

ECE 4750 Computer Architecture, Fall 2021

T10 Advanced Processors: Out-of-Order Execution

School of Electrical and Computer Engineering
Cornell University

revision: 2021-10-26-18-26

1	Incremental Approach to Exploring OOO Execution	2
2	I3L: IO Front-End/Issue/Completion, Late Commit	3
3	I2OE: IO Front-End/Issue, OOO Completion, Early Commit	5
4	I2OL: IO Front-End/Issue, OOO Completion, Late Commit	9
5	IO2E: IO Front-End, OOO Issue/Completion, Early Commit	14
6	IO2L: IO Front-End, OOO Issue/Completion, Late Commit	20

Copyright © 2018 Christopher Batten, Christina Delimitrou. All rights reserved. This hand-out was originally prepared by Prof. Christopher Batten at Cornell University for ECE 4750 / CS 4420. It has since been updated by Prof. Christina Delimitrou in 2017-2021. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

1. Incremental Approach to Exploring OOO Execution

- Gradually work through five different microarchitectures
- For each microarchitecture
 - overall pipeline structure
 - required hardware data-structures
 - example instruction sequence executing on microarchitecture
 - handling precise exceptions
- Several simplifications
 - all designs are single issue
 - assume code sequence never includes WAW or WAR dependencies
 - only support add, addi, mul

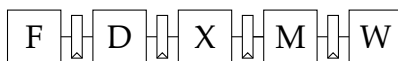
	Front-End or Fetch/Decode	Issue	Writeback or Completion	Commit	Data Structures
I3L	io	io	io	late	
I2OE	io	io	ooo	early	SB
I2OL	io	io	ooo	late	SB, ROB
IO2E	io	ooo	ooo	early	SB, IQ
IO2L	io	ooo	ooo	late	SB, IQ, ROB

```
a: mul   x1,  x2,  x3
b: addi  x11, x10, 1
c: mul   x5,  x1,  x4
d: mul   x7,  x5,  x6
e: addi  x12, x11, 1
f: addi  x13, x12, 1
g: addi  x14, x12, 2
```

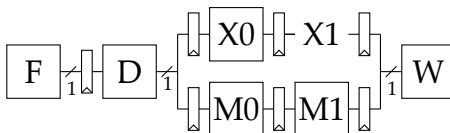
2. IO Front-End/Issue/Completion, Late Commit

	Front-End or Fetch/Decode	Issue	Writeback or Completion	Commit	Data Structures
I3L	io	io	io	late	
I2OE	io	io	ooo	early	SB
I2OL	io	io	ooo	late	SB, ROB
IO2E	io	ooo	ooo	early	SB, IQ
IO2L	io	ooo	ooo	late	SB, IQ, ROB

The following is the basic in-order single-issue pipeline.

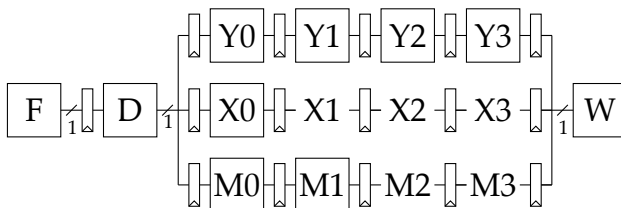


Split X/M stages into two functional units. Still single issue, so not strictly necessary but a nice incremental design step.

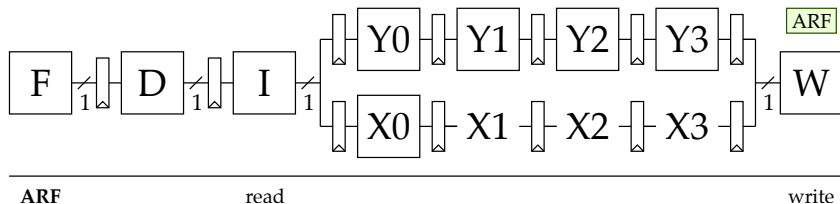


What if we want to incorporate a four-cycle pipelined integer multiplier?

Key Idea: Extend all pipelines to equal length.



Canonical I3L Pipeline



- To avoid increasing CPI, need full bypassing which can be expensive
- Add new issue stage which
 - reads architectural register file
 - performs hazard checking and includes bypass muxing
 - “issues” instruction to appropriate functional unit
- Include just X-pipe and Y-pipe since we are only focusing on add, addi, and mul instructions

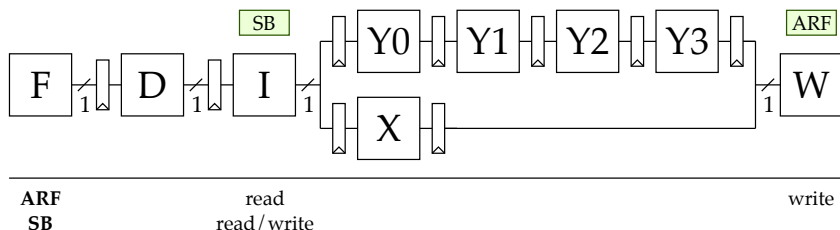
Example Execution Diagrams

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a: mul x1, x2, x3																				
b: addi x11, x10, 1																				
c: mul x5, x1, x4																				
d: mul x7, x5, x6																				
e: addi x12, x11, 1																				
f: addi x13, x12, 1																				
g: addi x14, x12, 2																				

3. IO Front-End/Issue, OOO Completion, Early Commit

	Front-End or Fetch/Decode	Issue	Writeback or Completion	Commit	Data Structures
I3L	io	io	io	late	
I2OE	io	io	ooo	early	SB
I2OL	io	io	ooo	late	SB, ROB
IO2E	io	ooo	ooo	early	SB, IQ
IO2L	io	ooo	ooo	late	SB, IQ, ROB

Canonical I2OE Pipeline



- Remove “dummy” pipeline stages
- Fewer bypass paths, significantly reduces hardware complexity
 - I3L has six bypass paths
 - I2OE has three bypass paths
 - Bypass from end of Y3, end of X, and W to end of I
- Scoreboard is used to centralize structural/data hazard detection
- WAW hazards are possible, which we ignore in this topic
- WAR hazards are not possible
- **NOTE: Fewer stages does not necessarily mean better performance!**

Data Structure: Scoreboard

	4		3		2		1		0	
	v	rdest	v	rdest	v	rdest	v	rdest	v	rdest
X	1	1	1	1	1	1	1	r1		
Y			1	r2	1	r3				

	DA					
	P	FU	4	3	2	1 0
r1	1	X				1
r2	1	Y		1		
r3	1	Y			1	
⋮						
r31						

- Indexed by functional unit
 - **V**: valid bit
 - **rdest**: destination reg specifier
 - Entries shift to right every cycle
- Structural hazards: add and addi check col 2 valid bit to ensure no structural hazard on WB port
- RAW hazards: I stage compares current instruction source reg specifiers with every valid entry in SB
 - match in col 2–4 = stall I
 - match in col 0–1 = bypass into I
 - no match = read ARF
- Large number of comparisons make accessing SB expensive
- Indexed by reg specifier
 - **P**: pending bit
 - **FU**: functional unit
 - **WA**: when available?
 - WA bits shift to right every cycle
- Structural hazards: add and addi check no bits are set in col 2 to ensure no structural hazard on WB port
- I stage checks pending bit for each source register specifier
 - pending bit set = check WA to see if stall or bypass (FU says where to bypass from)
 - pending bit clear = read ARF
- Can use SB to stall to prevent WAW hazards

Example Execution Diagrams

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

a:mul x1, x2, x3																			
b:addi x11, x10, 1																			
c:mul x5, x1, x4																			
d:mul x7, x5, x6																			
e:addi x12, x11, 1																			
f:addi x13, x12, 1																			
g:addi x14, x12, 2																			

WA Entry

cycle	D	I	x1	x5	x7	x11	x12	x13	x14
0									
1	a								
2	b	a							
3	c	b	10000						
4			01000			00010			
5			00100			00001			
6	d	c	00010						
7			00001	10000					
8				01000					
9				00100					
10	e	d		00010					
11	f	e		00001	10000				
12	g	f			01000		00010		
13					00100		00001	00010	
14		g			00010			00001	
15					00001				00010

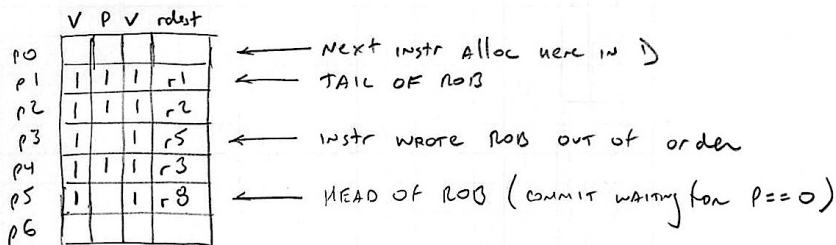
Handling Precise Exceptions

Early commit requires the commit point to be in the decode stage.
What if instruction c causes an exception?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a:mul x1, x2, x3																				
b:addi x11, x10, 1																				
c:mul x5, x1, x4																				
d:mul x7, x5, x6																				
e:addi x12, x11, 1																				
f:addi x13, x12, 1																				
g:addi x14, x12, 2																				

Not usually possible to detect all exceptions in the front-end, which motivates our interest in supporting late commit at the end of the pipeline.

Data Structure: Reorder Buffer



- ROB fields
 - V: valid bit (is this entry valid?)
 - P: pending bit (instruction in flight targeting this entry)
 - V: valid bit (is the dest reg specifier valid?)
 - rdst: destination reg specifier
- ROB managed like a queue, implemented with circular buffer
 - new instructions allocated ROB entries at tail
 - instructions update pending bit out-of-order
 - commit stage waits for pending bit of head to be clear

Example Execution Diagrams

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

a:mul x1, x2, x3																			
b:addi x11, x10, 1																			
c:mul x5, x1, x4																			
d:mul x7, x5, x6																			
e:addi x12, x11, 1																			
f:addi x13, x12, 1																			
g:addi x14, x12, 2																			

Physical Register File

x1	
x2	1
x3	2
x4	3
x5	
x6	4
x7	
x8	
x9	
x10	21
x11	
x12	
x13	
x14	
...	...
x31	

Architectural Register File

x1	
x2	1
x3	2
x4	3
x5	
x6	4
x7	
x8	
x9	
x10	21
x11	
x12	
x13	
x14	
...	...
x31	

Reorder Buffer

	p	v	rdest
p0			
p1			
p2			
p3			
p4			
p5			
p6			

We can use a table to compactly illustrate how the ROB works.

cycle	D	I	ROB Entry			
			0	1	2	3
0						
1	a					
2	b	a	x1*			
3	c	b		x11		
4					x5*	
5						
6	d	c		x11		
7						x7*
8			x1			
9				•		
10	e	d				
11	f	e	x12*			
12	g	f		x13*	x5	
13					x14*	
14		g	x12			
15				x13		
16						x7
17			•		x14	
18				•		
19					•	

Handling Precise Exceptions

Late commit means exceptions are handled in the C stage at the end of the pipeline. What if instruction a causes an exception?

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

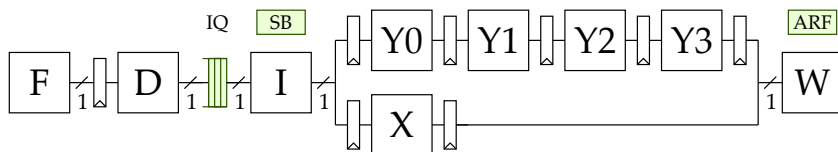
a:mul x1, x2, x3																			
b:addi x11, x10, 1																			
c:mul x5, x1, x4																			
d:mul x7, x5, x6																			
e:addi x12, x11, 1																			
f:addi x13, x12, 1																			
g:addi x14, x12, 2																			

Need to copy values from ARF to PRF on an exception before redirecting the front of the pipeline to the exception handler. This copy may take multiple cycles. Also possible to include additional bits in I stage to indicate whether the most recent version of every given architectural register is in the ARF or PRF.

5. IO Front-End, OOO Issue/Completion, Early Commit

	Front-End or Fetch/Decode	Issue	Writeback or Completion	Commit	Data Structures
I3L	io	io	io	late	
I2OE	io	io	ooo	early	SB
I2OL	io	io	ooo	late	SB, ROB
IO2E	io	ooo	ooo	early	SB, IQ
IO2L	io	ooo	ooo	late	SB, IQ, ROB

Canonical IO2E Pipeline



ARF		read		write
SB		read/write		
IQ	alloc	read/dealloc		write

- Still use scoreboard to centralize structural/data hazard detection
- Add issue queue (IQ) between D and I stages
 - allocated in-order in D stage
 - updated out-of-order in W stage
 - deallocated out-of-order in I stage
- Do not necessarily want to wait for W stage to update IQ; we will need to assume *aggressive bypassing* which requires combinational communication between last stage of functional unit and I stage
- WAW hazards are possible, which we ignore in this topic
- WAR hazards are possible, which we ignore in this topic

Data Structure: Issue Queue

V	op	imm	V	rdest	V	P	rsrc0	V	P	rsrc1	
1	ADD		1	r12	1		r11	1	1	r10	← next instr alloc new w b
1	MUL		1	r7	1		r1	1		r2	← waiting on r10
1	ADD	27	1	r5	1	1	r6				← ready to issue
1	MUL		1	r13	1	1	r14	1	1	r15	

- **IQ fields**
 - **V**: valid bit (is this entry valid?)
 - **op**: instruction opcode
 - **imm** immediate value
 - **V**: valid bit (is the dest/src reg specifier valid?)
 - **P**: pending bit (is the src data ready?)
 - **rdest/rsrc**: destination/source reg specifiers
- **IQ managed like a queue, implemented with circular buffer**
 - new instructions allocated IQ entries at tail
 - instructions leave IQ out-of-order when ready
- **WakeUp Logic**: An instruction needs to update pending bits of dependent instructions when that instruction is in W stage (actually need to do this earlier to enable aggressive bypassing)
- **Select Logic**: Determine which instructions are ready to be issued, and then select which one to actually issue. Usually issue oldest ready instruction.

```
inst_ready = ( !val_src0 || !p_src0 )
             && ( !val_src1 || !p_src1 )
             && no structural hazards
```

Example Execution Diagrams

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

a:mul x1, x2, x3																			
b:addi x11, x10, 1																			
c:mul x5, x1, x4																			
d:mul x7, x5, x6																			
e:addi x12, x11, 1																			
f:addi x13, x12, 1																			
g:addi x14, x12, 2																			

Architectural Register File		Issue Queue	op	imm	v	rdest	v	p	rsrc0	v	p	rsrc1
x1												
x2	1											
x3	2											
x4	3											
x5												
x6	4											
x7												
x8												
x9												
x10	21											
x11												
x12												
x13												
x14												
...	...											
x31												

We can use a table to compactly illustrate how the IQ works.

cycle	D	I	IQ Entry		
			0	1	2
0					
1	a				
2	b	a	x1/x2/x3		
3	c	b	x11/x10		
4	d		x5/x1*/x4		
5	e			x7/x5*/x6	
6	f	c	•		x12/x11
7	g	e	x13/x12		•
8		f	•		x14/x12
9					
10		d		•	
11		g			•
12					
13					
14					
15					
16					
17					
18					
19					

Handling Precise Exceptions

Early commit requires the commit point to be in the decode stage.
What if instruction e causes an exception?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a:mul x1, x2, x3																				
b:addi x11, x10, 1																				
c:mul x5, x1, x4																				
d:mul x7, x5, x6																				
e:addi x12, x11, 1																				
f:addi x13, x12, 1																				
g:addi x14, x12, 2																				

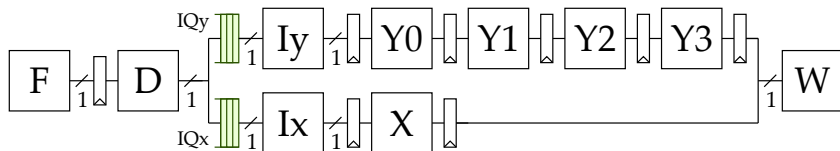
Performance Benefit of OOO Execution

Does IO2E improve performance compared to IOOE? Let's assume all instructions are in issue queue.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a:mul x1, x2, x3																				
b:addi x11, x10, 1																				
c:mul x5, x1, x4																				
d:mul x7, x5, x6																				
e:addi x12, x11, 1																				
f:addi x13, x12, 1																				
g:addi x14, x12, 2																				

Centralized vs. Distributed IQs

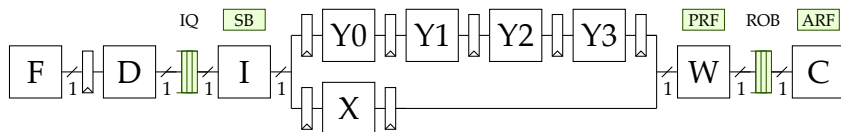
IQs can either be centralized or distributed across functional units. Distributed IQs are sometimes called **reservation stations**. This can naturally enable superscalar execution.



6. IO Front-End, OOO Issue/Completion, Late Commit

	Front-End or Fetch/Decode	Issue	Writeback or Completion	Commit	Data Structures
I3L	io	io	io	late	
I2OE	io	io	ooo	early	SB
I2OL	io	io	ooo	late	SB, ROB
IO2E	io	ooo	ooo	early	SB, IQ
IO2L	io	ooo	ooo	late	SB, IQ, ROB

Canonical IO2L Pipeline



ARF		read			
PRF		read			write
SB		read/write			read
IQ	alloc	read/dealloc			
ROB	alloc			write	read/dealloc

- Use scoreboard to centralize structural/ data hazard detection
- Use IQ to enable out-of-order issue, ROB to enable late commit
- Overall organization:
 - In-order fetc/decode (front-end of pipeline)
 - Out-of-order issue/completion (middle of pipeline)
 - In-order commit (back-end of pipeline)
- WAW hazards are possible, which we ignore in this topic
- WAR hazards are possible, which we ignore in this topic

Example Execution Diagrams

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a:mul x1, x2, x3																				
b:addi x11, x10, 1																				
c:mul x5, x1, x4																				
d:mul x7, x5, x6																				
e:addi x12, x11, 1																				
f:addi x13, x12, 1																				
g:addi x14, x12, 2																				

Handling Precise Exceptions

Late commit means exceptions are handled in the C stage at the end of the pipeline. What if instruction a causes an exception?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a:mul x1, x2, x3																				
b:addi x11, x10, 1																				
c:mul x5, x1, x4																				
d:mul x7, x5, x6																				
e:addi x12, x11, 1																				
f:addi x13, x12, 1																				
g:addi x14, x12, 2																				

Out-of-Order Dual-Issue Processor

Assume we can fetch, decode, issue, writeback, and commit two instructions per cycle.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

a:mul x1, x2, x3																			
b:addi x11, x10, 1																			
c:mul x5, x1, x4																			
d:mul x7, x5, x6																			
e:addi x12, x11, 1																			
f:addi x13, x12, 1																			
g:addi x14, x12, 2																			