R265: Advanced Topics in Computer Architecture

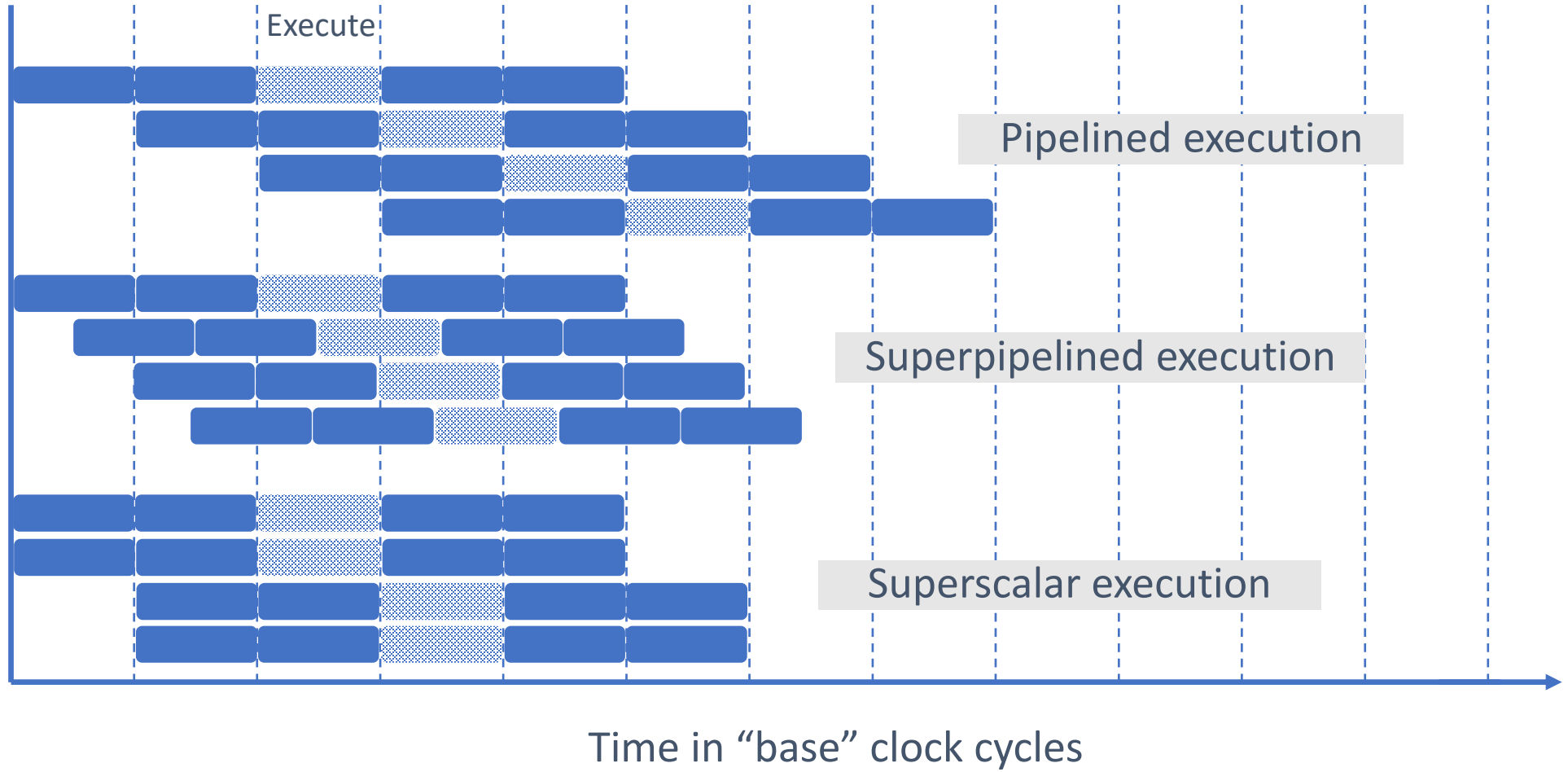
Seminar 2: State-of-the-art Processor Design

Robert Mullins

This lecture

- Examine design of modern microprocessors
- Superscalar and multicore techniques
- SoC design

Superpipelined and superscalar processors



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.

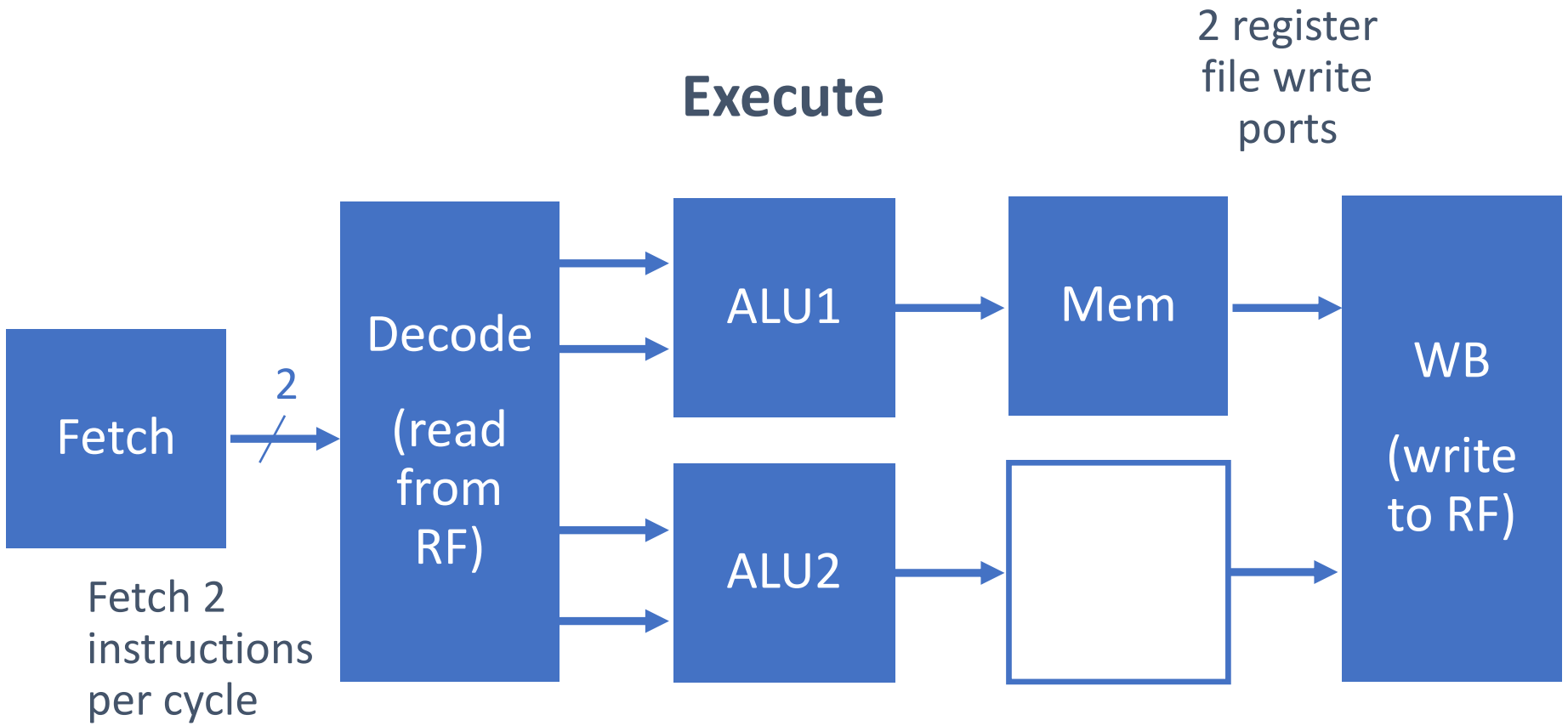
What extra hardware will this processor require?

- 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

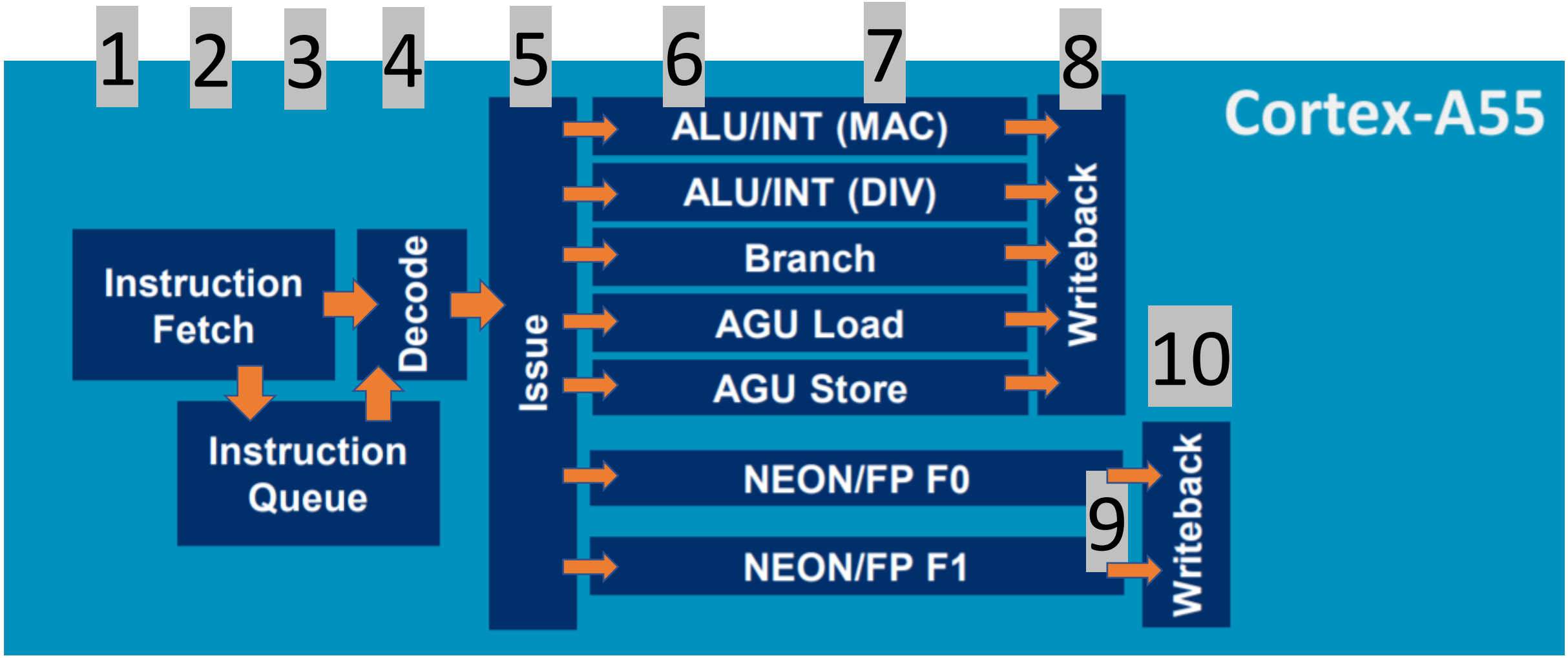
- We can create a simple (2-way) superscalar processor with a few changes to our scalar pipeline
- We will fetch and decode multiple instructions per cycle
- Instructions are sent to functional units in program order (in-order issue)
- We will issue and execute instructions in parallel if we can
- If 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 Cortex-A55



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

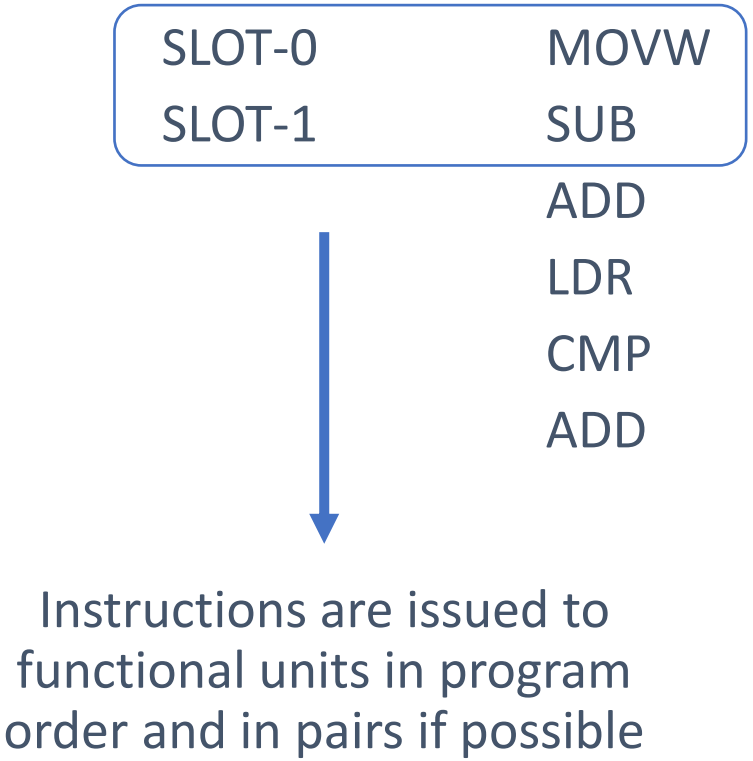
Issue slots

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 FU resource that has not been duplicated)
- The FU resource required by one of the instructions is busy

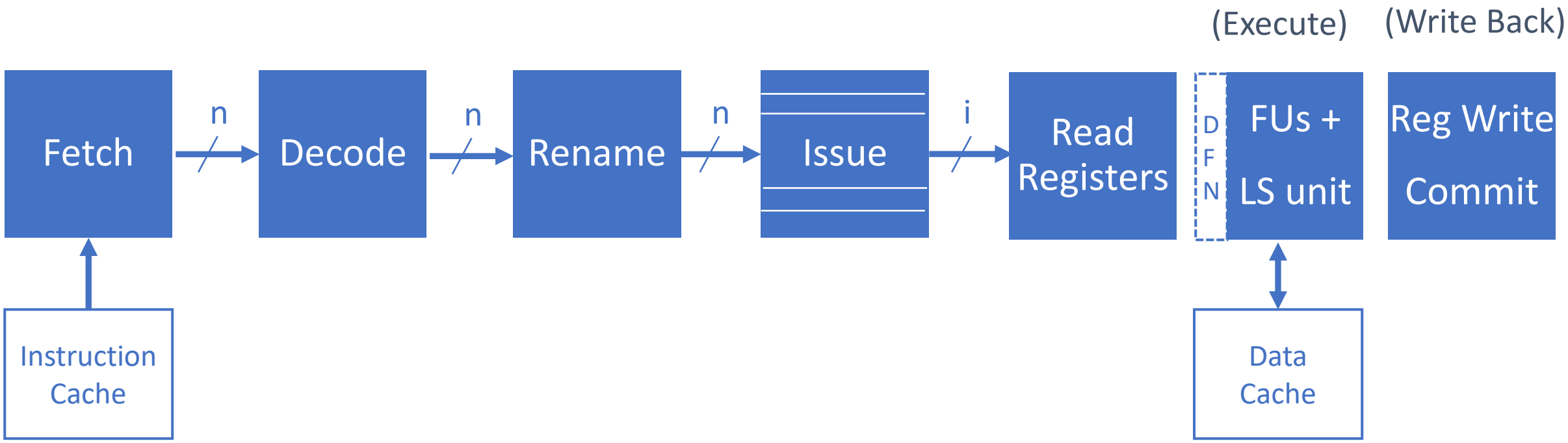


Exposing and exploiting more ILP

To expose more ILP we need to consider:

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

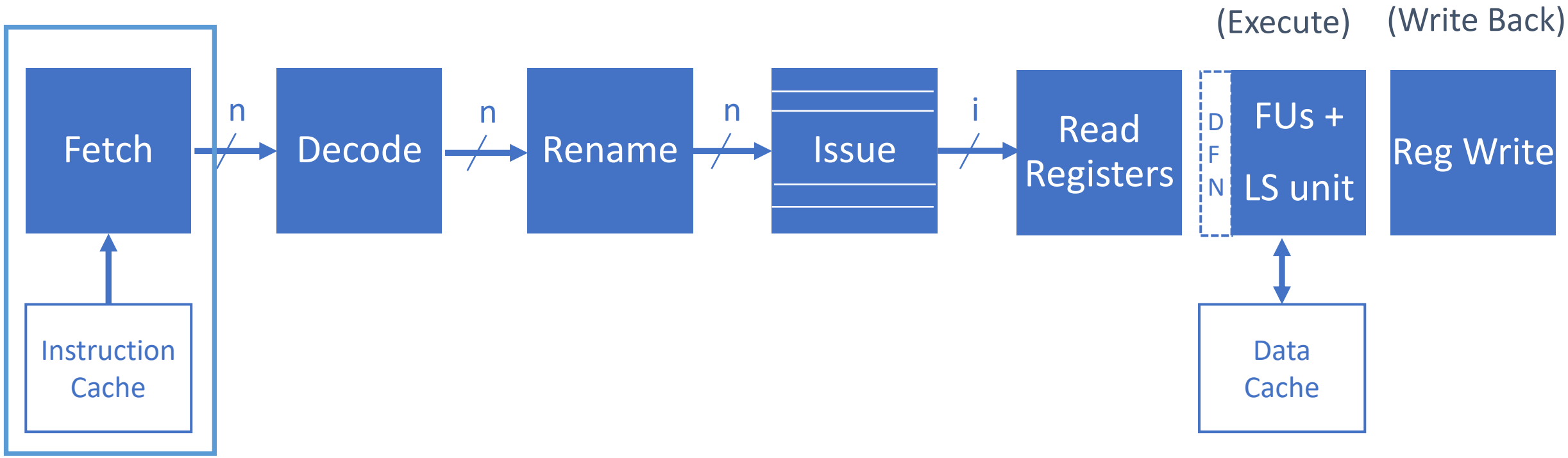
A generic superscalar processor



LS unit = Load/Store unit

DFN = Data Forwarding Network

A generic superscalar processor



LS unit = Load/Store unit

DFN = Data Forwarding Network

Superscalar processors: instruction fetch

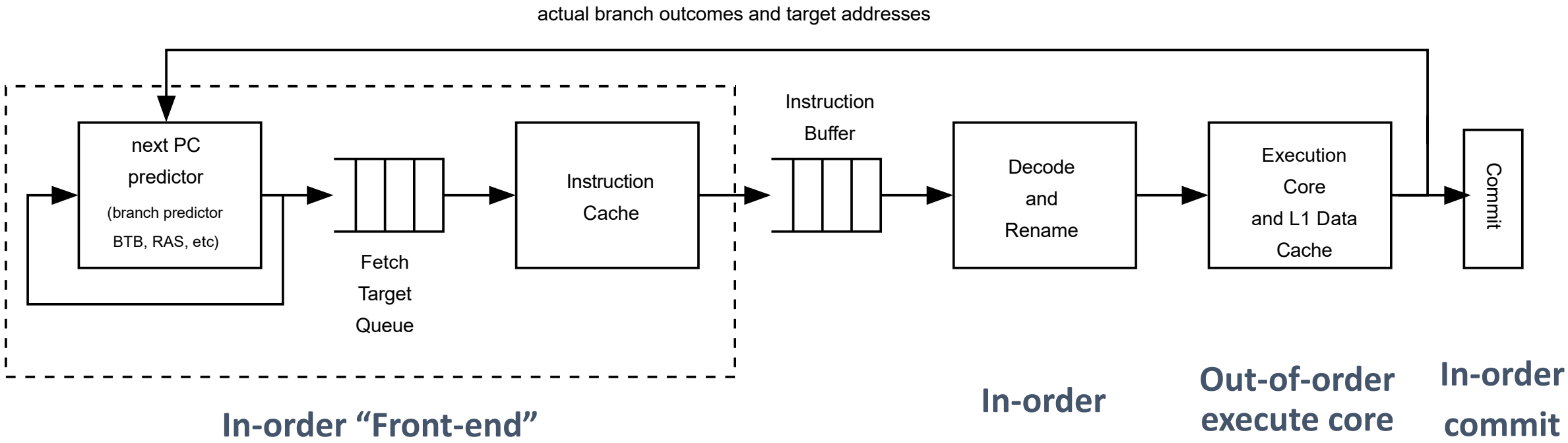
Our superscalar pipeline cannot process instructions faster than they are supplied, so maintaining a good instruction fetch rate is very important.

Potential limitations:

- Branch prediction accuracy
- Instruction cache performance
- Instruction fetch and alignment issues

Superscalar processors: instruction fetch

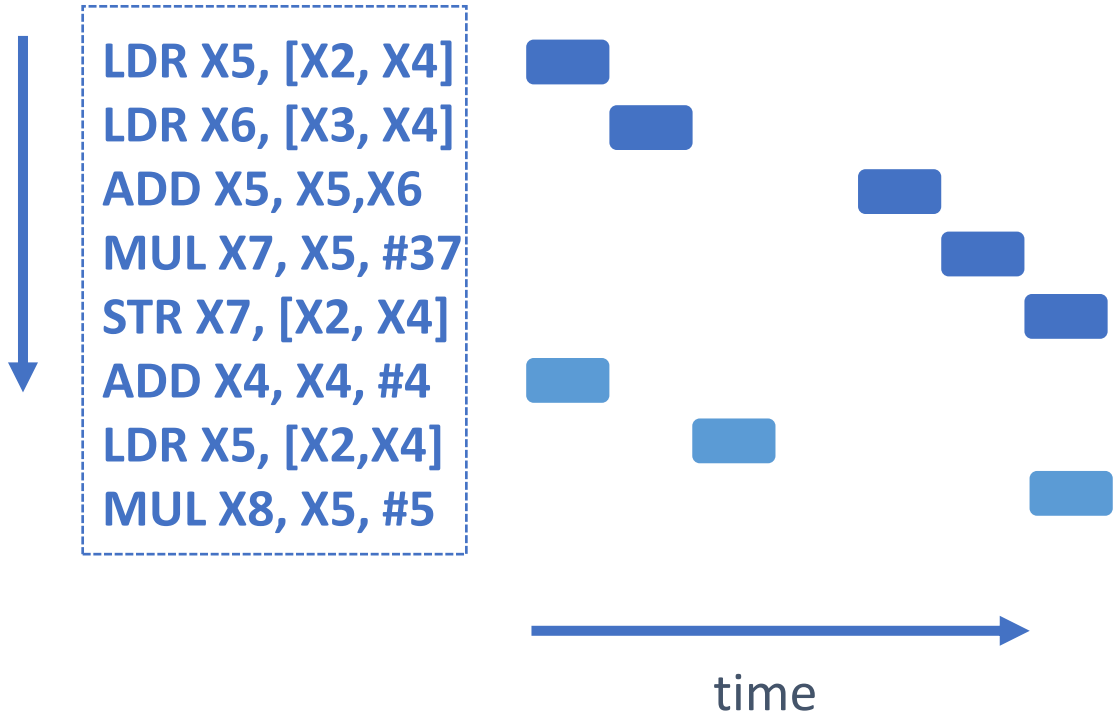
Our instruction fetch (front-end) can be decoupled from the part of the processor that actually executes instructions. The aim here is to run ahead, fill the instruction buffer and help keep our execution units fed.



Note: each block may represent multiple pipeline stages

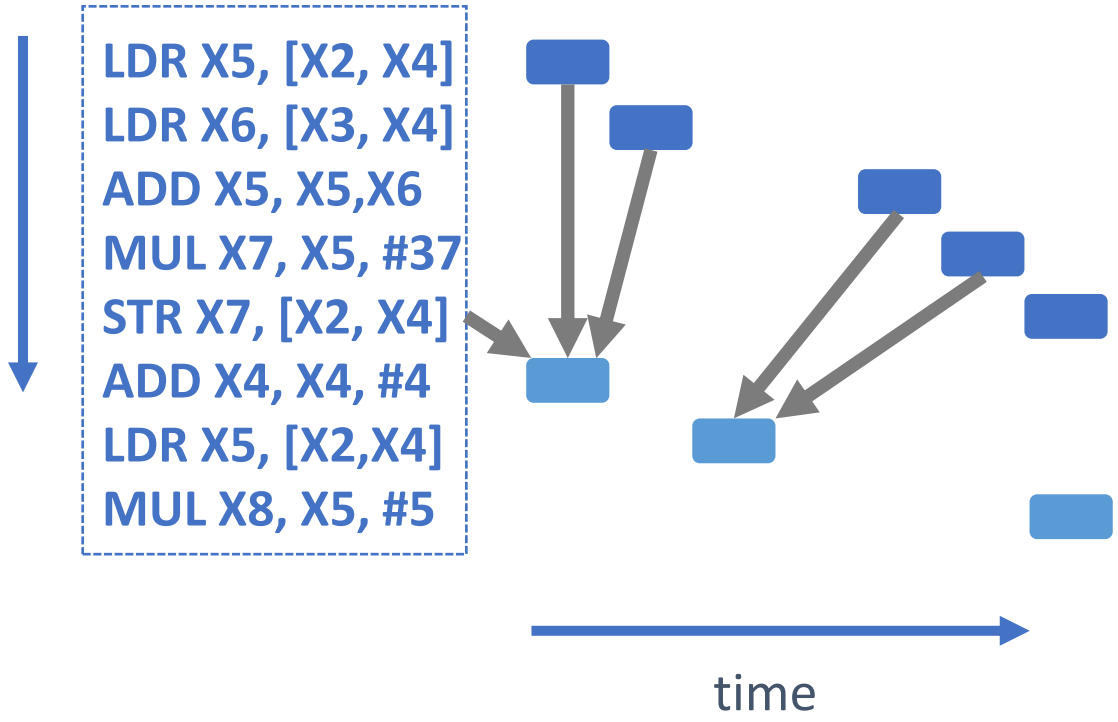
Superscalar processors: register renaming

- 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



Superscalar processors: register renaming

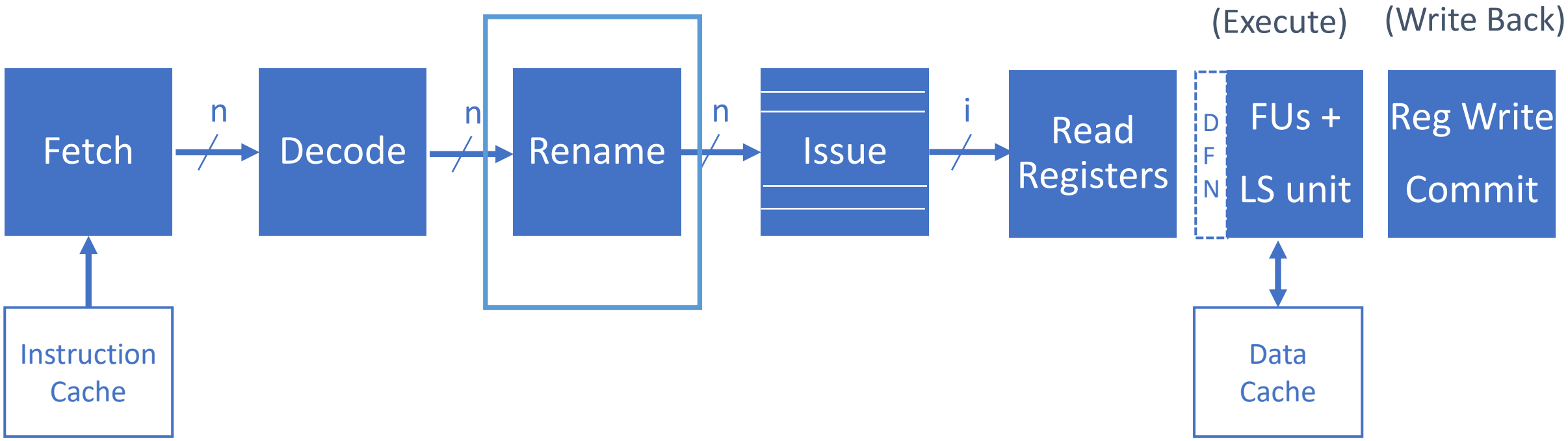
- In practice, name (or false) dependencies may limit our ability to perform this out-of-order instruction issue
- These are present as the compiler must reuse a limited number architectural (or logical) register names
- The arrows highlight the false dependencies present in this code snippet



Superscalar processors: register renaming

- Register renaming may be performed in hardware at run-time
- It provides each instruction with a unique physical destination register
- It removes 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

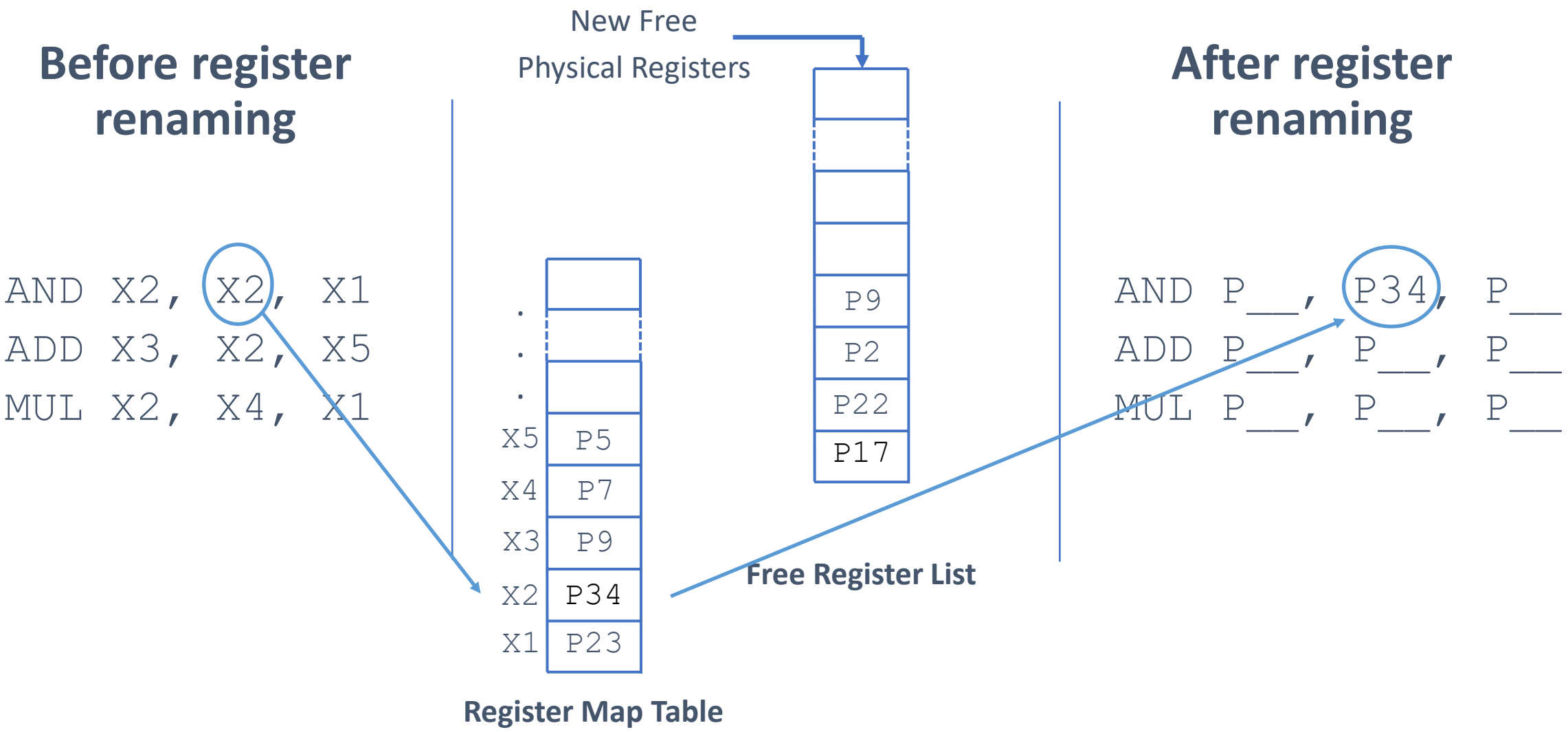
A generic superscalar processor



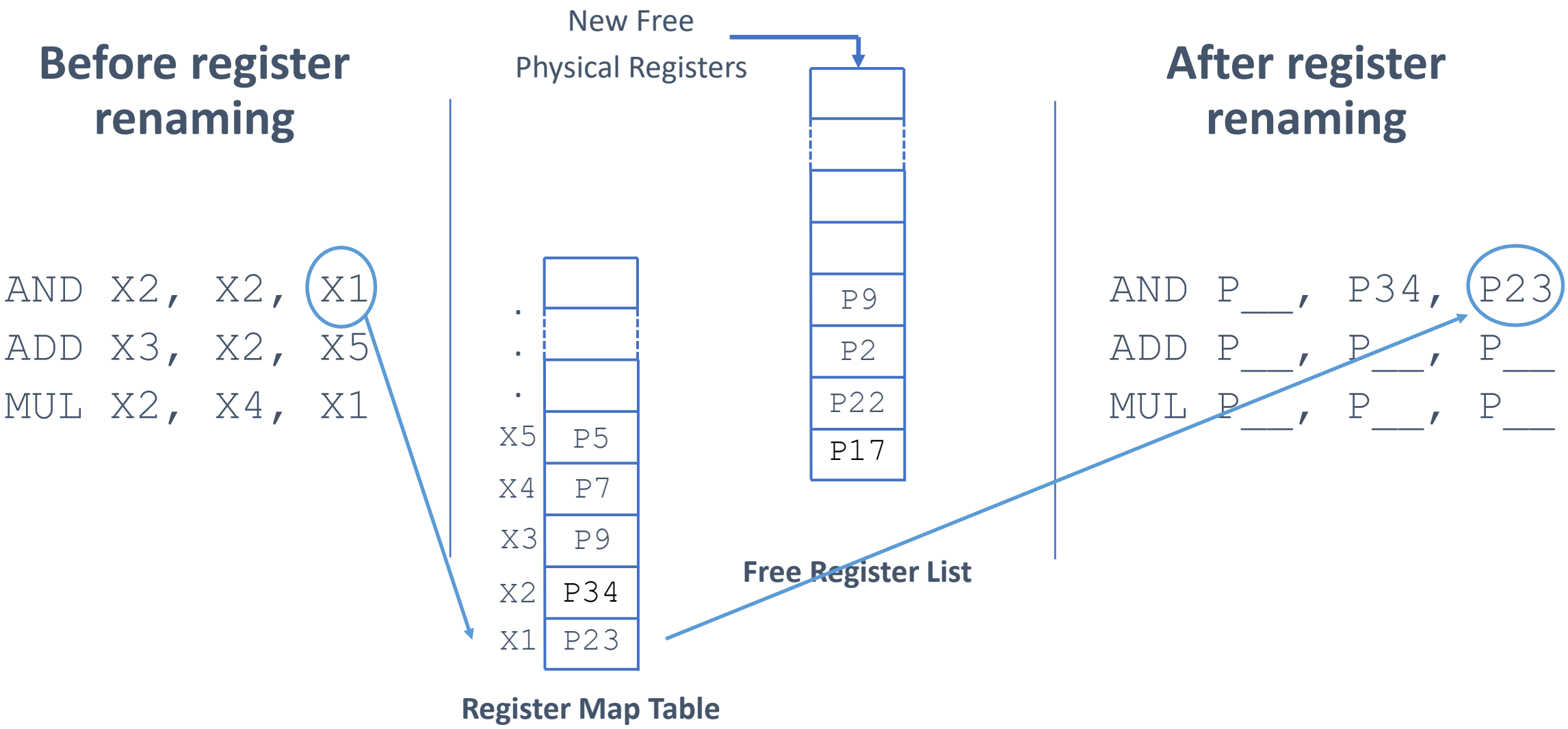
LS unit = Load/Store unit

DFN = Data Forwarding Network

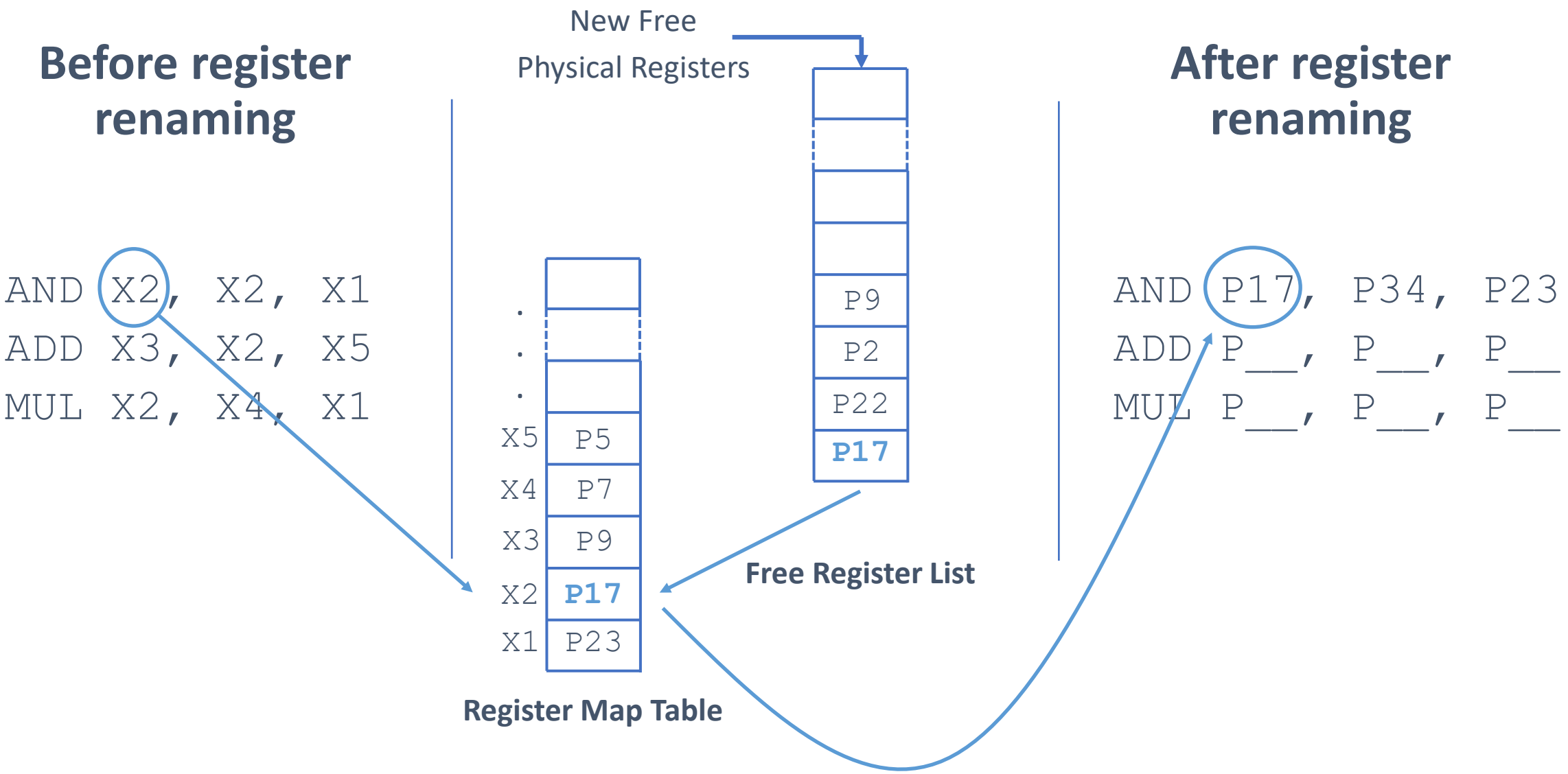
Superscalar processors: register renaming



Superscalar processors: register renaming



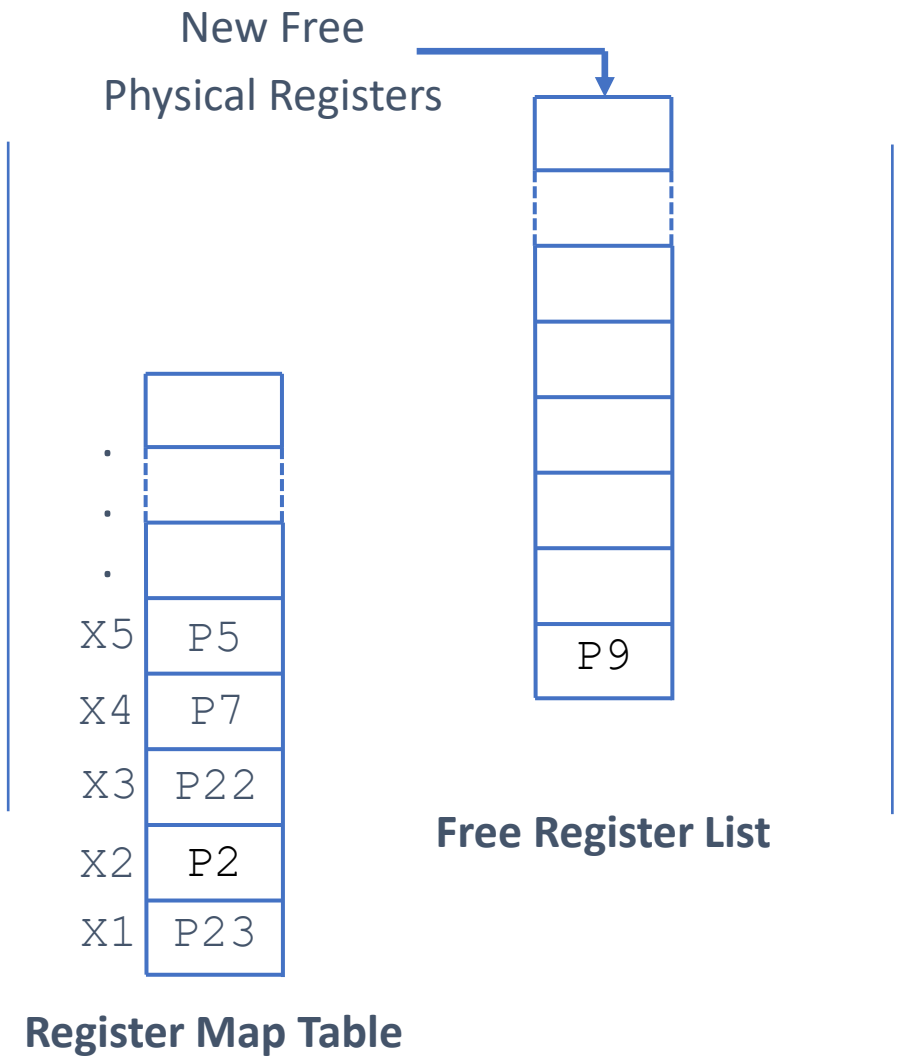
Superscalar processors: register renaming



Superscalar processors: register renaming

Before register renaming

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

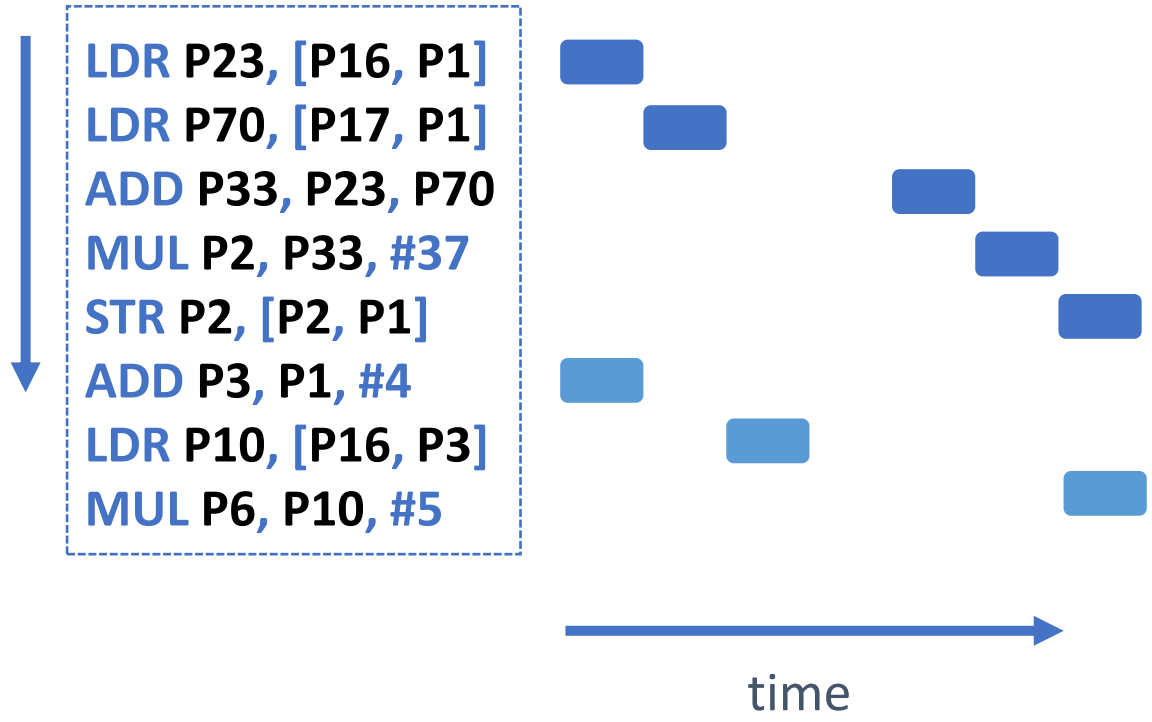


After register renaming

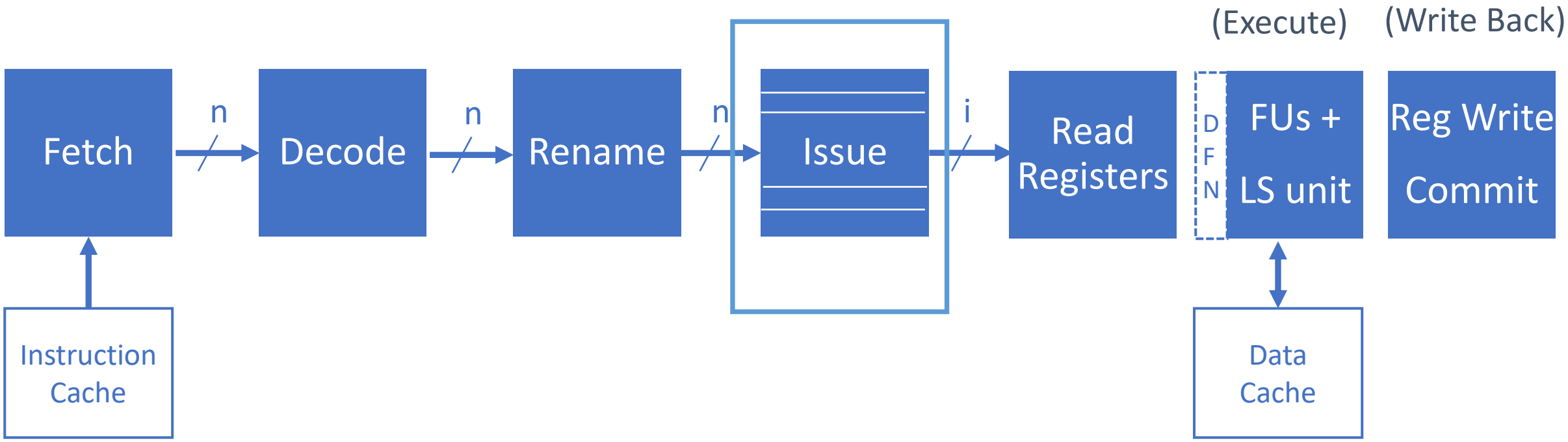
```
AND P17, P34, P23
ADD P22, P17, P5
MUL P2, P7, P23
```

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



A generic superscalar processor

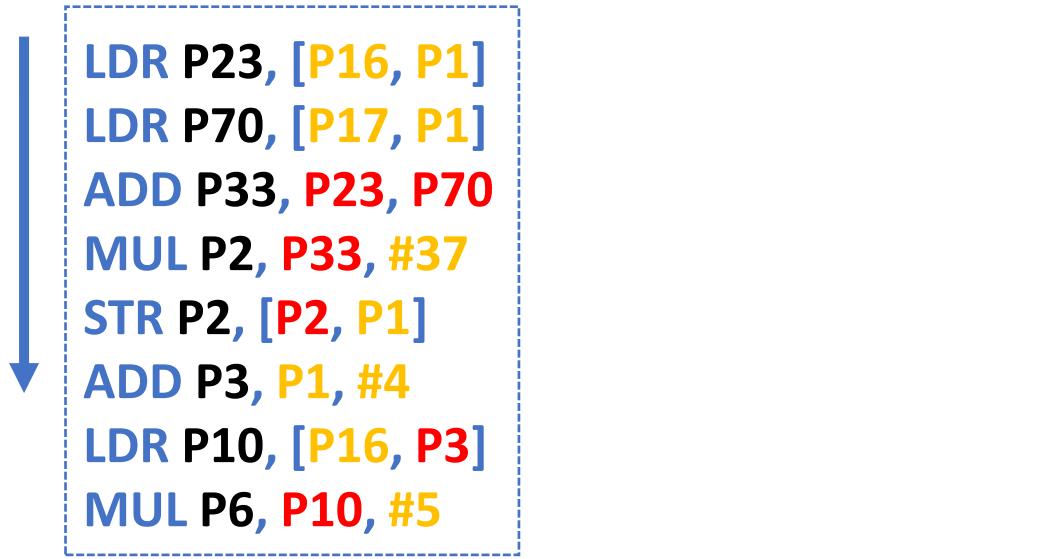


LS unit = Load/Store unit

DFN = Data Forwarding Network

Superscalar processors: register data flow

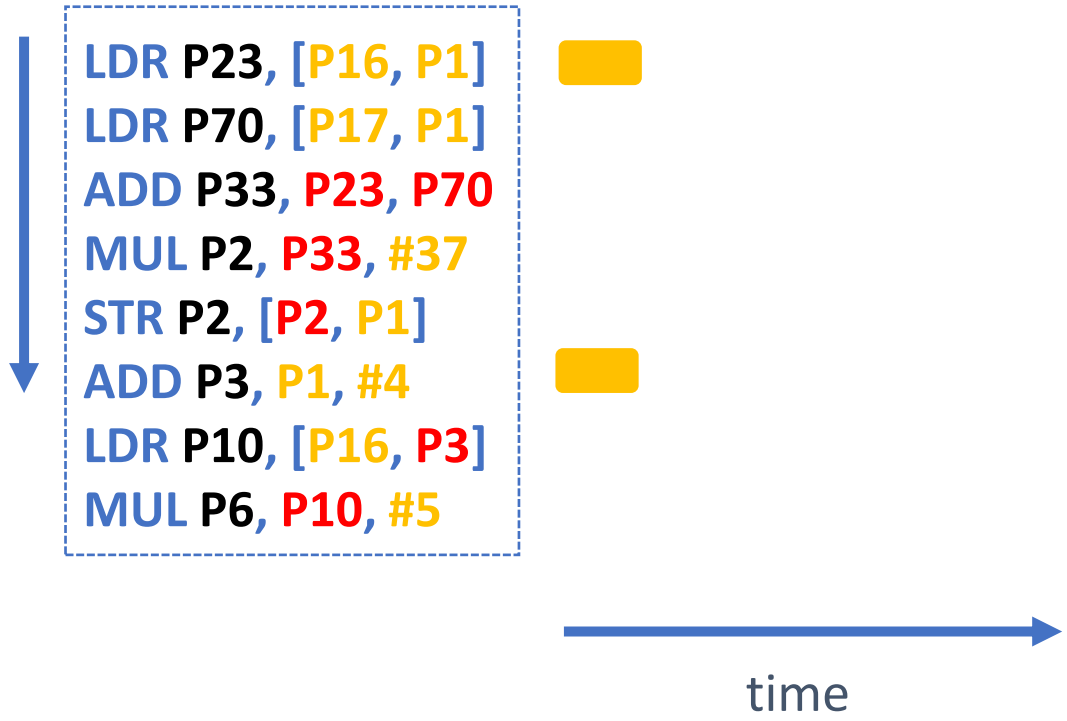
- The status of each instruction's operands are read and updated when they enter our issue window
- We can see that the first two loads are ready to issue and the second ADD



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

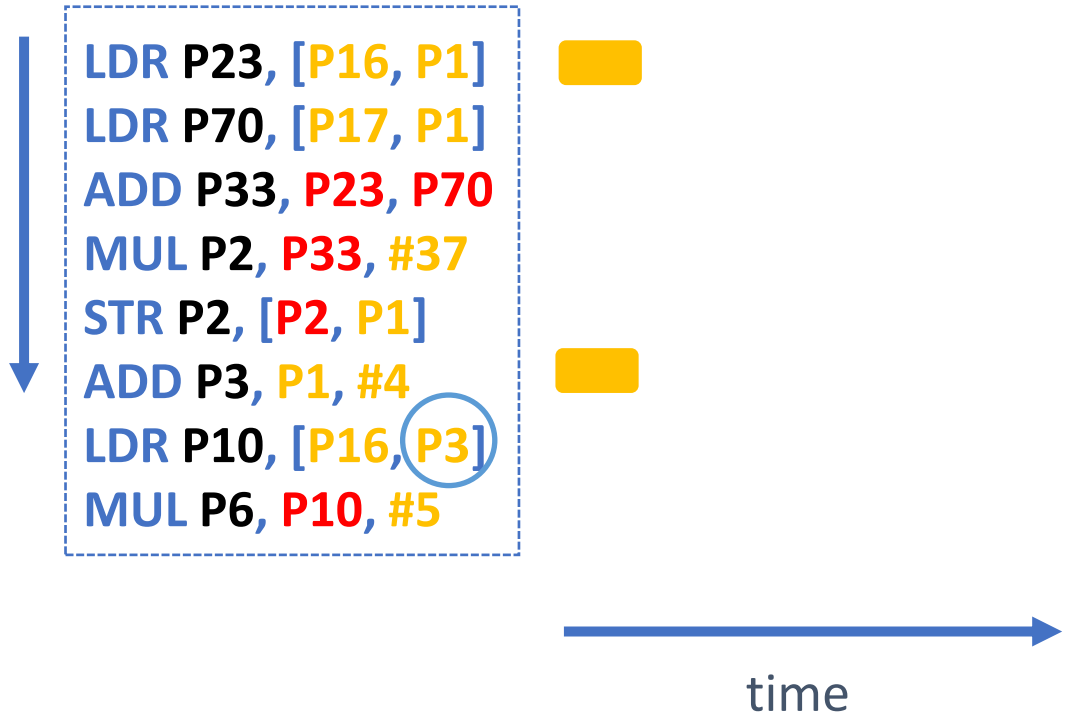
Superscalar processors: register data flow

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



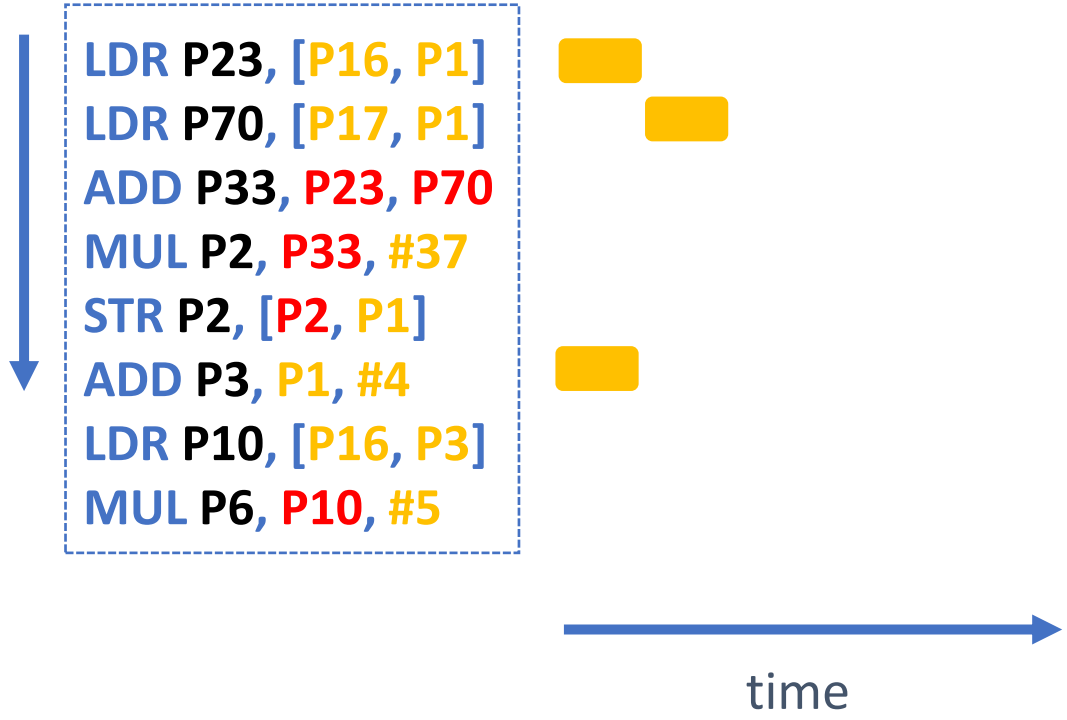
Superscalar processors: register data flow

- 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



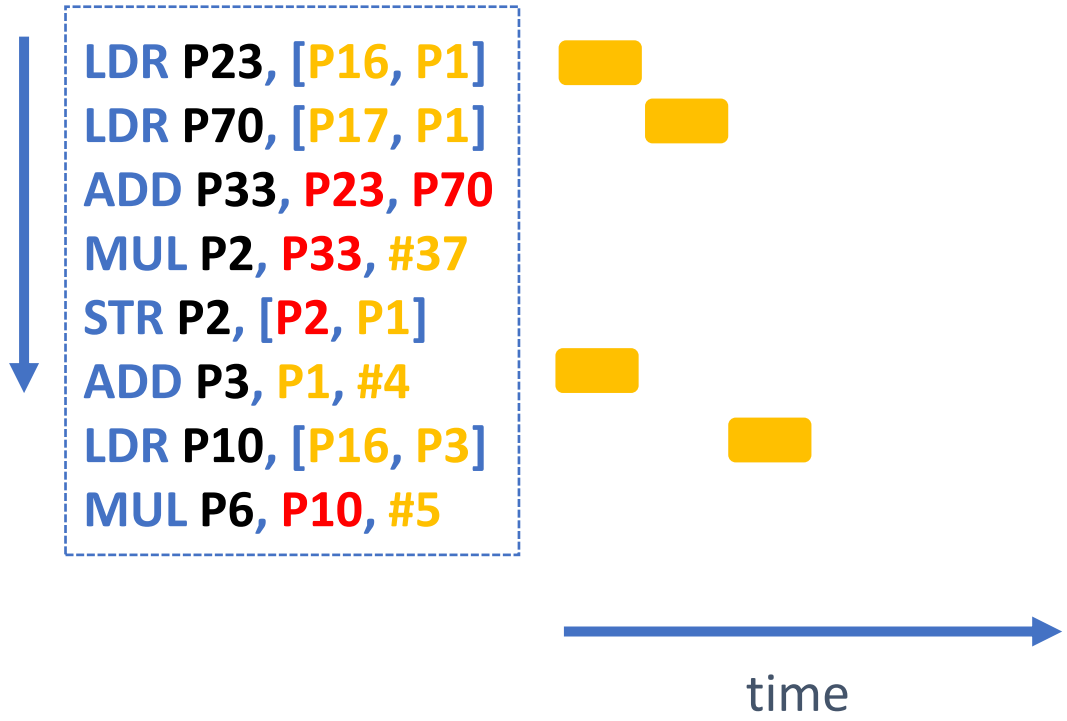
Superscalar processors: register data flow

- The second load is now issued



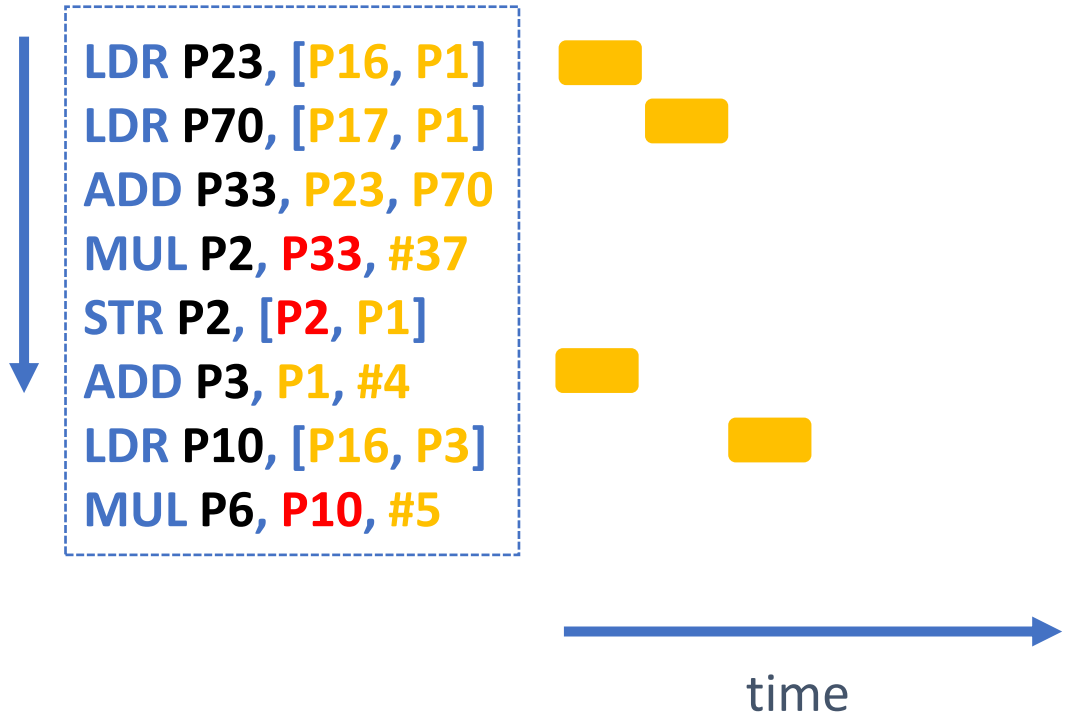
Superscalar processors: register data flow

- And now the third load instruction is issued



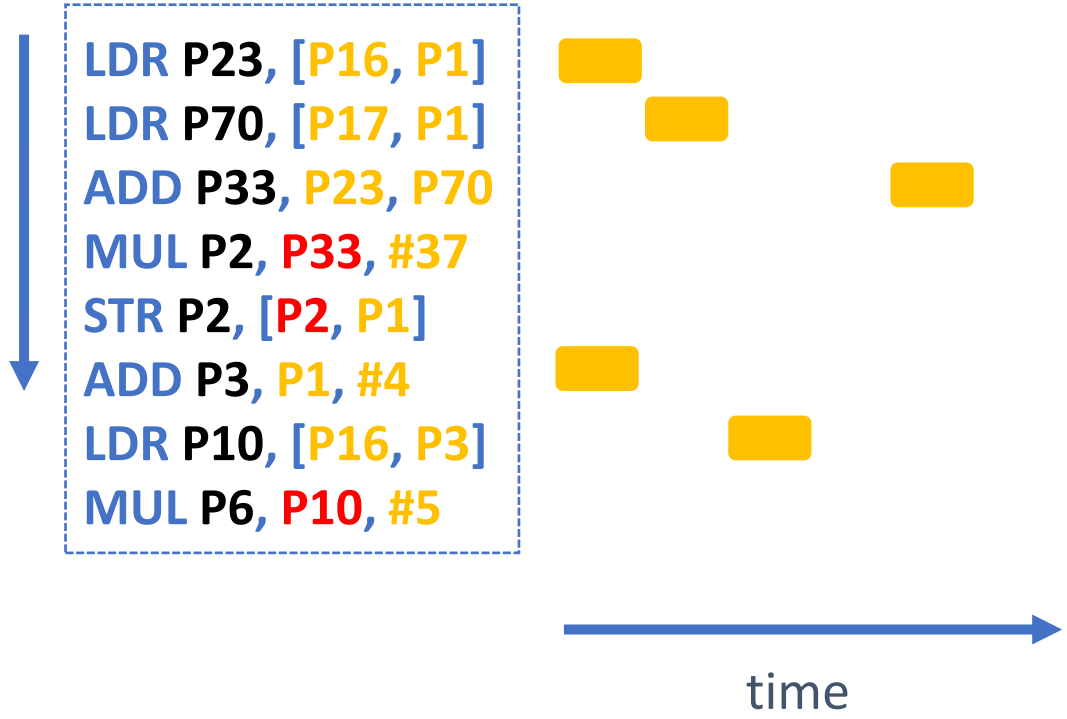
Superscalar processors: register data flow

- 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



Superscalar processors: register data flow

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



Superscalar processors: register data flow

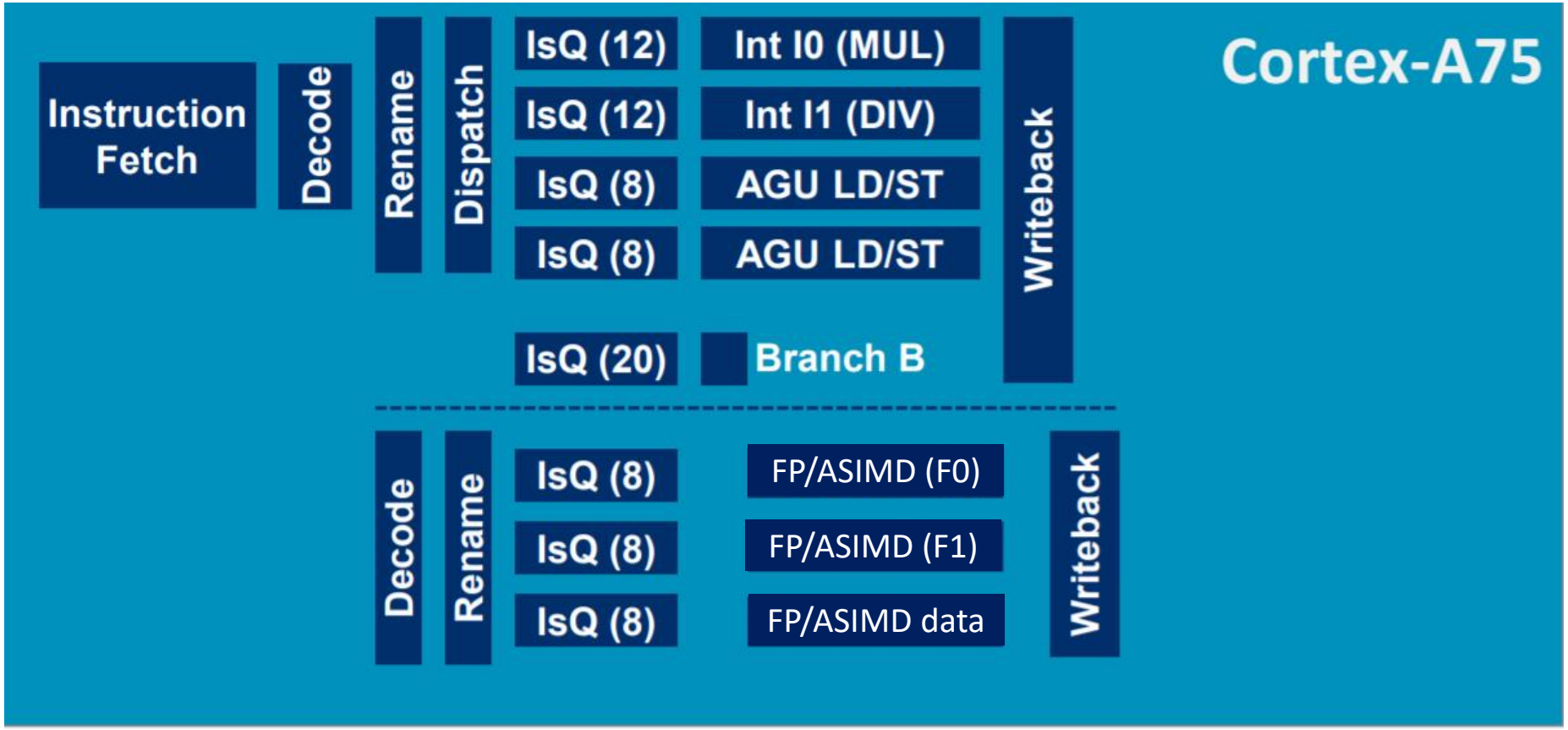
- The issue window is implemented as a large memory-like structure
- When an instruction is issued, its destination register is broadcast to all waiting instructions (perhaps after a short delay for longer latency operations)
- **Wakeup phase:** the waiting instructions compare the broadcast destination 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:

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

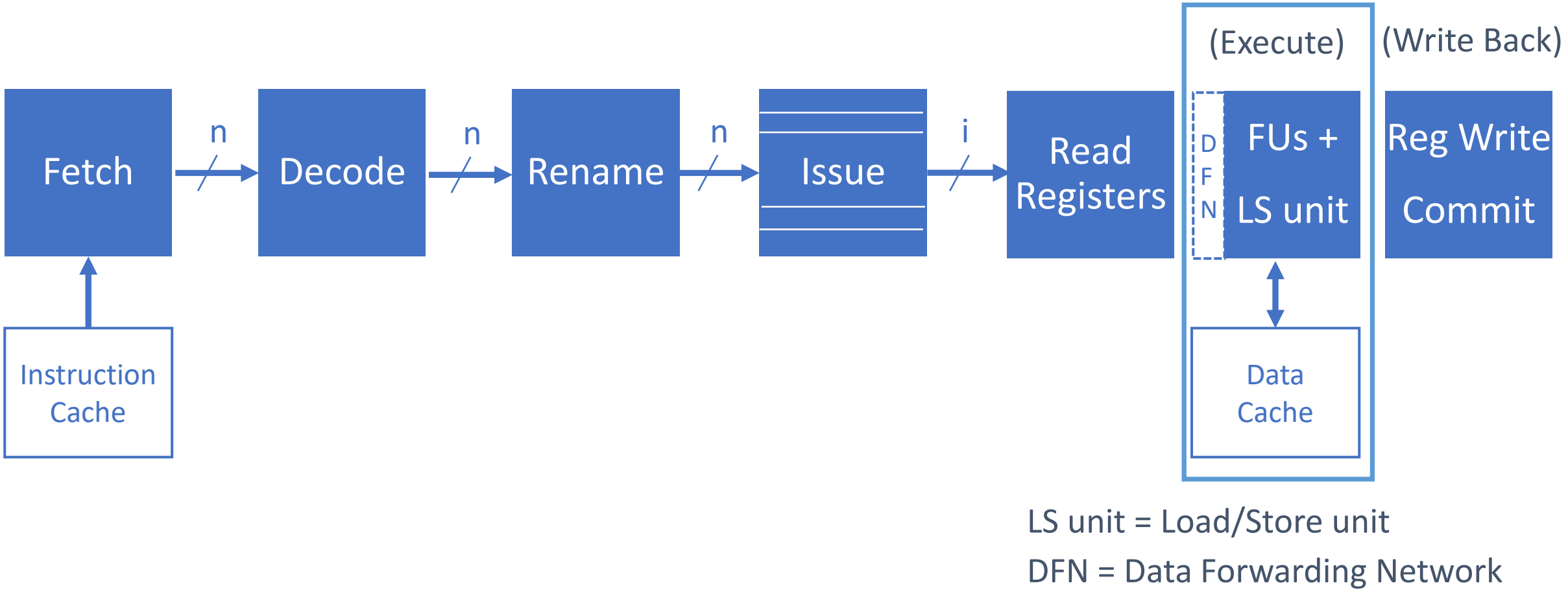
Example: A distributed instruction window



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)

A generic superscalar processor



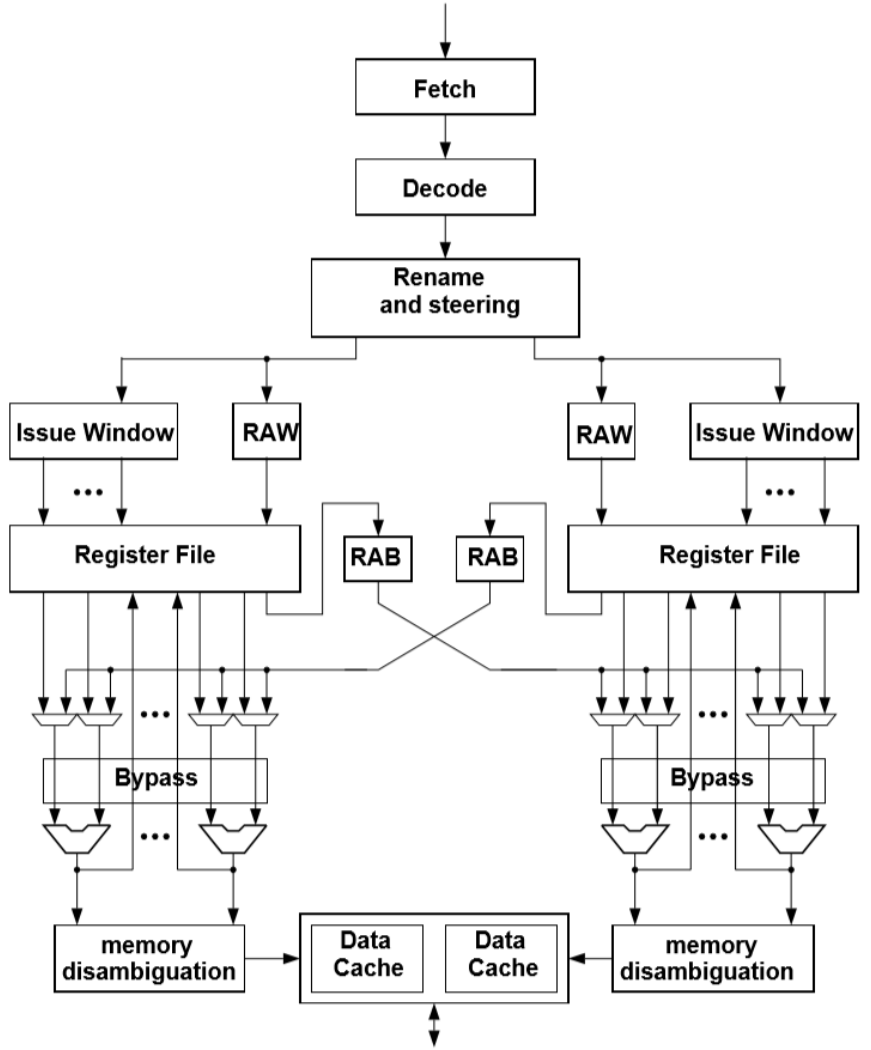
LS unit = Load/Store unit

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.



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

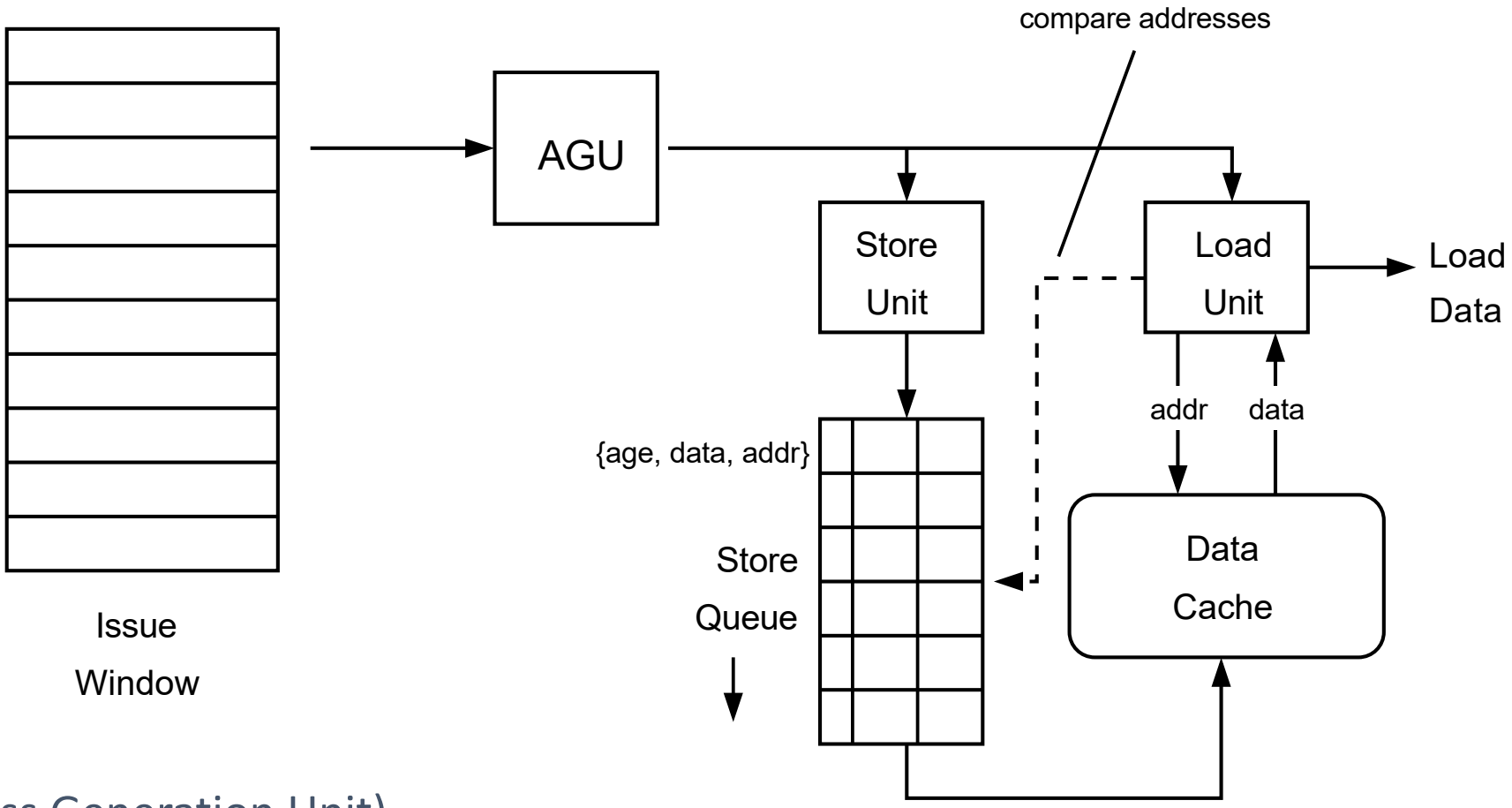
We will only permit stores to execute in program order

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

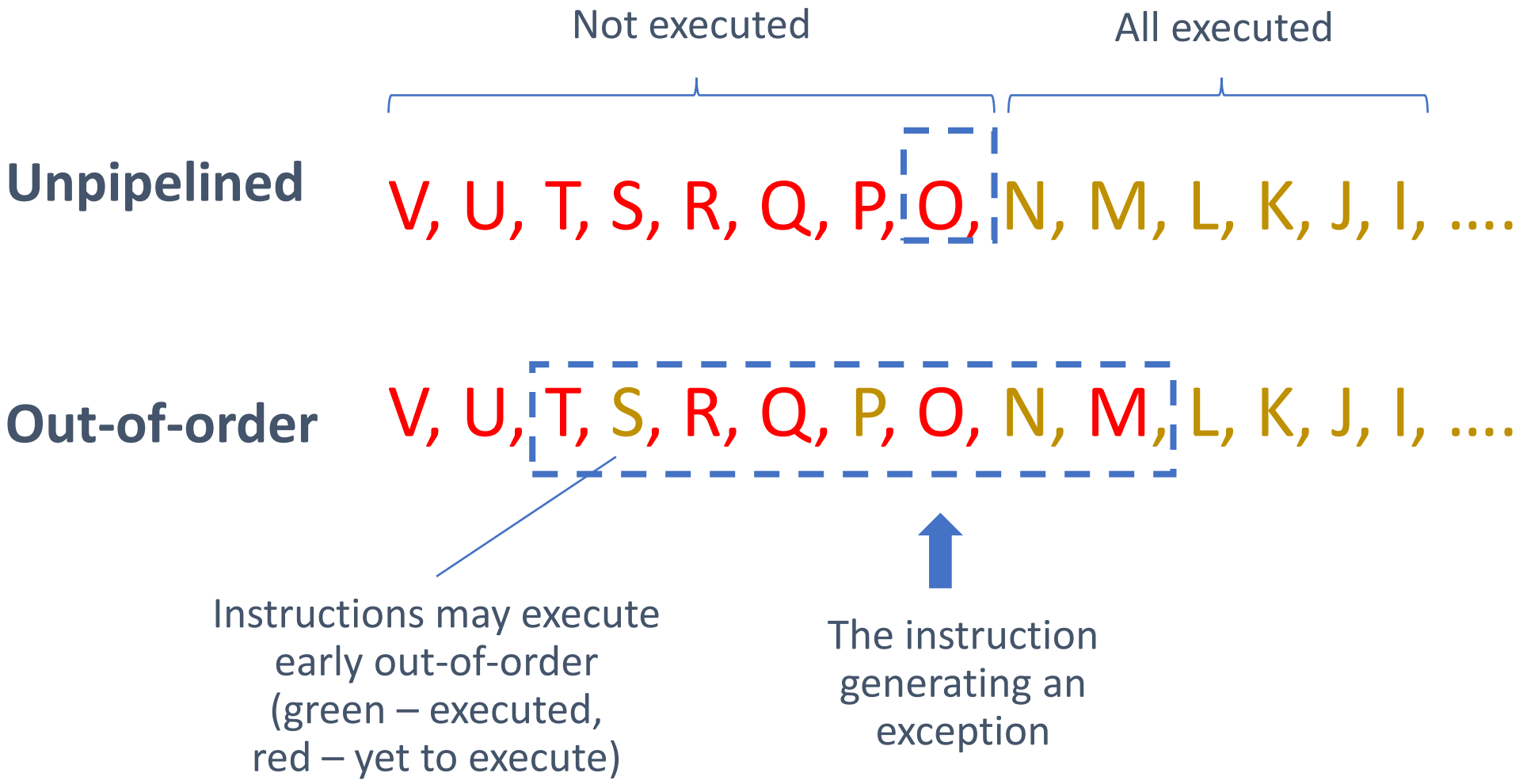
High-performance superscalar processors go further than this:

- 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

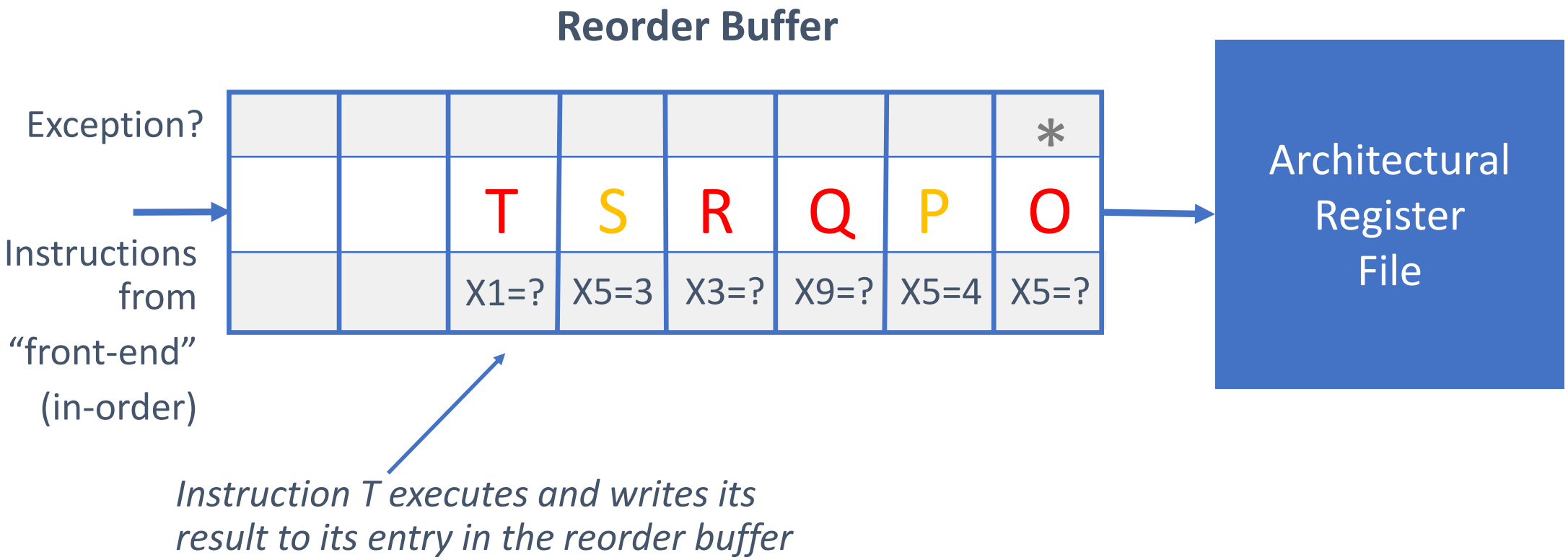
Superscalar processors: exceptions and speculation

- 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



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) By 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

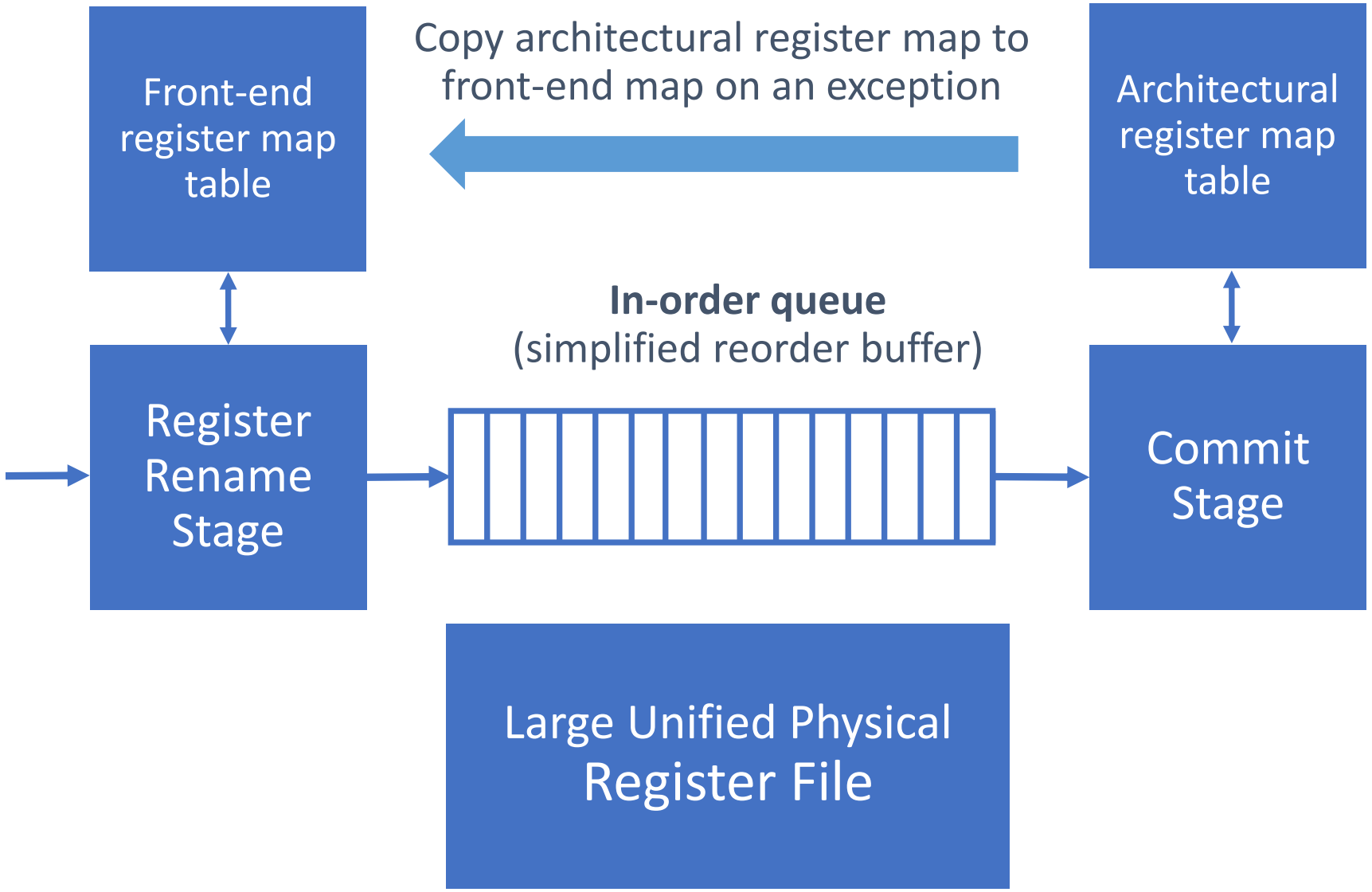
Superscalar processors: unified register file approach

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



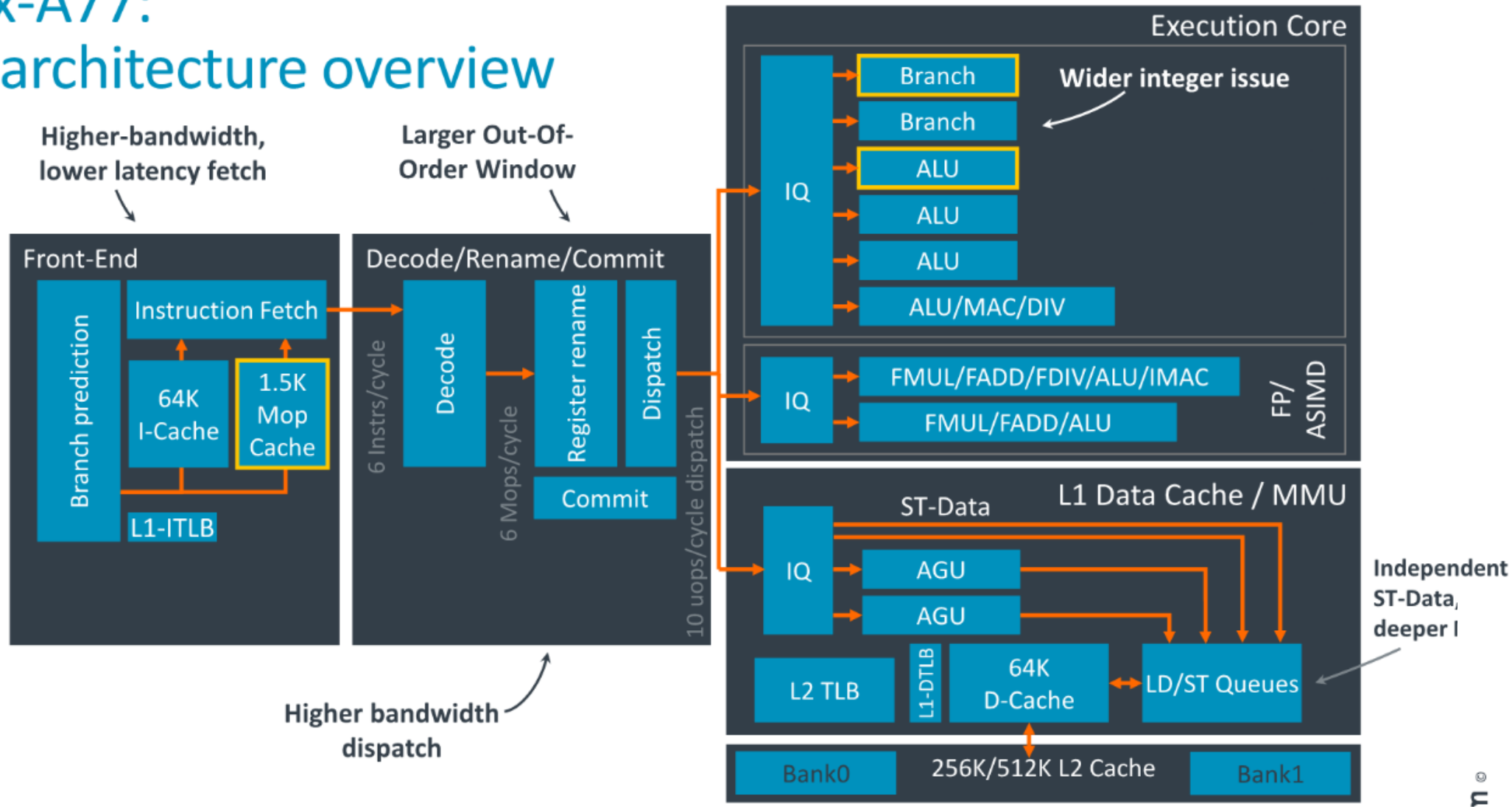
Superscalar processors: handling mispredicted branches

- 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 load/store queue holds 32-40 entries
- The target clock frequency is between 2.6 and 3GHz

Cortex-A77: Microarchitecture overview



Limits to superscalar processors

Ultimately the performance of a superscalar processor is limited by:

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

Why create a custom System-on-Chip (SoC)?

- Reduced cost
- Reduced PCB area and volume
- Increased performance and reduced power consumption
- Product differentiation

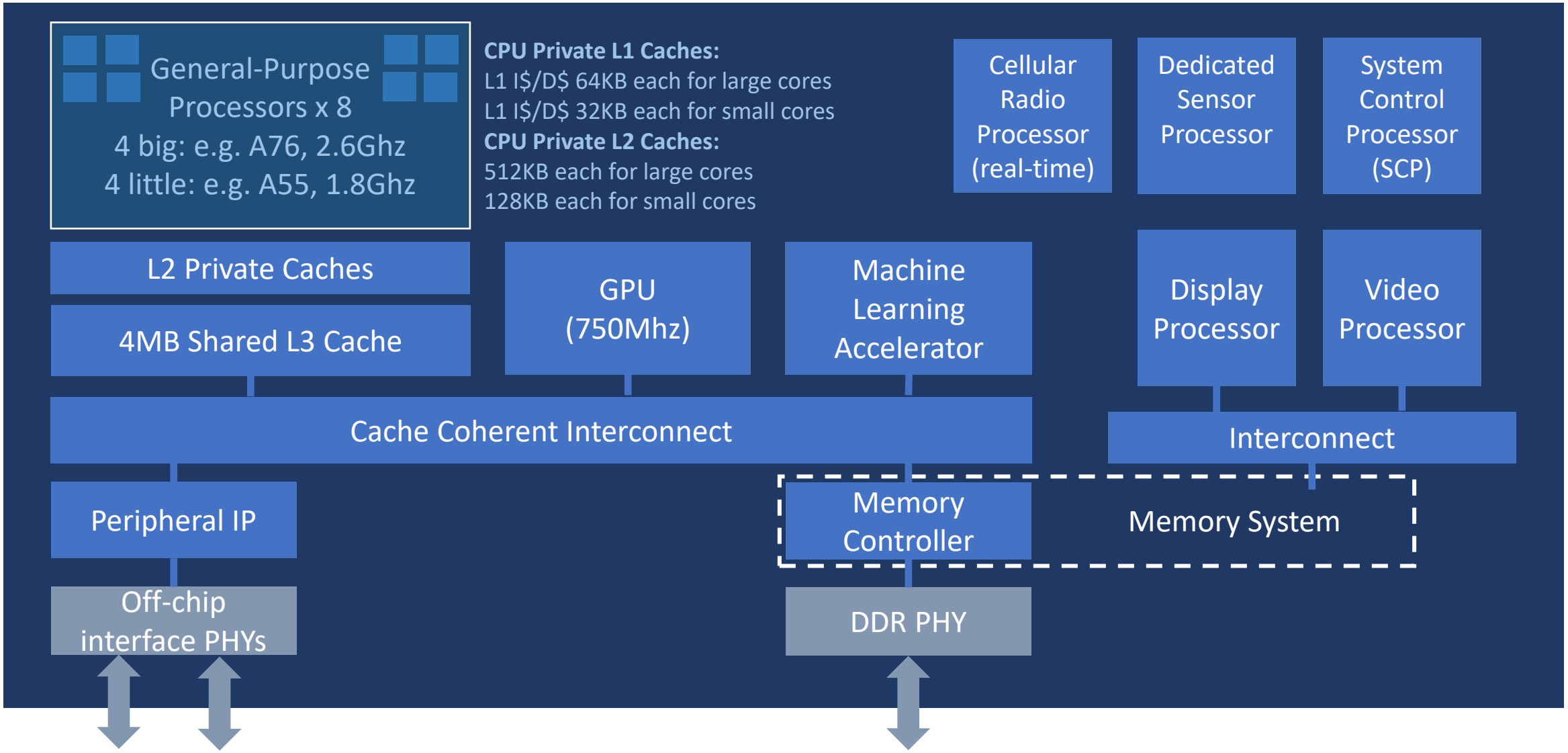
SoCs integrate a range of IP types: processors, custom processors, accelerators, on-chip memories, peripherals and interfaces, etc.

System-on-chip (SoC) design

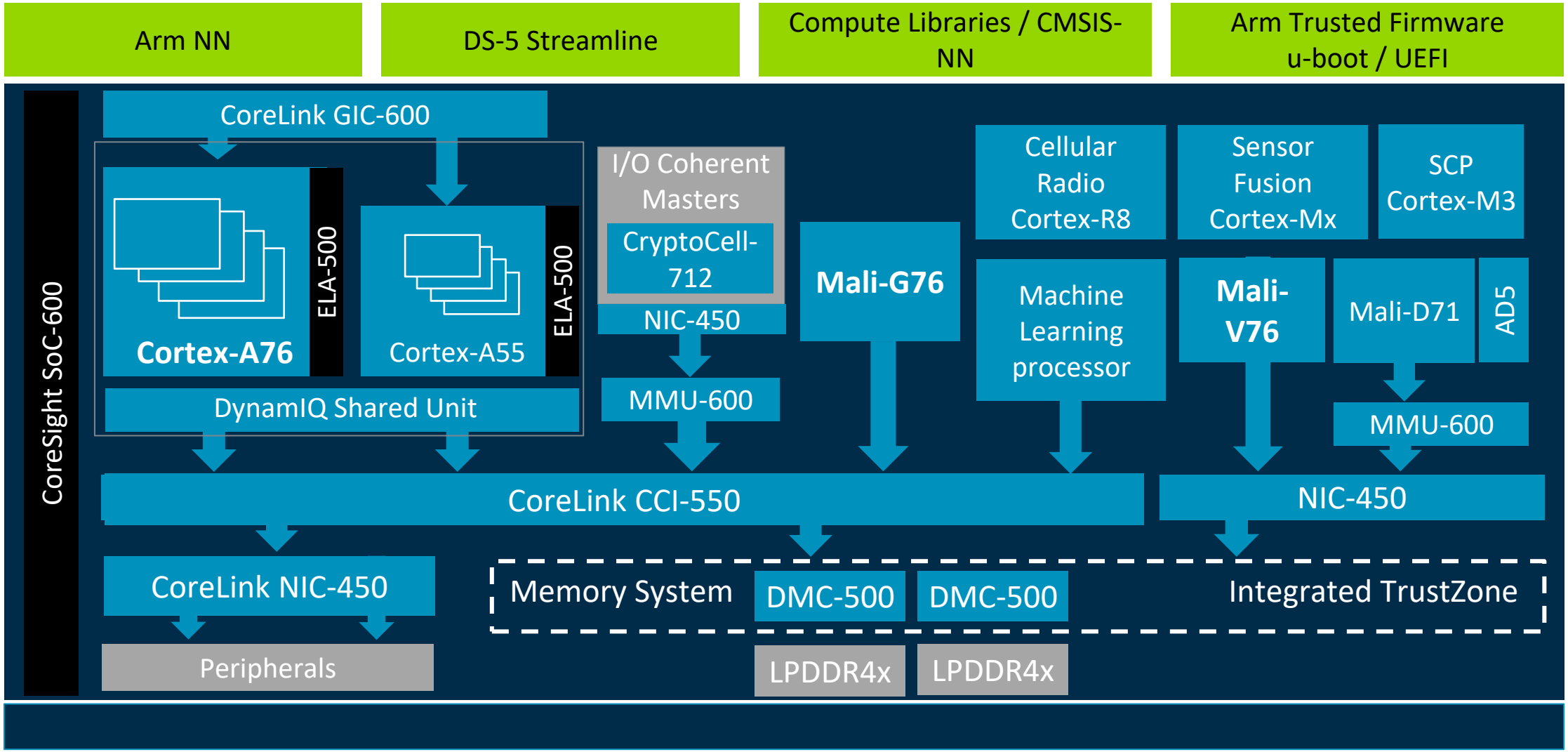
High-level design goals and constraints for a mobile SoC:

- **Target market**: client device (e.g. mobile phone, tablet etc.)
- **Cost per chip**: ~$20-$25, **Area** = ~70-80mm^2 in 7nm fabrication technology
- **Transistor budget**: ~7-8 billion transistors
- **Performance:** Excellent general-purpose performance over a wide range of workloads, e.g.: image processing, 2D/3D graphics, machine learning applications. Some requirements for real-time processing.
- **Off-chip memory bandwidth:** 32 GiB/s
- **Power:** 2-3W peak (only in short bursts on smartphones)

High-level view of our example System-on-Chip



A modern Arm System-on-Chip (SoC)



Heterogeneity

There are different levels of heterogeneity within the SoC

- Cores running the same ISA but with different microarchitectures (DynamIQ)
  - Allows general-purpose tasks to migrate to save power or increase performance
- Cores extracting different types of parallelism (Cortex-A cores vs Mali GPU)
  - E.g. the GPU is specialised to efficiently exploit data-parallel parallelism
- Cores specialised to specific tasks (machine-learning processor)
  - These are highly specialised hardware accelerators designed for a narrow range of workloads

The key aim for all of this is to reduce power consumption but increase performance

Heterogeneity in microarchitectures

Cortex A76

4-way superscalar, out-of-order processor

- 8-wide issue

13-stage pipeline

Multilevel branch-target cache

- Branch unit has 2x fetch-unit bandwidth

128-entry instruction window

Two load/store pipelines access 64KiB L1D

- Optimised for memory-level parallelism

Cortex A55

2-wide in-order superscalar

8-stage pipeline

- Sweet spot between power/area and frequency

Neural-network-based branch predictor

- 256-entry BTAC (branch target address cache)

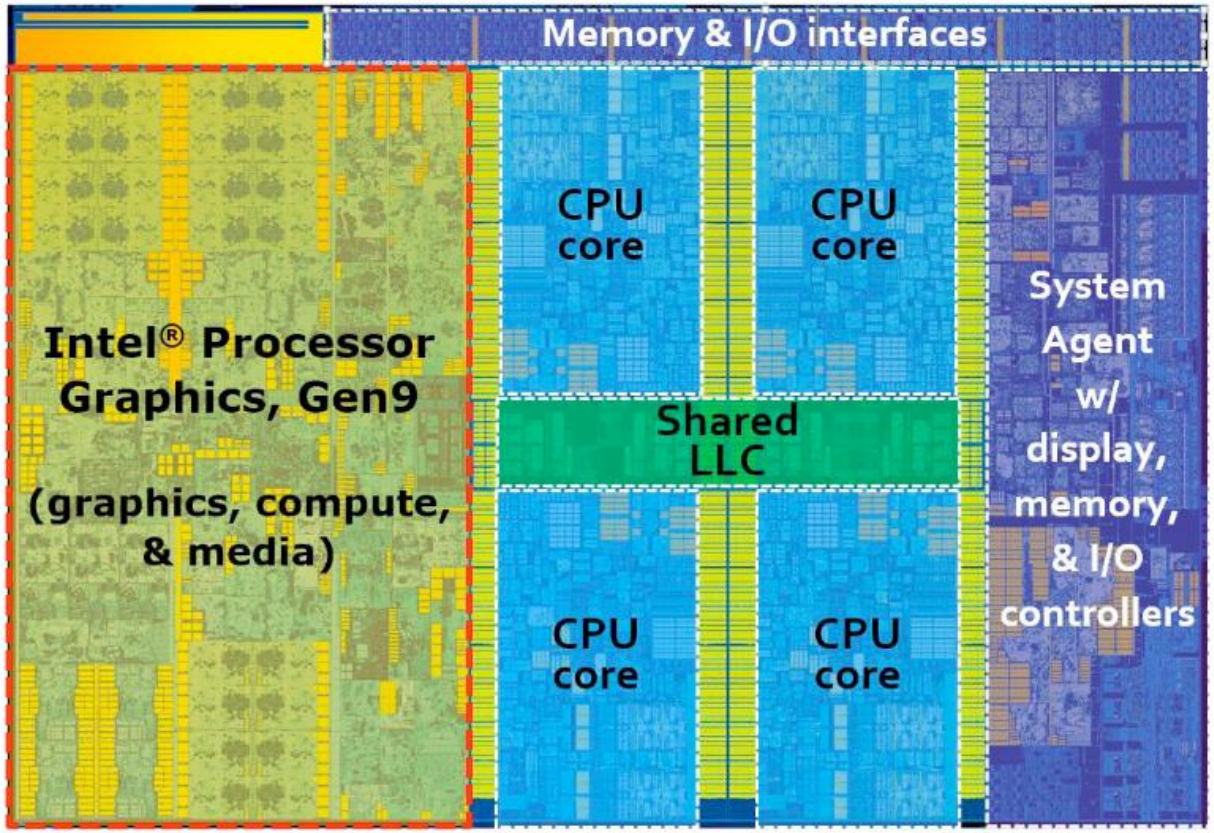
Configurable L1D cache size

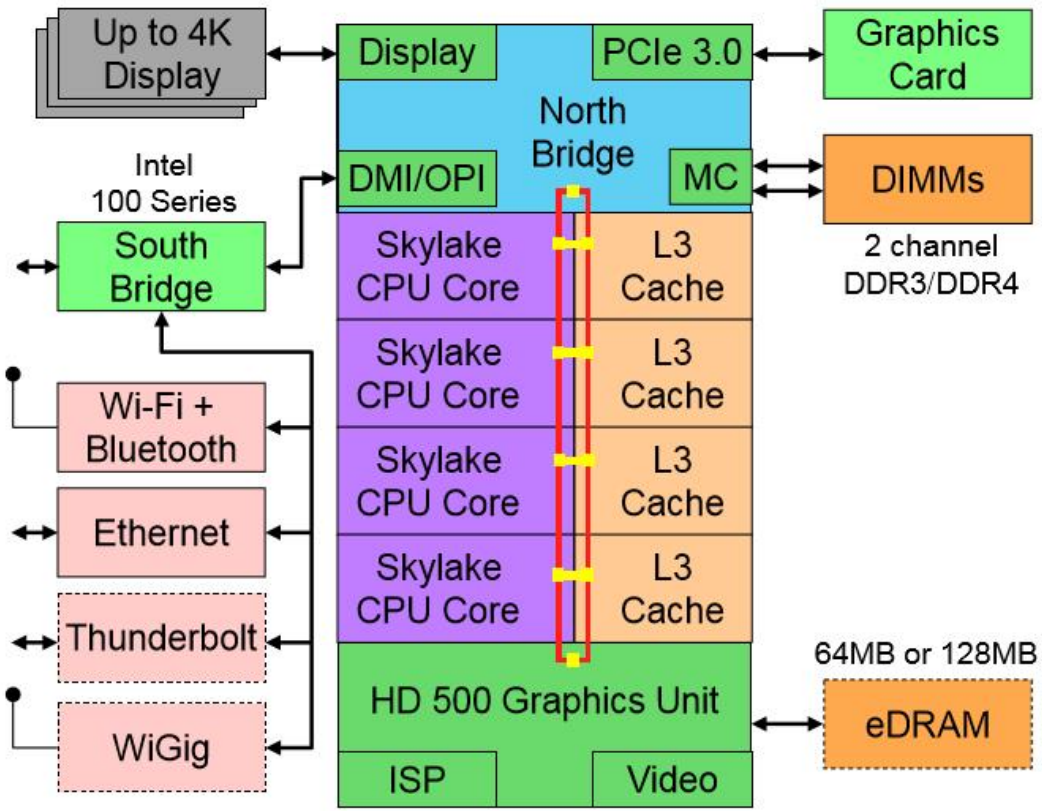
- Fully exclusive of L2

Independent load & store AGUs

- Address generation units

Intel Skylake i7-6700K (Q3, 2015)





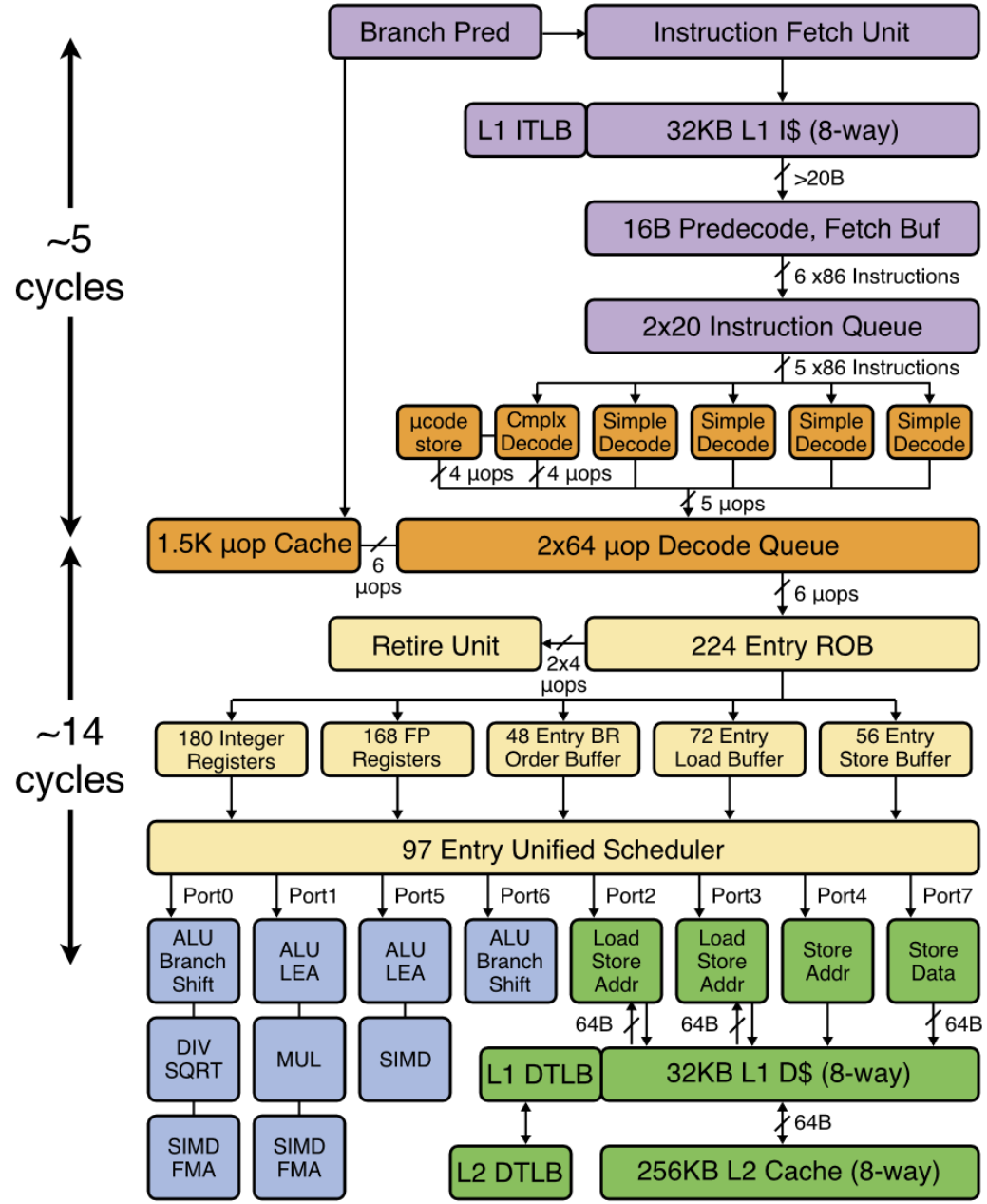
IO Fetch

IO Decode

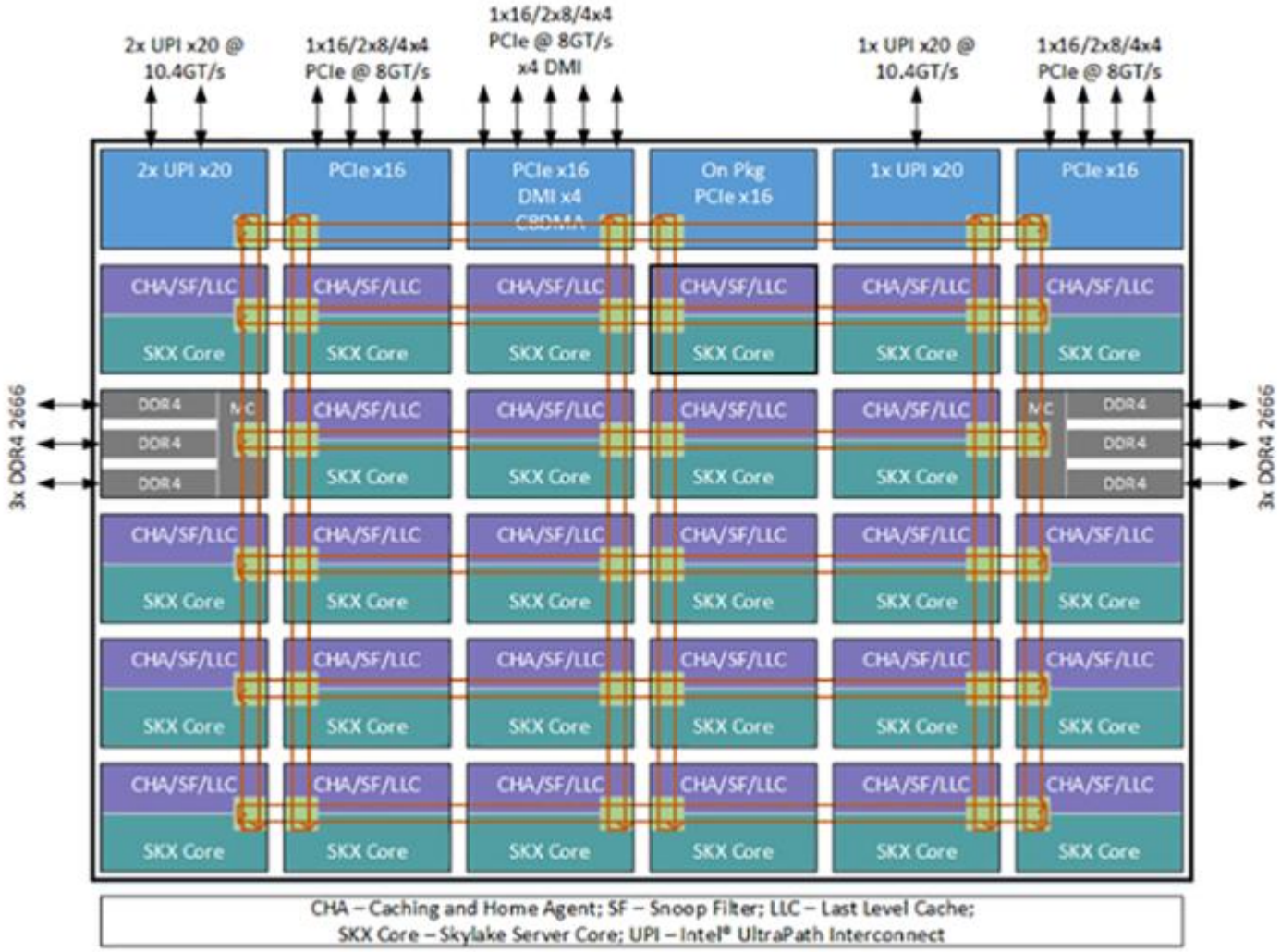
OOO Issue and Late Commit

Integer/FP Functional Units with OOO Writeback

Load/Store Execution



Intel Xeon scalable mesh architecture

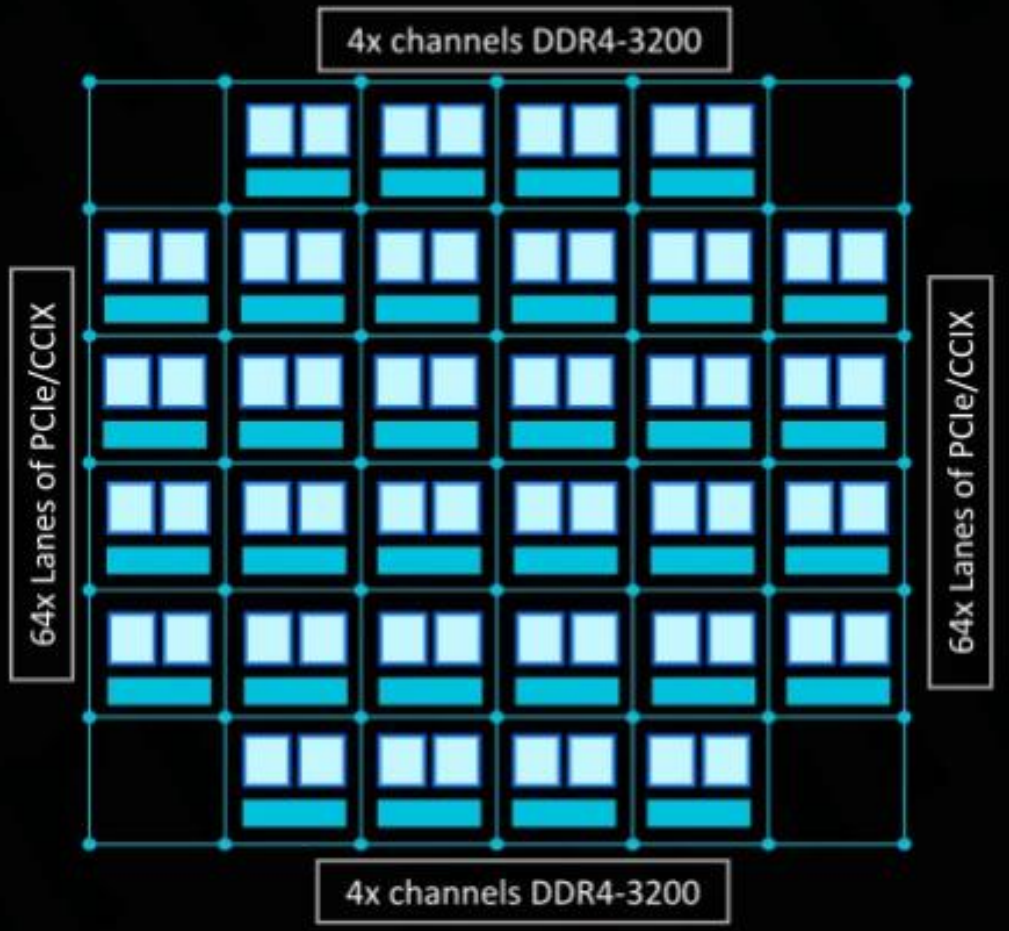


Arm Neoverse design (64-128 cores)

Neoverse N1 hyperscale reference design

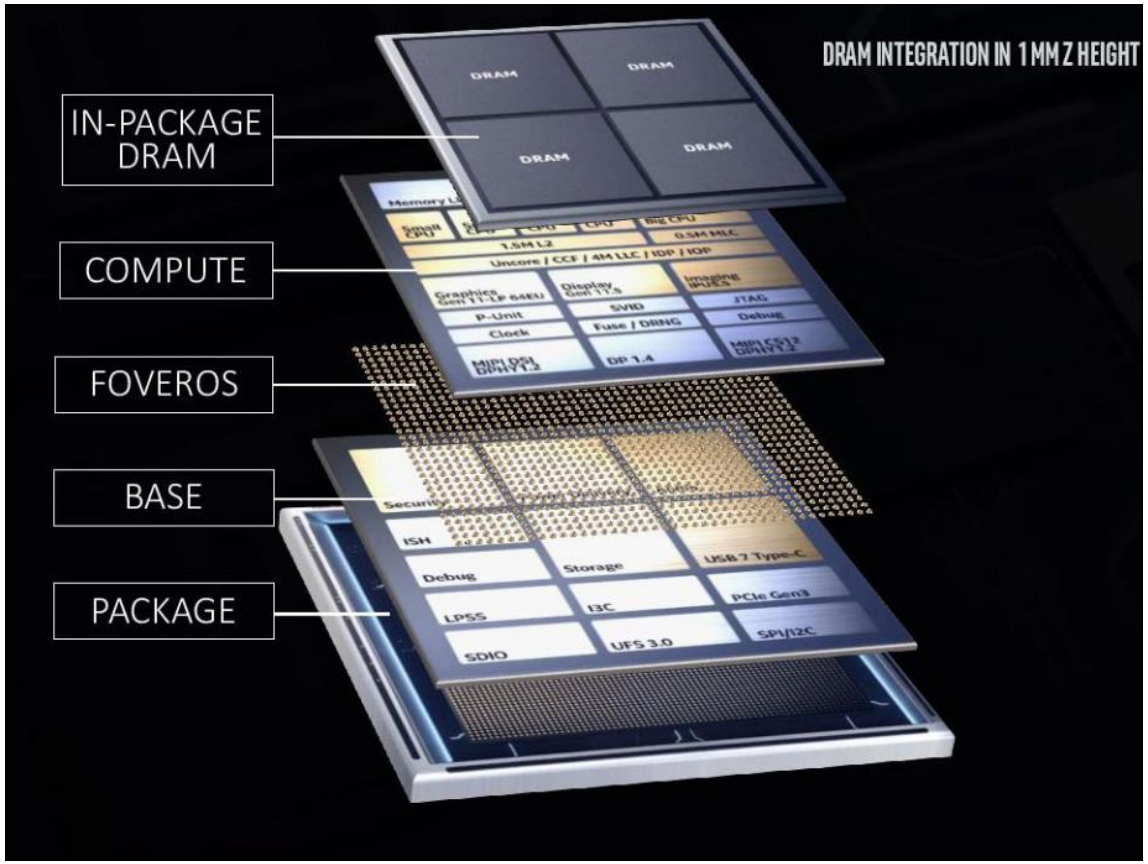
- 64x-128x Neoverse N1 CPU
- 8x8 Mesh with 64-128 MB of system level cache (SLC)
- 128x lanes, PCIe, CCIX
- 8x channels of DDR4 memory
- 7nm implementation

*Designed for RTL/emulation.



Intel Lakefield (Smartphones and laptops)

- Q4 2019
- 5W/7W configurations
- Graphics
- Heterogeneous cores (Intel Big-Bigger similar to Arm's big.LITTLE)



3D die stacking (Intel “Foveros”)

P1274 = 10nm process, P1222 is 22nm FinFET low power (low leakage current)
“Foveros” is the 3D face-to-face chip stacking technology, microbumps + Through Silicon Vias (**TSVs**)
POP = standard stacked Package-on-Package memory