The Verilog Language

Originally a modeling language for a very efficient event-driven digital logic simulator.
Later pushed into use as a specification language for logic synthesis.
Now, one of the two most commonly-used languages in digital hardware design (VHDL is the other).
Virtually every chip (FPGA, ASIC, etc.) is designed in part using one of these two languages.
Combines structural and behavioral modeling styles.

How Are Simulators Used?

Testbench generates stimulus and checks response.
Coupled to model of the system.
Pair is run simultaneously.

Structural Modeling

When Verilog was first developed (1984) most logic simulators operated on netlists.
Netlist: list of gates and how they’re connected.
A natural representation of a digital logic circuit.
Not the most convenient way to express test benches.
Behavioral Modeling

A much easier way to write testbenches
Also good for more abstract models of circuits
• Easier to write
• Simulates faster
More flexible
Provides sequencing
Verilog succeeded in part because it allowed both the model and the testbench to be described together

Two Main Components of Verilog: Behavioral

Concurrent, event-triggered processes (behavioral)
Initial and Always blocks
Imperative code that can perform standard data manipulation tasks (assignment, if-then, case)
Processes run until they delay for a period of time or wait for a triggering event

Two Main Components of Verilog: Structural

Structure (Plumbing)
Verilog program build from modules with I/O interfaces
Modules may contain instances of other modules
Modules contain local signals, etc.
Module configuration is static and all run concurrently

Two Main Data Types: Nets

Nets represent connections between things
Do not hold their value
Take their value from a driver such as a gate or other module
Cannot be assigned in an initial or always block

Two Main Data Types: Regs

Regs represent data storage
Behave exactly like memory in a computer
Hold their value until explicitly assigned in an initial or always block
Never connected to something
Can be used to model latches, flip-flops, etc., but do not correspond exactly
Actually shared variables with all their attendant problems

Discrete-event Simulation

Basic idea: only do work when something changes
Centered around an event queue that contains events labeled with the simulated time at which they are to be executed
Basic simulation paradigm
• Execute every event for the current simulated time
• Doing this changes system state and may schedule events in the future
• When there are no events left at the current time instance, advance simulated time soonest event in the queue

Four-valued Data

Verilog’s nets and registers hold four-valued data
0, 1: Obvious
Z: Output of an undriven tri-state driver. Models case where nothing is setting a wire’s value
X: Models when the simulator can’t decide the value
• Initial state of registers
• When a wire is being driven to 0 and 1 simultaneously
• Output of a gate with Z inputs

Four-valued Logic

Logical operators work on three-valued logic

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Outputs 0 if either input is 0
Outputs X if both inputs are gibberish
Nets and Registers

Wires and registers can be bits, vectors, and arrays

```verilog
wire a; // Simple wire
tri [15:0] dbus; // 16-bit tristate bus
tri #(5,4,8) b; // Wire with delay
reg [-1:4] vec; // Six-bit register
trireg (small) q; // Wire stores a small charge
integer imem[0:1023]; // Array of 1024 integers
reg [31:0] dcache[0:63]; // A 32-bit memory
```

Modules and Instances

Basic structure of a Verilog module:

```verilog
module mymod(out1, out2, in1, in2);
output out1;
output [3:0] out2;
input in1;
input [2:0] in2;
endmodule
```

Instantiating a Module

Instances of

```verilog
module mymod(y, a, b);
look like
mymod m1(y1, a1, b1); // Connect-by-position
mymod (y2, a1, b1),
(y3, a2, b2); // Instance names omitted
```

// Connect-by-name

```verilog
mymod mm2 (.a(a2), .b(b2), .y(c2));
```

Gate-level Primitives

Verilog provides the following:

```verilog
and nand logical AND/NAND
or nor logical OR/NOR
xor xnor logical XOR/XNOR
buf not buffer/inverter
bufif0 notif0 Tristate with low enable
bufif1 notif1 Tristate with high enable
```

Delays on Primitive Instances

Instances of primitives may include delays

```verilog
buf b1(a, b); // Zero delay
buf #3 b2(c, d); // Delay of 3
buf #(4,5) b3(e, f); // Rise=4, fall=5
buf #(3:4:5) b4(g, h); // Min-typ-max
```

Switch-level Primitives

Verilog also provides mechanisms for modeling CMOS transistors that behave like switches

A more detailed modeling scheme that can catch some additional electrical problems when transistors are used in this way

Now, little-used because circuits generally aren’t built this way

More seriously, model is not detailed enough to catch many of the problems

These circuits are usually simulated using SPICE-like simulators based on nonlinear differential equation solvers

User-Defined Primitives

Way to define gates and sequential elements using a truth table

Often simulate faster than using expressions, collections of primitive gates, etc.

Gives more control over behavior with X inputs

Most often used for specifying custom gate libraries

A Carry Primitive

```verilog
primitive carry(out, a, b, c);
output out;
input a, b, c;
table
  00? : 0;
  0?0 : 0;
  ?00 : 0;
  11?: 1;
  1?1 : 1;
  ?11 : 1;
endtable
endprimitive
```
A Sequential Primitive

Primitive dff( q, clk, data);
output q; reg q;
input clk, data;
table
// clk data q new-q
(01) 0 : ? : 0; // Latch a 0
(01) 1 : ? : 1; // Latch a 1
(0x) 1 : 1 : 1; // Hold when d and q both 1
(0x) 0 : 0 : 0; // Hold when d and q both 0
(?0) ? : ? : -; // Hold when clk falls
? (??) : ? : -; // Hold when clk stable
endtable
endprimitive

Continuous Assignment

Another way to describe combinational function
Convenient for logical or datapath specifications

wire [8:0] sum; Define bus widths
wire [7:0] a, b;
wire carryin;
assign sum = a + b + carryin;Continuous assignment: permanently sets the value of sum to be a+b+carryin. Recomputed when a, b, or carryin changes

Behavioral Modeling

Initial and Always Blocks

initial begin
// imperative statements
end
always begin
// imperative statements
end

Runs when simulation starts Runs when simulation starts
Terminates when control reaches the end Restart when control reaches the end
Good for providing stimulus Good for modeling or specifying hardware

Initial and Always

Run until they encounter a delay

initial begin
#10 a = 1; b = 0;
#10 a = 0; b = 1;
end

or a wait for an event
always @(posedge clk) q = d;
always begin
wait(i);
a = 0;
wait(˜i);
a = 1;
end

Procedural Assignment

Inside an initial or always block:

sum = a + b + cin;

Just like in C: RHS evaluated and assigned to LHS before next statement executes
RHS may contain wires and/or regs
LHS must be a reg
(only primitives or continuous assignment may set wire values)

Imperative Statements

if (select == 1) y = a;
else y = b;
case (op)
 2'b00: y = a + b;
 2'b01: y = a - b;
 2'b10: y = a ^ b;
default: y = 'hxxxx;
endcase

For Loops

Example generates an increasing sequence of values on an output

reg [3:0] i, output;
for ( i = 0 ; i <= 15 ; i = i + 1 ) begin
  output = i;
  #10;
end

While Loops

A increasing sequence of values on an output

reg [3:0] i, output;
i = 0;
while (i <= 15) begin
  output = i;
  #10 i = i + 1;
end
Modeling A Flip-Flop With Always

Very basic: an edge-sensitive flip-flop

```verilog
reg q;

always @(posedge clk)
  q = d;
q = d; assignment runs when clock rises: exactly the behavior you expect
```

Blocking vs. Nonblocking

Verilog has two types of procedural assignment

Fundamental problem:

- In a synchronous system, all flip-flops sample simultaneously
- In Verilog, `always @(posedge clk)` blocks run in some undefined sequence

A Flawed Shift Register

This does not work as you would expect:

```verilog
reg d1, d2, d3, d4;
always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;
These run in some order, but you don't know which
```

Nonblocking Assignments

This version does work:

```verilog
reg d1, d2, d3, d4;

always @(posedge clk) d2 <= d1;
always @(posedge clk) d3 <= d2;
always @(posedge clk) d4 <= d3;
```

Nonblocking Can Behave Oddly

A sequence of nonblocking assignments don’t communicate

<table>
<thead>
<tr>
<th>Blocking assignment:</th>
<th>Nonblocking assignment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b = c = 1</td>
<td>a = 1</td>
</tr>
<tr>
<td>b = old value of a</td>
<td>b = 1</td>
</tr>
<tr>
<td>c = old value of b</td>
<td>c = 1</td>
</tr>
</tbody>
</table>

Nonblocking Looks Like Latches

RHS of nonblocking taken from latches
RHS of blocking taken from wires

![Diagram]

Modeling FSMs Behaviorally

There are many ways to do it:

- Define the next-state logic combinationally and define the state-holding latches explicitly
- Define the behavior in a single `always @(posedge clk)` block
- Variations on these themes

```verilog
module FSM(o, a, b, reset);
  output o;
  reg o;
  input a, b, reset;
  reg [1:0] state, nextState;
  always @(a or b or state)
    case (#state)
      2'b00: begin
        o = a & b;
        nextState = a ? 2'b00 : 2'b01;
        end
      2'b01: begin
        o = 0;
        nextState = 2'b10;
        end
    endcase
  always @(posedge clk or reset)
    if (reset)
      state <= 2'b00;
    else
      state <= nextState;
endmodule
```

FSM with Combinational Logic

Output o is declared a reg because it is assigned procedurally, not because it holds state
FSM with Combinational Logic

```verilog
module FSM(o, a, b, reset);
output o;
reg o;
input a, b, reset;
reg [1:0] state, nextState;
always @(a or b or state)
  case (state)
    2'b00: begin
      o = a & b;
      nextState = a ? 2'b00 : 2'b01;
    end
    2'b01: begin
      o = 0;
      nextState = 2'b10;
    end
  endcase
always @(posedge clk or reset)
  if (reset)
    state <= 2'b00;
  else
    state <= nextState;
endmodule
```

FSM from a Single Always Block

```verilog
module FSM(o, a, b);
output o;
input a, b;
reg [1:0] state;
always @(posedge clk or reset)
  if (reset)
    state <= 2'b00;
  else case (state)
    2'b00: begin
      state <= a ? 2'b00 : 2'b01;
      o <= a & b;
    end
    2'b01: begin
      state <= 2'b10;
      o <= 0;
    end
  endcase
endmodule
```

Writing Testbenches

```verilog
module test;
reg a, b, sel;
mux m(y, a, b, sel);
initial begin
  $monitor($time, "a=%b b=%b sel=%b y=%b", a, b, sel, y);
  a = 0; b = 0; sel = 0;
  #10 a = 1;
  #10 sel = 1;
  #10 b = 1;
end
```

Simulating Verilog

Simulation Behavior

Concurrent processes (initial, always) run until they stop at one of the following
- #42 Schedule process to resume 42 time units from now
- wait(cf & of) Resume when expression “cf & of” becomes true
- @(a or b or y) Resume when a, b, or y changes
- @(posedge clk) Resume when clk changes from 0 to 1

Expression Moore machine behavior: Outputs are latched. Inputs only sampled at clock edges

Nonblocking assignments used throughout to ensure coherency. RHS refers to values calculated in previous clock cycle

Simulation Behavior

Scheduled using an event queue
Non-preemptive, no priorities
A process must explicitly request a context switch
Events at a particular time unordered
Scheduler runs each event at the current time, possibly scheduling more as a result

Two Types of Events

Evaluation events compute functions of inputs
Update events change outputs
Split necessary for delays, nonblocking assignments, etc.

Simulation Behavior

Infinite loops are possible and the simulator does not check for them This runs forever: no context switch allowed, so ready can never change

```verilog
while (˜ready)
  count = count + 1;
```

Instead, use

```verilog
wait(ready);
```

Simulation Behavior

Race conditions abound in Verilog
These can execute in either order: final value of a undefined:

```verilog
always @(posedge clk) a = 0;
always @(posedge clk) a = 1;
```

Evaluation event reads values of b and c, adds them, and schedules an update event

Evaluation event
reads values of b
and c, adds them,
and schedules an
update event

Update event
writes new value of a
and schedules any
evaluation events
that are sensitive to
a change on a

Update event
computes functions of inputs

Evaluation events
are necessary for delays, nonblocking assignments, etc.
Simulation Behavior

Semantics of the language closely tied to simulator implementation
Context switching behavior convenient for simulation, not always best way to model
Undefined execution order convenient for implementing event queue

Compiled-Code Discrete-Event Sim.

Most modern simulators use this approach
Verilog program compiled into C
Each concurrent process (e.g., continuous assignment, always block) becomes one or more C functions
Initial and always blocks split into multiple functions, one per segment of code between a delay, a wait, or event control (@)
Central, dynamic event queue invokes these functions and advances simulation time

Verilog and Logic Synthesis

Logic Synthesis

Verilog is used in two ways
Model for discrete-event simulation
Specification for a logic synthesis system
Logic synthesis converts a subset of the Verilog language into an efficient netlist
One of the major breakthroughs in designing logic chips in the last 20 years
Most chips are designed using at least some logic synthesis

Logic Synthesis Tools

 Mostly commercial tools
• Very difficult, complicated programs to write well
• Limited market
• Commercial products in $10k – $100k price range
Major vendors
• Synopsys Design Compiler, FPGA Express
• Cadence BuildGates
• Synplicity (FPGAs)
• Exemplar (FPGAs)
Academic tools
• SIS (UC Berkeley)

Logic Synthesis

Takes place in two stages:
1. Translation of Verilog (or VHDL) source to a netlist
   Register inference performed here
2. Optimization of the resulting netlist to improve speed and area
   Most critical part of the process
   Algorithms very complicated and beyond the scope of this class

Logic Optimization

Netlist optimization the critical enabling technology
Takes a slow or large netlist and transforms it into one that implements the same function more cheaply
Typical operations:
• Constant propagation
• Common subexpression elimination
• Function factoring
Time-consuming operation. Can take hours for large chips

Translating Verilog into Gates

Parts of the language easy to translate
Structural descriptions with primitives is already a netlist
Continuous assignment expressions turn into little datapaths
Behavioral statements the bigger challenge

What Can Be Translated

Every structural definition
Behavioral blocks
• Depends on sensitivity list
• Only when they have reasonable interpretation as combinational logic, edge, or level-sensitive latches
• Blocks sensitive to both edges of the clock, changes on unrelated signals, changing sensitivity lists, etc. cannot be synthesized
User-defined primitives
• Primitives defined with truth tables
• Some sequential UDPs can’t be translated (not latches or flip-flops)
What Is Not Translated

Initial blocks
- Used to set up initial state or describe finite testbench stimuli
- Don’t have obvious hardware component

Delays
- May be in the Verilog source, but are simply ignored

A variety of other obscure language features
- In general, things heavily dependent on discrete-event simulation semantics
- Certain “disable” statements
- Pure events

Register Inference

The main trick
A *reg* is not always a latch or flip-flop

Rule: Combinational if outputs always depend exclusively on sensitivity list
Sequential if outputs may also depend on previous values

Register Inference

A common mistake is not completely specifying a *case* statement

This implies a latch:

```
always @(a or b)
case {{a, b}}
  2'b00 : f = 0;
  2'b01 : f = 1;
  2'b10 : f = 1;
  default : f = 0;
endcase
```

Register Inference

The solution is to always have a default case

```
always @(a or b)
case {{a, b}}
  2'b00 : f = 0;
  2'b01 : f = 1;
  2'b10 : f = 1;
  default : f = 0;
endcase
```

Inferring Latches with Reset

Latches and Flip-flops often have reset inputs
Can be synchronous or asynchronous
Asynchronous positive reset:

```
always @(posedge clk or posedge reset)
  if (reset)
    q <= 0;
  else q <= d;
```

Simulation-synthesis Mismatches

Many possible sources of conflict
- Synthesis ignores delays (e.g., #10), but simulation behavior can be affected by them
- Simulator models X explicitly, synthesis does not
- Behaviors resulting from shared-variable-like behavior of regs is not synthesized:

```
always @ (posedge clk) a = 1;
```

New value of a may be seen by other *@ (posedge clk)* statements in simulation, never in synthesis

Summary of Verilog 1995

- Systems described hierarchically
  - Modules with interfaces
  - Modules contain instances of primitives, other modules
  - Modules contain initial and always blocks
Based on discrete-event simulation semantics
- Concurrent processes with sensitivity lists
- Scheduler runs parts of these processes in response to changes

Modeling Tools

Switch-level primitives: CMOS transistors as switches that move around charge
Gate-level primitives: Boolean logic gates
User-defined primitives: Gates and sequential elements defined with truth tables
Continuous assignment: Modeling combinational logic with expressions
Initial and always blocks: Procedural modeling of behavior
Language Features

Nets (wires) for modeling interconnection
- Non state-holding
- Values set continuously

Regs for behavioral modeling
- Behave exactly like memory for imperative modeling
- Do not always correspond to memory elements in synthesized netlist

Blocking vs. nonblocking assignment
- Blocking behaves like normal “C-like” assignment
- Nonblocking delays update, modeling synchronous behavior

Language Uses

Event-driven simulation
- Event queue containing things to do at particular simulated times
- Evaluate and update events
- Compiled-code event-driven simulation for speed

Logic synthesis
- Translating Verilog (structural and behavioral) into netlists
- Register inference: whether output is always updated
- Logic optimization for cleaning up the result

Little-used Language Features

Switch-level modeling
- Much slower than gate or behavioral-level models
- Insufficient detail for modeling most electrical problems
- Delicate electrical problems simulated with a SPICE-like differential equation simulator

Compared to VHDL

Verilog and VHDL are comparable languages
VHDL has a slightly wider scope
- System-level modeling
- Exposes even more discrete-event machinery

VHDL is better-behaved: Fewer sources of nondeterminism (e.g., no shared variables)
VHDL is harder to simulate quickly
VHDL has fewer built-in facilities for hardware modeling
VHDL is a much more verbose language: Most examples don’t fit on slides

Little-used Language Features

Delays
- Simulating circuits with delays does not improve confidence enough
- Hard to get timing models accurate enough
- Never sure you have simulated the worst case
- Static timing analysis has taken its place

In Conclusion

Verilog is a deeply flawed language
- Nondeterministic
- Often weird behavior due to discrete-event semantics
- Vaguely defined synthesis subset
- Many possible sources of simulation/synthesis mismatch

Verilog is widely used because it solves a problem
- Good simulation speed that continues to improve
- Designers use a well-behaved subset of the language
- Makes a reasonable specification language for logic synthesis
- Logic synthesis one of the great design automation success stories

Verilog 2001

Revised version of the Verilog language
IEEE Standard 1364-2001
Minor changes to the language:
- ANSI C style ports
- standard file I/O
- " attributes "
- multi dimensional arrays
- generate
- $value$plusargs
- configurations
- signed types
- localparam
- 'ifdef 'elsif 'line
- memory part selects
- automatic
- constant functions
- @*
- variable part select
- ** (power operator)
Implicit event lists

Common mistake: forgetting a variable in combinational sensitivity list

always @(a or b or c)
f = a & b | c & d;

Forgot to include d

Does not simulate like hardware behaves.

Verilog 2001’s implicit sensitivity list:

always @*    
f = a & b | c & d;

Makes process sensitive to all variables on right-hand side of assignments.

Generate

Hardware structures often very regular. Want to create them algorithmically.

Verilog’s generate: very clever macro expansion.

module gray2bin1 (bin, gray);
parameter SIZE = 8;
output [SIZE-1:0] bin;
input [SIZE-1:0] gray;

genvar i; // Compile-time only
generate for (i=0; i<SIZE; i=i+1)
begin:
    assign bin[i] = ^gray[SIZE-1:i];
end
endgenerate

Attributes

Logic synthesis has relied on hints in comments:

always @(posedge clk)
begin
    case (instr[6:5]) // synopsys full_case parallel_case
        0 : mask <= 8’h01;
        1 : mask <= 8’h02;
        2 : mask <= 8’h04;
        3 : mask <= 8’h08;
    endcase

full_case means one case will always be true,
parallel_case means at most one will be true.

Can greatly simplify the generated logic, but simulation/synthesis mismatch if assertion is not true.

Attributes

Such attributes now a first-class part of the language.
Simulator understands and checks validity.

always @(posedge clk) begin (* full_case, parallel_case=1 *)
    case (instr[6:5])
        0 : mask <= 8’h01;
        1 : mask <= 8’h02;
        2 : mask <= 8’h04;
        3 : mask <= 8’h08;
    endcase
end

ANSI C-style ports

Verilog 1995 ports could require three declarations:

module foo (myport1, myport2);
output myport1;
reg [7:0] myport1;
input [3:0] myport2;
...
endmodule

Verilog 2001 reduces this to one:

module foo (output reg [7:0] myport1, input [3:0] myport2);
...
endmodule

SystemVerilog

Much bigger change to the language.

Verification Features
assertions
biased random variables
test program blocks
process control
mailboxes
semaphores
clocking domains
direct C function calls

C++-like features
classes
dynamic arrays
inheritance
associative arrays
strings
references

Modeling Features
interfaces
dynamic processes
nested hierarchy
2-state modeling
unrestricted ports
packed arrays
implicit port connections
array assignments
enhanced literals
enhanced event control
time values & units
unique/priority case/if
logic-specific C functions
root name space access
C-like Features

New Types

<table>
<thead>
<tr>
<th>type</th>
<th>values</th>
<th>width</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>{0, 1, X, Z}</td>
<td>1+</td>
<td>✓</td>
</tr>
<tr>
<td>logic</td>
<td>{0, 1, X, Z}</td>
<td>1+</td>
<td>✓</td>
</tr>
<tr>
<td>integer</td>
<td>{0, 1, X, Z}</td>
<td>32</td>
<td>✓</td>
</tr>
<tr>
<td>bit</td>
<td>{0, 1}</td>
<td>1+</td>
<td>✓</td>
</tr>
<tr>
<td>byte</td>
<td>{0, 1}</td>
<td>8</td>
<td>✓</td>
</tr>
<tr>
<td>shortint</td>
<td>{0, 1}</td>
<td>16</td>
<td>✓</td>
</tr>
<tr>
<td>int</td>
<td>{0, 1}</td>
<td>32</td>
<td>✓</td>
</tr>
<tr>
<td>longint</td>
<td>{0, 1}</td>
<td>64</td>
<td>✓</td>
</tr>
</tbody>
</table>

reg & logic now the same: both permit either continuous or procedural assignment, but not both.

Other new types for two-valued functional simulation.

ifdef and typedef

Can define aliases for existing types. Useful, e.g., for switching between four- and two-valued simulation:

```systemverilog
'ifdef TWOSTATE
  typedef bit bit_t;
'else
  typedef logic bit_t;
'endif
```

module dff (
  output bit_t q,
  input bit_t d, clk, rst);
always @(posedge clk)
  if (rst) q <= 0;
  else q <= d;
endmodule

Structs and Unions

SystemVerilog provides C-like structs and unions in both packed and unpacked forms.

```systemverilog
typedef struct {
  logic PARITY;
  logic[3:0] ADDR;
  logic[3:0] DEST;
} pkt_t;

pkt_t mypkt;
mypkt.ADDR = 12;
```

Packed vs. Unpacked

Structs are unpacked by default. The alignment of their fields is implementation-dependent for efficiency, e.g., chosen by the C compiler.

```systemverilog
typedef struct {
  logic PARITY;
  logic[3:0] ADDR;
  logic[3:0] DEST;
} pkt_t;
31 3 1 0
```

```systemverilog
PARITY
ADDR
DATA
```

Packed Structs and Unions

```systemverilog
typedef struct packed {
  logic[15:0] source_port; tcp_t tcp_h;
  logic[15:0] dest_port; udp_t udp_h;
  logic[31:0] sequence; bit [63:0] bits;
} tcp_t;

typedef struct packed {
  logic[15:0] source_port; ip_t ip_h;
  logic[15:0] dest_port; udp_t udp_h;
  logic[15:0] checksum; // all are equivalent
  ip_h.upd_h.length = 5;
  ip_h.bits[31:16] = 5;
  ip_h.bytes[3:2] = 5;
} udp_t;
```

```systemverilog
tcp_t source_port dest_port sequence
udp_t source_port dest_port length checksum
```

Operator Overloading

SystemVerilog provides operator overloading facilities like those in C++ through the bind keyword.

```systemverilog
typedef struct {
  bit sign;
  bit [3:0] exponent;
  bit [10:0] mantissa;
} float;

bind + function float faddfr(float, real);
bind + function float faddff(float, float);
```

```systemverilog
float A, B, C, D;
assign A = B + C; // means A = faddfr(B, C);
assign D = A + 1.0; // means A = faddfr(A, 1.0);
```

Classes

SystemVerilog provides C++-like classes with automatic garbage collection.

```systemverilog
class Packet;
  bit [3:0] cmd;
  int status;
  header_t header;
  function int get_status();
  return status;
endfunction
extern task set_cmd(input bit [3:0] a);
endclass
```

```systemverilog
task Packet::set_cmd(input bit [3:0] a);
cmd = a;
endtask
initial begin
  Packet myPkt = new; // Create a new packet
end```
Inheritance

As in C++, classes can inherit from other classes:

class ErrPkt extends Packet;
  bit [3:0] err;
  // New function
  function bit [3:0] show_err;
  return err;
endfunction
  // Overrides Packet::set
  task set_cmd(input bit [3:0] a);
  cmd = a + 1;
endtask
endclass

Hardware Modeling

always_comb, _latch, and _ff

In RTL design, a Verilog always block models combinational logic, sequential logic driving flip-flops, or sequential logic driving latches, never more than one.

SystemVerilog's always_comb, always_ff, and always_latch keywords make the designer's intent clear to the compiler so it can issue error messages.

Priority/Unique

Verilog 1995 had no provision for checking uniqueness of conditions: synthesis tools placed pragmas in comments. Verilog 2001 added attributes for such conditions as first-class entities.

SystemVerilog introduces new keywords implying unique and complete conditions.

Cases must be complete  Condition must be unique

priority ✓ ✓
unique ✓ ✓ ✓

Priority Examples

// Error if none of irq0–irq2 is true
priority if (irq0) irq = 3'b1;
  else if (irq1) irq = 3'b2;
  else if (irq2) irq = 3'b3;
endcase

// Error if not exactly one of irq0–irq2 is true
unique if (irq0) irq = 3'b1;
  else if (irq1) irq = 3'b2;
  else if (irq2) irq = 3'b3;
endcase

// Error if both irq0 and irq1 are true
unique if (irq0) irq = 3'b1;
  else if (irq1) irq = 3'b2;
  else if (irq2) irq = 3'b3;
endcase

SystemVerilog:

Priority case (1'b1)
  irq0: irq = 3'b1 <= 0;
  irq1: irq = 3'b1 <= 1;
  irq2: irq = 3'b1 <= 2;
endcase

Unique/Priority

Verilog 1995 had no provision for checking uniqueness of conditions: synthesis tools placed pragmas in comments. Verilog 2001 added attributes for such conditions as first-class entities.

SystemVerilog introduces new keywords implying unique and complete conditions.

Cases must be complete  Condition must be unique

priority ✓ ✓ ✓
unique ✓ ✓ ✓

Unique Examples

// Error if not exactly one of irq0–irq2 is true
unique case (1'b1)
  irq0: irq = 3'b1 <= 0;
  irq1: irq = 3'b1 <= 1;
  irq2: irq = 3'b1 <= 2;
endcase

// Error if both irq0 and irq1 are true
unique if (irq0) irq = 3'b1;
  else if (irq1) irq = 3'b2;
  else if (irq2) irq = 3'b3;
endcase

// Error if both irq0 and irq1 are true:
unique case (1'b1)
  irq0: irq = 3'b1 <= 0;
  irq1: irq = 3'b1 <= 1;
  default: irq = 0;
endcase

Packages

package ComplexPkg;
  typedef struct {
    float i, r;
  } Complex;
  function Complex add(Complex a, b);
    add.r = a.r + b.r;
    add.i = a.i + b.i;
  endfunction
  function Complex mul(Complex a, b);
    mul.r = (a.r * b.r) + (a.i * b.i);
    mul.i = (a.r * b.i) + (a.i * b.r);
  endfunction
endpackage : ComplexPkg

module foo (input bit clk);
  import ComplexPkg::*;
  Complex a,b;
  always @(posedge clk)
    c = add(a,b);
endmodule
Implicity-named Ports

Hierarchy in Verilog usually for separating namespaces. Net and port names typically common across modules. Verbose in Verilog 1995:

```
module top;
wire [3:0] a;
wire [7:0] b;
wire [15:0] c;
foo foo(a, b, c);
bar bar(a, b, c);
endmodule
```

Implicitly-named Ports

Implicit ports plus ANSI-style declarations makes this cleaner, especially for modules with many ports.

```
module top;
wire [3:0] a;
wire [7:0] b;
wire [15:0] c;
foo foo1(*);
bar bar1(*);
endmodule
```

Port renaming also supported. Allows specific ports to be overridden or renamed as necessary.

```
module top;
wire [3:0] a;
wire [7:0] b;
wire [15:0] c;
foo foo(a, b, c);
module bar(a, b, c);
module memory(interface b);
module cpu(simple_bus bus, interface b);
module memory(interface b);
module cpu(simple_bus bus, interface b);
endmodule
endmodule
```

Interfaces

For communication among modules. Like a collection of shared variables.

```
interface simple_bus;
logic req, gnt, rdy;
logic start, rdy;
endinterface : simple_bus
```

Interfaces with implicit ports

Even more simple. Use the same names and let the compiler do the rest.

```
interface simple_bus;
logic req, gnt, rdy;
logic start, rdy;
endinterface : simple_bus
```

Generic bundles

You can leave the exact type of an interface unspecified to allow different implementations. Must connect explicitly.

```
interface simple_bus;
logic req, gnt, rdy;
logic start, rdy;
endinterface : simple_bus
```

Tasks and Functions in Interfaces

A way to constrain signal directions in interfaces.

```
module top;
logic clk = 0, bus_error;
interface bus( input bit clk, output bit bus_error);
logic req, gnt, rdy;
logic [7:0] addr, data;
logic [1:0] mode;
logic start, rdy;
endinterface : bus
```

Ports on interfaces

Interfaces are groups of shared variables. Ports on interfaces can bring connections in or out.

```
interface bus( input bit clk, output bit bus_error);
logic req, gnt, rdy;
logic [7:0] addr, data;
logi[1:0] mode;
logic start, rdy;
endinterface : bus
```

Modports in interfaces

A way to constrain signal directions in interfaces.

```
modport slave( input req, addr, clk, output gnt, rdy, inout data);
modport master( output req, addr, gnt, clk, output gnt, rdy, inout data);
endinterface : bus
```
Dynamically-sized Arrays

Truly software-like behavior.
```verilog
module dynamic_array;
  bit[3:0] myarray[]; // Creates null reference
  initial begin
    myarray = new[4]; // Allocate four 4-bit words
    // Double the size of the array, preserving its contents
    myarray = new[myarray.size() * 2](myarray);
  end
endmodule
```

Associative Arrays

Very abstract notion. Like maps in C++, hashtables in Java, or associative arrays in Perl, Python, Awk.
```verilog
module associative_array;
  typedef struct packed {
    int a;
    logic [7:0] b;
  } mykey_t;
  int myarray[mykey_t];
  // new, empty associative array
  initial begin
    mykey_t key1 = {-3, 8'xFE}; // structure literal
    myarray[key1] = 10;
    if (myarray.exists(key1))
      myarray[key1] = -5;
    myarray.delete(key1);
  end
endmodule
```

Queues

Often used to communicate between processes.
```verilog
module queues;
  int q[$] = { 2, 4, 8 }; // initial contents
  int sq[$:15]; // maximum size is 16
  initial begin
    int e = q[0]; // first item: 2
    e = q[$]; // last item: 8
    q = { q, 6 }; // append: now 2, 4, 8, 6
    q = { e, q }; // insert: now 8, 2, 4, 8, 6
    q = q[1:$-1]; // remove: now 4, 8
  end
endmodule
```

Process Management: join

Fork starts processes; join terminates when all blocks terminate.
```verilog
fork
  begin
    $display("0ns have elapsed\n");
    # 20ns; // delay
  end
begin
  # 20ns;
  $display("20ns have elapsed\n");
  # 5ns;
end
join
  # 5ns;
  $display("30ns have elapsed\n");
```
Semaphores

Mutually-exclusive keys in a bucket. get blocks if not enough keys are available.

```verilog
semaphore we_are_there = new; // initialize with no keys
task drive;
    fork
        begin
            # 100ns; // delay 100ns
            we_are_there.put(1); // put a single key in the semaphore
        end
        $display("Are we there yet?\n");
        we_are_there.get(1); // wait for a key
        $display("We made it\n");
    end
endtask
```

Semaphores and events

```verilog
event ask, answered;
semaphore answer = new; // only valid after answer
int winner; // only valid after answer
task gameshow;
    fork
        begin
            // the host
            $display("Id was first\n");
            winner = 1; // try to answer first
        end
        begin
            // contestant one
            think_about_answer(); answer.get(1); // wait for a key
            $display(\"\%d was first\n", winner);
        end
        begin
            // contestant two
            think_about_answer(); answer.get(1);
            // signal our success
        end
    end
endtask
```

Mailboxes

Possibly bounded semaphore-like queues.

```verilog
mailbox #(string) mybox = new(2); // capacity set to two
task mailbox_demo;
    fork
        mybox.put("first letter");
        $display("sent first\n");
        mybox.put("second letter");
        $display("sent second\n");
    end
    $display("got first letter\n");
    $display("got second letter\n");
end
```

Verifiication Features

Constrained Random Variables

Manually creating test cases tedious and difficult, yet appears necessary for functional verification.

Current best practice: Constrained random tests.

SystemVerilog has features for creating such tests.

```verilog
class Bus;
    rand bit[15:0] addr;
    rand bit[31:0] data;
endclass
Bus bus = new;
repeat (50) begin
    if (bus.randomize() with { addr[31] == 0 })
        $display("addr = %h data = %h\n", bus.addr, bus.data);
    else
        $display("overconstrained: no satisfying values exist\n");
end
```

Adding constraints

```verilog
class Bus;
    rand bit[15:0] addr;
    rand bit[31:0] data;

class MyBus extends Bus;
    rand AddrType atype;
endclass
task exercises_bus;
    int res;
    // Restrict to low addresses
    res = bus.randomize() with { atype == low; };
    // Restrict to particular address range
    res = bus.randomize() with { 10 <= addr && addr <= 20 };
    // Restrict data to powers of two
    res = bus.randomize() with { data & (data - 1) == 0 };
    // Disable word alignment
    bus.word_align.constraint_mode(0);
    res = bus.randomize() with { addr[0] || addr[1] };
    // Re-enable word alignment
    bus.word_align.constraint_mode(1);
endtask
```

Layering constraints

Constraints inherited, can be added in derived classes.

```verilog
typedef enum ( low, mid, high ) AddrType;
class MyBus extends Bus;
    rand AddrType atype;
endclass
```

Using Constraints

Very powerful constraint solving algorithm.

```verilog
task exercise_bus;
    int res;
    // Restrict to low addresses
    res = bus.randomize() with { atype == low; };
    // Restrict to particular address range
    res = bus.randomize() with { 10 <= addr && addr <= 20 };
    // Restrict data to powers of two
    res = bus.randomize() with { data & (data - 1) == 0 };
    // Disable word alignment
    bus.word_align.constraint_mode(0);
    res = bus.randomize() with { addr[0] || addr[1] };
    // Re-enable word alignment
    bus.word_align.constraint_mode(1);
endtask
```
Other types of constraints

// Set membership constraints
rand integer x, y, z;
constraint c1 { x inside {3, 5, [9:15], y:2*y}, z; }

integer fives[0:3] = { 5, 10, 15, 20 };
rand integer v;
constraint c2 { v inside fives; }

// Distribution constraints
rand integer w;
// make w 100 1/8 of time, 200 2/8, 300 5/8
constraint c3 { w dist {100 := 1, 200 := 2, 300 := 5}; }

// Implication constraints
bit [3:0] a, b;
// force b to 1 when a is 0
constraint c4 { (a == 0) -> (b == 1); }

Many, many more features

Variables that step through random permutations (randc)
If-then-else constraints
Algorithmic constraints over array entries (foreach)
Constraints among multiple objects
Variable ordering constraints (solve..before)
Static constraints controlled by one constraint
mode() call
Functions in constraints
Guarded constraints
pre- and post-randomize functions
Random variable disabling
Explicit randomization of arbitrary variables
Random sequence generation from a grammar

Coverage Checks

Once we have generated our tests, how good are they?
Current best practice: monitoring and improving coverage
Coverage: how many cases, statements, values, or combinations have the test cases exercised?

Covergroup

Defines something whose coverage is to be checked.
Creates bins and tracks whether values ever appeared.

// color: a three-valued variable whose coverage is to be checked
class color;
enum { red, green, blue } color;
covergroup g1 @(posedge clk); // Sample at posedge clk
color c;
coverpoint color; 
endgroup
g1 g1_inst = new; // Create the coverage object

At the end of simulation, reports whether color took all three of its values.

Cross Coverage

May want to monitor combinations of variables.

class color;
enum { red, green, blue } color;
covergroup g2 @(posedge clk);
color c;
coverpoint color, pixel_hue, pixel_offset; 
endgroup
g2 g2_inst = new; // Create a watcher

User-defined bins

May only want to track certain values of a variable.

bit [3:0] a; // Takes values 0-1023
covergroup c9 @(posedge clk);
coverpoint a { 
// Place values 0-63 and 65 in bin a
bin a[0..63], 65 );
// Create 65 bins, one of 127, 128, ... 191
bin b[1] = ( [127:150], [141:191] );
// Create three bins: 200, 201, and 202
bin c[1] = ( 200, 201, 202 );
// Place values 1000-1023 in bin d
bin d = (1000:$); 
// Place all other values (e.g., 64, 66, ..., 126, 192, ...) in their own bin
bin others[] = default;
}
endgroup
c9 c9_inst = new; // Create a watcher

Covering Transitions

May want to check transitions, not just a variable's values.

bit [3:0] a;
covergroup c9 @(posedge clk);
coverpoint a { 
// Place any of the sequences 4-5-6, 7-11, 8-11, 9-11, 10-11, 11, 7-12, 8-12, 9-12, and 10-12 into bin sa
bin sa = (4 => 5 => 6, [7:9] => 10 => 11, );
// Create separate bins for 4-5-6, 7-10, 8-10, and 9-10
bin sa[] = (4 => 5 => 6), ([7:9] => 10); 
// Look for the sequence 3-3-3-3
bin sc = 3 *[4];
// Look for any of the sequences 5-5-5-5, 5-5-5-5, 5-5-5-5
bin ed = 5 *[2:4]; 
// Look for any sequence of the form 6-6-6-6-6-6-6-6-6
where "-" represents any sequence that excludes 6
bin sa[] = 6 *[3];
}
endgroup
Assertions

We have generated our tests, they do a reasonable job covering the design, but how do we find problems?

Current best practice: Add assertions to the design that check for unwanted conditions.

Currently, the most effective way to reduce debugging time: bugs found more quickly, and easier to remedy.

Long used in software, growing use in hardware.

Main challenge in hardware: asserting temporal behavior. SystemVerilog has constructs specifically for checking sequences of things.

Immediate Assertions

Simplest assertions check an condition only when they are executed.

// Make sure req1 or req2 is true if we are in the REQ state
always @(posedge clk)
  if (state == REQ)
    assert (req1 || req2);

// Same, but report the error ourselves
always @(posedge clk)
  if (state == REQ)
    assert (req1 || req2)
  else
    $error("In REQ; req1 || req2 failed (%0t)", $time);

Concurrent Assertions

Another example: make sure the address strobe is not true for two consecutive cycles.

property no_two_astr;
  @(posedge clk) // Unless reset is true, make sure astr is not true for two cycles in a row.
    disable iff (reset) not (astr [*2]);
endproperty

assert property (no_two_astr);

Sequences and Properties

Sequences can be defined in isolation and used elsewhere.

// The own_bus signal goes high in 1 to 5 cycles, then the breq signal goes low one cycle later.
sequence own_then_release_breq;
  ##[1:5] own_bus
  ##1 !breq
endsequence

// Once breq has risen, own_bus should rise and breq should fall.
property legal_breq_handshake;
  @(posedge clk) // On every clock.
    disable iff (reset) // unless reset is true,
    once breq has risen, own_bus should rise and breq should fall.
    $rose(breq) |=-> own_then_release_breq;
endproperty

assert property (legal_breq_handshake);

Sequences (partial syntax)

seq :=
  expr Expression over signals
  expr [* int-or-range ] Consecutive repetition
  expr [= int-or-range ] Non-consecutive repetition
  expr [-> int-or-range ] Goto repetition
  seq [== int-or-range seq ] Delay between sequences
  seq or seq Either true
  seq and seq Both true
  seq intersect seq Both true, end simultaneously
  seq within seq Second starts/ends within first

Properties (partial syntax)

prop :=
  seq Sequence
  prop or prop Either holds
  prop and prop Both hold
  not prop Does not hold
  seq |=-> prop Prop holds when sequence ends
  seq [==] prop Prop holds cycle after sequence ends
  if ( expr ) prop If-then-else
  [ else prop ]