

CMPEN 431

Computer Architecture

Fall 2022

The Dynamic SuperScalar (OOO) Processor

Kiwan Maeng

[Adapted from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, MK

With additional thanks/credits to Amir Roth, Milo Martin, Onur Mutlu

Tomasulo's Algorithm example adapted from GeorgiaTech's HPC course:
https://www.youtube.com/playlist?list=PLAwTw4SYaPnhRXZ6wuHnnclMLfg_yjHs

Review: Multiple Instruction Issue Possibilities

- ❑ Fetch and issue **more than one** instruction in a cycle

1. **Statically-scheduled (in-order)**

- ❑ **Very Long Instruction Word (VLIW)** e.g., TransMeta (4-wide)
 - Compiler figures out what can be done in parallel, so the hardware can be dumb and low power
 - Compiler must group parallel instr's, requires new binaries
- ❑ **SuperScalar** e.g., Pentium (2-wide), ARM CortexA8 (2-wide)
 - Hardware figures out what can be done in parallel
 - Executes unmodified sequential programs
- ❑ **Explicitly Parallel Instruction Computing (EPIC)** e.g., Intel Itanium (6-wide)
 - A compromise: compiler does some, hardware does the rest

2. **Dynamically-scheduled (out-of-order) SuperScalar**

- ❑ Hardware dynamically determines what can be done in parallel (can extract much more ILP with OOO processing)
- ❑ E.g., Intel Pentium Pro/II/III (3-wide), Core i7 (4 cores, 4-wide, SMT2), IBM Power5 (5-wide), Power8 (12 cores, 8-wide, SMT8)

Why Out-of-Order (OOO) Execution?

- ❑ Consider the following instruction sequence:

I1

I2

I3

- ❑ If we do not employ OOO execution and I2 is stalled (if, for example, it depends on I1), I3 will be stalled as well
- ❑ An instruction (I3) is stalled because of an irrelevant instruction (I2)
 - ❑ A consequence of **in-order** execution
- ❑ **Solution:** Let I3 get scheduled and execute while I2 is waiting – **out-of-order (OOO)** execution
 - ❑ Improves utilization and performance

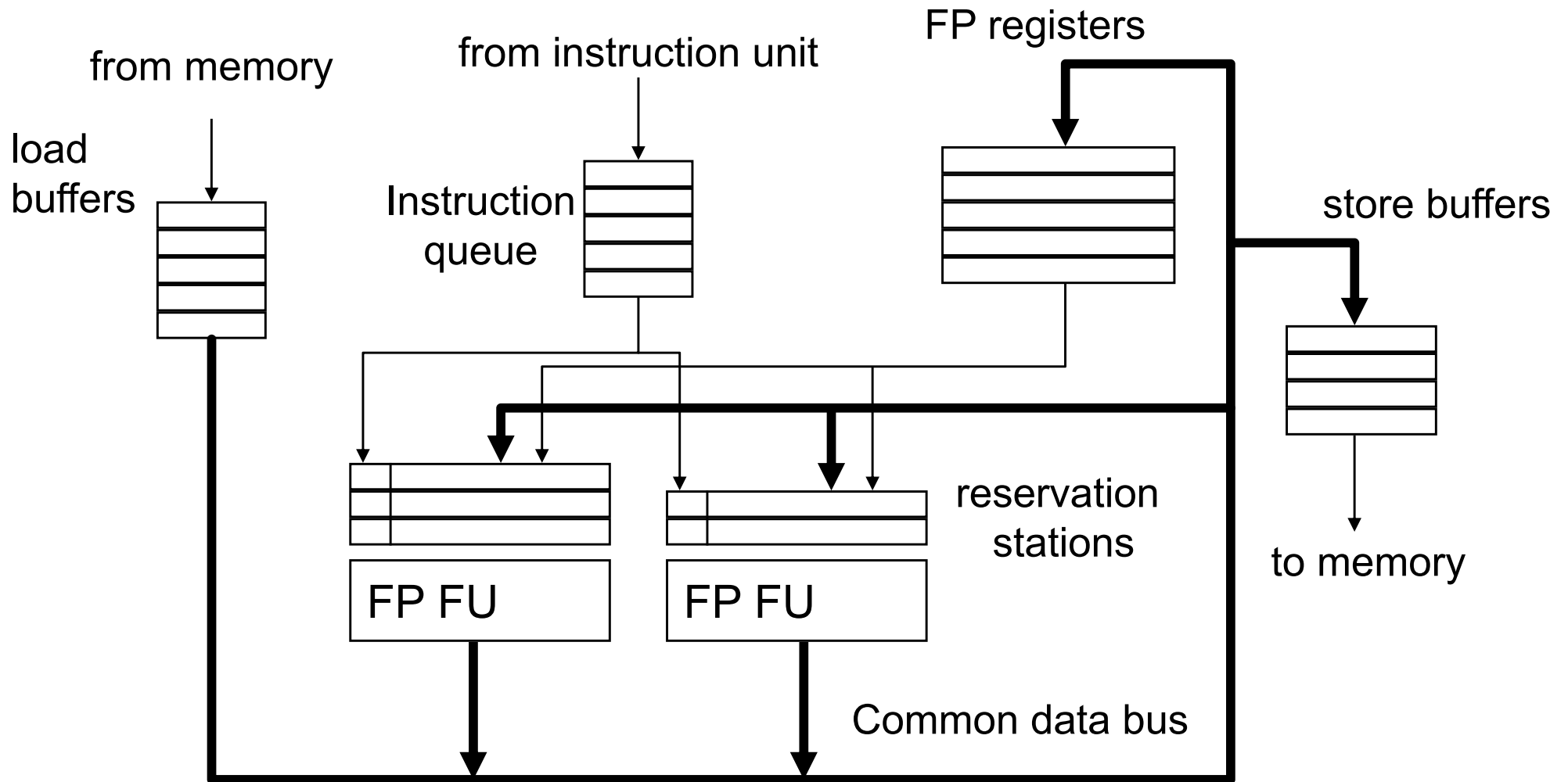
Tomasulo's Algorithm

- ❑ First publication in 1967
- ❑ Used in IBM 360/91
- ❑ Supports:
 - ❑ Out-of-order program execution
 - ❑ Dynamic register renaming
 - ❑ Originally only for floating point instructions
 - Today we will look at all instructions instead
- ❑ Does not support:
 - ❑ Precise exception handling
- ❑ I will show how it is done on a single issue width, but it can be generalized to multiple issue width.

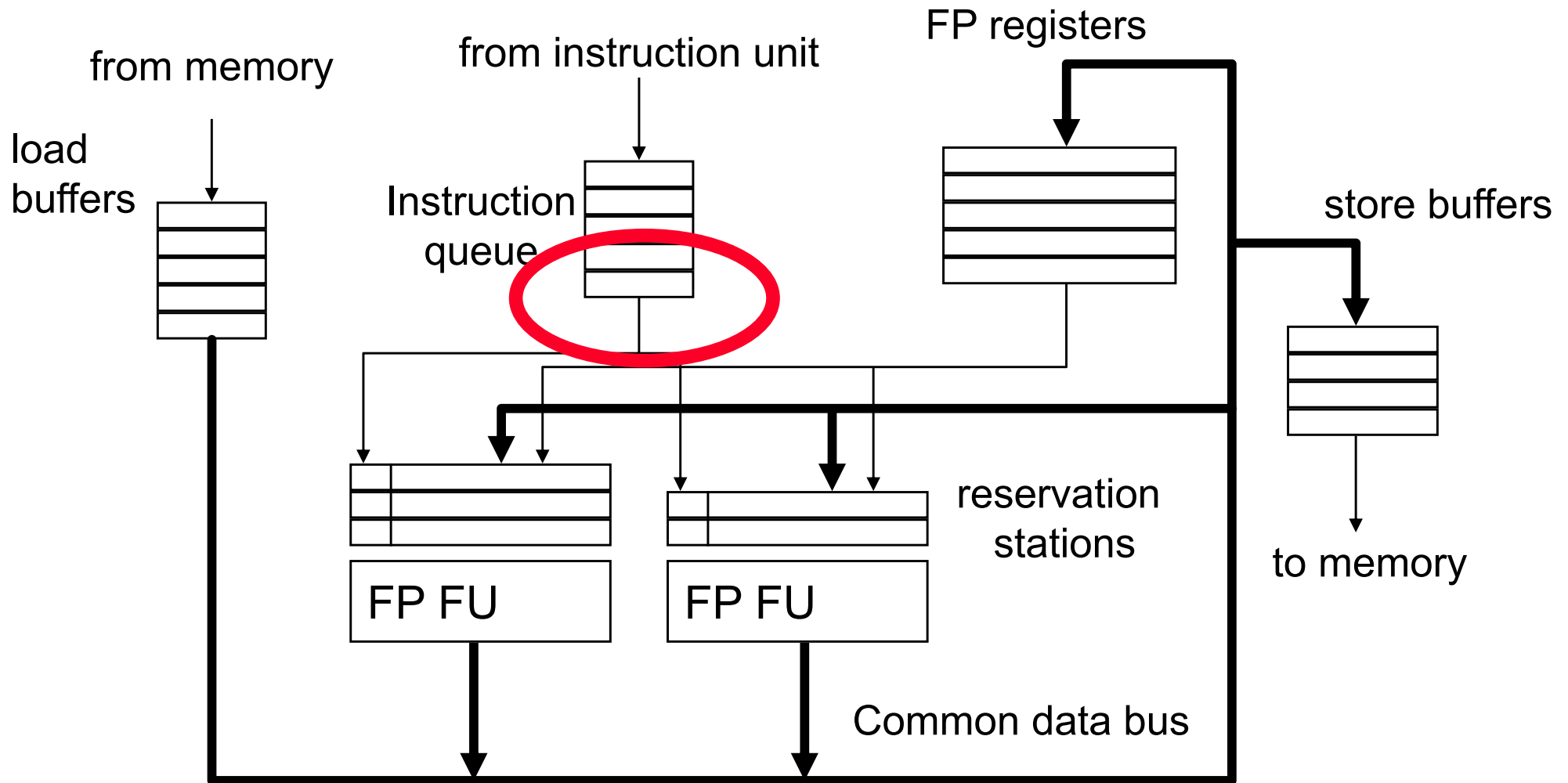
Tomasulo's Algorithm: Steps

- ❑ On each cycle, you:
 - ❑ Issue
 - ❑ Dispatch
 - ❑ Broadcast / writeback
 - ❑ (Commit)

Tomasulo's Algorithm



Tomasulo's Algorithm: Issue



Tomasulo's Algorithm: Issue

- ❑ Take an instruction in order from the instruction queue
- ❑ Rename the registers
- ❑ Place the instruction on the reservation station
- ❑ Update the register alias table (RAT)

Issue (Slightly Incorrect for Easy Understanding)

add \$1, \$2, \$3
sub \$4, \$1, \$2
div \$1, \$2, \$3
add \$2, \$4, \$1

Instruction
queue

\$1	3
\$2	-1
\$3	2
\$4	7

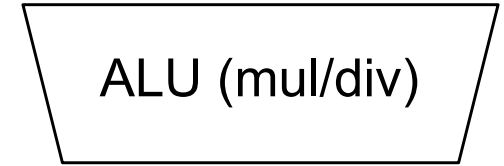
Register files

\$1	
\$2	
\$3	
\$4	

RAT

RS1	
RS2	
RS3	

RS4	
RS5	



Reservation station (RS)

Issue (Slightly Incorrect for Easy Understanding)

add \$1, \$2, \$3
sub \$4, \$1, \$2
div \$1, \$2, \$3
add \$2, \$4, \$1

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

- Place instructions in-order to the correct RS

\$1	
\$2	
\$3	
\$4	

RAT

RS1	
RS2	
RS3	

ALU (add/sub)

RS4	
RS5	

ALU (mul/div)

Issue (Slightly Incorrect for Easy Understanding)

....
add \$1, \$2, \$3
sub \$4, \$1, \$2
div \$1, \$2, \$3

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

- If the register values are ready, replace it with the value
- Update (rename) the destination register with RAT

\$1	
\$2	RS1
\$3	
\$4	

RAT

RS1	add, 7 , 3
RS2	
RS3	

ALU (add/sub)

RS4	
RS5	

ALU (mul/div)

Issue (Slightly Incorrect for Easy Understanding)

....
....
add \$1, \$2, \$3
sub \$4, \$1, \$2

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

- If the register value is in RAT, use RAT for renaming

\$1	RS4
\$2	RS1
\$3	
\$4	

RAT

RS1	add, 7, 3
RS2	
RS3	

ALU (add/sub)

RS4	div, RS1 , 2
RS5	

ALU (mul/div)

Issue (Slightly Incorrect for Easy Understanding)

....
....
....
add \$1, \$2, \$3

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

\$1	RS4
\$2	RS1
\$3	
\$4	RS2

RAT

RS1	add, 7, 3
RS2	sub, RS4 , RS1
RS3	

ALU (add/sub)

RS4	div, RS1, 2
RS5	

ALU (mul/div)

Issue (Slightly Incorrect for Easy Understanding)

....
....
....
....

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

\$1	RS3
\$2	RS1
\$3	
\$4	RS2

RAT

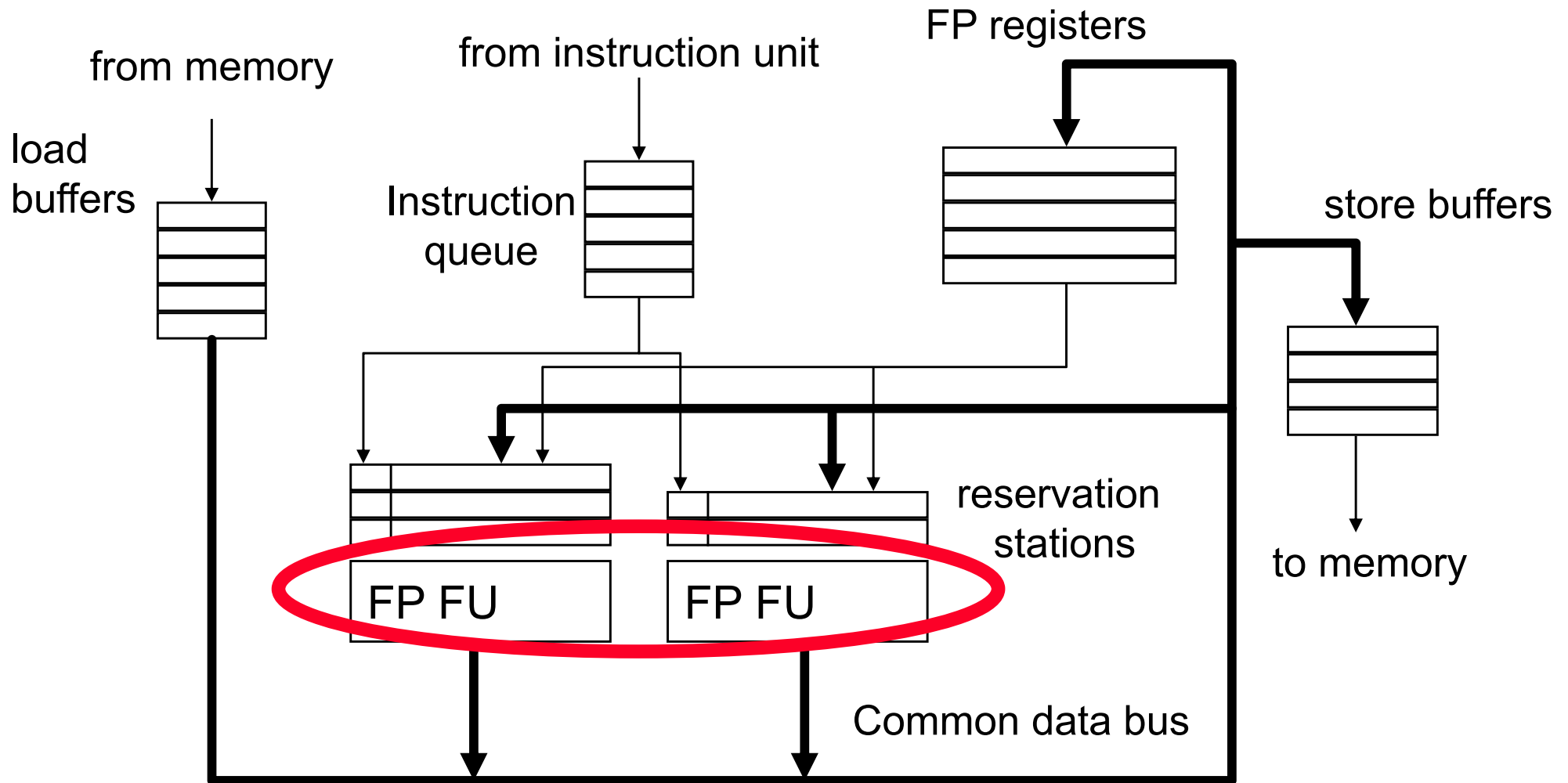
RS1	add, 7, 3
RS2	sub, RS4, RS1
RS3	add, RS1 , 2

ALU (add/sub)

RS4	div, RS1, 2
RS5	

ALU (mul/div)

Tomasulo's Algorithm: Dispatch



Tomasulo's Algorithm: Dispatch / Broadcast

- ❑ Run instructions who are ready
- ❑ Broadcast the result
- ❑ Update relevant source registers in the reservation station
- ❑ Update the register file
- ❑ Update RAT

Dispatch (Again, Slightly Incorrect)

Register files

\$1	3
\$2	-1
\$3	2
\$4	7

Ready to execute

\$1	RS3
\$2	RS1
\$3	
\$4	RS2

RAT

RS1	add, 7, 3
RS2	sub, RS4, RS1
RS3	add, RS1, 2

ALU (add/sub)

RS4	div, RS1, 2
RS5	

ALU (mul/div)

Dispatch (Again, Slightly Incorrect)

Register files

\$1	3
\$2	-1
\$3	2
\$4	7

\$1	RS3
\$2	RS1
\$3	
\$4	RS2

RAT

RS1	add, 7, 3
RS2	sub, RS4, RS1
RS3	add, RS1, 2

↓
ALU (add/sub)

RS4	div, RS1, 2
RS5	

ALU (mul/div)

- Execute ready instruction

Dispatch (Again, Slightly Incorrect)

Register files

\$1	3
\$2	-1
\$3	2
\$4	7

\$1	RS3
\$2	RS1
\$3	
\$4	RS2

RAT

RS1	
RS2	sub, RS4, RS1
RS3	add, RS1, 2

ALU (add/sub)

RS1 = 10

RS4	div, RS1, 2
RS5	

ALU (mul/div)

Dispatch (Again, Slightly Incorrect)

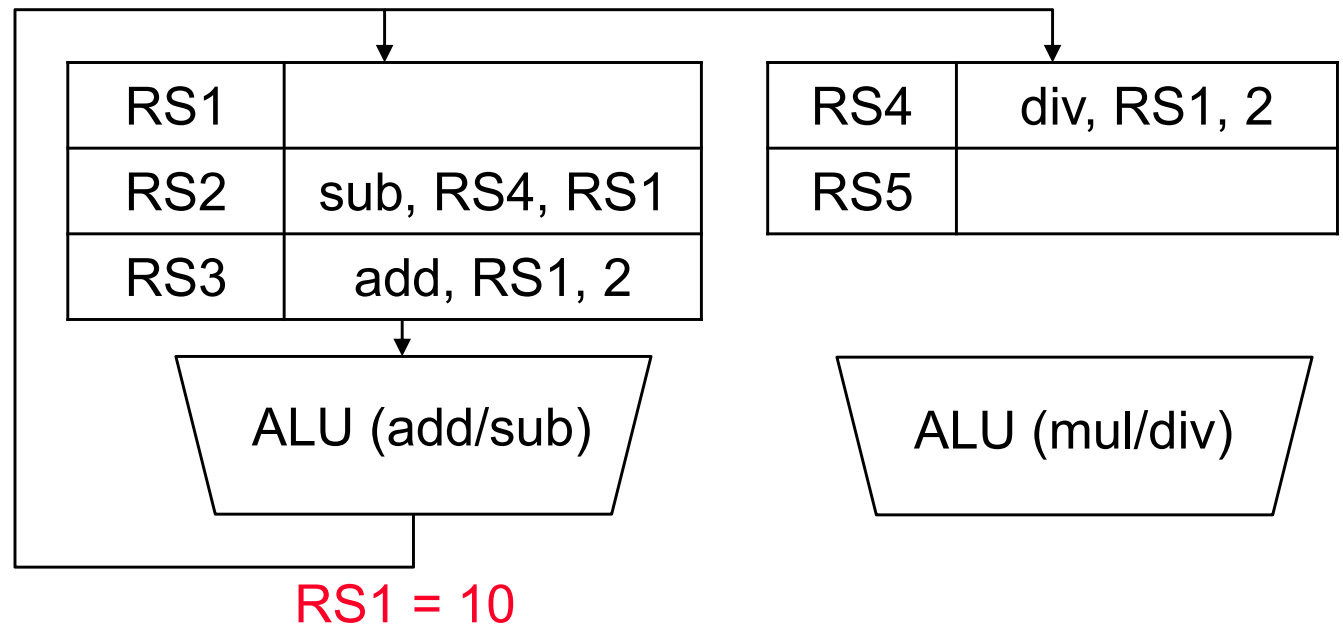
Register files

\$1	3
\$2	-1
\$3	2
\$4	7

- Broadcast the result

\$1	RS3
\$2	RS1
\$3	
\$4	RS2

RAT



Dispatch (Again, Slightly Incorrect)

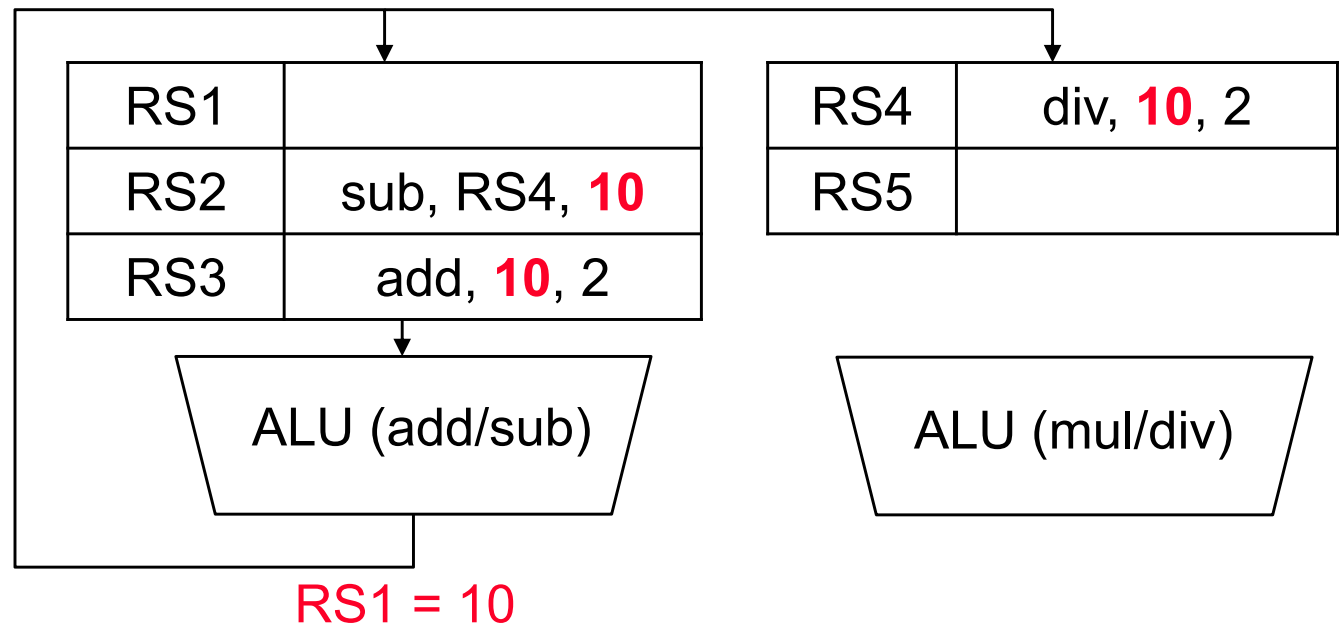
Register files

\$1	3
\$2	-1
\$3	2
\$4	7

- Broadcast the result

\$1	RS3
\$2	RS1
\$3	
\$4	RS2

RAT



Dispatch (Again, Slightly Incorrect)

Register files

\$1	3
\$2	10
\$3	2
\$4	7

- Update the reg file (by looking at the RAT)

\$1	RS3
\$2	RS1
\$3	
\$4	RS2

RAT

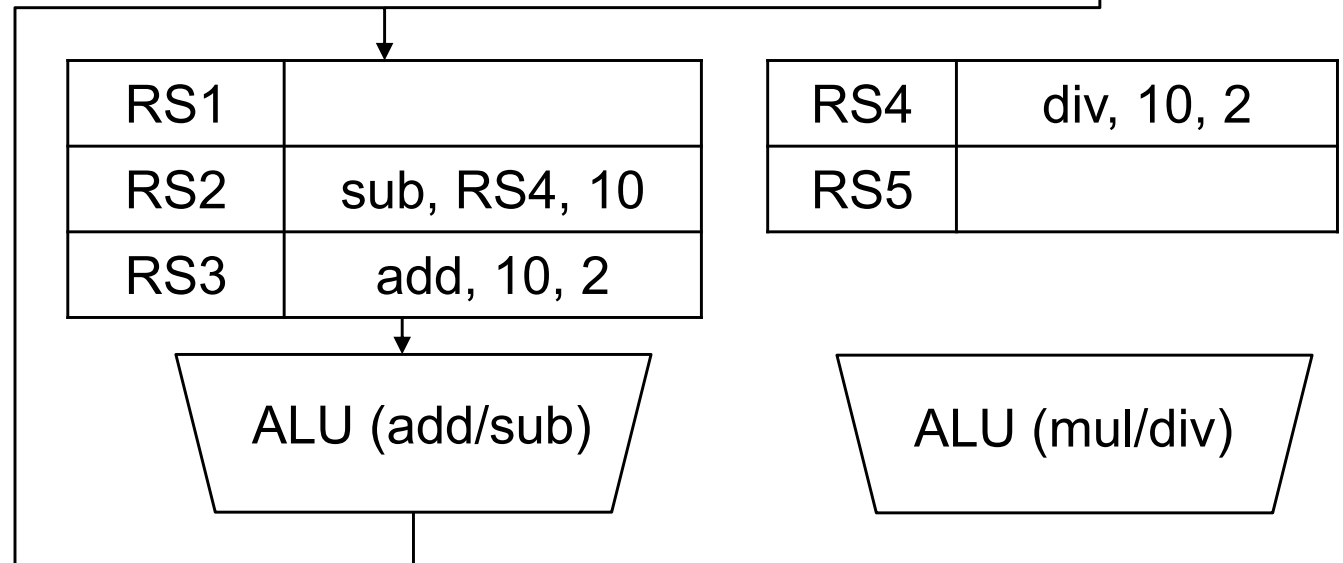
RS1	
RS2	sub, RS4, 10
RS3	add, 10, 2

ALU (add/sub)

RS1 = 10

RS4	div, 10, 2
RS5	

ALU (mul/div)



Dispatch (Again, Slightly Incorrect)

Register files

\$1	3
\$2	10
\$3	2
\$4	7

- Update (free) the RAT

\$1	RS3
\$2	
\$3	
\$4	RS2

RAT

RS1	
RS2	sub, RS4, 10
RS3	add, 10, 2

ALU (add/sub)

RS1 = 10

RS4	div, 10, 2
RS5	

ALU (mul/div)

Dispatch (Again, Slightly Incorrect)

Register files

\$1	3
\$2	10
\$3	2
\$4	7

- Can be executed out-of-order if all the regs are ready!

\$1	RS3
\$2	
\$3	
\$4	RS2

RAT

RS1	
RS2	sub, RS4, 10
RS3	add, 10, 2

ALU (add/sub)

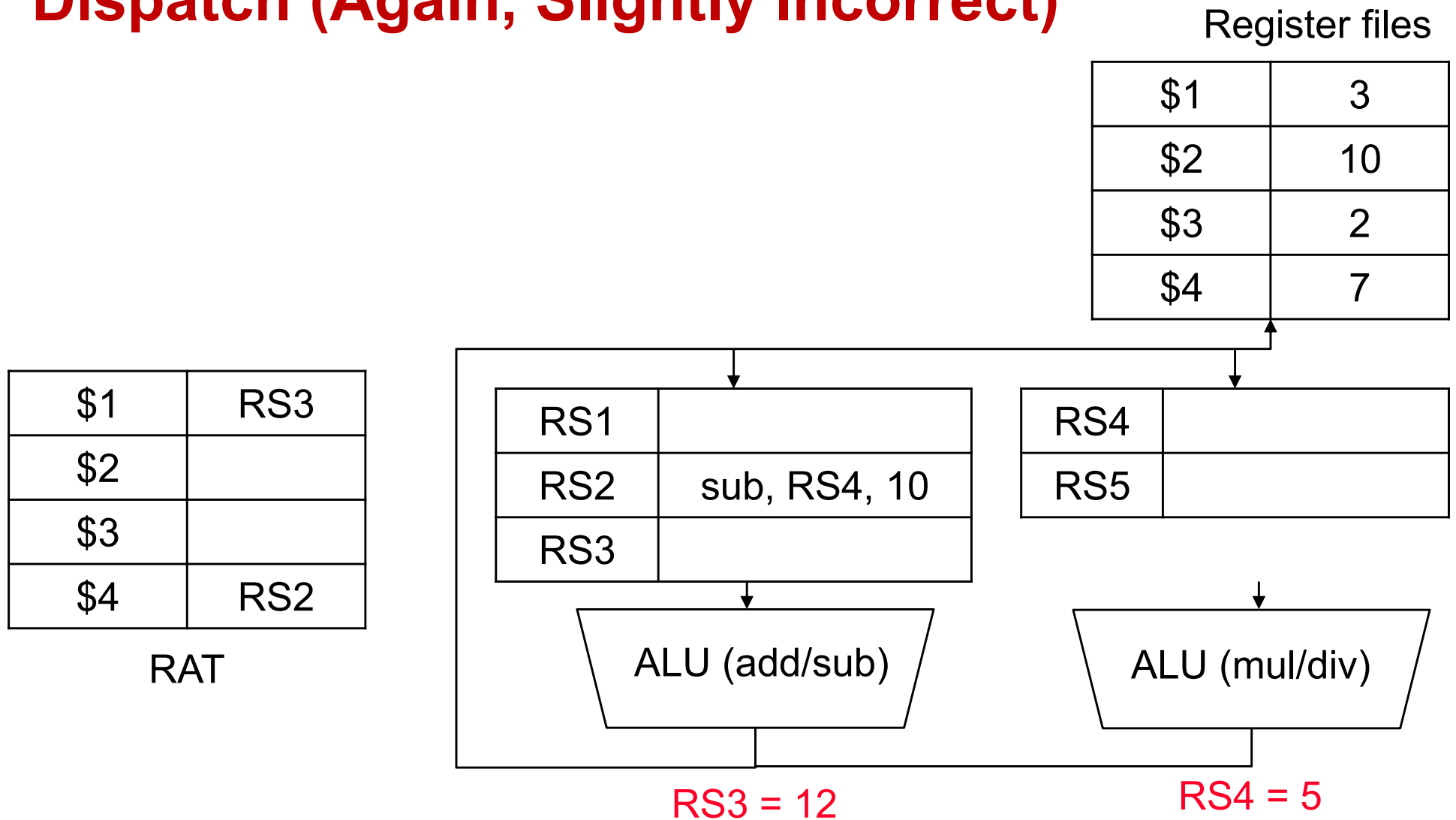
RS3 = 12

RS4	div, 10, 2
RS5	

ALU (mul/div)

RS4 = 5

Dispatch (Again, Slightly Incorrect)



Dispatch (Again, Slightly Incorrect)

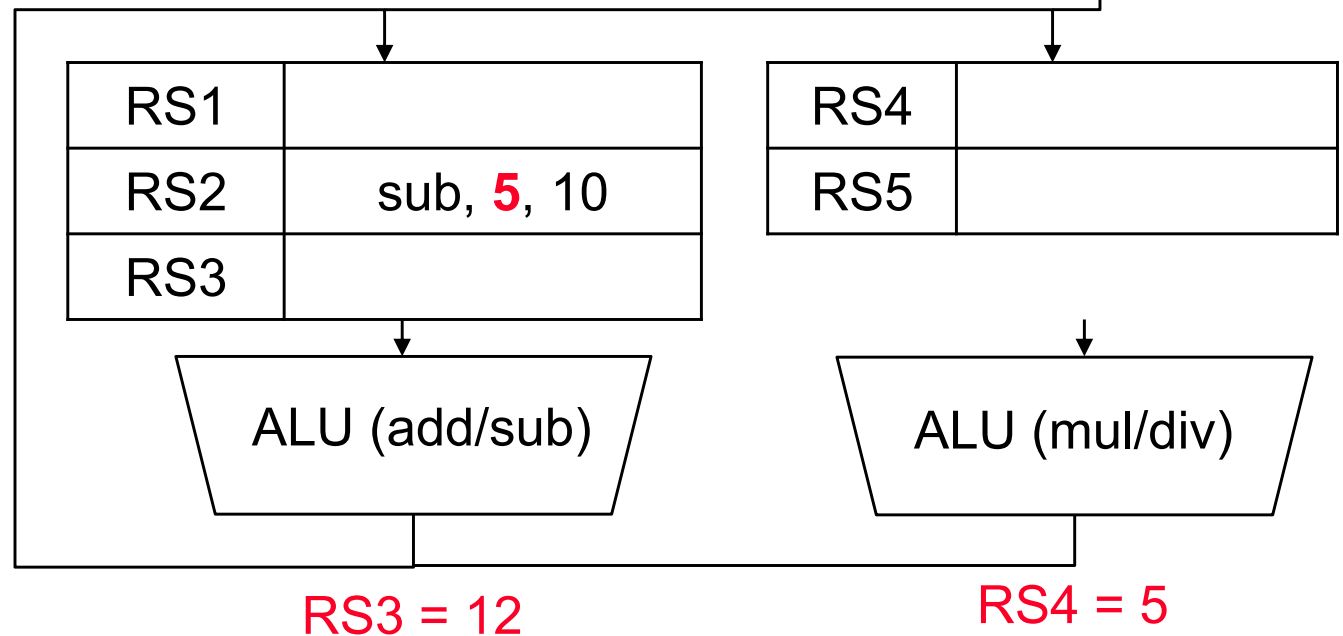
- Some results only update the RS
- Some results only update the reg file
- Some results do nothing (useless inst)

\$1	RS3
\$2	
\$3	
\$4	RS2

RAT

Register files

\$1	12
\$2	10
\$3	2
\$4	7



Dispatch (Again, Slightly Incorrect)

Register files

\$1	12
\$2	10
\$3	2
\$4	7

\$1	
\$2	
\$3	
\$4	RS2

RAT

RS1	
RS2	sub, 5, 10
RS3	

↓
ALU (add/sub)

RS4	
RS5	

↓
ALU (mul/div)

Dispatch (Again, Slightly Incorrect)

Register files

\$1	12
\$2	10
\$3	2
\$4	7

\$1	
\$2	
\$3	
\$4	RS2

RAT

RS1	
RS2	sub, 5, 10
RS3	

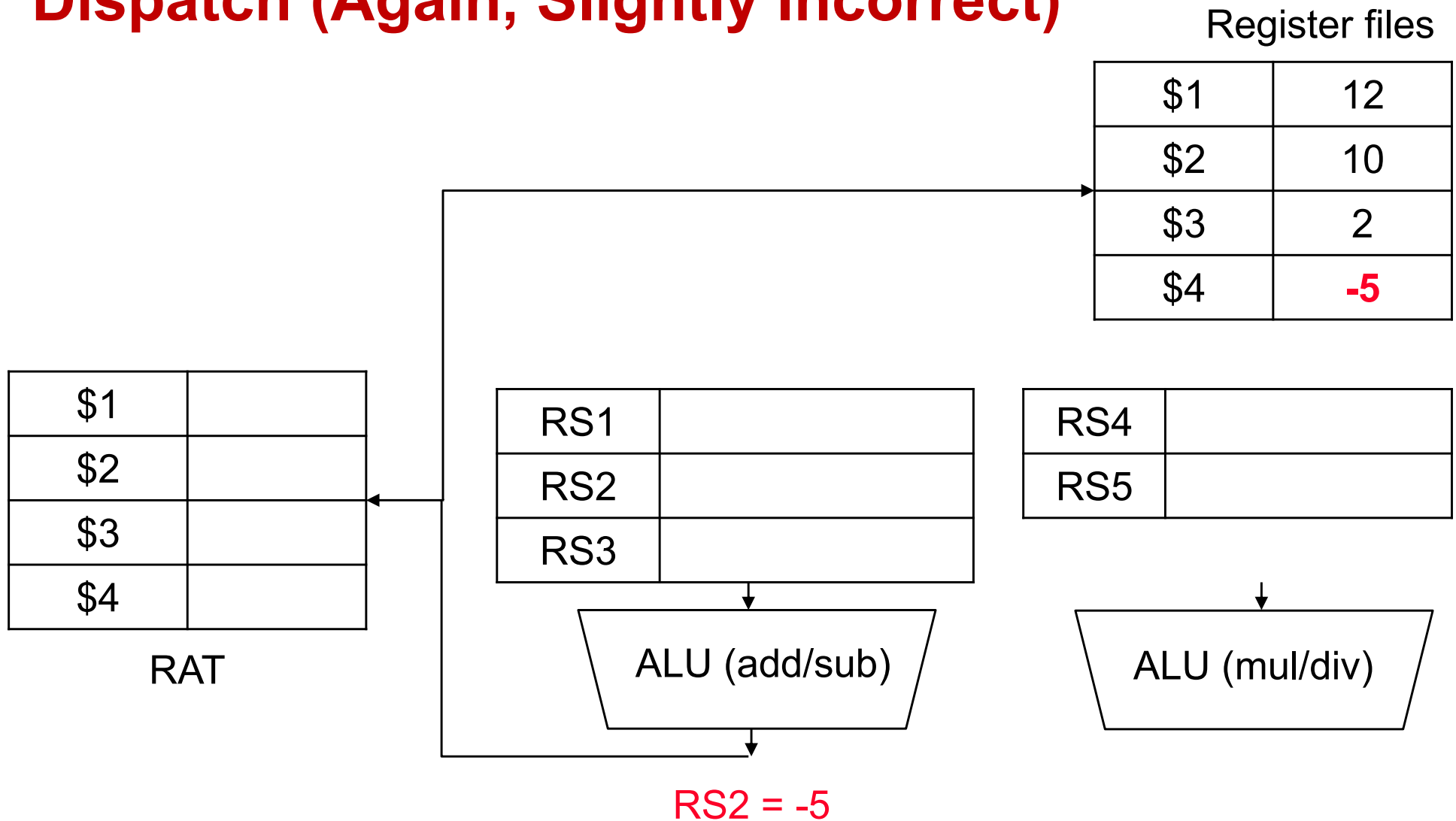
ALU (add/sub)

RS2 = -5

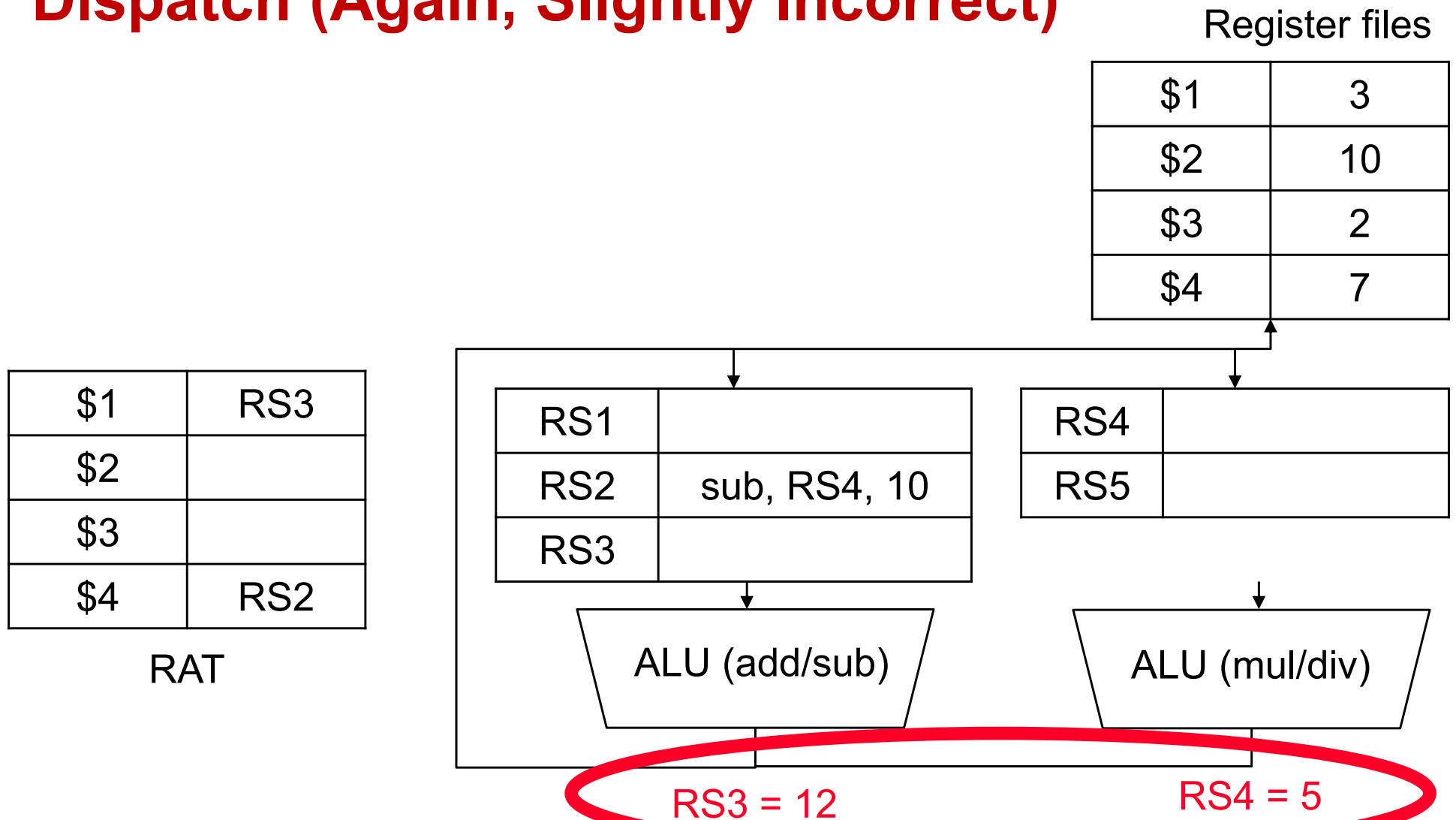
RS4	
RS5	

ALU (mul/div)

Dispatch (Again, Slightly Incorrect)

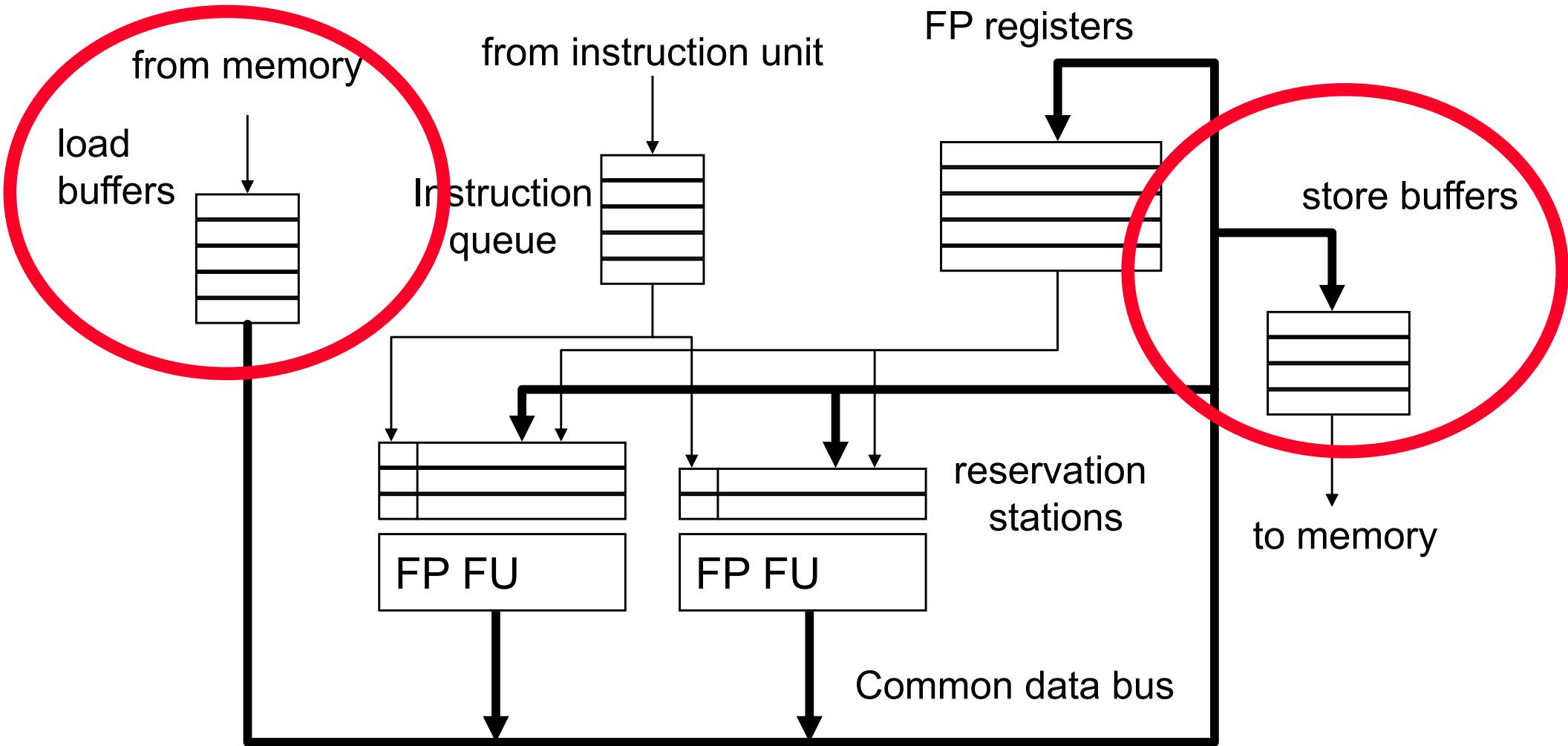


Dispatch (Again, Slightly Incorrect)



Bus can become a bottleneck!

Tomasulo's Algorithm: Load/Store



Can Load/Stores Be Reordered?

- ❑ We have to think about dependencies
 - ❑ RAW: `sw $1, 0($2) → lw $3, 0($4)`
 - ❑ WAR: `lw $1, 0($2) → sw $3, 0($4)`
 - ❑ WAW: `sw $1, 0($2) → sw $3, 0($4)`
 - ❑ Can renaming help?
- ❑ Reasonably simple approach:
 - ❑ Do not reorder (what we assume for now)
 - ❑ Load does not get reordered before store (for later)
 - ❑ Aggressive reordering and checking (for later)
 - Modern processors

Exception Dependence

- ❑ We also have to provide for **precise interrupts**, i.e., those synchronous to program (instruction) execution, to support virtual memory (TLB and/or page faults) and deal with undefined instructions, arithmetic overflow, etc.
- ❑ Any changes in instruction execution order must not change the order in which exceptions are raised, or cause new exceptions to be raised

- ❑ Example:

```
    beq $t0, $t1, L1
    lw   $t1, 0($s1)
L1:
```

- ❑ Can there be a problem with moving `lw` before `beq`?
- ❑ Original Tomasulo does not support this!

Precise Exception Support

EXCEPTION!!

Register files

\$1	12
\$2	10
\$3	2
\$4	7

\$1	
\$2	
\$3	
\$4	RS2

RAT

RS1	
RS2	sub, 5, 10
RS3	

ALU (add/sub)

RS4	
RS5	

ALU (mul/div)

Precise Exception Support

\$1	12
\$2	10
\$3	2
\$4	7

Register files after the exception

- Is this a valid program state?

add \$1, \$2, \$3
sub \$4, \$1, \$2
div \$1, \$2, \$3
add \$2, \$4, \$1

Instruction
queue

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

Solution: Reorder Buffer (ROB)

- ❑ Execute and broadcast out-of-order, but **commit in-order!**
- ❑ Hold temporary values in the **reorder buffer (ROB)** until it is okay to commit.

Issue w/ ROB

add \$1, \$2, \$3
sub \$4, \$1, \$2
div \$1, \$2, \$3
add \$2, \$4, \$1

ROB		

← commit
← issue

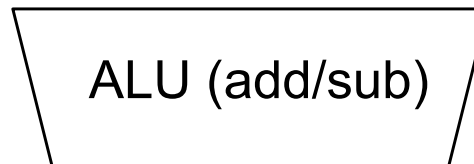
\$1	3
\$2	-1
\$3	2
\$4	7

Register files

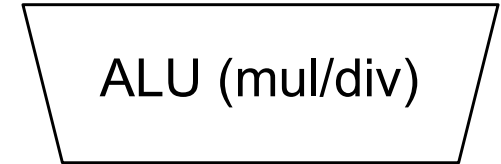
\$1	
\$2	
\$3	
\$4	

RAT

RS1	
RS2	
RS3	



RS4	
RS5	



Issue w/ ROB

add \$1, \$2, \$3
sub \$4, \$1, \$2
div \$1, \$2, \$3
add \$2, \$4, \$1

ROB		
\$2		

← commit
← issue

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

\$1	
\$2	
\$3	
\$4	

RAT

RS1	
RS2	
RS3	

ALU (add/sub)

RS4	
RS5	

ALU (mul/div)

Issue w/ ROB

....
add \$1, \$2, \$3
sub \$4, \$1, \$2
div \$1, \$2, \$3

ROB		
\$2		

← commit
← issue

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

\$1	
\$2	RS1
\$3	
\$4	

RAT

RS1	add, 7 , 3
RS2	
RS3	

ALU (add/sub)

RS4	
RS5	

ALU (mul/div)

Issue w/ ROB

....
add \$1, \$2, \$3
sub \$4, \$1, \$2
div \$1, \$2, \$3

ROB		
\$2		

← commit
← issue

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

\$1	
\$2	ROB1
\$3	
\$4	

RAT

ROB1	add, 7 , 3

ALU (add/sub)

ALU (mul/div)

Issue w/ ROB

....
....
add \$1, \$2, \$3
sub \$4, \$1, \$2

ROB		
\$2		
\$1		

← commit

← issue

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

\$1	ROB2
\$2	ROB1
\$3	
\$4	

RAT

ROB1	add, 7, 3

ALU (add/sub)

ROB2	div, ROB1 , 2

ALU (mul/div)

Issue w/ ROB

....
....
....
add \$1, \$2, \$3

ROB		
\$2		
\$1		
\$4		

← commit

← issue

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

\$1	ROB2
\$2	ROB1
\$3	
\$4	ROB3

RAT

ROB1	add, 7, 3
ROB3	sub, ROB2 , ROB1

ALU (add/sub)

ROB2	div, ROB1, 2

ALU (mul/div)

Issue w/ ROB

....
....
....
....

ROB		
\$2		
\$1		
\$4		
\$1		

← commit

← issue

\$1	3
\$2	-1
\$3	2
\$4	7

Register files

\$1	ROB4
\$2	ROB1
\$3	
\$4	ROB3

RAT

ROB1	add, 7, 3
ROB3	sub, ROB2, ROB1
ROB4	add, ROB1 , 2

ALU (add/sub)

ROB2	div, ROB1, 2

ALU (mul/div)

Dispatch w/ ROB

ROB

\$2		
\$1		
\$4		
\$1		

← commit

← issue

\$1	ROB4
\$2	ROB1
\$3	
\$4	ROB3

RAT

ROB1	add, 7, 3
ROB3	sub, ROB2, ROB1
ROB4	add, ROB1, 2

ALU (add/sub)

Register files

\$1	3
\$2	-1
\$3	2
\$4	7

ROB2	div, ROB1, 2

ALU (mul/div)

Dispatch w/ ROB

ROB

\$2		
\$1		
\$4		
\$1		

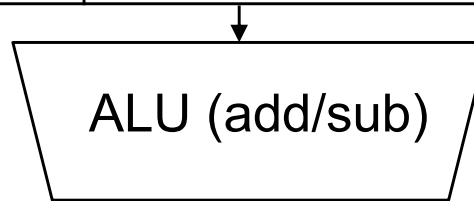
← commit

← issue

\$1	ROB4
\$2	ROB1
\$3	
\$4	ROB3

RAT

ROB1	add, 7, 3
ROB3	sub, ROB2, ROB1
ROB4	add, ROB1, 2

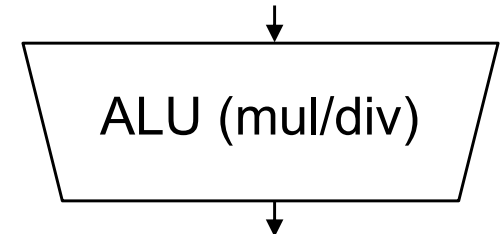


10

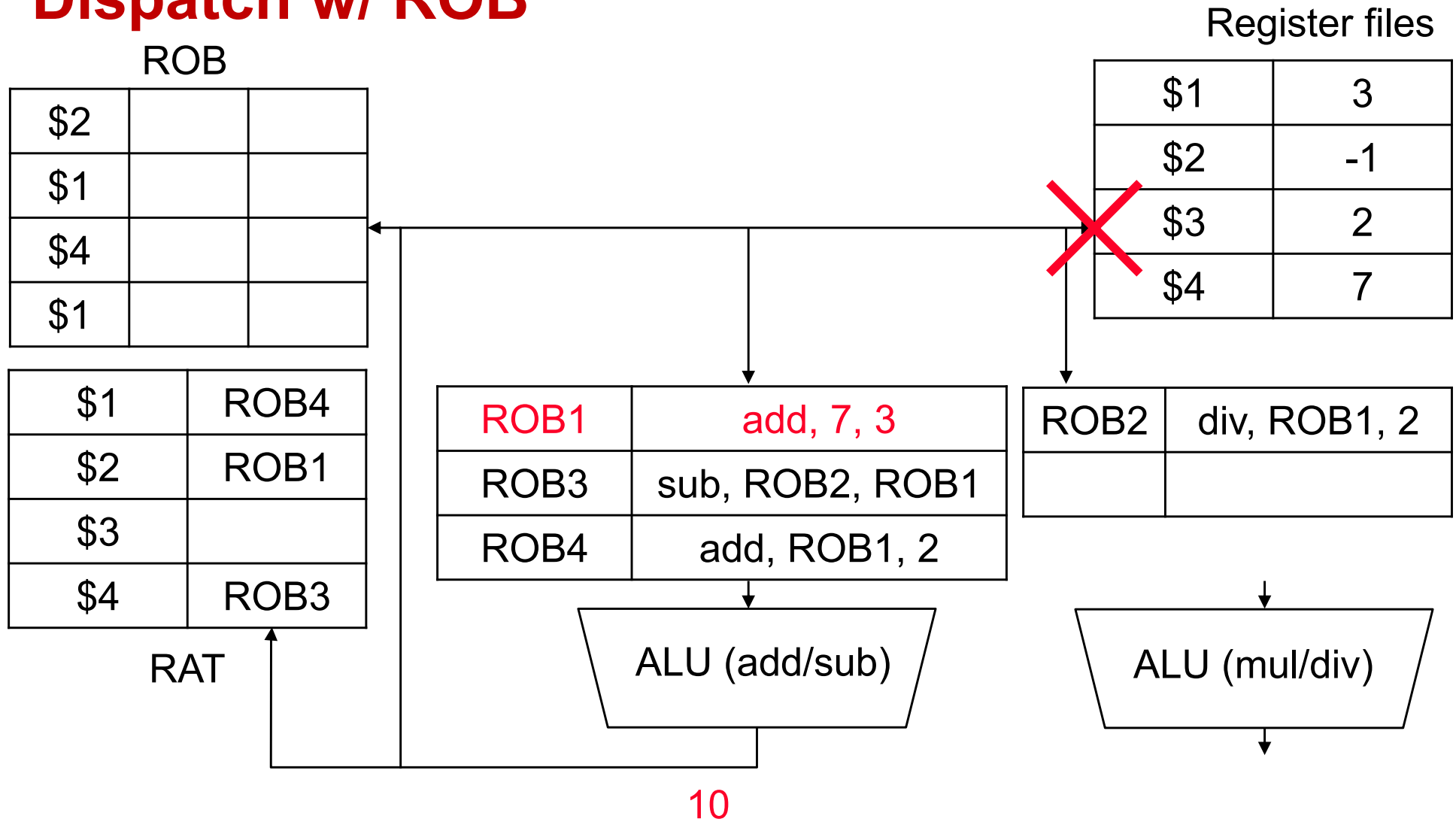
Register files

\$1	3
\$2	-1
\$3	2
\$4	7

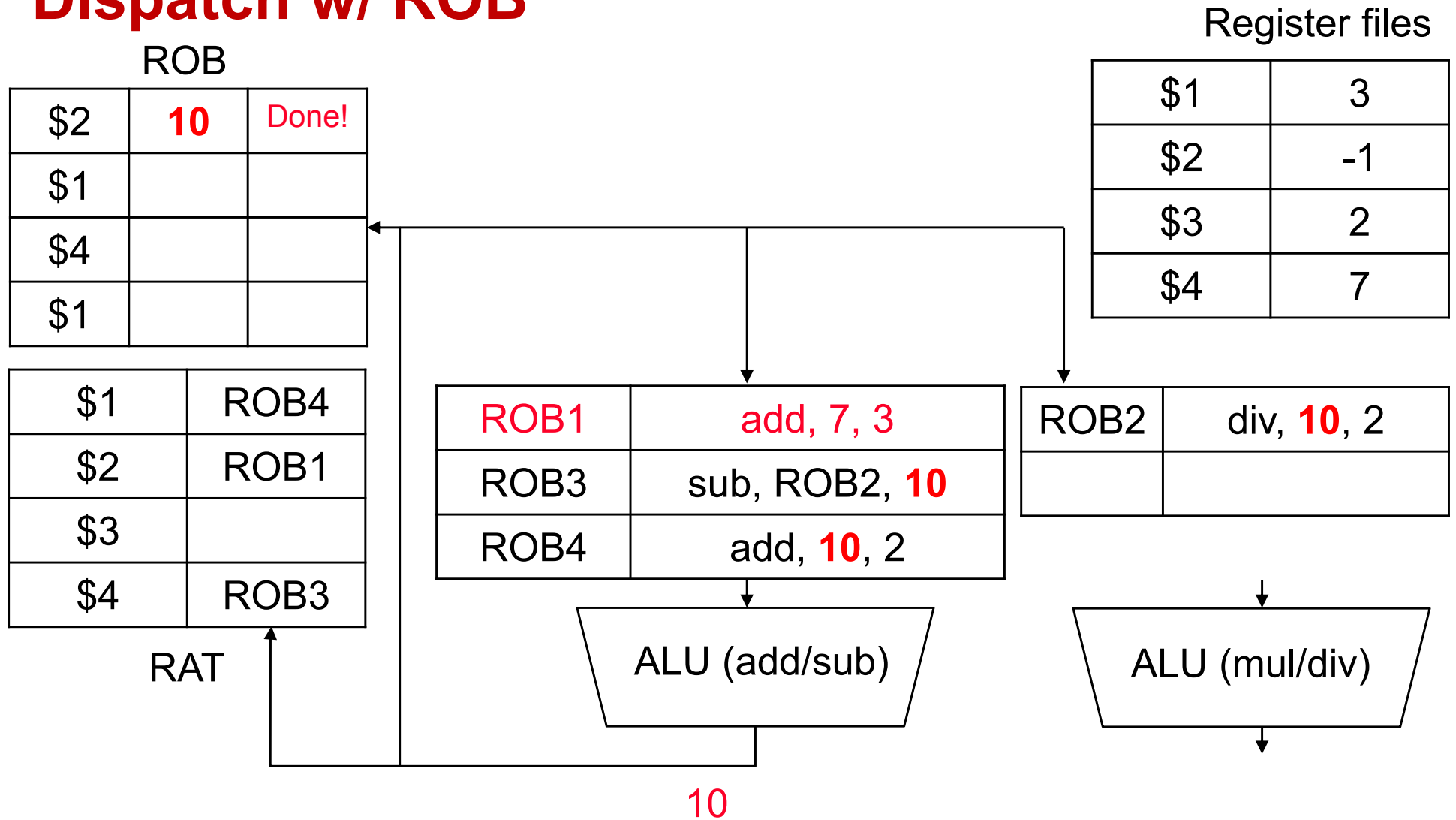
ROB2	div, ROB1, 2



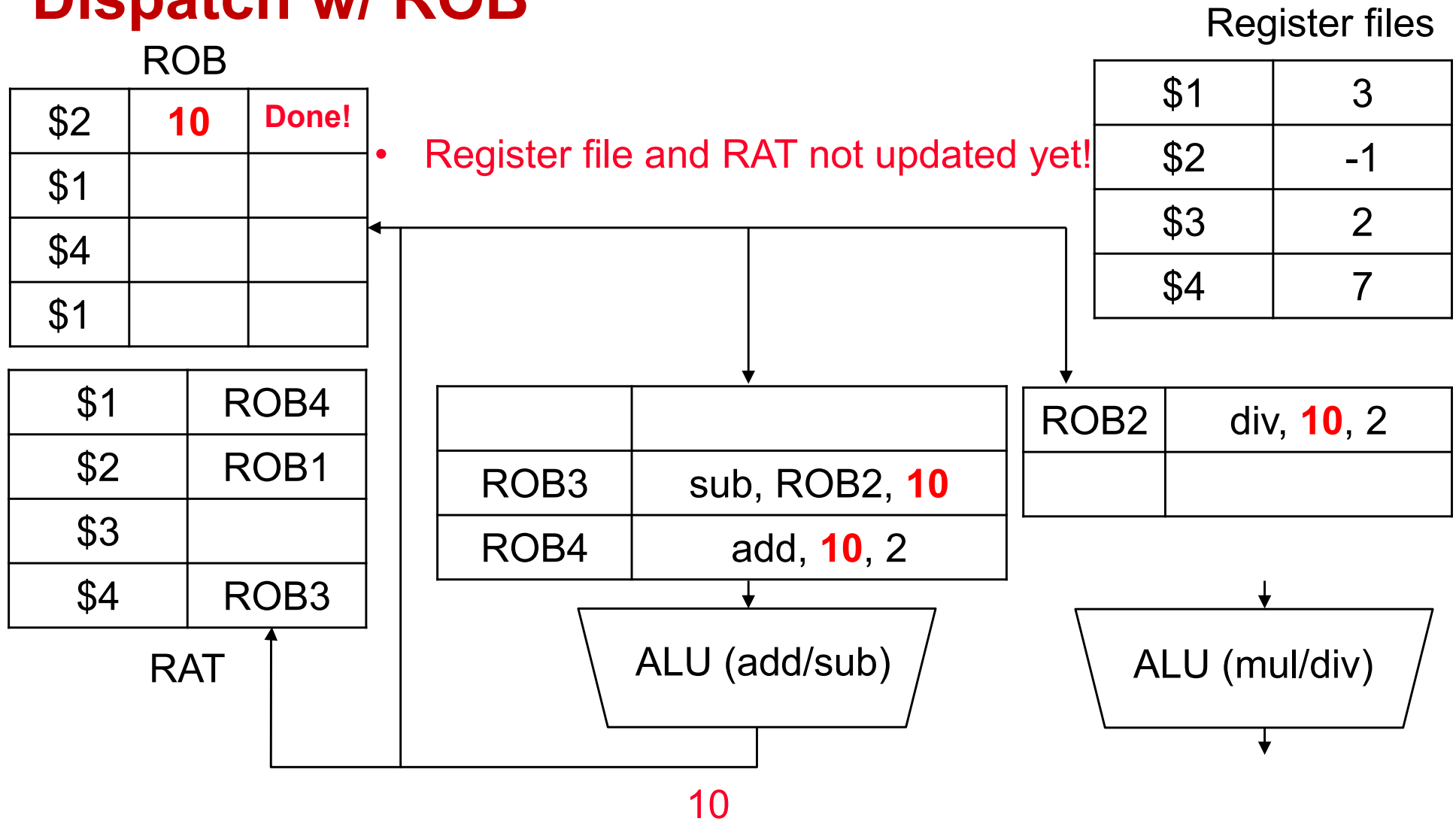
Dispatch w/ ROB



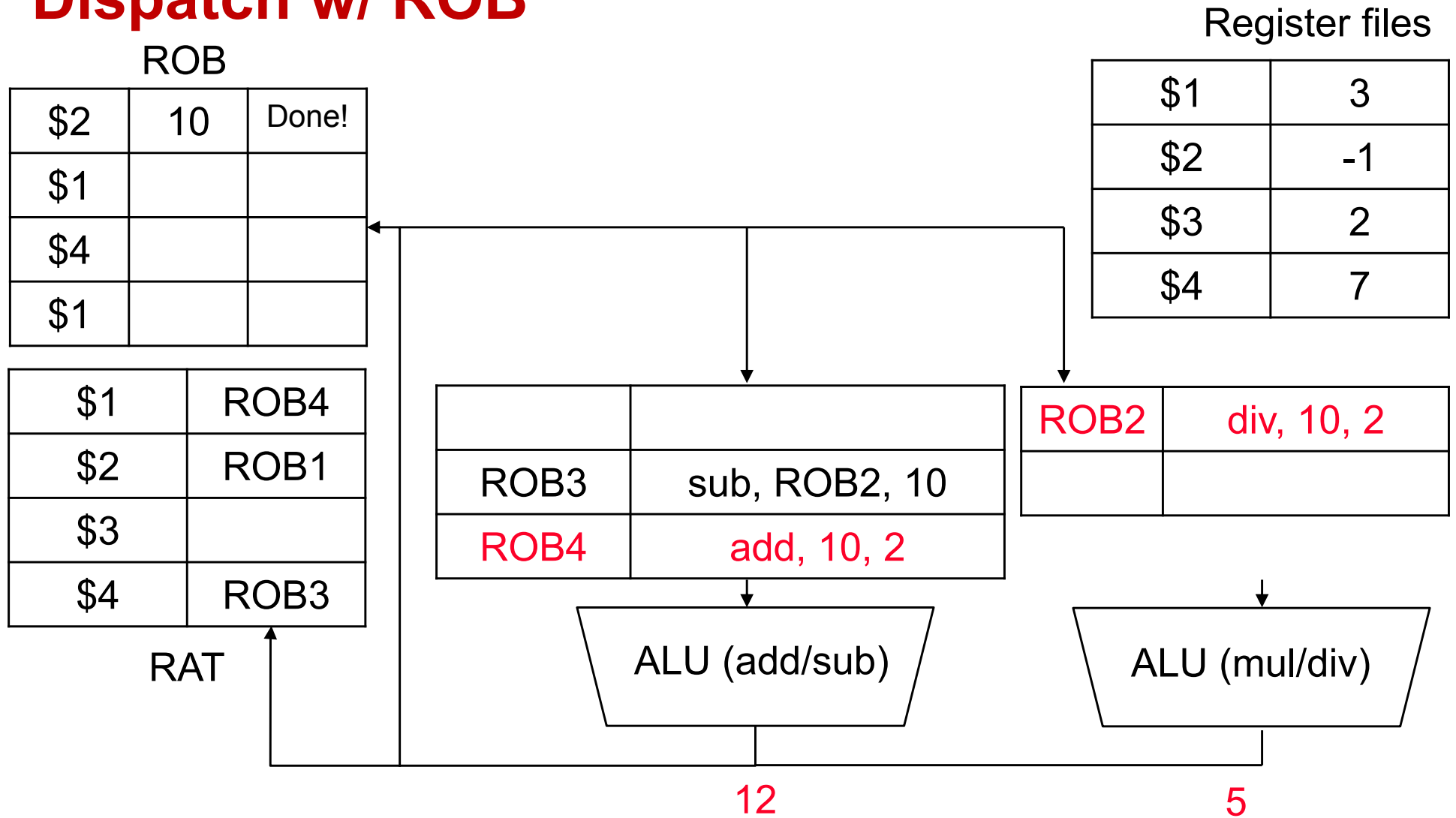
Dispatch w/ ROB



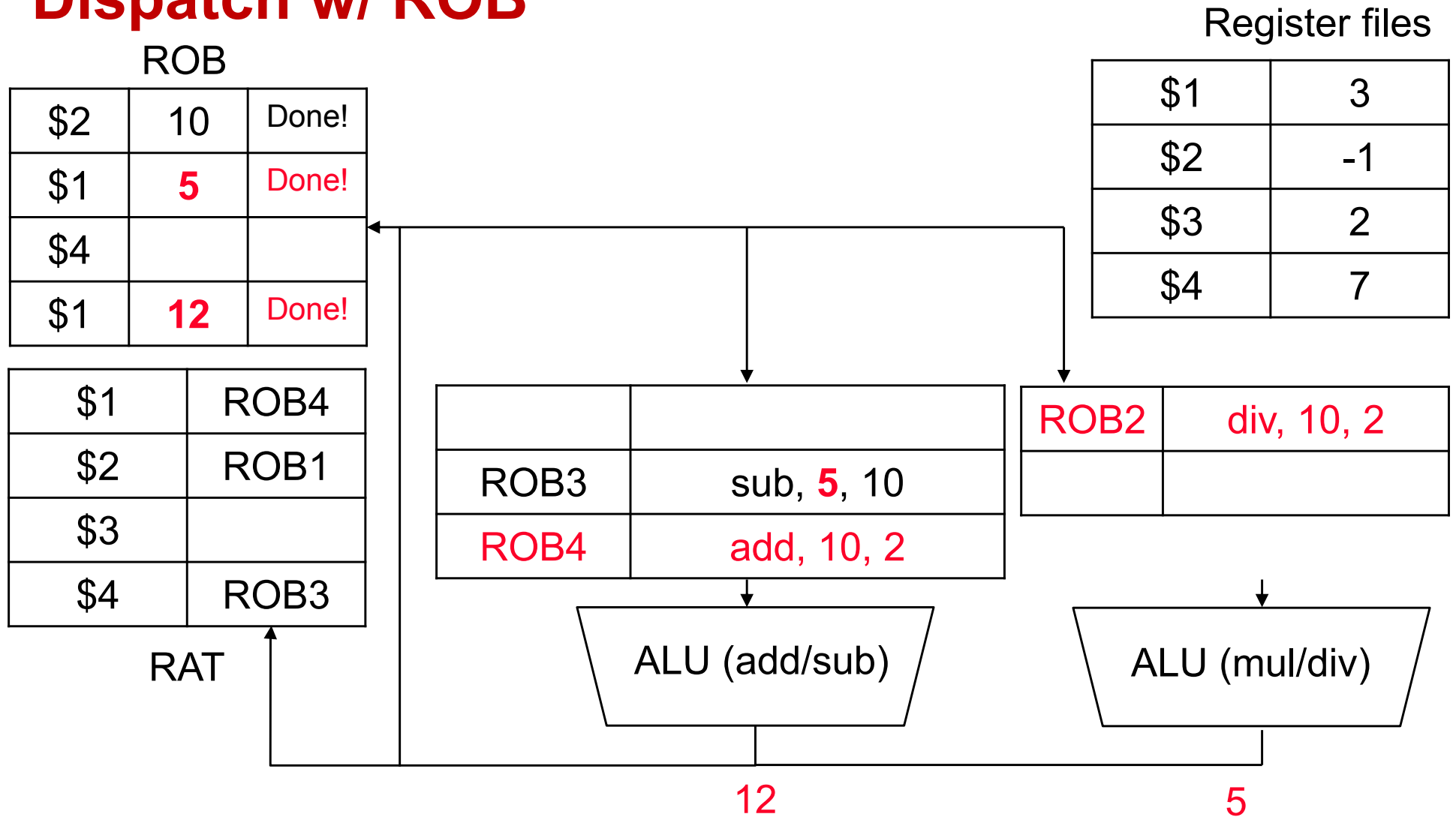
Dispatch w/ ROB



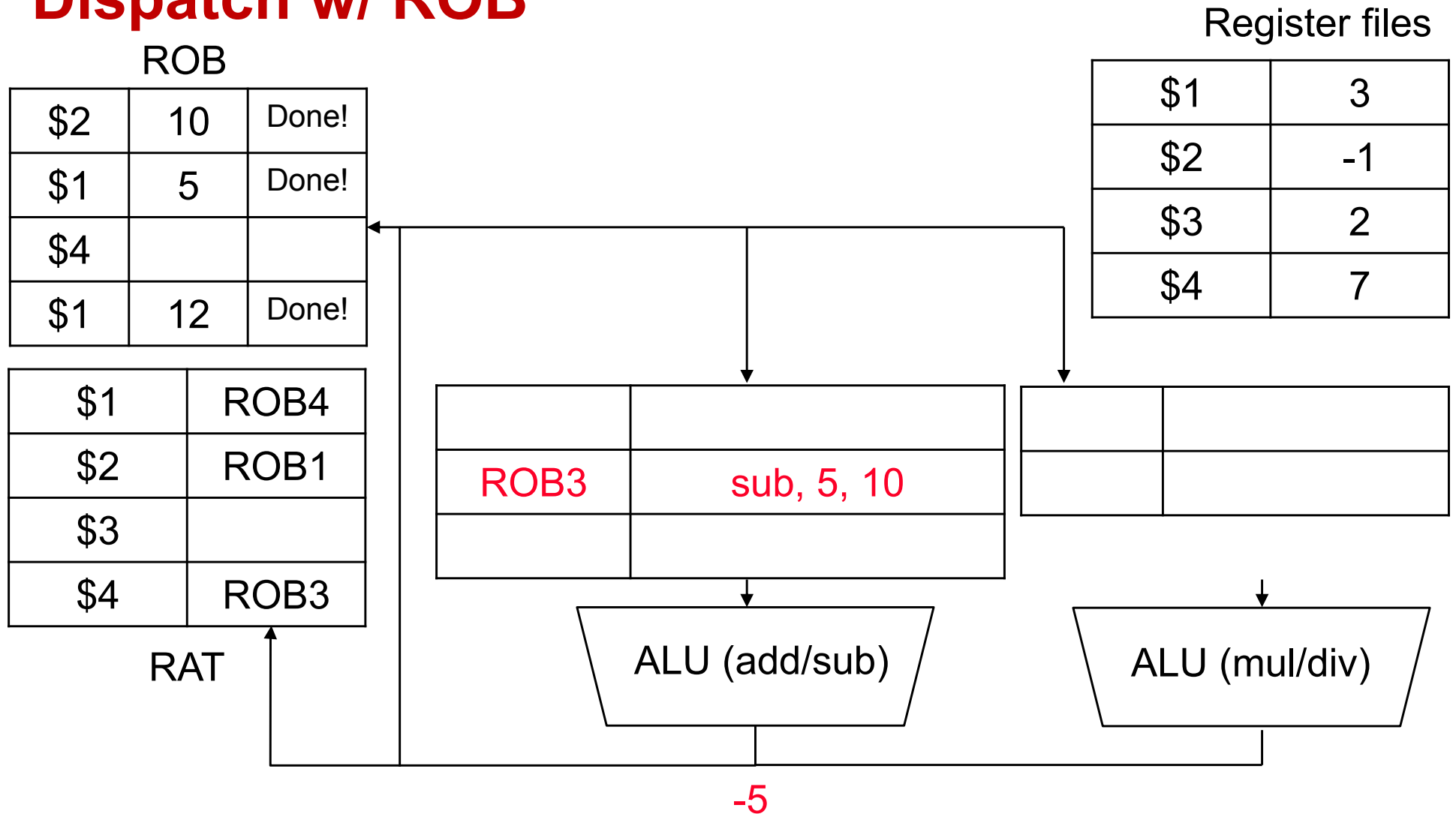
Dispatch w/ ROB



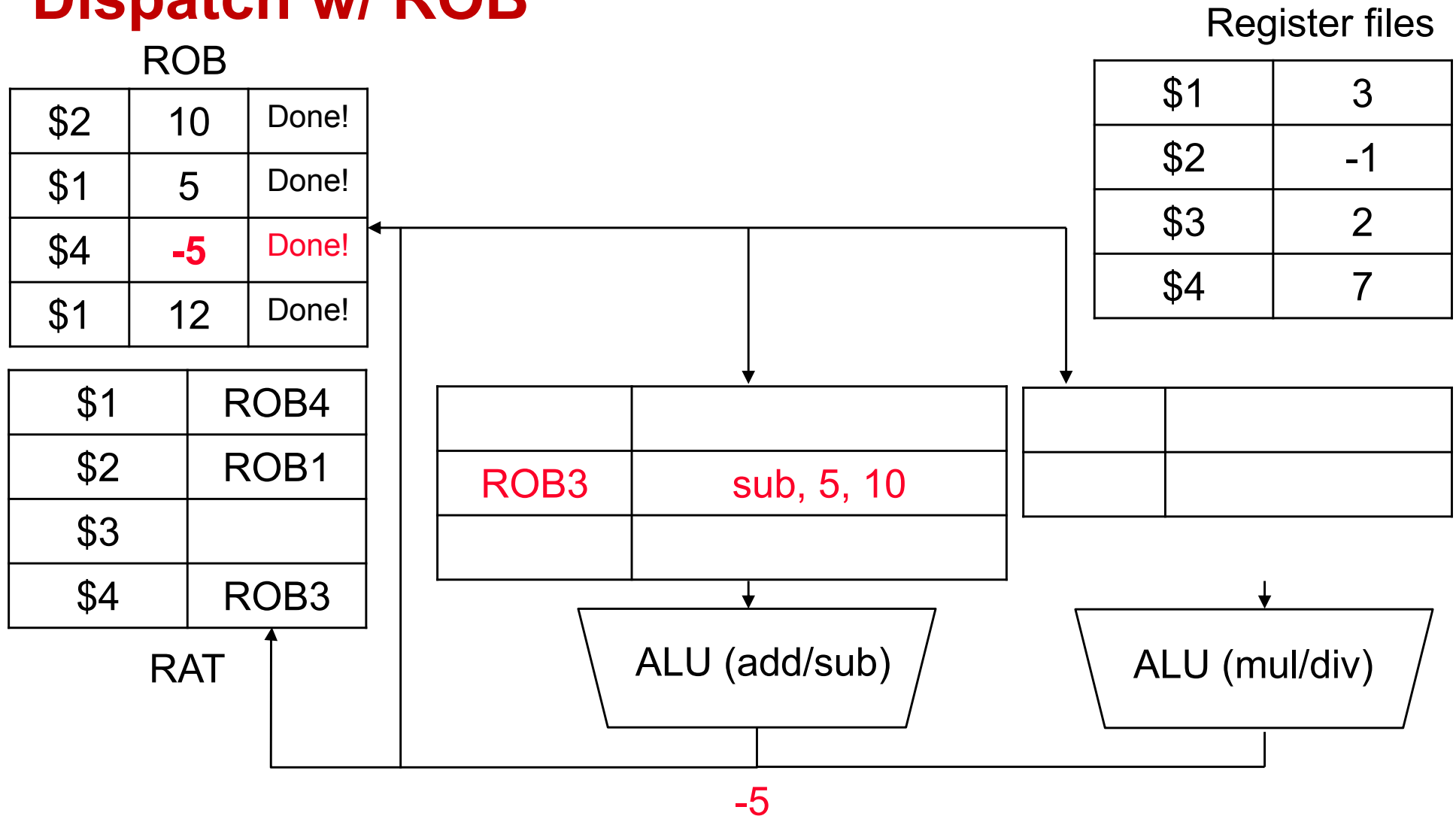
Dispatch w/ ROB



Dispatch w/ ROB



Dispatch w/ ROB



In-order Commit

ROB			
\$2	10	Done!	← commit
\$1	5	Done!	
\$4			
\$1	12	Done!	← issue

Register files	
\$1	3
\$2	-1
\$3	2
\$4	7

In-order Commit

ROB

\$2	10	Done!
\$1	5	Done!
\$4		
\$1	12	Done!

← commit

← issue

Register files

\$1	3
\$2	10
\$3	2
\$4	7

In-order Commit

ROB

\$1	5	Done!
\$4		
\$1	12	Done!

← commit

← issue

Register files

\$1	5
\$2	10
\$3	2
\$4	7

In-order Commit

ROB			
\$4			← commit
\$1	12	Done!	← issue

Register files	
\$1	5
\$2	10
\$3	2
\$4	7

- Cannot commit \$1 until \$4 is ready for commit
- Your register files are in a valid state even if you crash here!

In-order Commit: Memory

❑ What about loads and stores?

- ❑ They also gets an entry in the ROB.
- ❑ Store writes value to a temporary load-store queue (LSQ) instead of directly writing it into the cache/memory.
- ❑ Load searches LSQ first before reading the memory.
 - And read the youngest match
 - Conceptually similar to register forwarding (often called memory forwarding)
- ❑ LSQ only writes to the cache/memory on commit!

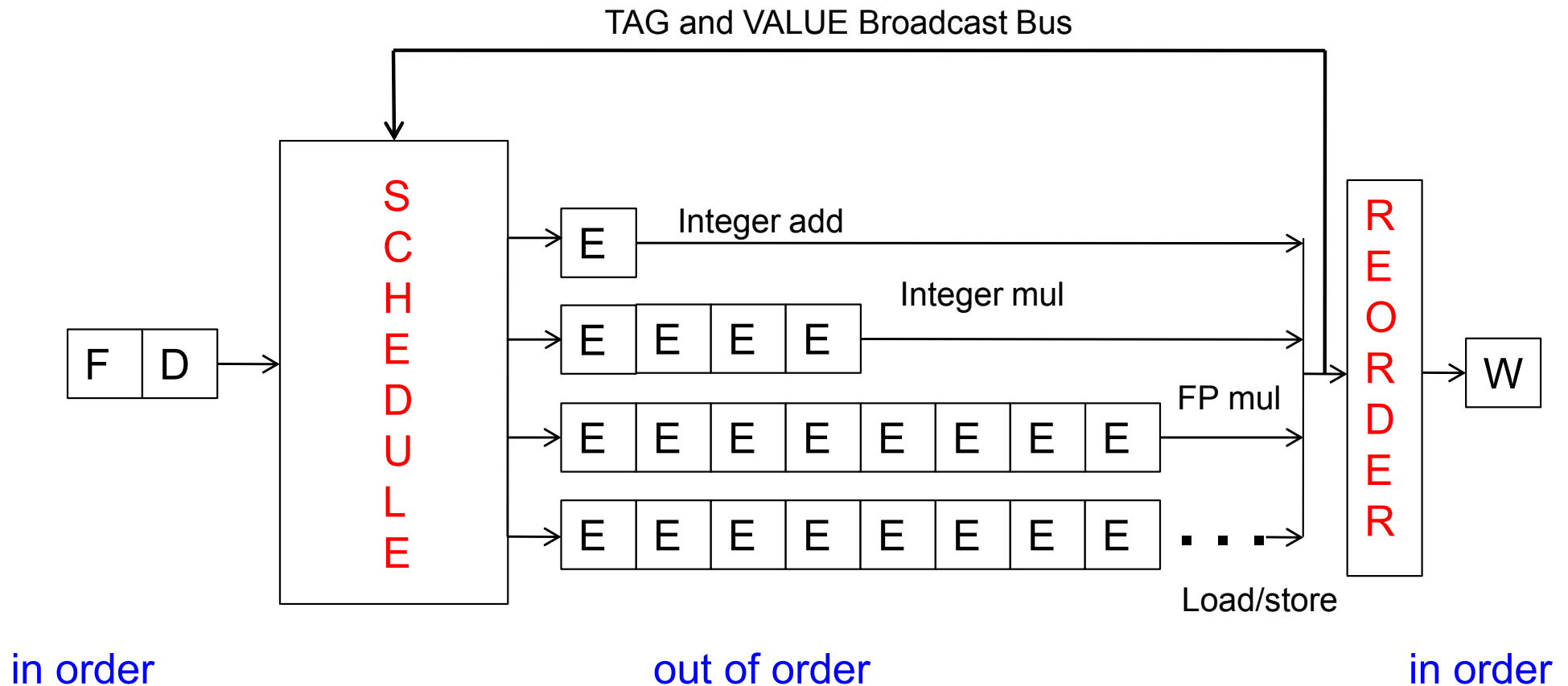
Tomasulo's (And Beyond) Algorithm: Details

- ❑ In fact, all (issue, dispatch, broadcast, commit) can happen in every cycle.
- ❑ What if the reservation station is full?
 - ❑ Stall (cannot issue)
- ❑ What if an ALU is not available (e.g., mul/div may take multiple cycles)?
 - ❑ Stall (cannot dispatch)
- ❑ What if more than one instruction is ready to dispatch?
 - ❑ Round-robin
 - ❑ Old one first
 - ❑ ...
- ❑ What if the ROB/LSQ is full?
 - ❑ Stall (cannot issue)
- ❑ Supporting speculative execution (branch prediction)
 - ❑ Can be done by not committing speculative execution in the ROB

Tomasulo's (And Beyond) Algorithm: Recap

- ❑ RAW → Eased by reordering
- ❑ WAR, WAW → Solved by renaming
- ❑ Memory RAW, WAR, WAW (load/store) → Do not reorder
- ❑ Precise exception → Supported by in-order commit w/ ROB and SQ (not supported in Tomasulo)

Two Humps in the Pipeline



- ❑ **Hump 1:** Reservation stations (scheduling window)
- ❑ **Hump 2:** Reordering (reorder buffer, aka instruction window or active window)

Aggressive Memory Reordering w/ LSQ

- ❑ What should a load do if there is a slow store?
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart

lw ..., 0(\$1)

sw ..., 0(\$2)

lw ..., 0(\$3)

sw ..., 0(\$4)

lw ..., 0(\$5)

L/S?	ADDR	VAL
L	0x100	...

Aggressive Memory Reordering w/ LSQ

- ❑ What should a load do if there is a slow store?
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart

```
lw ..., 0($1)
sw ..., 0($2)
lw ..., 0($3)
sw ..., 0($4)
lw ..., 0($5)
```

L/S?	ADDR	VAL
L	0x100	...
S	0x200	42

Store can precede load!

Aggressive Memory Reordering w/ LSQ

- ❑ What should a load do if there is a slow store?
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart

lw ..., 0(\$1)

sw ..., 0(\$2)

lw ..., 0(\$3)

sw ..., 0(\$4)

lw ..., 0(\$5)

L/S?	ADDR	VAL
L	0x100	...
S	0x200	42
L	0x200	

Aggressive Memory Reordering w/ LSQ

- ❑ What should a load do if there is a slow store?
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart

```
lw ..., 0($1)
sw ..., 0($2)
lw ..., 0($3)
sw ..., 0($4)
lw ..., 0($5)
```

L/S?	ADDR	VAL
L	0x100	...
S	0x200	42
L	0x200	42

Load gets the value from the youngest LSQ entry

Aggressive Memory Reordering w/ LSQ

- ❑ What should a load do if there is a slow store?
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart

```
lw ..., 0($1)
sw ..., 0($2)
lw ..., 0($3)
sw ..., 0($4)
lw ..., 0($5)
```

L/S?	ADDR	VAL
L	0x100	
S	0x200	42
L	0x200	42
S	...	

Aggressive Memory Reordering w/ LSQ

- ❑ What should a load do if there is a slow store?
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart

lw ..., 0(\$1)
sw ..., 0(\$2)
lw ..., 0(\$3)
sw ..., 0(\$4)
lw ..., 0(\$5)

L/S?	ADDR	VAL
L	0x100	
S	0x200	42
L	0x200	42
S	...	
L	0x300	

Can we go ahead and read from the cache/memory address 0x300?

- Conservative: Wait for the preceding store to finish
- Aggressive: Just go ahead!

Aggressive Memory Reordering w/ LSQ

- ❑ What should a load do if there is a slow store?
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart

```
lw ..., 0($1)
sw ..., 0($2)
lw ..., 0($3)
sw ..., 0($4)
lw ..., 0($5)
```

L/S?	ADDR	VAL
L	0x100	
S	0x200	42
L	0x200	42
S	...	
L	0x300	784

Aggressive Memory Reordering w/ LSQ

- ❑ What should a load do if there is a slow store?
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart

```
lw ..., 0($1)
sw ..., 0($2)
lw ..., 0($3)
sw ..., 0($4)
lw ..., 0($5)
```

L/S?	ADDR	VAL
L	0x100	
S	0x200	42
L	0x200	42
S	0x400	7
L	0x300	784

Fine!

Aggressive Memory Reordering w/ LSQ

- ❑ What should a load do if there is a slow store?
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart

```
lw ..., 0($1)
sw ..., 0($2)
lw ..., 0($3)
sw ..., 0($4)
lw ..., 0($5)
```

L/S?	ADDR	VAL
L	0x100	
S	0x200	42
L	0x200	42
S	0x300	7
L	0x300	784

Need to fix

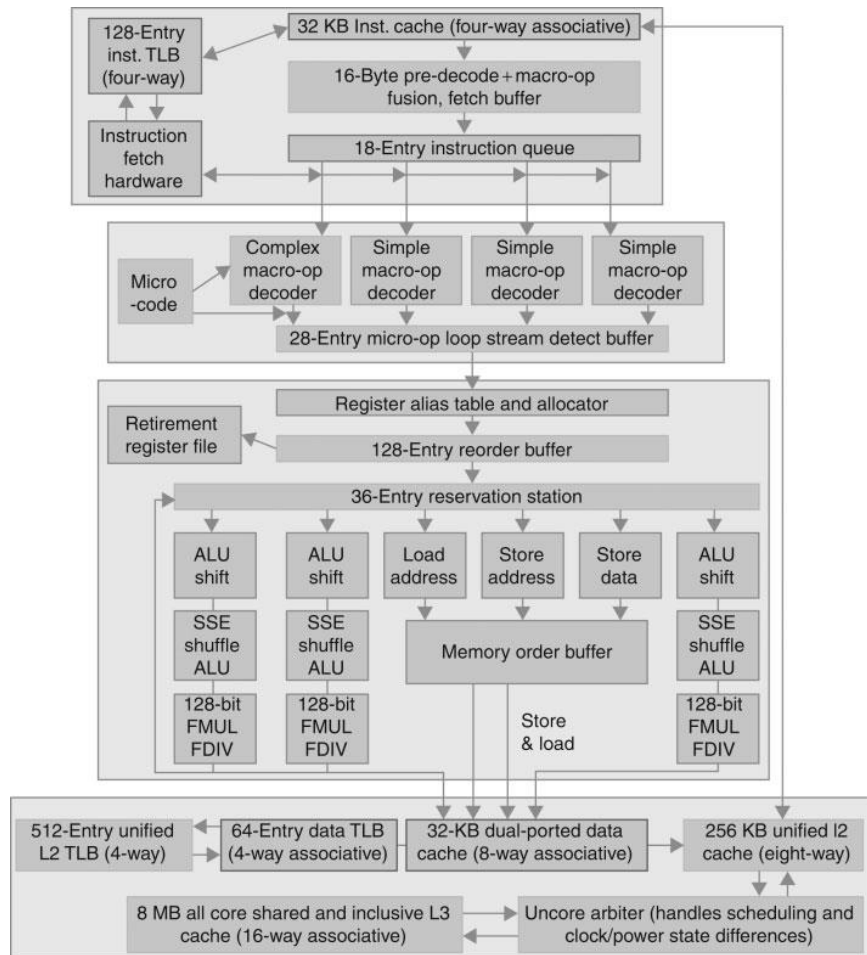
Aggressive Memory Reordering w/ LSQ

- ❑ Goal is to execute a load before a preceding store
- ❑ Speculation requires two things ...
 1. Detection of mis-speculations (guessed that memory addresses didn't match, and it turns out that they did)
 - Can be done by searching for younger load entries in the LSQ
 2. Recovery from mis-speculations
 - Squash instructions after offending load (they may be using the load data)
 - Saw how to squash instructions after mispredicted branches: same method

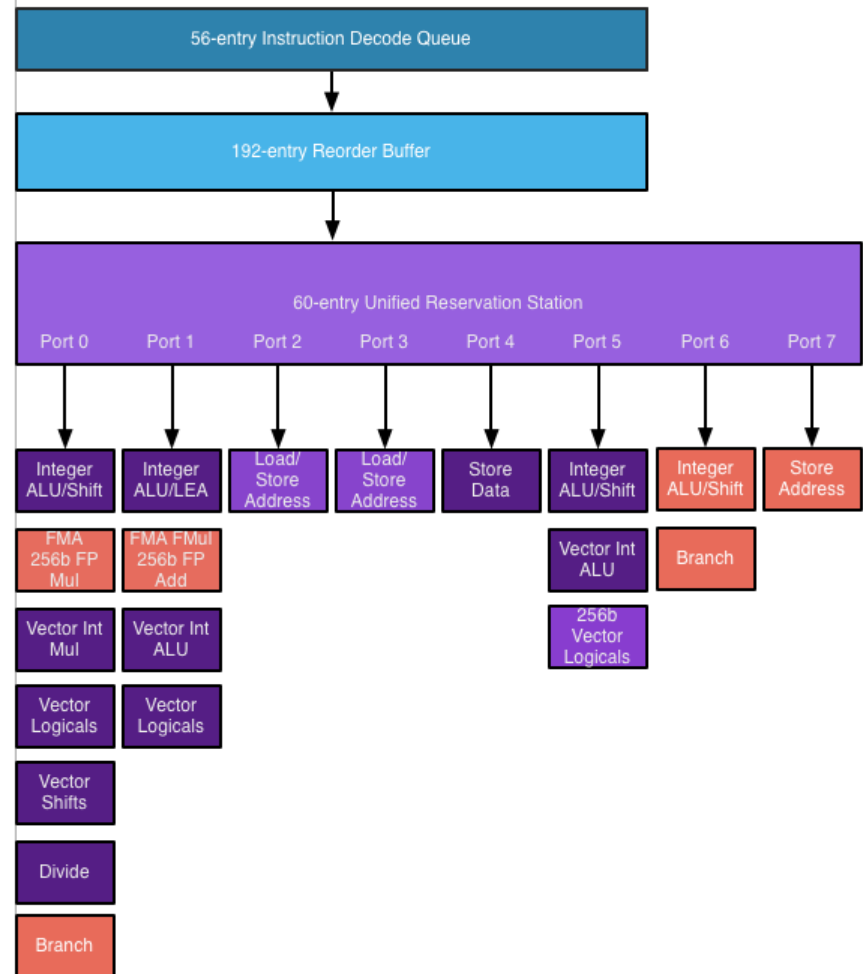
LSQ = Store Queue + Load Queue

- ❑ **Store Queue:** used for memory forwarding
 - ❑ Written into by stores (@ store execute)
 - ❑ Searched by loads (@ load execute) for store forwarding
 - ❑ Write to data cache (@ store Commit)
- ❑ **Load Queue:** used to detect ordering violations
 - ❑ Written into by loads (@ load execute)
 - ❑ Searched by stores (@ store execute) to detect load ordering violation
- ❑ **Both together**
 - ❑ Allow aggressive load scheduling
 - Stores don't constrain load execution
 - ❑ Help us improve performance significantly

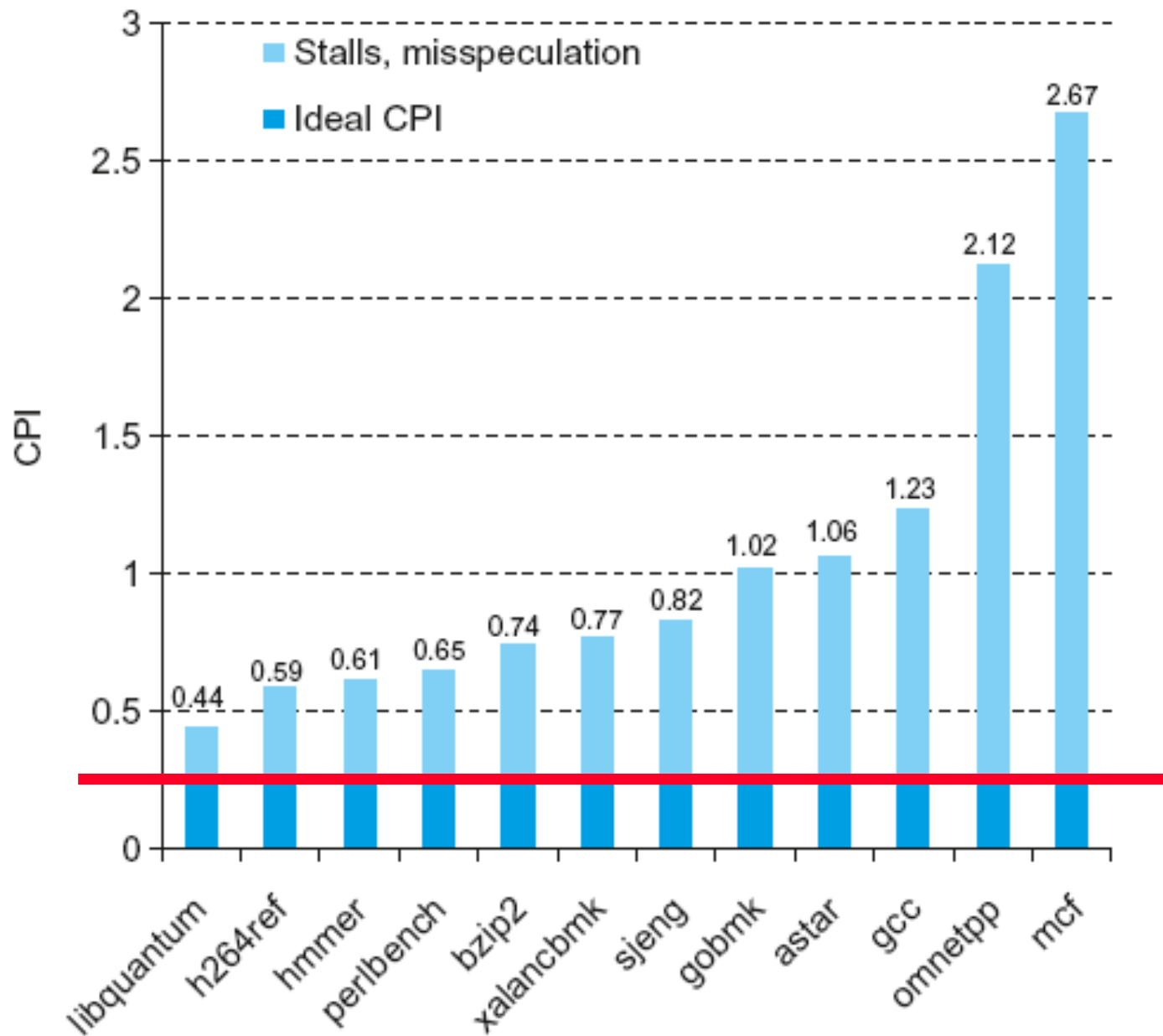
Real-world OoO processors



Intel Haswell Execution Engine




Core i7 Performance (SPEC2006 Int Benchmarks)



Cortex A8 versus Intel i7

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue?	Yes	Yes
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	2-level	2-level
1 st level caches/core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2 nd level caches/core	128-1024 KiB	256 KiB
3 rd level caches (shared)	-	2- 8 MB

Review: Multithreaded Implementations

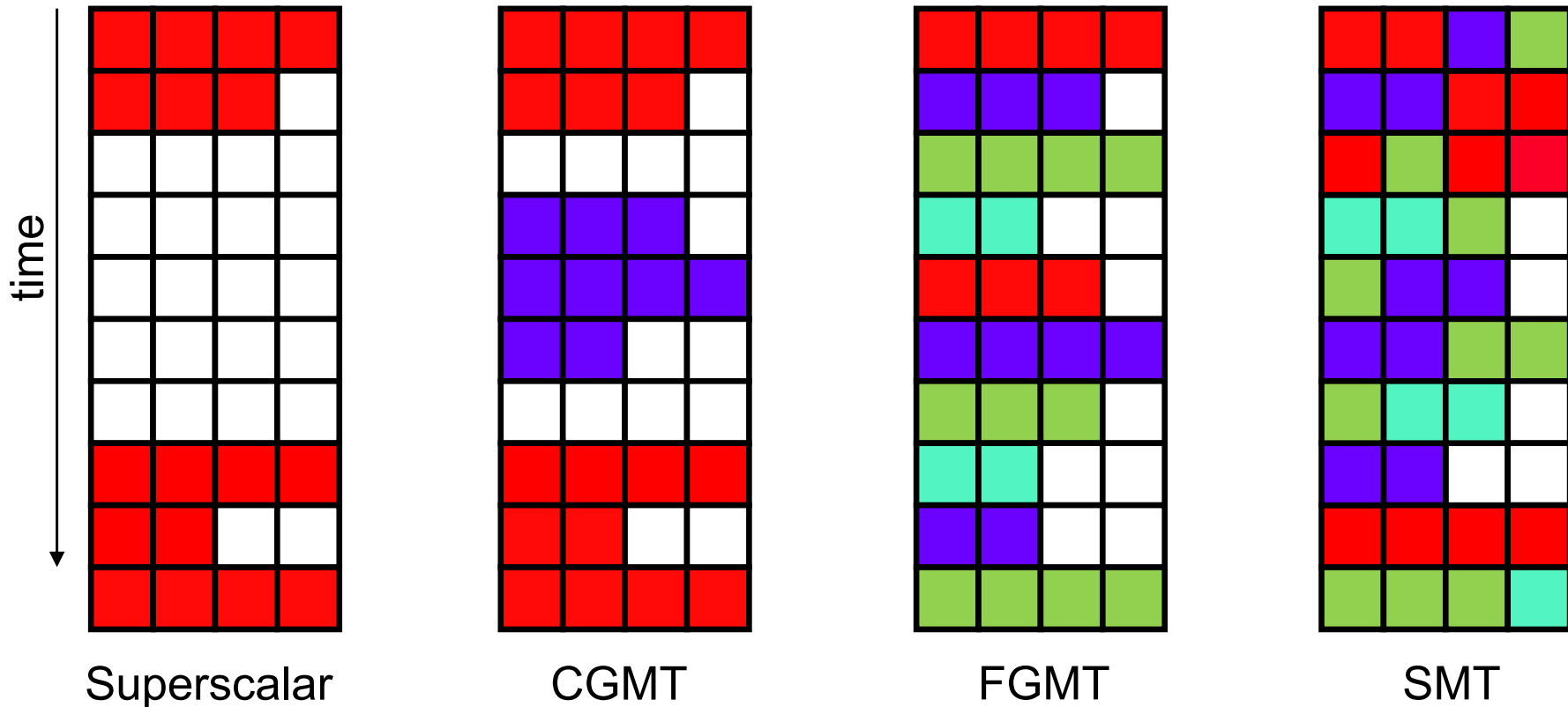
- ❑ MT trades (single-thread) latency for throughput
 - ❑ Sharing the datapath degrades the **latency** of individual threads, but improves the aggregate latency of both threads
 - ❑ And it improves **utilization** of the datapath hardware
- ❑ Main questions: **thread scheduling policy** and **pipeline partitioning**
 - ❑ When to switch from one thread to another?
 - ❑ How exactly do threads share the pipelined datapath itself?
- ❑ Choices depends on what kind of latencies you want to tolerate and how much single thread performance you are willing to sacrifice
 - ❑ Coarse-grain multithreading (**CGMT**)
 - ❑ Fine-grain multithreading (**FGMT**)
 - ❑ Simultaneous multithreading (**SMT**) 

Simultaneous MultiThreading (SMT)

- ❑ What can issue instr's from multiple threads in one cycle?
 - ❑ Same thing that issues instr's from multiple parts of same thread ...
 - ❑ ...out-of-order execution !!
- ❑ **Simultaneous multithreading (SMT): OOO + FGMT**
 - ❑ Most common implementation of multithreading!
 - ❑ Aka (by Intel) “hyper-threading”
 - ❑ Once instr's are renamed, issuer doesn't care which thread they come from (well, for non-loads at least)
 - ❑ Some examples
 - IBM Power5: 4-way, 2 threads; IBM Power7: 4-way, 4 threads
 - Intel Pentium4: 3-way, 2 threads; Intel Core i7: 4-way, 2 threads
 - AMD Bulldozer: 4-way, 2 threads
 - Alpha 21464: 8-way issue, 4 threads (canceled)

Time Evolution of Issue Slots

□ Color = thread



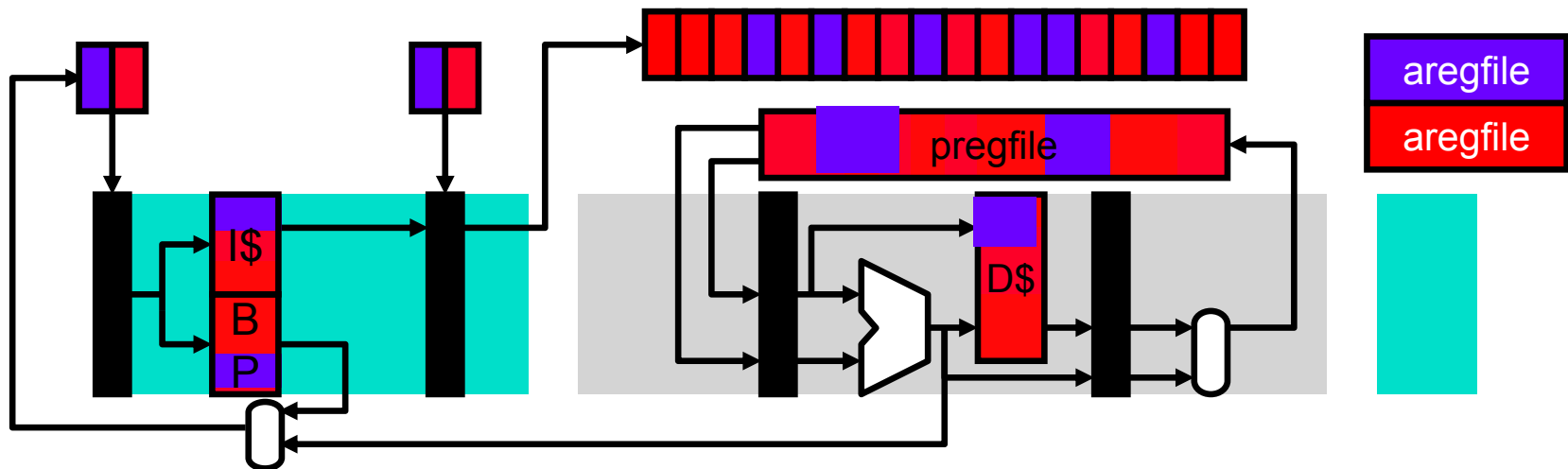
Static/Dynamic ROB & LSQ Partitioning (T threads and P partitions)

❑ Static partitioning

- ❑ T equal-sized contiguous partitions
- ± No starvation, sub-optimal utilization (**fragmentation**)

❑ Dynamic partitioning

- ❑ $P > T$ partitions, available partitions assigned on need basis
 - ± Better utilization, possible starvation
- ❑ Couple both with larger ROB/LSQs



TODO: Add some Tomasulo practice questions