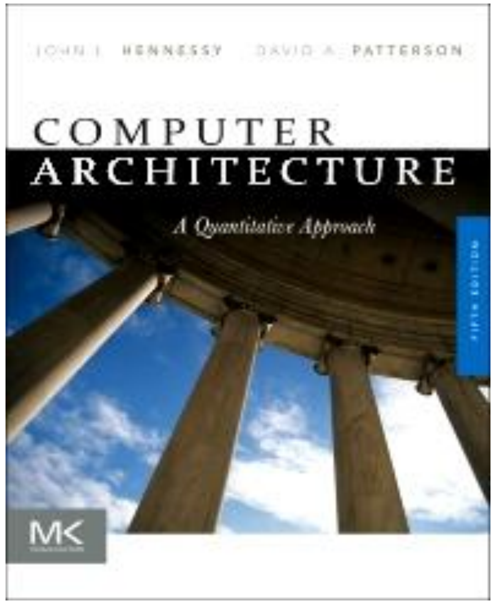
Informatics

Advanced Computer Architecture

C3 – VLIW, Superscalar and Multi-threading

Daniel Mueller-Gritschneder

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.

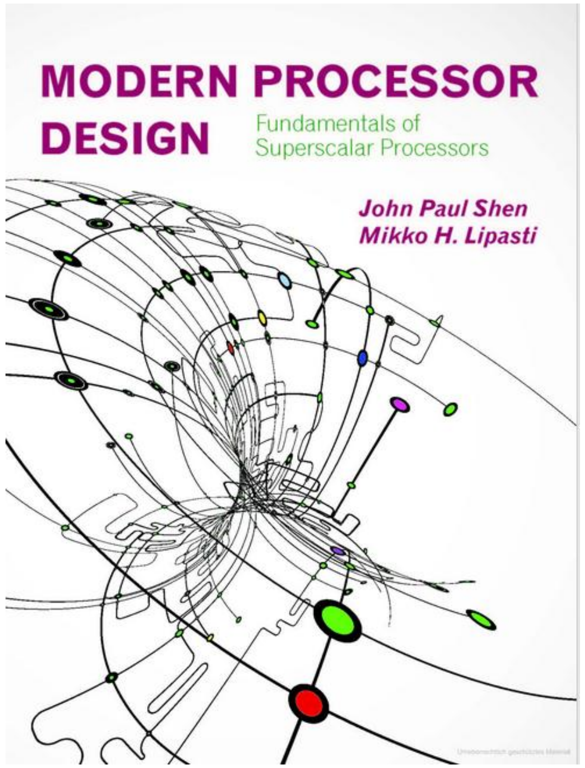
But: We will have a look at some current RISC-V processor designs

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

Sources



Advanced concepts for superscalar.

Literature: **Shen & Lipasti : Modern Processor Design (2005)**

Lecture slides available: https://pharm.ece.wisc.edu/mikko/

Content

- Processors‘ Performance
- Superpipelining
- VLIW
- Superscalar
- HW Multi-threading

Optional, not relevant for exam

- A look at a real RISC-V processor: BOOM, A15

C3-1 Increasing Processors’ Performance

Processors‘ Performance

- Recap of Last lecture: Superscalar processor reached CPI=1

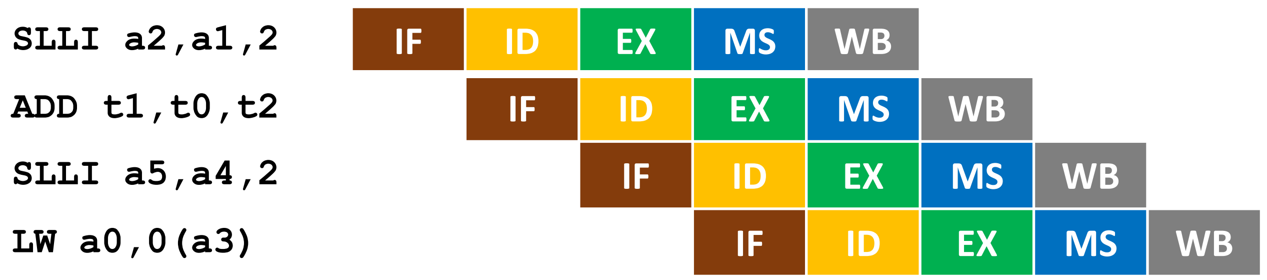
Performance of a processor (IC is instruction count):

$$Performance = \frac{1}{IC} \cdot \frac{Instructions}{Cycle} \cdot \frac{1}{Cycle\ Time} = \frac{IPC \cdot Freq}{IC} = \frac{Freq}{IC \cdot CPI}$$

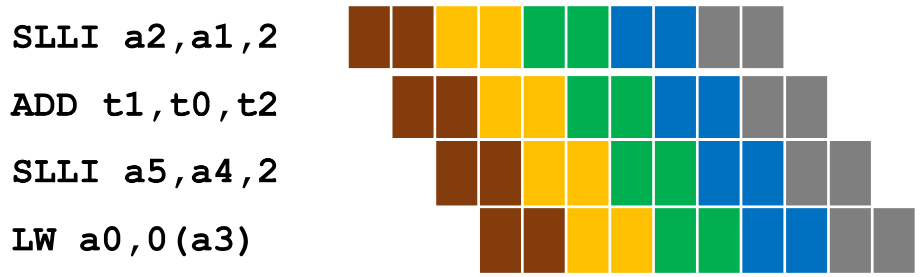
- Superpipelining aims at increasing performance via frequency
- Superscalar, VLIW aims at increasing performance via IPC
- Compiler optimization can improve instruction count (IC) and IPC

Superpipelining and Multi-Issue

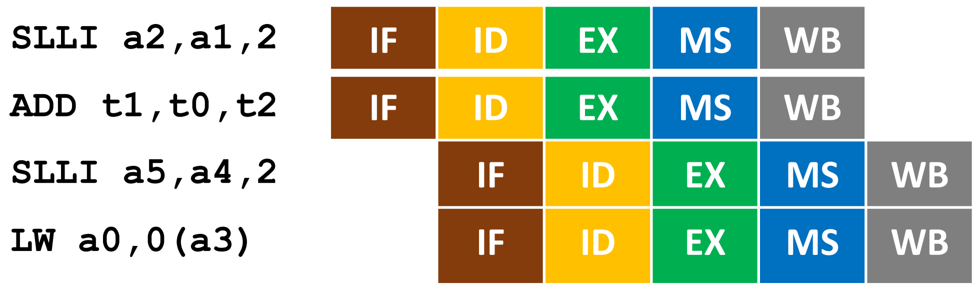
- Scalar five-stage pipeline



- Superpipelining concept:



Multi-Issue concept:

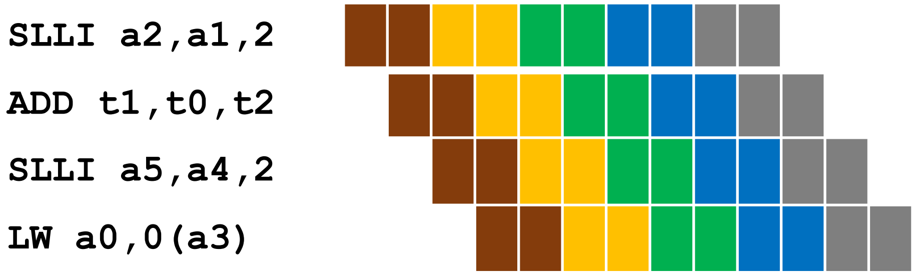


- **Superpipelining** aims at higher clock frequency by increasing number of pipeline stages!
- **Multi-Issue processors** enable CPI < 1 (IPC > 1) by fetching, decoding and executing multiple instructions in parallel

C3-2 Superpipelining

Superpipelining

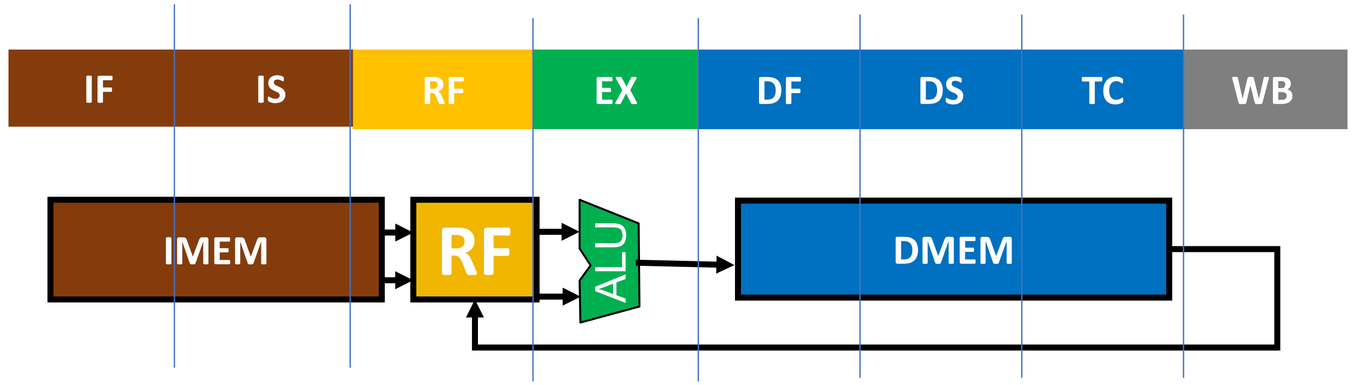
- **Superpipelining** aims to reduce cycle time (increase clock frequency)
- Deep pipelining or superpipelining: Having more stages than a given baseline (e.g. five-stage pipeline)



- Pipeline stages do not need to be split evenly

Example: MIPS R4000

- Example MIPS R4000 Pipeline*
  - Cache access time most critical in the design
  - Eight stages (registers not shown -> lines for cycle boundaries)

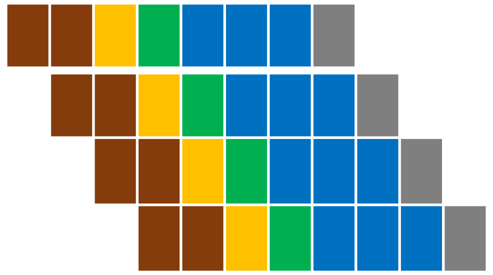


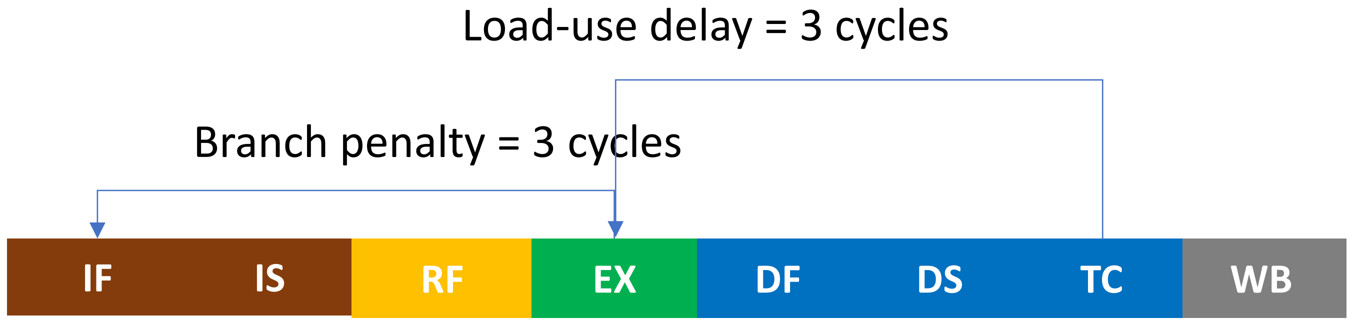
- **IF** — First half of instruction fetch;
- **IS** — Second half of instruction fetch, complete instruction cache access.
- **RF** — Instruction decode and register fetch
- **EX** — Execution, which includes effective address calculation, ALU operation, and branch-target computation and condition evaluation.
- **DF** — Data fetch, first half of data cache access.
- **DS** — Second half of data fetch, completion of data cache access.
- **TC** — Tag check, to determine whether the data cache access hit.
- **WB** — Write-back

-- *diagram according to Computer Architecture A Quantitative Approach – Section C6*

Example: MIPS R4000

- Execution Scheme





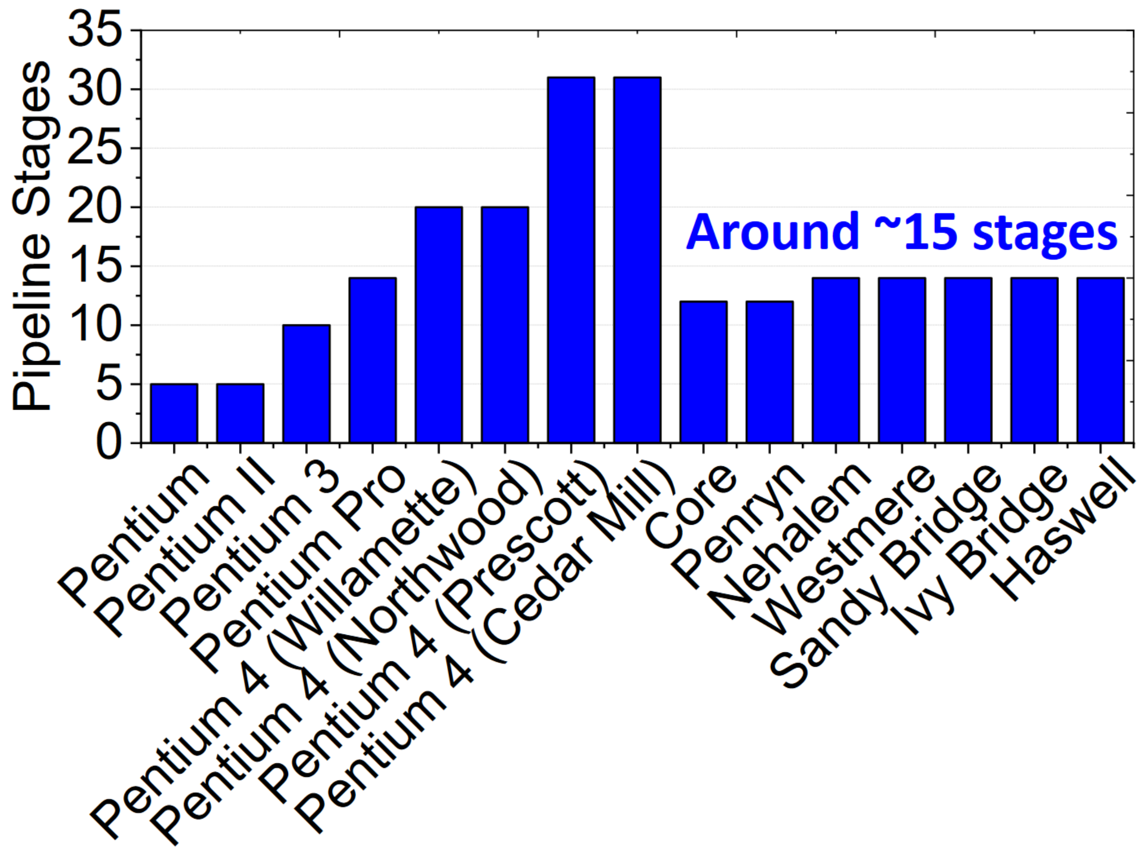
- Instruction dependences have higher penalties (due to deeper pipeline)
  - Branch decision later available -> prediction even more important as more instructions must be flushed (In MIPS R4000: branch computed in EX stage -> 3 cycles branch penalty)
  - Forwarding can‘t remove all stall cycles for RAW dependencies (e.g. Load-use data needs three cycles to become available).

Limits of Superpipelining

- Number of pipeline stages:
  Desktop CPUs: 12-20 stages.
- Embedded CPUs: all from 1-20 stages.

➢ Original Source “Runtime Aware Architectures”, Mateo Valero, HiPEAC CSW 2014,
taken from Lecture Myoungsoo Jung (Slide 6):
http://camelab.org/uploads/Main/lecture06-istruction-paralllel-processing.pdf

➢ See for example
https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures
for a list of the number of pipeline stages for recent Intel‘s processors



C3-3 Multi-issue

Static and Dynamic Multi-Issue

- **Static multiple issue** *(at compile time)*
  - Compiler groups instructions to be issued together in a bundle
  - Sorts them into **“issue slots”**
  - Compiler detects and avoids hazards
- **Dynamic multiple issue** *(during execution)*
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - **Speculate on branch outcome,** execute instructions after branch
    - Roll back, if path taken is different
  - **Speculate on store** that precedes load does not refer to same address
    - We can execute the load instruction before the store instruction
    - Roll back, if the store writes the same address the load reads from

Compiler or Hardware Speculation

- Compiler can **reorder** instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess

- Hardware can **look ahead** for instructions to execute
  - Buffer results until it determines they are actually needed (written to the registers or memory)
  - Flush buffers on incorrect speculation

C3-4 Very Long Instruction Word (VLIW)

Static multi-issue

Static Multiple Issue

- Compiler groups instructions into “issue packets” (sometimes also called bundles)
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required

- **Think of an issue packet as a very long instruction**
  - Specifies multiple concurrent operations
  - $\Rightarrow$ **Very Long Instruction Word (VLIW)**

Scheduling Static Multiple Issue

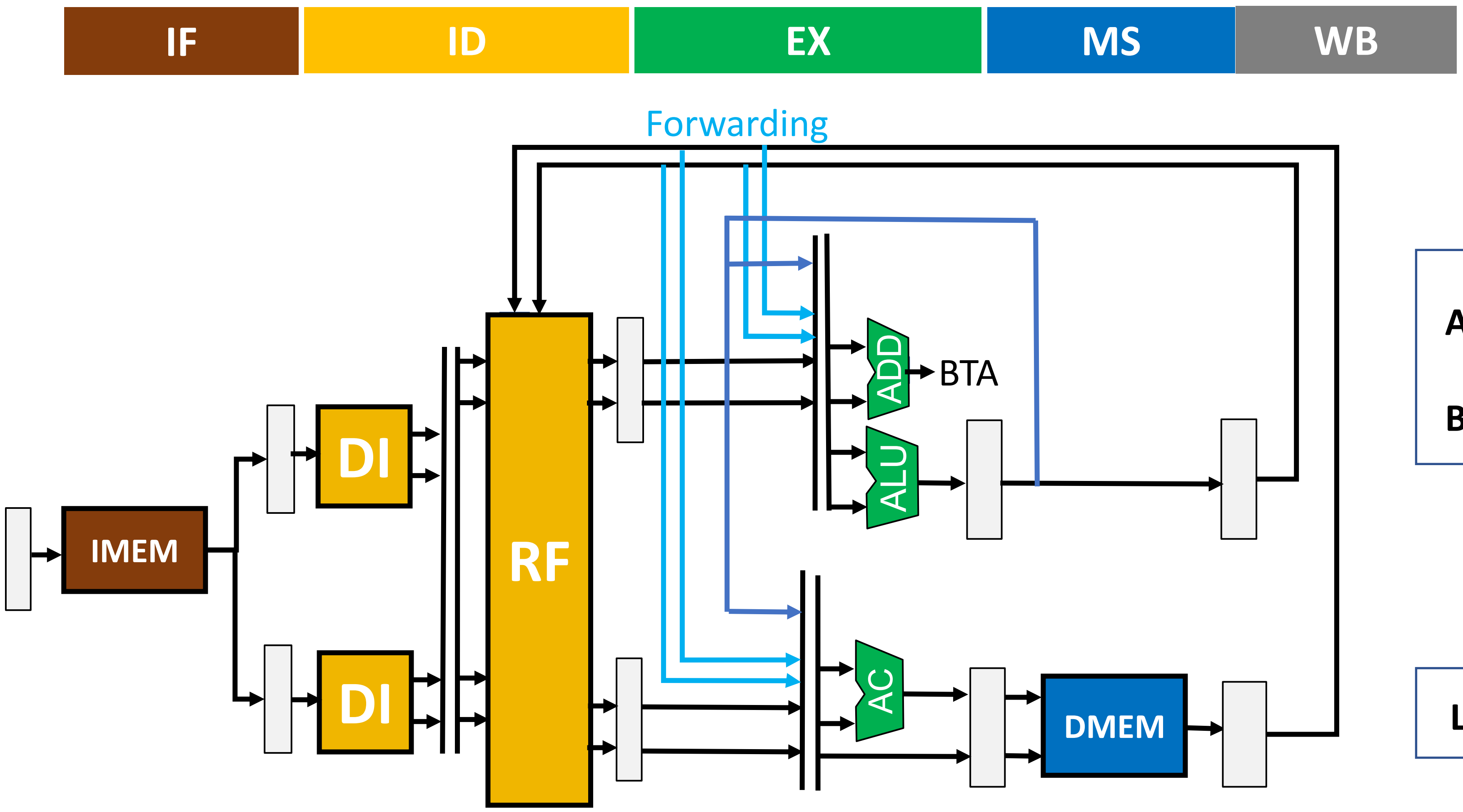
- Compiler must remove some/all hazards
  - **Reorder** instructions into issue packets
  - No dependencies within a packet

  - BUT: If we know the pipeline structure, we can allow WAR dependencies if read operand happens for all instructions in a packet before write back. WAW and RAW dependencies within a packet must still be avoided.

  - All dependencies between packets must be considered in the pipeline
  - Pad with **nop** if necessary

Example: Pipeline with Static Dual Issue

- We fetch and decode two instructions: One instructions is executed on slot 1 the other on slot 2 (Each way can execute certain instruction types)



Slot 1

ALU

Branch Comp.

Slot 2

Load/Store

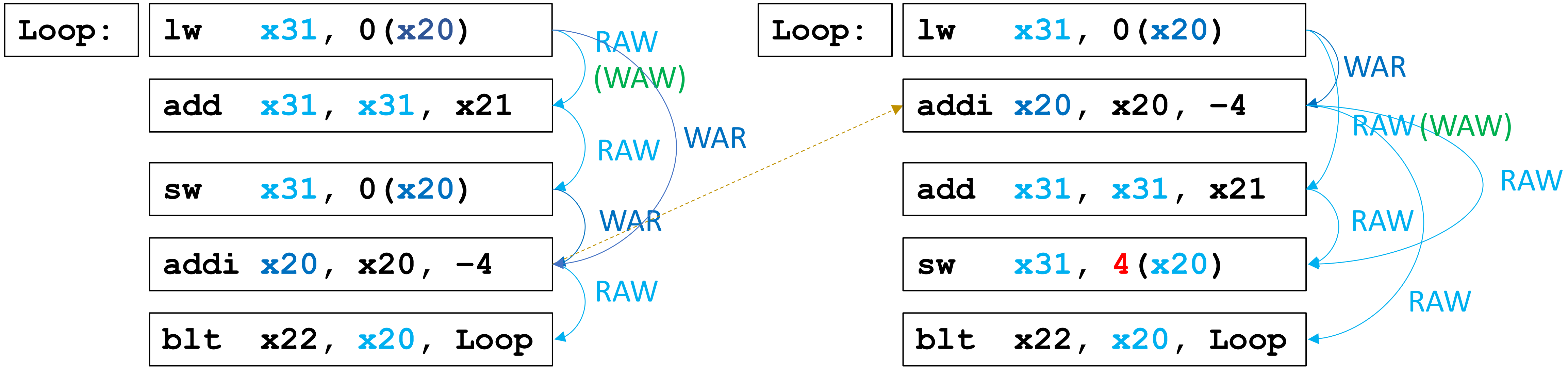
Hazards in the Dual-Issue RISC-V

- More instructions executing in parallel
- **RAW data hazard**
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet

    -
      ```
      add x10, x0, x1
      lw  x2, 0(x10)
      ```
    - Split into two packets, effectively a stall
- **Load-use hazard**
  - Still one cycle use latency, but now two instructions
- **More aggressive scheduling required**

Dependency Analysis

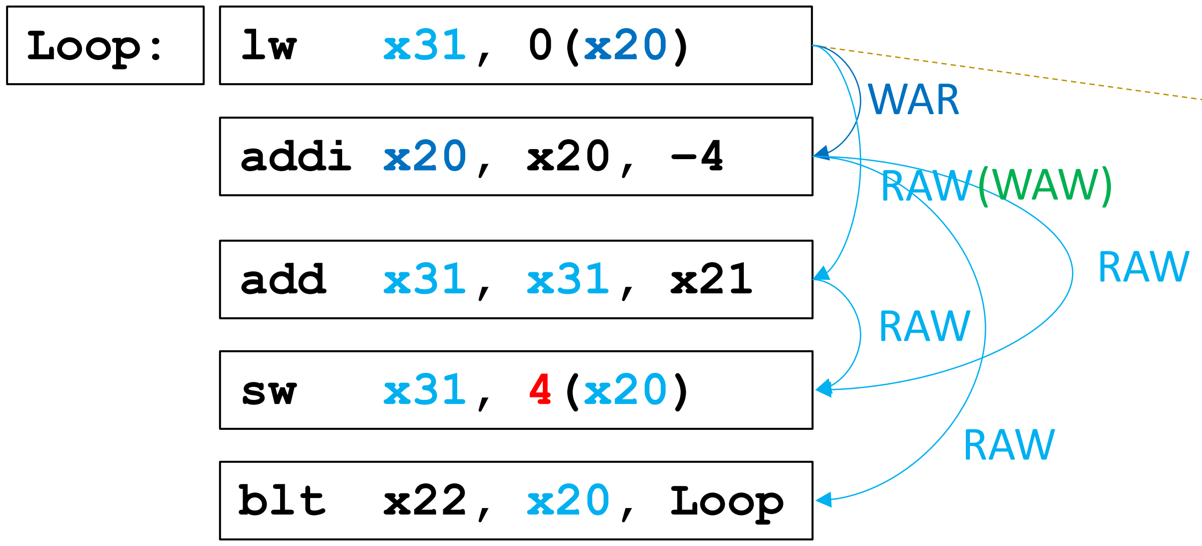
```
Loop: lw   x31, 0(x20)      # x31=array element
      add  x31, x31, x21    # add scalar in x21
      sw   x31, 0(x20)      # store result
      addi x20, x20, -4     # decrement pointer
      blt  x22, x20, Loop   # branch if x22 < x20
```



Compiler can reorder instructions, but needs to adopt the **offset** of the **sw**

Scheduling Example

- **Schedule this for dual-issue RISC-V**

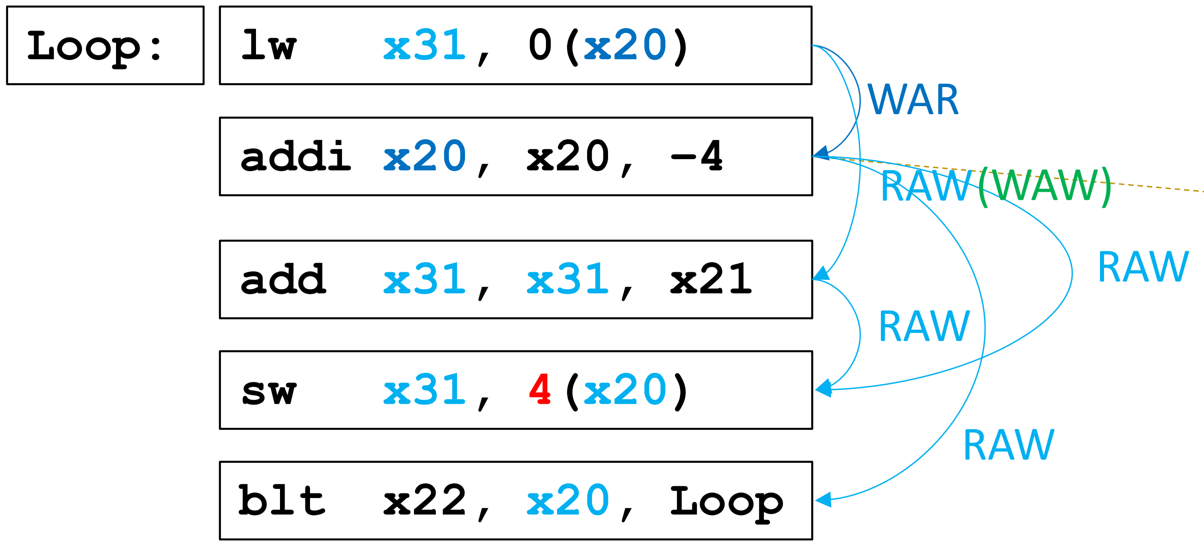


```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

	Slot1 : ALU/BRANCH	Slot 2: Load/store
Loop:		lw x31, 0(x20)

Scheduling Example

- **Schedule this for dual-issue RISC-V**



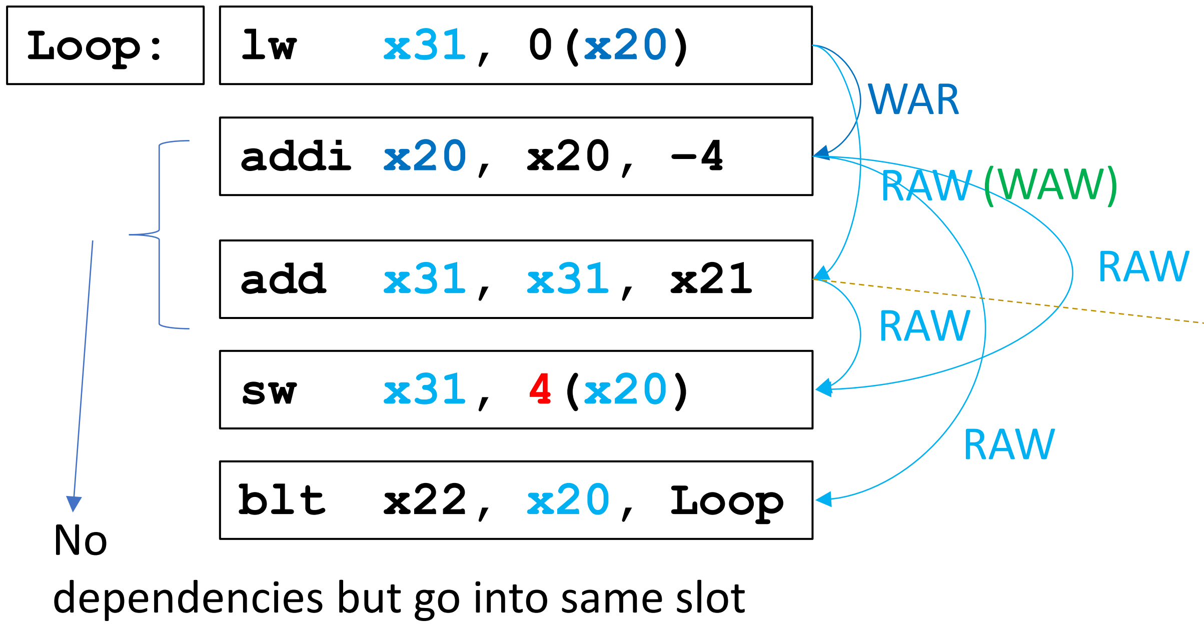
```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

WAR hazard to lw, can be allowed due to known pipeline structure

	Slot1 : ALU/BRANCH	**Slot 2: Load/store**
`Loop:`	`addi x20, x20, -4`	`lw x31, 0(x20)`

Scheduling Example

- **Schedule this for dual-issue RISC-V**



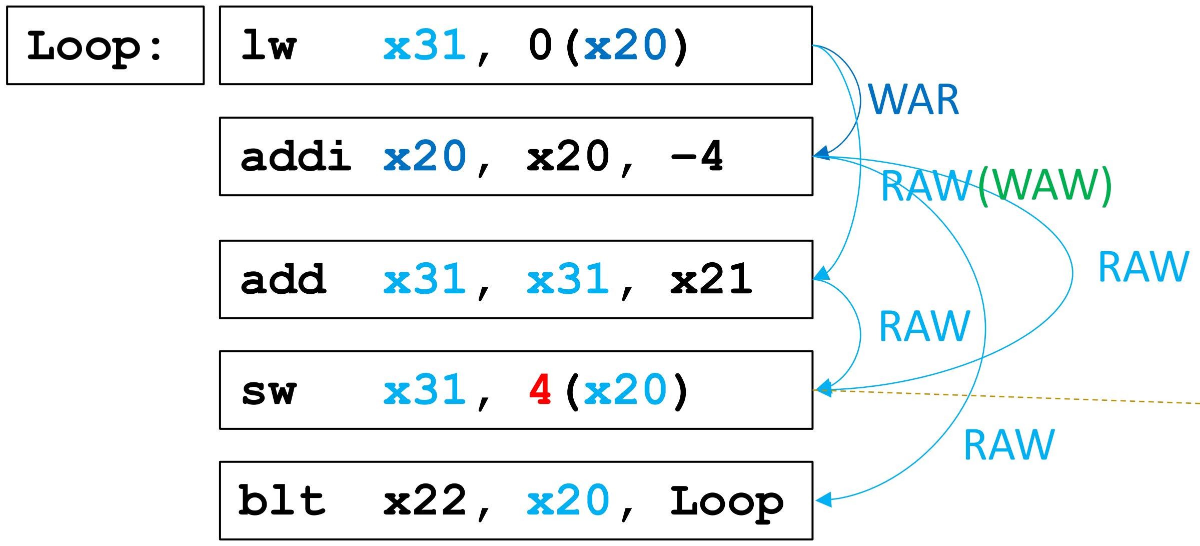
RAW + (WAW) hazard to lw

One cycle load use delay

	Slot1 : ALU/BRANCH	**Slot 2: Load/store**
`Loop:`	`addi x20, x20, -4`	`lw x31, 0(x20)`
	`add x31, x31, x21`	

Scheduling Example

- **Schedule this for dual-issue RISC-V**



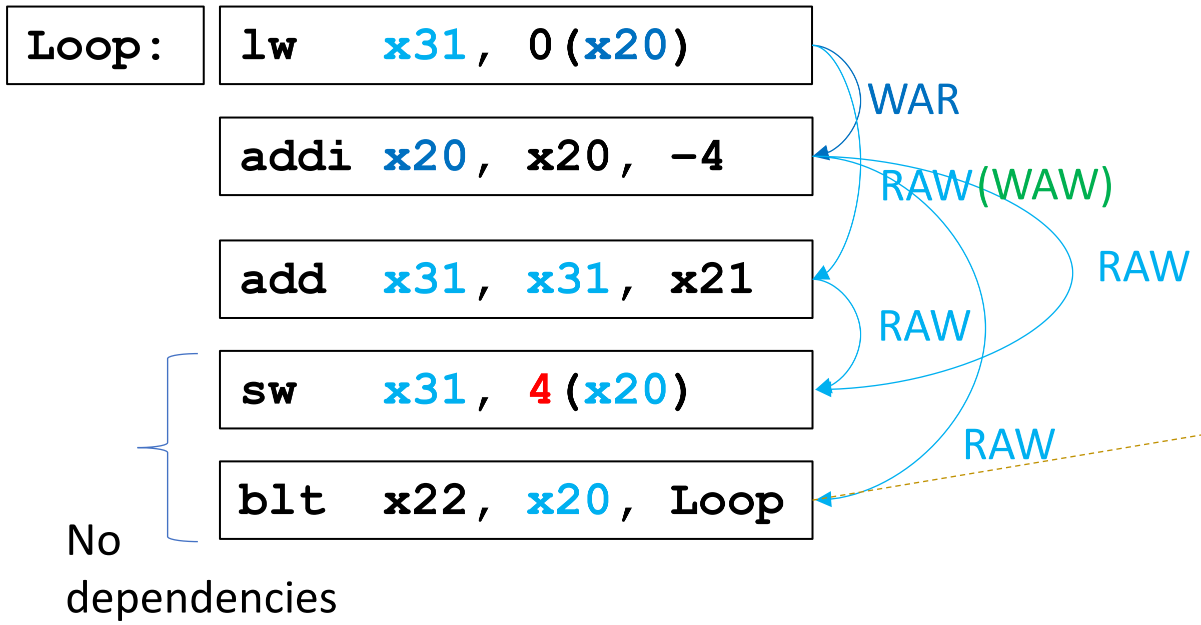
```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

RAW to add

	Slot1 : ALU/BRANCH	Slot 2: Load/store
`Loop:`	`addi x20, x20, -4`	`lw x31, 0(x20)`
	`add x31, x31, x21`	
		`sw x31, 4(x20)`

Scheduling Example

- **Schedule this for dual-issue RISC-V**



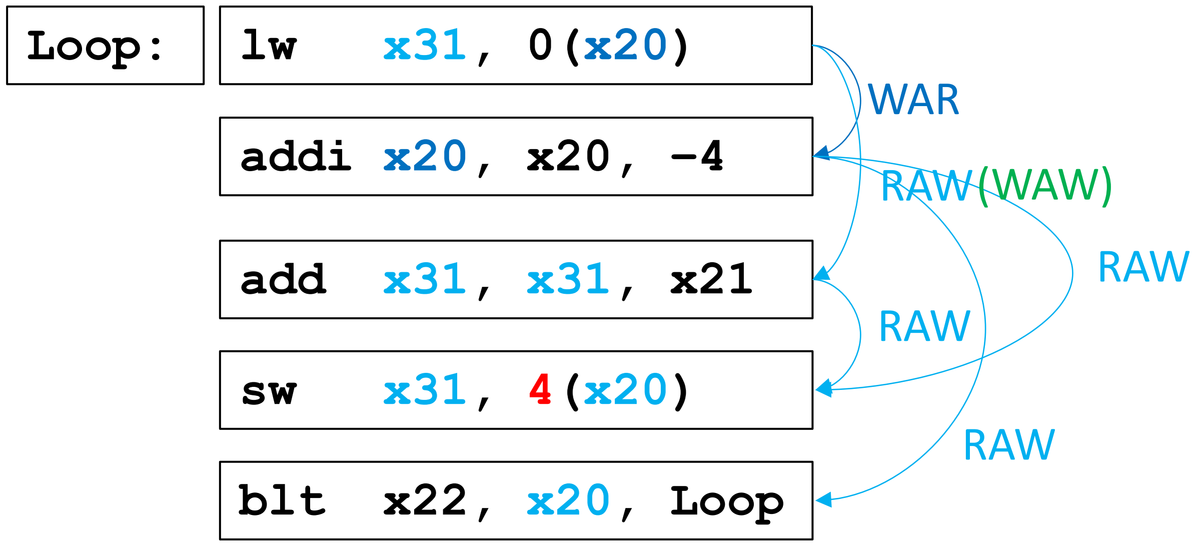
```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

Branches cannot be moved forward, needed to end basic block

	Slot1 : ALU/BRANCH	Slot 2: Load/store
`Loop:`	`addi x20, x20, -4`	`lw x31, 0(x20)`
	`add x31, x31, x21`	
	`blt x22, x20, Loop`	`sw x31, 4(x20)`

Scheduling Example

- **Schedule this for dual-issue RISC-V**



```
Loop:  lw    x31, 0(x20)
       addi  x20, x20, -4
       add   x31, x31, x21
       sw    x31, 4(x20)
       blt   x22, x20, Loop
```

Fill up with **nop**

	Slot1 : ALU/BRANCH	**Slot 2: Load/store**
`Loop:`	`addi x20, x20, -4`	`lw x31, 0(x20)`
	`nop`	`nop`
	`add x31, x31, x21`	`nop`
	`blt x22, x20, Loop`	`sw x31, 4(x20)`

Example Baseline VLIW Processor with Two Slots – Execution Latencies = 1

- Performance: IPC = 5 instr / 4 cycles = 1.25 (peak IPC = 2)

4 cycles

Slot1 : ALU/BRANCH	Slot 2: Load/store	1	2	3	4	5	6	7	8
`addi x20, x20, -4`		IF	ID	EX		WB			
	`lw x31, 0(x20)`	IF	ID	EX	MS	WB			
`nop`			IF	ID					
	`nop`		IF	ID					
`add x31, x31, x21`				IF	ID	EX		WB	
	`nop`			IF	ID				
`blt x22, x20, Loop`					IF	ID	EX		WB
	`sw x31, 4(x20)`				IF	ID	EX	MS	WB

Compiler Optimization - Loop Unrolling

- Replicate loop body to expose more parallelism
    - Reduces loop-control overhead
- Use different registers per replication
    - Compiler applies “register renaming” to eliminate all data dependencies that are not true data dependencies
    - Avoid loop-carried “anti-dependencies”
        - Store followed by a load of the same register
        - Aka “name dependence” - Reuse of a register name
- Unroll factor: Number of loop body replications
- Fully unrolled: Number of loop body replications equal to number of iterations

Unrolled Code Example

- Unroll factor = 4:

```
Loop: lw   x31, 0(x20)
      add  x31, x31, x21
      sw   x31, 0(x20)
      addi x20, x20, -4
      blt  x22, x20, Loop
```

```
lp:     lw      x28,0(x20)      # x28=array element
        Lw      x29,-4(x20)     # x29=array element
        lw      x30,-8(x20)     # x30=array element
        lw      x31,-12(x20)    # x31=array element
        add     x28,x28,x21     # add scalar in x21
        add     x29,x29,x21     # add scalar in x21
        add     x30,x30,x21     # add scalar in x21
        add     x31,x31,x21     # add scalar in x21
        sw      x28,0(x20)      # store result
        sw      x29,-4(x20)     # store result
        sw      x30,-8(x20)     # store result
        sw      x31,-12(x20)    # store result
        addi    x20,x20,-16     # decrement pointer
        blt     x22,x20,lp      # branch if x22 < x20
```

Loop Unrolling Example - – Optimized Code for VLIW

Optimization:

`lw`, `sw` **offsets** are adapted to move `addi` into first pack.

No load-use RAW data hazards, so no influence on performance

	ALU/branch	Load/store	cycle
`Loop:`	`addi x20, x20, -16`	`lw x28, 0(x20)`	1
	`nop`	`lw x29, 12(x20)`	2
	`add x28, x28, x21`	`lw x30, 8(x20)`	3
	`add x29, x29, x21`	`lw x31, 4(x20)`	4
	`add x30, x30, x21`	`sw x28, 16(x20)`	5
	`add x31, x31, x21`	`sw x29, 12(x20)`	6
	`nop`	`sw x30, 8(x20)`	7
	`blt x22, x20, Loop`	`sw x31, 4(x20)`	8

- **IPC = 14/8 = 1.75**
- Closer to 2, but at cost of registers and code size
- Instruction Count (IC) of loop also reduced, less loop iteration checks

Limits of VLIW

- Branches and Labels break sequential instruction execution (code basic blocks)
- Hard to find sufficient Instruction Level Parallelism in single basic block

➢Compiler Optimization techniques:
  - ➢ Loop unrolling
  - ➢ function inlining: function becomes part of the caller code
  - ➢ SW pipelining: schedules instructions from different iterations together
  - ➢ trace scheduling & superblocks: schedule beyond basic block boundaries

- Code Size Increase (e.g. due to loop unrolling, function inlining)

- Binary Compatibility: If the micro-architecture is changed, VLIW code may not be compatible anymore because it depends on the latencies.

C3-4 Superscalar
Dynamic multi-issue

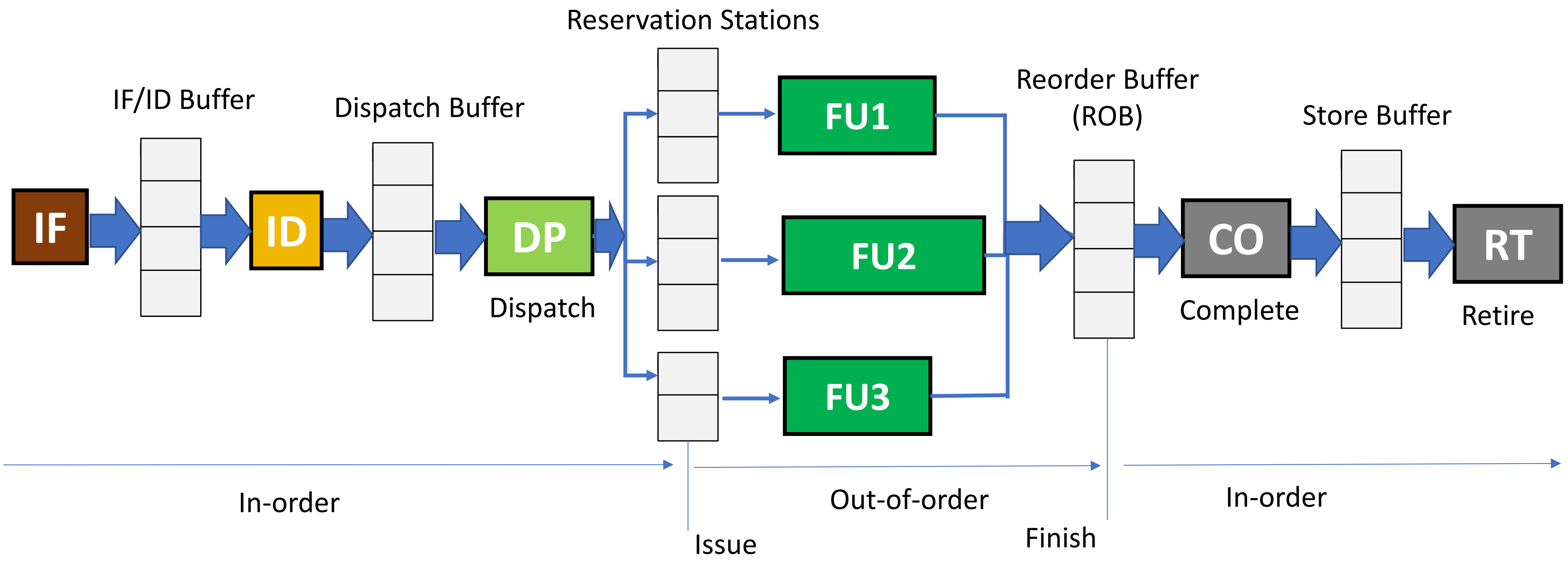
Superscalar

- Exploits Instruction Level Parallelism
- In-order: In order issue but pipeline (not compiler) selects issue bundles
- Out-of-order (OoO): dynamically scheduled

- Phases of instruction execution:
  Fetch – decode – rename – dispatch – issue – execute – complete – commit (retire)

Archetype of a OoO Superscalar Pipeline

- According to *Shen & Lipasti : Modern Processor Design (2005), Fig. 4.20.*



Superscalar vs. VLIW

- Superscalar requires more complex hardware for instruction scheduling

➢ issue buffers for OoO execution

➢ complicated multiplexing between instruction issue structure & functional units

➢ dependence checking logic between parallel instructions

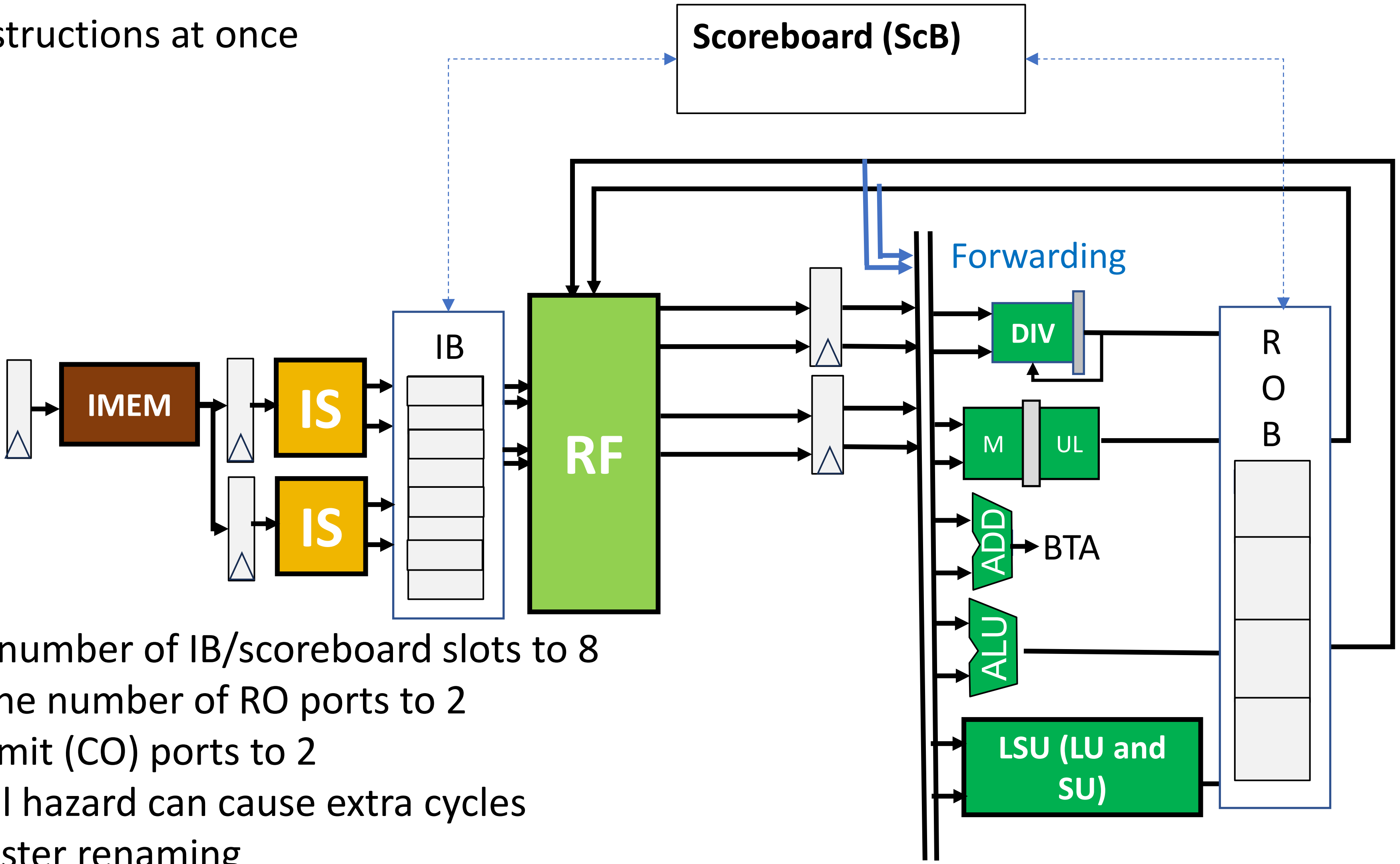
➢ functional unit hazard checking

➢ VLIW requires a complex compiler and higher code size (e.g. slower due to less efficient use of instruction cache)

➢ Superscalars can execute pipeline-dependent code more efficiently : don't have to recompile if binary is executed on different processors (pre-compiled libraries)

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

Wide instruction fetch can
fetch two instructions at once
Ideal IPC = 2



Change HW:

- Increase number of IB/scoreboard slots to 8
- Reduce the number of RO ports to 2
- and Commit (CO) ports to 2
- Structural hazard can cause extra cycles
- With register renaming

Unrolled Code Example

- Unroll factor = 4:

```
Loop: lw   x31, 0(x20)
      add  x31, x31, x21
      sw   x31, 0(x20)
      addi x20, x20, -4
      blt  x22, x20, Loop
```

```
lp:     lw      x28,0(x20)      # x28=array element
        Lw      x29,-4(x20)     # x29=array element
        lw      x30,-8(x20)     # x30=array element
        lw      x31,-12(x20)    # x31=array element
        add     x28,x28,x21     # add scalar in x21
        add     x29,x29,x21     # add scalar in x21
        add     x30,x30,x21     # add scalar in x21
        add     x31,x31,x21     # add scalar in x21
        sw      x28,0(x20)      # store result
        sw      x29,-4(x20)     # store result
        sw      x30,-8(x20)     # store result
        sw      x31,-12(x20)    # store result
        addi    x20,x20,-16     # decrement pointer
        blt     x22,x20,lp      # branch if x22 < x20
```

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

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
`addi x20,x20,-16`	IF	IS	RO	ALU	WB	CO									
`lw   x28, 0(x20)`	IF	IS	IB	RO	LU	LU	WB	CO							
`lw x29,12(x20)`		IF	IS	IB	RO	LU	LU	WB	CO						
`add x28,x28,x21`		IF	IS	IB	IB	RO	ALU	WB	CO						
`lw x30,8(x20)`			IF	IS	IB	RO	LU	LU	WB	CO					
`add x29,x29,x21`			IF	IS	IB	IB	RO	ALU	WB	CO					
`lw x31,4(x20)`				IF	IS	IB	RO	LU	LU	WB	CO				
`add x30,x30,x21`				IF	IS	IB	IB	RO	ALU	WB	CO				
`sw  x28,16(x20)`					IF	IS	IB	RO	SU	SB	SC				
`add x31,x31,x21`					IF	IS	IB	IB	RO	ALU	WB	CO			
`sw x29,12(x20)`						IF	IS	IB	RO	SU	SB	SC			
`sw  x30,8(x20)`						IF	IS	IB	IB	RO	SU	SB	SC		
`sw  x31,4(x20)`							IF	IS	IB	IB	RO	SU	SB	SC	
`blt x22,x20, Loop`							IF	IS	IB	RO	ADD				
`#instr in IB+RO+EX`	0	0	2	4	5	7	8	8	8	5	3	1			

! Renaming to avoid WAR and WAW hazards is omitted here, but it is assumed no stalls on WAR and WAW!

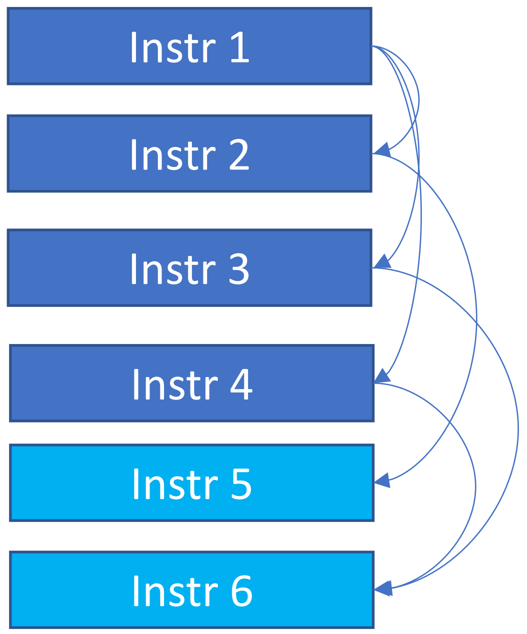
14 instructions

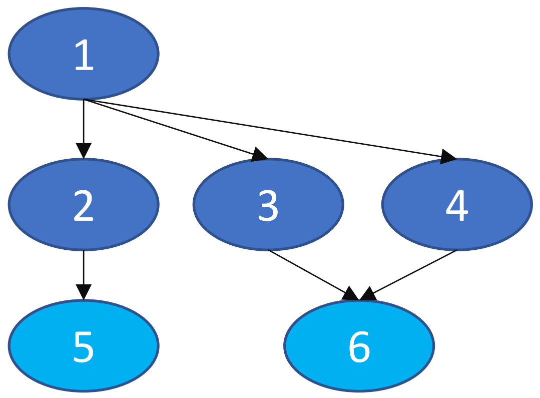
12-5=
7 cycles

CPI = 0,5
IPC=2

Instruction Scheduling for Superscalar

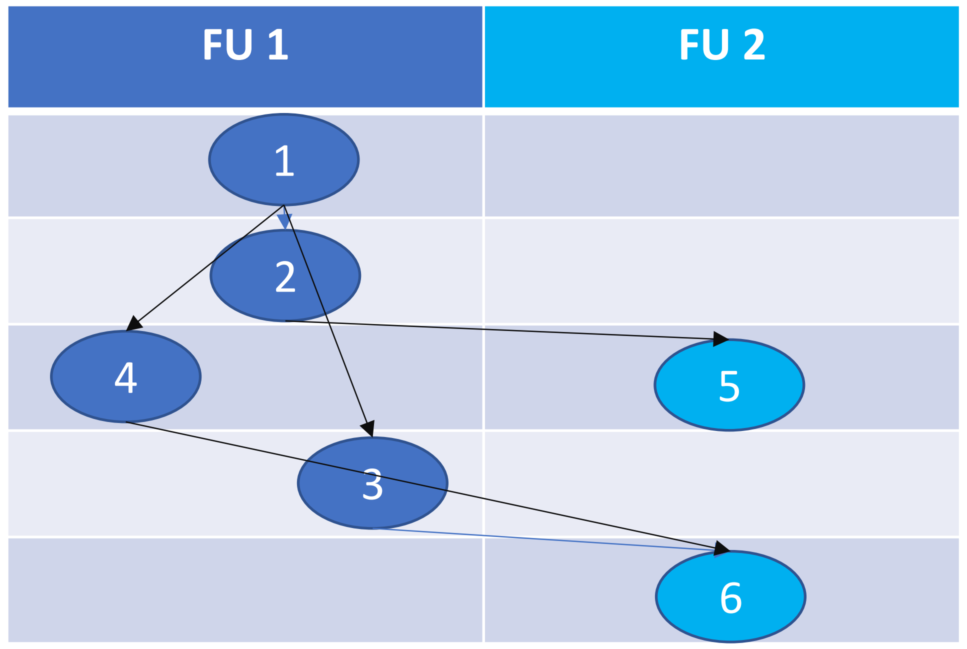
- The process of mapping a series of instructions into execution resources
- Decides when and where an instruction is executed





Dependence graph

1,2,3,4 can execute on FU1
5,6 can execute on FU 2



Derived from CA course of Mikko Lipasti-University of Wisconsin

Instruction Scheduling via Selection and Wakeup

- A set of wakeup and select operations
- **Wakeup**

➢Broadcasts the tags of parent instructions selected

➢Dependent instruction gets matching tags, determines if source operands are ready

➢Resolves RAW data dependencies

- **Select**

➢Picks instructions to issue among a pool of ready instructions

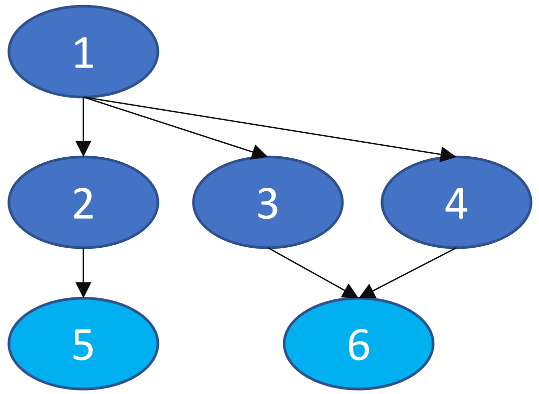
➢Resolves resource conflicts

➢Issue bandwidth

➢ Limited number of functional units / memory ports

Instruction Scheduling via Selection and Wakeup - Example

- Wakeup and Selection Example:



	FU 1	FU 2	Ready to Issue	Select and Wakeup
1	1		1	Select 1 Wakeup 2,3,4
2	2		2,3,4	Select 2 Wakeup 5
3	4	5	3,4,5	Select 4,5 Wakeup -
4	3		3	Select 3 Wakeup 6
5		6	6	Select 6

C3-5 HW Multi-threading

Threads

- **Thread**
  - has state and a current program counter
  - shares the address space of a single process, allowing a thread to easily access data of other threads within the same process.
- **Multithreading:**
  - multiple threads share a processor without requiring an intervening process switch.
  - The ability to switch between threads rapidly is what enables multithreading to be used to hide pipeline and memory latencies.
  - Exploiting **Thread-Level Parallelism (TLP)** to improve uniprocessor throughput (IPC)

Thread-level parallelism (TLP)

- Multithreading (MT) targets to exploit **thread-level parallelism (TLP)**
- MT allows multiple threads to share the FUs of a single processor
- MT does not duplicate the entire processor, duplicating only private state, such as the registers and PC.

- A more general method to exploit TLP is to use a multi-core processor that can execute multiple independent threads in parallel.
- Many recent compute platforms incorporate multi-core processors, for which each single core additionally provides multithreading support.

Example: Use of FUs by Single Thread

Superscalar

Cycle	ALU	MUL	DIV	LU/SU
i+1				🟧
i+2				🟧
i+3				🟧
i+4	🟧		🟧	
i+5		🟧	🟧	
i+6		🟧		
i+7				
i+8		🟧		
i+9		🟧		🟧
i+10	🟧			🟧
i+11	🟧			🟧

Time ↓

Pattern for Superscalar Execution:

- Cycles that a certain instruction of the thread uses a specific FU (EX stage)
- Time now runs from top to bottom.
- We need to rotate the pipeline diagram by 90 deg.

Fine-Grained vs. Coarse-Grained MT

- **Fine-grained multithreading**
  - switches between threads on each clock cycle,
  - execution of instructions from multiple threads to be interleaved. (often round-robin skipping stalled threads)
  - **Advantage**: hide the throughput losses that arise from both short and long stalls because instructions from other threads can be executed when one thread stalls, even if the stall is only for a few cycles.
  - **Disadvantage**: slows down the execution of an individual thread because a thread that is ready to execute without stalls will be delayed by instructions from other threads.

- **Coarse-grained multithreading**
  - switches threads only on costly stalls, such as level two or three cache misses.
  - **Advantage**: less likely to slow down the execution of any one thread
  - **Disadvantage**: it is limited in its ability to overcome throughput losses, especially from shorter stalls.

Simultaneous Multithreading (SMT)

- **Simultaneous multithreading (SMT)**:
  - dynamically scheduled (OoO) processors already have many of the hardware mechanisms needed to support SMT
  - Multithreading can be built on top of an out-of-order processor by adding
    - separate PCs and register files, and
    - the capability for instructions from multiple threads to commit.
  - Instructions from different threads can be issued in same cycle.

Patterns for Types of Multithreading (MT)

Time

Coarse-grained MT

Cycle	ALU	MUL	DIV	LU/SU
i+1				🟧
i+2				🟧
i+3				🟧
i+4	🟧		🟧	
i+5		🟧	🟧	
i+6		🟧		
i+7	🟦			
i+8		🟦		
i+9		🟦		🟦
i+10	🟦		🟦	🟦
i+11			🟦	

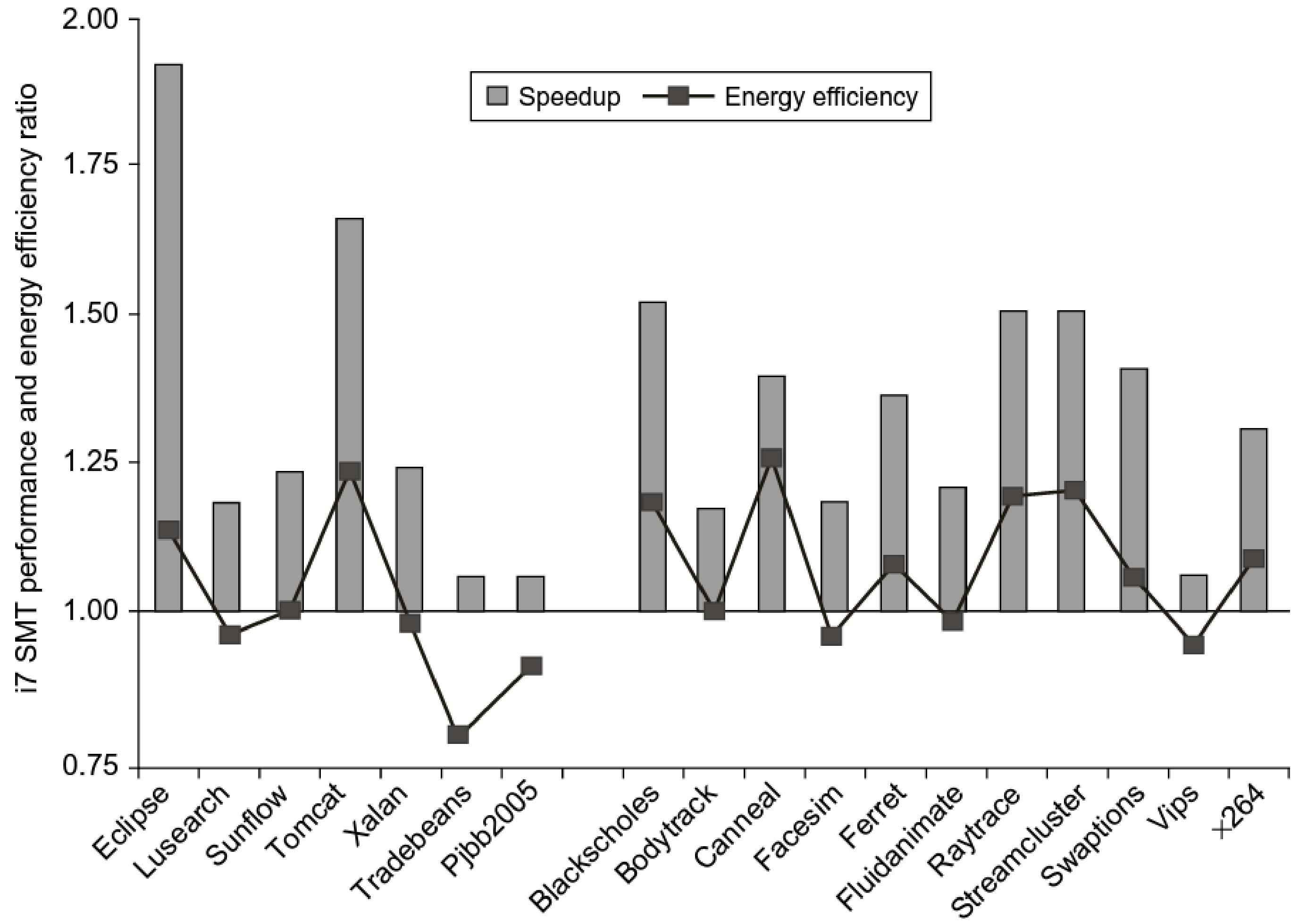
Fine-grained MT

Cycle	ALU	MUL	DIV	LU/SU
i+1		🟧		🟧
i+2	🟦			
i+3			🟧	🟧
i+4	🟦	🟦		
i+5				🟧
i+6		🟦		🟦
i+7	🟧		🟧	
i+8	🟦		🟦	🟦
i+9		🟧	🟧	
i+10			🟦	
i+11		🟧		

Simultaneous MT (SMT)

Cycle	ALU	MUL	DIV	LU/SU
i+1	🟦	🟧		🟧
i+2		🟦		🟧
i+3		🟦		🟧
i+4	🟧			🟦
i+5		🟧		🟦
i+6		🟧		
i+7	🟦			
i+8		🟦		
i+9		🟧		
i+10		🟦		🟦
i+11		🟧		🟧

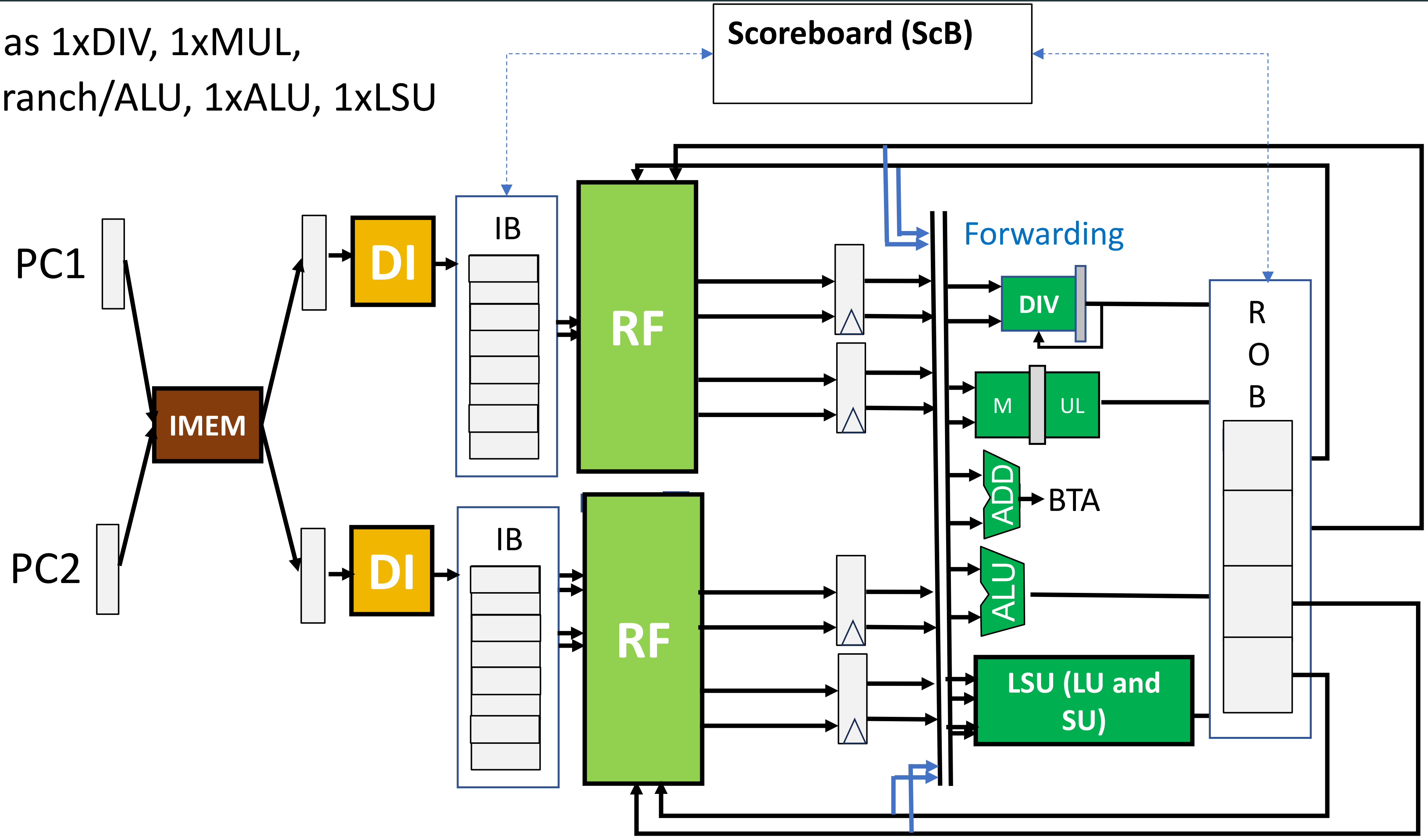
The speedup from using multithreading on one core on an i7 processor



Source: Computer Architecture – A Quantitative Approach 5th Edition Fig. 3.33

Example: Simple Dual Multi-threaded Processor

- EX has 1xDIV, 1xMUL,
- 1x Branch/ALU, 1xALU, 1xLSU



Example with Stall due to D-Cache Miss

```
Loop: lw   x31, 0(x20)
      add  x31, x31, x21
      sw   x31, 0(x20)
      addi x20, x20, -4
      blt  x22, x20, Loop
```

Cycle - i +	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
`lw x31,0(x20)`	IF	IS	RO	LSU	LSU	WB	CO									
`add x31,x31,x21`		IF	IS	RO	IB	ALU	WB	CO								
`sw x31,0(x20)`			IF	IS	RO	IB	SU	SB	SC							
`addi x20,x20,-4`				IF	IS	RO	ALU	WB	CO							
`blt x22,x20,Loop`					IF	IS	RO	BR								
`lw x31,0(x20)`						IF	IS	RO	LSU	Cache miss			LSU	CO		

Example with Stall due to D-Cache Miss

Cycle - i +	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
`lw x31,0(x20)`	IF	IS	RO	LSU	LSU	WB	CO									
`add x31,x31,x21`		IF	IS	RO	IB	ALU	WB	CO								
`sw x31,0(x20)`			IF	IS	RO	IB	SU	SB	SC							
`addi x20,x20,-4`				IF	IS	RO	ALU	WB	CO							
`blt x22,x20,Loop`					IF	IS	RO	BR								
`lw x31,0(x20)`						IF	IS	RO	LSU	Cache miss			LSU	CO		

FU USE - cycle i +	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
ALU						add	addi	blt								
LU				lw (s1)	lw (s2)				lw (s1)				lw (s2)			
SU							sw									

Utilization of functional units in EXE stage is low

Example with Stall due to D-Cache Miss

```
Loop: lw   x31, 0(x20)
      add  x31, x31, x21
      sw   x31, 0(x20)
      addi x20, x20, -4
      blt  x22, x20, Loop
```

Cycles run from top to bottom

Thread 1

Cycle	ALU	LSU	SU
i+3		lw (s1)	
i+4		lw (s2)	
i+5	add		
i+6	addi		sw
i+7	blt		
i+8		lw(s1)	
i+9		cache	
i+10		Miss	
i+11		…	
i+12		lw(s2)	
i+13	add		
i+14	addi		sw

Thread 2

Cycle	ALU	LSU	SU
i+3		lw (s1)	
i+4		lw (s2)	
i+5	add		
i+6	addi		sw
i+7	blt		
i+8		lw (s1)	
i+9		lw (s2)	
i+10	add		
i+11	addi		sw
i+12	blt		

Example with Stall due to D-Cache Miss

Thread 1

Cycle	ALU	LU	SU
i+3		lw (s1)	
i+4		lw (s2)	
i+5	add		
i+6	addi		sw
i+7	blt		
i+8		lw(s1)	
i+9		cache	
i+10		Miss	
i+11		…	
i+12		lw(s2)	
i+13	add		
i+14	addi		sw

Thread 2

Cycle	ALU	LU	SU
i+3		lw (s1)	
i+4		lw (s2)	
i+5	add		
i+6	addi		sw
i+7	blt		
i+8		lw (s1)	
i+9		lw (s2)	
i+10	add		
i+11	addi		sw
i+12	blt		

15 cycles SMT multi-theaded instead of 10 plus 12 cycles

Multithreaded - SMT

Cycle	ALU	LU	SU
i+3		lw (s1)	
i+4		lw (s2)	
i+5	add	lw (s1)	
i+6	addi	lw (s2)	sw
i+7	add		
i+8	addi		sw
i+9	blt		
i+10	blt	lw (s1)	
i+11		lw (s1)	
i+12		lw(s2)	
i+13	add	🟨	
i+14	addi	lw (s2)	sw
i+15	add		
i+16	blt		
i+17	addi		sw

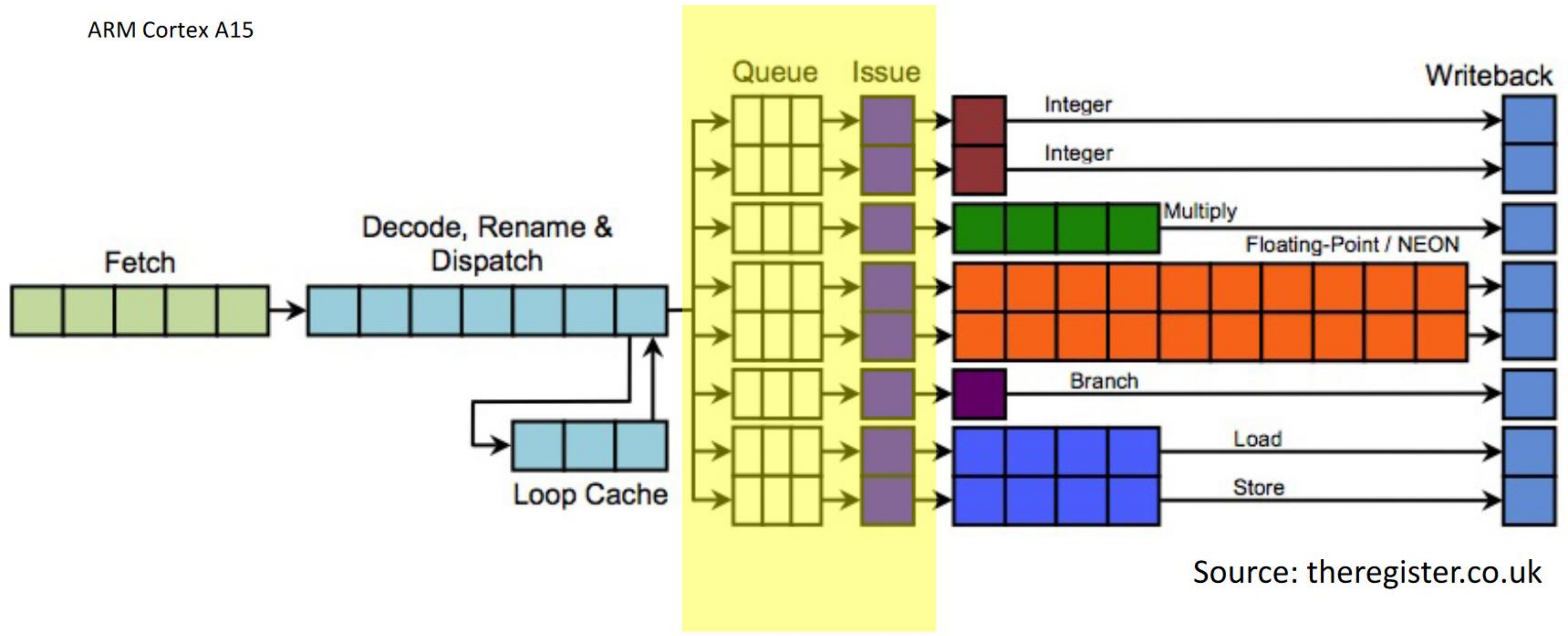
Optional, not relevant for exam

A Look at Real Processors

A15 and BOOM

ARM A15 Superscalar Core

- ARM A15 pipeline diagram:

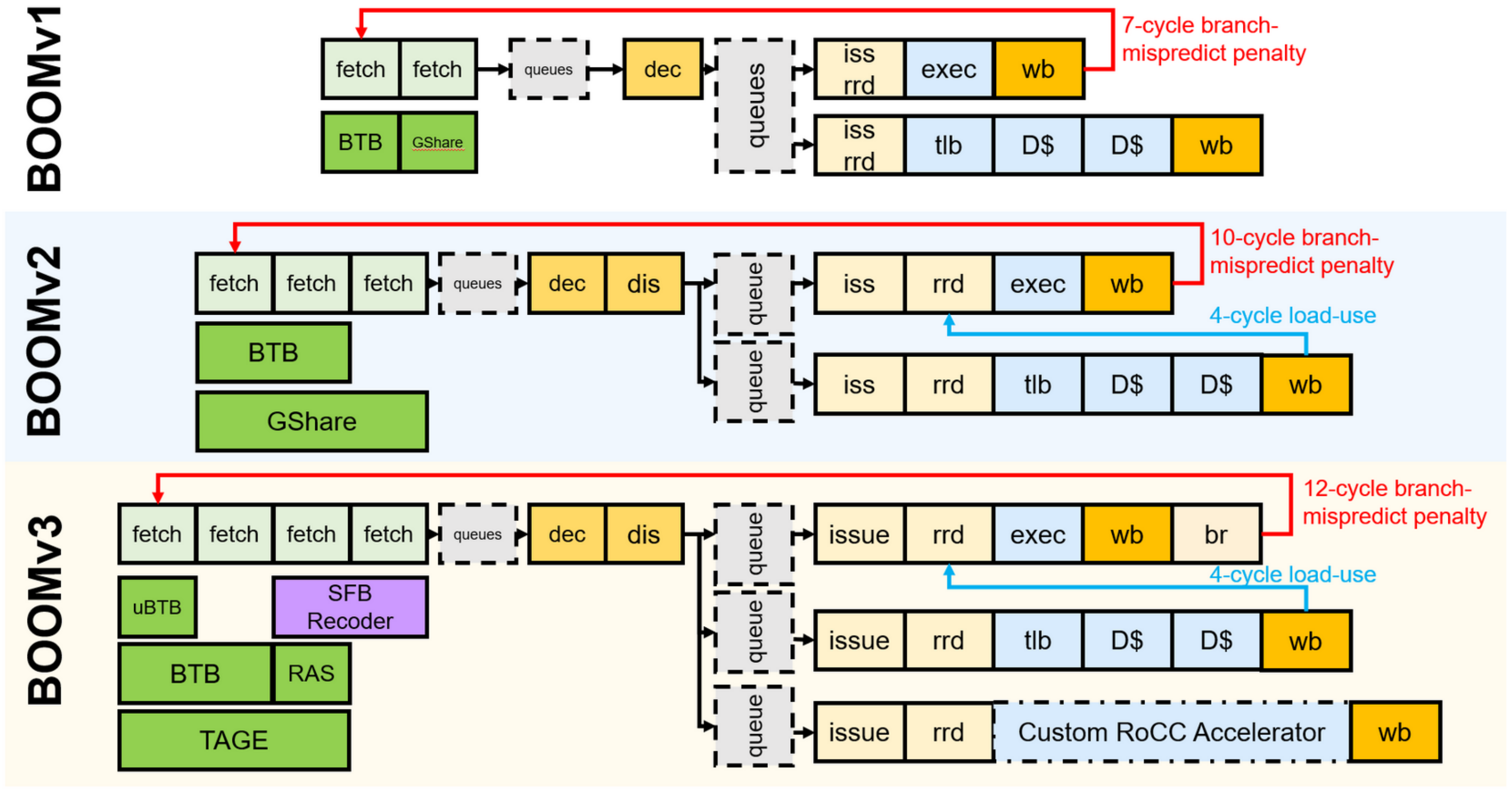


Source: theregister.co.uk

(Copied from from slides of CS course Mikko Lipasti-University of Wisconsin)

Berkeley Out-of-order Machine (BOOM)

- BOOM: an open-source out-of-order RISC-V core



Source: https://github.com/riscv-boom/riscv-boom

Summary

Where we are

- We covered the following features: **Branch prediction, Out of order execute, Scoreboard, Superpipelining, Multi-issue, Superscalar, VLIW, Multi-threading**

- Instruction Level Parallelism: VLIW, Superscalar
- Thread Level Parallelism: Multi-threaded Single Core Processor

- Upcoming:

➢ Thread Level Parallelism: Multi-Core (MIMD)

➢ Data level parallelism: Vector (SIMD)

Thank you for your attention!