TU WIEN Informatics

Embedded Computing Systems

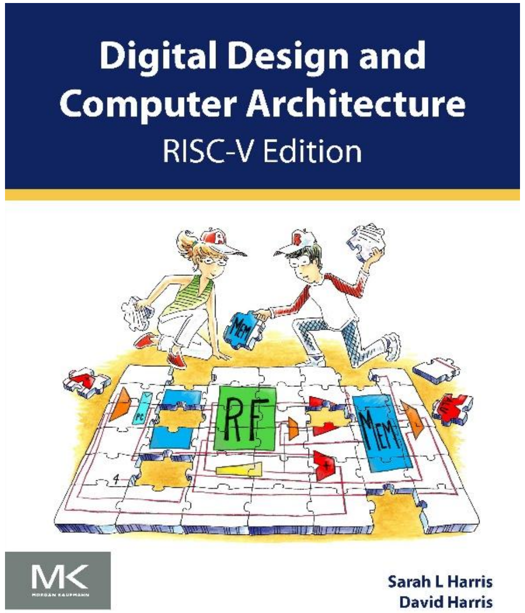
Computer Systems

Advanced Processor Pipelines 2

Daniel Mueller-Gritschneder

08.04.2025

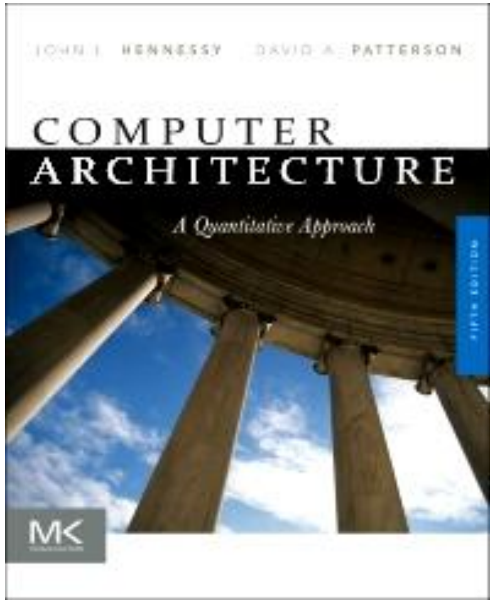
Sources



This book covers the basics of how to design a simple in-order scalar processor pipeline in detail in hardware.

- Literature: „Digital Design and Computer Architecture: RISC-V Edition", by Sarah L. Harris and David Harris
  - https://shop.elsevier.com/books/digital-design-and-computer-architecture-risc-v-edition/harris/978-0-12-820064-3
  - https://pages.hmc.edu/harris/ddca/ddcarv.html (Includes resources for students!)
  - They also provide slideshows – the basis for ours! You can investigate extended version at their website.
- Available at TU's library: https://catalogplus.tuwien.at/permalink/f/qknpf/UTW_alma21139903990003336

Sources



So-called application processors have many additional features:
Branch prediction, Out of order execute, Scoreboard, Superpipelining, Multi-issue, Superscalar, VLIW, Multi-threading, …

Disclaimer: The book provides advanced concepts from real complex processor designs. We only study the concepts at a high level. For simplicity, the used pipeline models in this lecture are reduced strongly in complexity.

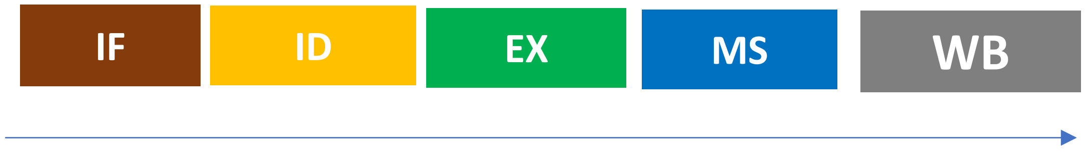
But: We will have a look at some current RISC-V processor designs

Literature: „**Computer Architecture A Quantitative Approach”** 5th Edition - September 16, 2011
Authors: John L. Hennessy, David A. Patterson eBook ISBN: 9780123838735

- https://shop.elsevier.com/books/computer-architecture/hennessy/978-0-12-383872-8

- Available at TU’s library: https://catalogplus.tuwien.at/permalink/f/8agg25/TN_cdi_askewsholts_vlebooks_9780123838735

RECAP: Five-Stage In-Order Scalar Pipeline



IF ID EX MS WB

- Five Stage
- In-order pipeline
- Scalar pipeline

- Each stage takes one cycle to complete

➢Single access cycle to instruction and data memory: Works for small and slow micro-controller-type processors with on-chip embedded SRAM memories

➢Single cycle operations, works for simple instructions (ADD, Compare,…)

`SLLI a2,a1,2`	IF	ID	EX	MS	WB		
`ADD a2,a0,a2`		IF	ID	EX	MS	WB	
`LW a0,0(a2)`			IF	ID	EX	MS	WB
`ADD a0,a0,a1`				IF	ID	stall	EX
`RET`					IF	stall	ID

- Scalar processor: Can execute at maximum 1 instruction per cycle (IPC <=1)

Content

- Multi-cycle Functional Units (FUs)
- Load and Store Optimizations
- Instruction Dependencies (RAW, WAW, WAR)
- Dynamic Scheduling with Scoreboard (Out of Order – OoO)
- Register Renaming
- Superscalar

Optional, not relevant for exam

- A look at a real RISC-V processor: CVA6
- Pipeline Support for Precise Traps

Multi-Cycle Operations

Integer Multiplication Instructions

- Signed-signed Multiplication
  - Multiplying two 32bit values can result in a value of up to 64 bit
  - `MUL a3,a1,a2`
    - Behavior: a3 ← a1*a2 // only the lower 32bit
  - `MULH a4,a1,a2`
    - Behavior: a4 ← a1*a2 // only the higher 32bit
  - Example:
    - `MULH a4,a1,a2`
    - `MUL  a3,a1,a2`
      Behavior: [a4 a3] = a1*a2 // full 64 bit

- Unsigned-unsigned multiplication `MULHU`
- Signed-Unsigned multiplication `MULHSU`

Integer Division Instructions

- Signed-signed Division
  - `DIV a3,a1,a2`
    - Behavior: a3 ← a1 / a2
  - `REM a4,a1,a2`
    - Behavior: a4 ← a1 modulo a2 // remainder
- Unsigned-unsigned division `DIVU, REMU`

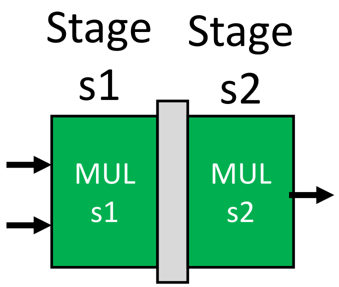
Pipelined Functional Units (FUs)

- Complex computations require deep circuit logic
- Critical path in deep logic limits the design’s frequency
- Similar to processor design, break FU into stages and integrate registers to build a pipeline

➢ **Latency** (in cycles) equals to number of pipeline stages

➢ **Initialization Interval**: Delay (in cycles) between start of two computations

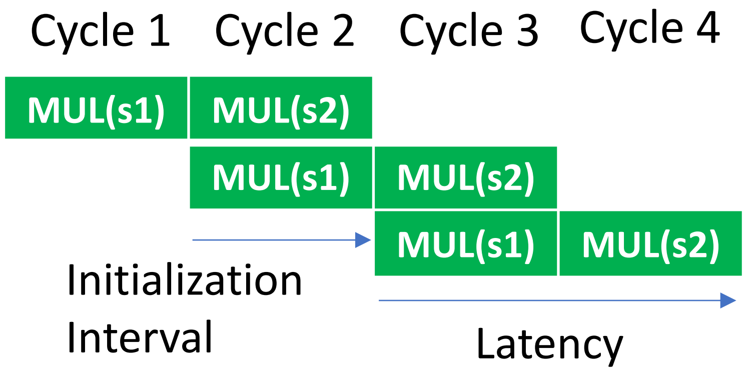
- Example: 2-stage Multiplier



Latency = 2 Cycles

Initialization Interval = 1 Cycle

```
MUL a0,a0,t0
MUL a1,a1,t1
MUL a2,a2,t2
```



Cycle 1	Cycle 2	Cycle 3	Cycle 4
MUL(s1)	MUL(s2)		
	MUL(s1)	MUL(s2)	
		MUL(s1)	MUL(s2)

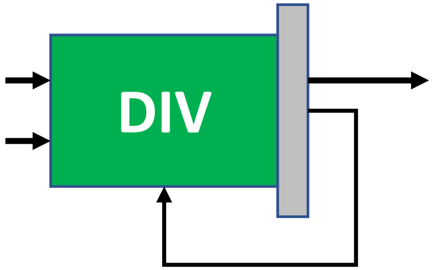
Serial Functional Units (FUs)

- Often complex operations such as divisions can be computed by iterative algorithms
- The number of iterations (required clock cycles) often depends on the input values
- These iterations can be implemented on a serial FU, which is busy as long as it computes

➢ **Latency** equals to number of cycles required for computation

➢ **Initialization Interval** equals to number of cycles required for computation

- Example: Serial Divider

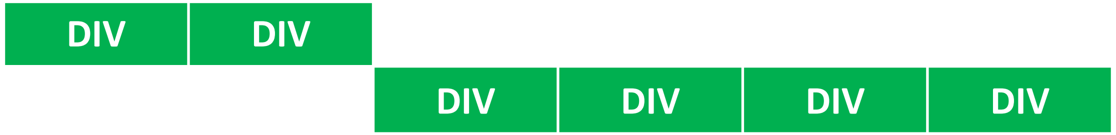


1-64 clock cycles

Latency = 1-64 Cycles

Initialization Interval = Latency

	Latency
`DIV a0,a0,t0`	2
`DIV a1,a1,t1`	4



Example: RISC-V CVA6 Processor

“Multiplier

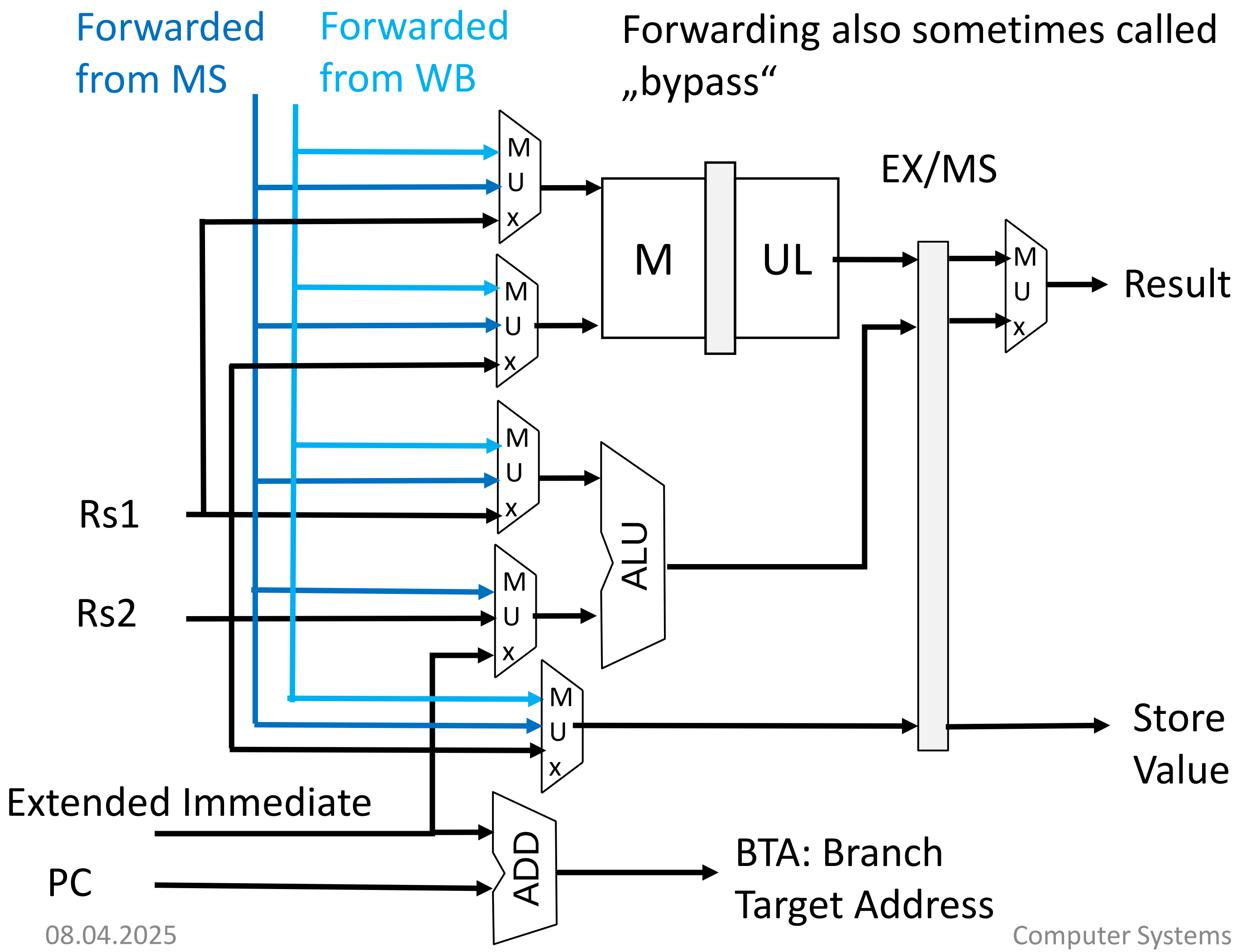
The multiplier contains a division and multiplication unit. Multiplication is performed in two cycles and is fully pipelined (re-timing needed). The division is a simple serial divider which needs 64 cycles in the worst case.”*

*https://docs.openhwgroup.org/projects/cva6-user-manual/03_cva6_design/ex_stage.html

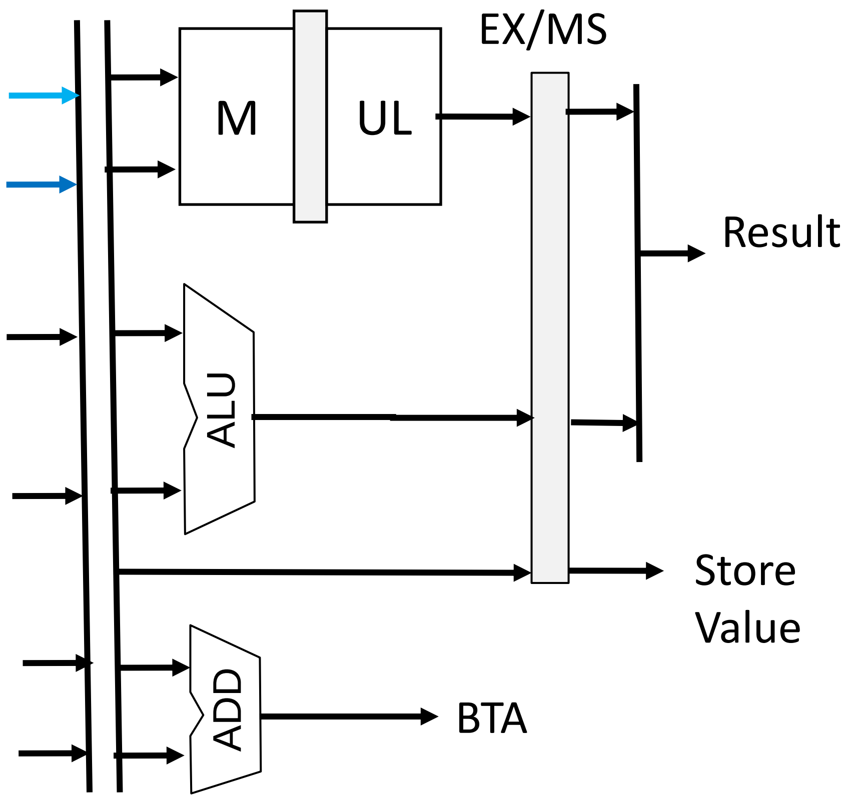
Integration of Multi-cycle Functional Units

- Multi-cycle Functional Units are integrated into the EX stage
- Example only for Multiplier

Forwarding also sometimes called „bypass“

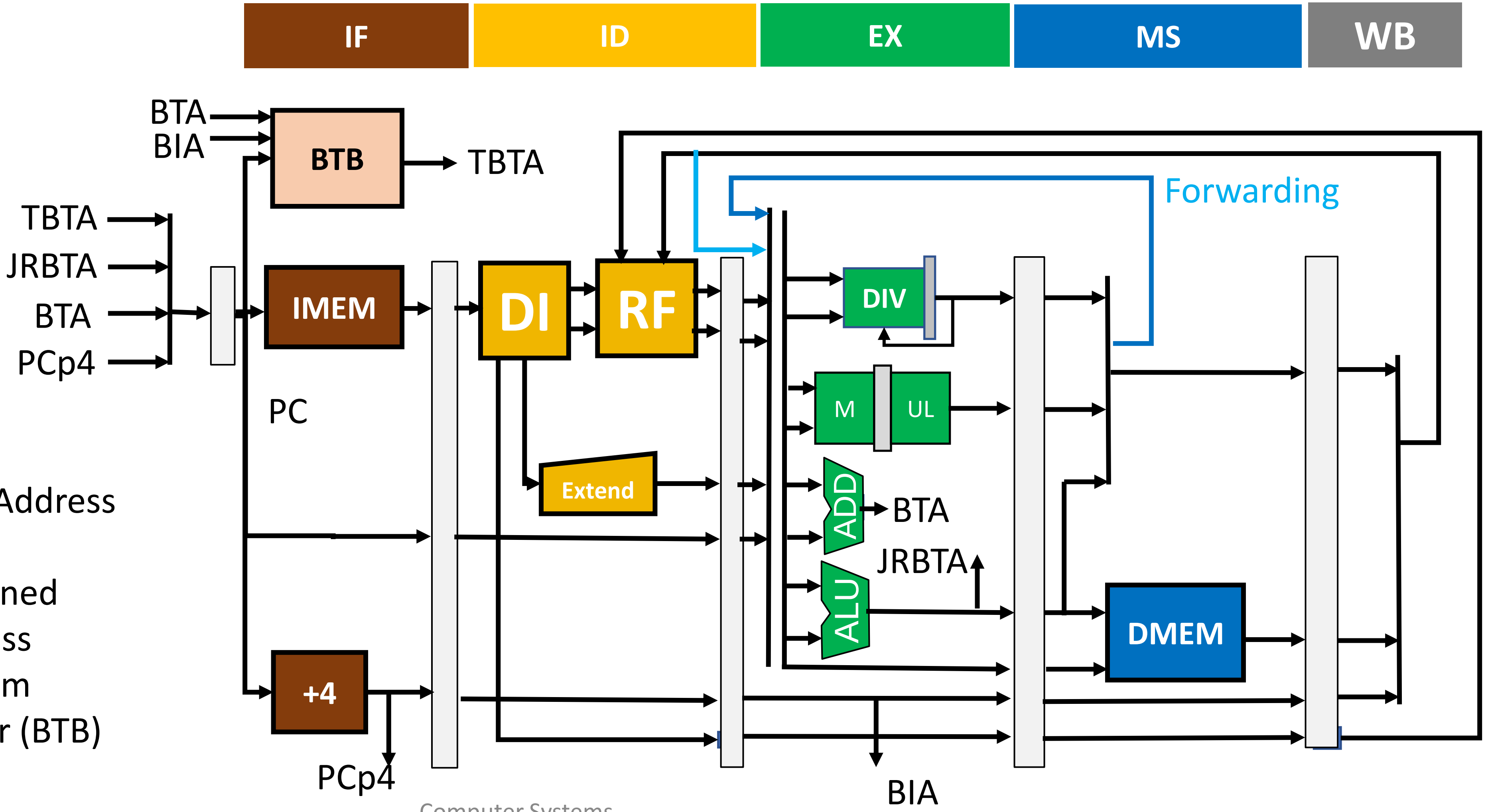


Simplified Illustration Style for Multiplexing



Scalar Five-Stage Pipeline with Multi-cycle FUs and Forwarding

- Multi-cycle Functional Units are integrated into the EX stage
- Simplified diagram

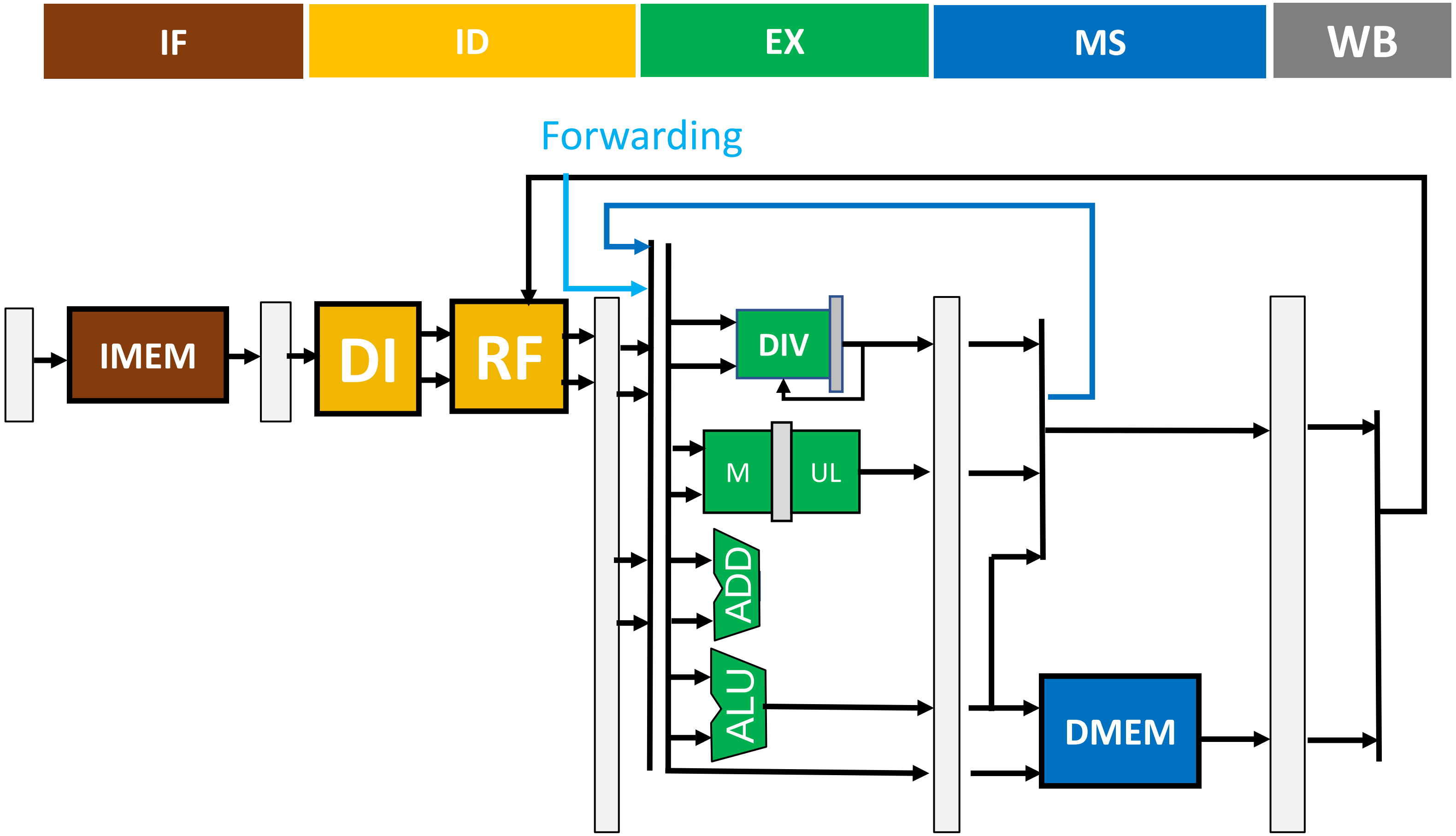


BTA: Branch Target Address
PCp4: PC+4
JRBTA: Register-defined branch target address
TBTA: Taken-BTA from Branch Target Buffer (BTB)

Scalar Five-Stage Pipeline with Multi-cycle FUs and Forwarding

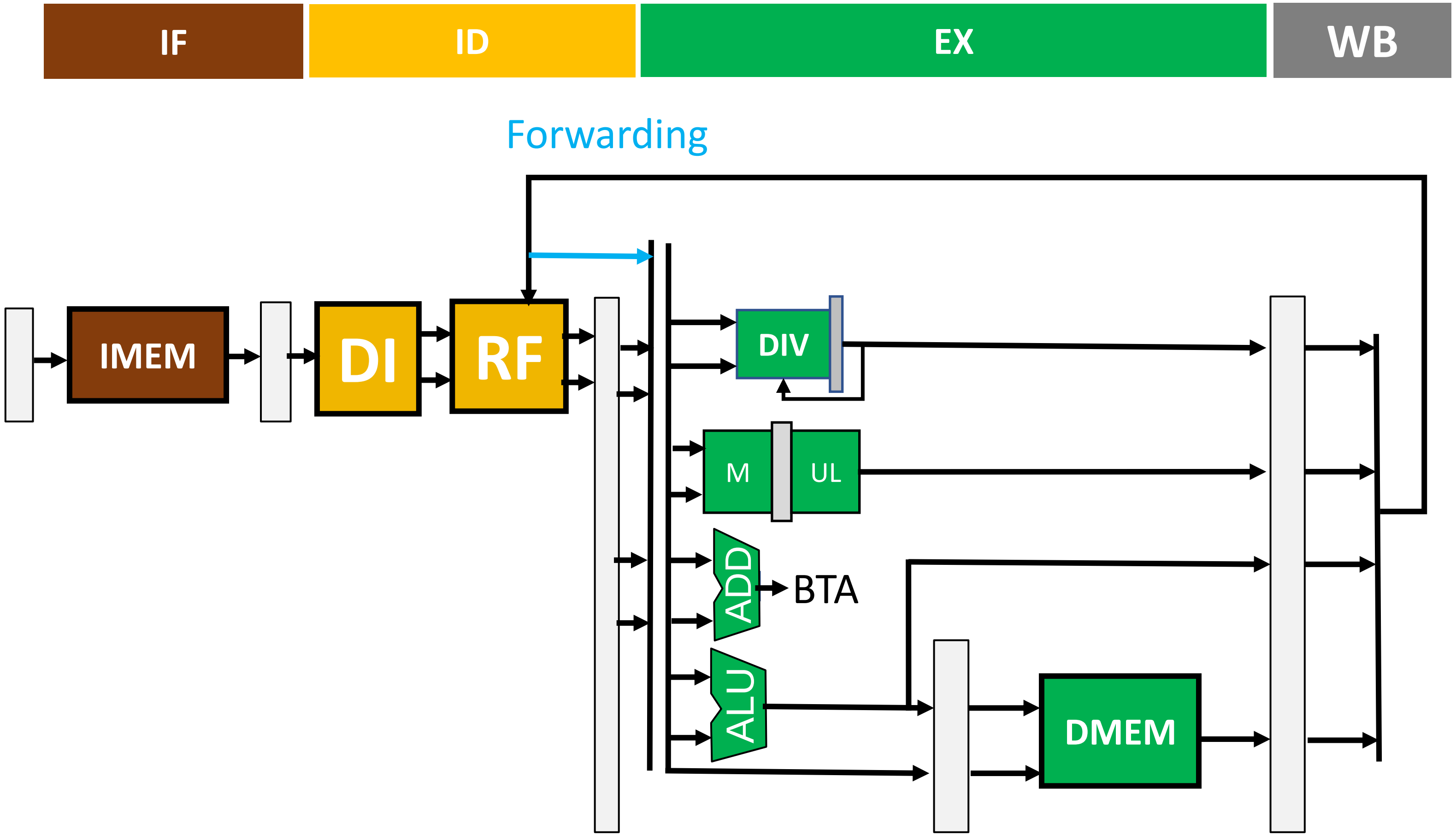
- Multi-cycle Functional Units are integrated into the EX stage
- **Further simplified diagram** (PC Generation, Extend, PC+rd address not shown, but of course still needed!)

Focus on the computation flow



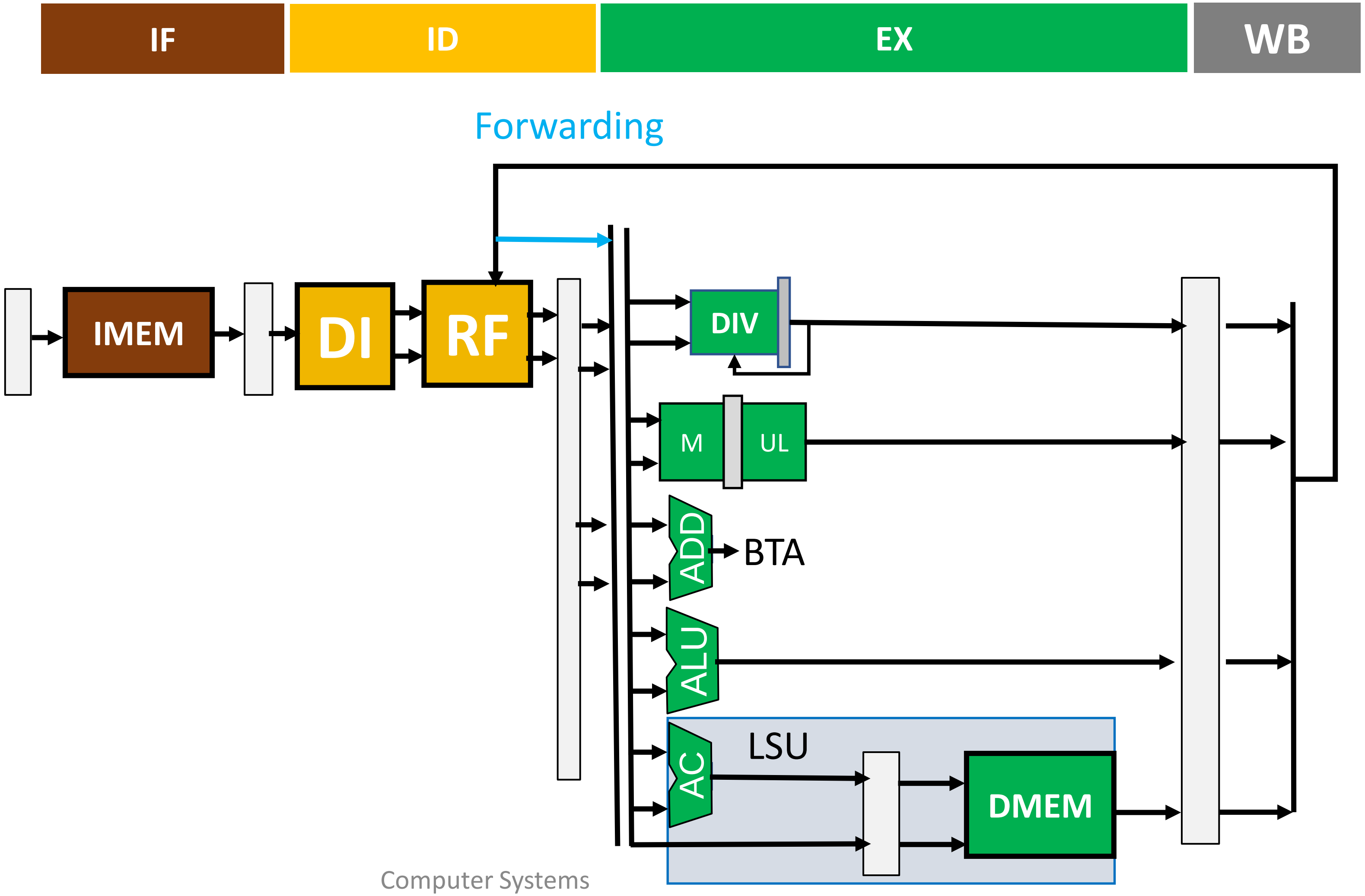
Scalar Four-Stage Pipeline with Multi-cycle FUs with Forwarding

- The DIV and MUL do not need to make memory accesses
- Move the memory stage (MS) after the ALU (which is required for the address computation for load/store)
- Merges MS and EX stage (four stages)
- Single forwarding path required in four-stage pipeline
- Such changes need additional control in control path



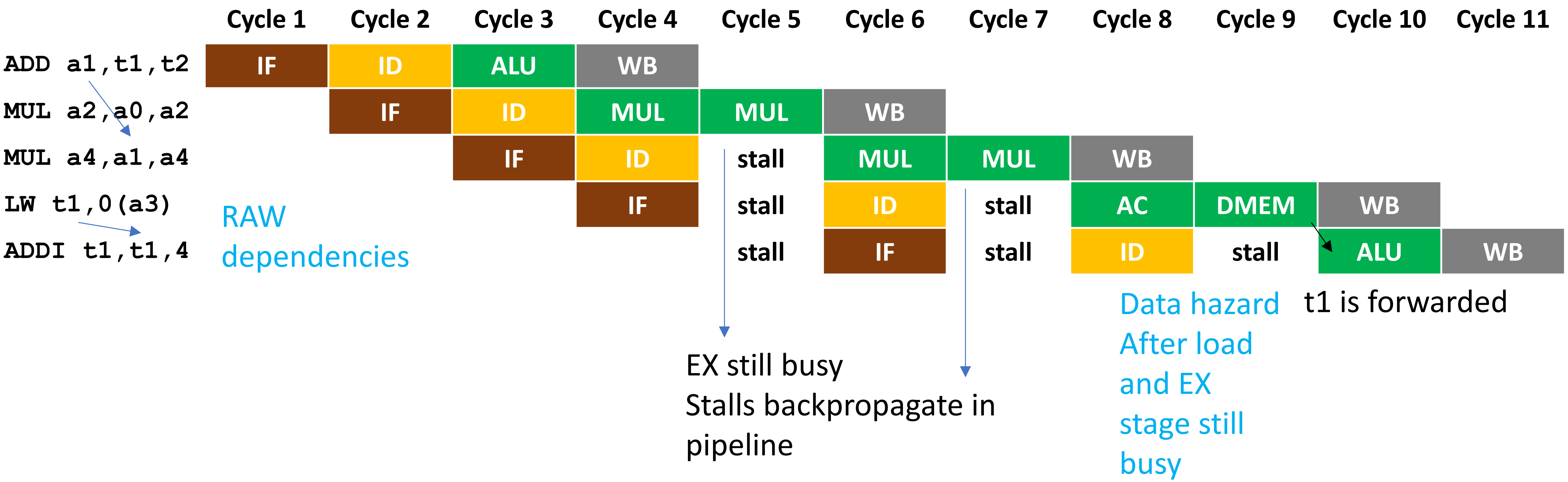
Scalar Four-Stage Pipeline with Multi-cycle FUs and Load Store Unit (LSU)

- We can add a second address computation adder (AC) to form a simple so-called load/store unit (LSU)



Execution Scheme: Four-Stage In-Order Scalar Pipeline

- The EX stage has an execution scheme defined by the processor control path
- Version 1: Static In-order Scheduling
  - ➢ Allow only one single instruction in the EX stage
  - ➢ Data hazards: Operands are forwarded by previous instruction



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11
`ADD a1,t1,t2`	IF	ID	ALU	WB							
`MUL a2,a0,a2`		IF	ID	MUL	MUL	WB					
`MUL a4,a1,a4`			IF	ID	stall	MUL	MUL	WB			
`LW t1,0(a3)`				IF	stall	ID	stall	AC	DMEM	WB	
`ADDI t1,t1,4`					stall	IF	stall	ID	stall	ALU	WB

RAW dependencies

EX still busy
Stalls backpropagate in pipeline

Data hazard After load and EX stage still busy

t1 is forwarded

Execution Scheme: Scalar Four-Stage Pipeline with Pipelined FUs

- Version 2: Static In-order Scheduling exploiting Pipelined FUs

➢ Allow only one single instruction in EX stage

➢ Except for: Pipelined MUL can use Initialization Interval for two consecutive MUL (still need to check for RAW dependency between the MUL)

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`ADD a1,t1,t2`	IF	ID	ALU	WB						
`MUL a2,a0,a2`		IF	ID	MUL(s1)	MUL(s2)	WB				
`MUL a4,a1,a4`			IF	ID	MUL(s1)	MUL(s2)	WB			
`LW t1,0(a3)`				IF	ID	stall	AC	DMEM	WB	
`ADDI t1,t1,4`					IF	stall	ID	stall	ALU	WB

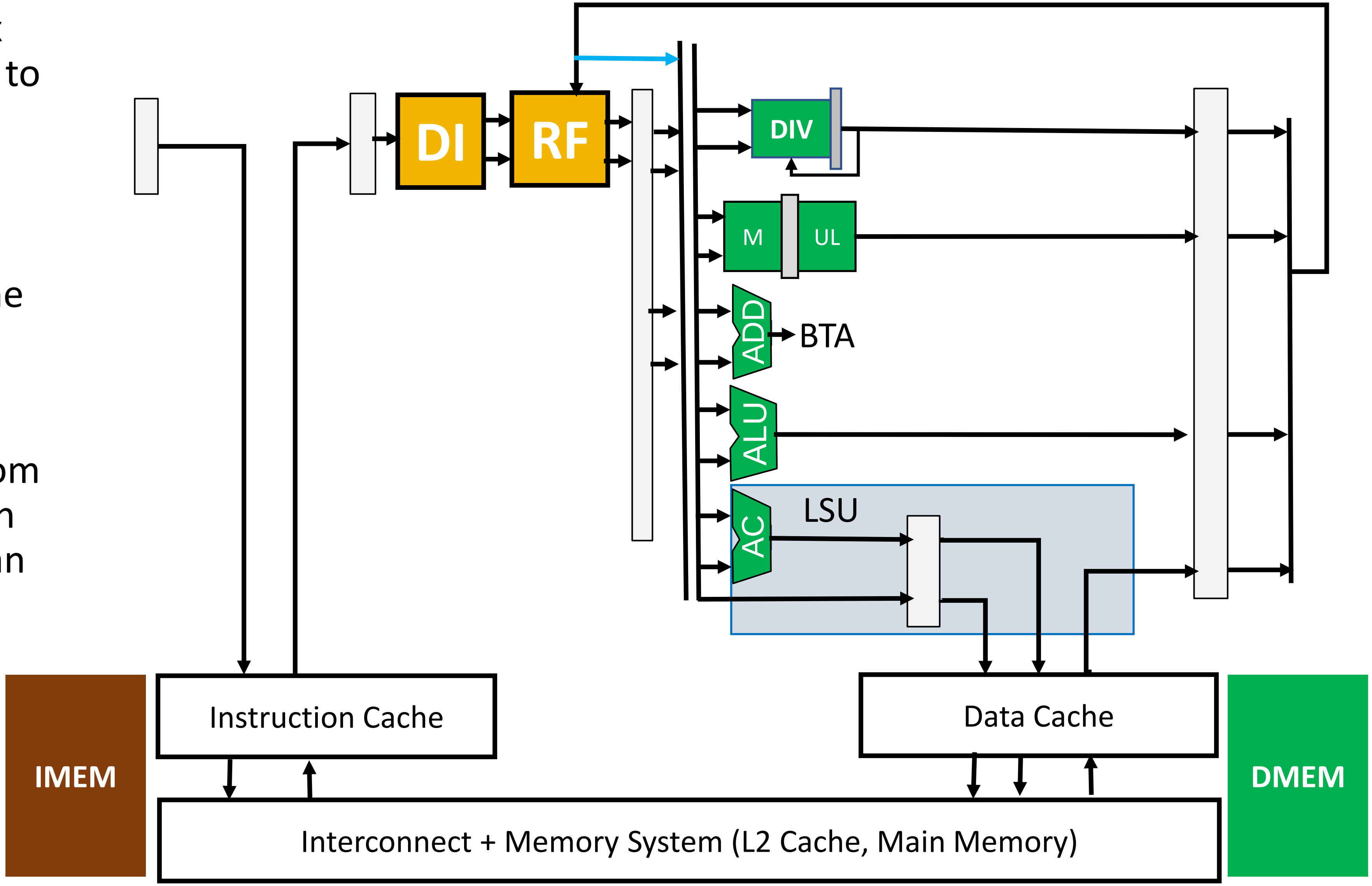
Load / Store Optimizations

Memory System

- The memory for more complex processors usually uses caches to allow for fast accesses

- Memory latency depends whether the data is found in the cache (cache hit/miss)

- Also instructions are loaded from caches, so also instruction fetch may require several cycles on an instruction cache miss.



Instruction Cache Misses

- Instruction cache miss causes several cycles of delay for instruction fetch (IF), depending on speed to catch fresh instruction block from memory system

- Instructions are usually reloaded to cache in blocks (cache line size) so that usually there are several cache hits after a cache miss (depending on jumps/branches in program)

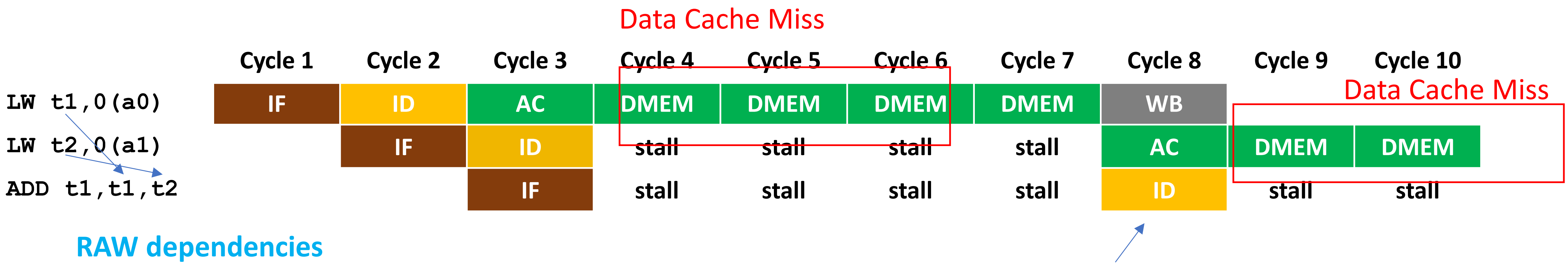
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`ADD a2,t1,t2`	IF	ID	ALU	WB						
`MUL a2,a0,a2`		IF	IF	IF	IF	ID	MUL	MUL	WB	
`MUL a4,a1,a4`						IF	ID	MUL	MUL	WB

Instruction Cache Miss

- Advanced caches pre-fetch the next block before the cache miss happens to hide cache refill latencies.

Load Cache Miss

- Data cache misses lead to extra cycles for loads as the data needs to get fetched from another memory (level 2 cache, main memory)
- Example (function vec_add, see first session): We load from two different addresses a0 and a1 (worst case both loads lead to a data cache miss)



Data Cache Miss

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`LW t1,0(a0)`	IF	ID	AC	DMEM	DMEM	DMEM	DMEM	WB		
`LW t2,0(a1)`		IF	ID	stall	stall	stall	stall	AC	DMEM	DMEM
`ADD t1,t1,t2`			IF	stall	stall	stall	stall	ID	stall	stall

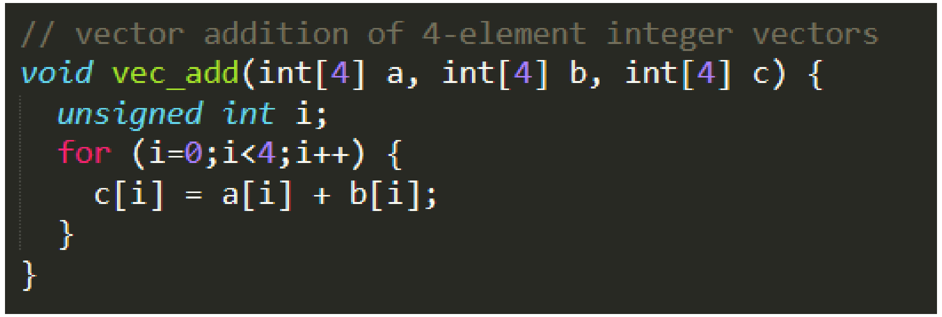
Data Cache Miss

RAW dependencies

ID here because stall on previous instruction finished

Example vec_add: Loads from two different addresses (a0,a1)

- Example C-Code 3



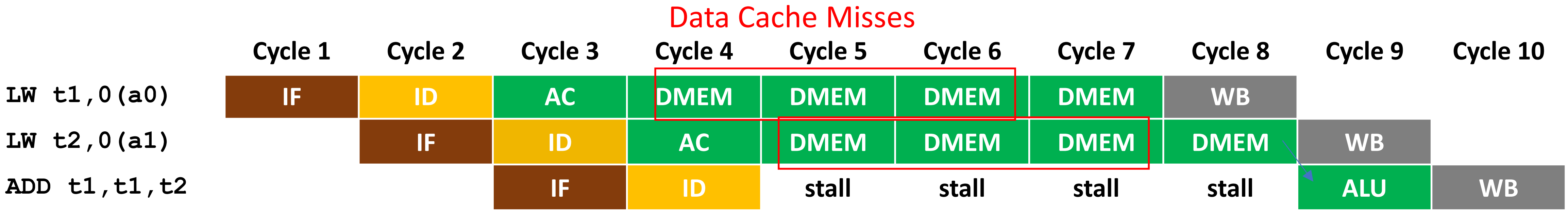
```c
// vector addition of 4-element integer vectors
void vec_add(int[4] a, int[4] b, int[4] c) {
  unsigned int i;
  for (i=0;i<4;i++) {
    c[i] = a[i] + b[i];
  }
}
```

RISC-V Code

```
# base address of a: a0,
# base address of b: a1,
# base address of c: a2,
# i: t0,  constant 4: t3
vec_add:
  LI t0,0        # i=0
  LI t3,4        # t3=4
vec_add_for:
  LW t1,0(a0)     # t1 = a[i]
  LW t2,0(a1)     # t2 = b[i]
  ADD t1,t1,t2    # t1 = a[i] + b[i]
  SW t1,0(a2)     # c[i] = t1
  ADDI a0,a0,4    #next element is base address + 4
  ADDI a1,a1,4    #next element is base address + 4
  ADDI a2,a2,4    #next element is base address + 4
  ADDI t0,t0,1    # i++
  BLTU t0,t3,vec_add_for # for (i < 4)
  RET   # void return
```

Nonblocking Loads (1/2)

- Load accesses are for longer times *in flight* due to cache misses
- Most interconnects/caches allow to overlap multiple memory accesses
- Allows to execute multiple load accesses in overlapping fashion
- Example (function vec_Add): Cache observes both addresses for load accesses and may need to reload cache lines for both accesses when both miss.

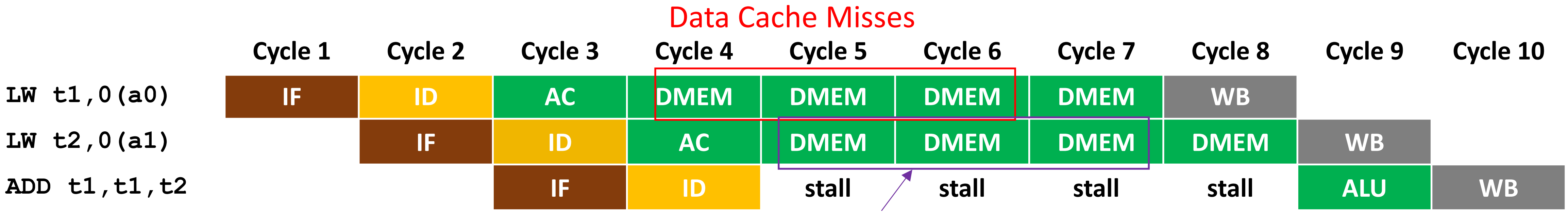


Data Cache Misses

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`LW t1,0(a0)`	IF	ID	AC	DMEM	DMEM	DMEM	DMEM	WB		
`LW t2,0(a1)`		IF	ID	AC	DMEM	DMEM	DMEM	DMEM	WB	
`ADD t1,t1,t2`			IF	ID	stall	stall	stall	stall	ALU	WB

Nonblocking Loads (2/2)

- Cache usually returns values in-order (some caches/interconnects support to return data out-of-order)
- Example (function 3): When only the first load misses, the second load still needs to wait in the LSU when the LSU returns results in-order.



Data Cache Misses

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`LW t1,0(a0)`	IF	ID	AC	DMEM	DMEM	DMEM	DMEM	WB		
`LW t2,0(a1)`		IF	ID	AC	DMEM	DMEM	DMEM	DMEM	WB	
`ADD t1,t1,t2`			IF	ID	stall	stall	stall	stall	ALU	WB

No data cache miss, but we need to wait for first cache access to finish.

Store Cache Miss

- Depending on Store Policy: Write-back data cache:
  - Additional latencies for stores possible when a dirty cache line needs to be replaced.
  - Dirty cache line needs first to be written to memory before it can be replaced
- Write through data cache:
  - Long store latency because the data is written not only to cache but also to main memory.

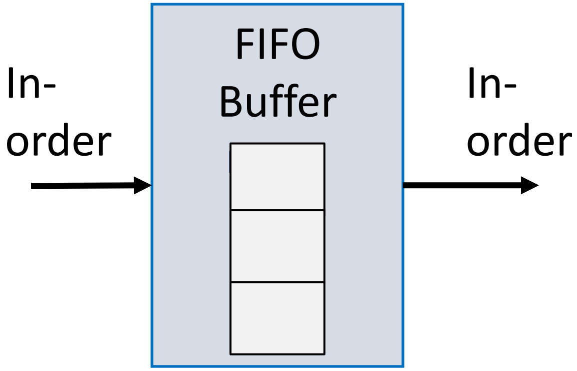
Example: We store to two different addresses a0 and a1 (first store misses)

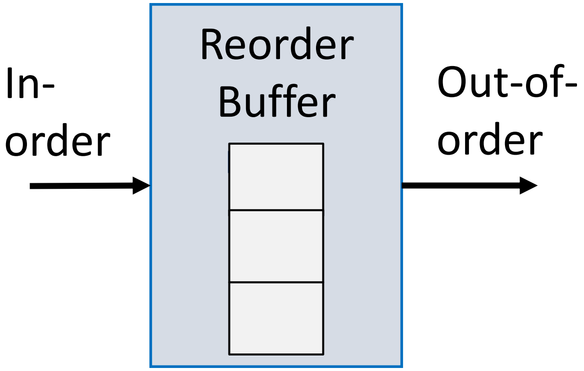
Data Cache Misses

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11
`SW t1,0(a0)`	IF	ID	AC	DMEM	DMEM	DMEM	DMEM	WB			
`SW t2,0(a1)`		IF	ID	stall	stall	stall	stall	AC	DMEM	WB	
`LI t2,4`			IF	stall	stall	stall	stall	ID	stall	ALU	WB

Buffers

- A buffer can store several values
- FIFO (First-in-first-out) buffer: Values can be read only from the buffer in the same order they are written to the buffer
- Reorder buffer: We can look up and read any value in the buffer





Store Buffer

- It is not really necessary to wait until a store write completes

- Store Unit (SU) with Store Buffer:
  - ➢ Put store address and data to store buffer (sometimes called *“Posted stores”)*
  - ➢ Store buffer performs memory store access (MSA) independently from pipeline
  - ➢ Only stall pipeline for stores when store buffer is full

- Load Unit (LU): Load more complex:
  - ➢ need to first look whether address is in store buffer then in cache
  - ➢ or need to wait until SB is empty.

Nonblocking Stores with Store Buffer

- Store accesses are for longer times *in flight* due to cache misses
- Store Buffer store accesses and pipeline continues execution
- Store Buffer writes data to memory via Memory Store Access (MSA).
- Only stall pipeline for stores when store buffer is full
- Example:

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`SW t1,0(a0)`	IF	ID	AC	SB	SB	SB	MSA			
`SW t2,0(a1)`		IF	ID	AC	SB	SB	SB	MSA		
`LI t2,4`			IF	ID	ALU	WB				

Execution Scheme: Scalar Four-Stage Pipeline with Pipelined FUs and Load Store Optimization

- Version 3: Static Scheduling with pipelined FUs and Load Store Optimization

➢ Allow only one single instruction in EX stage

➢ Except for:

  ➢ Pipelined MUL can use Initialization Interval for two consecutive MUL

  ➢ Certain number of nonblocking Loads can be in EX stage (then EX stalls)

  ➢ Certain number of stores can be posted in the SB depending on SB size (EX stalls when SB full). When Store is posted in SB, it does not count as instruction in EX stage.

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11	Cycle 12
`ADD a2,t1,t2`	IF	ID	ALU	WB								
`MUL a2,a0,a2`		IF	ID	MUL(s1)	MUL(s2)	WB						
`MUL a4,a1,a4`			IF	ID	MUL(s1)	MUL(s2)	WB					
`SW a2,0(a3)`				IF	ID	stall	AC	SB	SB	MSA		
`ADDI a3,a3,4`					IF	stall	ID	ALU	WB			
`SW a2,0(a3)`						stall	IF	ID	AC	SB	SB	MSA

Performance of Scalar Four-Stage Pipeline with Pipelined FUs and Load Store Optimization

- We still only allow one instruction to execute in EX stage except for some instruction types (MUL, Store, Load) in Version 3
- Multi-cycle operations cause many stalls (stiff scalar execution scheme)

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	
`ADD a2,t1,t2`	IF	ID	ALU	WB							
`MUL a2,a0,a2`		IF	ID	MUL(s1)	MUL(s2)	WB					
`DIV a4,a1,a4`			IF	ID	stall	DIV	DIV	DIV	DIV	WB	
`LW t1,0(a3)`				IF	stall	ID	stall	stall	stall	AC	…
`ADDI a3,a3,4`					stall	IF	stall	stall	stall	ID	…

- Can we interleave instructions to make better use of parallel units, maybe even just start them when they are ready, possibly out-of-order (OoO)?
- We want to exploit so-called **Instruction Level Parallelism**

Challenges for Exploiting Instruction Level Parallelism

Challenges for Exploiting Instruction Level Parallelism: Structural Hazards

- Start instructions in EX stage when FUs are available?
- Challenge: Structural Hazards, e.g. in WB Stage

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`ADD a2,t1,t2`	IF	ID	ALU	WB						
`MUL a2,a0,a2`		IF	ID	MUL(s1)	MUL(s2)	WB				
`MUL a4,a1,a4`			IF	ID	MUL(s1)	MUL(s2)	WB			
`LW t1,0(a3)`				IF	ID	AC	DMEM	WB		
`ADDI a3,a3,4`					IF	ID	ALU	WB		

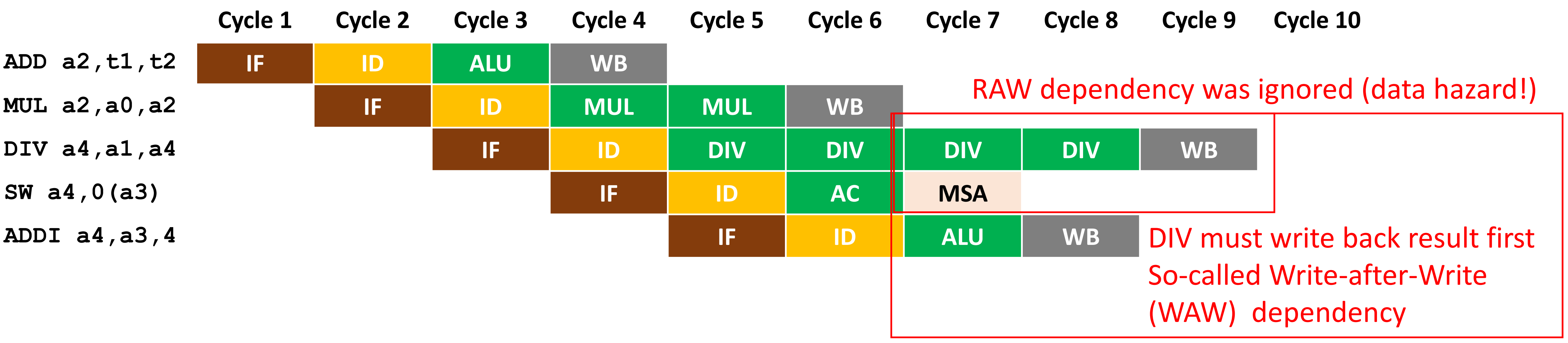
Two WB in same cycle!
WB collision!
Structural Hazard!

Challenges for Exploiting Instruction Level Parallelism: Instruction Dependencies

- Start instructions in EX stage when FUs are available?

➢Instructions can *overtake* each other due to different FU latencies.

- **Challenge:** The assembly program defines a **program order** for the instructions.
- Requires consideration of instruction dependencies during pipelined execution to preserve program order.



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`ADD a2,t1,t2`	IF	ID	ALU	WB						
`MUL a2,a0,a2`		IF	ID	MUL	MUL	WB				
`DIV a4,a1,a4`			IF	ID	DIV	DIV	DIV	DIV	WB	
`SW a4,0(a3)`				IF	ID	AC	MSA			
`ADDI a4,a3,4`					IF	ID	ALU	WB		

RAW dependency was ignored (data hazard!)

DIV must write back result first
So-called Write-after-Write (WAW) dependency

Instruction Dependencies

A closer look at RAW, WAR and WAW!

Types of Instruction Dependencies

- Read-after-Write (RAW): Also „*True dependency*“
  - Result of one instruction (write) is needed as input for another instruction (read)
  - May cause data hazards (*we seen this one already*)

- Write-after-Read (WAR): Also „*anti-dependency*“
  - A value is used (read) and then updated (write)
  - The update (write) is not allowed to overtake the use (read)

- Write-after-Write (WAW): Also „*output dependency*“
  - A value us updated (write) and then updated again (write)
  - The second update may not overtake the first update
  - Often created when registers are reused for different variables

Example for RAW:

XOR **a1**,a2,a4
RAW
ADD a3,**a1**,t1

Example for WAR:

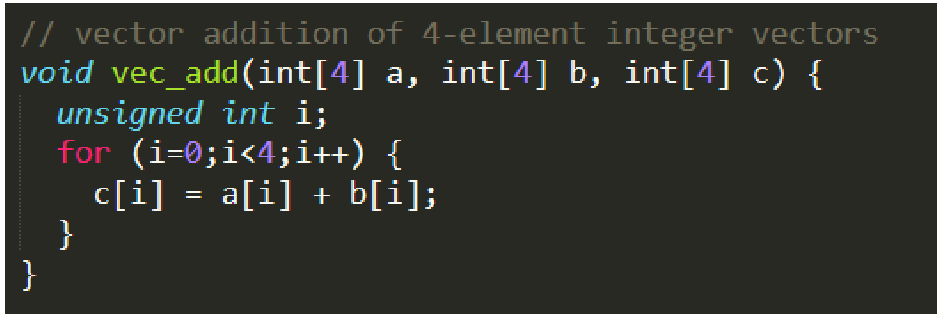
SW a1,0(**a2**)
WAR
ADDI **a2**,a3,4

Example for WAW:

LW **a1**,0(a2)
WAW
LI **a1**,a3,4

Dep. For Example Program (vec_add)

- Example C-Code 3



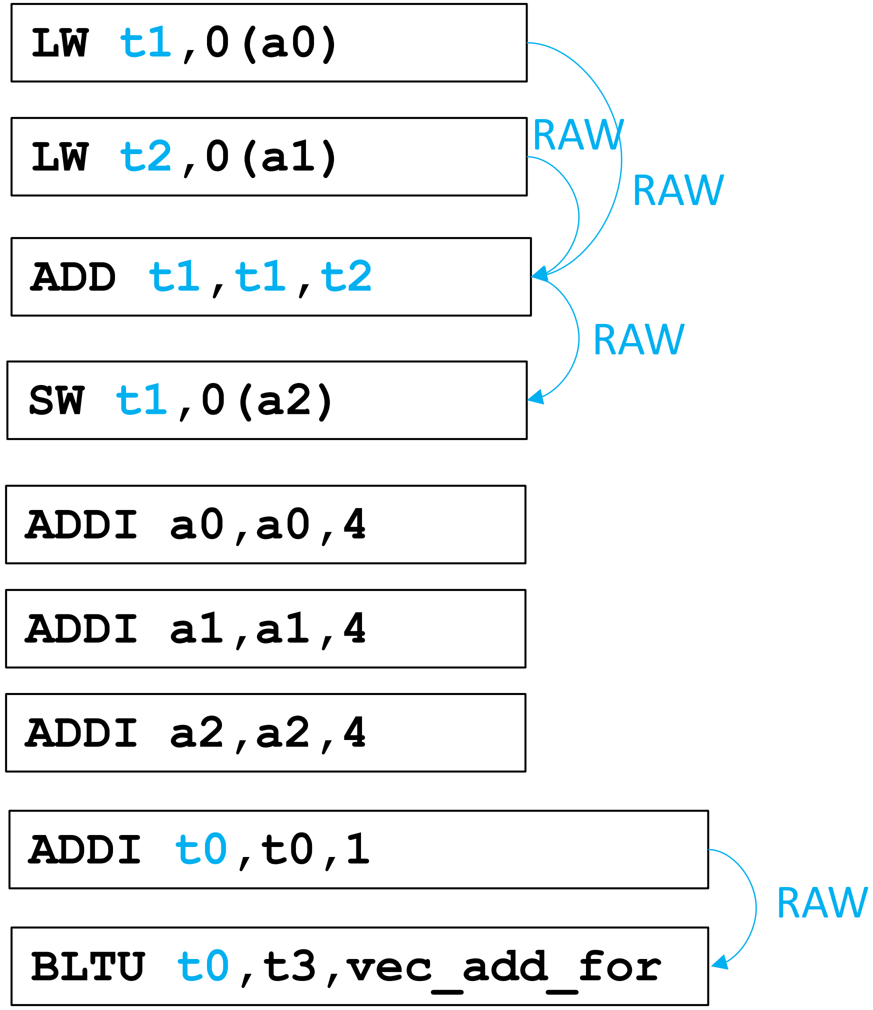
```c
// vector addition of 4-element integer vectors
void vec_add(int[4] a, int[4] b, int[4] c) {
  unsigned int i;
  for (i=0;i<4;i++) {
    c[i] = a[i] + b[i];
  }
}
```

```
# base address of a: a0,
# base address of b: a1,
# base address of c: a2,
# i: t0,  constant 4: t3
vec_add:
  LI t0,0        # i=0
  LI t3,4        # t3=4
vec_add_for:
  LW t1,0(a0)    # t1 = a[i]
  LW t2,0(a1)    # t2 = b[i]
  ADD t1,t1,t2   # t1 = a[i] + b[i]
  SW t1,0(a2)    # c[i] = t1
  ADDI a0,a0,4   #next element is base address + 4
  ADDI a1,a1,4   #next element is base address + 4
  ADDI a2,a2,4   #next element is base address + 4
  ADDI t0,t0,1   # i++
  BLTU t0,t3,vec_add_for # for (i < 4)
  RET   # void return
```

Dep. For Example Program (vec_add) (RAW)

- Mark all RAW dependencies for the following code block:

```
LI t0,0
LI t3,4
vec_add_for:
  LW t1,0(a0)
  LW t2,0(a1)
  ADD t1,t1,t2
  SW t1,0(a2)
  ADDI a0,a0,4
  ADDI a1,a1,4
  ADDI a2,a2,4
  ADDI t0,t0,1
  BLTU t0,t3,vec_add_for
  RET
```

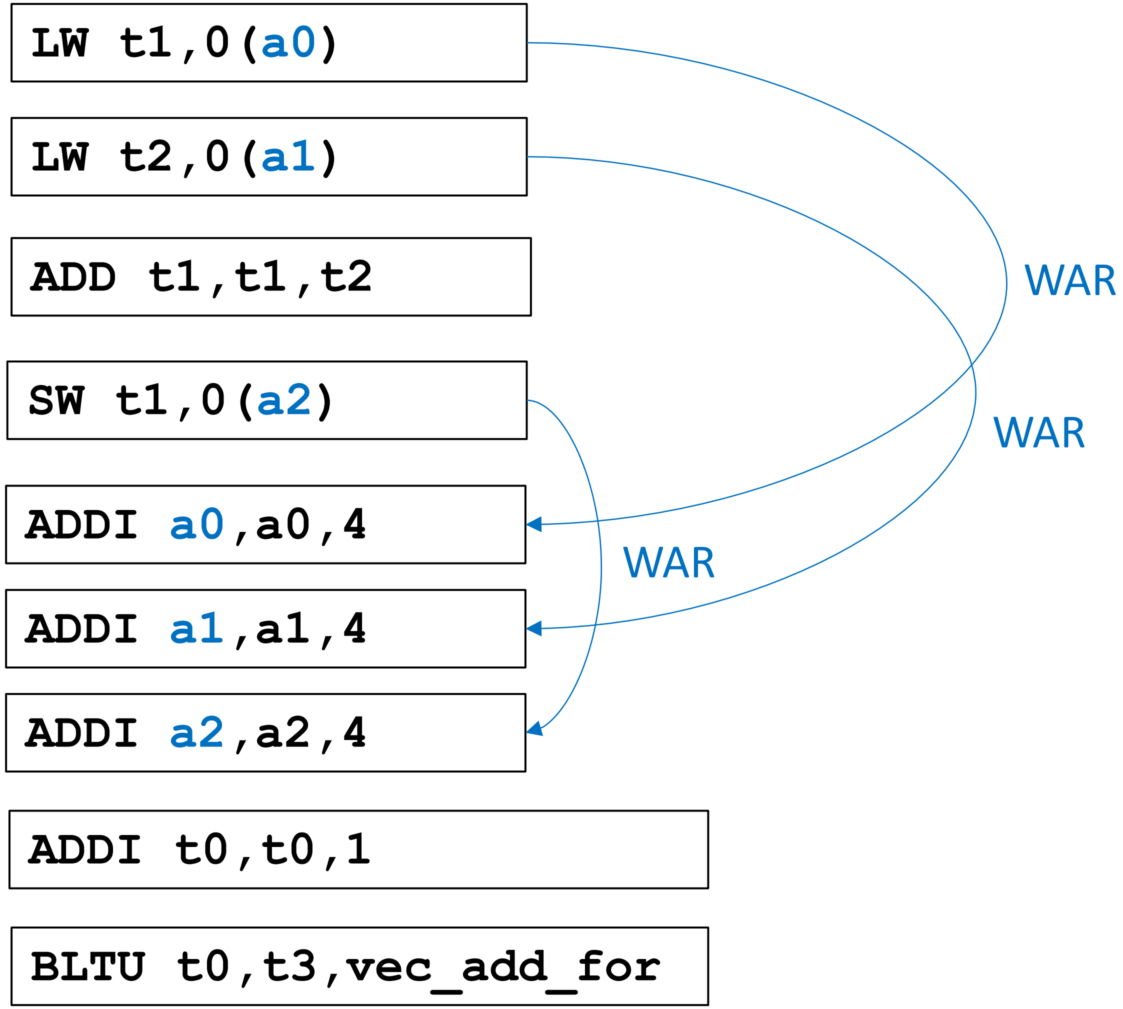


```
LW t1,0(a0)
LW t2,0(a1)
ADD t1,t1,t2
SW t1,0(a2)
ADDI a0,a0,4
ADDI a1,a1,4
ADDI a2,a2,4
ADDI t0,t0,1
BLTU t0,t3,vec_add_for
```

Dep. For Example Program (vec_add) (WAR)

- Mark all WAR dependencies for the following code block:

```
LI t0,0
LI t3,4
vec_add_for:
  LW t1,0(a0)
  LW t2,0(a1)
  ADD t1,t1,t2
  SW t1,0(a2)
  ADDI a0,a0,4
  ADDI a1,a1,4
  ADDI a2,a2,4
  ADDI t0,t0,1
  BLTU t0,t3,vec_add_for
  RET
```

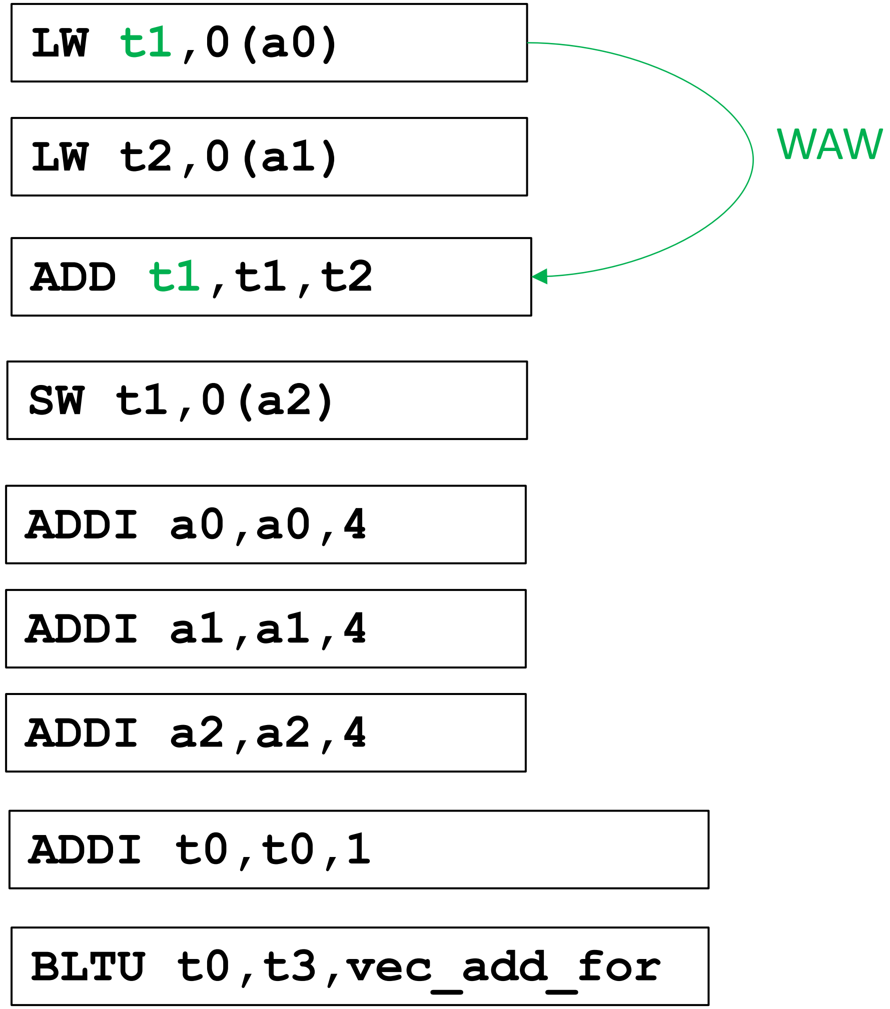


```
LW t1,0(a0)
LW t2,0(a1)
ADD t1,t1,t2
SW t1,0(a2)
ADDI a0,a0,4
ADDI a1,a1,4
ADDI a2,a2,4
ADDI t0,t0,1
BLTU t0,t3,vec_add_for
```

Dep. For Example Program (vec_add) (WAW)

- Mark all WAW dependencies for the following code block:

```
LI t0,0
LI t3,4
vec_add_for:
  LW t1,0(a0)
  LW t2,0(a1)
  ADD t1,t1,t2
  SW t1,0(a2)
  ADDI a0,a0,4
  ADDI a1,a1,4
  ADDI a2,a2,4
  ADDI t0,t0,1
  BLTU t0,t3,vec_add_for
  RET
```

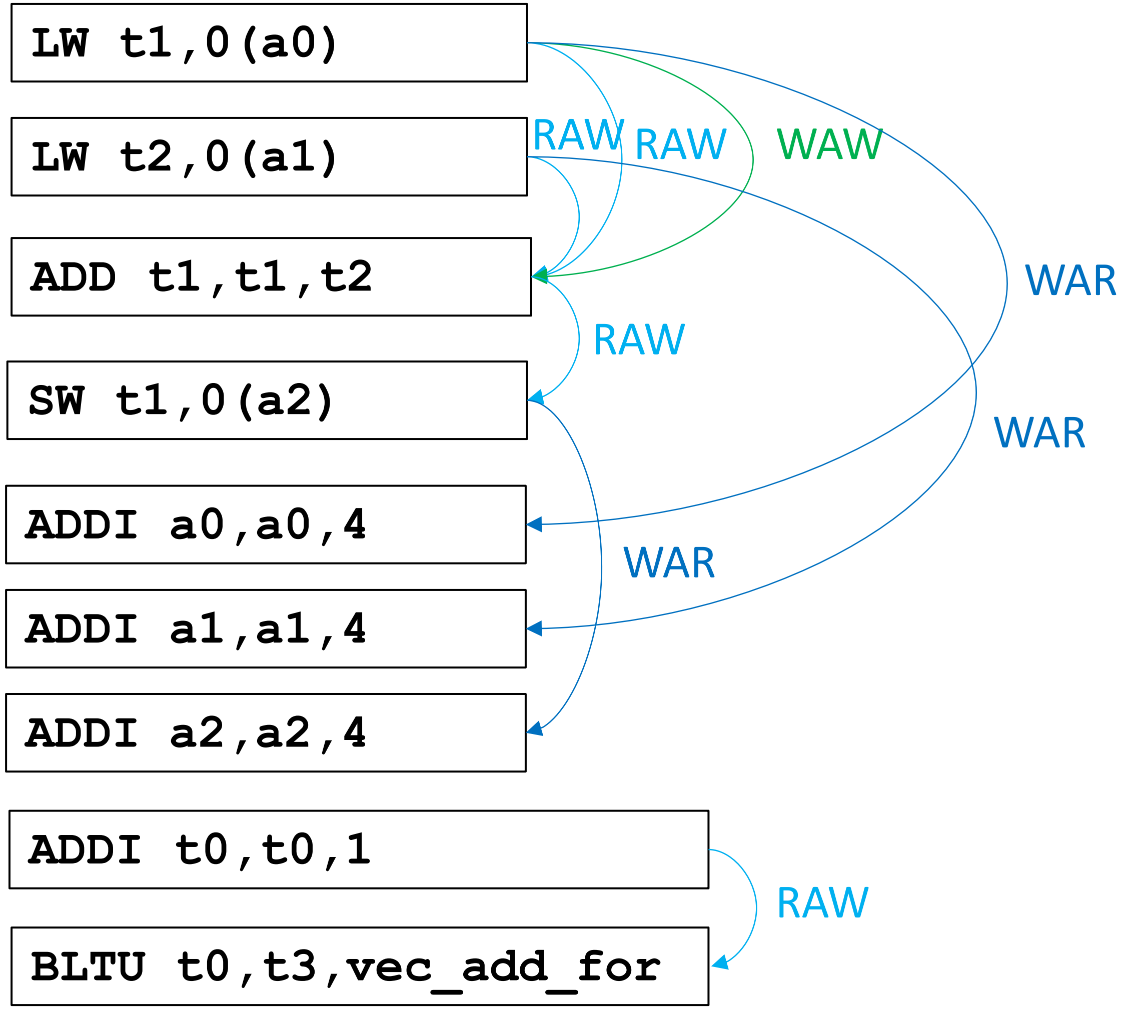


```
LW t1,0(a0)
LW t2,0(a1)
ADD t1,t1,t2
SW t1,0(a2)
ADDI a0,a0,4
ADDI a1,a1,4
ADDI a2,a2,4
ADDI t0,t0,1
BLTU t0,t3,vec_add_for
```

Dep. For Example Program (vec_add) (ALL)

- Mark all dependencies for the following code block:

```
LI t0,0
LI t3,4
vec_add_for:
  LW t1,0(a0)
  LW t2,0(a1)
  ADD t1,t1,t2
  SW t1,0(a2)
  ADDI a0,a0,4
  ADDI a1,a1,4
  ADDI a2,a2,4
  ADDI t0,t0,1
  BLTU t0,t3,vec_add_for
  RET
```



```
LW t1,0(a0)
LW t2,0(a1)
ADD t1,t1,t2
SW t1,0(a2)
ADDI a0,a0,4
ADDI a1,a1,4
ADDI a2,a2,4
ADDI t0,t0,1
BLTU t0,t3,vec_add_for
```

Challenges with Interleaving Instruction Execution in EX Stage

1. We have to consider **RAW, WAR** and **WAW** dependencies.

2. **Structural hazards** must be avoided, e.g., FU is already busy.

3. Some instructions can cause so-called **exceptions** (e.g. memory fault on load/store) (See optional content for what is required for precise exceptions).

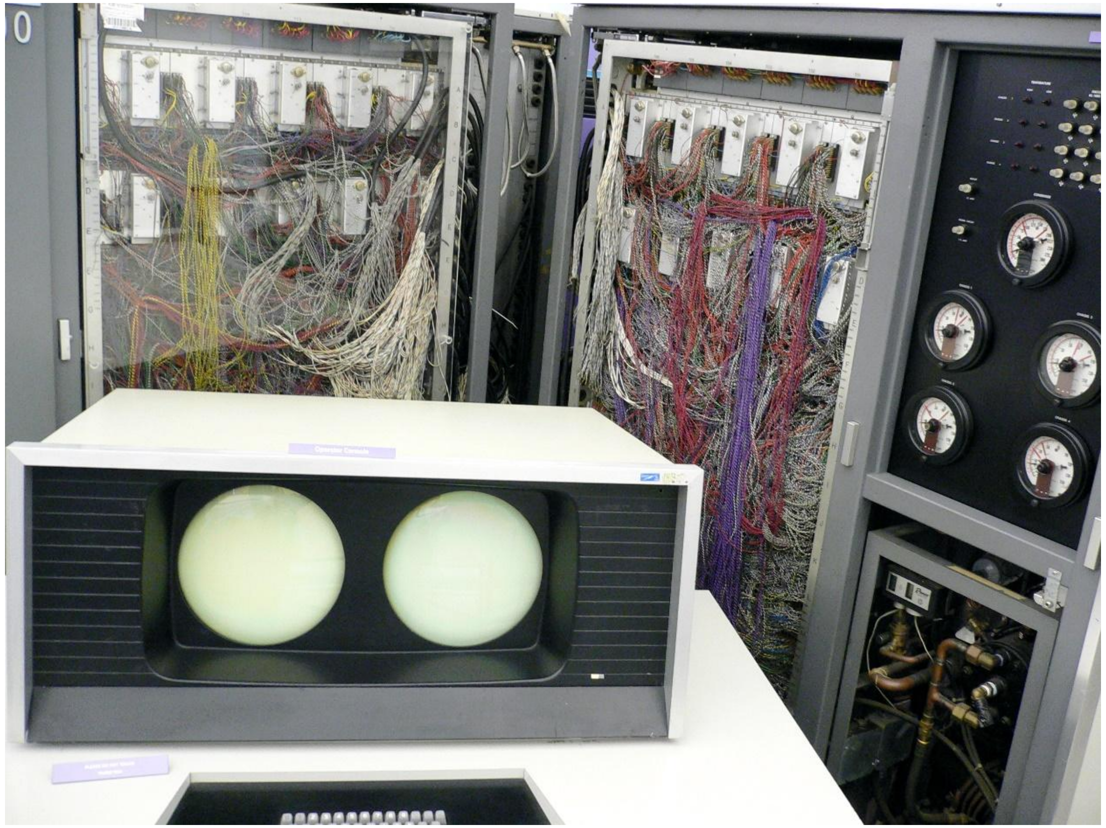
Dynamic Scheduling With Scoreboard

Out-of-Order (OoO, O3) Pipeline

Computer Architecture A Quantitative Approach – Section C7

The CDC 6600 Project [‘1964]

- First implementation of Scoreboard (Out-of-Order)
- 16 separate non-pipelined functional units (7 int, 4 Floating Point (FP), 5 memory)
- **Out-of-order (OoO) execution** is also called **dynamic instruction scheduling**



Steve Jurvetson
CC BY 2.0

The CDC 6600 Project [‘1964]

CDC 6600 Scoreboard

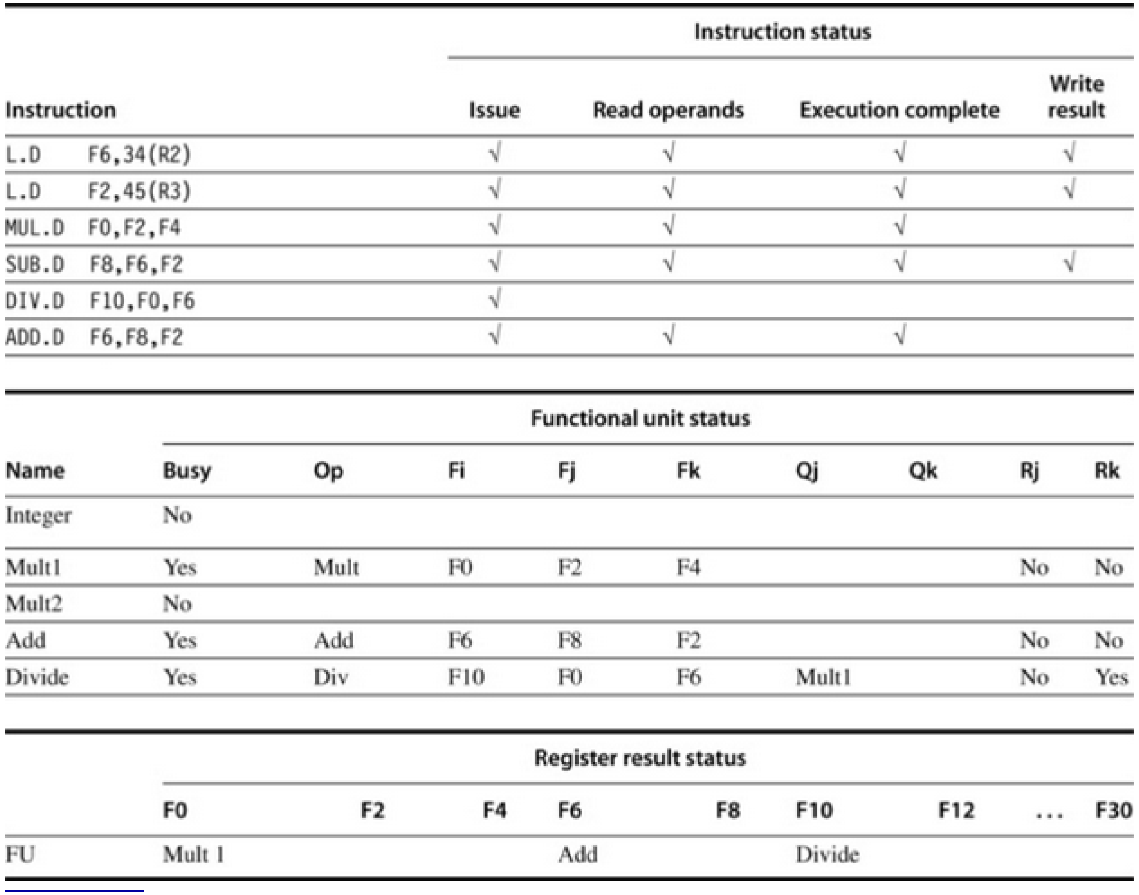
- Three main components

➢Instruction status

➢Functional unit status

➢Register result status

- For an example of use of Scoreboard in CDC 6600 see:
- *Computer Architecture*
  A Quantitative Approach – Section C7



	Instruction status			
Instruction	Issue	Read operands	Execution complete	Write result
`L.D    F6,34(R2)`	√	√	√	√
`L.D    F2,45(R3)`	√	√	√	√
`MUL.D  F0,F2,F4`	√	√	√	
`SUB.D  F8,F6,F2`	√	√	√	√
`DIV.D  F10,F0,F6`	√			
`ADD.D  F6,F8,F2`	√	√	√	

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult 1			Add		Divide			

Split of ID Stage

“To implement out-of-order execution, we must split the ID pipe stage into two stages:

- 1. *Issue*—Decode instructions, check for structural hazards.
- 2. *Read operands*—Wait until no data hazards, then read operands.”

- “In a **dynamically scheduled pipeline**, all instructions pass through the issue stage in order (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus **enter execution out of order**”

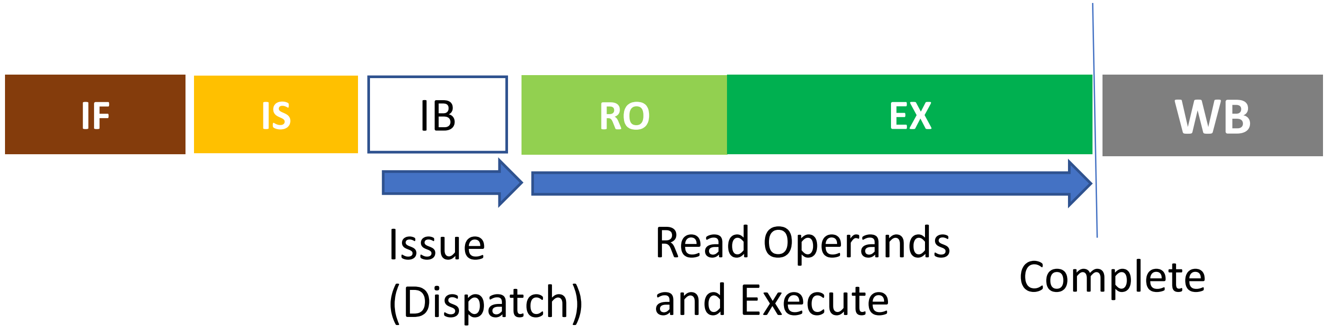
-- *Computer Architecture A Quantitative Approach – 5th Ed. Section C7*

Steps in Out-of-Order Execution (Scheme 1*)

- *1. **Issue***
  - **Functional unit is free**
  - No other active instruction has the same destination register (guarantee that **WAW hazards** cannot be present)
  - If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.
- 2. ***Read operands***
  - When source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution.
  - The scoreboard resolves **RAW hazards** dynamically in this step, and instructions may be sent into execution out of order.
- 3. ***Execution***
  - The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.
- 4. ***Write result***
  - Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for **WAR hazards** and stalls the completing instruction, if necessary.

-- **Computer Architecture A Quantitative Approach – 5th Ed. Section C7*

Steps in Out-of-Order Execution (Simpler Scheme 2**)



- **Issue Buffer (IB)** holds multiple instructions waiting to issue.
- Instruction Decode (ID) adds next instruction to IB if
  - there is space in IB and
  - the instruction does not have a **WAR** or **WAW dependency** with any instruction in IB.
- Instruction Issue (IS) can issue any instruction in IB whose
  - **RAW hazards are satisfied** to all previous instructions in IB
  - **FU is available**.
- Note: With writeback (WB) we delete the instruction from the IB, this may enable more instructions to issue as RAW dependencies are resolved.

-- **Inspired by *MIT course, Daniel Sanchez - http://csg.csail.mit.edu/6.823S20/Lectures/L09.pdf*

Example OoO Processor: Simple Scoreboard Data Structure

- Simplified CDC-style Scoreboard Data Structure to track execution
- For Scheme 2, One Issue Buffer
- Logical, not HW implementation

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete

Scoreboard (ScB)

FU Status (Ready?)

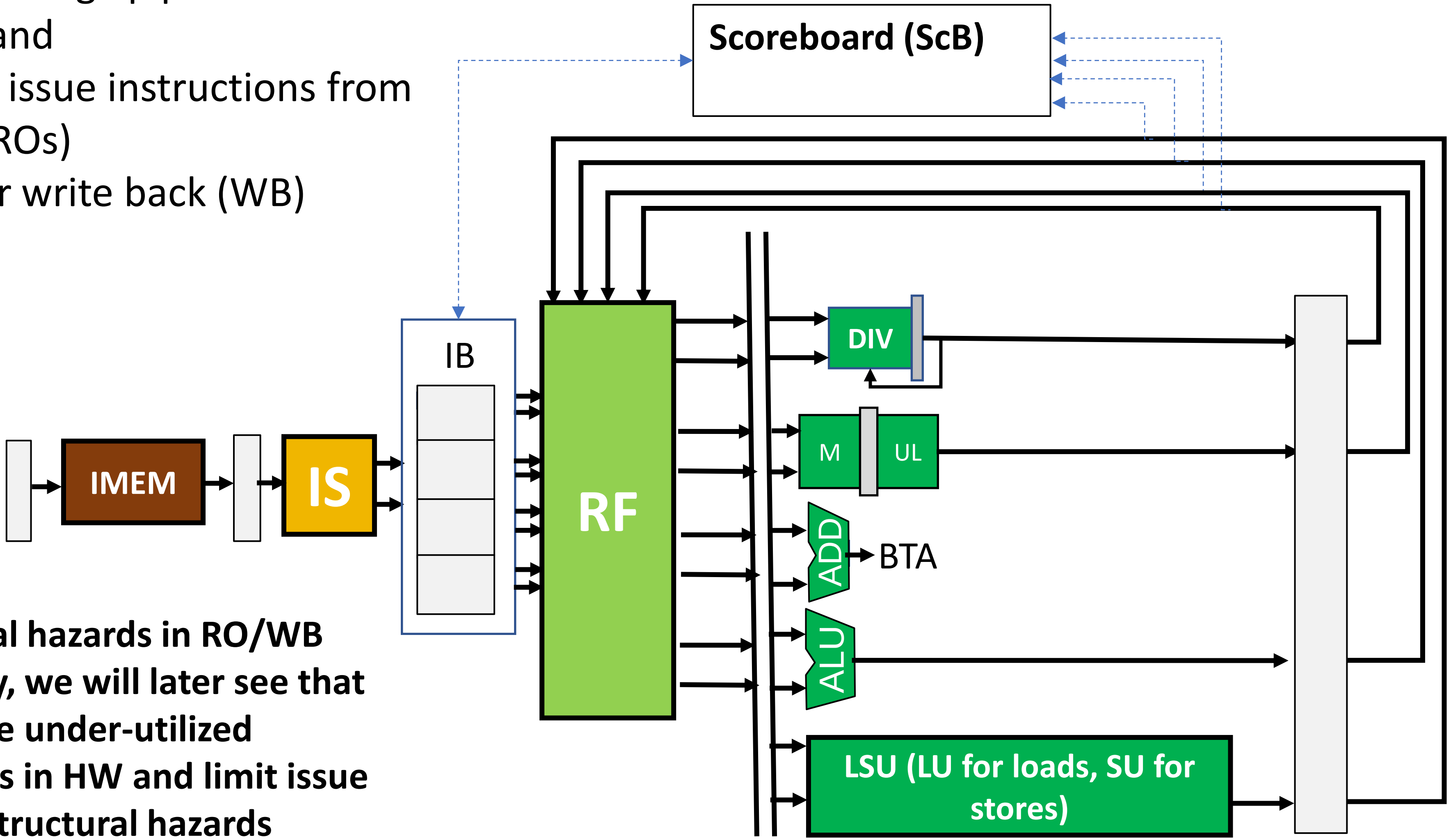
DIV	MUL	ALU	ADD	SU	LU

RO: Instruction read operands (started the computation)
Complete: Instruction finished computation (in last EX stage)

Example OoO Processor: Scoreboard Integration

Example four-stage pipeline with

- IB size 4 and
- 4 ports to issue instructions from buffer (4 ROs)
- 4 ports for write back (WB)



No structural hazards in RO/WB
This is costly, we will later see that the ports are under-utilized
-> limit ports in HW and limit issue or stall for structural hazards

Example OoO Processor: FUs in EX stage

For simplicity all FUs have fixed latency:

FU	**Latency**	**Initialization Interval**	
ALU	1	1	
ADD	1	1	
MUL	2	1	Pipelined
DIV	4	4	Serial (fixed latency)
LSU			
LU	2	1	Nonblocking
SU	1	1	Store buffered

- Instruction can only be issued when FU is available.
- SU and LU share same port, cannot be issued together
- We assume instruction cannot be issued to EX same cycle it was added to IB by ID

Example OoO Processor – Pipeline Diagram - Cycle 2

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
➡ `LW x12,8(x9)`	IF	IS																		
`LW x13,0(x7)`		IF																		
`DIV x17,x13,x12`																				
`ADDI x18,x12,28`																				
`MUL x19,x12,x18`																				
`MUL x10,x17,x14`																				
`ADD x10,x10,x13`																				
`SW x10,0(x11)`																				
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
➡ LW	x12	x9		8		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU

Example OoO Processor – Pipeline Diagram - Cycle 3

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
`LW x12,8(x9)`	IF	IS	LU																
➡ `LW x13,0(x7)`		IF	IS																
`DIV x17,x13,x12`			IF																
`ADDI x18,x12,28`																			
`MUL x19,x12,x18`																			
`MUL x10,x17,x14`																			
`ADD x10,x10,x13`																			
`SW x10,0(x11)`																			
`LW x10,4(x8)`																			
`ADDI X13,x10,4`																			

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x12	x9		8	x	
➡ LW	x13	x7		0		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
					Busy 1

Example OoO Processor – Pipeline Diagram - Cycle 4

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU																
`LW x13,0(x7)`		IF	IS	LU																
➡ `DIV x17,x13,x12`			IF	IS																
`ADDI x18,x12,28`				IF																
`MUL x19,x12,x18`																				
`MUL x10,x17,x14`																				
`ADD x10,x10,x13`																				
`SW x10,0(x11)`																				
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x12	x9		8	x	x
LW	x13	x7		0	x	
➡ DIV	x17	x13	x12			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
					Busy 2

Example OoO Processor – Pipeline Diagram - Cycle 5

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU															
`DIV x17,x13,x12`			IF	IS	IB	RAW														
`ADDI x18,x12,28`				IF	IS															
`MUL x19,x12,x18`					IF															
`MUL x10,x17,x14`																				
`ADD x10,x10,x13`																				
`SW x10,0(x11)`																				
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x13	x7		0	x	x
DIV	x17	x13	x12			
ADDI	x18	X12		28		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
					Busy 1

Example OoO Processor – Pipeline Diagram - Cycle 6

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV														
`ADDI x18,x12,28`				IF	IS	ALU														
`MUL x19,x12,x18`					IF	IS														
`MUL x10,x17,x14`						IF														
`ADD x10,x10,x13`																				
`SW x10,0(x11)`																				
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
DIV	x17	x13	x12		x	
ADDI	x18	x12		28	x	x
MUL	x19	x12	x18			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
Busy		Busy			

Example OoO Processor – Pipeline Diagram - Cycle 7

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV													
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL													
➡ `MUL x10,x17,x14`						IF	IS													
`ADD x10,x10,x13`							IF													
`SW x10,0(x11)`																				
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
DIV	x17	x13	x12		x	
MUL	x19	x12	x18		x	
➡ MUL	x10	x17	x14			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
Busy	Busy(s1)				

Example OoO Processor – Pipeline Diagram - Cycle 8

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV												
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL												
`MUL x10,x17,x14`						IF	IS	IB	RAW											
`ADD x10,x10,x13`							IF	stall	WAW											
`SW x10,0(x11)`								stall												
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
DIV	x17	x13	x12		x	
MUL	x19	x12	x18		x	x
MUL	x10	x17	x14			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
Busy	Busy(s2)				

Example OoO Processor – Pipeline Diagram - Cycle 9

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV											
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	RAW										
`ADD x10,x10,x13`							IF	stall	stall	WAW										
`SW x10,0(x11)`								stall	stall											
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
DIV	x17	x13	x12		x	x
MUL	x10	x17	x14			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
Busy					

Example OoO Processor – Pipeline Diagram - Cycle 10

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL										
➡ `ADD x10,x10,x13`							IF	stall	stall	stall	WAW									
`SW x10,0(x11)`								stall	stall	stall										
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
MUL	x10	x17	x14		x	

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
	Busy(s1)				

Example OoO Processor – Pipeline Diagram - Cycle 11

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL									
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	WAW								
`SW x10,0(x11)`								stall	stall	stall	stall									
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
MUL	x10	x17	x14		x	x

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
	Busy(s2)				

Example OoO Processor – Pipeline Diagram - Cycle 12

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS								
`SW x10,0(x11)`								stall	stall	stall	stall	IF								
`LW x10,4(x8)`																				
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
ADD	x10	x10	x13			

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU

Example OoO Processor – Pipeline Diagram - Cycle 13

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU							
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS							
`LW x10,4(x8)`													IF							
`ADDI X13,x10,4`																				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
ADD	x10	x10	x13		x	x
SW		x11	x10	0		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
		busy			

Example OoO Processor – Pipeline Diagram - Cycle 14

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU						
➡ `LW x10,4(x8)`													IF	stall	WAR					
`ADDI X13,x10,4`														stall						

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
SW		x11	x10	0	x	x

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
				Busy 1	

Example OoO Processor – Pipeline Diagram - Cycle 15

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU	SB					
➡ `LW x10,4(x8)`													IF	stall	IS					
`ADDI X13,x10,4`														stall	IF					

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
➡ LW	x10	x18		4		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU

Example OoO Processor – Pipeline Diagram - Cycle 16

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU	SB					
`LW x10,4(x8)`													IF	stall	IS	LU				
`ADDI X13,x10,4`														stall	IF	IS				

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x10	x18		4	x	
ADDI	x13	x10		4		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
					Busy 1

Example OoO Processor – Pipeline Diagram - Cycle 17

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU	SB					
`LW x10,4(x8)`													IF	stall	IS	LU	LU			
`ADDI X13,x10,4`														stall	IF	IS	IB	RAW		

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
LW	x10	x18		4	x	x
ADDI	x13	x10		4		

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
					Busy 1

Example OoO Processor – Pipeline Diagram - Cycle 18

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU	SB					
`LW x10,4(x8)`													IF	stall	IS	LU	LU	WB		
`ADDI X13,x10,4`														stall	IF	IS	IB	ALU		

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete
ADDI	x13	x10		4	x	x

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU
		busy			

Example OoO Processor – Pipeline Diagram - Cycle 19

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB															
`LW x13,0(x7)`		IF	IS	LU	LU	WB														
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB										
`ADDI x18,x12,28`				IF	IS	ALU	WB													
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB											
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB								
`ADD x10,x10,x13`							IF	stall	stall	stall	stall	IS	ALU	WB						
`SW x10,0(x11)`								stall	stall	stall	stall	IF	IS	SU	SB					
`LW x10,4(x8)`													IF	stall	IS	LU	LU	WB		
`ADDI x13,x10,4`														stall	IF	IS	IB	ALU	WB	

10 instructions

3 cycles ramp-up (4-stage pipeline)

Total 19 cycles -3 cycles = 16 cycles

CPI = 1,6

Issue Buffer (IB)

Instruction	rd	rs1	rs2	Imm	RO	Complete

FU Status (Ready?)

DIV	MUL	ALU	ADD	SU	LU

Terminology

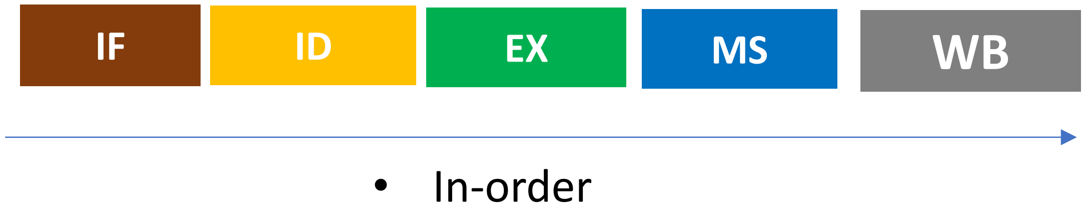
- Processors:

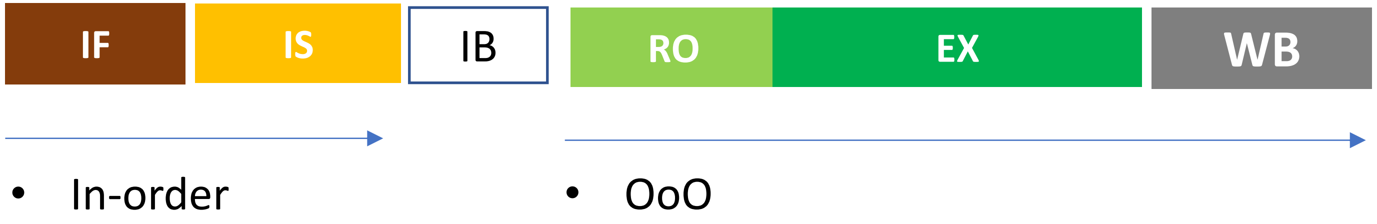
➢Scalar (CPI >= 1)

➢Some stages can be multi-issue, e.g. four WB ports

- In-order/OoO can be different for every stage.

➢But: OoO usually means instructions are scheduled OoO in EX stage.





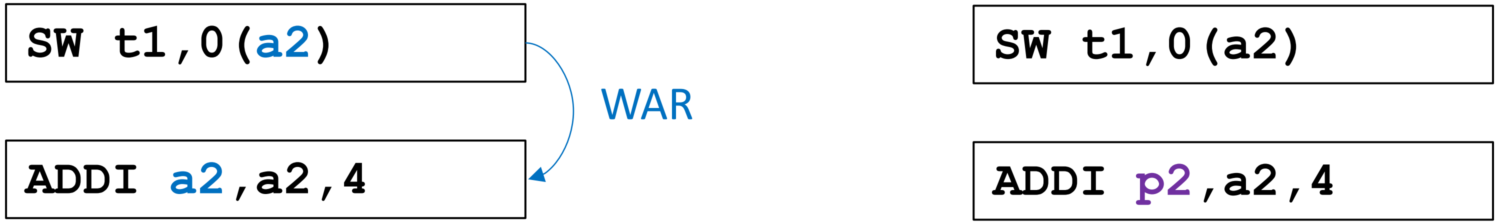
Register Renaming

Out-of-Order Limitations

- WAW and WAR limit further reordering
  - Not real dependencies
  - Artificially added: limitation of registers
- Problem with limited registers
  - Number of registers limited by ISA
  - Compiler optimizations limited
  - Especially with different execution paths
- Approach: CPU solves problem by register renaming

Register Renaming

- Approach: Rename to microarchitecture register names
  - More microarchitecture registers than logical ISA registers
  - Entirely eliminates WAR and WAW hazards
  - Not visible to the outside world



```
SW t1,0(a2)
ADDI a2,a2,4
```

```
SW t1,0(a2)
ADDI p2,a2,4
```

- Introduced by Robert Tomasulo (1967)
  - Reservation stations (FU-specific IBs) before FUs store instructions and reg. names
  - Tomasulo Algorithm: *Computer Architecture A Quantitative Approach 5th Ed. – Chapter 3*

Example: Register Renaming removes WAW, RAW stalls

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`LW x12,8(x9)`	IF	IS	LU	LU	WB									
`LW x13,0(x7)`		IF	IS	LU	LU	WB								
`DIV x17,x13,x12`			IF	IS	IB	DIV	DIV	DIV	DIV	WB				
`ADDI x18,x12,28`				IF	IS	ALU	WB							
`MUL x19,x12,x18`					IF	IS	MUL	MUL	WB					
`MUL x10,x17,x14`						IF	IS	IB	IB	MUL	MUL	WB		
`ADD p10,x10,x13`							IF	IS	IB	IB	IB	ALU	WB	
`SW p10,0(x11)`								IF	IS	IB	IB	IB	SU	SB
`LW p11,4(x8)`									IF	IS	LU	LU	WB	
`ADDI x13,p11,4`										IF	IS	IB	ALU	WB

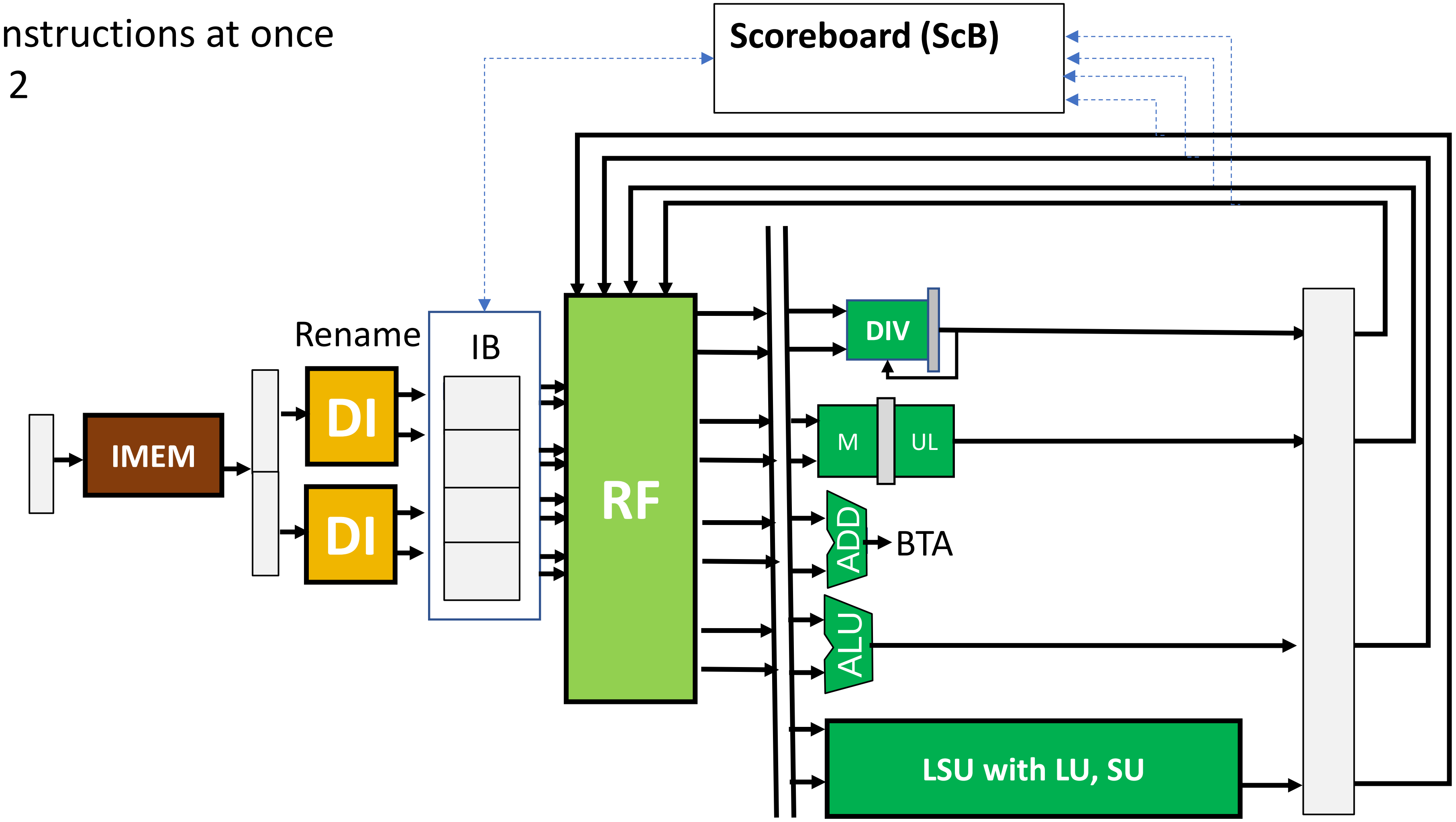
CPI = 1,1

We do not have to stall IF and IS on WAW and WAR, but RAW still makes instruction wait in IB for operands.
In this example the ADD caused 4 stall cycles that are gone now but the RAW still requires it to wait.

BUT: Instructions behind ADD can execute earlier in OoO fashion.

Simple Superscalar (Scoreboard) – Dual Fetch and Decode

Instruction fetch can
fetch two instructions at once
Ideal IPC = 2



Simple Superscalar (Scoreboard) – Dual Instruction Fetch and Decode – Example

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB										
`LW x13,0(x7)`	IF	IS	IB	LU	LU	WB									
`DIV x17,x13,x12`		IF	IS	IB	IB	DIV	DIV	DIV	DIV	WB					
`ADDI x18,x12,28`		IF	IS	ALU	WB										
`MUL x19,x12,x18`			IF	IS	MUL	MUL	WB								
`MUL x10,x17,x14`			IF	IS	IB	IB	IB	IB	IB	MUL	MUL	WB			
`ADD p10,x10,x13`				IF	IS	IB	IB	IB	IB	IB	IB	ALU	WB		
`SW p10,0(x11)`				IF	IS	Stall	IB	IB	IB	IB	IB	IB	SU	SB	
`LW p11,4(x8)`					IF	IS	stall	stall	stall	LU	LU	WB			
`ADDI x13,p11,4`					IF	IS	stall	stall	stall	stall	stall	ALU	WB		
…						IF	stall	stall	stall	stall	stall	IS	…		
…						IF	stall	stall	stall	stall	stall	IS	…		
#INSTR in IB/EX	0	0	2	4	4	4	4	4	4	4	4	3	…		

Due to this chain of RAW dependencies IPC <=1, Otherwise IPC could go above 1

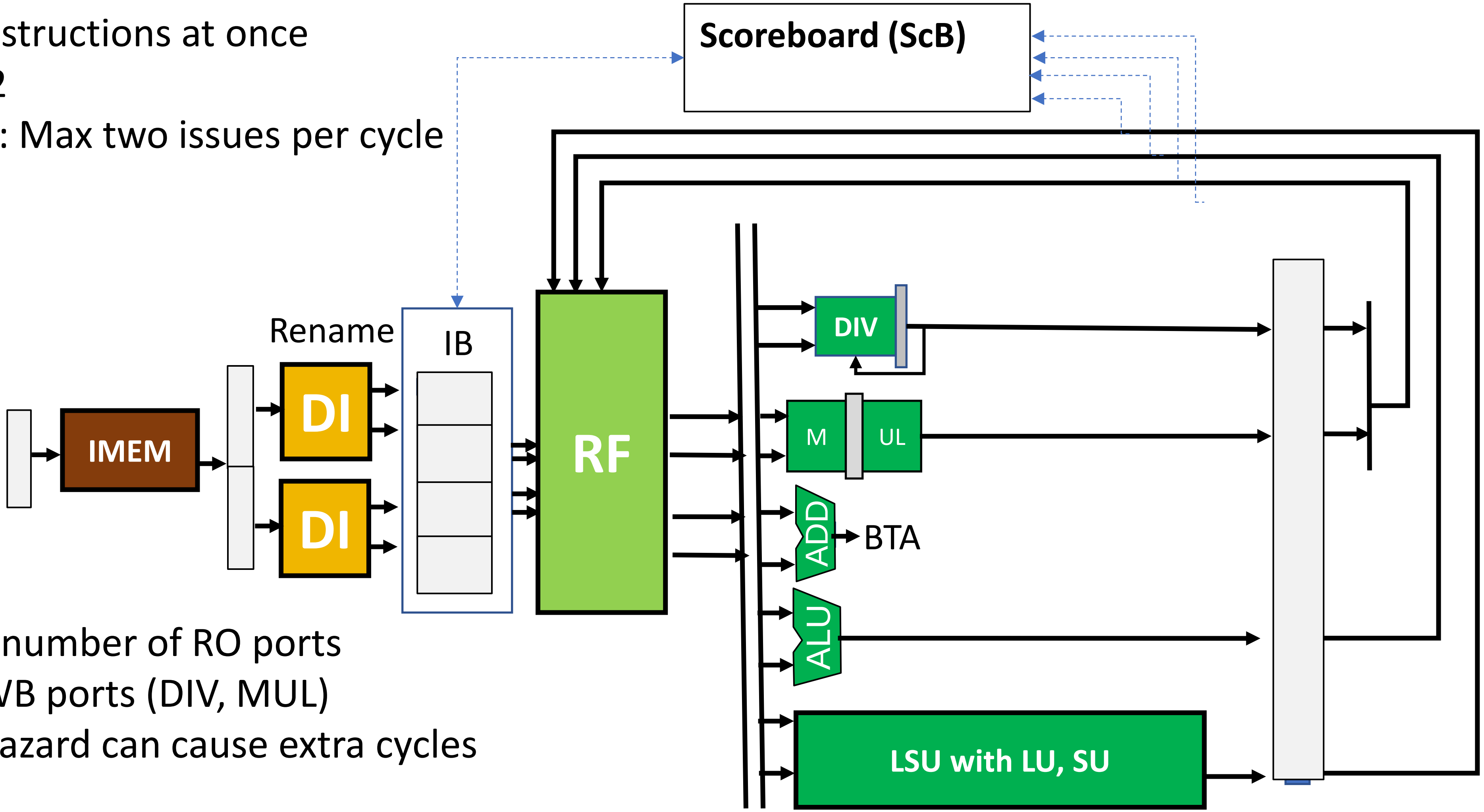
CPI = 1,0

Issue buffer full

Fetching more instructions assures the issue buffer is always filled.
BUT: Instruction Level Parallelism can limit instructions executing in parallel

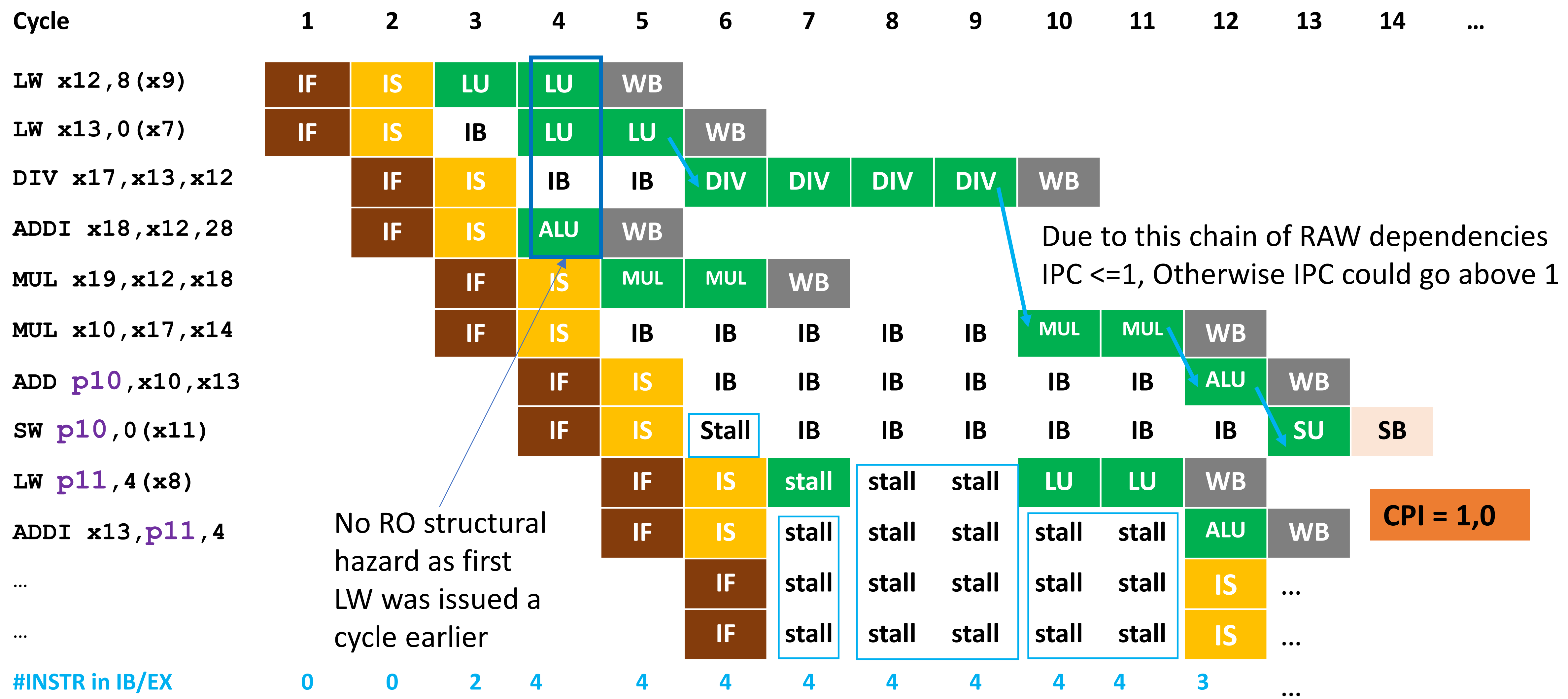
Simple Superscalar (Scoreboard) – Dual Fetch, Decode and Issue

Wide instruction fetch can
fetch two instructions at once
Ideal IPC = 2
Reduce HW: Max two issues per cycle



Reduce the number of RO ports
and share WB ports (DIV, MUL)
Structural hazard can cause extra cycles

Simple Superscalar (Scoreboard) – Dual Instruction Fetch and Decode – Example



Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
`LW x12,8(x9)`	IF	IS	LU	LU	WB										
`LW x13,0(x7)`	IF	IS	IB	LU	LU	WB									
`DIV x17,x13,x12`		IF	IS	IB	IB	DIV	DIV	DIV	DIV	WB					
`ADDI x18,x12,28`		IF	IS	ALU	WB										
`MUL x19,x12,x18`			IF	IS	MUL	MUL	WB								
`MUL x10,x17,x14`			IF	IS	IB	IB	IB	IB	IB	MUL	MUL	WB			
`ADD p10,x10,x13`				IF	IS	IB	IB	IB	IB	IB	IB	ALU	WB		
`SW p10,0(x11)`				IF	IS	Stall	IB	IB	IB	IB	IB	IB	SU	SB	
`LW p11,4(x8)`					IF	IS	stall	stall	stall	LU	LU	WB			
`ADDI x13,p11,4`					IF	IS	stall	stall	stall	stall	stall	ALU	WB		
…						IF	stall	stall	stall	stall	stall	IS	…		
…						IF	stall	stall	stall	stall	stall	IS	…		
#INSTR in IB/EX	0	0	2	4	4	4	4	4	4	4	4	3	…		

Due to this chain of RAW dependencies IPC <=1, Otherwise IPC could go above 1

No RO structural hazard as first LW was issued a cycle earlier

CPI = 1,0

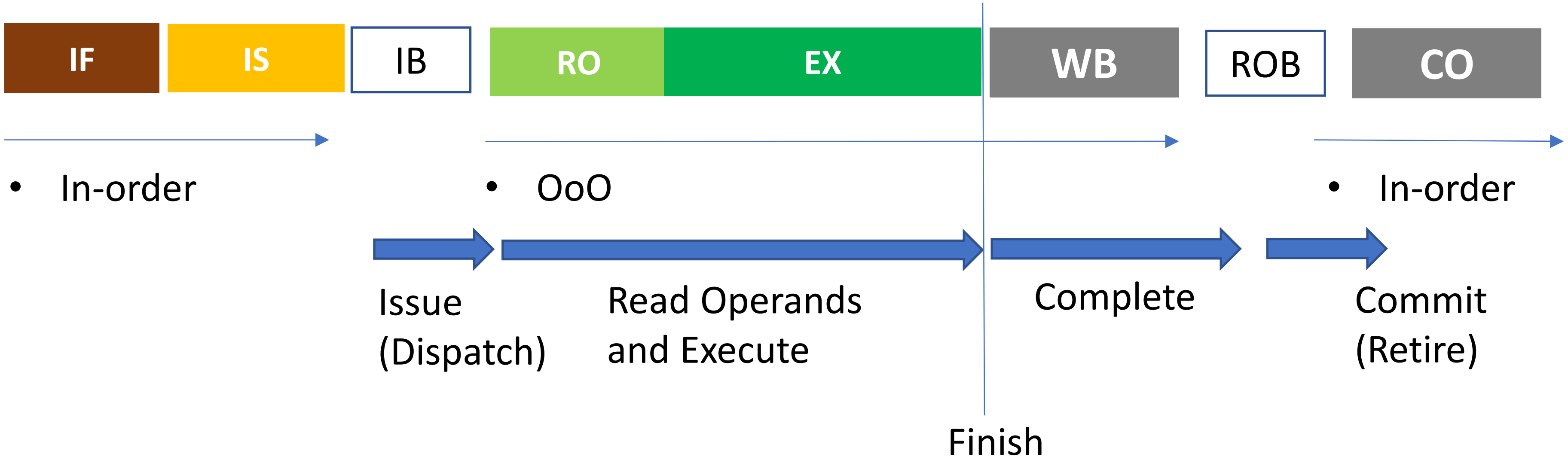
Issue buffer full

We still observe no structural hazards. ILP limits instruction issue below 2.

Reorder Buffer (ROB)

Reorder Buffer (ROB)

- Reorder buffer: Orders the WBs and commits them in-order
- Also assures stores are committed in order with WBs (needed for precise exceptions)

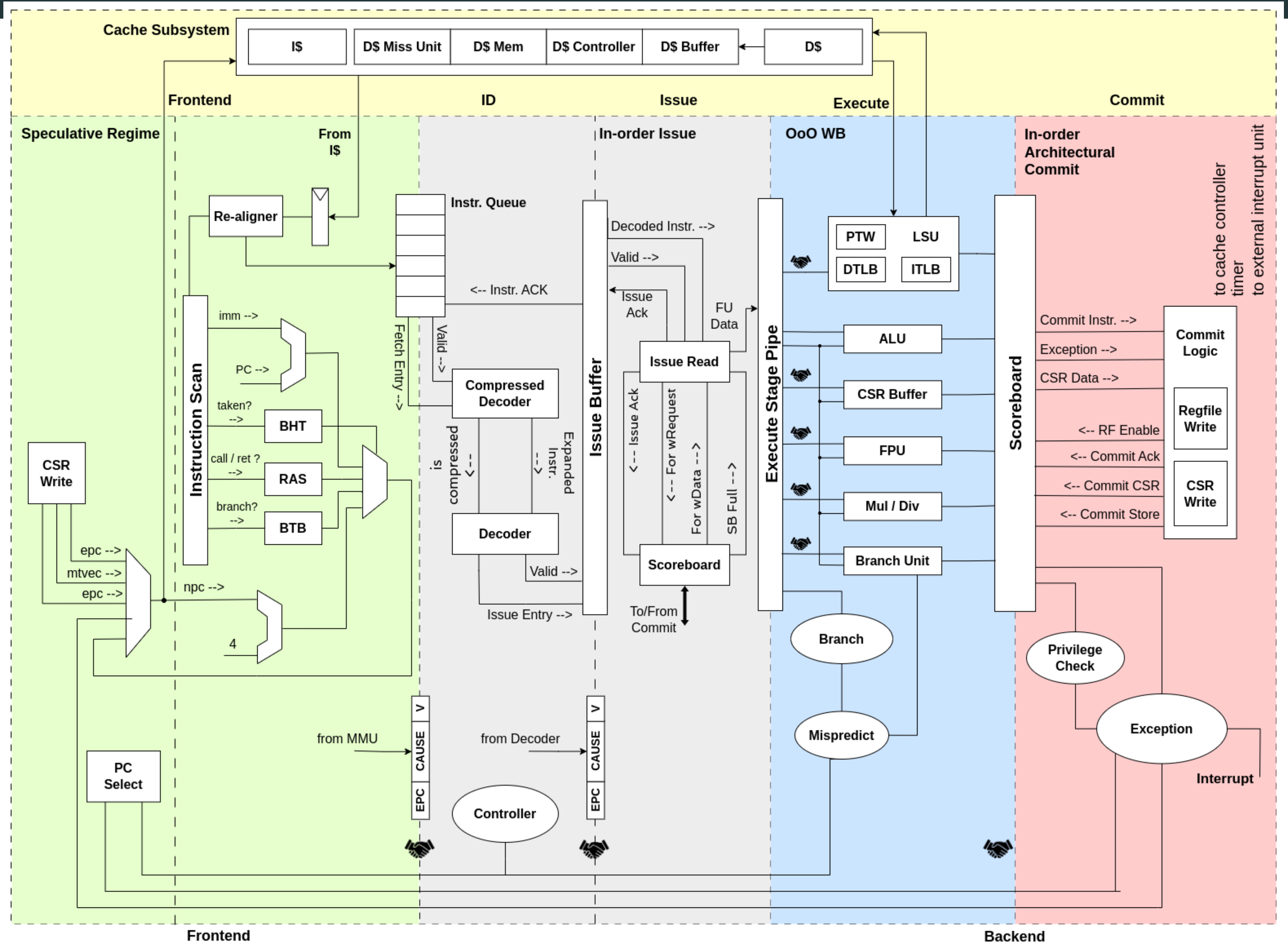


Optional, not relevant for exam

A Look at a Real Processor

CVA6

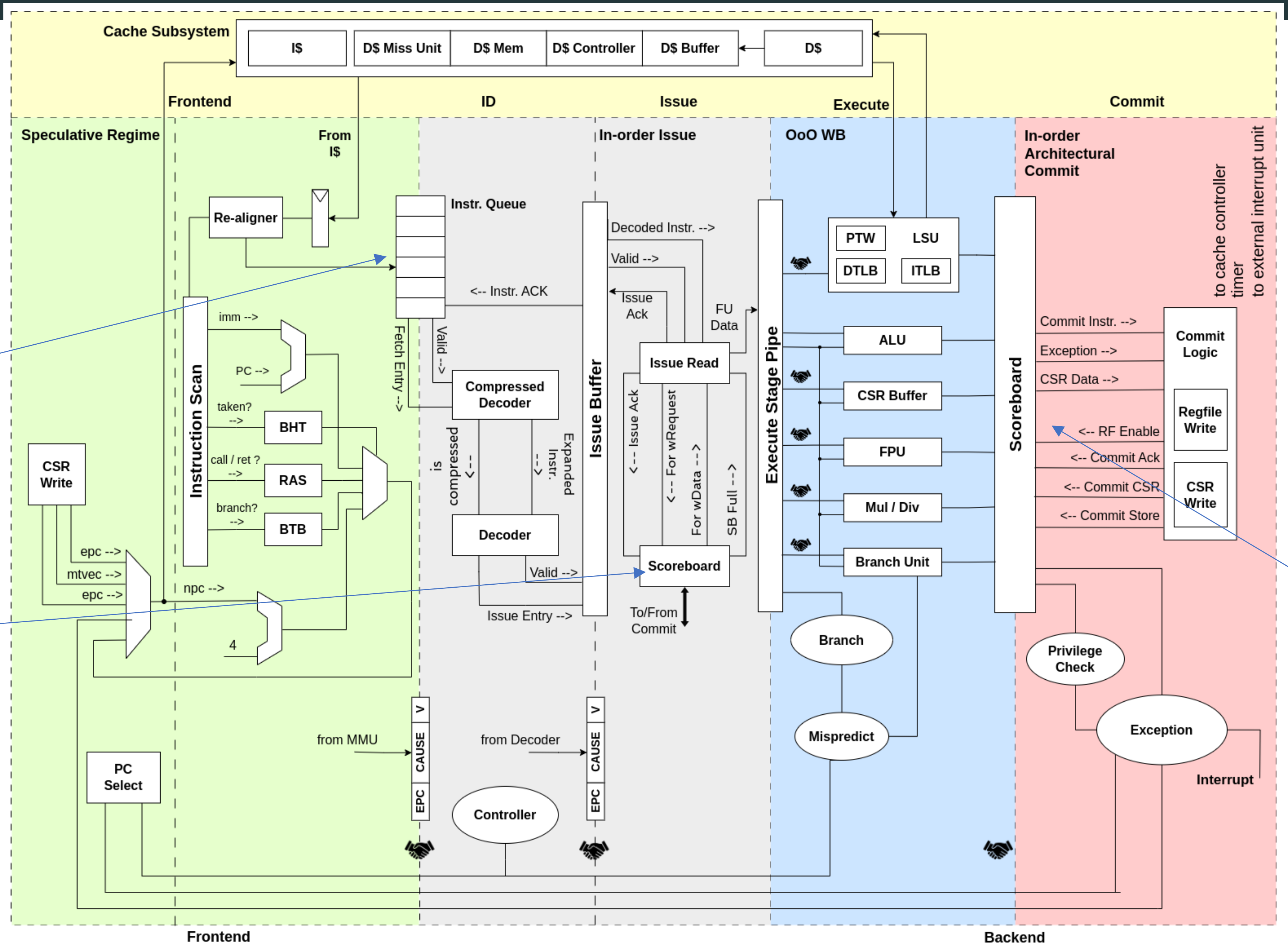
CVA6 Pipeline Diagram: https://github.com/openhwgroup/cva6



CVA6 Pipeline Diagram: https://github.com/openhwgroup/cva6

Fetch buffer between IF and ID

Scoreboard



In-order commit. Sorts the OoO WB

Optional, not relevant for exam

Pipeline Support for Precise Traps

Challenge with OoO Pipelines and Exceptions

- Some instructions can cause exceptions
  - Memory fault on load/store
  - Before entering exception handling all previous instructions should have committed (done their write back)
  - No instruction after the one that caused the exception should have committed (done their write back)

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10
`SW t1,0(a0)`	IF	IS	AC	SB	SB	SB	MSA			
`SW t2,0(a1)`		IF	IS	AC	SB	SB	SB	FAULT		
`LI t2,4`			IF	IS	ALU	WB				

LI would have committed before we observe the memory store fault exception (imprecise exception)

Implementing Precise Exceptions in OoO Pipelines

- For Precise Exception:
  - Before entering exception handling all previous instructions should have committed
  - All previous stores should have written to memory or SB should continue to write them to memory

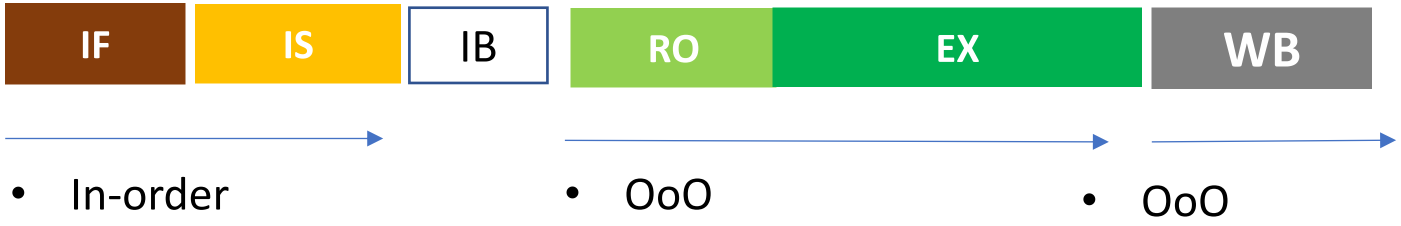
  - No instruction after the instruction that caused the exception should have committed, instead they should be deleted (killed)
  - No store after the instruction that caused the exception should have written to memory from the SB, instead they should be deleted (killed) from the SB

- Scoreboard approach did not support precise exceptions
- Different approaches to implement precise exceptions: e.g. Reorder-Buffer (ROB) sorts all WB commits and makes sure store buffer only sends committed stores to memory

Summary

Where we are

- Four-Stage Superscalar Out-of-order Processor Pipeline
  - Exploit Instruction Level Parallelism to hide extra cycles of multi-cycle FUs.
  - Scoreboard to track instruction dependencies



- Four Stage
- Out-of-order (OoO) pipeline
- Superscalar pipeline (Multi-Issue)

- Upcoming Lecture: More on Multi-Issue Processors (targeting IPC > 1)

Thank you for your attention!