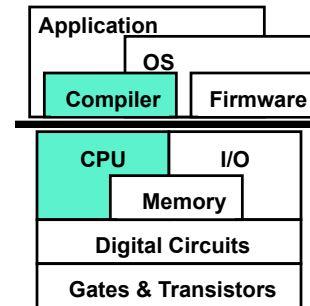


CIS 501 Computer Architecture

Unit 8: Static and Dynamic Scheduling

This Unit: Scheduling (Static + Dynamic)



- Previously:
 - Pipelining
 - Multiple stages
 - Different instructions each stage
 - Superscalar
 - Multiple instructions in each stage
 - "N-wide"
- Now:
 - Compiler (static) scheduling
 - Hardware (dynamic) scheduling

Readings

- H+P
 - None (not happy with explanation of this topic)
- Papers
 - Alpha 21164
 - Due today
 - Discussion
 - Alpha 21264
 - Due next week

Review Example

loop:	1	2	3	4	5	6	7	8	9	A	B
ld [r1] -> r2	F	D	X	M	W						
add r2 + r3 -> r2		F	D	d*	X	M	W				
st r2 -> [r1]			F	p*	D	X	M	W			
addi r1 + 4 -> r1					F	D	X	M	W		
sub r1, r5 -> r6						F	D	X	M	W	
inz r6, loop							F	D	X	M	W

$$IPC = 6/7 = 0.86$$

On a single-issue, 5-stage pipeline,

How many cycles does each loop iteration take?

Assume all cache hits and perfect branch prediction

Review Example

loop:	1	2	3	4	5	6	7	8	9	A	B
ld [r1] -> r2	F	D	X	M	W						
add r2 + r3 -> r2	F	D	d*	d*	X	M	W				
st r2 -> [r1]	F	p*	p*	D	X	M	W				
addi r1 + 4 -> r1	F	p*	p*	D	X	M	W				
sub r1, r5 -> r6					F	D	X	M	W		
jnz r6, loop					F	d*	D	X	M	W	

IPC = 6/6 = 1.0

16% speedup

What if the pipeline is 2-wide?

Would we get any more performance by going 4-wide?

Un-optimized code

```

do {
    *p = *p + x;
    p++;
}
while (p != &a[N]);

loop:
    ld [r1] -> r2
    add r2 + r3 -> r2
    st r2 -> [r1]
    addi r1 + 4 -> r1
    sub r1, r5 -> r6
    jnz r6, loop
    
```

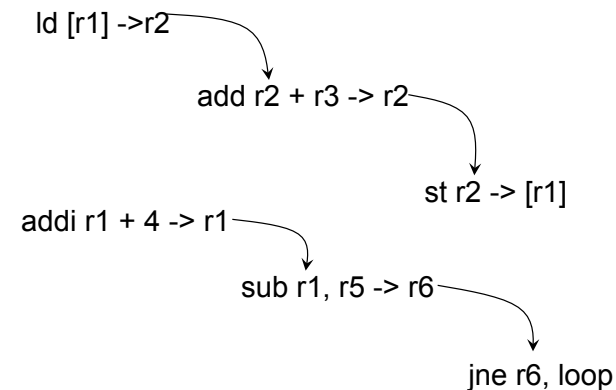
Compiler has taken the code and just emitted assembly in the order statements appear

Can we do better?

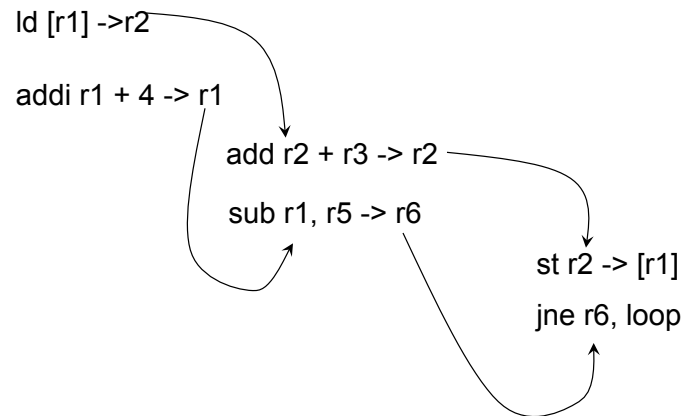
Scheduling Code

- Compiler can re-order instructions
 - Eliminate RAW stalls
 - Place independent instructions near each other
- Called **static scheduling**
- Must be careful to preserve program behavior in **all** cases!

Dataflow graph



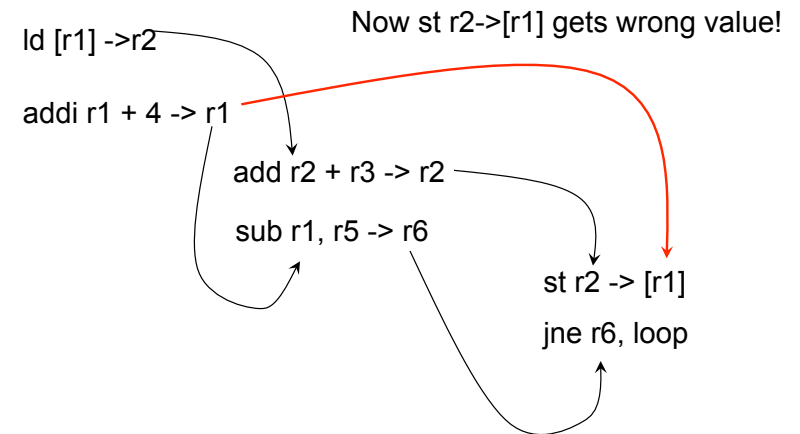
Optimization



CIS 501: Scheduling

9

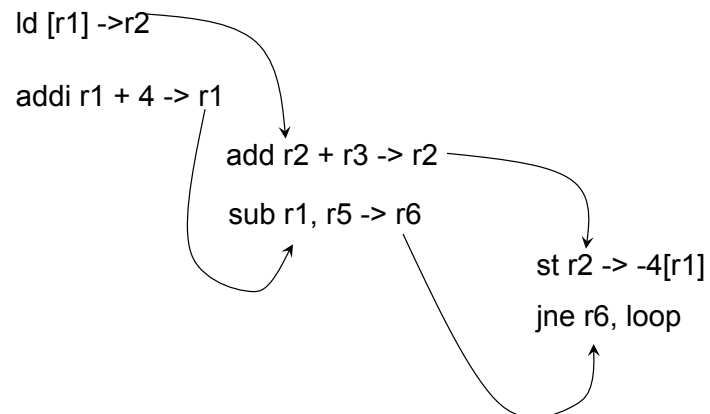
A problem with that



CIS 501: Scheduling

10

Fixed



CIS 501: Scheduling

11

Optimized code

```
loop:
ld [r1] -> r2
addi r1 + 4 -> r1
add r2 + r3 -> r2
sub r1, r5 -> r6
st r2 -> [r1]
jnz r6, loop

do {
    *p = *p + x;
    p++;
}
while (p != &a[N]);
```

Now pieces of each statement are interleaved.

(Aside: why debugging optimized code is confusing)

CIS 501: Scheduling

12

How fast now?

loop:	1	2	3	4	5	6	7	8	9	A	B
ld [r1] -> r2	F	D	X	M	W						
addi r1 + 4 -> r1		F	D	X	M	W					
add r2 + r3 -> r2			F	D	X	M	W				
sub r1, r5 -> r6				F	D	X	M	W			
st r2 -> -4[r1]					F	D	X	M	W		
inzb r6, loop						F	D	X	M	W	

IPC = 6/6 = 1.0

as fast as 2-wide
unoptimized

How fast is this on a 1-wide machine?

How fast now?

loop:	1	2	3	4	5	6	7	8	9	A	B
ld [r1] -> r2	F	D	X	M	W						
addi r1 + 4 -> r1	F	D	X	M	W						
add r2 + r3 -> r2		F	D	d*	X	M	W				
sub r1, r5 -> r6		F	D	p*	X	M	W				
st r2 -> -4[r1]			F	p*	D	X	M	W			
inzb r6, loop			F	p*	D	X	M	W			

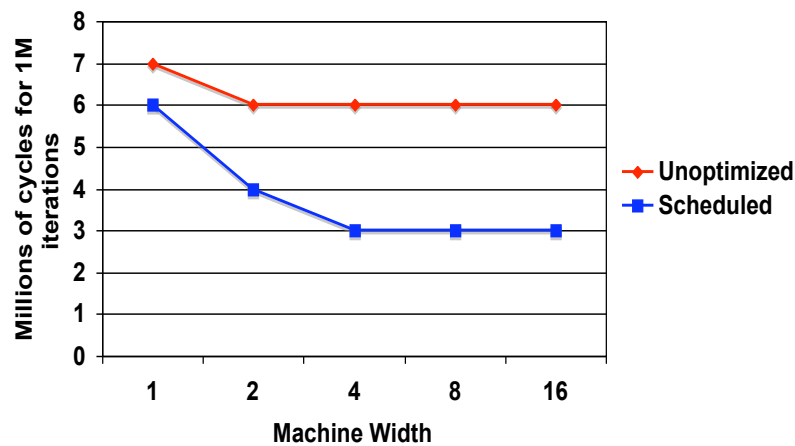
IPC = 6/4 = 1.5

50% speedup

How fast is this on a 2-wide machine?

What about 4-wide? 8? 16?

Performance vs width



Room to improve?

- Code is much better
 - 2-wide performance greatly improved
 - 4-wide now useful
- Can we do better?
 - With the **scheduling scope** shown: no
 - Larger scheduling scope: yes

Scheduling Scope

- Window of instructions we can re-order in
 - Larger => better schedules
 - Compiler: theoretically whole program
 - Not practical for many reasons...
- How?
 - One way: Loop un-rolling
 - Others exist: not the scope of this class

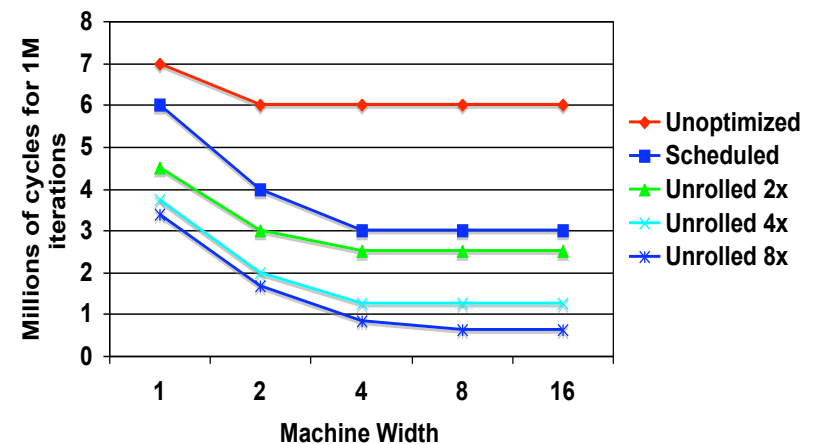
Loop un-rolling

- Take 2 (or more) iterations
- Remove extra loop control
 - Getting rid of extra instructions saves time!
 - Note: must compare cycles, not IPC now
- Re-schedule both together
 - Larger scope to schedule from
 - Register names may need changing

Loop unrolling

<pre> ld [r1] -> r2 add r2 + r3 -> r2 st r2 -> [r1] addi r1 + 4 -> r1 sub r1, r5 -> r6 jnz r6, loop </pre>	→	<pre> ld [r1] -> r2 add r2 + r3 -> r2 st r2 -> [r1] addi r1 + 4 -> r1 sub r1, r5 -> r6 jnz r6, loop </pre>	→	<pre> ld [r1] -> r2 addi r1 + 8 -> r1 ld -4[r1] -> r7 sub r1, r5 -> r6 add r2 + r3 -> r2 add r7 + r3 -> r7 st r2 -> -8[r1] st r7 -> -4[r1] jnz r6, loop </pre>
---	---	---	---	--

Performance vs width



Why not unroll 1K times?

- More unrolling => more performance
 - Fewer dynamic instructions
 - Better scheduling
- Downsides / limiting factors?
 - Number of registers
 - More static instructions => \$I pressure

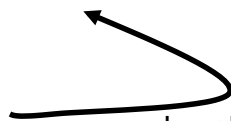
Limitations of static scheduling

- Assumes cache hits
 - Common case
 - Miss? Different schedule maybe better
- Compiler must be conservative
 - Needs to guarantee correctness
 - Sometimes tough to tell if re-ordering is legal

Re-ordering barrier: branches

loop:

```
jz r1, not_found
ld [r1] -> r2
sub r1, r2 -> r2
jz r2, found
ld 4[r1] -> r1
jmp loop
```



Legal to move load up?

No: if r1 is null, will cause a fault

Re-ordering barrier: ld/st

```
ld [r1] -> r2
st r3 -> [r4]
```


Can these be switched?

No: r1 and r4 may be same value

- Technical term: **alias**
 - Two names for same memory location

An example

```
void f(int * a, int *b, int *c, int N) {  
  for (int i =0; i < N; i++) {  
    a[i] = b[i] + c[i];  
  }  
}
```



```
ld [r1] -> r5  
ld [r2] -> r6  
add r5 + r6 -> r7  
st r7 -> [r3]
```

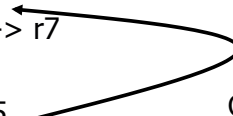
Loop unrolled 2x

```
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
```

What can we re-order here?

Loop unrolled 2x

```
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]  
// add 8 to r1, r2, and r3
```



Can we move this load up?
No: r1+4 might equal r3

Aliasing problems

- Must be conservative
 - f(ptr+4, ptr, ptr) not common case
 - but is possible
- If only we could speculate...
 - Allow re-ordering in the common case
 - Get correctness in the rare case
- Anything software can do, hardware can do better..

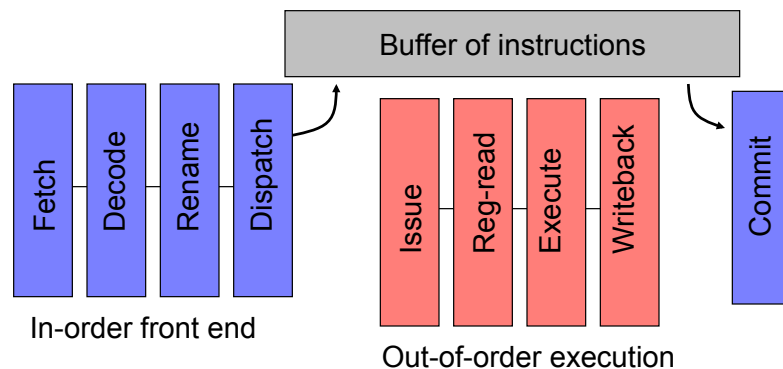
Out-of-order execution

- Hardware can speculate
 - Load/store ordering
 - Branches
- D\$ misses?
 - Compiler: no idea
 - Hardware: knows when they happen
- Out-of-order execution
 - Aka dynamic scheduling

Out-of-order execution

- Execute out of program order
 - Execute oldest **ready** instruction
 - Ready: all input values available
 - Reduce RAW stalls
- Retain appearance of in-order
 - Maintain correctness

Out-of-order pipeline



Register renaming

- Recall static scheduling:

```
xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1
```

- sub/add can be re-ordered
- Must change register of sub

→

```
xor r1 ^ r2 -> r3
sub r5 - r2 -> r7
add r3 + r4 -> r4
addi r7 + 1 -> r1
```

Register renaming

- Same principle applies to hardware
 - Might re-order anything
 - Create unique names
- Logical registers => physical registers
 - Map table: holds translation
 - Indexed by logical register
 - Holds physical register numbers

Register renaming steps

- Read input numbers from map table
- Allocate new physical register
 - None available? => stall
- Update map table with destination reg

Renaming example

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

Renaming example

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

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

p6
p7
p8
p9
p10

Free-list

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map table

p7
p8
p9
p10

Free-list

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map table

p7
p8
p9
p10

Free-list

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map table

p7
p8
p9
p10

Free-list

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

$\text{xor } p1 \wedge p2 \rightarrow p6$
 $\text{add } p6 + p4 \rightarrow p7$

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map table

p8
p9
p10

Free-list

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

$\text{xor } p1 \wedge p2 \rightarrow p6$
 $\text{add } p6 + p4 \rightarrow p7$
 $\text{sub } p5 - p2 \rightarrow$

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map table

p8
p9
p10

Free-list

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

$\text{xor } p1 \wedge p2 \rightarrow p6$
 $\text{add } p6 + p4 \rightarrow p7$
 $\text{sub } p5 - p2 \rightarrow p8$

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map table

p8
p9
p10

Free-list

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

$\text{xor } p1 \wedge p2 \rightarrow p6$
 $\text{add } p6 + p4 \rightarrow p7$
 $\text{sub } p5 - p2 \rightarrow p8$

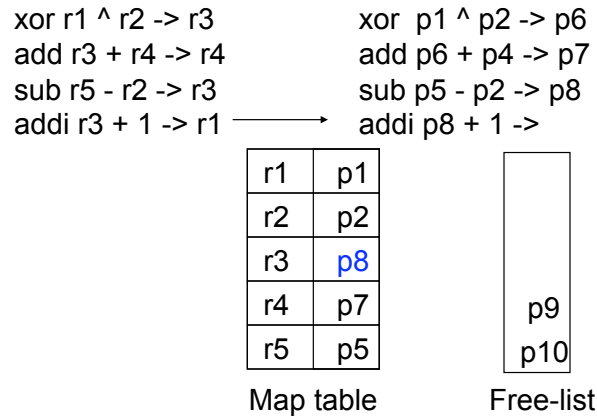
r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map table

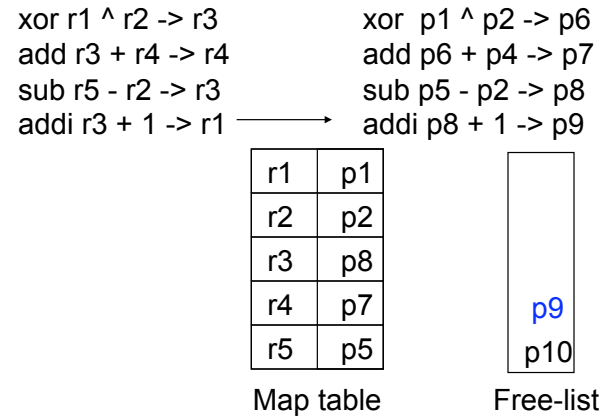
p9
p10

Free-list

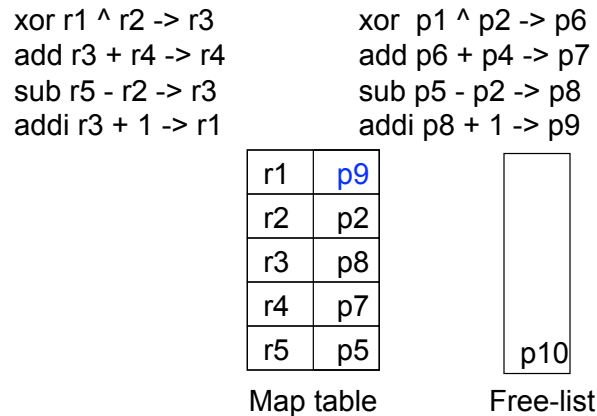
Renaming example



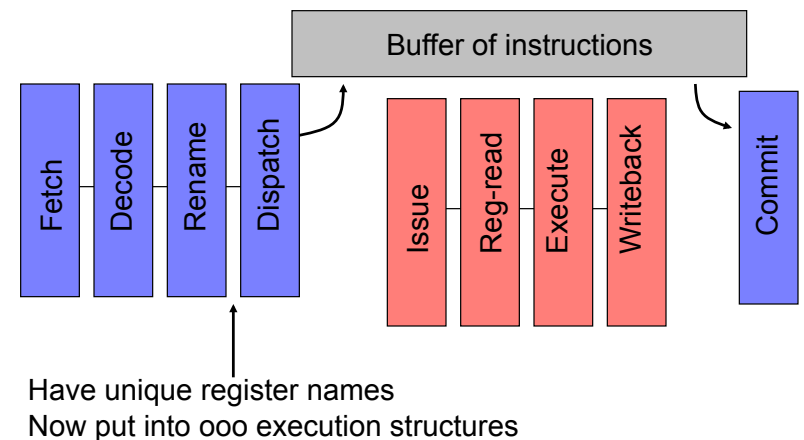
Renaming example



Renaming example



Out-of-order pipeline



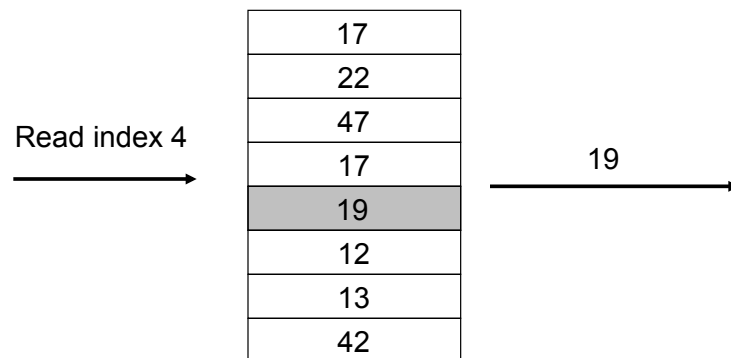
Dispatch

- Renamed instructions into ooo structures
 - Re-order buffer (ROB)
 - All instruction until commit
 - Issue Queue
 - Un-executed instructions
 - Central piece of scheduling logic
 - Content Addressable Memory (CAM)

RAM vs CAM

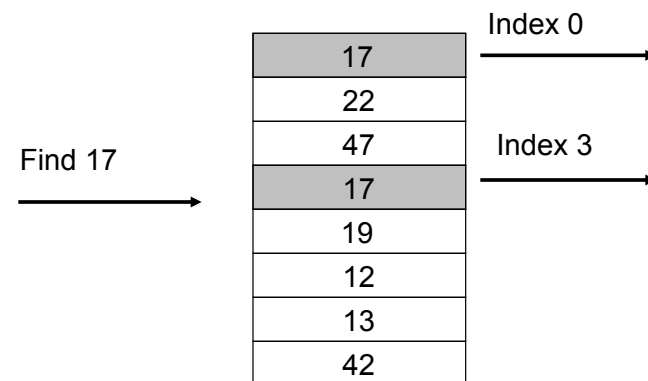
- Random Access Memory
 - Read/write specific index
 - Get/set value there
- Content Addressable Memory
 - Search for a value
 - Find matching indices
- One structure can have ports of both types

RAM vs CAM: RAM



RAM: read/write specific index

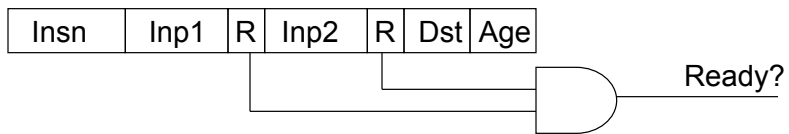
RAM vs CAM: CAM



CAM: search for value

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

CIS 501: Scheduling

Ready bits

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

57

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

CIS 501: Scheduling

Ready bits

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

58

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

CIS 501: Scheduling

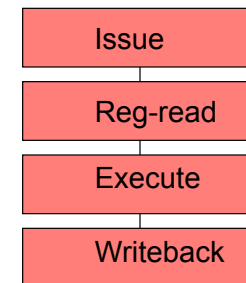
Ready bits

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

59

Out-of-order pipeline

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



CIS 501: Scheduling

60

Issue = Select + Wakeup

- **Select** N oldest, ready instructions

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	

- N == 1? xor
- N >= 2? xor and sub
- Note: may have resource constraints: i.e. ld/st/fp

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

Issue

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

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

Register Read

- Not done at decode
 - Must read **physical** register (renamed)
 - Must be done when value ready
 - Or gone thru when expecting bypass
- Physical register file may be large
 - Multi-cycle read

Renaming review

Everyone rename this instruction:

mul r4 * r5 -> r1

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

p6
p7
p8
p9
p10

Free-list

Dispatch Review

Everyone dispatch this instruction:

div p7 / p6 -> p1

Insn	Inp1	R	Inp2	R	Dst	Age

Ready bits

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

Select Review

Insn	Inp1	R	Inp2	R	Dst	Age
add	p3	y	p1	y	p2	0
mul	p2	n	p4	y	p5	1
div	p1	y	p5	n	p6	2
xor	p4	y	p1	y	p9	3

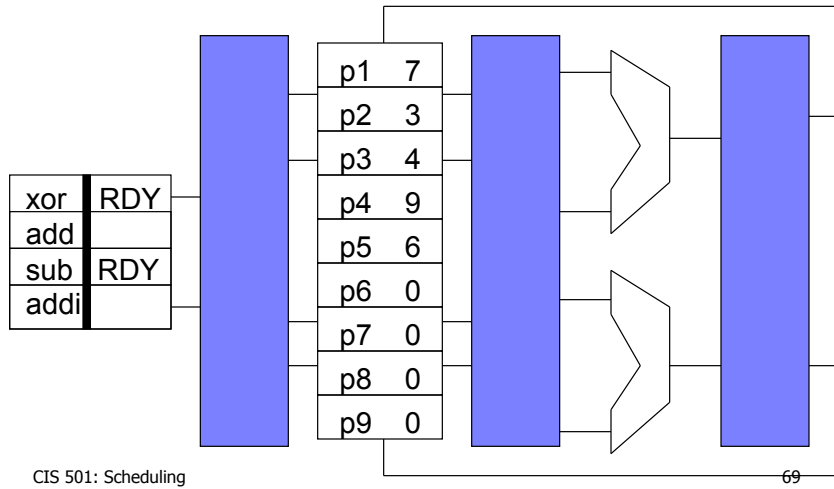
Determine which instructions are ready.
Which will be issued on a 1-wide machine?
Which will be issued on a 2-wide machine?

Wakeup Review

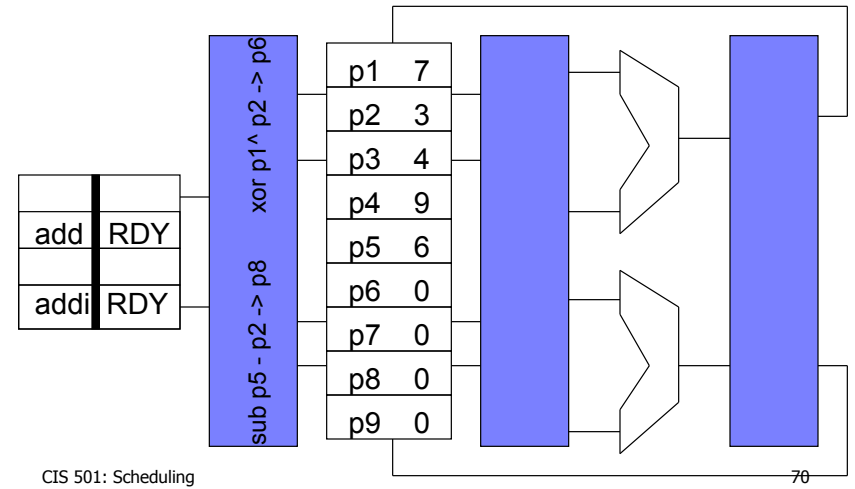
Insn	Inp1	R	Inp2	R	Dst	Age
add	p3	y	p1	y	p2	0
mul	p2	n	p4	y	p5	1
div	p1	y	p5	n	p6	2
xor	p4	y	p1	y	p9	3

What information will change if we issue the add?

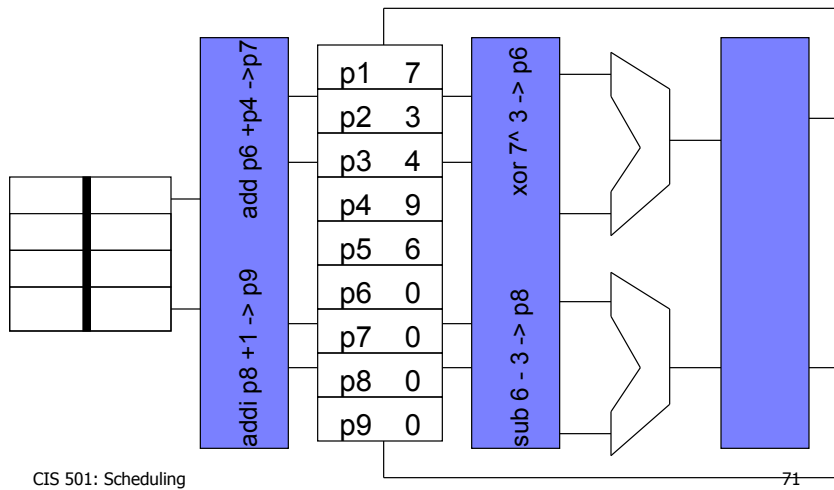
OOO execution (2-wide)



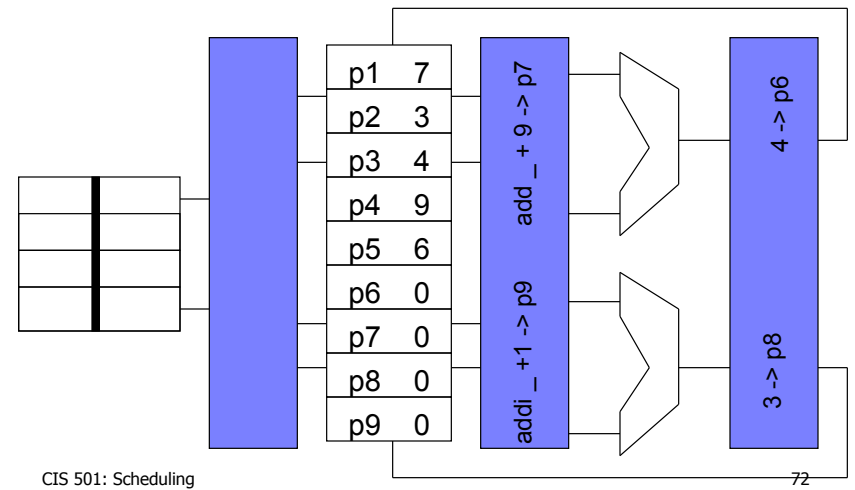
OOO execution (2-wide)



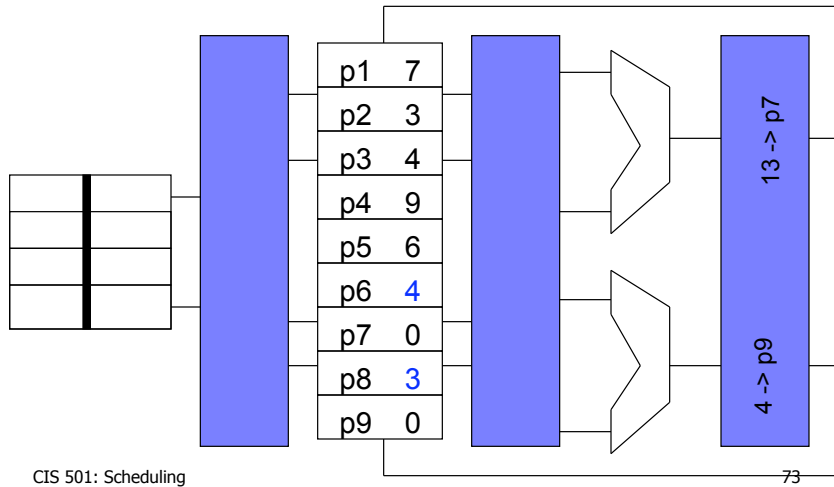
OOO execution (2-wide)



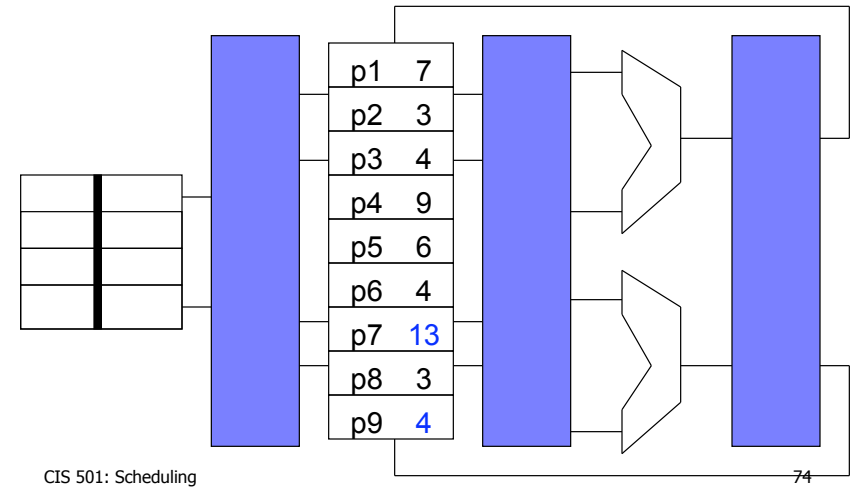
OOO execution (2-wide)



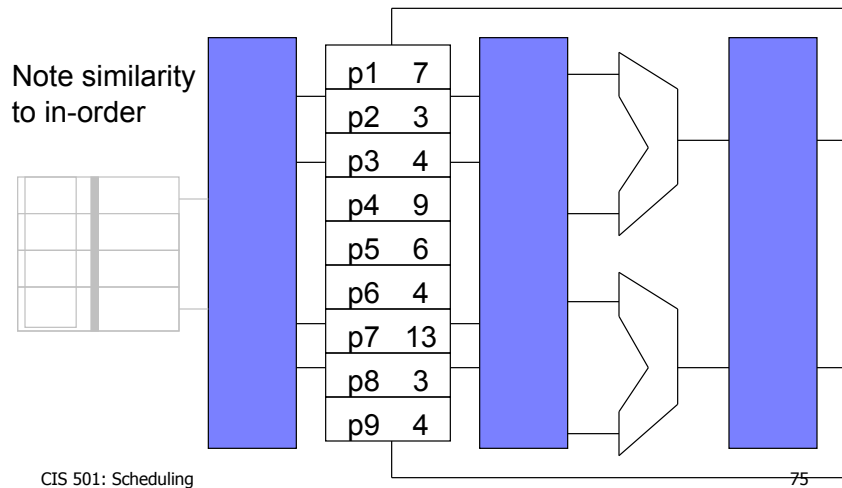
OOO execution (2-wide)



OOO execution (2-wide)



OOO execution (2-wide)



Multi-cycle operations

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

Re-order Buffer (ROB)

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

Renaming revisited

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

Renaming example

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

Renaming example

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

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

\longrightarrow
 $\text{xor } p1 \wedge p2 \rightarrow p6$ [p3]
 $\text{add } p6 + p4 \rightarrow$ [p4]
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map table

p7
p8
p9
p10

Free-list

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

\longrightarrow
 $\text{xor } p1 \wedge p2 \rightarrow p6$ [p3]
 $\text{add } p6 + p4 \rightarrow$ [p4]
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map table

p7
p8
p9
p10

Free-list

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

\longrightarrow
 $\text{xor } p1 \wedge p2 \rightarrow p6$ [p3]
 $\text{add } p6 + p4 \rightarrow p7$ [p4]
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map table

p8
p9
p10

Free-list

Renaming example

$\text{xor } r1 \wedge r2 \rightarrow r3$
 $\text{add } r3 + r4 \rightarrow r4$
 $\text{sub } r5 - r2 \rightarrow r3$
 $\text{addi } r3 + 1 \rightarrow r1$

\longrightarrow
 $\text{xor } p1 \wedge p2 \rightarrow p6$ [p3]
 $\text{add } p6 + p4 \rightarrow p7$ [p4]
 $\text{sub } p5 - p2 \rightarrow$ [p6]
 $\text{addi } r3 + 1 \rightarrow r1$

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map table

p8
p9
p10

Free-list

Recovery

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

Recovery example

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]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table



Free-list

Recovery example

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]

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map table



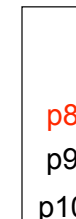
Free-list

Recovery example

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]

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map table



Free-list

Recovery example

bnz r1 loop bnz p1, loop []
 xor r1 ^ r2 -> r3 xor p1 ^ p2 -> p6 [p3]
 add r3 + r4 -> r4 add p6 + p4 -> p7 [p4]

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map table

p7
p8
p9
p10

Free-list

Recovery example

bnz r1 loop bnz p1, loop []
 xor r1 ^ r2 -> r3 xor p1 ^ p2 -> p6 [p3]

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

p6
p7
p8
p9
p10

Free-list

Recovery example

bnz r1 loop bnz p1, loop []

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

p6
p7
p8
p9
p10

Free-list

What about stores

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

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

Commit

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]

- Commit: instruction becomes **architected state**
 - In-order, only when instructions are finished
 - Free overwritten register (why?)

Freeing over-written register

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]

- P3 was r3 **before** xor
- P6 is r3 **after** xor
 - Anything older than xor should read p3
 - Anything younger than xor should p6 (until next r3 writing instruction)
- At commit of xor, no older instructions exist

Commit Example

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]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table



Free-list

Commit Example

xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

xor p1 ^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9

[p3]
[p4]
[p6]
[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3

Free-list

CIS 501: Scheduling

101

Commit Example

add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9

[p4]
[p6]
[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3
p4

Free-list

CIS 501: Scheduling

102

Commit Example

sub r5 - r2 -> r3
addi r3 + 1 -> r1

sub p5 - p2 -> p8
addi p8 + 1 -> p9

[p6]
[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3
p4
p6

Free-list

CIS 501: Scheduling

103

Commit Example

addi r3 + 1 -> r1

addi p8 + 1 -> p9

[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3
p4
p6
p1

Free-list

CIS 501: Scheduling

104

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

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

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 ld and add

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 1:

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

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 and 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

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:

- Commit xor and ld (2-wide: can do both at once)

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?

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?

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 != 7

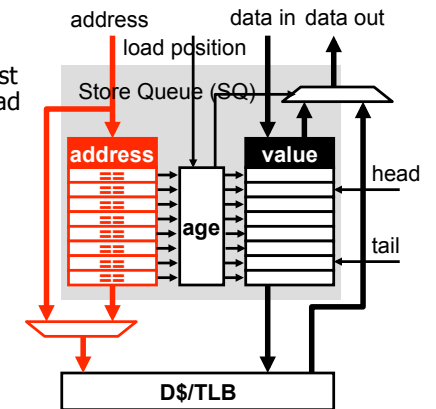
Can ld execute now?

Forwarding

- Stores write cache at commit
 - Commit is in-order, delayed by all instructions
- Loads read cache
 - But execution is critical
- Forwarding
 - Allow store -> load communication before store commit
 - Conceptually like bypassing, but very different implementation

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 committed) 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
ld [p1] -> p5	1			
ld [p2] -> p6	1			
add p5 + p6 -> p7				
st p7 -> [p3]				
ld 4[p1] -> p8				
ld 4[p2] -> p9				
add p8 + p9 -> p4				
st p4 -> 4[p3]				

Suppose 2 wide, conservative scheduling. May issue 1 load per cycle. Loads take 3 cycles to complete.

Our example from before

	Disp	Issue	WB	Commit
ld [p1] -> p5	1	2	5	
ld [p2] -> p6	1			
add p5 + p6 -> p7	2			
st p7 -> [p3]	2			
ld 4[p1] -> p8				
ld 4[p2] -> p9				
add p8 + p9 -> p4				
st p4 -> 4[p3]				

Our example from before

	Disp	Issue	WB	Commit
ld [p1] -> p5	1	2	5	
ld [p2] -> p6	1	3	6	
add p5 + p6 -> p7	2			
st p7 -> [p3]	2			
ld 4[p1] -> p8	3			
ld 4[p2] -> p9	3			
add p8 + p9 -> p4				
st p4 -> 4[p3]				

Our example from before

	Disp	Issue	WB	Commit
ld [p1] -> p5	1	2	5	
ld [p2] -> p6	1	3	6	
add p5 + p6 -> p7	2			
st p7 -> [p3]	2			
ld 4[p1] -> p8	3			
ld 4[p2] -> p9	3			
add p8 + p9 -> p4	4			
st p4 -> 4[p3]	4			

Conservative load scheduling: can't issue ld4[p1]->p8

Our example from before

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

Our example from before

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

Our example from before

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

Our example from before

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

Our example from before

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

Our example from before

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

Our example from before

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

Our example from before

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

Our 2-wide ooo processor may as well be 1-wide in-order!

Our example from before

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

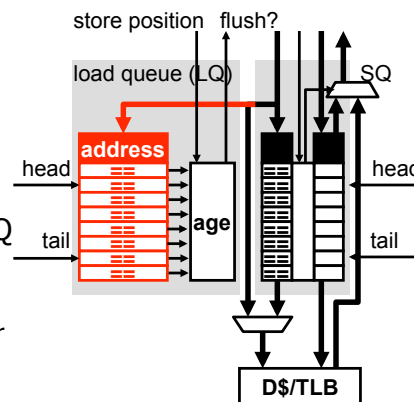
- It would be nice if we could issue ld 4[p1]->p8 in c4.
 - Can we speculate and issue it then?

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 load ordering violations
- Load execution: Write address into LQ
 - Also note any store forwarded from
- Store execution: Search LQ
 - Younger load with same addr?
 - Didn't forward from younger store?



Store Queue + Load Queue

- Store Queue: handles forwarding
 - Written by stores
 - Searched by loads
- Load Queue: detects ordering violations
 - Written by loads
 - Searched by stores
- Both together
 - Allows aggressive load scheduling
 - Stores don't constrain load execution

Our example from before

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

- Aggressive load scheduling?
 - Issue ld 4[p1]->p8 in cycle 4

Our example from before

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

Our example from before

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

Saves 4 cycles over conservative
Actually uses ooo-ness

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?

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 is not so great
- More complex predictors used in practice
 - Predict which stores loads should wait for

Out of Order: Window Size

- Scheduling scope = ooo window size
 - Larger = better
 - Constrained by physical registers
 - 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

OOO scalability research

- Checkpoint Processing and Recovery [Akkary '03]
 - Attacks scaling of register file
 - Take checkpoints at rename
 - Only recover to those
 - Free Pregs aggressively
- Continual Flow Pipelines [Srinivasan '04]
 - Attacks scaling of Issue Queue
 - Put L2 misses and dependents out of IQ
 - Place back in when L2 miss returns
- Store Vulnerability Window [Roth '05] + Store Queue Index Prediction [Sha '05]
 - Scalable (non-associative) load/store queues
 - Predict store queue index for forwarding
 - Filtered load re-execution prior to commit

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

Dependence types

- RAW (Read After Write) = "true dependence"
ld [r1] -> r2
add r2 + r3 -> r4
- WAW (Write After Write) = "output dependence"
ld [r1] -> r2
add r1 + r3 -> r2
- WAR (Write After Read) = "anti-dependence"
ld [r1] -> r2
add r3 + r4 -> r1

Memory dependences

- RAW (Read After Write)
st r1 -> [r2]
ld [r2] -> r4
- WAW (Write After Write)
st r1 -> [r2]
st r3 -> [r2]
- WAR (Write After Read)
ld [r1] -> r2
st r3 -> [r1]

More on dependences

- RAW
 - When more than one applies, RAW dominates:
add r1 + r2 -> r3
addi r3 + 1 -> r3
 - Must be respected: no trick to avoid
- WAR/WAW on registers
 - Two things happen to use same name
 - Can be eliminated by renaming
- WAR/WAW on memory
 - Can't rename memory
 - Need to use other tricks (later this lecture)

Out of Order: Top 5 things to know

- Register renaming
 - How to perform is 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 LQ
- Loads
 - Conservative/aggressive/predictive scheduling
 - Violation detection