18-447

Computer Architecture Lecture 13: State Maintenance and Recovery

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2013, 2/15/2013

Reminder: Homework 3

- Homework 3
 - Due Feb 25
 - REP MOVS in Microprogrammed LC-3b, Pipelining, Delay Slots, Interlocking, Branch Prediction

Reminder: Lab Assignment 2 Due Today

- Lab Assignment 2
 - Due today, Feb 15
 - Single-cycle MIPS implementation in Verilog
 - All labs are individual assignments
 - No collaboration; please respect the honor code
 - Do not forget the extra credit portion!

Lab Assignment 3 Out

- Lab Assignment 3
 - Due Friday, March 1
 - Pipelined MIPS implementation in Verilog
 - All labs are individual assignments
 - No collaboration; please respect the honor code
 - Extra credit: Optimize for execution time!
 - Top assignments with lowest execution times will get extra credit.
 - And, it will be fun to optimize...

A Note on Testing Your Code

- Testing is critical in developing any system
- You are responsible for creating your own test programs and ensuring your designs work for all possible cases
- That is how real life works also...
 - Noone gives you all possible test cases, workloads, users, etc. beforehand

Feedback Sheet

- Due today (Feb 15), in class
- We would like your honest feedback on the course
- Please turn it in even if you are late...

Readings for Today

- Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans on Computers 1988 (earlier version in ISCA 1985).
- 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

Last Lecture

- Wrap up control dependence handling
 - Predicated execution
 - Wish branches
 - Multipath execution
 - Call/return prediction
 - Indirect branch prediction
 - Latency issues in branch prediction
- Interrupts vs. exceptions
- Reorder buffer as a state maintenance/recovery mechanisms

Today

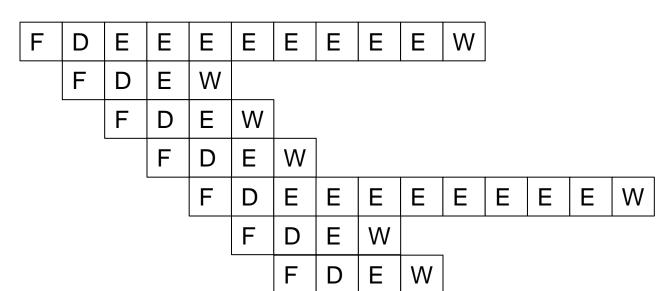
- State maintenance and recovery mechanisms
- (Maybe) Begin out-of-order execution

Pipelining and Precise Exceptions: Preserving Sequential Semantics

Review: Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
 - Integer ADD versus FP MULtiply

FMUL R4 \leftarrow R1, R2 ADD R3 \leftarrow R1, R2



- FMUL R2 \leftarrow R5, R6 ADD R4 \leftarrow R5, R6
 - What is wrong with this picture?
 - What if FMUL incurs an exception?
 - Sequential semantics of the ISA NOT preserved!

Review: Precise Exceptions/Interrupts

- The architectural state should be consistent when the exception/interrupt is ready to be handled
- 1. All previous instructions should be completely retired.
- 2. No later instruction should be retired.

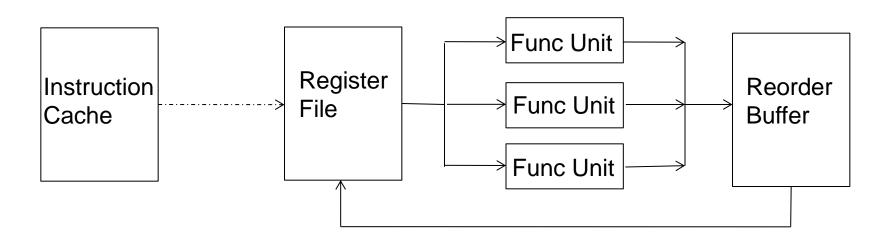
Retire = commit = finish execution and update arch. state

Review: 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.

Review: Solution I: Reorder Buffer (ROB)

- 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



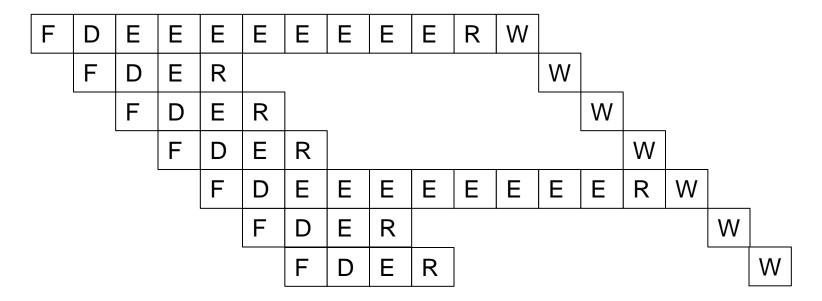
Review: What's in a ROB Entry?

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

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

Review: Reorder Buffer: Independent Operations

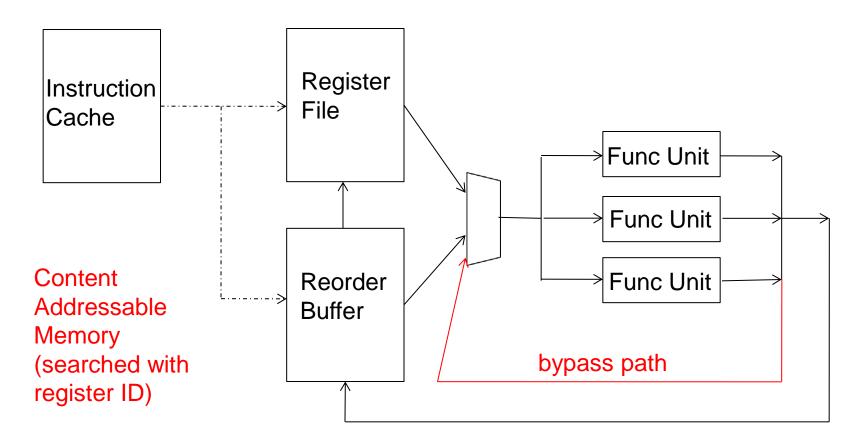
 Results first written to ROB, then to register file at commit time



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

Reorder Buffer: How to Access?

 A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)



Simplifying Reorder Buffer Access

- 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	Exc?
---	-----------	------------	-----------	-----------	-------	-------------	------

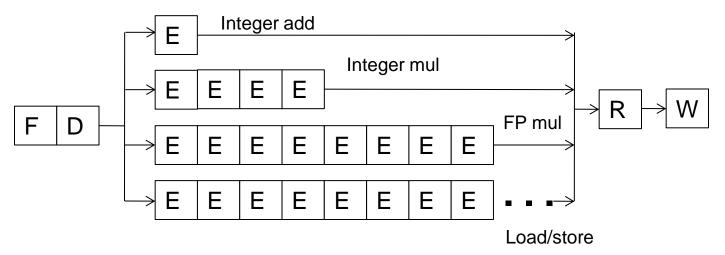
Can it be simplified further?

Aside: 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

In-Order Pipeline with Reorder Buffer

- 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



Reorder Buffer Tradeoffs

Advantages

- Conceptually simple for supporting precise exceptions
- Can eliminate false dependencies

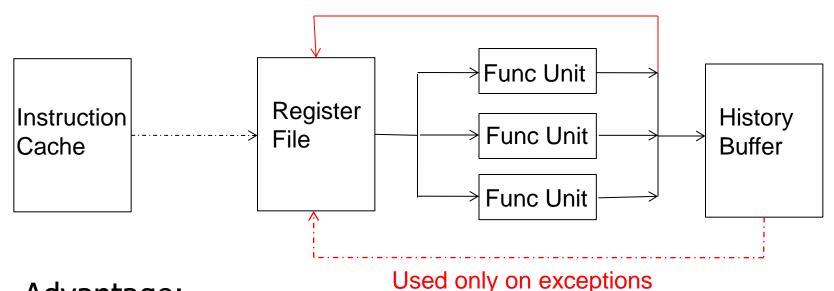
Disadvantages

- 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 disadvantages
 - History buffer
 - Future file
 - Checkpointing

Solution II: History Buffer (HB)

- 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



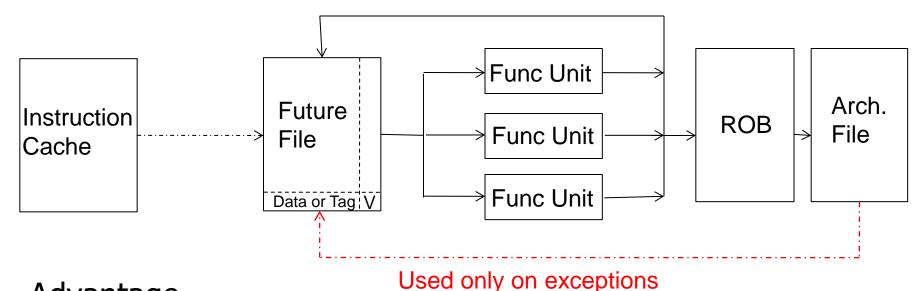
Advantage:

- 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

Solution III: Future File (FF) + 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 updates
 - 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

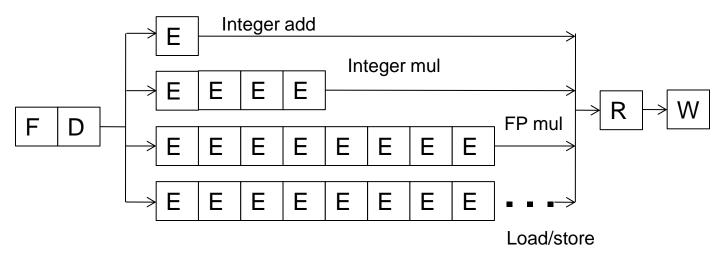
Future File



- 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

In-Order Pipeline with Future File and Reorder Buffer

- Decode (D): Access future file, 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 and future file
- Retirement/Commit (W): Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline, copy architectural file to future file, and start from exception handler
- In-order dispatch/execution, out-of-order completion, in-order retirement



Checking for and Handling 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
 - Vectored exceptions

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 are much more common
 - → need fast state recovery to minimize performance impact of mispredictions

How Fast Is State Recovery?

- Latency of state recovery affects
 - Exception service latency
 - Interrupt service latency
 - Latency to supply the correct data to instructions fetched after a branch misprediction
- Which ones above need to be fast?
- How do the three state maintenance methods fare in terms of recovery latency?
 - Reorder buffer
 - History buffer
 - Future file

Branch State Recovery Actions and Latency

Reorder Buffer

- Wait until branch is the oldest instruction in the machine
- Flush entire pipeline

History buffer

- Undo all instructions after the branch by rewinding from the tail of the history buffer until the branch & restoring old values one by one into the register file
- Flush instructions in pipeline younger than the branch

Future file

- Wait until branch is the oldest instruction in the machine
- Copy arch. reg. file to future file
- Flush entire pipeline

Can We Do Better?

- 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 at the time a branch is fetched and keep the checkpointed state updated with results of instructions older than the branch
- Hwu and Patt, "Checkpoint Repair for Out-of-order Execution Machines," ISCA 1987.

Checkpointing

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

Checkpointing

Advantages?

Disadvantages?

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)

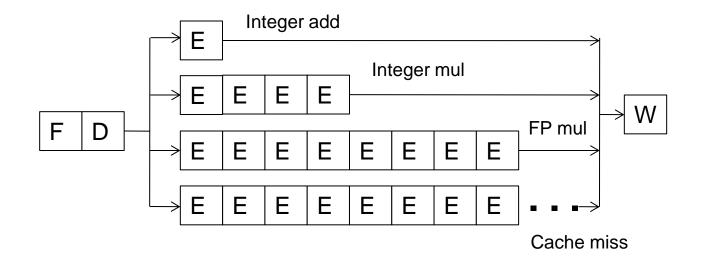
Maintaining Speculative Memory State: Stores

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

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 \leftarrow R1, R2

ADD R3 \leftarrow R3, R1

ADD R1 \leftarrow R6, R7

IMUL R5 \leftarrow R6, R8

ADD R7 \leftarrow R3, R5
```

```
LD R3 \leftarrow R1 (0)

ADD R3 \leftarrow R3, R1

ADD R1 \leftarrow R6, R7

IMUL R5 \leftarrow R6, R8

ADD R7 \leftarrow 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

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 (scheduling, or execution)
 - Solution: out-of-order dispatch (scheduling, or execution)

Out-of-order Execution (Dynamic Scheduling)

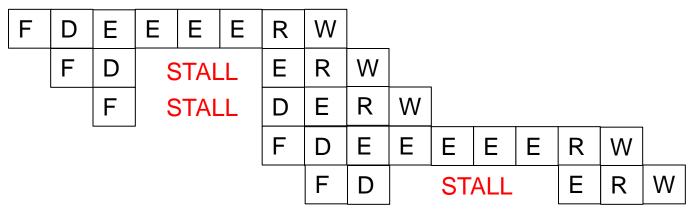
- Idea: Move the dependent instructions out of the way of independent ones
 - Rest areas for dependent instructions: Reservation stations
- Monitor the source "values" of each instruction in the resting area
- When all source "values" of an instruction are available, "fire" (i.e. dispatch) the instruction
 - 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:



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

Tomasulo + precise exceptions:

F	D	Е	Е	Е	Е	R	W				
	F	D	V	۷AI٦	<u></u>	Е	R	W			
		F	D	Е	R				W		
			F	D	Е	Е	Е	Е	R	V	
				F	D	WAIT		E	R	W	

16 vs. 12 cycles

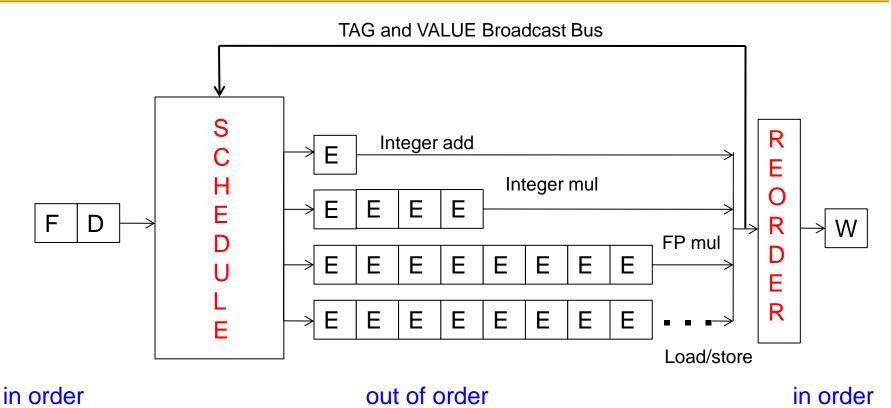
Enabling OoO Execution

- 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 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

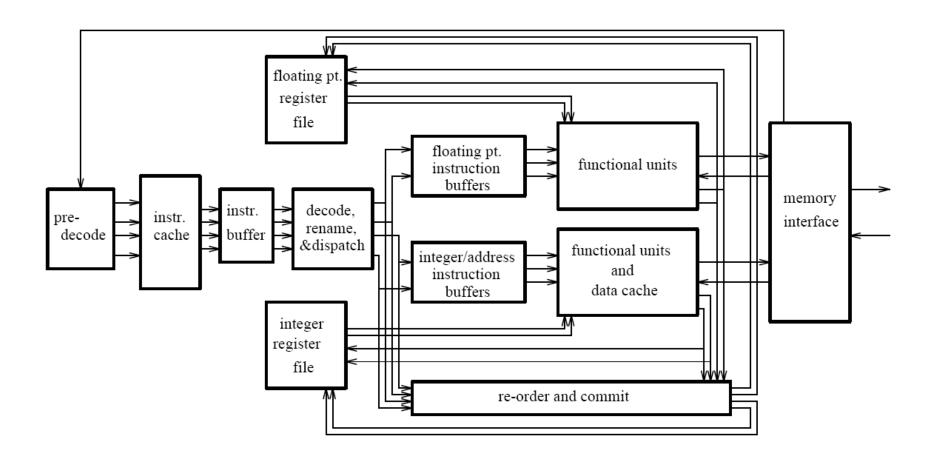
- OoO with register renaming invented by Robert Tomasulo
 - Used in IBM 360/91 Floating Point Units
 - Read: Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of R&D, Jan. 1967.
- What is the major difference today?
 - Precise exceptions: IBM 360/91 did NOT have 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



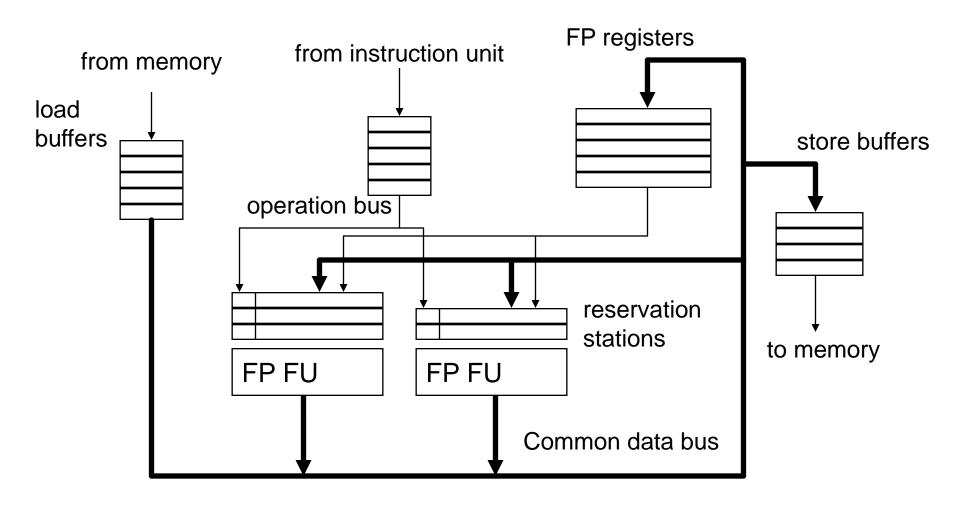
- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

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!

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
 - 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)