arm

Exploiting Instruction-level Parallelism

Module 5

Module Syllabus

- Superpipelined vs superscalar processor
- Instruction-level parallelism (ILP)
  - Simple in-order superscalar processor
- Generic superscalar processor and its functional units
  - Instruction fetch, register renaming, register data flow, data forwarding network
  - Loads and stores, exceptions and speculation, reorder buffer, unified register file approach, handling mispredicted branches
- Limits to superscalar processor

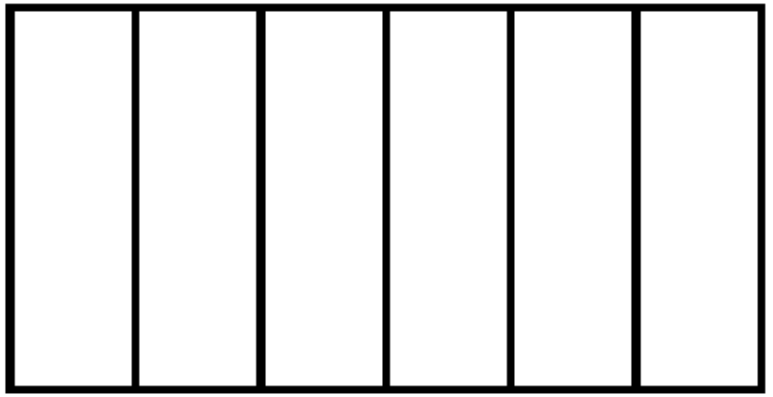
Recap: Non-pipelined and Pipelined Processors

Non-pipelined processor

IPC = 1, Clock Period = T

(Note: IPC may be less than one if we assume we sometimes need to access a slow main memory.)

Pipelined processor



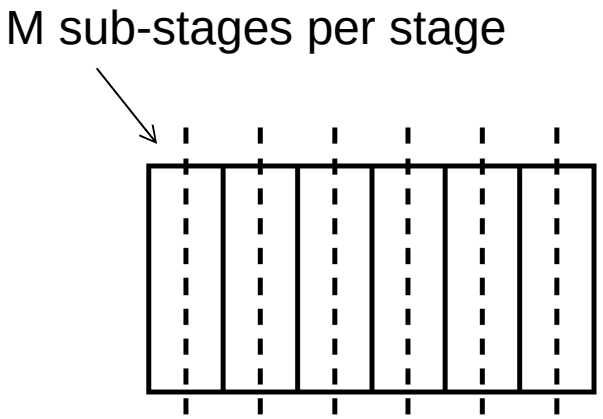
S – number of pipeline stages

IPC ≤ 1

Clock Period = T/S+C

(C = pipelining overhead)
This is a **scalar pipeline.**

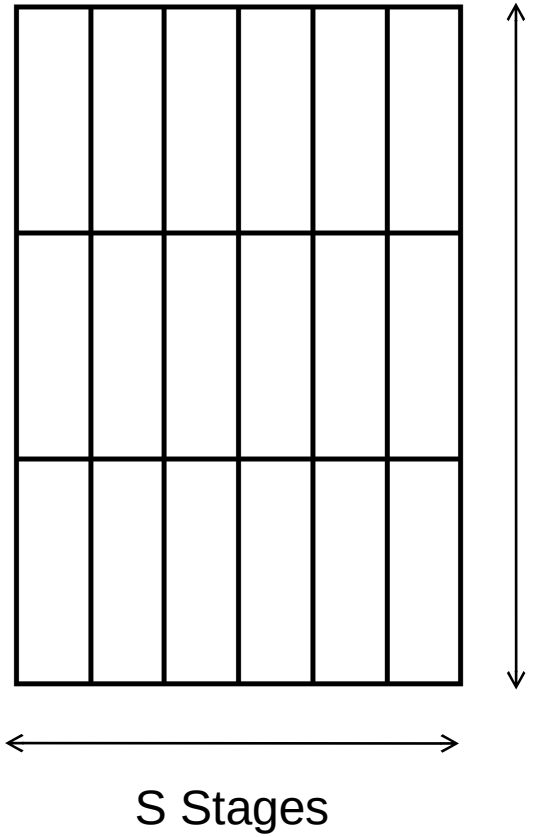
Superpipelined and Superscalar Processors



A Superpipelined Processor

The S pipeline stages (here S = 6) are further divided into M sub-stages (here M=2).

This processor executes M instructions during each of the original pipelined processor's clock periods. Its clock is M times faster.



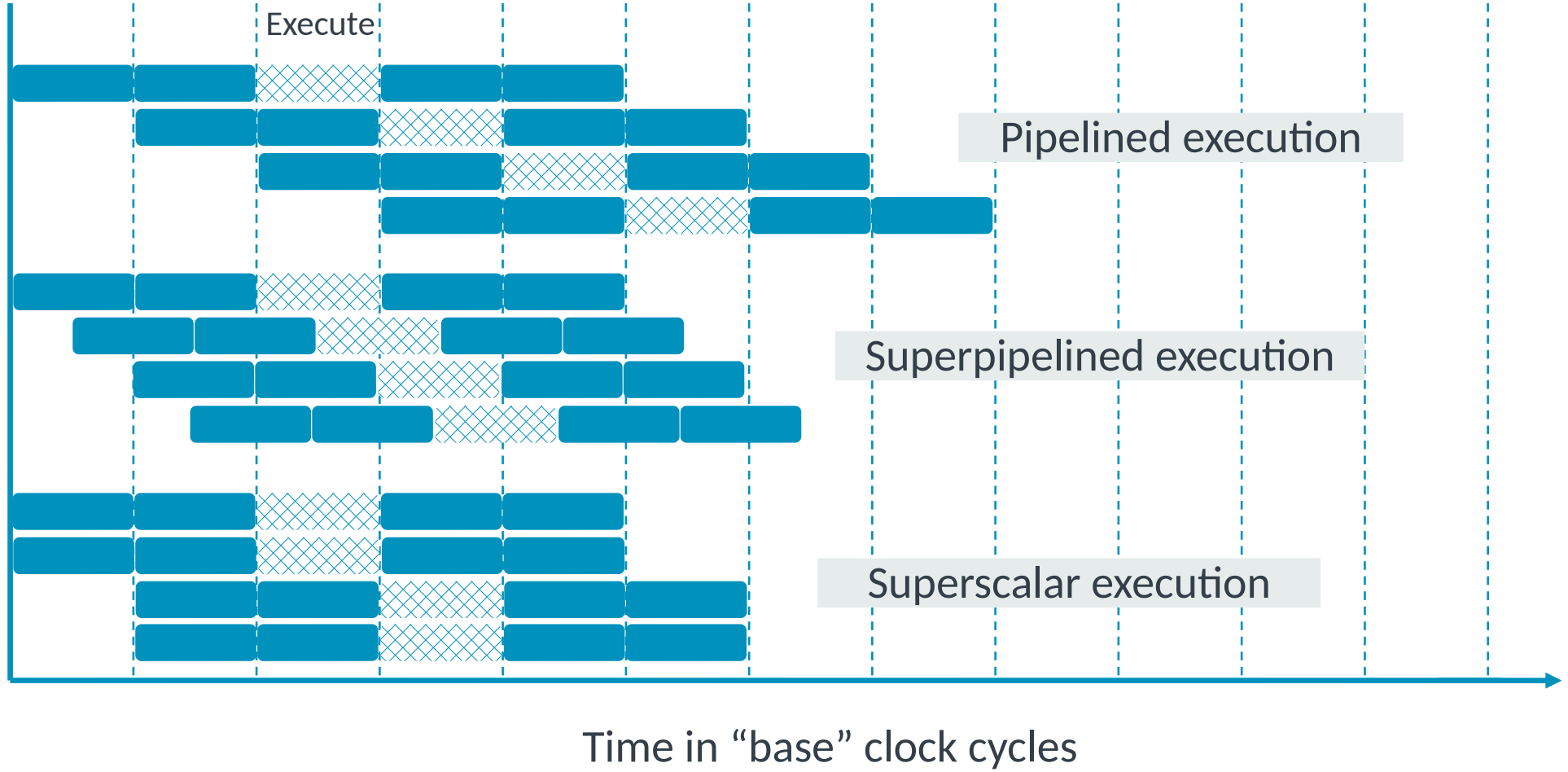
A Superscalar Processor

P instructions are processed in each pipeline stage.

IPC ≤ P

Clock Period = T/S+C

Superpipelined and Superscalar Processors



Superpipelined and Superscalar Processors

- If we ignore implementation issues, a superpipelined machine of degree M and a superscalar machine of degree P should have roughly the same performance.
- In either case, we must find (M or P) independent instructions from the program that can execute in parallel in each clock cycle. We could use software or hardware techniques to do this.

Superpipelined and Superscalar Processors

In practice, it has proved better to produce superscalar processors, often with deep pipelines, rather than purely superpipelined processors:

- Practical limits to clock frequency
- Some operations or modules are difficult to pipeline.
- The need to balance logic in pipeline stages

Instruction-level Parallelism (ILP)

We could simply fetch two instructions per clock cycle and, if they are independent, issue them together to different functional units.

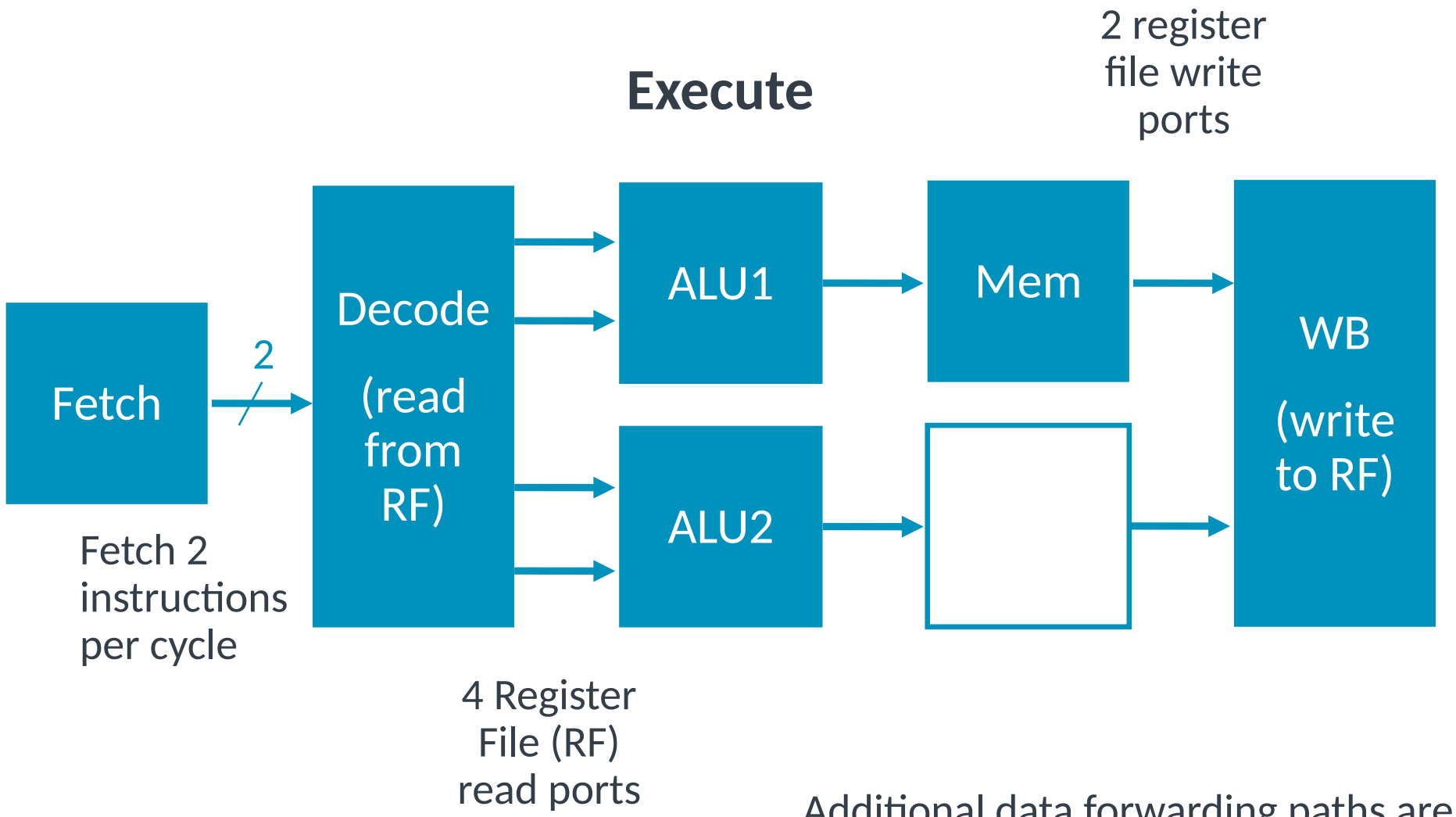
What extra hardware will this processor require?

- extra logic in decode stage to decode two instructions and check for dependencies
- register file ports? (extra read and write ports)
- functional units?
- additional data forwarding paths?

Simple In-order Superscalar Processors

- We can create a simple (2-way) superscalar processor with a few changes to our scalar pipeline.
- We will fetch and decode multiple instructions per cycle.
- Instructions are sent to functional units in program order (in-order issue).
- We will issue and execute instructions in parallel if we can.
- If we can't issue two instructions together, we simply issue one and then try to issue the waiting instruction on the next cycle.

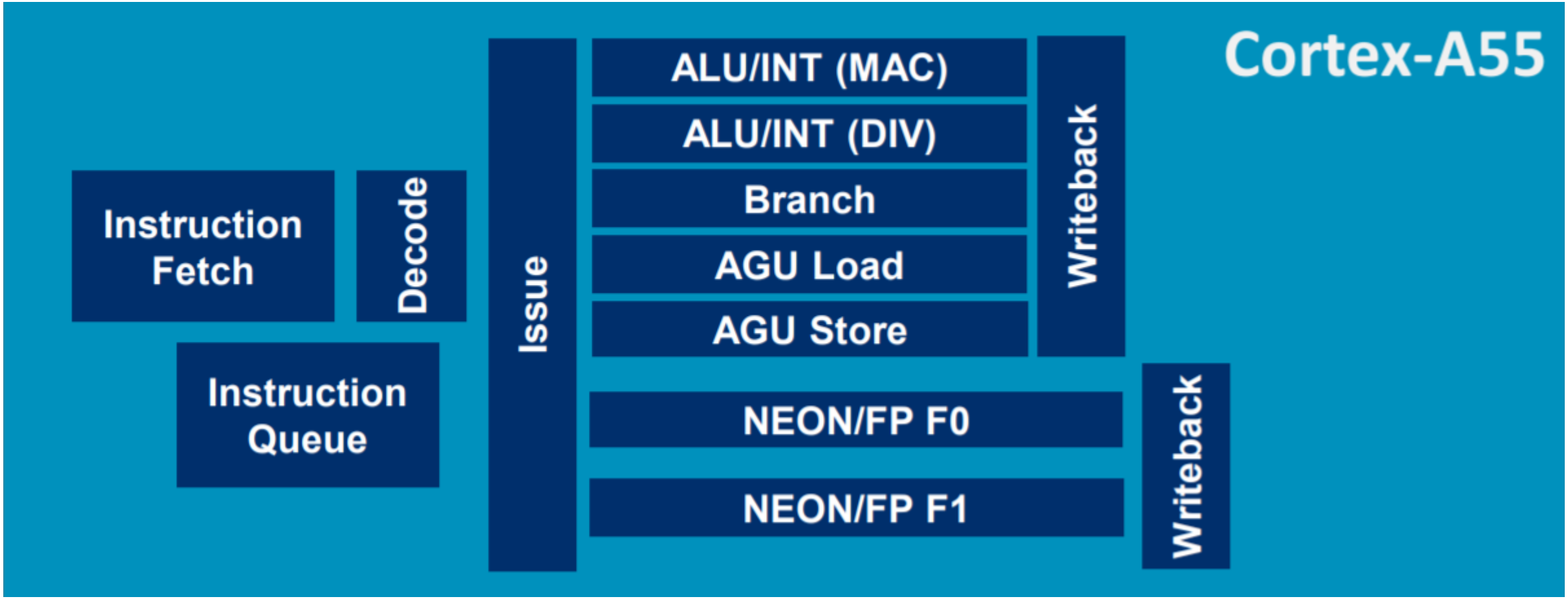
Simple In-order Superscalar Processor



Additional data forwarding paths are also required (not shown here), from and to both ALUs.

arm

Arm Cortex-A55



2-wide instruction fetch, in-order “dual” instruction issue, 8-stage integer pipeline
(Armv8.2-A architecture)

arm

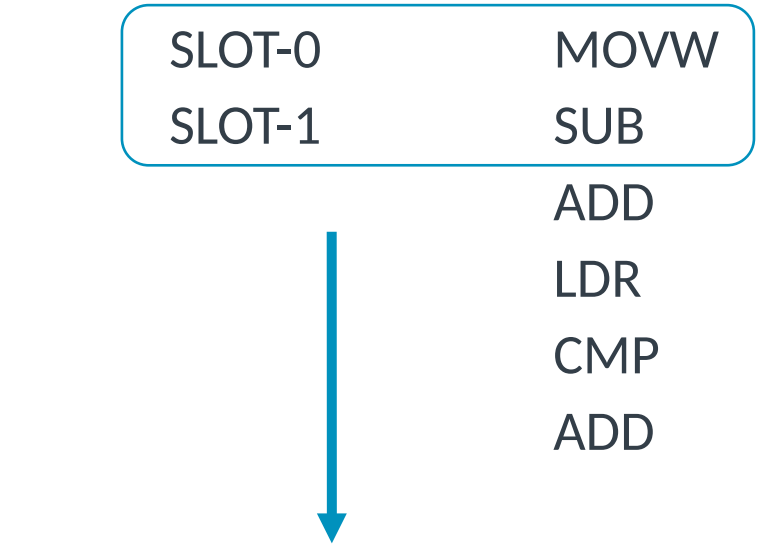
Issue Slots

A dual-issue, in-order pipeline

Here, issue “slot-0” and “slot-1” operate as a sliding window or shift register.

In general, we can’t dual-issue if:

- There is a data dependence between the two instructions.
- There is a structural dependence (i.e., they both need the same function unit (FU) resource that has not been duplicated).
- The FU resource required by one of the instructions is busy.



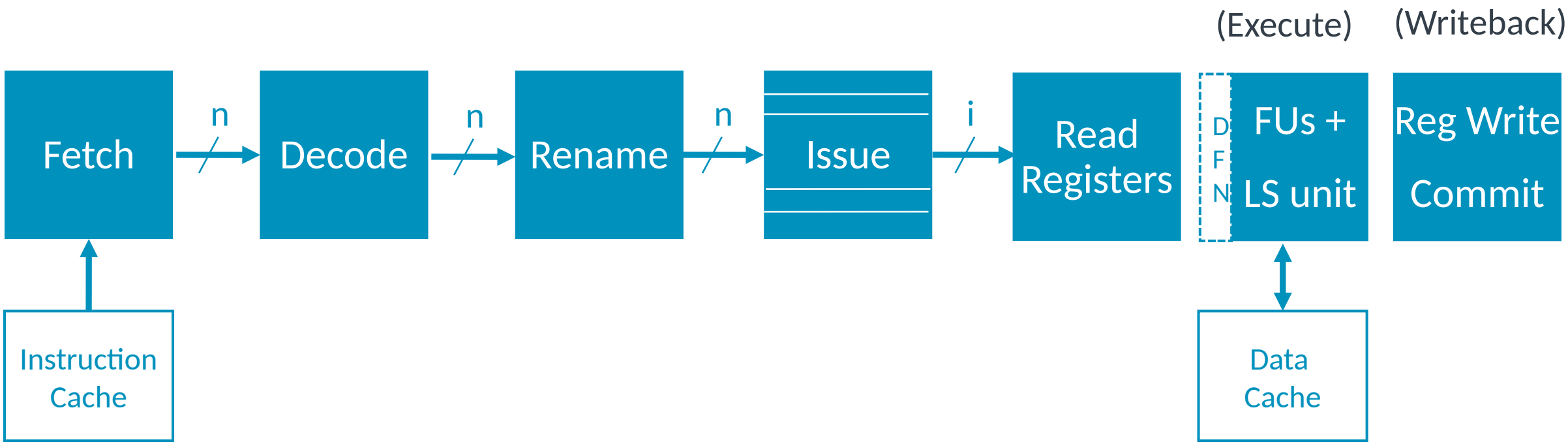
Instructions are issued to functional units in program order and in pairs, if possible

Exposing and Exploiting More ILP

To expose more ILP, we need to consider:

- Branch prediction and speculative execution
- Removing name (or false) data dependencies
- Dynamic instruction scheduling

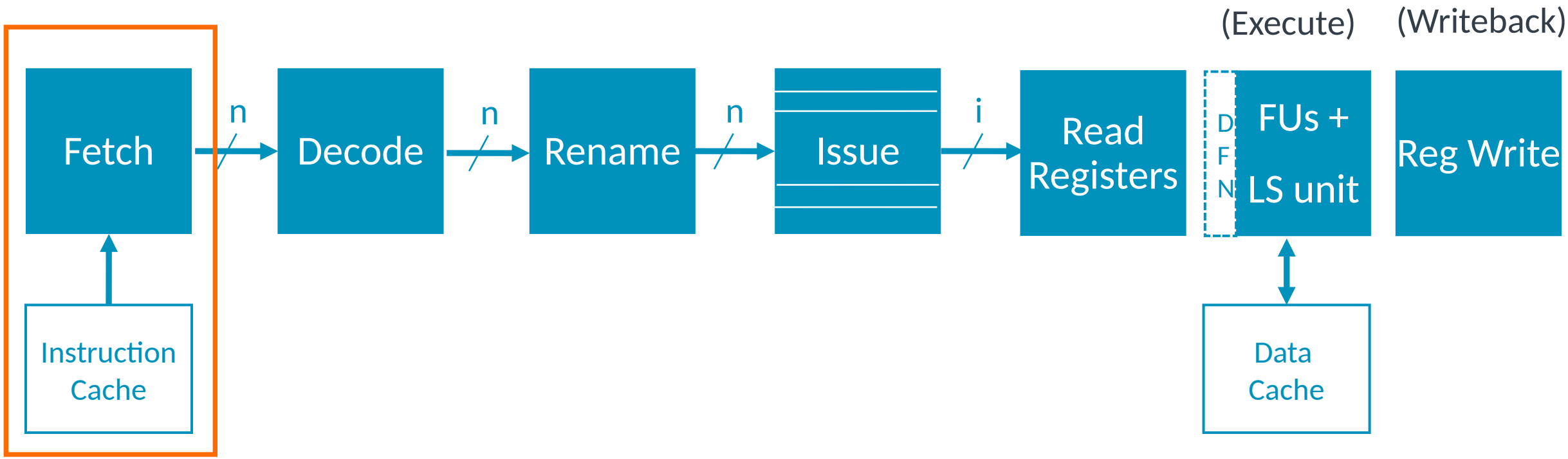
A Generic Superscalar Processor



LS unit = Load/Store unit

DFN = Data Forwarding Network

A Generic Superscalar Processor



LS unit = Load/Store unit

DFN = Data Forwarding Network

arm

Superscalar Processors: Instruction Fetch

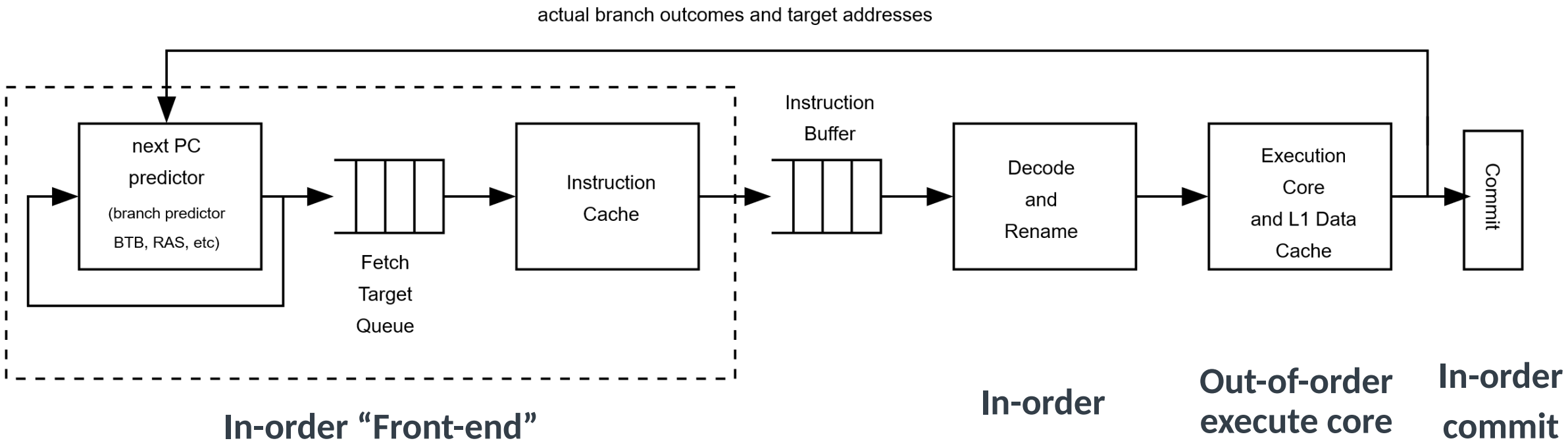
Our superscalar pipeline cannot process instructions faster than they are supplied, so maintaining a good instruction fetch rate is very important.

Potential limitations:

- Branch prediction accuracy (may also need to predict multiple branches at a time)
- Instruction cache performance (see later module on caches)
- Instruction fetch and alignment issues
  - The instructions we need to fetch might be in different cache lines.

Superscalar Processors: Instruction Fetch

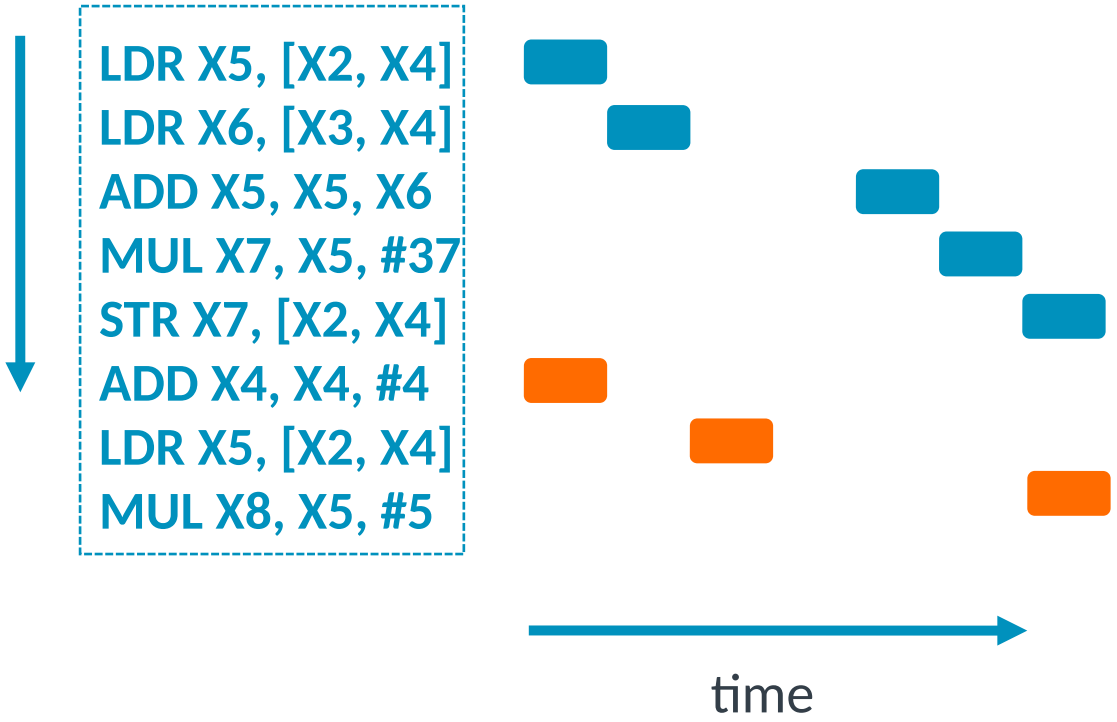
Our instruction fetch (front-end) can be decoupled from the part of the processor that actually executes instructions. The aim here is to run ahead, fill the instruction buffer, and help keep our execution units fed.



Note: each block may represent multiple pipeline stages

Superscalar Processors: Register Renaming

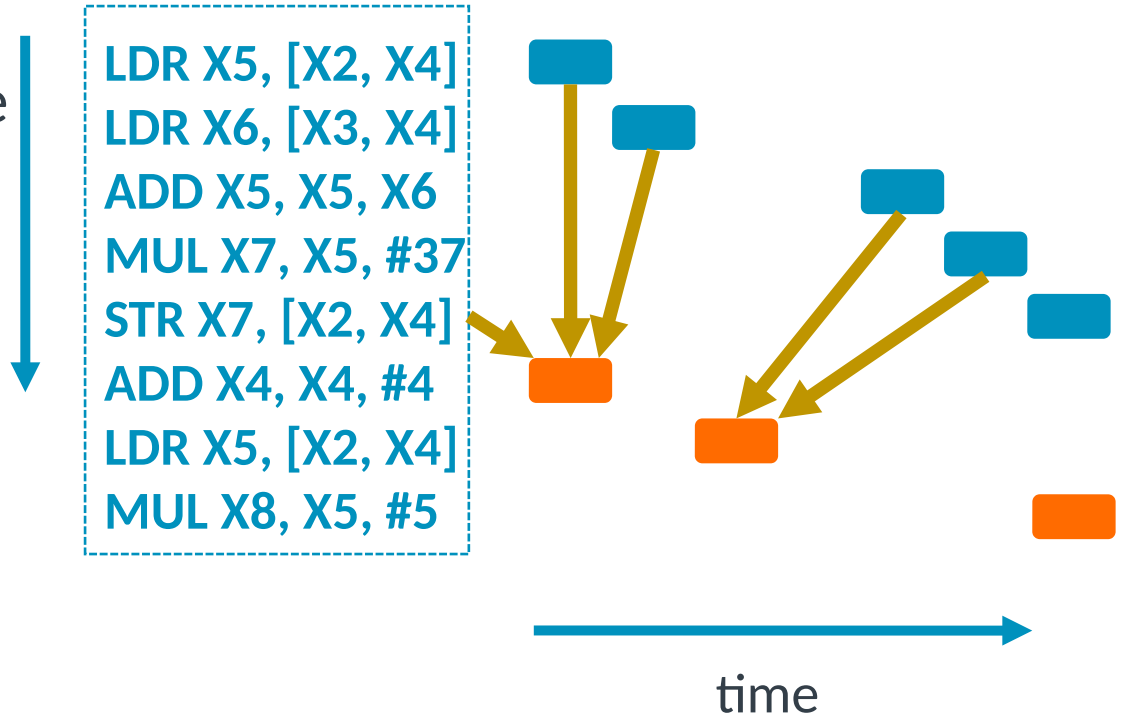
- High-performance superscalar processors are able to maintain a window into the dynamic instruction stream.
- They are able to issue instructions from anywhere in this window when their operands are ready.



arm

Superscalar Processors: Register Renaming

- In practice, name (or false) dependencies may limit our ability to perform this out-of-order instruction issue.
- These are present as the compiler must reuse a limited number architectural (or logical) register names.
- The arrows highlight the false dependencies present in this code snippet.

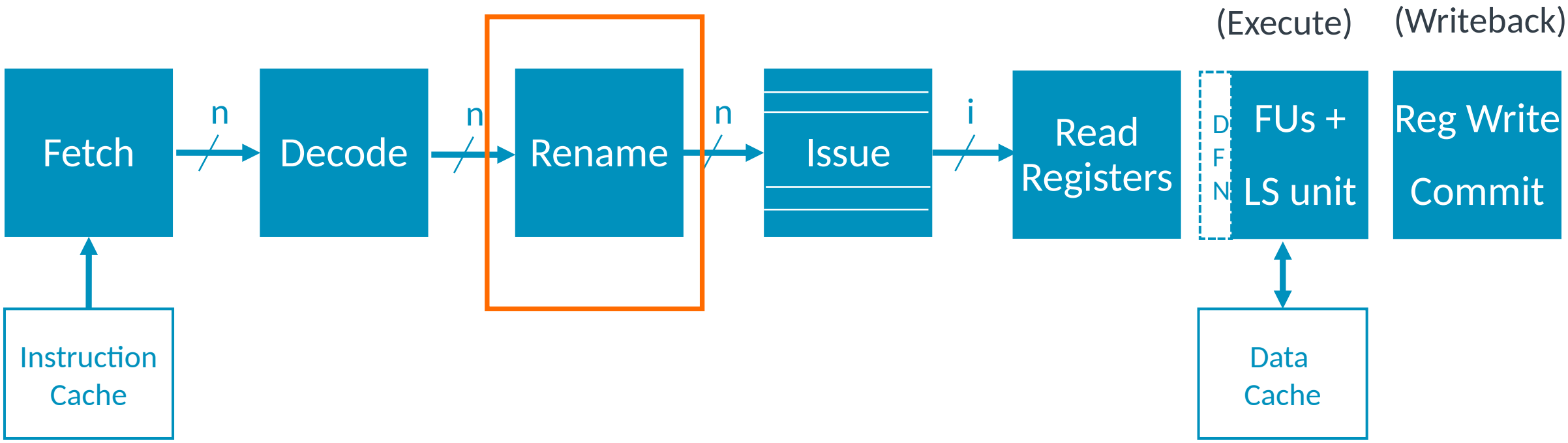


arm

Superscalar Processors: Register Renaming

- Register renaming may be performed in hardware at run-time.
- It provides each instruction with a unique physical destination register.
- Given enough physical registers, renaming can remove all name dependencies.
- The processor has many more physical registers than architectural ones, e.g.:
  - The A64 ISA provides 31 (64-bit) general-purpose registers that the compiler may use.
  - A high-performance superscalar Arm processor may provide 128 or more physical registers.
- The architectural register names are “renamed” to physical ones early in the pipeline.

A Generic Superscalar Processor

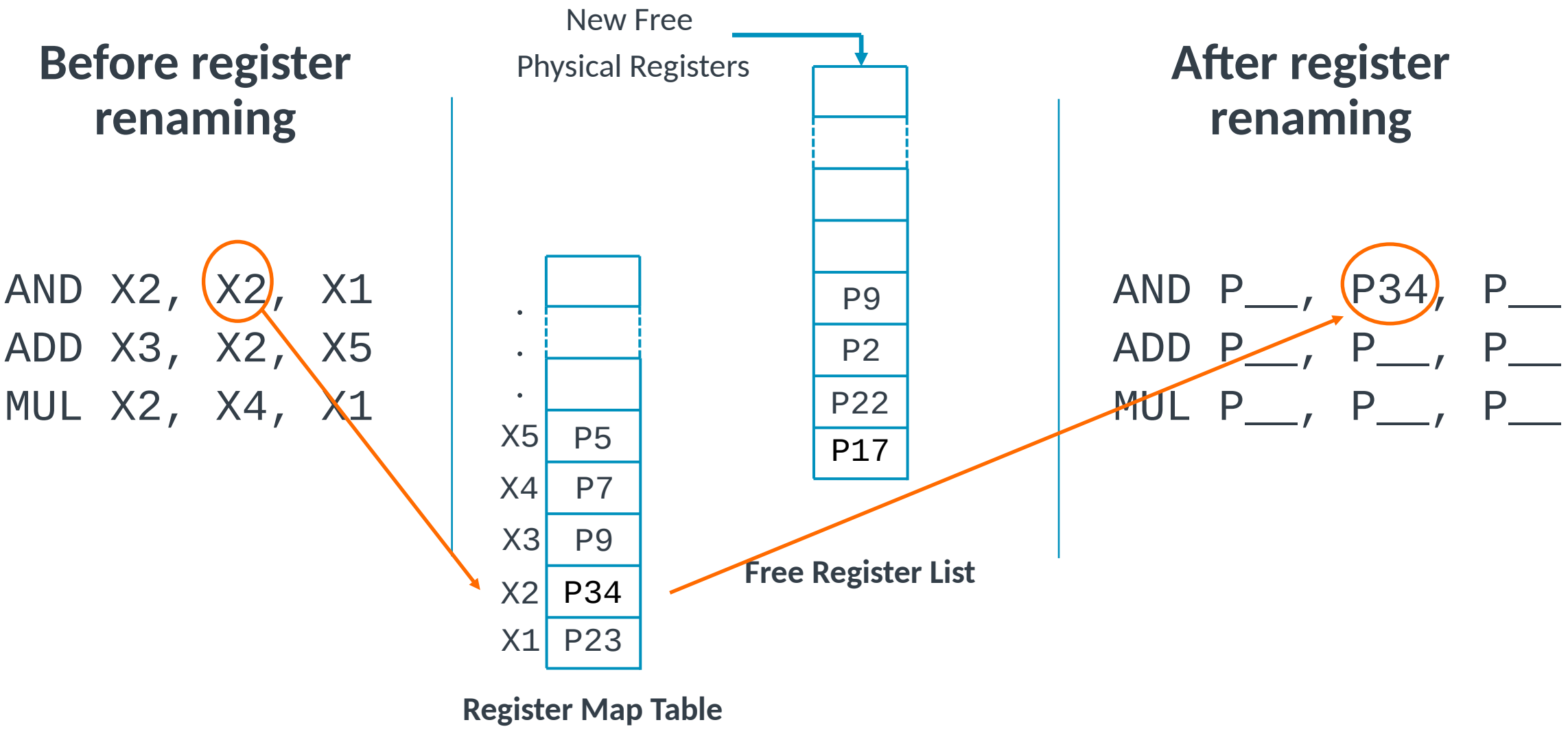


LS unit = Load/Store unit

DFN = Data Forwarding Network

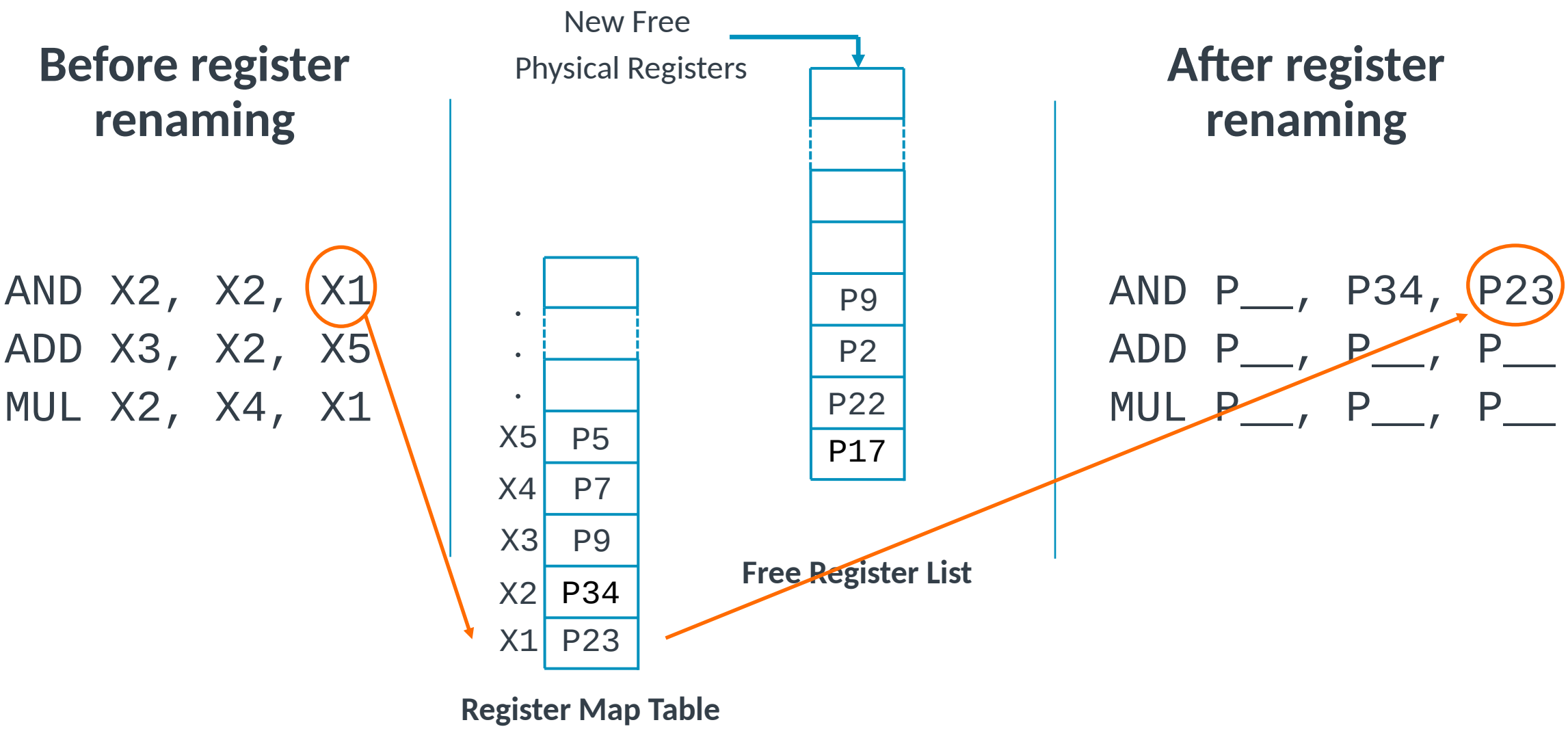
arm

Superscalar Processors: Register Renaming



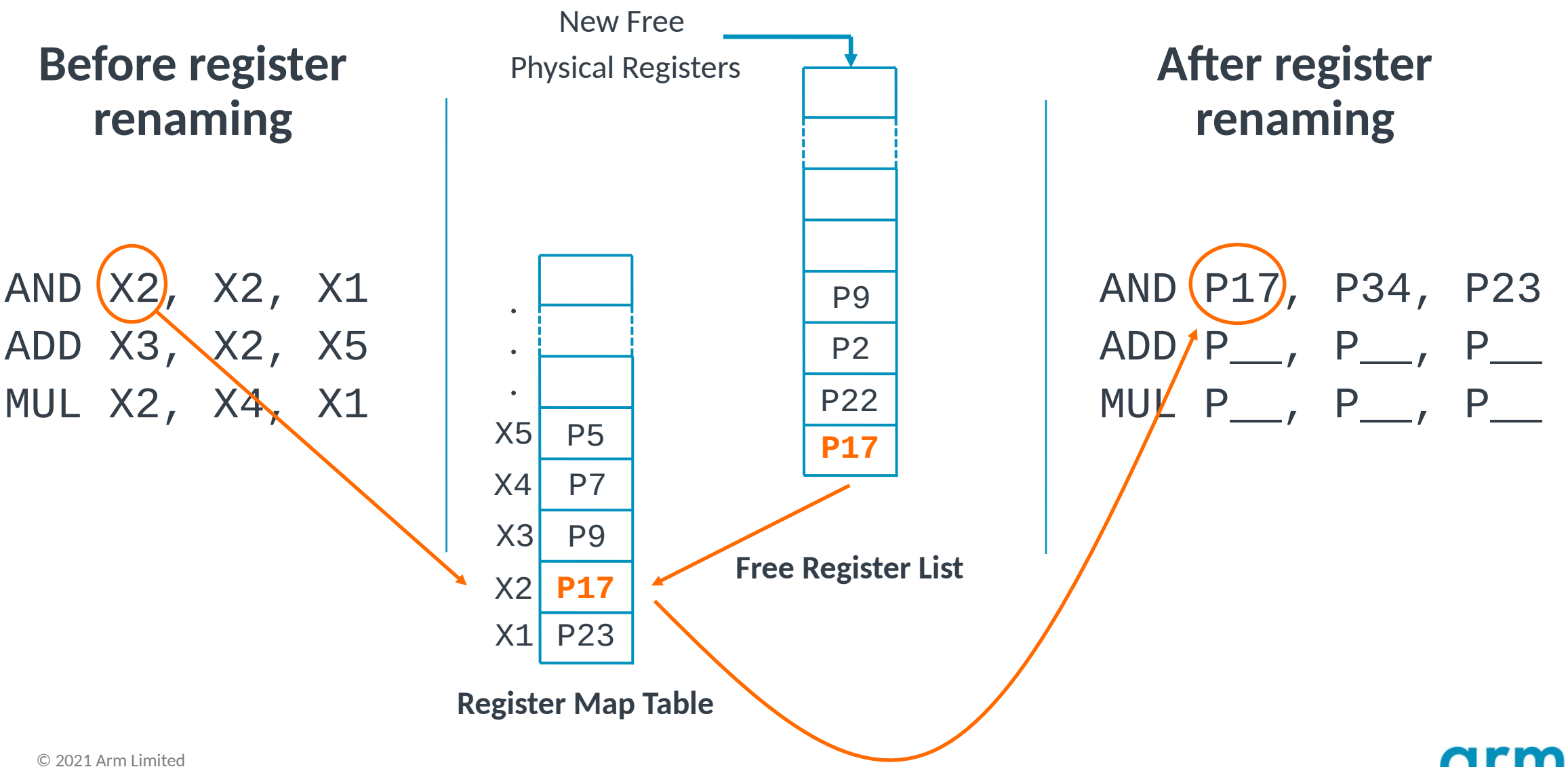
arm

Superscalar Processors: Register Renaming



arm

Superscalar Processors: Register Renaming

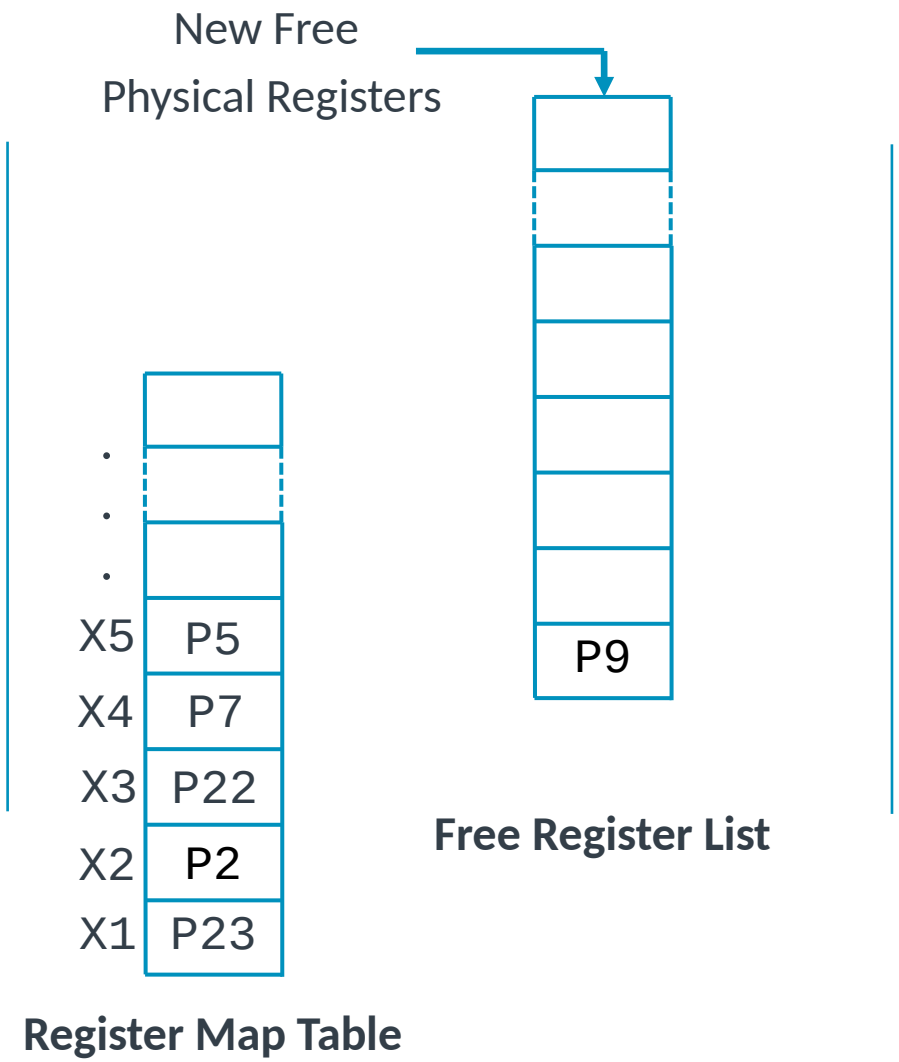


arm

Superscalar Processors: Register Renaming

Before register renaming

```
AND X2, X2, X1
ADD X3, X2, X5
MUL X2, X4, X1
```



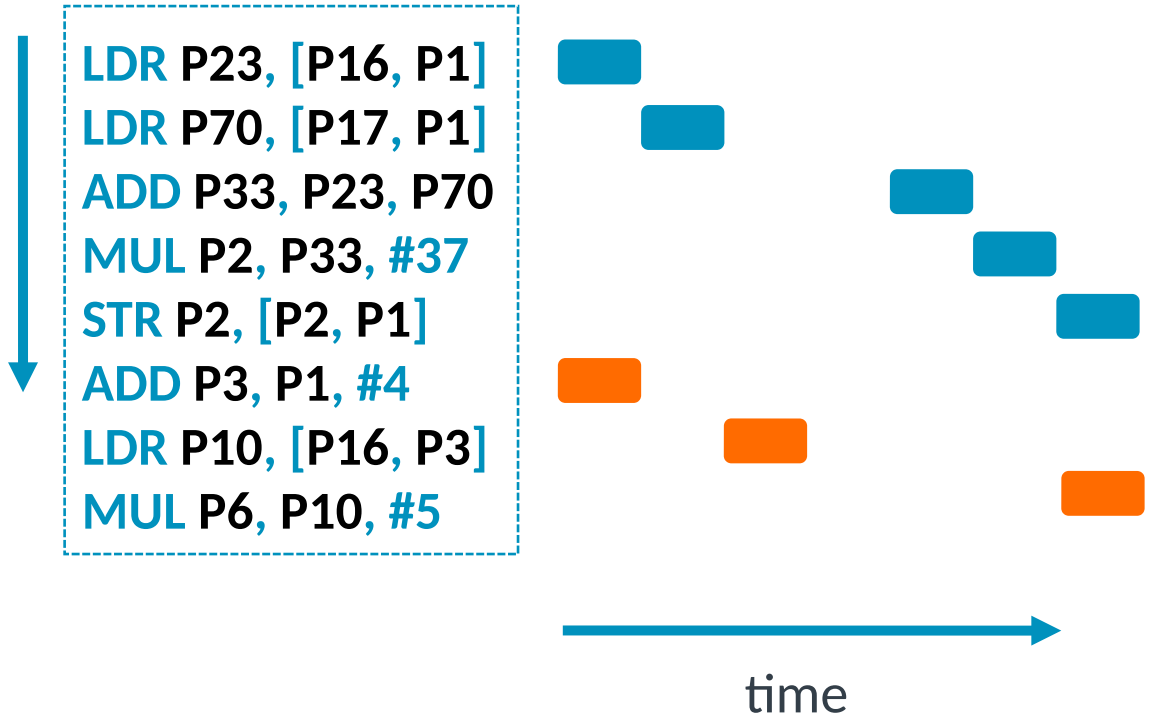
After register renaming

```
AND P17, P34, P23
ADD P22, P17, P5
MUL P2, P7, P23
```


arm

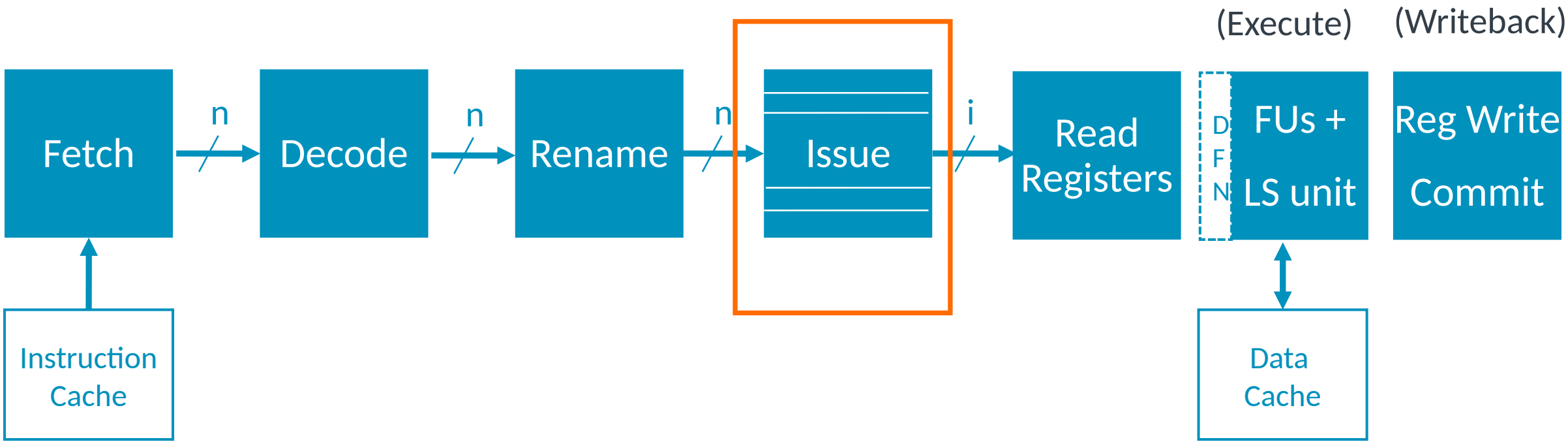
Superscalar Processors: Register Renaming

- Each instruction now has a unique physical destination register.
- All name dependencies have been removed.
- The processor is now free to issue an instruction as soon as its operands are ready and an appropriate FU is free.



arm

A Generic Superscalar Processor

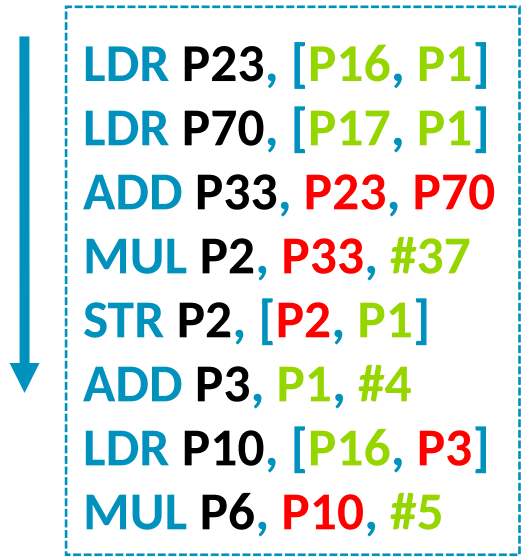


LS unit = Load/Store unit

DFN = Data Forwarding Network

Superscalar Processors: Register Data Flow

- The status of each instruction's operands are read and updated when they enter our issue window.
- We can see that the first two loads are ready to issue and the second ADD.

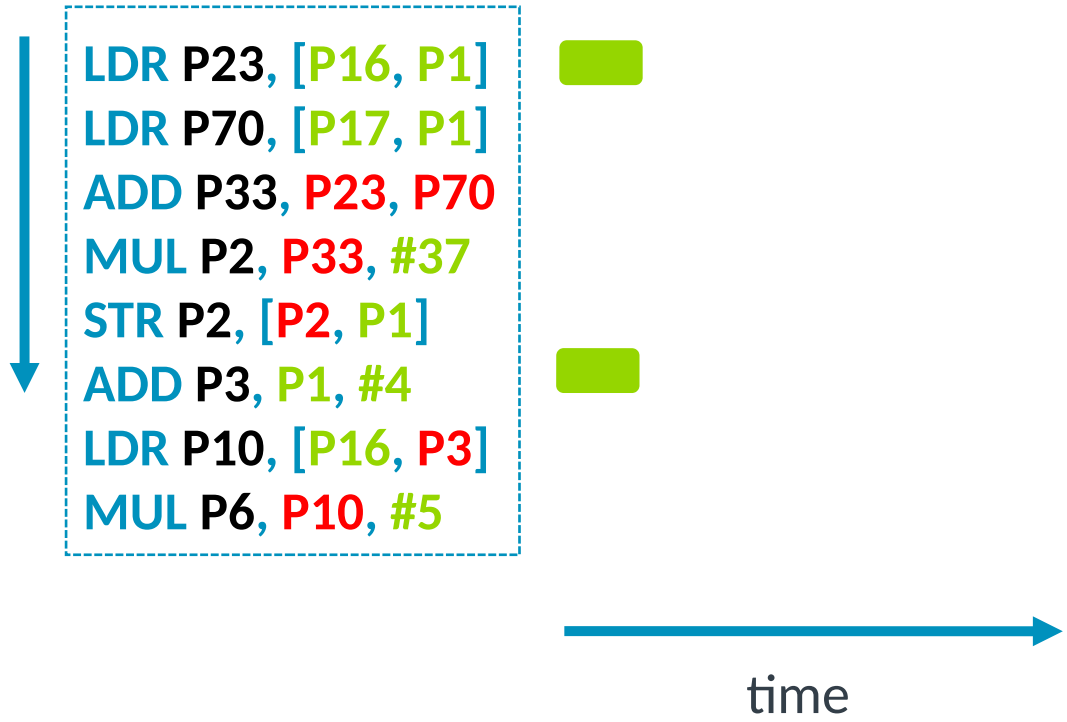


Operands are ready if shown in green and are not available if shown in red; destination registers are shown in black

arm

Superscalar Processors: Register Data Flow

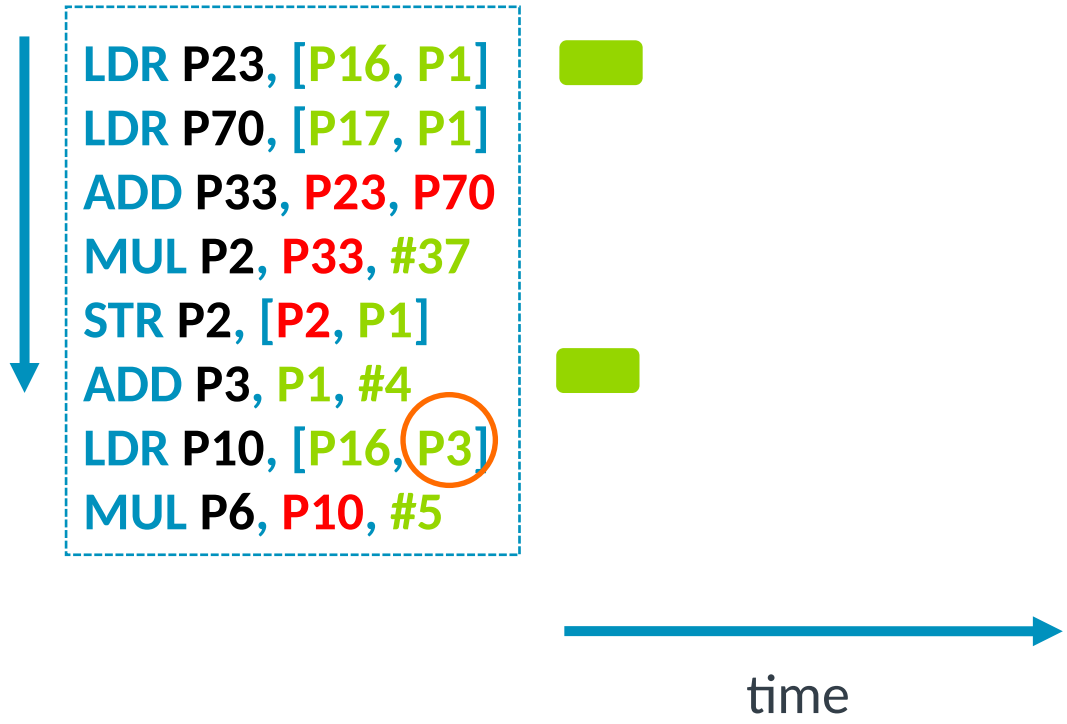
- The first load instruction ("LDR P23…") is issued together with a more recent ADD instruction.



arm

Superscalar Processors: Register Data Flow

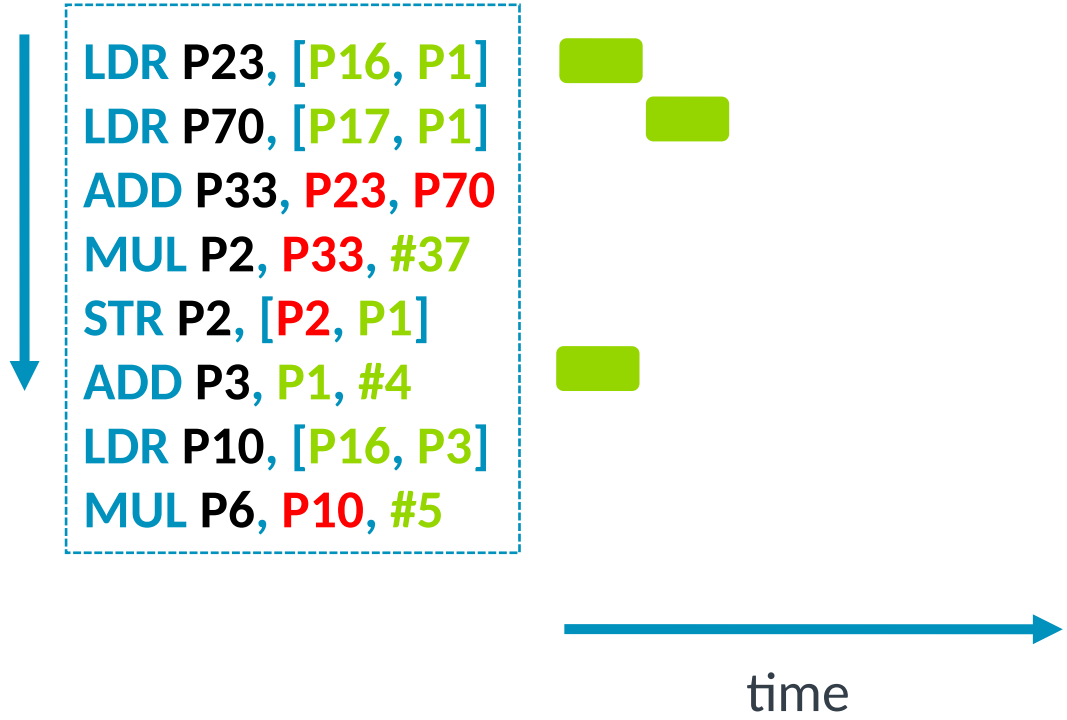
- After the ADD instruction is issued, we update the status of register P3 in any waiting instruction.
- We will also broadcast the register identifier P23 in a similar way.
- As the load's latency will be greater than a single cycle, we delay this operation for a few clock cycles.



arm

Superscalar Processors: Register Data Flow

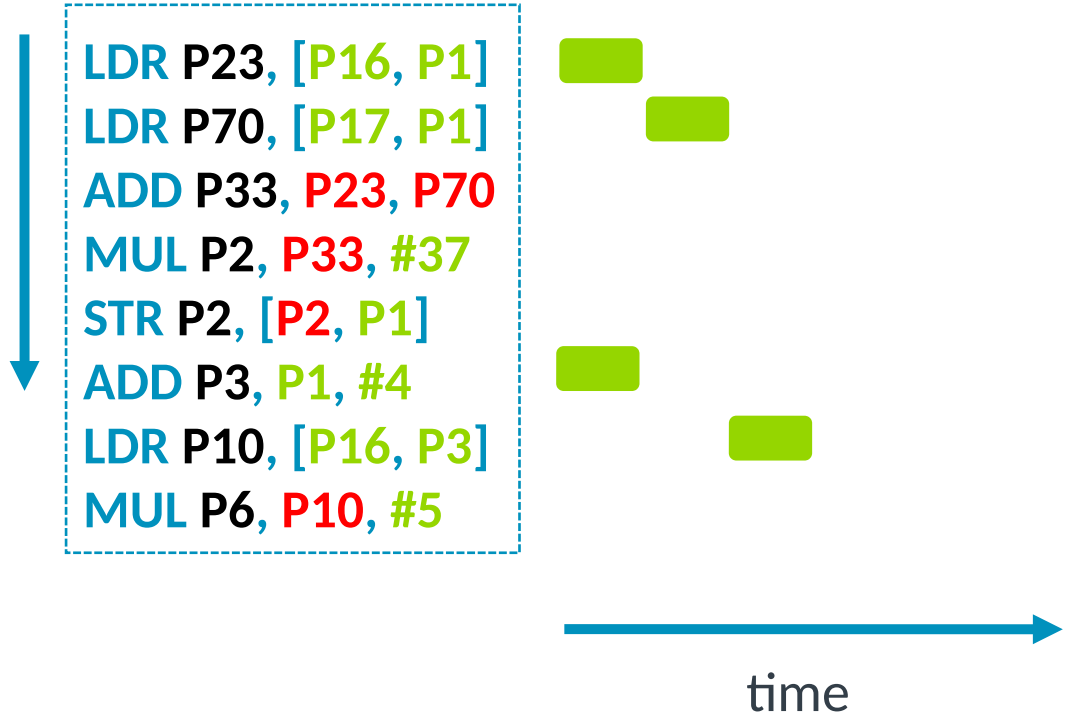
- The second load is now issued.



arm

Superscalar Processors: Register Data Flow

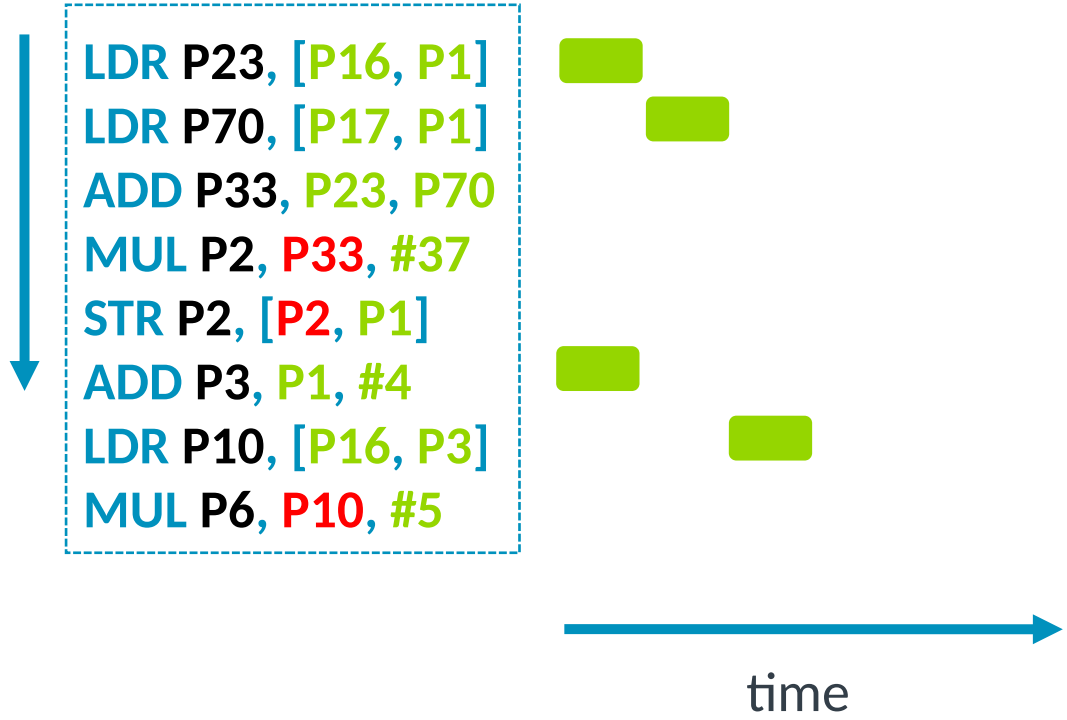
- And now the third load instruction is issued.



arm

Superscalar Processors: Register Data Flow

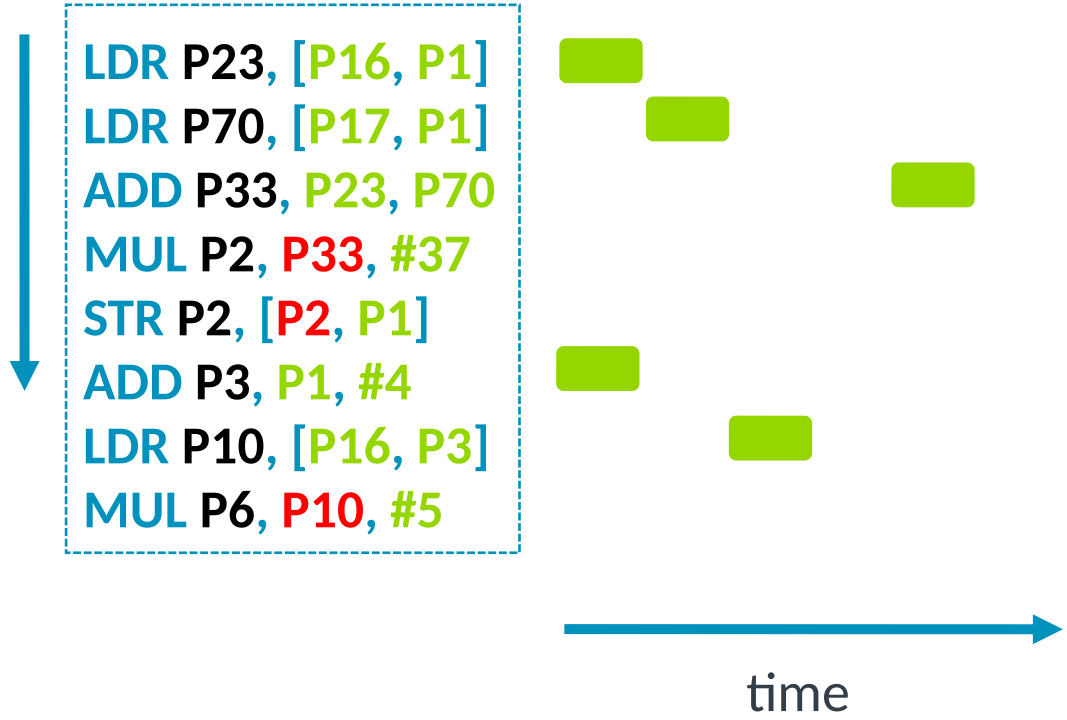
- We now expect the result of the first load soon, so we update those instructions waiting for result P23.
- Then, on the next clock cycle, P70.



arm

Superscalar Processors: Register Data Flow

- The first ADD instruction can now be issued.
- We continue in this way until all the instructions are executed.



arm

Superscalar Processors: Register Data Flow

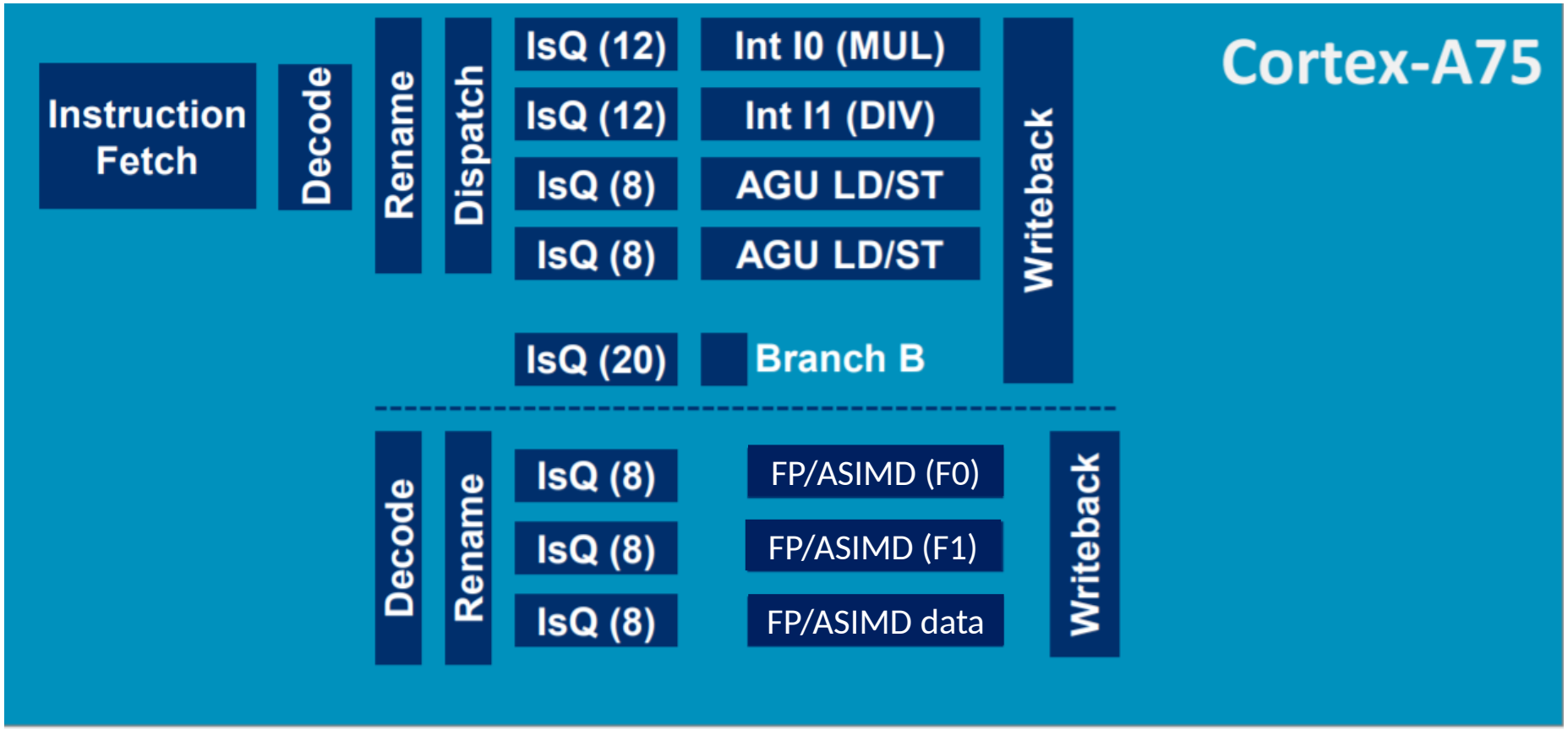
- The issue window is implemented as a large memory-like structure.
- When an instruction is issued, its destination register is broadcast to all waiting instructions (perhaps after a short delay for longer latency operations).
- **Wakeup phase:** the waiting instructions compare the broadcast destination registers with their own operands. When the register identifiers match, the operand is marked as ready.
- **Selection and issue phase:** select as many ready instructions as possible and issue them to waiting FUs.

Superscalar Processors: Instruction Issue

Design choices:

- Centralized or distributed instruction window
- Compacted or non-compacted
- Position of register file? Before or after instruction window

Example: A Distributed Instruction Window (Arm Cortex-A75)

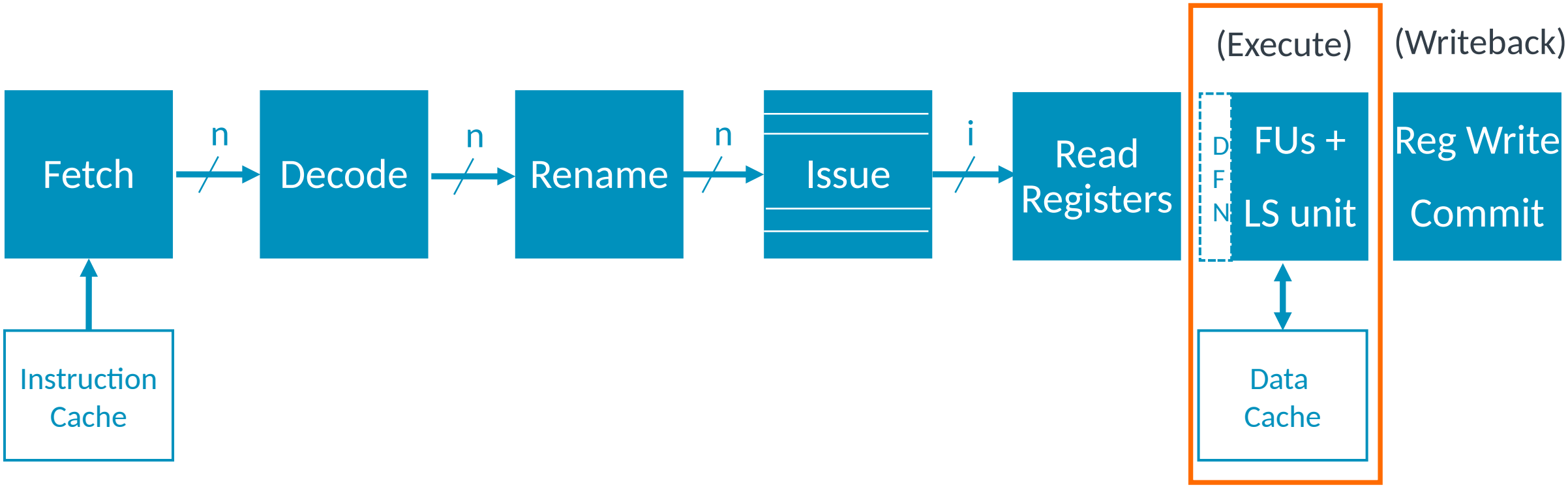


Instruction fetch can provide at most 4 instructions per cycle,
3-way superscalar, 11-13 stage integer pipeline

64KB Instruction cache, 7 independent issue queues.
(Armv8.2-A architecture)

arm

A Generic Superscalar Processor

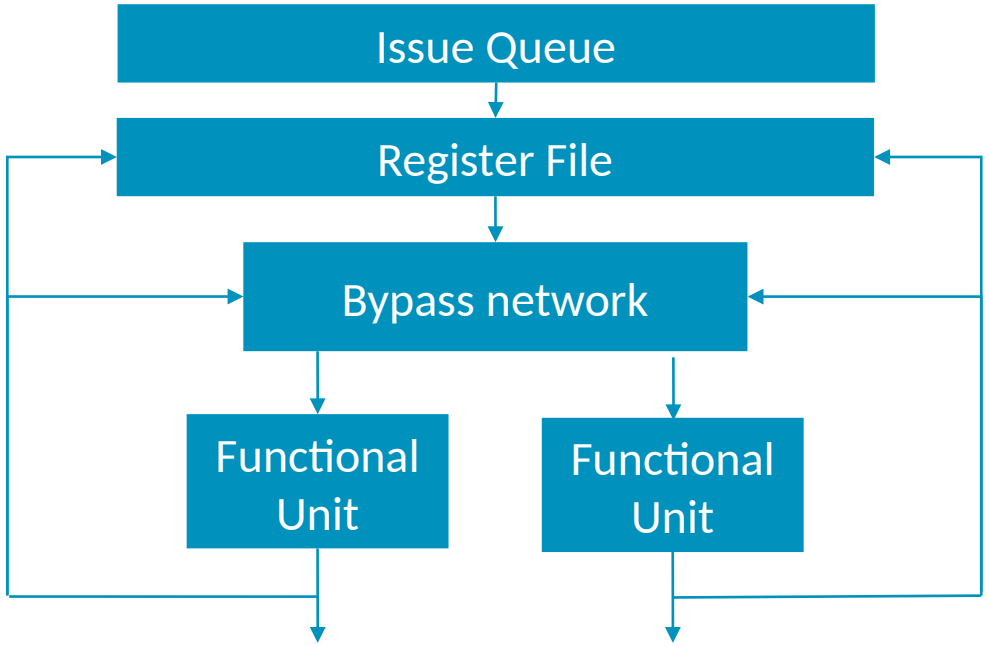


LS unit = Load/Store unit

DFN = Data Forwarding Network

Superscalar Processors: Data Forwarding (Bypass) Network

- Data forwarding in a scalar pipeline is relatively simple, consisting of a few extra buses and multiplexers.
- In a superscalar processor, we have many parallel functional units and may need to forward any recently generated results to the input of any functional unit. For example:



arm

Superscalar Processors: Loads and Stores

Memory-carried data dependencies

- Scheduling loads and stores is complicated by the fact that a load and store may access the same memory location.
- If we blindly execute these instructions out-of-order, we may violate memory-carried data dependencies.

Superscalar Processors: Loads and Stores

Stores and speculative execution

- Store operations cannot be undone. The implication of this is that:
  - To provide precise exceptions, we must ensure stores are not performed until we know that no earlier instruction will raise an exception.
  - We should not execute stores that are “speculative,” i.e., an earlier branch has been predicted, but we are yet to confirm if the prediction was correct.

Superscalar Processors: Loads and Stores

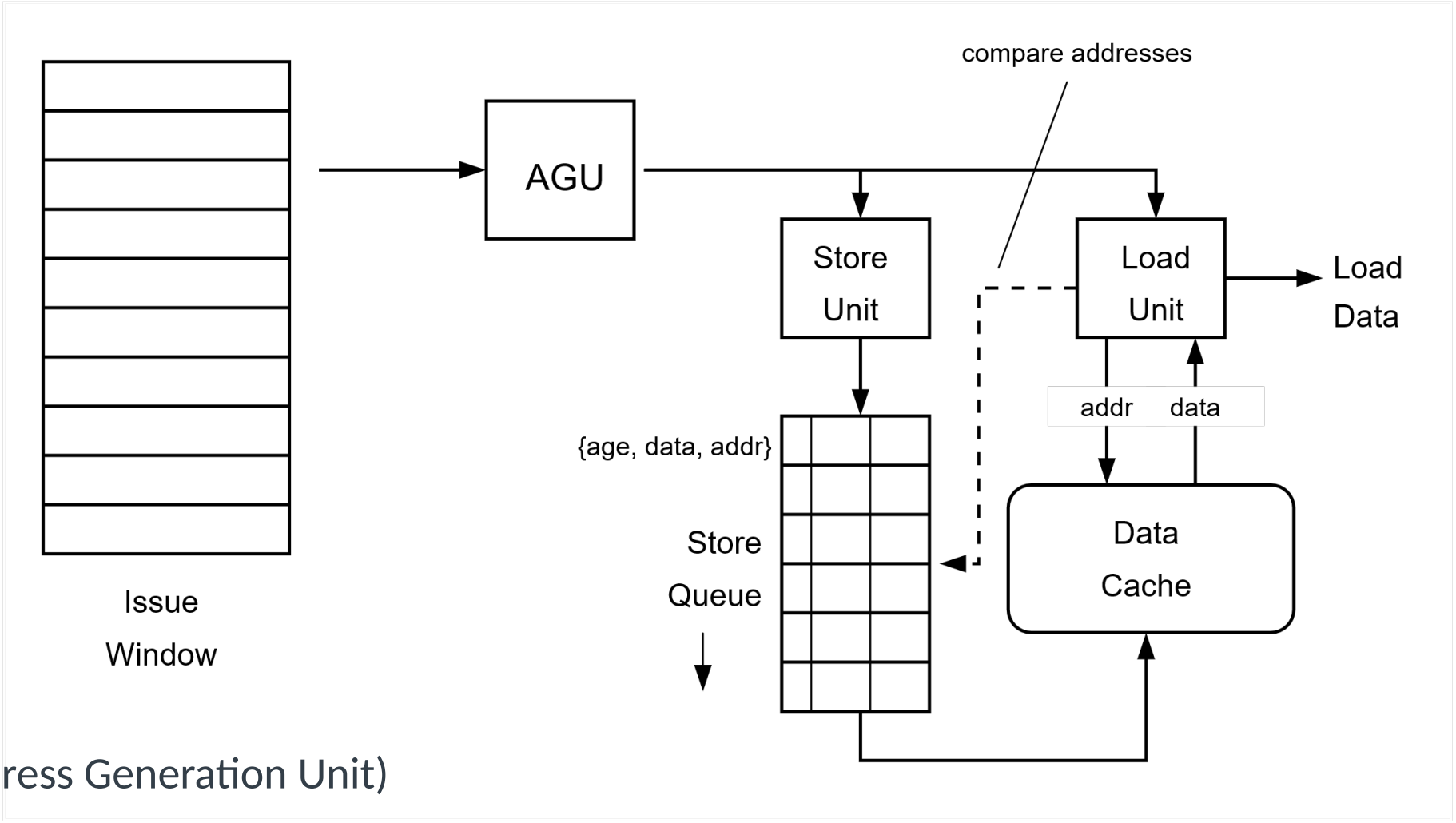
We will only permit stores to execute in program order.

They will wait in the “store queue” until they are the oldest unexecuted instruction.

So we have:

1. Issue load/stores out-of-order to Address Generation Unit (AGU).
2. Buffer stores and only execute them in program order.
3. For loads, check **all** addresses of older stores. If any match or addresses are unknown, stall load; otherwise, it may access the data cache (**load-bypassing**).

Superscalar Processors: Loads and Stores



(AGU – Address Generation Unit)

Superscalar Processors: Loads and Stores

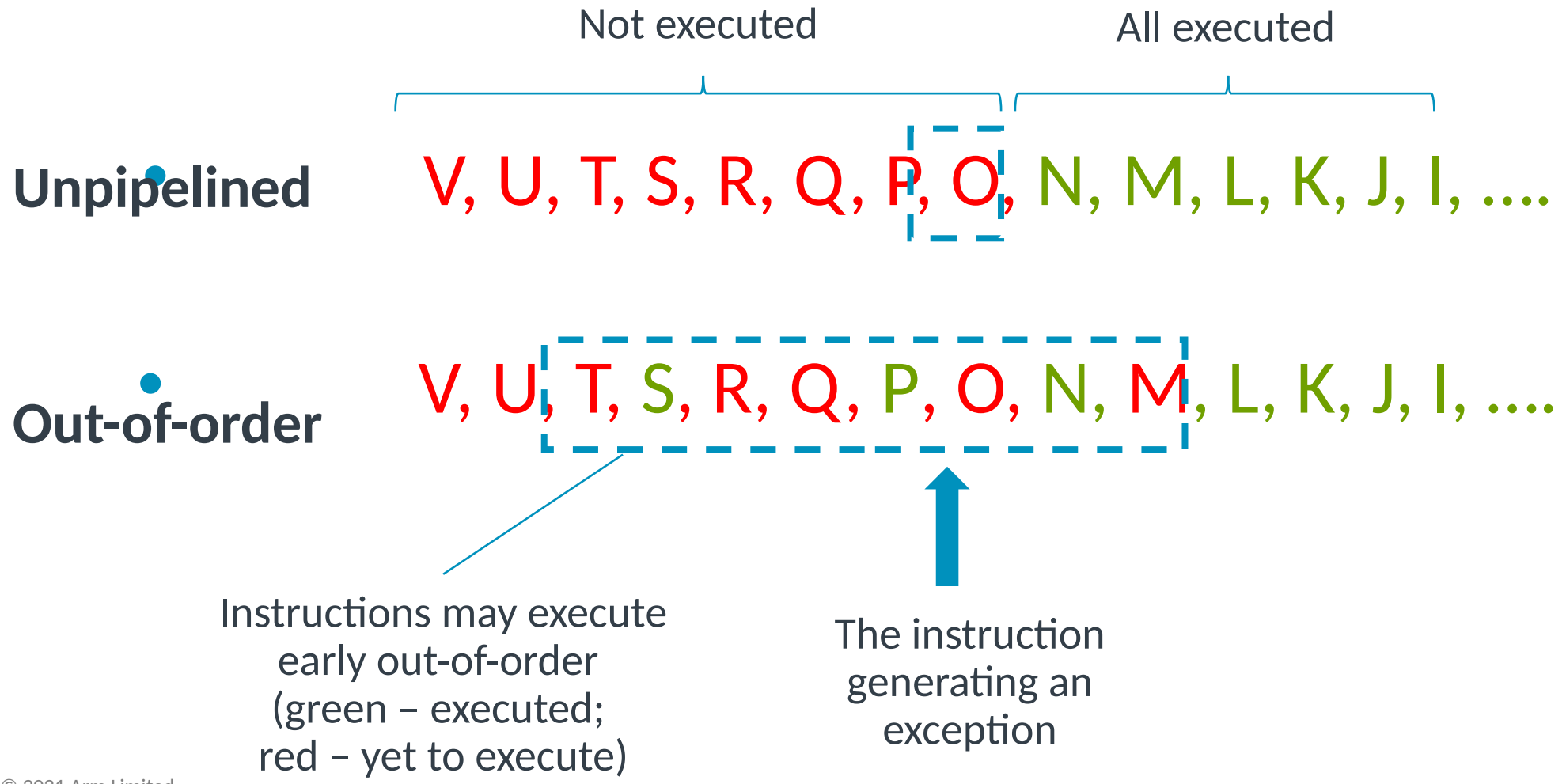
High-performance superscalar processors go further than this:

- Store-to-load forwarding
  - Allows data to be forwarded directly from a pending store to a load instruction
- Speculative loads
  - Allow loads to access the data cache speculatively even when there are older stores that have not calculated their addresses

Superscalar Processors: Exceptions and Speculation

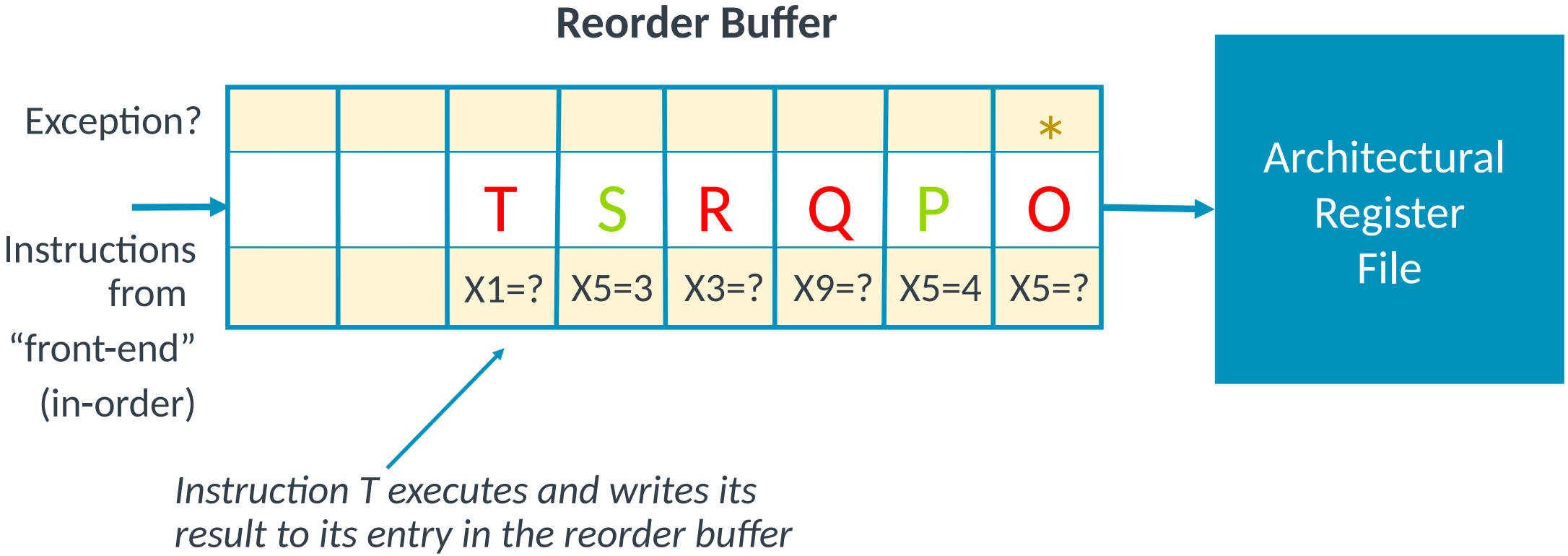
- Mispredicted branches and exceptions force us to roll back state.
- To support precise exceptions, we also need to track the architectural state of the processor.
  - i.e., the state corresponding to the **in-order** execution of instructions

Superscalar Processors: Exceptions and Speculation



Superscalar Processors: The Reorder Buffer

A simple technique for providing precise exceptions is to only update our register file in program order. We will buffer any results generated out-of-order in a **reorder buffer.**



arm

Superscalar Processors: The Reorder Buffer

Committing instructions

When an instruction reaches the end of the reorder buffer, we know all earlier instructions have completed. At this point, we can:

- Update our (architectural) register file.
- Check if branches have been mispredicted.
  - If so, flush the reorder buffer and re-execute the branch.
- Check if the instruction needs to raise an exception.
  - If it does, flush the reorder buffer and raise the exception.
- Signal that store operations can write to the data cache.

Superscalar Processors: The Reorder Buffer

We now have two potential sources when trying to obtain the latest value of a register: the reorder buffer and our architectural register file.

To ensure instructions receive the correct data, we rename registers to reorder buffer entries (or entries in the register file).

This is done at the rename stage either by:

(1) maintaining an explicit mapping table determining the latest source of a particular logical register

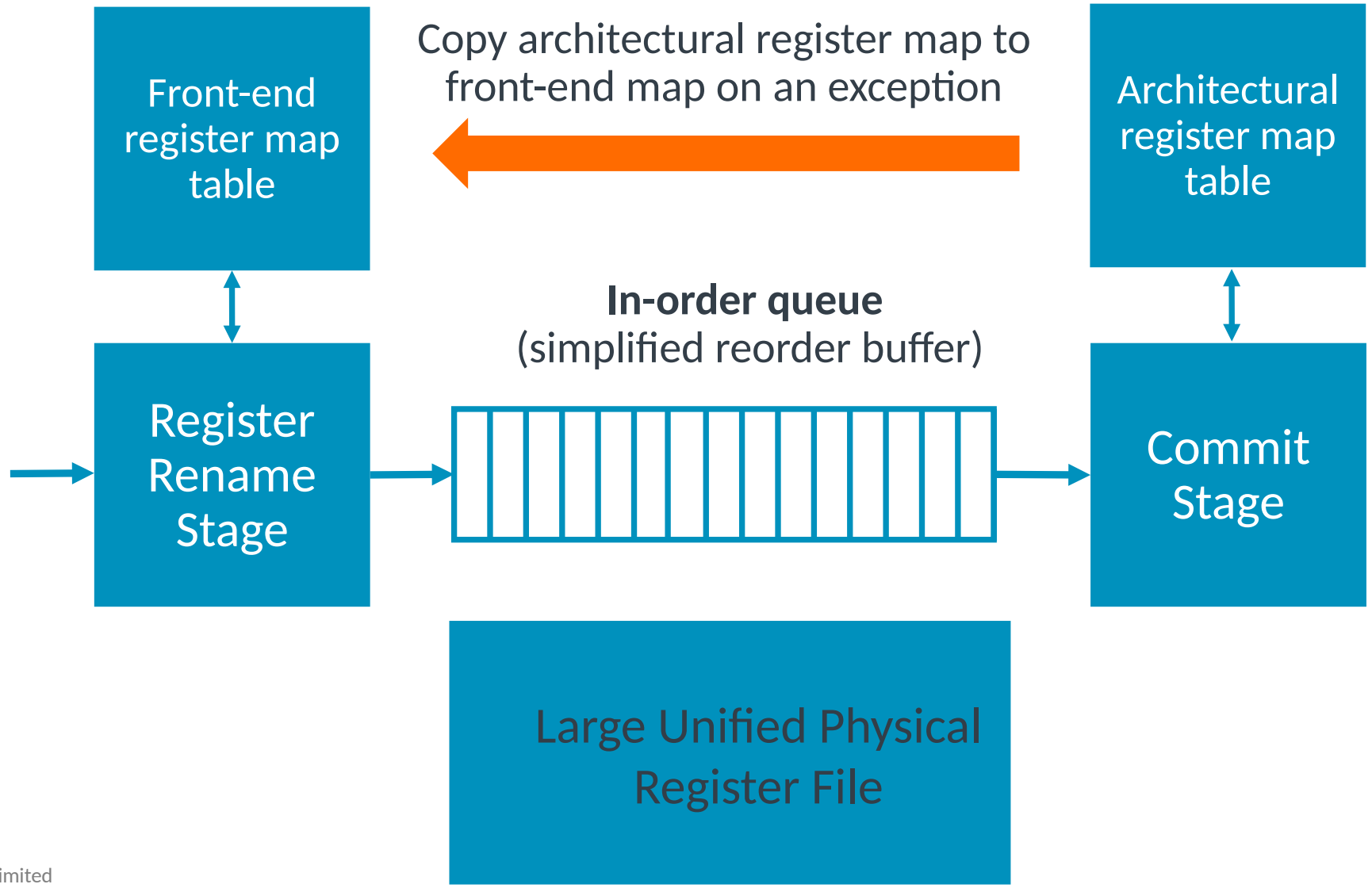
(2) searching the reorder buffer for the latest version of a logical register

Each instruction's destination register is renamed to the assigned reorder buffer slot.

Superscalar Processors: Unified Register File Approach

- The reorder buffer complicates our design by introducing a new source of operands.
- An alternative approach is to maintain a large physical register file that holds all results.
- Here, we rename registers, as described earlier, and maintain a register mapping table (that holds the mapping of architectural register names to physical ones).

Superscalar Processors: Unified Register File Approach



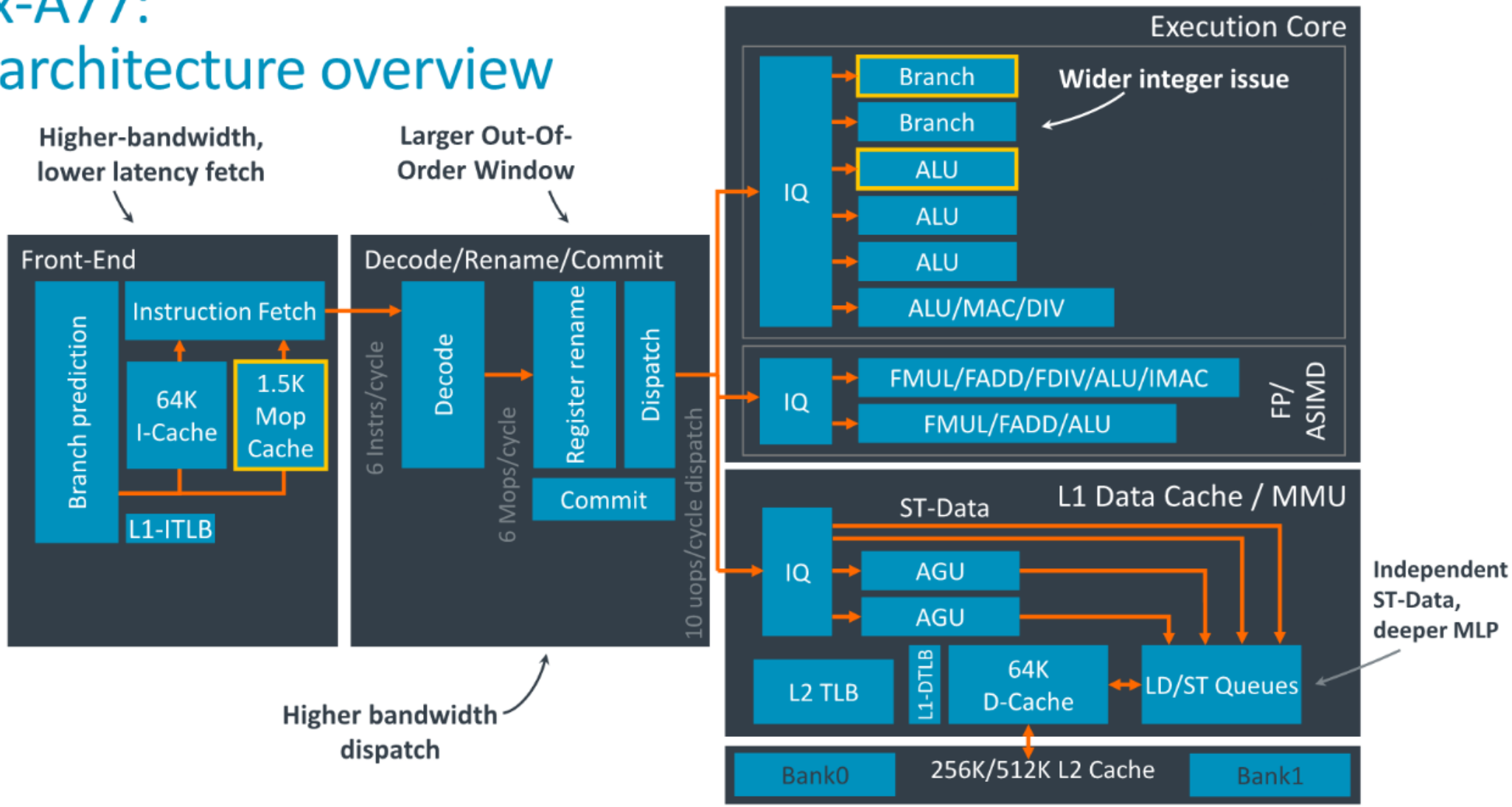
Superscalar Processors: Handling Mispredicted Branches

- Some processors attempt to handle mispredicted branches before they commit.
- This can be achieved by saving the register map table each time we encounter a branch.
- As soon as we detect a mispredicted branch, we can quickly restore the mapping that existed before the branch was predicted.
- This restoration of state is itself speculative, an older branch or exception may cause us to roll back execution again.

Example: Putting It All Together (Cortex-A77, 2019)

- The Cortex-A77 can fetch and decode 4 instructions/cycle.
- It can issue (dispatch) up to 10 uops/cycle to the integer, FP, and load/store units.
- The branch mispredict penalty is 10 cycles in the best case.
- The out-of-order windows size and reorder buffer hold 160 instructions.
- The target clock frequency is between 2.6 and 3 GHz.

Cortex-A77: Microarchitecture overview



Limits to Superscalar Processors

Ultimately, the performance of a superscalar processor is limited by:

- Increasing hardware cost of extracting more ILP
- Memory bandwidth
- Limits to branch prediction and caches
- Interconnect scaling
- Power consumption