Precise Exception: Keeping sequential execution with pipeline

Seehwan Yoo

Dankook University

Dept. of Mobile Systems Engineering

DKU DANKOOK UNIVERSITY

Review: Pipelined u-Arch

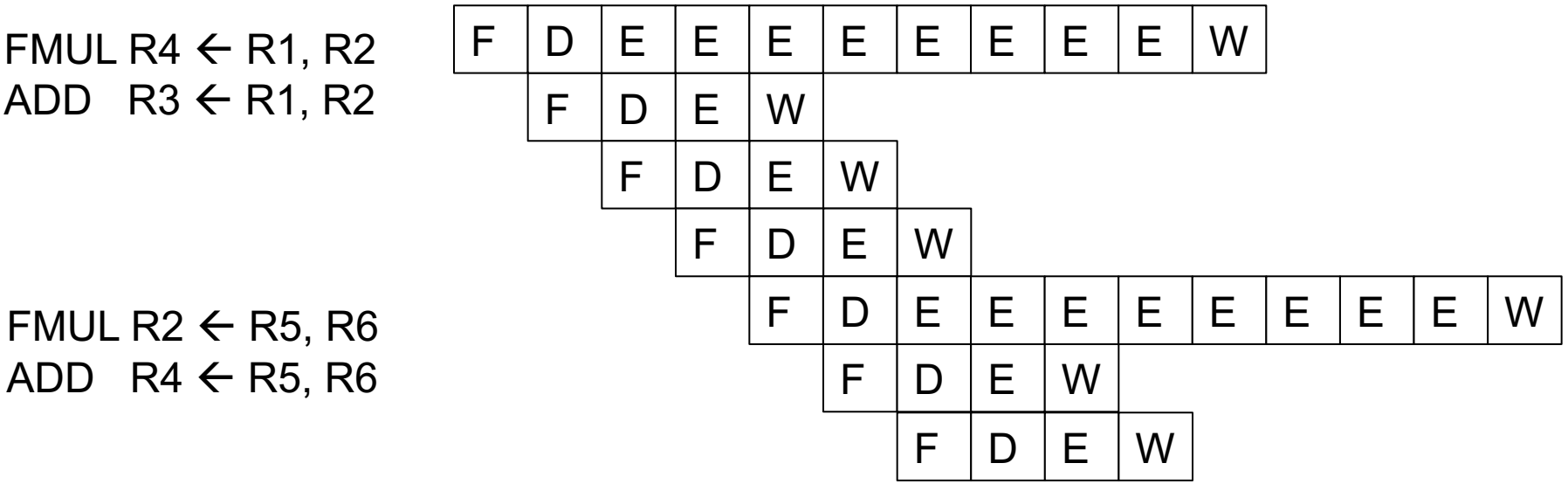
- Divide instruction processing into stages
  - Fetch, Decode, Execute, Memory, Writeback
  - Could be much more
- Each stage utilizes different hardware (logic circuit)
  - IMem, Decoder/RegFile, ALU/adder, DMem, RegFile
- Why not utilize them all at the same time?
  - Overlap the execution of different stages
    - Fill in different instructions into pipeline stages
  - Instruction-Level Parallelism
    - Parallel execution of multiple instructions

Pipeline in a real world

- Take care dependencies
- Data dependency
  - Some instructions have dependency
- Control dependency
  - Some instruction changes control flow
- Handling dependency
  - Stall (hardware-based interlocking)
  - Software-based interlocking (add NOP instruction)
  - Forwarding, Scoreboarding
  - Delayed branch
  - Branch prediction, Predicated execution
  - Fine-grained multi-threading

Consider Multicycle Functional units!

- Integer add is much easier than floating-point add
  - And multiply, and memory operations
- Integer execution 1 cycle
- FPMultiply execution 8 cycles
- Say, Memory 10 cycles
- Consume cycles until the result is produced



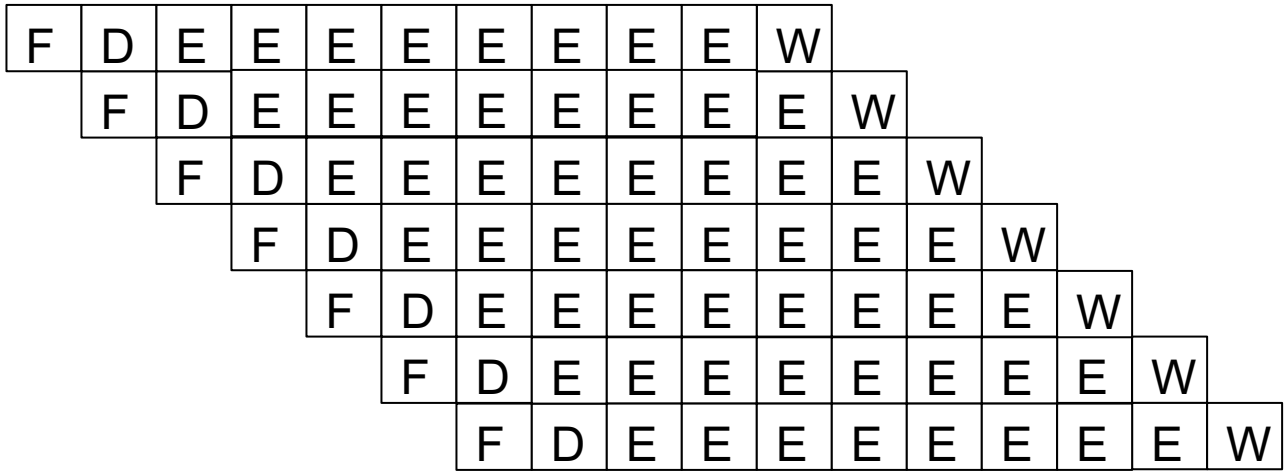
Sequential execution is broken!

- Completes instructions out-of-order!
  - Note that fetch instructions in-order
- How would you debug?
- What if exception/interrupt occurs?
- Need to preserve ordered completion
- Commit == graduate == retire

A non-fix

- Simple dumb idea

FMUL R3 ← R1, R2
ADD R4 ← R1, R2



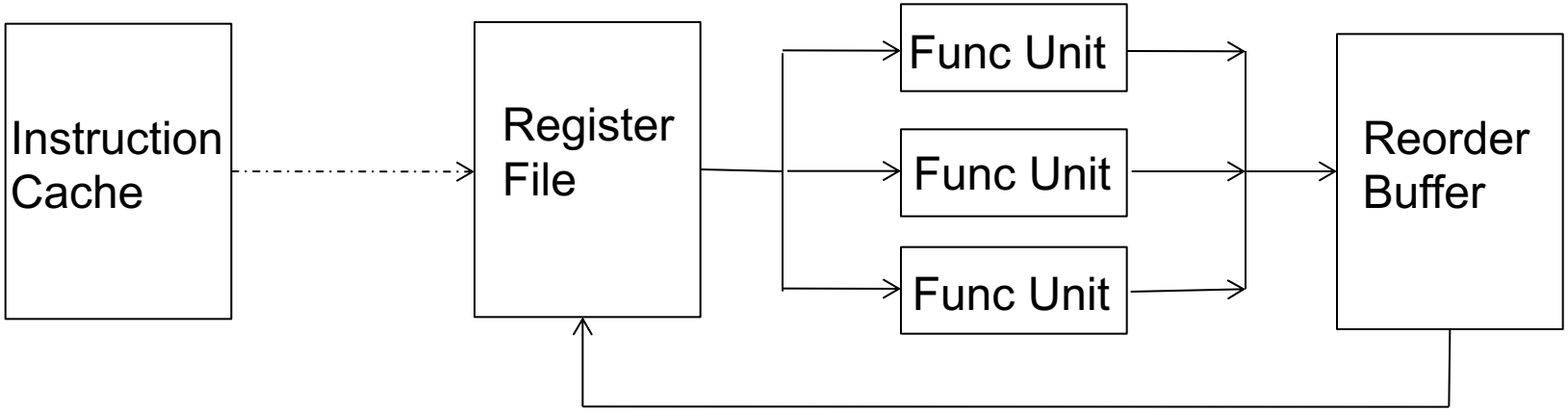
- What about memory operation?
  - Each functional unit takes 500 cycles?

Some solutions

- Reorder buffer
- History buffer
- Future register file
- Checkpointing
- Recommended Reading
  - Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans on Computers 1988 and ISCA 1985.
  - Hwu and Patt, "Checkpoint Repair for Out-of-order Execution Machines," ISCA 1987.

ROB (Reorder-buffer)

- Idea: Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is decoded it reserves an entry in the ROB
- When instruction completes, it writes result into ROB entry
- When instruction oldest in ROB and it has completed without exceptions, its result moved to reg. file or memory



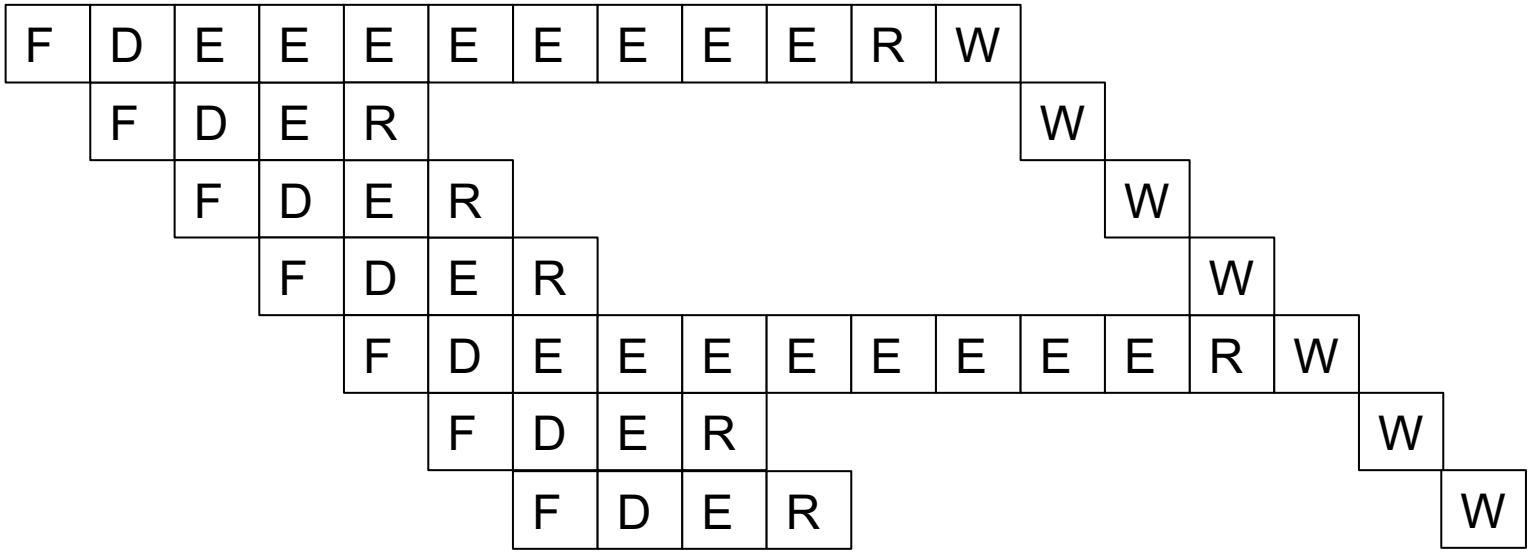
ROB entry

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC	Valid bits for reg/data + control bits	Exc?

- Need valid bits to keep track of readiness of the result(s)

ROB: independent ops

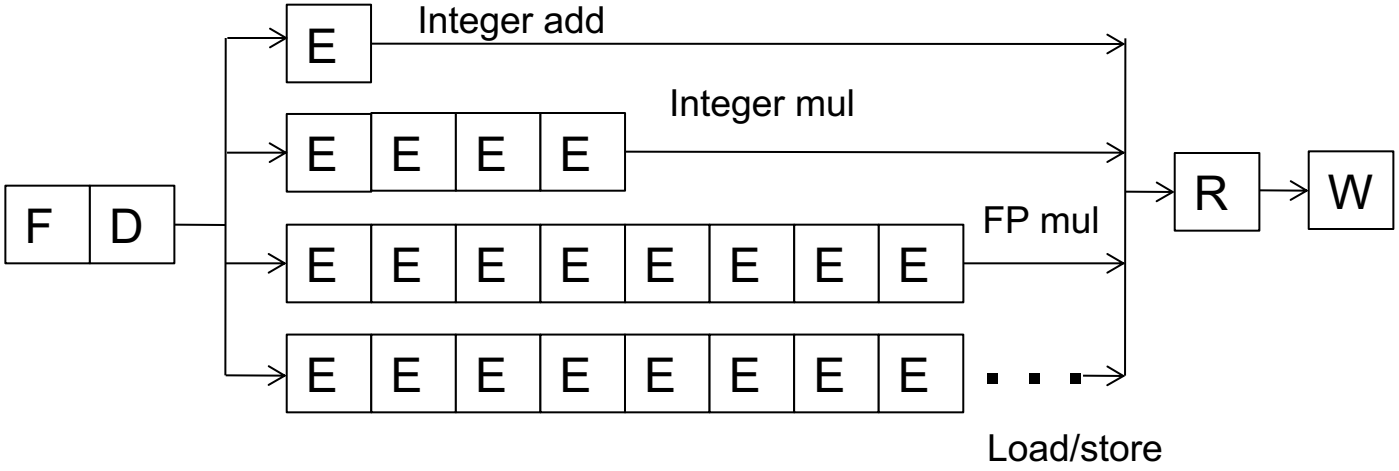
- Results first written to ROB, then to register file at commit time



- What if a later operation needs a value in the reorder buffer?
  - Read reorder buffer in parallel with the register file. How?

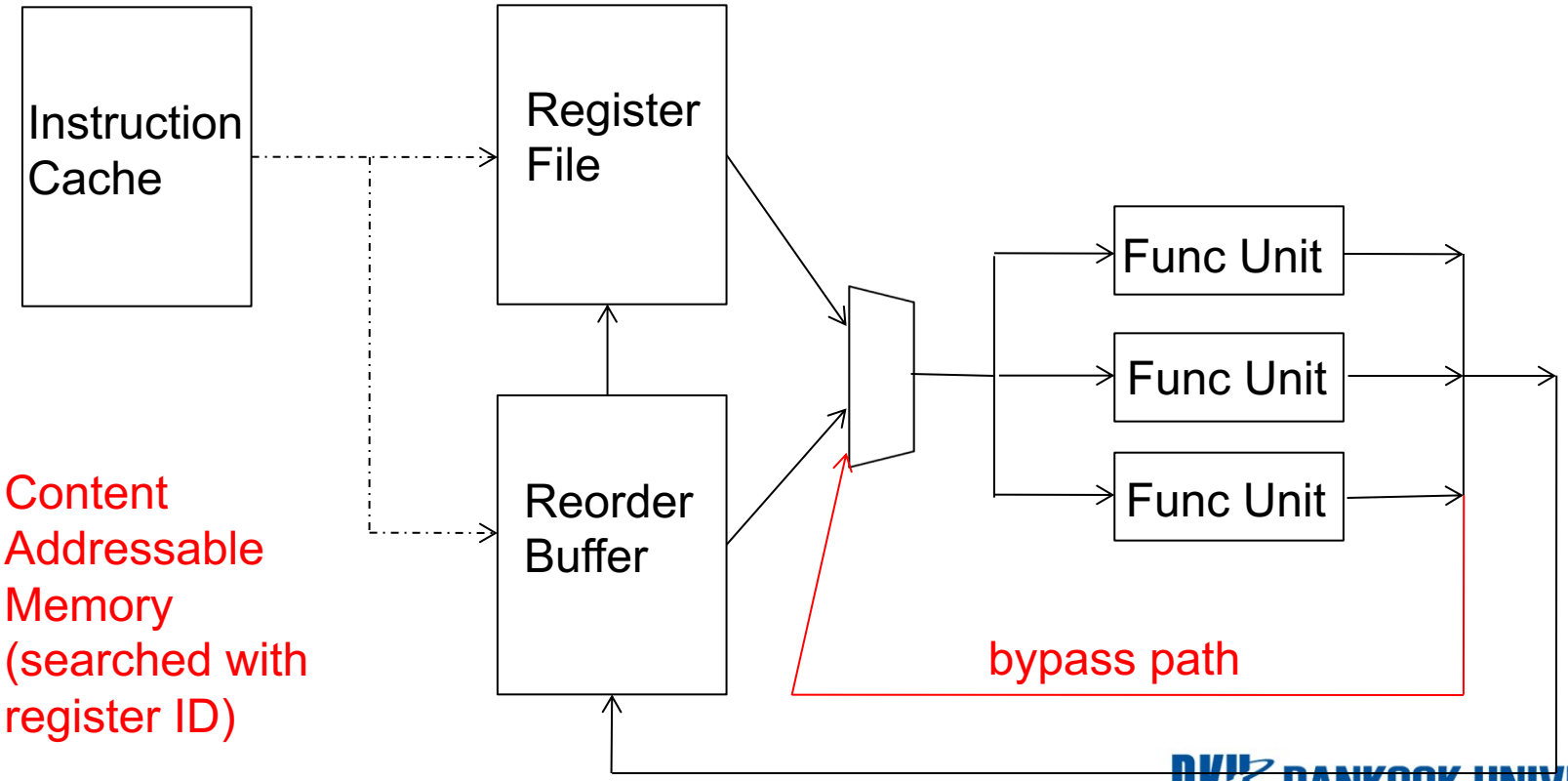
In-order pipeline with ROB

- Decode (D): Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- 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



What if ∃data dependency?

- Now, register file has no recent data!
- Reorder buffer has updated value!
- Or you can get value from forwarding path (bypass network)



How to find value in ROB?

- Idea: Use indirection
- Access register file first
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
  - Mapping of the register to a ROB entry
- Access reorder buffer next

- What is in a reorder buffer entry?

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC/IP	Control/val id bits	Exc?

  - Can it be simplified further?

Register renaming

- Output and anti dependencies are not true dependencies
  - 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- dependencies
  - Gives the illusion that there are a large number of registers

ROB: +/-

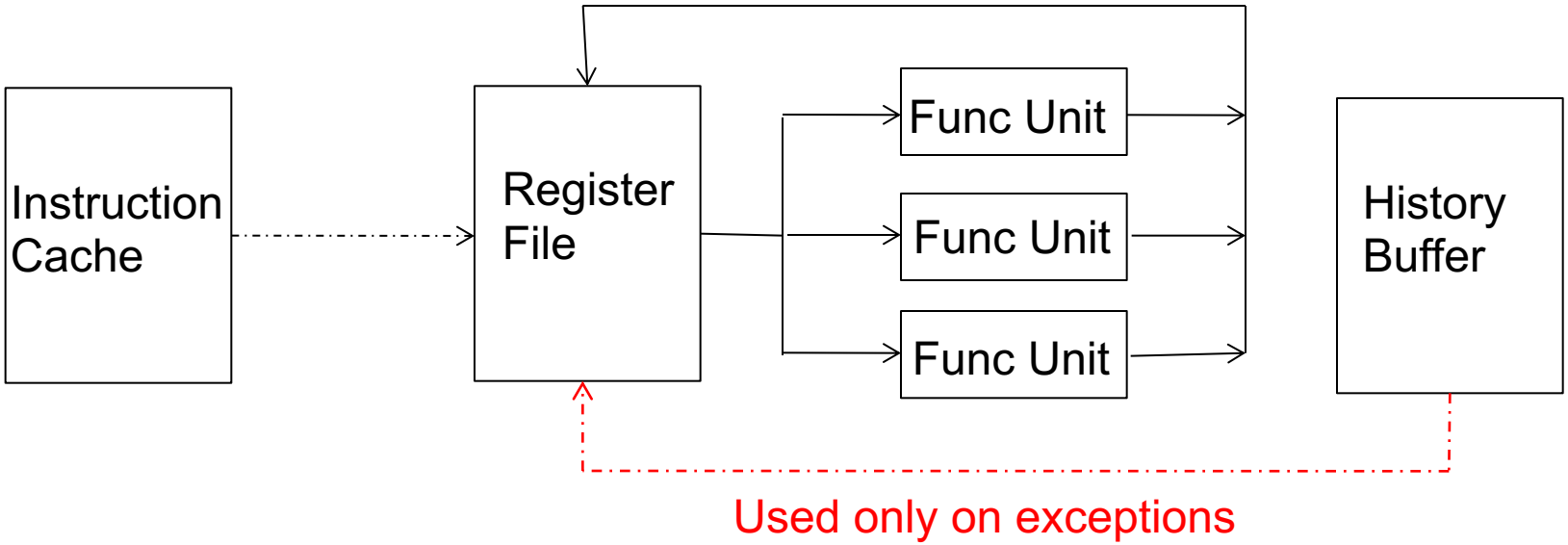
- Advantages
  - Conceptually simple for supporting precise exceptions
  - Can eliminate false dependencies
- Disadvantages
  - Reorder buffer needs to be accessed to get the results that are yet to be written to the register file
    - CAM or indirection → increased latency and complexity
- Other solutions aim to eliminate the cons
  - History buffer
  - Future file
  - Checkpointing

History Buffer

- Idea: Update the register file when instruction completes, but UNDO UPDATES when an exception occurs

- When instruction is decoded, it reserves an HB entry
- When the instruction completes, it stores the old value of its destination in the HB
- When instruction is oldest and no exceptions/interrupts, the HB entry discarded
- When instruction is oldest and an exception needs to be handled, old values in the HB are written back into the architectural state from tail to head

History Buffer



- Advantage:
  - Register file contains up-to-date values. History buffer access not on critical path
- Disadvantage:
  - Need to read the old value of the destination register
  - Need to unwind the history buffer upon an exception → increased exception/interrupt handling latency

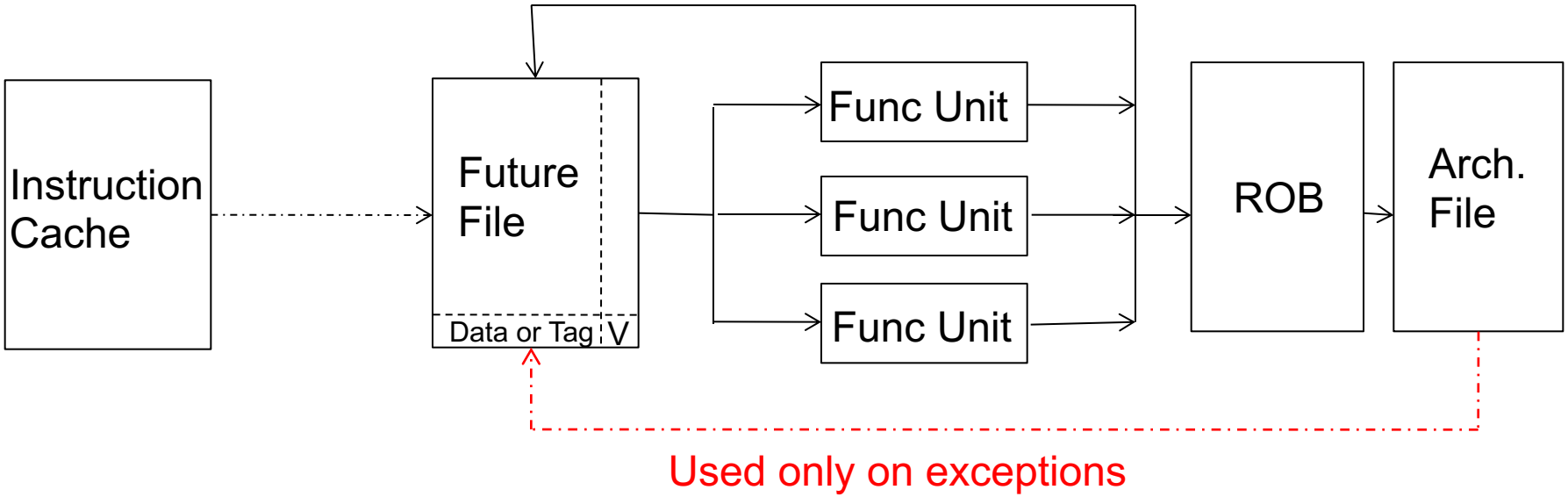
Future File + ROB

- Idea: Keep two register files (speculative and architectural)
  - Arch reg file: Updated in program order for precise exceptions
    - Use a reorder buffer to ensure in-order updated
  - Future reg file: Updated as soon as an instruction completes (if the instruction is the youngest one to write to a register)

- Future file is used for fast access to latest register values (speculative state)
  - Frontend register file

- Architectural file is used for state recovery on exceptions (architectural state)
  - Backend register file

Future File



- Advantage
  - No need to read the values from the ROB (no CAM or indirection)
- Disadvantage
  - Multiple register files
  - Need to copy arch. reg. file to future file on an exception

Side Note: Branch misprediction

- Similar to Exception
  - Except it is not visible to SW
- Much more common than exception, so state recovery has to be fast

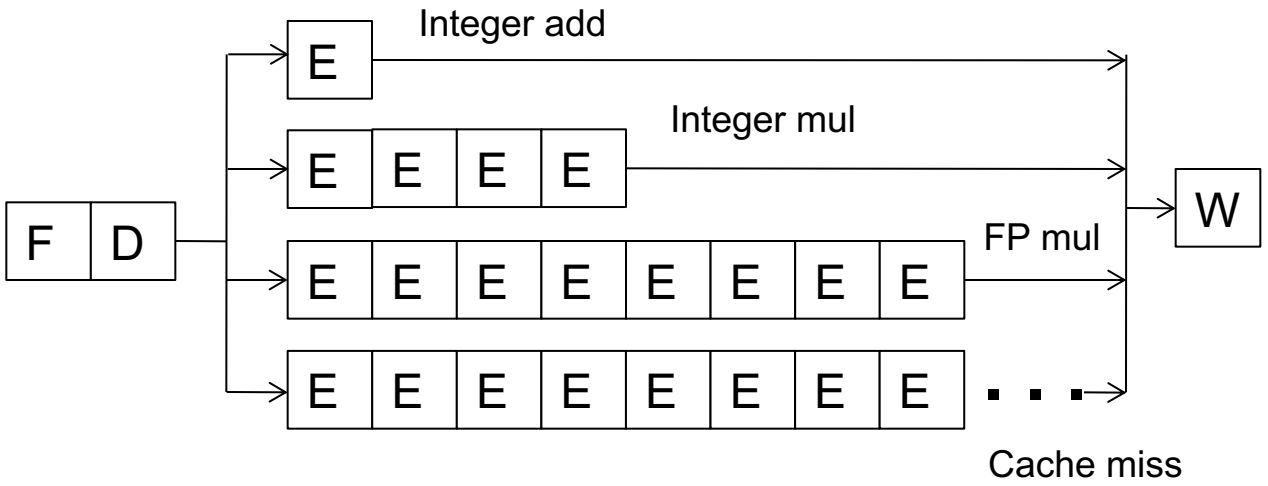
- Reorder Buffer
  - Wait until branch is the oldest instruction in the machine
  - Flush pipeline afterwards
- History buffer
  - Undo all instructions after the branch by rewinding from the tail of the history buffer until the branch & restoring old values into the register file
- Future file
  - Wait until branch is the oldest instruction in the machine
  - Copy arch. reg. file to future file and flush pipeline

Side Note2: Store Buffer

- Handling out-of-order completion of memory operations
  - UNDOing a memory write more difficult than UNDOing a register write. Why?
  - One idea: Keep store address/data in reorder buffer
    - How does a load instruction find its data?
  - Store/write buffer: Similar to reorder buffer, but used only for store instructions
    - Program-order list of un-committed store operations
    - When store is decoded: Allocate a store buffer entry
    - When store address and data become available: Record in store buffer entry
    - When the store is the oldest instruction in the pipeline: Update the memory address (i.e. cache) with store data

Out-of-order Execution Making irrationally intelligent HW

An In-order Pipeline



- Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units
- Dispatch: Act of sending an instruction to a functional unit

A motivational case for OoO

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

```
IMUL  R3 ← R1, R2          LD    R3 ← R1 (0)
ADD   R3 ← R3, R1          ADD   R3 ← R3, R1
ADD   R1 ← R6, R7          ADD   R1 ← R6, R7
IMUL  R5 ← R6, R8          IMUL  R5 ← R6, R8
ADD   R7 ← R3, R5          ADD   R7 ← R3, R5
```

- Answer: First ADD stalls 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

Out-of-order Execution (Dynamic Scheduling)

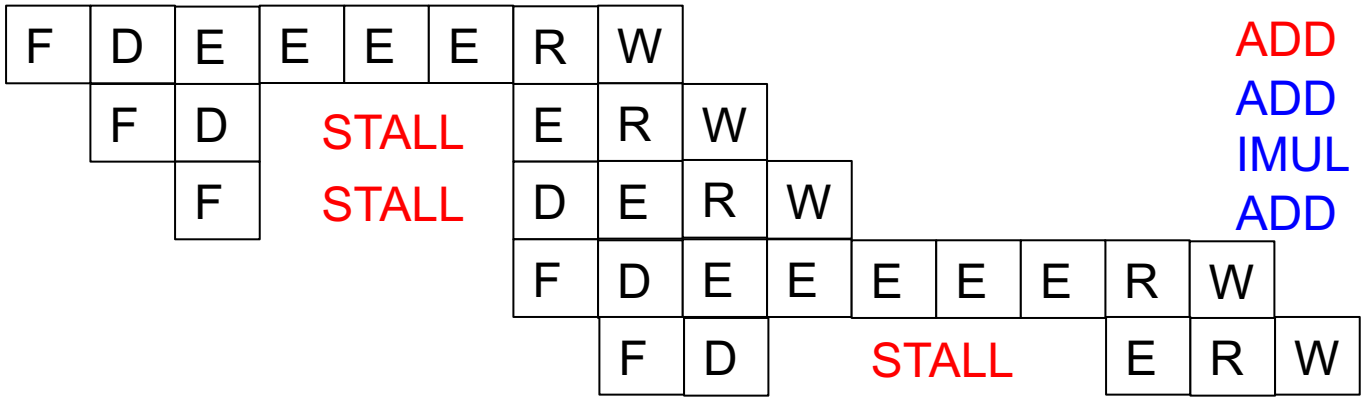
- Idea: Move the dependent instructions out of the way of independent ones
  - Rest areas for dependent instructions: Reservation stations

- Monitor the source "values" of each instruction in the re sting 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

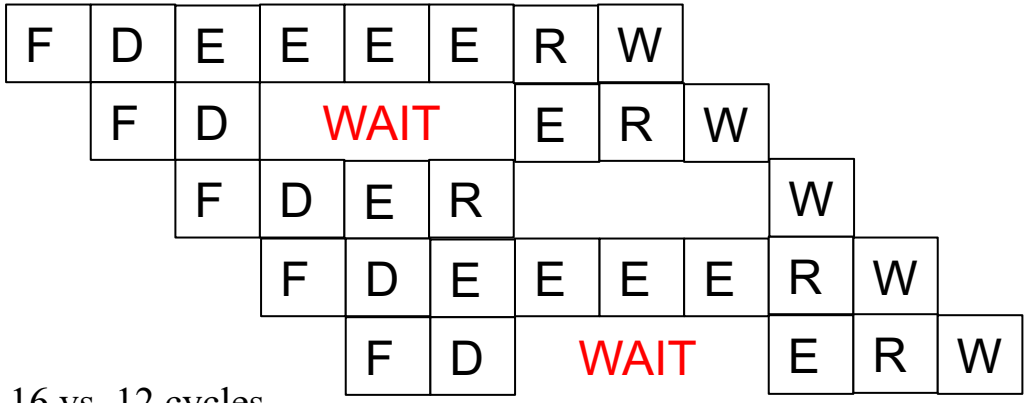
In-order vs. Out-of-order Dispatch

- In order dispatch:



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

- Tomasulo + 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)
   - What if more instructions become ready than available FUs?

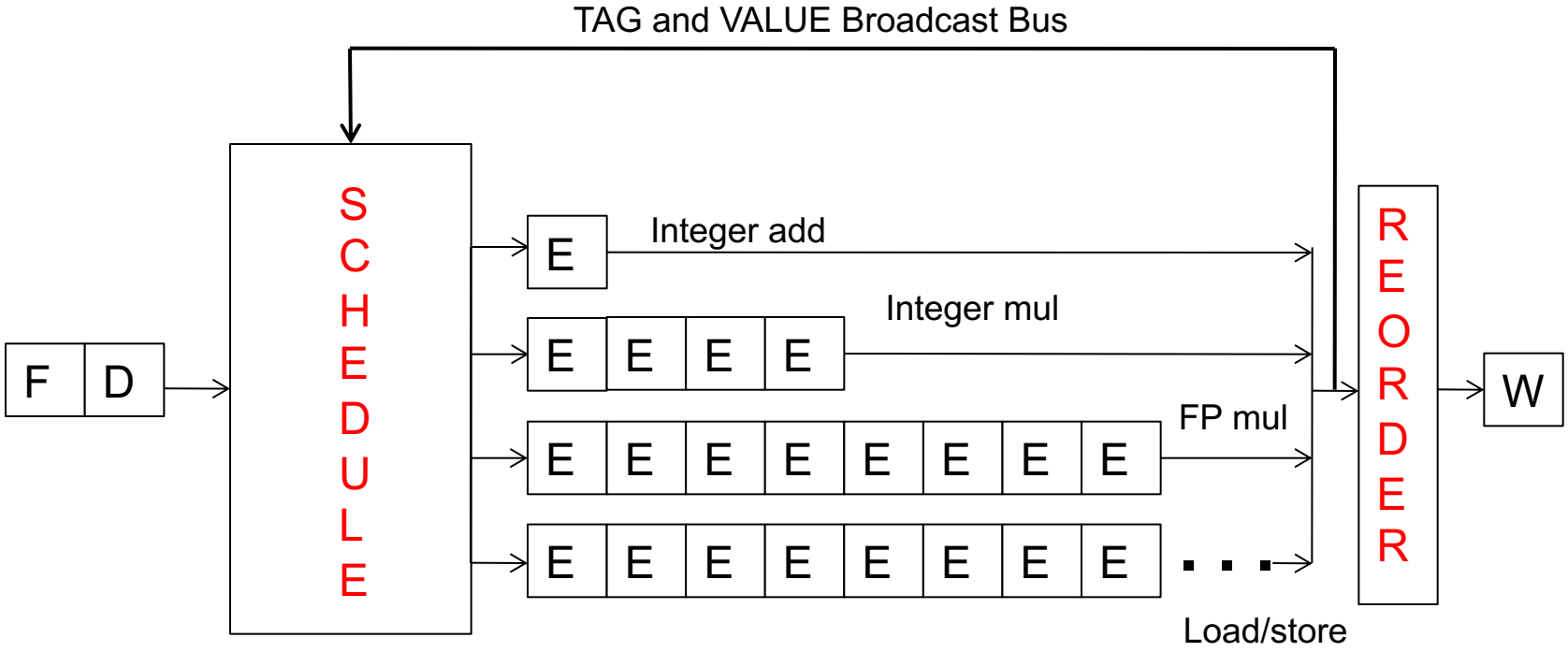
Tomasulo's Algorithm

- OoO with register renaming invented by Robert Tomasulo
  - Used in IBM 360/91 Floating Point Units
  - **Read:** Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of R&D, Jan. 1967.

- What is the major difference today?
  - Precise exceptions: IBM 360/91 did NOT have this
  - 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.

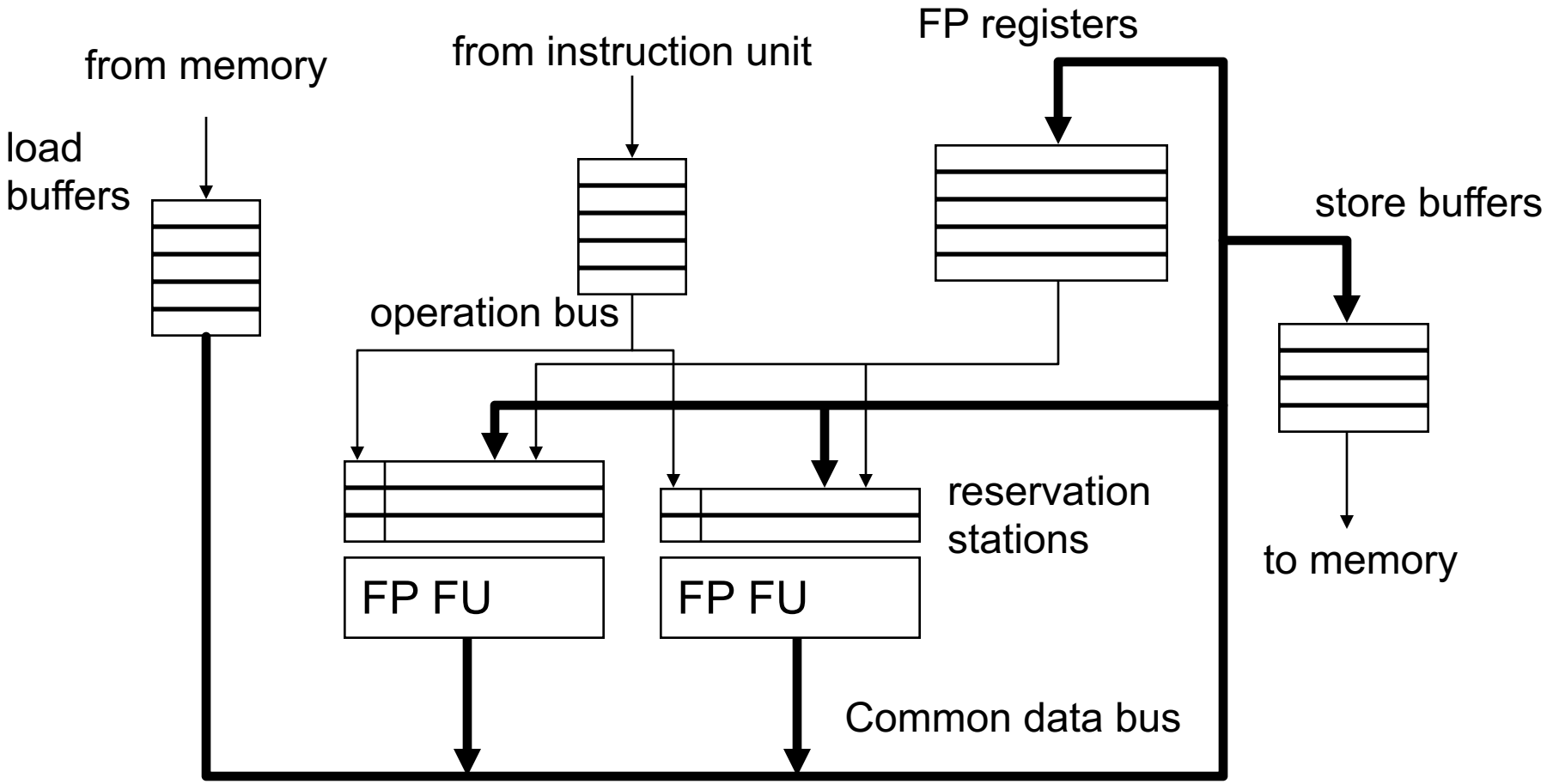
- Variants used in most high-performance processors
  - Initially in Intel Pentium Pro, AMD K5,
  - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15

Two Humps in a Modern Pipeline

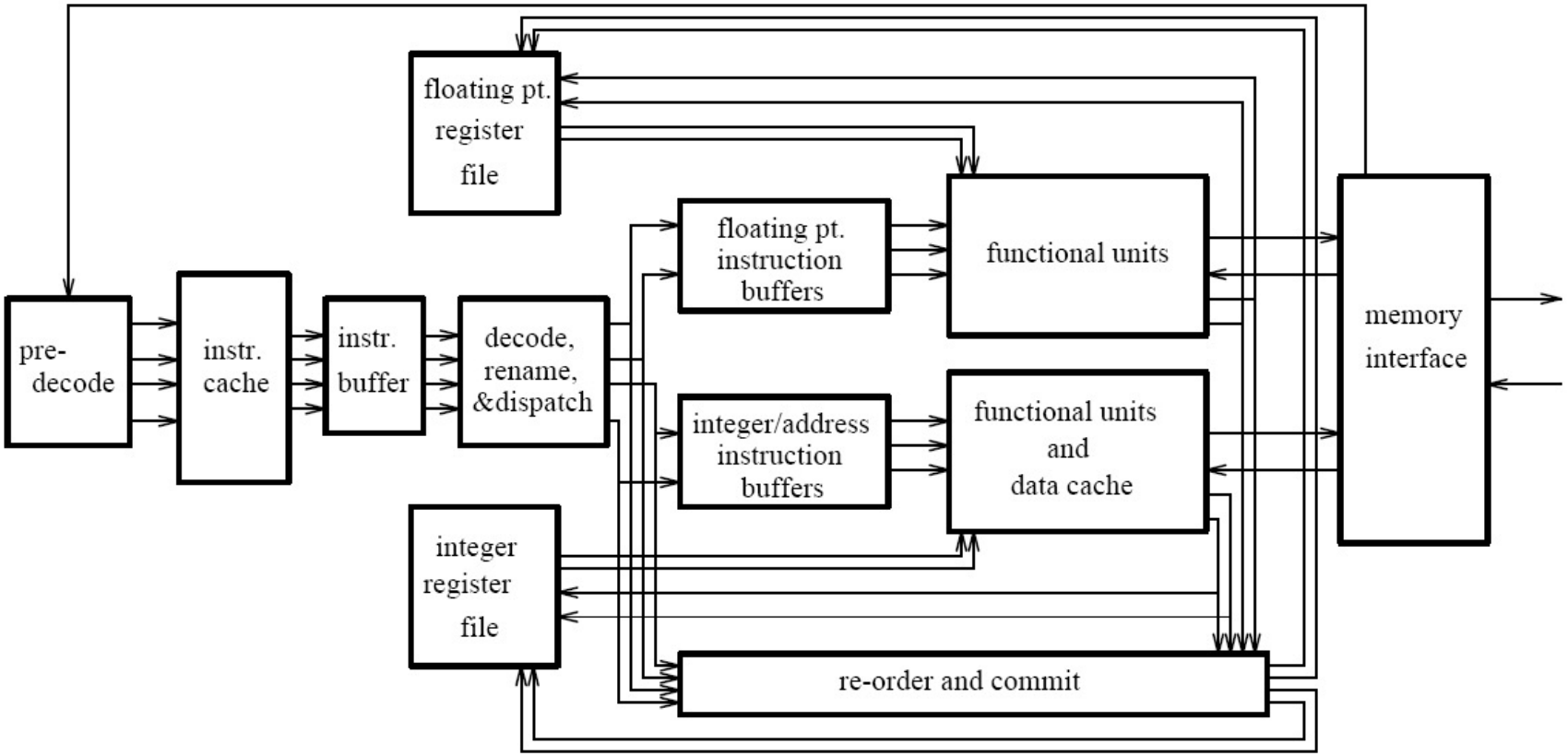


- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

Tomasulo's Machine: IBM 360/91



General Organization of an OOO Processor



Register Renaming

- Output and anti dependencies are not true dependencies
  - 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- dependencies
  - 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)

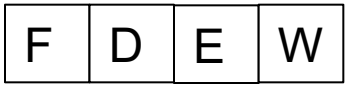
	tag	value	valid?
R0			1
R1			1
R2			1
R3			1
R4			1
R5			1
R6			1
R7			1
R8			1
R9			1

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

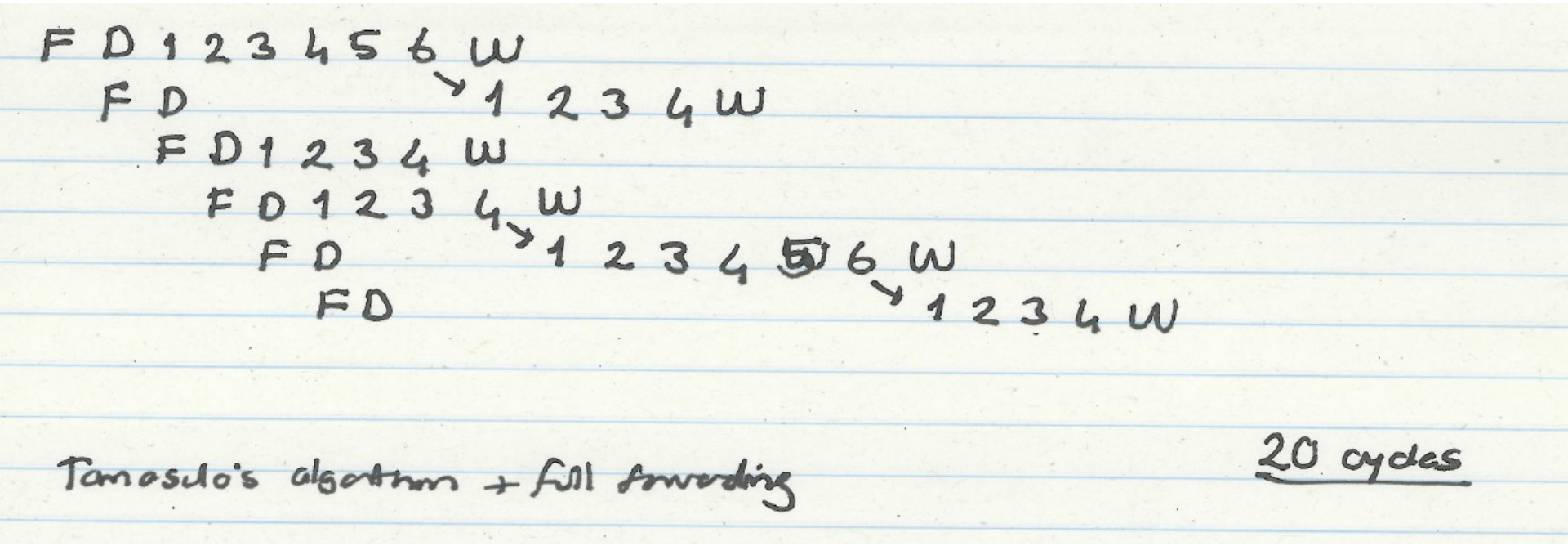


- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
  - in a non-pipelined machine
  - in an in-order-dispatch pipelined machine with imprecise exceptions (no forwarding and full forwarding)
  - in an out-of-order dispatch pipelined machine imprecise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

Stall vs. Forwarding from ROB

OoO execution with ROB Forwarding

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



How it works

- RAT (Register Alias Table)

	v?	tag	value
R0	1		
R1	1		
R2	1		
R3	1		

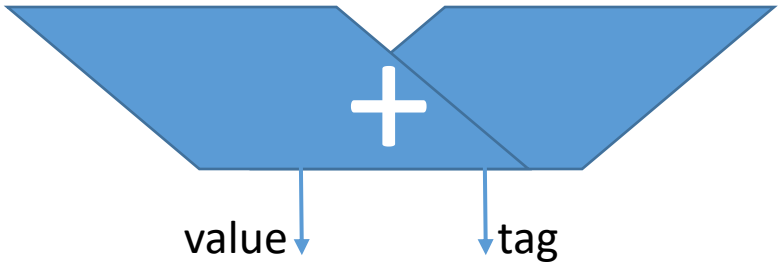
- Reservation Station for Functional Unit (FU)

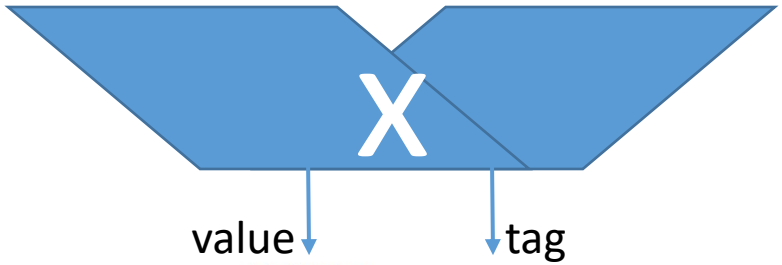
	v?	tag	value
a	1		
b	1		
c	1		
d	1		

	v?	tag	value
a	1		
b	1		
c	1		
d	1		

	v?	tag	value
x	1		
y	1		
z	1		
w	1		

	v?	tag	value
x	1		
y	1		
z	1		
w	1		





Cycle 0

- RAT all valid
- Let's assume R0=0, R1=1, R2=2, R3=3, …
- Reservation station are all invalid

Cycle2

- Decode MUL R3 ← R1, R2
- Allocate MUL ALU RS
- R3 is invalid

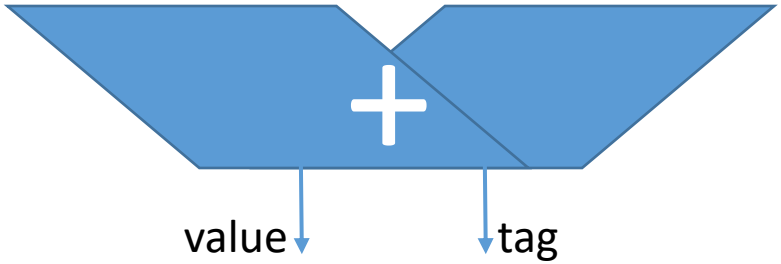
	v?	tag	value
R1	1		1
R2	1		2
R3	0	x	?
R4	1		

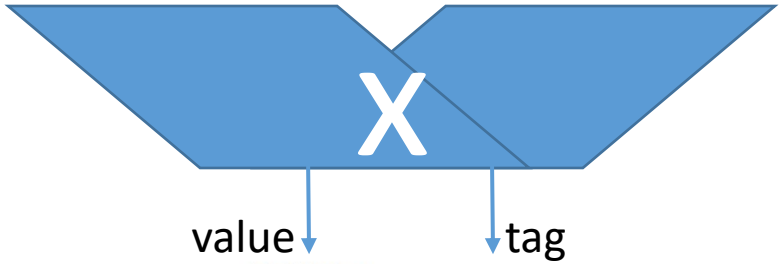
	v?	tag	value
a	1		
b	1		
c	1		
d	1		

	v?	tag	value
a	1		
b	1		
c	1		
d	1		

	v?	tag	value
x	1	~	1
y	1		
z	1		
w	1		

	v?	tag	value
x	1	~	2
y	1		
z	1		
w	1		





Cycle 3

- MUL ALU begins execution R3 invalid
- Decode ADD R5 ← R3, R4

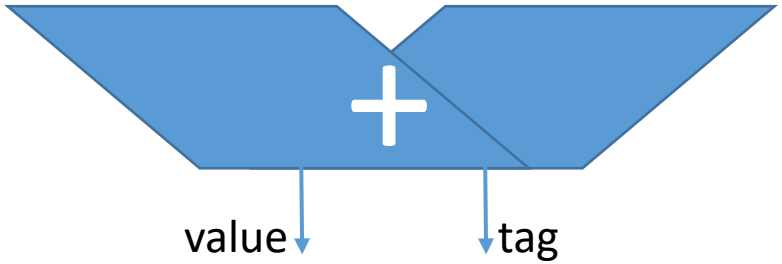
	v?	tag	value
R3	0	x	?
R4	1	~	4
R5	0	a	?
R6	1		

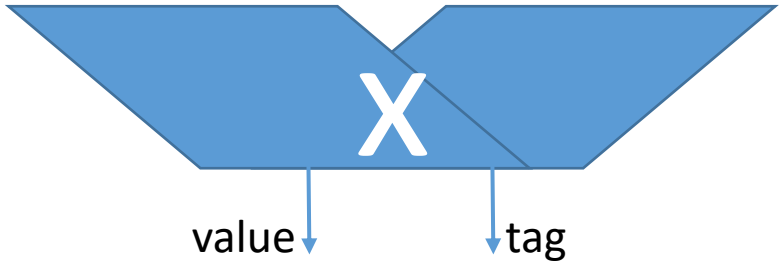
	v?	tag	value
a	0	x	?
b	1		
c	1		
d	1		

	v?	tag	value
a	1	~	4
b	1		
c	1		
d	1		

	v?	tag	value
x	1	~	1
y	1		
z	1		
w	1		

	v?	tag	value
x	1	~	2
y	1		
z	1		
w	1		





Cycle 4

- ADD cannot begin execution (Stall in RS_a)
- MUL consumed 2 cycles
- Decode ADD R7 ← R2, R6

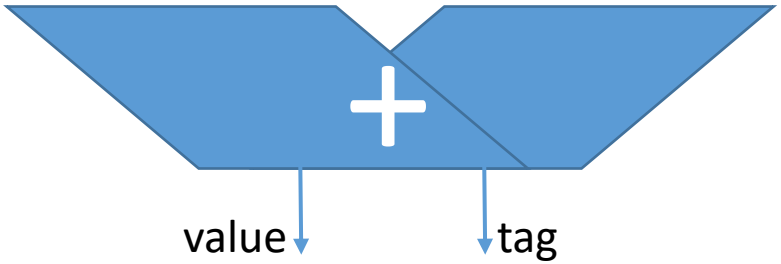
	v?	tag	value
R2	1	~	2
R3	0	x	?
R4	1	~	4
R5	1	a	?
R6	1	~	6
R7	0	b	?

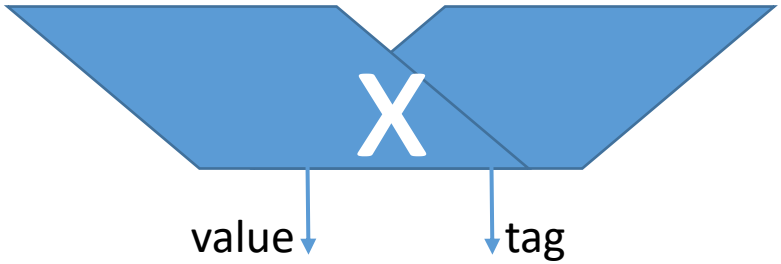
	v?	tag	value
a	0	x	?
b	1	~	2
c	1		
d	1		

	v?	tag	value
a	1	~	4
b	1	~	6
c	1		
d	1		

	v?	tag	value
x	1	~	1
y	1		
z	1		
w	1		

	v?	tag	value
x	1	~	2
y	1		
z	1		
w	1		





Cycle 7

- ADD cannot begin execution (Stall in RS_a)
- MUL consumed 2 cycles
- Decode ADD R7 ← R2, R6

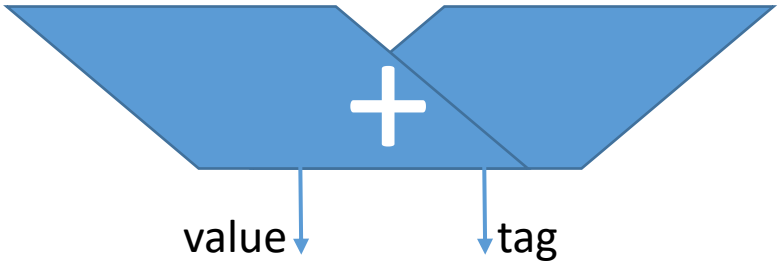
	v?	tag	value
R2	1	~	2
R3	0	x	?
R4	1	~	4
R5	1	a	?
R6	1	~	6
R7	0	b	?
R8	1	~	8
R9	1	~	9
R10	0	c	?
R11	0	y	?

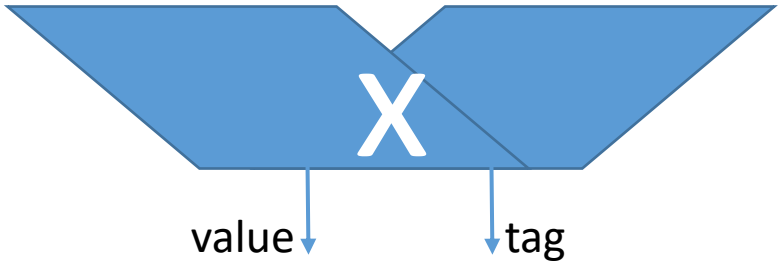
	v?	tag	value
a	0	x	?
b	1	~	2
c	1	~	8
d	0	a	?

	v?	tag	value
a	1	~	4
b	1	~	6
c	1	~	9
d	0	y	?

	v?	tag	value
x	1	~	1
y	0	b	?
z	1		
w	1		

	v?	tag	value
x	1	~	2
y	0	c	?
z	1		
w	1		





And so on…

- Brief overview of OoO.
  - Fetch in-order, execute OoO, commit in-order
  - Do not wait previous long-running instruction
  - Execute as fast as you can (when no flow-dependency)
  - Register renaming
- Recent trends
  - SIMD, MIMD parallel data flow model
  - Parallel programming many cores utilization
  - For large data handling…
  - 2K cores programming
- Next week, we will move on cache/memory hierarchy