

Parallelism I: Inside the Core

Today

- Wide issue and out-of-order
- CMPs

Remember!

- 141L projects due on Friday
- 141L CAPE tomorrow (please come)
- 141 Final on Monday (3-6pm)
- 141L Pizza party on Monday (6pm in CSE4140)

The final

- Comprehensive
- Same general format as the Midterm.
- Review the homeworks, the slides, and the quizzes.

Key Points

- What is wide issue mean?
- How does does it affect performance?
- How does it affect pipeline design?
- What is VLIW? What are its advantages?
- What is the basic idea behind out-of-order execution?
- What is the difference between a true and false dependence?
- How do OOO processors remove false dependences?
- What is Simultaneous Multithreading?

Parallelism

- $ET = IC * CPI * CT$
- IC is more or less fixed
- We have shrunk cycle time as far as we can
- We have achieved a CPI of 1.
- Can we get faster?

Parallelism

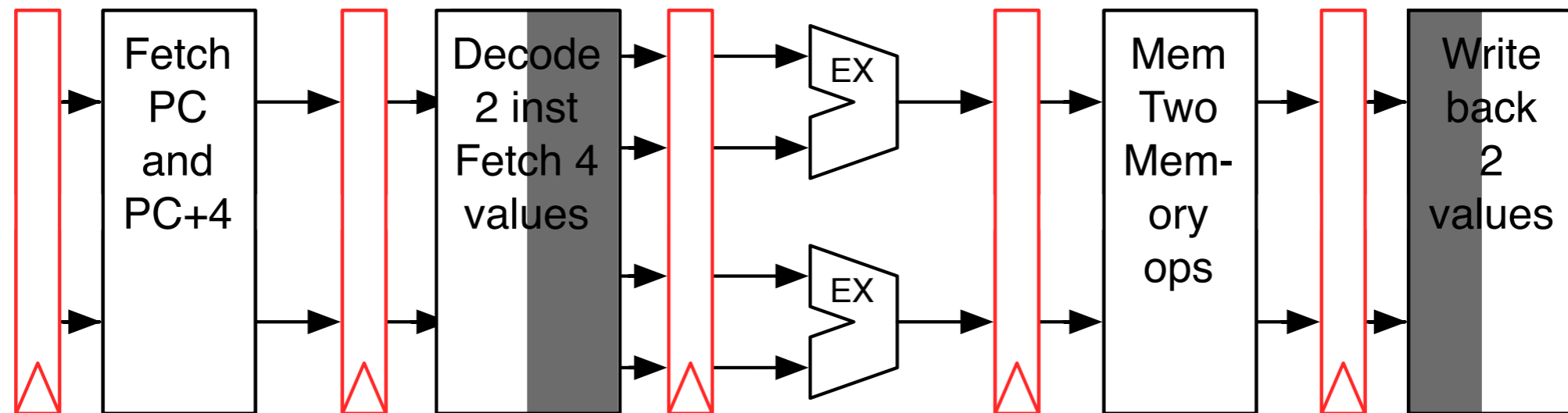
- $ET = IC * CPI * CT$
- IC is more or less fixed
- We have shrunk cycle time as far as we can
- We have achieved a CPI of 1.
- Can we get faster?

We can reduce our CPI to less than 1.

The processor must do multiple operations at once.

This is called Instruction Level Parallelism (ILP)

Approach 1: Widen the pipeline



- Process two instructions at once instead of 1
- Often 1 “odd” PC instruction and 1 “even” PC
 - This keeps the instruction fetch logic simpler.
- 2-wide, in-order, superscalar processor
- Potential problems?

Single issue refresher

	cycle 0	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6	cycle 7	cycle 8
add \$s1,\$s2,\$s3	F	D	E	M	W				
sub \$s2,\$s4,\$s5		F	D	E	M	W			
ld \$s3, 0(\$s2)			F	D	E	M	W		
add \$t1, \$s3, \$s3				F	D	D	E	M	W

Single issue refresher

	cycle 0	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6	cycle 7	cycle 8
add \$s1,\$s2,\$s3	F	D	E	M	W				
sub \$s2,\$s4,\$s5		F	D	E	M	W			
ld \$s3, 0(\$s2)			F	D	E	M	W		
add \$t1, \$s3, \$s3				F	D	D	E	M	W

Forwarding

Single issue refresher

	cycle 0	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6	cycle 7	cycle 8
add \$s1,\$s2,\$s3	F	D	E	M	W				
sub \$s2,\$s4,\$s5		F	D	E	M	W			
ld \$s3, 0(\$s2)			F	D	E	M	W		
add \$t1, \$s3, \$s3				F	D	D	E	M	W

Forwarding

Forwarding

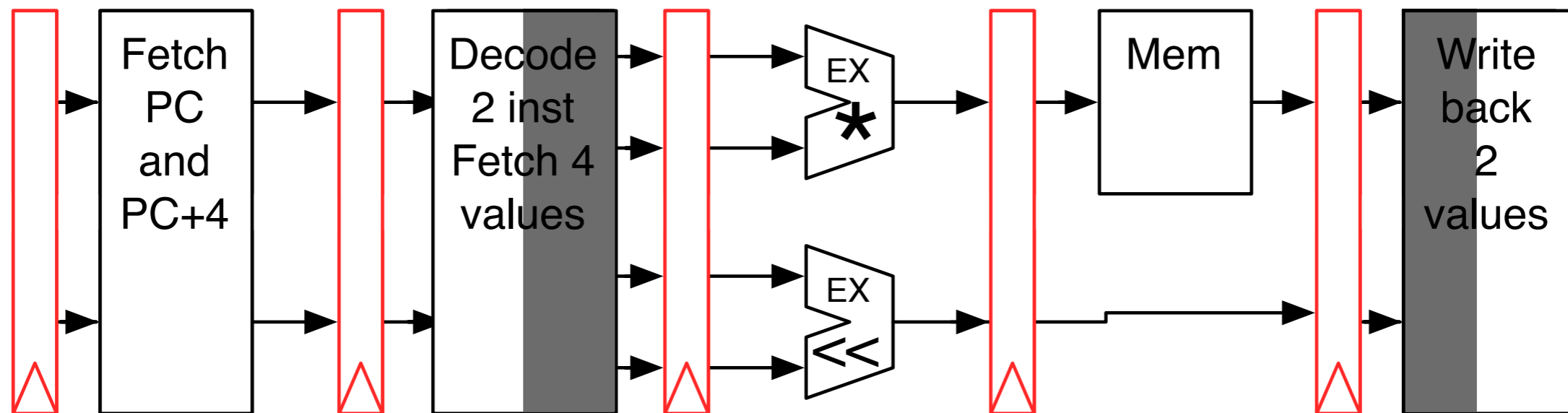
Dual issue: Ideal Case

	0	1	2	3	4	5	6	7	8
add \$s1,\$s2,\$s3	F	D	E	M	W				
sub \$s2,\$s4,\$s5	F	D	E	M	W				
ld \$s3, 0(\$s2)		F	D	E	M	W			
add \$t1, \$s3, \$s3		F	D	E	M	W			
...			F	D	E	M	W		
...			F	D	E	M	W		
...				F	D	E	M	W	
...				F	D	E	M	W	
...					F	D	E	M	W
...					F	D	E	M	W

CPI == 0.5!

Dual issue: Structural Hazards

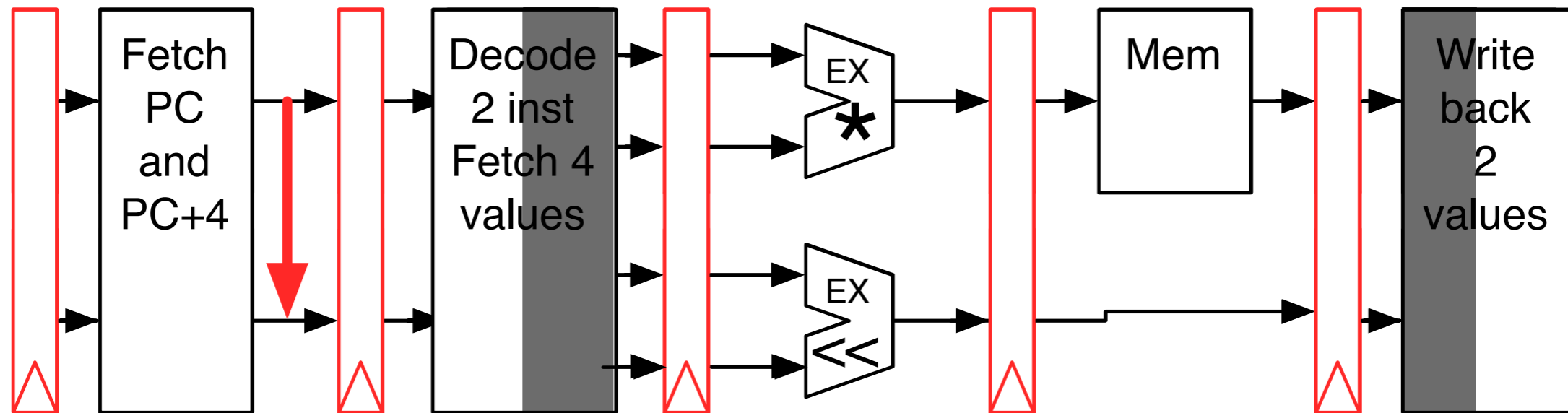
- Structural hazards
 - We might not replicate everything
 - Perhaps only one multiplier, one shifter, and one load/store unit
 - What if the instruction is in the wrong place?



If an “upper” instruction needs the “lower” pipeline,
squash the “lower” instruction

Dual issue: Structural Hazards

- Structural hazards
 - We might not replicate everything
 - Perhaps only one multiplier, one shifter, and one load/store unit
 - What if the instruction is in the wrong place?



If an “upper” instruction needs the “lower” pipeline,
squash the “lower” instruction

Dual issue: dealing with hazards

	0	1	2	3	4	5	6	7	8	9	10
add	F	D	E	M	W						
sub	F	D	E	M	W						
Mul		F	D	E	M	W					
Shift		F	D	E	M	W					
Shift			F	D	E	M	W				
Ld			F	x	x	x	x				
Shift				x	x	x	x	x			
Ld				F	D	E	M	W			

Dual issue: dealing with hazards

PC = 0

	0	1	2	3	4	5	6	7	8	9	10
add	F	D	E	M	W						
sub	F	D	E	M	W						
Mul		F	D	E	M	W					
Shift		F	D	E	M	W					
Shift			F	D	E	M	W				
Ld			F	x	x	x	x				
Shift				x	x	x	x	x			
Ld				F	D	E	M	W			

Dual issue: dealing with hazards

	0	1	2	3	4	5	6	7	8	9	10
PC = 0	add	F	D	E	M	W					
	sub	F	D	E	M	W					
PC = 8	Mul		F	D	E	M	W				
	Shift		F	D	E	M	W				
Shift			F	D	E	M	W				
Ld			F	x	x	x	x				
Shift				x	x	x	x	x			
Ld				F	D	E	M	W			

Dual issue: dealing with hazards

	0	1	2	3	4	5	6	7	8	9	10
PC = 0	add	F	D	E	M	W					
	sub	F	D	E	M	W					
PC = 8	Mul		F	D	E	M	W				
	Shift		F	D	E	M	W				
PC = 12	Shift			F	D	E	M	W			
	Ld			F	x	x	x	x			
	Shift				x	x	x	x	x		
	Ld				F	D	E	M	W		

Shift moves to lower pipe
load is squashed

Dual issue: dealing with hazards

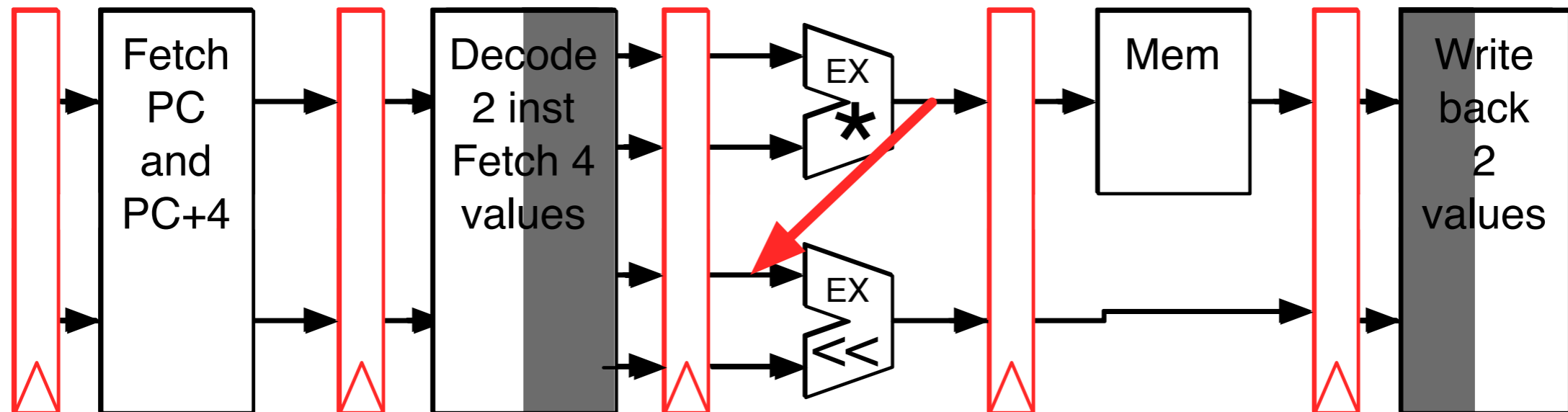
	0	1	2	3	4	5	6	7	8	9	10
PC = 0	add	F	D	E	M	W					
	sub	F	D	E	M	W					
PC = 8	Mul		F	D	E	M	W				
	Shift		F	D	E	M	W				
PC = 12	Shift			F	D	E	M	W			
	Ld			F	x	x	x	x			
PC = 12	Shift				x	x	x	x	x		
	Ld				F	D	E	M	W		

Shift moves to lower pipe
load is squashed

Load uses lower pipe
Shift becomes a noop

Dual issue: Data Hazards

- The “lower” instruction may need a value produced by the “upper” instruction
- Forwarding cannot help us -- we must stall.



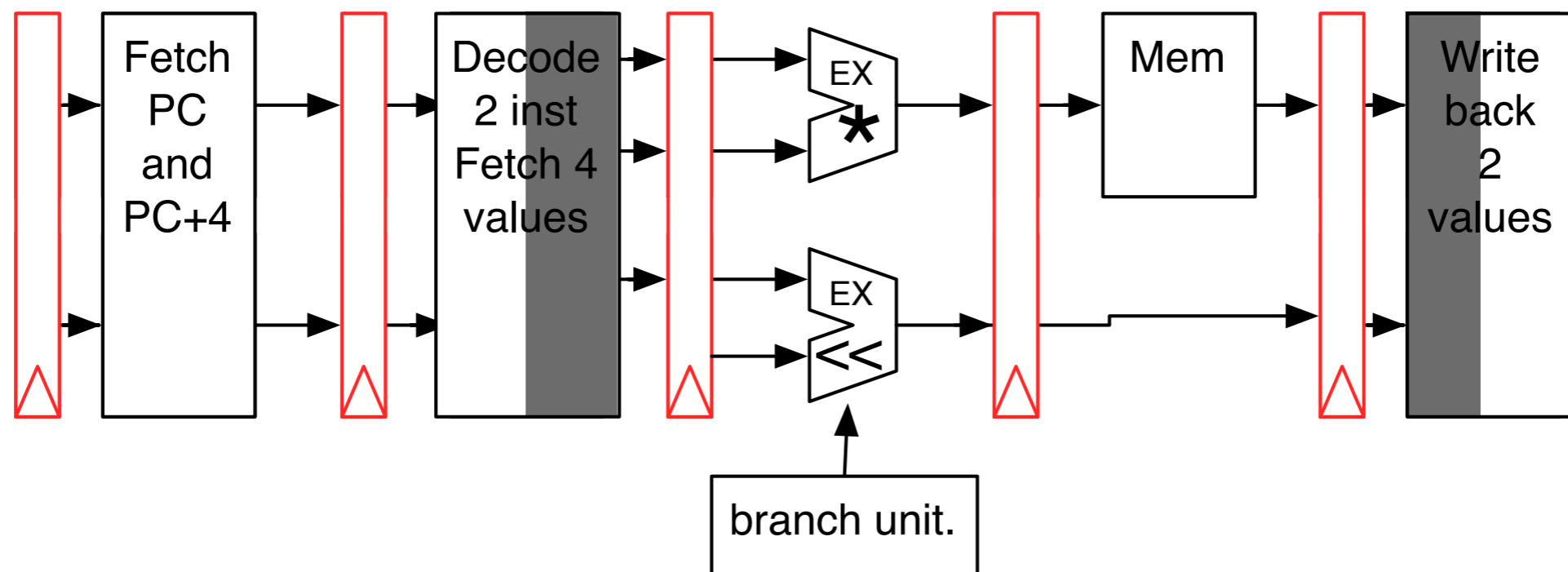
Dual issue: dealing with hazards

- Forwarding is essential!
- Both pipes stall.

	0	1	2	3	4	5	6	7	8	9	10
add \$s1, \$s3, #4	F	D	E	M	W						
sub \$s4, \$s1, #4	F	D	D	E	M	W					
add ...		F	F	D	E	M	W				
sub ...		F	F	D	E	M	W				
and ...				F	D	E	M	W			
or ...				F	D	E	M	W			

Dual issue: Control Hazards

- The “upper” instruction might be branch.
- The “lower” instruction might be on the wrong path
- Solution 1: Require branches to execute in the lower pipeline -- See “structural hazards”.
- What about consecutive branches? -- Exercise for the reader
- What about branches to odd addresses? -- Squash the upper pipe



Compiling for Dual Issue

- The compiler should
 - Pair up non-conflicting instructions
 - Align branch targets (by potentially inserting noops above them)

Beyond Dual Issue

- Wider pipelines are possible.
 - There is often a separate floating point pipeline.
- Wide issue leads to hardware complexity
- Compiling gets harder, too.
- In practice, processors use of two options if they want more ILP
 - If we can change the ISA: VLIW
 - If we can't: Out-of-order

Very Long Instruction Words

- VLIW
 - In dual-issue the compiler paired up instructions, but it was just an optimization.
 - VLIW makes this a requirement
- Each instruction word contains multiple operations
 - The semantics of the ISA say that they happen in parallel
 - The compiler can (and must) respect this constraint

VLIW Example

- $\$s1 = 1$; $\$s2 = 1$, $\$s3 = 4$
- VLIW instruction word :
 - `<add $s2, $s1, $s3; sub $s5, $s2, $s3>`
 - What value does the 'sub' see for $\$s2$? -- 1 or 5?

VLIW Example

- $\$s1 = 1$; $\$s2 = 1$, $\$s3 = 4$
- VLIW instruction word :
 - `<add $s2, $s1, $s3; sub $s5, $s2, $s3>`
 - What value does the 'sub' see for \$s2? -- 1 or 5?

VLIW's History

- VLIW has been around for a long time
 - It's the simplest way to get ILP, because the burden of avoiding hazards lies completely with the compiler.
 - When hardware was expensive, this seemed like a good idea.
- However, the compiler problem is extremely hard.
 - There end up being lots of noops in the long instruction words.
- As a result, they have either
 - 1. met with limited commercial success as general purpose machines (many companies) or,
 - 2. Become very complicated in new and interesting ways (for instance, by providing special registers and instructions to eliminate branches), or
 - 3. Both 1 and 2 -- See the Itanium from intel.

VLIW Today

- VLIW is alive and well in Digital signal processors
- They are simple and low power, which is important in embedded systems.
- DSPs run the same, very regular loops all the time, and VLIW machines are very good at that.
- It is worthwhile to hand-code these loops in assembly.

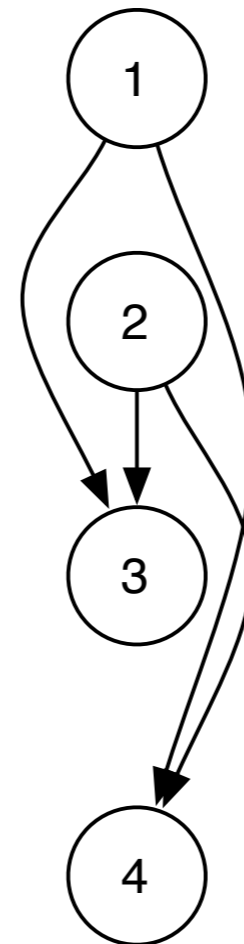
Going Out of Order: Data dependence refresher.

1: add \$t1, \$s2, \$s3

2: sub \$t2, \$s3, \$s4

3: or \$t5, \$t1, \$t2

4: add \$t3, \$t1, \$t2



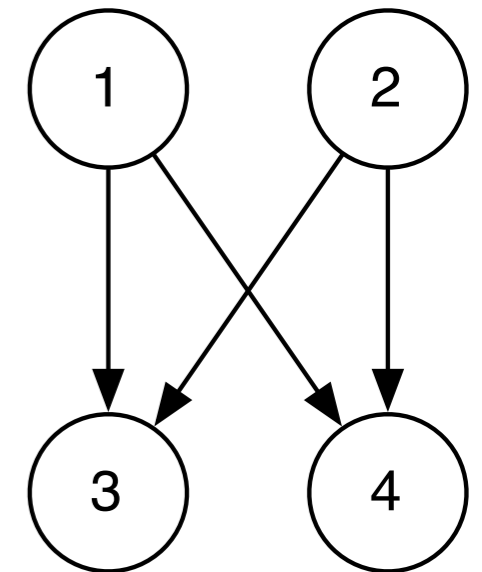
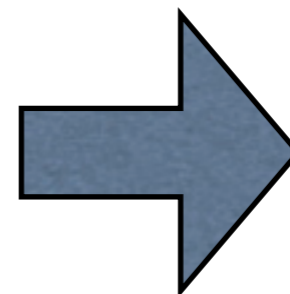
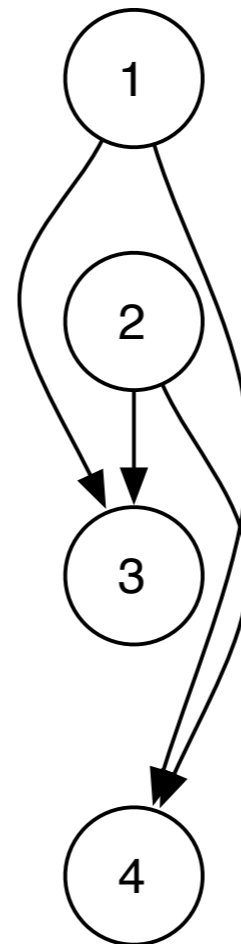
Going Out of Order: Data dependence refresher.

1: add \$t1, \$s2, \$s3

2: sub \$t2, \$s3, \$s4

3: or \$t5, \$t1, \$t2

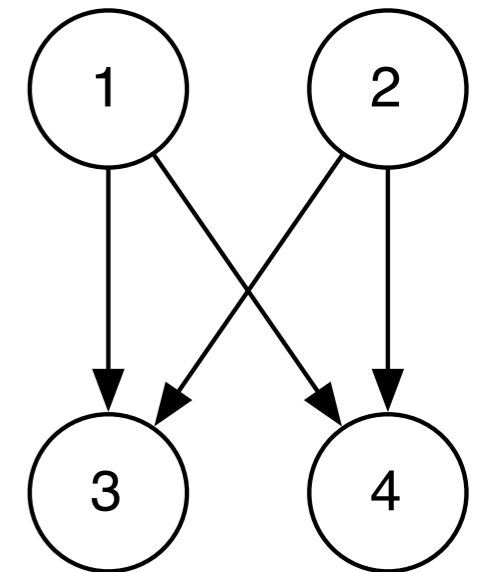
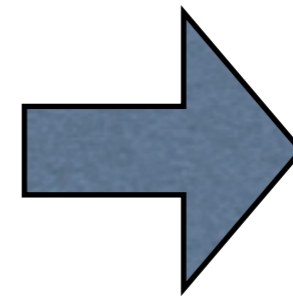
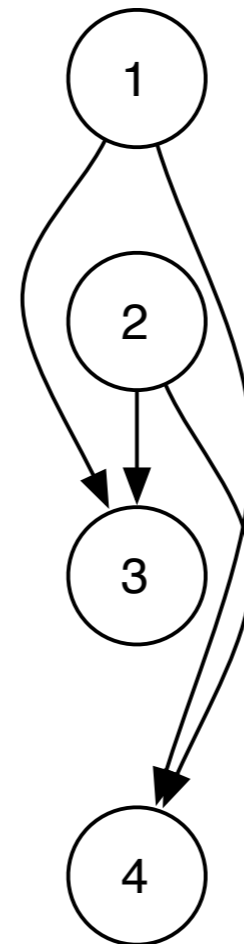
4: add \$t3, \$t1, \$t2



There is parallelism!!
We can execute
1 & 2 at once
and
3 & 4 at once

Going Out of Order: Data dependence refresher.

1: add \$t1, \$s2, \$s3
2: sub \$t2, \$s3, \$s4
3: or \$t5, \$t1, \$t2
4: add \$t3, \$t1, \$t2



There is parallelism!!
We can execute
1 & 2 at once
and
3 & 4 at once

We can parallelize instructions that do not have a “read-after-write” dependence (RAW)

Data dependences

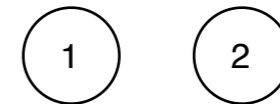
- In general, if there is no dependence between two instructions, we can execute them in either order or simultaneously.

- But beware:

- Is there a dependence here?

```
1: add $t1, $s2, $s3
```

```
2: sub $t1, $s3, $s4
```



- Can we reorder the instructions?

```
2: sub $t1, $s3, $s4
```

```
1: add $t1, $s2, $s3
```

- Is the result the same?

Data dependences

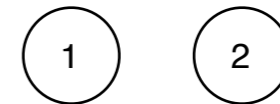
- In general, if there is no dependence between two instructions, we can execute them in either order or simultaneously.

- But beware:

- Is there a dependence here?

1: add \$t1, \$s2, \$s3

2: sub \$t1, \$s3, \$s4



- Can we reorder the instructions?

2: sub \$t1, \$s3, \$s4

1: add \$t1, \$s2, \$s3

No! The final value of \$t1 is different

- Is the result the same?

False Dependence #1

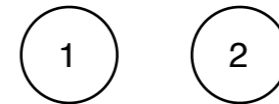
- Also called “Write-after-Write” dependences (WAW) occur when two instructions write to the same value
- The dependence is “false” because no data flows between the instructions -- They just produce an output with the same name.

Beware again!

- Is there a dependence here?

1: add \$t1, \$s2, \$s3

2: sub \$s2, \$s3, \$s4



- Can we reorder the instructions?

2: sub \$s2, \$s3, \$s4

1: add \$t1, \$s2, \$s3

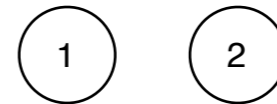
- Is the result the same?

Beware again!

- Is there a dependence here?

1: add \$t1, \$s2, \$s3

2: sub \$s2, \$s3, \$s4



- Can we reorder the instructions?

2: sub **\$s2**, \$s3, \$s4

1: add \$t1, **\$s2**, \$s3

No! The value in \$s2 that I needs will be destroyed

- Is the result the same?

False Dependence #2

- This is a Write-after-Read (WAR) dependence
- Again, it is “false” because no data flows between the instructions

Out-of-Order Execution

- Any sequence of instructions has set of RAW, WAW, and WAR hazards that constrain its execution.
- Can we design a processor that extracts as much parallelism as possible, while still respecting these dependences?

The Central OOO Idea

1. Fetch a bunch of instructions
2. Build the dependence graph
3. Find all instructions with no unmet dependences
4. Execute them.
5. Repeat

Example

1: add \$t1, \$s2, \$s3

2: sub \$t2, \$s3, \$s4

3: or \$t3, \$t1, \$t2

4: add \$t5, \$t1, \$t2

— WAR —→

— WAW —→

— RAW —→

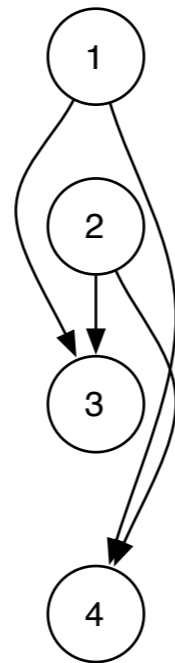
Example

1: add \$t1, \$s2, \$s3

2: sub \$t2, \$s3, \$s4

3: or \$t3, \$t1, \$t2

4: add \$t5, \$t1, \$t2



— WAR —→

— WAW —→

— RAW —→

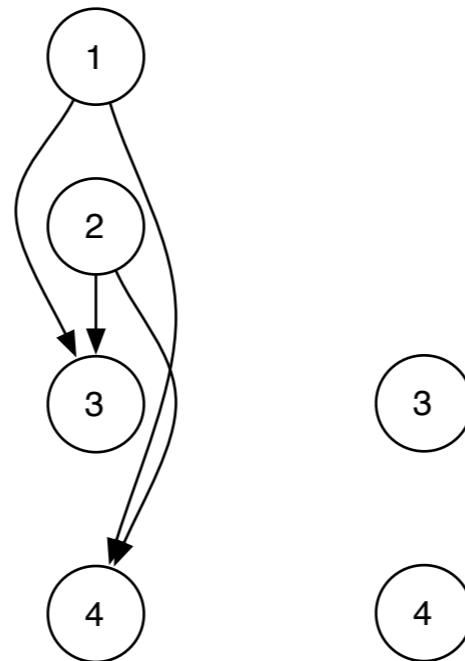
Example

1: add \$t1, \$s2, \$s3

2: sub \$t2, \$s3, \$s4

3: or \$t3, \$t1, \$t2

4: add \$t5, \$t1, \$t2



— WAR —→

— WAW —→

— RAW —→

Example

1: add \$t1, \$s2, \$s3

2: sub \$t2, \$s3, \$s4

3: or \$t3, \$t1, \$t2

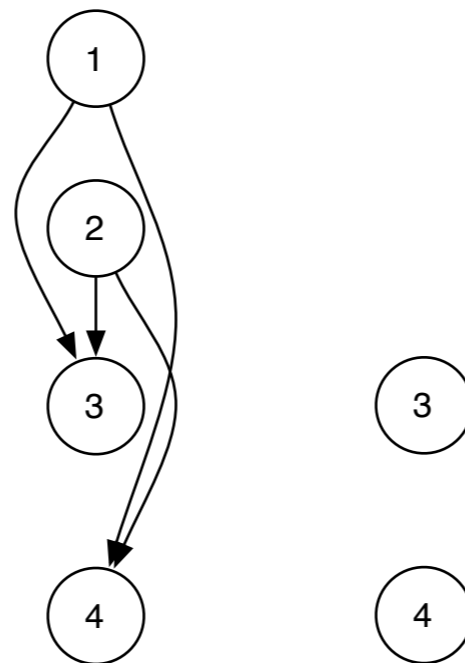
4: add \$t5, \$t1, \$t2

5: or \$t4, \$s1, \$s3

6: mul \$t2, \$t3, \$s5

7: sl \$t3, \$t4, \$t2

8: add \$t3, \$t5, \$t1



— WAR —→

— WAW —→

— RAW —→

Example

1: add \$t1, \$s2, \$s3

2: sub \$t2, \$s3, \$s4

3: or \$t3, \$t1, \$t2

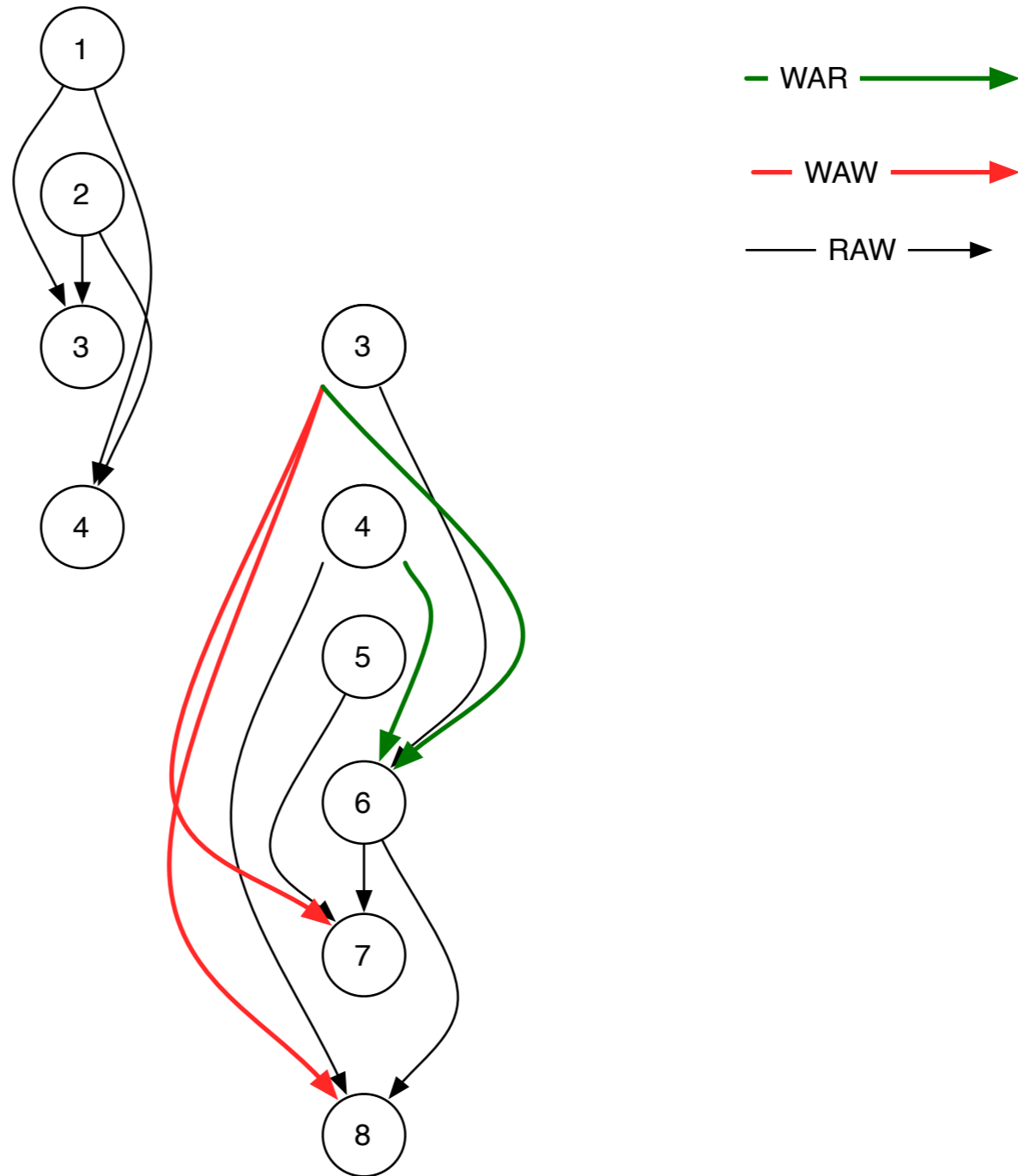
4: add \$t5, \$t1, \$t2

5: or \$t4, \$s1, \$s3

6: mul \$t2, \$t3, \$s5

7: sl \$t3, \$t4, \$t2

8: add \$t3, \$t5, \$t1



Example

1: add \$t1, \$s2, \$s3

2: sub \$t2, \$s3, \$s4

3: or \$t3, \$t1, \$t2

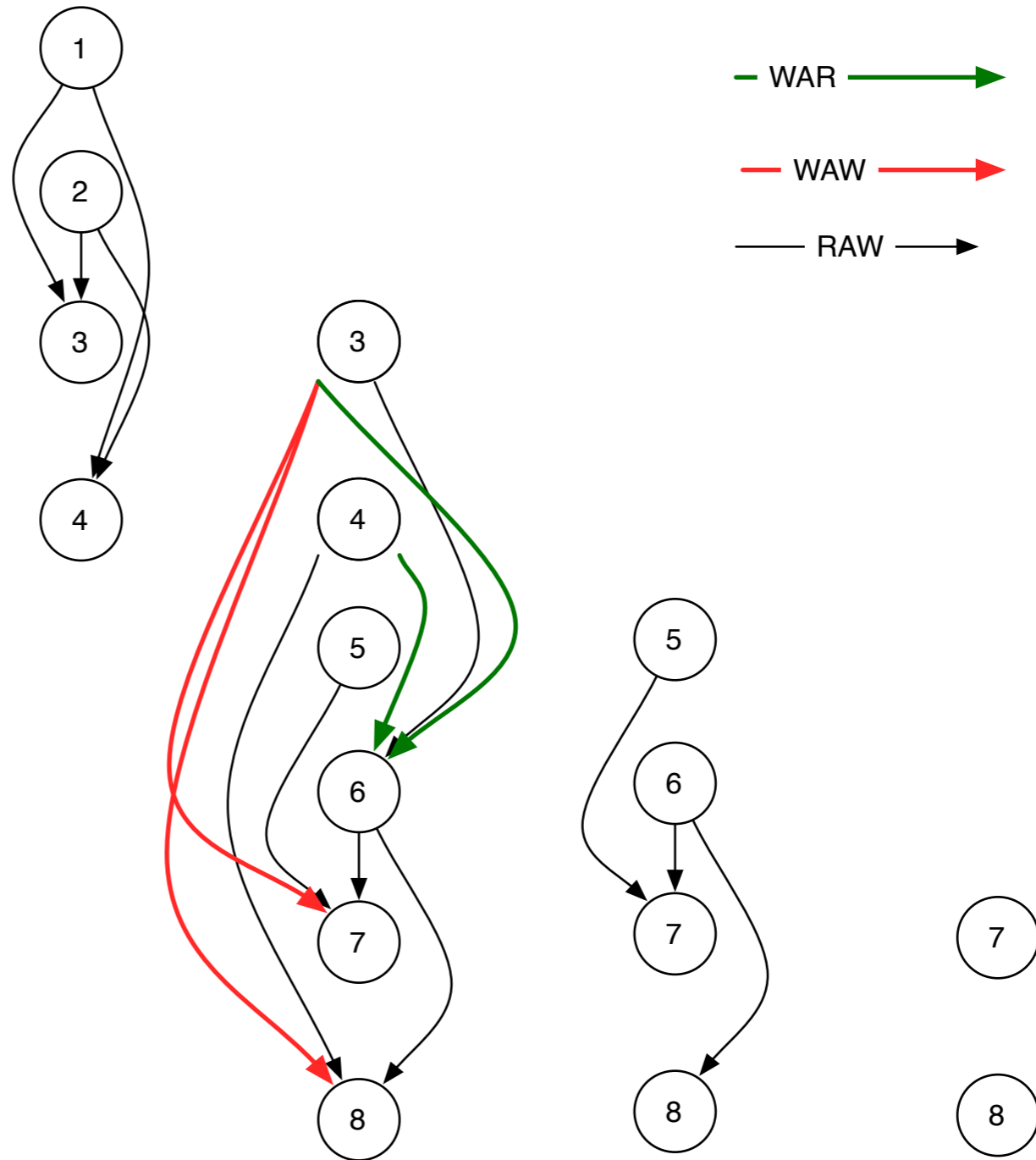
4: add \$t5, \$t1, \$t2

5: or \$t4, \$s1, \$s3

6: mul \$t2, \$t3, \$s5

7: sl \$t3, \$t4, \$t2

8: add \$t3, \$t5, \$t1



Example

1: add \$t1, \$s2, \$s3

2: sub \$t2, \$s3, \$s4

3: or \$t3, \$t1, \$t2

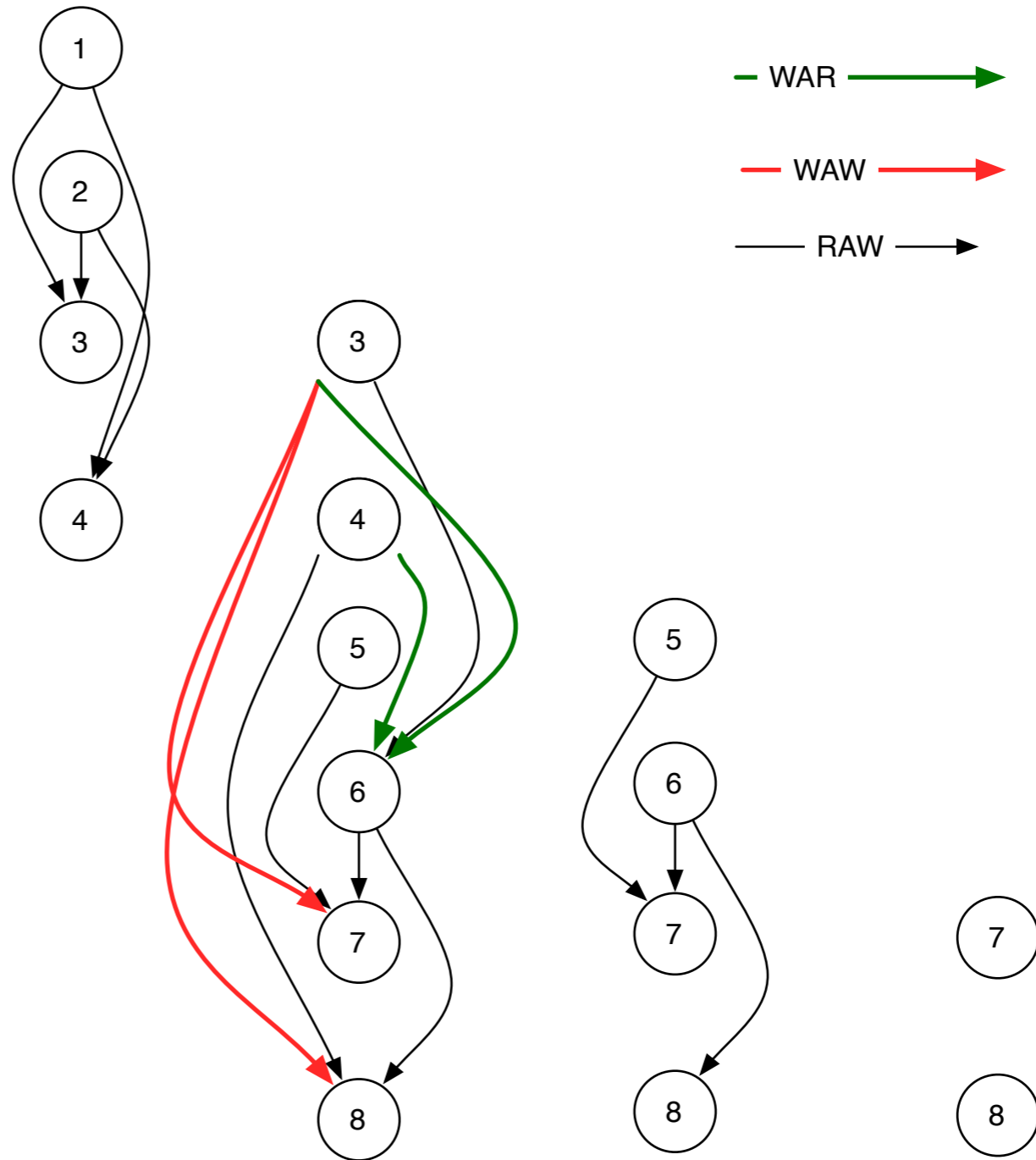
4: add \$t5, \$t1, \$t2

5: or \$t4, \$s1, \$s3

6: mul \$t2, \$t3, \$s5

7: sl \$t3, \$t4, \$t2

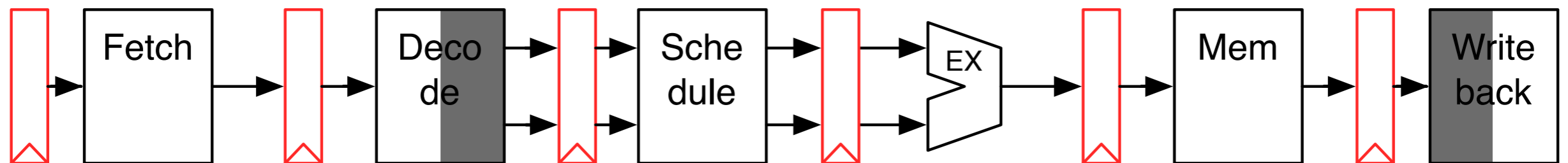
8: add \$t3, \$t5, \$t1



8 Instructions in 5 cycles

Simplified OOO Pipeline

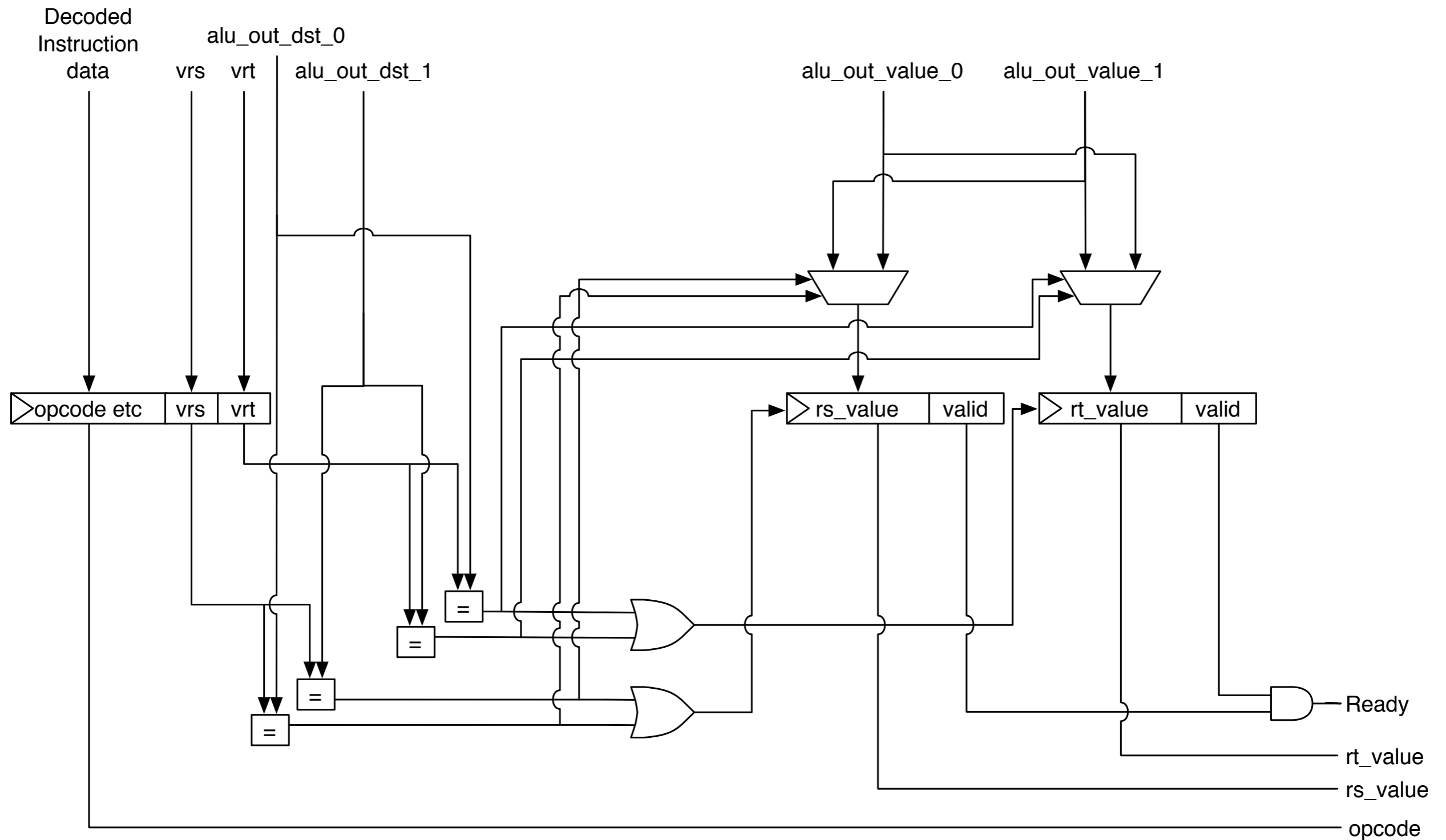
- A new “schedule” stage manages the “Instruction Window”
- The window holds the set of instruction the processor examines
 - The fetch and decode fill the window
 - Execute stage drains it
- Typically, OOO pipelines are also “wide” but it is not necessary.
- Impacts
 - More forwarding, More stalls, longer branch resolution
 - Fundamentally more work per instruction.



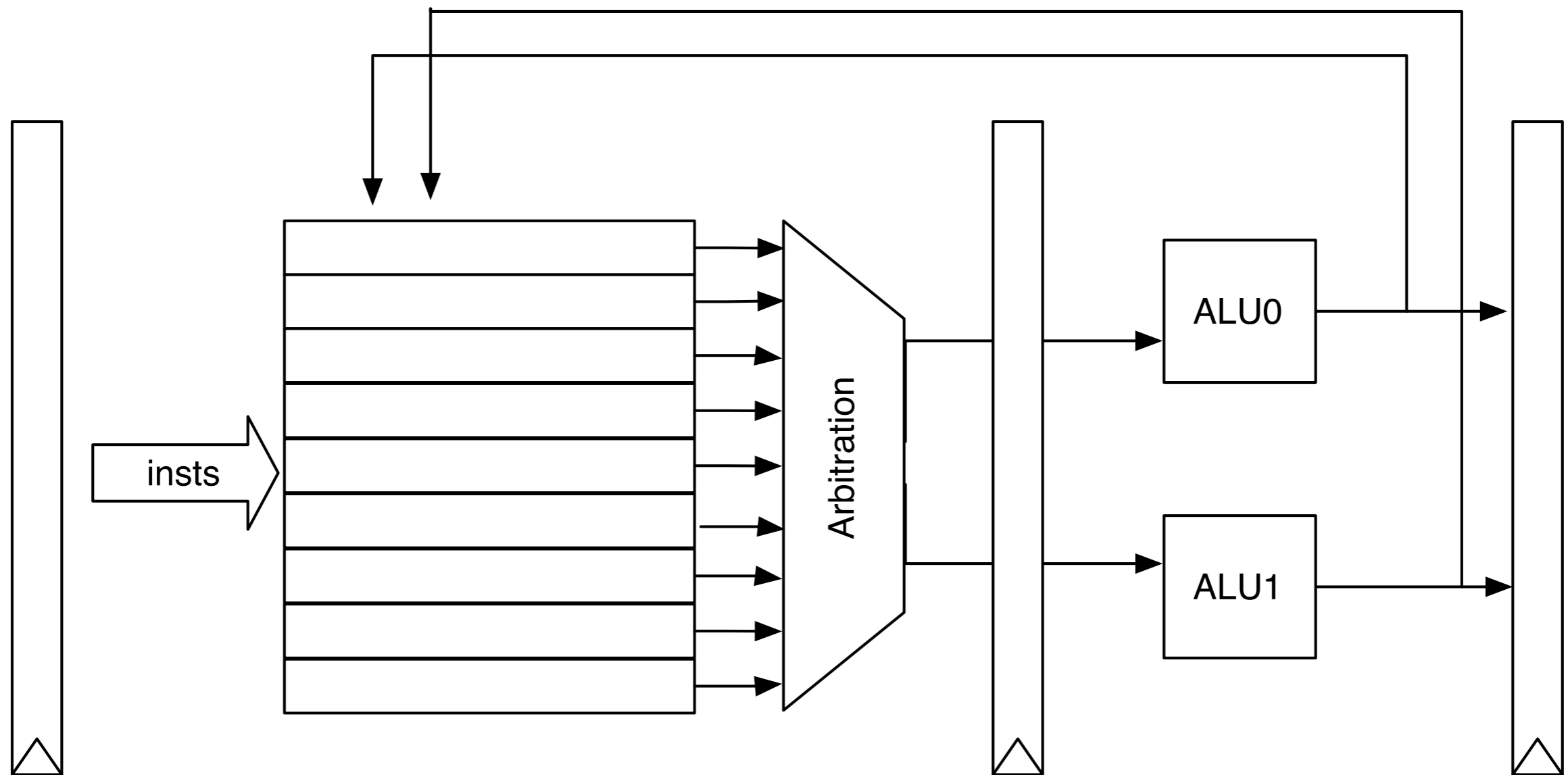
The Instruction Window

- The “Instruction Window” is the set of instructions the processor examines
 - The fetch and decode fill the window
 - Execute stage drains it
- The larger the window, the more parallelism the processor can find, but...
- Keeping the window filled is a challenge

The Issue Window



The Issue Window



Keeping the Window Filled

- Keeping the instruction window filled is key!
- Instruction windows are about 32 instructions
 - (size is limited by their complexity, which is considerable)
- Branches are every 4-5 instructions.
- This means that the processor predict 6-8 consecutive branches correctly to keep the window full.
- On a mispredict, you flush the pipeline, which includes the emptying the window.

How Much Parallelism is There?

- Not much, in the presence of WAW and WAR dependences.
- These arise because we must reuse registers, and there are a limited number we can freely reuse.
- How can we get rid of them?

Removing False Dependences

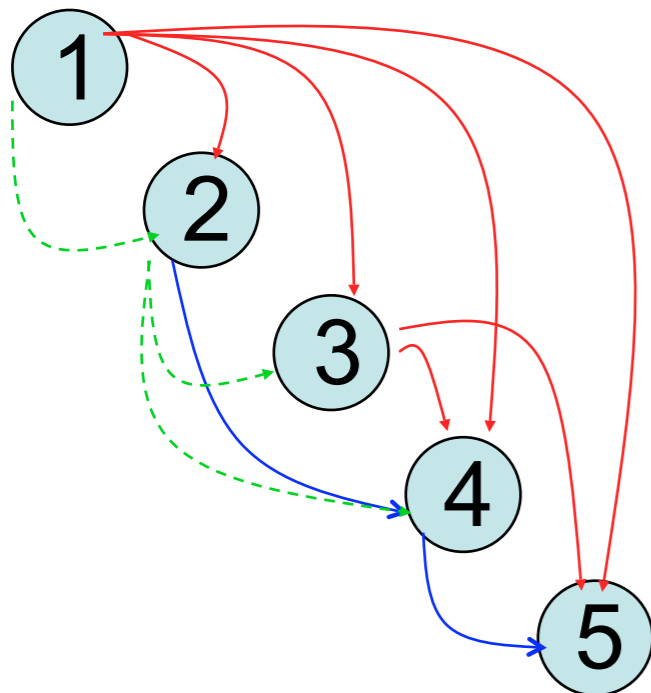
- If WAW and WAR dependences arise because we have too few registers
 - Let's add more!
- But! We can't! The Architecture only gives us 32 (why or why did we only use 5 bits?)
- Solution:
 - Define a set of internal "physical" register that is as large as the number of instructions that can be "in flight" -- 128 in the latest intel chip.
 - Every instruction in the pipeline gets a registers
 - Maintaining a register mapping table that determines which physical register currently holds the value for the required "architectural" registers.
- This is called "Register Renaming"

Alpha 21264: Renaming

Register map table

```
1: Add  r3, r2, r3
2: Sub  r2, r1, r3
3: Mult r1, r3, r1
4: Add  r2, r3, r1
5: Add  r2, r1, r3
```

	r1	r2	r3
0:	p1	p2	p3
1:			
2:			
3:			
4:			
5:			



→ RAW → WAW - - - - - WAR

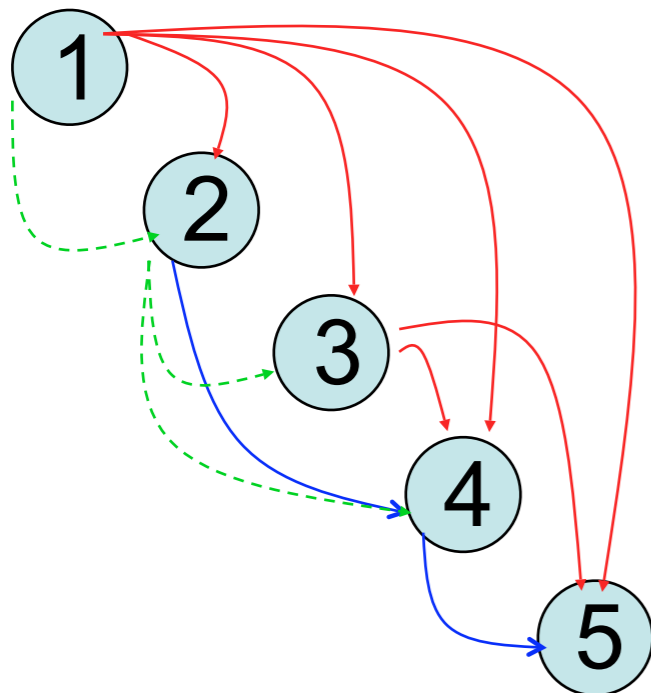
Alpha 21264: Renaming

Register map table

```
1: Add  r3, r2, r3
2: Sub  r2, r1, r3
3: Mult r1, r3, r1
4: Add  r2, r3, r1
5: Add  r2, r1, r3
```

	r1	r2	r3
0:	p1	p2	p3
1:			
2:			
3:			
4:			
5:			

p1 currently holds the value of architectural registers r1



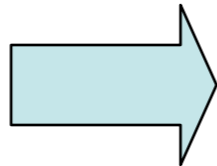
→ RAW → WAW - - - - - WAR

Alpha 21264: Renaming

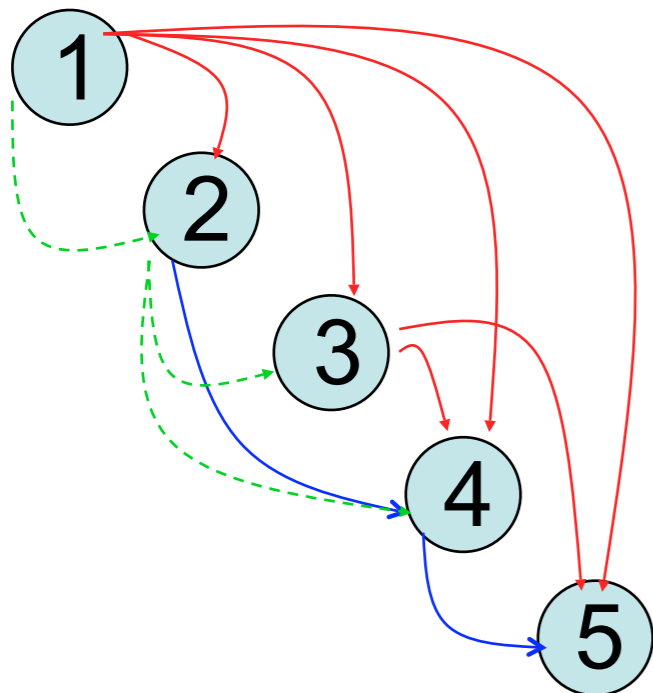
```




1: Add  r3, r2, r3
2: Sub  r2, r1, r3
3: Mult r1, r3, r1
4: Add  r2, r3, r1
5: Add  r2, r1, r3
    
```

p4, p2, p3



	r1	r2	r3
	p1	p2	p3
1:	p1	p2	p4
2:			
3:			
4:			
5:			



 RAW
  WAW
  WAR

Alpha 21264: Renaming

```

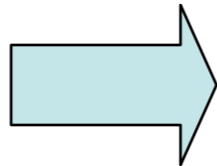
1: Add  r3, r2, r3
2: Sub  r2, r1, r3
3: Mult r1, r3, r1
4: Add  r2, r3, r1
5: Add  r2, r1, r3

```

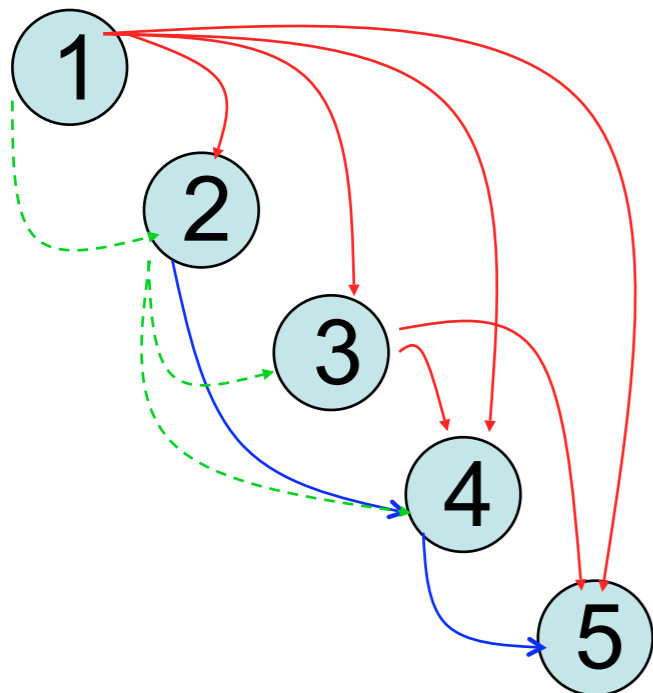
```

p4, p2, p3
p5, p1, p4

```



	r1	r2	r3
0:	p1	p2	p3
1:	p1	p2	p4
2:	p1	p5	p4
3:			
4:			
5:			



→ RAW
 → WAW
 - - - - - WAR

Alpha 21264: Renaming

```

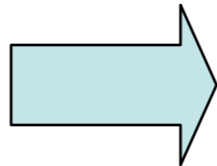
1: Add  r3, r2, r3
2: Sub  r2, r1, r3
3: Mult r1, r3, r1
4: Add  r2, r3, r1
5: Add  r2, r1, r3

```

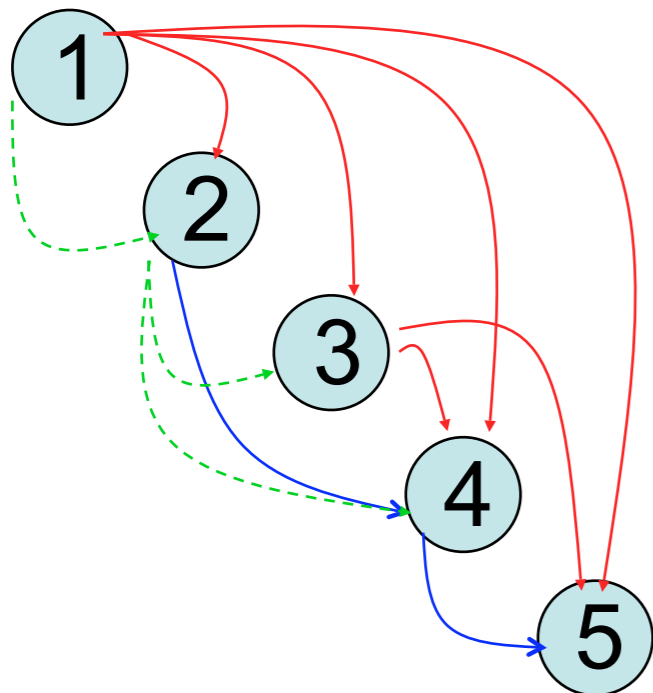
```

p4, p2, p3
p5, p1, p4
p6, p4, p1

```



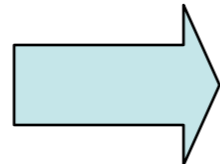
	r1	r2	r3
0:	p1	p2	p3
1:	p1	p2	p4
2:	p1	p5	p4
3:	p6	p5	p4
4:			
5:			



→ RAW
 → WAW
 - - - - -> WAR

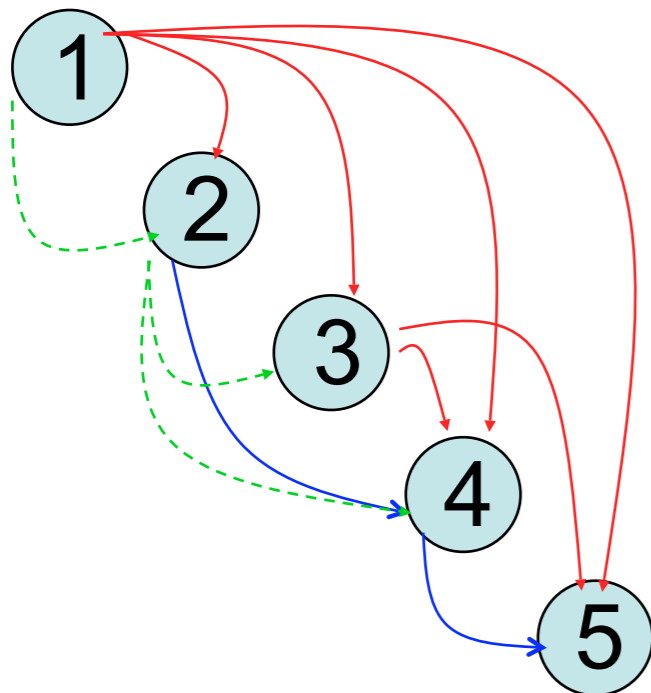
Alpha 21264: Renaming




1: Add r3, r2, r3
 2: Sub r2, r1, r3
 3: Mult r1, r3, r1
 4: Add r2, r3, r1
 5: Add r2, r1, r3



p4, p2, p3
 p5, p1, p4
 p6, p4, p1
 p7, p4, p6

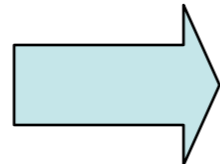
	r1	r2	r3
0:	p1	p2	p3
1:	p1	p2	p4
2:	p1	p5	p4
3:	p6	p5	p4
4:	p6	p7	p4
5:			



 RAW
  WAW
  WAR

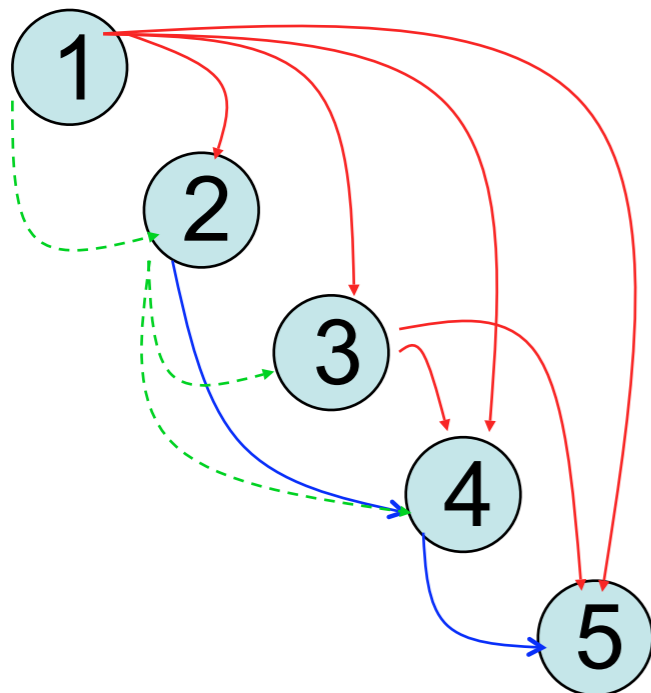
Alpha 21264: Renaming

1: Add r3, r2, r3
 2: Sub r2, r1, r3
 3: Mult r1, r3, r1
 4: Add r2, r3, r1
 5: Add r2, r1, r3



p4, p2, p3
 p5, p1, p4
 p6, p4, p1
 p7, p4, p6
 p8, p6, p4

	r1	r2	r3
0:	p1	p2	p3
1:	p1	p2	p4
2:	p1	p5	p4
3:	p6	p5	p4
4:	p6	p7	p4
5:	p6	p8	p4

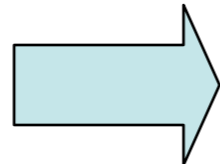


RAW
 WAW
 WAR

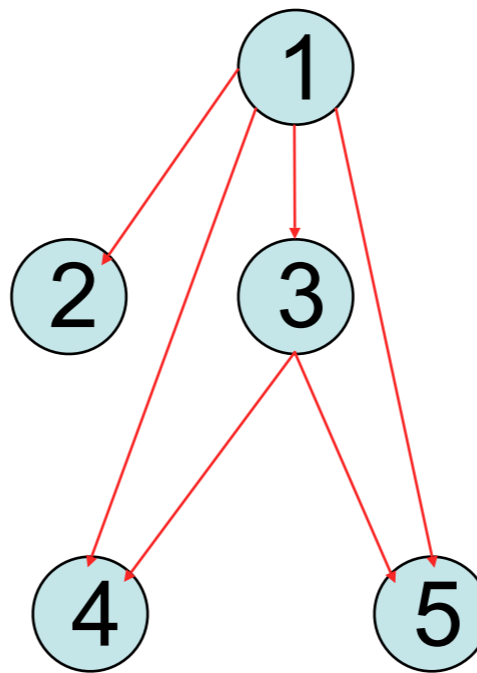
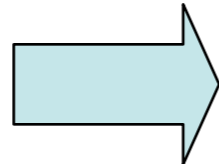
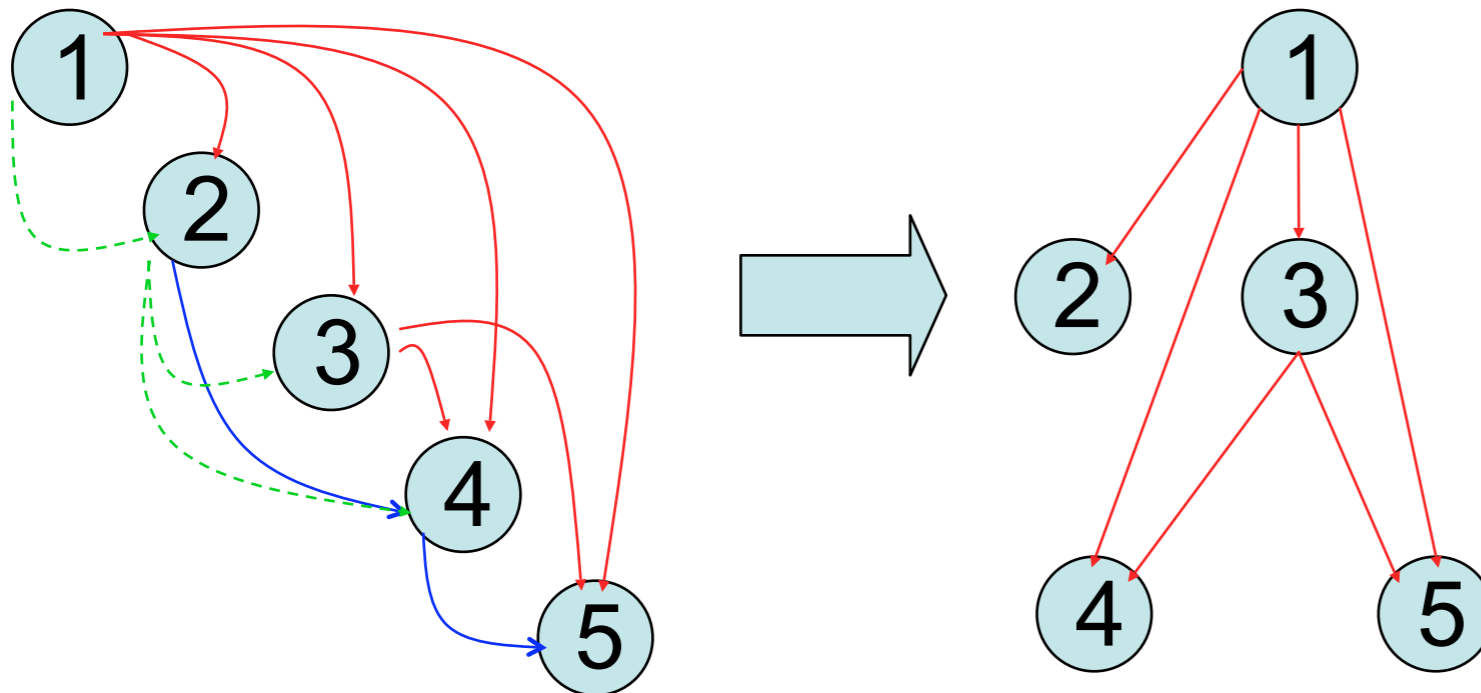
Alpha 21264: Renaming


1: Add r3, r2, r3
 2: Sub r2, r1, r3
 3: Mult r1, r3, r1
 4: Add r2, r3, r1
 5: Add r2, r1, r3

p4, p2, p3
 p5, p1, p4
 p6, p4, p1
 p7, p4, p6
 p8, p6, p4



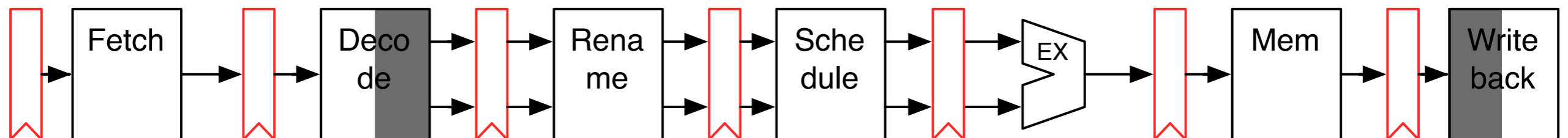
	r1	r2	r3
0:	p1	p2	p3
1:	p1	p2	p4
2:	p1	p5	p4
3:	p6	p5	p4
4:	p6	p7	p4
5:	p6	p8	p4



 RAW
  WAW
  WAR

New OOO Pipeline

- The register file is larger (to hold the physical registers)
- The pipeline is longre
 - more forwarding
 - Long branch delay
- The payoff had better be significant (and it is)



Modern OOO Processors

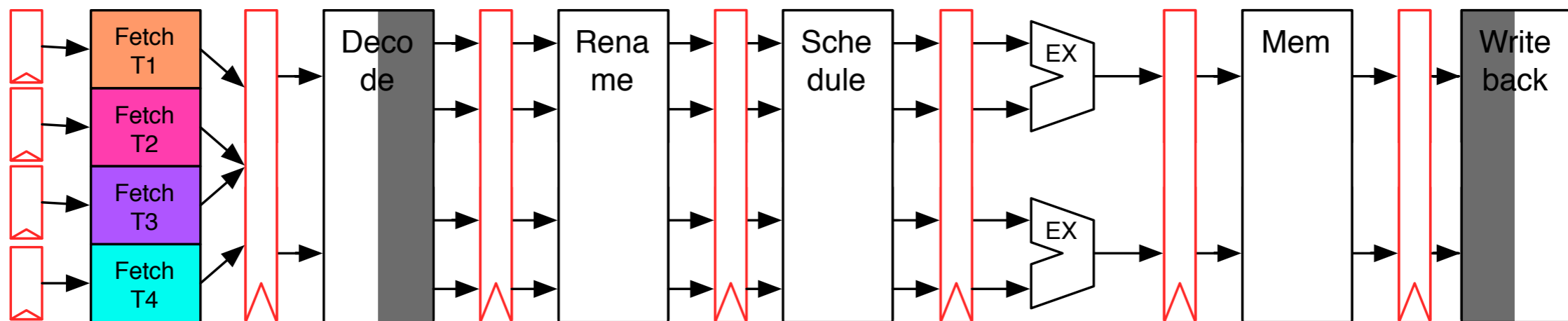
- The fastest machines in the world are OOO superscalars
- AMD Barcelona
 - 6-wide issue
 - 106 instructions in flight at once.
- Intel Nehalem
 - 5-way issue to 12 ALUs
 - > 128 instructions in flight
- OOO provides the most benefit for memory operations.
 - Non-dependent instructions can keep executing during cache misses.
 - This is so-called “memory-level parallelism.”
 - It is enormously important. CPU performance is (almost) all about memory performance nowadays (remember the memory wall graphs!)

The Problem with OOO

- Even the fastest OOO machines only get about 1-2 IPC, even though they are 4-5 wide.
- Problems
 - Insufficient ILP within applications. -- 1-2 per thread, usually
 - Poor branch prediction performance
 - Single threads also have little memory parallelism.
- Observation
 - On many cycles, many ALUs and instruction queue slots sit empty

Simultaneous Multithreading

- AKA HyperThreading in Intel machines
- Run multiple threads at the same time
- Just throw all the instructions into the pipeline
- Keep some separate data for each
 - Renaming table
 - TLB entries
 - PCs
- But the rest of the hardware is shared.
- It is surprisingly simple (but still quite complicated)



SMT Advantages

- Exploit the ILP of multiple threads at once
- Less dependence on branch prediction (fewer correct predictions required per thread)
- Less idle hardware (increased power efficiency)
- Much higher IPC -- up to 4 (in simulation)
- Disadvantages: threads can fight over resources and slow each other down.

- Historical footnote: Invented, in part, by our own Dean Tullsen when he was at UW