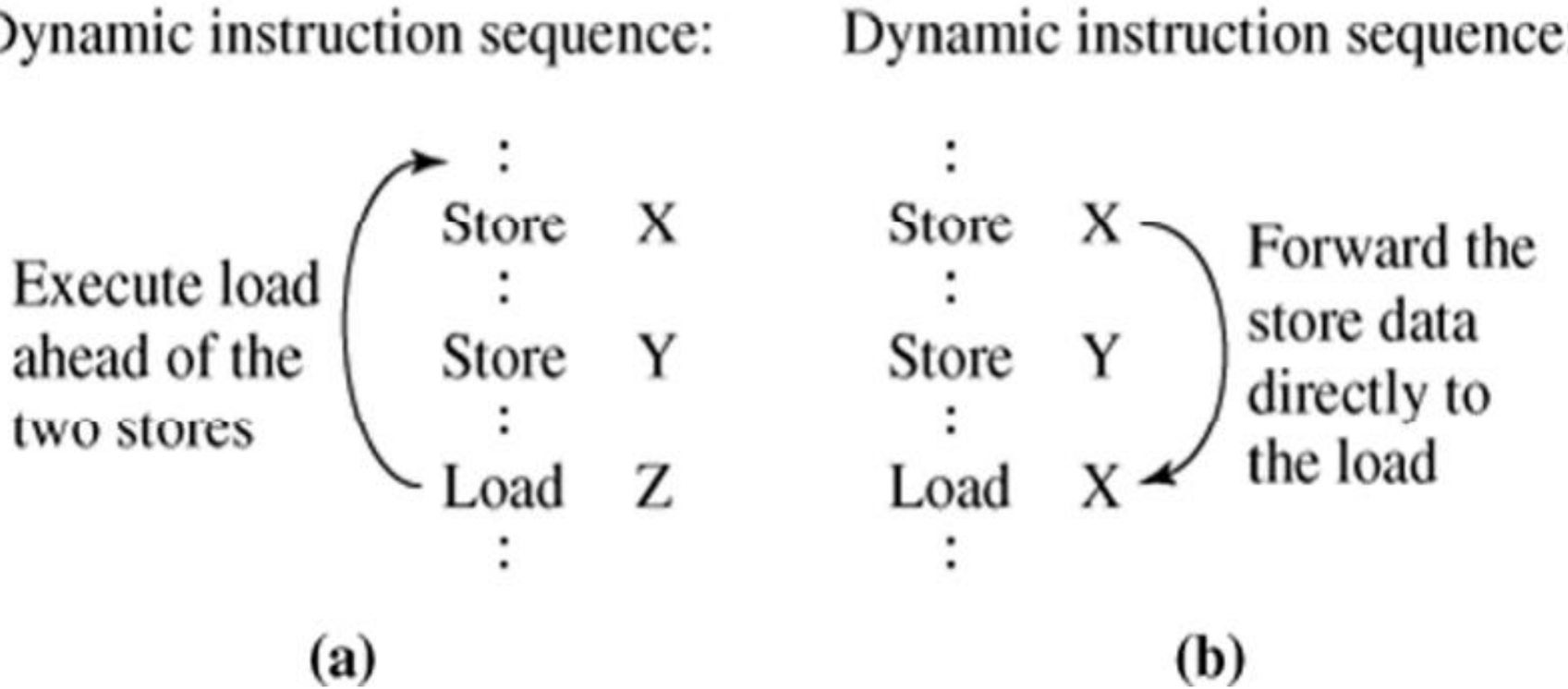
Processing of Load/Store Instructions



Load/Store Units and Store Buffer



Load Bypassing and Load Forwarding: Motivation



Load Bypassing

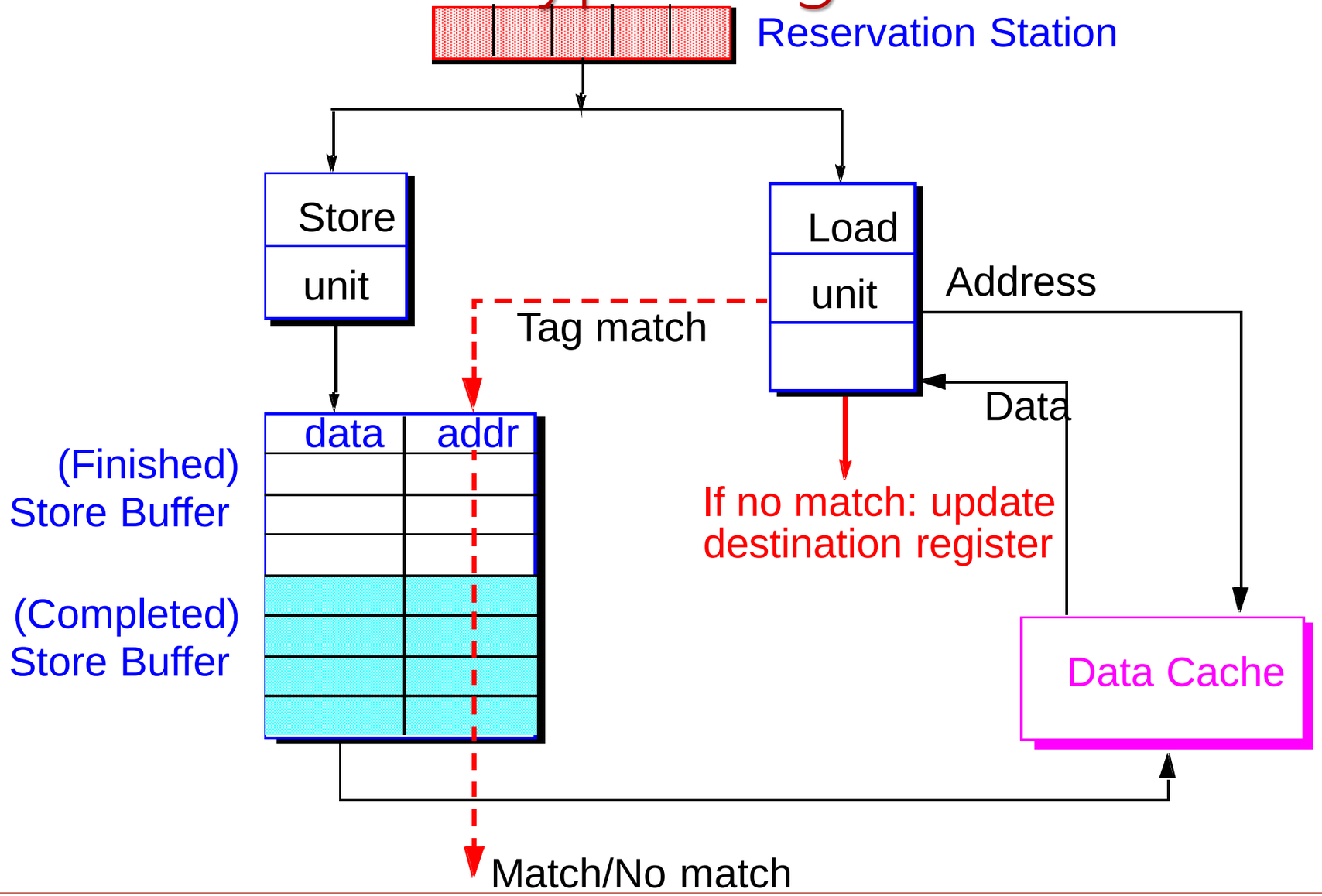
Load Forwarding

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.

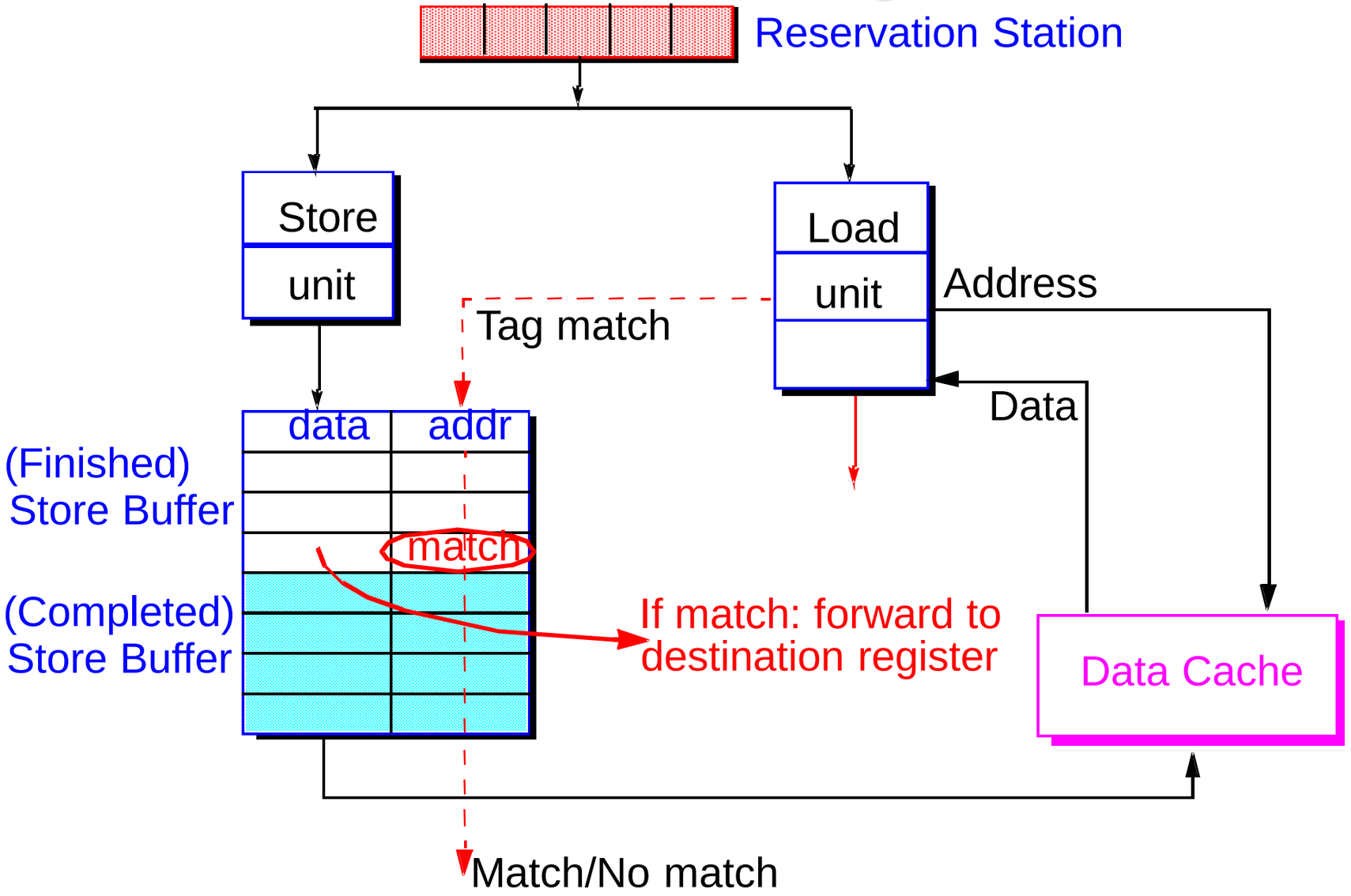
Illustration of Load Bypassing



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

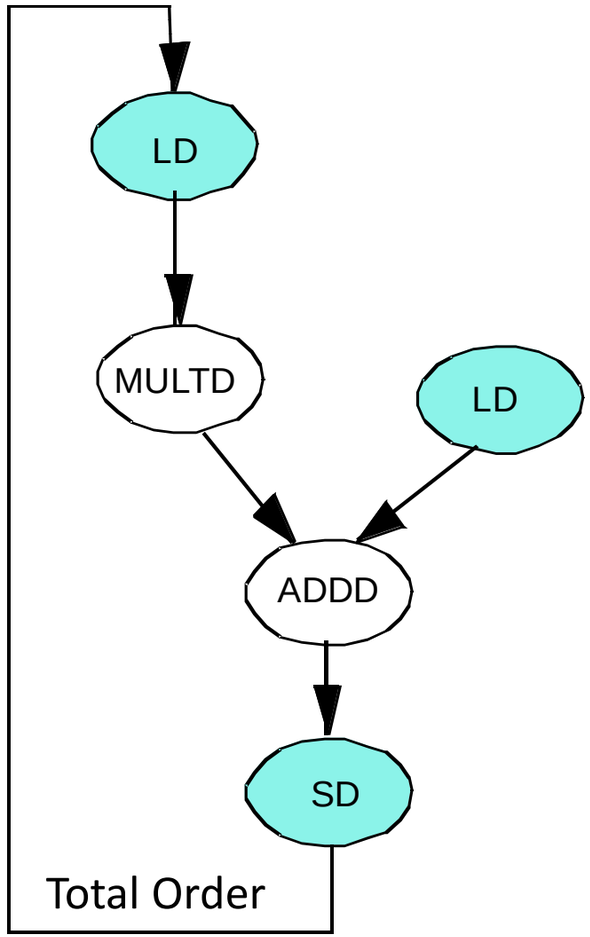


The "DAXPY" Example

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

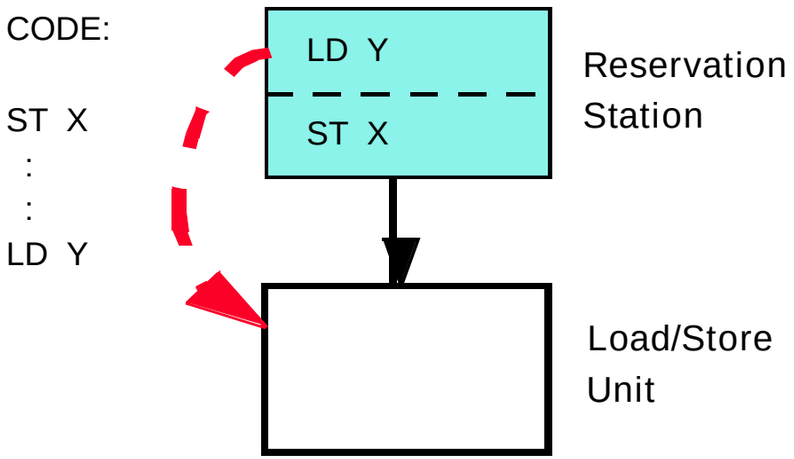
```
        LD      F0, a
        ADDI    R4, Rx, #512        ; last address

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
```

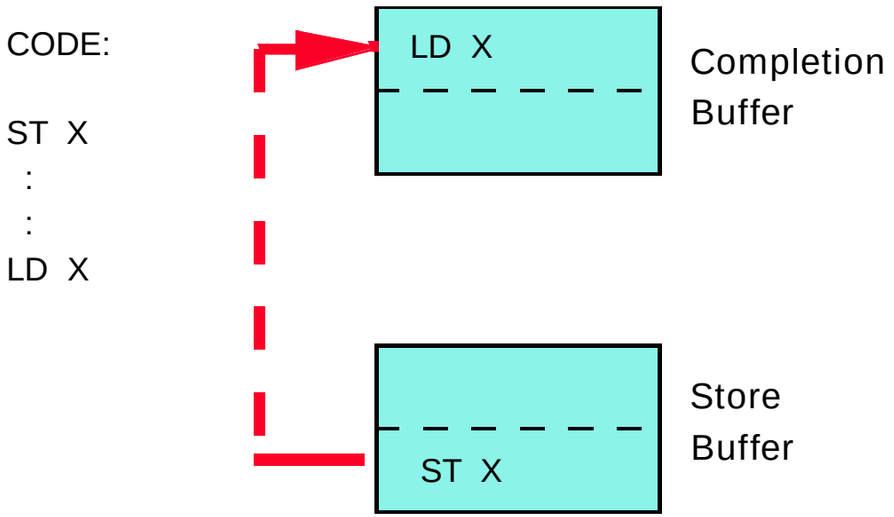


Performance Gains From Weak Ordering

Load Bypassing:



Load Forwarding:



Performance gain:

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

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

Optimizing Load/Store Disambiguation

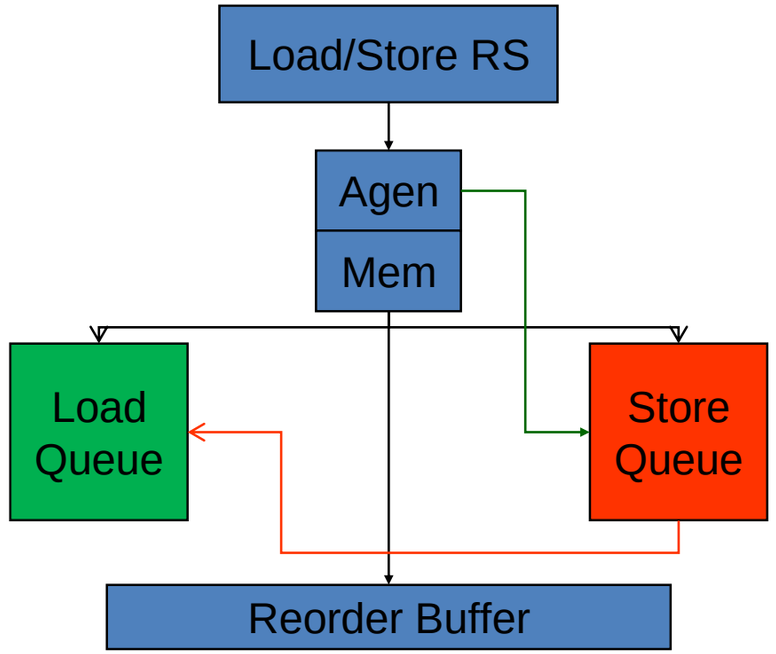
- Non-speculative load/store disambiguation
  1. Loads wait for addresses of all prior stores
  2. Full address comparison
  3. Bypass if no match, forward if match

- (1.) can limit performance:

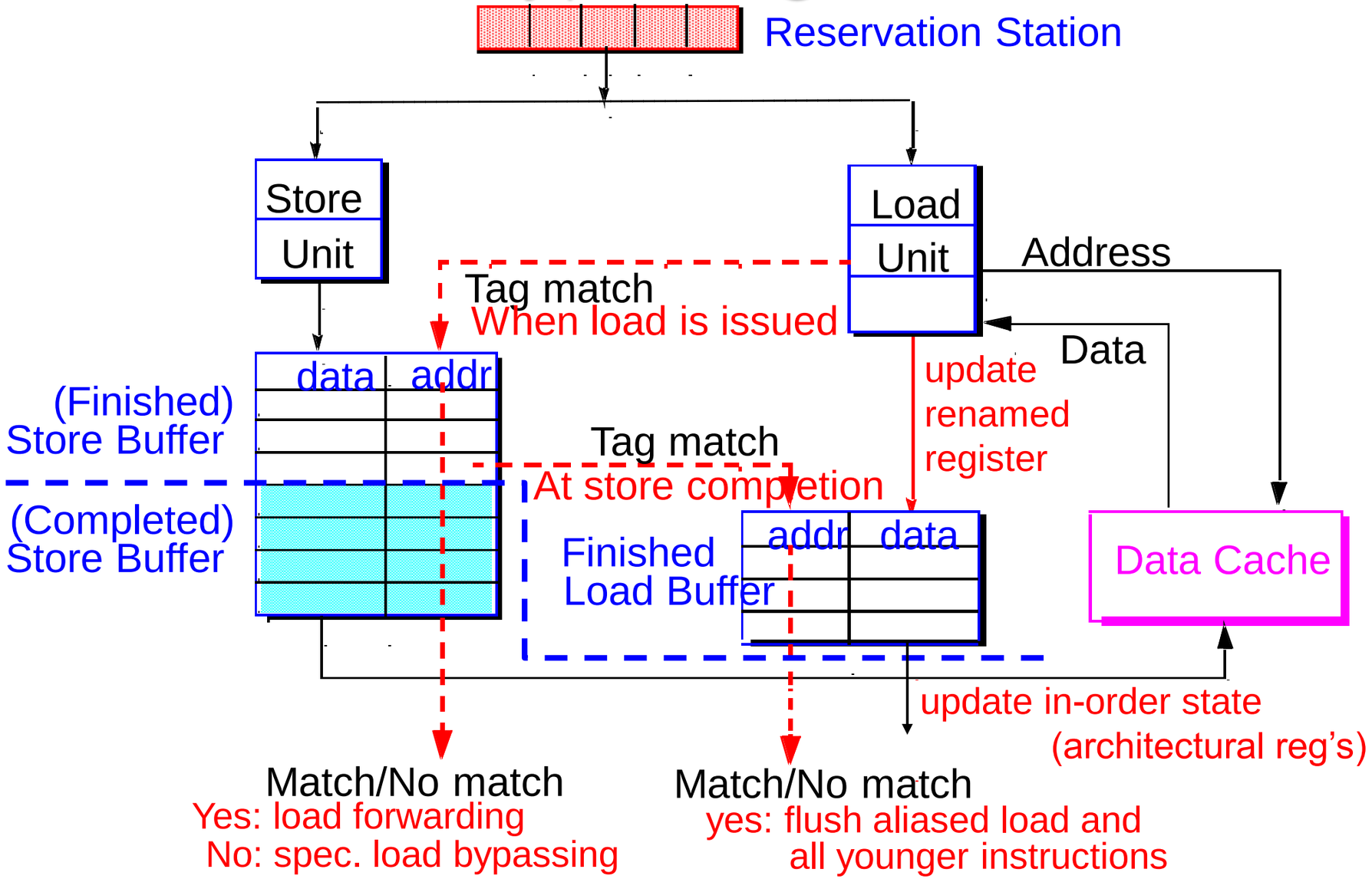
```
load    r5,MEM[r3]  ← cache miss
store   r7, MEM[r5] ← RAW for agen (addr. Gen.), stalled
…
load    r8, MEM[r9] ← independent load stalled
```

Speculative Disambiguation

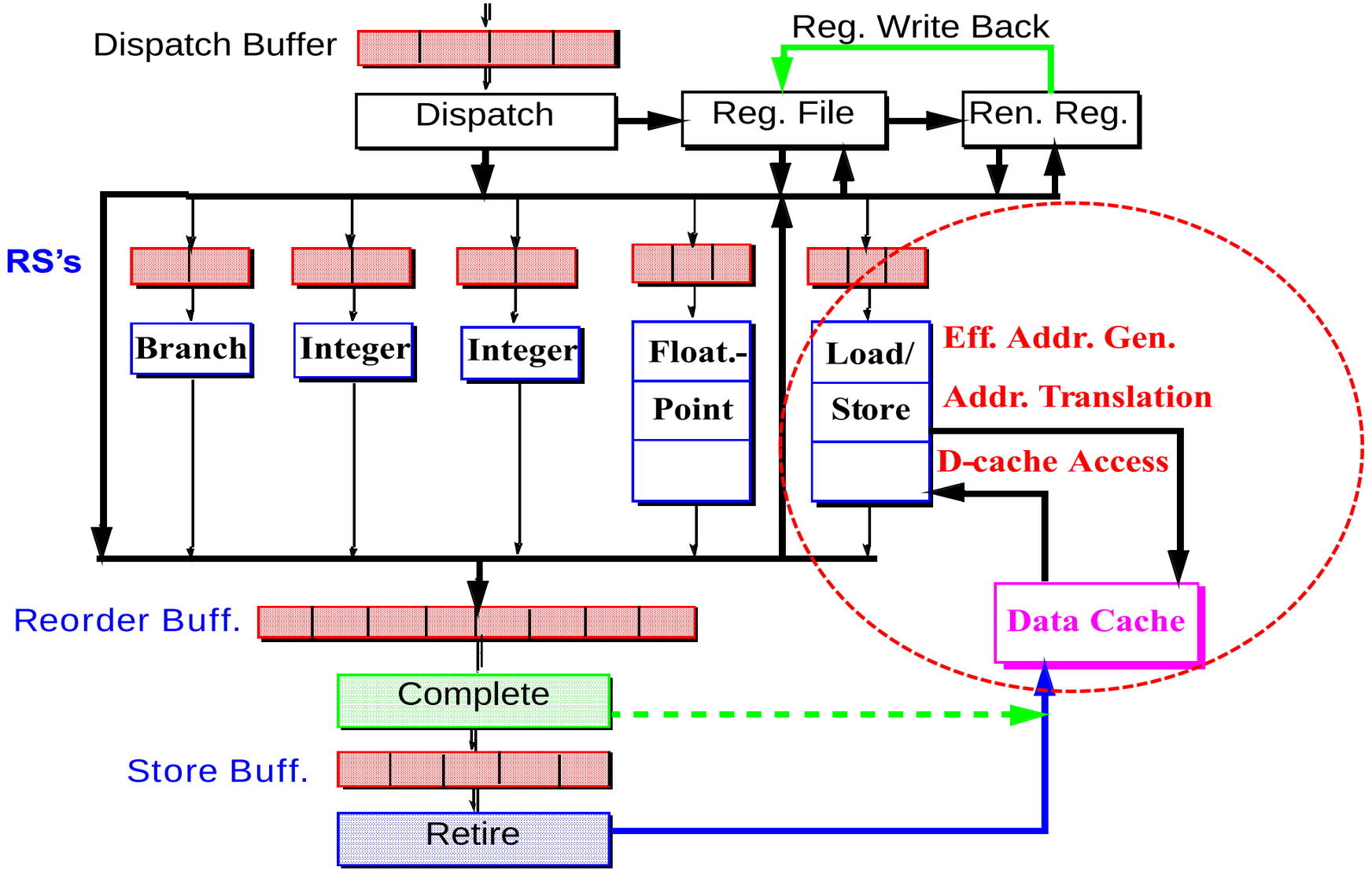
- What if aliases are rare?
  1. Loads don't need to wait for addresses of all prior stores
  2. Full address comparison of stores that are ready
  3. Bypass if no match, forward if match
  4. Check all store addresses when they commit
     - No matching loads – speculation was correct
     - Matching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



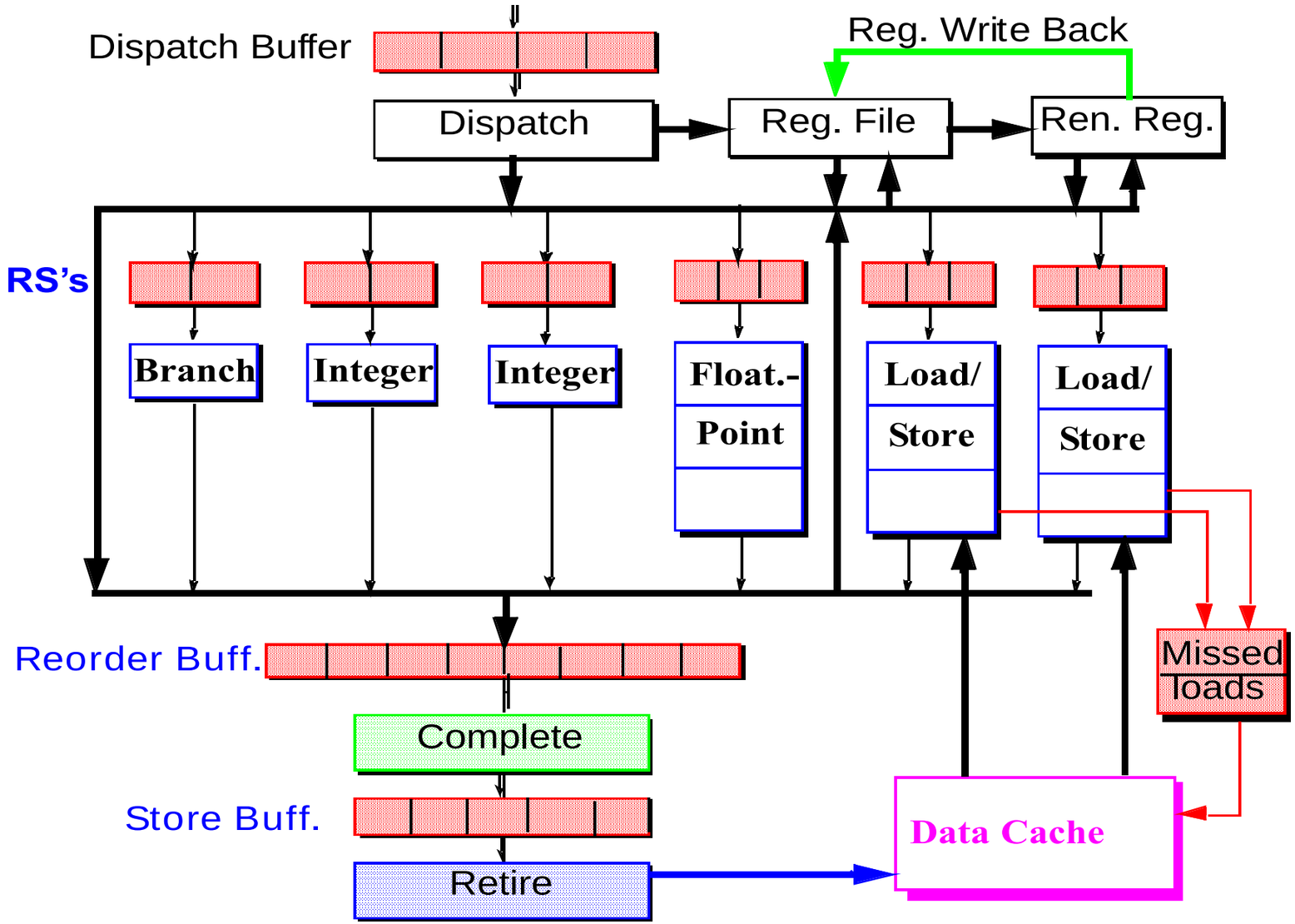
Speculative Load Bypassing



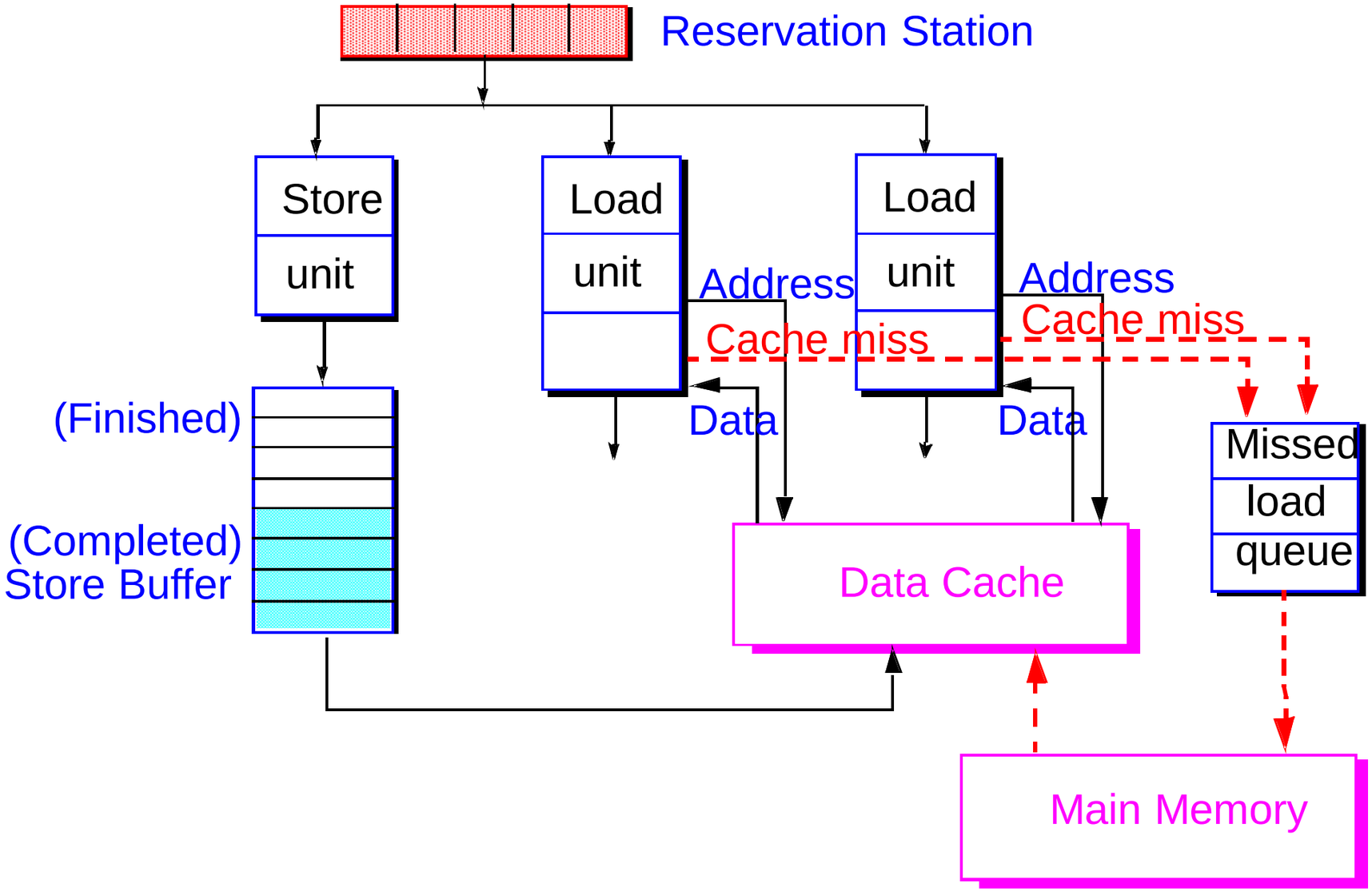
The Memory Bottleneck



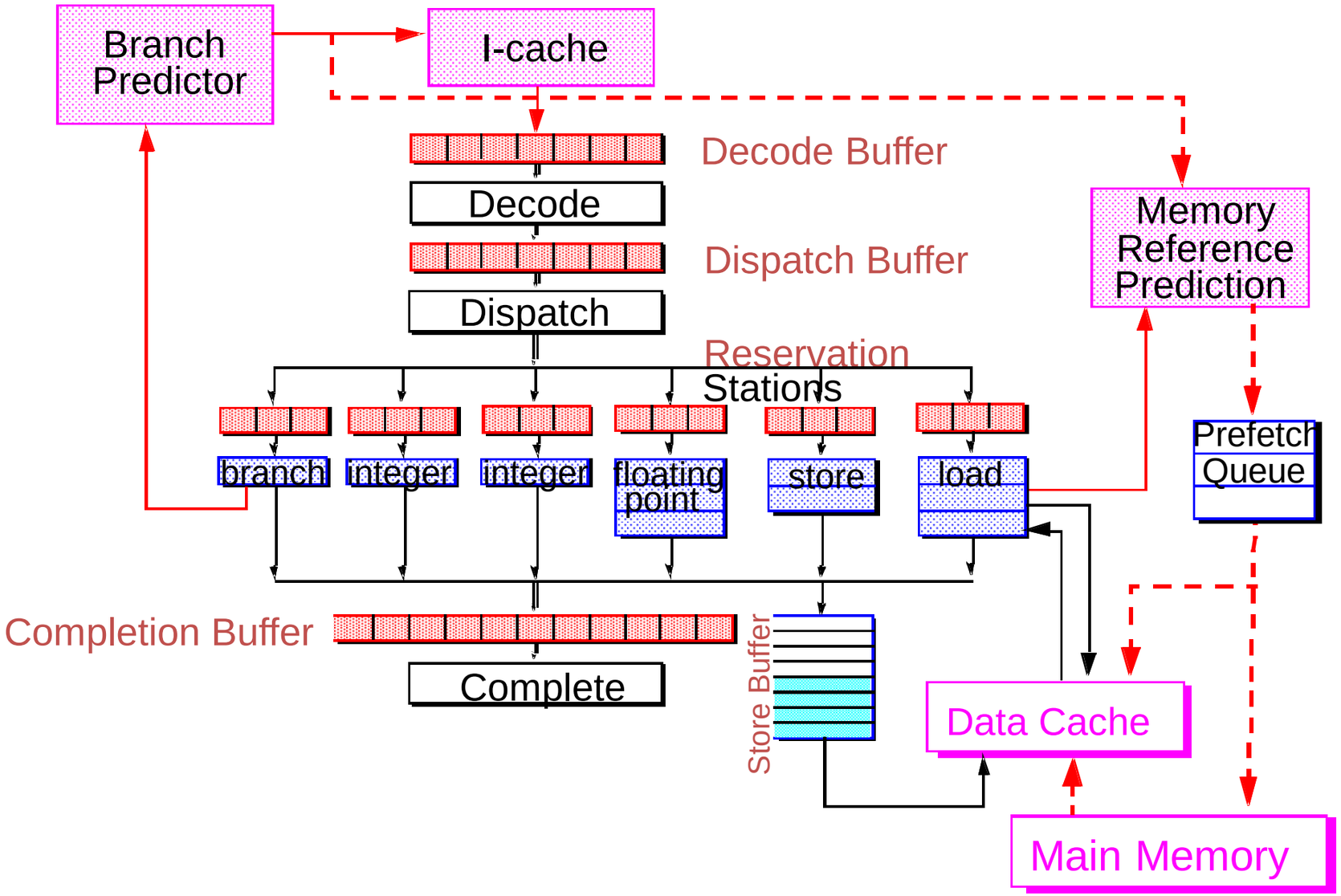
Easing the Memory Bottleneck (missed-load buffer)



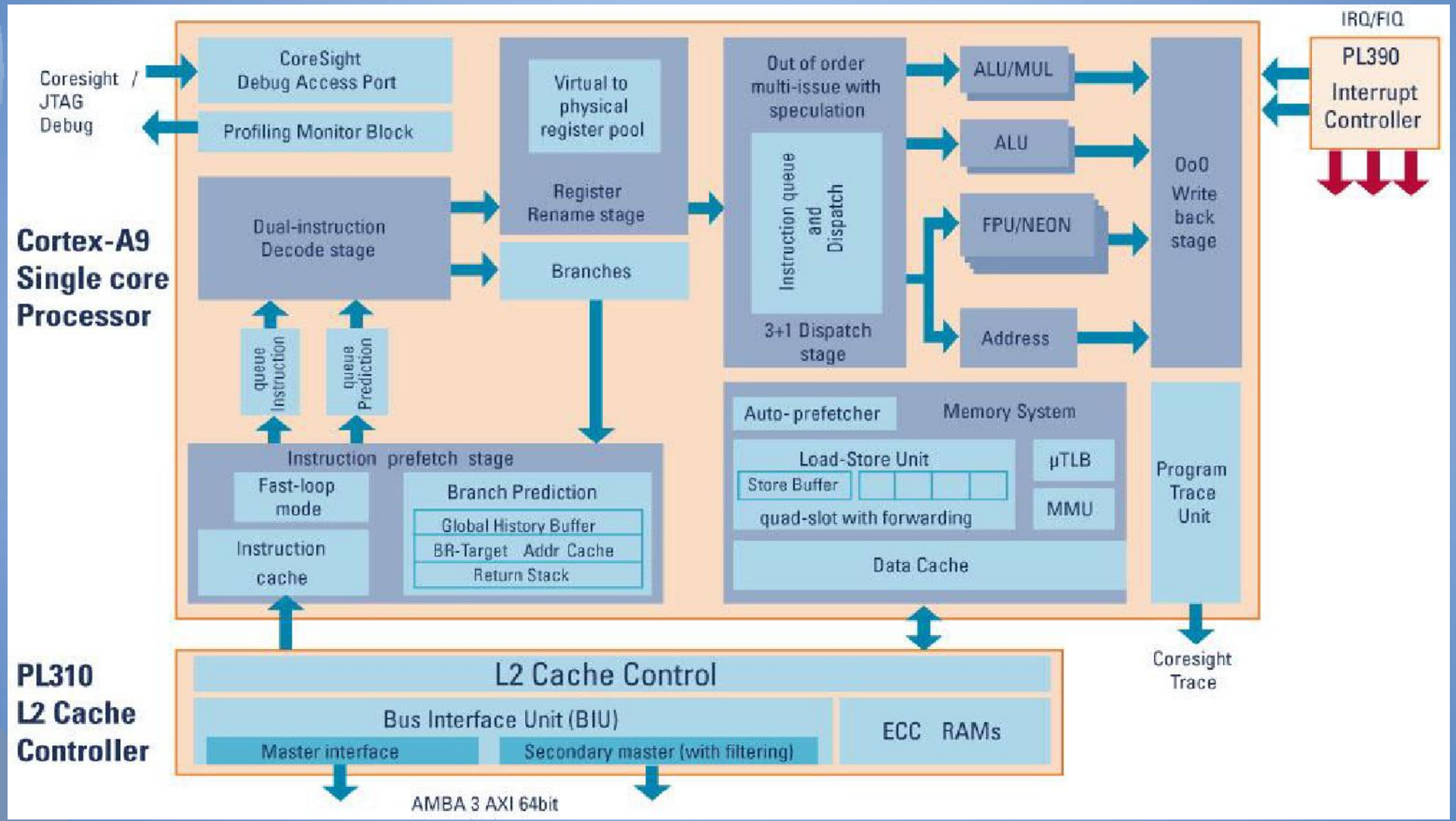
Dual-Ported Non-Blocking Cache



Prefetching Data Cache



Cortex-A9 Single Core Microarchitecture



Cortex-A9 Microarchitecture Structure and the Single Core Interfaces

18-600 Foundations of Computer Systems

Lecture 12: "The Memory Hierarchy"

Zhiyi Yu & John P. Shen

October 10, 2016

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

Carnegie Mellon University