CS3350B Computer Organization
Chapter 4: Instruction-Level Parallelism
Part 1: Pipelining

Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Thursday March 7, 2019

Outline

Instruction-Level Parallelism

For a computer architecture, its **instruction-level parallelism** (ILP) is a measure of the number of instructions it can perform simultaneously.

ILP is usually achieved *dynamically*—after compile time—by the processor itself manipulating program execution.

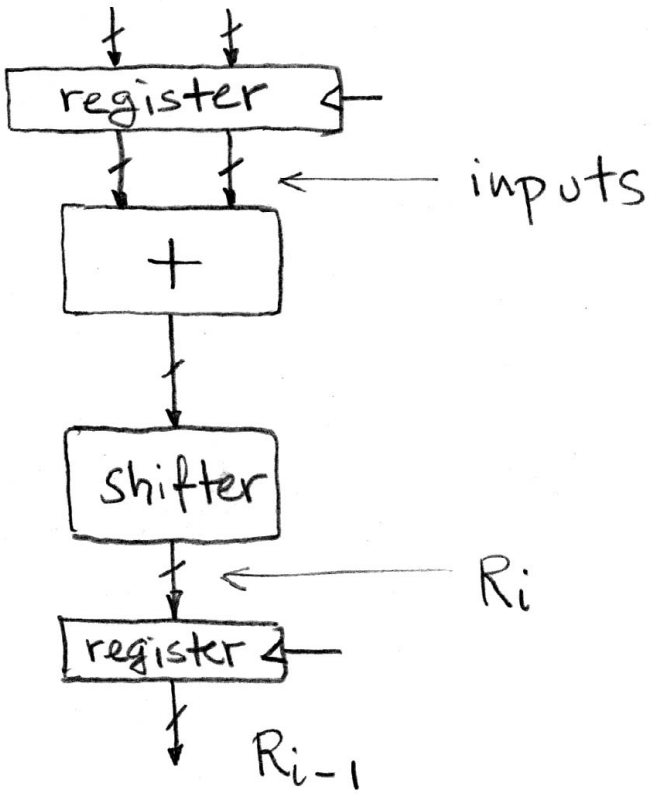
Circuitry (and appropriate control signals) needs to be added to the processor to handle the execution of many instructions simultaneously and to handle the dynamic nature of ILP.

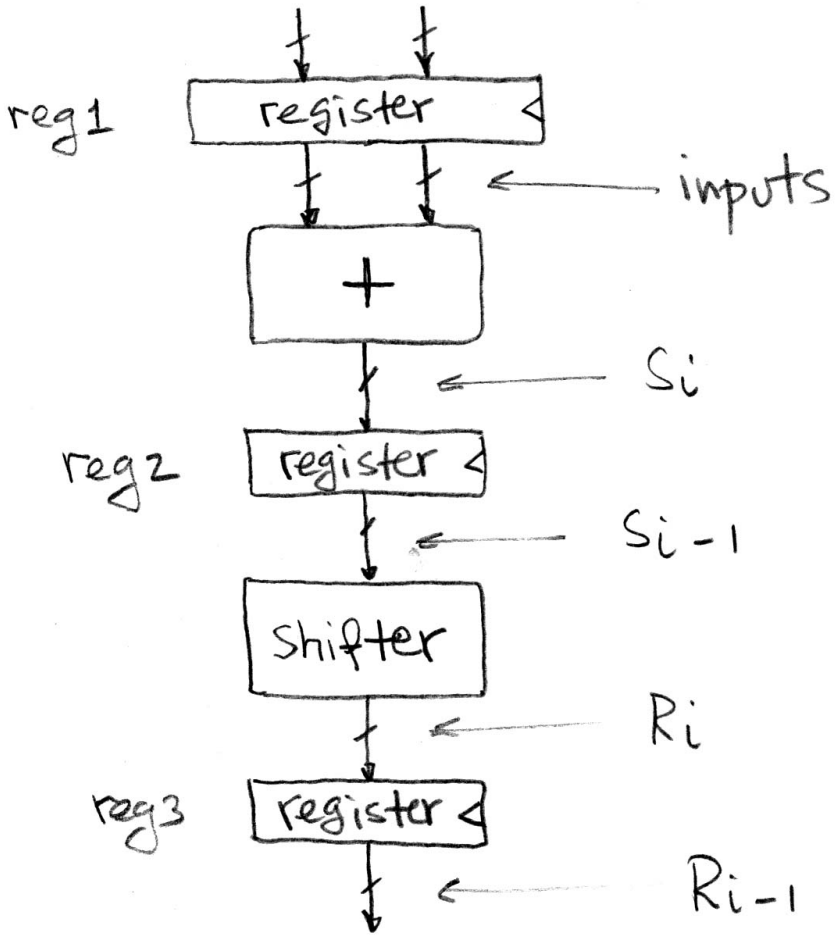
Achieving ILP

ILP can be achieved in many ways. Some topics we will look at:

- **Pipelining**
- **Superscalar** execution
- **VLIW** – very long instruction word
- **Register renaming**
- **Branch prediction**

"Pipelining" in Combinational Circuits





Break up a combinational circuit, reduce propagation delay, insert a register to store intermediate results, increase clock frequency.

Pipe, Pipeline, Pipelining

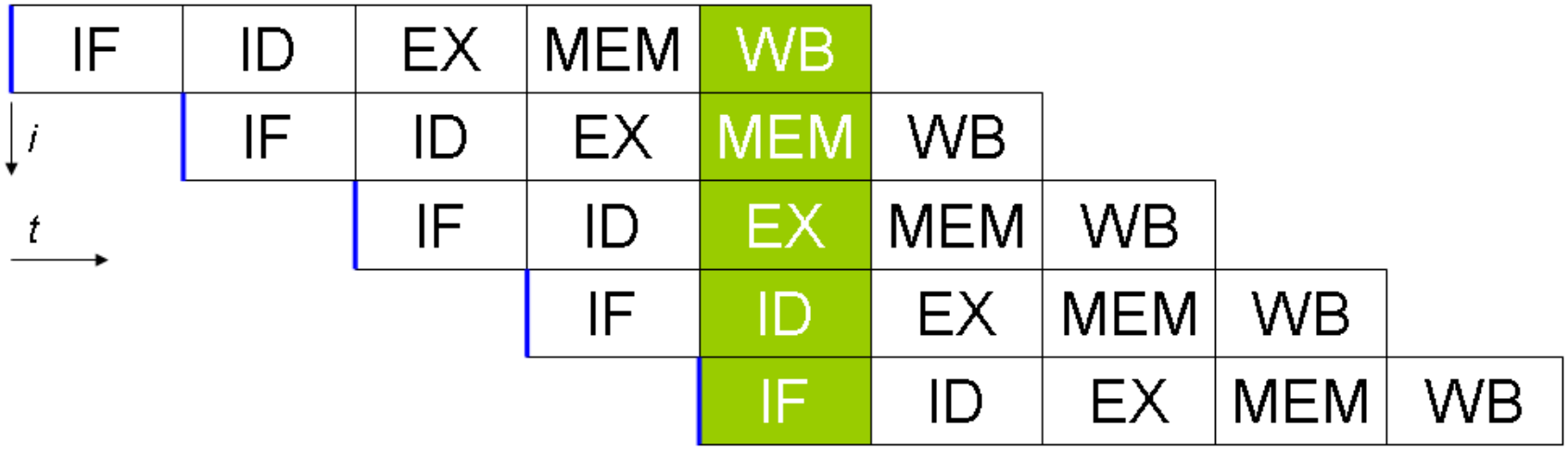
Unix pipe: pass data from one program to another.

```
ls -la | grep "foo.txt"
```

Data pipeline: a sequential series of processing elements (CPUs, circuits, programs, etc.) where the output of one is passed as the input to another. Buffer storage is needed between elements to store temporary data.

Pipelining: a technique for instruction-level parallelism where each stage of the datapath is always kept busy. Instructions are **overlapped**.

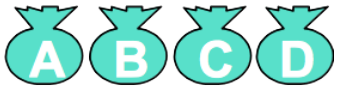
Pipelining the RISC Datapath

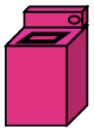


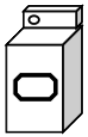
- Each stage is executing a *different* instruction.
- 5 stages $\Longrightarrow$ 5 instructions executed at once.

Outline

Doing Laundry

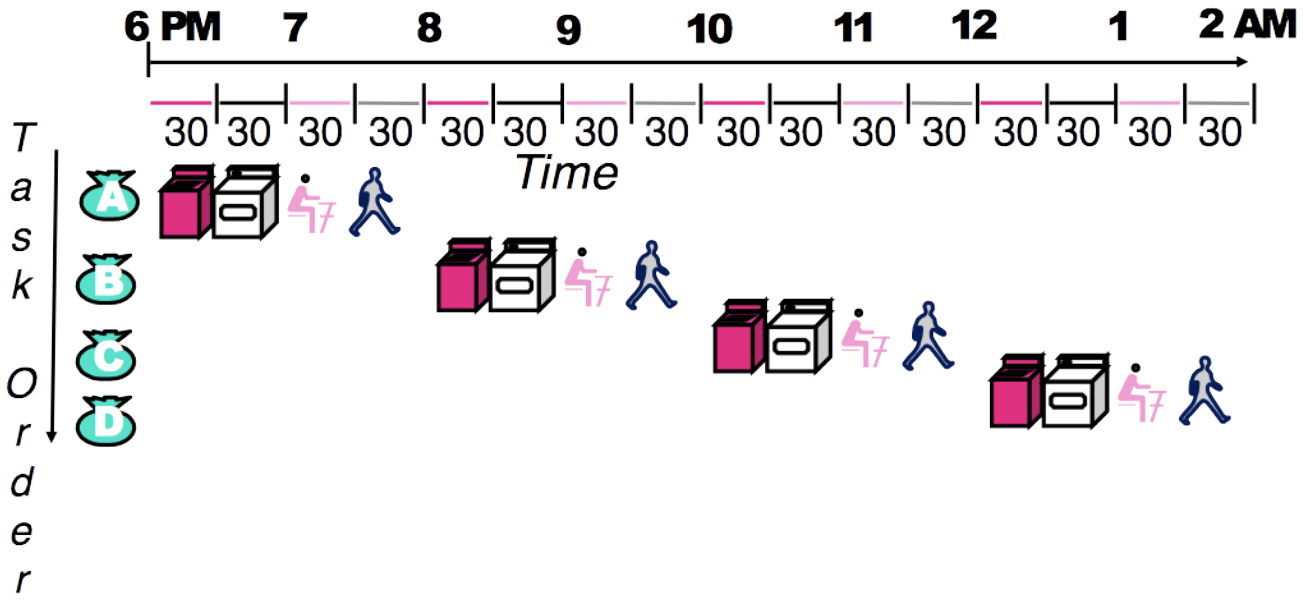






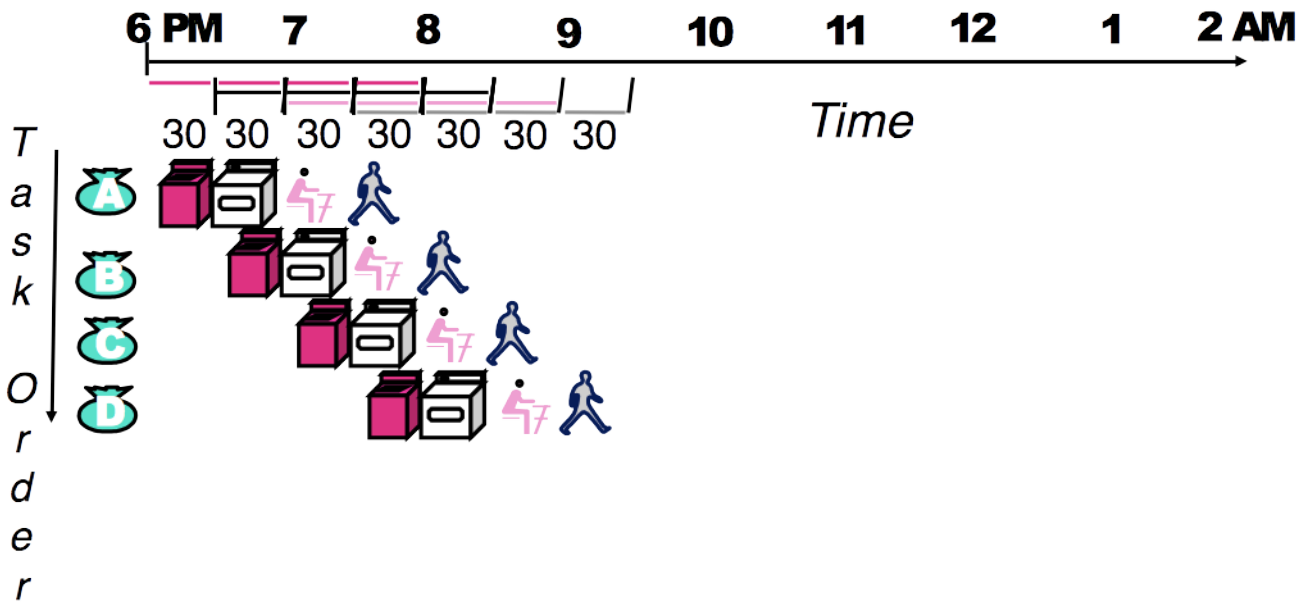
- We have 4 loads of laundry to do: A, B, C, D.
- To process each load we need to:
  - ↳ Wash
  - ↳ Dry
  - ↳ Fold
  - ↳ Put-away
- Each stage of doing laundry takes 30 minutes.
- Could process each load sequentially or use *pipelining*.

Doing Laundry: Sequentially



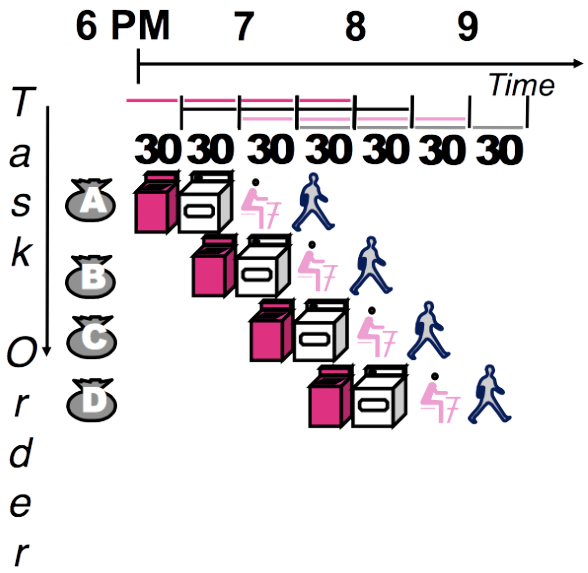
- Each load of laundry is done one at a time:
  - ↳ Wash A, Dry A, Fold A, Put-away A.
  - ↳ Wash B, Dry B, Fold B, Put-away B.
  - ⋮
- Takes 8 hours in total. There has to be a better way.

Doing Laundry: Pipeline



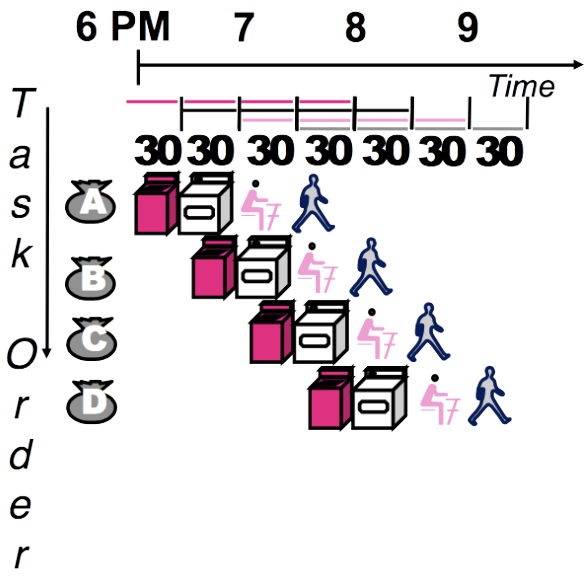
- Each *stage* of doing laundry must process each load sequentially.
- **But** each load of laundry can overlap.
- No dependency between drying load A and washing load B, etc.
- Put-away A while Folding B while drying C while washing D.
- Takes 3.5 hours in total.

Pipelining Terms via Analogy



- Pipelining: many tasks (loads of laundry) being executed simultaneously using different resources (washer, dryer, etc.).
- Time to complete a single task (**latency**) *does not* change.
  - ↳ Each load by itself still takes 2 hours.
- Number of tasks that can be completed in one unit of time (**throughput**) increases.
- Potential speed up via pipelining equals the number of stages in pipeline.
- Actual speed-up never exactly equals potential.

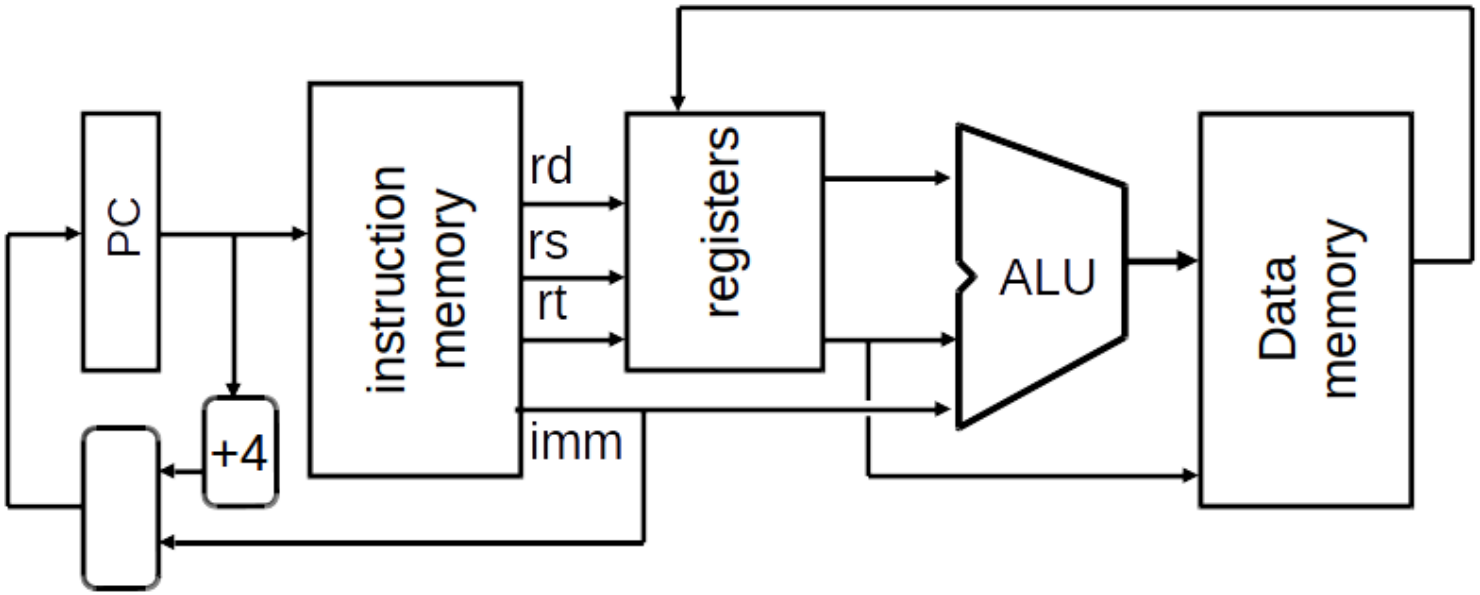
Pipelining Terms via Analogy



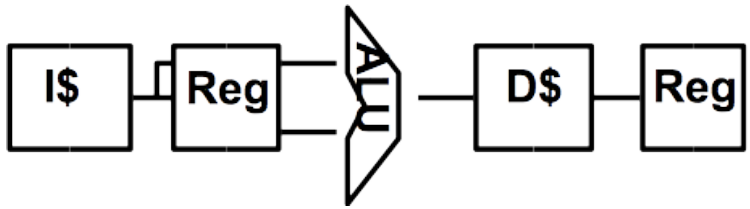
- Actual speed-up never exactly equals potential.
- **Fill time:** time taken to "fill" the pipeline. Initially, not every stage is used.
- **Drain time:** time taken to "empty" the pipeline. Not all stages are used once the last task begins.
- Imagine a new washing machine takes only 20 minutes. This *does not* increase pipeline speed.
  - ↳ Dryer still takes 30 minutes.
  - ↳ Washer must wait for dryer to finish before laundry can move from washer to dryer.

Outline

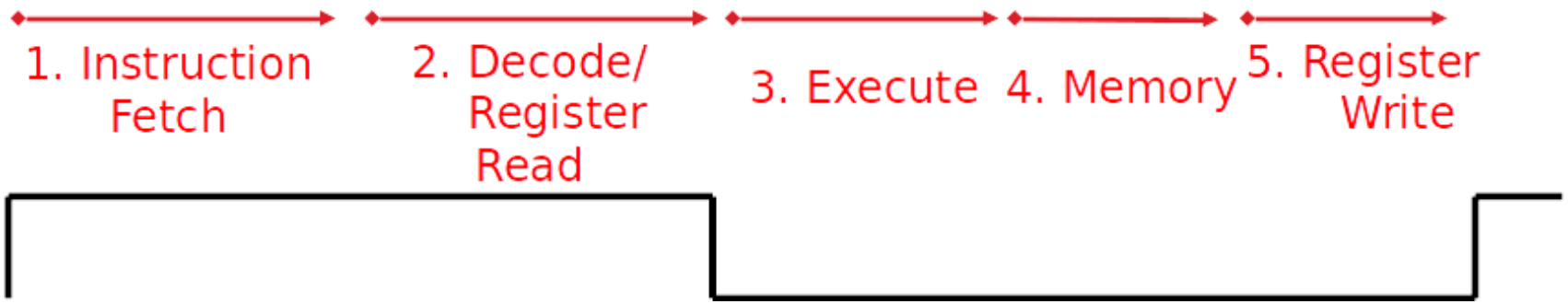
The RISC Datapath



IF ID EX MEM WB



Review: Single Cycle Datapath



- Clock cycle is long enough to handle *critical path* through datapath.
- Time for data to pass through entire datapath.

Performance of Single Cycle Datapath

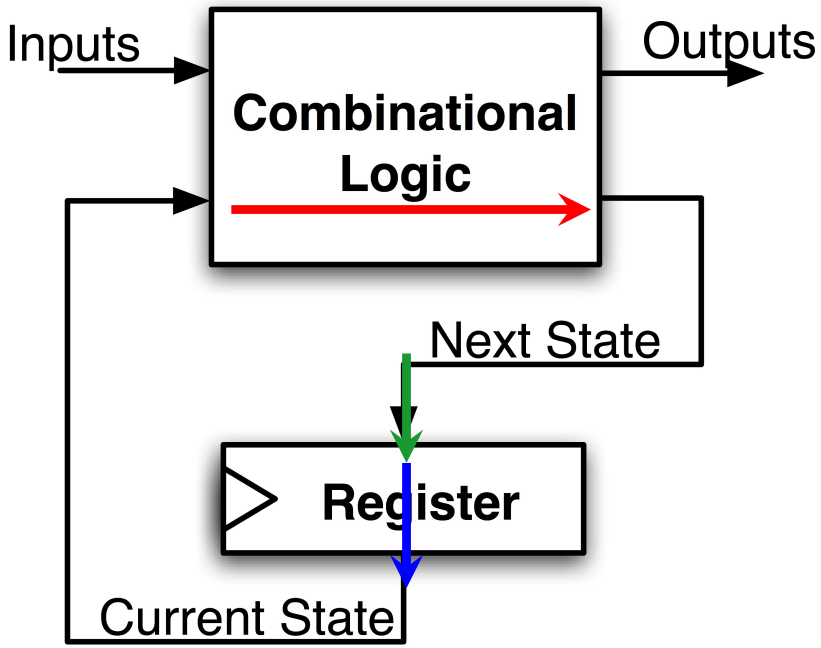
- Let's assume that accessing memory takes 200ps and ALU propagation delay is 200ps.
  - ↳ IF stage, EX stage, MEM stage.
- Let's assume accessing registers takes 100ps.
  - ↳ ID stage, WB stage.

- **What is the minimum clock cycle?**
  - ↳ Sum of all stages since some instructions use all stages.
  - ↳ $200 + 100 + 200 + 200 + 100 = 800$ps.

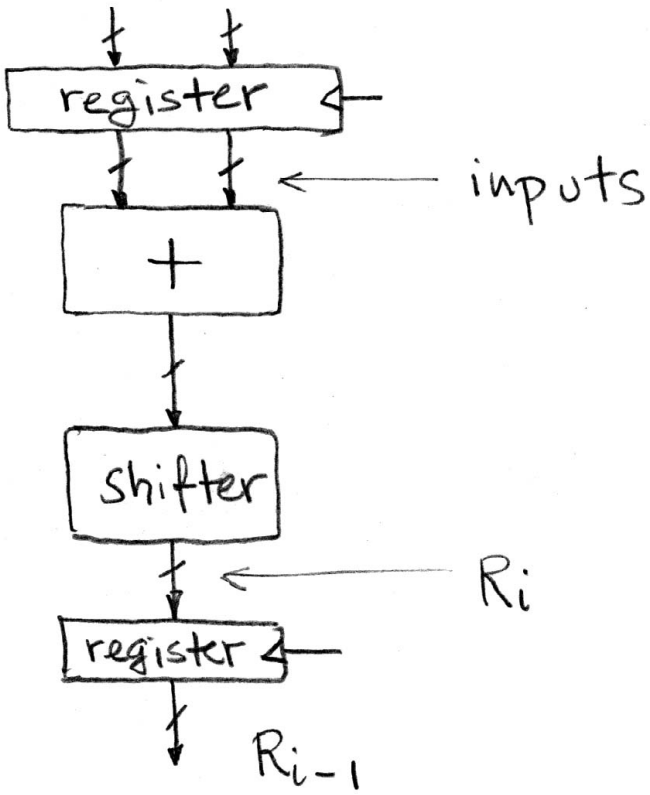
Instr.	IF	ID	EX	MEM	WB	Total
R-type	200ps	100ps	200ps	-	100ps	600ps
Branch	200ps	100ps	200ps	-	-	500ps
sw	200ps	100ps	200ps	200ps	-	700ps
lw	200ps	100ps	200ps	200ps	100ps	**800ps**

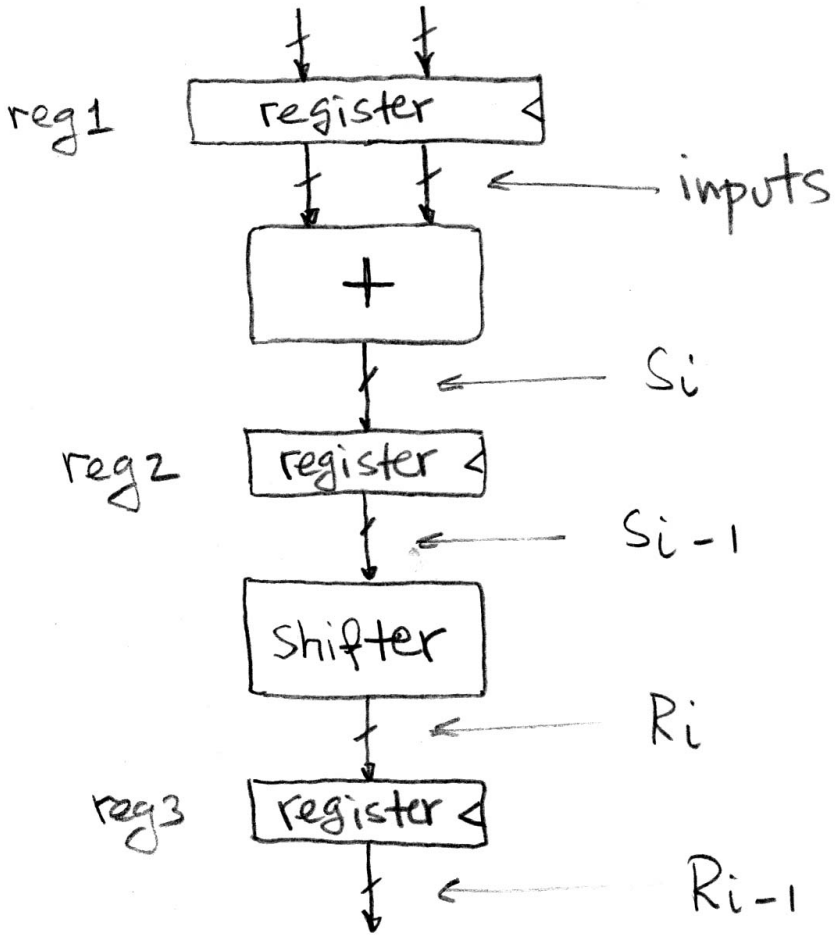
Improving Performance of Datapath

- **Clock frequency**
- Parallel execution of instructions via overlap: **pipelining**.
- Superscalar, VLIW (to come later).
- Branch prediction (to come later).



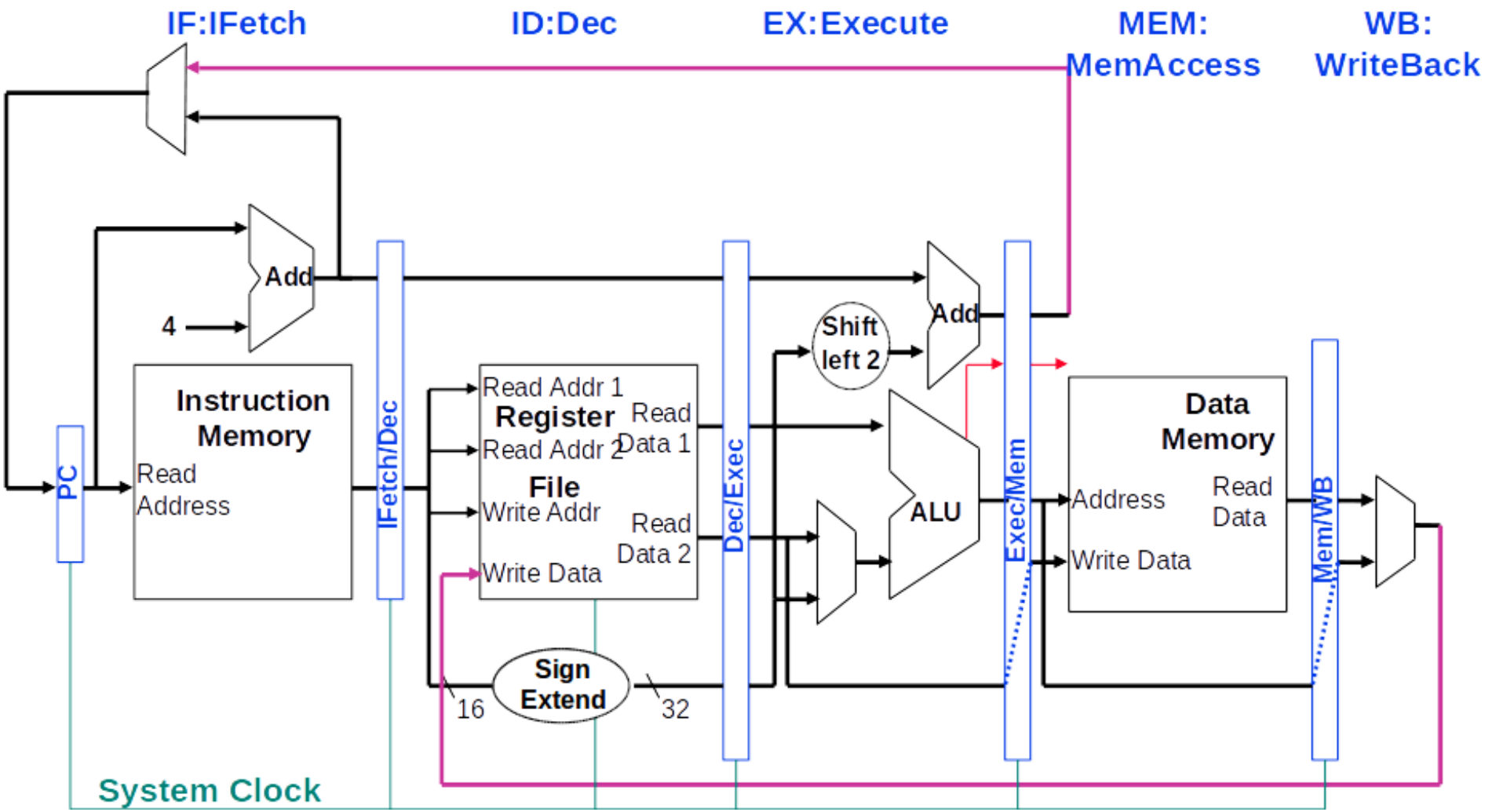
Review: Pipelining for Combinational Circuits



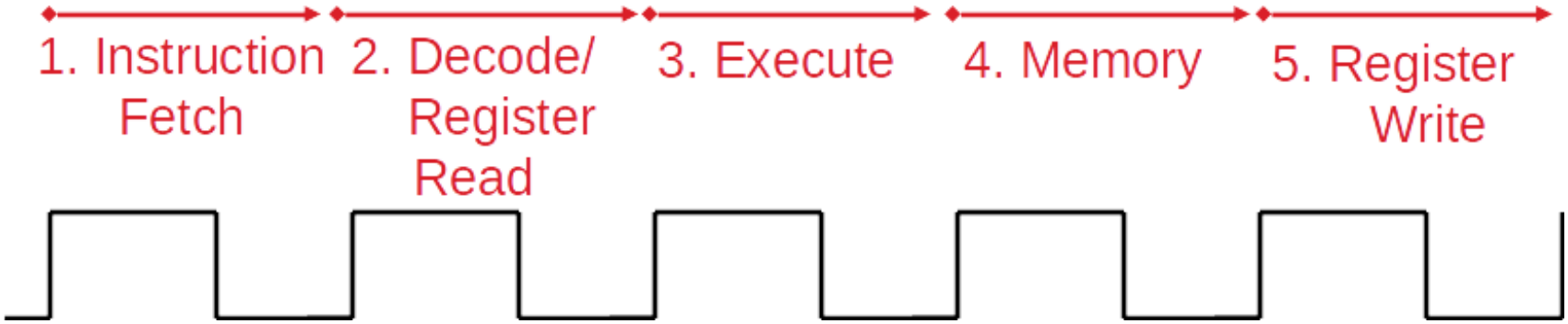


Break up a combinational circuit, reduce propagation delay, insert a register to store intermediate results, increase clock frequency.

Pipelining for MIPS



Multi-Cycle Datapath



- Clock cycle is long enough to handle *slowest stage* of the pipeline.
- Time for data to pass through one (the slowest) stage of pipeline.

Example: Minimum clock cycle is 200ps.

Instr.	**IF**	ID	**EX**	**MEM**	WB	Total
R-type	**200ps**	100ps	**200ps**	-	100ps	600ps
Branch	**200ps**	100ps	**200ps**	-	-	500ps
sw	**200ps**	100ps	**200ps**	**200ps**	-	700ps
lw	**200ps**	100ps	**200ps**	**200ps**	100ps	800ps

Pipelining for Performance

- Further increase clock frequency?
- *Could* break up datapath into more and more stages but...
  - ↳ More registers.
  - ↳ More complexity in datapath and controller design ⇒ overhead.
  - ↳ Still limited by slowest stage (memory).
- Leverage the parallelism gained by pipelining.
- Parallelism in execution of instructions yields fewer *cycles per instruction* (CPI)

The Classic Performance Equation

$$\begin{aligned} \text{CPU time} &= \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle} \\ &\text{or} \\ \text{CPU time} &= \text{Instruction\_count} \times \text{CPI} / \text{clock\_rate} \end{aligned}$$

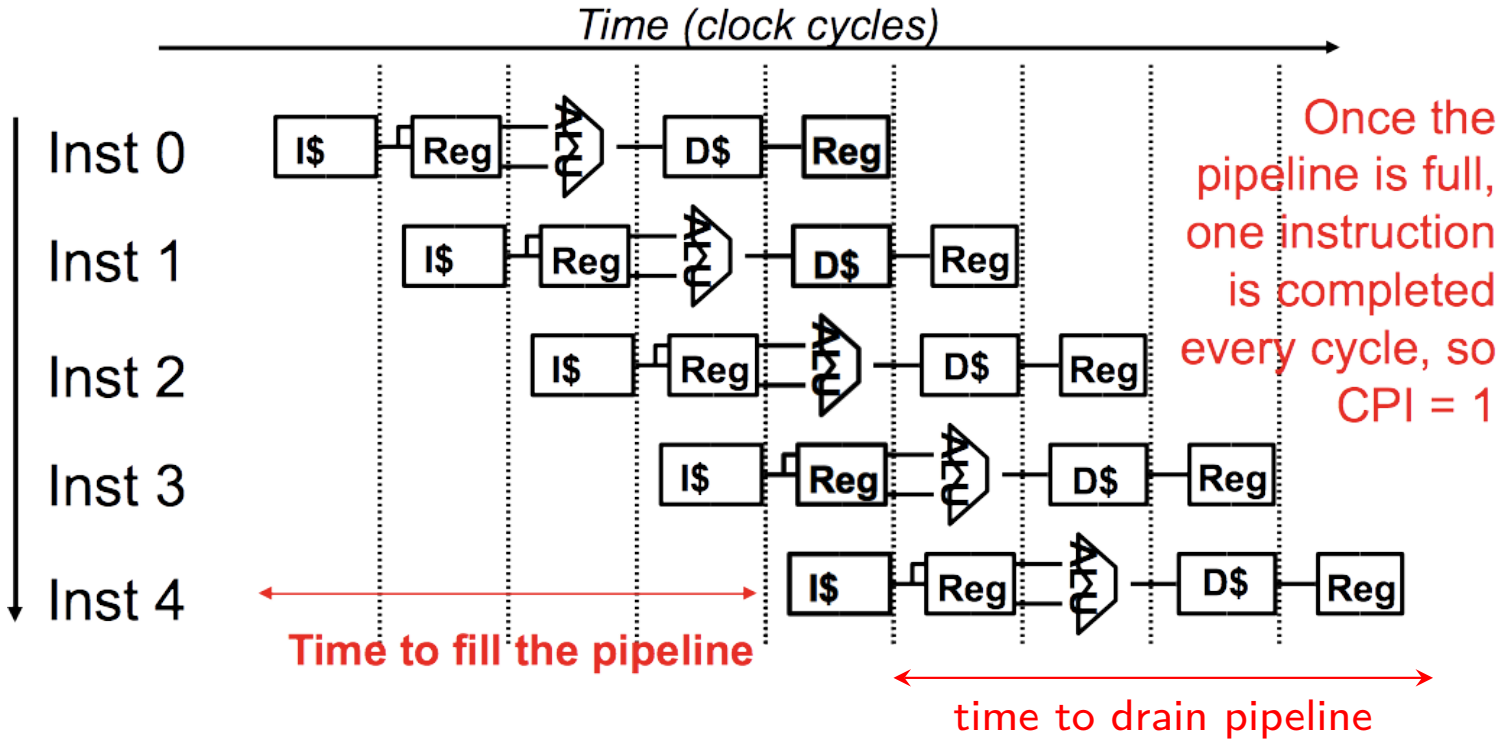
RISC Pipeline Performance

- Overlap instructions, start the next before the former completes.
- Some instructions will "waste" a cycle as they flow through unused stages.

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
`lw`	IFetch	Dec	Exec	Mem	WB			
`sw`		IFetch	Dec	Exec	Mem	WB		
`add`			IFetch	Dec	Exec	Mem	WB	

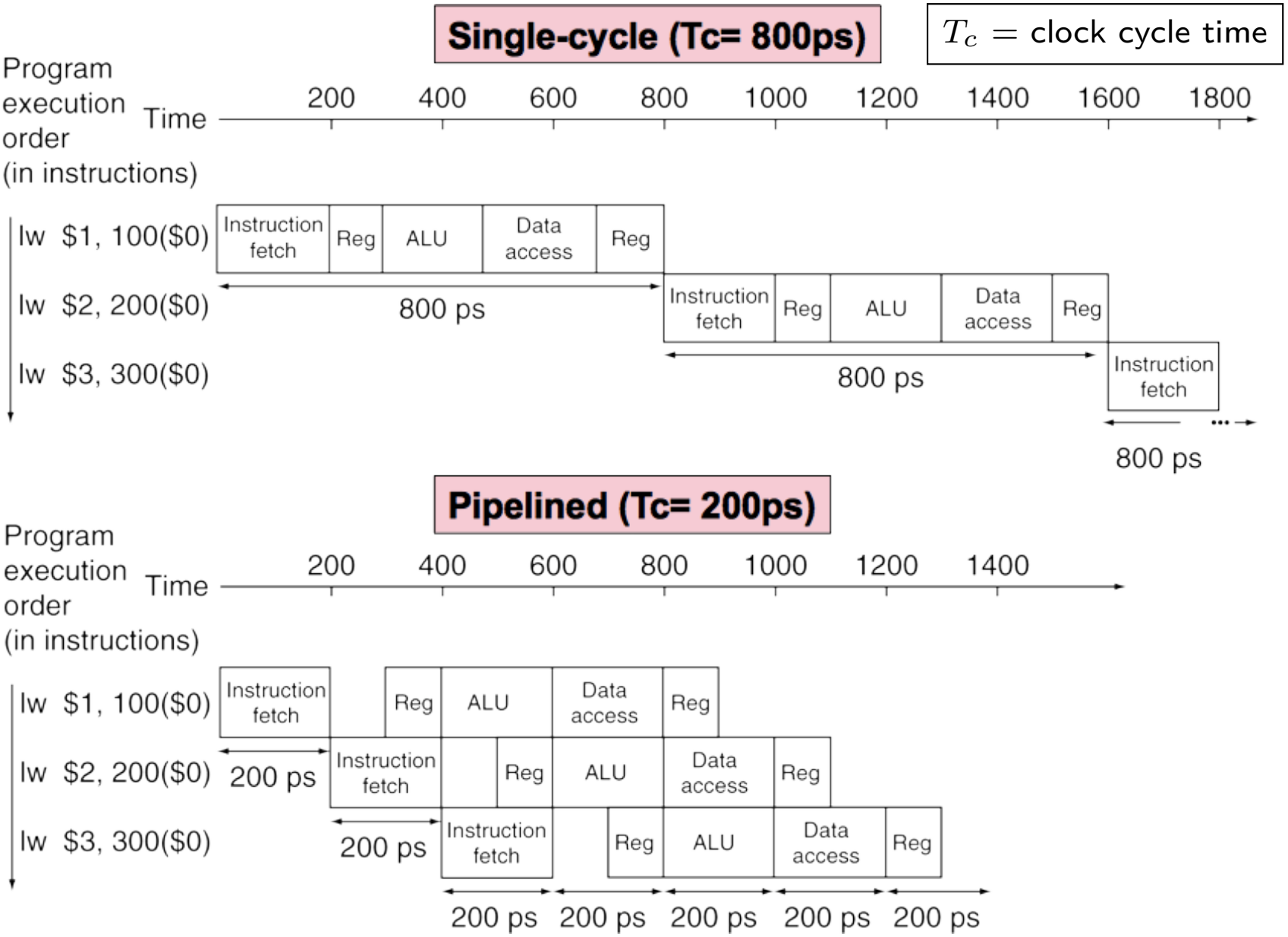
- **Latency:** time to complete one instruction. Does not change with pipelining.
- **Throughput**: number of instructions that can be completed in some amount of time. Increases with pipelining.
- Once pipeline is full CPI is 1.

Pipeline Parallelism



- Potential speed-up via parallelism is equal to the number of stages.
- 5 stages $\Longrightarrow$ 5x potential speed up.
- A pipeline is "full" when every stage is occupied by an instruction (every stage does not have to necessarily be doing work).
- Pipeline **fill time** and **drain time** reduce actual speed up.

Performance: With and Without Pipelining



Quantifying Pipelined Speedup

If the time for each stage is the same:

$$\textbf{Ideal Speedup} = \text{Number of Stages}$$

If the time for each stage is not the same:

$$\textbf{Ideal Speedup} = \frac{\text{Time between instructions}_{\text{non-pipelined}}}{\text{Time between instructions}_{\text{pipelined}}}$$

$$\textbf{Actual Speedup} = \frac{\text{Time to complete}_{\text{non-pipelined}}}{\text{Time to complete}_{\text{pipelined}}}$$

Calculating Speedup

From previous example:

- Single-cycle datapath: 800ps clock cycle.
- Pipelined: 200ps clock cycle.
- **Uneven time for each stage**. ID and WB only 100ps.
- 3 `lw` instructions.

$$\text{Ideal Speedup} = \frac{800}{200} = 4 \qquad \text{Actual Speedup} = \frac{2400}{1400} = 1.714$$

- If we have 1000000 `lw` instructions?

$$\text{Actual Speedup} = \frac{1000000 \times 800}{1000000 \times 200 + 800} \approx 4$$

Calculating Pipelined Time

Classic Performance Equation:

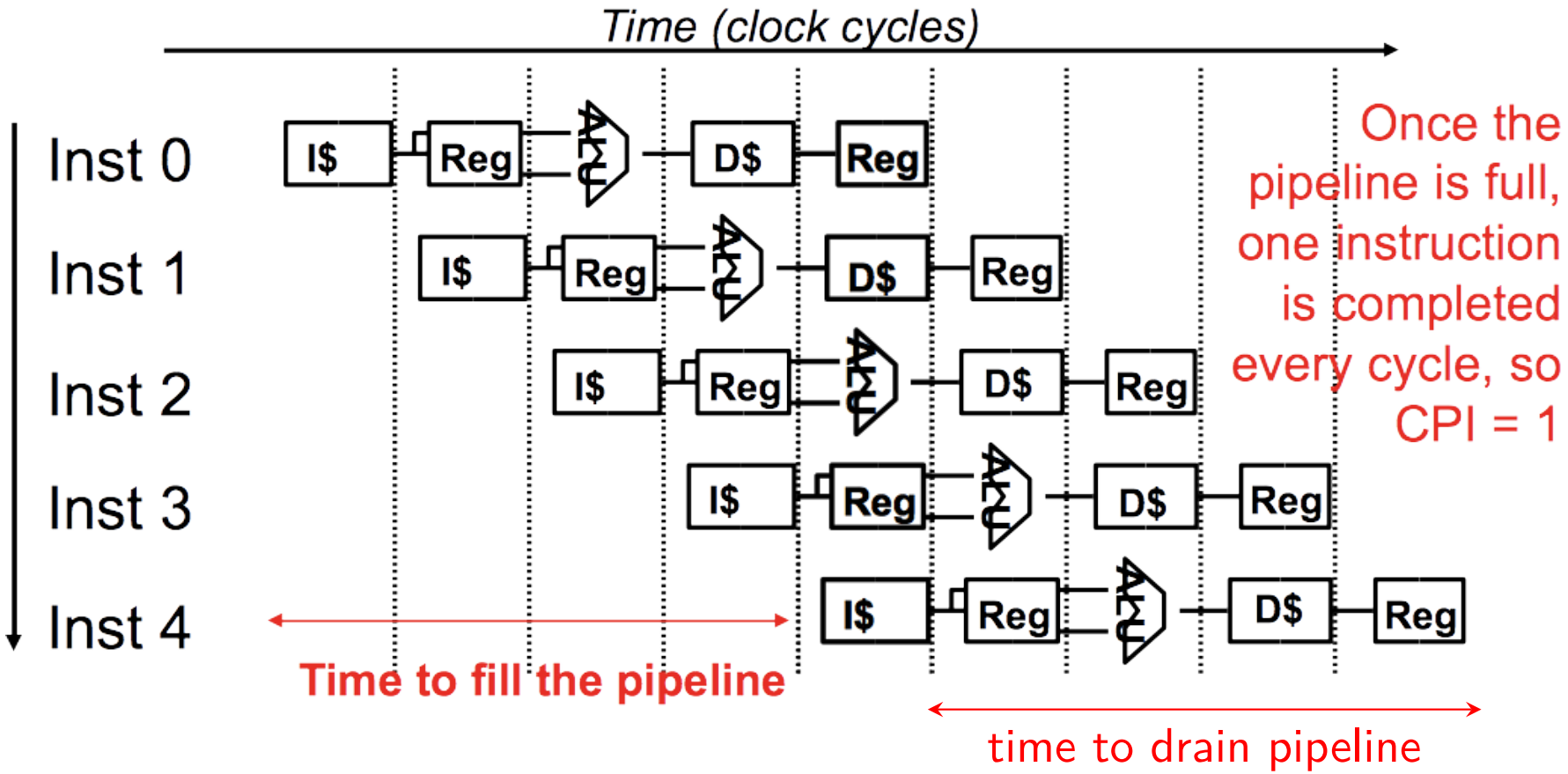
$$\text{CPU time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock cycle}$$

Time for pipelined execution:

$$\text{Time}_{\text{pipelined}} = \text{Fill time} + (\text{IC} \times \text{clock cycle})$$

- Once pipeline is full, one instr. completes every cycle ⇒ CPI is 1.
  - ↳ Gives IC × 1 × clock cycle
- Pipeline is only *not* full during fill or drain time.
- Fill time = Drain time = (number of stages - 1) × clock cycle
  - ↳ Assuming number of instructions > number of stages.

Calculating Pipelined Time



Summary

- Pipelining is the simultaneous execution of multiple instructions each in a different stage of the datapath.
- Pipelining gives increased clock frequency by multi-cycle datapath.
- Limited by the slowest stage.
- Pipelining gives essentially a CPI of 1.
- Speed-up must account for fill time and drain time.
- All of the discussion so far assumed there is **no conflicts** between instructions, hardware, circuits, etc.
  - ↳ **Pipeline hazards** severely impact performance and potential speed-up.
  - ↳ Chapter 4: Part 2: Pipeline hazards.

CS3350B Computer Organization
Chapter 4: Instruction-Level Parallelism
Part 2: Pipeline Hazards

Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Thursday March 14, 2019

Outline

Pros and Cons of Pipelining

- Pipelining overlaps the execution of instructions to keep each stage of the datapath busy at all times.
  - ↳ Improves **throughput** but not **latency**.
  - ↳ Might actually *increase* latency.
- Can increase clock frequency using multi-cycle datapath.
- Ideal speedup can be up to the number of stages.
- Ideal speed up *never* reached.
  - ↳ **Fill time** and **drain time** limits speedup.
  - ↳ Must account for dependencies between results of previous instructions and operands of future instructions.
  - ↳ Sometimes the same hardware is needed simultaneously by different pipeline stages and different instructions (e.g. ID and WB stages).

Categorizing Pipeline Hazards

Structural Hazards

- Conflicts in hardware/circuit use.
- Different stages or different instructions attempt to use same piece of hardware at the same time.

Data Hazards

- Dependencies between the result of an instruction and the input to another instruction.
- Data being used before it is finished being computed or written to memory/registers.

Control Hazards

- Ambiguity in the control flow of the program being executed.
- **Branch instructions**—if/else, loops.
- Take the branch? Don't take the branch? Which instruction follows a branch instruction in the pipeline?

"Resolving" Pipeline Hazards

Not an easy task. Simplest solution: just wait or **stall**.

↳ Any hazard can always be solved by just waiting.

But:

- Ruins potential speedup.
  - ↳ Might end up being slower than a single-cycle datapath.
  - ↳ Since latency can increase in pipelining, with enough stalls becomes slower.
- Increases CPI.
- Works against entire principle of pipelining.
  - ↳ Where's the performance?
- Nonetheless, sometimes it really is the only solution.

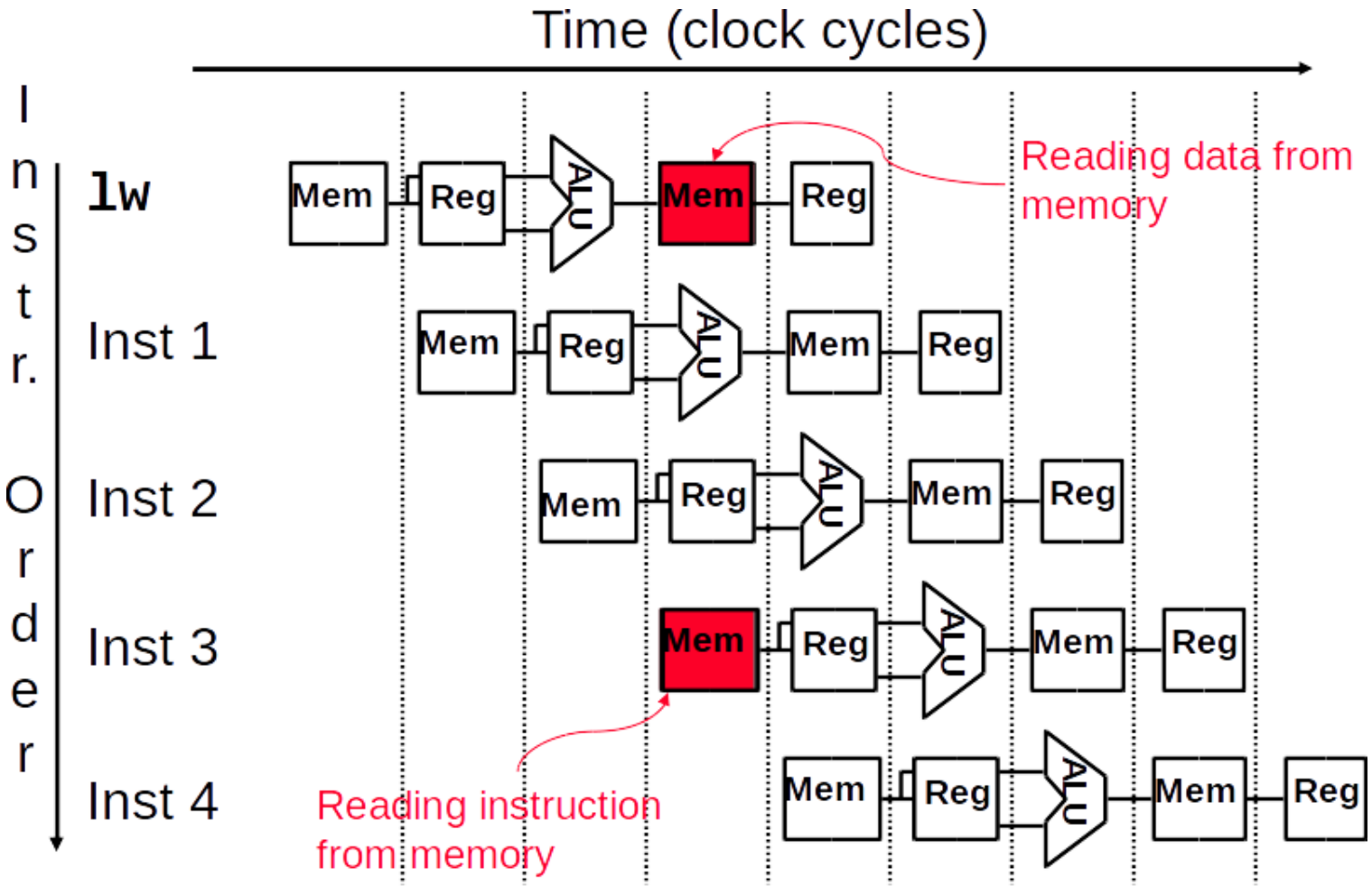
Outline

Structural Hazards: Causes and Resolutions

- Structural hazards are caused by two instructions needing to use the same hardware at the same time.
- Easiest to resolve? Just add in redundant hardware.
  - ↳ Works for combinational circuits.
  - ↳ Redundant memory would cause problems in needing to keep both consistent.
- Real structural hazards thus lie in state circuits: registers and memory.
  - ↳ IF stage and MEM stage.
  - ↳ ID stage and WB stage.

Structural Hazards In Memory (1/2)

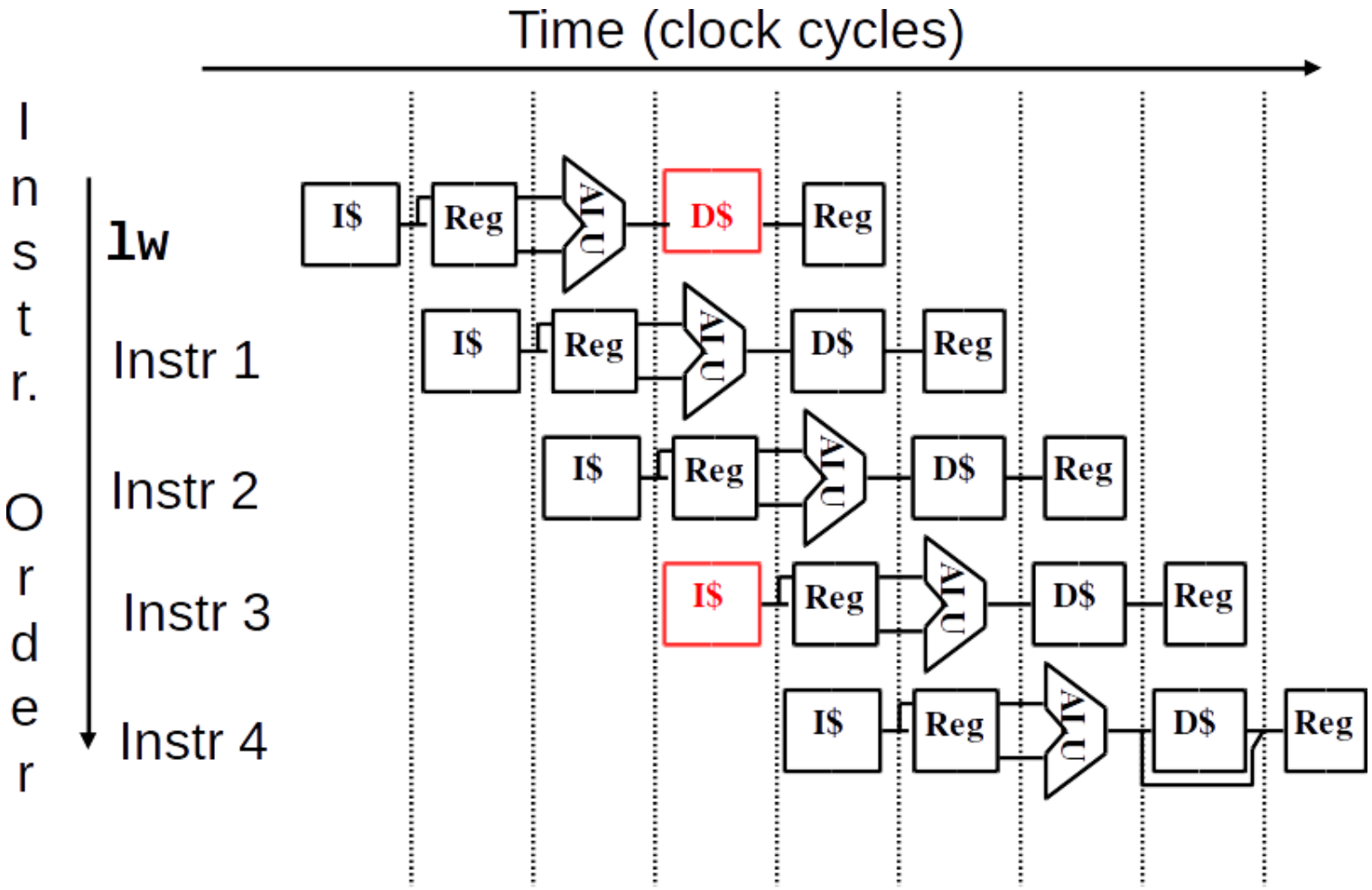
Consider a *unified* L1 cache. Reading instructions and reading/writing data could overlap for pipelined instructions.



Structural Hazards In Memory (2/2)

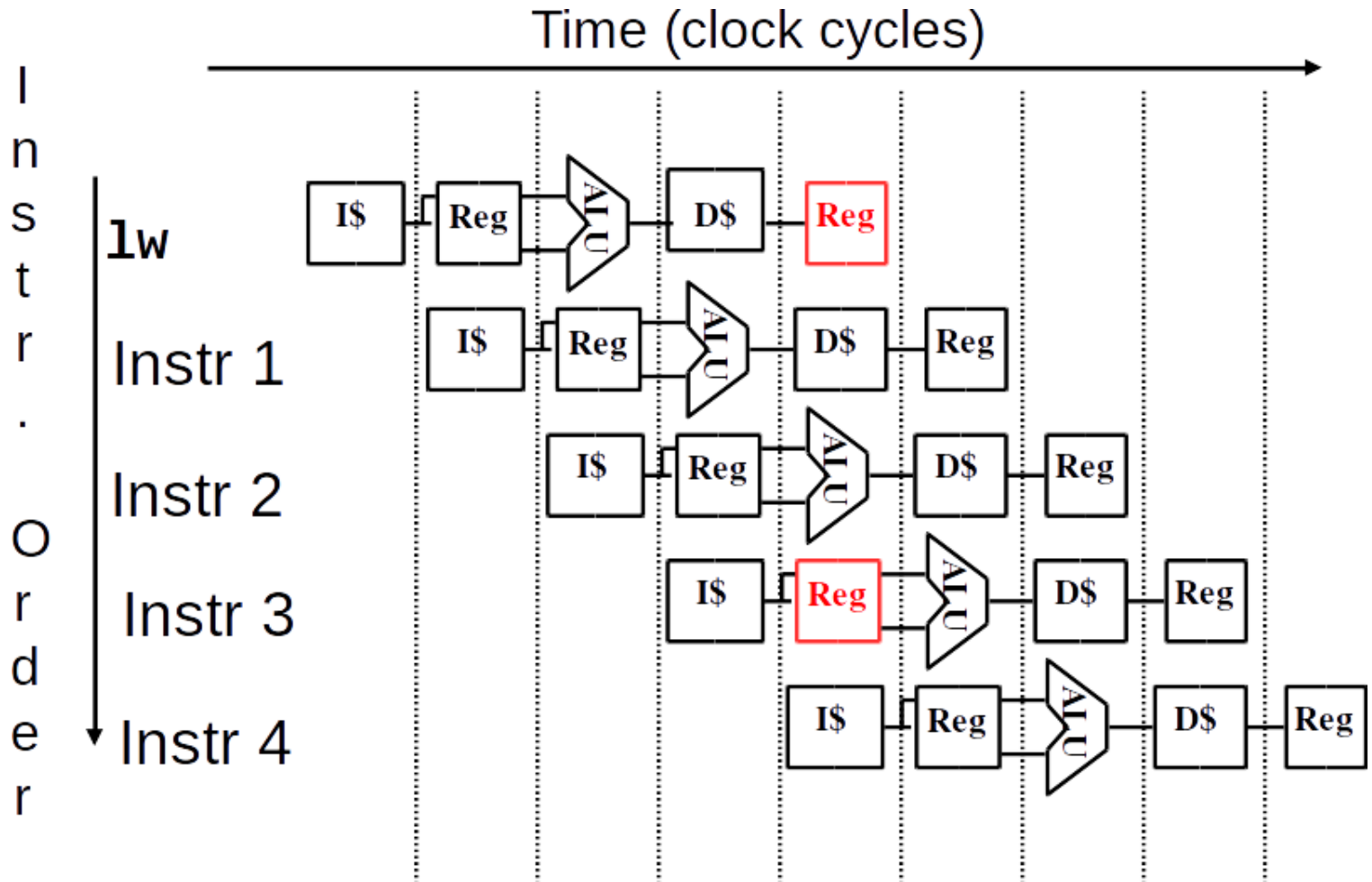
Simple fix: separate instruction memory from data memory.

- Can use a banked cache.



Structural Hazards In Register File (1/2)

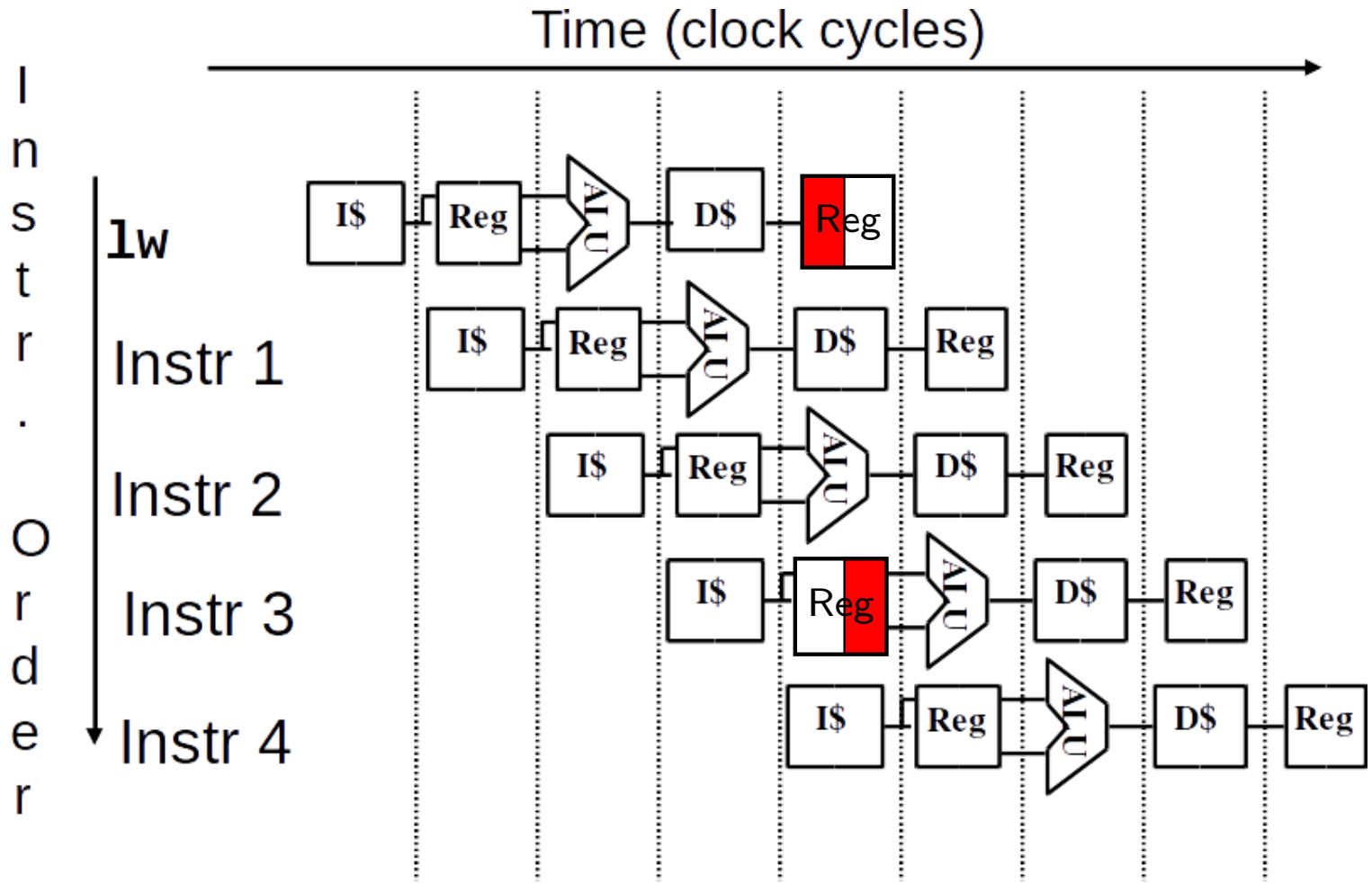
ID stage must read from registers while WB stage must write to registers.



Structural Hazards In Register File (2/2)

In reality, reading from register file is very fast; clock cycle is long enough to allow both ID and WB to occur within a single clock cycle.

- Needs independent read and write ports.



Outline

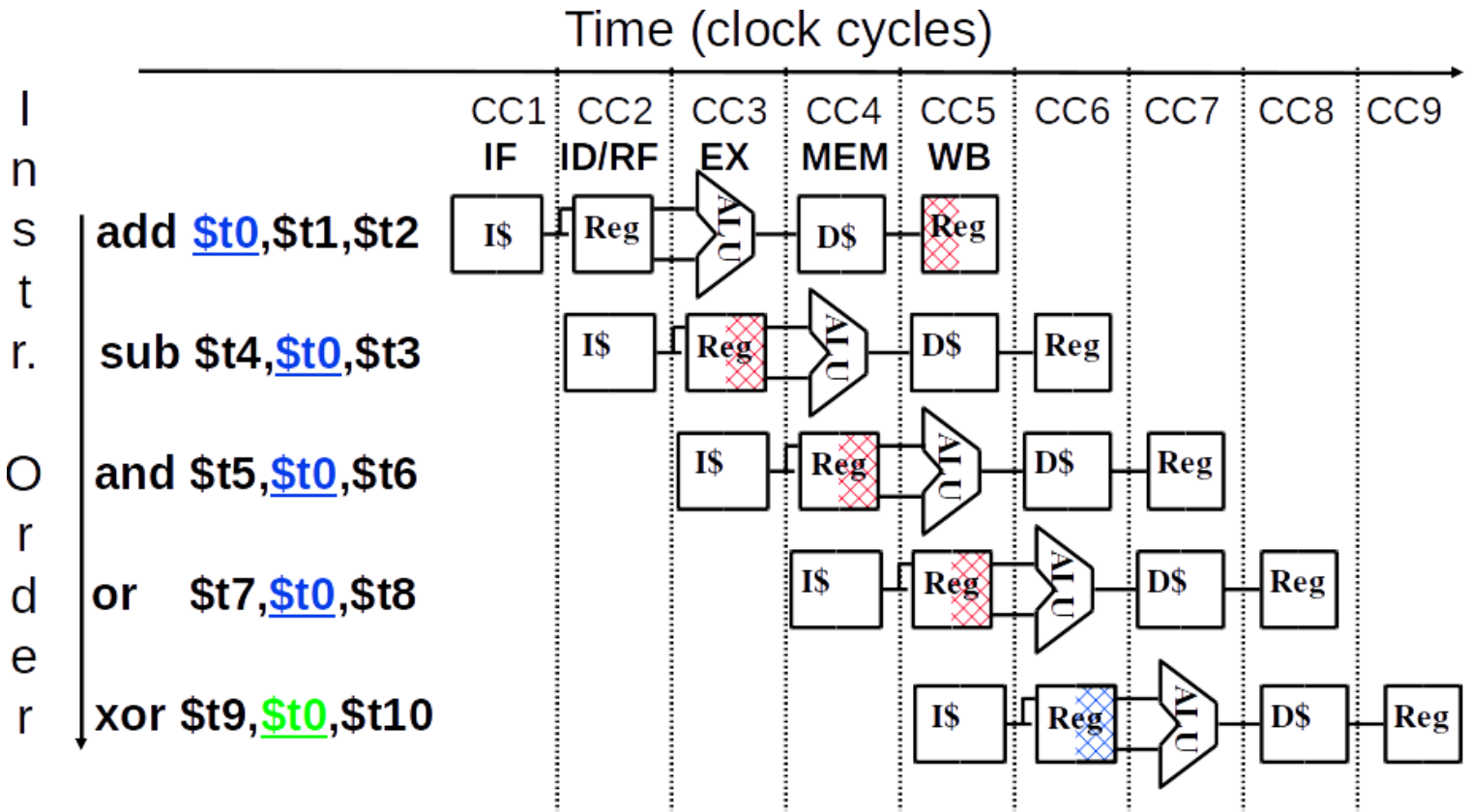
Data Hazards: Causes and Resolutions

- Data hazards are caused by dependencies between instruction operands or results.
  - ↳ Read After Write (RAW) only *true* dependency.
  - ↳ Read After Read not a hazard.
  - ↳ Write After Read (WAR) and Wriate After Write (WAW) only a hazard for **out-of-order execution** ⇒ Superscalar machines
  - ↳ Prelude to **register renaming**.
- Can always be solved by *stalling* the pipeline.
- Can be solved by special **forwarding** (also called bypass).
- Most common type of hazard.
  - ↳ It's the logical way to write programs; *locality*.

Data Hazard Example 1 (1/3)

`add` produces a result which is then read by `sub`, `and`, `or`, `xor`.

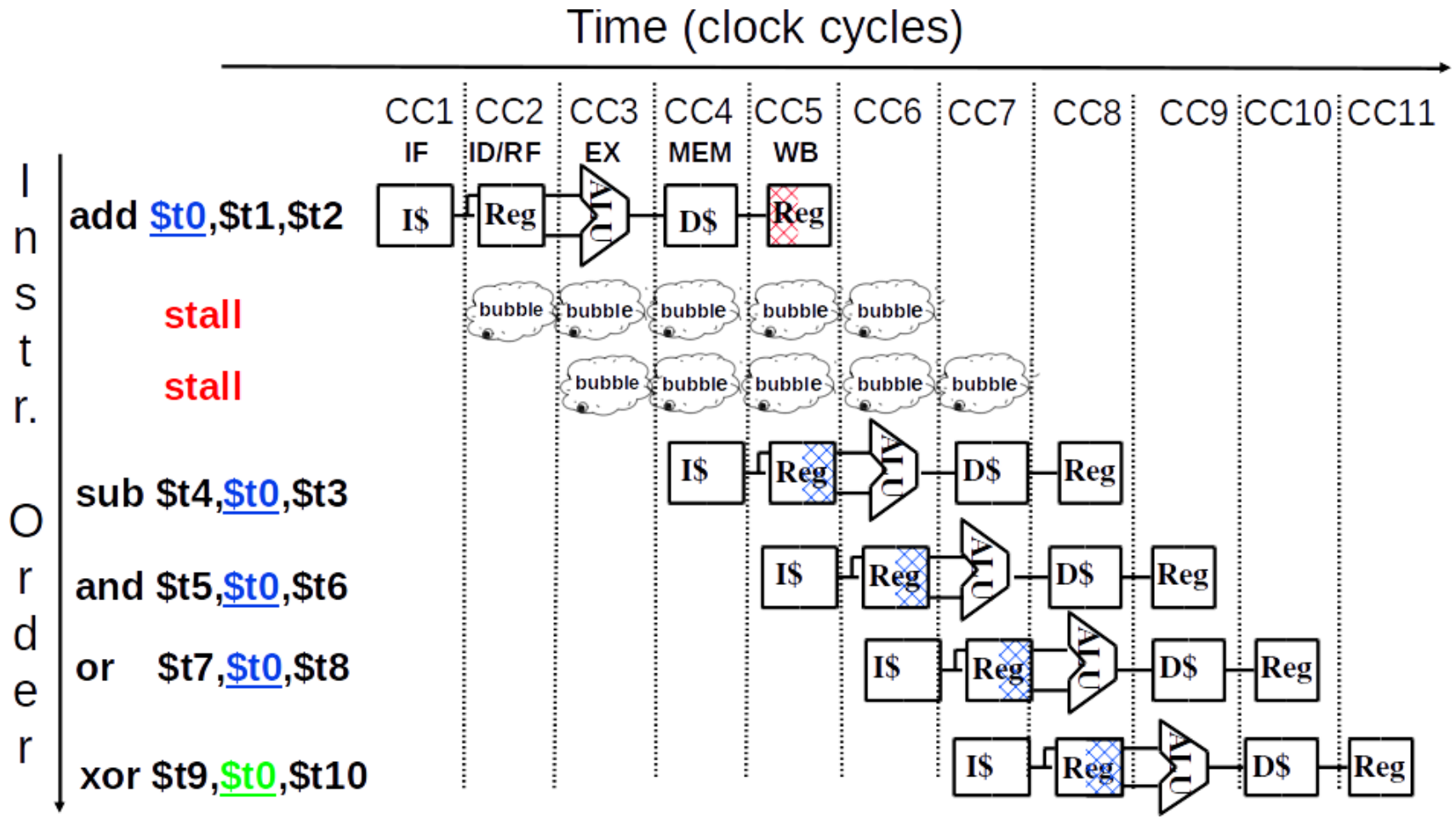
- Read After Write hazard.
- `xor` is far enough in the future to be okay.
- `sub`, `and`, `or` need more work.



Data Hazard Example 1 (2/3)

Possible (but not great) solution: **stall** the execution.

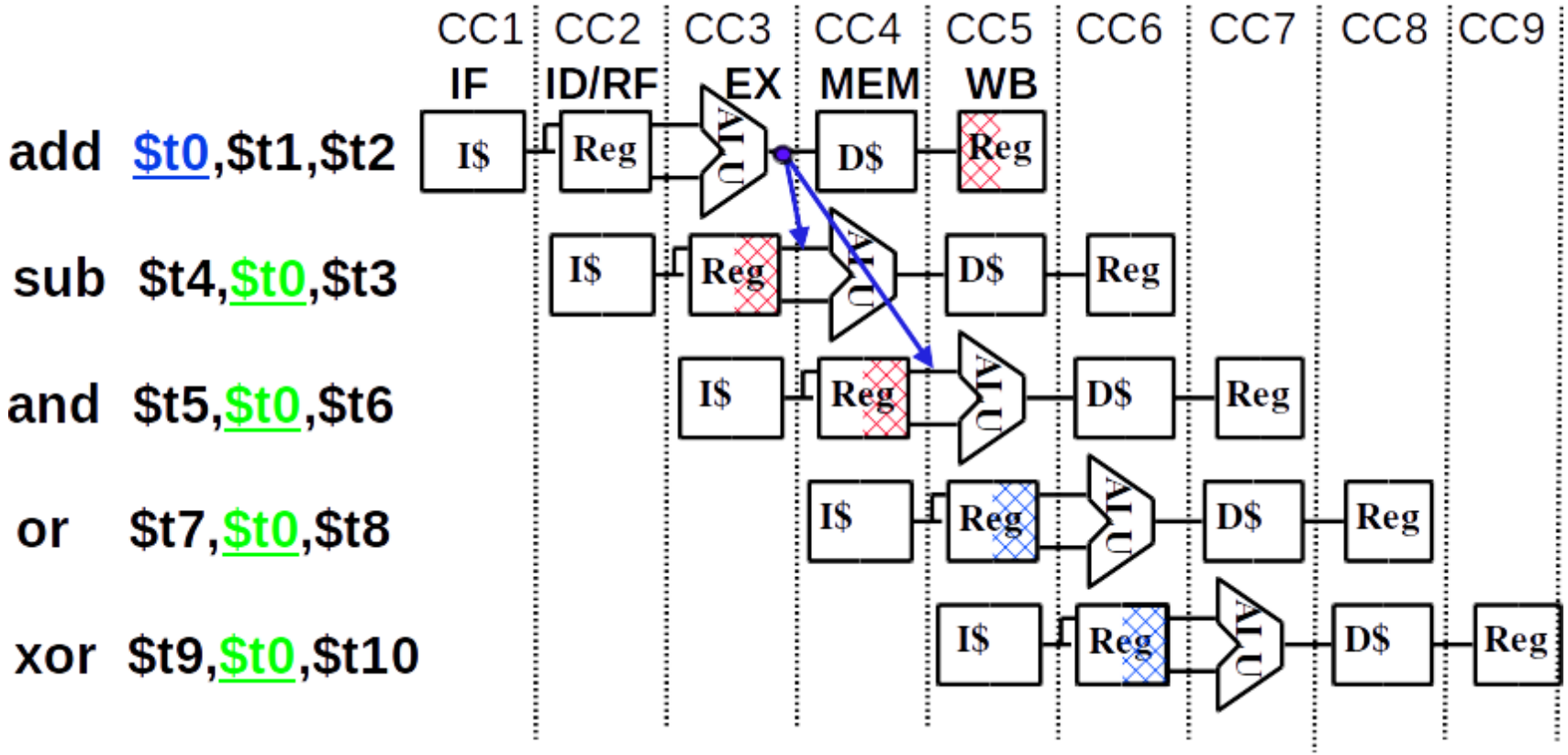
- `sub` structural hazard already solved.



Data Hazard Example 1 (3/3)

Another possible solution: **forwarding**.

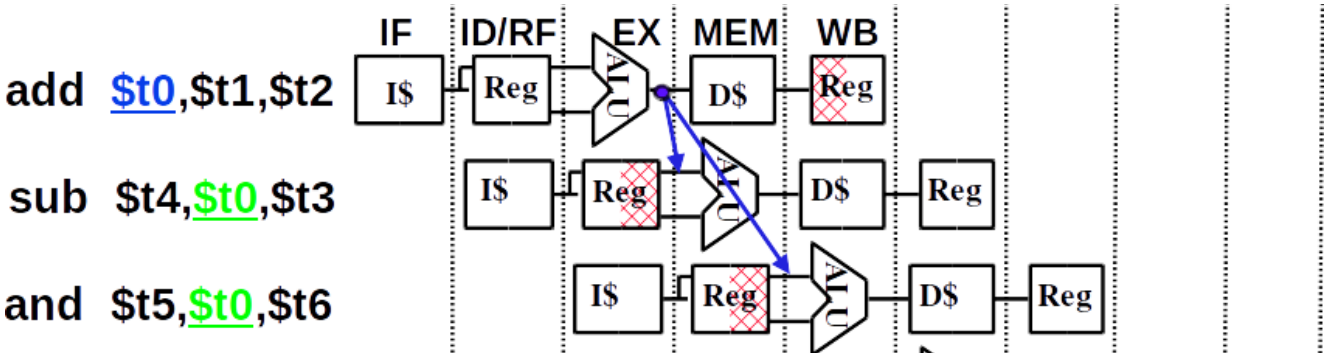
- No more stalls!
- ALU-ALU forwarding for `add` to `sub` and `add` to `and`.
- `or` structural hazard already solved.

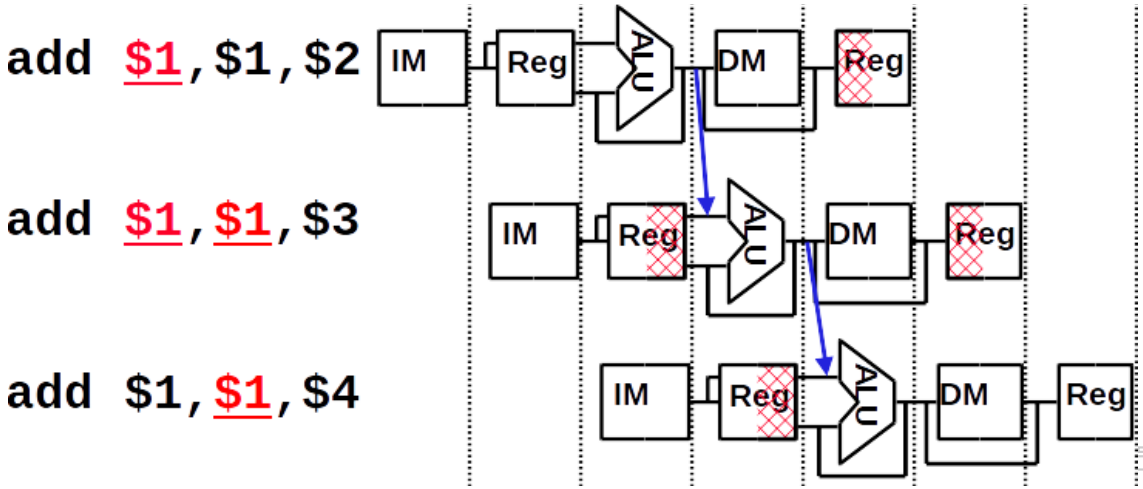


More ALU-ALU Forwarding

Two kinds of ALU-ALU forwarding:

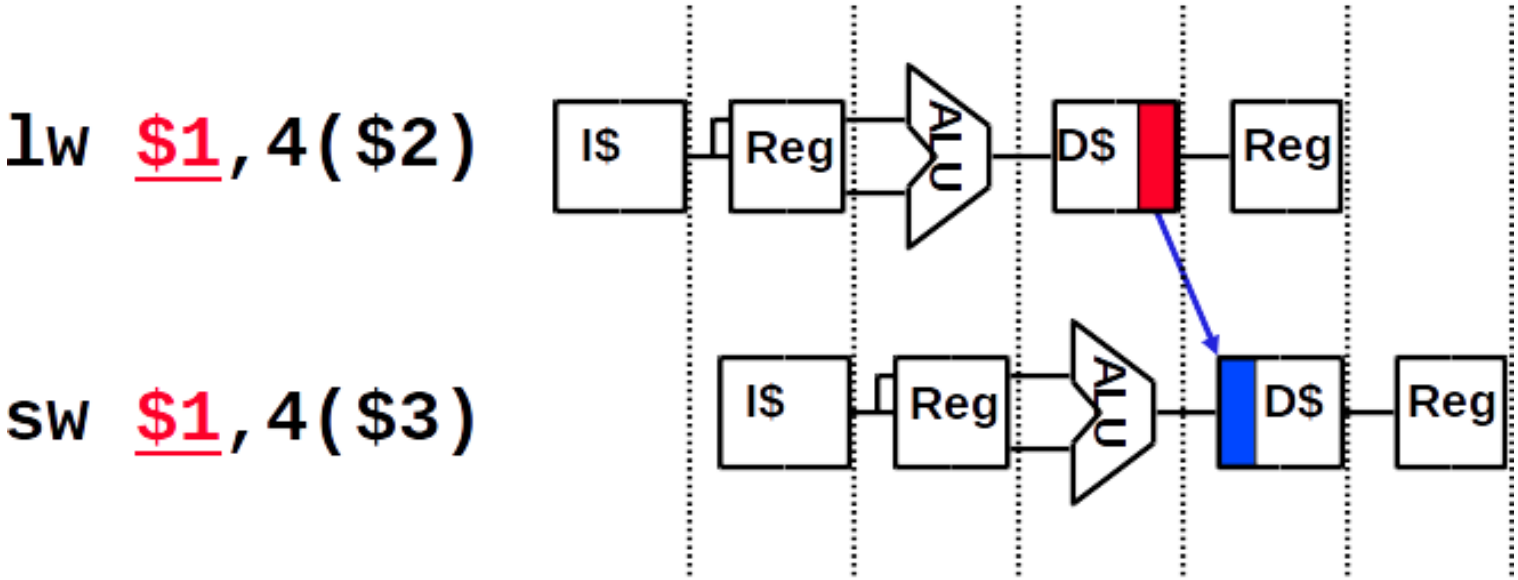
- Instruction currently in MEM stage to ALU.
- Instruction currently in WB stage to ALU.
  - ↳ Also called MEM-ALU forwarding.
- Which to choose? $\Longrightarrow$ More control, more MUX.





MEM-MEM Forwarding

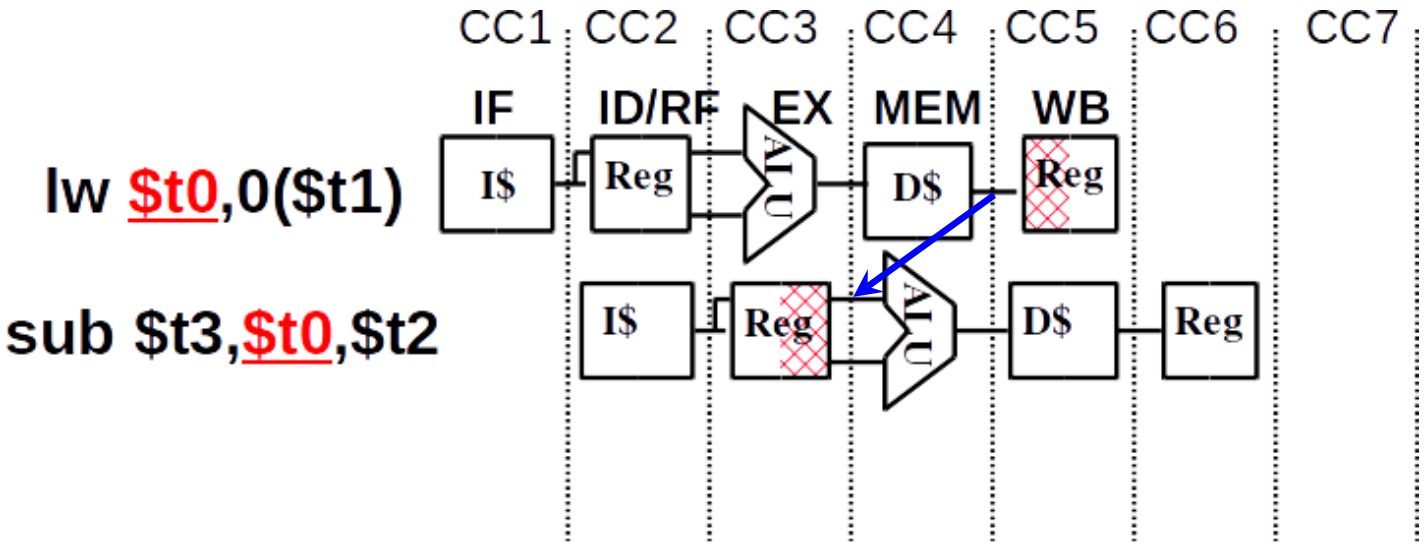
- For efficient memory copies (a common operation) this optimization results in no stalls.
  - ↳ Otherwise, two stalls required.
  - ↳ Eight great ideas in computer arch.: make the common case fast.



Load-Use Data Hazard

Load-use data hazard, a special kind of RAW hazard.

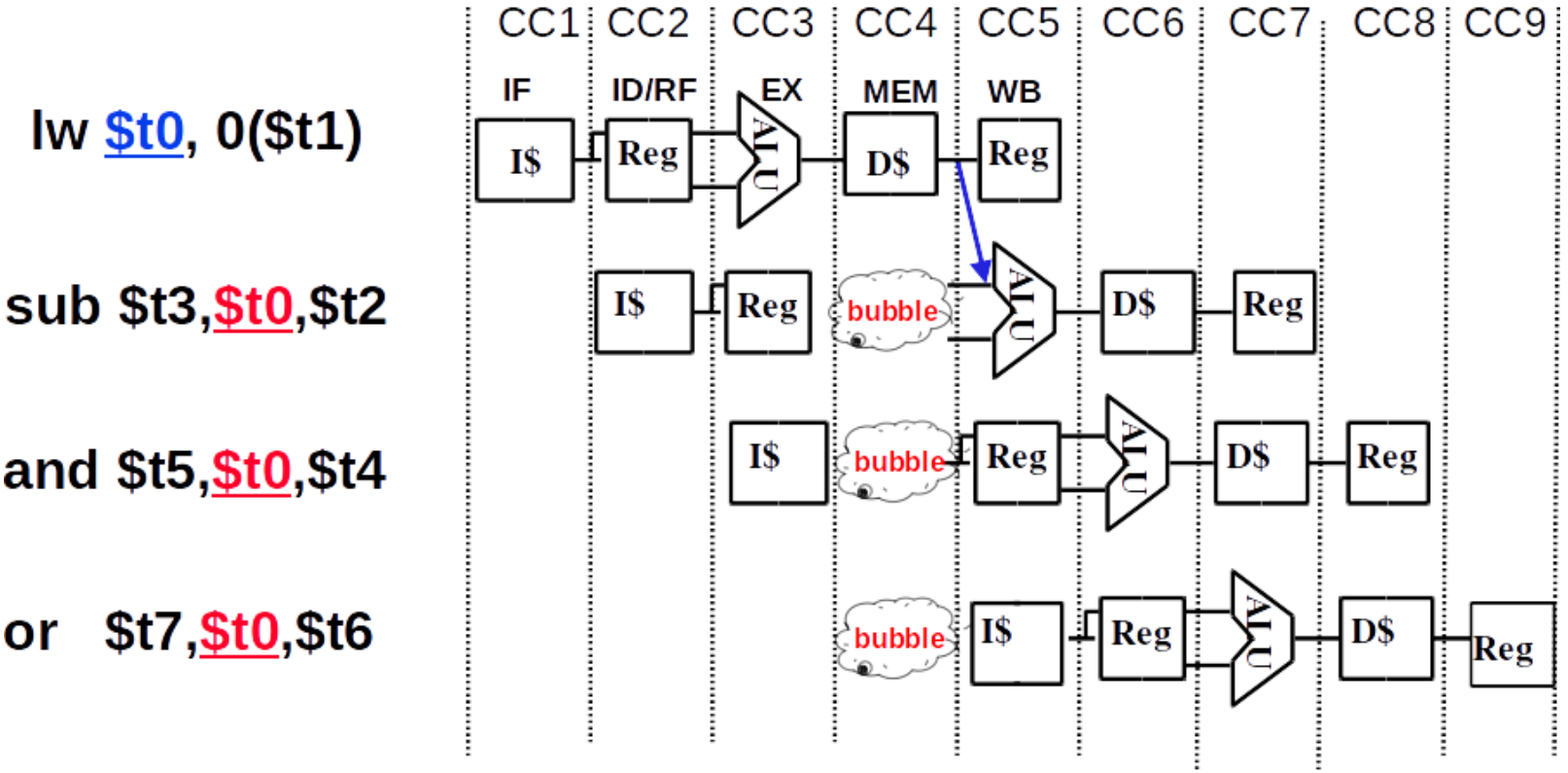
- Forwarding does not help here, still going backwards in time.
- A **stall** is required.



Implementing a Stall: Pipeline Interlock

Pipeline Interlock—**hardware** detects hazard and stalls the pipeline.

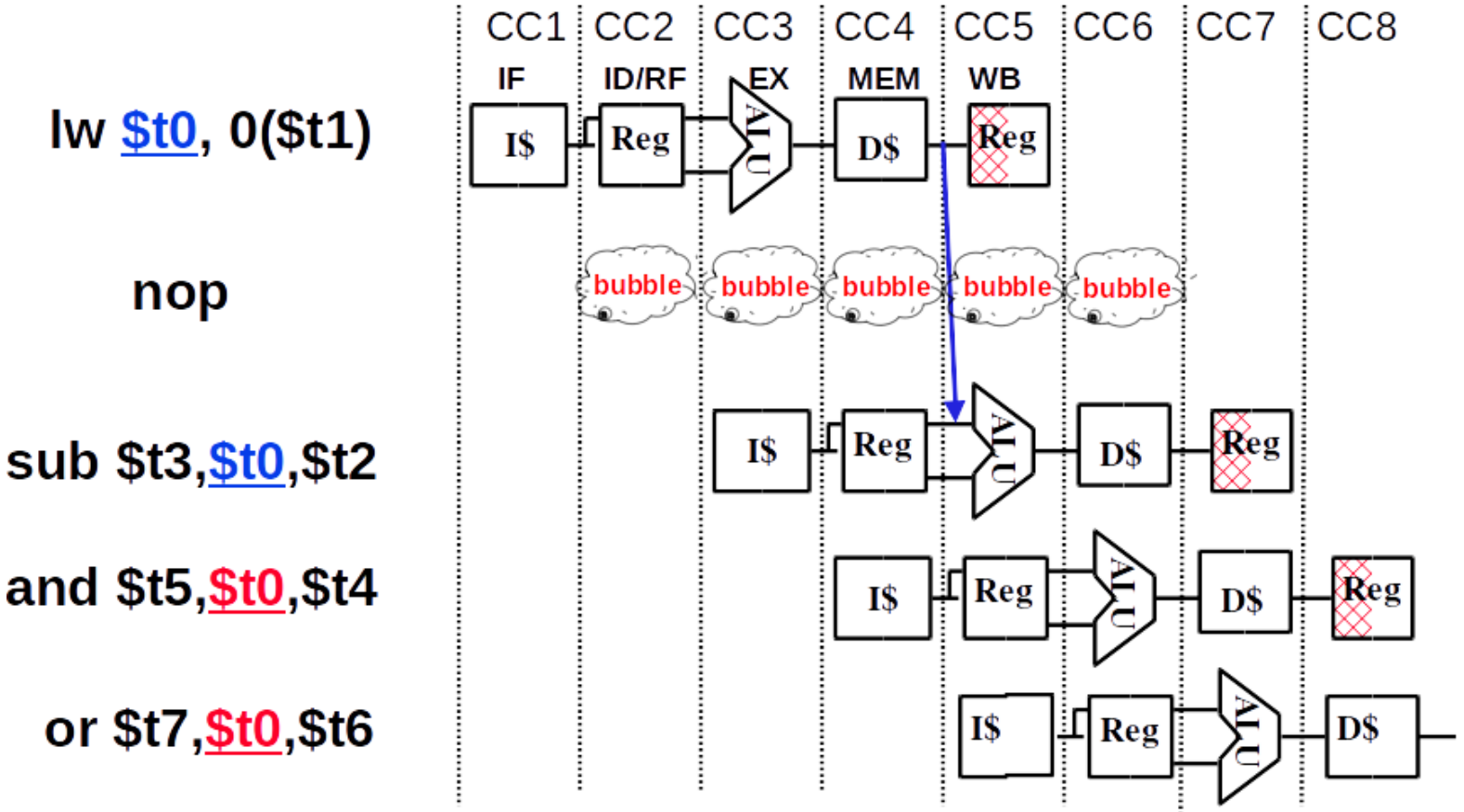
- Quite literally locks the flow of data between stages (locking writes to inter-stage registers).
- Essentially inserts an air bubble into pipeline.



Implementing a Stall: NOP

NOP—a "no operation" special instruction inserted into instruction flow by **compiler**.

- Hazards are detected and fixed at compile-time.
- Can be combined with forwarding; MEM-ALU in this case.



Pipeline Interlock vs NOP

- Interlocking requires special circuity to dynamically detect hazards and stall the datapath.
- `nop` requires extra effort at compile time to detect and resolve hazards.
- Inserted `nop` instructions bloat instruction memory.
- More work at compile time for `nop` insertion but simpler (= faster?) datapath and controller.
- MIPS: Microprocessor *without Interlocked* Pipelined Stages

Data Hazards and Code Structure

Some data hazards are "fake".

- Only caused by the order of instructions and not a true dependency.
- Re-order code (if possible) so an independent instruction performed instead of a `nop`.
  - ↳ Where the nop would be inserted is called the **load delay slot**.
  - ↳ Load delay slot can be filled with a nop or an independent instruction.
- Need at least one instruction between `lw` and using the loaded word.

```
        lw   $t1, 0($t0)          lw   $t1, 0($t0)
        lw   $t2, 4($t0)          lw   $t2, 4($t0)
stall   add  $t3, $t1, $t2        lw   $t4, 8($t0)
        sw   $t3, 12($t0)       add  $t3, $t1, $t2
        lw   $t4, 8($t0)          sw   $t3, 12($t0)
stall   add  $t5, $t1, $t4      add  $t5, $t1, $t4
        sw   $t5, 16($t0)         sw   $t5, 16($t0)
```

13 cycles | 11 cycles

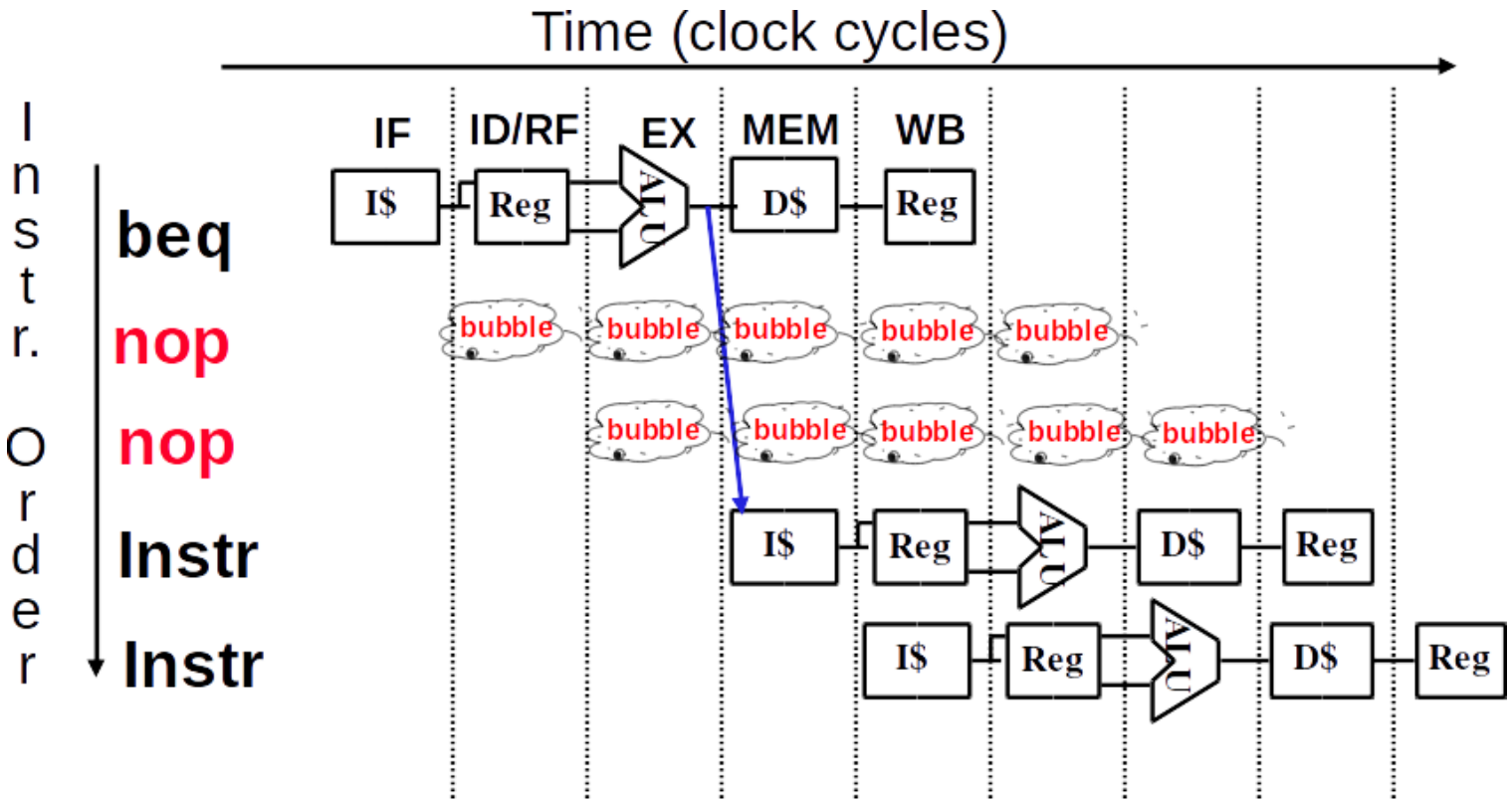
Outline

Control Hazards: Causes and Resolutions

- Control hazards are caused by instructions which change the flow of control.
  - ↳ Branching.
  - ↳ If statements, loops.
- Sometimes called branch hazards.
- Since branch condition (beq, bne) not determined until after EX stage, cannot be *certain* about next instruction to fetch.

Control Hazard Resolution: Wait

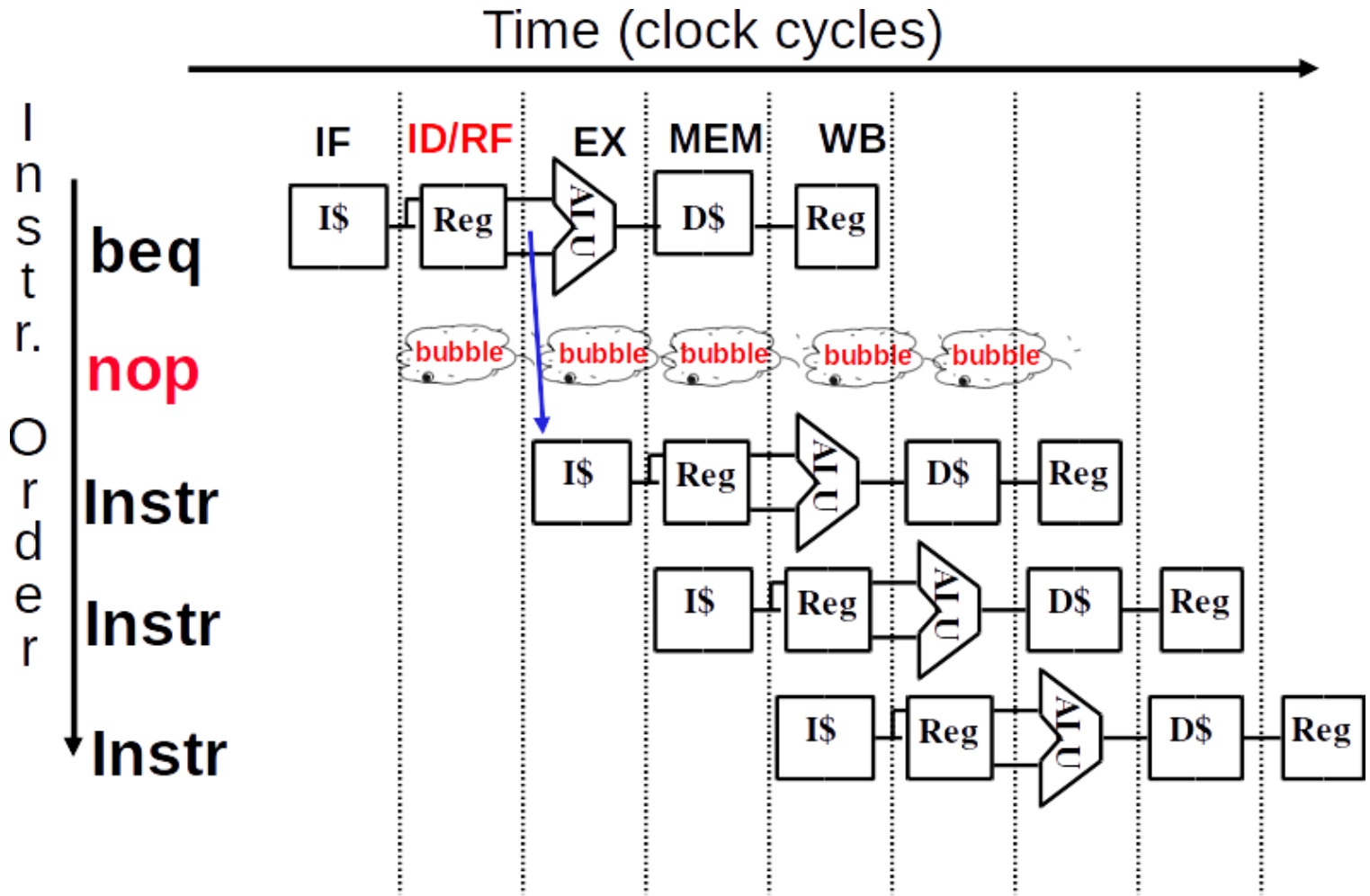
The simplest resolution is to just wait until branch condition is calculated before fetching next instruction.



Control Hazard Resolution: Add a Branch Comparator

Add a special circuit used to calculate branch conditions.

- Now only one stall needed instead of two.
- Similar to load-use hazard we now have a **branch delay slot**.



Delayed Branching

- The **branch delay slot** is the instruction immediately following a branch. Can be a `nop` or a useful instruction.
- In **delayed branching** the instruction in the branch delay slot is **always** executed whther or not the branch condition holds.
  - ↳ Used in conjunction with a special branch comparator.
  - ↳ Filling the branch delay slot (and other code re-organization) is usually handled by compiler/assembler.
  - ↳ Cannot fill slot with an instruction that influences branch condition.
- Jump instructions also have a delay slot.

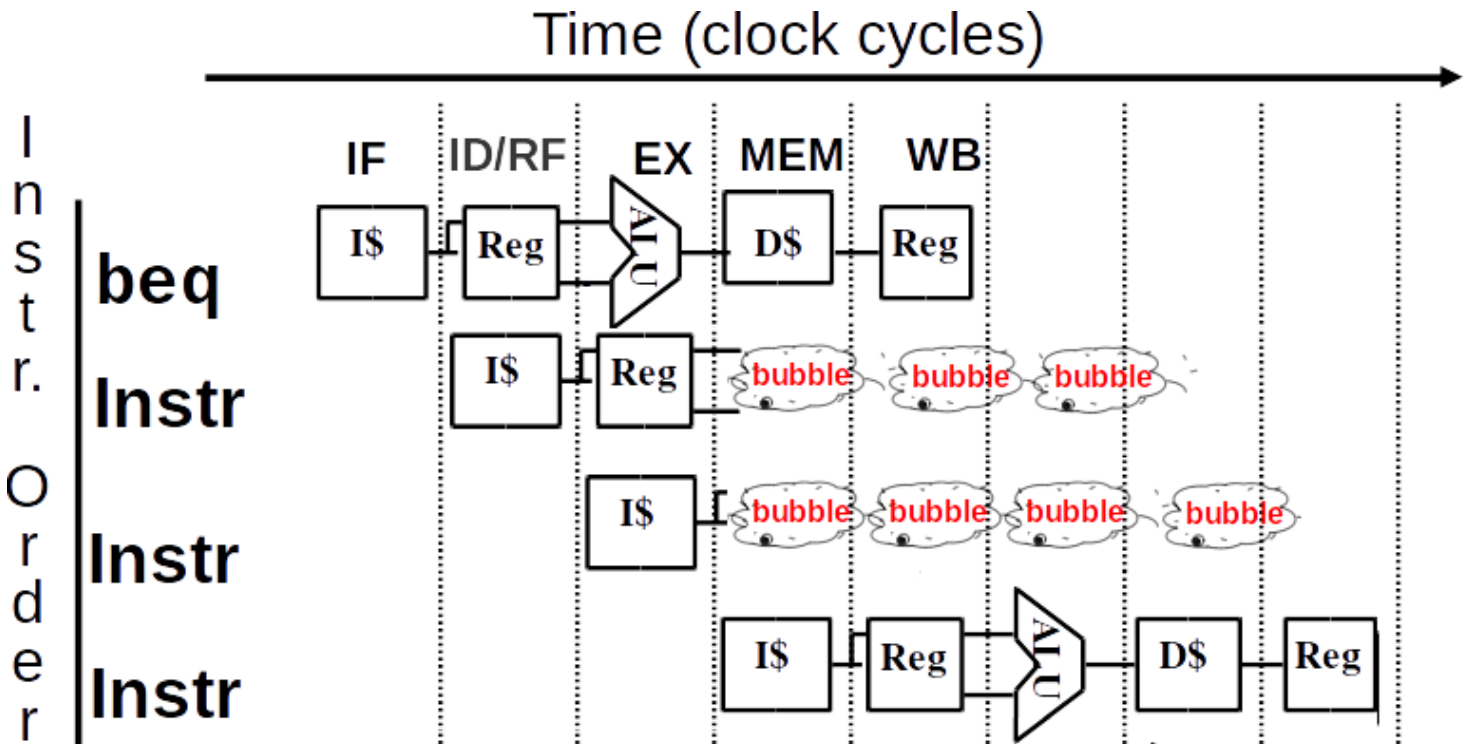
```
   addi $v0, $0, 1
   add $t0, $s0, $s1
   add $t1, $s2, $s3
   beq $t0, $t1, L
     ⋮
L:   ...
```

```
   add $t0, $s0, $s1
   add $t1, $s2, $s3
   beq $t0, $t1, L
   addi $v0, $0, 1
     ⋮
L:   ...
   # addi executed regardless
```

Control Hazard Resolution: Branch Prediction

Hardware predicts whether branch will occur of not.

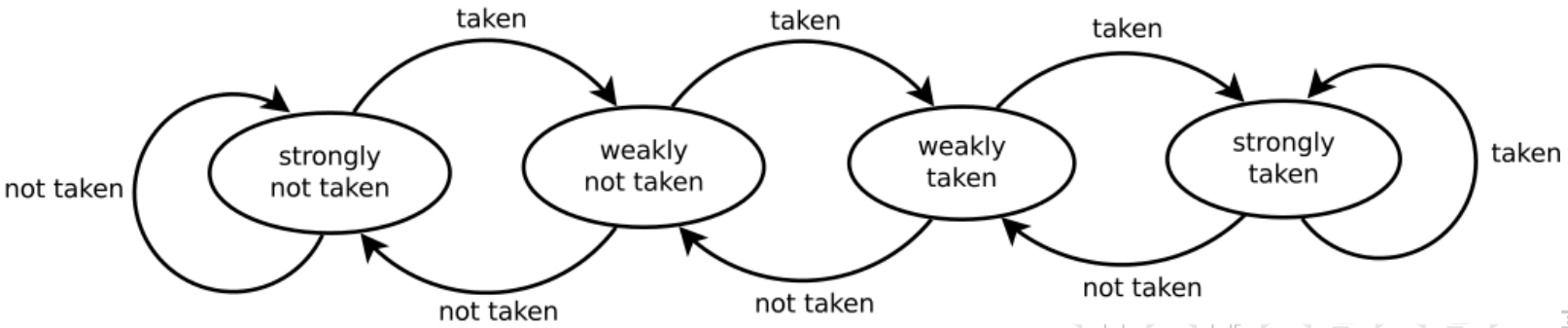
- If the branch condition ends up being opposite of prediction **flush** the pipeline.
- This flush shows a pipeline *without* a special branch comparator in ID stage. Otherwise, only one instruction needs to be flushed.



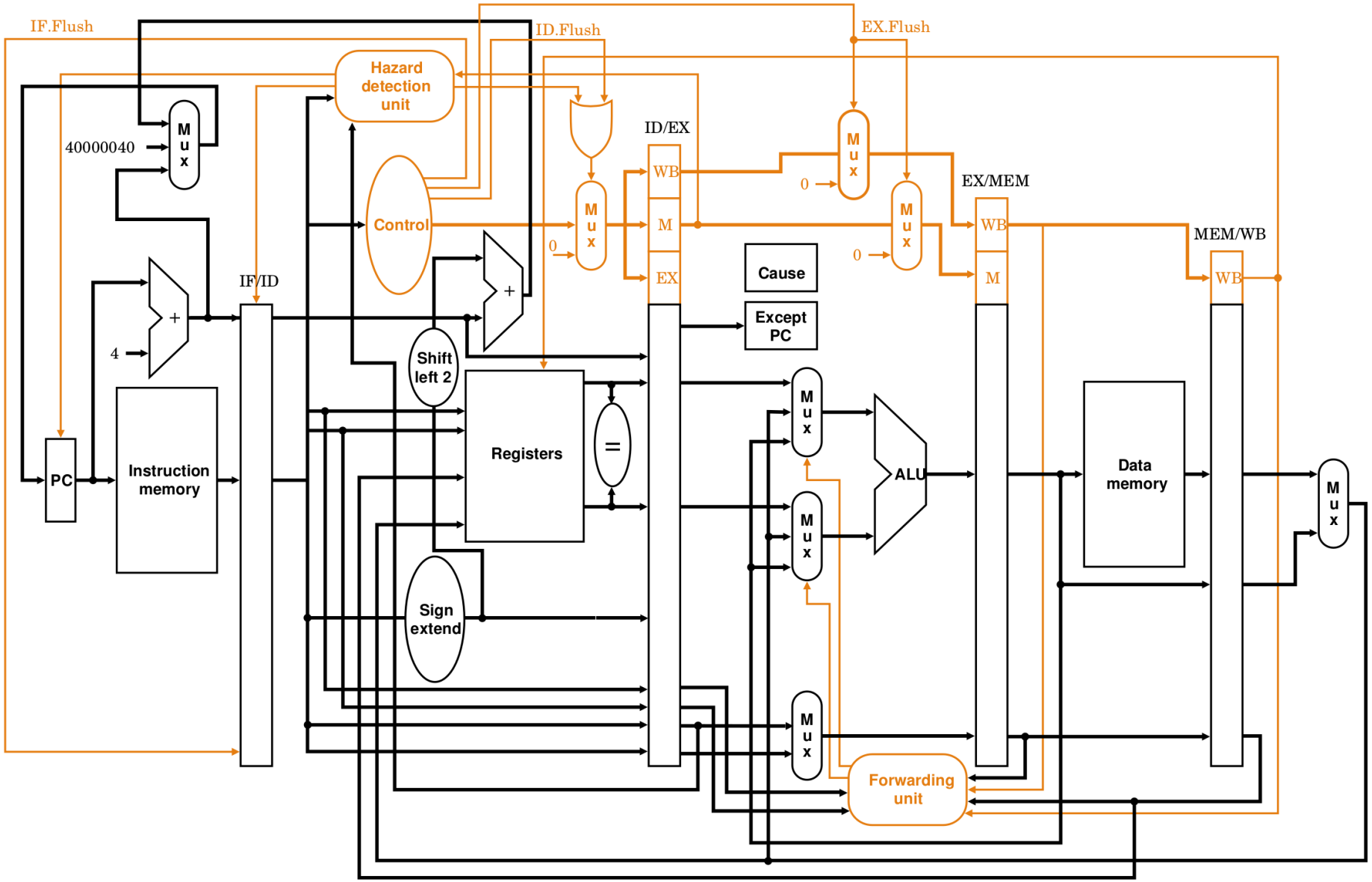
Implementing Branch Prediction

- Branches have exactly two possibilities: taken or not taken.
- In MIPS branches are *statically predicted* to never happen.
- **Dynamic branch prediction** uses run-time information to change prediction between taken or not taken.
  - ↳ Use **branch history** to predict future branches.
  - ↳ Simplest method is to use a *saturated counter*: increment counter if branch actually taken, decrease counter if branch not taken.
  - ↳ Predict based on current count.
  - ↳ More advanced predictors evaluate *patterns* in branch history.
- **Random branch prediction**: statistically 50% correct prediction.

A two-bit saturated counter:



Datapath With Forwarding and Flushing



Hazard Summary

- **Structural hazards** caused by conflicts accessing hardware.
  - ↳ Register access fast enough to happen twice in one clock cycle.
  - ↳ Banked L1 cache for simultaneous instruction and data access.
- **Data hazards** caused by Read After Write (RAW).
  - ↳ ALU-ALU forwarding.
  - ↳ MEM-MEM forwarding (memory copies).
  - ↳ Load-use hazard: stall (load-delay slot) and MEM-ALU forward.
- **Control hazards** caused by branch instructions.
  - ↳ Special branch comparator in ID stage.
  - ↳ Branch delay slot; **delayed branching**.
  - ↳ Branch prediction and **pipeline flush**.
- Compiler handles `nop` insertion to fix hazards.
- Hardware handles fixing hazards with **pipeline interlock**.

CS3350B Computer Organization
Chapter 4: Instruction-Level Parallelism
Hazard Examples

Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Thursday March 14, 2019

Introduction

- In pipelining examples, assume we always start with the "basic" datapath; the one as of the end of Lecture 11.
  - ↳ This datapath implicitly already solves the two structural hazards in memory and register file.
  - ↳ That is, we do not consider structural hazards.

- Each optimization should be explicitly added in the question or in your answer for a possible resolution.
  - ↳ Each type of forwarding (ALU-ALU, MEM-ALU, MEM-MEM).
  - ↳ Filling the load delay slot with something other than `nop`.
  - ↳ Branch comparator in ID stage.
  - ↳ Delayed branching and branch delay slot.

Example 1

```
lw    $t0, 0($s1)
addu  $t0, $t0, $s2
subu  $t4, $t0, $t3
addi  $s1, $s1, -4
add   $t1, $t1, $t2
```

- If any dependencies exist where are they and what type are they?

Example 1

```
lw   $t0, 0($s1)
addu $t0, $t0, $s2
subu $t4, $t0, $t3
addi $s1, $s1, -4
add  $t1, $t1, $t2
```

- If any dependencies exist where are they and what type are they?
  - ↳ Load-use (RAW) between `lw` and `addu`.
  - ↳ WAW between `lw` and `addu`.
  - ↳ RAW between `addu` and `sub`.

Example 1

```
lw    $t0, 0($s1)
addu  $t0, $t0, $s2
subu  $t4, $t0, $t3
addi  $s1, $s1, -4
add   $t1, $t1, $t2
```

- On the basic datapath, how many cycles does it take to execute the code fragment (including stalls)?

Example 1

```
lw   $t0, 0($s1)
addu $t0, $t0, $s2
subu $t4, $t0, $t3
addi $s1, $s1, -4
add  $t1, $t1, $t2
```

- On the basic datapath, how many cycles does it take to execute the code fragment (including stalls)?
  - ↳ 2 `nop` between `lw` and `addu`. MEM of `lw` and IF of `addu` can overlap.
  - ↳ 2 `nop` between `addu` and `sw`. MEM of `addu` and IF of `sw` can overlap.
  - ↳ On 5th cycle `lw` completes and then one cycle per instruction after that.
  - ↳ Including `nop` we get: $5 + 2\ \texttt{nop} + 1 + 2\ \texttt{nop} + 2 + 1 = 13$.

Example 1

```
lw    $t0, 0($s1)
addu  $t0, $t0, $s2
subu  $t4, $t0, $t3
addi  $s1, $s1, -4
add   $t1, $t1, $t2
```

	Clock												
	1	2	3	4	5	6	7	8	9	10	11	12	13
lw	IF	ID	EX	ME	WB								
nop		x	x	x	x	x							
nop			x	x	x	x	x						
addu				IF	ID	EX	ME	WB					
nop					x	x	x	x	x				
nop						x	x	x	x	x			
subu							IF	ID	EX	ME	WB		
addi								IF	ID	EX	ME	WB	
add									IF	ID	EX	ME	WB

Example 1

```
lw   $t0, 0($s1)
addu $t0, $t0, $s2
subu $t4, $t0, $t3
addi $s1, $s1, -4
add  $t1, $t1, $t2
```

- What optimizations can be added to the datapath to reduce the number of cycles? How many cycles are needed to execute the code fragment after optimizations are added?

Example 1

```
lw    $t0, 0($s1)
addu  $t0, $t0, $s2
subu  $t4, $t0, $t3
addi  $s1, $s1, -4
add   $t1, $t1, $t2
```

- What optimizations can be added to the datapath to reduce the number of cycles? How many cycles are needed to execute the code fragment after optimizations are added?
  - ↳ MEM-ALU forwarding for load-use. Reduces `nop` count to 1.
  - ↳ ALU-ALU forwarding removes both `nop` between `addu` and `sub`
  - ↳ Clock cycles: $5 + 1$ nop $+ 4 = 10$.

Example 1

```
lw    $t0, 0($s1)
addu  $t0, $t0, $s2
subu  $t4, $t0, $t3
addi  $s1, $s1, -4
add   $t1, $t1, $t2
```

	Clock									
	1	2	3	4	5	6	7	8	9	10
lw	IF	ID	EX	ME	WB					
nop		x	x	x	x	x				
addu			IF	ID	EX	ME	WB			
subu				IF	ID	EX	ME	WB		
addi					IF	ID	EX	ME	WB	
add						IF	ID	EX	ME	WB

Example 1

```
lw    $t0, 0($s1)
addu  $t0, $t0, $s2
subu  $t4, $t0, $t3
addi  $s1, $s1, -4
add   $t1, $t1, $t2
```

- Can code re-organization along with datapath optimizations be used to further improve the number of clock cycles needed to execute the code? If so, re-order the code and declare any additional optimizations; what is the number of cycles needed to execute the re-ordered code?

Example 1

```
lw    $t0, 0($s1)
addu  $t0, $t0, $s2
subu  $t4, $t0, $t3
addi  $s1, $s1, -4
add   $t1, $t1, $t2
```

- Can code re-organization along with datapath optimizations be used to further improve the number of clock cycles needed to execute the code? If so, re-order the code and declare any additional optimizations; what is the number of cycles needed to execute the re-ordered code?
  - ↳ Yes.
  - ↳ Move `addi` or `add` into **load-delay slot**.
  - ↳ 9, since we remove the nop.

Example 1

```
lw   $t0, 0($s1)
addu $t0, $t0, $s2
subu $t4, $t0, $t3
addi $s1, $s1, -4
add  $t1, $t1, $t2
```

	Clock								
	1	2	3	4	5	6	7	8	9
lw	IF	ID	EX	ME	WB				
addi		IF	ID	EX	ME	WB			
addu			IF	ID	EX	ME	WB		
subu				IF	ID	EX	ME	WB	
add					IF	ID	EX	ME	WB

Example 2

```
sub   $t2, $t1, $t3
and   $t7, $t2, $t5
or    $t8, $t6, $t2
add   $t9, $t2, $t2
sw    $t5, 12($t2)
```

- If any dependencies exist where are they and what type are they?

Example 2

```
sub   $t2, $t1, $t3
and   $t7, $t2, $t5
or    $t8, $t6, $t2
add   $t9, $t2, $t2
sw    $t5, 12($t2)
```

- If any dependencies exist where are they and what type are they?
  - ↳ RAW between `sub` and `and`.
  - ↳ RAW between `sub` and `or`.
  - ↳ RAW between `sub` and `and`.
  - ↳ RAW between `sub` and `sw`.

Example 2

```
sub   $t2, $t1, $t3
and   $t7, $t2, $t5
or    $t8, $t6, $t2
add   $t9, $t2, $t2
sw    $t5, 12($t2)
```

- Consider the basic datapath with ALU-ALU and MEM-ALU forwarding added. In this code fragment where do forwards occur? How many cycles does it take to execute the code fragment?

Example 2

```
sub   $t2, $t1, $t3
and   $t7, $t2, $t5
or    $t8, $t6, $t2
add   $t9, $t2, $t2
sw    $t5, 12($t2)
```

- Consider the basic datapath with ALU-ALU and MEM-ALU forwarding added. In this code fragment where do forwards occur? How many cycles does it take to execute the code fragment?
  - ↳ ALU-ALU from `sub` to `and`.
  - ↳ MEM-ALU from `sub` to `or`.
  - ↳ `sub` to `and` RAW solved by register file design.
  - ↳ $5 + 1 + 1 + 1 + 1 = 9$

Example 2

```
sub   $t2, $t1, $t3
and   $t7, $t2, $t5
or    $t8, $t6, $t2
add   $t9, $t2, $t2
sw    $t5, 12($t2)
```

	Clock								
	1	2	3	4	5	6	7	8	9
sub	IF	ID	EX	ME	WB				
and		IF	ID	EX	ME	WB			
or			IF	ID	EX	ME	WB		
and				IF	ID	EX	ME	WB	
sw					IF	ID	EX	ME	WB

Example 3

```
for:  beq   $t6, $t7, end
      add   $t0, $t0, $t1
      addi  $t6, $t6, 1
      j for
end:  sub   $t1, $t6, $0
```

- Assuming the basic data path how many cycles does it take to execute two loops within the code fragment (therefore, excluding the `sub`)?

Example 3

```
for: beq  $t6, $t7, end
     add  $t0, $t0, $t1
     addi $t6, $t6, 1
     j for
end: sub  $t1, $t6, $0
```

- Assuming the basic data path how many cycles does it take to execute two loops within the code fragment (therefore, excluding the `sub`)?
  - ↳ Careful! Since a loop, RAW dependency between `andi` and `beq`.
  - ↳ Two `nop` follows `beq` for control hazard.
  - ↳ One `nop` follows `j` for control hazard.
  - ↳ First loop: $5 + 2$ `nop` $+ 3 + 1$ `nop`.
  - ↳ In the second loop `beq` overlaps with previous instructions.
  - ↳ Second loop: $1 + 2$ `nop` $+ 3 + 1$ `nop`.
  - ↳ Total: 18.

Example 3

```
for: beq  $t6, $t7, end
     add  $t0, $t0, $t1
     addi $t6, $t6, 1
     j for
end: sub  $t1, $t6, $0
```

	Clock																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
beq	IF	ID	EX	ME	WB													
nop		-	-	-	-	-												
nop			-	-	-	-	-											
add				IF	ID	EX	ME	WB										
addi					IF	ID	EX	ME	WB									
j						IF	ID	EX	ME	WB								
nop							-	-	-	-	-							
beq								IF	ID	EX	ME	WB						
nop									-	-	-	-	-					
nop										-	-	-	-	-				
add											IF	ID	EX	ME	WB			
addi												IF	ID	EX	ME	WB		
j													IF	ID	EX	ME	WB	
nop														-	-	-	-	-

Example 3

```
for:  beq  $t6, $t7, end
      add  $t0, $t0, $t1
      addi $t6, $t6, 1
      j for
end:  sub  $t1, $t6, $0
```

- Using any datapath optimizations and code re-ordering, minimize the clock cycles required to execute the loop two times. Name the optimizations used. How many cycles does it take to execute this optimized version?

Example 3

```
for: beq  $t6, $t7, end
     add  $t0, $t0, $t1
     addi $t6, $t6, 1
     j for
end: sub  $t1, $t6, $0
```

- Using any datapath optimizations and code re-ordering, minimize the clock cycles required to execute the loop two times. Name the optimizations used. How many cycles does it take to execute this optimized version?
  - ↳ Special branch comparator in ID stage.
  - ↳ Careful! Cannot fill branch delay slot.
  - ↳ Using `add` would change code meaning.
  - ↳ Value of $t6 used again after loop so cannot use `addi`.
  - ↳ Cannot use jump for obvious control-flow reasons.
  - ↳ Total savings: 1 `nop` per branch $\Rightarrow$ 16 cycles now.
  - ↳ (If using branch prediction, all `nops` are removed).

Example 3

```
for: beq  $t6, $t7, end
     add  $t0, $t0, $t1
     addi $t6, $t6, 1
     j for
end: sub  $t1, $t6, $0
```

	Clock															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
beq	IF	ID	EX	ME	WB											
nop		-	-	-	-	-										
add			IF	ID	EX	ME	WB									
addi				IF	ID	EX	ME	WB								
j					IF	ID	EX	ME	WB							
nop						-	-	-	-	-						
beq							IF	ID	EX	ME	WB					
nop								-	-	-	-	-				
add									IF	ID	EX	ME	WB			
addi										IF	ID	EX	ME	WB		
j											IF	ID	EX	ME	WB	
nop												-	-	-	-	-

CS3350B Computer Organization
Chapter 4: Instruction-Level Parallelism
Part 3: Beyond Pipelining

Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Tuesday March 19, 2019

Outline

Instruction-Level Parallelism (ILP)

Instruction-level parallelism involves executing multiple instructions at the same time.

- Instructions may simply overlap (pipelining) or,
- Instructions may be executed completely in parallel (**superscalar**).

There are many techniques which are used to provide ILP or to support ILP in achieving greater speed-up.

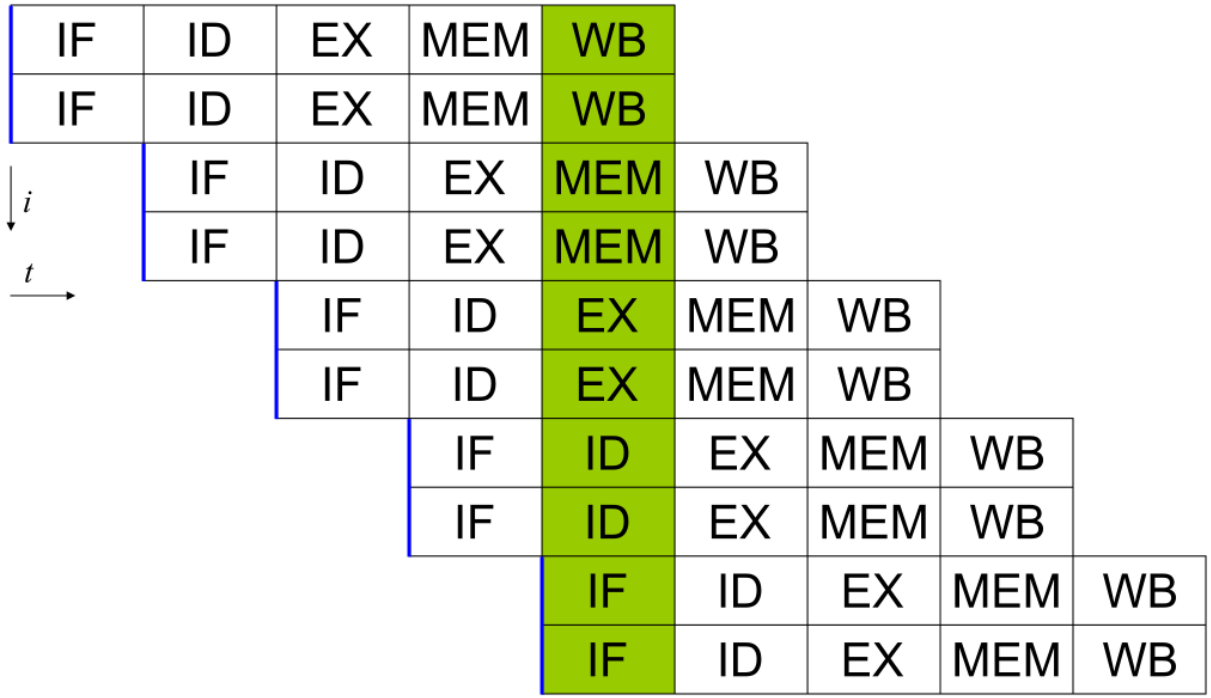
- Pipelining.
- Branch prediction.
- **Superscalar** execution.
- **Very Long Instruction Word** (VLIW).
- **Register renaming**.
- **Loop unrolling**.

Multiple Issue Processors

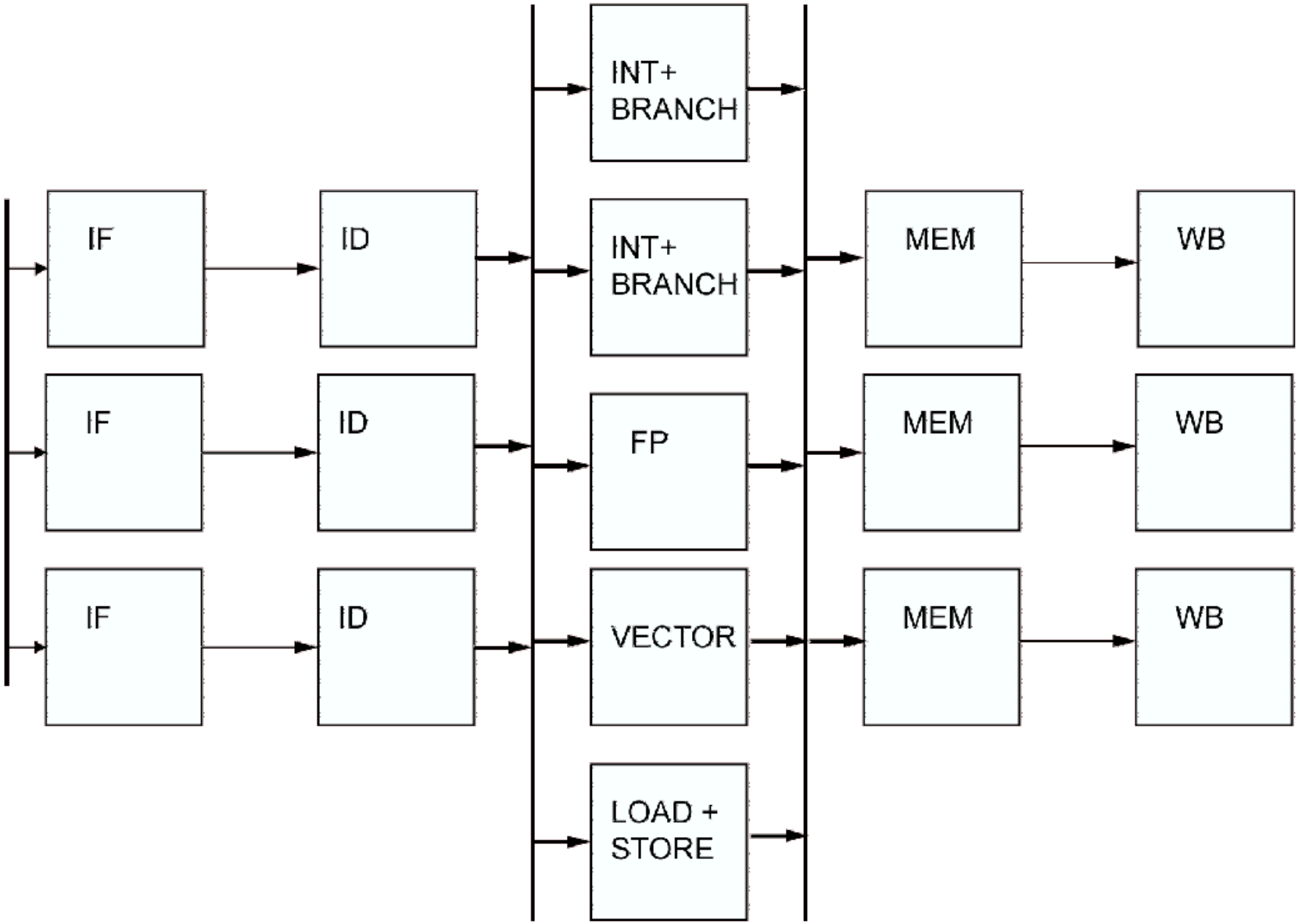
- A **multiple issue processor** issues (executes) multiple instructions within a clock cycle. (Aims for CPI < 1)
  - ↳ VLIW Processors.
  - ↳ Static Superscalar Processors (essentially same as VLIW).
  - ↳ Dynamic Superscalar Processors.
- By their nature, all multiple issue processors have multiple execution units (ALUs) in their datapath.
- Depending on the type of multiple issue processor, other circuitry may also be duplicated or augmented.
- *Note:* multiple issue processors are not necessarily pipelined (these concepts are separate) but in reality pipelining is so good and multiple issue came after so all multiple issue processors are also pipelined.

Static Superscalar Processors

- The name static implies the **code scheduling** is done by compiler.
- Basically side-by-side datapaths simultaneously executing instructions.
- Compiler handles dependencies and hazards and scheduling code so that instructions on different datapaths don't conflict.
- Near identical to VLIW so we'll skip the details.



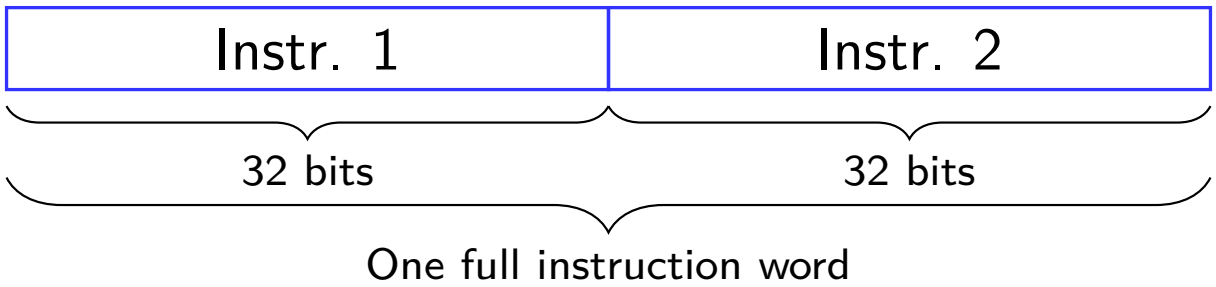
Static Superscalar Pipeline



Outline

VLIW Processors (1/2)

- VLIW processors have *very long instruction words*.
- Essentially, multiple instructions are encoded within a single (long) instruction memory word called an **issue packet**.
- The instructions which can be packed together are limited. Usually only one `lw`/`sw`, only one branch, rest arithmetic.
- In this case instructions word size $\neq$ data memory word size.
- Simplest scheme: just concatenate multiple instructions together.
- Ex: Two 32-bit instrs. together in a single 64-bit instruction word.

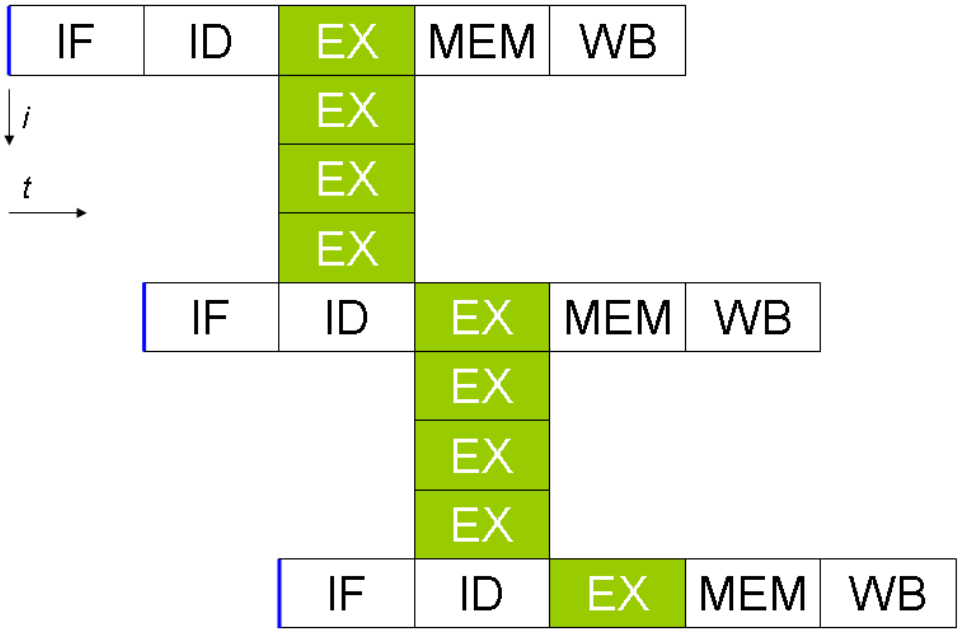


VLIW Processors (2/2)

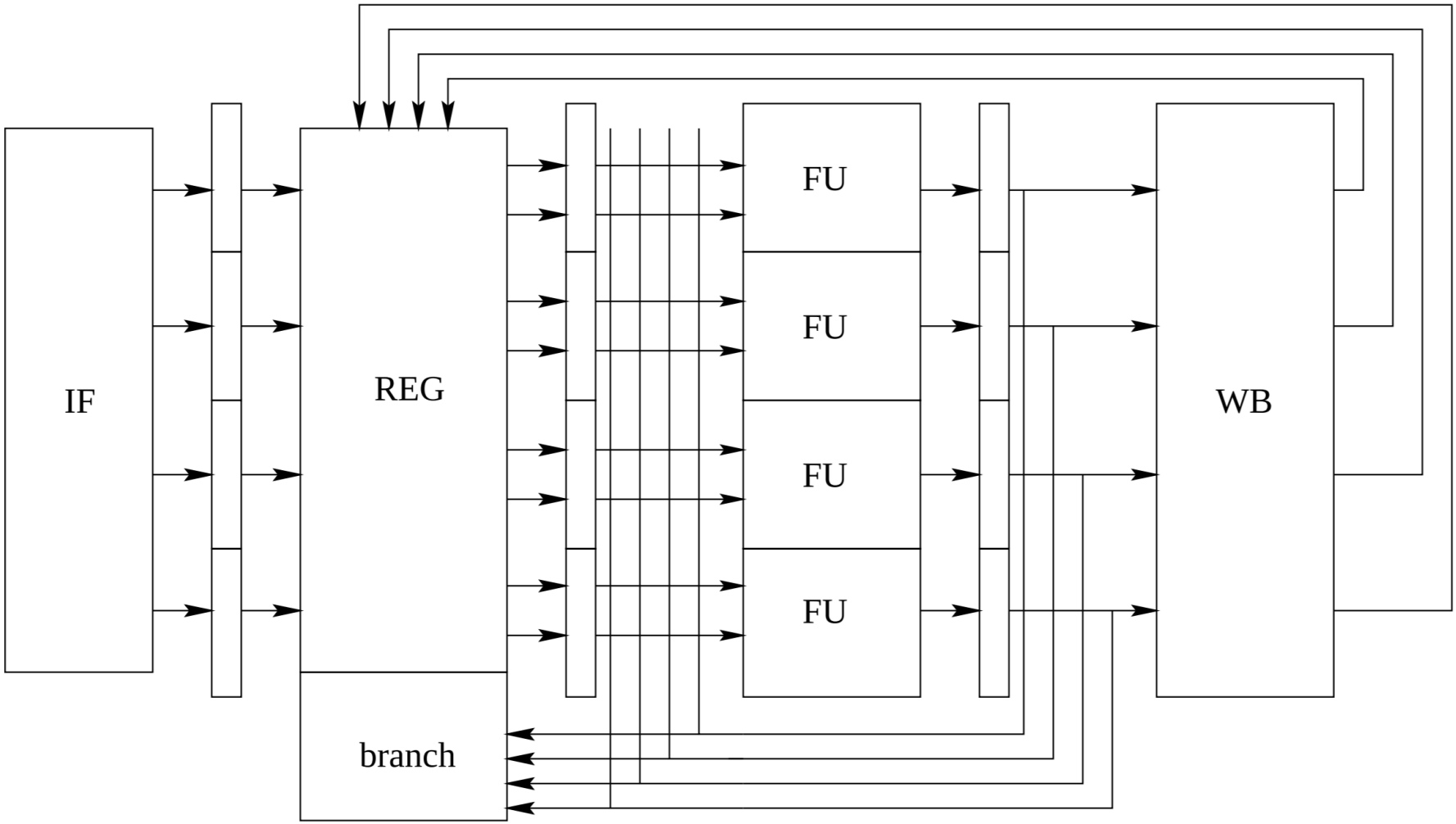
In a VLIW pipeline:

- One IF unit fetches a single long word encoding multiple instrs.
- One ID ⇒ register file must handle multiple simultaneous reads.
- In EX stage, each instr. is **issued** to a different execution unit (ALU).

- Only one data memory to read/write from!
  There is a limitation on which kinds of instructions can be executed simultaneously.
- In the WB stage the register file must handle multiple writes (to different registers, obviously).



4-Stage VLIW (without MEM stage for simplicity)

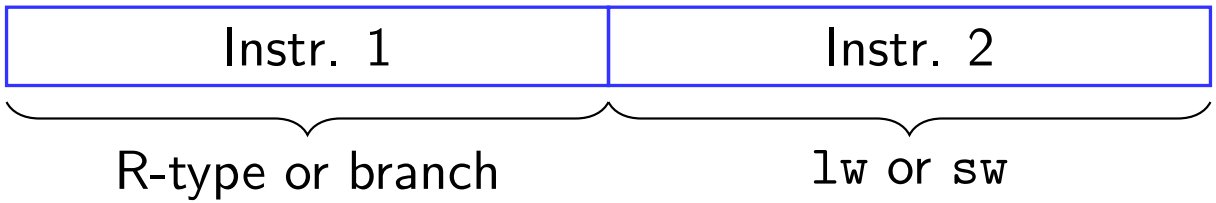


M. Oskin et al. Exploiting ILP in page-based intelligent memory. In *ACM/IEEE International Symposium on MICRO-32*, Proceedings, pages 208-218, 1999.

A VLIW Example (1/3)

Consider a 2-issue extension of MIPS.

- The first slot of the issue packet must be an R-type instruction or a branch.
- The second slot of the issue packet must be `lw` or `sw`.
- If compiler cannot find an instruction, insert `nop`.
  - ↳ Much like load-delay slot or branch-delay slot.



A VLIW Example (2/3)

```
loop:  lw $t0, 0($s1)       # $t0=array element
       addu $t0, $t0, $s2   # add scalar in $s2
       sw $t0, 0($s1)       # store result
       addi $s1, $s1, -4    # decrement pointer
       bne $s1, $0, loop    # branch if $s1 != 0
```

```
for (int i=n; i>0; --i) {
    A[i] += s2;
}
```

Need to schedule code for 2-issue.

- Instructions in same issue packet must be independent.
- Assume perfect branch prediction.
- Load-use and RAW dependencies still need to be handled.
  - ↳ But, assume all possible datapath optimizations (forwarding).

A VLIW Example (3/3)

```
loop:  lw $t0, 0($s1)       # $t0=array element
       addu $t0, $t0, $s2   # add scalar in $s2
       sw $t0, 0($s1)       # store result
       addi $s1, $s1, -4    # decrement pointer
       bne $s1, $0, loop    # branch if $s1 != 0
```

	ALU or branch	Data transfer	CC
loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1, -4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $0, loop	sw $t0, 4($s1)	4

- CPI is 4 cycles / 5 instructions = 0.8.
- nops don't count towards performance.
- Sometimes when scheduling code you need to adjust offsets.

Outline

Loop Unrolling

- Compilers use **loop unrolling** to *expose more parallelism*.
- Essentially, body of loop is replaced with multiple copies of itself (still in a loop).
- Avoids unnecessary branch delays, can more effectively schedule code and fill load-use slots.
- Ex: 4-time loop unrolling. Notice `i += 4` in unrolled code.

```
int i = 0;
for (i; i < n; i++) {
    A[i] += 10;
}
```

```
int i = 0;
for (i; i < n; i += 4) {
    A[i] += 10;
    A[i+1] += 10;
    A[i+2] += 10;
    A[i+3] += 10;
}
```

Unrolling MIPS

```
loop:   lw $t0, 0($s1)
        addu $t0, $t0, $s2
        sw $t0, 0($s1)
        addi $s1, $s1, -4
        bne $s1, $0, loop
```

```
for (int i=n; i>0; --i) {
    A[i] += s2;
}
```

```
loop: lw   $t0,0($s1)  # $t0=array element
      lw   $t1,-4($s1) # $t1=array element
      lw   $t2,-8($s1) # $t2=array element
      lw   $t3,-12($s1)# $t3=array element
      addu $t0,$t0,$s2 # add scalar in $s2
      addu $t1,$t1,$s2 # add scalar in $s2
      addu $t2,$t2,$s2 # add scalar in $s2
      addu $t3,$t3,$s2 # add scalar in $s2
      sw   $t0,0($s1)  # store result
      sw   $t1,-4($s1) # store result
      sw   $t2,-8($s1) # store result
      sw   $t3,-12($s1)# store result
      addi $s1,$s1,-16 # decrement pointer
      bne  $s1,$0,loop   # branch if $s1 != 0
```

- Notice, loop body is not exactly copied 4 times.
- Static register renaming: $t0 becomes $t1, $t2, $t3 for successive loops.
- Can now easily reschedule code and fill load-delay slots.
- Much fewer branch instr. and branch delay slots.

Combining Loop Unrolling and VLIW

Same 2-issue extension of MIPS. First slot is ALU, second slot is data transfer. Datapath has all forwarding and possible optimizations.

Remember:

- Need one instruction between load and use.
- Insert `nop` if no instruction possible

	ALU or branch	Data transfer	CC
`loop:`	`addi $s1,$s1,-16`	`lw $t0,0($s1)`	1
	`nop`	`lw $t1,12($s1) #-4`	2
	`addu $t0,$t0,$s2`	`lw $t2,8($s1) #-8`	3
	`addu $t1,$t1,$s2`	`lw $t3,4($s1) #-12`	4
	`addu $t2,$t2,$s2`	`sw $t0,16($s1) #0`	5
	`addu $t3,$t3,$s2`	`sw $t1,12($s1) #-4`	6
	`nop`	`sw $t2,8($s1) #-8`	7
	`bne $s1,$0,loop`	`sw $t3,4($s1) #-12`	8

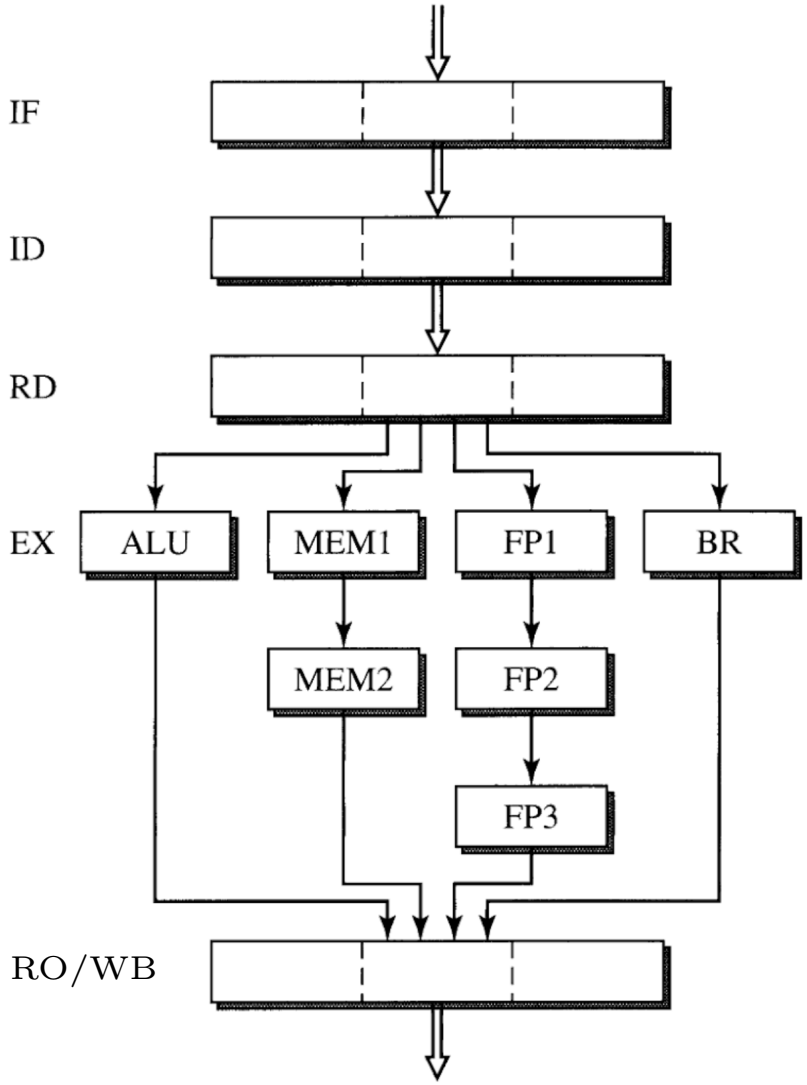
Outline

Dynamic Superscalar Processors

- Dynamic in the name implies that the hardware handles code scheduling.
- Because of the dynamic nature **out-of-order execution** can occur.
  - ↳ Instructions are actually executed in a different order than they are fetched.
- This scheme allows for different types of execution paths which all take a different amount of time:
  - ↳ Ex: Normal ALU, Floating point unit, Memory load/store path.
- This scheme is particularly good at overcoming stalls due to cache misses and other dependencies.
- However, hardware becomes *much* more complex than static schemes.

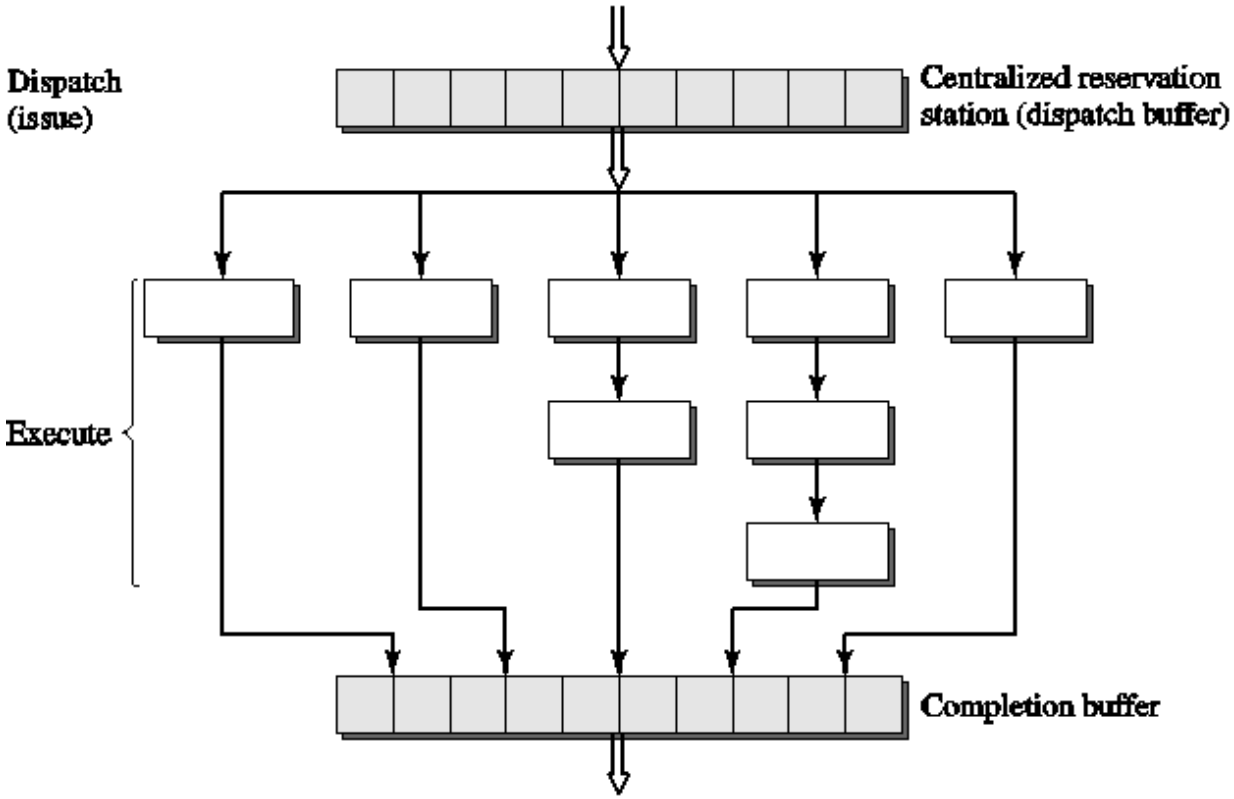
A Dynamic Superscalar Pipeline (1/2)

- Fetch one instr. per cycle as normal.
- "Pre-decode" instr. and add to **instruction buffer** in **RD** (dispatch) stage.
  - ↳ Out-of-order execution ⇒ must wait for updated values of operands.
- Once instr. operands are ready, **dispatcher** issues instr. to one of many execution units. This is where out-of-order execution is introduced.
- **Dispatch**: adding instr. to buffer.
- **Issue**: sending instr. to execution unit.



A Dynamic Superscalar Pipeline (2/2)

- **Before** WB stage, a **reorder buffer** or **completion stage** makes sure instructions are in-order before writing results back (a.k.a committing).
  - ↳ Out-of-order execution means WAR and WAW dependencies matter.
- Finally, instructions are **retired**.



A **centralized reservation station** is both a dispatcher and issuer.

Pros and Cons of Dynamic Scheduling

- Compiler is only so good at scheduling code. Data hazards are hard to resolve.
  - ↳ Compiler only sees pointers but hardware sees actual memory addresses.
- Dynamic scheduling overcomes unpredictable stalls (cache misses) but requires complex circuitry.
- Dynamic scheduling more aggressively overlaps instructions since operand values are read and then queued in reservation stations.
- Instructions are fetched in-order, executed out-of-order, and then committed/retired in-order and sequentially.
- *Flynn's bottleneck*: can only retire as many instr. per cycle as are fetched.
  - ↳ Superscalar machines usually augmented with fetching *multiple* instructions at once.

Comparing Multiple Issue Processors (1/2)

VLIW

- Static scheduling.
- In-order execution.
- Single IF unit but many EX units.
- Instructions packed together in issue packet.

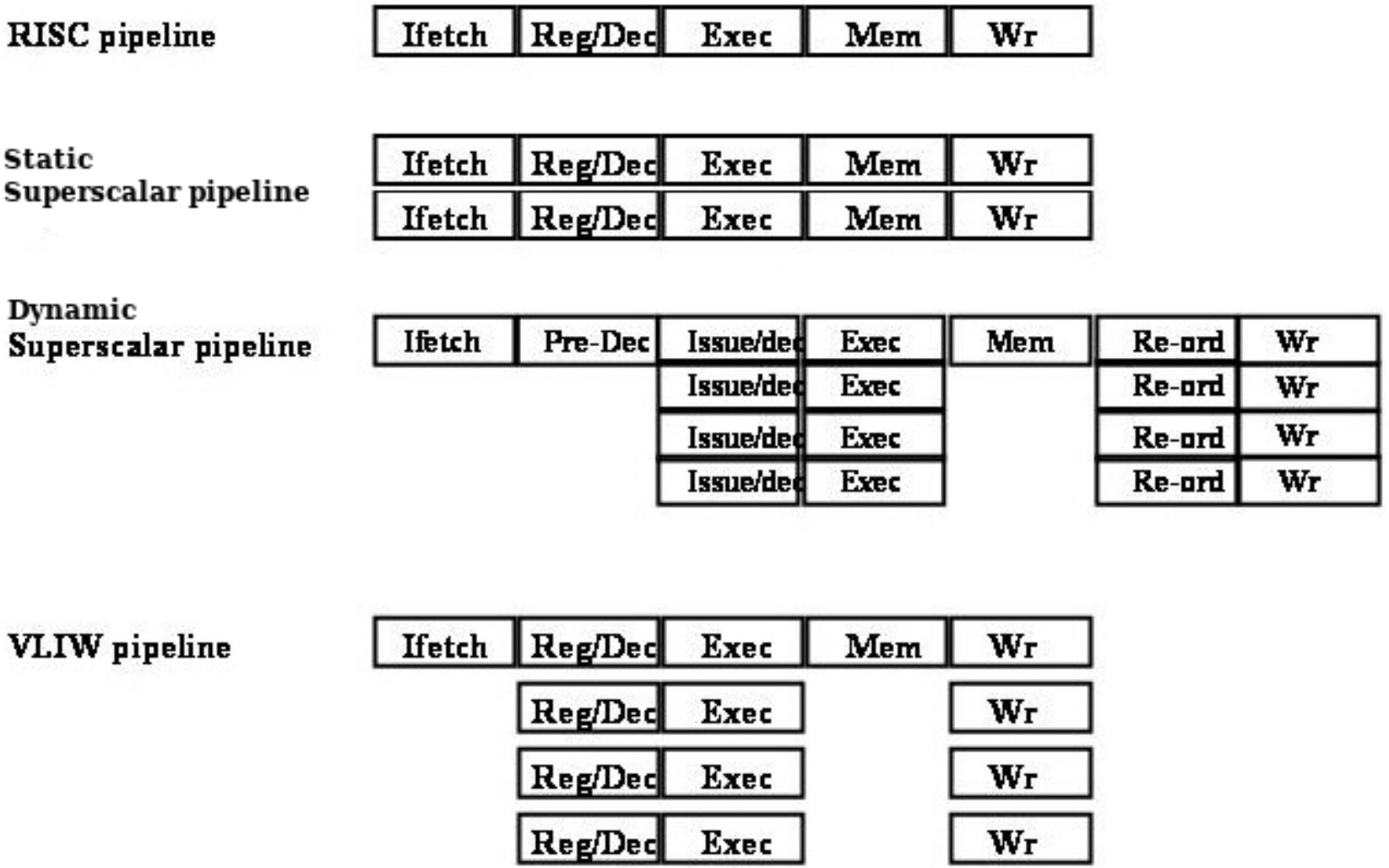
Static SS

- Static scheduling.
- In-order execution.
- Many IF units (or one IF fetching multiple instr.) and many EX units.
- Compiler explicitly schedules each datapath.

Dynamic SS

- Dynamic scheduling.
- Out-of-order exec.
- Single IF unit but many EX units.
- IF unit might fetch multiple instr. per cycle.

Comparing Multiple Issue Pipelines (2/2)



Tapani Ahonen, University of Tampere

Outline

Writer After X Dependencies

Out-of-order execution causes hazards beyond RAW dependencies.

- WAR dependency – write a value after it is read by a previous instruction.
- Ex: Cannot write to $t1 in until `addi` has read its value of $t1.

```
addi   $t0, $t1, 2
add    $t1, $t3, $0
```

- WAW dependency – write a value after its destination has been written to by a previous instruction. Needed to maintain consistent values for future instructions.
- Ex: If `add` executed before `addi` then value of $t1 incorrect.

```
addi   $t1, $t4, 12
add    $t1, $t3, $0
```

Register Renaming

- **Register renaming** is both a static and dynamic technique used to help superscalar pipelines and can fix WAR and WAW dependencies.
- Essentially, code is modified so that every destination is replaced with a unique "logical" destination (sometimes called value name).
  - ↳ Reservation stations provide a hardware buffer for storing logical destinations.
  - ↳ For dynamic renaming, hardware maintains a mapping from register names to logical destinations to modify operands for incoming instructions.

Ex:

`add $t6, $t0, $t2`	RAW: $t6 in `add` and `xor`.
`sub $t4, $t2, $t0`	WAR: $t0 in `sub` and `xor`.
`xor $t0, $t6, $t2`	WAR: $t2 in `sub` and `and`.
`and $t2, $t2, $t6`	WAR: $t2 in `sub` and `and`.

Register Renaming Example

Instr.	$t0	$t2	$t4	$t6	Renamed Instr.
Initially	V0	V1	V2	V3	—
`add $t6, $t0, $t2`				V4	`add V4, V0, V1`
`sub $t4, $t2, $t0`			V5		`sub V5, V1, V0`
`xor $t0, $t6, $t2`	V6				`xor V6, V4, V1`
`and $t2, $t2, $t6`					`add ??, V1, ??`

Register Renaming Example

Instr.	$t0	$t2	$t4	$t6	Renamed Instr.
Initially	V0	V1	V2	V3	—
`add $t6, $t0, $t2`				V4	`add V4, V0, V1`
`sub $t4, $t2, $t0`			V5		`sub V5, V1, V0`
`xor $t0, $t6, $t2`	V6				`xor V6, V4, V1`
`and $t2, $t2, $t6`					`add ??, V1, ??`

Instr.	$t0	$t2	$t4	$t6	Renamed Instr.
Initially	V0	V1	V2	V3	—
`add $t6, $t0, $t2`				V4	`add V4, V0, V1`
`sub $t4, $t2, $t0`			V5		`sub V5, V1, V0`
`xor $t0, $t6, $t2`	V6				`xor V6, V4, V1`
`and $t2, $t2, $t6`		V7			`add V7, V1, V4`

Summary

- Multiple issue processors execute multiple instructions simultaneously for CPI < 1.
- VLIW uses statically-scheduled issue packets.
- Loop unrolling exposes more parallelism *and* removes some branching overhead.
- Dynamic superscalar processors combat against unexpected stalls (cache misses) by allowing for out-of-order execution.
- Register renaming fixes WAR and WAW dependencies.
- RAW dependencies are the only true dependency and still must be accounted for in scheduling.