
CSE 530

Fundamentals of Computer Architecture

Spring 2021

Superscalar Out of Order Datapaths

John (Jack) Sampson (cse.psu.edu/~sampson)

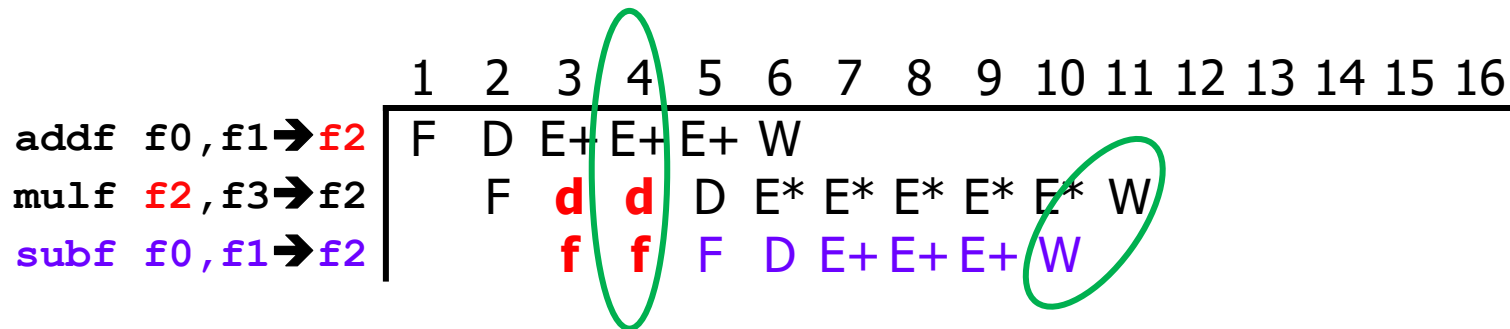
Course material on Canvas

[Adapted in part from Mary Jane Irwin, V. Narayanan, Amir Roth, Milo Martin,
and others]

In-order superscalar (static schedule) limitations

- ❑ Limited number of registers (set by the ISA)
- ❑ Several N^2 hardware issues: dependence cross-checking (e.g., load-use) logic, bypass (forwarding) logic, register file read/write ports
- ❑ Limited compiler scheduling scope
 - ❑ Example: can't generally move memory operations past branches
 - ❑ Inexact memory aliasing information often prevents reordering of loads above stores (don't know the memory address yet)
- ❑ Caches misses (or any runtime event) confound compiler scheduling
 - ❑ How can the compiler know which loads will miss vs hit?

The problem with in-order pipelines



❑ What's happening in cycle 4?

❑ **mulf** stalls due to RAW **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

❑ Does it maintain in-order writes to the register file?

- ❑ If we allow **subf** to proceed (after D in cycle 6) then \$f2 ends up with the wrong value (the value from from **mulf**, not **subf**)

Can hardware overcome these limitations?

❑ Yes! With dynamically-scheduled processors

- ❑ Also called “out-of-order” (OoO) processors
- ❑ Hardware re-schedules instructions ...
 - ...within a sliding window of VonNeumann instructions
- ❑ As with pipelining and superscalar, the ISA is unchanged
 - Same hardware/software interface, “appearance” of in-order

❑ Increases scheduling scope

- ❑ Does loop unrolling transparently (by the hardware)
 - Uses branch prediction to “unroll” branches

❑ Examples:

- ❑ Pentium Pro/II/III (3-wide), Core 2/Nehalem/I-(3/5/7) (4-wide), Alpha 21264 (4-wide), MIPS R10000 (4-wide), Power5 (5-wide)
- ❑ Even mobile (ARMv8) chips with decode at 2-3, and up to 6 micro-ops wide

Data dependence types

- ❑ To exploit ILP, must determine which instructions can be executed in parallel (without any stalls)
 - ❑ *Must* preserve (external appearance of) **program order**

- ❑ RAW (Read After Write) = “true dependence”

ld [r1] → r2

add r2 + r3 → r4

ld [r2] → r1

st r1 → [r3]

- ❑ WAW (Write After Write) = “output dependence”

ld [r1] → r2

add r1 + r3 → r2

st r1 → [r2]

st r3 → [r2]

- ❑ WAR (Write After Read) = “anti-dependence”

ld [r1] → r2

add r3 + r4 → r1

ld [r1] → r2

st r3 → [r1]

More on data 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 different things can happen when using the same name depending on instruction ordering
- ❑ Can be eliminated by **register renaming**

❑ WAR/WAW on memory

- ❑ Can't rename memory and don't know if there is an actual dependency until the effective address is known (in X)
- ❑ Need to use other tricks (later)

Control (branch) dependence

- Every instruction is control dependent on some set of branches

```
if p1 {  
    S1;  
};  
if p2 {  
    S2;  
}
```

S1 is control dependent on p1, and S2
is control dependent on p2 but not on
p1

- But control dependence need not be preserved ...
 - **If** we can do so (e.g., using branch prediction) without affecting the correctness of the program. We can execute instructions that should **not** have been executed (i.e., the prediction is incorrect), thereby violating the control dependences, as long as we **don't change the visible machine state**.

Exception dependence

- ❑ Two properties critical to program correctness are
 - ❑ Maintaining correct data flow (RAW dependencies)
 - ❑ Guaranteeing that mispredicted branches don't change the machine state
 - ❑ Providing precise exceptions (to support virtual memory)
- ❑ Preserving exception behavior \Rightarrow any changes in instruction execution order must not change the order in which exceptions are raised, or cause new exceptions to be raised in the program

- ❑ Example:

```
daddu r3, r4 → r2
beqz  r2, L1
lw    (r2) → r1
L1:
```

- ❑ Can there be a problem with moving `lw` before `beqz`?

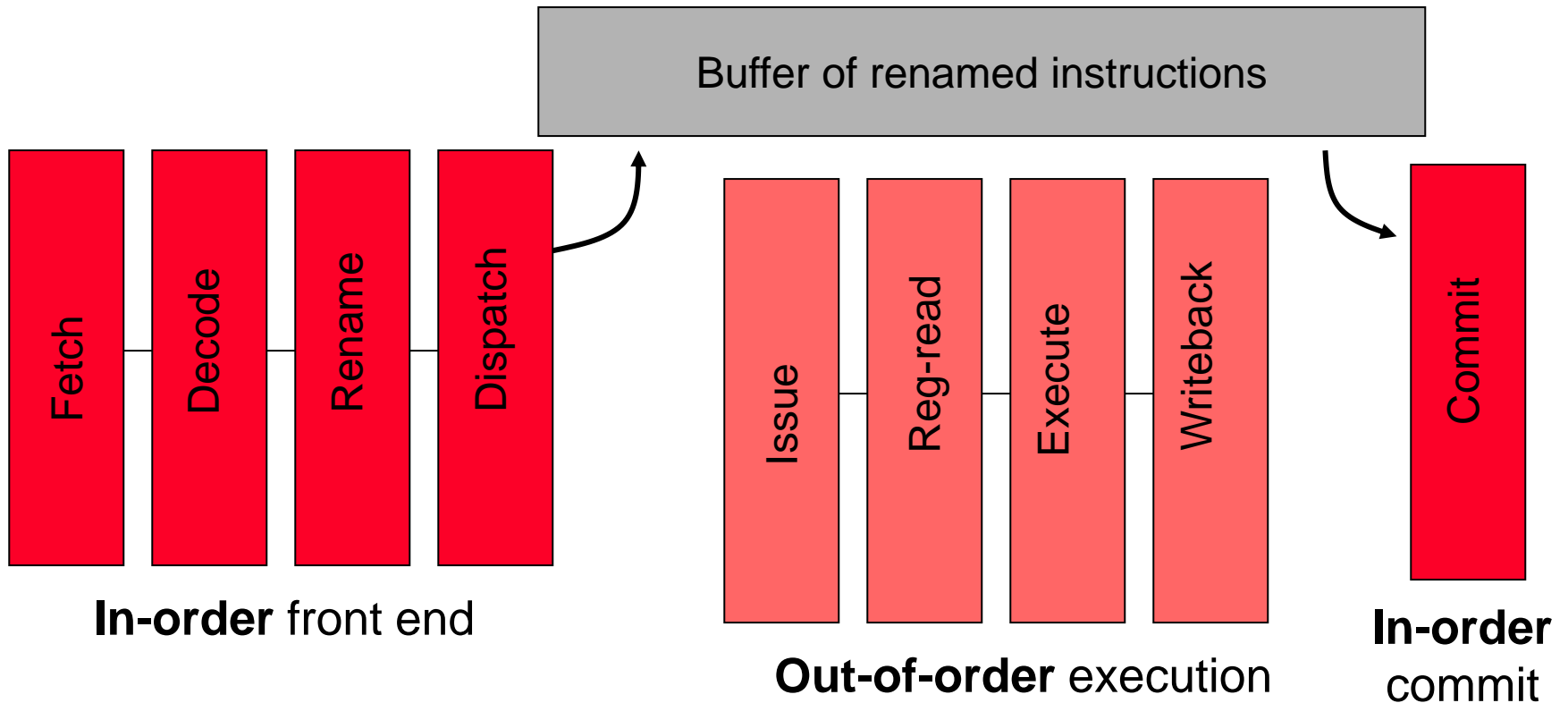
Early dynamic scheduling (OoO) approaches

- ❑ Scoreboarding (HP Appendix A.7) – CDC 6600 (Thornton) first publication in 1964
 - ❑ Used **centralized** hazard detection logic (scoreboard) to support OoO execution. Instr's are stalled when their FU is busy, for RAW dependencies, *and for WAW and WAR dependencies*.
- ❑ Tomasulo (HP 2.4) – IBM 360/91 (Tomasulo) first publication in 1967
 - ❑ Used **distributed** hazard detection logic (reservation stations feeding each FU) to support OoO execution with *register renaming* that eliminates WAW and WAR dependencies; distributes results from FUs to reservation stations on a Common Data Bus (potential bottleneck)
 - ❑ Writes results to register file and memory when instr's complete – possibly out-of-order – so *can not support precise exceptions or speculative execution* (e.g., branch speculation)
 - ❑ <http://www.ecs.umass.edu/ece/koren/architecture/Tomasulo1/tomasulo.htm>

Modern dynamic OoO “approaches”

- ❑ HPS – (Patt, Hwu, Shebanow) first publication in 1985
 - ❑ Uses a register alias table and distributed node alias tables that feed each FUs (essentially reservation stations) to support OoO execution with **register renaming**; distributes results from FUs to reservation stations on multiple distribution buses (one per FU)
 - ❑ Supports precise exceptions and speculative execution with a checkpoint repair mechanism
- ❑ RUU – (Sohi) first publication in 1987
 - ❑ Uses a centralized Register Update Unit (RUU) that 1) receives new instr's from decode, 2) renames registers, 3) monitors the (single) result bus to resolve dependencies, 4) determines when instr's are ready to issue (send for execution), and 5) holds completed instr's until they can **commit**
 - ❑ Supports precise exceptions and speculative execution with **in-order commit** out of the RUU
 - ❑ Basis of SimpleScalar's architecture

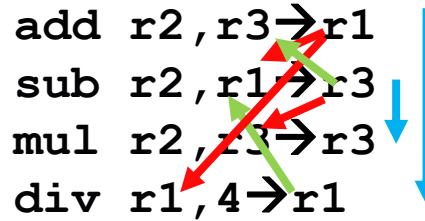
Out-of-order pipeline – the simple view



Code example

Code: Raw instr's

```
add r2, r3 → r1
sub r2, r1 → r3
mul r2, r3 → r3
div r1, 4 → r1
```



RAW

WAR

WAW

- ❑ “True” (red) & “False” (artificial) dependencies
- ❑ Divide instr independent of subtract and multiply instr's
 - ❑ 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) dynamically schedule

Step #1: Register renaming

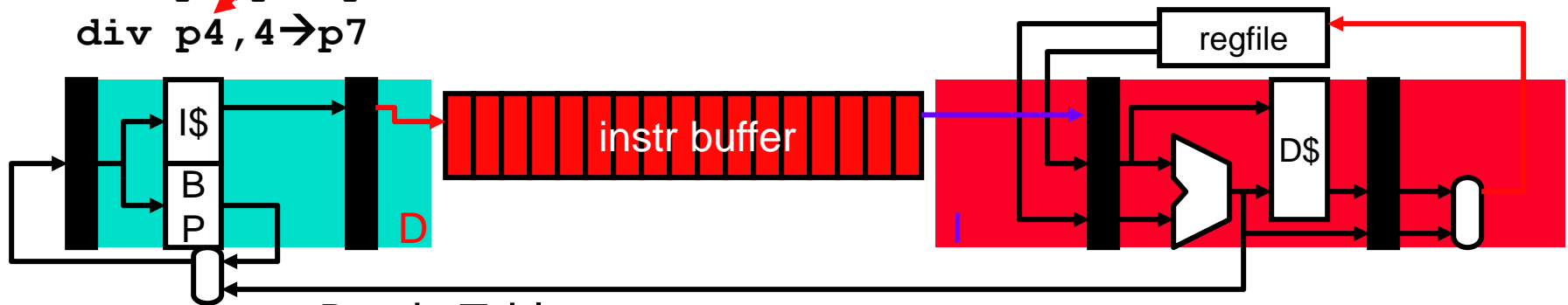
- ❑ To eliminate (WAW, WAR) register conflicts/hazards
- ❑ “Architected” vs “Physical” registers – level of indirection
 - ❑ Architected (ISA) register names: $r1, r2, r3$
 - ❑ Physical register locations: $p1, p2, p3, p4, p5, p6, p7$
 - ❑ Original mapping: $r1 \rightarrow p1, r2 \rightarrow p2, r3 \rightarrow p3$, $p4-p7$ are “available”

MapTable			Free List	Original instr's	Renamed instr's
$r1$	$r2$	$r3$			
$p1$	$p2$	$p3$	$p4, p5, p6, p7$	add $r2, r3 \rightarrow r1$	add $p2, p3 \rightarrow p4$
$p4$	$p2$	$p3$	$p5, p6, p7$	sub $r2, r1 \rightarrow r3$	sub $p2, p4 \rightarrow p5$
$p4$	$p2$	$p5$	$p6, p7$	mul $r2, r3 \rightarrow r3$	mul $p2, p5 \rightarrow p6$
$p4$	$p2$	$p6$	$p7$	div $r1, 4 \rightarrow r1$	div $p4, 4 \rightarrow p7$

- ❑ Renaming – conceptually write each register once
 - + Removes **false** dependences (WAW and WAR)
 - + Leaves **true** dependences (RAW) intact!
- ❑ When to *reuse* a physical register? After overwriting instr commits.

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

- ❑ Instr's fetch/decoded/renamed go into an *Instruction Buffer*
 - ❑ Also called "instruction window" or "instruction scheduler"
- ❑ Instr's (conceptually) check Ready Table bits every cycle
 - ❑ Execute when ready

REGISTER RENAMING

Register renaming algorithm

❑ Data structures:

- ❑ `maptable[architectural_reg] → physical_reg`
- ❑ Free list: get/put free register

❑ Algorithm: at Rename for each instruction:

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

❑ At Commit

- ❑ Once all older instructions have committed, copy physical dst register to architectural register and free physical register
`put_free_phys_reg(instr.phys_to_free)`

Which register to free at Commit ?

- ❑ The **over-written** register
 - ❑ Freed at Commit (i.e., added back to the Free List)
 - ❑ Also restored in the Map Table on **recovery**
 - Branch mis-prediction, exception recovery
- So also must be kept track of at Rename

Renaming example, 0

```
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, 1

Over-written 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

Renaming example, 2

Over-written Reg

`xor r1 ^ r2 → r3`
`add r3 + r4 → r4` → `xor p1 ^ p2 → p6` `[p3]`
`sub r5 - r2 → r3` `add p6 + p4 → p7` `[p4]`
`addi r3 + 1 → r1`

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map Table

p7
p8
p9
p10

Free List

Renaming example, 3

Over-written 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			

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map Table

p8
p9
p10

Free List

Renaming example, 4

Over-written Reg

```

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

p9
p10

Free List

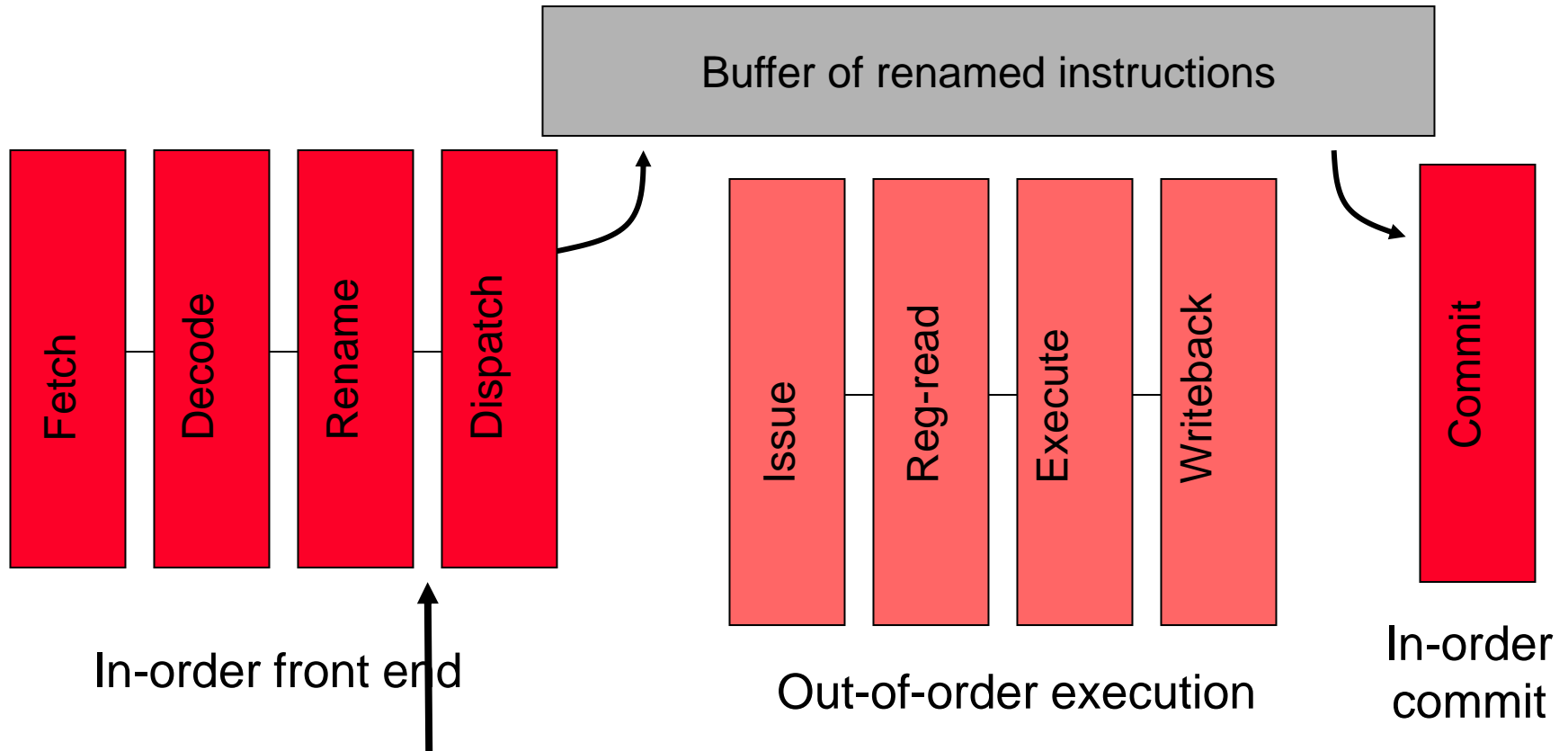
Freeing over-written registers

Over-written 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]

- ❑ p3 was r3 **before** xor
- ❑ p6 is r3 **after** xor
 - ❑ Anything older (earlier in the code) than xor should read p3
 - ❑ Anything younger (later in the code) than xor should read p6 (until next r3 writing instruction)
- ❑ At Commit of xor, no older instructions exist, so free p3!

Out-of-order pipeline – the simple view



Instr's now have unique register names, so
can now put into OoO execution structures

DYNAMIC SCHEDULING

Dispatch

- ❑ Renamed instructions placed into OoO structures

1. Issue Queue (IQ) (SimpleScalar's RUU)

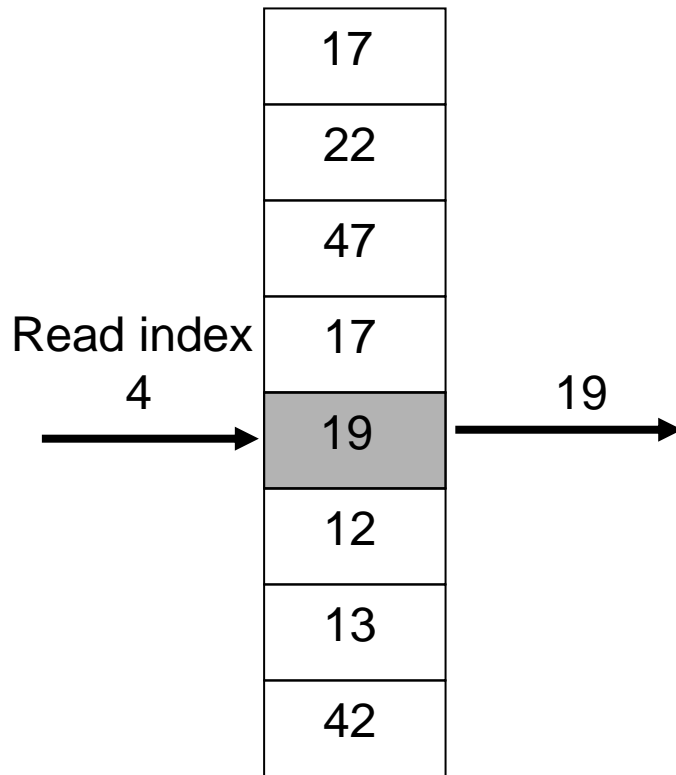
- ❑ Holds un-executed instructions
- ❑ Central piece of scheduling logic
- ❑ Accessible as both a RAM and a CAM (Content Addressable Memory)

2. Re-order buffer (ROB) (also folded into SimpleScalar's RUU)

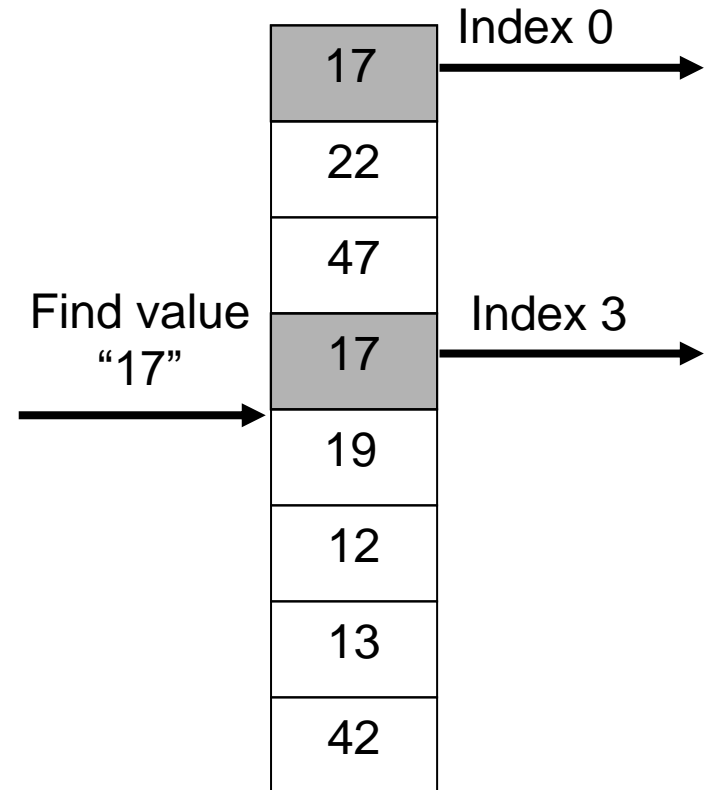
- ❑ Holds all instruction until Commit

RAM vs CAM

RAM: read/write specific index



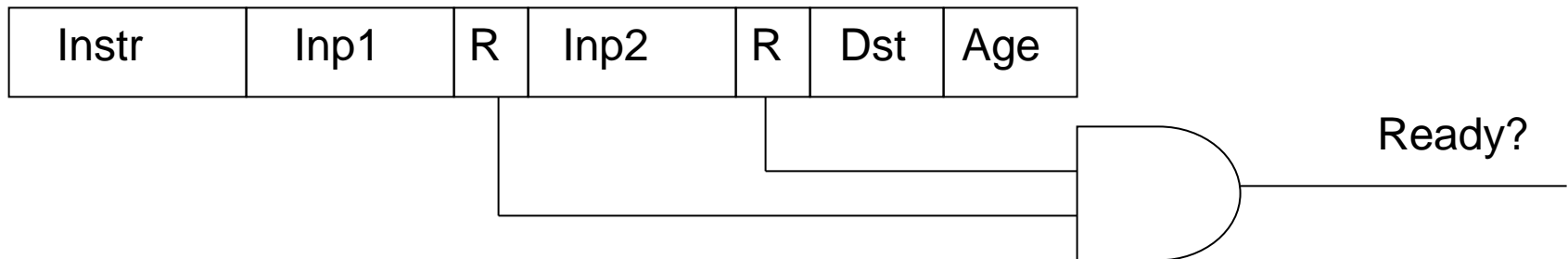
CAM: search for value



❑ One structure can have ports of both types (RAM & CAM)

Issue Queue (IQ)

- ❑ Holds un-executed instructions
 - ❑ Instruction op and instruction “age”
- ❑ Tracks ready inputs
 - ❑ Physical (renamed) source register names + ready bit
 - AND the ready bits to tell if the instruction is ready to issue
 - ❑ Physical (renamed) destination register name



Dispatch steps

- ❑ Allocate IQ (and ROB) slot
 - ❑ Full? Stall
- ❑ Read **ready bits** of inputs from Ready Table
 - ❑ Ready Table 1-bit per physical register
- ❑ Clear **ready bit** of output in Ready Table
 - ❑ Instruction has not produced value yet
- ❑ Write data in IQ slot

Dispatch example, 0

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

Issue Queue

Instr	Inp1	R	Inp2	R	Dst	Age

Ready Table

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

Dispatch example, 1

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

Issue Queue

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

Ready Table

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

Dispatch example, 2

xor p1 ^ p2 → p6

add p6 + p4 → p7

sub p5 - p2 → p8

addi p8 + 1 → p9

Ready Table

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

Issue Queue

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

Dispatch example, 3

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

Issue Queue

Instr	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 Table

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

Dispatch example, 4

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

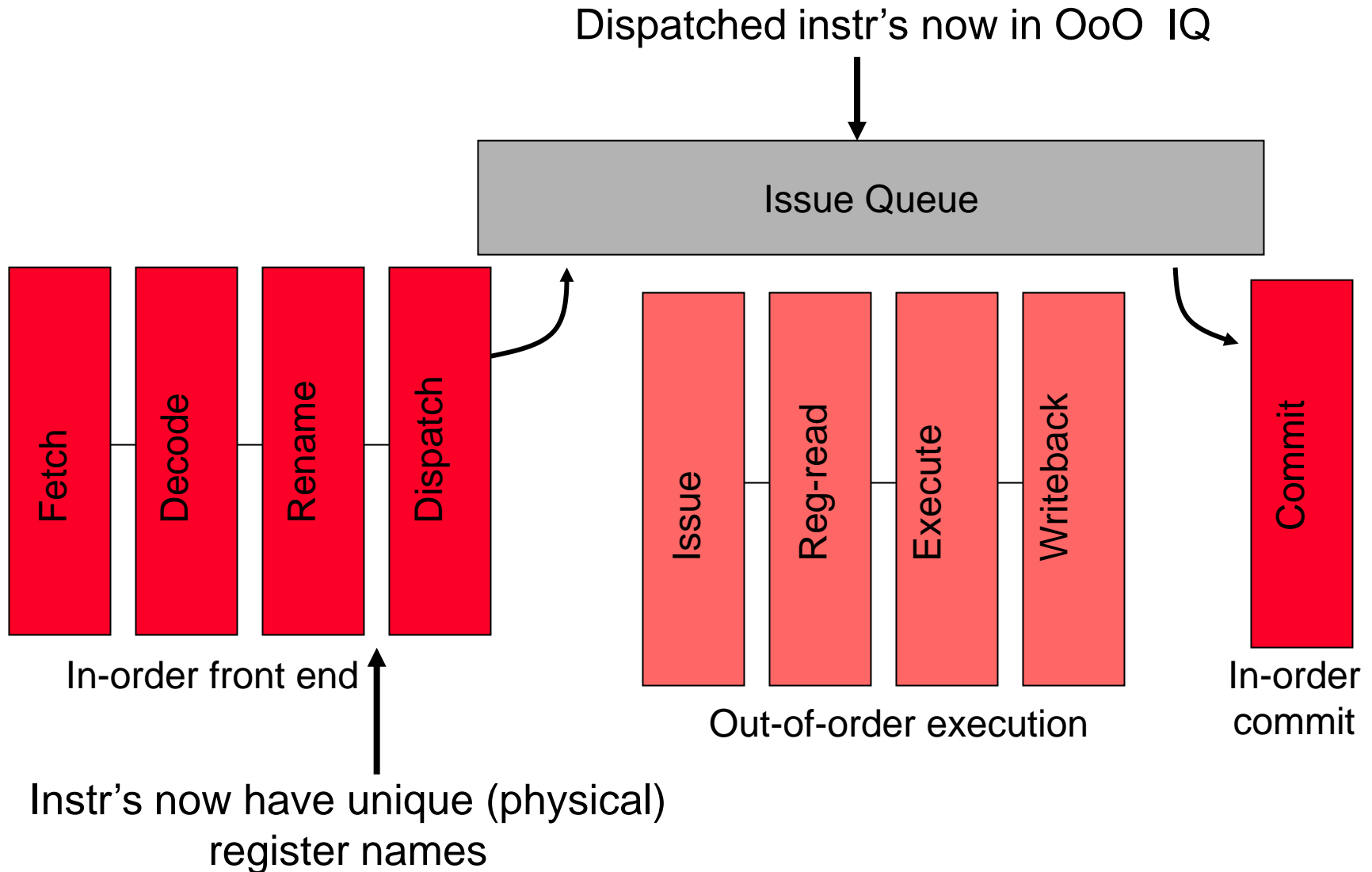
Issue Queue

Instr	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	"1"	y	p9	3

Ready Table

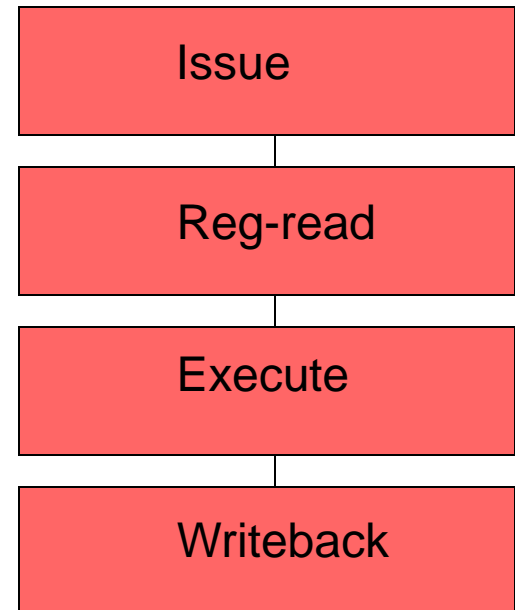
p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	n
p9	n

Out-of-order pipeline – the simple view



Out-of-order pipeline stages

- ❑ Execution (OoO) stages
- ❑ **Select** ready instructions
 - ❑ Send (Issue) them 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[instr.phys_input1] == ready &&  
        table[instr.phys_input2] == ready then  
        instr is “ready”
```

```
select the oldest “ready” instructions
```

```
    table[instr.phys_output] = ready
```

Issue = Select + Wakeup

- ❑ **Select** N oldest, ready instructions to send for execution
 - “xor” is the oldest ready instruction below
 - “xor” and “sub” are the two oldest ready instructions below
- ❑ Note: may have resource constraints and/or structural hazards blocking the issue: i.e., load/store/fp

Issue Queue

Instr	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	“1”	y	p9	3	

Issue = Select + Wakeup

❑ Wakeup dependent instructions

- ❑ CAM search for Dst in input fields
- ❑ Set ready bit on match
- ❑ Also update Ready Table bit for Dispatch of future instructions

Assoc Search for p6 and p8 Assoc Search for p6 and p8

↓ ↓

Instr	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	"1"	y	p9	3

Ready Table

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

Issue (next)

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

Issue Queue

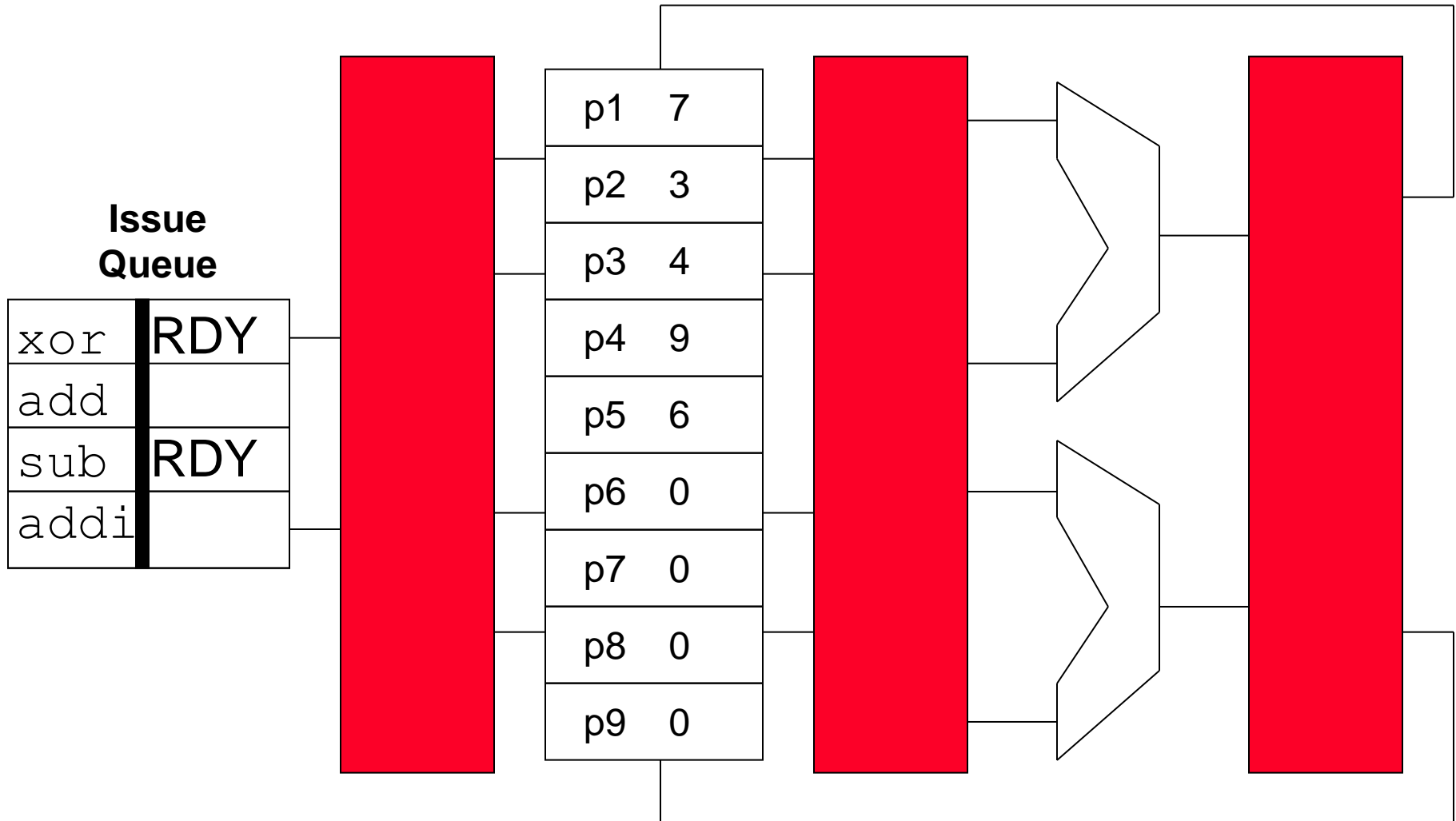
Insn	Inp1	R	Inp2	R	Dst	Age	
add	p6	y	p4	y	p7	1	Ready!
addi	p8	y	---	y	p9	3	Ready!

- ❑ Note that once the instruction is issued it is removed from the IQ

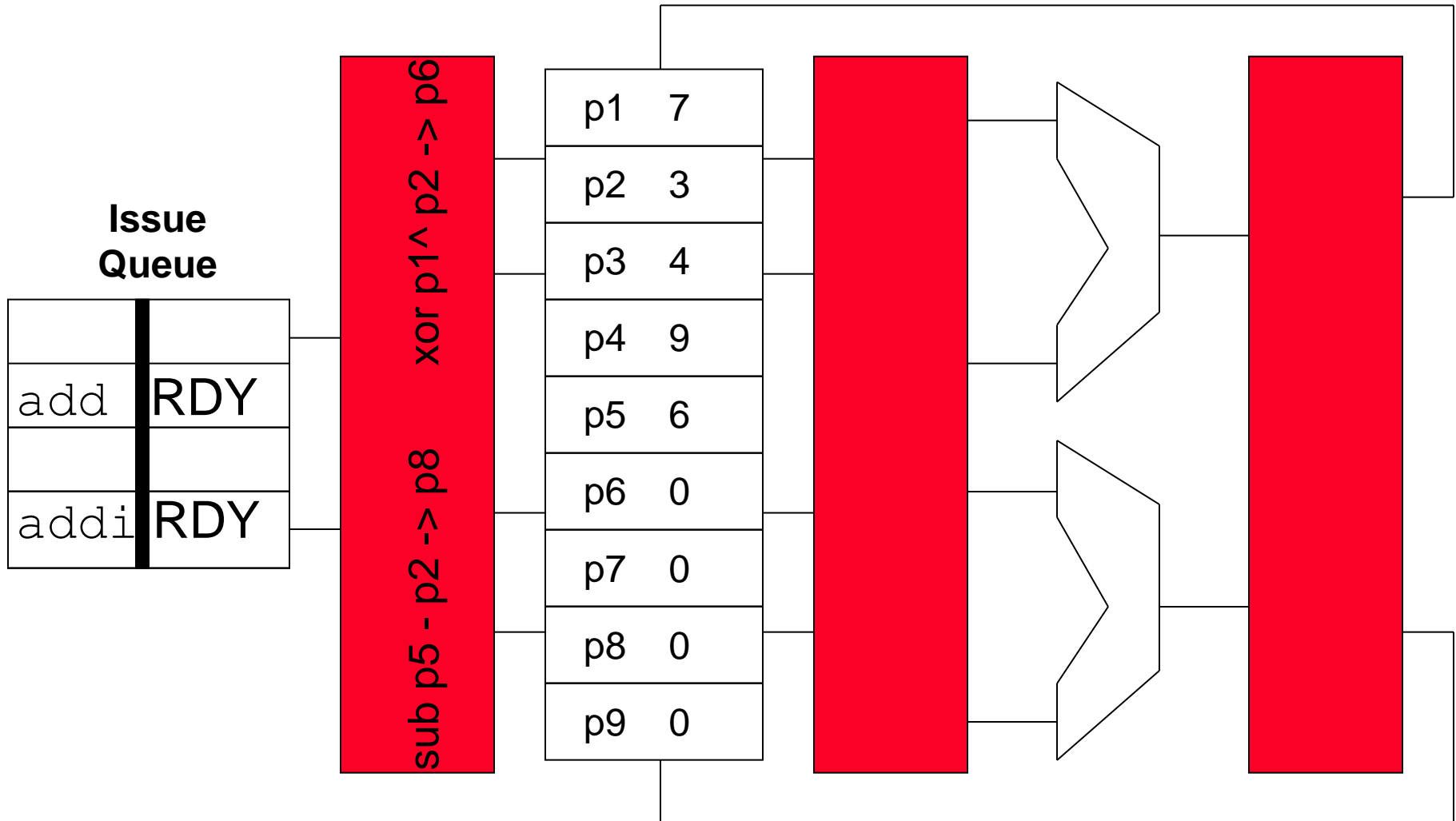
Register read

- ❑ When do instructions read the physical register file?
 - ❑ Obviously not done at decode (not renamed yet)
- ❑ Option #1: after Issue (Select), right before Execute
 - ❑ Read **physical** (renamed) register
 - ❑ Or get value via bypassing (based on physical register name)
 - ❑ Pentium 4, MIPS R10k, Alpha 21264 style
- ❑ Physical register file may be large
 - ❑ Could be a multi-cycle read
- ❑ Option #2: as part of Dispatch, keep actual data values in the IQ (along with the regaddr for the Issue (Wakeup) associative search)
 - ❑ Means bigger IQ entries (+32b or 64b per source value)
 - ❑ Pentium Pro, Core 2, Core i7

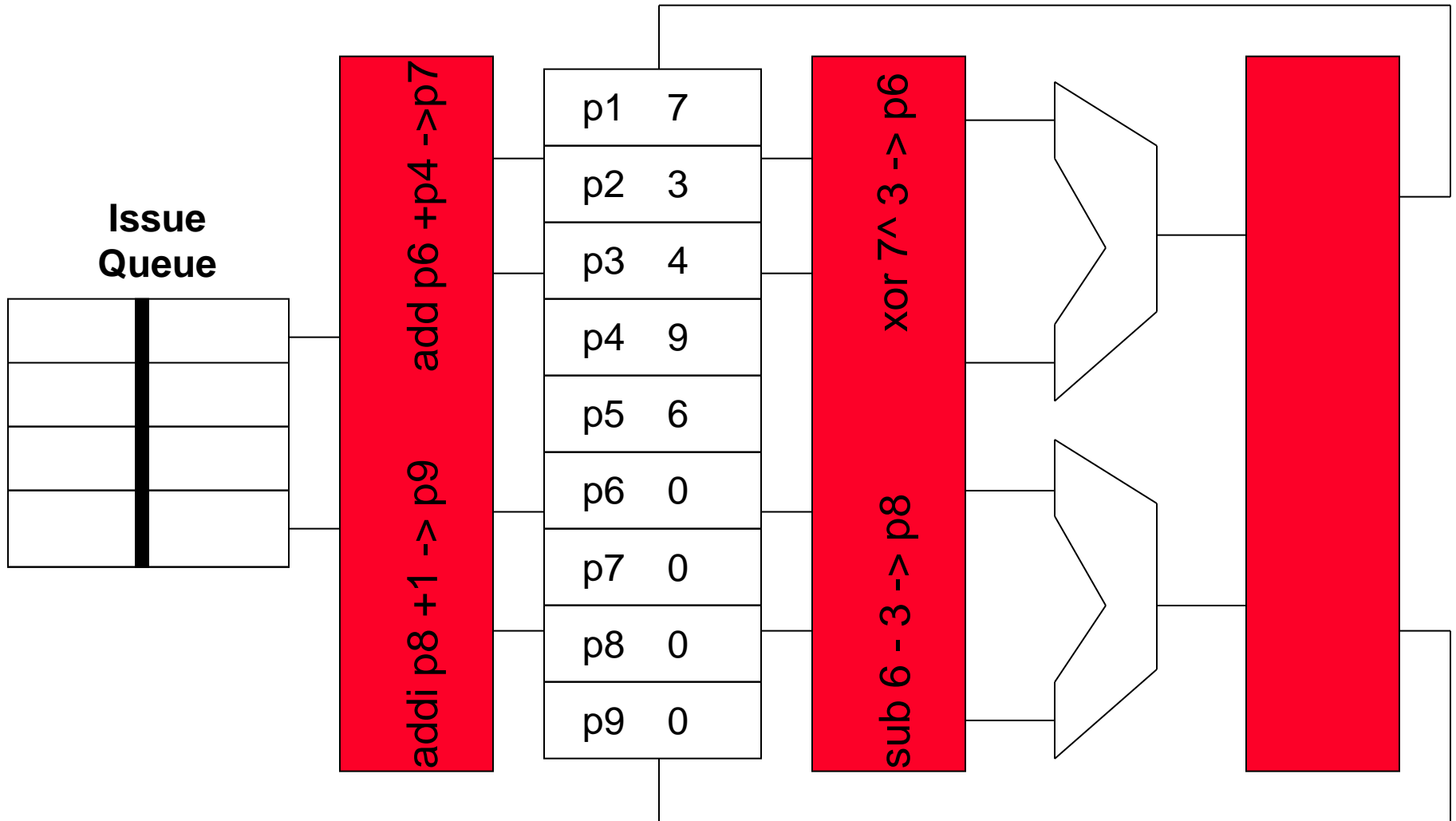
OoO execution (2-wide), 0



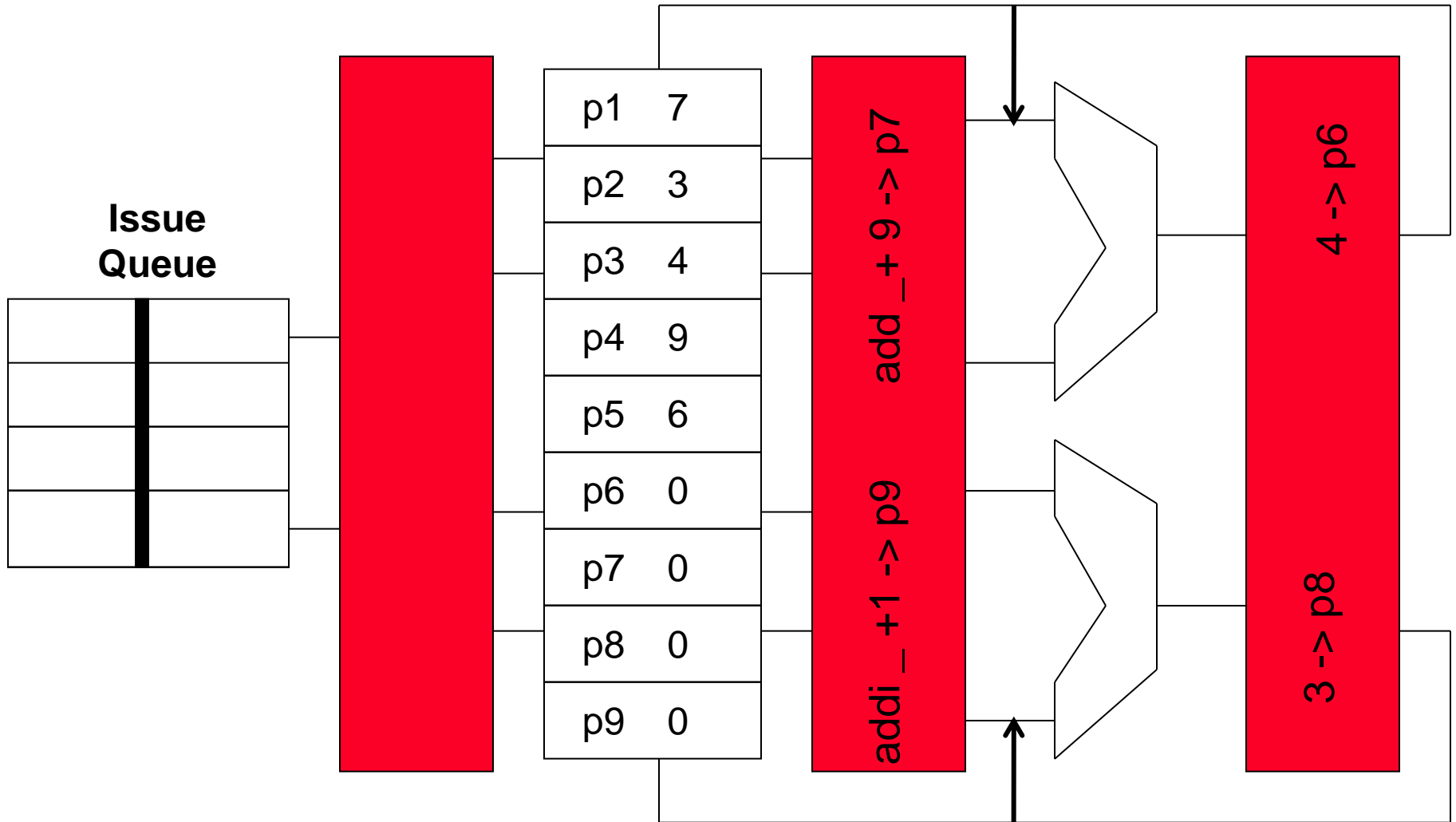
OoO execution (2-wide), 1



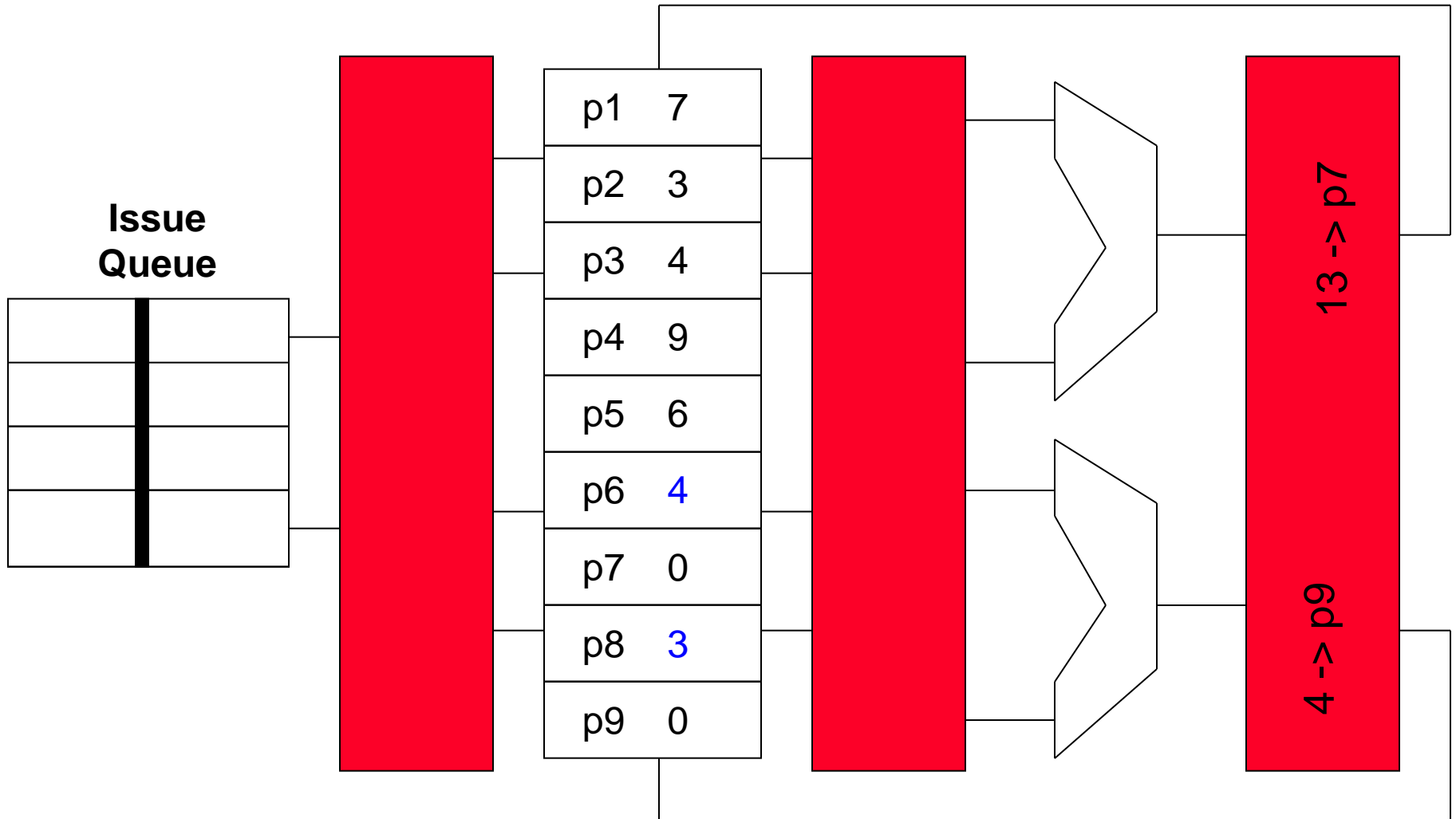
OoO execution (2-wide), 2



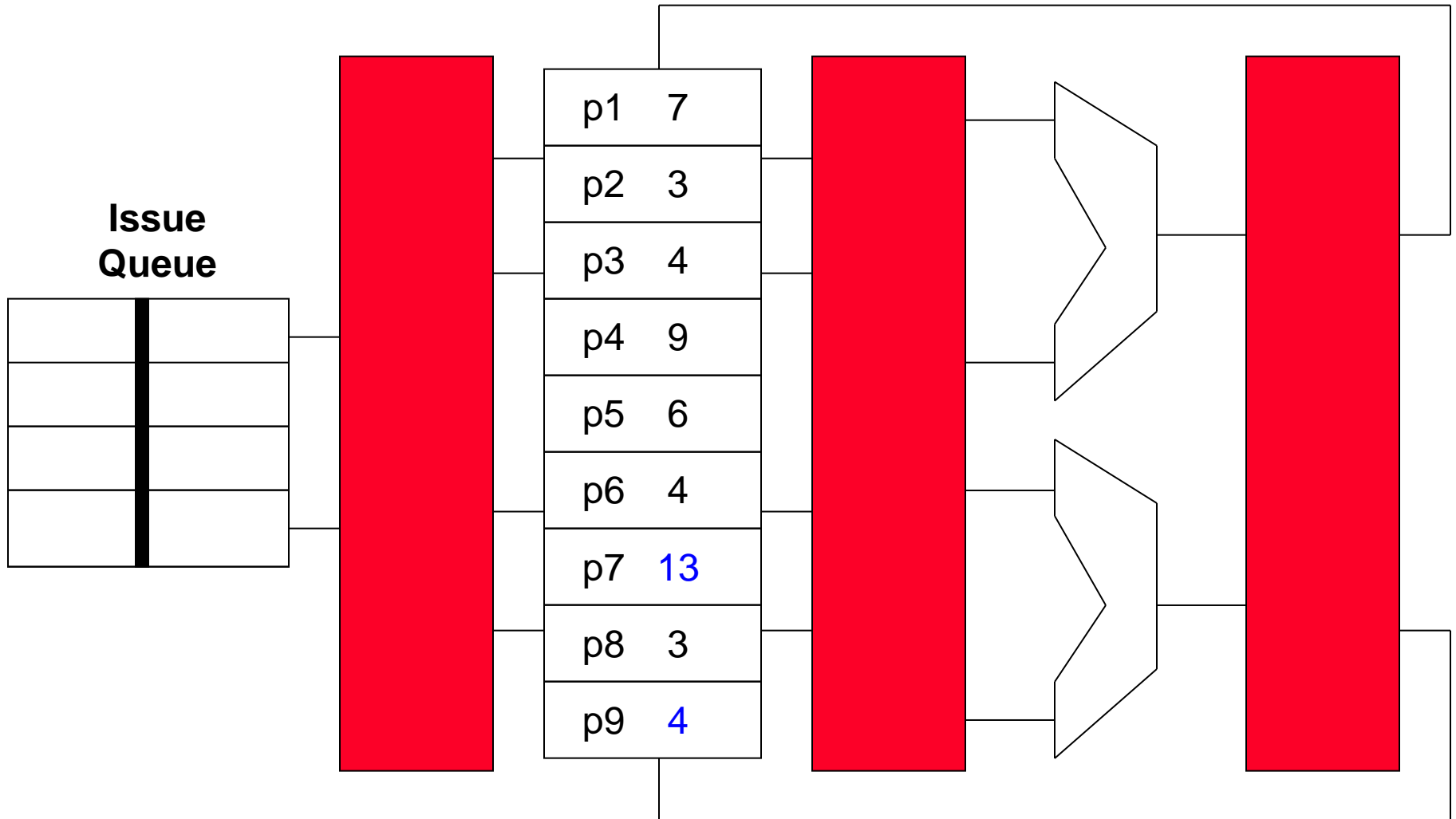
OoO execution (2-wide), 3



OoO execution (2-wide), 4



OoO execution (2-wide), 5



How about multi-cycle operations ?

- ❑ Multi-cycle ops (loads (effective address calc + DTLB + D\$), multiply, fp, etc.)
 - ❑ Wakeup may have to be deferred a few cycles
 - Can't issue until are sure that there will not be a structural hazard on WB (i.e., enough write ports to the physical register file)
- ❑ Dealing with potential cache misses?
 - ❑ Do speculative Wakeup (assume a hit)
 - ❑ On cache miss, cancel execution of dependents
 - ❑ Re-issue load (and dependent instr's) later
 - ❑ Details: complicated, not that important

Renaming review

Everyone rename this instruction:

`mul r4 * r5 → r1`

Any changes to the Map Table and/or Free List?

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`

Any changes to the Ready Table?

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Age

Ready Table

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

Select review

Which instructions are ready?

Which will be issued on a 1-wide machine?

Which will be issued on a 2-wide machine?

Issue Queue

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

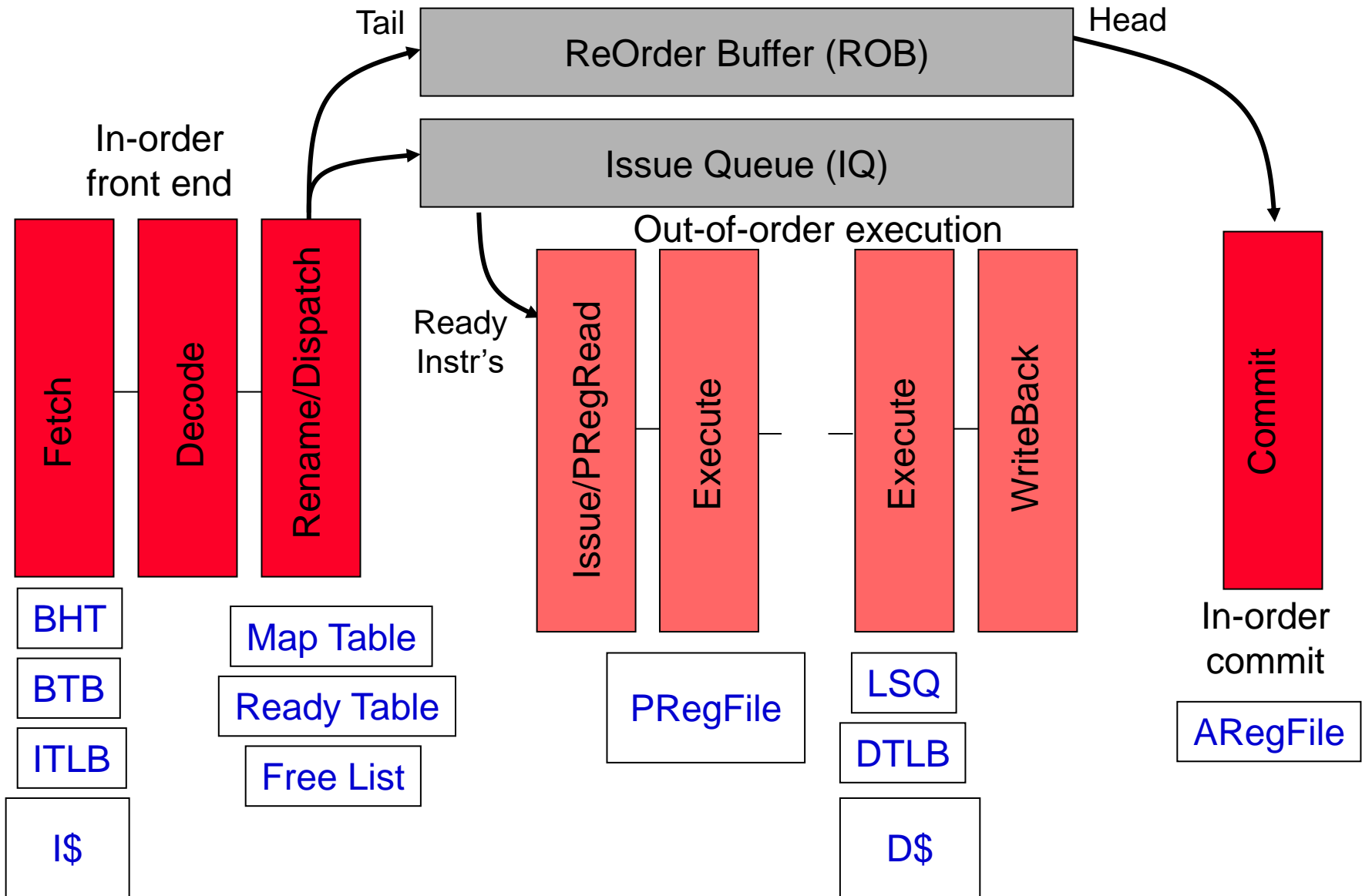
Wakeup review

What information will change if we issue the add?

Issue Queue

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

Out-of-order pipeline – the detailed view



ReOrder Buffer: ROB

- ❑ ROB entry holds all info needed for Commit & Recover
 - ❑ Physical (renamed) register dst and its architectural (ISA) equivalent
 - ❑ Overwritten physical register name
 - ❑ Instruction address (PC) and type (in particular store, branch)
 - ❑ A way of determining when the instruction completes execution
 - ❑ Exception information
- ❑ Dispatch: insert at tail
 - ❑ Full? Stall
- ❑ Commit: remove from head
 - ❑ Not completed? Stall

http://www.ecs.umass.edu/ece/koren/architecture/ROB/rob_simulator.htm

Re-Order Buffer (ROB)

- ❑ All instructions Commit in order
 - ❑ At commit write the physical register values to the ISA registers and free the overwritten physical register, write the entries in the store buffer to the D\$ (more on this soon), and free the store buffer and ROB entries for reuse
- ❑ Two other purposes
 - ❑ To support **recovery** from branch misprediction
 - On misprediction of branch at ROB head, update BHT and BTB, restart the pipeline at the branch (with the correct prediction this time), flush the ROB (wasted time, wasted power – why accurate branch prediction is **so** important)
 - ❑ To support **precise exceptions**
 - On exception of instruction at ROB head, service the exception, restart the pipeline at the interrupted instruction, flush the ROB (not as bad since exceptions are relatively infrequent)

Commit

- ❑ Commit: instruction takes on its architected state
 - ❑ In-order, so only when the instruction is finished and at the head of the ROB
 - ❑ Copy the result from the physical dst register to the ISA (architected) register in the RegFile
 - ❑ Free the overwritten physical register

```
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
```

Free these

↑

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

- ❑ Can we also free p_6 , p_7 , p_8 , p_9 ? Why or why not?

Commit example, 0

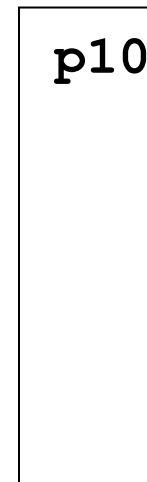
xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1

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



Free List

Commit example, 1

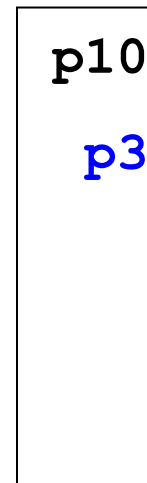
xor r1 ^ r2 → r3
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → r1

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



Free List

Commit example, 2

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

Commit example, 3

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

Commit example, 4

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

Recovery (from mispredicted branch, exception)

- ❑ Completely remove wrong path instructions
 - ❑ Flush from IQ
 - ❑ Flush from ROB
 - ❑ Restore Map Table to before misprediction
 - ❑ Restore Ready Table to before misprediction
 - ❑ Free physical destination registers (update Free List)

Recovery example, 0

- bnz resolves (just after addi was Dispatched to the IQ/ROB) and was mispredicted

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]

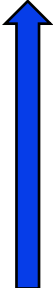
r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map Table



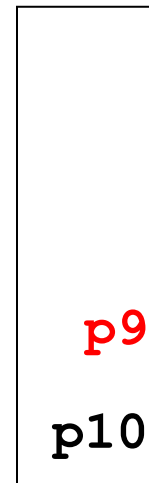
Free List

Recovery example, 1

	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, 2

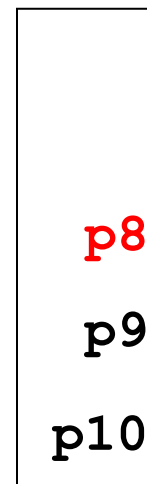
```
bnz r1, loop
xor  r1 ^ r2 → r3
add  r3 + r4 → r4
sub  r5 - r2 → r3
```

```
bnz p1, loop
xor  p1 ^ p2 → p6
add  p6 + p4 → p7
sub  p5 - p2 → p8
```

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

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map Table



Free List

Recovery example, 3

```
bnz r1, loop
xor  r1 ^ r2 → r3
add  r3 + r4 → r4
```

```
bnz p1, loop
xor  p1 ^ p2 → p6
add  p6 + p4 → p7
```

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

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map Table

p7
p8
p9
p10

Free List

Recovery example, 4

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, 5

- ❑ Update BHT and BTB and reissue branch (which will now go the other direction)

`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

Out of order pipeline diagrams

- ❑ Standard style: large and cumbersome so we will change the layout slightly
 - ❑ Columns = stages (dispatch, issue, etc)
 - ❑ Rows = instructions
 - ❑ Content of boxes = cycle number
- ❑ For our purposes: Dispatch, Issue, Exec, WB, Commit each take 1 cycle
 - ❑ We will ignore Fetch, Decode, Rename, physical regread latencies
 - ❑ Load-use, loads, mul, div, and FP take more than one cycle
 - ❑ Issue only when sure there will be no structural hazard on FU's (e.g., div) or on WB (i.e., write ports to the PRefFile)

Out of order pipeline diagram example, 0

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

2-wide

Infinite ROB, IQ, Phys.regs

Loads: 3 cycles

Out of order pipeline diagram example, 1

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 diagram example, 2

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 record its WB cycle when you do this
(Note: don't issue if WB ports at that cycle are already full)
Can't issue add since its sources aren't both ready yet

Out of order pipeline diagram example, 3

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 (load-use and RAW hazards)
- 2nd ld is ready - issue it and record its WB cycle

Out of order pipeline diagram example, 4&5

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 (WX forwarding of p2 in datapath)

Out of order pipeline diagram example, 6

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 ld can commit (oldest instruction and finished)
- xor can issue (MX forwarding of p4 in datapath)

Out of order pipeline diagram example, 7

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 diagram example, 8

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)

HANDLING MEMORY OPS

What about stores

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

- No (at least not easily)
 - ❑ Cache writes are unrecoverable
 - ❑ Stores: only when certain
 - So at Commit (until then, hold stores in a Store Queue)

Dynamically scheduling memory ops

- ❑ Compilers must schedule memory ops conservatively
- ❑ Options for hardware:
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart
 3. Learn violations over time, selectively reorder (**predictive**)

Conservative

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

Aggressive (Wrong Outcome?)

```
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)
```


Loads and stores

Instruction	Disp	Issue	WB	Commit
<code>fdiv p1/p2 → p3</code>	1	2	25	
<code>st p4 → [p5]</code>	1	2	3	
<code>st p3 → [p6]</code>	2			
<code>ld [p7] → p8</code>	2			

Cycle 3:

- Can `ld [p7] → p8` issue?
- Why or why not?

Aliasing (again)

- `p5 == p7?`
- `p6 == p7?`

Loads and stores

Instruction	Disp	Issue	WB	Commit
<code>fdiv p1/p2 → p3</code>	1	2	25	
<code>st p4 → [p5]</code>	1	2	3	
<code>st p3 → [p6]</code>	2			
<code>ld [p7] → p8</code>	2			

Suppose `p5 == p7` and `p6 != p7`
Can `ld` issue now?

Memory forwarding

- ❑ Stores write cache at Commit (until then hold store value and address in a Store Queue)
 - ❑ Commit is in-order, so allows stores to (also) be “undone” on branch mis-predictions, etc.
- ❑ Loads read cache
 - ❑ Early execution of loads is critical
- ❑ Forwarding
 - ❑ Allow store to (following) load communication before store Commits (e.g., in the previous example forward `p4` to `p8`)
 - ❑ Conceptually like register bypassing, but different implementation
 - Why? Addresses unknown until Execute

Store Queue with load forwarding

❑ Store Queue (SQ)

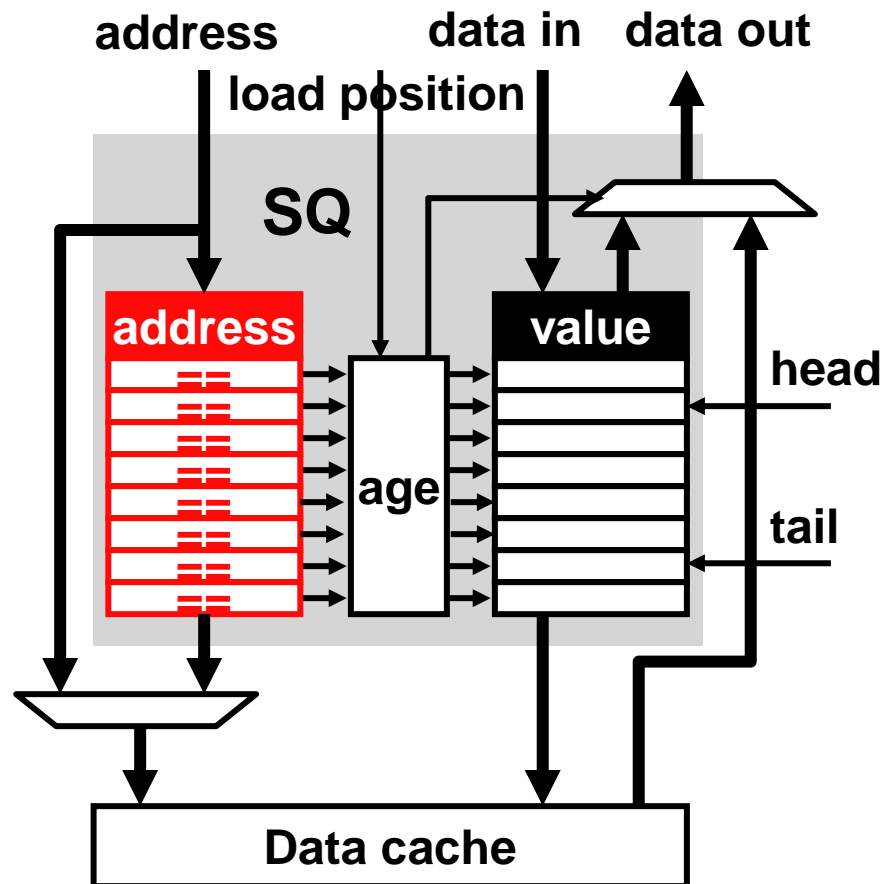
- ❑ Holds all in-flight stores
- ❑ **CAM**: searchable by address
- ❑ Age logic: determine *youngest* matching store that is *older* than the load

❑ Store execution

- ❑ Write into SQ
 - address + data value
- ❑ Store from SQ into D\$ at Commit

❑ Load execution

- ❑ Assoc search SQ addresses
 - Match? Forward youngest matching store value to load request
- ❑ Else read value from D\$



Load scheduling

❑ Load scheduling

- ❑ Determine when load can execute with regard to older stores

❑ Conservative load scheduling

- ❑ Don't issue new loads until 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 (sort of) 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
```

```
//renamed as
ld [p1] → p5
ld [p2] → p6
add p5 + p6 → p7
st p7 → [p3]
ld 4[p1] → p8
ld 4[p2] → p9
add p8 + p9 → p4
st p4 → 4[p3]
// loop control here
```

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

Our example from before, cycle 1

	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 only 1 load per cycle. Loads take 3 cycles to complete.

Our example from before, 2

	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, 3

	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, 4

	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 so can't issue
ld 4[p1] → p8 because st p7 → [p3] hasn't
finished executing (entered WB stage)

Our example from before, 6

	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, 7

	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			

- ❑ WB for the store is to the SB (*not* to the cache and not to the prefile)

Our example from before, 8

	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			

- ❑ May issue `ld 4[p1] → p8` now because can check for matching address with `st p7 → [p3]`
- ❑ Remember, only may issue 1 load per cycle

Our example from before, 9

	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, 12, 13, 14, & 15

	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 !!

A preferred (if possible) schedule

	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 cycle 4
 - Can we speculate and issue it then ?

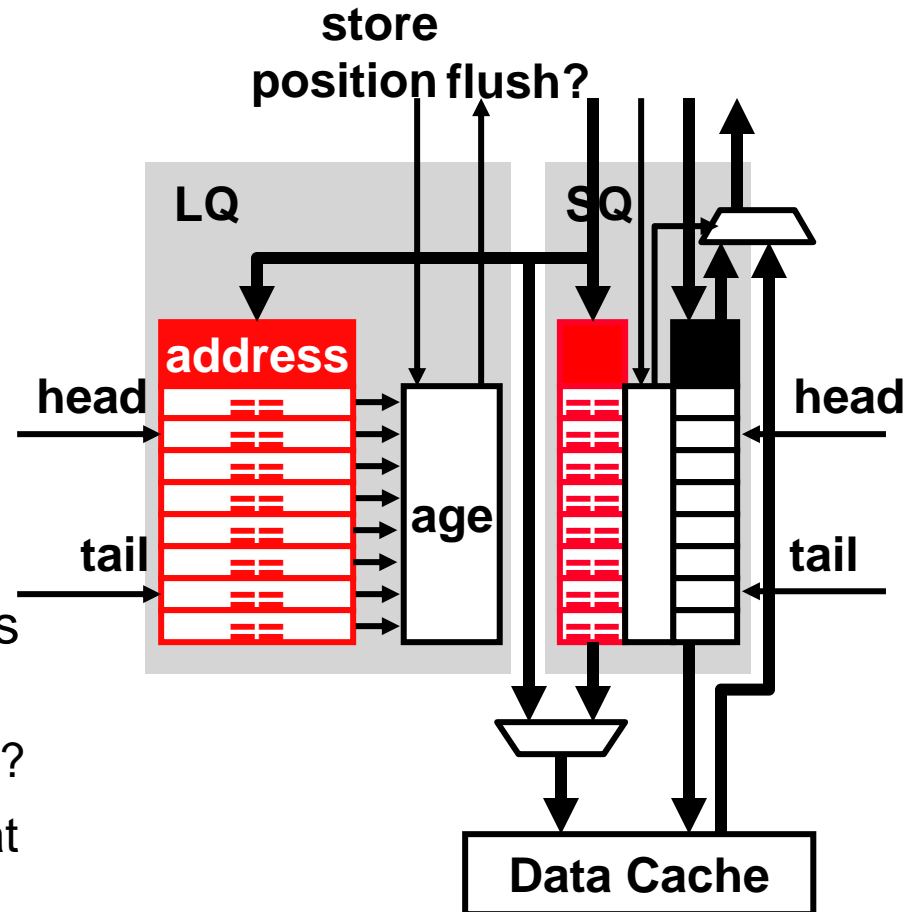
Load speculation

❑ Speculation requires two things.....

1. Detection of mis-speculations (guessed that memory addresses didn't match and it turns out that they did)
 - How can we do this?
2. Recovery from mis-speculations
 - Squash instr's after offending load (they may be using the load data)
 - Saw how to squash instr's after mispredicated branches: same method

Store Queue (SQ) + Load Queue (LQ)

- ❑ LQ detects load ordering violations
- ❑ Load execution
 - ❑ Write address into LQ
 - ❑ Also note store forwarded from if any
- ❑ Store execution
 - ❑ Assoc search LQ addresses
 - Found a younger (later in code) load with same addr?
 - If didn't forward data to that load from the youngest (older) store?
 - Then mis-speculated the load so initiate recovery



LSQ = Store Queue + Load Queue

- ❑ Store Queue: handles forwarding
 - ❑ Written into by stores (@ store execute)
 - ❑ Searched by loads (@ load execute) for store forwarding
 - ❑ Write to data cache (@ store Commit)

- ❑ Load Queue: detects ordering violations
 - ❑ Written into by loads (@ load execute)
 - ❑ Searched by stores (@ store execute) to detect load ordering violation

- ❑ Both together
 - ❑ Allows aggressive load scheduling
 - Stores don't constrain load execution

Our example from before, 4

	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, 5

	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			

❑ Aggressive load scheduling

- Issue ld 4[p2] → p9 in cycle 5

Our example from before, ... 11

	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

- ❑ Aggressive load scheduling saves 4 cycles over conservative load scheduling
 - Allowing loads to issue before older stores actually *uses* the OoO-ness of the hardware

Aside: Predictive load scheduling

- ❑ For loads that violate load ordering (i.e., addr matches that of a previous store (and no store forwarding was done))
 - ❑ Conflict => squash => worse performance than waiting
- ❑ Predict which loads must wait for stores
 - ❑ Loads default to aggressive
 - ❑ Keep table of load PCs that have been caused squashes
 - From this point on 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

1. Constrained by the number of physical registers (#preg)

- ROB roughly limited by #preg
- Big register file = expensive (area) and slow

2. Constrained by size of Issue Queue

- Limits number of un-executed instructions
- CAMs = can't make too big (power + area)

3. Constrained by sizes of Load + Store Queues

- Limits number of loads/stores
- CAMs = can't make too big (power + area)

- ❑ Active area of research: scaling window sizes

❑ Usefulness of large window: limited by branch prediction

- ❑ 95% branch mis-prediction rate: 1 in 20 branches, or 1 in 100 instr's

Out of Order: Benefits

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

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 before commit, just flush the pipeline
- ❑ Current machines: 100+ instruction scheduling window

Out of Order: Top 5 things to know

- ❑ Register renaming
 - ❑ How to perform it and how to recover it
- ❑ 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
- ❑ Commit
 - ❑ Precise state (ROB)
 - ❑ How/when registers are freed

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