Understanding the Tomasulo Algorithm

Yichao Cheng

Jul 23, 2013

Background

- IBM System/360 Model 91
- FPU's add/mul/div takes 2/3/13 cycles
- Can performance be improved through utilizing multiple execution units?

Major Contributions

Proposed three innovative mechanisms:

- Common data busing(CDB)
- Register tagging scheme
- Reservation station

which ***permits:***

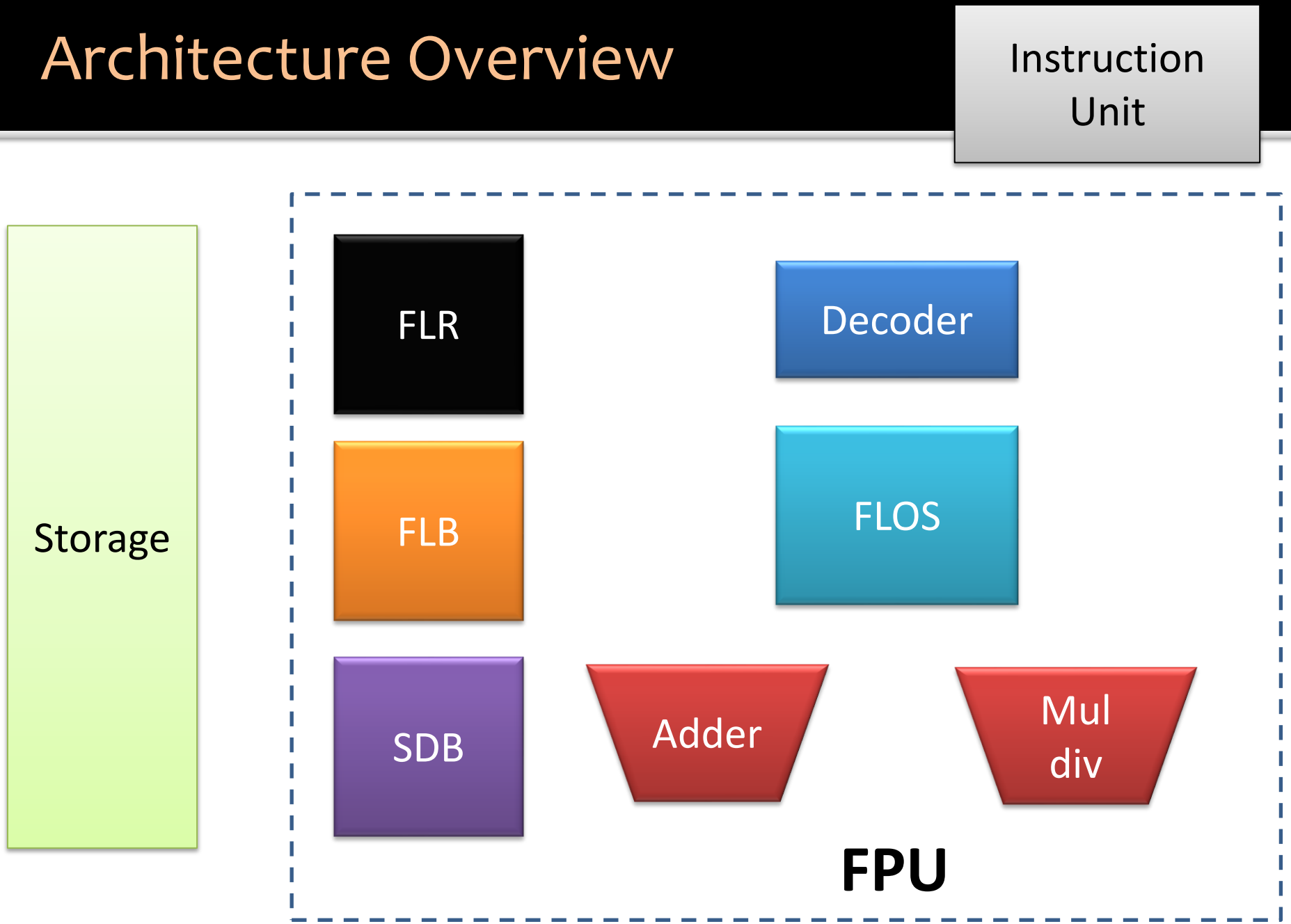
- **Out-of-order** execution of independent instructions
- while preserving the essential **precedences** in the instruction stream

Doubt

- When people talk about Tomasolu algorithm, they talk about register renaming
- However this word can't be found in the original paper

How could anyone invent a thing without noticing it?

Architecture Overview



From a FPU's perspective

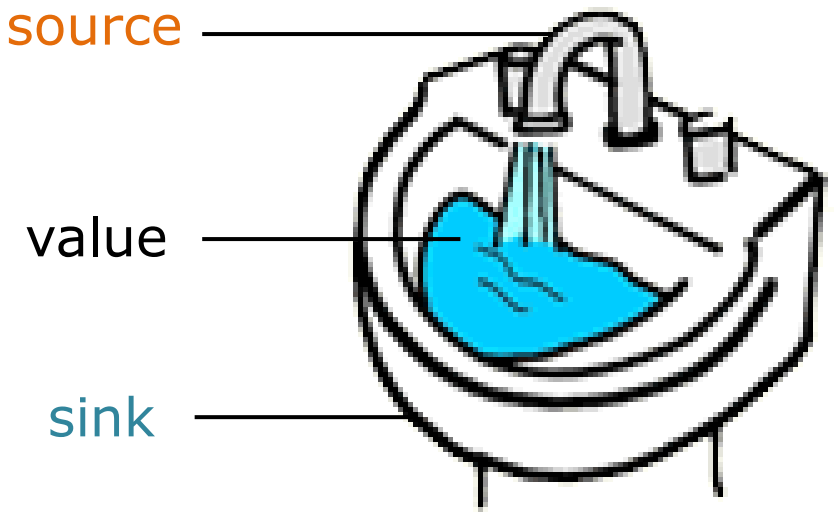
All instructions are 'register-to-register'

- Register-to-register arithmetic
- Storage-to-register arithmetic
- Load
- Store

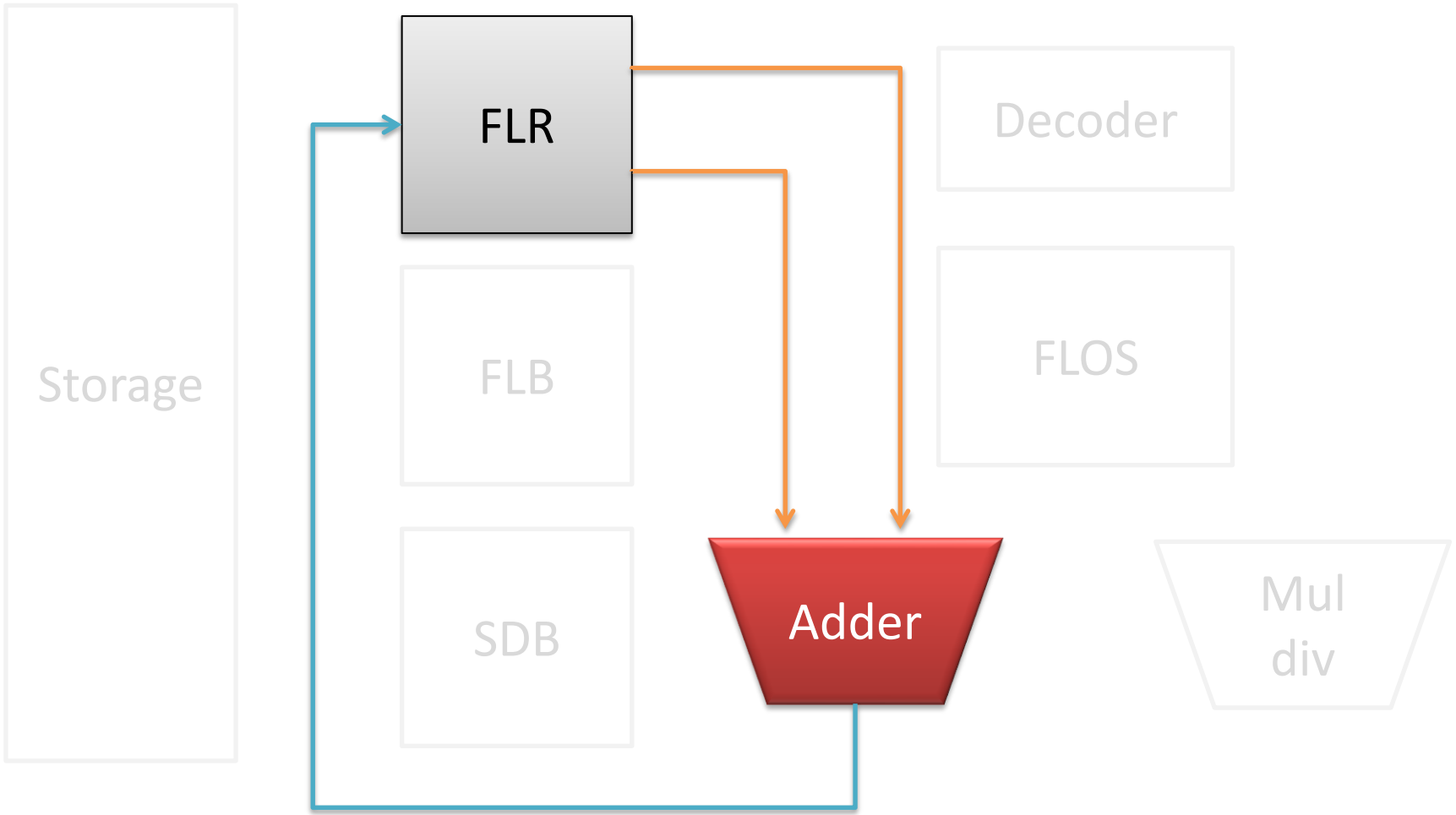
Instruction Unit(outside FPU) is in charge of the address generation and memory access.

‘sink’ and ‘source’

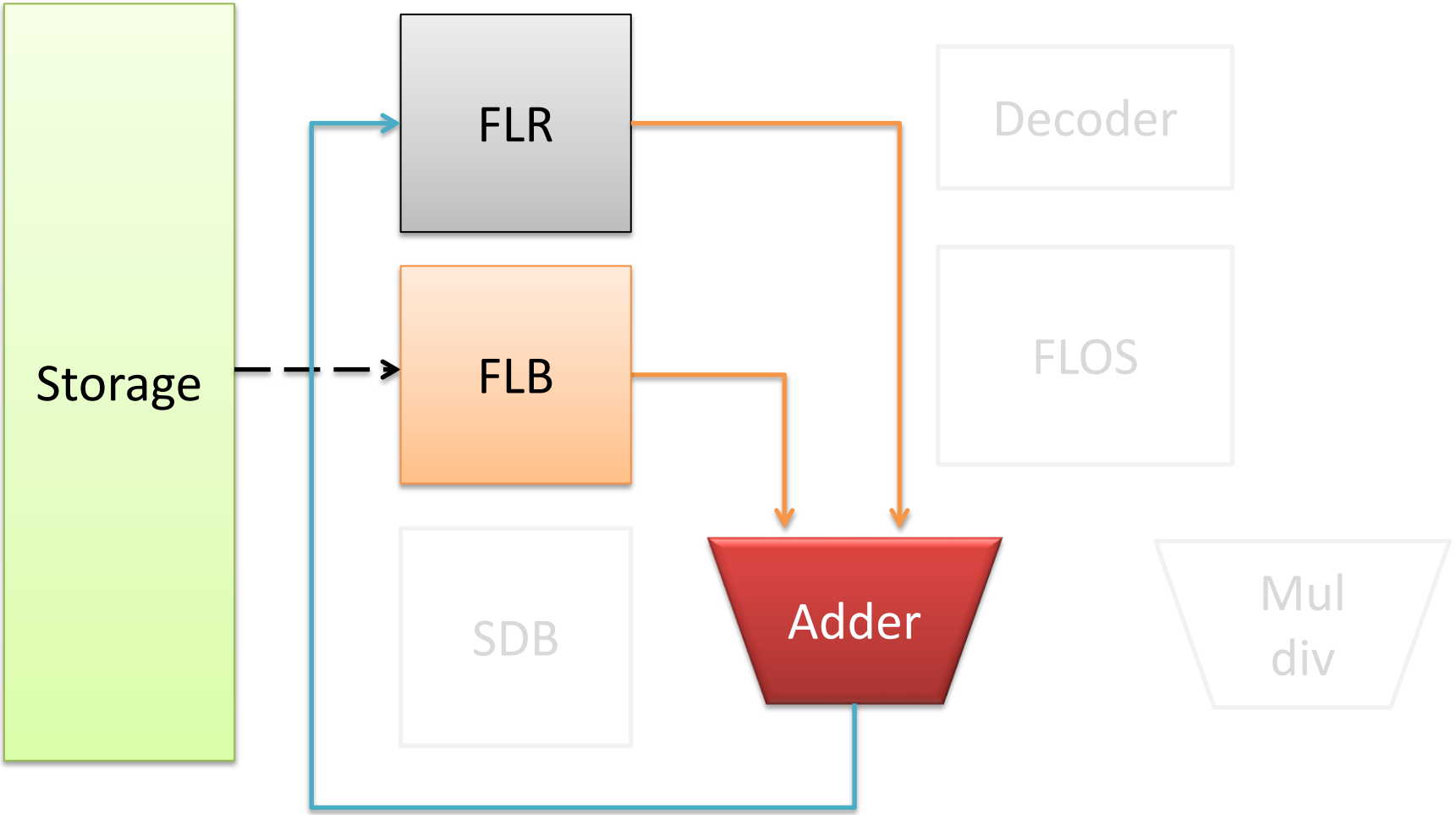
- Be equivalent to *destination* and *source*
- For example, AD R1, R2
- R1 is both a *sink* and a *source*



1.Reg-to-reg arithmetic AD R1, R2

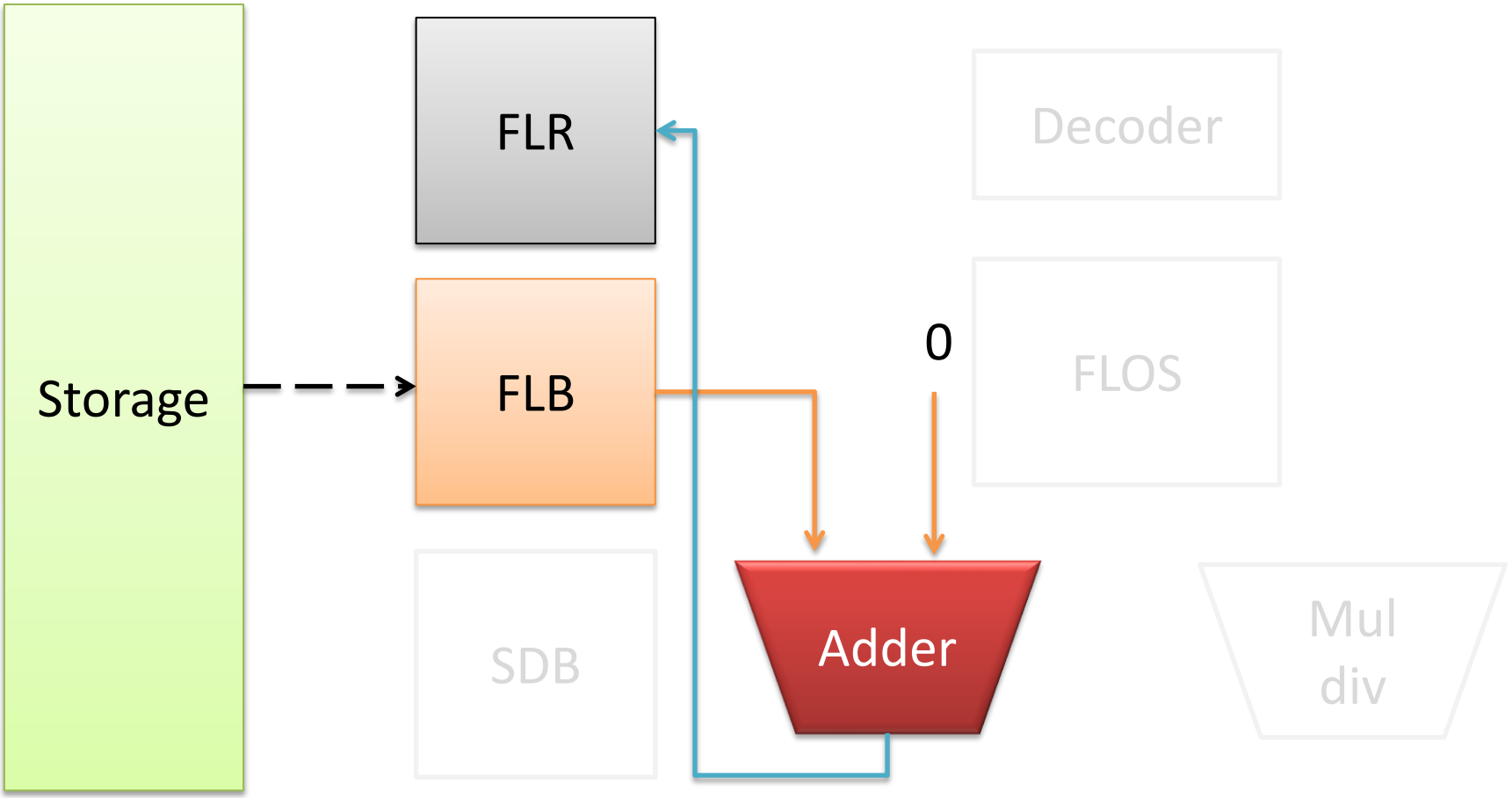


2.Storage-to-reg arithmetic AD R1, FLB



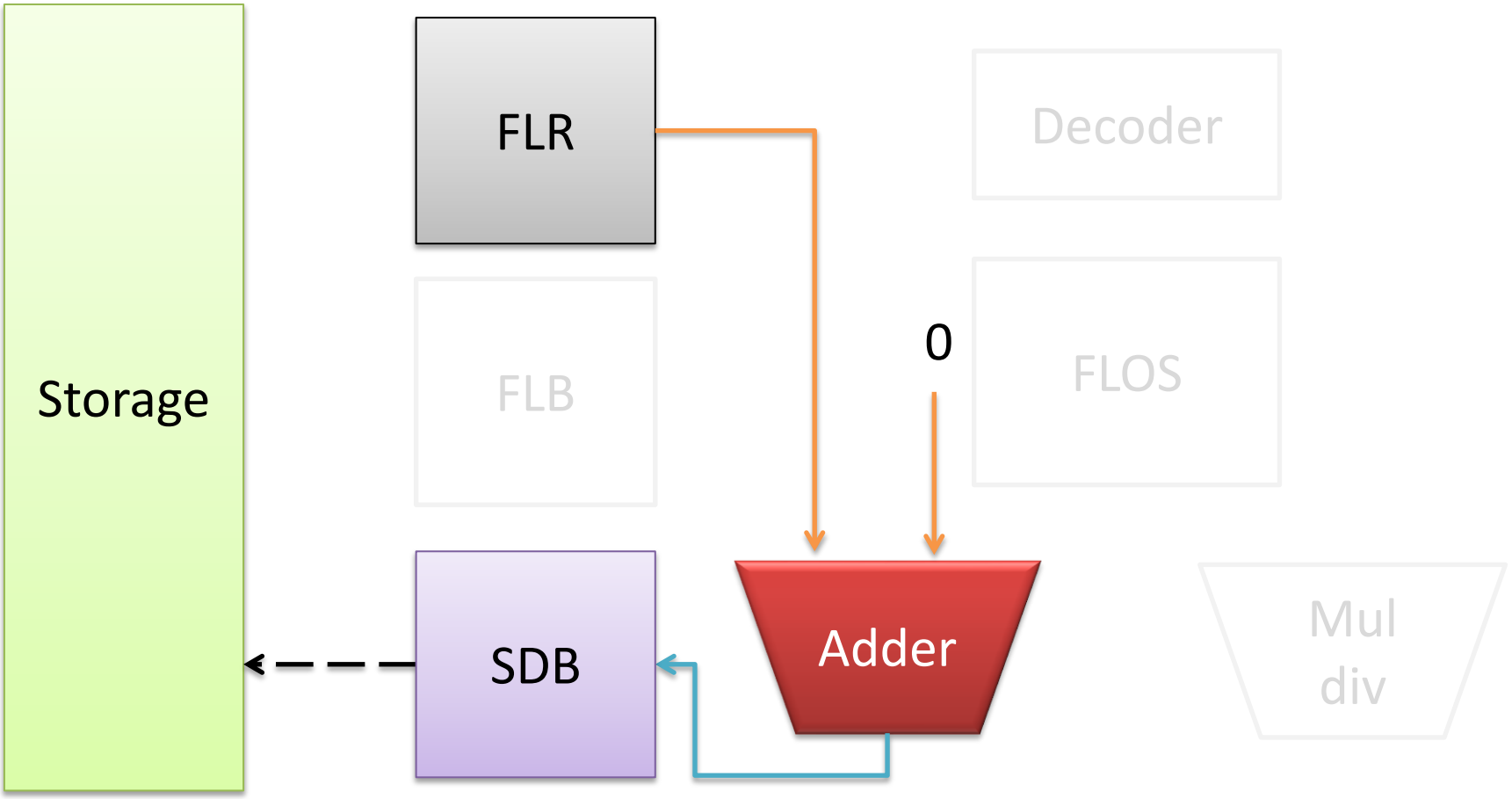
3.Load

LD R1, FLB1

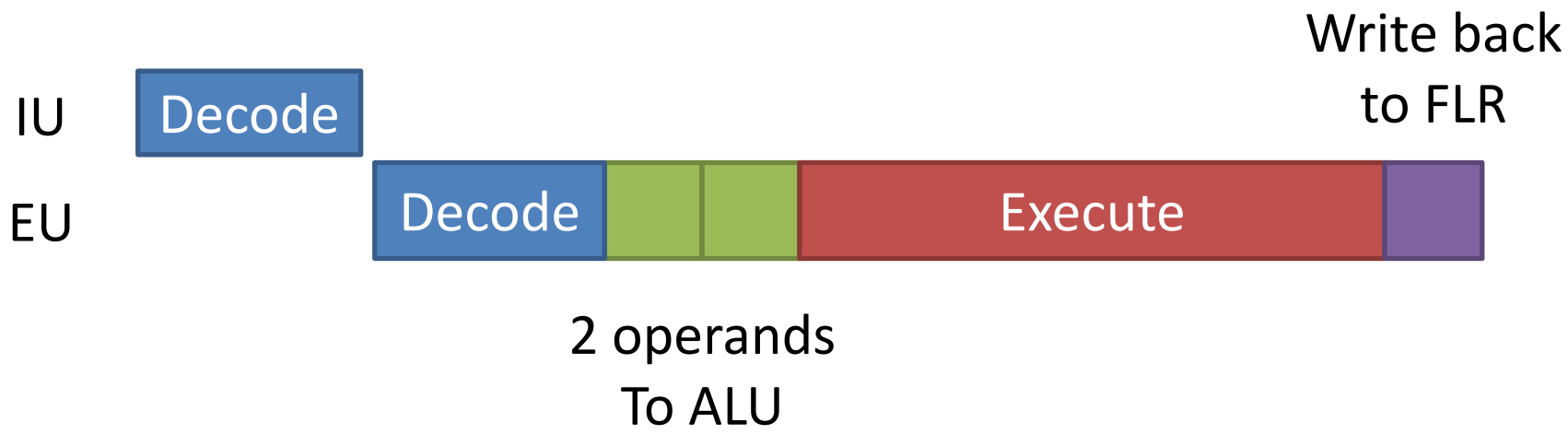


4.Store

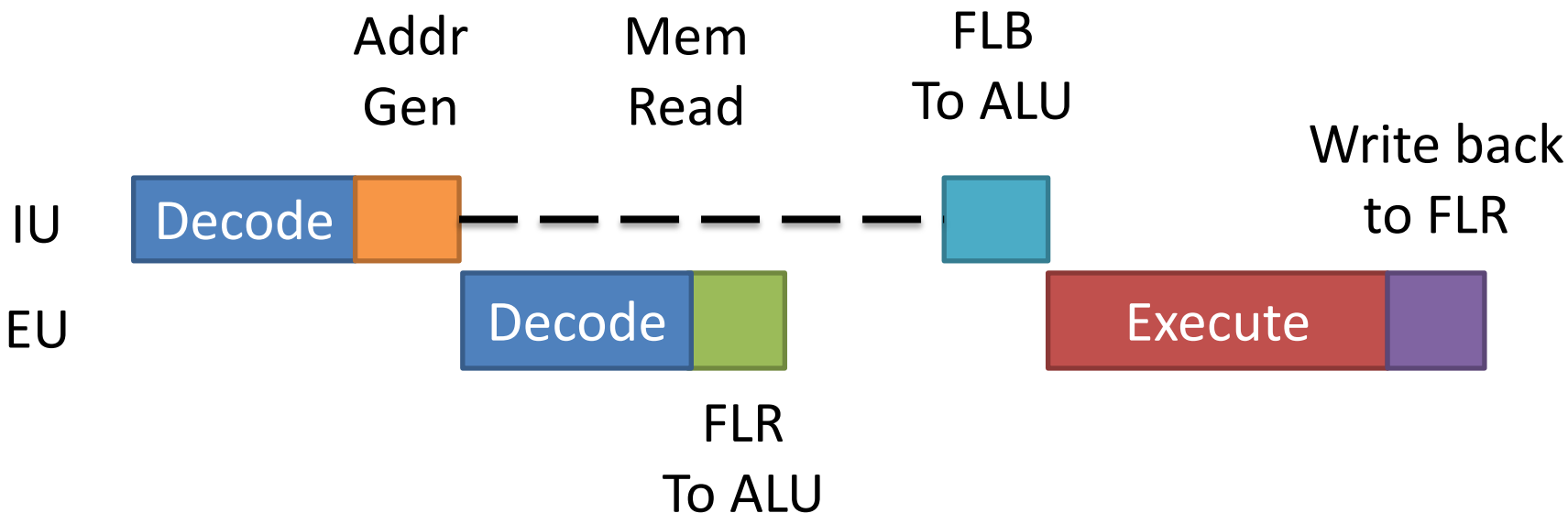
STD R1, SDB1



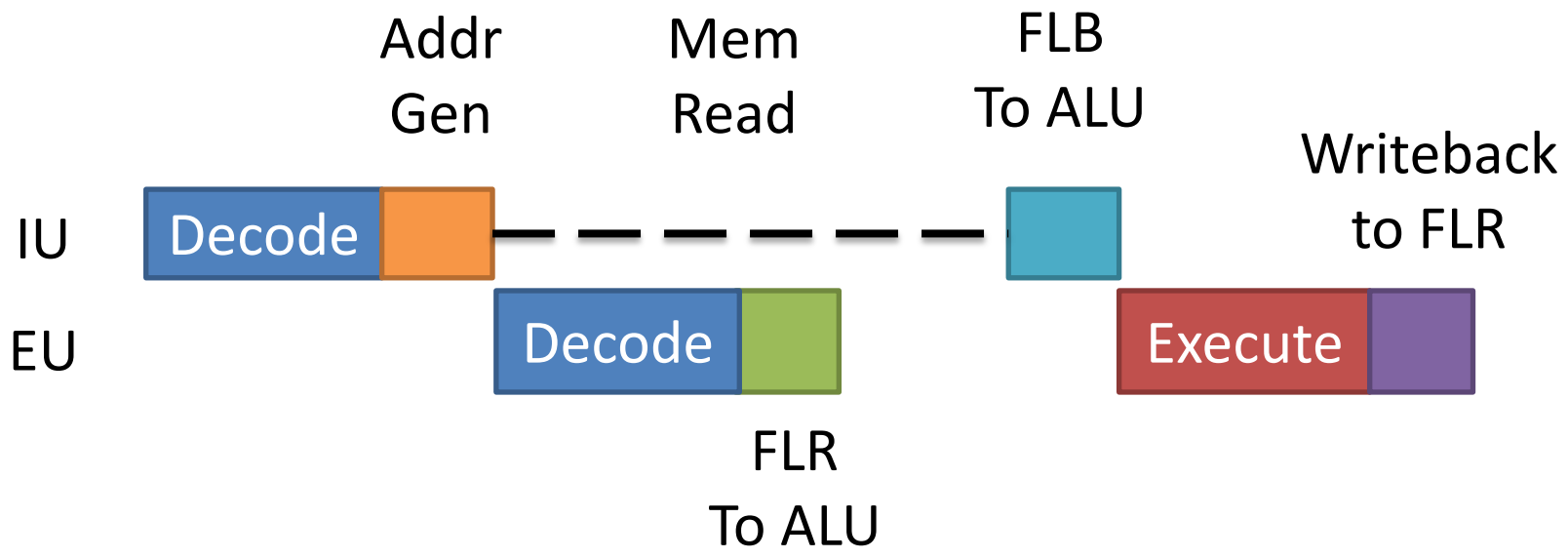
Timing Sequence: 1. reg-to reg arithmetic



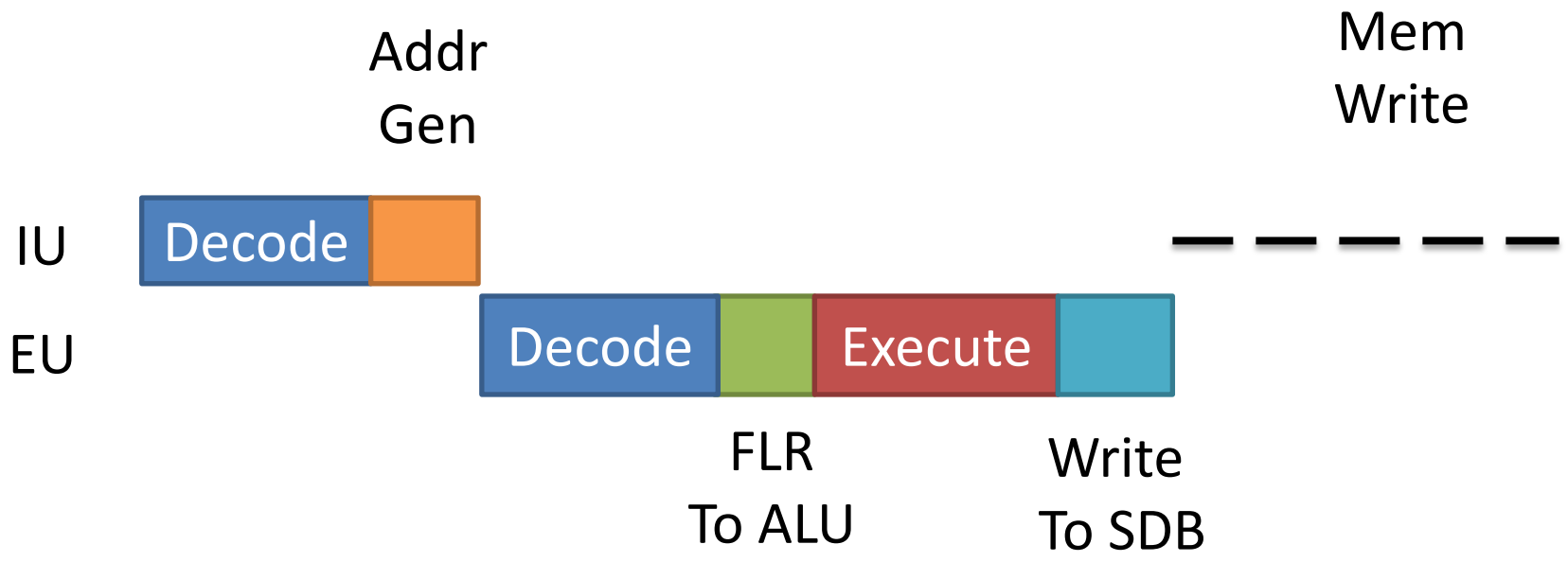
2. storage-to-reg arithmetic



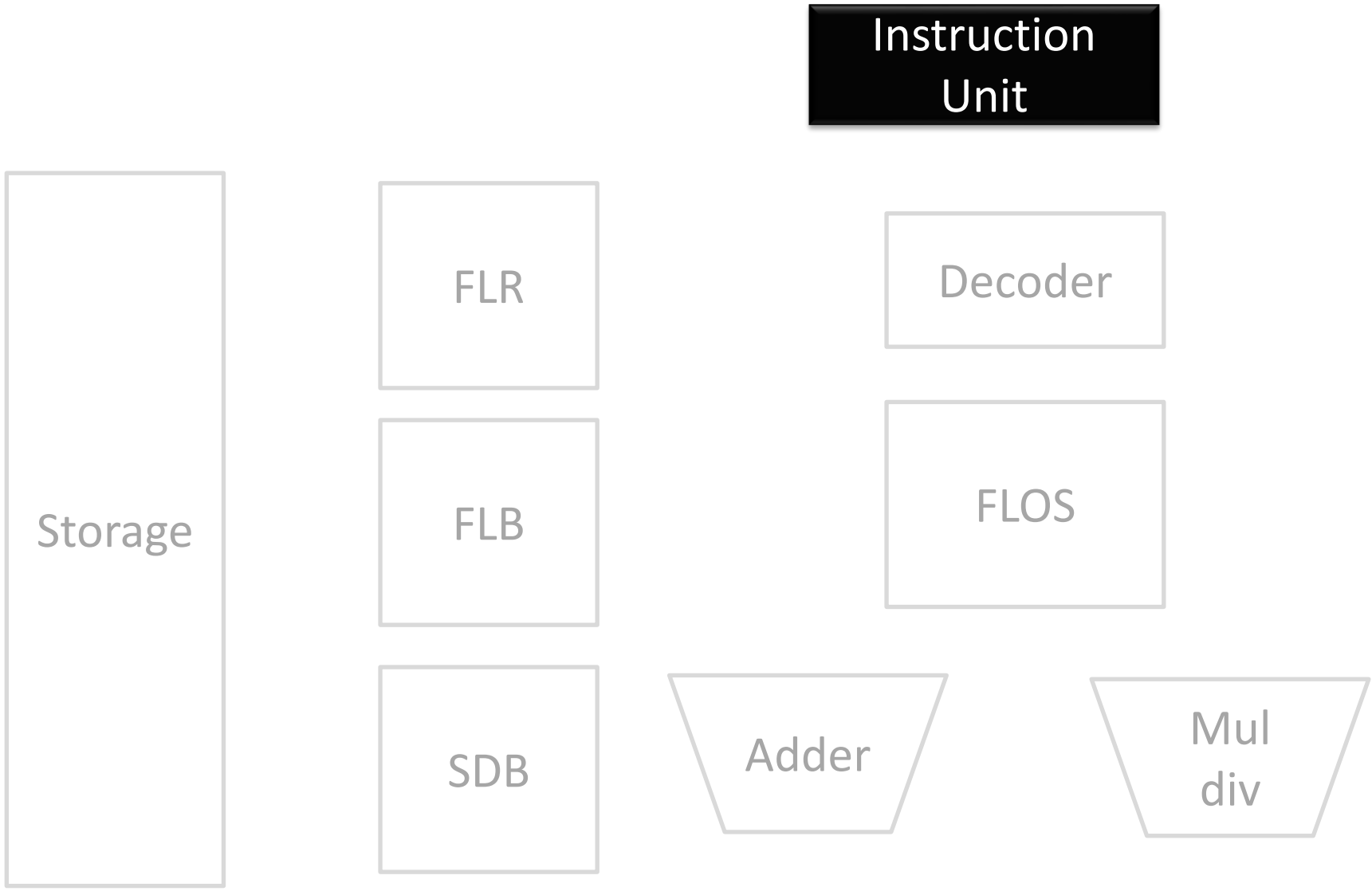
3.Load


Addr
Gen
Mem
Read
FLB
To ALU
Writeback
to FLR
IU
Decode
EU
Decode
Execute
FLR
To ALU

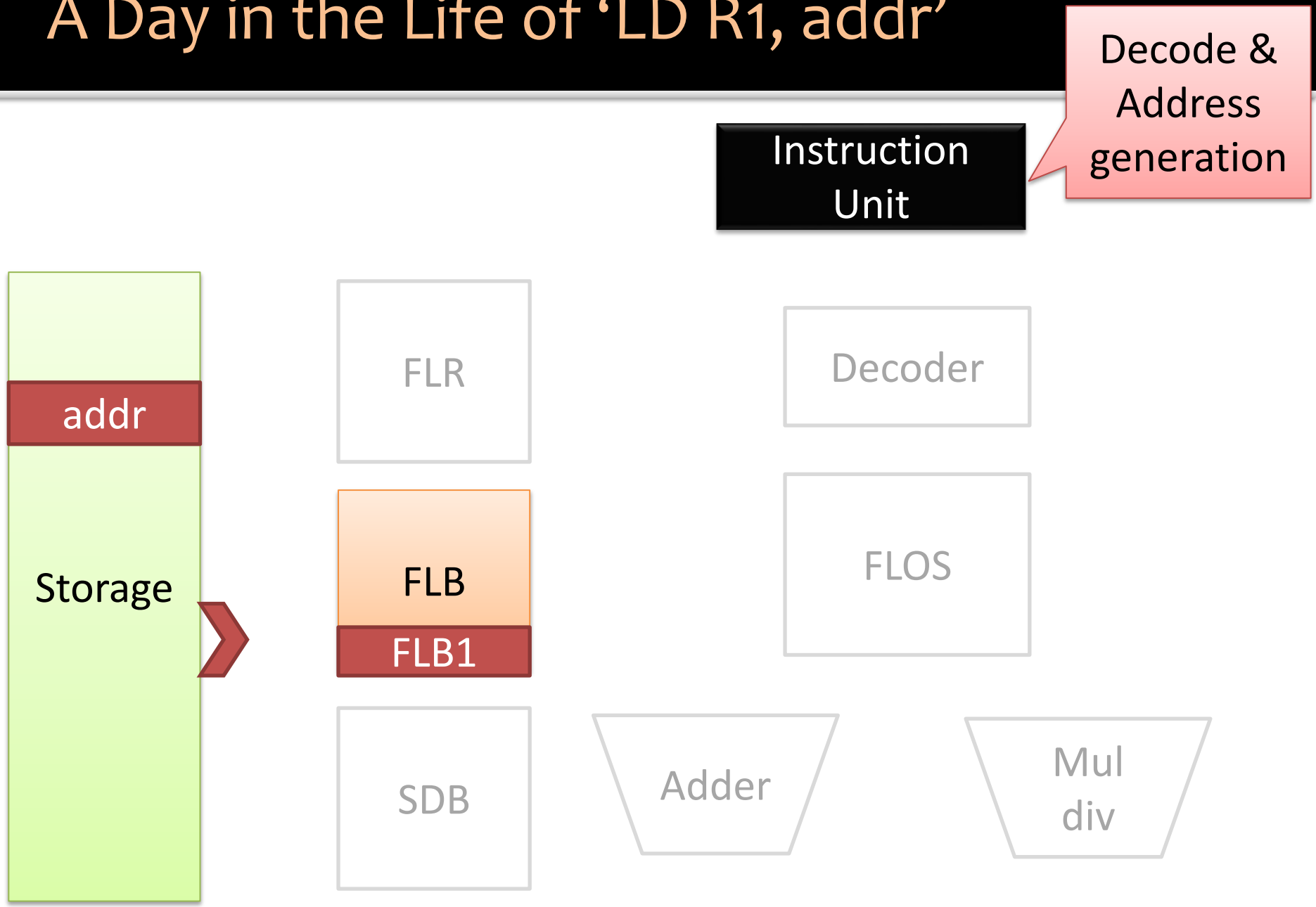
4.Store


Addr
Gen
Mem
Write
IU
Decode
EU
Decode
Execute
FLR
To ALU
Write
To SDB

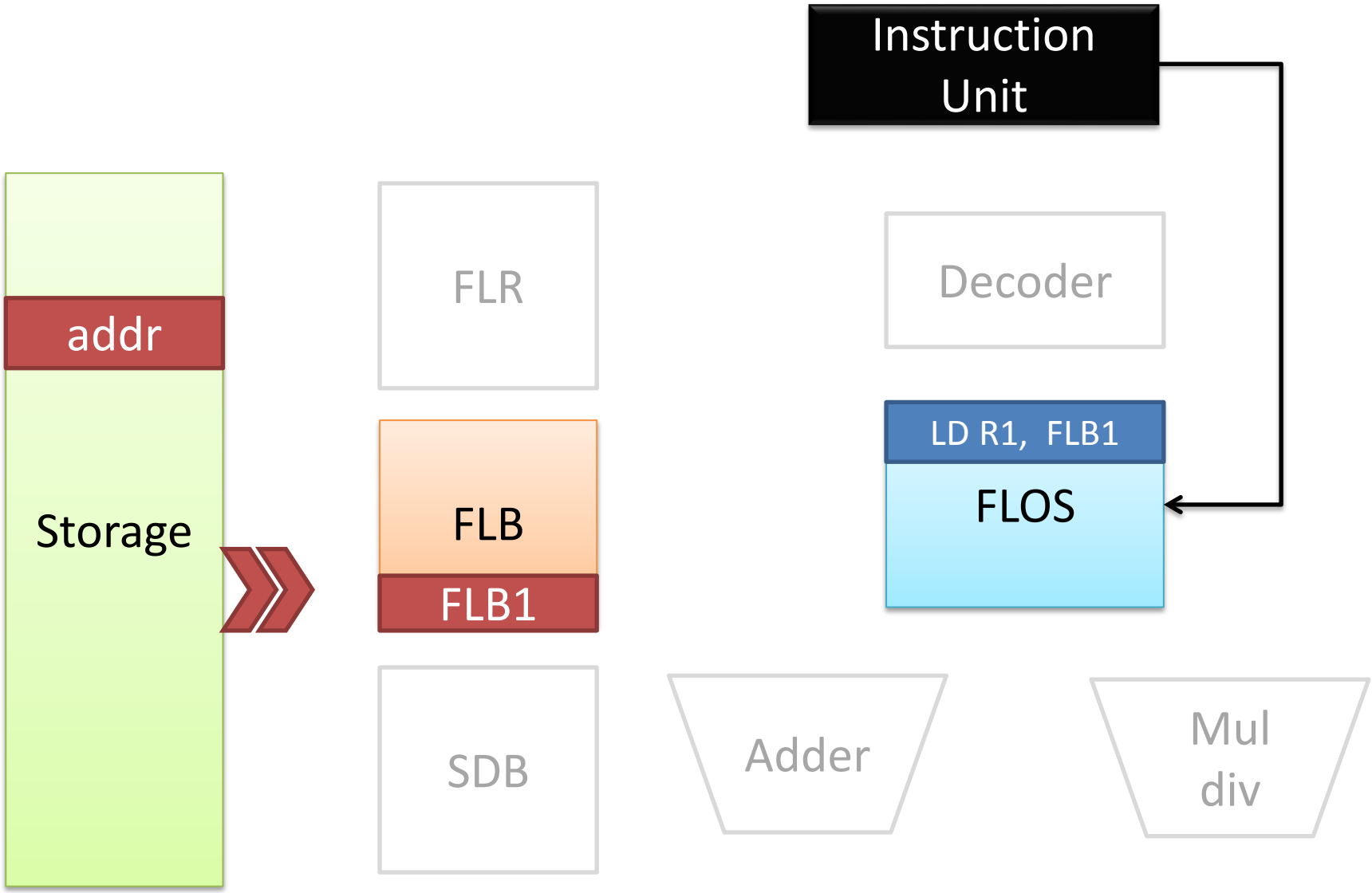
A Day in the Life of ‘LD R1, addr’



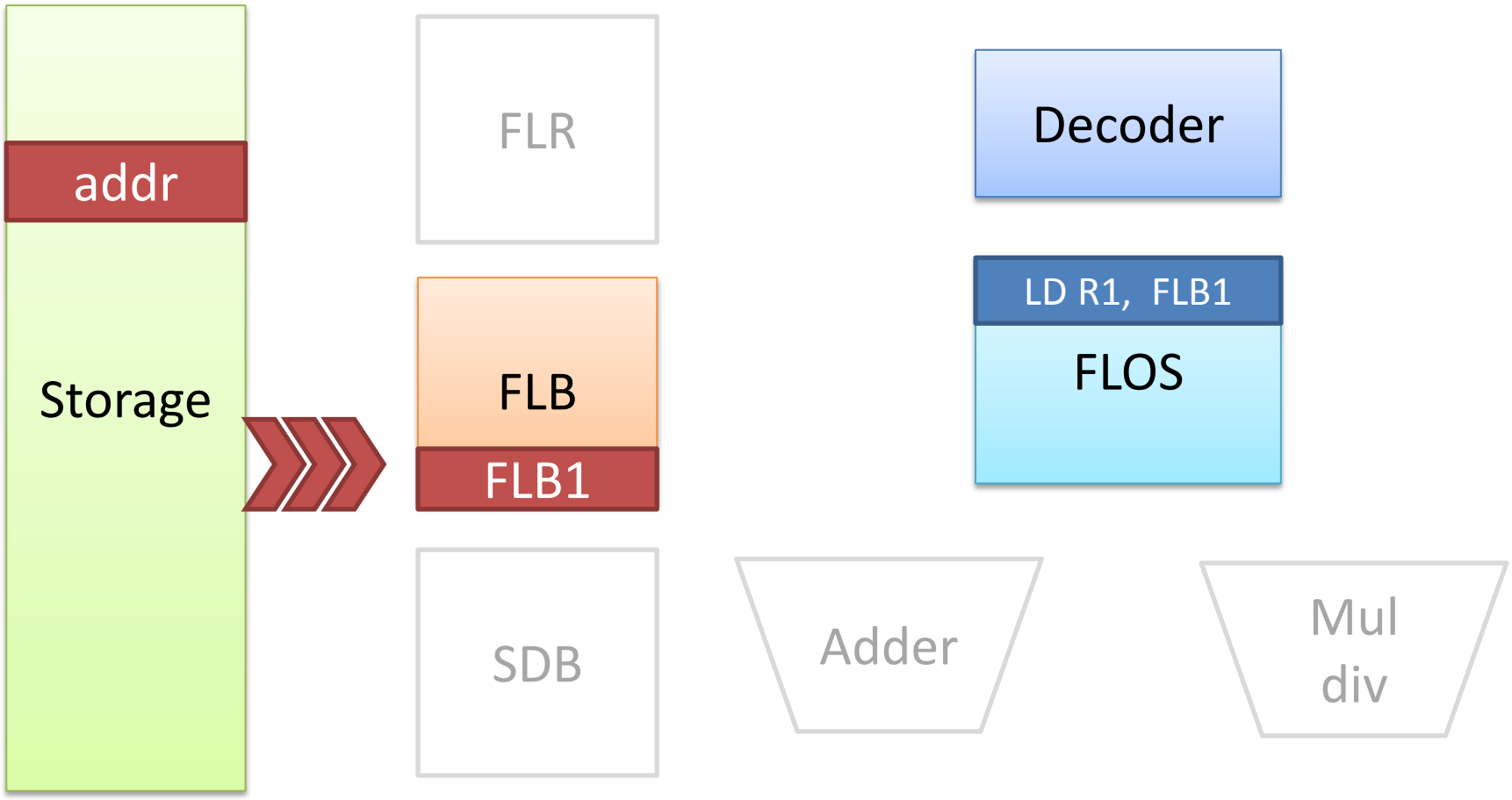
A Day in the Life of 'LD R1, addr'



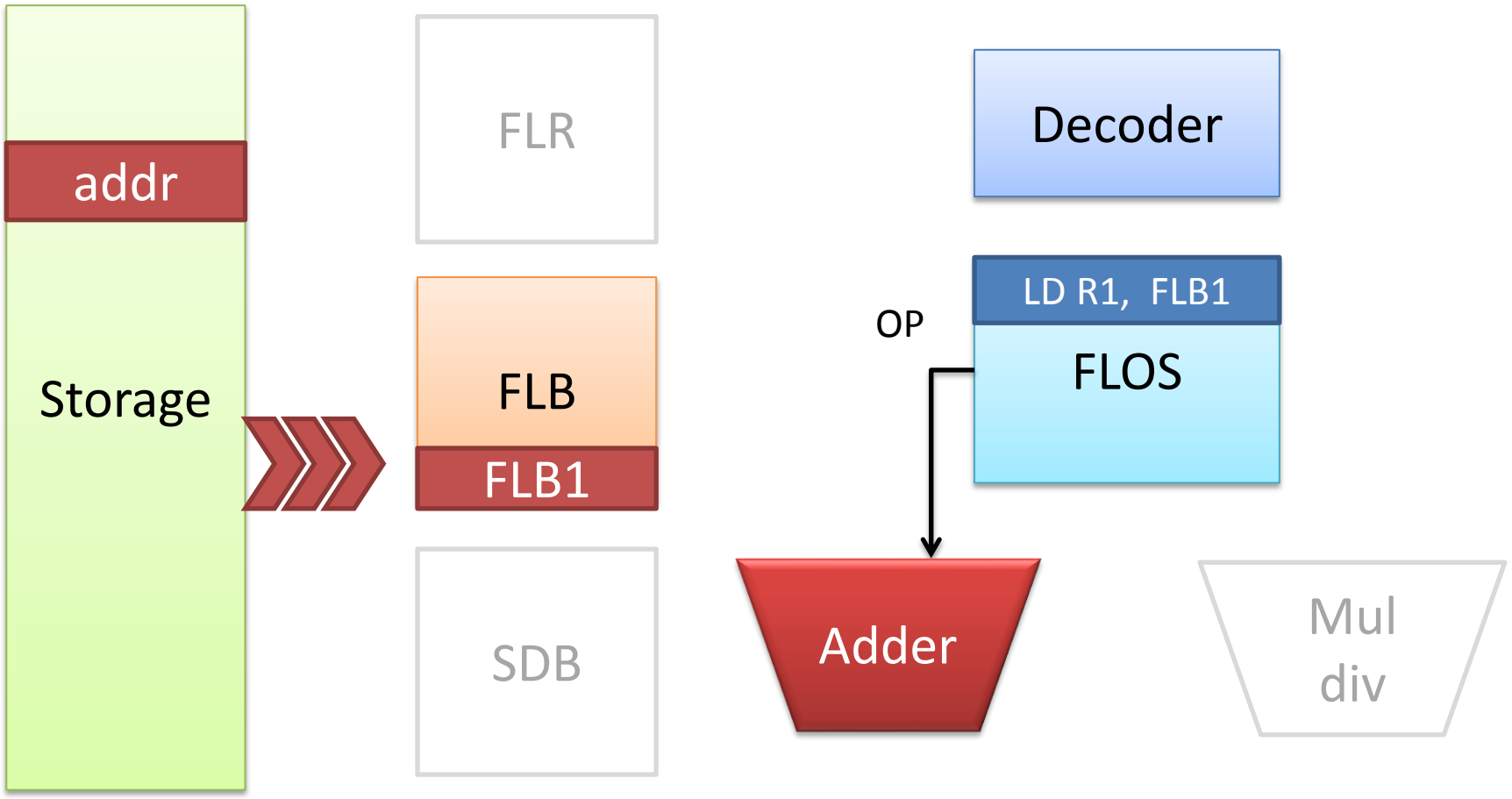
A Day in the Life of 'LD R1, addr'



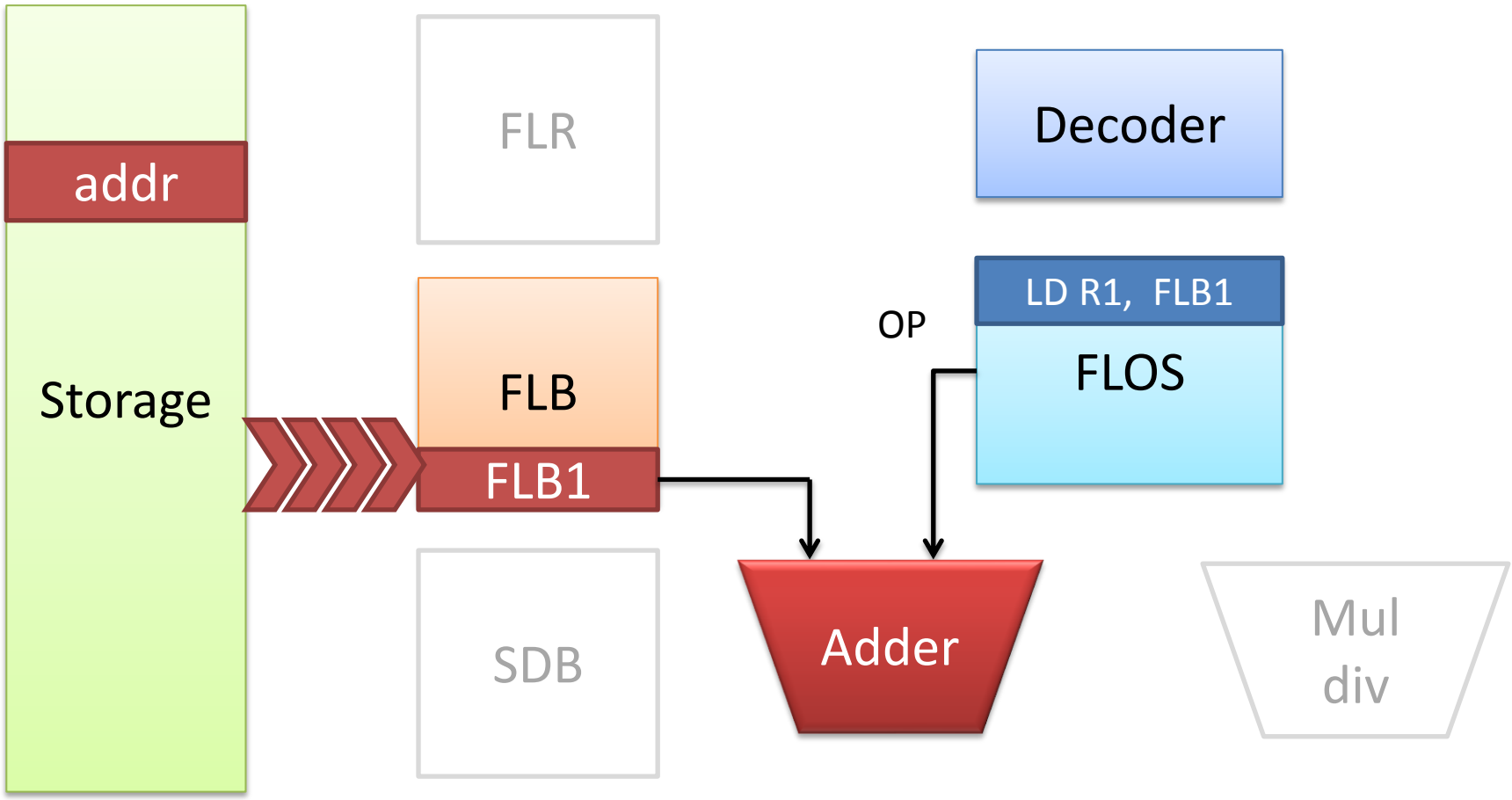
A Day in the Life of ‘LD R1, addr’



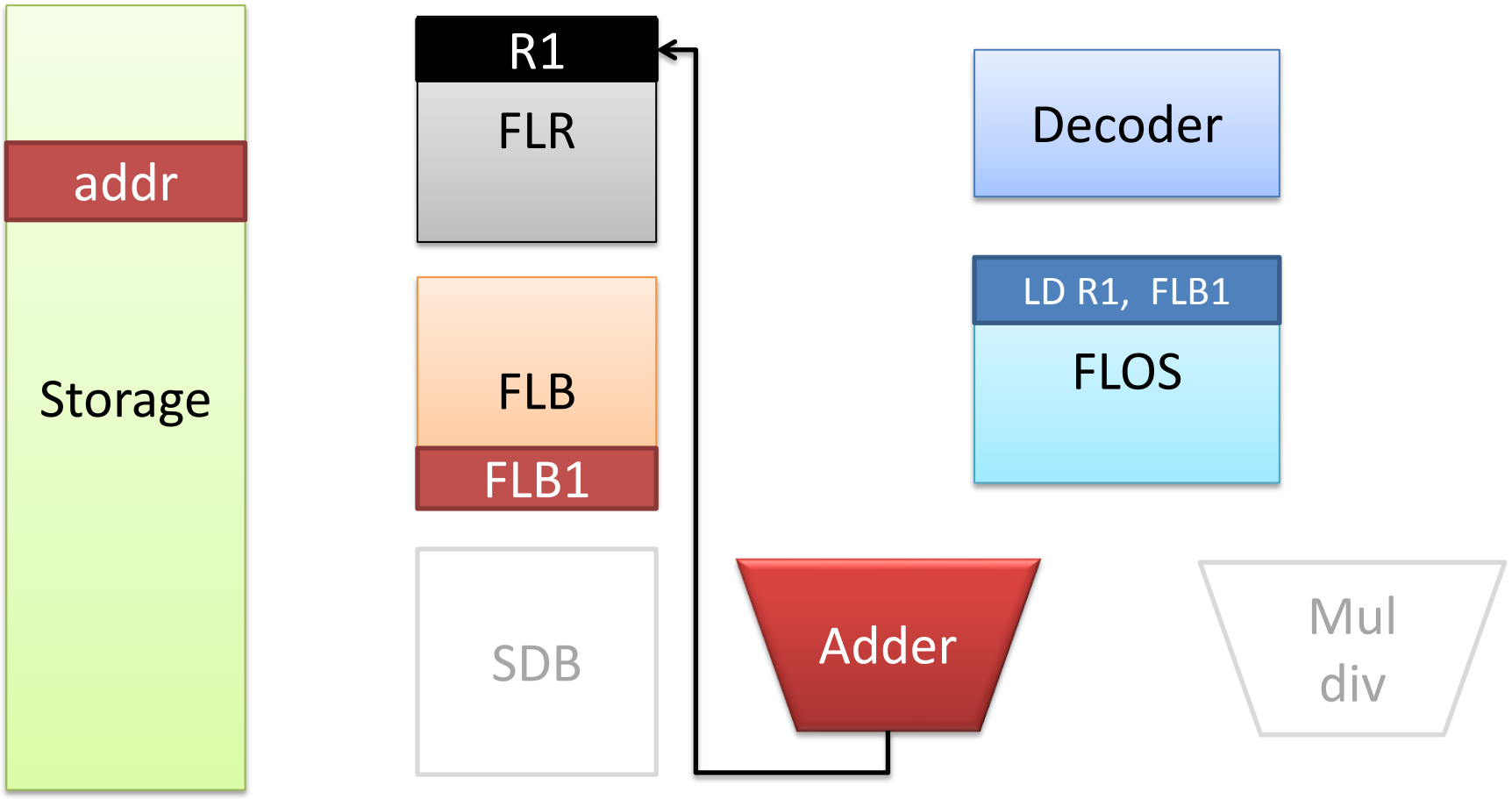
A Day in the Life of ‘LD R1, addr’



A Day in the Life of ‘LD R1, addr’



A Day in the Life of ‘LD R1, addr’



An Example of Dependence

```
LD F0, FLB1
MD F0, FLB2
```

to exploit parallelisim

What if send them to different execution units at the same time?

An Example of Dependence

```
LD F0, FLB1
MD F0, FLB2
```

The result(F0) cannot reflect the impact of LD, because MD uses the old value of F0

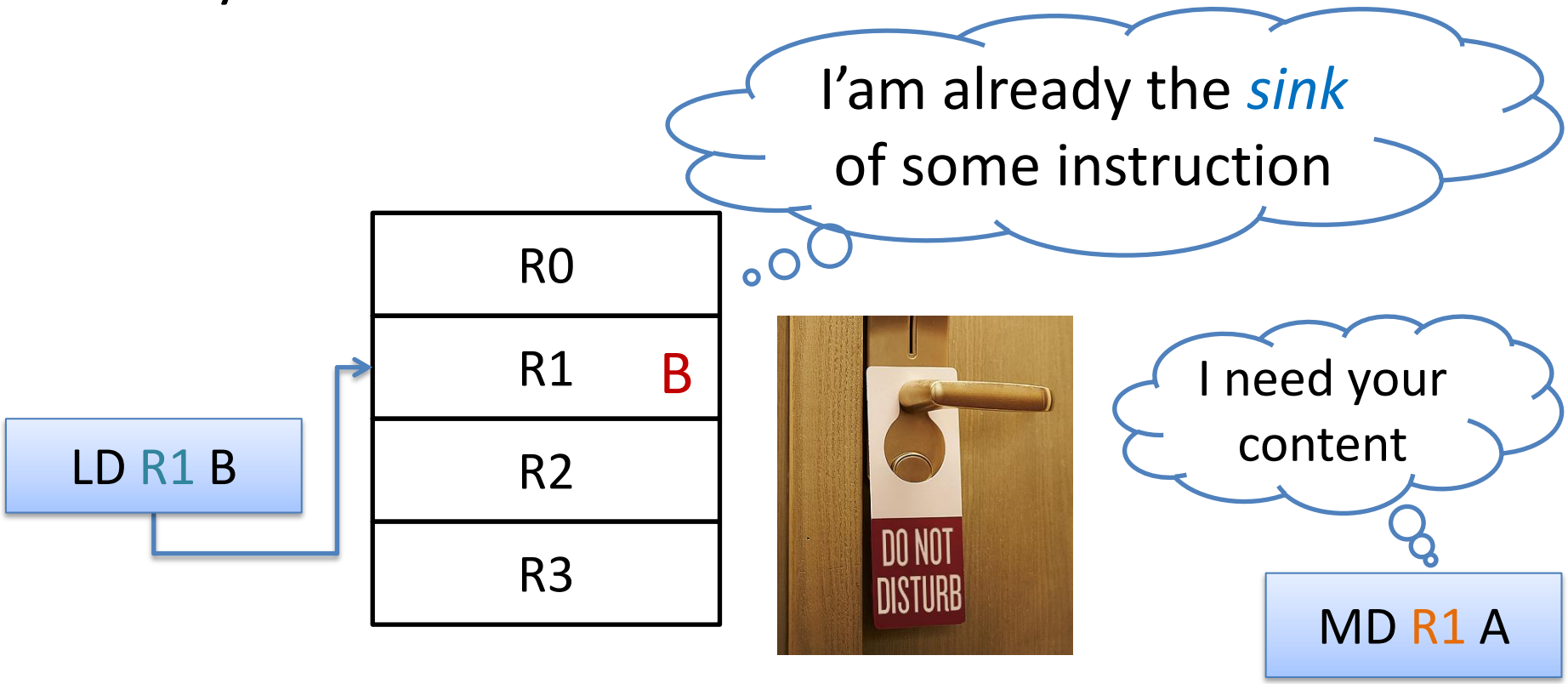
An Example of Dependence

```
LD F0, FLB1
MD F0, FLB2
```

It is also called *true dependence*, *a.k.a. RAW*

A Simple Solution

- ‘busy’ bit scheme



Performance Degrades…

- When the code keep using one register
- E.g.

```
MD F0, E
AD F2, F0
AD F4, A
AD F2, F4
```

The second AD is *qualified* to issue

overlap *fails* because the first AD depends on MD, though the others don't

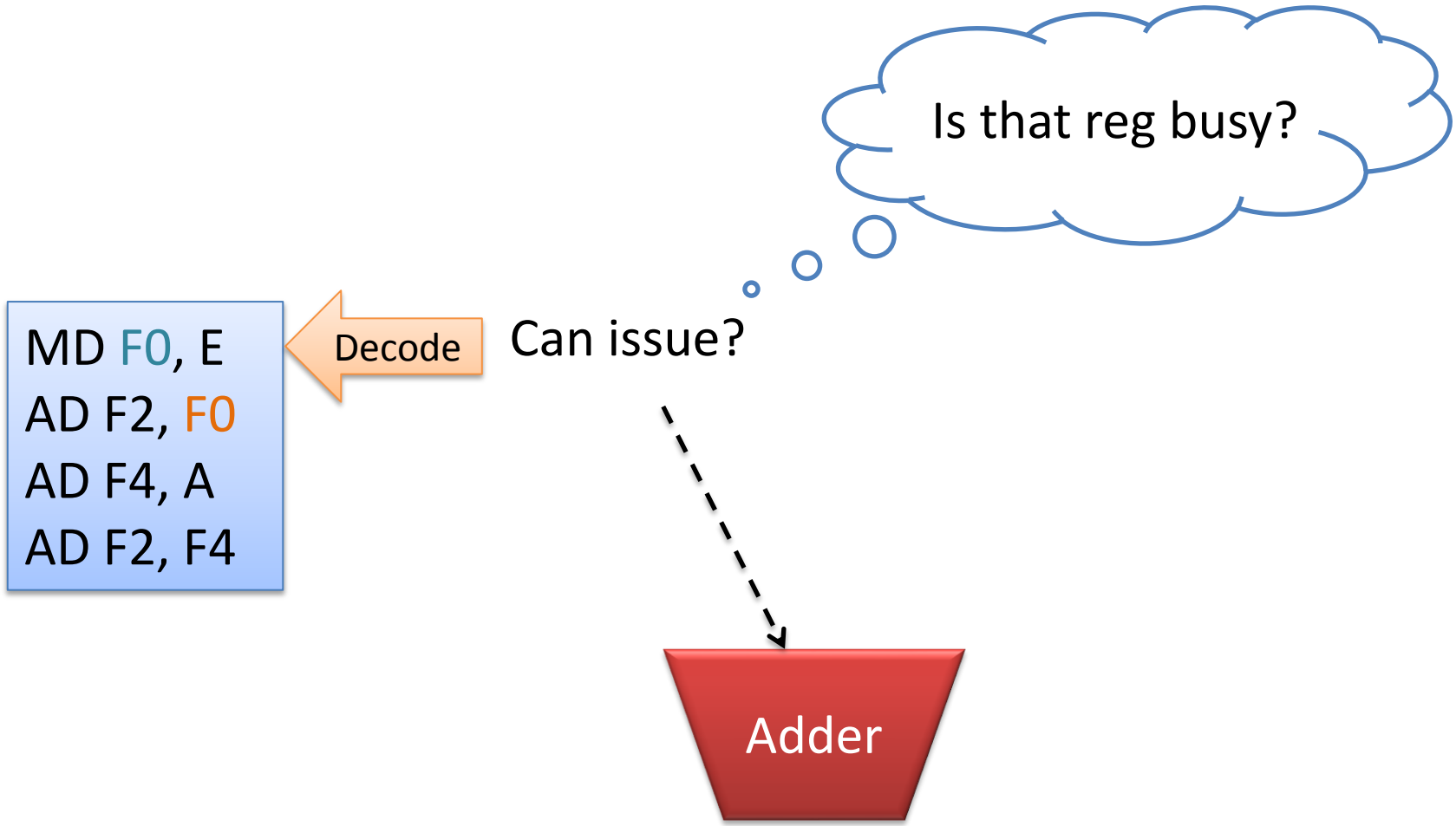
Cause of the Problem

- If one instruction gets stuck(due to dependence), the following can't be decoded(even it is *qualified* to issue)

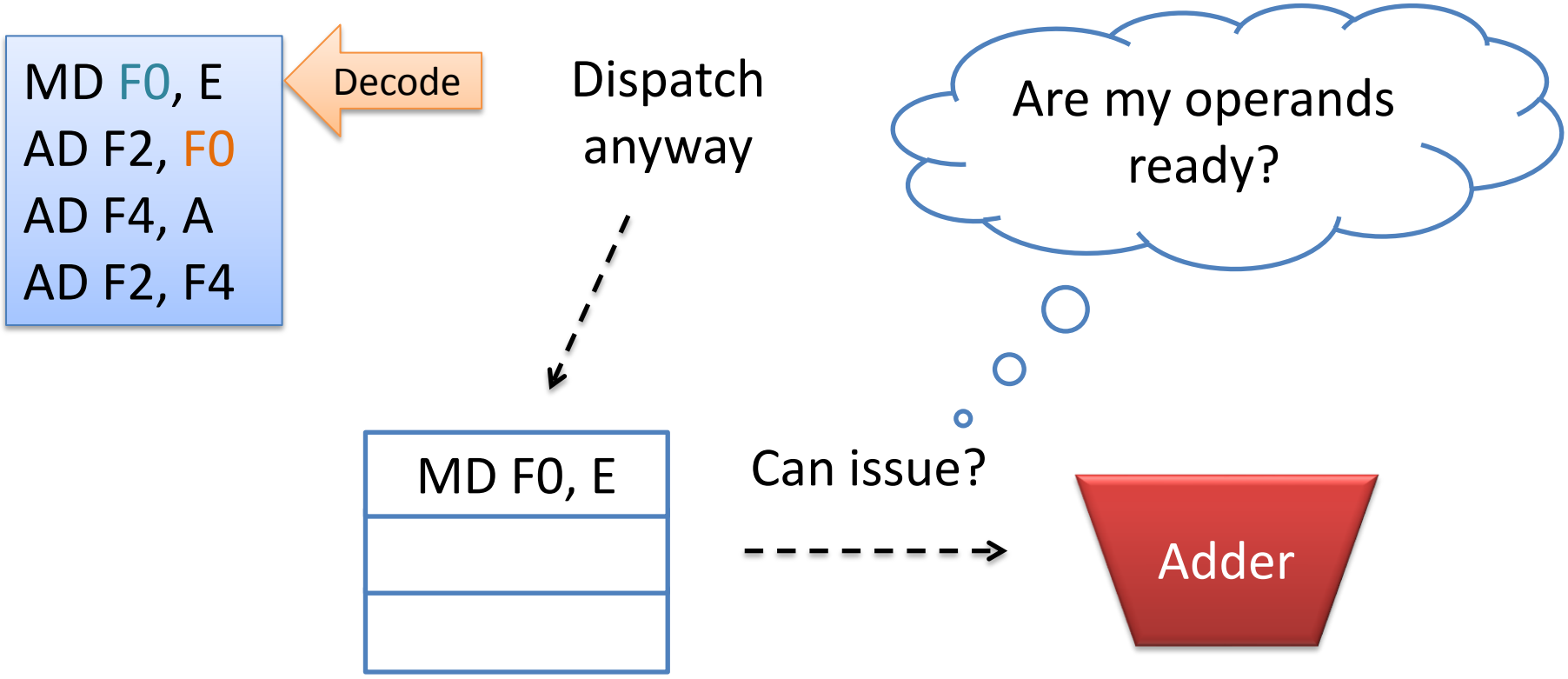
Solution :

- Decouple the dependence mantainance from decoding
- Look ahead more instructions for concurrency

Dispatch and Issue Decoupling



Dispatch and Issue Decoupling



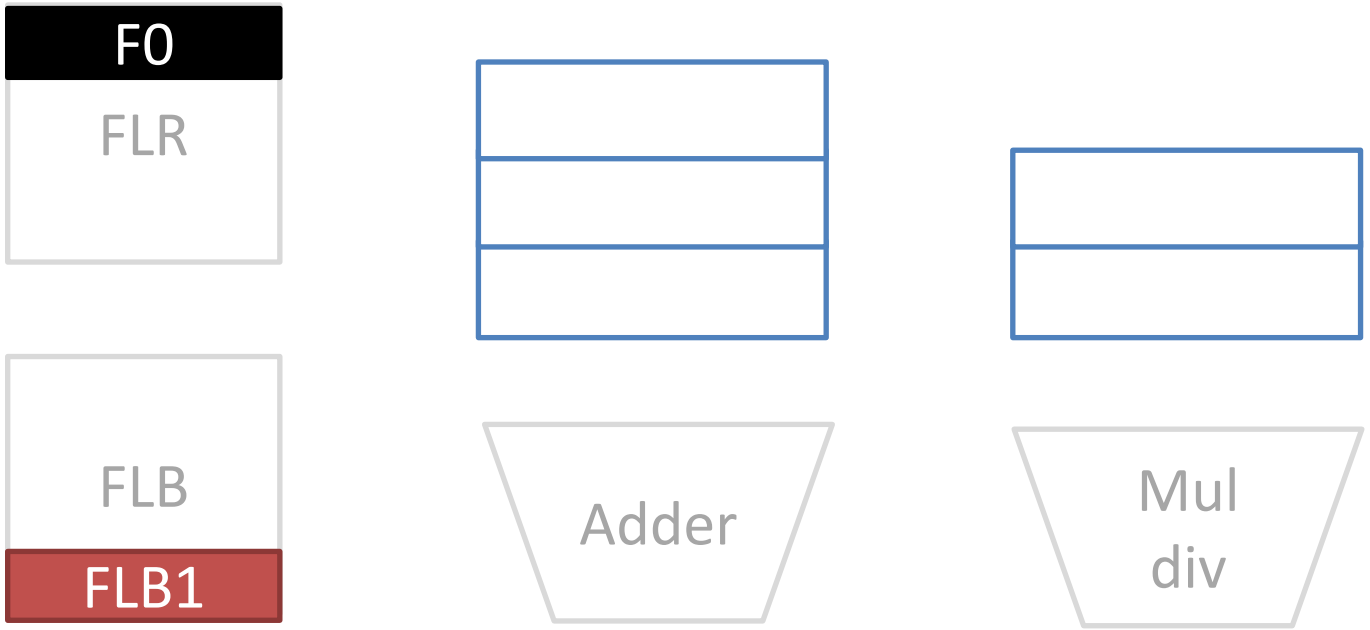
An Example of True Dependence

LD F0, FLB1
AD F2, F0

F0 as *sink*

F0 as *source*

Assume CDB has not been introduced yet

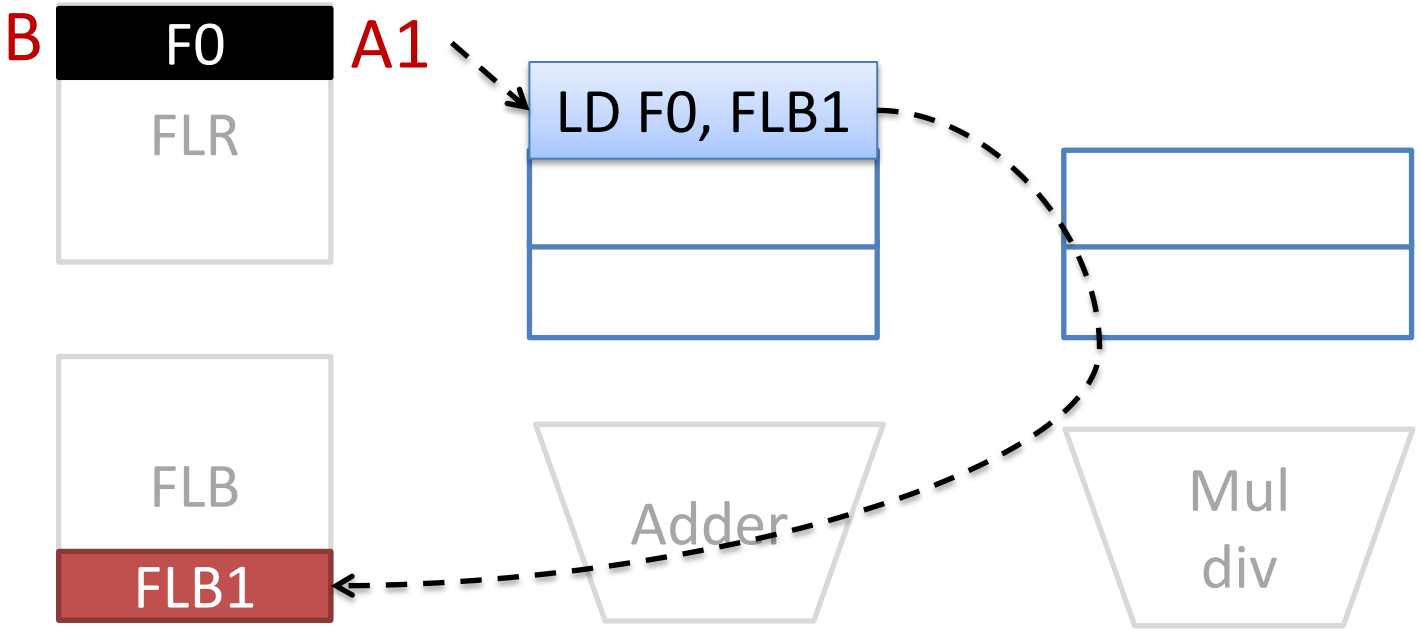


An Example of True Dependence

LD F0, FLB1
AD F2, F0

dispatches to A1

F0 is reserved for some instruction

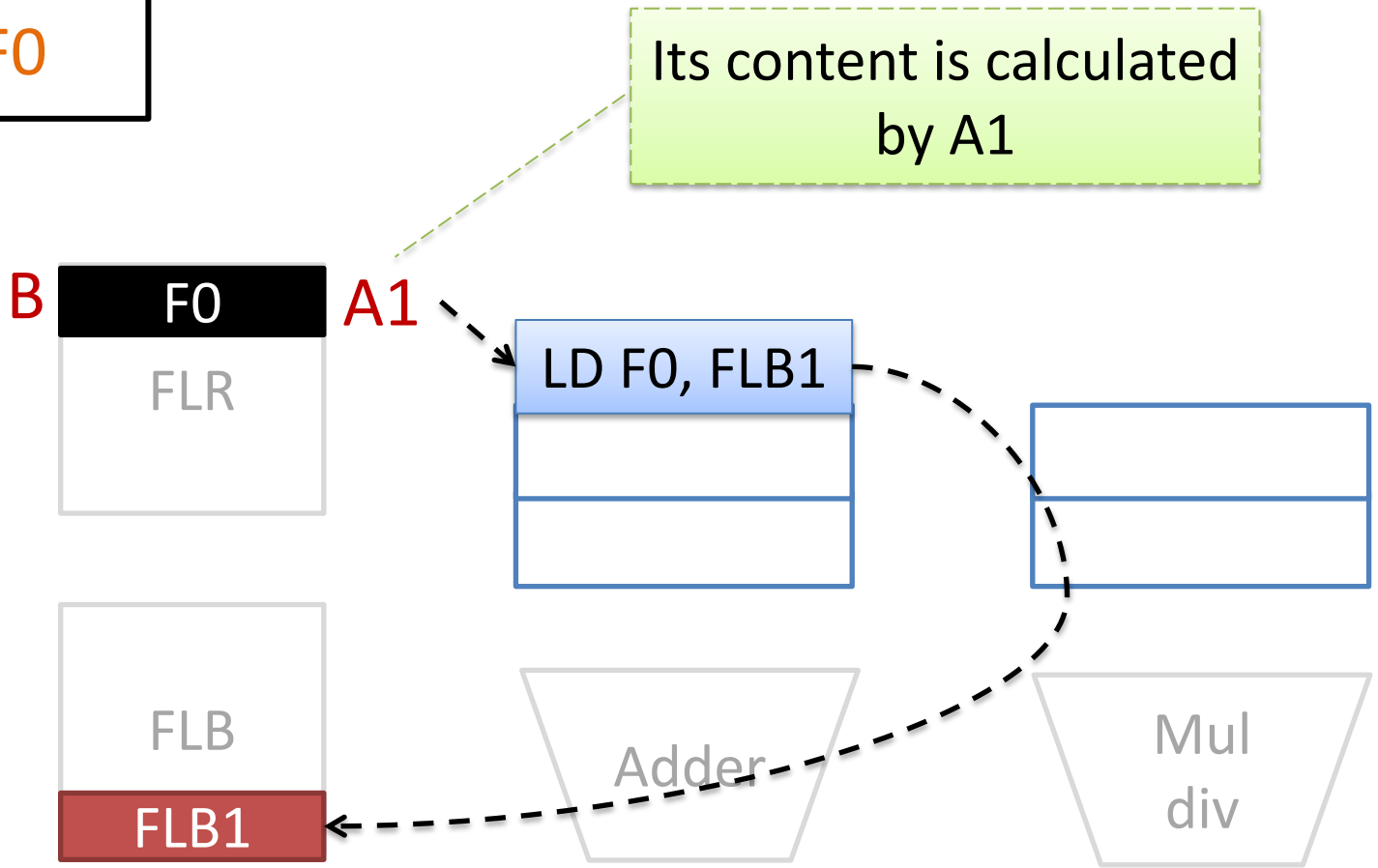


An Example of True Dependence

LD F0, FLB1

AD F2, F0

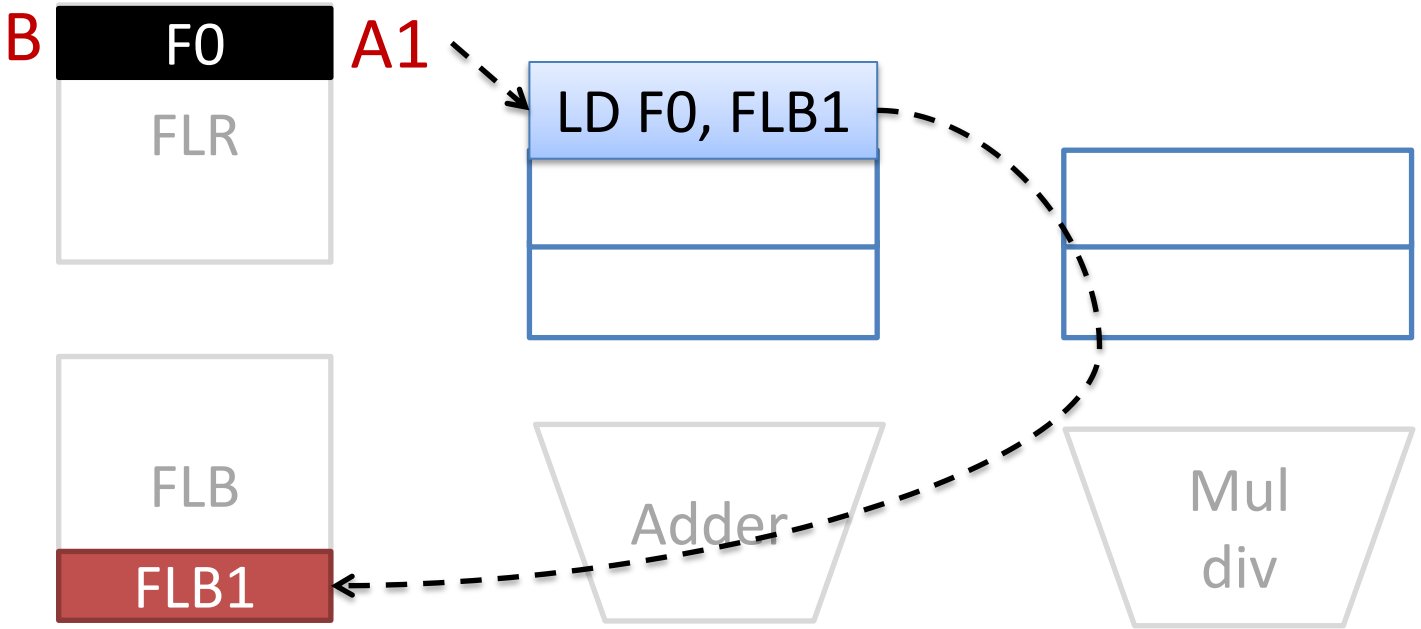
dispatches to A1



An Example of True Dependence

LD F0, FLB1
AD F2, F0

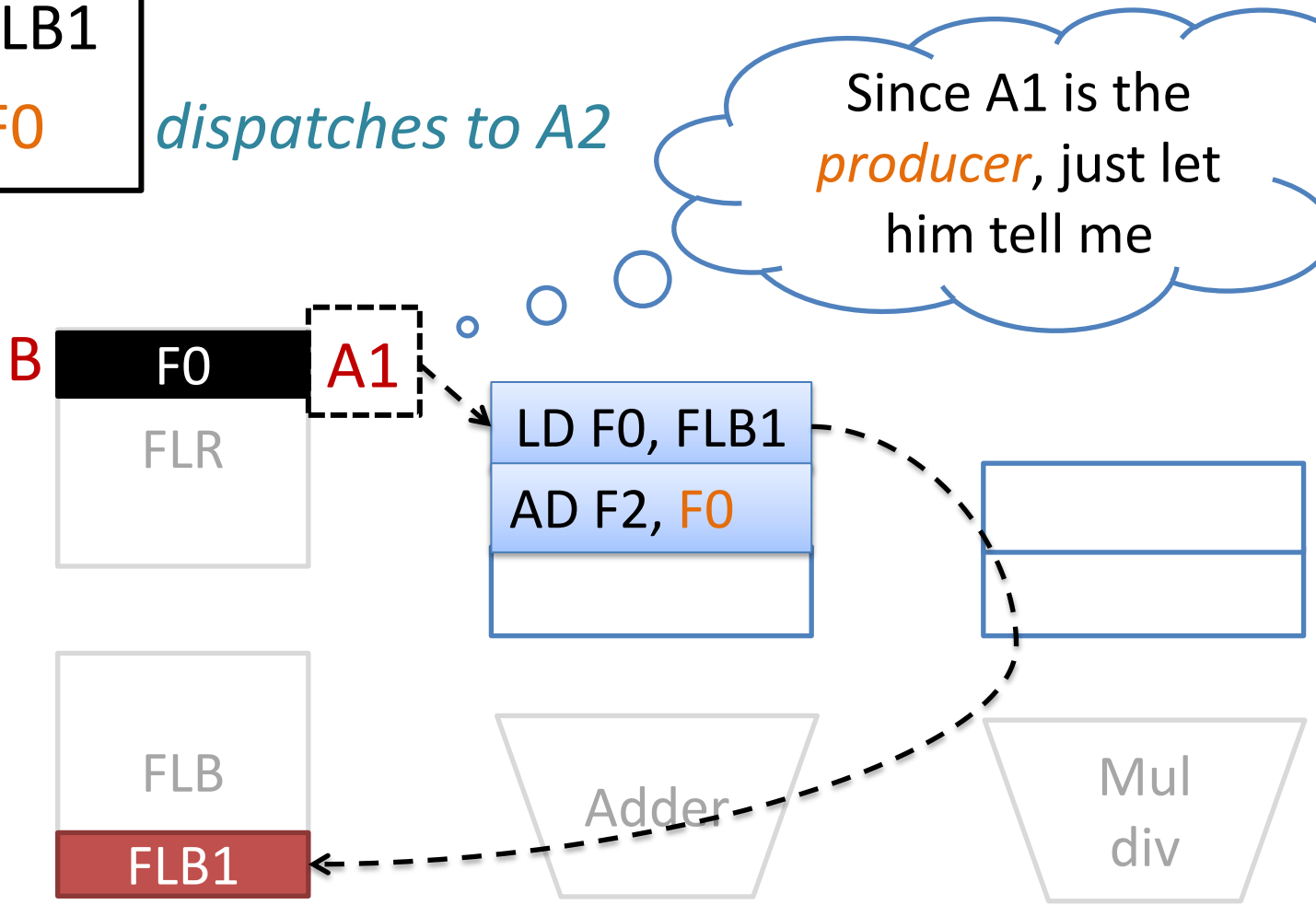
I need the value of F0,
but he seems to be busy



An Example of True Dependence

LD F0, FLB1
AD F2, F0

dispatches to A2

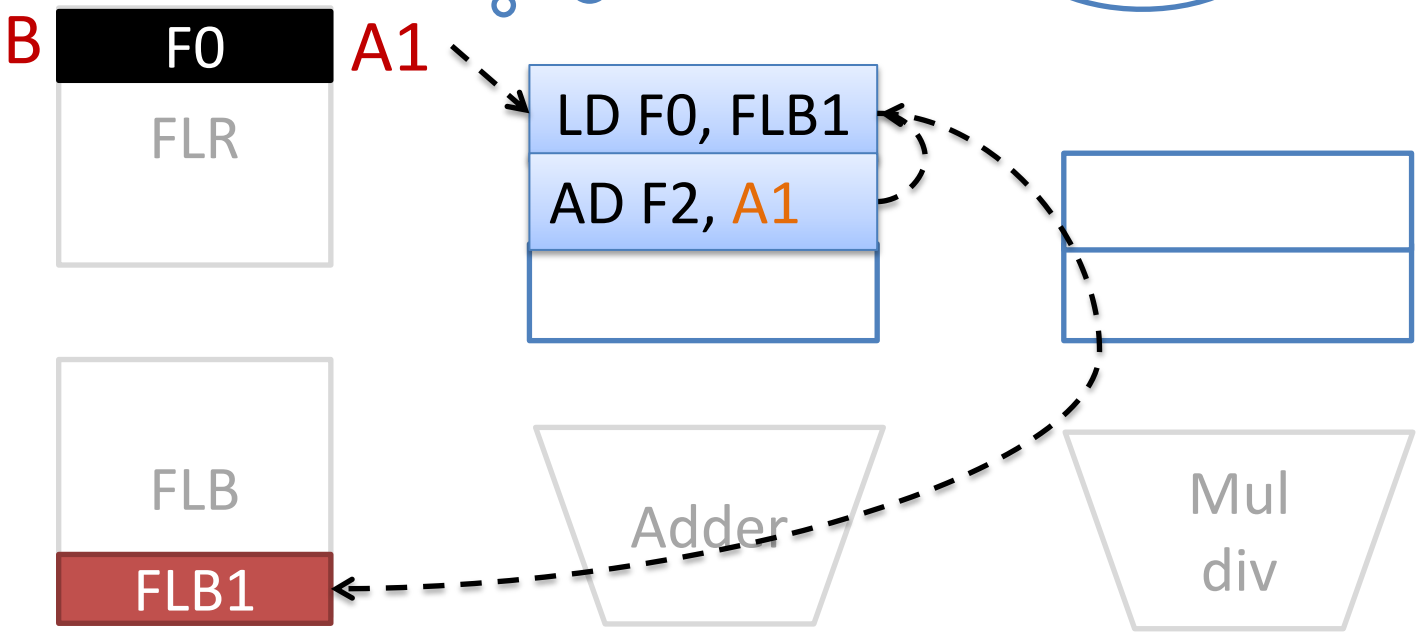


An Example of True Dependence

LD F0, FLB1
AD F2, F0

dispatches to A2

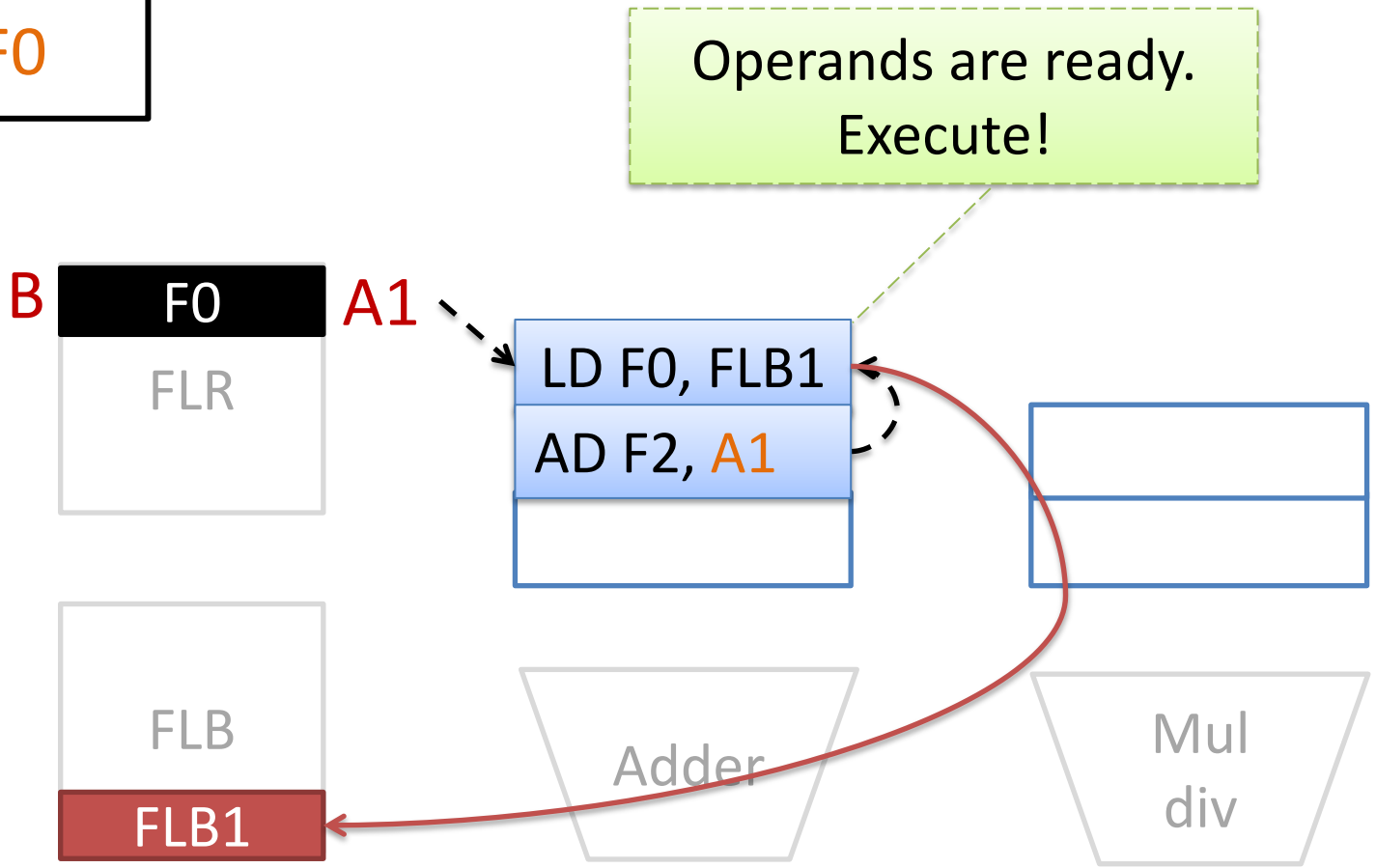
Since A1 is the *producer*, just ask him for it



An Example of True Dependence

LD F0, FLB1
AD F2, F0

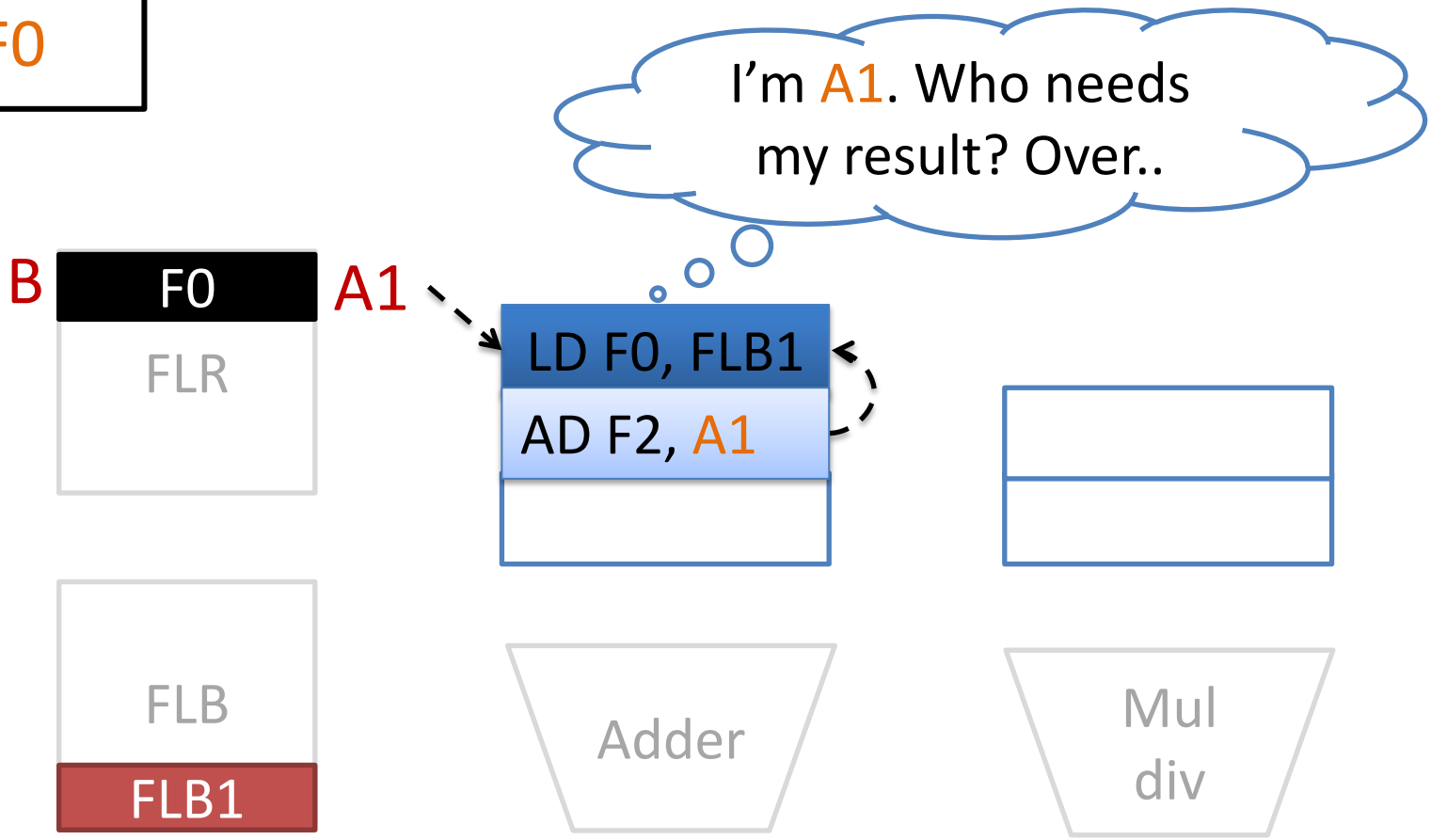
executing



An Example of True Dependence

LD F0, FLB1
AD F2, F0

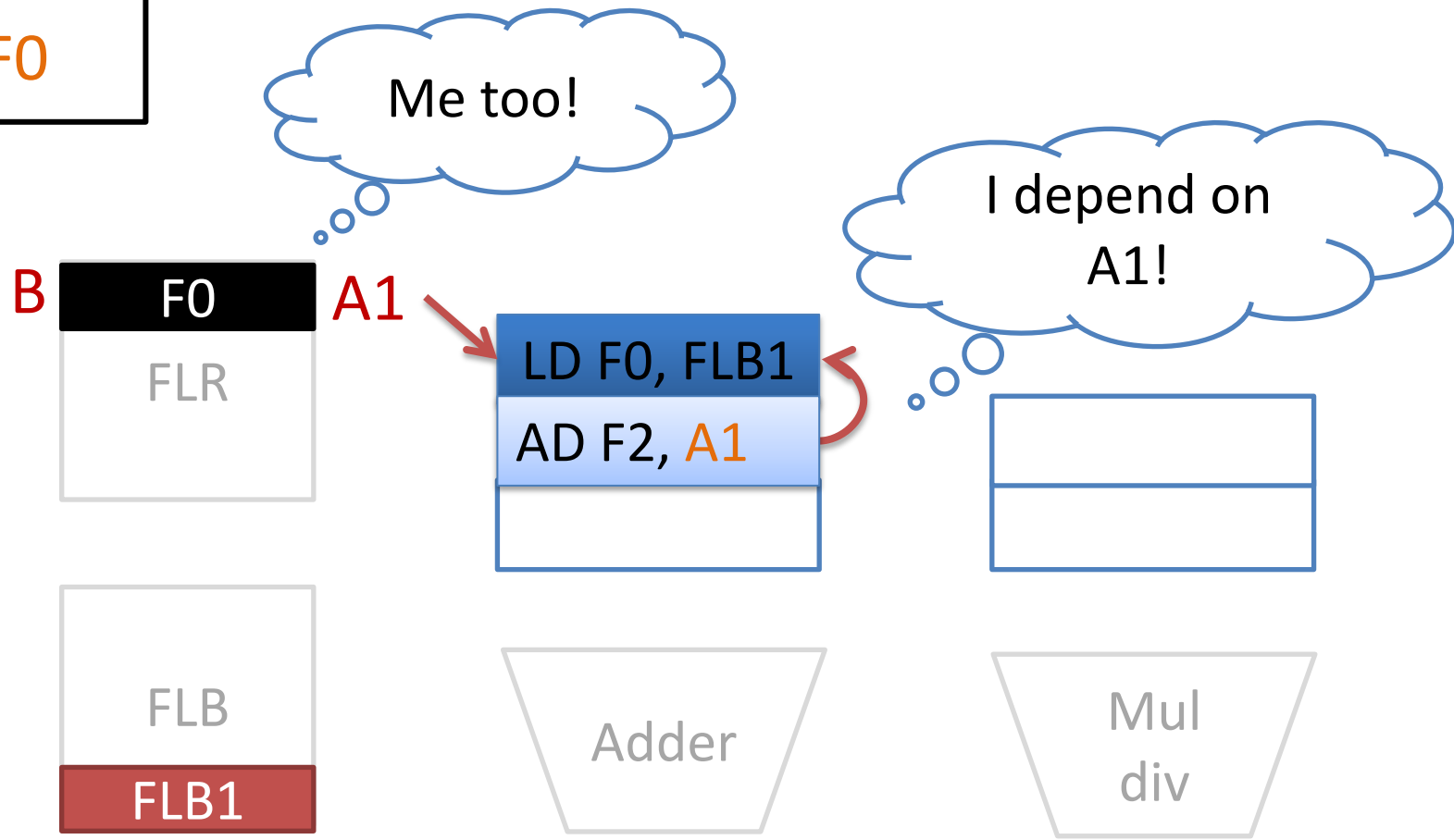
broadcasts it's result to the air



An Example of True Dependence

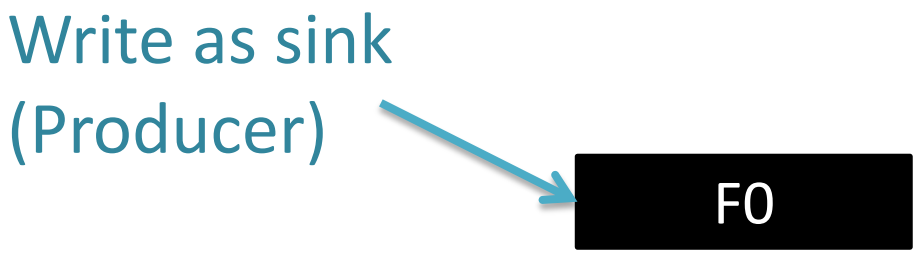
LD F0, FLB1
AD F2, F0

broadcasts it's result to the air



The Role of CDB

- Common Data Bus is in charge of value forwarding
- In reg-to-reg model, a value is passed through a register(write & read)

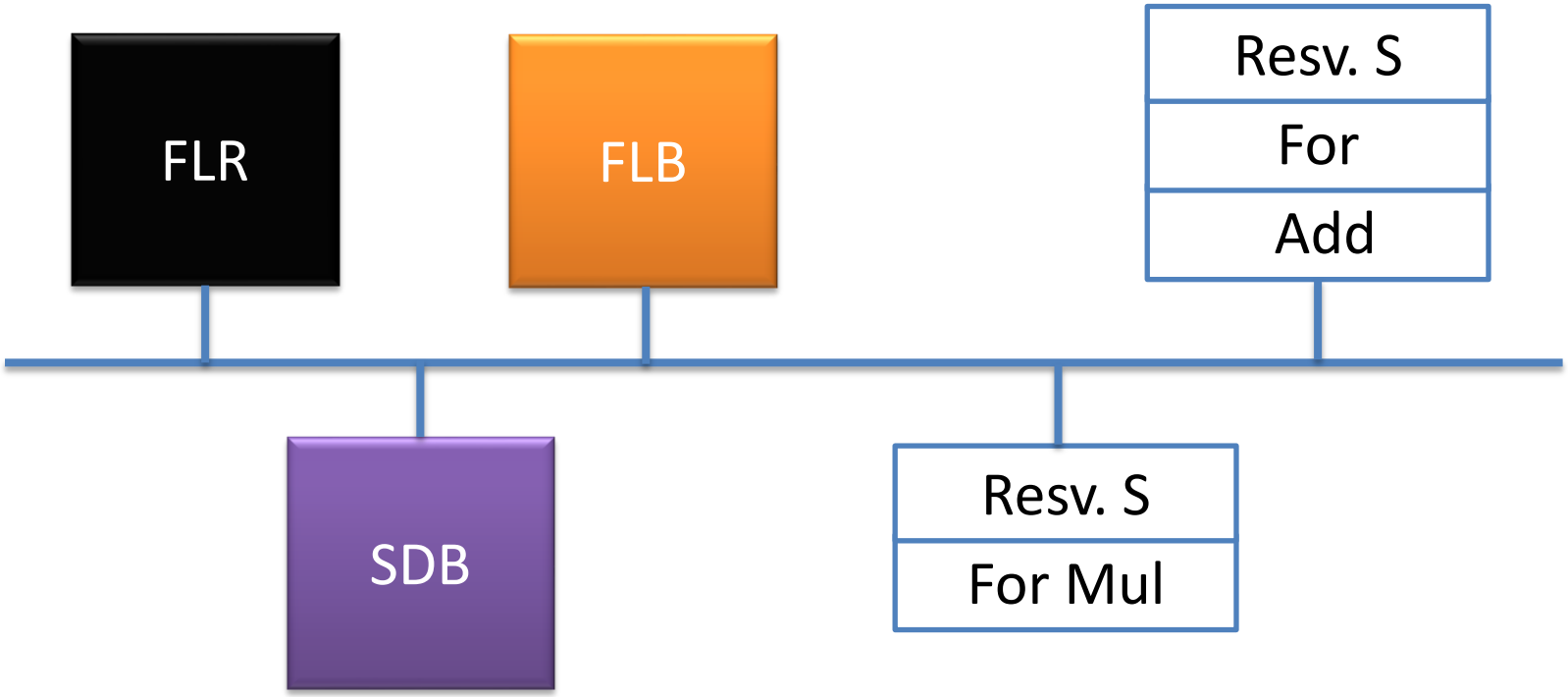


The Role of CDB

- Common Data Bus is in charge of value forwarding
- In reg-to-reg model, a value is passed through a register(write & read)

The Role of CDB

- Load/Store doesn't need to go through ALU
- The dependence management is decoupled from execution as expected



The Role of CDB

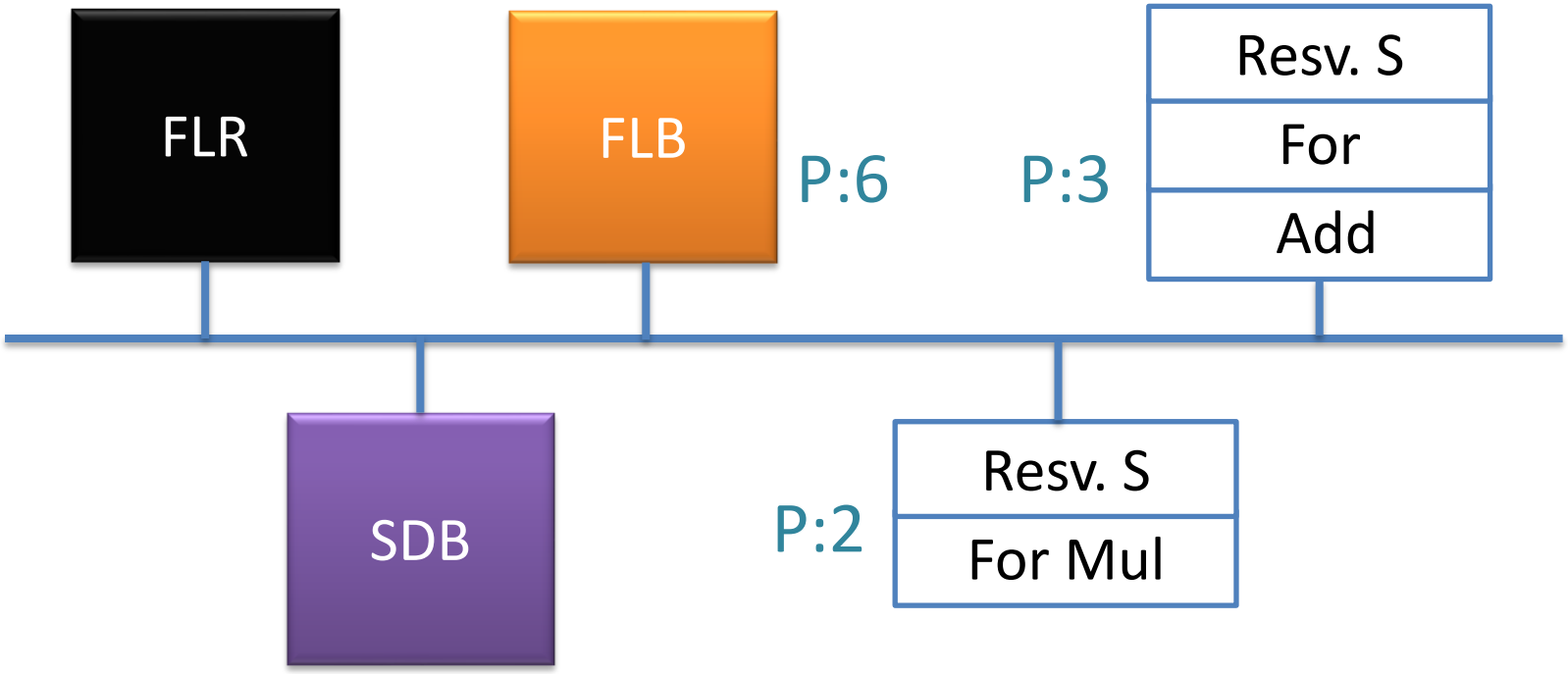
Producer

All units which can alter a register

CDB

Consumer

All units which may take register as an operand



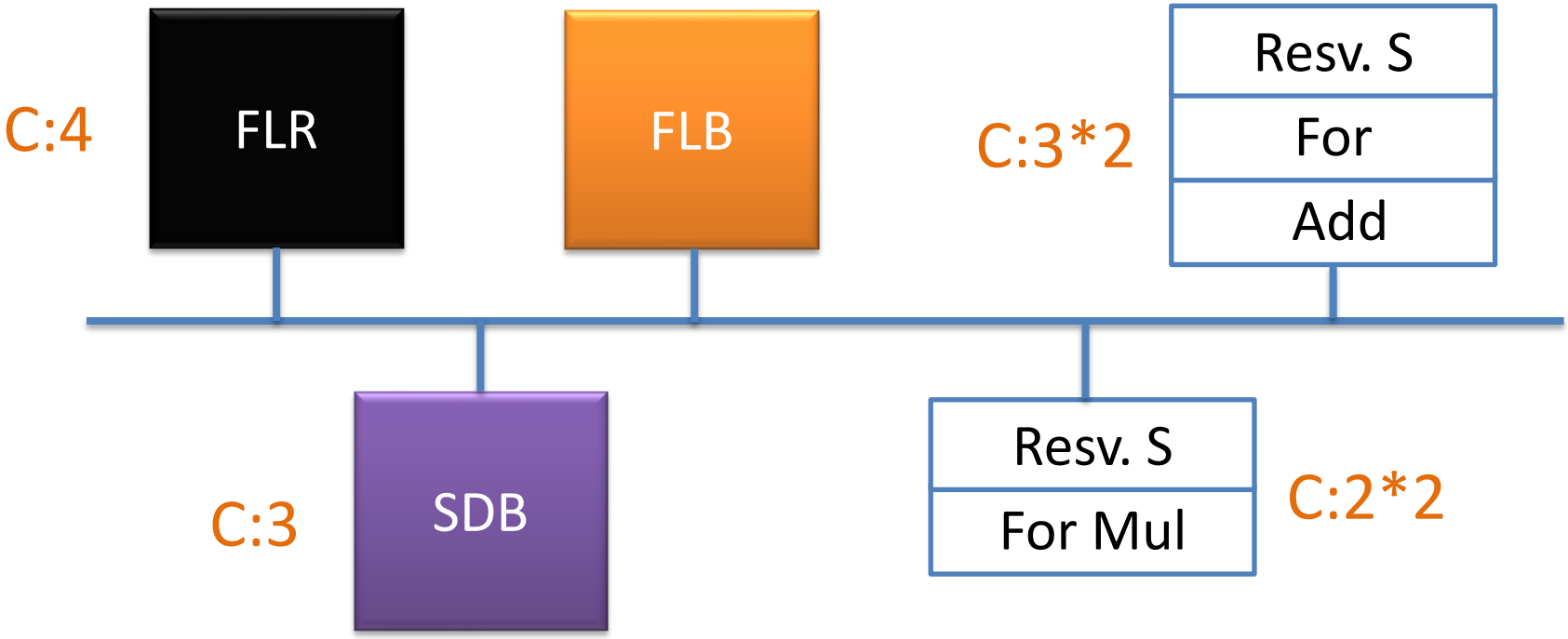
The Role of CDB

Producer

All units which can alter a register

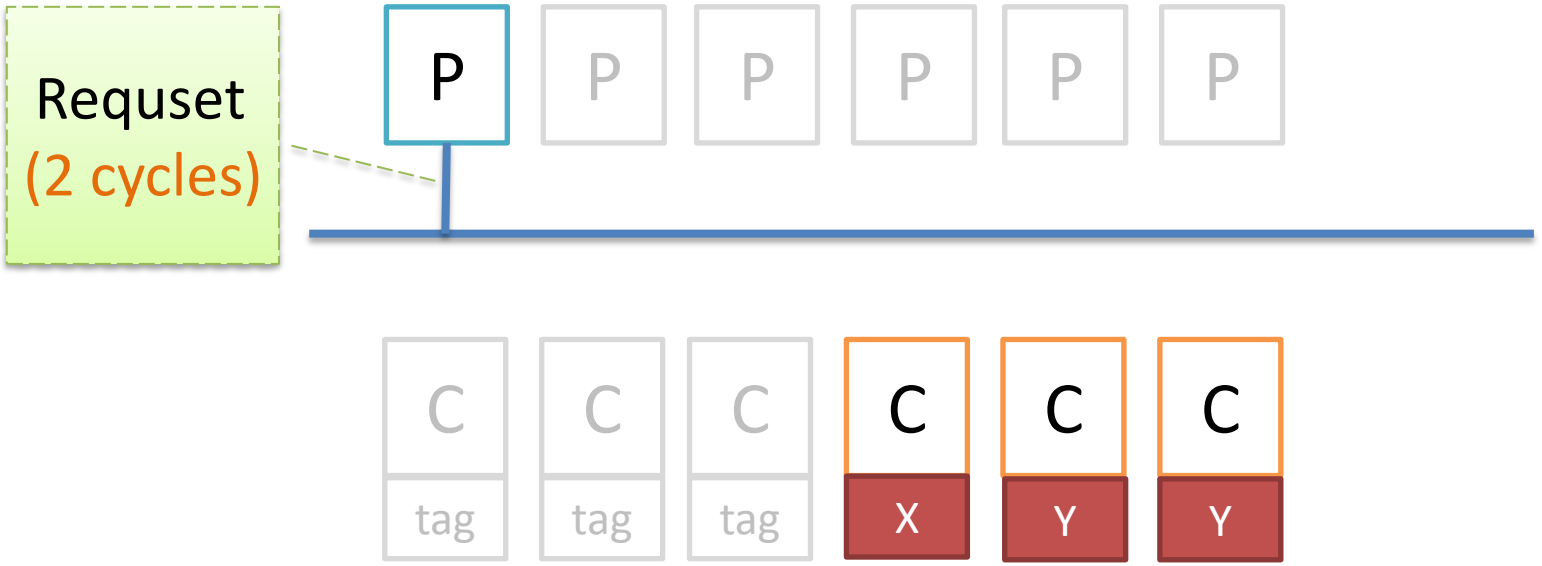
Consumer

All units which may take register as an operand



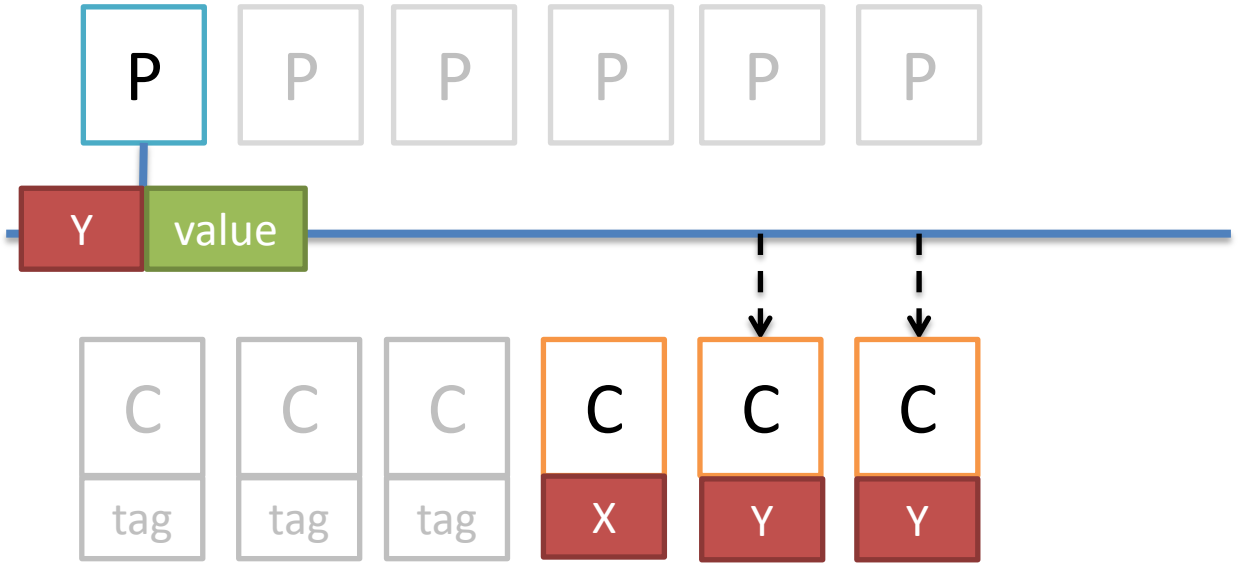
The Implementation of CDB

- A consumer recognizes his producer by tagging
- Producers throw <tag, value> on the bus by turns(make a request first)
- If tag matches , consumer ingates the value



The Implementation of CDB

- A consumer recognizes his producer by tagging
- Producers throw <tag, value> on the bus by turns(make a request first)
- If tag matches , consumer ingates the value

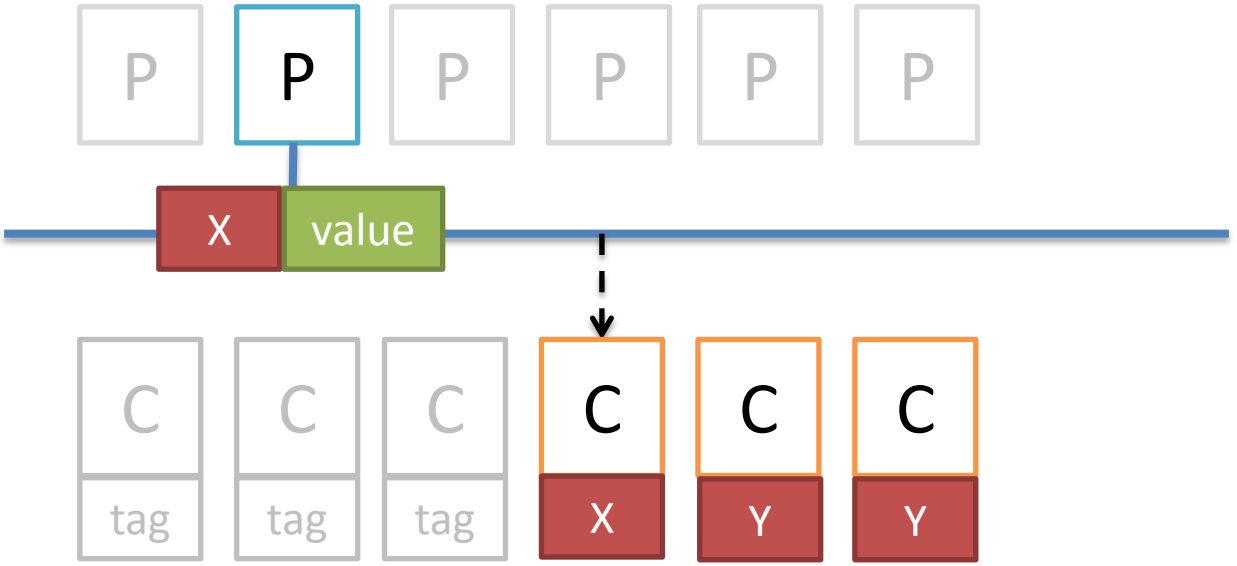


The Implementation of CDB

- A consumer recognizes his producer by tagging
- Producers throw <tag, value> on the bus by turns(make a request first)
- If tag matches , consumer ingates the value

The Implementation of CDB

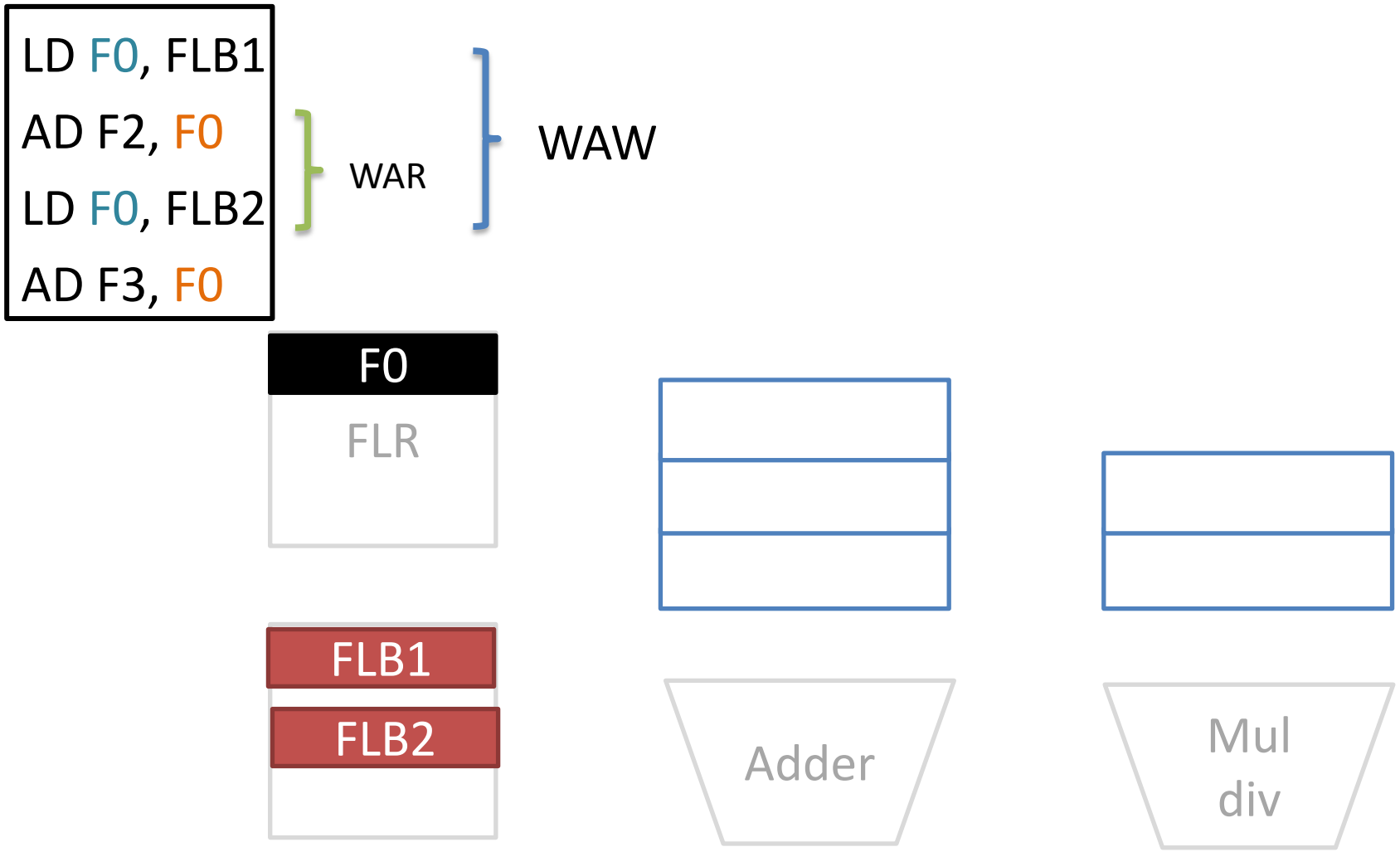
- A consumer recognizes his producer by tagging
- Producers throw <tag, value> on the bus by turns(make a request first)
- If tag matches , consumer ingates the value



The Principle behind the Scene

- Tag is a *pointer* pointing to the producer of the value required by the current instruction
- The pointers construct the dependency information which are hidden by the reg-reg model(*discuss later*)
- With the information, the order of execution can be resolved
- CDB enables ‘producer-consumer’ style data flow

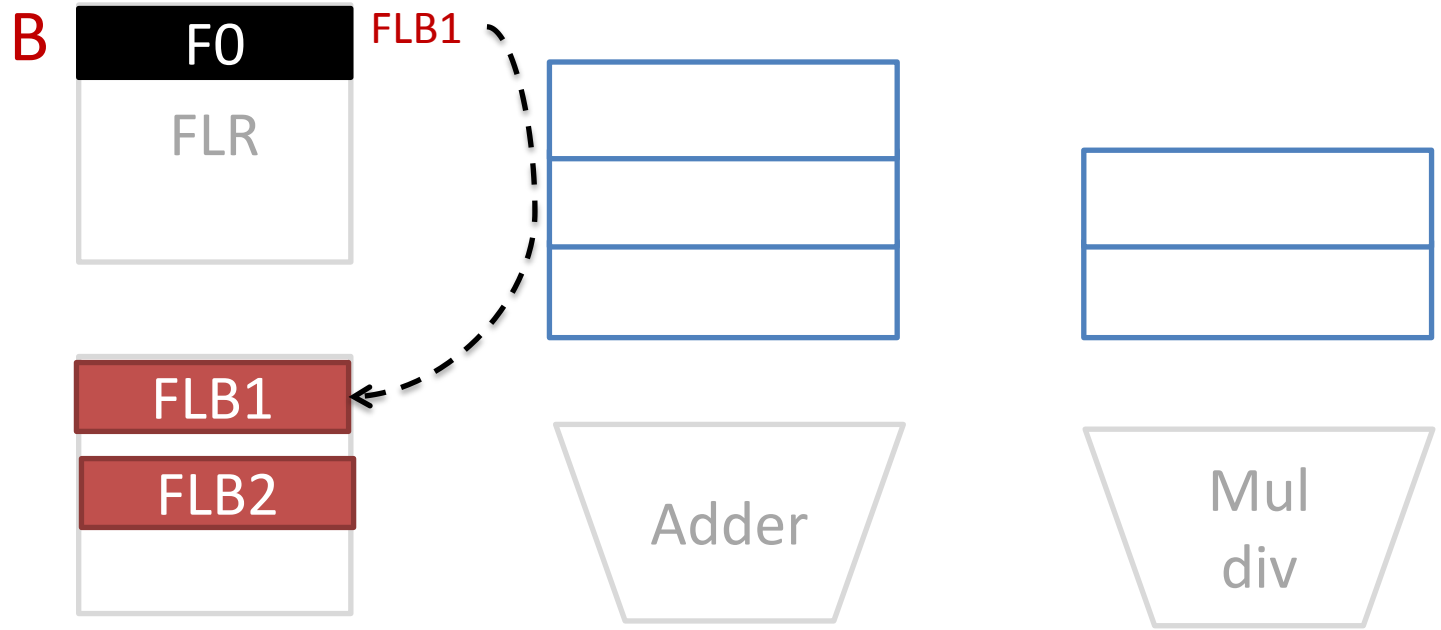
An Example for False Dependence



An Example for False Dependence

```
LD F0, FLB1
AD F2, F0
LD F0, FLB2
AD F3, F0
```

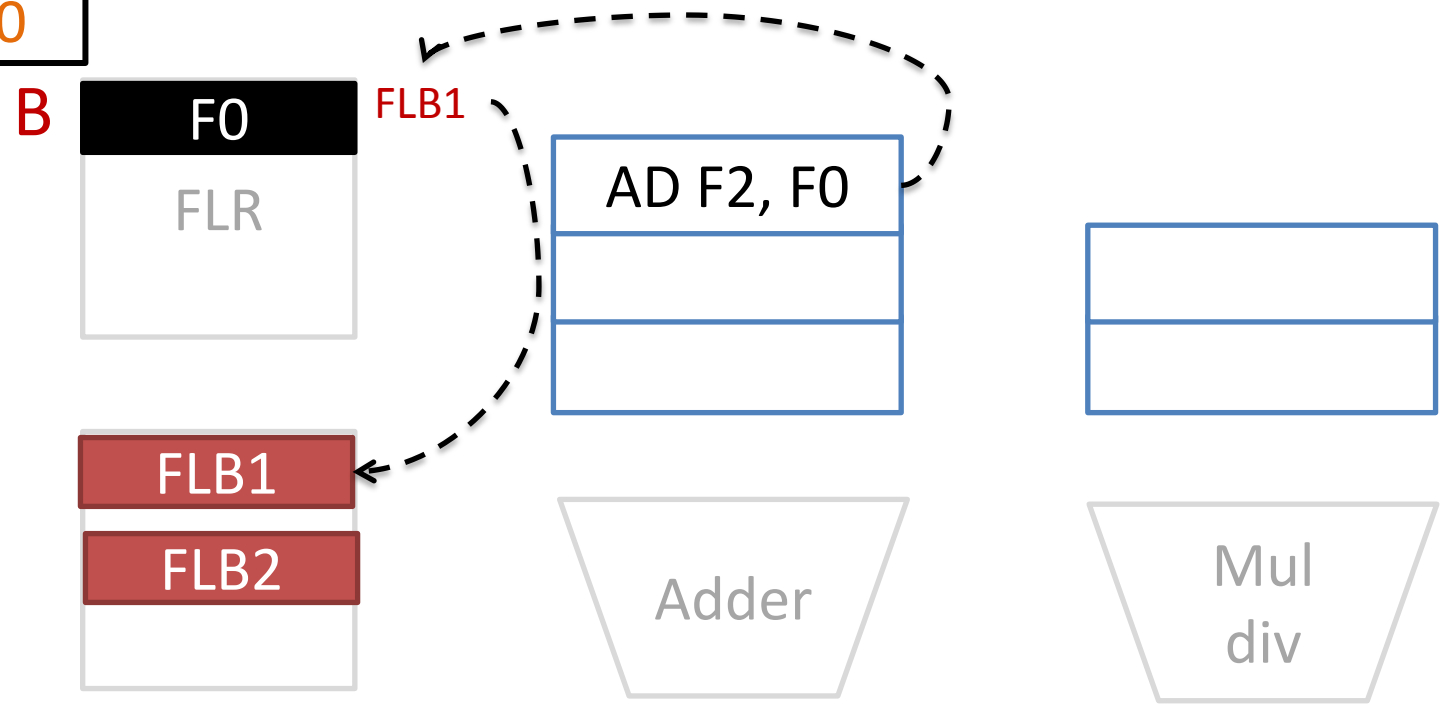
dispatches



An Example for False Dependence

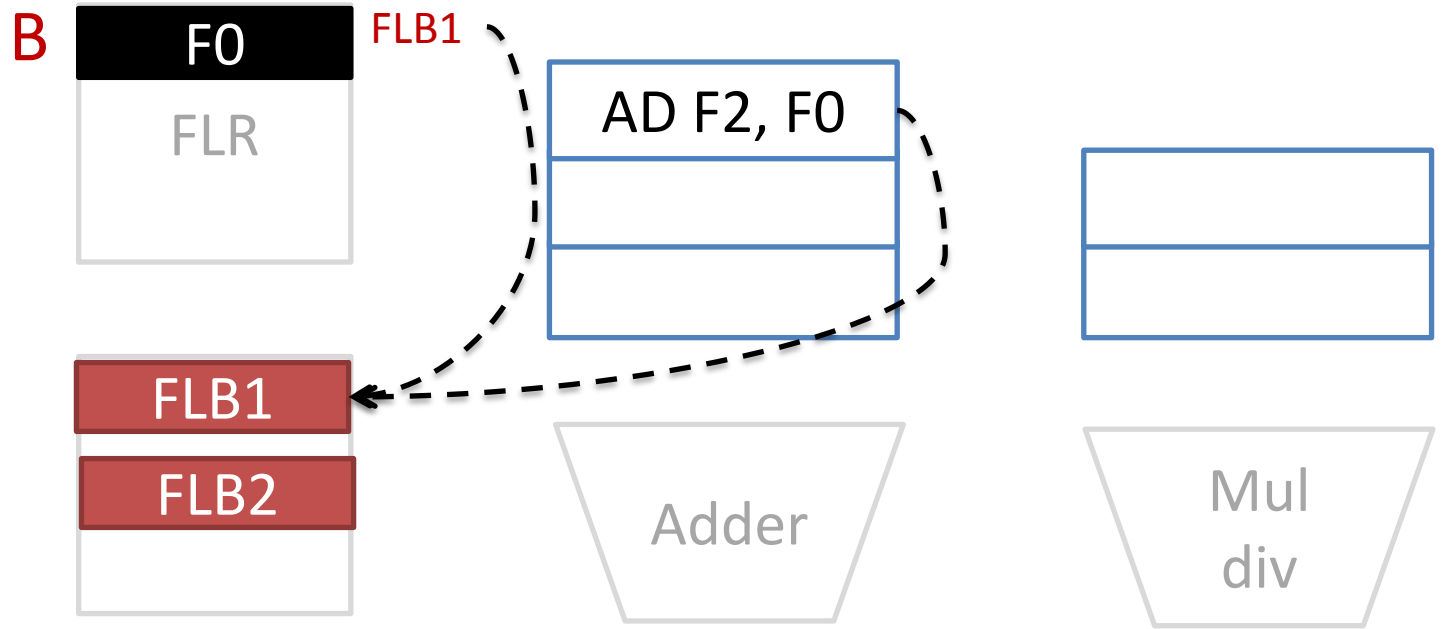
```
LD F0, FLB1
AD F2, F0
LD F0, FLB2
AD F3, F0
```

dispatches to A1



An Example for False Dependence

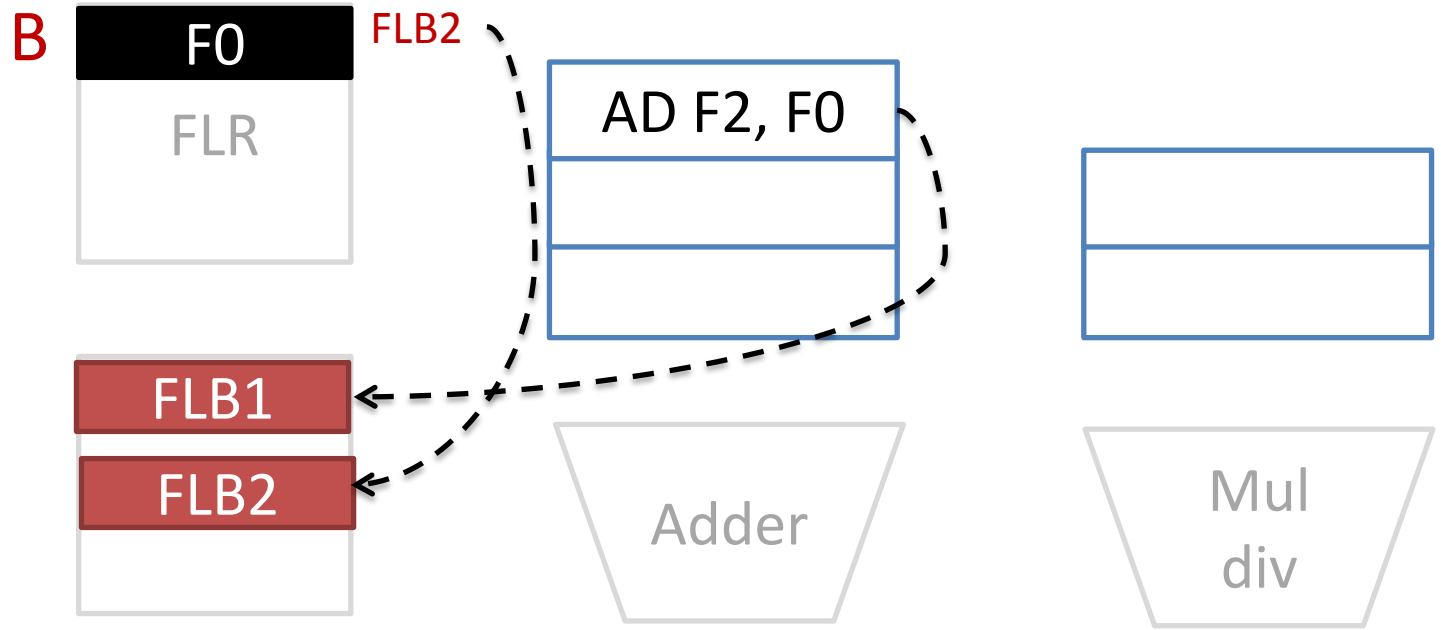
```
LD F0, FLB1
AD F2, F0
LD F0, FLB2
AD F3, F0
```



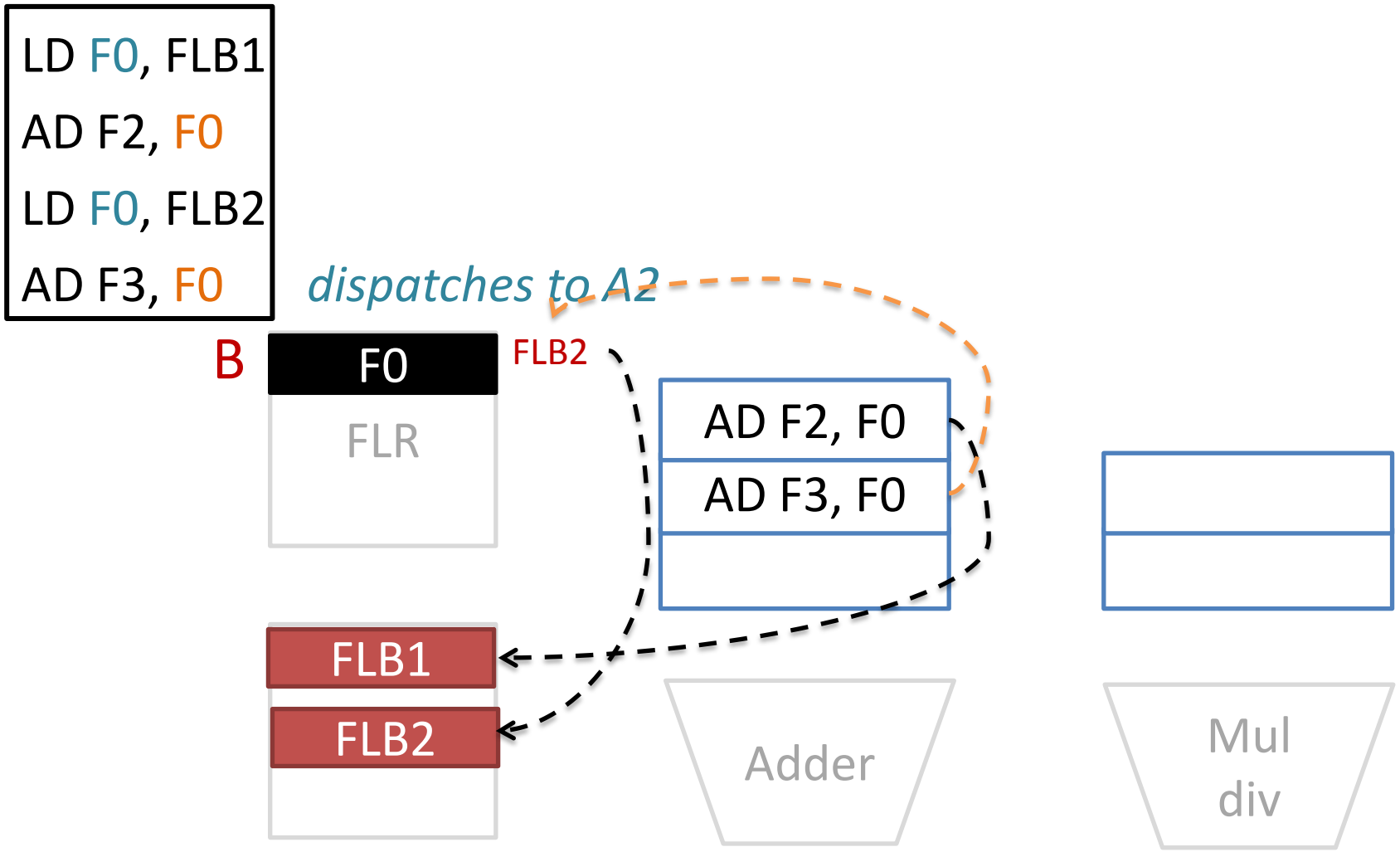
An Example for False Dependence

```
LD F0, FLB1
AD F2, F0
LD F0, FLB2
AD F3, F0
```

dispatches



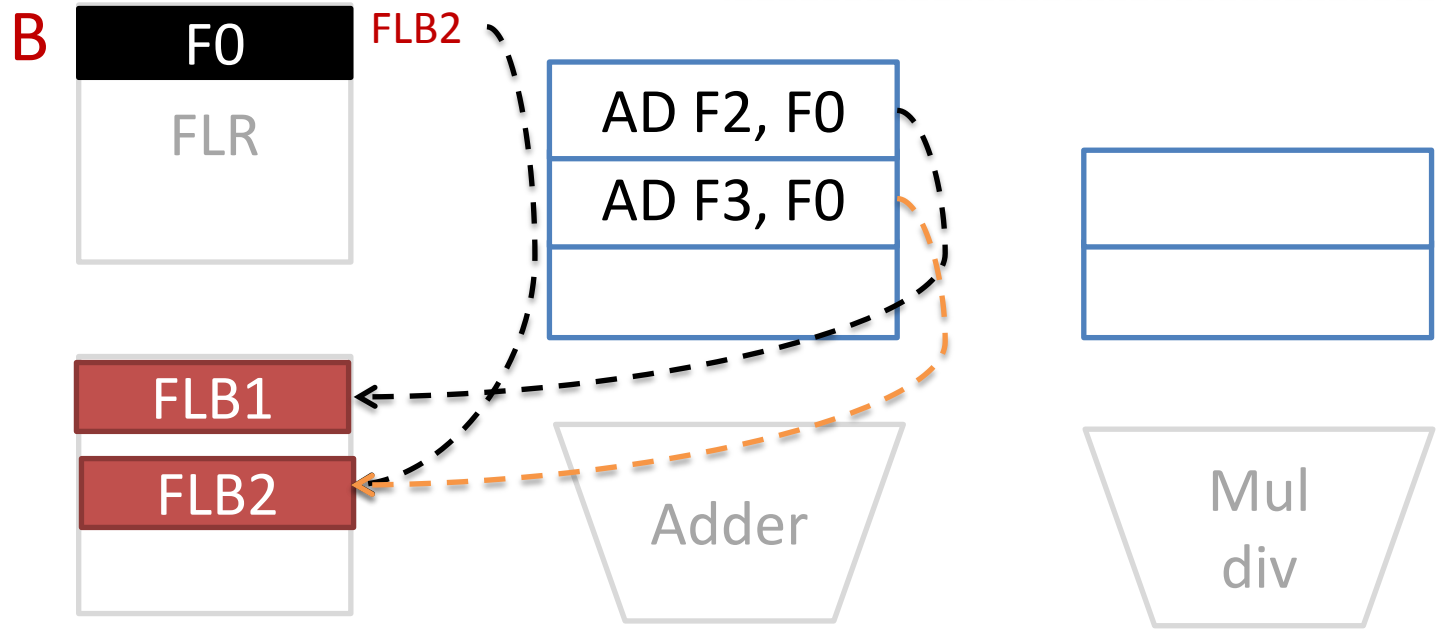
An Example for False Dependence



An Example for False Dependence

```
LD F0, FLB1
AD F2, F0
LD F0, FLB2
AD F3, F0
```

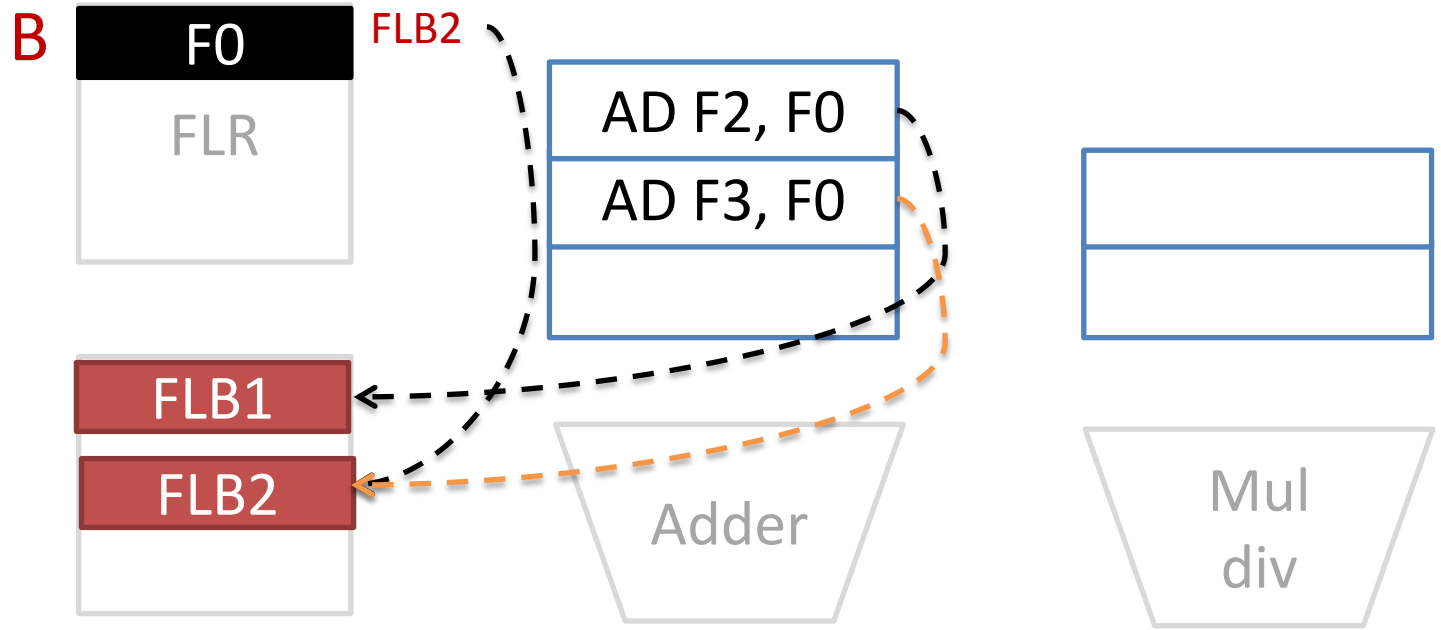
Keep tracing the source of the value instead of the register holding it



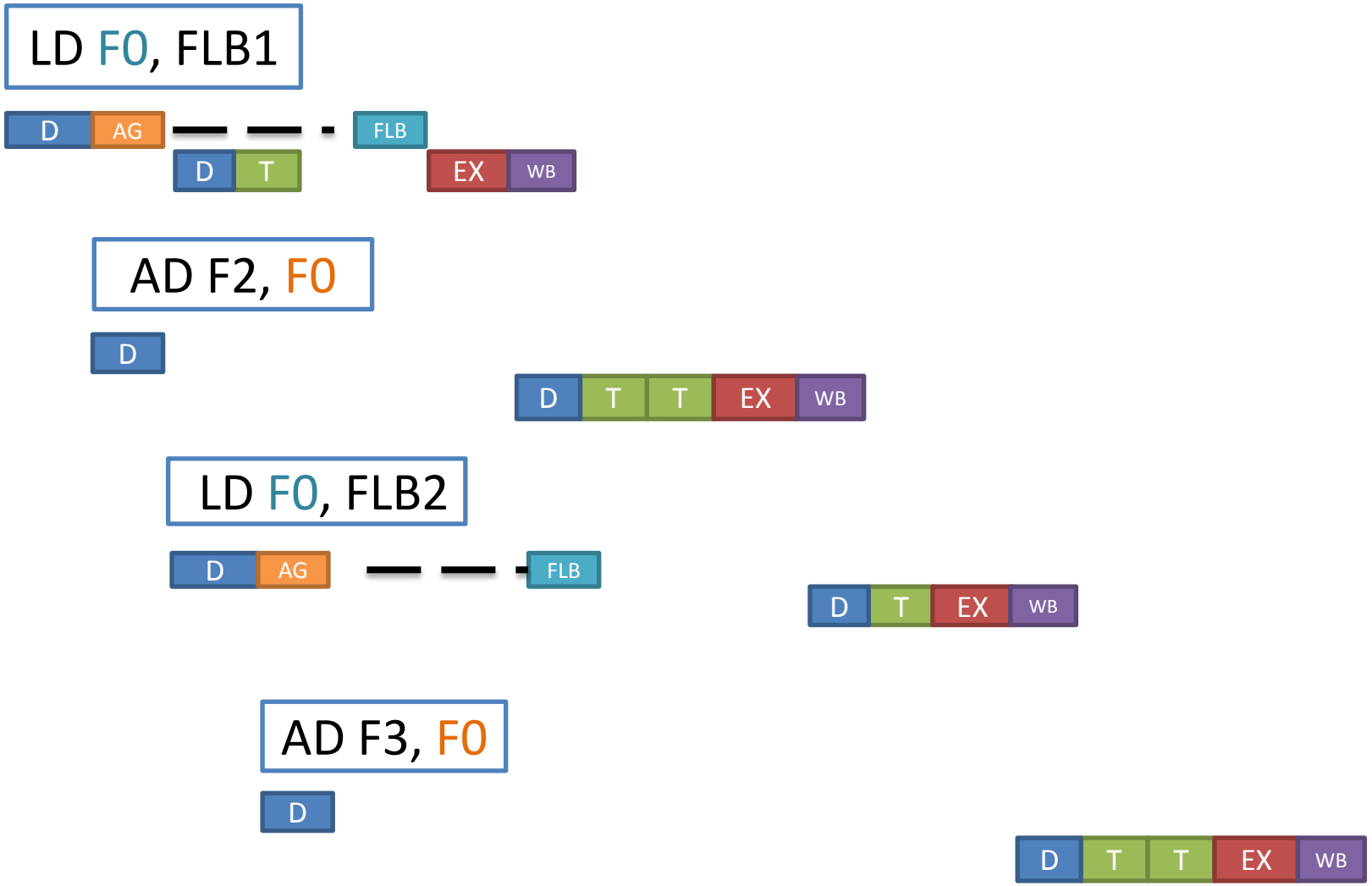
An Example for False Dependence

```
LD F0, FLB1
AD F2, F0
LD F0, FLB2
AD F3, F0
```

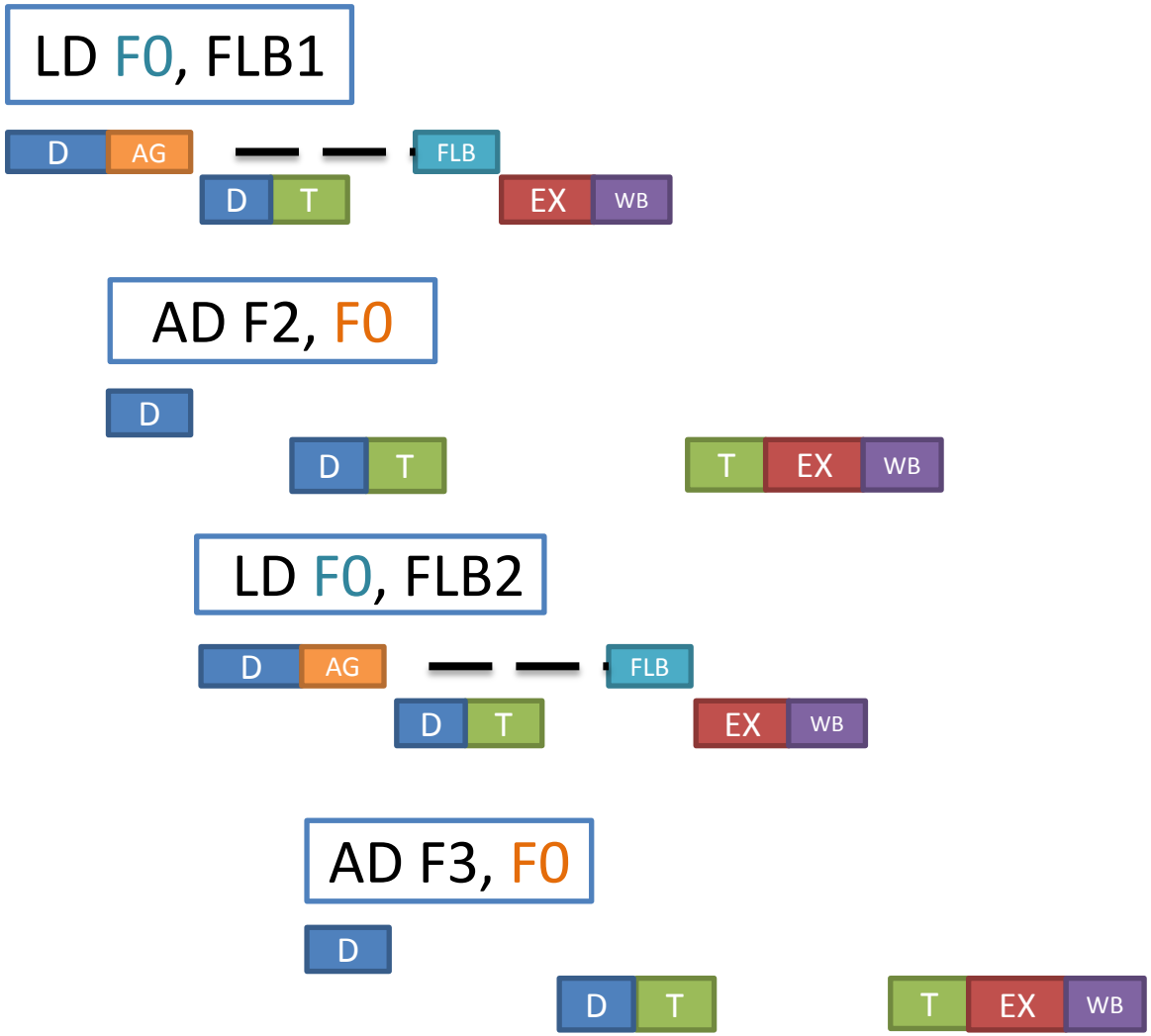
There's no need to rename a register(Naming is just a way of referring values)



Timing Sequence with Busy Bit

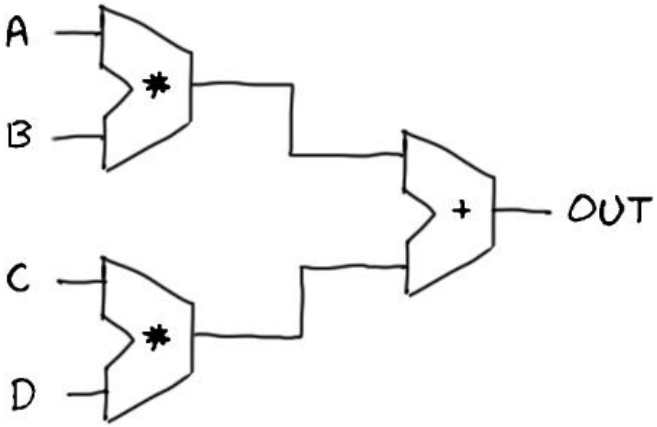


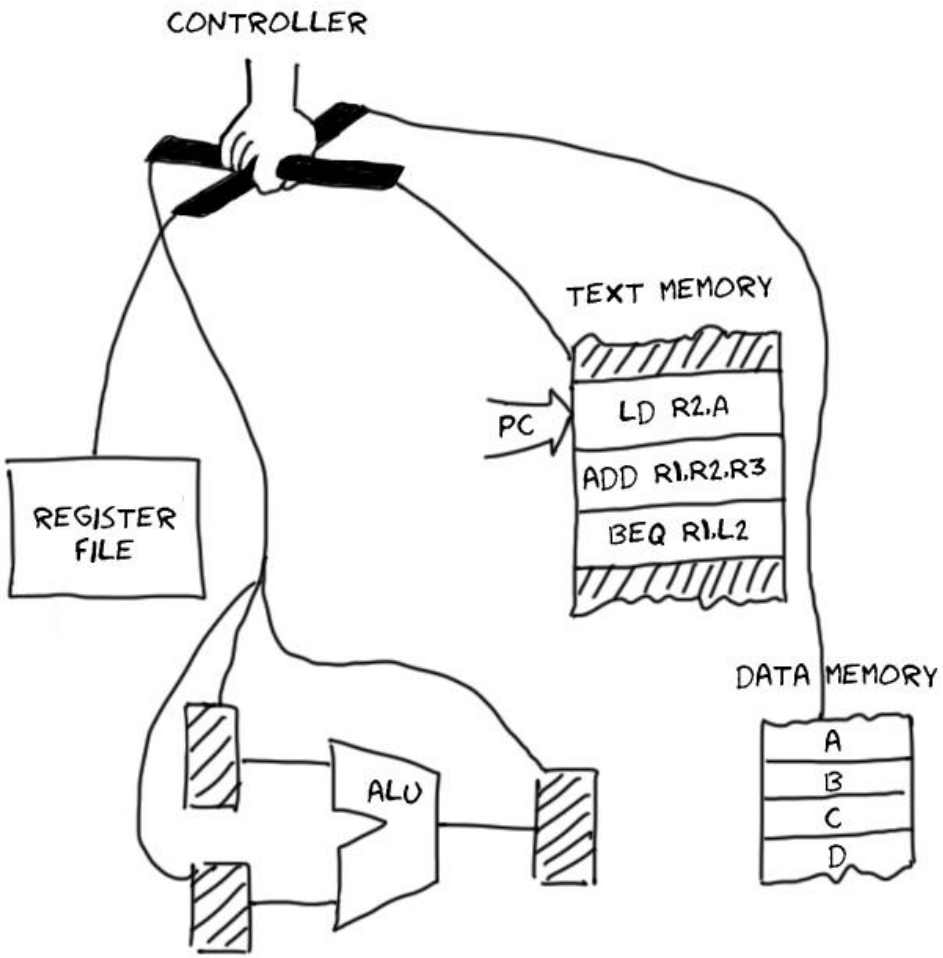
Timing Sequence with Reservation Station



The Side Effect of Register Machine

- What are the differences between a circuit and a register machine?





The Side Effect of Register Machine

- What are the differences between a circuit and a register machine?

Circuit

- Special purpose
- Data-driven
- Exposed dependence

Register Machine

- General purpose
- Control-driven
- Implict dependence via registers

…But registers are rare

Conclusion

- Tomasulo algorithm has nothing to do with register renaming
- It resolves the WAR & WAW by elimating the side effect of using register to pass value
- By using Tomasulo algorithm, the execution of a program is driven by data flow thus exploiting maximum concurrency