CDA 5106
Module 6
Instruction Level Parallelism

Yan Solihin

University of Central Florida

http://cyber.ucf.edu
http://arpers.cs.ucf.edu

Objectives and Outline

- Objectives:
  - Understand key Instruction Level Parallelism (ILP): superscalar and out-of-order scheduling
- Outline:
  - Dynamic or out-of-order (OOO) scheduling: instructions don’t have to execute in original program order
  - Register renaming
  - Precise interrupts and reorder buffer
  - Superscalar and VLIW processors

Module 6.1: Overview and CDC-6600

Objectives and Outline

- Objectives:
  - Understand the concept of dynamic or out-of-order scheduling of instructions
- Outline:
  - Problem with static scheduling
  - Scheduling instructions based on the availability of data (dataflow)
  - Decoupling Decode into Dispatch + Issue stages
  - Dynamic scheduling in CDC-6600 architecture

Overview

- Dynamic or Out of Order (OOO) scheduling
  - Expose more parallelism by ignoring artificial serial constraints of "sequential" program (insight: instructions don't have to execute in original program order)
  - Early machines: CDC-6600 and IBM 360/91
- Register renaming
  - Often used with dynamic scheduling
- Precise interrupts
  - Why we care about precise interrupts
  - Reorder buffer
- Superscalar and VLIW processors
  - Fetch and execute more than 1 instruction each cycle

True data dependences

- Consider simple DLX pipeline
  - Most operations take only 1 cycle to execute
  - Bypasses handle most RAW hazards optimally (i.e., no stalls)
  - Loads are a problem
    - Cache hit: 2 cycles (EX generates address, MEM loads data)
    - Cache miss: takes forever
    - Data dependent instruction must stall
    - Worse: the entire pipeline stalls, even other independent instructions
- Longer latencies make matters worse
  - E.g., floating-point operations
- Superscalar processing makes matters worse
  - Even with short latencies, *long serial dependence chains* limit parallelism

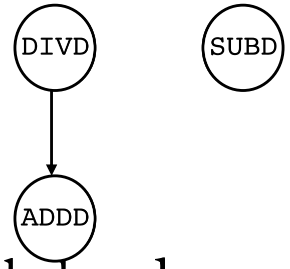
Dynamic scheduling

- Lessen impact of true data dependences
  - Only data dependent instructions must wait for data (stall)
  - Independent instructions after the dependent ones don’t stall if their data is available
  - Execute instructions in the order specified by true data dependences, not the artificial program order
  - *Dataflow graph* defines execution order

Program fragment

```
DIVD  F0 , F2, F4
ADDD  F10, F0, F8
SUBD  F8 , F8, F14
```

Dataflow Graph



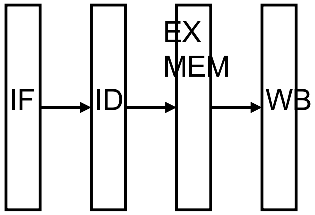
  - Divide takes many cycles: stall the dependent ADDD but not the independent SUBD
  - OOO execution allows SUBD to execute before ADDD

Pipeline modification

- Fundamental pipeline change needed
  - Stalled dependent instructions cannot block later independent instructions
  - Split ID (decode stage) into two new stages
    - DISPATCH (ID): instruction dispatch
    - ISSUE (IS): instruction schedule / issue
  - The ISSUE stage buffers instructions waiting (stalled) for data
  - This allows later instructions to make progress (FETCH and DISPATCH are not stalled)

Pipeline modification

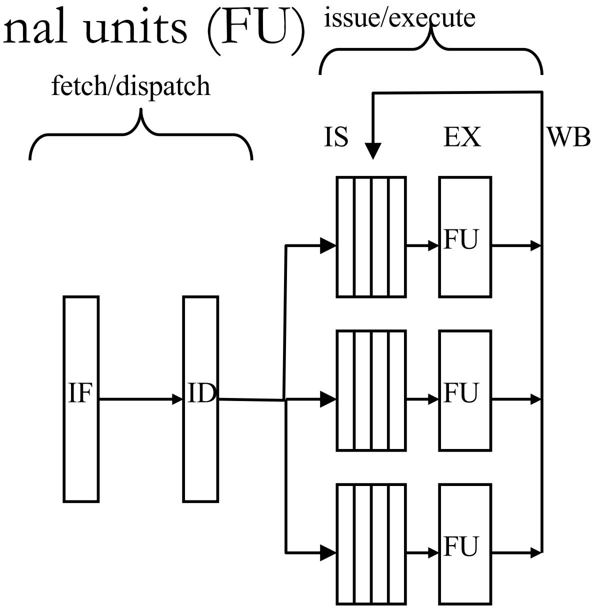
- Simple in-order pipeline



- Stalled instruction *has no where to go*
  - Stalls in decode (ID) stage
  - This stalls later instruction in IF

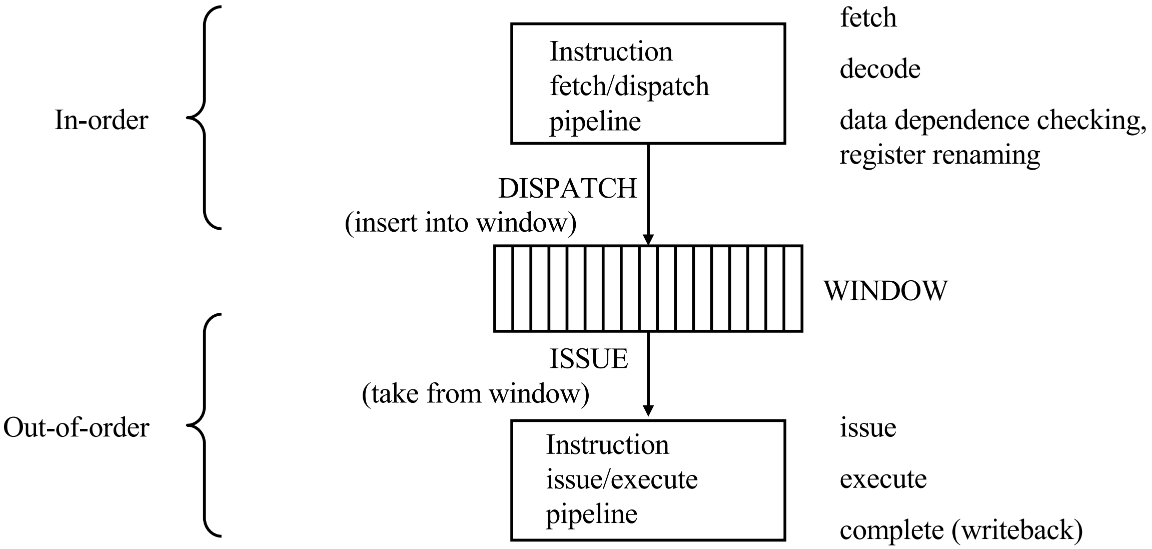
Pipeline modification

- Out-of-order pipeline
  - Two decoupled pipelines:
    - FETCH/DISPATCH pipeline brings instructions into processor in program order: scan instructions in program order – required to determine data dependences
    - ISSUE/EXECUTE pipeline executes instructions based on data dependences and available functional units (FU)



High-level view

- Out-of-order pipeline
  - Two decoupled pipelines: fetch/dispatch and issue/execute
  - Pipelines decoupled by buffers
    - Many names: reservation stations, issue queues/buffers, scheduling queues or "instruction window"



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 file in parallel (via the Common Data Bus)
  - Introduced register renaming: handles WAR/WAW hazards without stalling

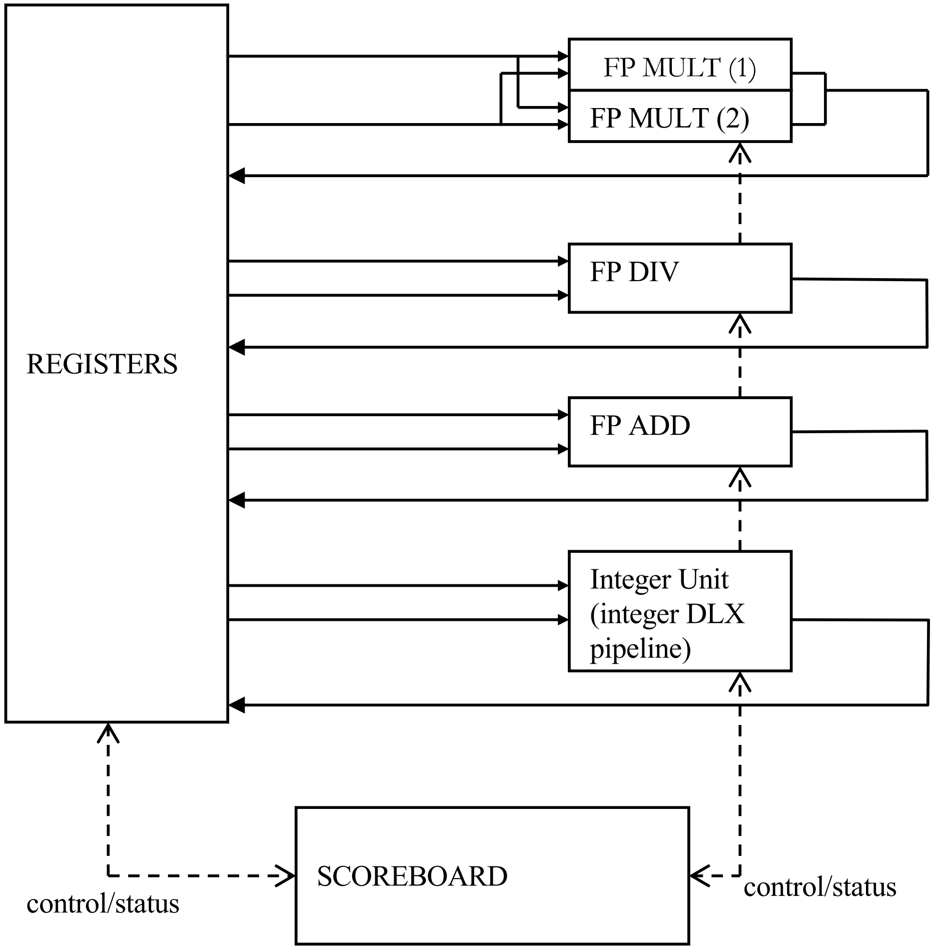
CDC-6600 (DLX version)

- Four stages after fetch
  - Dispatch
    - Check for structural and WAW hazards
      - Structural: stall in dispatch stage if FU 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
      - Then issues to FU 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

Warning – H&P naming

- H&P uses different names than what we will use
  - US: dispatchTHEM: issue (ugh!)
  - US: issue THEM:read operands (this *does* occur)
  - US: execute THEM: execute
  - US: write result THEM: write result

CDC-6600 (DLX version)



Scoreboard

- Three data structures
  1. Instruction Status
     - Which stage is the instruction in
  2. Functional Unit Status
     - Busy - FU is busy executing another instr.
     - Op - what instr. is the FU busy with
     - F_i - destination register
     - F_j, F_k - source registers
     - Q_j, Q_k - func. units producing source regs
     - R_j, R_k - flags indicating src regs are ready
  3. Register Result Status
     - Which FU is going to write each register

Running example

- Example used for CDC and Tomasulo

```
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

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	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

- MULTD about to write result…

Instruction Status

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

LD F6 , 34(R2)	ID	IS	EX	EX	WR																				
LD F2 , 45(R3)		ID	IS	EX	EX	WR																			
MULTD F0 , F2, F4			ID	IS	IS	IS 1	IS	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	WR							
SUBD F8 , F6, F2				ID	IS	IS 2	IS	EX	EX	WR															
DIVD F10, F0, F6					ID	IS	IS	IS	IS	IS	IS	IS	IS	IS	IS	IS	IS	IS 3	IS	EX	…..	…..			
ADDD F6 , F8, F2						ID	ID	ID	ID 5	IS 4	IS	EX	EX	EX	EX	EX	EX	EX	EX 6	WR					

=> Execution latencies: LD (2 – agen + 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

Structural

WAR

1. LD-MULTD (F2)
2. LD-SUBD (F2)
3. MULTD-DIVD (F0)
4. SUBD-ADDD (F8)
5. SUBD-ADDD (ADD unit)
6. DIVD-ADDD (F6)

Remaining bottlenecks

- CDC-6600 does good job of dynamic scheduling around RAW hazards
- Remaining performance limitations
  - Amount of instruction-level parallelism (ILP) in the program
    - Are there sufficient data-independent operations?
    - Needs large window to look farther ahead
    - Requires branch prediction to support large window
  - Number of scoreboard entries (window size)
    - Dictates how far processor can look ahead
  - Number and type of functional units, register ports, etc.
    - Structural hazards
    - Lengthy binding of FU to Instruction
  - 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 – even when operands were ready some time ago)
    - WAW handled like a structural hazard in dispatch

Module 6.2: Tomasulo's Algorithm

Objectives and Outline

- Objectives:
  - Understand the foundation of out-of-order scheduling: Tomasulo algorithm
- Outline:
  - Key aspects of Tomasulo and differences vs. CDC-6600
  - Pipeline stages and structures in Tomasulo (operation queues, reservation stations, common data bus)
  - Register renaming as a key technique to remove WAR/WAW dependences and hazards
  - Step by step illustration of Tomasulo's algorithm

Tomasulo's Algorithm

- Born of necessity
  - Used in IBM 360/91 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 <u>instances</u> of the same register specifier

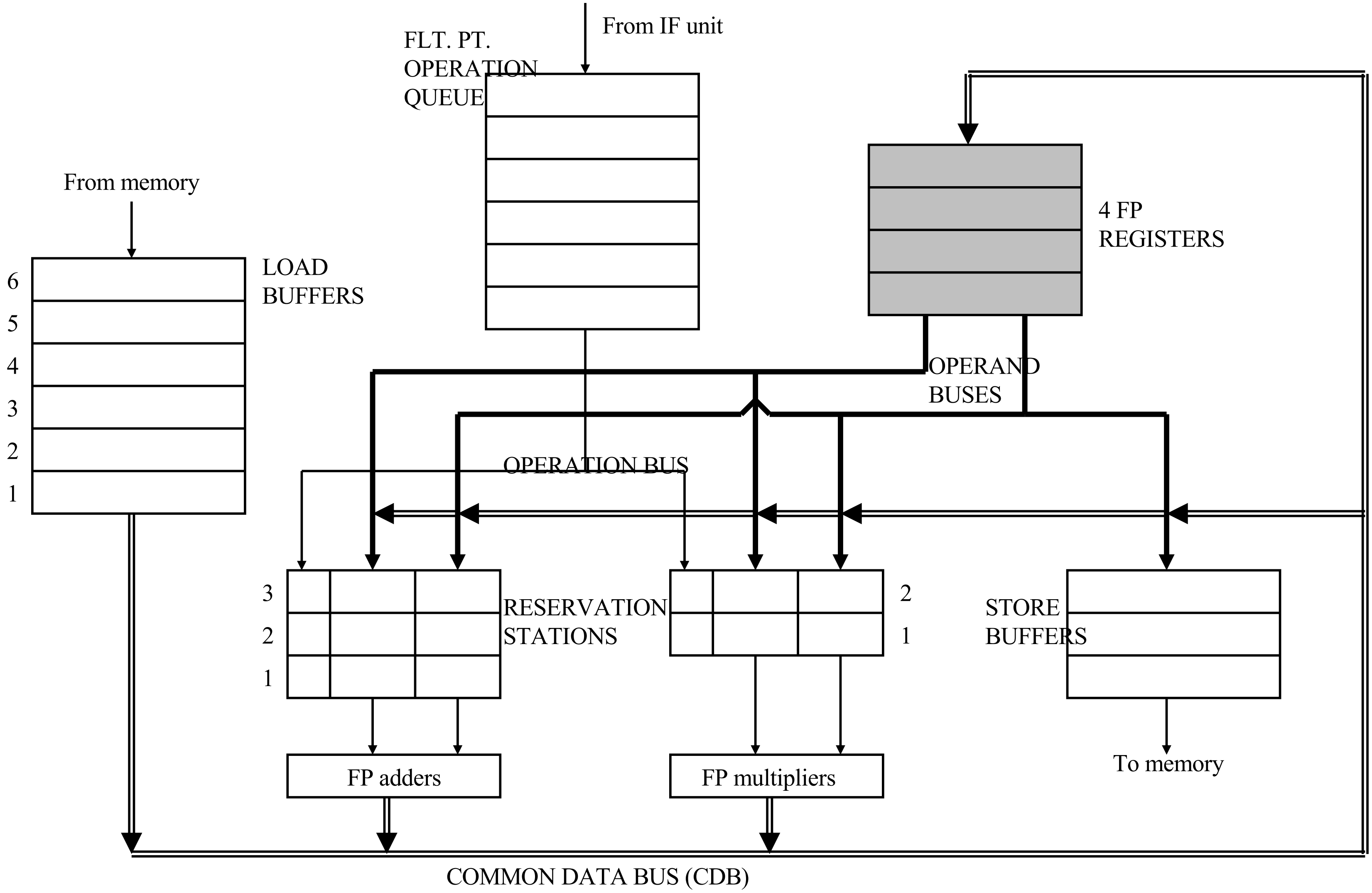
Key aspects of Tomasulo' s Alg.

- Read register operands at dispatch stage
  - If operands are available, the data is buffered along with the instruction in "reservation stations"
  - 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 condition
- 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

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 bypasses)
  - Functional units don't contend for register file ports

360/91 FP unit (DLX version)



Tomasulo's Alg. (DLX version)

- Four stages after 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 until operands become available 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 register file if the CDB tag matches the tag in the register file

Four stages (cont…)

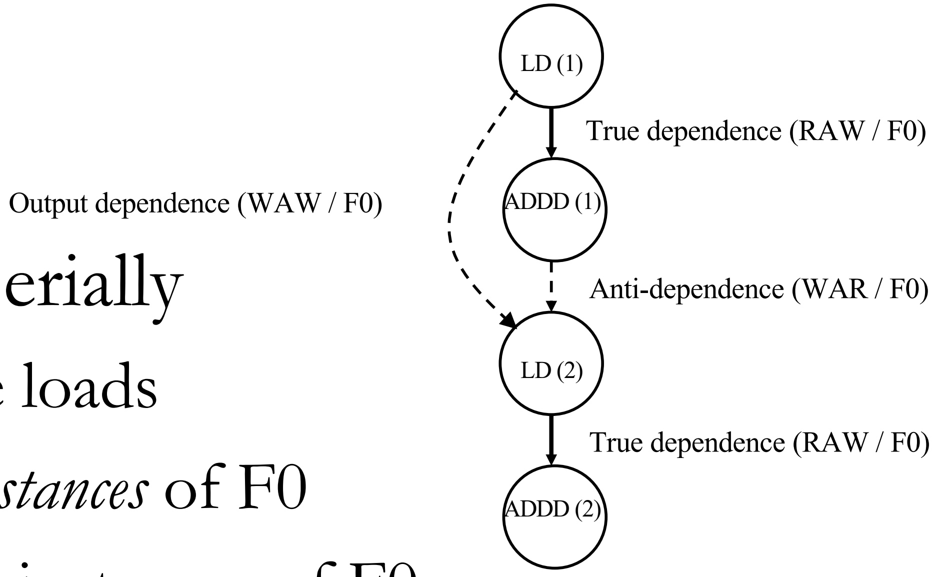
- Notice:
  - Same four stages as CDC, but read register file at dispatch stage and replace register specifiers with tags
  - No WAW / WAR checks (using tags eliminates these)
- Again, H&P uses different names than what we will use
  - Worse, H&P specifies only three stages for Tomasulo
  - But *any* dynamic scheduling inherently has dispatch, issue, execute, write result…

Register renaming

- Consider simple example with register reuse

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

- Dataflow graph with both true *and* false dependences


LD (1)
True dependence (RAW / F0)
ADDD (1)
Output dependence (WAW / F0)
Anti-dependence (WAR / F0)
LD (2)
True dependence (RAW / F0)
ADDD (2)

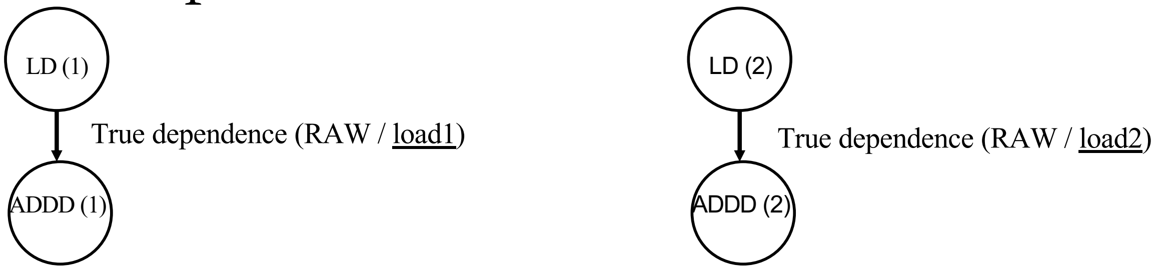
- All instructions execute serially
  - Due to reuse of F0 by the loads
  - But those are 2 distinct *instances* of F0
  - Use different names for 2 instances of F0

Register renaming

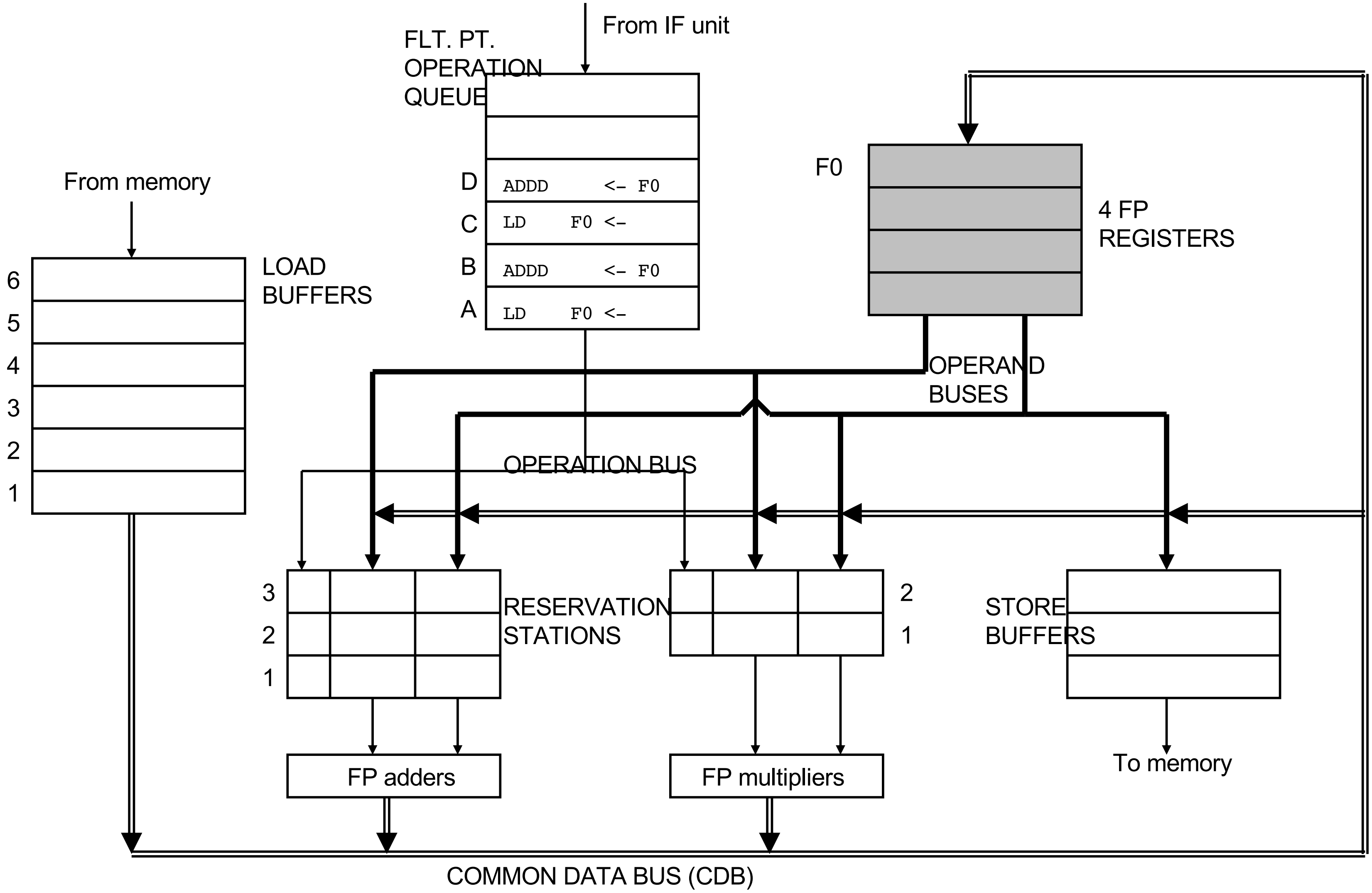
- Same program segment with F0 renamed
  - Tomasulo Alg: use reservation station number (tag) of producer instruction (e.g. load buffer 1 = load1)
    - This guarantees unique names for unique values

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

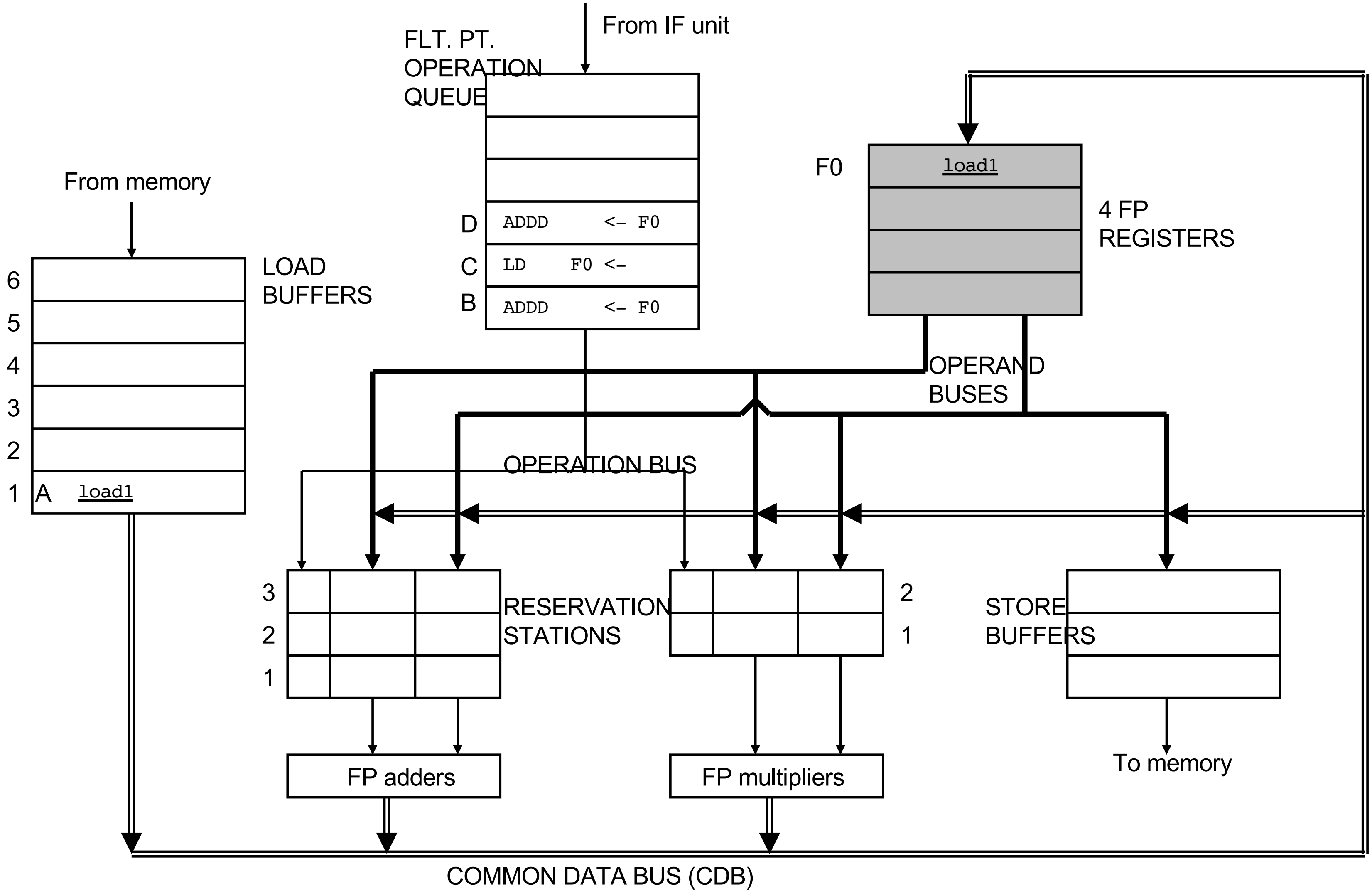
- Dataflow graph with only true dependences
  - Renaming removes output and anti-dependences
  - Parallelism is exposed



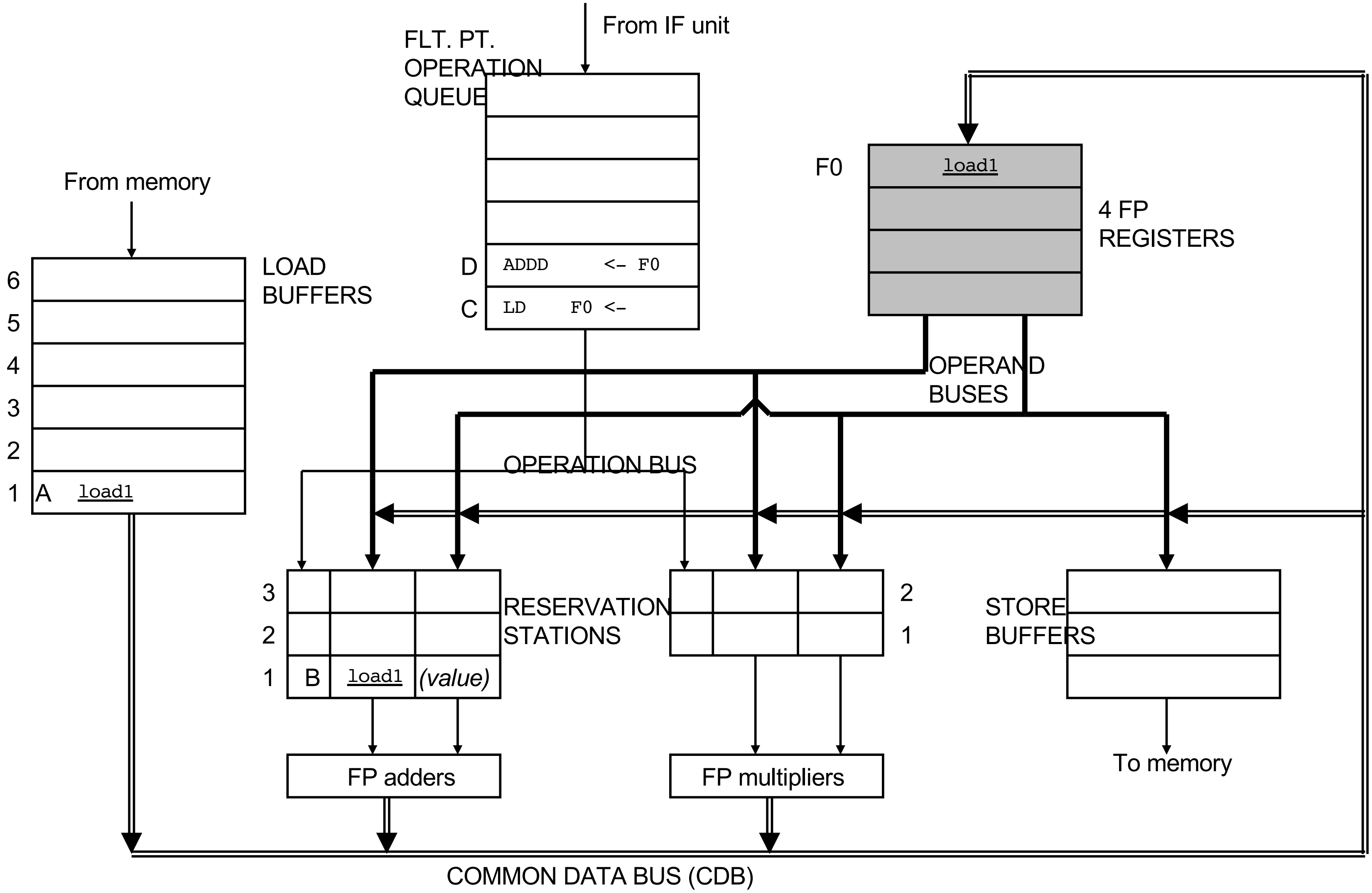
How (Tomasulo) renaming works (1)



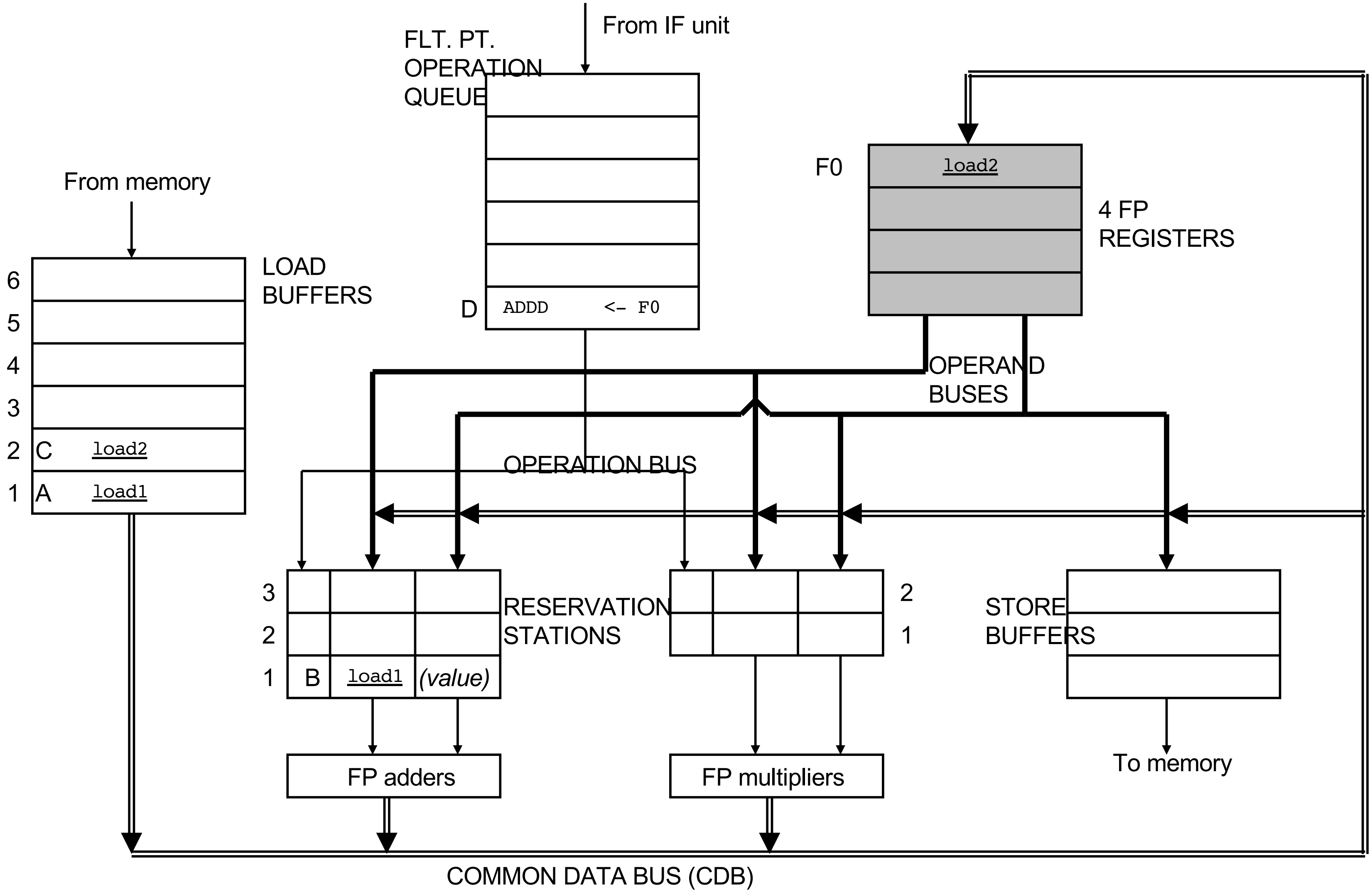
(2)


From IF unit
FLT. PT.
OPERATION
QUEUE
D
ADDD <– F0
C
LD F0 <–
B
ADDD <– F0
F0
load1
4 FP
REGISTERS
From memory
6
5
4
3
2
1
A
load1
LOAD
BUFFERS
OPERAND
BUSES
OPERATION BUS
3
2
1
RESERVATION
STATIONS
2
1
STORE
BUFFERS
FP adders
FP multipliers
To memory
COMMON DATA BUS (CDB)

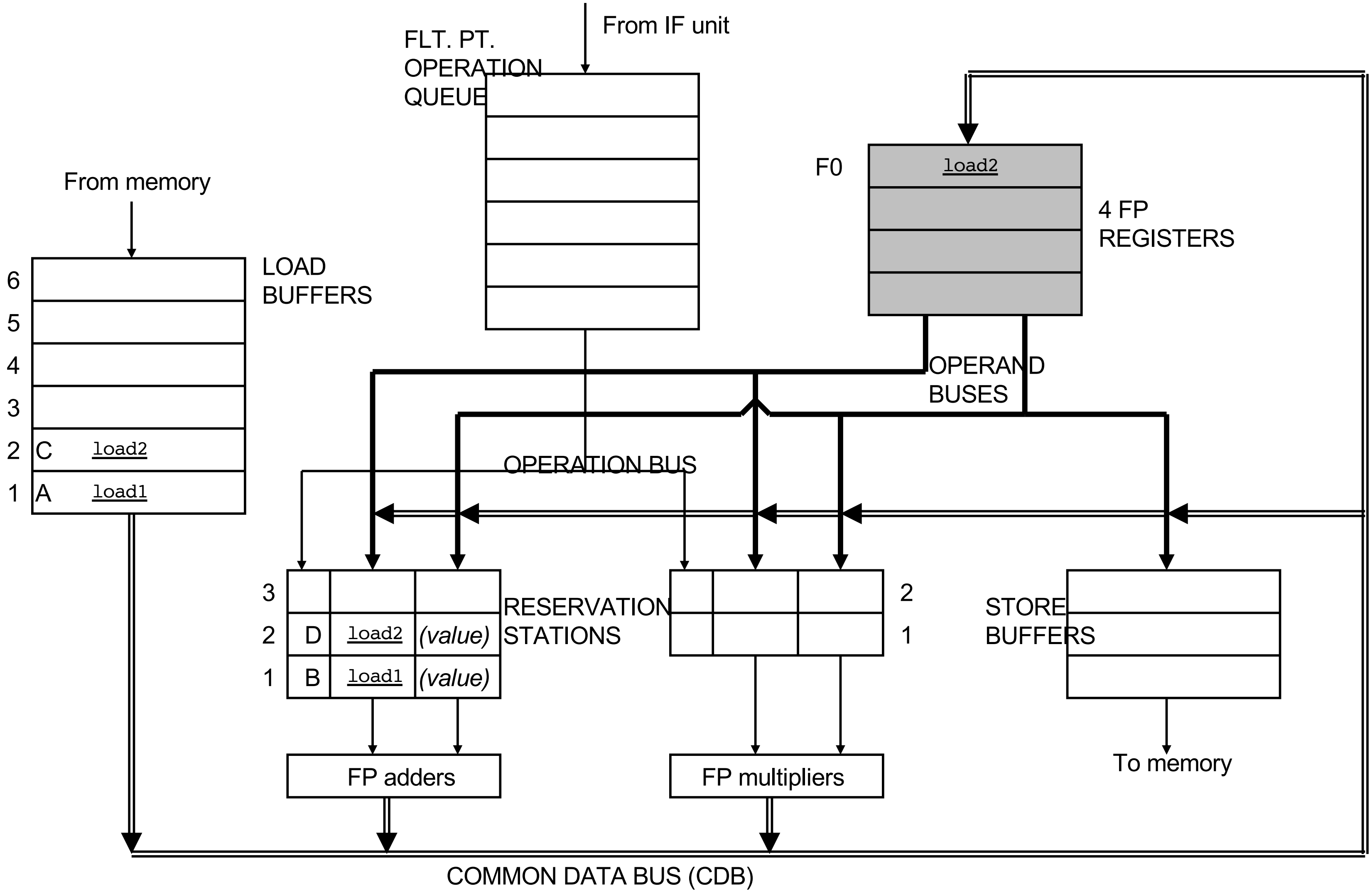
(3)


From IF unit
FLT. PT.
OPERATION
QUEUE
F0
load1
4 FP
REGISTERS
From memory
LOAD
BUFFERS
6
5
4
3
2
1
A
load1
D
ADDD <- F0
C
LD F0 <-
OPERAND
BUSES
OPERATION BUS
3
2
1
B
load1
(value)
RESERVATION
STATIONS
2
1
STORE
BUFFERS
FP adders
FP multipliers
To memory
COMMON DATA BUS (CDB)

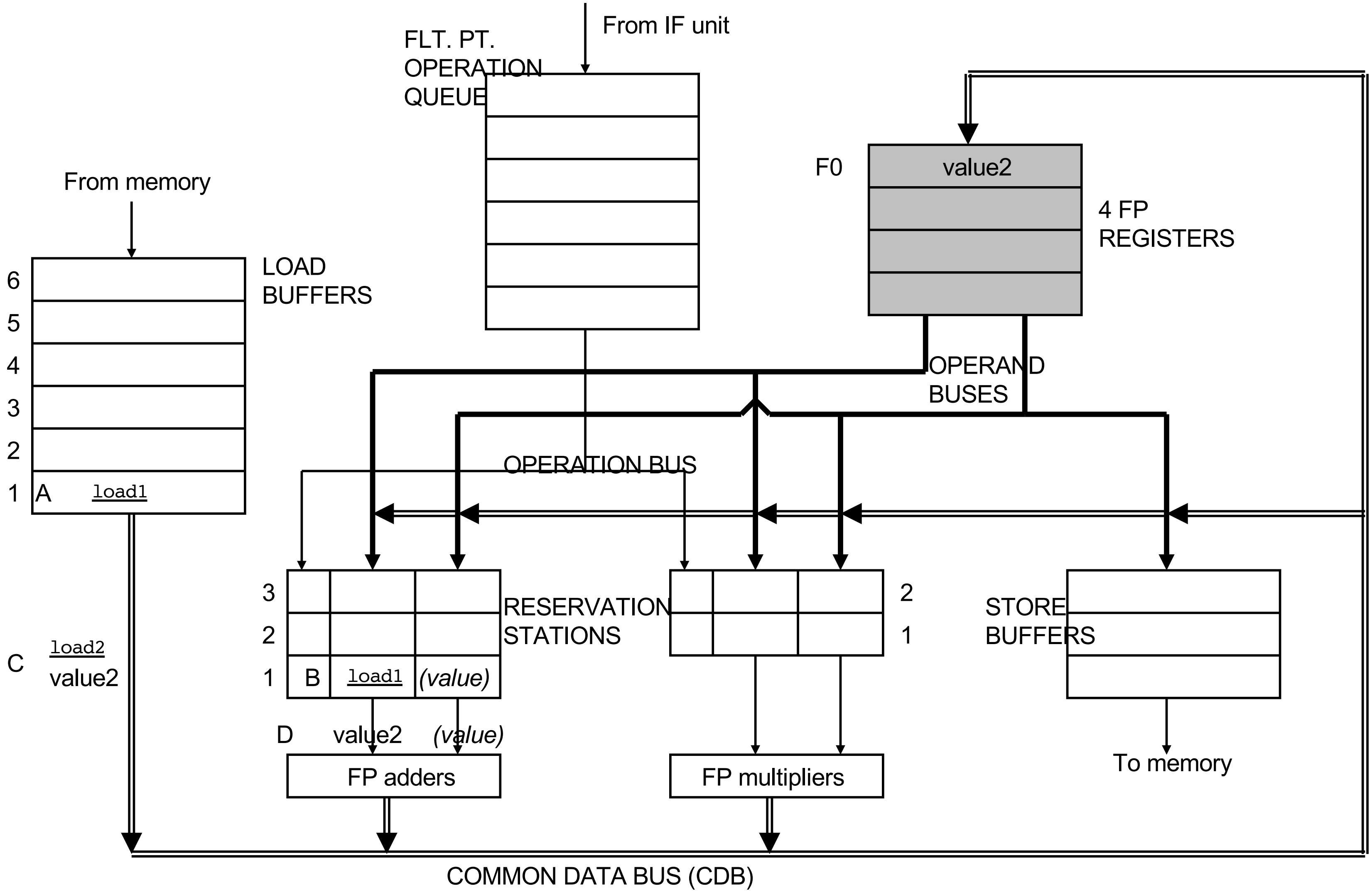
(4)


From IF unit
FLT. PT.
OPERATION
QUEUE
D
ADDD <- F0
F0
load2
4 FP
REGISTERS
From memory
LOAD
BUFFERS
6
5
4
3
2
C
load2
1
A
load1
OPERAND
BUSES
OPERATION BUS
3
2
1
B
load1
(value)
RESERVATION
STATIONS
2
1
STORE
BUFFERS
FP adders
FP multipliers
To memory
COMMON DATA BUS (CDB)

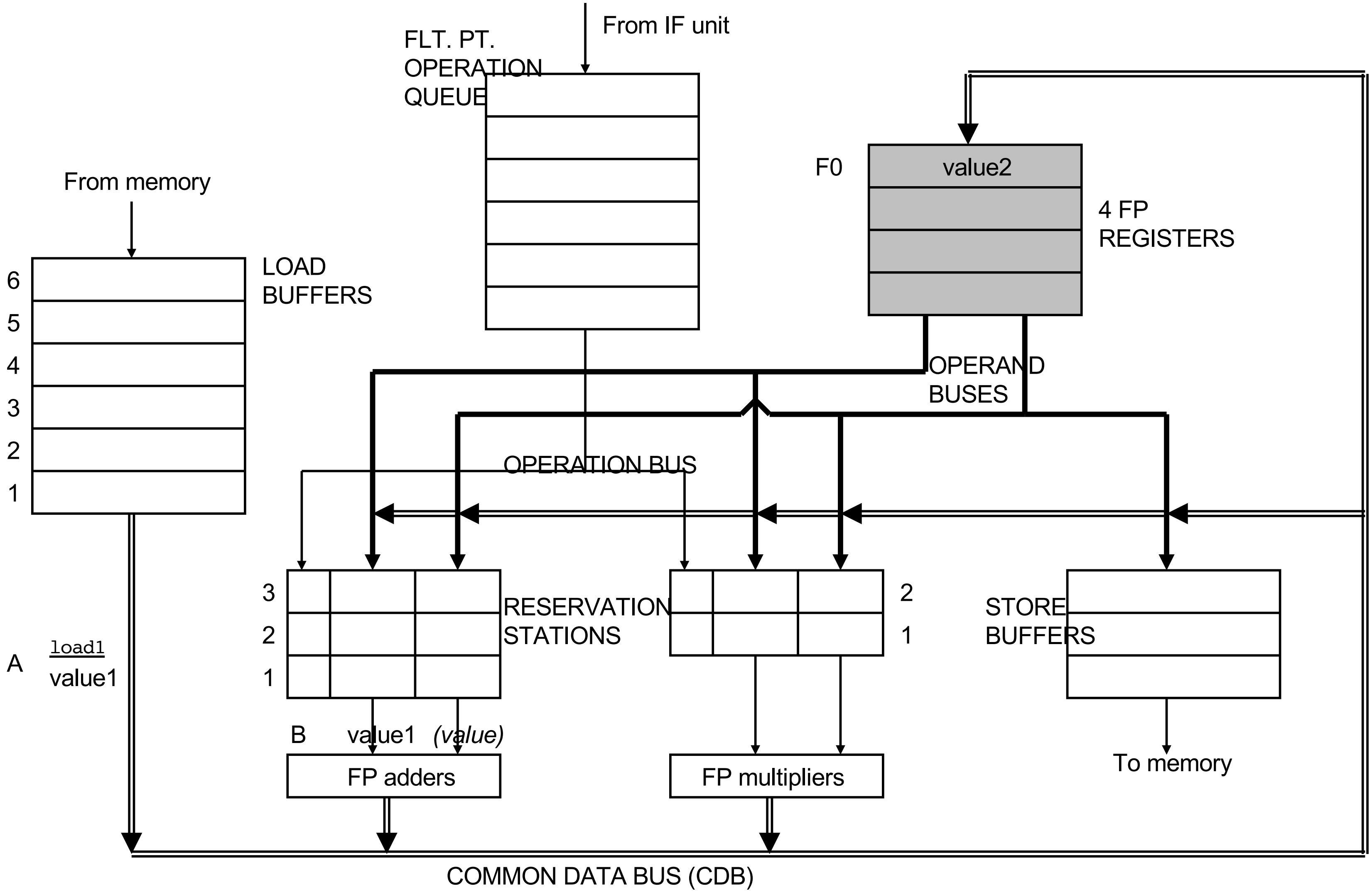
(5)


From IF unit
FLT. PT.
OPERATION
QUEUE
F0
load2
4 FP
REGISTERS
From memory
6
5
4
3
2 C load2
1 A load1
LOAD
BUFFERS
OPERAND
BUSES
OPERATION BUS
3
2 D load2 (value)
1 B load1 (value)
RESERVATION
STATIONS
2
1
STORE
BUFFERS
FP adders
FP multipliers
To memory
COMMON DATA BUS (CDB)

(6) Second load completes 1st



(7)


From IF unit
FLT. PT.
OPERATION
QUEUE
F0
value2
4 FP
REGISTERS
From memory
LOAD
BUFFERS
6
5
4
3
2
1
OPERAND
BUSES
OPERATION BUS
3
2
1
RESERVATION
STATIONS
2
1
STORE
BUFFERS
A
load1
value1
B
value1
(value)
FP adders
FP multipliers
To memory
COMMON DATA BUS (CDB)

Recall our on-going example

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 1	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 2	IS	EX	…..	…..							
ADDD F6 , F8, F2						IF	ID	IS	EX	EX	WB															
CDB:						F6	F2		F8		F6’						F0									

Shaded boxes indicate stalls.

▨ RAW   1. LD-MULTD (F2)   2. MULTD-DIVD (F0)

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

Module 6.3: Precise Interrupt and Reorder Buffer

Objectives and Outline

- Objectives:
  - Understand the problem of precise interrupts
  - Understand reorder buffer as a solution to the precise interrupt problem
- Outline:
  - Types of interrupts
  - Correct handling of interrupts and why OOO execution may affect it
  - Various approaches including reorder buffer (ROB)
  - Tomasulo algorithm with ROB: mechanism and illustration
  - Handling branch misprediction using precise interrupt mechanism

Precise interrupts

- Out-of-order execution and interrupts don't mix!
  - When an interrupt occurs, O/S wants precise state
  - Precise state means:
    - All instructions *before* the instruction causing the interrupt have completed
    - All instructions *after* have not completed
  - Why are precise interrupts important?
    1. 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!
    2. Programmer debugging – user presumes sequential program
  - Why is OOO-execution bad for precise interrupts?
    - Instructions *after* interrupt may have updated state!

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: function of program and memory state (e.g., arithmetic overfow, page fault)
  - 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 / Nonmaskable
  - User maskable: can be (temporarily) ignored (e.g., arithmetic overflow, breakpoint)
  - Nonmaskable: must be handled (e.g., page fault, power fail)

Classifying Interrupts (cont.)

- 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

- Precise interrupts (Sequential Semantics)
  - Complete instructions before offending one
  - Squash (effects of) instructions after
  - Save PC
  - Force trap instruction into IF
- Must handle simultaneous/OOO interrupts
  - IF – memory problems (page fault, misaligned reference, protection violation)
  - ID – illegal or privileged instruction
  - EX – arithmetic exception
  - MEM – memory problems (see IF above)
  - WB – none
  - Which interrupt should be handled first?

Handling interrupts (cont.)

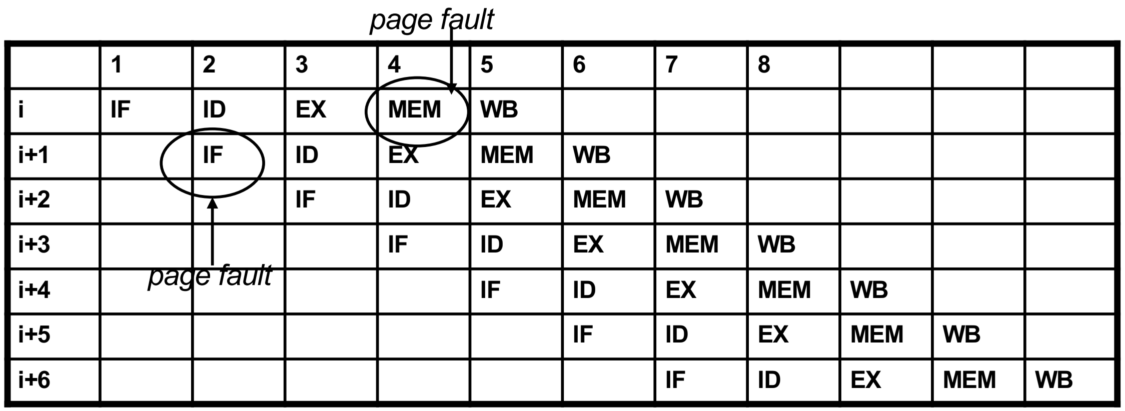
- Example: data page fault

	1	2	3	4	5	6	7	8			
i	IF	ID	EX	MEM	WB	*page fault*					
i+1		IF	ID	EX	MEM	WB					
i+2			IF	ID	EX	MEM	WB				*squash*
i+3				IF	ID	EX	MEM	WB			
i+4					IF	ID	EX	MEM	WB		
i+5				*trap*		IF	ID	EX	MEM	WB	
i+6				*trap handler*			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 (cont.)

- Out-of-order interrupts
  - Example: which page fault should we take?



page fault

	1	2	3	4	5	6	7	8			
i	IF	ID	EX	MEM	WB						
i+1		IF	ID	EX	MEM	WB					
i+2			IF	ID	EX	MEM	WB				
i+3				IF	ID	EX	MEM	WB			
i+4					IF	ID	EX	MEM	WB		
i+5						IF	ID	EX	MEM	WB	
i+6							IF	ID	EX	MEM	WB

page fault

- Answer
  - Even though ‘i+1’ faults first, must service ‘i’ fault first
  - Solution: post interrupts
    - Check interrupt bit on entering WB
    - This maintains precise interrupts

OOO-execution and interrupts

- 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			
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

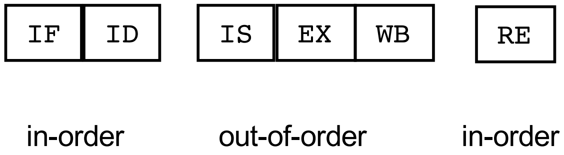
- Imprecise interrupts
  - Ignore the problem
  - Makes page faults difficult
  - IEEE Standard strongly suggests precise interrupts
- In-order completion
  - Stall pipeline when necessary
- Software clean-up
  - Save information for trap handlers (pipeline state)
  - Machine dependent
- Hardware clean-up
  - Restore consistent state
- Re-order instructions
  - Complete out-of-order
  - *Retire* in-order

Can you tell which one is preferred?

Reorder Buffer (ROB)

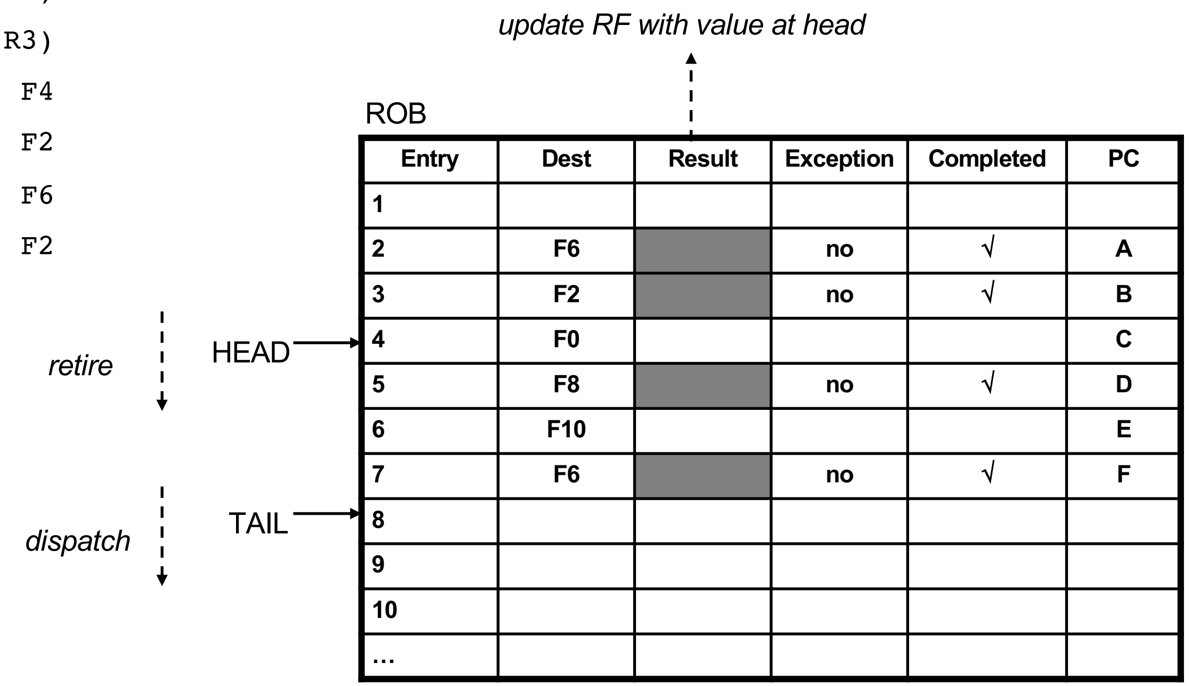
- Operation of ROB
  - ROB is a FIFO with head and tail pointer
  - Instruction dispatch
    - Reserve ROB entry at tail, advance tail ptr
    - Record ROB entry (ROB tag) with instruction
  - Instruction execution/completion
    - Write value into ROB entry specified by the instruction's tag
    - DO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

New stage: Retire (RE)



Reorder Buffer (cont.)

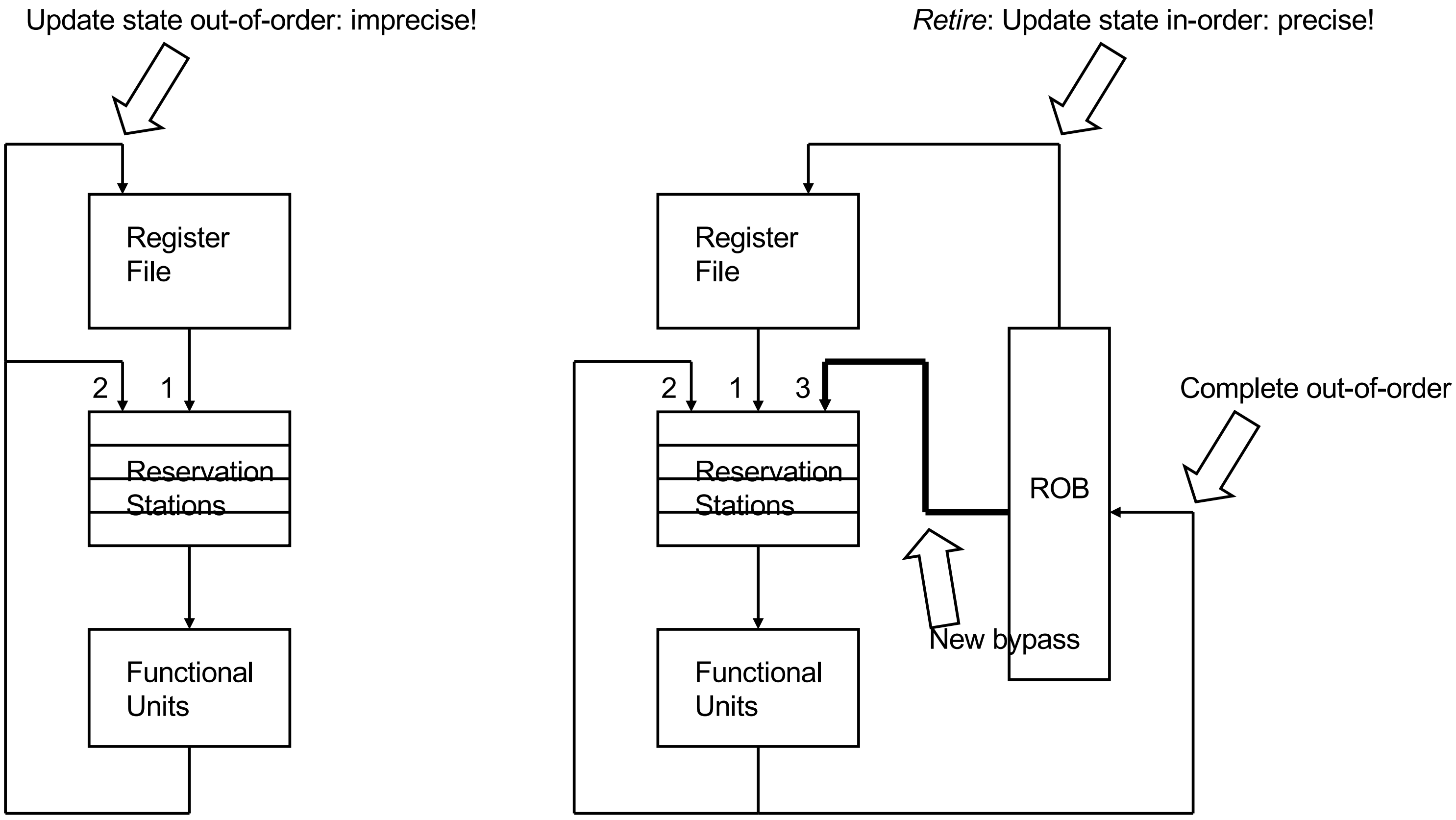
```
A   LD    F6 , 34(R2)
B   LD    F2 , 45(R3)
C   MULTD F0 , F2, F4
D   SUBD  F8 , F6, F2
E   DIVD  F10, F0, F6
F   ADDD  F6 , F8, F2
```



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

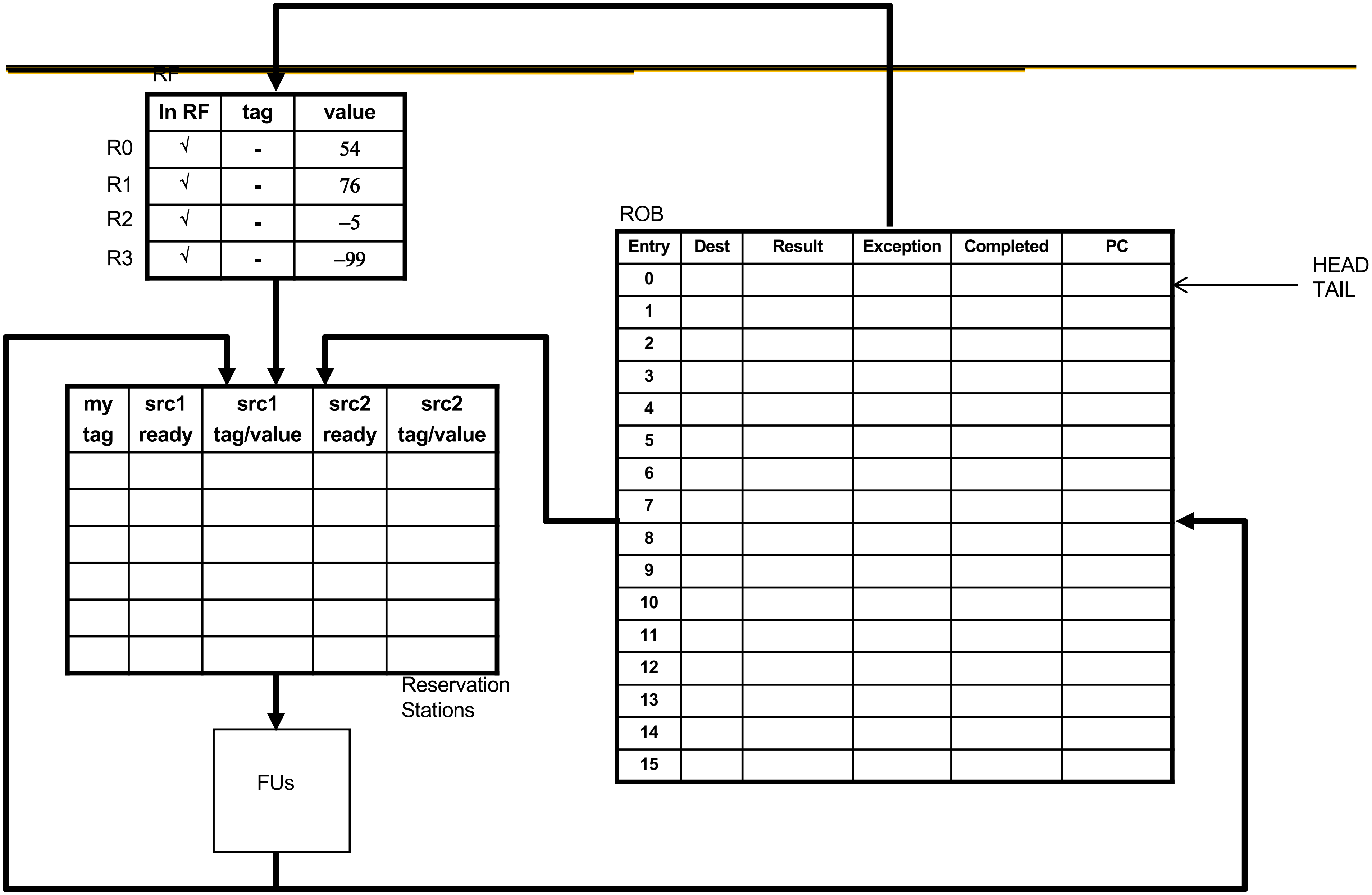
Shown: The two loads (A and B) have completed and retired.
The MULTD has not completed.
The ADDD and SUBD have completed out-of-order, but the ROB prevents them from retiring (updating state).
THUS: *if the MULTD posts an exception, the Register File state is precise.*

Precise Tomasulo Pipeline with ROB



ROB requires bypass

- Scenario
  - Producer instruction completed – value in ROB
  - Producer instruction not retired – value *not* in register file
  - Then, dependent instruction is dispatched into window
    - 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)
    - This new bypass is labeled (3) in previous diagram



RF

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

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

Reservation Stations

ROB

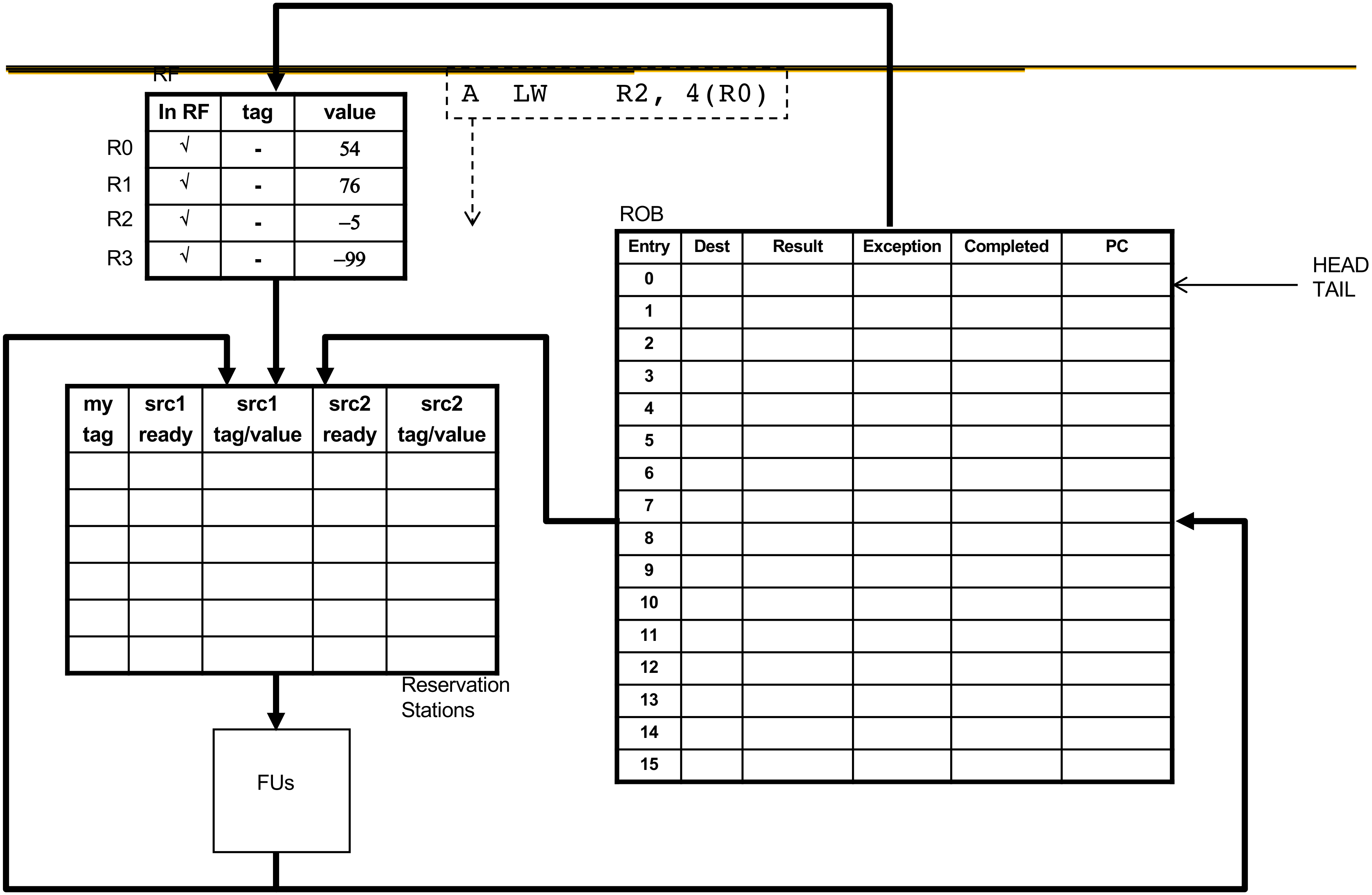
Entry	Dest	Result	Exception	Completed	PC
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

Example

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



F   ADD    R0, R1, R2
```



```
A   LW     R2, 4(R0)
```

RF

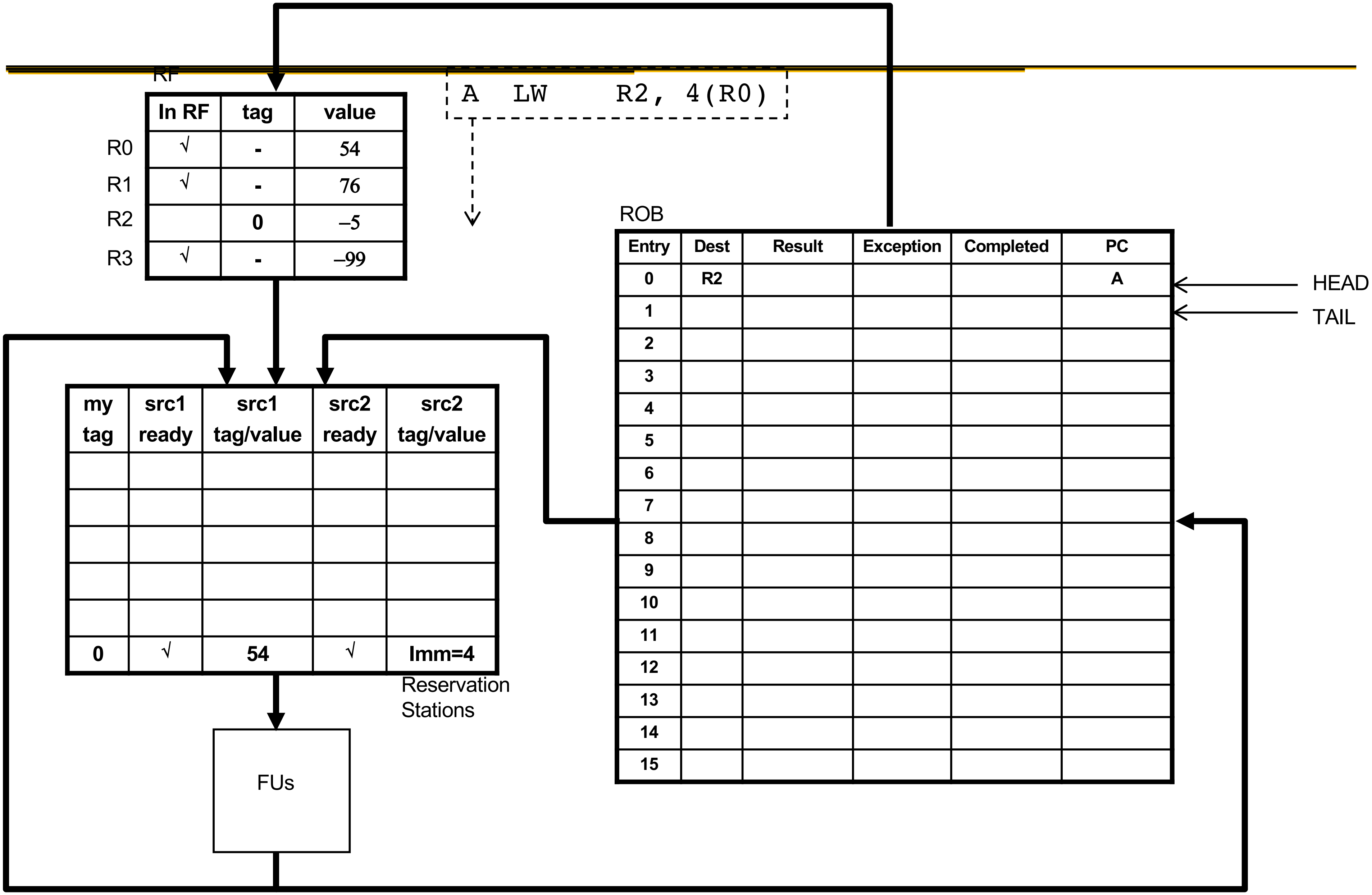
	In RF	tag	value
R0	√	-	54
R1	√	-	76
R2	√	-	–5
R3	√	-	–99

Reservation Stations

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

ROB

Entry	Dest	Result	Exception	Completed	PC
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



```
A   LW     R2, 4(R0)
```

RF

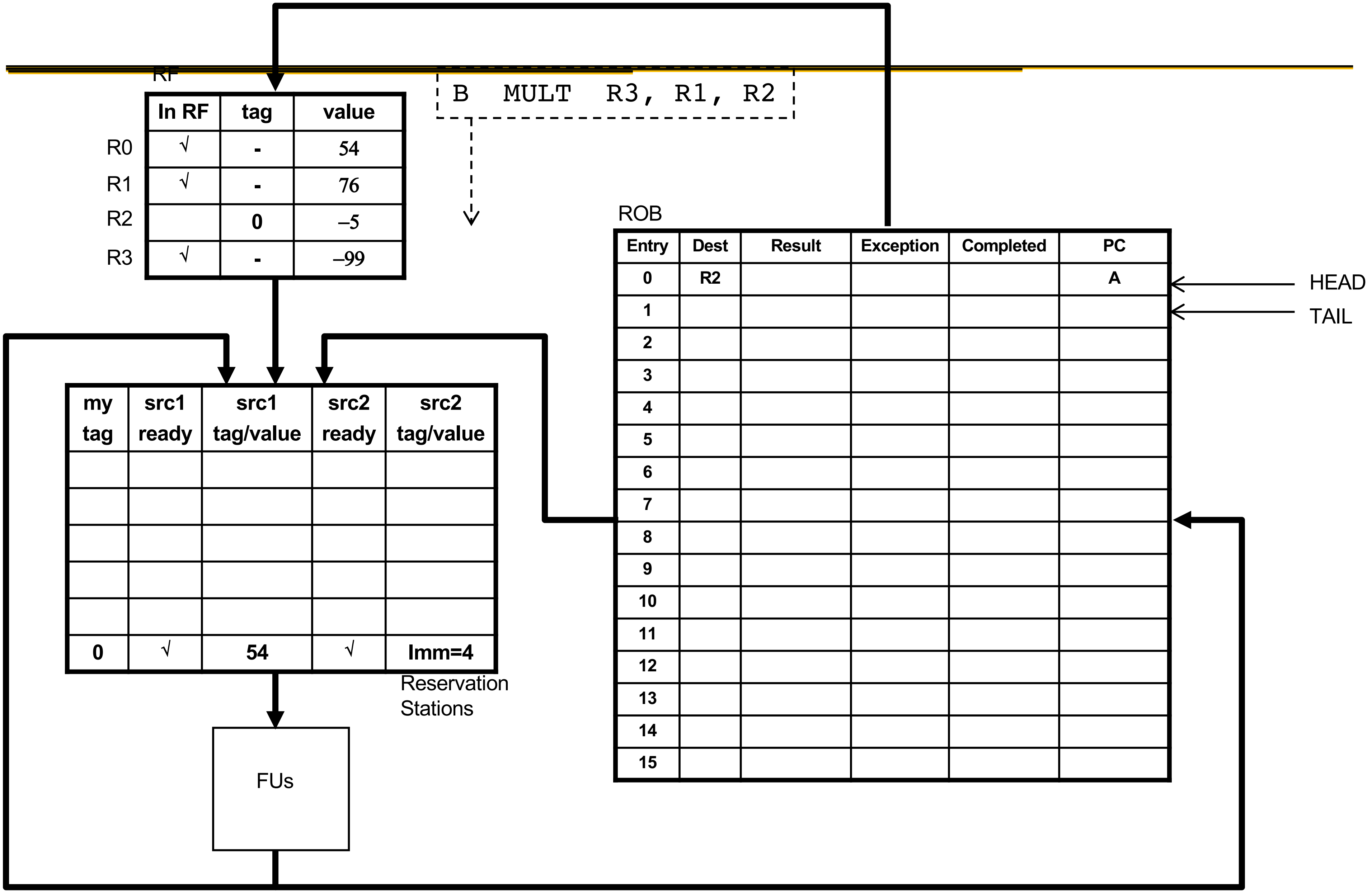
	In RF	tag	value
R0	√	-	54
R1	√	-	76
R2		0	–5
R3	√	-	–99

Reservation Stations

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
0	√	54	√	Imm=4

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

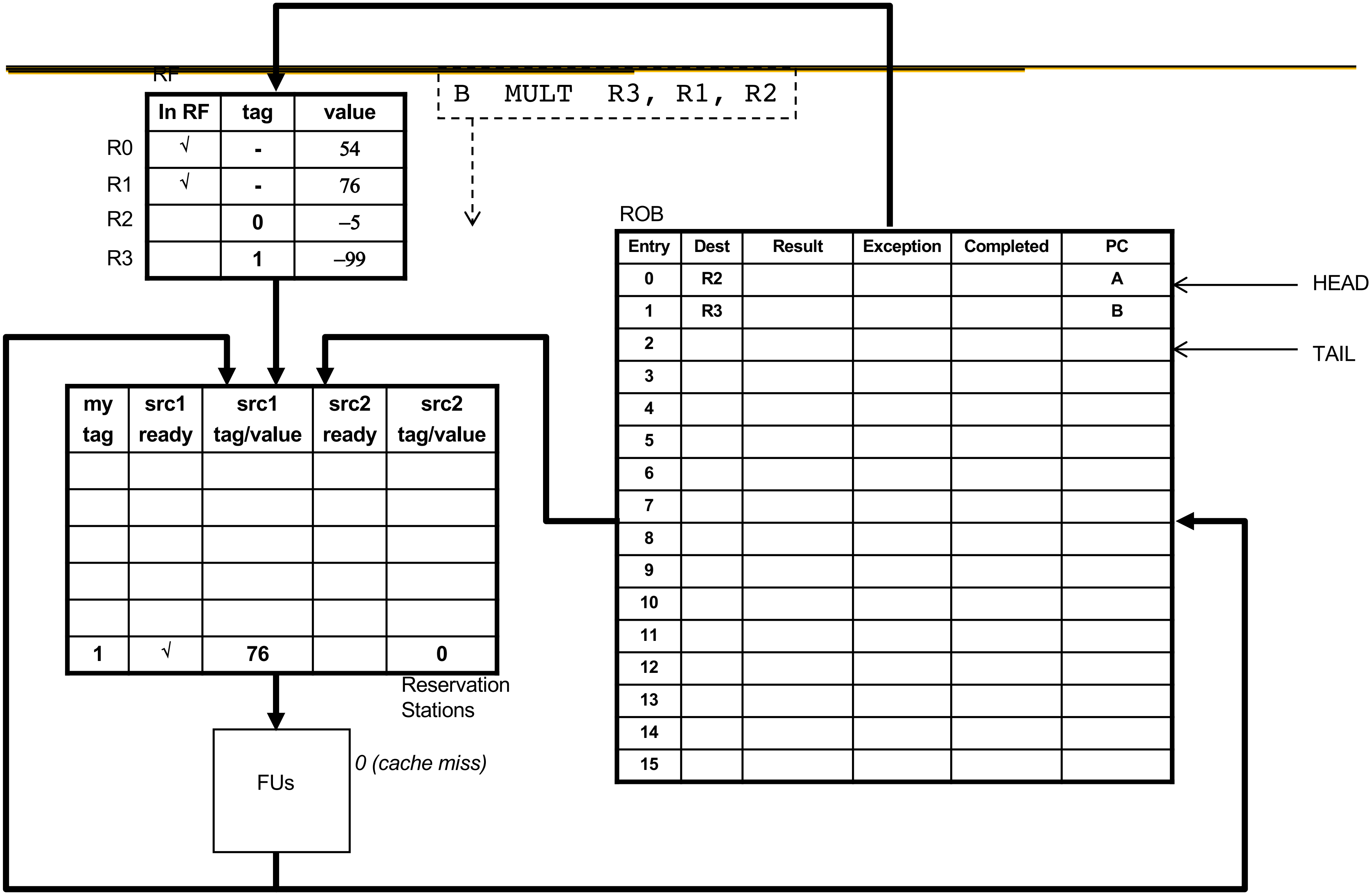


B MULT R3, R1, R2

	In RF	tag	value
R0	√	-	54
R1	√	-	76
R2		0	–5
R3	√	-	–99

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
0	√	54	√	Imm=4

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



RF

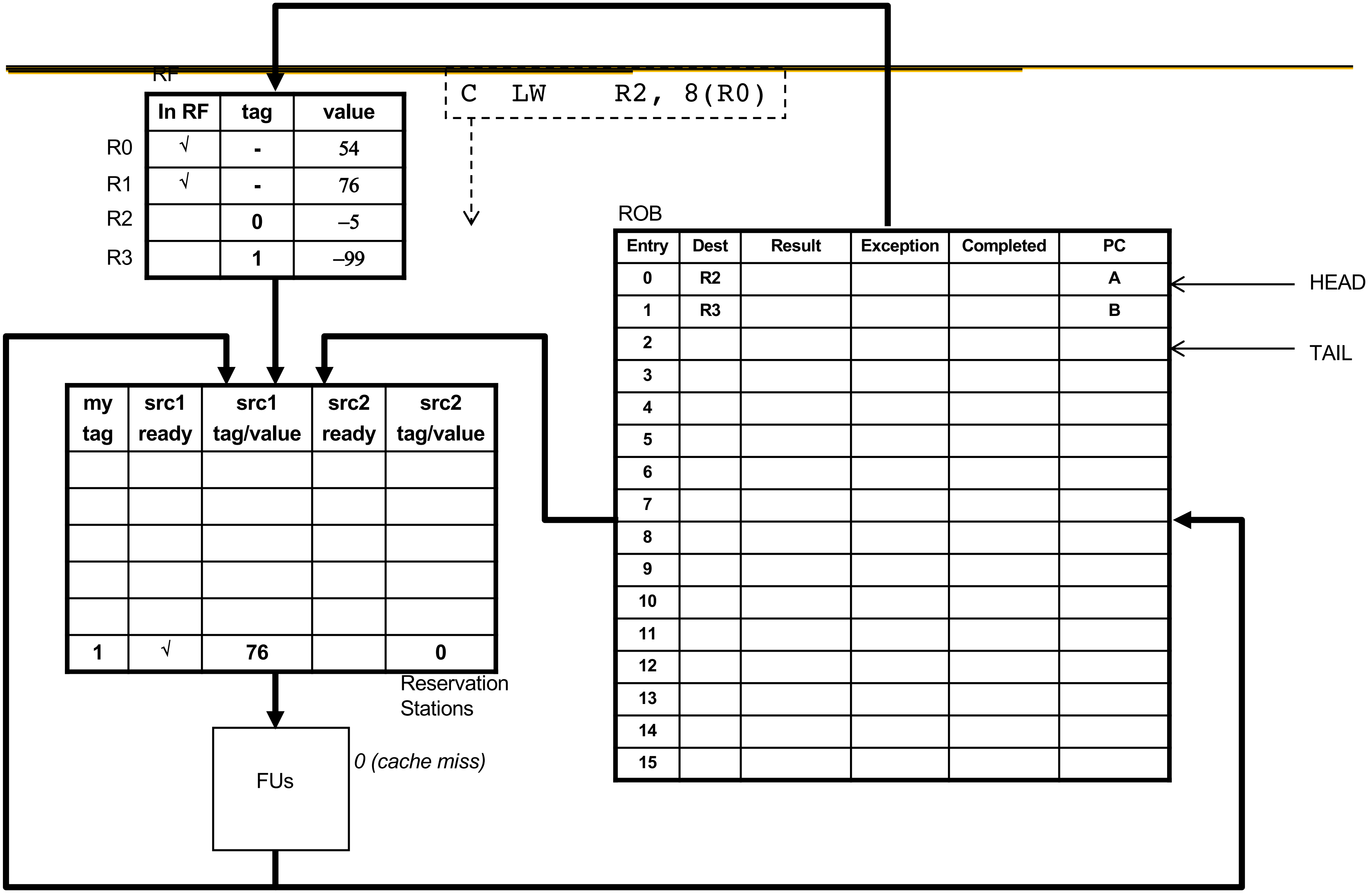
	In RF	tag	value
R0	√	-	54
R1	√	-	76
R2		0	–5
R3		1	–99

Reservation Stations

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
1	√	76		0

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

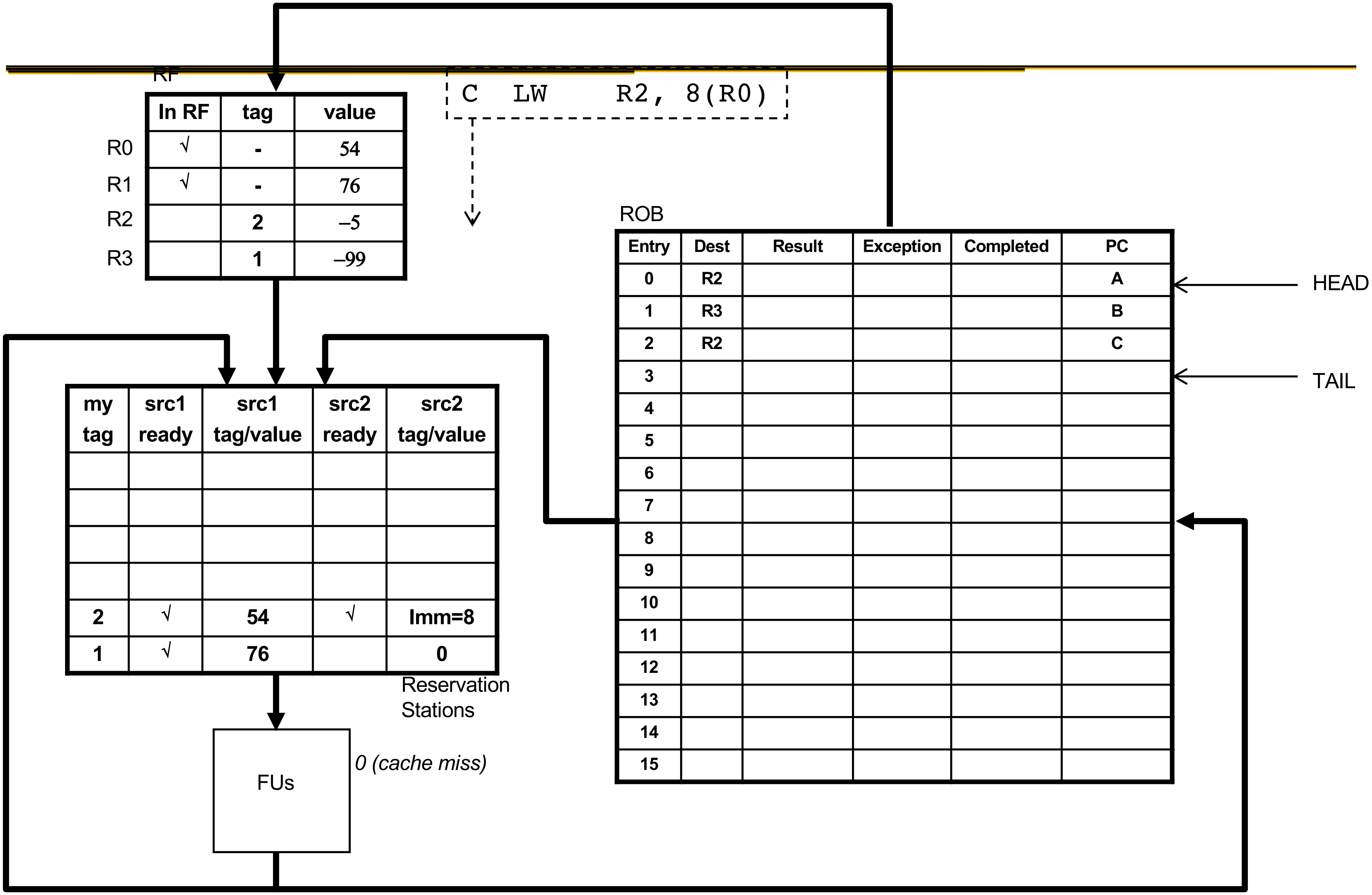


```
C   LW     R2, 8(R0)
```

	In RF	tag	value
R0	√	-	54
R1	√	-	76
R2		0	–5
R3		1	–99

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

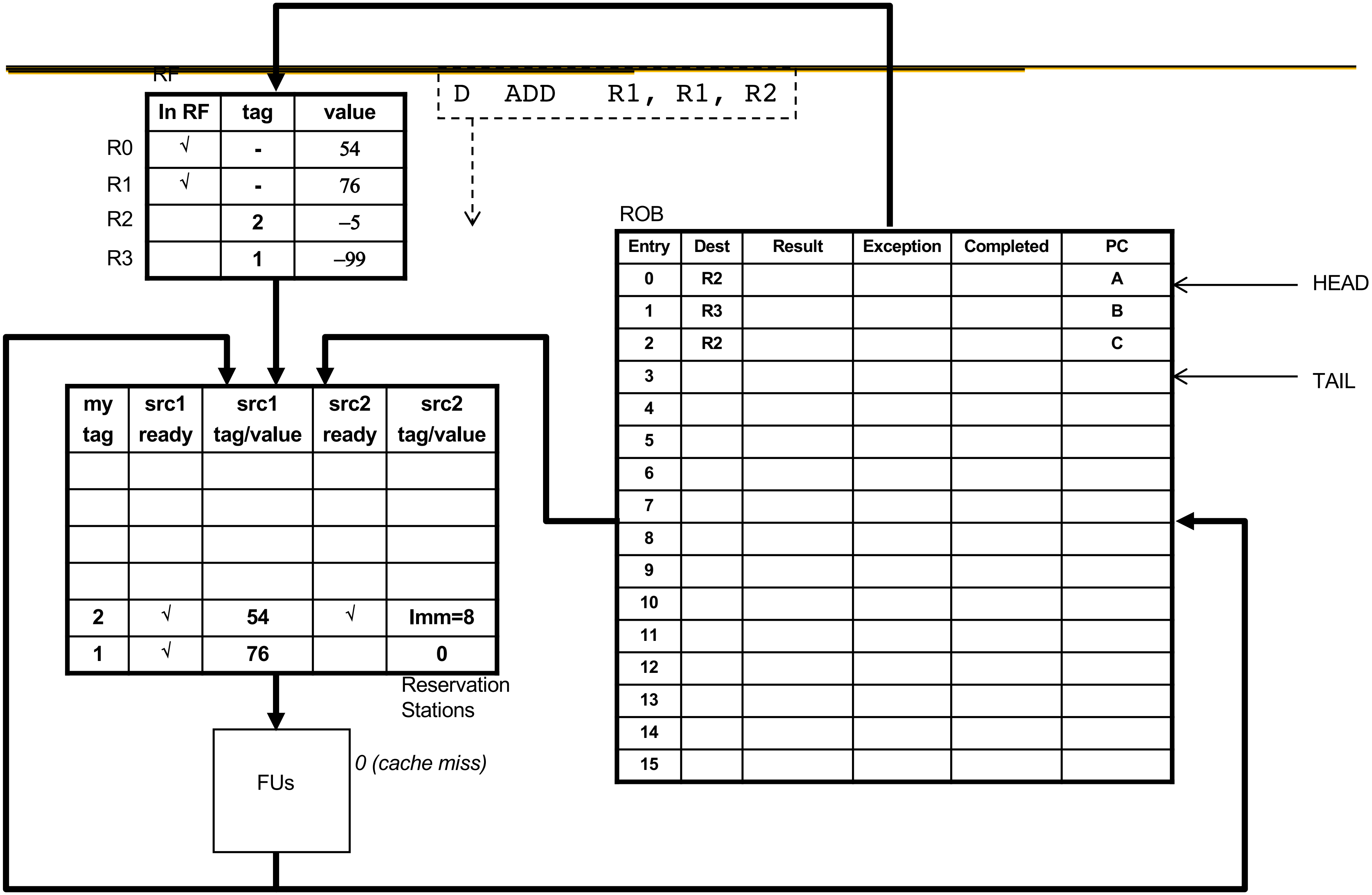
my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
1	√	76		0



	In RF	tag	value
R0	√	-	54
R1	√	-	76
R2		2	–5
R3		1	–99

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
2	√	54	√	Imm=8
1	√	76		0

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2				C
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



RF

D ADD R1, R1, R2

	In RF	tag	value
R0	√	-	54
R1	√	-	76
R2		2	–5
R3		1	–99

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
2	√	54	√	Imm=8
1	√	76		0

Reservation Stations

FUs

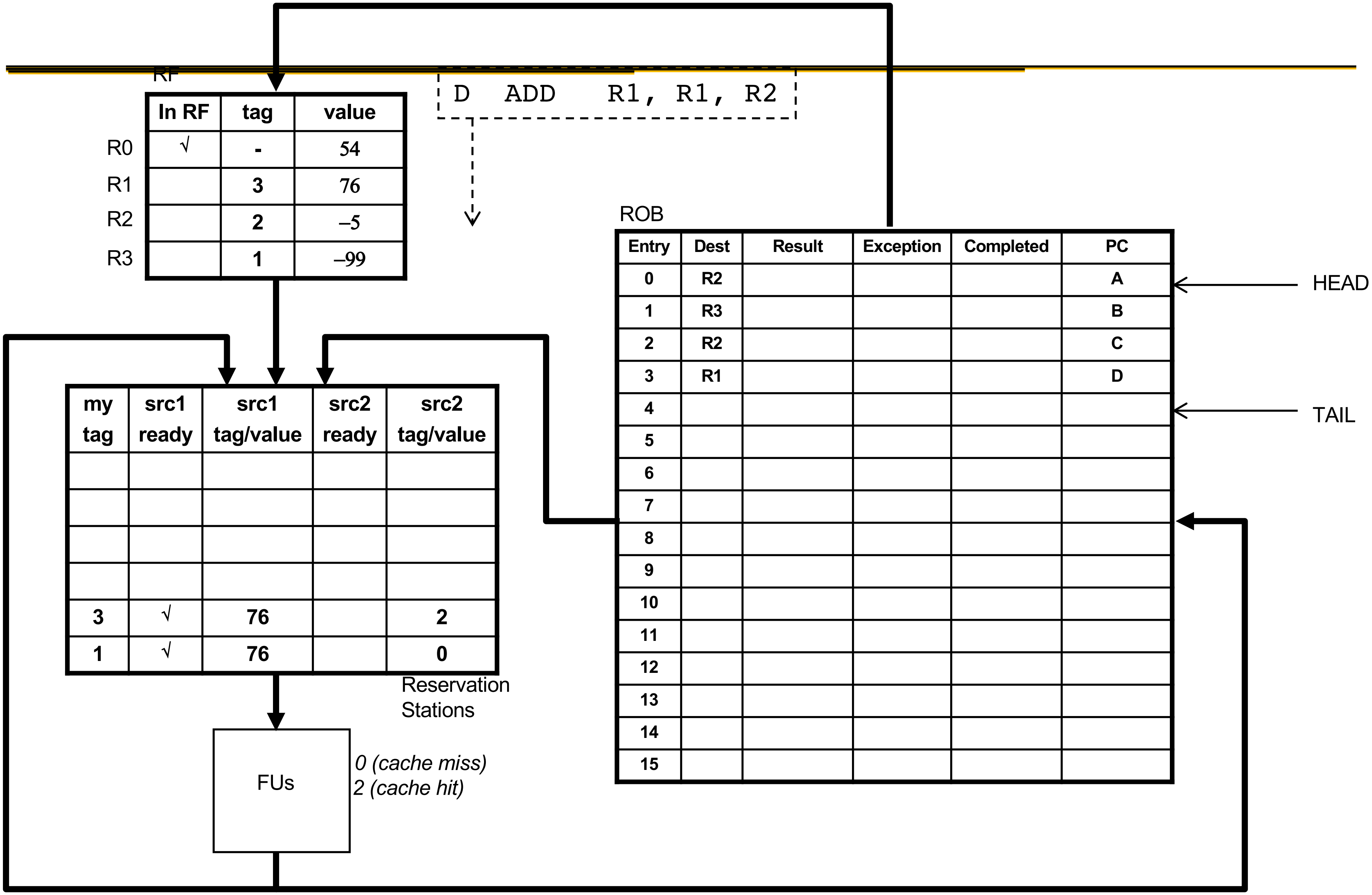
0 (cache miss)

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2				C
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

HEAD

TAIL



RF

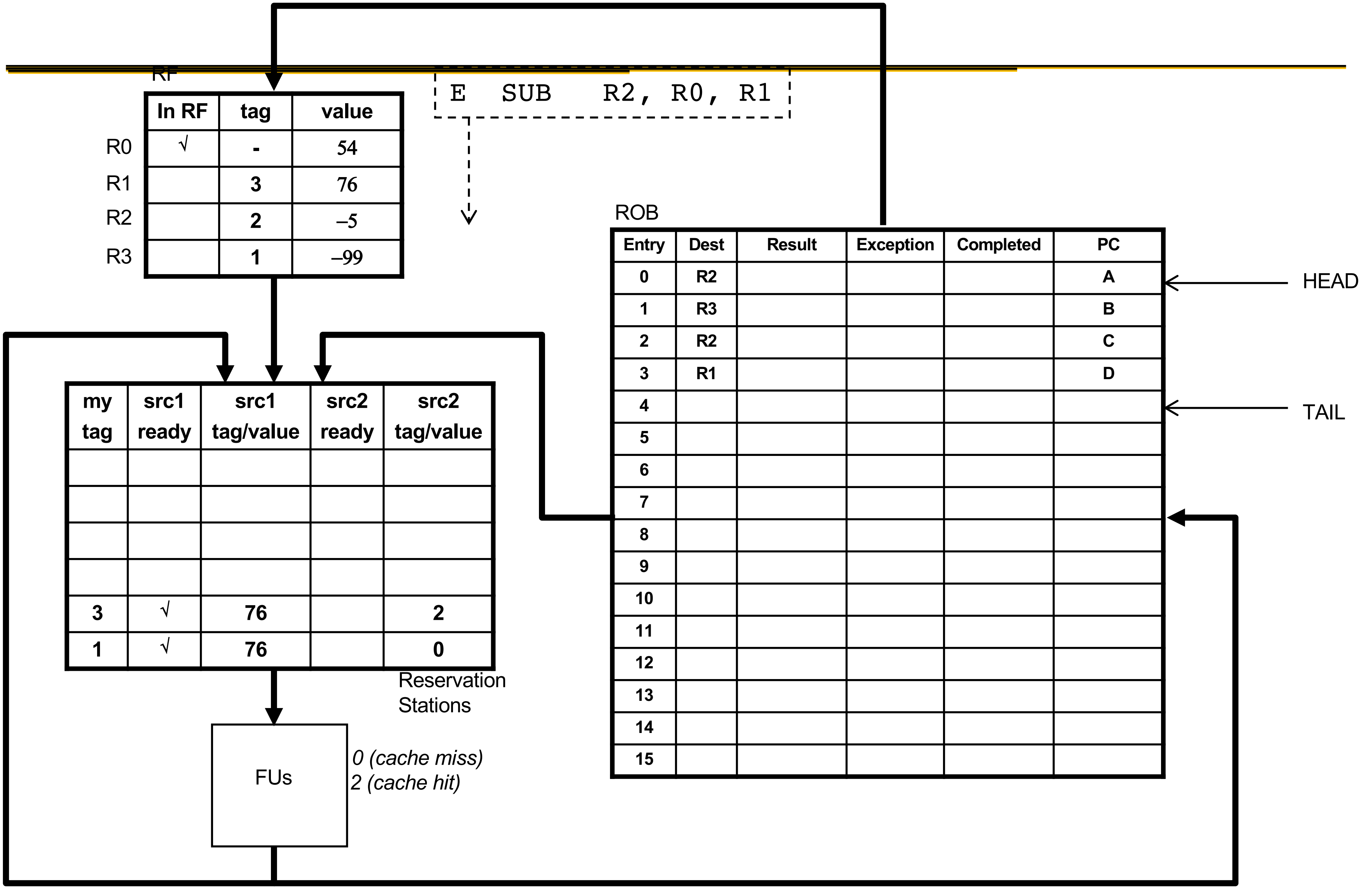
	In RF	tag	value
R0	√	-	54
R1		3	76
R2		2	–5
R3		1	–99

Reservation Stations

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
3	√	76		2
1	√	76		0

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2				C
3	R1				D
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



```
E   SUB    R2, R0, R1
```

RF

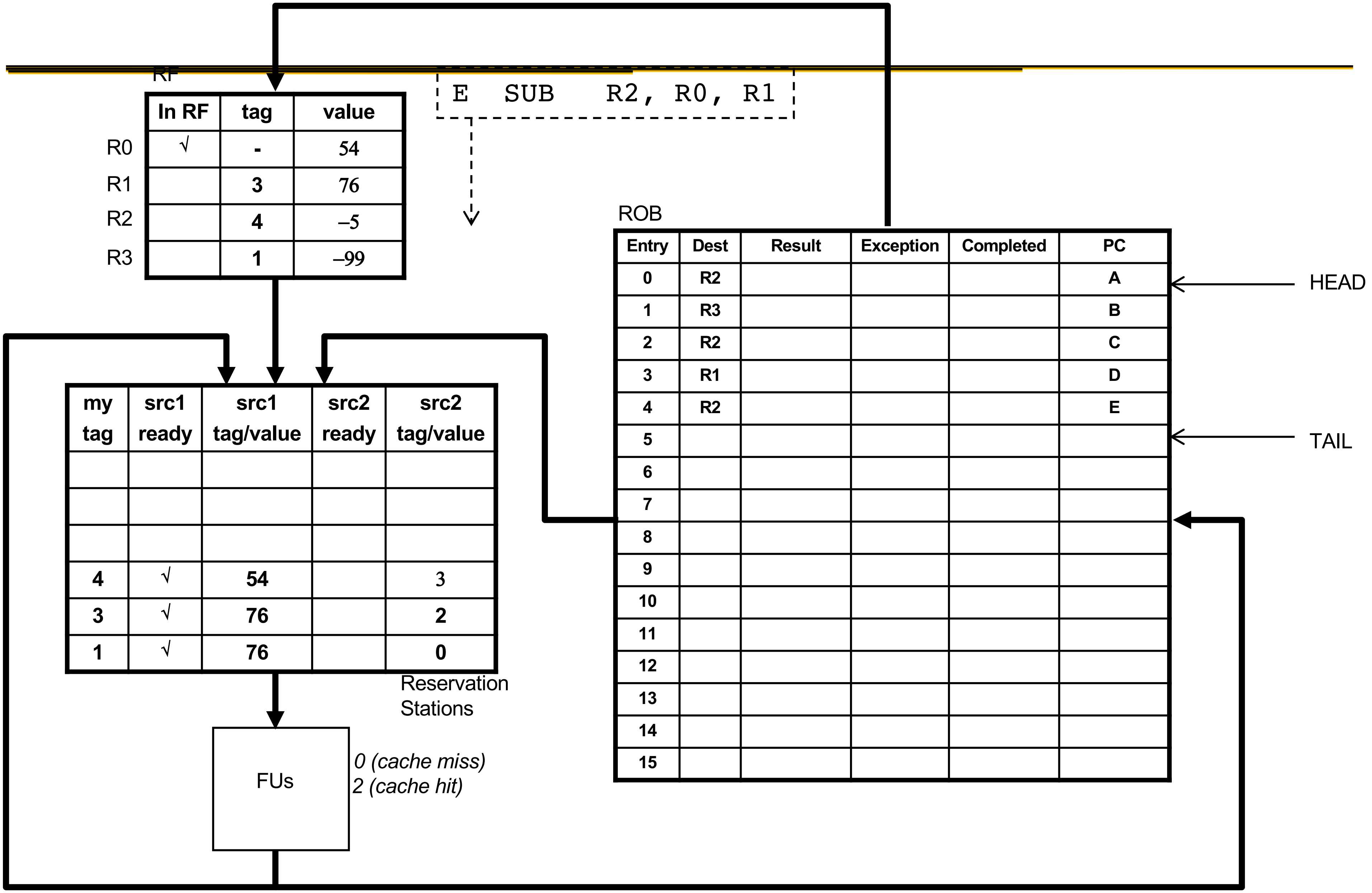
	In RF	tag	value
R0	√	-	54
R1		3	76
R2		2	–5
R3		1	–99

Reservation Stations

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
3	√	76		2
1	√	76		0

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2				C
3	R1				D
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



RF

	In RF	tag	value
R0	√	-	54
R1		3	76
R2		4	–5
R3		1	–99

```
E   SUB   R2, R0, R1
```

ROB

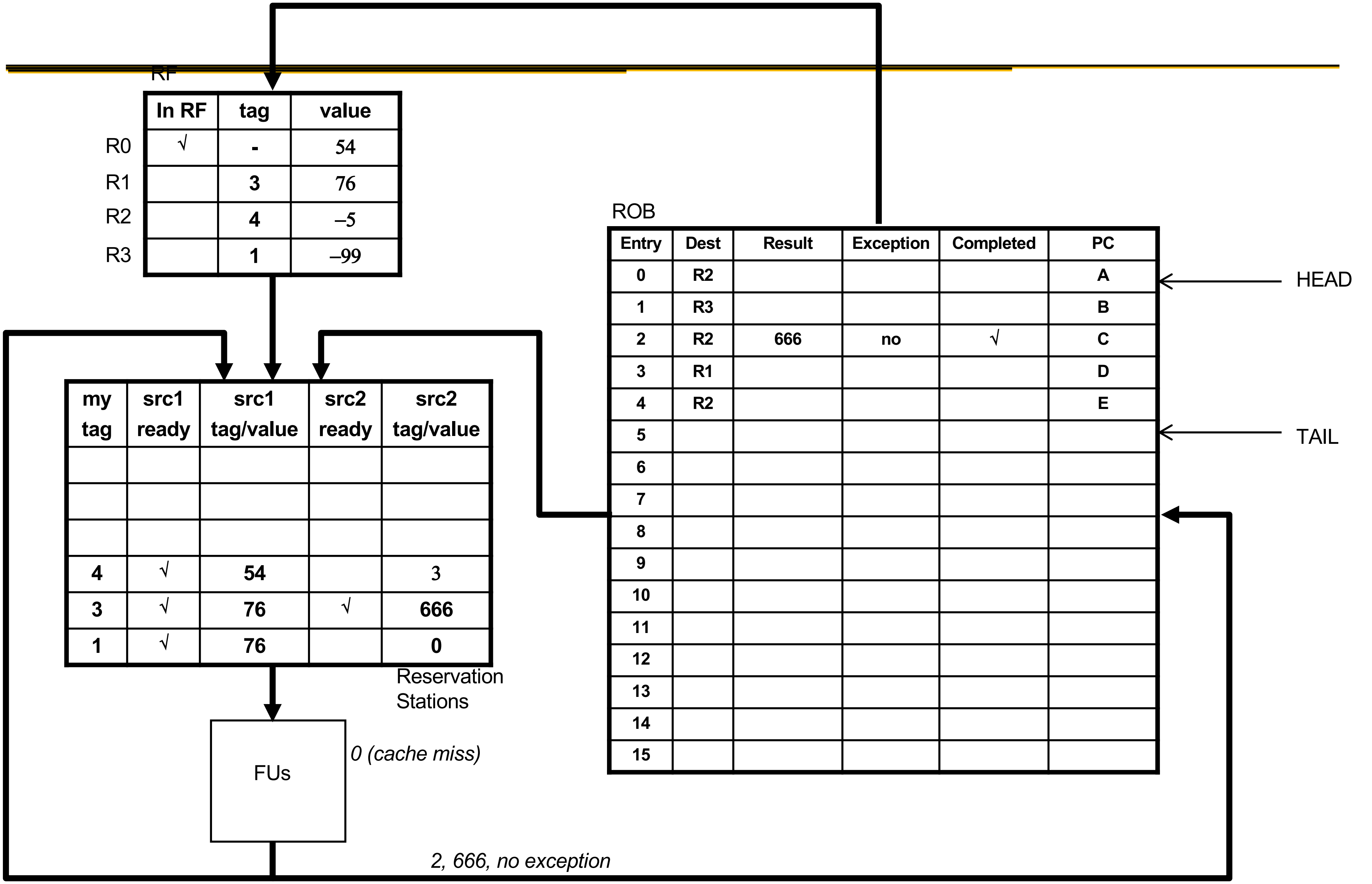
Entry	Dest	Result	Exception	Completed	PC	
0	R2				A	HEAD
1	R3				B	
2	R2				C	
3	R1				D	
4	R2				E	
5						TAIL
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
4	√	54		3
3	√	76		2
1	√	76		0

Reservation Stations

FUs

0 (cache miss)
2 (cache hit)



RF

	In RF	tag	value
R0	√	-	54
R1		3	76
R2		4	–5
R3		1	–99

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
4	√	54		3
3	√	76	√	666
1	√	76		0

Reservation Stations

FUs

0 (cache miss)

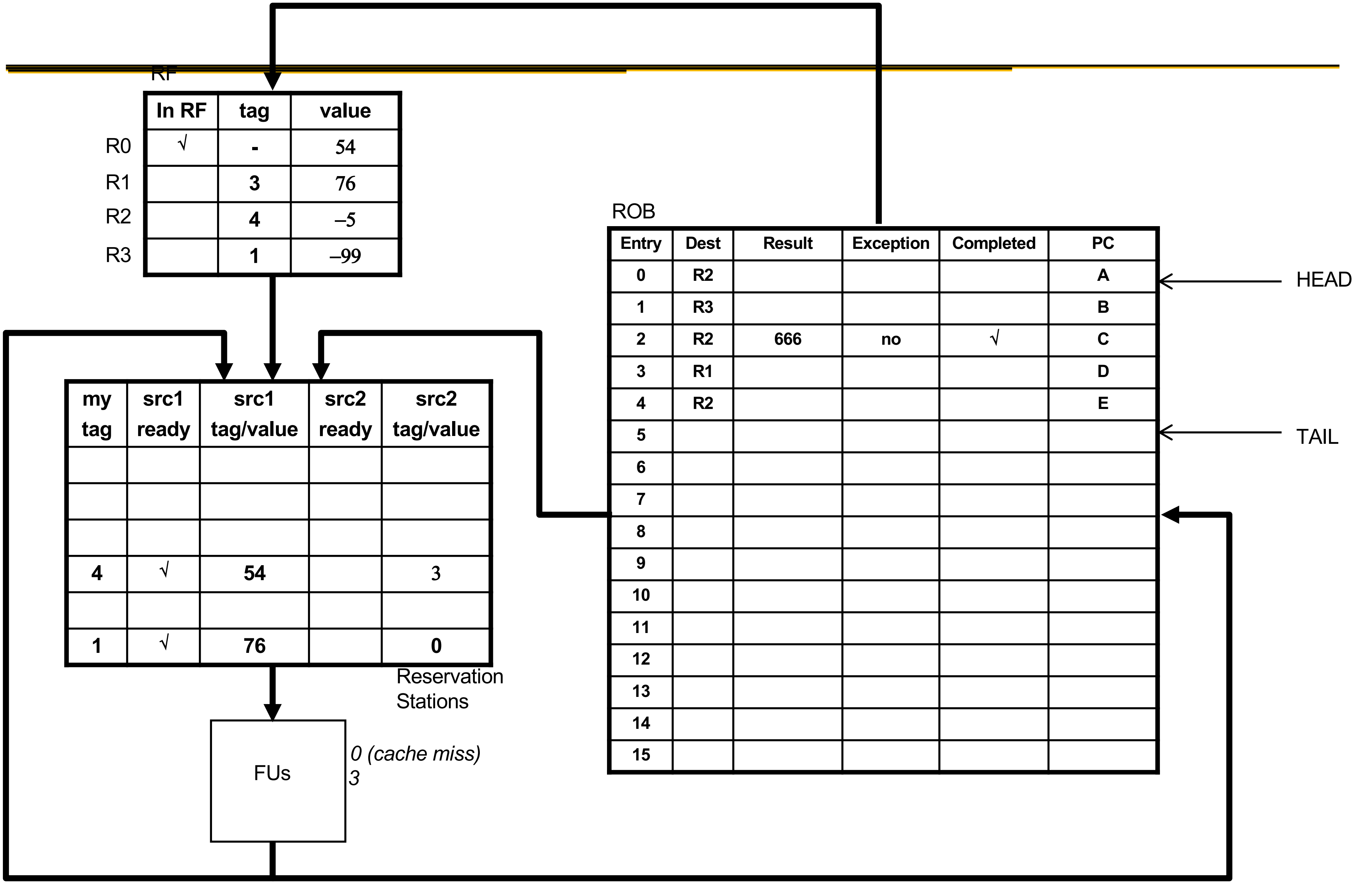
2, 666, no exception

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2	666	no	√	C
3	R1				D
4	R2				E
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

HEAD

TAIL



RF

	In RF	tag	value
R0	√	-	54
R1		3	76
R2		4	–5
R3		1	–99

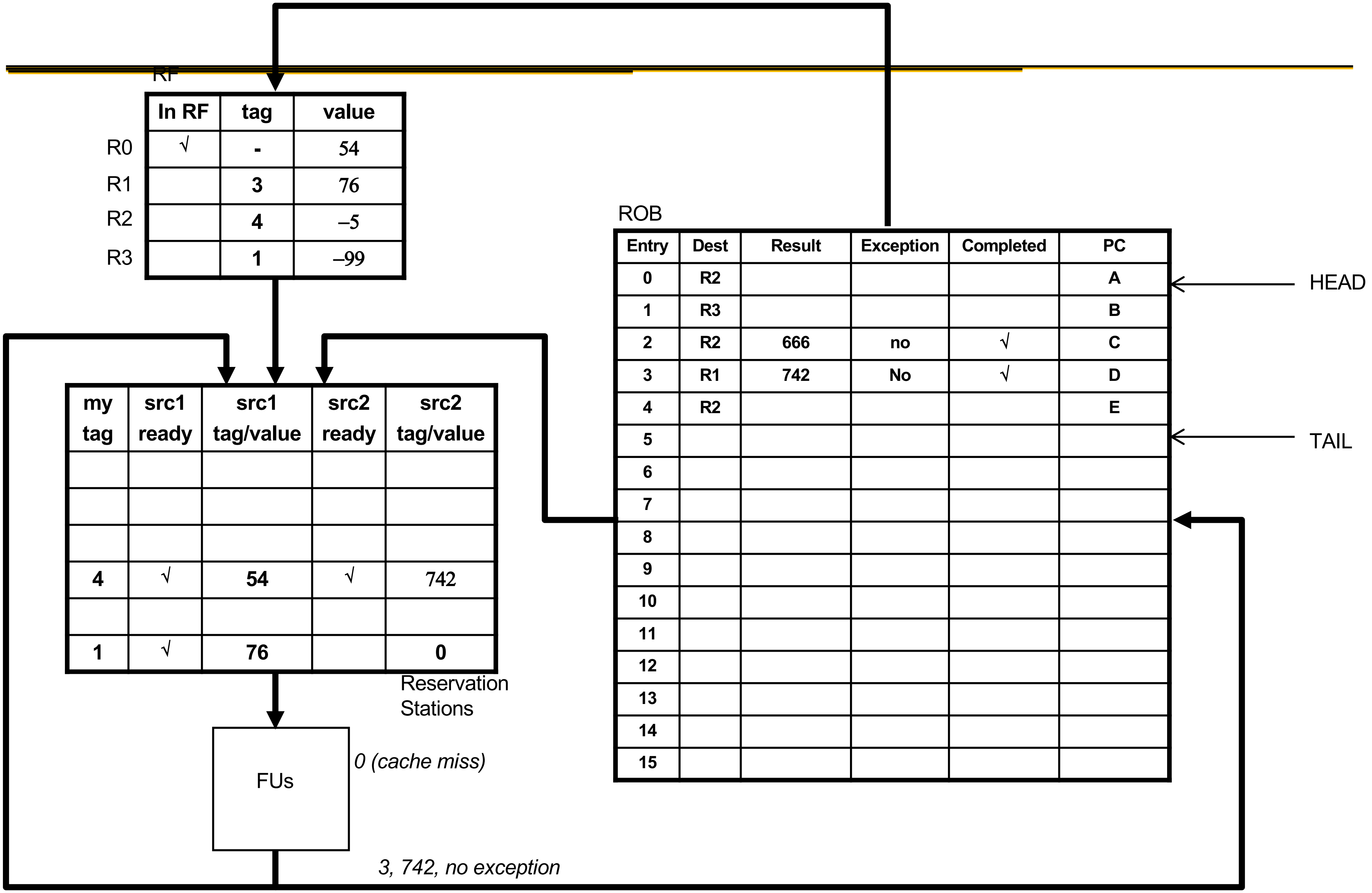
my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
4	√	54		3
1	√	76		0

Reservation Stations

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2	666	no	√	C
3	R1				D
4	R2				E
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

0 (cache miss)
3



RF

	In RF	tag	value
R0	√	-	54
R1		3	76
R2		4	–5
R3		1	–99

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
4	√	54	√	742
1	√	76		0

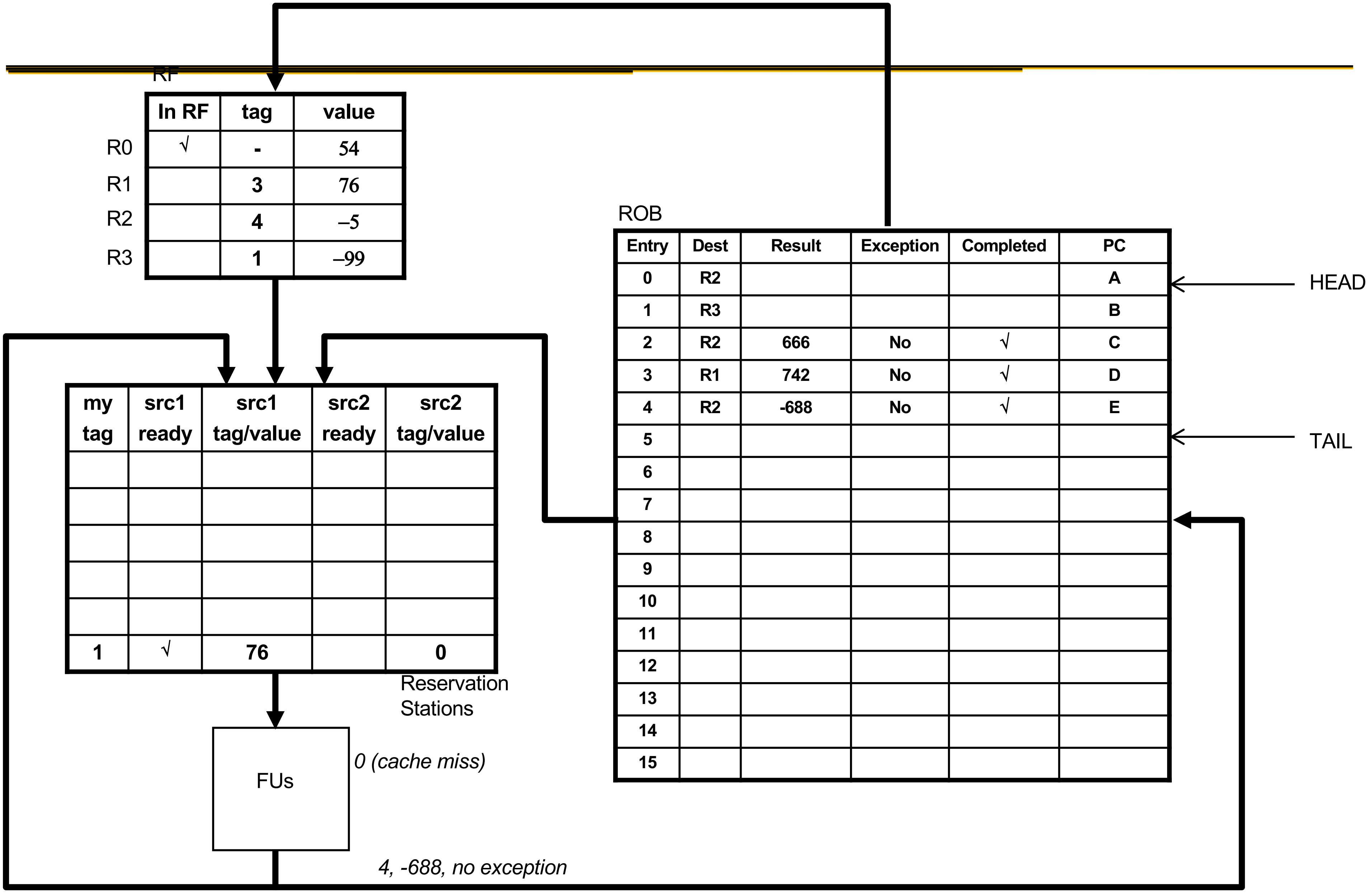
Reservation Stations

0 (cache miss)

3, 742, no exception

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2	666	no	√	C
3	R1	742	No	√	D
4	R2				E
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



RF

	In RF	tag	value
R0	√	-	54
R1		3	76
R2		4	–5
R3		1	–99

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
1	√	76		0

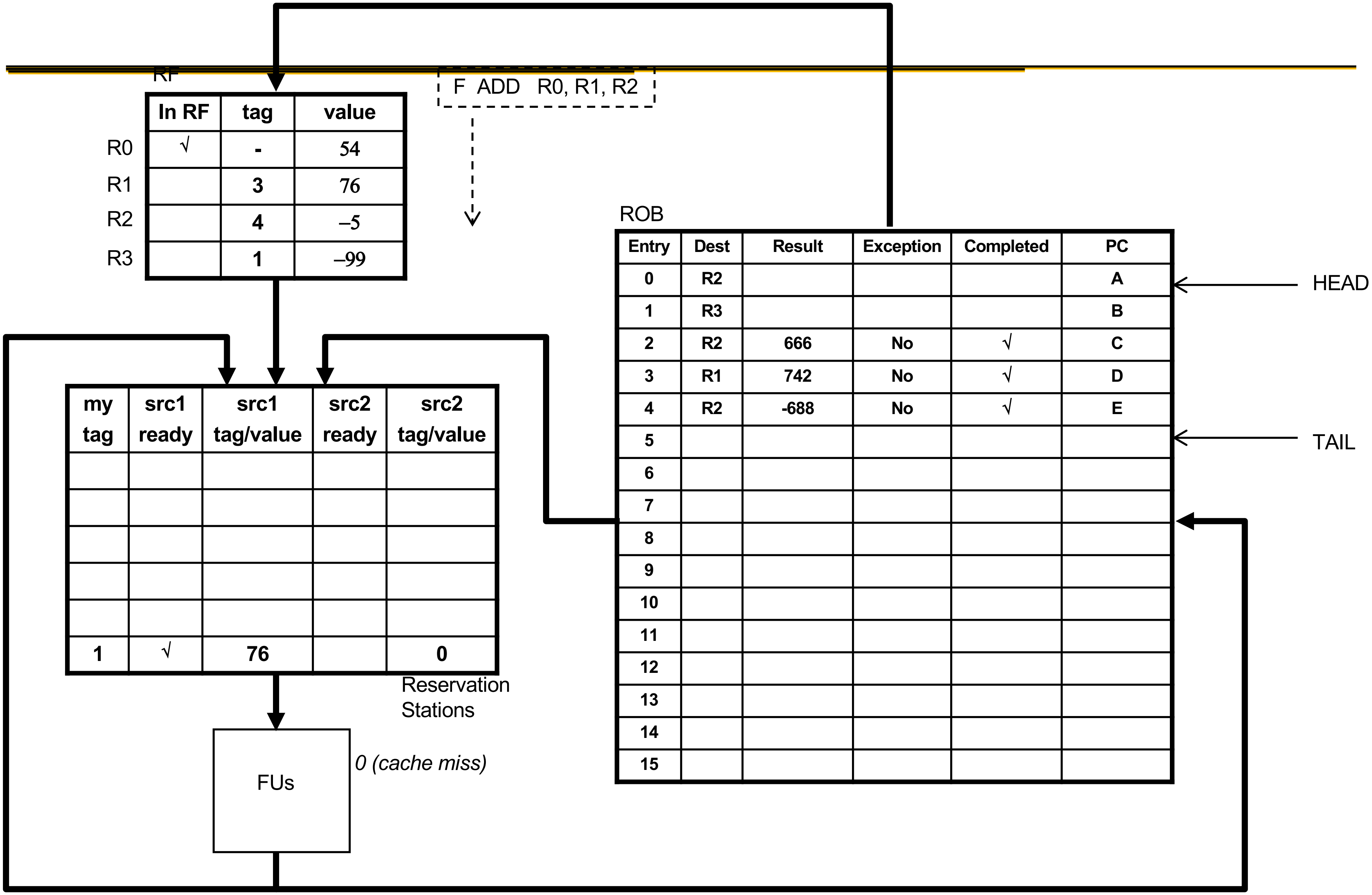
Reservation Stations

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2	666	No	√	C
3	R1	742	No	√	D
4	R2	-688	No	√	E
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

0 (cache miss)

4, -688, no exception



RF

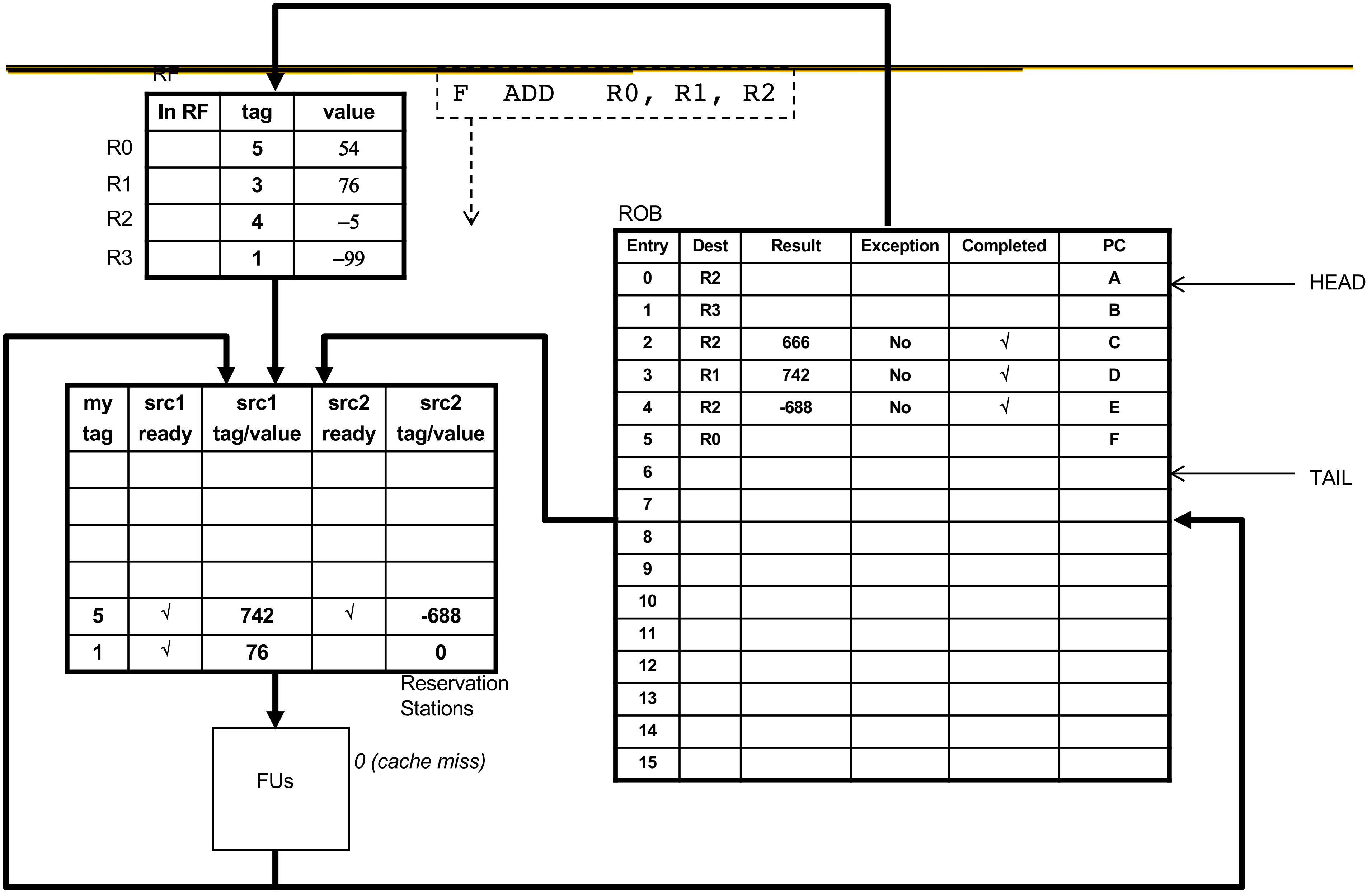
	In RF	tag	value
R0	√	-	54
R1		3	76
R2		4	–5
R3		1	–99

Reservation Stations

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
1	√	76		0

ROB

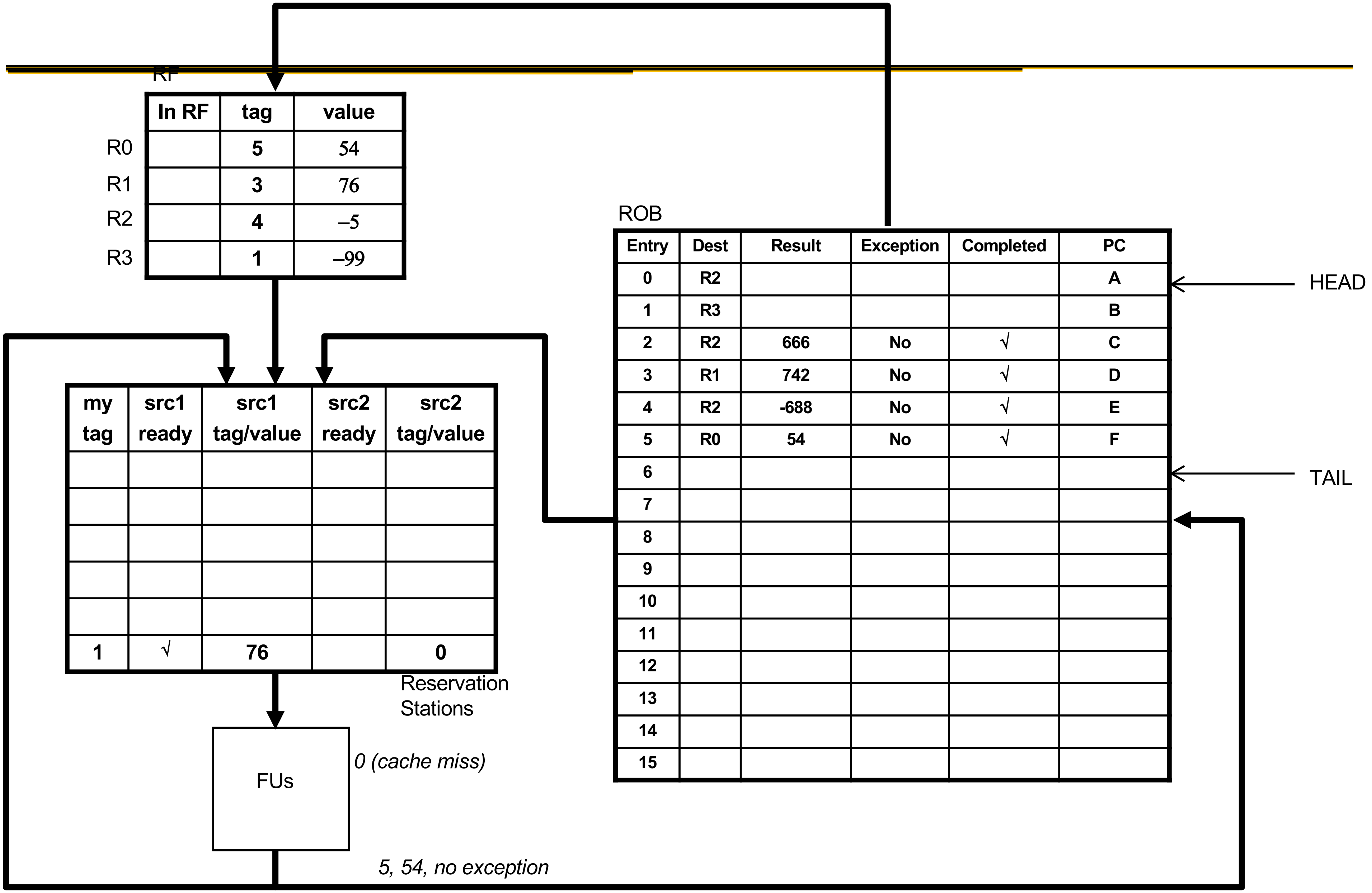
Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2	666	No	√	C
3	R1	742	No	√	D
4	R2	-688	No	√	E
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



	In RF	tag	value
R0		5	54
R1		3	76
R2		4	–5
R3		1	–99

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
5	√	742	√	-688
1	√	76		0

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2	666	No	√	C
3	R1	742	No	√	D
4	R2	-688	No	√	E
5	R0				F
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



RF

	In RF	tag	value
R0		5	54
R1		3	76
R2		4	–5
R3		1	–99

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
1	√	76		0

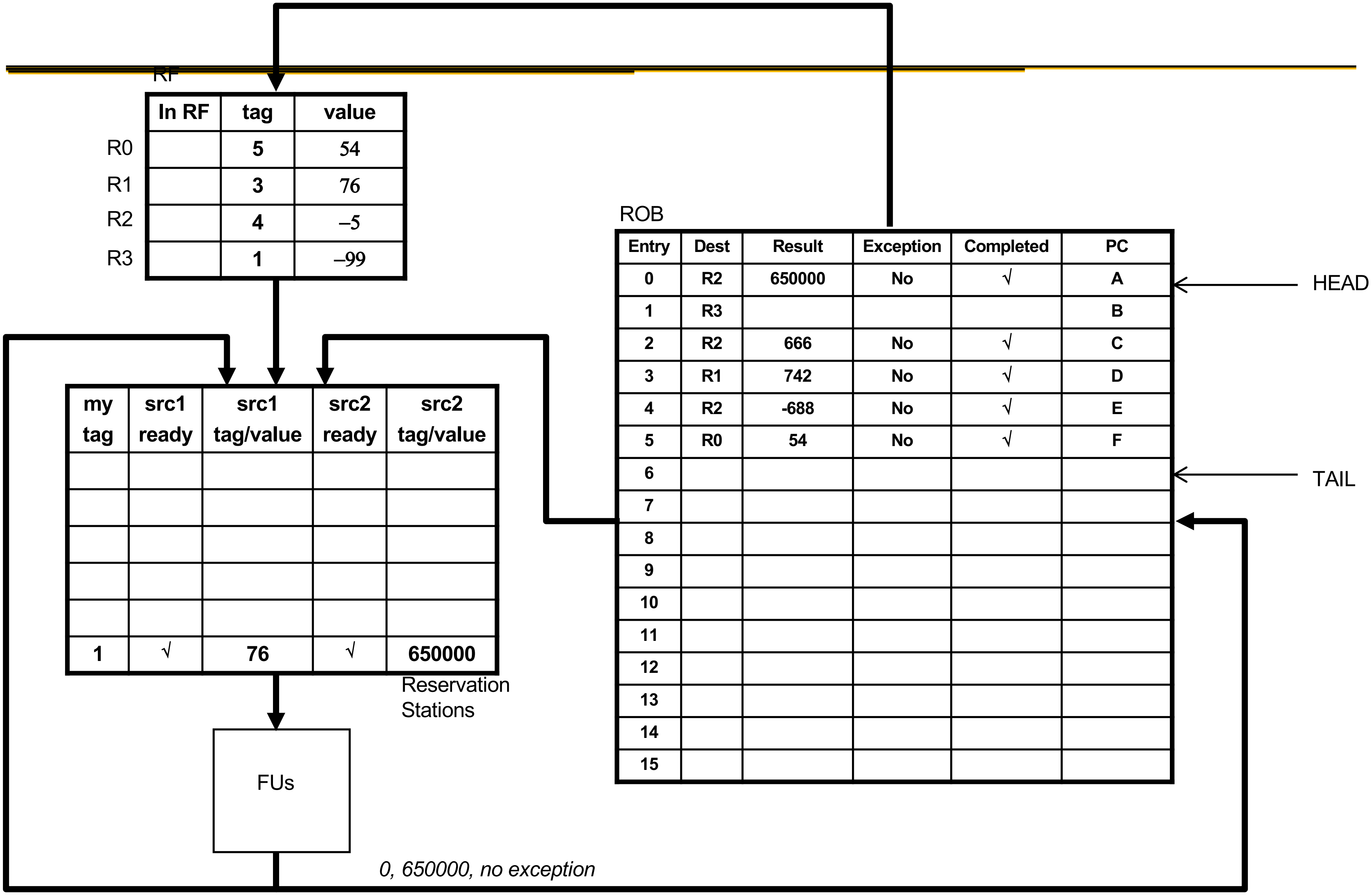
Reservation Stations

0 (cache miss)

5, 54, no exception

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2				A
1	R3				B
2	R2	666	No	√	C
3	R1	742	No	√	D
4	R2	-688	No	√	E
5	R0	54	No	√	F
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



RF

	In RF	tag	value
R0		5	54
R1		3	76
R2		4	–5
R3		1	–99

ROB

Entry	Dest	Result	Exception	Completed	PC
0	R2	650000	No	√	A
1	R3				B
2	R2	666	No	√	C
3	R1	742	No	√	D
4	R2	-688	No	√	E
5	R0	54	No	√	F
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

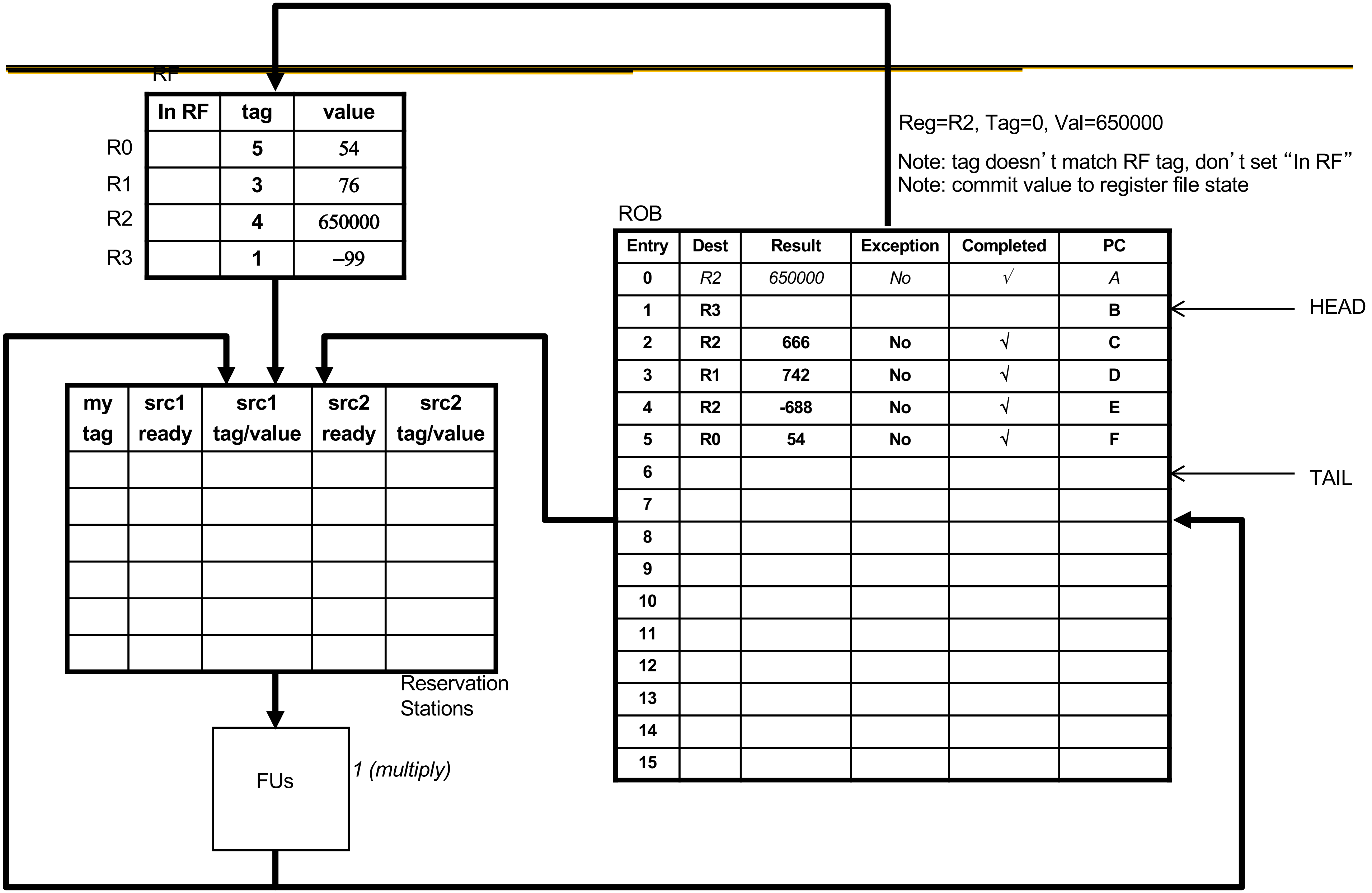
HEAD

TAIL

my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value
1	√	76	√	650000

Reservation Stations

0, 650000, no exception



RF

	In RF	tag	value
R0		5	54
R1		3	76
R2		4	650000
R3		1	–99

Reg=R2, Tag=0, Val=650000

Note: tag doesn’t match RF tag, don’t set “In RF”
Note: commit value to register file state

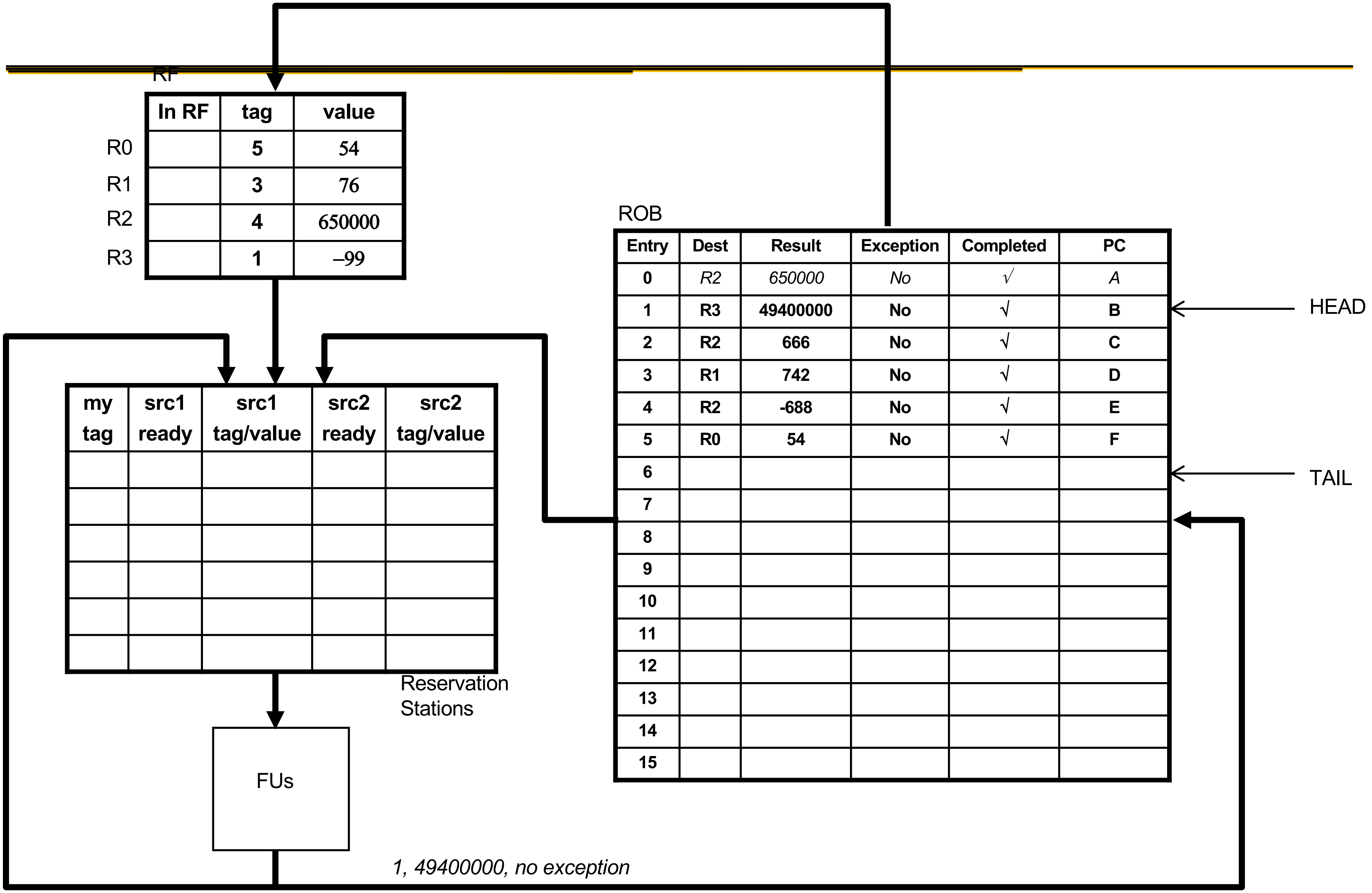
ROB

Entry	Dest	Result	Exception	Completed	PC
0	*R2*	*650000*	*No*	*√*	*A*
1	R3				B
2	R2	666	No	√	C
3	R1	742	No	√	D
4	R2	-688	No	√	E
5	R0	54	No	√	F
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

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

Two scenarios next…

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



RF

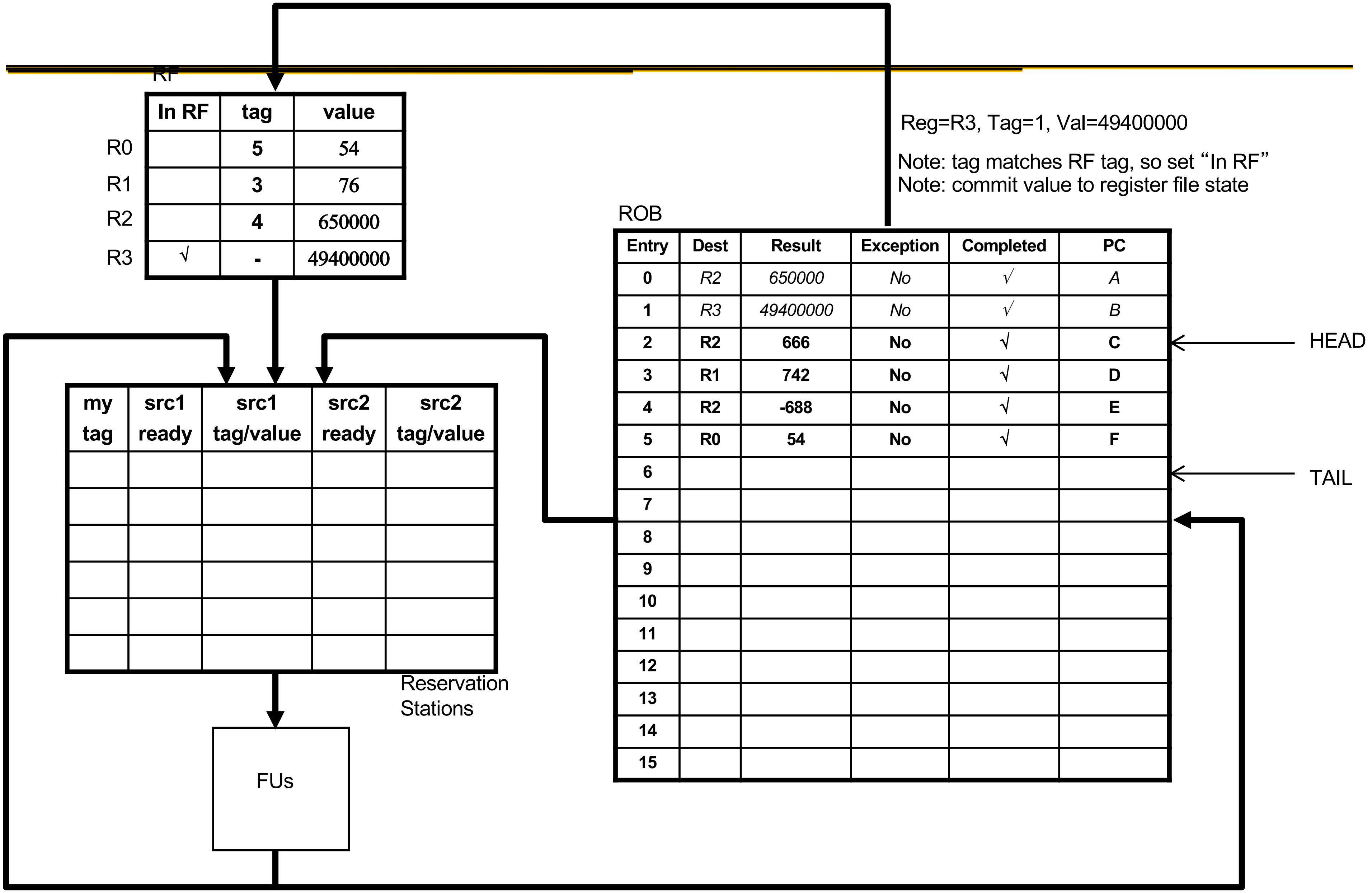
	In RF	tag	value
R0		**5**	54
R1		**3**	76
R2		**4**	650000
R3		**1**	–99

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

Reservation Stations

ROB

Entry	Dest	Result	Exception	Completed	PC
0	*R2*	*650000*	*No*	√	*A*
1	**R3**	**49400000**	**No**	√	**B**
2	**R2**	**666**	**No**	√	**C**
3	**R1**	**742**	**No**	√	**D**
4	**R2**	**-688**	**No**	√	**E**
5	**R0**	**54**	**No**	√	**F**
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



RF

	In RF	tag	value
R0		5	54
R1		3	76
R2		4	650000
R3	√	-	49400000

Reg=R3, Tag=1, Val=49400000

Note: tag matches RF tag, so set "In RF"
Note: commit value to register file state

ROB

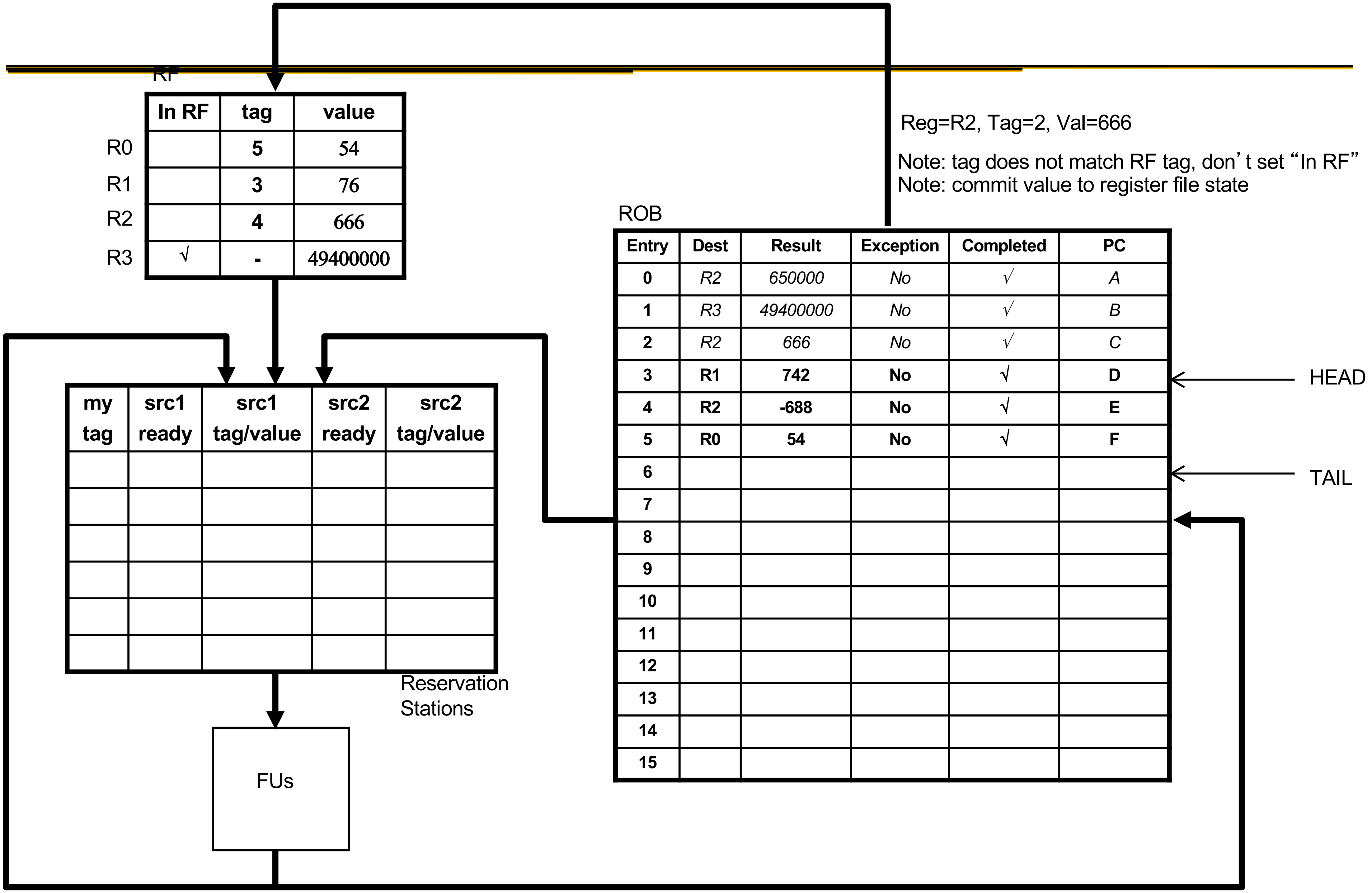
Entry	Dest	Result	Exception	Completed	PC
0	*R2*	*650000*	*No*	√	*A*
1	*R3*	*49400000*	*No*	√	*B*
2	R2	666	No	√	C
3	R1	742	No	√	D
4	R2	-688	No	√	E
5	R0	54	No	√	F
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

HEAD

TAIL

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

Reservation Stations



RF

	In RF	tag	value
R0		5	54
R1		3	76
R2		4	666
R3	√	-	49400000

Reg=R2, Tag=2, Val=666

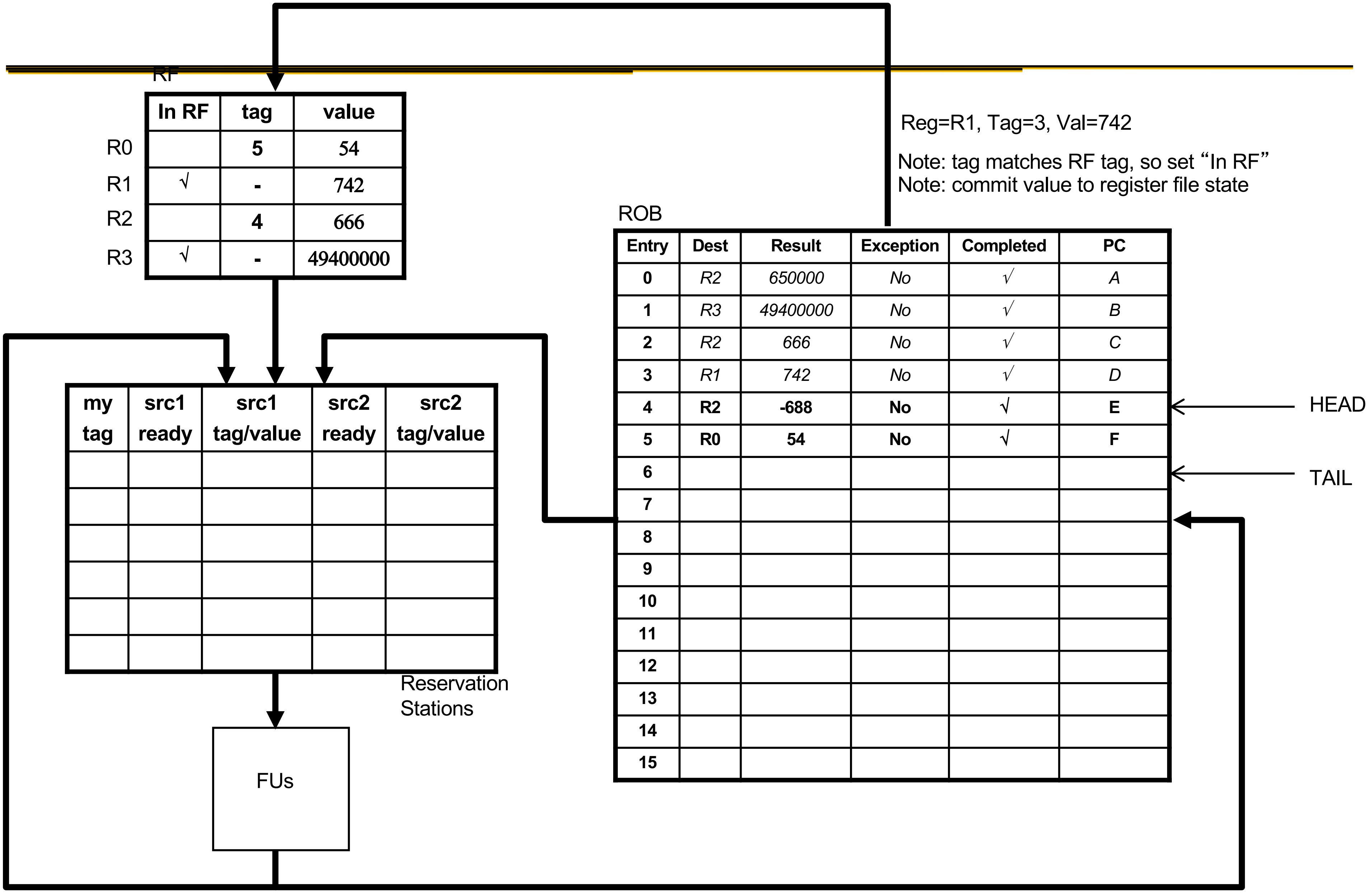
Note: tag does not match RF tag, don’t set “In RF”
Note: commit value to register file state

Reservation Stations

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

ROB

Entry	Dest	Result	Exception	Completed	PC
0	*R2*	*650000*	*No*	√	*A*
1	*R3*	*49400000*	*No*	√	*B*
2	*R2*	*666*	*No*	√	*C*
3	R1	742	No	√	D
4	R2	-688	No	√	E
5	R0	54	No	√	F
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



RF

	In RF	tag	value
R0		5	54
R1	√	-	742
R2		4	666
R3	√	-	49400000

Reservation Stations

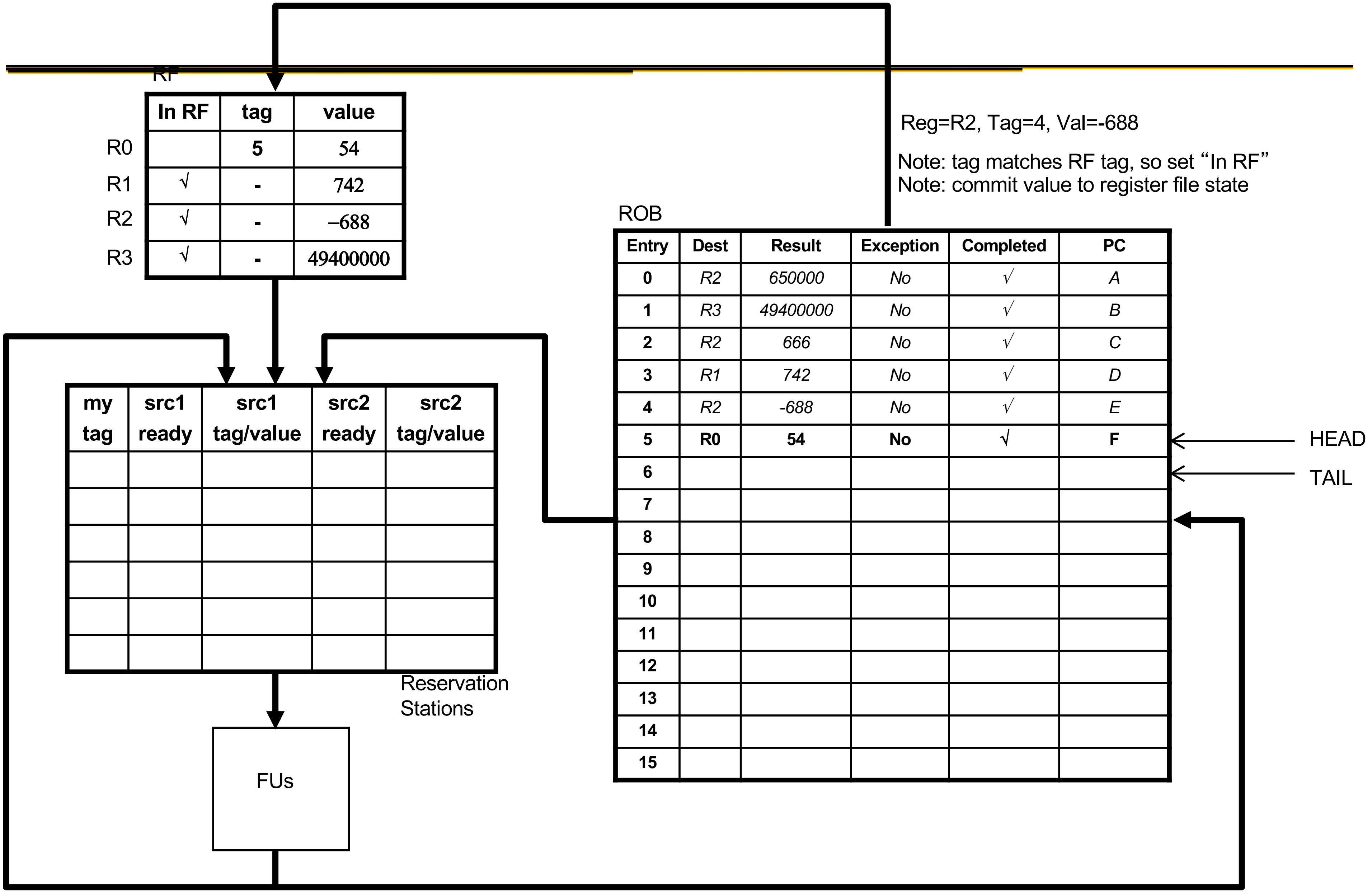
my tag	src1 ready	src1 tag/value	src2 ready	src2 tag/value

Reg=R1, Tag=3, Val=742

Note: tag matches RF tag, so set “In RF”
Note: commit value to register file state

ROB

Entry	Dest	Result	Exception	Completed	PC
0	*R2*	*650000*	*No*	√	*A*
1	*R3*	*49400000*	*No*	√	*B*
2	*R2*	*666*	*No*	√	*C*
3	*R1*	*742*	*No*	√	*D*
4	**R2**	**-688**	**No**	√	**E**
5	**R0**	**54**	**No**	√	**F**
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



RF

	In RF	tag	value
R0		5	54
R1	√	-	742
R2	√	-	–688
R3	√	-	49400000

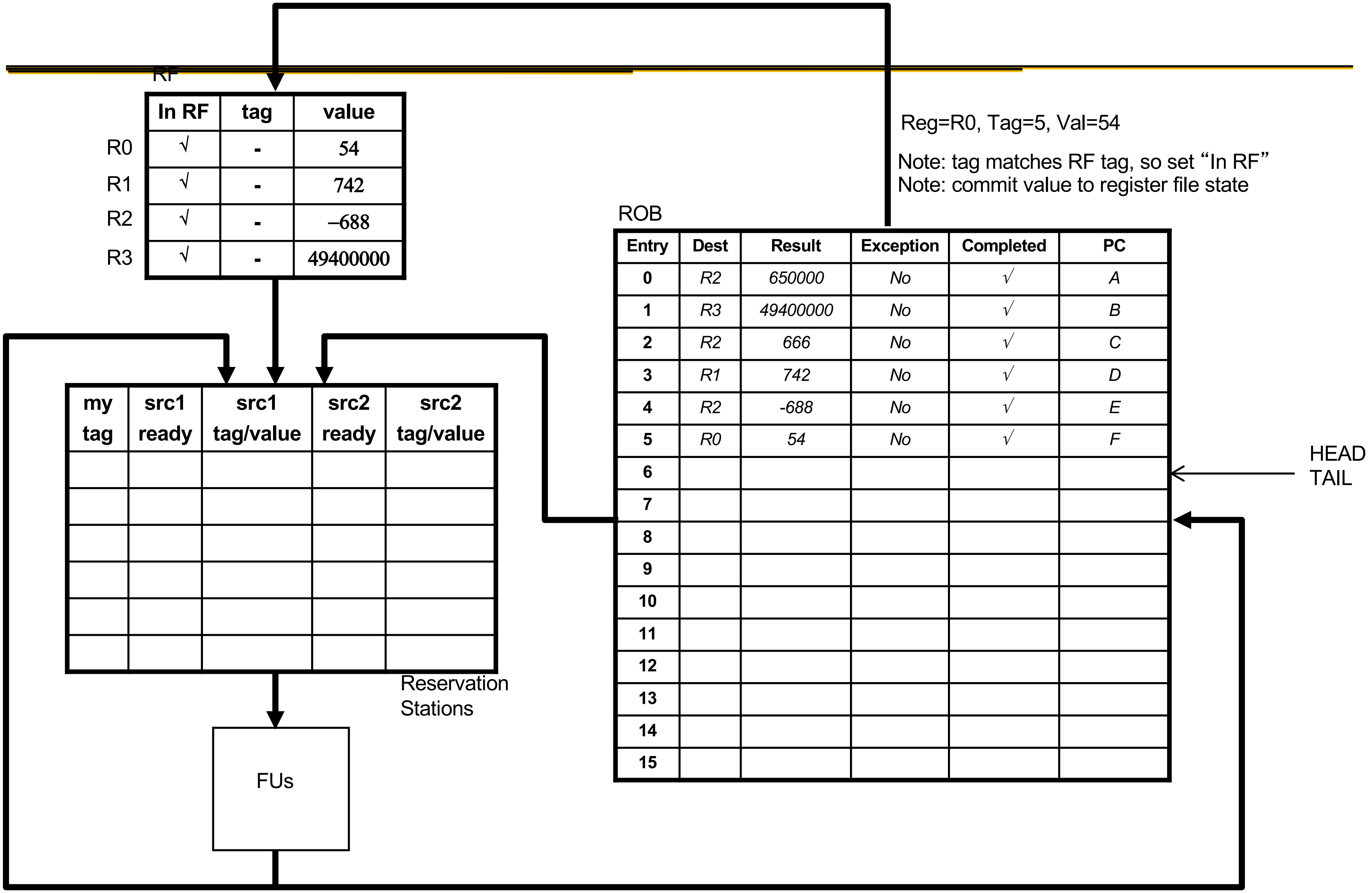
Reg=R2, Tag=4, Val=-688

Note: tag matches RF tag, so set "In RF"
Note: commit value to register file state

ROB

Entry	Dest	Result	Exception	Completed	PC
0	*R2*	*650000*	*No*	√	*A*
1	*R3*	*49400000*	*No*	√	*B*
2	*R2*	*666*	*No*	√	*C*
3	*R1*	*742*	*No*	√	*D*
4	*R2*	*-688*	*No*	√	*E*
5	**R0**	**54**	**No**	√	**F**
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

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



Reg=R0, Tag=5, Val=54

Note: tag matches RF tag, so set “In RF”
Note: commit value to register file state

RF

	In RF	tag	value
R0	√	-	54
R1	√	-	742
R2	√	-	–688
R3	√	-	49400000

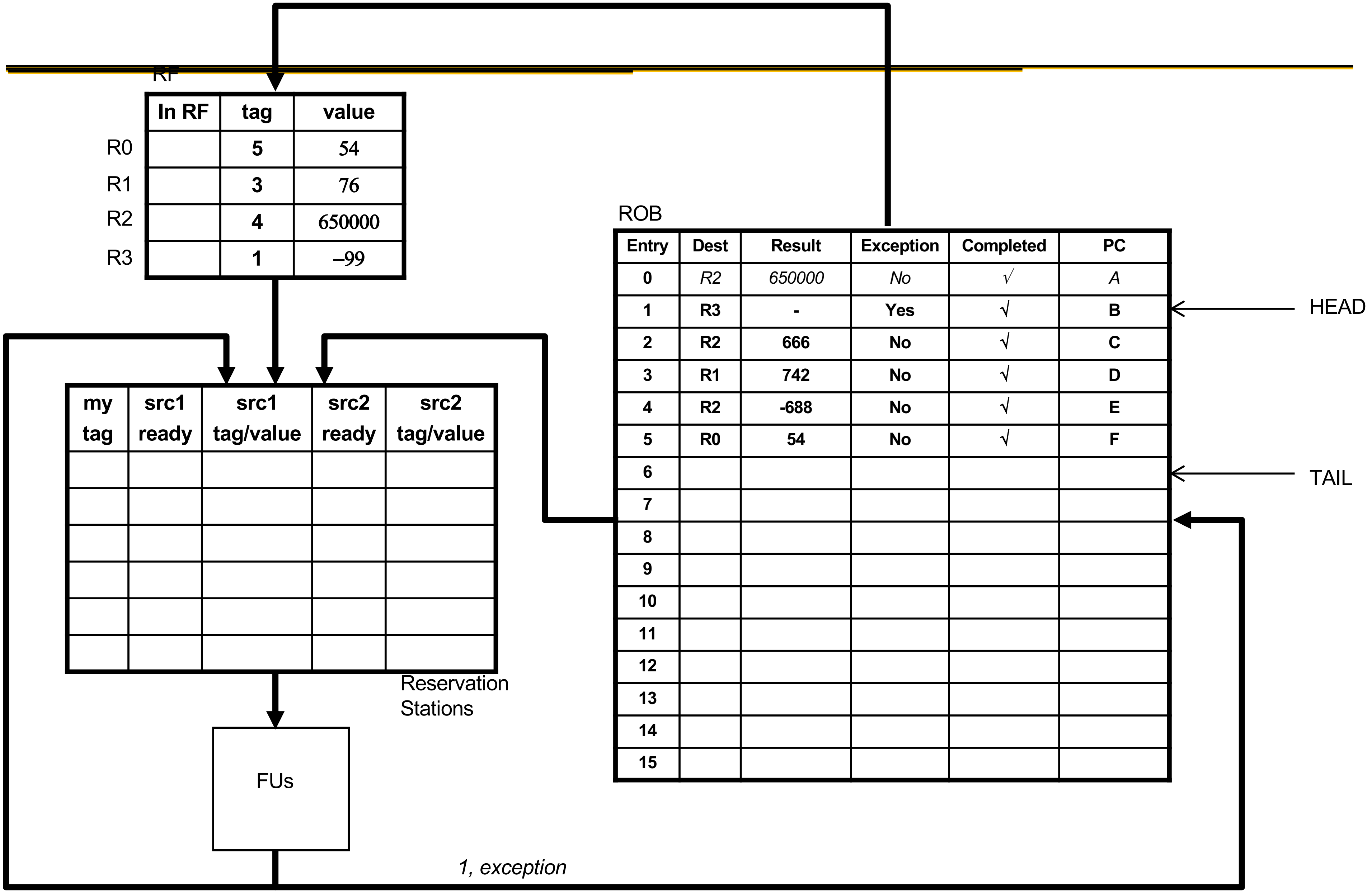
Reservation Stations

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

ROB

Entry	Dest	Result	Exception	Completed	PC
0	*R2*	*650000*	*No*	√	*A*
1	*R3*	*49400000*	*No*	√	*B*
2	*R2*	*666*	*No*	√	*C*
3	*R1*	*742*	*No*	√	*D*
4	*R2*	*-688*	*No*	√	*E*
5	*R0*	*54*	*No*	√	*F*
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

Now do scenario #2



RF

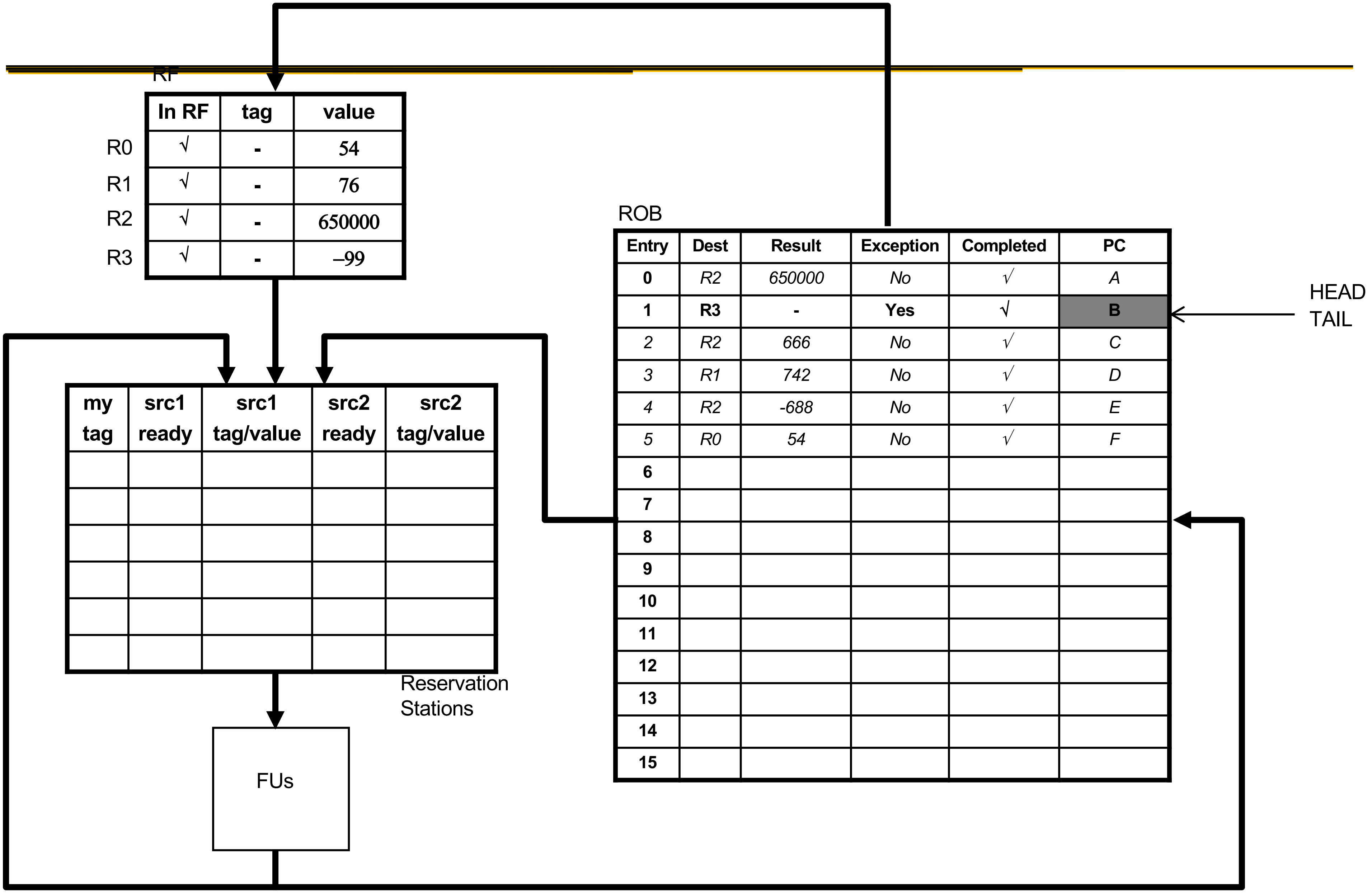
	In RF	tag	value
R0		5	54
R1		3	76
R2		4	650000
R3		1	–99

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

Reservation Stations

ROB

Entry	Dest	Result	Exception	Completed	PC
0	*R2*	*650000*	*No*	*√*	*A*
1	R3	-	Yes	√	B
2	R2	666	No	√	C
3	R1	742	No	√	D
4	R2	-688	No	√	E
5	R0	54	No	√	F
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					



RF

	In RF	tag	value
R0	√	-	54
R1	√	-	76
R2	√	-	650000
R3	√	-	–99

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

Reservation Stations

FUs

ROB

Entry	Dest	Result	Exception	Completed	PC
0	*R2*	*650000*	*No*	√	*A*
1	**R3**	**-**	**Yes**	√	**B**
2	*R2*	*666*	*No*	√	*C*
3	*R1*	*742*	*No*	√	*D*
4	*R2*	*-688*	*No*	√	*E*
5	*R0*	*54*	*No*	√	*F*
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					

HEAD
TAIL

Handling Branch Mispredictions

- Branch mispredictions are like exceptions?
  - Yes and No
  - Handle exactly like exception (low performance)
    - Wait until mispredicted branch reaches head of ROB
    - Squash entire ROB, set all "In RF" bits
    - Restart along 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.

Module 6.4: ILP 4, History Buffer, Future File, Memory Dependences and Ordering

Objectives and Outline

- Objectives:
  - Understand variations of ROB: history buffer and future file
  - Understand how memory dependences are handled with OOO scheduling
- Outline:
  - History Buffer and Future File
  - Memory dependences: WAW, WAR, RAW
  - Handling loads and stores in the pipeline
  - Load forwarding
  - Memory disambiguation and its speculation

Historical Perspective

- 1967: Tomasulo's algorithm
- ~1985: history buffer
- 1985: Andre Pleszkun & James E Smith invented reorder buffer and future file to support precise interrupt
- 1987: Guri Sohi showed reorder buffer could be used for renaming and supporting speculative execution

History Buffer (HB)

- Operation of HB
  - HB is a FIFO with head and tail pointer (like ROB)
  - Instruction dispatch
    - Reserve HB entry at tail, advance tail ptr
    - 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 instruction completes
    - Updates occur out-of-order: "messy register file" (imprecise)
    - If instruction caused an exception, mark exception bit in the HB for offending instruction
  - Instruction retire
    - Check instruction at the head of the HB

      If completed, check exception bit

      If no exception, simply advance head ptr

      If exception, restore precise register file state using the saved values in HB – "undo" the speculative updates

History Buffer (cont.)

Initial register file contents

F0	
F2	
F4	
F6	50
F8	60
F10	

HB and Register File when exception reaches at HEAD

F0	
F2	
F4	
F6	189
F8	-23
F10	

HB

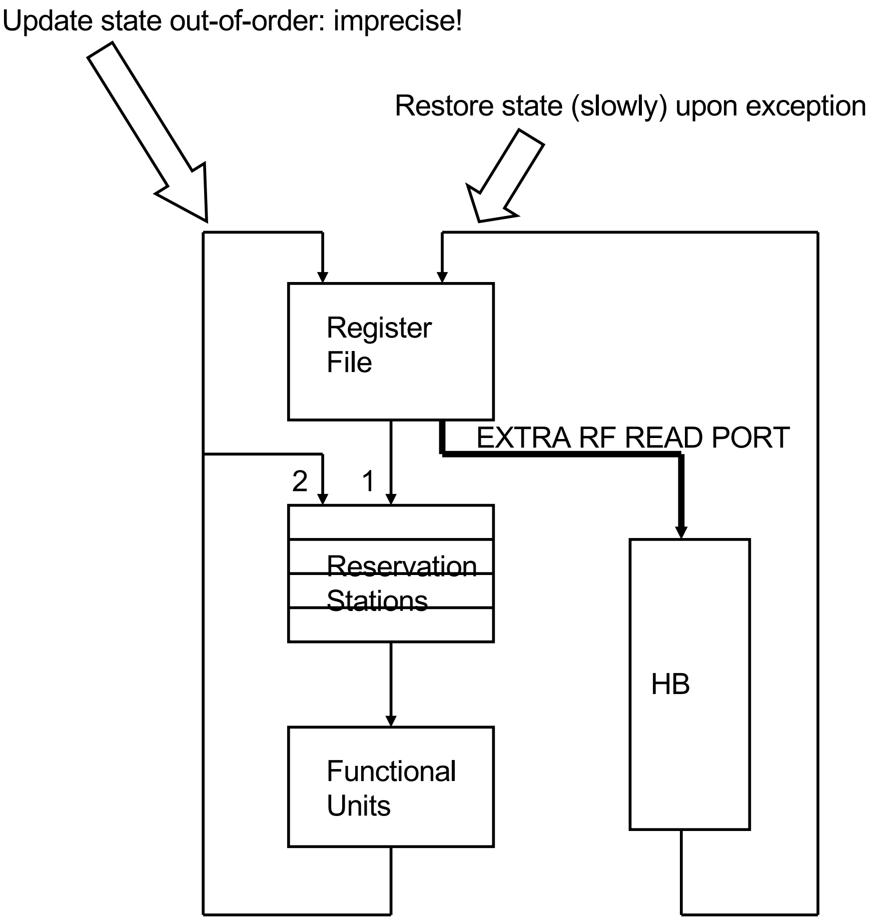
	Entry	Dest	OLD VALUE	Exception	Valid	PC
	1					
	2	F6		no	√	A
	3	F2		no	√	B
HEAD	4	F0		yes	√	C
	5	F8	60		√	D
	6	F10				E
	7	F6	50		√	F
TAIL	8					
	9					
	10					
	…					

Recover register file state
from tail to head

Restored (precise) register file contents

F0	
F2	
F4	
F6	50
F8	60
F10	

Precise Tomasulo Pipeline with HB



Update state out-of-order: imprecise!

Restore state (slowly) upon exception

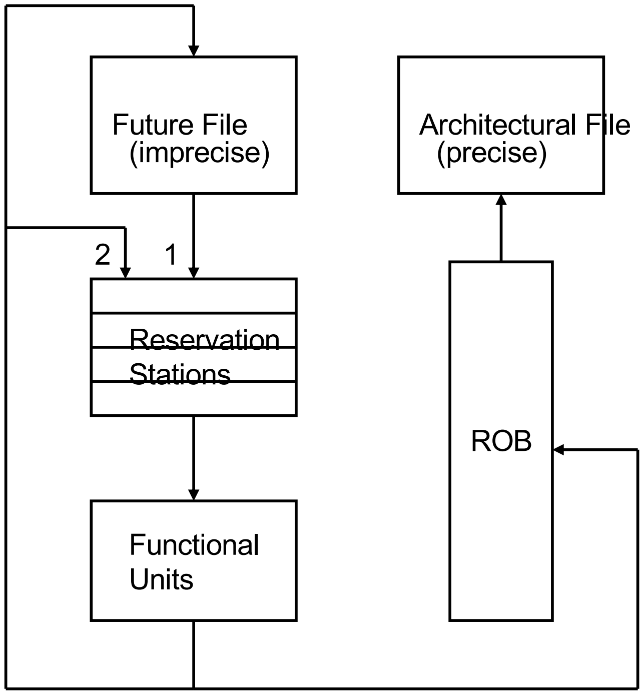
HB analysis

- 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

Future File (FF)

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

Precise Tomasulo Pipeline with FF



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)

Recall Memory Dependences…

- Dependences occur not just between register addresses, but also memory addresses: Memory Dependences, e.g.
  - A: ST R1, [R2]   // Mem[R2] <- R1
  - B: LD R3, [R2]   // R3 <- Mem[R2]
  - C: LD R5, 10[R4]   // R5 <- Mem[R4+10]
- B is dependent on A, but is C dependent on A?
  - True dependent if R4+10 = R2
  - Not dependent if R4+10 != R2
- Memory dependences are harder to detect because they are based on register values (instead of register IDs)

Memory Data Dependences

- Types of Dependences

WAW	WAR	RAW
store X	load X	store X
:	:	:
store X	store X	load X

- Differences vs. register dependences
  - It depends on register contents, not register IDs
  - Renaming cannot be done at hardware (expensive to do in compiler)
  - Affected by choice of memory consistency

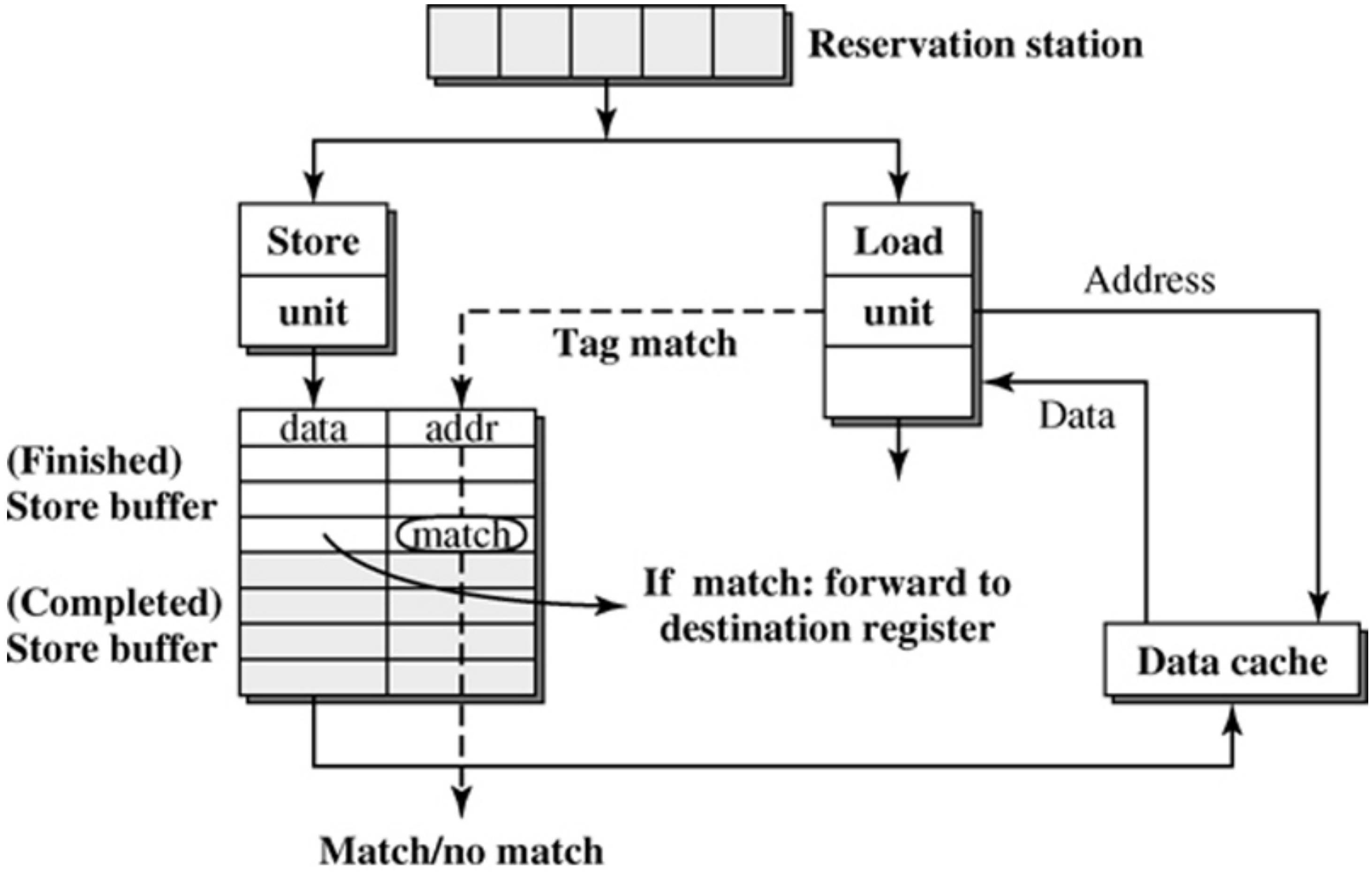
Memory Dependency

- Stores are released to cache in order
  - This preserves sequential (in-order) state in cache/memory (so that recovery from exceptions is possible)
  - This takes care of output dependences
- Loads access the cache out of order
  - with respect to each other
  - with respect to previous, pending, *independent* stores
  - But not with respect to previous, pending, *dependent* stores
- A load/store aliases with another load/store if their effective addresses match

How Loads and Stores are Handled

- Store
  - D stage: place on store reservation station
  - EX stage: compute effective address of store
  - RE stage: place in store buffer (to be release to cache)
- Load
  - D stage: place on load reservation station
  - EX stage: compute effective address of load
  - MEM stage:
    - Check store buffer for dependences. If any, get data (load forwarding)
    - If none, get data from cache (load bypassing, load bypasses older store)
    - If it can't be determined, *stall it* or *speculate* (memory dependence speculation)

Load Forwarding

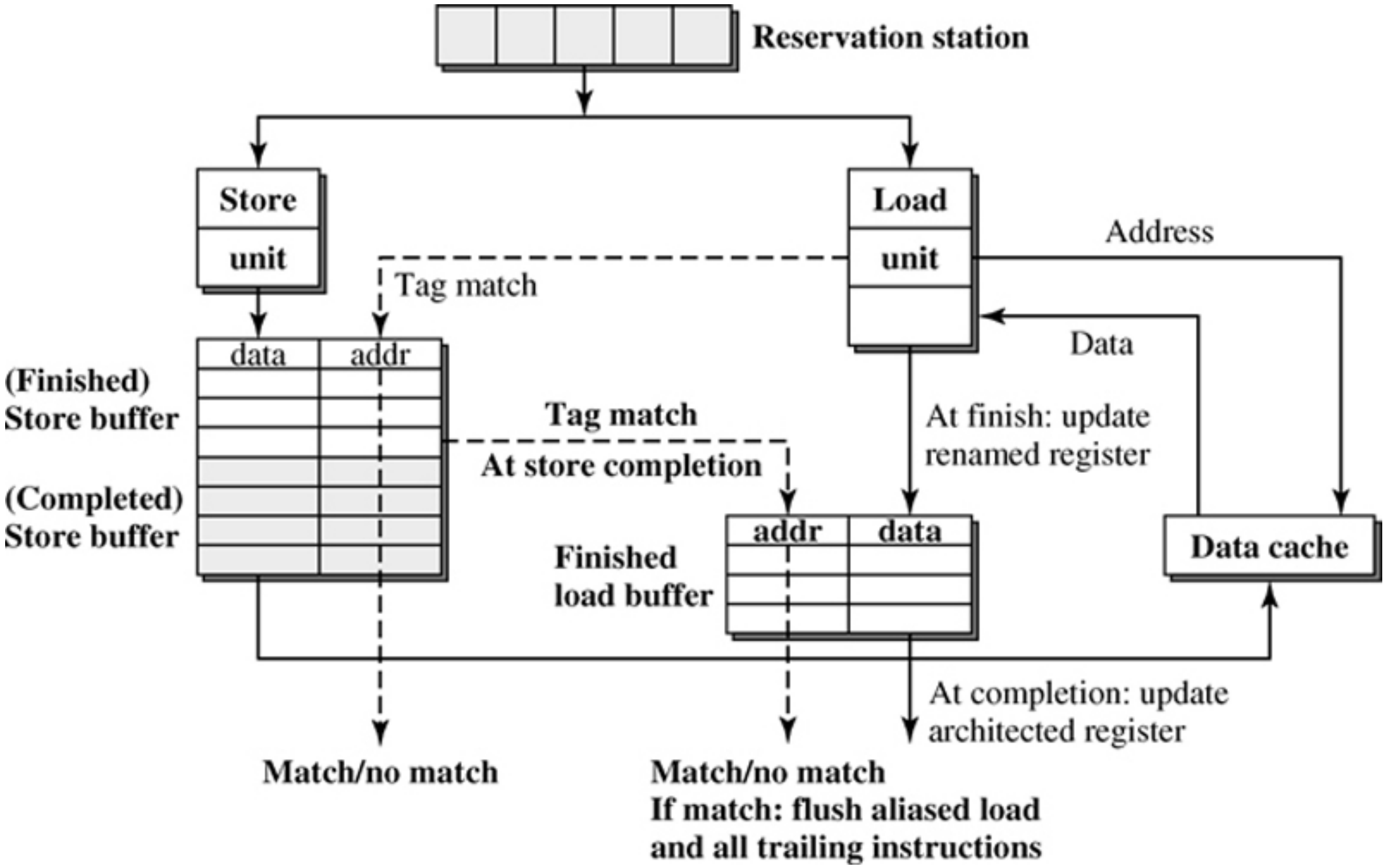


Memory Disambiguation Speculation

- What if aliases are rare?

1. Loads don't wait for addresses of all prior stores
2. Full address comparison of stores that are ready
3. Bypass if no match, forward if match
4. Check all store addresses when they commit
    - – No matching loads – speculation was correct
    - – Matching unbypassed load – incorrect speculation
5. Replay starting from incorrect load

Speculative Memory Disambiguation



Use of Prediction

- If aliases are rare: static prediction
  - Predict no alias every time
    - Why even implement forwarding? PowerPC 620 doesn’t
  - Pay misprediction penalty rarely
- If aliases are more frequent: dynamic prediction
  - Use PHT-like history table for loads
    - If alias predicted: delay load
    - If aliased pair predicted: forward from store to load

      More difficult to predict pair [store sets, Alpha 21264]
  - Pay misprediction penalty rarely
- Memory cloaking [Moshovos, Sohi]
  - Predict load/store pair
  - Directly copy store data register to load target register
  - Reduce data transfer latency to absolute minimum

Module 6.5: Superscalar and the Limit of ILP

Objectives and Outline

- Objectives:
  - Understand the limit of pipelining
  - Understand the concept and challenges of superscalar processing
  - Understand the complexity of superscalar implementation
- Outline:
  - The limit of pipelining
  - Superscalar concept and what each pipeline stage must do
  - Techniques to support superscalar processing
  - Superscalar implementation complexity

Limits of Pipelining

- CPU_time = IC x CPI x CT
- Scalar pipelining
  - CPI ≥ 1 (Flynn's bottleneck)
  - ∴ reduce CT further, increase instructions per *second*
- Problem: limits of pipelining
  1. Latch overhead and clock skew
     - Overhead becomes a larger component of cycle
     - Pipelining eventually becomes inefficient
     - Diminishing returns
  2. Can only break logic up so far
  3. Pipelining is ineffective when there are dependences
     - Ex: Can pipeline L1 cache but it still takes X ns to get data out. An instruction that needs that data must wait X ns regardless of how fast you make the clock and how many stages there are.

Superscalar processing

- scalar = 1 instruction/cycle
- superscalar: N instructions/cycle
  - Increase *bandwidth* at each stage of the pipeline

IF	ID	EX	MEM	WB		
IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB
		IF	ID	EX	MEM	WB

- Formal definitions versus popular usage
  - Formal: superscalar applies to any multiple-issue machine
  - Popular: superscalar also implies dynamic scheduling
  - Statically scheduled, multiple-issue machines are popularly called VLIW – "very long instruction word"
    - compiler groups multiple independent instructions into a larger package

How to Get to Superscalar

- Inorder scalar -> inorder superscalar

In-order Pipeline

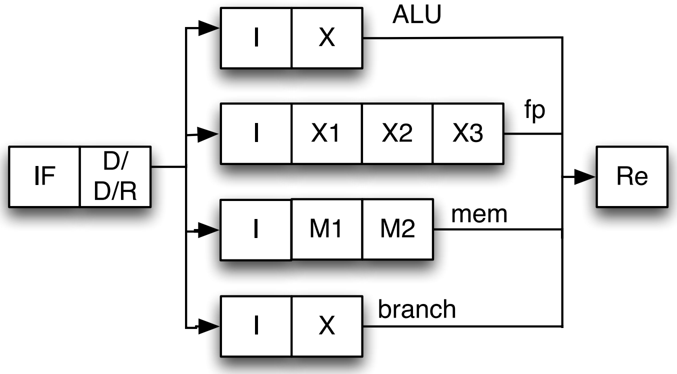
IF	ID	EX	M	WB

Superscalar in-order pipeline

IF	ID	EX	M	WB
IF	ID	EX	M	WB

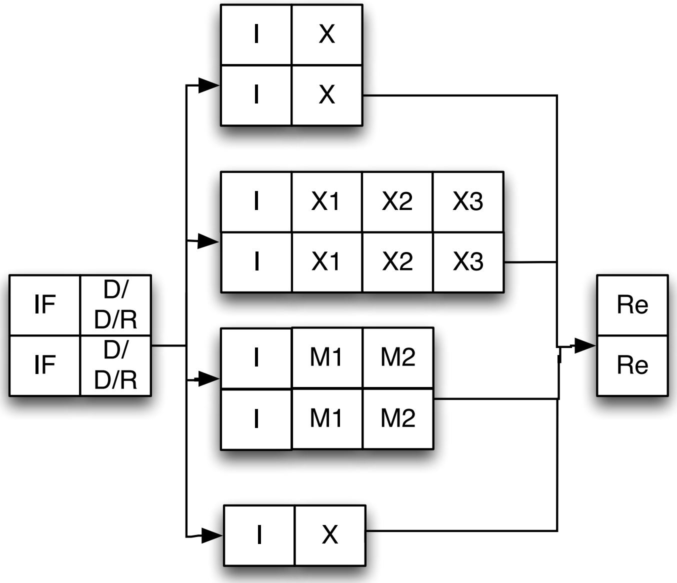
- OOO scalar -> OOO superscalar

Out-of-order (Dynamic) Pipeline



IF D/ D/R
I X ALU
I X1 X2 X3 fp
I M1 M2 mem
I X branch
Re

Superscalar OOO pipeline



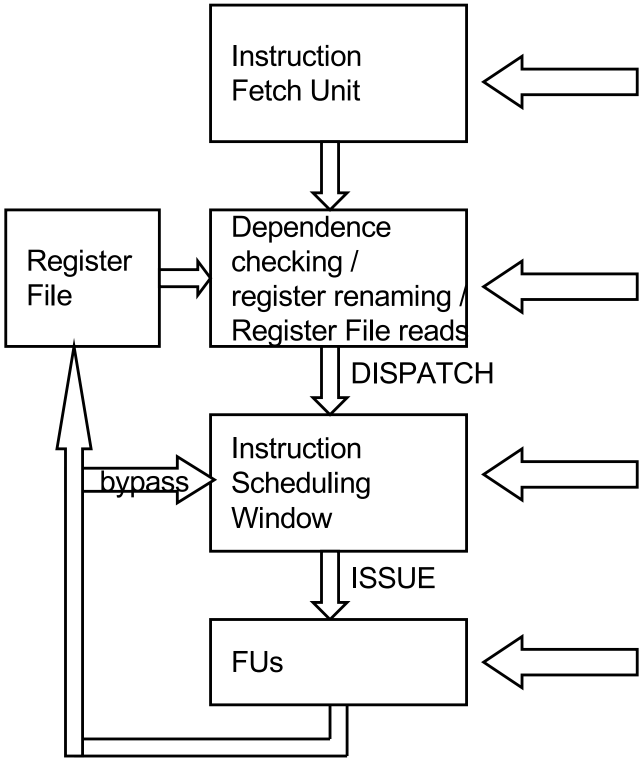
IF D/ D/R
IF D/ D/R
I X
I X
I X1 X2 X3
I X1 X2 X3
I M1 M2
I M1 M2
I X
Re
Re

New complexity

- IF
  - Parallel access to I$
  - Instruction alignment issues
- ID
  - Parallel register access: highly multi-ported register file
  - RAW hazard *cross-checks* required
    - Renaming/dependence checking now more complex than RF lookup
    - Must also check prior instructions in fetch group
- IS/EX
  - Parallel execution: replicate datapaths, FUs, etc.
- MEM
  - > 1 per cycle? If so, multi-ported D$
- WB
  - Many result buses (bypasses) and register file write ports

Superscalar complexity and cycle time

- The IPC / CT trade-off
  - Increasing superscalar complexity can hurt cycle time
  - Microarchitect must balance instr. per cycle (IPC) and cycle time (CT) to achieve best overall performance
- Where complexity (potentially) impacts cycle time


Instruction Fetch Unit
Register File
Dependence checking / register renaming / Register File reads
DISPATCH
bypass
Instruction Scheduling Window
ISSUE
FUs

•Interleaved I$ for good fetch bandwidth requires extra shift/alignment logic (more later).
•I$ / branch predictor performance critical for superscalar: larger I$/predictor is slower.

•Must cross-check dependences among newly-fetched instructions – increases rename latency.
•Highly multi-ported register file increases register file read latency.

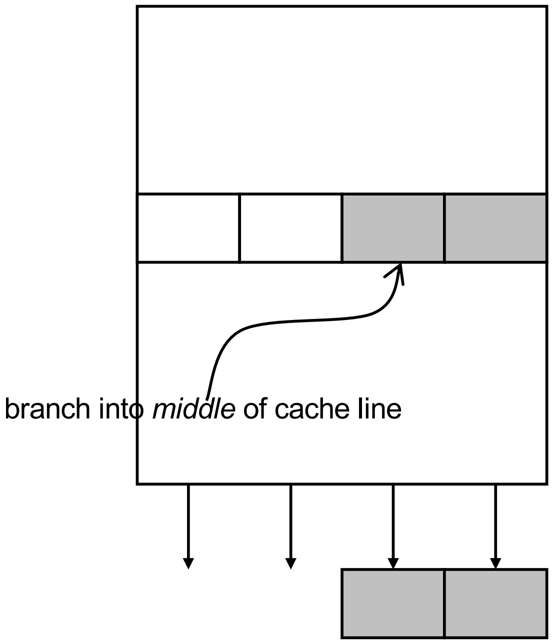
•Exposing more parallelism requires LARGE window (e.g., 64-entry or 128-entry ROB)
•Superscalar means high issue rate (multiple instructions issued per cycle)
issue latency $\propto$ WINDOW SIZE × ISSUE RATE

•<u>Many</u>, <u>long</u> bypasses.
•Wire delay is not scaling as well as transistor delay in future technologies.
•Bypasses are becoming a cycle time problem.

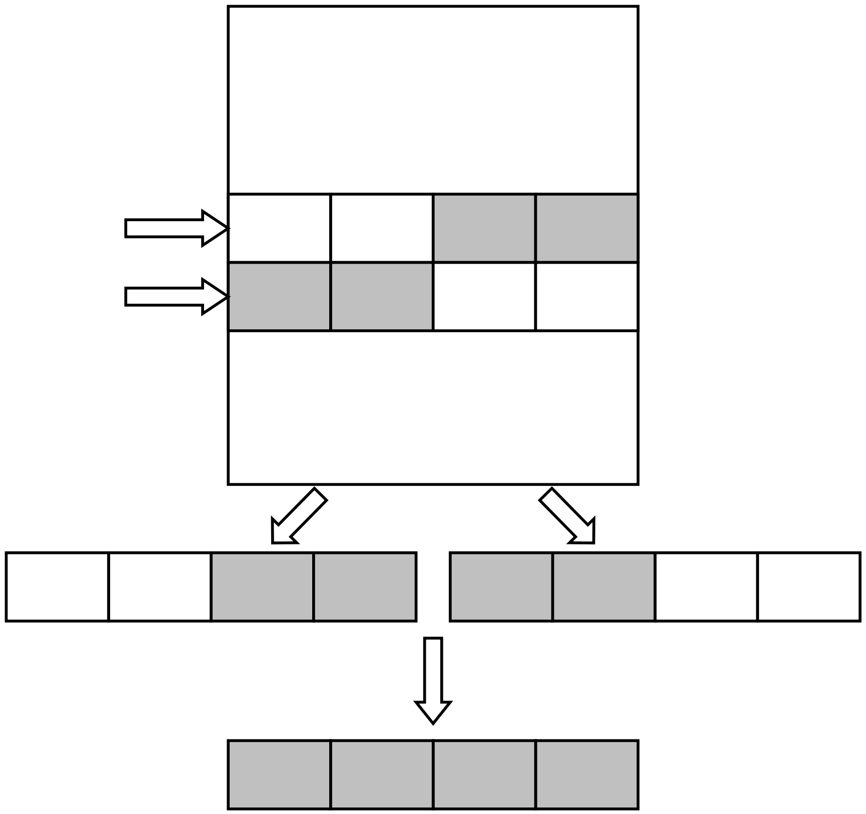
Superscalar evolution

- Historical perspective
  - 1 integer + 1 floating-point
  - Any 2 instructions
  - Any 4 instructions
  - Any "n" instructions? How far?

High-bandwidth instruction fetching



Average fetch bandwidth < 4/cycle



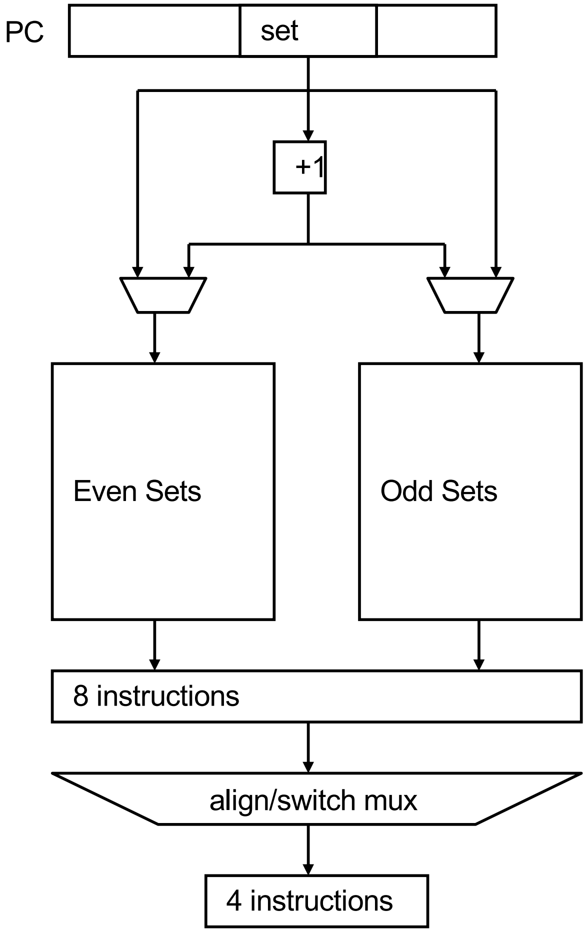
- Dual-port the I$ in order to read out two *consecutive* cache lines
- Merge needed instructions together to get 4/cycle bandwidth
- EXPENSIVE: true dual-porting increases area quadratically

Inexpensive dual-porting: interleaving

- Realization
  - Don’t need to read out *any* two cache lines
  - Just need to read out two *consecutive* cache lines
  - Split cache into two banks: one bank holds even addresses, other holds odd addresses
  - This is called 2-way interleaving or banking
  - Guarantees fetching full cache line worth of instructions (if branch not encountered first!)

Interleaving (cont.)

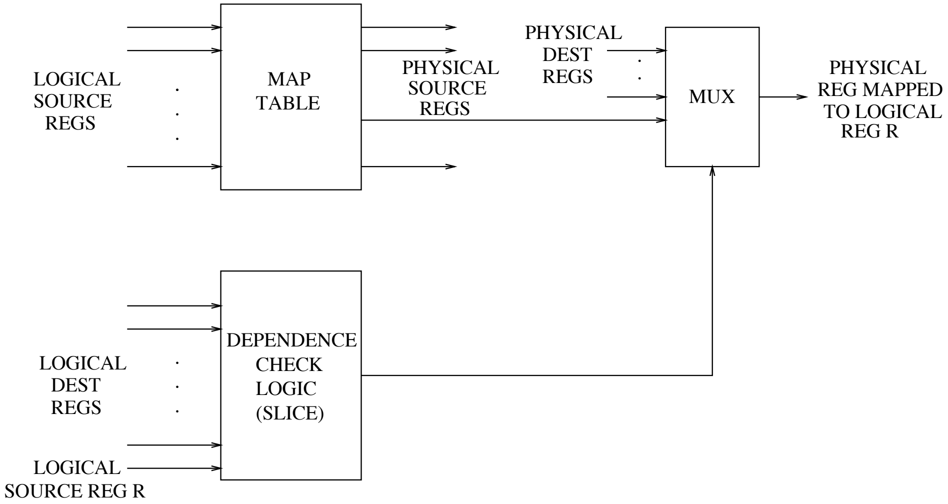
- Simplified IBM POWER solution



Total cache size hasn’t changed
(1 bank = ½ cache)

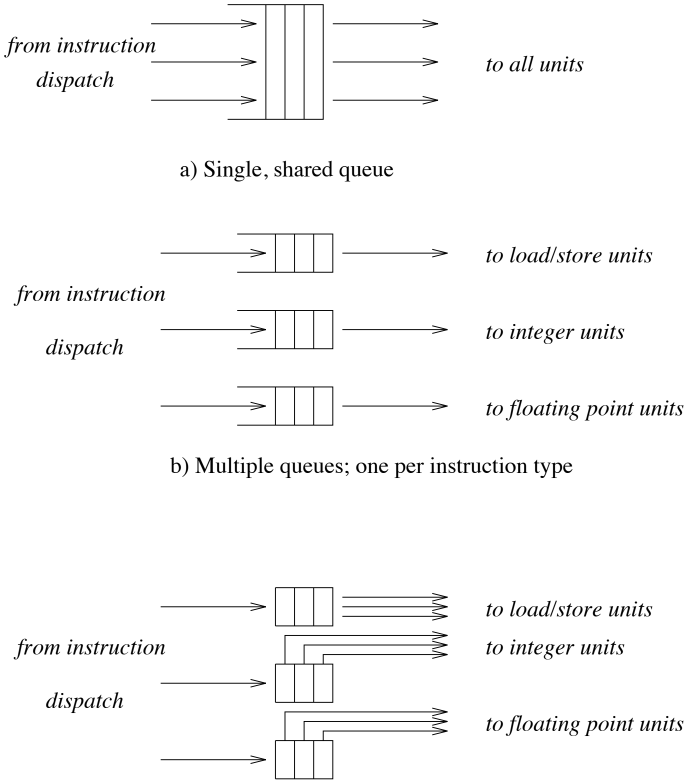
Dispatch and Renaming

- Dependence checking must be done across instructions that are dispatched in the same cycle
- Illustration [Palacharla95]



Issue

- Issue Logic



a) Single, shared queue

b) Multiple queues; one per instruction type

c) Multiple reservation stations; one per instruction type

Fig. 9. Methods of organizing instruction issue queues.

Issue

- Wakeup logic
- Figures' source: Palacharla98, Smith&Sohi95

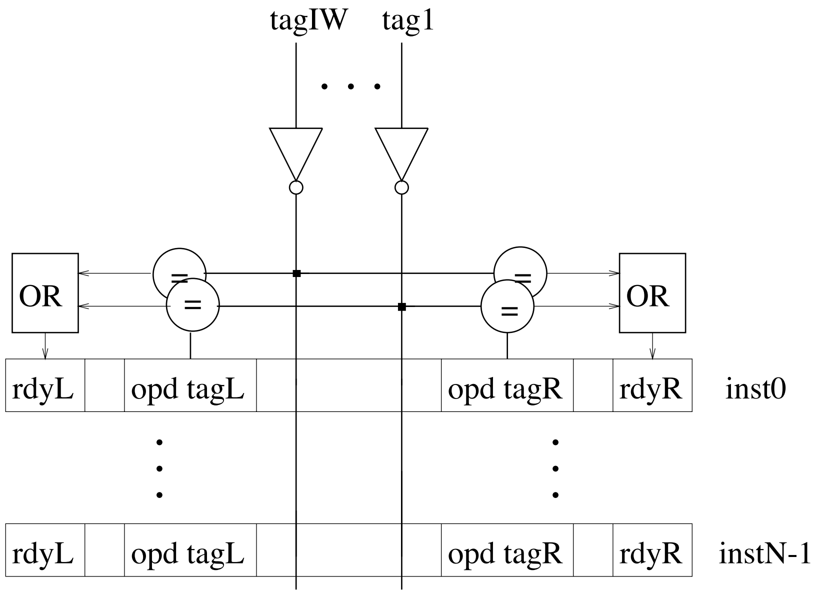


Figure 4: Wakeup logic.

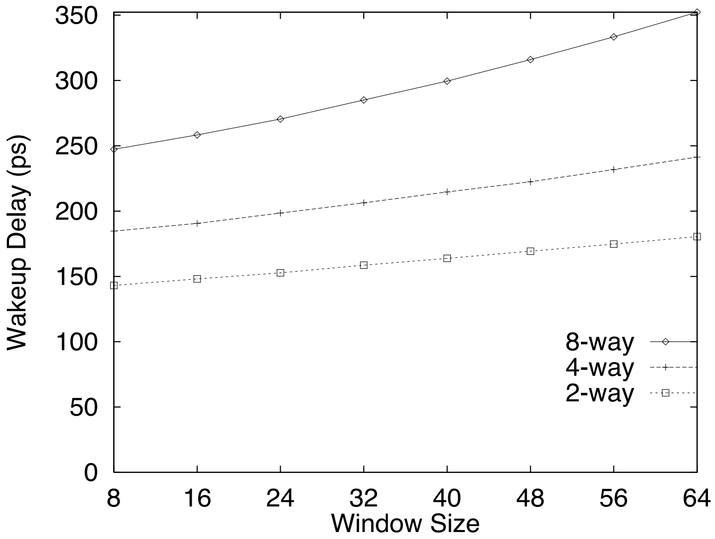


Figure 5: Wakeup logic delay versus window size.

Issue

- Select logic

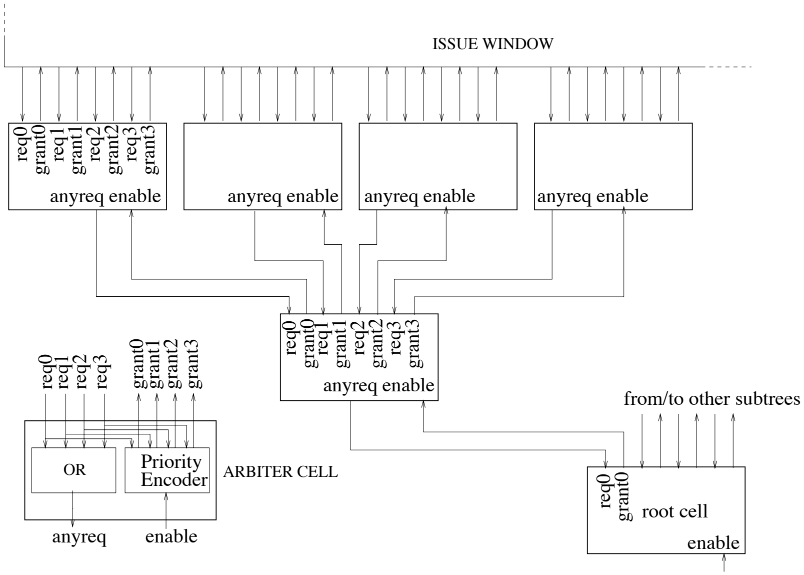


Figure 7: Selection logic.

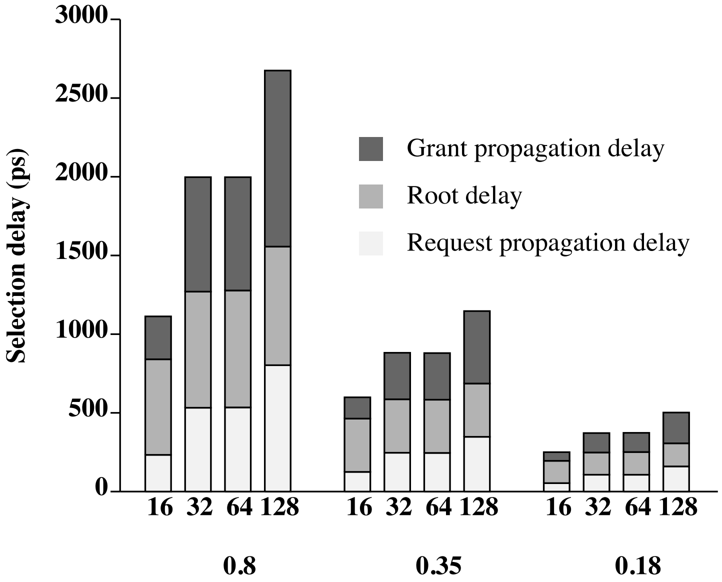


Figure 8: Selection delay versus window size.

Execute

- Multiple functional units are needed, more units increase the bypass network complexity
- Alternative: pipeline the functional unit

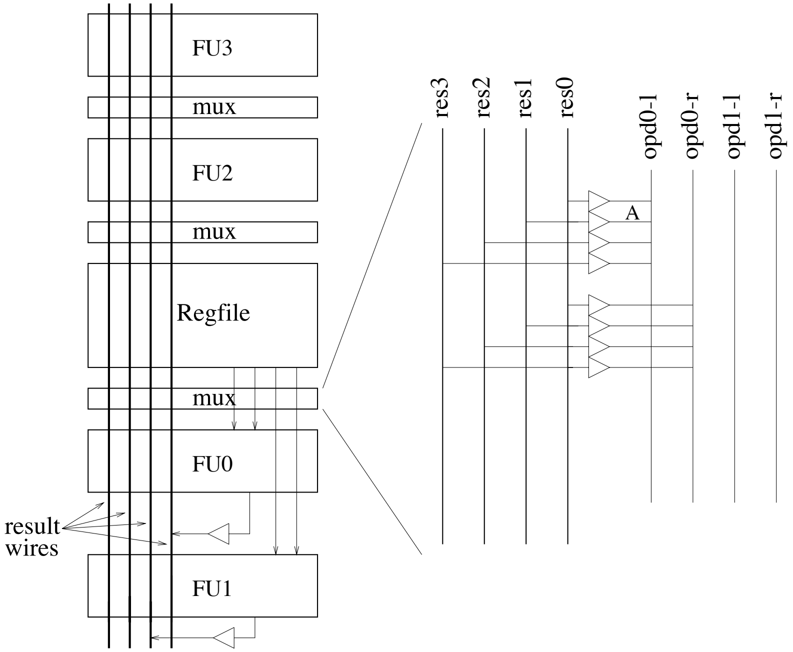


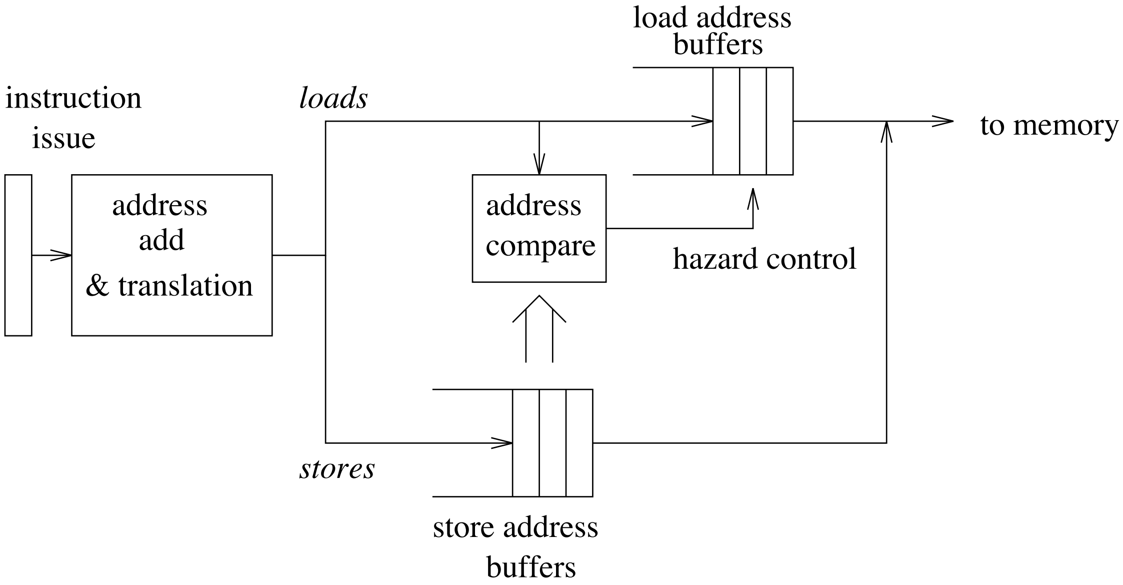
Figure 9: Bypass logic.

Issue width	Wire length (λ)	Delay (ps)
4	20500	184.9
8	49000	1056.4

Table 1: Bypass delays for a 4-way and a 8-way processor.

Retirement of Load vs. Store

- Recall Memory dependences
- Store Queue
- Load Queue



Module 6.6: ILP 6 - Compilation for Superscalar Execution

Objectives and Outline

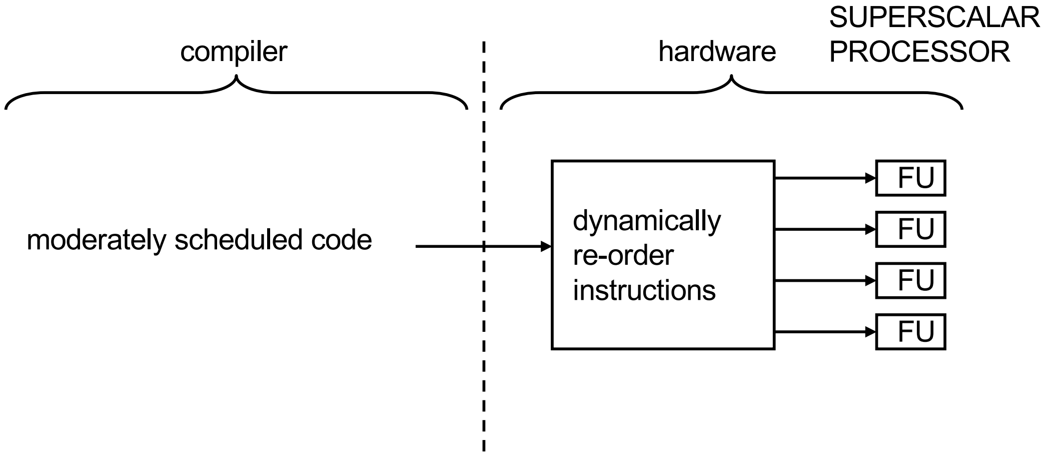
- Objectives:
  - Understand static vs. dynamic instruction scheduling
  - Understand local and global scheduling
  - Understand various optimizations that help ILP
- Outline:
  - Differences between static and dynamic scheduling and how they interact
  - Local vs. global scheduling
  - Loop unrolling, software pipelining, trace scheduling, and predication

Static Scheduling

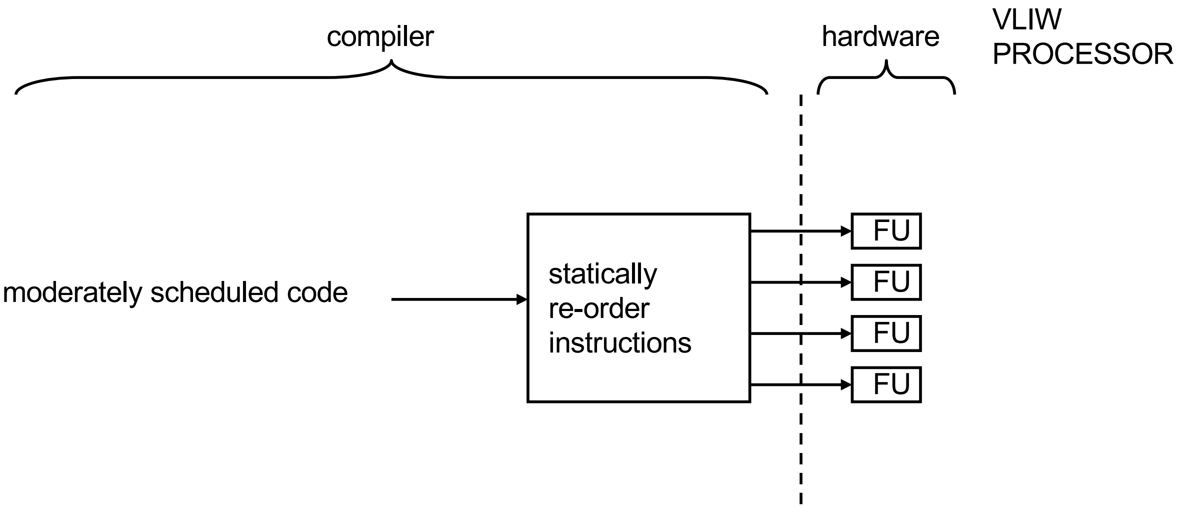
- Have compiler re-order code to improve performance
- Concepts covered
  - A comparison of static and dynamic scheduling
    - Each has advantages/disadvantages
    - Bottom line: find good combination of the two
  - Support for high-performance static scheduling
    - Register pressure: large architectural 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 = basic block scheduling
  - (+) simple: no speculation (no region formation)
  - (-) limited parallelism (usually < 2)
- 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
```

```
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 (e.g., four times)

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

```
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
```

More registers needed!

```
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 iterations
(3.5 cycles/iteration)

Benefits:
- Less dynamic instruction overhead (SUB/BNEZ)
- Less branches => larger basic block: can reschedule operations easier

Loop Unrolling

- Positives/negatives
  - (+) larger block for scheduling
  - (+) reduces branch frequency
  - (+) reduces dynamic instruction count (loop overhead)
  - (-) expands code size
  - (-) have to handle excess iterations ("strip mining")

Software Pipelining

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

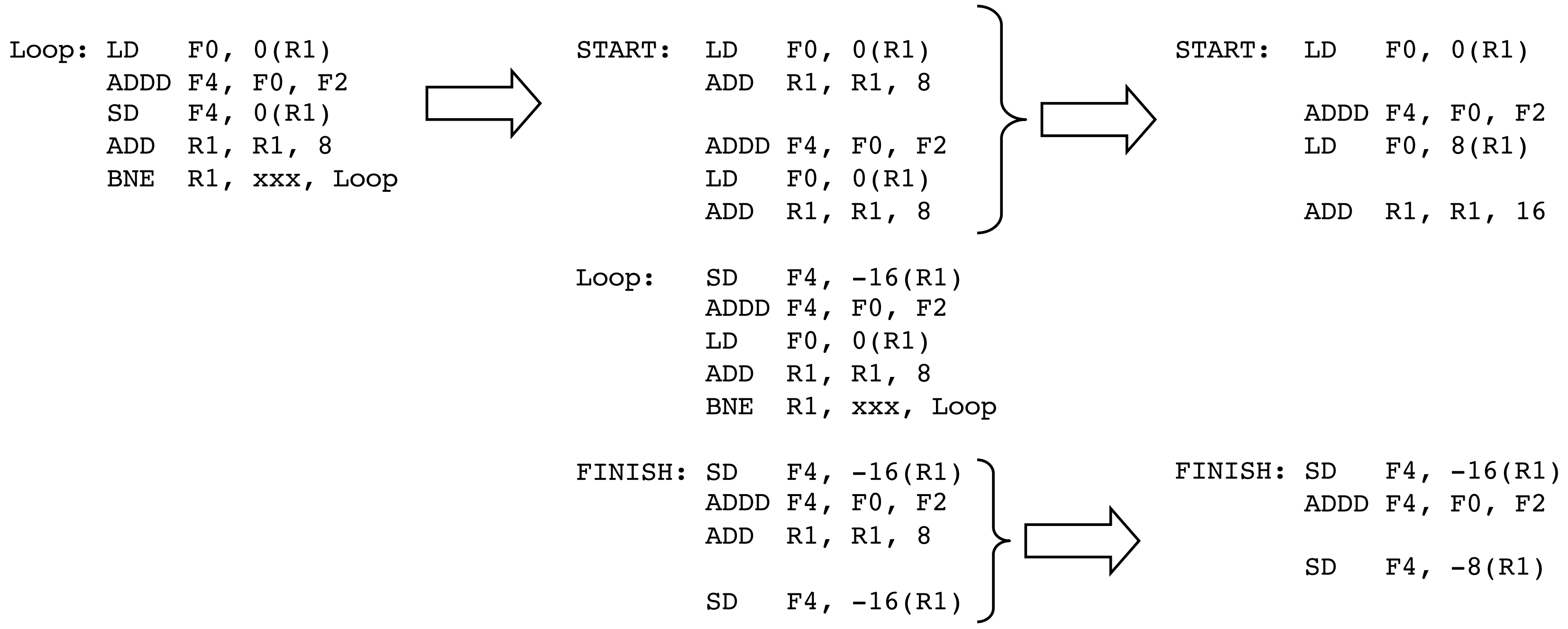
```
for (i=1; i<=N; i++) {
   // a[i] = a[i] + k;
   load a[i]
   add a[i]
   store a[i]
}
```

```
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[1]	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

- 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

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)
```

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

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

        ADD  R1, R1, 16
```

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

        SD   F4, -8(R1)
```

Software Pipelining

- Positives/negatives
  - (+) no dependences in loop body
  - (+) same effect as loop unrolling (hide latencies), but don’t need to replicate iterations (code size)
  - (-) still have extra code for prologue/epilogue (pipeline fill/drain)
  - (-) does not reduce branch frequency

Trace Scheduling

- Select most common path – a trace
  - Use profiling to select a trace
  - Allows global scheduling, i.e., scheduling across branches
  - This is speculation because schedule assumes certain path through region
  - If trace is wrong (other paths taken), execute repair code

Original code

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

Trace to be scheduled

```
   b[i] = `old’
   a[i] =
   b[i] = `new’
   c[i] =
   if (a[i] <= 0) goto A
B:
```

Repair code

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

Predication

- 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[i] = `old’
a[i] =
if (a[i] > 0) then
   b[i] = `new’    // common case
else
   X
c[i] =
```

Predicated code

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

Predication

- Positives/negatives
  - (+) Larger scheduling scope *without* speculation
  - (-) 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, treegions, …
    - H/W: selective multi-path execution, control independence, …