Superscalar Design

Memory Data Flow

Virendra Singh

Associate Professor

Computer Architecture and Dependable Systems Lab

Department of Electrical Engineering

Indian Institute of Technology Bombay

http://www.ee.iitb.ac.in/~viren/

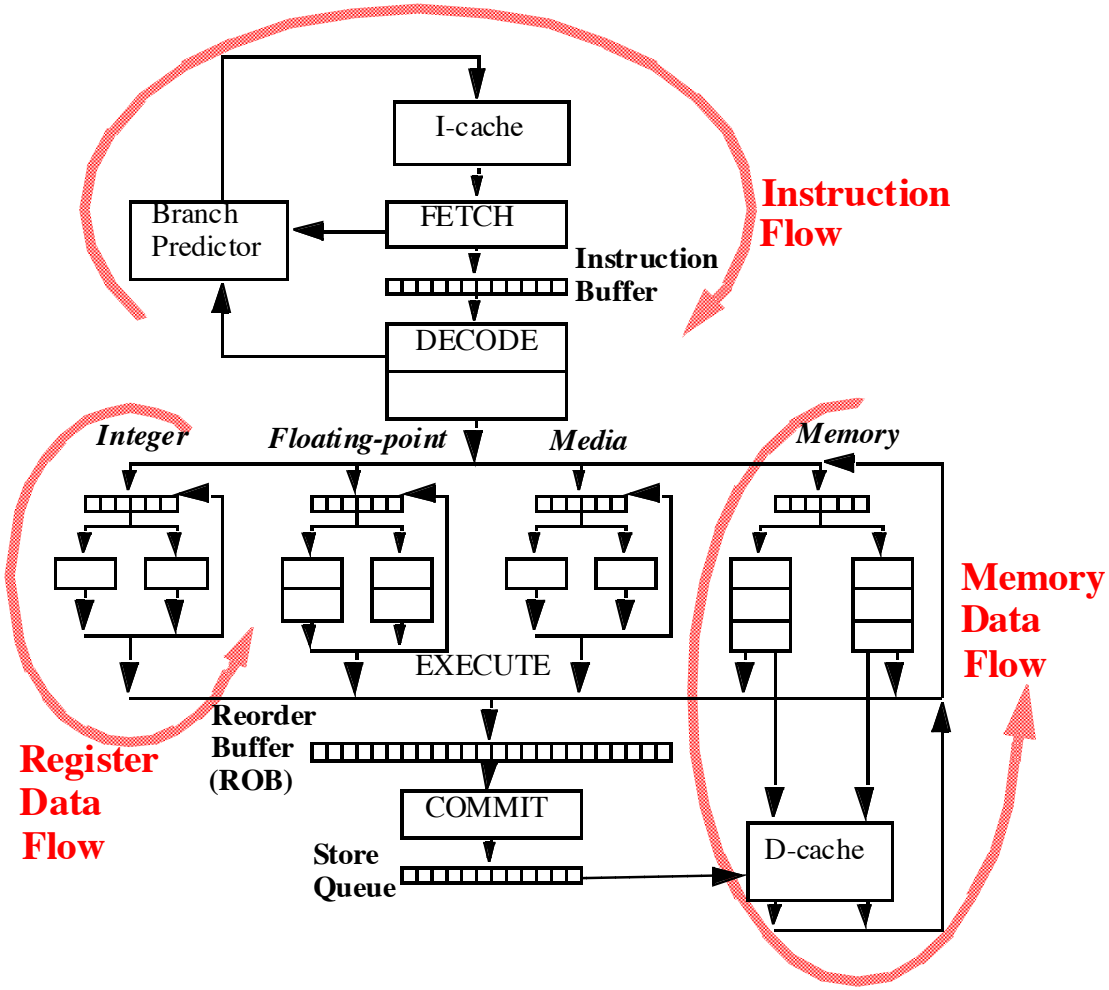
E-mail: viren@ee.iitb.ac.in

EE-739: Processor Design

Lecture 7 (04 Feb 2015)

CADSL

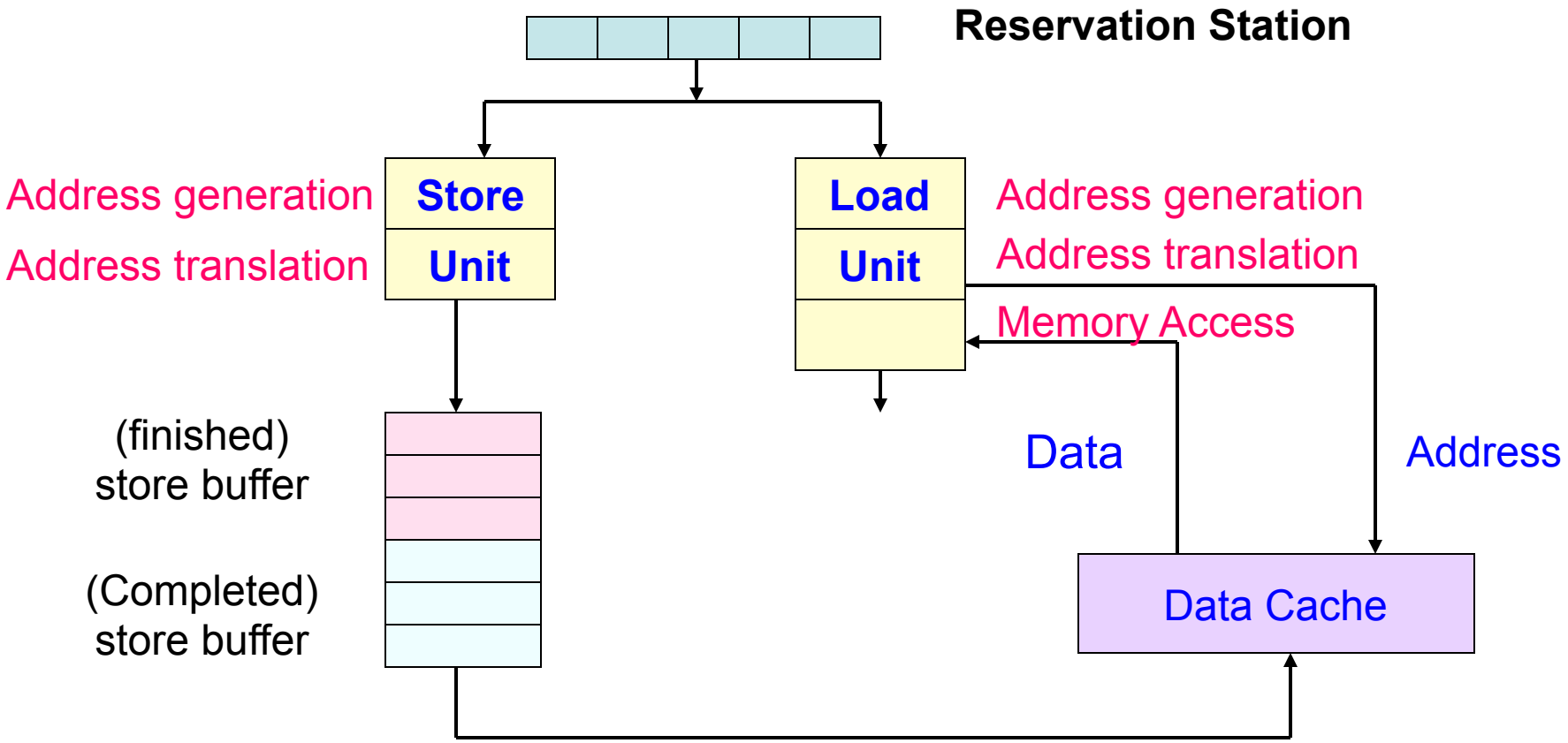
Impediments to High IPC



Memory Data Flow Techniques

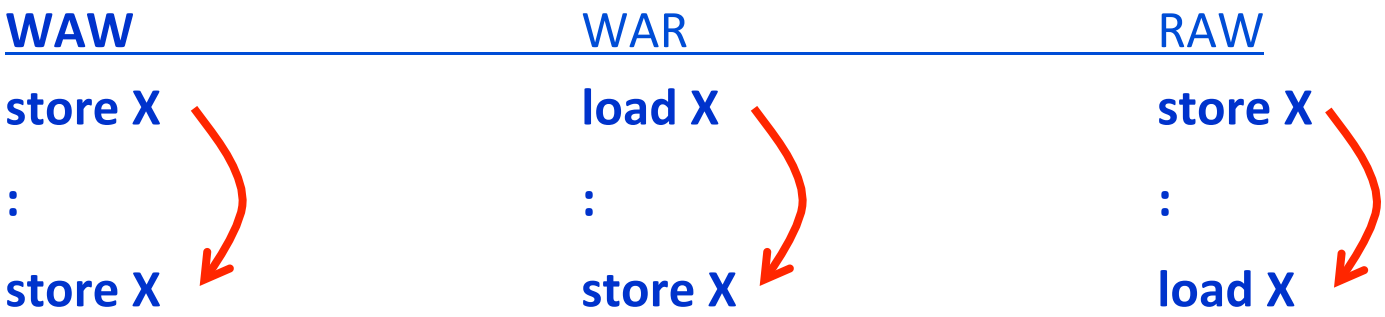
- Move data between memory and RF
- Long Latency
- Bottleneck
- Operations
  - Address Generation
  - Address Translation
  - Read/write data

Load/Store Processing



Memory Data Dependences

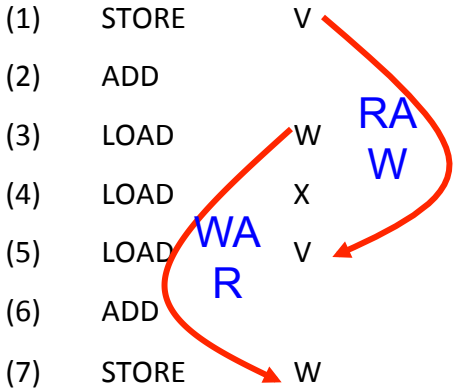
- Besides branches, long memory latencies are one of the biggest performance challenges today.
- To preserve sequential (in-order) state in the data caches and external memory (so that recovery from exceptions is possible) stores are performed in order. This takes care of anti-dependences and output-dependences 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 Dependences

- Memory Aliasing = Two memory references involving the same memory location (collision of two memory addresses).
- Memory Disambiguation = Determining 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

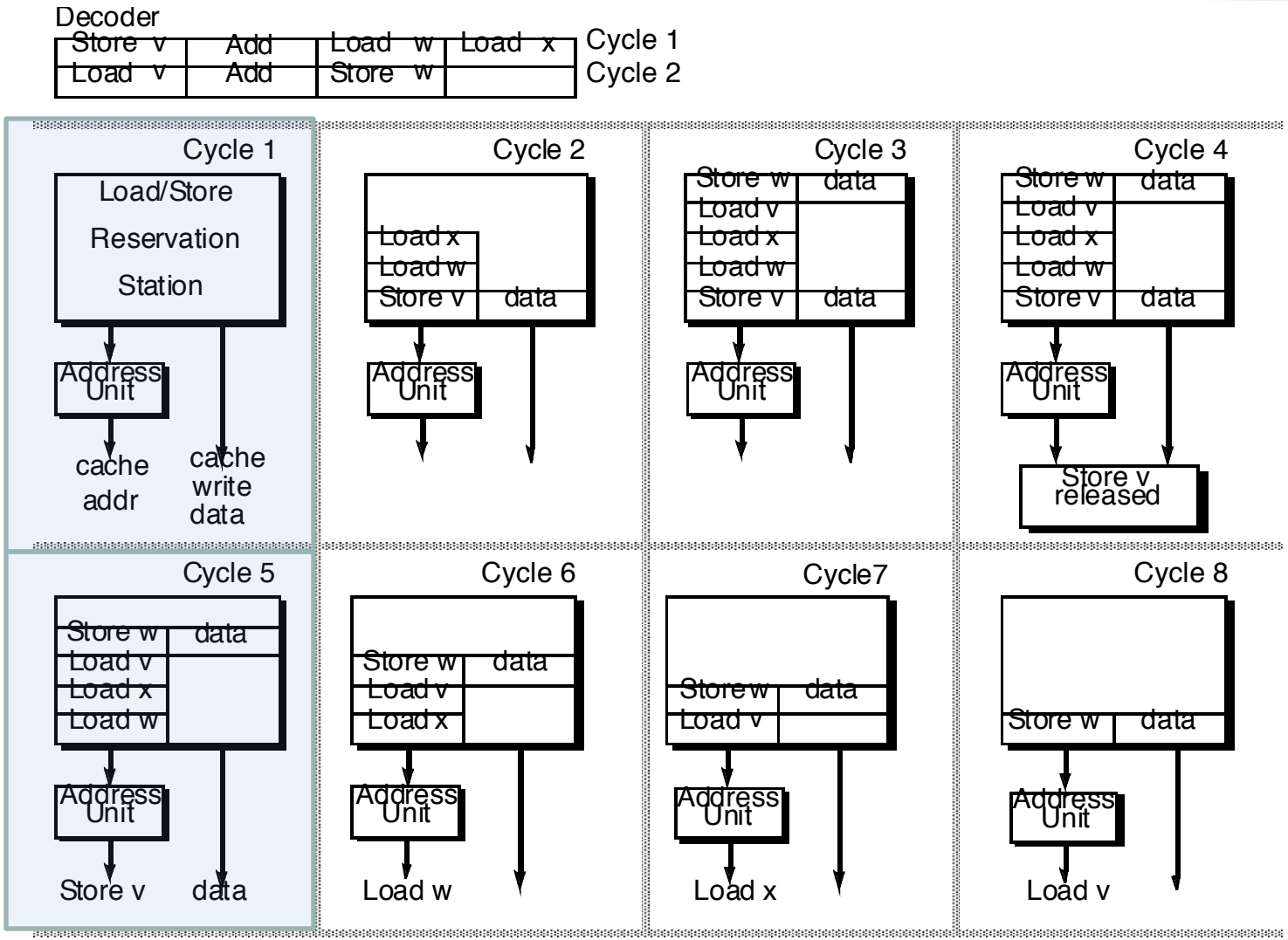
Example code:



Total Order 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.
- 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 Order



ISSUING LOADS AND STORES WITH TOTAL ORDERING

Load Bypassing

- Loads can be allowed to bypass stores (if no aliasing).
- Two separate reservation stations and address generation units are employed for loads and stores.
- Store addresses still need to be computed before loads can be issued to allow checking for 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).
- Stores are kept in ROB until all previous instructions complete; and kept in the store buffer until gaining access to cache port.
  - Store buffer is "future file" for memory

Load Bypassing

Dynamic instruction sequence

Execute load ahead of the two stores

Load Bypassing

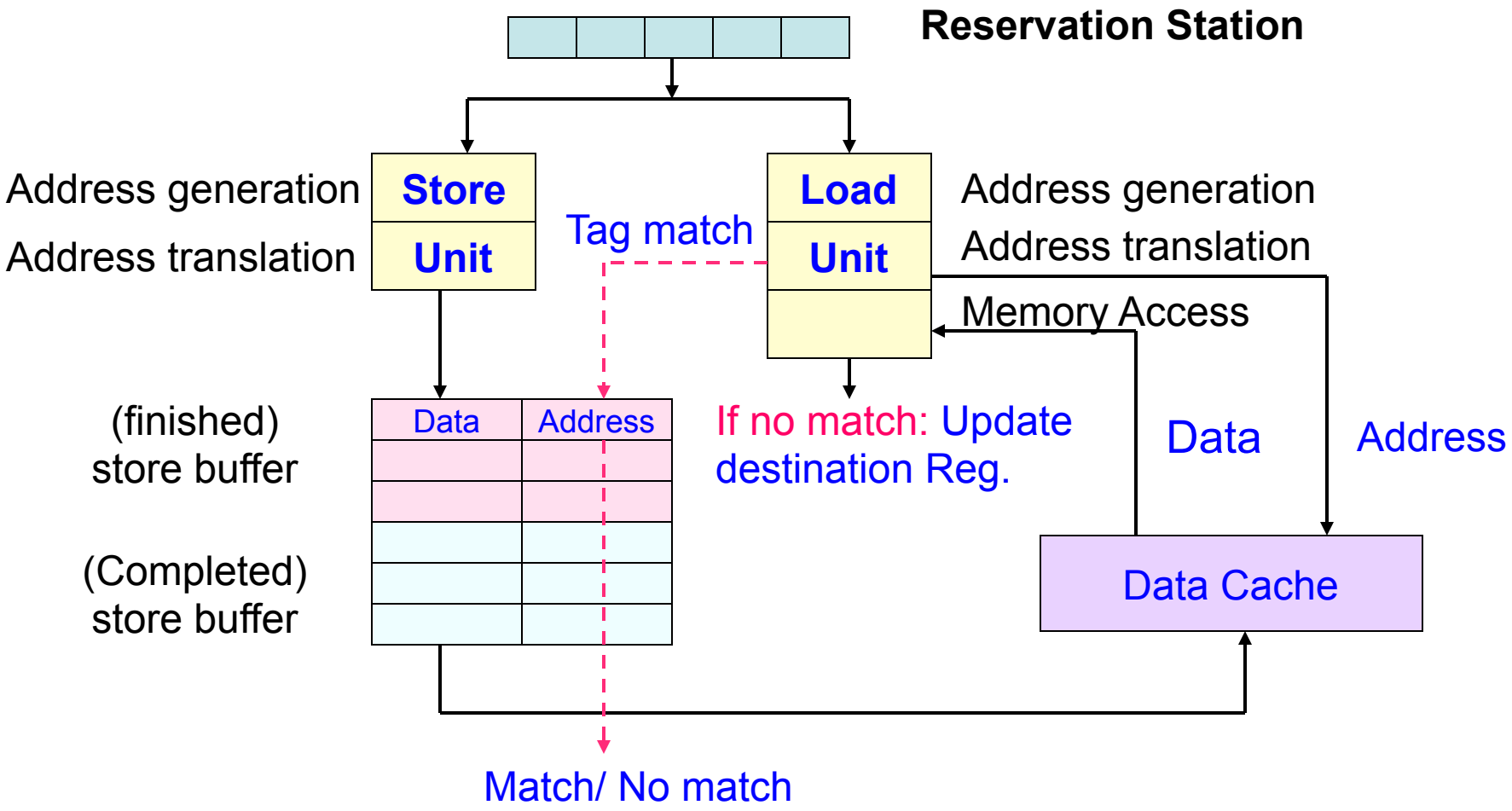
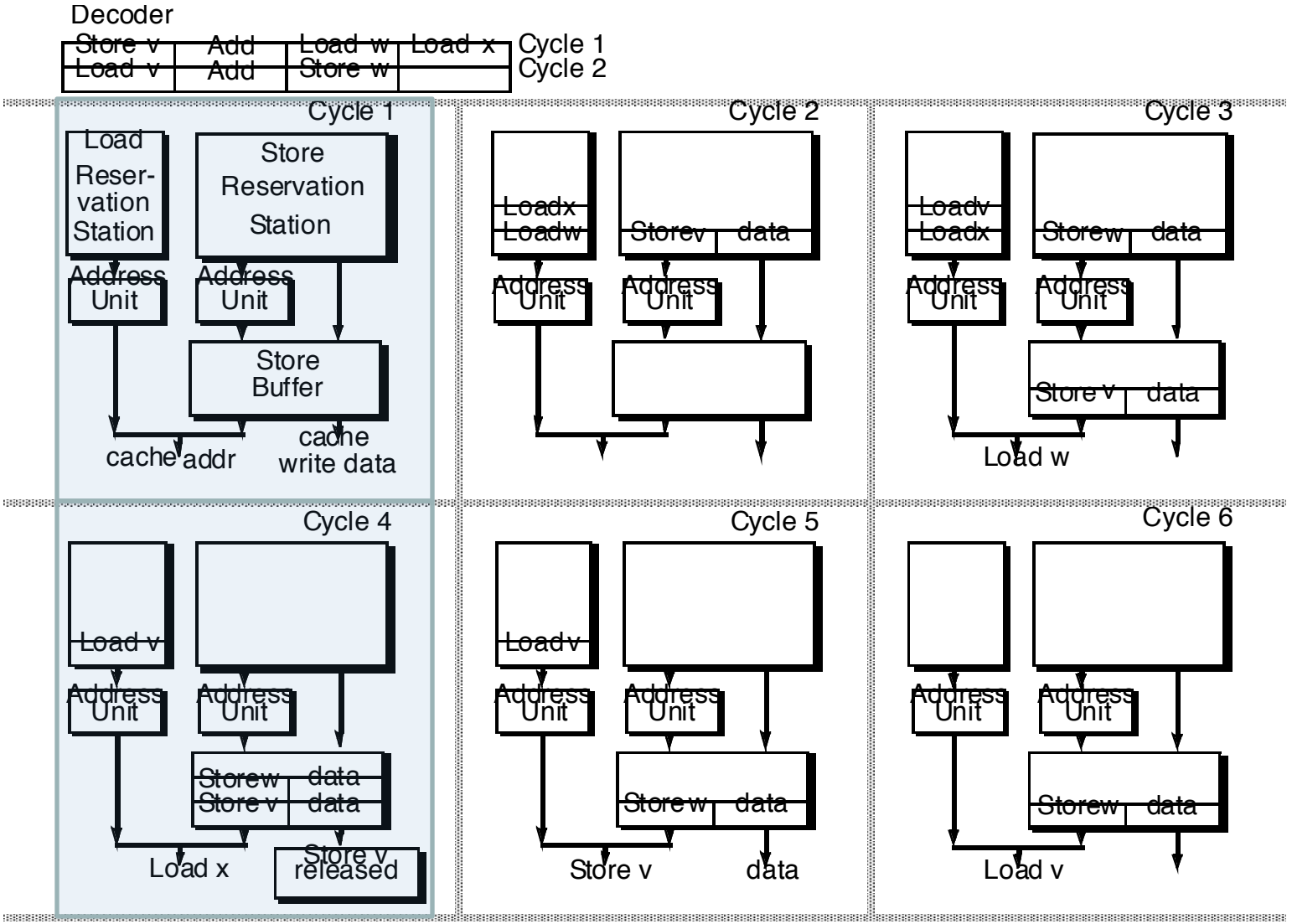


Illustration of Load Bypassing



LOAD BYPASSING OF STORES

Load Forwarding

- If a subsequent load has a dependence on a 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 the address is valid and the data is available in the store buffer.
- Since data is sourced from the store buffer:
  - Could avoid accessing the cache to reduce power/latency

Load Forwarding

Dynamic instruction sequence

. . . .

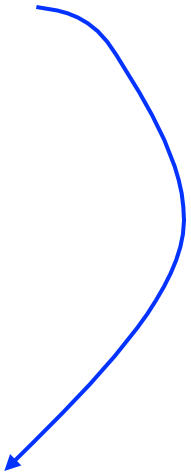
Store X

. . . . .

Store Y

. . . .

Load X



Forward Store data directly to the Load

Load Forwarding

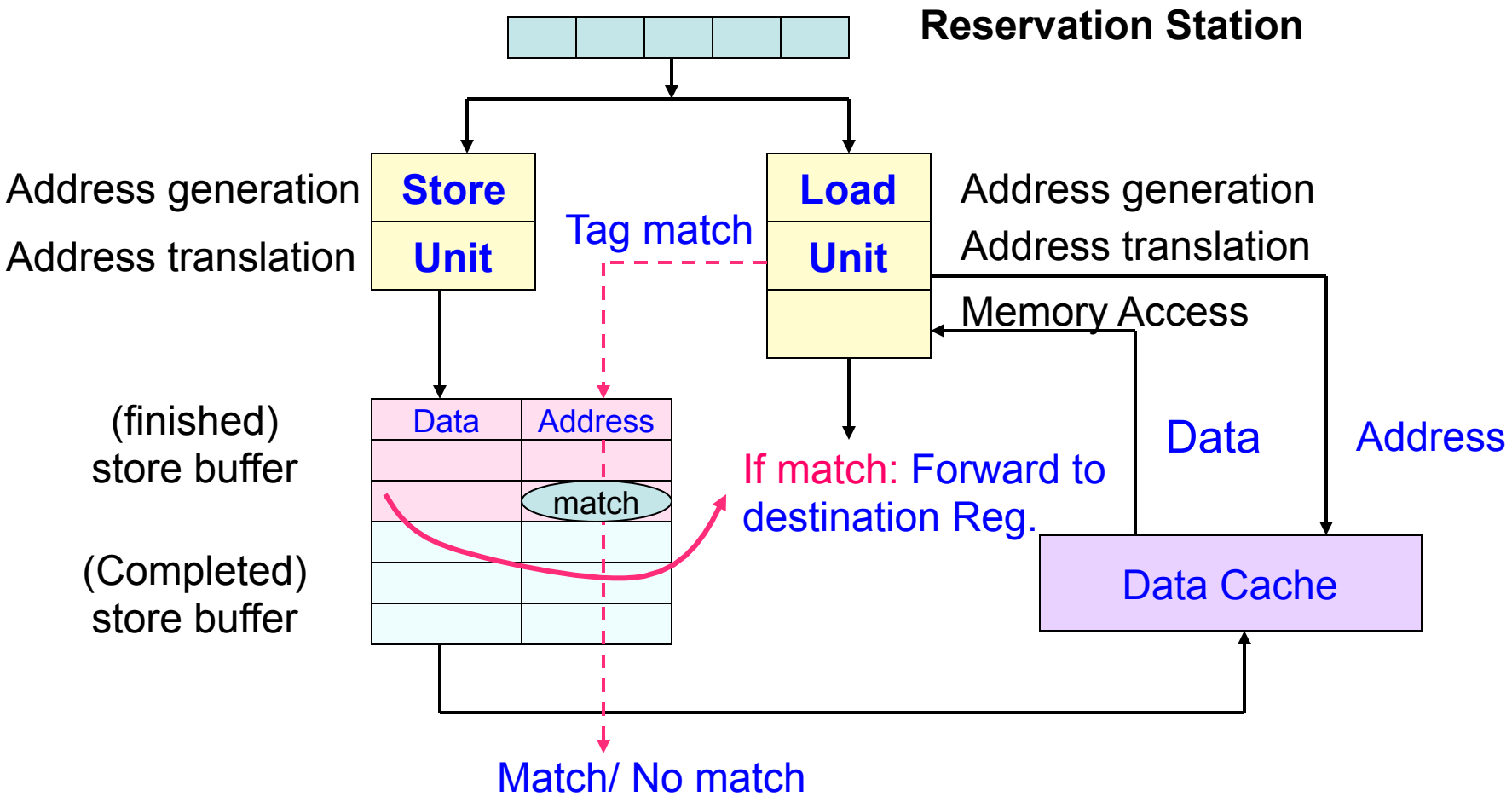
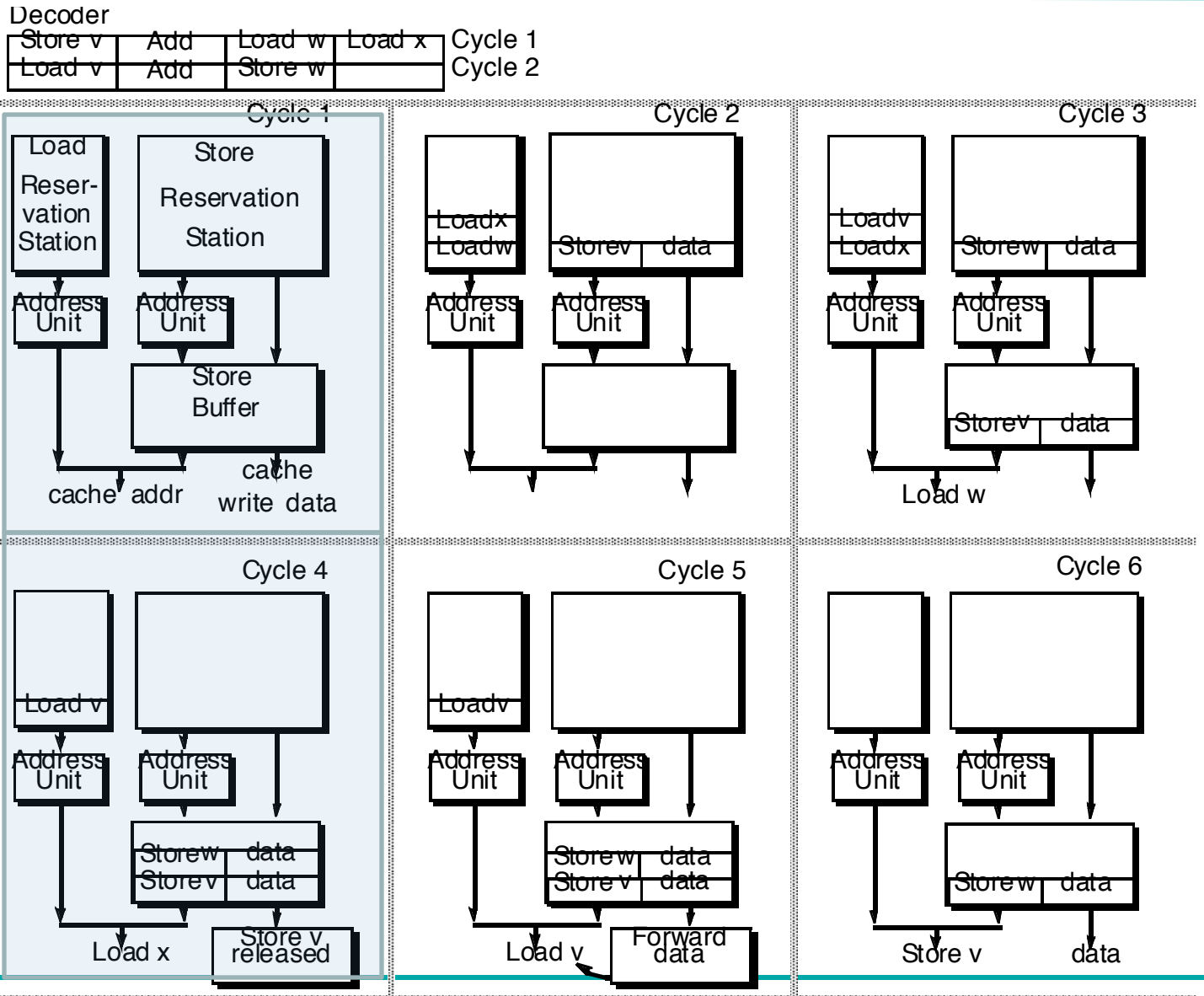


Illustration of Load Forwarding

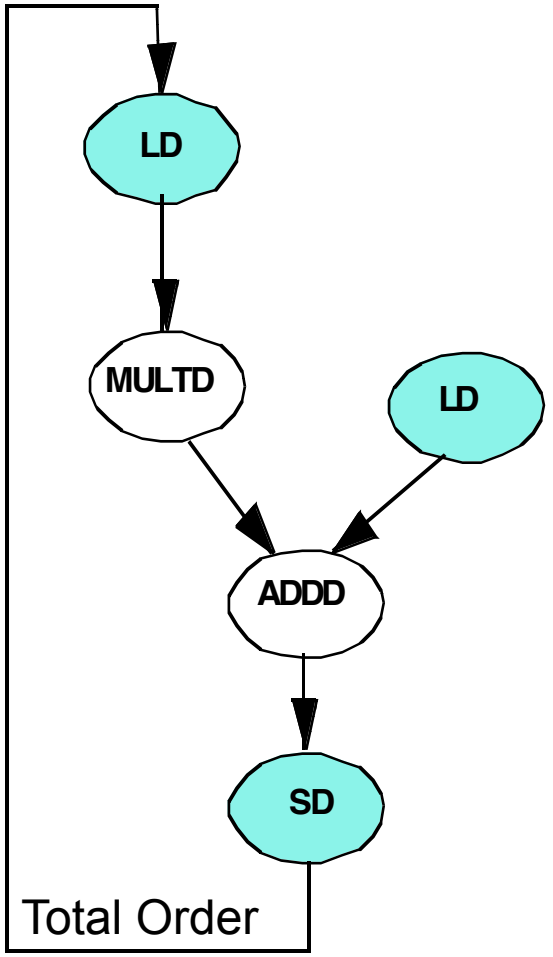


The DAXPY Example

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

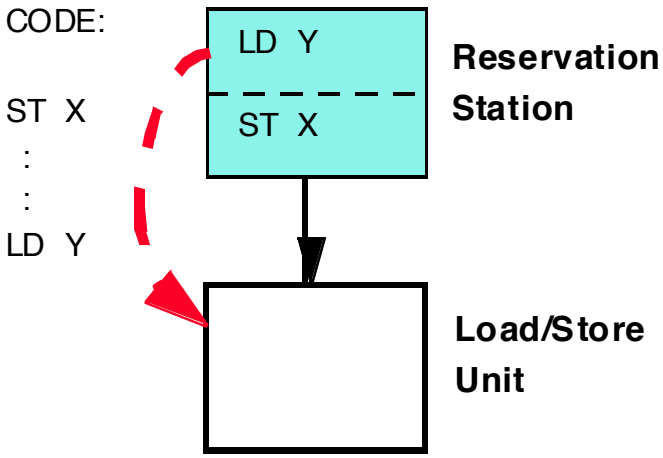
```
        LD      F0, a
        ADDI    R4, Rx, #512        ; last address

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
```

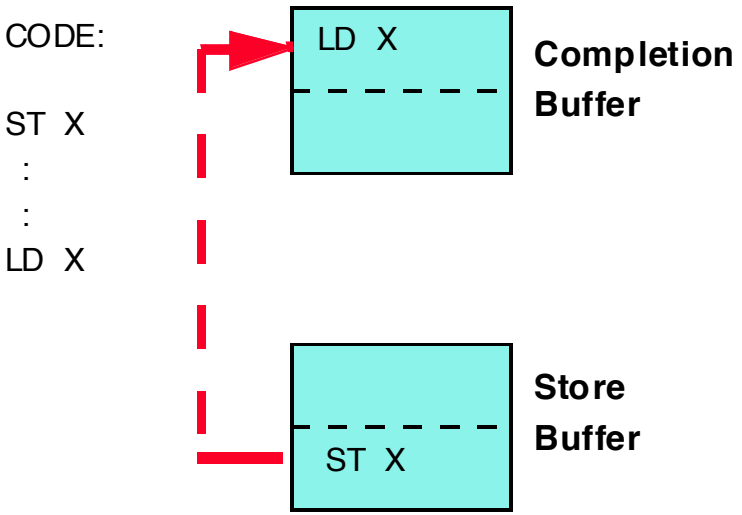


Performance Gains From Weak Ordering

Load Bypassing:



Load Forwarding:



Performance gain:

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

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

Thank You