ECE 4750 Computer Architecture, Fall 2024

Topic 7: Advanced Processors Out-of-Order Execution

School of Electrical and Computer Engineering
Cornell University

revision: 2024-11-25-13-49

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`

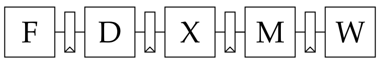
	Front-End or **Fetch/Decode**	**Issue**	**Writeback** or **Completion**	**Commit**	**Data Structures**
I3L	io	io	io	late	
I2OE	io	io	ooo	early	SB
I2OL	io	io	ooo	late	SB, ROB
IO2E	io	ooo	ooo	early	SB, IQ
IO2L	io	ooo	ooo	late	SB, IQ, ROB

```
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
```

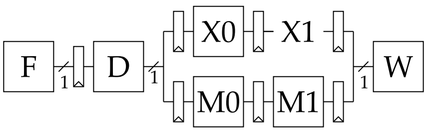
2. IO Front-End/Issue/Completion, Late Commit

	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

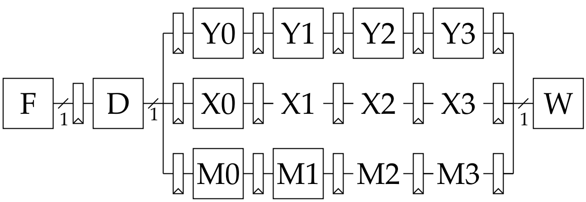
The following is the basic in-order single-issue pipeline. T07



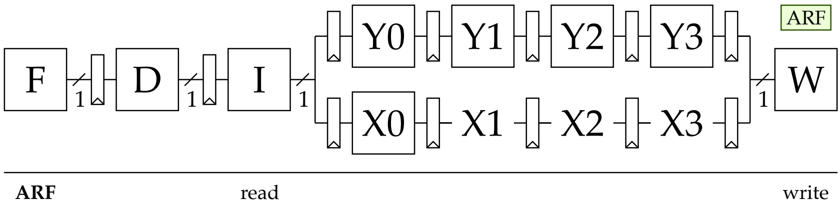
Split X/M stages into two functional units. Still single issue, so not strictly necessary but a nice incremental design step.



What if we want to incorporate a four-cycle pipelined integer multiplier?
Key Idea: Extend all pipelines to equal length.



Cannonical 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

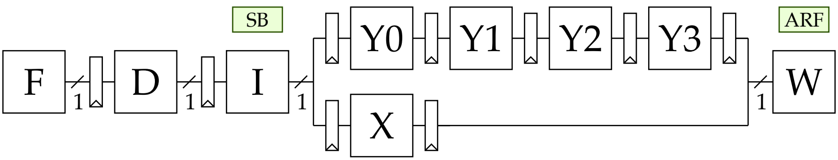
Example Execution Diagrams

	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

Cannonical I2OE Pipeline



ARF		read	write
SB		read / write	

- 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

Design 1

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

1 row per pipe

1 column for each stage (when could WB happen)

"an insn in the Y pipe will write r2 in 3 cycles"

Design 2 (more efficient!)

			WA				
	P	FU	4	3	2	1	0
r1	1	X				1	
r2	1	Y		1			
. . .	1	Y			1		
. . .							
r31							

1 row per architectural register

- 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

Since there are no memory instructions, W would be a natural place for the commit point. But since W happens out of order, a commit point in W would no longer support *precise* exceptions. (Notice below: **insn b** will reach W before **insn a** because it travels down the shorter X pipe. If **insn a** had an exception handled at W, **insn b** would have already finished! That's imprecise! That's bad!)

	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`																				

Conclusion: I2OE must have an early commit so that exceptions can be precise (at decode, for example). Unfortunately, it's not usually possible to detect all exceptions in the front-end, so we'd really like to find a way to support late commit at the end of the pipeline.

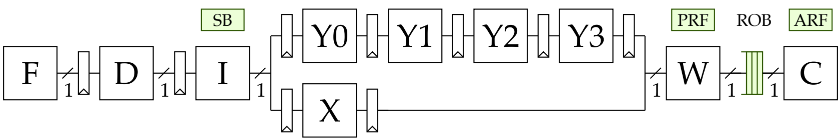
Can we do this?

Turn the page to find out....

4. IO Front-End/Issue, OOO 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

Cannonical I2OL Pipeline



	F	D	I	W	C
ARF					write
PRF			read	write	read
SB			read / write		
ROB		alloc		write	read / dealloc

- Add extra C stage for commit at end of pipeline
- Still use scoreboard to centeralize structural / data hazard detection
- Add physical regfile (PRF) and reorder buffer (ROB) between W / C
- PRF keeps uncommited results (a.k.a. future regfile, working regfile)
- Reorder buffer (ROB)
  - allocated in-order in D stage
  - updated out-of-order in W stage
  - deallocated in-order in C stage
- WAW hazards are possible, which we ignore in this topic
- WAR hazards are not possible

Data Structure: Reorder Buffer

		V	P	V	rdest	
	p0					↙ HEAD of ROB
oldest	p1	1	1	1	r1	*commit stage waits for HEAD entry to have P==0.*
	p2	1	1	1	r2	*then remove the entry and move head down*
	p3	1	0	1	r5	*no longer pending! insn wrote ROB OoO*
	p4	1	1	1	r3	
youngest	p5	1	1	1	r8	← TAIL of ROB
	p6					← *next insn to be decoded will be allocated here*
	p7					
		(this is a circular structure)				

- 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?)
  - **rdest**: 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.

			ROB Entry			
cycle	**D**	**I**	**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?

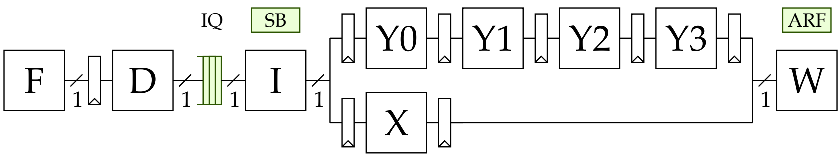
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a : `mul x1, x2, x3`																				
b : `addi x11, x10, 1`																				
c : `mul x5, x1, x4`																				
d : `mul x7, x5, x6`																				
e : `addi x12, x11, 1`																				
f : `addi x13, x12, 1`																			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

Cannonical IO2E Pipeline



	F	D	I		W
ARF			read		write
SB			read/write		
IQ		alloc	read/dealloc		write

- Still use scoreboard to centeralize 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	
HEAD →	1	addu		1	r12	1		r11	1	1	r10	*waiting on R10*
	1	mul		1	r7	1		r1	1		r2	← READY to issue (no pending srcs)
	1	addiu	27	1	r5	1	1	r6				
TAIL →	1	mul		1	r13	1	1	r14	1	1	r15	
												← *next decoded insn goes here*

- 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

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

Issue Queue

op	imm	v	rdest	v	p	rsrc0	v	p	rsrc1

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

			IQ Entry		
cycle	**D**	**I**	**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`																				

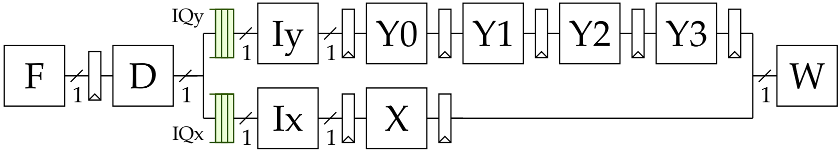
Performance Benefit of OOO Execution

Does IO2E improve performance compared to I2OE? 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

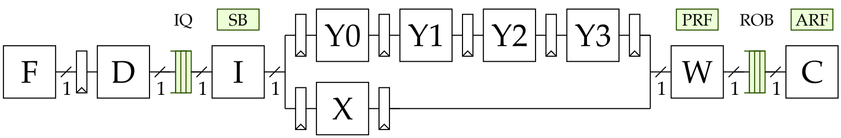
IQs can either be centralized or distributed across functional units. Distributed IQs are sometimes called reservation stations. This can naturally enable superscalar execution.



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

Cannonical IO2L Pipeline



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

- Use scoreboard to centeralize 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`																				