CS 520 Computer Architecture

06. Instruction Level Parallelism (2)

Early Dynamically Scheduled Machines

- **CDC-6600**
- Centralized control: "CDC scoreboard"
- Many replicated functional units
- All values pass through the register file
- Stall on WAR/WAW hazards

- **IBM 360/91 (Tomasulo's algorithm)**
- Distributed control: reservation stations
- Several fully-pipelined functional units (equivalent to replicating functional units)
- Values broadcast to waiting instructions and register files in parallel (via the common data bus)
- Introduced register renaming: handles WAR/WAW hazards without stalling

CDC-6600 (DLX version) (1)

- **Fetch**
- **Dispatch**

- Check for structural and WAW hazards
  - ▸ Structural hazard: stall in dispatch stage if FUs are busy
  - ▸ WAW: stall in dispatch stage if an outstanding instruction in the scoreboard writes the same destination register
- Enter instruction into scoreboard: determine data dependences
- Route instruction to a free FU, where it waits until data operands are available

- **Issue**

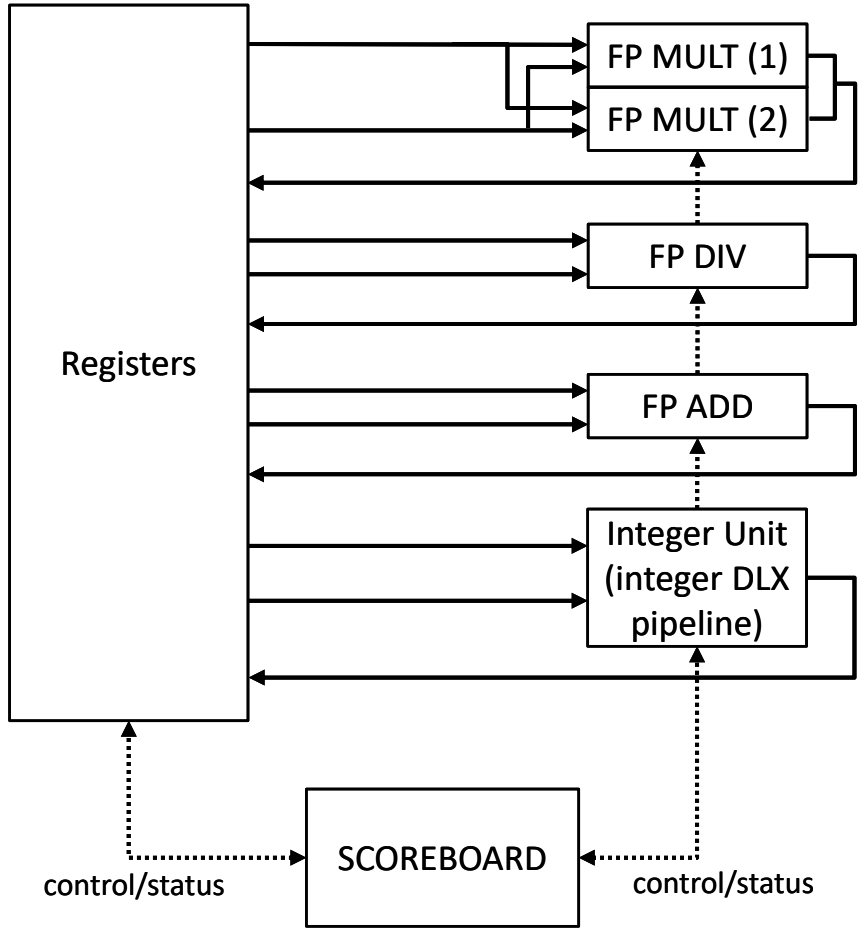
- Wait for operands to become ready
- Scoreboard signals when operands ready
  - ▸ Instruction reads registers from the register file and issues to FUs for execution

- **Execute**
- **Write result**

- Check for WAR hazard: stall if an outstanding, prior instruction in the scoreboard reads the same register being written, and the read has not yet taken place

CDC-6600 (DLX version) (2)

Manufacturer	Control Data Corporation
Designer	Seymour Cray
Release date	September 1964
Units sold	100+
Price	US$2,370,000 (equivalent to $19,540,000 in 2019)
Casing	
Dimensions	Height : 2,000 mm (79 in) Cabinet width: 810 mm (32 in) Cabinet length : 1,710 mm (67 in) Width overall : 4,190 mm (165 in)
Weight	about 12,000 lb (6.0 short tons; 5.4 t)
Power	30 kW @ 208 V 400 Hz
System	
Operating system	SCOPE, KRONOS
CPU	60-bit processor @ 10 MHz
Memory	Up to 982 kilobytes (131000 x 60 bits)
MIPS	2 MIPS
Predecessor	CDC 1604
Successor	CDC 7600



CDC-6600 Scoreboard

- **Three data structures**

- Instruction status
  - Which stage is the instruction in
- Functional unit status
  - Busy - FU is busy executing another instruction
  - Op - what instruction is the FU busy with
  - F_i - destination register
  - F_j, F_k - source registers
  - Q_j, Q_k - function units producing source registers
  - R_j, R_k - flags indicating source registers are ready
- Register result status
  - Which FU is going to write each register

Running example

- **Example used for CDC**

- LD F6, 34 (R2)
- LD F2, 45 (R3)
- MULTD F0, F2, F4
- SUBD F8, F6, F2
- DIVD F10, F0, F6
- ADDD F6, F8, F2

CDC-6600 Example (1)

- Initial status

[In-order][Out-of-order]

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)				
LD F2, 45 (R3)				
MULTD F0, F2, F4				
SUBD F8, F6, F2				
DIVD F10, F0, F6				
ADDD F6, F8, F2				

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	no								
MULT1	no								
MULT2	no								
ADD	no								
DIV	no								

Register Result Status

F0	F2	F4	F6	F8	F10	F12

CDC-6600 Example (2)

- LD(1) is dispatched.

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)	x			
LD F2, 45 (R3)				
MULTD F0, F2, F4				
SUBD F8, F6, F2				
DIVD F10, F0, F6				
ADDD F6, F8, F2				

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	yes	LD	F6	R2				yes	
MULT1	no								
MULT2	no								
ADD	no								
DIV	no								

Register Result Status

F0	F2	F4	F6	F8	F10	F12
			Integer			

CDC-6600 Example (3)

- LD(1) writes result. LD(2) is dispatched.

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)	x	x	x	x
LD F2, 45 (R3)	x			
MULTD F0, F2, F4				
SUBD F8, F6, F2				
DIVD F10, F0, F6				
ADDD F6, F8, F2				

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	yes	LD	F2	R3				yes	
MULT1	no								
MULT2	no								
ADD	no								
DIV	no								

Register Result Status

F0	F2	F4	F6	F8	F10	F12
	Integer					

CDC-6600 Example (4)

- **LD(2) is issued. MULTD is dispatched.**

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)	x	x	x	x
LD F2, 45 (R3)	x	x		
MULTD F0, F2, F4	x			
SUBD F8, F6, F2				
DIVD F10, F0, F6				
ADDD F6, F8, F2				

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	yes	LD	F2	R3				yes	
MULT1	yes	MULTD	F0	F2	F4	Integer		no	yes
MULT2	no								
ADD	no								
DIV	no								

Register Result Status

F0	F2	F4	F6	F8	F10	F12
MULT1	Integer					

CDC-6600 Example (5)

- LD(2) is executed. SUBD is dispatched

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)	x	x	x	x
LD F2, 45 (R3)	x	x	x	
MULTD F0, F2, F4	x			
SUBD F8, F6, F2	x			
DIVD F10, F0, F6				
ADDD F6, F8, F2				

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	yes	LD	F2	R3				yes	
MULT1	yes	MULTD	F0	F2	F4	Integer		no	yes
MULT2	no								
ADD	yes	SUBD	F8	F6	F2		Integer	yes	no
DIV	no								

Register Result Status

F0	F2	F4	F6	F8	F10	F12
MULT1	Integer			ADD		

CDC-6600 Example (6)

- **DIVD is dispatched**

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)	x	x	x	x
LD F2, 45 (R3)	x	x	x	
MULTD F0, F2, F4	x RAW (F2)			
SUBD F8, F6, F2	x RAW (F2)			
DIVD F10, F0, F6	x RAW (F0)			
ADDD F6, F8, F2				

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	yes	LD	F2	R3				yes	
MULT1	yes	MULTD	F0	F2	F4	Integer		no	yes
MULT2	no								
ADD	yes	SUBD	F8	F6	F2		Integer	yes	no
DIV	yes	DIVD	F10	F0	F6	MULT1		no	yes

Register Result Status

F0	F2	F4	F6	F8	F10	F12
MULT1	Integer			ADD	DIV	

CDC-6600 Example (7)

- **LD(2) writes result.**

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)	x	x	x	x
LD F2, 45 (R3)	x	x	x	x
MULTD F0, F2, F4	x			
SUBD F8, F6, F2	x			
DIVD F10, F0, F6	x			
ADDD F6, F8, F2				

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	yes	LD	F2	R3				yes	
MULT1	yes	MULTD	F0	F2	F4			yes	yes
MULT2	no								
ADD	yes	SUBD	F8	F6	F2			yes	yes
DIV	yes	DIVD	F10	F0	F6	MULT1		no	yes

Register Result Status

F0	F2	F4	F6	F8	F10	F12
MULT1				ADD	DIV	

CDC-6600 Example (8)

- **MULTD and SUBD are issued.**

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)	x	x	x	x
LD F2, 45 (R3)	x	x	x	x
MULTD F0, F2, F4	x	x		
SUBD F8, F6, F2	x	x		
DIVD F10, F0, F6	x			
ADDD F6, F8, F2				

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	yes	LD	F2	R3				yes	
MULT1	yes	MULTD	F0	F2	F4			yes	yes
MULT2	no								
ADD	yes	SUBD	F8	F6	F2			yes	yes
DIV	yes	DIVD	F10	F0	F6	MULT1		no	yes

Register Result Status

F0	F2	F4	F6	F8	F10	F12
MULT1				ADD	DIV	

CDC-6600 Example (9)

- **MULTD is executed. SUBD updates the register.**

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)	x	x	x	x
LD F2, 45 (R3)	x	x	x	x
MULTD F0, F2, F4	x	x	x	
SUBD F8, F6, F2	x	x	x	x
DIVD F10, F0, F6	x			
ADDD F6, F8, F2				

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	yes	LD	F2	R3				yes	
MULT1	yes	MULTD	F0	F2	F4			yes	yes
MULT2	no								
ADD	no								
DIV	yes	DIVD	F10	F0	F6	MULT1		no	yes

Register Result Status

F0	F2	F4	F6	F8	F10	F12
MULT1					DIV	

CDC-6600 Example (10)

- **ADDD is dispatched.**

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)	x	x	x	x
LD F2, 45 (R3)	x	x	x	x
MULTD F0, F2, F4	x	x	x	
SUBD F8, F6, F2	x	x	x	x
DIVD F10, F0, F6	x			
ADDD F6, F8, F2	x			

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	yes	LD	F2	R3				yes	
MULT1	yes	MULTD	F0	F2	F4			yes	yes
MULT2	no								
ADD	yes	ADDD	F6	F8	F2			yes	yes
DIV	yes	DIVD	F10	F0	F6	MULT1		no	yes

Register Result Status

F0	F2	F4	F6	F8	F10	F12
MULT1			ADD		DIV	

CDC-6600 Example (11)

- **MULTD about to write result**

Instruction Status

	DISPATCH	ISSUE	EXECUTE	WRITE RESULT
LD F6, 34 (R2)	x	x	x	x
LD F2, 45 (R3)	x	x	x	x
MULTD F0, F2, F4	x	x	x finishing	
SUBD F8, F6, F2	x	x	x	x
DIVD F10, F0, F6	x RAW (F0)			
ADDD F6, F8, F2	x	x	x WAR (F6)	

Functional Unit Status

FU	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	no								
MULT1	yes	MULTD	F0	F2	F4			yes	yes
MULT2	no								
ADD	yes	ADDD	F6	F8	F2			yes	yes
DIV	yes	DIVD	F10	F0	F6	MULT1		no	yes

Register Result Status

F0	F2	F4	F6	F8	F10	F12
MULT1			ADD		DIV	

CDC-6600 timing diagram

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
LD F6, 34(R2)	IF	ID	IS	EX	EX	WR																	
LD F2, 45(R3)		IF	ID	IS	EX	EX	WR																
MULTD F0, F2, F4			IF	ID	IS	IS	IS	IS	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	WR				
SUBD F8, F6, F2				IF	ID	IS	IS	IS	EX	EX	WR												
DIVD F10, F0, F6					IF	ID	IS	IS	IS	IS	IS	IS	IS	IS	IS	IS	IS	IS	IS	IS	EX	...	...
ADDD F6, F8, F2						IF	ID	ID	ID	ID	IS	IS	EX	EX	EX	EX	EX	EX	EX	EX	WR		

Execution latencies: LD (2 – addr. gen. + access), MULTD (10), DIVD (40), SUBD/ADDD(2)

Notice there are always 2 cycles between EX of data dependent instructions (e.g., LD and MULTD): producer does WR and consumer does last IS cycle in which registers are read from the register file
This is an artifact of the CDC-6600: all values must first pass through the register file (no bypasses).

Shaded boxes indicate stalls

RAW — LD-MULTD (F2), LD-SUBD (F2), MULTD-DIVD (F0), SUBD-ADDD (F8)

Structural — SUBD-ADDD (ADD unit)

WAR — DIVD-ADDD (F6)

CDC-6600 - Limitation

- **Performance in CDC-6600**

- CDC-6600 does a good job of dynamic scheduling around RAW hazards

- **Performance limitation**

- Amount of instruction-level parallelism (ILP) in the program
  - Maybe not enough data-independent operations
  - Increase size of window to look farther ahead (also requires branch prediction)
- Number of scoreboard entries (instruction window size)
  - Dictates how far processor can look ahead
- Number and type of functional units, register ports, etc.
  - Structural hazards
- Anti and output dependences
  - Dynamic scheduling exposes more WAW+WAR hazards because early (OOO) writes are possible
  - WAR made worse in CDC due to late reads (read operands when finally issuing)
  - WAW handled like a structural hazard in dispatch

Tomasulo's Algorithm (1)

IBM 360/91

- **Born of necessity**
- Used in IBM 360/91 (1966) floating-point unit
- Many long-latency operations
  - Need dynamic scheduling: Mitigate long stalls
- ISA specified only 4 floating-point registers
  - Need register renaming: With only 4 registers, WAW/WAR hazards pop up quickly
  - Especially in floating-point code: Loops by definition cause repeated writes to same registers
  - Renaming: Recognize and give unique names to different dynamic instances of the same register specifier

Tomasulo's Algorithm (2)

- **Key aspects**

- Read register operands at dispatch stage
  - If operands are available, the data is buffered along with the instruction in reservation station
  - CDC: Only buffers the instruction, all operands are read from register file when all operands are ready (operands read at issue stage)
  - CDC – late reads/Tomasulo – early reads: Early reads help WAR conditions
- Unavailable registers are renamed at dispatch stage
  - Waiting instructions replace register specifiers with a "tag" indicating the producer instruction
  - Register specifiers are used only once at dispatch
  - Renaming eliminates WAR/WAW hazards
- Successive writes to a register
  - Only last one is actually used to update register: Helps WAW condition
  - CDC: stall in dispatch until WAW hazard goes away

Tomasulo's Algorithm (3)

- **Other differences with CDC**

- Distributed control
  - Reservation stations (versus scoreboard)
- Results broadcast to both register file and functional units
  - Result bus called the common data bus (CDB)
  - Don't have to wait for value to go through register file (i.e., use forwarding)
  - Functional units don't contend for register file ports
- Renaming example
  - Reservation station number (tag) of producer instruction (e.g. load buffer 1 = load1)
  - This guarantees unique names for unique values

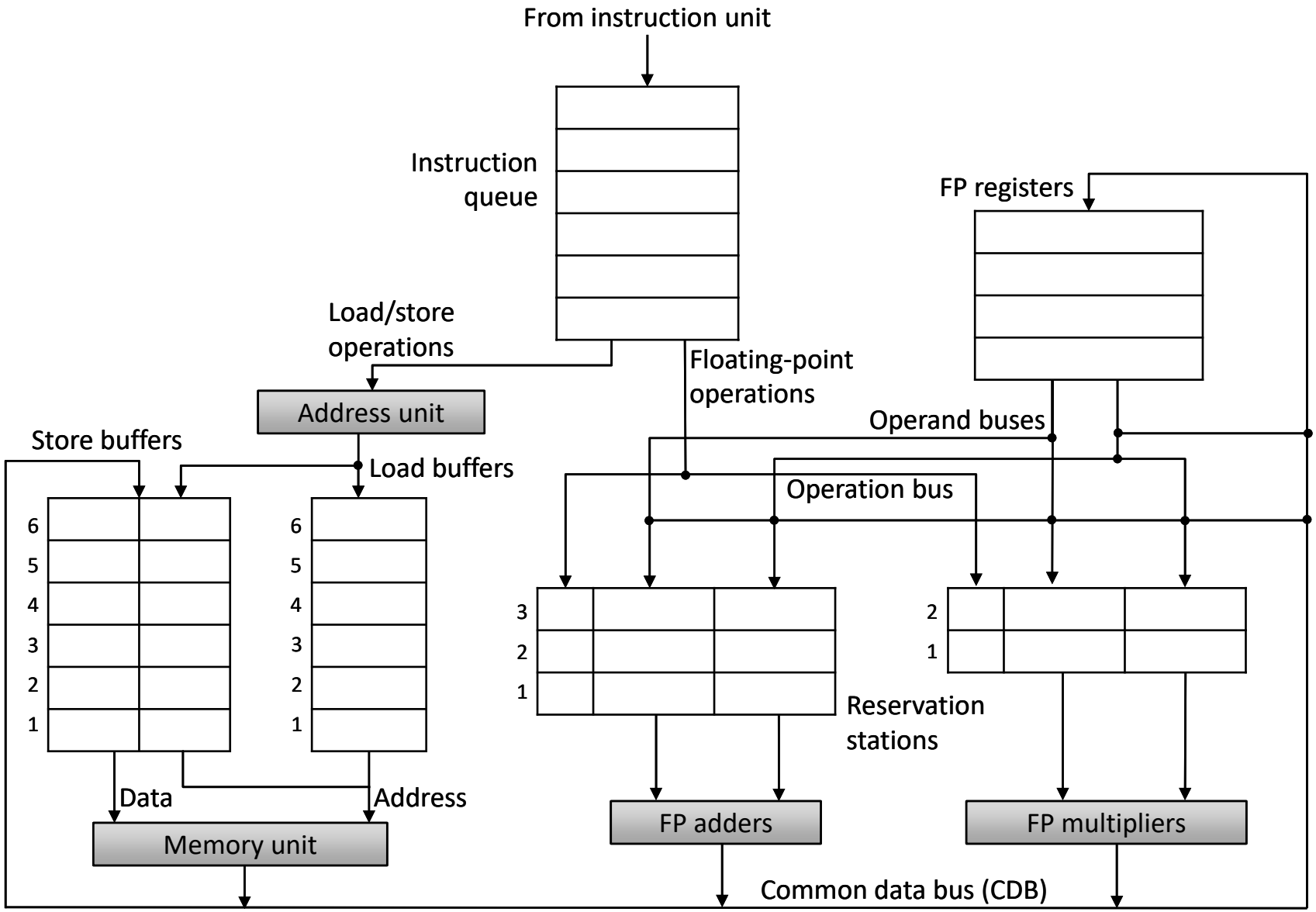
Example codes

```
LD    F0, 34 (R2)
ADDD  F4, F0, F2
LD    F0, 45 (R3)
ADDD  F8, F0, F6
```

After renaming

```
LD    load1, 34 (R2)
ADDD  F4, load1, F2
LD    load2, 45 (R3)
ADDD  F8, load2, F6
```

360/91 FP unit (DLX Version)



Tomasulo's Algorithm (4)

- **Fetch**
- **Dispatch**

- Check for structural hazard
  - Stall in dispatch stage if no free reservation station
- Read register file
  - Read data operands if available
  - Read "tag" if data operand unavailable: A tag is the reservation station number of the producer instruction
- Route instruction plus data or tags to reservation station, where it waits until all operands are available

- **Issue**

- Wait for operands to become ready on the CDB
  - Match CDB tag against operand tags
- Grab operands from CDB and issue to FU (if free)

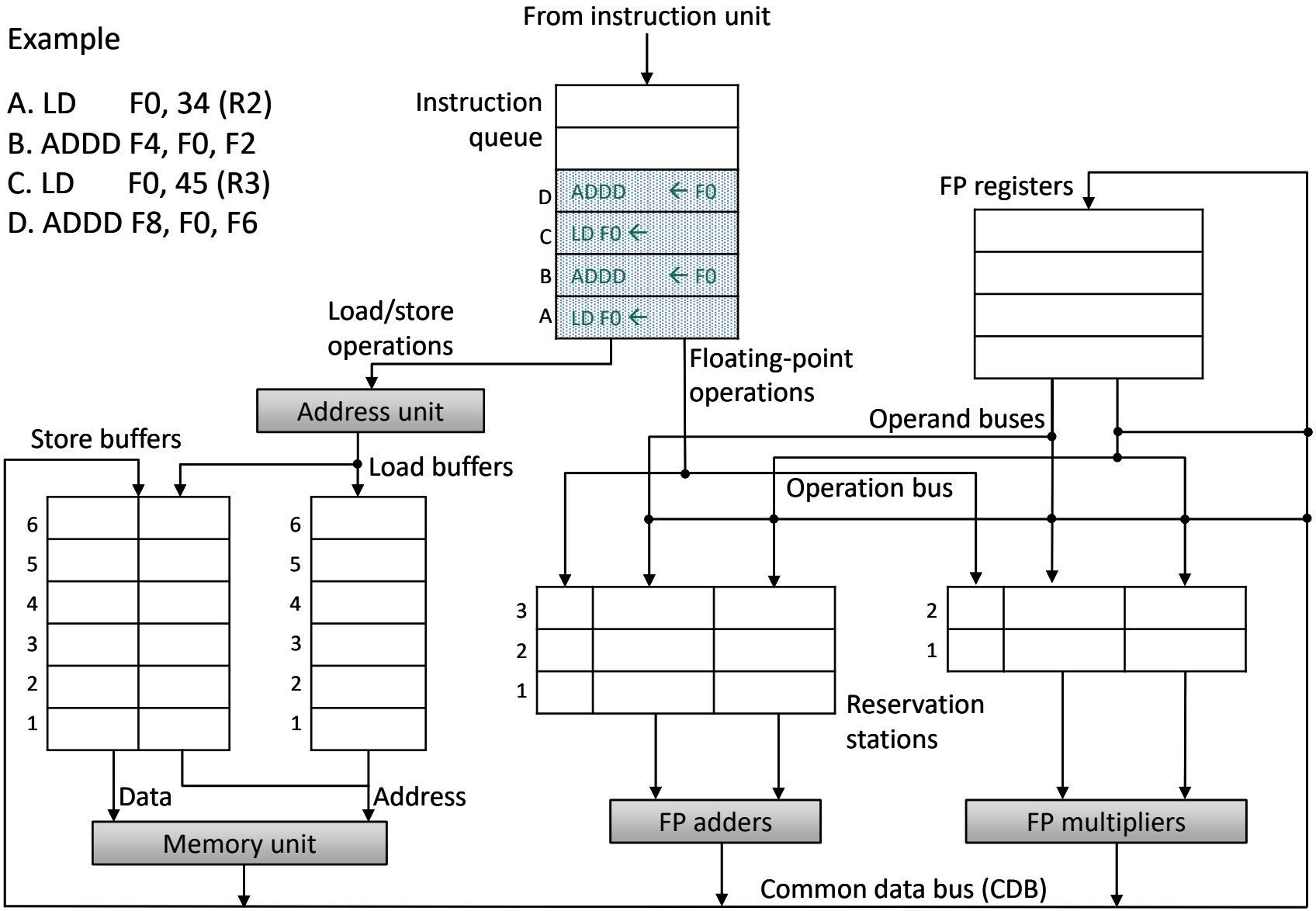
- **Execute**
- **Write result**

- Broadcast result + tag to all reservation stations, store buffers, and register file via the CDB
- Only write the register file if the CDB tag matches the tag in the register file

How Tomasulo renaming works (1)

Example

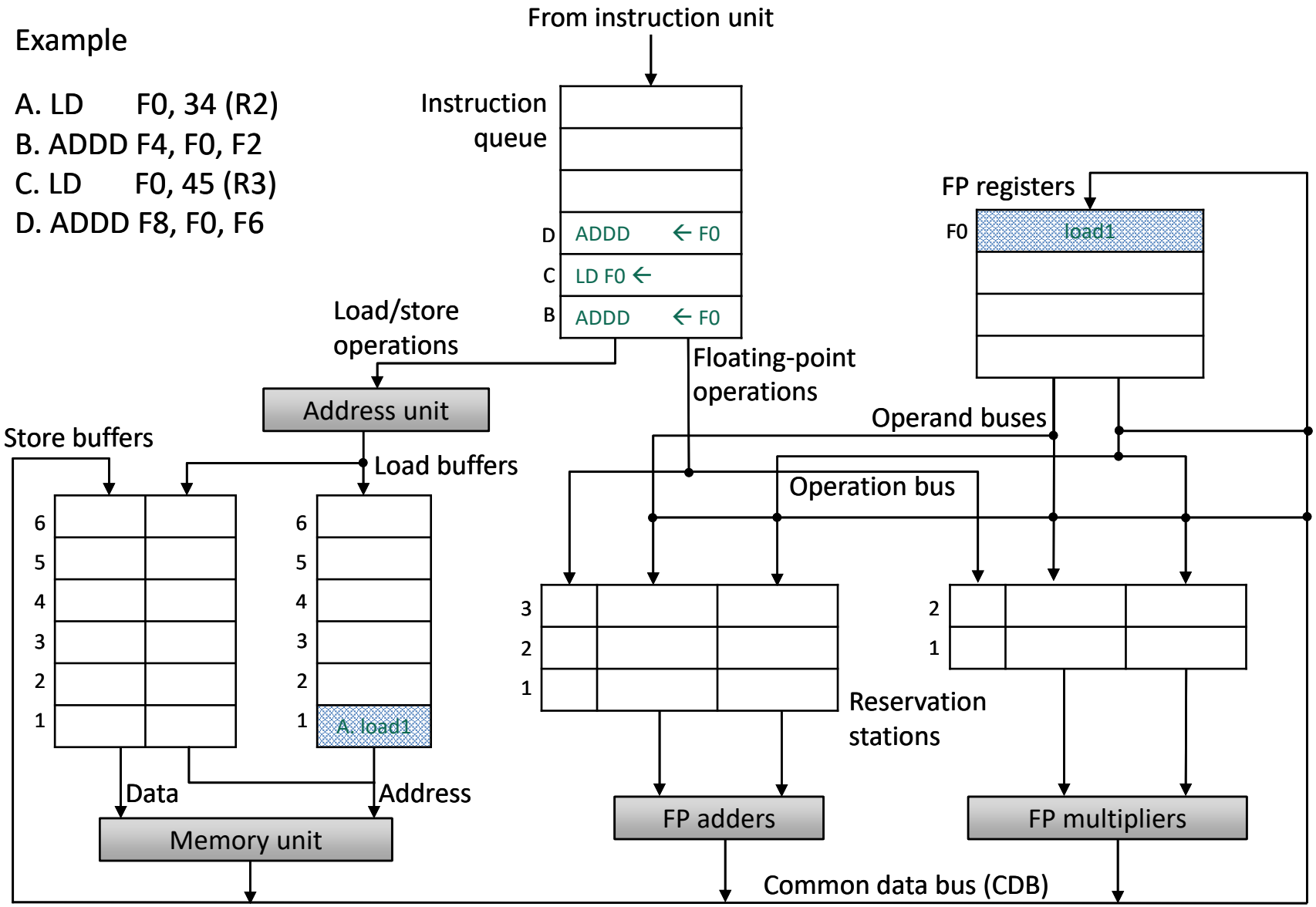
A. LD F0, 34 (R2)
B. ADDD F4, F0, F2
C. LD F0, 45 (R3)
D. ADDD F8, F0, F6



How Tomasulo renaming works (2)

Example

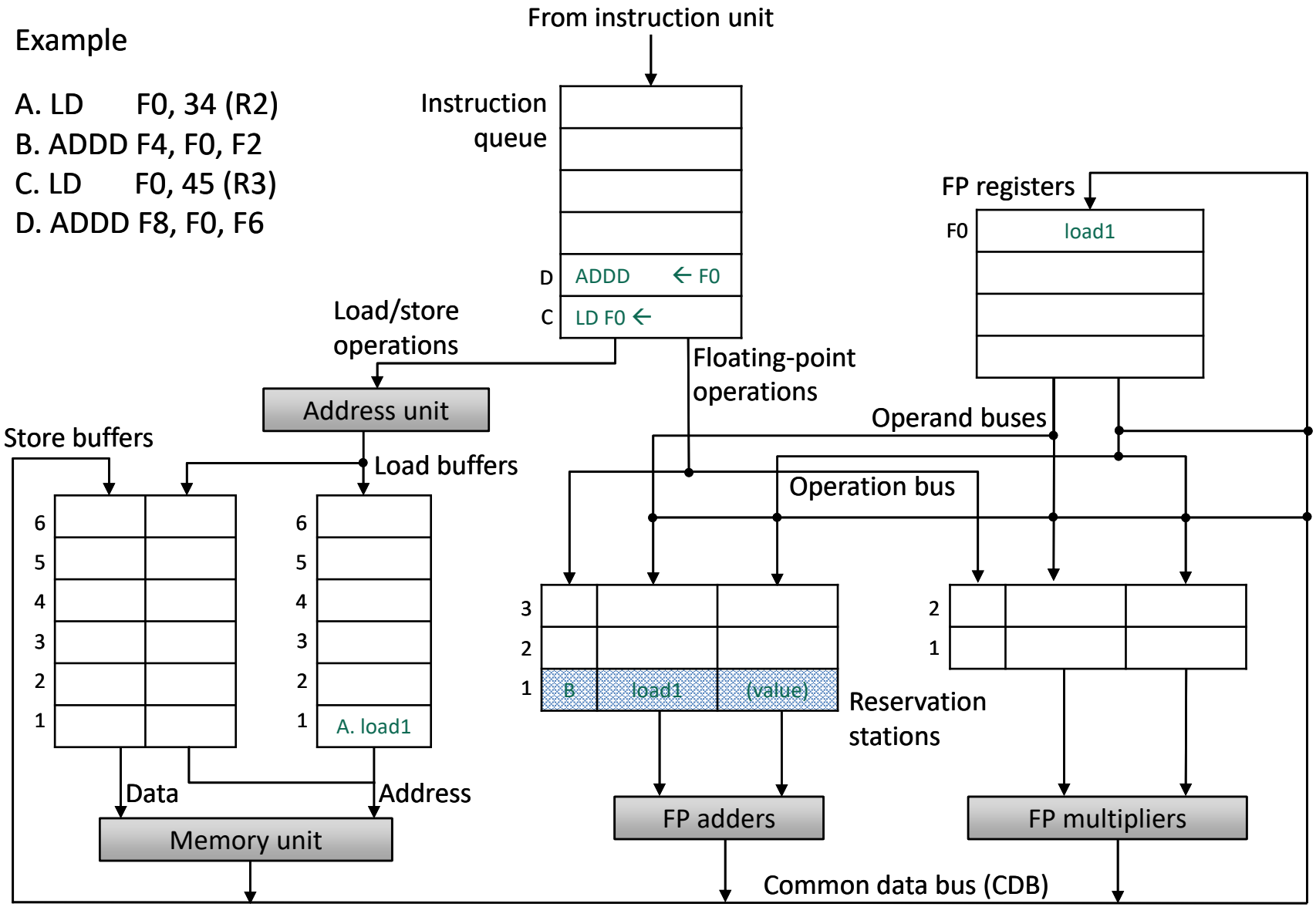
A. LD F0, 34 (R2)
B. ADDD F4, F0, F2
C. LD F0, 45 (R3)
D. ADDD F8, F0, F6



How Tomasulo renaming works (3)

Example

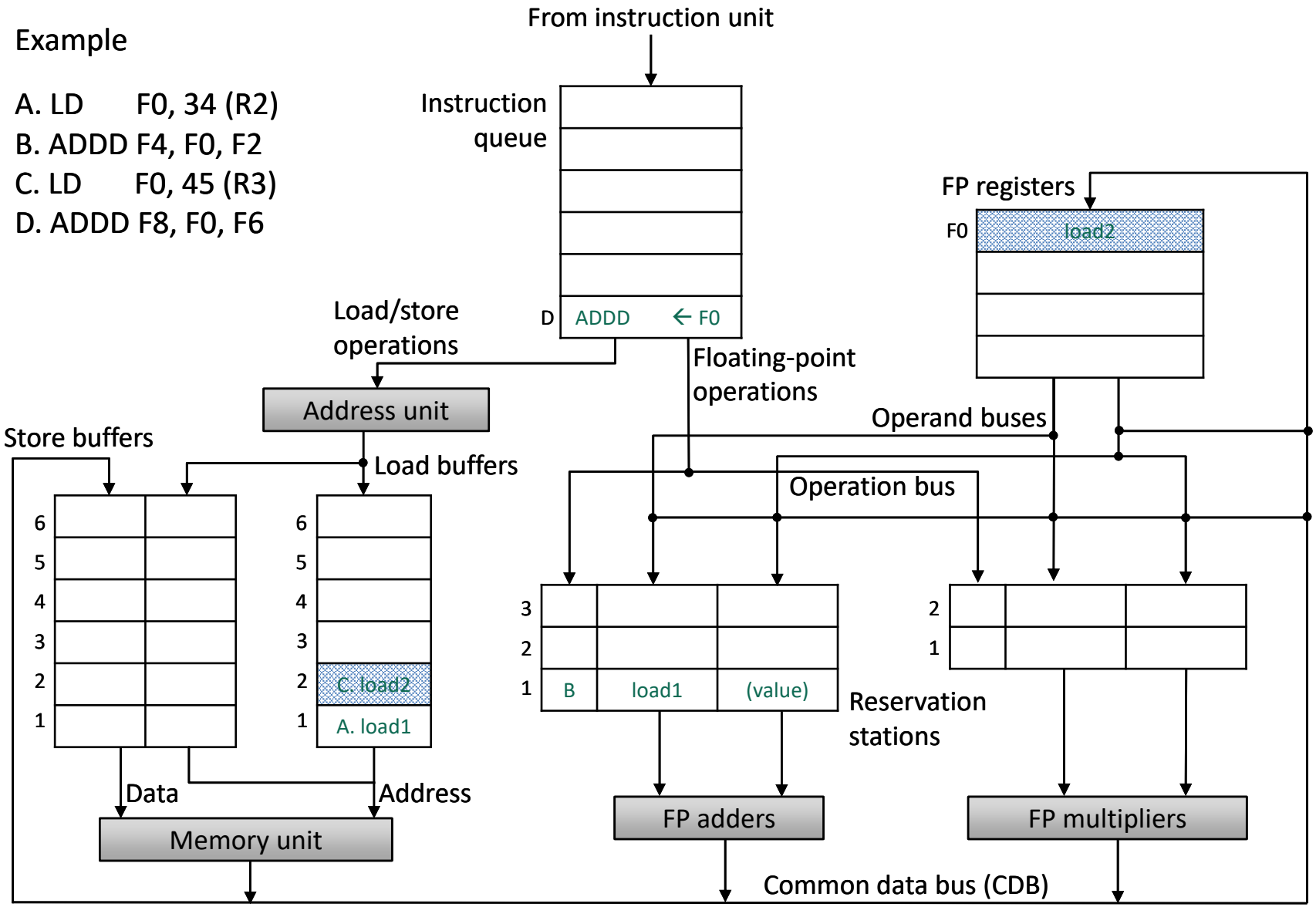
A. LD F0, 34 (R2)
B. ADDD F4, F0, F2
C. LD F0, 45 (R3)
D. ADDD F8, F0, F6



How Tomasulo renaming works (4)

Example

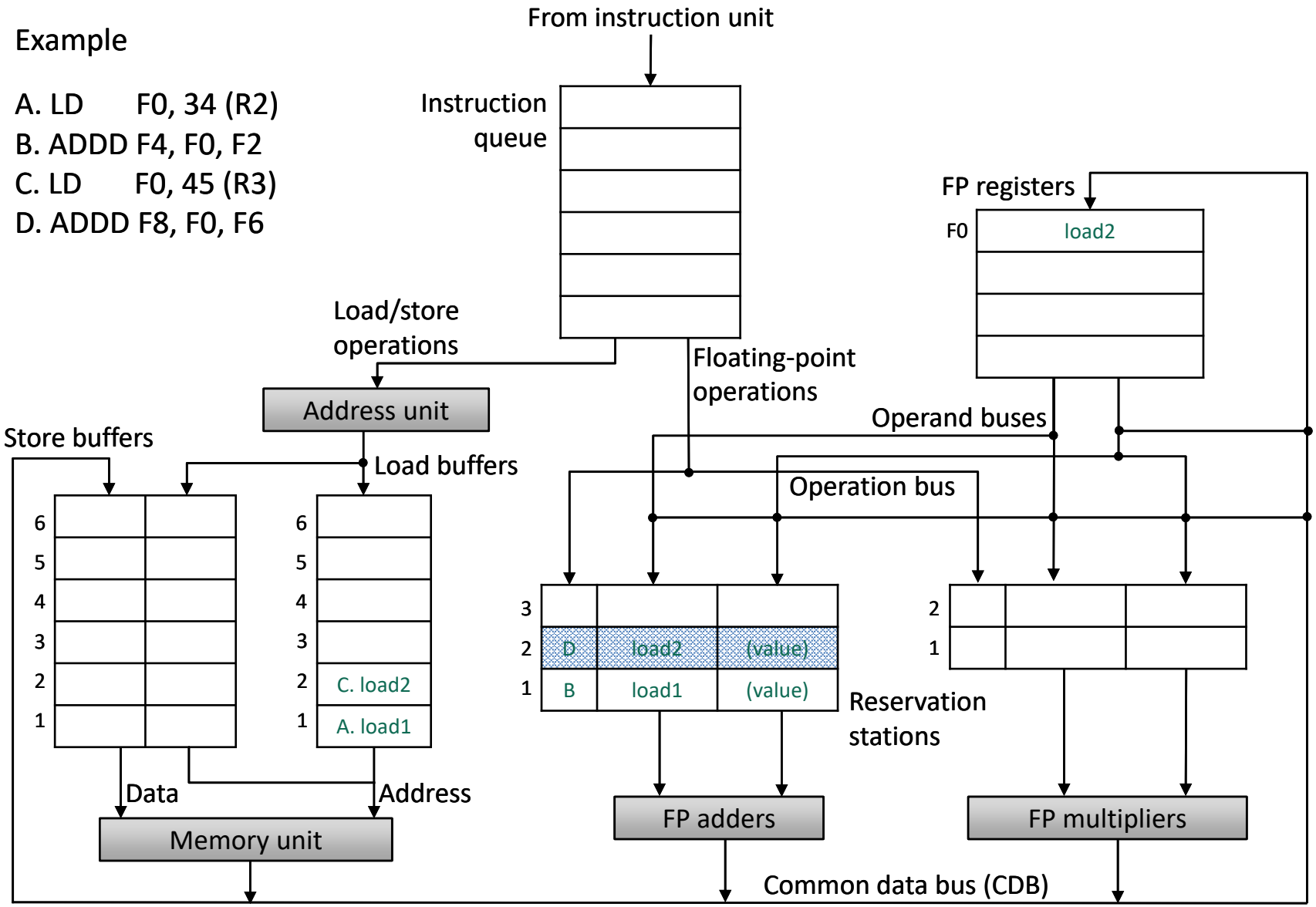
A. LD F0, 34 (R2)
B. ADDD F4, F0, F2
C. LD F0, 45 (R3)
D. ADDD F8, F0, F6



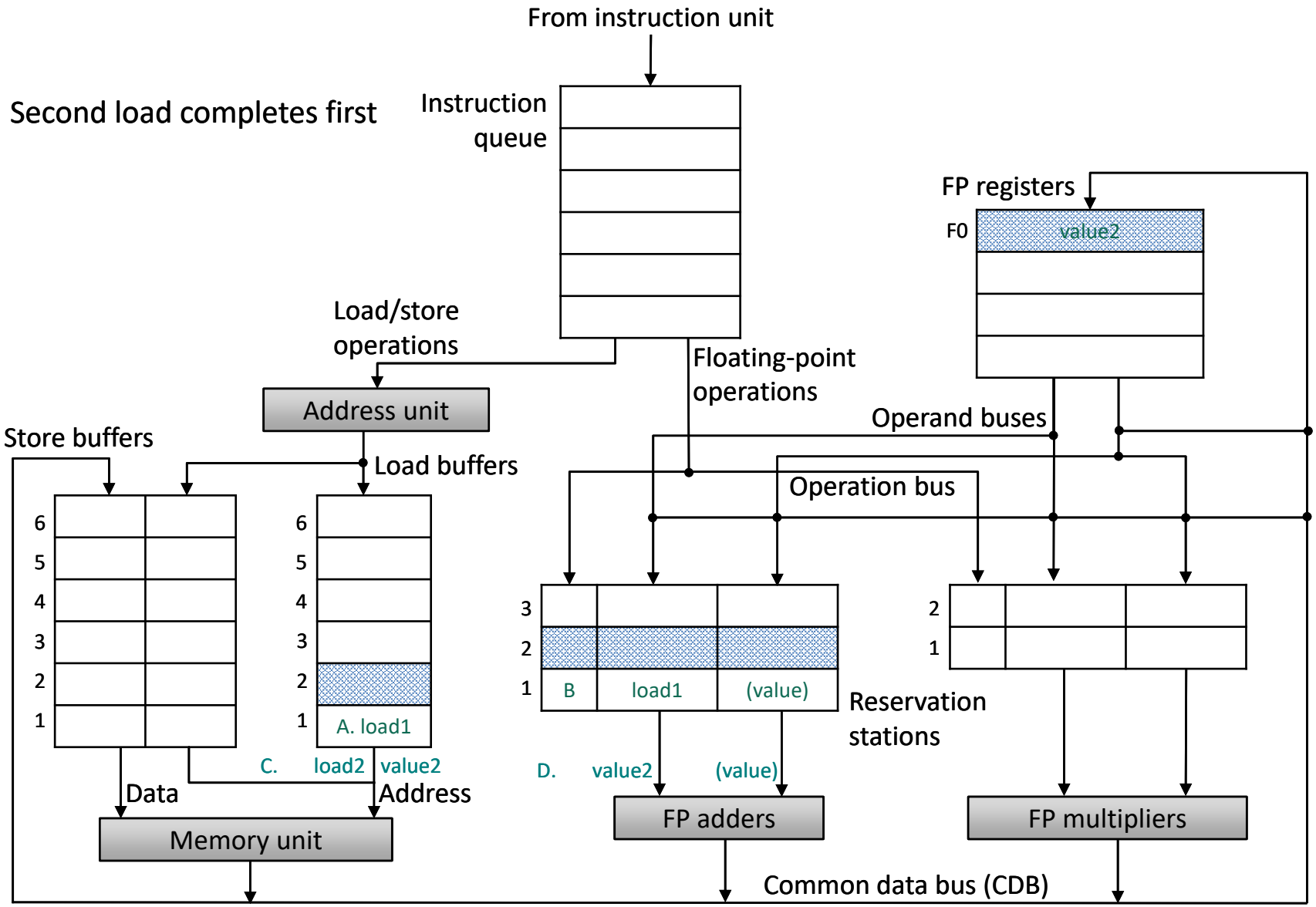
How Tomasulo renaming works (5)

Example

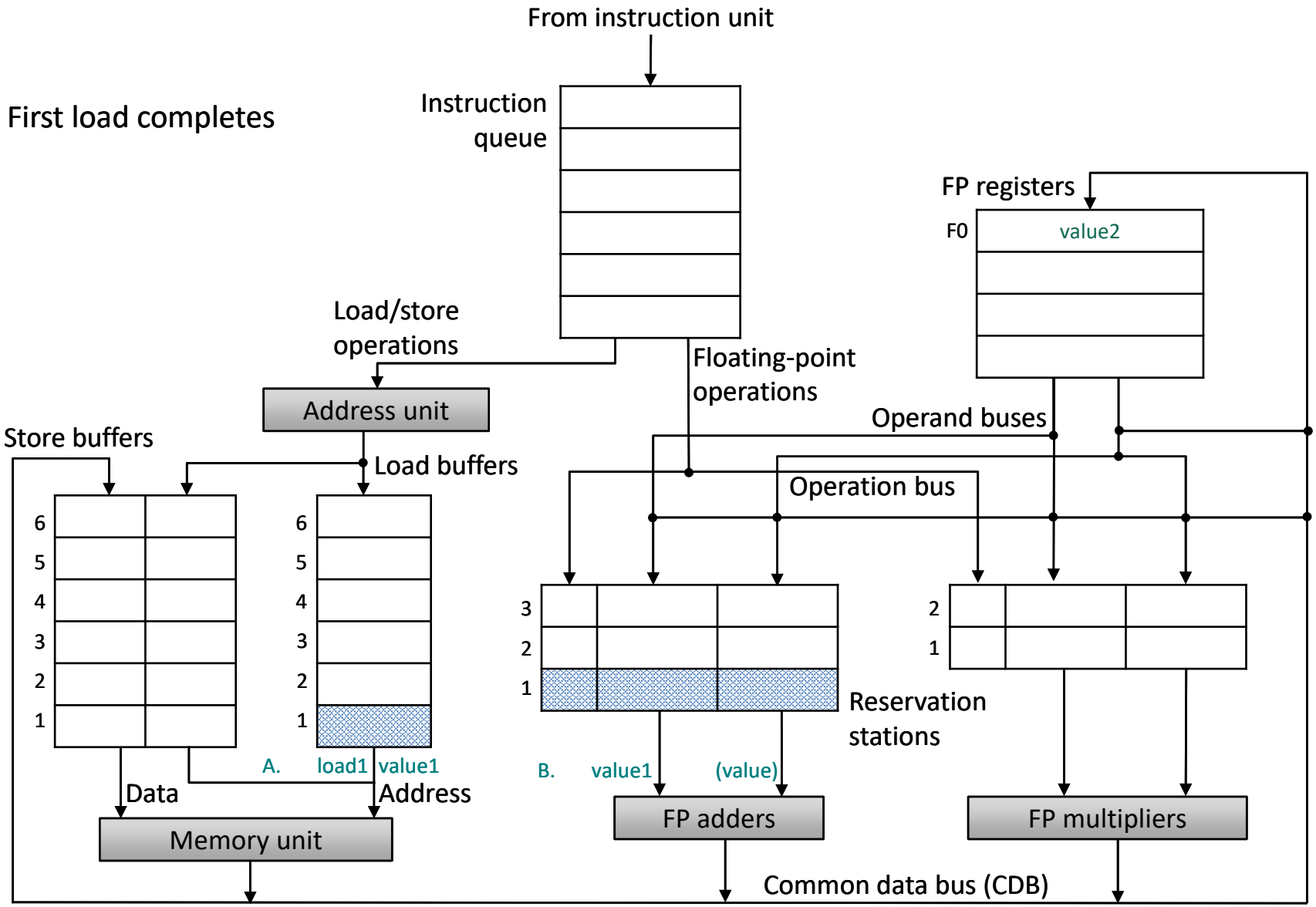
A. LD F0, 34 (R2)
B. ADDD F4, F0, F2
C. LD F0, 45 (R3)
D. ADDD F8, F0, F6



How Tomasulo renaming works (6)



How Tomasulo renaming works (7)



Tomasulo's Algorithm Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
LD F6, 34(R2)	IF	ID	IS	EX	EX	WB																	
LD F2, 45(R3)		IF	ID	IS	EX	EX	WB																
MULTD F0, F2, F4			IF	ID	IS	IS	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	WB						
SUBD F8, F6, F2				IF	ID	IS	EX	EX	WB														
DIVD F10, F0, F6					IF	ID	IS	IS	IS	IS	IS	IS	IS	IS	IS	IS	EX	EX	...				
ADDD F6, F8, F2						IF	ID	IS	EX	EX	WB												

Execution latencies: LD (2 – addr. gen. + access), MULTD (10), DIVD (40), SUBD/ADDD(2)

CDB F6 F2 F8 F6' F0

Shaded boxes indicate stalls

RAW LD-MULTD (F2), MULTD-DIVD (F0)

Execution latencies: LD (2 – agen + access), MULTD (10), DIVD (40), SUBD/ADDD (2)

Precise interrupts

- **Out-of-order execution and interrupts**
  - When an interrupt occurs, O/S wants a precise state
  - Precise state means
    - All instructions before the instruction causing the interrupt have completed
    - All instructions after have not completed
  - In out-of-order execution, instructions after the interrupt may have updated the state
- Why are precise interrupts important?
  - O/S needs to save the state of the process and restart the process after servicing interrupt
    - Restart occurs from the PC of interrupted instruction
    - The restart state must reflect only changes up to that PC
  - Programmer debugging
    - User presumes sequential program

Interrupts

- **Some examples**

- External interrupts
  - I/O device request
  - Timer interrupt
  - Power failing
- Exceptions
  - Arithmetic overflow, divide-by-0, etc.
  - Page fault
- O/S calls
  - Also called system call or trap
  - Initiated via an explicit instruction in ISA
- External interrupts are asynchronous, exceptions are synchronous, O/S calls are synchronous and user-initiated

Classifying Interrupts

- **Synchronous/Asynchronous**

- Synchronous/Asynchronous
  - Synchronous: function of program and memory state
    (e.g., arithmetic overflow, page fault, divide by 0)
  - Asynchronous: external device or hardware malfunction (e.g., printer ready, bus error)
- User request/Coerced
  - From user program (O/S call – system call or trap)
  - From O/S or hardware (e.g., page fault, protection violation)
- User maskable/Non-maskable
  - User maskable: Can be (temporarily) ignored (e.g., arithmetic overflow, breakpoint)
  - Non-maskable: must be handled (e.g., page fault, power fail)
- Within an instruction / Between instructions
  - Within: Must be dealt with to complete an instruction (e.g., page fault)
  - Between: Not part of an instruction (e.g., I/O device request, O/S call)
- Resume/Terminate
  - Resume: Must transparently return to user process (e.g., page fault, I/O device request)
  - Terminate: Give up and die (e.g., protection violation, power failure)

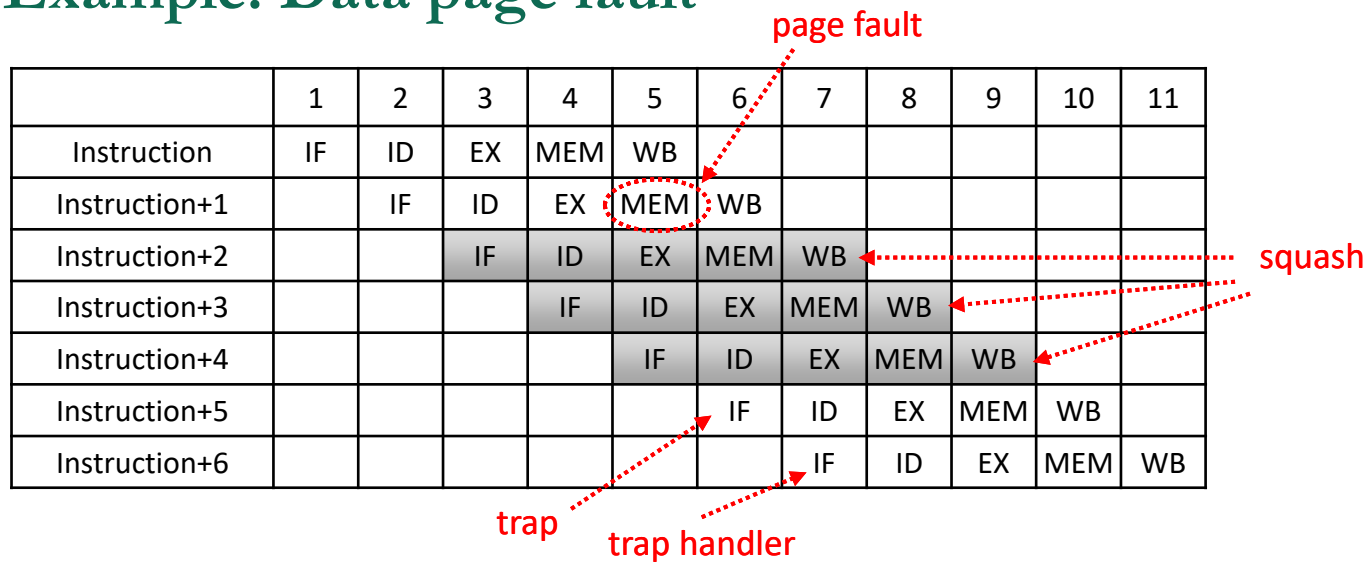
Handling Interrupts (1)

- **Precise interrupts (Sequential Semantics)**
- Complete instructions before the faulting instruction
- Squash instructions after the faulting instruction
- Save PC of the faulting instruction
- Force a trap instruction into the pipeline on the next IF

- **Must handle simultaneous/OOO interrupts**
- IF: Memory problems (page fault, mis-aligned reference, protection violation)
- ID: Illegal or privileged instruction
- EX: Arithmetic exception
- MEM: Memory problems (page fault, mis-aligned reference, protection violation)
- WB
- Which interrupt should be handled first?

Handling Interrupts (2)

- **Example: Data page fault**



	1	2	3	4	5	6	7	8	9	10	11
Instruction	IF	ID	EX	MEM	WB						
Instruction+1		IF	ID	EX	MEM	WB					
Instruction+2			IF	ID	EX	MEM	WB				
Instruction+3				IF	ID	EX	MEM	WB			
Instruction+4					IF	ID	EX	MEM	WB		
Instruction+5						IF	ID	EX	MEM	WB	
Instruction+6							IF	ID	EX	MEM	WB

- **Events**

- Preceding instruction already complete
- Squash succeeding instructions
  - Prevent from modifying state
- Inject 'trap' instruction, jumps to trap handler
- Trap handler saves precise state

Handling Interrupts (3)

- **Example: Out-of-order interrupts**

page fault

	1	2	3	4	5	6	7	8	9	10	11
Instruction	IF	ID	EX	MEM	WB						
Instruction+1		IF	ID	EX	MEM	WB					
Instruction+2			IF	ID	EX	MEM	WB				
Instruction+3				IF	ID	EX	MEM	WB			
Instruction+4					IF	ID	EX	MEM	WB		
Instruction+5						IF	ID	EX	MEM	WB	
Instruction+6							IF	ID	EX	MEM	WB

page fault

- Which page fault should we take?
- Even though 'instruction+2' fault first, must service 'instruction+1' fault first
- Solution: post interrupts
  - Check interrupt bit on entering WB
  - This maintains precise interrupts

Handling Interrupts (4)

- **Preceding example assumed in-order pipeline**
  - WB occurs in-order
  - This makes precise interrupts easy
- **Consider OOO-execution**

floating-point arithmetic exception

	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F2, F4	IF	ID	IS	EX	EX	EX	EX	EX	WB		
ADDD F6, F8, F8		IF	ID	IS	EX	WB					

  - Problem: ADD has already completed
    - State is imprecise

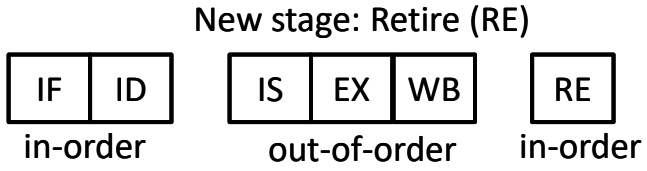
Solutions

- **Ignore the Problem**
  - Make difficult to handle restartable exceptions (e.g., page fault)
  - IEEE Standard strongly suggests precise interrupts
- **Hybrid (in-order + out-of-order) completion**
  - Stall pipeline when necessary
  - Only allow out-of-order execution when no exceptions are guaranteed
- **Imprecise Exception (Software cleanup)**
  - Save enough information for trap handlers (pipeline state)
    - Remember all in-completed instructions
    - After handling exception, the trap handle make a precise sequence for the exception by executing all in-complete instructions
  - Complex to execute the not-yet finished instructions
- **Re-order instructions (Re-order buffer)**
  - Complete out-of-order
  - Retire in-order

Reorder Buffer (1)

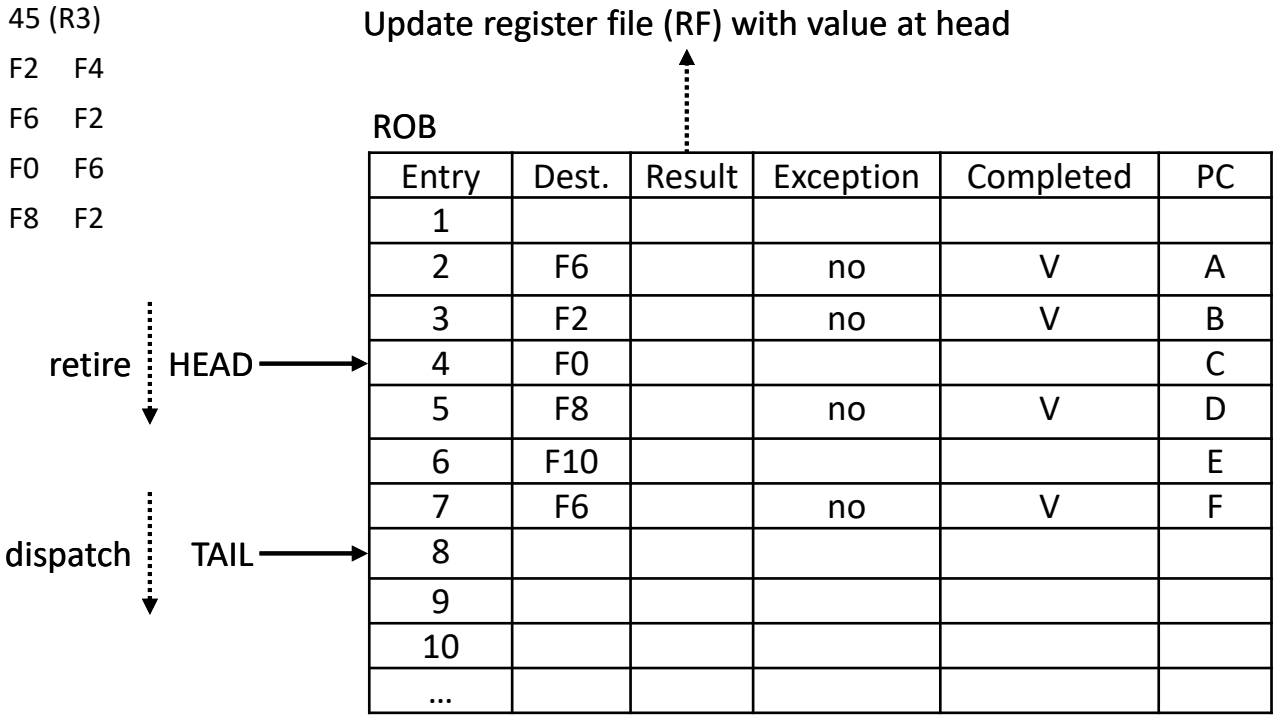
- **Operation of reorder buffer (ROB)**

- ROB is a FIFO with head and tail pointer
- Instruction dispatch
  - Reserve ROB entry at tail, advance tail pointer
  - Record ROB entry (ROB tag) with instruction
- Instruction execution/completion
  - Write value into ROB entry specified by the instruction's tag
  - DO NOT write result into the register file
  - If instruction caused an exception, mark exception bit in the ROB for the faulting instruction
- Instruction retire (a.k.a. commit, graduate)
  - If completed, check exception bit
  - If no exception, commit state (write value into register file) and advance head pointer
  - If exception, squash subsequent instructions in ROB (back-up the tail pointer to the head pointer)



Reorder Buffer (2)

A	LD	F6	34 (R2)	
B	LD	F2	45 (R3)	
C	MULTD	F0	F2	F4
D	SUBD	F8	F6	F2
E	DIVD	F10	F0	F6
F	ADDD	F6	F8	F2



Entry	Dest.	Result	Exception	Completed	PC
1					
2	F6		no	V	A
3	F2		no	V	B
4	F0				C
5	F8		no	V	D
6	F10				E
7	F6		no	V	F
8					
9					
10					
…					

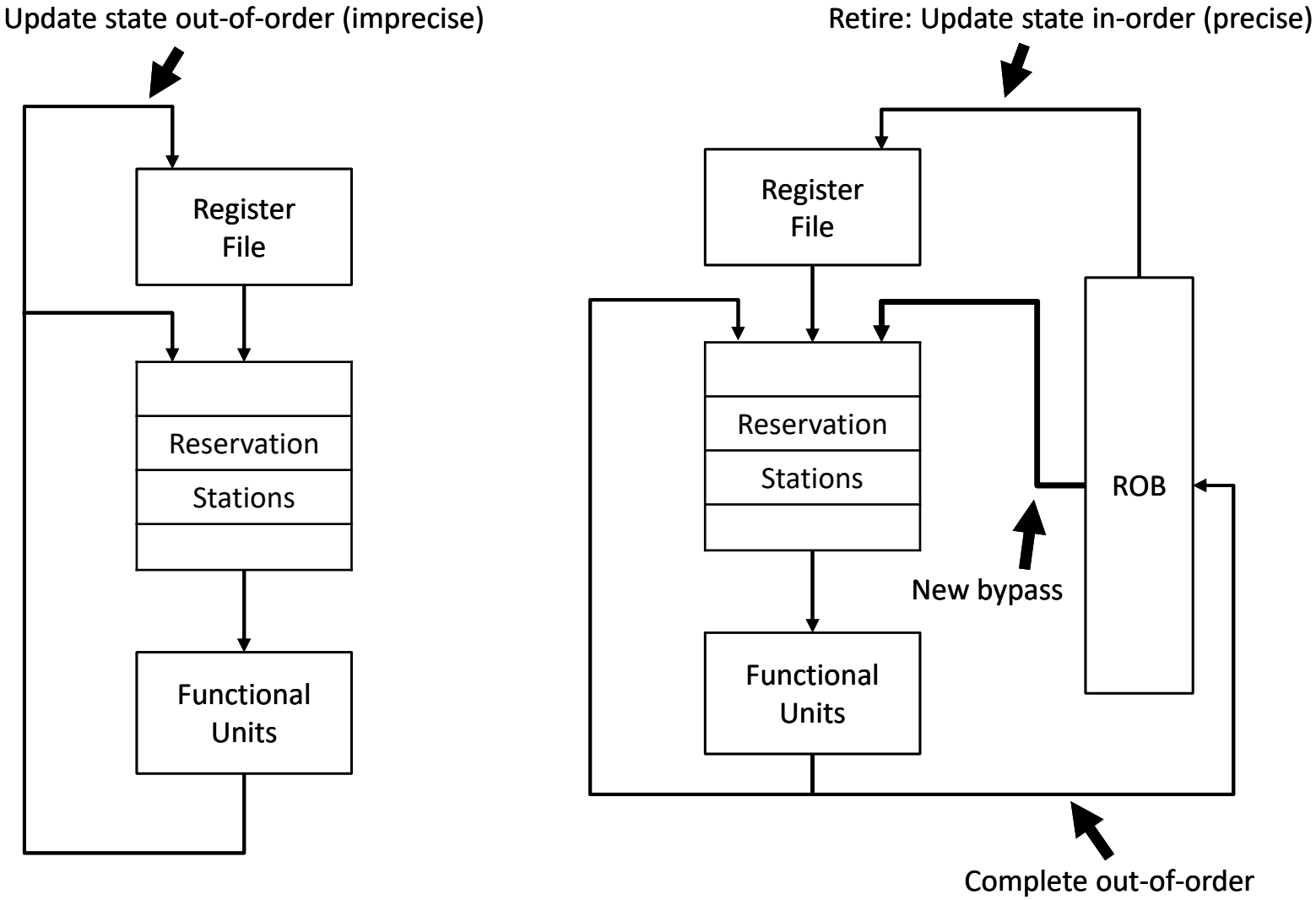
Shown: The two loads (A and B) have completed and retired.
The MULTD has not completed.
The ADDD and SUBD have completed out-of-order, but the ROB prevents them from retiring (updating state).
THUS: if the MULTD posts an exception, the Register File state is precise

Reorder Buffer (3) – Bypass/forward

- **Bypass/forward scenario**

- S1. Producer instruction completed
  - Value in ROB
- S2. Producer instruction not retired
  - Value not in register file
- S3. Then, dependent instruction is dispatched
  - Value not read from register file
  - Value can't be grabbed from result bus – producer already completed
  - Only place data resides is in ROB – need to read value from ROB (ROB bypass)

Precise Tomasulo Pipeline with ROB



Reorder Buffer Example (1)

Example Codes

```
A   LD     R2,  4 (R0)
B   MULT   R3,  R1,  R2
C   LD     R2,  8 (R0)
D   ADD    R1,  R1,  R2
E   SUB    R2,  R0,  R1
F   ADD    R0,  R1,  R2
```

RF

	In RF	Tag	value
R0	√	-	54
R1	√	-	76
R2	√	-	-5
R3	√	-	-99

ROB

Entry	Dest.	Result	Exception	Completed	PC
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					

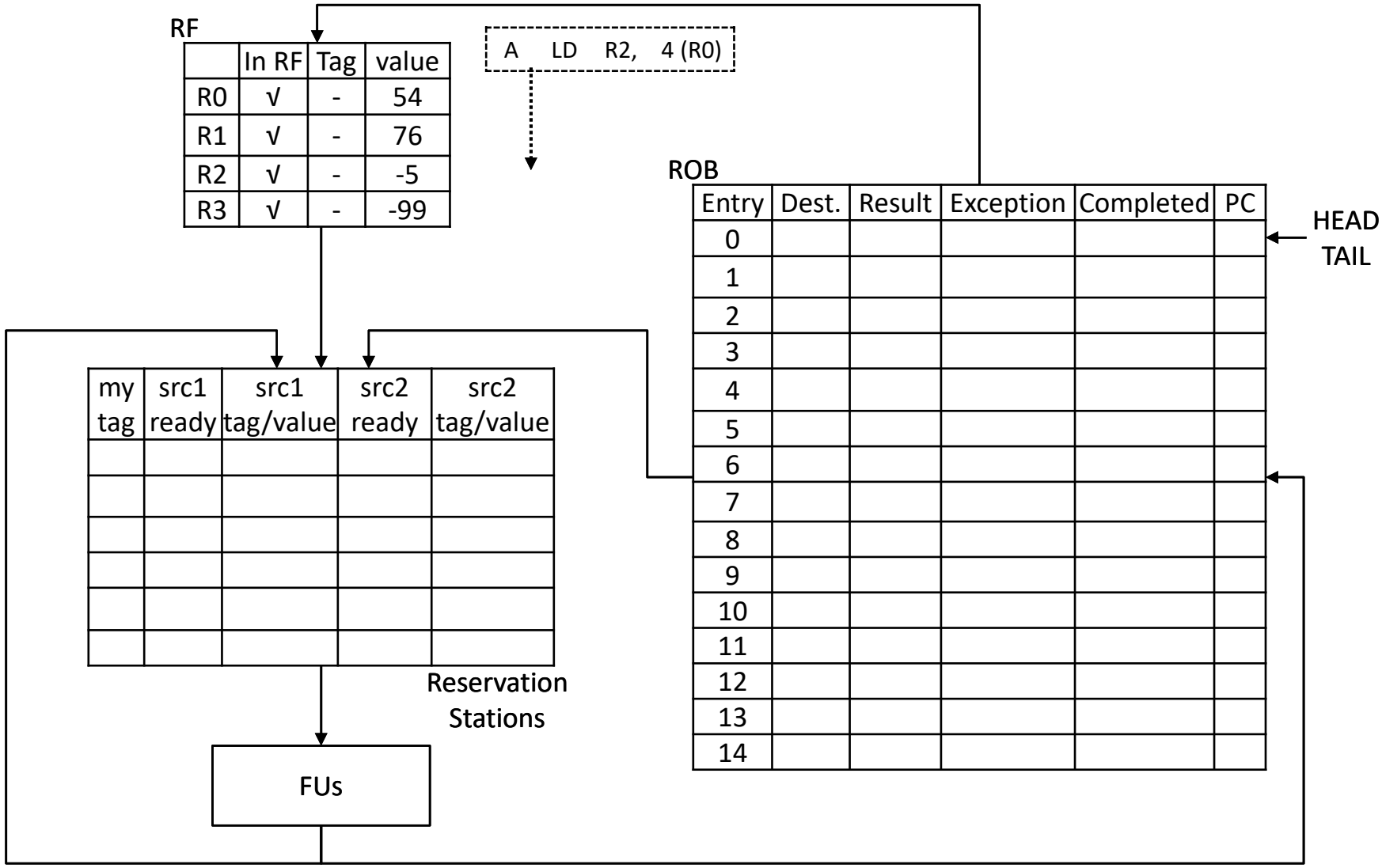
HEAD
TAIL

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value

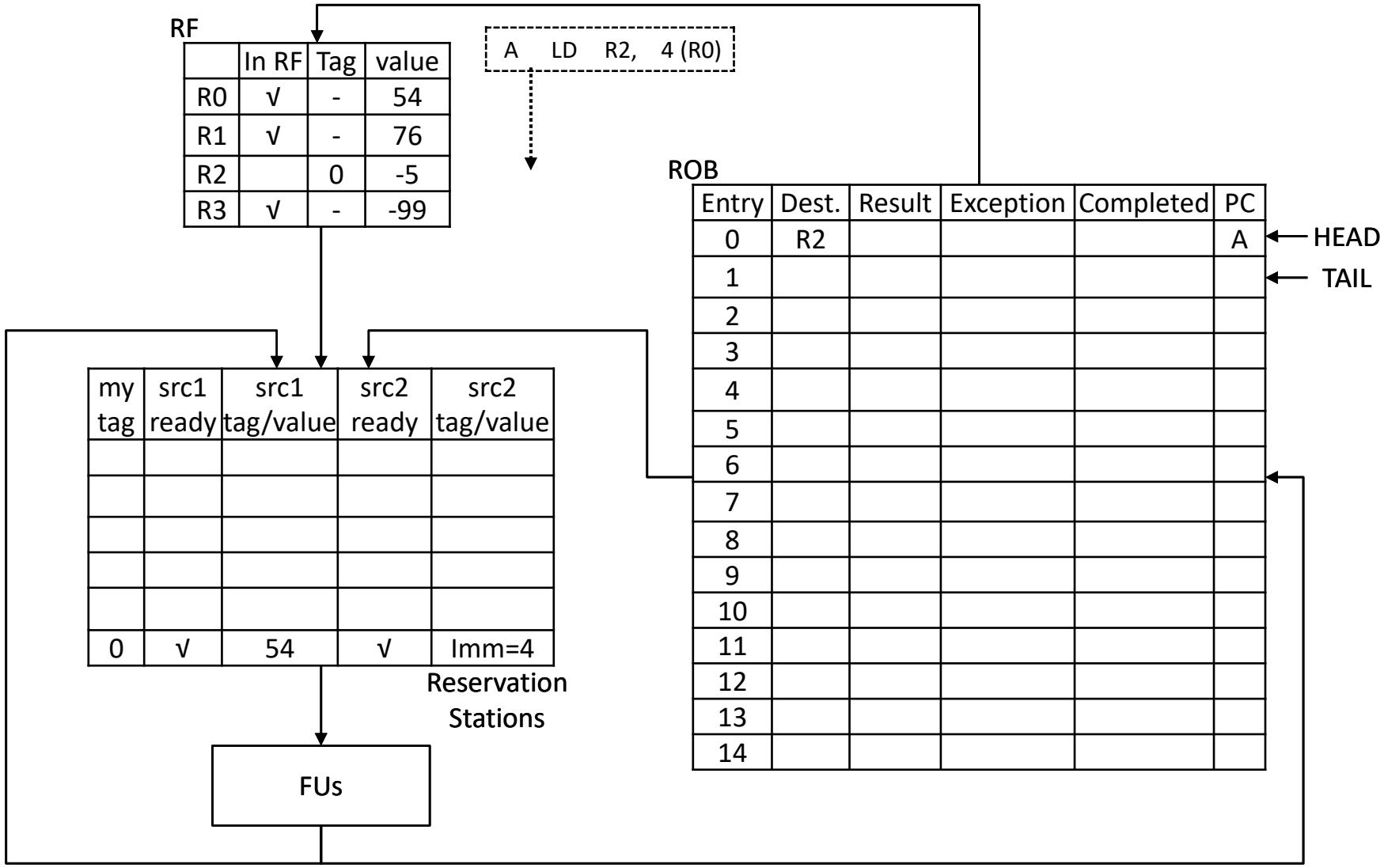
Reservation Stations

FUs

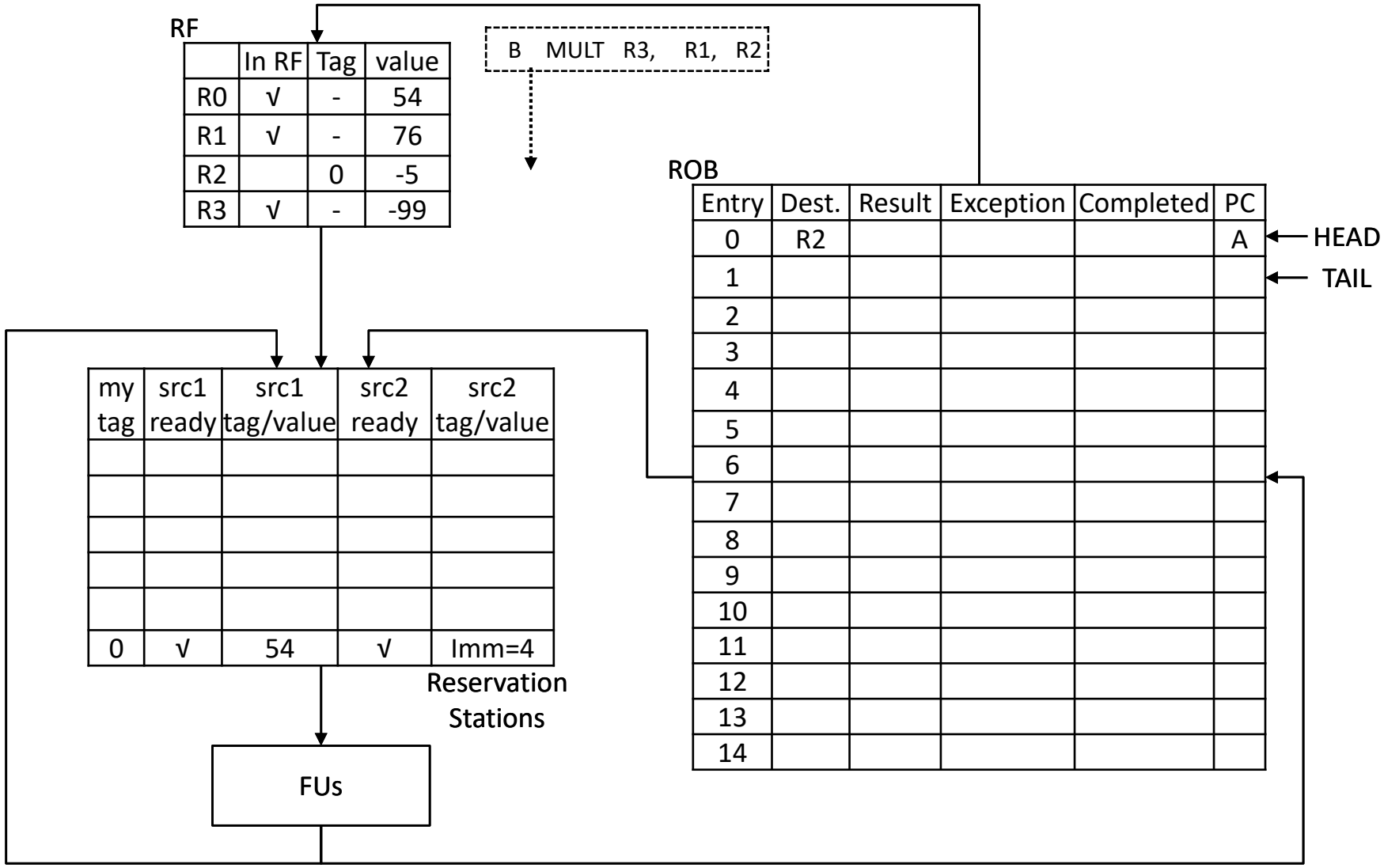
Reorder Buffer Example (2)



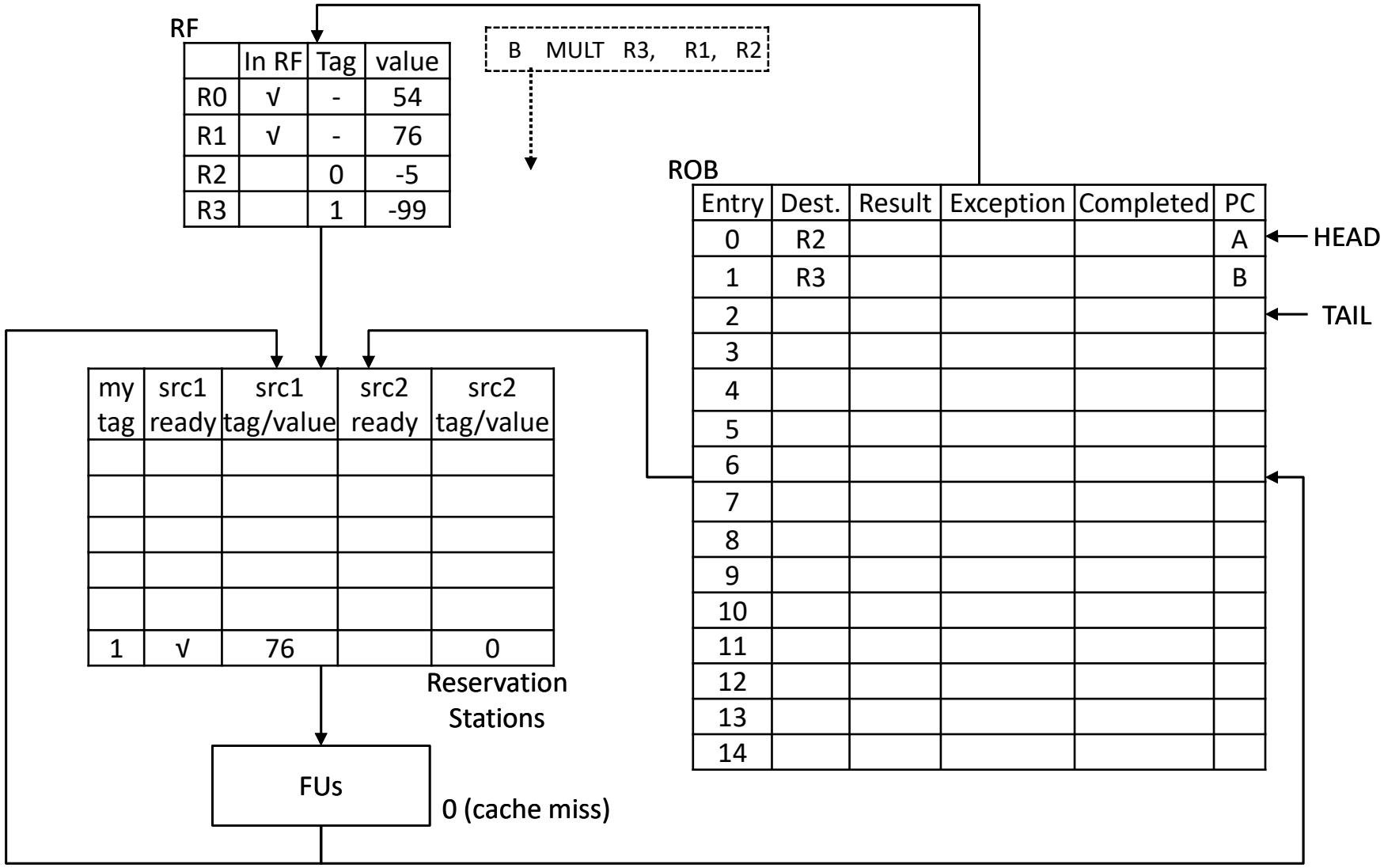
Reorder Buffer Example (3)



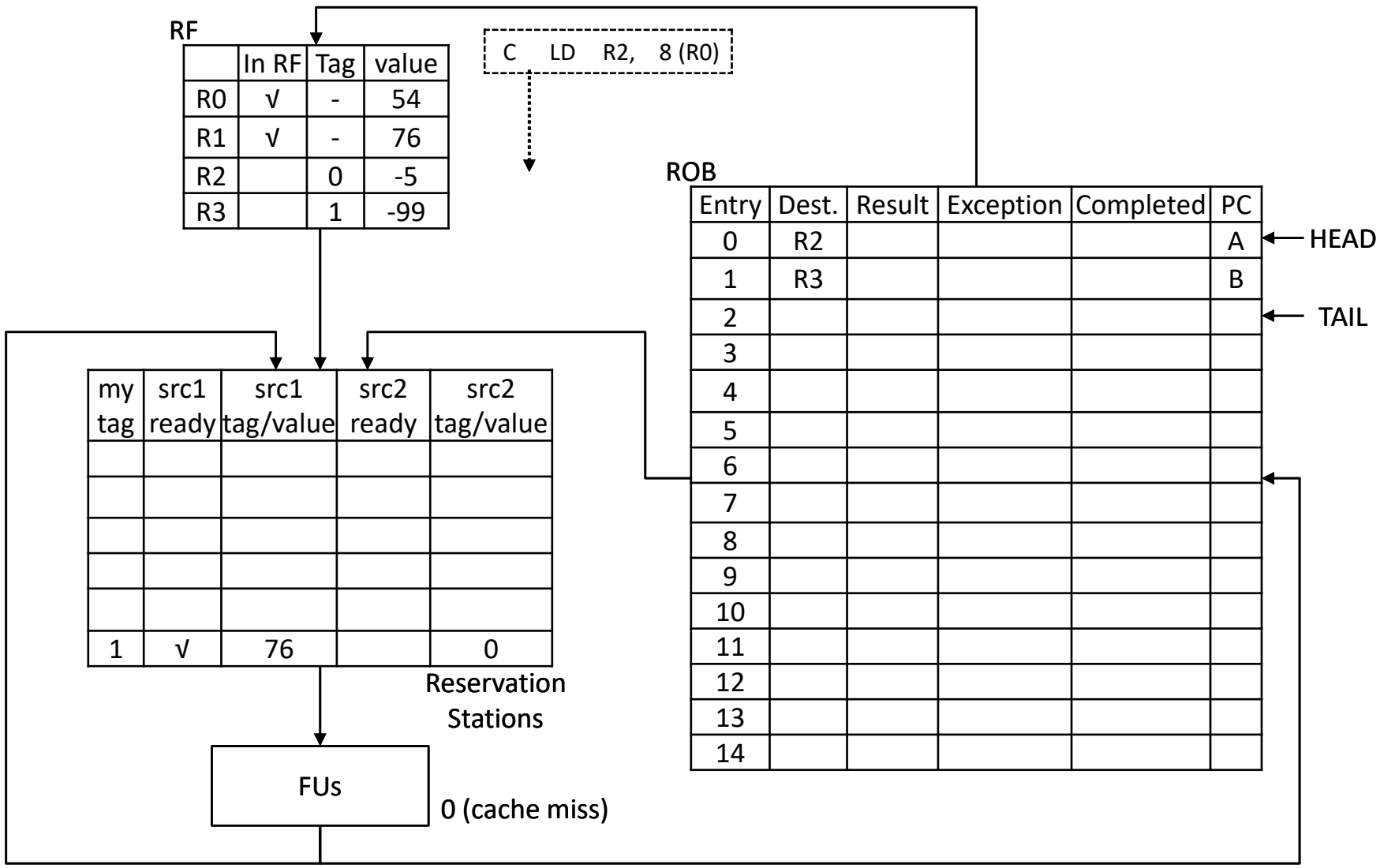
Reorder Buffer Example (4)



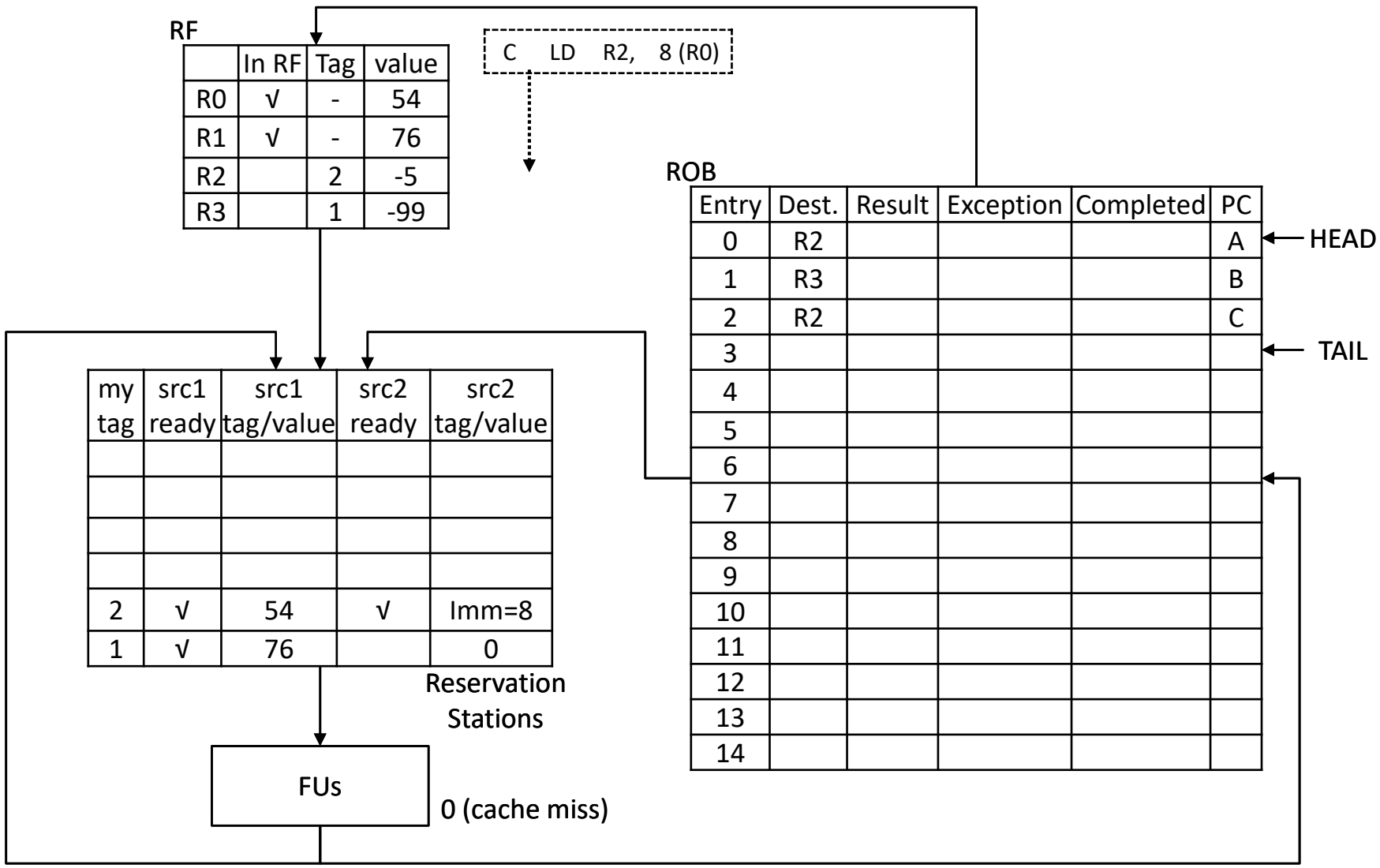
Reorder Buffer Example (5)



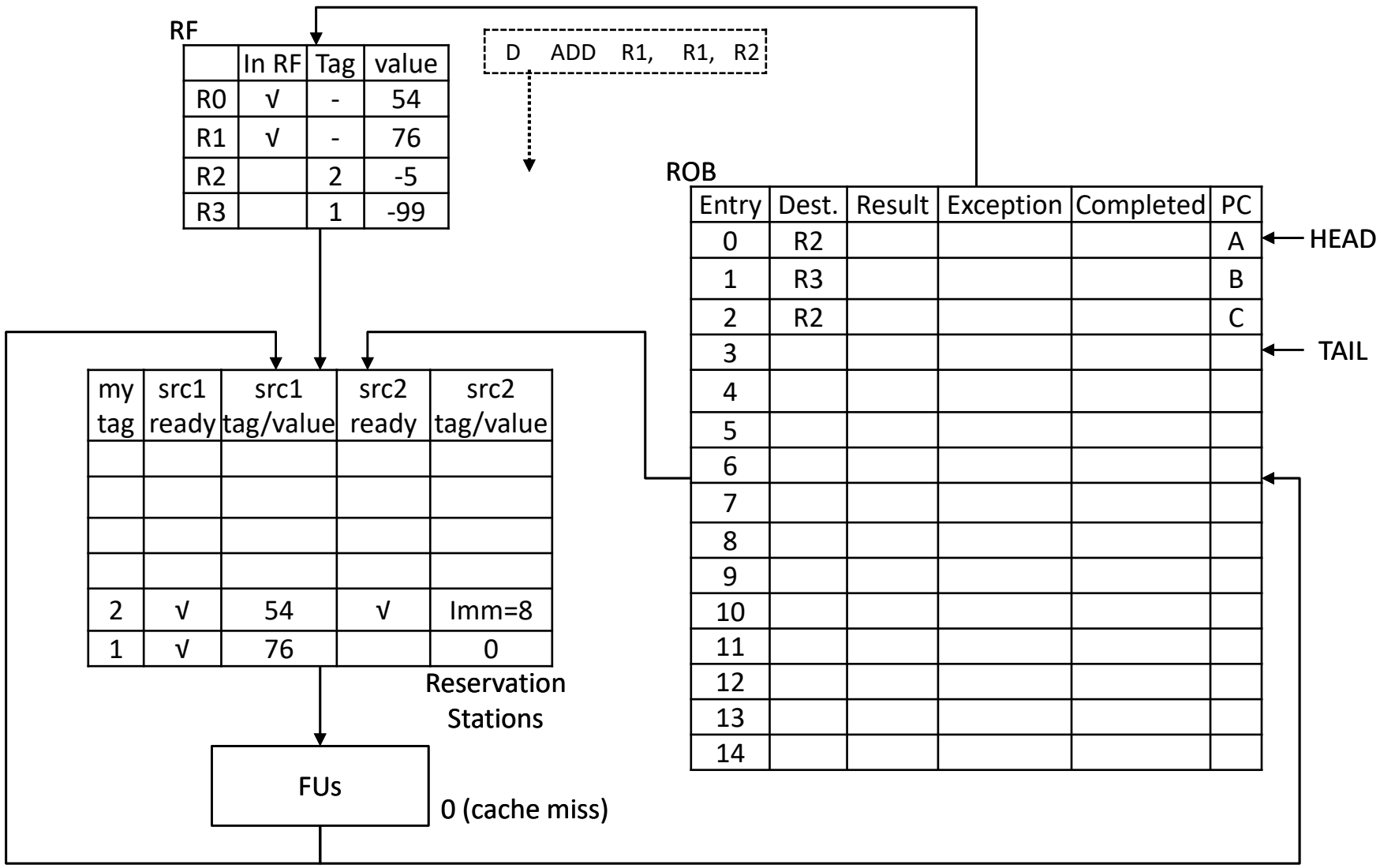
Reorder Buffer Example (6)



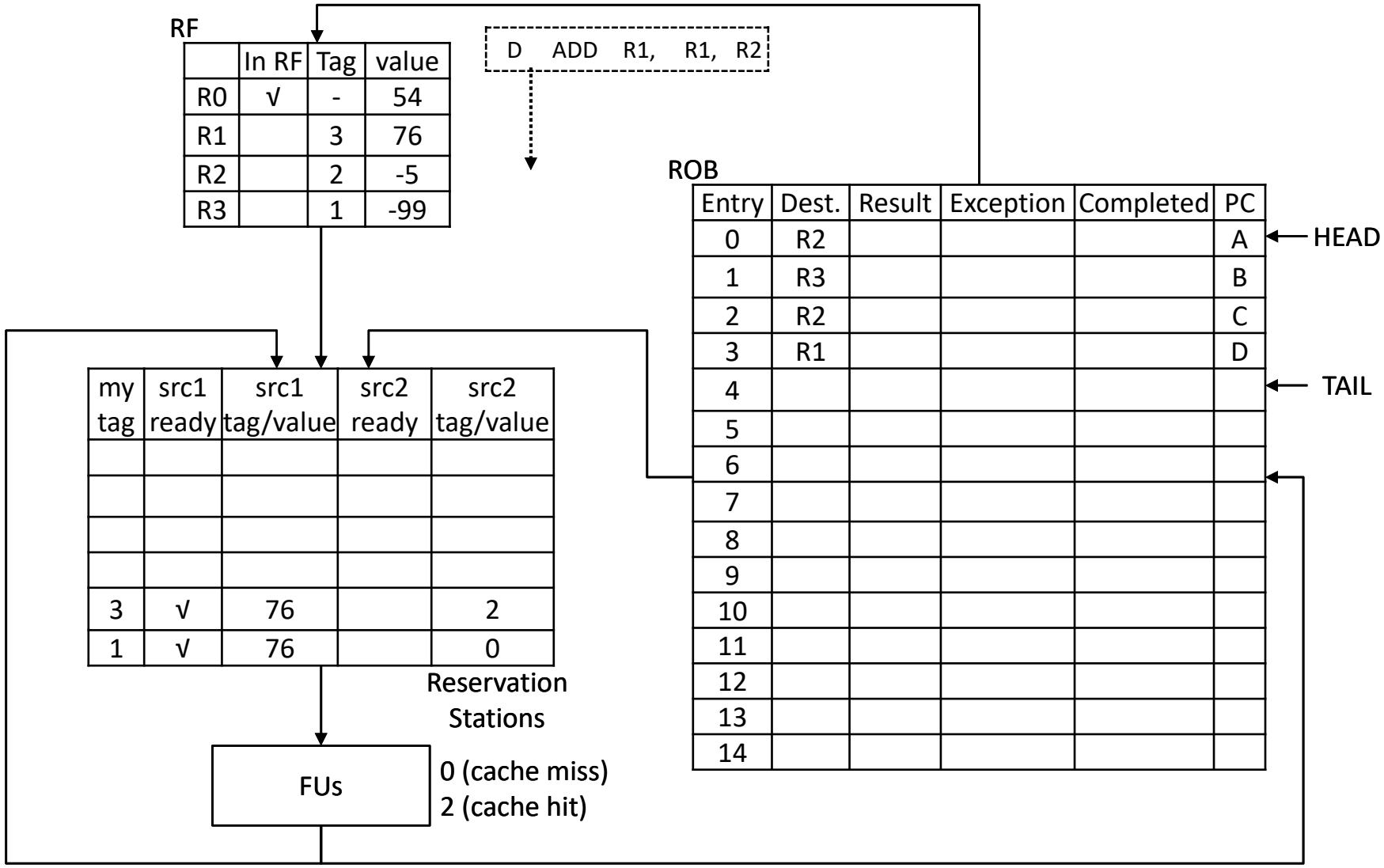
Reorder Buffer Example (7)



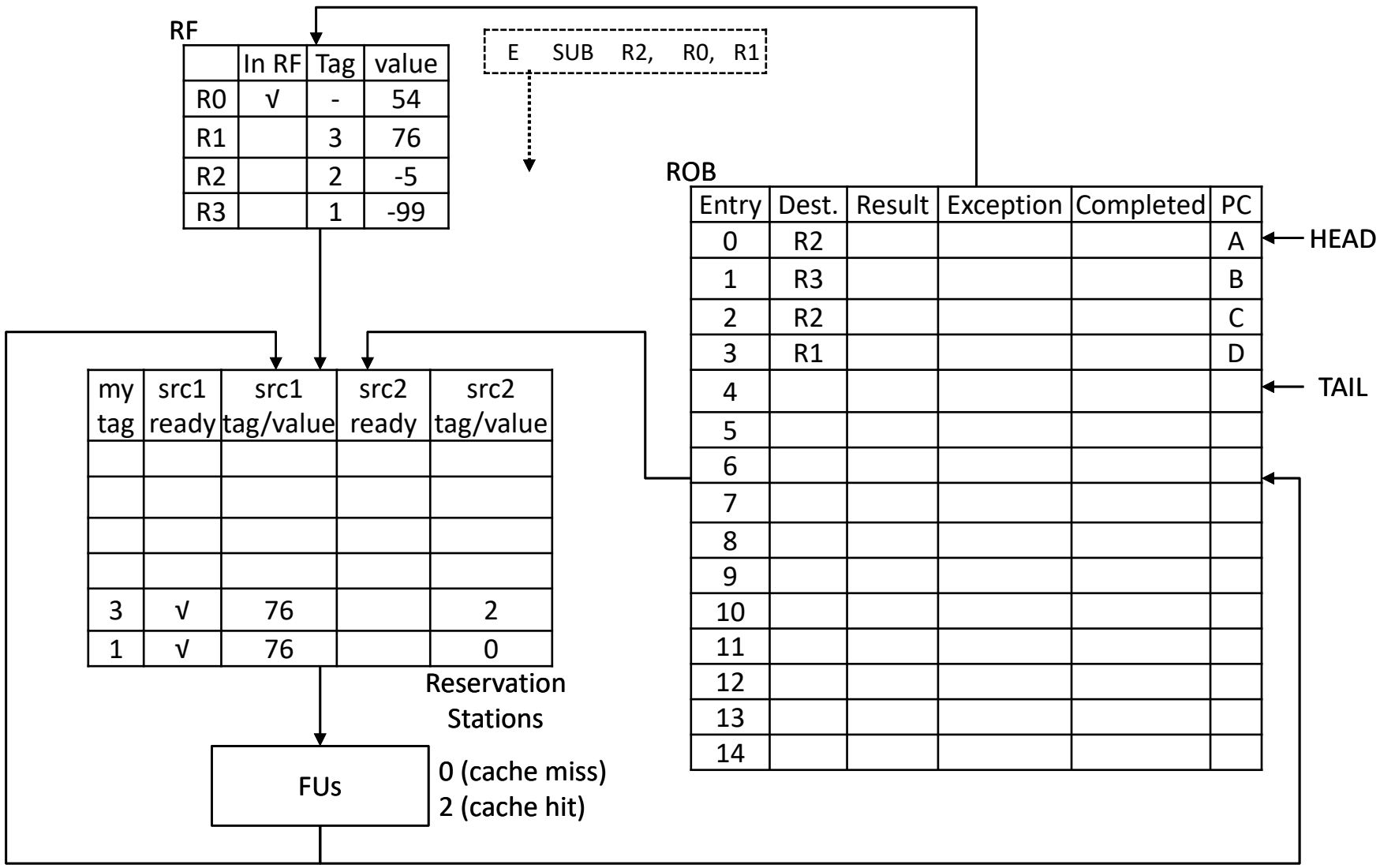
Reorder Buffer Example (8)



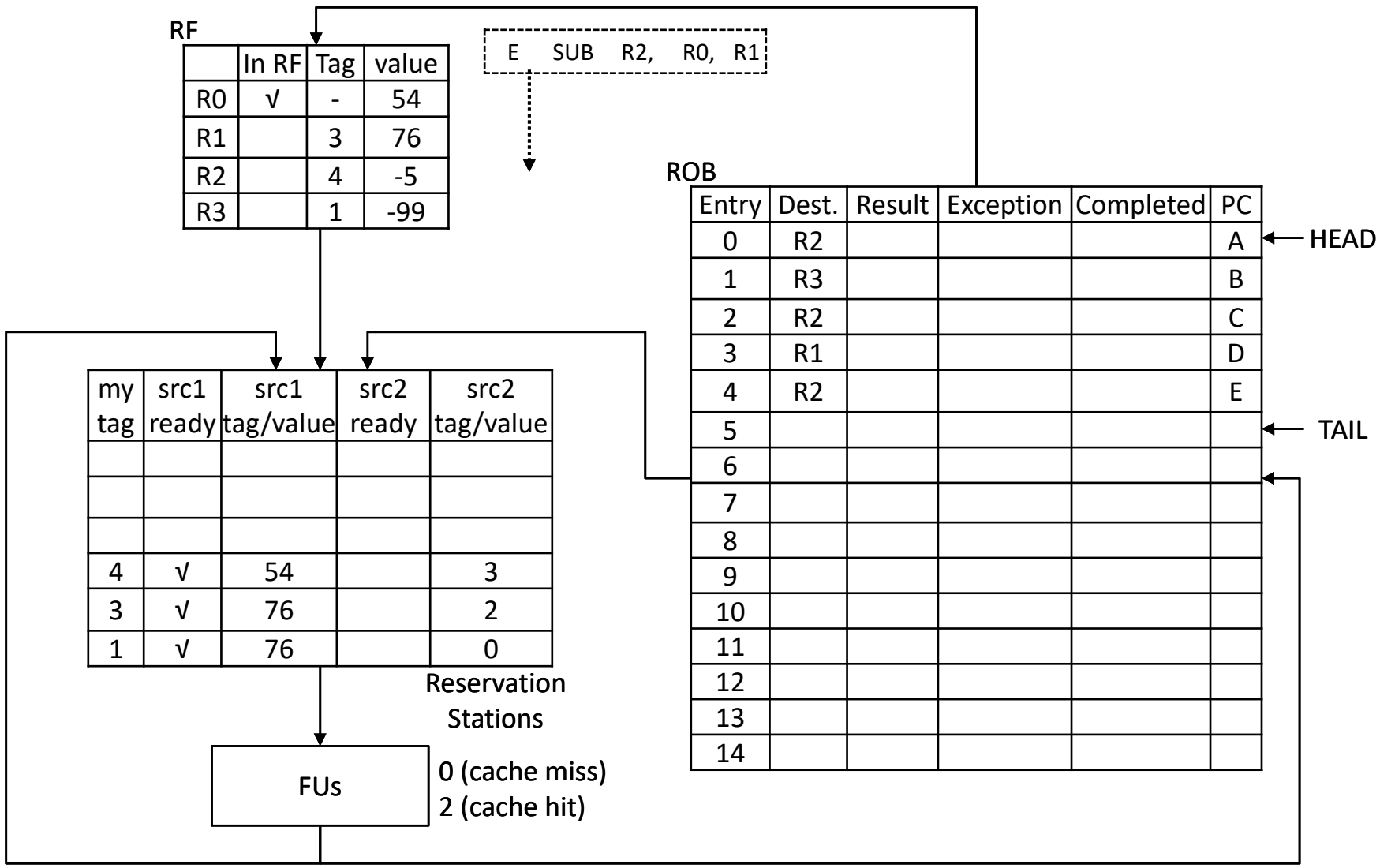
Reorder Buffer Example (9)



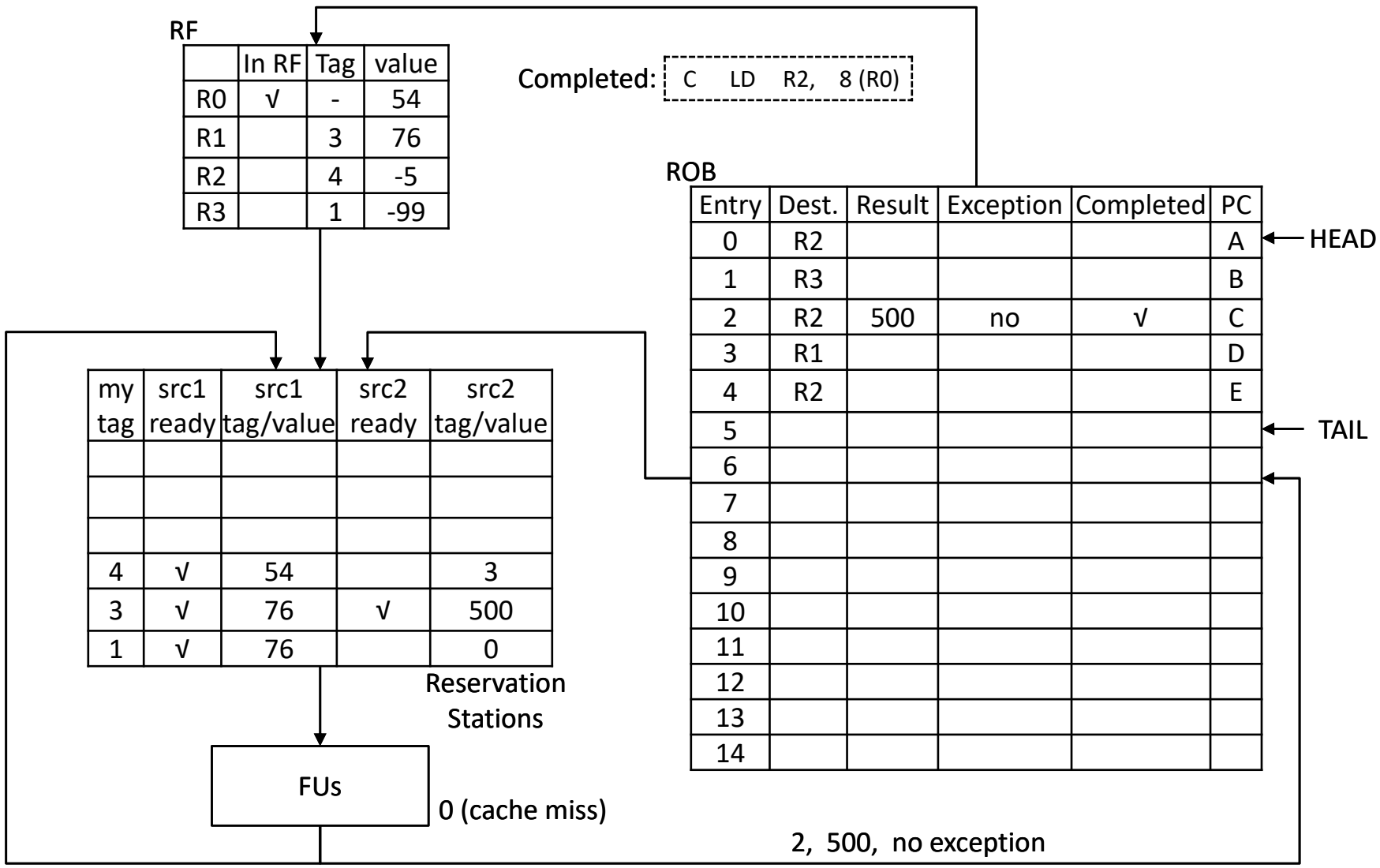
Reorder Buffer Example (10)



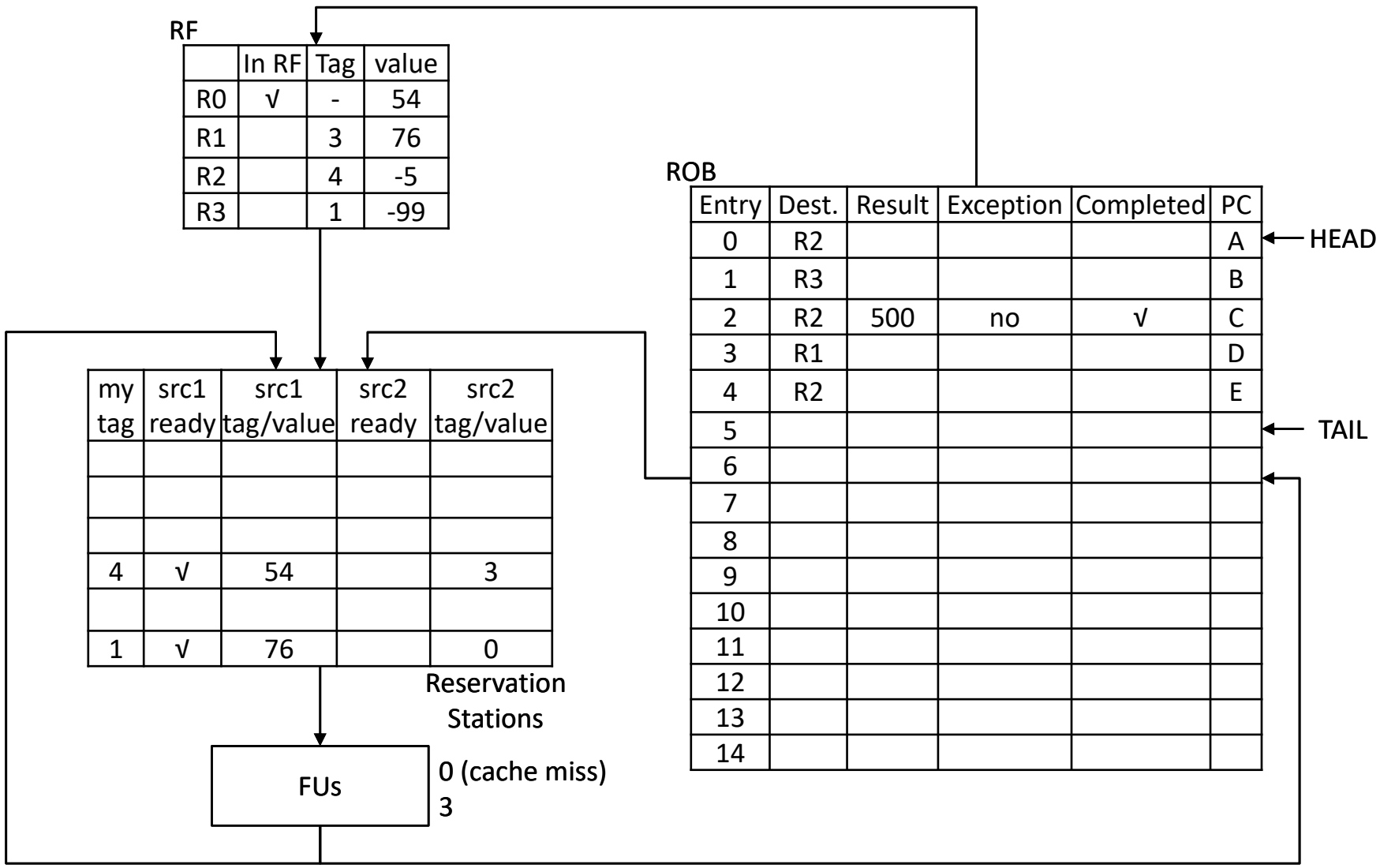
Reorder Buffer Example (11)



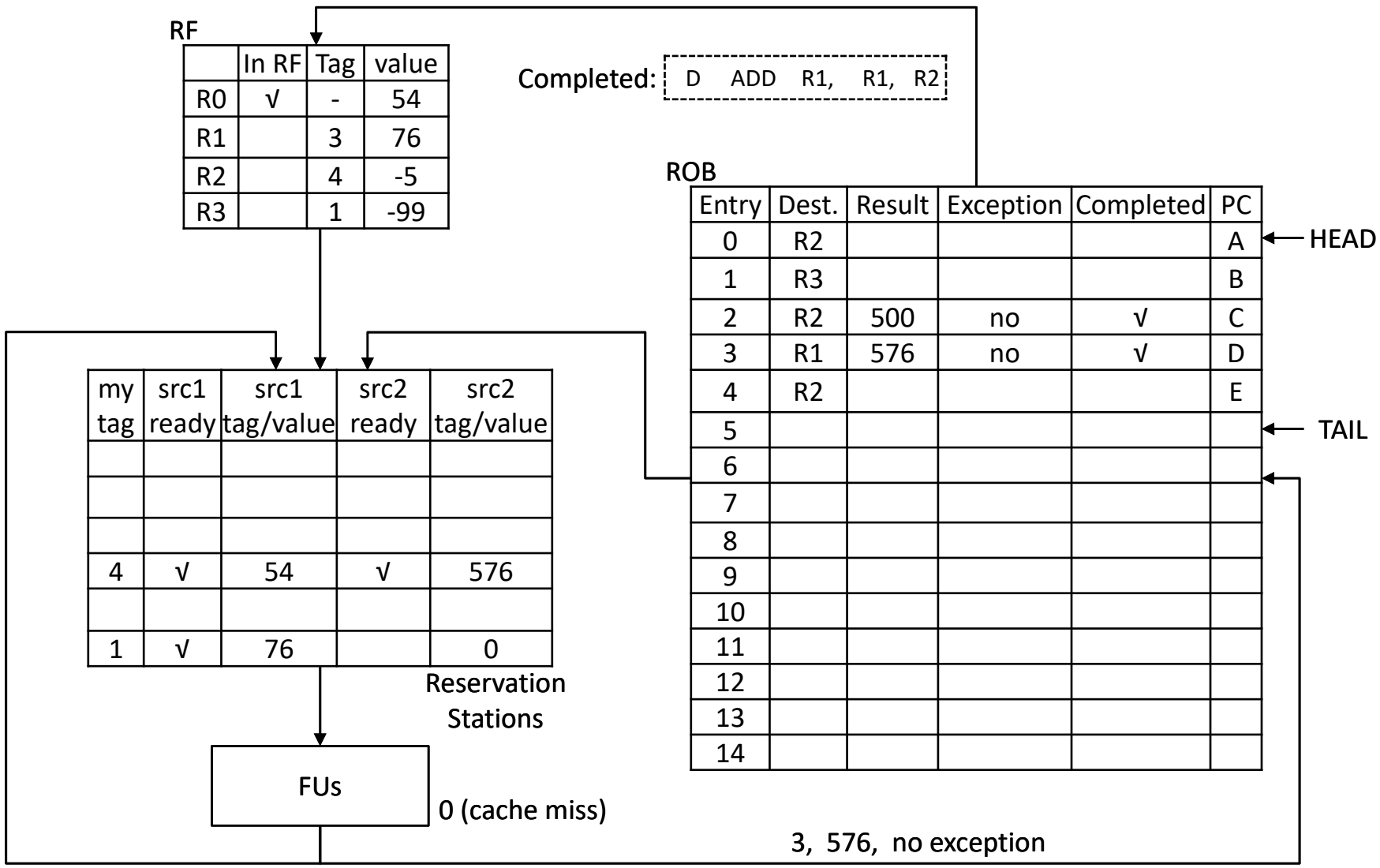
Reorder Buffer Example (12)



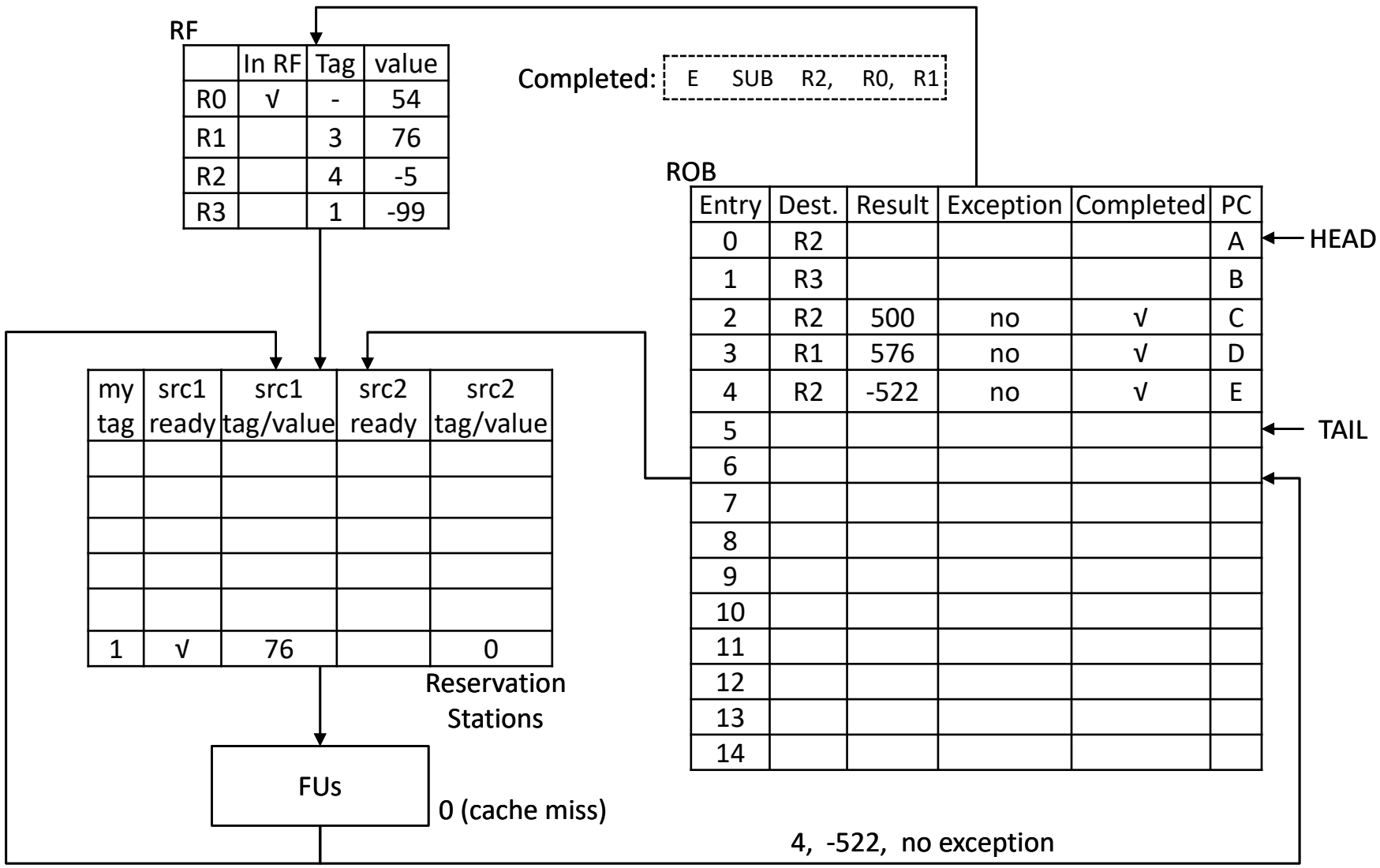
Reorder Buffer Example (13)



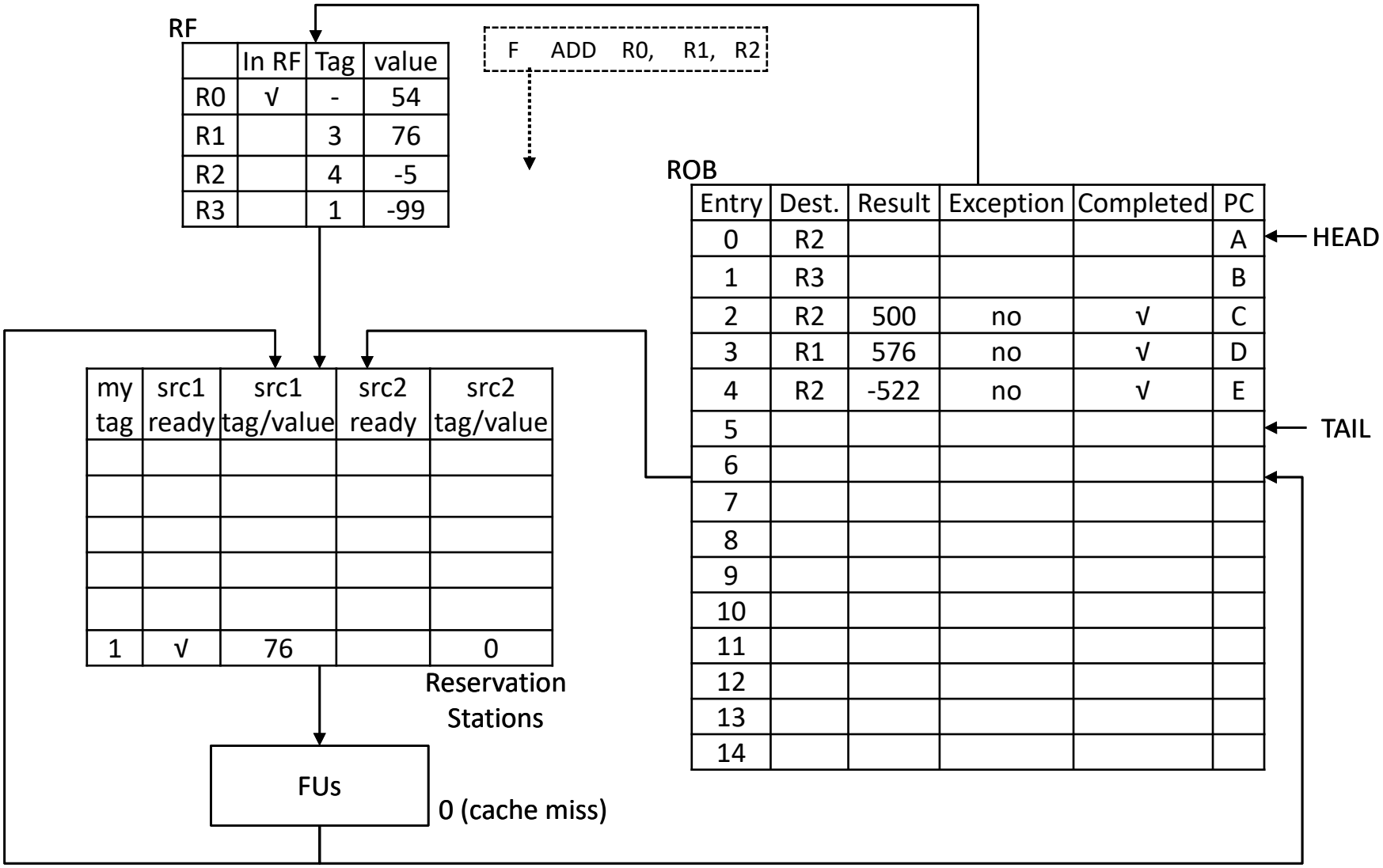
Reorder Buffer Example (14)



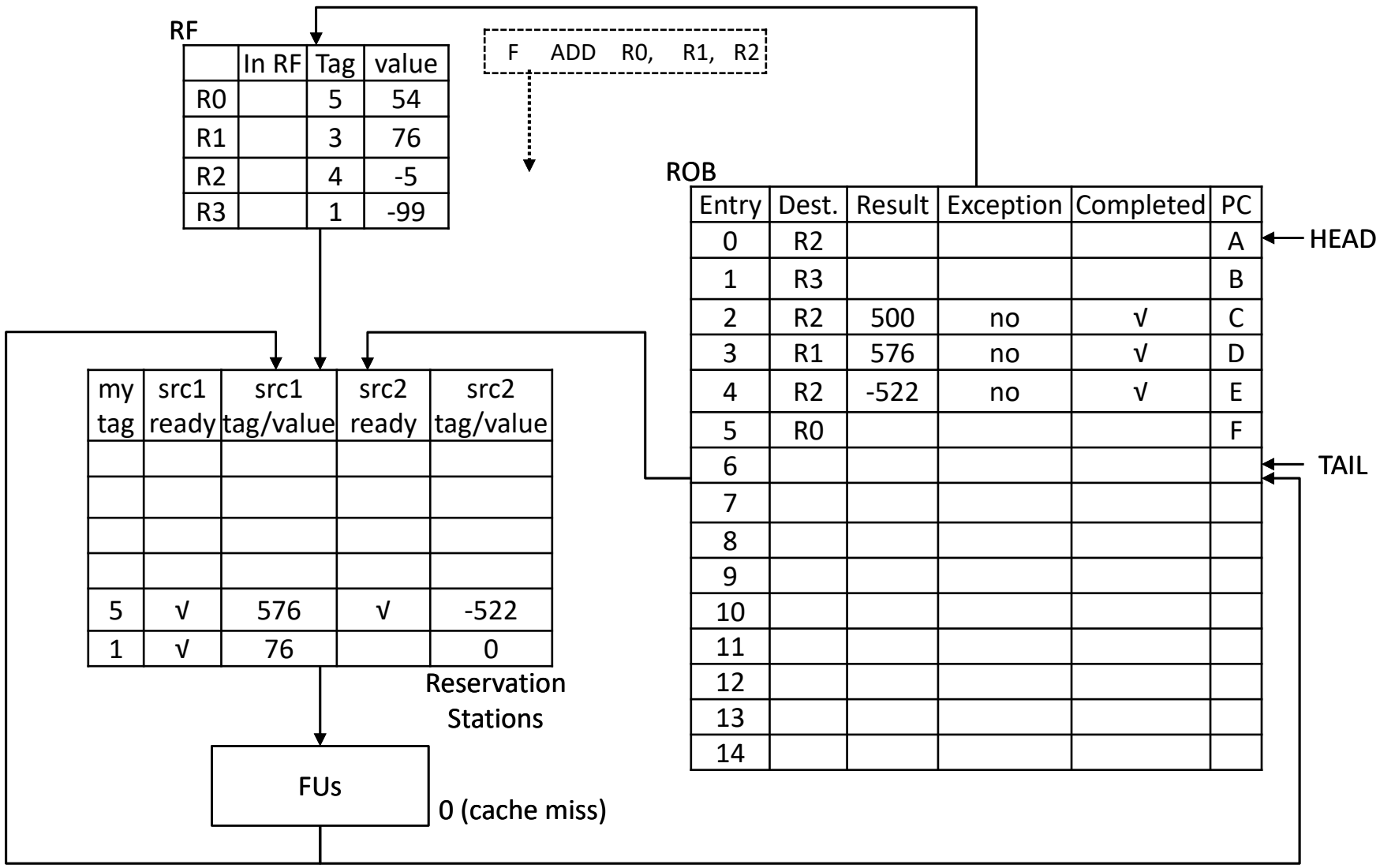
Reorder Buffer Example (15)



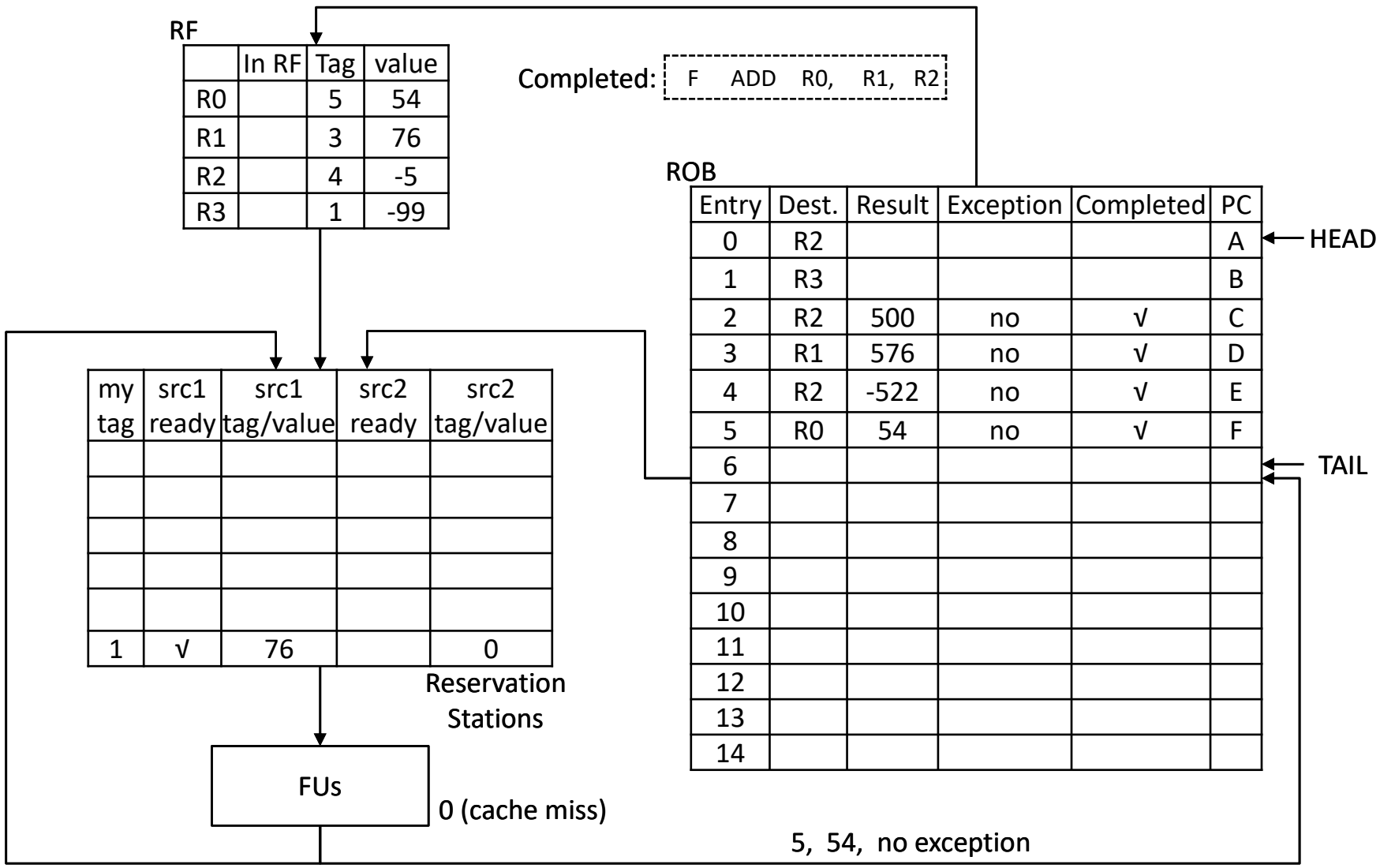
Reorder Buffer Example (16)



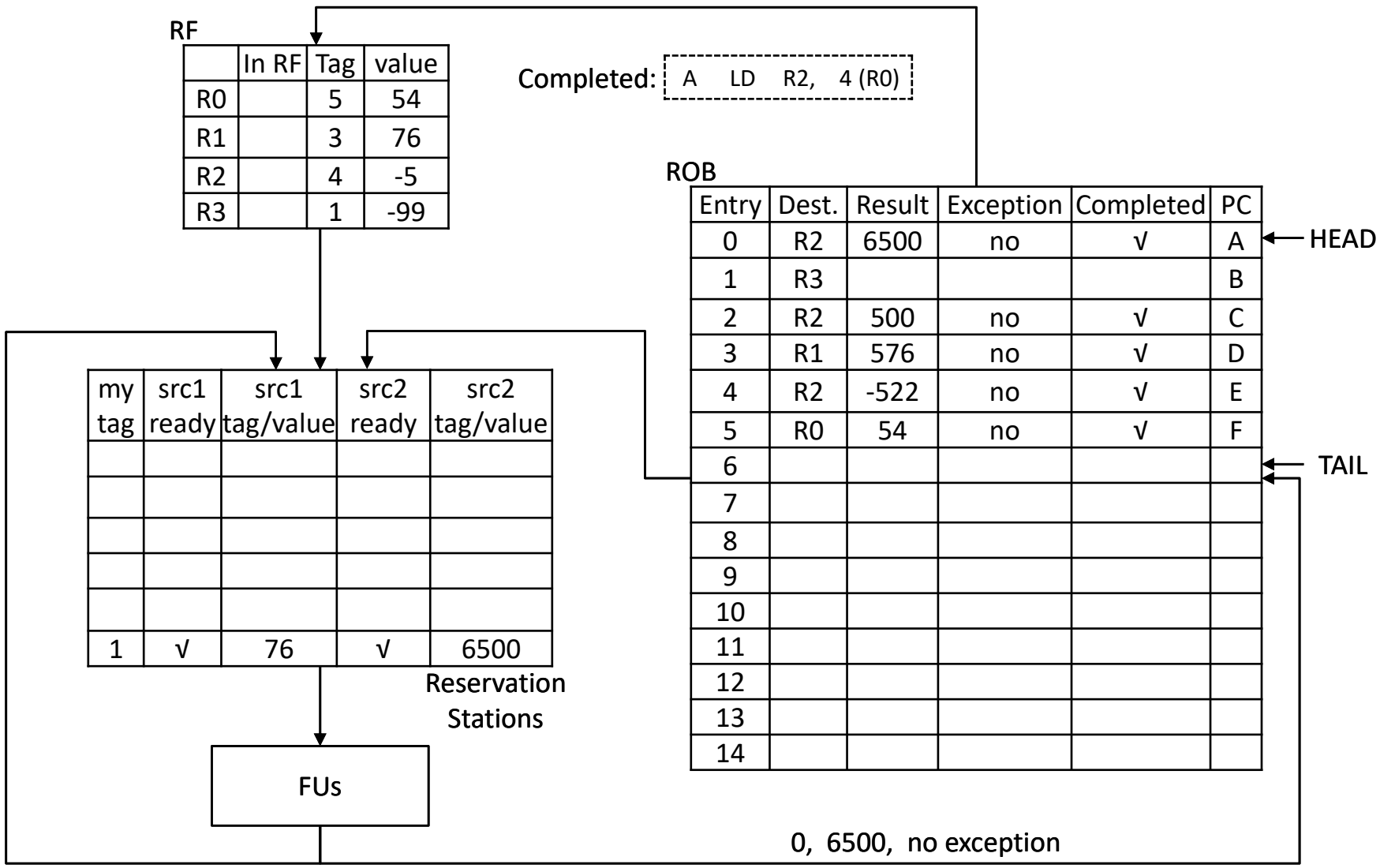
Reorder Buffer Example (17)



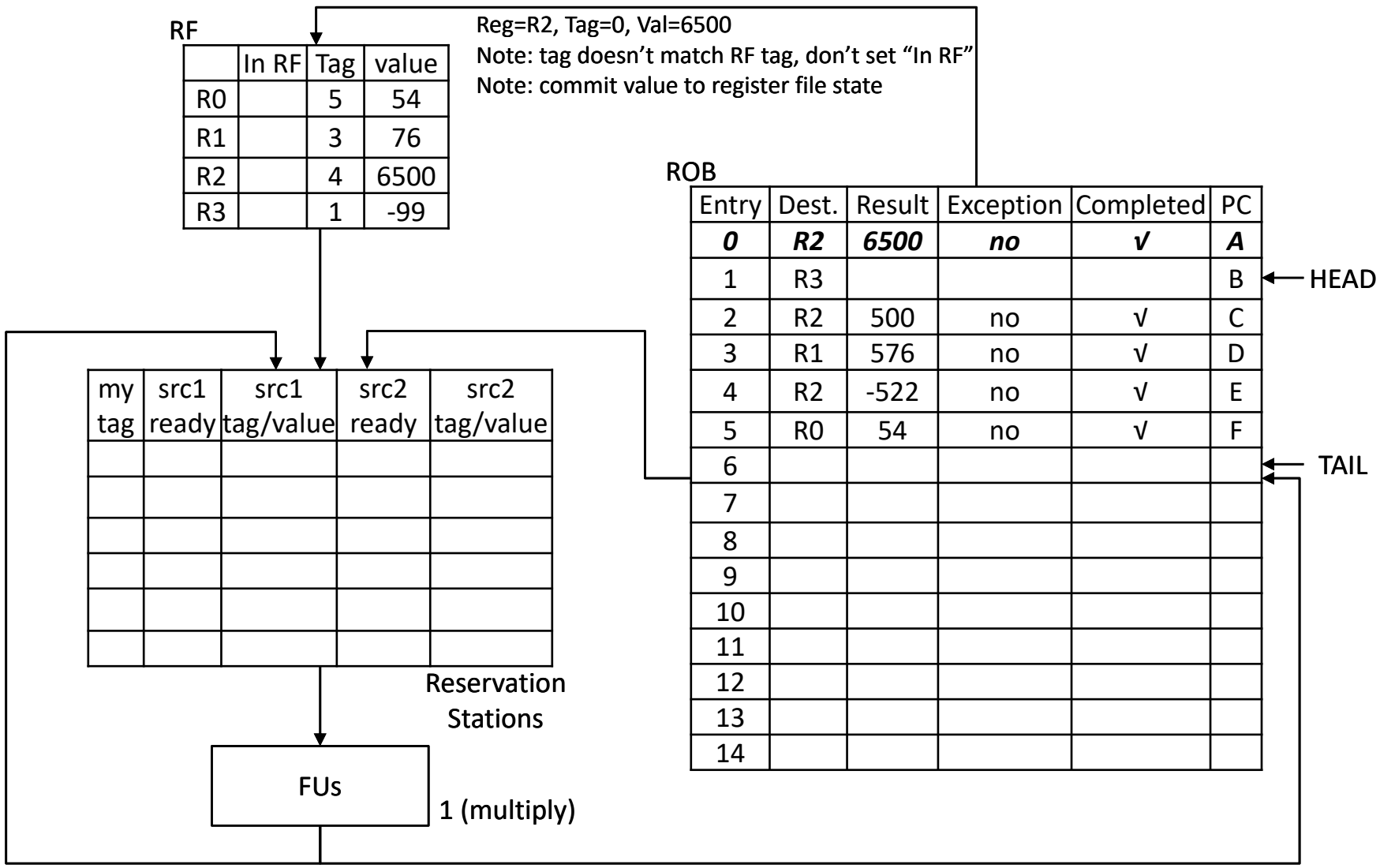
Reorder Buffer Example (18)



Reorder Buffer Example (19)



Reorder Buffer Example (20)



	In RF	Tag	value
R0		5	54
R1		3	76
R2		4	6500
R3		1	-99

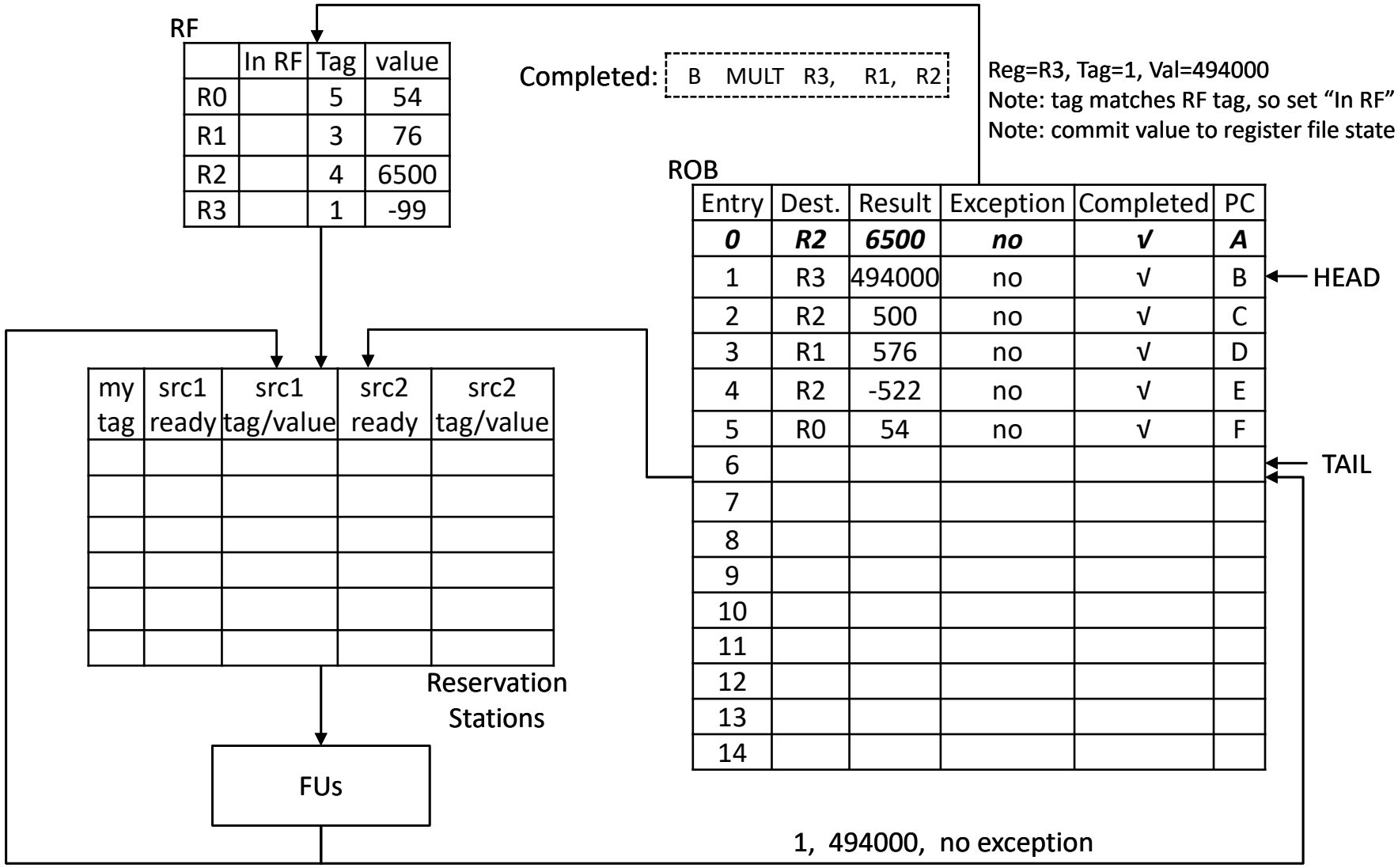
my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value

Entry	Dest.	Result	Exception	Completed	PC
0	***R2***	***6500***	***no***	***√***	***A***
1	R3				B
2	R2	500	no	√	C
3	R1	576	no	√	D
4	R2	-522	no	√	E
5	R0	54	no	√	F
6					
7					
8					
9					
10					
11					
12					
13					
14					

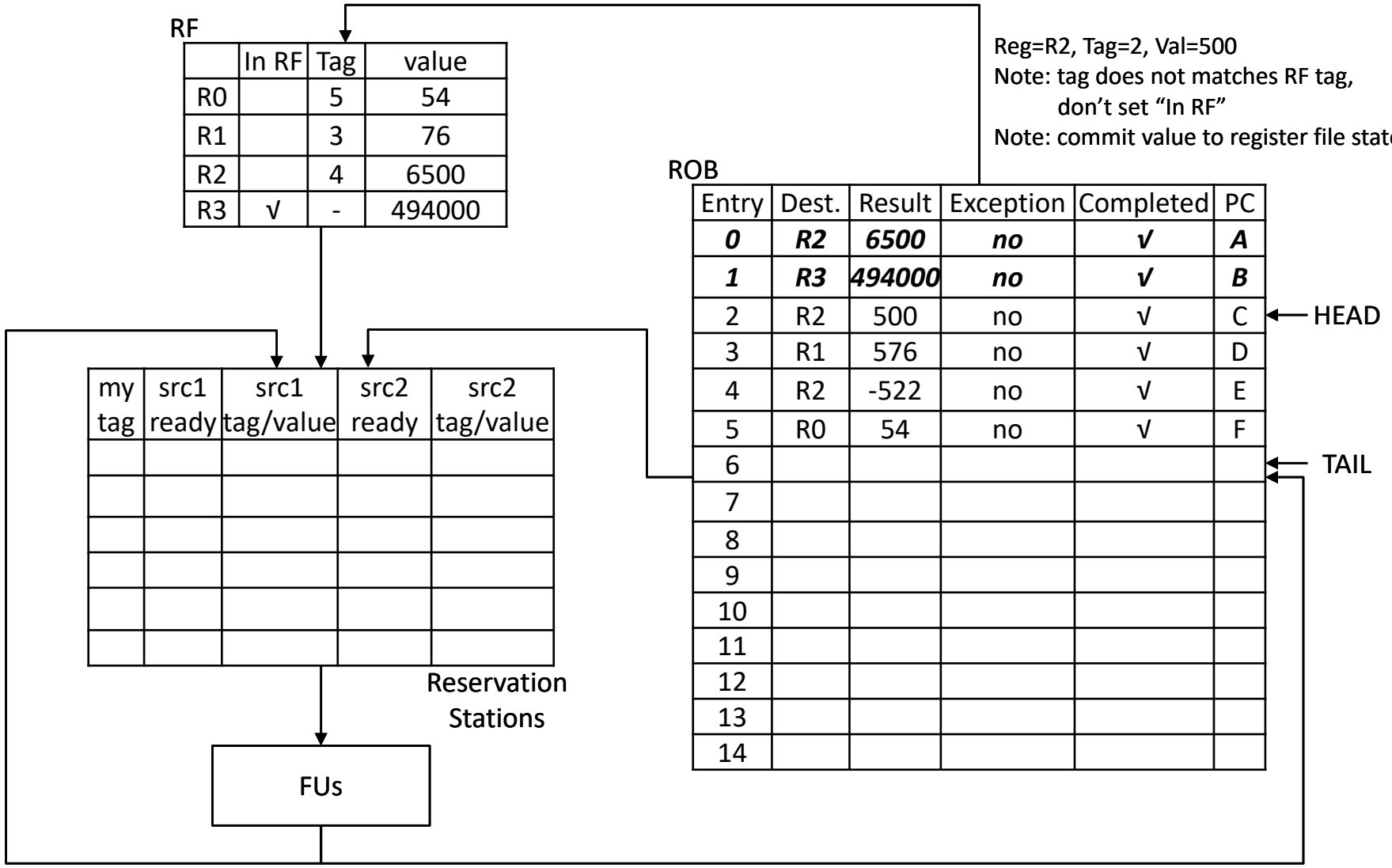
Two Scenarios

- **Multiply**
- Scenario #1: Completes without exception
- Scenario #2: Raises exception

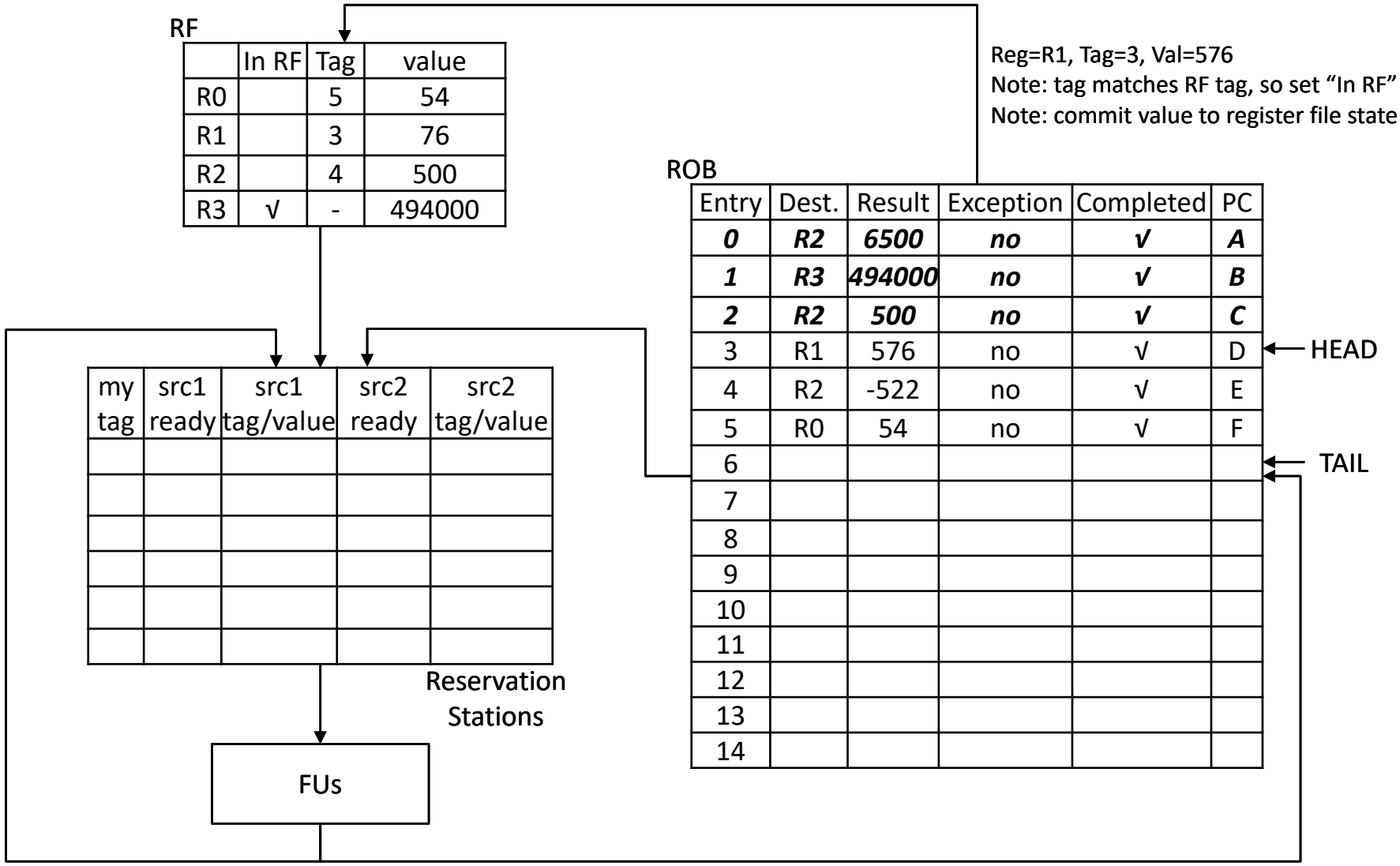
Complete without exception (1)



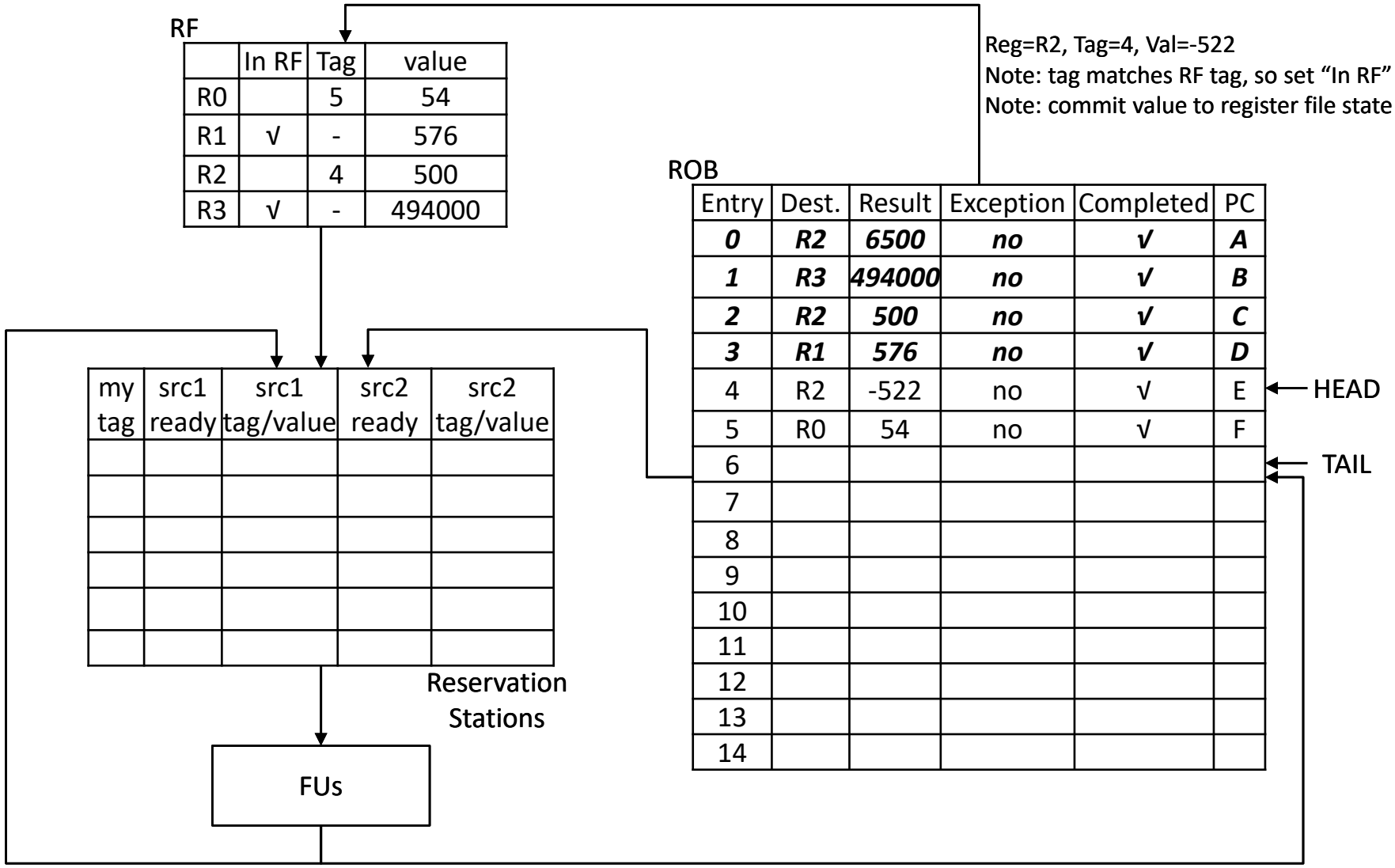
Complete without exception (2)



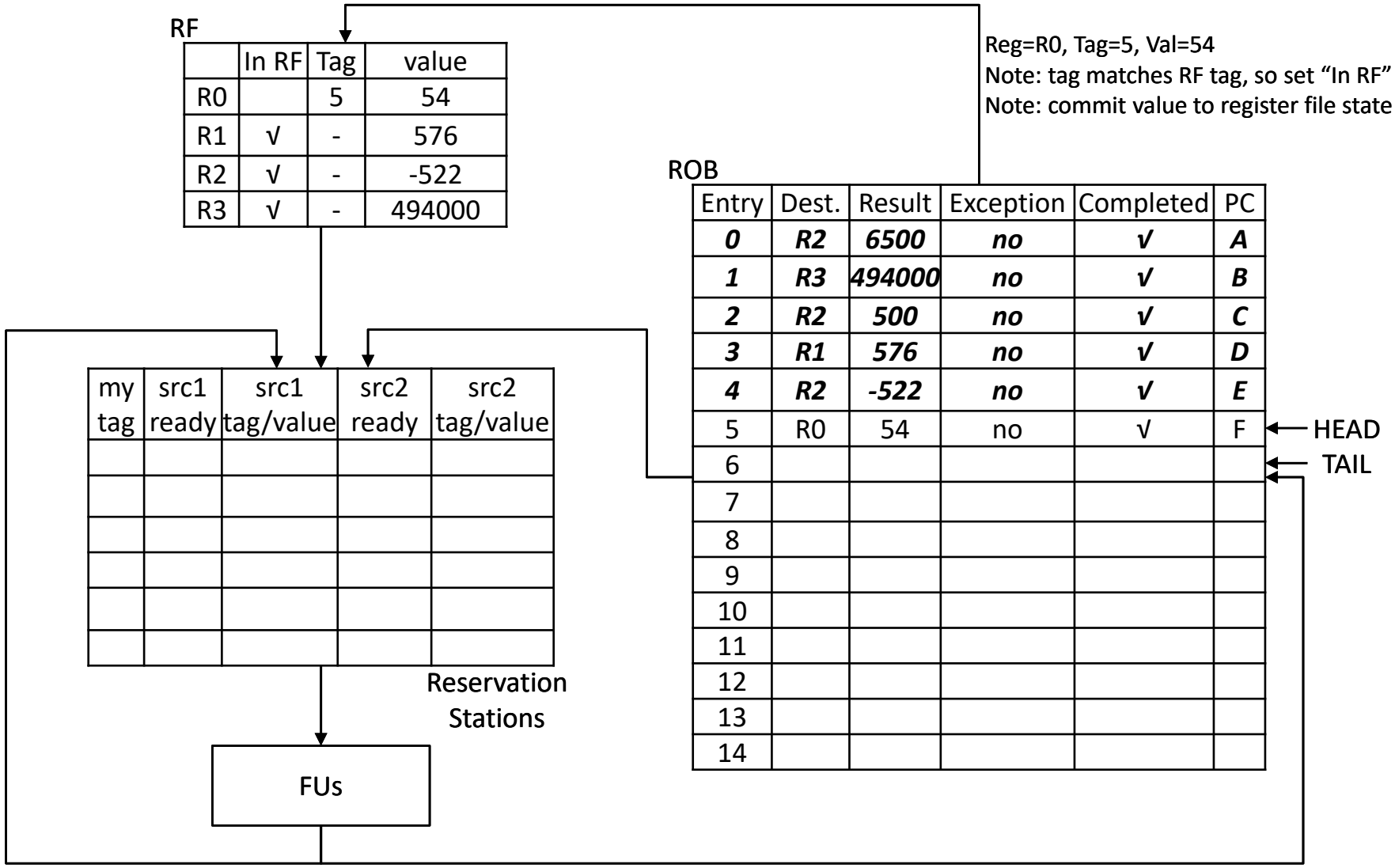
Complete without exception (3)



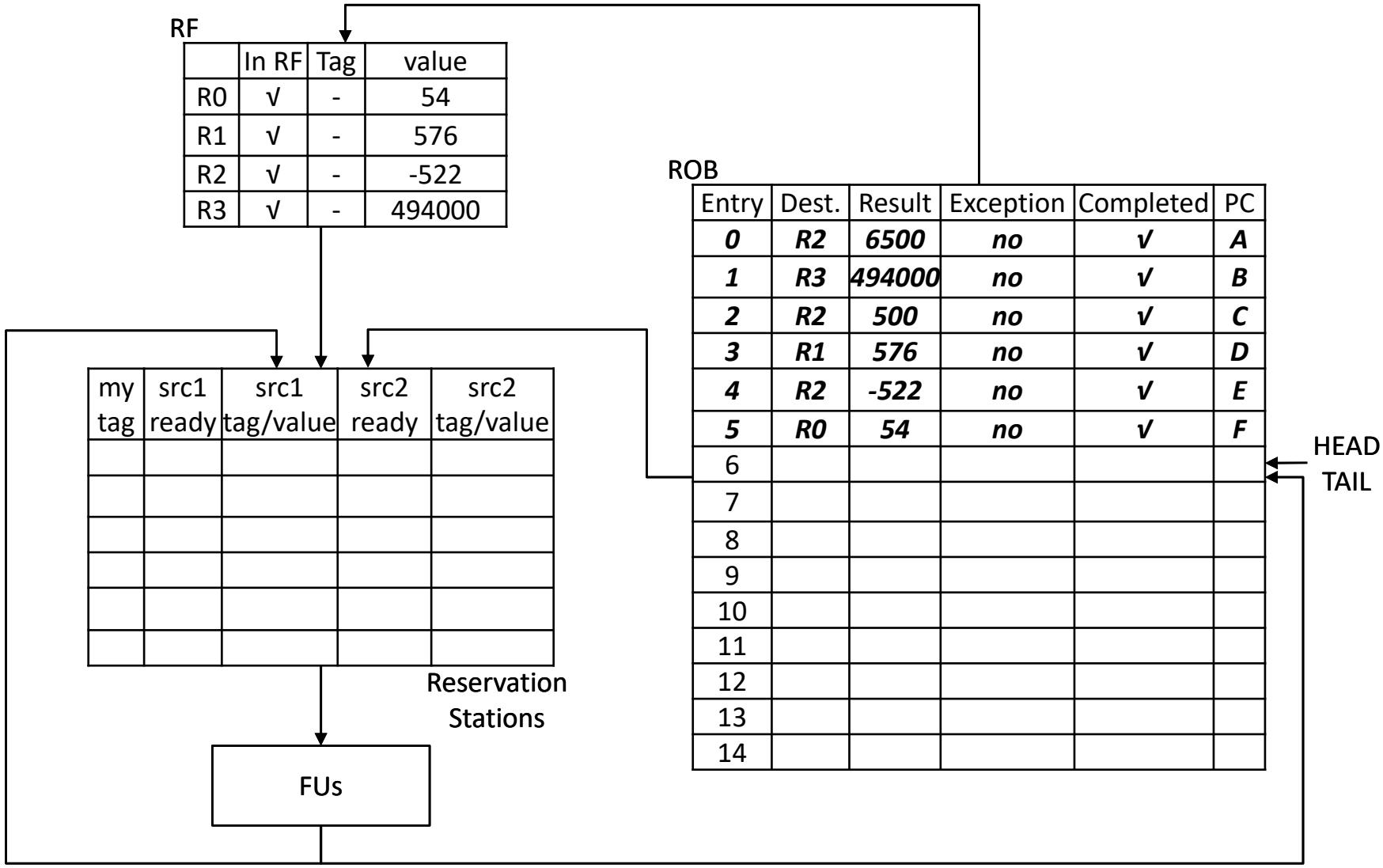
Complete without exception (4)



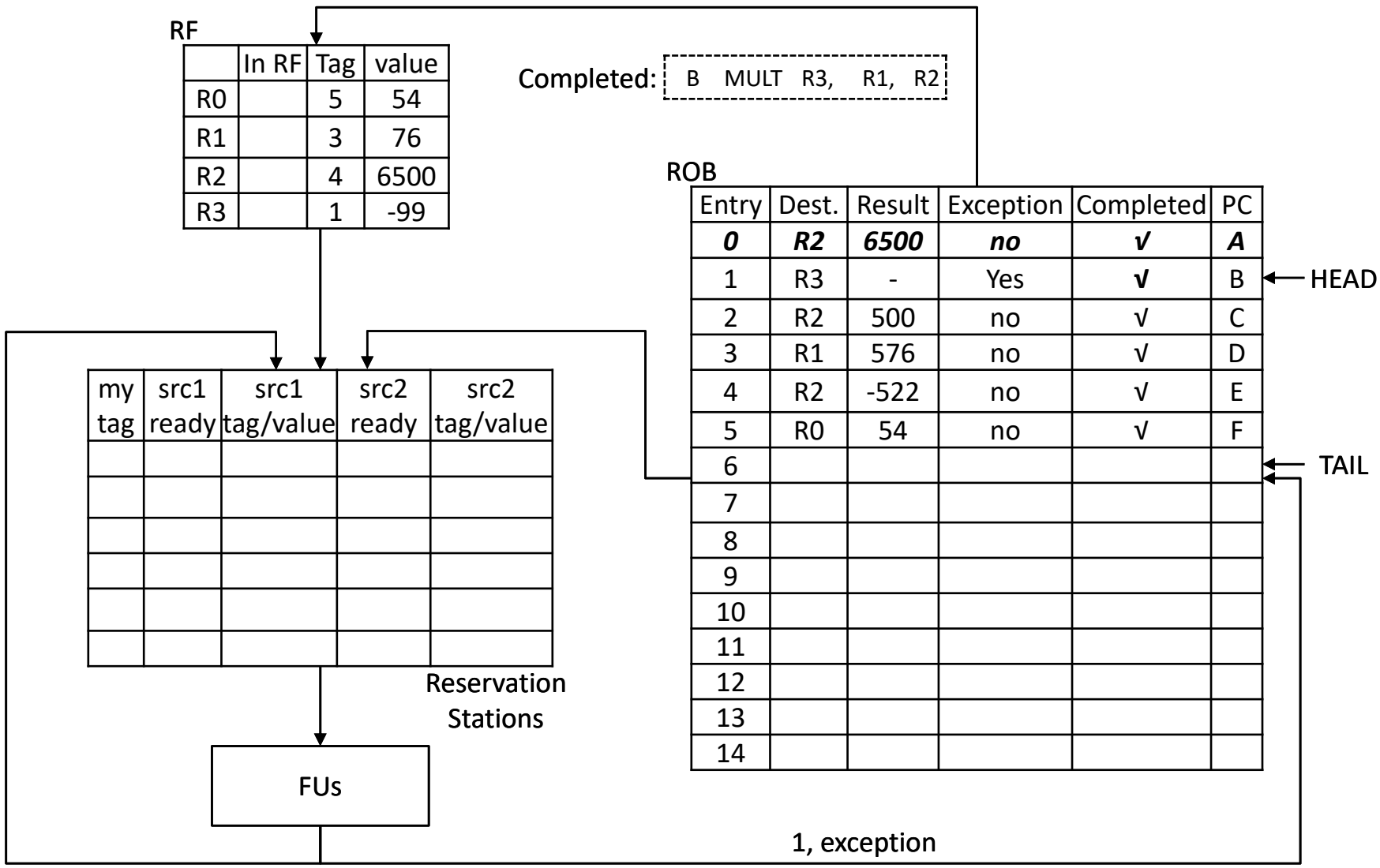
Complete without exception (5)



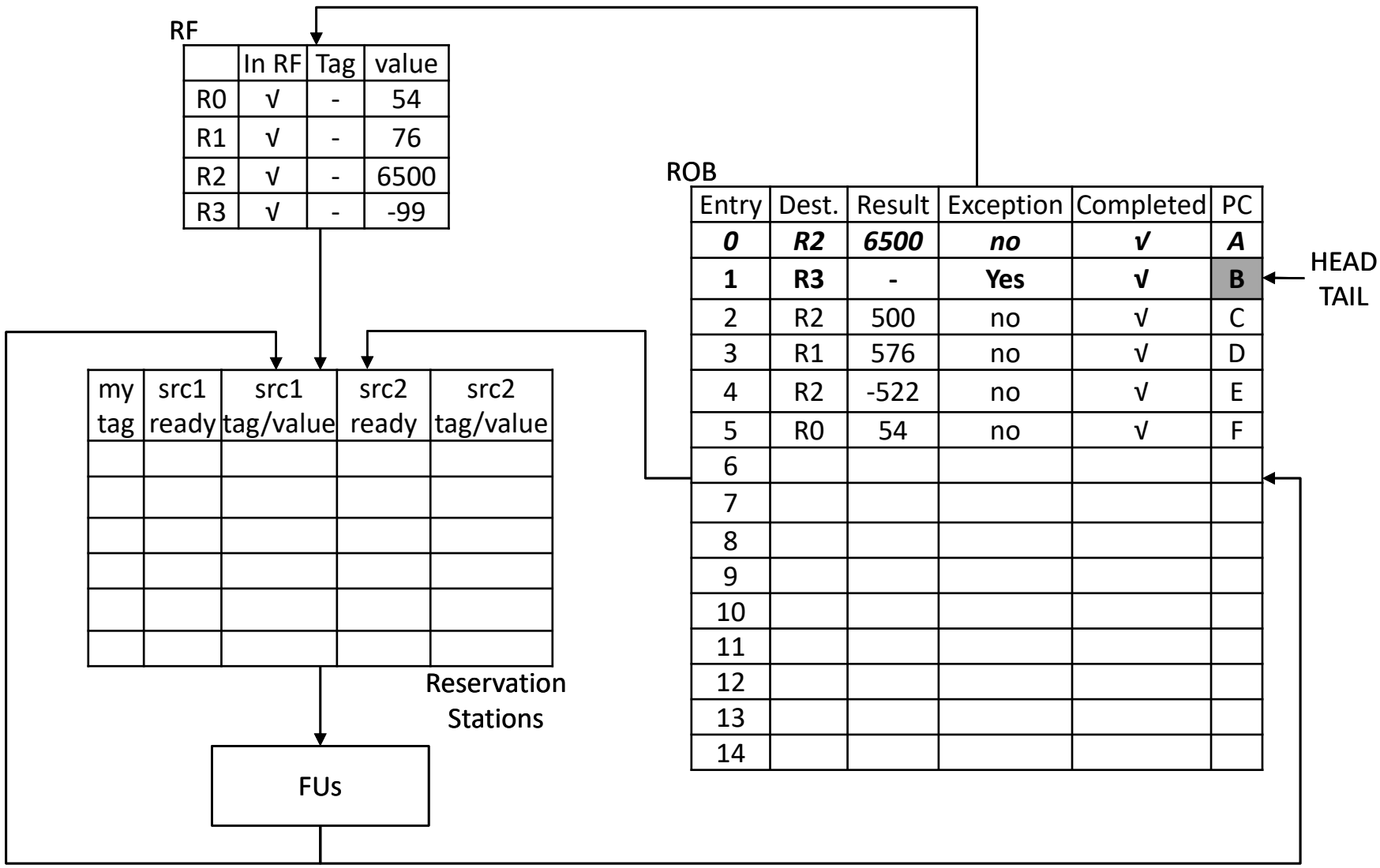
Complete without exception (6)



Raise Exception (1)



Raise Exception (2)



History Buffer (1)

- **Operation of history buffer (HB)**

- HB is a FIFO with head and tail pointer (like ROB)
- Instruction dispatch
  - ▸ Reserve HB entry at tail, advance tail pointer
  - ▸ If instruction writes register Rx, read old value of Rx from register file and save it in the HB
- Instruction execution/completion
  - ▸ Write register file when an instruction completes
  - ▸ Updates occur out-of-order: "messy register file" (imprecise)
  - ▸ If instruction caused an exception, mark exception bit in the HB for faulting instruction
- Instruction retire
  - ▸ Check instruction at the head of the HB

    if completed, check exception bit

    if no exception, simply advance head pointer

    if exception, restore precise register file state using the saved values in HB

    – "undo" the speculative updates

History Buffer (2)

Initial register file contents

R0	
R2	
R4	
R6	50
R8	60
R10	

HB and Register File when exception reaches at HEAD

R0	
R2	
R4	
R6	189
R8	-23
R10	

History Buffer

Entry	Dest.	Old Value	Exception	Valid	PC
0					
1	F6		no	√	A
2	F2		no	√	B
3	F0		yes	√	C
4	F8	60		√	D
5	F10				E
6	F6	50		√	F
7					
8					
9					
…					

HEAD →

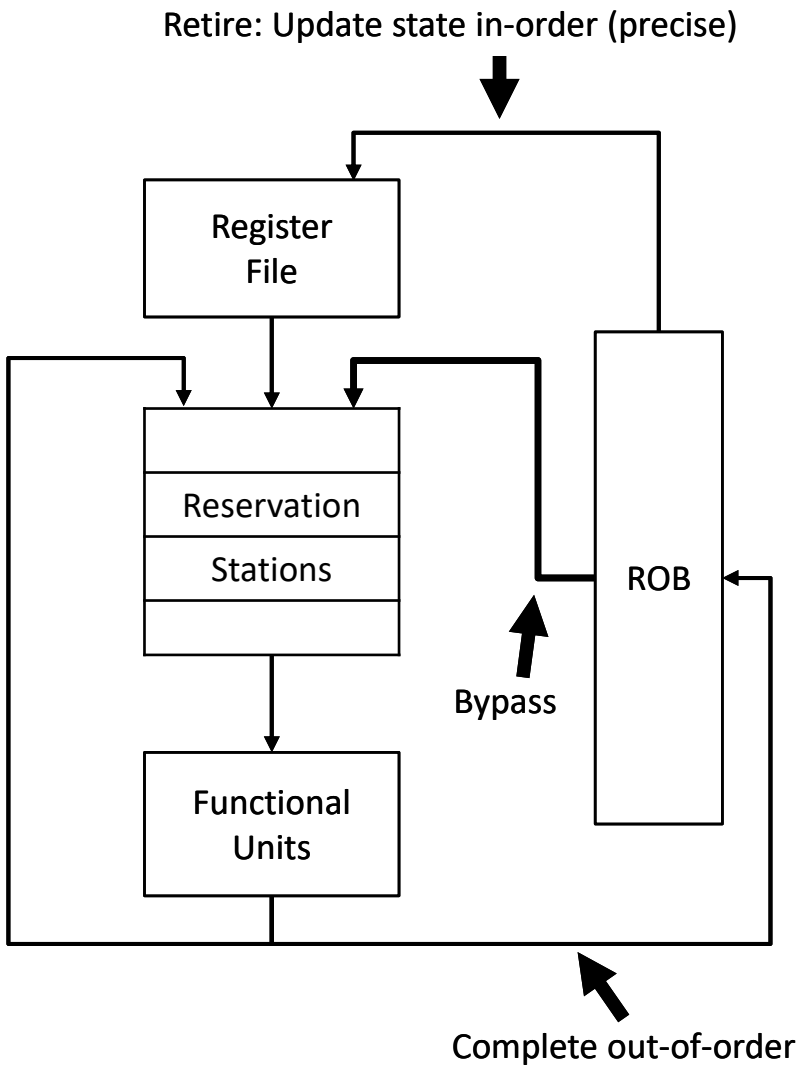
TAIL →

Recover register file state from tail to head

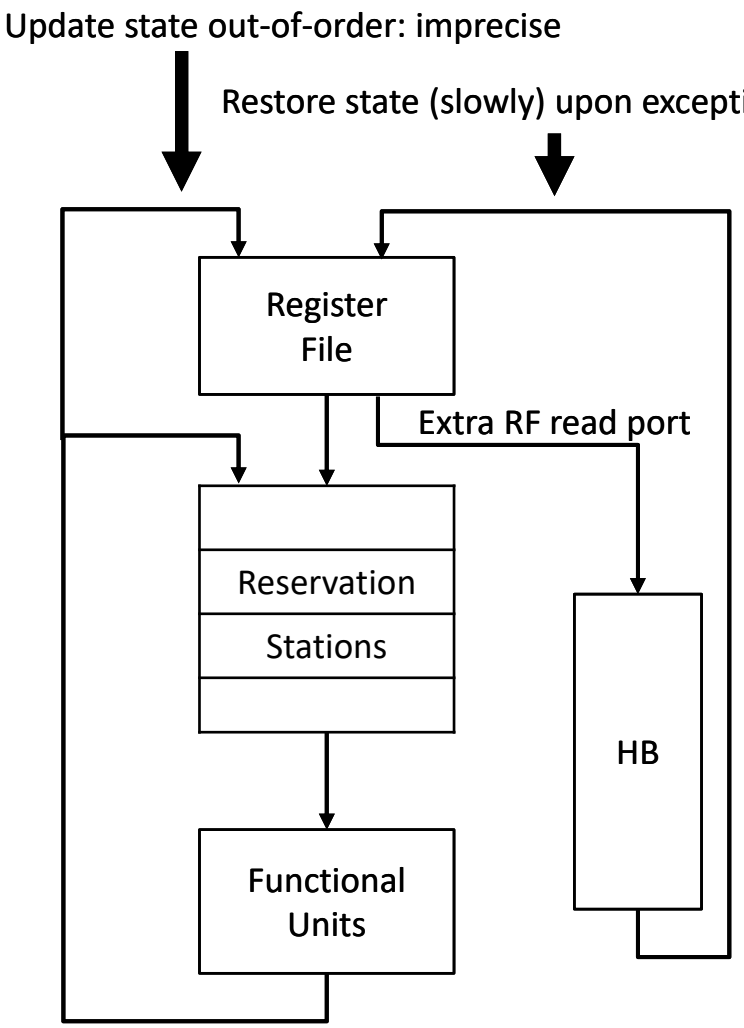
Restored (precise) register file contents

R0	
R2	
R4	
R6	50
R8	60
R10	

Precise Tomasulo Pipeline with HB



Tomasulo Pipeline with ROB



Tomasulo Pipeline with HB

History Buffer (3)

- **Advantages**

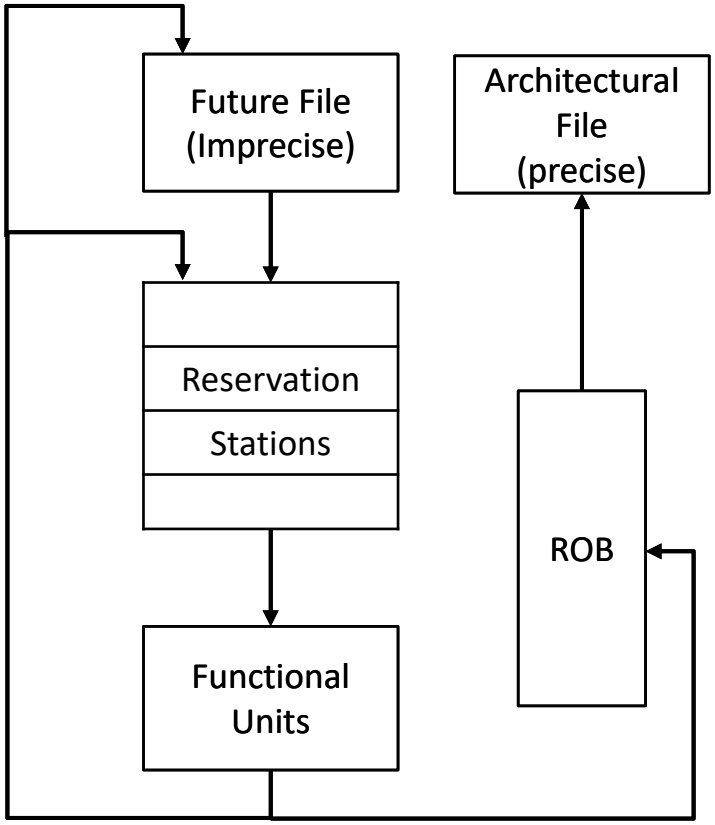
- Unlike ROB, no extra bypass is needed

- **Disadvantages**

- Add another RF read port, for reading destination register (getting old value)
- Slow exception handling
  - Restore precise values serially from HB to RF
  - Then jump to trap handler

Precise Tomasulo Pipeline with FF

- **FF operation**
  - Maintain two register files
    - Future ("messy") register file – imprecise Tomasulo pipeline is unchanged, future register file updated OOO
    - A separate, architectural register file is updated in-order by a reorder buffer
  - Exception handling
    - The architectural register file always contains precise state



Tomasulo Pipeline with Future File

Future File Analysis

- **FF analysis**

- Advantages
  - Unlike ROB, no extra bypass is needed
  - Unlike HB, no extra register file read port
- Disadvantages
  - Slow exception handling (like HB): Copy architectural file to future file, then call trap handler
  - Chip area overhead (2 register files instead of 1)

Branch Mispredictions

- **Branch mispredictions are like exceptions**

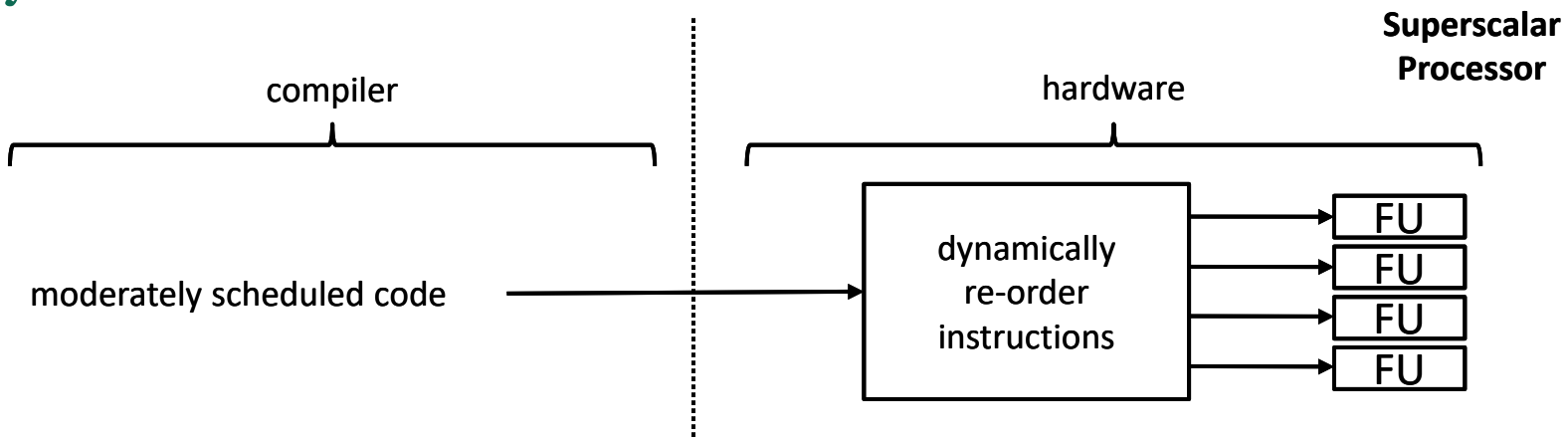
- Yes and no
- Handle exactly like exception (low performance)
  - ▸ Wait until mis-predicted branch reaches head of ROB
  - ▸ Squash entire ROB, set all "In RF" bits
  - ▸ Restart alone correct path, no problem
  - ▸ Delaying recovery until retirement is a big performance penalty
- Better performance
  - ▸ Easy part: Squash bad instructions immediately by moving ROB tail pointer to just after branch entry
  - ▸ Hard part: Restore register file tags to their state before the branch so that renaming restarts correctly
  - ▸ How? Checkpoint register file tags at every branch instruction. Restore tags from checkpoint when misprediction detected

Static Scheduling

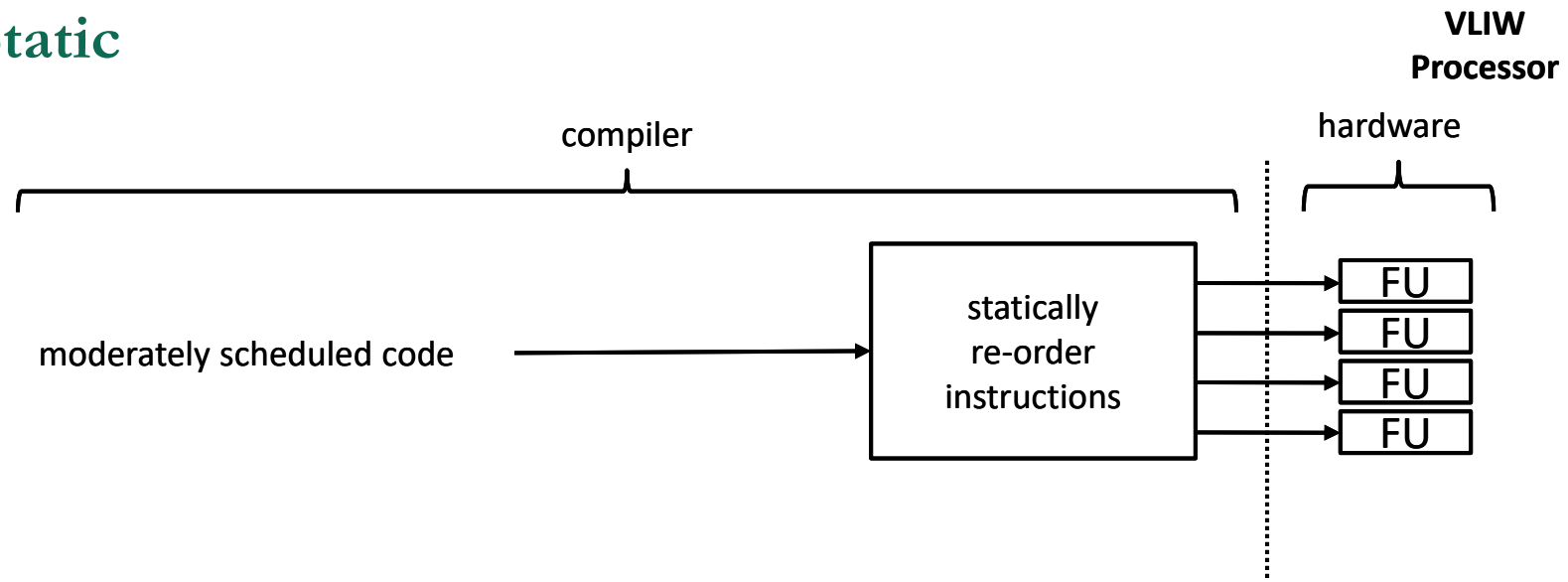
- **Have compiler re-order code to improve performance**
- A comparison of static and dynamic scheduling
  - Each has advantages/disadvantages
  - Bottom line: Fine good combination of the two
- Support for high-performance static scheduling
  - Register pressure: Large architectural register file
  - Branches: Compile-time region formation
  - Ambiguous memory dependences: Speculative loads
- Static scheduling techniques
  - Local scheduling (within a basic block)
  - Loop unrolling
  - Software pipelining (modulo scheduling)
  - Trace scheduling
  - Predication

Static/Dynamic Scheduling

- **Dynamic**



- **Static**



Local Scheduling

- **Local scheduling**
  - A.k.a., basic block scheduling
    - Advantage: simple and no speculation (no region formation)
    - Disadvantage: limited parallelism
  - Example

```
Loop: LD  F0,  0 (R1)
      (stall)
      ADDD  F4,  F0,  F2
      (stall)
      (stall)
      SD  F4,  0(R1)
      ADD  R1,  R1,  8
      BNE   R1, XXX, Loop
```

8 cycles/iteration

```
Loop: LD  F0,  0 (R1)
      (stall)
      ADDD  F4,  F0,  F2
      ADD  R1,  R1,  8
      (stall)
      SD  F4,  -8(R1)
      BNE   R1, XXX, Loop
```

7 cycles/iteration

```
Loop: LD  F0,  0 (R1)
      (stall)
      ADDD  F4,  F0,  F2
      ADD  R1,  R1,  8
      BNE   R1, XXX, Loop
      SD  F4,  -8(R1)
```

Assumes delayed branch

Loop Unrolling

- Unroll the loop

```
Loop: LD    F0,   0 (R1)
      ADDD  F4,  F0,  F2
      SD    F4,   0 (R1)
      ADD   R1,  R1,  8
      BNE   R1, XXX, Loop
```

More registers needed!

```
Loop: LD    F0,   0 (R1)
      ADDD  F4,  F0,  F2
      SD    F4,   0 (R1)
      LD    F6,   8 (R1)
      ADDD  F8,  F6,  F2
      SD    F8,   8 (R1)
      LD    F10,  16 (R1)
      ADDD  F12, F10, F2
      SD    F12,  16 (R1)
      LD    F14,  24 (R1)
      ADDD  F16, F14, F2
      SD    F16,  24 (R1)
      ADD   R1,  R1,  32
      BNE   R1, XXX, Loop
```

```
Loop: LD    F0,   0 (R1)
      LD    F6,   8 (R1)
      LD    F10,  16 (R1)
      LD    F14,  24 (R1)
      ADDD  F4,  F0,  F2
      ADDD  F8,  F6,  F2
      ADDD  F12, F10, F2
      ADDD  F16, F14, F2
      SD    F4,   0(R1)
      SD    F8,   8(R1)
      SD    F12,  16(R1)
      SD    F16,  24(R1)
      ADD   R1,  R1,  32
      BNE   R1, XXX, Loop
```

14 cycles/4 iteration
(3.5 cycles/iteration)

Positives
- Larger basic block: can reschedule operations easier
- Less branch frequency
- Less dynamic instruction count (ADD/BNEZ)

Negatives
- Expands code size
- Increased register usage in a single iteration
- Less readable

Software Pipelining (1)

- **Treat dependent operations in a loop as a pipeline**
- LD(i) → ADDD →SD(i)
- Hide dependencies by placing different stages in successive iterations
  - LD(i) goes in the first iteration
  - ADDD(i) goes in the second iteration
  - SD(i) goes in the third iteration
  - Stated another way: LD(i), ADDD(i-1), SD(i-2) go in the same iteration

```
for (i=1; i<=N; i++) {
        // a[i] = a[i] + k;
        load a[i]
        add a[i]
        store a[i]
}
```

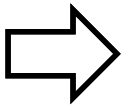
```
START-UP-BLOCK
for (i=3; I <= N; i++) {
        load a[i]
        add a[i-1]
        store a[i-2]
{
FINISH-UP-BLOCK
```

	START-UP (prologue)		LOOP			FINISH-UP (epilogue)	
load	a[i]	a[2]	a[3]	a[i]	a[N]		
add		a[1]	a[2]	a[i-1]	a[N-1]	a[N]	
store			a[1]	a[i-2]	a[N-2]	a[N-1]	a[N]

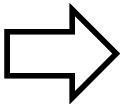
Software Pipelining (2)

- Assembly code version

```
Loop: LD    F0,   0 (R1)
      ADDD  F4,  F0,  F2
      SD    F4,   0 (R1)
      ADD   R1,  R1,  8
      BNE   R1, XXX, Loop
```



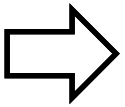
```
Start: LD    F0,   0 (R1)
       ADD   R1,  R1,  8

       ADDD  F4,  F0,  F2
       LD    F0,   0 (R1)
       ADD   R1,  R1,  8
```



```
START: LD    F0,   0 (R1)

       ADDD  F4,  F0,  F2
       LD    F0,   8 (R1)

       ADD   R1,  R1,  16
```

```
LOOP: SD    F4,   -16 (R1)
      ADDD  F4,  F0,  F2
      LD    F0,   0 (R1)
      ADD   R1,  R1,  8
      BNE   R1, XXX, Loop
```

```
FINISH: SD   F4,   -16 (R1)
        ADDD F4,  F0,  F2
        ADD  R1,  R1,  8

        SD   F4,   -16 (R1)
```



```
FINISH: SD   F4,   -16 (R1)
        ADDD F4,  F0,  F2

        SD   F4,   -8 (R1)
```

Positives
- No dependences in a loop body
- Same effect as loop unrolling, but don't need to replicate iterations (code size)

Negatives
- Still have extra code for prologue/epilogue
- Does not reduce branch frequency

Trace Scheduling

- **Select most common path – a trace**
- Use profiling to select a trace
- Allow global scheduling, i.e., scheduling across branches
- Scheduling assumes certain path
- If trace is wrong (other paths taken), execute repair code

Original code

```
b[1] = ‘old’
a[1] =
If (a[1] > 0) then
        b[1] = ‘new’ // common case
else
        X
c[1] =
```

Trace to be scheduled

```
b[1] = ‘old’
a[1] =
b[1] = ‘new’
c[1] =
If (a[1] <= 0)
        Goto A
B:
```

Repair code

```
A: restore old b[1]
   X
   maybe recalculate c[1]
   goto B
```

Predication (1)

- **ISA role**
  - Provide predicate registers
  - Provide predicate-setting instructions (e.g., compare)
  - Subset of opcodes can be guarded with predicates
- **Compiler role**
  - Replace branch with predicate computation
  - Guard alternate paths with <predicate> and <!predicate>
- **Hardware role**
  - Execute all predicated code, i.e., both paths after a branch
  - Do not commit either path until predicate is known
  - Conditionally commit or squash, depending on predicate

Original code

```
b[1] = 'old'
a[1] =
If (a[1] > 0) then
        b[1] = 'new' // common case
else
        X
c[1] =
```

Predicated code

```
b[1] = 'old'
a[1] =
pred1 = (a[1] > 0)
<pred1>: b[1] = 'new'
<!pred1>: X
c[1] =
```

Predication (2)

- **Positives**
- Larger scheduling scope
- **Negatives**
- ISA extensions: opcode pressure, extra register specifier
- May degrade performance if over-commit fetch/execute resources
- Convert control dependence to data dependence
  - Does this really fix the branch problem?
  - Not predicting control flow delays resolving register dependences
  - Can lengthen schedule w.r.t. trace scheduling
- **Above discussion is simplified**
- Predication issues are much, much more complex
- Complicating matters
  - S/W: trace scheduling, predication, superblocks, hyperblocks, …
  - H/W: selective multi-path execution, control independence, …