Tutorial: a Practical Introduction to using Event-B for Complex Hardware and Embedded System Specification and Design

John Colley

FDL 2012

Vienna

Introduction

- Background to Event-B
- Event-B in the Design/Verification Flow
- Complex hardware specification/verification
  - Pipelines
  - Elastic Buffering
- Embedded system specification/verification
  - Temporal Modeling in Cyber-physical systems
  - Animating and Model Checking Event-B models
- Assertion-based verification
  - Deriving assertions from the specification
- Summary

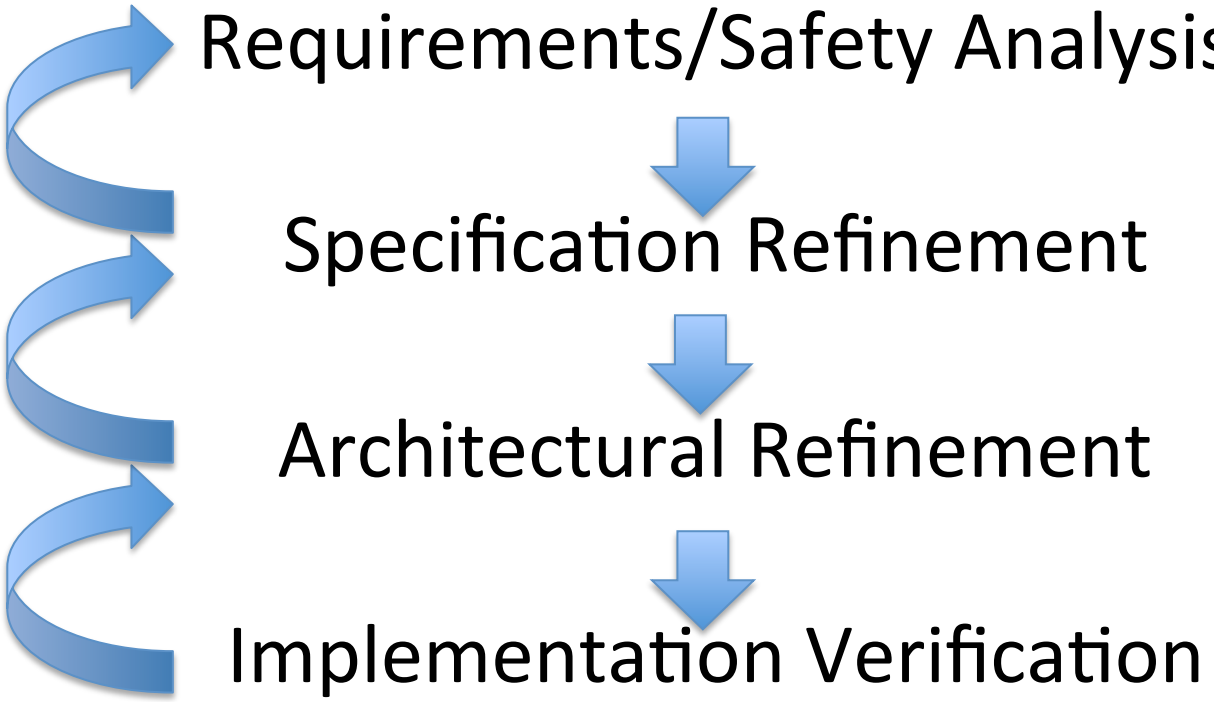
Background to Event-B

- **Event-B** is a formal method for system-level modelling and analysis which uses
  - set theory as a modelling notation
  - refinement to represent systems at different abstraction levels
  - mathematical proof to verify consistency between refinement levels.
- The **Rodin Platform** is an Eclipse-based IDE for Event-B
  - provides support for refinement and mathematical proof
  - open source
- **ProB** is an animator and model checker in the Rodin environment

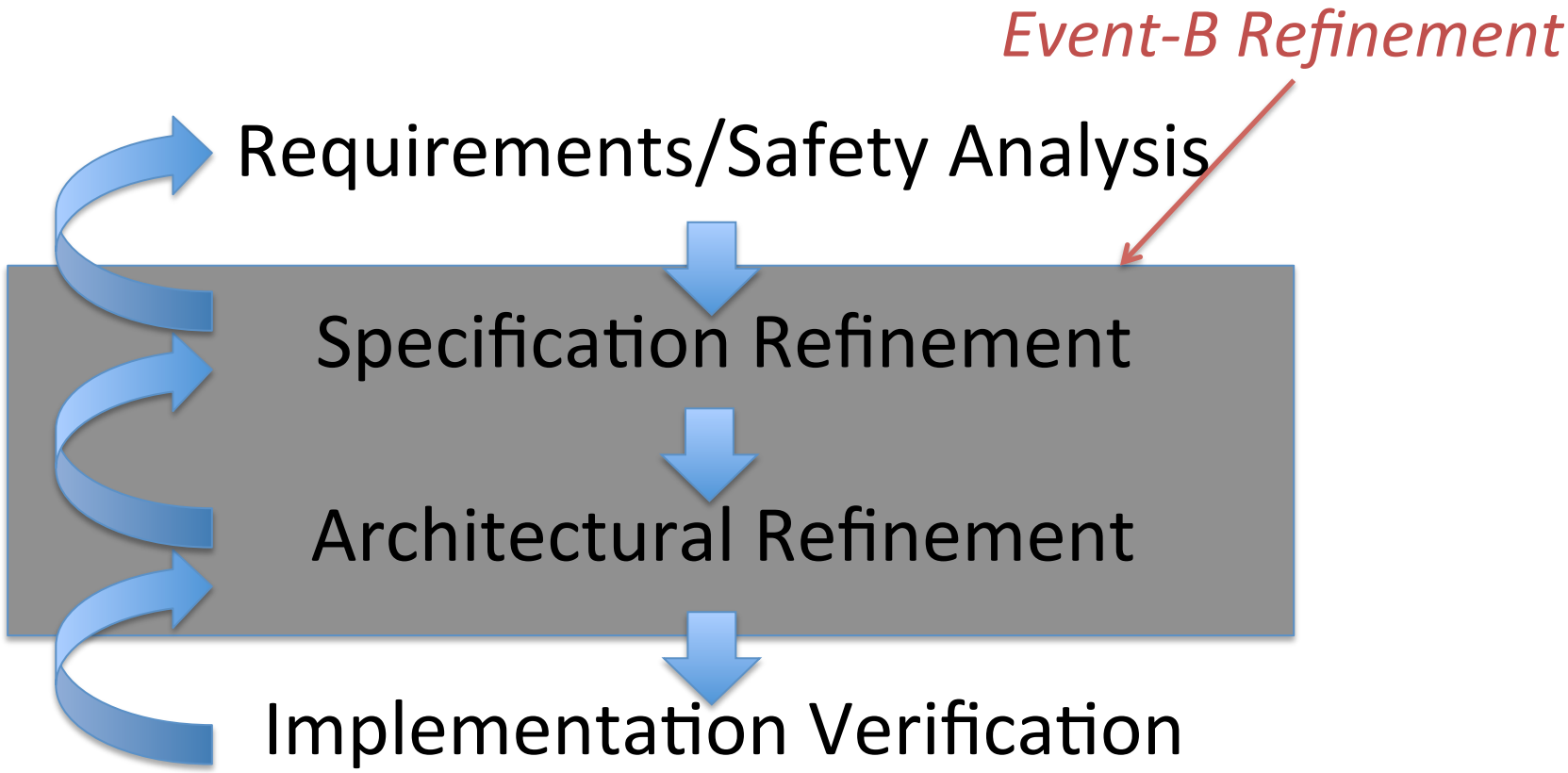
Industrial Deployment of Event-B

- ***Deploy*** (FP7 – completed 2012)
  - **Bosch** have been working on developing a cruise control system and a start-stop system
  - **Siemens Transportation** have been working on train control and signalling systems
  - **Space Systems Finland** have been working on part of the BepiColombo space probe and on Attitude and Orbit Control System software(AOCS)
  - **SAP** have been working on analysis of business choreography models
  - **Systerel** are working on railway and aerospace systems
- ***ADVANCE*** (FP7 – started Oct 2011)
  - Event-B for Cyber-Physical Systems
  - http://www.advance-ict.eu/

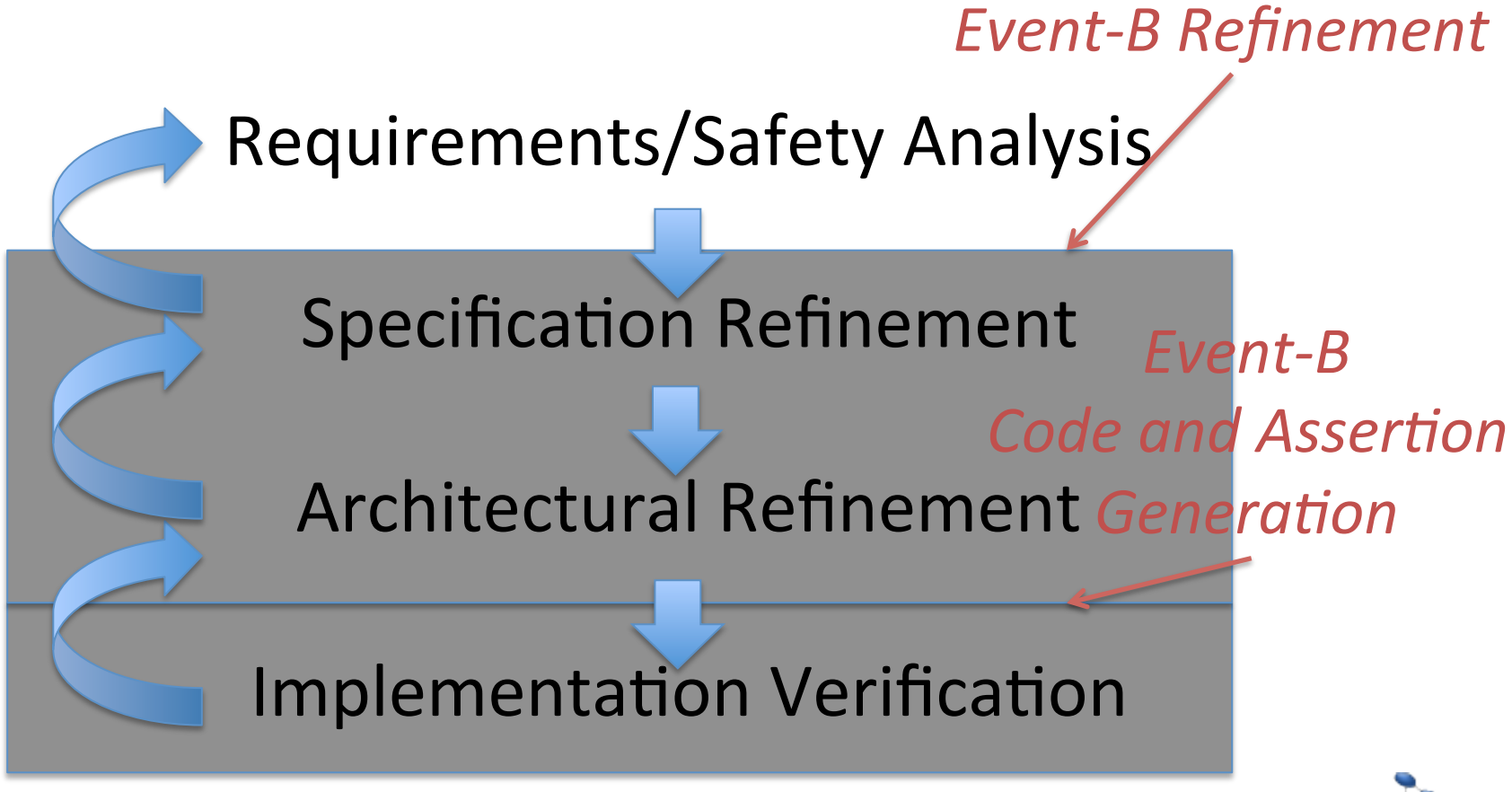
Event-B in a Design /Verification Flow



Event-B in a Design /Verification Flow

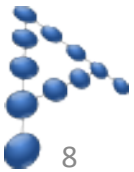


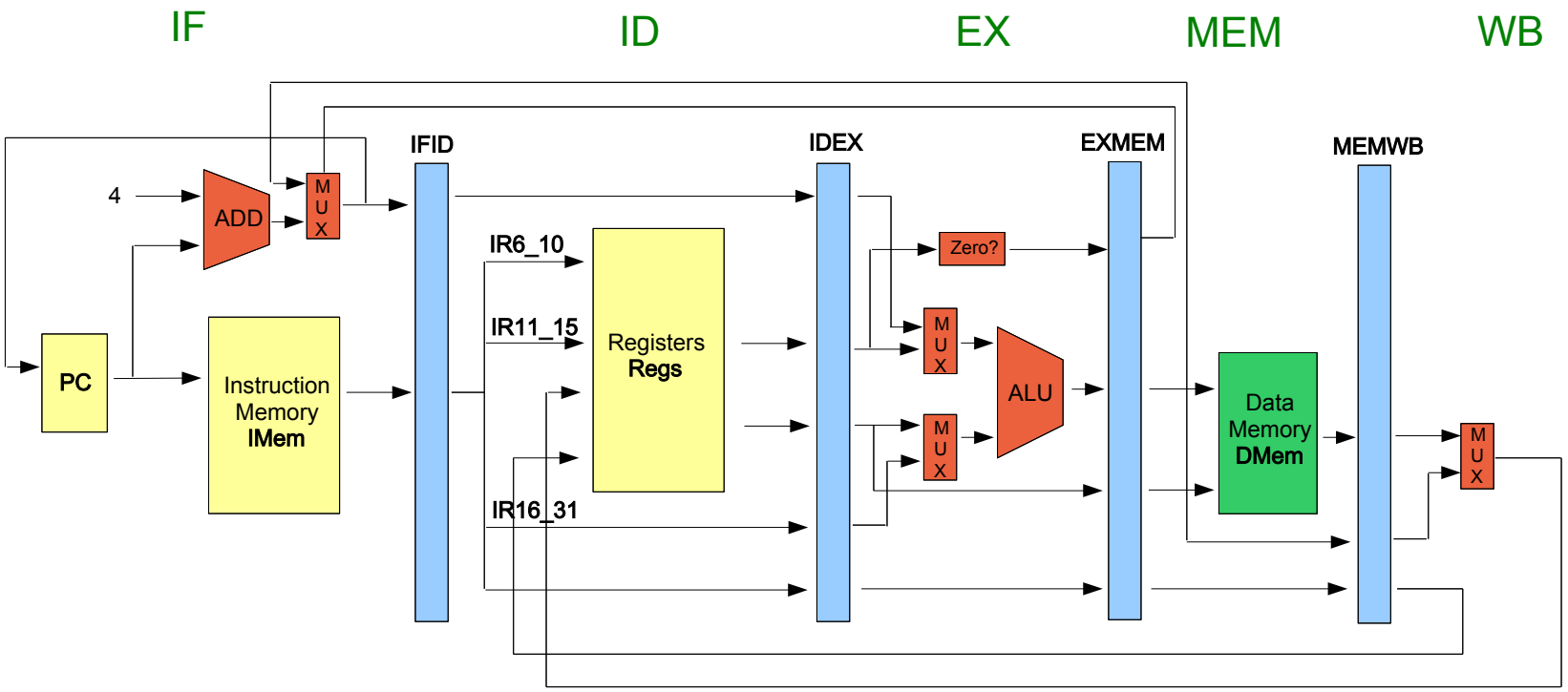
Event-B in a Design /Verification Flow



A MicroProcessor Pipeline

- Each pipeline stage is a process running concurrently with all the other stages
- Communication is by shared variables (pipeline registers)
- New high-level languages speed up design
  - Bluespec
    - Guarded atomic actions
    - High-level synthesis to RTL
- But verification is still
  - performed on low-level RTL description
  - predominantly test-based





Generic Operations

Load
Store
Branch
ArithRR
ArithImm

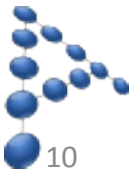
Pipeline Stages

Instruction Fetch (IF)
Instruction Decode (ID)
Execute (EX)
Memory Access (MEM)
Writeback (WB)

- pipeline register

Pipeline Verification Goals

- Start Verification at the Specification Level
- Explore micro-architectural alternatives at the specification level
- Close the gap between specification and implementation
- Exploit synergy with Bluespec
- Incorporate proof-based techniques into the established SoC verification flow

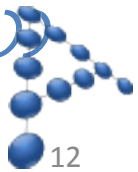


Specifying an Arithmetic Instruction

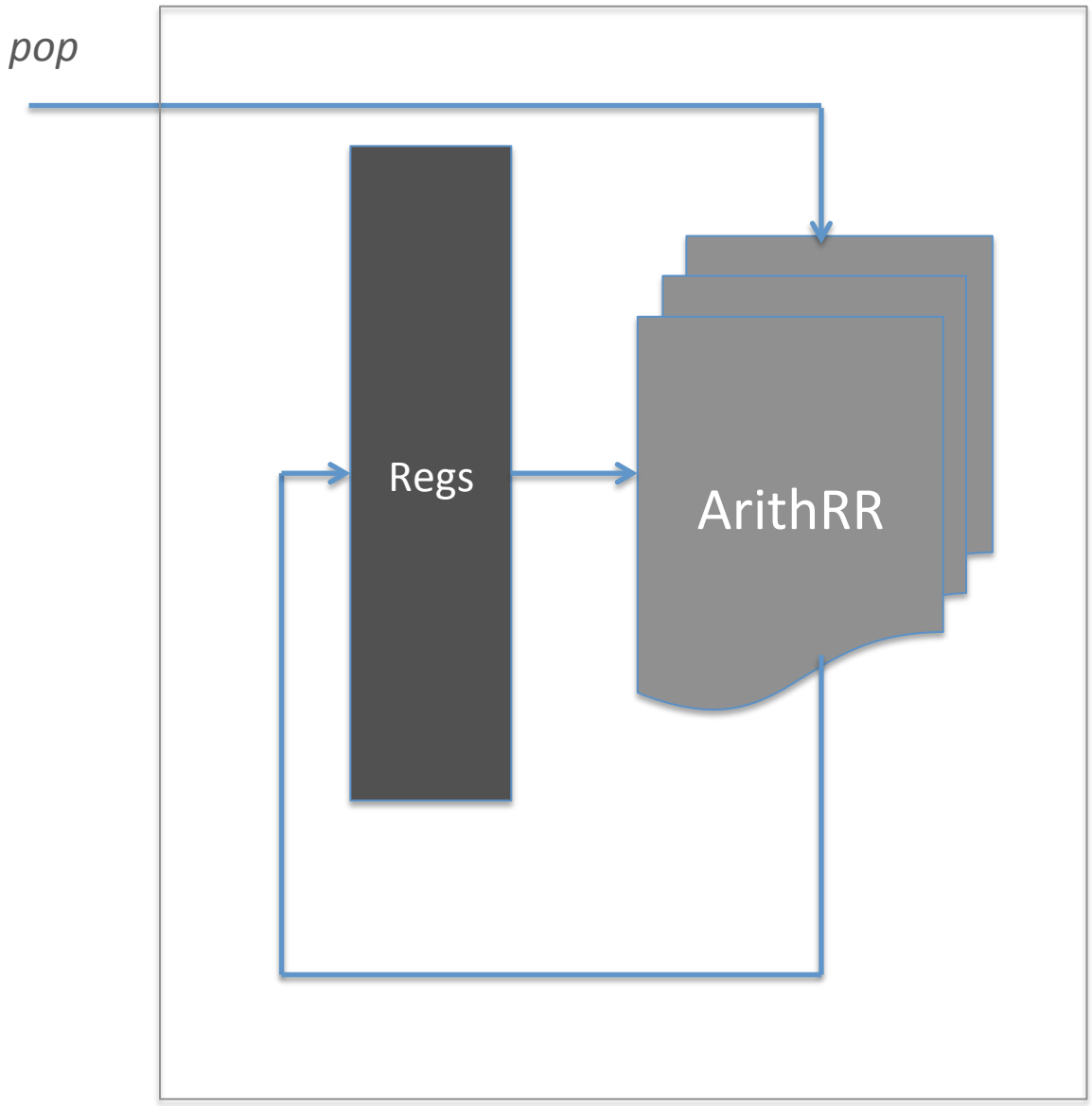
```
context PIPEC

constants Register Rr Ra Rb ArithRROp

sets Op // Operations


axioms
  @axm1 Register ⊆  ℕ // Processor Register Identifier
  @axm2 Rr ∈ Op → Register // Destination Register
  @axm3 Ra ∈ Op → Register // First Source Register
  @axm4 Rb ∈ Op → Register // Second Source Register
  @axm5 ArithRROp ⊆ Op // Register/Register Arithmetic Ops
end
```

The Abstract Machine

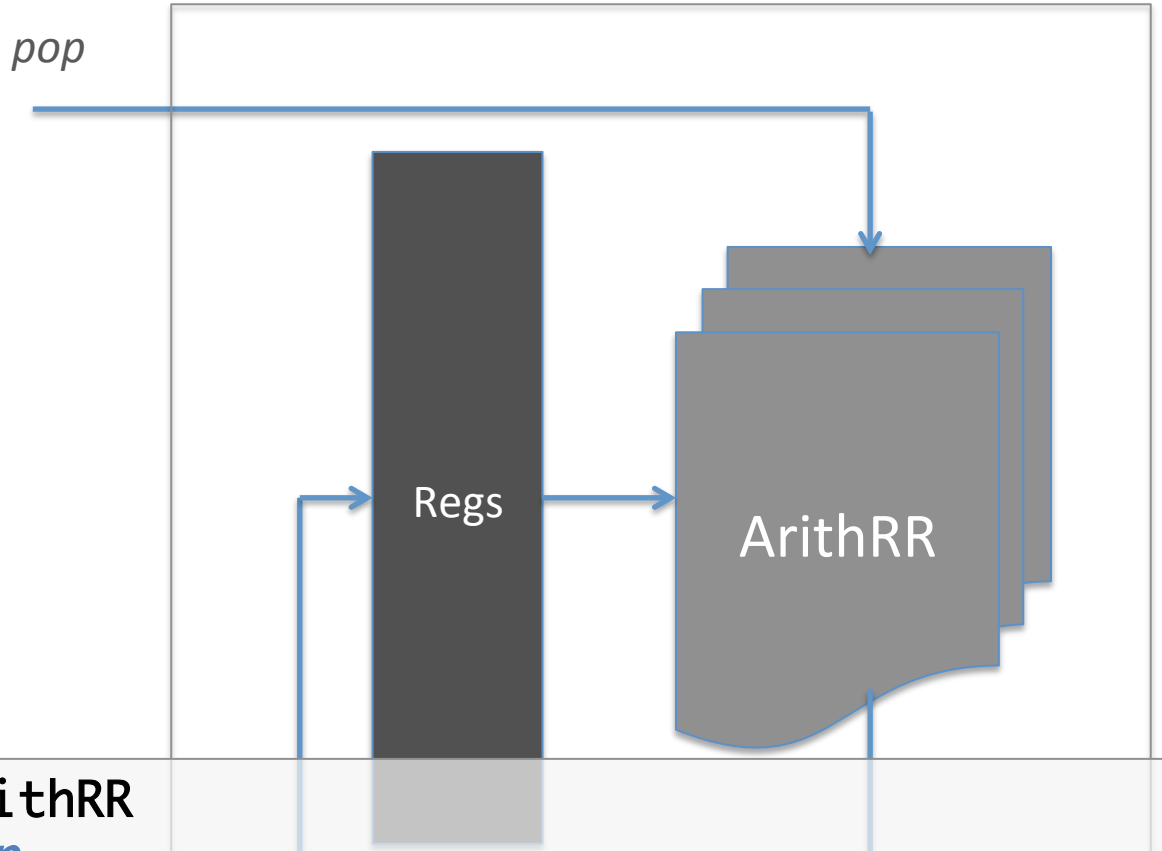
```
machine PIPEM sees PIPEC

variables Regs

invariants
  @inv1 Regs ∈ Register → ℤ // The Processor Register File

events
  event INITIALISATION
    then
      @act1 Regs ≔ Register × {0}
  end

  event ArithRR
    any pop
    where
      @grd1 pop ∈ ArithRROp
    then
      @act1 Regs(Rr(pop)) ≔ Regs(Ra(pop)) + Regs(Rb(pop))
  end
end
```



The Abstract Architecture

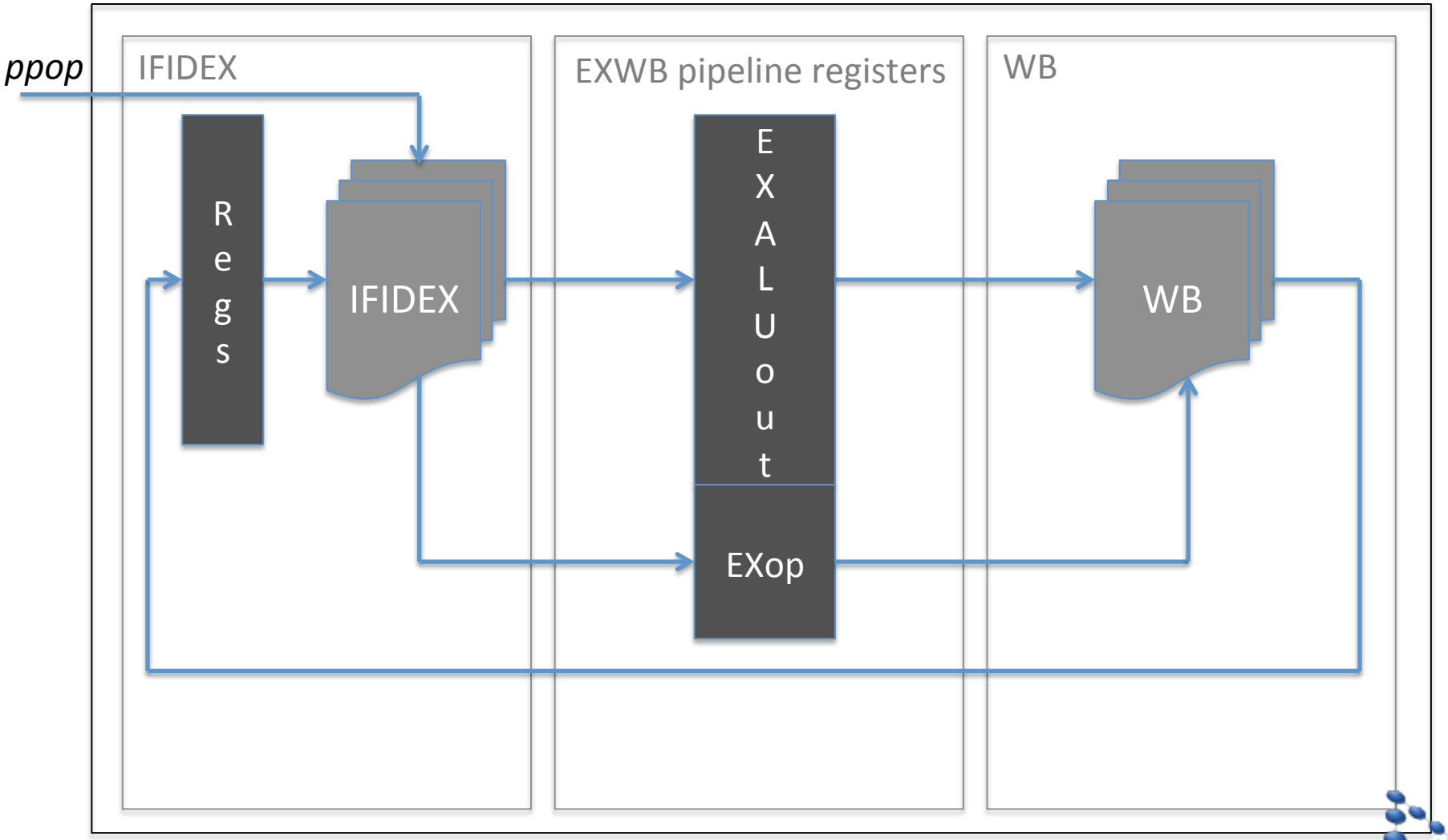


The Abstract Architecture

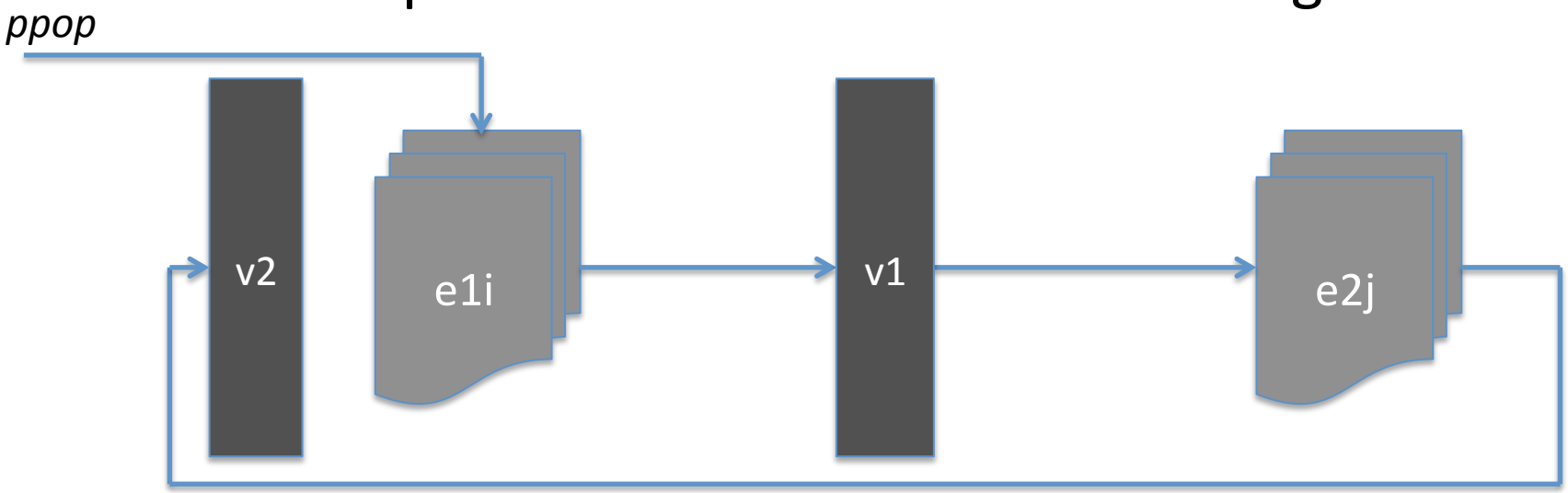



```
event ArithRR
  any pop
  where
    @grd1 pop ∈ ArithRROp
  then
    @act1 Regs(Rr(pop)) ≔ Regs(Ra(pop)) + Regs(Rb(pop))
end
```

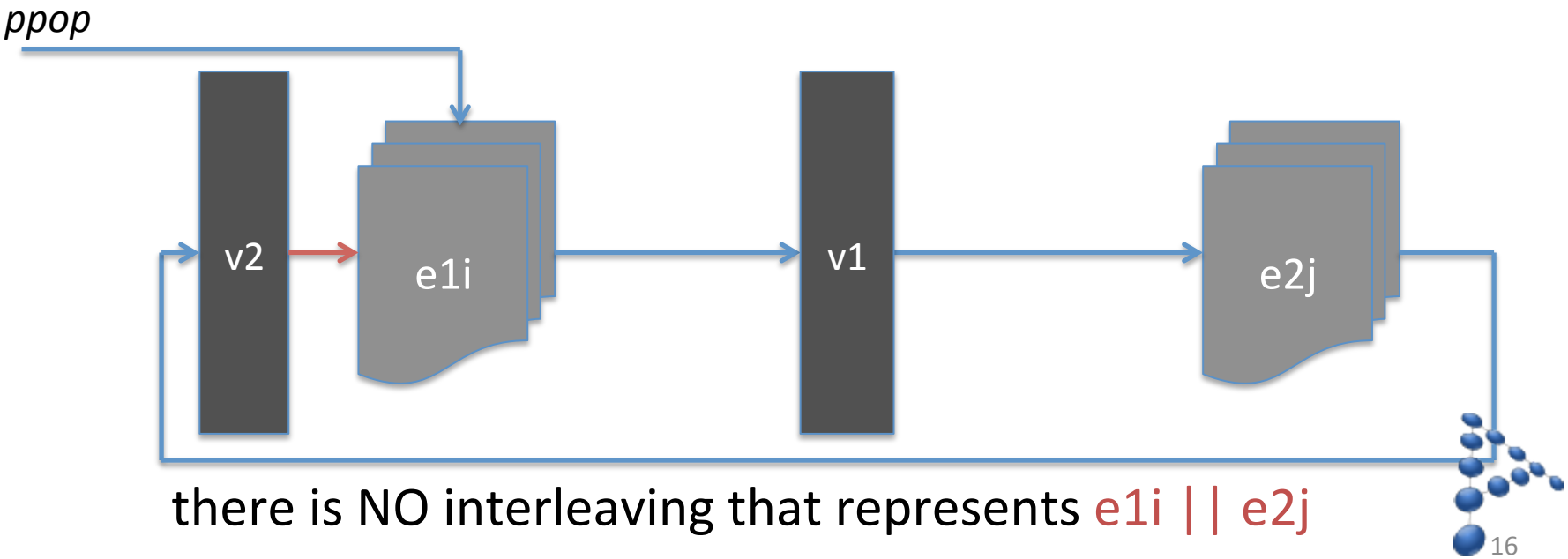
First Refinement: a two-stage pipeline



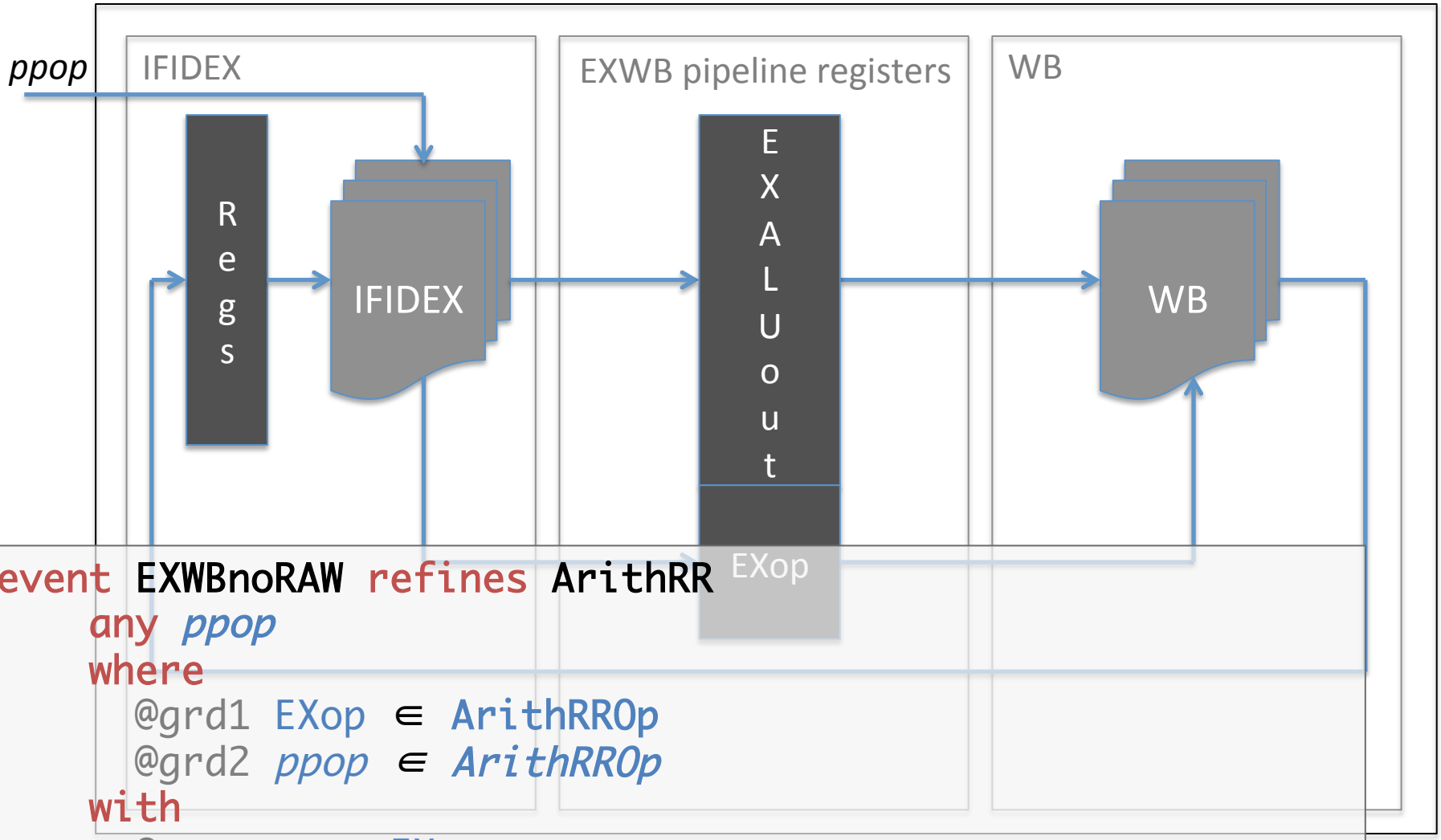
Pipeline Feedback and Interleaving



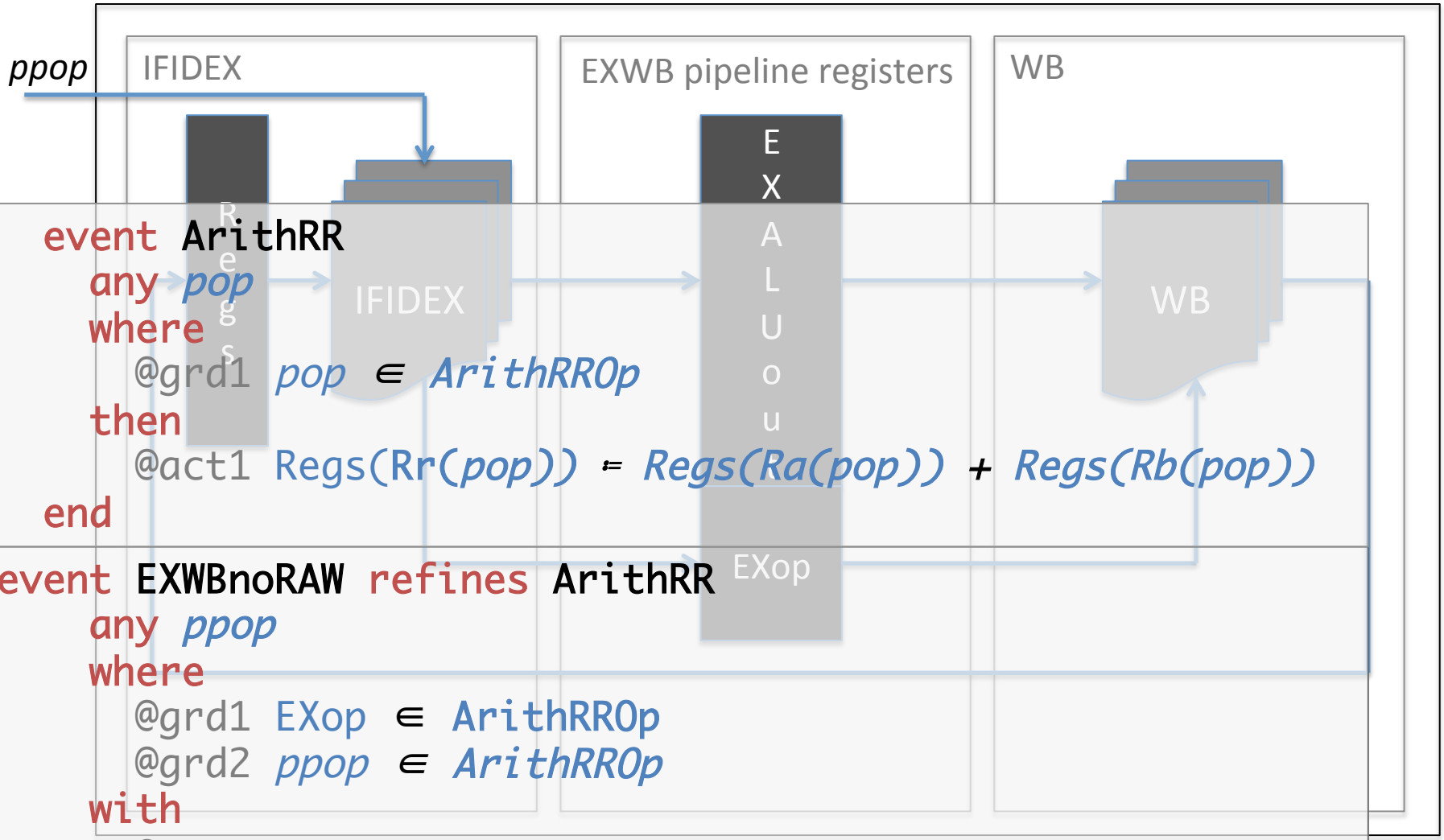
e2j followed by e1i (e2j;e1i) is equivalent to e1i || e2j



there is NO interleaving that represents e1i || e2j



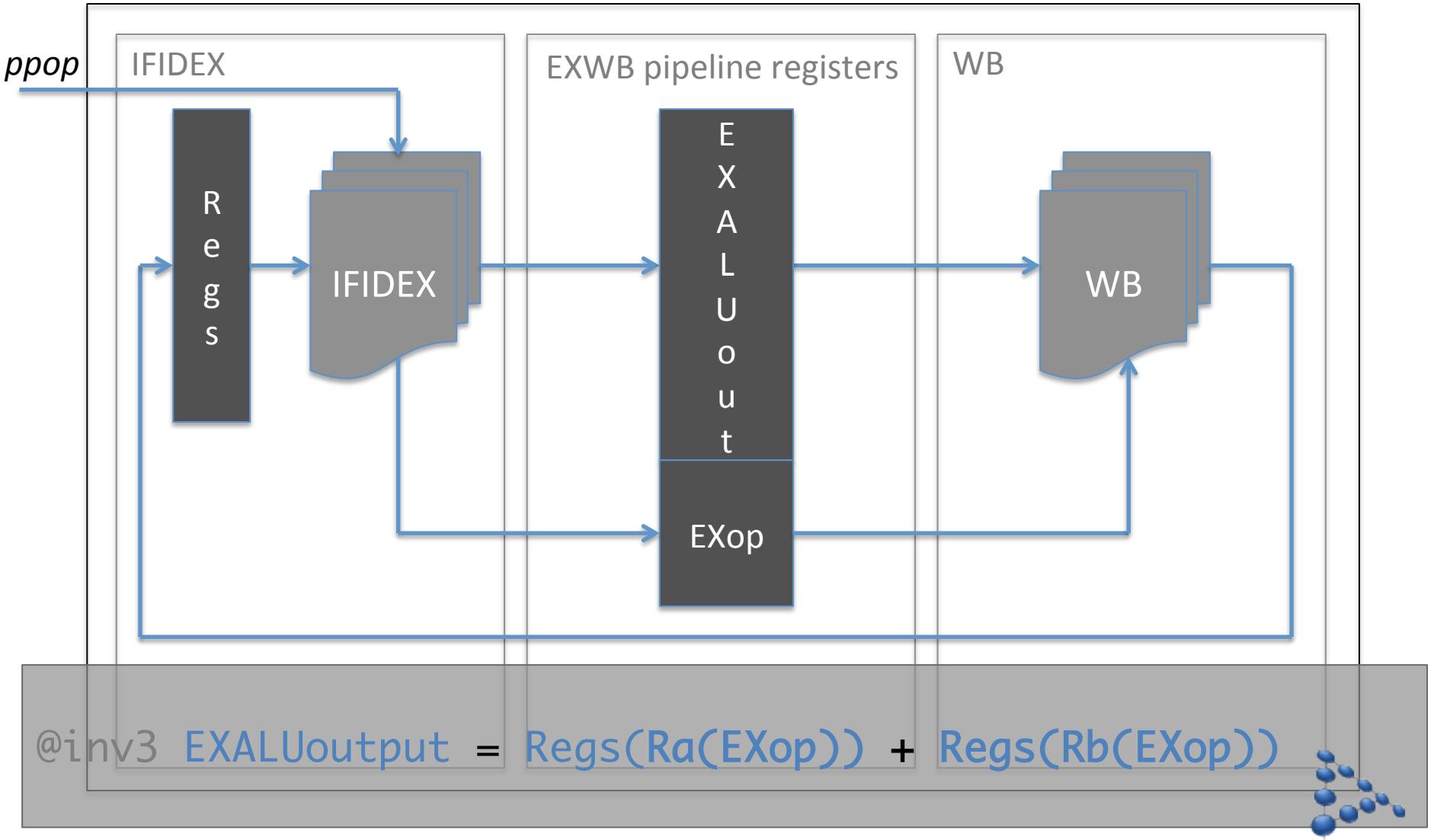

```
event EXWBnoRAW refines ArithRR
    any ppop
    where
      @grd1 EXop ∈ ArithRROp
      @grd2 ppop ∈ ArithRROp
    with
      @pop pop = EXop
    then
      @act1 Regs(Rr(EXop)) ≔ EXALUoutput
      @act2 EXALUoutput ≔  Regs(Ra(ppop)) + Regs(Rb(ppop))
      @act3 EXop ≔ ppop
  end
```



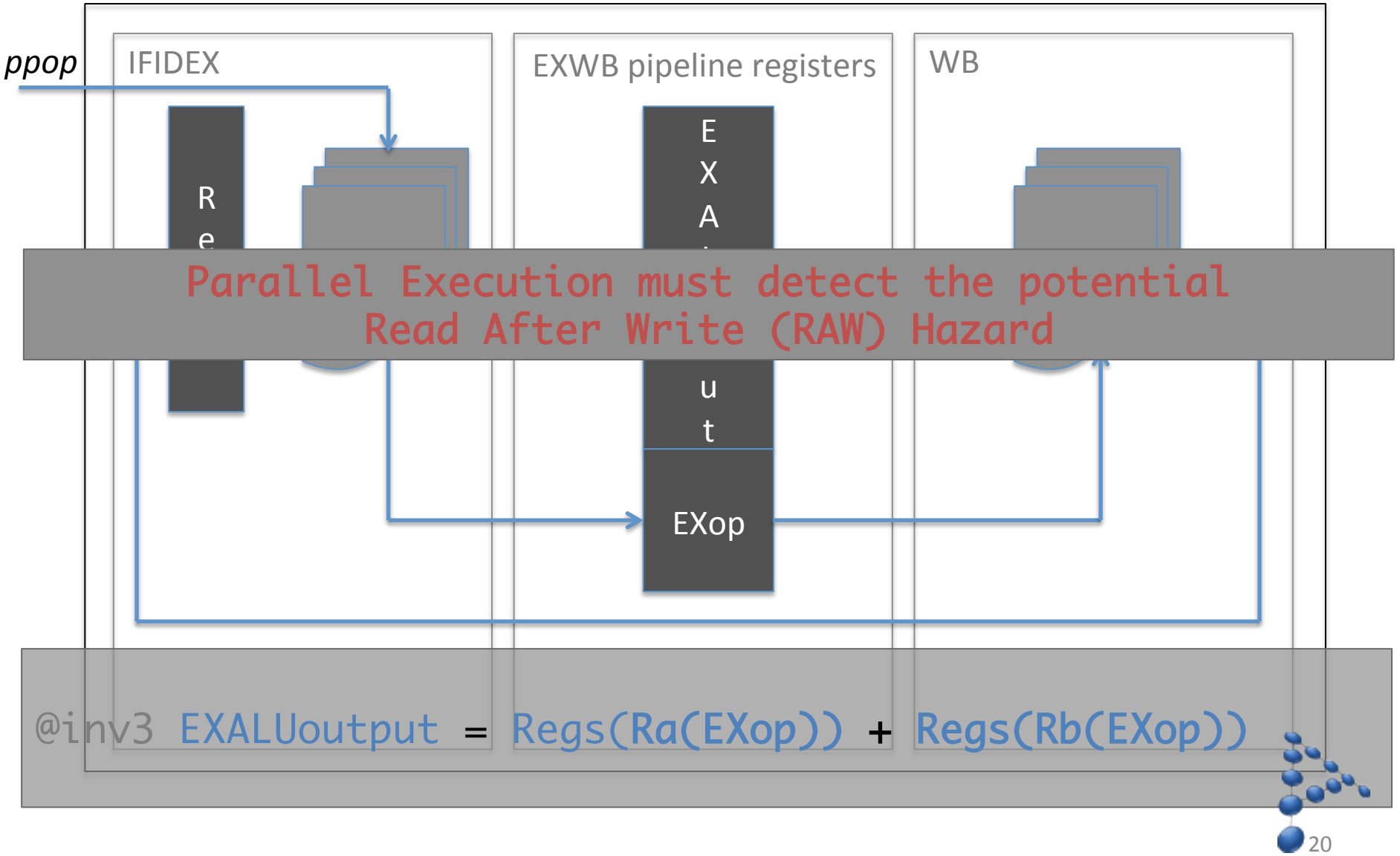

```
event ArithRR
  any pop
  where
    @grd1 pop ∈ ArithRROp
  then
    @act1 Regs(Rr(pop)) ≔ Regs(Ra(pop)) + Regs(Rb(pop))
  end
```

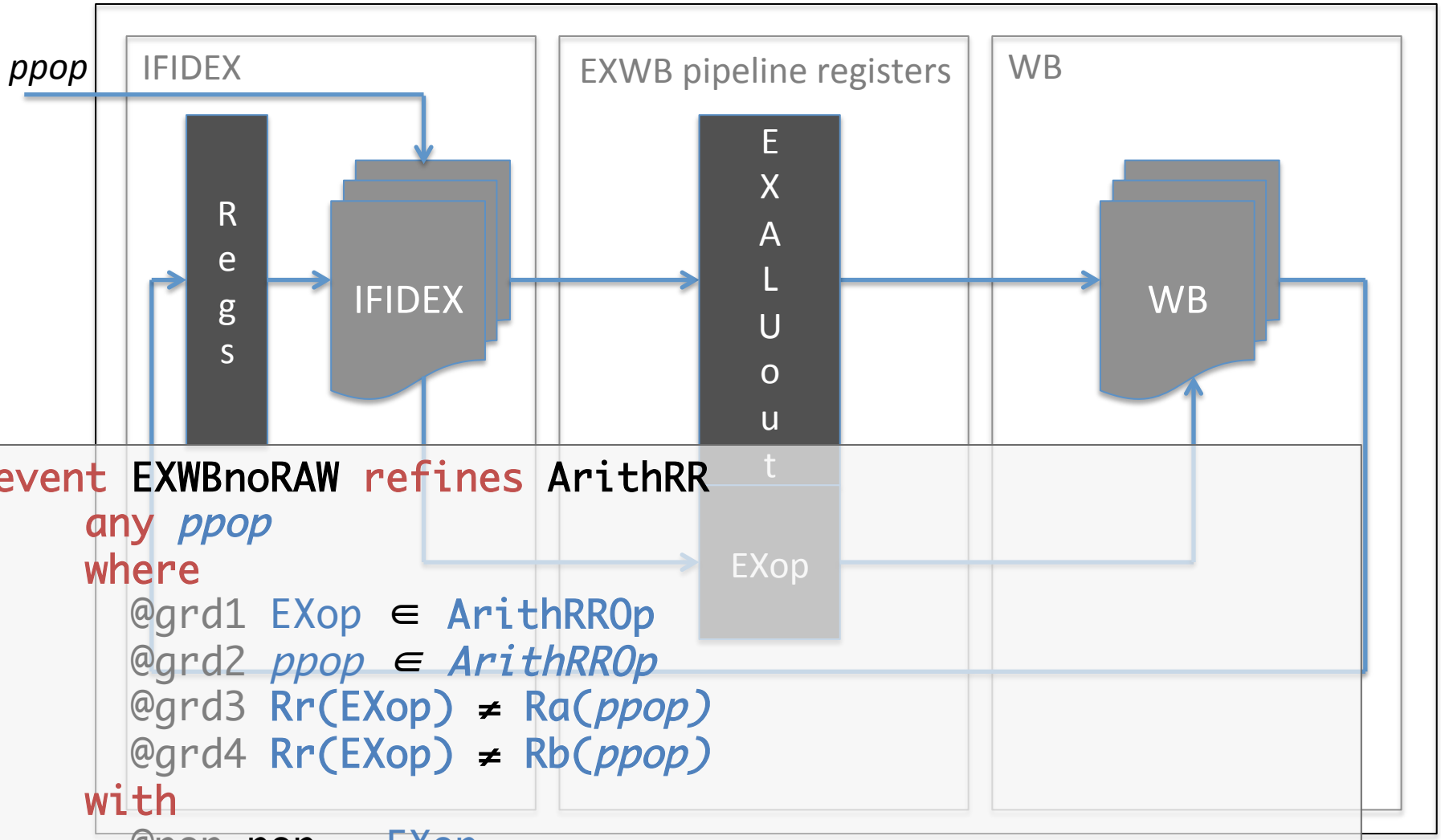
```
event EXWBnoRAW refines ArithRR
  any ppop
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 ppop ∈ ArithRROp
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 EXALUoutput ≔  Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act3 EXop ≔ ppop
  end
```

The Gluing Invariant

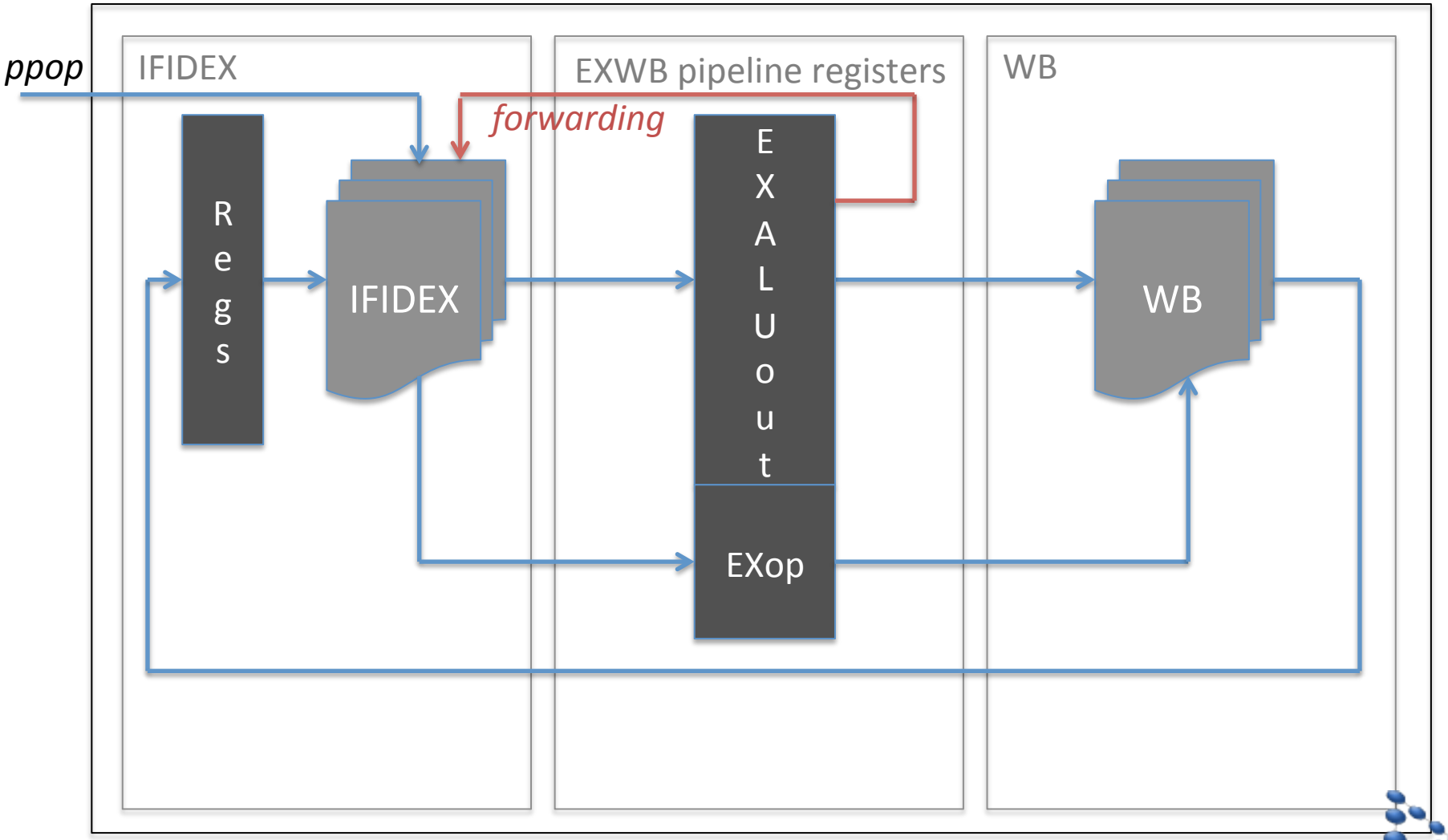


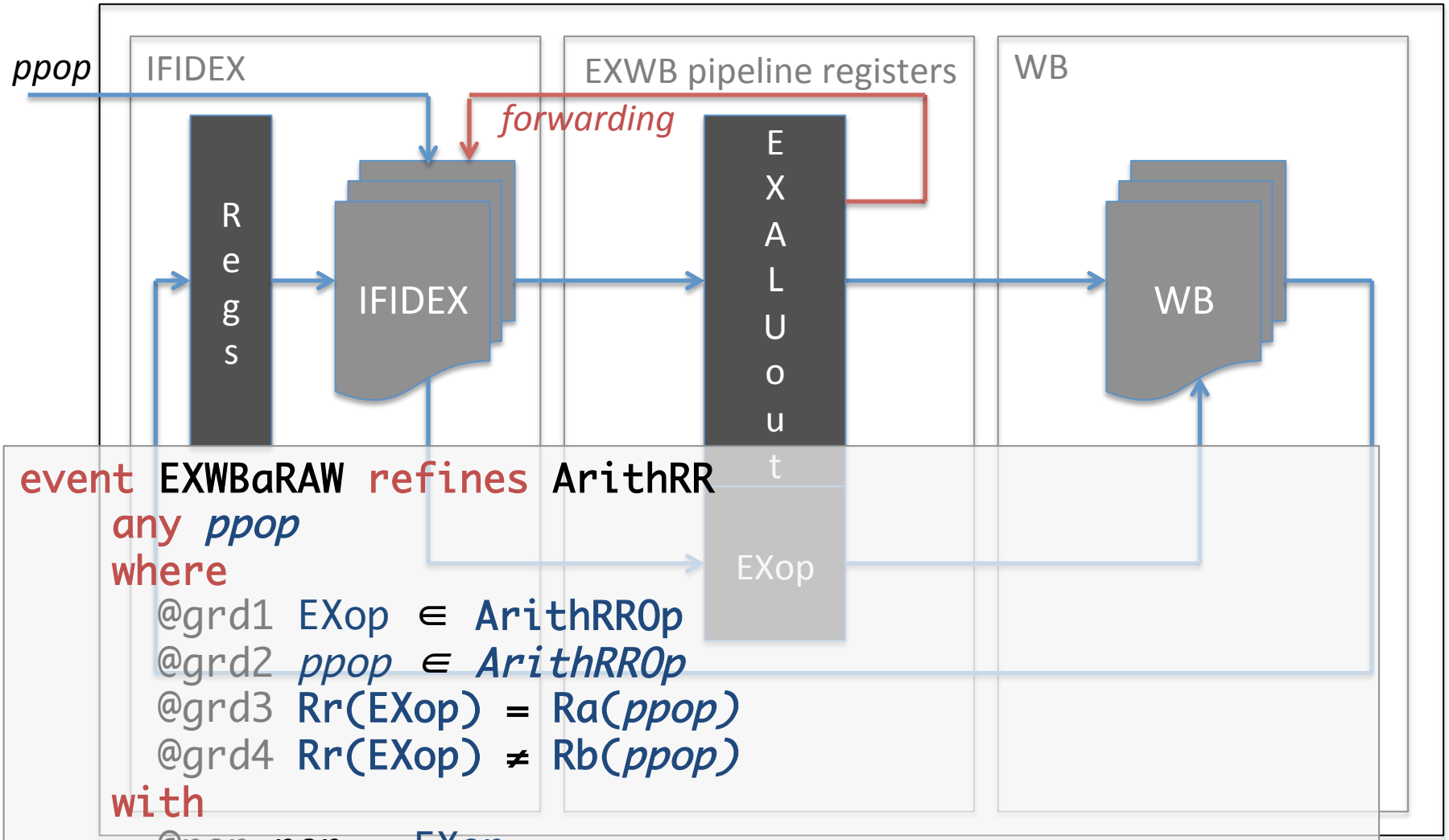
The Gluing Invariant



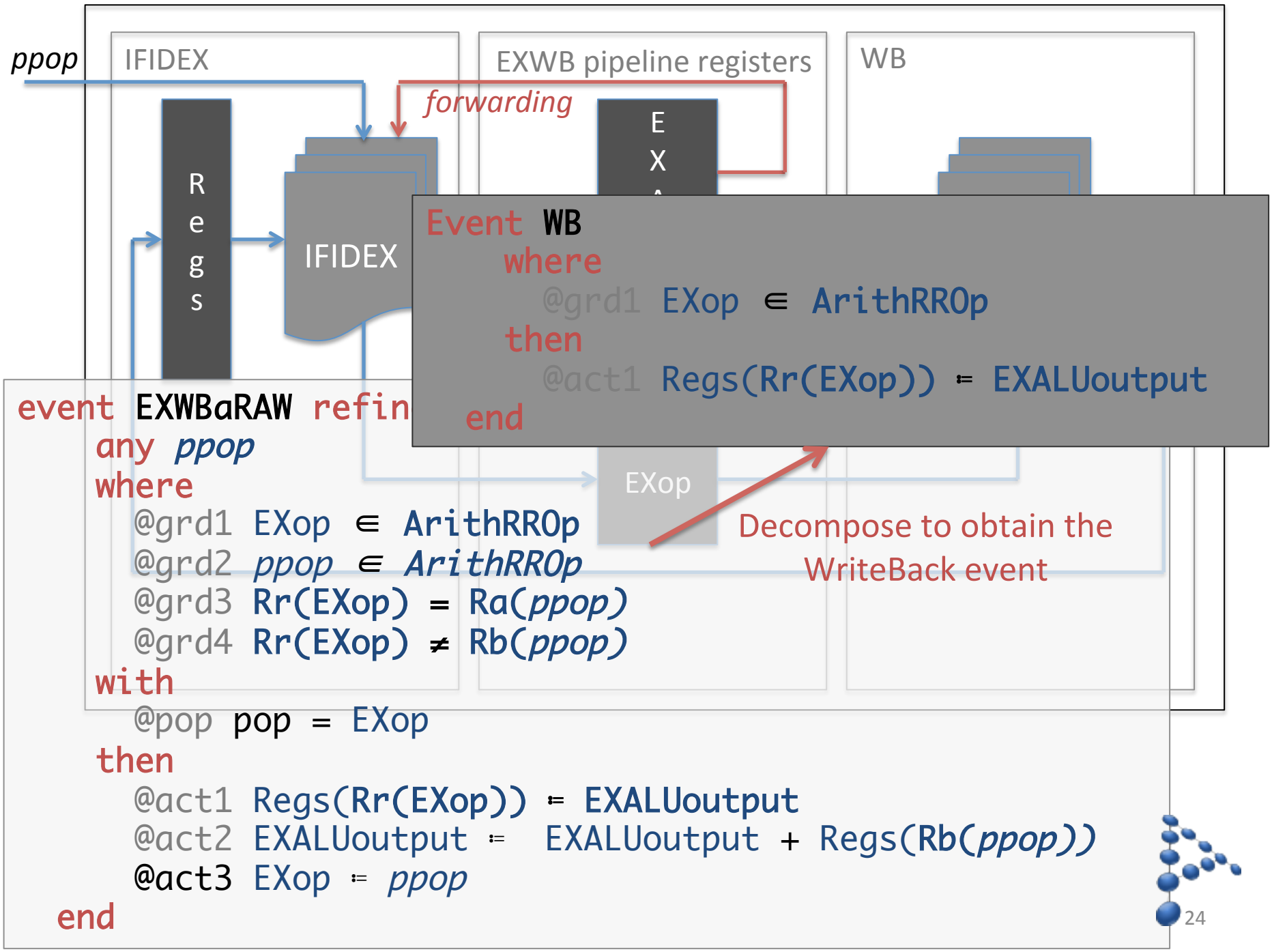



```
event EXWBnoRAW refines ArithRR
    any ppop
    where
      @grd1 EXop ∈ ArithRROp
      @grd2 ppop ∈ ArithRROp
      @grd3 Rr(EXop) ≠ Ra(ppop)
      @grd4 Rr(EXop) ≠ Rb(ppop)
    with
      @pop pop = EXop
    then
      @act1 Regs(Rr(EXop)) ≔ EXALUoutput
      @act2 EXALUoutput ≔  Regs(Ra(ppop)) + Regs(Rb(ppop))
      @act3 EXop ≔ ppop
  end
```


ppop
IFIDEX
EXWB pipeline registers
WB
forwarding
Regs
IFIDEX
EXALUout
EXop
WB



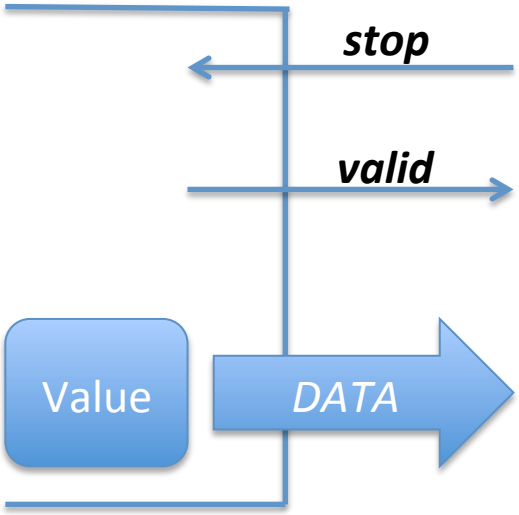

```
event EXWBaRAW refines ArithRR
    any ppop
    where
      @grd1 EXop ∈ ArithRROp
      @grd2 ppop ∈ ArithRROp
      @grd3 Rr(EXop) = Ra(ppop)
      @grd4 Rr(EXop) ≠ Rb(ppop)
    with
      @pop pop = EXop
    then
      @act1 Regs(Rr(EXop)) ≔ EXALUoutput
      @act2 EXALUoutput ≔  EXALUoutput + Regs(Rb(ppop))
      @act3 EXop ≔ ppop
  end
```

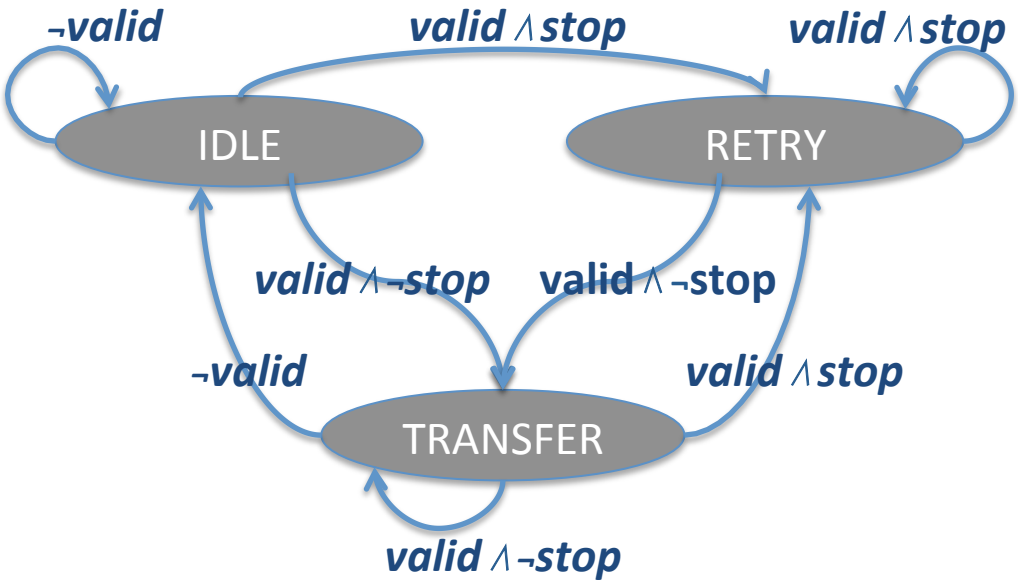

ppop
IFIDEX
EXWB pipeline registers
WB
forwarding
R e g s
IFIDEX
E X
EXop
Event WB
where
@grd1 EXop ∈ ArithRROp
then
@act1 Regs(Rr(EXop)) := EXALUoutput
end
event EXWBaRAW refin
any ppop
where
@grd1 EXop ∈ ArithRROp
@grd2 ppop ∈ ArithRROp
@grd3 Rr(EXop) = Ra(ppop)
@grd4 Rr(EXop) ≠ Rb(ppop)
with
@pop pop = EXop
then
@act1 Regs(Rr(EXop)) := EXALUoutput
@act2 EXALUoutput := EXALUoutput + Regs(Rb(ppop))
@act3 EXop := ppop
end
Decompose to obtain the WriteBack event

Forwarding and Centralised Stalling

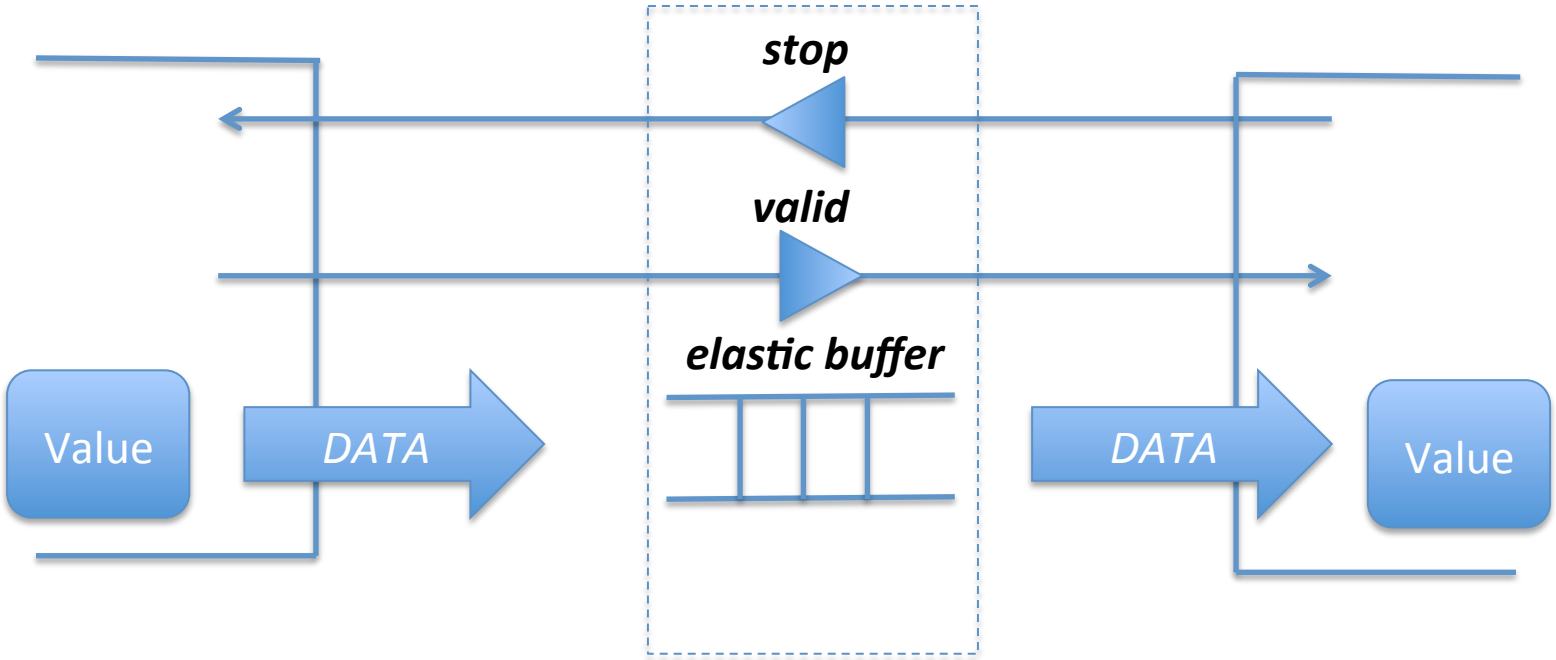
- As the pipeline gets longer
  - More forwarding tracks are required
    - the feedback tracks get longer
  - Centralised stalling to manage branching becomes more complex and difficult to verify
    - the feedback tracks get longer
- Synchronous Elastic Buffers provide an alternative solution
  - Latency insensitive
  - Distributed stalling
  - First used by Intel to meet timing requirements

The SELF Protocol

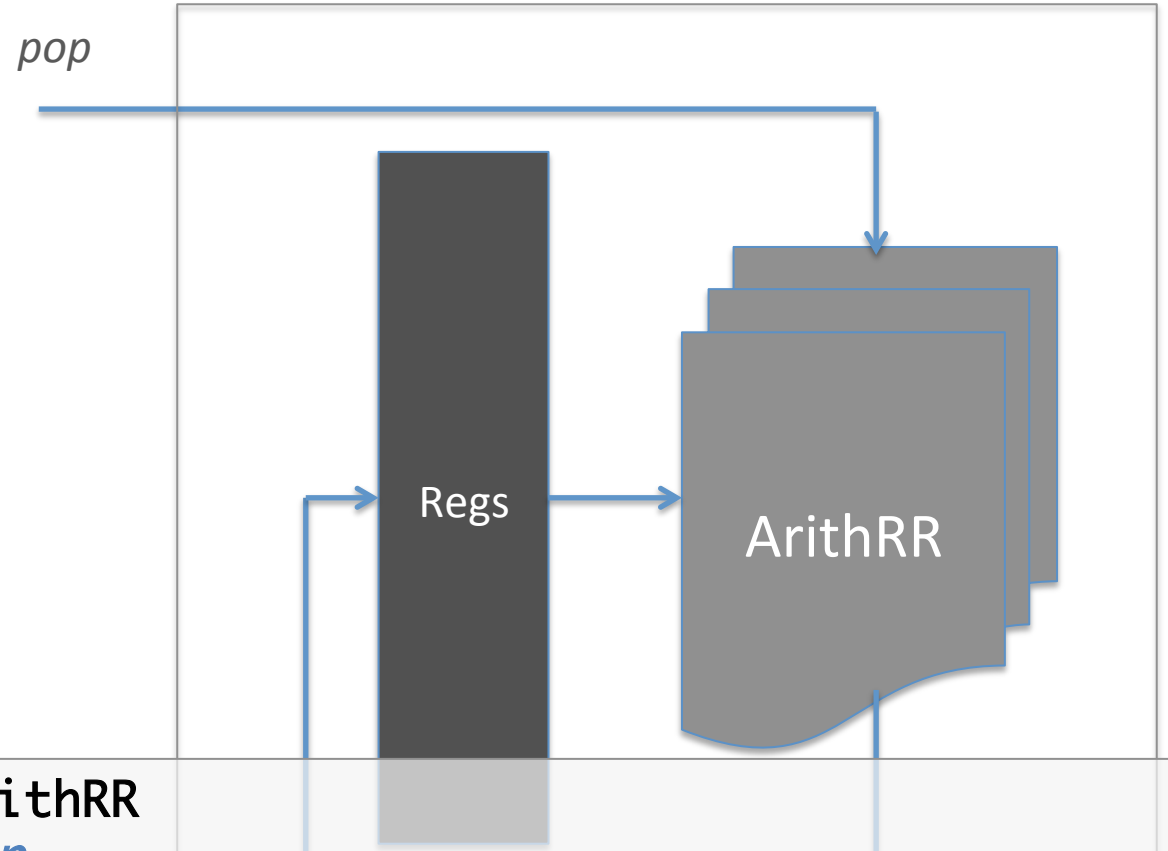




Connecting two Elastic Components

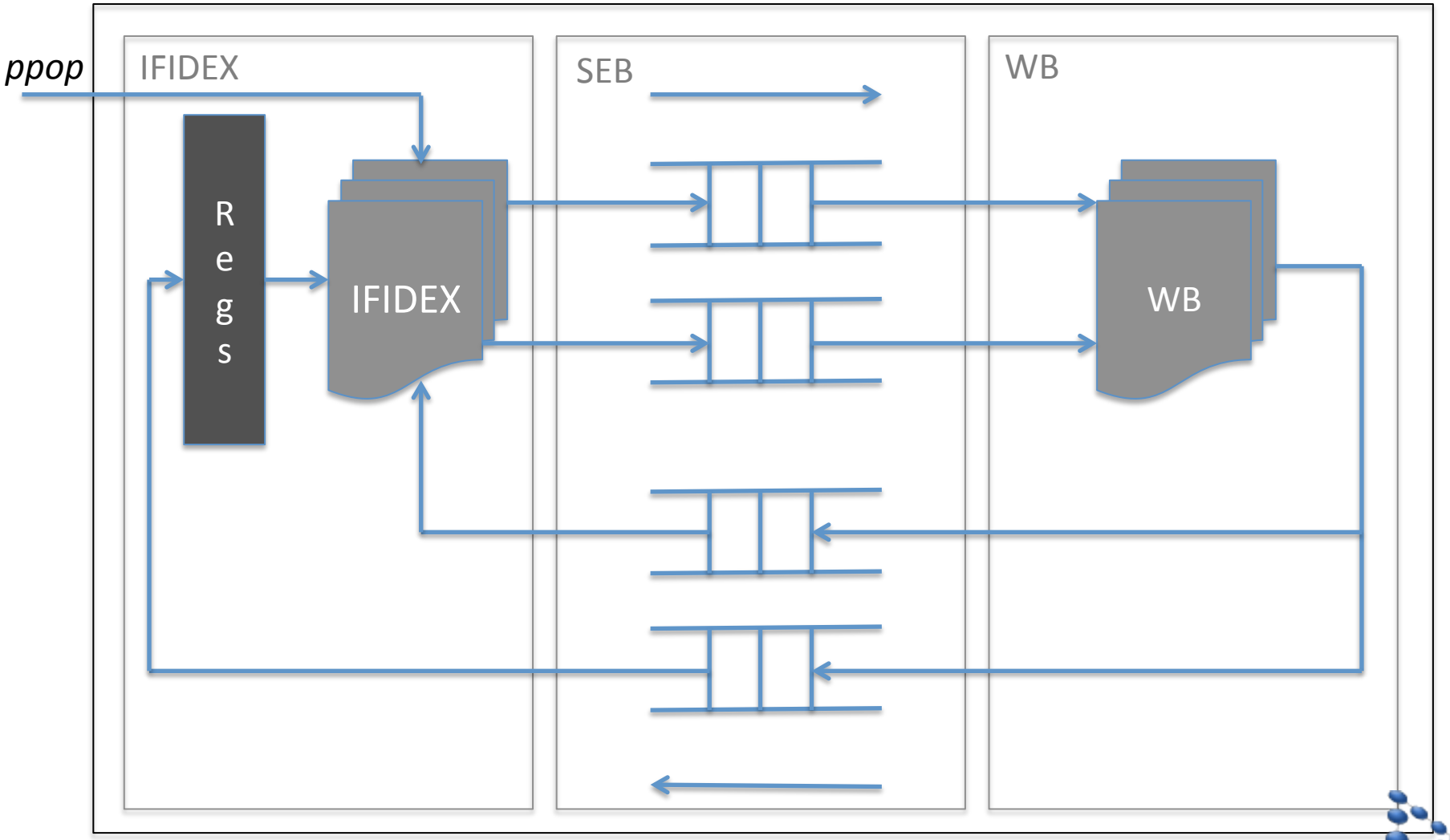


Recall Abstract Machine Micro-architecture

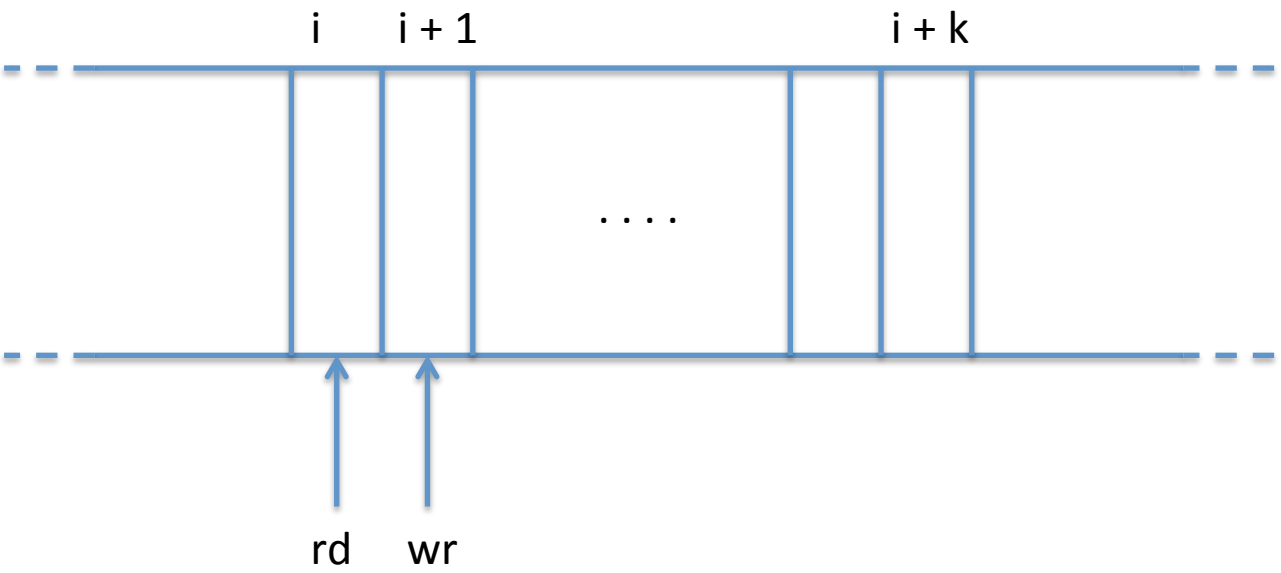



```
event ArithRR
  any pop
  where
    @grd1 pop ∈ ArithRROp
  then
    @act1 Regs(Rr(pop)) ≔ Regs(Ra(pop)) + Regs(Rb(pop))
end
```

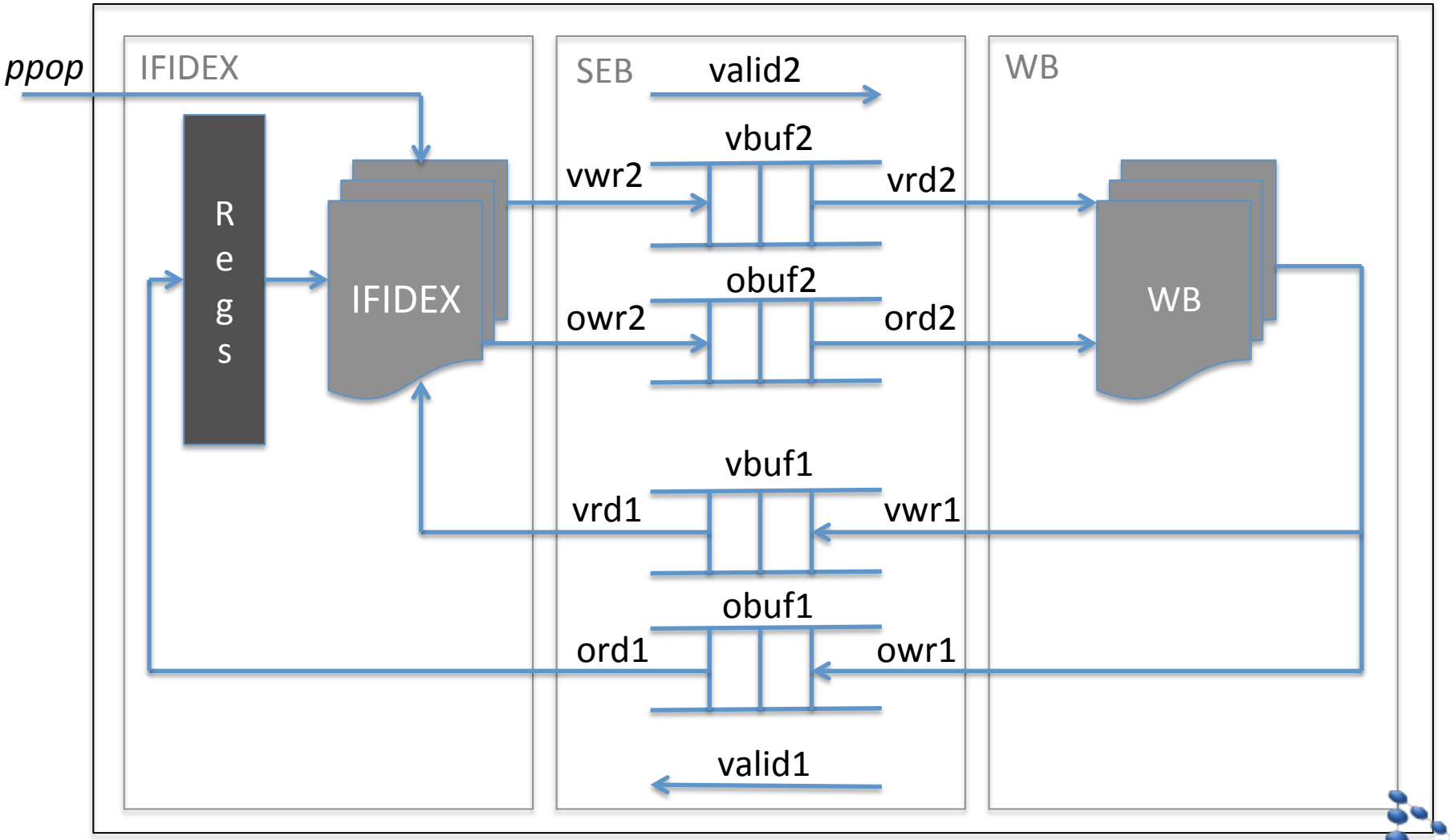
Refinement with Synchronous Elastic Buffers

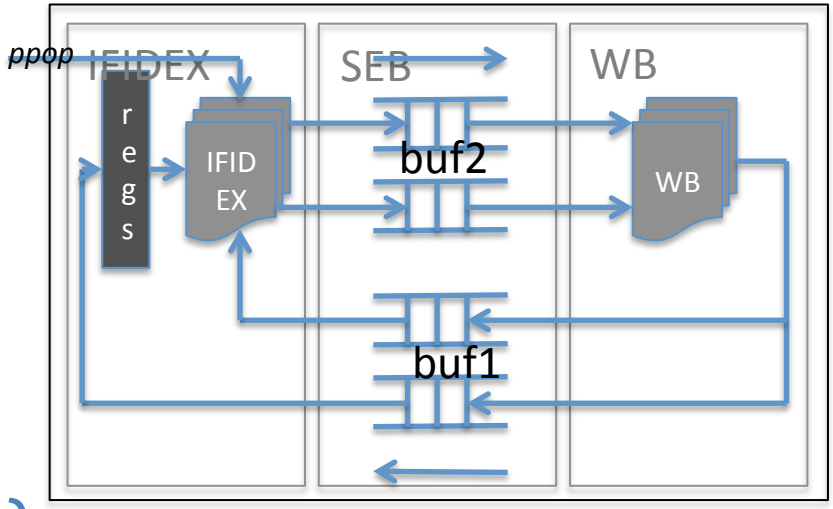


Abstract Synchronous Elastic Buffer



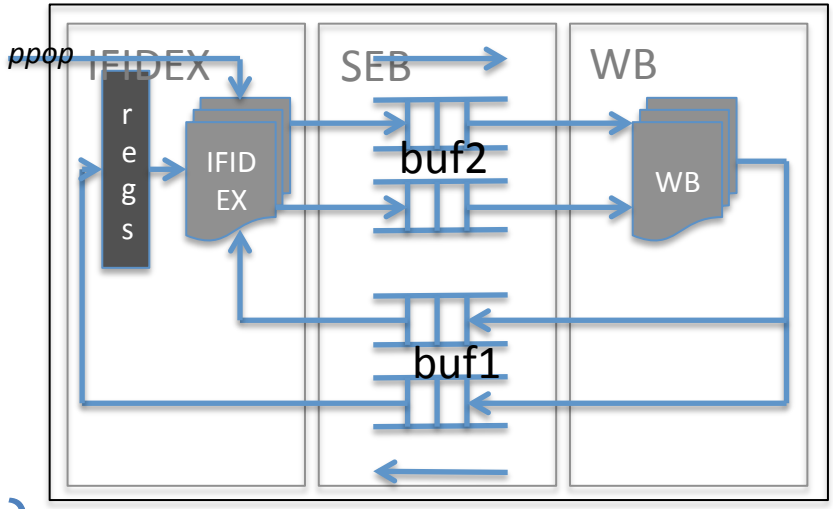
Refinement with Synchronous Elastic Buffers






```
event EXWBnoRAW refines ArithRR
  any pppop
  where
    @grd1 pppop ∈ ArithRROp
    @grd2 obuf1(ord1) ∈ ArithRROp
    @grd3 obuf2(ord2) ∈ ArithRROp
    @grd4 Valid1 = TRUE
    @grd5 Rr(obuf1(ord1)) ≠ Ra(pppop)
    @grd6 Rr(obuf1(ord1)) ≠ Rb(pppop)
    @grd7 Rr(obuf2(ord2)) ≠ Ra(pppop)
    @grd8 Rr(obuf2(ord2)) ≠ Rb(pppop)
    @grd9 Valid2 = TRUE
  with
    @pop pop = obuf1(ord1)
  then
    @act1 Regs(Rr(obuf1(ord1))) ≔ vbuf1(vrd1)
    @act2 obuf1(owr1) ≔ obuf2(ord2)
    @act3 vbuf1(vwr1) ≔ vbuf2(vrd2)
    @act4 vbuf2(vwr2) ≔ Regs(Ra(pppop)) + Regs(Rb(pppop))
    @act5 obuf2(owr2) ≔ pppop
       … // update buffer indices
end
```

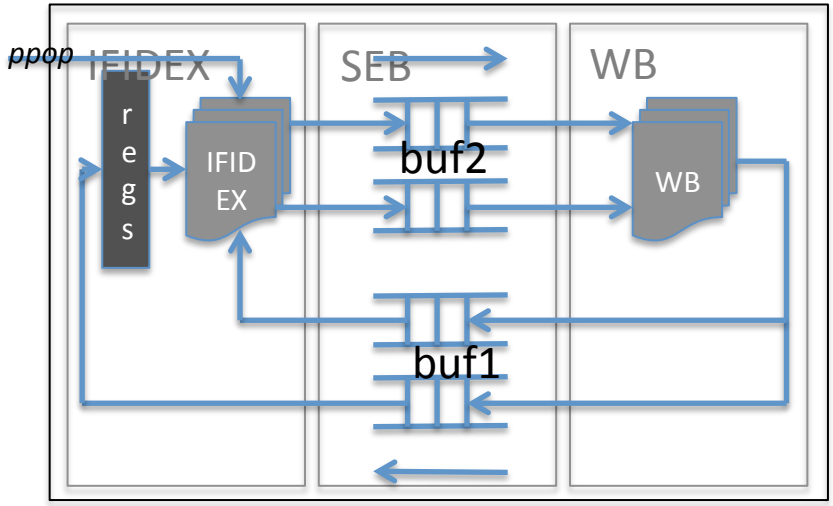
```
event EXWBnoRAW refines ArithRR
  any pppop
  where
    @grd1 pppop ∈ ArithRROp
    @grd2 obuf1(ord1) ∈ ArithRROp
    @grd3 obuf2(ord2) ∈ ArithRROp
    @grd4 Valid1 = TRUE
    @grd5 Rr(obuf1(ord1)) ≠ Ra(pppop)
```



Gluing Invariants

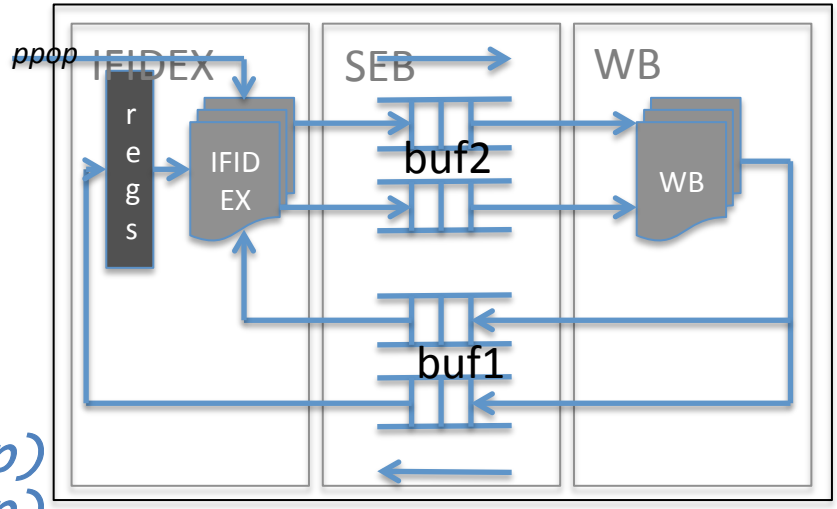
```
Valid1 = TRUE ⇒
vbuf1(vrd1) = Regs(Ra(obuf1(ord1))) + Regs(Rb(obuf1(ord1)))

Valid2 = TRUE ⇒
vbuf2(vrd2) = Regs(Ra(obuf2(ord2))) + Regs(Rb(obuf2(ord2)))

EXALUoutput = Regs(Ra(EXop)) + Regs(Rb(EXop))
```

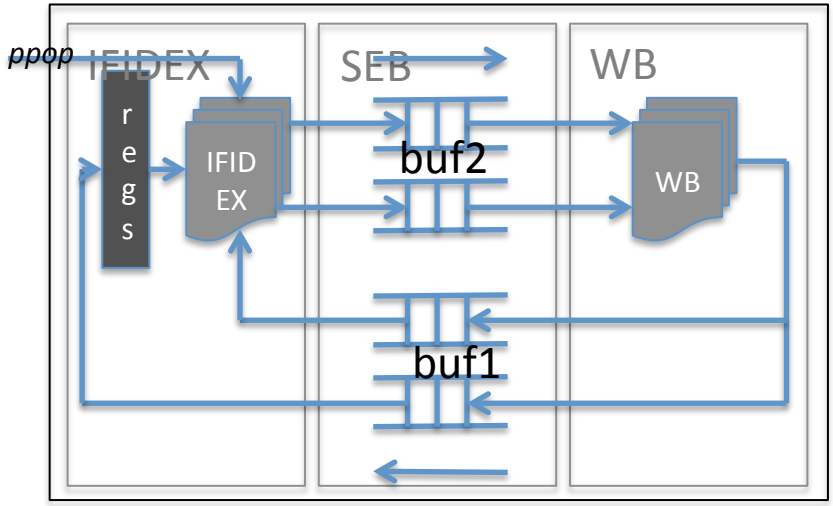
```
    … // update buffer indices
end
```



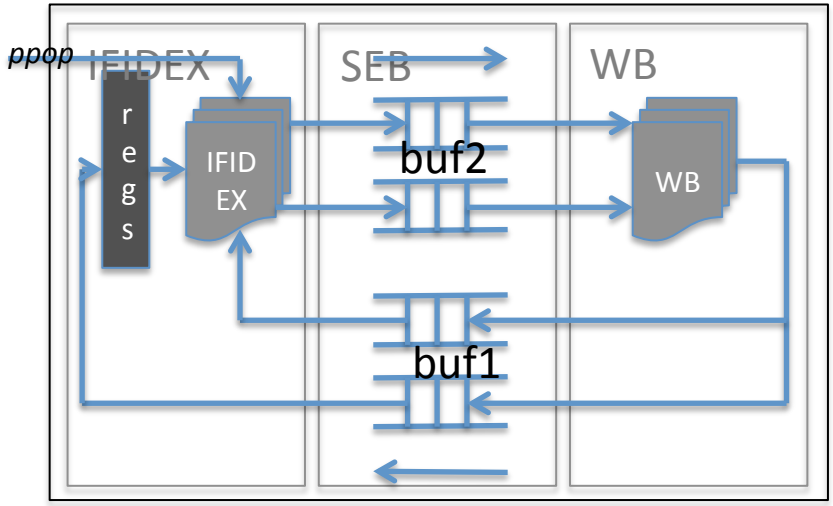

```
event EXWBRAWa refines ArithRR
  any pppop
  where
    @grd1 pppop ∈ ArithRROp
    @grd2 obuf1(ord1) ∈ ArithRROp
    @grd3 obuf2(ord2) ∈ ArithRROp
    @grd4 Valid1 = TRUE
    @grd5 Rr(obuf1(ord1)) = Ra(pppop)
    @grd6 Rr(obuf1(ord1)) ≠ Rb(pppop)
    @grd7 Rr(obuf2(ord2)) ≠ Ra(pppop)
    @grd8 Rr(obuf2(ord2)) ≠ Rb(pppop)
    @grd9 Valid2 = TRUE
  with
    @pop pop = obuf1(ord1)
  then
    @act1 Regs(Rr(obuf1(ord1))) ≔ vbuf1(vrd1)
    @act2 obuf2(owr2) ≔ pppop
    @act3 obuf1(owr1) ≔ obuf2(ord2)
    @act4 vbuf1(vwr1) ≔ vbuf2(vrd2)
    @act5 Valid2 ≔ FALSE
       … // update buffer indices
end
```

```
event EXstallWB refines ArithRR
  any pppop
  where
    @grd1 pppop ∈ ArithRROp
    @grd2 obuf1(ord1) ∈ ArithRROp
    @grd3 obuf2(ord2) ∈ ArithRROp
    @grd4 Valid1 = TRUE
    @grd5 Rr(obuf1(ord1)) ≠ Ra(pppop)
    @grd6 Rr(obuf1(ord1)) ≠ Rb(pppop)
    @grd7 Rr(obuf2(ord2)) ≠ Ra(pppop)
    @grd8 Rr(obuf2(ord2)) ≠ Rb(pppop)
    @grd9 Valid2 = FALSE
  with
    @pop pop = obuf1(ord1)
  then
    @act1 Regs(Rr(obuf1(ord1))) ≔ vbuf1(vrd1)
    @act2 obuf1(owr1) ≔ obuf2(ord2)
    @act3 vbuf1(vwr1) ≔ vbuf2(vrd2)
    @act4 vbuf2(vwr2) ≔ Regs(Ra(pppop)) + Regs(Rb(pppop))
    @act5 obuf2(owr2) ≔ pppop
    @act6 Valid1 ≔ FALSE
    @act7 Valid2 ≔ TRUE
      … // update buffer indices
  end
```






```
event EXWBstall
  any pppop
  where
    @grd1 pppop ∈ ArithRROp
    @grd2 obuf2(ord2) ∈ ArithRROp
    @grd2 obuf2(ord2) = obuf1(ord1)
    @grd3 Valid1 = FALSE
    @grd4 Rr(obuf1(ord1)) ≠ Ra(pppop)
    @grd5 Rr(obuf1(ord1)) ≠ Rb(pppop)
    @grd6 Rr(obuf2(ord2)) ≠ Ra(pppop)
    @grd7 Rr(obuf2(ord2)) ≠ Rb(pppop)
    @grd8 Valid2 = TRUE
  then
    @act1 vbuf2(vwr2) := Regs(Ra(pppop)) + Regs(Rb(pppop))
    @act2 obuf2(owr2) := pppop
    @act3 vbuf1(vwr1) := vbuf2(vrd2)
    @act4 obuf1(owr1) := obuf2(ord2)
    @act5 Valid1 := TRUE
      … // update buffer indices
 end
```

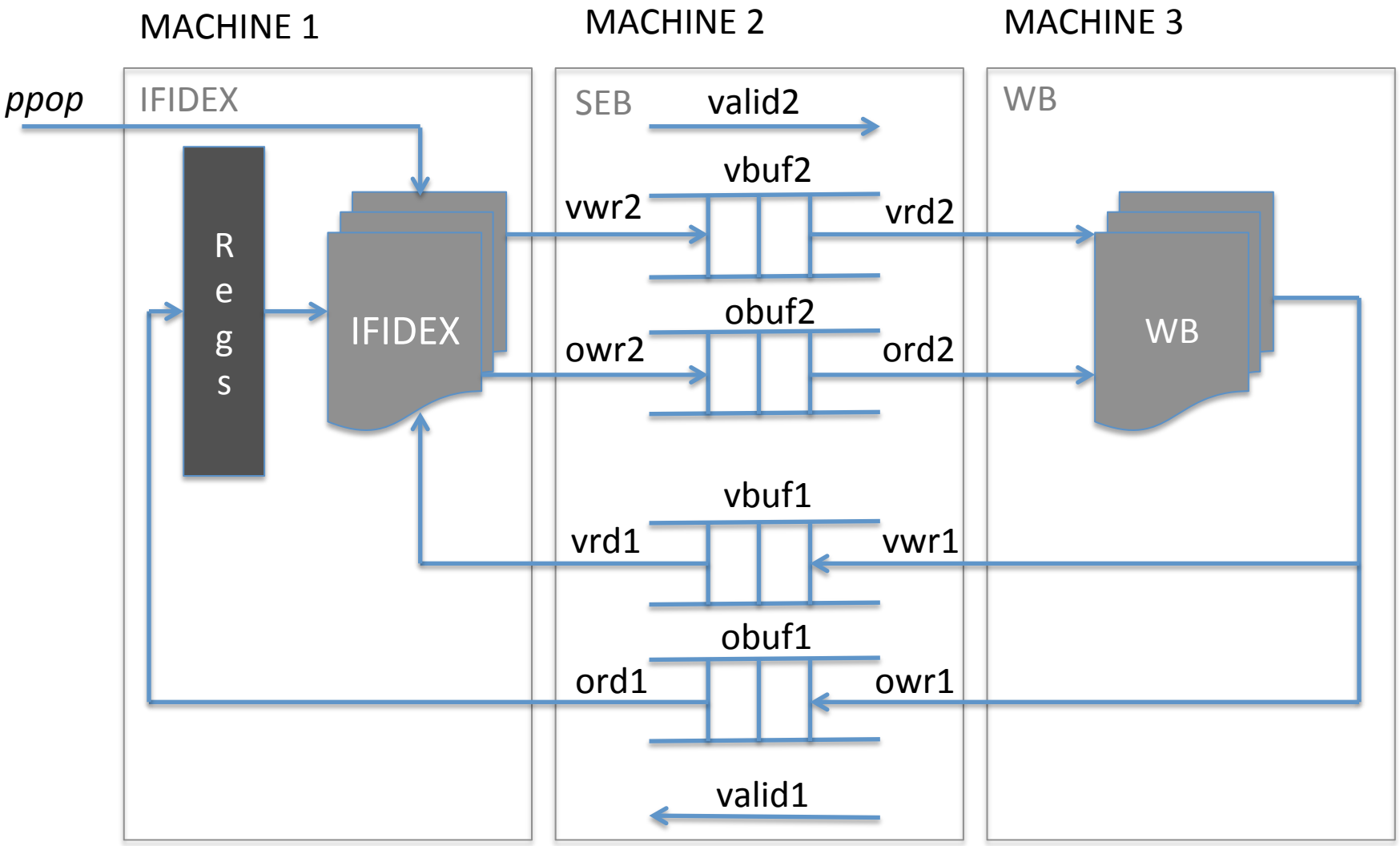



```
event EXWBstall
  any pppop
  where
    @grd1 pppop ∈ ArithRROp
    @grd2 obuf2(ord2) ∈ ArithRROp
    @grd2 obuf2(ord2) = obuf1(ord1)
    @grd3 Valid1 = FALSE
    @grd4 Rr(obuf1(ord1)) = Rg(pppop)
    @grd5 Rr
    @grd6 Rr
    @grd7 Rr
    @grd8 Va
  then
    @act1 vb                                    op))
    @act2 ob
    @act3 vb
    @act4 ob
    @act5 Valid1 := TRUE
      … // update buffer indices
end
```

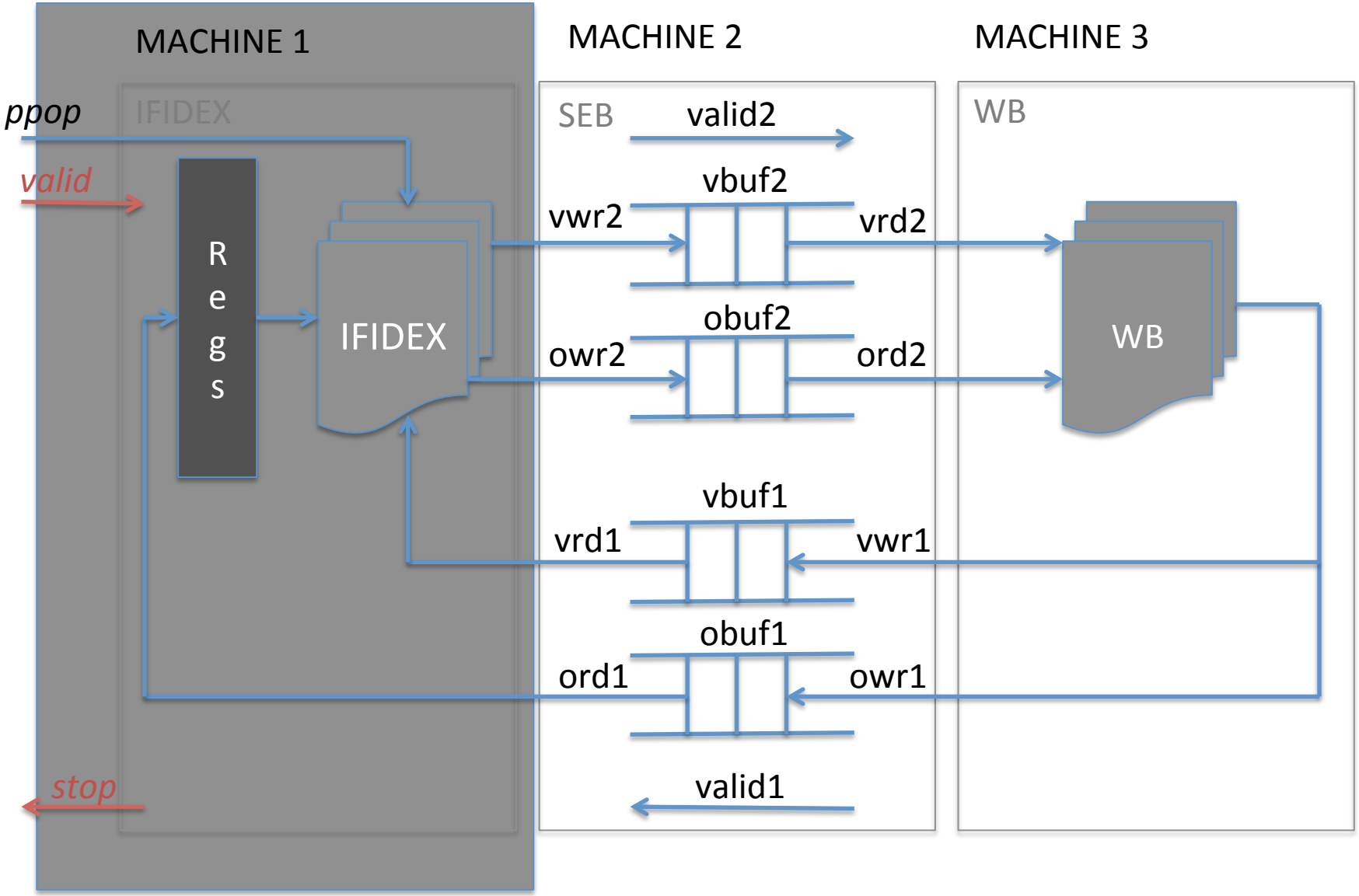
Responsiveness

$$Valid1 = TRUE \lor Valid2 = TRUE$$

Shared Event Pipeline Decomposition

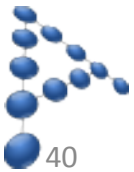


Second Refinement: MACHINE 1



Pipelining Summary

- Micro-architectural exploration is raised to the Specification Level using Event-B
- An alternative to forwarding and centralised stalling has been explored using Synchronous Elastic Buffers
- Latency Insensitivity is introduced at Low Cost
- Track lengths are reduced
- Synchronous Elastic Buffers allow performance goals to be met in a verifiable way
- Verification is raised to the Specification Level



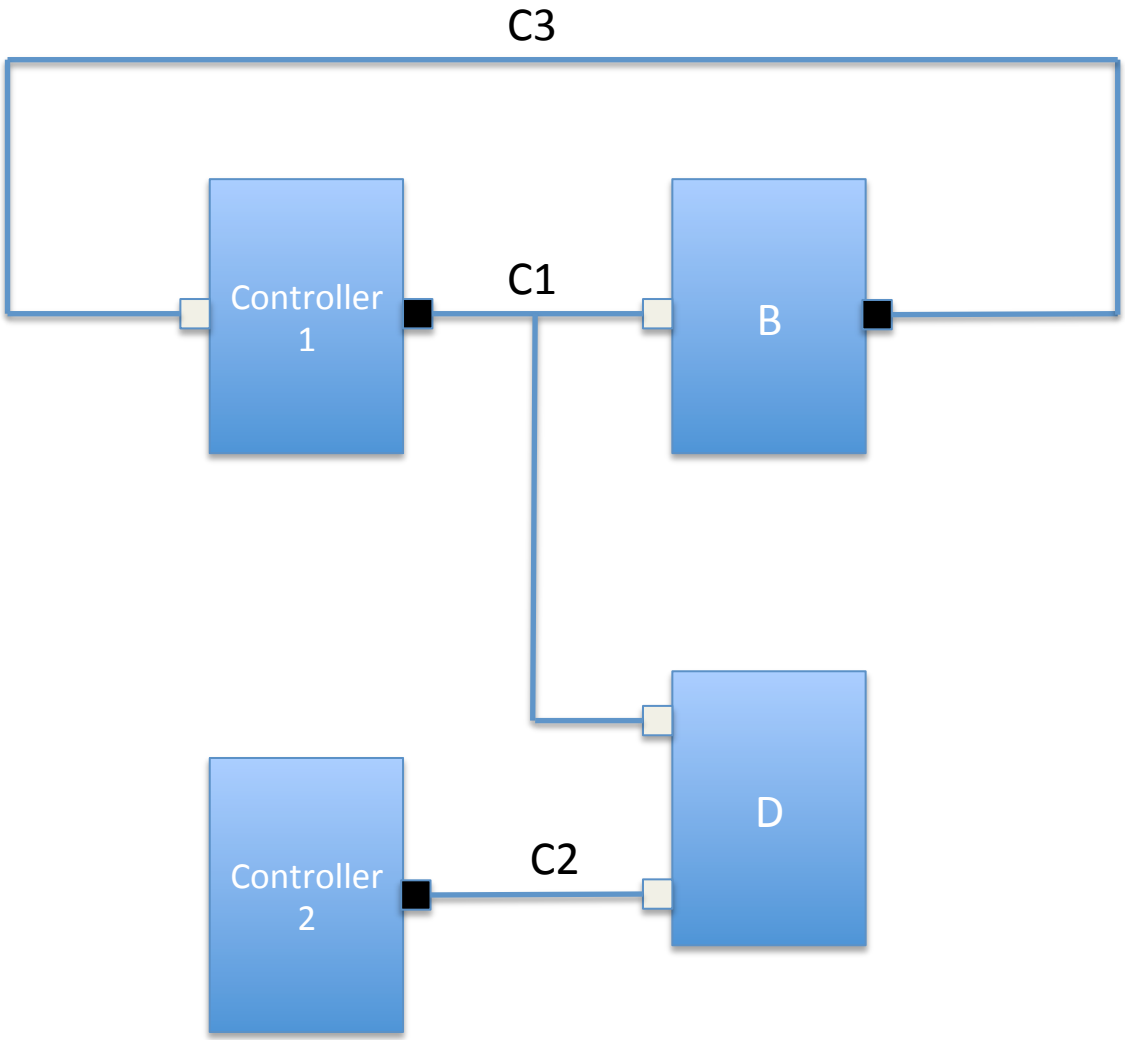
Temporal Modeling in Cyber-physical systems

- Simulating Formal Models
- Modeling Timing Cycles
  - Component Modes
  - Generalised Update/Evaluation Modes
- A Simple Example
- Summary

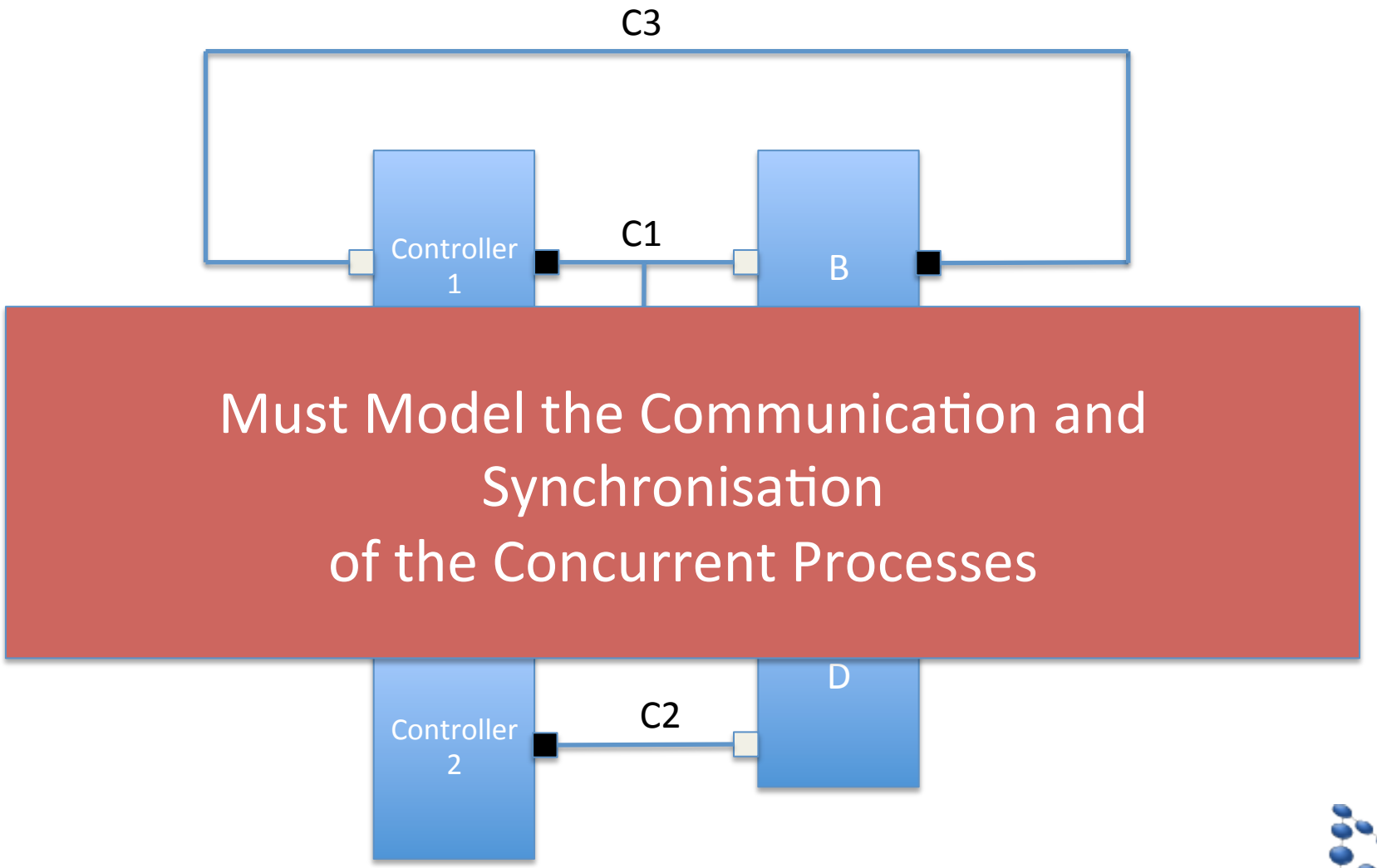
Temporal Modeling in Cyber-physical Systems: Requirements

- Distributed Function and Control
- Managing Safety Hazards
- Verifying the relationships between Inputs and Outputs

Distributed Function and Control

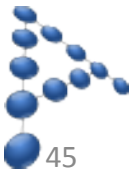


Distributed Function and Control



Managing Safety Hazards

- Plant model Evaluates
- Potential Hazards are Detected
- The Controller manages the Hazards
- The Controller predicts the future behaviour of the Plant
- Loop must be generalised for multiple controllers and distributed Plant



Verifying the relationships between Inputs and Outputs

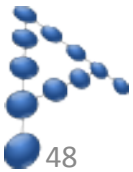
- DO-178C Formal Supplement
  - Must show that
    1. Outputs fully satisfy Inputs
    2. Each Output data item is necessary to satisfy some Input data item (*No unintended behaviour*)
  - Must show that
    - Input/Output specification is preserved by chosen implementation architecture
- ACSL
  - Ansi ISO C Specification Language
  - Code annotations – used by Airbus

Simulating Formal Models

- Abstract Model(s) may be *untimed*
- Refined Models represent Concurrent, Communicating Processes
  - will need to introduce some notion of a ***tick***
    - Cycle-based execution
    - Timed execution of *Delays* and *Deadlines*
- ProB has the notion of the *next state*
  - an event is executed (LTL *X*)
- We need the notion of the *next tick*

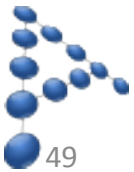
Modeling Timing Cycles: Component Modes

- Eg for Hazard Analysis
  - ***Plant*** Mode
  - Detect Mode
  - ***Controller*** Mode
  - Predict Mode
- Necessary to define an *ordering* on the modes
- The Plant may need to evaluate at a much higher rate than the Controller

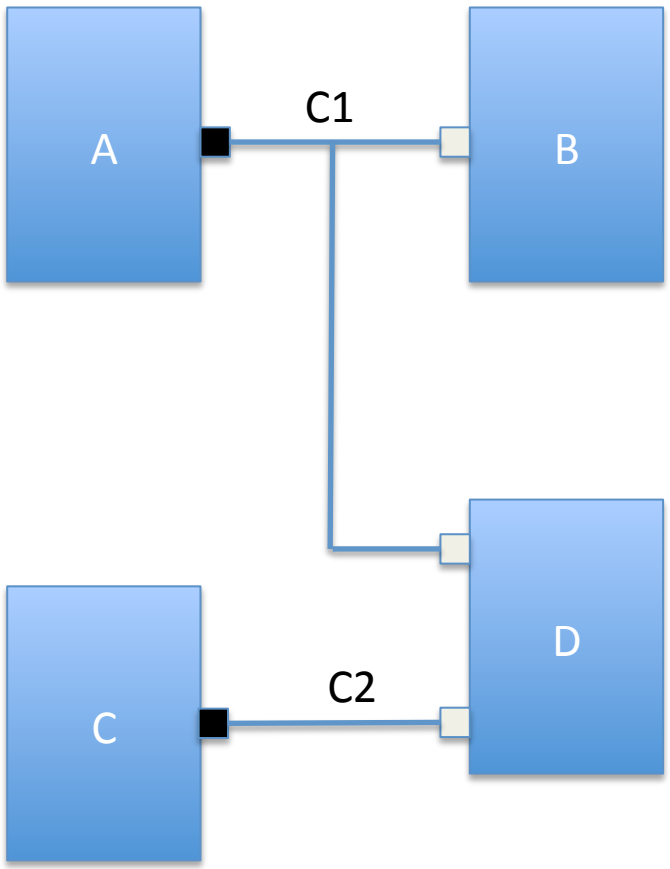


Modeling Timing Cycles: Update/Evaluate Modes

- Used by many Commercial Discrete Event Simulators
  - SystemC
  - Verilog/VHDL
- Supports arbitrary topology complexity
- No zero-delay communication between components
  - Components can evaluate in any order
- Components “suspend” between wake-ups
  - Input change
  - Self wake
  - Components can evaluate at different rates
- Discrete Time, Cycle-based or both



Discrete Event Simulation



COMPONENT VIEW

Components: A, B, C, D (processes)

Connections: C1, C2 (unidirectional)

Ports: IN □ OUT ■

SIMULATOR API

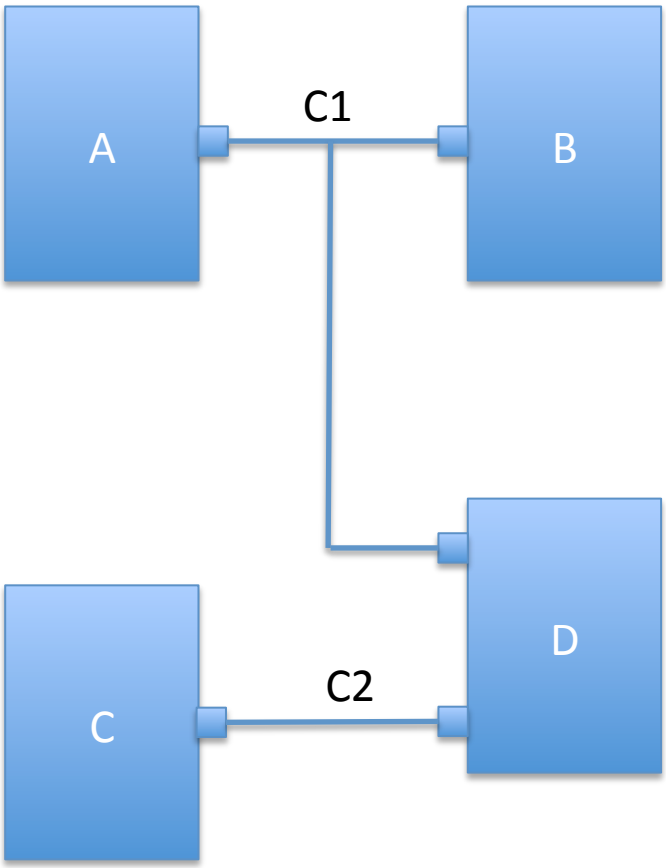
GetValue(port)

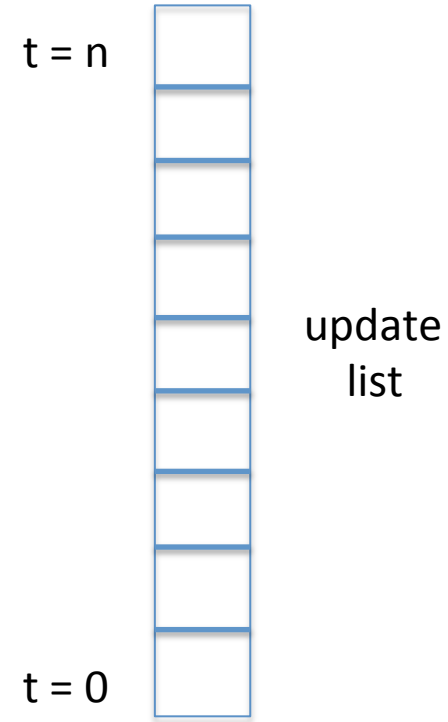
HasChanged(port)

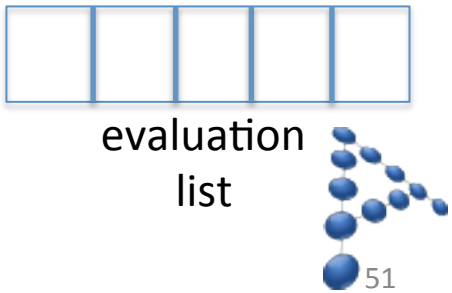
SetValue(OUT port, val, delay)

ScheduleEval(component, delay)

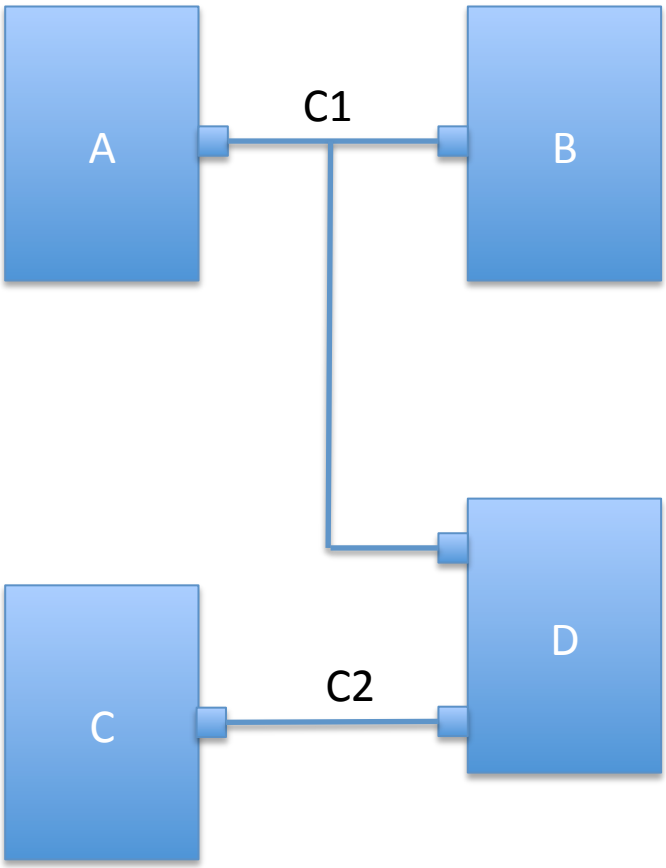
The Two-list Simulation Algorithm



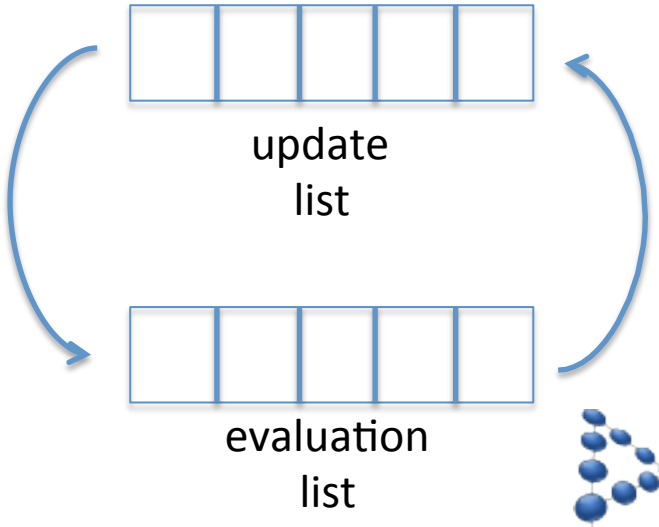




Simulating Models without Discrete Delays – *Unit Delay*



Each evaluate/update cycle advances time by one *tick*



A Simple Arithmetic Example
The Abstract Specification – *Output* as a function of *Inputs*

constants Inputs In1 In2

sets Parameters

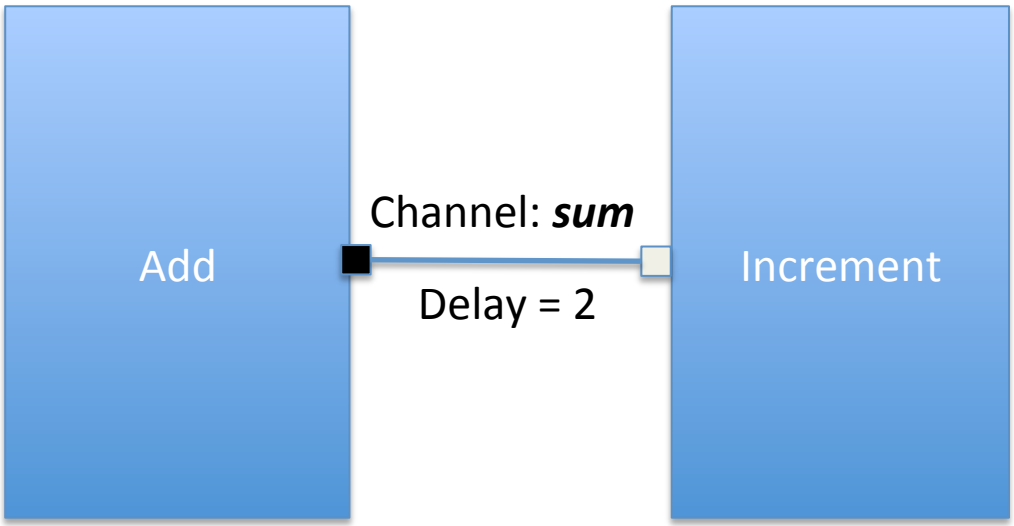
axioms
@axm1 **Inputs ⊆ Parameters**
@axm2 **In1 ∈ Inputs → ℕ**
@axm3 **In2 ∈ Inputs → ℕ**
end

event AddInc
any *p*
where
@grd1 *p* ∈ ***Inputs***
@grd2 **In1(*p*) ∈ ℕ**
@grd3 **In2(*p*) ∈ ℕ**
then
@act1 v ≔ **In1(*p*) + *In2(p)* + 1**
end

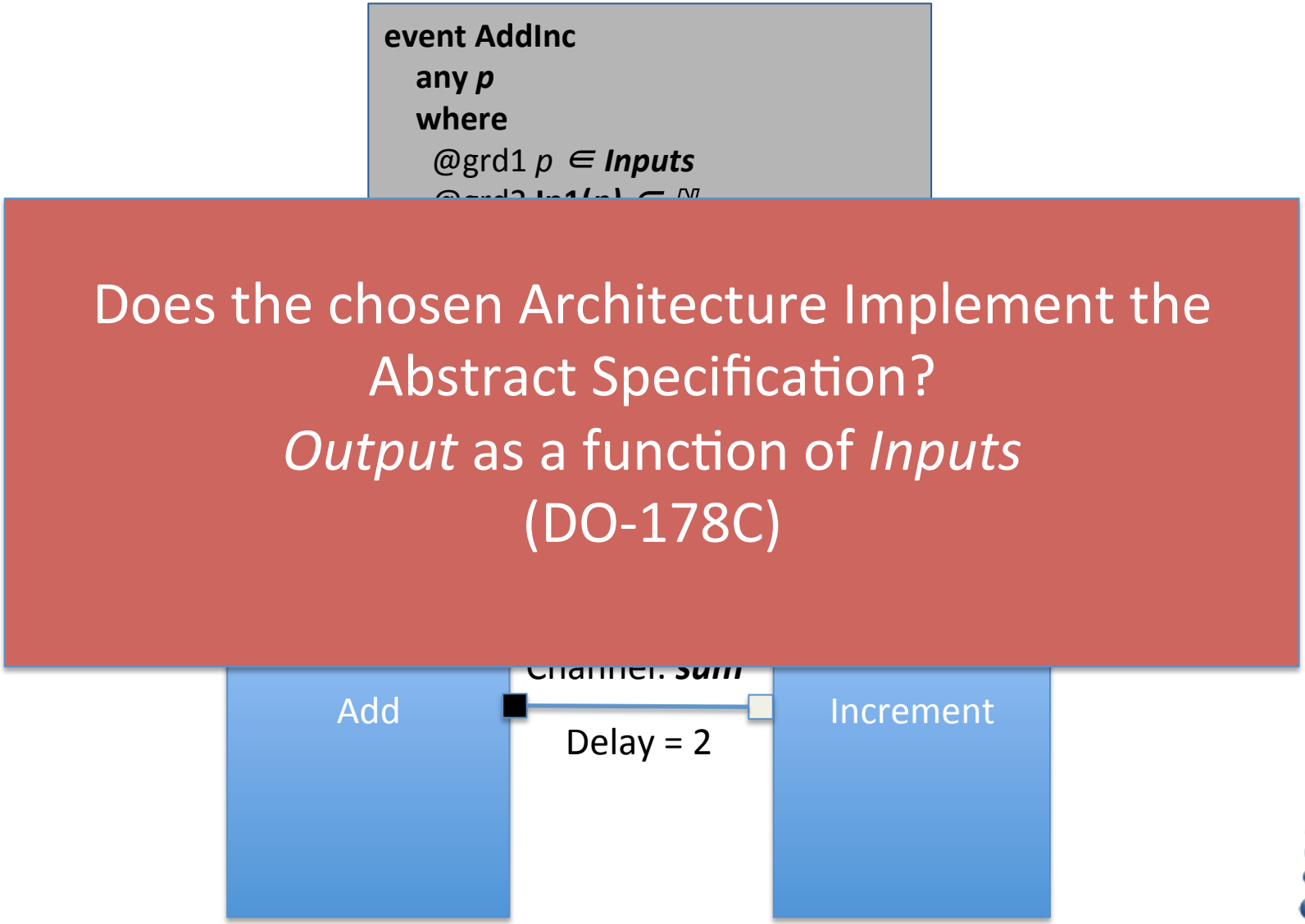
The Implementation Architecture

event AddInc
  any *p*
  where
    @grd1 *p* ∈ ***Inputs***
    @grd2 **In1(*p*)** ∈ ℕ
    @grd3 **In2(*p*)** ∈ ℕ
  then
    @act1 v ≔ **In1(*p*) + *In2(p)* + 1**
end

⊆

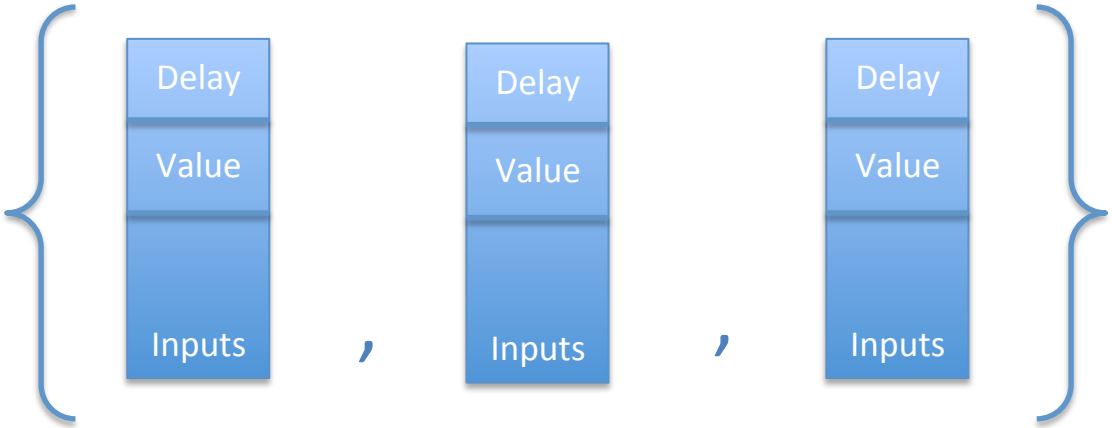


The Implementation Architecture



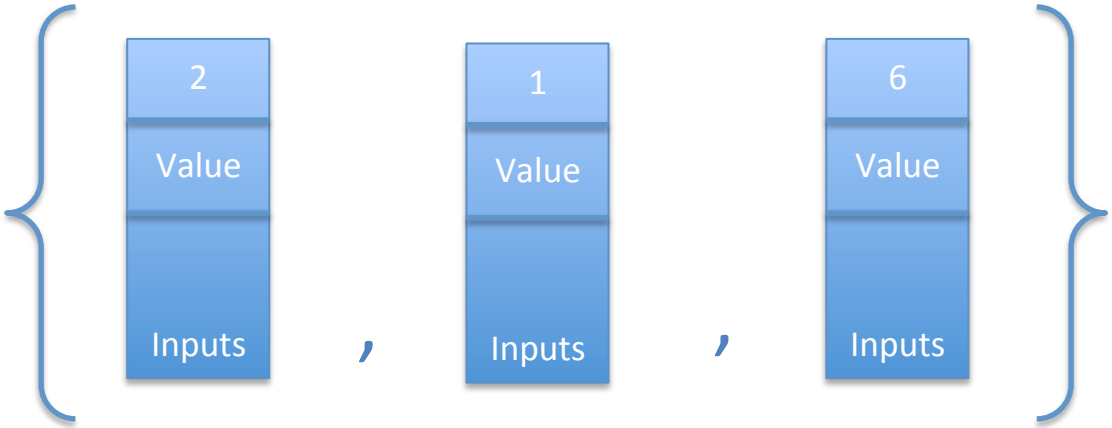
Modelling Channels with Delay in Event-B

- A Channel is a Set of Schedules
- A Schedule comprises
  - a Delay (greater than or equal to 0)
  - a Value (optional)
  - the Input Values that correspond to the Output Value (optional)



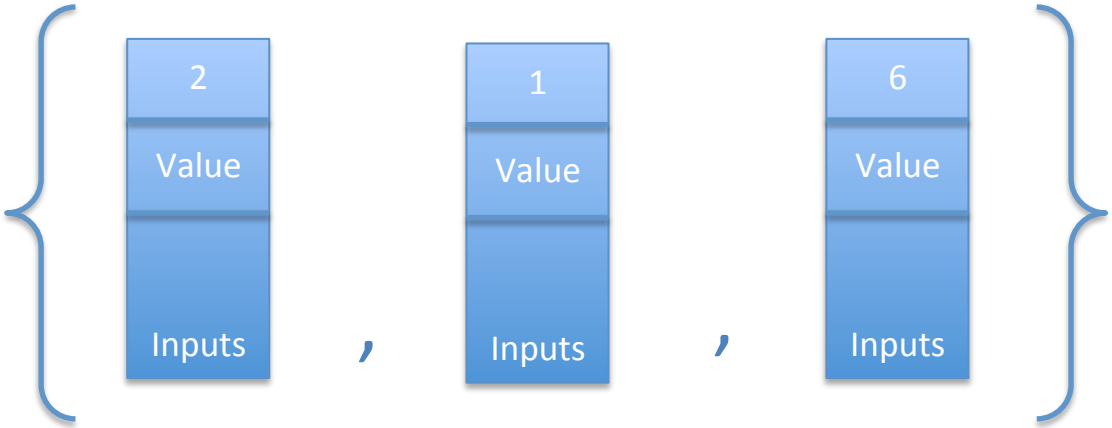
Writing to a Channel

- A Channel write is accomplished by creating a new schedule with a delay of at least one
- Multiple Schedules may be added
  - Prevent multiple schedules for same time *OR*
  - Choose one non-deterministically



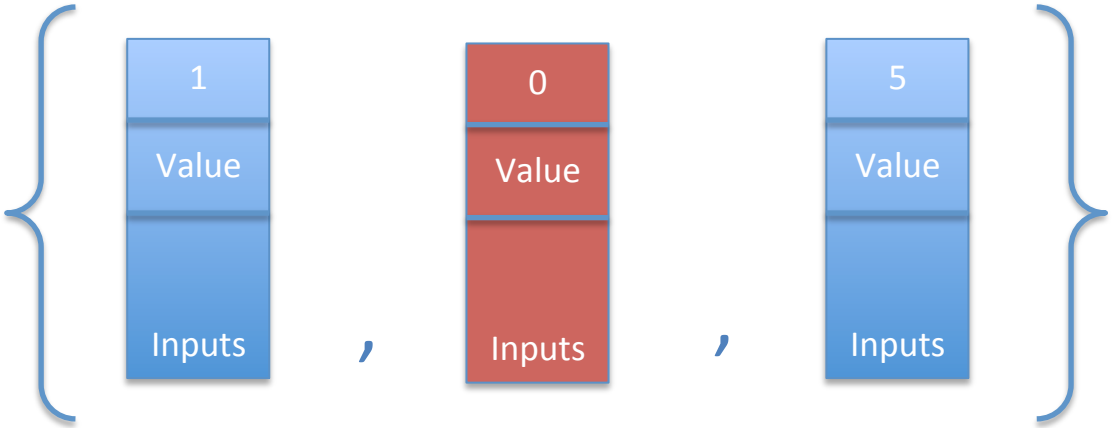
The Update/Evaluate Cycle

- ***Update*** is modeled using a single Event-B event
- ***Evaluate*** is represented by one or more enabled Component Events



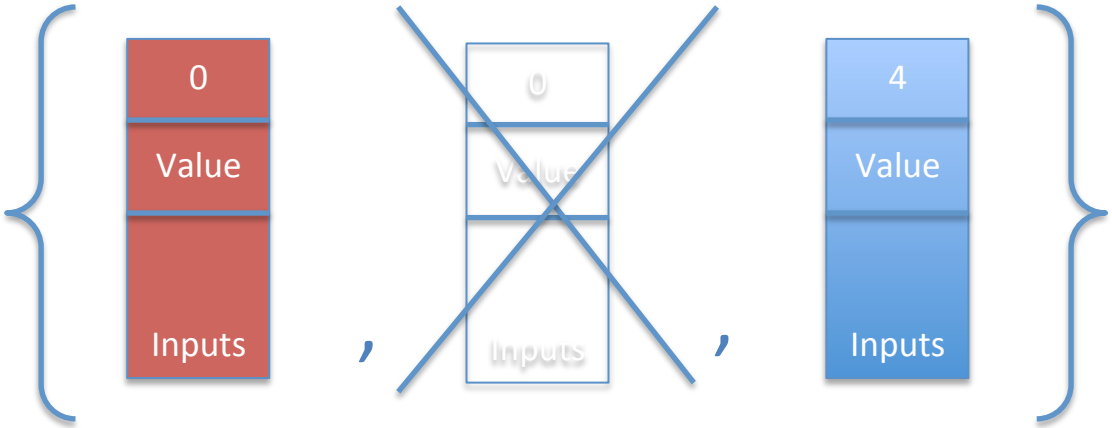
Evaluation Mode

- All Components, where one or more of their *Input* Channels has a schedule with delay 0, *resume* (at least one Component event is enabled and the Update event is disabled)
  - Change local state
  - Create new Schedules on Output Channels
  - *Suspend*



Update Mode

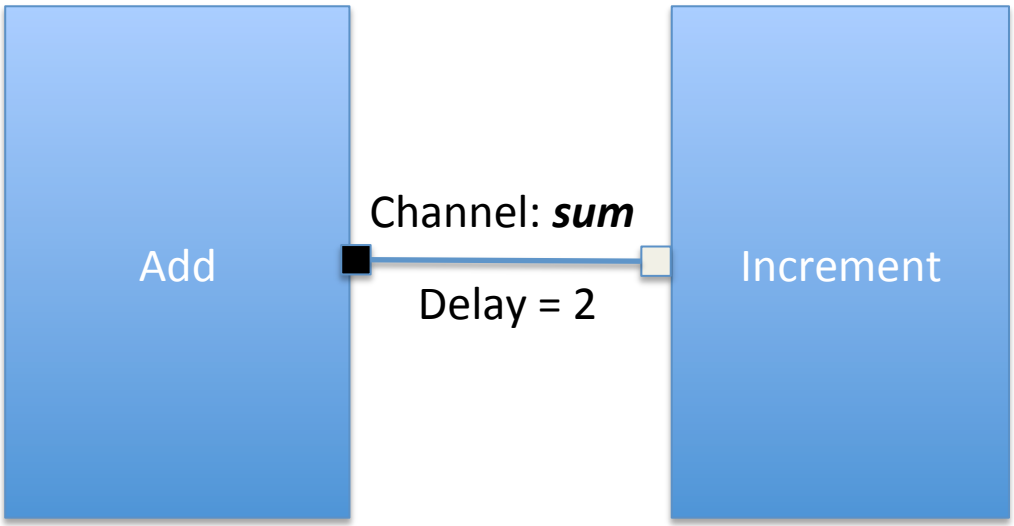
- The Update Event is enabled when *all* the Components have been evaluated
  - Schedules with 0 delay are deleted
  - All other schedule delays are decremented
  - The current *tick* is therefore complete
- The Update Event is re-enabled if no schedule has 0 delay, resulting in another *tick*



Our Example: The First Refinement

Modelling the Channel

@inv3 sum_value ∈ **Schedule** ⇸ ℕ
@inv4 sum_delay ∈ **Schedule** ⇸ ℕ
@inv5 sum_inputs ∈ **Schedule** ⇸ **Inputs**



The Add and Increment Components

event Add
  any ***p s***
  where
    @grd1 *p* ∈ ***Inputs***
    @grd2 **In1(*p*)** ∈ ℕ
    @grd3 **In2(*p*)** ∈ ℕ
    @grd4 *s* ∉ *dom(sum_delay)*
    @grd5 adder_evaluated = FALSE
  then
    @act1 sum_value(*s*) ≔ ***In1(p) + In2(p)***
    @act2 sum_delay(*s*) ≔ *2*
    @act3 sum_inputs(*s*) ≔ *p*
    @act4 adder_evaluated ≔ TRUE
end

event Increment refines AddInc
  any ***s***
  where
    @grd1 *s* ∈ *dom(sum_delay)*
    @grd2 sum_delay(*s*) = *0*
    @grd3 incrementer_evaluated = FALSE
  with
    @p p = sum_inputs(*s*)
  then
    @act1 v ≔ sum_value(*s*) + *1*
    @act2 inc_sum ≔ sum_value(*s*)
    @act3 incrementer_evaluated ≔ TRUE
end

@inv3 sum_value ∈ **Schedule** ⇸ ℕ
@inv4 sum_delay ∈ **Schedule** ⇸ ℕ
@inv5 sum_inputs ∈ **Schedule** ⇸ **Inputs**

The Update Event/Synchronisation

event Update
where
@grd1 adder_evaluated = TRUE
@grd2 $0 \notin ran(sum\_delay) \lor incrementer\_evaluated = TRUE$
then
@act1 adder_evaluated ≔ FALSE
@act2 incrementer_evaluated ≔ FALSE
@act3 $sum\_delay \coloneqq \lambda i \cdot i \in dom(sum\_delay) \land sum\_delay(i) > 0 \mid sum\_delay(i) - 1$
@act4 $sum\_value \coloneqq \lambda i \cdot i \in dom(sum\_value) \land i \in dom(sum\_delay) \land sum\_delay(i) > 0 \mid sum\_value(i)$
@act5 $sum\_inputs \coloneqq \lambda i \cdot i \in dom(sum\_inputs) \land i \in dom(sum\_delay) \land sum\_delay(i) > 0 \mid sum\_inputs(i)$
end

event Add
any ***p s***
where
@grd1 $p \in$ ***Inputs***
@grd2 **In1(*p*)** $\in \mathbb{N}$
@grd3 **In2(*p*)** $\in \mathbb{N}$
@grd4 $s \notin dom(sum\_delay)$
@grd5 adder_evaluated = FALSE
then
@act1 sum_value(*s*) ≔ ***In1(p) + In2(p)***
@act2 sum_delay(*s*) ≔ *2*
@act3 sum_inputs(*s*) ≔ *p*
@act4 adder_evaluated ≔ TRUE
end

event Increment refines AddInc
any ***s***
where
@grd1 $s \in dom(sum\_delay)$
@grd2 sum_delay(*s*) = *0*
@grd3 incrementer_evaluated = FALSE
with
@p p = sum_inputs(*s*)
then
@act1 v ≔ sum_value(*s*) + *1*
@act2 inc_sum ≔ sum_value(*s*)
@act3 incrementer_evaluated ≔ TRUE
end

The Gluing Invariant

Represented as the function of Inputs to Output as preserved in the Schedules

@inv9 $\forall s \cdot s \in dom(sum\_value) \Rightarrow sum\_value(s) =$ ***In1(sum_inputs(s)) + In2(sum_inputs(s))***

The Second Refinement – Remove Inputs

event Update
  where
    @grd1 adder_evaluated = TRUE
    @grd2 $0 \notin ran(sum\_delay) \lor incrementer\_evaluated = TRUE$
  then
    @act1 adder_evaluated ≔ FALSE
    @act2 incrementer_evaluated ≔ FALSE
    @act3 $sum\_delay \coloneqq \lambda i \cdot i \in dom(sum\_delay) \land sum\_delay(i) > 0 \mid sum\_delay(i) - 1$
    @act4 $sum\_value \coloneqq \lambda i \cdot i \in dom(sum\_value) \land i \in dom(sum\_delay) \land sum\_delay(i) > 0 \mid sum\_value(i)$
end

event Add
  any ***p s***
  where
    @grd1 $p \in$ ***Inputs***
    @grd2 $\mathbf{In1}(p) \in \mathbb{N}$
    @grd3 $\mathbf{In2}(p) \in \mathbb{N}$
    @grd4 $s \notin dom(sum\_delay)$
    @grd5 adder_evaluated = FALSE
  then
    @act1 $sum\_value(s) \coloneqq \mathbf{In1}(p) + \mathbf{In2}(p)$
    @act2 $sum\_delay(s) \coloneqq 2$
    @act4 adder_evaluated ≔ TRUE
end

event Increment refines AddInc
  any ***s***
  where
    @grd1 $s \in dom(sum\_delay)$
    @grd2 $sum\_delay(s) = 0$
    @grd3 incrementer_evaluated = FALSE
  then
    @act1 $v \coloneqq sum\_value(s) + 1$
    @act2 $inc\_sum \coloneqq sum\_value(s)$
    @act3 incrementer_evaluated ≔ TRUE
end

Cycle-Based Simulation- Represent Schedules as *Pairs* of Variables

event Update refines Update
where
@grd1 adder_evaluated = TRUE
@grd2 msg_rcvd_on_sum = FALSE $\vee$ incrementer_evaluated = TRUE
then
@act1 msg_rcvd_on_sum $\coloneqq$ msg_sent_on_sum
@act2 sum $\coloneqq$ sum_prime
@act3 msg_sent_on_sum $\coloneqq$ FALSE
@act4 adder_evaluated $\coloneqq$ FALSE
@act5 incrementer_evaluated $\coloneqq$ FALSE
end

event Add refines Add
any *p*
where
@grd1 $p \in$ ***Inputs***
@grd2 **In1(*p*)** $\in \mathbb{N}$
@grd3 **In2(*p*)** $\in \mathbb{N}$
@grd4 adder_evaluated = FALSE
then
@act1 sum_prime $\coloneqq$ **In1(*p*) + *In2(p)***
@act2 msg_sent_on_sum $\coloneqq$ TRUE
@act3 adder_evaluated $\coloneqq$ TRUE
end

event Increment refines Increment
where
@grd1 msg_rcvd_on_sum = TRUE
@grd2 incrementer_evaluated = FALSE
then
@act1 v $\coloneqq$ sum + 1
@act2 incrementer_evaluated $\coloneqq$ TRUE
end

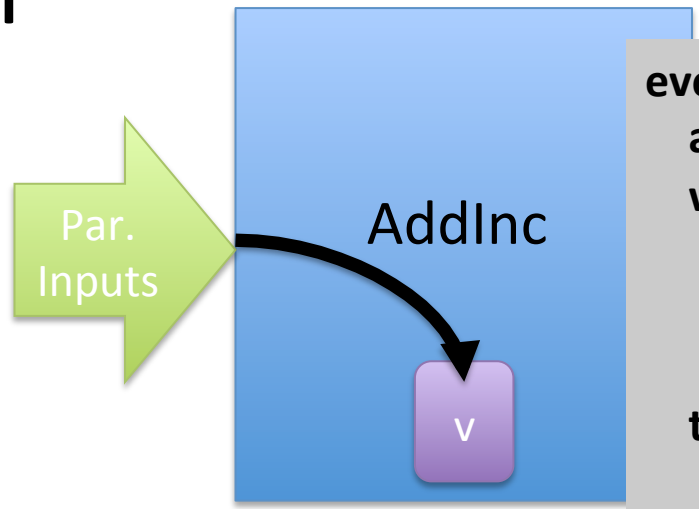
Cycle-Based Simulation- A *Pair* of Gluing Invariants

@inv9 msg_rcvd_on_sum = TRUE ⇒ sum = **In1(sum_inputs) + In2(sum_inputs)**
@inv10 msg_sent_on_sum = TRUE ⇒ sum_prime = **In1(sum_inputs_prime) + In2(sum_inputs_prime)**

Animating and Model Checking Event-B models with ProB

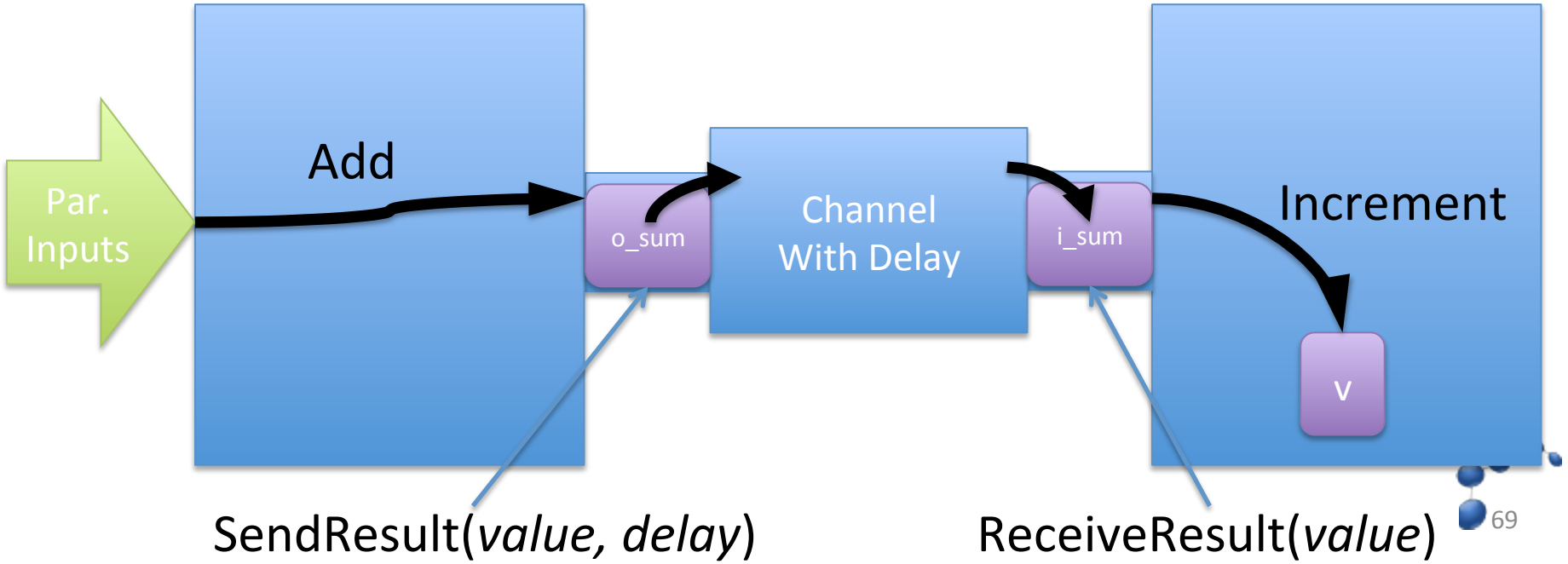
DEMONSTRATION

Architectural Refinement




```
event AddInc
  any p
  where
    @grd1 p ∈ Inputs
    @grd2 In1(p) ∈ ℕ
    @grd3 In2(p) ∈ ℕ
  then
    @act1 v ≔ In1(p) + In2(p) + 1
end
```

⊆



Cyber-physical Modeling Summary

- Update/Evaluate Simulation semantics can be formalised in Event-B
- Event-B component models can be simulated/co-simulated with third party simulators
- Event-B refinement supports naturally the DO-178C requirement to verify the relationship between Inputs and Outputs at Specification and Implementation Level
- Update/Evaluate modes provides a suitable basis for a Formal Safety Analysis

Assertion-based Verification

- Identify Assertions at the Specification Level
  - Event-B Invariants
    - Abstract Level
    - Concrete Level
- Translate Invariants to
  - PSL
  - SVA
- Translate Synthesised Assertions to Event-B
  - Formal Proof
  - Model Checking
- Assertion Coverage
  - Trace back to Requirements

Summary

- Background to Event-B
- Event-B in the Design/Verification Flow
- Complex hardware specification/verification
  - Pipelines
  - Elastic Buffering
- Embedded system specification/verification
  - Temporal Modeling in Cyber-physical systems
  - Animating and Model Checking Event-B models
- Assertion-based verification
  - Deriving assertions from the specification