Fakultät für
Informatik und
Mathematik

HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN
MÜNCHEN

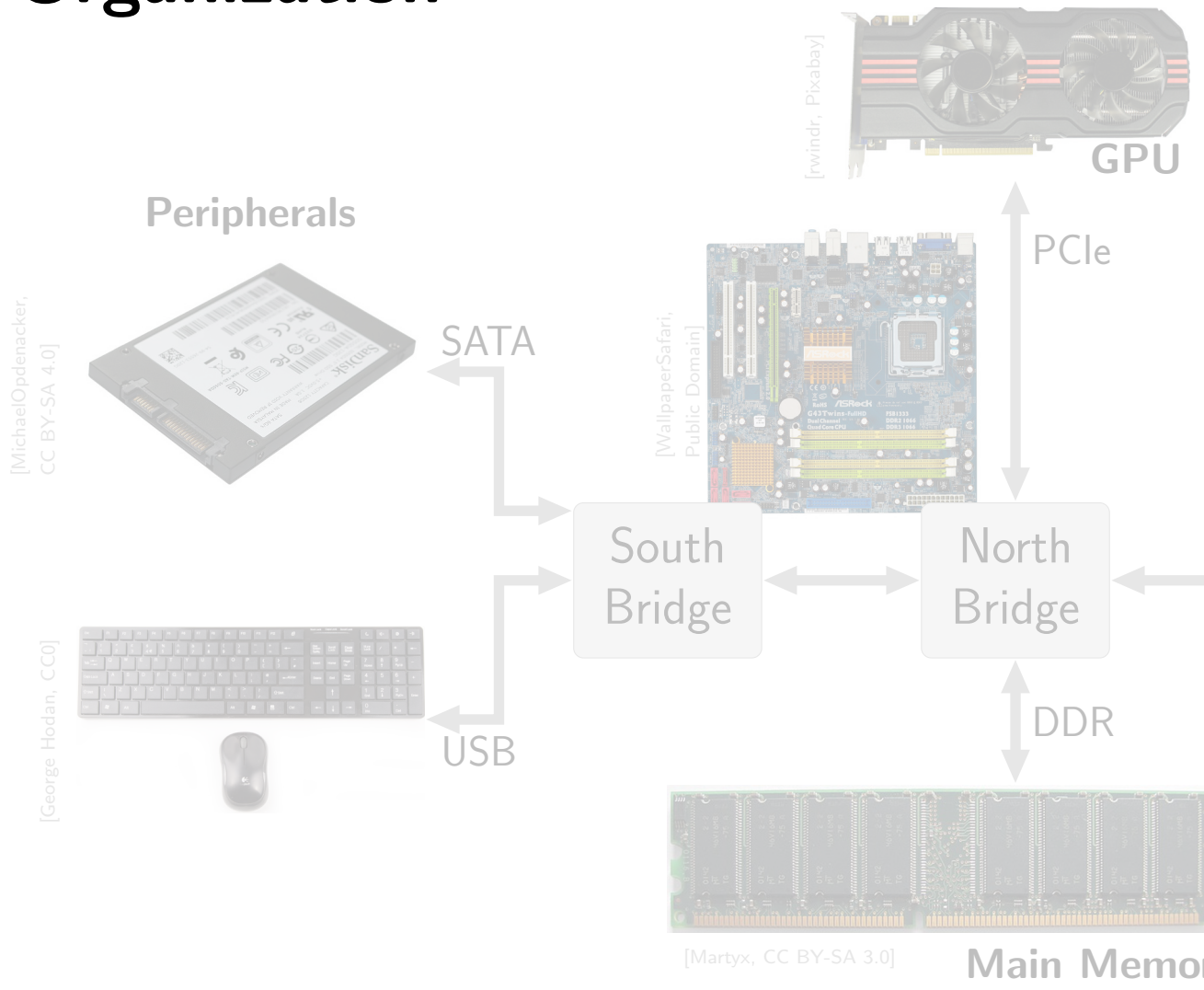
COMPUTER ARCHITECTURE

Chapter 4 – Complex Pipelining

Prof. Dr.-Ing. Stefan Wallentowitz

Department 07 – Munich University of Applied Sciences

Course Organization



[Intel DE, CC BY-SA 2.0]

Central Processing Unit (CPU)

- Caches
- Basics
- Pipelining
- Compley Pipelining

Pipeline Optimizations

Goal: Bring IPC up (near to one or even above)

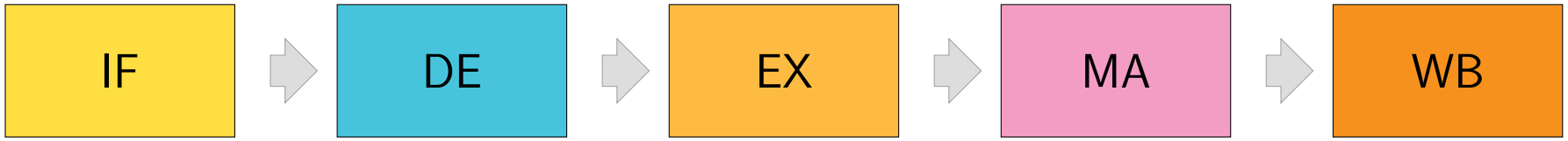
Speculative Execution

- Branch prediction: Reduce the impact of branch decisions
- Other kinds of speculation: Address, data, ...

Parallelism

- Instruction Level Parallelism (ILP)
  - Pipelining
  - Superscalar execution, out-of-order execution (lecture part 4)
- Data parallelism
  - Data vectors, single instruction multiple data
- Thread parallelism
  - Execution of multiple different instruction streams

Recap: Pipelining



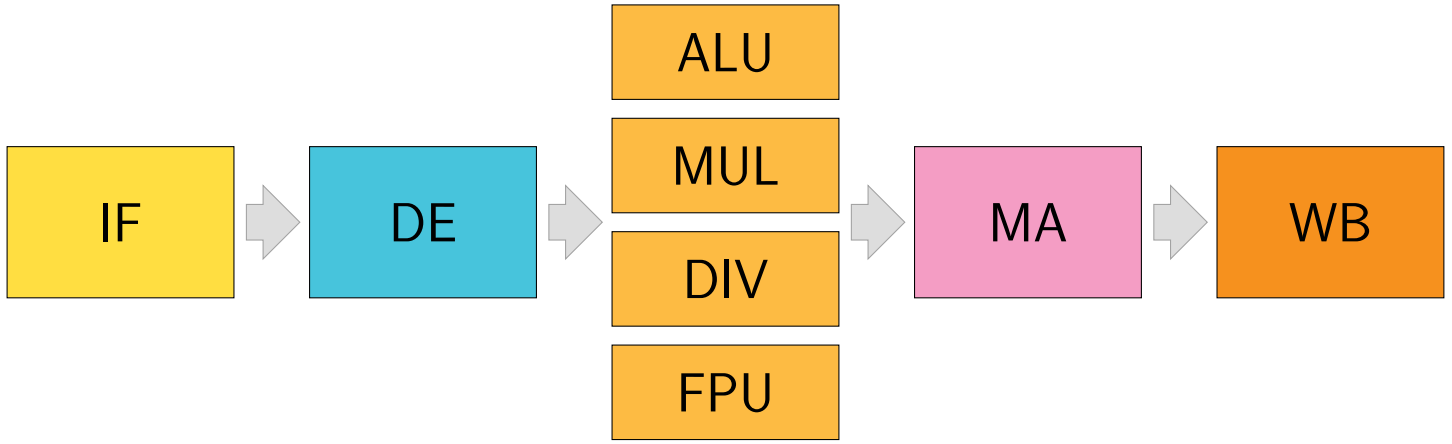
Assumption: Each instruction takes one cycle per stage

- General exception:Memory accesses take multiple cycles

Implementation of execute stage:

- Basically: Arithmetic and Logical Unit (ALU)
- But also:
  - Branch offset ALU
  - Multiplier/Divider (RISC-V M extension)
  - Floating Point Unit (RISC-V F/D extension)

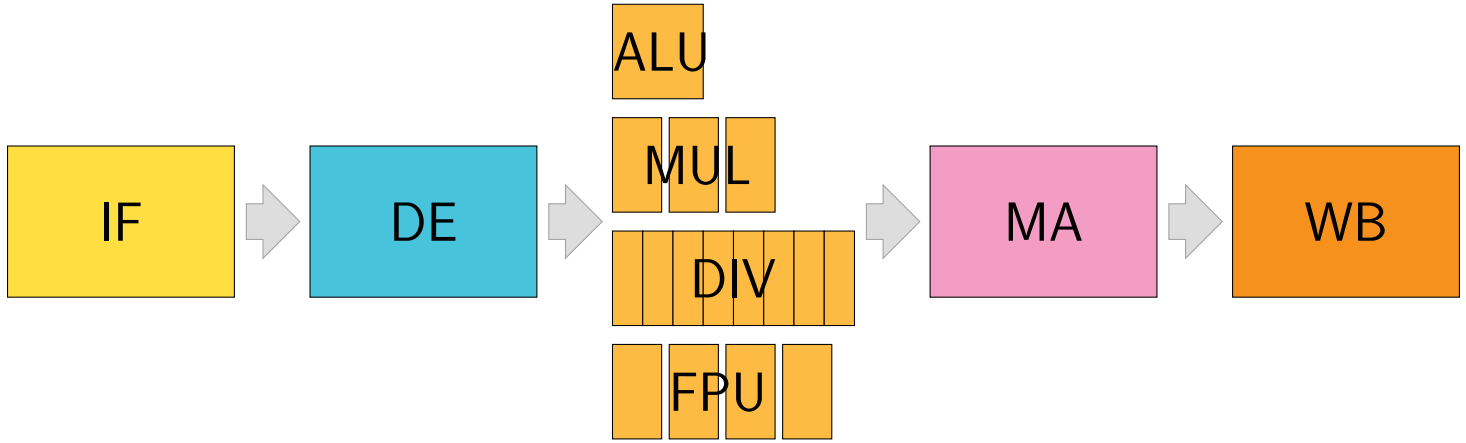
Pipeline: Functional Units



Split EX into functional units (FU): Different hardware building blocks

Multicycle instructions: Instructions don't complete in one cycle:
Multiplier, Divider, Floating Point Unit (FPU)

Functional Units: Multicycle Pipelining



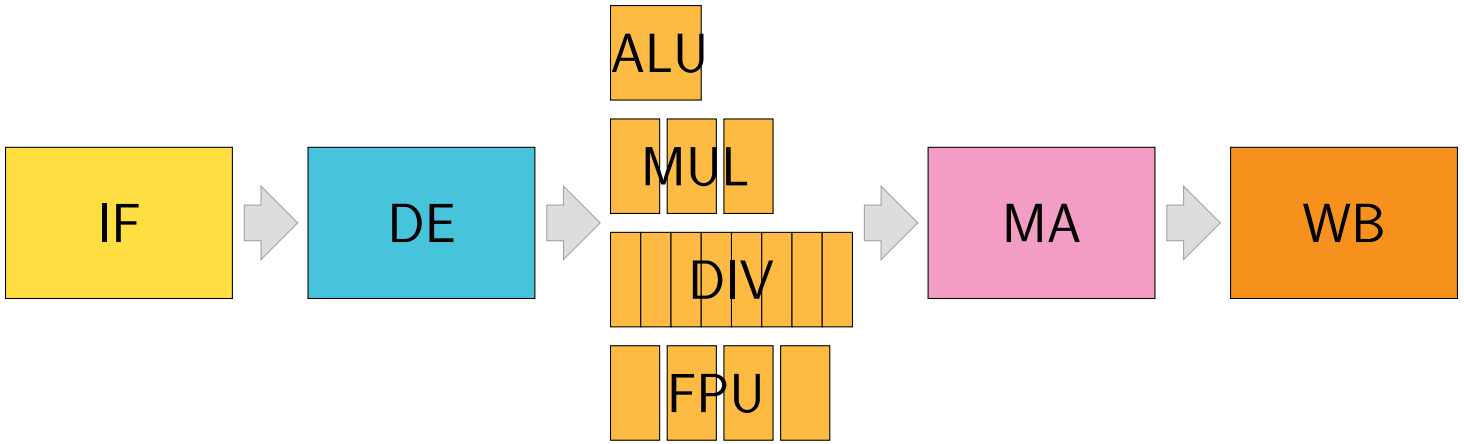
Multicycle FUs may be pipelined: Decomposition of operation

Sometimes not possible: DIV often shares one unit over multiple cycles

Allows for parallel execution of multiple instructions in one FU (not always the case)

(note: In the diagram each block corresponds to one clock cycle, differently scaled)

Multicycle Metrics



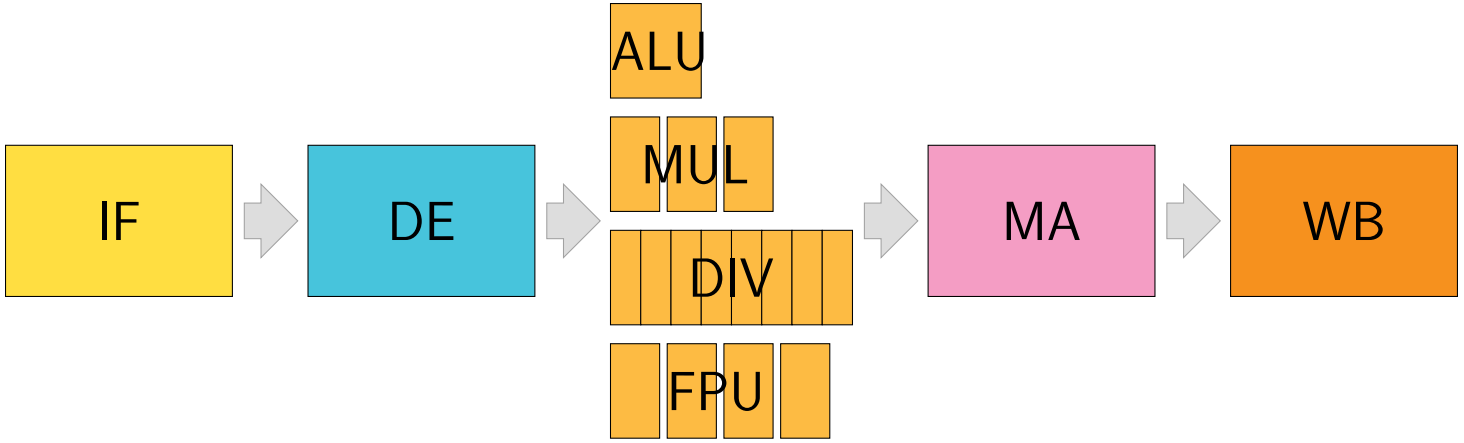
Latency

- Minimal time for instruction to traverse a functional unit

Initiation Interval

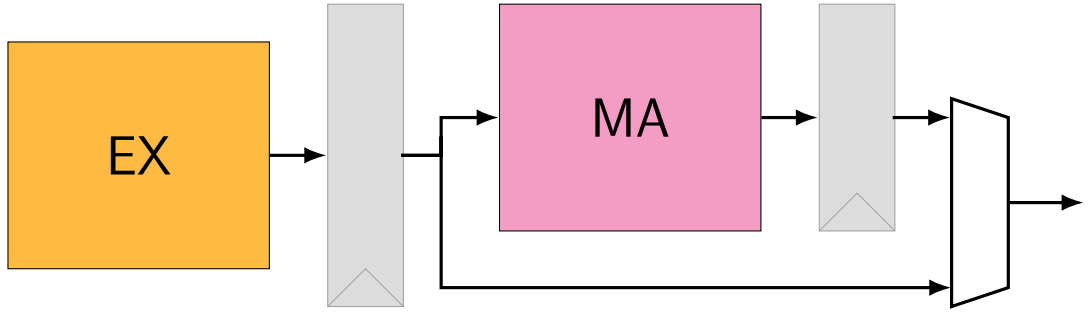
- Minimal duration between two instructions can be started in a functional unit

Multicycle Metrics



	Latency	Initiation Interval
(Integer) ALU	1	1
Multiplier	3	1
Divider	8	8
FPU	4	1

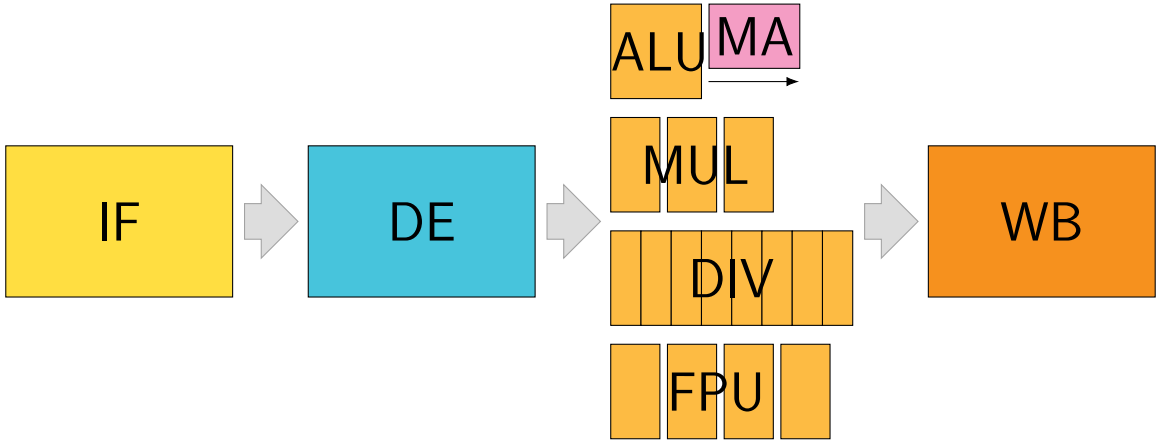
Load-Store Access



Memory access can be optional as most operations don't use it

Specifically after split into functional units

Load-Store Access



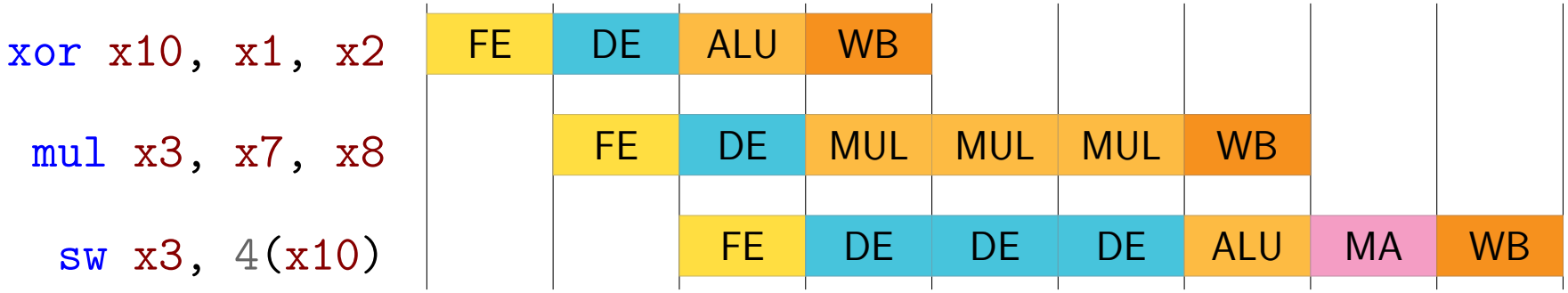
In pipeline with FUs, MA can be an optional extra stage after ALU

- Other paths are not concerned
- For ALU operations its optional to traverse MA

Example: Only one instruction in FUs

Similar as before: Only one instruction can be in any FU at any time

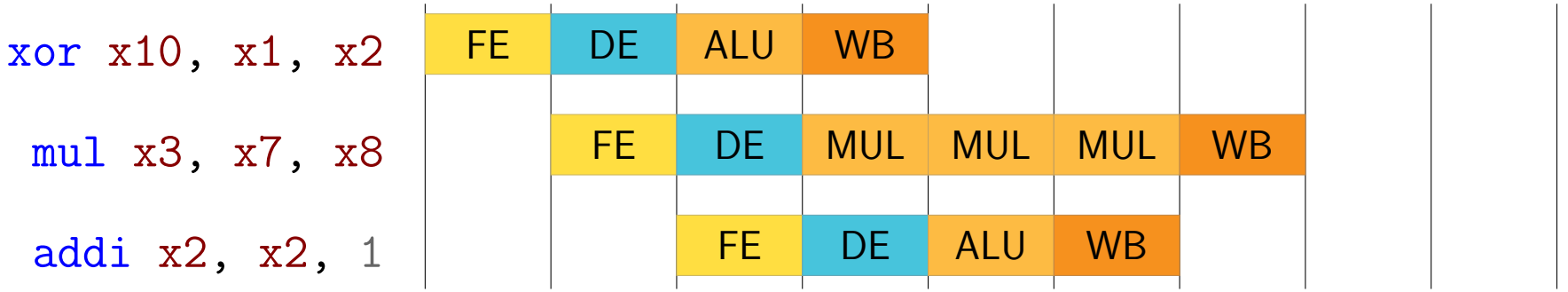
Structural hazard for multicycle operations



Example: Overlap FU processing

Issue at most one instruction per cycle

But: multicycle instructions may still be ongoing



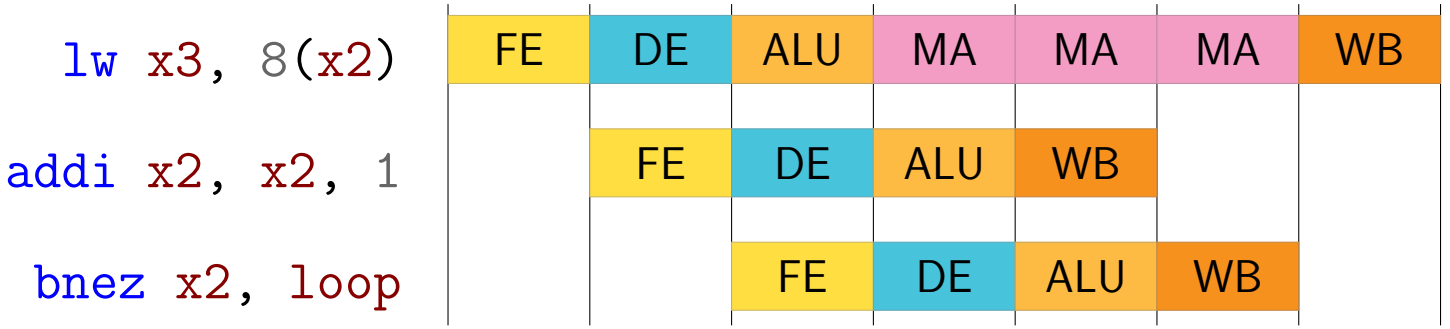
Due to different latency: Instructions can "overtake" others

Out-of-Order Completion

Even when started in correct order, instructions can complete *out-of-order*

Structural hazard on writeback stage, can be resolved

Example:



Problems?

- What when there is an exception with the load?
- Example: Access fault, handled by OS, then continue

Out-of-Order Completion: Exception

Problem:

- Following instructions completed when load exception occurs
- Exception is handled, for example by operating system
- Processing continues with re-issuing instructions starting with `lw`
- `addi` and `bnez` will be executed again, functional error

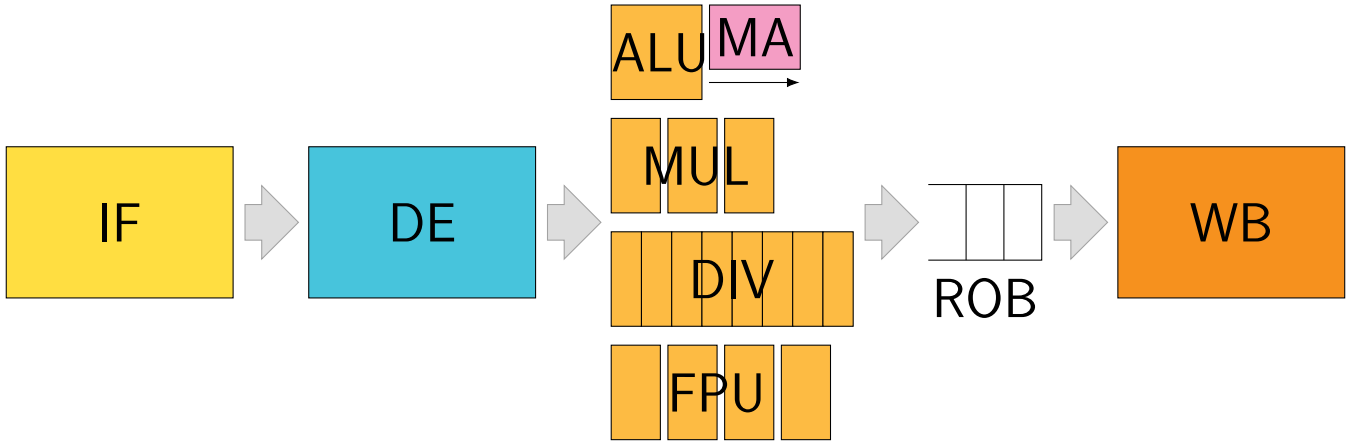
Potential solutions

- Imprecise exceptions: The exception handler needs to clean up
- Start instruction processing only after sure no exception can occur
- Buffer results and commit in correct order (forwarding needs to look there too!)

Re-Order Buffer

Split between instruction *retire* and architectural *commit*

Re-order buffer (ROB) buffers results after out-of-order retire, commits in-order



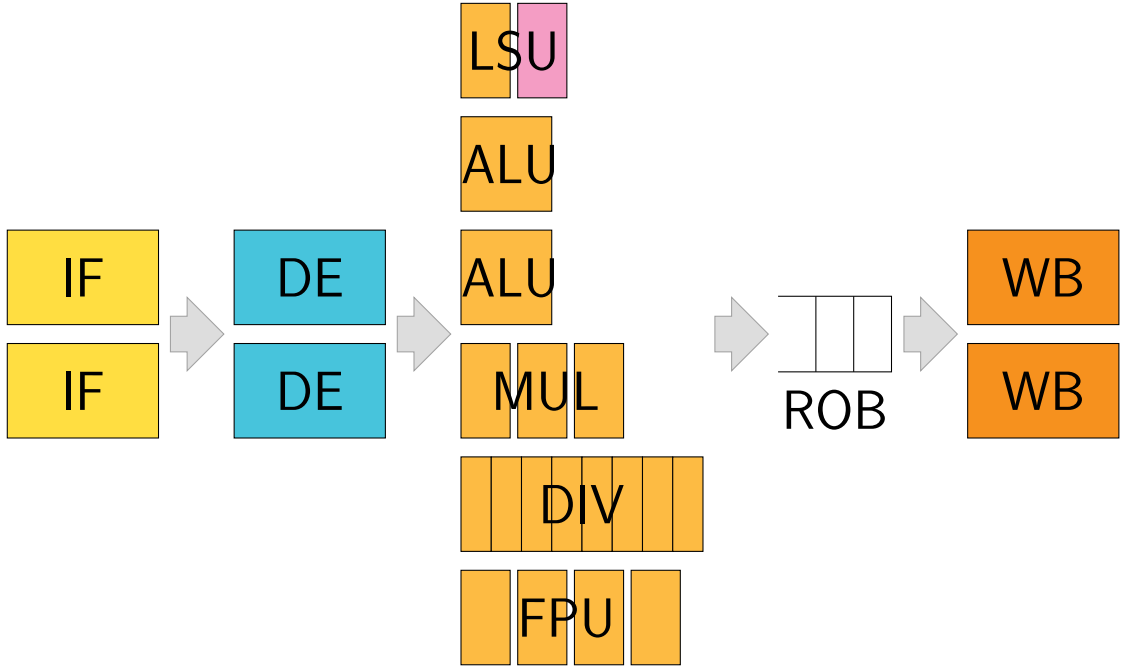
Superscalarity

Execute multiple instructions in parallel

Usually: replicate FUs

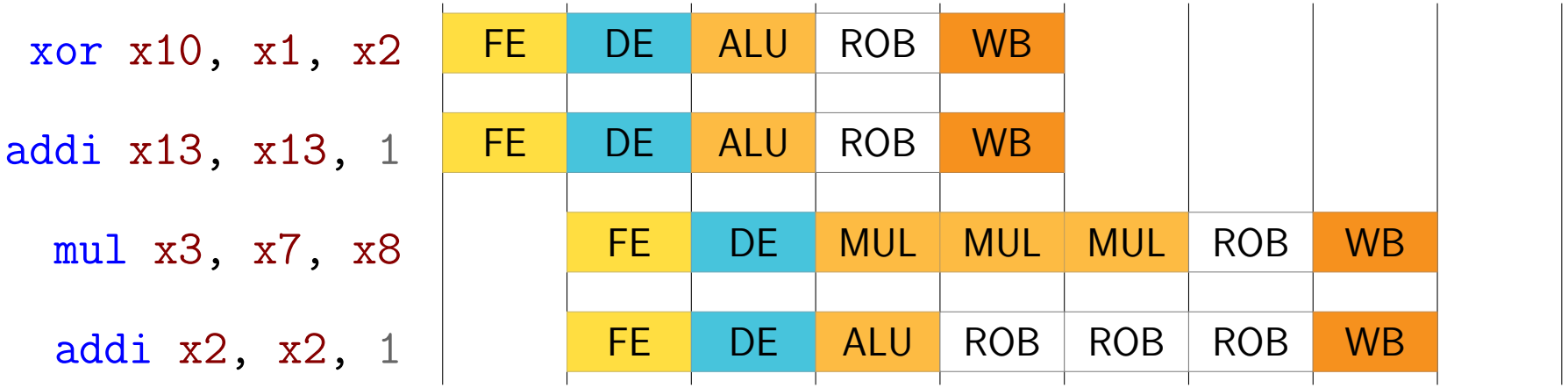
- ALU is often used
- LSU as separate FU

Increases theoretical IPC by number of parallel instructions (**issue width**, here: 2)



Superscalarity: Example

Exploit instruction level parallelism



Need issue width at each part of the pipeline, otherwise limits speedup

Instruction stream split obvious here, but how do we schedule instructions in general?

Instruction Scheduling

Scheduling: Select instructions to be started in EX stage

- From sequential order (as stored in memory)
- Need to obey data dependencies

Static Scheduling

- Execution of instructions pre-determined

Dynamic Scheduling

- Selection of instructions at runtime
  - In-order (in sequential program order)
  - Out-of-order (whenever instructions are ready)

Scheduling and Superscalarity are independent concepts

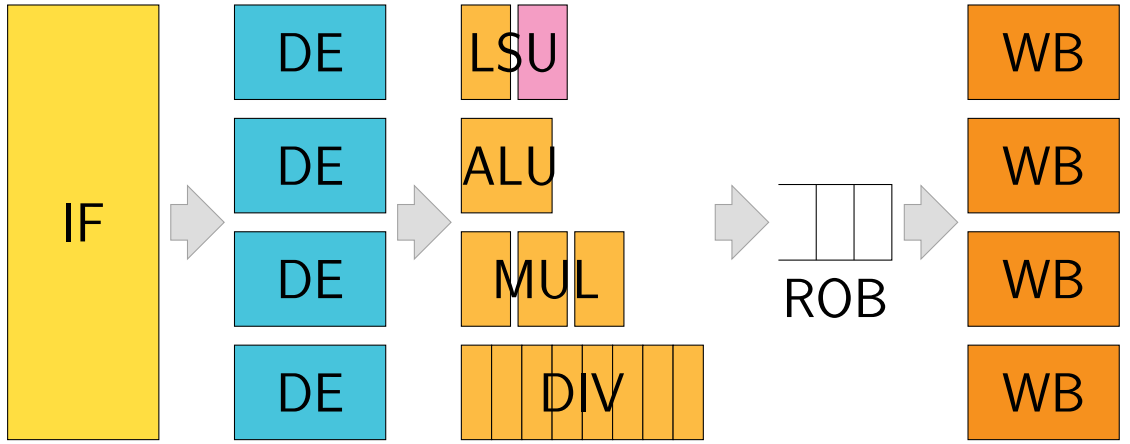
Static Scheduling: VLIW

Very Long Instruction Word (VLIW)

Combine multiple instructions in slots of a long "packet"

Start complete packet once all instructions in it can be started

Not much hardware overhead, but compilers are hard, challenge: use slots



Dynamic Instruction Scheduling

Approach: Start instructions as soon as possible

- Once structural hazards are solved: functional unit is free
- Once data hazards are solved: dependencies resolved

Keep next instruction(s) in buffer

- Instructions are still issued in order (FIFO instruction buffer)
- Where to put it? (IF-DE or DE-EX)

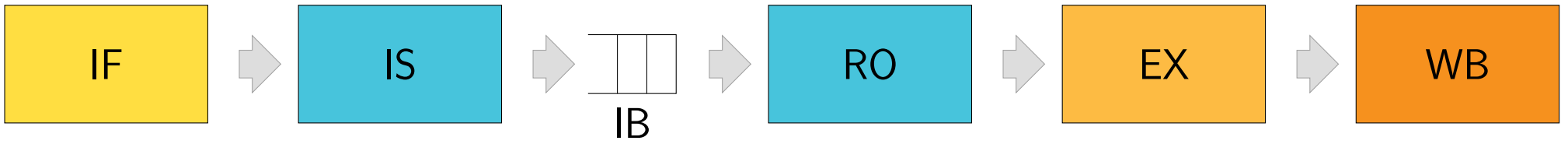
Split the Decode Phase

Issue

- Decode instruction
- Check for structural hazards (can also be instruction buffer full)

Read Operands

- Executed after data dependencies are resolved
- Then reads the operands



Scoreboard

Scoreboard: Data structure to track execution

Keeps all active instructions (in FUs)

Check current instruction for conflict

Instruction	FU	rd	rs1	rs2

Scoreboard

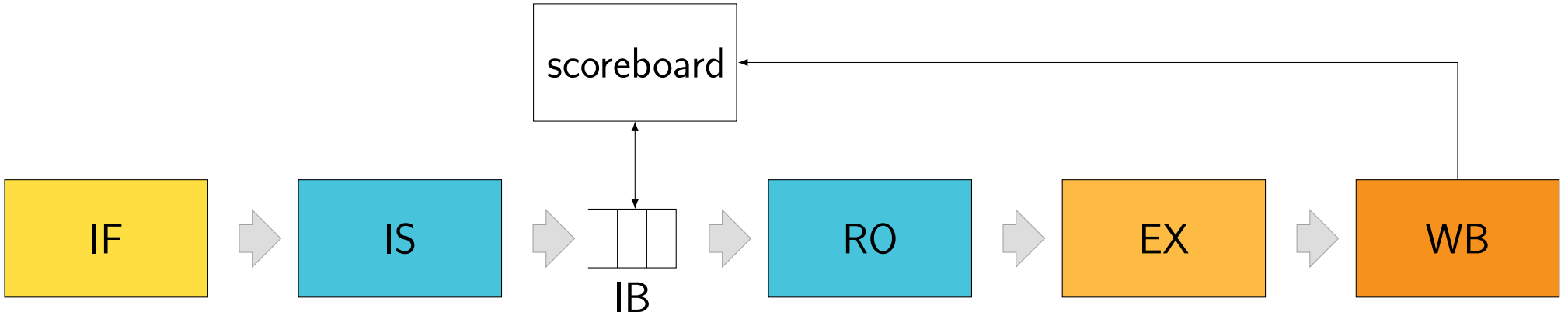
Check scoreboard for every instruction

- Structural hazard: Check if FU can start instruction
- Read-after-Write: Check SB destination registers for instruction source registers
- Write-after-Write: Check SB source registers for instruction destination register
- Write-after-Write: Check SB destination register for instruction destination register

Lifecycle of entries

- Start instruction from instruction buffer once no hazards are left, add to SB
- Remove from SB once result was written

Scoreboard: Integration



In-Order Scoreboard Scheduling

Use scoreboard for in-order dynamic scheduling

Typical: No bypassing (complexity of many functional units)

Scoreboard for in-order can be simplified

- Only the next instruction in buffer can be used

Can we have WAR and WAW for the next instruction?

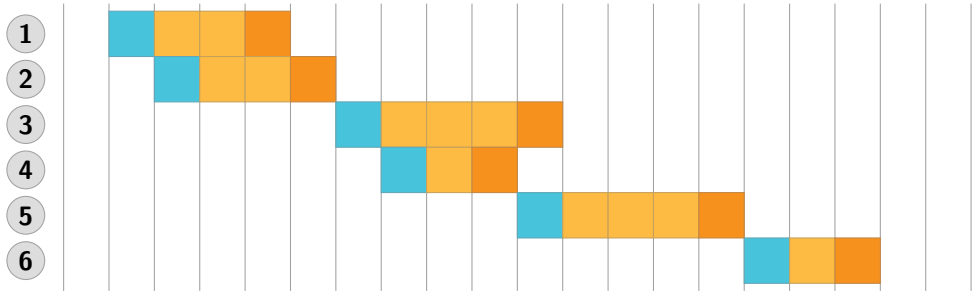
- **No WAR** as we are in order
- **WAW** can occur, hence need to track destination registers
- We only need to know the pending writes (destination registers)

Scoreboard: Example

```
1 ld x12, 8(x9)
2 ld x13, 0(x7)
3 mul x17, x13, x12
4 subi x18, x12, 2
5 mul x13, x12, x18
6 add x10, x17, x13
```

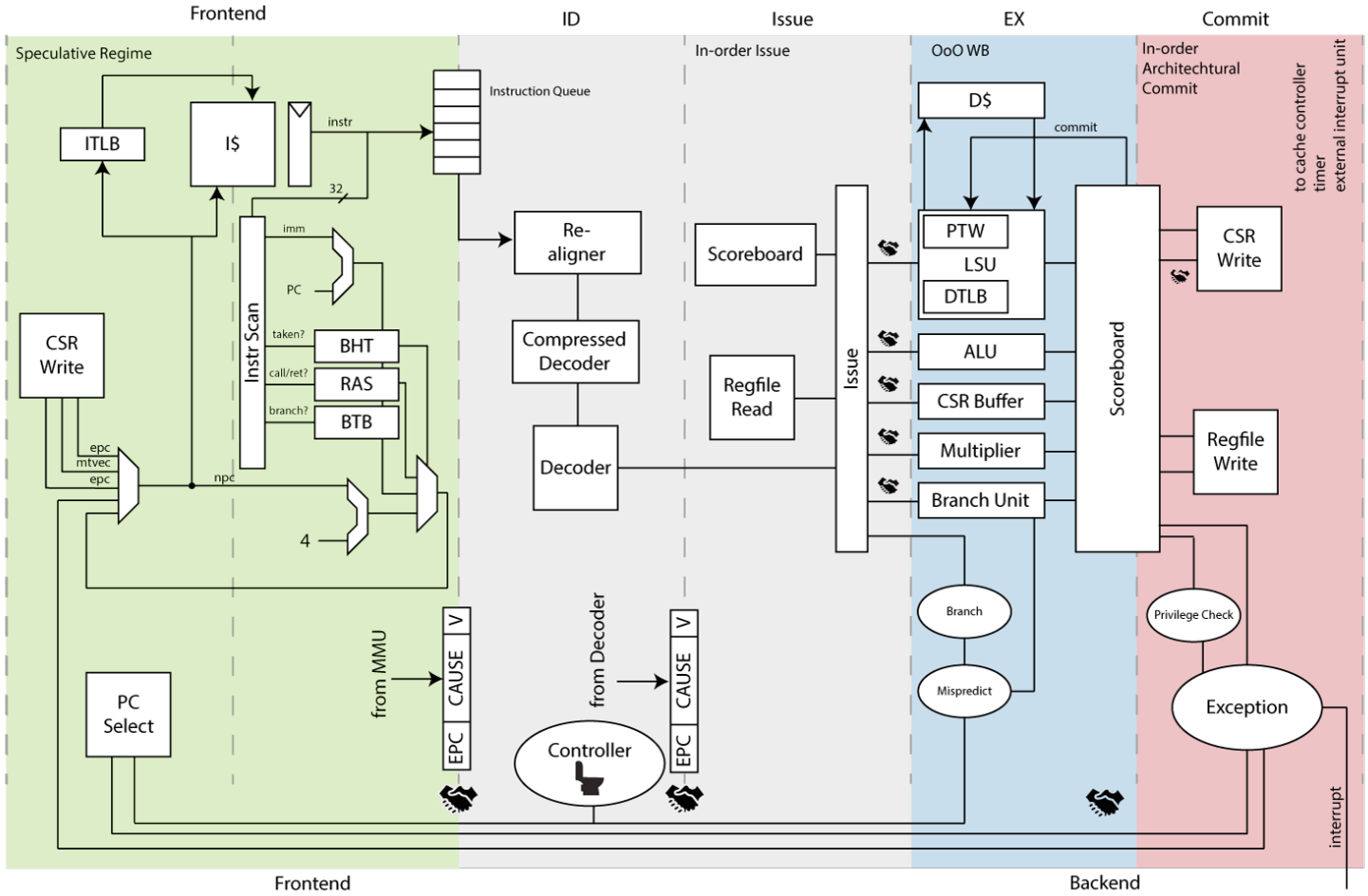
Instruction	FU	rd
~~6~~	~~MUL~~	~~x10~~
~~5~~	~~MUL~~	~~x18~~

Execution Diagram



0: #1 can be started
1: #2 can be started, #1 reads operands
2: #3 is blocked (hazards)
3: #1 will complete now
4: #1 writes result, #2 will complete too
5: #2 in WB, #3 can be started
6: #4 can be started
7: #5 blocked
9: #4 in WB, #5 can be started
10: #6 is blocked
14: #5 in WB, #6 can be started

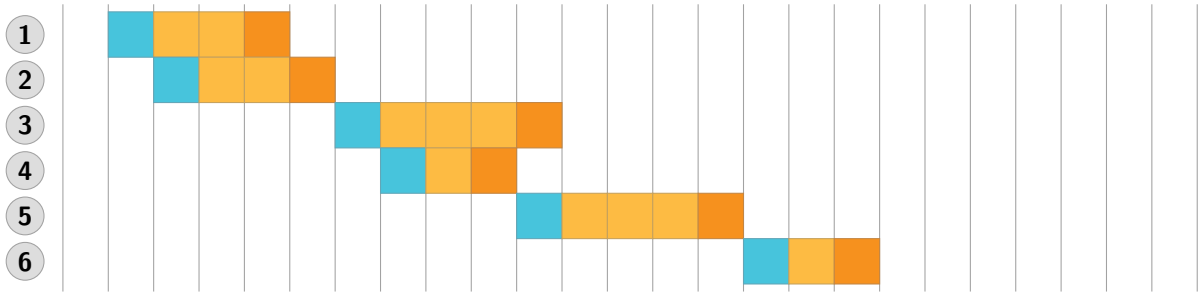
Example: Ariane CPU



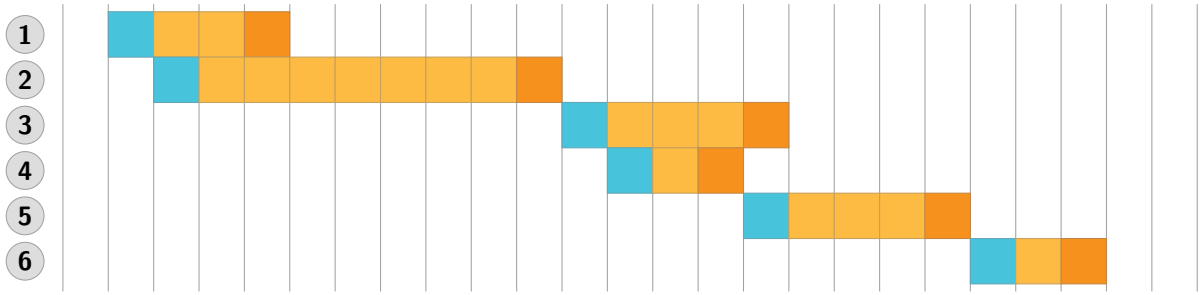
https://github.com/pulp-platform/ariane/

Limits of In-Order Issue

```
1 ld x12, 8(x9)
2 ld x13, 0(x7)
3 mul x17, x13, x12
4 subi x18, x12, 2
5 mul x13, x12, x18
6 add x10, x17, x13
```



Delayed Load Instruction



Recap: data flow model, approach: reorder instructions

Out-of-Order Issue

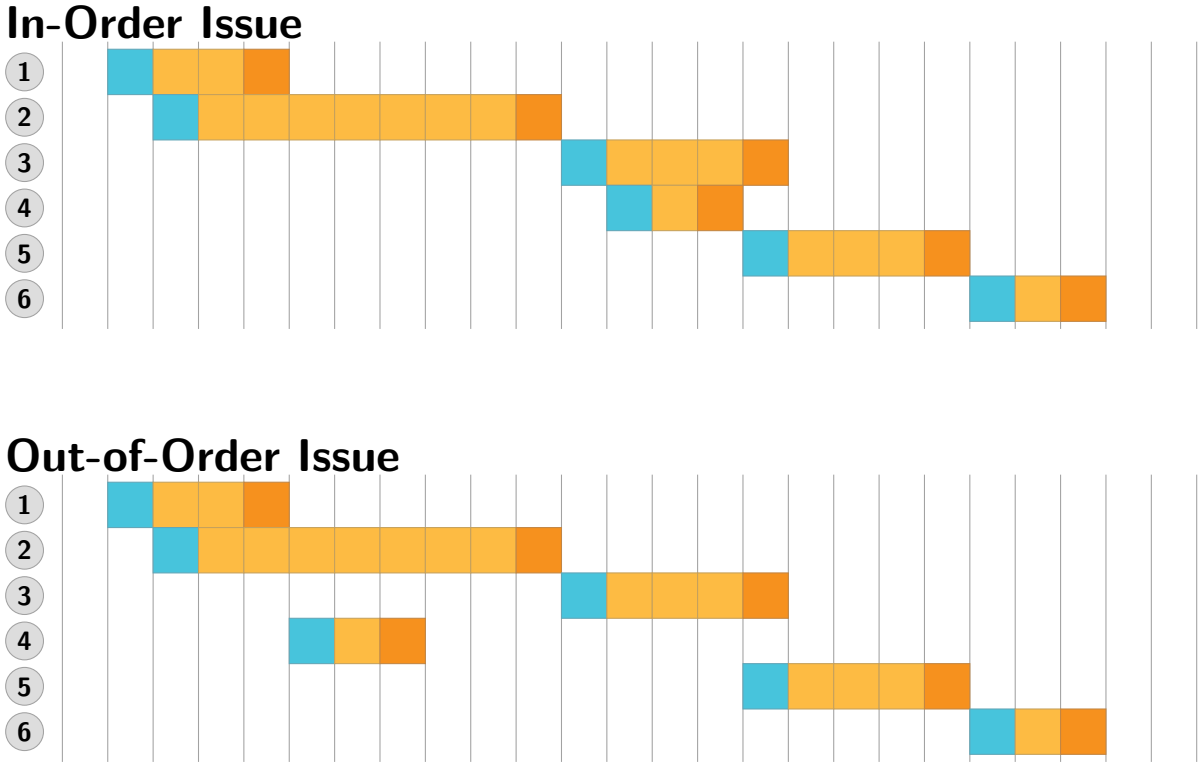
Start of instructions in arbitrary order

- As soon as hazards of each instruction are resolved
- Instruction buffer has window of next N instructions

But: Out-of-order issue does not improve very much alone!

Out-of-Order: Example

```
1 ld x12, 8(x9)
2 ld x13, 0(x7)
3 mul x17, x13, x12
4 subi x18, x12, 2
5 mul x13, x12, x18
6 add x10, x17, x13
```



Out-of-Order Limits

WAW and WAR limit further reordering

- Not real dependencies
- Artificially added: limitation of registers

Problem with limited registers

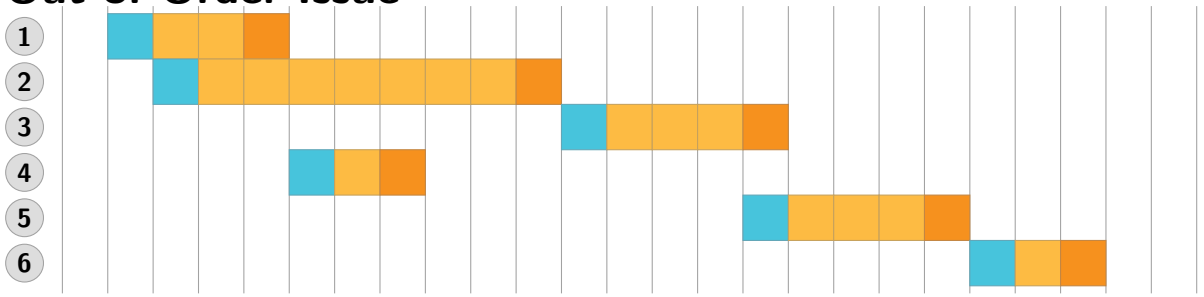
- Number of registers limited by ISA
- Compiler optimizations limited
- Especially with different execution paths

Approach: CPU solves problem by register renaming

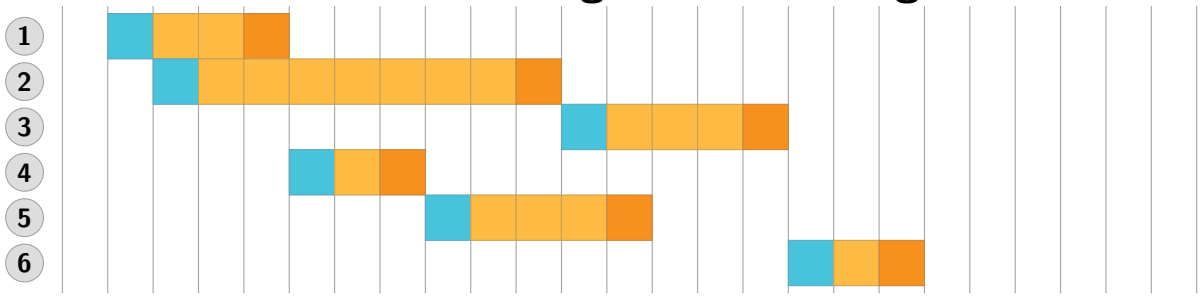
Register Renaming: Example

```
1 ld x12, 8(x9)
2 ld x13, 0(x7)
3 mul x17, x13, x12
4 subi x18, x12, 2
5 mul x13, x12, x18
6 add x10, x17, x13
```

Out-of-Order Issue



Out-of-Order Issue with Register Renaming



Register Renaming

Approach: Use microarchitecture ("virtual" register names)

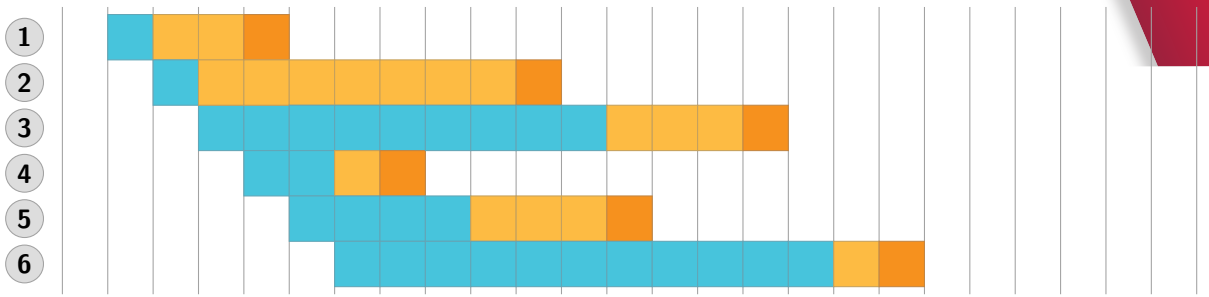
- Entirely eliminates WAR and WAW hazards
- Not visible to the outside world

Introduced by Robert Tomasulo (1967)

- Reservation stations store instructions and renames
- Format of reservation stations (multiple entries per FU)
  - Op: Operation
  - Qj, Qk: Reservation station that produces source registers (pending)
  - Vj, Vk: Value of source register (once available)
  - Busy: Reservation station is active
- Additionally: Register result status shows which RS produces registers

Tomasulo: Example

```
1 ld x12, 8(x9)
2 ld x13, 0(x7)
3 mul x17, x13, x12
4 subi x18, x12, 2
5 mul x13, x12, x18
6 add x10, x17, x13
```



	ALU					LSU					MUL				
	Insn	Vj	Vk	Qj	Qk	Insn	Vj	Vk	Qj	Qk	Insn	Vj	Vk	Qj	Qk
0															
1															

	ALU					LSU					MUL				
	Insn	Vj	Vk	Qj	Qk	Insn	Vj	Vk	Qj	Qk	Insn	Vj	Vk	Qj	Qk
0						1	8	...	-	-					
1															

	ALU					LSU					MUL				
	Insn	Vj	Vk	Qj	Qk	Insn	Vj	Vk	Qj	Qk	Insn	Vj	Vk	Qj	Qk
0						1	8	...	-	-					
1						2	0								