On Proving with Event-B that a Pipelined Processor Model Implements its ISA Specification

John Colley
Dependable Systems and Software Engineering

July 2009

Supervisor
Michael Butler

UNIVERSITY OF
Southampton
School of Electronics
and Computer Science

Introduction

- System-on-Chip (SoC) Microprocessors
- Motivation
- Instruction Set Architecture (ISA) Specification
- Arithmetic Instruction Specification in Event-B
- Deriving a Pipelined Implementation with Refinement
- Summary and Future Work

System-on-Chip (SoC) Microprocessors

- Typically 5-stage pipeline RISC
- Based on DLX architecture
- “Small is Beautiful” Kurt Keutzer, UCB, 2008
  - Silicon Constraints
    - Interconnect
    - Power and Energy
    - Variability
    - Reliability
    - *Verifiability*
- Mobile Applications – ARM, MIPS

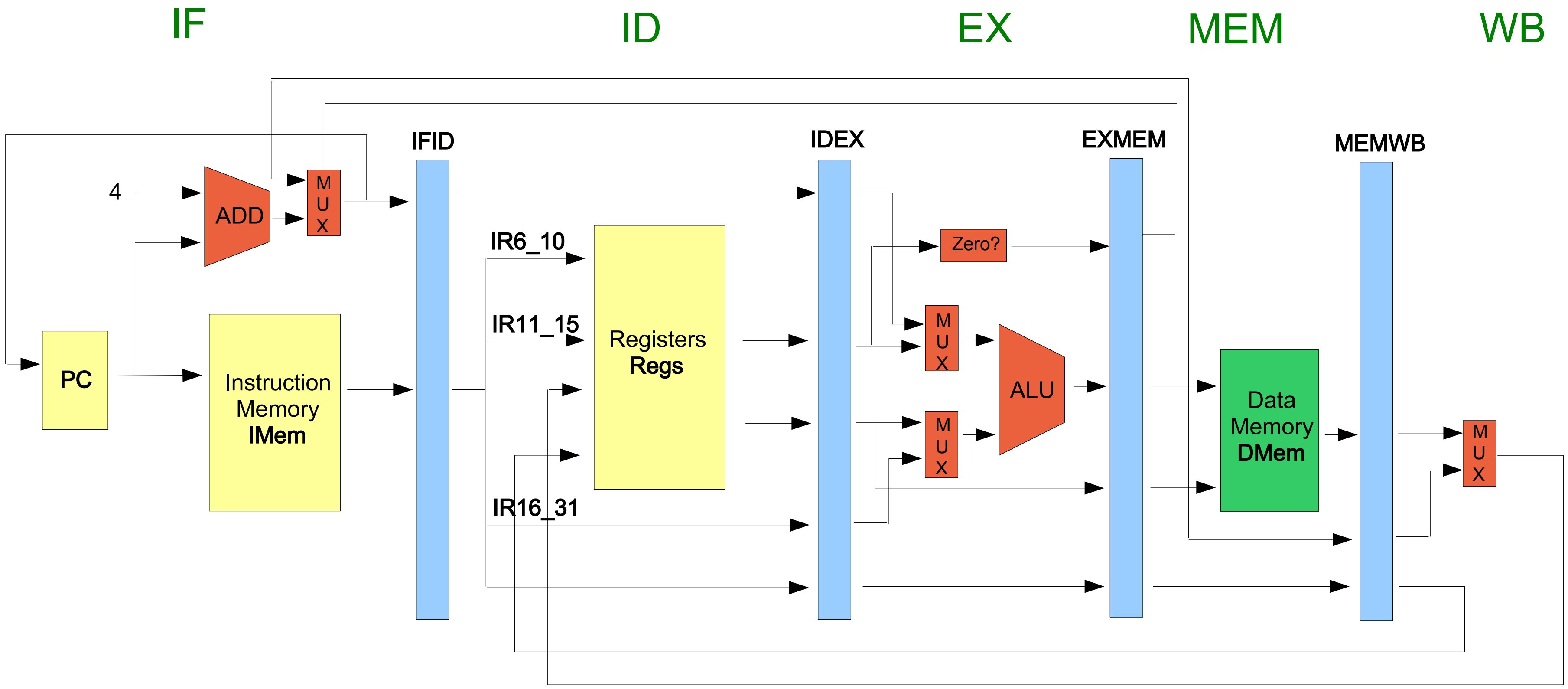
Motivation

- 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, CAL
    - high-level synthesis to RTL
    - based on Guarded Atomic Actions
- ***But***, verification is still
    - performed on low-level, RTL description
    - predominantly test-based

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, CAL
- Incorporate Proof-based techniques into the established SoC Verification Flow

5-stage RISC SoC Processor



Generic Operations

Load
Store
Branch
ArithRR
ArithImm

Pipeline Stages

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

🟦 - pipeline register

UNIVERSITY OF
Southampton
School of Electronics
and Computer Science

Microprocessor Specification: Term Rewriting Systems

†Defined as a tuple (S, R, S_0) where

- S is a set of terms
- R is a set of re-writing rules
- S_0 is a set of initial terms, S_0 S

States: represented by TRS terms
Transitions: represented by TRS rules:-

s1 if *p*(*s1*)
s2

where *s1* and *s2* are terms and *p* is a predicate

Example : Microprocessor Op rule

Proc(pc, *rf, im)*   if *im*[*pc*] = *Rr* := Op(*Ra, Rb*)
Proc(pc + 1, *rf* [*Rr* := *v*], *im)* where *v* := Op(*rf* [*Ra*], rf[*Rb*])

† Using Term Rewriting Systems to Design and Verify Processors
Arvind and Shen, IEEE MICRO 1999

UNIVERSITY OF
Southampton
School of Electronics
and Computer Science

Microprocessor Specification:
Term Rewriting Systems

†Defined as a tuple (S, R, S_0) where

- S is a set of terms
- R is a set of re-writing rules
- S_0 is a set of initial terms S_0 S

States: represented by TRS terms
Transitions: represented by TRS rules:-

s1 if *p*(*s1*)
s2

where *s1* and *s2* are terms and *p* is a predicate

Operation specified as a transformation on the processor registers

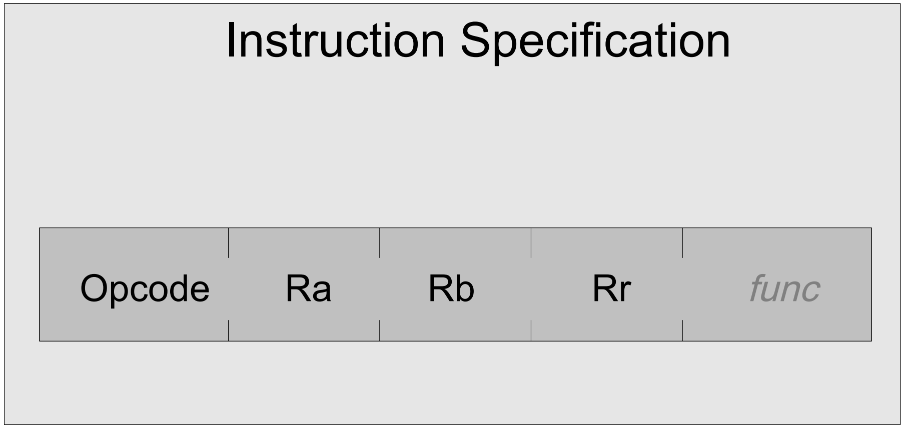
Example : Microprocessor Op rule

Proc(pc, *rf, im)* if *im*[*pc*] = *Rr* := Op(*Ra*, *Rb*)
Proc(pc + 1, ***rf* [*Rr* := v]**, *im)* where *v* := Op(*rf* [*Ra*], rf[*Rb*])

[†] Using Term Rewriting Systems to Design and Verify Processors
Arvind and Shen, IEEE MICRO 1999

UNIVERSITY OF
Southampton
School of Electronics
and Computer Science

Abstract Context: Arithmetic Instruction



Instruction Specification

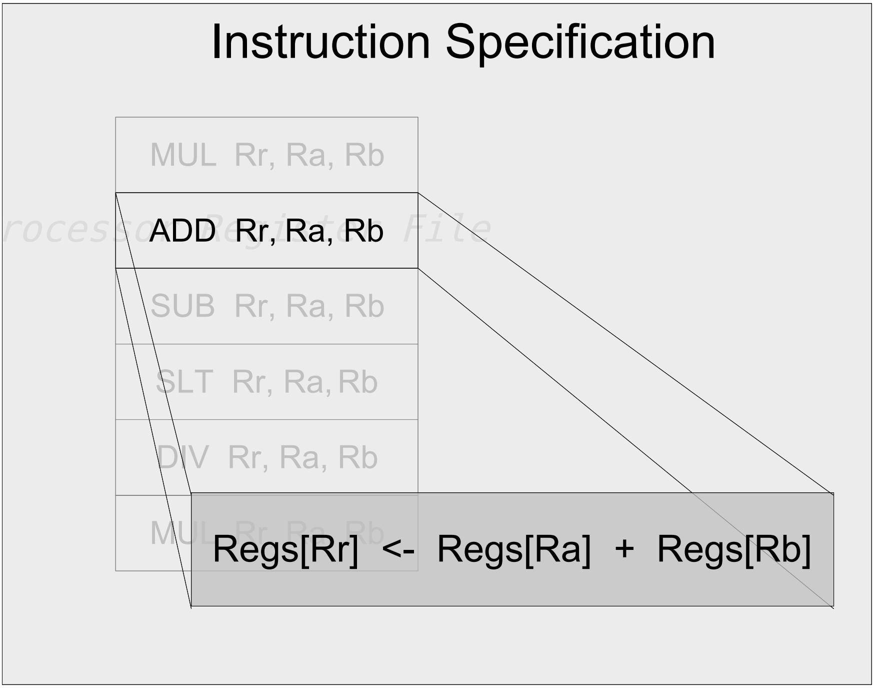
Opcode	Ra	Rb	Rr	*func*

```
context PIPEC

constants Register Rr Ra Rb NOP 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 Operations
  @axm6 NOP ∈ Opcode // No Operation
  @axm7 NOP ∉ ArithRROp
end
```

† A proposal for records in Event-B
Evans and Butler, Lecture Notes in Computer Science, 2006

Abstract Machine: Arithmetic Instruction

```
machine PIPEM sees PIPEC

variables Regs WBop

invariants

  @inv1 Regs ∈ Register ⟶ ℤ   // The Processor Register File

    .
    .

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

       .
  end
end
```



Instruction Specification

MUL Rr, Ra, Rb

ADD Rr, Ra, Rb

SUB Rr, Ra, Rb

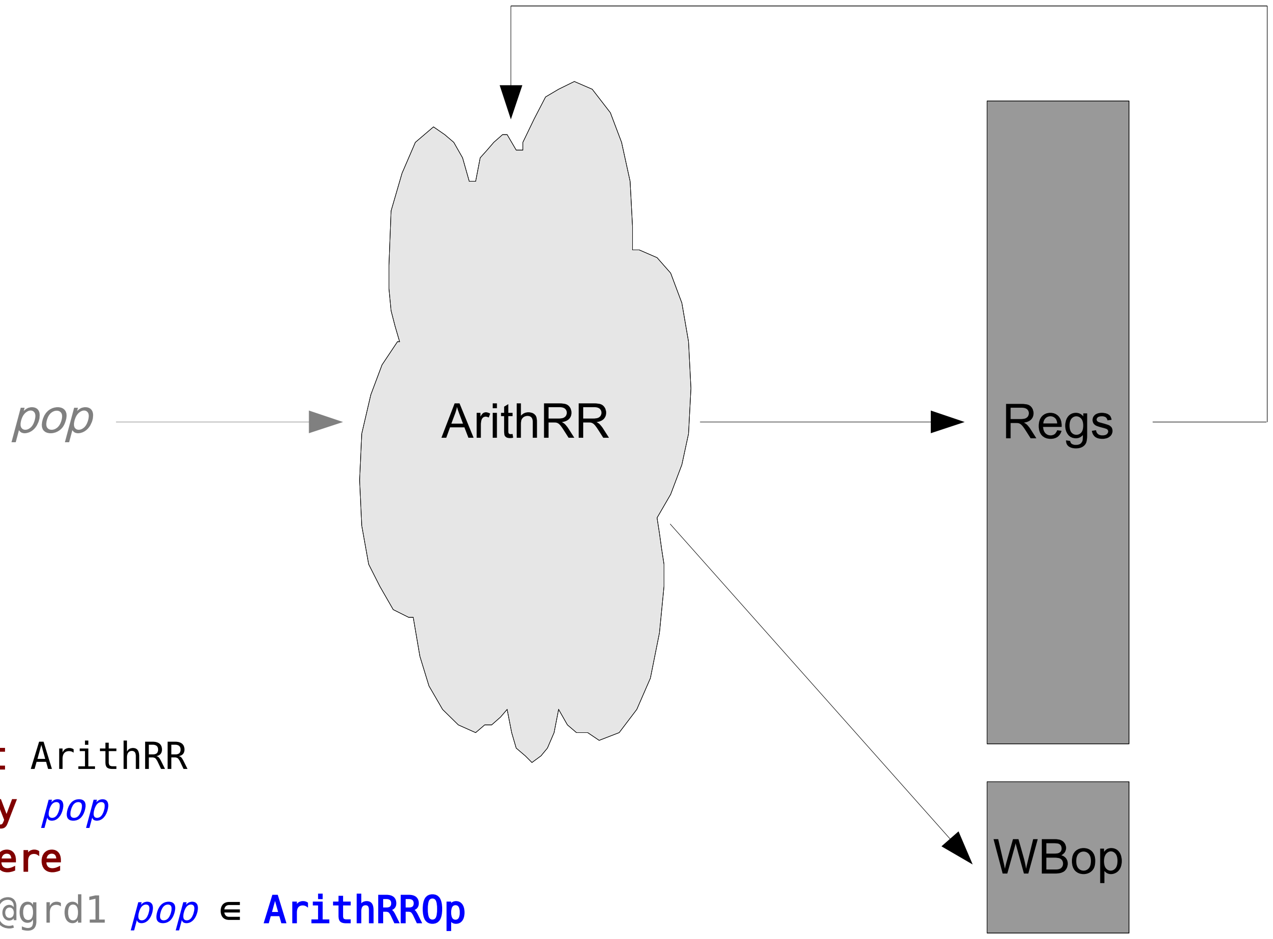
SLT Rr, Ra, Rb

DIV Rr, Ra, Rb

MUL Rr, Ra, Rb

Regs[Rr] <- Regs[Ra] + Regs[Rb]

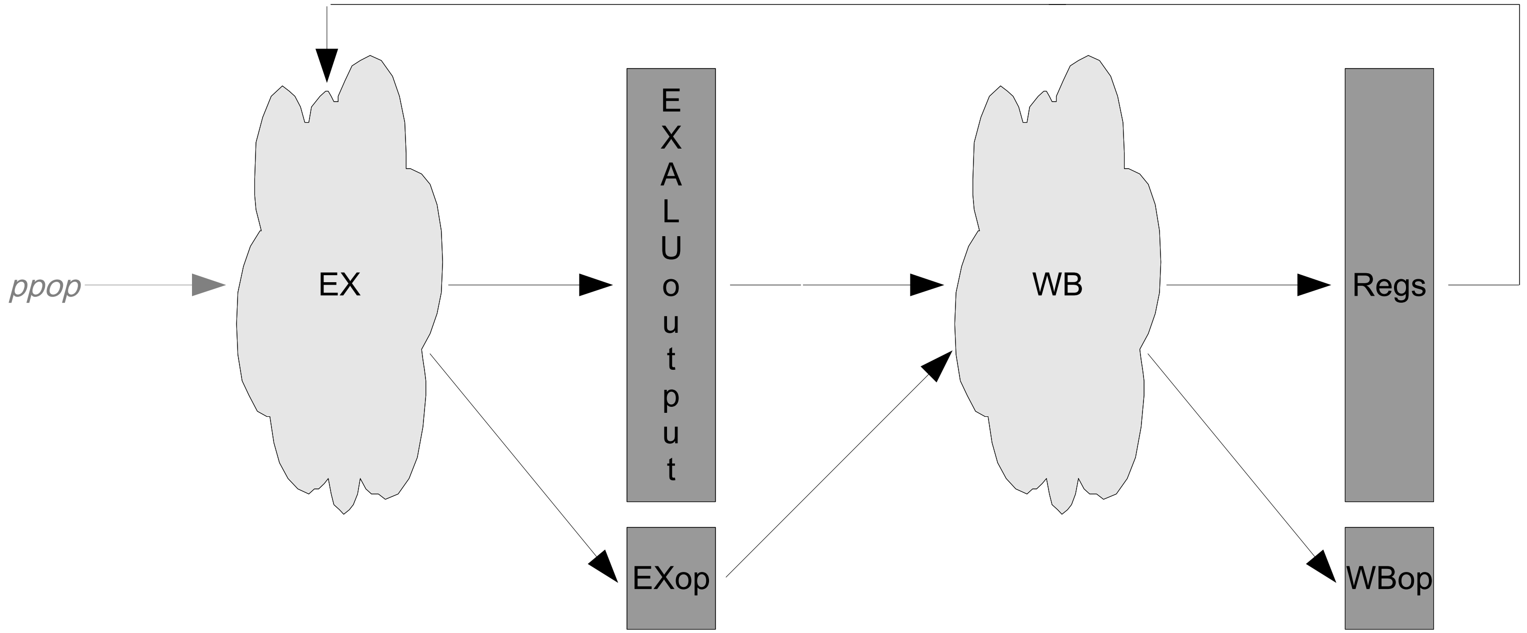
Abstract Machine: Microarchitecture



```
event ArithRR
   any pop
   where
     @grd1 pop ∈ ArithRROp
   then
     @act1 Regs(Rr(pop)) ≔ Regs(Ra(pop)) + Regs(Rb(pop))
     @act2 WBop ≔ pop
 end
```

UNIVERSITY OF Southampton
School of Electronics
and Computer Science

Refinement: 2-stage pipeline (EXecute and WriteBack)

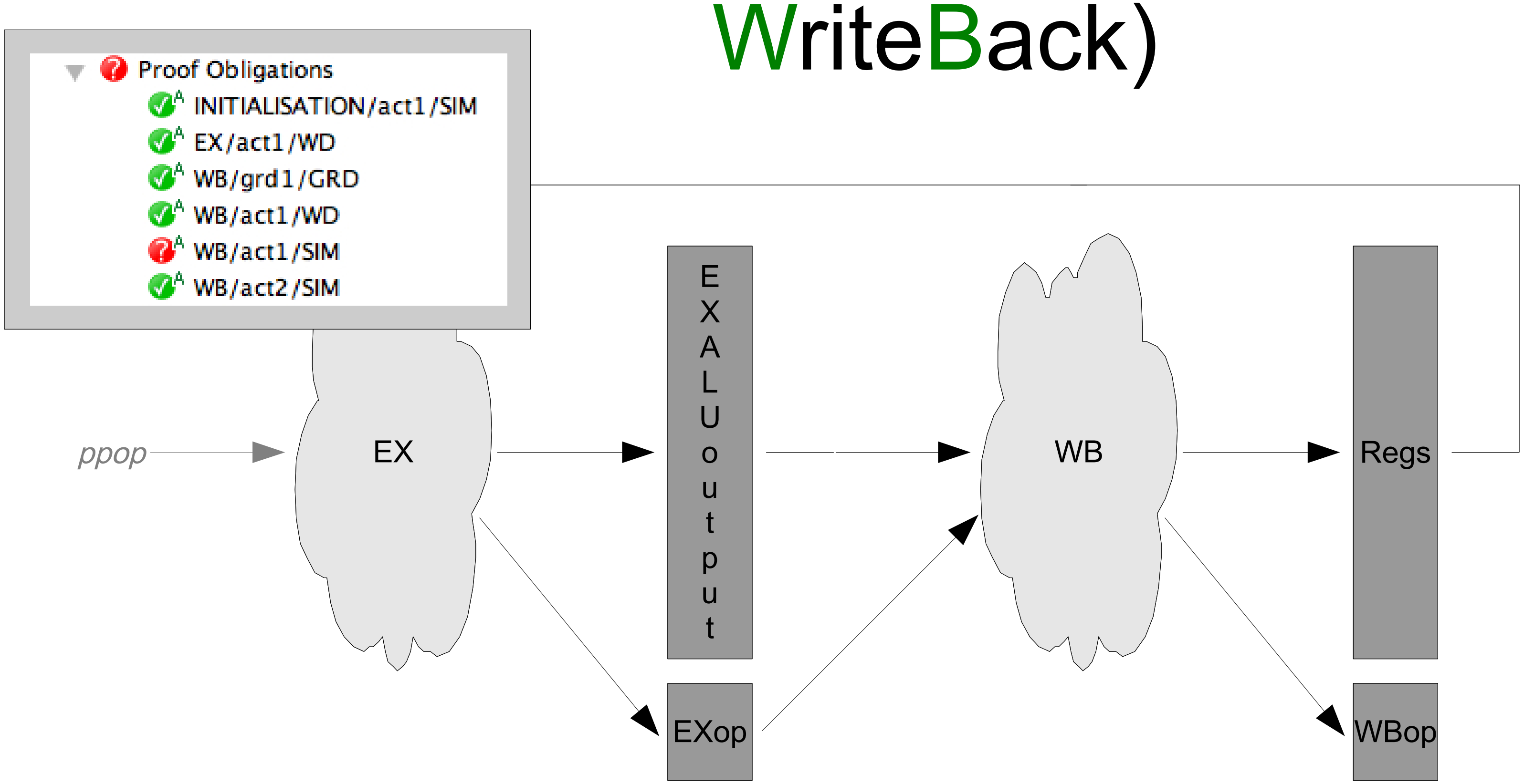


```
event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

```
event WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

UNIVERSITY OF Southampton
School of Electronics
and Computer Science

Refinement: 2-stage pipeline (EXecute and WriteBack)



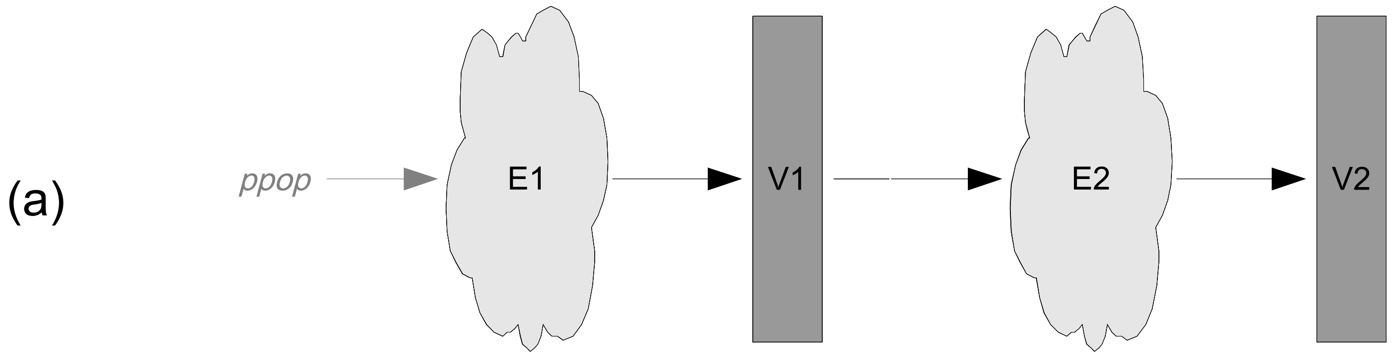
- ? Proof Obligations
  - ✓ INITIALISATION/act1/SIM
  - ✓ EX/act1/WD
  - ✓ WB/grd1/GRD
  - ✓ WB/act1/WD
  - ? WB/act1/SIM
  - ✓ WB/act2/SIM

```
event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

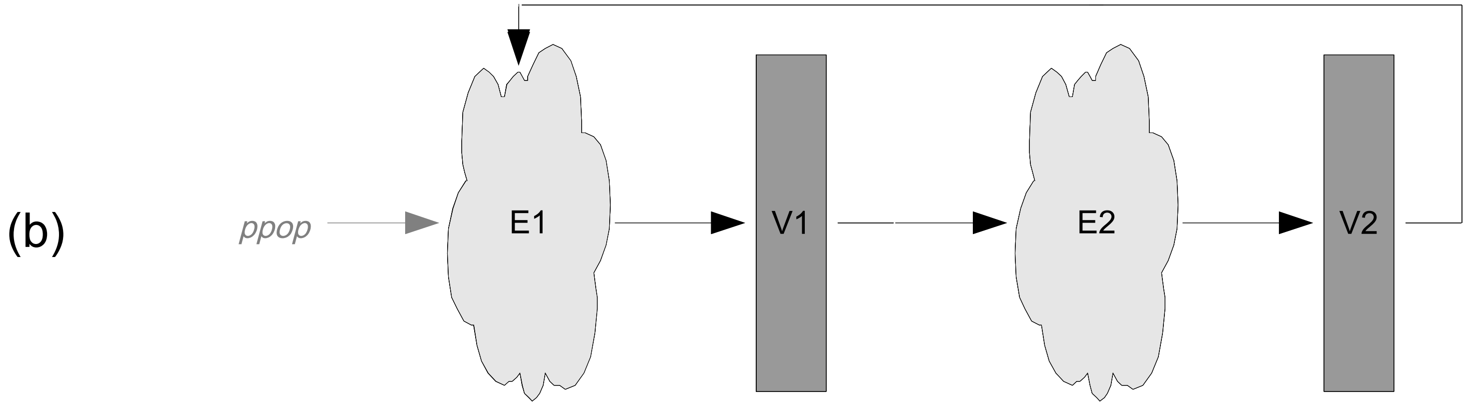
```
event WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

UNIVERSITY OF Southampton
School of Electronics
and Computer Science

Pipeline Feedback and Interleaving



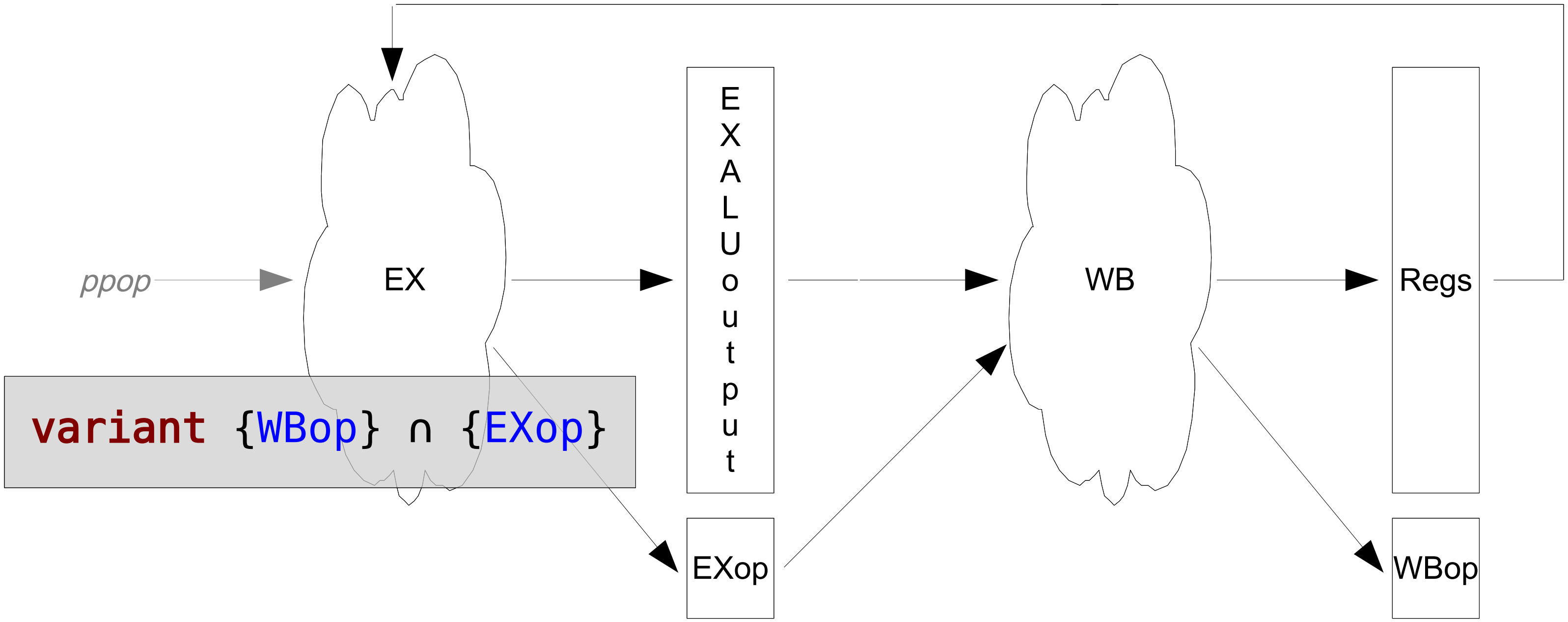
E2 followed by *E1 (E2;E1)* is equivalent to *E1 || E2*



There is *NO* Interleaving that represents *E1 || E2*

UNIVERSITY OF Southampton
School of Electronics
and Computer Science

Consider *Sequential* Execution[†]



variant {WBop} ∩ {EXop}

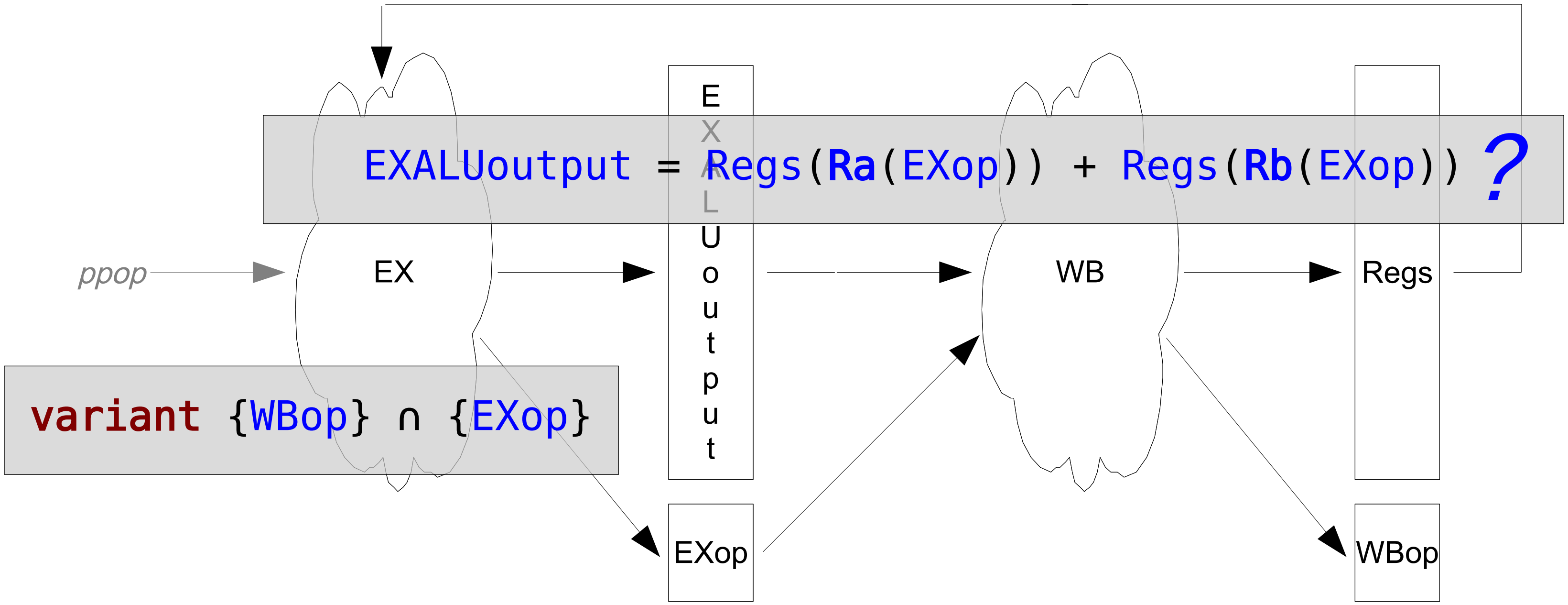
```
convergent event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 ppop ≠ EXop
    @grd3 WBop = EXop
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

```
event WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 WBop ≠ EXop
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

[†] Computer Architecture: Complexity and Correctness
Müller and Paul, Springer, 2000

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Sequential Execution: Discovering the Invariant



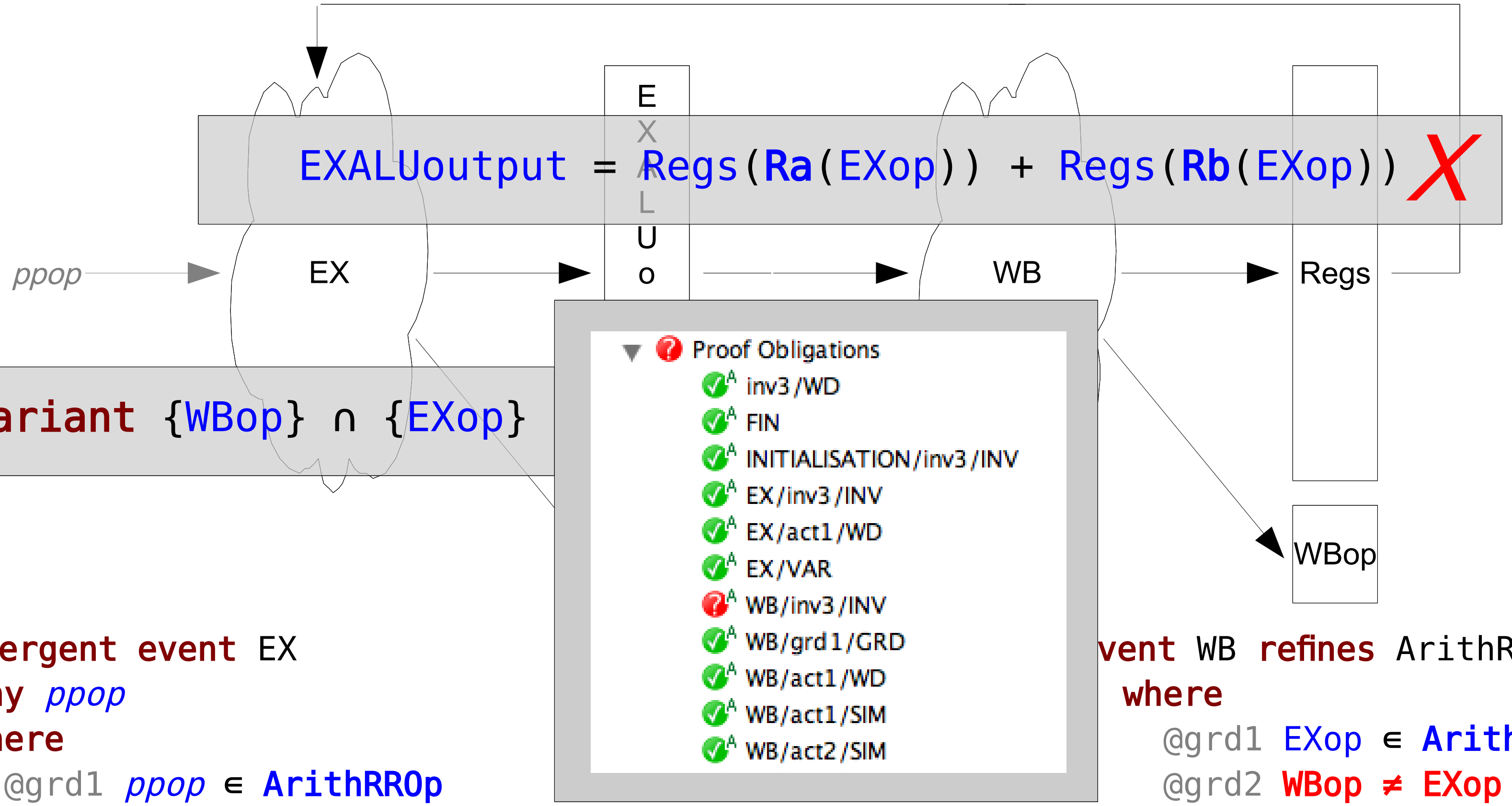
```
EXALUoutput = Regs(Ra(EXop)) + Regs(Rb(EXop)) ?
```

```
variant {WBop} ∩ {EXop}
```

```
convergent event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 ppop ≠ EXop
    @grd3 WBop = EXop
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

```
event WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 WBop ≠ EXop
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

Sequential Execution: Discovering the Invariant



EXALUoutput = Regs(**Ra**(EXop)) + Regs(**Rb**(EXop)) ✗

variant {WBop} ∩ {EXop}

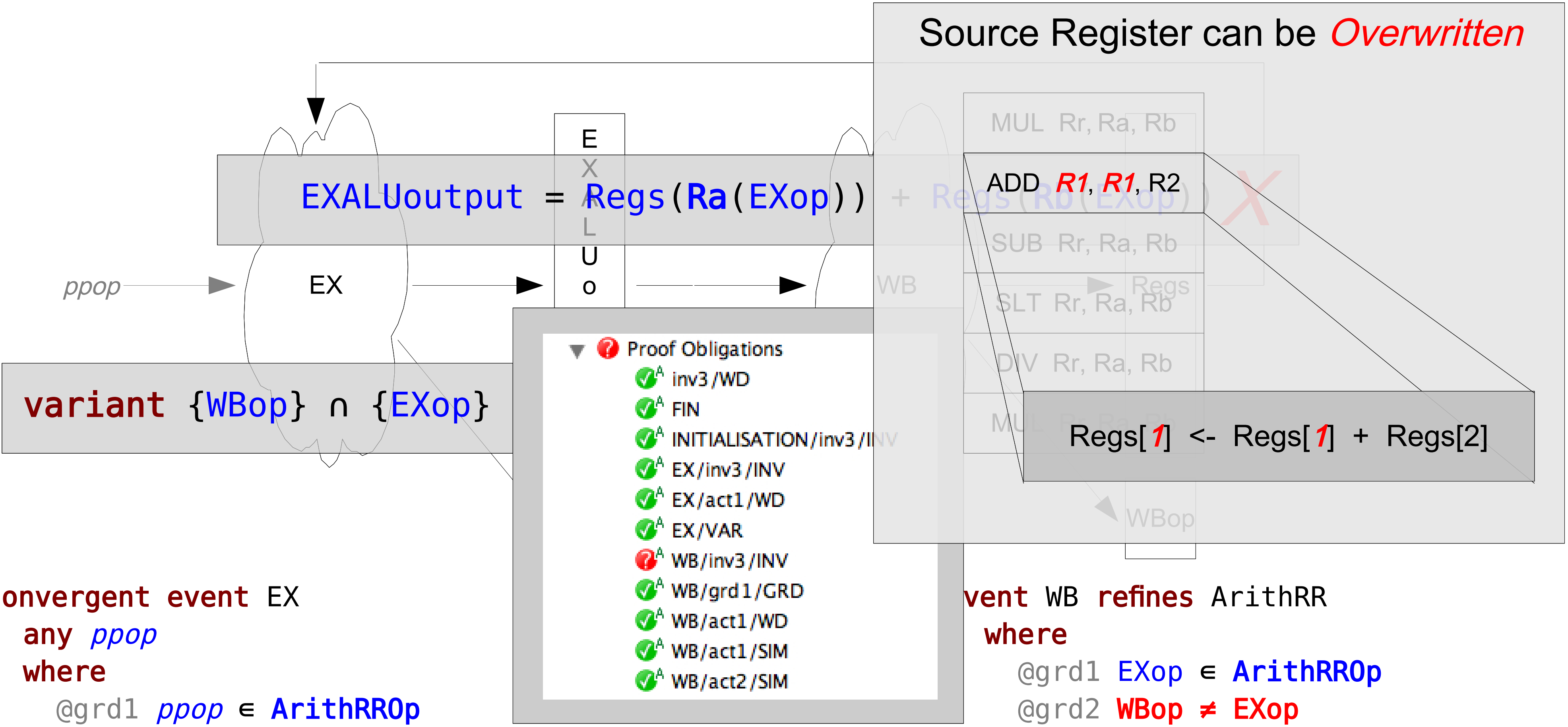
- ? Proof Obligations
  - ✓ inv3/WD
  - ✓ FIN
  - ✓ INITIALISATION/inv3/INV
  - ✓ EX/inv3/INV
  - ✓ EX/act1/WD
  - ✓ EX/VAR
  - ? WB/inv3/INV
  - ✓ WB/grd1/GRD
  - ✓ WB/act1/WD
  - ✓ WB/act1/SIM
  - ✓ WB/act2/SIM

```
convergent event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 ppop ≠ EXop
    @grd3 WBop = EXop
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

```
vent WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 WBop ≠ EXop
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

UNIVERSITY OF Southampton
School of Electronics
and Computer Science

Sequential Execution: Discovering the Invariant



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

```
variant {WBop} ∩ {EXop}
```

Proof Obligations
- ✓ inv3/WD
- ✓ FIN
- ✓ INITIALISATION/inv3/INV
- ✓ EX/inv3/INV
- ✓ EX/act1/WD
- ✓ EX/VAR
- ? WB/inv3/INV
- ✓ WB/grd1/GRD
- ✓ WB/act1/WD
- ✓ WB/act1/SIM
- ✓ WB/act2/SIM

Source Register can be *Overwritten*

MUL Rr, Ra, Rb
ADD *R1*, *R1*, R2
SUB Rr, Ra, Rb
SLT Rr, Ra, Rb
DIV Rr, Ra, Rb
MUL Rr, Ra, Rb

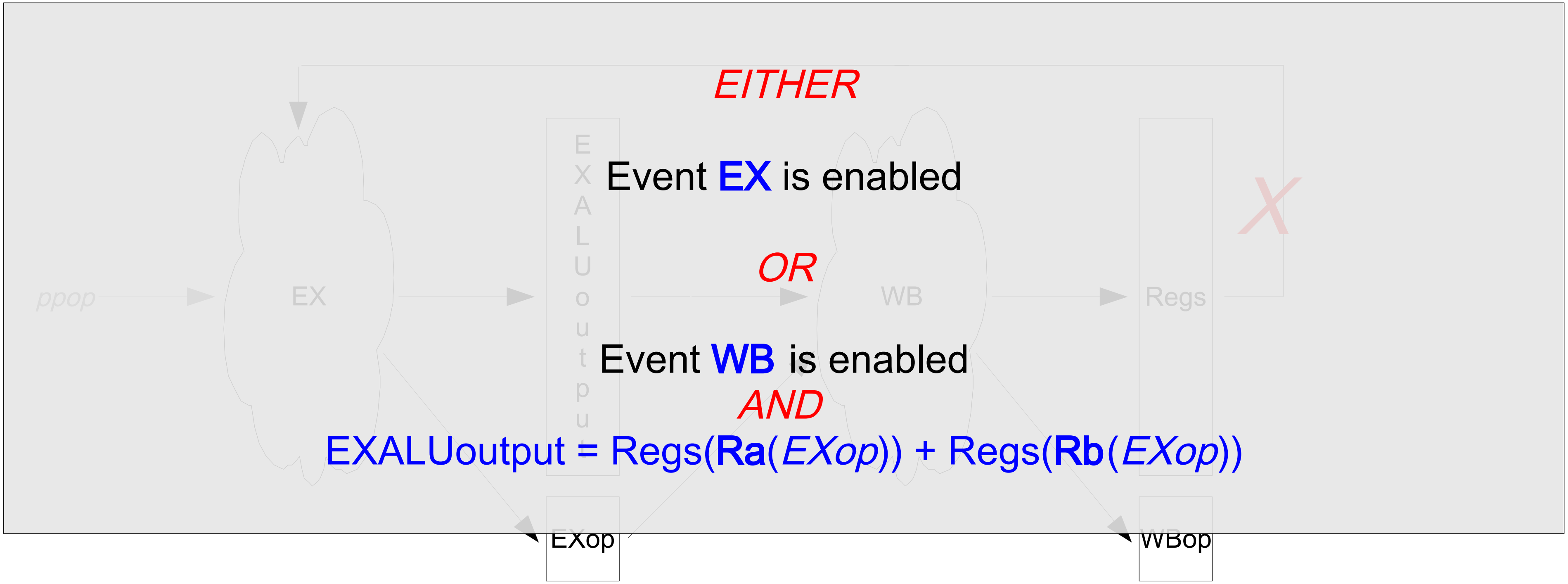
Regs[*1*] <- Regs[*1*] + Regs[2]

```
convergent event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 ppop ≠ EXop
    @grd3 WBop = EXop
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

```
vent WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 WBop ≠ EXop
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

UNIVERSITY OF Southampton
School of Electronics
and Computer Science

Sequential Execution: Discovering the Invariant



EITHER

Event **EX** is enabled

OR

Event **WB** is enabled

AND

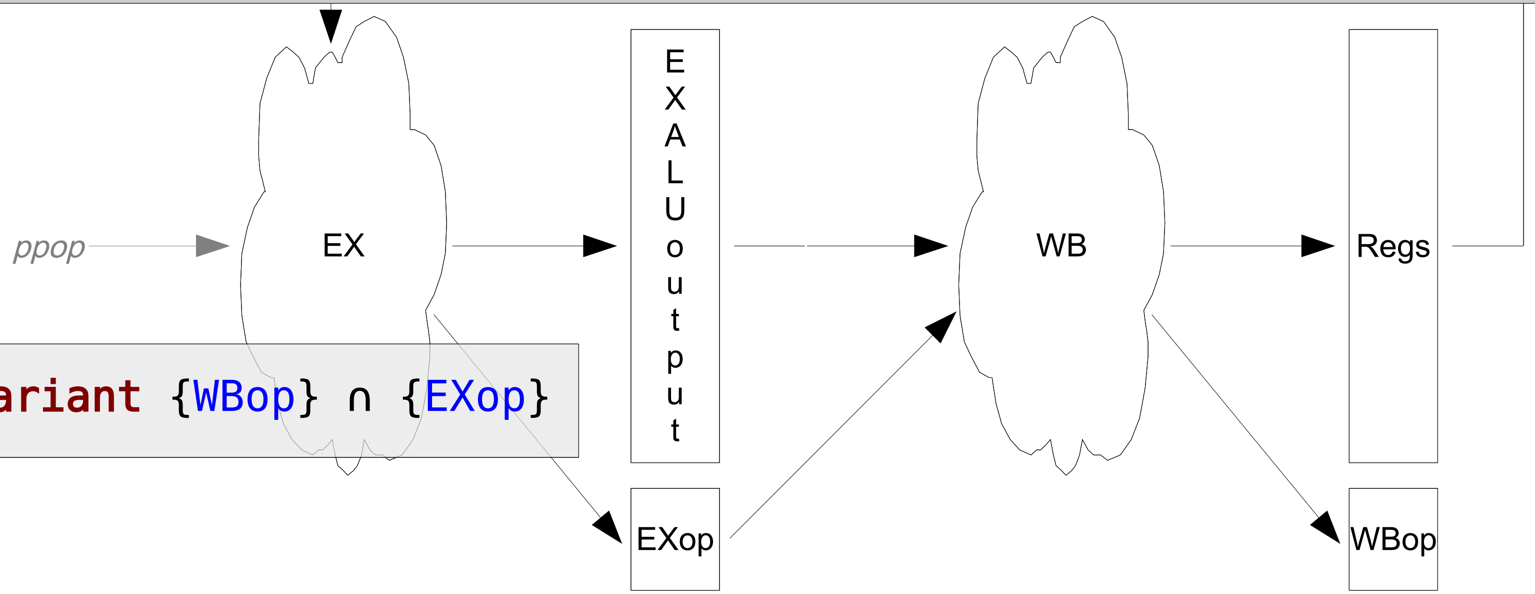
EXALUoutput = Regs(**Ra**(*EXop*)) + Regs(**Rb**(*EXop*))

```
convergent event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 ppop ≠ EXop
    @grd3 WBop = EXop
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

```
event WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 WBop ≠ EXop
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

Sequential Execution: Discovering the Invariant

```
WBop = EXop ∨ (WBop ≠ EXop ∧ EXALUoutput = Regs(Ra(EXop)) + Regs(Rb(EXop)))
```



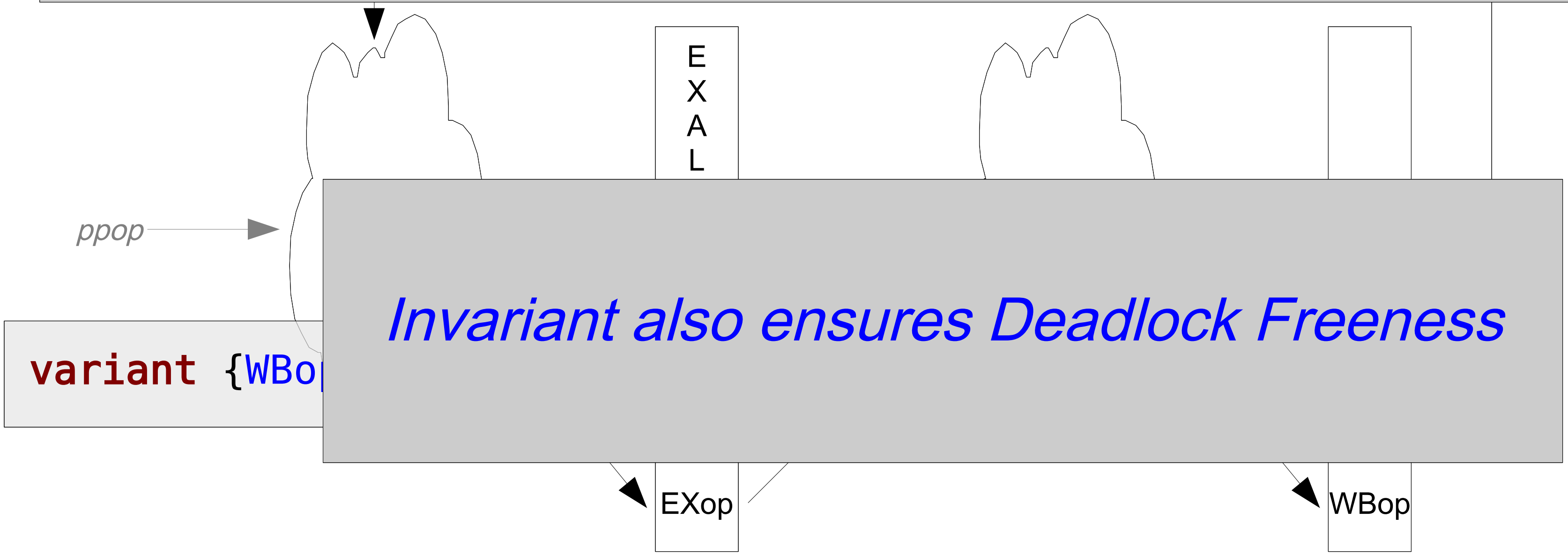
```
variant {WBop} ∩ {EXop}
```

```
convergent event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 ppop ≠ EXop
    @grd3 WBop = EXop
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

```
event WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 WBop ≠ EXop
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

Sequential Execution: Discovering the Invariant

```
WBop = EXop ∨ (WBop ≠ EXop ∧ EXALUoutput = Regs(Ra(EXop)) + Regs(Rb(EXop)))
```



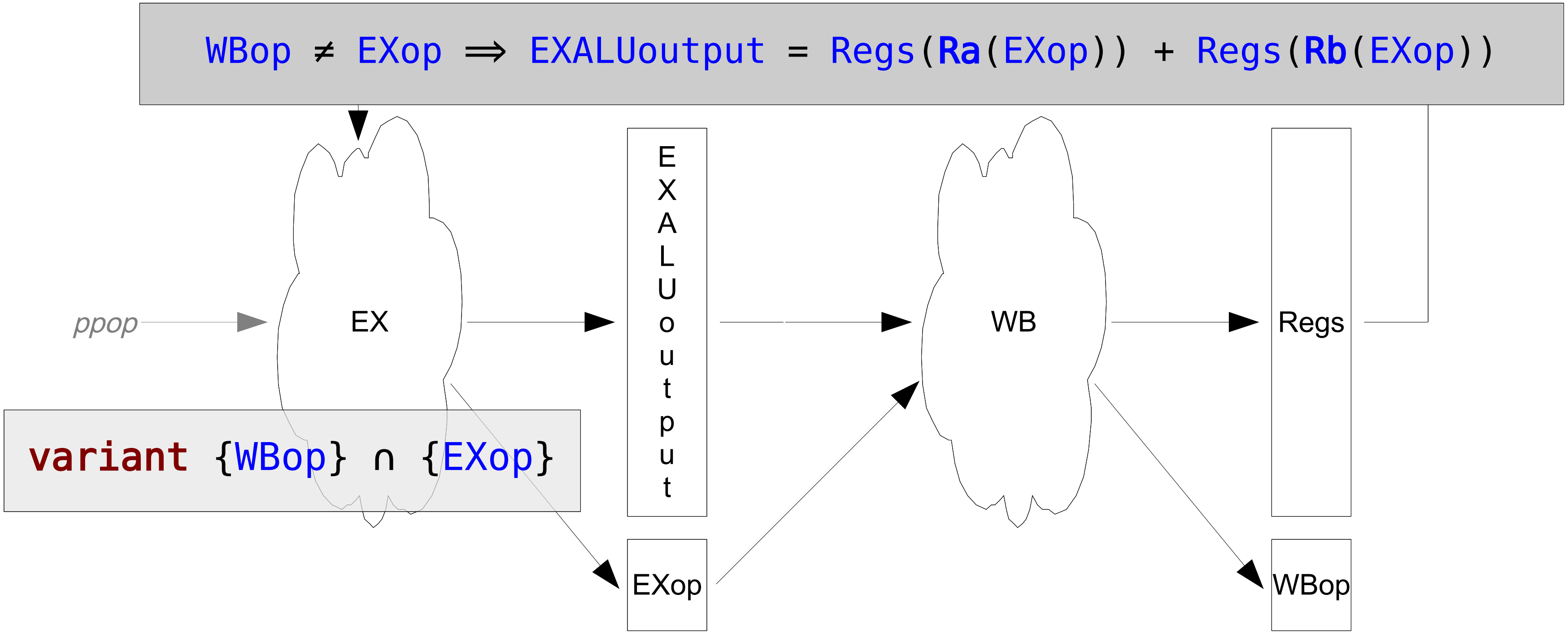
variant {WBo

Invariant also ensures Deadlock Freeness

```
convergent event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 ppop ≠ EXop
    @grd3 WBop = EXop
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

```
event WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 WBop ≠ EXop
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

Sequential Execution: Simplifying the Invariant



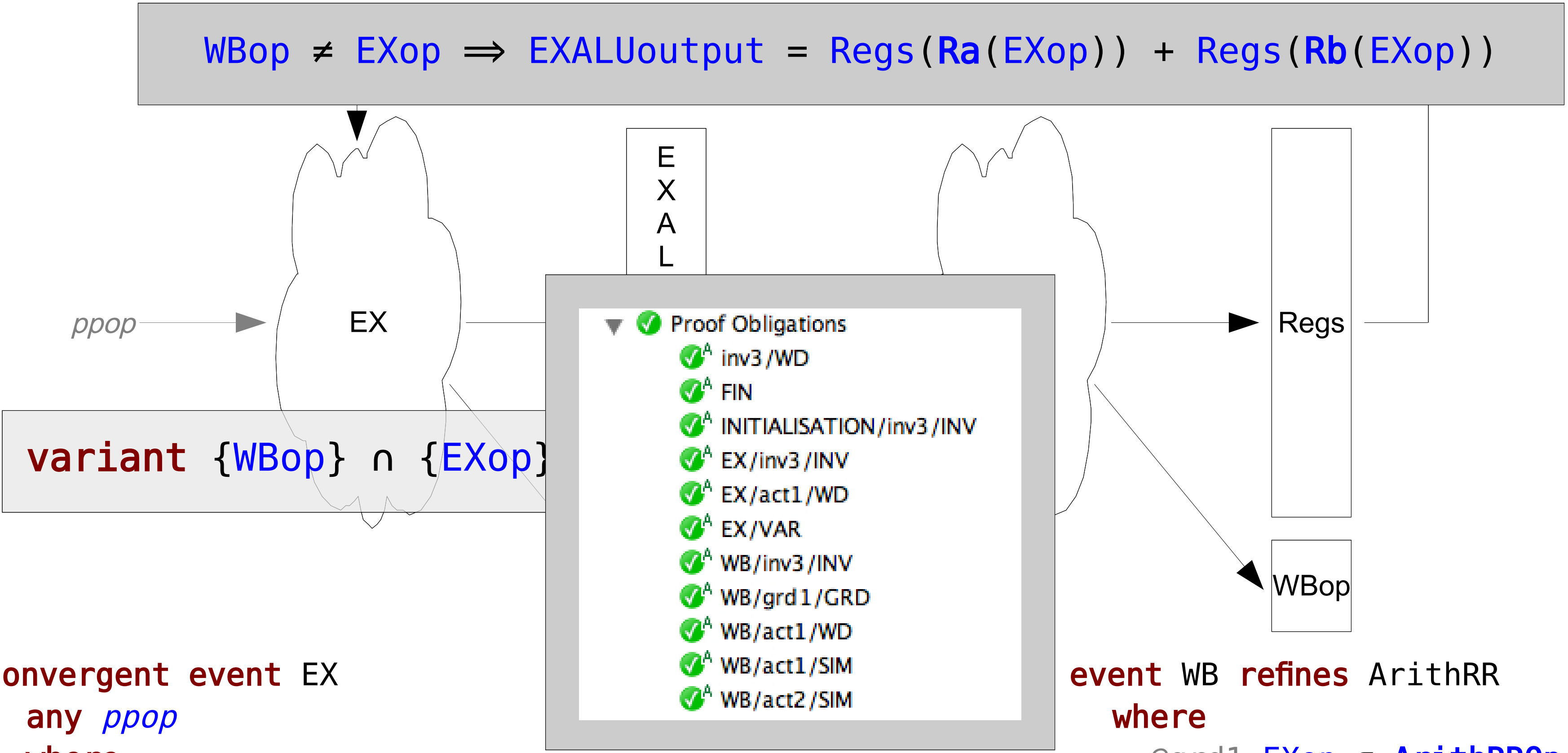
WBop ≠ EXop ⟹ EXALUoutput = Regs(**Ra**(EXop)) + Regs(**Rb**(EXop))

variant {WBop} ∩ {EXop}

```
convergent event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 ppop ≠ EXop
    @grd3 WBop = EXop
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

```
event WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 WBop ≠ EXop
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

Sequential Execution: A Correct Refinement of the Abstract Model



WBop ≠ EXop ⟹ EXALUoutput = Regs(Ra(EXop)) + Regs(Rb(EXop))

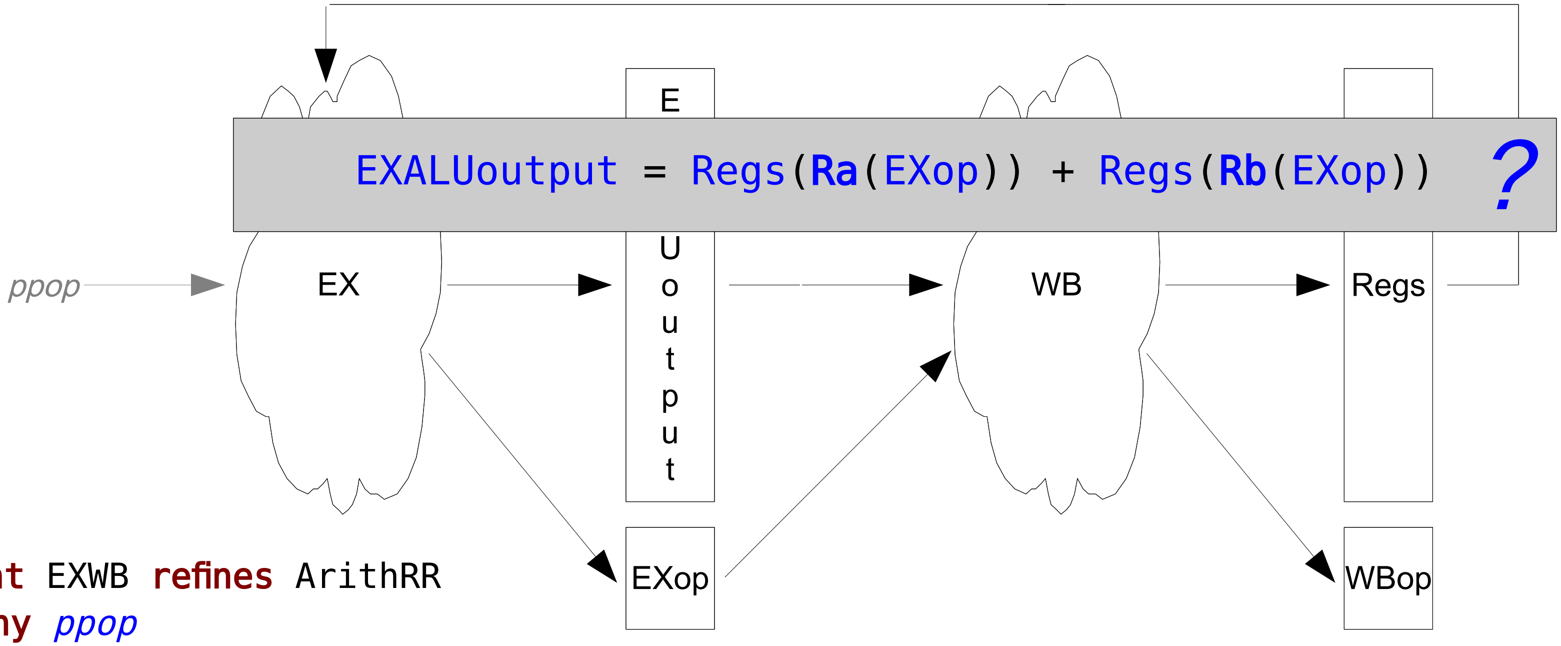
variant {WBop} ∩ {EXop}

- Proof Obligations
  - inv3/WD
  - FIN
  - INITIALISATION/inv3/INV
  - EX/inv3/INV
  - EX/act1/WD
  - EX/VAR
  - WB/inv3/INV
  - WB/grd1/GRD
  - WB/act1/WD
  - WB/act1/SIM
  - WB/act2/SIM

```
convergent event EX
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 ppop ≠ EXop
    @grd3 WBop = EXop
  then
    @act1 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act2 EXop ≔ ppop
end
```

```
event WB refines ArithRR
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 WBop ≠ EXop
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
end
```

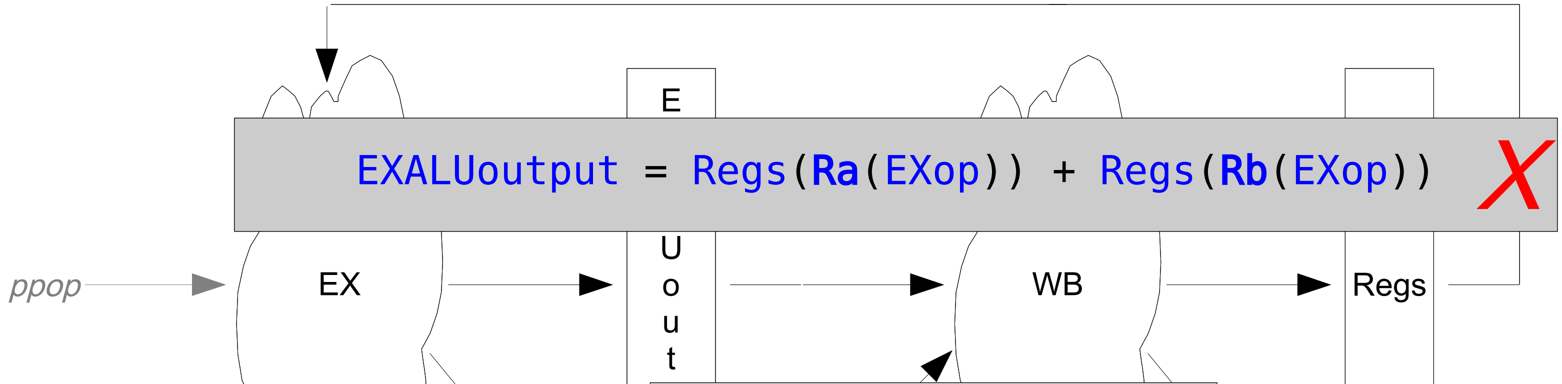
Consider *Parallel* Execution



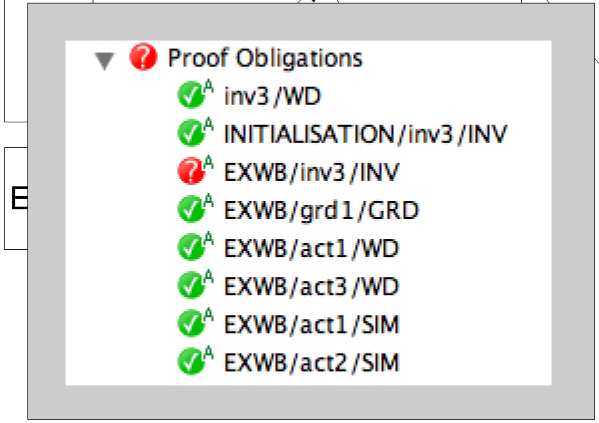
EXALUoutput = Regs(**Ra**(EXop)) + Regs(**Rb**(EXop)) ?

```
event EXWB refines ArithRR
  any ppop
  where
    @grd1 EXop ∊ ArithRROp
    @grd2 ppop ∊ ArithRROp
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
    @act3 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act4 EXop ≔ ppop
end
```

Consider *Parallel* Execution



EXALUoutput = Regs(**Ra**(EXop)) + Regs(**Rb**(EXop)) ✗

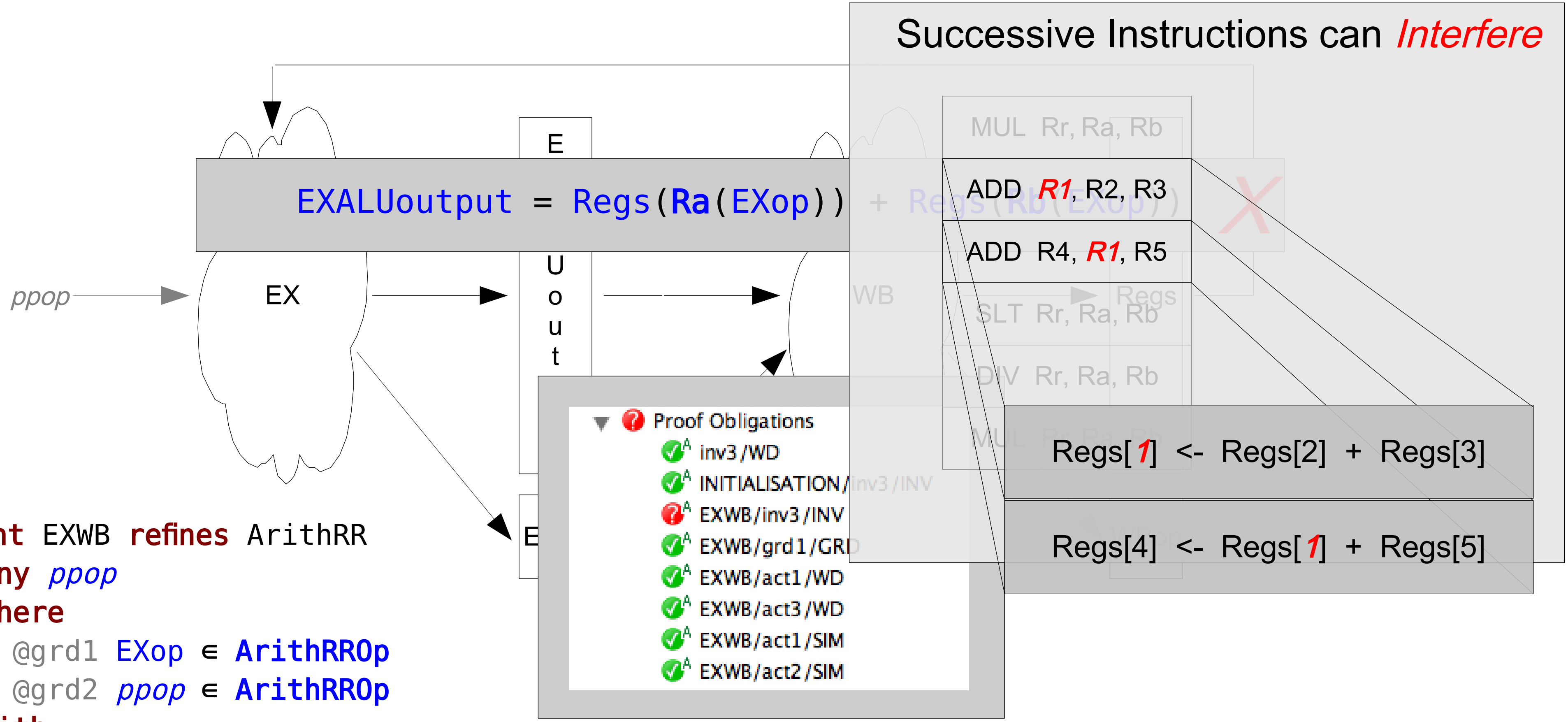


- ? Proof Obligations
  - ✓ inv3/WD
  - ✓ INITIALISATION/inv3/INV
  - ? EXWB/inv3/INV
  - ✓ EXWB/grd1/GRD
  - ✓ EXWB/act1/WD
  - ✓ EXWB/act3/WD
  - ✓ EXWB/act1/SIM
  - ✓ EXWB/act2/SIM

```
event EXWB refines ArithRR
  any ppop
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 ppop ∈ ArithRROp
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
    @act3 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act4 EXop ≔ ppop
end
```

UNIVERSITY OF
Southampton
School of Electronics
and Computer Science

Consider *Parallel* Execution



Successive Instructions can *Interfere*

MUL Rr, Ra, Rb
ADD *R1*, R2, R3
ADD R4, *R1*, R5
SLT Rr, Ra, Rb
DIV Rr, Ra, Rb
MUL Rr, Ra, Rb

Regs[*1*] <- Regs[2] + Regs[3]

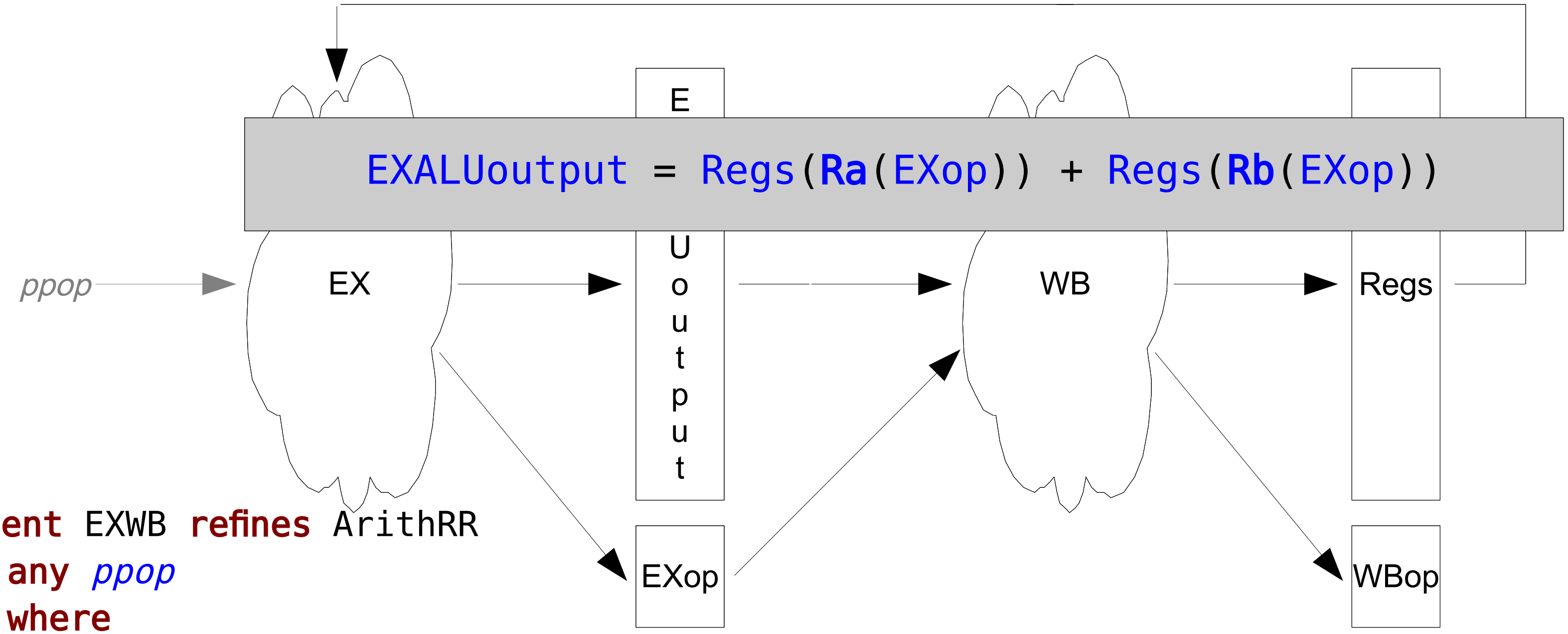
Regs[4] <- Regs[*1*] + Regs[5]

Proof Obligations
- ✓ inv3/WD
- ✓ INITIALISATION/inv3/INV
- ? EXWB/inv3/INV
- ✓ EXWB/grd1/GRD
- ✓ EXWB/act1/WD
- ✓ EXWB/act3/WD
- ✓ EXWB/act1/SIM
- ✓ EXWB/act2/SIM

```
event EXWB refines ArithRR
  any ppop
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 ppop ∈ ArithRROp
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
    @act3 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
    @act4 EXop ≔ ppop
end
```

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Parallel Execution must detect potential *RAW Hazard*[†]....

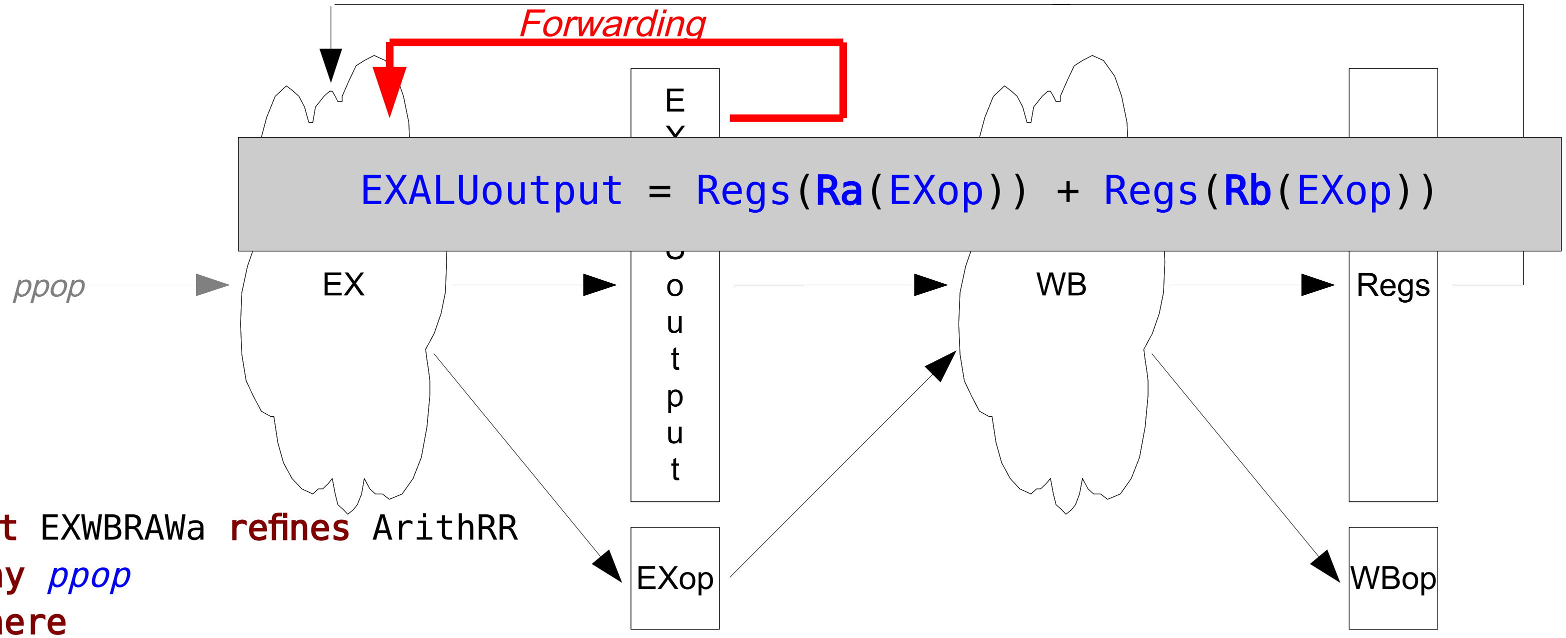


EXALUoutput = Regs(**Ra**(EXop)) + Regs(**Rb**(EXop))

```
event EXWB refines ArithRR
   any ppop
   where
     @grd1 EXop ∊ ArithRROp
     @grd2 ppop ∊ ArithRROp
     @grd3 Rr(EXop) ≠ Ra(ppop) // no RAW hazard on register a
     @grd4 Rr(EXop) ≠ Rb(ppop) // no RAW hazard on register b
   with
     @pop pop = EXop
   then
     @act1 Regs(Rr(EXop)) ≔ EXALUoutput
     @act2 WBop ≔ EXop
     @act3 EXALUoutput ≔ Regs(Ra(ppop)) + Regs(Rb(ppop))
     @act4 EXop ≔ ppop
 end
```

Computer Architecture: A Quantitative Approach
Hennessy and Patterson, 1990

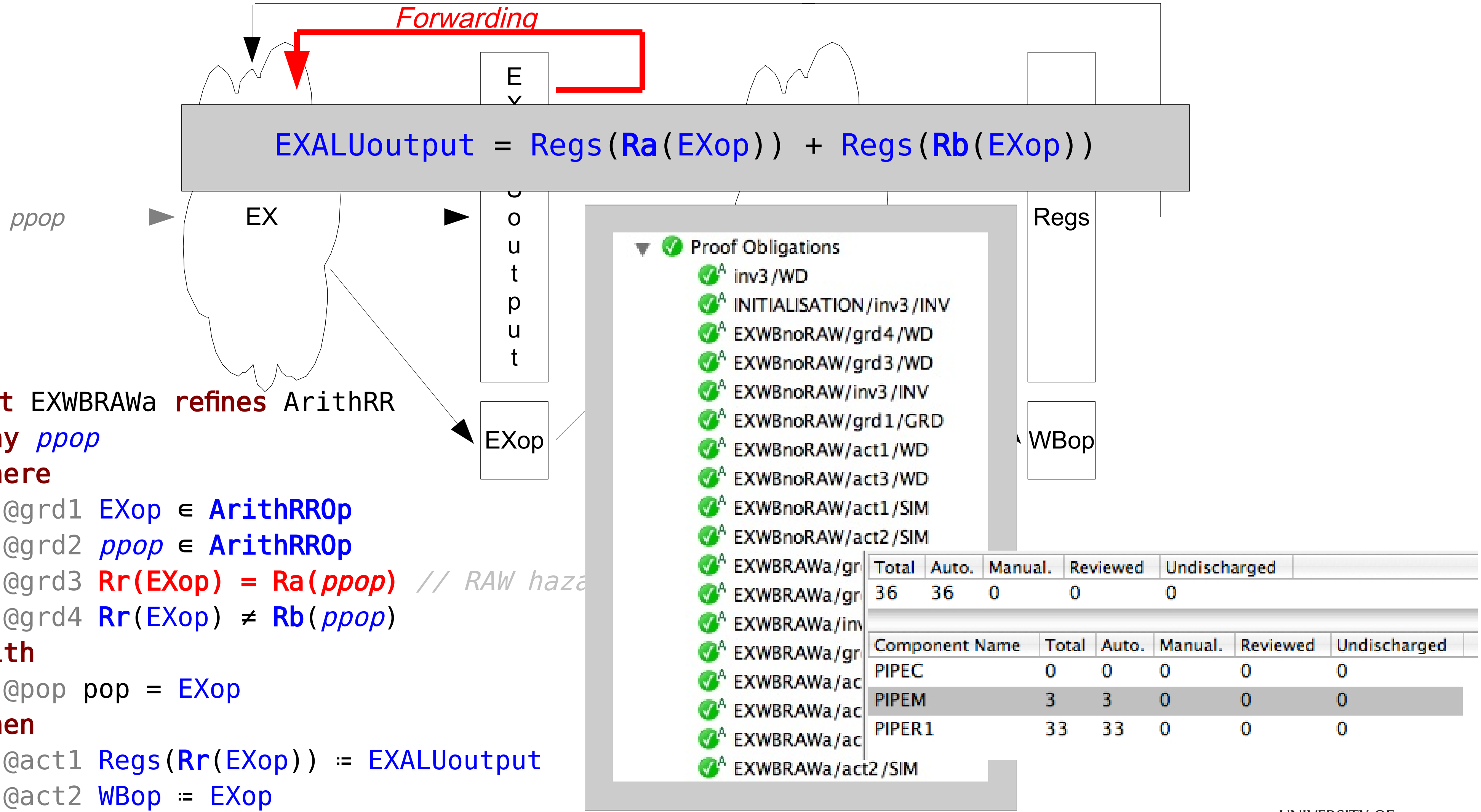
.... and deal with the *RAW Hazard* correctly



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

```
event EXWBRAWa refines ArithRR
  any ppop
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 ppop ∈ ArithRROp
    @grd3 Rr(EXop) = Ra(ppop) // RAW hazard on register a
    @grd4 Rr(EXop) ≠ Rb(ppop)
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
    @act3 EXALUoutput ≔  EXALUoutput + Regs(Rb(ppop))
    @act4 EXop ≔ ppop
end
```

.... and deal with the *RAW Hazard* correctly



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

Proof Obligations
- ✓ inv3/WD
- ✓ INITIALISATION/inv3/INV
- ✓ EXWBnoRAW/grd4/WD
- ✓ EXWBnoRAW/grd3/WD
- ✓ EXWBnoRAW/inv3/INV
- ✓ EXWBnoRAW/grd1/GRD
- ✓ EXWBnoRAW/act1/WD
- ✓ EXWBnoRAW/act3/WD
- ✓ EXWBnoRAW/act1/SIM
- ✓ EXWBnoRAW/act2/SIM
- ✓ EXWBRAWa/gr
- ✓ EXWBRAWa/gr
- ✓ EXWBRAWa/inv
- ✓ EXWBRAWa/gr
- ✓ EXWBRAWa/ac
- ✓ EXWBRAWa/ac
- ✓ EXWBRAWa/ac
- ✓ EXWBRAWa/act2/SIM

Total	Auto.	Manual.	Reviewed	Undischarged
36	36	0	0	0

Component Name	Total	Auto.	Manual.	Reviewed	Undischarged
PIPEC	0	0	0	0	0
PIPEM	3	3	0	0	0
PIPER1	33	33	0	0	0

```
event EXWBRAWa refines ArithRR
  any ppop
  where
    @grd1 EXop ∈ ArithRROp
    @grd2 ppop ∈ ArithRROp
    @grd3 Rr(EXop) = Ra(ppop) // RAW haza
    @grd4 Rr(EXop) ≠ Rb(ppop)
  with
    @pop pop = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 WBop ≔ EXop
    @act3 EXALUoutput ≔  EXALUoutput + Regs(Rb(ppop))
    @act4 EXop ≔ ppop
end
```

UNIVERSITY OF
Southampton
School of Electronics
and Computer Science

Summary and Future Work

- A Systematic Method for Pipelined Hardware Component Specification is being developed using Event-B refinement and automatic proof
  - Micro-architectural Exploration and Verification can be raised to the Specification Level
  - A route to Bluespec, CAL is being explored
  - Can potentially be incorporated into an existing High-Level Synthesis Methodology