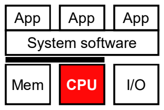
CSE 560
Computer Systems Architecture

Dynamic Scheduling

Slides originally developed by Drew Hilton (IBM)
and Milo Martin (University of Pennsylvania)

This Unit: Dynamic Scheduling



- Code scheduling
  - To reduce pipeline stalls
  - To increase ILP (insn level parallelism)

Two approaches to scheduling
- Last Unit:
  - Static scheduling by the compiler
- This Unit:
  - Dynamic scheduling by the hardware

Scheduling: Compiler or Hardware

Compiler
- + Potentially large scheduling scope (full program)
- + Simple hardware → fast clock, short pipeline, and low power
- – Low branch prediction accuracy (profiling?)
- – Little information on memory dependences (profiling?)
- – Can't dynamically respond to cache misses
- – Pain to speculate and recover from mis-speculation (h/w support?)

Hardware
- + High branch prediction accuracy
- + Dynamic information about memory dependences
- + Can respond to cache misses
- + Easy to speculate and recover from mis-speculation
- – Finite buffering resources fundamentally limit scheduling scope
- – Scheduling machinery adds pipeline stages and consumes power

Can Hardware Overcome These Limits?

- **Dynamically-scheduled processors**
  - Also called "out-of-order" processors
  - Hardware re-schedules insns…
  - …within a sliding window of VonNeumann insns
  - As with pipelining and superscalar, ISA unchanged
    - Same hardware/software interface, appearance of in-order
- Increases scheduling scope
  - Does loop unrolling transparently
  - Uses branch prediction to "unroll" branches
- Examples: Pentium Pro/II/III (3-wide), Core 2 (4-wide), Alpha 21264 (4-wide), MIPS R10000 (4-wide), Power5 (5-wide)
- Basic overview of approach

The Problem With In-Order Pipelines

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`addf f0,f1➔f2`	F	D	E+	E+	E+	W										
`mulf f2,f3➔f2`		F	d*	d*	D	E*	E*	E*	E*	E*	W					
`subf f0,f1➔f4`			F	p*	p*	D	E+	E+	E+	W						

- What's happening in cycle 4?
  - **`mulf`** stalls due to **data dependence**
    - OK, this is a fundamental problem
  - **`subf`** stalls due to **pipeline hazard**
    - Why? **`subf`** can't proceed into D because **`mulf`** is there
    - That is the only reason, and it isn't a fundamental one
  - Maintaining in-order writes to reg. file (both write **`f2`**)
- Why can't **`subf`** go into D in cycle 4 and E+ in cycle 5?

A Word About Data Hazards

- Real insn sequences pass values via registers/memory
  - Three kinds of **data dependences** (where's the fourth?)

	Read-after-write (RAW) True-dependence	Write-after-read (WAR) Anti-dependence	Write-after-write (WAW) Output-dependence

REG	`add r2,r3 ➔r1`
	`sub r1,r4 ➔r2`
	`or  r6,r3 ➔r1`
		`add r2,r3 ➔ r1`
		`sub r5,r4 ➔ r2`
		`or  r6,r3 ➔ r1`
			`add r2,r3 ➔ r1`
			`sub r1,r4 ➔ r2`
			`or  r6,r3 ➔ r1`

MEM	`st r1  ➔ [r2]`
	`ld[r2] ➔ r4`
		`ld[r1] ➔ r2`
		`st r3  ➔ [r1]`
			`st r1 ➔ [r2]`
			`st r3 ➔ [r2]`

  - Only one dependence between any two insns (RAW has priority)
- Focus on **RAW dependences**
- WAR and WAW: less common, just bad naming luck
  - Eliminated by using new register names, (can't rename memory!)

Find the RAW, WAR, and WAW dependences

```
add r1 ← r2, r3
sub r4 ← r1, r5
and r2 ← r4, r7
xor r10 ← r2, r11
or r12 ← r10, r13
mult r1 ← r10, r13
```

7

7

Find the RAW, WAR, and WAW dependences

```
add r1 ← r2, r3
sub r4 ← r1, r5
and r2 ← r4, r7
xor r10 ← r2, r11
or r12 ← r10, r13
mult r1 ← r10, r13
```

RAW dependencies:
- **r1** from **add** to **sub**
- **r2** from **and** to **xor**
- **r10** from **xor** to **or**
- **r10** from **xor** to **mult**

WAR dependencies:
- **r2** from **add** to **and**
- **r1** from **sub** to **mult**

WAW dependencies:
- **r1** from **add** to **mult**

8

8

Code Example

- Raw insns:

True Dependencies	False Dependencies
`add r2,r3➔r1`	`add r2,r3➔r1`
`sub r2,r1➔r3`	`sub r2,r1➔r3`
`mul r2,r3➔r3`	`mul r2,r3➔r3`
`div r1,4➔r1`	`div r1,4➔r1`

- "True" (real) & "False" (artificial) dependencies
- Divide insn independent of subtract and multiply insns
  - Can execute in parallel with subtract
- Many registers re-used
  - Just as in static scheduling, the register names get in the way
  - How does the hardware get around this?
- Approach: (step #1) rename registers, (step #2) schedule

9

9

Step #1: Register Renaming

- To eliminate register conflicts/hazards
- **Architected** vs. **Physical** registers – level of indirection
  - Names: **r1,r2,r3**
  - Locations: **p1,p2,p3,p4,p5,p6,p7**
  - Original mapping: **r1**→**p1**, **r2**→**p2**, **r3**→**p3**, **p4**–**p7** are available

MapTable

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p4	p2	p6
p7	p2	p6

FreeList	Original insns	Renamed insns
p4,p5,p6,p7		
p5,p6,p7	add r2,r3,r1	add p2,p3,p4
p6,p7	sub r2,r1,r3	sub p2,p4,p5
p7	mul r2,r3,r3	mul p2,p5,p6
	div r1,4,r1	div p4,4,p7

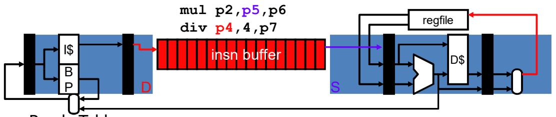
- Renaming: conceptually write each register once
  - + Removes **false** dependences
  - + Leaves **true** dependences intact!
- When to reuse a physical register? After overwriting insn done

10

10

Step #2: Dynamic Scheduling

```
add p2,p3,p4
sub p2,p4,p5
mul p2,p5,p6
div p4,4,p7
```



Ready Table

P2	P3	P4	P5	P6	P7
Yes	Yes				
Yes	Yes	Yes			
Yes	Yes	Yes	Yes		Yes
Yes	Yes	Yes	Yes	Yes	Yes

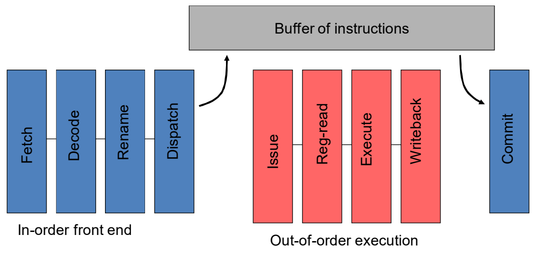
```
add p2,p3,p4
sub p2,p4,p5   and div p4,4,p7
mul p2,p5,p6
```

- Instructions fetch/decoded/renamed into *Instruction Buffer*
  - AKA "instruction window" or "instruction scheduler"
- Instructions (conceptually) check ready bits every cycle
  - Execute when ready

11

11

Out-of-order Pipeline



12

12

REGISTER RENAMING

13

13

Register Renaming Algorithm

- Data structures:
  - maptable[architectural_reg] ➔ physical_reg
  - Free list: get/put free register
- Algorithm: at decode for each instruction:

```
insn.phys_input1 = maptable[insn.arch_input1]
insn.phys_input2 = maptable[insn.arch_input2]
insn.phys_to_free = maptable[arch_output]
new_reg = get_free_phys_reg()
insn.phys_output = new_reg
maptable[arch_output] = new_reg
```

- At "commit"
  - Once all older instructions have committed, free register

```
put_free_phys_reg(insn.phys_to_free)
```

14

14

Renaming example

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Map table

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Free-list

p6
p7
p8
p9
p10

15

15

Renaming example

Original insns

```
xor r1, r2 → r3      ——→   xor  p1, p2 →
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

Map table

r1	**p1**
r2	**p2**
r3	p3
r4	p4
r5	p5

Free-list

p6
p7
p8
p9
p10

16

16

Renaming example

Original insns

```
xor r1, r2 → r3      ——→   xor  p1, p2 → p6
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

Map table

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Free-list

p6
p7
p8
p9
p10

17

17

Renaming example

Original insns

```
xor r1, r2 → r3      ——→   xor  p1, p2 → p6
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

Map table

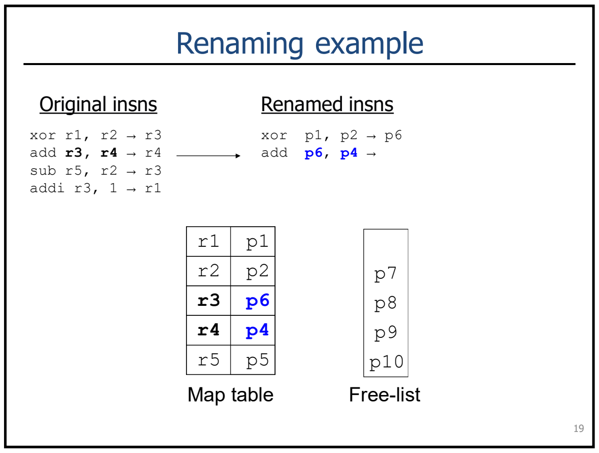
r1	p1
r2	p2
r3	**p6**
r4	p4
r5	p5

Free-list

p7
p8
p9
p10

18

18



Renaming example

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 →
```

r1	p1
r2	p2
r3	**p6**
r4	**p4**
r5	p5

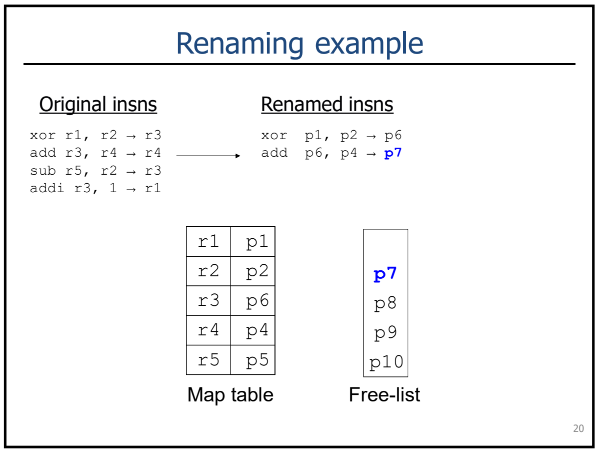
Map table

p7
p8
p9
p10

Free-list

19

19



Renaming example

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 → p7
```

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

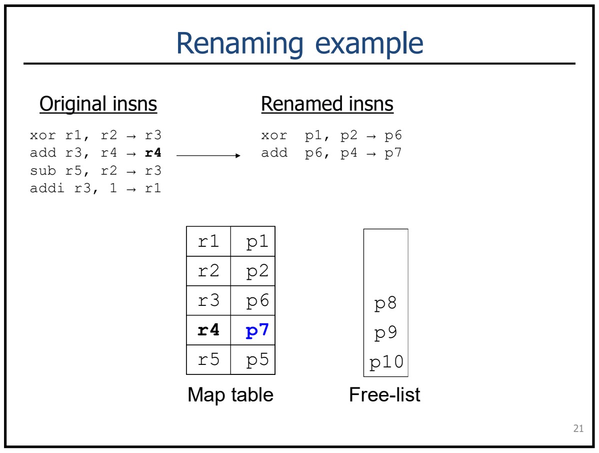
Map table

p7
p8
p9
p10

Free-list

20

20



Renaming example

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 → p7
```

r1	p1
r2	p2
r3	p6
r4	**p7**
r5	p5

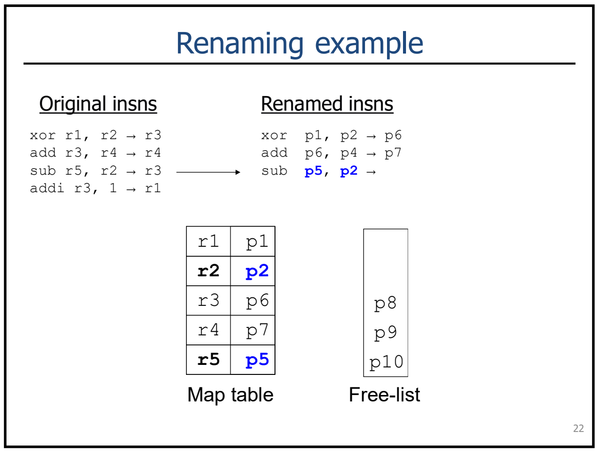
Map table

p8
p9
p10

Free-list

21

21



Renaming example

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 →
```

r1	p1
r2	**p2**
r3	p6
r4	p7
r5	**p5**

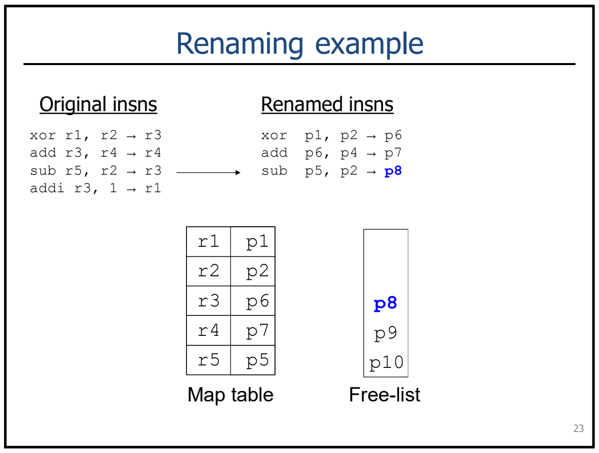
Map table

p8
p9
p10

Free-list

22

22



Renaming example

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
```

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

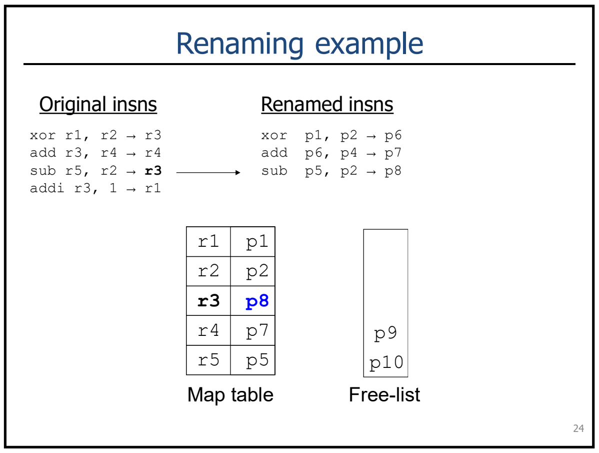
Map table

p8
p9
p10

Free-list

23

23



Renaming example

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
```

r1	p1
r2	p2
r3	**p8**
r4	p7
r5	p5

Map table

p9
p10

Free-list

24

24

Renaming example

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
addi p8, 1  →
```

Map table

r1	p1
r2	p2
r3	**p8**
r4	p7
r5	p5

Free-list

p9
p10

25

Renaming example

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
addi p8, 1  → p9
```

Map table

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p9
p10

26

Renaming example

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
addi p8, 1  → p9
```

Map table

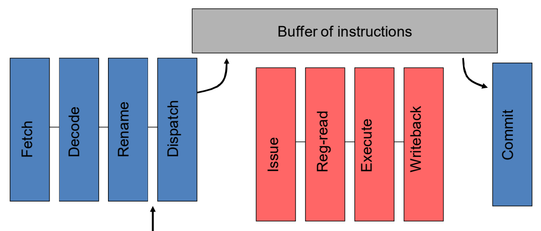
r1	**p9**
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10

27

Out-of-order Pipeline



Have unique register names
Now put into ooo execution structures

28

DYNAMIC SCHEDULING

29

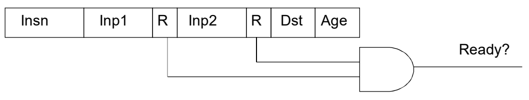
Dispatch

- Renamed instructions into ooo structures
  - Re-order buffer (ROB)
    - Holds all instructions until they commit
  - Issue Queue
    - Un-executed instructions
    - Central piece of scheduling logic
    - Content Addressable Memory (CAM) (more later)

30

Issue Queue

- Holds un-executed instructions
- Tracks ready inputs
  - Physical register names + ready bit
  - AND to tell if ready



Dispatch Steps

- Allocate IQ slot
  - Full? Stall
- Read **ready bits** of inputs
  - Table 1-bit per preg
- Clear **ready bit** of output in table
  - Instruction has not produced value yet
- Write data in IQ slot

Dispatch Example

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
addi p8, 1  → p9
```

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Age

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	y
p7	y
p8	y
p9	y

Dispatch Example

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
addi p8, 1  → p9
```

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	p6	0

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	**n**
p7	y
p8	y
p9	y

Dispatch Example

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
addi p8, 1  → p9
```

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	**n**
p8	y
p9	y

Dispatch Example

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
addi p8, 1  → p9
```

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1
sub	p5	y	p2	y	p8	2

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	**n**
p9	y

Dispatch Example

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
addi p8, 1  → p9
```

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1
sub	p5	y	p2	y	p8	2
addi	p8	n	---	y	p9	3

Ready bits

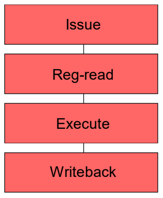
p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	n
p9	**n**

37

37

Out-of-order pipeline

- Execution (ooo) stages
- **Select** ready instructions
  - Send for execution
- **Wakeup** dependents



38

38

Dynamic Scheduling/Issue Algorithm

- Data structures:
  - Ready table[phys_reg] ➔ yes/no (part of issue queue)
- Algorithm at "schedule" stage (prior to read registers):

```
foreach instruction:
   if table[insn.phys_input1] == ready &&
       table[insn.phys_input2] == ready then
           insn is "ready"
select the oldest "ready" instruction
   table[insn.phys_output] = ready
```

39

39

Issue = Select + Wakeup

- **Select** N oldest, ready instructions
  - N=1, "xor"
  - N=2, "xor" and "sub"
  - Note: may have execution resource constraints: *i.e.,* load/store/fp

Insn	Inp1	R	Inp2	R	Dst	Age	
xor	p1	**y**	p2	**y**	p6	0	**Ready!**
add	p6	n	p4	y	p7	1	
sub	p5	**y**	p2	**y**	p8	2	**Ready!**
addi	p8	n	---	y	p9	3	

40

40

Issue = Select + Wakeup

- **Wakeup** dependent instructions
  - CAM search for Dst in inputs
  - Set ready
  - Also update ready-bit table for future instructions

Insn	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	**p6**	0
add	**p6**	**y**	p4	y	p7	1
sub	p5	y	p2	y	**p8**	2
addi	**p8**	**y**	---	y	p9	3

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	**y**
p7	n
p8	**y**
p9	n

41

41

Issue

- **Select/Wakeup** one cycle
- Dependents go back to back
  - Next cycle: add/addi are ready:

Insn	Inp1	R	Inp2	R	Dst	Age
add	p6	**y**	p4	y	p7	1
addi	p8	**y**	---	y	p9	3

42

42

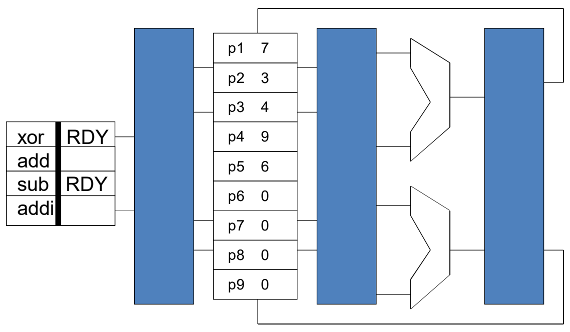
Register Read

- When do instructions read the register file?
- Option #1: after select, right before execute
  - (Not done at decode)
  - Read **physical** register (renamed)
  - Or get value via bypassing (based on physical register name)
  - This is Pentium 4, MIPS R10k, Alpha 21264 style
- Physical register file may be large
  - Multi-cycle read
- Option #2: as part of issue, keep values in Issue Queue
  - Pentium Pro, Core 2, Core i7

43

43

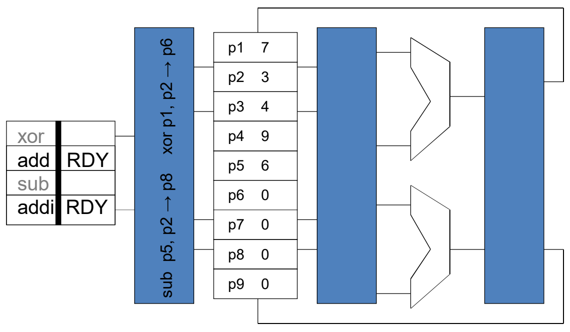
OOO execution (2-wide)



48

48

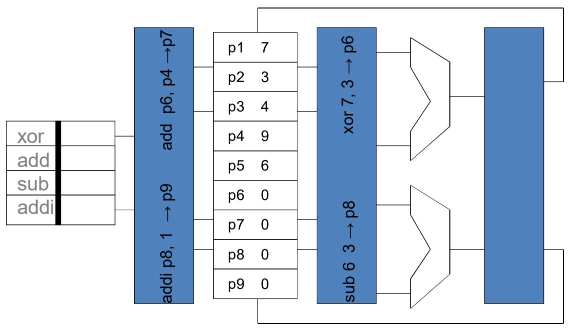
OOO execution (2-wide)



49

49

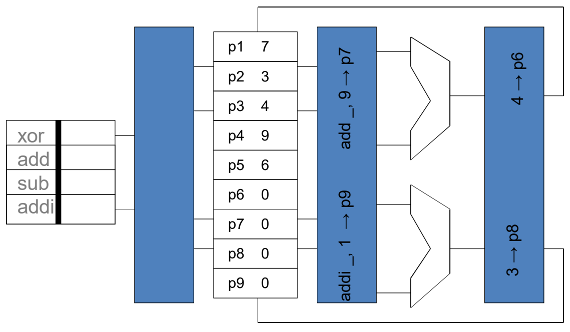
OOO execution (2-wide)



50

50

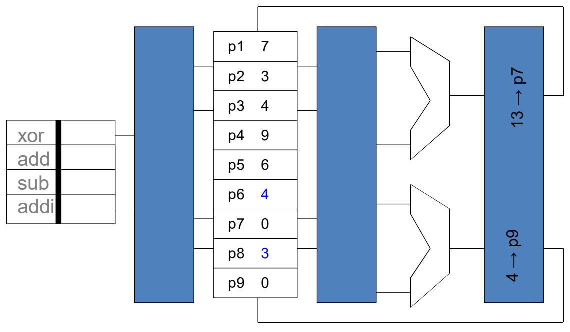
OOO execution (2-wide)



51

51

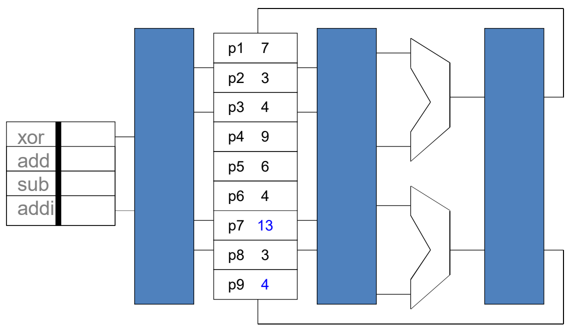
OOO execution (2-wide)



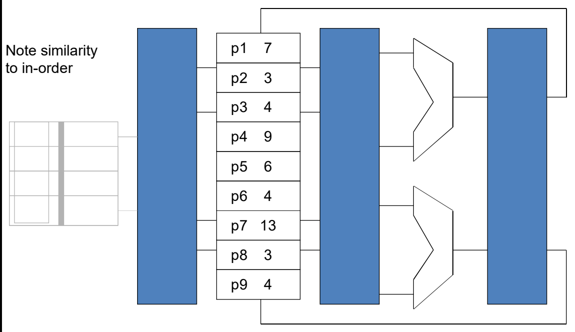
52

52

OOO execution (2-wide)



OOO execution (2-wide)



Multi-cycle operations

- Multi-cycle ops (load, fp, multiply, *etc.*)
  - Wakeup deferred a few cycles
    - Structural hazard?
- Cache misses?
  - Speculative wake-up (assume hit)
  - Cancel exec of dependents
  - Re-issue later
  - Details: complicated, not important

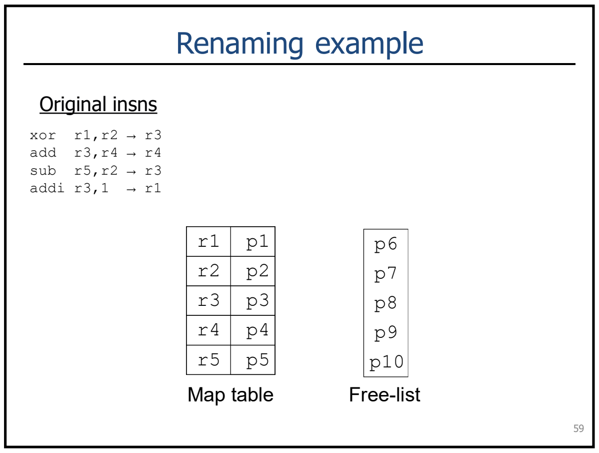
Re-order Buffer (ROB)

- All instructions in order
- Two purposes
  - Misprediction recovery
  - In-order commit
    - Maintain appearance of in-order execution
    - Freeing of physical registers

RENAMING REVISITED

Renaming revisited

- Overwritten register
  - Freed at commit
  - Restore in map table on recovery
    - Branch mis-prediction recovery
  - Also must be read at rename



Renaming example

Original insns

```
xor  r1,r2 → r3
add  r3,r4 → r4
sub  r5,r2 → r3
addi r3,1  → r1
```

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

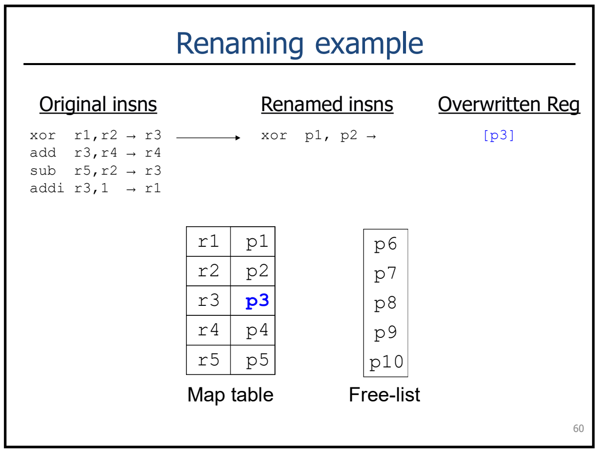
Map table

p6
p7
p8
p9
p10

Free-list

59

59



Renaming example

```
Original insns               Renamed insns     Overwritten Reg
xor  r1,r2 → r3  ─────────→  xor  p1, p2 →     [p3]
add  r3,r4 → r4
sub  r5,r2 → r3
addi r3,1  → r1
```

r1	p1
r2	p2
r3	**p3**
r4	p4
r5	p5

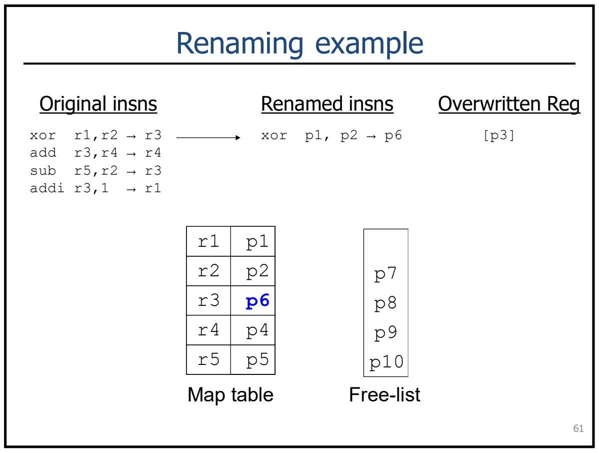
Map table

p6
p7
p8
p9
p10

Free-list

60

60



Renaming example

```
Original insns               Renamed insns        Overwritten Reg
xor  r1,r2 → r3  ─────────→  xor  p1, p2 → p6     [p3]
add  r3,r4 → r4
sub  r5,r2 → r3
addi r3,1  → r1
```

r1	p1
r2	p2
r3	**p6**
r4	p4
r5	p5

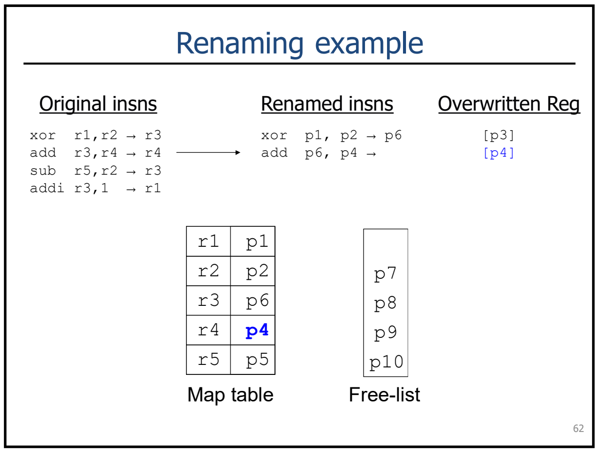
Map table

p7
p8
p9
p10

Free-list

61

61



Renaming example

```
Original insns               Renamed insns        Overwritten Reg
xor  r1,r2 → r3              xor  p1, p2 → p6     [p3]
add  r3,r4 → r4  ─────────→  add  p6, p4 →        [p4]
sub  r5,r2 → r3
addi r3,1  → r1
```

r1	p1
r2	p2
r3	p6
r4	**p4**
r5	p5

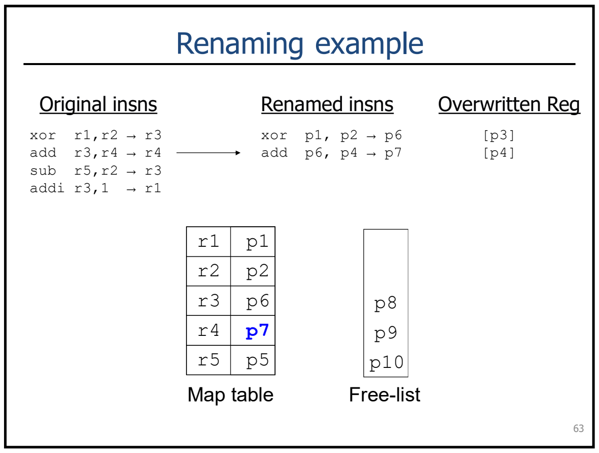
Map table

p7
p8
p9
p10

Free-list

62

62



Renaming example

```
Original insns               Renamed insns        Overwritten Reg
xor  r1,r2 → r3              xor  p1, p2 → p6     [p3]
add  r3,r4 → r4  ─────────→  add  p6, p4 → p7     [p4]
sub  r5,r2 → r3
addi r3,1  → r1
```

r1	p1
r2	p2
r3	p6
r4	**p7**
r5	p5

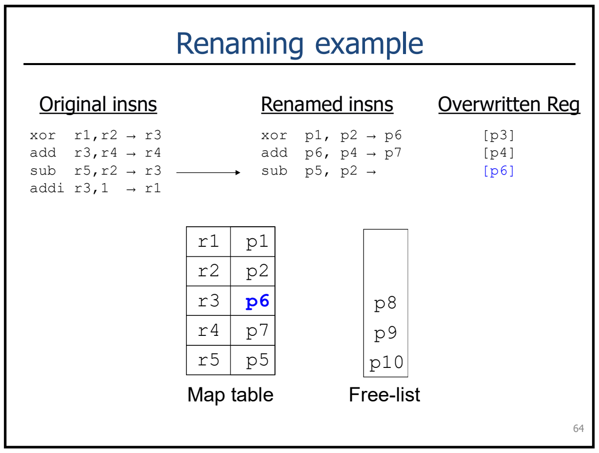
Map table

p8
p9
p10

Free-list

63

63



Renaming example

```
Original insns               Renamed insns        Overwritten Reg
xor  r1,r2 → r3              xor  p1, p2 → p6     [p3]
add  r3,r4 → r4              add  p6, p4 → p7     [p4]
sub  r5,r2 → r3  ─────────→  sub  p5, p2 →        [p6]
addi r3,1  → r1
```

r1	p1
r2	p2
r3	**p6**
r4	p7
r5	p5

Map table

p8
p9
p10

Free-list

64

64

Renaming example

```
Original insns            Renamed insns        Overwritten Reg
xor  r1,r2 → r3           xor  p1, p2 → p6     [p3]
add  r3,r4 → r4           add  p6, p4 → p7     [p4]
sub  r5,r2 → r3   ──────→ sub  p5, p2 → p8     [p6]
addi r3,1  → r1
```

Map table

r1	p1
r2	p2
r3	**p8**
r4	p7
r5	p5

Free-list

p9
p10

65

Renaming example

```
Original insns            Renamed insns        Overwritten Reg
xor  r1,r2 → r3           xor  p1, p2 → p6     [p3]
add  r3,r4 → r4           add  p6, p4 → p7     [p4]
sub  r5,r2 → r3           sub  p5, p2 → p8     [p6]
addi r3,1  → r1   ──────→ addi p8, 1 →         [p1]
```

Map table

r1	**p1**
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p9
p10

66

Renaming example

```
Original insns            Renamed insns        Overwritten Reg
xor  r1,r2 → r3           xor  p1, p2 → p6     [p3]
add  r3,r4 → r4           add  p6, p4 → p7     [p4]
sub  r5,r2 → r3           sub  p5, p2 → p8     [p6]
addi r3,1  → r1           addi p8, 1  → p9     [p1]
```

Map table

r1	**p9**
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10

67

ROB

- ROB entry holds all info for recover/commit
  - Logical register names
  - Physical register names
  - Instruction types
- Dispatch: insert at tail
  - Full? Stall
- Commit: remove from head
  - Not completed? Stall

68

Recovery

- Completely remove wrong path instructions
  - Flush from IQ
  - Remove from ROB
  - Restore map table to before misprediction
  - Free destination registers

69

Recovery example

```
Original insns            Renamed insns        Overwritten Reg
bnz r1 loop               bnz p1, loop         [  ]
xor r1, r2  → r3          xor  p1, p2 → p6     [p3]
add r3, r4 → r4           add  p6, p4 → p7     [p4]
sub r5, r2 → r3           sub  p5, p2 → p8     [p6]
addi r3, 1 → r1           addi p8, 1  → p9     [p1]
```

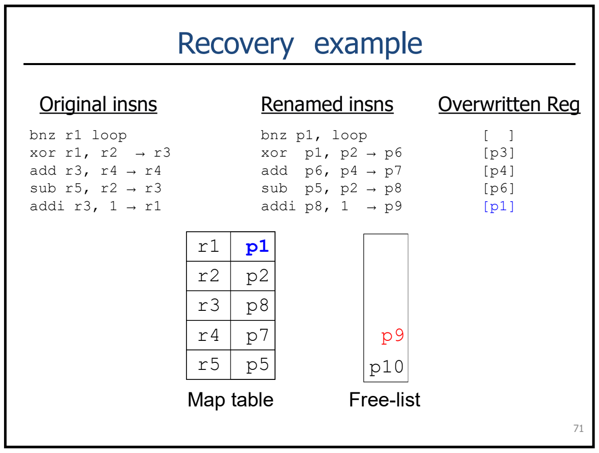
Map table

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10

70



Recovery example

Original insns | Renamed insns | Overwritten Reg

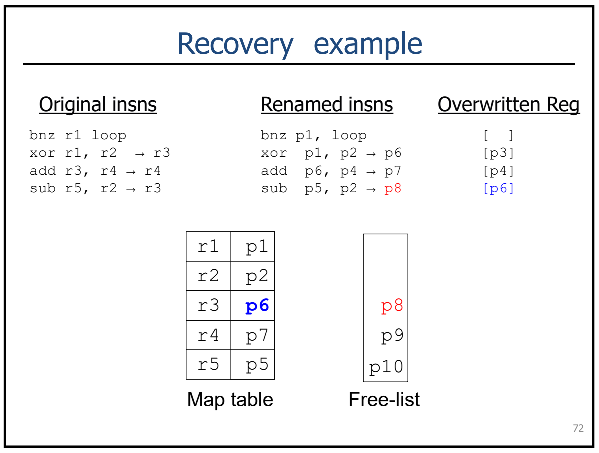
```
bnz r1 loop              bnz p1, loop       [  ]
xor r1, r2  → r3         xor  p1, p2 → p6   [p3]
add r3, r4 → r4          add  p6, p4 → p7   [p4]
sub r5, r2 → r3          sub  p5, p2 → p8   [p6]
addi r3, 1 → r1          addi p8, 1  → p9   [p1]
```

Map table	
r1	**p1**
r2	p2
r3	p8
r4	p7
r5	p5

Free-list: p9, p10

71

71



Recovery example

Original insns | Renamed insns | Overwritten Reg

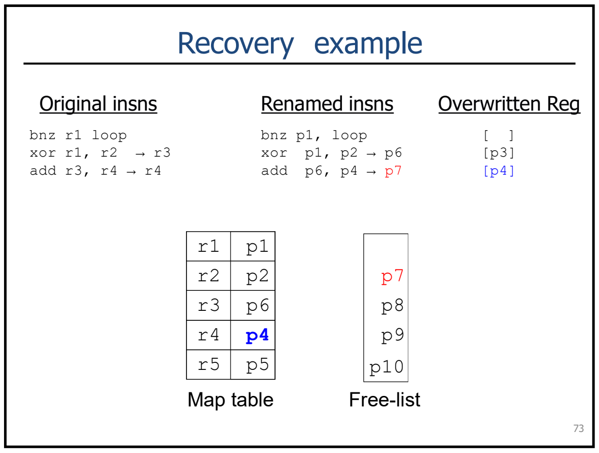
```
bnz r1 loop              bnz p1, loop       [  ]
xor r1, r2  → r3         xor  p1, p2 → p6   [p3]
add r3, r4 → r4          add  p6, p4 → p7   [p4]
sub r5, r2 → r3          sub  p5, p2 → p8   [p6]
```

Map table	
r1	p1
r2	p2
r3	**p6**
r4	p7
r5	p5

Free-list: p8, p9, p10

72

72



Recovery example

Original insns | Renamed insns | Overwritten Reg

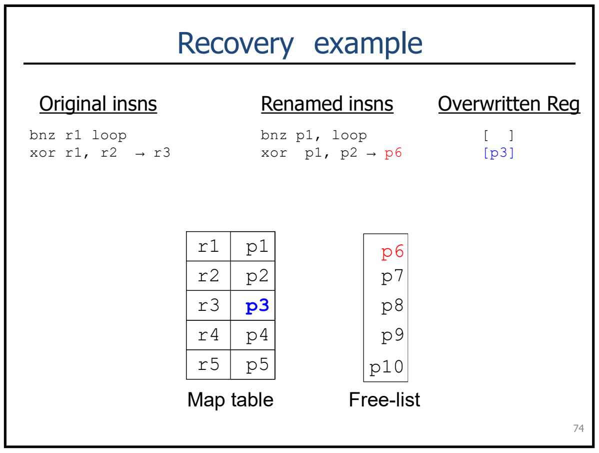
```
bnz r1 loop              bnz p1, loop       [  ]
xor r1, r2  → r3         xor  p1, p2 → p6   [p3]
add r3, r4 → r4          add  p6, p4 → p7   [p4]
```

Map table	
r1	p1
r2	p2
r3	p6
r4	**p4**
r5	p5

Free-list: p7, p8, p9, p10

73

73



Recovery example

Original insns | Renamed insns | Overwritten Reg

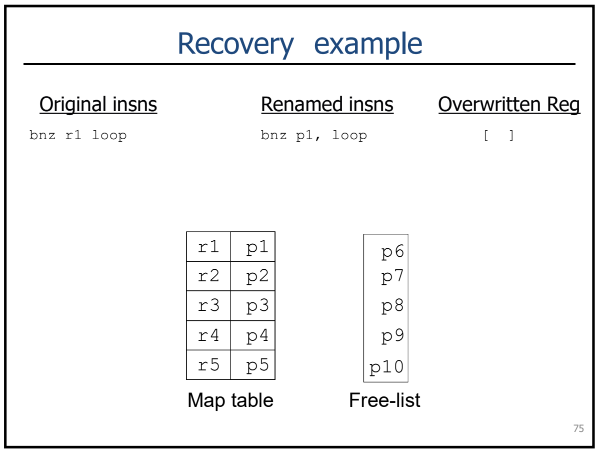
```
bnz r1 loop              bnz p1, loop       [  ]
xor r1, r2  → r3         xor  p1, p2 → p6   [p3]
```

Map table	
r1	p1
r2	p2
r3	**p3**
r4	p4
r5	p5

Free-list: p6, p7, p8, p9, p10

74

74



Recovery example

Original insns | Renamed insns | Overwritten Reg

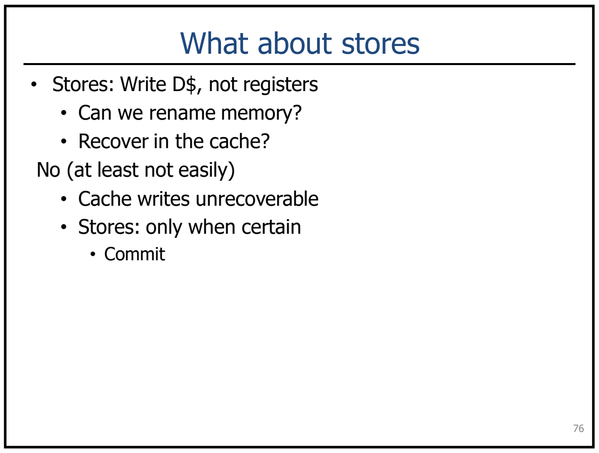
```
bnz r1 loop              bnz p1, loop       [  ]
```

Map table	
r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Free-list: p6, p7, p8, p9, p10

75

75



What about stores

- Stores: Write D$, not registers
  - Can we rename memory?
  - Recover in the cache?

No (at least not easily)
  - Cache writes unrecoverable
  - Stores: only when certain
    - Commit

76

76

Commit

Original insns	Renamed insns	Overwritten Reg
`xor r1, r2 → r3`	`xor p1, p2 → p6`	`[p3]`
`add r3, r4 → r4`	`add p6, p4 → p7`	`[p4]`
`sub r5, r2 → r3`	`sub p5, p2 → p8`	`[p6]`
`addi r3, 1 → r1`	`addi p8, 1 → p9`	`[p1]`

- At commit: instruction becomes architected state
- In-order
- Only when instructions are finished
- Free overwritten register (why?)

77

77

Freeing over-written register

Original insns	Renamed insns	Overwritten Reg
`xor r1,r2 → r3`	`xor p1, p2 → p6`	`[p3]`
`add r3,r4 → r4`	`add p6, p4 → p7`	`[p4]`
`sub r5,r2 → r3`	`sub p5, p2 → p8`	`[p6]`
`addi r3,1 → r1`	`addi p8, 1 → p9`	`[p1]`

- Before xor: **`r3`→ `p3`**
- After xor: **`r3`→ `p6`**
  - Insns older than xor reads p3
  - Insns younger than xor read p6 (until next r3-writing instruction)
- At commit of xor, no older instructions exist
  - No one else needs p3 → free it!

78

78

Commit Example

Original insns	Renamed insns	Overwritten Reg
`xor r1,r2 → r3`	`xor p1, p2 → p6`	`[p3]`
`add r3,r4 → r4`	`add p6, p4 → p7`	`[p4]`
`sub r5,r2 → r3`	`sub p5, p2 → p8`	`[p6]`
`addi r3,1 → r1`	`addi p8, 1 → p9`	`[p1]`

Map table

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10

79

79

Commit Example

Original insns	Renamed insns	Overwritten Reg
`xor r1,r2 → r3`	`xor p1, p2 → p6`	`[p3]`
`add r3,r4 → r4`	`add p6, p4 → p7`	`[p4]`
`sub r5,r2 → r3`	`sub p5, p2 → p8`	`[p6]`
`addi r3,1 → r1`	`addi p8, 1 → p9`	`[p1]`

Map table

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10
p3

80

80

Commit Example

Original insns	Renamed insns	Overwritten Reg
`add r3,r4 → r4`	`add p6, p4 → p7`	`[p4]`
`sub r5,r2 → r3`	`sub p5, p2 → p8`	`[p6]`
`addi r3,1 → r1`	`addi p8, 1 → p9`	`[p1]`

Map table

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10
p3
p4

81

81

Commit Example

Original insns	Renamed insns	Overwritten Reg
`sub r5,r2 → r3`	`sub p5, p2 → p8`	`[p6]`
`addi r3,1 → r1`	`addi p8, 1 → p9`	`[p1]`

Map table

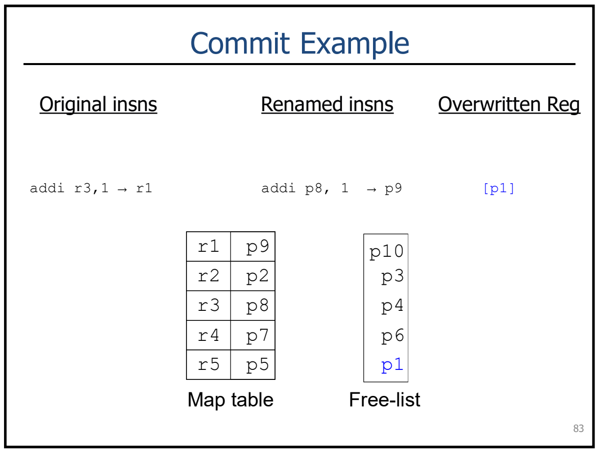
r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10
p3
p4
p6

82

82



Commit Example

Original insns	Renamed insns	Overwritten Reg
`addi r3,1 → r1`	`addi p8, 1 → p9`	[p1]

Map table

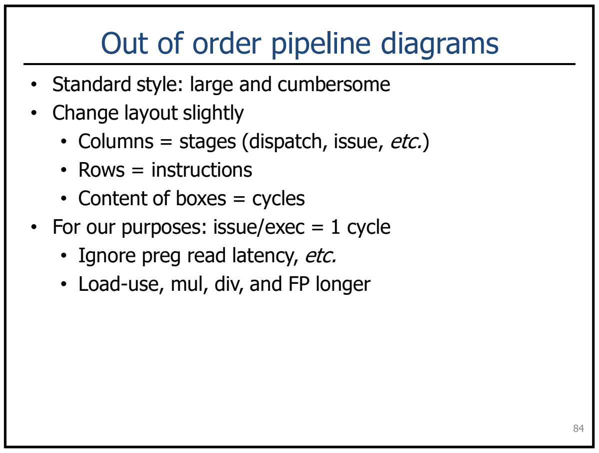
r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10
p3
p4
p6
p1

83

83

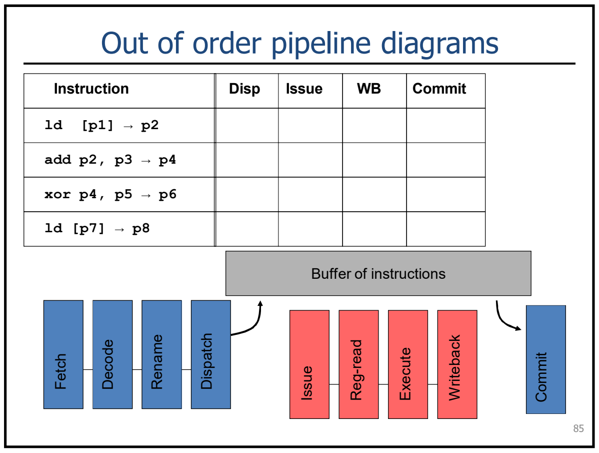


Out of order pipeline diagrams

- Standard style: large and cumbersome
- Change layout slightly
  - Columns = stages (dispatch, issue, *etc.*)
  - Rows = instructions
  - Content of boxes = cycles
- For our purposes: issue/exec = 1 cycle
  - Ignore preg read latency, *etc.*
  - Load-use, mul, div, and FP longer

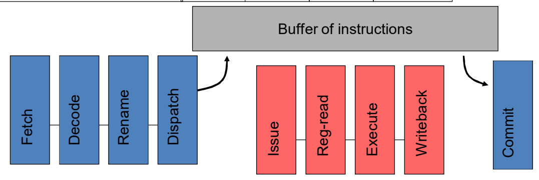
84

84



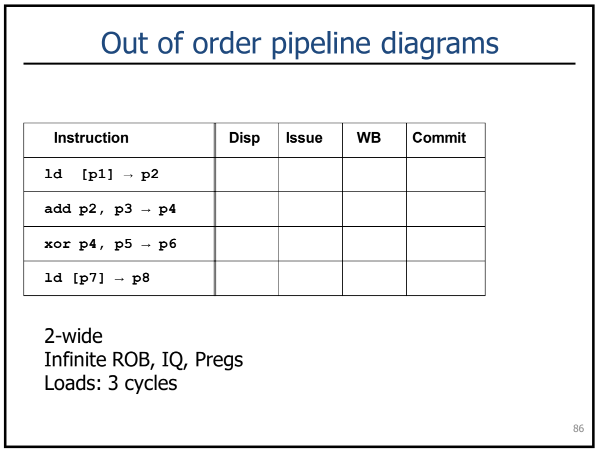
Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`ld [p1] → p2`				
`add p2, p3 → p4`				
`xor p4, p5 → p6`				
`ld [p7] → p8`				



85

85



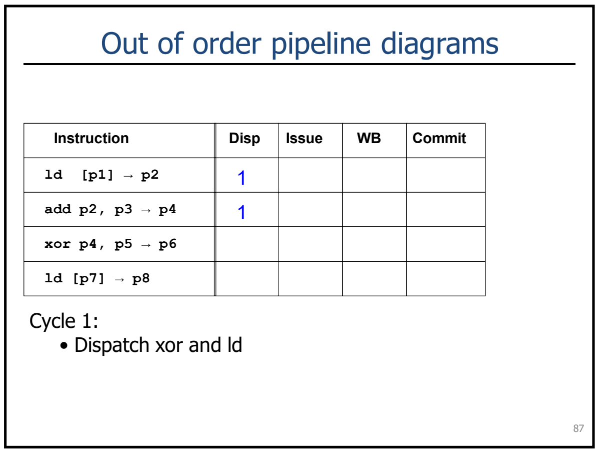
Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`ld [p1] → p2`				
`add p2, p3 → p4`				
`xor p4, p5 → p6`				
`ld [p7] → p8`				

2-wide
Infinite ROB, IQ, Pregs
Loads: 3 cycles

86

86



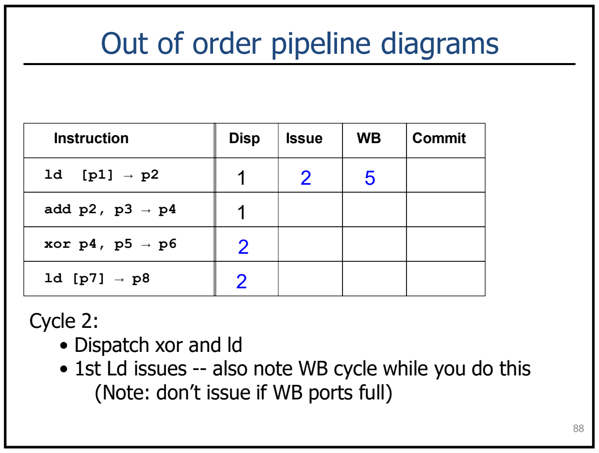
Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`ld [p1] → p2`	1			
`add p2, p3 → p4`	1			
`xor p4, p5 → p6`				
`ld [p7] → p8`				

Cycle 1:
- Dispatch xor and ld

87

87



Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`ld [p1] → p2`	1	2	5	
`add p2, p3 → p4`	1			
`xor p4, p5 → p6`	2			
`ld [p7] → p8`	2			

Cycle 2:
- Dispatch xor and ld
- 1st Ld issues -- also note WB cycle while you do this
  (Note: don't issue if WB ports full)

88

88

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`ld  [p1] → p2`	1	2	5	
`add p2, p3 → p4`	1			
`xor p4, p5 → p6`	2			
`ld [p7] → p8`	2	3	6	

Cycle 3:
- add and xor are not ready
- 2nd load is → issue it

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`ld  [p1] → p2`	1	2	5	
`add p2, p3 → p4`	1	5	6	
`xor p4, p5 → p6`	2			
`ld [p7] → p8`	2	3	6	

Cycle 4:
- nothing

Cycle 5:
- add can issue

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`ld  [p1] → p2`	1	2	5	6
`add p2, p3 → p4`	1	5	6	
`xor p4, p5 → p6`	2	6	7	
`ld [p7] → p8`	2	3	6	

Cycle 6:
- 1st load can commit (oldest instruction & finished)
- xor can issue

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`ld  [p1] → p2`	1	2	5	6
`add p2, p3 → p4`	1	5	6	7
`xor p4, p5 → p6`	2	6	7	
`ld [p7] → p8`	2	3	6	

Cycle 7:
- add can commit (oldest instruction & finished)

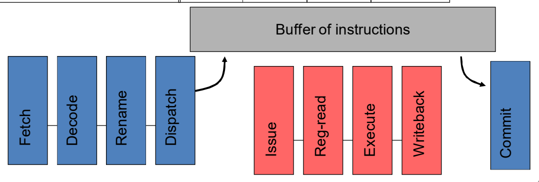
Out of order pipeline diagrams

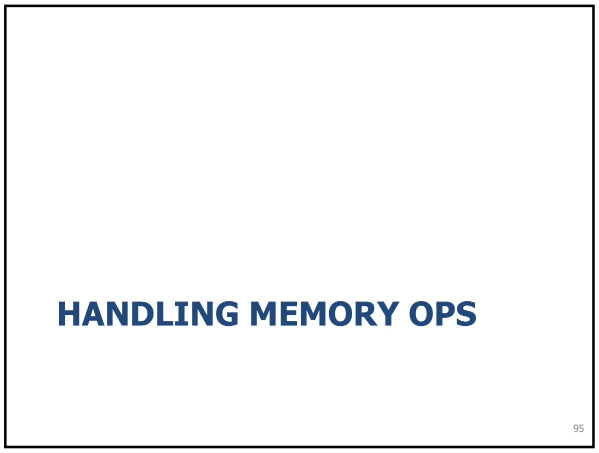
Instruction	Disp	Issue	WB	Commit
`ld  [p1] → p2`	1	2	5	6
`add p2, p3 → p4`	1	5	6	7
`xor p4, p5 → p6`	2	6	7	8
`ld [p7] → p8`	2	3	6	8

Cycle 8:
- xor and ld can commit (2-wide: can do both at once)

Out of order pipeline diagrams

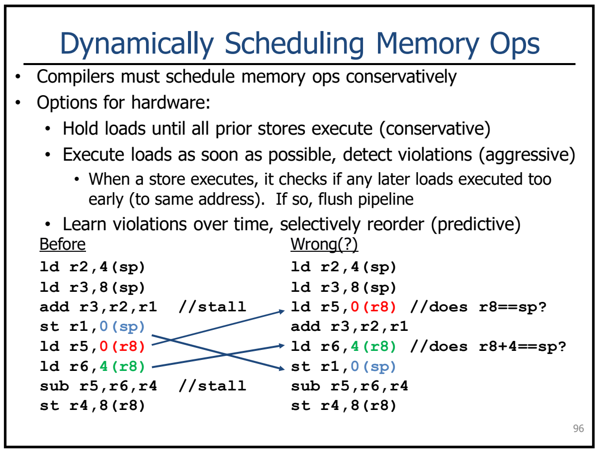
Instruction	Disp	Issue	WB	Commit
`ld  [p1] → p2`	1	2	5	6
`add p2, p3 → p4`	1	5	6	7
`xor p4, p5 → p6`	2	6	7	8
`ld [p7] → p8`	2	3	6	8





HANDLING MEMORY OPS

95



Dynamically Scheduling Memory Ops

- Compilers must schedule memory ops conservatively
- Options for hardware:
  - Hold loads until all prior stores execute (conservative)
  - Execute loads as soon as possible, detect violations (aggressive)
    - When a store executes, it checks if any later loads executed too early (to same address). If so, flush pipeline
  - Learn violations over time, selectively reorder (predictive)

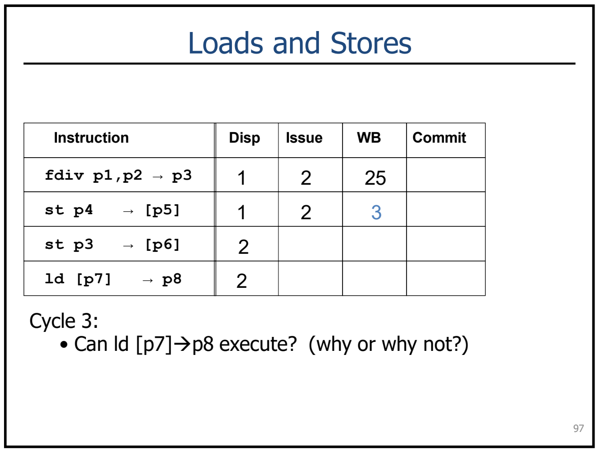
<u>Before</u>

```
ld r2,4(sp)
ld r3,8(sp)
add r3,r2,r1   //stall
st r1,0(sp)
ld r5,0(r8)
ld r6,4(r8)
sub r5,r6,r4   //stall
st r4,8(r8)
```

<u>Wrong(?)</u>

```
ld r2,4(sp)
ld r3,8(sp)
ld r5,0(r8)  //does r8==sp?
add r3,r2,r1
ld r6,4(r8)  //does r8+4==sp?
st r1,0(sp)
sub r5,r6,r4
st r4,8(r8)
```

96

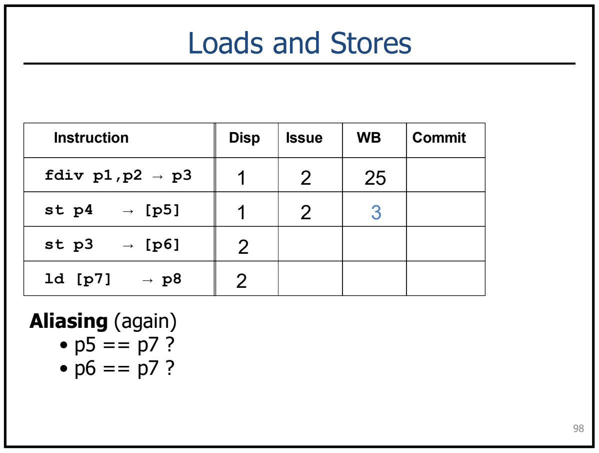


Loads and Stores

Instruction	Disp	Issue	WB	Commit
`fdiv p1,p2 → p3`	1	2	25	
`st p4 → [p5]`	1	2	3	
`st p3 → [p6]`	2			
`ld [p7] → p8`	2			

Cycle 3:
- Can ld [p7]→p8 execute? (why or why not?)

97

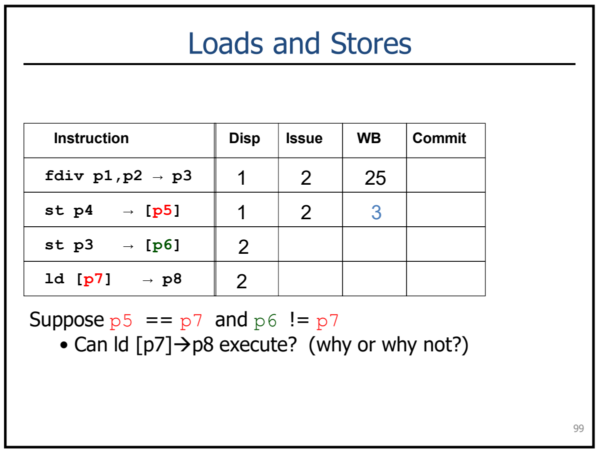


Loads and Stores

Instruction	Disp	Issue	WB	Commit
`fdiv p1,p2 → p3`	1	2	25	
`st p4 → [p5]`	1	2	3	
`st p3 → [p6]`	2			
`ld [p7] → p8`	2			

Aliasing (again)
- p5 == p7 ?
- p6 == p7 ?

98

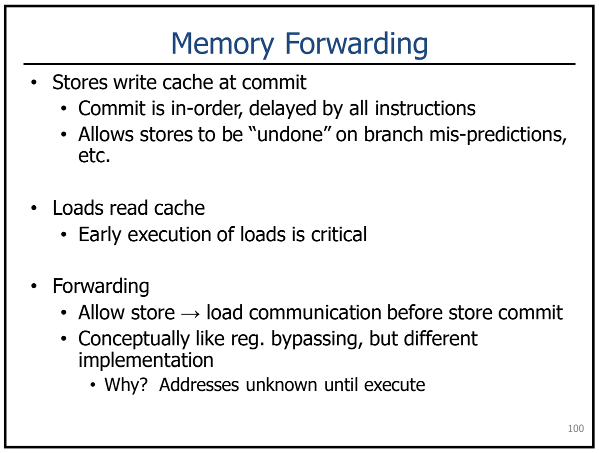


Loads and Stores

Instruction	Disp	Issue	WB	Commit
`fdiv p1,p2 → p3`	1	2	25	
`st p4 → [p5]`	1	2	3	
`st p3 → [p6]`	2			
`ld [p7] → p8`	2			

Suppose `p5` == `p7` and `p6` != `p7`
- Can ld [p7]→p8 execute? (why or why not?)

99



Memory Forwarding

- Stores write cache at commit
  - Commit is in-order, delayed by all instructions
  - Allows stores to be "undone" on branch mis-predictions, etc.
- Loads read cache
  - Early execution of loads is critical
- Forwarding
  - Allow store → load communication before store commit
  - Conceptually like reg. bypassing, but different implementation
    - Why? Addresses unknown until execute

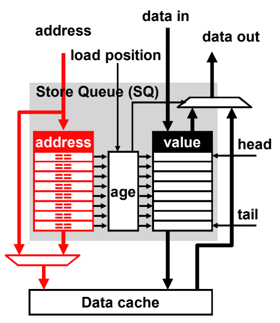
100

Forwarding: Store Queue

Store Queue
- Holds all in-flight stores
- CAM: searchable by address
- Age logic: determine youngest matching store older than load

Store execution
- Write Store Queue
  - Address + Data

Load execution
- Search SQ
  - Match? Forward
- Read D$



Load scheduling

- Store→Load Forwarding:
  - Get value from executed (but not comitted) store to load
- Load Scheduling:
  - Determine when load can execute with regard to older stores

- Conservative load scheduling:
  - All older stores have executed
  - Some architectures: split store address / store data
    - Only require known address
  - Advantage: always safe
  - Disadvantage: performance (limits out-of-orderness)

Our example from before

```
ld   [r1] → r5
ld   [r2] → r6
add  r5,r6 → r7
st   r7 → [r3]
ld   4[r1] → r5
ld   4[r2] → r6
add  r5,r6 → r7
st   r7 → 4[r3]
// loop control here
```

With conservative load scheduling, what can go out of order?

Our example from before

		Disp	Issue	WB	Commit
1	ld [p1] → p5	1			
2	ld [p2] → p6	1			
3	add p5,p6 → p7				
4	st p7 → [p3]				
5	ld 4[p1] → p8				
6	ld 4[p2] → p9				
7	add p8,p9 → p4				
8	st p4 → 4[p3]				

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 1:
Dispatch insns #1, #2

Our example from before

		Disp	Issue	WB	Commit
1	ld [p1] → p5	1	2	5	
2	ld [p2] → p6	1			
3	add p5,p6 → p7	2			
4	st p7 → [p3]	2			
5	ld 4[p1] → p8				
6	ld 4[p2] → p9				
7	add p8,p9 → p4				
8	st p4 → 4[p3]				

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

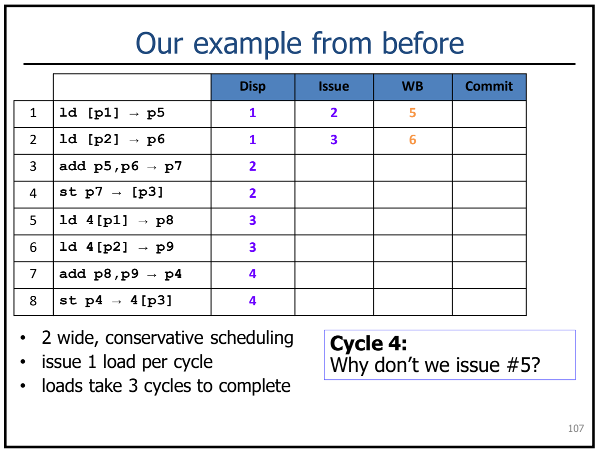
Cycle 2:
Why don't we issue #2?

Our example from before

		Disp	Issue	WB	Commit
1	ld [p1] → p5	1	2	5	
2	ld [p2] → p6	1	3	6	
3	add p5,p6 → p7	2			
4	st p7 → [p3]	2			
5	ld 4[p1] → p8	3			
6	ld 4[p2] → p9	3			
7	add p8,p9 → p4				
8	st p4 → 4[p3]				

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 3:
Why don't we issue #3?
Why don't we issue #4?



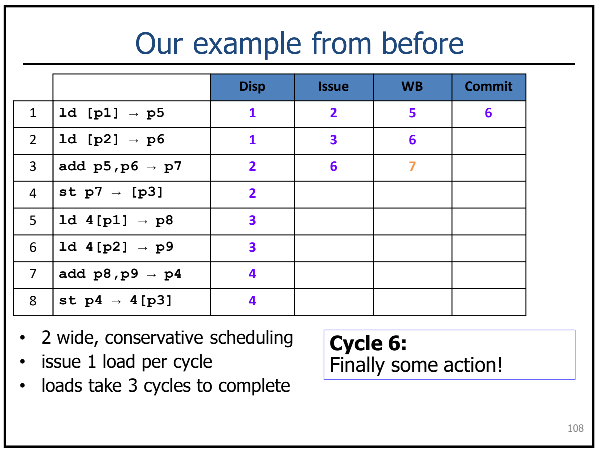
Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	
2	`ld [p2] → p6`	1	3	6	
3	`add p5,p6 → p7`	2			
4	`st p7 → [p3]`	2			
5	`ld 4[p1] → p8`	3			
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 4:
Why don't we issue #5?

107



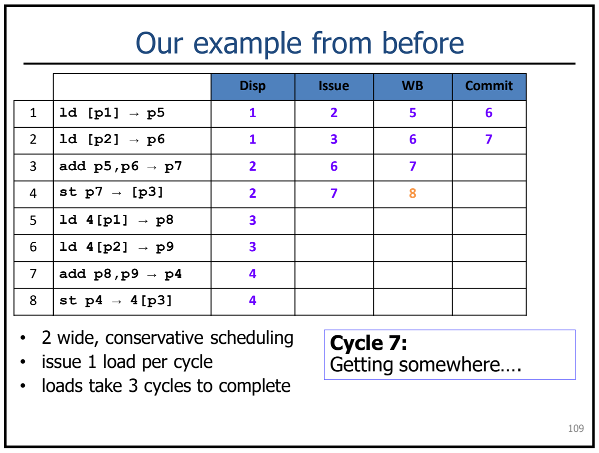
Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	
3	`add p5,p6 → p7`	2	6	7	
4	`st p7 → [p3]`	2			
5	`ld 4[p1] → p8`	3			
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 6:
Finally some action!

108



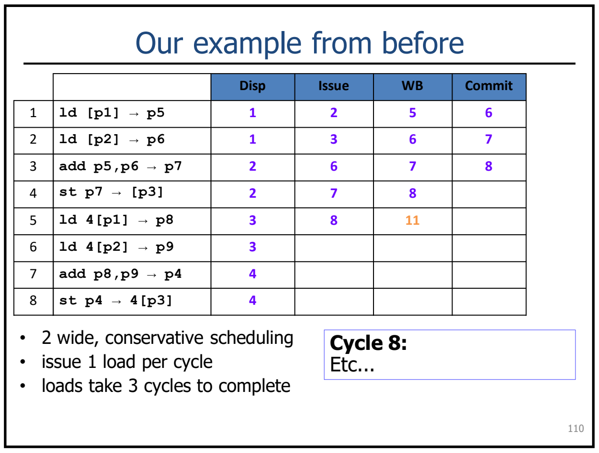
Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	
4	`st p7 → [p3]`	2	7	8	
5	`ld 4[p1] → p8`	3			
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 7:
Getting somewhere….

109



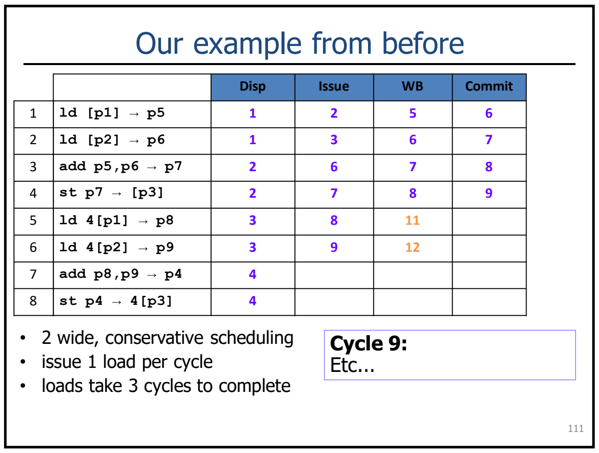
Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	
5	`ld 4[p1] → p8`	3	8	11	
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 8:
Etc…

110



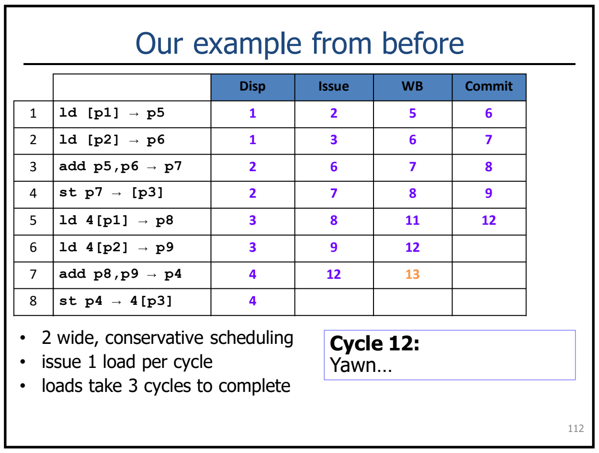
Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	9
5	`ld 4[p1] → p8`	3	8	11	
6	`ld 4[p2] → p9`	3	9	12	
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 9:
Etc…

111



Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	9
5	`ld 4[p1] → p8`	3	8	11	12
6	`ld 4[p2] → p9`	3	9	12	
7	`add p8,p9 → p4`	4	12	13	
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 12:
Yawn…

112

Our example from before

		Disp	Issue	WB	Commit
1	ld [p1] → p5	1	2	5	6
2	ld [p2] → p6	1	3	6	7
3	add p5,p6 → p7	2	6	7	8
4	st p7 → [p3]	2	7	8	9
5	ld 4[p1] → p8	3	8	11	12
6	ld 4[p2] → p9	3	9	12	13
7	add p8,p9 → p4	4	12	13	
8	st p4 → 4[p3]	4	13	14	

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 13:
Stretch…

Our example from before

		Disp	Issue	WB	Commit
1	ld [p1] → p5	1	2	5	6
2	ld [p2] → p6	1	3	6	7
3	add p5,p6 → p7	2	6	7	8
4	st p7 → [p3]	2	7	8	9
5	ld 4[p1] → p8	3	8	11	12
6	ld 4[p2] → p9	3	9	12	13
7	add p8,p9 → p4	4	12	13	14
8	st p4 → 4[p3]	4	13	14	

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 14:
Zzzzzz…

Our example from before

		Disp	Issue	WB	Commit
1	ld [p1] → p5	1	2	5	6
2	ld [p2] → p6	1	3	6	7
3	add p5,p6 → p7	2	6	7	8
4	st p7 → [p3]	2	7	8	9
5	ld 4[p1] → p8	3	8	11	12
6	ld 4[p2] → p9	3	9	12	13
7	add p8,p9 → p4	4	12	13	14
8	st p4 → 4[p3]	4	13	14	15

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 15:
2-wide ooo = 1-wide inorder
I am going to cry.

Our example from before

		Disp	Issue	WB	Commit
1	ld [p1] → p5	1	2	5	6
2	ld [p2] → p6	1	3	6	7
3	add p5,p6 → p7	2	6	7	8
4	st p7 → [p3]	2	7	8	9
5	ld 4[p1] → p8	3	8	11	12
6	ld 4[p2] → p9	3	9	12	13
7	add p8,p9 → p4	4	12	13	14
8	st p4 → 4[p3]	4	13	14	15

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

What was **#5** waiting for??

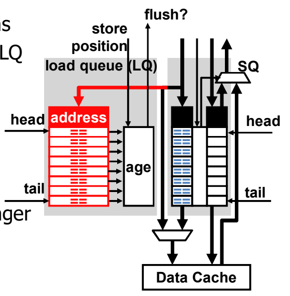
Can I speculate?

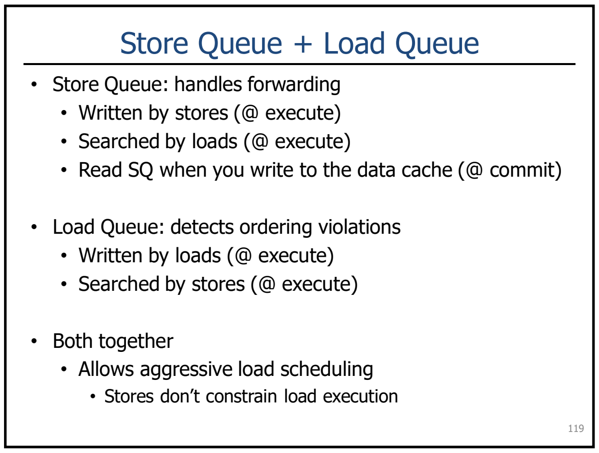
Load Speculation

- Speculation requires two things…..
  - Detection of mis-speculations
    - How can we do this?
  - Recovery from mis-speculations
    - Squash from offending load
    - Saw how to squash from branches: same method

Load Queue

- Detects ld ordering violations
- Execute load: write addr to LQ
  - Also note any store forwarded from
- Execute store: search LQ
  - Younger load with same addr?
  - Didn't forward from younger store?

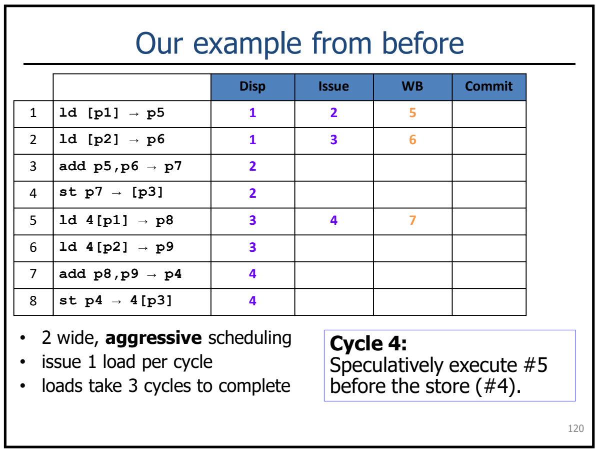




Store Queue + Load Queue

- Store Queue: handles forwarding
  - Written by stores (@ execute)
  - Searched by loads (@ execute)
  - Read SQ when you write to the data cache (@ commit)
- Load Queue: detects ordering violations
  - Written by loads (@ execute)
  - Searched by stores (@ execute)
- Both together
  - Allows aggressive load scheduling
    - Stores don't constrain load execution

119



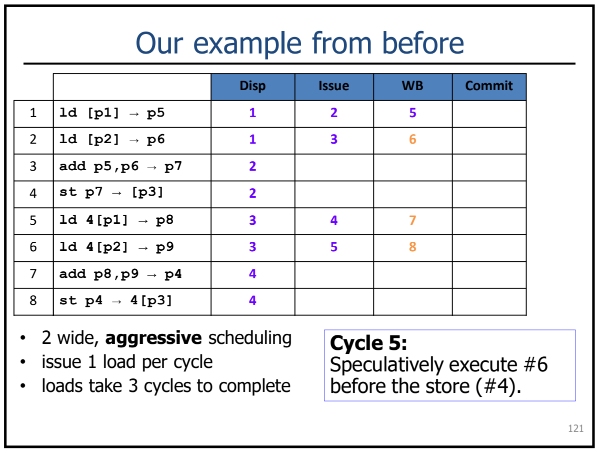
Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	
2	`ld [p2] → p6`	1	3	6	
3	`add p5,p6 → p7`	2			
4	`st p7 → [p3]`	2			
5	`ld 4[p1] → p8`	3	4	7	
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, **aggressive** scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 4:
Speculatively execute #5 before the store (#4).

120



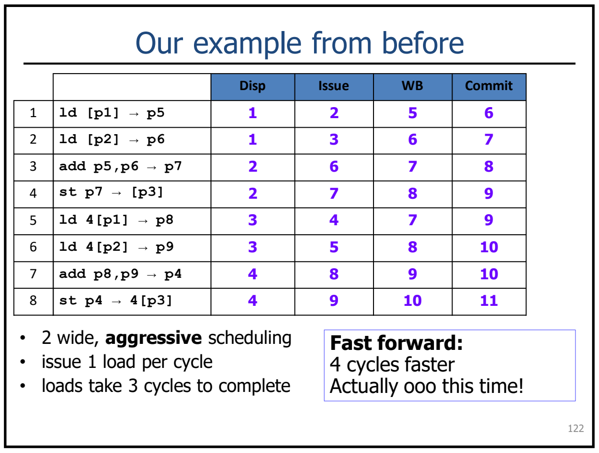
Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	
2	`ld [p2] → p6`	1	3	6	
3	`add p5,p6 → p7`	2			
4	`st p7 → [p3]`	2			
5	`ld 4[p1] → p8`	3	4	7	
6	`ld 4[p2] → p9`	3	5	8	
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, **aggressive** scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 5:
Speculatively execute #6 before the store (#4).

121



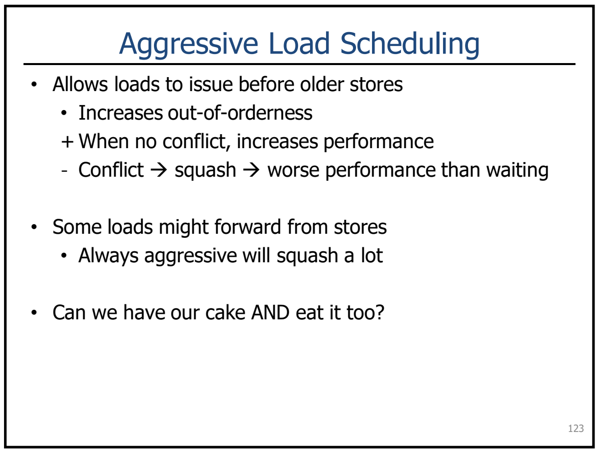
Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	9
5	`ld 4[p1] → p8`	3	4	7	9
6	`ld 4[p2] → p9`	3	5	8	10
7	`add p8,p9 → p4`	4	8	9	10
8	`st p4 → 4[p3]`	4	9	10	11

- 2 wide, **aggressive** scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Fast forward:
4 cycles faster
Actually ooo this time!

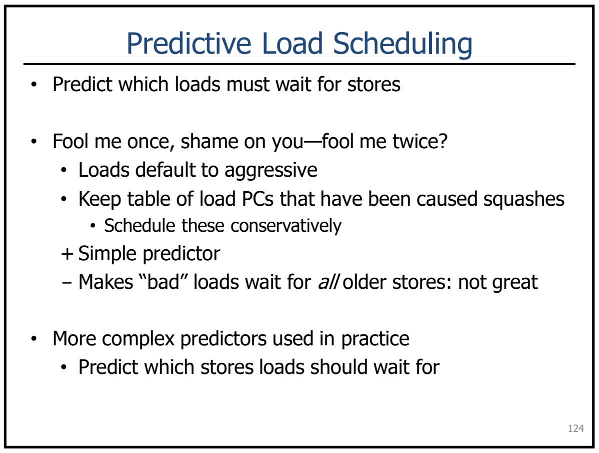
122



Aggressive Load Scheduling

- Allows loads to issue before older stores
  - Increases out-of-orderness
  - + When no conflict, increases performance
  - - Conflict → squash → worse performance than waiting
- Some loads might forward from stores
  - Always aggressive will squash a lot
- Can we have our cake AND eat it too?

123



Predictive Load Scheduling

- Predict which loads must wait for stores
- Fool me once, shame on you—fool me twice?
  - Loads default to aggressive
  - Keep table of load PCs that have been caused squashes
    - Schedule these conservatively
  - + Simple predictor
  - – Makes "bad" loads wait for *all* older stores: not great
- More complex predictors used in practice
  - Predict which stores loads should wait for

124

Out of Order: Window Size

- Scheduling scope = ooo window size
  - Larger = better
  - Constrained by physical registers (#preg)
    - ROB roughly limited by #preg = ROB size + #logical registers
    - Big register file = hard/slow
  - Constrained by issue queue
    - Limits number of un-executed instructions
    - CAM = can't make big (power + area)
  - Constrained by load + store queues
    - Limit number of loads/stores
    - CAMs
    - Active area of research: scaling window sizes
- Usefulness of large window: limited by branch prediction
  - 95% branch mis-prediction rate: 1 in 20 branches, 1 in 100 insns

125

125

Out of Order: Benefits

- Allows speculative re-ordering
  - Loads / stores
  - Branch prediction
- Schedule can change due to cache misses
  - Different schedule optimal from on cache hit
- Done by hardware
  - Compiler may want different schedule for different hw configs
  - Hardware has only its own configuration to deal with

126

126

Static vs. Dynamic Scheduling

- If we can do this in software…
- …why build complex (slow-clock, high-power) hardware?
  - + Performance portability
    - Don't want to recompile for new machines
  - + More information available
    - Memory addresses, branch directions, cache misses
  - + More registers available
    - Compiler may not have enough to schedule well
  - + Speculative memory operation re-ordering
    - Compiler must be conservative, hardware can speculate
  - – But compiler has a larger scope
    - Compiler does as much as it can (not much)
    - Hardware does the rest

127

127

Out of Order: Top 5 Things to Know

- Register renaming
  - How to perform it and how to recover it
- Commit
  - Precise state (ROB)
  - How/when registers are freed
- Issue/Select
  - Wakeup: CAM
  - Choose N oldest ready instructions
- Stores
  - Write at commit
  - Forward to loads via SQ
- Loads
  - Conservative/aggressive/predictive scheduling
  - Violation detection via LQ

128

128

Summary: Dynamic Scheduling

- Dynamic scheduling
  - Totally in the hardware
  - Also called "out-of-order execution" (OoO)
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past (multiple) branches
  - Flush pipeline on branch misprediction
- Rename to avoid false dependencies
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies more tricky
- "Commit" instructions in order
  - Anything strange happens pre-commit, just flush the pipeline
- Current machines: 100+ instruction scheduling window

129

129