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

```plaintext
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

```scala
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 Documentation; 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

```verilog
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

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
   \[\text{en}_A = \text{in}_\text{val} \neq 1 \text{ || stall}\]

2. **Input rdy bug**
   May drop input messages when the design is not ready to accept an input
   \[\text{in}_\text{rdy} \text{ not depend on stall/squash}\]

3. **Output val bug**
   May produce invalid output messages when the design is not valid to produce an output
   \[\text{out}_\text{val} \text{ not depend on stall/squash}\]

4. **Output rdy bug**
   May drop output messages when the output interface is not ready
   \[\text{en}_B = \text{out}_\text{rdy} \neq 1\]
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.

**Slide 7 of 24**

Motivation

Stall Invariant Property

Verification Approach

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

Motivation

Stall Invariant Property

Verification Approach

Case Studies
Latency-Insensitive Interface Terminology

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ingress</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Egress</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Producer
- Consumer
- Ingress LI Interface
- Egress LI Interface
- Design under Verification
- val msg rdy

Slide 9 of 24
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

#### 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).

#### 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).

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

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ingress</strong></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Egress</strong></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ingress</strong></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td>#</td>
<td>#</td>
<td>D</td>
</tr>
<tr>
<td><strong>Egress</strong></td>
<td>A</td>
<td>B</td>
<td>#</td>
<td>#</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Ingress</th>
<th>Egress</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>A B C D</td>
</tr>
<tr>
<td></td>
<td>A B C D</td>
</tr>
</tbody>
</table>

Reference Strict Behavior

<table>
<thead>
<tr>
<th>Ingress</th>
<th>Egress</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>A B C D</td>
</tr>
<tr>
<td></td>
<td>A B C D</td>
</tr>
</tbody>
</table>

Cannot be Stall Invariant

<table>
<thead>
<tr>
<th>Ingress</th>
<th>Egress</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>A B C D</td>
</tr>
<tr>
<td></td>
<td>A B C D</td>
</tr>
</tbody>
</table>

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.
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

<table>
<thead>
<tr>
<th>Strict DUV</th>
<th>Perturb. DUV</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>val</td>
</tr>
<tr>
<td>rdy</td>
<td>rdy</td>
</tr>
<tr>
<td>msg</td>
<td>msg</td>
</tr>
</tbody>
</table>

Strict Path

Perturbed Path
Formal Verification Harness of the Stall Invariant

- 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

- 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!
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

\[ \text{ast_same_msg}: \text{assert property ( @(posedge clk) disable iff (rst) ( (s_val \& p_val) \rightarrow (s_msg == p_msg)) ); } \]

Liveness Property
- What if the perturbed DUVs drop messages?
- Safety property will not detect them because the precondition will not be satisfied (i.e., \text{ast_same_msg} passes vacuously)
- The \text{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

\[ \text{ast_same_vals}: \text{assert property ( @(posedge clk) disable iff (rst) ( (s_val \& \neg p_val) \rightarrow s_eventually (s_val \& p_val) ) and (\neg s_val \& p_val) \rightarrow s_eventually (s_val \& p_val) )); } \]
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:
```verilog
assign rdyN = valW && rdyS && rdyE;
assign rdyW = valN && rdyS && rdyE;
```

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

- **Correct:**
  
  ```
  assign ostallM = valM && (dmem_req_type_M == ld) && !dmem_res_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

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

2\textsuperscript{nd} 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

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

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. \texttt{lw x1, 4(x0)}  # 0xDEADBEEF
2. \texttt{lw x2, 8(x0)}  # 0xDEADBEF0
3. \texttt{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.
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.