18-640 Foundations of Computer Architecture

Lecture 8:
"Memory Data Flow Techniques"

John Paul Shen
September 18, 2014

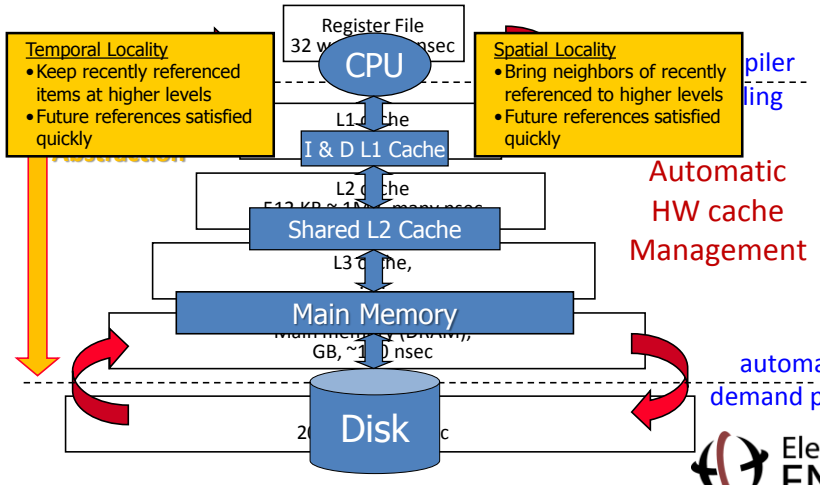
- Required Reading Assignment:
  - **Chapter 3 of Shen and Lipasti (SnL).**
- Recommended References:
  - Chapter 2 and Appendix B of Hennessey and Patterson 5th Edition.

18-640 Foundations of Computer Architecture

Lecture 8:
"Memory Data Flow Techniques"

A. Memory Hierarchy Revisited
B. Virtual Memory Revisited
C. Memory Data Flow Techniques
   a. Memory Data Dependences
   b. Load Bypassing
   c. Load Forwarding
   d. Speculative Disambiguation
   e. The Memory Bottleneck

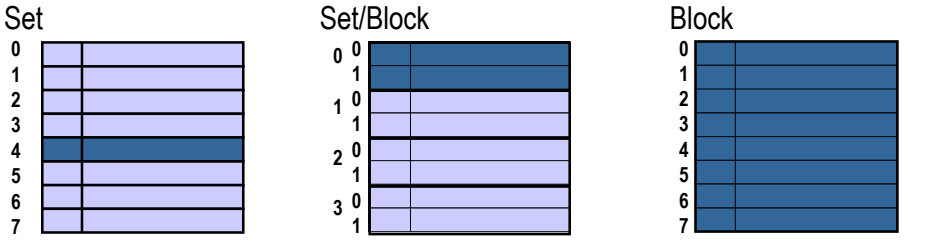
A. Memory Hierarchy Revisited



Block Placement and Replacement

Where should I go to look for address 12 (b'1100) on a read?

Which block can be evicted to make room for address 12 (b'1100) ?



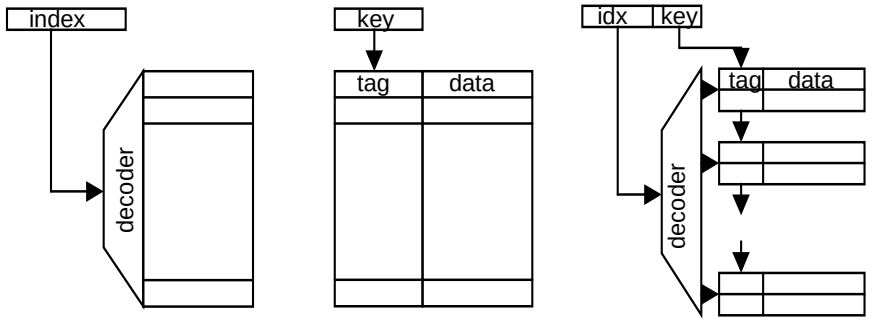
Direct-mapped
block goes in exactly one frame

Set-associative
a block goes any frame in exactly one set

Fully-associative
block goes in any frame

(think 1 frame per set) ⟷ (frames grouped into sets) ⟷ (think all frames in 1 set)

Cache Memory Implementation Options



Indexed Memory

k-bit index
2^k blocks

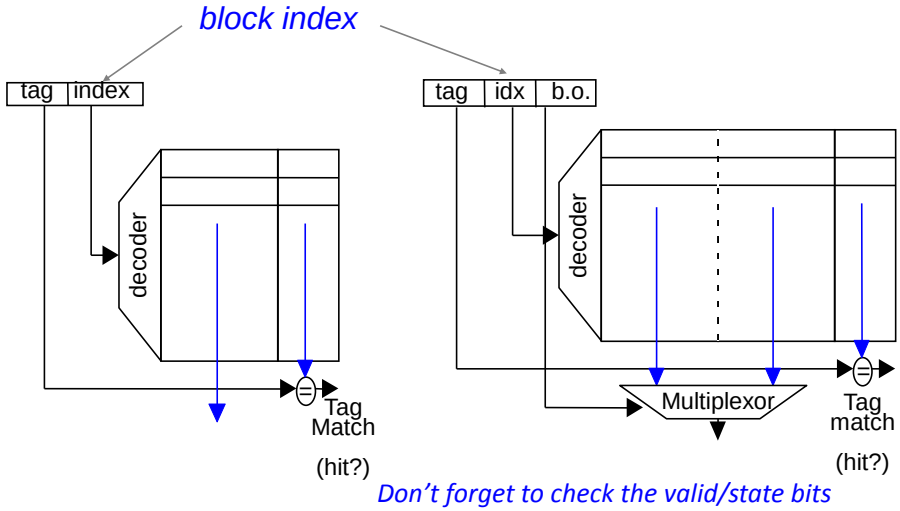
Associative Memory (CAM)

no index
unlimited blocks

N-Way Set-Associative Memory

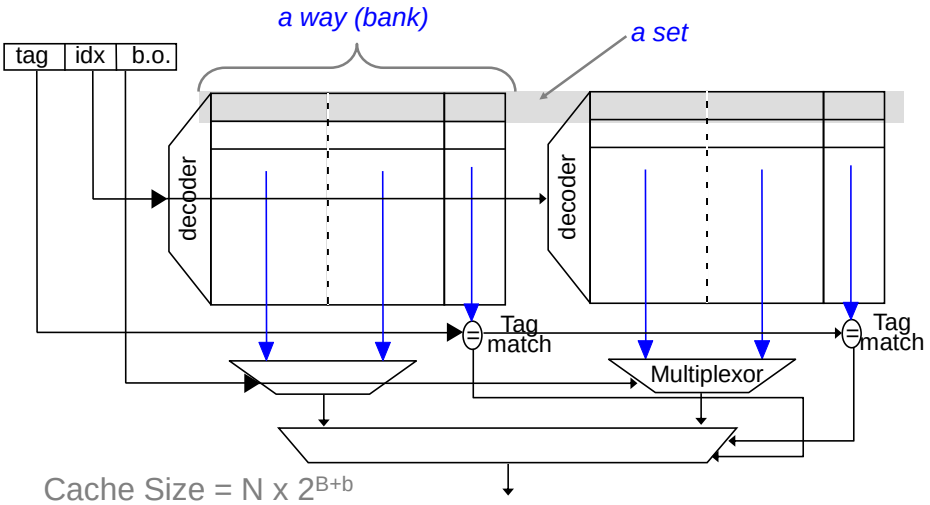
k-bit index
2^k • N blocks

Direct Mapped Caches



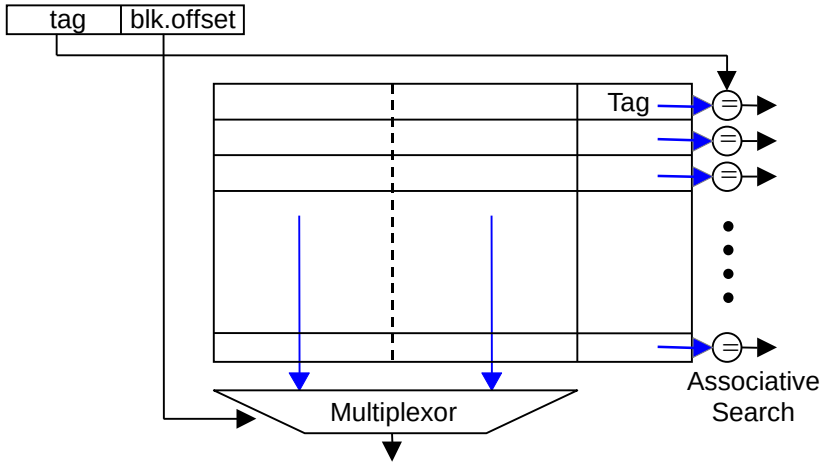
Don't forget to check the valid/state bits

N-Way Set Associative Cache

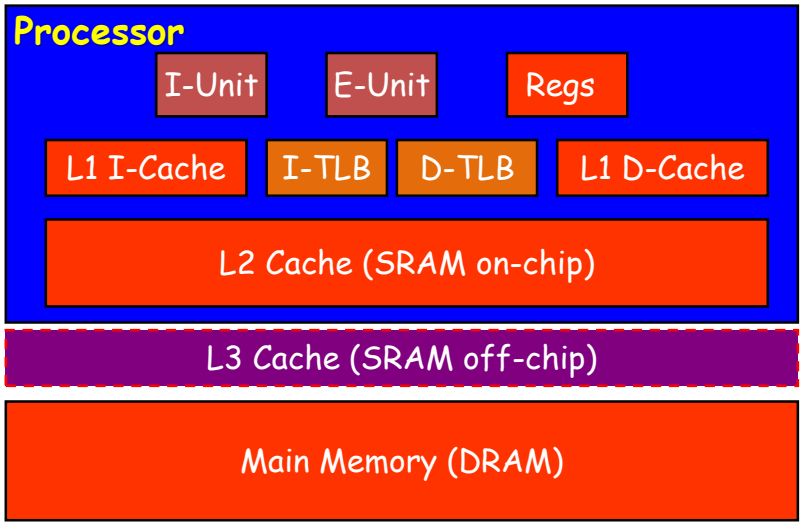


Cache Size = N x 2^{B+b}

Fully Associative Cache



Processor and Multi-level Cache Design



Cache Performance

$t_{hit,i}$ or hit time = time to access the cache

$t_{miss,i}$ or miss penalty =

- time to replace block in the cache + deliver to upper level
- **access time** = time to get first word
- **transfer time** = time for remaining words

$$t_{avg,i} = t_{hit,i} + \text{miss ratio} \times t_{miss,i}$$

this first

then this

$$t_{avg,i} = t_{hit,i} + \text{miss ratio} \times t_{miss,i}$$

$$t_{miss,i} = t_{avg,\ i+1}$$

Miss Classification (3+1 C's)

Compulsory
- "cold miss" on first access to a block
  - — defined as: miss in infinite cache

Capacity
- misses occur because cache not large enough
  - — defined as: miss that would occur in a fully-associative, perfect-replacement cache of the same capacity

Conflict
- misses occur because of restrictive mapping strategy
- only in set-associative or direct-mapped cache
  - — defined as: not attributable to compulsory or capacity

Coherence
- misses occur because of sharing among multiprocessors

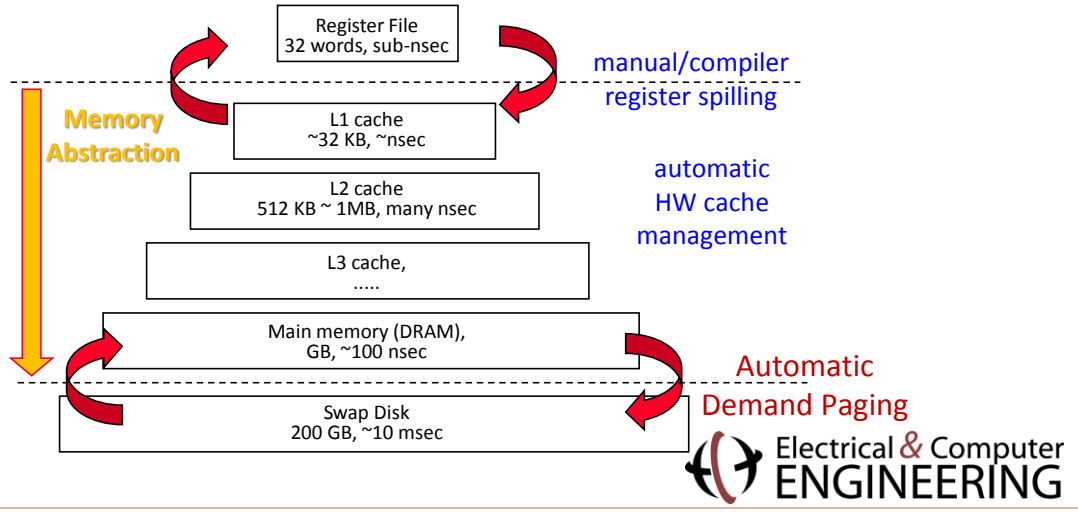
Improving Cache Performance: Summary

Miss Rate
- large block size
- higher associativity
- victim caches
- skewed-/pseudo-associativity
- hardware/software prefetching
- compiler optimizations

Miss Penalty
- give priority to read misses over writes/writebacks
- subblock placement
- early restart and critical word first
- non-blocking caches
- multi-level caches

Hit Time
- small and simple caches
- avoiding translation during L1 indexing (later)
- pipelining writes for fast write hits
- subblock placement for fast write hits in write through caches

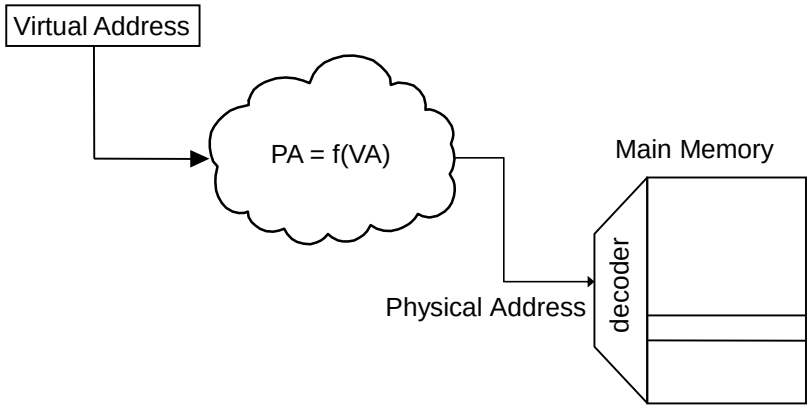
B. Virtual Memory Revisited



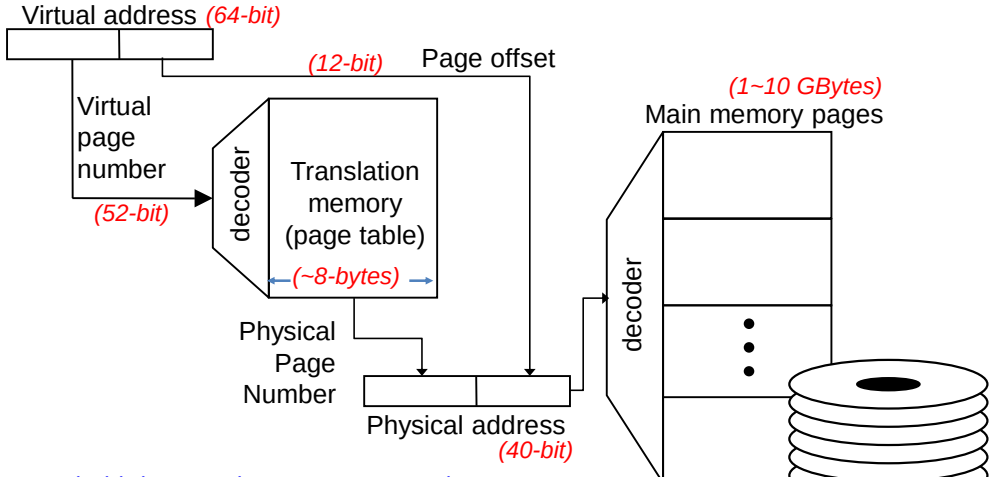
Two Parts to Modern Virtual Memory

- *In a multi-tasking system, Virtual Memory (VM) provides each process with the illusion of a large, private, uniform memory*
- Part A: Protection
  - Each process sees a large, contiguous memory segment without holes
  - Each process's memory space is private, i.e. protected from access by other processes
- Part B: Demand Paging
  - Capacity of secondary memory (swap space on disk)
  - At the speed of primary memory (DRAM)
- Based on a common HW mechanism: **address translation**
  - User process operates on "virtual" or "effective" addresses
  - HW translates from virtual to physical address on each reference
    - Controls which physical locations can be named by a process
    - Allows dynamic relocation of physical backing store (DRAM vs. HD)
  - VM HW and memory management policies controlled by the OS

Virtual to Physical Address Translation

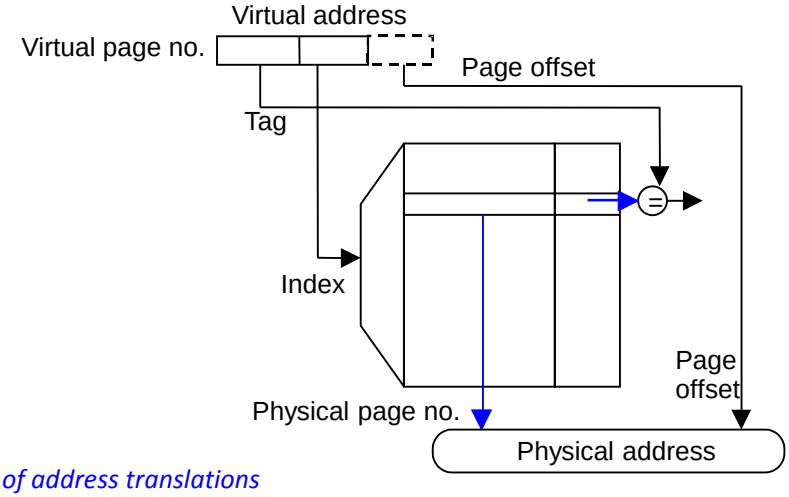


Page-Based Virtual Memory



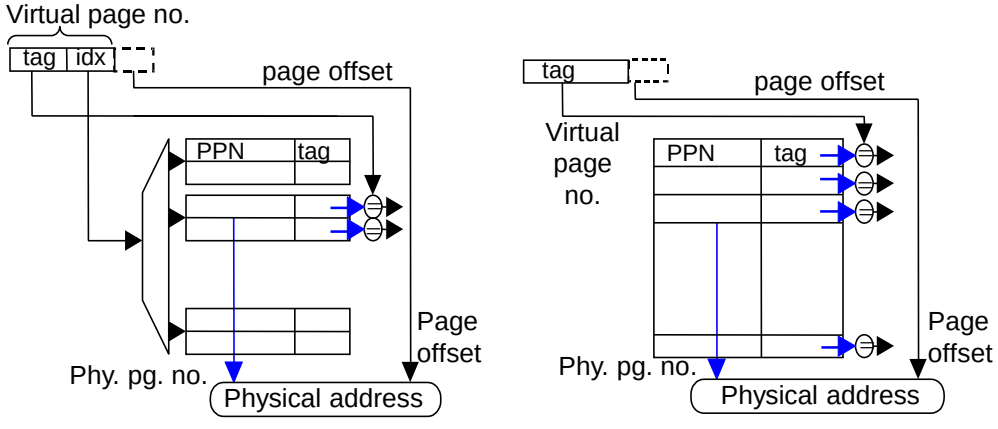
Where to hold this translation memory and how much translation memory do we need?

Translation Look-aside Buffer (TLB)

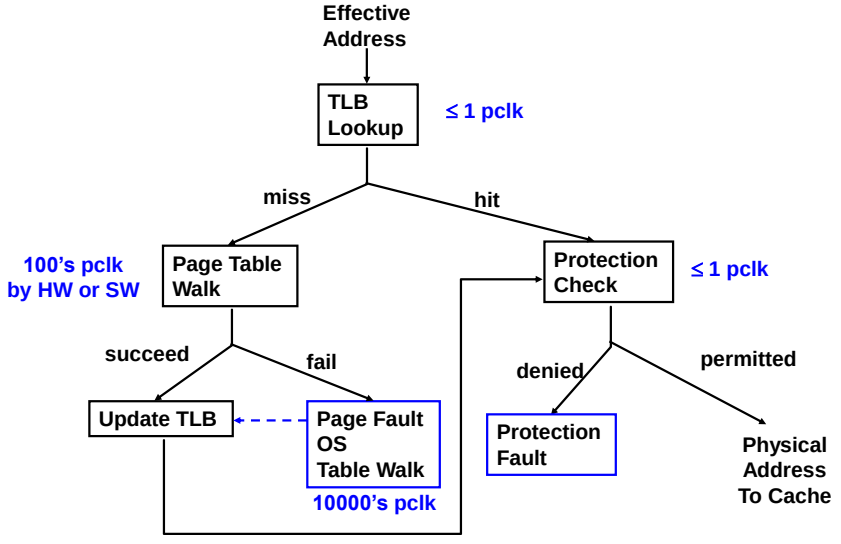


A cache of address translations

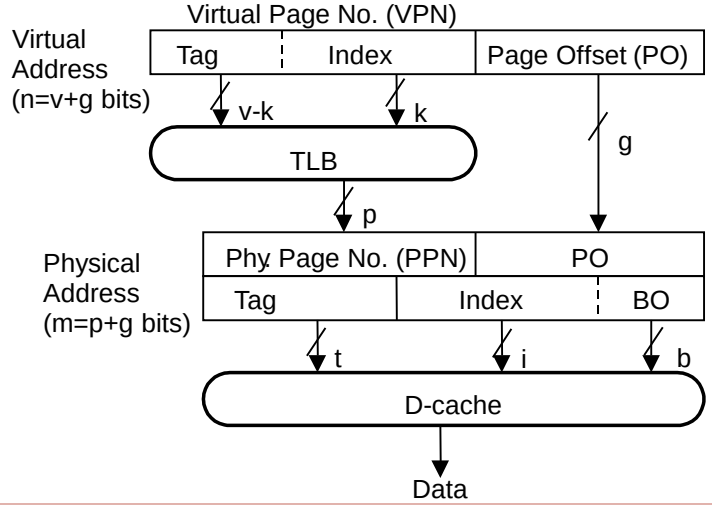
Set-Associative and Fully Associative TLBs



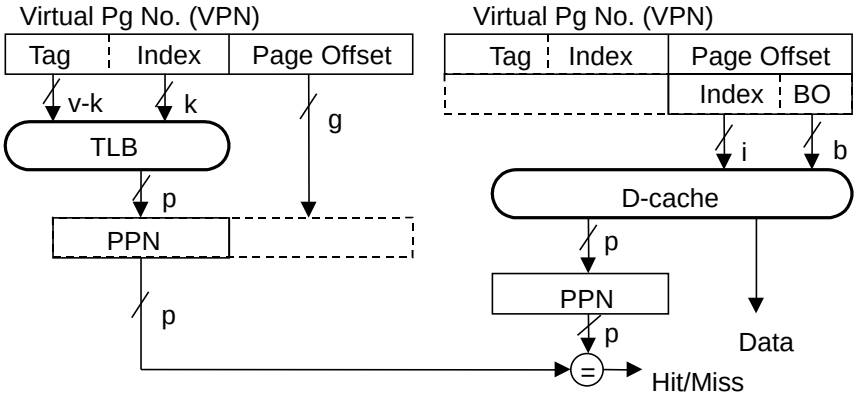
Virtual to Physical Address Translation



Physically Indexed Cache



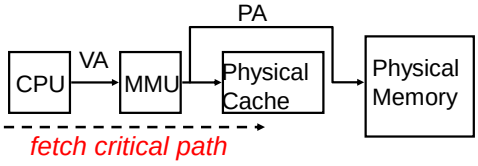
Virtually Indexed Cache



How big can a virtually indexed cache get?

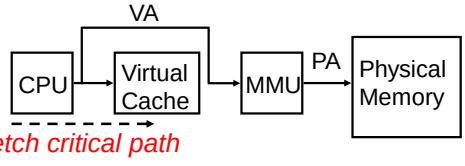
Cache Placement and Address Translation

Physical Cache (Most Systems)



longer hit time

Virtual Cache (SPARC2's)



aliasing problem

cold start after context switch

Virtual caches are not popular anymore because MMU and CPU can be integrated on one chip

C. Memory Data Flow Techniques

a. Memory Data Dependences
b. Load Bypassing
c. Load Forwarding
d. Speculative Disambiguation
e. The Memory Bottleneck

Electrical & Computer ENGINEERING

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)

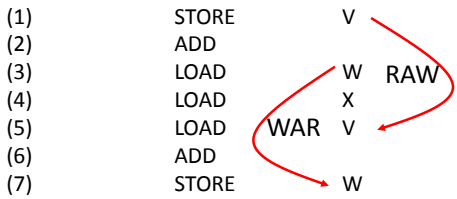
C.a. Memory Data Dependences

- 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 Dependence 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

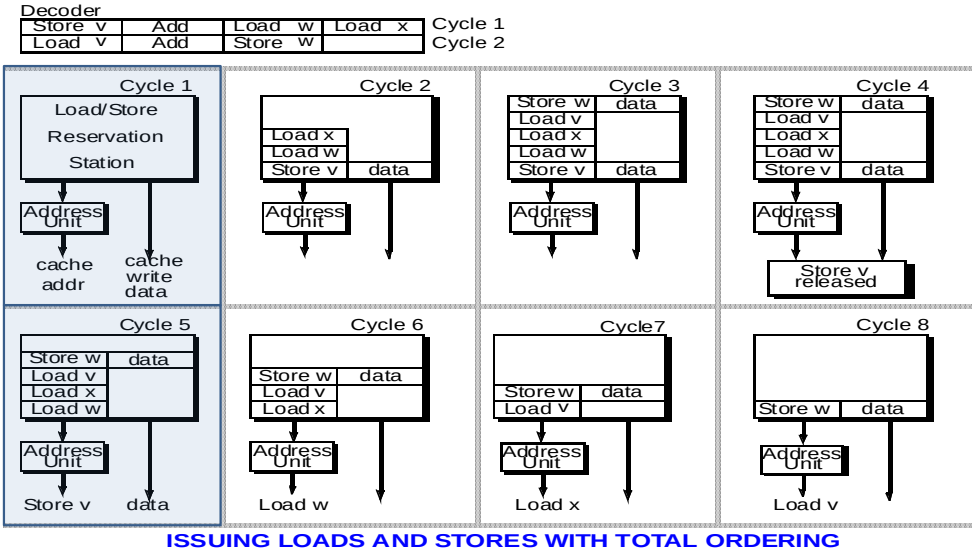
Example code:



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

Illustration of Total Ordering



ISSUING LOADS AND STORES WITH TOTAL ORDERING

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

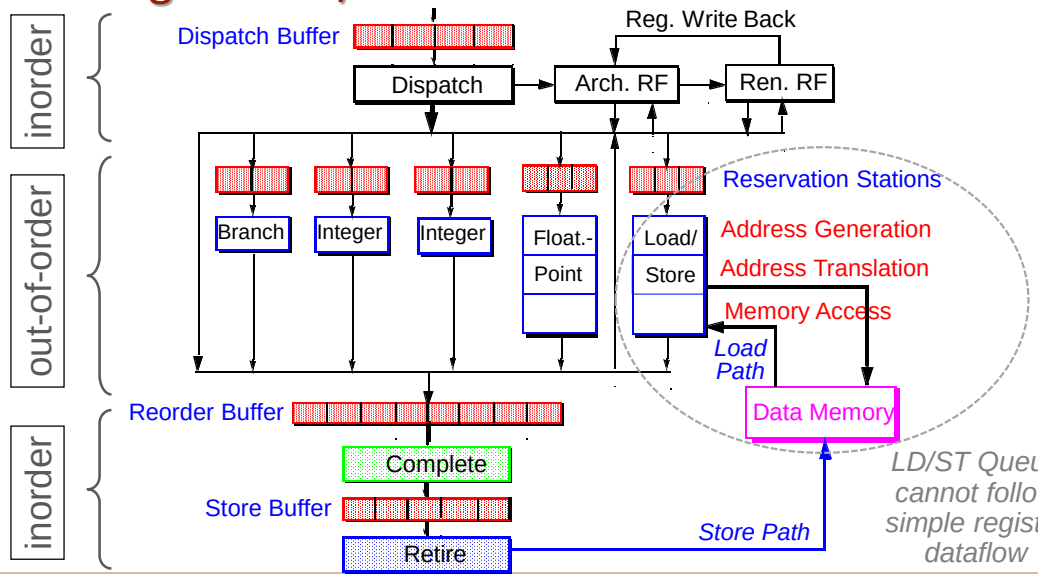
  No memory WAW or WAR

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

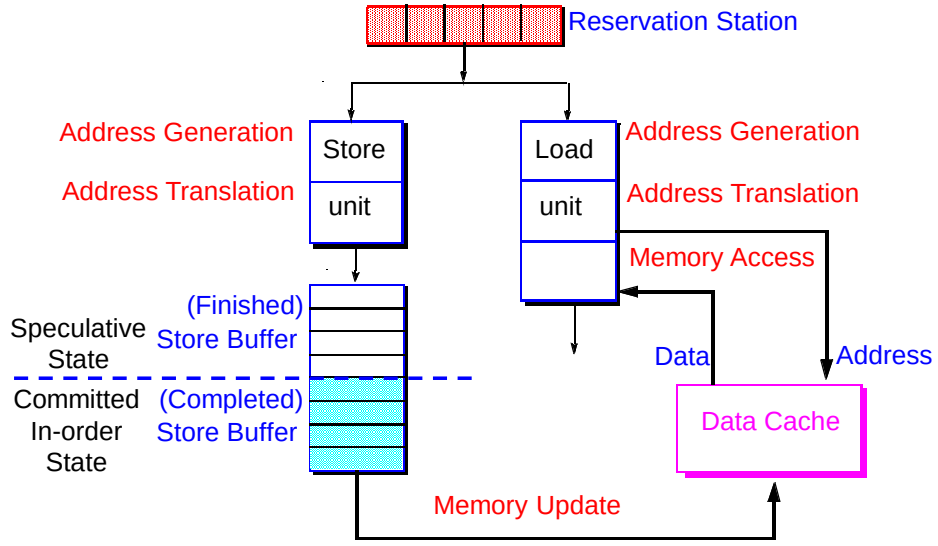
    or
  - Must know the addresses *(VA or PA??)* of all older stores

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

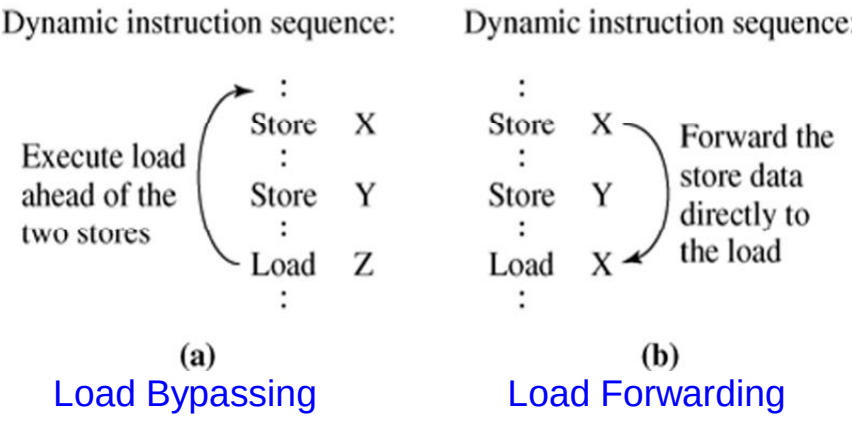
Processing of Load/Store Instructions



Load/Store Units and Store Buffer



Load Bypassing and Load Forwarding: Motivation



C.b. 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

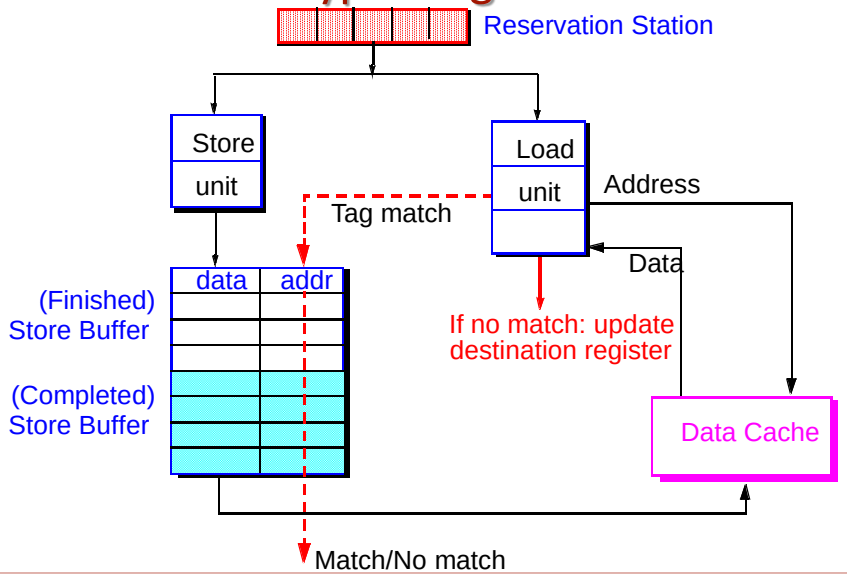
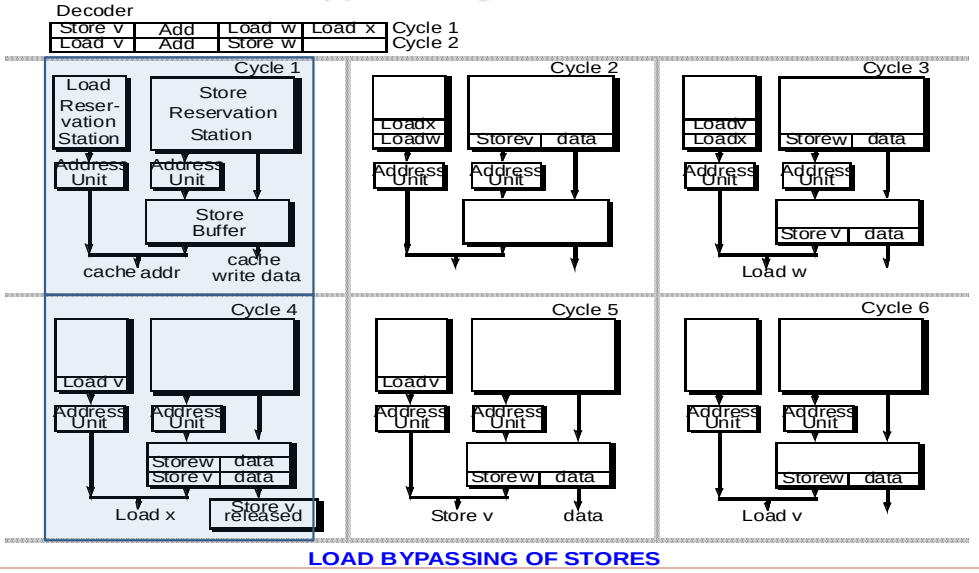


Illustration of Load Bypassing



C.c. 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

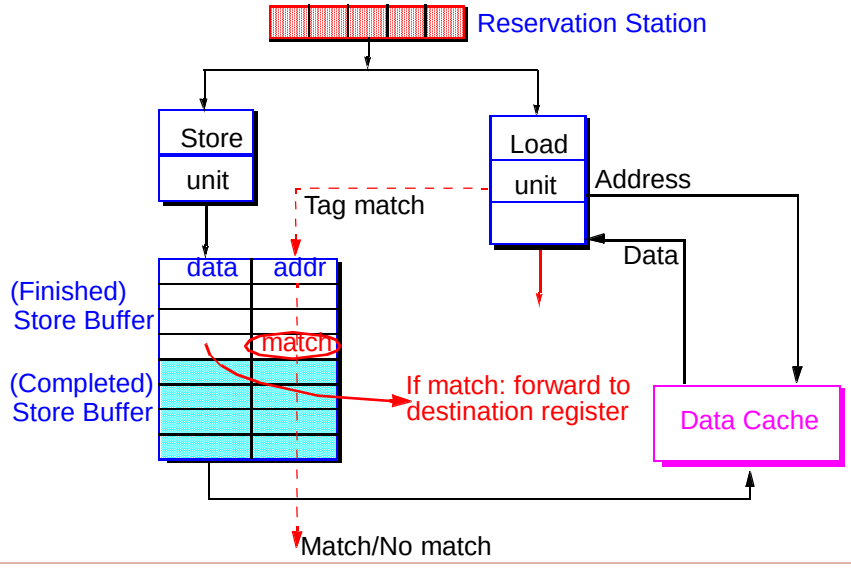
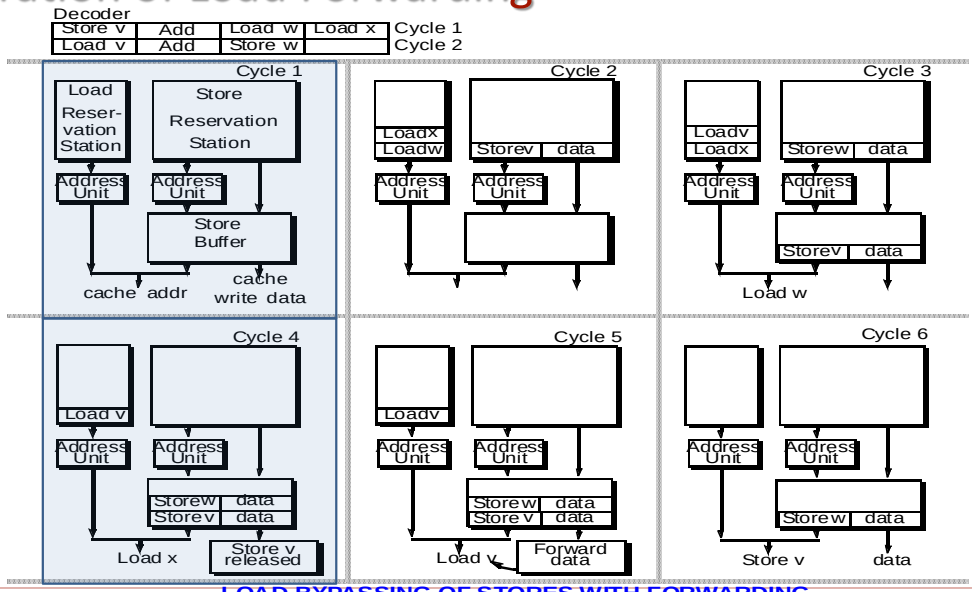


Illustration of Load Forwarding

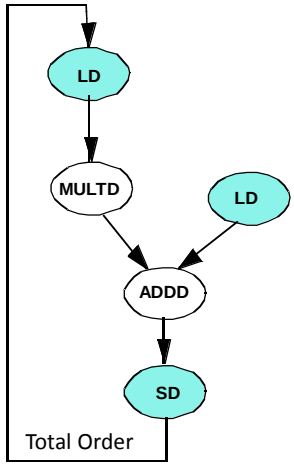


The "DAXPY" Example

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

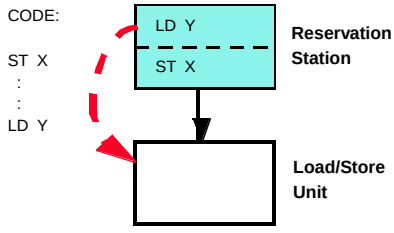
```
        LD      F0, a
        ADDI    R4, Rx, #512        ; last address

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
```

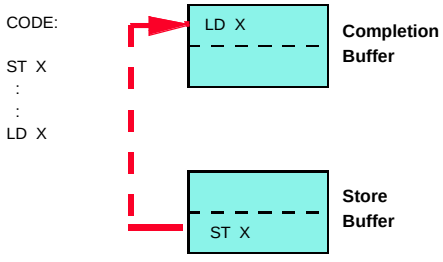


Performance Gains From Weak Ordering

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

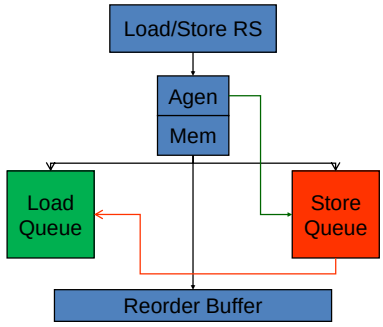
- Non-speculative load/store disambiguation
  1. Loads wait for addresses of all prior stores
  2. Full address comparison
  3. Bypass if no match, forward if match

- (1.) can limit performance:

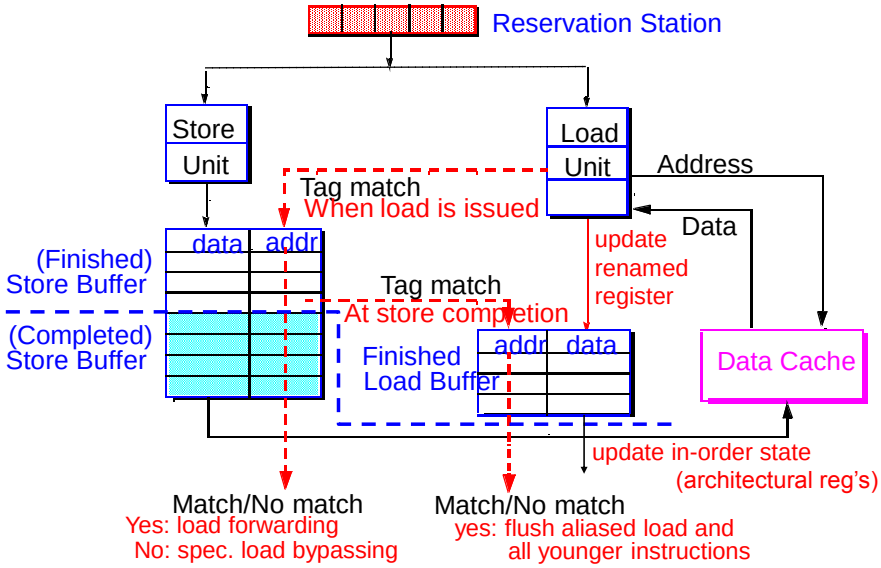
```
load    r5,MEM[r3]  ← cache miss
store   r7, MEM[r5] ← RAW for agen (addr. Gen.), stalled
…
load    r8, MEM[r9] ← independent load stalled
```

C.d. Speculative Disambiguation

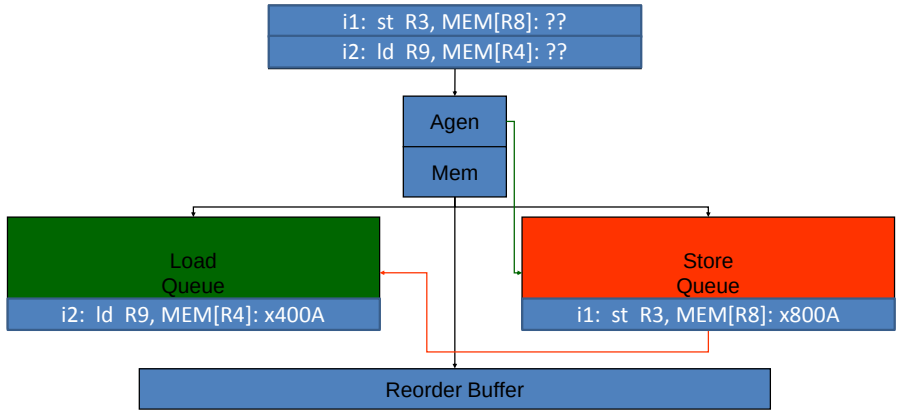
- What if aliases are rare?
  1. Loads don't need to wait for addresses of all prior stores
  2. Full address comparison of stores that are ready
  3. Bypass if no match, forward if match
  4. Check all store addresses when they commit
     - No matching loads – speculation was correct
     - Matching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



Speculative Load Bypassing

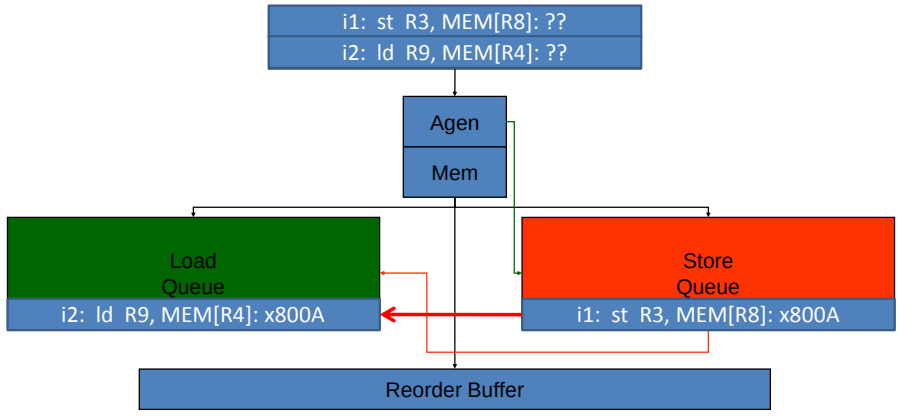


Speculative Disambiguation: Load Bypassing



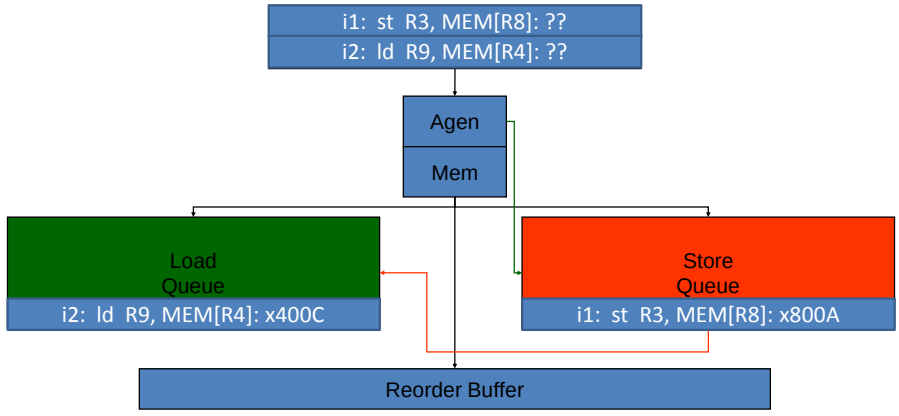
- i1 and i2 issue in program order
- i2 checks store queue (no match)

Speculative Disambiguation: Load Forwarding



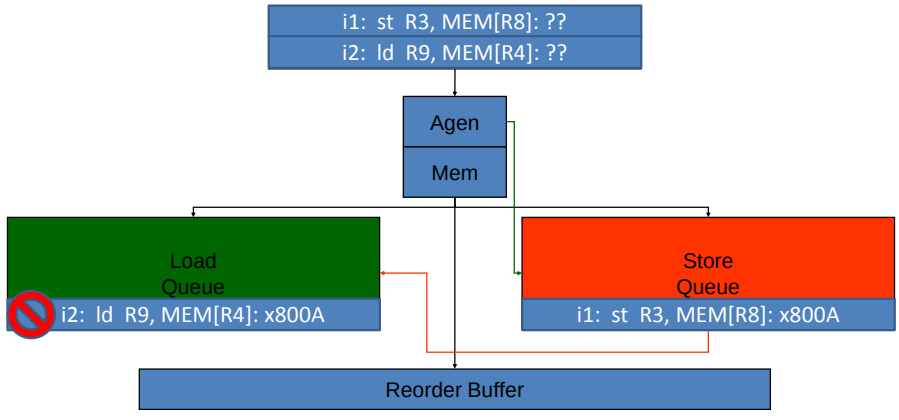
- i1 and i2 issue in program order
- i2 checks store queue (match=>forward)

Speculative Disambiguation: Safe Speculation



- i1 and i2 issue out of program order
- i1 checks load queue at commit (no match)

Speculative Disambiguation: Violation



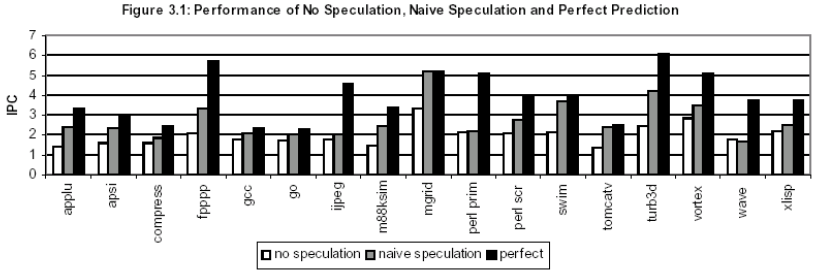
- i1 and i2 issue out of program order
- i1 checks load queue at commit (match)
  - i2 marked for replay

Use of Memory Alias Prediction

- If aliases are rare: static prediction
  - Predict no alias every time
    - Why even implement forwarding? PowerPC 620 doesn't
  - Pay mis-prediction penalty rarely
- If aliases are more frequent: dynamic prediction
  - Use PHT-like history table for loads
    - If alias predicted: delay load
    - If aliased pair predicted: forward from store to load
      - More difficult to predict pair [store sets, Alpha 21264]
  - Pay mis-prediction penalty rarely
- Memory cloaking [Moshovos, Sohi]
  - Predict load/store pair
  - Directly copy store data register to load target register
  - Reduce data transfer latency to absolute minimum

Motivation for Selective Speculation

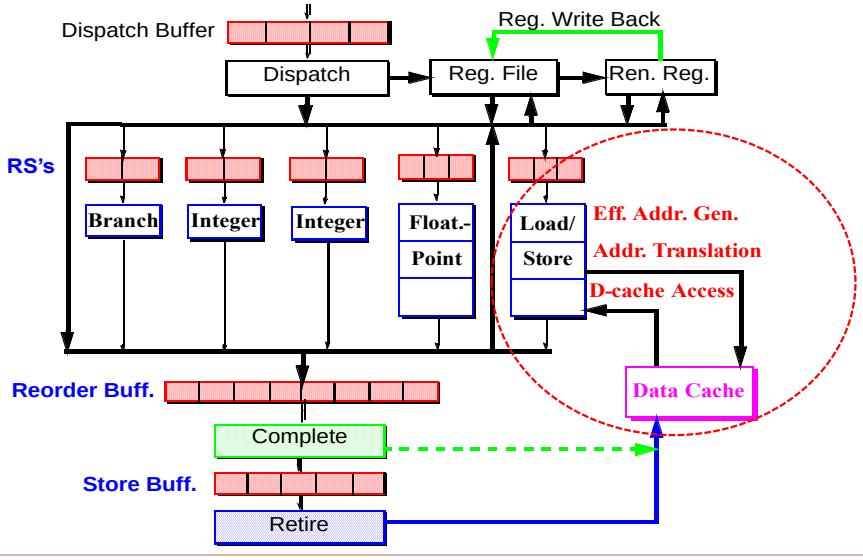
- Speculation options when store address not available
  - Pessimistic: always assume dependence is there
  - Naïve: always assume there is no dependence
  - Selective: use a scheme to select profitable loads to speculate
- What is the effect of dependence misprediction?



Load/Store Disambiguation Discussion

- RISC ISA:
  - Many registers, most variables allocated to registers; Aliases are rare
  - Most important to not delay loads (bypass)
  - Alias predictor may/may not be necessary
- CISC ISA:
  - Few registers, many operands from memory
  - Aliases much more common, forwarding necessary
  - Incorrect load speculation should be avoided
  - If load speculation allowed, predictor probably necessary
- Address Translation:
  - Can't use virtual address (must use physical)
  - Wait till after TLB lookup is done
  - Or, use subset of untranslated bits (page offset)
    - Safe for proving inequality (bypassing OK)
    - Not sufficient for showing equality (forwarding not OK)

C.e. The Memory Bottleneck



Summary of Load/Store Processing

For both Loads and Stores:

1. Effective Address Generation:
   - Must wait on register value
   - Must perform address calculation
2. Address Translation:
   - Must access TLB
   - Can potentially induce a page fault (exception)

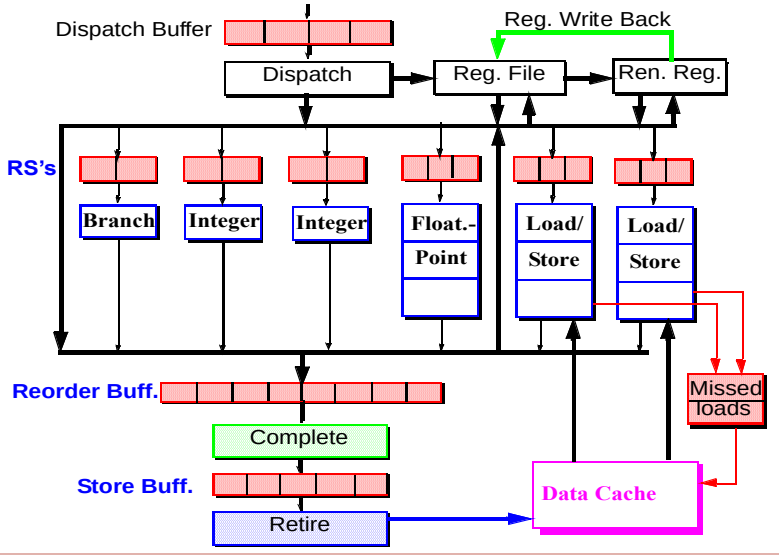
For Loads: D-cache Access (Read)

- Can potentially induce a D-cache miss
- Check aliasing against store buffer for possible load forwarding
- If bypassing store, must be flagged as "speculative" load until completion

For Stores: D-cache Access (Write)

- When completing must check aliasing against "speculative" loads
- After completion, wait in store buffer for access to D-cache
- Can potentially induce a D-cache miss

Easing the Memory Bottleneck (missed-load buffer)



Memory Bottleneck Techniques

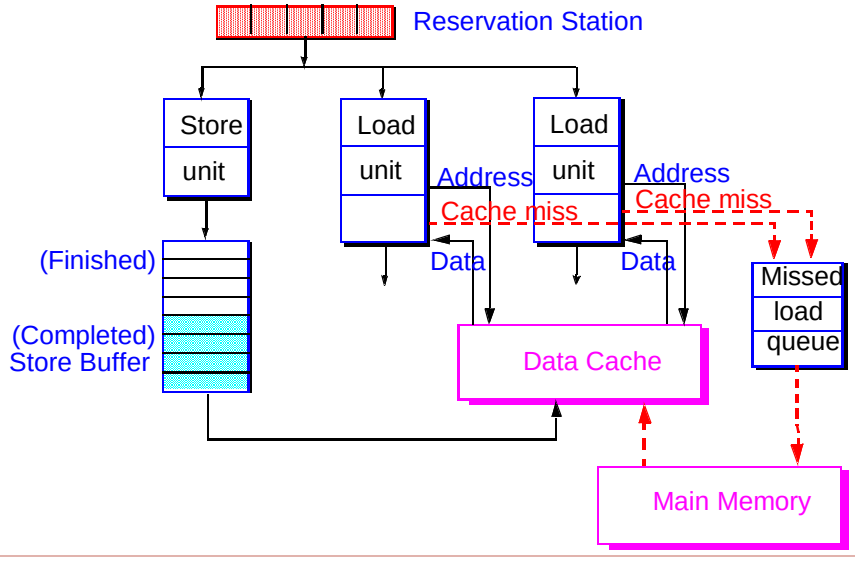
Dynamic Hardware (Microarchitecture):

- 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)

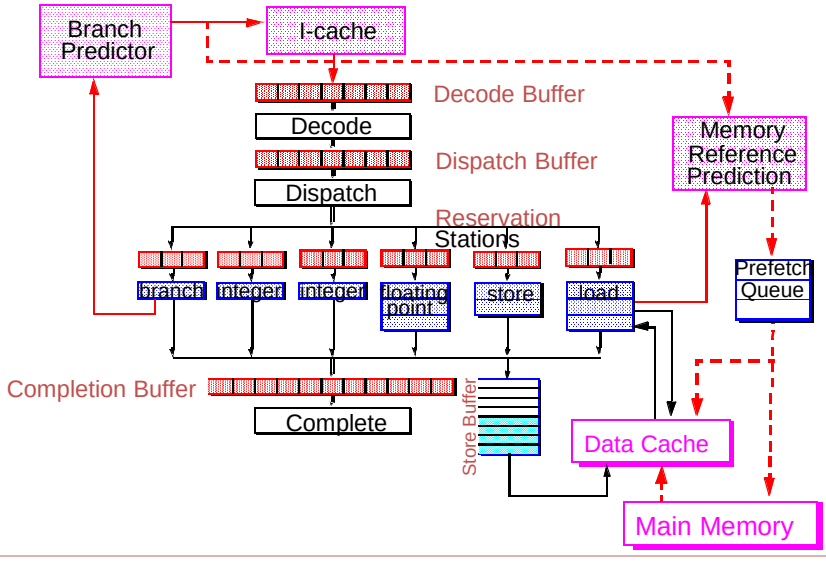
Static Software (Code Transformation):

- 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)

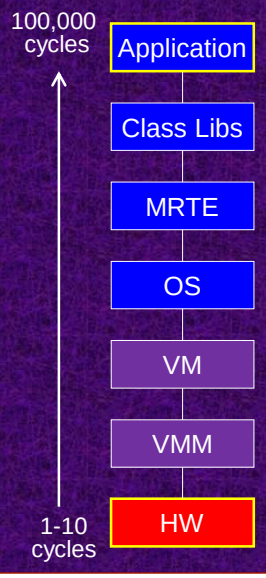
Dual-Ported Non-Blocking Cache



Prefetching Data Cache



The SW Stack



The HW Stack