15-740/18-740 Computer Architecture Lecture 10: Out-of-Order Execution

Prof. Onur Mutlu
Carnegie Mellon University

Fall 2011, 10/3/2011

Review: Solutions to Enable Precise Exceptions

- Reorder buffer
- History buffer
- Future register file
- Checkpointing
- 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.

More on Last Lecture

- Handling of branch mispredictions
- Handling of stores and loads

Today

- Out-of-order execution

Pipelining Issues: Branch Mispredictions

- A branch misprediction resembles an "exception"
  - Except it is not visible to software
- What about branch misprediction recovery?
  - Similar to exception handling except can be initiated before the branch is the oldest instruction
  - All three state recovery methods can be used
- Difference between exceptions and branch mispredictions?
  - Branch mispredictions more common: need fast recovery

Pipelining Issues: Stores

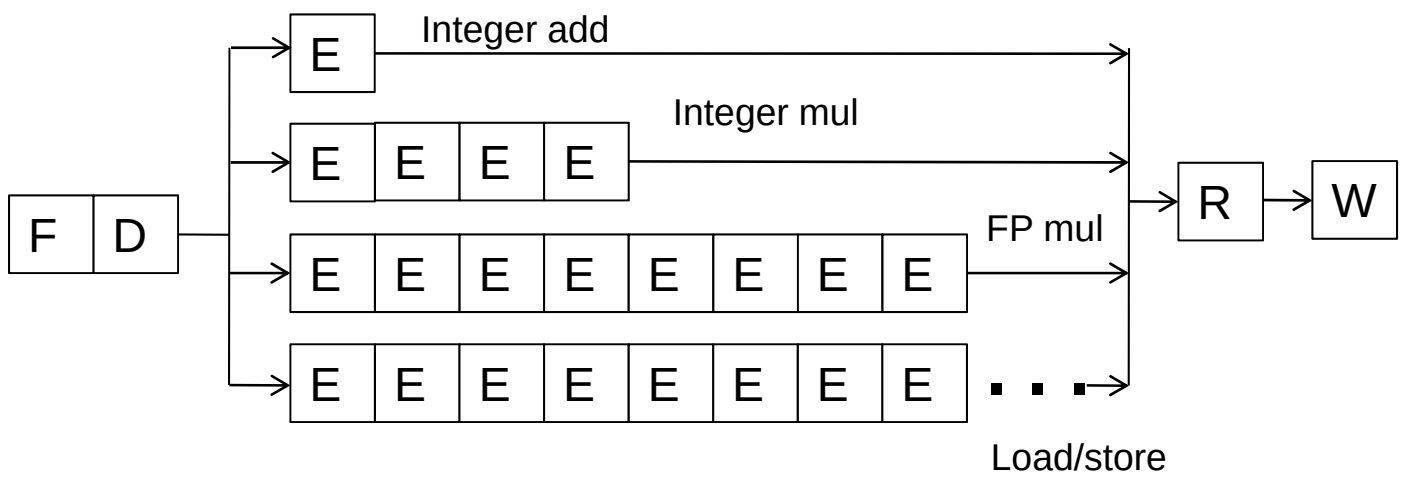
- Handling out-of-order completion of memory operations
  - UNDOing a memory write more difficult than UNDOing a register write. Why?
  - One idea: Keep store address/data in reorder buffer
    - How does a load instruction find its data?
  - Store/write buffer: Similar to reorder buffer, but used only 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

Summary: Precise Exceptions in Pipelining

- When the oldest instruction ready-to-be-retired is detected to have caused an exception, the control logic
  - Recovers architectural state (register file, IP, and memory)
  - Flushes all younger instructions in the pipeline
  - Saves IP and registers (as specified by the ISA)
  - Redirects the fetch engine to the exception handling routine

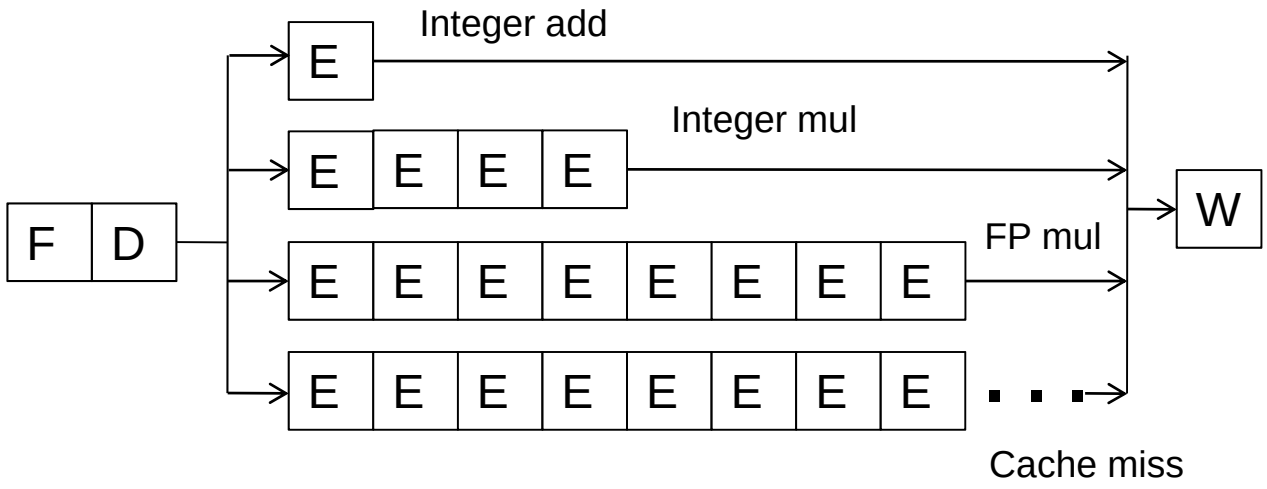
Putting It Together: In-Order Pipeline with Future File

- Decode (D): Access future file, allocate entry in reorder buffer, store buffer, check if instruction can execute, if so **dispatch** instruction
- Execute (E): Instructions can complete out-of-order, store-load dependencies determined
- Completion (R): Write result to reorder/store buffer
- Retirement/Commit (W): Write result to architectural register file or memory
- In-order dispatch/execution, out-of-order completion, in-order retirement



Out-of-Order Execution
(Dynamic Instruction Scheduling)

Review: 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
ADD   R3 ← R3, R1
ADD   R1 ← R6, R7
IMUL  R3 ← R6, R8
ADD   R7 ← R3, R9
```

```
LD    R3 ← R1 (0)
ADD   R3 ← R3, R1
ADD   R1 ← R6, R7
IMUL  R3 ← R6, R8
ADD   R7 ← R3, 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 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 (issue, execution)
  - Solution: out-of-order dispatch (issue, execution)

Terminology

- Issue vs. dispatch
- Scheduling
- Execution, completion, retirement/commit
- Graduation
- Out-of-order execution versus superscalar processing

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 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:

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 R3 ← R6, R8
ADD R7 ← R3, R9

- Tomasulo + 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	

- 16 vs. 12 cycles

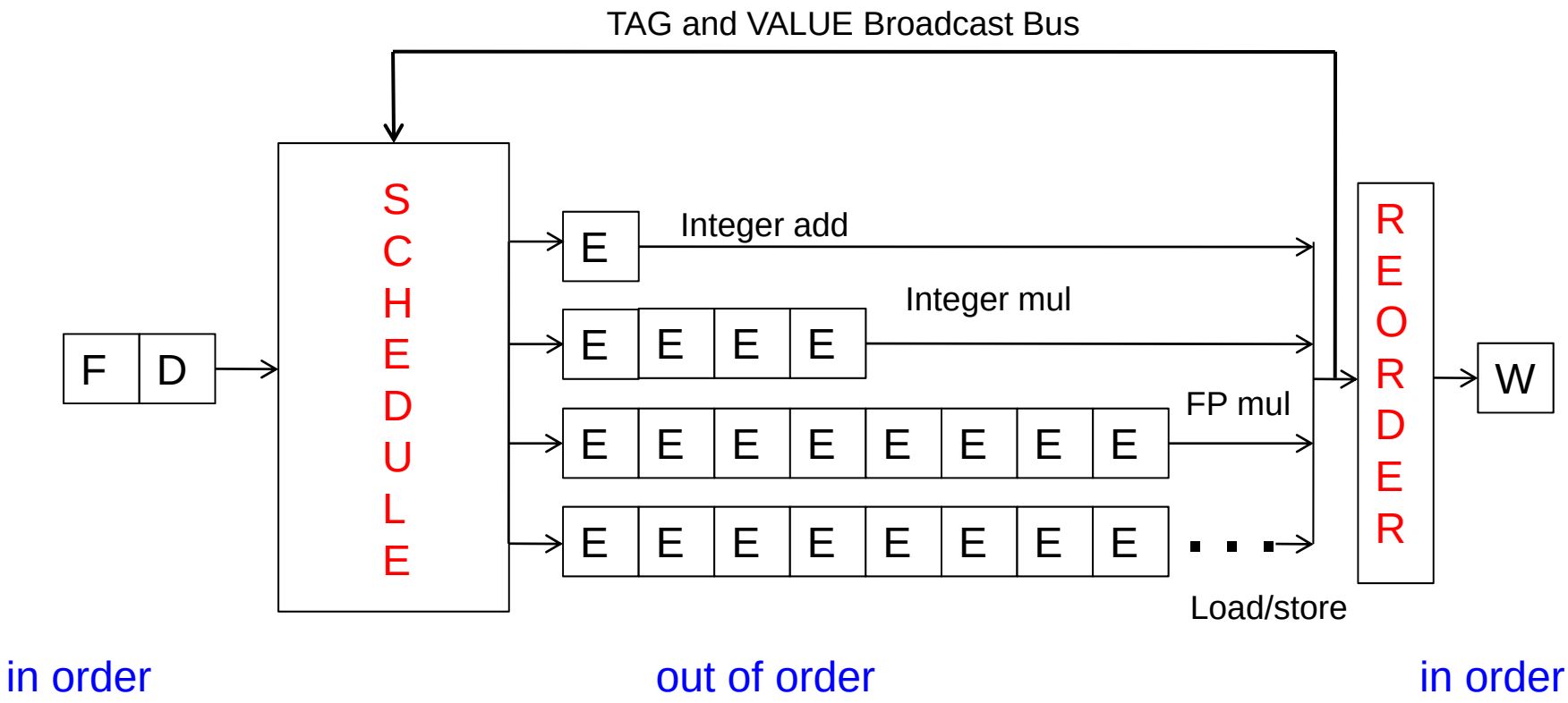
Enabling OoO Execution

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
   - 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, dispatch the instruction to functional unit (FU)
   - What if more instructions become ready than available FUs?

Tomasulo's Algorithm

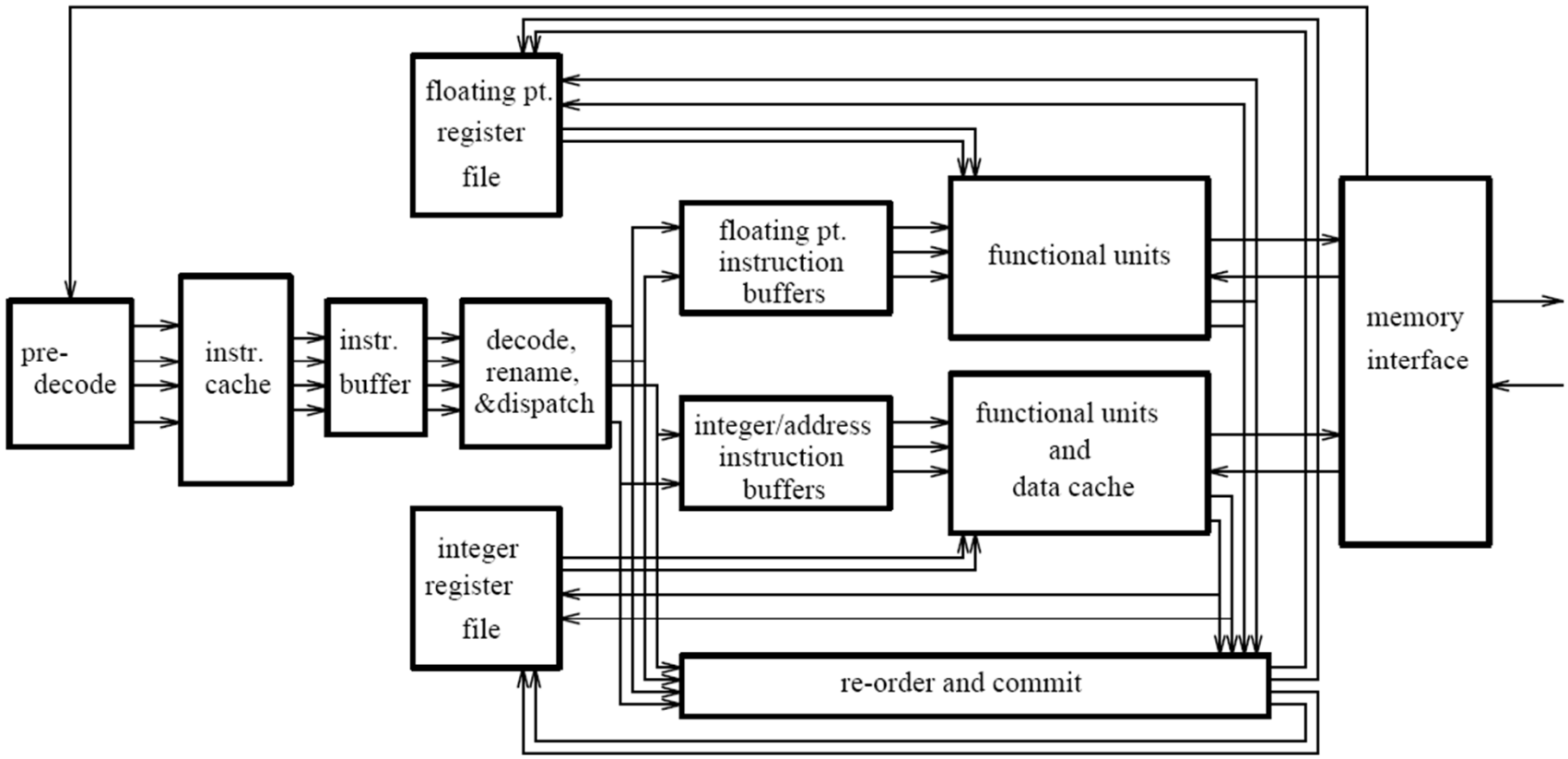
- OoO with register renaming invented by Robert Tomasulo
  - Used in IBM 360/91 Floating Point Units
  - **Read:** Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of R&D, Jan. 1967.
- Variants of it used in most high-performance processors
  - Most notably Pentium Pro, Pentium M, Intel Core(2), AMD K series,
  - Alpha 21264, MIPS R10000, IBM POWER5
- 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.

Two Humps in a Modern Pipeline



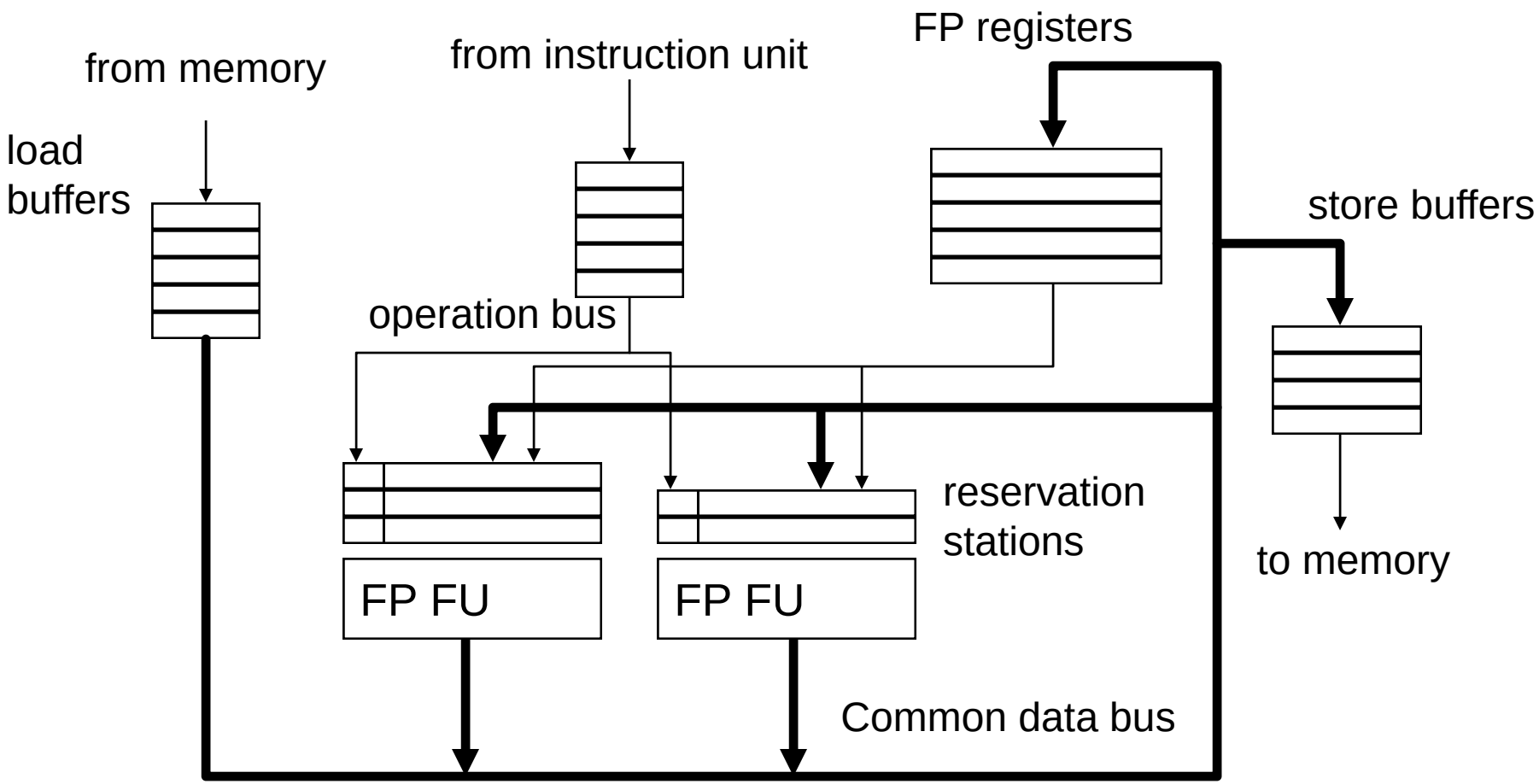
- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

General Organization of an OOO Processor



- 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!

The Remaining Slides Not Covered and Are for Your Benefit for the Next Lecture

Register Renaming and OoO Execution

- Architectural registers dynamically renamed
  - Mapped to reservation stations

	tag	value	valid?
R0	-	V0	1
R1	S2	new1V1	0
R2	-	V2	1
R3	S0	new3V3	0
R4	-	V4	1
R5	-	V5	1
R6	-	V6	1
R7	S4	V7	0
R8	-	V8	1
R9	-	V9	1

IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R3 ← R6, R8
ADD R7 ← R3, R9

IMUL S0 ← V1, V2
ADD S1 ← S0, V4
ADD S2 ← V6, V7
IMUL S3 ← V6, V8
ADD S4 ← S3, V9

	Src1 tag	Src1 value	V?	Src2 tag	Src2 value	V?	Ctl	S?
S0	-	Retired --- Entry Deallocated					nul	1
S1	S0	Retired --- Entry Deallocated					add	1
S2	-	Completed --- Wait for Retirement					add	1
S3	-	BROADCAST S3 and new3V3				1	add	1
S4	S3	new3V3	0	-	V9	1	add	

Summary of OOO Execution Concepts

- Renaming eliminates false dependencies
- Tag broadcast enables value communication between instructions → dataflow
- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
  - which piece?
    - Limited to the instruction window
  - Can we do it for the whole program? Why would we like to?
    - How can we have a large instruction window efficiently?