



고려대학교
KOREA UNIVERSITY

COM515 Advanced Computer Architecture

Lecture 5. Dynamic Scheduling II

Prof. Taeweon Suh

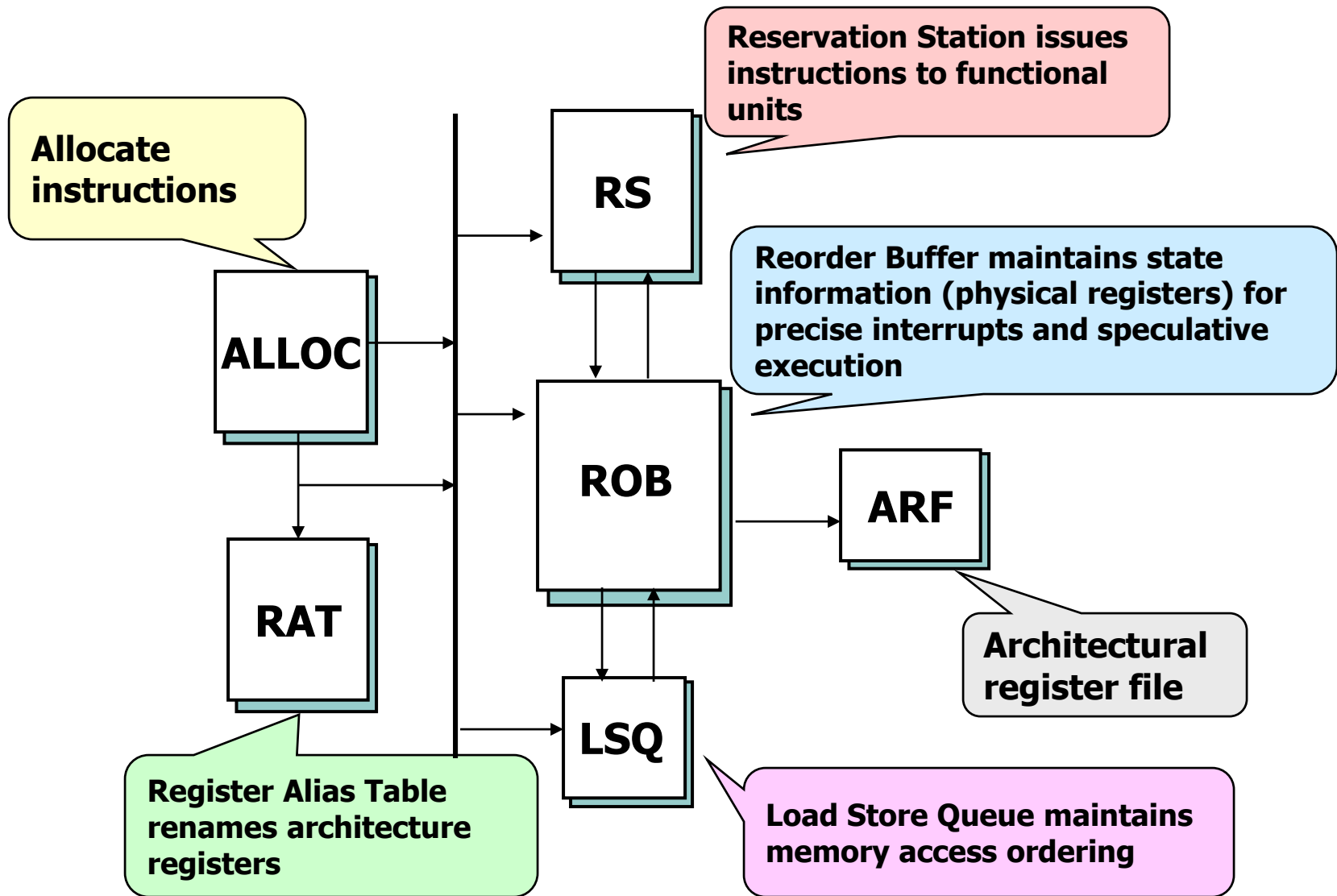
Computer Science Education

Korea University

Modern Processors

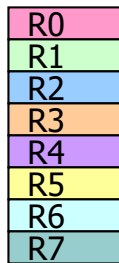
- Branch Prediction results in speculative execution
- Speculative instructions (if wrongly speculated) must not alter the architecture states
 - Architecture Registers
 - Memory
- Requirement of precise exception/interrupts

Modern Out-of-Order Core

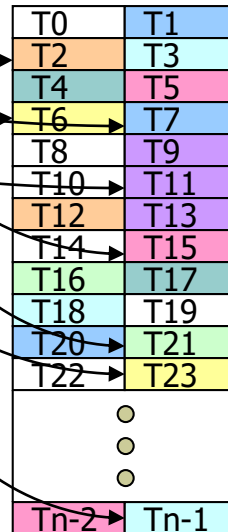


Register Renaming

Architectural
Registers



Physical
Registers



Original
Code

```

R2 = R1 + R3
R4 = R2 - R6
...
R2 = R7 / R5
BEQ R2, #1
...
R2 = R4 * R1
R6 = Load [R2]
    
```

 WAW
 WAR

Renamed
Code

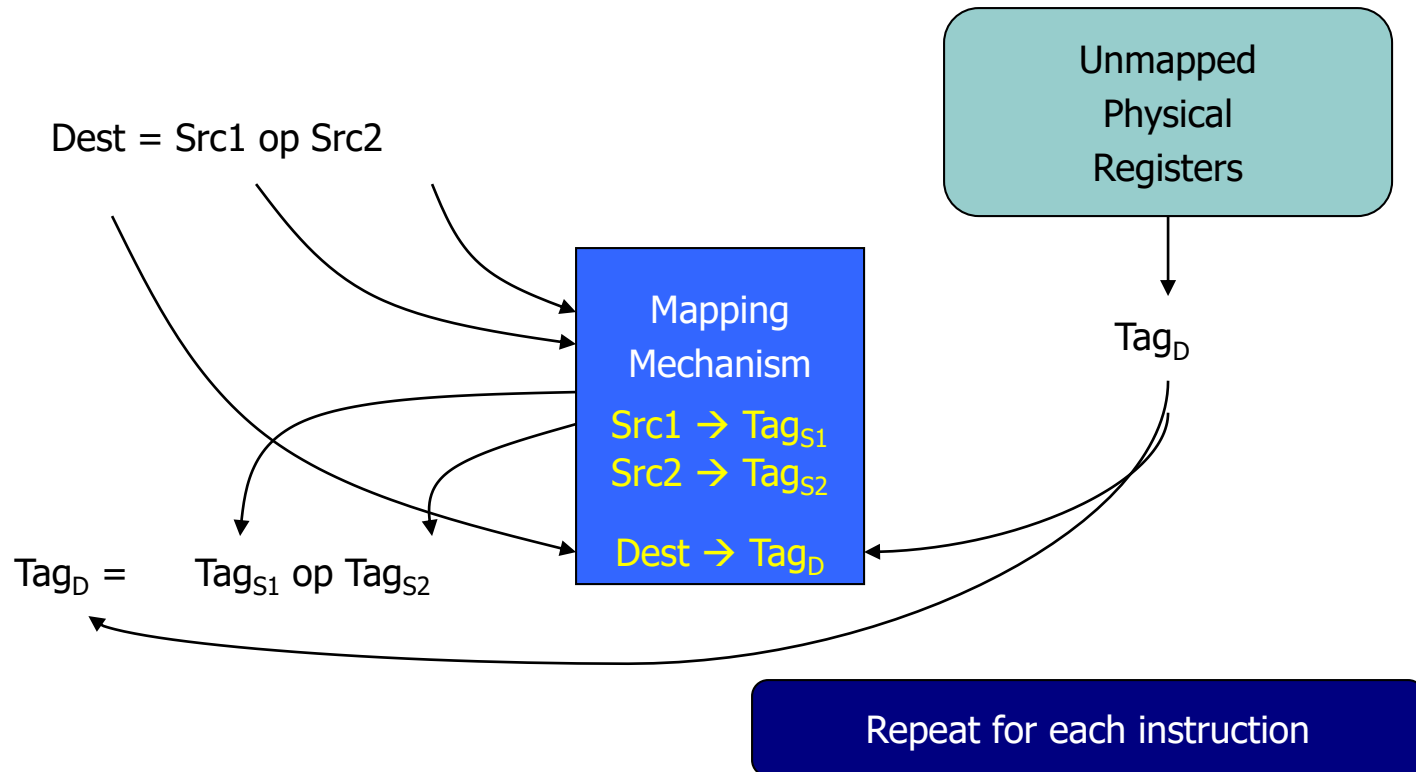
```

T1 = R1 + R3
R4 = T1 - R6
...
T20 = R7 / R5
BEQ T20, #1
...
T7 = R4 * R1
R6 = Load [T7]
    
```

No False
Dependencies!

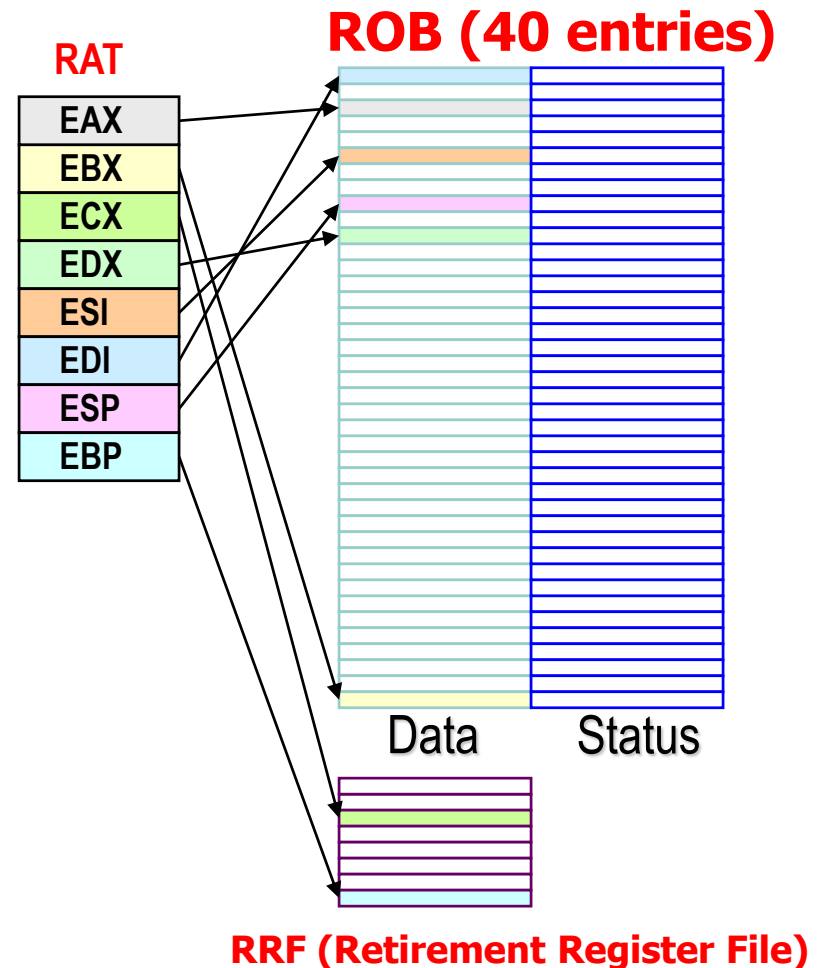
Sandy Bridge:
160 PRs for INT
144 PRs for FP

Register Renaming



Register Alias Table (RAT)

- Use a lookup table for renaming
- One entry per architectural register
- Each entry maps to the most recent version of the architectural register, could be in
 - Physical register file
 - Architectural register file

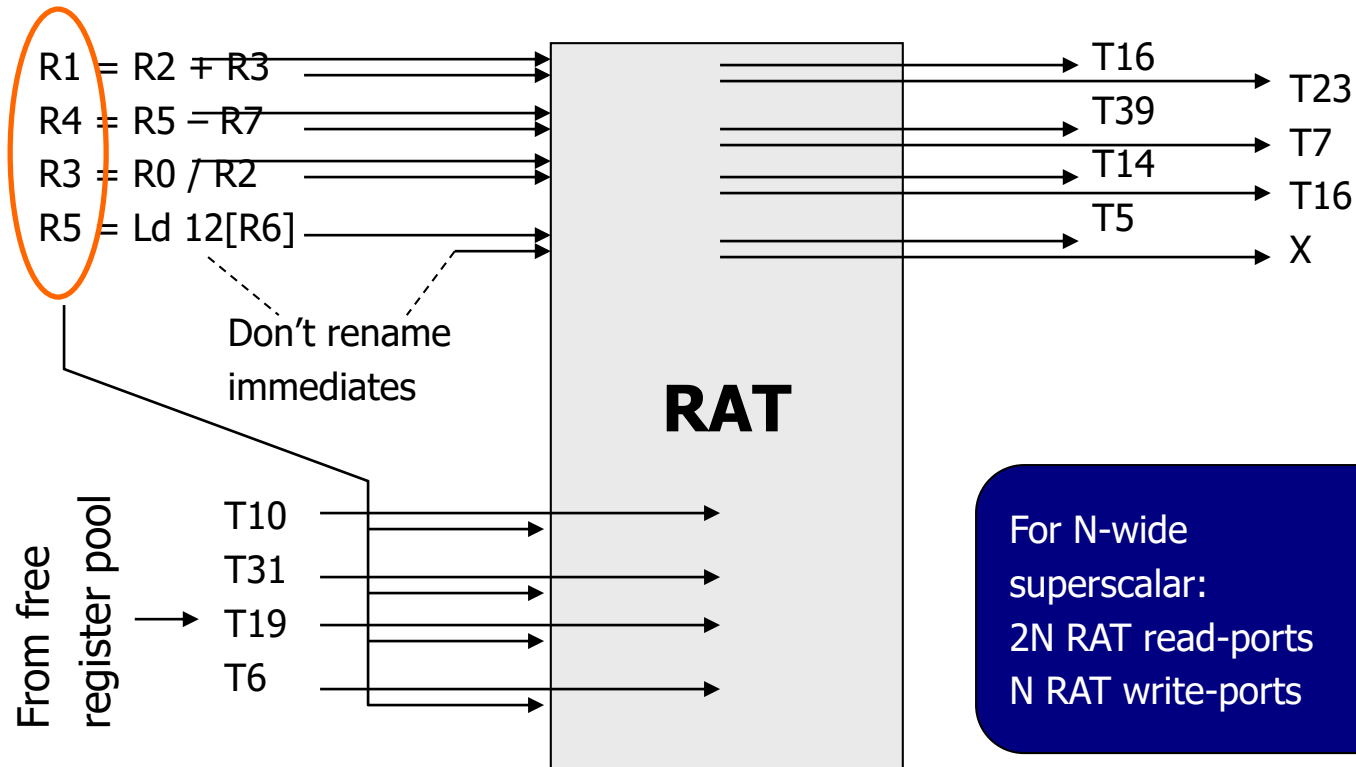


**P6 Style Register Renaming
(So does HP-PA8000, PPC604)**

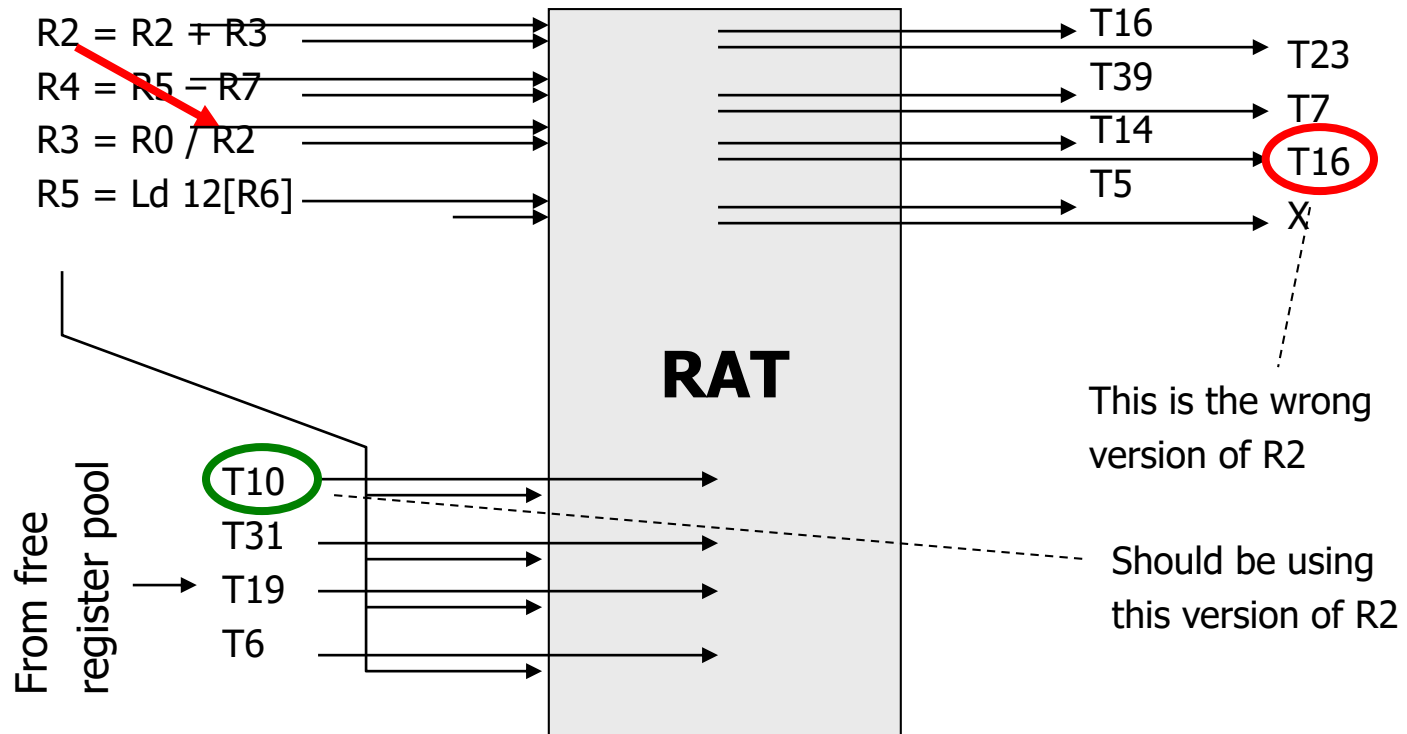
RAT Example

	R0	R1	R2	R3	R4	R5	R6	R7	Free Physical Regs
$R1 = R2 + R3$	-	-	-	-	-	-	-	-	T13, T14, T15, T16
T13 = R2 + R3									
$R5 = R4 - R1$	-	13	-	-	-	-	-	-	T14, T15, T16
T14 = R4 - T13									
$R1 = R1 * R5$	-	13	-	-	-	14	-	-	T15, T16
T15 = T13 * T14									
$R2 = R5 / R1$	-	15	-	-	-	14	-	-	T16
T16 = T14 / T15									
	-	15	16	-	-	14	-	-	

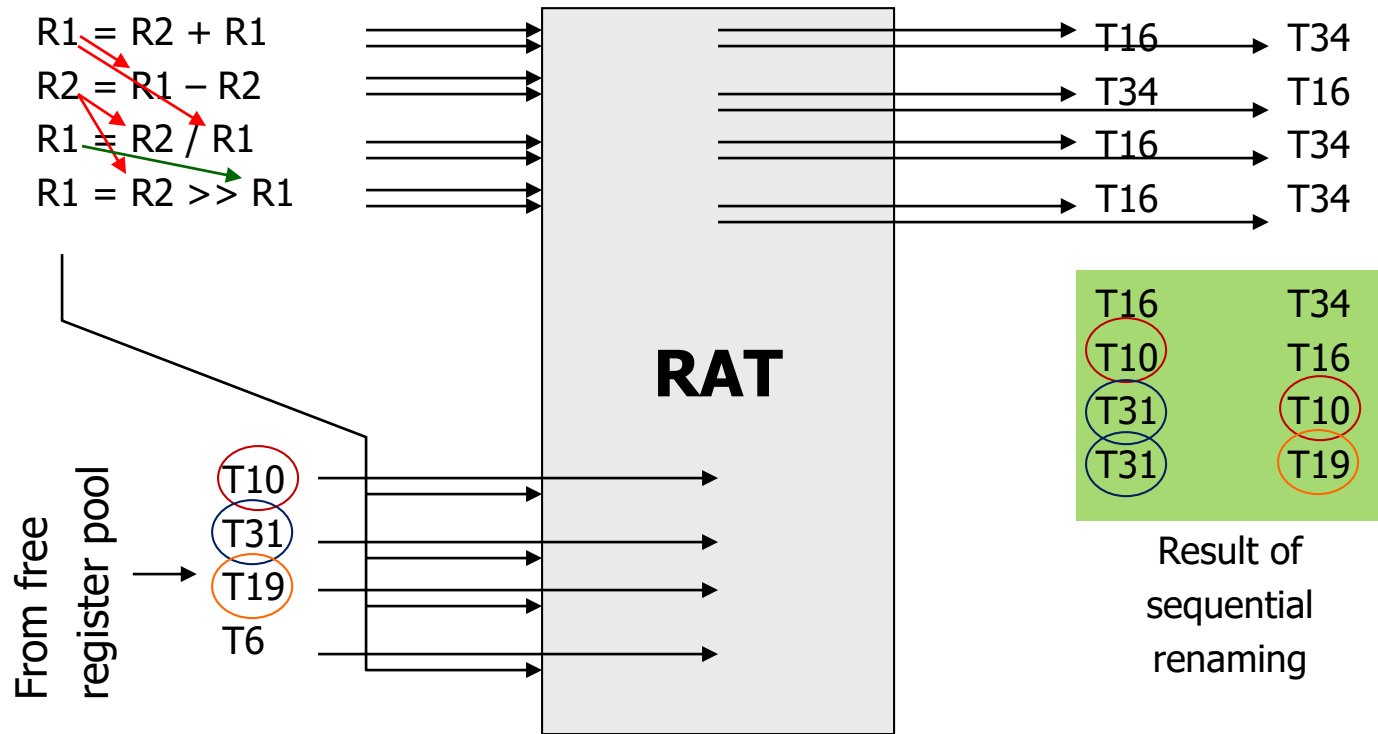
Superscalar Rename



Intra-Group Dependencies

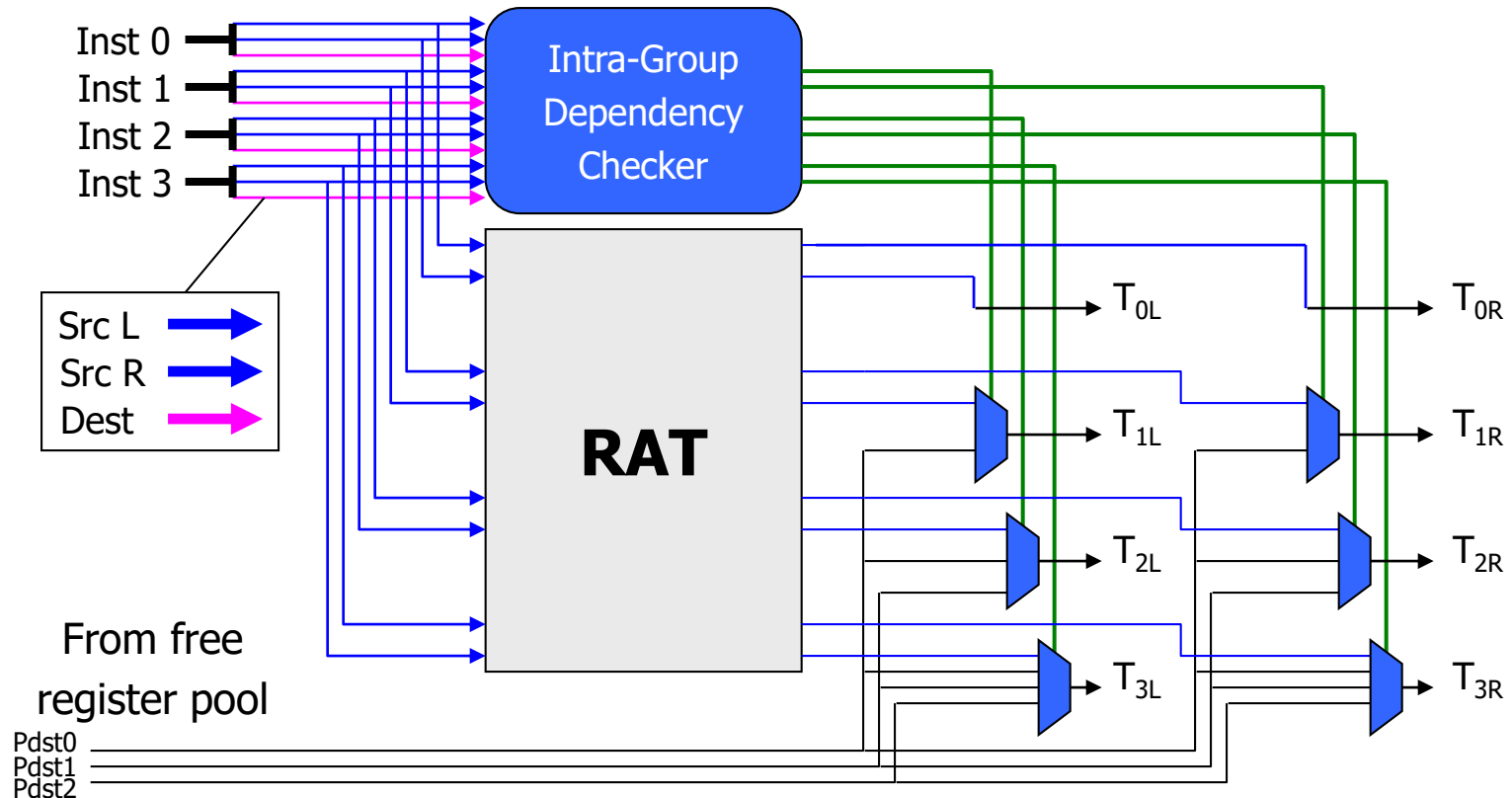


Intra-Group Dependencies

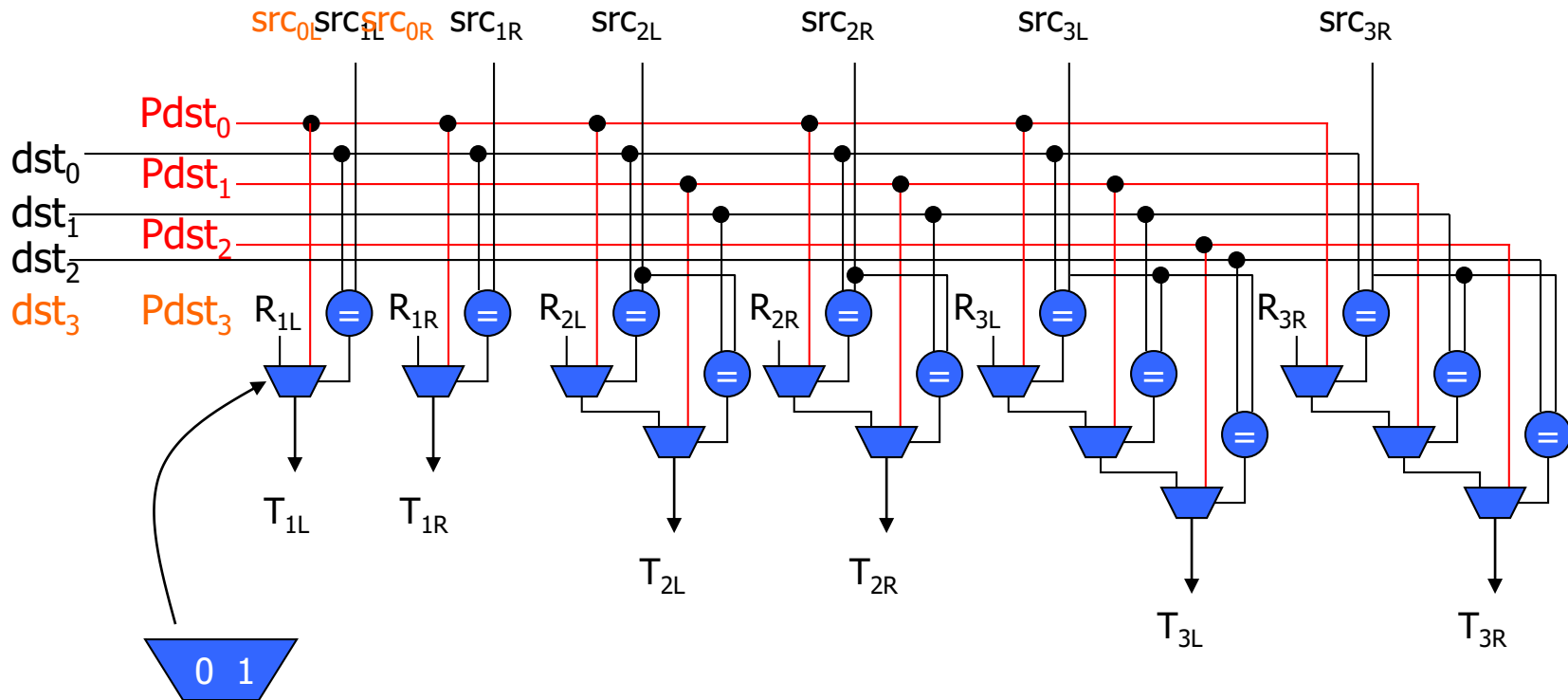


Correct final renamed registers

Resolving Intra-Group Dependencies



Intra-Group Dependency Checking

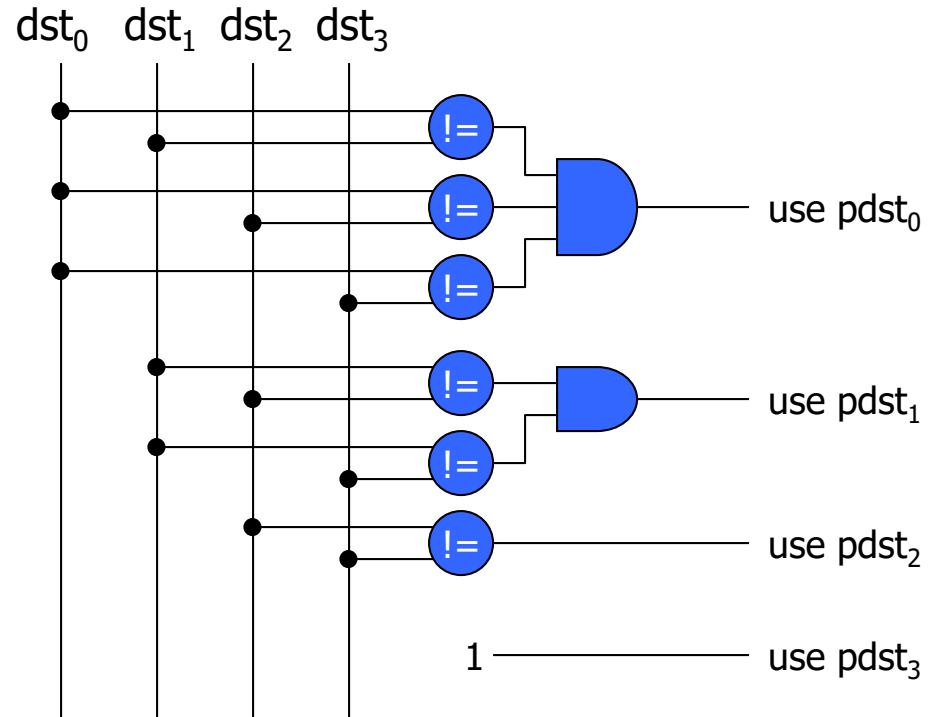


Mapping Selection

$R1 = R2 + R1$
 $R2 = R1 - R2$
 $R1 = R2 / R1$
 $R1 = R2 \gg R1$

Only this mapping
for **R1** should be
written into the RAT

Condition: use mapping
if instruction is last
writer to the register



Issue with Imprecise Interrupt

```
lw    r5, 8(r10)
add   r10, r9, r8
add   r12, r10, r7
```

- add instructions take one cycle
- E.g.,
 - Load (left side) induces a “data page fault”;
- If out-of-order completion is allowed
 - R10 and r12 will be modified
 - Wrong values will be used by the re-issued load
- Interrupt classes
 - Program interrupts (exceptions or traps)
 - External interrupts (asynchronous)

Precise Interrupts

- To reflect a sequential architecture model $\Rightarrow$ Serially correct (think about a single issue, non-pipelined processor)
- Keep “Precise State” of an execution
 - All instructions before the interrupted instruction must be completed
 - The state should appear as if no instruction issued after the interrupted instruction
 - The interrupted PC should be presented to the interrupt handler (restartable)
- Similar to branch misprediction handling
- Out-of-order execution makes the ordering hard
 - Undo what comes after an interrupt

Why Support Precise Interrupts

- Need to maintain a precise state (for recovery)
- Software debugging
- I/O or timer interrupts
- Virtual memory (page fault)
- Instruction emulation
- Virtual machines

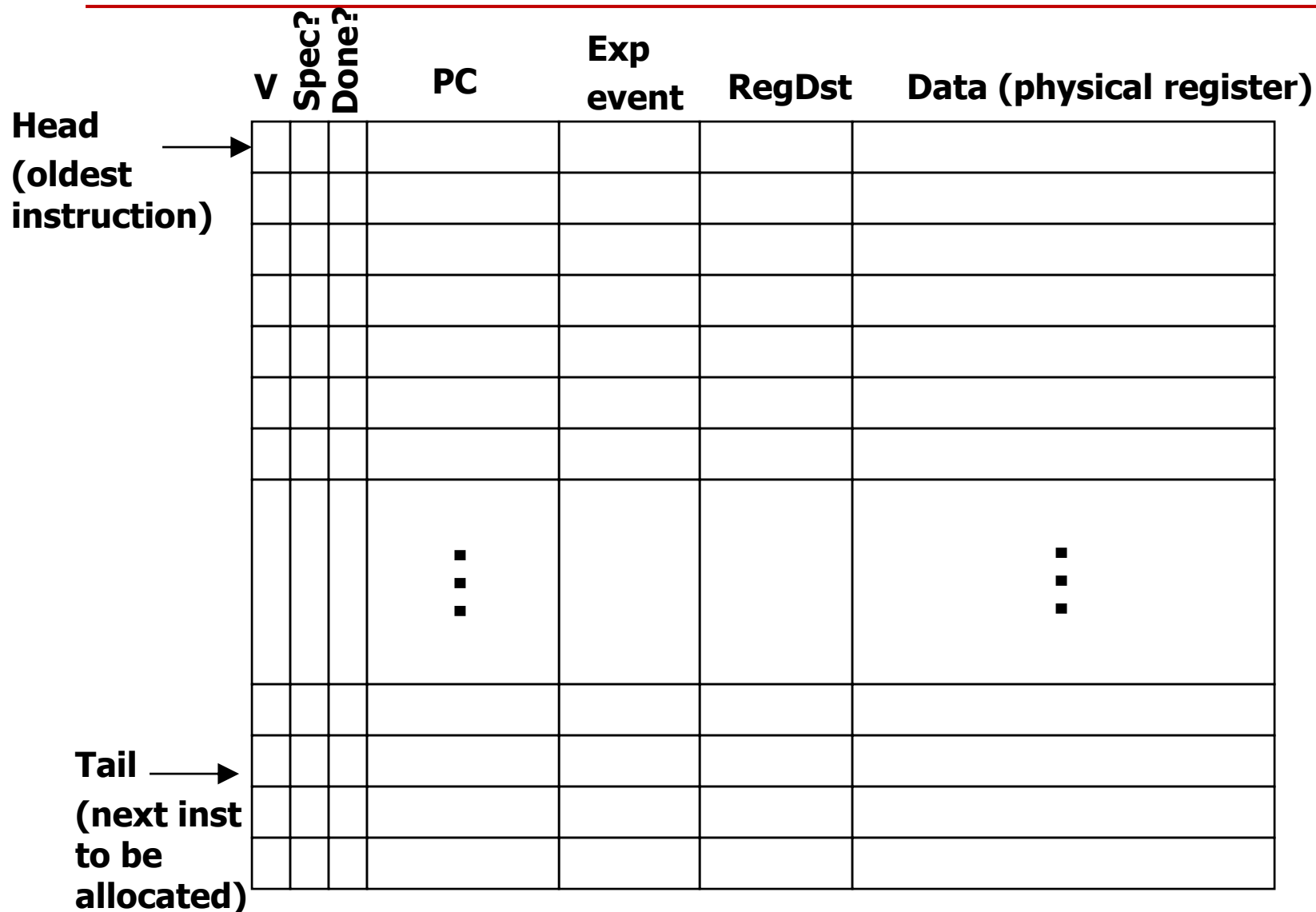
Support Precise Interrupt

- Buffer results
- Can reconstruct the scenario (state) as sequential execution
- Restart from saved PC with saved PC state

Reorder Buffer (ROB) [SmithPlezkun'85 '88]

- Architecture Register File keeps “In-order state”
- Reorder Buffer (ROB)
 - A circular buffer
 - Contains all *in-flight* instructions
 - buffers the “Lookahead state”
 - In-order allocation/deallocation with head/tail pointers
- When an exception occurs
 - Halt instruction issues
 - Revert to in-order state using RF and discard ROB results
- Also used for branch misprediction recovery
- Pentium Pro/II/III integrates physical register file within ROB
- Pentium 4 decouples ROB and physical register file

ROB (with physical registers)



Handling Precise Interrupts

[illegible]

Handling Precise Interrupts

	V	Spec?	Done?	PC	Exp event	RegDst	Data (physical register)	
	0							
Head →	1	0	0	xA004	0000	R2		$R2 = R2 * 2$
	1	0	0	xA008	0000	FR1		$FR1 = FR2 / 0.0$
	1	0	0	xA00C	0000	R3		$R3 = R3 + 1$
Tail →								
				⋮			⋮	ARF
								R1 11
								R2 2
								R3 3
								R4 4

$R2 = R2 * 2$
 $FR1 = FR2 / 0.0$
 $R3 = R3 + 1$

	ARF
R1	11
R2	2
R3	3
R4	4
R31	

Handling Precise Interrupts

	V	Spec?	Done?	PC	Exp event	RegDst	Data (physical register)	
	0							
Head →	1	0	0	xA004	0000	R2		$R2 = R2 * 2$
	1	0	0	xA008	0000	FR1		$FR1 = FR2 / 0.0$
	1	0	1	xA00C	0000	R3	4	$R3 = R3 + 1$
	1	0	0	xA010	0000	R4		$R4 = R4 * 2$
Tail →								
				⋮			⋮	ARF
								R1 11
								R2 2
								R3 3
								R4 4

	ARF
R1	11
R2	2
R3	3
R4	4
R31	

Handling Precise Interrupts

	V	Spec?	Done?	PC	Exp event	RegDst	Data (physical register)	
	0							
Head →	1	0	0	xA004	0000	R2	4	$R2 = R2 * 2$
	1	0	0	xA008	0010	FR1		$FR1 = FR2 / 0.0$
	1	0	1	xA00C	0000	R3	4	$R3 = R3 + 1$
	1	0	1	xA010	0000	R4	8	$R4 = R4 * 2$
	1	0	0	xA014	0000	FR4		$FR4 = FR4 * 2.0$
Tail →								
				⋮			⋮	

	ARF
R1	11
R2	4
R3	3
R4	4
R31	

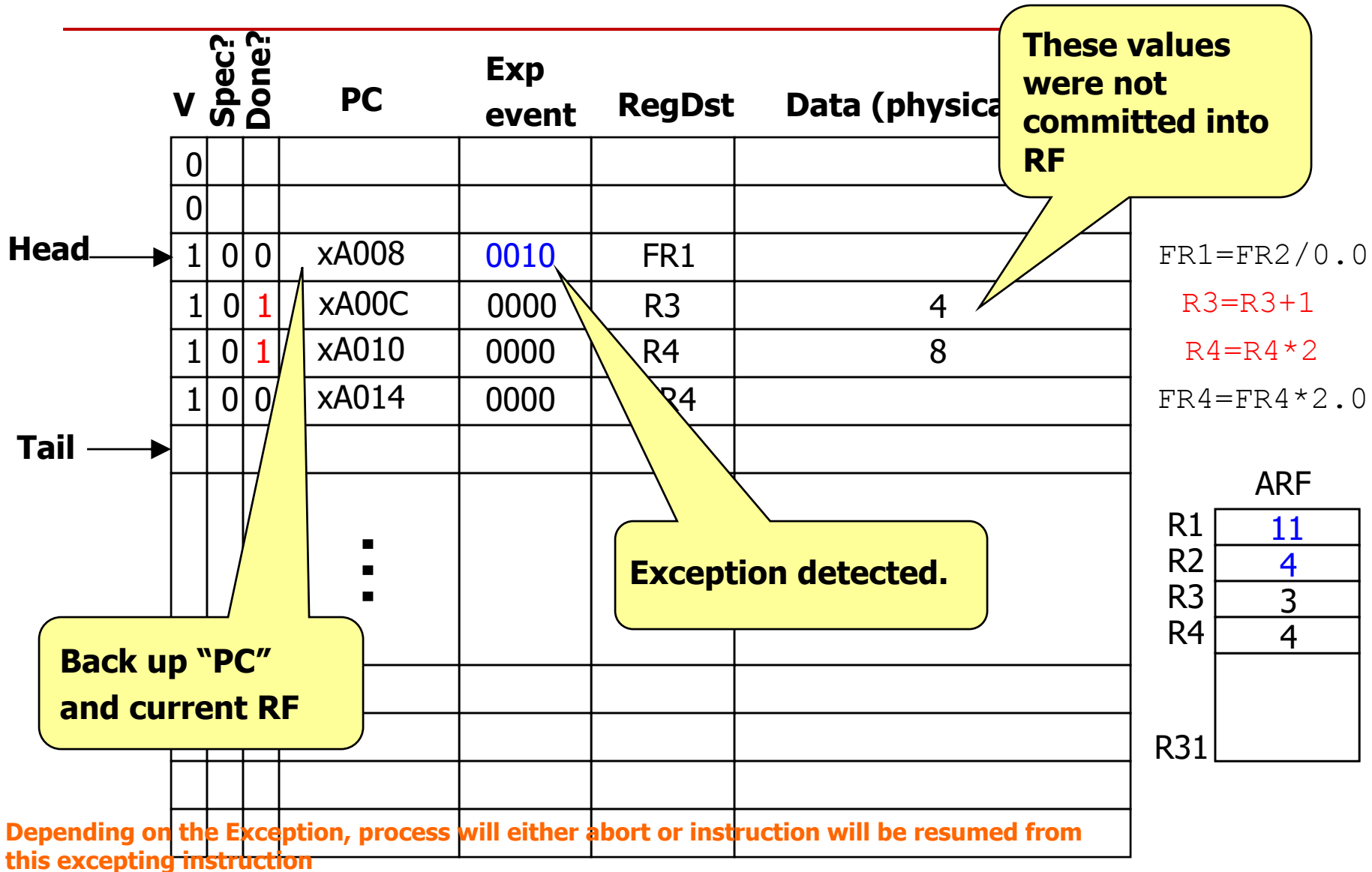
Handling Precise Interrupts

	V	Spec?	Done?	PC	Exp event	RegDst	Data (physical register)	
	0							
Head →	0	0	1	xA004	0000	R2	4	$R2 = R2 * 2$
	1	0	0	xA008	0010	FR1		$FR1 = FR2 / 0.0$
	1	0	1	xA00C	0000	R3	4	$R3 = R3 + 1$
	1	0	1	xA010	0000	R4	8	$R4 = R4 * 2$
	1	0	0	xA014	0000	FR4		$FR4 = FR4 * 2.0$
Tail →								
				⋮			⋮	

ARF

R1	11
R2	4
R3	3
R4	4
R31	

Handling Precise Interrupts



Handling Speculative Execution

	V	Spec?	Done?	PC	Exp event	RegDst	Data (physical register)	
Head →	1	0	0	xB000	0000	R1		R1=R1+10
	1	0	0	xB004	0000			BEQ R1,R0,L1
Tail →								
				⋮			⋮	

	ARF
R1	1
R2	2
R3	3
R4	4
R31	

Handling Speculative Execution

	V	Spec?	Done?	PC	Exp event	RegDst	Data (physical register)	
Head →	1	0	0	xB000	0000	R1		R1=R1+10
	1	0	0	xB004	0000			BEQ R1,R0,L1
	1	1	1	xC100	0000	R2	12	R2=R3<<2
	1	1	0	xC104	0000	R1		R1=R2*R3
	1	1	0	xC108	0000			BEQ R3,R0,L1
	1	1	1	xD2B0	0000	R1	8	R1=R7+1
Tail →								
				⋮			⋮	
BEQ R1, R0, L1 is predicted TAKEN								

ARF	
R1	1
R2	2
R3	3
R4	4
R31	

Handling Speculative Execution

	V	Spec?	Done?	PC	Exp event	RegDst	Data (phys)	
Head →	1	0	0	xB004	0000			
	1	1	1	xC100	0000	R2	12	BEQ R1, R0, L1 R2=R3<<2
	1	1	0	xC104	0000	R1		R1=R2*R3
	1	1	0	xD2AC	0000			BEQ R3, R0, L1
	1	1	1	xD2B0	0000	R1	8	R1=R7+1
Tail →								
				⋮			⋮	

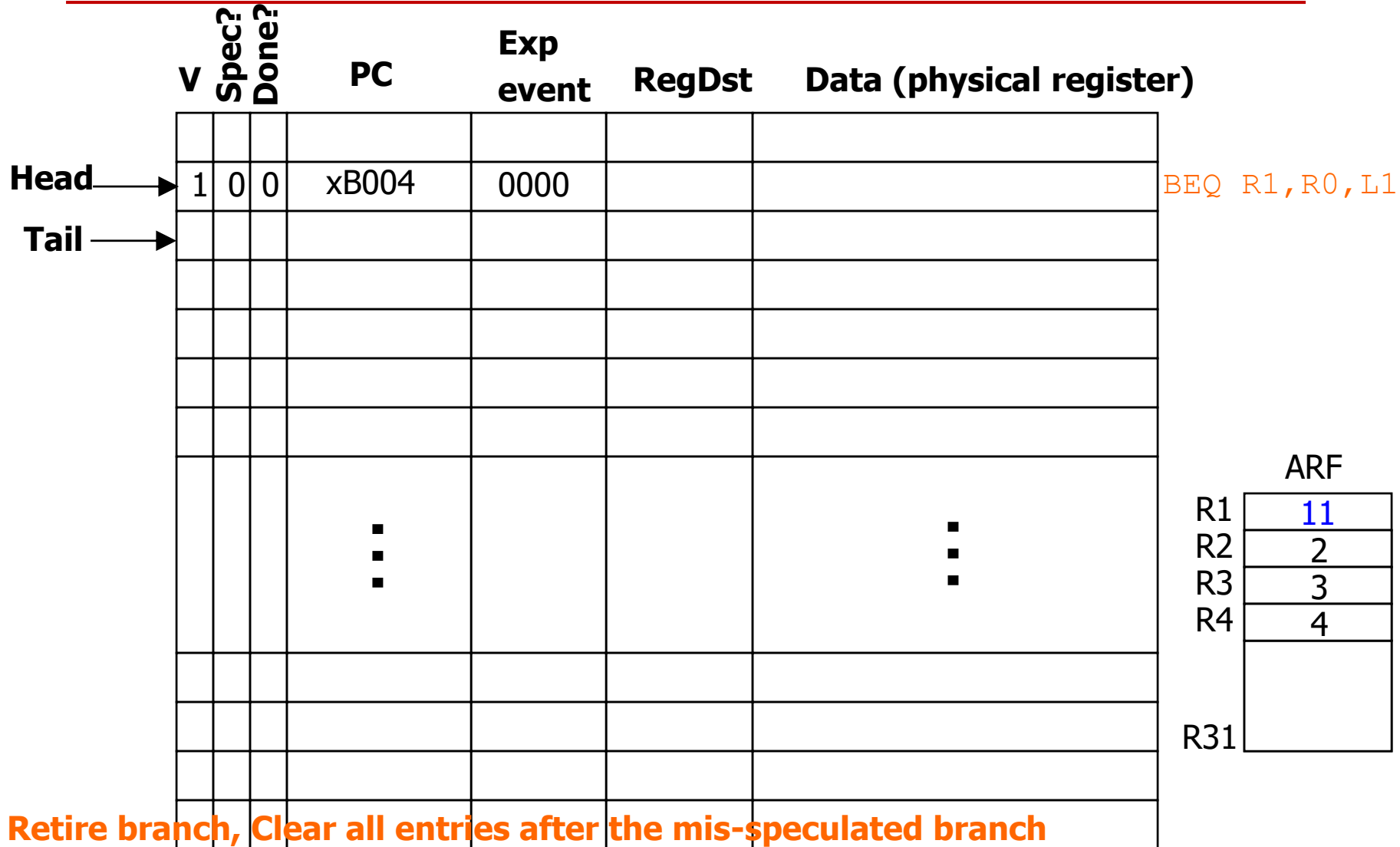
BEQ R1, R0, L1 is resolved, actually NOT TAKEN !!



ARF

R1	11
R2	2
R3	3
R4	4
R31	

Handling Speculative Execution



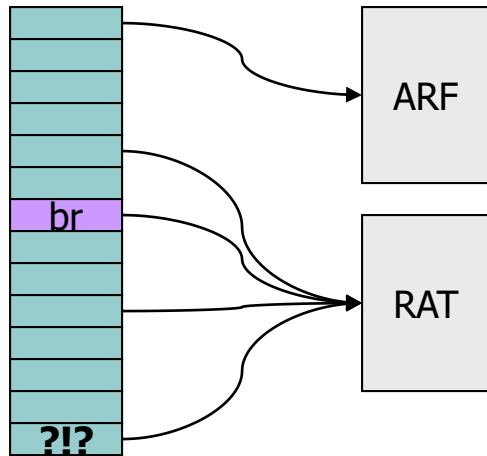
Handling Speculative Execution

	V	Spec?	Done?	PC	Exp event	RegDst	Data (physical register)	
Head →	1	0	0	xB008	0000	R2		R2=R5<<4
Tail →								
				⋮			⋮	ARF R1 11 R2 2 R3 3 R4 4

R31

Continue execution from the correct path (Fall through in this case)

RAT Recovery



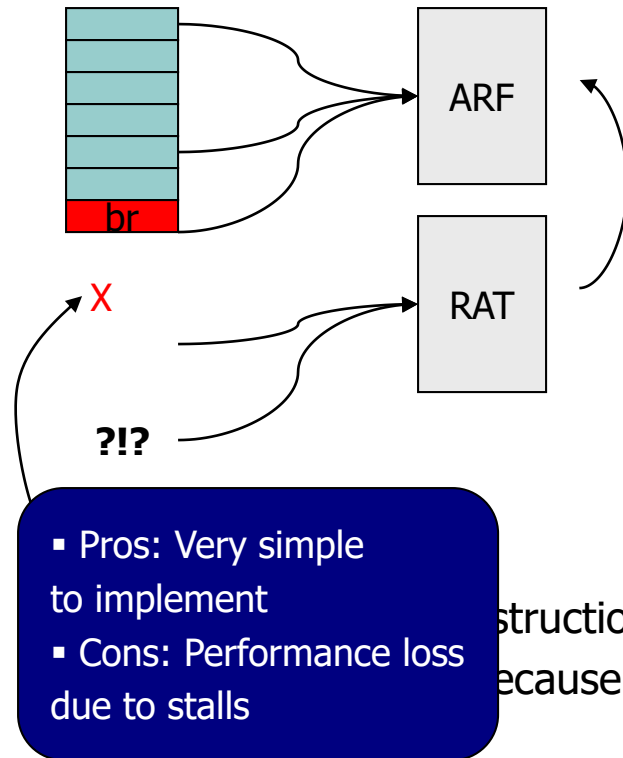
ARF state corresponds to state prior to oldest non-committed instruction

As instructions are processed, the RAT corresponds to the register mapping after the most recently renamed instruction

On a branch misprediction, wrong-path instructions are flushed from the machine

The RAT is left with an invalid set of mappings corresponding to the wrong-path instruction state

Solution: Stall and Drain



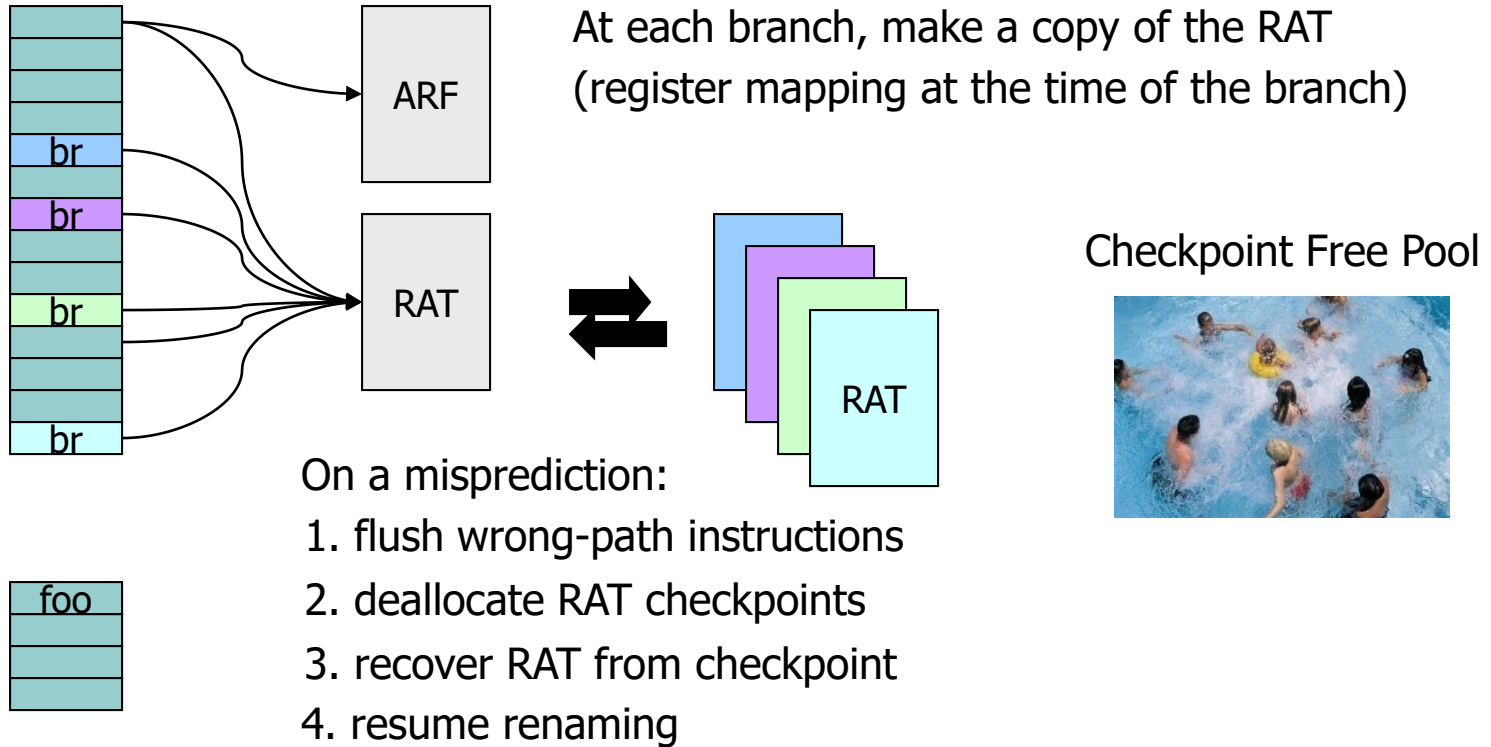
Allow all instructions to execute and commit; ARF corresponds to last committed instruction

ARF now corresponds to the state right before the next instruction to be renamed (foo)

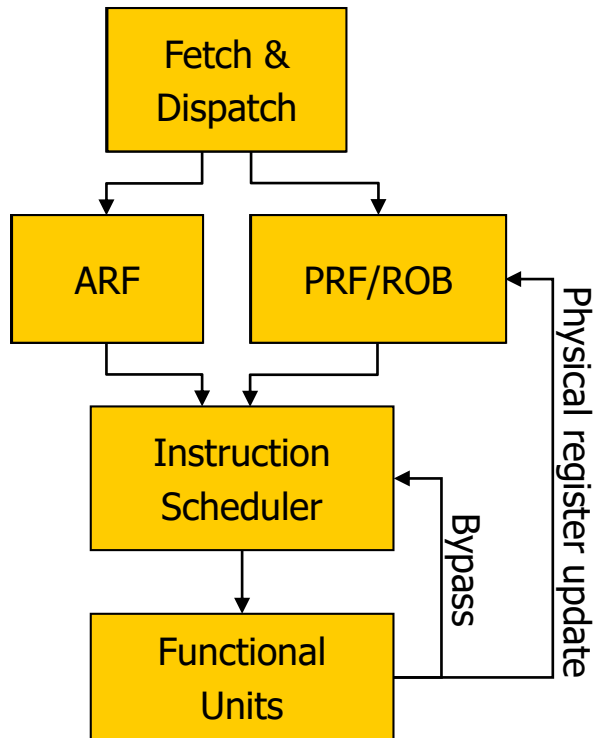
Reset RAT so that all mappings refer to the ARF

Resume renaming the new correct-path instructions from fetch; because RAT is wrong

Another Solution: Checkpointing



Modern Instruction Scheduler

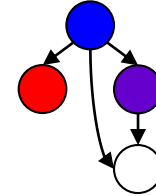
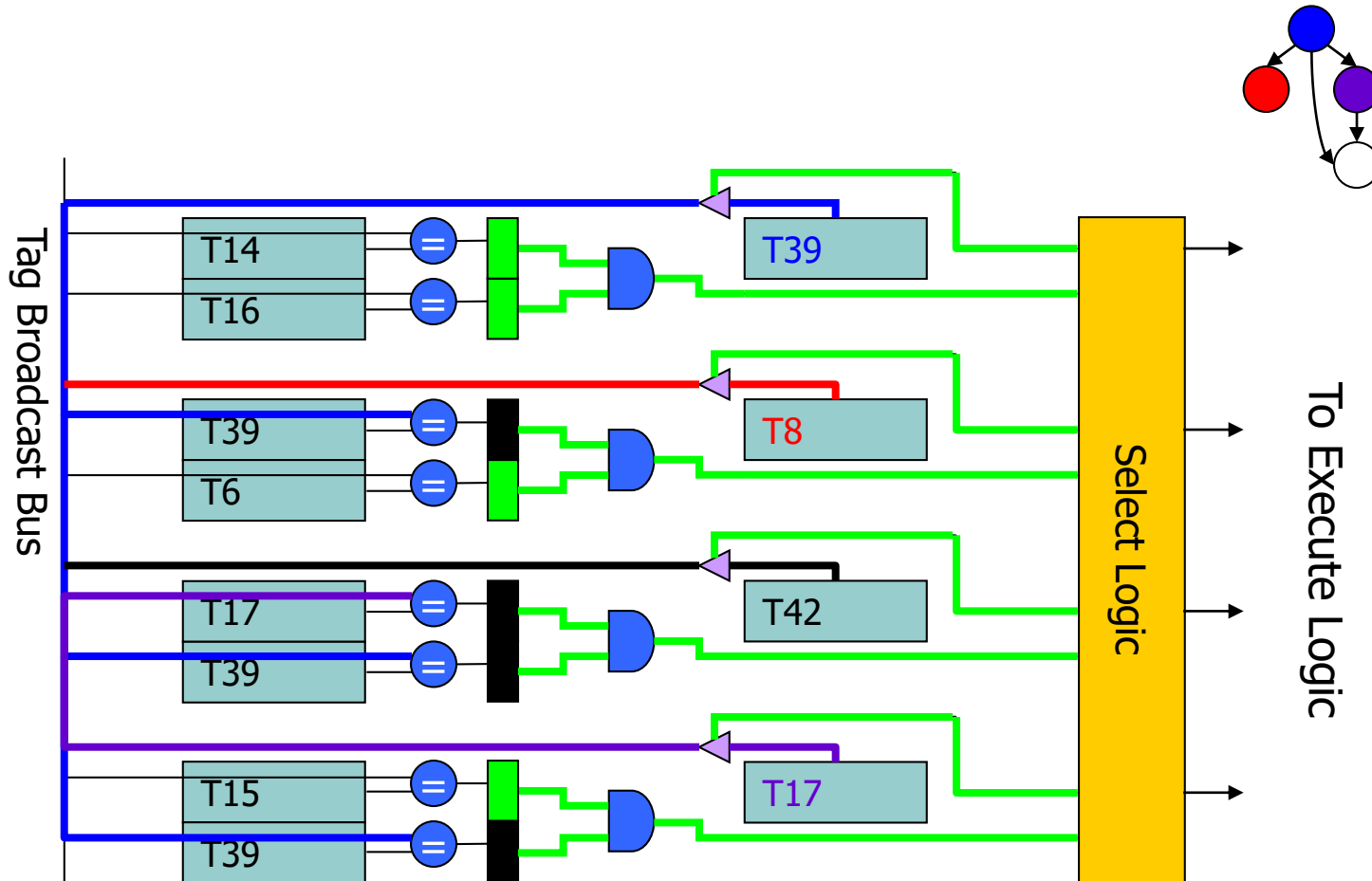


- At dispatch, instruction read all available operands from the register files and store a copy in the scheduler (Tomasulo's algorithm)
- Unavailable operands will be "captured" from the functional unit outputs (CDB broadcast)
- When ready, instructions can issue directly from the scheduler without reading additional operands from any other register files (Wakeup and select)

Instruction Scheduling: Wakeup and Select

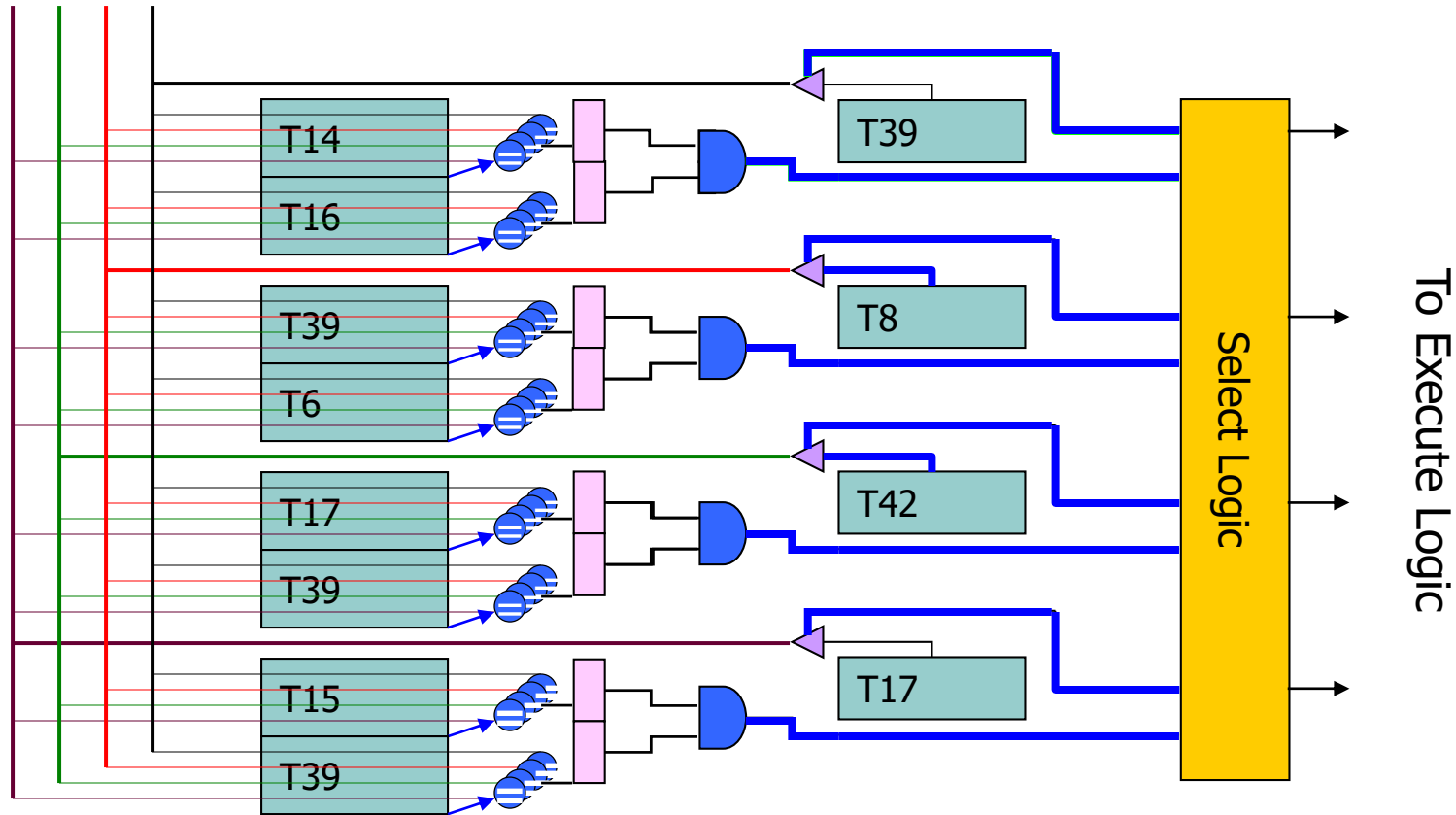
- Wakeup Logic
 - To notify the resolution of data dependency of input operands
 - Wake up instructions with zero input dependency
- Select Logic
 - Choose and fire ready instructions
 - Deal with structure hazard
- Wakeup-select is likely on the critical path
 - Associative match

Scalar Scheduler (Issue Width = 1)



Superscalar Scheduler (Issue Width = 4)

Tag Broadcast Bus [3..0]

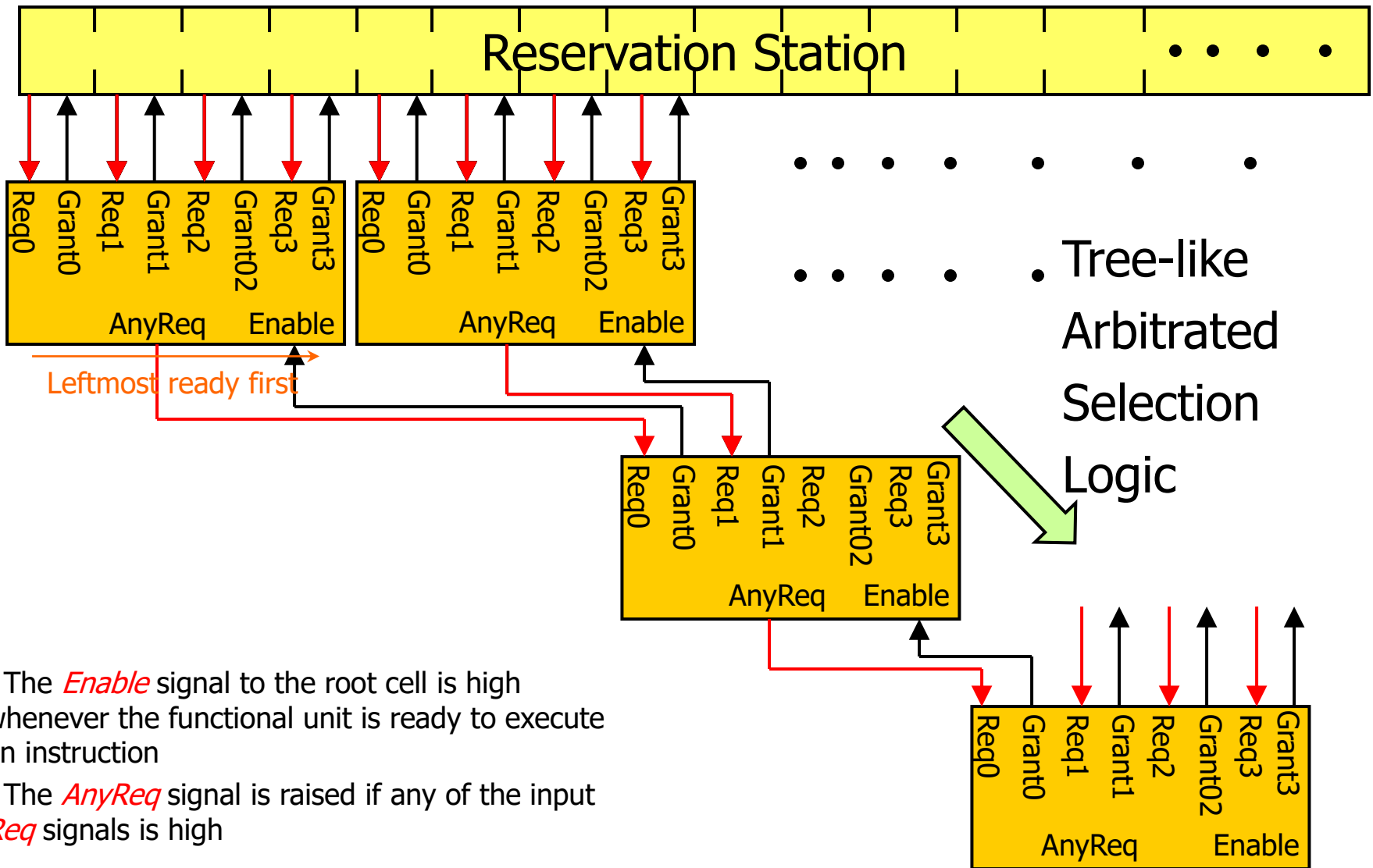


Snapshot of RS (only 4 entries shown)

Selection Logic

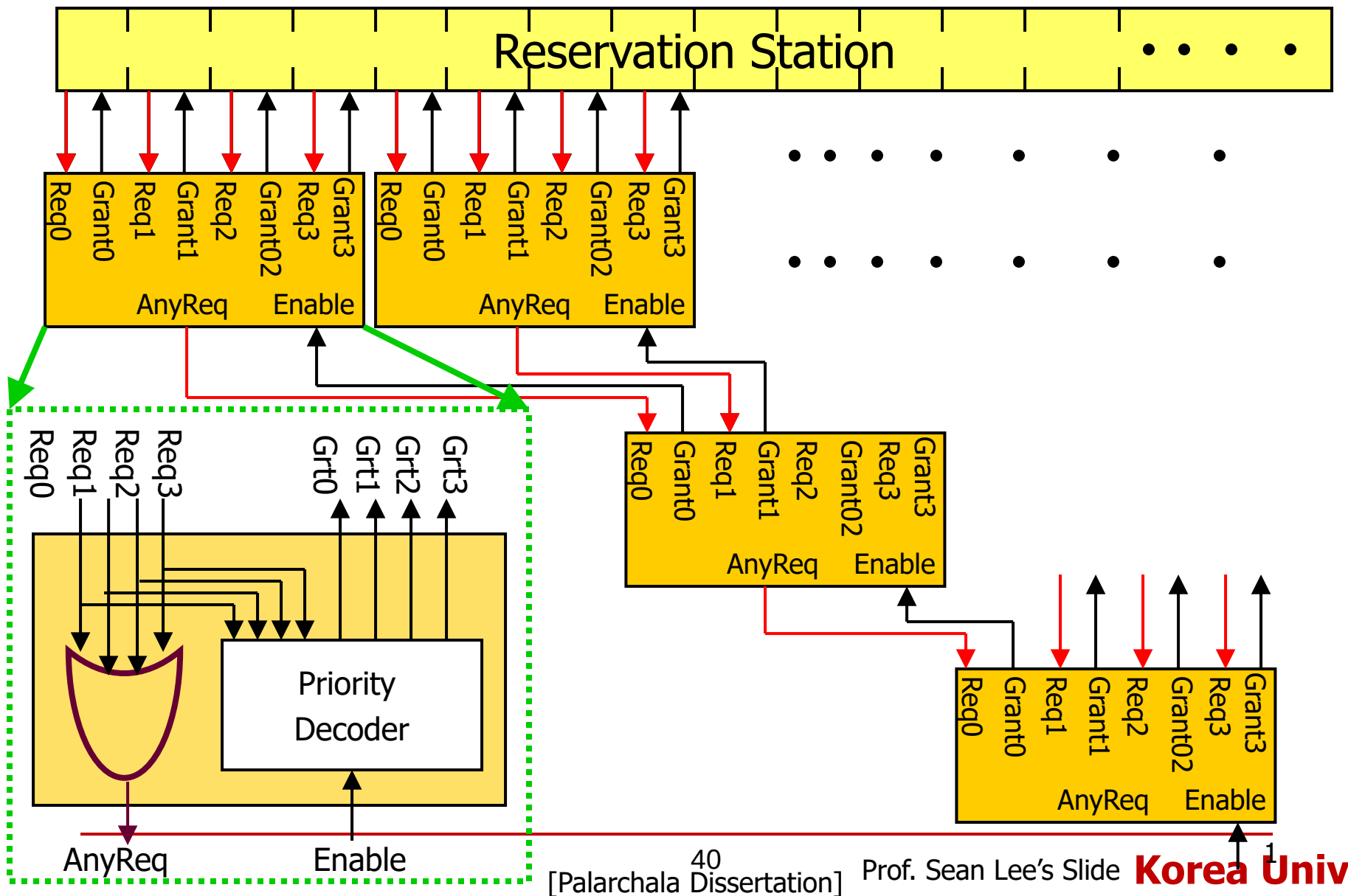
- Select ready instructions to be issued
- Goal: to reduce the height of DFG
- Methods
 - Location-based (e.g., leftmost ready first)
 - Allow simple, faster hardware
 - Oldest ready first
 - Can use location-based (in-order issue) with “compaction”
 - Compact the issue window to the left every time instructions are issued and by inserting new instructions at the right end
 - Can be slow and complex

Simple Select Logic Implementation

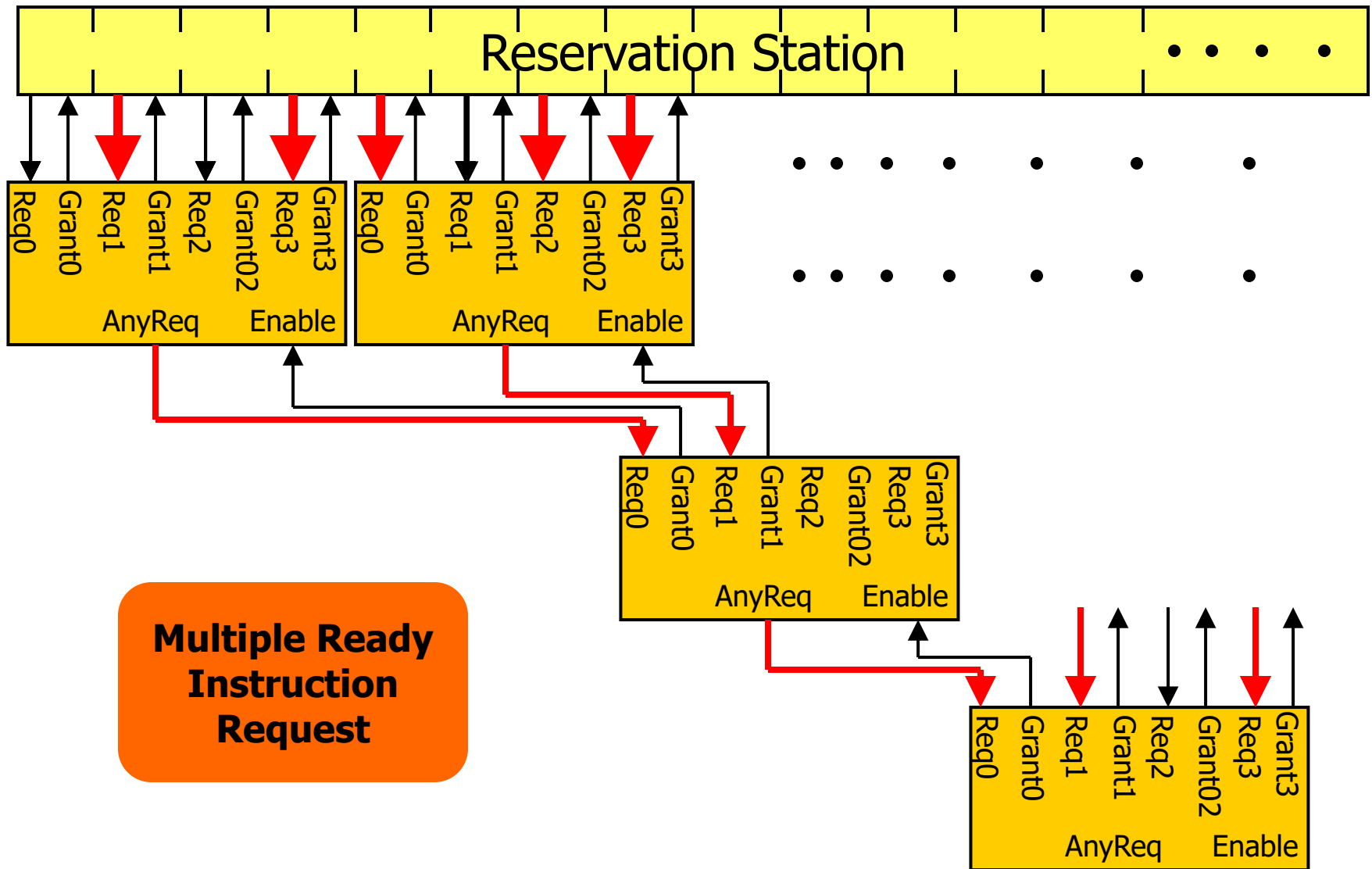


- The *Enable* signal to the root cell is high whenever the functional unit is ready to execute an instruction
- The *AnyReq* signal is raised if any of the input *Req* signals is high

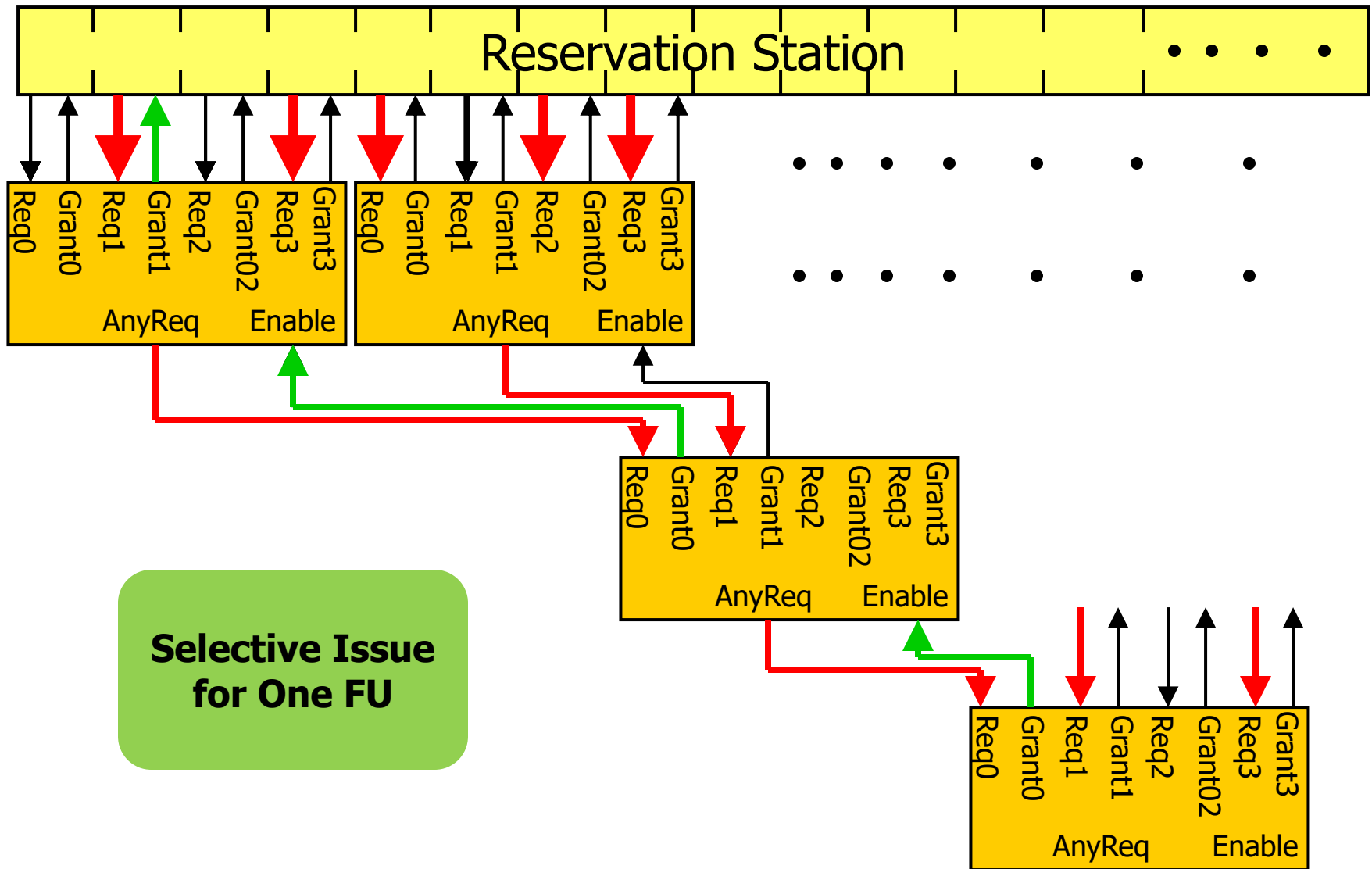
Simple Select Logic Implementation



Simple Select Logic Implementation

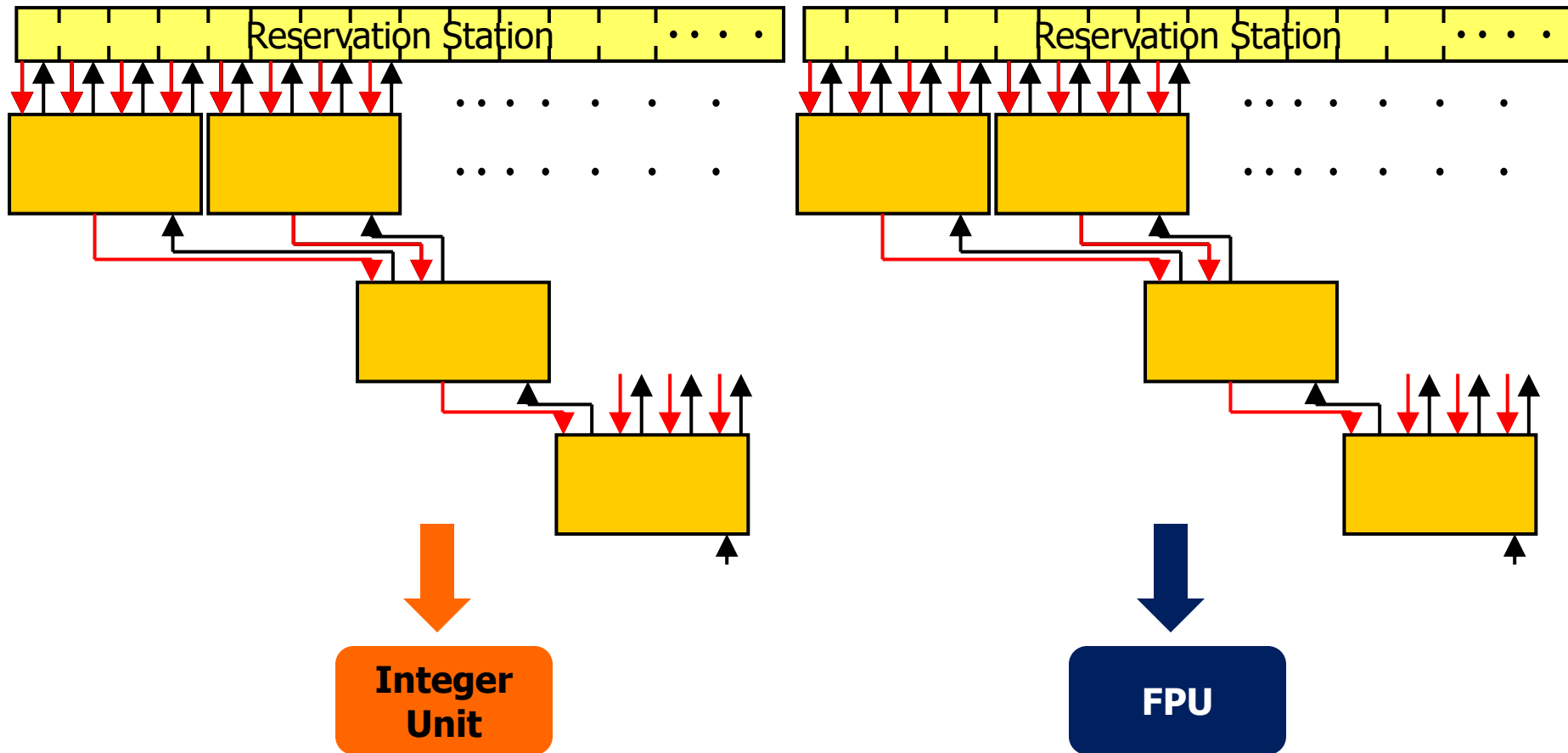


Simple Select Logic Implementation



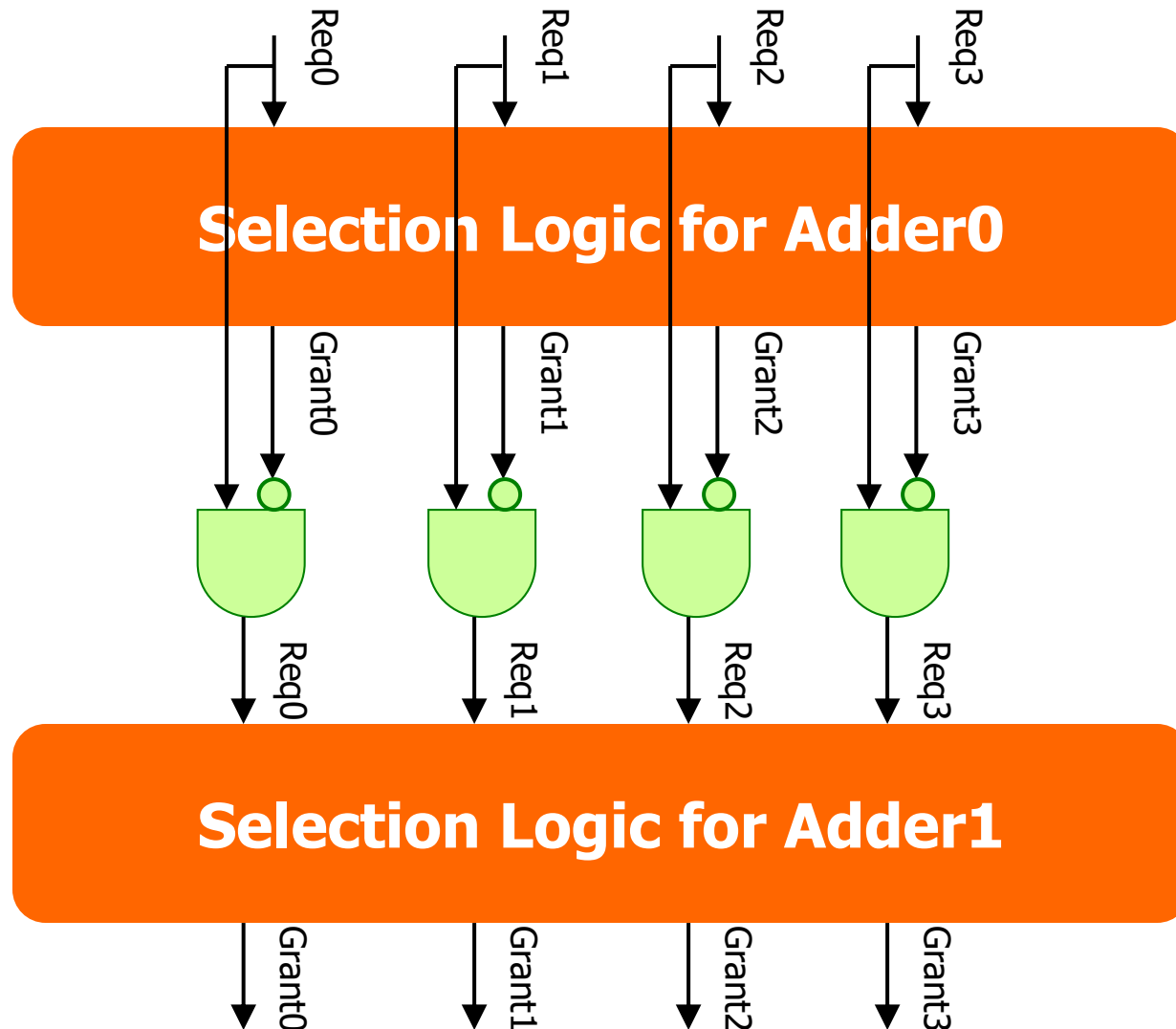
Issues to Distinctive Functional Units

Distributed Instruction Windows (e.g., MIPS R1000 or Alpha 21264)



Faster to have separate instruction schedulers for different instruction types

Dual Issues to Multiple Units (e.g., 2 Adders)



Memory Disambiguation

- Can we “undo” stores?
- Stores cannot be committed to memory until they are marked ready to retire
- Completed stores are queued and waiting in a store queue or store buffer
- Disambiguate (and resolve) memory dependency dynamically

Memory Ordering

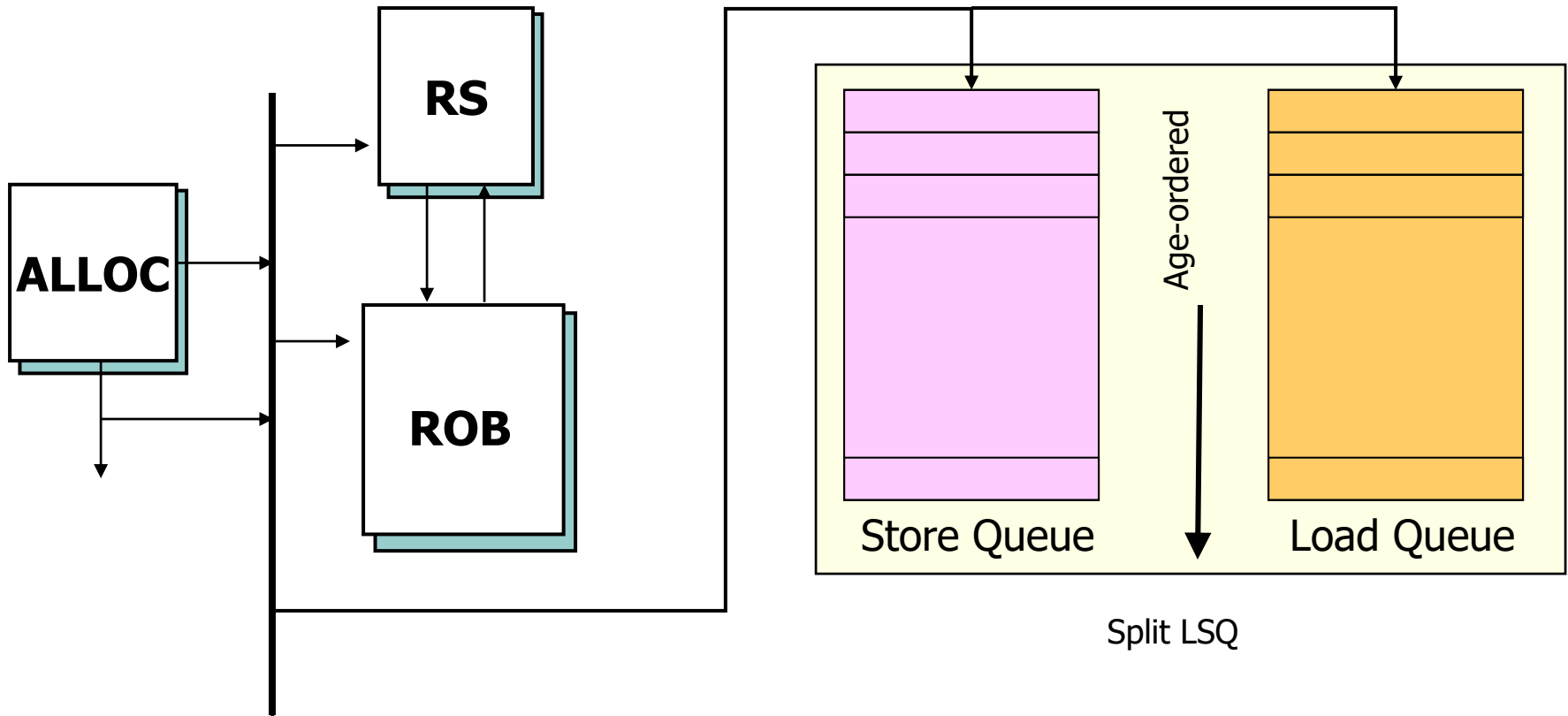
Table 2–10 Memory Reference Ordering

Source: Alpha 21264 HRM

First Instruction in Pair	Second Instruction in Pair	Reference Order
Load memory to address X	Load memory to address X	Maintained (litmus test 1)
Load memory to address X	Load memory to address Y	Not maintained
Store memory to address X	Store memory to address X	Maintained
Store memory to address X	Store memory to address Y	Maintained
Load memory to address X	Store memory to address X	Maintained
Load memory to address X	Store memory to address Y	Not maintained
Store memory to address X	Load memory to address X	Maintained
Store memory to address X	Load memory to address Y	Not maintained

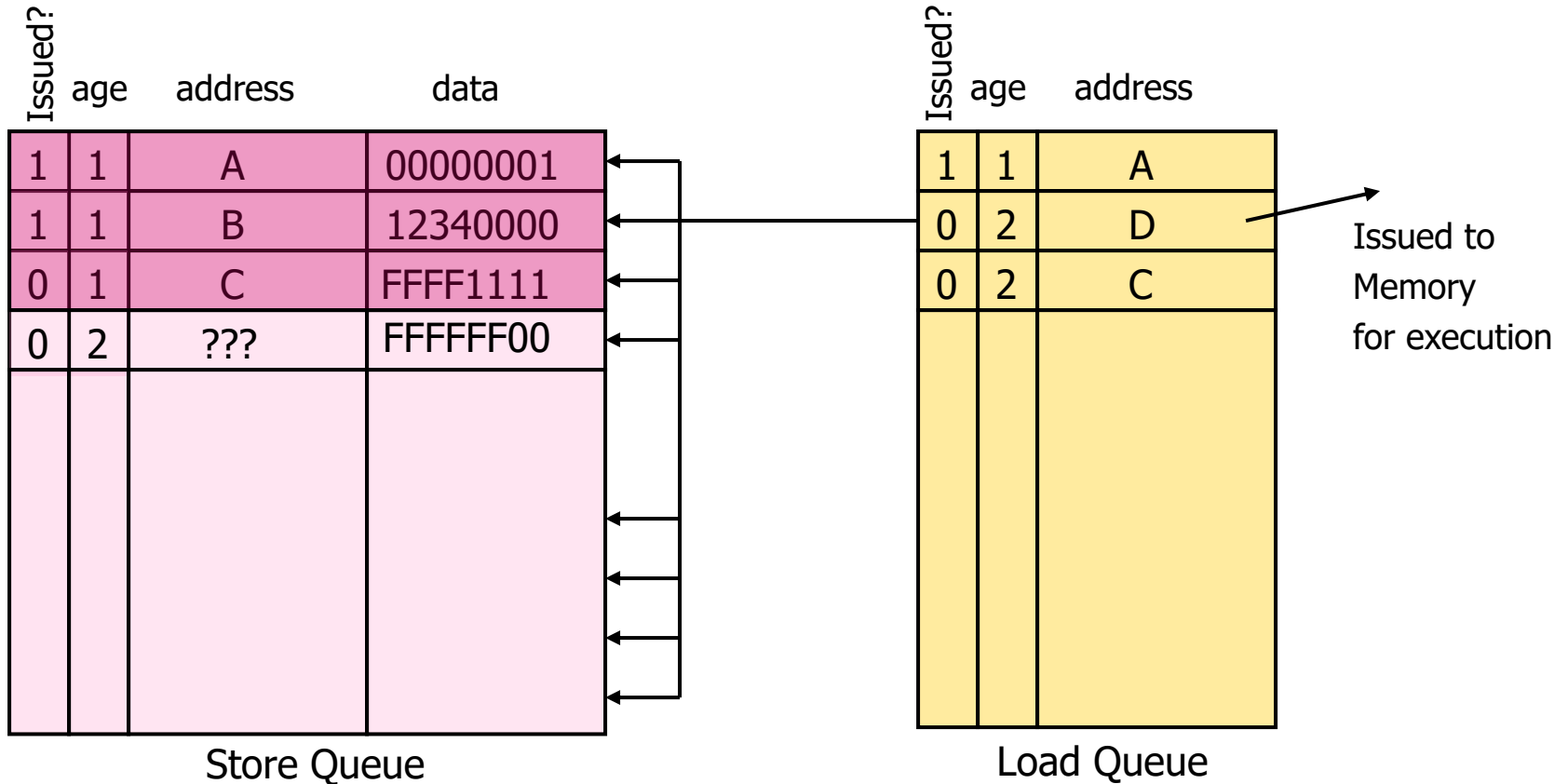
- Load X bypassing Load X violates certain memory consistency model (e.g., sequential consistency)
- Load-load order trap replays

Load Store Queue (LSQ)



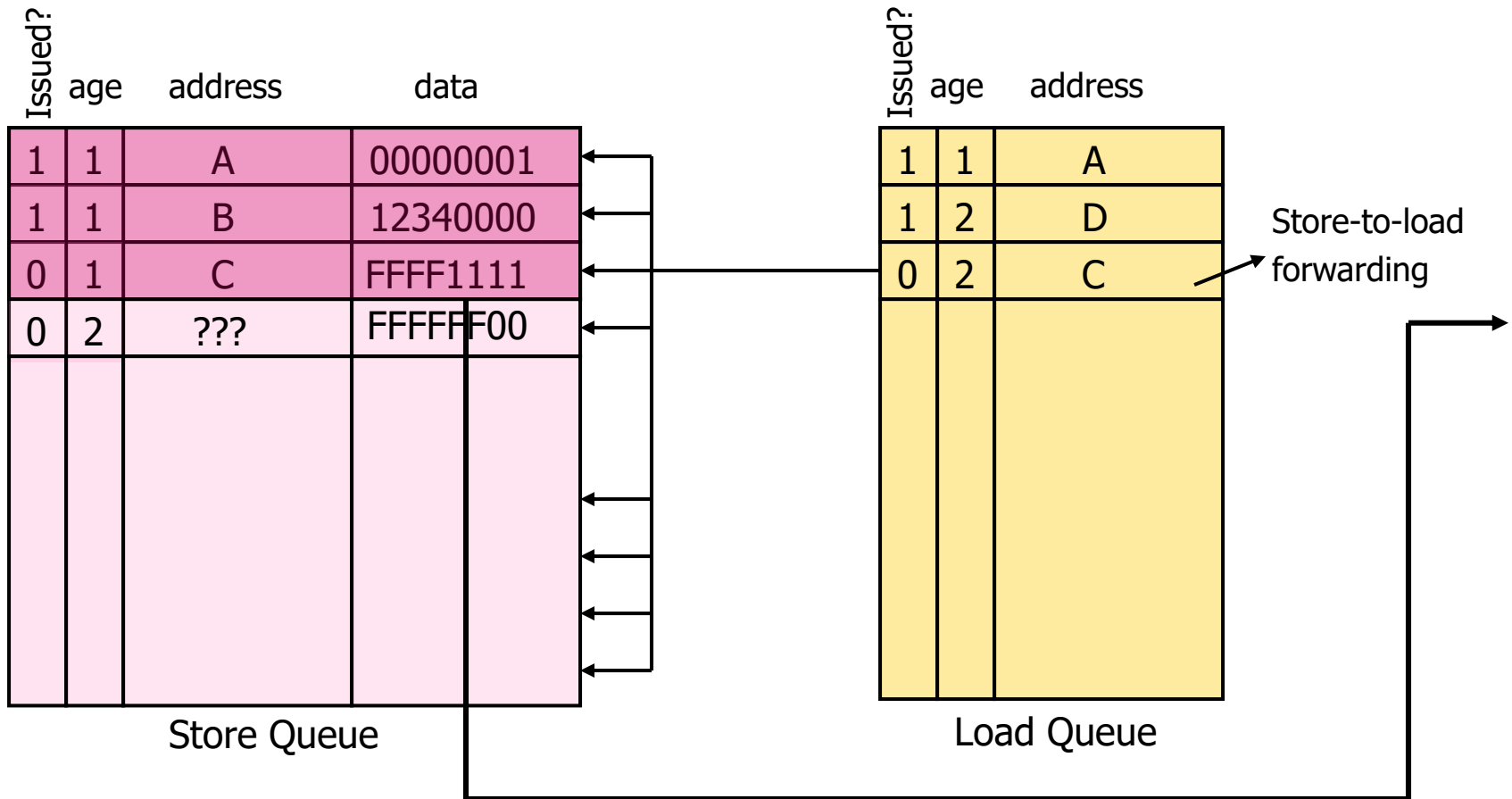
- Memory instructions are allocated into LSQ in program order
- LSQ manages memory reference ordering
- Unified LSQ vs. Split LSQ
- Sandy Bridge: 64 Load buffers, 36 Store buffers

Issuing a Load for Execution



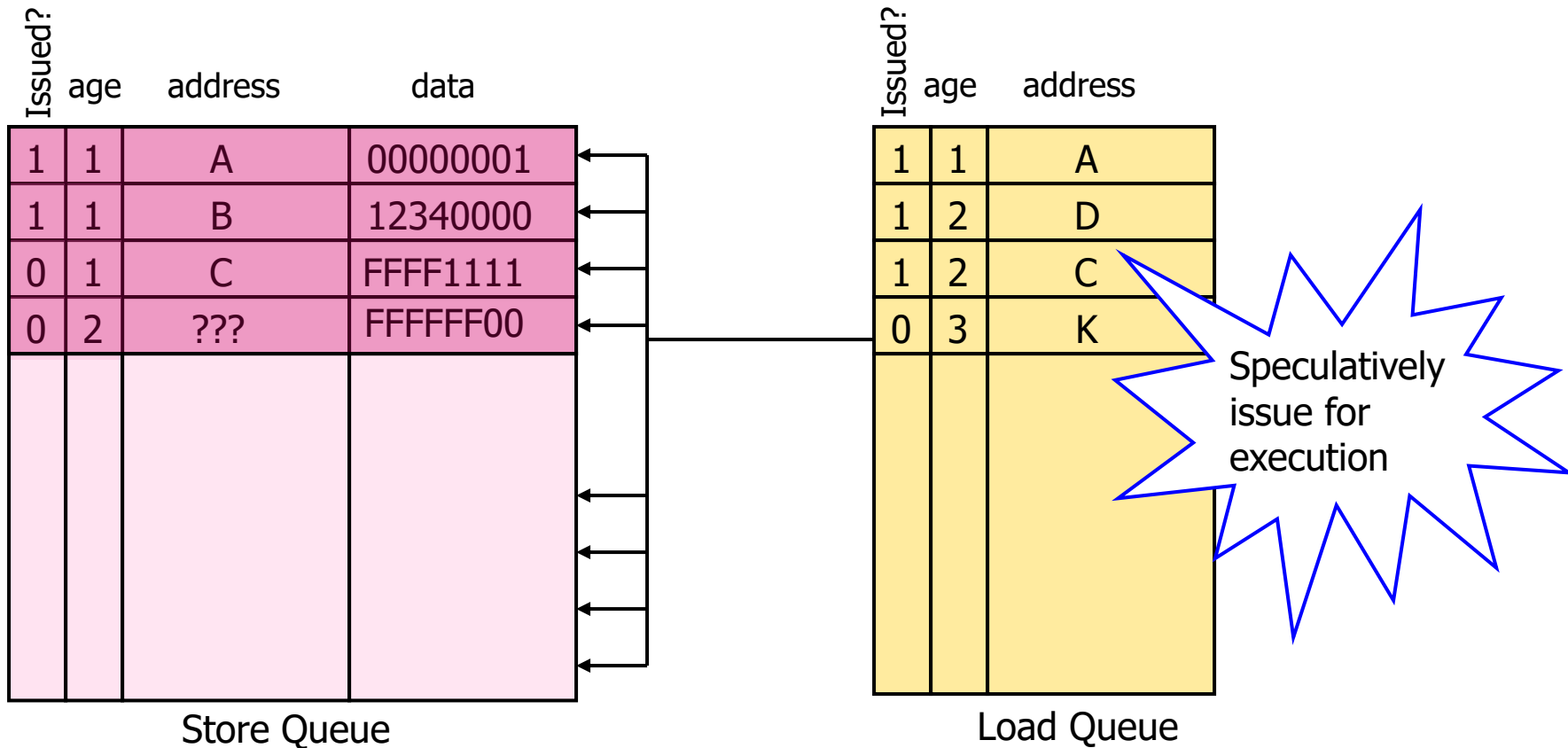
- Each load checks against older stores
 - Associative search
 - A performance issue of scalability

Issuing a Load for Execution



- Implementation dependent: comprehensive size matching can be prohibitively expensive
- Simple method: forward when a larger store (word) precedes a smaller load (half)

Issuing a Load for Execution



- Can speculatively issue loads for shortening latency (Alpha 21264, Pentium 4 (Prescott))
- Store, when address ready, checks newer loads in the Load Queue
- "Replay" needed if speculation turns out to be incorrect (e.g. Alpha's store-load replay)

Store Checks Pre-Mature Loads

Issued?	age	address	data
1	1	A	00000001
1	1	B	12340000
1	1	C	FFFF1111
0	2	K	FFFFFF00

Store Queue

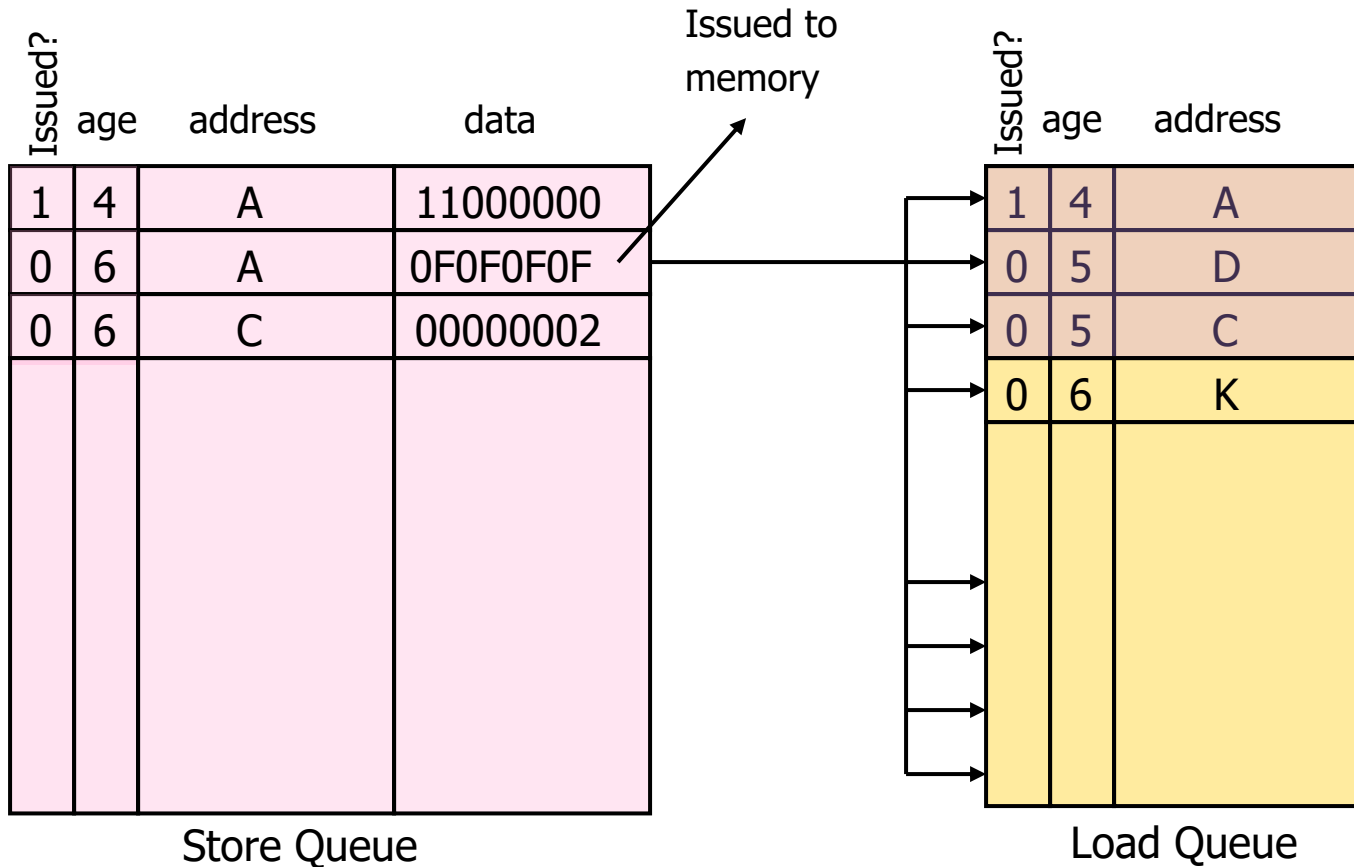
Issued?	age	address
1	1	A
1	2	D
1	2	C
1	3	K
1	3	M
1	4	P

Load Queue

Conflict detected!
Replay the load

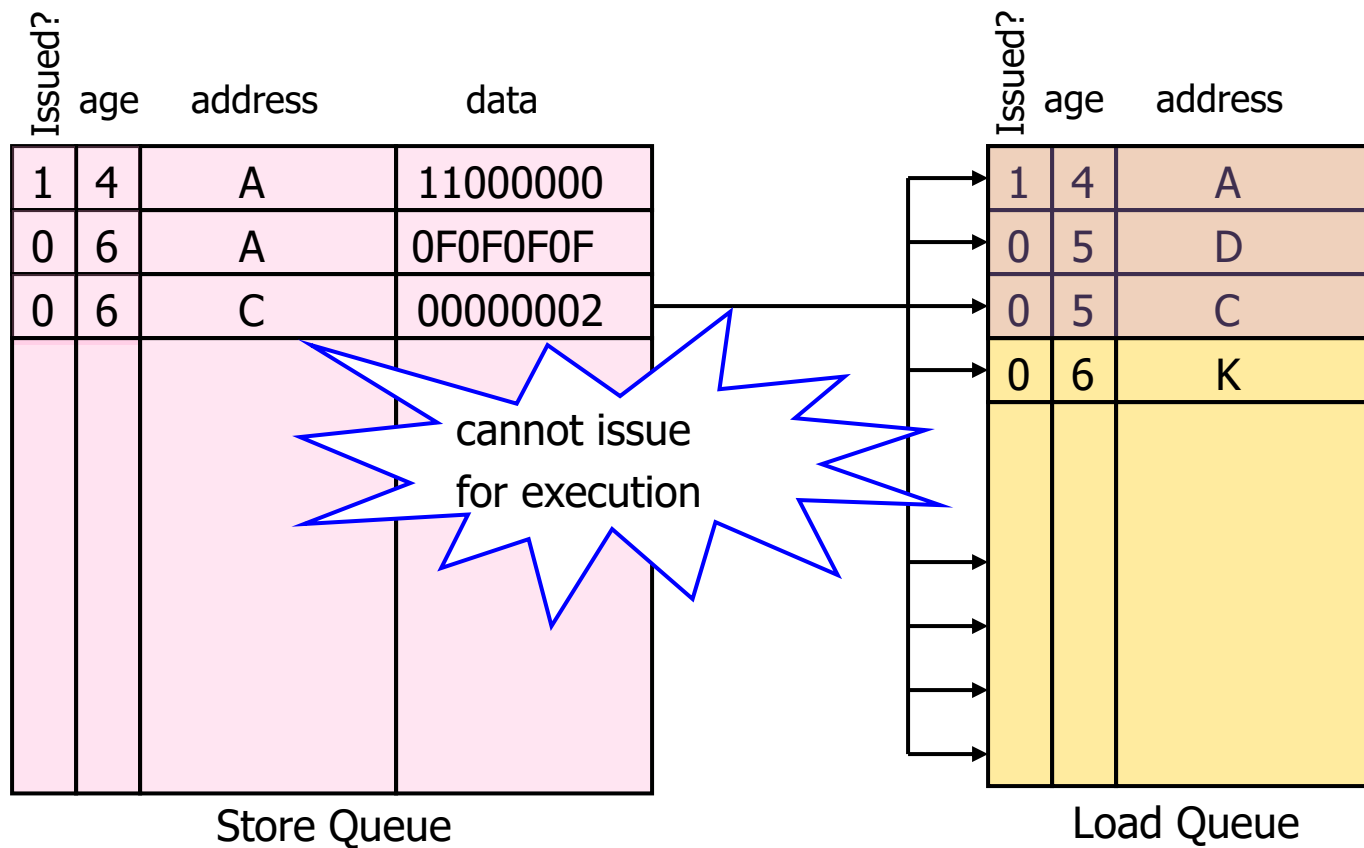
- Store, when address ready, checks newer loads in the Load Queue
 - Associative Search
- "Replay" needed if speculation turns out to be incorrect (e.g. Alpha's store-load replay)

Issuing a Store for Execution



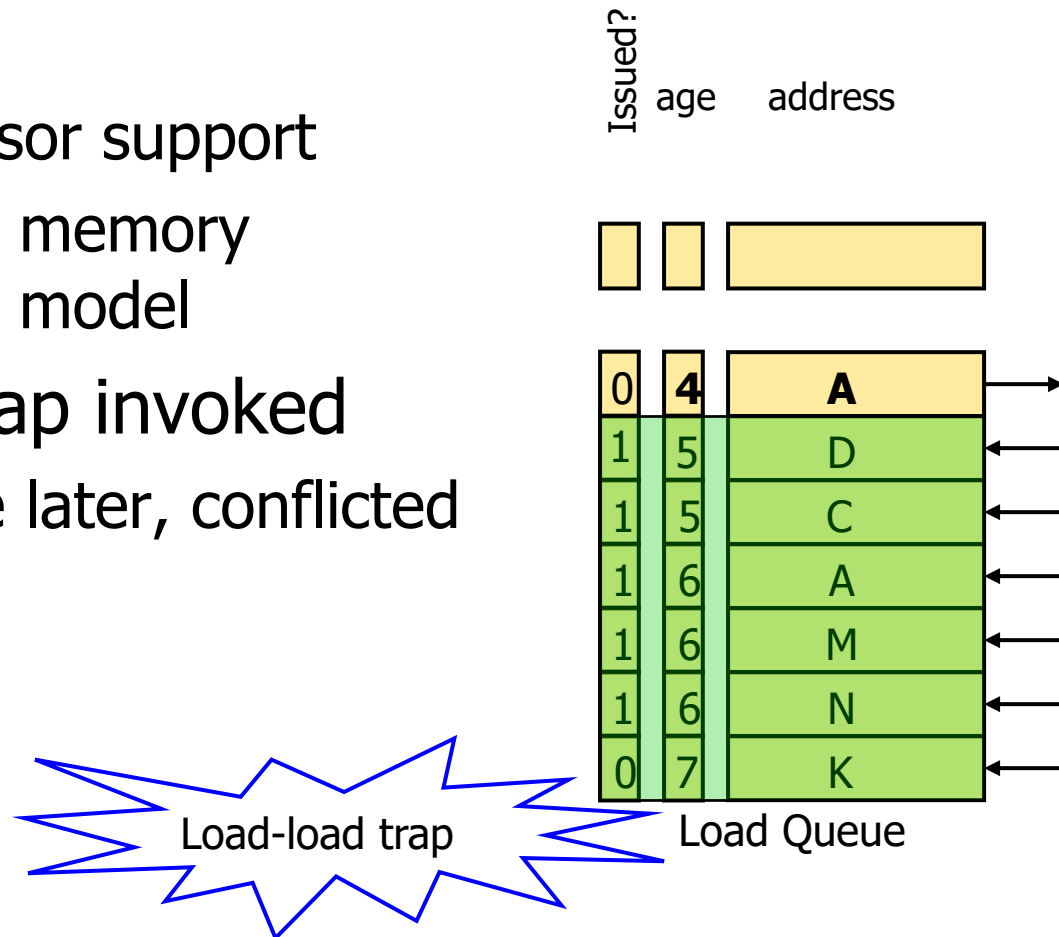
- Shown above the basic concept
- Implementation dependent
 - Not allow store bypassing load, since it has little impact on performance
 - Perform associative search

Issuing a Store for Execution



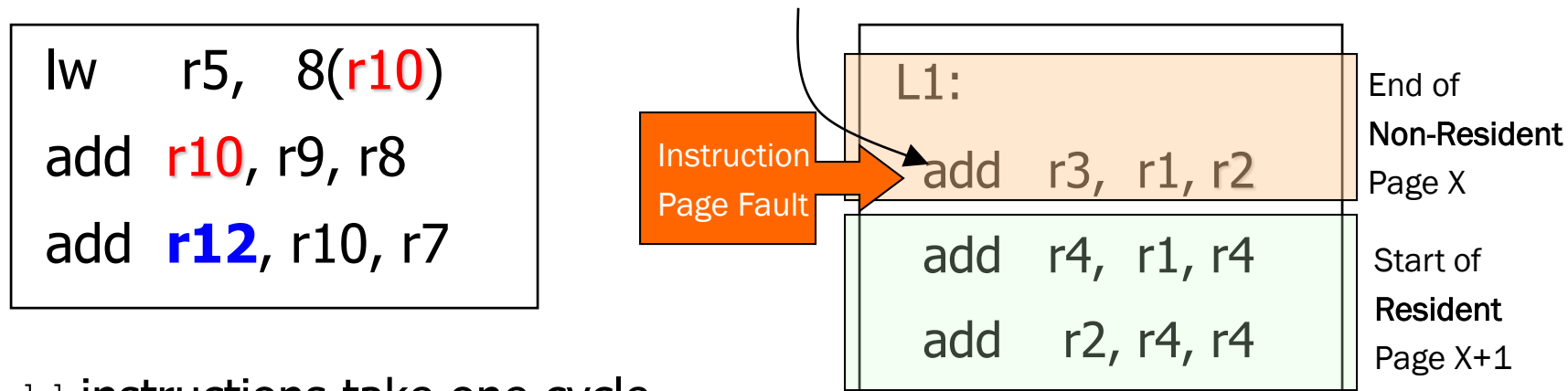
Load-Load Ordering

- Needed for
 - Multiprocessor support
 - Maintaining memory consistency model
- Load-load trap invoked
 - Trap on the later, conflicted instructions
 - Replay



Backup Slides

Issue with Imprecise Interrupt



- add instructions take one cycle
- E.g.,
 - Load (left side) induces a “data page fault”;
 - Add (right side) induces an “instruction page fault”
- If out-of-order completion is allowed
 - r10, r12, (or r2, r4) ... will be modified
 - Wrong values will be used by the re-issued load
- Interrupt classes
 - Program interrupts (exceptions or traps)
 - External interrupts (asynchronous)