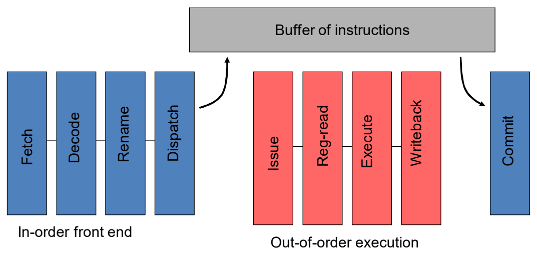
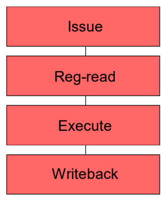
Out-of-order Pipeline



Out-of-order pipeline

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



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
```

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	

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

Issue

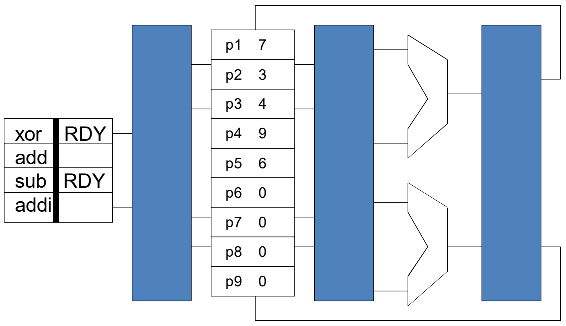
- **Select/Wakeup** one cycle
- Dependents go back to back
  - Next cycle: add/addi are ready:

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

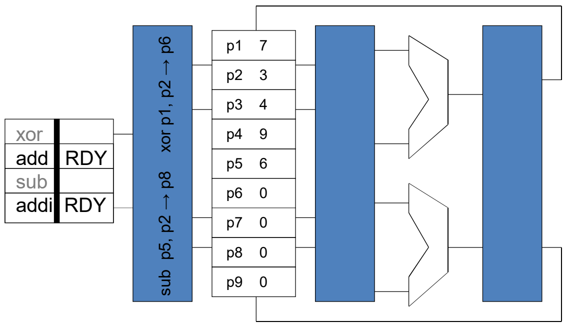
Register Read

- 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 dispatch, keep values in Issue Queue
  - Pentium Pro, Core 2, Core i7

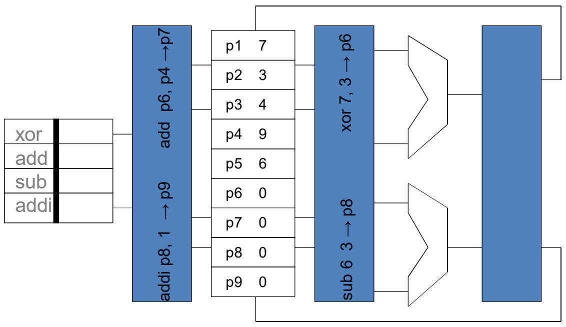
OOO execution (2-wide)



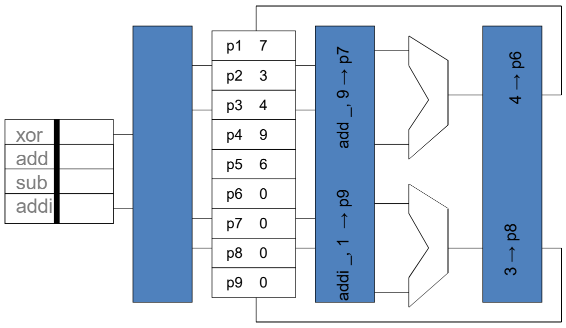
OOO execution (2-wide)



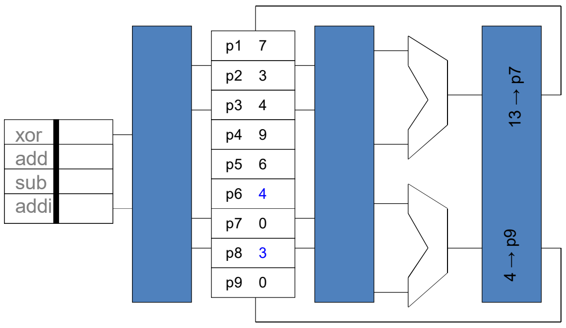
OOO execution (2-wide)



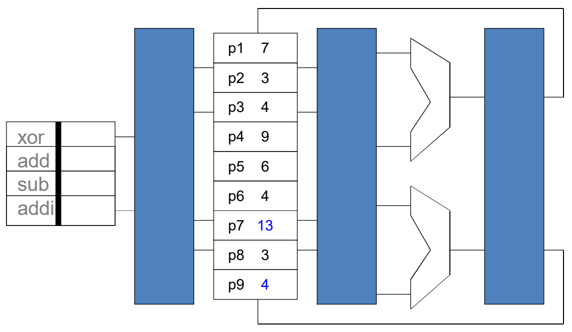
OOO execution (2-wide)



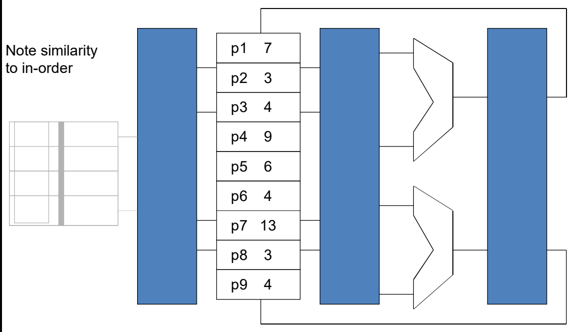
OOO execution (2-wide)



OOO execution (2-wide)



OOO execution (2-wide)



Multi-cycle operations

- Multi-cycle ops (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

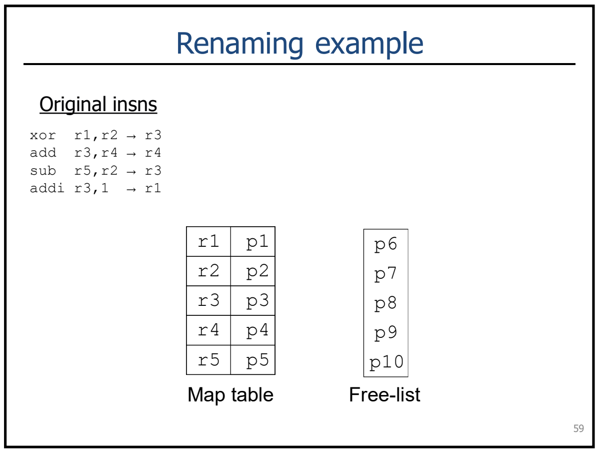
Re-order Buffer (ROB)

- 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

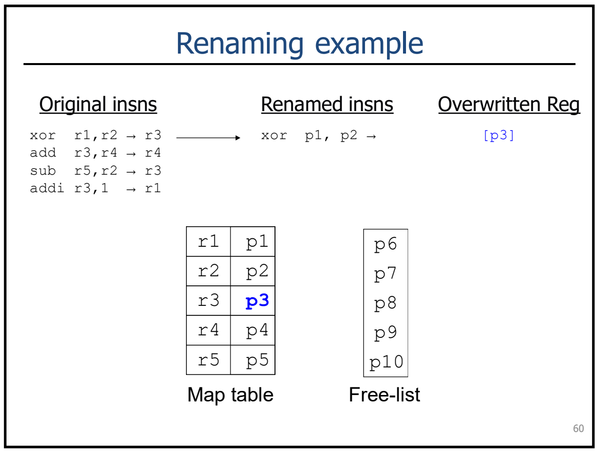
```
xor  r1,r2 → r3
add  r3,r4 → r4
sub  r5,r2 → r3
addi r3,1  → r1
```

Map table

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Free-list

p6
p7
p8
p9
p10



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 →
```

Overwritten Reg

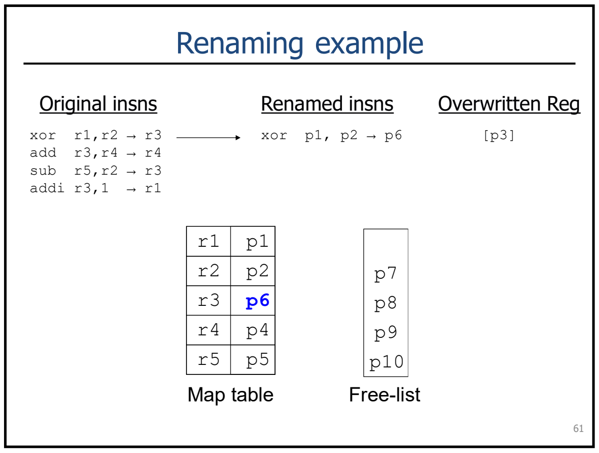
```
[p3]
```

Map table

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

Free-list

p6
p7
p8
p9
p10



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
```

Overwritten Reg

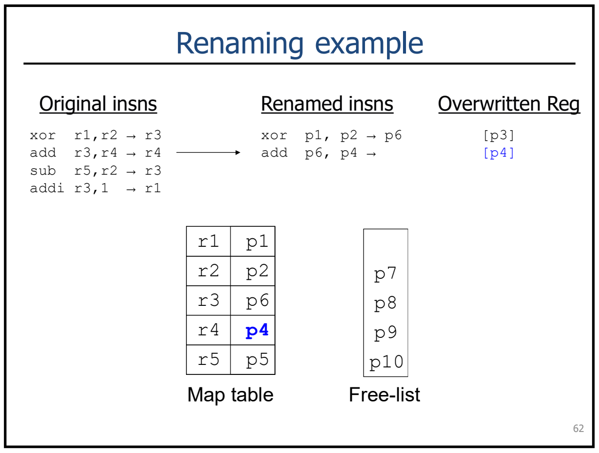
```
[p3]
```

Map table

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

Free-list

p7
p8
p9
p10



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 →
```

Overwritten Reg

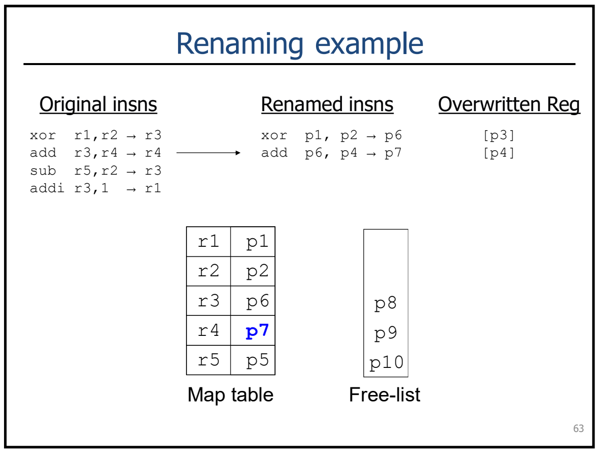
```
[p3]
[p4]
```

Map table

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

Free-list

p7
p8
p9
p10



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
```

Overwritten Reg

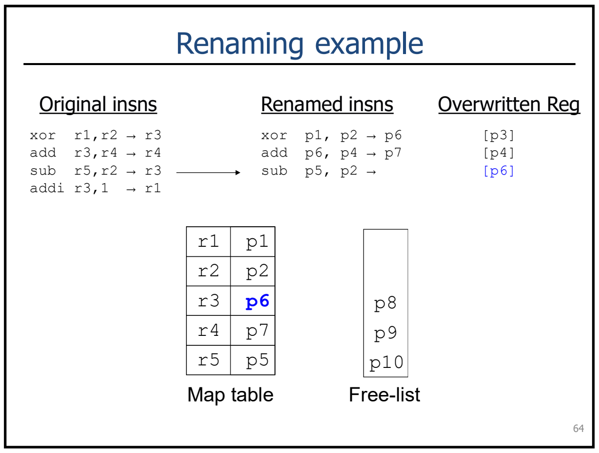
```
[p3]
[p4]
```

Map table

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

Free-list

p8
p9
p10



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 →
```

Overwritten Reg

```
[p3]
[p4]
[p6]
```

Map table

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

Free-list

p8
p9
p10

Renaming example

<u>Original insns</u>

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

<u>Renamed insns</u>

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

<u>Overwritten Reg</u>

```
[p3]
[p4]
[p6]
```

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

Map table

p9
p10

Free-list

Renaming example

<u>Original insns</u>

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

<u>Renamed insns</u>

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

<u>Overwritten Reg</u>

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

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

Map table

p9
p10

Free-list

Renaming example

<u>Original insns</u>

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

<u>Renamed insns</u>

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

<u>Overwritten Reg</u>

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

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

Map table

p10

Free-list

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

Recovery

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

Recovery example

<u>Original insns</u>

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

<u>Renamed insns</u>

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

<u>Overwritten Reg</u>

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

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10

Free-list

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]

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

Map table

p9
p10

Free-list

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]

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

Map table

p8
p9
p10

Free-list

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]

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

Map table

p7
p8
p9
p10

Free-list

Recovery example

Original insns	Renamed insns	Overwritten Reg
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

Original insns	Renamed insns	Overwritten Reg
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?

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

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?)

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!

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

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

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

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

Commit Example

Original insns	Renamed insns	Overwritten Reg
`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

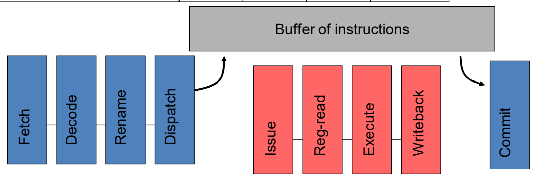
Free-list

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`				



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 1st 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 2:
- Dispatch xor and 2nd 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 & 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)

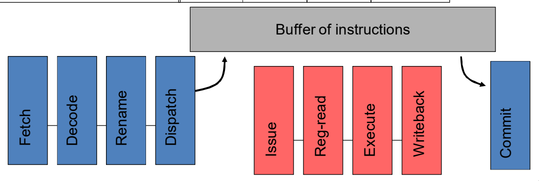
Out of order pipeline diagrams

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

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)

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

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

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

Forwarding: Store Queue

Store Queue

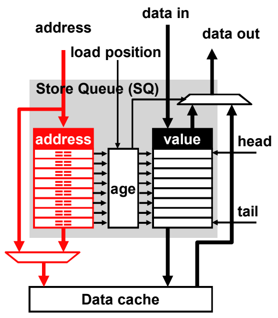
- Holds all in-flight stores
- CAM: searchable by address
- Age logic: determine youngest matching store older than load

Store execution

- Write Store Queue
  - Address + Data

Load execution

- Search SQ
  - Match? Forward
- Read D$


data in
address
data out
load position
Store Queue (SQ)
address
value
age
head
tail
Data cache