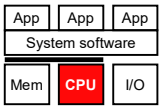
CSE 5600
Computer Systems Architecture

Dynamic Scheduling

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

1

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

2

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

3

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

4

The Problem With In-Order Pipelines

```
                1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
addf f2←f0,f1   F  D  E+ E+ E+ W
mulf f2←f2,f3      F  d* d* D  E* E* E* E* E* W
subf f4←f0,f1         F  p* p* 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` uses W in cycle 11
    - That is the only reason, and it isn't a fundamental one
  - Maintaining in-order writes to reg. file (both don't write `f2`)
- Why can't `subf` go into D in cycle 4 and E+ in cycle 5?

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

R E G	`add r1 ← r2,r3`
	`sub r2 ← r1,r4`
	`or r1 ← r6,r3`
		`add r1 ← r2,r3`
		`sub r2 ← r5,r4`
		`or r1 ← r6,r3`
			`add r1 ← r2,r3`
			`sub r2 ← r1,r4`
			`or r1 ← r6,r3`

M E M	`sw [r2] ← r1` `lw r4 ← [r2]`	`lw r2 ← [r1]` `sw [r1] ← r3`	`st [r2] ← r1` `st [r2] ← r3`

  - 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!)

6

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**
- **r4** from **sub** to **and**
- **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

```
add  r1←r2,r3
sub  r3←r2,r1
mul  r3←r2,r3
divi r1←r1,4
```

False Dependencies

```
add  r1←r2,r3
sub  r3←r2,r1
mul  r3←r2,r3
divi r1←r1,4
```

- "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

```
p4,p5,p6,p7
p5,p6,p7
p6,p7
p7
```

Original insns

```
add r1←r2,r3
sub r3←r2,r1
mul r3←r2,r3
divi r1←r1,4
```

Renamed insns

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

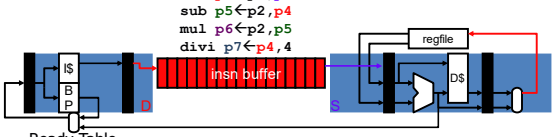
- Renaming: conceptually write each register once
  + Removes **false** dependences
  + Leaves **true** dependences intact!
- When to reuse a physical register? After overwriting insn done

10

10

Step #2: Dynamic Scheduling

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



Ready Table

p2	p3	p4	p5	p6	p7
Yes	Yes				
Yes	Yes	Yes			
Yes	Yes	Yes	Yes		Yes
Yes	Yes	Yes	Yes	Yes	Yes

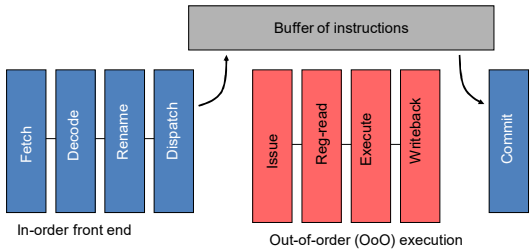
```
add p4←p2,p3
sub p5←p2,p4 and divi p7←p4,4
mul p6←p2,p5
```

- 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

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)
```


Renaming example

Original insns

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

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

p6
p7
p8
p9
p10

Free-list

Renaming example

Original insns → Renamed insns

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

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

Map table

p6
p7
p8
p9
p10

Free-list

Renaming example

Original insns → Renamed insns

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

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

p6
p7
p8
p9
p10

Free-list

Renaming example

Original insns → Renamed insns

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

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

Map table

p7
p8
p9
p10

Free-list

Renaming example

Original insns → Renamed insns

```
xor r3 ← r1, r2        xor  p6 ← p1, p2
add r4 ← r3, r4   →    add  p7 ← p6, p4
sub r3 ← r5, r2        sub  p8 ← p5, p2
addi r1 ← r3, 1        addi r1 ← r3, 1
```

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map table

p7
p8
p9
p10

Free-list

19

19

Renaming example

Original insns → Renamed insns

```
xor r3 ← r1, r2        xor  p6 ← p1, p2
add r4 ← r3, r4   →    add  p7 ← p6, p4
sub r3 ← r5, r2
addi r1 ← r3, 1
```

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map table

p7
p8
p9
p10

Free-list

20

20

Renaming example

Original insns → Renamed insns

```
xor r3 ← r1, r2        xor  p6 ← p1, p2
add r4 ← r3, r4        add  p7 ← p6, p4
sub r3 ← r5, r2   →
addi r1 ← r3, 1
```

r1	p1
r2	p2
r4	p7
r3	p6
r5	p5

Map table

p8
p9
p10

Free-list

21

21

Renaming example

Original insns → Renamed insns

```
xor r3 ← r1, r2        xor  p6 ← p1, p2
add r4 ← r3, r4        add  p7 ← p6, p4
sub r3 ← r5, r2   →
addi r1 ← r3, 1
```

r1	p1
r2	p2
r4	p7
r3	p6
r5	p5

Map table

p8
p9
p10

Free-list

22

22

Renaming example

Original insns → Renamed insns

```
xor r3 ← r1, r2        xor  p6 ← p1, p2
add r4 ← r3, r4        add  p7 ← p6, p4
sub r3 ← r5, r2   →    sub  p8 ← p5, p2
addi r1 ← r3, 1        addi r1 ← r3, 1
```

r1	p1
r2	p2
r4	p7
r3	p8
r5	p5

Map table

p9
p10

Free-list

23

23

Renaming example

Original insns → Renamed insns

```
xor r3 ← r1, r2        xor  p6 ← p1, p2
add r4 ← r3, r4        add  p7 ← p6, p4
sub r3 ← r5, r2   →    sub  p8 ← p5, p2
addi r1 ← r3, 1        addi r1 ← r3, 1
```

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p9
p10

Free-list

24

24

Renaming example

Original insns

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

Renamed insns

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

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

Map table

p9
p10

Free-list

25

25

Renaming example

Original insns

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

Renamed insns

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

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p9
p10

Free-list

26

26

Renaming example

Original insns

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

Renamed insns

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

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

Map table

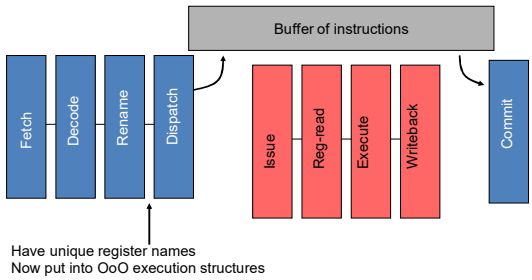
p10

Free-list

27

27

Out-of-order Pipeline



28

28

DYNAMIC SCHEDULING

29

29

Dispatch

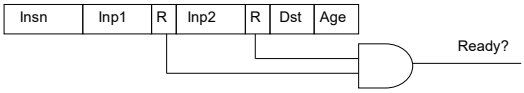
- Renamed instructions into OoO structures
  - Re-order buffer (ROB)
    - Holds all instructions until they commit
  - Issue Queue
    - Un-executed instructions
    - Central piece of scheduling logic
    - Content Addressable Memory (CAM) (more later)

30

30

Issue Queue

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



31

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

32

Dispatch Example

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

Ready bits

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

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Age

33

Dispatch Example

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

Ready bits

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

Issue Queue

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

34

Dispatch Example

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

Ready bits

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

Issue Queue

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

35

Dispatch Example

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

Ready bits

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

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

36

Dispatch Example

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

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

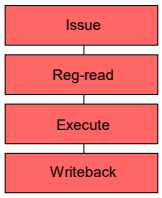
p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	n
p9	**n**

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

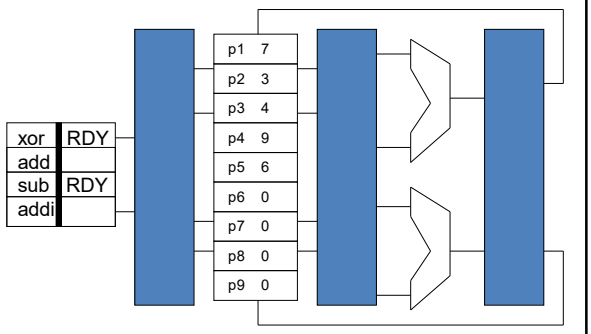
42

Register Read

- When do instructions read the register file?
- Option #1: after issue, 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 dispatch, keep values in Issue Queue
  - Pentium Pro, Core 2, Core i7

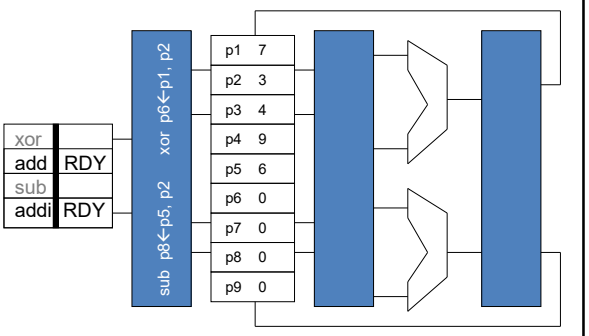
43

OoO execution (2-wide)



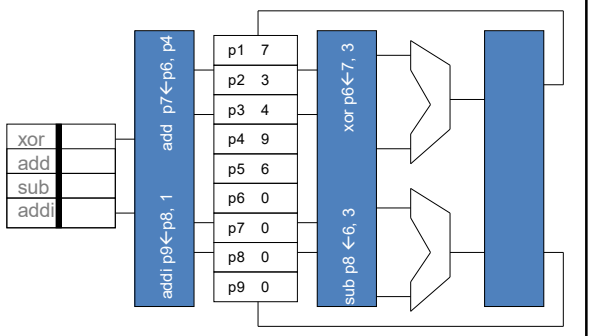
48

OoO execution (2-wide)



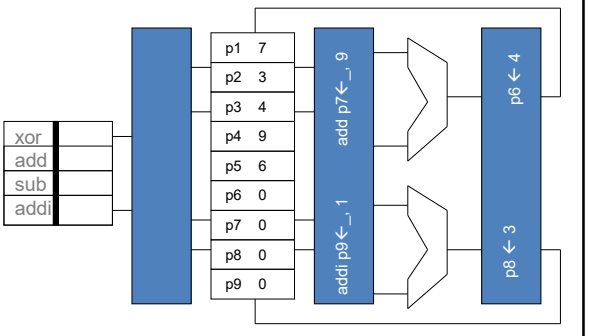
49

OoO execution (2-wide)



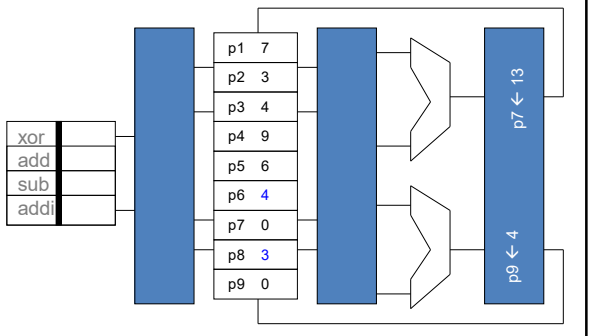
50

OoO execution (2-wide)



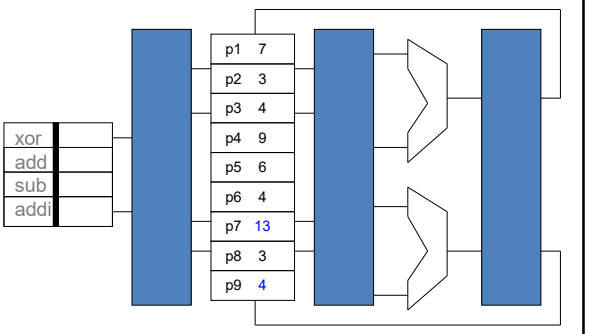
51

OoO execution (2-wide)



52

OoO execution (2-wide)



53

53

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

54

54

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

55

55

RENAMING REVISITED

56

56

Renaming revisited

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

57

57

Renaming example

Original insns

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

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

p6
p7
p8
p9
p10

Free-list

58

58

Renaming example

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

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 r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4		
sub r3 ← r5, r2		
addi r1 ← r3, 1		

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

Map table

p7
p8
p9
p10

Free-list

60

60

Renaming example

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

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 r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4	add p7 ← p6, p4	[p4]
sub r3 ← r5, r2		
addi r1 ← r3, 1		

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

Map table

p8
p9
p10

Free-list

62

62

Renaming example

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

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 r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4	add p7 ← p6, p4	[p4]
sub r3 ← r5, r2	sub p8 ← p5, p2	[p6]
addi r1 ← r3, 1		

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

Map table

p9
p10

Free-list

64

64

Renaming example

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

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map table

Free-list: p9, p10

65

65

Renaming example

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

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

Free-list: p10

66

66

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

67

67

Recovery

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

68

68

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]
xor r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4	add p7 ← p6, p4	[p4]
sub r3 ← r5, r2	sub p8 ← p5, p2	[p6]
addi r1 ← r3, 1	addi p9 ← p8, 1	[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

Free-list: p10

69

69

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]
xor r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4	add p7 ← p6, p4	[p4]
sub r3 ← r5, r2	sub p8 ← p5, p2	[p6]
addi r1 ← r3, 1	addi p9 ← p8, 1	[p1]

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map table

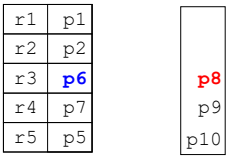
Free-list: p9, p10

70

70

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]
xor r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4	add p7 ← p6, p4	[p4]
sub r3 ← r5, r2	sub **p8** ← p5, p2	**[p6]**



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

Map table

p8
p9
p10

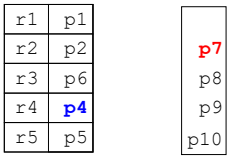
Free-list

71

71

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]
xor r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4	add **p7** ← p6, p4	**[p4]**



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

Map table

p7
p8
p9
p10

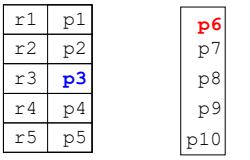
Free-list

72

72

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]
xor r3 ← r1, r2	xor **p6** ← p1, p2	**[p3]**



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

Map table

p6
p7
p8
p9
p10

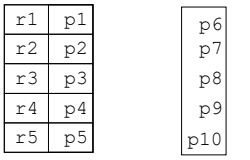
Free-list

73

73

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]



r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

p6
p7
p8
p9
p10

Free-list

74

74