Dipendenze e architetture superscalari

Andrea Bartolini <a.bartolini@unibo.it>

(Architettura dei) Calcolatori Elettronici, 2023/2024

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls

- Parallelism with basic blocks is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

Data Dependence

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)

- Challenges:
  - Data dependency
    - Instruction *j* is data dependent on instruction *i* if
      - Instruction *i* produces a result that may be used by instruction *j*
      - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*

- Dependent instructions cannot be executed simultaneously

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall

- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism

- Dependencies that flow through memory locations are difficult to detect

Data dependency

Consider the following block of code, with x_i = i

```
add  x1, x2,x3     % x1 becomes 5
add  x6,x1,x10     % x6 becomes 15
```

If we reverse the order:

```
add  x6,x1,x10     % x6 becomes 11
add  x1, x2,x3     % x1 becomes 5
```

=> Data Hazard !!!

Two instructions are independent (there is no dependency) if we can reverse their order without changing the result.
This is called "data flow".

In reality, to preserve the program's correctness, it is necessary to ensure that the new order preserves how exceptions occur.
This condition is called "exception behavior"

Data dependency #2

Consider the following block of code, with x_i = i

```
add  x1, x2,x3     % x1 becomes 5
add  x6,x20,x10  % x6 becomes 30
```

If we reverse the order:

```
add  x6,x20,x10  % x6 becomes 30
add  x1, x2,x3     % x1 becomes 5
```

The two instructions are not dependent => ILP!

Two instructions are independent (there is no dependency) if we can reverse their order without changing the result.
This is called "data flow".

In reality, to preserve the program's correctness, it is necessary to ensure that the new order preserves how exceptions occur.
This condition is called "exception behavior"

Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, but is a problem when reordering instructions
  - Antidependence: instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - Output dependence: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- To resolve, use register renaming techniques

Name dependency (exit dependency)

Consider the following block of code, with x_i = i

```
div  x1, x6,x2        % x1 becomes 3
add  x1, x10,x11      % x1 becomes 21
```

If we reverse the order:

```
add  x1, x10,x11      % x1 becomes 21
div  x1, x6,x2        % x1 becomes 3
```

=> Hazard !!!

Two instructions are independent (there is no dependency) if we can reverse their order without changing the result.
This is called "data flow".

In reality, to preserve the program's correctness, it is necessary to ensure that the new order preserves how exceptions occur.
This condition is called "exception behavior"

Name dependency (anti-dependency)

Consider the following block of code, withx_i = i

```
add  x6, x1, x20     % x6 becomes 21
add  x1, x2, x3      % x1 becomes 5
```

If we reverse the order:

```
add  x1, x2, x3      % x1 becomes 5
add  x6, x1, x20     % x6 becomes 25
```

=> Hazard !!!

Two instructions are independent (there is no dependency) if we can reverse their order without changing the result.
This is called "data flow".

In reality, to preserve the program's correctness, it is necessary to ensure that the new order preserves how exceptions occur.
This condition is called "exception behavior"

Name Dependencies

Both name dependencies are false dependencies and can be resolved both statically and dynamically through a procedure called «register renaming»

- If the name (register or memory location) is changed to avoid conflict, we can reorder or execute instructions in parallel.

```
div  x1, x6,x2          div  T,  x6,x2
add  x1, x10,x11   →    add  x1, x10,x11

add  x6, x1, x20        add  x6, x1, x20
add  x1, x2, x3    →    add  T,  x2, x3
```

The two instructions are not dependent => ILP!

Other Factors

- Data Hazards
  - Read after write (RAW): *j prova a leggere una sorgente di dato (registro/memoria) prima che i lo scriva, j erroneamente legge il valore "vecchio".*
  - Write after write (WAW): *j scrive un operando (registro/memoria) prima che i lo scriva. La scrittura avviene nell'ordine sbagliato, lasciando il valore scritto da i al posto di quello scritto da j.*
    Possibile solo in pipeline che permettono la scrittura in più stadi o che permettono alle istruzioni di eseguire fuori ordine.
  - Write after read (WAR): *j scrive un operando (registro/memoria) prima che i lo legga , i erroneamente legge il valore "nuovo".*
    Non può succedere se la pipeline legge gli operandi in ID, quindi prima del fetch dell'istruzione successiva. Può invece accadere se alcune istruzioni scrivono nei primi stadi della pipeline o se le istruzioni eseguono fuori ordine.

Control Dependencies

- Ordering of instruction i with respect to a branch instruction
  - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
  - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

```
  add  x2, x3, x4
  beq  x2, x0, L1
  ld        x1, 0(x2)
L1:
  ..
```

```
  add  x2, x3, x4
  ld        x1, 0(x2)
  beq  x2, x0, L1
L1:
  ..
```

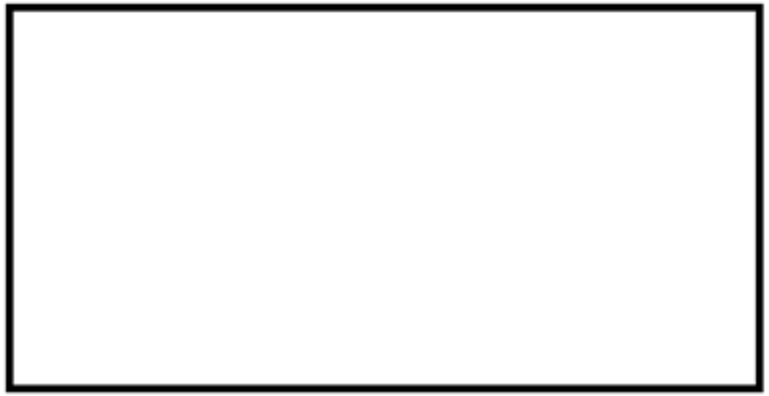
=> Alea di Controllo!!!

How to use ILP with blocking «in-order» architectures

- Superpipelined Architettures
- Superscalar Architettures

Recap: Non-pipelined and Pipelined Processors

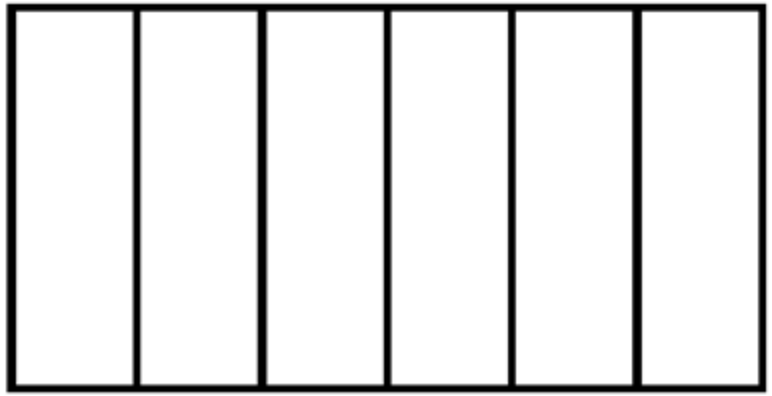
Non-pipelined processor



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

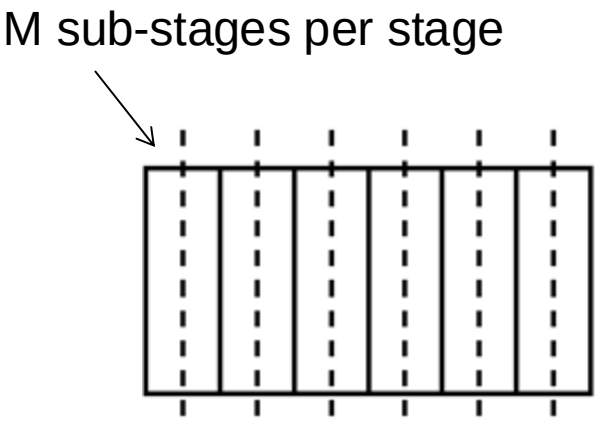
IPC ≤ 1

Clock Period = T/S+C

(C = pipelining overhead)

This is a **scalar pipeline.**

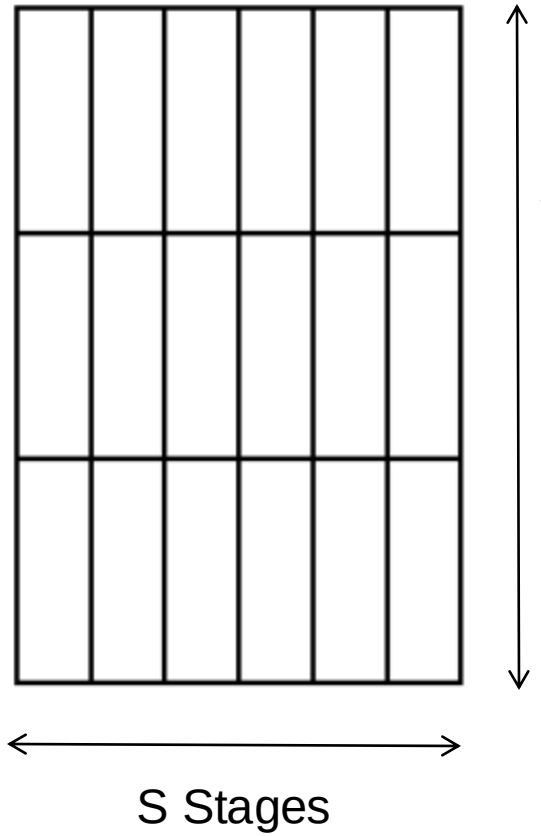
Superpipelined and Superscalar Processors



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.



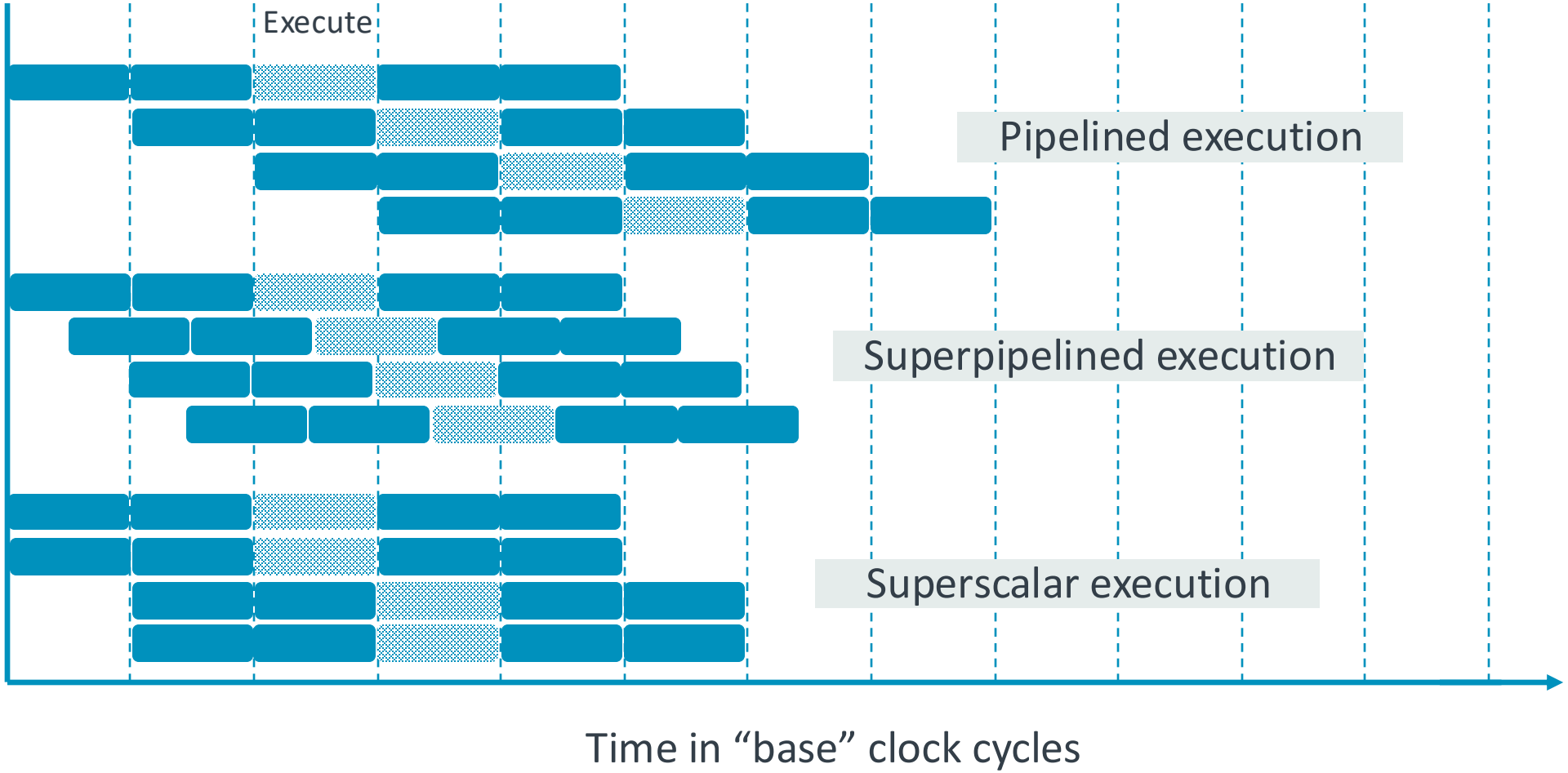
A Superscalar Processor

P instructions are processed in each pipeline stage.

IPC ≤ P

Clock Period = T/S+C

Superpipelined and Superscalar Processors



arm

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.

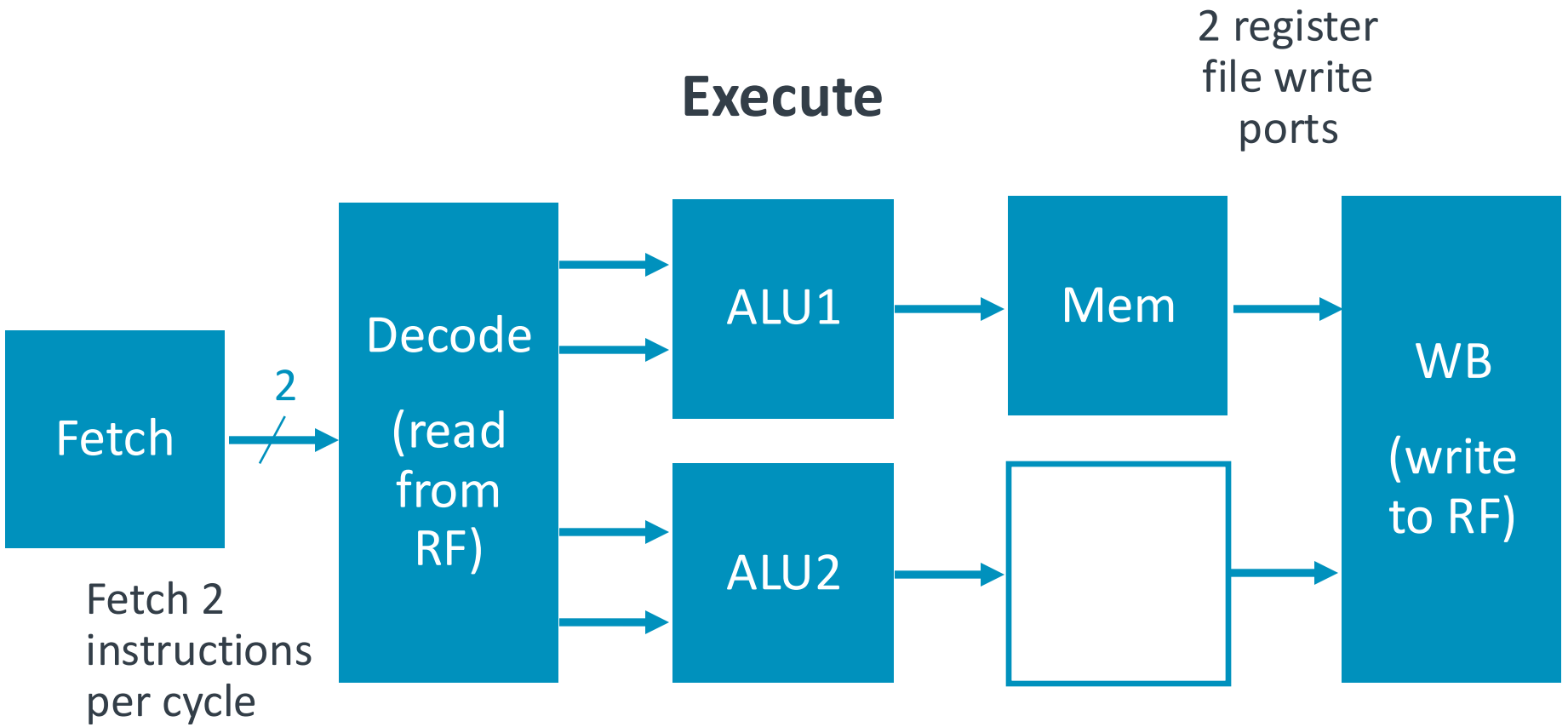
What extra hardware will this processor require?

- 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.

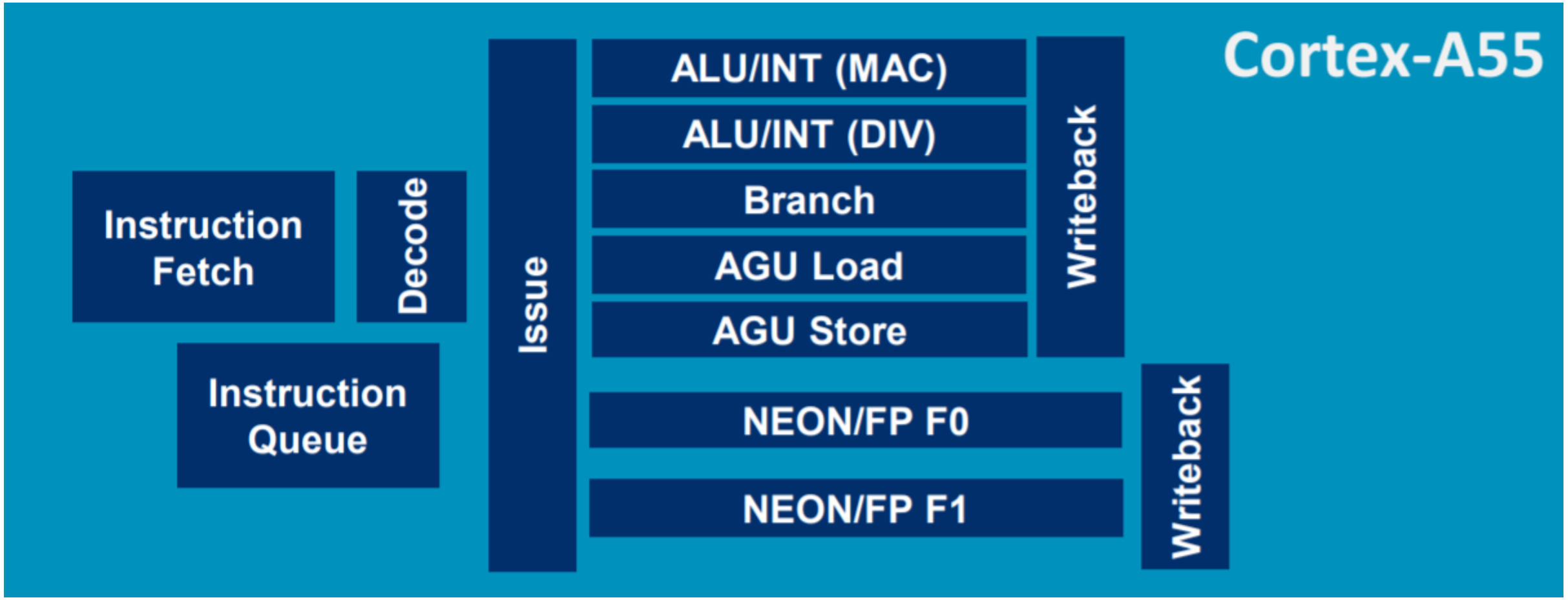
Simple In-order Superscalar Processor



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

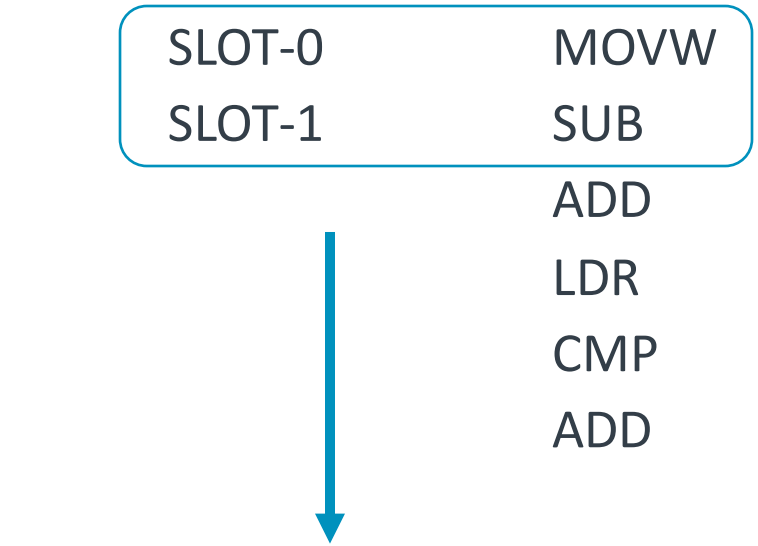
Issue Slots

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

arm

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