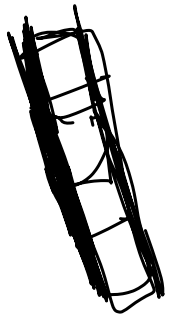
Dipendenze e architetture superscalari

Andrea Bartolini <a.bartolini@unibo.it>

(Architettura dei) Calcolatori Elettronici, 2023/2024



Quando si stalla vuol dire che bisogna fermarsi perché ci sono dei conflitti ⇒ instruction level parallelism. cioè le istruz. possono essere eseguite contemporaneamente.

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

Dipendenza di dato

Consideriamo il seguente blocco di codice, con x_i = i

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

Se invertiamo l'ordine:

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

=> Alea di dato!!!

Due istruzioni sono indipendenti (non c'è dipendenza) se possiamo invertire l'ordine delle istruzioni senza modificare il risultato.
Questa condizione è detta «data flow».

In realtà affinchè sia preservato la correttezza del programma serve garantire che il nuovo ordine preservi il modo in cui le eccezioni avvengono.
Questa condizione è detta "exception behavior"

Dipendenza di dato #2

Consideriamo il seguente blocco di codice, con x_i = i

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

Se invertiamo l'ordine:

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

Le due istruzioni non sono dipendenti => ILP!

Due istruzioni sono indipendenti (non c'è dipendenza) se possiamo invertire l'ordine delle istruzioni senza modificare il risultato.
Questa condizione è detta «data flow».

In realtà affinchè sia preservato la correttezza del programma serve garantire che il nuovo ordine preservi il modo in cui le eccezioni avvengono.
Questa condizione è detta "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

Dipendenza di Nome (dipendenza di uscita)

Consideriamo il seguente blocco di codice, con x_i = i

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

Se invertiamo l'ordine:

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

=> Alea !!!

Due istruzioni sono indipendenti (non c'è dipendenza) se possiamo invertire l'ordine delle istruzioni senza modificare il risultato.
Questa condizione è detta «data flow».

In realtà affinchè sia preservato la correttezza del programma serve garantire che il nuovo ordine preservi il modo in cui le eccezioni avvengono.
Questa condizione è detta "exception behavior"

Dipendenza di Nome (antidipendenza)

Consideriamo il seguente blocco di codice, con x_i = i

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

Avrei potuto chiamarlo in un altro modo, es. x7

Se invertiamo l'ordine:

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

=> Alea !!!
```

Due istruzioni sono indipendenti (non c'è dipendenza) se possiamo invertire l'ordine delle istruzioni senza modificare il risultato.
Questa condizione è detta «data flow».

In realtà affinchè sia preservato la correttezza del programma serve garantire che il nuovo ordine preservi il modo in cui le eccezioni avvengono.
Questa condizione è detta "exception behavior"

Dipendenze di Nome

Entrambe le due dipendenze di nome sono delle false dipendenze e possono essere risolte sia staticamente che dinamicamente tramite una procedure chiamata «register renaming»

- Possiamo riordinare o eseguire parallelamente le istruzioni se il nome (registro o locazione di memoria) viene cambiato per evitare il conflitto.

```
div  x1, x6,x2          div  T,  x6,x2        add  x6, x1, x20        add  x6, x1, x20
add  x1, x10,x11   ➡    add  x1, x10,x11      add  x1, x2, x3    ➡    add  T,  x2, x3
```

Le due istruzioni non sono dipendenti => 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. Non da problemi perchè il wb della 1ª istruzione avviene prima del wb della 2ª istruzione.
  - 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.
    Questa può dare problemi

Dipendenze di Controllo

- 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!!!

Come usare l’ILP con le architetture bloccanti «in-order»

- Architetture superpipelined
- Architetture superscalari

Recap: Non-pipelined and Pipelined Processors

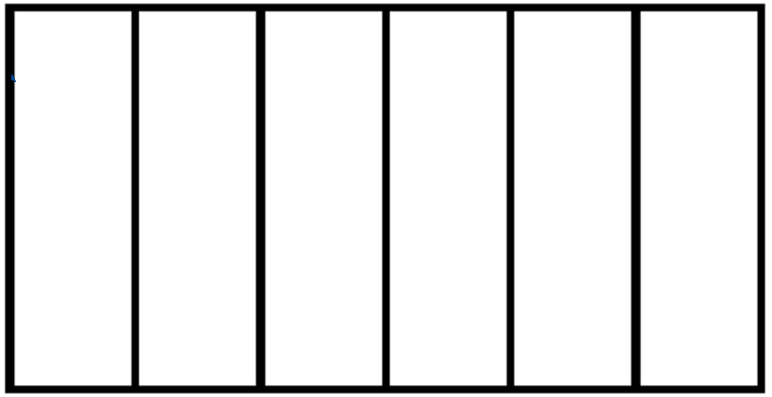
SENZA PIPELINE SI ESEGUE 1 ISTRUZIONE PER CK. OVVIAMENTE IL CK SARÀ PIÙ LUNGO.

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

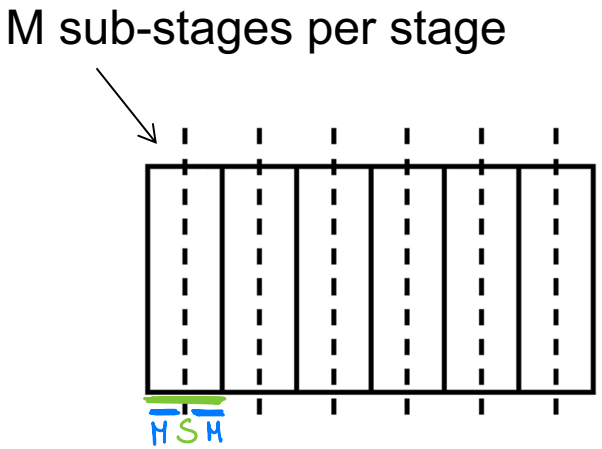
Clock Period = T/S+C

(C = pipelining overhead)

This is a **scalar pipeline.**

arm

Superpipelined and Superscalar Processors

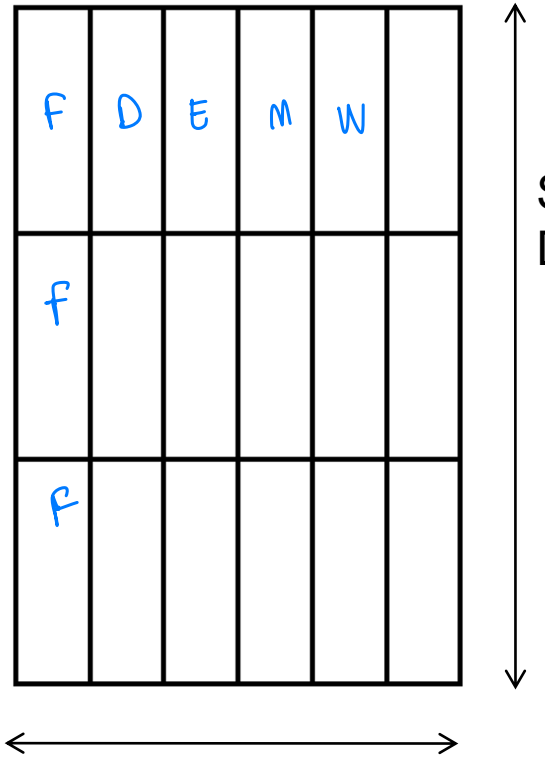


A Superpipelined Processor

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.

Divido le parti della pipeline in sotto blocchetti
↳ replica con fattore 3 → 3 vie.



ogni ck leggo dalla memoria 3 istruzioni che non eseguiranno in sequenta, ma in contemporanea.

unico fetch: un ck poi vanno avanti tu //.

add → 1ª via
sub → 2ª via
add → 3ª via
sub
add
ld → 2° fetch di gruppo

In 1 Tck esegue istruz.

A Superscalar Processor

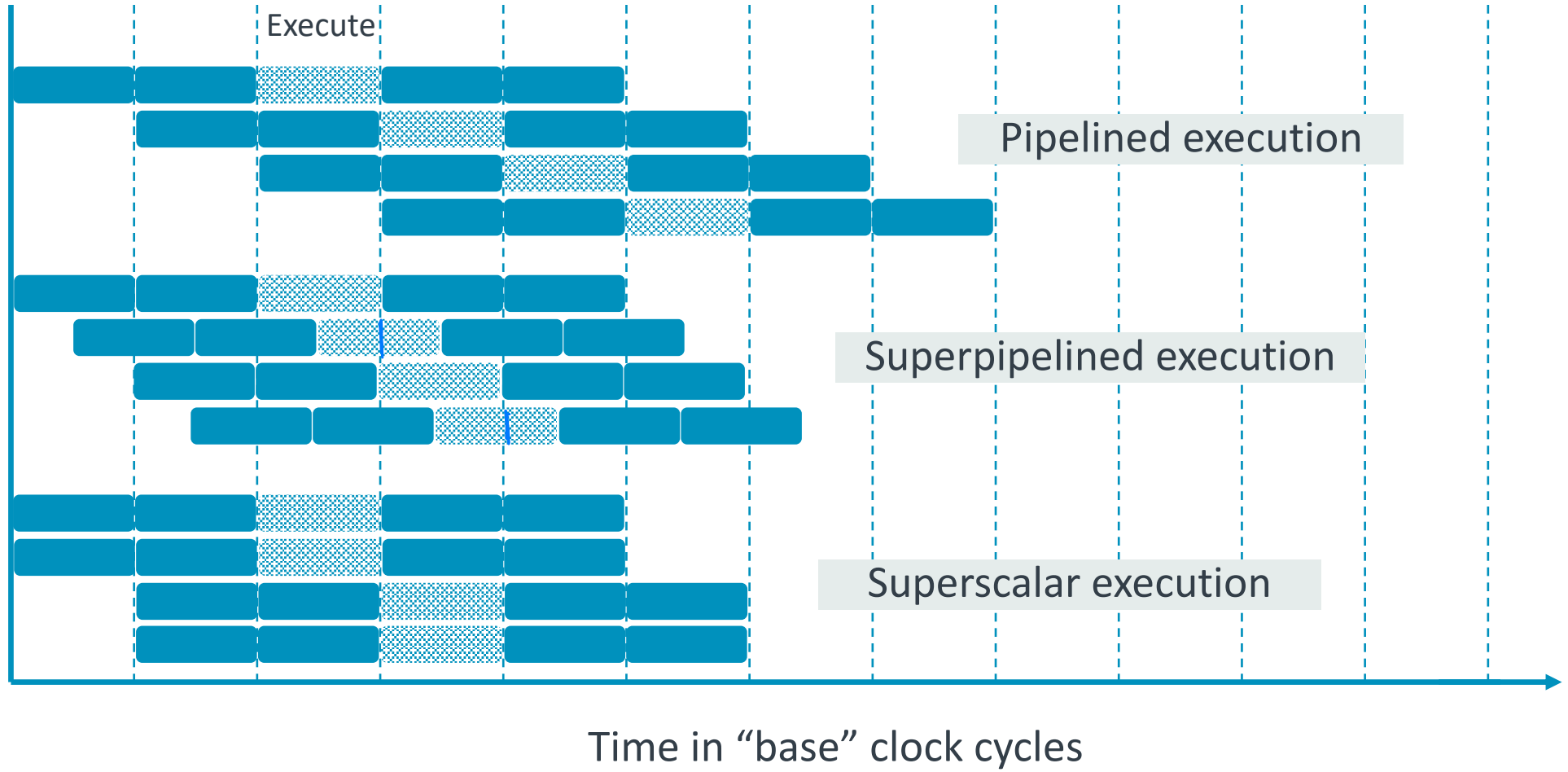
P instructions are processed in each pipeline stage.

IPC ≤ P

Clock Period = T/S+C

Qui c'è un problema la dipendenta di nome

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.

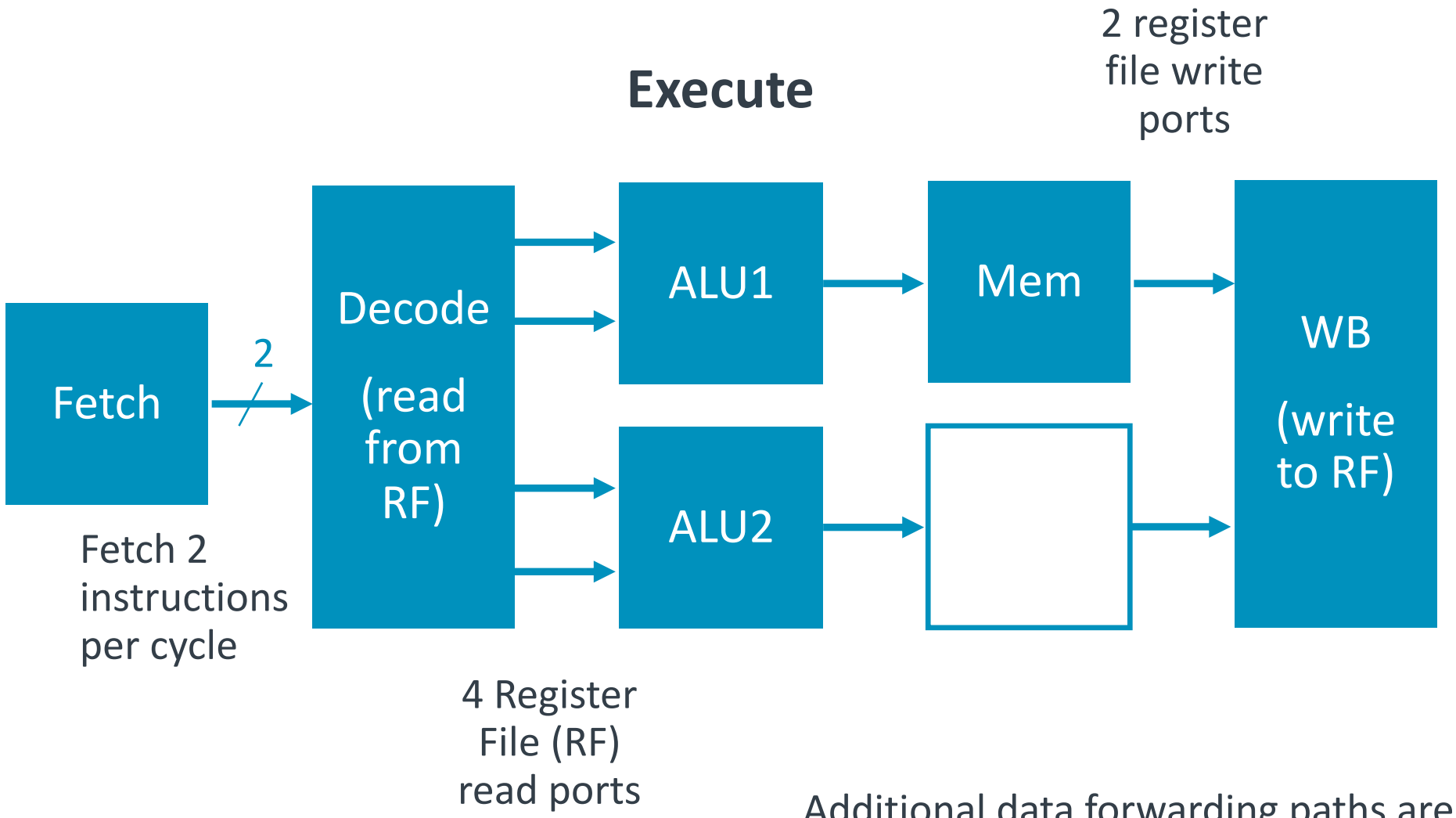
What extra hardware will this processor require?

- 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

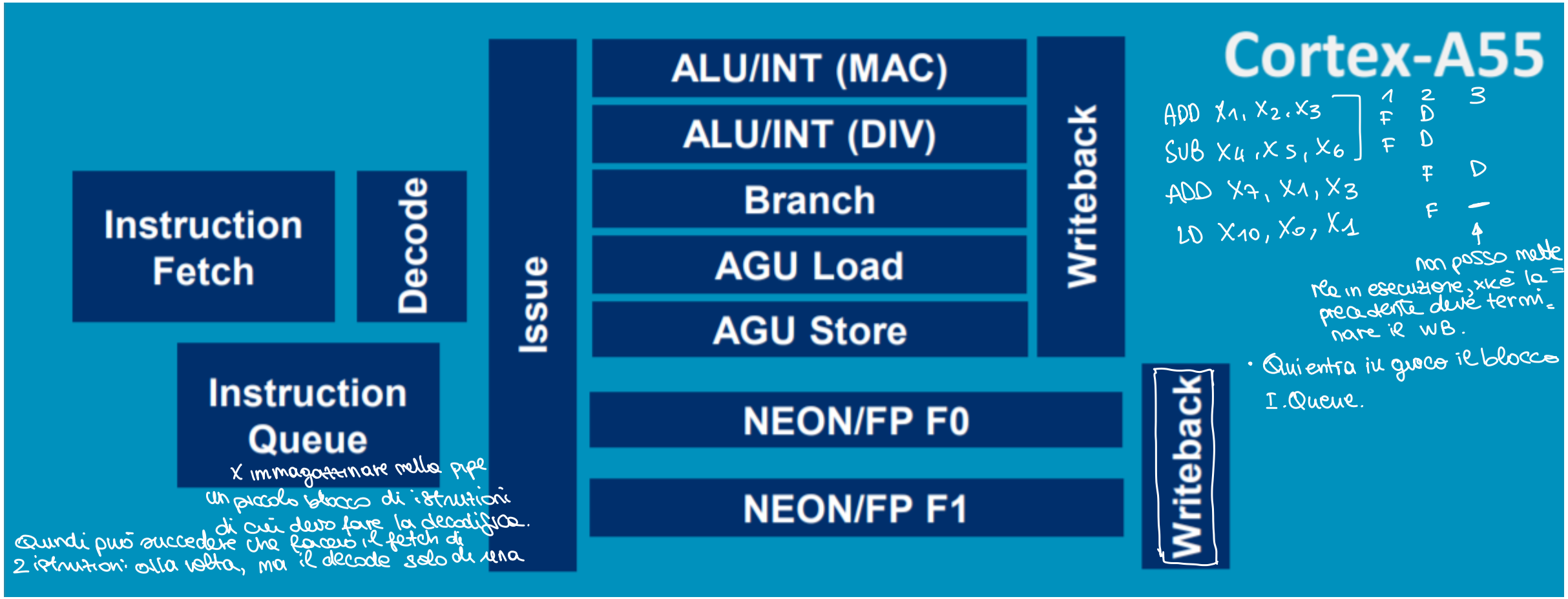
- We can create a simple (2-way) superscalar processor with a few changes to our scalar pipeline.
- We will fetch and decode multiple instructions per cycle.
- Instructions are sent to functional units in program order (in-order issue).
- We will issue and execute instructions in parallel if we can.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



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

arm

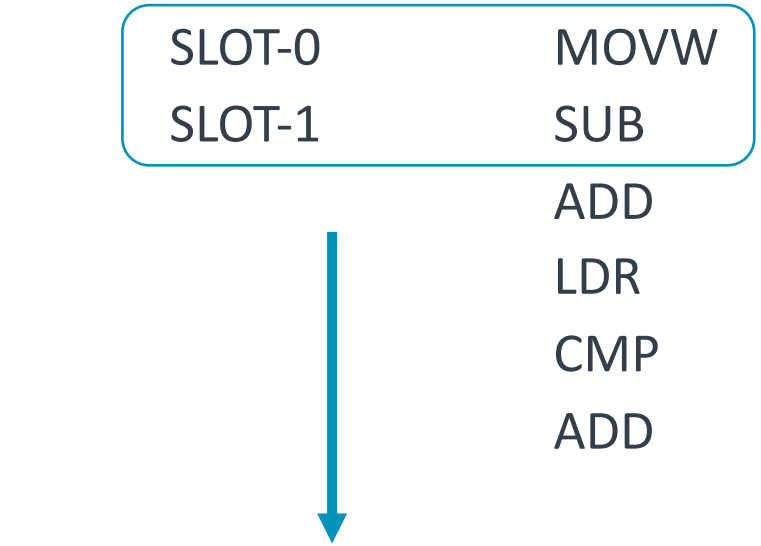
Issue Slots

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