Standard Gotchas
Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know!

Stuart Sutherland
Sutherland HDL, Inc.
Portland, Oregon
stuart@sutherland-hdl.com

Don Mills
Microchip
Chandler, Arizona
don.mills@microchip.com
Presentation Overview

- What is a “gotcha”?
- Why do standards have gotchas?
- What’s covered in this paper
- Several example gotchas, and how to avoid them!
- Summary
What Is A Gotcha?

- In programming, a “gotcha” is a legal language construct that does not do what the designer expects

**A Classic C programming Gotcha…**

```
if (day = 15)
/* process payroll */
```

**Gotcha!**

- If middle of the month, then pay employees…

**GOTCHA!** This code will assign the value of 15 to day, and then if day is not zero, pay the employees

- In hardware design and verification, most gotchas will simulate, but give undesired results
  - Gotchas can be difficult to find and debug
  - A gotcha can be disastrous if not found before tape-out!

Engineers need to know how to recognize and avoid gotchas in hardware modeling!
Why Do Standards Have Gotchas?

- Standards developers are idiots
- Users of standards are idiots
- Languages can be used the right way, or the wrong way

```c
if (day = 15) /* process payroll */
```

Gotcha!

```c
while (data = fscanf(...)) /* read in data until it is 0 */
```

A dumb way to use “assignment within an expression”

A clever way to use “assignment within an expression”

- Hardware models are not just simulated, they are synthesized, analyzed, emulated, prototyped, formally proved, …
  - Each type of tool needs different information from the language
- Verilog and SystemVerilog allow designers to prove what will — and what will not — work correctly
  - It needs to be legal syntax to model bad hardware
Is This a Verilog Gotcha?

- Is the classic C gotcha also a gotcha in Verilog?
  
  ```verilog
  always @(state)
  if (state = LOAD)
  ...
  ```
  
  Legal or Illegal?
  Illegal! Verilog does not allow assignment statements inside of expressions

- What about in SystemVerilog?
  - SystemVerilog extends Verilog with more C and C++ features

  ```verilog
  always @(state)
  if (state = LOAD)
  ...
  ```
  
  Legal or Illegal?
  If you don’t know the answer, then you really need to read this paper!
  (We will answer this question at the end of our presentation...)
What’s In This Paper...

- Detailed descriptions of 57 gotchas…and how to avoid them!

  - Case sensitivity
  - Implicit net declarations
  - Escaped identifiers in hierarchy paths
  - Verification of dynamic data
  - Variables declared in unnamed blocks
  - Hierarchical references to package items
  - Variables not dumped to VCD files
  - Shared variables in modules
  - Shared variables in interfaces, packages
  - Shared variables in tasks and functions
  - Importing enum types from packages
  - Importing from multiple packages
  - Resetting 2-state models
  - Locked state machines
  - Hidden design problems
  - Out-of-bounds indication lost
  - Signed versus unsigned literal integers
  - Default base of literal integers
  - Size mismatch in literal integers
  - Literal size mismatch in assignments
  - Z extension backward compatibility
  - Filling vectors
  - Passing real types through ports
  - Port connection rules
  - Back-driven input ports
  - Self- & context-determined operations
  - Operation size and sign extension
  - Signed math operations
  - Bit and part select operations
  - Increment and decrement operations
  - Pre-increment versus post-increment
  - Multiple read/writes in one statement
  - Operator evaluation short circuiting
  - Assignments in expressions
  - Procedural block activation
  - Combinational logic sensitivity lists
  - Arrays in sensitivity lists
  - Vectors in sensitivity lists
  - Operations in sensitivity lists
  - Sequential blocks with begin...end
  - Sequential blocks with partial reset
  - Blocking assigns in sequential blocks
  - Evaluation of true/false on 4-state values
  - Not operator versus invert operator
  - Nested if...else blocks
  - Casez/casex masks in case expressions
  - Incomplete or redundant decisions
  - Out-of-bounds in enumerated types
  - Statements that hide design problems
  - Simulation versus synthesis mismatches
  - Multiple levels of same virtual method
  - Event trigger race conditions
  - Using semaphores for synchronization
  - Using mailboxes for synchronization
  - Coverage reporting
  - $unit declarations
  - Compiling $unit
An undeclared signal used in a netlist infers an implicit net
- Implicit nets can save typing hundreds of lines of code in a large, gate-level design

But...
- An undeclared vector connection infers a 1-bit wire, not a vector;
- A typographical error in a netlist becomes a functional bug

Nets \text{n0} “en-zero” and \text{nO} “en-oh” are inferred, but are not connected together
Nets \text{n1} “en-one” and \text{n1} “en-ell” are inferred, but are not connected together
Nets \text{addr} and \text{data} are inferred as 1-bit wires, but should probably be vectors

To avoid this Gotcha...
- Verilog’s \text{default_nettype none} turns off implicit net types
- SystemVerilog \text{name} and \text{*} port connections will not infer nets
Gotcha: Default Base of Literal Numbers is Decimal

- Optionally, literal numbers can be defined with a base
  
  `’hC` // unsized hex value  |  `2’b01` // sized binary value

- **But...the default base is decimal**

```
reg [1:0] state;

always @(state)
  case (state)
    00: // do State 0 stuff
    01: // do State 1 stuff
    10: // do State 2 stuff
    11: // do State 3 stuff
  endcase
```

Why don't states 2 and 3 ever execute?

- **To avoid this Gotcha...**
  - Use `unique case` to detect the error
  - Use based numbers to fix the problem (e.g `2’b10`)

Hint: There are 10 types of people in the world... those that know binary, and those that don’t!
Gotcha: Literal Numbers Are Zero-extended to Their Size

- Literal number syntax is: `<size>'<signed><base><value>
  - `<size>` (optional) is the number of bits (default size is 32 bits)
  - `<signed>` (optional) is the letter `s` or `S` (default is unsigned)
  - `<base>` is `b`, `o`, `d`, `h` for binary, octal, decimal or hex (not case sensitive)
    - `’hC` // unsized hex value
    - `2’b01` // sized binary value

- If the size does not match the number of bits in the value:
  - If the left-most bit of value is 0 or 1, the value is left-extended with 0
  - If the left-most bit of value is `Z`, the value is left-extended with `Z`
  - If the left-most bit of value is `X`, the value is left-extended with `X`
  - **But... a signed value is not sign-extended!**
    - `8’hA` unsigned value extends to `00001010`
    - `8’shA` signed value extends to `00001010`

- To avoid this Gotcha...
  - Engineers need to learn Verilog’s sign extension rules! 😊
Gotcha: Importing Enumerated Types from Packages

- Packages declarations can be “imported” into modules
  - But...importing an enumerated type does not import the enumerated labels

```verilog
class chip_types;
  typedef enum {RESET, WAITE, LOAD, READY} states_t;
endclass

module chip (...);
  import chip_types::states_t;
  states_t state, next_state;
  always @(posedge clk, negedge rstN)
    if (!resetN) state <= RESET;
  ...
```

- To avoid this Gotcha...
  - Either use a wildcard import to import the full package, or explicitly import each enumerated label

```verilog
import chip_types::*;
```
Enumerated types are useful for modeling FSM state names
- **But...** enumerated variables start simulation with uninitialized values (the starting value of the base data type of the enum)

```verilog
module chip (...);
  typedef enum {WAITE, LOAD, READY} states_t;
  states_t  State, nState;
  always_ff @(posedge clk, negedge rstN)
    if (!rstN) State <= WAITE;
    else       State <= nState;
  always @(State)
    case (State)
      WAITE: nState = LOAD;
      ...
endcase
```

- **State** and **nState** both begin with **WAITE**
  - The default base data type is int
  - The uninitialized value of an int is 0
  - The default value of **WAITE** is 0

- To avoid this Gotcha...
  - Use **always_comb** instead of **always @(State)** or **always @***
  - and/or declare the enumerated type with a 4-state base type
Enumerated types define a legal set of values for a variable
- It is illegal to assign an enumerated variable a value not in its list
- But... power-up and/or casting can cause out-of-bounds values

```
module chip (...);
typedef enum logic [2:0] {WAITE=3'b001, LOAD=3'b010, READY=3'b100} states_t;
states_t State, nState;
always_comb begin
    if (enable) nState = states_t'(State + 1); // move to next label in list?
...
```

- If `State` is `WAITE`, adding 1 results in `nState` having 3'b010 (the value of `LOAD`)
- If `State` is `LOAD`, adding 1 results in `nState` having 3'b011 (not in the enumerated list!)
- If `State` is `WAITE`, adding 1 results in `nState` having 3'b101 (not in the enumerated list!)
- At start of simulation, `State` and `nState` have 3'bxxx (not in the enumerated list!)

**To avoid this Gotcha...**
- Out-of-bounds at power-up is useful—it indicates reset problems
- Out-of-bounds assignments can be prevented using enumerated methods
  ```
  nState = State.next; // move to next label in list
  ```
Gotcha: Hierarchical References to Package Items

- Verification can peek inside a design scope using hierarchy paths
  - Hierarchy paths reference objects *where there are declared*
  - **But** *package items are not declared within the design scope*

  ```verilog
  package chip_types;
  typedef enum {RESET, WAITE, LOAD, READY} states_t;
  endpackage
  
  module chip (...);
  import chip_types::*;
  always @(posedge clk, negedge rstN)
    if (!resetN) state <= RESET;
  ...
  
  module test (...);
  chip dut (...);
  ...
  master_reset = 1;
  ##1 assert (dut.state == dut.RESET);
  ```

  **Gotcha!**
  - Module chip can use **RESET**, but **RESET** is not defined in chip

  To avoid this Gotcha...
  - The package must also be imported into test bench, OR...
  - Package items can be referenced with a “scope resolution operator”

  ```verilog
  ##1 assert (dut.state == chip_types::RESET);
  ```
There are two types of operators in Verilog and SystemVerilog:

- **Self-determined operators** do not modify their operand(s)

  ```verilog
lógic [1:0] a;
lógic [3:0] b;
ena = a && b;
```

  Logical-AND is self-determined (the operation on $a$ does not depend on the size or signedness of $ena$ or $b$; the operation on $b$ does not depend on $ena$ or $a$)

- **Context-determined operators** modify all operands to have the same size, using zero-extension or sign-extension

  **But,… the context for size and sign are different!**

  ```verilog
  logic signed [3:0] a;
lógic signed [4:0] c;
a = -1;
c = a + 1'b1;
  ```

  ADD is context-determined (the operation depends on the size of $a$, $c$ and $1'b1$) and signedness of $a$ and $1'b1$)

  **Gotcha!** What is $-1 + 1$?

  - Answer A: $-1 + 1'b1 = -16$ !
  - Answer B: $-1 + 1'sb1 = -2$ !
  - Answer C: $-1 + 2'sb01 = 0$ !

**To avoid this Gotcha…**

You have got to know if an operator is self-determined or context-determined — See the table in this paper!
Explanation of Operation Size and Sign Extension Example

- **Context-determined operators** modify all operands to have the same size, using zero-extension or sign-extension

- **But,… the context for size and sign are different!**
  - The context for size is both the right and left-hand side of an assignment
  - The context for signedness is just the right-hand side of an assignment

```
logic signed [3:0] a = -1;
logic signed [4:0] c;
```

Why is “c = a + 1'b1” = -16?

- Size context is 5-bits (largest expression size is c)
- Sign context is unsigned (1'b1 is an unsigned expression)
  - a is 1111 (-1) which extends to 01111 (15)
  - 1'b1 is 1 (1) which extends to 00001 (1)
  - c = 10000 (-16)

Why is “c = a + 1'sb1” = -2?

- Size context is 5-bits (largest expression size is c)
- Sign context is signed (a and 1'sb1 are signed expressions)
  - a is 1111 (-1) which extends to 11111 (-1)
  - 1'sb1 is 1 (1) which extends to 11111 (-1)
  - c = 11110 (-2)
Gotcha: Signed Arithmetic

- Math operations are context-determined
  - Signed arithmetic is done if all operands are signed
  - **But,... If any operand in the context is unsigned, then unsigned arithmetic is done**

```
logic signed [3:0] a, b;
logic ci;
logic signed [4:0] sum;
sum = a + b + ci;
```

**Unsigned adder**, even though a, b and sum are signed
- Size context is largest vector on left and right sides
- a, b and ci are extended to 5 bits before being added
- Sign context is only the operands on right side
- ci is unsigned, so the context is unsigned
- a and b are converted to unsigned and zero extended

```
logic signed [3:0] a, b;
logic signed ci;
logic signed [4:0] sum;
sum = a + b + ci;
```

**Signed adder that subtracts carry in**
- a, b and ci are signed, so the context is signed
- if ci is 1, sign extending to 5 bits gives 11111 (binary)
- As a signed value, 11111 (binary) is -1!

**To avoid this Gotcha**...
- Engineers must know Verilog’s context-operation rules!

```
sum = a + b + signed’({1'b0, ci});
```
Gotcha: Operations in Sensitivity Lists

- Verilog sensitivity lists can contain operations
  - *But,... the list is only sensitive to changes in the operation result*

```verilog
always @(a or b)
  sum = a + b;
```

“or” is a separator, not an operation; Sensitive to changes on `a` or `b`

If `a` is 1, and `b` changes from 0 to 1, the block will trigger

Gotcha!

```verilog
always @(a || b)
  sum = a + b;
```

“||” is the logical-OR operator; Sensitive to changes on the result of the test “is `a` true, OR is `b` true”

If `a` is 1, and `b` changes from 0 to 1, the block will **not** trigger

---

- **To avoid this Gotcha...**
  - For combinational logic, use `always @*` or `always_comb` to infer a correct combinational logic
  - For sequential logic, using a `comma` instead of “*or*” to separate items in the sensitivity list
Gotcha: Assignments in Expressions

- Is the classic C gotcha also a gotcha in SystemVerilog?

```verilog
always @ (state)
  if (state == LOAD)
  ...
```

- SystemVerilog allows assignments in expressions…
  - But,… the syntax is different than C — the assign statement must be enclosed in parentheses

```verilog
always @ (state)
  if ((state == LOAD))
  ...
```

- To avoid this Gotcha…
  - “It is what it is” — Engineers need to learn the unique SystemVerilog syntax
Summary

- Programming languages have gotchas
  - A legal construct used in a way that gives unexpected results
  - Gotchas occur because useful language features can be abused
  - Some gotchas are because Verilog and SystemVerilog allow engineers to prove what won’t work in hardware

- A gotcha in a hardware model can be disastrous
  - Difficult to find and debug
  - If not found before tape-out, can be very costly

- This paper describes 57 Verilog and SystemVerilog gotchas
  - Detailed explanations of each gotcha
  - Guidelines on how to avoid each gotcha
  - Lots of code examples
Questions & Answers…

Do you have a favorite gotcha that is not in the paper?

Please send it to Stu or Don!

And then stay tuned for a “Standard Gotchas, Part 2” paper at some future SNUG…

stuart@sutherland.com  don.mills@microchip.com