Advanced Superscalars

Yuri Baida
yuri.baida@gmail.com
yuriy.v.baida@intel.com

March 19, 2011

MDSP Project | Intel Lab
Moscow Institute of Physics and Technology

Agenda

- Review
  - Tomasulo algorithm vs. Scoreboard
  - Effectiveness of Tomasulo algorithm
- Advanced superscalars
  - Explicit register renaming
  - Precise exceptions
  - Speculative execution
  - Speculative loads/stores

Review

Tomasulo Algorithm vs. Scoreboard

- Instruction status

Instr	D	S_1	S_2
LD	F6	34+	R2
LD	F2	45+	R3
MULTD	F0	F2	F4
SUBD	F8	F6	F2
DIVD	F10	F0	F6
ADDD	F6	F8	F2

Issue	Write Result
1	4
2	5
3	16
4	8
5	57
6	11

Issue	Write Result
1	4
5	8
6	20
7	12
8	62
13	22

- Why take longer on scoreboard/6600?
  - Structural hazards
  - Lack of forwarding

Effectiveness of Tomasulo Algorithm

- Register renaming
  - Eliminate WAR and WAW dependences without stalling
  - Implicit: space in register file may or may not be used by results!
- Dynamic scheduling
  - Track & resolve true-data dependences (RAW)
  - Scheduling hardware
    - Instruction window, reservation stations, common data bus
- Was first implemented in 1969 in IBM 360/91
  - Did not show up in the subsequent models until mid-nineties
  - Why?
    - 1. Effective on a very small class of programs
    - 2. Did not address the memory latency problem
    - 3. Made exceptions imprecise

Explicit Register Renaming

Explicit Register Renaming

- Tomasulo provides implicit register renaming
  - User registers renamed to reservation station tags
- Explicit register renaming
  - Use physical register file larger than specified by ISA
  - Keep a translation table
    - When register is written, replace table entry with new register
    - Physical register becomes free when not being used by any instructions
  - Pipeline can be exactly like "standard" pipeline
- Advantages
  - Removes all WAR and WAW hazards
  - Allows data to be fetched from a single register file
  - Like Tomasulo, good for allowing full out-of-order retirement
  - Makes speculative execution and precise exceptions easier

Register Renaming Implementation

- Renaming buffer organization (how are registers stored)
  - Unified register file
    - Registers change role architecture to renamed
    - MIPS R10K, Alpha 21264
  - Split register file (ARF + RRF)
    - Holds new values until they are committed to ARF
    - Extra data transfer
    - PA 8500, PPC 620
  - Renaming in ROB
    - Pentium III
- Register mapping (how do I find the register I am looking for)
  - Allocation, de-allocation, tracking
- Number of renaming registers, read/write ports?

Summary

- Reservations stations
  - Renaming to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR and WAW hazards of scoreboard
  - Allows loop unrolling in HW
- Explicit register renaming
  - All registers concentrated in single register file
  - Can utilize bypass network that looks more like 5-stage pipeline
  - Introduces a register-allocation problem
    - Need to handle branch misprediction and precise exceptions differently
    - Ultimately makes things simpler

Precise Exceptions

Classification of Exceptions

- Traps
  - Relevant to the current process
    - Faults, arithmetic traps, and synchronous traps
    - Invoke software on behalf of the currently executing process
- Interrupts
  - Caused by asynchronous, outside events
    - I/O devices requiring service (keyboard, disk, network)
    - Clock interrupts (real time scheduling)
- Machine checks
  - Caused by serious hardware failure
    - Not always restartable
    - Indicate bad things (non-recoverable ECC error, power outage)

Why are Precise Exceptions Desirable?

- Exception is precise if there is a single instruction for which:
  - All instructions before have committed their state
  - No following instructions have modified any state
    - Including the interrupting instruction
- Many types of interrupts/exceptions need to be restartable
  - Easier to figure out what actually happened
    - TLB fault: need to fix translation, then restart load/store
    - IEEE gradual underflow, illegal operation, etc.
  - Restartability doesn't require preciseness
    - However, preciseness makes it a lot easier to restart
- Simplify the task of operating system a lot
  - Less state needs to be saved away if unloading process
  - Quick to restart (making for fast interrupts)

How to Make Exceptions Precise

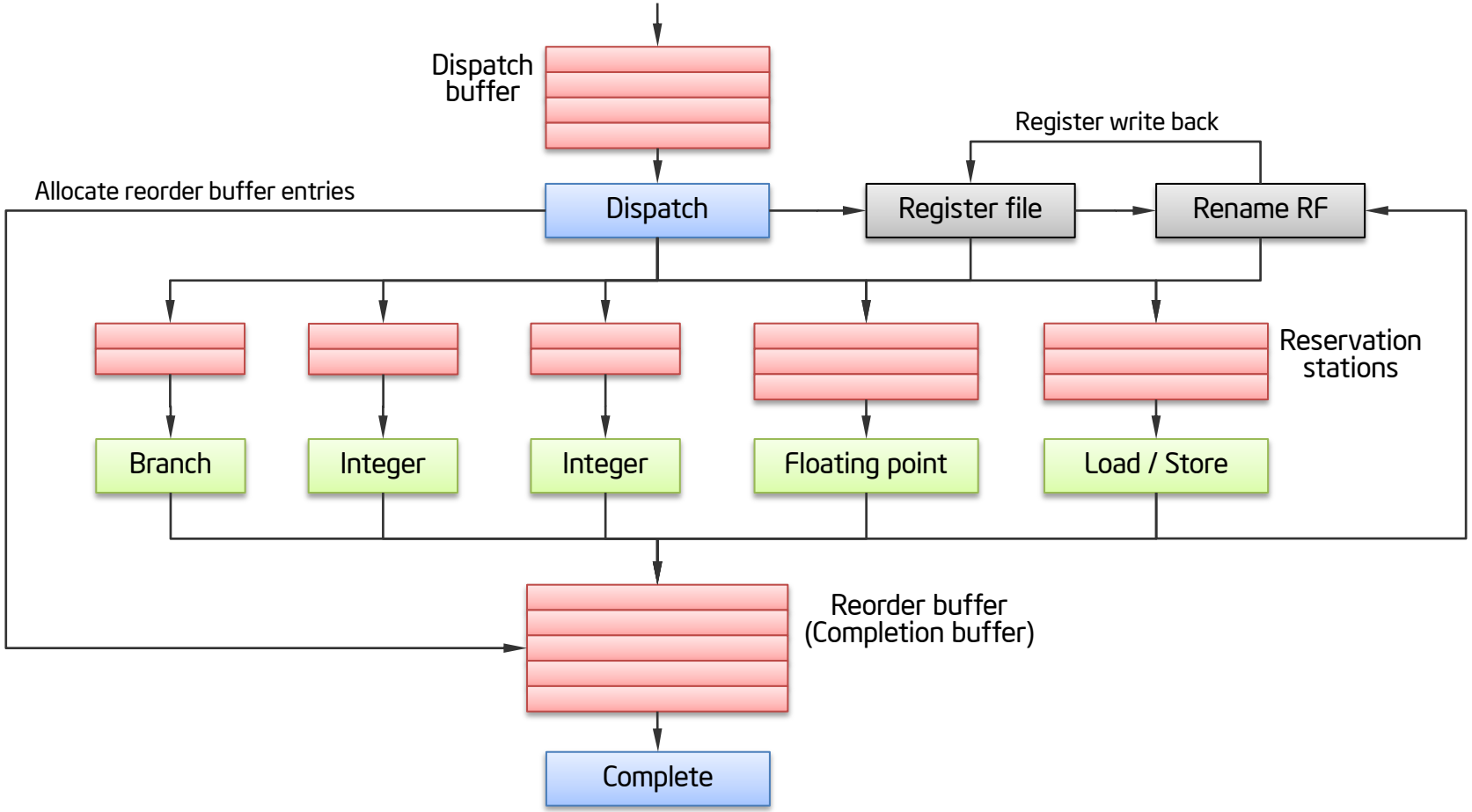
- Both scoreboard and Tomasulo algorithm have
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
- Need way to resynchronize execution with instruction stream
  - Easiest way is with in-order completion (reorder buffer)
  - Other techniques
    - Future file
    - History buffer

Reorder Buffer

- Idea
  - Allow instructions to execute out-of-order
  - Reorder them to complete in-order
- On issue
  - Reserve slot at tail of ROB
  - Record destination register, instruction pointer
- When execution is done
  - Deposit result in ROB slot
  - Mark exception state
- Write back
  - Get head of ROB, check exception, handle
  - Write register value, or commit the store

Dynamic Scheduling in Modern Superscalar

(Explicit register renaming + Reservation stations + ROB)



Steps in Dynamic Scheduling (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 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 Scheduling (2)

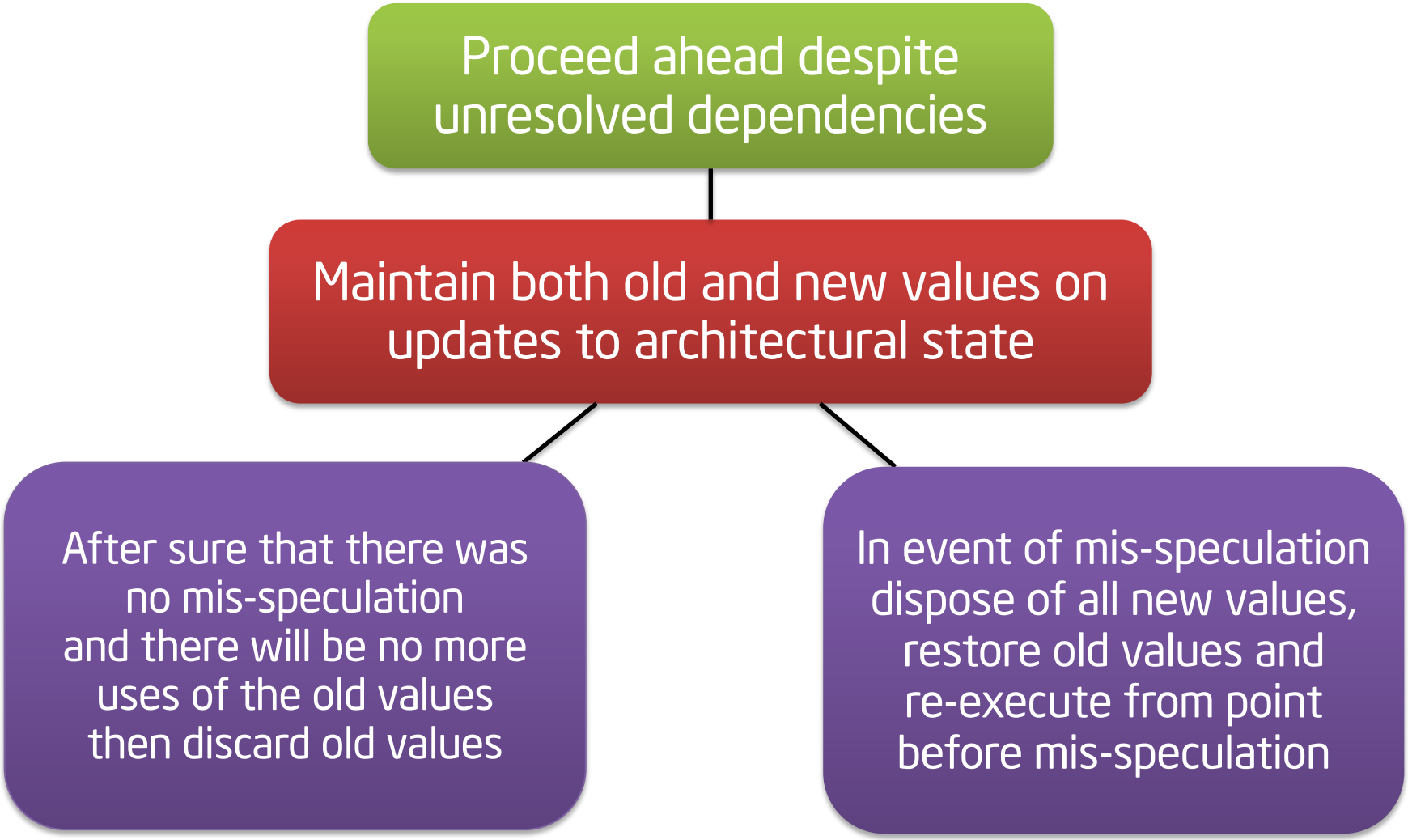
- Issue & execute (out-of-order, speculative)
  - Scheduler entry monitors result bus for rename register tag(s)
    - Find out if source operand becomes ready
  - When all operands ready, issue instruction into Functional Unit (FU) and deallocate scheduler entry (wake-up & select)
    - Subject to structural hazards & priorities
  - When execution finishes, broadcast result to waiting scheduler entries and RRF entry
- Retire/commit (in-order, non-speculative)
  - When ready to commit result into "in-order" state (head of the ROB)
    - Update architectural register from RRF entry, deallocate RRF entry,
    - If it is a store instruction, advance it to Store Buffer
    - Deallocate ROB entry
    - Update predictors based on instruction result

Speculative Execution

Prediction is Essential for Good Performance

- We already discussed predicting branches
- However, architects are now predicting everything
  - Data dependencies
  - Actual data
  - Results of groups of instructions
- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies
    - Artifacts of way that humans/compilers think about problems

Speculative Execution Recipe



We can Use Two Value Management Strategies

- Greedy update
  - Update value in place
  - Maintain a log of old values to use for recovery
    - History file
    - Future file
- Lazy update
  - Buffer new value leaving old value in place
    - Old value can be used after new value is generated
    - Simplified recovery
  - Replace old value only at commit time
    - Reorder buffer

Reorder Buffer + Speculation

- Idea
  - Issue branch into ROB, mark with prediction
    - Branch must resolve before leaving ROB
  - Fetch and issue predicted instructions speculatively
  - Resolve correct
    - Commit following instructions
  - Resolve incorrect
    - Mark following instructions in ROB as invalid
    - Let them clear
- Forwarding?
  - Forward uncommitted results to later uncommitted instructions
  - Match source registers against all destination registers in ROB
    - Forward last (once available)

Speculative Loads/Stores

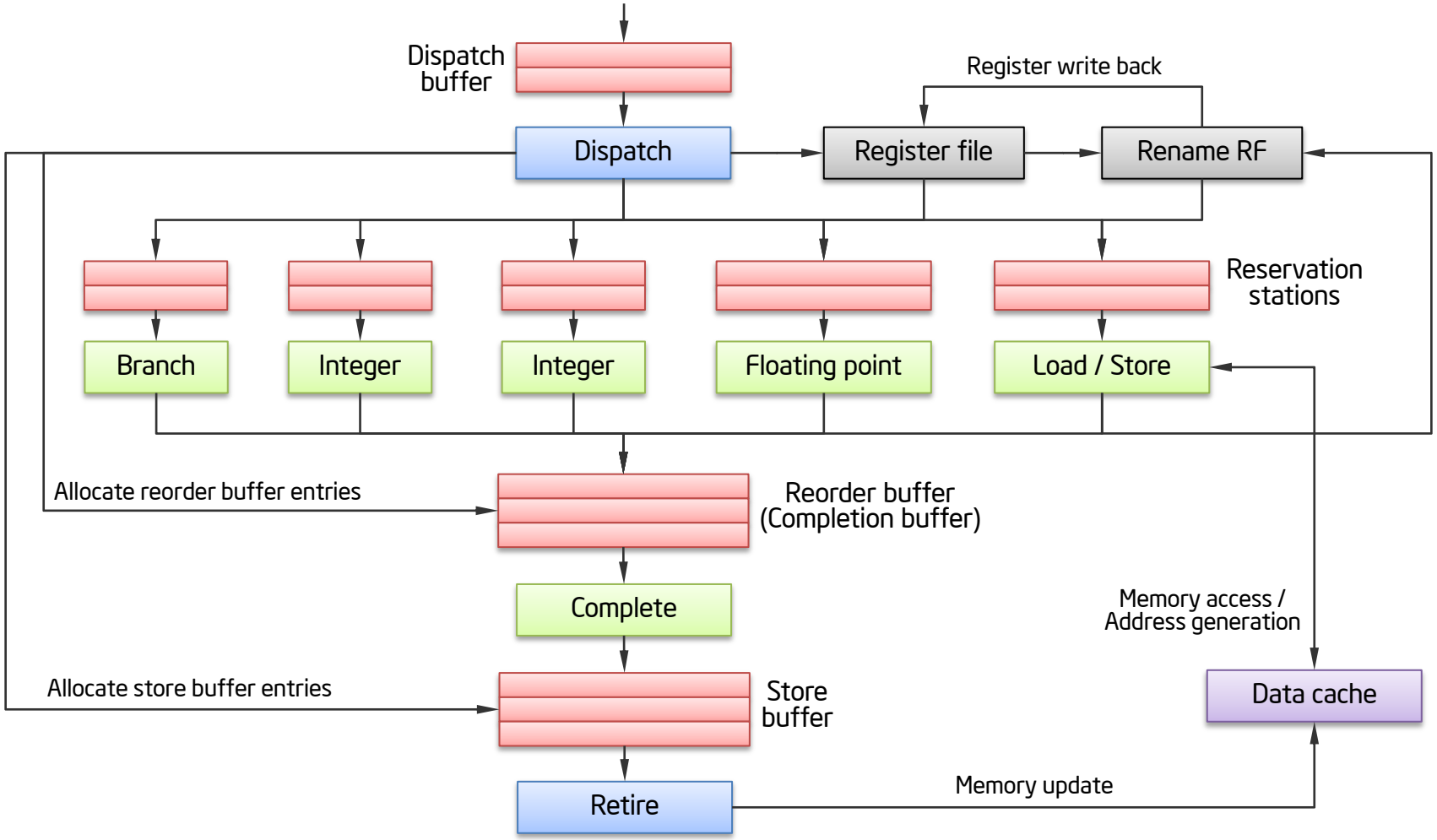
Memory Dependencies are Harder to Handle

- So far, we used register numbers to determine dependencies
- What about load/stores?
  - Memory address are much wider than register names
  - Memory dependencies are not static
  - Memory instructions take longer to execute than other instructions
- Terminology
  - Memory aliasing: two memory references the same memory location
  - Memory disambiguation: determining whether two memory references will alias or not
- Steps in load/store processing
  - Generate address (not fully encoded by instruction)
  - Translate address (virtual ⇒ physical)
  - Execute access (actual load/store)

Problem

- Stores should not permanently change the architectural memory state until it is committed
  - Just like register updates
- Data update policy: greedy or lazy?
  - Lazy: keep queue of stores, in program order
    - Watch for position of new loads relative to existing stores
  - Typically, this is a different buffer than ROB!
    - Store buffer
    - Could be ROB (has right properties), but too expensive
- Handling of store-to-load data hazards
  - Stall?
  - Bypass?
  - Speculate?

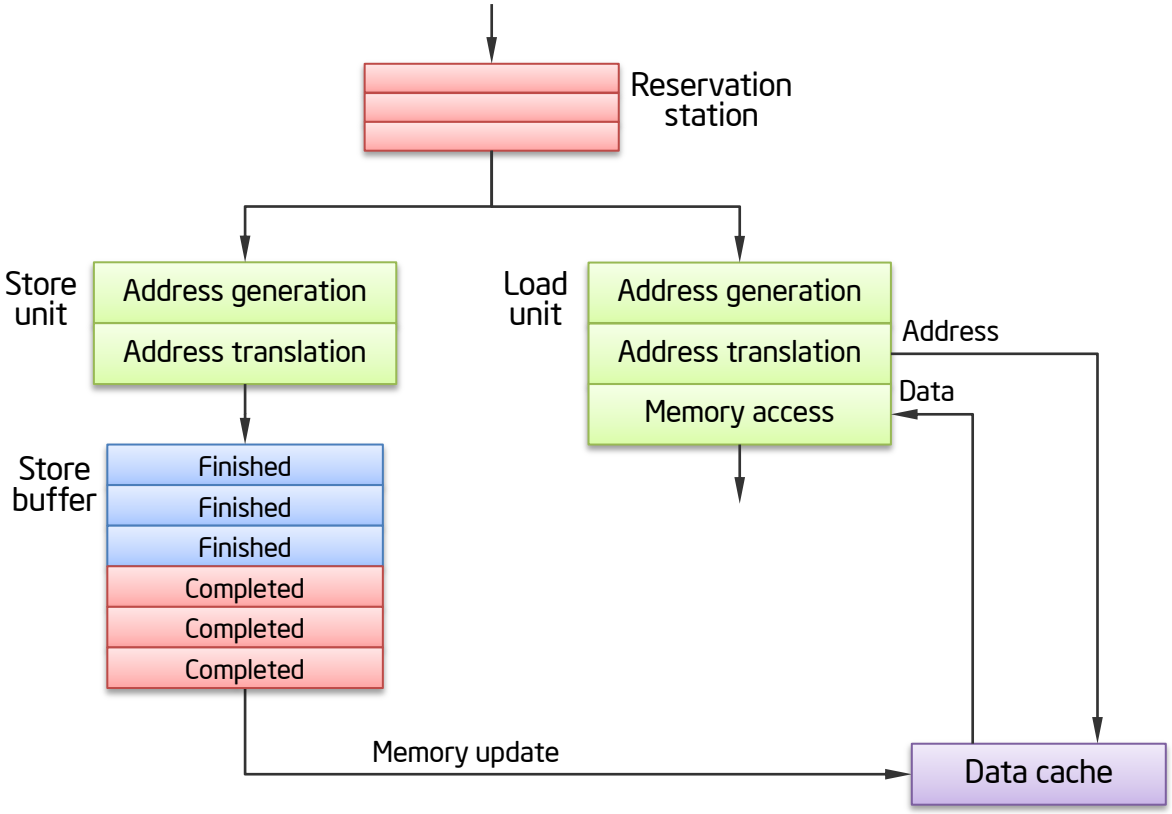
Load/Stores Cannot Follow Register Dataflow



Total Ordering of Loads/Stores
(The Simple Solution)

- Keep all loads and stores totally in order
  - Allocate store buffer entry at dispatch (in-order)
  - When register value available, issue & calculate address ("finished")
  - When all previous instructions retire, store considered "completed"
    - Store buffer split into "finished" and "completed" part though pointers
    - Completed stores go to memory in order
- Loads and stores can execute out-of-order to other instructions while obeying register data-dependence
  - Loads remember the store buffer entry of the last store before them
  - A load can issues when both
    - Address register value is available
    - All older stores are considered "completed"

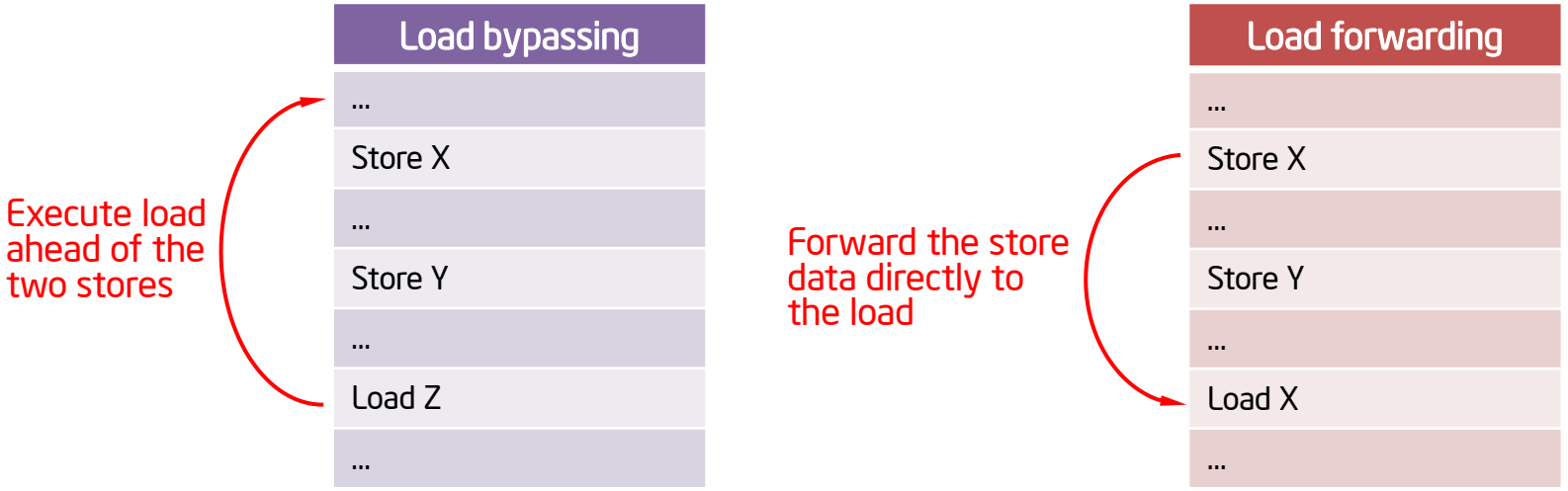
Total Ordering of Loads/Stores
(The Simple Solution)



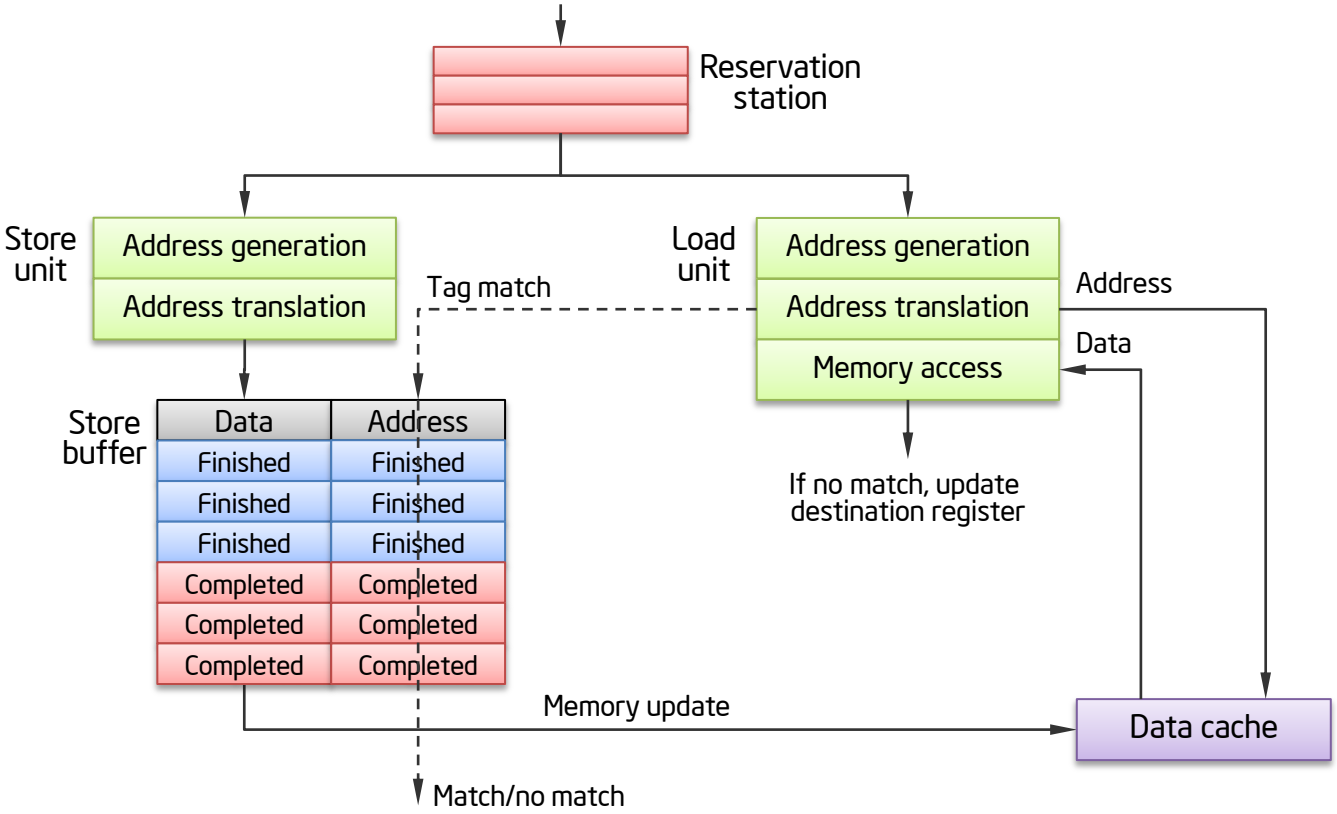
- Performance implications of this model?
  - Consider a simple vector add example: for (i=…) a[i]=b[i]+c[i];

Dynamic Reordering of Load/Stores
(The Better Solution)

- We cannot execute early or reorder store operations
  - Since they must commit in order
  - Therefore, we cannot have WAW or WAR dependencies
- Allow out-of-order execution of loads
  - Can execute load before store, if addresses known and not alias (RAW)
    - Each load address compared with all previous uncommitted stores

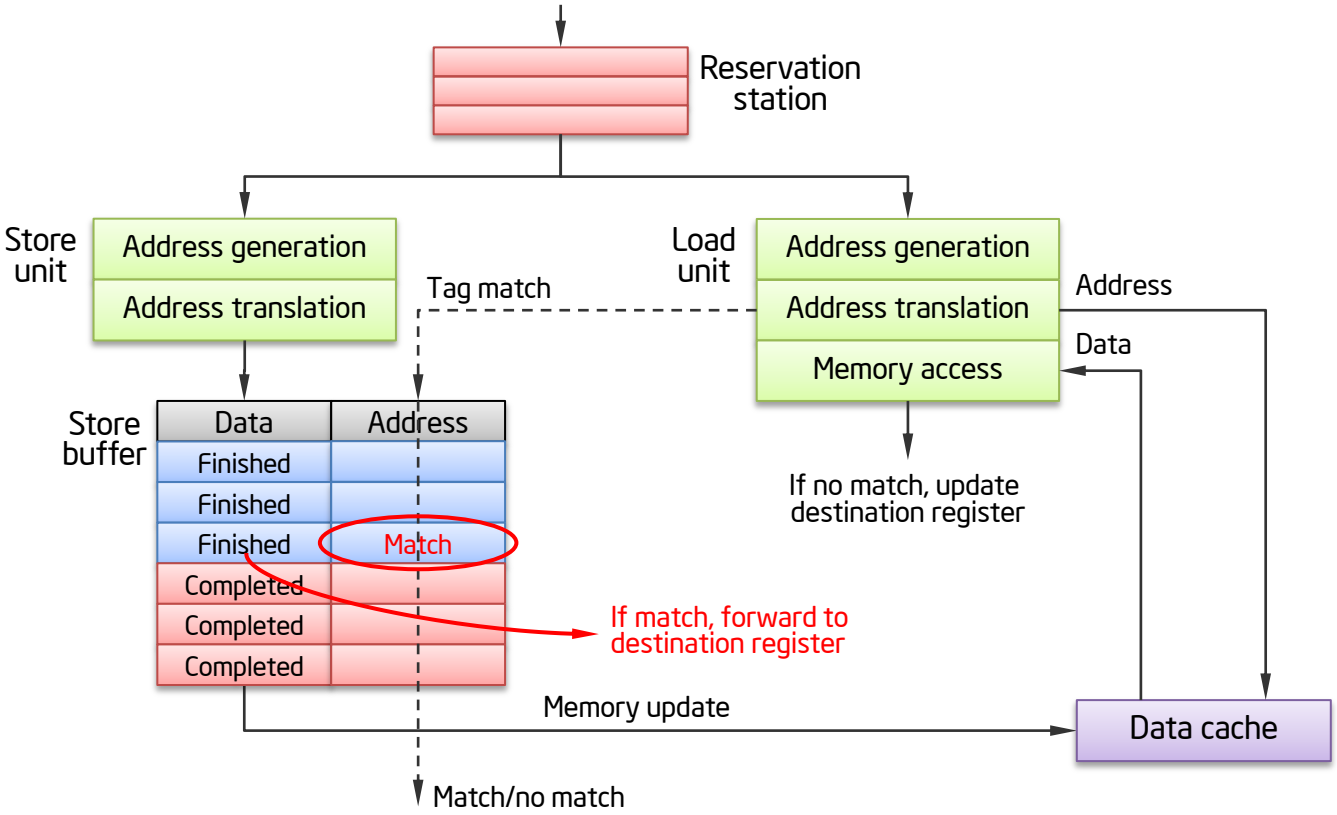


Load Bypassing



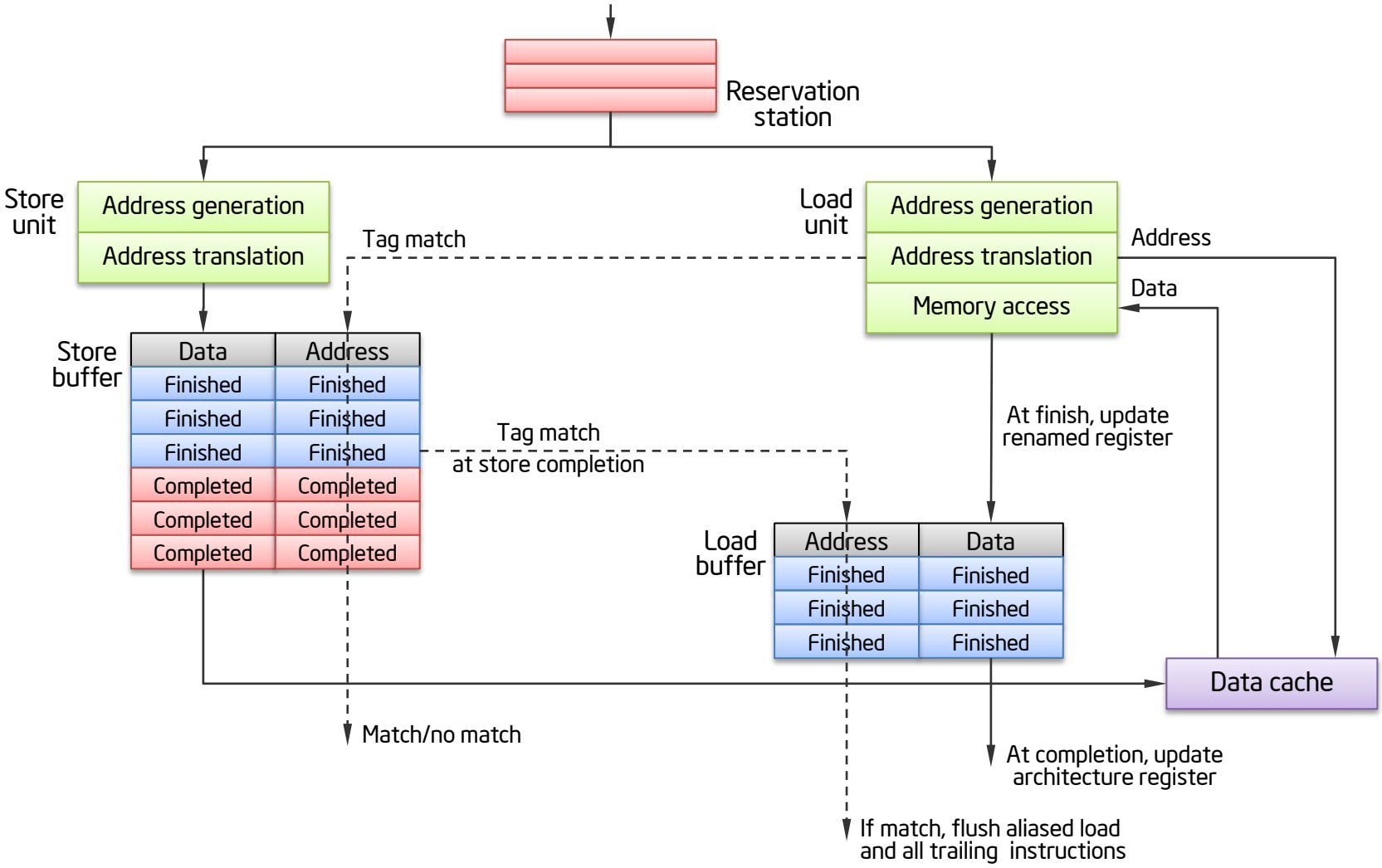
- Priority to loads over stores
  - +10% to 20% IPC over total ordering

Load Forwarding



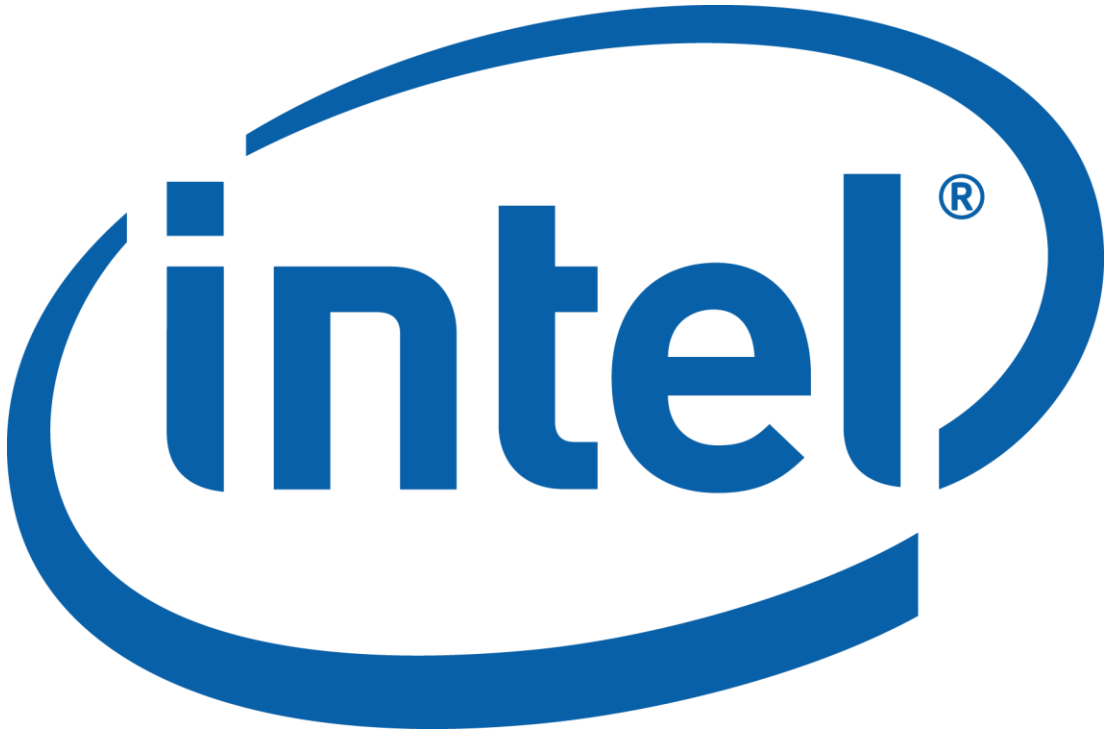
- Which store do we forward from?

Out of Order & Speculative Load Issue



Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - Krste Asanovic (MIT/UC Berkeley)
  - Christos Kozyrakis (Stanford University)


intel