Cornell University
Computer Systems Laboratory

Formal Verification of the Stall Invariant Property for Latency-Insensitive RTL Modules

Peitian Pan, **Christopher Batten**

MEMOCODE @ Hamburg, Germany

September 22nd, 2023

Latency-Insensitive (LI) Interfaces

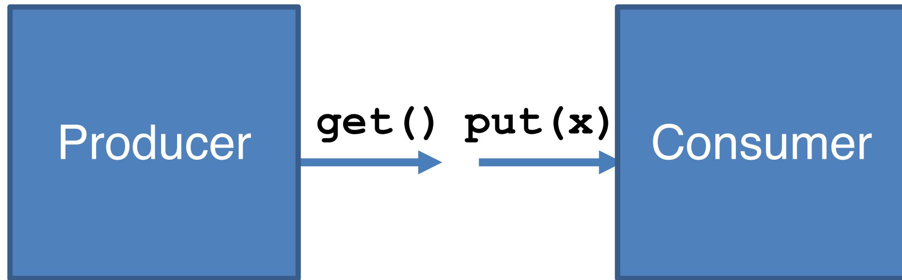
Communication channel that does not depend on the latency of the producer or the consumer

Benefit #1: Encapsulation
Internal timing details are not exposed in the interface of the hardware component

Benefit #2: Composition
A standard communication protocol facilitates composing different hardware components

Benefit #3: Timing closure
Can retime channels without changing the implementation of hardware components

Latency-Insensitive (LI) Interfaces



Communication channel that does not depend on the latency of the producer or the consumer

Benefit #1: Encapsulation
Internal timing details are not exposed in the interface of the hardware component

Benefit #2: Composition
A standard communication protocol facilitates composing different hardware components

Benefit #3: Timing closure
Can retime channels without changing the implementation of hardware components

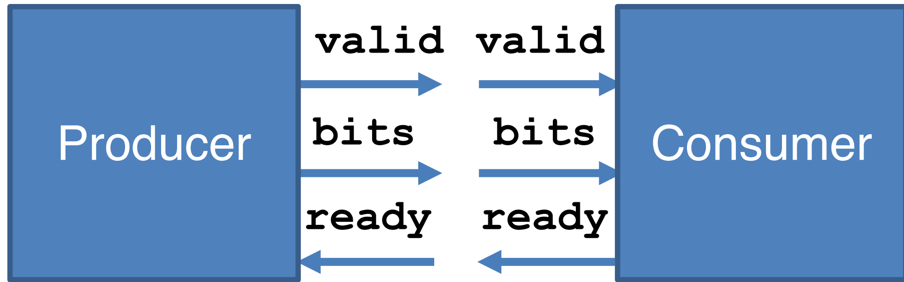
Bluespec Get/Put Interface

```
module mkMyFifoUpstream (Get#(int));
...
  method ActionValue#(int) get();
    f.deq;
    return f.first;
  endmethod

module mkMyFifoDownstream (Put#(int));
...
  method Action put(int x);
    f.enq(x);
  endmethod
```

Source: Bluespec Reference Guide; Appendix C.7.1

Latency-Insensitive (LI) Interfaces



Communication channel that does not depend on the latency of the producer or the consumer

Benefit #1: Encapsulation
Internal timing details are not exposed in the interface of the hardware component

Benefit #2: Composition
A standard communication protocol facilitates composing different hardware components

Benefit #3: Timing closure
Can retime channels without changing the implementation of hardware components

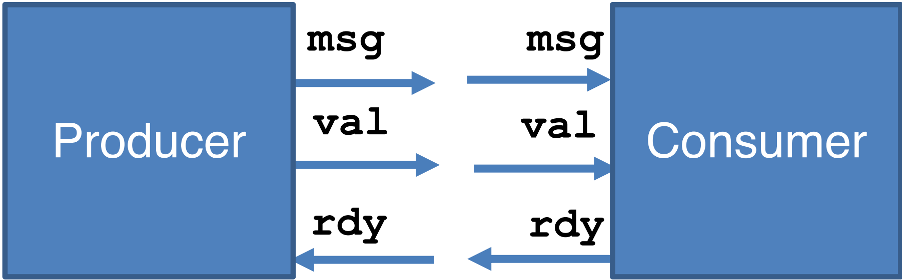
Chisel Decoupled IO Bundles

```
class Producer extends Module {
  val io = IO(new Bundle {
    val readyVal = Decoupled(UInt(32.W))
  })
  // use io.readyVal.ready
  io.readyVal.valid := true.B
  io.readyVal.bits := 5.U
}

class Consumer extends Module {
  val io = IO(new Bundle {
    val readyVal = Flipped(Decoupled(UInt(32.W)))
  })
  // use io.readyVal.valid
  // use io.readyVal.bits
  io.readyVal.ready := false.B
}
```

Source: Chisel/FIRRTL Documnetation; Interfaces and Connections

Latency-Insensitive (LI) Interfaces



Communication channel that does not depend on the latency of the producer or the consumer

Benefit #1: Encapsulation
Internal timing details are not exposed in the interface of the hardware component

Benefit #2: Composition
A standard communication protocol facilitates composing different hardware components

Benefit #3: Timing closure
Can retime channels without changing the implementation of hardware components

BaseJump Verilog Standard Template Library

```
module bsg_two_fifo
#(
  parameter width_p = 32
)(
  input clk_i,
  input reset_i,

  // Producer side
  input [width_p-1:0] msg_i,
  input               val_i,
  output              rdy_o,

  // Consumer side
  output[width_p-1:0] msg_o,
  output              val_o,
  input               rdy_i
);
```

val: producer asserts at beginning of cycle if has valid message to send

rdy: consumer asserts at beginning of cycle if ready to receive message

msg: if **val** & **rdy**, **msg** is transferred at the end of the cycle

Two-FIFO Source: BaseJump STL Github Repo

Latency-Insensitive RTL Design Bugs

in_val
Control
IDLE
out_val
③
④
①
in_rdy
DONE
CALC
out_rdy
②
b_lsb
reg_en
mux_sel
mul_a
Data Path
>>
mul_b
<<
+
0
product

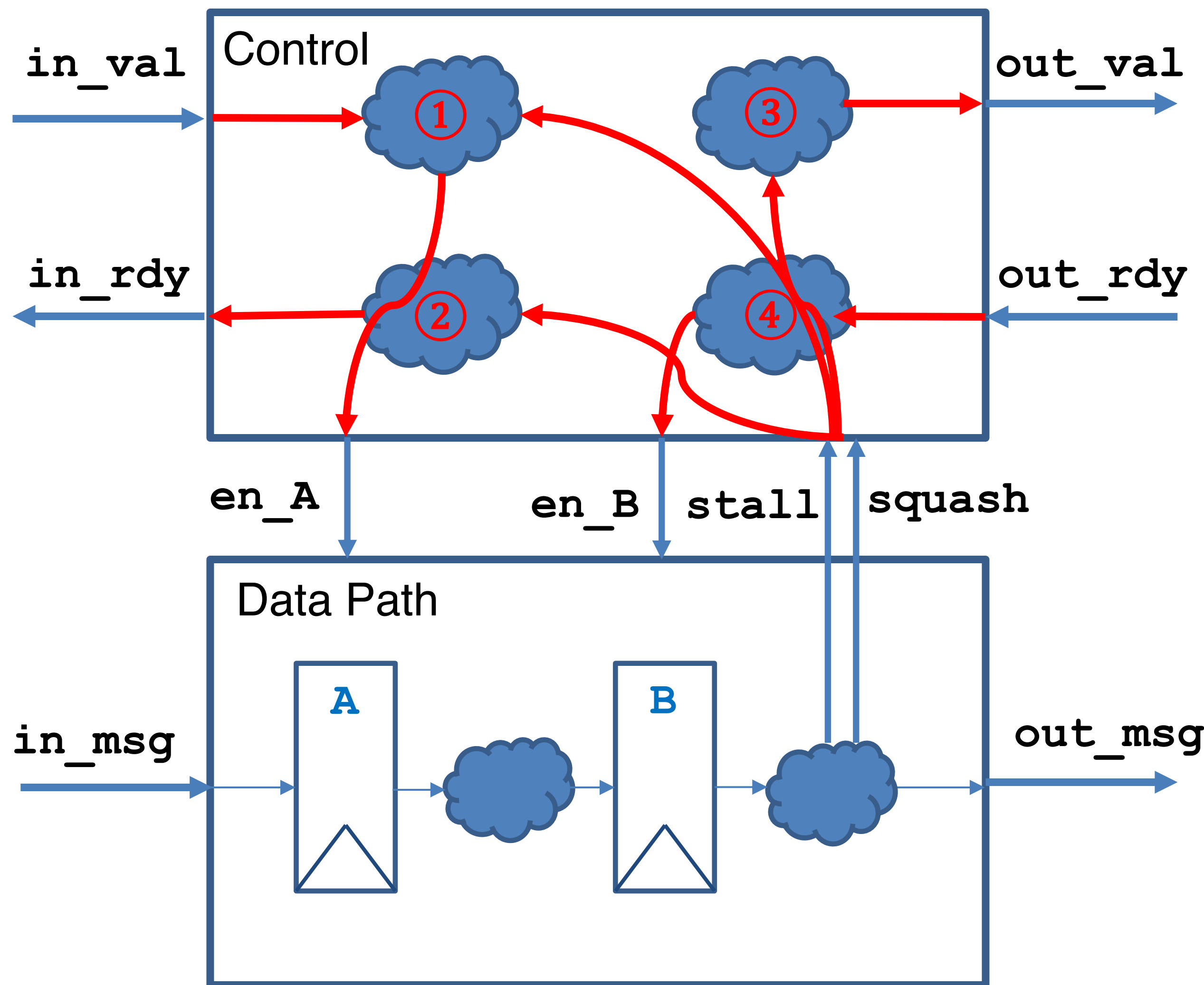
1. Input val bug
May consume invalid input messages when the input interface is not valid
① `IDLE to CALC when in_val != 1`

2. Input rdy bug
May drop input messages when the design is not ready to accept an input
② `in_rdy = (STATE != IDLE)`

3. Output val bug
May produce invalid output messages when the design is not valid to produce an output
③ `out_val = (STATE != DONE)`

4. Output rdy bug
May drop output messages when the output interface is not ready
④ `DONE to IDLE when out_rdy != 1`

Latency-Insensitive RTL Design Bugs



1. Input val bug
May consume invalid input messages when the input interface is not valid
① `en_A = (in_val != 1) || stall`

2. Input rdy bug
May drop input messages when the design is not ready to accept an input
② `in_rdy not depend on stall/squash`

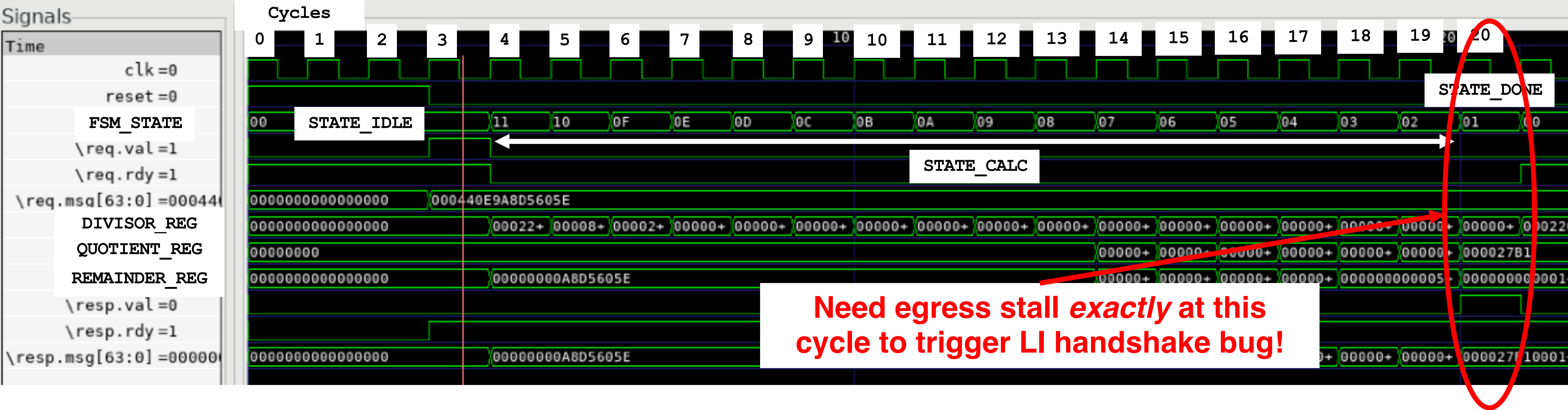
3. Output val bug
May produce invalid output messages when the design is not valid to produce an output
③ `out_val not depend on stall/squash`

4. Output rdy bug
May drop output messages when the output interface is not ready
④ `en_B = (out_rdy != 1)`

Latency-Insensitive RTL Module Verification Challenge

Step 1: Directed or random testing on the DUV without any stalls (verifies functionality)

Step 2: Inject ingress or egress stalls (verifies LI handshakes)



Research Goal: After step 1, use formal verification to prove the stall invariant property of latency-insensitive RTL modules

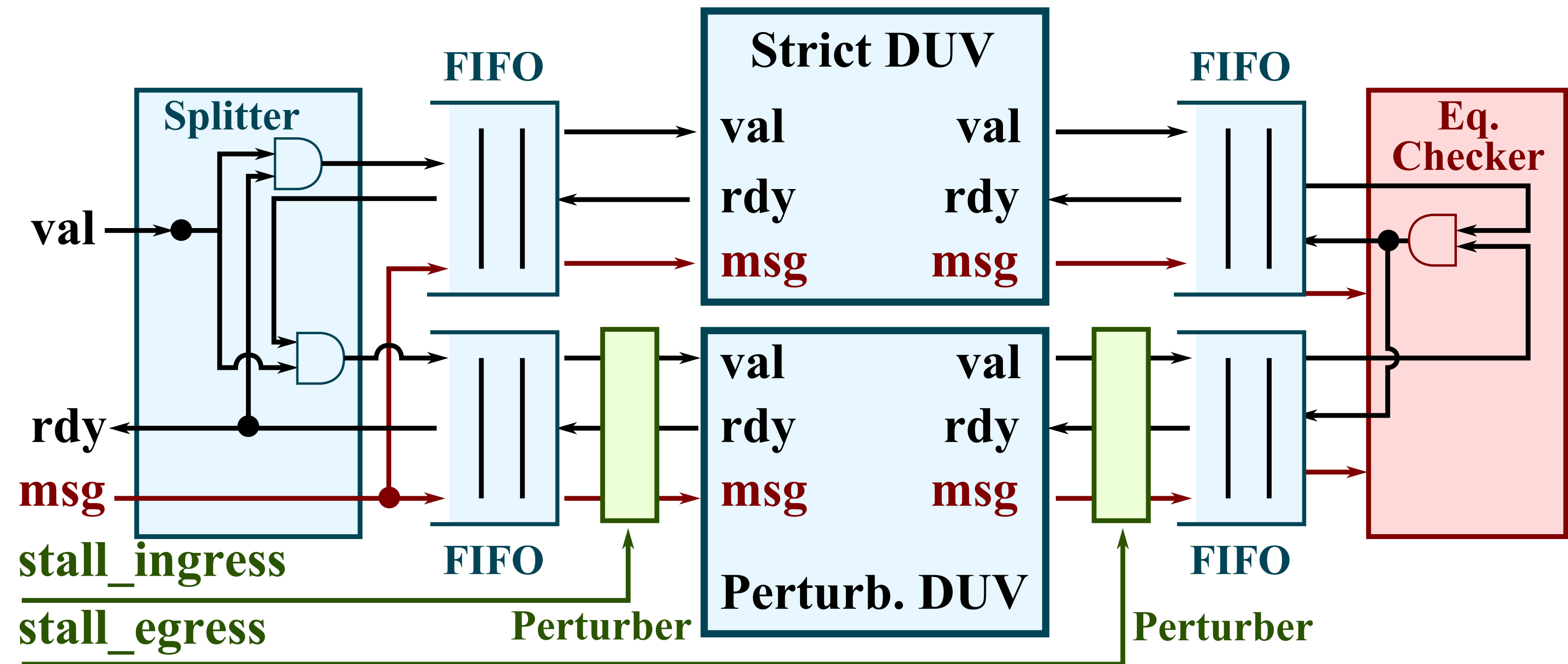
Formal Verification of the Stall Invariant Property for Latency-Insensitive RTL Modules

Motivation

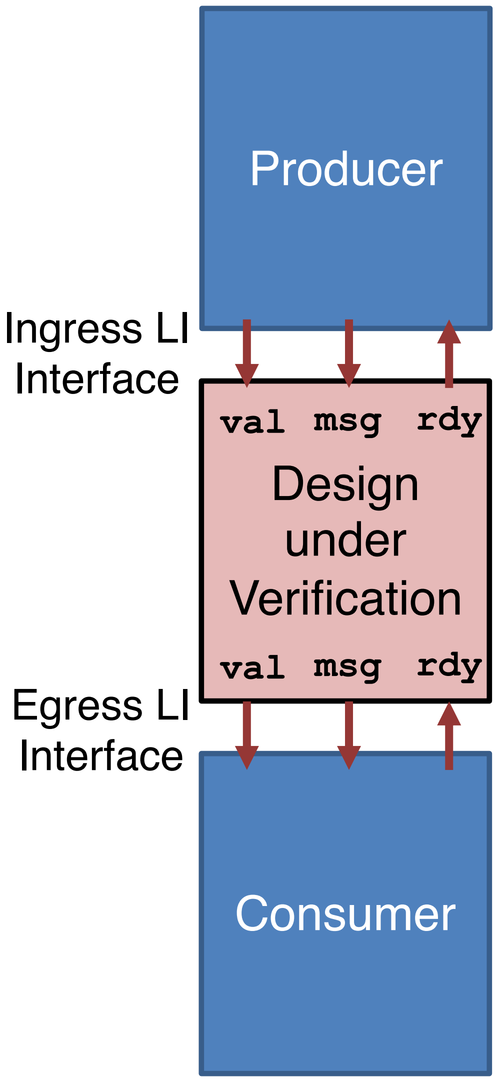
Stall Invariant Property

Verification Approach

Case Studies



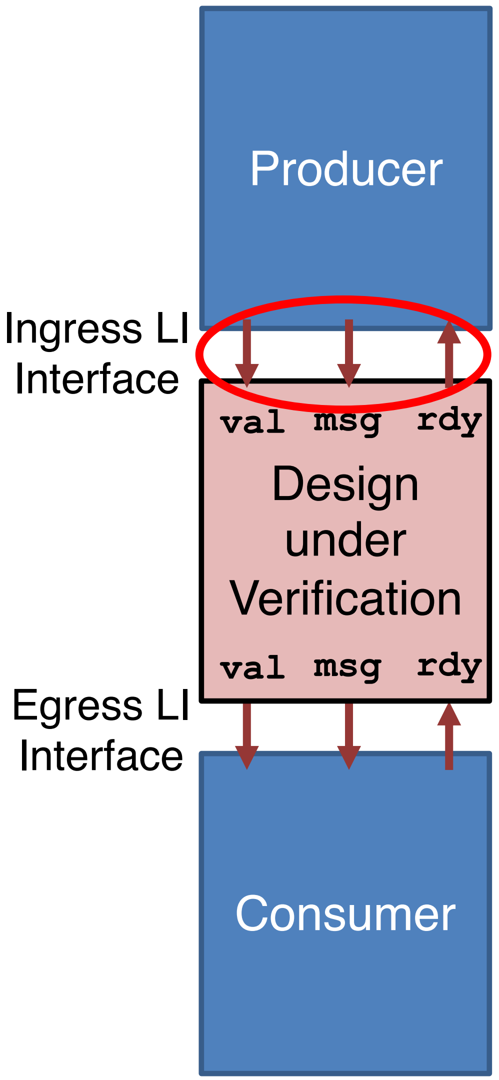
Latency-Insensitive Interface Terminology



Cycles

	1	2	3	4	5	6	7	8
Ingress	A	B	C	D	E			
Egress		A	B	C	D	E		

Latency-Insensitive Interface Terminology



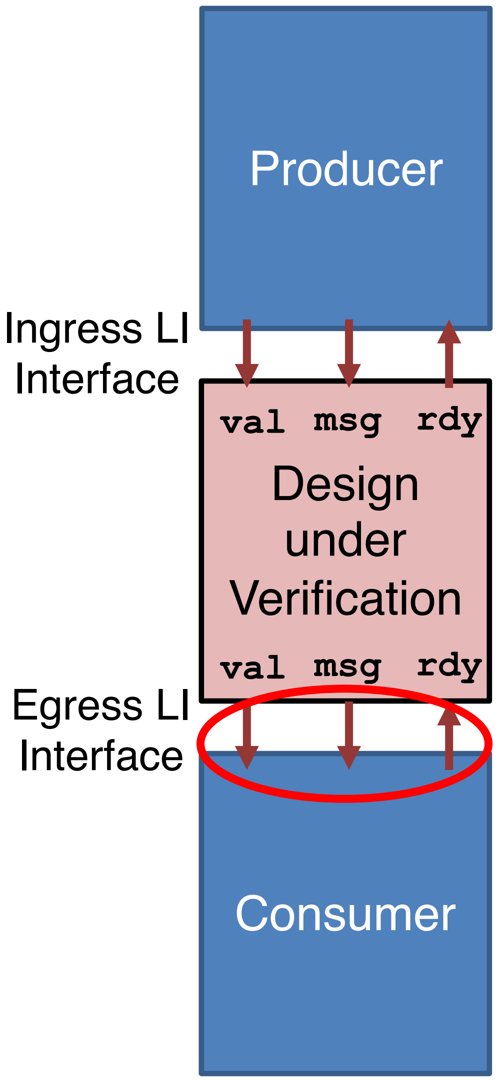
Cycles →

	1	2	3	4	5	6	7	8
Ingress	A	B	C	D	E			
Egress		A	B	C	D	E		

	1	2	3	4	5	6	7	8
Ingress	A	B	C	_	_	D	E	
Egress		A	B	C	_	_	D	E

Ingress stall events:
a stall that happens on the ingress LI interface because the upstream producer does not produce a message that cycle (`val` == 0)

Latency-Insensitive Interface Terminology



Cycles →

	1	2	3	4	5	6	7	8
Ingress	A	B	C	D	E			
Egress		A	B	C	D	E		

Ingress stall events:
a stall that happens on the ingress LI interface because the upstream producer does not produce a message that cycle (**val** == 0)

	1	2	3	4	5	6	7	8
Ingress	A	B	C	_	_	D	E	
Egress		A	B	C	_	_	D	E

Egress Stall Events:
a stall that happens on the egress LI interface because the downstream consumer is not ready to accept a message that cycle (**rdy** == 0)

	1	2	3	4	5	6	7	8
Ingress	A	B	C	#	#	D	E	
Egress		A	B	#	#	C	D	E

Informative Events:
transactions that happen on a LI interface (ingress or egress) with **val** == 1 and **rdy** == 1

Definition of Stall Invariant Property

A DUV is *stall invariant* if for any given sequence of informative events at the ingress LI interfaces, it produces the same sequence of informative events at the egress LI interfaces under all possible interleaving of stall events

	1	2	3	4	5	6
Ingress	A	B	C	D		
Egress		A	B	C	D	

Reference Strict Behavior

	1	2	3	4	5	6
Ingress	A	B	C	_	D	
Egress		A	B	C	_	D

Necessary for Stall Invariant

	1	2	3	4	5	6	7
Ingress	A	B	_	C	#	D	
Egress		A	B	_	#	C	D

Necessary for Stall Invariant

	1	2	3	4	5	6
Ingress	A	B	C	D		
Egress		A	B	C	D	E

Cannot be Stall Invariant

	1	2	3	4	5	6
Ingress	A	B	C	_	D	
Egress		A	B	C	_	

Cannot be Stall Invariant

	1	2	3	4	5	6	7
Ingress	A	B	_	C	#	D	
Egress		A	B	_	#	C	E

Cannot be Stall Invariant

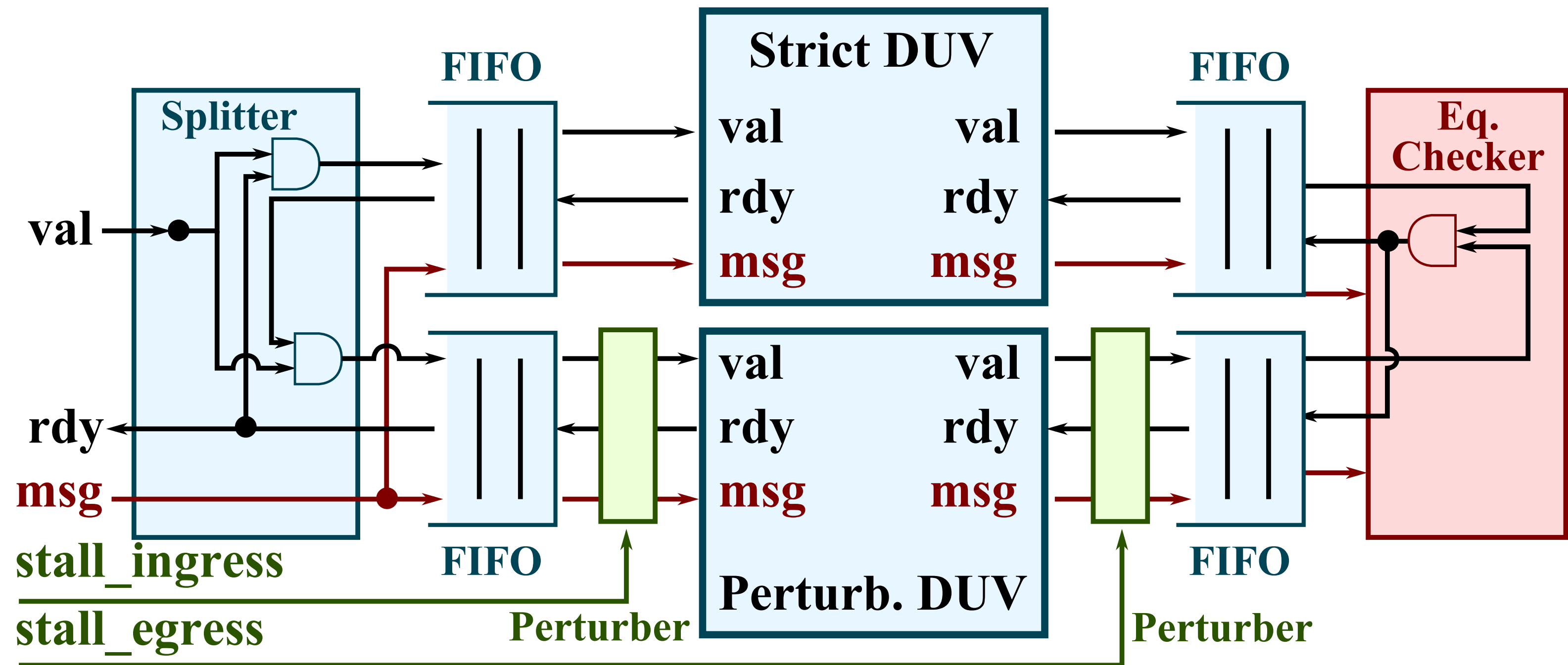
Formal Verification of the Stall Invariant Property for Latency-Insensitive RTL Modules

Motivation

Stall Invariant Property

Verification Approach

Case Studies



Formal Verification Harness of the Stall Invariant

Strict DUV

val	val
rdy	rdy
msg	msg

Strict Path

val	val
rdy	rdy
msg	msg

Perturb. DUV

Perturbed Path

Formal Verification Harness of the Stall Invariant

Splitter

val

rdy

msg

Strict DUV

val val

rdy rdy

msg msg

val val

rdy rdy

msg msg

Perturb. DUV

Strict Path

Perturbed Path

Eq. Checker

s_val

s_rdy

s_msg

p_val

p_rdy

p_msg

s_msg == p_msg

- The same transactions are split and fed into both the strict and perturbed DUVs
- The equivalence checker compares if the informative events of the strict and perturbed DUVs are the same

Formal Verification Harness of the Stall Invariant

Strict DUV
val val
rdy rdy
msg msg

Strict Path

Eq. Checker
s_val
s_rdy
s_msg
p_val
p_rdy
p_msg
s_msg == p_msg

Splitter
val
rdy
msg
stall_ingress
stall_egress

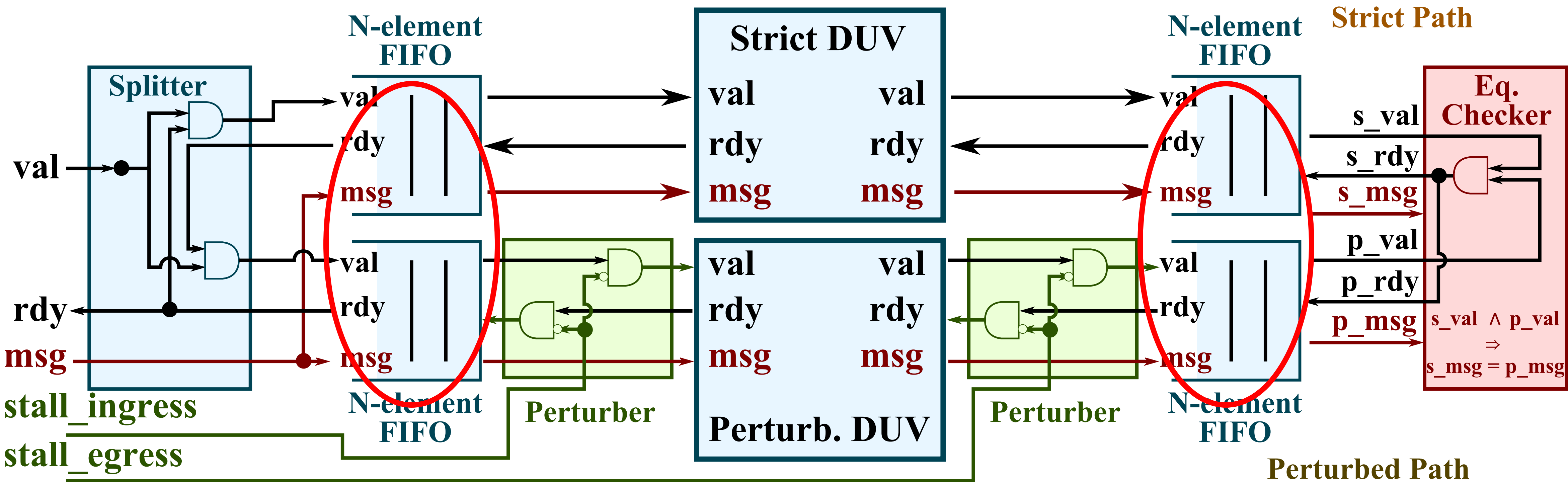
val
rdy
msg
Perturber

Perturb. DUV
val val
rdy rdy
msg msg

Perturber
Perturbed Path

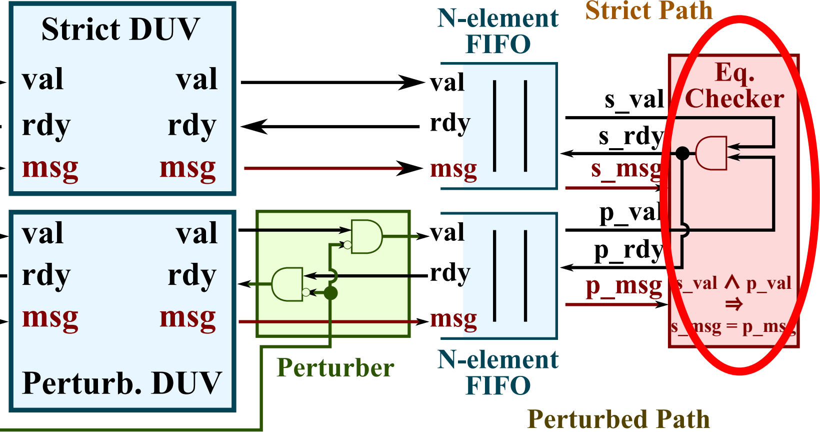
- The perturbers use free variables (`stall_ingress` and `stall_egress`) to introduce stall events
- This harness does NOT work as intended because the strict and perturbed DUVs are latency-coupled!

Formal Verification Harness of the Stall Invariant



- Ideally, we want FIFOs of unbounded depths to fully decouple the strict and perturbed DUVs
- Practically we must use bounded FIFOs; transactions on the strict DUV may still stall because FIFOs are full
- Technically this means we are not fully proving the stall invariant property
- FIFOs with larger depths improve decoupling but also increase formal verification tool time

Specification of Equivalence Checker Properties



Safety Property

- Compare if the produced informative events from the strict and perturbed DUVs are the same

```
ast_same_msg: assert property (
 @(posedge clk) disable iff (rst) (
  (s_val & p_val) |-> (s_msg == p_msg)
 ));
```

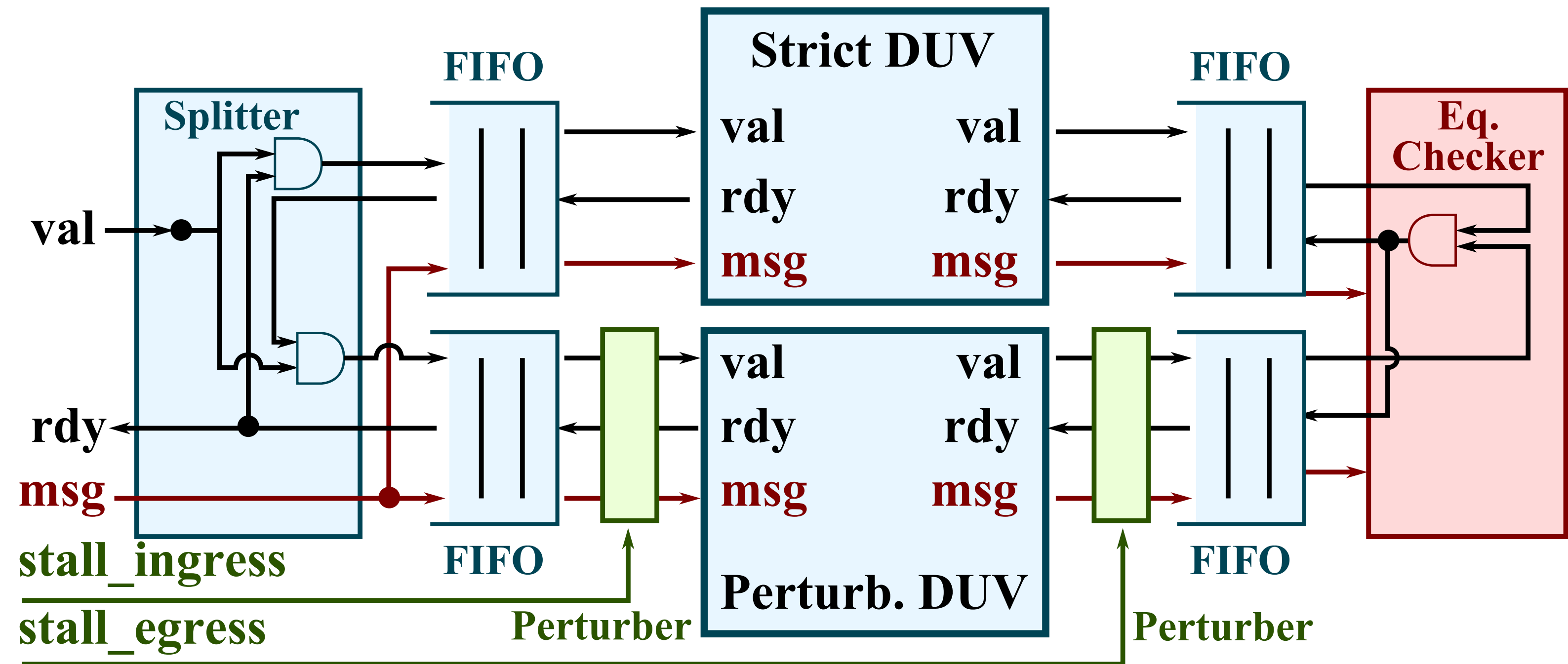
Liveness Property

- What if the perturbed DUVs drop messages?
- Safety property will not detect them because the precondition will not be satisfied (i.e., **`ast_same_msg`** passes *vacuously*)
- The **`ast_same_vals`** property checks if the strict DUV produces a message, then *after finite number of cycles* the perturbed one will produce a message

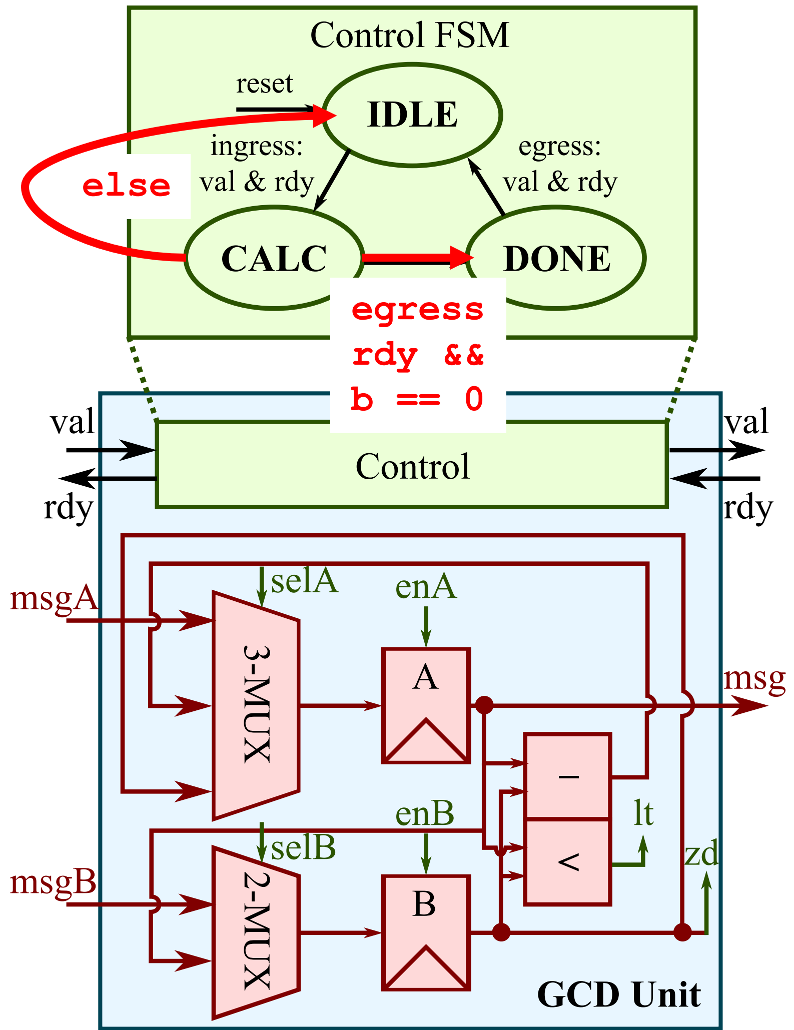
```
ast_same_vals: assert property (
 @(posedge clk) disable iff (rst) (
  (s_val & ~p_val) |->
             s_eventually (s_val & p_val)
  and
  (~s_val & p_val) |->
             s_eventually (s_val & p_val)
 ));
```

Formal Verification of the Stall Invariant Property for Latency-Insensitive RTL Modules

- Motivation
- Stall Invariant Property
- Verification Approach
- **Case Studies**



Case Study #1: Greatest Common Divisor (GCD) Accelerator



GCD Design Bug

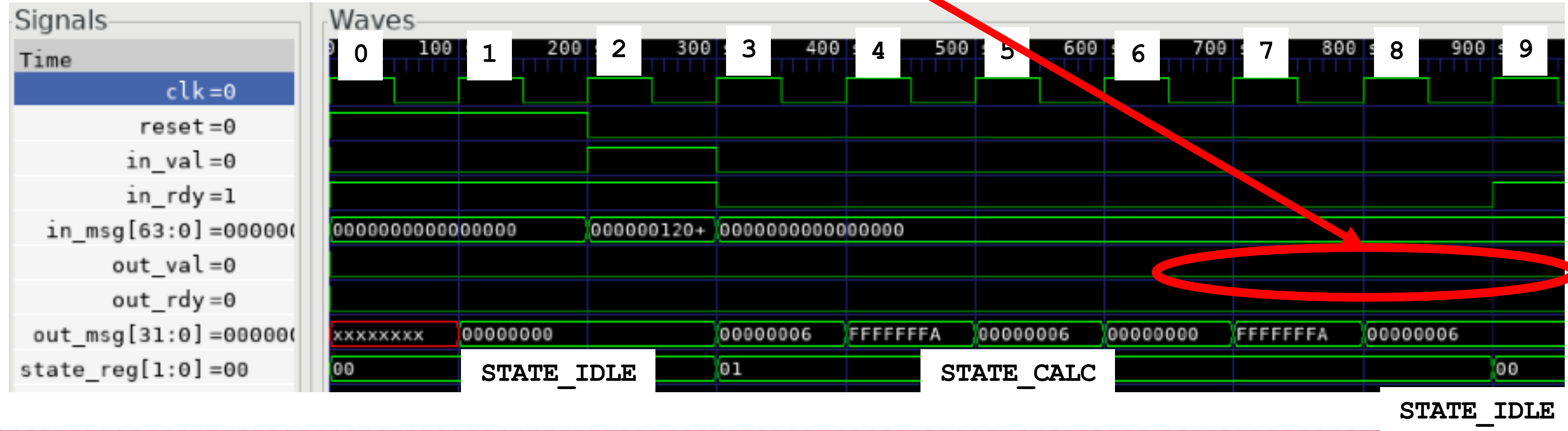
- **CALC** transits to **DONE** only if egress interface is ready
- Output message may never appear on the egress interface

Detecting Bug with Formal Verification

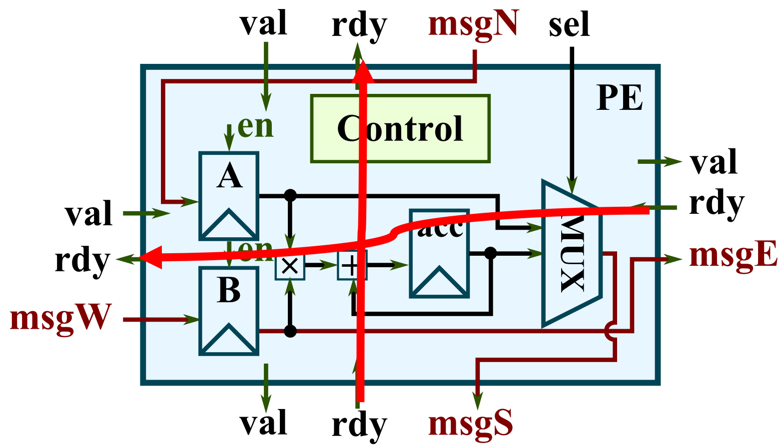
- **ast_same_vals** *liveness* assertion fails with two-message counter example

Proof of Stall Invariant Property for Bug-Free GCD Accelerator

- Replaces subtraction with bit-wise XOR; unconditional CALC to DONE to reduce the number of cycles JasperGold has to reason about
- Proof achieved in ~20 minutes



Case Study #2: Processing Element



PE Functionality

- PE takes input from north and west and generates output to south (accumulation result) and east (west message)

PE Design Bug

- Bypasses east and south **`rdy`** to west and north
- Real bug from a graduate student implementing the PE

Correct Verilog code for **`rdy`** in north and west LI interface:

```
assign rdyN = valW && rdyS && rdyE;
assign rdyW = valN && rdyS && rdyE;
```

Buggy Verilog code for **`rdy`** in north and west LI interface:

```
assign rdyN = rdyS;
assign rdyW = rdyE;
```

Bug symptom: messages from north and west are not consumed at the same cycle when south and east are not ready at the same cycle

Passed all of student's dynamic verification because always injected egress stalls on all egress interfaces on the same cycle

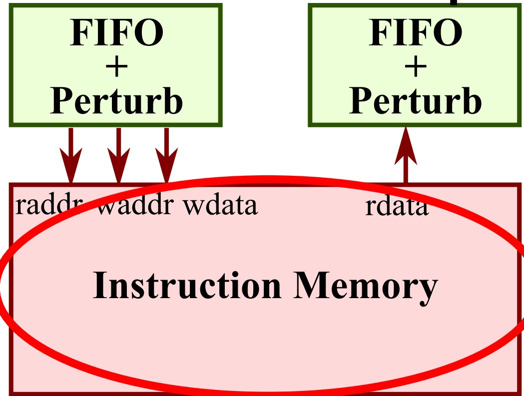
Detecting Bug with Formal Verification

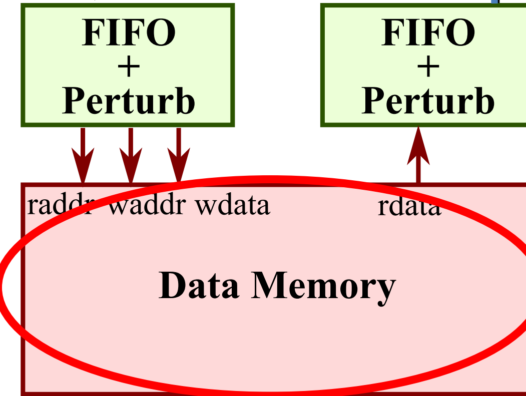
- JasperGold finds a 5-cycle counter example to the **`ast_same_msg`** property

Proof of Stall Invariant for Correct PE

- Replaced multiplier with bit-wise XOR to accelerate convergence
- JasperGold proves PE is stall invariant in ~1.5 hours

Case Study #3: RISC-V Processor





Processor Functionality

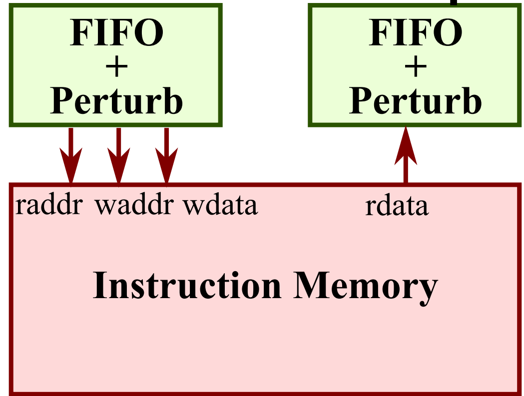
- 5-stage RISC-V processor
- Supports ADD, ADDI, BNE, LW
- Instruction and data memory interfaces are not stall invariant
- RVVI interface to observe the committed instructions

Verification Harness Setup

- 32-bit datapath
- 64 32-bit words in inst and data memory
- Strict and perturbed processors share the same instruction and data memory

Case Study #3: RISC-V Processor Design Bug

Processor imem_req_addr
enF enD enX enM enW
F D inst jmp X br_taken M W
RegFile (Read) ALU RegFile (Write)
dmem_req_addr dmem_req_data
1'b0 dmem_resp_val
ostallF stallD ostallD stallX ostallX stallM ostallM stallW ostallW
RVVI val order insn pc_wdata pc_rdata x_wdata x_wb
imem_req imem_resp dmem_req dmem_resp

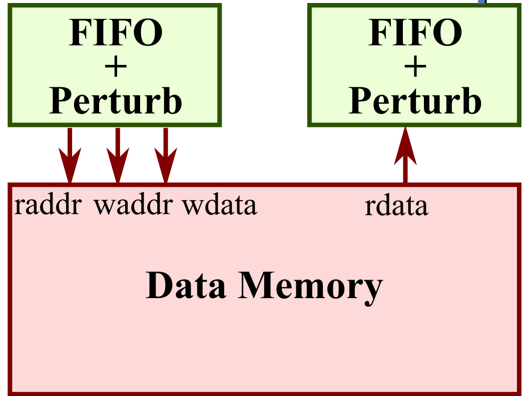


Correct:

```
assign ostallM = valM
  && (dmem_req_type_M == ld)
  && !dmem_resp_val;
```

Bug:

```
assign ostall_M = 1'b0;
```



- M stage does not stall on a pending dmem response
- Passes simulation tests with a behavioral data memory (no stalls)

Case Study #3: Correct Processor with Data Memory Stalls

Signals	Waves
Time	1, 100, 2, 200, 3, 300, 4, 400, 5, 500, 6, 600, 7, 700, 8, 800, 9, 900, 10, 1000
clk=0	
reset=0	
val_F=1	Fetch
val_D=1	Decode
val_X=1	Execute
val_M=1	Memory
stall_M=0	
val_W=1	Writeback
valid=1	
order[31:0]=0	00000000, 00000001, 00000002
x_wb[31:0]=0	00000000, 00000002, 00000000, 00000004, 00000008
x_wdata[31:0]=0	00000000, DEADBEEF, 00000000, DEADBEF0, DEADBEF1

```
1. lw x1, 4(x0)   # 0xDEADBEEF
2. lw x2, 8(x0)   # 0xDEADBEF0
3. lw x3, 12(x0)  # 0xDEADBEF1
```

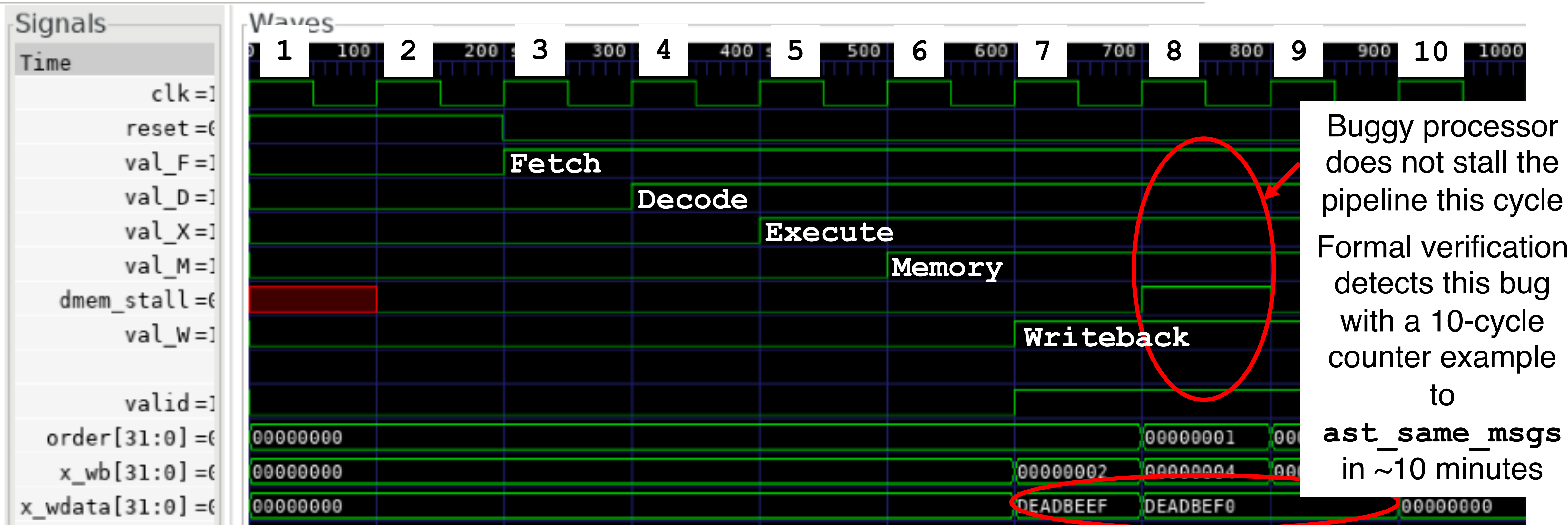
2nd instruction stalls the pipeline at cycle 7 and writes back the expected values at cycle 9

Case Study #3: Buggy Processor without Data Memory Stalls


```
1. lw x1, 4(x0)   # 0xDEADBEEF
2. lw x2, 8(x0)   # 0xDEADBEF0
3. lw x3, 12(x0)  # 0xDEADBEF1
```

The bug does not manifest when the data memory does not stall on the response path

Case Study #3: Buggy Processor with Data Memory Stalls




```
1. lw x1, 4(x0)   # 0xDEADBEEF
2. lw x2, 8(x0)   # 0xDEADBEF0
3. lw x3, 12(x0)  # 0xDEADBEF1
```

The data memory stalls on cycle 8, but the processor does not stall. This leads to a different sequence of values to be written back, which violates the stall invariant property on the RVVI

Related Work

- **Latency equivalence** [Carloni'99, Suhaib'06]
  - Latency equivalence: two designs produce the same sequence of informative events under a given sequence of input informative events
  - Stall invariant property concerns the informative events of only one design.
- **Latency equivalence checking in high-Level synthesis** [Piccolboni'19, Dai'21]
  - Use latency equivalence checking to improve the confidence of HLS
  - Our work focuses on proving stall invariant with formal verification
- **Bounded model checking** [Appenzeller'95, Biere'99, Bjesse'01, Clarke'01]
  - Bounded model checking has successfully identified design issues on modern hardware designs
  - We use a formal verification tool based on bounded model checking to identify violations of the specified stall invariant properties

Formal Verification of the Stall Invariant Property for Latency-Insensitive RTL Modules

This work explores a **practical** formal verification technique to prove the stall invariant property

- does not require test vectors to cover all possible stall events;
- reuses key verification components to facilitate harness construction; and
- produces counter example traces to help identify a bug's root cause.