Topic 4

Advanced Processors II

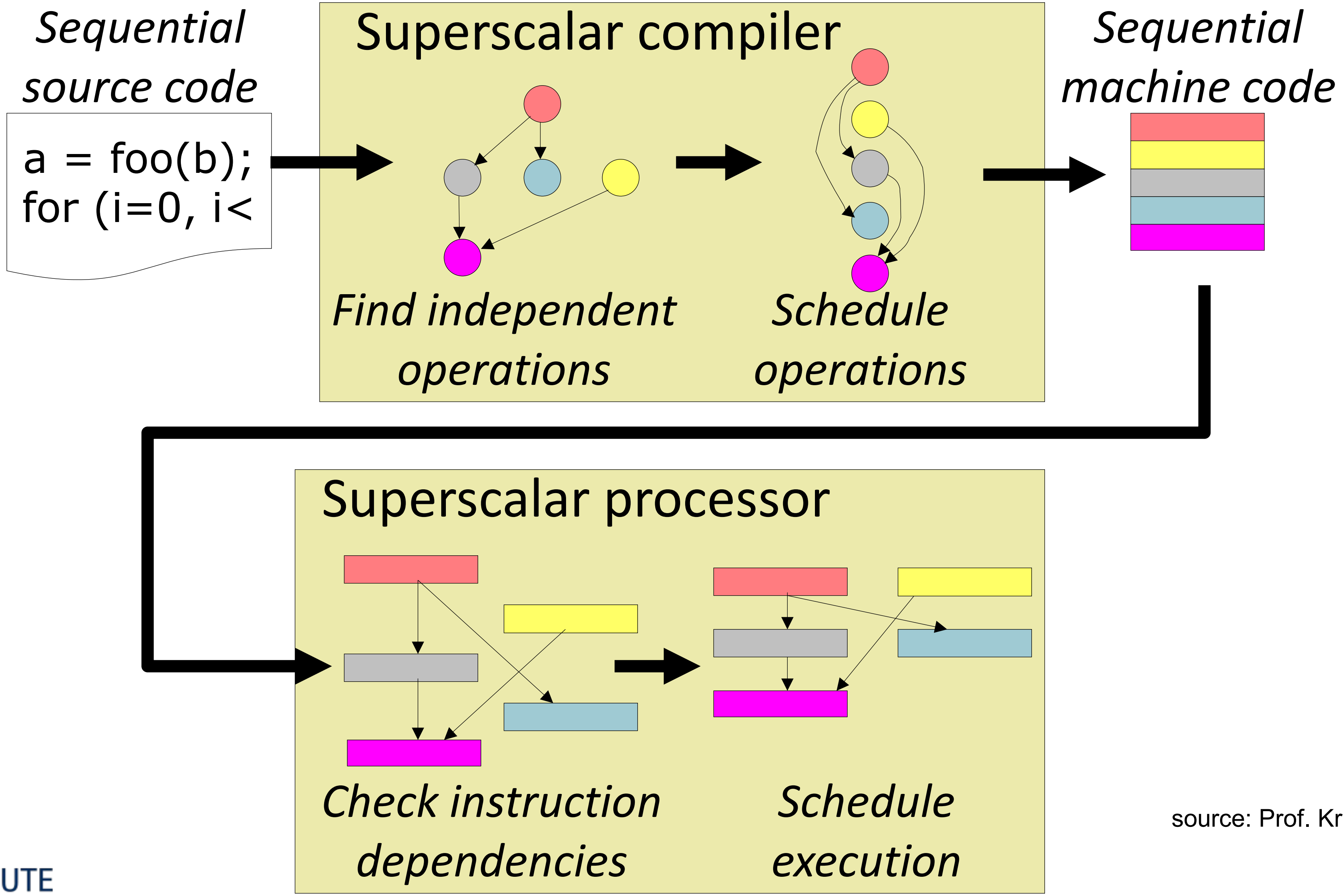
Xinfei Guo
xinfei.guo@sjtu.edu.cn

June 3rd, 2024

T4 learning goals

- Advanced Processors
  - Section I: Superpipelined & Superscalar Pipelines
  - Section II & III: Out-of-order (OoO) Pipelines

Sequential ISA Bottleneck



Sequential source code

```
a = foo(b);
for (i=0, i<
```

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code

Superscalar processor

Check instruction dependencies

Schedule execution

source: Prof. Krste Asanovic @UCB

Multiple Issue

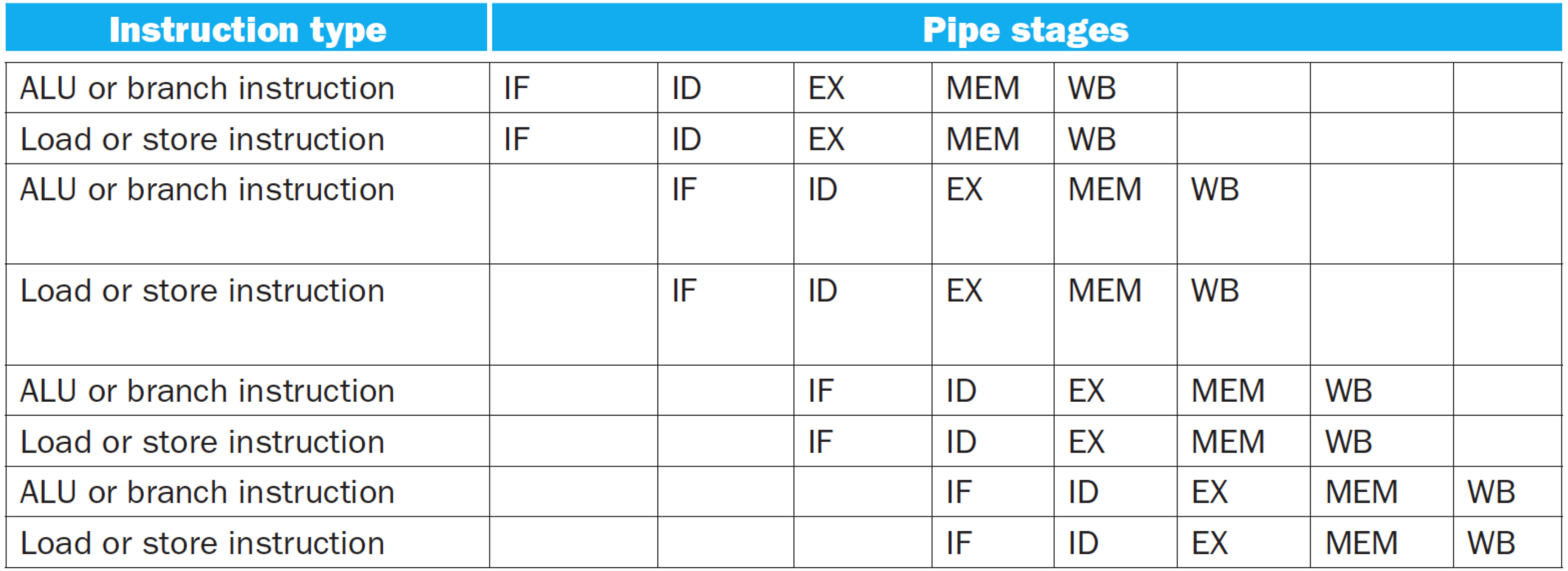
- Static multiple issue (software based)
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue (hardware based)
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

Example

- A two-issue (dual-issue) pipeline in operation



Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Loop Unrolling

- A Compiler Technique
- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

Loop Unrolling Example (4 iterations)

	ALU/branch	Load/store	cycle
Loop:	addi x20,x20,-32	ld x28, 0(x20)	1
	nop	ld x29, 24(x20)	2
	add x28,x28,x21	ld x30, 16(x20)	3
	add x29,x29,x21	ld x31, 8(x20)	4
	add x30,x30,x21	sd x28, 32(x20)	5
	add x31,x31,x21	sd x29, 24(x20)	6
	nop	sd x30, 16(x20)	7
	blt x22,x20,Loop	sd x31, 8(x20)	8

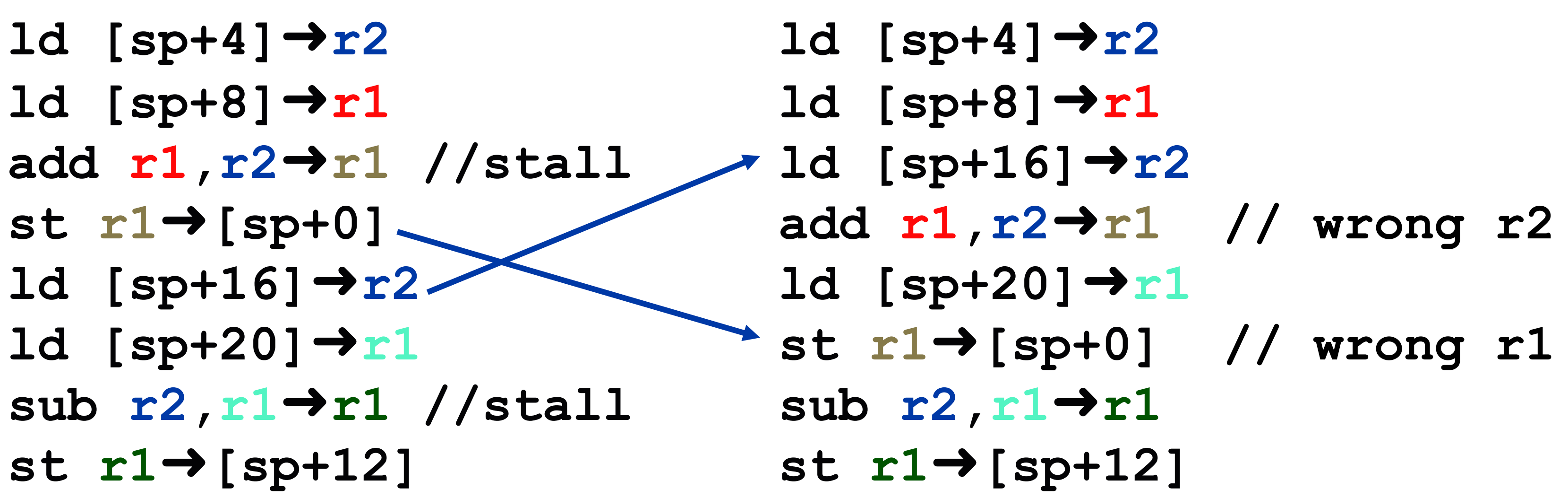
X28, x29, x30 are new!
Note how the address was updated!

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

```
Loop: ld   x31,0(x20)      // x31=array element
      add  x31,x31,x21     // add scalar in x21
      sd   x31,0(x20)      // store result
      addi x20,x20,-8      // decrement pointer
      blt  x22,x20,Loop    // branch if x22 < x20
```

Compiler Scheduling Requires

- **Enough registers**
  - To hold additional "live" values
  - Example code contains 7 different values (including `sp`)
  - Before: max 3 values live at any time → 3 registers enough
  - After: max 4 values live → 3 registers not enough



Original

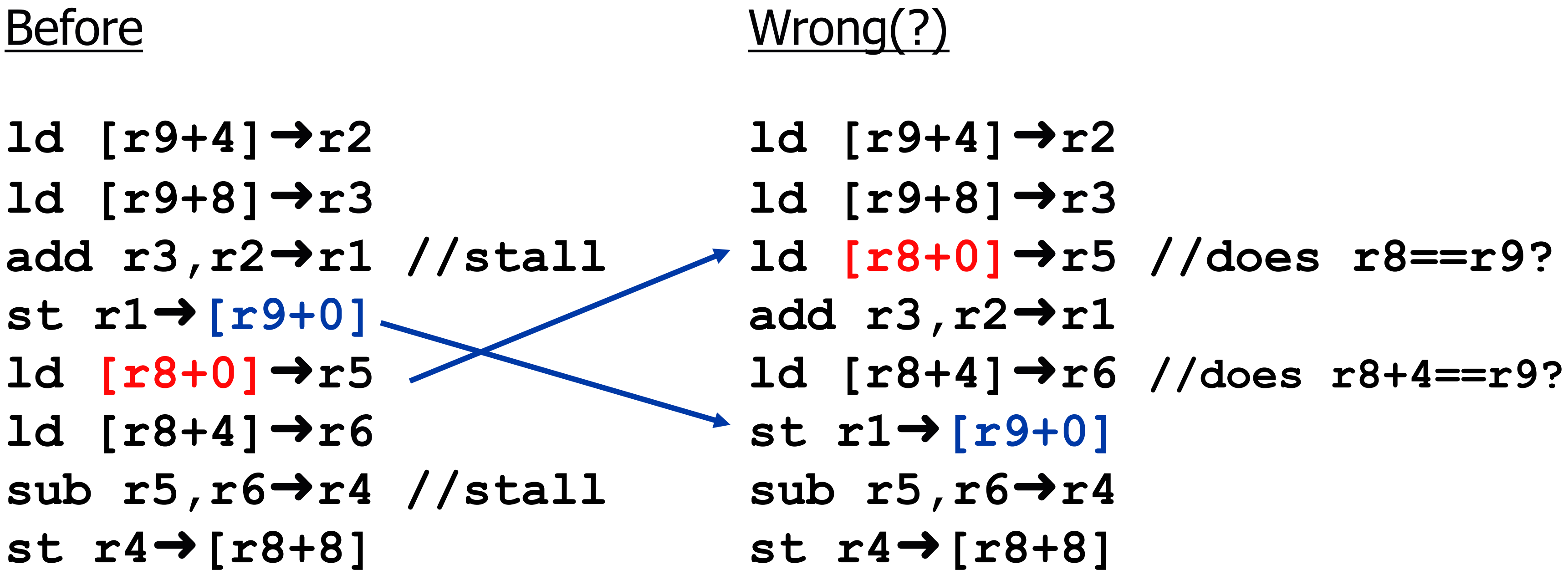
```
ld  [sp+4]→r2
ld  [sp+8]→r1
add r1,r2→r1  //stall
st  r1→[sp+0]
ld  [sp+16]→r2
ld  [sp+20]→r1
sub r2,r1→r1  //stall
st  r1→[sp+12]
```

Wrong!

```
ld  [sp+4]→r2
ld  [sp+8]→r1
ld  [sp+16]→r2
add r1,r2→r1   // wrong r2
ld  [sp+20]→r1
st  r1→[sp+0]   // wrong r1
sub r2,r1→r1
st  r1→[sp+12]
```

Compiler Scheduling Requires

- **Alias analysis**
  - Ability to tell whether load/store reference same memory locations
    - Effectively, whether load/store can be rearranged
  - Previous example: easy, loads/stores use same base register (**`sp`**)
  - New example: can compiler tell that **`r8`** != **`r9`**?
  - Must be **conservative**



Before

```
ld  [r9+4]➔r2
ld  [r9+8]➔r3
add r3,r2➔r1 //stall
st r1➔[r9+0]
ld  [r8+0]➔r5
ld  [r8+4]➔r6
sub r5,r6➔r4 //stall
st r4➔[r8+8]
```

Wrong(?)

```
ld  [r9+4]➔r2
ld  [r9+8]➔r3
ld  [r8+0]➔r5 //does r8==r9?
add r3,r2➔r1
ld  [r8+4]➔r6 //does r8+4==r9?
st r1➔[r9+0]
sub r5,r6➔r4
st r4➔[r8+8]
```

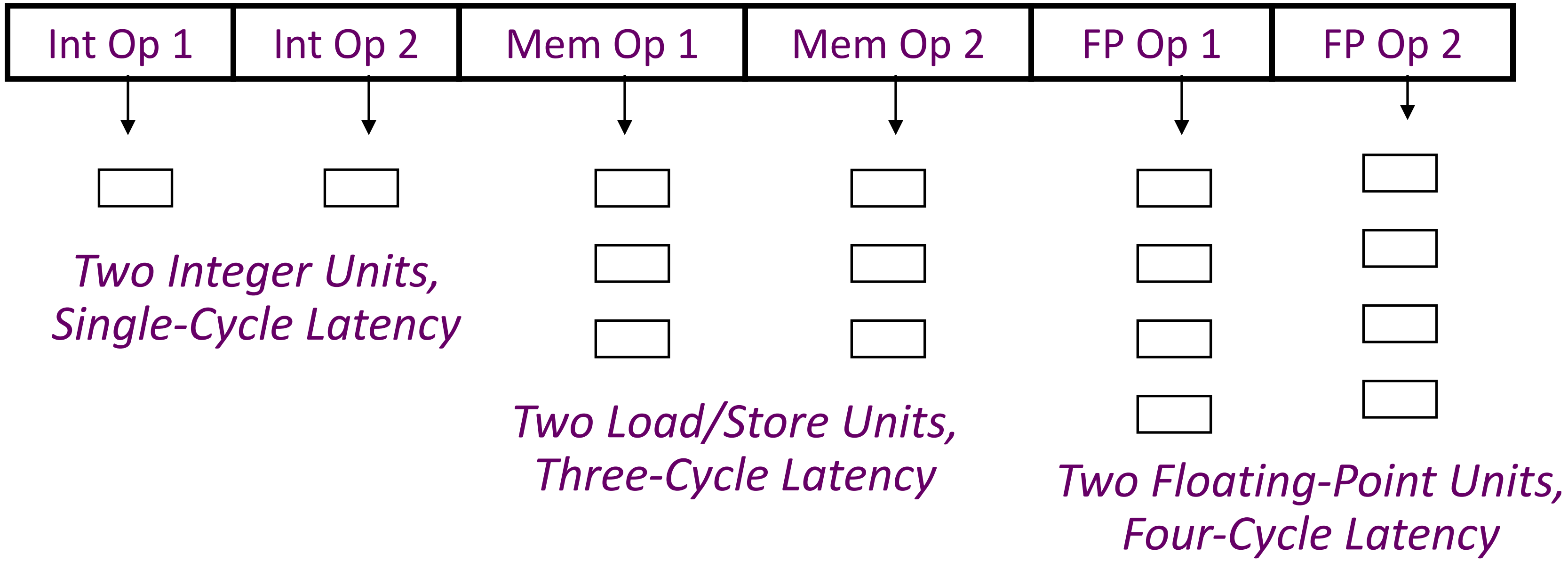
VLIW: Very Long Insn Word

- A compiler-centric technique
  - Effectively, a 1-wide pipeline, but unit is an N-insn group
  - Compiler guarantees insns within a VLIW group are independent
    - If no independent insns, slots filled with nops
  - Group travels down pipeline as a unit
  - Simpler I$/branch prediction
    - Compiler guarantees all instructions in bundle independent
  - Doesn’t help bypasses or register file
  - Not compatible across machines of different widths

VLIW Compiler Responsibilities

- Schedule operations to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedule to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs

VLIW: Very Long Instruction Word



Int Op 1	Int Op 2	Mem Op 1	Mem Op 2	FP Op 1	FP Op 2

Two Integer Units, Single-Cycle Latency

Two Load/Store Units, Three-Cycle Latency

Two Floating-Point Units, Four-Cycle Latency

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
    - Parallelism within an instruction => no cross-operation RAW check
    - No data use before data ready => no data interlocks

Multiple-Issue Implementations

- **Statically-scheduled (in-order) superscalar**
  - **What we’ve talked about thus far**
  - + Executes unmodified sequential programs
  - – Hardware must figure out what can be done in parallel
  - E.g., Pentium (2-wide), UltraSPARC (4-wide), Alpha 21164 (4-wide)
- **Very Long Instruction Word (VLIW)**
  - - **Compiler identifies independent instructions**, new ISA
  - + Hardware can be simple and perhaps lower power
  - E.g., TransMeta Crusoe (4-wide), most DSPs

Static Scheduling Limitations

- Scheduling scope
  - Example: can’t generally move memory operations past branches
- Limited number of registers (set by ISA)
- Inexact “memory aliasing” information
  - Often prevents reordering of loads above stores by compiler
- Caches misses (or any runtime event) confound scheduling
  - How can the compiler know which loads will miss vs hit?
  - Can impact the compiler’s scheduling decisions

Multiple-Issue Implementations

- Statically-scheduled (in-order) superscalar
  - **What we've talked about thus far**
  - + Executes unmodified sequential programs
  - – Hardware must figure out what can be done in parallel
  - E.g., Pentium (2-wide), UltraSPARC (4-wide), Alpha 21164 (4-wide)
- Very Long Instruction Word (VLIW)
  - – **Compiler identifies independent instructions**, new ISA
  - + Hardware can be simple and perhaps lower power
  - E.g., TransMeta Crusoe (4-wide), most DSPs
- **Dynamically-scheduled Superscalar**
  - **Hardware extracts more ILP by on-the-fly reordering**
  - Intel Atom/Core/Xeon, AMD Opteron/Ryzen, some ARM A-series

Dynamic Pipeline Scheduling (OoO)

- Allow the CPU to execute instructions out of order to avoid stalls
  - Hardware re-schedules insns…
  - But commit result to registers in order
  - As with pipelining and superscalar, ISA unchanged
    - Same hardware/software interface, appearance of in-order
- Example

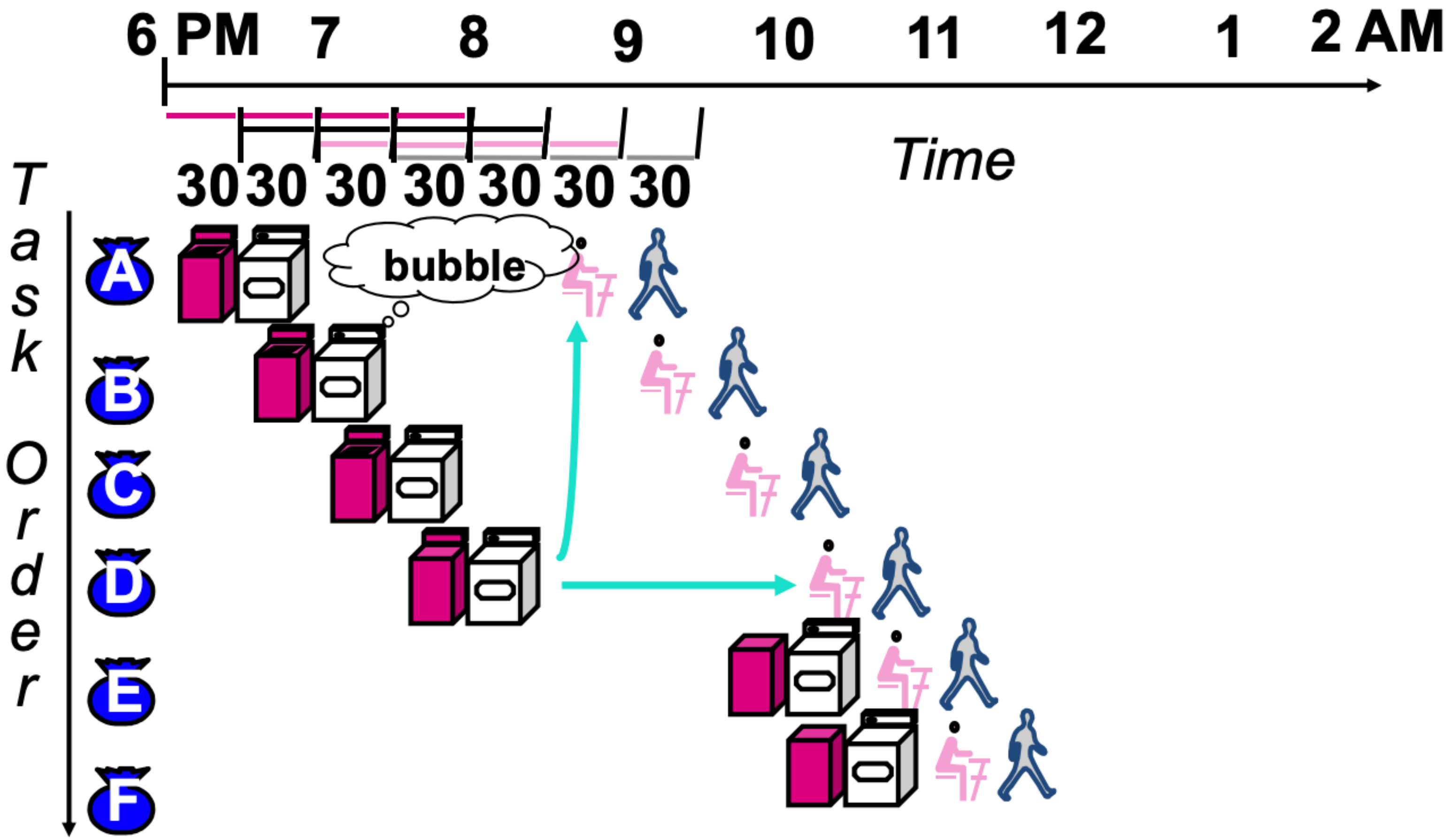
```
ld   x31,20(x21)
add  x1,x31,x2
sub  x23,x23,x3
andi x5,x23,20
```

Can start `sub` while `add` is waiting for ld

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

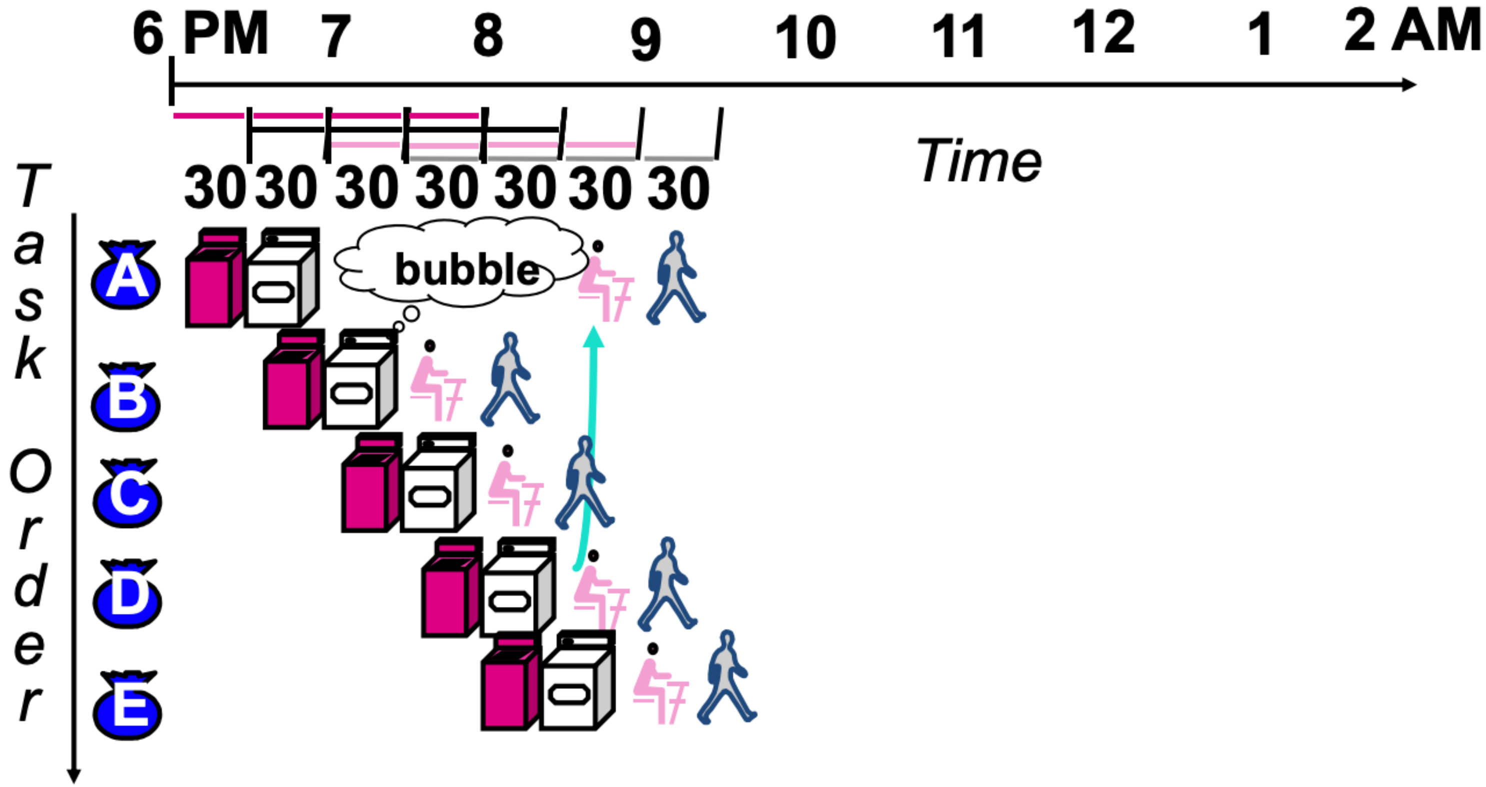
Laundry Analogy: Matching socks in later load



A depends on D; stall since folder is tied up

Image: https://inst.eecs.berkeley.edu/~cs61c/

Out of Order (OoO): Don’t wait!



A depends on D; let the rest continue

- Need more resources to allow out-of-order (2 folders)

Image: https://inst.eecs.berkeley.edu/~cs61c/

Example: In-Order Limitations #1

Note: 6-stage pipeline in this example

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [r1] → r2	F	D	X	M_1	M_2	W							
add r2 + r3 → r4	F	D	**d***	**d***	**d***	X	M_1	M_2	W				
xor r4 ^ r5 → r6		F	D	**d***	**d***	**d***	X	M_1	M_2	W			
ld [r7] → r4		F	D	**d***	**d***	**d***	X	M_1	M_2	W			

- In-order pipeline, three-cycle load-use penalty
  - 2-wide
- Why not the following?

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [r1] → r2	F	D	X	M_1	M_2	W							
add r2 + r3 → **r4**	F	D	**d***	**d***	**d***	X	M_1	M_2	W				
xor **r4** ^ r5 → r6		F	D	**d***	**d***	**d***	X	M_1	M_2	W			
ld [r7] → **r4**		F	D	**X**	**M_1**	**M_2**	**W**						

Example: In-Order Limitations #2

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [p1] → p2	F	D	X	M_1	M_2	W							
add p2 + p3 → p4	F	D	**d***	**d***	**d***	X	M_1	M_2	W				
xor p4 ^ p5 → p6		F	D	**d***	**d***	**d***	X	M_1	M_2	W			
ld [p7] → p8		F	D	**d***	**d***	**d***	X	M_1	M_2	W			

- In-order pipeline, three-cycle load-use penalty
  - 2-wide
- Why not the following:

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [p1] → p2	F	D	X	M_1	M_2	W							
add p2 + p3 → **p4**	F	D	**d***	**d***	**d***	X	M_1	M_2	W				
xor **p4** ^ p5 → p6		F	D	**d***	**d***	**d***	X	M_1	M_2	W			
ld [p7] → **p8**		F	D	**X**	**M_1**	**M_2**	**W**						

Out-of-Order Pipeline (high-level view)

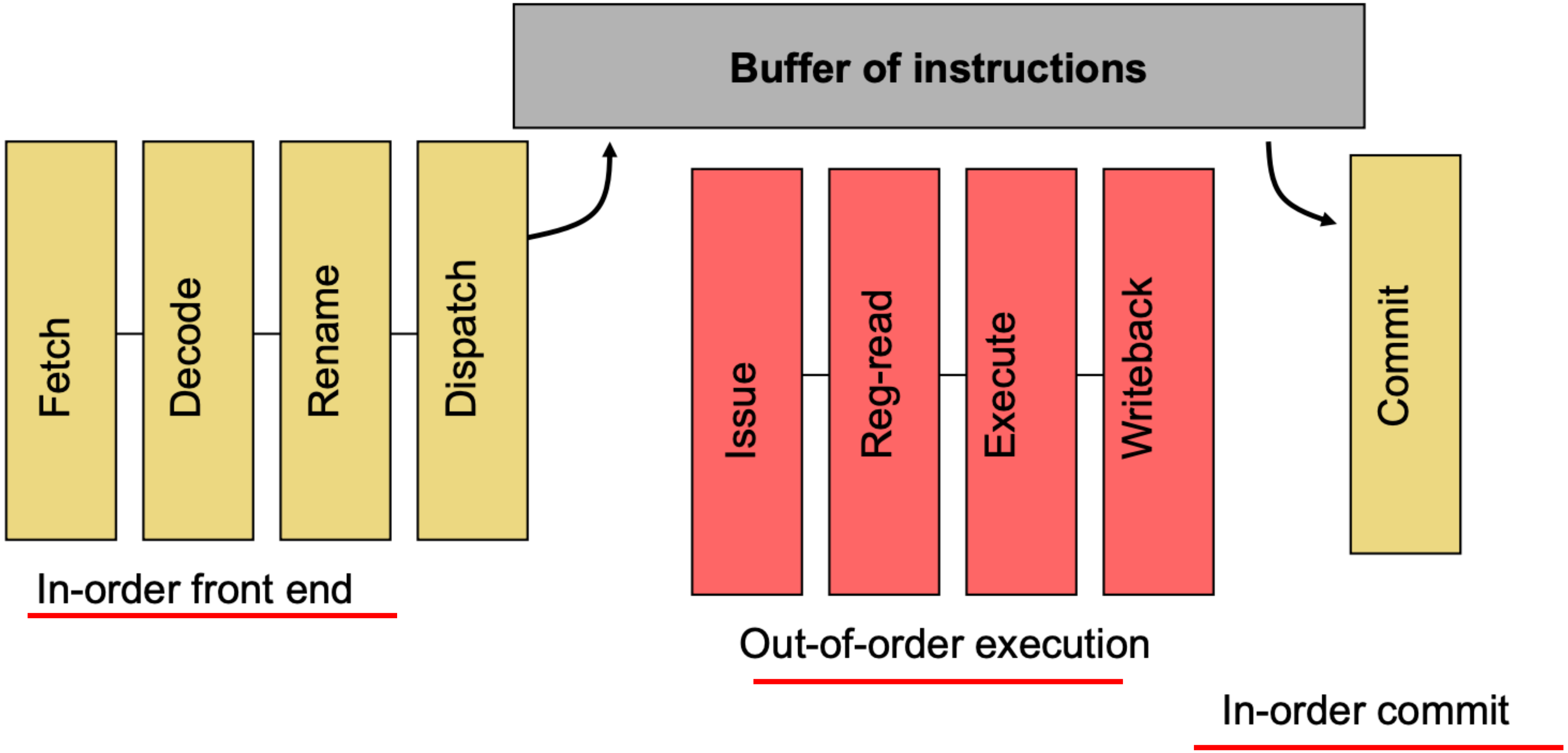
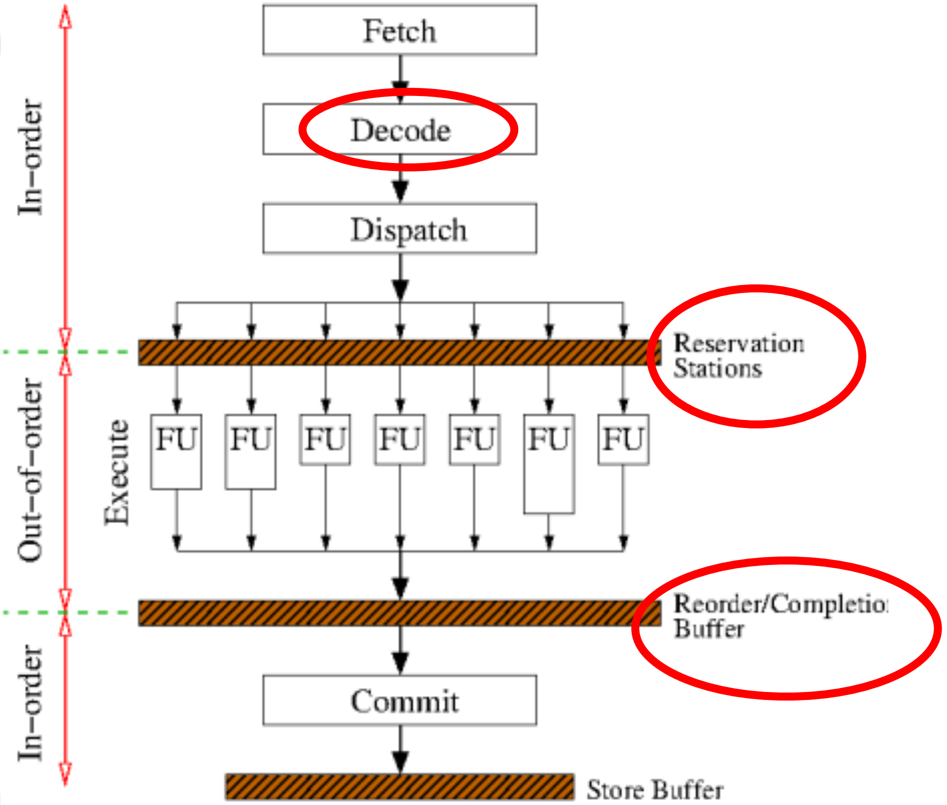


Image: cis.upenn.edu/~cis571/

Out-of-Order Pipeline (another view)

- **Fetch & decode in order**
  - **Multiple instructions are fetched/decoded in parallel**
  - **Insts. put in reservation stations (RS)**
- **Execute instructions that are ready in the reservation stations**
  - **Instruction operands must be ready**
  - **Available execution resources**
- **Following execution:**
  - **Broadcast result on bypass network**
  - **Signal all dependent instructions that data is ready**
- **Commit instructions <u>in-order</u>**
  - **Can commit an instruction only after all preceding instructions (in program order) have committed**



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
  - HW examines a sliding window of consecutive instructions
  - Each cycle, picks the next ready instruction(s)

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

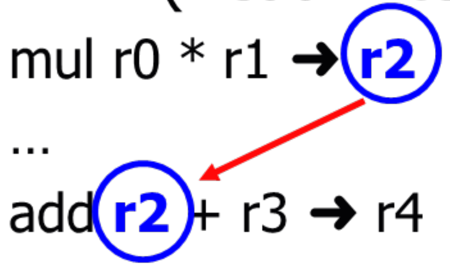
  Step #1: Register renaming – to avoid “false” dependencies

  Step #2: Dynamically schedule – to enforce “true” dependencies

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

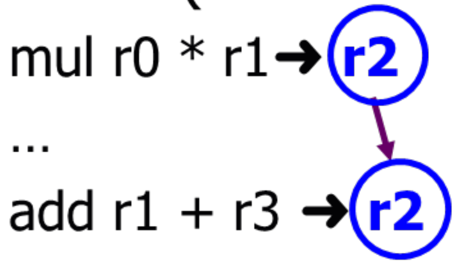
Dependence types (recap)

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



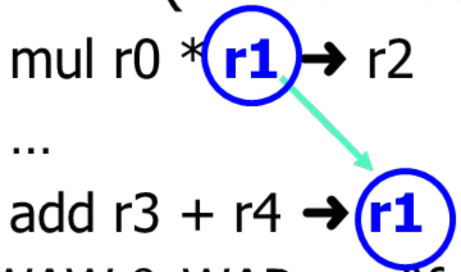
mul r0 * r1 → r2
…
add r2 + r3 → r4

- **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”

Register Renaming Algorithm

Two key data structures:

- maptable[architectural_reg] ➔ physical_reg
- Free list: allocate (new) & free registers (implemented as a queue)

Algorithm: at “decode” stage for each instruction:

```
insn.phys_input1 = maptable[insn.arch_input1]
insn.phys_input2 = maptable[insn.arch_input2]
insn.old_phys_output = maptable[insn.arch_output]
new_reg = new_phys_reg()
maptable[insn.arch_output] = new_reg
insn.phys_output = new_reg
```

At “commit”

- Once all older instructions have committed, free register

```
free_phys_reg(insn.old_phys_output)
```

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

Map table

p10

Free-list

Register Renaming Summary

- 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”

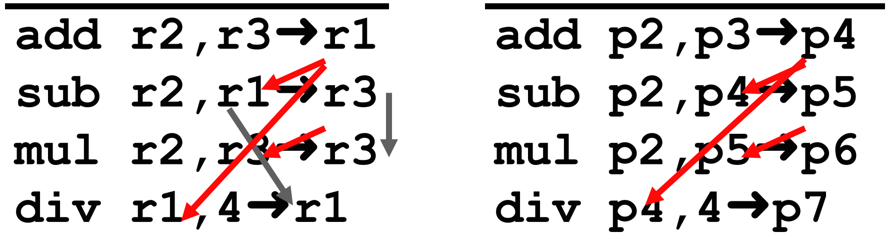
MapTable

Time

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p4	p2	p6

FreeList

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



Original insns

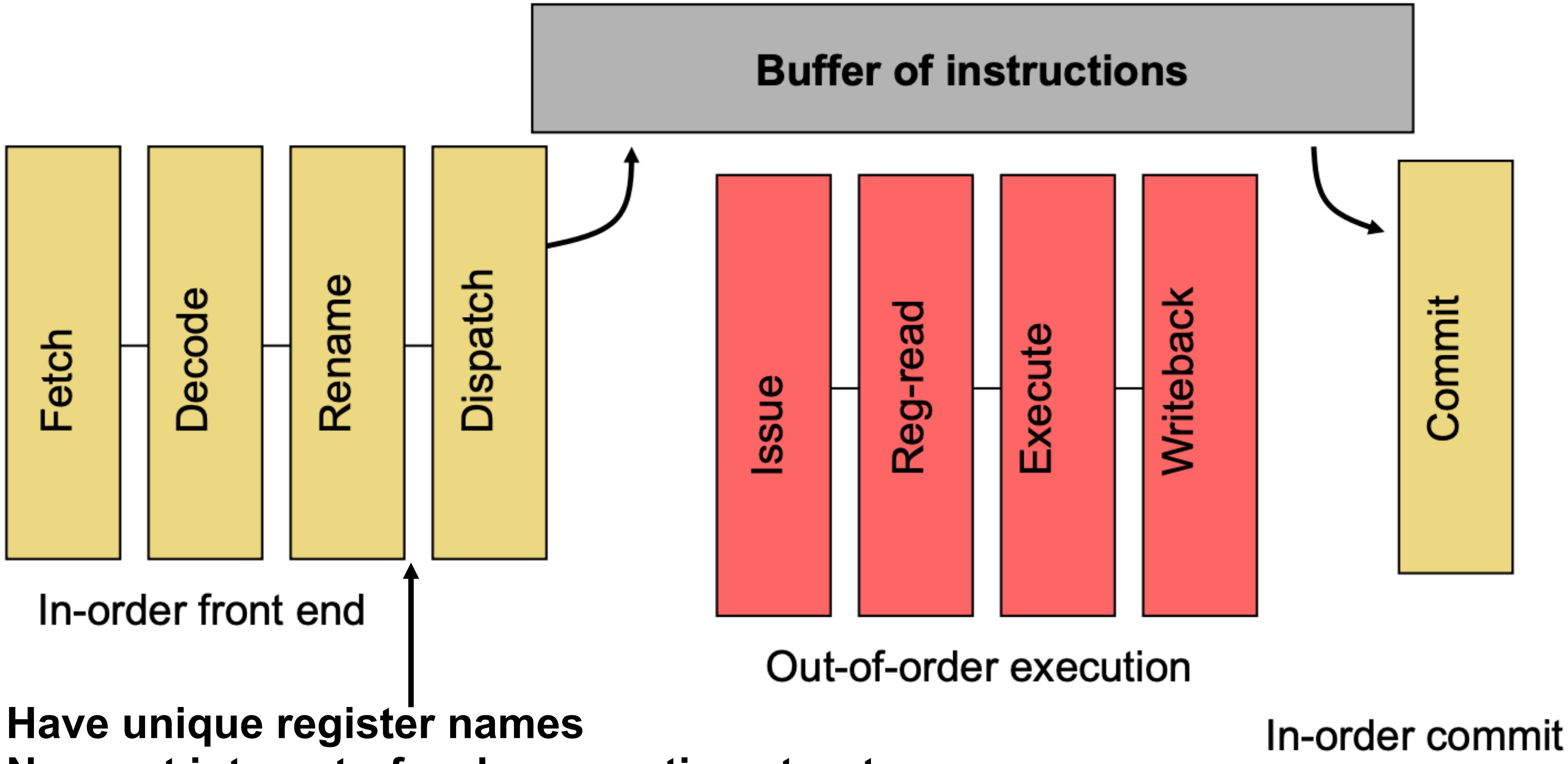
```
add r2,r3➔r1
sub r2,r1➔r3
mul r2,r3➔r3
div r1,4➔r1
```

Renamed insns

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

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

Out-of-Order Pipeline

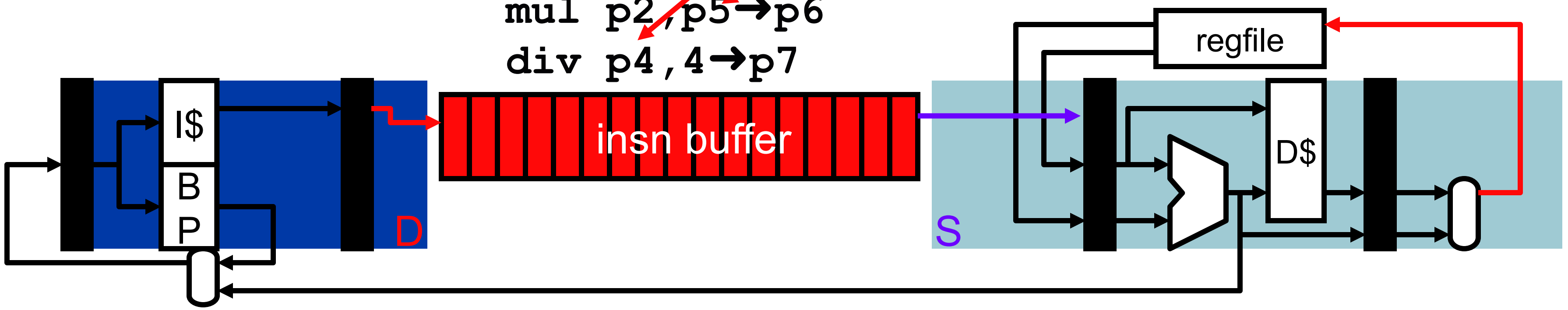


Have unique register names
Now put into out-of-order execution structures

Image: cis.upenn.edu/~cis571/

Dynamic Scheduling Overview

```
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”)

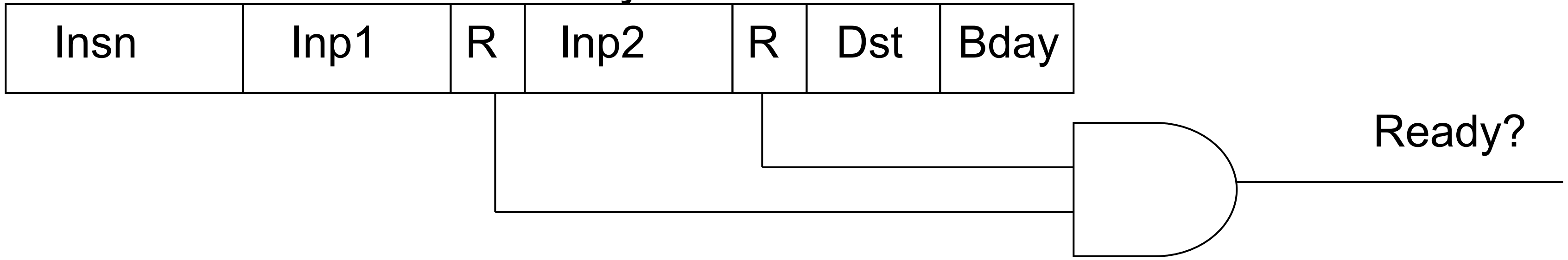
- Algorithm at “issue” stage (prior to read registers):

```
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 insn with latency of N, set “ready” bit N-1 cycles in future

Dispatch

- Put renamed instructions into out-of-order structures
- Re-order buffer (ROB)
  - Holds instructions from Fetch through Commit
- Issue Queue/Reservation Station
  - 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



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

Ready bits

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

Issue Queue/Reservation Station

Insn	Inp1	R	Inp2	R	Dst	Bday

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

Issue Queue

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

Ready bits

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

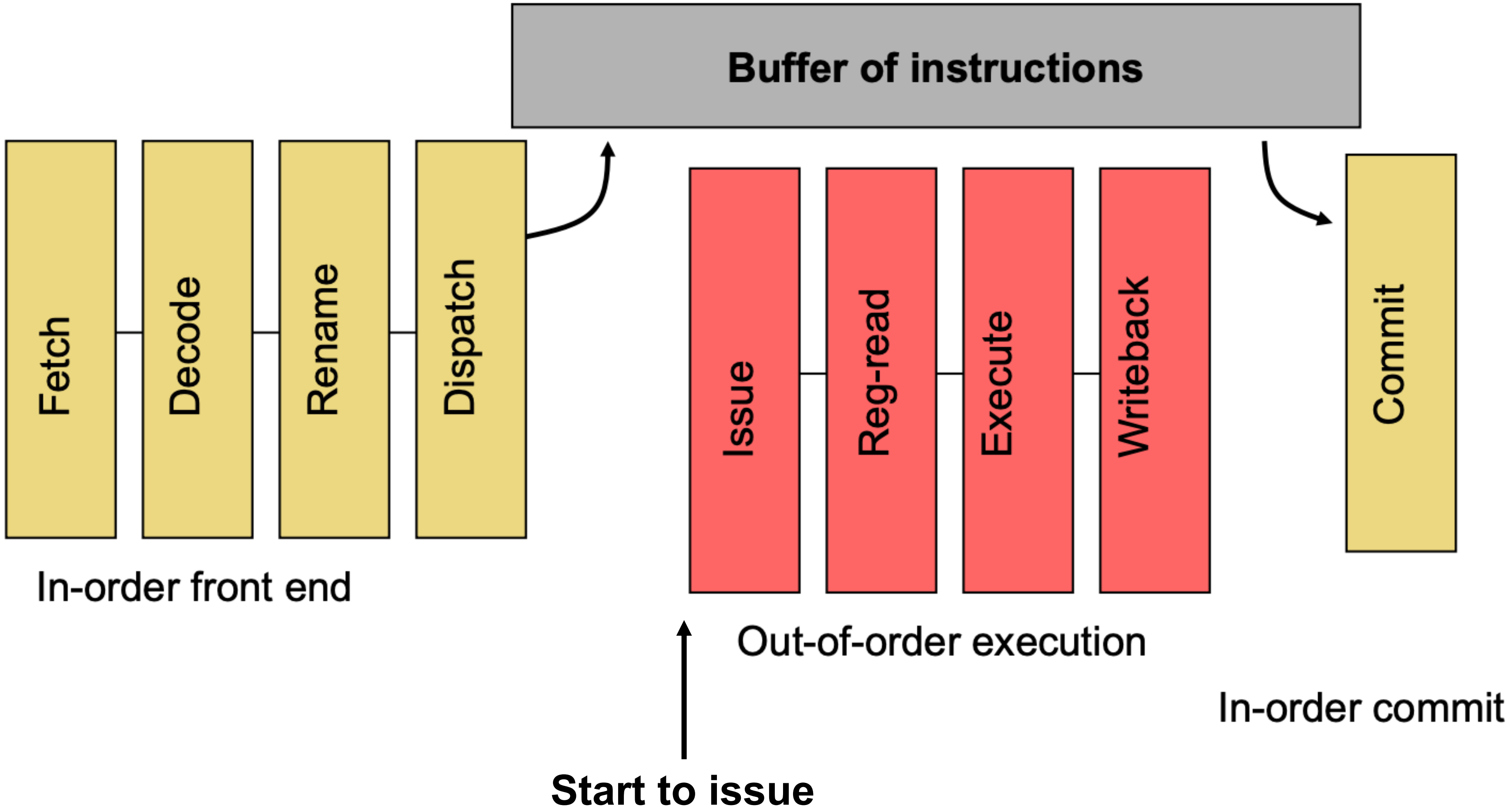
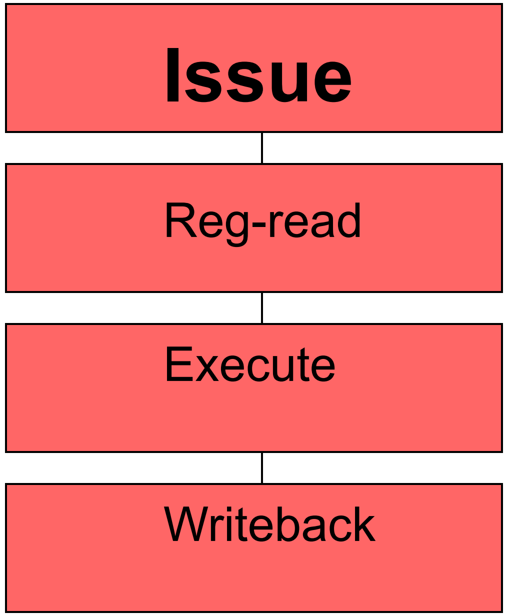


Image: cis.upenn.edu/~cis571/

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

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

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

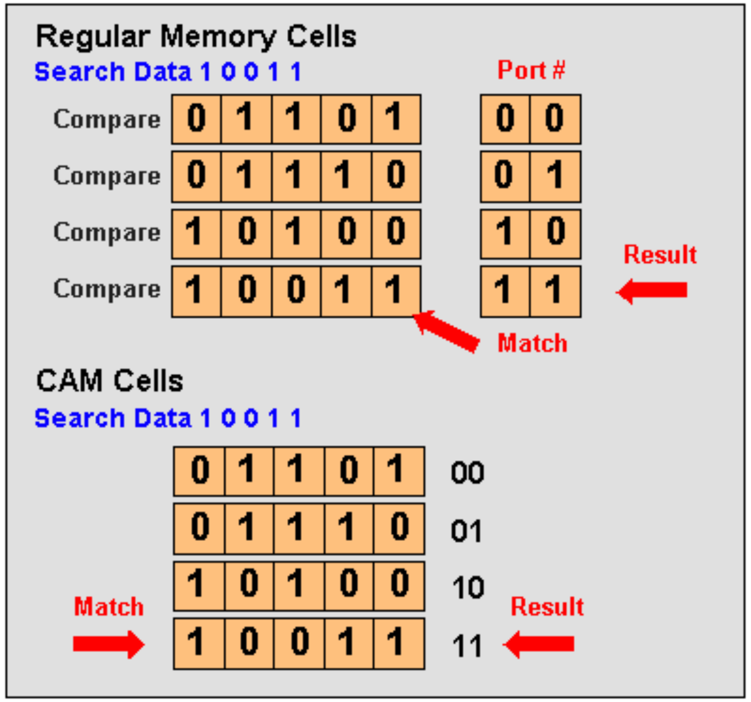
Ready bits

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

CAM: Content Addressable Memory

- A circuit that combines comparison and storage in a single device
- Send the data and the CAM looks to see if it has a copy and returns the index of the matching row
- can afford to implement much higher set associativity



More details on canvas
Files>Reading Materials>
CAM_Reading.pdf

Source: https://www.pcmag.com/encyclopedia/term/content-addressable-memory

Issue

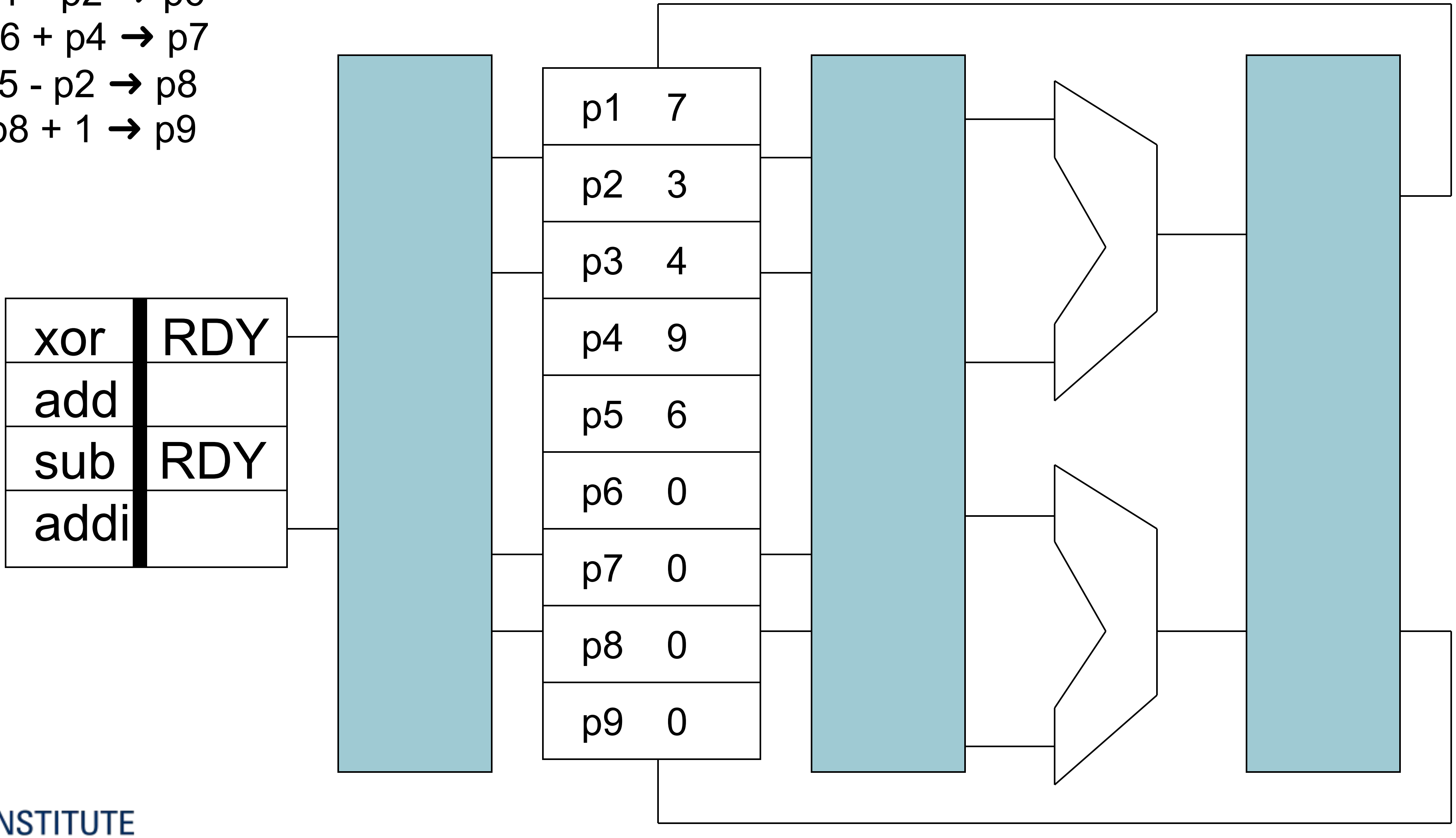
- **Select/Wakeup** one cycle
- Dependent instructions execute on back-to-back cycles
  - Next cycle: add/addi are ready:

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)

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

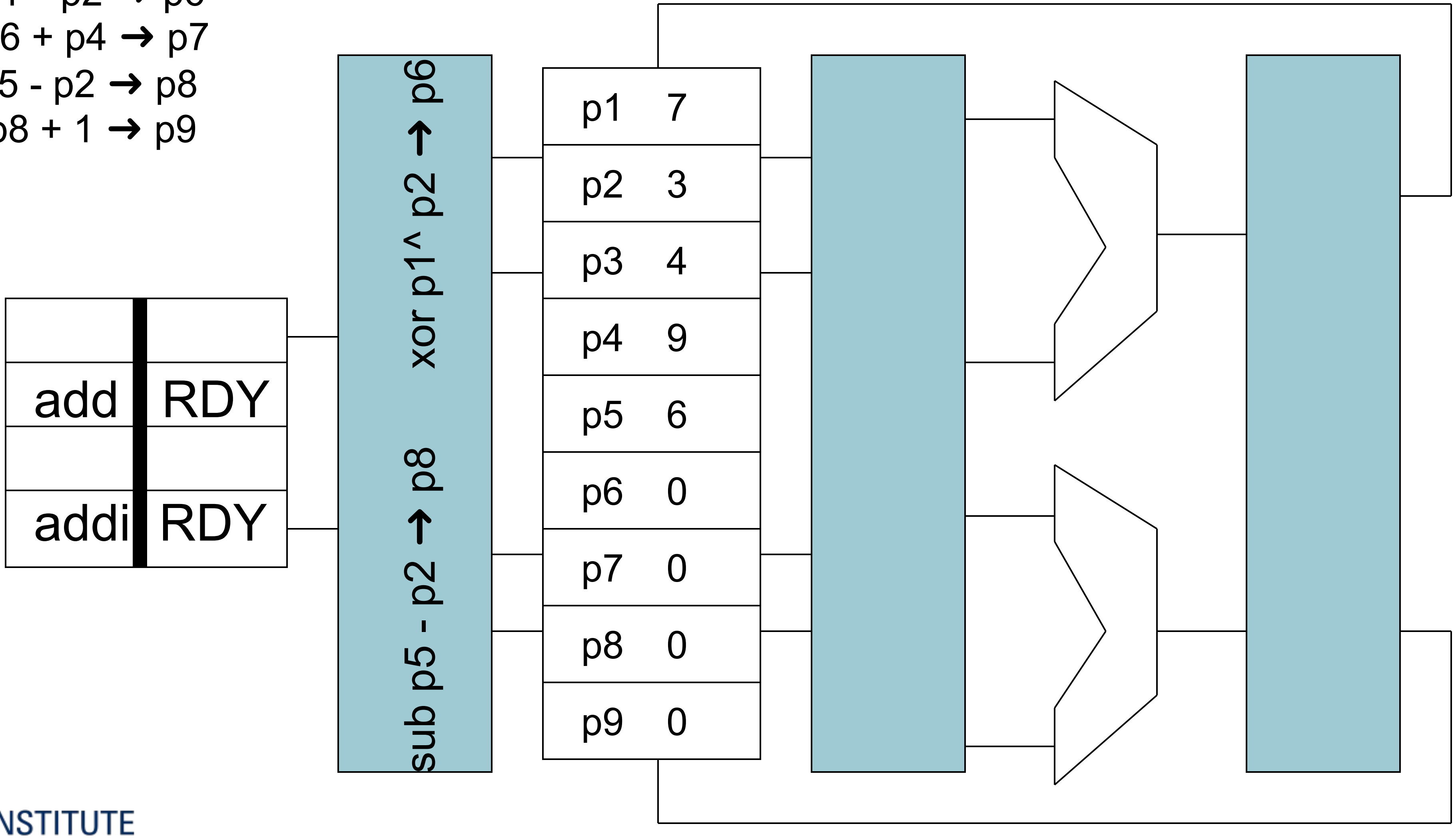


xor	RDY
add	
sub	RDY
addi	

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

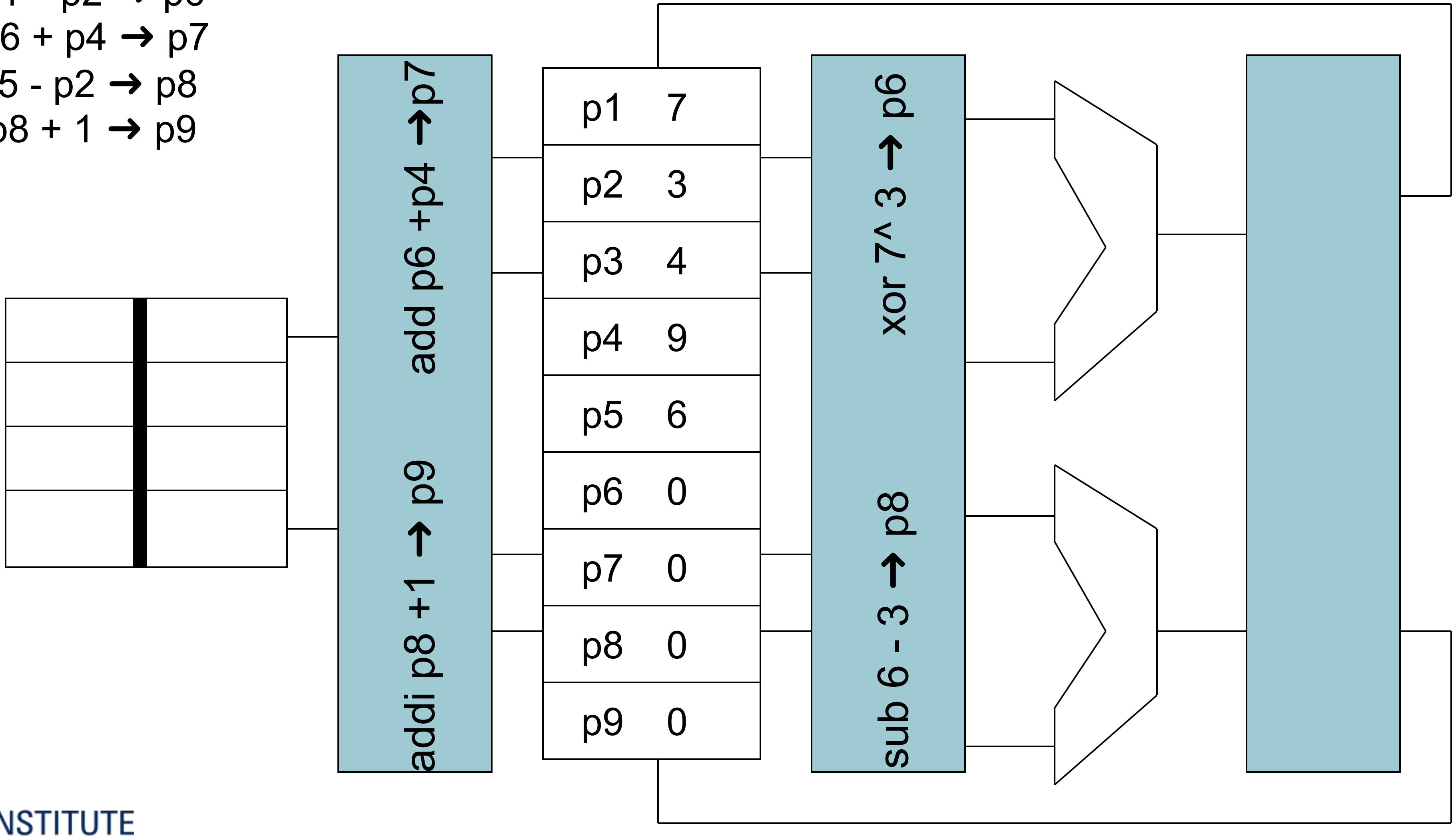
OOO execution (2-wide)

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



OOO execution (2-wide)

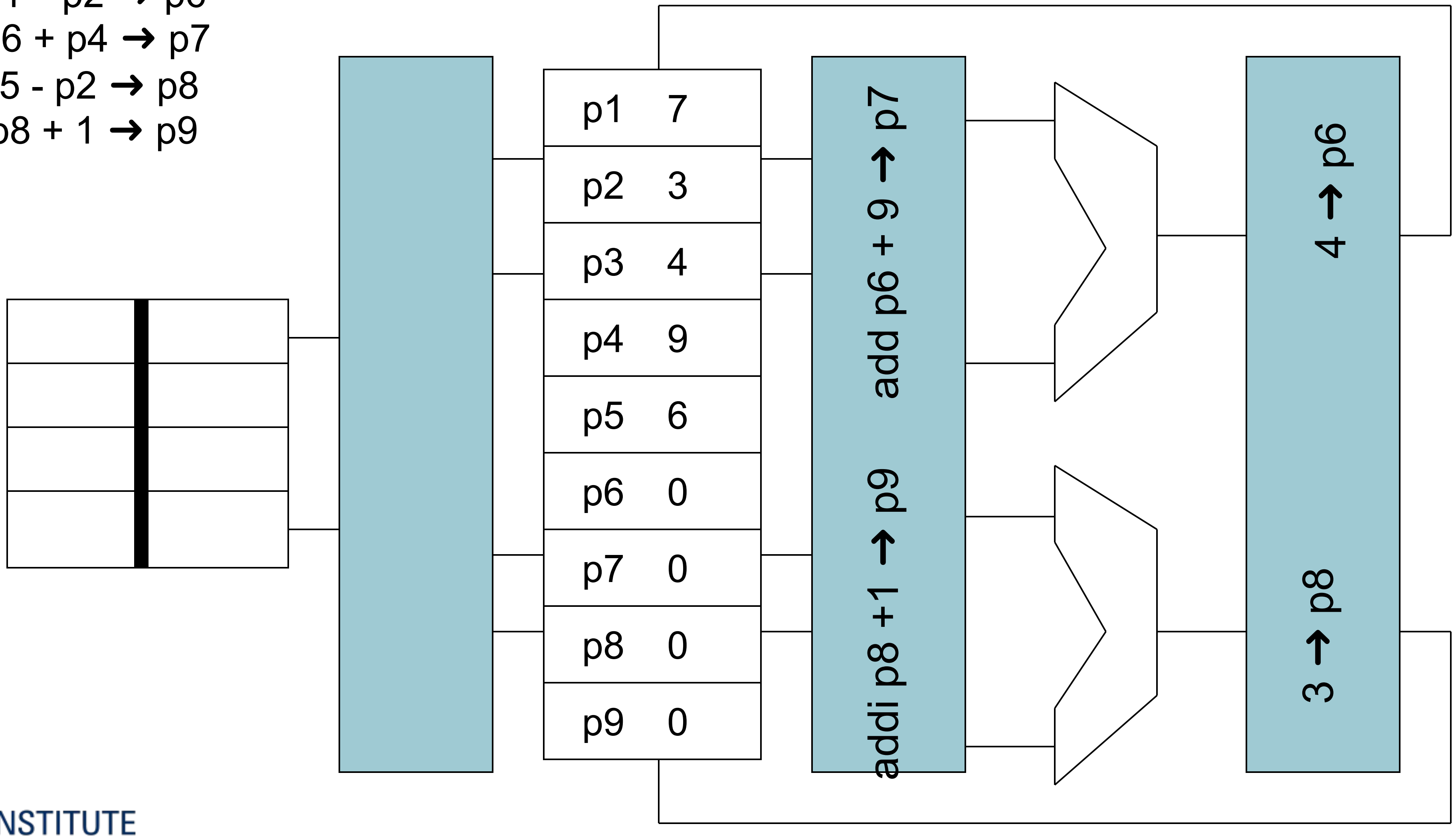
xor p1 ^ p2 ➜ p6
add p6 + p4 ➜ p7
sub p5 - p2 ➜ p8
addi p8 + 1 ➜ p9



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

OOO execution (2-wide)

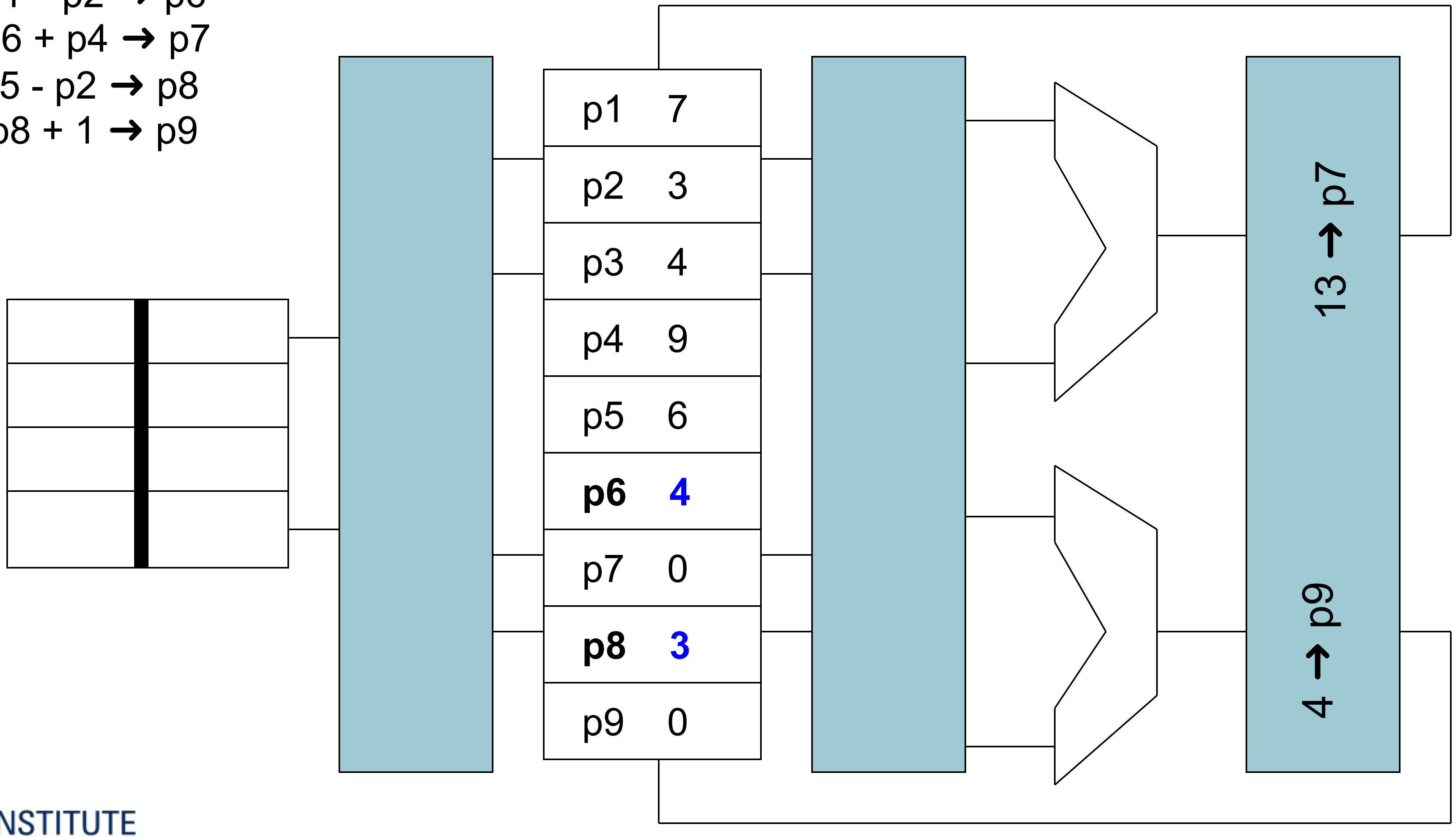
xor p1 ^ p2 ➔ p6
add p6 + p4 ➔ p7
sub p5 - p2 ➔ p8
addi p8 + 1 ➔ p9



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

OOO execution (2-wide)

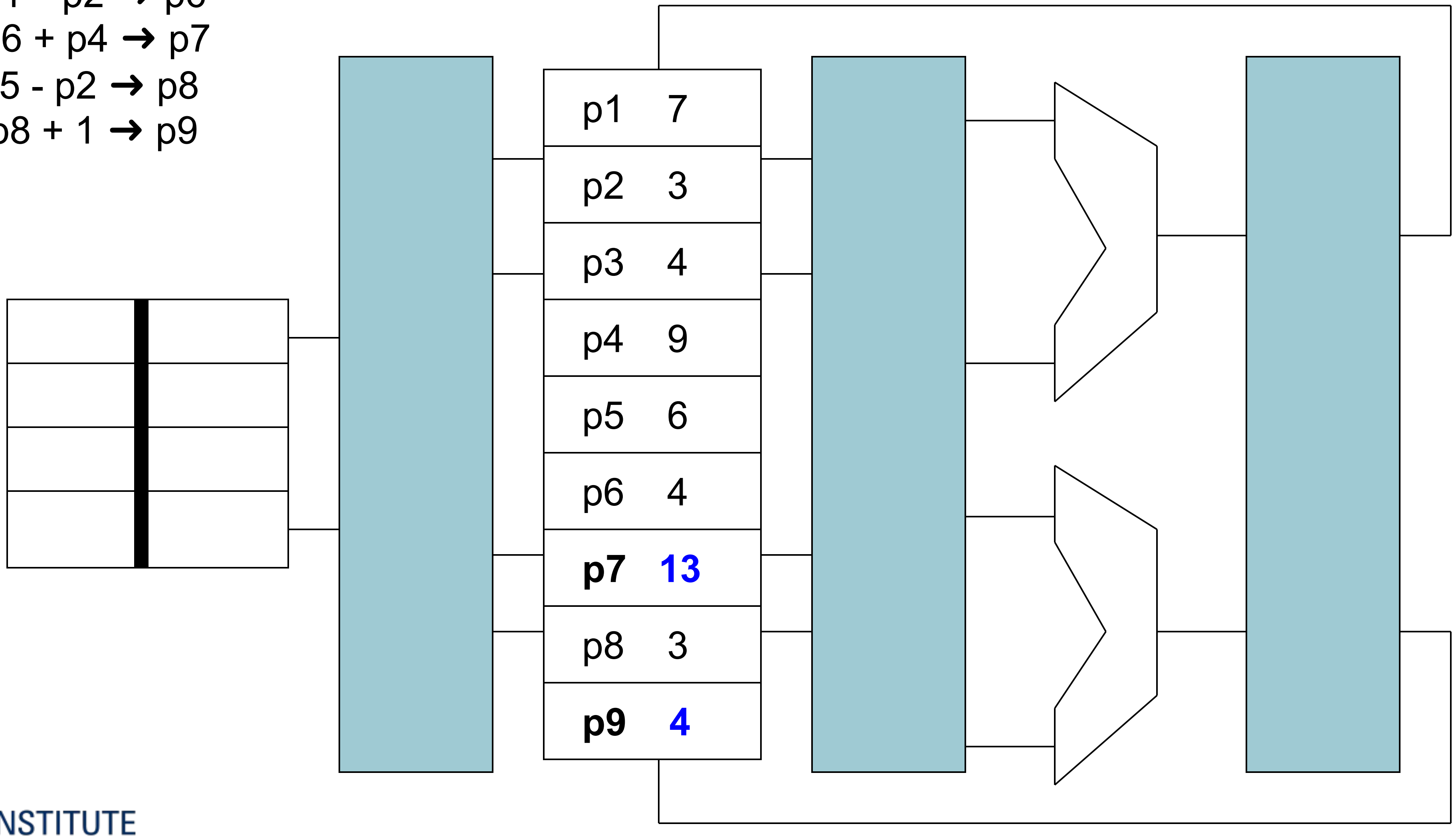
xor p1 ^ p2 ➔ p6
add p6 + p4 ➔ p7
sub p5 - p2 ➔ p8
addi p8 + 1 ➔ p9



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

OOO execution (2-wide)

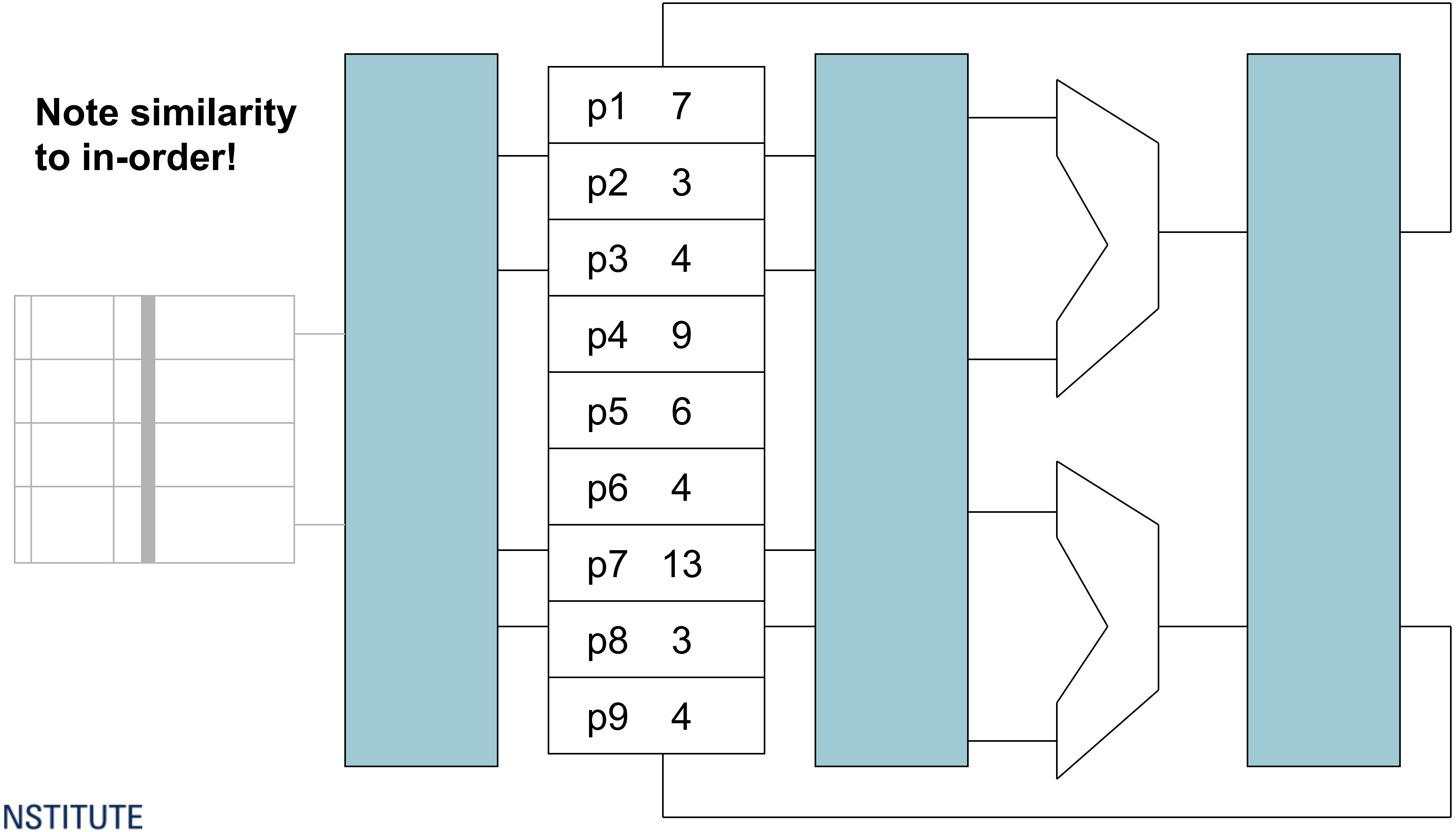
xor p1 ^ p2 ➜ p6
add p6 + p4 ➜ p7
sub p5 - p2 ➜ p8
addi p8 + 1 ➜ p9



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

Re-order Buffer (ROB)

- ROB entry holds all info for recovery/commit
  - **All instructions** & in order
  - Architectural register names, physical register names, insn type
  - Not removed until very last thing ("commit")

- Operation
  - Fetch: insert at tail (if full, stall)
  - Commit: remove from head (if not yet done, stall)

- Purpose: tracking for in-order commit
  - Maintain appearance of in-order execution
  - Needed to support:
    - **Misprediction recovery**
    - **Freeing of physical registers**

Out-of-Order Pipeline

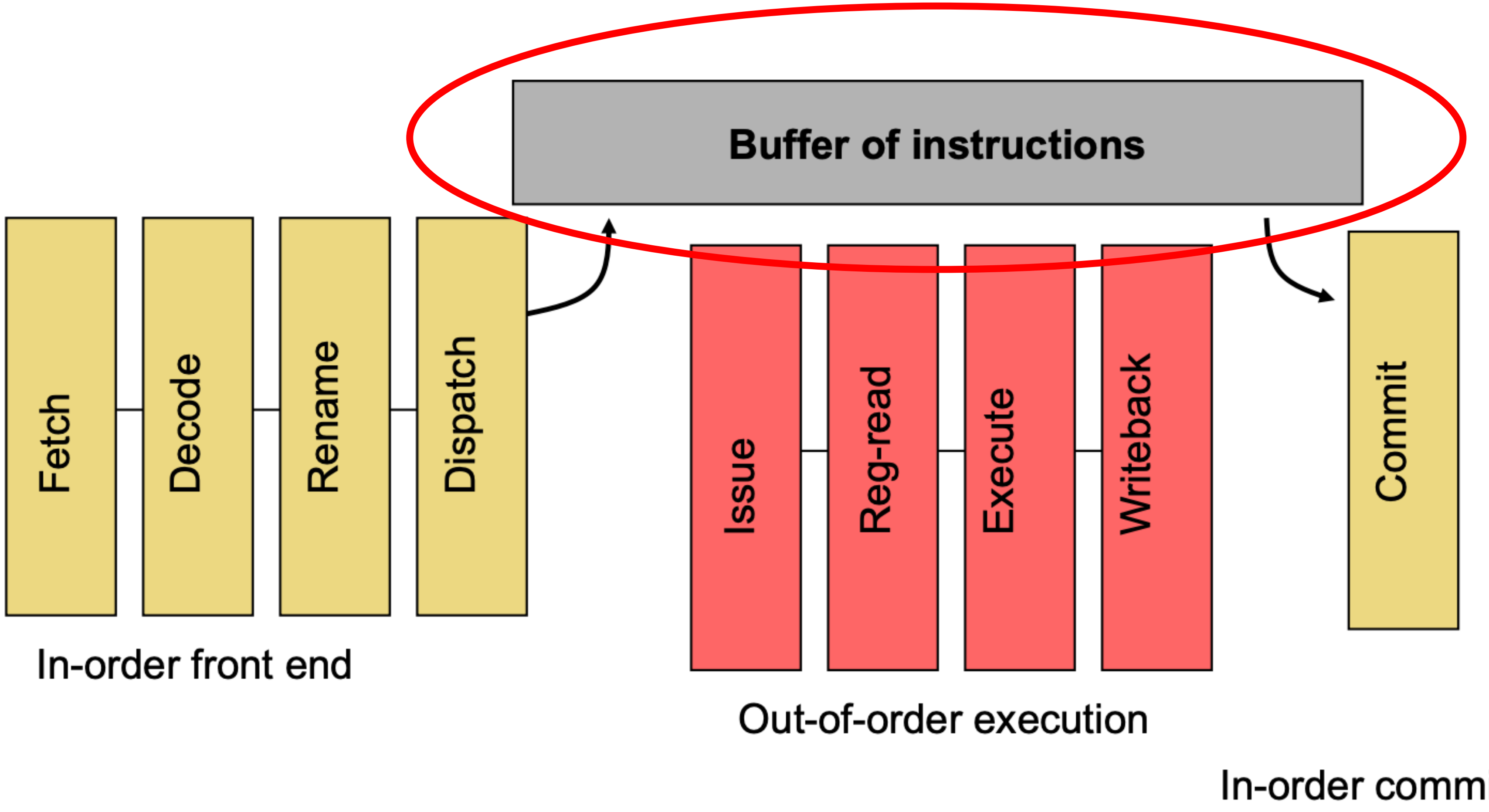


Image: cis.upenn.edu/~cis571/

Renaming revisited

- Track (or “log”) the “overwritten register” in ROB
  - Free this register at commit
  - Also used to restore the map table on “recovery”
    - Used for branch misprediction recovery

Register Renaming Algorithm (Full)

- Two key data structures:
  - maptable[architectural_reg] ➔ physical_reg
  - Free list: allocate (new) & free registers (implemented as a queue)
- Algorithm: at “decode” stage for each instruction:

```
insn.phys_input1 = maptable[insn.arch_input1]
insn.phys_input2 = maptable[insn.arch_input2]
insn.old_phys_output = maptable[insn.arch_output]
new_reg = new_phys_reg()
maptable[insn.arch_output] = new_reg
insn.phys_output = new_reg
```

- **At “commit”**
  - **Once all older instructions have committed, free register**
    `free_phys_reg(insn. old_phys_output)`

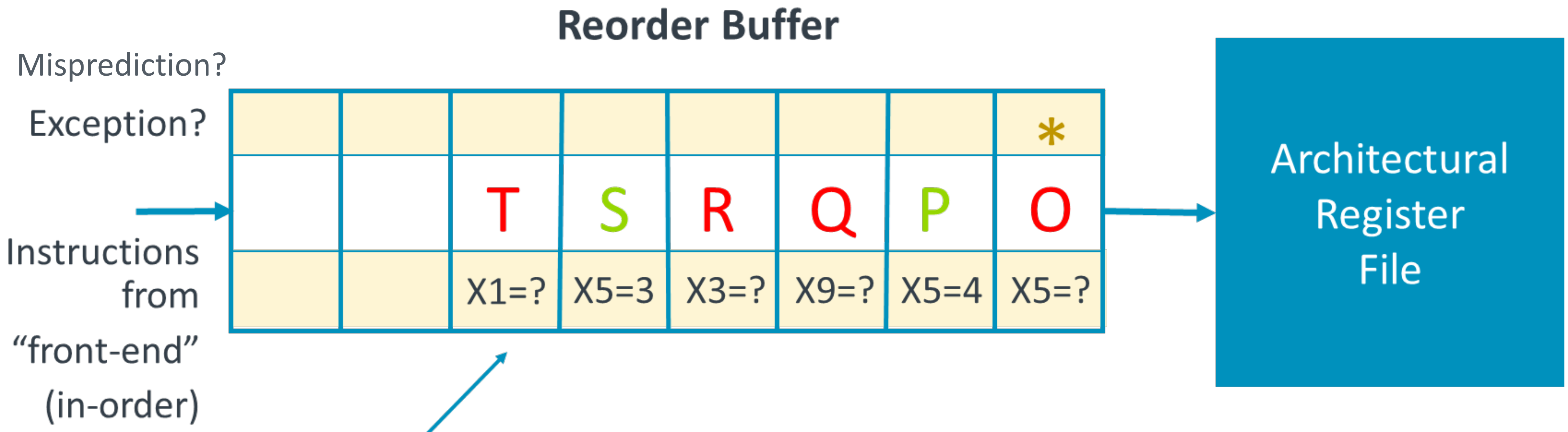
The Reorder Buffer

Committing instructions

When an instruction reaches the end of the reorder buffer, we know all earlier instructions have completed. At this point, we can:

- Update our (architectural) register file.
- Check if branches have been mispredicted.
  - If so, flush the reorder buffer and re-execute the branch.
- Check if the instruction needs to raise an exception.
  - If it does, flush the reorder buffer and raise the exception.
- Signal that store operations can write to the data cache.

The Reorder Buffer (ROB)



Instruction T executes and writes its result to its entry in the reorder buffer

Recovery

- Completely remove wrong path instructions
  - Flush from IQ
  - Remove from ROB
  - Restore map table to before misprediction
  - Free destination registers
- How to restore map table/register map?
  - Option #1: log-based reverse renaming to recover each instruction
    - Tracks the old mapping to allow it to be reversed
    - Done sequentially for each instruction (slow)
    - See next slides
  - Option #2: checkpoint-based recovery
    - Checkpoint state of maptable and free list each cycle
    - Faster recovery, but requires more state
  - Option #3: hybrid (checkpoint for branches, unwind for others)

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 ➜ [p3]

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 [p3]

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 ➜

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

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

Map table

p8
p9
p10

Free-list

Renaming 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 ➜              [ p6 ]
addi r3 + 1 ➜ r1
```

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        [ p3 ]
add p6 + p4 ➜ p7         [ p4 ]
sub p5 - p2 ➜ p8         [ p6 ]
```

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 [p3]
add p6 + p4 ➜ p7 [p4]
sub p5 - p2 ➜ p8 [p6]
addi p8 + 1 ➜ [p1]

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	[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

p10

Free-list

Recovery Example

Now, let’s use this info. to recover from a branch misprediction

Branch on Not Zero

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

p10

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

p9
p10

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

p8
p9
p10

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
xor r1 ^ r2 → r3

bnz p1, loop
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

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 read p6 (until another insn writes r3)
- 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

p10

Free-list

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

p10
p3

Free-list

Commit Example

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

p10
p3
p4

Free-list

Commit Example

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

p10
p3
p4
p6

Free-list

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

Commit Example

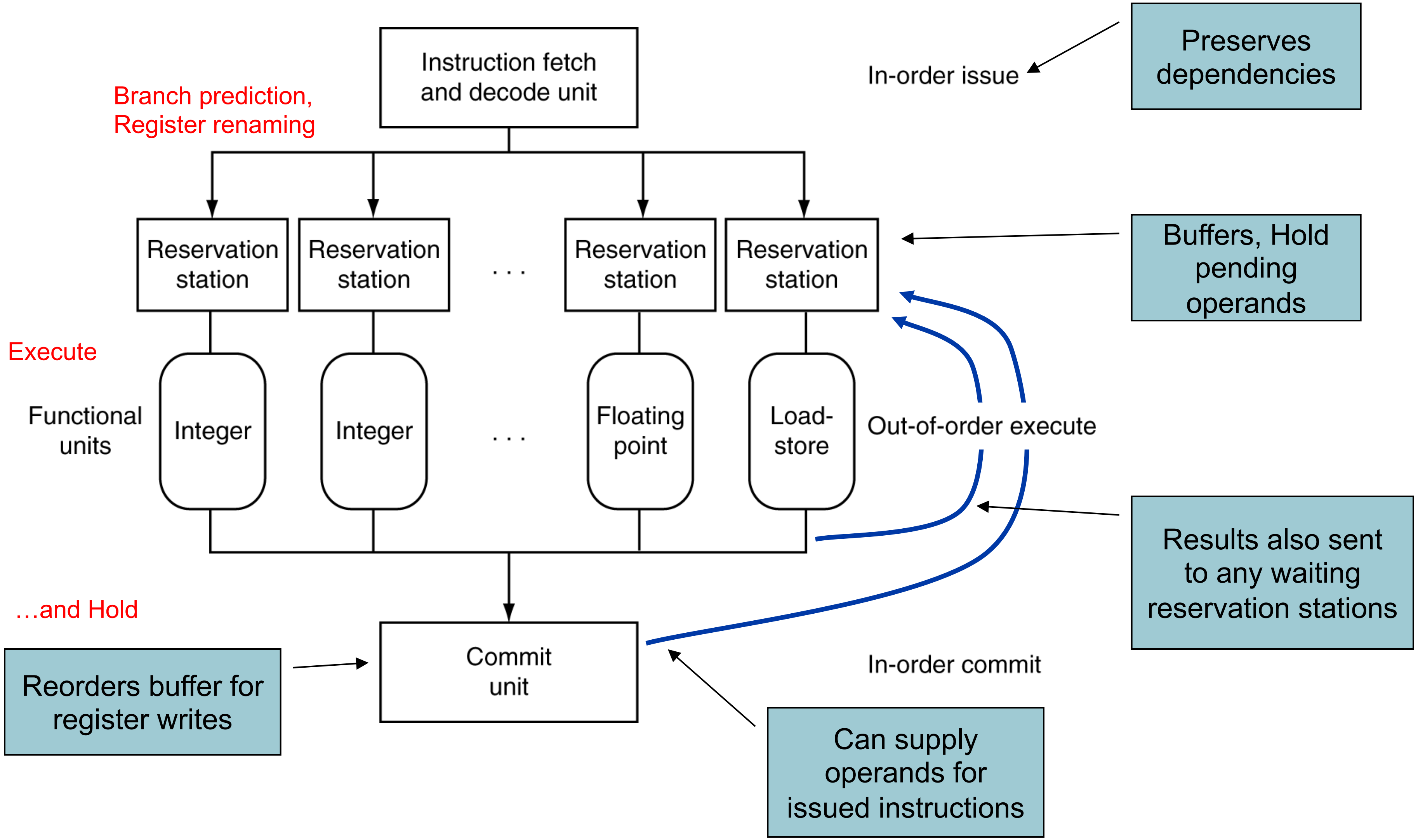
r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3
p4
p6
p1

Free-list

OoO Summary


Instruction fetch and decode unit
Branch prediction, Register renaming
In-order issue
Preserves dependencies
Reservation station
Reservation station
. . .
Reservation station
Reservation station
Buffers, Hold pending operands
Execute
Functional units
Integer
Integer
. . .
Floating point
Load-store
Out-of-order execute
Results also sent to any waiting reservation stations
…and Hold
Commit unit
In-order commit
Reorders buffer for register writes
Can supply operands for issued instructions

Out-of-Order Execution (Major Steps)

- Basically, “unroll loops” in hardware
  - Step 1: Fetch instructions in program order
  - Step 2: Predict branches as taken/untaken
  - Step 3: Register renaming – to avoid “false” dependencies
  - Step 4: Collection of renamed instructions might execute in a **window**
  - Step 5: Execute instructions with ready operands in 1 of multiple functional units
  - Step 6: Buffer results of executed instructions until predicted branches are resolved in **reorder buffer**
  - Step 7: If predicted branch correctly, **commit** results in program order
  - Step 8: If predicted branch incorrectly, discard all dependent results and start with correct PC

OoO Summary

- 3 major units operating in parallel:
  - Instruction fetch and issue unit
    - Issues instructions in program order
  - Many parallel functional (execution) units
    - Each unit has an input buffer called a Reservation Station
    - Holds operands and records the operation
    - Can execute instructions out-of-order (OOO)
  - Commit unit
    - Saves results from functional units in Reorder Buffers
    - Stores results once branch resolved so OK to execute
    - Commits results in program order

Important Concepts (So far)

- Static Scheduling
- Dynamic Scheduling
- OoO
- Register Renaming
- Dispatch
- Issue
- Commit

Dynamic Scheduling Example

Dynamic Scheduling Example

- The following slides are a detailed but concrete example
- Yet, it contains enough detail to be overwhelming
  - Try not to worry about the details
- Focus on the big picture:

Hardware can reorder instructions
to extract instruction-level parallelism

Recall: Motivating Example

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [p1] → p2	F	Di	I	RR	X	M_1	M_2	W	C				
add p2 + p3 → p4	F	Di				I	RR	X	W	C			
xor p4 ^ p5 → p6		F	Di				I	RR	X	W	C		
ld [p7] → p8		F	Di	I	RR	X	M_1	M_2	W		C		

- How would this execution occur cycle-by-cycle?
- Execution latencies assumed in this example:
  - Loads have two-cycle load-to-use penalty
    - Three cycle total execution latency
  - All other instructions have single-cycle execution latency
- "Issue queue": hold all waiting (un-executed) instructions
  - Holds ready/not-ready status
  - Faster than looking up in ready table each cycle

Out-of-Order Pipeline – Cycle 0

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F												
add r2 + r3 → r4	F												
xor r4 ^ r5 → r6													
ld [r7] → r4													

Map Table

r1	p8
r2	p7
r3	p6
r4	p5
r5	p4
r6	p3
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	---
p10	---
p11	---
p12	---

Reorder Buffer

Insn	To Free	Done?
ld		no
add		no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy

Out-of-Order Pipeline – Cycle 1a

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di											
add r2 + r3 → r4	F												
xor r4 ^ r5 → r6													
ld [r7] → r4													

Map Table

r1	p8
r2	p9
r3	p6
r4	p5
r5	p4
r6	p3
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	---
p11	---
p12	---

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add		no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
ld	p8	yes	---	yes	p9	0

Out-of-Order Pipeline – Cycle 1b

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di											
add r2 + r3 → r4	F	Di											
xor r4 ^ r5 → r6													
ld [r7] → r4													

Map Table

r1	p8
r2	p9
r3	p6
r4	p10
r5	p4
r6	p3
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	---
p12	---

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
ld	p8	yes	---	yes	p9	0
add	p9	no	p6	yes	p10	1

Out-of-Order Pipeline – Cycle 1c

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di											
add r2 + r3 → r4	F	Di											
xor r4 ^ r5 → r6		F											
ld [r7] → r4		F											

Map Table

r1	p8
r2	p9
r3	p6
r4	p10
r5	p4
r6	p3
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	---
p12	---

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor		no
ld		no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
ld	p8	yes	---	yes	p9	0
add	p9	no	p6	yes	p10	1

Out-of-Order Pipeline – Cycle 2a

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I										
add r2 + r3 → r4	F	Di											
xor r4 ^ r5 → r6		F											
ld [r7] → r4		F											

Map Table

r1	p8
r2	p9
r3	p6
r4	p10
r5	p4
r6	p3
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	---
p12	---

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor		no
ld		no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
ld	p8	yes	---	yes	p9	0
add	p9	no	p6	yes	p10	1

Out-of-Order Pipeline – Cycle 2b

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I										
add r2 + r3 → r4	F	Di											
xor r4 ^ r5 → r6		F	Di										
ld [r7] → r4		F											

Map Table

r1	p8
r2	p9
r3	p6
r4	p10
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	no
p12	---

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld		no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
add	p9	no	p6	yes	p10	1
xor	p10	no	p4	yes	p11	2

Out-of-Order Pipeline – Cycle 2c

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I										
add r2 + r3 → r4	F	Di											
xor r4 ^ r5 → r6		F	Di										
ld [r7] → r4		F	Di										

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	no
p12	no

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
add	p9	no	p6	yes	p10	1
xor	p10	no	p4	yes	p11	2
ld	p2	yes	---	yes	p12	3

Out-of-Order Pipeline – Cycle 3

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR									
add r2 + r3 → r4	F	Di											
xor r4 ^ r5 → r6		F	Di										
ld [r7] → r4		F	Di	I									

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	no
p10	no
p11	no
p12	no

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~		~~yes~~	~~p9~~	~~0~~
add	p9	no	p6	yes	p10	1
xor	p10	no	p4	yes	p11	2
ld	p2	yes	---	yes	p12	3

Issue = Select + Wakeup

Out-of-Order Pipeline – Cycle 4

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X								
add r2 + r3 → r4	F	Di											
xor r4 ^ r5 → r6		F	Di										
ld [r7] → r4		F	Di	I	RR								

[Slide #65]
For multi-cycle operations (loads, floating point)

- Wakeup deferred a few cycles
- Include checks to avoid structural hazards

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	yes
p10	no
p11	no
p12	no

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
add	p9	yes	p6	yes	p10	1
xor	p10	no	p4	yes	p11	2
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 5a

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X	M_1							
add r2 + r3 → r4	F	Di				I							
xor r4 ^ r5 → r6		F	Di										
ld [r7] → r4		F	Di	I	RR	X							

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	yes
p10	yes
p11	no
p12	no

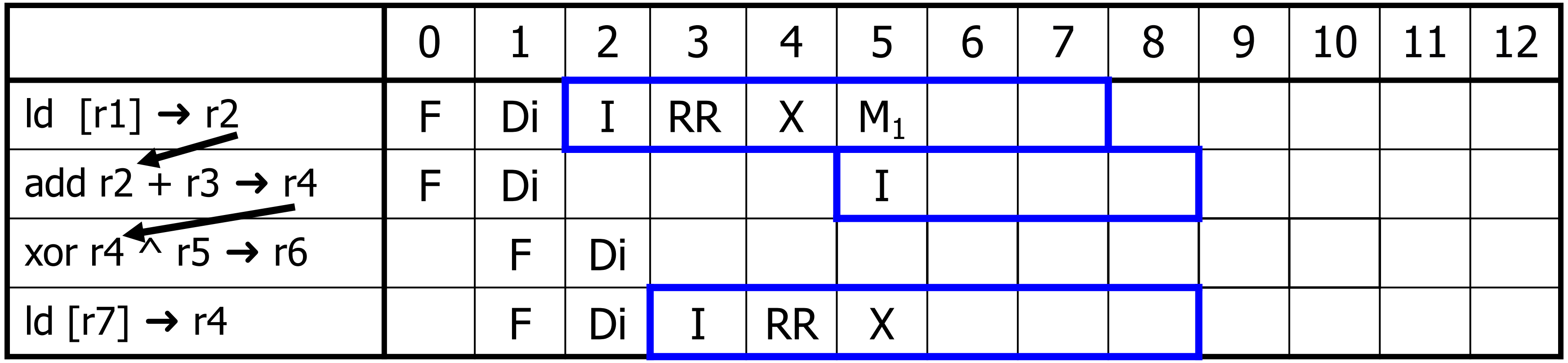
Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
add	p9	yes	p6	yes	p10	1
xor	p10	yes	p4	yes	p11	2
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 5b



	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X	M_1							
add r2 + r3 → r4	F	Di				I							
xor r4 ^ r5 → r6		F	Di										
ld [r7] → r4		F	Di	I	RR	X							

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	yes
p10	yes
p11	no
p12	yes

Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
xor	p10	yes	p4	yes	p11	2
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 6

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X	M_1	M_2						
add r2 + r3 → r4	F	Di				I	RR						
xor r4 ^ r5 → r6		F	Di				I						
ld [r7] → r4		F	Di	I	RR	X	M_1						

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

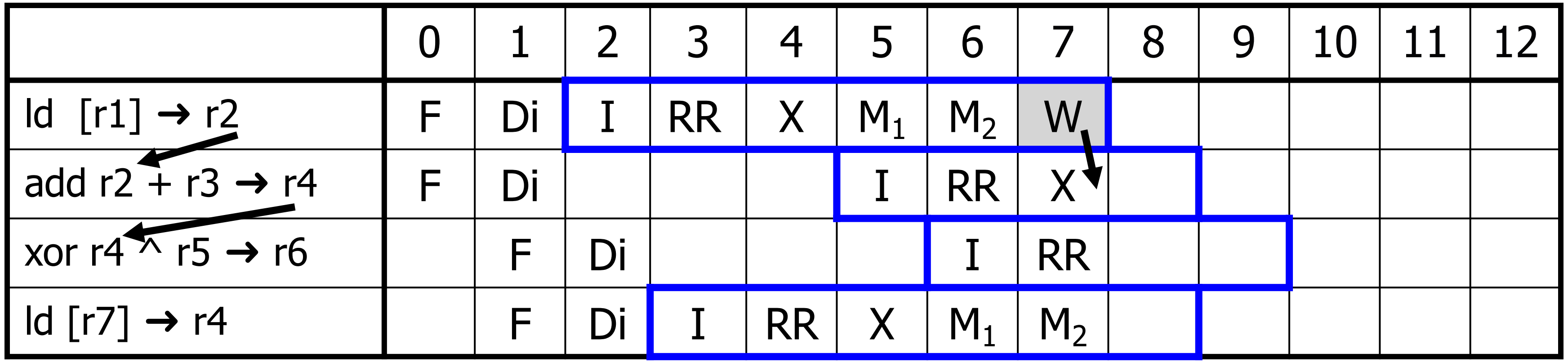
Reorder Buffer

Insn	To Free	Done?
ld	p7	no
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
xor	p10	yes	p4	yes	p11	2
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 7



	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X	M_1	M_2	W					
add r2 + r3 → r4	F	Di				I	RR	X					
xor r4 ^ r5 → r6		F	Di				I	RR					
ld [r7] → r4		F	Di	I	RR	X	M_1	M_2					

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	yes
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
ld	p7	yes
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 8a

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 → r4	F	Di				I	RR	X					
xor r4 ^ r5 → r6		F	Di				I	RR					
ld [r7] → r4		F	Di	I	RR	X	M_1	M_2					

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	---
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
add	p5	no
xor	p3	no
ld	p10	no

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 8b

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X	M$_1$	M$_2$	W	C				
add r2 + r3 → r4	F	Di				I	RR	X	W				
xor r4 ^ r5 → r6		F	Di				I	RR	X				
ld [r7] → r4		F	Di	I	RR	X	M$_1$	M$_2$	W				

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	yes
p6	yes
p7	---
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
add	p5	yes
xor	p3	no
ld	p10	yes

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 9a

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 → r4	F	Di				I	RR	X	W	C			
xor r4 ^ r5 → r6		F	Di				I	RR	X				
ld [r7] → r4		F	Di	I	RR	X	M_1	M_2	W				

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	---
p6	yes
p7	---
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
~~add~~	~~p5~~	~~yes~~
xor	p3	no
ld	p10	yes

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 9b

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 → r4	F	Di				I	RR	X	W	C			
xor r4 ^ r5 → r6		F	Di				I	RR	X	W			
ld [r7] → r4		F	Di	I	RR	X	M_1	M_2	W				

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	yes
p4	yes
p5	---
p6	yes
p7	---
p8	yes
p9	yes
p10	yes
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
~~add~~	~~p5~~	~~yes~~
xor	p3	yes
ld	p10	yes

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Cycle 10

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 → r4	F	Di				I	RR	X	W	C			
xor r4 ^ r5 → r6		F	Di				I	RR	X	W	C		
ld [r7] → r4		F	Di	I	RR	X	M_1	M_2	W		C		

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	---
p4	yes
p5	---
p6	yes
p7	---
p8	yes
p9	yes
p10	---
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
~~add~~	~~p5~~	~~yes~~
~~xor~~	~~p3~~	~~yes~~
~~ld~~	~~p10~~	~~yes~~

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Out-of-Order Pipeline – Done!

	0	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1] → r2	F	Di	I	RR	X	M_1	M_2	W	C				
add r2 + r3 → r4	F	Di				I	RR	X	W	C			
xor r4 ^ r5 → r6		F	Di				I	RR	X	W	C		
ld [r7] → r4		F	Di	I	RR	X	M_1	M_2	W		C		

Map Table

r1	p8
r2	p9
r3	p6
r4	p12
r5	p4
r6	p11
r7	p2
r8	p1

Ready Table

p1	yes
p2	yes
p3	---
p4	yes
p5	---
p6	yes
p7	---
p8	yes
p9	yes
p10	---
p11	yes
p12	yes

Reorder Buffer

Insn	To Free	Done?
~~ld~~	~~p7~~	~~yes~~
~~add~~	~~p5~~	~~yes~~
~~xor~~	~~p3~~	~~yes~~
~~ld~~	~~p10~~	~~yes~~

Issue Queue

Insn	Src1	R?	Src2	R?	Dest	Bdy
~~ld~~	~~p8~~	~~yes~~	~~---~~	~~yes~~	~~p9~~	~~0~~
~~add~~	~~p9~~	~~yes~~	~~p6~~	~~yes~~	~~p10~~	~~1~~
~~xor~~	~~p10~~	~~yes~~	~~p4~~	~~yes~~	~~p11~~	~~2~~
~~ld~~	~~p2~~	~~yes~~	~~---~~	~~yes~~	~~p12~~	~~3~~

Where are we Heading?

- T4: Advanced Processors III

Acknowledgement

Slides in this topic are inspired in part by material developed and copyright by:

- ARM Courseware
- Prof. Prof. Joe Devietti @ Upenn, CIS 571
- Prof. Ron Dreslinski @ UMich, EECS 470
- Prof. Hakim Weatherspoon @ Cornell, CS 3410
- Prof. Krste Asanovic @ UCB, CS252
- Xinfei Guo @ JI, VE370 2021 SU

Action Items

- Reading Materials
    - Ch. 3.4-3.9
    - Ch. Appendix C.7