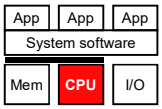
CSE 560
Computer Systems Architecture

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

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

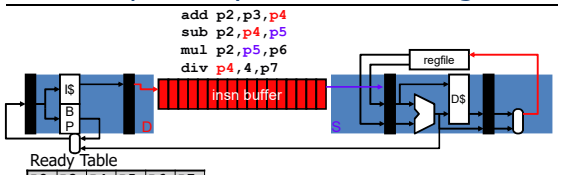
Original insns	Renamed insns
add r2,r3,r1	add p2,p3,p4
sub r2,r1,r3	sub p2,p4,p5
mul r2,r3,r3	mul p2,p5,p6
div r1,4,r1	div p4,4,p7

- 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



Ready Table

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

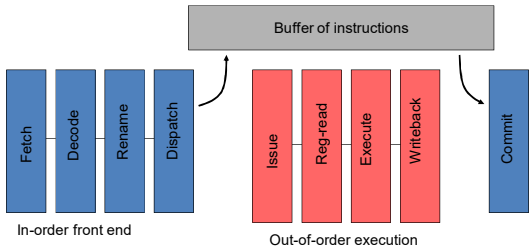
add p2,p3,p4
sub p2,p4,p5 **and** div p4,4,p7
mul p2,p5,p6

- 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

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

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

15

Renaming example

Original insns → Renamed insns

```
xor r1, r2 → r3      xor  p1, p2 →
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

16

Renaming example

Original insns → Renamed insns

```
xor r1, r2 → r3      xor  p1, p2 → p6
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

17

Renaming example

Original insns → Renamed insns

```
xor r1, r2 → r3      xor  p1, p2 → p6
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

18

Renaming example

Original insns	Renamed insns
xor r1, r2 → r3	xor p1, p2 → p6
add **r3**, **r4** → r4	add **p6**, **p4** →
sub r5, r2 → r3	
addi r3, 1 → r1	

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

Free-list
p7
p8
p9
p10

19

19

Renaming example

Original insns	Renamed insns
xor r1, r2 → r3	xor p1, p2 → p6
add r3, r4 → r4	add p6, p4 → **p7**
sub r5, r2 → r3	
addi r3, 1 → r1	

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

Free-list
p7
p8
p9
p10

20

20

Renaming example

Original insns	Renamed insns
xor r1, r2 → r3	xor p1, p2 → p6
add r3, r4 → **r4**	add p6, p4 → p7
sub r5, r2 → r3	
addi r3, 1 → r1	

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

Free-list
p8
p9
p10

21

21

Renaming example

Original insns	Renamed insns
xor r1, r2 → r3	xor p1, p2 → p6
add r3, r4 → r4	add p6, p4 → p7
sub r5, r2 → r3	sub **p5**, **p2** →
addi r3, 1 → r1	

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

Free-list
p8
p9
p10

22

22

Renaming example

Original insns	Renamed insns
xor r1, r2 → r3	xor p1, p2 → p6
add r3, r4 → r4	add p6, p4 → p7
sub r5, r2 → r3	sub p5, p2 → **p8**
addi r3, 1 → r1	

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

Free-list
p8
p9
p10

23

23

Renaming example

Original insns	Renamed insns
xor r1, r2 → r3	xor p1, p2 → p6
add r3, r4 → r4	add p6, p4 → p7
sub r5, r2 → **r3**	sub p5, p2 → p8
addi r3, 1 → r1	

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

Free-list
p9
p10

24

24

Renaming example

Original insns	Renamed insns
xor r1, r2 → r3	xor p1, p2 → p6
add r3, r4 → r4	add p6, p4 → p7
sub r5, r2 → r3	sub p5, p2 → p8
addi r3, 1 → r1	addi **p8**, 1 →

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

Free-list: p9, p10

25

25

Renaming example

Original insns	Renamed insns
xor r1, r2 → r3	xor p1, p2 → p6
add r3, r4 → r4	add p6, p4 → p7
sub r5, r2 → r3	sub p5, p2 → p8
addi r3, 1 → r1	addi p8, 1 → **p9**

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

Free-list: **p9**, p10

26

26

Renaming example

Original insns	Renamed insns
xor r1, r2 → r3	xor p1, p2 → p6
add r3, r4 → r4	add p6, p4 → p7
sub r5, r2 → r3	sub p5, p2 → p8
addi r3, 1 → **r1**	addi p8, 1 → p9

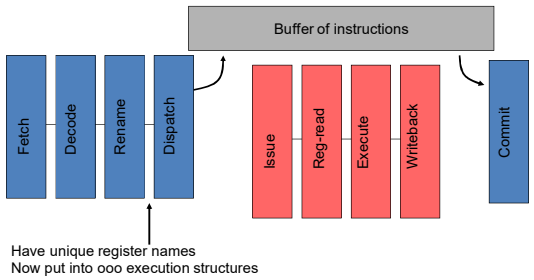
Map table	
r1	**p9**
r2	p2
r3	p8
r4	p7
r5	p5

Free-list: p10

27

27

Out-of-order Pipeline



28

28

DYNAMIC SCHEDULING

29

29

Dispatch

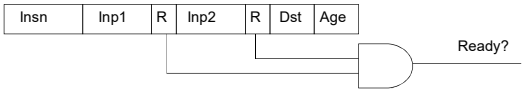
- Renamed instructions into ooo structures
  - Re-order buffer (ROB)
    - Holds all instructions until they commit
  - Issue Queue
    - Un-executed instructions
    - Central piece of scheduling logic
    - Content Addressable Memory (CAM) (more later)

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

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

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

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

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

Ready bits

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

Issue Queue

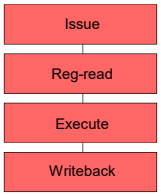
Insn	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1
sub	p5	y	p2	y	p8	2
addi	p8	n	---	y	p9	3

37

37

Out-of-order pipeline

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



38

38