Computer Architecture

Spring 2016

Lecture 14: Speculation II

Shuai Wang

Department of Computer Science and Technology

Nanjing University

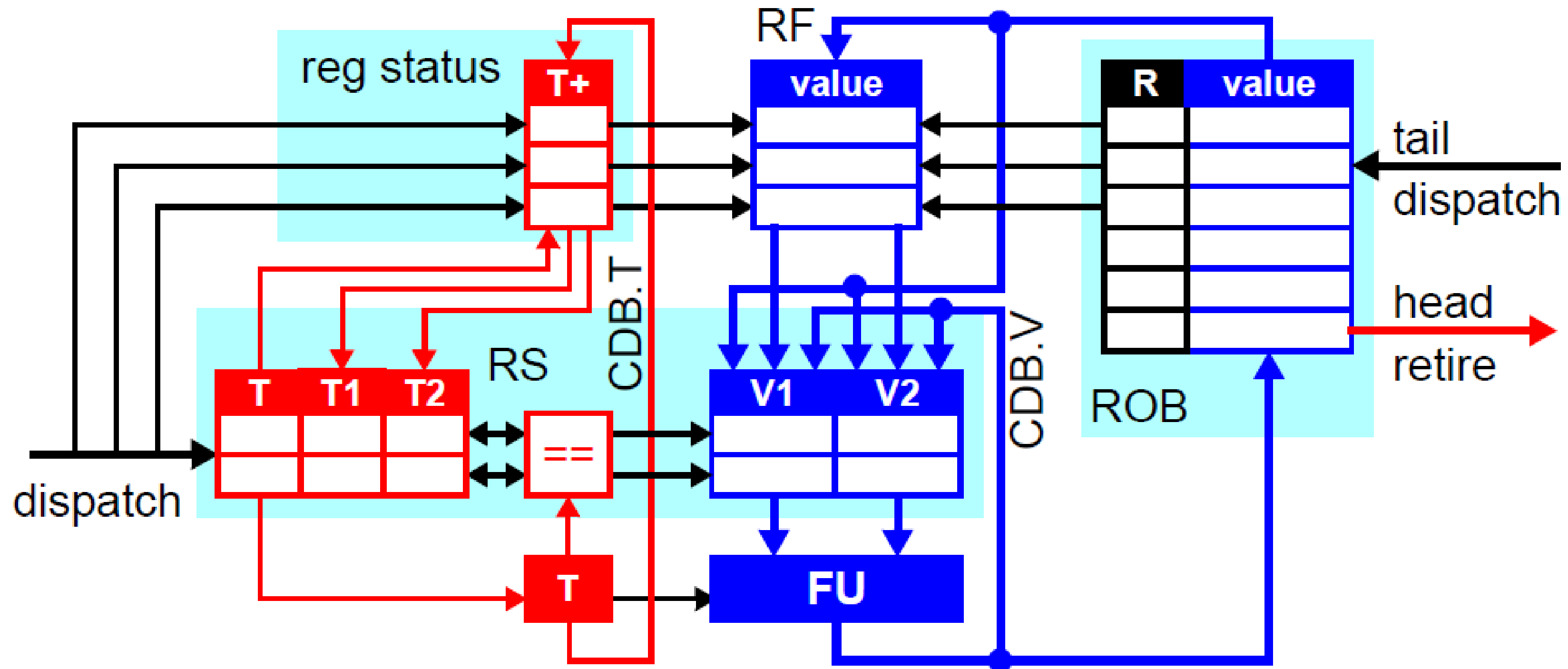
[Slides adapted from CS 246, Harvard University]

Tomasulo+ROB

- Add ROB to Tomasulo's algorithm
  - combined ROB and RS are called RUU (or Sohi's method)
    - RUU – register update unit
  - separate ROB and RS are called P6-style (Intel P6 = Pentium Pro)

- Our example: Simple-P6
  - separate ROB and RS
  - same RS organization as before: 1 ALU, 1 load, 1 store, 2 3-cycle FP

P-6 Style Organization

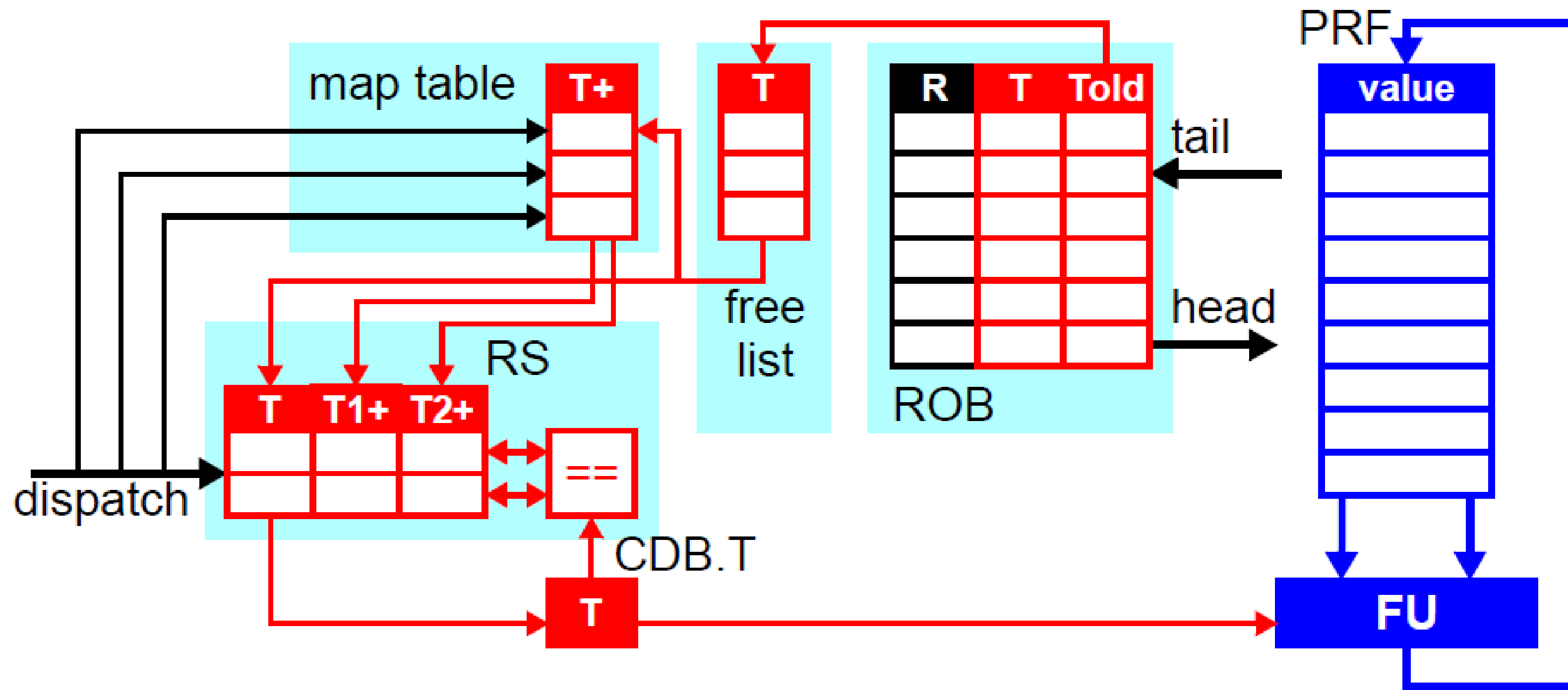


- instruction fields and ready bits
- tags
- values

Problems with P6

- Problems for high performance implementations
  - too much value movement (regfile/ROB→RS → ROB → regfile)
  - too many muxes/buses (values from too many places)
  - RS mixes values with control tags (long data paths make clock slow)

Alternative Implementation: MIPS R10K



- separate control (ROB/RS) from data (registers/FU)
  - one big physical register file (PRF) holds all data → no copies
  - ROB and RS used only for control and tags → small
  - register file close to FUs, everything else is on the side

R10K Register Renaming

- no architectural register file!
- physical register file holds all values
  - #physical registers > #architectural registers
  - map architectural registers to physical registers
  - removes WAW&WAR hazards (physical registers replace RS copies)
- register status table replaced by register map table
  - mappings cannot be 0 (there is no architectural register file)
- *free list* keeps track of unallocated physical registers
  - ROB responsible for returning physical registers to free list
- conceptually: true register renaming
  - have seen an example a few lectures ago ... here it is again ...

Freeing Registers in R10K

- freeing physical registers
  - P6
    - no need to free speculative storage explicitly
    - temporary storage comes with ROB entry
    - copy value from ROB to register file, free ROB entry
  - R10K
    - can't free physical register when writing instruction retires
    - no architectural register to copy value to
    - but…
    - we can free physical register previously mapped to same logical register
    - why? All instructions that will ever read that value have retired

Freeing Registers in R10K

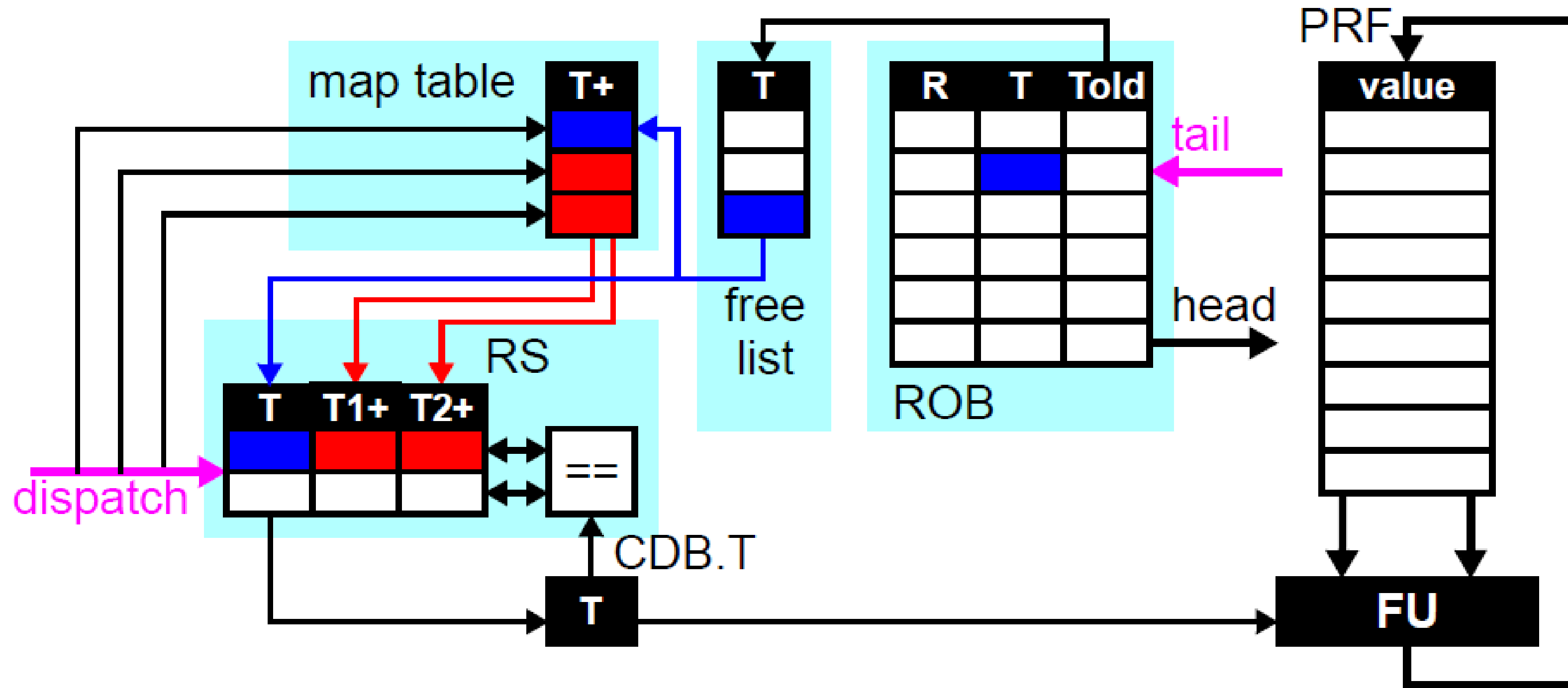
raw instruction	map table r1	r2	r3	free locations	renamed instruction
	l1	l2	l3	l4,l5,l6,l7	
add r1,r2,r3	l4	l2	l3	l5,l6,l7	add l4,l2,l3
sub r3,r2,r1	l4	l2	l5	l6,l7	sub l5,l2,l4
mul r1,r2,r3	l6	l2	l5	l7	mul l6,l2,l5
div r2,r1,r3	l6	l7	l5		div l7,l6,l5

- when **add** commits: free **l1**
- when **sub** commits: free **l3**
- when **mul** commits: free **?**
- when **div** commits: free ?
- see the pattern?

R10K Pipeline

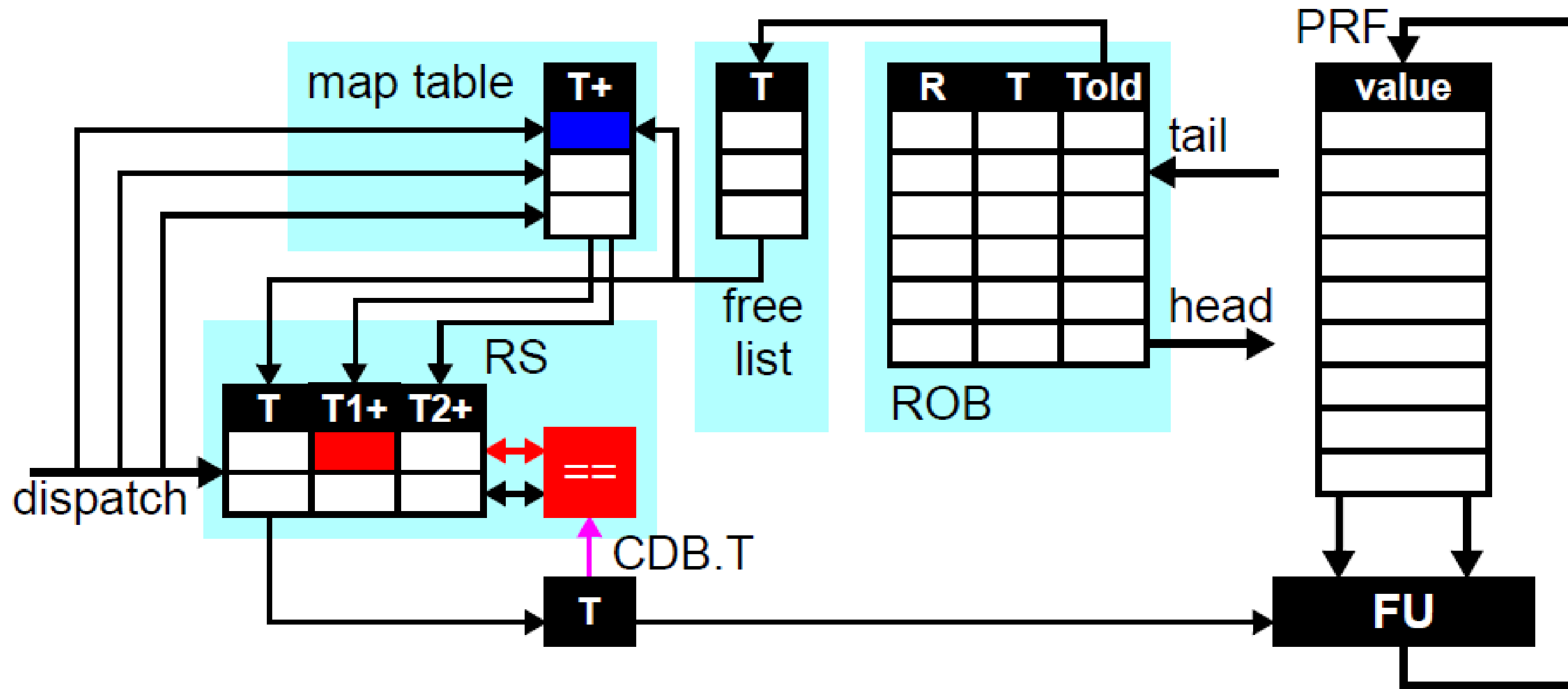
- same pipeline structure: IF,*DS*,IS,EX,*CM*,*RT*
  - *DS (dispatch)*
    - (RS or ROB full or no free physical registers) ? (stall) :
    - (allocate RS and ROB entries AND physical register)
  - *IS (issue)*
    - (read physical registers)
  - *CM (completion)*
    - (writeback destination register, mark ROB entry complete)
  - *RT (retire, commit, graduate)*
    - (ROB head not complete) ? (stall) :
    - free ROB entries, free previous physical register

R10K: Dispatch (DS)



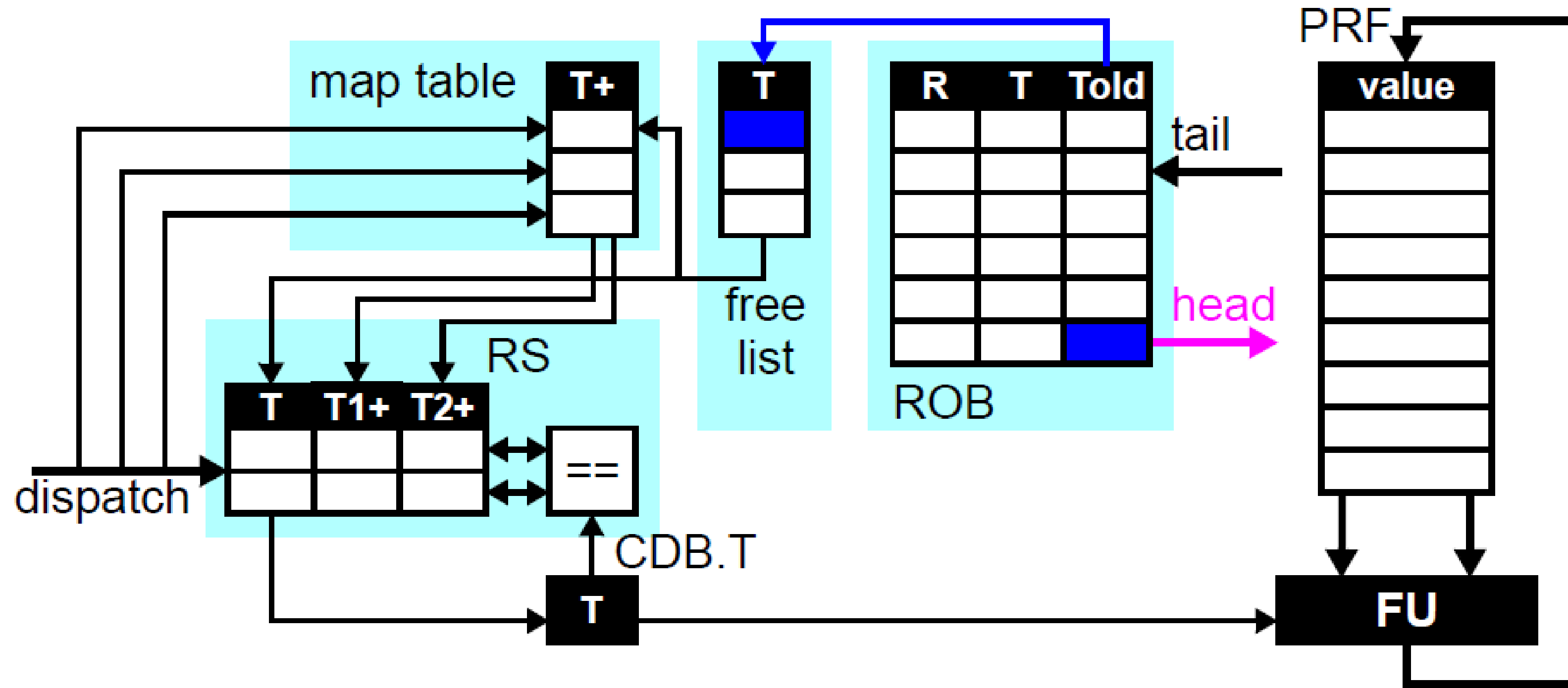
- stall if no free RS, ROB, or physical register (preg)
  - allocate RS and ROB entry
  - read physical register tags for input registers, store in RS
  - allocate new physical register for output, set in RS, ROB and map table

R10K: Complete (CM)



- wait for free CDB
  - broadcast tag on CDB.T
  - set instruction's output register ready bit in map table
  - set ready bits for matching input tags in RS

R10K: Retire (RT)



- stall until instruction at ROB head is complete
  - return Told of ROB head to free list
  - free ROB head entry

R10K Precise State

- problem with R10K way? precise state is more difficult
  - registers already written
    - but that's OK
    - why? because there is no architectural register file
    - we can "free" written registers and restore old ones
  - we need to restore register map table to the way it was
    - option 1: roll back ROB serially (slow)
    - option 2: restore from a checkpoint (fast)

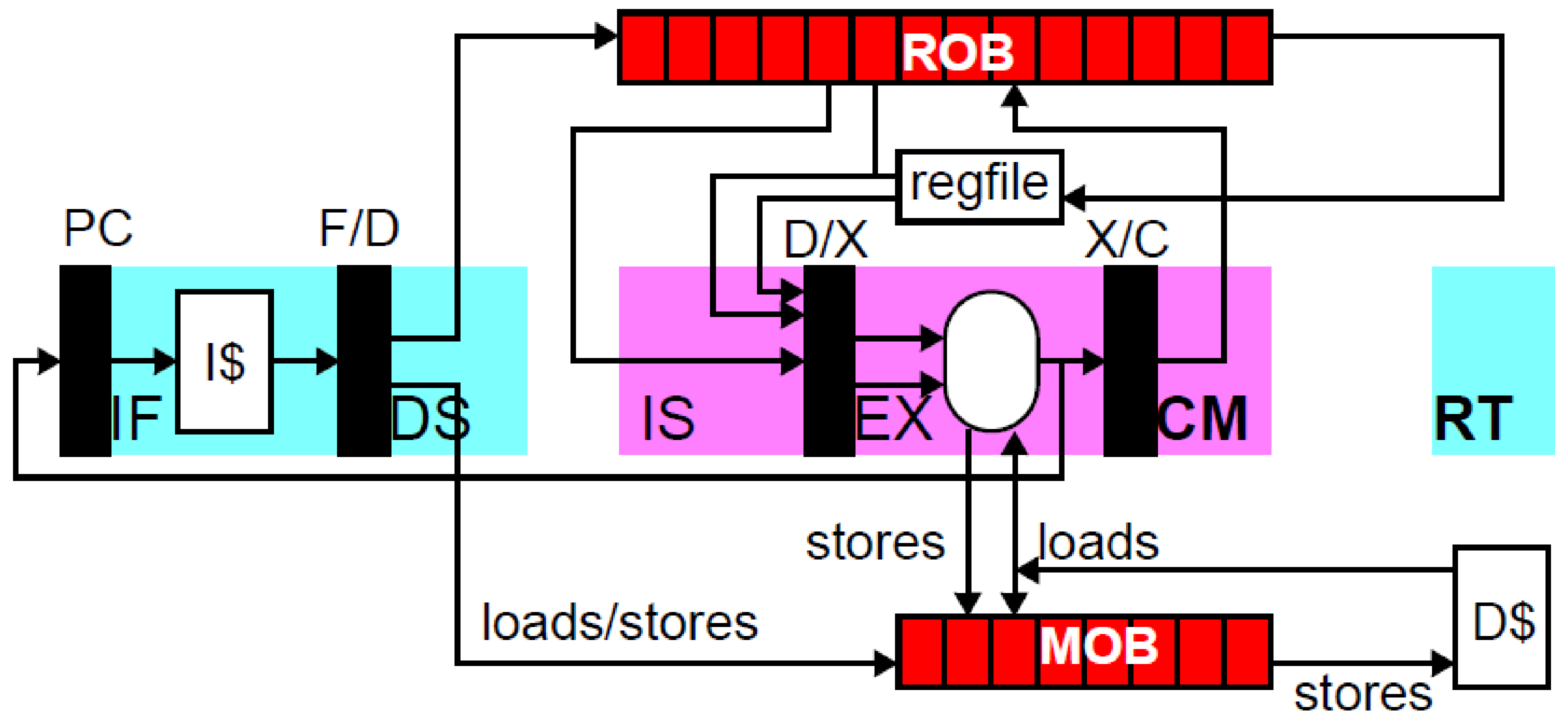
P6 vs. R10K

feature	P6	R10K
value storage	RF, ROB, RS	PRF
register read	DS: RF/ROB → RS	IS:PRF → FU
register write	RT:ROB →RF	CM:FU →PRF
spec value free	RT: automatic with ROB	When overwriting instruction retires
datapaths	RF/ROB →RS, RS →FU, FU →ROB, ROB →RF	PRF →FU, FU →PRF
precise state	Simple, zero all structures	Complex: serial/checkpoint

Memory Ordering Buffer (MOB)

- ROB makes register write in-order, but what about stores?
  - same as before (i.e., to D$ in MEM stage)?
    - bad idea! Imprecise memory worse than imprecise registers
    - most do same trick for stores
- *Memory Ordering Buffer (MOB)*
  - a.k.a store buffer, store queue, load/store queue (LSQ)
  - completed (but not retired) stores write to MOB
  - to retire store, write head of MOB to D$
  - loads look at MOB and D$ in parallel
    - Forward from MOB if matching store (i.e. to same address)

MOB + ROB

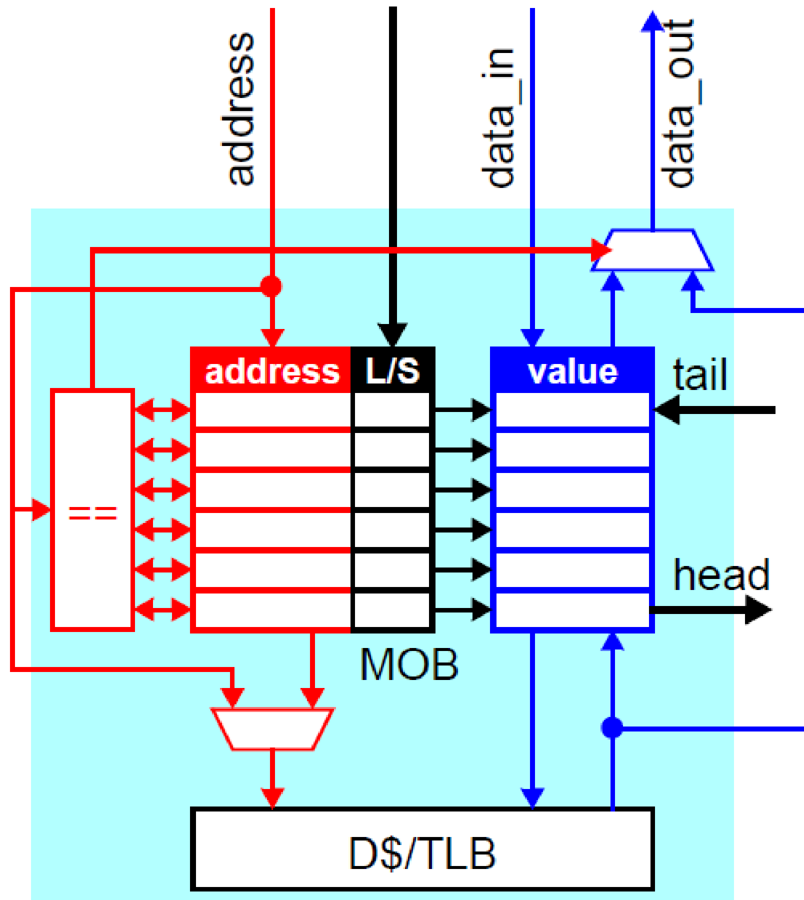


Advanced Topic: Load Scheduling

- all instructions except for loads are easy in Tomasulo
  - register inputs only
  - register renaming captures all dependences
  - tags tell you exactly when you can execute

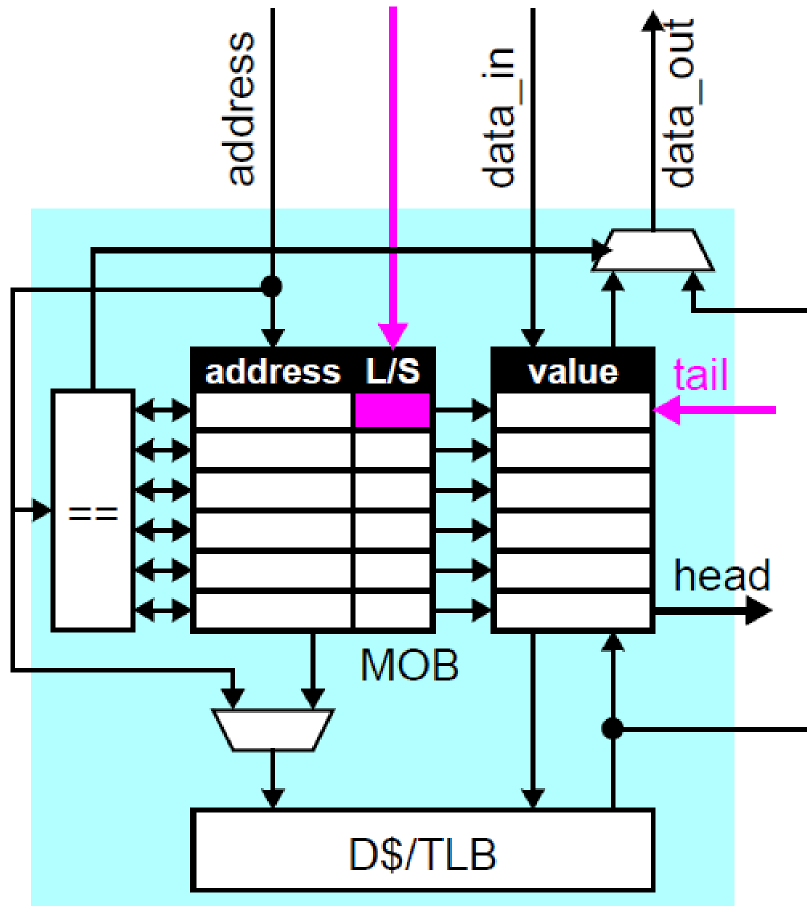
- loads not so easy
  - must check for older active stores with same address
  - Register renaming doesn’t tell you that

The data memory FU



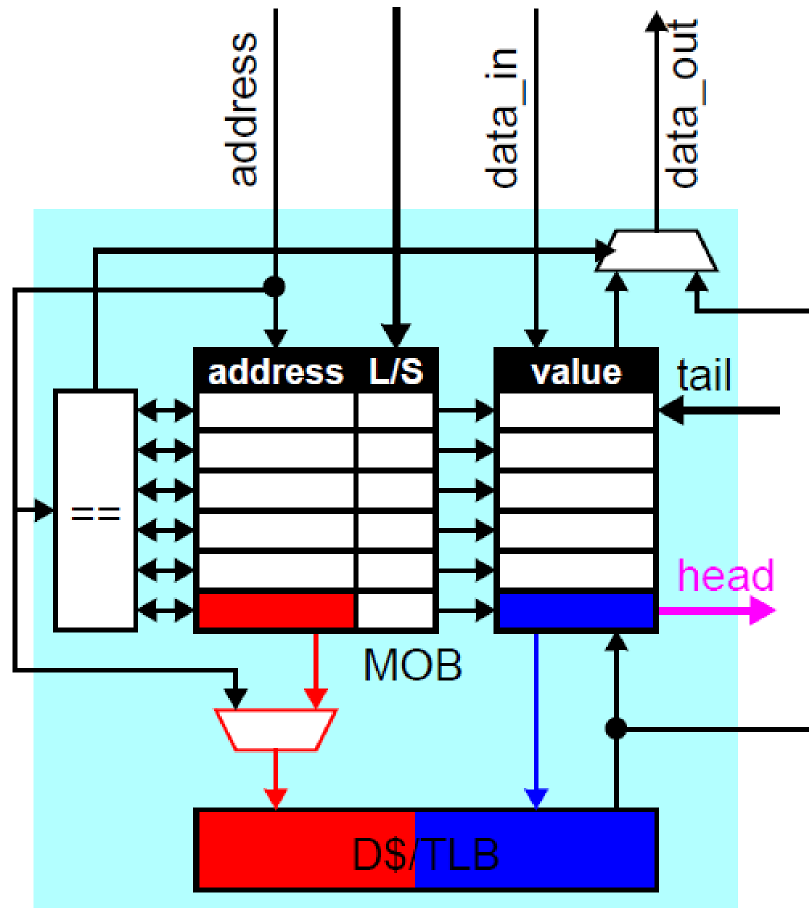
- MOB+D$ = memory FU
  - just like any other FU
  - 2 reg inputs: addr, data_in
  - 1 reg output: data_out
- what actually happens?

Store/Load Dispatch



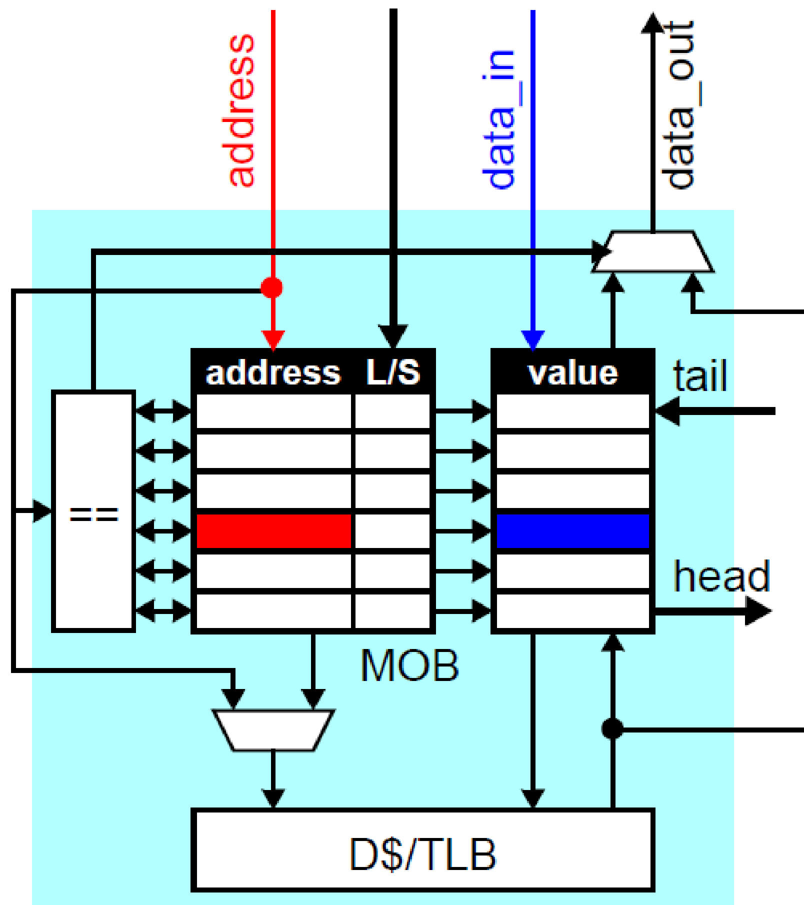
- allocate MOB entry (tail)
  - indicate store/load
  - remember MOB#

Store/Load Retire



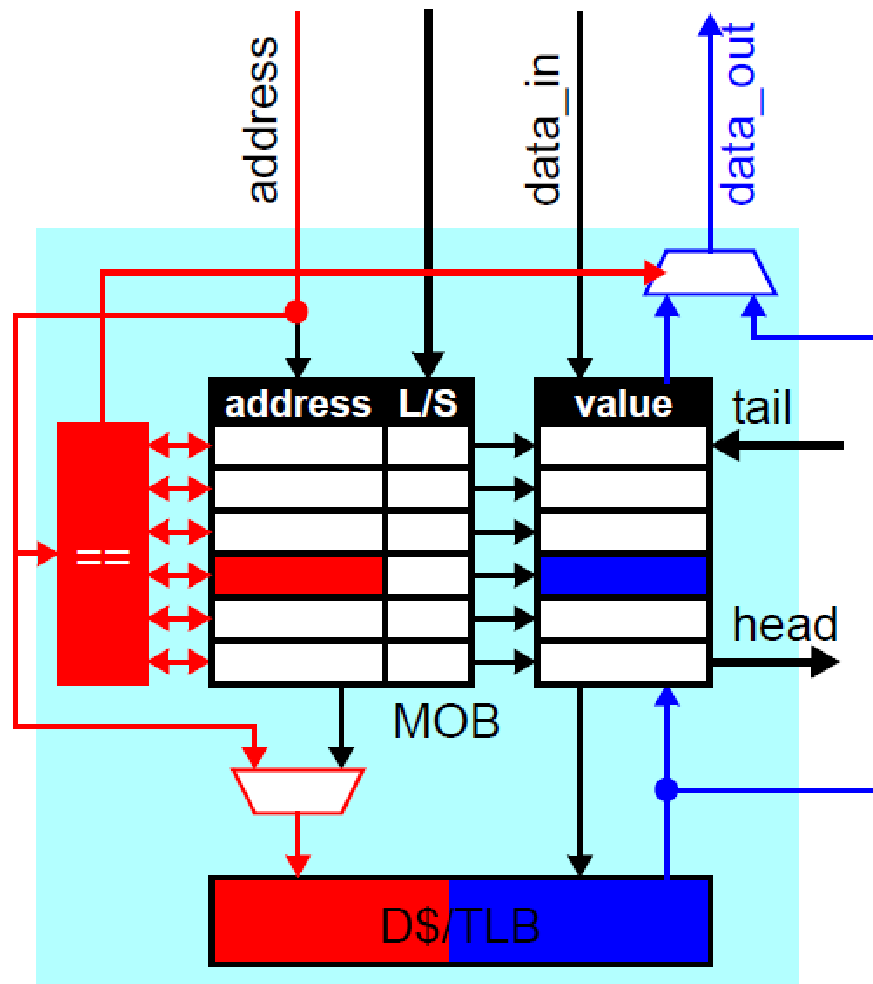
- free MOB entry (head)
- load?
  - done
- store?
  - address + value to D$/TLB

Store Execute



- address + value to MOB
  - can be done separately
  - e.g., Pentium II: store →2 µ ops

Load Execute



in parallel

- address to D$
  - read value
- address to MOB
  - compare to older store addrs

no match ⇒ D$ value

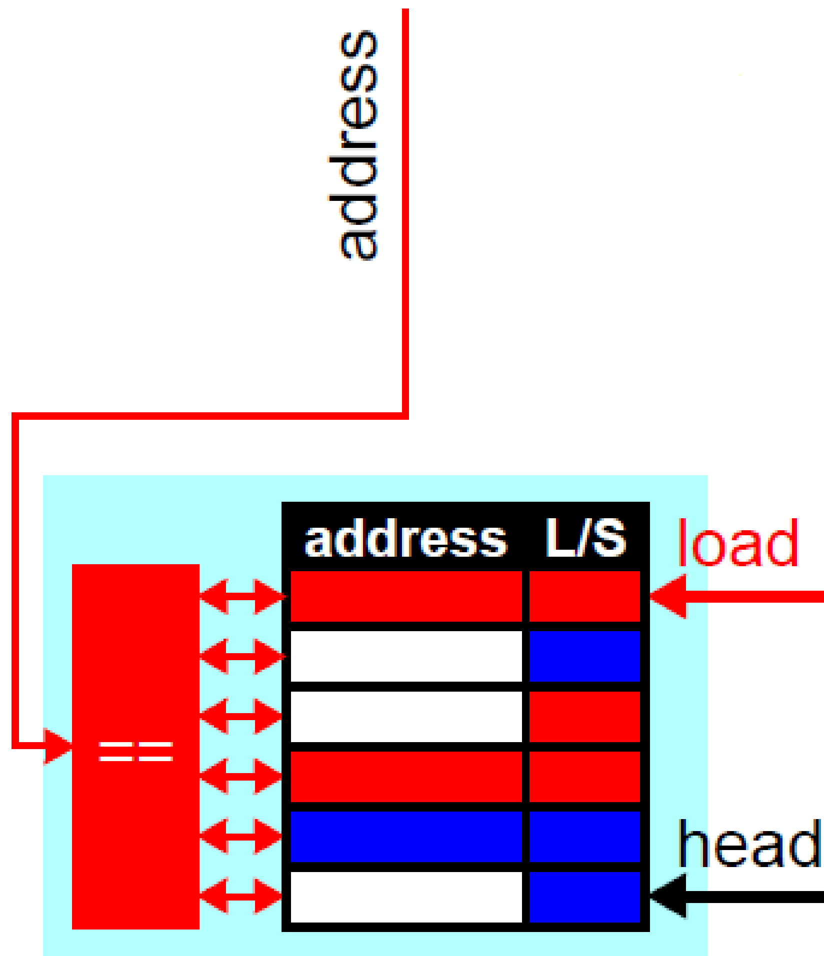
match +value ⇒ MOB value

- multiple matches? youngest
- forwarding or bypassing
- same latency as D$ hit (why?)

match –value ⇒ stall

- try executing load again later

Memory Disambiguation Problem



- at load execution
  - what if older store address unknown?
    - how to determine match?
    - can't determine, have to guess
    - called "memory disambiguation"
- what if older load address unknown?
  - who cares

Memory Disambiguation Alternatives

- conservative: loads in-order with respect to stores
  - don’t know address? assume match, wait
  - pretty simple
  - many unnecessary waits on non-matching stores
- opportunistic: out-of-order loads
  - don’t know address? assume no match, go
  - higher performance (most cases are not matching)
  - mis-speculations: went too soon? recover (complex + expensive!)
- selective: combination of conservative and opportunistic
  - start out opportunistic
  - load mis-speculation? remember PC in table, next time conservative
  - pretty accurate prediction $\Rightarrow$ pretty good performance