CS3014: Concurrent Systems

Static & Dynamic Instruction Scheduling

Slides originally developed by
Drew Hilton, Amir Roth, Milo Martin and Joe Devietti
at University of Pennsylvania

Instruction Scheduling & Limitations

Instruction Scheduling

- Scheduling: act of finding independent instructions
  - “Static” done at compile time by the compiler (software)
  - “Dynamic” done at runtime by the processor (hardware)

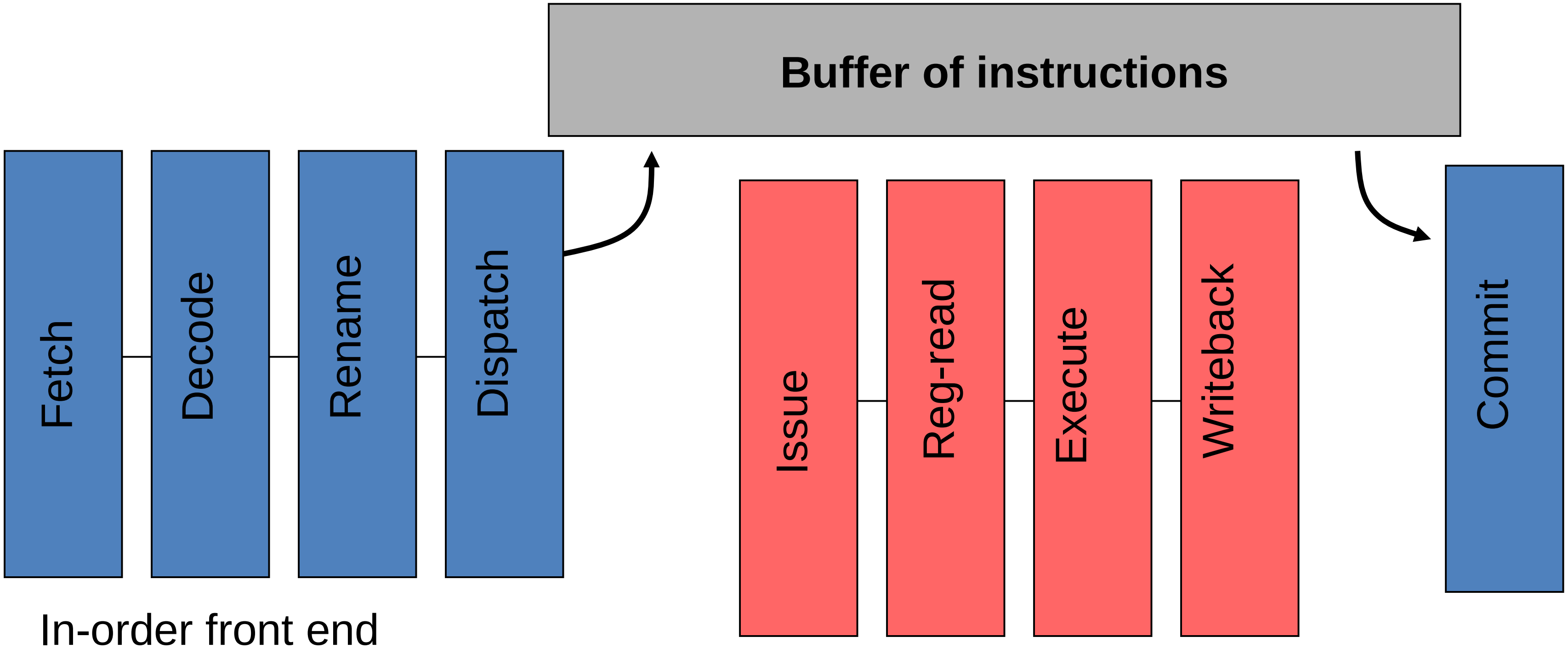
- Why schedule code?
  - Scalar pipelines: fill in load-to-use delay slots to improve CPI
  - Superscalar: place independent instructions together
    - As above, load-to-use delay slots
    - Allow multiple-issue decode logic to let them execute at the same time

Dynamic (Execution-time) Instruction Scheduling

Can Hardware Overcome These Limits?

- **Dynamically-scheduled processors**
  - Also called “out-of-order” processors
  - Hardware re-schedules instructions…
  - …within a sliding window of instructions
  - As with pipelining and superscalar, ISA unchanged
    - Same hardware/software interface, appearance of in-order
- Increases scheduling scope
  - Does loop unrolling transparently!
  - Uses branch prediction to “unroll” branches
- Examples:
  - Pentium Pro/II/III (3-wide), Core 2 (4-wide),
    Alpha 21264 (4-wide), MIPS R10000 (4-wide), Power5 (5-wide)

Out-of-Order Pipeline



Buffer of instructions

Fetch

Decode

Rename

Dispatch

Issue

Reg-read

Execute

Writeback

Commit

In-order front end

Out-of-order execution

In-order commit

Out-of-Order Execution

- Also called “Dynamic scheduling”
  - Done by the hardware on-the-fly during execution

- Looks at a “window” of instructions waiting to execute
  - Each cycle, picks the next ready instruction(s)

- Two steps to enable out-of-order execution:

Step #1: Register renaming – to avoid “false” dependencies
Step #2: Dynamically schedule – to enforce “true” dependencies

- Key to understanding out-of-order execution:
  - **Data dependencies**

Dependence types

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

mul r0 * r1 ➜ **r2**
…
add **r2** + r3 ➜ r4

- **WAW** (Write After Write) = “output dependence”   (false)

mul r0 * r1➜ **r2**
…
add r1 + r3 ➜ **r2**

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

mul r0 * **r1** ➜ r2
…
add r3 + r4 ➜ **r1**

- WAW & WAR are “false”, Can be **totally eliminated** by “renaming”

Step #1: Register Renaming

- To eliminate register conflicts/hazards
- “Architected” vs “Physical” registers – level of indirection
  - Names: **r1,r2,r3**
  - Locations: **p1,p2,p3,p4,p5,p6,p7**
  - Original mapping: **r1**→**p1**, **r2**→**p2**, **r3**→**p3**, **p4**–**p7** are “available”

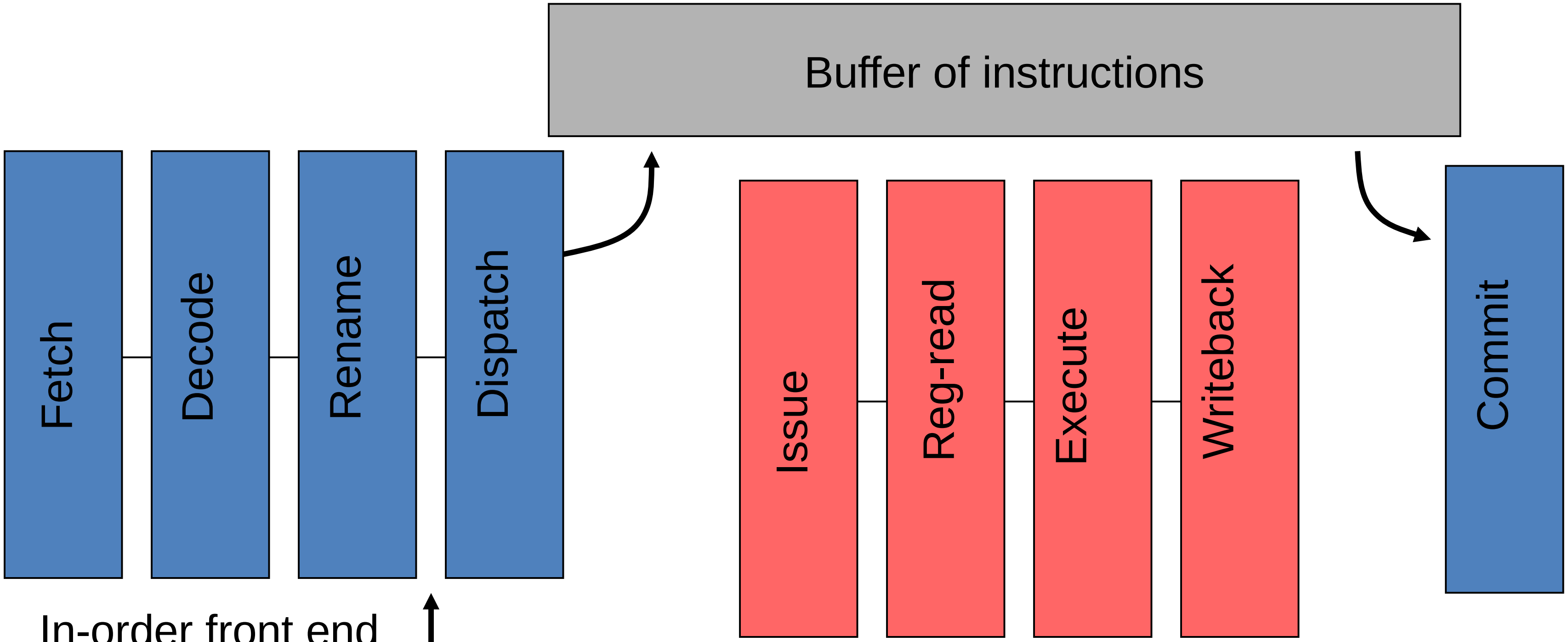
MapTable

Time

r1	r2	r3	FreeList	Original insns	Renamed insns
p1	p2	p3	p4,p5,p6,p7	add r2,r3→r1	add p2,p3→p4
p4	p2	p3	p5,p6,p7	sub r2,r1→r3	sub p2,p4→p5
p4	p2	p5	p6,p7	mul r2,r3→r3	mul p2,p5→p6
p4	p2	p6	p7	div r1,#4→r1	div p4,#4→p7

  - Renaming – conceptually write each register once
    - + Removes **false** dependences
    - + Leaves **true** dependences intact!
  - When to reuse a physical register? After overwriting instruction is complete

Out-of-order Pipeline



In-order front end

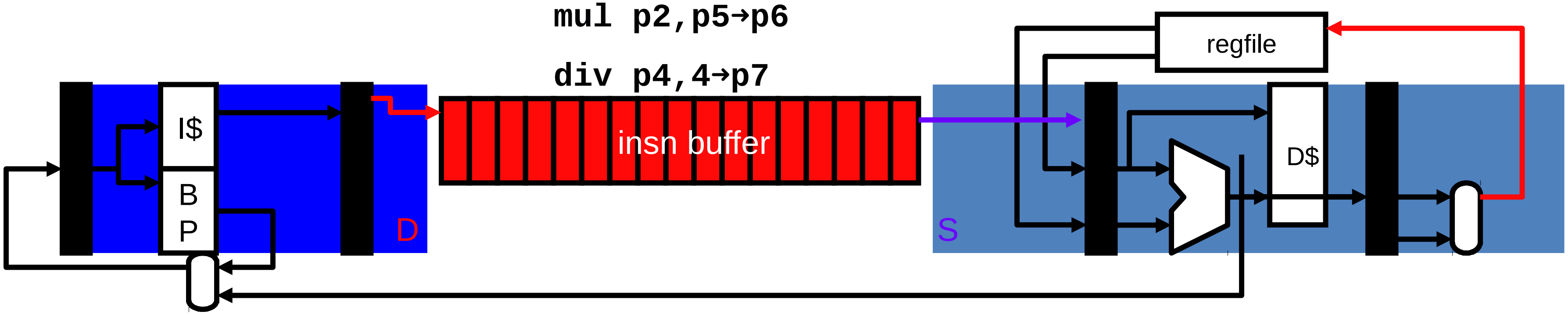
Out-of-order execution

In-order commit

Have unique register names
Now put into out-of-order execution structures

Step #2: Dynamic Scheduling

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



Ready Table

Time

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

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

- Instructions fetch/decoded/renamed into *Instruction Buffer*
  - Also called “instruction window” or “instruction scheduler”
- Instructions (conceptually) check ready bits every cycle
  - Execute oldest “ready” instruction, set output as “ready”

Dynamic Scheduling/Issue Algorithm

- Data structures:
  - Ready table[phys_reg] → yes/no (part of “issue queue”)

- Algorithm at “issue” 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
```

- Multiple-cycle instructions? (such as loads)
  - For an instruction with latency of N, set “ready” bit N-1 cycles in future

Register Renaming

Register Renaming Algorithm (Simplified)

- Two key data structures:
  - maptable[architectural_reg] → physical_reg
  - Free list: allocate (new) & free registers (implemented as a queue)
    - ignore freeing of registers for now
- Algorithm: at “decode” stage for each instruction:
  - Rewrites instruction with “physical” registers (rather than “architectural” registers

```
insn.phys_input1 = maptable[insn.arch_input1]
insn.phys_input2 = maptable[insn.arch_input2]

new_reg = new_phys_reg()
maptable[insn.arch_output] = new_reg
insn.phys_output = new_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
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1

⟶

xor **p1** ^ **p2** ➜

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

→

xor p1 ^ p2 ➜ **p6**

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**
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1

→

xor p1 ^ p2 ➜ p6

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

Map table

p7
p8
p9
p10

Free-list

Renaming example

xor r1 ^ r2 ➜ r3

add **r3** + **r4** ➜ r4

sub r5 - r2 ➜ r3

addi r3 + 1 ➜ r1

xor p1 ^ p2 ➜ p6

add **p6** + **p4** ➜

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

Map table

p7

p8

p9

p10

Free-list

Renaming 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**

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map table

p7
p8
p9
p10

Free-list

Renaming 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

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

Map table

p8
p9
p10

Free-list

Renaming 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** ➜

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

Map table

p8
p9
p10

Free-list

Renaming 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**

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map table

p8
p9
p10

Free-list

Renaming 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

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

Map table

p9

p10

Free-list

Renaming 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 ➜

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

Map table

p9
p10

Free-list

Renaming 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**

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p9
p10

Free-list

Renaming example

xor r1 ^ r2 ➜ r3	xor p1 ^ p2 ➜ p6
add r3 + r4 ➜ r4	add p6 + p4 ➜ p7
sub r5 - r2 ➜ r3	sub p5 - p2 ➜ p8
addi r3 + 1 ➜ **r1**	addi p8 + 1 ➜ p9

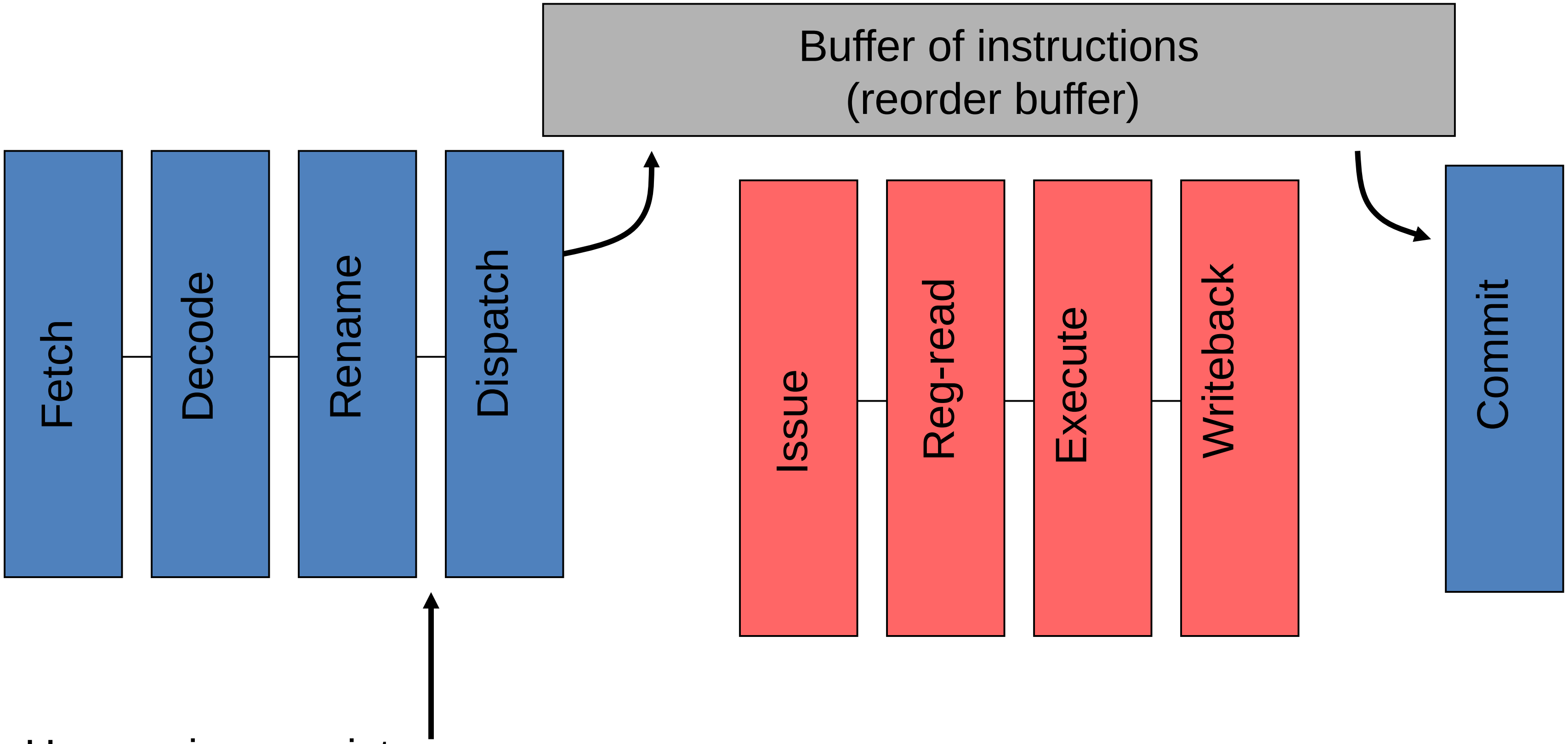
r1	**p9**
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10

Free-list

Out-of-order Pipeline

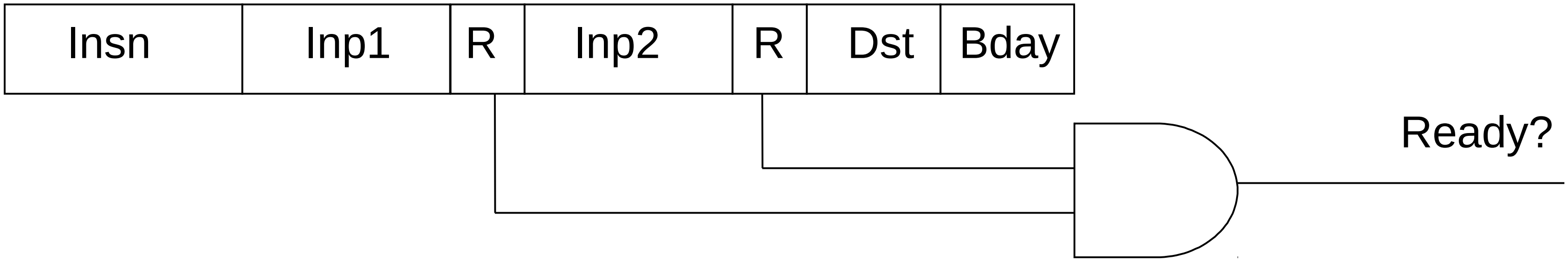


Have unique register names
Now put into out-of-order execution structures

Dynamic Instruction Scheduling Mechanisms

Dispatch

- Put renamed instructions into out-of-order structures
- Re-order buffer (ROB)
  - Holds instructions from Fetch through Commit
- Issue Queue
  - Central piece of scheduling logic
  - Holds instructions from Dispatch through Issue
  - Tracks ready inputs
    - Physical register names + ready bit
    - “AND” the bits to tell if ready



Insn	Inp1	R	Inp2	R	Dst	Bday

Ready?

Dispatch Steps

- Allocate Issue Queue (IQ) slot
  - Full? Stall
- Read **ready bits** of inputs
  - 1-bit per physical reg
- Clear **ready bit** of output in table
  - Instruction has not produced value yet
- Write data into Issue Queue (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	Bday

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

Ready bits

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

Issue Queue

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

Dispatch Example

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

Ready bits

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

Issue Queue

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

Dispatch Example

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

Ready bits

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

Issue Queue

Insn	**Inp1**	**R**	**Inp2**	**R**	**Dst**	**Bday**
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1
sub	p5	y	p2	y	p8	2

Dispatch Example

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

Ready bits

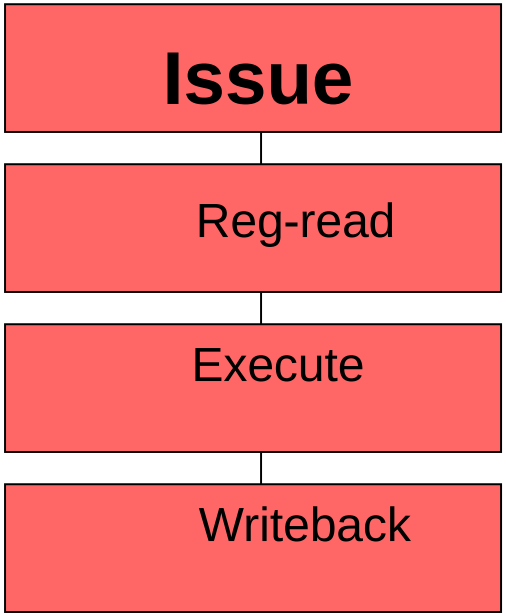
p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	n
p9	**n**

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Bday
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

Out-of-order pipeline

- Execution (out-of-order) 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** oldest of “ready” instructions
  - “xor” is the oldest ready instruction below
  - “xor” and “sub” are the two oldest ready instructions below
  - Note: may have resource constraints: i.e. load/store/floating point

Insn	Inp1	R	Inp2	R	Dst	Bday	
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
  - Search for destination (Dst) in inputs & set "ready" bit
    - Implemented with a special memory array circuit called a Content Addressable Memory (CAM)
  - Also update ready-bit table for future instructions

Insn	Inp1	R	Inp2	R	Dst	Bday
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

- For multi-cycle operations (loads, floating point)
  - Wakeup deferred a few cycles
  - Include checks to avoid structural hazards

Note: Content Addressable Memory

- A content addressable memory (CAM) is indexed by the **content** of each location, not by the address
  - Sometimes known as associative memory
- It compares an input “key” against a table of keys, and returns the location of the key in the table
  - In software this might be implemented with a hash table
  - Hardware hash table is also possible, but potentially slow
- To search all locations in a single cycle
  - You need to be able to compare the search key to all keys in the table simultaneously
    - This requires a *lot* of hardware
    - Fast CAMs are very hardware expensive
  - If you need to be able to do multiple searches in the same cycle, the hardware requirements are even greater

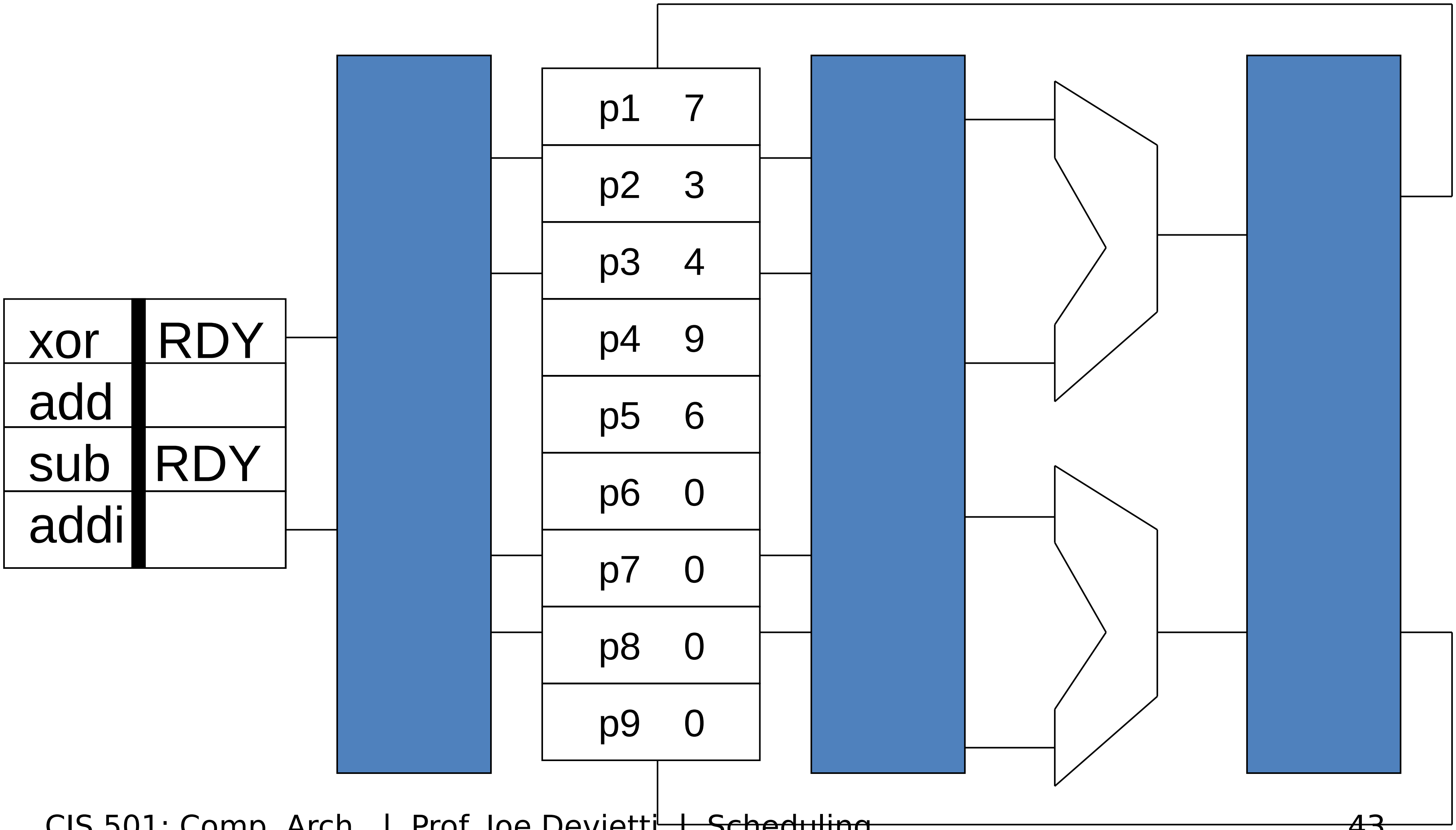
Issue

- **Select/Wakeup** one cycle
- Dependent instructions execute on back-to-back cycles
  - Next cycle: add/addi are ready:

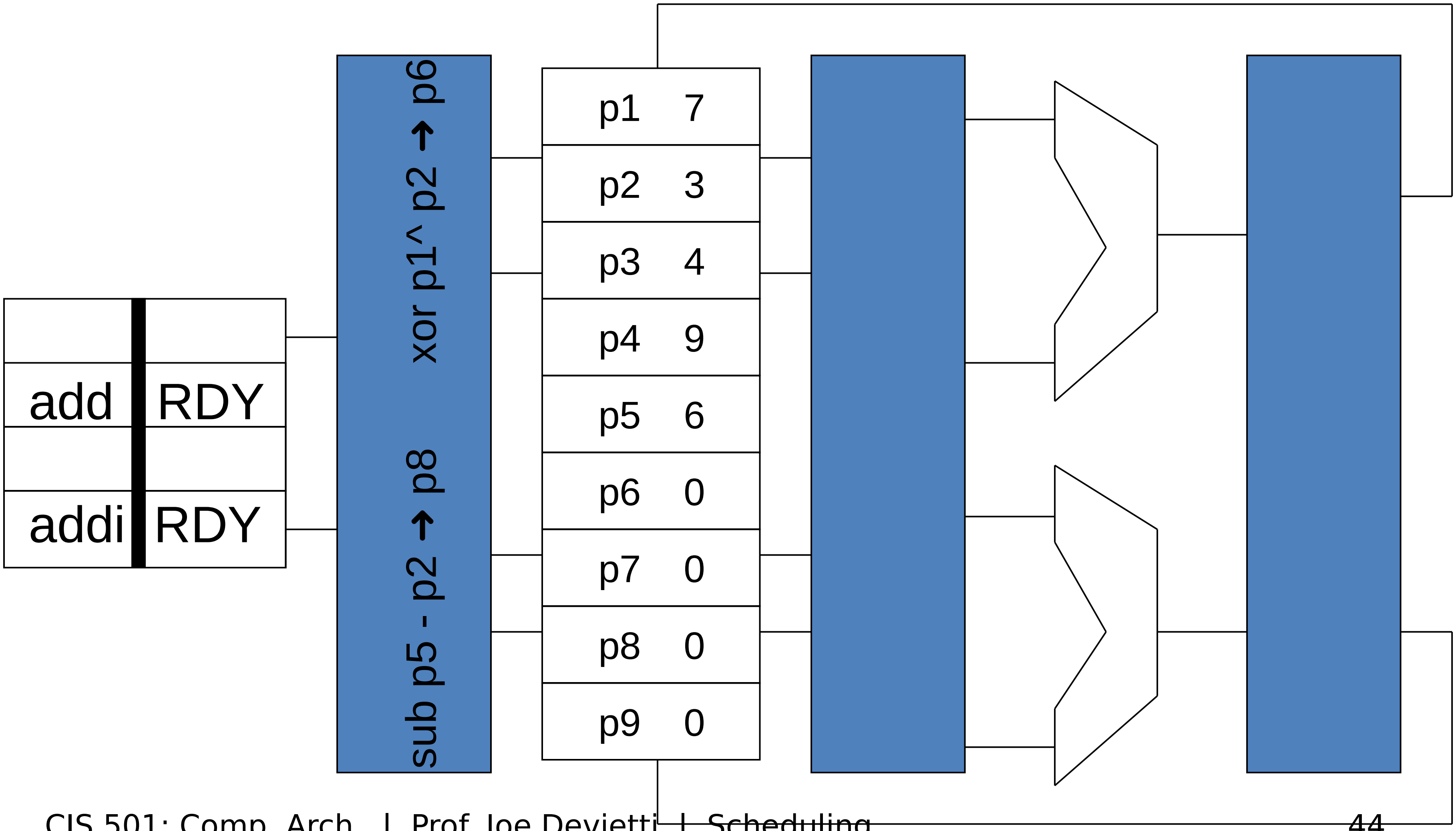
Insn	Inp1	R	Inp2	R	Dst	Bday
add	p6	**y**	p4	y	p7	1
addi	p8	**y**	---	y	p9	3

- Issued instructions are removed from issue queue
  - Free up space for subsequent instructions

OOO execution (2-wide)



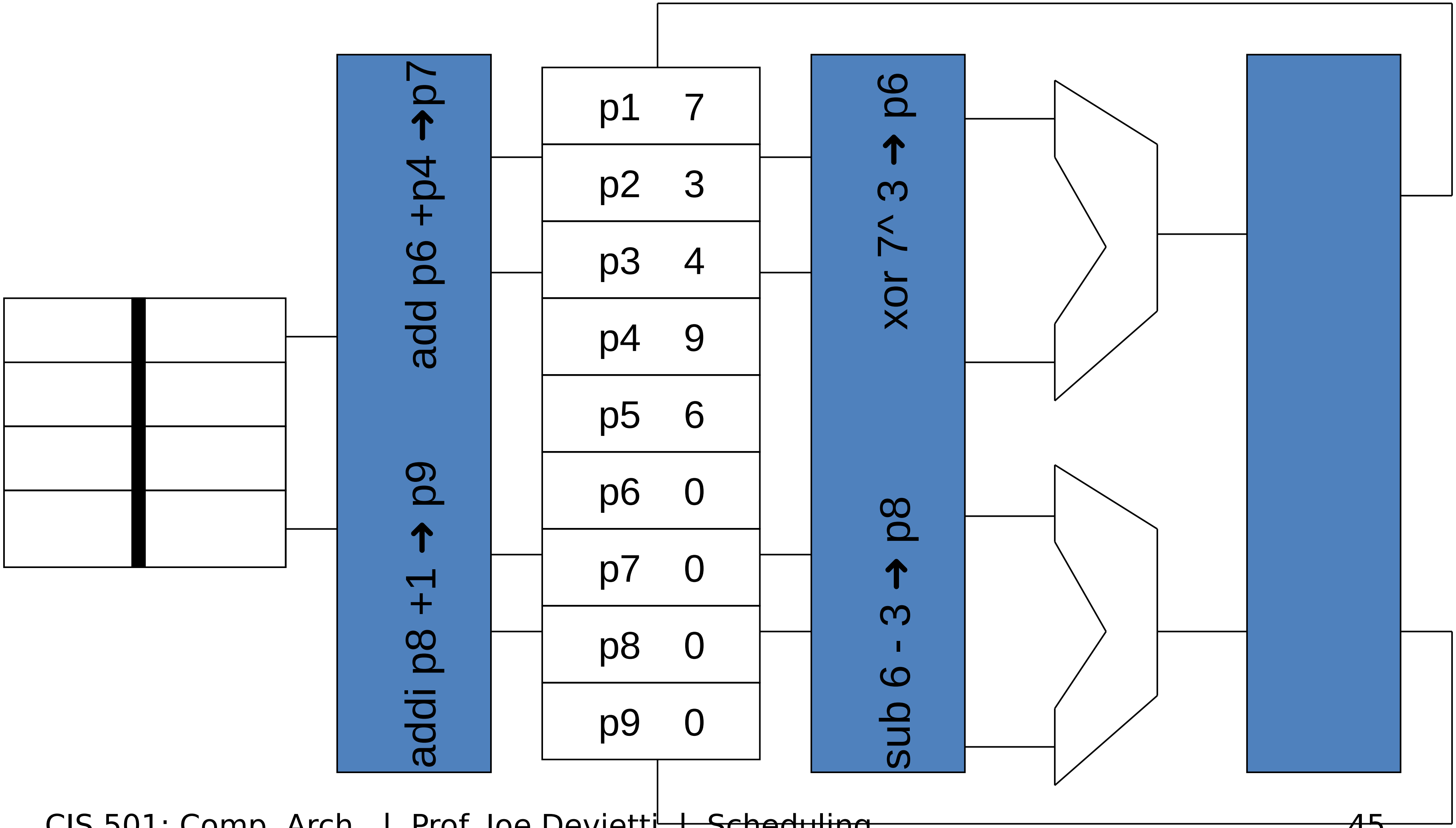
OOO execution (2-wide)



add	RDY
addi	RDY

p1	7
p2	3
p3	4
p4	9
p5	6
p6	0
p7	0
p8	0
p9	0

OOO execution (2-wide)



add p6 +p4 →p7

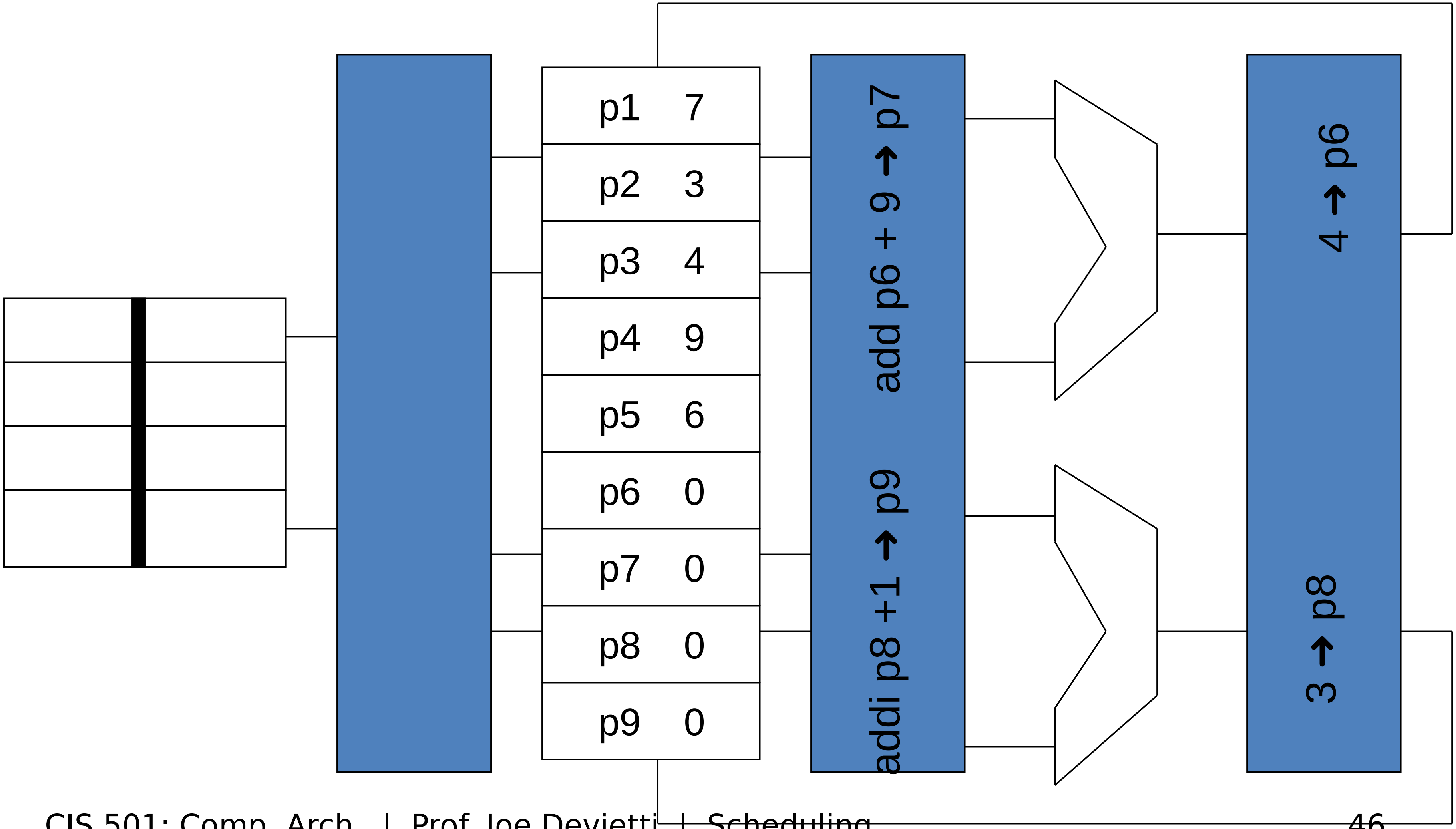
addi p8 +1 → p9

p1	7
p2	3
p3	4
p4	9
p5	6
p6	0
p7	0
p8	0
p9	0

xor 7^ 3 → p6

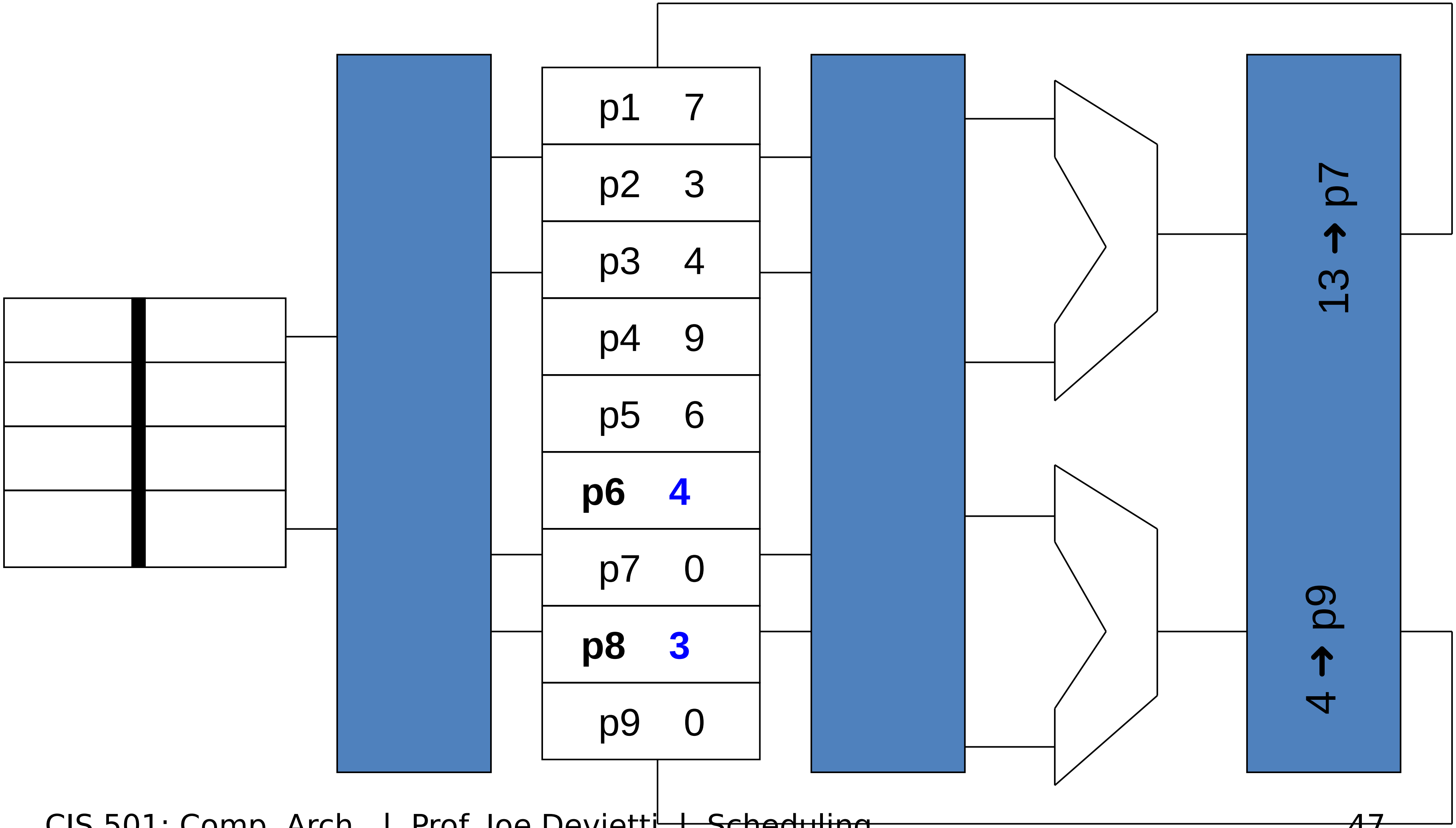
sub 6 - 3 → p8

OOO execution (2-wide)



p1	7
p2	3
p3	4
p4	9
p5	6
p6	0
p7	0
p8	0
p9	0

OOO execution (2-wide)

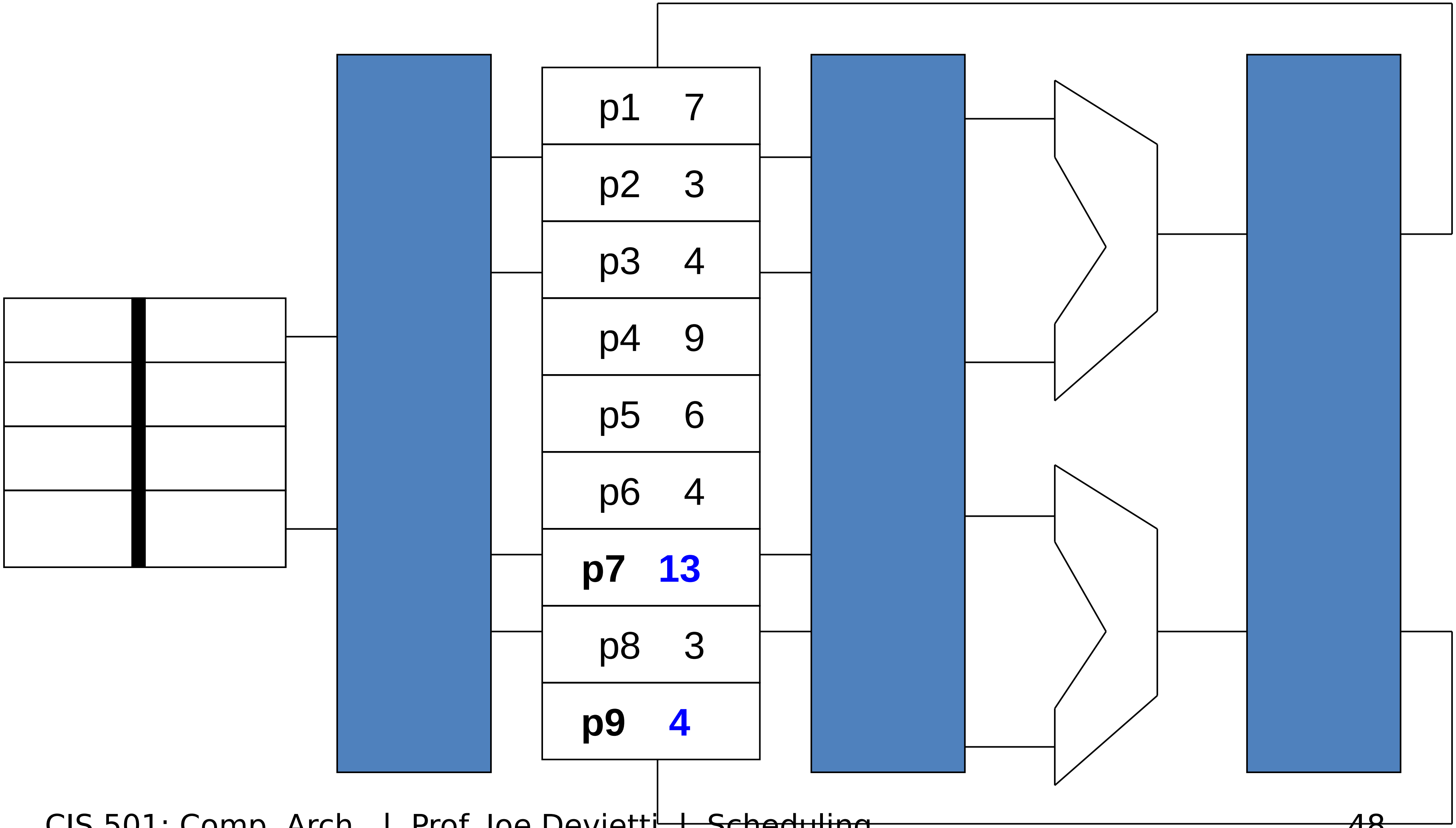


p1	7
p2	3
p3	4
p4	9
p5	6
p6	**4**
p7	0
p8	**3**
p9	0

13 → p7

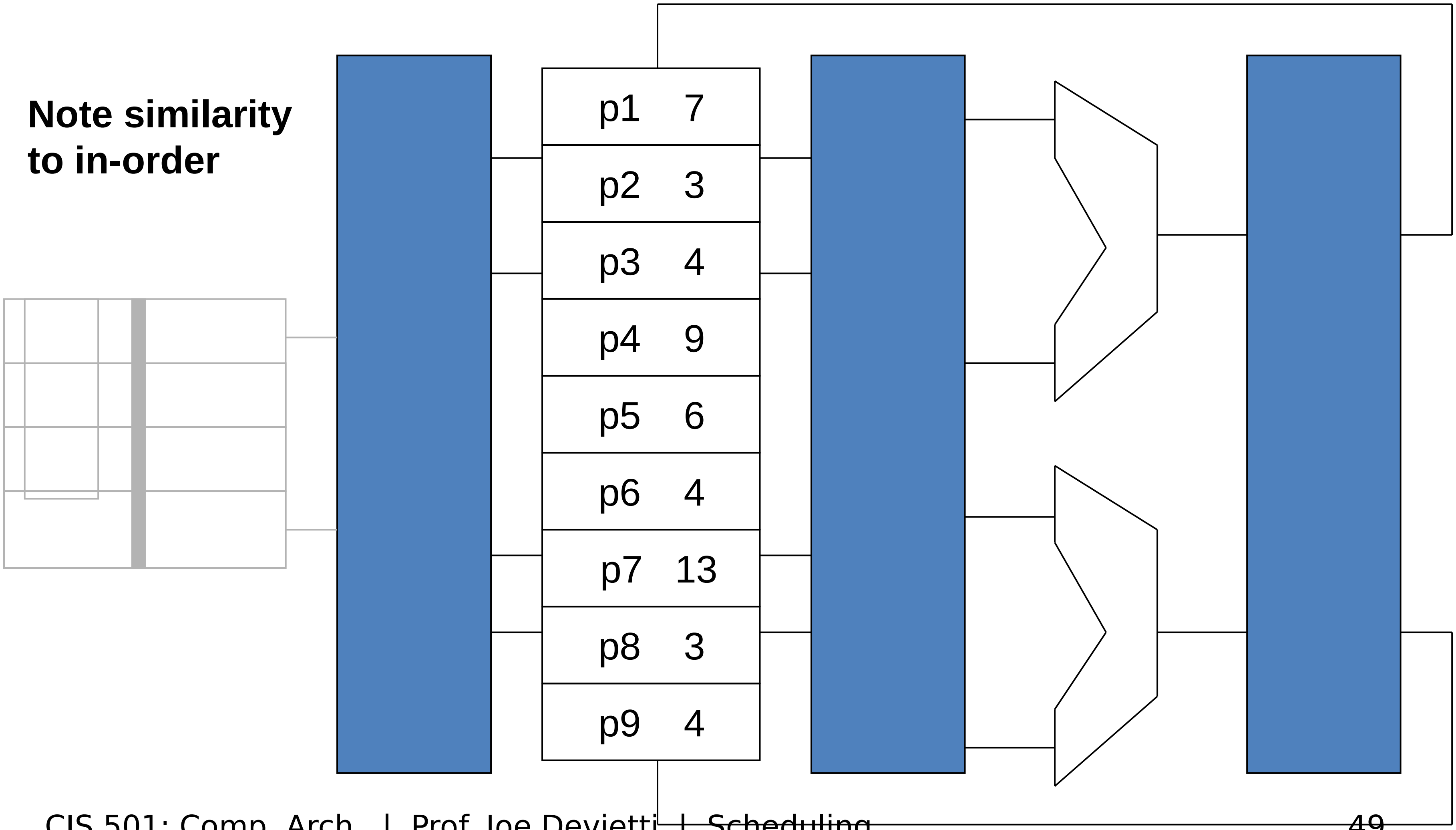
4 → p9

OOO execution (2-wide)



p1	7
p2	3
p3	4
p4	9
p5	6
p6	4
p7	**13**
p8	3
p9	**4**

OOO execution (2-wide)



Note similarity to in-order

p1	7
p2	3
p3	4
p4	9
p5	6
p6	4
p7	13
p8	3
p9	4

Out-of-Order: Benefits & Challenges

Dynamic Scheduling Operation (Recap)

- Dynamic scheduling
  - Totally in the hardware (not visible to software)
  - 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 registers to avoid false dependencies
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies is harder
- “Commit” instructions in order
  - Anything strange happens before commit, just flush the pipeline

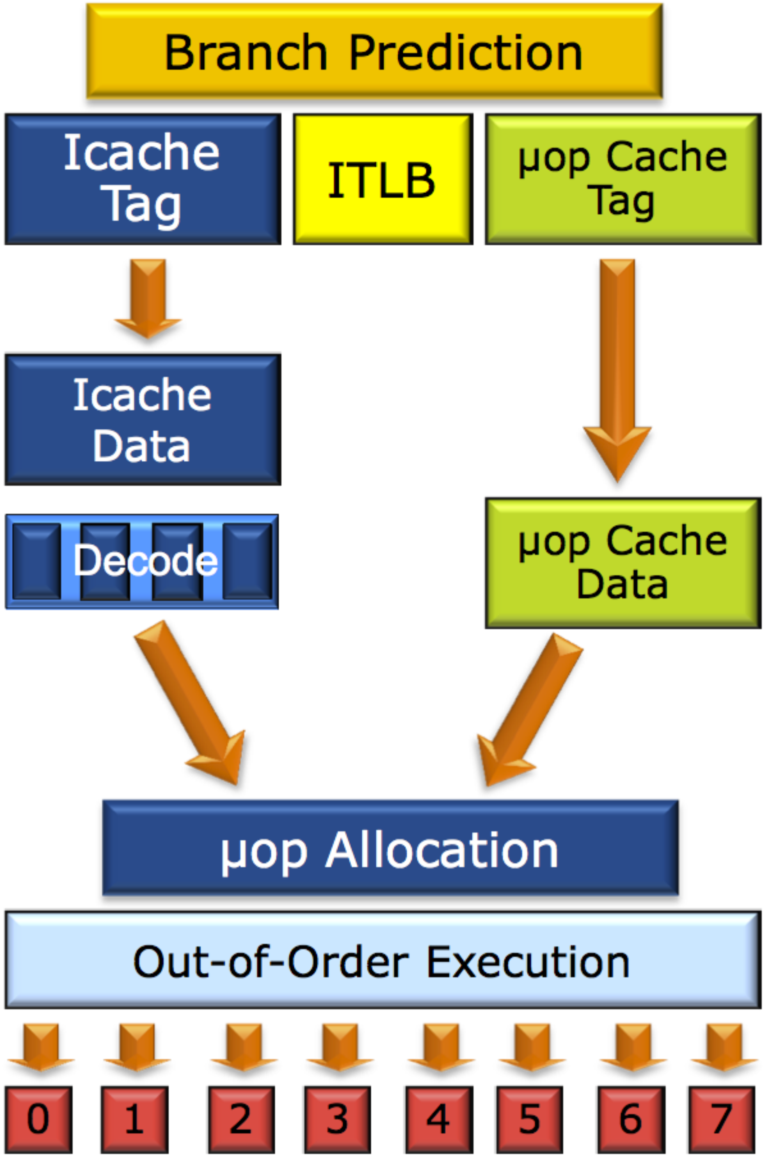
Haswell Buffer Sizes

Extract more parallelism in every generation

	Nehalem	Sandy Bridge	Haswell	
Out-of-order Window	128	168	192	⬆
In-flight Loads	48	64	72	⬆
In-flight Stores	32	36	42	⬆
Scheduler Entries	36	54	60	⬆
Integer Register File	N/A	160	168	⬆
FP Register File	N/A	144	168	⬆
Allocation Queue	28/thread	28/thread	56	⬆

Intel® Microarchitecture (Haswell); Intel® Microarchitecture (Nehalem); Intel® Microarchitecture (Sandy Bridge)

Haswell Core at a Glance



Next generation branch prediction

- Improves performance *and* saves wasted work

Improved front-end

- Initiate TLB and cache misses speculatively
- Handle cache misses in parallel to hide latency
- Leverages improved branch prediction

Deeper buffers

- Extract more instruction parallelism
- More resources when running a single thread

More execution units, shorter latencies

- Power down when not in use

More load/store bandwidth

- Better prefetching, better cache line split latency & throughput, double L2 bandwidth
- New modes save power without losing performance

No pipeline growth

- Same branch misprediction latency
- Same L1/L2 cache latency

Core Cache Size/Latency/Bandwidth

Metric	Nehalem	Sandy Bridge	Haswell
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way
Fastest Load-to-use	4 cycles	4 cycles	4 cycles
Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	**64 Bytes/cycle**
Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	**32 Bytes/cycle**
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
Fastest load-to-use	10 cycles	11 cycles	11 cycles
Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	**64 Bytes/cycle**
L1 Instruction TLB	4K: 128, 4-way 2M/4M: 7/thread	4K: 128, 4-way 2M/4M: 8/thread	4K: 128, 4-way 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	**4K+2M shared: 1024, 8-way**

All caches use 64-byte lines

Intel® Microarchitecture (Haswell); Intel® Microarchitecture (Sandy Bridge); Intel® Microarchitecture (Nehalem)

OoO Execution is all around us

- Example phone CPU: Qualcomm Krait 400 processor
    - based on ARM Cortex A15 processor
    - **out-of-order** 2.5GHz quad-core
    - 3-wide fetch/decode
    - 4-wide issue
    - 11-stage integer pipeline
    - 28nm process technology
    - 4/4KB DM L1$, 16/16KB 4-way SA L2$, 2MB 8-way SA L3$

Out of Order: Benefits

- Allows speculative re-ordering
  - Loads / stores
  - Branch prediction to look past branches
- Done by hardware
  - Compiler may want different schedule for different hw configs
  - Hardware has only its own configuration to deal with
- Schedule can change due to cache misses
- **Memory-level parallelism**
  - Executes “around” cache misses to find independent instructions
  - Finds and initiates independent misses, reducing memory latency
    - Especially good at hiding L2 hits (~12 cycles in Core i7)

Challenges for Out-of-Order Cores

- Design complexity
  - More complicated than in-order? Certainly!
  - But, we have managed to overcome the design complexity
- Clock frequency
  - Can we build a "high ILP" machine at high clock frequency?
  - Yep, with some additional pipe stages, clever design
- Limits to (efficiently) scaling the window and ILP
  - Large physical register file
  - Fast register renaming/wakeup/select/load queue/store queue
    - Active areas of micro-architectural research
  - Branch & memory depend. prediction (limits effective window size)
    - 95% branch mis-prediction: 1 in 20 branches, or 1 in 100 insn.
  - Plus all the issues of building "wide" in-order superscalar
- Power efficiency
  - Today, even mobile phone chips are out-of-order cores

Redux: HW vs. SW Scheduling

- Static scheduling
  - Performed by compiler, limited in several ways
- Dynamic scheduling
  - Performed by the hardware, overcomes limitations
- Static limitation ➜ dynamic mitigation
  - Number of registers in the ISA ➜ register renaming
  - Scheduling scope ➜ branch prediction & speculation
  - Inexact memory aliasing information ➜ speculative memory ops
  - Unknown latencies of cache misses ➜ execute when ready
- Which to do? **Compiler does what it can, hardware the rest**
  - Why? dynamic scheduling needed to sustain more than 2-way issue
  - **Helps with hiding memory latency** (execute around misses)
  - Intel Core i7 is four-wide execute w/ scheduling window of 100+
  - Even mobile phones have dynamically scheduled cores (ARM A9, A15)