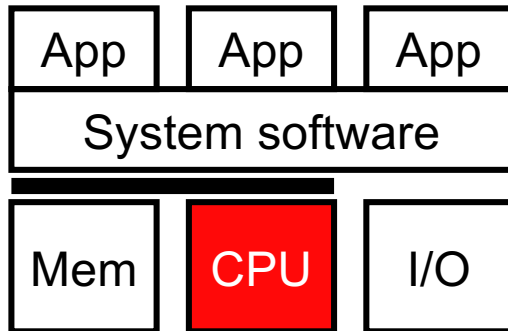


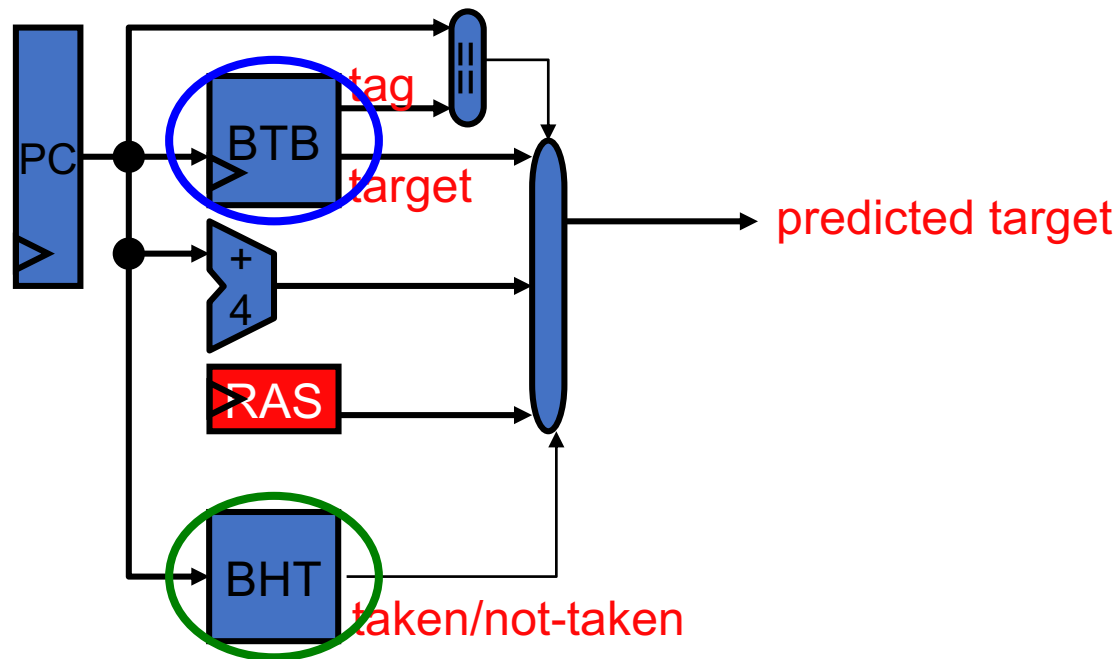
Review: Pipelining



- Single-cycle datapaths
- Latency vs. throughput & performance
- Basic pipelining
- Data hazards
 - Bypassing
 - Load-use stalling
- Pipelined multi-cycle operations
- Control hazards
 - Branch prediction
- Hardware multithreading

Review: What's inside a branch predictor?

- BTB & branch direction predictor during fetch

























- Step #1: is it a branch? BTB (branch target buffer)
- Step #2: is the branch taken or not taken? BHT (branch history table)
- Step #3: if the branch is taken, where does it go? BTB, RAS (return address stack)



2-bit Branch Prediction Concept - Example

Edgar Bernal - Wednesday, November 9, 2016

8:44 PM

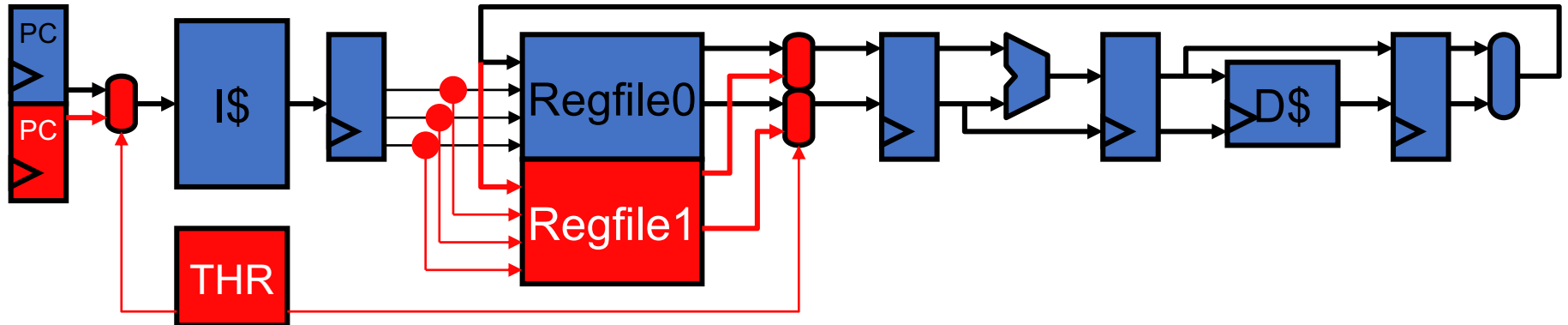
Time	State		Prediction	Outcome	Result?	
1	N		N	T	Wrong	
2	n		N	T	Wrong	
3	t		T	T	Correct	
4	T		T	N	Wrong	
5	t		T	T	Correct	
6	T		T	T	Correct	
7	T		T	T	Correct	
8	T		T	N	Wrong	
9	t		T	T	Correct	

N	
n	
t	
T	

Correct:	
Wrong:	

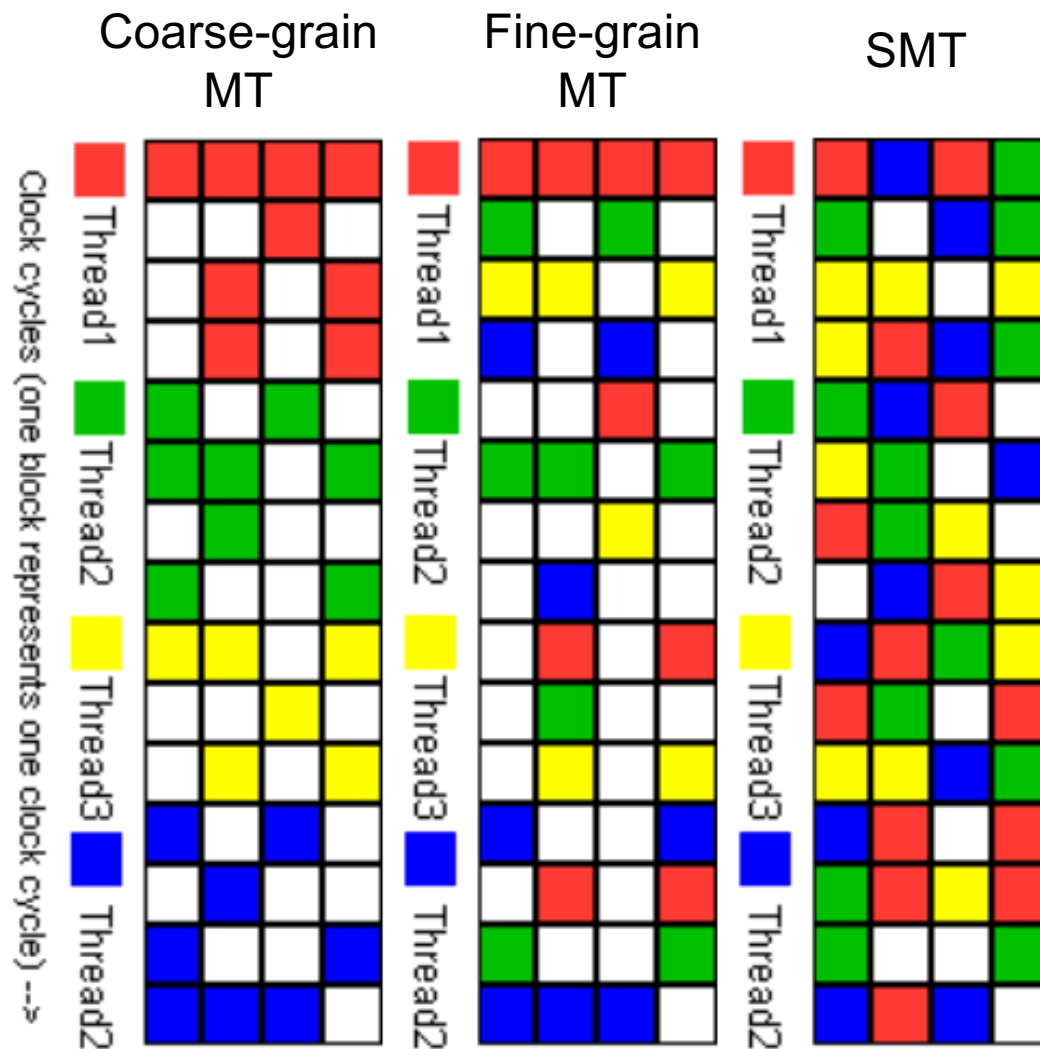
Review: Hardware Multithreading

- **Not** the same as software multithreading!
- A **hardware thread** is a sequential stream of insns
 - could be a software thread or a single-threaded process

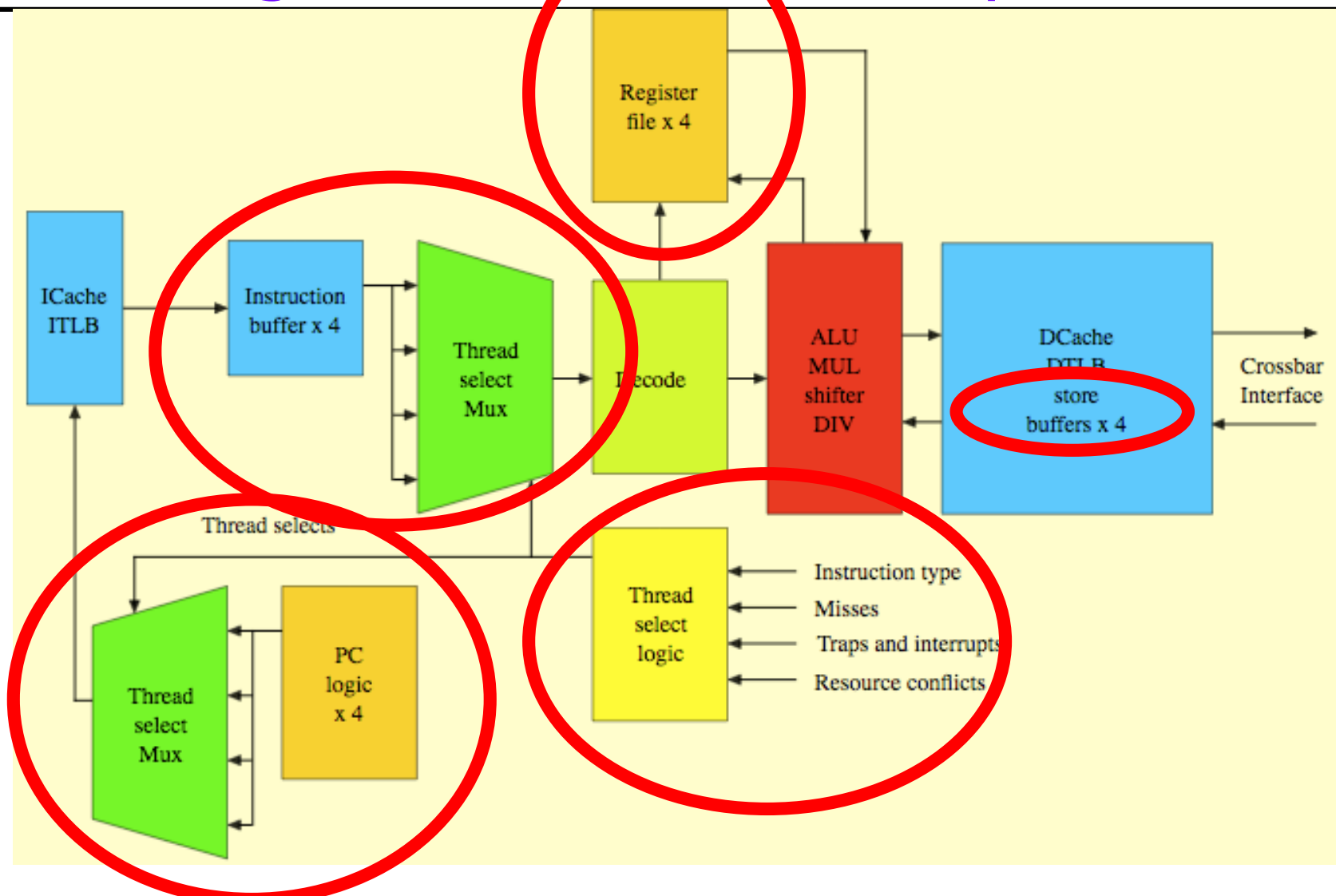


- **Hardware Multithreading (MT)**
 - Multiple hardware threads dynamically share a single pipeline
 - Replicate only per-thread structures: program counter & registers
 - Hardware interleaves instructions

Review: Hardware Multithreading (MT)



Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

Preliminary: Questions to Ponder

- **Instruction scheduling**
 - Determine the order of instructions get executed in the pipeline
 - Execution order does not necessarily matches the order written in software, as long as program correctness is guaranteed
- **What is the role of the hardware vs. the software in the order in which instructions are executed in the pipeline?**
 - Software based instruction scheduling → **static scheduling** (HW2)
 - Hardware based instruction scheduling → **dynamic scheduling**
- **What information does the compiler not know that makes static scheduling difficult? → Why dynamic scheduling?**
 - Answer: Anything that is determined at run time
 - **Variable-length operation latency, memory addr, branch direction**

Preliminary: Dynamic Instruction Scheduling

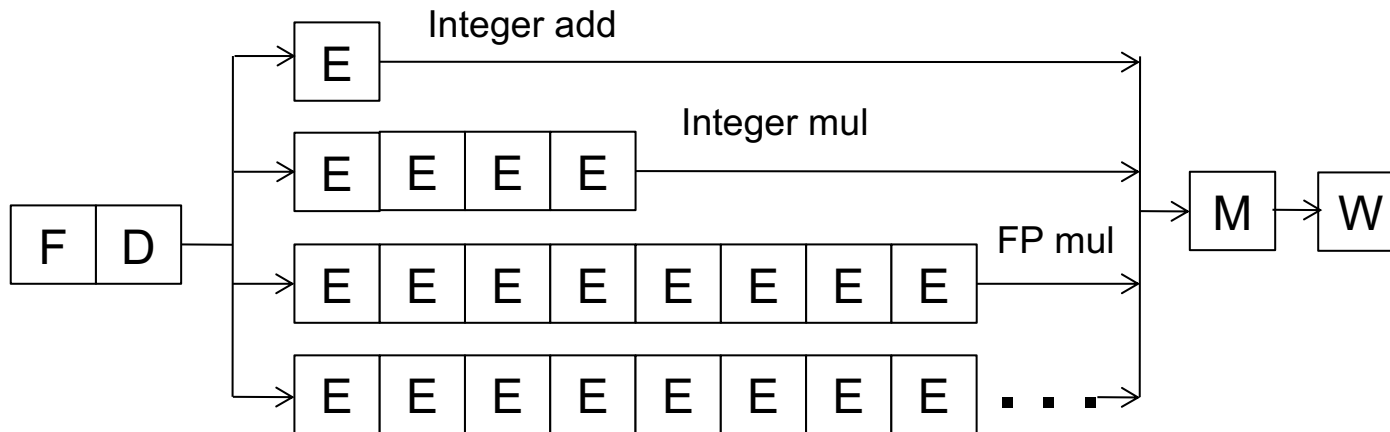
- Hardware has knowledge of dynamic events on a per-instruction basis (i.e., at a very fine granularity)
 - Branch mispredictions
 - Load/store addresses
 - Cache misses
- Wouldn't it be nice if hardware did the scheduling of instructions?

Dynamic OoO approaches

- ❑ Scoreboarding (HP Appendix A.7) – CDC 6600 (Thornton) first publication in 1964
 - Used **centralized** hazard detection logic (scoreboard) to support OoO execution. Instr's are stalled when their FU is busy, for RAW dependencies, *and for WAW and WAR dependencies*.
- ❑ Tomasulo (HP 2.4) – IBM 360/91 (Tomasulo) first publication in 1967
 - Used **distributed** hazard detection logic (reservation stations feeding each FU) to support OoO execution with *register renaming* that eliminates WAW and WAR dependencies; distributes results from FUs to reservation stations on a Common Data Bus (potential bottleneck)
 - Writes results to register file and memory when instr's complete – possibly out-of-order – so *can not support precise interrupts or speculative execution* (e.g., branch speculation)
 - <http://www.ecs.umass.edu/ece/koren/architecture/Tomasulo1/tomasulo.htm>

An In-order Pipeline

F, D, **E**, M, W



- Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units
- Dispatch: Act of sending an instruction to an execution functional unit (i.e., in E stage)
- Load and store instructions can take multiple cycles in M stage

Can We Do Better?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

IMUL R3 \leftarrow R1, R2

ADD R3 \leftarrow R3, R1

ADD R1 \leftarrow R6, R7

IMUL R5 \leftarrow R6, R8

ADD R7 \leftarrow R9, R9

LD R3 \leftarrow R1 (0) // M stage may take multiple cycles

ADD R3 \leftarrow R3, R1

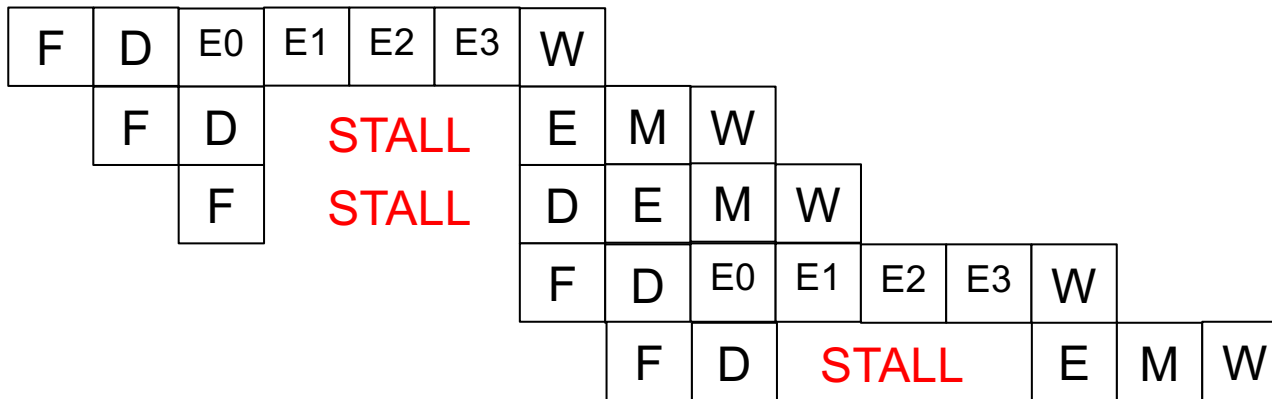
ADD R1 \leftarrow R6, R7

IMUL R5 \leftarrow R6, R8

ADD R7 \leftarrow R9, R9

In-order vs. Out-of-order Dispatch

- In order dispatch + precise exceptions:



IMUL R3 \leftarrow R1, R2
ADD R3 \leftarrow R3, R1
ADD R1 \leftarrow R6, R7
IMUL R5 \leftarrow R6, R8
ADD R7 \leftarrow R3, R5

- 15 cycles

Can We Do Better?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

IMUL R3 \leftarrow R1, R2

ADD R3 \leftarrow R3, R1

ADD R1 \leftarrow R6, R7

IMUL R5 \leftarrow R6, R8

ADD R7 \leftarrow R9, R9

LD R3 \leftarrow R1 (0) // M stage may take multiple cycles

ADD R3 \leftarrow R3, R1

ADD R1 \leftarrow R6, R7

IMUL R5 \leftarrow R6, R8

ADD R7 \leftarrow R9, R9

- Answer: First ADD can stall the whole pipeline!**
 - ADD cannot dispatch because its source registers unavailable
 - Later **independent** instructions cannot get executed
- How are the above code portions different?
 - Answer: Load latency is variable (unknown until runtime)**
 - What does this affect? Think compiler vs. microarchitecture

Preventing Dispatch Stalls

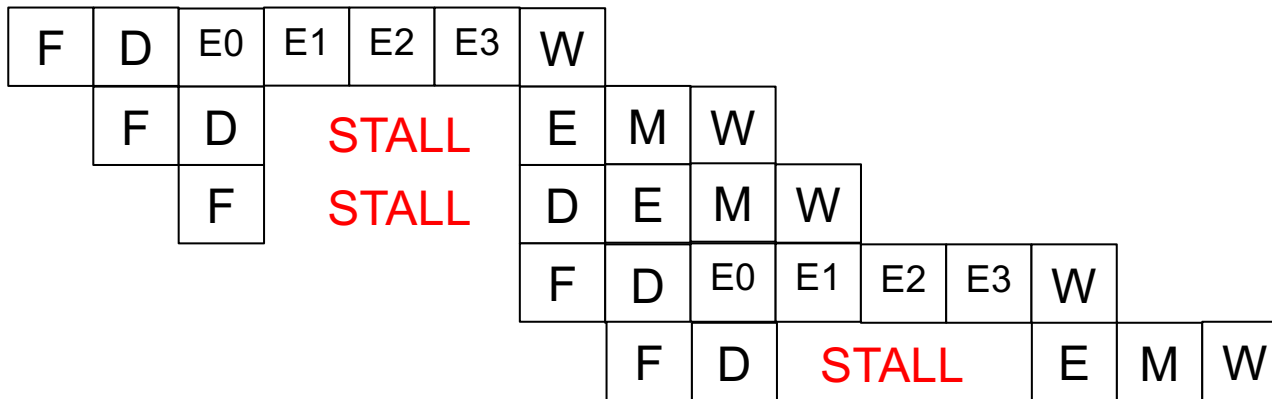
- Multiple ways of doing it
- You have already seen at least one:
 - Fine-grained multithreading

May see more later on (maybe in other classes):

 - Value prediction
 - Compile-time instruction scheduling/reordering (i.e., static instruction scheduling)
- What are the disadvantages of fine-grained multithreading?
- Any other way to prevent dispatch stalls?
 - Problem: in-order dispatch (scheduling, or execution)
 - Solution: out-of-order dispatch (scheduling, or execution)

In-order vs. Out-of-order Dispatch

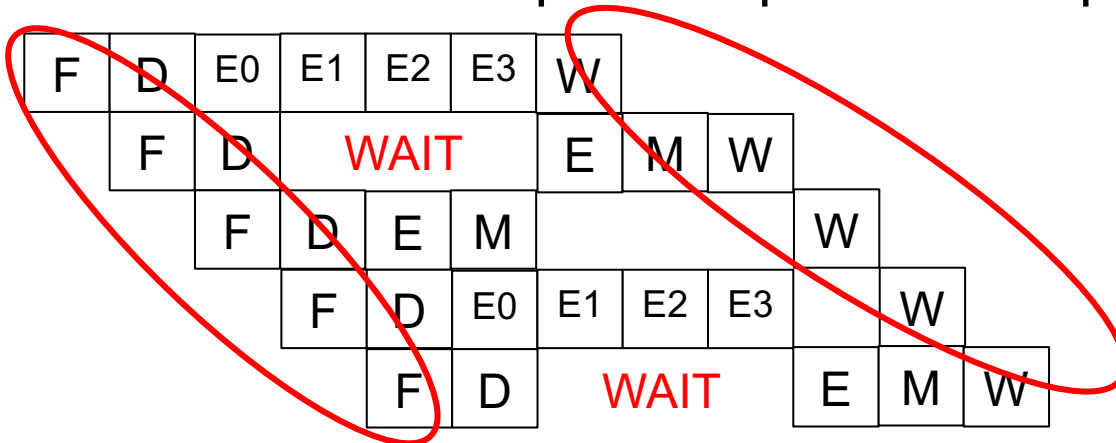
- In order dispatch + precise exceptions:



```

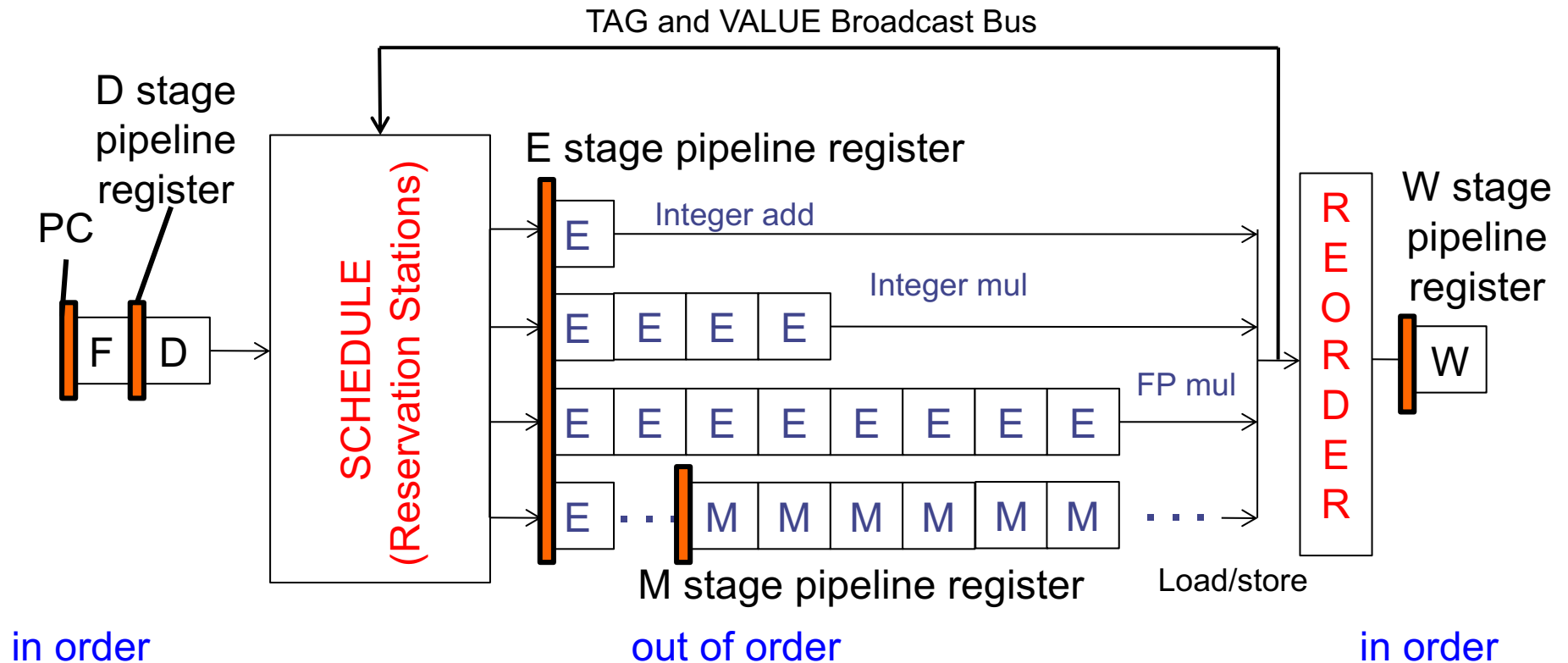
IMUL R3 ← R1, R2
ADD  R3 ← R3, R1
ADD  R1 ← R6, R7
IMUL R5 ← R6, R8
ADD  R7 ← R3, R5
    
```

- Out-of-order dispatch + precise exceptions:



- 15 vs. 12 cycles

Two Humps in a Modern Pipeline



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

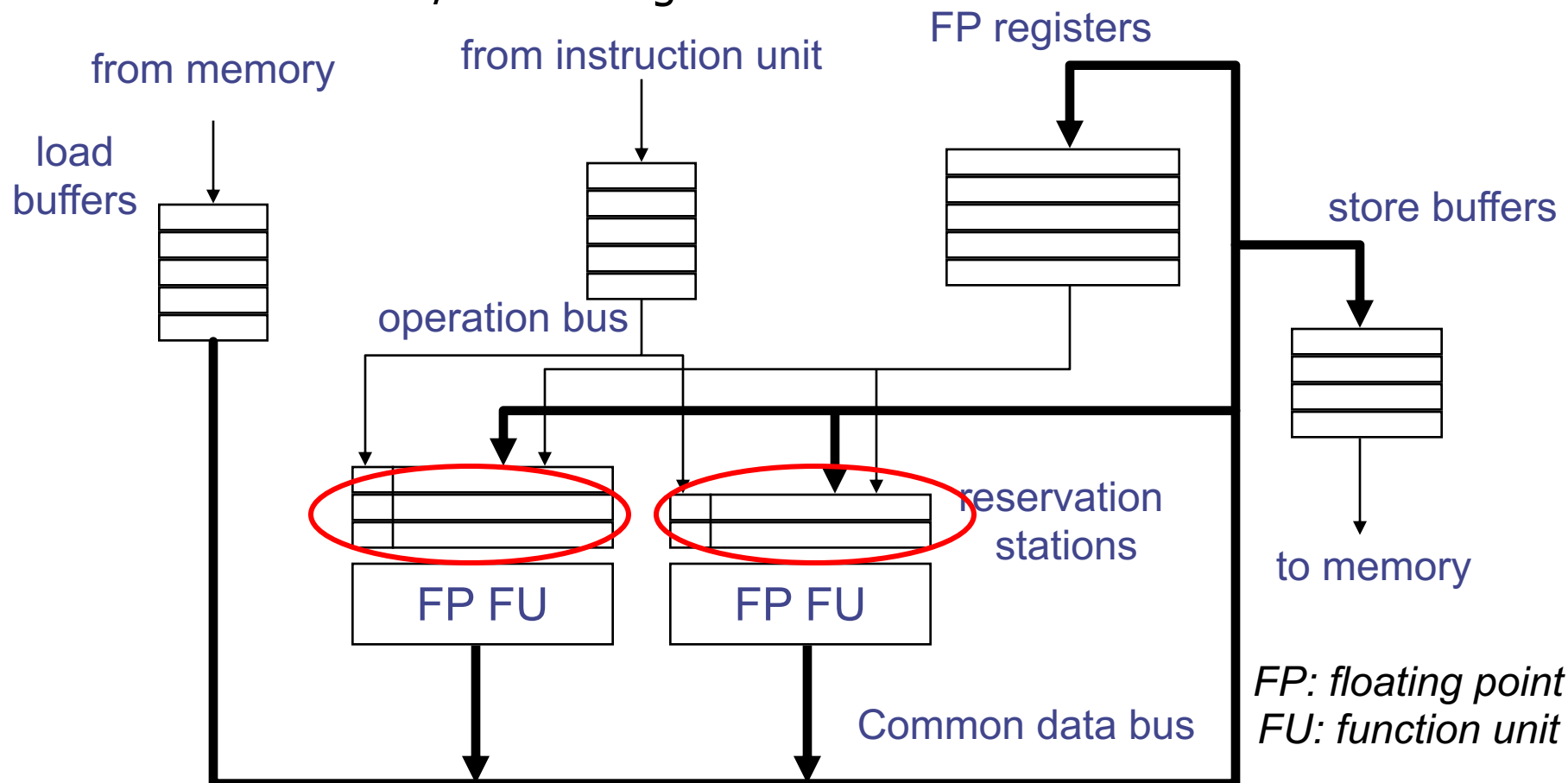
Enabling OoO Execution

1. Need to link the consumer of a value to the producer
 - **Register renaming:** Associate a “tag” with each data value
2. Need to buffer instructions until they are ready to execute
 - Insert instruction into **reservation stations** after renaming
3. Instructions need to keep track of readiness of source values
 - **Broadcast the “tag”** when the value is produced
 - Instructions **compare their “source tags”** to the broadcast tag → if match, source value becomes ready
4. When all source values of an instruction are ready, need to dispatch the instruction to its functional unit (FU)
 - Instruction **wakes up** if all sources are ready
 - If multiple instructions are awake, need to **select** one per function unit

Register renaming

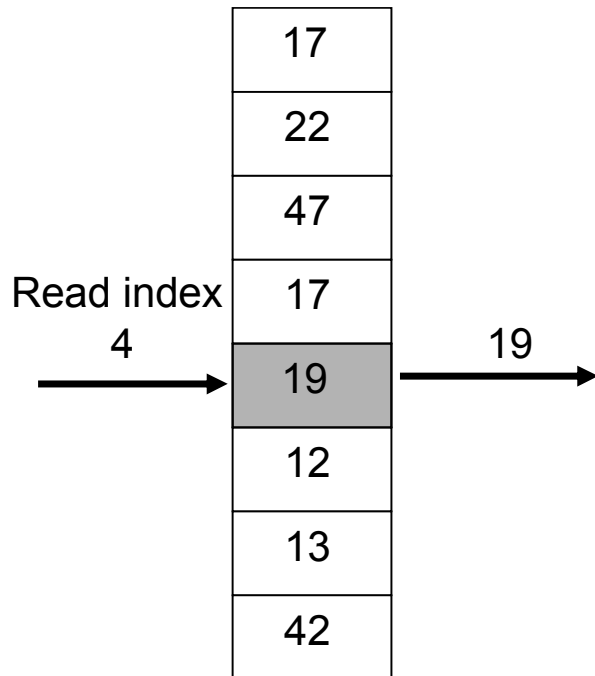
Out-of-order execution (with register renaming) invented by Robert Tomasulo in 1960's

- Used in IBM 360/91 Floating Point Units

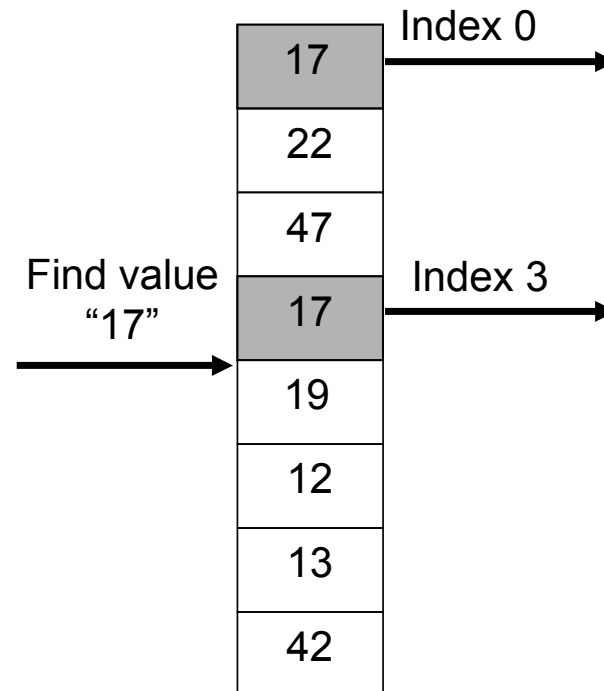


Reservation station (issue queue)

RAM: read/write specific index



CAM: search for value



- One structure can have ports for both types (RAM & CAM)

Detail 1: Register renaming

The processor only has 4 registers: R1 ~ R4

- Some dependencies are not true dependencies
 - The same register refers to values that have nothing to do with each other *Not true dependency:*
`add R3<- R1, R2`
`addi R1<- R4, #100`
 - **They exist because not enough register ID's (i.e. names) in the ISA** *True dependency:*
`addi R1<- R4, #100`
`add R3<- R1, R2`
- Solution: The register ID is **renamed** to the reservation station entry that will hold the register's value
 - Register ID → Reservation Station entry ID
 - After renaming, Reservation Station entry ID is used to refer to the register
- This eliminates the dependency shown in our example
 - Achieves (approximates) the performance of having a large number of registers, even though ISA does not support that many

Register renaming

- ❑ To eliminate (WAW, WAR) register conflicts/hazards
- ❑ “Architected” vs “Physical” registers – level of indirection
 - Architected (ISA) register names: **r1** , **r2** , **r3**
 - Physical register locations: **p1** , **p2** , **p3** , **p4** , **p5** , **p6** , **p7**
 - Original mapping: **r1**→**p1**, **r2**→**p2**, **r3**→**p3**, **p4**–**p7** are “available”

MapTable			Free List	Original instr's	Renamed instr's
r1	r2	r3	p4 , p5 , p6 , p7	add r2 , r3 → r1	add p2 , p3 → p4
p1	p2	p3	p5 , p6 , p7	sub r2 , r1 → r3	sub p2 , p4 → p5
p4	p2	p3	p6 , p7	mul r2 , r3 → r3	mul p2 , p5 → p6
p4	p2	p5	p7	div r1 , 4 → r1	div p4 , 4 → p7
p4	p2	p6			

- Renaming – conceptually write each register once
 - + Removes **false** dependences (WAW and WAR)
 - + Leaves **true** dependences (RAW) intact!
- When to *reuse* a physical register? After overwriting instr done.

Register renaming 1

```
xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1
```

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map Table

p6
p7
p8
p9
p10

Free List

Register renaming 1a

`xor r1 ^ r2 → r3` \longrightarrow `xor p1 ^ p2 →` `[p3]`
`add r3 + r4 → r4`
`sub r5 - r2 → r3`
`addi r3 + 1 → r1`

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map Table

p6
p7
p8
p9
p10

Free List

Register renaming 1b

`xor r1 ^ r2 → r3` → `xor p1 ^ p2 → p6` `[p3]`
`add r3 + r4 → r4`
`sub r5 - r2 → r3`
`addi r3 + 1 → r1`

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map Table

p7
p8
p9
p10

Free List

Register renaming 2a

`xor r1 ^ r2 → r3`
`add r3 + r4 → r4` → `xor p1 ^ p2 → p6` [p3]
`sub r5 - r2 → r3` `add p6 + p4 →` [p4]
`addi r3 + 1 → r1`

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map Table

p7
p8
p9
p10

Free List

Register renaming 2b

`xor r1 ^ r2 → r3`
`add r3 + r4 → r4`
`sub r5 - r2 → r3`
`addi r3 + 1 → r1`

→

`xor p1 ^ p2 → p6`
`add p6 + p4 → p7`

`[p3]`
`[p4]`

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map Table

p8
p9
p10

Free List

Register renaming 3a

xor r1 ^ r2 → r3		xor p1 ^ p2 → p6	[p3]
add r3 + r4 → r4		add p6 + p4 → p7	[p4]
sub r5 - r2 → r3	→	sub p5 - p2 →	[p6]
addi r3 + 1 → r1			

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map Table

p8
p9
p10

Free List

Register renaming 3b

<code>xor r1 ^ r2 → r3</code>		<code>xor p1 ^ p2 → p6</code>	<code>[p3]</code>
<code>add r3 + r4 → r4</code>		<code>add p6 + p4 → p7</code>	<code>[p4]</code>
<code>sub r5 - r2 → r3</code>	→	<code>sub p5 - p2 → p8</code>	<code>[p6]</code>
<code>addi r3 + 1 → r1</code>			

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map Table

p9
p10

Free List

Register renaming 4a

<code>xor r1 ^ r2 → r3</code>	<code>xor p1 ^ p2 → p6</code>	<code>[p3]</code>
<code>add r3 + r4 → r4</code>	<code>add p6 + p4 → p7</code>	<code>[p4]</code>
<code>sub r5 - r2 → r3</code>	<code>sub p5 - p2 → p8</code>	<code>[p6]</code>
<code>addi r3 + 1 → r1</code>	<code>addi p8 + 1 →</code>	<code>[p1]</code>

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map Table

p9
p10

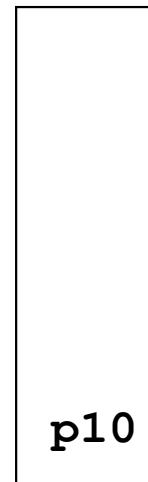
Free List

Register renaming 4b

<code>xor r1 ^ r2 → r3</code>	<code>xor p1 ^ p2 → p6</code>	<code>[p3]</code>
<code>add r3 + r4 → r4</code>	<code>add p6 + p4 → p7</code>	<code>[p4]</code>
<code>sub r5 - r2 → r3</code>	<code>sub p5 - p2 → p8</code>	<code>[p6]</code>
<code>addi r3 + 1 → r1</code>	<code>addi p8 + 1 → p9</code>	<code>[p1]</code>

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map Table



Free List

Freeing overwritten registers

xor r1 ^ r2 → r3	xor p1 ^ p2 → p6	[p3]
add r3 + r4 → r4	add p6 + p4 → p7	[p4]
sub r5 - r2 → r3	sub p5 - p2 → p8	[p6]
addi r3 + 1 → r1	addi p8 + 1 → p9	[p1]

❑ p3 was r3 **before** xor

❑ p6 is r3 **after** xor

- Anything older (earlier in the code) than xor should read p3
- Anything younger (later in the code) than xor should read p6 (until next r3 writing instruction)

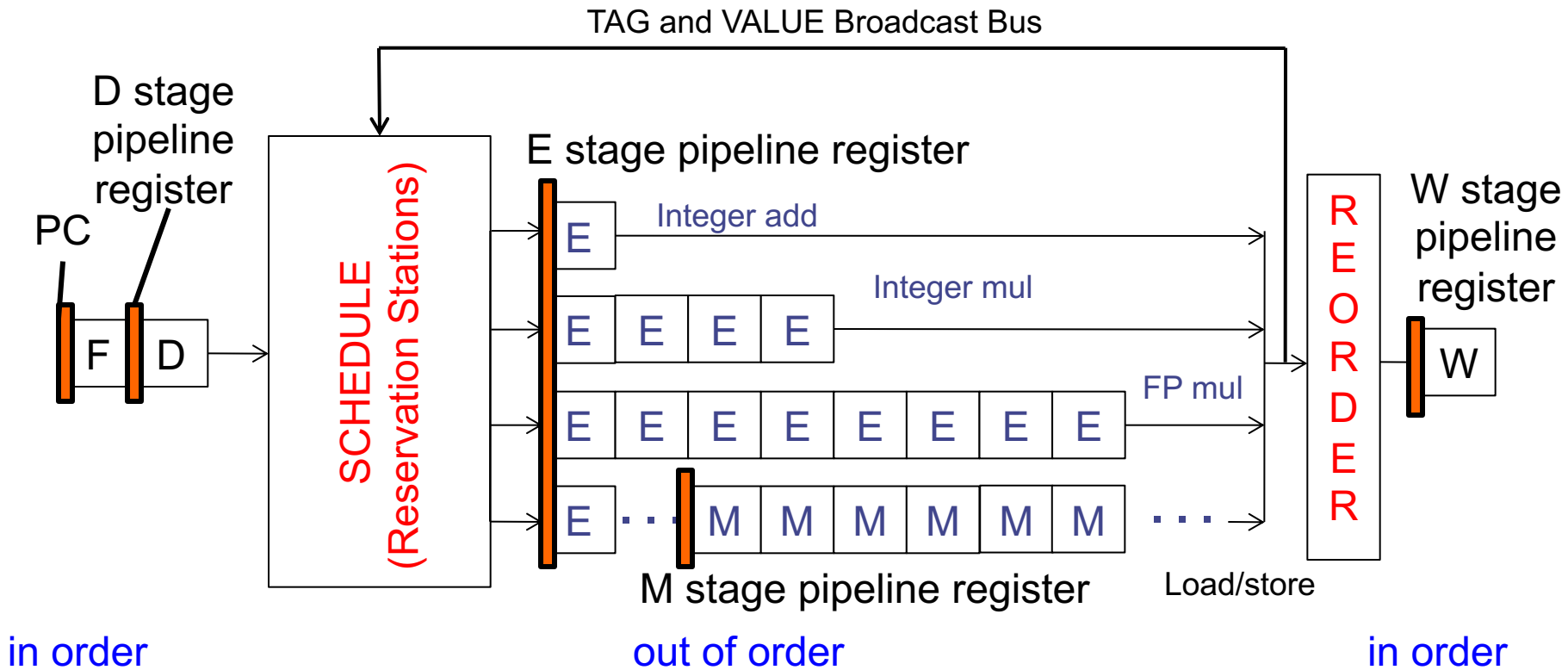
❑ At Commit of xor, no older instructions exist, so free p3!

Register rename table

- Register rename table (register alias table)

	tag	value	valid?
R0			1
R1			1
R2			1
R3			1
R4			1
R5			1
R6			1
R7			1
R8			1
R9			1

Detail 2: Dispatch



- Dispatch: Act of sending an instruction to an execution functional unit (i.e., in E stage)

Dispatch steps

- Allocate reservation station (and ROB) slot
 - Full? Stall
- Read ready bits of inputs from Ready Table
 - Ready table 1-bit per physical register
- Clear ready bit of output in Ready Table
 - Instruction has not produced value yet
- Write data in reservation station slot
 - Record instr's "age"

Dispatch example step 0

xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8
addi p8 + 1 → p9

Issue Queue

Instr	Inp1	R	Inp2	R	Dst	Age

Ready Table

p1	y
p2	y
p3	y
p4	y
p5	y
p6	y
p7	y
p8	y
p9	y

Dispatch example step 1

xor p1 ^ p2 → p6

add p6 + p4 → p7

sub p5 - p2 → p8

addi p8 + 1 → p9

Issue Queue

Instr	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	p6	0

Ready Table

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	y
p8	y
p9	y

Dispatch example step 2

xor p1 ^ p2 → p6

add p6 + p4 → p7

sub p5 - p2 → p8

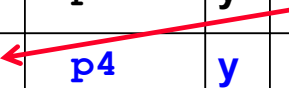
addi p8 + 1 → p9

Ready Table

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	y
p9	y

Issue Queue

Instr	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1



Dispatch example step 3

xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8
addi p8 + 1 → p9

Issue Queue

Instr	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1
sub	p5	y	p2	y	p8	2

Ready Table

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	n
p9	y

Dispatch example step 4

xor p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8
addi p8 + 1 → p9

Issue Queue

Instr	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1
sub	p5	y	p2	y	p8	2
addi	p8	n	"1"	y	p9	3

Ready Table

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	n
p9	n

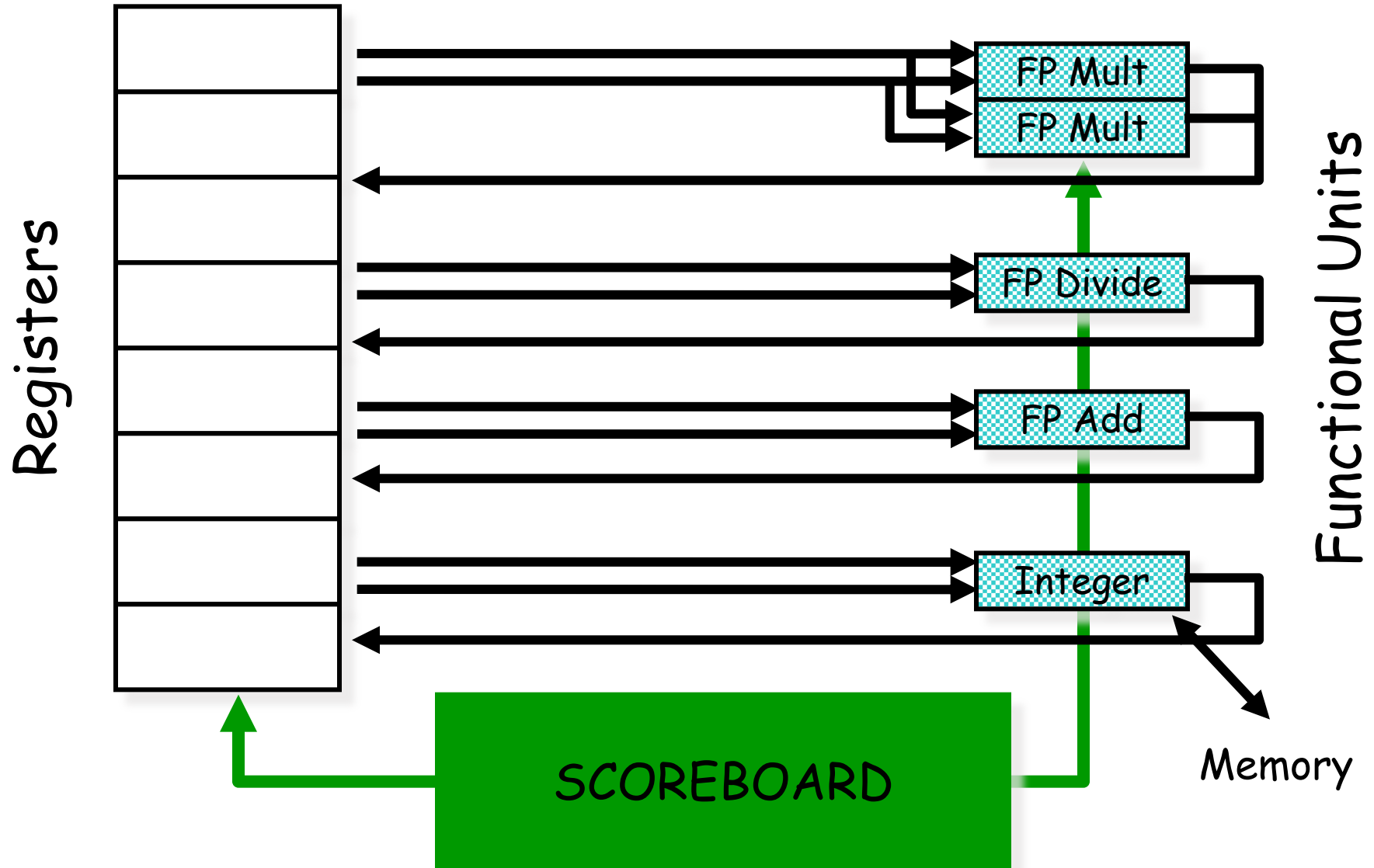
Summary: Tomasulo's Algorithm

- If reservation station available before renaming
 - Instruction + renamed operands (registers) inserted into the reservation station
 - Only rename if reservation station is available
- Else stall
- While in reservation station, each instruction:
 - Watches common data bus (CDB) for tag of its sources
 - When tag seen, grab value for the source and keep it in the reservation station
 - When both operands available, instruction ready to be dispatched
- Dispatch instruction to the Functional Unit when instruction is ready
- After instruction finishes in the Functional Unit
 - Arbitrate for CDB
 - Put tagged value onto CDB (tag broadcast)
 - Register file is connected to the CDB
 - Register contains a tag indicating the latest writer to the register
 - If the tag in the register file matches the broadcast tag, write broadcast value into register (and set valid bit)
 - Reclaim rename tag
 - no valid copy of tag in system!

Other than Tomasulo approach

- “Scoreboarding” technique
 - Also a **dynamic scheduling** scheme
 - Monitors each instruction waiting to be dispatched. Once it determines that all the source operands and the required functional units are available, it dispatches the instruction so that it can be executed.


Scoreboard Architecture(CDC 6600)



Data Dependence Types

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR)

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write
(WAW)

Key idea of Scoreboard

- Allow instructions behind stall to proceed (Decode => Issue instr & read operands)
 - Enables out-of-order execution => **out-of-order completion**
 - D stage checked both for structural & data dependencies
 - Original version (CDC 6600) did not handle forwarding.
 - No automatic register renaming

Scoreboard Implications

- **Out-of-order completion** => WAR, WAW data hazards?
- Solutions for WAR:
 - Stall writeback until registers have been read
 - Read registers only during Read Operands stage
- Solution for WAW:
 - Detect hazard and stall issue of new instruction until other instruction completes
- **No register renaming!**
- Need to have multiple instructions in execution phase => multiple execution units or pipelined execution units
- Scoreboard keeps track of dependencies between instructions that have **already issued**.
- Scoreboard replaces D, X, W with 4 stages...see next slide

Four Stages of Scoreboard Control

- **Issue**—decode instructions & check for structural hazards (D1 stage)
 - Instructions issued in program order (for hazard checking)
 - Don't issue if **structural hazard**
 - Don't issue if instruction is **output dependent** on any previously issued but uncompleted instruction (no WAW hazards)
- **Read operands**—wait until no data hazards, then read operands (D2 stage)
 - All real dependencies (RAW hazards) resolved in this stage, since we wait for instructions to write back data.
 - **No forwarding of data** in this model!

Four Stages of Scoreboard Control

- **Execution**—operate on operands (X)
 - The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.
- **Write result**—finish execution (W)
 - Stall until no WAR hazards with previous instructions:

Example:

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

CDC 6600 scoreboard would stall SUBD until ADDD reads operands

OoO execution summary: What we learned

Basic idea of out-of-order execution:

- In-order F, D, W; out-of-order X (and M)
- Tomasulo approach: Components added to pipeline:
 - Before X: reservation station
 - Before W: reorder buffer
- Register renaming
 - Eliminates false dependencies (those not true dependencies)

Must know

- How to draw pipeline table, when out-of-order execution is supported
- How to draw pipeline table, when register renaming is supported
- What are components added to pipeline to support out-of-order execution