18-747 Lecture 12: Memory Dataflow Techniques

James C. Hoe
Dept of ECE, CMU
October 8, 2001

Reading Assignments: *MJ Ch8*

Announcements: Exam 1 on Monday 10/15

*** This is the last lecture to be included on Exam 1*

**** Exam 2 on Monday 12/3**

HW 2 due Wednesday 10/10 (start of class)

Project 1 due Friday 10/12

Handouts: *"The micorarchitecture of superscalar processors"*

Practice exam solution

Principle Behind Hierarchical Storage

- Each level memoizes values stored at lower levels
- Instead of paying the full latency for the "furthermost" level of storage each time

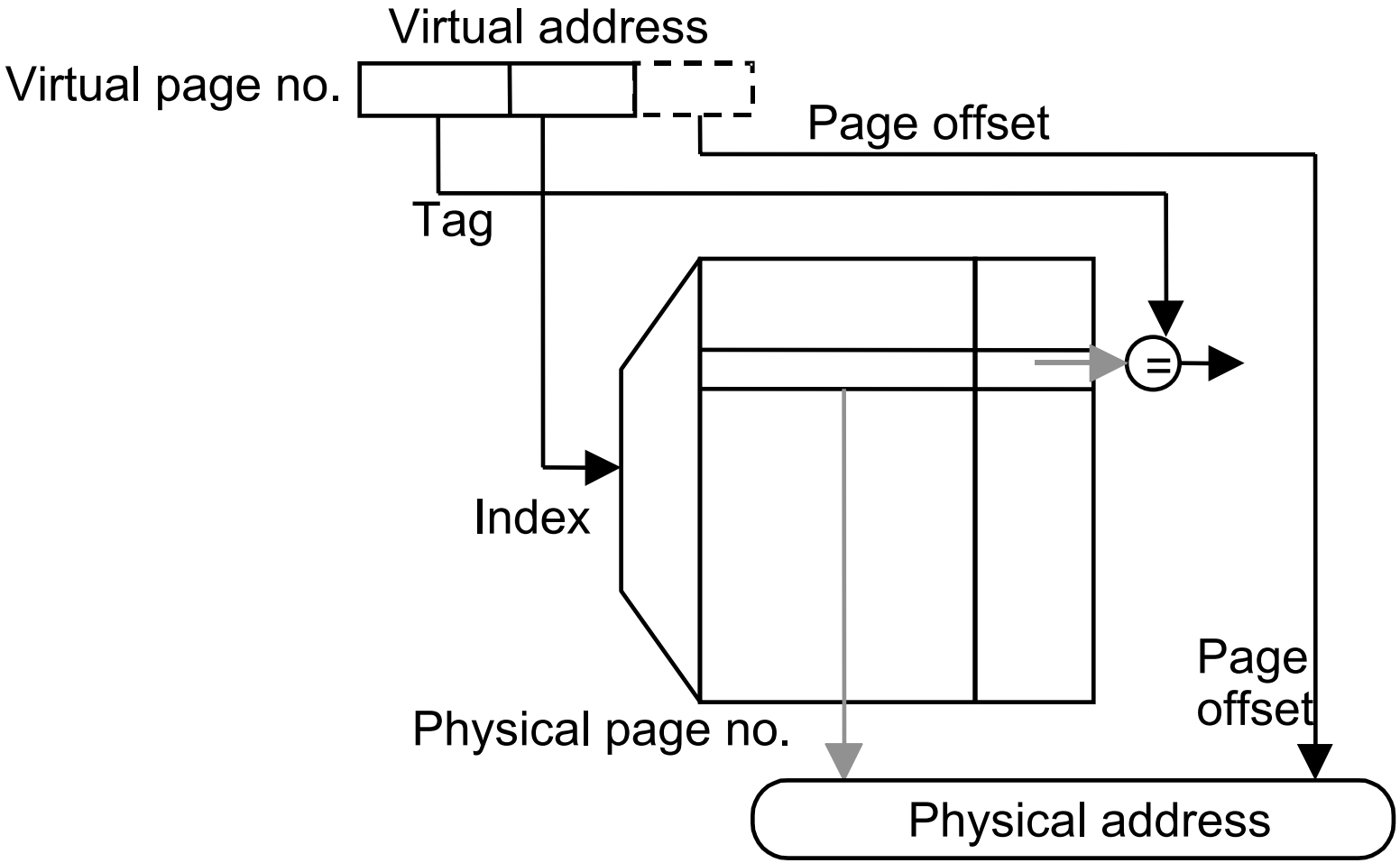
  Effective Access $T_i = h_i \cdot t_i + (1 - h_i) \cdot T_{i+1}$

  - where h_i is the 'hit' ratio, the probability of finding the desired data memoized at level i
  - t_i is the raw access time of memory at level i
- Given a program with good locality of reference

  $S_{working\text{-}set} < s_i \Rightarrow h_i \approx 1 \Rightarrow T_i \approx t_i$
- A balanced system achieves the best of both worlds
  - the performance of higher-level storage
  - the capacity of lower-level low-cost storage.

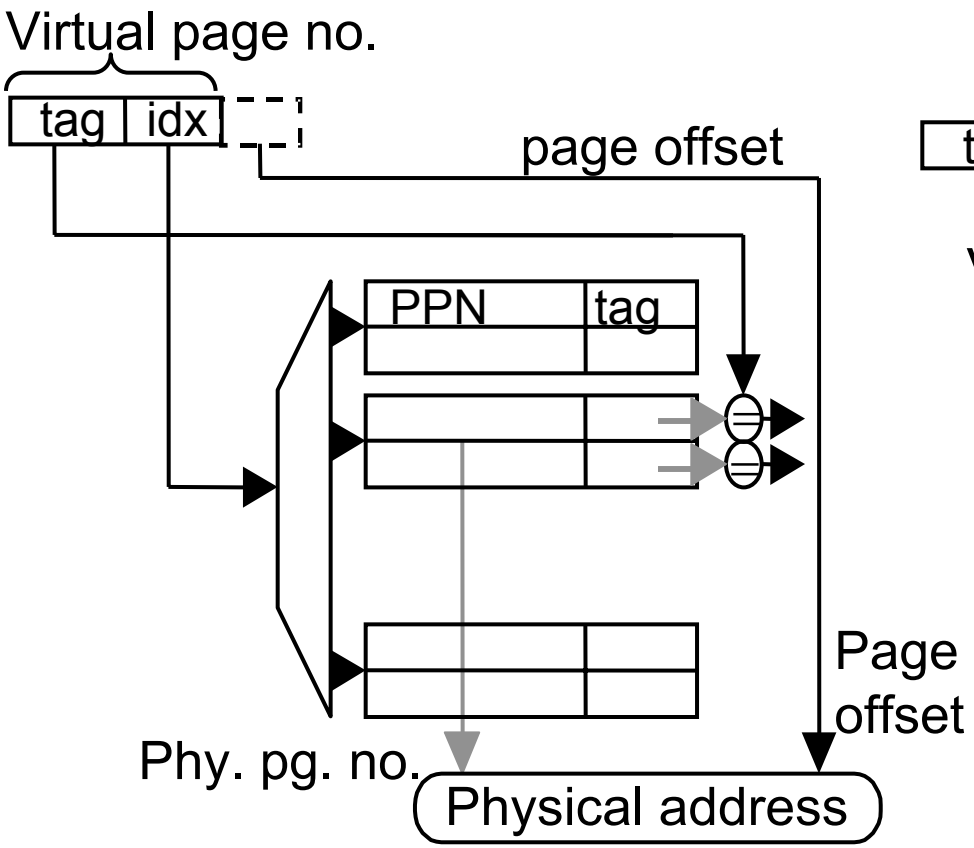
Assumption: faster memory is more expensive

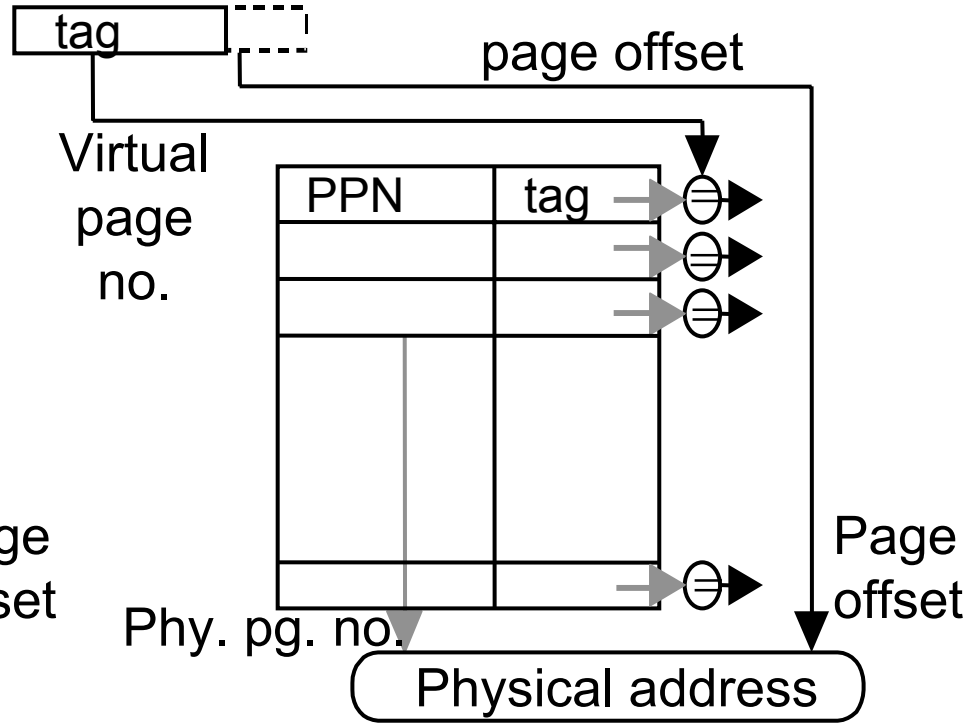
Translation Look-aside Buffer (TLB)



A cache of address translations

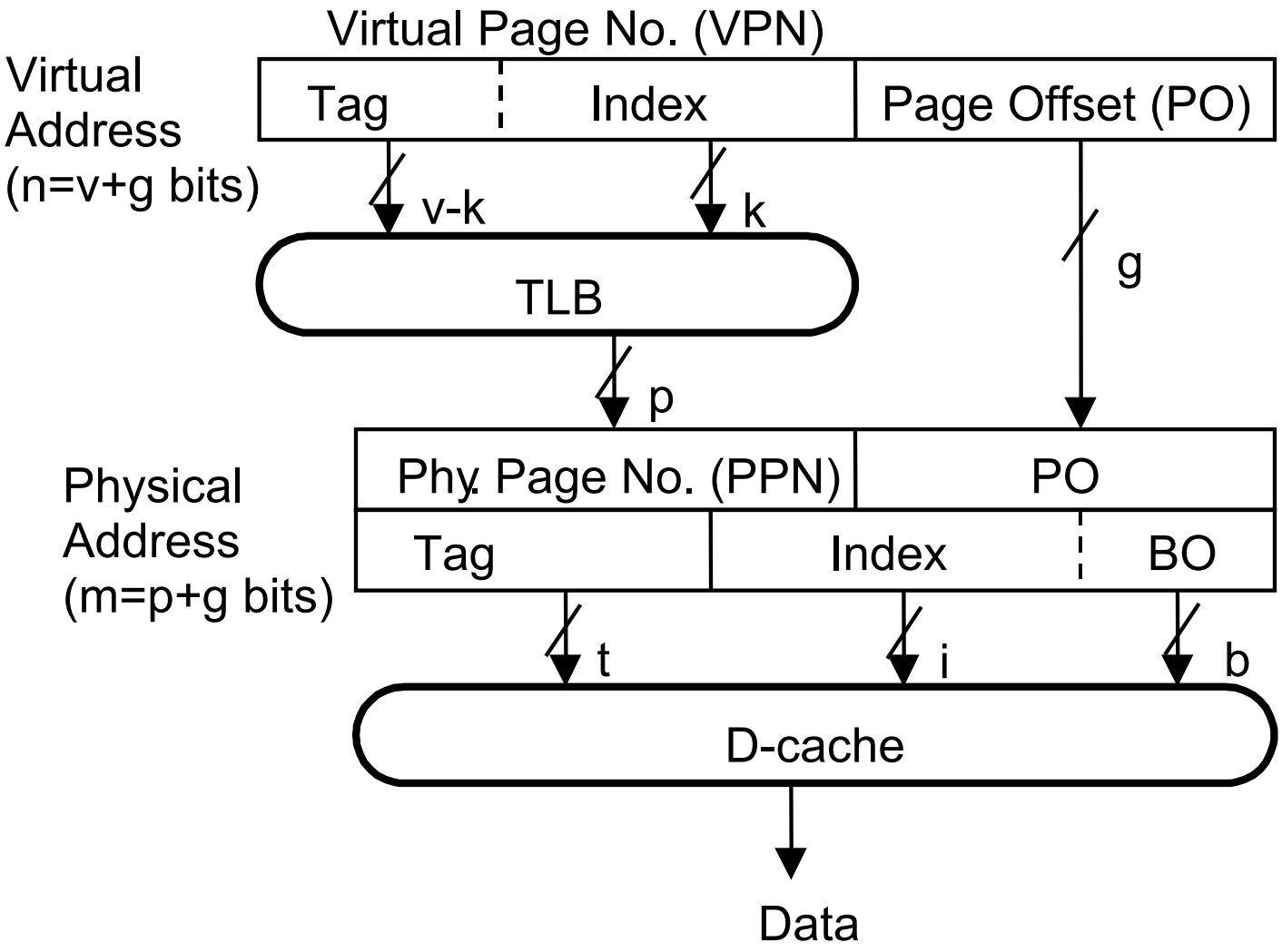
Set-Associative and Fully Associative TLBs





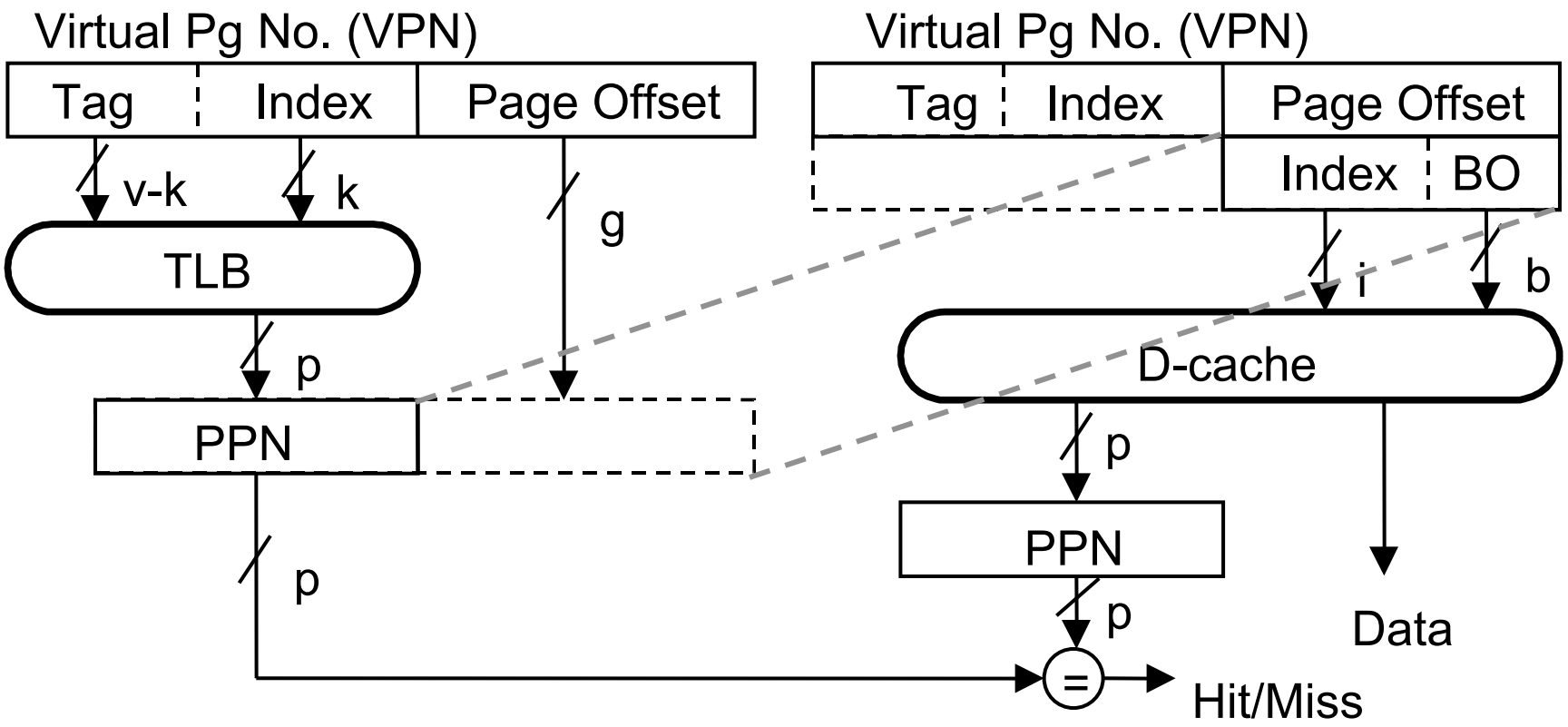
What are relative sizes of ITLB, BTB and I-cache?

Physically Indexed Cache



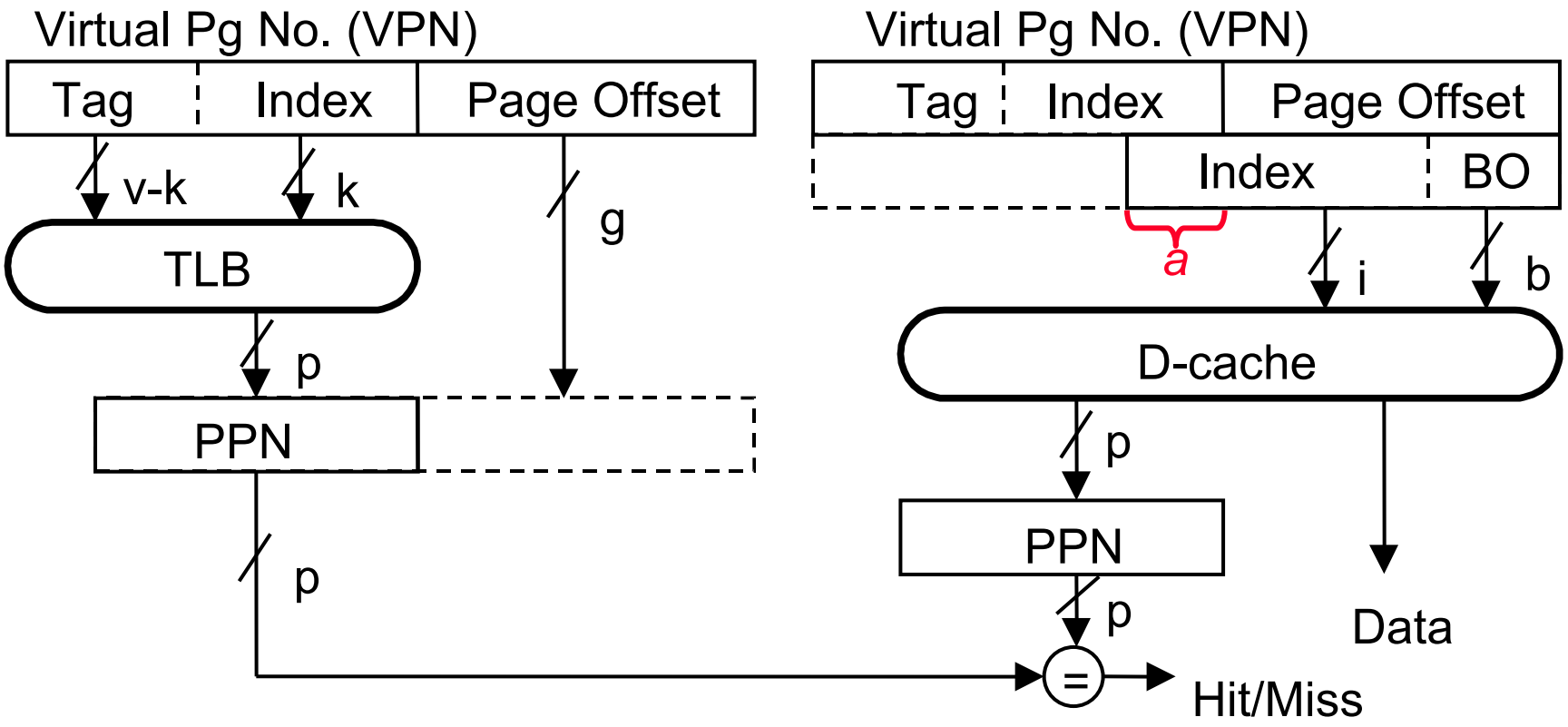
Virtually Indexed Cache

Parallel Access to TLB and Cache arrays



How large can a virtually indexed cache get?

Large Virtually Indexed Cache

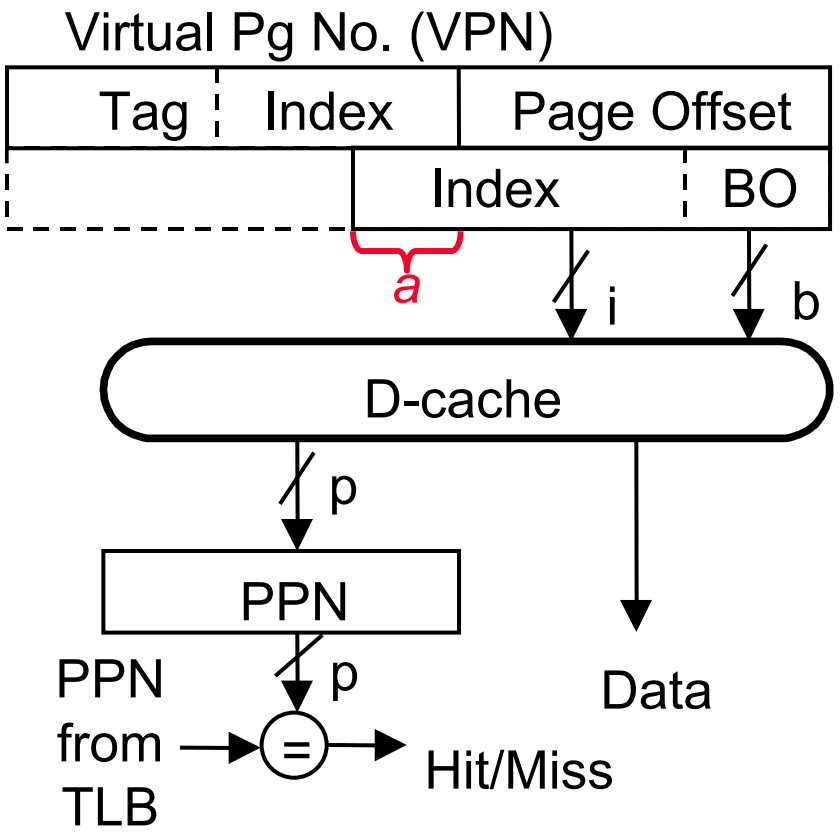


If two VPNs differs in *a*, but both map to the same PPN then there is an aliasing problem

Synonym (or Aliasing)

- When VPN bits are used in indexing, two virtual addresses that map to the same physical address can end up sitting in two cache lines
- In other words, two copies of the same physical memory location may exist in the cache

  ⇒ *modification to one copy won't be visible in the other*

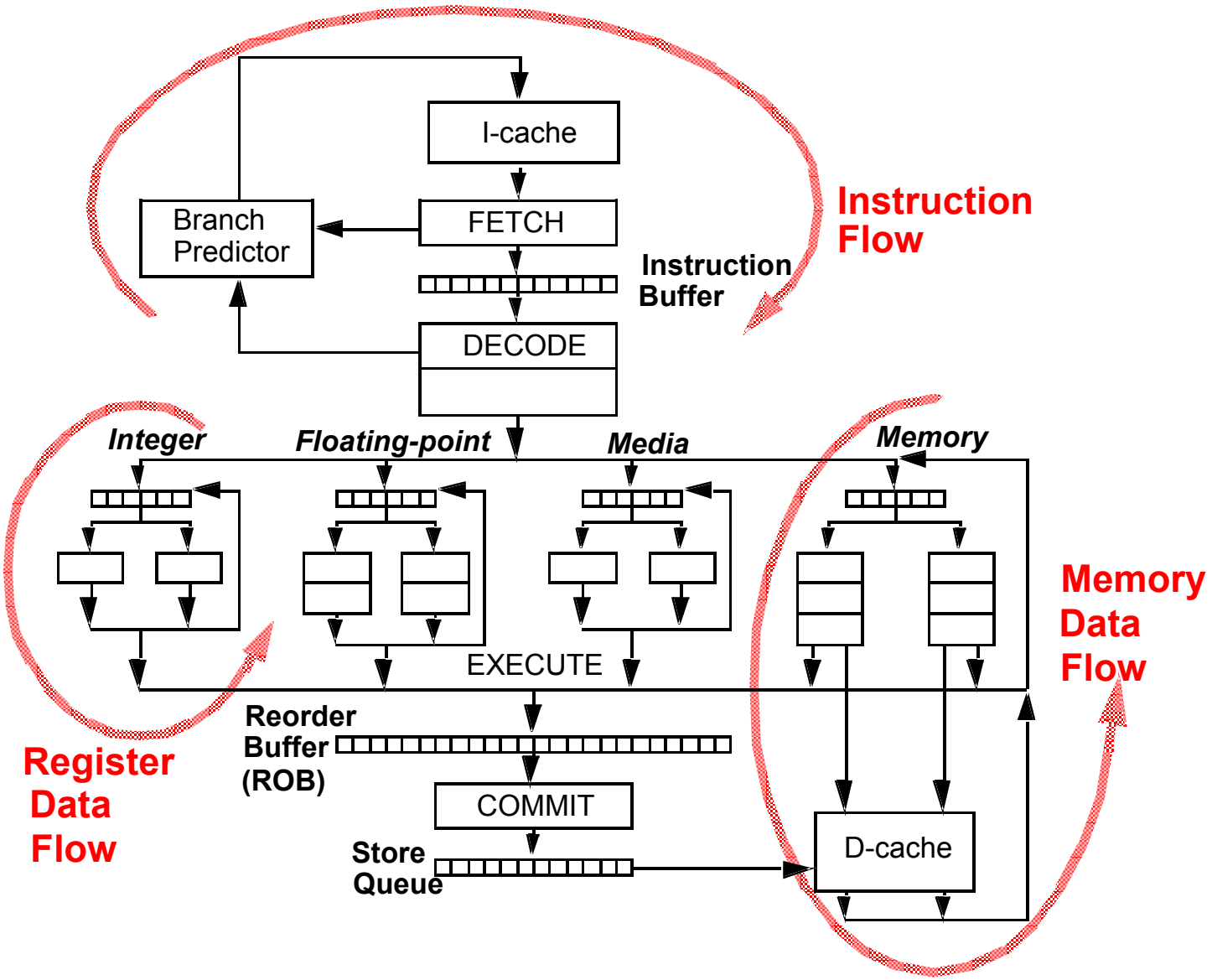


*If the two VPNs do not differ in **a** then there is no aliasing problem*

R10000's Virtually Index Caches

- 32KB 2-Way Virtually-Indexed L1
  - needs 10 bits of index and 4 bits of block offset
  - page offset is only 12-bits $\Rightarrow$ 2 bits of index are VPN[1:0]
- Direct-Mapped Physical L2
  - L2 is *Inclusive* of L1
  - VPN[1:0] is appended to the "tag" of L2
- Given two virtual addresses *VA* and *VB* that differs in *a* and both map to the same physical address *PA*
  - Suppose *VA* is accessed first so blocks are allocated in L1&L2
  - What happens when *VB* is referenced?
    1. *VB* indexes to a different block in L1and misses
    2. *VB* translates to *PA* and goes to the same block as *VA* in L2
    3. Tag comparison fails (*VA*[1:0]$\neq$*VB*[1:0])
    4. Treated just like as a L2 conflict miss $\Rightarrow$ *VA*'s entry in L1 and L2 are both ejected due to inclusion policy

Memory Dataflow Techniques



Uniprocessor Load and Store Semantics

- Given $Store_i(\ a, v\ ) << Load_j(\ a\ )$ *("<<" means precedes)*

  Load(*a*) must return *v* if there does not exist another $Store_k$ such that

  $$Store_i(\ a, v\ ) << Store_k(\ a, v'\ ) << Load_j(a)$$

- This can be guaranteed by observing data dependence
  - RAW Store(*a*, *v*) followed by Load(*a*)
  - WAW Store(*a*, *v'*) followed by Store(*a*, *v*)
  - WAR Load(*a*) followed by Store(*a*, *v'*)

For a uniprocessor, do we need to worry about loads and stores to different addresses? What about SMPs?

Total Ordering of Loads and Stores

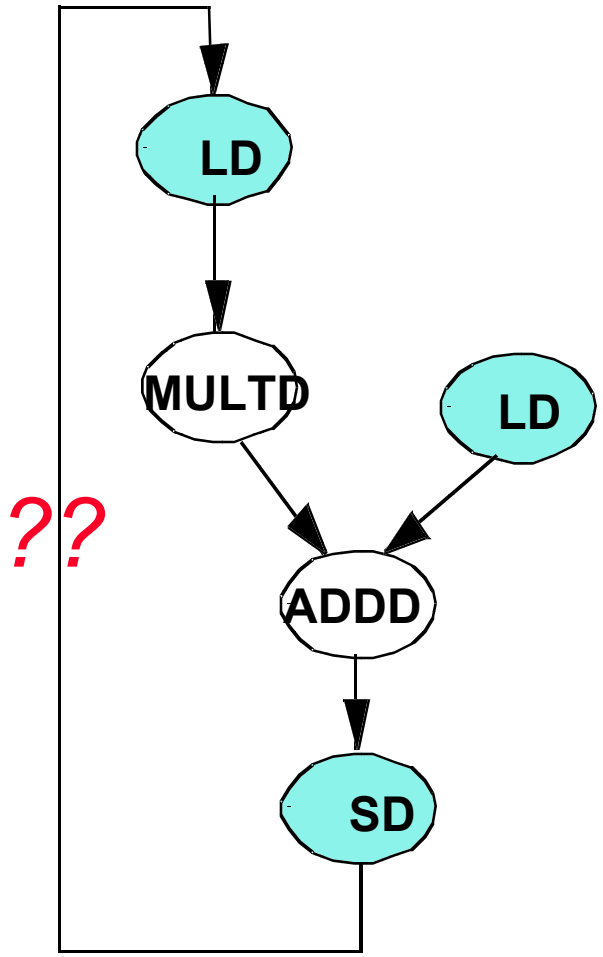
- Keep all loads and stores totally in order with respect to each other
- However, loads and stores can execute out of order with respect to other types of instructions (while obeying register data-dependence)

Except, stores must still be held for all previous instructions

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
```



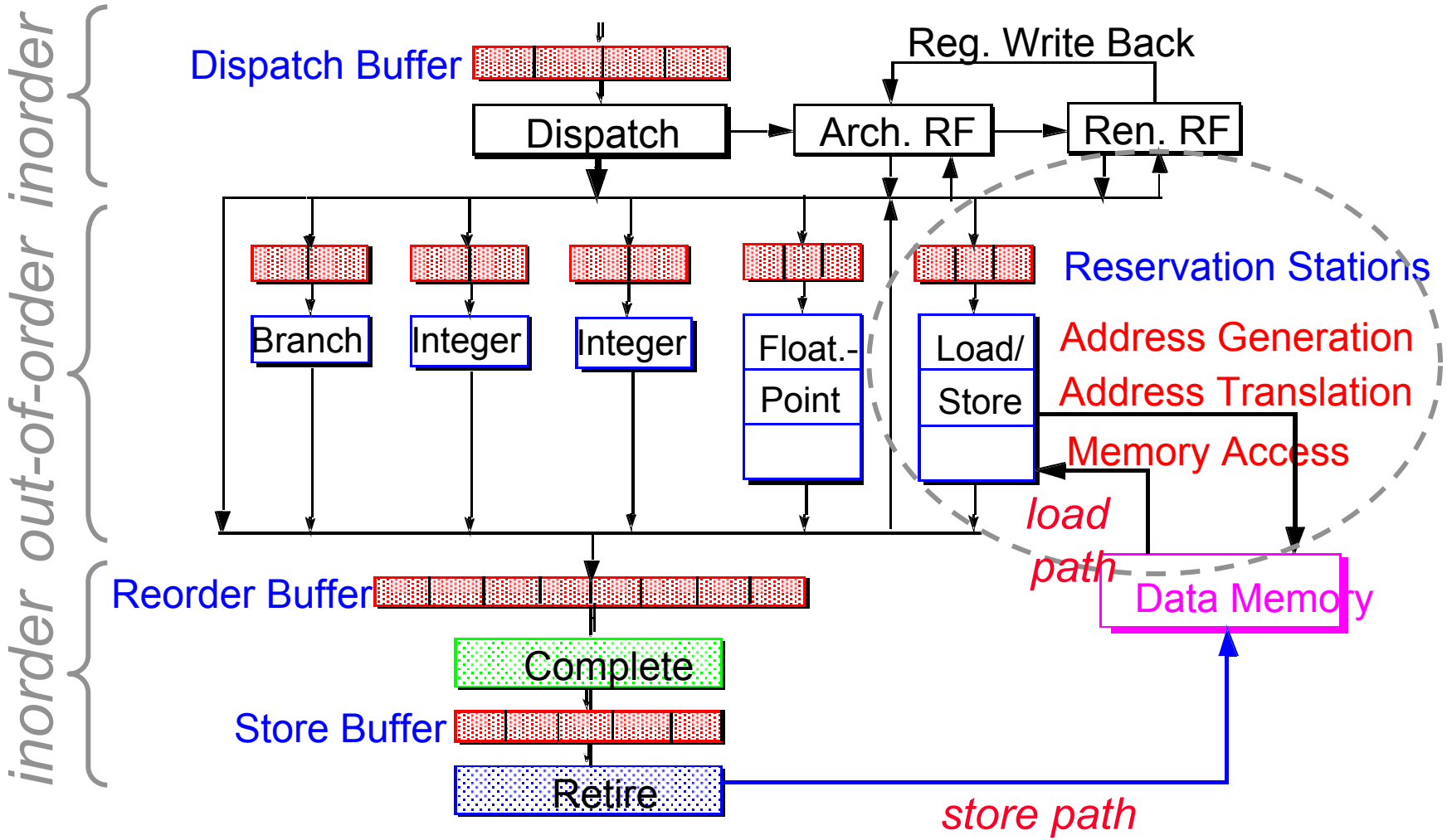
Dynamic Reordering of Memory Operations

- 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

- Allow out-of-order execution of Loads that do not have RAW memory-dependence
- What is hard about managing memory-dependence?
  - memory address are much wider than reg names
  - memory dependencies are not static
    - a load (or store) instruction's address can change
    - addresses need to be calculated and translated first
  - memory instructions take longer to execute relative to other instructions types

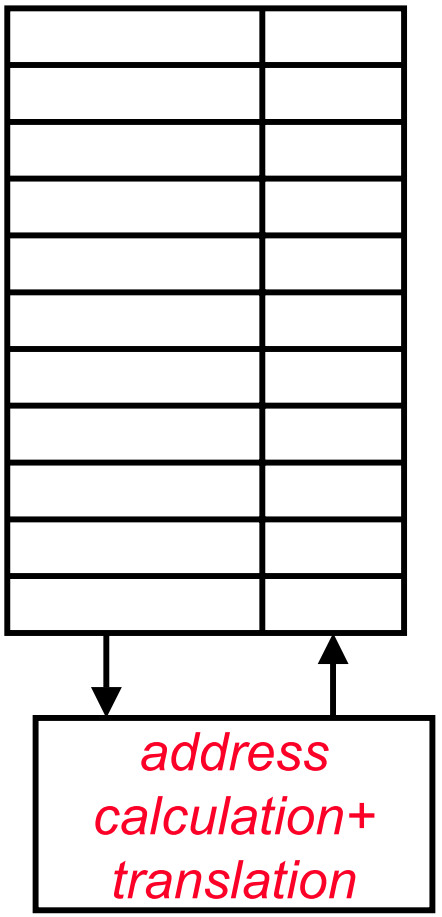
Processing of Load/Store Instructions



LD/ST Queue cannot follow simple register dataflow

Load/Store Queue

- Operates as a circular FIFO
- Loads and store instructions are stored in program order
  - allocate on dispatch
  - de-allocate on retirement
- Issue to address unit in register dataflow order
- A matrix records memory address dependence (also considers relative age of entries)
  - store ops are held until retirement
  - load ops are issued when no dependency exists (all older store addresses must be known)

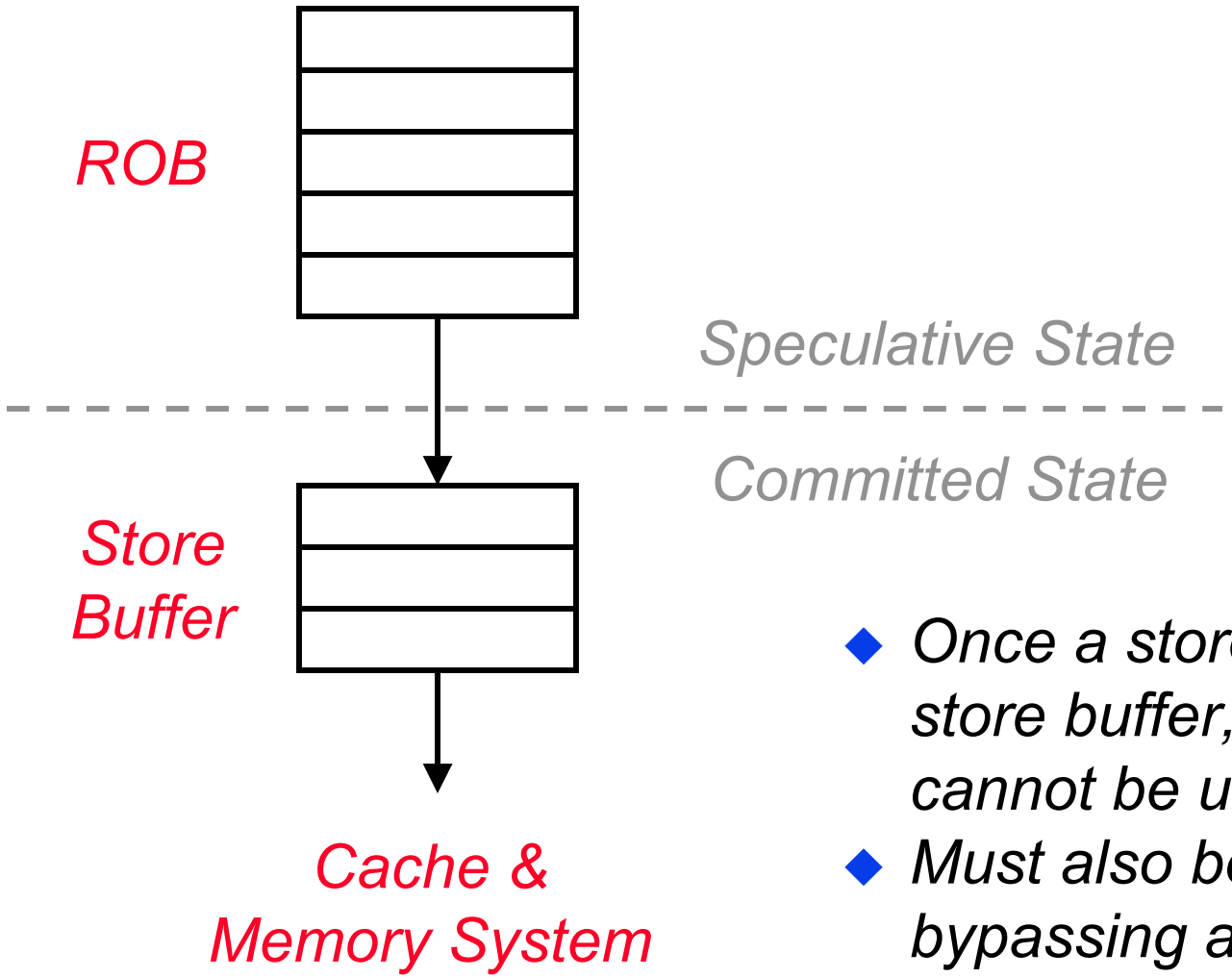


Load Bypassing

- 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
- Alternatively, a load can assume no aliasing and bypass older stores *speculatively*
  - validation of no aliasing with previous stores must be done and provide mechanism for reversing the effect
- Stores are kept in ROB and LD/Store address queue until all older instructions have completed
- At completion time, a store is moved to the Store Buffer to wait for turn to access cache

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

Store Buffer



- *Once a store enters the store buffer, its effect cannot be undone*
- *Must also be check by load bypassing and forwarding*

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
- Forward from which store?

$$Store_i(\ a,\ v\ ) << Store_k(\ a,\ v'\ ) << Load_j(a)$$

- 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
- This avoids the latency of accessing the data cache

Very important for x86 processors. Why?

Memory Ordering for Shared Memory Multiprocessors

- Consider these two programs running to two processors that communicate via shared memory

```
Proc A:                          Proc B:
    MEM[Y] is initially 1            ……
    ……                               do {
    compute V                                lock=Load Y
    Store (X, V)                     while (lock)
    Store (Y, 0)                     data = Load X
    ……                               ……
```

- Can the order of Loads and Stores be swapped during dynamic execution?

Much more to come on this later!!