

ECE 4750 Computer Architecture, Fall 2024

Topic 8: Advanced Processors

Register Renaming

School of Electrical and Computer Engineering
Cornell University

revision: 2024-12-02-13-40

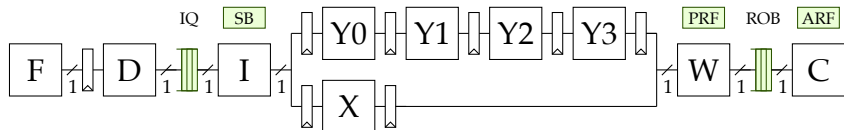
Copyright © 2024 Anne Bracy. All rights reserved. This handout was prepared by Prof. Anne Bracy at Cornell University for ECE 4750 Computer Architecture (derived from previous handouts prepared and copyrighted by Prof. Christopher Batten). Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

1. WAW and WAR Hazards

```
a: mul    x1,    x2,    x3
b: mul    x4,    x1,    x5
c: addi   x6,    x4,    1
d: addi   x4,    x7,    1
```

- RAW data hazards vs. WAW/WAR name hazards
 - RAW dependencies are “true” data dependencies because we actually pass data from the writer to the reader
 - WAW/WAR dependencies are not “true” data dependencies
 - WAW/WAR dependencies exist because of limited “names”
 - Can always avoid WAW/WAR hazards by renaming registers in software, but eventually we will run out of register names
 - **Key Idea: Provide more “physical registers” and rename architectural to physical registers in hardware**

WAW/WAR name hazards in IO2L microarchitecture



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

[illegible]

- Explore two different schemes
 - Store pointers in the IQ and ROB
 - Store values in the IQ and ROB
- For each scheme
 - overall pipeline structure
 - required hardware data-structures
 - example instruction sequence executing on microarchitecture
- Several simplifications
 - all designs are single issue
 - only support add, addi, mul

Data Structures: FL, RT, Modified ROB

Rename Table			Free List		Reorder Buffer					
	p	preg		free?	v	p	v	preg	areg	ppreg
x1	1	p0 p7	p0	0	1	1	1	p7	x8	p10
x2	0	p1	p1	0	1	1	1	p8	x4	p3
x3	0	p2	1	1	1	p9	x6	p5
x4	1	p3 p8	p8	0	0	--	--	--	--	--
x5	0	p4	p9	0						
x6	1	p5 p9	p10	1						
x7	0	p6	p11	1						
...							
x31	0	p63	p63	0						

- Free List (FL)
 - **free**: one if corresponding preg is free
 - Use priority encoder to allocate first free preg
- Rename Table (RT)
 - **p**: pending bit, is a write to this areg in flight?
 - **preg**: what preg the corresponding areg maps to
 - Entries in RT are always valid
- Modified Reorder Buffer (ROB)
 - Include three fields with pointers to PRF and ARF
 - **preg**: pointer to register in PRF that holds result value
 - **areg**: pointer to register in ARF to copy value into
 - **ppreg**: pointer to previous register in PRF for this areg

Can only free a physical register when we can guarantee no reads of that physical register are still in flight!

Example Execution Diagrams

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: mul x1, x2, x3															
b: mul x4, x1, x5															
c: addi x6, x4, 1															
d: addi x4, x7, 1															

	op	v	imm	v	dest	v	p	src0	v	p	src1
Issue Queue											

IQ entries are
physical registers

	p	preg
x1	0	p0
x2	0	p1
x3	0	p2
x4	0	p3
x5	0	p4
x6	0	p5
x7	0	p6
...		...
x31		

	free?
p0	0
p1	0
p2	0
p3	0
p4	0
p5	0
p6	0
p7	1
p8	1
p9	1
p10	1

Free List

	value
p0	
p1	1
p2	2
p3	
p4	4
p5	
p6	5
p7	...
p8	
p9	
p10	

Physical Register File

	value
x1	
x2	1
x3	2
x4	
x5	4
x6	
x7	5
...	...
x31	

Architectural Register File

	p	v	preg	areg	ppreg
Reorder Buffer					

@ Decide:

1. find pregs for src
2. alloc preg for dest
3. update RT
4. alloc ROB entry

@ eXecute:

clear pending bits in IQ

@ Writeback:

write PRF

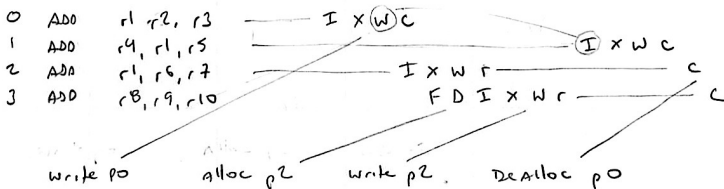
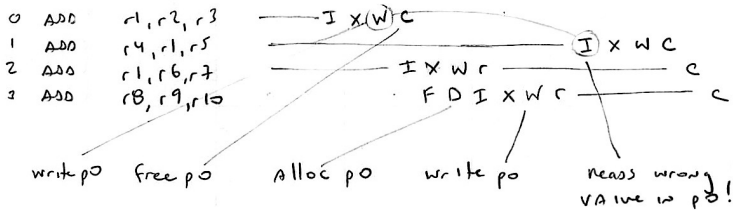
@ Commit:

copy PRF val to ARF
free *previous* preg

Cycle	D	I	W	C	Rename Table							Free List	Issue Queue			Reorder Buffer					
					x1	x2	x3	x4	x5	x6	x7		0	1	2	3	0	1	2	3	
0					p0	p1	p2	p3	p4	p5	p6	p7, p8, p9, p10									
1	a											p7, p8, p9, p10									
2	b	a			p7*							p8, p9, p10	p7/p1/p2					p7*/x1/p0			
3	c							p8*				p9, p10	p8/p7*/p4					p8*/x4/p3			
4	d									p9*		p10			p9/p8*					p9*/x6/p5	
5								p10*													p10*/x4/p8
6	b														*						
7	d	a																			
8				a	p7													p7/x1/p0			
9			d									p0									
10	c							p10				p0									p10/x4/p8
11			b									p0			*						
12			c	b								p0									
13			c									p0, p3						p8/x4/p3			
14			d									p0, p3, p5								p9/x6/p5	
15												p0, p3, p5, p8									*

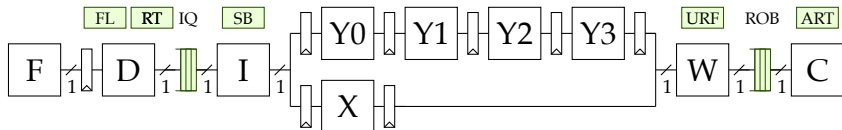
Freeing Physical Registers

ADD r1, r2, r3 ← ASSUME areg r1 maps to preg p0
 ADD r4, r1, r5
 ADD r1, r6, r7 ← NEXT write of areg r1, maps to preg p1
 ADD r8, r9, r10



if areg r_i is mapped to preg p_j , we can free p_j when the next instruction that writes r_i commits

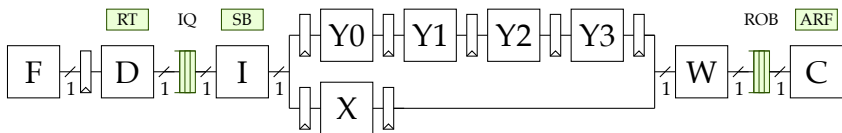
Unified Physical/Architectural Register File



ART				write
URF		read		write
SB		read/write		
IQ	alloc	read/dealloc		
ROB	alloc		write	read/dealloc
FL	read/alloc			dealloc
RT	read/write		write	

- Combine the PRF and ARF into one large unified register file (URF)
- Replace ARF with an architectural rename table (ART)
- Instead of copying *values*, C stage simply copies the preg pointer into the appropriate entry of the ART
- URF can be smaller than area for separate PRF/ARF
- Sometimes in the literature URF is just called PRF (and there is no “real” ARF, just the ART)

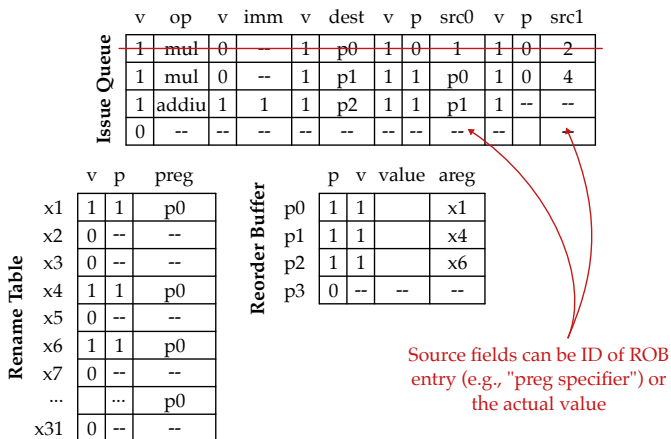
3. IO2L Value-Based Register Renaming Scheme



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

- Instead of storing future values in a separate PRF, we store these future values in the actual ROB
- No need for FL, since “physical registers” are now really ROB entry IDs and managed naturally through ROB allocation/deallocation
- Add rename table (RT) in D stage
 - RT maps architectural registers to physical registers
 - Registers renamed in D stage, entries cleared in C
 - Destination register renamed in D stage
 - Look up renamed source registers in D, and write these physical register specifiers into the IQ
- Modify scoreboard, IQ, ROB
 - Scoreboard indexed by preg instead of areg
- NOTE: Values can be bypassed or read from either the ROB or ARF
- I/X/Y/W stages only manipulate physical registers

Data Structures: RT, Modified IQ, ROB



- Rename Table (RT)
 - **v**: valid bit
 - **p**: pending bit, is a write to this areg in flight?
 - **preg**: what preg the corresponding areg maps to
 - Entries are only valid if instruction is in-flight
 - Valid bit is cleared after instruction has committed
- Modified Issue Queue (IQ)
 - **src0/src1**: when pending bit is set, source fields contain the preg specifier (i.e., ROB entry ID) that we are waiting on; when pending bit is clear, source fields contain the *values*
- Modified Reorder Buffer (ROB)
 - Replace single rdest field with two new fields
 - **value**: actual result value
 - **areg**: pointer to register in ARF to copy value into

Example Execution Diagrams

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: mul x1, x2, x3															
b: mul x4, x1, x5															
c: addi x6, x4, 1															
d: addi x4, x7, 1															

	op	v	imm	v	dest	v	p	src0	v	p	src1
Issue Queue											

	v	p	preg
Rename Table	x1		
	x2		
	x3		
	x4		
	x5		
	x6		
	x7		
	
	x31		

	p	v	value	areg
Reorder Buffer	p0			
	p1			
	p2			
	p3			

value		
Architectural Register File	x1	
	x2	1
	x3	2
	x4	
	x5	4
	x6	
	x7	5

	x31	

We can use a table to compactly illustrate how IO2L value-based register renaming works. We show the state of the RT and ROB at the beginning of every cycle.

Cycle	D	I	W	C	Rename Table							Issue Queue				Reorder Buffer			
					x1	x2	x3	x4	x5	x6	x7	0	1	2	3	0	1	2	3
0																			
1	a																		
2	b	a			p0*							p0/x2/x3				p0*/x1			
3	c							p1*				p1/p0*/x5					p1*/x4		
4	d									p2*			p2/p1*					p2*/x6	
5								p3*							p3/r7				p3*/x4
6		b										•							
7			a																
8		d		a	•											•	p0/x1		
9			d																
10		c						p3						•					p3/x4
11			b																
12			c	b													p1/x4		
13				c						•								p2/x6	
14				d				•											•
15																			