What, If Anything, In SystemVerilog Will Help Me With FPGA-based Design

Stuart Sutherland, Consultant and Trainer, Sutherland HDL, Inc.

UBM Electronics

About the Presenter...

- **Stuart Sutherland, SystemVerilog wizard**
  - Independent SystemVerilog consultant and trainer
    - Hardware design engineer
    - Have worked with Verilog since 1988
    - Specialize in Verilog and SystemVerilog training
    - BS in Computer Science, MS in eLearning Education
  - Member of the IEEE Verilog and SystemVerilog standards groups
    - Involved with the IEEE Verilog standard since its beginning
    - Involved with the definition of SystemVerilog since its inception
    - Technical editor of every generation of the Verilog and SystemVerilog Language Reference Manuals

UBM Electronics

What This Presentation Will Discuss

- **Verilog versus SystemVerilog**
  - There is more to the story than you might be aware of
- **Synthesizable SystemVerilog Constructs**
  - There's a lot!
  - Why these constructs are beneficial
- **What do FPGA synthesis tools support?**
  - What commercial synthesis compilers
  - FPGA vendor synthesis compilers
- **Recommendations**
  - When (if ever) will it be time to adopt SystemVerilog?

A History Lesson...

- **Verilog (IEEE standard 1364)**
  - Began in 1983 as a proprietary language
  - Opened to the public in 1992
  - Became an IEEE standard in 1995 (updated in 2001 and 2005)
  - **Between 1983 and 2005 design sizes increased tremendously**
- **SystemVerilog (IEEE standard 1800)**
  - Intended to be the 2005 update to Verilog
  - Contains hundreds of enhancements and extensions to Verilog
  - Published in 2005 as a separate document
  - **Officially superseded Verilog in 2009**

UBM Electronics

SystemVerilog

verification

- classes
- inheritance
- constrained randomization
- functional coverage
- assertions
- test program blocks
- clocking domains
- process control
- mailboxes
- semaphores
- dynamic arrays
- associative arrays
- reference arguments
- direct C function calls

design

- interfaces
- nested hierarchy
- unrestricted ports
- automatic port connect
- enhanced literals
- time values and units
- specialized procedures
- packages
- 2-state modeling
- packed arrays
- array assignments
- queues
- unique/priority case/if
- compilation unit space
- int
- shortint
- longint
- byte
- shortreal
- void
- alias
- globals
- enum
- typedef
- structures
- unions
- casting
- const
- break
- continue
- return
- do–while
- ++ -- += -= *= /=
- >>= <<= >>>= <<<=
- &= |= ^= %=

Verilog-2001/2005

- ANSI C style ports
- generate
- localparam
- constant functions
- standard file I/O
- $value$plusargs
- `ifndef `elsif `line
- @*
- (* attributes *)
- configurations
- memory part selects
- variable part select
- multi dimensional arrays
- signed types
- automatic
- ** (power operator)

Verilog-1995

- modules
- parameters
- function/tasks
- always @
- assign
- $finish $fopen $fclose
- $display $write
- $monitor
- `define `ifdef `else
- `include `timescale
- initial
- disable
- events
- wait # @
- fork–join
- wire reg
- integer real
- time
- packed arrays
- 2D memory
- begin–end
- while
- for forever
- if–else
- repeat
- + = * /
- %
- >> <<

Let's Look At The Details!

- **All we have time for is a quick review of synthesizable SystemVerilog**
  - We will emphasize the benefits of each construct
  - Read the paper for more details

Only synthesizable enhancements are listed — if we don't talk about it in the paper, it is probably not synthesizable by most FPGA tools

SystemVerilog Variables

- **SystemVerilog adds synthesizable variable types**
  - **`logic`** — single bit 4-state variable (replaces Verilog **`reg`** type)
  - **`bit`** — single bit 2-state variable
  - **`byte`** — 8-bit 2-state variable
  - **`shortint`** — 16-bit 2-state variable
  - **`int`** — 32-bit 2-state variable
  - **`longint`** — 64-bit 2-state variable

- **So what?**
  - The **`logic`** type prevents confusion on hardware registers
  - 2 state types are used for constrained random verification

Enumerated Types

- **SystemVerilog adds enumerated types to Verilog**
  - **enum** defines variables or nets with a legal set of values
    - Can be a simple enumerated type

```
enum {WAITE, LOAD, READY} state;
```

- **The first label defaults to a value of 0**
- **Subsequent labels increment from the previous label value**

    - Can specify an explicit size and explicit values for each label

```
enum logic [2:0] {WAITE=3'b001, LOAD=3'b010, READY=3'b100} state;
```

  - Enumerated types have strict rules
    - Can only be assigned a label or another identical enumerated type
- **So what?**

  - Enumerated types can prevent inadvertent (and hard to debug) coding errors (example on next slide)

```
localparam [2:0] WAIT=3'b001, LOAD=3'b010, DONE=3'b100;
localparam [2:0] RED=3'b001, GREEN=3'b010, BLUE=3'b100;
logic [2:0] State1, nState1;
logic [1:0] State2, nState2;
always @(posedge clk or negedge rstN)
  if (!rstN) State1 <= 0;
  else       State1 <= nState2;
always @(State1)
  case (State1) // next state logic
    WAIT : nState1 = State1 + 1;
    LOAD : nState1 = State1 + 1;
    DONE : nState1 = State1 + 1;
```

legal, but a bug – sizes don't match

legal, but a bug – not a valid state value

legal, but a bug – wrong nState variable

legal, and functionally OK

legal, but a bug – not a valid state value

legal, but a bug – not a valid state value

```
enum logic [2:0] {WAIT=3'b001, LOAD=3'b010, DONE=3'b100} State1, nState1;
enum logic [1:0] {RED=3'b001, GREEN=3'b010, BLUE=3'b100} State2, nState2;
always @(posedge clk or negedge rstN)
  if (!rstN) State1 <= 0;
  else       State1 <= nState2;
always @(State1)
  case (State1) // next state logic
    WAIT : nState1 = State1 + 1;
    LOAD : nState1 = State1 + 1;
    DONE : nState1 = State1 + 1;
```

illegal, sizes must match

illegal, must assign a label name

illegal, not from same definition

illegal, must assign a label name

illegal, must assign a label name

illegal, must assign a label name

Structures

- **SystemVerilog adds structures to Verilog**
  - Structures bundle multiple variables together under one name
    - Structure variables can be assigned individually
    - The entire structure can be assigned a list of values
    - Structures can copied and can be passed through module ports

```
struct {
  logic [47:0] opcode;
  logic [63:0] data;
} operation_s;
```

```
operation_s.data = 48'hF;
```

```
operation_s = '{55, 1024};
```

- **So what?**

  - Structures can significantly reduce the amount of code required to model complex functionality
  - Structures can make code easier to read and maintain

User-defined Types

- **SystemVerilog adds user-defined types to Verilog**
  - `typedef` defines a new type
    - Can be based on built-in types or other user-defined types
    - Variables and nets can be declared as a user-defined type

```
typedef enum {FALSE, TRUE} boolean_t;
boolean_t ready;  // variable "ready" can be FALSE or TRUE
```

```
typedef enum {WAIT, LOAD, READY} states_t;
states_t state, next_state;
```

- **So what?**

  - User-defined types reduce the number of times complex types (such as enum and struct) must be declared
  - User-defined types ensure declarations are consistent

SystemVerilog Nets

- **SystemVerilog adds a single-driver net**
  - **`uwire`** — a wire that is restricted to at most one source
- **SystemVerilog adds complex net types**

```
wire integer w2;
```

32-bit 4-state net

```
typedef enum bit {FALSE, TRUE} bool_t;
wire bool_t w3;
```

net with bool_t values

- **So what?**
  - Enforced single-driver nets prevent hard-to-find coding errors
    - (Verilog net types (e.g., **`wire`**) permit any number of sources)
  - Complex nets transfer any variable type to other modules

Packages

- **SystemVerilog adds a package construct to Verilog**
  - Allows the same definition to be used by multiple design modules

```
package definitions;
  typedef enum {s1,s2,s3} states_t;
  task automatic fetch ...
endpackage
```

  - Package definitions can be imported 3 ways
    - Explicit references of package items

```
definitions::states_t state;
```

    - Explicit import of package items

```
import definitions::fetch;
```

    - Implicit “wildcard” import of package

```
import definitions::*;
```

- **So what?**

  - Packages can reduce duplicate code, make code more maintainable, and make code easier to reuse

Data Arrays

- **Packed arrays (old Verilog vectors)**
  - Vectors can be divided into sub fields
- **Unpacked arrays (Verilog arrays)**
  - Arrays of structures, user-defined types, etc.
  - C-like array declarations
  - Copy arrays
  - Assign list of values
  - Assign to slices of an array
  - Pass arrays through module ports
- **So what?**
  - Manipulating entire arrays substantially reduces lines of code (see example on next page)

```
logic [3:0] [7:0] a;
```

a[3]	a[2]	a[1]	a[0]
[7:0]	[7:0]	[7:0]	[7:0]

```
int a1 [2][4]; //C-like declaration
int a2 [0:1][0:3] = '{default:'1};
 // initialize array at declaration
a1 = a2;  // copy array
a2 = '{'{7,3,0,5},'{2,0,1,6}};
          // assign list of values
```

Array Manipulation Reduces Code

```
package design_types;
  typedef struct {
    logic [ 3:0] GFC;
    logic [ 7:0] VPI;
    logic [15:0] VCI;
    logic        CLP;
    logic [ 2:0] T;
    logic [ 7:0] HEC;
    logic [ 7:0] Payload [48];
  } uni_t; // UNI cell definition
endpackage
```

4 lines of code in SystemVerilog replaces 216 lines in old Verilog! (and ensures consistency in all 4 places)

This structure bundles 54 variables together (including the array of 48 Payload variables)

54 ports in old Verilog

another 54 ports

```
module transmit_reg (output design_types::uni_t  data_reg,
                     input  design_types::uni_t  data_packet,
                     input  logic                clock, resetN);
  always @(posedge clock or negedge resetN)
    if (!resetN) data_reg <= '{default:0};
    else         data_reg <= data_packet;
endmodule
```

54 separate assignment statements in old Verilog

another 54 assignments

UBM Electronics

Type, Size and Sign Casting

- **SystemVerilog adds casting operations to Verilog**
  - <type>'(<expression>) — cast expression to different data type
  - <size>'(<expression>) — casts expression to a vector size
  - <sign>'(<expression>) — casts expression to signed or unsigned

SystemVerilog style

```
real r;
logic [ 2:0] b;
logic [31:0] a;

always_comb
  y = a + logic [31:0]'(r ** b);
```

```
real r;
reg [ 2:0] b;
reg [31:0] a;
reg [31:0] temp;

always @(a or b or r) begin
  temp = r ** b;
  y = a + temp;
end
```

old Verilog style to make synthesis infer an integer instead of floating-point adder

- **So what?**
  - Fewer lines of code
  - Self-documenting code
  - More maintainable and reusable code

Hardware Specific Procedural Blocks

- **SystemVerilog adds special hardware-oriented procedures: `always_ff`, `always_comb`, and `always_latch`**
  - Enforce several semantic rules required by synthesis
  - Simulation, synthesis and formal tools to use same rules

old Verilog

```
always @(mode)
  if (!mode)
    y = a + b;
  else
    y = a - b;
```

synthesis must guess what type of logic was intended (might infer wrong type)

SystemVerilog

```
always_comb
  if (!mode)
    y = a + b;
  else
    y = a - b;
```

sensitivity list is inferred

contents checked for adherence to synthesis rules for combinational logic

- **So what?**
  - Self-documenting code
  - Non-synthesizable code won't simulate

Operators and Programming Statements

- **SystemVerilog adds many new synthesizable constructs:**
  - **`++`** and **`--`** increment and decrement
  - **`+=`**, **`-=`**, **`*=`**, **`/=`**, **`%=`**, **`&=`**, **`^=`**, **`|=`**, **`<<=`**, **`>>=`**, **`<<<=`**, **`>>>=`** assignment
  - **`==?`** and **`!=?`** wild equality/inequality operators
  - Variables on left-hand side of continuous assignments
  - **`foreach`** and **`do`**…**`while`** loops
  - **`break`**, **`continue`** and **`return`** jump statements
  - **`unique`** and **`priority`** decision statements (see next slide)
- **So what?**

  - Model more functionality in fewer lines of code
  - Self-documenting code
  - Built-in error checking (see example on next slide)

Unique and Priority Decisions

- **`unique` and `priority` decisions…**
  - Turn on synthesis **`parallel_case`** and/or **`full_case`** pragmas
  - Simulation and synthesis warn if detect overlap in the decisions
  - Simulation has run-time warning if there is no matching condition

```
unique casez (a)
  2'b?0: ...
  2'b1?: ...
  default: ...
endcase
```

- **Will get warnings if `a` decodes to multiple branches (not parallel_case)**
- **Will get warnings if `a` doesn't decode to any branch (not full_case)**

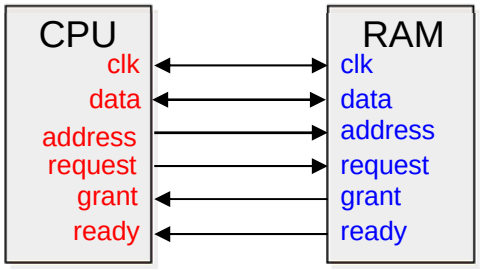
```
unique if (a == 0) ...
  else if (a == 2) ...
  else if (a == 2) ...
  else ...
```

- **So what?**

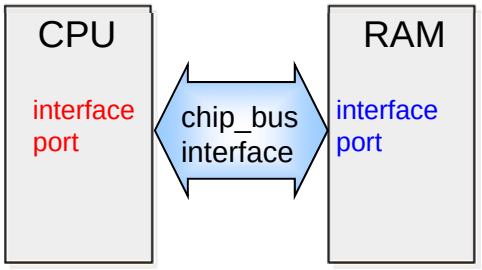
  - Automatic run-time checking that the decision statement will synthesize as intended
  - Prevents very subtle, difficult to debug, design errors

Interface Ports

Verilog discrete ports



SystemVerilog interface ports




```
interface chip_bus;
  logic [31:0] data, address;
  logic        clk, request,
               grant, ready;
endinterface

module CPU (interface chip_bus);
  ...
endmodule
```

- **SystemVerilog interfaces are a compound, multi-signal port**
  - Bundles any number of signals (nets and variables) together
  - Bundles "methods" with the signals (e.g. a handshake sequence)
  - Bundles assertion checks with the signals
- **So what?**

  - Simplifies complex bus definitions and interconnections
  - Ensures consistency throughout the design

Synthesis Support

- **The SystemVerilog constructs presented in this paper are all synthesizable** (and there's more that we didn't discuss…)
  - Commercial FPGA synthesis tools
    - *Synopsys Synplify-Pro* has excellent support
    - *Synopsys DC* has excellent support
    - *Mentor Precision* has excellent support
  - Proprietary FPGA synthesis tools
    - *Xilinx XST* has limited support (planned for version 14.1 in 2012)
    - *Altera Quartus* has good support (but with several exceptions)
    - *Lattice* has excellent support (via commercial tools)
    - *Microsemi* (Actel) has excellent support (via commercial tools)

Conclusions

- ✓**SystemVerilog is Verilog (on steroids)**
  - SystemVerilog-2009 supersedes Verilog-2005
  - The IEEE terminated the Verilog standard in 2009
- ✓**A substantial portion of SystemVerilog is synthesizable**
  - SystemVerilog is a Hardware Design Language
- ✓**Using SystemVerilog you can …**
  - Model more functionality with much less code
  - Reduce potential functionality errors in RTL code
  - Gain built-in compile and run-time synthesis rule checking
- ✓**You can use SystemVerilog today for FPGA design!**
  - FPGA synthesis has excellent support for SystemVerilog

Questions?

A copy of this presentation is available at www.sutherland-hdl.com/papers

About Sutherland HDL

- **Sutherland HDL, Inc. provides SystemVerilog training**
  - ✓SystemVerilog for Design and Synthesis
  - ✓SystemVerilog for Verification
  - ✓SystemVerilog Assertions
  - ✓SystemVerilog UVM
  - All training workshops are available on-site and as online *eTutored*™ training
  - Sutherland HDL instructors are expert trainers and experienced design and verification engineers

Sutherland HDL helps engineers become true SystemVerilog wizards!

visit www.sutherland-hdl.com for workshop descriptions