18-447
Computer Architecture
Lecture 12: Out-of-Order Execution (Dynamic Instruction Scheduling)

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2015, 2/13/2015

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, …
- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, …
- Alternative Approaches to Instruction Level Parallelism

Reminder: Announcements

- Lab 3 due next Friday (Feb 20)
  - Pipelined MIPS
  - Competition for high performance
    - You can optimize both cycle time and CPI
    - Document and clearly describe what you do during check-off

- Homework 3 due Feb 25
  - A lot of questions that enable you to learn the concepts via hands-on exercise
  - Remember this is all for your benefit (to learn and prepare for exams)
    - HWs have very little contribution to overall grade
    - Solutions to almost all questions are online anyway

Readings Specifically for Today

- 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

- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Recap of Last Lecture

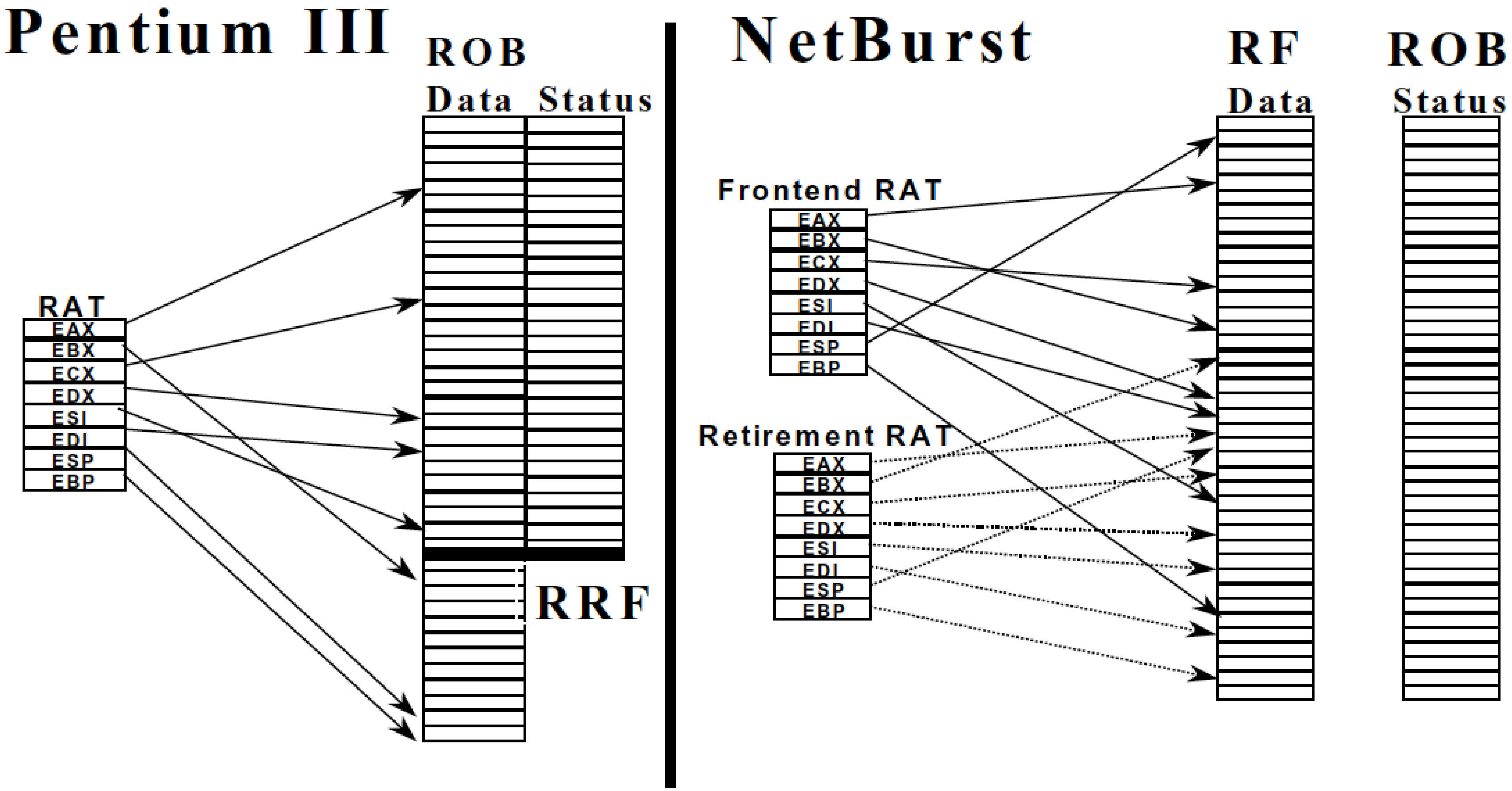
- Issues with Multi-Cycle Execution
- Exceptions vs. Interrupts
- Precise Exceptions/Interrupts
- Why Do We Want Precise Exceptions?
- How Do We Ensure Precise Exceptions?
  - Reorder buffer
  - History buffer
  - Future register file (best of both worlds)
  - Checkpointing
- Register renaming with a reorder buffer
- How to Handle Exceptions
- How to Handle Branch Mispredictions
- Speed of State Recovery: Recovery and Interrupt Latency
  - Checkpointing
- Registers vs. Memory

Important: Register Renaming with a Reorder Buffer

- 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

Review: Register Renaming Examples



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

Review: Checkpointing Idea

- Goal: Restore the frontend state (future file) such that the correct next instruction after the branch can execute right away after the branch misprediction is resolved

- Idea: Checkpoint the frontend register state/map at the time a branch is decoded and keep the checkpointed state updated with results of instructions older than the branch
  - Upon branch misprediction, restore the checkpoint associated with the branch

  - Hwu and Patt, "Checkpoint Repair for Out-of-order Execution Machines," ISCA 1987.

Review: Checkpointing

- When a branch is decoded
  - Make a copy of the future file/map and associate it with the branch
- When an instruction produces a register value
  - All future file/map checkpoints that are younger than the instruction are updated with the value
- When a branch misprediction is detected
  - Restore the checkpointed future file/map for the mispredicted branch when the branch misprediction is resolved
  - Flush instructions in pipeline younger than the branch
  - Deallocate checkpoints younger than the branch

Review: 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)

Maintaining Speculative Memory State: Stores

- 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
- We will get back to this after today!

Remember:
Static vs. Dynamic Scheduling

Remember: Questions to Ponder

- What is the role of the hardware vs. the software in the order in which instructions are executed in the pipeline?
  - Software based instruction scheduling → static scheduling
  - Hardware based instruction scheduling → dynamic scheduling

- What information does the compiler not know that makes static scheduling difficult?
  - Answer: Anything that is determined at run time
    - Variable-length operation latency, memory addr, branch direction

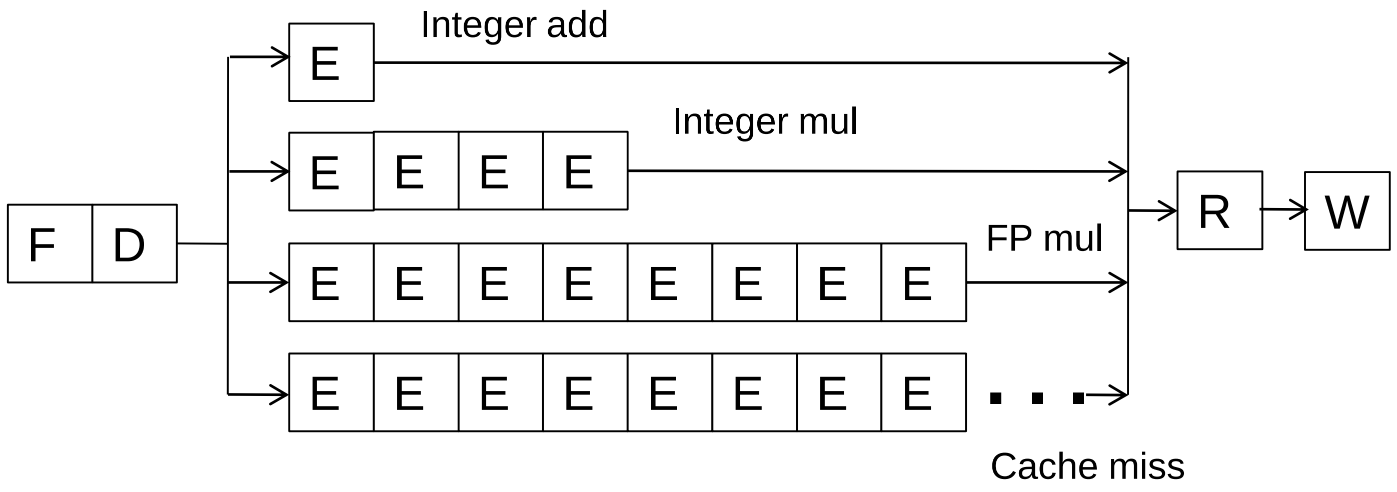
Dynamic Instruction Scheduling

- Hardware has knowledge of dynamic events on a per-instruction basis (i.e., at a very fine granularity)
  - Cache misses
  - Branch mispredictions
  - Load/store addresses

- Wouldn't it be nice if hardware did the scheduling of instructions?

Out-of-Order Execution (Dynamic Instruction Scheduling)

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

Can We Do Better?

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

```
IMUL  R3 ← 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 ← R9, R9              ADD   R7 ← R9, R9
```

- 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

Preventing Dispatch Stalls

- Multiple ways of doing it
- You have already seen at least THREE:
  - 1.
  - 2.
  - 3.
- What are the disadvantages of the above three?

- Any other way to prevent dispatch stalls?
  - Actually, you have briefly seen the basic idea before
    - Dataflow: fetch and "fire" an instruction when its inputs are ready
  - Problem: in-order dispatch (scheduling, or execution)
  - Solution: out-of-order dispatch (scheduling, or execution)

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 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:

F	D	E	E	E	E	R	W								
	F	D		STALL		E	R	W							
		F		STALL		D	E	R	W						
						F	D	E	E	E	E	E	R	W	
							F	D			STALL		E	R	W

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

- Out-of-order dispatch + precise exceptions:

F	D	E	E	E	E	R	W				
	F	D		WAIT		E	R	W			
		F	D	E	R				W		
			F	D	E	E	E	E	R	W	
				F	D		WAIT		E	R	W

This slide is actually correct

- 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

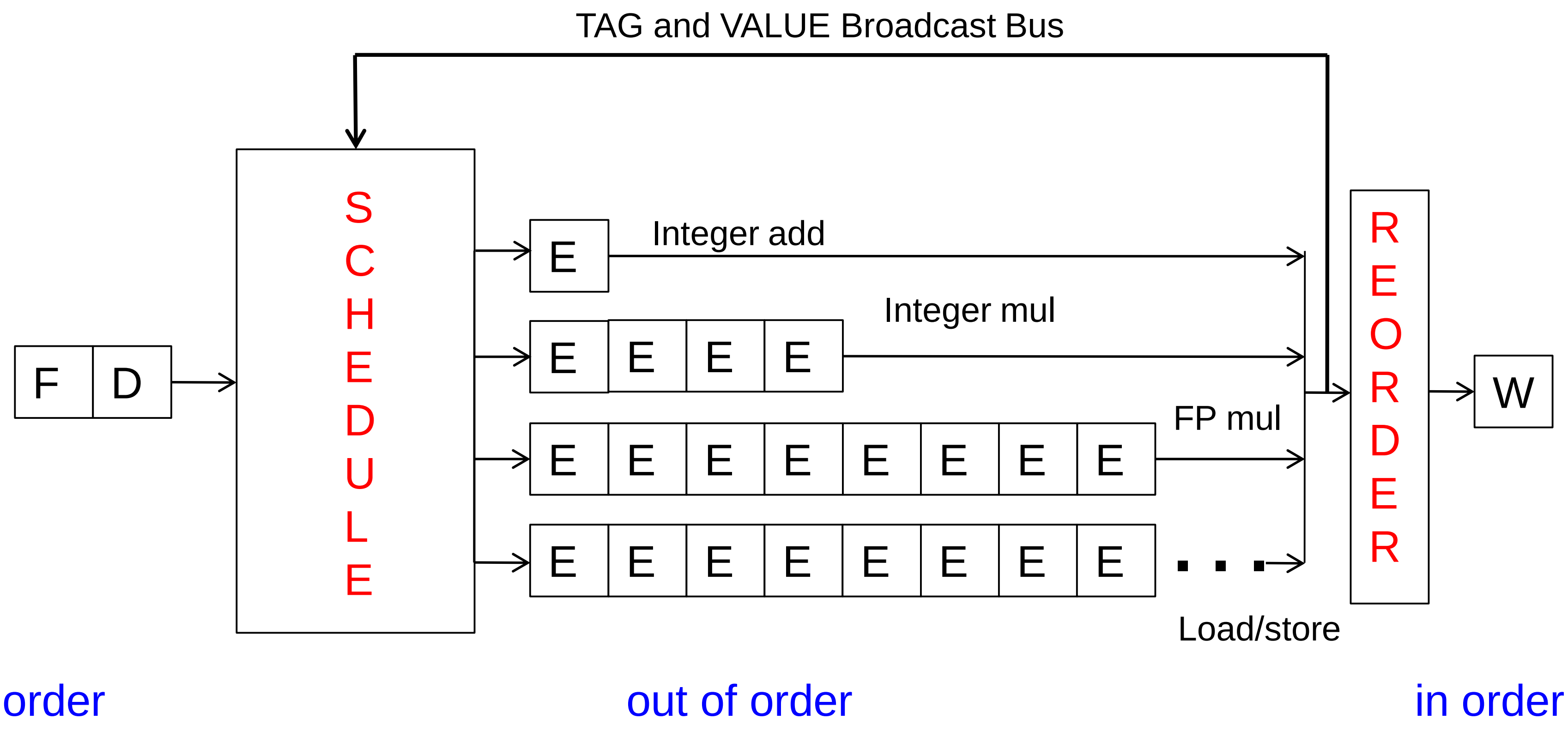
Tomasulo's Algorithm

- 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.

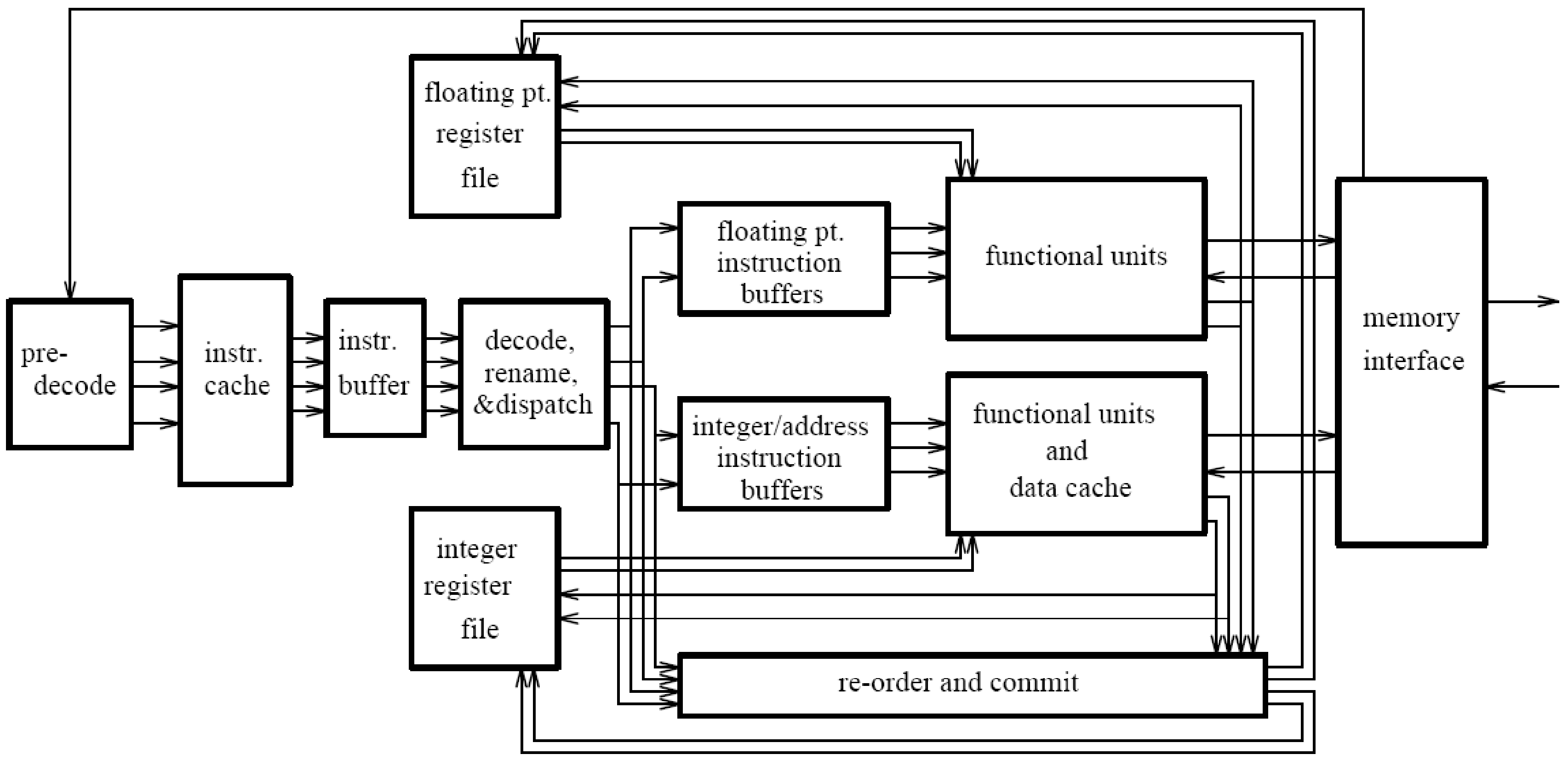
- Variants are used in most high-performance processors
  - Initially in Intel Pentium Pro, AMD K5
  - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15

Two Humps in a Modern Pipeline



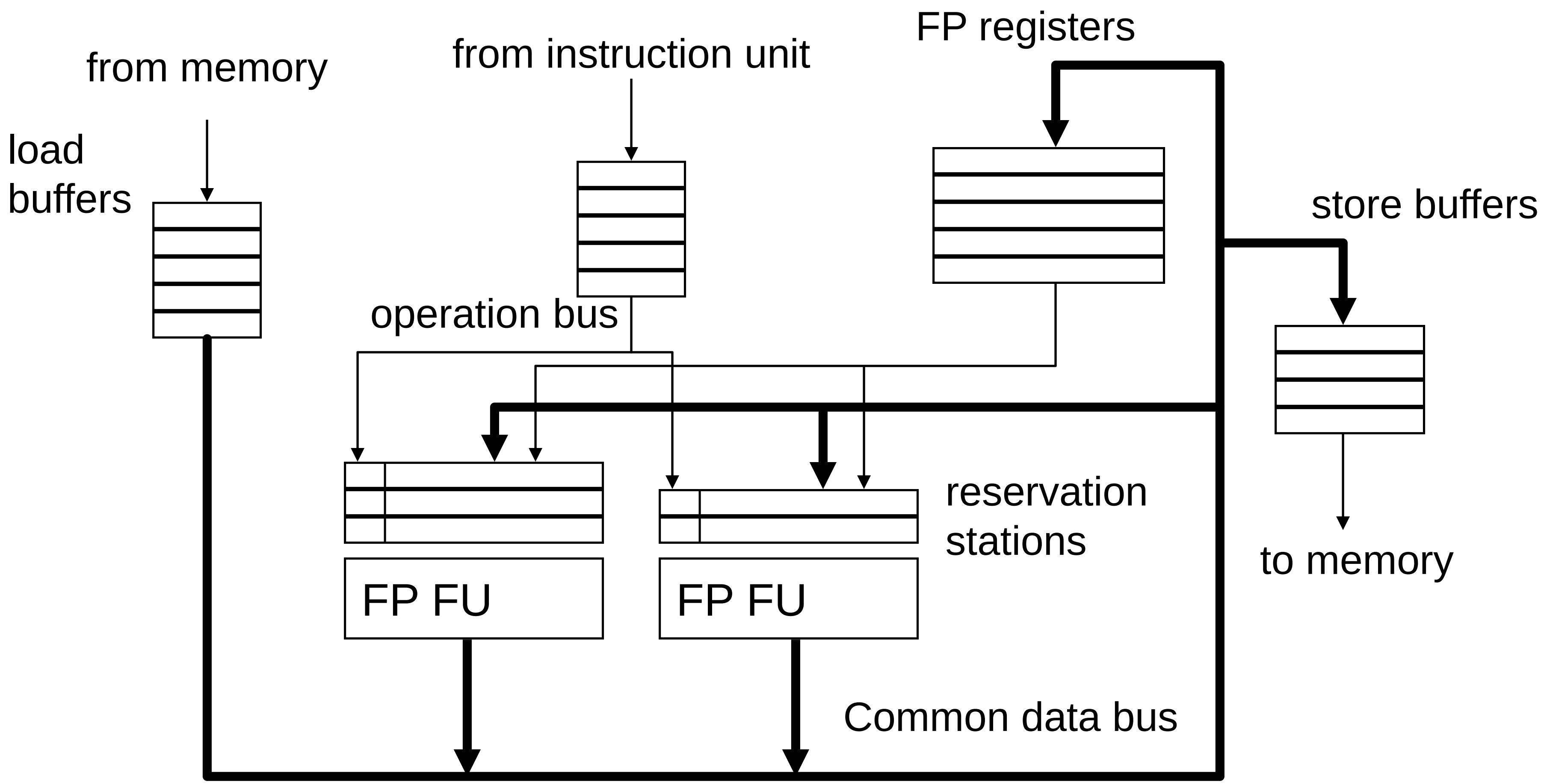
- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

General Organization of an OOO Processor


floating pt. register file
pre- decode
instr. cache
instr. buffer
decode, rename, &dispatch
floating pt. instruction buffers
functional units
integer/address instruction buffers
functional units and data cache
memory interface
integer register file
re-order and commit

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

Tomasulo’s Machine: IBM 360/91



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!

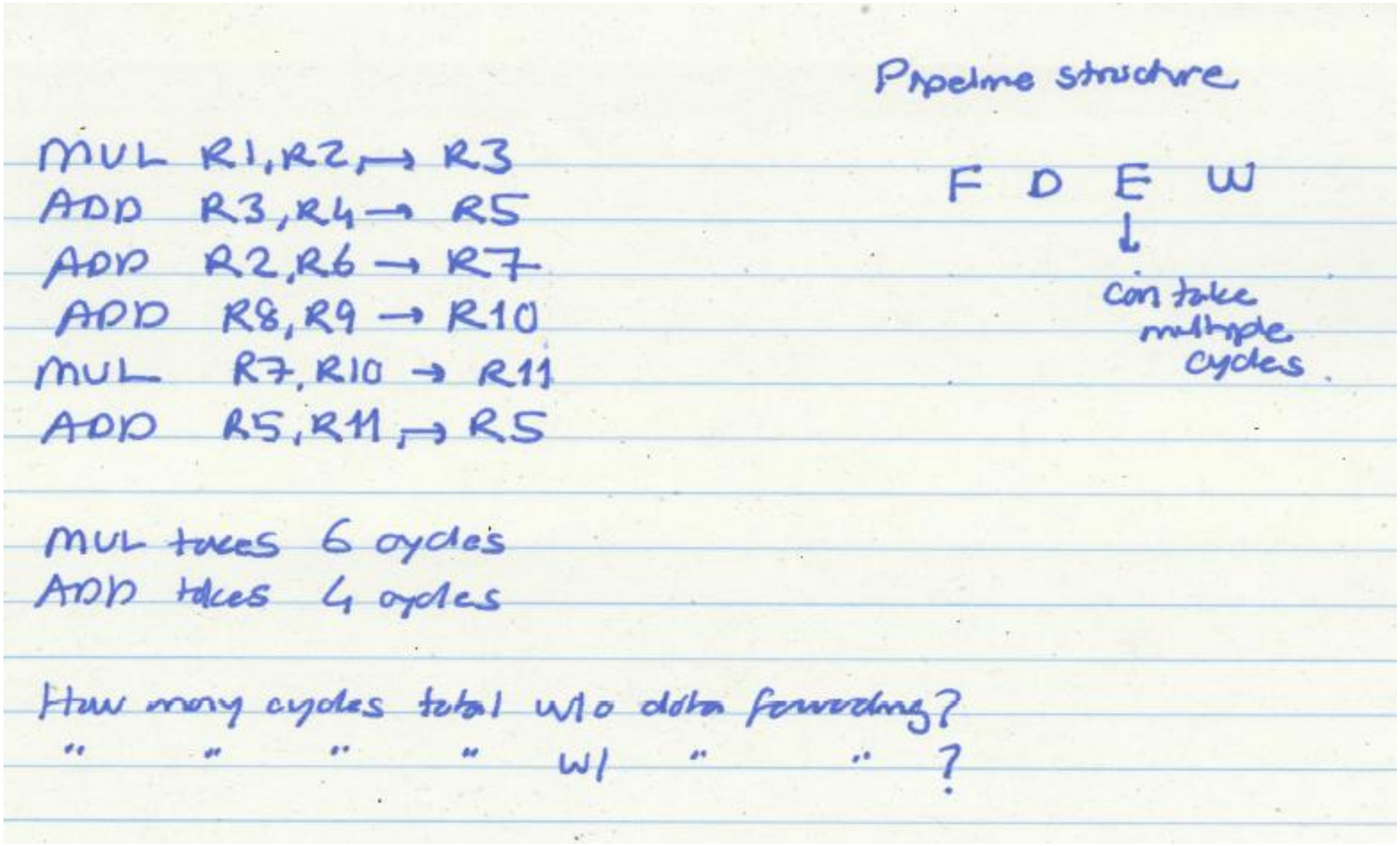
An Exercise

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

F	D	E	W

- 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)

Exercise Continued



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

Pipeline structure

F D E W

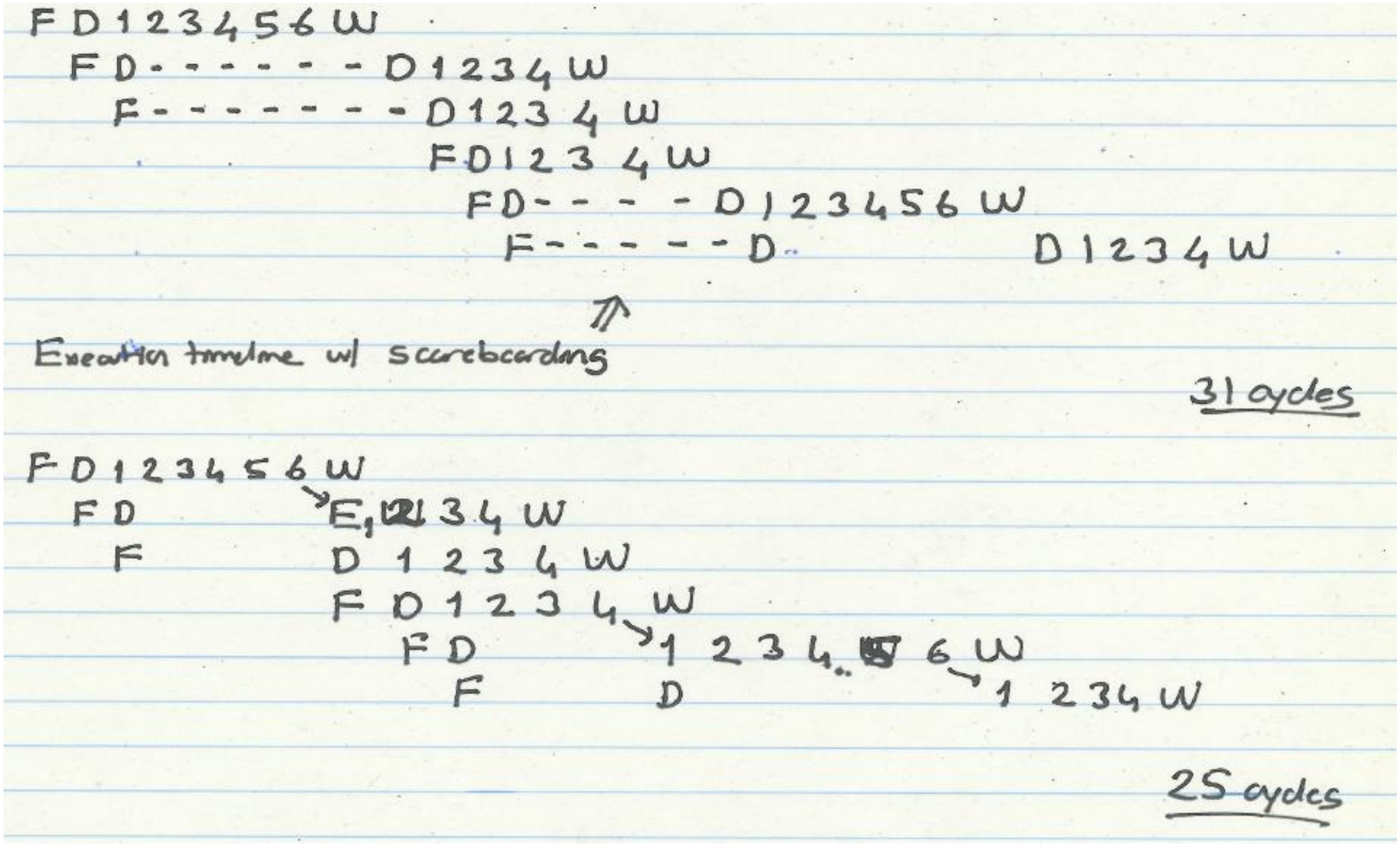
$\downarrow$ (from E)

can take multiple cycles.

MUL takes 6 cycles
ADD takes 4 cycles

How many cycles total w/o data forwarding?
" " " " w/ " " ?

Exercise Continued

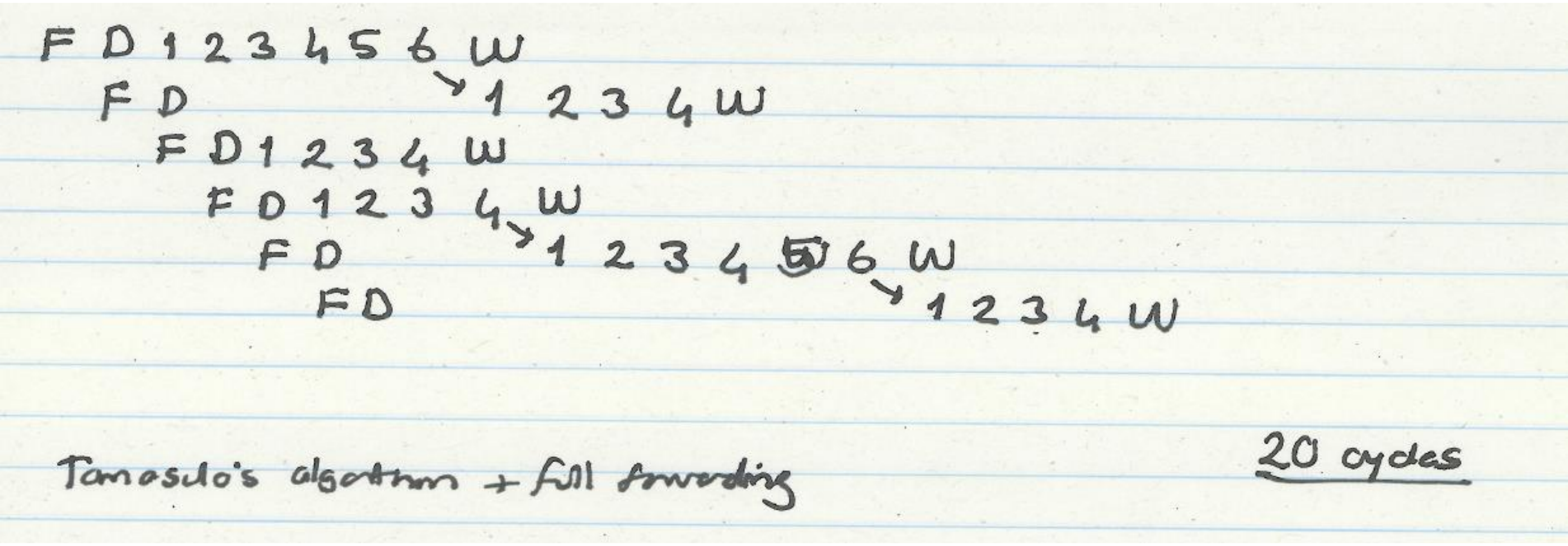


```
F D 1 2 3 4 5 6 W
  F D - - - - - - - D 1 2 3 4 W
    F - - - - - - - - D 1 2 3 4 W
                      F D 1 2 3 4 W
                        F D - - - - D 1 2 3 4 5 6 W
                          F - - - - - D                 D 1 2 3 4 W
                                 ⇗
Execution timeline w/ scoreboarding
                                                           31 cycles

F D 1 2 3 4 5 6 W
  F D            ↘E1 2 3 4 W
    F             D 1 2 3 4 W
                  F D 1 2 3 4 W
                      F D          ↘1 2 3 4 5 6 W
                        F           D              ↘1 2 3 4 W
                                                           25 cycles
```

Exercise Continued

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

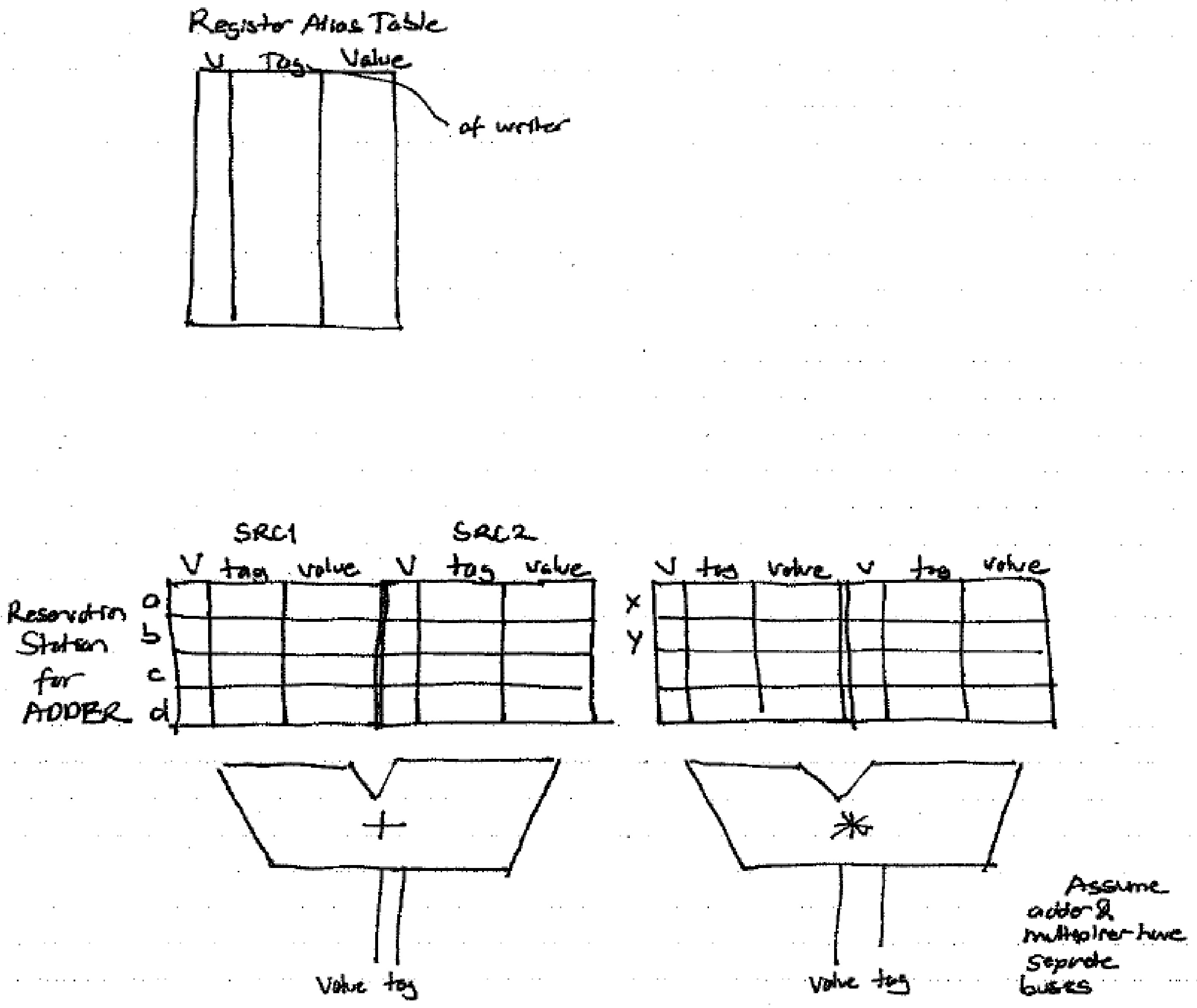


```
F D 1 2 3 4 5 6 W
  F D           ↘ 1 2 3 4 W
    F D 1 2 3 4 W
      F D 1 2 3 4 W
        F D       ↘ 1 2 3 4 5 6 W
          F D                   ↘ 1 2 3 4 W
```

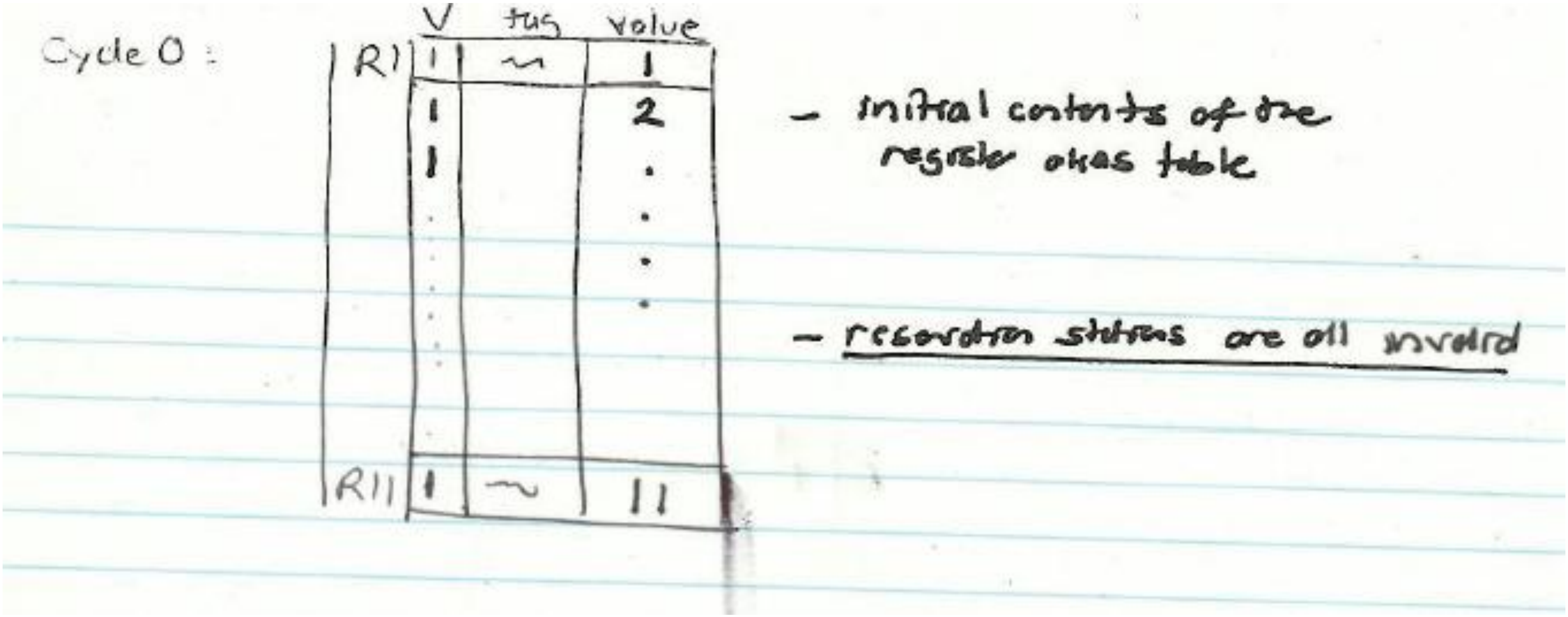
Tomasulo's algorithm + full forwarding

20 cycles

How It Works


Register Alias Table
V
Tag
Value
of writer
SRC1
SRC2
V
tag
value
V
tag
value
Reservation Station for ADDER
a
b
c
d
x
y
V
tag
value
V
tag
value
+
*
Value
tag
Value
tag
Assume adder & multiplier have separate buses

Cycle 0



Cycle 0 :

	V	tag	value
R1	1	~	1
	1		2
	1		.
	.		.
	.		.
	.		.
R11	1	~	11

- initial contents of the register alias table
- reservation stations are all invalid

Cycle 2

cycle 2 : MUL R1, R2 → R3 – reads its sources from the RAT

– writes to its destination in the RAT (renames its destination

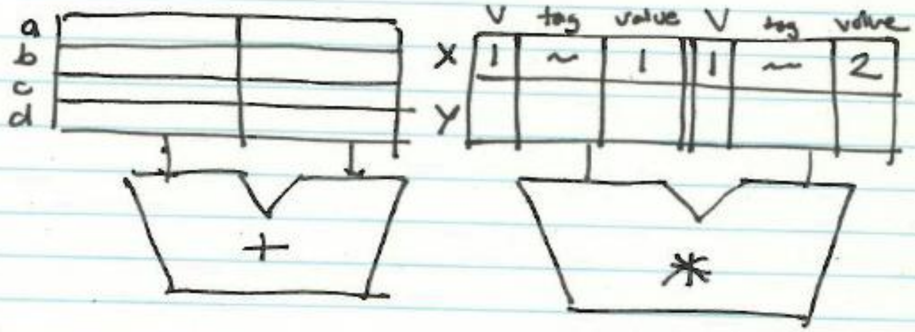
→ allocates a reservation station entry

→ allocates a tag for its destination register

– places its sources in the reservation station entry that is allocated.

end of cycle 2:

	V	tag	value
R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R11	1	~	11



– MUL at X becomes ready to execute
(What if multiple instructions become ready at the same time)
→ both of its sources are valid in the reservation station X

Cycle 3

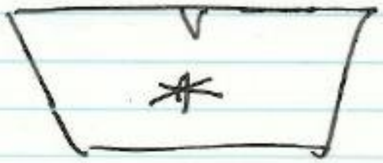
cycle 3: → MUL at X starts execution
→ ADD R3, R4 → R5 gets renamed and placed into the ADDER reservation stations

end of cycle 3:

R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R5	0	a	~
R6	1	~	6
R11	1	~	11

	V	tag	value	V	tag	value
a	0	X	~	1	~	4
b	1					
c						
d						

X	1	~	1	1	~	2
Y						



– ADD at a cannot be ready to execute because one of its sources is not ready
→ It is waiting for the value with the tag X to be broadcast (by the MUL in X)

Aside: Does the tag need to be associated with the RS entry of the producer?

Answer: No: Tag is a tag for the value that is communicated.

enables data-flow like value communication

RS is a place to hold the instructions while they become ready.

These two are completely orthogonal.

Cycle 4

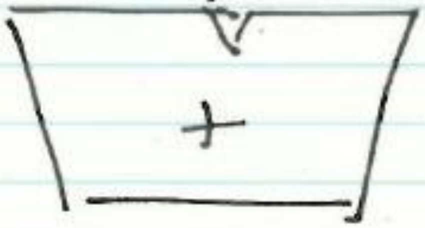
5

cycle 4: — ADD R2, R6 → R7 gets renamed and placed into RS

end of cycle 4:

R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R5	0	a	~
R6	1	~	6
R7	0	b	~
R11	1	~	11

a	0	X	~	1	~	4
b	1	~	2	1	~	6
c						
d						



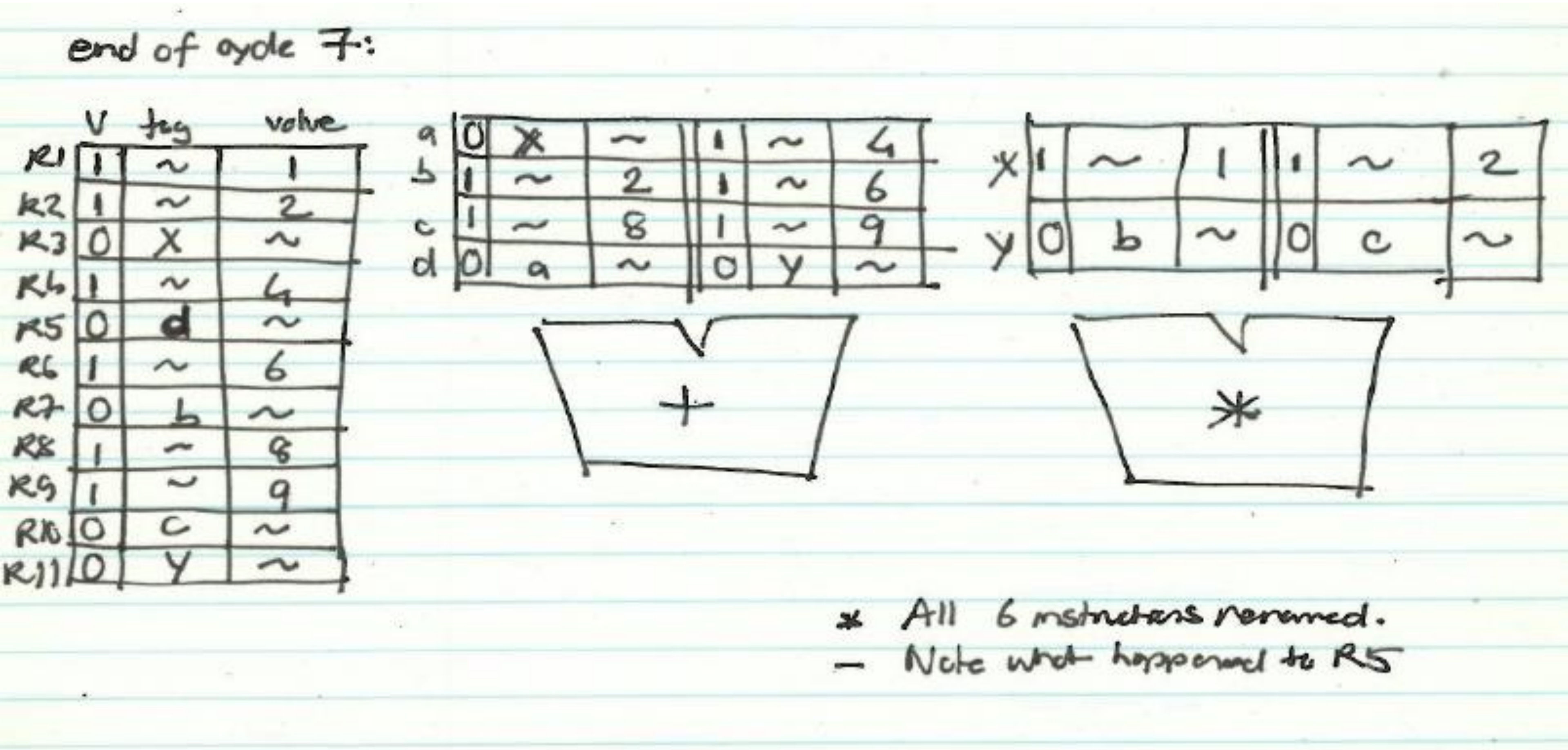
Same as cycle 3

— ADD at b becomes ready to execute (both sources are ready!)

— At cycle 5, it is sent to the adder (out-of-program order!)

→ It is executed before the add in a

Cycle 7



end of cycle 7:

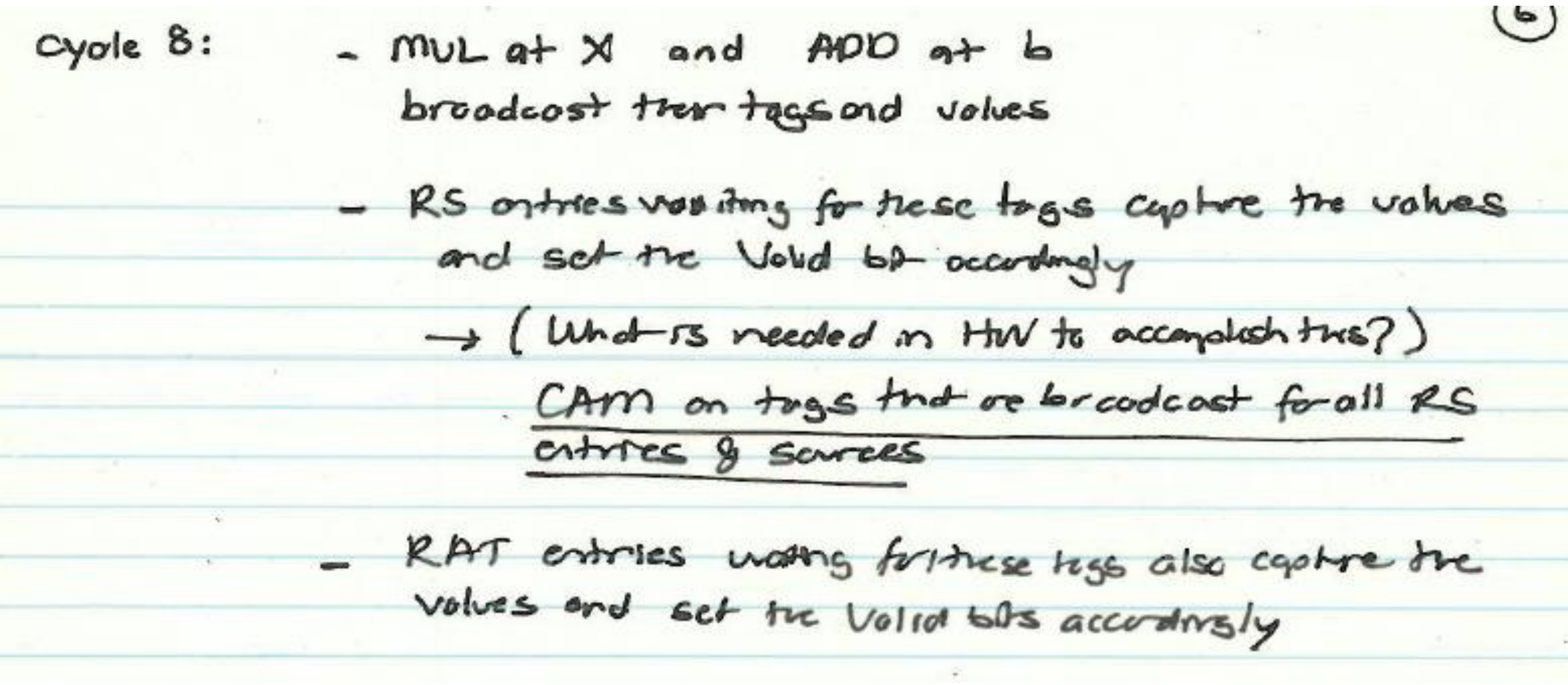
	V	tag	value
R1	1	~	1
R2	1	~	2
R3	0	x	~
R4	1	~	4
R5	0	d	~
R6	1	~	6
R7	0	b	~
R8	1	~	8
R9	1	~	9
R10	0	c	~
R11	0	y	~

a	0	x	~	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a	~	0	y	~

x	1	~	1	1	~	2
y	0	b	~	0	c	~

* All 6 instructions renamed.
– Note what happened to R5

Cycle 8



6

Cycle 8:

- MUL at x and ADD at b broadcast their tags and values
- RS entries waiting for these tags capture the values and set the Valid bit accordingly

  → (What is needed in HW to accomplish this?)

  <u>CAM on tags that are broadcast for all RS entries & sources</u>
- RAT entries waiting for these tags also capture the values and set the Valid bits accordingly

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?

- What can potentially become the critical path?
  - Tag broadcast → value capture → instruction wake up

- How can you reduce the potential critical paths?

An Exercise, with 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
```

F	D	E	R	W

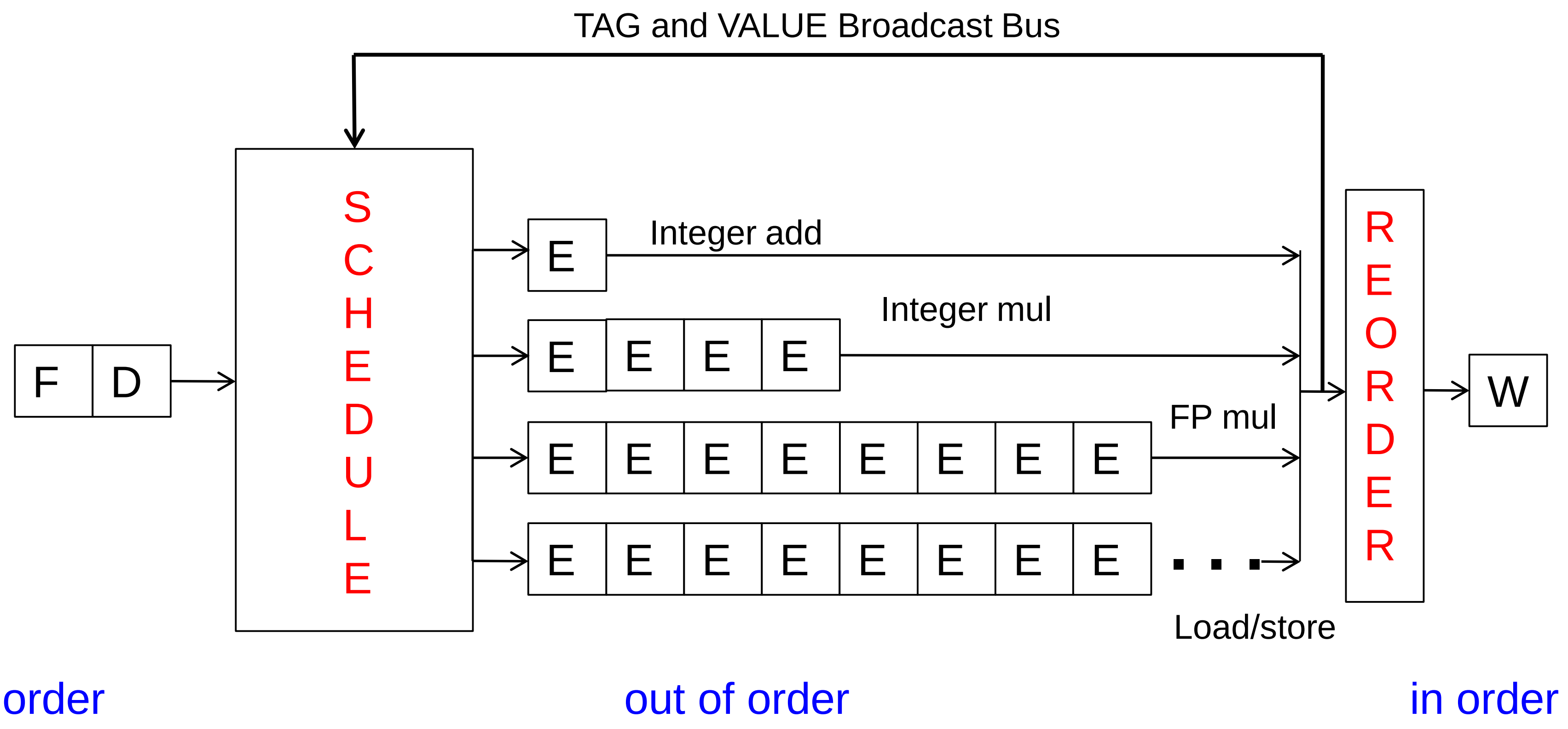
- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
  - in a non-pipelined machine
  - in an in-order-dispatch pipelined machine with reorder buffer (no forwarding and full forwarding)
  - in an out-of-order dispatch pipelined machine with reorder 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 register alias table (essentially a future file) when it completes execution
- An instruction updates the architectural register file when it is the oldest in the machine and has completed execution

Out-of-Order Execution with Precise Exceptions

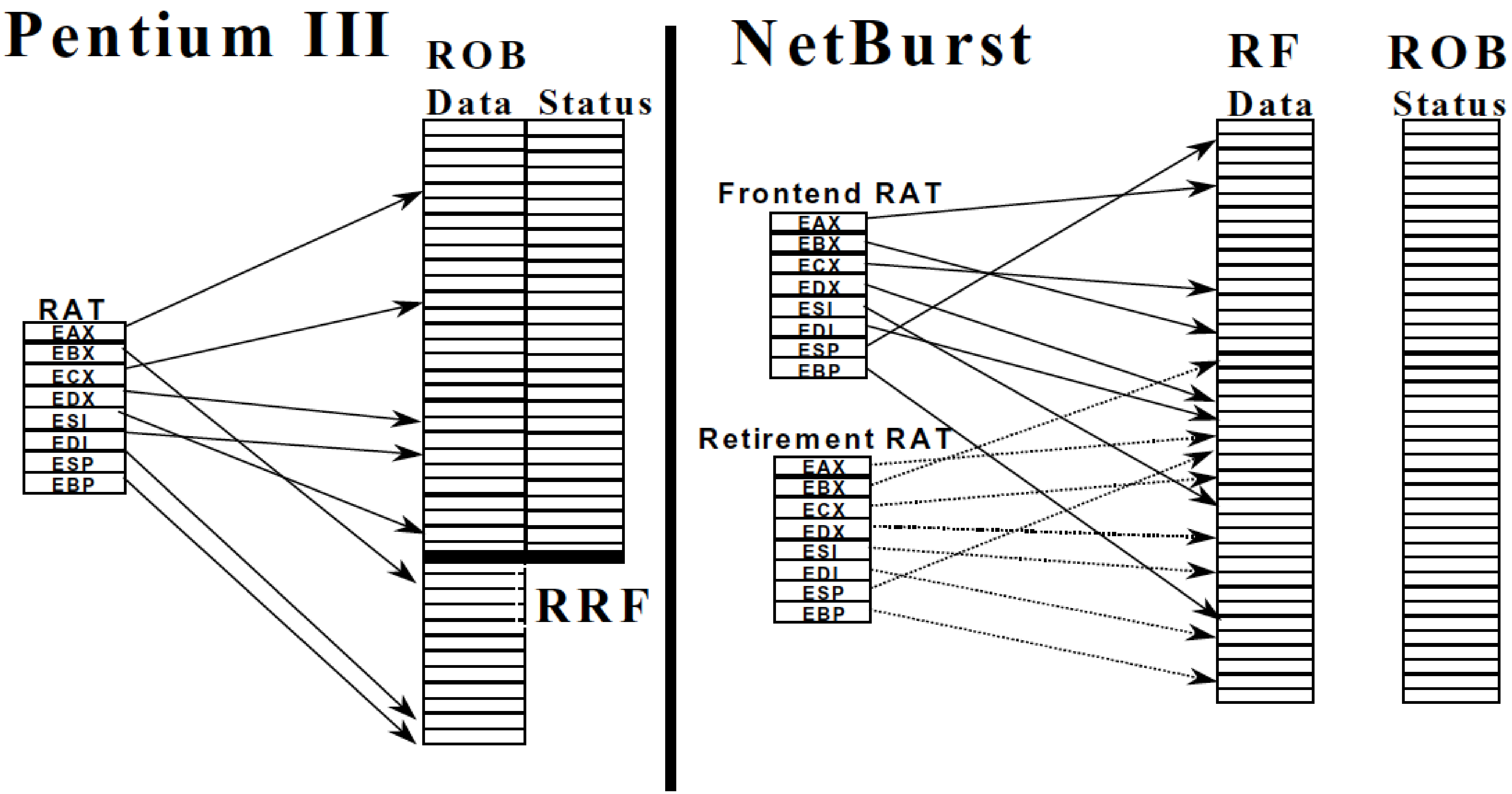


- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

Modern OoO Execution w/ Precise Exceptions

- Most modern processors use
  - Reorder buffer to support in-order retirement of instructions
  - A single register file to store registers (speculative and architectural) – INT and FP are still separate
  - Future register map → used for renaming
  - Architectural register map → used for state recovery

An Example from Modern Processors



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

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 dependencies, enables linking of producer to consumers
- Buffering enables the pipeline to move for independent ops
- 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?

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
```

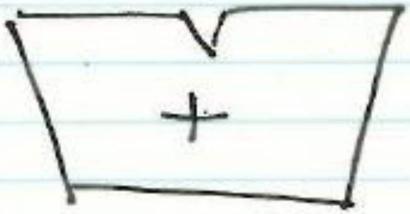
State of RAT and RS in Cycle 7

end of cycle 7:

	V	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	~

a	0	x	~	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a	~	0	y	~

x	1	~	1	1	~	2
y	0	b	~	0	c	~



* All 6 instructions renamed.
– Note what happened to R5

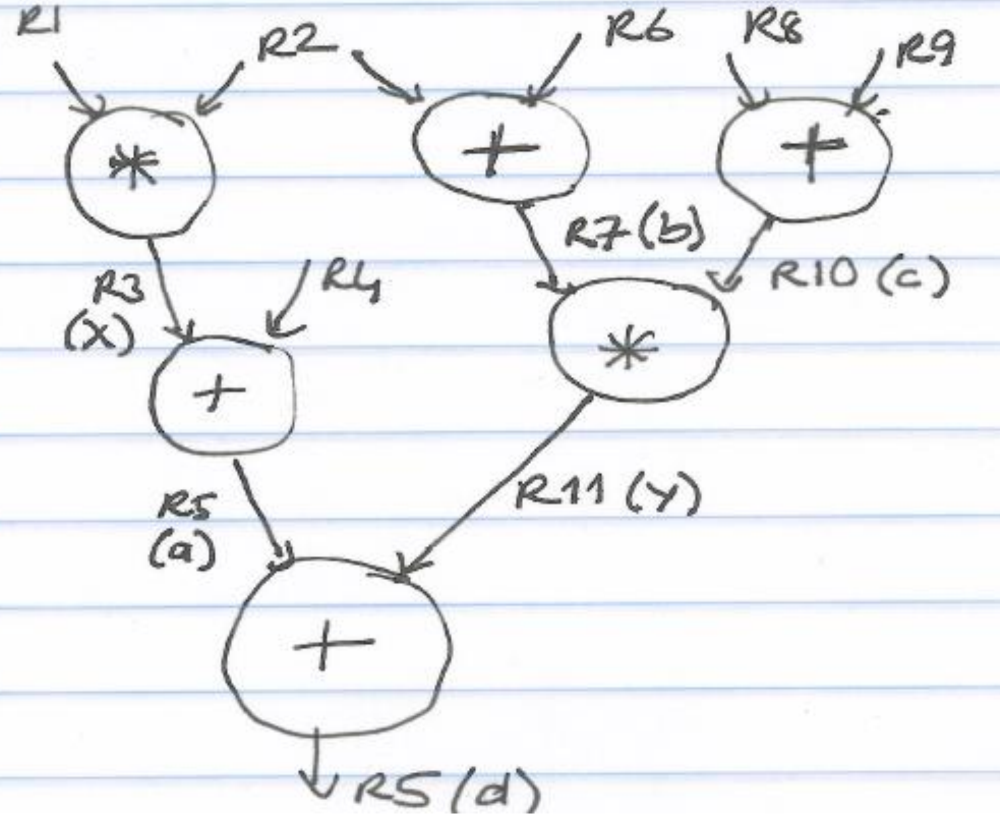
Dataflow Graph

```
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

Arcs: tags in Tomasulo's algorithm



We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Restricted Data Flow

- An out-of-order machine is a “restricted data flow” machine
  - Dataflow-based execution is restricted to the microarchitecture level
  - ISA is still based on von Neumann model (sequential execution)

- Remember the data flow model (at the ISA level):
  - Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no instruction pointer
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received

Questions to Ponder

- Why is OoO execution beneficial?
  - What if all operations take single cycle?
  - Latency tolerance: OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently

- What if an instruction takes 500 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?
    - Active/instruction window size: determined by both scheduling window and reorder buffer size