

Digital Design & Computer Arch.

Lecture 16: Out-of-Order Execution

Prof. Onur Mutlu

ETH Zürich
Spring 2022
28 April 2022

Roadmap for Today (and Past Two Weeks)

- Prior to last week: Microarchitecture Fundamentals
 - Single-cycle Microarchitectures
 - Multi-cycle Microarchitectures
- Last week: Pipelining & Precise Exceptions
 - Pipelining
 - Pipelined Processor Design
 - Control & Data Dependence Handling
 - Precise Exceptions: State Maintenance & Recovery
- Today: Out-of-Order Execution
 - Out-of-Order Execution
 - Issues in OoO Execution: Load-Store Handling, ...

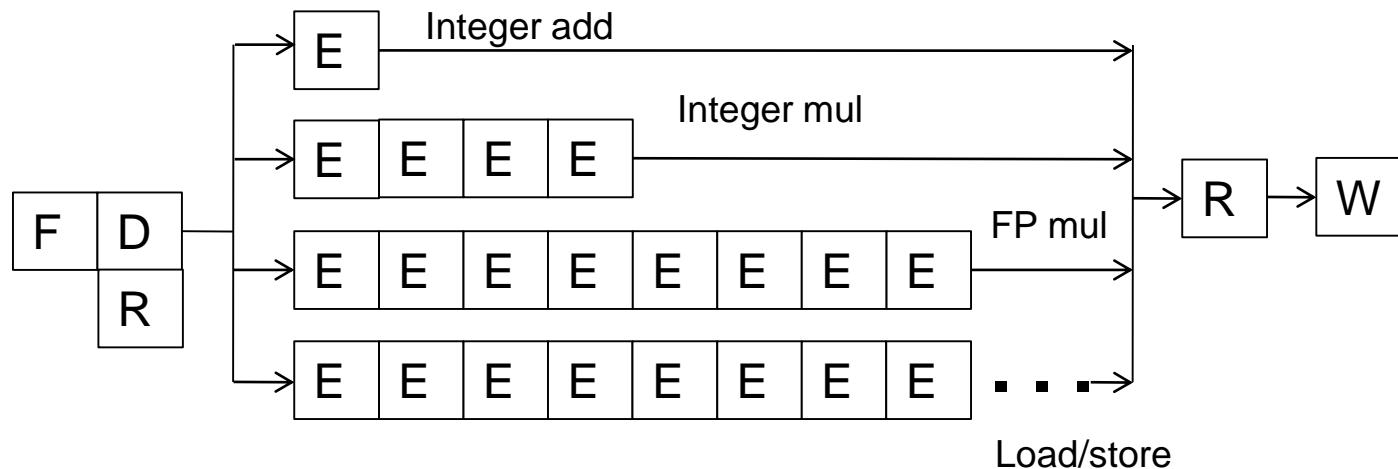
Problem
Algorithm
Program/Language
System Software
SW/HW Interface
Micro-architecture
Logic
Devices
Electrons

Readings

- This week
 - Out-of-order execution
 - H&H, Chapter 7.8-7.9
 - Smith and Sohi, “**The Microarchitecture of Superscalar Processors**,” Proceedings of the IEEE, 1995
 - More advanced pipelining
 - Interrupt and exception handling
 - Out-of-order and superscalar execution concepts
- Optional
 - Kessler, “**The Alpha 21264 Microprocessor**,” IEEE Micro 1999.
- Tomorrow & Next Week
 - McFarling, “**Combining Branch Predictors**,” DEC WRL Technical Report, 1993.

Review: In-Order Pipeline with Reorder Buffer

- **Decode (D)**: Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction (send to functional unit)
- **Execute (E)**: Instructions can complete out-of-order
- **Completion (R)**: Write result **to reorder buffer**
- **Retirement/Commit (W)**: Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**



Recall: Data Dependence Types

Flow dependence

$$\begin{array}{l} r_3 \leftarrow r_1 \text{ op } r_2 \\ r_5 \leftarrow r_3 \text{ op } r_4 \end{array}$$

Read-after-Write
(RAW)

Anti dependence

$$\begin{array}{l} r_3 \leftarrow r_1 \text{ op } r_2 \\ r_1 \leftarrow r_4 \text{ op } r_5 \end{array}$$

Write-after-Read
(WAR)

Output dependence

$$\begin{array}{l} r_3 \leftarrow r_1 \text{ op } r_2 \\ r_5 \leftarrow r_3 \text{ op } r_4 \\ r_3 \leftarrow r_6 \text{ op } r_7 \end{array}$$

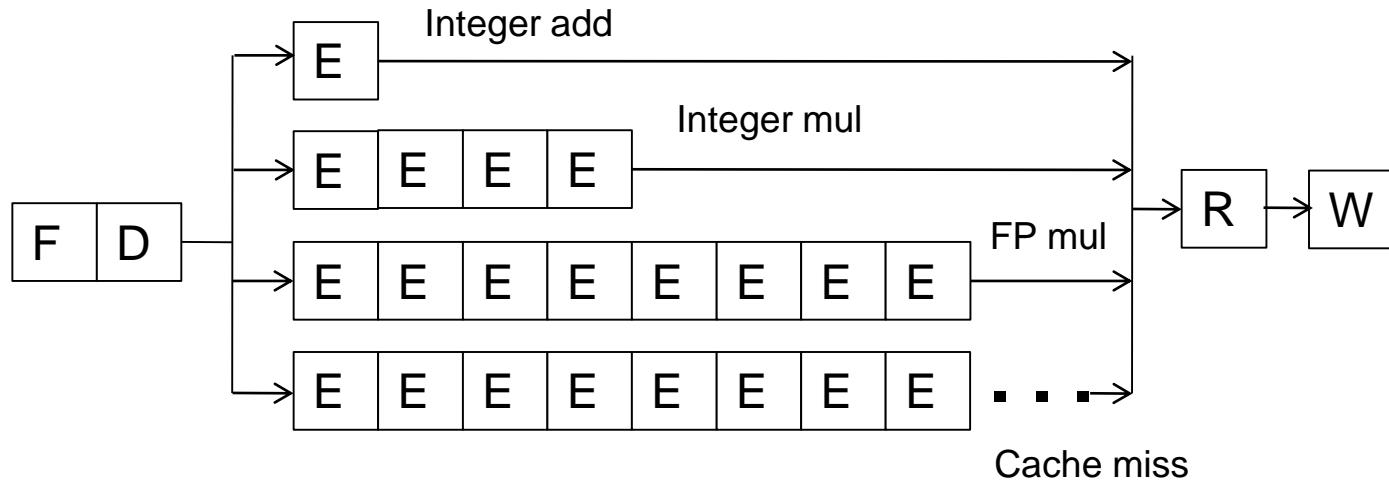
Write-after-Write
(WAW)

Recall: Register Renaming with a Reorder Buffer

- Output and anti dependences are **not true dependences**
 - WHY? The same register refers to values that have nothing to do with each other
 - **They exist due to lack of register ID's (i.e. names) in the ISA**
- The register ID is **renamed** to the reorder buffer entry that will hold the register's value
 - Register ID → ROB entry ID
 - Architectural register ID → Physical register ID
 - After renaming, ROB entry ID used to refer to the register
- This eliminates anti and output dependences
 - Gives the illusion that there are a large number of registers

Out-of-Order Execution (Dynamic Instruction Scheduling)

An In-order Pipeline



- Dispatch: Act of sending an instruction to a functional unit
- Renaming with ROB eliminates stalls due to false dependences
- Problem: A true data dependence stalls dispatch of younger instructions into functional (execution) units

An Example of True Data Dependence



An Example of True Data Dependence



Time: 12:57

An Example of True Data Dependence



Time: 12:58

An Example of True Data Dependence



Time: 13:00

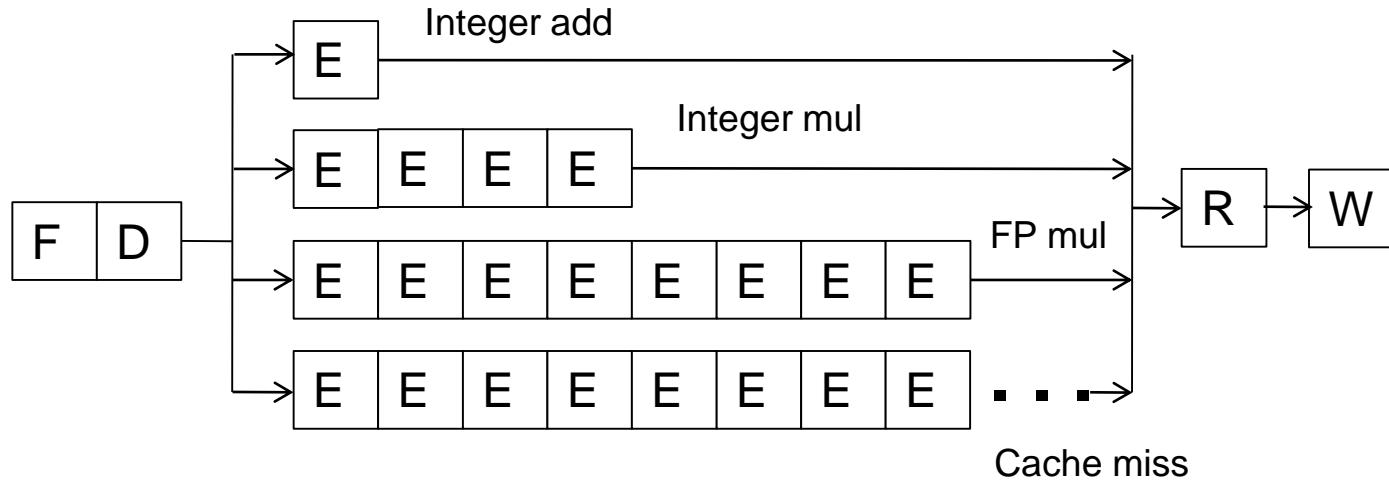
Another View



Stalling Done & Independents Execute



An In-order Pipeline



- Dispatch: Act of sending an instruction to a functional unit
- Renaming with ROB eliminates stalls due to false dependences
- **Problem: A true data dependence stalls dispatch of younger instructions into functional (execution) units**

Can We Do Better?

How Can We Do Better?

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

MUL R3 \leftarrow R1, R2
ADD R3 \leftarrow R3, R1
ADD R4 \leftarrow R6, R7
MUL R5 \leftarrow R6, R8
ADD R7 \leftarrow R9, R9

LD R3 \leftarrow R1 (0)
ADD R3 \leftarrow R3, R1
ADD R4 \leftarrow R6, R7
MUL R5 \leftarrow R6, R8
ADD R7 \leftarrow R9, R9

- Answer: First ADD stalls the whole pipeline!
 - ADD cannot dispatch because its source register 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

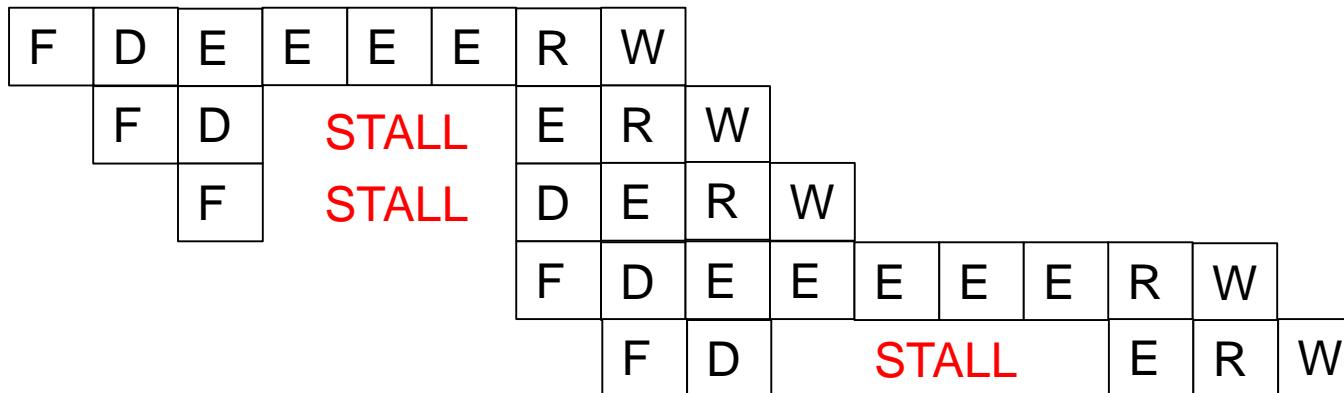
- Problem: **in-order** dispatch (scheduling, or execution)
- Solution: **out-of-order** dispatch (scheduling, or execution)
- Actually, we have seen the basic idea before:
 - **Dataflow**: “fire” an instruction only when its inputs are ready
 - We will use similar principles, but not expose it in the ISA
- Aside: Any other way to prevent dispatch stalls?
 1. Compile-time instruction scheduling/reordering
 2. Value prediction
 3. Fine-grained multithreading

Out-of-order Execution (Dynamic Scheduling)

- Idea: Move the dependent instructions out of the way of independent ones (s.t. independent ones can execute)
 - Rest areas for dependent instructions: Reservation stations
- Monitor the source “values” of each instruction in the resting (waiting) area
- When all source “values” of an instruction are available, “fire” (i.e., dispatch) the instruction
 - Instructions dispatched in dataflow (not control-flow) order
- Benefit:
 - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long-latency operation

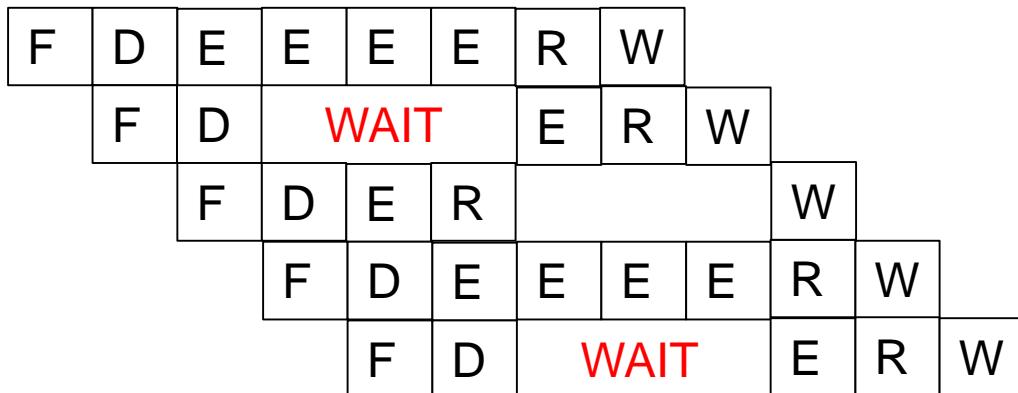
In-order vs. Out-of-order Dispatch

- 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

- Out-of-order dispatch + precise exceptions:



- 16 vs. 12 cycles

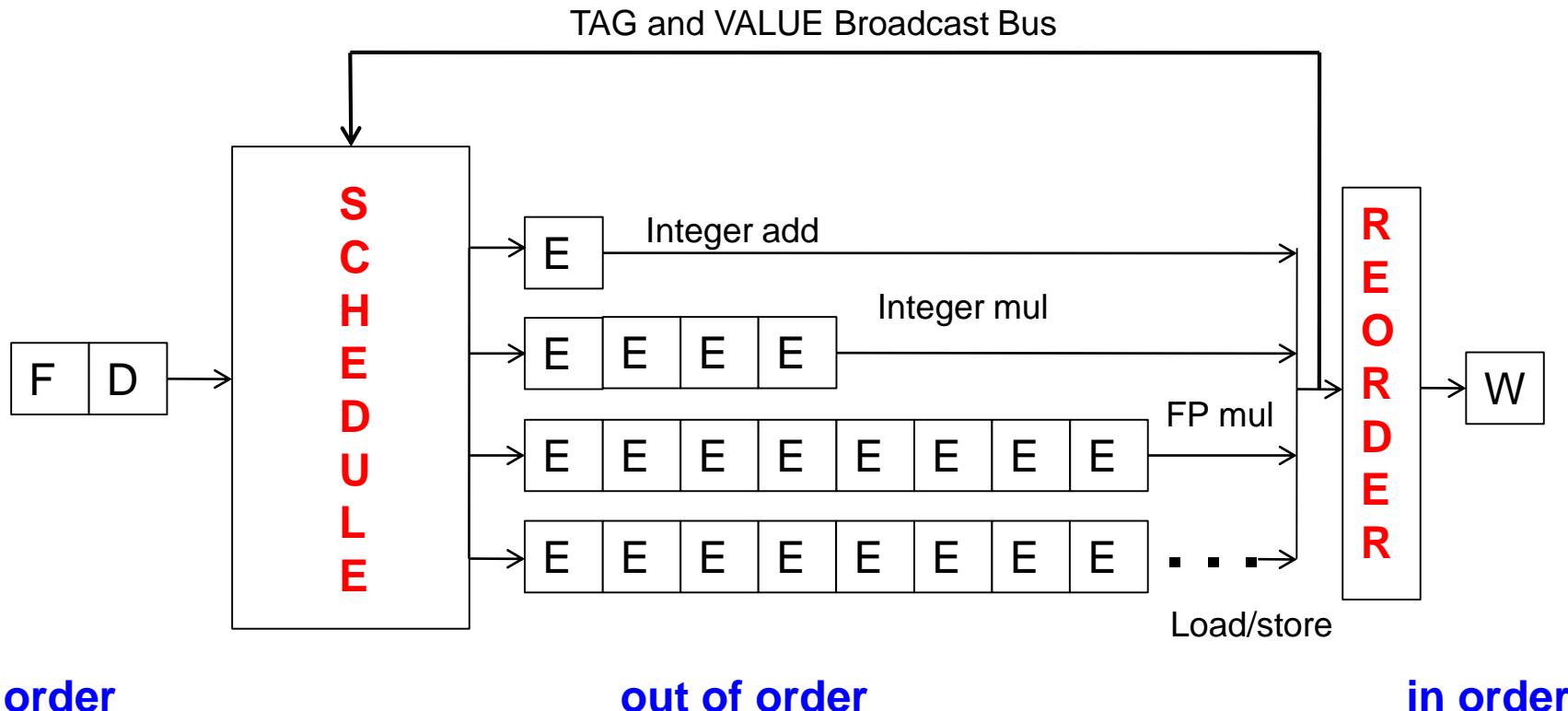
Enabling OoO Execution

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 FU

Tomasulo's Algorithm for OoO Execution

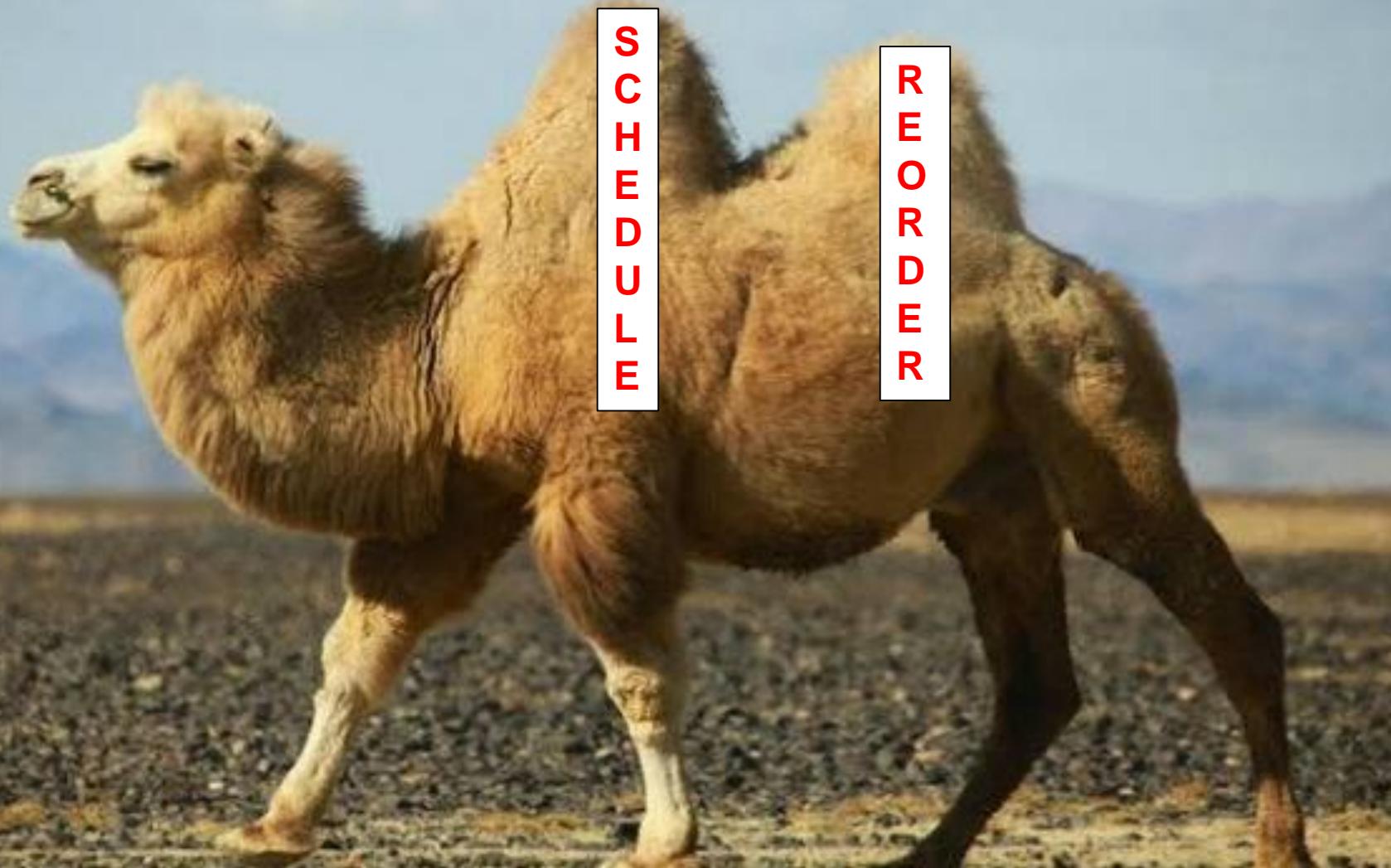
- OoO with register renaming invented by Robert Tomasulo
 - Used in IBM 360/91 Floating Point Units
 - **Reading:** Tomasulo, “**An Efficient Algorithm for Exploiting Multiple Arithmetic Units,**” IBM Journal of R&D, Jan. 1967.
 - What is the major difference today?
 - Precise exceptions
 - Provided by
 - Patt, Hwu, Shebanow, “**HPS, a new microarchitecture: rationale and introduction,**” MICRO 1985.
 - Patt et al., “**Critical issues regarding HPS, a high performance microarchitecture,**” MICRO 1985.
 - OoO variants are used in most high-performance processors
 - Initially in Intel Pentium Pro, AMD K5
 - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15, Apple M1, ...
-

Two Humps in a Modern Pipeline

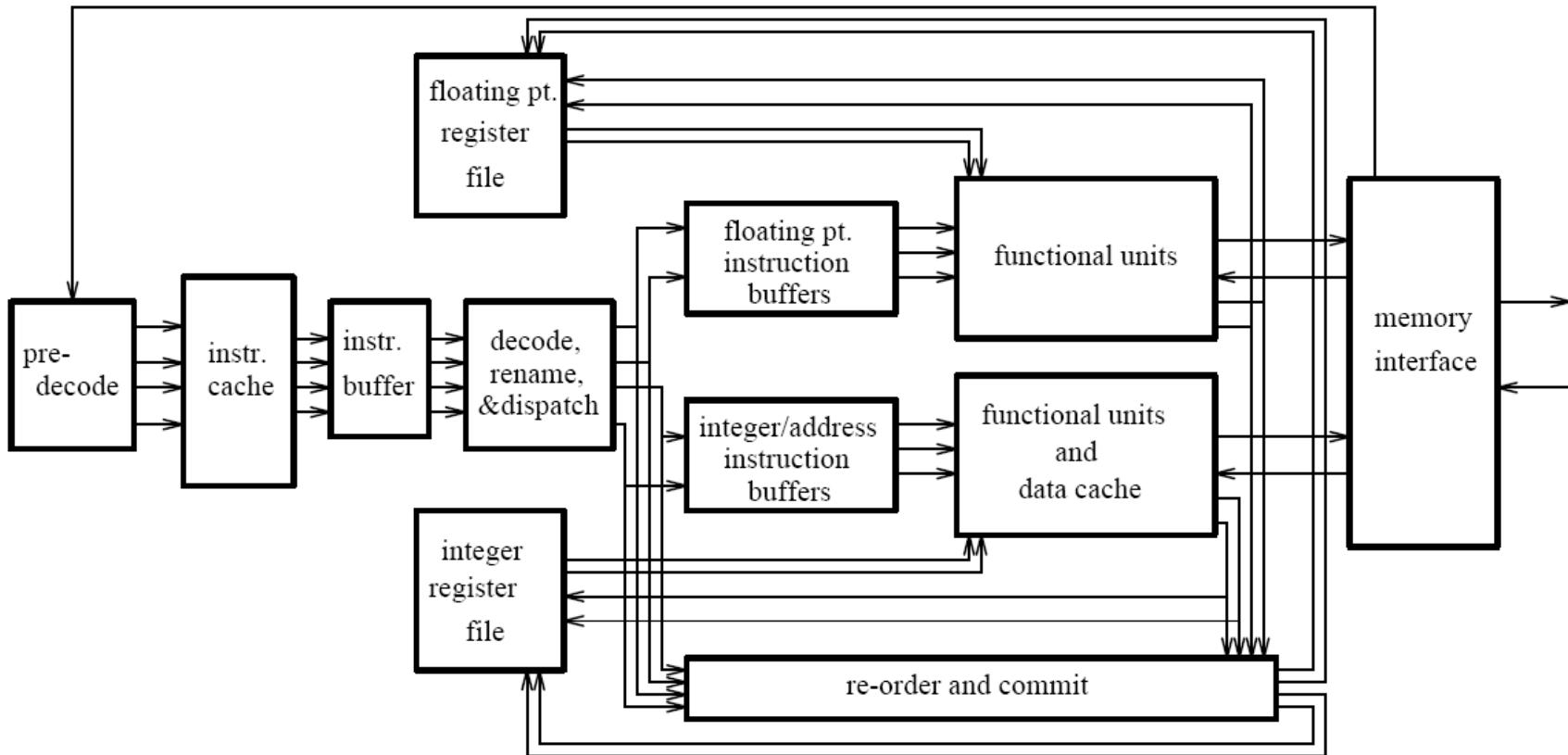


- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

Two Humps in a Modern Pipeline

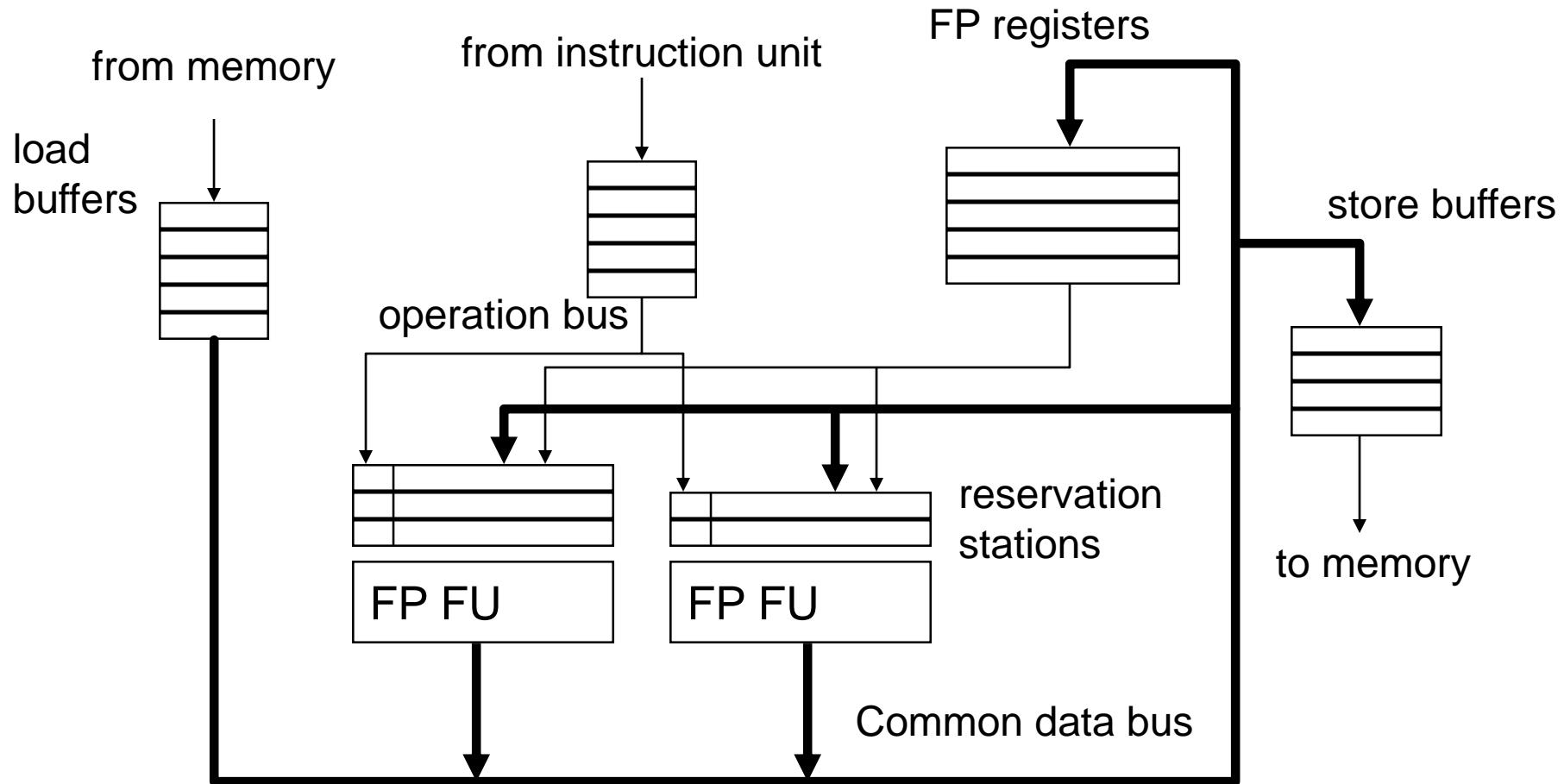


General Organization of an OOO Processor



- Smith and Sohi, “**The Microarchitecture of Superscalar Processors**,” Proc. IEEE, Dec. 1995.

Tomasulo's Machine: IBM 360/91



IBM 360/91 in Real World



IBM 360/91 in Real World



Recall Once More: Register Renaming

- Output and anti dependences are not true dependences
 - WHY? The same register refers to values that have nothing to do with each other
 - **They exist because not enough register ID's (i.e. names) in the ISA**
- The register ID is **renamed** to the reservation station entry that will hold the register's value
 - Register ID → RS entry ID
 - Architectural register ID → Physical register ID
 - After renaming, RS entry ID used to refer to the register
- This eliminates anti- and output- dependences
 - Approximates the performance effect of a large number of registers even though ISA has a small number

Tomasulo's Algorithm: Renaming

- 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

If Valid bit is set, the Value in the table is correct.
Otherwise, Tag specifies where to find the correct value.
Tag is a unique name for the Value to be produced.

Recall from Precise Exceptions Lecture

Register File (RF)

R0		
R1		
R2		
R3		
R4		
R5		
R6		
R7		

Value Tag
(pointer to
ROB entry)

Reorder Buffer (ROB)

Entry 0				
Entry 1				
Entry 2				
Entry 8				
Entry 13				
Entry 14				
Entry 15				

Entry Valid?
Dest reg ID
Dest reg value
Dest reg written?

Oldest instruction
Youngest instruction

This Lecture

Register File (RF) or Register Alias Table (RAT)

R0		
R1		
R2		
R3		
R4		
R5		
R6		
R7		

Value Valid? Value Tag
**(pointer to the
reservation station entry
that will produce the value)**

We will ignore Reorder Buffer for simplicity

Tomasulo's Algorithm

- If reservation station available before renaming
 - Instruction + renamed operands (source value/tag) 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!

An Exercise

MUL R3 \leftarrow R1, R2

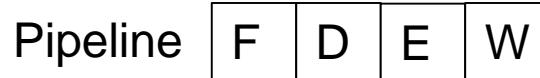
ADD R5 \leftarrow R3, R4

ADD R7 \leftarrow R2, R6

ADD R10 \leftarrow R8, R9

MUL R11 \leftarrow R7, R10

ADD R5 \leftarrow R5, R11



- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
 - in a non-pipelined machine: **50 cycles** ($4*7 + 2*11$)
 - in an in-order-dispatch pipelined machine with imprecise exceptions (no forwarding and forwarding)
 - in an out-of-order dispatch pipelined machine imprecise exceptions (forwarding)

Exercise Continued

FD123456W

FD- - - - D1234W

F- - - - - D1234W

FD1234W

FD- - - D123456W

F- - - - D

D1234W

in-order-dispatch pipelined machine
w/o forwarding: 31 cycles



Execution timeline w/ scoreboard

31 cycles

FD123456W

FD → E, D1234W

F D 1 2 3 4 W

FD 1 2 3 4 W

FD → 1 2 3 4, 6 W

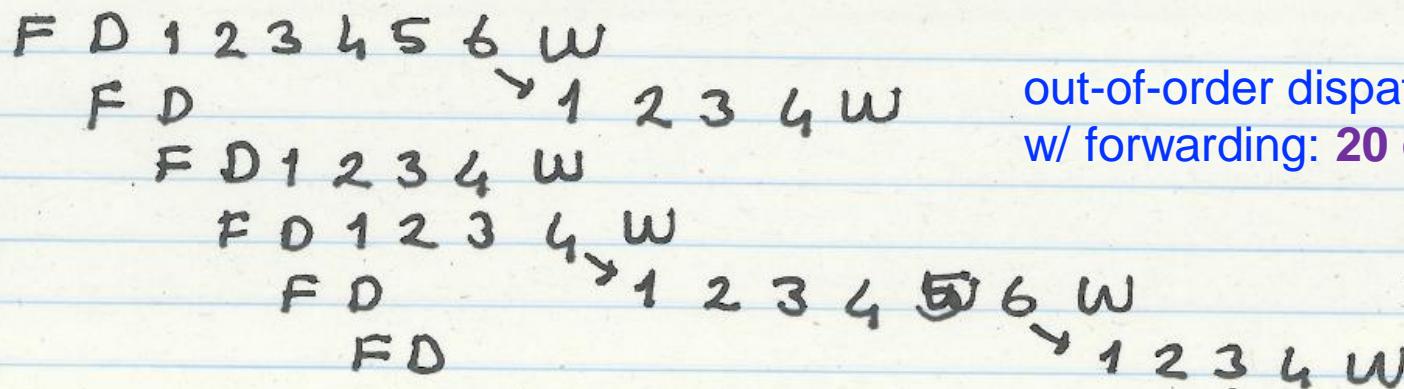
F D 1 2 3 4 W

in-order-dispatch pipelined machine
w/ forwarding: 25 cycles

25 cycles

Exercise Continued

MUL R3 \leftarrow R1, R2
ADD R5 \leftarrow R3, R4
ADD R7 \leftarrow R2, R6
ADD R10 \leftarrow R8, R9
MUL R11 \leftarrow R7, R10
ADD R5 \leftarrow R5, R11



out-of-order dispatch pipelined machine
w/ forwarding: **20 cycles**

Tomasulo's algorithm + full forwarding

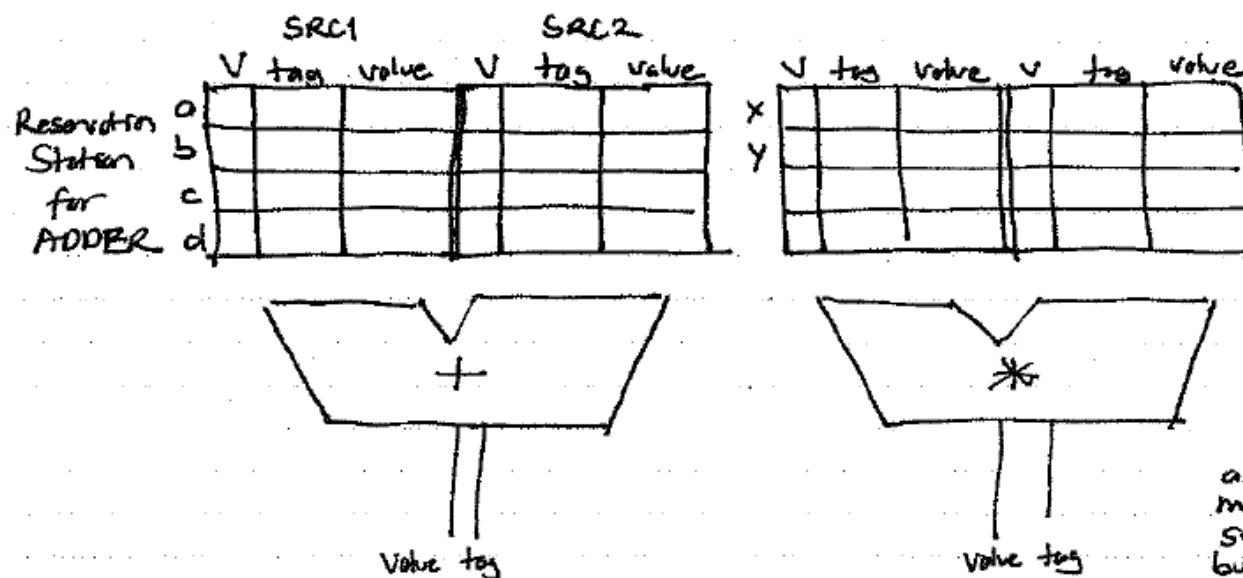
20 cycles

How It Works

Register Alias Table

V	Tag	Value

of writer



Our First OoO Machine Simulation

Program We Will Simulate

```
MUL R1, R2 → R3  
ADD R3, R4 → R5  
ADD R2, R6 → R7  
ADD R8, R9 → R10  
MUL R7, R10 → R11  
ADD R5, R11 → R5
```

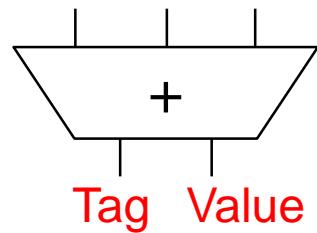
Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Initially:

1. Reservation Stations (RS's) are all Invalid (Empty)
2. All Registers are Valid

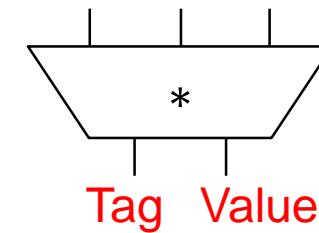
RS for ADD Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						



Register Alias Table

ADD and MUL Execution Units
have separate Tag & Value buses

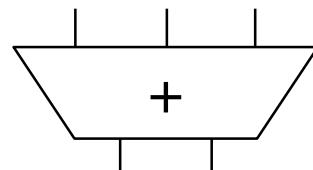
Cycle 0

Cycle

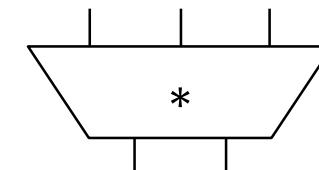
MUL	R1, R2	→	R3
ADD	R3, R4	→	R5
ADD	R2, R6	→	R7
ADD	R8, R9	→	R10
MUL	R7, R10	→	R11
ADD	R5, R11	→	R5

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						



Cycle 1

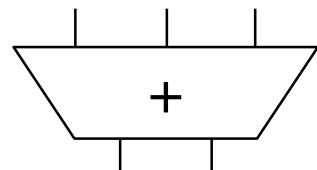
Cycle 1

```
MUL R1, R2 → R3
ADD R3, R4 → R5
ADD R2, R6 → R7
ADD R8, R9 → R10
MUL R7, R10 → R11
ADD R5, R11 → R5
```

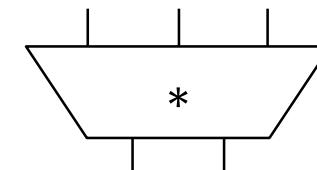
F

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						



Cycle 2

		Cycle 1	Cycle 2
MUL	R1, R2 → R3	F	D
ADD	R3, R4 → R5	F	F
ADD	R2, R6 → R7		
ADD	R8, R9 → R10		
MUL	R7, R10 → R11		
ADD	R5, R11 → R5		

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

MUL gets decoded and allocated into RS x

Step 1: Check if reservation station available. Yes: x

Step 2: Access the Register Alias Table

Step 3: Put source registers into reservation station x

Step 4: Rename destination register R3 → x

R3 is now renamed to x.
Its new value will be produced by the *reservation station* that is identified by tag x.

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		

Cycle 3

1. MUL in RS x starts executing

2. ADD gets decoded and allocated into RS a

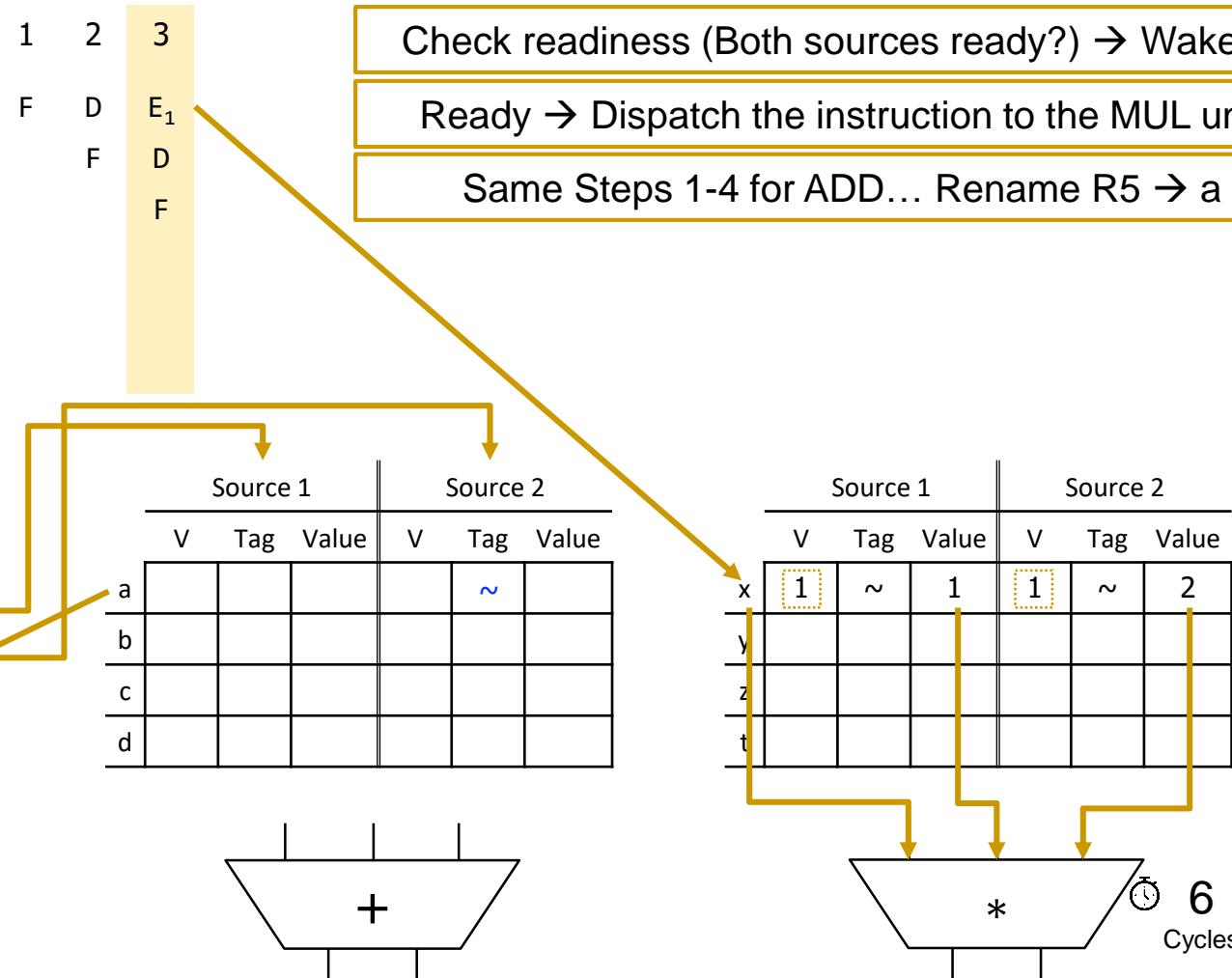
	Cycle		
	1	2	3
MUL	R1, R2 → R3	F	E ₁
ADD	R3, R4 → R5	D	D
ADD	R2, R6 → R7	F	F
ADD	R8, R9 → R10		
MUL	R7, R10 → R11		
ADD	R5, R11 → R5		

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Check readiness (Both sources ready?) → Wakeup

Ready → Dispatch the instruction to the MUL unit

Same Steps 1-4 for ADD... Rename R5 → a



ADD in RS a cannot execute in the next cycle: one source is not valid

Cycle 4

		Cycle	1	2	3	4
MUL	R1, R2 → R3	F	D	E ₁	E ₂	
ADD	R3, R4 → R5	F	D	-		
ADD	R2, R6 → R7	F	D			
ADD	R8, R9 → R10					F
MUL	R7, R10 → R11					
ADD	R5, R11 → R5					

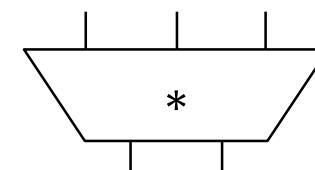
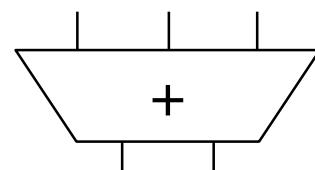
ADD in RS a waits because one source is not valid.

Rename R7 → b

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b		~			~	
c						
d						

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y						
z						
t						



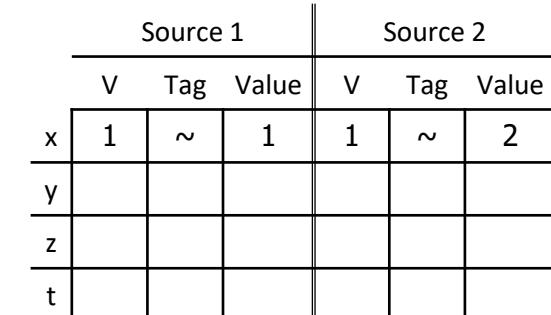
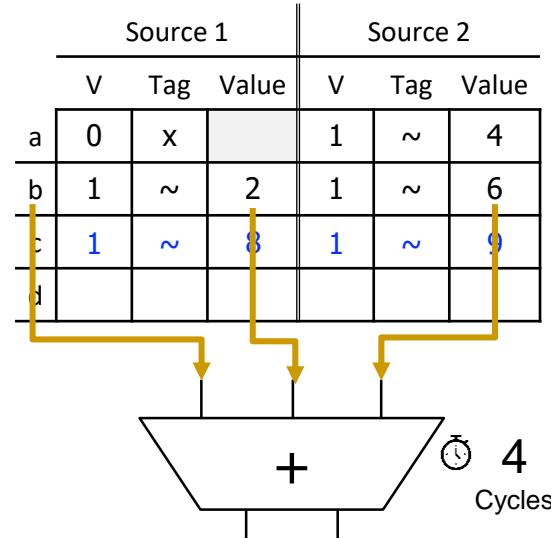
ADD in RS b is ready to execute in the next cycle!

It will be executed out of order in the next cycle.

Cycle 5

	Cycle	1	2	3	4	5
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	
ADD R3, R4 → R5	F	D	-	-		
ADD R2, R6 → R7		F	D	E ₁		
ADD R8, R9 → R10		F	D			
MUL R7, R10 → R11			F			
ADD R5, R11 → R5						

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	1		11



ADD in RS c is ready to execute in the next cycle!

Cycle 6

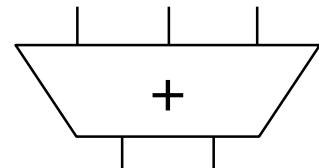
Cycle 1 2 3 4 5 6

MUL	R1, R2	\rightarrow	R3
ADD	R3, R4	\rightarrow	R5
ADD	R2, R6	\rightarrow	R7
ADD	R8, R9	\rightarrow	R10
MUL	R7, R10	\rightarrow	R11
ADD	R5, R11	\rightarrow	R5

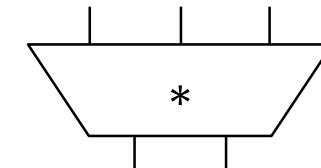
F	D	E_1	E_2	E_3	E_4
F	D	-	-	-	-
F	D	E_1	E_2		
F	D		E_1		
F	D				F

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

Source 1			Source 2		
V	Tag	Value	V	Tag	Value
a	0	x	1	\sim	4
b	1	\sim	2	1	\sim
c	1	\sim	8	1	\sim
d					



Source 1			Source 2		
V	Tag	Value	V	Tag	Value
x	1	\sim	1	\sim	2
y	0	b	0	c	
z					
t					



Cycle 7

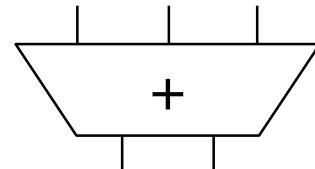
All six instructions are now decoded and renamed

Note what happened to R5: Renamed twice!

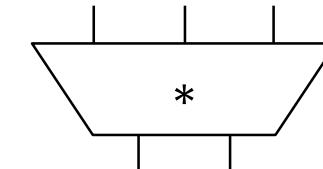
		Cycle	1	2	3	4	5	6	7
MUL	R1, R2	\rightarrow	R3	F	D	E_1	E_2	E_3	E_4
ADD	R3, R4	\rightarrow	R5	F	D	-	-	-	-
ADD	R2, R6	\rightarrow	R7	F	D	E_1	E_2	E_3	
ADD	R8, R9	\rightarrow	R10	F	D	E_1	E_2		
MUL	R7, R10	\rightarrow	R11	F	D	-			
ADD	R5, R11	\rightarrow	R5	F	D				

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

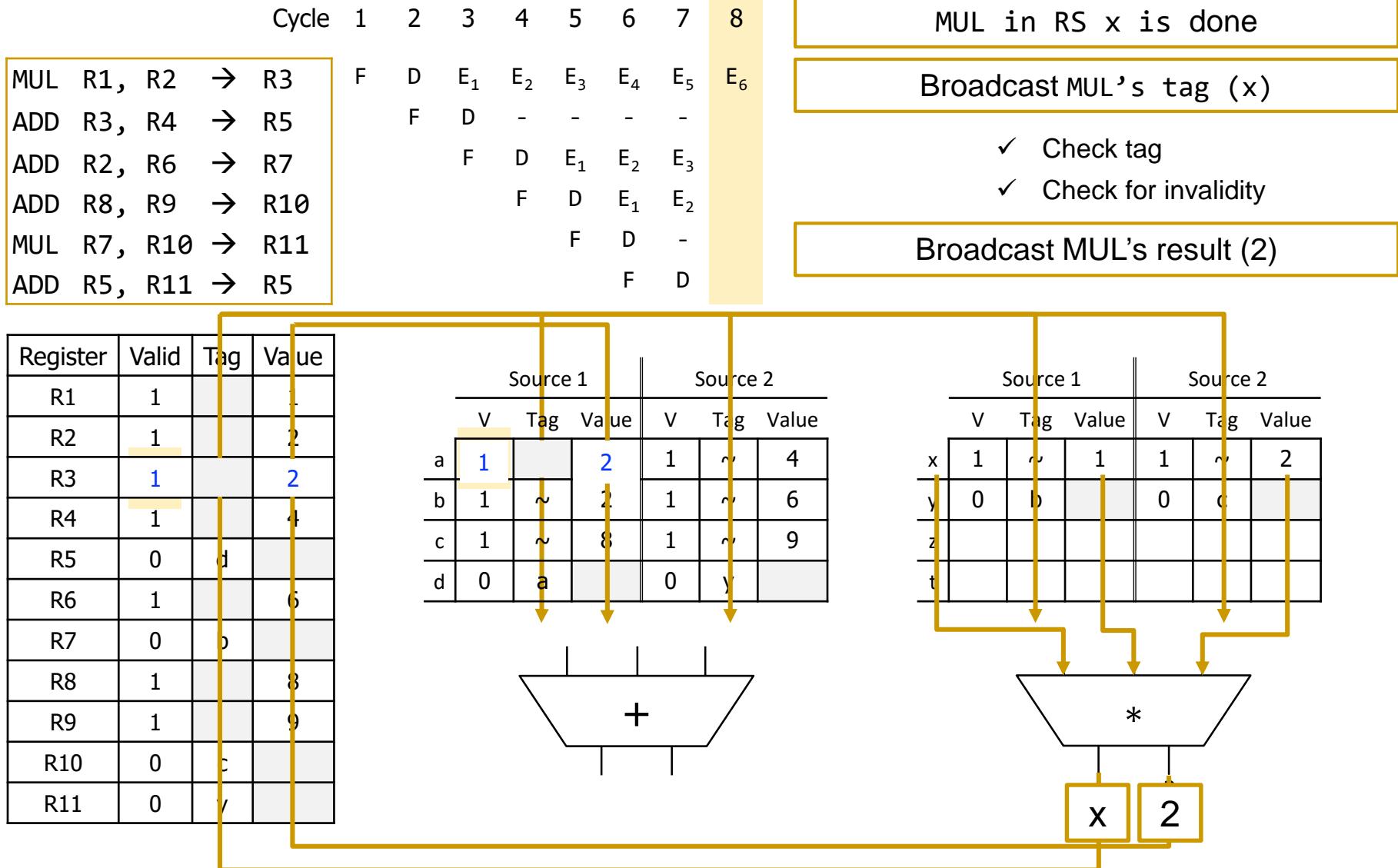
Source 1			Source 2		
V	Tag	Value	V	Tag	Value
a	0	x	1	~	4
b	1	~	2	1	~
c	1	~	8	1	~
d	0	a	0	y	



Source 1			Source 2		
V	Tag	Value	V	Tag	Value
x	1	~	1	~	2
y	0	b	0	c	
z					
t					



Cycle 8 (First Slide)



Cycle 8 (Second Slide)

		Cycle	1	2	3	4	5	6	7	8
MUL	R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	
ADD	R3, R4 → R5	F	D	-	-	-	-	-	-	
ADD	R2, R6 → R7	F	D	E ₁	E ₂	E ₃	E ₄			
ADD	R8, R9 → R10	F	D	E ₁	E ₂					
MUL	R7, R10 → R11	F	D	-						
ADD	R5, R11 → R5	F	D							

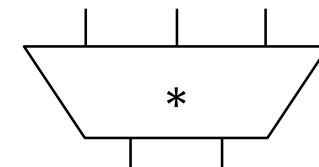
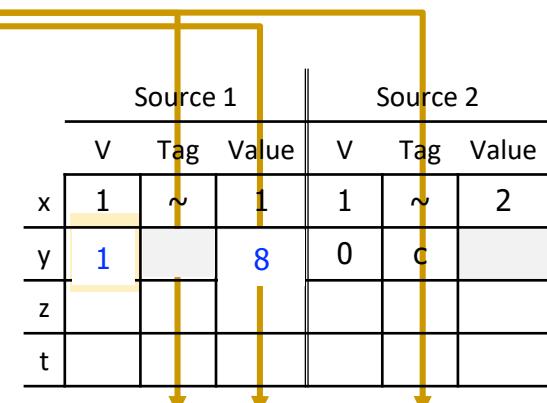
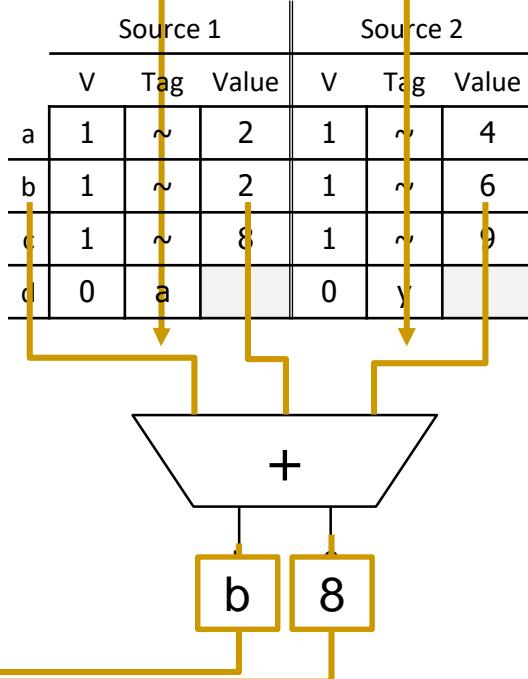
ADD in RS b is also done

Broadcast ADD's tag (b)

- ✓ Check tag
- ✓ Check for invalidity

Broadcast ADD's result (8)

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	0	
R6	1		6
R7	1	8	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	



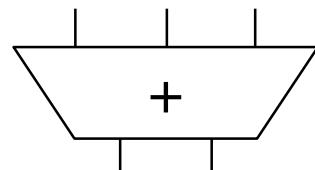
MUL in RS y is still NOT ready to execute in the next cycle!

Cycle 8 (Third Slide)

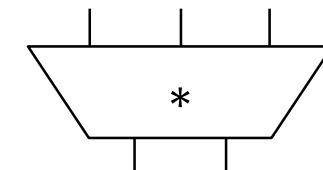
		Cycle	1	2	3	4	5	6	7	8	
MUL	R1, R2	\rightarrow	R3	F	D	E_1	E_2	E_3	E_4	E_5	E_6
ADD	R3, R4	\rightarrow	R5	F	D	-	-	-	-	-	-
ADD	R2, R6	\rightarrow	R7		F	D	E_1	E_2	E_3	E_4	
ADD	R8, R9	\rightarrow	R10		F	D	E_1	E_2	E_3		
MUL	R7, R10	\rightarrow	R11		F	D	-	-			
ADD	R5, R11	\rightarrow	R5		F	D	-				

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	\sim	2	1	\sim	4
b	1	\sim	2	1	\sim	6
c	1	\sim	8	1	\sim	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	\sim	1	1	\sim	2
y	1	\sim	8	0	c	
z						
t						



Cycle 9

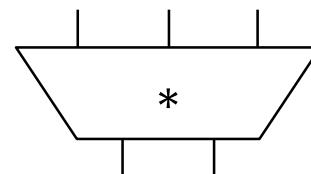
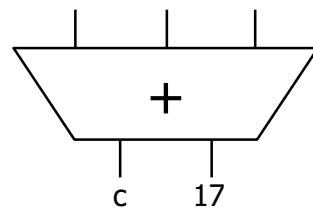
	Cycle	1	2	3	4	5	6	7	8	9
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W	
ADD R3, R4 → R5	F	D	-	-	-	-	-	-	E ₁	
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄		W	
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃		E ₄		
MUL R7, R10 → R11		F	D	-	-	-				
ADD R5, R11 → R5		F	D	-	-					

Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



MUL in RS y is ready to execute in the next cycle!

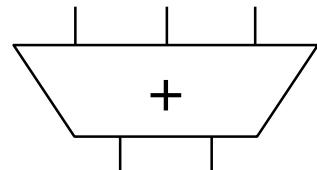
Cycle 10

		Cycle	1	2	3	4	5	6	7	8	9	10
MUL	R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W		
ADD	R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂		
ADD	R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W			
ADD	R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W			
MUL	R7, R10 → R11		F	D	-	-	-	-	E ₁			
ADD	R5, R11 → R5		F	D	-	-	-	-	-			

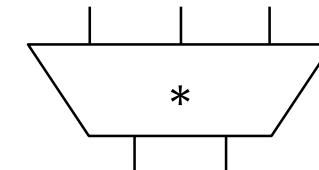
	Cycle	1	2	3	4	5	6	7	8	9	10
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W			
F	D	-	-	-	-	-	-	E ₁	E ₂		
F	D	E ₁	E ₂	E ₃	E ₄	W					
F	D	E ₁	E ₂	E ₃	E ₄	W					
F	D	-	-	-	-	E ₁					
F	D	-	-	-	-	-					

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						

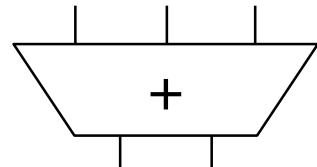


Cycle 11

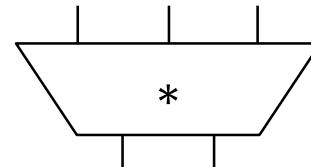
	Cycle	1	2	3	4	5	6	7	8	9	10	11
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W			
ADD R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂		E ₃	
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W				
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W				
MUL R7, R10 → R11		F	D	-	-	-	E ₁				E ₂	
ADD R5, R11 → R5		F	D	-	-	-	-	-			-	

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



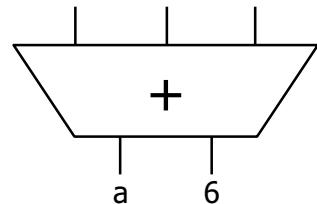
Cycle 12

	Cycle	1	2	3	4	5	6	7	8	9	10	11	12
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W				E ₄
ADD R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂	E ₃			
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W					
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W					
MUL R7, R10 → R11		F	D	-	-	-	E ₁	E ₂	E ₃				
ADD R5, R11 → R5		F	D	-	-	-	-	-	-	-			

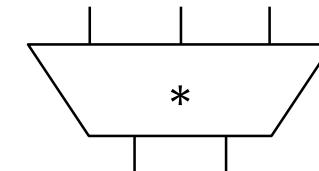
Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

Source 1			Source 2		
V	Tag	Value	V	Tag	Value
a	1	~	2	1	~
b	1	~	2	1	~
c	1	~	8	1	~
d	1	~	6	0	y



Source 1			Source 2		
V	Tag	Value	V	Tag	Value
x	1	~	1	1	~
y	1	~	8	1	~
z					
t					

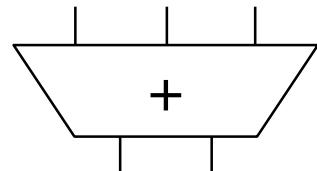


Cycle 13

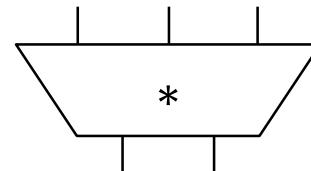
	Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W					
ADD R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W		
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W						
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W						
MUL R7, R10 → R11		F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	W			
ADD R5, R11 → R5		F	D	-	-	-	-	-	-	-	-	-	-	

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	0	y	



Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						

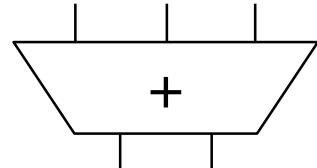


Cycle 14

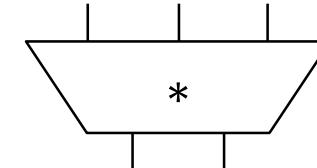
	Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W						
ADD R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W			
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W							
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W							
MUL R7, R10 → R11		F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅				
ADD R5, R11 → R5		F	D	-	-	-	-	-	-	-	-	-			

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

Source 1			Source 2		
V	Tag	Value	V	Tag	Value
a	1	~	2	1	~
b	1	~	2	1	~
c	1	~	8	1	~
d	1	~	6	0	y



Source 1			Source 2		
V	Tag	Value	V	Tag	Value
x	1	~	1	1	~
y	1	~	8	1	~
z					
t					



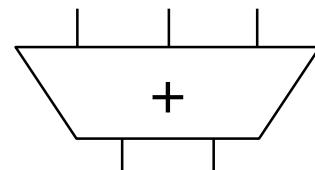
Cycle 15

	Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W							
ADD R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W				
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W								
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W								
MUL R7, R10 → R11		F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆				
ADD R5, R11 → R5		F	D	-	-	-	-	-	-	-	-	-	-	-	-	

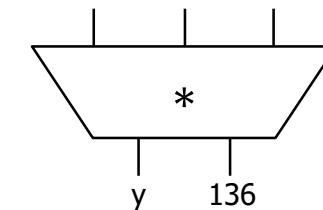
Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

Source 1			Source 2		
V	Tag	Value	V	Tag	Value
a	1	~	2	1	~
b	1	~	2	1	~
c	1	~	8	1	~
d	1	~	6	1	~
					136



Source 1			Source 2		
V	Tag	Value	V	Tag	Value
x	1	~	1	1	~
y	1	~	8	1	~
z					
t					



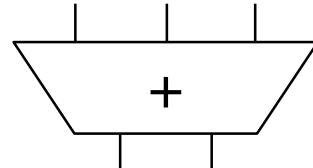
ADD in RS d is ready to execute in the next cycle!

Cycle 16

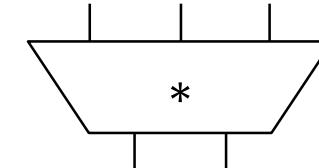
	Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W								
ADD R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W					
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W									
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W									
MUL R7, R10 → R11		F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W				
ADD R5, R11 → R5		F	D	-	-	-	-	-	-	-	-	-	E ₁				

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						

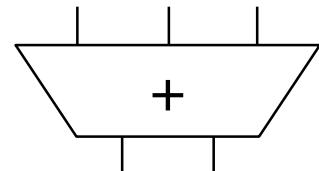


Cycle 17

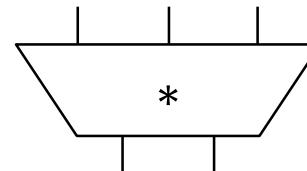
	Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W									
ADD R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W						
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W										
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W										
MUL R7, R10 → R11		F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W					
ADD R5, R11 → R5		F	D	-	-	-	-	-	-	-	-	-	E ₁	E ₂				

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						

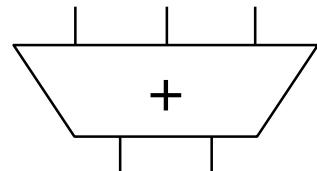


Cycle 18

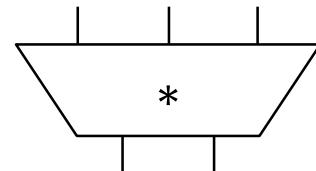
	Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W										
ADD R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W							
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W											
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W											
MUL R7, R10 → R11		F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W						
ADD R5, R11 → R5		F	D	-	-	-	-	-	-	-	-	E ₁	E ₂	E ₃					

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						

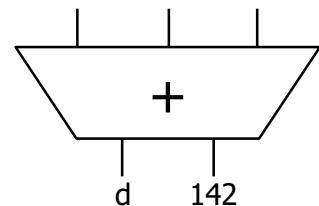


Cycle 19

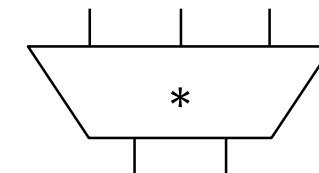
	Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W											
ADD R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W								
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W												
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W												
MUL R7, R10 → R11		F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W							
ADD R5, R11 → R5		F	D	-	-	-	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄					

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	1		142
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



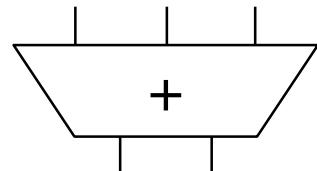
Cycle 20

Cycle 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

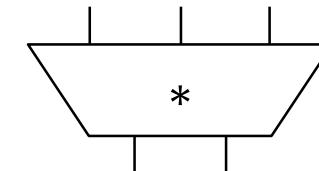
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W									
ADD R3, R4 → R5	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W						
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃	E ₄	W										
ADD R8, R9 → R10		F	D	E ₁	E ₂	E ₃	E ₄	W										
MUL R7, R10 → R11		F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W					
ADD R5, R11 → R5		F	D	-	-	-	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W		

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	1		142
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



Source 1			Source 2			
V	Tag	Value	V	Tag	Value	
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



Some Questions

- What is needed in hardware to perform tag broadcast and value capture?
 - make a value valid
 - wake up an instruction
- Does the tag have to be the ID of the Reservation Station Entry?
 - No, could be any unique name that enables linking of producer to consumer
- What can potentially become the critical path?
 - **Tag broadcast → value capture → instruction wake up**
- How can you reduce the potential critical paths?
 - **More pipelining and prediction**

Dataflow Graph for Our Example

```
MUL R3 ← R1, R2
ADD R5 ← R3, R4
ADD R7 ← R2, R6
ADD R10 ← R8, R9
MUL R11 ← R7, R10
ADD R5 ← R5, R11
```

Easy task for you: **Draw the dataflow graph for the above code**

State of RAT and RS in Cycle 7

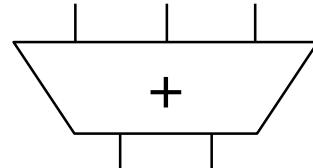
	Cycle	1	2	3	4	5	6	7	
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅		All 6 instructions are decoded and renamed
ADD R3, R4 → R5	F	D	-	-	-	-	-		Note what happened to R5: Renamed twice!
ADD R2, R6 → R7		F	D	E ₁	E ₂	E ₃			
ADD R8, R9 → R10		F	D	E ₁	E ₂				
MUL R7, R10 → R11		F	D	-					
ADD R5, R11 → R5		F	D						

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

Register Alias Table

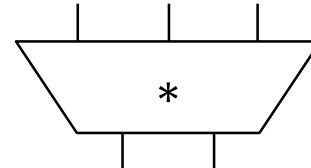
RS for ADD Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						



State of RAT and RS in Cycle 7

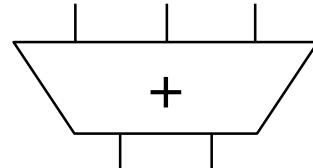
Slightly harder tasks for you:

1. Draw the dataflow graph for the executing code
2. Provide the executing code in sequential order

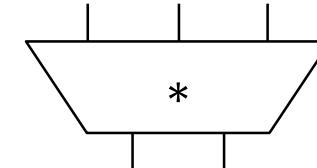
Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

Register Alias Table

Source 1	Source 2		
	V	Tag	Value
	V	Tag	Value
a	0	x	
b	1	~	2
c	1	~	8
d	0	a	



Source 1	Source 2		
	V	Tag	Value
	V	Tag	Value
x	1	~	1
y	0	b	
z			
t			



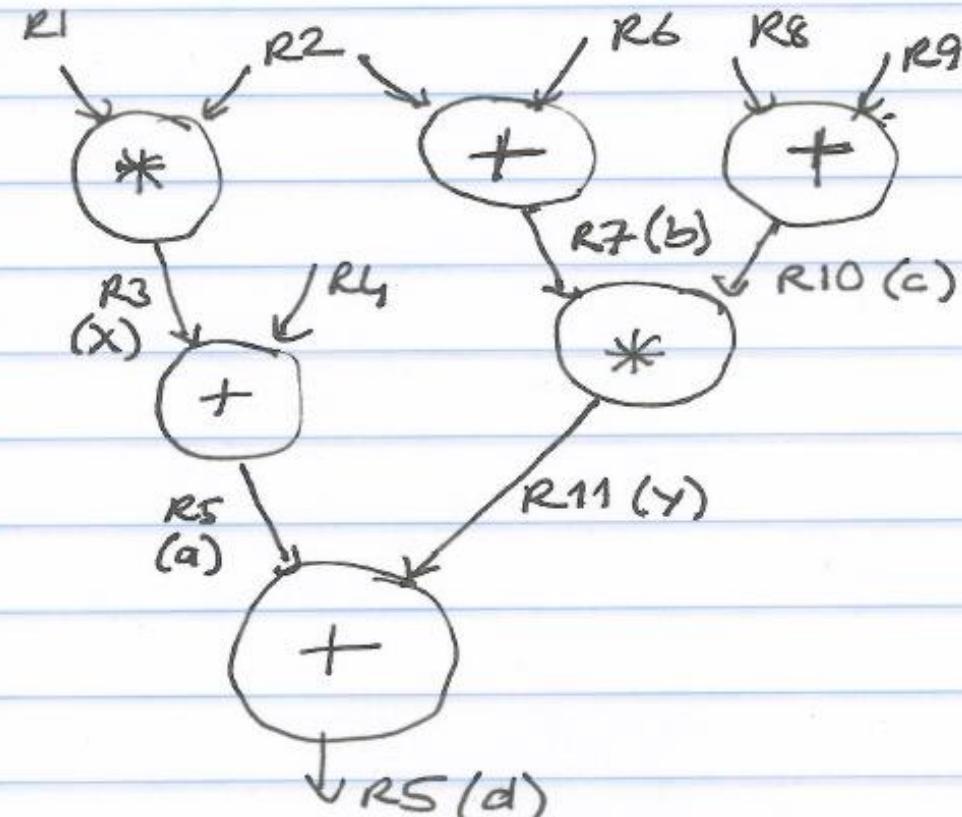
Corresponding Dataflow Graph (Reverse Engineered)

MUL R1, R2 → R3 (x)
ADD R3, R4 → R5 (a)
ADD R2, R6 → R7 (b)
ADD R8, R9 → R10 (c)
MUL R7, R10 → R11 (y)
ADD R5, R11 → R5 (d)

Dataflow graph

Nodes: operations performed by the instruction

Arcs: tags in Tomasulo's algorithm



We can “easily” reverse-engineer the dataflow graph of the executing code!

Some More Questions (Design Choices)

- When is a reservation station entry deallocated?
 - Should the reservation stations be dedicated to each functional unit or global across functional units?
 - Centralized vs. Distributed: What are the tradeoffs?
 - Should reservation stations and ROB store data values or should there be a centralized physical register file where all data values are stored?
 - What are the tradeoffs?
 - Timing: Exactly when does an instruction broadcast its tag?
 - Many other design choices for OoO engines
-

Recall: Our Exercise (We Did This!)

MUL R3 \leftarrow R1, R2

ADD R5 \leftarrow R3, R4

ADD R7 \leftarrow R2, R6

ADD R10 \leftarrow R8, R9

MUL R11 \leftarrow R7, R10

ADD R5 \leftarrow R5, R11



- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
 - in a non-pipelined machine: 50 cycles ($4*7 + 2*11$)
 - in an in-order-dispatch pipelined machine with imprecise exceptions (no forwarding and forwarding)
 - in an out-of-order dispatch pipelined machine imprecise exceptions (forwarding)

For You: An Exercise, w/ Precise Exceptions

MUL R3 ← R1, R2

ADD R5 ← R3, R4

ADD R7 ← R2, R6

ADD R10 ← R8, R9

MUL R11 ← R7, R10

ADD R5 ← R5, R11

Pipeline F D E R W

ADD F D E E E E R W

MUL F D E E E E E E E R W

- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
 - in an **in-order-dispatch** pipelined machine **with reorder buffer** (no forwarding and full forwarding)
 - in an **out-of-order dispatch** pipelined machine **with reorder buffer** (full forwarding)

Out-of-Order Execution with Precise Exceptions

- **Idea:** Use a reorder buffer to reorder instructions before committing them to architectural state
 - An instruction updates the RAT when it completes execution
 - Also called **frontend register file**
 - An instruction updates a **separate** architectural register file when it retires
 - i.e., when it is the oldest in the machine and has completed execution
 - In other words, **the architectural register file is always updated in program order**
 - On an exception: flush pipeline, copy architectural register file into frontend register file
-

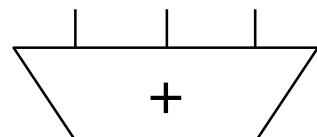
Recall: Our Initial OoO Machine

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register Alias Table

RS for ADD Unit

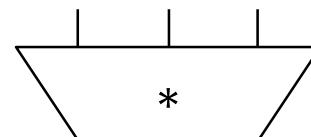
	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



Tag Value

RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						

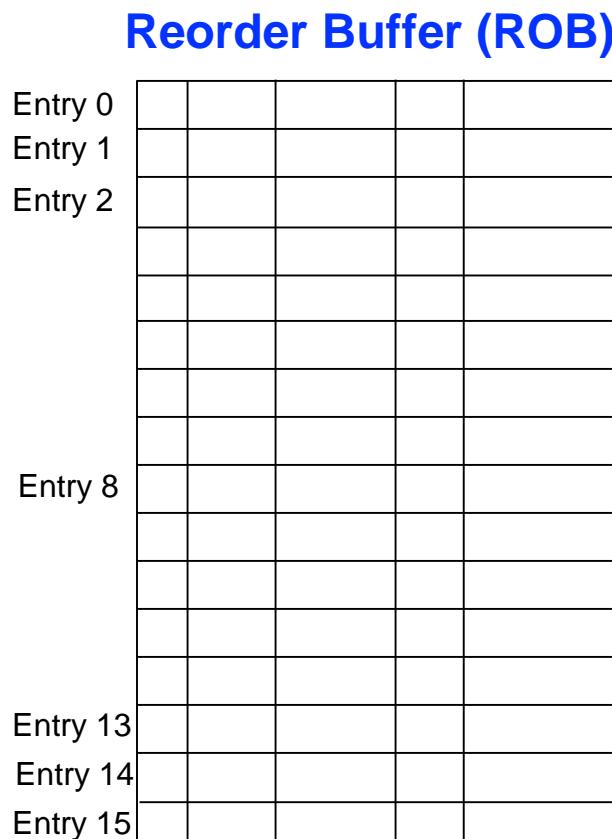


Tag Value

Add Arch Reg File & ROB for Precise Exceptions

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

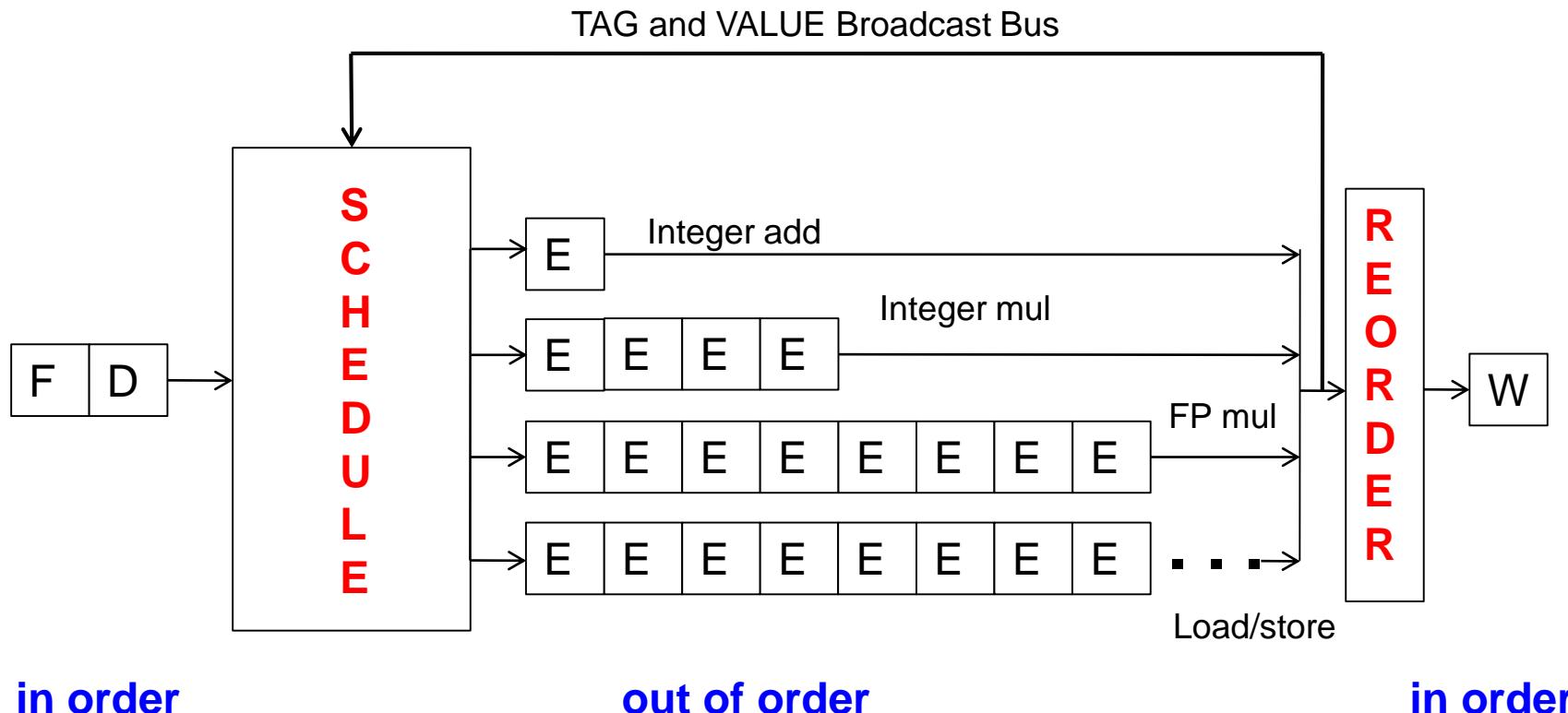
Frontend Register File



Register	Value
R1	1
R2	2
R3	3
R4	4
R5	5
R6	6
R7	7
R8	8
R9	9
R10	10
R11	11

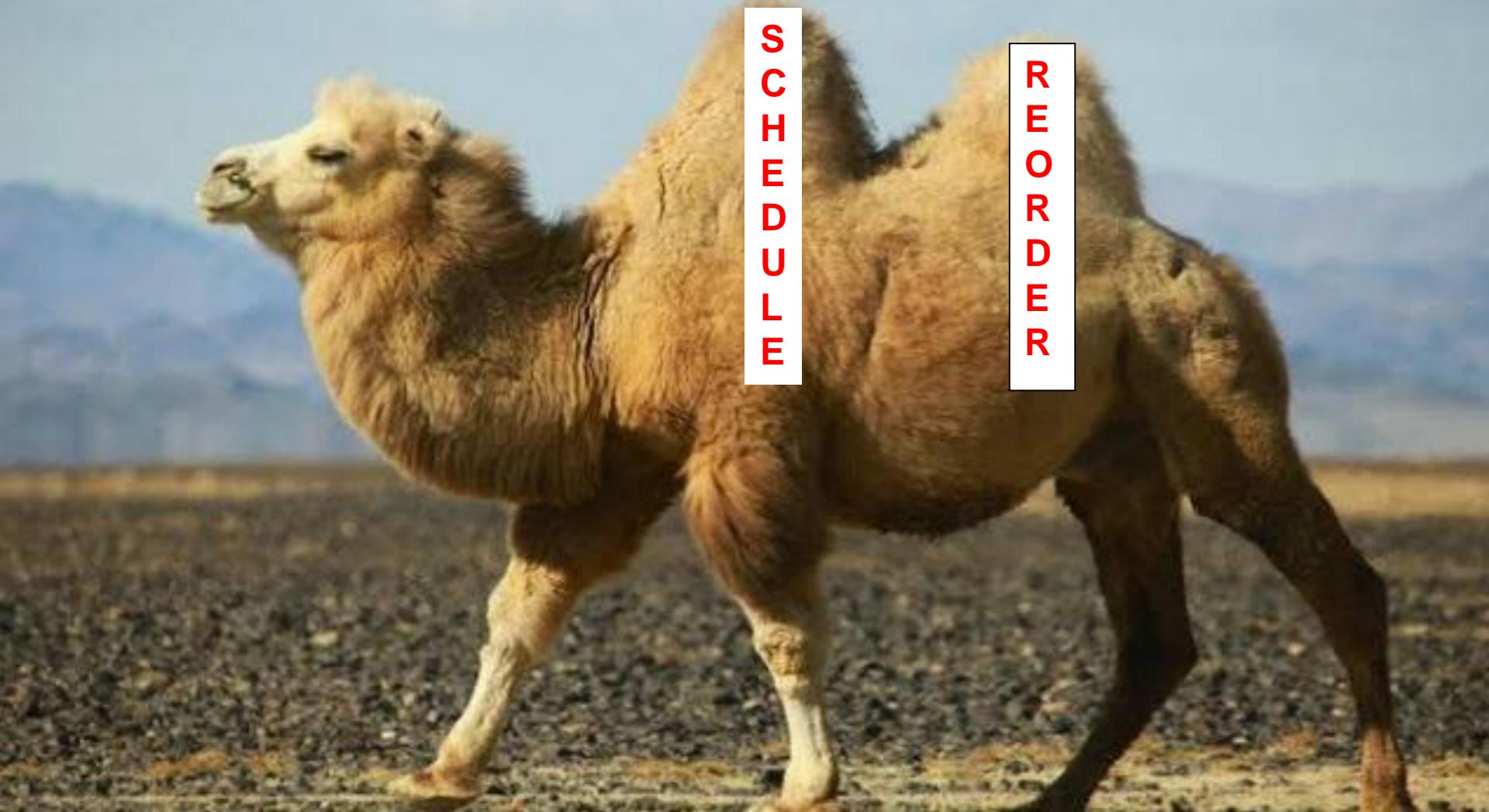
Architectural Register File

Out-of-Order Execution with Precise Exceptions



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

Two Humps in a Modern Pipeline



One Issue: Value Replication All Over the Place

RS for ADD Unit

	Source 1		Source 2			
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

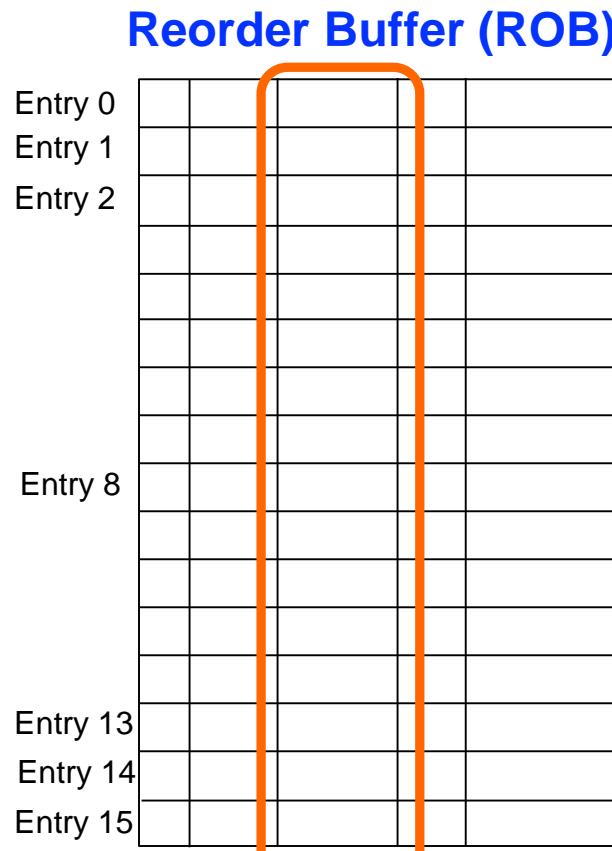
Frontend Register File

RS for MUL Unit

	Source 1		Source 2			
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						

Register	Value
R1	1
R2	2
R3	3
R4	4
R5	5
R6	6
R7	7
R8	8
R9	9
R10	10
R11	11

Architectural Register File



Getting Rid of Replicated Values

Frontend Register Map

Register	PR
R1	18
R2	13
R3	10
R4	22
R5	14
R6	19
R7	17
R8	20
R9	3
R10	4
R11	1

Pointers to PRF

PR	Value
PR1	1
PR2	2
PR3	3
PR4	4
PR5	5
PR6	6
PR7	7
PR8	8
PR9	9
PR10	10
PR11	11
PR12	12
PR13	13
PR14	14
PR15	15
PR16	16
PR17	17
PR18	18
PR19	19
PR20	20
PR21	21
PR22	22

Reorder Buffer (ROB)

Entry 0			
Entry 1			
Entry 2			
Entry 8			
Entry 13			
Entry 14			
Entry 15			

Physical Centralized Register File Value Storage

Architectural Register Map

Register	PR
R1	12
R2	2
R3	10
R4	22
R5	5
R6	9
R7	11
R8	20
R9	7
R10	6
R11	1

Pointers to PRF

Modern OoO Execution w/ Precise Exceptions

- Most modern processors use the following
- Reorder buffer to support in-order retirement of instructions
- A single register file (physical RF) to store **all registers**
 - Both speculative and architectural registers
 - INT and FP are still separate
- Two register **maps** store **pointers** to the physical RF
 - Future/frontend register map → used for renaming
 - Architectural register map → used for maintaining precise state
- This design avoids value replication in RSs, ROB, etc.

Getting Rid of Replicated Values (I)

Pointers
to PRF

Register	PR
R1	18
R2	13
R3	10
R4	22
R5	14
R6	19
R7	17
R8	20
R9	3
R10	4
R11	1

Frontend
Register Map

PR	Value
PR1	1
PR2	2
PR3	3
PR4	4
PR5	5
PR6	6
PR7	7
PR8	8
PR9	9
PR10	10
PR11	11
PR12	12
PR13	13
PR14	14
PR15	15
PR16	16
PR17	17
PR18	18
PR19	19
PR20	20
PR21	21
PR22	22

Physical Centralized
Register Value
File Storage
(PRF)

Reorder Buffer (ROB)

Entry 0			
Entry 1			
Entry 2			
Entry 8			
Entry 13			
Entry 14			
Entry 15			

Pointers
to PRF

Register	PR
R1	12
R2	2
R3	10
R4	22
R5	5
R6	9
R7	11
R8	20
R9	7
R10	6
R11	1

Architectural
Register Map

Getting Rid of Replicated Values (II)

At Decode/Rename: Allocate DestPR to Dest Reg

At Decode/Rename: Read and Update Frontend Register Map

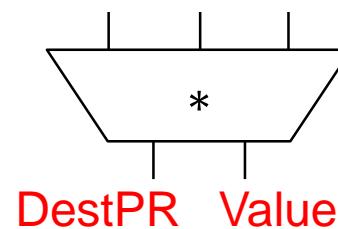
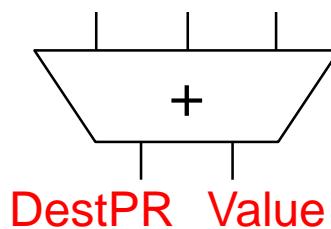
RS for ADD Unit

	Source 1	Source 2
	PR	PR
a		
b		
c		
d		

RS for MUL Unit

	Source 1	Source 2
	PR	PR
a		
b		
c		
d		

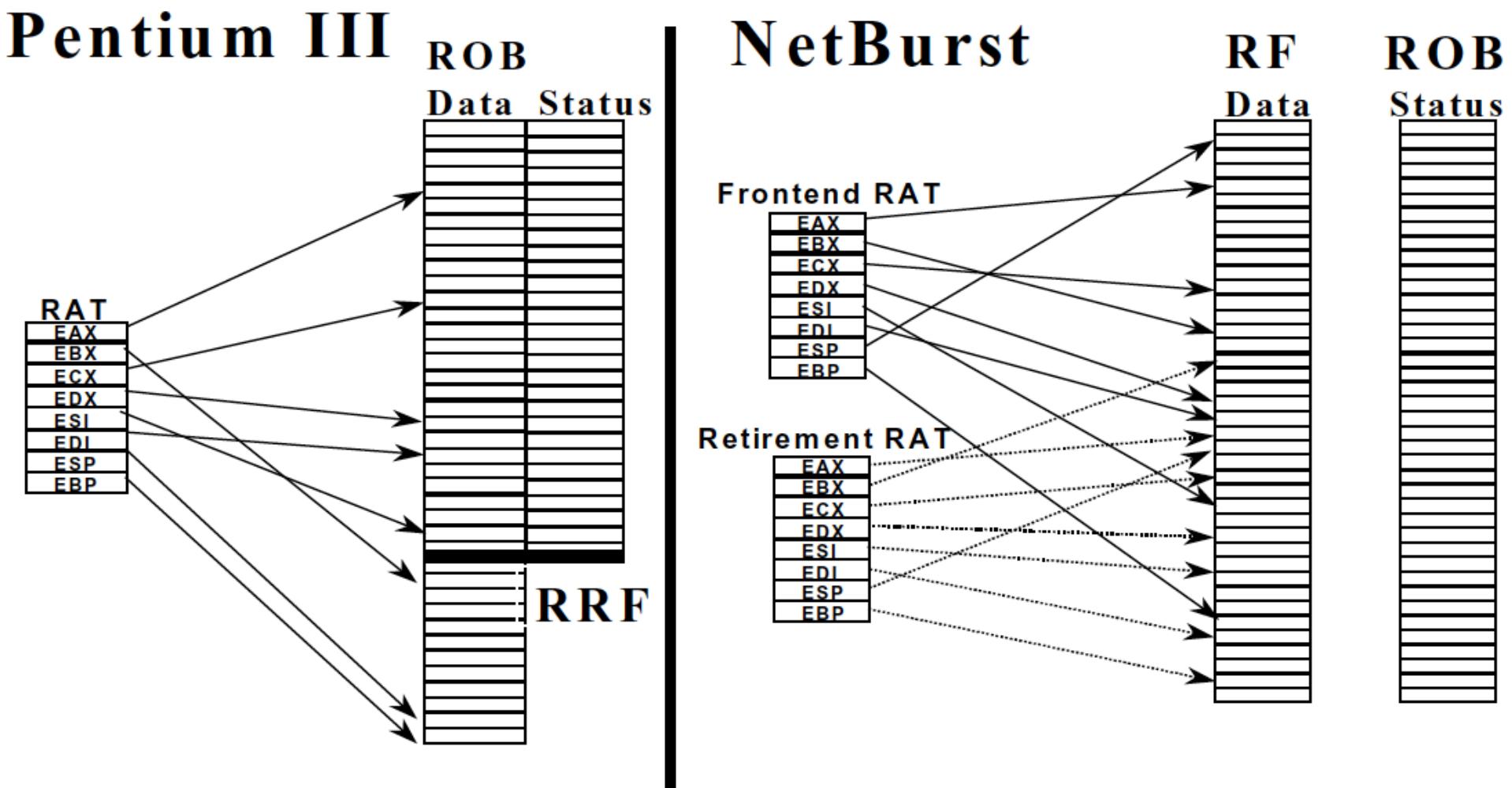
Before Execution: Access Physical Register File to Get Source Values



After Execution: Access Physical Register File to Write Result Values

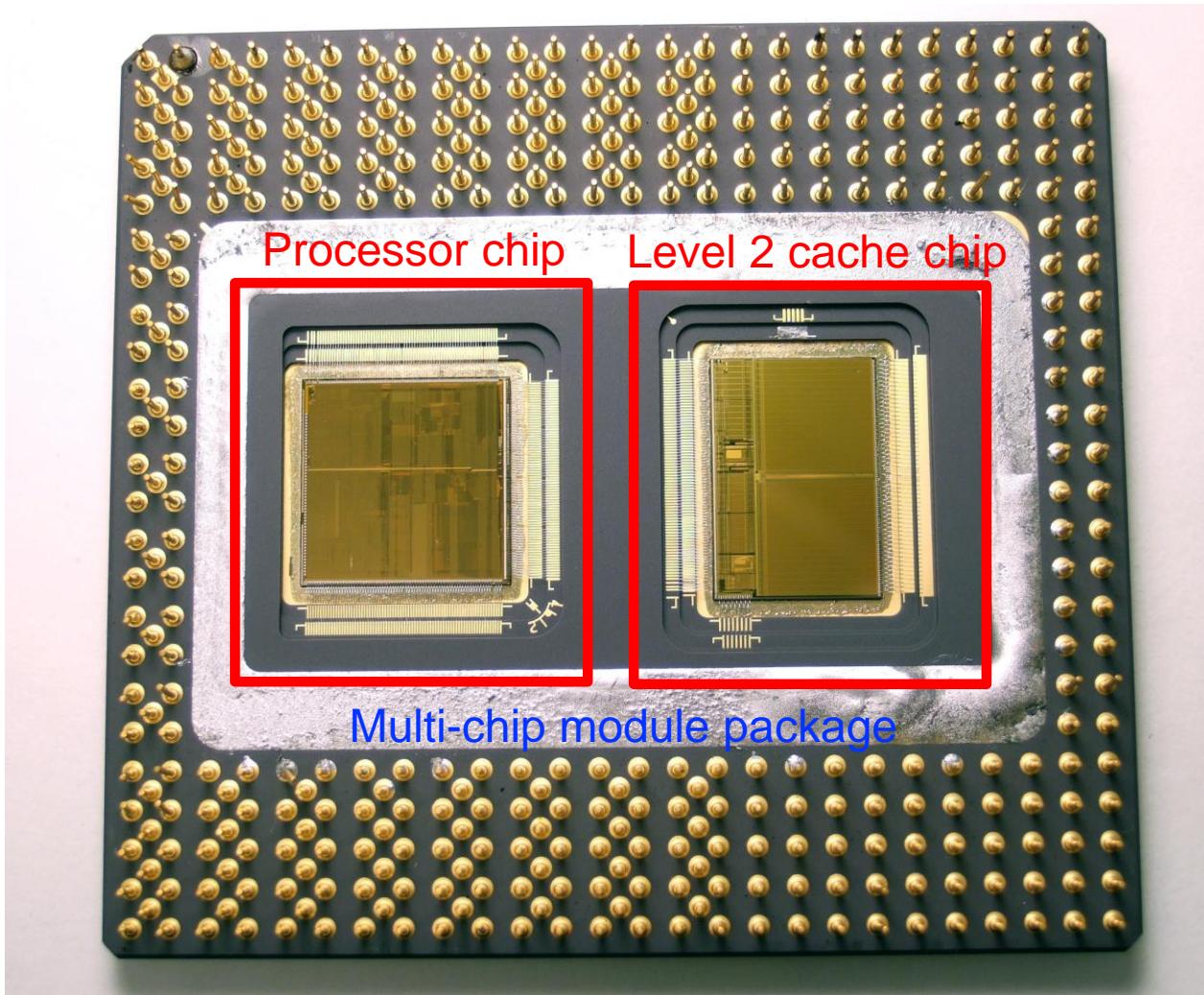
At Retirement : Update Architectural Register Map with DestPR

An Example from Modern Processors

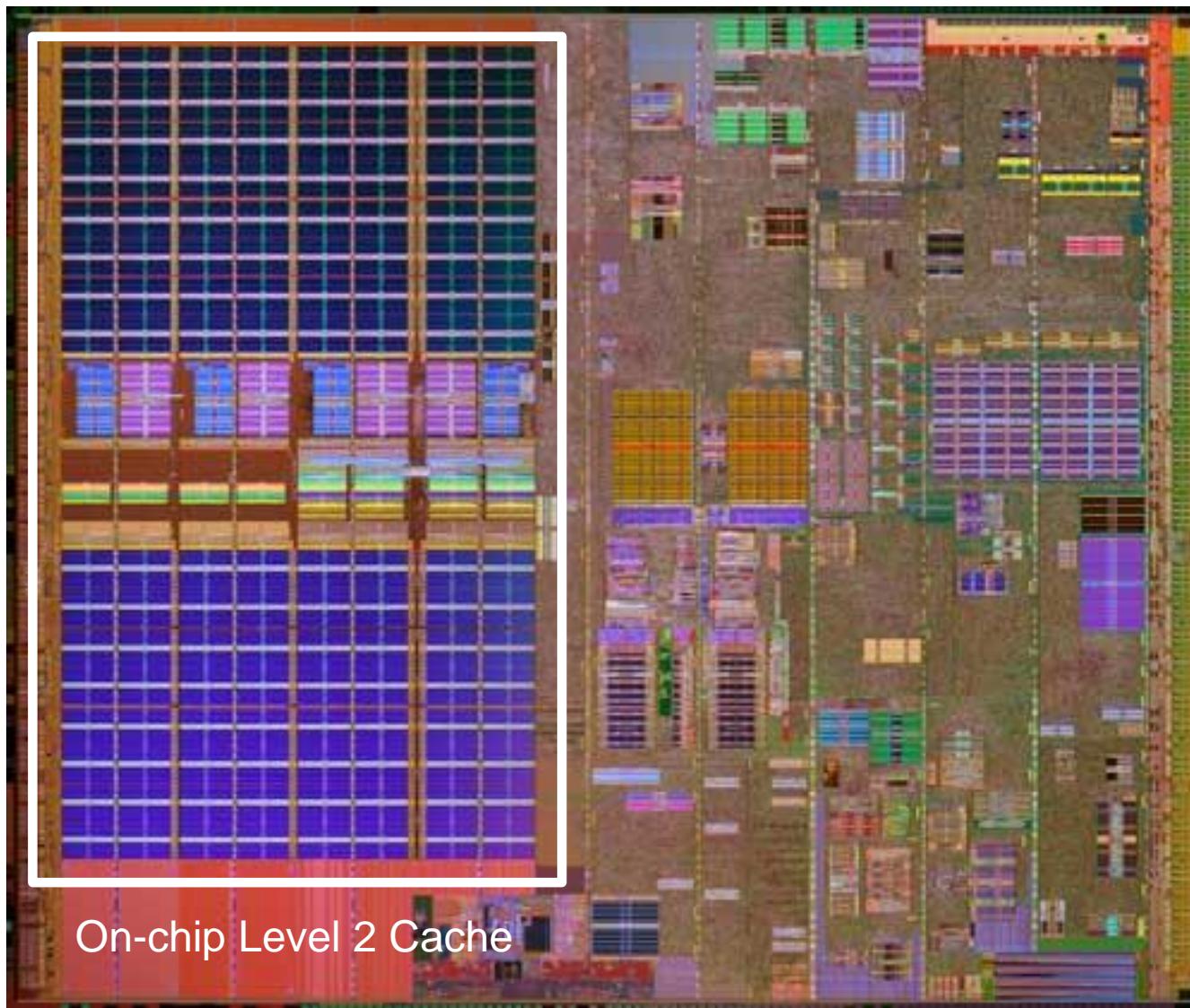


Boggs et al., “[The Microarchitecture of the Pentium 4 Processor](#),”
Intel Technology Journal, 2001.

Intel Pentium Pro (1995)



Intel Pentium 4 (2000)



Enabling OoO Execution, Revisited

1. **Link** the consumer of a value to the producer
 - ❑ **Register renaming:** Associate a “tag” with each data value
2. **Buffer** instructions until they are ready
 - ❑ Insert instruction into **reservation stations** after renaming
3. Keep **track** of **readiness** of source values of an instruction
 - ❑ **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, **dispatch** the instruction to functional unit (FU)
 - ❑ **Wakeup and select/schedule** the instruction

Summary of OOO Execution Concepts

- Register renaming eliminates false dependences, enables linking of producer to consumers
- Buffering in reservation stations enables the pipeline to move for independent instructions
- Tag broadcast enables communication (of readiness of produced value) between instructions
- Wakeup and select enables out-of-order dispatch

OOO Execution: Restricted Dataflow

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
 - which piece?
 - The dataflow graph is limited to the **instruction window**
 - Instruction window: all decoded but not yet retired instructions
 - Can we do it for the whole program?
 - Why would we like to?
 - In other words, how can we have a large instruction window?
 - Can we do it efficiently with Tomasulo's algorithm?
-

State of RAT and RS in Cycle 7

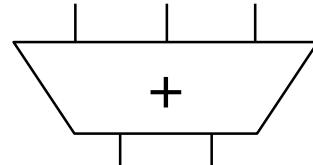
Slightly harder tasks for you:

1. Draw the dataflow graph for the executing code
2. Provide the executing code in sequential order

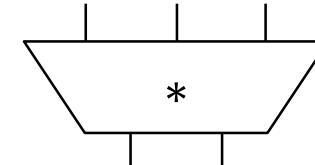
Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

Register Alias Table

Source 1	Source 2		
	V	Tag	Value
	V	Tag	Value
a	0	x	
b	1	~	2
c	1	~	8
d	0	a	



Source 1	Source 2		
	V	Tag	Value
	V	Tag	Value
x	1	~	1
y	0	b	
z			
t			



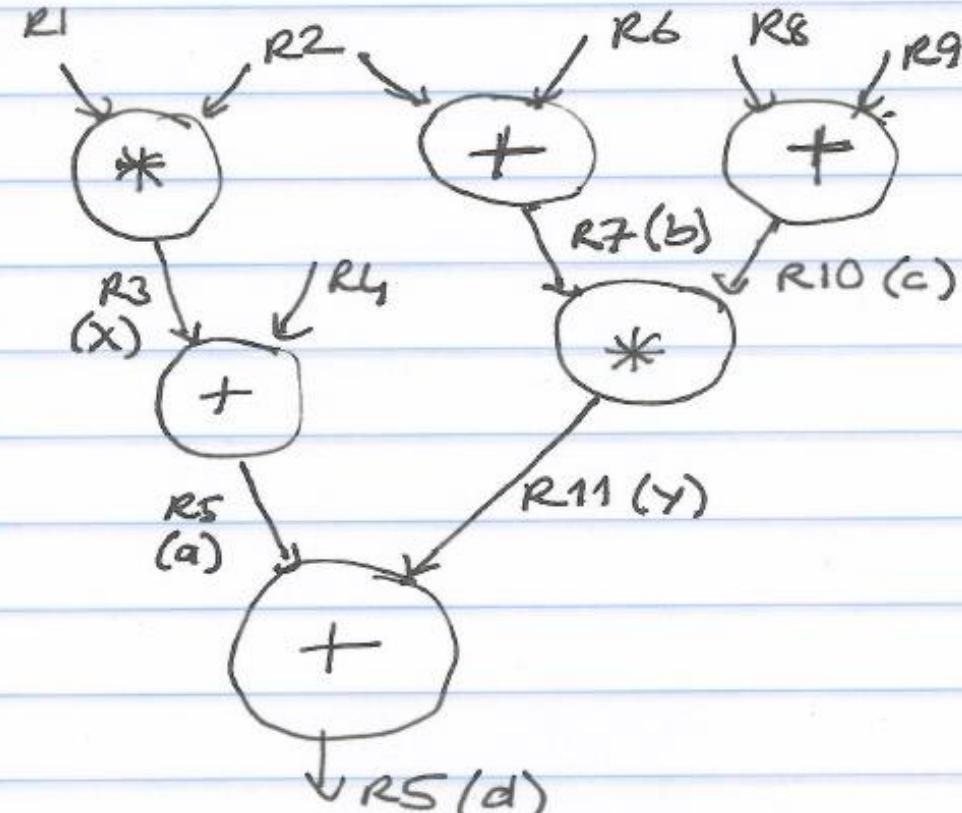
Recall: Reverse Engineered Dataflow Graph

MUL R1, R2 → R3 (x)
ADD R3, R4 → R5 (a)
ADD R2, R6 → R7 (b)
ADD R8, R9 → R10 (c)
MUL R7, R10 → R11 (y)
ADD R5, R11 → R5 (d)

Dataflow graph

Nodes: operations performed by the instruction

Arches: tags in Tomasulo's algorithm

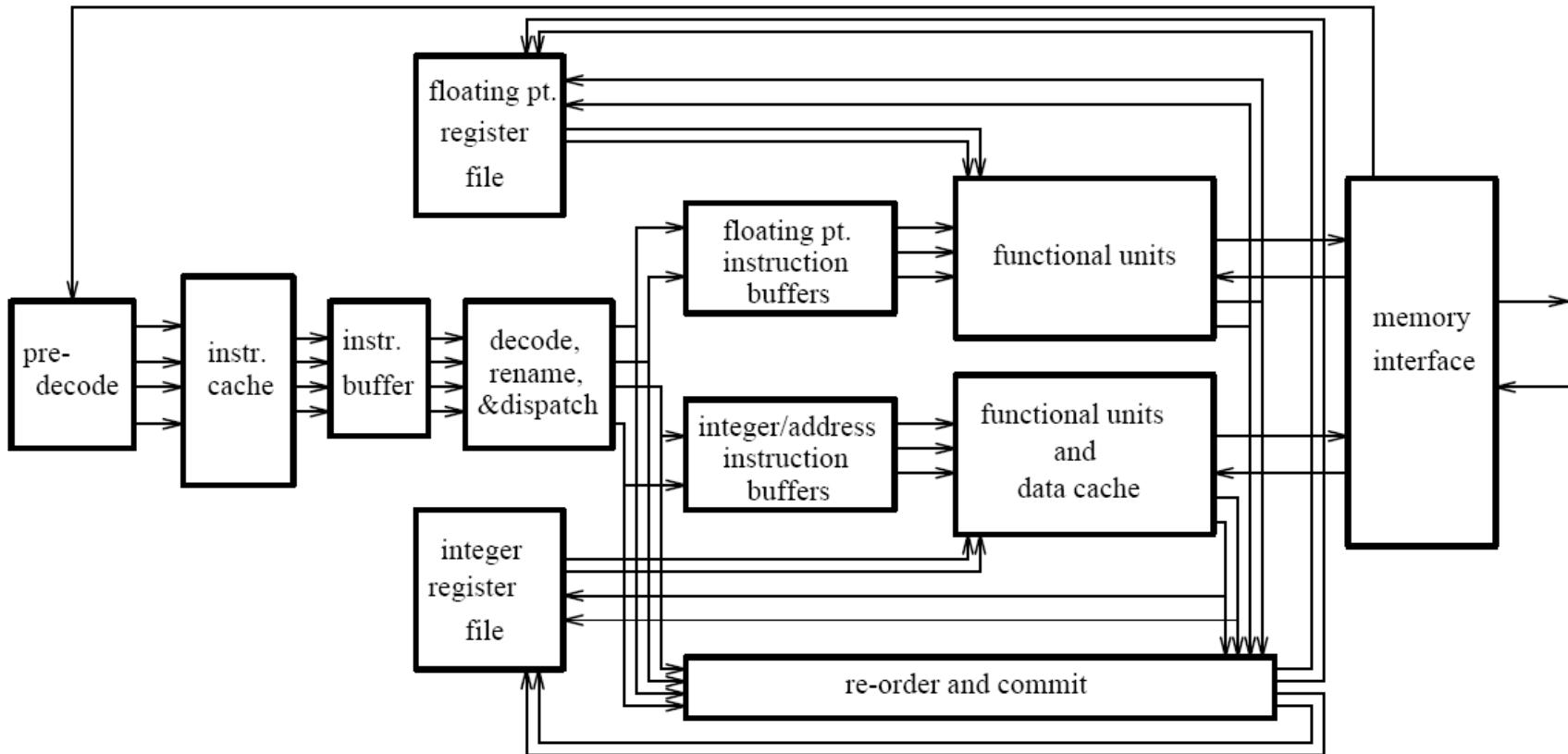


We can “easily” reverse-engineer the dataflow graph of the executing code!

Questions to Ponder

- Why is OoO execution beneficial?
 - What if all operations take a single cycle?
 - **Latency tolerance:** OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently
- What if an instruction takes 1000 cycles?
 - How large of an instruction window do we need to continue decoding?
 - How many cycles of latency can OoO tolerate?
 - **What limits the latency tolerance scalability of Tomasulo's algorithm?**
 - **Instruction window size:** how many decoded but not yet retired instructions you can keep in the machine.

General Organization of an OOO Processor



- Smith and Sohi, “**The Microarchitecture of Superscalar Processors**,” Proc. IEEE, Dec. 1995.

A Modern OoO Design: Intel Pentium 4

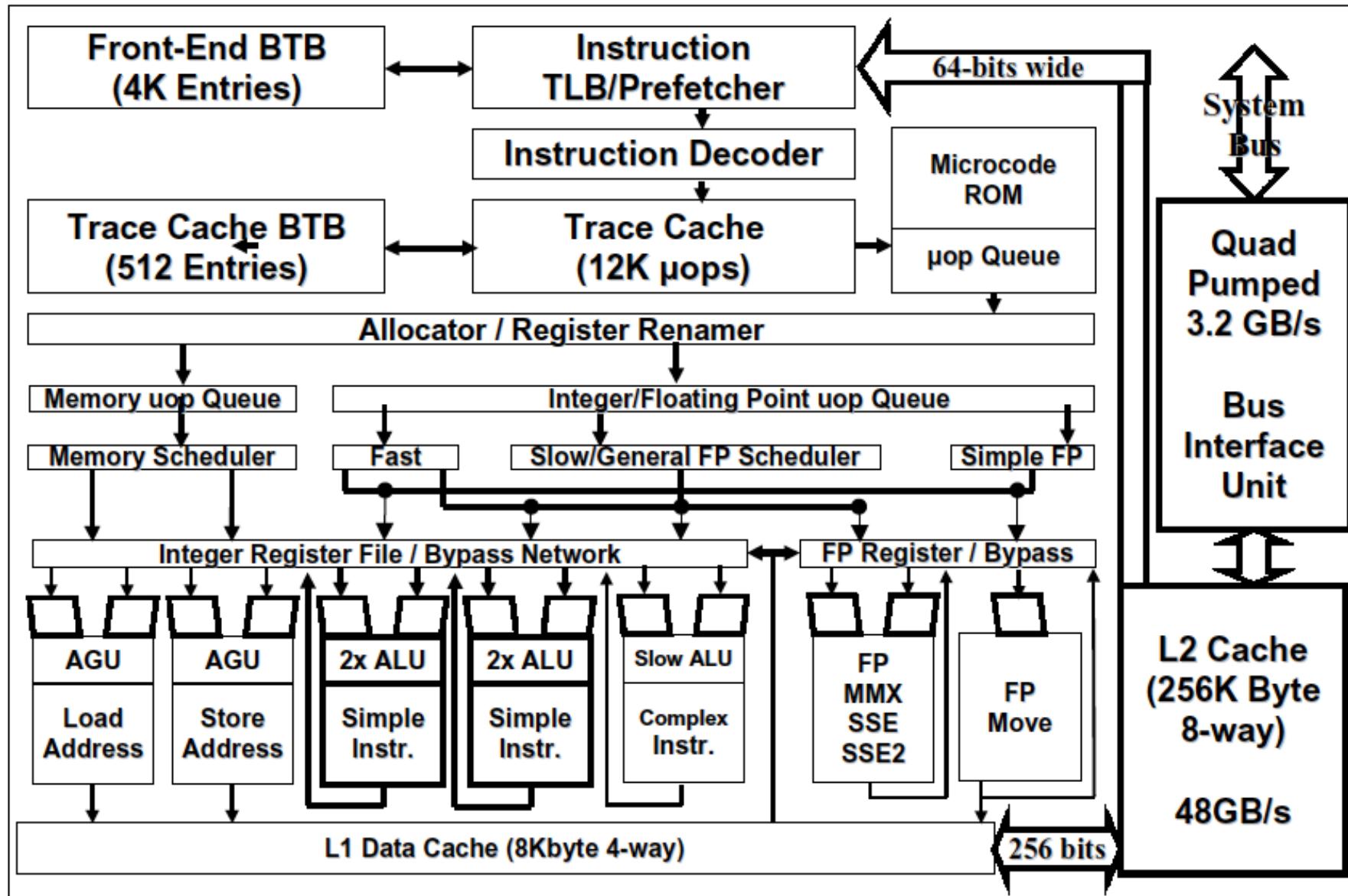
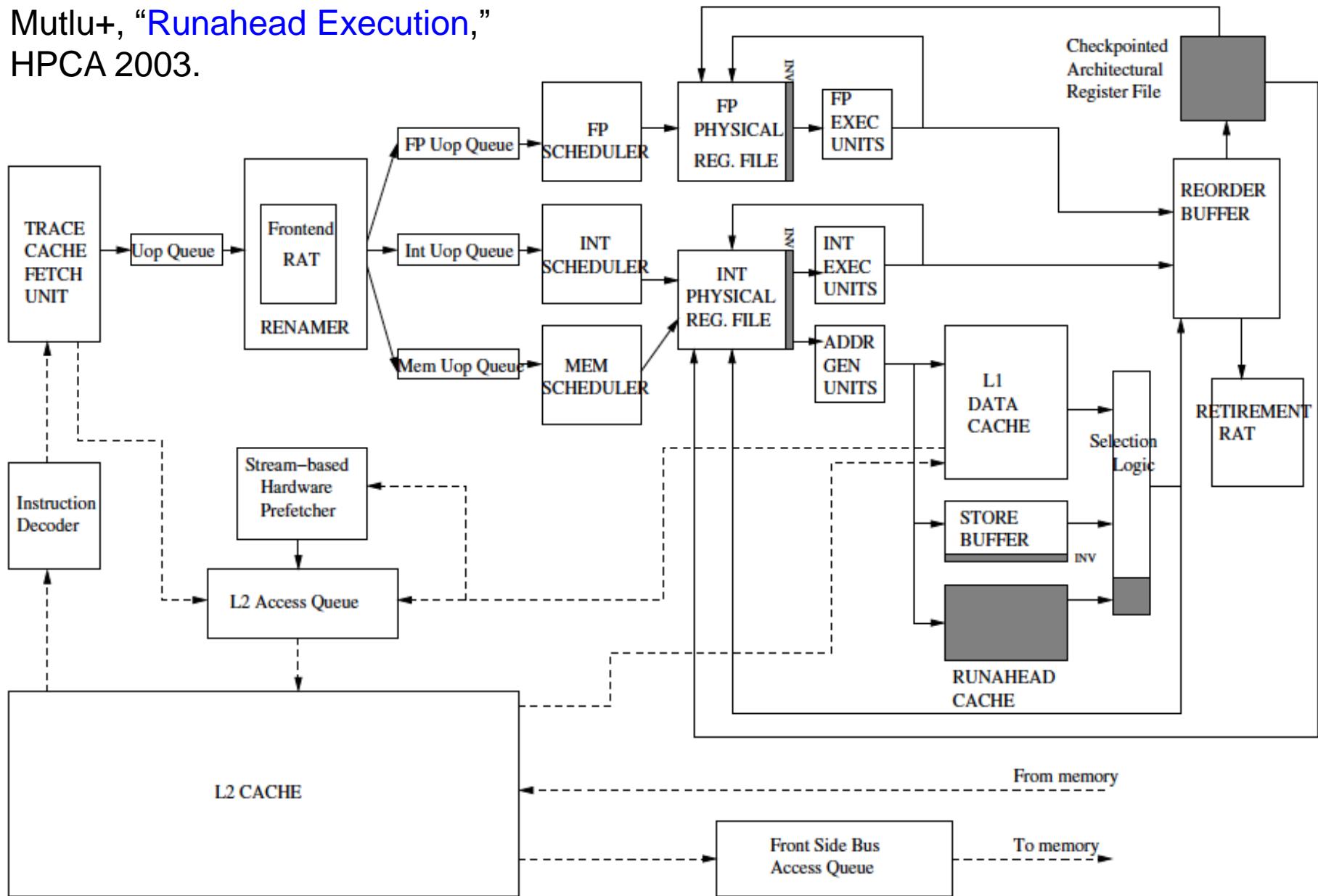


Figure 4: Pentium® 4 processor microarchitecture

Intel Pentium 4 Simplified

Mutlu+, "Runahead Execution,"
HPCA 2003.



Alpha 21264

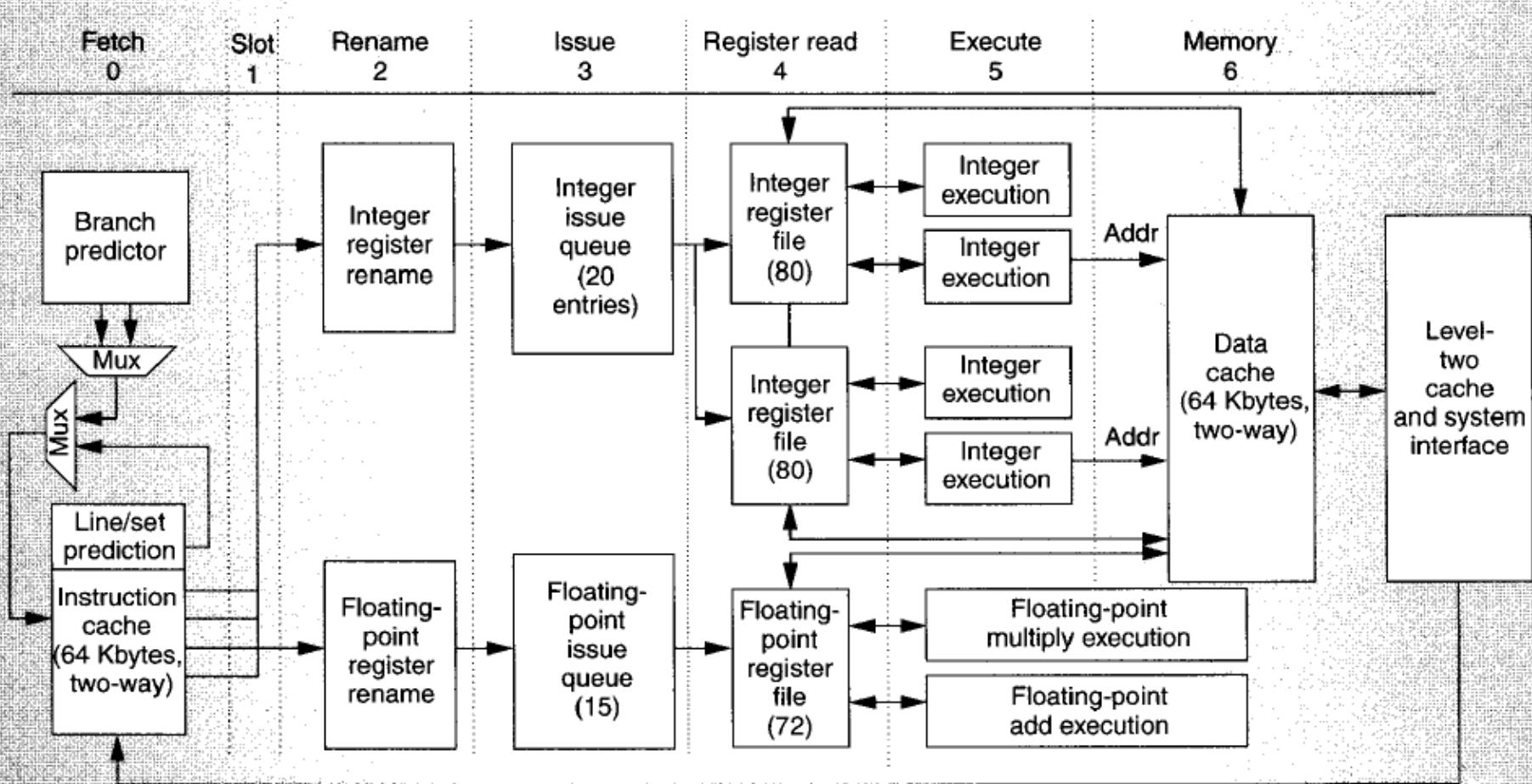
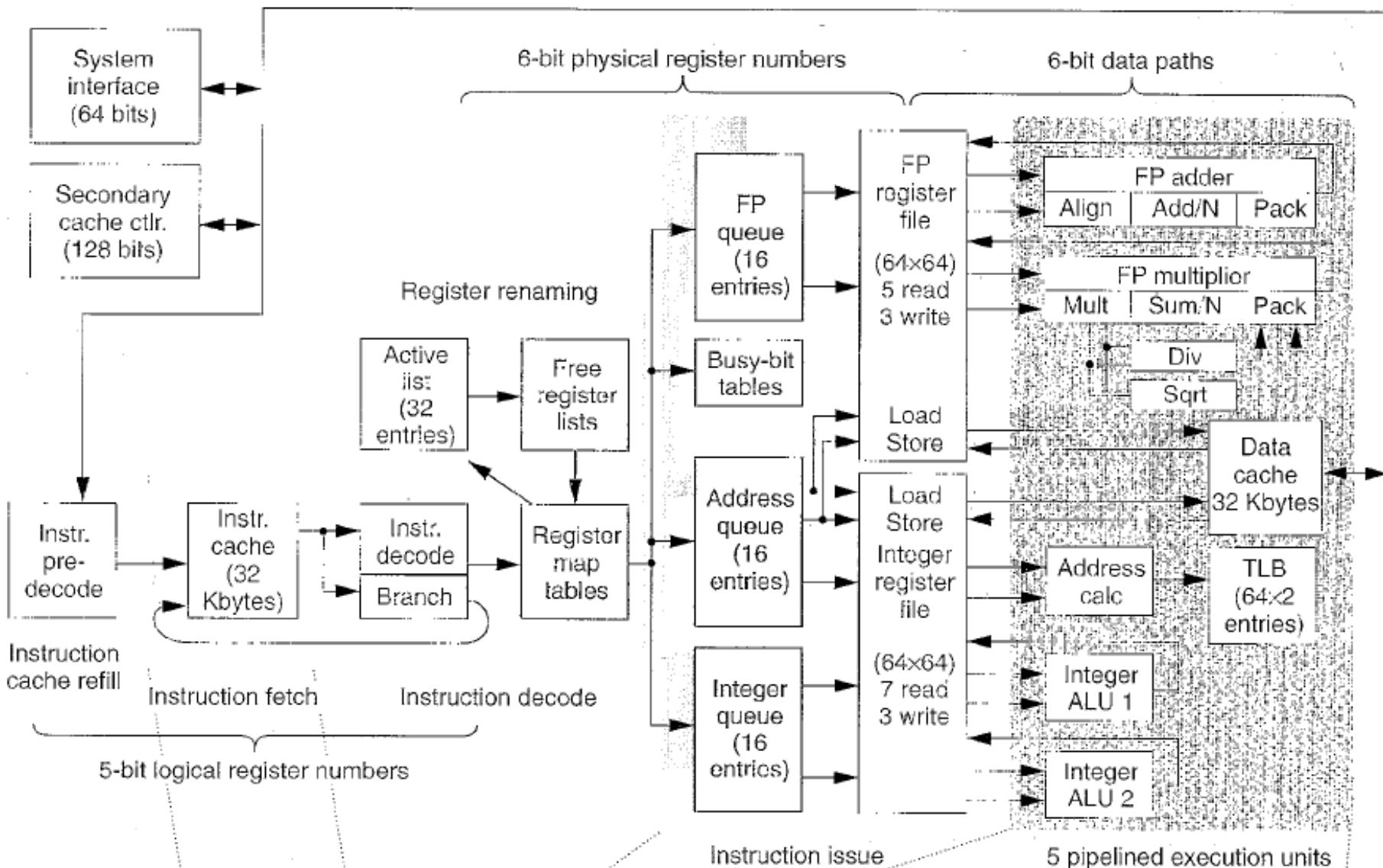


Figure 2. Stages of the Alpha 21264 instruction pipeline.

MIPS R1000

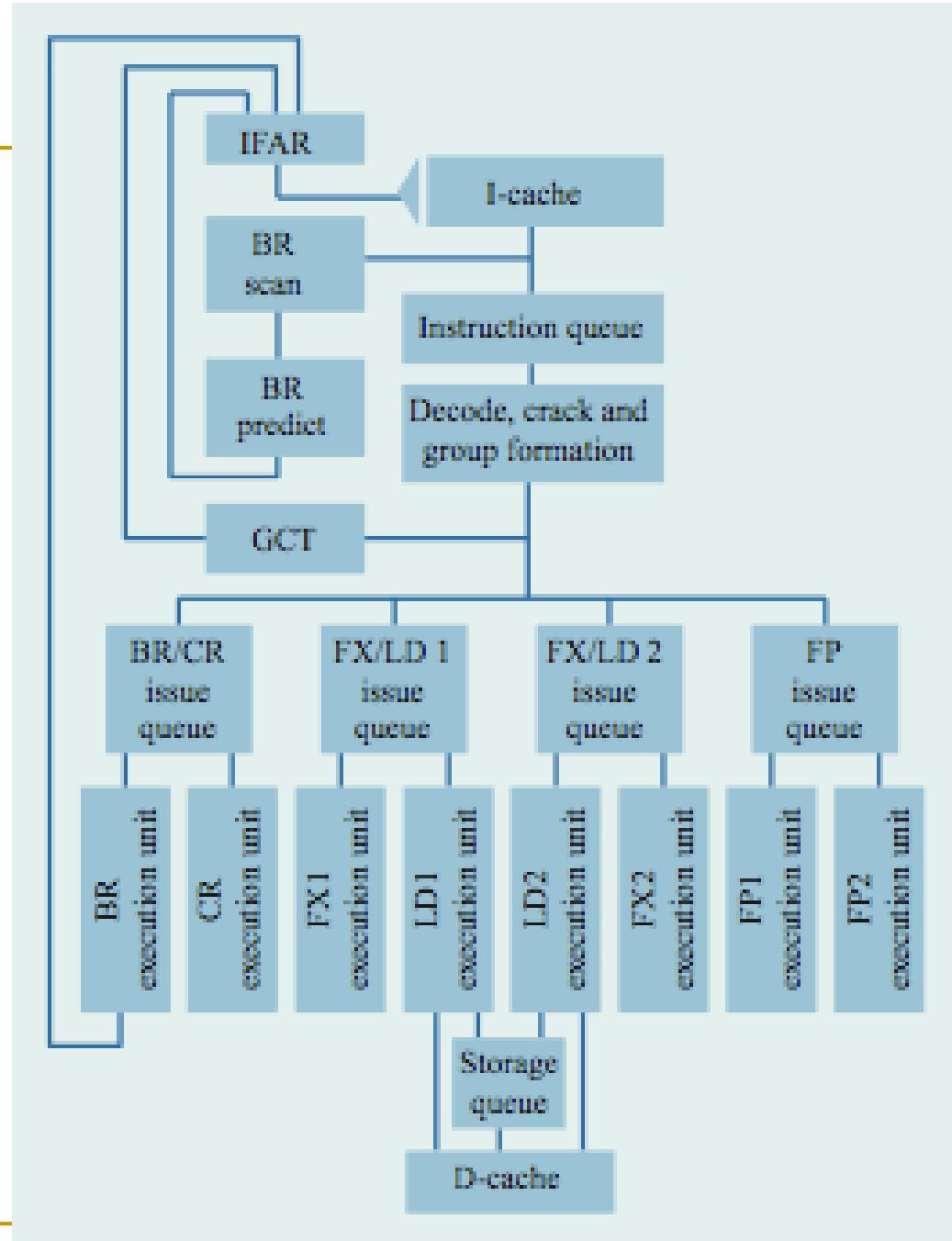
External interface



(a)

IBM POWER4

- Tendler et al.,
“POWER4 system
microarchitecture,”
IBM J R&D, 2002.



IBM POWER4

- 2 cores, out-of-order execution
- 100-entry instruction window in each core
- 8-wide instruction fetch, issue, execute
- Large, local+global hybrid branch predictor
- 1.5MB, 8-way L2 cache
- Aggressive stream based prefetching

IBM POWER5

- Kalla et al., “IBM Power5 Chip: A Dual-Core Multithreaded Processor,” IEEE Micro 2004.

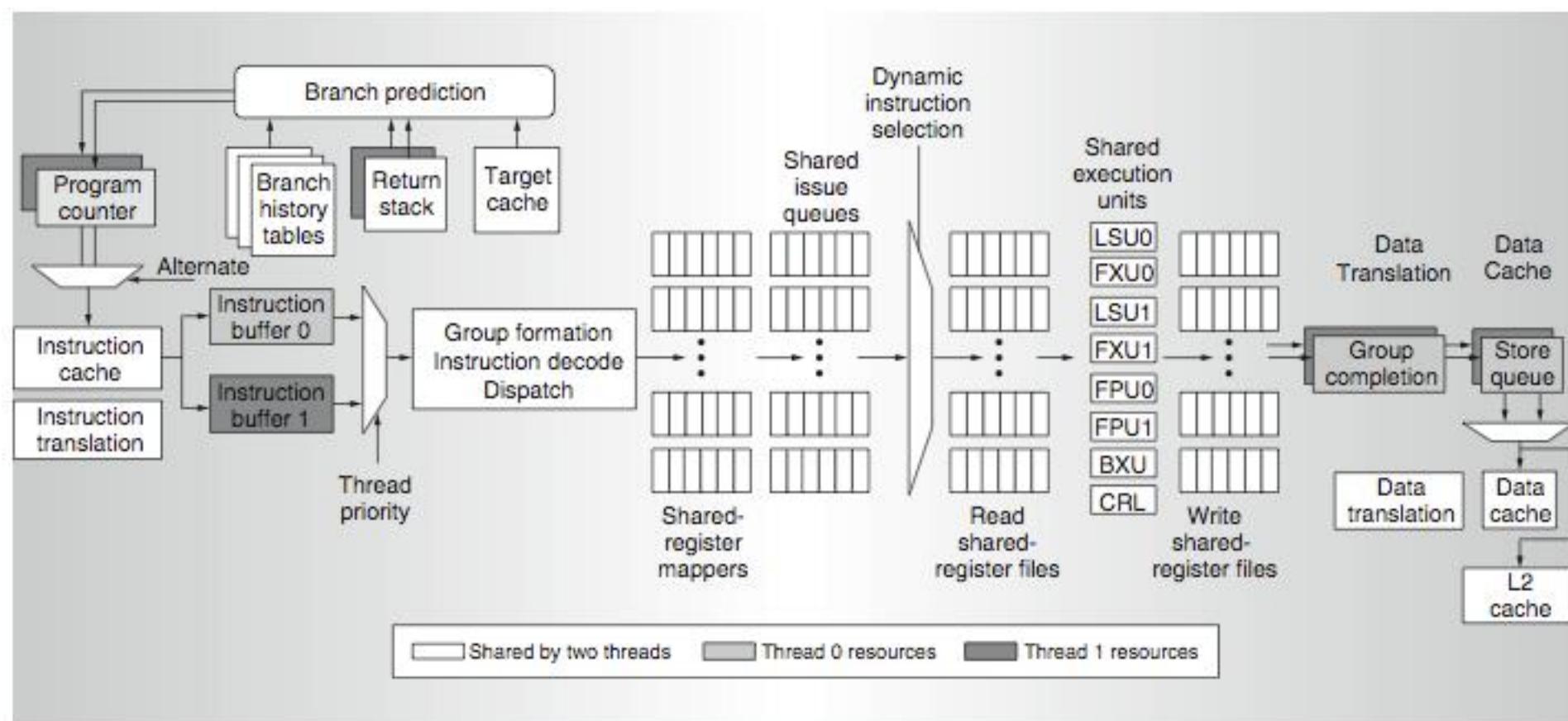
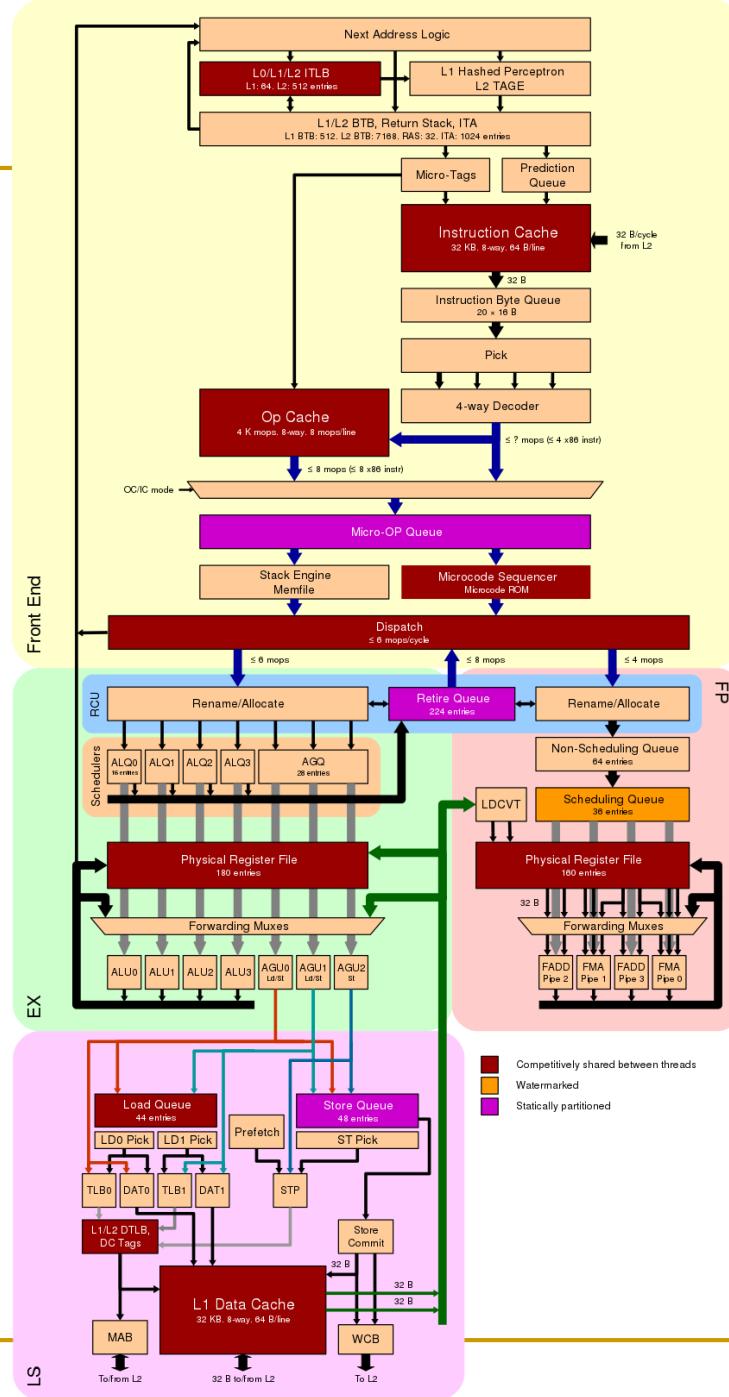
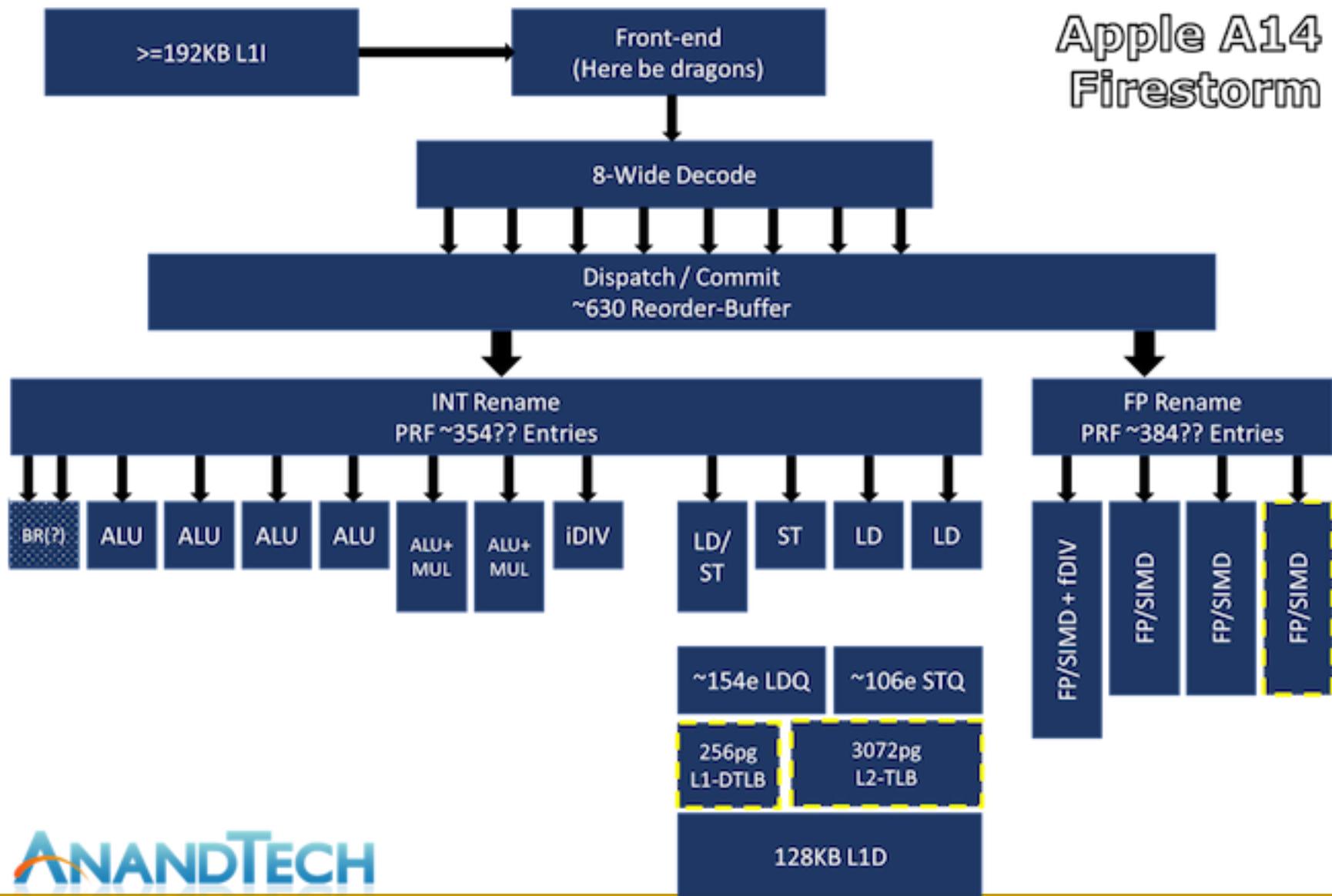


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

AMD Zen2? (2019)



Apple M1 FireStorm? (2020)



See Backup Slides for:
Handling Out-of-Order Execution
of Loads and Stores

Digital Design & Computer Arch.

Lecture 16: Out-of-Order Execution

Prof. Onur Mutlu

ETH Zürich
Spring 2022
28 April 2022

Handling Out-of-Order Execution of Loads and Stores

Registers versus Memory

- So far, we considered mainly registers as part of state
- What about memory?
- What are the fundamental differences between registers and memory?
 - Register dependences known statically – memory dependences determined dynamically
 - Register state is small – memory state is large
 - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)

Memory Dependence Handling (I)

- Need to obey memory dependences in an out-of-order machine
 - and need to do so while providing high performance
- Observation and Problem: Memory address is not known until a load/store executes
- Corollary 1: Renaming memory addresses is difficult
- Corollary 2: Determining dependence or independence of loads/stores has to be handled *after* their (partial) execution
- Corollary 3: When a load/store has its address ready, there may be older/younger stores/loads with unknown addresses in the machine

Memory Dependence Handling (II)

- When do you schedule a load instruction in an OOO engine?
 - Problem: A younger load can have its address ready before an older store's address is known
 - Known as the **memory disambiguation** problem or the **unknown address** problem
- Approaches
 - **Conservative:** Stall the load until all previous stores have computed their addresses (or even retired from the machine)
 - **Aggressive:** Assume load is independent of unknown-address stores and schedule the load right away
 - **Intelligent:** Predict (with a more sophisticated predictor) if the load is dependent on any unknown address store

Handling of Store-Load Dependencies

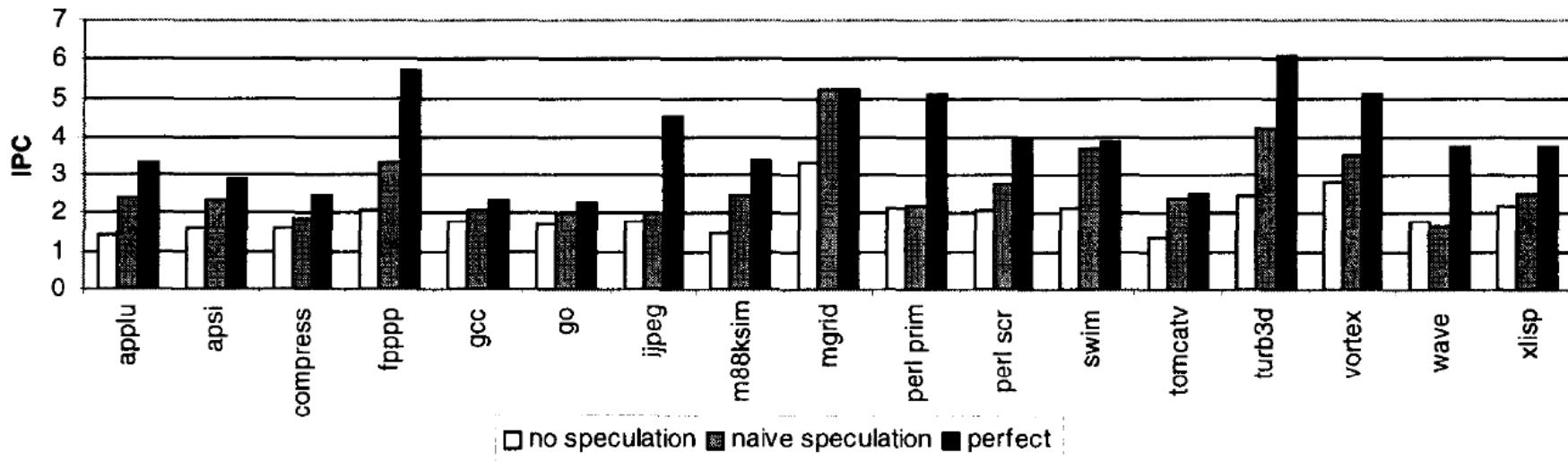
- **A load's dependence status is not known** until all previous store addresses are available.
- How does the OOO engine detect dependence of a load instruction on a previous store?
 - Option 1: Wait until all previous stores committed (no need to check for address match)
 - Option 2: Keep a list of pending stores in a store buffer and check whether load address matches a previous store address
- How does the OOO engine treat the scheduling of a load instruction wrt previous stores?
 - Option 1: Assume load dependent on all previous stores
 - Option 2: Assume load independent of all previous stores
 - Option 3: Predict the dependence of a load on an outstanding store

Memory Disambiguation (I)

- Option 1: Assume load is dependent on all previous stores
 - + No need for recovery
 - Too conservative: delays independent loads unnecessarily
- Option 2: Assume load is independent of all previous stores
 - + Simple and can be common case: no delay for independent loads
 - Requires recovery and re-execution of load and dependents on misprediction
- Option 3: Predict the dependence of a load on an outstanding store
 - + More accurate. Load store dependences persist over time
 - Still requires recovery/re-execution on misprediction
 - ❑ Alpha 21264 : Initially assume load independent, delay loads found to be dependent
 - ❑ Moshovos et al., “**Dynamic speculation and synchronization of data dependences**,” ISCA 1997.
 - ❑ Chrysos and Emer, “**Memory Dependence Prediction Using Store Sets**,” ISCA 1998.

Memory Disambiguation (II)

- Chrysos and Emer, “Memory Dependence Prediction Using Store Sets,” ISCA 1998.



- Predicting store-load dependences important for performance
- Simple predictors (based on past history) can achieve most of the potential performance

Data Forwarding Between Stores and Loads

- We cannot update memory out of program order
 - Need to buffer all store and load instructions in instruction window
- Even if we **know** all addresses of past stores when we generate the address of a load, two questions still remain:
 1. How do we check whether or not it is dependent on a store
 2. How do we forward data to the load if it is dependent on a store
- Modern processors use a LQ (load queue) and a SQ for this
 - Can be combined or separate between loads and stores
 - A load searches the SQ after it computes its address. Why?
 - A store searches the LQ after it computes its address. Why?

Out-of-Order Completion of Memory Ops

- When a store instruction finishes execution, it writes its address and data in its reorder buffer entry (or SQ entry)
- When a later load instruction generates its address, it:
 - searches the SQ with its address
 - accesses memory with its address
 - receives the value from the youngest older instruction that wrote to that address (either from ROB or memory)
- This is a complicated “search logic” implemented as a Content Addressable Memory
 - Content is “memory address” (but also need *size* and *age*)
 - Called **store-to-load forwarding logic**

Store-Load Forwarding Complexity

- Content Addressable Search (based on Load Address)
- Range Search (based on Address and Size of both the Load and earlier Stores)
- Age-Based Search (for last written values)
- Load data can come from a combination of multiple places
 - One or more stores in the Store Buffer (SQ)
 - Memory/cache

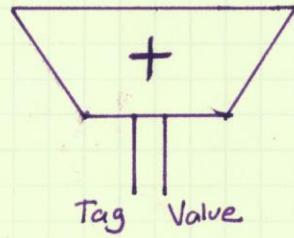
CYCLE —

MUL R1, R2 → R3
ADD R3, R4 → R5
ADD R2, R6 → R7
ADD R8, R9 → R10
MUL R7, R10 → R11
ADD R5, R11 → R5

Register Alias Table

	V	Tag	Value
R1			
R2			
R3			
R4			
R5			
R6			
R7			
R8			
R9			
R10			
R11			

	SOURCE 1			SOURCE 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



	SOURCE 1			SOURCE 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						

