Superscalar* Club Meeting #4

*we really mean: superscalar speculative out-of-order

James C. Hoe
Department of ECE
Carnegie Mellon University

Goal set when we started

- Achieve an RTL-precise understanding of superscalar speculative out-of-order register dataflow—as with 5-stage pipeline in 18-447
  - know you could make a working design and know what it does (how good is it?)
  - know shortcomings and limits in the simplifications to help get started
  - know what you have not figured out yet

Test: Able to read the R10K paper and say definitely what you understand and what you don't

Plan of Attack

- Focus
  - mainly on register dataflow
  - lightly on memory dataflow
  - not at all on i-fetch (a well decoupled subject both conceptually and physically)
- Path
  - further develop concepts in L20 we didn't have time for
  - study Metaflow DRIS to flesh out conceptual-level understanding
  - study how things were really done in R10K
  - play with an RTL-precise executable model (in C++)

Review: Tomasulo with Physical Register File Rename

Problem Setup

I0: r4 ← r4 + r6
I1: r2 ← r2 + r6
I2: r4 ← r2 * r4
I3: r4 ← r4 + r6
I4: r2 ← r2 + r6
I5: r2 ← r2 + r4

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
Add: stage 1 ( )					

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	J
r4	D
r6	I
r8	C

Phy. Reg		Rdy
A	6	Y
B	1	Y
C	6	Y
D	2	Y
E	10	Y
F	5	Y
G	0	Y
H	10	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB								

Free List: head G, E, F, A, H, B tail

Timing Assumptions (Arbitrary)

- Up to two instructions of any type can be dispatched to their respective RS in each cycle
- An instruction can begin execution as early as the same cycle it is dispatched to an RS
- Execution priority is given to the older instruction in RS
- One 1-cyc adder and one pipelined 2-cyc multiplier
- Producer notifies its dependent insts in its last execution cycle (CBD-based, no “dead reckoning”)
- An instruction is removed from RS after its last execution cycle
- If appropriate, an instruction can commit/retire as early as the cycle after its last execution cycle.

Worksheet for You to Try

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
Add: stage 1 ( )					

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	
r4	
r6	
r8	

Phy. Reg		Rdy
A		
B		
C		
D		
E		
F		
G		
H		
I		
J		

	7	6	5	4	3	2	1	0
ROB								

Free List: head ____ tail ____

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
Add: stage 1 ( )					

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	
r4	
r6	
r8	

Phy. Reg		Rdy
A		
B		
C		
D		
E		
F		
G		
H		
I		
J		

	7	6	5	4	3	2	1	0
ROB								

Free List: head ____ tail ____

Initial State @ Start of Cycle #1

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
Add: stage 1 ( )					

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	J
r4	D
r6	I
r8	C

Phy. Reg		Rdy
A	6	Y
B	1	Y
C	6	Y
D	2	Y
E	10	Y
F	5	Y
G	0	Y
H	10	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB								

head tail

Free List: G, E, F, A, H, B

End of Cycle #1

I0: r4 ← r4 + r6; **I1:** r2 ← r2 + r6

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
0	G	D	Y	I	Y
1	E	J	Y	I	Y
Add: stage 1 (0)					

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	E
r4	G
r6	I
r8	C

Phy. Reg		Rdy
A	6	Y
B	1	Y
C	6	Y
D	2	Y
E	10	N
F	5	Y
G	0	N
H	10	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB							I1/J	I0/D

head tail

Free List: F, A, H, B

End of Cycle #1

I0: r4 ← r4 + r6; **I1:** r2 ← r2 + r6

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
0	G	D	Y	I	Y
1	E	J	Y	I	Y
Add: stage 1 (0)					

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	E
r4	G
r6	I
r8	C

Phy. Reg		Rdy
A	6	Y
B	1	Y
C	6	Y
D	2	Y
E	10	N
F	5	Y
G	0	N
H	10	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB							$I1_{/J}$	$I0_{/D}$

head tail

Free List: F, A, H, B

End of Cycle #2

I1: r2 ← r2 + r6; **I2:** r4 ← r2 * r4; **I3:** r4 ← r4 + r6

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
1	E	J	Y	I	Y
3	A	F	N	I	Y
Add: stage 1 (1)					

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
2	F	E	N	G	Y
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	E
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	6	N
B	1	Y
C	6	Y
D	2	Y
E	10	N
F	5	N
G	5	Y
H	10	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB					$I3_{/F}$	$I2_{/G}$	$I1_{/J}$	

head tail

Free List: H, B, D

End of Cycle #2

I1: r2 ← r2 + r6; **I2:** r4 ← r2 * r4; **I3:** r4 ← r4 + r6

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
1	E	J	Y	I	Y
3	A	F	N	I	Y
Add: stage 1 (1)					

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
2	F	E	N	G	Y
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	E
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	6	N
B	1	Y
C	6	Y
D	2	Y
E	10	N
F	5	N
G	5	y
H	10	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB					$I3_{/F}$	$I2_{/G}$	$I1_{/J}$	

Free List: head H, B, D tail

End of Cycle #3

I2: r4 ← r2 * r4; **I3:** r4 ← r4 + r6; **I4:** r2 ← r2 + r6; **I5:** r2 ← r2 + r4

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
4	H	E	Y	I	Y
5	B	H	N	A	N
3	A	F	N	I	Y
Add: stage 1 (4)					

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
2	F	E	Y	G	Y
Mult: stage 1 (2) stage 2 ( )					

Map Table	
r2	B
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	6	N
B	1	N
C	6	Y
D	2	Y
E	5	Y
F	5	N
G	5	y
H	10	N
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB			I5/ H	I4/ E	I3/ F	I2/ G		

Free List: head D, J tail

End of Cycle #3

I2: r4 ← r2 * r4; I3: r4 ← r4 + r6; I4: r2 ← r2 + r6; I5: r2 ← r2 + r4

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
4	H	E	Y	I	Y
5	B	H	N	A	N
3	A	F	N	I	Y
Add: stage 1 (4)					

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
2	F	E	Y	G	Y
Mult: stage 1 (2) stage 2 ( )					

Map Table	
r2	B
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	6	N
B	1	N
C	6	Y
D	2	Y
E	5	Y
F	5	N
G	5	Y
H	10	N
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB			I5/ H	I4/ E	I3/ F	I2/ G		

head tail

Free List: D, J

End of Cycle #4

I2: r4 ← r2 * r4; I3: r4 ← r4 + r6; I4: r2 ← r2 + r6; I5: r2 ← r2 + r4

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
5	B	H	Y	A	N
3	A	F	N	I	Y
Add: stage 1 ( )					

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
2	F	E	Y	G	Y
Mult: stage 1 ( ) stage 2 (2)					

Map Table	
r2	B
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	6	N
B	1	N
C	6	Y
D	2	Y
E	5	Y
F	5	N
G	5	Y
H	8	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB			I5/ H	I4/ E	I3/ F	I2/ G		

head tail

Free List: D, J

End of Cycle #4

I2: r4 ← r2 * r4; **I3:** r4 ← r4 + r6; **I4:** r2 ← r2 + r6; **I5:** r2 ← r2 + r4

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
5	B	H	Y	A	N
3	A	F	N	I	Y
Add: stage 1 ( )					

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
2	F	E	Y	G	Y
Mult: stage 1 ( ) stage 2 (2)					

Map Table	
r2	B
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	6	N
B	1	N
C	6	Y
D	2	Y
E	5	Y
F	5	N
G	5	Y
H	8	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB			I5/ H	I4/ E	I3/ F	I2/ G		

head tail

Free List: D, J

End of Cycle #5

I3: r4 ← r4 + r6; **I4:** r2 ← r2 + r6; **I5:** r2 ← r2 + r4

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
5	B	H	Y	A	N
3	A	F	Y	I	Y
Add: stage 1 (3)					

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	B
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	6	N
B	1	N
C	6	Y
D	2	Y
E	5	Y
F	25	Y
G	5	Y
H	8	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB			I5/ H	I4/ E	I3/ F			

head tail

Free List: D, J, G

End of Cycle #5

I3: r4 ← r4 + r6; **I4:** r2 ← r2 + r6; **I5:** r2 ← r2 + r4

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
5	B	H	Y	A	N
3	A	F	Y	I	Y
Add: stage 1 (3)					

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	B
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	6	N
B	1	N
C	6	Y
D	2	Y
E	5	Y
F	25	Y
G	5	y
H	8	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB			I5/ H	I4/ E	I3/ F			

head tail

Free List: D, J, G

End of Cycle #6

I5: r2 ← r2 + r4

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
5	B	H	Y	A	Y
Add: stage 1 (5)					

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	B
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	28	Y
B	1	N
C	6	Y
D	2	Y
E	5	Y
F	25	Y
G	5	y
H	8	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB			I5/ H					

head tail

Free List: D, J, G, F, E

End of Cycle #6

I5: r2 ← r2 + r4

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
5	B	H	Y	A	Y
Add: stage 1 (5)					

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	B
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	28	Y
B	1	N
C	6	Y
D	2	Y
E	5	Y
F	25	Y
G	5	Y
H	8	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB			I5/ H					

head tail

Free List: D, J, G, F, E

End of Cycle #7

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
Add: stage 1 ( )					

ROB ID	Dst	Src1 Tag	Src1 Rdy	Src2 Tag	Src2 Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	B
r4	A
r6	I
r8	C

Phy. Reg		Rdy
A	28	Y
B	36	Y
C	6	Y
D	2	Y
E	5	Y
F	25	Y
G	5	Y
H	8	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB								

head tail

Free List: D, J, G, F, E, H

Precise Exception Rewind

- Suppose at the end of I2's second cycle of execution, I2 encounters an exception and cannot complete.

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
Add: stage 1 ( )					

ROB ID	Dst	Src1		Src2	
		Tag	Rdy	Tag	Rdy
Mult: stage 1 ( ) stage 2 ( )					

Map Table	
r2	E
r4	G
r6	I
r8	C

Phy. Reg		Rdy
A	6	Y
B	1	Y
C	6	Y
D	2	Y
E	5	Y
F	5	Y
G	5	Y
H	8	Y
I	3	Y
J	2	Y

	7	6	5	4	3	2	1	0
ROB								

Free List: head F, A, H, B, D, J tail

- See https://github.com/jhoecmu/ooo-beta for more

Memory Dataflow Concepts

Superscalar Speculative Out-of-Order

- Modern CPUs can have over 100 instructions in out-of-order execution scope
  - ROB has 100's of entries
  - RS do not have 100's of entries
- Keep in mind, if average basic block is 7 inst, over 10 levels of branch prediction

- **Important Question:**
  - **how much more ILP is uncovered with look ahead**
  - **how much useful work is done during look ahead**

Ans: not much and not much

Truth about Superscalar Speculative OOO

- If memory speed kept up with core speed, we would still be building in-order pipelines
- But, by 2005 we were seeing

  e.g., Intel P4 at 4+GHz

  - 16KB L1 D-cache
    - t_1 = 4 cyc int (9 cycle fp)
  - 1024KB L2 D-cache
    - t_2 = 18 cyc int (18 cyc fp)
  - Main memory
    - t_3 = ~ 50ns or 180 cyc

- Speculative OOO has really been about
  - finding independent work to do after cache hit&miss
  - getting to future cache misses as early as possible
  - overlapping multiple cache misses for BW (aka MLP)

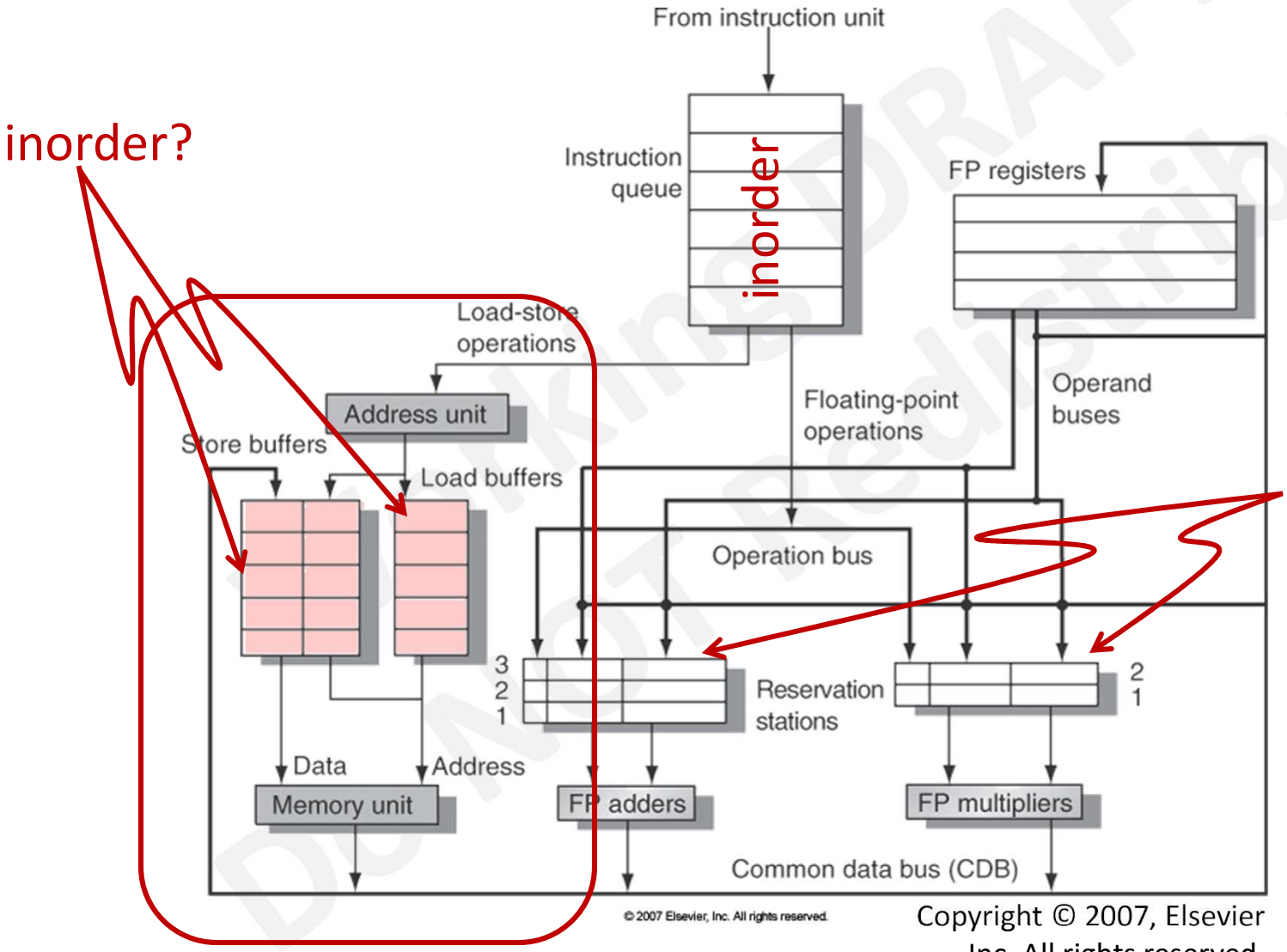
Scheduling Memory Operations

- Memory data dependence ⇒ RAW, WAR, WAW
- Addresses use GPR ⇒ register dependence
- Storing has side-effect that cannot be undone

  ⇒ must wait until commit
- When earliest to safely start LW <u>on uniprocessor</u>?
  - no more older pending SW *OR*
  - no older SW with conflicting address (requires knowing all older SW addresses) *OR*
  - just go if no known conflict; reload if new RAW hazard later

What about MP memory consistency?

IBM 360/91 FP Module [1967]



inorder?

out-of-order issue window

Tomasulo's Algorithm

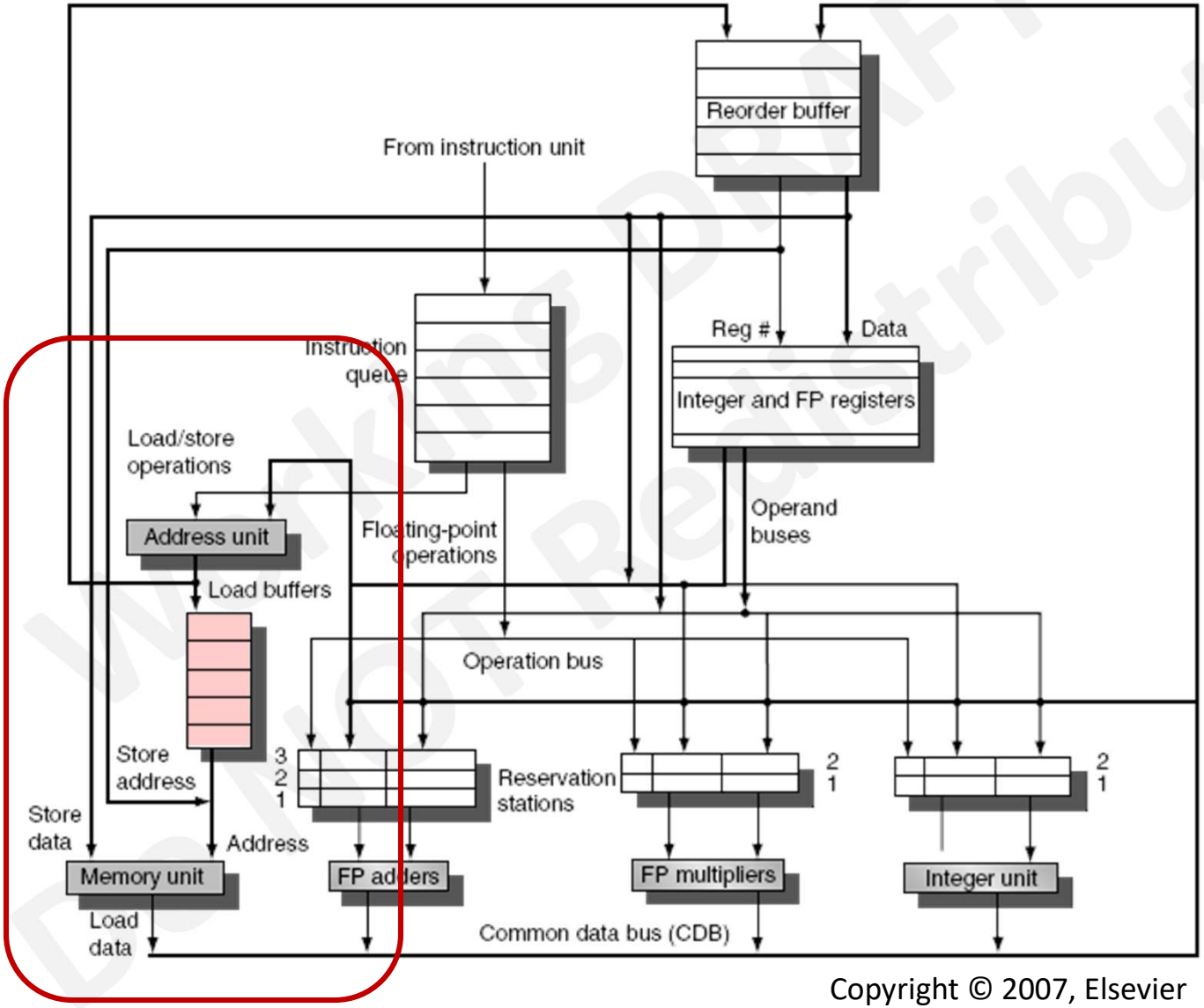
Instruction state	Wait until	Action or bookkeeping
Issue FP operation	Station r empty	`if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Q ← r;`
Load or store	Buffer r empty	`if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;`
Load only		`RegisterStat[rt].Qi ← r;`
Store only		`if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
Execute FP operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load-store step 1	RS[r].Qj = 0 & r is head of load-store queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 complete	Read from Mem[RS[r].A]
Write Result FP operation or load	Execution complete at r & CDB available	`∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;`
Store	Execution complete at r & RS[r].Qk = 0	`Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;`

from [Hennessy&Patterson, CAAQA]

RS entry
- **Busy**: in use
- **Op**: operand
- **Vj**: op1 value
- **Vk**: op2 value
- **Qj**: op1 RS-tag
- **Qk**: op2 RS-tag

case (RegisterStat)
0: RF val current
RS-tag:
to be produce by corresponding instruction

Tomasulo + Speculative Execution



Tomasulo's Algorithm + ROB

Status	Wait until	Action or bookkeeping
Issue all instructions		`if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/ {h ← RegisterStat[rs].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;};`
	Reservation station (r) and ROB (b) both available	`RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no;`
FP operations and stores		`if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};`
FP operations		`RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;`
Loads		`RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;`
Stores		`RS[r].A ← imm;`

from [Hennessy&Patterson, CAAQA]

ROB Entry:

is **Busy** / **Instruction** / logical **Dest** reg / dest **Value** / value is **Ready**

RegisterStat: reg is **Busy** / renamed to **Reorder** buffer entry #

Tomasulo's Algorithm + ROB

Execute FP op	(RS[r].Qj == 0) and (RS[r].Qk == 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj == 0) and there are no stores earlier in the queue	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from Mem[RS[r].A]
Store	(RS[r].Qj == 0) and store at queue head	ROB[h].Address ← RS[r].Vj + RS[r].A;
Write result all but store	Execution done at r and CDB available	b ← RS[r].Dest; RS[r].Busy ← no; ∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0}); ∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0}); ROB[b].Value ← result; ROB[b].Ready ← yes;
Store	Execution done at r and (RS[r].Qk == 0)	ROB[h].Value ← RS[r].Vk;

Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes	d ← ROB[h].Dest; /* register dest, if exists */
		if (ROB[h].Instruction==Branch)
		{if (branch is mispredicted)
		{clear ROB[h], RegisterStat; fetch branch dest;};}
		else if (ROB[h].Instruction==Store)
		{Mem[ROB[h].Destination] ← ROB[h].Value;}
		else /* put the result in the register destination */
		{Regs[d] ← ROB[h].Value;};
		ROB[h].Busy ← no; /* free up ROB entry */
		/* free up dest register if no one else writing it */
		if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;};

? subtle

from [Hennessy&Patterson, CAAQA]

DRIS is also Address Queue

- LW/SW issued 1st-time as "ADD"
  - compute base+offset
  - result written to **Data**; complete in dataflow order
- SW issue 2nd-time on retire
- LW issue 2nd-time when RAW-free
  1. wait until all older SW addresses available
  2. check (by CAM) LW addr against older SW addr's
  3. If conflict, forward youngest matching SW to LW
- OOO for shared-memory MP must obey additional ordering rules

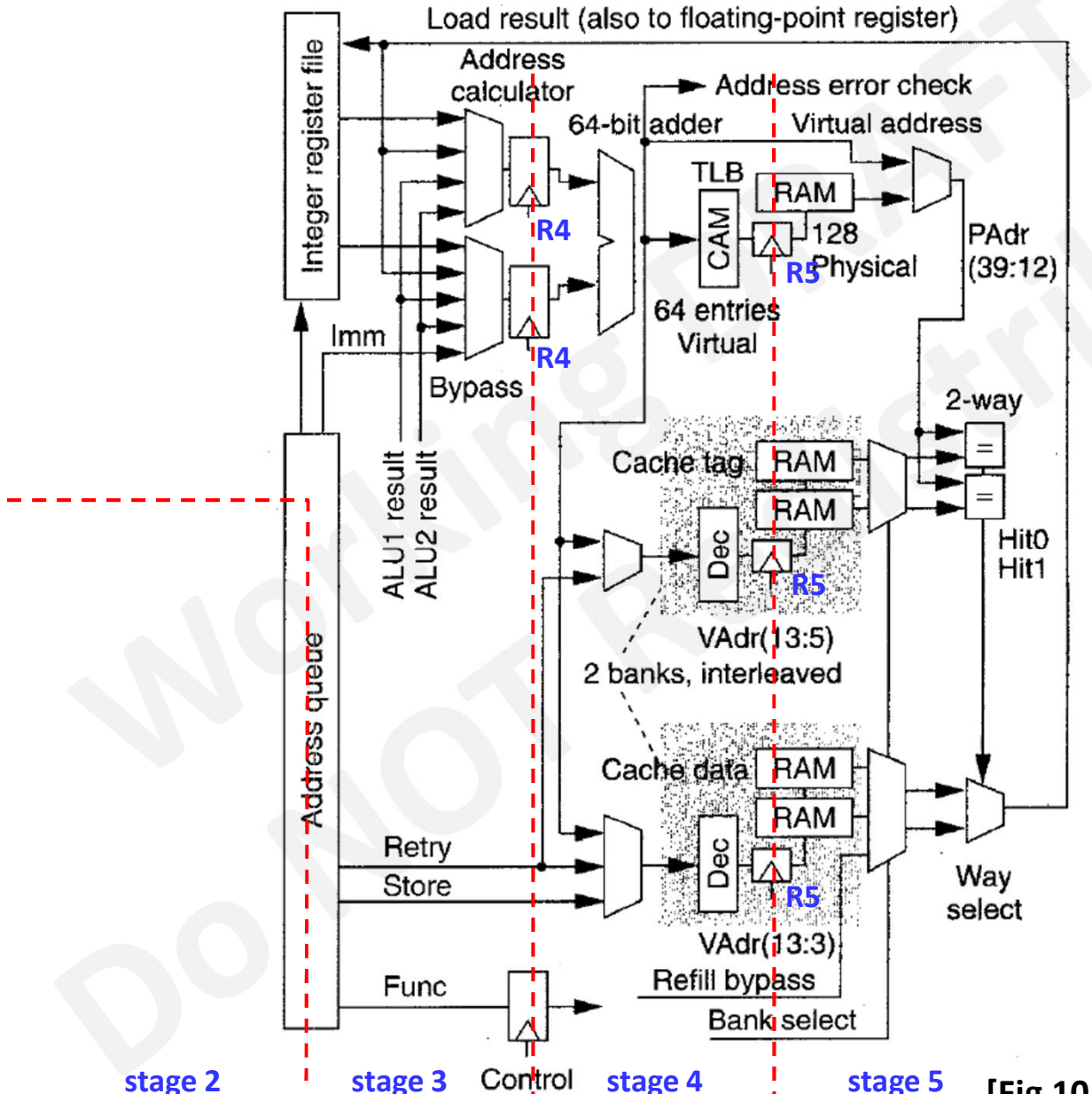
How many address-width comparators do you imagine??

Memory Dataflow Realities

What makes it hard

- Memory dependencies are not static
  - LW and SW addresses need to be calculated, and
  - translated (to check RAW/WAR/WAW synonyms)
- Memory addresses are wider than register names
- Memory instructions take longer to execute relative to other instructions types
- Scheduling takes into account $O(N^2)$ dependency relationships in program order
- More special ordering rules if shared memory MP
- **Want to load as soon as possible!!!**

R10K Load Data Path



[Fig 10, Yeager 1996, IEEE Micro]

Memory Disambiguation

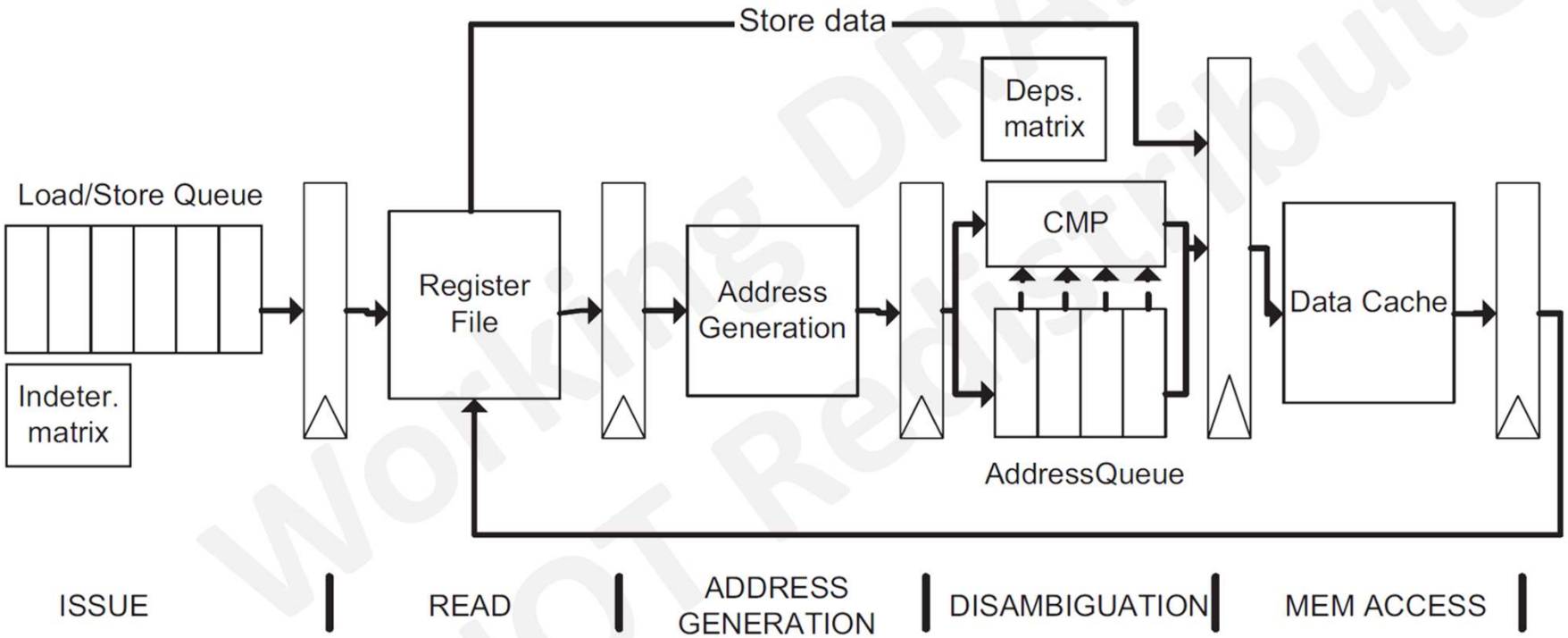
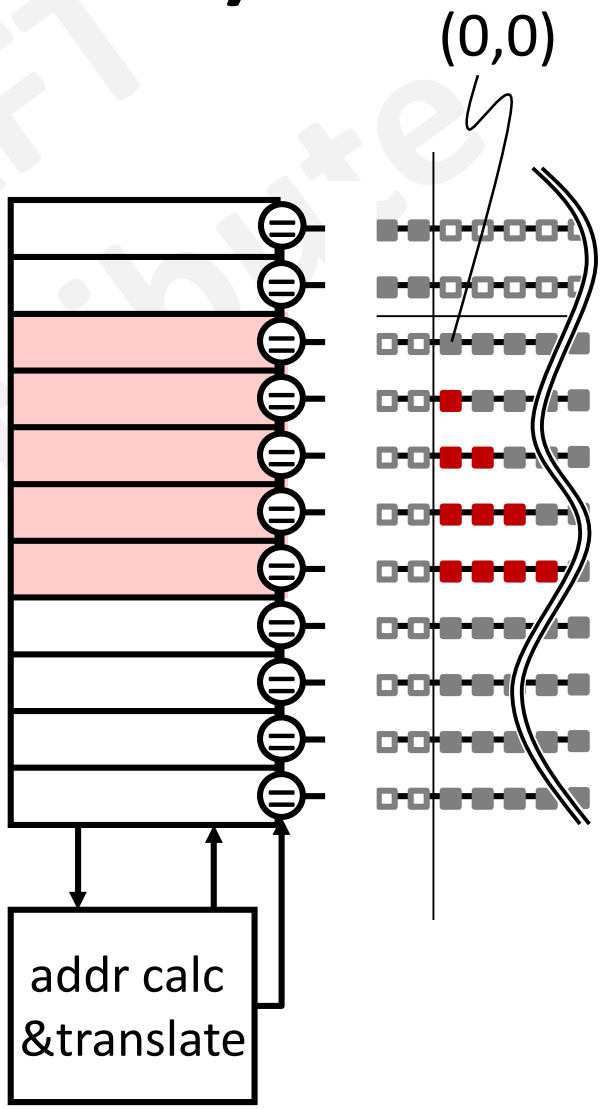


FIGURE 6.8: Schematics of the MIPS R10000 pipeline to implement the partial ordering memory disambiguation policy. from [Gonzalez, et al., 2010]

Address Queue (16 entries)

- Track LW/SW and addr in program order
- Addr calculation issued in dataflow order
- N-comparators compares new/external addr to each known-so-far addr to detect
  - LW/SW RAW hazards
  - cache line return after load miss
  - external invalidations (more later)
- Pair-wise RAW hazards matrix (N^2 bits) updated incrementally after each LW/SW addr becomes known
- LW do not issue against known/potential RAW hazard or pending cache miss



Dependence Matrix

store

Set on allocation

Unset on @ computation

oldest

youngest

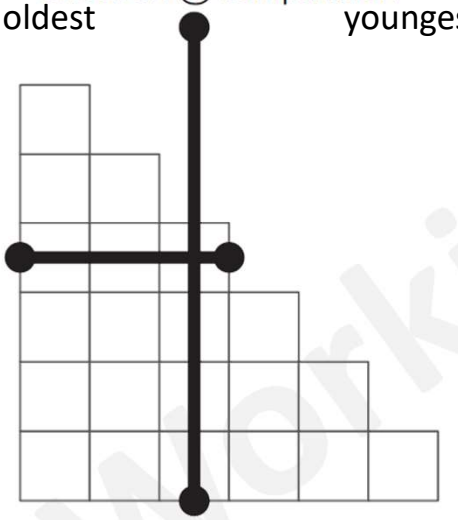
Oldest memory operation

load

Issue only if all entries unset

also check dep matrix

Youngest memory operation



indetermination matrix

at commit

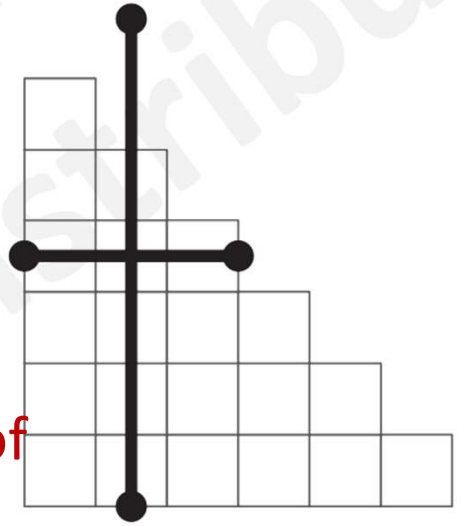
Store resets all entries when it updates the memory

Loads set to 1 the bits of the entries associated to stores they depend

at later of 2 addr in a RAW pair (st<-lw)

Oldest memory operation

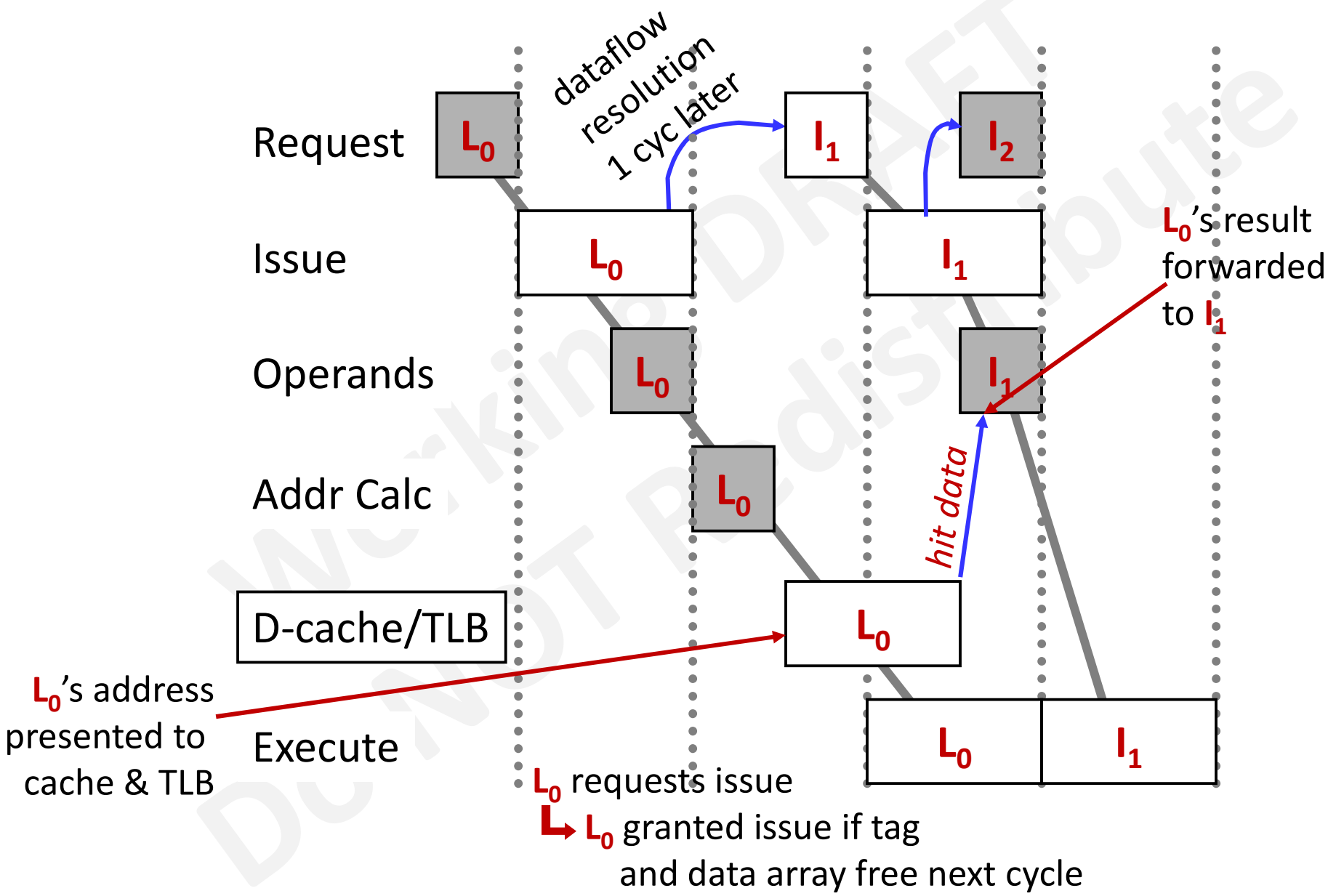
Youngest memory operation



dependence matrix

from [Gonzalez, et al., 2010]

Best Case Load Timing



Last Slide is only for "Best Case"

1. TLB, tag bank, and data bank all available
   - 3 independently contended/arbitrated resources
   - external CC lookup has priority use of tag bank
2. TLB and tag bank both have to hit
3. No load-after-store dependence

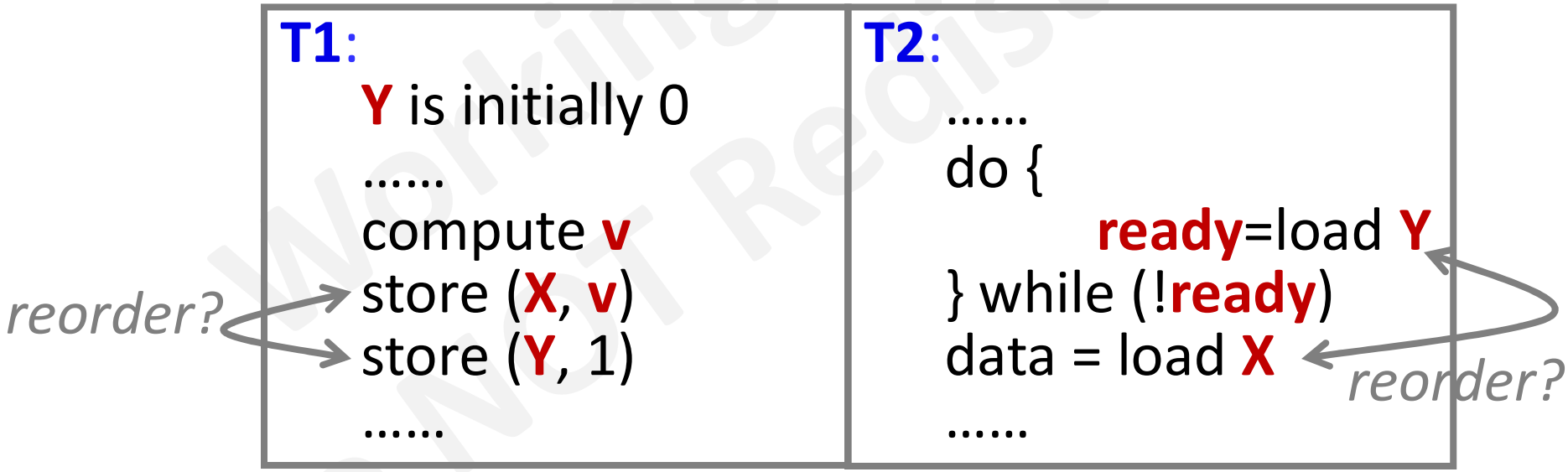
Probably requires no older stores at all to succeed; no time to check address queue after TLB lookup

- Extra Note: no store-to-load forwarding in R10K
  - quite hard *(e.g., how to know store data is ready?)*
  - little value added for large GPR ISAs *(whereas x86 required it due to frequent register spills)*

OOO and Memory Consistency

SC doesn't come freely in OOO

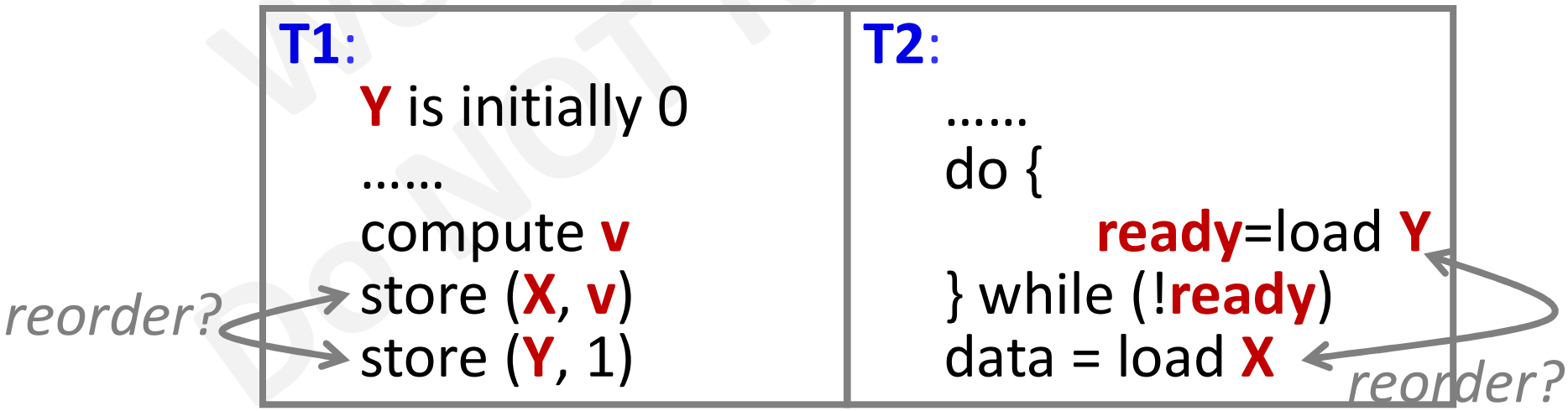
- Threads **T1** and **T2** communicate via shared memory locations **X** and **Y**
  - **T1** produces result in **X** to be consumed by **T2**
  - **T1** signals readiness to **T2** by setting **Y**



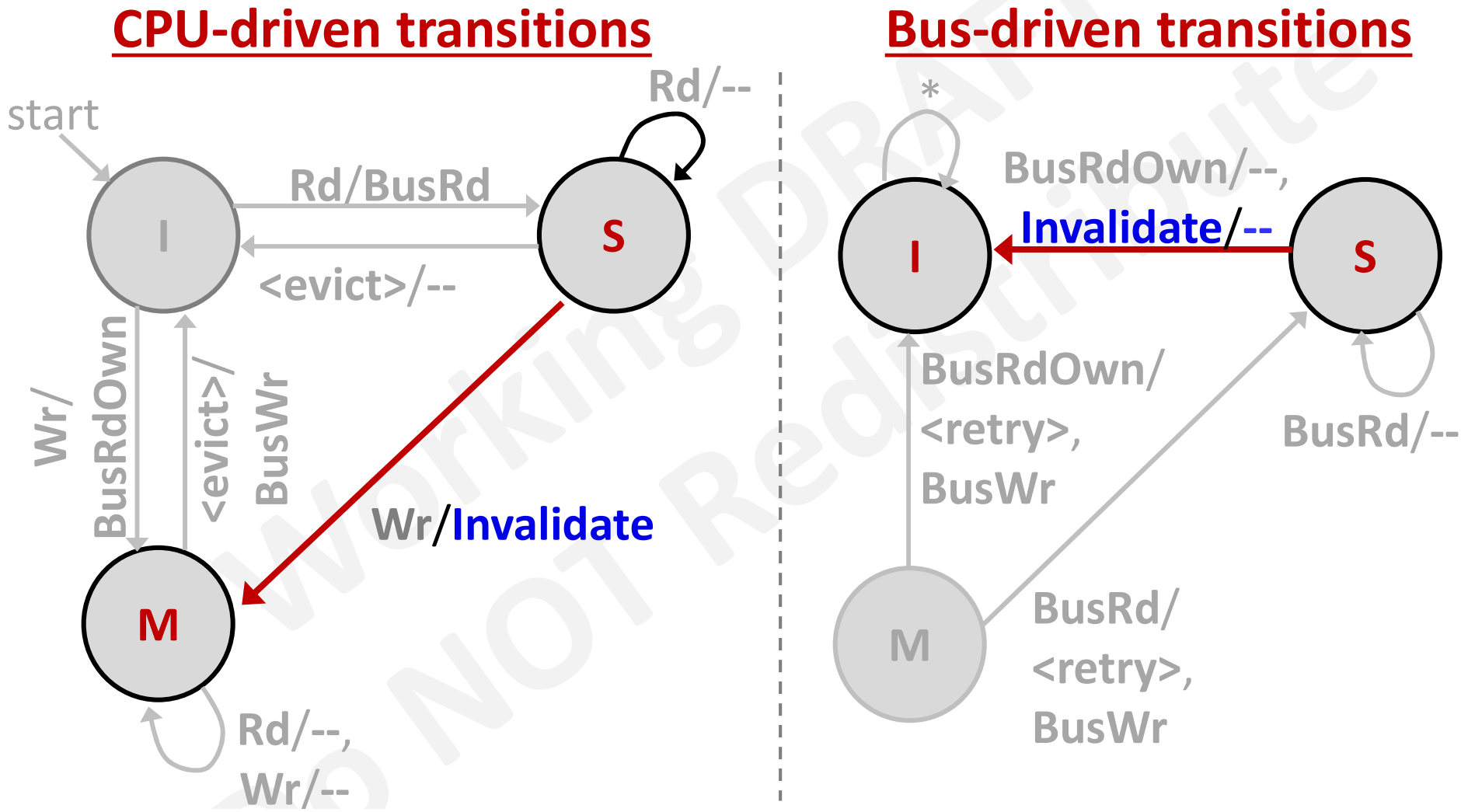
- This works because SC says **T1** and **T2** must see the stores to **X** and **Y** in the same order

OOO and Consistency

- **T1:** inorder store commit by core no guarantee
  - reorder by cache (e.g., non-blocking miss handling)
  - non-atomic write (e.g., write-through cache, or MSI that allow M and S to overlap transiently)
- **T2:** speculative OOO allows reorder across data-dependent control dependence *(whether load X happens depends on value of load Y)*



Aside: Write Atomicity



Q: when can writer's cache promote S→M after issuing invalidate?

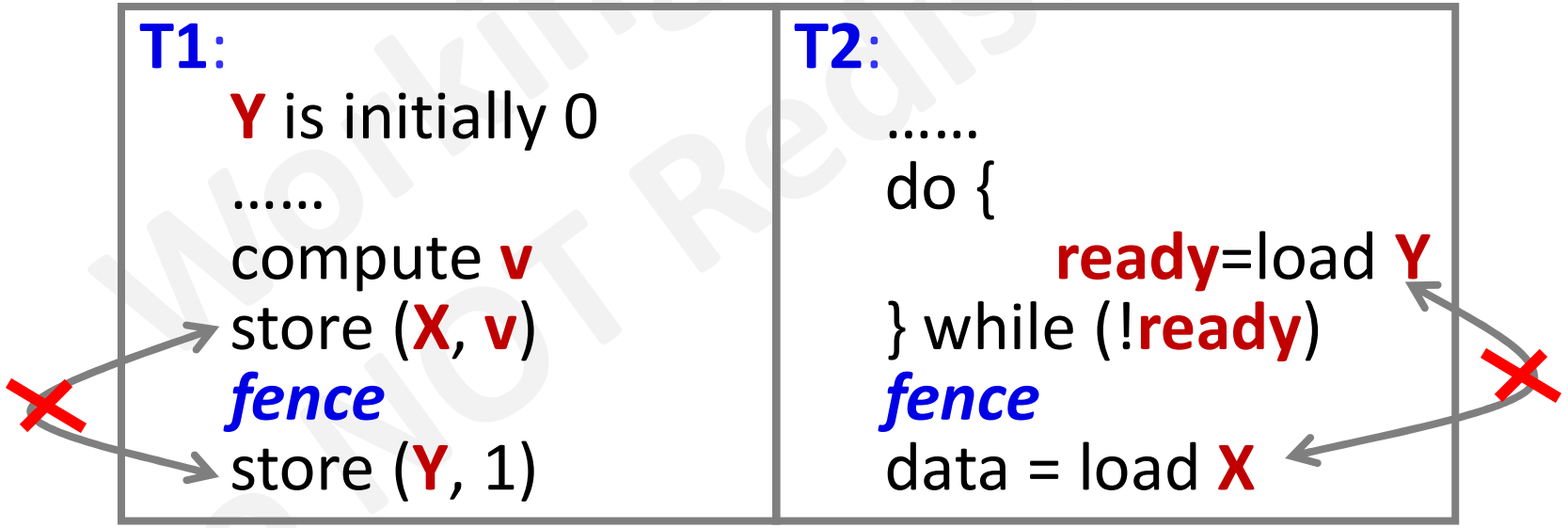
A: if WC, go for it; if SC, strictly after all S→I *(how to know?)*.

Weak Consistency (WC)

- WC only imposes uniprocessor memory dependence: $R_i(\mathbf{x})<W_j(\mathbf{x})$; $W_i(\mathbf{x})<R_j(\mathbf{x})$; $W_i(\mathbf{x})<W_j(\mathbf{x})$

  All optimizations apply on sequential parts of prgm

- Program insert explicit **memory fence** instructions to force ordering when it matters



- Implementation wise, heavy-weight fence stalls subsequent LW/SW progress until full ordering

It can get weird: "Ghost Loads"

- Assume **X** and **Y** initially 0

T1:	**T2**:
v1=load **X** store (**Y**, v1)	v2=load **Y** store (**X**, 3)

- What are allowed values for v2 under WC?

  Only values on the table are 0 and 3, can it be 3?

WC says T2's load can see the value 3 introduced by its own future store!!

R10K Speculative SC

TSO if just this

- Stores commit in-order when retired, but also need
  - cache block in M ready and locked at SW retirement
  - full invalidation ack in acquiring M (write atomicity)
- Loads attempted speculatively and out-of-order
  - address of invalidated cache blocks checked against speculative loads in load/store queue
  - if a loaded cache-block invalidated before load retires, "soft exception" restart from oldest affected load
  - load retires only if the fetched value is still "current"

⇒ *Loads and stores appears executed in program order at the commit point*

⇒ *No penalty for sequential programs!!!*