SystemVerilog Design: User Experience Defines Multi-Tool, Multi-Vendor Language Working Set

Ways Design Engineers Can Benefit from the Use of SystemVerilog Assertions

Stuart Sutherland, Sutherland HDL, Inc.
The primary goal of this presentation is to encourage RTL design engineers to take advantage the many ways in which SystemVerilog Assertions can help them!

Why SystemVerilog Assertions are important to me

- From my perspective as a SystemVerilog trainer
- From my perspective as a design and verification consultant

The main topics we will be discussing are:

- Part 1: A short tutorial on SystemVerilog Assertions
- Part 2: Assertions that Design Engineers should write
- Part 3: SystemVerilog constructs with built-in assertion-like checks
- Part 4: Simulation and Synthesis support for SVA
Part One

A Short Tutorial On SystemVerilog Assertions
What Is An Assertion?

- An assertion is a statement that a certain property must be true

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>req</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Design Specification:**
After the request signal is asserted, the acknowledge signal must arrive 1 to 3 clocks later.

- Assertions are used to:
  - Document design intent (e.g.: every request has an acknowledge)
  - Verify design meets the specification over simulation time
  - Verify design assumptions (e.g.: state value is one-hot)
  - Localize where failures occur in the design instead of at the output
  - Provide semantics for formal verification
  - Describe functional coverage points
  - And… requires clarifying ambiguities in spec
Embedded Verification Checking and Synthesis

- Without assertions, embedded checks must be hidden from Synthesis using conditional compilation or pragmas
  - Embedded checking can make RTL code look ugly!

```plaintext
if (mode) {
    // do true statements
} else {
    //synthesis translate_off
    if (mode == 0) {
        //synthesis translate_on
        // do the not true statements
    } else {
        //synthesis translate_on
        $display("mode is either an X or Z");
    }
} else {
    //synthesis translate_off
}
```

- SystemVerilog Assertions are easier, and synthesis ignores SVA

```plaintext
assert (!$isunknown(mode)) else $error("mode is either an X or Z");
if (mode) ... // do true statements
else ... // do not true statements
```

- SystemVerilog Assertions can be disabled in simulation

How many design engineers do you know that will add this much extra code to an if...else RTL statement?

This checking code is hidden from synthesis, but it is always active in simulation (not easy to disable for reset or for low-power mode)
SystemVerilog Has Two Types of Assertions

- **Immediate assertions** test for a condition at the current time
  - Similar to an `if...else` statement, but with assertion controls

```verilog
always_comb begin
  assert ($onehot(state)) else $error;
  case (state) ... // RTL code
end
```

reports an error if the `state` variable is not a one-hot value

- **Concurrent assertions** test for a sequence of events spread over multiple clock cycles
  - Execute as a background process in parallel with the RTL code

```verilog
a_reqack: assert property (@(posedge data_clk) req |-> ##[1:3] ack) else $error;
always_ff @(posedge clock) // RTL code
  if (data_ready) req <= 1;
  ...
```

reports an error if `ack` is not high within 1 to 3 clock cycles after `req`

## represents a “cycle delay” – a “cycle” in this example is from one posedge of `data_clk` to the next positive edge
Concurrent Assertions Can Span Multiple Clock Cycles

- **##n** specifies a fixed number of clock cycles

  ```
  request ##3 grant;
  ```
  After evaluating request, skip 2 clocks and then evaluate grant on the 3rd clock

- **##[min_count:max_count]** specifies a range of clock cycles
  - min_count and max_count must be non-negative constants

  ```
  request ##[1:3] grant;
  ```
  After evaluating request, grant must be true between 1 and 3 clocks later

- The dollar sign (\$) is used to specify an infinite number of cycles
  - Referred to as an “eventuality assertion”

  ```
  request ##[1:$] grant;
  ```

Design Spec: request must true at the current cycle; grant must become true sometime between 1 cycle after request and the end of time
Concurrent Assertions Run in the Background Throughout Simulation

- Concurrent assertions start a new check every clock cycle

```verilog
assert property (@(posedge clk)
                  req ###2 ack)
else $error;
```

- Assertions can be qualified with implication operators ( |->, |=>)
  - If a condition is true, the sequence is evaluated
  - If a condition is false, the sequence is not evaluated (a don’t care)

```verilog
assert property (@(posedge clk)
                 req |-> ###2 ack)
else $error;
```

- Antecedent — the expression before the implication operator
  - The evaluation only continues if the antecedent is true
- Consequent — the expression after the implication operator
- Vacuous success — when the antecedent is false, the check is not of interest, so evaluation is aborted without considering it a failure
Concurrent Assertions Only Sample Values on Clock Edges

- Concurrent assertions can sample **logic levels** on each clock cycle

```
assert property (@(posedge clk)
  req |-> ##2 ack)
else $error;
```

**GOTCHA** – the assertion passes even though **ack** did not change

**Is it OK for **ack** to be pulled high?**

**Writing assertions encourages engineers to read/clarify the spec!**

- Concurrent assertions can look for a change between the **last sampled value** and the **current sampled value**
  - **$rose** – returns true if there was a rising change in the current cycle
  - **$fell** – returns true if there was a falling change in the current cycle
  - **$changed** – returns true if there any change in the current cycle
  - **$stable** – returns true if there no change in the current cycle

```
assert property (@(posedge clk)
  $rose(req) |-> ##2 $rose(ack)
else $error;
```

**req and **ack** must transition**

```
assert property (@(posedge clk)
  req |-> ##2 ack)
else $error;
```

```
clk
req
ack
assertion
```

```
clk
req
ack
assertion
```

**accellera SYSTEMS INITIATIVE**
SVA Property Blocks and Sequence Blocks

- The argument to `assert property()` is a property specification
  - Properties are typically defined in a `property block`
  - Contains the definition of a sequence of events

```verilog
always @(posedge clock)
  if (State == FETCH)
    assert property (request ##3 grant) else $error;
```

- A simple sequence can be specified directly in the `assert property` call

```verilog
property pReq2E
  @(posedge clock) (request ##1 grant ##1 (qABC and qDE));
endproperty: pReq2E
```

- A complex sequence can be partitioned into named sequence blocks
  - Low level building blocks for sequence expressions

```verilog
sequence qABC;
  (a ##3 b ##1 c);
endsequence: qABC
```

```verilog
sequence qDE;
  (d ##[1:4] e);
endsequence: qDE
```

- The clock cycle is inferred from where the assertion is called
Immediate and Concurrent Assertion Pros and Cons

**Immediate Assertions**

- **Pros:**
  - Easy to write – simple syntax
  - Close to code being checked
  - Can check asynchronous values between clock cycles
  - Self-documenting code

- **Cons:**
  - Cannot be bound (next page)
  - Difficult to disable during reset or low-power
  - Must following good RTL practices to prevent race conditions (just like any programming statement)

**Concurrent Assertions**

- **Pros:**
  - Background task – define it and it just runs
  - Cycle based – no glitches between cycles
  - Can use binding (next page)
  - Works with simulation and formal verification

- **Cons:**
  - More difficult to define (and debug)
  - Can be far from code being checked
  - Cannot detect glitches

Which pros and cons are most important in your project?
There are many reasons signals might change more than once during a single clock cycle (a potential glitch):
- Combinatorial decoding, clock domain crossing, async reset, ...

When To Use Immediate Assertions, When To Use Concurrent Assertions

- Immediate assertions are programming statements that can evaluate values at any time
- Concurrent assertions are cycle based, and only evaluate values on a clock edge
SystemVerilog assertions can be defined in a separate file and:
- Bound to all instances of a design module or interface
- Bound to a specific instance of a design module or interface

Binding allows verification engineers to add assertions to a design without modifying the design files.

Binding allows updating assertions without affecting RTL code timestamps (which could trigger unnecessary synthesis runs).

Binding can also be used to bind in coverage and other functionality.

NOTE: Only concurrent assertions can be bound into other modules.
Embedded Versus Bound Assertions Pros and Cons

**Assertion Binding**

- **Pros:**
  - Do not need RTL file access permissions to add assertions
  - Adding assertions does not impact RTL file time-stamps

- **Cons:**
  - Assertions can be far from code being checked
  - RTL engineers must edit multiple files to add assertions while the RTL modes is being developed
  - Cannot use immediate assertion

**Assertions Embedded in RTL**

- **Pros:**
  - Close to the code being verified
  - Can use both concurrent and immediate assertions
  - Document designer’s assumptions and intentions
  - Assertion errors originate from same file as the failure

- **Cons:**
  - Adding/modifying an assertion could trigger automated regression or synthesis scripts
When To Embed Assertions, When To Bind-in Assertions

Sutherland HDL recommends ...

- **Design engineers** should **embed assertions** into the RTL code
  - Validate all assumptions (e.g. control inputs are connected)
  - Trap invalid data values where they first show up
  - Embedded assertions should be written at the same time the RTL code is being developed!

- **Verification engineers** should **add bound-in assertions**
  - Verify the design functionality matches the specification
  - Verify that corner cases work as expected (e.g.: FIFO full)
  - Verify coverage of critical data points
  - By using binding:
    - There is no need to check out and modify the RTL model files
    - Adding assertions not affect RTL file time stamps

There can be exceptions to this guideline – you get paid the big money to figure out which way of specifying assertions is best for your projects!
Part Two

Assertions That Design Engineers Should Write
Design Engineers Should Add Assertions to RTL!

- RTL models *assume* inputs and other values are valid
  - Input ports are connected (no floating input values)
  - Control signals are never a logic X
  - State machine encoding is a legal value
  - Data values are in an expected range
  - Parameter redefinitions meet design requirements

- These assumptions *can be* *should be* verified using assertions
  - Most of these can be done with simple 1-line assertions

- The examples on the next few pages show how to:
  - Validate assumptions on reset values
  - Validate assumptions regarding value ranges
  - Validate assumptions on pulse widths
  - Validate parameter values after parameter redefinition
  - Eliminate problems with X-pessimism and X-optimism
Validating Assumptions On Critical Control Signals

- **RTL models assume** that control signals have known values
  - Reset is either 0 or 1, Enable is either 0 or 1, etc.

- **A 1-line immediate assertion or simple concurrent assertion**\(^1\) can check this assumption!
  - Catch problems when and where they occur

```verilog
module data_reg (input resetN, ...);
    always_ff @(posedge clk) begin
        if (!resetN) q <= 0;
        else q <= d;
    end

Assumes resetN input is properly connected (an unconnected reset will set q <= d every clock cycle)

1Immediate and concurrent assertions handle glitches differently – use the type that best meets the needs of your project!

Something went wrong right here!
```
RTL code often assumes data values are within a valid range
- Out of range values propagate as a functional bug
- Can be difficult to detect
- Might not be detected until downstream in both logic and clock cycles

A 1-line immediate assertion can check that values are within a required range!

```verilog
testbench # (1-line immediate assertion)
assert (b inside {[1:3]}) else $error("b is out of range for shift");
result <= a >> b;
end
endcase
```
Validating Assumptions On Pulse Widths

- RTL models sometimes assume certain signals remain true for some number of clock cycles
  - So that reset to propagate through multiple stages of logic
  - To allow devices to enter or leave low-power mode

- A simple concurrent assertion can check pulse widths!

```verilog
module jcounter (input logic clk, rstN, output logic [3:0] q);

always_ff @(posedge clk) begin
  q[0] <= ~q[3] & rstN;
  q[1] <= q[0];
  q[2] <= q[1];
  q[3] <= q[2];
end
endmodule
```

Assumes `rstN` input meets the pulse width required by this model

```
assert property (@(posedge clk) $fell(rstN) |-> !rstN[*4])
else $error("rstN did not remain low for at least 4 clock cycles");
```
Validating Parameter Values After Parameter Redefinition

- Parameterized models **assume** exterior code redefines the parameters to viable values.
- An *elaboration-time assertion* can ensure redefined parameters have expected values!

```verilog
module muxN // 2:1 MUX (S == 1) or 4:1 MUX (S == 2)
#(parameter N=8, S=1)  // Assumes S is only redefined to be 1 or 2
(output logic [N-1:0] y,
 input logic [N-1:0] a, b,
 input logic [N-1:0] c=0, d=0, // c, d have default value if unconnected
 input logic [S-1:0] sel
);

    generate
        if (S inside {[1:2]}); else $fatal(0,"S must be 1 or 2");
    endgenerate

    always_comb begin
        ...
    end
endmodule
```

 Uh Oh, S was redefined to a value that won't work!

(if...else is used in generate blocks instead of assert...else)
Eliminating X-Pessimism and X-Optimism Gotchas

- RTL models are notorious for hiding problems involving X values
  - A non-X value is propagated instead of a logic X
    - Verification must determine the non-X value is incorrect functionality
    - Bugs must be traced back through logic and clock cycles to figure out where the problem first occurred

- A 1-line immediate assertion\(^1\) can trap X values!
  - Do not need to detect and debug resulting functional bugs

```verilog
always_comb begin
  assert final (!$isunknown(sel)) else $error("sel is X or Z");
  if (sel) y = a;
  else y = b;
end
```

\(^1\)Most immediate assertions can also be written as concurrent assertions, but there is a difference on how the assertion types handle glitches – use the assertion type that best meets the needs of your project!
Self-Checking Interfaces

- An RTL interface port can be used to model bus protocols
  - Encapsulates the bus-related signals into a single port

- Embedded assertions in an interface can automatically detect protocol errors
  - Protocol violations are detected at the moment they occur

interface AMBA_APB;
logic [31:0] addr;
logic [7:0] rdata, wdata;
logic selx, enable, write;

property p_sel_enable;
 @(posedge clk)
 $rose(selx) |-> ##1 $rose(enable);
endproperty: p_sel_enable
assert property (p_sel_enable);
... // additional protocol checks
endinterface
Part Three

SystemVerilog Constructs With Built-in Assertion-like Checking
SystemVerilog Adds Better RTL Constructs to Verilog

- Traditional Verilog will allow writing code with functional errors
  - Allows engineers to model faulty behavior in order to prove a design will not work correctly
  - Puts a burden on Design Engineers to avoid dysfunctional code
  - Puts a burden on Verification Engineer to find dysfunctional code

- SystemVerilog adds constructs with built-in error checking!
  - Self-checking RTL modeling blocks
  - Self-checking decision statements
  - Self-checking assignment statements

- Using these constructs is like getting free assertions!
  - Can detect and prevent many types of functional bugs before synthesis
Self-Checking RTL Modeling Blocks

- Verilog *always* procedures model all types of design logic
  - Synthesis must “infer” (guess) whether an engineer intended to have combinational, latched or sequential functionality

- SystemVerilog has hardware-specific *always* procedures: always_comb, always_latch, always_ff
  - Documents designer intent
  - Enforces several synthesis RTL rules
  - Synthesis can check against designer intent

```verilog
always @(mode)
  if (!mode)
    o1 = a + b;
  else
    o2 = a - b;
```

Warning: test.sv:5:
Netlist for always_comb block contains a latch

Free, built-in code checking - I like this!
Self-Checking Decision Statements

- Verilog only defines simulation semantics for decision statements:
  - Evaluate sequentially; only the first matching branch is executed.

- Specifying synthesis `parallel_case` and `full_case` pragmas causes gate-level optimizations:
  - Evaluate decisions in parallel, do Karnaugh mapping, etc.

**WARNING:** These optimizations are **NOT verified** in simulation!

- SystemVerilog adds `unique`, `unique0` and `priority` decisions:
  - Enable synthesis `parallel_case` and/or `full_case` pragmas.
  - Enable run-time simulation checking for when the decision might not work as expected if synthesized with the pragma.

```verilog
always_comb
    unique case (state)
    ...
endcase
```

- **Will get simulation warnings** if `state` matches multiple branches (not a valid `parallel_case`)
- **Will get simulation warnings** if `state` doesn’t match any branch (not a valid `full_case`)
Self-Checking Assignment Statements

parameter [2:0]
    WAIT = 3'b001,
    LOAD = 3'b010,
    DONE = 3'b001;

parameter [1:0]
    READY = 3'b101,
    SET   = 3'b010,
    GO    = 3'b110;

reg [2:0] state, next_state;
reg [2:0] mode_control;

always @(posedge clk or negedge rstN)
    if (!resetN) state <= 0;
    else state <= next_state;

always @(state) // next state decoder
    case (state)
        WAIT : next_state = state + 1;
        LOAD : next_state = state + 1;
        DONE : next_state = state + 1;
    endcase

always @(state) // output decoder
    case (state)
        WAIT : mode_control = READY;
        LOAD : mode_control = SET;
        DONE : mode_control = DONE;
    endcase

SystemVerilog adds enumerated types

enum logic [2:0]
    {WAIT = 3'b001,
    LOAD = 3'b010,
    DONE = 3'b001}
state, next_state;

enum logic [1:0]
    {READY = 3'b101,
    SET   = 3'b010,
    GO    = 3'b110}
mode_control;

always_ff @(posedge clk or negedge rstN)
    if (!resetN) state <= 0;
    else state <= next_state;

always_comb // next state decoder
    case (state)
        WAIT : next_state = state + 1;
        LOAD : next_state = state + 1;
        DONE : next_state = state + 1;
    endcase

always_comb // output decoder
    case (state)
        WAIT : mode_control = READY;
        LOAD : mode_control = SET;
        DONE : mode_control = DONE;
    endcase

6 functional bugs (must detect, debug and fix)

7 syntax errors (compiler finds all the bugs)
Part Four

Simulation and Synthesis Support for SystemVerilog Assertions
Simulation and Synthesis Support for Assertions

- Simulation should execute assertions; Synthesis should ignore

<table>
<thead>
<tr>
<th>Assertion Construct</th>
<th>Vendor A</th>
<th>Vendor B</th>
<th>Vendor C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sim</td>
<td>Synth</td>
<td>Sim</td>
</tr>
<tr>
<td>Embedded Immediate Assertions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Embedded Concurrent Assertions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Property Blocks</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sequence Blocks</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Disable Assertion During Reset</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deferred Immediate Assertions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Let Statements</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Checker Statement</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validate Reset Example</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Validate Value Range Example</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Validate Pulse Width Example</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Validate Parameters</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>always_comb with Latch Logic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Enumerated Types with Faulty Logic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Summary

- SystemVerilog Assertions really do work!
  - An effective way to verify many aspects of design functionality
  - Find errors that functional verification might miss

- RTL Design Engineers should embed assertions that validate assumptions directly into RTL code as the code is being written
  - Embed relatively simple immediate and concurrent assertions
  - Use RTL modeling constructs with built-in assertion-like checking
  - Synthesis compilers properly ignore embedded assertions

- There are big advantages to RTL designers specifying assertions
  - Validate assumptions on which the RTL model depends
  - Localizes where functional problem occurred
  - Clarify specification ambiguities
  - Help to avoid RTL modeling gotchas
Thank you!
SystemVerilog Design: User Experience Defines Multi-Tool, Multi-Vendor Language Working Set

Experience from Four Years of SVD Adoption

Junette Tan, PMC
Agenda

- Motivating Factors for SV Adoption
- Migration Challenges
- Benefits Gained
Motivating Factors for SV Adoption

- **Align with industry momentum on Verilog**
  - New technologies being introduced and tested in Verilog
  - No use wasting resources trying to push developments in VHDL

- **Target one HDL/HVL language for the company**
  - Mixed language usage increases complexity

- **Powerful new constructs in SystemVerilog**
  - User-defined enumerations, packed structs, interfaces increase code readability
    - `always_comb` and `always_ff` procedural blocks decrease coding errors
  - Code compactness, design reuse, scalability = 10-20% increase in productivity
Timeline of Adoption

2005
- IEEE 1800-2005 released

2010
- SV training and coding guidelines created

2011
- First project developed in SV and company-wide deployment
Migration Challenges
Migration Challenges

- **Resistance to change**
  - Promoted migration through company-wide presentation
  - Organized multiple instructor-led training sessions
  - Created self-paced online training modules
  - Migrated re-use components to SystemVerilog
Migration Challenges

- Lots of VHDL knowledge, sparse Verilog knowledge
  - Coordinated effort with training vendor to create coding guidelines and custom training, took conservative approach knowing tool limitations (e.g., no interfaces)
  - Provided VHDL to SystemVerilog examples
  - Created library of SystemVerilog code for design community

```vhdl
module shift_register #(
    parameter NUM_ELEMENTS = 8,
    parameter NUM_BITS = 4
)(
    output logic [NUM_ELEMENTS-1:0][NUM_BITS-1:0] data_out,
    ...
);
always_ff @ (posedge clk or negedge rstb)
begin : sr_logic
  if ( rstb != 1'b1 ) begin
    data_out <= '{default:0};
  end
  ...
end : sr_logic
```

- Use `always_ff` for sequential logic
- Use logic for everything
- Use assignment patterns for arrays
Migration Challenges

Troublesome tools
- Early adoption meant working closely with EDA vendor to flush out all bugs:
  • Packages
  • Size casting with parameters
  • Assignment patterns
  • Enumerated types
  • Packed structs and unions
  • Multi-dimensional arrays
  • Mixed language usage
- Weekly calls to drive all issues to closure before they could become gates in the project schedule
- Unexpected number of issues faced with all tools in the design flow (e.g., simulator, emulation tool, FPGA tool, synthesis tool, equivalence checker)
Migration Challenges

- **SystemVerilog Gotchas**
  - Compiler directive `default_nettype` was not interpreted consistently across tools

  ```verilog
  `default_nettype none
  module bad_example (  
    input logic a,  
    input logic b,  
    ...  
  );
  no net type declared!

  `default_nettype none
  module correct_example (  
    input wire logic a,  
    input wire logic b,  
    ...  
  );
  net type wire specified for input ports
  ```

**Why bother with `default_nettype`?**

```verilog
cpu cpu_inst (  
  ...  
  .halted(halter),  
  ...  
);
```

 typo would result in incorrect implicit net declaration!
Migration Challenges

- **SystemVerilog Gotchas**
  - `int` as reserved word caused grief when connecting to VHDL ports named “int”

```vhdl
vhdl_block vhdl_inst (  
    ...  
    .int(irq),  
    ...  
);
```

can’t use “int” as port name, so must create SV-friendly wrapper for VHDL code
# Metrics

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projects using SVD to date (gate count including reuse IP)</td>
<td>7 projects</td>
</tr>
<tr>
<td></td>
<td>(1043.8M gates)</td>
</tr>
<tr>
<td>Number of SV files in 1(^{st}) Project</td>
<td>2788</td>
</tr>
<tr>
<td>Number of SV issues reported</td>
<td>120</td>
</tr>
<tr>
<td>Number of engineers trained in SV</td>
<td>150</td>
</tr>
</tbody>
</table>
Current Challenges

- **Interfaces**
  - Still have yet to take advantage of connectivity gains that interfaces can provide

```verilog
interface apb_intf;
    apb_addr_t paddr;
    logic pwrite;
    apb_data_t pwdata;
    apb_data_t prdata;
...  
endinterface

module apb_master (apb_intf intf,
    ...
);

module apb_slave (apb_intf intf,
    ...
);

module top_level (...);

apb_intf intf_inst();
apb_master master_inst (intf(intf_inst),
    ...
);
apb_slave slave_inst (intf(intf_inst),
    ...
);
endmodule
```

Interfaces must be instantiated

Interface connections are bi-directional
Current Challenges

- Interfaces
  - Currently use structs as a workaround

```verilog
package apb_pkg;
    ...
    typedef struct packed {
        apb_addr_t paddr;
        logic pwrite;
        apb_data_t pwdata;
        ...
    } apb_ctrl;

    typedef struct packed {
        logic pready;
        apb_data_t prdata;
        ...
    } apb_resp;
endpackage

module apb_master ( 
    input wire apb_pkg::apb_resp resp,
    output apb_pkg::apb_ctrl ctrl,
    ...
);

module apb_slave ( 
    input wire apb_pkg::apb_ctrl ctrl,
    output apb_pkg::apb_resp resp
    ...
);
```

Must specify correct direction!

Signals grouped by direction
Current Challenges

- **Low Power**
  - Simulation tool doesn’t support adding isolation when the DUT port is connected to logic or unpacked array port in the testbench
    - “bit, byte, shortint, int, longint, user-defined types, enumerated data types, structures, unions, [and so on,] and constructs, such as clocking blocks, program blocks, classes, packages, and so on, are not supported and cannot be part of a power-down domain” – EDA vendor
    - Logic datatypes had to be converted to wire or reg
    - Unpacked arrays had to be split up
Benefits Gained

- **Aligned with industry momentum**
  - Many vendor issues, but no project delays due to conservative migration

- **One HDL to rule them all**
  - All reuse IP successfully migrated to SV, no need to support multiple languages
  - All new IP designed in SV

- **Productivity gains**
  - Code compaction achieved with `always_comb` (no more sensitivity lists) and .name shorthand notation
    ```verilog
cpu cpu_inst (.
    .clk,
    .rstb,
    .name shorthand notation looks clean
    and reduces chance of error
    ```
  - Abstraction achieved with new `logic` datatype (no more `wire` or `reg`)
  - No more configuration files!
  - Simulation performance increased
Thank you!
SystemVerilog Design: User Experience Defines Multi-Tool, Multi-Vendor Language Working Set

No Excuses for Not Using SystemVerilog in Your Next Design

Mike Schaffstein, Qualcomm
Who is Mike Schaffstein?

- 20+ years of design, architecture and methodology experience
- Introduced limited SV coding to design at previous company in 2010
  - Chip in production
- Last 18 months with Qualcomm® Adreno™ graphics team
  - Created initiative to use SV coding in design
  - Used a larger portion of the language than at previous company
  - Code exposed to the full tool flow
Remember 2005?

2005

Flip Phone

versus

Bar Phone

newly ratified…

SystemVerilog

IEEE 1800-2005!

2015

FINGERPRINT

SENSOR

HEARTRATE

MONITOR

LTE

3D GAMING

MULTICORE

HD VIDEO

APPs

GHz+GB

4K DISPLAY

(still using Verilog-2001☹️)
You should emerge from this tutorial with…

- No excuse not to use SystemVerilog in your next design
- A clear idea of what to expect from the process through tapeout
- A template for how to proceed

Phase 1: APPROVAL
Phase 2: PLANNING
Phase 3: DESIGN
Phase 4: TOOLS
APPROVAL: Justification

- **Communicate your intent**
  - A stealth effort will likely backfire

- **Prepare a rock-solid argument for why this is good for your product**
  - Higher productivity
  - Time to market
  - Fewer bugs
  - Flexible, future proof coding

- **Build consensus**
  - Find progressive, like-minded people to back you
  - Use strength in numbers to sway others
I want to use SystemVerilog in our next project.

Of course not, the spec has been around for a decade.

Not exactly.

Will it jeopardize our revenue stream?

So everyone is using it?

We’ll need a plan. I’ll set up a bunch of meetings.
Is this SystemVerilog like trying to synthesize SystemC?

No, it’s the same level of abstraction as Verilog. It’s just a more efficient language for coding hardware constructs.

Will our area be impacted?

Nope.

Will our timing blow up?

No again.
There may be some hiccups the first time through.

Maybe we can restrict the language to a simple set of constructs?

Or maybe we can limit SystemVerilog to one block as a pilot program?

But we can’t push out the schedule.

Not bad, keep talking.

Good, I like having options.
Do our tools support SystemVerilog?

What if the tools produce bad results and our chip is DOA?

So if functional tool issues aren’t a big concern, where are the tools weakest?

Well, they all say they do.

The tools have supported similar VHDL constructs for years. This is familiar territory for them.

SystemVerilog language parsing.
PLANNING: Educate Your Team

- IEEE 1800-2012
- Sutherland HDL
- Your synthesis tool’s SystemVerilog user’s guide
- DV team members
- Web searches and web sites
- Sample code in this presentation
Define Your Synthesizable Subset

- Arguably the most important step!
- Don’t bite off more than you can chew
  - Know your design team: coding style, diligence, patience with tools, etc.
  - Keep it simple if need be
- Stick to IEEE 1800-2005 constructs for now
  - 1800-2009, 1800-2012 constructs are too new
- Use your synthesis tool’s documentation
My Synthesizable Subset

- typedef
- logic
- enum
- struct
- package
- interface
- always_ff/always_comb
- $clog2, $bits, $size,...
- Operator enhancements
- Packed arrays
- Arrays as ports
- Literal enhancements
- Casting
- Wildcard and .name ports
- Sets
- Streaming operators

Roll out more in the next generation!
DESIGN: Where and When?

- Code all the new features/blocks with SystemVerilog

- What about all the legacy code?
  - If it isn’t broken, don’t fix it
  - But where your new SV code interacts with your old code consider updating
  - And anywhere you think SV will make the code easier to maintain long term

- Consider sharing SV package definitions with DV
DESIGN: Exploit the language

- Progressive language demands progressive coding
  - Use `typedef` datatypes everywhere you can
  - Use `package` to organize shared types
  - Put simple naming guidelines in place

- Review any code examples you can find…and use the good stuff
module apb_top ( input logic clk, input logic ares, apb_if.slave host );

import apb_pkg::*;

apb_if slave[SLAVE_NUM]();

apb_bridge bridge (.master(host), .slave);

reg_temps temps (.apb(slave[SLAVE_TEMPS]), .temps(), .*);
reg_bd_month bdmonth (.apb(slave[SLAVE_BD_MONTH]), .bdMonth(), .*);
reg_day_mask daymask (.apb(slave[SLAVE_DAY_MASK]), .dayMask(), .*);

endmodule

Check out the bonus material for the full code example
module apb_top ( input logic clk, input logic ares, 
apb_if.slave host );

import apb_pkg::*;
apb_if slave[SLAVE_NUM]();
apb_bridge bridge (.master(host), .slave);
reg_temps temps  (.apb(slave[SLAVE_TEMPS]),    .temps(),   .*);
reg_bd_month bdmonth(.apb(slave[SLAVE_BD_MONTH]), .bdMonth(), .*);
reg_day_mask daymask(.apb(slave[SLAVE_DAY_MASK]), .dayMask(), .*);
output host_ready,
output [31:0] host_rdata

endmodule
module apb_top ( input logic clk, input logic ares, apb_if.slave host );

import apb_pkg::*;

apb_if slave[SLAVE_NUM]();

localparam SLAVE_TEMPS    = 0,
SLAVE_BD_MONTH = 1,
SLAVE_DAY_MASK = 2,
SLAVE_NUM      = 3;

wire [19:2]   slave_addr [SLAVE_NUM];
wire [31:0]   slave_wdata [SLAVE_NUM];
wire          slave_write [SLAVE_NUM];
wire          slave_enable [SLAVE_NUM];
wire          slave_ready [SLAVE_NUM];
wire [31:0]   slave_rdata [SLAVE_NUM];

endmodule
module apb_top ( input logic clk, input logic ares, apb_if.slave host );
import apb_pkg::*;
apb_if slave[SLAVE_NUM]();
apb_bridge bridge (.master(host), .slave);
reg_temps temps (.apb(slave[SLAVE_TEMPS]), .temps(), .*);
reg_bd_month bdmonth (.apb(slave[SLAVE_BD_MONTH]), .bdMonth(), .*);
reg_day_mask daymask (.apb(slave[SLAVE_DAY_MASK]), .dayMask(), .*);
endmodule
EXAMPLE: Port Optimizations (continued)

```verilog
module apb_top ( input logic clk, input logic ares,
                   apb_if slave host );

import apb_pkg::*;

apb_if slave[SLAVE_NUM]();
apb_bridge bridge (.master(host), .slave);
reg_temps temps  (.apb(slave[SLAVE_TEMPS]),    .temps(),   .*);
reg_bd_month bdmonth(.apb(slave[SLAVE_BD_MONTH]), .bdMonth(), .*);
reg_day_mask daymask(.apb(slave[SLAVE_DAY_MASK]), .dayMask(), .*);

endmodule
```
module apb_top

  input clk,
  input ares,
  input [19:2] host_addr,
  input [31:0] host_wdata,
  input host_write,
  input host_enable,
  output host_ready,
  output [31:0] host_rdata;

localparam SLAVE_TEMPS    = 0,
SLAVE_BD_MONTH = 1,
SLAVE_DAY_MASK = 2,
SLAVE_NUM      = 3;

wire [19:2]   slave_addr [SLAVE_NUM];
wire [31:0]   slave_wdata [SLAVE_NUM];
wire          slave_write [SLAVE_NUM];
wire          slave_enable [SLAVE_NUM];
wire          slave_ready [SLAVE_NUM];
wire [31:0]   slave_rdata [SLAVE_NUM];

apb_bridge bridge (.master_addr(host_addr),
                   .master_wdata(host_wdata),
                   .master_write(host_write),
                   .master_enable(host_enable),
                   .master_ready(host_ready),
                   .master_rdata(host_rdata),
                   .slave0_addr(slave_addr[SLAVE_TEMPS]),
                   .slave0_wdata(slave_wdata[SLAVE_TEMPS]),
                   .slave0_write(slave_write[SLAVE_TEMPS]),
                   .slave0_enable(slave_enable[SLAVE_TEMPS]),
                   .slave0_ready(slave_ready[SLAVE_TEMPS]),
                   .slave0_rdata(slave_rdata[SLAVE_TEMPS]),
                   .slave1_addr(slave_addr[SLAVE_BD_MONTH]),
                   .slave1_wdata(slave_wdata[SLAVE_BD_MONTH]),
                   .slave1_write(slave_write[SLAVE_BD_MONTH]),
                   .slave1_enable(slave_enable[SLAVE_BD_MONTH]),
                   .slave1_ready(slave_ready[SLAVE_BD_MONTH]),
                   .slave1_rdata(slave_rdata[SLAVE_BD_MONTH]),
                   .slave2_addr(slave_addr[SLAVE_DAY_MASK]),
                   .slave2_wdata(slave_wdata[SLAVE_DAY_MASK]),
                   .slave2_write(slave_write[SLAVE_DAY_MASK]),
                   .slave2_enable(slave_enable[SLAVE_DAY_MASK]),
                   .slave2_ready(slave_ready[SLAVE_DAY_MASK]),
                   .slave2_rdata(slave_rdata[SLAVE_DAY_MASK]));

reg_temps temps (.clk(clk),
                  .ares(ares),
                  .temps(),
                  .apb_addr(slave_addr[SLAVE_TEMPS]),
                  .apb_wdata(slave_wdata[SLAVE_TEMPS]),
                  .apb_write(slave_write[SLAVE_TEMPS]),
                  .apb_enable(slave_enable[SLAVE_TEMPS]),
                  .apb_ready(slave_ready[SLAVE_TEMPS]),
                  .apb_rdata(slave_rdata[SLAVE_TEMPS]));

reg_bd_month bdmonth (.clk(clk),
                       .ares(ares),
                       .bdMonth(),
                       .apb_addr(slave_addr[SLAVE_BD_MONTH]),
                       .apb_wdata(slave_wdata[SLAVE_BD_MONTH]),
                       .apb_write(slave_write[SLAVE_BD_MONTH]),
                       .apb_enable(slave_enable[SLAVE_BD_MONTH]),
                       .apb_ready(slave_ready[SLAVE_BD_MONTH]),
                       .apb_rdata(slave_rdata[SLAVE_BD_MONTH]));

reg_day_mask daymask (.clk(clk),
                      .ares(ares),
                      .dayMask(),
                      .apb_addr(slave_addr[SLAVE_DAY_MASK]),
                      .apb_wdata(slave_wdata[SLAVE_DAY_MASK]),
                      .apb_write(slave_write[SLAVE_DAY_MASK]),
                      .apb_enable(slave_enable[SLAVE_DAY_MASK]),
                      .apb_ready(slave_ready[SLAVE_DAY_MASK]),
                      .apb_rdata(slave_rdata[SLAVE_DAY_MASK]));

endmodule

module apb_top ( input logic clk, input logic ares,
                 apb_if.slave host );

import apb_pkg::*;

apb_if slave[SLAVE_NUM]();

apb_bridge bridge (.master(host), .slave);

reg_temps temps  (.apb(slave[SLAVE_TEMPS]),
                   .temps(),
                   .);

reg_bd_month bdmonth(.apb(slave[SLAVE_BD_MONTH]),
                      .bdMonth(),
                      .);

reg_day_mask daymask (.apb(slave[SLAVE_DAY_MASK]),
                      .dayMask(),
                      .);

endmodule
module apb_bridge ( apb_if.slave master, apb_if.master slave[SLAVE_NUM] );

// code ommitted...

always_comb begin
    master.resp = '{ default: 0 };
    foreach ( slave[i] )
        master.resp = master.resp | slave[i].resp;
end

endmodule
module apb_bridge ( apb_if.slave master,
                    apb_if.master slave[SLAVE_NUM] );

// code ommitted...
apb_resp_s resp [$size(slave)];

for ( genvar i = 0; i < $size(slave); i++ ) begin : gen_select
    always_comb resp[i] = slave[i].resp;
end

assign master.resp = resp.or;
endmodule

assign master_ready = slave0_ready | slave1_ready | slave2_ready;
assign master_rdata = slave0_rdata | slave1_rdata | slave2_rdata;
enum logic [23:0] {
    RED   = 24'hff0000,
    GREEN = 24'h00ff00,
    BLUE  = 24'h0000ff
} color, next_color;

always_comb next_color = color.next;

always @*
case (color)
    RED:    next_color = GREEN;
    GREEN:  next_color = BLUE;
    default: next_color = RED;
endcase

localparam [23:0]
    RED   = 24'hff0000,
    GREEN = 24'h00ff00,
    BLUE  = 24'h0000ff;

reg [23:0] color, next_color;
EXAMPLE: State Machine *enum* (continued)

Waveform viewers understand enumerations

<table>
<thead>
<tr>
<th>Verilog</th>
<th>SystemVerilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>color</td>
</tr>
<tr>
<td>next_color</td>
<td>next_color</td>
</tr>
<tr>
<td>FF0000</td>
<td>RED</td>
</tr>
<tr>
<td>00FF00</td>
<td>GREEN</td>
</tr>
<tr>
<td>0000FF</td>
<td>BLUE</td>
</tr>
<tr>
<td>0000FF</td>
<td>RED</td>
</tr>
<tr>
<td>00FF00</td>
<td>GREEN</td>
</tr>
<tr>
<td>FF0000</td>
<td>BLUE</td>
</tr>
</tbody>
</table>
TOOLS

- **Know your tool flow**
  - Understand which tools digest RTL directly
  - How good is your tool vendor support?
  - Let your tool vendors’ support people know about this undertaking

- **Build flow and setup files may need adjustments**
  - SV switches
    - `package / interface` compile order dependencies

- **Use the latest versions if possible**
TOOLS: Dealing with SV Issues

- Is it a tool bug, a documented tool limitation or bad syntax?
- There is [almost] always a workaround
  - The trick is finding a syntax that all the tools digest
- Log tool issues
  - Use `ifdef` in the code to show workaround code next to broken code
  - Keep a database with code snippet, tool version, error #, error message
- Run as many tools as possible at the block level
  - Finds issues earlier
- Work with vendors to resolve issues
  - But don’t expect timely bug fixes during the project cycles
TOOLS: Hot Spots for SV Issues

- Module parameter port lists and optional `parameter` keyword
- Use of an `enum` where a `localparam int` would normally be
- Tools get iffy when lots of SV nested together

```vhdl
assign month = anIF[anENUM].aStruct.month_enum.next;
```

- See the workaround examples in the conference handouts
SV Parsing Issues by Tool Category

Average SV Parsing Issues

- **2H 2012**
  - Frontend: 2
  - Backend: 6.5
  - Emulation: 6

- **2H 2014**
  - Frontend: 4.25
  - Backend: 1.25
  - Emulation: 4

[4 tools] [4 tools] [2 tools]

Accellera Systems Initiative
Current SV Parsing Issue Landscape

2H 2014 TOOL VERSIONS USED

Number of Tools Exhibiting SV Parsing Issue

SV Parsing Issue Reference Number

[10 tools]
Need one more reason?

- It’s good for you too!

- SystemVerilog competency is a **VALUABLE INDUSTRY SKILL**

- Stay **COMPETITIVE, stay RELEVANT** – don’t be a designasaur

- Be the **INNOVATOR** at your company

Check out the bonus material with code examples and tool workarounds
Thank you!
SystemVerilog Design: User Experience Defines Multi-Tool, Multi-Vendor Language Working Set

APB Example Code
Tool vendors and designers: Use this code to screen your tools

APB EXAMPLE CODE
APB Example

HOST

APB BRIDGE

TEMPS REGS

DAY MASK REGS

BD MONTH REGS
package apb_pkg;

// AMBA 3 APB Protocol Specification v1.0

// Addr width unspecified by APB
// so make a type for flexibility.
typedef logic [31:0] apb_data_t;

// APB request (master to slave).
typedef struct packed {
apb_addr_e addr;
logic enable;
logic write;
apb_data_t wdata;
} apb_req_s;

// APB response (slave to master).
typedef struct packed {
logic ready;
apb_data_t rdata;
} apb_resp_s;

typedef enum int {
SLAVE_TEMPS,
SLAVE_BD_MONTH,
SLAVE_DAY_MASK,
SLAVE_NUM
} SLAVE_e;

typedef enum apb_addr_t {
REG_BD_MONTH_MOM        = 'h40,
REG_BD_MONTH_DAD        = 'h41,
REG_BD_MONTH_DAUGHTER   = 'h42,
REG_BD_MONTH_SON        = 'h43,
REG_DAY_MASK            = 'h50,
REG_TEMP_JAN            = 'h61,
REG_TEMP_FEB            = 'h62,
REG_TEMP_MAR            = 'h63,
REG_TEMP_APR            = 'h64,
REG_TEMP_MAY            = 'h65,
REG_TEMP_JUN            = 'h66,
REG_TEMP_JUL            = 'h67,
REG_TEMP_AUG            = 'h68,
REG_TEMP_SEP            = 'h69,
REG_TEMP_OCT            = 'h6a,
REG_TEMP_NOV            = 'h6b,
REG_TEMP_DEC            = 'h6c
} apb_addr_e;

typedef enum apb_addr_t {
REG_BD_MONTH_MOM        = 'h40,
REG_BD_MONTH_DAD        = 'h41,
REG_BD_MONTH_DAUGHTER   = 'h42,
REG_BD_MONTH_SON        = 'h43,
REG_DAY_MASK            = 'h50,
REG_TEMP_JAN            = 'h61,
REG_TEMP_FEB            = 'h62,
REG_TEMP_MAR            = 'h63,
REG_TEMP_APR            = 'h64,
REG_TEMP_MAY            = 'h65,
REG_TEMP_JUN            = 'h66,
REG_TEMP_JUL            = 'h67,
REG_TEMP_AUG            = 'h68,
REG_TEMP_SEP            = 'h69,
REG_TEMP_OCT            = 'h6a,
REG_TEMP_NOV            = 'h6b,
REG_TEMP_DEC            = 'h6c
} apb_addr_e;

endpackage
package other_pkg;

typedef logic [3:0] MONTH_t;
typedef enum MONTH_t {
  JAN, FEB, MAR, APR, MAY, JUN,
  JUL, AUG, SEP, OCT, NOV, DEC
} MONTH_e;

typedef enum int { LO, HI } RANGE_e;

typedef enum int {
  MOM, DAD, DAUGHTER, SON,
  FAMILY_SIZE
} FAMILY_e;

typedef logic [7:0] temp_t;

endpackage
interface apb_if;

import apb_pkg::*;

logic    sel;
apb_req_s req;
apb_resp_s resp;

function automatic logic WriteReg( input apb_addr_e addr );
    WriteReg = sel & req.write & req.enable & (req.addr == addr);
endfunction

function automatic logic ReadReg( input apb_addr_e addr );
    ReadReg = sel & ~req.write & req.enable & (req.addr == addr);
endfunction

modport master( output req, sel, input  resp );
modport slave ( input  req, sel, output resp,
    import WriteReg, import ReadReg );

endinterface
module reg_day_mask ( input logic clk, input logic ares, apb_if.slave apb, output logic [31:1] dayMask );

import apb_pkg::*;

always_ff @( posedge clk or posedge ares )
if ( ares ) begin
dayMask <= '1;
apb.resp <= '{ default: 0 };
end
else begin
if ( apb.WriteReg( REG_DAY_MASK ) )
dayMask <= apb.req.wdata[0+:$bits(dayMask)];

if ( apb.ReadReg( REG_DAY_MASK ) )
apb.resp <= '{ ready: 1, rdata: apb_data_t'(dayMask) }
else
apb.resp <= '{ default: 0 };
end
endmodule
module reg_bd_month (input logic clk, input logic ares, apb_if.slave apb, output MONTH_e [FAMILY_SIZE-1:0] bdMonth);

import apb_pkg::*;

apb_resp_s resp;
const apb_addr_e start_reg = REG_BD_MONTH_MOM;

always_comb begin
    resp = '{ default: 0);
    foreach ( bdMonth[i] ) begin
        if ( apb.ReadReg( start_reg.next(i) ) )
            resp = '{ ready: 1,
                        rdata: apb_data_t'( bdMonth[i] )
            end
    end
end
localparam int width = $bits(MONTH_e);

always_ff @( posedge clk or posedge ares ) begin
    if ( ares ) begin
        bdMonth <= '{ default: bdMonth[$low(bdMonth)].first }
        apb.resp <= '{ default: 0 }
    end
    else begin
        foreach ( bdMonth[i] ) begin
            if ( apb.WriteReg( start_reg.next(i) ) )
                bdMonth[i] <= MONTH_e'(apb.req.wdata[0+:width]);
        end
        apb.resp <= resp;
    end
end
endmodule
import other_pkg::*;

module reg_temps ( input logic clk, input logic ares,
    apb_if.slave apb,
    output temp_t [DEC:JAN][HI:LO] temps );

import apb_pkg::*;

apb_resp_s resp;
const apb_addr_e start_reg = REG_TEMP_JAN;
const int shift = 16;

always_comb begin
    resp = '{ default: 0 };
    foreach ( temps[month,range] ) begin : rd_loop
        apb_data_t rdata;
        rdata = apb_data_t'(temps[month][range]);
        if ( apb.ReadReg( start_reg.next(month) ) ) begin
            resp.ready = 1;
            resp.rdata = resp.rdata | ( rdata << (range*shift) );
        end
    end
end
always_ff @(posedge clk or posedge ares) begin
    if (ares) begin
        temps <= 0;
        apb.resp <= '{ default: 0 };
    end
    else begin
        foreach (temps[month,range]) begin : wr_loop
            localparam int width = $bits(temp_t);
            apb_data_t wdata;
            wdata = apb.req.wdata[shift+:width];
            if (apb.WriteReg(start_reg.next(month)))
                temps[month][range] <= wdata;
        end
        apb.resp <= resp;
    end
end
endmodule
import apb_pkg::*;

module apb_bridge ( apb_if.slave master, apb_if.master slave[SLAVE_NUM] );

logic sel [$size(slave)];
always_comb begin
    if ( ~master.sel )
        sel = '{ default: 0 };
    else begin
        sel[SLAVE_TEMPS] = master.req.addr
            inside { [REG_TEMP_JAN:REG_TEMP_DEC] };
        sel[SLAVE_BD_MONTH] = master.req.addr
            inside { REG_BD_MONTH_MOM, REG_BD_MONTH_DAD,
            REG_BD_MONTH_SON, REG_BD_MONTH_DAUGHTER };
        sel[SLAVE_DAY_MASK] = master.req.addr == REG_DAY_MASK;
    end
end
apb_bridge.sv (continued)

```verilog
apb_resp_s resp [size(slave)];

for (genvar i = 0; i < size(slave); i++) begin : gen_select
    always_comb begin
        slave[i].sel = sel[i];
        slave[i].req = sel[i] ? master.req
                         : '{ addr: apb_addr_e'(0),
                           default: 0 };
        resp[i]      = slave[i].resp;
    end
end

assign master.resp = resp.or;
endmodule
```

- **Genvar without the use of the generate keyword.**
- **Loop variable declaration inside a for loop.**
- **Packed array of a struct.**
- **Plus-plus operator.**
- **Struct assignment with enum and default.**
- **Decomposition of array of interfaces using a genvar.**
- **Unpacked array reduction method.**
module apb_top (input logic clk, input logic ares,
apb_if.slave host);

import apb_pkg::SLAVE_TEMPS;
import apb_pkg::SLAVE_BD_MONTH;
import apb_pkg::SLAVE_DAY_MASK;

apb_if slave[apb_pkg::SLAVE_NUM]() {
    apb_bridge bridge (.master(host), .slave);
    reg_temps temps (.apb(slave[SLAVE_TEMPS]), .temps(), .*);
    reg_bd_month bdmonth (.apb(slave[SLAVE_BD_MONTH]), .bdMonth(), .*);
    reg_day_mask daymask (.apb(slave[SLAVE_DAY_MASK]), .dayMask(), .*);
}

endmodule
Each slide represents an actual tool issue overcome during project development

TOOL WORKAROUNDS
Enums as index

- **Original code:**
  ```
  typedef enum int { BUS_A, BUS_B, BUS_C } bus_targets_e;
  logic [BUS_C:BUS_A] all_valid;
  logic valid;
  assign valid = |all_valid;
  block_a u_block_a ( all_valid[BUS_A] );
  block_b u_block_b ( all_valid[BUS_B] );
  block_c u_block_c ( all_valid[BUS_C] );
  ```

- **Unsuccessful workaround attempt:**
  ```
  always_comb begin
    valid = 0;
    for ( int i = BUS_A; i <= BUS_C; i++ )
      valid = valid | all_valid[i];
  end
  ```

- **Workaround code:**
  ```
  assign valid = all_valid[BUS_A] | all_valid[BUS_B] | all_valid[BUS_C];
  ```

- **Conclusion:**
  - Tool complains that bits of all_valid are not driven
  - Explicitly OR the bits together
  - May also work with localparam instead of enum
Use of enum in an interface select

- Original code:
  ```vhdl
  interface readback_if;
  logic [31:0] data;
  logic valid;
endinterface

typedef enum int { UART, SPI } target_e;
readback_if rdata[SPI:UART]( );

logic valid;
logic [31:0] data;
always_comb begin
  valid = rdata[UART].valid | rdata[SPI].valid;
  data  = rdata[UART].data  | rdata[SPI].data;
end
  ```

- Workaround code:
  ```vhdl
  valid = rdata[int'(UART)].valid | rdata[int'(SPI)].valid;
  data  = rdata[int'(UART)].data  | rdata[int'(SPI)].data;
  ```

- Conclusion:
  - Cast interface select enums to an int
Broadside struct default assignments

- **Original code:**
  ```
  typedef struct packed {
    logic [4:0] hour;
    logic [5:0] minute, second;
  } time_s;

  time_s noon;
  always_comb begin
    noon = '0; // Should assign all fields to zero
    noon.hour = 12;
  end
  ```

- **Workaround code:**
  ```
  always_comb begin
    noon.second = 0;
    noon.minute = 0;
    noon.hour = 12;
  end
  ```

- **Conclusion:**
  - Tool complains that noon.second and noon.minute are unassigned
  - Explicitly assign each field of the struct
Parameter keyword in parameter port lists

- Original code:
```
package test_case_pkg;
  typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
endpackage

import test_case_pkg::*;
module test_case_module #(days_e THIS_DAY = days_e'( 0 ), int WIDTH = 1 )
  // Code here...
endmodule
```

- Workaround code:
```
import test_case_pkg::*;
module test_case_module #(parameter days_e THIS_DAY = days_e'( 0 ), parameter int WIDTH = 1 )
  // Code here...
endmodule
```

- Conclusion:
  - Add parameter keyword even though LRM says it isn’t needed
Loss of type in parameter port lists after an enum

- **Original code:**
  ```verilog
  package test_case_pkg;
    typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
  endpackage

  import test_case_pkg::*;
  module test_case_module #( days_e THIS_DAY = days_e'( 0 ), int WIDTH = 1 )
    // Code here...
  endmodule
  ```

- **Workaround code:**
  ```verilog
  import test_case_pkg::*;
  module test_case_module #( days_e THIS_DAY = days_e'( 0 ), int WIDTH = int'( 1 ) )
    // Code here...
  endmodule
  ```

- **Conclusion:**
  - Add parameter keyword even though LRM says it isn’t needed
Macro with open parenthesis on different line

- Original code:
  ```verilog
counter = 0;
  if `REPlicate (counter < 10) {
    counter++;
  }
  ```

- Workaround code:
  ```verilog
counter = 0;
  if `REPlicate (counter < 10, counter++) {
  }
  ```

- Conclusion:
  - Open parenthesis must be on the same line as the macro
  - Just plain Verilog
Use of enum with a genvar for loop

- **Original code:**
  ```verilog
typedef enum int { HEADS, TAILS } coin_e;
  logic [1:0] coin_side;
  for ( genvar i = HEADS; i <= TAILS; i++ ) begin : gen_coin
    assign coin_side[i] = i;
  end
  ```

- **Workaround code:**
  ```verilog
  for ( genvar i = int'( HEADS ); i <= int'( TAILS ); i++ ) begin : gen_coin
    assign coin_side[i] = i;
  end
  ```

- **Conclusion:**
  - Recast enum to int
Casting using $bits as the vector size

- **Original code:**
  ```
  typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
  
  days_e day_as_enum;
  logic [7:0] day_as_byte;
  
  assign day_as_enum = THU;
  assign day_as_byte = $bits(day_as_byte)'( day_as_enum );
  ```

- **Code to fix one tool’s order of operation issue:**
  ```
  assign day_as_byte = ($bits(day_as_byte))'( day_as_enum );
  ```

- **Workaround code:**
  ```
  typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
  typedef logic [7:0] byte_t;

  days_e day_as_enum;
  byte_t day_as_byte;

  assign day_as_enum = THU;
  assign day_as_byte = byte_t'( day_as_enum );
  ```

- **Alternate code (do not use):**
  ```
  assign day_as_byte = type(day_as_byte)'( day_as_enum );
  ```

- **Conclusion:**
  - Different issues in multiple tools
  - Recommend casting with **typedef** types or constants
Packed array of a type defined logic vector

- **Original code:**
  
  ```
  typedef logic [7:0] byte_t;
  byte_t [3:0] dword;
  ```

- **Workaround code:**
  
  ```
  logic [3:0][7:0] dword;
  // Or...
  typedef logic [3:0][7:0] dword_t;
  dword_t dword;
  ```

- **Conclusion:**
  - Combine into a single declaration or typedef
package import is forgotten

- Original code:

```haskell
class test_case_pkg;
    typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
endclass

import test_case_pkg::*;
module test_case_module;
    days_e day;
    assign day = WED;
endmodule
```

- Workaround code:

```haskell
import test_case_pkg::*;
module test_case_module;
    days_e test_case_pkg::day;
    assign day = test_case_pkg::WED;
endmodule
```

- Conclusion:
  - Rare occurrence – no pattern to when or where
  - In such cases explicitly specify the source package
Enumerated module parameters lose their type when assigned to a struct or interface member

- **Original code:**

```verilog
package test_case_pkg;
    typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
    typedef struct packed {
        days_e day;
        logic [4:0] hour;
    } time_s;
endpackage

import test_case_pkg::*;
module test_case_module #( parameter days_e THIS_DAY = SUN )
    ( input logic [4:0] hour; output time_s present );
    assign present.hour = hour;
    assign present.day = THIS_DAY;
endmodule
```

- **Workaround code:**

```verilog
assign present.day = days_e'( THIS_DAY );
```

- **Conclusion:**

  - Re-cast to remind the tool of the member’s type
Enumerated methods (i.e. .first, .next)

- Original code:
  ```verilog
  always_ff @( posedge clk or posedge ares )
  if ( ares )
    qTraffic_light <= qTraffic_light.first;
  else if ( timer_expired )
    qTraffic_light <= qTraffic_light.next;
  ```

- Workaround code:
  ```verilog
  always_ff @( posedge clk or posedge ares )
  if ( ares )
    qTraffic_light <= RED;
  else if ( timer_expired )
    case ( qTraffic_light )
      RED:      qTraffic_light <= GREEN;
      GREEN:    qTraffic_light <= YELLOW;
      default:  qTraffic_light <= RED;
    endcase
  ```

- Conclusion:
  - Tool’s documents explicitly state that enumerated methods aren’t supported
  - No choice but to be explicit which decreases coding efficiency
Use `.num` method in constant expression

- **Original code:**
  ```
  typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
  days_e day;
  localparam int days_per_week = day.num; // produces a value of 7
  logic [days_per_week-1:0] busy_that_day;
  ```

- **Workaround code:**
  ```
  localparam int days_per_week = 2**$bits( days_e ); // produces a value of 8
  ```

- **Conclusion:**
  - Brute force the vector creation by any number of methods
  - Method chosen is flexible and responds to changes in enumeration
    - But likely results in superfluous bits
Modport in instantiation port connectivity

- **Original code:**
  ```
  interface payload_if;
    logic payload, rts, rtr;
    modport initiator( output payload, rts, input rtr );
    modport target ( input payload, rts, output rtr );
  endinterface

  payload_if my_payload;
  initiator_submodule u_initiator ( .my_payload( my_payload.initiator ) );
  target_submodule u_target ( .my_payload( my_payload.target ) );
  ```

- **Workaround code:**
  ```
  payload_if my_payload;
  initiator_submodule u_initiator ( .my_payload ); // Wildcard is now possible
  target_submodule u_target ( .my_payload );
  ```

- **Conclusion:**
  - Only workaround is to be less specific and eliminate the modport in instance
  - The fringe benefit is that this allows the use of instance wildcarding
Implied flop enable to a flop that is always zero

- **Original code:**
  ```verilog
l的角色 [31:0] irq, set_irq;
logic set_src_a, set_src_b;
assign set_irq = { 15'b0, set_src_b, 15'b0, set_src_a };
always_ff @(posedge sclk_g or posedge ares) begin
    if (ares)
        irq <= '0;
    else
        for (int i = 0; i < $bits(irq); i++)
            if (set_irq[i]) irq[i] <= 1'b1;
end
```

- **Workaround code:**
  ```verilog
always_ff @(posedge sclk_g or posedge ares) begin
    if (ares)
        irq <= '0;
    else
        irq <= irq | set_irq;
end
```

- **Conclusion:**
  - Tool cannot separate an always zero flop from the coding style
  - Just plain Verilog
Enumerated constants as select in array of interfaces

- **Original code:**
  ```
  typedef enum int { CABLE_BOX, BLURAY, NETWORK } video_sources_e;
  interface stream_if;
      logic [31:0] data;
      logic rts, rtr;
  endinterface

  stream_if video_stream[2:0]();

  assign video_stream[CABLE_BOX].rtr = cable_box_selected & rtr;
  ```

- **Workaround code:**
  ```
  localparam int CABLE_BOX=0, BLURAY=1, NETWORK=2;
  interface stream_if;
      logic [31:0] data;
      logic rts, rtr;
  endinterface

  stream_if video_stream[2:0]();

  assign video_stream[CABLE_BOX].rtr = cable_box_selected & rtr;
  ```

- **Conclusion:**
  - Use a `localparam` in place of `enum`
Unpacked array of interfaces

- **Original code:**
  ```
  interface stream_if;
  logic [31:0] data;
  logic rts, rtr;
  endinterface

  stream_if my_stream[3]();
  ```

- **Workaround code:**
  ```
  interface stream_if;
  logic [31:0] data;
  logic rts, rtr;
  endinterface

  stream_if my_stream[2:0]();
  ```

- **Conclusion:**
  - Packed array format is accepted
Use of $clog2 in parameter definition to override a module

- Original code:

```
module inv_addr #( parameter int WIDTH = 8 )
  ( input [WIDTH-1:0] iAddr, output [WIDTH-1:0] oAddr );
  assign iAddr = ~oAddr;
  $display( "Addr width %d", $bits( iAddr ) ); // Should be 8 but Conformal says 1
endmodule

localparam int WIDTH = $clog2( 256 );
inv_addr #( WIDTH ) uinv_addr ( .iAddr( addr_in ), .oAddr( addr_out ) );
```

- Workaround code:

```
localparam int WIDTH = 8;
inv_addr #( WIDTH ) uinv_addr ( .iAddr( addr_in ), .oAddr( addr_out ) );
```

- Conclusion:
  - Remove the $clog2 from the parameter definition
Generate loops for RAM instances

- Original code:
  ```vhdl
  for (genvar i = 0; i < 4; i++) begin : gen_ram
    cache_data u_ram
    (  
      .clk (ifRam.clk),
      .cs_n (ifRam.csn),
      .addr (addr[i]),
      .din  (wdata[i]),
      .we_n (ifRam.wen),
      .dout (rdata[i])
    );
  end
  ```

- Conclusion:
  - Manually unroll the loop (too large to include code here)
Multi-line loops in macros

- Original code:

```
`define RESET_MACRO(reset_code) (ares) begin reset_code end
`define RESET_MACRO_SYNC(reset_code,sres) \
  `RESET_MACRO(reset_code) else if (sres) begin reset_code end

logic [7:0][2:0] qArray1, qArray2;

always_ff @( posedge clk or posedge ares )
  if `RESET_MACRO_SYNC(
    for ( int i = 0; i < 8; i++ ) begin
      qArray1[i] <= i;
      qArray2[i] <= 7-i;
    end
  )
    ,software_reset
)

- Workaround code:

```
always_ff @( posedge clk or posedge ares )
  if `RESET_MACRO_SYNC(
    for ( int i = 0; i < 8; i++ ) qArray1[i] <= i;
    for ( int i = 0; i < 8; i++ ) qArray2[i] <= 7-i;
  )
    ,software_reset
)
```

- Conclusion:
  - Keep code on a single line until the semicolon
  - No begin/end allowed
`define using the `` syntax in macro

- Original code:
  ```haskell
  `ifdef VENDOR_SPECIFIC_COVERAGE_TOOL_DEFINE
  `define COVERAGE(cov) /* pragma coverage cov
  `else
  `define COVERAGE(cov) /* coverage comment for other tools
  `endif
  ```

- Workaround code:
  ```haskell
  `ifdef VENDOR_SPECIFIC_COVERAGE_TOOL_DEFINE
  `define COVERAGE(cov) /* pragma coverage cov
  `else
  `define COVERAGE(cov)
  `endif
  ```

- Conclusion:
  - An empty macro seems to work
Inline comments in nested macros

- **Original code:**

  ```verilog
  `define RESET_MACRO(reset_code) (ares) begin reset_code end
  `define ENUM_FIRST(enumtype) enumtype'( 0 )
  
typedef enum logic [1:0] { RED, GREEN, YELLOW } traffic_light_e;
typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
traffic_light_e qTraffic_light;
days_e qDay;

  always_ff @( posedge clk or posedge ares )
    if `RESET_MACRO(
      qTraffic_light <= `ENUM_FIRST( traffic_light_e ); // SVWORKAROUND qTraffic_light.first;
      qDay <= `ENUM_FIRST( days_e ); // SVWORKAROUND qDay.first;
    )

  ```

- **Workaround code:**

  ```verilog
  always_ff @( posedge clk or posedge ares )
    // SVWORKAROUND qTraffic_light.first;
    // SVWORKAROUND qDay.first;
    if `RESET_MACRO(
      qTraffic_light <= `ENUM_FIRST( traffic_light_e );
      qDay <= `ENUM_FIRST( days_e );
    )

  ```

- **Conclusion:**
  - Move comments out of the outermost macro
  - Loss of context for the comment
Use of default assign to struct with enum field

- **Original code:**
  ```
  typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
  typedef struct packed { 
      days_e day;
      logic [4:0] hour;
  } time_s;
  ```
  ```
  time_s start_time;
  assign start_time = '{ default: 0 };
  ```

- **Workaround code:**
  ```
  assign start_time = '{ day: SUN, default: 0 };
  ```

- **Conclusion:**
  - Be explicit about enum assignments
  - Other tools complain with warnings not errors
### Reuse of genvar for loop variable

#### Original code:

```verilog
typedef enum int { HEADS, TAILS } coin_e;
typedef enum int { ROCK, PAPER, SCISSORS } roshambo_e;
logic [1:0] coin_side;
for ( genvar i = HEADS; i <= TAILS; i++ ) begin : gen_coin
    assign coin_side[i] = i;
end
logic [2:0] roshambo_turn;
for ( genvar i = ROCK; i <= SCISSORS; i++ ) begin : gen_roshambo
    assign roshambo_turn[i] = i;
end
```

#### Workaround code:

```verilog
for ( genvar j = ROCK; j <= SCISSORS; j++ ) begin : gen_roshambo
    assign roshambo_turn[j] = j;
end
```

#### Conclusion:

- Scoping rules allow reuse
- Change to a different genvar name to make it work
Original code:
```verbatim
begin_keywords "1800-2005"
module test_case;
    // Only 1800-2005 compliant keywords allowed
endmodule
end_keywords
```

Workaround code:
```verbatim
module test_case;
    // No keyword protection
endmodule
```

Conclusion:
- Only provides keyword checking, not full syntax checking
- With limited value, no harm in eliminating
Set operator inside used in continuous assignment

- **Original code:**
  ```
  typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
  days_e day;
  logic is_weekend;

  assign is_weekend = day inside { SUN, SAT };
  ```

- **Workaround code:**
  ```
  always_comb is_weekend = day inside { SUN, SAT };
  ```

- **Conclusion:**
  - Must use procedural assignment
Packed array of interfaces that doesn’t start at zero

- **Original code:**
  ```
  interface month_if;
    logic [4:0] day;
  endinterface
  
  month_if calendar_Q4_months[12:10]();
  ```

- **Workaround code:**
  ```
  month_if calendar_Q4_months[2:0]();
  ```

- **Conclusion:**
  - Some tools mistake [12:10] as a bit slice
  - Bit-sliced interface usage is explicitly called out as illegal in the LRM
  - Reverting to [n:0] loses its self-documenting appearance
`define using the `` syntax as non-macro

- **Original code:**
  ```vhdl
  `ifdef VENDOR_SPECIFIC_COVERAGE_TOOL_DEFINE
  `define COVERAGE_ON   /``/ pragma coverage on
  `define COVERAGE_OFF  /``/ pragma coverage off
  `else
  `define COVERAGE_ON   /``/ coverage comment for other tools
  `define COVERAGE_OFF  /``/ coverage comment for other tools
  `endif
  ```

- **Workaround code:**
  ```vhdl
  `ifdef VENDOR_SPECIFIC_COVERAGE_TOOL_DEFINE
  `define COVERAGE(cov) /``/ pragma coverage cov
  `else
  `define COVERAGE(cov) /``/ coverage comment for other tools
  `endif
  ```

- **Conclusion:**
  - No issue with `` syntax when used in a `define macro
Non-overridden parameter in an interface

- **Original code:**
  ```
  interface stream_if;
    parameter NDATA = 32;
    logic [NDATA-1:0] data;
    logic rts, rtr;
  endinterface
  ```

- **Workaround code:**
  ```
  interface stream_if;
    logic [31:0] data;
    logic rts, rtr;
  endinterface
  ```

- **Conclusion:**
  - Parameter will throw an error even if the parameter is not overridden
Enumerated interface members loose their type

- **Original code:**
  ```
typedef enum logic [2:0] { SUNNY, OVERCAST, FOG, RAIN, SNOW } weather_e;
interface weather_report_if;
  logic update;
  weather_e condition;
endinterface

weather_report_if weather_report();
weather_e current_weather;

always_latch begin
  if ( weather_report.update )
    current_weather <= weather_report.condition;
end
  ```

- **Workaround code:**
  ```
always_latch begin
  if ( weather_report.update )
    current_weather <= weather_e'( weather_report.condition );
end
  ```

- **Conclusion:**
  - Re-cast to remind the tool of the member’s type
Enumerated module parameters assigned to an enumerated constant

- **Original code:**
  ```verilog
classical code
  package test_case_pkg;
  
  typedef enum logic [2:0] { SUN, MON, TUE, WED, THU, FRI, SAT } days_e;
  
  import test_case_pkg::*;
  module test_case_module #( parameter days_e THIS_DAY = SUN )
  // Code here...
  endmodule
  
  ```

- **Workaround code:**
  ```verilog
  import test_case_pkg::*;
  module test_case_module #( parameter days_e THIS_DAY = days_e'( 0 ) )
  // Code here...
  endmodule
  ```

- **Conclusion:**
  - Re-cast to remind the tool of the member’s type
Ternary after a for loop

- **Original code:**
  ```verilog
  logic [4:0] hour;
  logic [1:0] data;

  always @( posedge hour_clk or posedge ares )
    if ( ares )
      data <= 1;
      hour <= '0;
    else begin
      for ( int i = 0; i < 2; i++ ) data[i] <= ~data[i];
      hour <= ( hour == 23 ) ? 0 : hour + 1; // Tool sees two coverage blocks
    end
  ```

- **Workaround code:**
  ```verilog
  else begin
    hour <= ( hour == 23 ) ? 0 : hour + 1; // Tool sees one coverage block
    for ( int i = 0; i < 2; i++ ) data[i] <= ~data[i];
  end
  ```

- **Conclusion:**
  - Not a parsing issue!
  - Probably just a Verilog issue
  - Tool creates a superfluous coverage block which is never covered
  - Move the for loop after the code
Thank you!