Computer Architecture: Out-of-Order Execution

Prof. Onur Mutlu (editted by Seth) Carnegie Mellon University

Reading 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

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 ← 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

Preventing Dispatch Stalls

- Multiple ways of doing it
- You have already seen THREE:
 - **1**.
 - **2**.
 - **3**.

Preventing Dispatch Stalls

- Multiple ways of doing it
- You have already seen THREE:
 - **1**. Fine-grained multithreading
 - 2. Value prediction
 - **3**. Compile-time instruction scheduling/reordering
- 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 + precise exceptions:



Out-of-order dispatch + precise exceptions:



Add waits on multiply producing R3

Commit happens in-order

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 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



in order

out of order

in order

- 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 \rightarrow RS entry ID
 - Architectural register ID \rightarrow 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)



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)

Exercise Continued

	Proveline structure
MUL RI, RZ, R3 ADD R3, R4- R5 ADD R2 R6- R7	FDEW
ADD R8, R9 -> R10 MUL R7, R10 -> R11	contake milinde cycles
ADD RS, RM, RS	
MUL toxes 6 oydes ADD tokes 4 oydes	
How mony cycles total who dota ,	Ferwarding? ?

Exercise Continued



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



Tamosilo's algorithm + full Anverding

20

E	Io	W	It W	Vorks Cycle 0 MUL $R3 \leftarrow R1, R2$ ADD $R5 \leftarrow R3, R4$
	v	tag	value	ADD R7 ← R2, R6
r1	1		1	$\begin{array}{c} ADD R10 \leftarrow R8, R9 \\ MUL R11 \leftarrow R7, R10 \end{array}$
r2	1		2	ADD R5 \leftarrow R5, R11
r3	1		3	
r4	1		4	
r5	1		5	v tag value v tag value v tag value v tag value
r6	1		6	
r7	1		7	
r8	1		8	
r9	1		9	
r10	1		10	
r11	1		11	
				+ *







H	lo	W	It W	0	rk	s (Cy	7 C	le	4			M A	UL R	3 € 5 €	←	R1, F R3, I	R2 R4 .
	v	tag	value									_			7	←	R2, I	R6
r1	1		1										A < M	UU R' UI R'	10 11	← ←	- R8, - R7	R9 R10
r2	1		2										A	DD R	5 •	÷	R5, I	R11
r3	0	Х	?															
r4	1		4															
r5	0	а	?		tac	u valu	e	V	tag	value		V	tag	value		V	tag	value
r6	1		6		$\frac{1}{2}$		2	1	tug		v	v 1	tug	1		v 1	lug	2 2
r7	0	b	?	b 1			: ว	1		4	× V	-			-	1		Ζ
r8	1		8	c			2	-		0	y Z				-			
r9	1		9	d											-			
r10	1		10															
r11	1		11						_						\backslash		X in	E (2)
							+	-	/						*			

H	0	ow It Works Cycle 5 MUL $R3 \leftarrow R1, R2$ ADD $R5 \leftarrow R3, R4$										R2 R4						
	V	tag	value										A	DD	R7	÷	R2, I	R6
r1	1		1										A ♪M	DD UI	R10 R11) ← 1 ←	- R8, - R7	R9 R10
r2	1		2										A	DD	R5	÷	R5, I	R11
r3	0	Х	?															
r4	1		4															
r5	0	а	?		V	tag	value	V	tag	value		V	tag	valu	e	V	tag	value
r6	1		6	0	v ∩	v	2	1			v	v 1	lug		1	1	tag	2
r7	0	b	?	a h	1	^	:	1		4	× V	-						
r8	1		8	C	1		2 0	1		0	y Z							
r9	1		9	d	•		0	-		7								
r10	0	С	?															
r11	1		11						b in E	Ξ (1)							X in	E (3)
							_	┢	/						*		/	

E	Io	W	It W	Vorks Cycle 6 MUL $R3 \leftarrow R1, R2$ ADD $R5 \leftarrow R3, R4$										R2 R4				
	v	tag	value										A	DD I	R 7	÷	R2, I	R6
r1	1		1										A M	UD H UI F	۲۱(۲۲۲) € 1 €	- R8, - R7	R9 R10
r2	1		2										> A	DD I	R 5	÷	R5, I	R11
r3	0	Х	?															
r4	1		4															
r5	0	а	?		V	taq	value	V	tag	value		V	taq	value		V	tag	value
r6	1		6	2	ں ر	v	2	1	g	1	v	1		1		1		2
r7	0	b	?	a h	1	^	:	1		4			h	-			C	2
r8	1		8		1		2	1		0	у 7	U	D	:			C	f
r9	1		9	d	1		8			9	2				_			
r10	0	С	?															
r11	0	у	?						c in E	(1)							X in	E (4)
+ *																		







E	Iow It Works Cycle 10										M A	UL R DD R	3	← ←	R1, I R3, I	R2 R4 .		
	v	tag	value										A	DD R	7	÷	R2, I	R6
r1	1		1										A M	UU R	.1(11	∢ (←	- R8, - R7	R9 R10
r2	1		2										A	DD R	5	÷	R5, I	R11
r3	1		2										\rightarrow					
r4	1		4															
r5	0	d	?		V	tag	value	V	tag	value		V	tag	value	1	V	tag	value
r6	1		6	2	v 1	v	2	1	i ag		v	1	tag	1		1	tag	2
r7	1		8	h b	' 1	Λ	2	1		т 6	× V	1	h	Q		1	C	17
r8	1		8	C	י 1		2	1		0	, Z	-	U	0			0	1 /
r9	1		9	d		2	2			7								
r10	1	¢	17		U	a	:	0	у	:								
r11	0	ý	?						A ' F								v in F	= (1)
	A in E (2) +												*		/			

An Exercise, with Precise Exceptions





- 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



in order

out of order



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

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 \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

State of RAT and RS in Cycle 7



Dataflow Graph



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 register file, scheduling window, reorder buffer

Registers versus Memory, Revisited

- So far, we considered register based value communication between instructions
- 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 need to be handled after their execution
- Corollary 3: When a load/store has its address ready, there may be younger/older loads/stores with undetermined 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

What if M[r5] == r4?

Ld	r2,r5	; r2 <- M[r5]
St	r1,r2	; M[r2] <- r1
Ld	r3,r4	; r3 <- M[r4]

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 the/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)
 - 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 dependent on all previous stores
 - + No need for recovery
 - -- Too conservative: delays independent loads unnecessarily

Option 2: Assume load 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 dependencies 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 dependencies important for performance
- Simple predictors (based on past history) can achieve most of the potential performance

Food for Thought for You

- Many other design choices
- Should reservation stations be centralized or 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?
- Exactly when does an instruction broadcast its tag?

More Food for Thought for You

- How can you implement branch prediction in an out-oforder execution machine?
 - Think about branch history register and PHT updates
 - Think about recovery from mispredictions
 - How to do this fast?
- How can you combine superscalar execution with out-oforder execution?
 - These are different concepts
 - Concurrent renaming of instructions
 - Concurrent broadcast of tags
- How can you combine superscalar + out-of-order + branch prediction?

Recommended Readings

- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro, March-April 1999.
- Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.
- Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, April 1996
- Tendler et al., "POWER4 system microarchitecture," IBM Journal of Research and Development, January 2002.