Verilog HDL and Its Ancestors and Descendants

PETER FLAKE, Elda Technology Ltd, UK
PHIL MOORBY, Rockport, Mass., USA
STEVE GOLSON, Trilobyte Systems, USA
ARTURO SALZ, Synopsys, Inc, USA
SIMON DAVIDMANN, Imperas Software Ltd, UK

Shepherd: Keshav Pingali, University of Texas at Austin, USA

This paper describes the history of the Verilog hardware description language (HDL), including its influential predecessors and successors. Since its creation in 1984 and first sale in 1985, Verilog has completely revolutionized the design of hardware. Verilog enabled the development and wide acceptance of logic synthesis. For large-scale digital logic design, previous schematic-based techniques have transformed into textual register-transfer level (RTL) descriptions written in Verilog. As of 2018 about 80% of integrated circuit design teams worldwide use Verilog and its compatible descendant SystemVerilog.

CCS Concepts: • **Hardware → Hardware description languages and compilation**; *Software tools for EDA*; Combinational synthesis; Sequential synthesis.

Additional Key Words and Phrases: Verilog, SystemVerilog, HILO, Superlog, Vera, HDL, HVL

ACM Reference Format:
Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. 2020. Verilog HDL and Its Ancestors and Descendants. *Proc. ACM Program. Lang.* 4, HOPL, Article 87 (June 2020), 90 pages. https://doi.org/10.1145/3386337

Contents

Authors' addresses: Peter Flake, Elda Technology Ltd, UK, flake@elda.demon.co.uk; Phil Moorby, Rockport, Mass., USA, prm1024@gmail.com; Steve Golson, Trilobyte Systems, USA, sgolson@trilobyte.com; Arturo Salz, Synopsys, Inc, USA, arturo.salz@synopsys.com; Simon Davidmann, Imperas Software Ltd, UK, simond@imperas.com.

2475-1421/2020/6-ART87
https://doi.org/10.1145/3386337

1 INTRODUCTION

Verilog is by far the most widely used hardware description language (HDL) today.

This paper is a historical account of Verilog, the HDLs that preceded Verilog, and the extensions and enhancements of Verilog into Superlog and SystemVerilog. Figure 1 shows the family tree of these HDLs. We explore the relationships between these HDLs, their syntax and behavior, and the individuals and corporations that developed them.

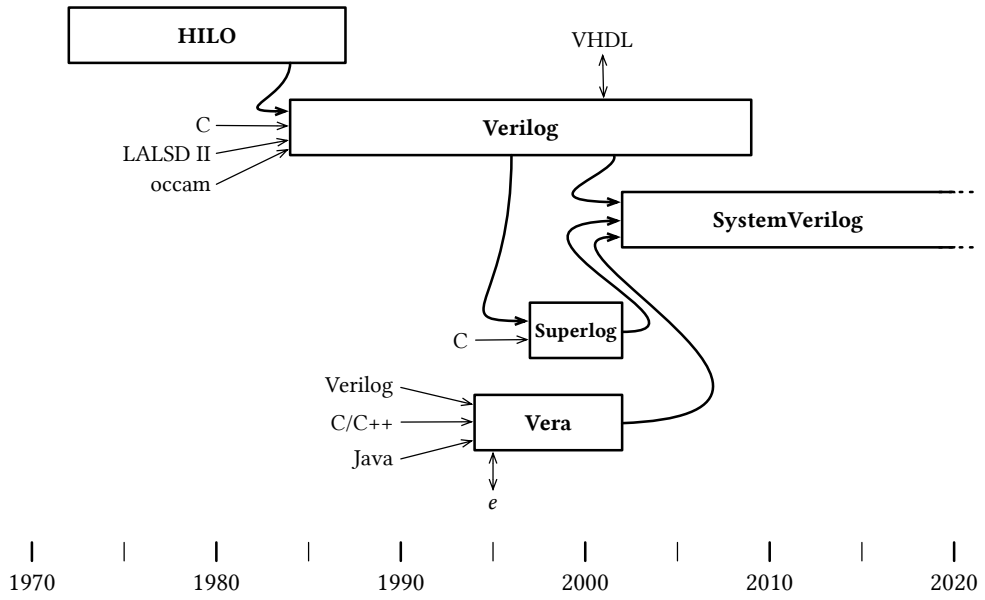


Fig. 1. The Verilog family tree, 1972–2020 and beyond.

1.1 Characteristics of Hardware Description Languages

A hardware description language is a programming language tuned to the overall needs of describing hardware. As such, an HDL has several key attributes that distinguish it from a conventional programming language:

Inherent parallelism: Hardware is inherently parallel, so a good HDL must be capable of describing this parallelism.

Support for timing: Clocks and delays are fundamental concepts in hardware, so an HDL must be capable of expressing clocks, sequencing of operations, and delays in signals.

Supremacy of the bit: Hardware constructs such as buses are inherently data-parallel so the HDL must allow a description of such constructs using vectors while maintaining full awareness of the underlying bits.

Connectivity: Express structural hierarchy, and signal propagation along nets.

Also, an HDL must be supported by tools for *verifying* hardware designs and for *synthesizing* hardware from these designs.

This paper is organized as follows. The rest of Section 1 introduces the design flow and the sequence of tools that are used to design a chip. Section 2 discusses the history of HDLs. Section 3 discusses the HILO family of HDLs. Section 4 introduces Verilog. Section 5 discusses how HDLs and synthesis have revolutionized ASIC design. Section 6 discusses the Superlog extensions to the base Verilog language. Section 7 introduces the Vera verification language, while Section 8 discusses assertions. Finally Section 9 discusses how all these parts were woven together to create the SystemVerilog language. Section 10 briefly reviews some other hardware languages. A detailed Timeline is on Page 69. More information about People mentioned in this paper can be found on Page 73, while a Glossary of terms and acronyms can be found on Page 75.

1.2 An Example Design Flow Using a Modern HDL

To understand how HDLs are used, it is useful to consider a design flow for a circuit. There are four major steps to complete when designing an integrated circuit:

High-Level and Logic Design: Create the architecture. Specify and implement the required system behavior.

Verification: Confirm that the design description meets the specification. Run tests that verify correct operation. Simulate, check, and prove correctness using formal methods.

Logic Synthesis: Convert the high-level description into a gate-level logical netlist.

Physical Implementation: Create a physical layout suitable for fabrication.

Figure 2 shows a typical modern-day design flow, indicating the use of Verilog and SystemVerilog throughout the flow.

Consider a hardware engineer who has been asked to design a 4-bit shift register, with parallel load capability, and an asynchronous active-low clear. Using the SystemVerilog HDL, our engineer might write the following *behavioral description* (or *register transfer level (RTL) description*):[1]

[1]The specifics of these languages will be explained later in the paper. This section is just an overview giving examples of how these languages are used.

```
module shiftRegister4 (                     // declare the module name
      input clock, clear, load, serialIn,   // declare inputs and outputs
      input [3:0] dataIn,
      output [3:0] dataOut);
  reg [3:0] q;                              // 'q' is four-bit register
  assign dataOut = q;                       // connect register to outputs
  always @ (posedge clock or negedge clear) // when an input changes:
    if (!clear) q <= 0;                        // clear, or
    else        q <= load ? dataIn             // load, or
                          : {serialIn,q[3:1]}; // shift
endmodule
```

Verification can be done in several ways. The most important is to use a *logic simulator* tool to exercise this behavioral hardware design with a description of its inputs (stimulus). This stimulus comes from a *testbench* which is also written in SystemVerilog:

```
// Simple testbench for the shift register design
module top();
   logic clock, clear, load, serialIn; // connections to shift register
   logic [3:0] dataIn, dataOut;        //
   parameter halfPeriod = 5;

   // instantiate the shift register module
   shiftRegister4 dut (clock,clear,load,serialIn,dataIn,dataOut);

   always #halfPeriod clock = ~clock; // oscillating clock signal

   initial begin
      clock = 0;                    // set initial states of inputs
      clear = 1;
      serialIn = 1;

      load = 1;                     // enable parallel load
      dataIn = 4'b0010;             // loading binary data 0010
      @(negedge clock);             // after load, check expected value
      if (dataOut !== 4'b0010)
        $display("Error: bad parallel load");

      load = 0;                     // enable serial shift
      repeat(2) @(negedge clock);   // shift for two clocks and check
      if (dataOut !== 4'b1100)
        $display("Error: bad serial shift");

      $finish;                      // end the simulation
   end
endmodule
```

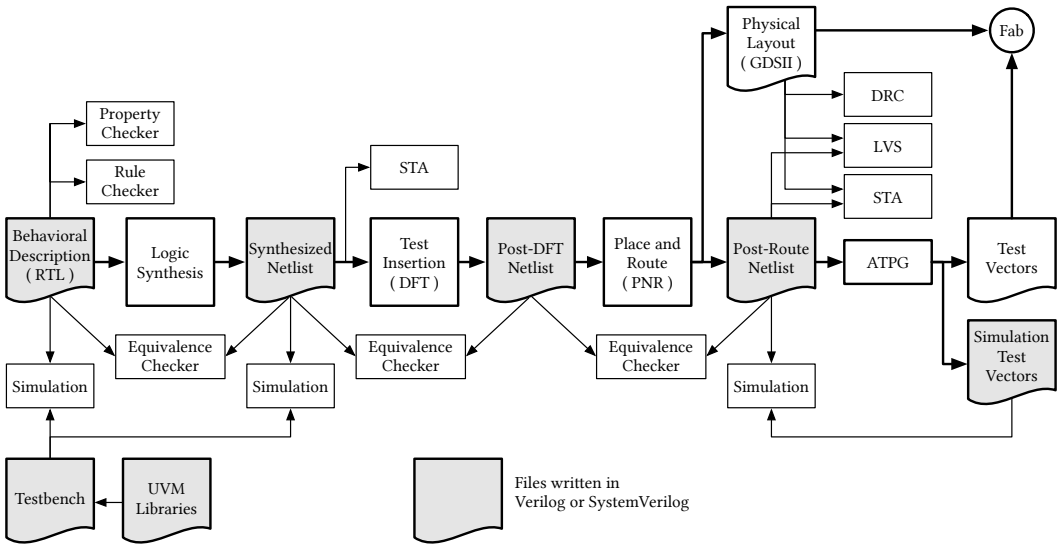


Fig. 2. Tool flow diagram for IC design, showing usage of Verilog throughout the flow.

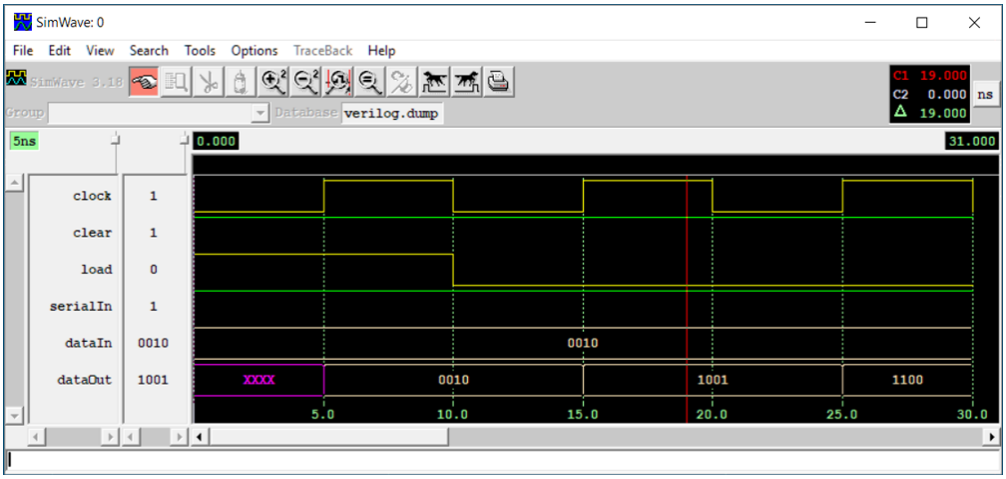


Fig. 3. Waveform display from logic simulator, showing results from execution of the simple testbench.

Logic simulation has some resemblance to the execution of a software program. The simulation results can be automatically compared with expected outputs (response) and they can be viewed by designers using *waveform display* tools (Fig. 3). This simple testbench does some self-checking to ensure proper operation of the *design under test (DUT)*.

Other verification methods do not use a stimulus, rather they operate on the behavioral description (source) directly. *Rule checking* tools look for such things as uninitialized variables, like similar tools for programming languages. *Property checkers* use assumptions to verify assertions formally. These are useful for finding bugs that manifest themselves only in rare cases.

Continuing the design process, our engineer uses a *logic synthesis* tool to convert the behavioral description into an optimized structural (gate-level) description called a *netlist*:

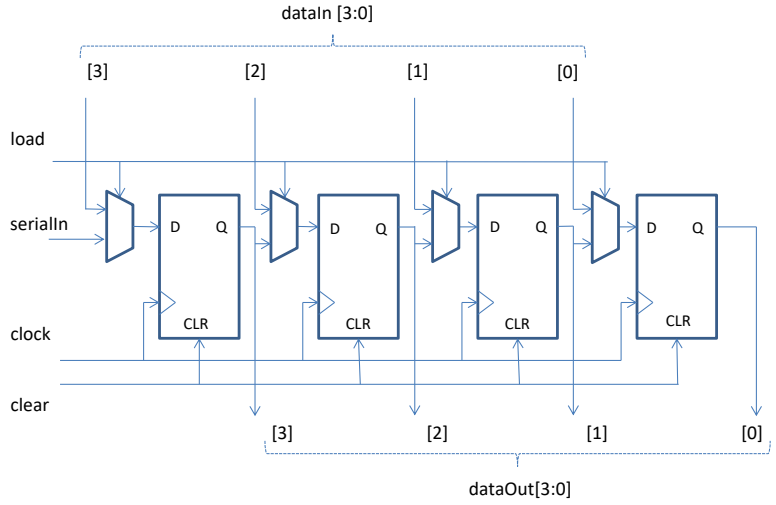


Fig. 4. Schematic view of synthesized example 4-bit shift register.

```
module shiftRegister4 ( clock, clear, load, serialIn, dataIn, dataOut );
  input [3:0] dataIn;
  output [3:0] dataOut;
  input clock, clear, load, serialIn;
  wire   n1, n2, n3, n4;
  FD2 q_reg_3  ( dataOut[3], clock, clear, n4 );
  FD2 q_reg_2  ( dataOut[2], clock, clear, n1 );
  FD2 q_reg_1  ( dataOut[1], clock, clear, n2 );
  FD2 q_reg_0  ( dataOut[0], clock, clear, n3 );
  MUX21 mu4 ( n4, load, serialIn, dataIn[3] );
  MUX21 mu3 ( n3, load, dataOut[1], dataIn[0] );
  MUX21 mu2 ( n3, load, dataOut[2], dataIn[1] );
  MUX21 mu1 ( n3, load, dataOut[3], dataIn[2] );
endmodule
```

This netlist description also uses SystemVerilog syntax, with an alternative form of inputs and outputs. Figure 4 shows the schematic for this netlist.

Formal equivalence tools (checkers) can verify that two hardware descriptions are externally the same and have the same behavior. Our engineer can use such a tool to confirm that the behavioral description and the netlist perform the same logical function.

The logic simulator can also be used here. Since the netlist syntax is SystemVerilog, the same testbench used on the behavioral description can be used to exercise this gate-level structural description, verifying that the response is still as expected.

Another step in the flow involves adding extra logic to the netlist to support testing during manufacturing. This is called *design for test (DFT)*. After these modifications are done, the equivalence checker can confirm that the post-DFT netlist has the same function as the pre-DFT netlist, verifying that the insertion of test logic had no unintended effect on the normal logic. These manufacturing tests may be created using *automatic test pattern generation (ATPG)* tools. The logic simulator can be

used to verify that these ATPG vectors (stimulus and response) properly execute on the gate-level netlist.

After this, the netlist is handed off to a *place and route (PNR)* tool to create the physical description (transistors and wires) for this design. The logical design is modified further by PNR, perhaps adding buffers on long routes, and using *clock tree synthesis (CTS)* to build the clock buffer tree. The PNR tool will generate a post-route netlist. Again the formal equivalence checker will confirm that the functionality has stayed constant, even as the design description has changed.

Many other tools are used in the design flow. For example, *static timing analysis (STA)* tools are used to verify correct timing behavior of the netlist. *Layout versus schematic (LVS)* checks are done to ensure that the physical layout (wires and transistors) exactly agrees with the logical post-route netlist. *Design rule checks (DRC)* verify that none of the complex layout rules have been violated. After all checks are complete, the design is ready for *sign-off* and sent to the *fab*.

The behavioral description, testbench, and all netlists use SystemVerilog syntax. All the tools in the flow, from many different vendors, rely on the same HDL. The rest of this paper explains the history and development of Verilog and SystemVerilog, and how they gained such a dominant position.

2 EARLY HARDWARE DESCRIPTION LANGUAGES

The earliest reference to an HDL may be found in Claude Shannon's 1940 master's thesis, which introduced a method for describing circuit behavior using Boolean algebra, manipulating these equations into their simplest form, and then synthesizing the corresponding relay switching circuits [Shannon 1940]. Then, in 1952 Irving S. Reed proved that Boolean algebraic equations can be physically realized as electronic circuits and recommended "...in the initial synthesis of a digital computer it is desirable to concentrate one's attention on the abstract model of the digital computer [Reed 1952]." By 1956 Reed and his colleagues had extended this work into a "register transfer language" or RTL, which was used in the design and simulation of several special and general purpose computers [Reed 1956].

Research and development into what were then called "computer description languages" continued through the 1960s [Darringer 1968]. The *Workshop on Computer Description Languages*, held in May 1973 at Rutgers University, was the first conference dedicated to this topic [SIGARCH 1973]. The following year a second conference was held in Germany, however now named the *Workshop on Computer Hardware Description Languages* [SIGARCH 1974] and this conference series began to be called *CHDL*. By late 1973 the term "hardware description language" or HDL was in common use [Jordan and Smith 1973]. A 1974 survey listed over fifty HDLs from all over the world [IEEE 1974] including HILO at Brunel University in the UK. These HDLs were mainly developed at universities and in industrial research laboratories.

A shift away from simple netlists and toward traditional computer languages occurred in the 1970s. For example the ISP language, which was originally conceived in 1971 to describe a computer's instruction processor, was generalized and extended to conceive the concept of Register Transfer Level (RTL) [Barbacci et al. 1972]. Two ISP-derived languages, ISPL and ISPS, were introduced in the mid-1970s. ISPS was well suited to express the relationship between the inputs and outputs of a design, and was thus adopted by Digital Equipment Corporation (DEC) to describe the PDP-16 minicomputer. These RTL languages, which were promoted by DEC, were not widely adopted and were abandoned by the mid-1980s.

The techniques that emerged in the 1980s, particularly very large-scale integration (VLSI), motivated the pursuit of more powerful languages that could sustain those new technologies. The

introduction of *logic synthesis*[2] put HDLs at the forefront of digital design [Brayton et al. 1984]. A logic synthesis tool compiles HDL code into an optimized physical netlist suitable for manufacture. At first, designers using the labor-intensive schematic capture and manual layout were able to create smaller and faster circuits than those produced by synthesis. However synthesis quality improved rapidly, and the productivity advantages offered by synthesis quickly replaced manual techniques, turning synthesis into the preeminent method of digital circuit design.

3 HILO

3.1 HILO 1

The HILO project[3] started at the University of Bradford in 1972, when the simulation of a digital system was being researched as an alternative to using a hardware prototype for design and test-bench, especially under fault conditions. Ian White, the first student on the project, was supported by Smiths Industries Avionics Division, which produced military digital systems. Under the supervision of Gerry Musgrave, White was joined by Peter Flake. Later Musgrave and Flake moved to Brunel University and in 1974 were joined by Mike Shorland.

HILO 1 [Flake et al. 1975a, 1974, 1975b; White 1975] was designed as an efficient logic simulator at a time when computer resources were very limited: the University of Bradford had an ICL 1909 mainframe with 32K words of magnetic core memory. The objective was to minimize the memory and CPU time used. It had one language for hardware description, and a different language for stimulus and simulator control. The HDL is completely declarative, consisting of a hierarchical netlist of sub-units, primitive gates, and flip-flops, as well as configurable functions (sum of product expressions and partial truth tables). The basic syntax is:

```
<output_wire_list> = <function_type> ( <input_list> ) [ comment ]
```

The wires are unidirectional, so each name can appear in the left hand (output) list only once. An input can be a wire name, with an optional '-' for inversion, or a constant. The originality at the time was in the use of vectors of wires inspired by DDL [Duley and Dietmeyer 1968]. There are `MERGE` and `SPLIT` functions for concatenation and slicing, as well as `RSHIFT` and `LSHIFT` functions. For example, here is the definition of a 4-bit shift register called `RSH` made of D-type edge-triggered flip-flops and multiplexers, and illustrated in Fig. 5:

```
#BLOCK
  DATAOUT(4) = %RSH4(DATAIN(4), LOAD, SERIALIN, CLOCK, CLEAR)
  R(4) = RSHIFT (DATAOUT(), SERIALIN)
  S(4) = %MUX4 (DATAIN(), R(), LOAD)           [1 INSTANCE OF MUX4]
  DATAOUT(4) = DET(S(), CLOCK, 1111, CLEAR)    [4 INSTANCES OF DET]
#END
#BLOCK
  OUT(4) = %MUX4(A(4), B(4), S) [A 4 BIT MULTIPLEXER WITH 2 DATA INPUTS]
  OUT(4) = OR (AA(), BB())      [4 INSTANCES OF 2-INPUT OR GATE]
  AA(4) = AND (A(), S)        [4 INSTANCES OF 2-INPUT AND GATE WITH COMMON S]
  BB(4) = AND (B(), -S)       [INVERTED INPUT S]
#END
```

[2]Brayton coined the term *logic synthesis* to denote compilation and optimization of a design description into digital hardware [Brayton et al. 1984] Earlier usage of *synthesis* meant straightforward transformation into circuitry with little if any optimization.

[3]Initially the project and resulting simulator were both called simply *HILO*. In this paper we use the name *HILO 1* to distinguish this early simulator from the later *HILO 2* and *HILO 3*. We use *HILO* to refer to the overall project.

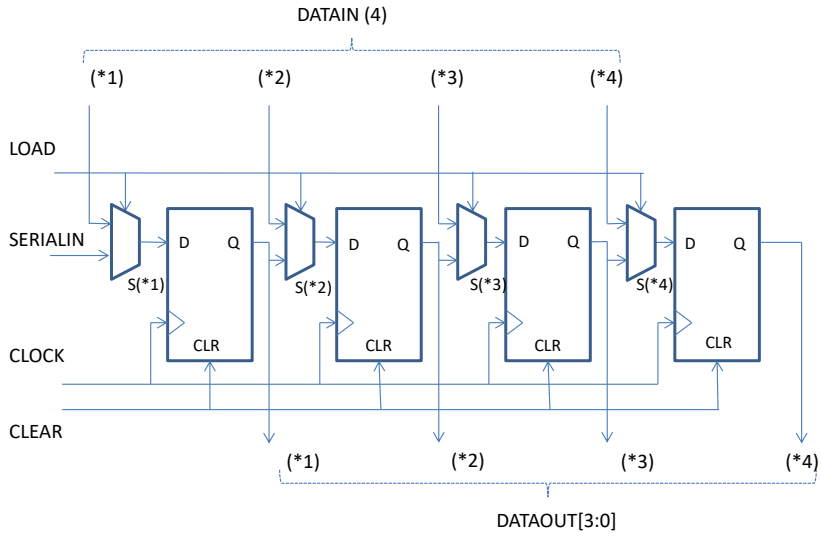


Fig. 5. 4-bit shift register example, using HILO 1 notation.

The first line of RSH defines the ports (external connections) as a vector of 4 output wires, a vector of 4 input wires, and four scalar inputs. The second line declares a vector R of four wires as three left-hand bits of DATAOUT and one SERIALIN. The third line declares an instance of a 4-bit 2-input multiplexer which is defined elsewhere and has output S. The empty parentheses () indicate the whole vector and are equivalent to (4) in this case. The fourth line declares DATAOUT to be the output of four D-type edge-triggered flip-flops, which share the CLOCK and CLEAR inputs, and have the preset input unused (set to 1).

The first line of MUX defines the ports as an output vector of four wires, two input vectors of 4 wires, and a scalar input. The next three lines define the output as the OR of two AND gates, one of which has an inverted input. Note that the multiplexer is not drawn as a single block on the diagram for clarity. All the names are in upper case because the ICL 1909 peripherals would handle only upper case.

Delays, fault injection, tracing points, and initial conditions are treated separately from the netlist. A function can be locally identified by any of its outputs, and a hierarchical path name can identify a unique instance of a function, sub-unit, or wire.

The stimulus language was procedural with absolute time steps, time being just an integer. The following shows the same stimulus as used on Page 5:

```
#WAVEFORM
   @0: CLOCK = 0, CLEAR = 1, LOAD = 1, DATAIN = 0010, SERIALIN = 1
   @5: CLOCK = 1
   @10:/IFNOT(DATAOUT = 0010)/PRINTID(DATAOUT), CLOCK = 0
   CLOCK = CHANGE(@15, &5, &5)  [@ IS ABSOLUTE, & IS RELATIVE]
   @30:/IFNOT(DATAOUT = 1100/PRINTID(DATAOUT), CLOCK = 0
#END
```

This also shows a construct for specifying a sequence of change times for a single identifier, and commands for simulator control, such as printing a particular value, controlling tracing, or

stopping the simulator. They included /ATNEXT and /ATEACH commands to act on a change of value and could be entered interactively.

The implementation of HILO 1 was done in assembly code for the ICL 1900 series computers, which had a 24-bit word which could be accessed as 6-bit characters. For ease of implementation, the vector size was limited to 24 bits, allowing a vector state to be represented in one word. A wire state was only binary, which was good for efficiency but sometimes made it hard to find a consistent initial state even though a levelization algorithm was used.

After initialization, the simulation was event directed and the scheduler was based on Ulrich's time wheel [Ulrich 1969]. Scalar wires used static memory, and the delay was inertial and deterministic; pulses shorter than the delay were rejected, and pulses equal to the delay were passed through. Vector wires used dynamic memory allocation for event scheduling, because it was possible to have each bit changing at a different time.

The resulting simulator was benchmarked against a commercial simulator and used an order of magnitude less time and (data) memory. However, its limitations were: not handling three-state logic, not having an unknown initial value, and not having a portable implementation. That meant it was difficult to break out of its university environment, even though the project had industrial sponsorship.

3.2 HILO 2

To overcome the limitations stated above, HILO 2 was a complete re-design and re-implementation of HILO 1, financed by the UK Ministry of Defence with Smiths Industries as prime contractor and Brunel University as sub-contractor. HILO 2 work began in 1976 with Gerry Musgrave as "Managerial Authority" and Peter Flake as "Technical Authority." Phil Moorby joined the team part-time, spending the rest of his time working at Brunel on his PhD on timing analysis. The project goal was to produce a simulator for production test of digital boards and custom ICs as they existed in the late 1970s, so it included fault simulation and automatic test pattern generation.

Functional board testing involved applying stimuli to the board connector and observing responses. It was helpful to use simulation to develop the test, and to use fault simulation to measure the coverage of stuck-at faults and record responses for each fault. (This is like mutation testing for software, where a mutation is an input or output held at a constant 1 or 0.) In those days the only alternative means of production test was to use a "bed of nails" to separately test each component and interconnection. This allowed each component input or output to be sensed or over-driven by the tester.

The hardware description language had to cope with three-state buses, as well as wired-or and wired-and buses. This meant that a neat division into inputs and outputs could not be used, and output names could not be used to identify gates or sub-units. The language also had to cope with complex chips where only a functional or behavioral description was available, so register transfer level (RTL) constructs were added.

3.2.1 Structural Description. To model the various kinds of buses, as well as behaviors without any drive, HILO 2 provides a range of wire types: INPUT, INPUT1, INPUT0, UNID, WAND, WOR, TRI, TRI1, TRI0, SUPPLY1, and SUPPLY0. A uni-directional UNID wire is driven by a single gate output. A bus is a wire (or set of wires) that may be driven by more than one gate output. Three-state buses allow the same wires to be shared by multiple devices; they avoid contention by enabling only one output at a time. A wired-and or wired-or bus uses a Boolean function to resolve the bus value. In the case of a three-state bus, each gate can drive a 0 value, a 1 value, or no drive at all, which is called a high-impedance state and is denoted by Z. Thus if no gate drives a (1 or 0) value onto a TRI bus, it resolves to a Z value. Similarly, if two (or more) gates drive the bus with opposite Boolean

values, the contention results in an unknown value, denoted by X. The following tables shows the resolved bus value from two driving gates:

TRI	0	1	Z	X
0	0	X	0	X
1	X	1	1	X
Z	0	1	Z	X
X	X	X	X	X

TRI0	0	1	Z	X
0	0	X	0	X
1	X	1	1	X
Z	0	1	0	X
X	X	X	X	X

TRI1	0	1	Z	X
0	0	X	0	X
1	X	1	1	X
Z	0	1	1	X
X	X	X	X	X

The X value is also useful to represent a consistent uninitialized state (see Section 4.11). Consequently, HILO 2 averts the HILO 1 problems by extending the Boolean operators with a four-valued logic:

AND	0	1	Z	X
0	0	0	0	0
1	0	1	X	X
Z	0	X	X	X
X	0	X	X	X

OR	0	1	Z	X
0	0	1	X	X
1	1	1	1	1
Z	X	1	X	X
X	X	1	X	X

NOT	
0	1
1	0
Z	X
X	X

From an initial all-X state, a "reset" sequence is typically used to clear the X values and overwrite them with Boolean initial values.

A range of primitive gates are provided: `AND`, `NAND`, `OR`, `NOR`, `XOR`, `NXOR`, `BUF`, `NOT`, `BUFIF0`, `BUFIF1`, `TRANIF0`, `TRANIF1`, `CLOCK0`, `CLOCK1`, and `BALR`. The `TRANIF0` and `TRANIF1` gates are bidirectional switches with two bidirectional ports and one control input [Flake et al. 1980]. The `BALR` (balanced line receiver) produces X when the inputs have the same value.

Wires can be declared as arrays with specified left and right bounds; `TRI` wires can also designate hold delays.[4] Gates and sub-units can also be arrays. Comments are indicated by `**` (like `//` in C++). For example, here is the same 4-bit shift register definition, and is illustrated in Fig. 6:

```
CCT RSH4(DATAOUT[3:0], DATAIN[3:0], LOAD, SERIALIN, CLOCK, CLEAR)
  MUX (4:6:8, 2:3:4) ** multiplexer with minimum, typical, maximum delays
    M[3:0] (S[3:0], DATAIN[3:0], LOAD, (SERIALIN, DATAOUT[3:1])); ** 4 instances
  DET (6:8:10, 4:6:8) ** D-type flip-flop with minimum, typical, maximum delays
    D[3:0] (DATAOUT[3:0], S[3:0], CLOCK, CLEAR); ** 4 instances
  UNID DATAOUT[3:0], S[3:0]; ** wire arrays with single drivers
  INPUT DATAIN[3:0], LOAD, SERIALIN, CLOCK, CLEAR.
```

[4] A *hold delay* is the amount of time a value is held by an undriven bus before it becomes Z.

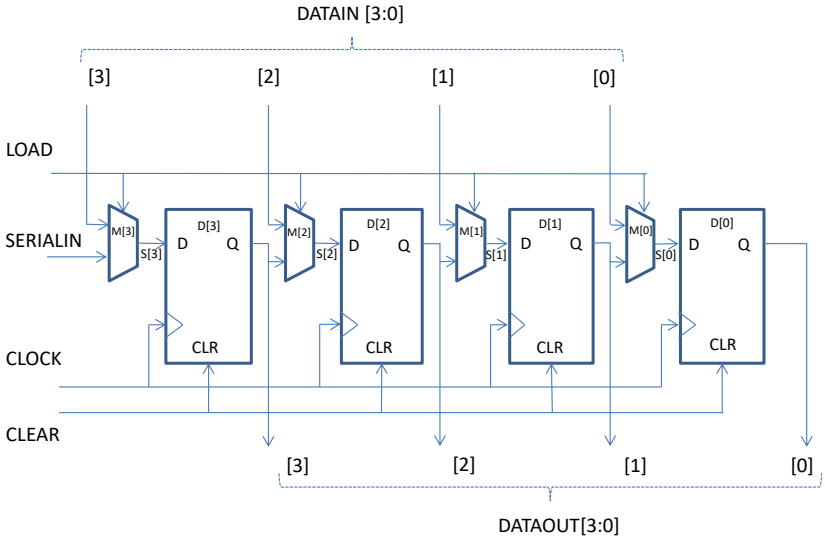


Fig. 6. 4-bit shift register example, using HILO 2 notation.

The RSHIFT operator in HILO 1, like the SPLIT and MERGE, was replaced by concatenation and slicing, to avoid creating new names for the same physical wire. All wires have to be declared. Delays can be specified as single numbers or as triples (minimum:typical:maximum). A gate can specify five distinct timing parameters: rise delay, fall delay, marginal rise delay,[5] marginal fall delay, and capacitance value. These parameters allow delays to be computed as a function of the load (that is, the capacitance). A NAND gate type can thus be written with these five parameters:

```
NAND ( 20:30:40, 30:45:60, 2:3:4, 3:4:5, 2 )
```

Delays can be parameterized and passed down the structural hierarchy.

The HILO 2 structural description syntax and keywords have survived in Verilog and SystemVerilog, with minor changes. For example, HILO 2 is not case sensitive, whereas the later languages are, and have lower case keywords.

3.2.2 Behavioral Description. Whereas HILO 1 had five built-in flip-flops, which reflected the components available at the time, the arrival of custom integrated circuits enabled the creation of many kinds of flip-flops. Therefore flip-flops are not built-in primitives in HILO 2; they must be described either *structurally* as a netlist of gates, or *functionally* using partial continuous assignments for level-sensitive behavior and event statements for edge-sensitive behavior:

```
CCT (DEL) DET (Q, D, CLOCK, CLEAR)  ** parameter DEL, type name DET, four ports
 UNID
   Q = R;                ** output
 INPUT1                  ** if unconnected, value defaults to 1 (i.e. pull up)
   D, CLOCK, CLEAR;
 REGISTER (DEL, DEL)     ** declare a register with delays as parameter
   R = 0 LOADIF0 CLEAR   ** level sensitive statement: set R to 0 if CLEAR is 0

 WHEN CLOCK (0 TO 1) DO R = D.    ** event statement: set R when clock rises
```

[5] *Marginal delay* means delay per unit capacitance.

Register declarations are similar to wire declarations, with two differences: first, the delays must be non-zero to guarantee deterministic results when simultaneous input changes occur. Also, if the declaration contains an expression, there must exist input combinations that do not return a value, so that the previous value is held. In the example, CLEAR = 0 gives R = 0, otherwise the previous value of R is maintained until the clock rises to sample the D input.

Random Access Memories (RAM) can be read and written only with event statements and do not have level-sensitive behavior. The RAM declaration can be used for small two-dimensional arrays:

```
RAM (0:15) INDEX[3:0] ** 16 4-bit registers named INDEX
```

Event statements can also be used for timing checks and can trigger a named event, which has no associated data value:

```
WHEN PHASE1 (1:X TO 0:X) DO        ** a falling value to or from unknown (X)
  IF PHASE1 = 0 & PHASE2 = 1 & STEADY (PHASE2, 150) ** no change for last 150
    THEN EVENT CK1 ELSE EVENT CKX ENDIF;  ** trigger either named event
```

The STEADY operator returns true if the wire has not changed or the event has not occurred within the specified time. The WIDTH operator returns true if the time between the last two changes or event occurrences is within the specified time.

An event expression can be complex:

```
WHEN 12*(CKA THEN CKB) THEN (EVA OR EVB) WAIT 5 RESET EVX DO ...
```

This means that the action is triggered after a change of CKA followed by a change of CKB occurs 12 times, and is followed by either event EVA or EVB occuring, and then waits for 5 time units to pass. The trigger is cancelled if event EVX occurs.

The reason for providing such complex event expressions was to allow a test generation algorithm that given the goal of achieving the action, could generate the necessary sequence. Such an algorithm was never implemented in HILO, and it would be many years before property checkers could handle such sequences in an HDL. Although a large repetition count was useful for expressing long reset sequences, models of microprocessors mainly used simple event expressions.[6]

3.2.3 Simulators. HILO 2 had two simulators: *fault-free simulator* was a logic simulator used to explore the design's behavior and for debugging, and *fault simulator* for simulating the design with faults injected in order to grade tests for use in functional board testing.

The simulators could run with a selection of either minimum, typical, or maximum delays being used. The fault simulator was slower and needed more memory. It used the parallel method: 31 different faults and the fault-free description were simulated in one run, with each wire value corresponding to a bit position in two words. The primitive gates and Boolean operators in the HDL were coded using bit-wise Boolean operators in the implementation source code.

The simulation algorithm had two phases: value changes at time ticks and event statement evaluation at half ticks. This was possible because registers could not have zero delays, and the result was a deterministic simulation, which is necessary for fault simulation.

Like HILO 1, the simulators had a Waveform Description Language (WDL) to specify both stimulus and response in testbenches, and for simulation control such as tracing and specifying fault lists.

3.2.4 Implementation and Exploitation. The implementation language chosen was BCPL [Moorby 2013] as this handles pointers and bit-wise operators efficiently, and the code targeted a 32-bit virtual memory machine, as (in the late 1970s) this was seen to be the future architecture. The

[6]Verilog supports only simple event expressions and named events, but SystemVerilog does have a syntax for complex sequences.

BCPL compilation system has an intermediate form called OCODE, which can be turned into a more efficient interpretable form called INTCODE (like bytecode). Because Brunel University did not have a 32-bit virtual memory machine in 1977, an INTCODE interpreter was written first on the ICL 1904 at the university computer center, then on the DEC PDP-11, which arrived in the electrical engineering department. The only medium in common between these machines was paper tape, but the ICL used spools and the DEC used fan-fold tape. So, to move the code from ICL to DEC needed one person to hold a pencil with the spool of tape in the fan-fold reader while another person operated the console!

Fortunately the ICL 1904 was replaced by a Honeywell Multics machine, which had a BCPL compiler. Phil Moorby has told the story of a particularly tricky compiler bug [Moorby 2013].[7] By mid-1979 Moorby had stopped his PhD work and joined the team full time as a programmer. The team was now six people (Fig. 7).

The first paper presenting HILO 2 was at CHDL '81 in Kaiserslautern, Germany [Flake et al. 1981]. In the audience was Prabhu Goel, author of the PODEM algorithm [Goel 1980] for test pattern generation, who had just left IBM to join Wang and create an EDA department [Goel 2017]. He was interested enough to visit Brunel University and start evaluating HILO 2, typing at a hard copy terminal and creating a behavioral model. At the same time, the Ministry of Defence and Smiths Industries agreed to hand over the HILO project to a small company, Cirrus Computers Ltd., which specialized in writing test programs for a GenRad digital tester but whose staff were familiar with logic simulation. Goel's group at Wang became the first customer, using DEC VAX computers running the VMS operating system (which also had a BCPL compiler with a bug that was hard to find: it generated bad code for conditional jumps of exactly 256 bytes).

HILO 2 was the first commercially available hardware description language that combined register transfer level with gate- and switch-level descriptions including accurate timing [Dettmer 2004].

GenRad acquired Cirrus Computers in 1983, and set up a new EDA unit in Santa Clara, California. The gate array ASIC market was starting to develop, and the Silicon Valley sales, marketing, and support operation had some success, reaching $10M per year at its peak [Harris et al. 1984a,b]. Part of this was in OEM sales to companies (Tektronix, Hewlett Packard, Calma, and Intergraph) that wanted to compete in the ASIC workstation market, where Daisy, Valid, and Mentor Graphics were leaders. Not all the machines had BCPL compilers, and so staff at Cirrus wrote a BCPL to C translator that needed only a relatively small amount of manual re-work on the translated code.

3.3 HILO 3

The desire to save memory in HILO 2 had been very strong, and the simulation data structure had a 32-bit word split into a 12-bit port number and a 20-bit pointer (BCPL uses word addressing). As the million-word limit was becoming a problem, the simulator was re-written in C with 16-bit port numbers and 32-bit pointers. The other parts of the software remained largely the same with the exception of the fault simulation implementation [Moorby 1983].

Part of the feedback from Silicon Valley customers was that the switch-level simulation using only four values was inadequate. When two gate outputs are connected together, one output may be stronger than the other. When the strong output is at Z (that is, not driving a value) then the weak output can drive a 1 or a 0 on to the wire. When the strong output is driving a 1 or a 0, the weak output has no effect. This can be modeled using more than four values: strong 0/1/X, weak

[7] The compiler used wall clock time to determine the compilation order of procedures. This made the compiler nondeterministic—it changed behavior each time the same code was compiled—and made it hard to debug a problem or assess a fix.

Fig. 7. HILO 2 team circa 1981. Left to right: Simon Davidmann, Peter Flake (Technical Authority), Phil Moorby, Gerry Musgrave (Managerial Authority), Robert Harris, Richard Wilson. Additional information about each can be found in People on Page 73. Photo provided by Simon Davidmann.

0/1/X, and Z. There could be several unknown values, and the resulting algebra was presented at DAC in 1983 [Flake et al. 1983].

This new simulator was called HILO 3 and was ported to a large number of Unix-based machines, including the ASIC workstations mentioned above. The HDL was renamed GHDL (GenRad HDL).

3.4 HITEST

As well as the ASIC design market, GenRad was interested in using HILO for its PCB testers in functional board test mode, and for its chip testers. This drove a new project at Cirrus Computers to re-implement HILO 3 as System HILO, with a new testbench language and interactive testbench generation system called HITEST [Bending 1984]. The result was a loss of focus on the ASIC design market when a competitive simulator, Verilog-XL, appeared.

4 VERILOG

4.1 Transition to Verilog

Prabhu Goel left Wang in July 1982 to start a new company called Automated Integrated Design Systems, soon renamed Gateway Design Automation. Goel saw a need in the market for an automated test generation program for circuits with scan paths. His plan was to use his consulting and lecturing income to fund product development. Working out of his house, by December 1983 Goel had sold the first two licenses for Gateway's TestScan software. With the proceeds in hand from this first successful product, Goel next wanted to develop the ultimate mixed-level simulator,

able to handle register transfer level as well as gate- and switch-level. This simulator would be aimed initially at the test market, but later support logic verification and logic synthesis [Bell 1991; Goel 2017].

Goel remembered Phil Moorby from his visit with the HILO team in 1981, and toward the end of 1983 he invited Moorby to leave the HILO team at Cirrus/GenRad and move to the US to work at Gateway. Moorby was tasked with developing a new language and simulator to support logic verification, fault simulation, timing analysis, and synthesis. This new language was named Verilog. Goel also hired Chi-lai Huang, who had recently completed his PhD work on logic synthesis using the LALSD II language [Huang 1981]. Huang worked with Moorby right from the start to ensure that Verilog would be synthesizable, that is, suitable for logic synthesis [Moorby 2013].

Whereas HILO had one language for hardware description and a different language for stimulus and simulator control, Verilog featured a single combined language with one syntax and one set of semantic rules. This made it easier to learn and write complex algorithmic procedures for specifying both the design and its verification code. Furthermore, this language extended HILO's declarative style to include a procedural style of programming and enabled a natural mix of the two styles. As hardware designs were getting larger and incorporating more complex algorithms, a need was growing for the ability to write those algorithms in a user- and debugger-friendly sequential procedure. Rather than requiring a user to write many hard-to-follow declarative expressions, a procedural programming language style was developed. It was important to get the semantics of mixing declarative and procedural parts right, so that signals (wires) from low-level declarative assignments (including gates and switches) could be read by the procedural statements, and the values calculated in the procedural parts could be read by the declarative parts. As is explained in later sections, the timing of the interactions between the different parts (conceptually separate processes of execution) was sometimes tricky to get right. That is why hardware designers must become skilled and experienced at writing correct synthesizable code, for the synthesis tool to check that the code meets certain design criteria, for the simulator to extensively verify that the logic is fully correct, and finally for the timing analyzer to verify that all the timing constraints are met.

4.2 Initial Development of the Verilog Language

The initial design of the Verilog hardware description language was created and completed over the winter of 1983–84. Its design was largely inspired from the work on HILO and the collaboration between Moorby and Huang to bring in the experience from the LALSD II language work. Its implementation, consisting of the parser, elaborated data structure builder, and a full executable simulator, was completed by early 1985.[8]

The primary motive for creating a new language was to support both synthesis and verification of hardware designs on an equal basis, with a secondary goal of supporting various other EDA tools, such as timing analysis and fault simulation. The knowledge to implement the simulator was largely well understood, but synthesis was a new technology yet to be proven in practice, and the industry was just starting to explore how effective it could be in the automation of hardware design. The promise for synthesis was to remove all the tedium of explicitly connecting up individual gates and transistors, and that an easy-to-use high-level language would automatically generate the low-level implementation details. Also important was the ability to automatically target different implementation technologies.

[8]The lack of citations in this and following sections is due to Verilog's origin as a proprietary language. Access to early language references is restricted by non-disclosure agreements. See Section 5.4.

A major language design decision, and a long-debated issue with HDLs in general, was whether to support a stacking mechanism to allow functions and tasks to be re-entrant and allow dynamically allocated memory for data during run time. It was well known that an HDL for synthesizing efficient hardware could not make use of an arbitrary large stack or dynamically allocated memory for data. Every run-time variable in a synthesizable hardware description corresponds to hardware and so has a static lifetime. This is in contrast to the normal design of a programming language. It was decided to forego this software feature and limit the memory design to that of a static model. This meant that all memory size requirements are known at elaboration time and fitted well with how hardware was generated efficiently through the process of synthesis. This is perhaps one of the central differences between languages meant for hardware design, versus convenient features of software language practices. In 1984 at the start of designing Verilog, it was decided to favor hardware design practices and plan to extend this later if necessary.

Although the initial version of the Verilog hardware description language was focused toward synthesis, the intention all along was to produce a compatible very fast and accurate gate- and switch-level simulator. At the time that Verilog was being designed and implemented, gate- and switch-level verification was growing in demand within the chip design industry.

In general, the actual design approach taken was to start at the bottom and build up. At the bottom were the basic variables and expressions, where the gate- and switch-level modeling of the scalar "bit" needs to be extended to a vector of bits for the RTL descriptions.

It became well understood from customer designs that at least four levels of strength were needed at the gate and switch level, leading to a large number of distinct scalar values. But at the RTL level, the emphasis was on using long data words to model wide registers in parallel for the purpose of simulation speed. So the compromise was to model scalars with a full set of logic strength values, and to limit the number of logic values for vector registers to four: 0, 1, X, and Z. The X represented the unknown value, and Z the high-impedance value (same as HILO, as described in Section 3.2.1). This enabled a relatively successful smooth interface between the high-level RTL representation, and the low-level gate and switch representations.

4.3 Concurrency, Declarative and Procedural Styles, and General Timing Controls

Verilog retains the declarative style of HILO. The circuit of Fig. 8 comprising one inverter, a 3-input OR gate, and a 4-input NAND gate, could be described using Verilog gate primitives as follows:

```
wire anot, or1, out2;    // these signals are "wire" type
not  U0 (anot, a);              // anot is the inverse of a
or   U1 (or1, anot, b, c);      // or1 is the OR of (anot,b,c)
nand U2 (out2, d, e, f, or1);   // out2 is the not-AND of (d,e,f,or1)
```

Signals `anot`, `or1`, and `out2` are declared as `wire` because they are *continuously driven* by the gate primitives.

Each gate primitive represents an independent simulation process. For a given gate, if any of its inputs change, then its output changes accordingly, and propagates throughout the design. These gates operate *concurrently*. The textual order of the gates doesn't matter; they could just as easily (and correctly) have been written as follows:

```
wire anot, or1, out2;
nand U2 (out2, d, e, f, or1);
or U1 (or1, anot, b, c);
not U0 (anot, a);
```

Verilog also supports the *continuous assignment* statement which allows the use of a rich collection of operators. Using these statements, our simple example could be written as follows:

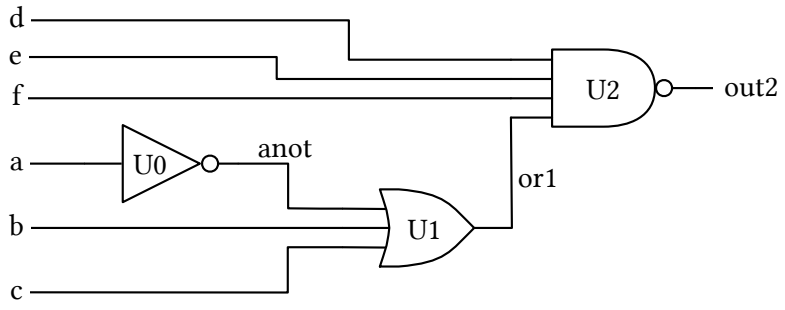


Fig. 8. Schematic of three gates.

```
wire anot, or1, out2;
assign out2 = ~(d & e & f & or1);   // out2 is the not-AND of (d,e,f,or1)
assign or1 = anot | b | c;          // or1 is the OR of (anot,b,c)
assign anot = ~a;                   // anot is the inverse of a
```

Again, the nets are declared as `wire` type because they are continuously driven by their respective `assign` statements. These statements execute concurrently; if a signal changes value, then any continuous assign statement using that signal as input will evaluate itself and drive its output accordingly. These statements could appear in any order.

In addition to these declarative styles, Verilog added a *procedural style* similar to more typical software programming languages. The keyword `always` indicates that the following statement (or `begin...end` block) should always be executed. Each `always` represents a separate and concurrent execution process. Using this procedural style, our example might look as follows:

```
reg anot, or1, out2;  // these signals are "reg" type
always begin
  anot = ~a;                 // this statement executes first
  or1 = anot | b | c;        // then this statement
  out2 = ~(d & e & f & or1); // and finally this one
end
```

The three statements within the `begin...end` block are executed *sequentially*, in a single thread,[9] so their order matters. Here the signals are declared as type `reg` because they are driven by *procedural assignment* statements. Rather than being updated continuously, these signals are updated only when their individual procedural statements are executed.

When process control gets to the `end`, control returns back to the `always` and the block immediately executes again. This is an infinite loop! and will hang the simulator. We need to add a *timing control* to our `always` block:

```
reg anot, or1, out2;
always @ (a or b or c or d or e or f) begin
  anot = ~a;                 // updating anot
  or1 = anot | b | c;        // updating or1
  out2 = ~(d & e & f & or1); // updating out2
end
```

[9]In Verilog, the terms *process* and *thread* are used interchangeably. Refer to the Glossary on Page 75.

The timing control `@()` means "wait here until the event inside the `()` occurs." We read this "always at a or b or c or d or e or f, then do this block." Now when execution reaches the `end`, this process returns to the `always`, encounters the `@()` timing control, and halts. When one of the signals listed in the timing control changes value, then the `begin...end` block is executed again.

Timing controls can specify a particular edge of a signal. Here is a flip-flop that changes state on the rising edge of signal `clock`:

```
always @ (posedge clock)
  q = d;
```

In contrast to `always` which repeats forever, Verilog has the `initial` block which executes only once, at the start of simulation time. This is useful for testbenches. Refer back to the testbench on Page 5 for an example.

Another timing control is the explicit delay operator `#<value>`. Here is an example:

```
reg clock;
initial
   clock = 0;
always #5
   clock = ~clock;
```

This is a common way to create a clock signal with half-period of 5 time units. Note that Verilog's procedural style allows updating a signal in more than one block/process.[10]

This delay syntax can be used to add propagation delay to gate primitives and continuous assignments:

```
not #4 U0 (anot, a);  // anot changes 4 time units after a
assign #6 or1 = anot | b | c; // or1 changes 6 time units after inputs
```

Our design can be written with a mix of gate primitive declarations, continuous assignments, and procedural blocks, all running concurrently and independently:

```
wire anot, or1;
reg out2, out2not;

always @ (or1 or d or e or f) begin
  out2not = d & e & f & or1;
  out2 = ~out2not;
end

assign #2 or1 = anot | b | c;

not #4 U0 (anot, a);
```

From the very beginning of the Verilog language design, a generalized mixing of sequential and concurrent processes was supported. In all threads of execution, timing controls could be added to time the start of any statement. As is usual for HDLs, Verilog used integer time, which improved simulation performance and made simultaneity unambiguous.

4.4 Expression Bit Widths

One of the first considerations in the design of Verilog HDL, and critical for synthesis to be able to create optimized hardware, was the rule governing vector widths for operands and operators

[10] Although this is not supported by synthesis. See Section 5.2.

in expressions. In hardware design, both the number and the timing of the gates and transistors generated by the synthesis tool are critical in achieving successful designs.

The rules governing the simulation semantics at the RTL level must be identical to what is assumed by the synthesis tool when translating the RTL design into gates and transistors. A goal of the translation is to achieve the same simulation results before and after, and that these results are what the designer intends. In our introductory 4-bit shift register example, all the vector operations are clean 4-bits wide. Arithmetic operations in most hardware designs, on the other hand, have implicit overflow conditions that frequently need to be captured. A common example is the carry bit when two vectors are added together. Consider this assignment:

```
sum[3:0] = a[3:0] + b[3:0];
```

The potential carry overflow is lost. This is acceptable if the designer either knows there will never be an overflow, or is a don't care situation. If the design intent is to capture the overflow bit then in Verilog the user can simply extend the LHS to receive it:

```
sum[4:0] = a[3:0] + b[3:0];
```

Here `sum[4]` receives the carry bit. For simulation semantics, the two RHS operands are first zero-extended to 5-bits, and then the addition operator is performed as a 5-bit add where it is known by the rules of addition that there will be no overflow and all relevant bits can be captured into the LHS. Although the language semantics defines this as a 5-bit addition of two zero-extended operands, the synthesis tool is free to use any implementation circuitry that gives the same results, for example, a 4-bit adder with a special carry-bit output could be used. The goal of the language syntax is to be concise and that the design intent should be obvious.

The concatenation operator `{}` joins together bits resulting from two or more expressions. This provides another method for performing the example addition:

```
wire carry;            // declare carry as a single bit
wire [3:0] a, b, sum;  // declare a,b,sum as 4-bit buses
{carry,sum} = a[3:0] + b[3:0]; // add two 4-bit values to produce 5-bit result
```

Concatenations may be arbitrarily nested, and may have a repetition multiplier:

```
{4{w}}  // This is equivalent to {w, w, w, w}
```

Concatenation can be used on both the left-hand and right-hand sides of an expression. This provides a powerful mechanism for bit-level manipulaton of signals and buses.

Note in the final line above there is no explicit part select on `sum`, thus it defaults to its declared width of four bits. The assignment behaves the same as if we had written the following:

```
{carry,sum[3:0]} = a[3:0] + b[3:0];
```

Now consider this assignment and these declarations:

```
reg[16:0] accum;
reg[7:0] a, b, c;
accum[7:0] = (a + b) * c;
```

The intent here is to do the addition and multiply operators using 8-bit arithmetic. However, an addition of two 8-bit values has the potential of generating nine bits, and a multiply of this 9-bit value with an 8-bit value has the potential of generating 17 bits of significant data. Sometimes the design intent is to capture all this information. This can easily be expressed as follows:

```
accum = (a + b) * c;
```

This captures all the potential overflow into the LHS most significant bits. Note that `accum` is declared to be 17 bits wide.

In general, the number of vector bits used within expression operators is intuitive and follows design intent depending on the context where the expression is given, and simulation results will reflect the semantics that the synthesizer assumes. The number of vector bits of the LHS affects most, but not necessarily all, of the number of bits of the RHS operators and operands. For operators, such as arithmetic and bit-wise logical, a natural promotion to the maximum size is used. Some operators have operands that are considered to be self-determined, such as the conditional expression in the ternary conditional operator, and the concatenation operator.

The goal was to achieve consistent semantics across all tools and a natural extension of the rules to cover all operators and operands.

4.5 Combinational Logic with Continuous Assignment

Combinational logic is where there is no stored state from one clock cycle to the next, and thus the values of the outputs are a pure function of the values of the inputs. A designer can express any amount of combinational logic with a series of declarative-styled continuous assignments that form an acyclic graph structure from inputs to outputs. The expressions making up both sides of the assignment can be any arbitrary expression, so long as the acyclic graph structure is maintained. Value changes that arrive on the inputs during simulation must be propagated through the assignments structure in such a manner as to always maintain the correct values of the outputs. The RHS expression must be calculated without any time advancement, but delays could be specified to model the propagation delays from when the RHS changes value to when the LHS would be updated.

There are various ways a simulator can implement this, but the usual method is to use selective trace (event-directed) techniques where only the nodes in the graph structure that change value are re-evaluated. Sometimes, it may be faster to recalculate the whole acyclic graph structure for each clock cycle, and this largely depends on the structure and rate at which the inputs change. The particular technique employed was left for the simulator to decide the best approach; it only had to produce the correct output values at the correct simulation time. Synthesis tools assumed the same semantics, however during synthesis the focus was on optimizing the combinational Boolean logic formed by the acyclic graph of assignments and producing what the designer would consider the most optimized implementation.

Various operators were provided in the language: arithmetic, bitwise, logical, and these followed the C language. But there was a need for some extra operators not found in most programming languages, for example concatenation and reduction operators.

Also, some extra operators were introduced to deal specifically with the four-valued logic, such as the identity operator '===' where a useful simulation-specific test for X and Z values could be deterministically expressed, but which a logic synthesizer would ignore.

4.6 Functions

Another important construct that can be used within the RHS of continuous assignments is the `function` call. Functions are meant to model zero-delay combinational logic, meaning that they must not contain any timing controls and must return a value without involving any simulation time increments.

For example, we could define a function as follows:

```
function [7:0] myAdder(a,b);
   input [7:0] a, b;
   myAdder = a + b;
endfunction
```

Then call the function:

```
accum = myAdder(a,b) * c;
```

The idea of a `task` (see Section 4.10), in contrast to a `function`, is that a `task` can have embedded timing controls (plus a few other differences such as the way values are passed back) that would make tasks very difficult to call inside expressions. All expressions must evaluate without any simulation time increments or side effects (that is, a `function` call that changes some state external to the `function`). This requirement actually made for an awkward situation in functions—what to do about the need to display messages and values in the body of a `function` for debugging purposes in the simulator. Ultimately, `$display` statements (see Section 4.13) were allowed in the simulator and ignored in the synthesis tools; the rationale was that displaying information does not change the state of the simulated design, hence, it is not considered a side effect.

Another requirement for functions (and tasks in this case) is that they cannot be called recursively because the actual hardware has no arbitrary-sized stack to hold the calling control mechanism and data information. This comes from the static nature of Verilog, and the requirement from synthesis that function calls must be inlined and expanded to generate optimal hardware.

4.7 Sequential Logic

When considering sequential logic, the focus is on how to specify the clocked next state equations. An initial consideration was whether to explicitly declare a clock signal. An HDL that allows the mixing of low-level gate and transistors with high-level clocked RTL assignments has no requirement to declare the clock signal explicitly. Consider this edge-sensitive flip-flop code:

```
always @(posedge clock) q = data;
```

The clock signal is inferred implicitly by the synthesis tool. This statement uses a single thread of control that is started at the beginning of simulation, and where the keyword `always` indicates that the thread never ends. The `@` symbol indicates that for each time around the `always` loop, control will halt and wait for a positive edge of the clock.

There is a problem when multiple such statements are combined, and state variables are shared:

```
always @(posedge clock) r0 = data;
always @(posedge clock) r1 = r0;
```

This has a race condition because the order of the two assignments is undefined.

In hardware design there are two race-free kinds of register constructs that can implement the next-state assignments of classical state machine theory: master-slave and edge-triggered [Bartee 1960, Sec. 4-25]. To model these two constructs in Verilog the following were first designed into the language:

```
always @(posedge clock)
   rMS = @(negedge clock) a+b; // master-slave form

always @(posedge clock)
   rET = #1 a+b; // edge-triggered form
```

The master-slave form evaluates the `a+b` expression when the positive edge of the clock signal occurs, and stores the result in a hidden register. Then, when the negative edge of the clock occurs, the stored hidden register is assigned to the LHS.

The edge-triggered form again evaluates and stores into a hidden register the RHS at the positive clock edge, but then waits for one simulation time unit to elapse (called a unit delay) before advancing to assign the hidden register value to the LHS. All edge-triggered flip-flops have what

are called *essential hazards* and the explicit non-zero delay provides the necessary determinism in the simulation semantics.

Now, we can share registers in the following way:

```
always @(posedge clock) r0 = #1 data;
always @(posedge clock) r1 = #1 r0;
```

This has deterministic semantics, and similarly for the master-slave form.

We might attempt to combine these statements into a single `always` block:

```
always @(posedge clock) begin
   r0 = #1 data;
   r1 = #1 r0;
end
```

But this would not work because these `#1` constructs are blocking, and the second assignment picks up the new value of `r0`, rather than the old value as expected in state machine descriptions.

One answer is this more verbose coding style:

```
reg r0Tmp, r1Tmp;
wire #1 r0=r0Tmp, r1=r1Tmp;
always @(posedge clock) begin
   r0Tmp = data;
   r1Tmp = r0;
end
```

The assignment to `r1Tmp` gets the (correct) old value of `r0`, and the unit-delayed signal update has been moved to a pair of continuous assignments.

Section 5.1 introduces another method for modeling sequential logic.

4.8 Asynchronous Set and Reset of Flip-Flops

A typical mechanism in flip-flop devices is the use of asynchronous set and reset inputs. These are signals that would force the device to a known state regardless of whether the device is being clocked.

```
always @(posedge clock or negedge reset)
  if (!reset)
     assign q = 0; // executes when reset asserts (goes low)
  else begin
     deassign q; // executes on posedge clock when reset is negated (high)
     q = dataIn;
  end
```

The `assign` statement acts as a temporary continuous assignment, and is called a *quasi-continuous assignment*. The output value of such assignments are held until the `deassign` is applied. This style correctly models a flip-flop with active-low asynchronous reset.

Section 5.2 discusses another style for modeling flops with asynchronous set and reset.

4.9 Named Events and the "fork...join"

The `fork...join` construct provides a structured approach to parallel procedures. This idea actually was inspired by the `SEQ` and `PAR` constructs in the occam programming language [Moorby 2013], where the `SEQ` construct specifies a block of statements which execute in the normal sequential manner, while the `PAR` construct specifies a block of statements which execute simultaneously in parallel. In both cases, execution continues past the block once all statements inside the block

have completed. The main characteristic of these constructs is that they conserve the structure of a procedure such that every block of code is a 1-in and 1-out structure of flow control.

Consider this example code:

```
event ev1, ev2;    // declare two named events

always @(negedge c1) #10 ->ev1;  // event ev1 triggers #10 after c1 negedge
always @(negedge c2) #12 ->ev2;  // event ev2 triggers #12 after c2 negedge

always @(posedge clock) begin
   #5 r = accum;
   fork
      @ev1 p1 = a & b;  // at event ev1, do the assign to p1
      @ev2 p2 = a | b;  // at event ev2, do the assign to p2
   join
end
```

Named events, which have the same semantics as in HILO (see Section 3.2.2), can be declared, such as `ev1` and `ev2`, and used in timing controls. This `fork...join` block creates two threads of control that each wait for their respective events to occur before joining and continuing back to wait for another `posedge clock` event.

The first two `always` statements are controlled by the negative edge of two different clock signals, and if they do occur and the `fork...join` is in a primed state, will trigger the two forked statements at their respective times.

4.10 Tasks and the "disable" Statement

Both functions and tasks were designed into Verilog from the start. As we have seen, functions were aimed specifically to model combinational logic. Tasks, on the other hand, are more general and can contain timing controls. For example let us suppose that the following code was added to the previous example in Section 4.9:

```
task t;
 input[15:0] i0,i1;
 output[15:0] o0,o1;
 begin
   @ev1 o0 = i0 & i1;
   @ev2 o1 = i0 | i1;
 end
endtask
```

This task `t` can then be called in a statement:

```
t(r1,r2, s1,s2);
```

As with functions, there is no implicit arbitrarily sized stack to support recursive task calling—all variables were statically defined at elaboration time. Although the simulator allows multiple calls to the same task from different places, this has limited use and the coder has to be careful to manipulate the static variables.[11]

To prematurely kill activity using a structured approach, the `disable` statement was introduced. The idea was that from any place and time, any block of code could be disabled using a single statement. Both `begin...end` and `fork...join` blocks could be given a name, and this is used to

[11] Later revisions of Verilog introduced automatic tasks and functions that allow recursive calls. See Section 5.5.

do the disabling. Tasks and functions could also be disabled, which allows for an early return, or a continue, or break from a loop. For example, in our task above we could write the following on the outside of this task body:

```
always @reset disable t;
```

If the asynchronous `reset` event occurred, and the task was waiting for `ev1` or `ev2`, then the task activity would be killed.

4.11 The Challenge of Nondeterminism

Verilog is used to describe digital hardware made from assemblies of logic components that implement Boolean logic functions. Classical Boolean logic whose variables represent one of only two truth values, True (1) and False (0), is insufficient to model all of the hardware's behaviors and effects. As mentioned in Section 3.2.1, Boolean logic is inadequate to describe three-state logic or uninitialized values. Then again, the limitations of Boolean logic are not specific to hardware descriptions; mathematicians and logicians have proposed multi-valued logic alternatives to address the limits of bivalent logic. The first multi-valued logic formulation is the three-valued logic of Łukasiewicz [1920], which was motivated by philosophical concerns regarding the certainty that can be ascertained about future events. Since statements about the future cannot be assigned a true or false value, Łukasiewicz suggested the addition of a third value $\frac{1}{2}$ to the set {0,1} to represent "possible." In the mid-1970s Belnap [1977] proposed a four-valued logic to enable reasoning under incomplete and inconsistent information. This logic was aimed at an information processor capable of answering queries on propositions computed from an accumulation of atomic propositions. Belnap proposed a logic with *four* truth values: the classical ones, denoted by **T** and **F**, and two new ones, None and Both, denoted by **N** and **B**. The value N denotes lack of information whereas the value B indicates an inconsistency—a statement designated as both true and false. Although motivated by widely different concerns, Verilog's and Belnap's four-valued logic share remarkable similarities. In Verilog, the value Z denotes *no drive* which is consistent with the N truth value. Likewise, Verilog uses the value X to denote bus contention when driven with contradicting Boolean values, which is consistent with the B value (that is, both 0 and 1). Verilog's four-valued logic for simulating hardware is common but not universal (see Section 10.1).

Another significant influence in using the X value in logic simulation comes from the work at IBM in the 1960s, and in particular Eichelberger [1965]. In this work, timing hazards in digital gate-level circuits are detected using a three-valued logic, while simulating for the possible occurrence of spikes and hazards, when the timing of dynamically changing signals is uncertain. Eichelberger proved that if an output is computed to have an X-value (the third state), then there exists some distribution of delays on the inputs, logic gates, and their interconnections that could in theory produce a spike on that output. In practice, designers generally find this overly conservative. Nonetheless, the approach was incorporated into the tools that used the very successful level-sensitive scan design (LSSD) system developed at IBM [Eichelberger and Williams 1977] as well as in the design of the IBM System/38 computer in the 1970s [Berglund 1979].

Generally, the X value is used by Verilog to model the nondeterminism that arises when the Boolean value of a signal (or wire) cannot be determined. Nondeterministic values arise in hardware due to several effects: A bus contention, as mentioned above, leads to an unresolved value. But, far more common is the nondeterminism due to uninitialized hardware registers and memories. Unlike software, which provides various mechanisms to initialize any memory with a deterministic value, uninitialized hardware registers have no such mechanism. When hardware is powered-up, these uninitialized registers are set to arbitrary values; some bits are set to 0 while others are set to 1.[12]

[12]In rare situations some may even be set to a metastable value that lies in between 0 and 1 [Kleeman and Cantoni 1987].

Uninitialized registers at power-up[13] are by far the most common source of X values during simulation, but they are not the only cause. Verilog produces an X value whenever a result is unknown or indeterminate as well as when error conditions occur. Specifically, an X can arise due to the following:

(a) uninitialized values
(b) bus contention
(c) unconnected inputs
(d) reading an array using out-of-bounds indexes
(e) reading an array using an X index
(f) timing violations
(g) arithmetic exceptions such as divide-by-zero
(h) explicit assignment by a user model or testbench

It is noteworthy that most of these (a)–(e) are due to nondeterminism, thus offering little choice in the design of the language. However when a simulation exception or violation occurs, producing an X result (instead of halting the simulation) was a choice and part of the language philosophy. An answer is preferable to a program termination, because the hardware has no such mechanism.

Using X values to denote nondeterministic results (uninitialized or unknown) is a very valuable simulation abstraction that allows engineers to run a single simulation of a reset sequence to verify that all variables are properly initialized with valid Boolean values. In contrast, if only Boolean values were available, it would require an exponential number of runs to make the same exhaustive determination: 2^N runs for a hardware block with N bits of storage. In this sense Verilog's X value acts very much as Belnap's B value—it is the projection of both true and false values. For this simulation scheme to be effective, the X values must propagate correctly through the hardware model. This *X-propagation* process does present its own set of challenges and problems.

The simulation semantics that treats X as a distinct logic value results in two X-propagation effects [Turpin 2003]:

- **X-Pessimism**: An X value propagates *more* Xs than are viable in the hardware.
- **X-Optimism**: An X value propagates *fewer* Xs than are possible in the hardware.

Verilog has a variety of constructs that model X-propagation differently. Consider a 2-to-1 multiplexer—a very common device used to select one of two signals depending on the value of a *selector* signal. Such a multiplexer can be written in Verilog in several ways. Here are three examples: first, using a procedural `if-else` statement, then using the `?:` conditional or ternary operator (inspired by the similar C construct), and finally using gate primitives:

```
if ( sel )
   v = a;
else
   v = b;
```

```
v = sel ? a : b;
```

```
not U0 (selnot, sel);
and U1 (net1, a, sel);
and U2 (net2, b, selnot);
or  U3 (v, net1, net2);
```

Figure 9 shows a schematic corresponding to the gate primitive example. (This schematic is also representative of a gate-level implementation generated by synthesis, using either of the first two code examples as source input to the synthesizer.)

[13] Sub-systems may be turned on and off during system operation, not only at start-up time. The term power-up covers both.

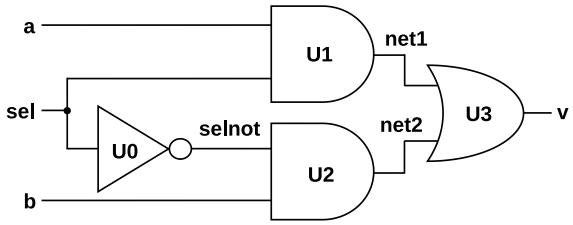


Fig. 9. Gate-level implementation of a 2-to-1 multiplexer.

For all three examples, when `sel=1` then `v=a`. Likewise when `sel=0` then `v=b`. However when `sel=X` the results differ:

sel	a	b	if else	?:	gates
X	0	0	0	0	0
X	0	1	1	X	X
X	1	0	0	X	X
X	1	1	1	1	X

X-Optimism is caused by the simulation semantics associated with the language constructs that must *choose* between two or more alternatives. The simplest such construct is the conditional `if` statement: an X condition is considered as not-true (that is, false), which is inconsistent with any realizable hardware. X is a simulation abstraction; hence, actual hardware cannot detect an indeterminate value and treat it as not-true. Optimism is caused by the conversion of a nondeterministic value into a deterministic one.

In the `if-else` example above, if the select signal `sel` is X during simulation then the `else` (not-true) branch will be taken and assign `v=b`. The nondeterministic value thus fails to propagate and instead produces a deterministic result. In the actual hardware, either branch could be evaluated and propagate either `a` or `b`.

This optimistic behavior of `if-else` was a deliberate decision by Moorby. He realized that in such a procedural context, evaluating both sides of the `if-else` expression could be enormously complicated. Reducing optimism requires execution time in the simulator. More importantly, implementing a complex feature like this would increase development risk and time. Instead, Moorby focused on keeping the intial implementation of the language and simulator both simple and clean.

Other Verilog constructs that exhibit X-optimism are the edge event controls `posedge` and `negedge`. Transitions involving X may be no transition at all, yet Verilog considers them as *true* transitions. For example, a 0→X transition is a `posedge` and triggers the execution of a flip-flop process (as described in Section 4.7) causing it to optimistically store new data, which is incorrect if the X represents a 0.

The practical impact of X-optimism is that it may cause tests to falsely pass, thereby concealing functional bugs. From a risk perspective, the severity of X-optimism is much higher than that of X-pessimism.

X-Pessimism is almost always caused by the reconvergence of correlated signals into one gate. Referring to the schematic of Fig. 9, from a purely Boolean perspective when both inputs `a` and `b` are 1, the value of `v` is 1 regardless of whether `sel` is 0 or 1. If X were consistent with Belnap's B, we would expect the output to be 1 when `sel` is X. However, that is not the case; the simulation assigns X to the output `v`. This X-pessimism is due to the straightforward implementation of the

simulation scheme: Each Boolean gate is considered in isolation thus failing to account for the Boolean correlation of the two signals `sel` and `selnot`.

Verilog's gate primitives stayed consistent with the behavior defined by HILO. Thus inverter `U0` has input `sel=X` and drives its output to the same value `selnot=X`, as defined by the NOT Boolean operation shown in the table on Page 12.

The practical impact of X-pessimism is that the additional Xs often cause a simulation to falsely fail, which requires time-consuming manual analysis and intervention such as temporarily injecting a 1 or 0 to overcome the pessimistic X. Because of the extra effort, many consider X-pessimism as undesirable, yet others regard it as more conservative and thus safer. In fact, there are situations that may warrant such X-pessimism: *timing hazards* and *metastability* are the most notorious. A signal in a metastable state is a physical manifestation of Belnap's B because in the hardware, a metastable value may be considered True by some gates and False by others. This real-world X thus violates the Boolean invariants built into the design and may lead to a failure (in other words, a system crash).[14]

Finally, the conditional operator `?:` is neither as pessimistic nor as optimistic as the other two. This operator considers both expressions when the condition is X, and produces an X only if the expression's bits are contradictory. This is straightforward to implement; the simulator evaluates the two expressions and drives the output appropriately.

If `a`, `b`, `v` are multi-bit signals, then consider this example:

```
v[7:0] = sel ? a[7:0]
             : b[7:0];
```

The gate-level representation will have eight multiplexers like the one in Fig. 9. For the same `sel=X` condition as before, the bits at each index are considered separately. For example, if `a[4]=b[4]=1` then `v[4]` will be 1. This is independent of any of the other `a` or `b` bit values. The bit-wise Boolean operators (`~`, `&`, `|`, `^`) behave in the same fashion; they consider each bit independently.

In contrast, the arithmetic operators return an all-X result if any operand bit value is the unknown value X or the high-impedance value Z.

Different constructs thus provide varying degrees of X-optimism and X-pessimism. We might be tempted to say that `?:` is *more accurate* than the other two examples, but that depends entirely on the tool doing the analysis, and the needs of the designer! For example, if `sel=X` indicates a timing hazard, then even when `a=b=1` the output of `?:` should drive `v=X` such that the downstream logic will see an indication of the hazard.[15]

Existence of an X value as an inherent part of the language allowed its repurposing (by both users and tools other than simulation) for uses other than those for which it was originally intended. In particular, synthesis tools do not need to model nondeterminism since their only concern is to produce optimal hardware that correctly implements the functionality described by the RTL. Synthesis interprets an explicit X value as a *don't care* and thus an opportunity to optimize the hardware. Here are two functionally equivalent Verilog examples of an explicit X assignment in a hardware model that can be used by synthesis to optimize the resulting hardware:

[14] Metastability and hazards are not the only reasons X pessimism may be useful. Phil Moorby recalls some ASIC vendors demanded multi-drivers of three-state buses to always result in an X value, even when driving the same logic value—the reason given was "the drivers might have different electrical properties." Although the concerns of driving the same logic value with different electrical properties (for example, voltages) were very real, they were never incorporated into Verilog.

[15] Moorby says that with the benefit of hindsight, this was probably a mistake in the language, and `?:` should have been designed with more pessimism: if the condition evaluates to X (condition expressions always evaluate to a single bit) then the result should be all Xs. Interestingly, this is still not the same as the gate circuit.

```
if( !a )                          case( {a, b} )
   if( b ) out = v + w;              2'b00:   out = v * w;
   else    out = v * w;              2'b01:   out = v + w;
else                                 default: out = 'X;  // don't care
   out = 'X;  // don't care       endcase
```

The X assignment in the two examples above is conditioned on the value of a being true. Hence, when a is false the outcome is determined based on the value of b: addition for b=1 and multiplication for b=0. But when a is false, the synthesis tool is free to choose any behavior, and in this case is likely to remove a from the circuit altogether and consider only b.

Explicit X assignments by a user model are also sometimes used to signal exceptional conditions. For example, the function below implements a simple decrement-by-one, but it also guards against a negative result by returning an X instead of a negative value.

```
function integer decr( integer data )
  if( data == 0 )
     return 'X;      // this is an exception, not a don't care
  else
     return (data - 1);
endfunction
```

One problem is due to the fact that an X represents different concepts to different tools: *unknown* to simulation and *don't care* to synthesis. Synthesis tools that consider Xs as don't care use them for opportunistic optimizations, but they may contradict the simulation behavior.

4.12 Debugging a Design

Language features for verifying that a design is correct, and if not, for debugging it to find out what is wrong, were built into the same Verilog syntax as the rest of the language. All parts had the same semantics. One important construct to have in the language was that of hierarchical names. This provided the ability to "reach" from one section of the module hierarchy over to another section by specifying a global path name uniquely identifying a specific instance in the module hierarchy, and a variable contained within that instance. This meant that the design could essentially be left untouched while important debugging code could be added without having to add extra ports to the module instances all along the path. Including hierarchical names in the testbench and design provided much more flexibility than HILO, where they were limited to a control/monitoring language.

For example, let us modify the testbench design from the example on Page 5 and create two instances of the shift register design, with appropriate connections between them:

```
module top();
   logic clock, clear, load, serialIn;
   logic [3:0] dataIn, dataOut0, dataOut1;
   parameter halfPeriod = 5;
   // instantiate two shift registers, with instance names dut0 and dut1
   shiftRegister4 dut0 (clock,clear,load,serialIn,   dataIn,  dataOut0),
                  dut1 (clock,clear,load,dataOut0[0],dataOut0,dataOut1);
   always #halfPeriod clock = ~clock;
   initial begin
      clock = 0;
      ...
```

Let us assume we narrowed down some verification problem we were having in the design where we wanted to reach into the first shift register instance `dut0` and obtain some information on the variable `q` through time. We could specify a statement as follows:

```
always @(top.dut0.q)  // hierarchical reference to q inside dut0
   $display("At time %d, top.dut0.q changes to %b", $time, top.dut0.q);
```

This would print out a message each time `q`, in the first shift register, changes value.

It turns out that all the variable references in this statement happen to be globally defined, so it could be written in any of the Verilog modules. Module `top` is not instantiated, so its module name and instance names are the same, and have global scope. System function `$time` acts like a built-in global variable. In the Verilog-XL simulator (see Section 4.15) this statement (strictly speaking a similar statement where the keyword `forever` would be written instead of `always`) could be typed in as an interactive command, or written in some separate module that could be added on the "side" of the usual module hierarchy.

Another debugging feature in the Verilog language was the `force` and `release` pair of statements. These actually acted in a similar way to the procedural `assign/deassign` pair, but had an overriding ability to force a variable to some value:

```
initial begin
  repeat (10) @(posedge clock) force   top.dut0.load = 1;
  repeat (5)  @(posedge clock) release top.dut0.load;
end
```

Here we wait for the first 10 cycles of the clock, force the variable `load` inside instance `dut0`, and then release this force 5 cycles later.

Although `force` and `release` are debugging constructs, they can create race conditions within the design if they interfere with the normal clocking actions in the design on the positive edge of the clock. So a better method that is race-free might be to instead use the negative edge of the clock:

```
initial begin
  repeat (10) @(negedge clock) force   top.dut0.load = 1;
  repeat (5)  @(negedge clock) release top.dut0.load;
end
```

Debugging features such as force and release, and hierarchical netnames, were generally not supported by synthesis. See Section 5.2.

4.13 Verilog's Programming Language Interface

Verilog's Programming Language Interface (PLI) came about from the frustration of keeping up with user demands for enhancements. From previous experiences of customers wanting to access and control what was going on during simulation, a C interface was designed that first allowed certain restricted access to the internals of the simulator data structures. The main purpose for doing this was for various kinds of verification activity. One could argue that the basic method of printing out simulation values, `$display`, was a form of PLI introspection. So in this sense, the `$display` system task and the `$time` system function were the first manifestation of the PLI. Building on this, other PLI tasks and functions were defined that gave further access and led to what is called today *reflection* in computer science, that is, the ability to examine, introspect, and modify a program's own structure and behavior at run time.

An interface was provided to the C programming language that enabled users to link in separately developed C-written libraries modelling other parts of the system being simulated. The C library

would be called at specific places and times during simulation, and the C code could read simulation values and write values back into the simulation, which the simulator would then propagate through the normal signal fanout structures that were built and controlled by the simulator.

As Verilog evolved, the PLI was re-designed and expanded to become a fully defined system of introspection and reflection of the complete design. This allowed a user to have full control over all aspects of the language and its tools, without disruption or changes to the design—an essential feature for the testbench verification environment.

4.14 Origin of the Verilog Name

During initial development, Moorby referred to the project by the acronym EST, "Expression of a System of Tasks." In early 1985 Gateway had the opportunity to get their product description into Sun Microsystem's Catalyst Program, and Prabhu Goel asked Moorby to come up with a name for the language and simulator [Moorby 2013].

Moorby began by making a list of words that meant something to do with the product (Fig. 10). His plan was to pick two words and combine them into something unique. "Verification" and "logic" were two strong contenders, and this led him quickly to the name Verilog (Fig. 11).

Figure 12 shows the product description from the Catalyst catalog.

4.15 Verilog-XL

Conventional logic simulators of the time used multiple passes with separate linking and loading steps. In contrast, Moorby's Verilog simulator featured a novel single-pass interpreted operation with extremely fast lexical analysis and compilation. This gave Verilog a significant speed advantage over its competitors, even before the actual simulation began. Having the language front-end processing taken care of, allowed Moorby to concentrate his development efforts on optimizing the simulator kernel itself.

The initial product was overall quite speedy, but soon Moorby had written an even faster simulator. With careful coding, and judicious use of assembler on the Sun 2 workstation, the core of the simulation algorithm was whittled down to some fifty instructions per gate-level event. An event here was the code that performed a value change on a wire or register, followed by scanning and evaluating its fanout and updating the connected gate, and scheduling further events if necessary (there were an average of 2 to 3 gates on the fanout of each wire or register). For gate-level designs, this produced overall execution times within a factor of 3 of dedicated (and expensive) hardware accelerators such as Zycad. This new simulator product, released in 1987 and now named Verilog-XL, became a huge success for Gateway and, as we shall see, a key part of the ASIC revolution in chip design [Moorby 2013].

5 RISE OF THE ASIC MARKET AND ITS TOOLS

By the late 1970s MOS transistors had taken over from bipolar as the technology of choice for very large-scale integrated circuits (VLSI). The performance of these early IC designs was dominated by gate delays, as the net delays between gates were relatively small. However as transistor geometries continued to shrink, by the mid 1980s there were 10,000-gate chips being designed with significant net delays between gates. This drove the need for functional and timing-accurate simulation.

Major semiconductor companies typically built their own EDA tools—the main tools being gate-level simulator and place-and-route (P&R)—all using proprietary formats, thus keeping data internal. Most of their product revenues came from standard products/chips which were usually designed in-house, thus internal tools and proprietary formats were an acceptable solution.

Requirements changed from standard products to more application-specific ICs (ASICs) and ASIC became a new direction [Smith 1993]. Semiconductor companies formed new ASIC business

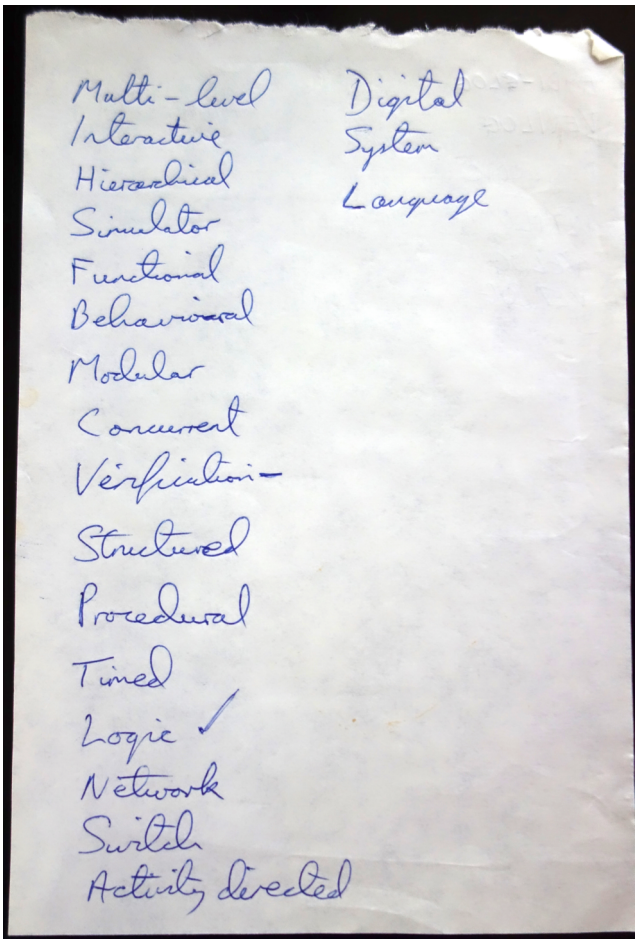


Fig. 10. In early 1985, Phil Moorby was asked to come up with a name for his new product. He generated this list of words. Moorby later recalled, "My basic idea was well, why don't I think of a whole bunch of words and put two of them together as a way of creating something unique. So this is the piece of paper where I started to scribble all of the possible words that meant something to do with the product. So you go through them and you try and think of what are the good words to use? And the obvious words that came out, 'logic' was always a good strong word. And so we had obviously become amateur marketeers for creating names, thinking of names. So 'verification' and 'logic.' So those actually was even with the first list of words, were ticked off as being the best words to play on. And then you play this game of putting parts of each word together [Moorby 2013]." See Fig. 11 for what happened next. Photo by Steve Golson, of an artifact owned by Phil Moorby.

units that were often seen as risky, experimental, and immature. ASICs required different tools and simulators, and required new features that were not necessarily needed for the mainstream standard products. The ASIC groups were compelled to put resources into developing their own simulation solutions—often contracting the existing groups in their companies to modify or enhance their existing proprietary simulation tools. Customers began designing their own chips, and new companies emerged to serve as dedicated semiconductor foundries. These customers needed design tools, and generally they were provided by the ASIC vendor as a *design kit* which also included

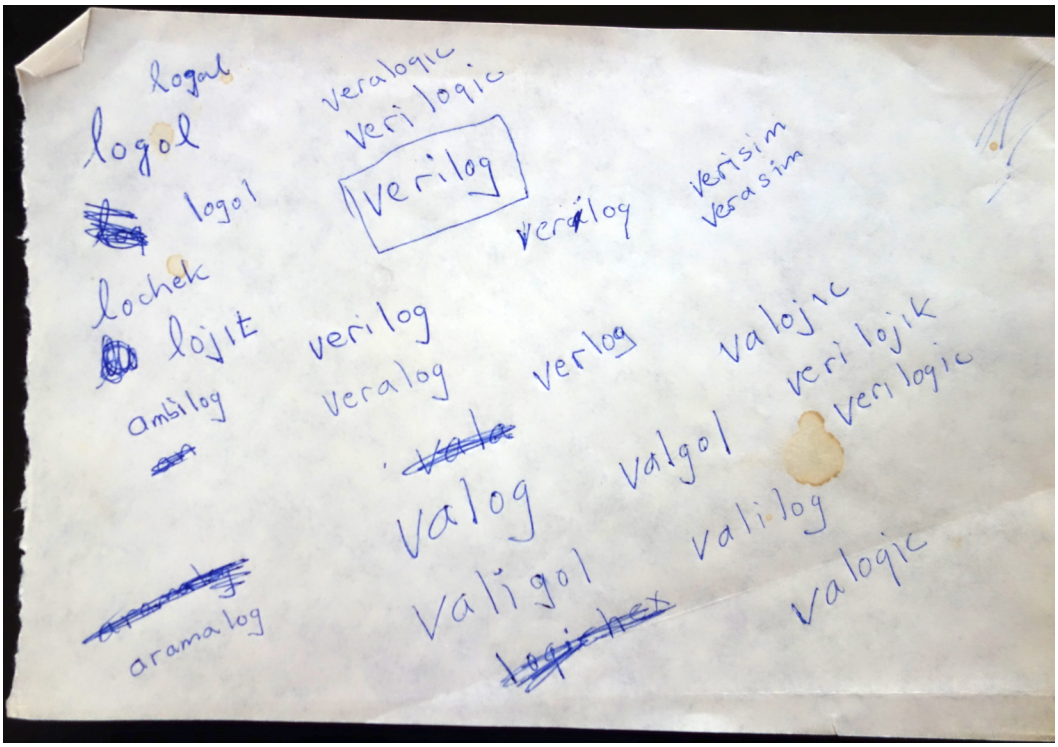


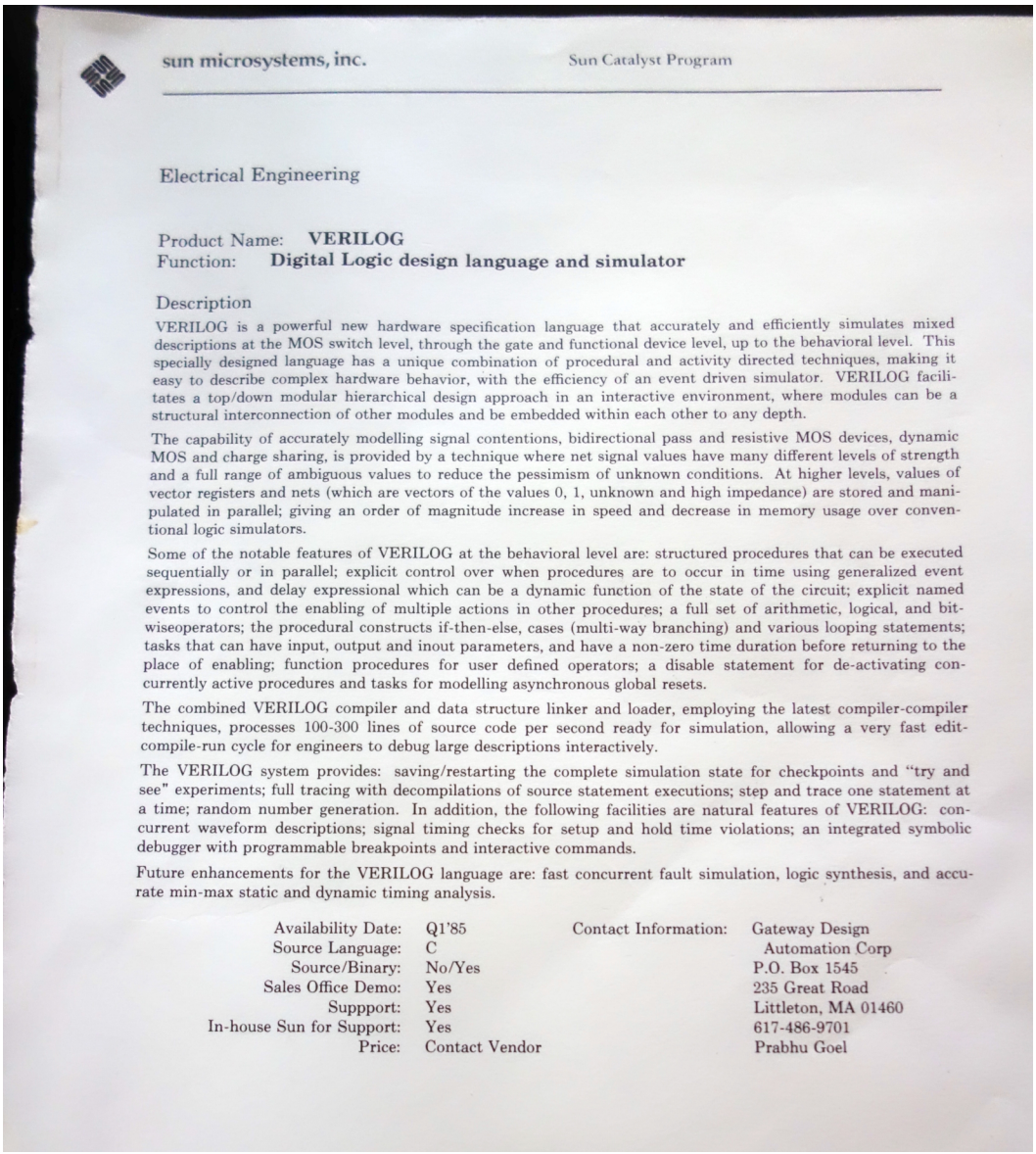
Fig. 11. Phil Moorby's possible names for the new language, in early 1985. See Fig. 10 for the start of this naming exercise. Moorby recalled, "So a whole bunch of wordplay to put parts of the two words together. And I think Verilog sort of popped out very quickly.... But it never quite sounded—initially I thought it didn't roll over the tongue, really that well [Moorby 2013]." Photo by Steve Golson, of an artifact owned by Phil Moorby.

the vendor's *cell library* comprising different views (for example, functional, timing, layout) of the various logic gates (cells) provided by the vendor.

Verilog offered many advantages to these new chip designers that entered the industry as a result of the ASIC revolution:

- **Simplicity**: The simple yet powerful syntax was attractive to hardware designers (many with little software engineering experience).
- **Procedural style**: Starting with the heavily declarative style of HILO 2, Verilog added rich procedural elements with support for function calls.
- **HDL and testbench in one**: In addition to pure hardware description, features of the language enabled sophisticated test benches.
- **Very fast multilevel simulator**: Behavioral, gate-level, and switch-level constructs were all supported.

By the late 1980s Gateway offered Verilog HDL for gate-level design and testbench with Verilog-XL as a fast simulator featuring accurate modelling of gate and switch elements with simulated delays. Gateway spent considerable effort working with ASIC vendors to convince them to use Verilog, initially internally and then externally for their customers, with Verilog-XL becoming part of their ASIC design kits. Gateway and its Verilog were not the only vendors/solutions being



sun microsystems, inc. Sun Catalyst Program

Electrical Engineering

Product Name: **VERILOG**
Function: **Digital Logic design language and simulator**

Description

VERILOG is a powerful new hardware specification language that accurately and efficiently simulates mixed descriptions at the MOS switch level, through the gate and functional device level, up to the behavioral level. This specially designed language has a unique combination of procedural and activity directed techniques, making it easy to describe complex hardware behavior, with the efficiency of an event driven simulator. VERILOG facilitates a top/down modular hierarchical design approach in an interactive environment, where modules can be a structural interconnection of other modules and be embedded within each other to any depth.

The capability of accurately modelling signal contentions, bidirectional pass and resistive MOS devices, dynamic MOS and charge sharing, is provided by a technique where net signal values have many different levels of strength and a full range of ambiguous values to reduce the pessimism of unknown conditions. At higher levels, values of vector registers and nets (which are vectors of the values 0, 1, unknown and high impedance) are stored and manipulated in parallel; giving an order of magnitude increase in speed and decrease in memory usage over conventional logic simulators.

Some of the notable features of VERILOG at the behavioral level are: structured procedures that can be executed sequentially or in parallel; explicit control over when procedures are to occur in time using generalized event expressions, and delay expressional which can be a dynamic function of the state of the circuit; explicit named events to control the enabling of multiple actions in other procedures; a full set of arithmetic, logical, and bit-wiseoperators; the procedural constructs if-then-else, cases (multi-way branching) and various looping statements; tasks that can have input, output and inout parameters, and have a non-zero time duration before returning to the place of enabling; function procedures for user defined operators; a disable statement for de-activating concurrently active procedures and tasks for modelling asynchronous global resets.

The combined VERILOG compiler and data structure linker and loader, employing the latest compiler-compiler techniques, processes 100-300 lines of source code per second ready for simulation, allowing a very fast edit-compile-run cycle for engineers to debug large descriptions interactively.

The VERILOG system provides: saving/restarting the complete simulation state for checkpoints and "try and see" experiments; full tracing with decompilations of source statement executions; step and trace one statement at a time; random number generation. In addition, the following facilities are natural features of VERILOG: concurrent waveform descriptions; signal timing checks for setup and hold time violations; an integrated symbolic debugger with programmable breakpoints and interactive commands.

Future enhancements for the VERILOG language are: fast concurrent fault simulation, logic synthesis, and accurate min-max static and dynamic timing analysis.

Availability Date:	Q1'85	Contact Information:	Gateway Design
Source Language:	C		Automation Corp
Source/Binary:	No/Yes		P.O. Box 1545
Sales Office Demo:	Yes		235 Great Road
Suppport:	Yes		Littleton, MA 01460
In-house Sun for Support:	Yes		617-486-9701
Price:	Contact Vendor		Prabhu Goel

Fig. 12. Early 1985 description of Verilog from Sun Catalyst Program catalog [Moorby 2013]. This is the first public reference to Verilog. The description emphasizes Verilog's ability as a mixed-level simulator "…at the MOS switch level, through the gate and functional device level, up to the behavioral level." Note the final paragraph listing future enhancements: "…fault simulation, logic synthesis, and accurate min-max static and dynamic timing analysis." Photo by Steve Golson, of an artifact owned by Phil Moorby.

offered in the commercial market. Some of the competitors to Verilog included HILO, Tegas, Ella, Validsim, SimuCad, Speedsim, Finsim, and VHDL tools proposed by many vendors.

A key factor that led to the acceptance of Verilog by ASIC vendors was that Gateway listened very carefully to feedback from partners' technical teams and enhanced Verilog HDL and simulators

Fig. 13. Image from a 1987 Gateway Design Automation marketing brochure. Left to right: Chi-lai Huang, Prabhu Goel, Phil Moorby. Additional information about each can be found in People on Page 73. Artifact owned by Phil Moorby.

to address their issues. Examples of enhancements to Verilog driven by ASIC vendors were accurate pin-to-pin delays (using `specify` blocks), charge storage and decay modelling, timing checks and back-annotating delay calculator (using SDF, the Standard Delay Format) which would allow different timing to be used for different ASIC libraries and different circuit topologies.

These enhancements indicated to the ASIC vendors that the developers of Verilog really understood their issues and eventually led to wide industry adoption of Verilog as the standard ASIC design language and Verilog-XL as the ASIC sign-off simulator.

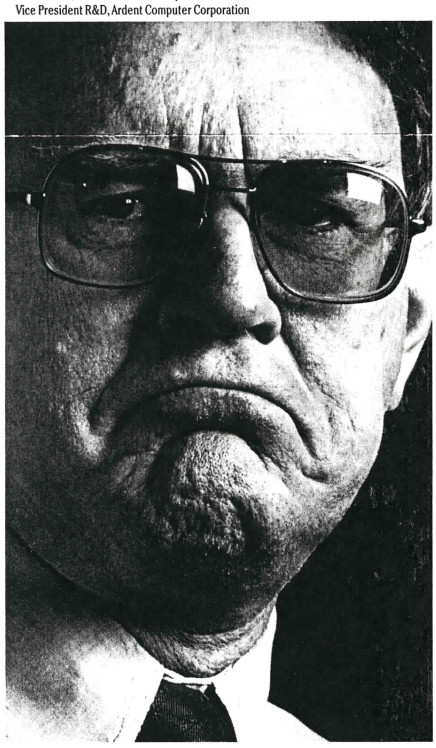
As adoption of Verilog HDL accelerated for ASIC design, the Verilog-XL simulator became the de facto tool in verifying that the gate-level design of the chip was correct. ASIC vendors would check the detailed results of the simulation in terms of logical and timing accuracy and classify the Verilog-XL simulator with their ASIC libraries as "sign-off quality." This meant they accepted its results as true and correct, and that the user would get the same results from the silicon. If the ASIC vendor also used Verilog-XL internally as their reference, then Verilog-XL was referred to as their Golden Simulator. Verilog-XL became the sign-off (or reference) simulator and Golden Simulator for most ASIC vendors.

Early Gateway revenues came just from test generation products, but now Gateway grew very quickly thanks to their industry-leading Verilog-XL simulator. Annual revenue increased from $1M in 1985 to $12M in 1989, with a backlog of orders worth $17M [Bell 1991].

Gateway approached product marketing in a nontraditional manner. For example, in 1988 Gateway ran a series of full-page ads in *EE Times* with testimonials and photographs of Gene Amdahl, Forest Baskett of Silicon Graphics, Gordon Bell of Ardent, and Tom West of Data General. The humorous ads were filled with quotes from these famous computer engineers touting how Verilog-XL enabled them to design world-class products (Fig. 14). Within a few years, most CAD companies had switched to testimonial-style ads.

GORDON BELL GAVE US A PIECE OF HIS MIND

Gordon Bell, father of the minicomputer & VAX,*
Vice President R&D, Ardent Computer Corporation



When we tried to persuade Gordon Bell to tell you that using Gateway's Verilog-XL® had given him some time off from his work as a world famous computer guru, he really gave us a piece of his mind.

"Are you kidding? Do you know what it takes to produce a new generation of computers?

"I'm talking about a *complex* project...a new class of computer. This graphics supercomputer operates at 64 mips, 64 megaflops, and delivers 50 million pixels per second. It's got 10 custom chips and over 100,000 gates!

"Now, it's *impossible* to build a machine like this without modern simulation tools. New computers – especially new classes of computers – have to be built by new companies. And because start-ups are fragile by nature, there's no margin for error. The machine must work right the first time.

"When I told Glen Miranker, our chief architect, that we *had* to simulate our Titan,™ he told me that Gateway's Verilog-XL was the only way to go.

"And he was dead right. We needed the best tool for architectural exploration and the best tool for gate-level implementation, and they had to work *together*. Gateway's put them together in one tool that's an engineer's dream."

If luminaries like Gordon Bell say this about Verilog-XL, don't you think it's time to give it a piece of *your* mind?

To qualify for our interactive Verilog-XL demonstration program, or to examine our line of verification and test products, call 1-800-356-2631 today. Gateway Design Automation Corporation, Six Lyberty Way, P.O. Box 573, Westford, Massachusetts 01886.

Gateway

VAX is a registered trademark of Digital Equipment Corp. Titan is a trademark of Ardent Computer Corp.

Fig. 14. 1988 display ad from *EE Times* featuring Gordon Bell, with a humorous photo and quotes explaining the importance of Verilog-XL. Artifact owned by Phil Moorby.

Fig. 15. Gateway Design Automation corporate party in late 1989, around the time of the Cadence acquisition. Most employees were working on Verilog and related products. See Fig. 22 on Page 82 for an annotated version of this image. Photo provided by Simon Davidmann.

By 1989 most Gateway employees were working on Verilog and related products (Fig. 15). Many customers referred to the company simply as "Verilog."

5.1 Logic Synthesis

The most important breakthrough that enabled the ASIC revolution was the adoption of Verilog by Synopsys for their Design Compiler synthesis tool.

Synopsys initially marketed Design Compiler as a logic optimization tool, which converted a gate-level description (netlist) into a functionally equivalent netlist that was faster and perhaps smaller. However their intent all along was to enable design at higher levels of abstraction by synthesizing from a behavioral RTL description into a netlist comprising gates selected from a target ASIC library [Domic 2015].

Up until the late 1980s, the conventional way to design digital hardware was with a schematic drawing system, comprising a workstation with a graphic display and schematic capture software. This software had a library of cells such as logic gates and flip-flops that matched the layout software to each specific ASIC vendor. There were also translators from the schematic data to various netlist languages. A netlist is fairly simple: it is primarily a description of the electronic components in a circuit and their interconnections. A netlist can take many forms such as a "simulation netlist" annotated with (gate) delay information, a "physical netlist" with geometric information for placement and routing on a silicon die, or a "generic netlist" such as EDIF (Electronic Design Interchange Format), which was devised as a standard but not extensively used in ASIC design flows.

With the arrival of logic synthesis, the design was entered using a text editor, and a schematic became an output rather than an input of the design flow. Consequently the choice of HDL became important: designers have to be trained in its use, and have access to a choice of design tools that accept the language. This drove the adoption of standard HDLs.

But, which language to use? VHDL was one consideration, but by this time Verilog-XL was the predominant and fastest gate-level simulator, and there was no equivalent simulator yet available for VHDL. In 1987 Synopsys approached Gateway and asked for permission to use Verilog, which was still a proprietary language. After some persuasion Gateway agreed, and this strategic partnership benefited both companies greatly [Jones 2009].

The Verilog-XL simulator and Design Compiler had to agree on their interpretation of each language feature. There were many aspects of the language that were codified and cleaned up as various discrepancies were uncovered, for example expression bit widths, the inference and modeling of state elements (flip-flops), and signed vs unsigned operations. Also new features were added to the language to support synthesis, for example nonblocking assignments. Design Compiler allowed synthesis of logic in ways not anticipated by Moorby when he first defined the language [Moorby 2013].

Nonblocking Assignments. Recall from Section 4.7 that in early Verilog, sequential logic was typically modeled as follows:

```
always @(posedge clock) r0 = #1 data;
always @(posedge clock) r1 = #1 r0;
```

When Synopsys started to use Verilog for its Design Compiler synthesis tool, their tool would ignore all explicit delays and thus render the continuous assignments into zero-delay calculations and updates. This gets us back to the same old non-deterministic race condition.

Essentially, two things are required in classic state machine descriptions: one is a mechanism for temporarily saving the state assignment evaluations before assigning to the state variables, and the other is a mechanism for delaying the updates to the state variables until all state assignment evaluations are complete [Bartee 1960]. In classic RTL simulation this is known as a two-pass algorithm. It was also required by the synthesis tools that there be no advance in simulation time before the state variables were updated.

Dave Rich was an applications engineer at Gateway, and was most familiar with Design Compiler from working with customers. He proposed to Phil Moorby that Verilog be extended to provide a cleaner mechanism to model these assignments, and Moorby quickly came up with a solution. The hardest part was convincing Prabhu Goel to allow this enhancement to Verilog, as he by default would not allow features to be added unnecessarily. This new feature, now called *nonblocking assignments*, was available in Verilog-XL by 1989 and in Design Compiler by mid-1991.

The simulator implemented this by creating an additional event queue, called the nonblocking assignment (NBA) update event region, where the RHS results could be temporarily stored before assigning to the state variables in the NBA update event region. Also, a new syntax <= was created to indicate that an assignment must use this new region:

```
always @(posedge clock) begin
   r0 <= data;
   r1 <= r0;
end
```

The two forms of assignments (blocking and nonblocking) can be mixed to allow for software-like procedural code and hardware-like state machine code:

```
always @(posedge clock) begin
   data = a + b;    // update data immediately
   r0 <= data;      // and use the new value of data
   r1 <= r0;        // use the old value of r0
end
```

Although being convenient, there are caveats in mixing the two types of assignments. First of all, too much mixing of the assignment types gets confusing. Secondly, the outputs from both forms of assignments can feed into continuous assignment and gate inputs, that in turn can eventually trigger other sequential blocks. Although the use of multiple event regions has made describing hardware

easier, it is still impossible to completely remove hardware race conditions. The combination of simulation verification, synthesis checks, and timing analysis all help to detect these conditions.

5.2 The Synthesizable Subset of Verilog

Not every legal Verilog expression needed to be synthesized. There were many constructs that were useful only in testbenches, and other constructs which were unreasonably complicated to synthesize. Defining Verilog's *synthesizable subset* that would be recognizable by Design Compiler was a key enabler that made RTL synthesis feasible.

Some unsupported constructs would cause the synthesizer to fail with an error, some would allow the tool to complete with warnings, and still others would be silently ignored:

- Delays such as `#4` are ignored.
- An `always` block must have a single event control.
- A `reg` may be written from only a single `always` block.
- Specific coding styles must be used for flop and latch inference.
- Division `/` and modulus `%` are limited to constant power of 2.
- An `always` block sensitivity list is ignored. Synthesis assumes a complete specification.
- Synthesis assumes that `function` creates combinational logic.
- No `initial` blocks are allowed.
- The `===` operator is not allowed.
- The `$display` statement is ignored.
- Hierarchical references are not allowed.

Some restrictions were eased as synthesis tools became more sophisticated. Violating these restrictions could lead to *synthesis-simulation mismatch* [Mills and Cummings 1999].

Design Compiler introduced the use of pragmas embedded in Verilog comments. Here is an example:

```
// synopsys parallel_case
```

When applied to a `case` statement, this pragma tells the synthesis tool that all case items should be evaluated in parallel, rather than building a priority encoder as required by the language definition. These pragmas must be used carefully as they cause disagreement between the RTL simulation behavior and the post-synthesis netlist behavior.

Different synthesis tools supported various subsets of the language. Although some effort was put toward standardizing the synthesizable language subset [IEEE 1364-2002], ultimately this decision was left up to each tool vendor.

Register Inferencing With Asynchronous Set and Reset. Recall from Section 4.8 that in early Verilog, flip-flops with asynchronous set and reset inputs were typically modeled as follows:

```
always @(posedge clock or negedge reset)
  if (!reset)
     assign q = 0;
  else begin
     deassign q;
     q = dataIn;
  end
```

Synopsys used a simpler style in its Design Compiler synthesis tool:

```
always @(posedge clock or negedge reset)
  if (!reset)
     q = 0;
  else
     q = dataIn;
```

This meant that Design Compiler was not required to support quasi-continuous assignments. However this style sometimes caused subtle simulation problems, generally at simulation start-up.

5.3 Other Tools in the Flow

Another key influence on the acceptance of Verilog HDL was the relationship that developed between Gateway with its Verilog language/tools and Cadence Design Systems with its physical design tools. Many other tools emerged from startups as well as the existing EDA vendors, including formal equivalence checkers, timing analysis tools, fault simulators and test pattern generators.

With partnerships in place and a recommended design flow from RTL to silicon, ASIC vendors and end users could see a use model that would enable them to efficiently design and verify their ever-larger chip designs. And, end users were not locked into a particular ASIC vendor's proprietary design tools, formats, and flows.

A further factor in the move away from the proprietary in-house tools of the ASIC vendor was the cost of maintaining internal software. As there were more design requirements and software got more complex, larger teams were needed to develop and support the new required tool features. ASIC vendors' core competency was in the silicon and semiconductor processes and not software tools. Over time they decided they were not in the business of building software. With Gateway working hard to meet their requirements and providing good technical solutions, there was less need for ASIC vendors to provide their own simulation tools. By the early 1990s Verilog was the most common language used for design tools and related methodology in designing ASICs.

5.4 Cadence, Opening Up Verilog, and Language Wars

Verilog was not the only HDL available. VHDL was a nonproprietary language developed by US Department of Defense, and in 1987 it became an IEEE standard [IEEE 1076-1987]. VHDL was popular with many users and EDA tool vendors because it could be used with no licensing cost. Also US military suppliers were required to document their designs using VHDL [MIL-STD-454L 1988]. However, vendors had dissimilar interpretations of the standard, and this became a source of frustration for early VHDL users [Carroll 1993]. Furthermore while Verilog-XL offered a standard timing back-annotation procedure, VHDL specified no such mechanism, thus each VHDL simulator vendor used a different scheme [Smith 1993].

In 1988 the 1st VHDL Users Group meeting was held as a "birds-of-a-feather" session at DAC (Design Automation Conference). In the fall of 1988, an independent VHDL Users Group meeting was held, and semiannual meetings continued afterwards. In 1991, the organization VHDL International (VI) was founded and the conference was renamed VIUF (VHDL International Users Forum) [Accellera 2002b].

Meanwhile, Gateway had discussions with Synopsys about merging the two companies, and Gateway also considered going public via an IPO [Jones 2009]. Instead, in late 1989 Cadence Design Systems acquired Gateway, and Cadence continued to sell Verilog-XL and Gateway's other Verilog tools [Goel 2017; Nenni and McLellan 2019].

By the late 1980s the stage was set for what became known as the "language wars," pitting Verilog versus VHDL. As an IEEE standard, VHDL was free to use. Many companies were eager to use Verilog in a similar way, and this put pressure on Cadence to open up its language. In 1990, Cadence released the Verilog language to a newly formed nonprofit organization called Open Verilog International (OVI). The language definition entered the public domain and became available to any vendor [Sutherland et al. 2006]. The first public description of the language was a textbook written by Don Thomas of CMU and Phil Moorby, and published in December 1990 [Thomas and Moorby 1991].

Nevertheless VHDL was enjoying widespread support. The 1993 IEEE VHDL standard officially clarified the language ambiguities and introduced a few new features [IEEE 1076-1993]. The problem of inconsistent ASIC library models and different back-annotation methodologies across vendors was finally solved with the VITAL (VHDL Initiative Towards ASIC Libraries) IEEE standard issued in 1995 [IEEE 1076.4-1995].

The conventional wisdom of the early 1990s was that VHDL was going to win the "language wars" [Collett 1993]. Industry analysts such as Ron Collett and Gary Smith predicted VHDL revenues would overtake Verilog in 1992–1994 [Cooley 1996]. A 1992 survey by MJ Associates predicted Verilog usage would decline, and that VHDL would command 75% of the market by 1997 [Jain 1993].

Despite the predictions, Verilog continued to prosper. The first significant competition to Cadence's Verilog-XL simulator came from a new startup company called Chronologic Simulation (their simulator known as VCS) founded by John Sanguinetti and Peter Eichenberger in 1991. The following year there were about six U.S. companies who had announced or were developing Verilog simulators [Borrione et al. 1992], and by 1993 Verilog had twice the market share of VHDL [Thomas and Moorby 1995].

OVI began sponsoring the annual International Verilog Conference (IVC) in 1992. That same year, OVI began working toward establishing Verilog as an IEEE standard. The working group held its first meeting in late 1993. There were many calls to make significant changes to the language as part of this first standards process, but ultimately the working group decided to instead codify the existing behavior of Verilog-XL (and other tools such as Design Compiler) and leave any significant improvements for the future. Verilog became an IEEE standard in December 1995 [IEEE 1364-1995]. The standard is organized somewhat like a user's guide, as it was based directly from the OVI document, which itself came from the Gateway/Cadence Verilog-XL manual [Sutherland 2000].

By 1995, every major CAD vendor was supporting Verilog. Simulators, synthesizers, and tools were available from over 40 companies [Thomas and Moorby 1995].

VHDL usage began to lag. The fall 1995 VHDL International User Forum (VIUF) had a difficult time attracting attendees [Cooley 1995]. In contrast, the International Verilog Conference (IVC) continued to have strong growth. For 1996, the two HDL conferences (VIUF for VHDL, and IVC for Verilog) announced they would co-locate [Madhavan 1997]. Two years later they were combined into a true joint conference [Baird 1998], and in 1999 it was renamed the 8th Annual HDL Conference and Exhibition (HDLCon) [Baird 1999]. In 2003, the name changed again to Design and Verification Conference and Exhibition (DVCon) [Weiler 2003].

In February 2000, the two language groups VHDL International (VI) and Open Verilog International (OVI) merged to form Accellera [Moretti 2015].

5.5 New Features for Verilog

After the 1995 standard was complete, the IEEE working group began addressing the many suggested enhancements to the language. Users working at higher abstraction levels wanted to expand and

improve Verilog for behavioral and RTL modeling, while low-level users wished to improve the capability for ASIC design and signoff. The specific issues and desires included the following:

- Clean up and consolidate the 1995 standard
- Generate statement (inspired by VHDL `generate`)
- Multi-dimensional arrays
- Enhanced Verilog file I/O
- Re-entrant tasks
- Standardize Verilog configurations
- Enhance timing representation
- Enhance the VPI routines

Ultimately all of these enhancements were incorporated into the new standard. Several other improvements were also included:

- Constant functions
- ANSI C-style port declarations
- Standard syntax for attributes
- Part selects for memories and variables
- Improved syntax for sensitivity lists: `@*`
- Signed types

After five years of work, this new standard was approved in early 2001 [IEEE 1364-2001]. This was the first major enhancement to Verilog HDL since 1985.

6 SUPERLOG

After over a decade of extensive use, and as hardware designs and testbenches became larger and more complex, by the mid-1990s Verilog began to exhibit its shortcomings. There was much speculation of replacing HDLs altogether, instead using C++ or Java for hardware design. However, having apparently won the language wars against VHDL, the existing and enthusiastic base of Verilog designers were reluctant to give up their favorite language. Switching away from Verilog would be an expensive and risky move, given the large amounts of legacy Verilog code, the experienced and knowledgeable user base, and the significant existing investment in tools and flows.

Another possibility was to keep the strengths of Verilog while extending the language.

Following several years of discussion, in 1997 Simon Davidmann and Peter Flake (two of the original HILO team) set up a company, Co-Design Automation Inc., to design and implement a new language and simulator. The company name showed the desire to include software/hardware co-design, but there was little customer interest in this compared with hardware design and verification, and even system specification.

Their original vision of Superlog (derived from “super” and “Verilog”) was to have a single language for system specification, hardware design, hardware verification, and software development. This was influenced by VHDL, which was originally intended as a specification language but had some success as a hardware design language and was claimed to be better than Verilog for writing complex testbenches. However Davidmann and Flake felt that Verilog was a much better HDL because it is more compact, closer to hardware, and allows faster simulation. What Verilog lacked were the features of a general-purpose programming language, which are useful for the complex testbenches. Given the syntactic similarity to C, the performance benefits of C, and the fact that C was well known both in the EDA community and in the embedded systems community, C was the obvious choice for features to copy. However, both Verilog and C have fixed size data types, and

Fig. 16. The whole of Co-Design attends Design Automation Conference (DAC) in 1999. Left to right: Dave Kelf, Christian Burisch, Lee Moore, James Kenney, Simon Davidmann, Peter Flake, Matthew Hall. Additional information can be found in People on Page 73.
In a news story [Clarke 1999] published just before DAC, well-known EDA analyst Gary Smith remarked that Co-Design has a fair chance of establishing its language, and said, "The Verilog guys are saying they've run out of steam. The VHDL guys are pretty much saying VHDL is dead. C++ isn't going to work at all, and the C guys can't come up with a solution unless they really restrict the problem."
Photo provided by Simon Davidmann.

this makes it clumsy to handle data such as strings, queues, or sparse arrays. So, these variable size data types were added too.

Another influence was VHDL+, a language developed at ICL in Manchester in the late 1990s [Wilkes and Hashmi 1999]. This had the concept of interface as a bundle of ports in different directions with a description of the communication protocol used. The ability to name a group of wires, like a structure names a group of variables, is useful for synthesis as well as simulation. Enumerated types and new control structures are also useful for synthesis, as discussed below.

Co-Design obtained its first seed round of funding in June 1998. Early investors included Andy Bechtolsheim (co-founder of Sun Microsystems), Rich Davenport (CEO of Simulation Technologies), John Sanguinetti (developer of VCS and co-founder of Chronologic), Rajeev Madhavan (co-founder of Ambit and Magma), and Venk Shukla (VP at Ambit) [Clarke 1999; Sutherland et al. 2006]. All were quite interested to see a new HDL developed to make digital designers more productive.

These key technology leaders in EDA were backing the Co-Design vision of extending Verilog and creating a super Verilog.

Flake and Davidmann had discussed their ideas with Phil Moorby during the earliest days of Co-Design, and he served on the Co-Design technical advisory board. In late 1999 Moorby joined Co-Design full time.

While the first version of Superlog was based on Verilog-95, it was not a strict superset [Flake and Davidmann 2000]. There was a desire to "clean up" some of the messy parts of Verilog, and to avoid switch-level modeling. The simulator was implemented for both languages, with different parsers for Verilog (.v) and Superlog (.sl) files. To allow existing tools to be used, a translator from a Superlog subset to Verilog was implemented. Subsequently, when Dave Rich joined Co-Design Automation in 2000, he persuaded everyone that making Superlog a strict superset of Verilog would make it much more acceptable to customers. This should include Verilog-2001 features, which were in the process of being approved (see Section 5.5). The change involved re-working test cases in the regression suite as well as code and documentation, but it was essential to adoption and success.

The idea of software/hardware co-design led to the idea that just using the C features of Superlog should allow code generation that is just as efficient as C. Therefore the C features should be implemented to match the output of a C compiler. This would also allow a smooth interface between Superlog and C.

6.1 Packed and Unpacked Data Types

The first task in extending Verilog with C features was to produce a coherent set of data types and operators. In Verilog the essential data type is a vector of elements based on four values 0,1,X,Z. In C the essential data type is the integer, normally 32 bits, with shorter and longer variations. The arithmetic and bitwise logical operators operate on these essential data types.

The unification had two kinds of elements, four-valued and two-valued, called `logic` and `bit` respectively. A cast from `logic` to `bit` maps 1 to 1, and the other values (0, X, and Z) to 0. This matched the Verilog behavior with conditional expressions. Logic vectors and bit vectors could be of any static length up to 4095, and could be signed or unsigned, the latter being the default. For Verilog or C compatibility, keywords defined some vector lengths. An `integer` is a signed logic vector of length 32, a `time` is of length 64. An `int` is a signed bit vector of length 32, a `shortint` is of length 16, `char` of length 8, and a `longint` is of length 64.

These vectors are both operands and results of the arithmetic and bitwise operators. Verilog also had the memory data type, which was an array that was processed one indexed element at a time, and C had multi-dimensional arrays which were also processed one element at a time, usually in a `for` loop. This formed the distinction between a packed array and an unpacked array. It was possible to have an unpacked array of packed array elements, but not vice versa:

```
bit [7:0] Mem [0:4095]; // unpacked array of 4096 bytes (packed array of 8 bits)
```

It was convenient to map the index of a packed array to its bit position, using pairs of words for a logic array. This allowed a cast to be implemented using bitwise operators. Using a single operator for many bits gave a much higher performance than processing one bit at a time, and continued the implementation techniques of HILO 1 and Verilog-XL.

Both C and VHDL can have structures or records of named bit fields. Data packets are one example where they are useful. In Superlog each field is a `bit` or `logic` vector, and the whole type can be expressed as a packed structure, which can be used as an operand for arithmetic, just like a packed array, and can be signed. The data type can be named as in C:

```
typedef struct packed {   // default unsigned
  bit [3:0] GFC;          // field of four bits
  bit [7:0] VPI;
  bit [11:0] VCI;
  bit CLP;                // single bit
  bit [3:0] PT ;
  bit [7:0] HEC;
  bit [47:0] [7:0] Payload; // packed 2-D array
  bit [2:0] filler;
} s_atmcell; // structure name
```

An unpacked structure is like a C structure in that it cannot be used as an operand for arithmetic and can contain other data types such as reals and pointers. It can be implemented to match a C compiler, with unused space between small fields. An unpacked structure is similar to a VHDL record.

A packed union is a union of packed arrays or packed structures of the same size. A packed union can also be used as an operand for arithmetic.

```
typedef union packed {  // default unsigned
   s_atmcell acell;     // structure
   bit [423:0] bit_slice; // viewed as 1-D array
   bit [52:0][7:0] byte_slice; //viewed as 2-D
} u_atmcell;  //union name
```

Writing one member and reading another gives predictable results, because all the members are the same size and ordered left to right.

An unpacked union is like a C union and can be implemented to match a C compiler. There is no corresponding concept in VHDL. Because the members do not have to be the same size, writing one member and reading another can give an unpredictable result.

An anonymous union, without a typedef, can be used to create aliases.

6.2 More C Types

Enumerations in C are just integer constants, and this is the default in Superlog. In hardware it is important to specify how many bits are needed to represent the constants and this led to two alternative methods:

```
enum {bronze=4'h3, silver, gold} medal4;
      // bronze is a 4 bit constant, hexadecimal 3
      // medal4 variable is also 4 bits wide

enum logic [1:0] {red, yellow, green, unknown = 'x};
                  // constants 2 bits wide 4-valued 0,1,2, or X
```

The C syntax for declaring pointers can be confusing:

```
int * a, b;      /* a is a pointer to int, b is an int */
```

Superlog borrowed the Algol syntax:

```
ref int a, b;    /* a and b are both pointers to int */
```

This requires more use of typedefs for complex data types than C does, but is easier to read.

Pointers to tasks or functions can be declared as in C:

```
// pointer to function with string argument returning int
ref function int(string) convert;

// pointer to task with string argument
ref task (string) printout;
```

The unary & is a vector reduction operator in Verilog, so the `ref` keyword is also used to get an address, and the `deref` keyword to use an address like unary `*` in C.

In floating point, the `real` in Verilog corresponds to the `double` in C, and so a new type `shortreal` was added, corresponding to `float` in C.

6.3 Data Declarations

The second task in extending Verilog with C features was to add declarations with initializers and dynamic memory. In Verilog-95 all memory is static, corresponding to hardware, and initialized to unknown X. In C, variables default to automatic (that is, stack) within functions. For Verilog compatibility, the keyword `automatic` is required in declarations of variables, functions, tasks, or blocks, otherwise static behavior is assumed.

Verilog does not allow variable declarations at the start of unnamed `begin...end` blocks, so this C feature was added in Superlog.

The Verilog distinction between registers (`reg`) and wires (`wire`) is blurred in Superlog. A new keyword `logic` declares a variable which, if static, can either be driven by a continuous assignment; or an output port (like a wire); or written by one or more procedural statements, where the last write determines the value (like a register). An assignment in the declaration is treated as an initializer, not as a continuous assignment.

6.4 Expressions

The syntax of Verilog expressions was extended to include the C assignment operators such as `+=` and `-=` as well as increment `++` and decrement `--` operators. Parentheses are required around statements to make them expressions, removing a common source of error: the use of = instead of ==.

The Verilog rules on sign and size propagation were used, and these differ from C, which converts operands to `int`.

6.5 Functions and Tasks

In Verilog a `function` can return only a single value, and all its arguments are inputs. In Superlog this is generalized so that arguments can be output or inout, and the `function` can return void if it is called as a statement, like in C.

In Verilog all `task` arguments are passed by value/result. Superlog adds passing by reference, using the keyword `port` for a variable which can be used in an event expression, and `shared` for one which cannot. Passing by reference means that changes during the execution time of the `task` are visible to the calling environment.

6.6 Memory Management

Superlog followed the C model of explicit allocation (`$alloc(data type)`) and de-allocation (`$free`) of heap memory, with the option of a safe mode for pointers. In fast mode the pointers were implemented as in C. In safe mode each pointer went via a structure which is on a linked list of references to the object pointed to. This structure was inserted in the list when the pointer was

assigned to that object and removed when the pointer variable was re-assigned or disappeared. This allowed warning of dangling pointers or memory leaks.

In practice, safe mode was not used very much because the C interface required fast mode for C compatibility, and the new data types reduced the need for explicit pointer manipulation.[16]

6.7 New Data Types

String manipulation is often required for debugging purposes and to present output in a user-friendly way. Both Verilog and C are awkward to use beyond formatted output functions `$display` and `printf` because the size of every array has to be declared, and the memory freed manually. Providing a string data type of variable size with automatic memory management, combined with the concatenation and slicing operators from Verilog makes this much easier.

A `list` or `queue` is a variable size unpacked array of a particular data type. It is implemented with bidirectional pointers to be equally efficient whichever end is used for insertion and removal.

```
int q1[0:$];     // unlimited queue of integers
int n, item;

q1 = {n, q1};    // insert n to the left of the queue:
q1 = {q1, n};    // insert n to the right of the queue:
item = q1[0];    // read the left most item:
item = q1[$];    // read the right (last) item:
n = q1.$num;     // number of items in the queue:
q1 = q1[1:$];    // delete the left item
q1 = q1[0:$-1];  // delete the right (last) item
q = {};          // clear queue
```

To model large memories without requiring even more simulator memory, a sparse array must be used. This is tedious and error-prone for a user to do with pointers, so the keyword `sparse` can be used with an unpacked array, providing a built-in lookup tree implementation. A similar implementation is used for associative arrays such as:

```
int lookup [string] = {default:0};   //unpacked array of int indexed by string
```

It is an error to read a non-existent entry unless a default value has been set.

6.8 New Control Flows to Prevent Synthesis-Simulation Mismatch

Synthesis needs additional information in Verilog, which is supplied by formatted comments (see Section 5.2). For example, the pragma `full_case` means that an unstated case cannot occur; the default action is don't care. Pragma `parallel_case` means that all values are unique; multiple match conditions cannot occur. Simulators do not act on these comments as they are not part of Verilog, so language enhancements were needed.

The keyword `unique` before an `if...else if` or a `case` statement means that there are no overlapping values in the list of conditions and no missing value can occur. The keyword `priority` before an `if...else if` means that there are overlapping values, so priority logic must be synthesized. For either keyword, if a missing value occurs in a simulation, the simulator would issue a warning.

The keyword `always_comb` introduces a block which is intended to model combinational logic. This means that all variables assigned in the block must be assigned under all conditions. The restriction applies to functions called within the block. The block is triggered by any value change

[16]Explicit de-allocation was abandoned in SystemVerilog, as discussed in Section 9.

in a variable read within the block or a function called from the block. This facility was added at customer request and allows large blocks of combinational logic to be modeled in an imperative style.

6.9 C Interface

Superlog can call C functions or tasks provided the prototype is declared with `import "C"`. Here is an example:

```
import "C" function int atoi(string);
import "C" task mytask(output int, input char, input shortreal);
```

The C function must match the prototype within the specifications of the tool used:

```
int atoi(char * s);
mytask(int *, char c, float f);
```

Note that the string data type is allowed to match a `char *` in C. This causes a string copy, unlike a `ref char` data type. When using an unpacked array of `char`, C compatibility requires a pointer to the first element, not a pointer to the whole array.

Superlog functions or tasks can be called from C provided they are exported. If the export is global they are given unique global names:

```
export "C" u1.anyfunc mycfunc;
```

If the export is local to a module, it can only be called from a C function that has been locally imported into that module, so that the context can be maintained.

6.10 Interface Construct

One of the methods of modeling a system before its detailed design is to use the "block diagram" topology with messages passed between the modules. This is sometimes called a *transaction level model* (TLM), where the transactions may be blocking (waiting for a response) or nonblocking (using polling or another process). Although Verilog allows one module to call a task in another module, it requires a hierarchical name, including the instance name of the module called, which prevents re-use. A means of providing re-usability is to route the task call through ports, with a netlist specifying the module instances and communication channels. A single task call can call multiple task instances if the `extern forkjoin task` declaration is used.

One of the goals of Superlog was to allow the same netlist code in the transaction level model and the register transfer level model. The `interface` construct provides this encapsulation of communication between modules. In its simplest form it is a bundle of wires, like a `struct` is a bundle of variables, where each wire or array of wires still has its own name. Because an interface instance can be connected through ports, it can reduce the size of code required for a netlist, even in the absence of a TLM.

To allow some wires to be connected differently, the interface itself can have ports.

For a TLM an interface can carry task or function calls from one module to another, using the same top-level netlist, by providing import and export ports. In addition, the interface can contain procedural code: tasks, functions, and processes. This allows communication to be monitored or encapsulated outside the modules. It also allows the abstraction level to be changed between transactions and wires, with a task setting shared logic values or responding to value change events.

Just as the interconnect is bundled into an interface, it is convenient to bundle the ports on a module with the `modport` construct. This is declared within the interface as shown in the trivial example of a light switch:

```
interface lightControlWires; // bundle of wires
   logic sig;  // control wire
   const logic gnd = 0; // ground wire
   modport master (output sig, input gnd); // connections to switch
   modport slave  (input sig, gnd); // connections to light
endinterface

module systemWires;
   lightControlWires li1; // bundle of wires
   switchWires S1(li1);   // switch
   lightWires L1(li1);    // light
endmodule
```

Here is a transaction level model of a light and its switch:[17]

```
interface lightControlTasks; // define channel
   modport master(import task on( ),
                  import task off( )); // initiates transactions
   modport slave (export task on( ),
                  export task off( )); // responds to transactions
endinterface

module systemTasks;
   lightControlTasks li1; // channel instance
   switchTasks1 S1(li1);  // switch instance
   lightTasks1 L1(li1);   // light instance
endmodule

module switchTasks1(lightControlTasks.master co);
   always begin  //switch initiates transactions
     co.on(); # 100s; // call task on() via the interface - may be several
     co.off(); # 100s;// off() 100 seconds later
   end
endmodule

module lightTasks1(lightControlTasks.slave co);
   task co.on(); // task to be called via the interface, hence 'co.'
      $display("on"); // print message
   endtask
   task co.off(); // light responds to transactions
      $display("off");
   endtask
endmodule
```

[17] This simple example demonstrates only control. A more realistic model would include data transfer.

Fig. 17. Most of the Co-Design staff, along with a few friends and consultants, attends Design Automation Conference (DAC) in June 2002 in New Orleans. See Fig. 23 on Page 83 for an annotated version of this image. Photo provided by Simon Davidmann.

6.11 Standardization

While the Superlog simulator, SystemSim, achieved modest commercial success including at Intel, the fact that Superlog was a proprietary language under non-disclosure agreement limited its appeal. So, in June 2001 Co-Design Automation donated the hardware design subset of Superlog to Accellera. Simon Davidmann and Peter Flake participated in a working group which spent a year debating and drafting a standard add-on to IEEE Verilog-2001. Because Superlog was a proprietary name, Accellera came up with a new name: SystemVerilog.[18] Since the Verilog-2001 standard was also called Verilog 2.0, the version 3.0 was chosen, and in June 2002 Accellera published the SystemVerilog 3.0 standard [Accellera 2002a].

7 VERA

Verilog's procedural constructs and its PLI interface provided engineers with a comprehensive solution for writing testbenches. However, as hardware designs grew more complex and incorporated increasing number of interfaces and protocols, the size and complexity of the testbench code needed to validate the design increased significantly. The ability to quickly create short, complex, and reusable testbenches motivated the development of Hardware Verification Languages (HVLs). The

[18]The Accellera working group had earlier names, including Verilog++ and HDL+.

Vera HVL was initially developed at Sun Microsystems in 1994 by the engineers Atsushi Kasuya and Eugene Zhang as a skunkworks project. The first prototype along with an enhancement wish list was transferred to Systems Science, Inc., which proceeded to productize and significantly enhance the language. In 1998 Systems Science was acquired by Synopsys, which donated the Vera language to Accellera in 2002.

Vera was implemented as a co-simulation engine working alongside a hardware simulator. Vera was initially developed to work with Verilog and using PLI for communication; it was later integrated with other HDLs such as VHDL. Because Vera was intended to work with multiple languages and simulators, it was important that the testbench behave identically regardless of the underlying HDL simulator. Vera guarantees determinism by avoiding the transient simulation effects and focusing on the steady state of the hardware DUT (device under test). This is accomplished by sampling DUT signals at precise times and executing at the end of the simulation time slot—after all clocks as well as all signal propagation has taken place.

Vera's syntax is largely based on Verilog with a few lexical and operator additions from C. The original Vera supports the same four-state data types as Verilog (integer, 1-bit scalar, bit vectors and named events); it also uses the same operators and obeys the same expression bit-width rules as Verilog. Vera creates two new data types to communicate with the simulator: `class_var` to hold opaque data (pointers) and `bind_var` to hold DUT signals. Subsequent Vera versions added support for enumerations, strings, dynamic arrays, associative arrays, and object-oriented classes. Vera supports all the Verilog procedural constructs including functions and tasks, however, Vera enhances the latter to support pass-by-reference as well as reentrancy.

7.1 Connecting the Testbench to the DUT: Interfaces, Ports, and Binds

A key Vera construct is the `interface`. The interface is a user-defined declaration that specifies the size and direction (input, output, or inout) of the DUT signals that interact with the testbench. In addition, the interface formalizes the signal timing by including a reference clock (signal) and the specific timing for sampling and driving each signal. The per-process race avoidance mechanisms used by Verilog (for example, nonblocking assignment) were deemed too cumbersome for the testbench, particularly when the testbench is concerned with functional aspects often expressed in terms of cycles. Associating the clock with the interface allows Vera code to express very compact cycle-based checks and stimulus generators that completely abstract the low-level timing of the signals. Consequently, every interface designates a clock signal—if omitted, a system-clock is added by the compiler—and the timing to read (sample) input signals or write (drive) output signals. The timing is specified by means of a clock transition designation (Positive or Negative) and time delay or skew measured from the clock transition. Each interface signal is one of five types:

NHOLD	Output is driven skew time units after the Negative clock transition and held at that value
PHOLD	Output is driven skew time units after the Positive clock transition and held at that value
NSAMPLE	Input is sampled skew time units before the Negative transition of the clock
PSAMPLE	Input is sampled skew time units before the Negative transition of the clock
CLOCK	Designates the clock—one for each interface, no transition or skew allowed

A testbench can include multiple interfaces; each one designates a clock domain. Here is an example of a simple Vera interface:

```
interface channel {
   input      clock CLOCK;      // "clock" is the clock signal
   output     reset PHOLD #1;   // single-bit signal driven 1 time-unit
                                //   after the rising clock edge
   input [7:0] data PSAMPLE #1; // 8-bit signal sampled 1 time-unit
                                //   before the rising clock edge
   input    request PSAMPLE;    // signal sampled infinitesimal time
                                //   before the rising clock edge
   output       ack NHOLD #2;   // signal driven 2 time-units
                                //   after the falling clock edge
}
```

Vera includes two constructs, `port` and `bind`, that allow the creation of virtual ports: variables that can be used to pass sets of interface (DUT) signals around a testbench. The `port` construct defines a symbolic container that associates a set of names with a particular port declaration. The `bind` construct declares a `bind_var` variable of the designated (virtual) port and binds it to a specific set of interface signals. An example of these two constructs is shown below:

```
port tx_port {
  req;
  data;
  ack;
}
```

```
bind tx_port port_A {
  req  channel.request;
  ack  channel.ack;
  data channel.data;
}
```

```
tx_port p1 = port_A;
p1.$ack = 1;
```

The `bind_var` variable[19] `port_A` is a specific (bound) instance of port `tx_port`. One port can be associated (bound) with multiple sets of interface signals. The port name can be used to declare variables of that (virtual) port and those variables can be assigned any of the (bound) variables created by a corresponding `bind`. This is shown in the third column above: after variable `p1` is assigned `port_A`, it can be used to access the underlying interface signals. The same virtual port variable can be assigned different `bind_var` variables during the execution of the testbench. A port can also be used as a function argument, thus allowing a single piece of code to manipulate multiple hardware interfaces. It's also noteworthy that a virtual port inherits all the timing information from the Vera interfaces used during the bind; all the clocking and synchronization information has been abstracted by the interface.

7.2 Driving Stimulus and Checking Results

Generally, a testbench generates stimuli for the DUT, and it also includes the set of expected DUT outputs, that is, the response or change of state that indicates the success or failure of a test. For a test to succeed, the DUT must respond with the correct values, but they must also occur in a timely manner—occurring at specific times or within some bounded interval. Vera provides mechanisms to specify both the stimuli and the expected response in a simple and succinct manner. Vera provides four basic operations to operate on interface signals: Drive, Sample, Expect, and Synchronization.

[19]The binding of these variables cannot change once bound—in this sense they are similar to C++ references.

A *Drive* statement drives an output interface signal to the specified value; a drive can optionally specify a strength (soft) as well as a number of cycles to wait before the value is presented to the DUT. The drive operation uses the Verilog assignment operator, and likewise provides two types of drives: blocking and nonblocking. Here are some examples of drives:

```
bus.data = 25;        // drive value 25 immediately (as per the specified interface timing)
@5 bus.data <= 31;    // drive value 31 5-clocks in the future, and continue execution (nonblocking)
```

The *Sample* statement also uses the Verilog assignment operator but using the interface signal as the source (right-hand side) expression—the nonblocking form is not allowed:

```
local_var = bus.data;   // Assigns to local_var the sampled value of the interface signal data
```

The *Expect* statement is a logical assertion that a given signal has the designated value within a specified time interval or at a given time. If the assertion fails, an error is generated. The expect syntax combines Verilog's event control (`@`) and equality/inequality operators (`==`, `!=`). Three types of expect statements are possible: Simple, Full, and Restricted. The Simple expect (`@`) checks that a signal changes to an expected value within a time interval or at a point in time. The Full expect (`@@`) checks that a signal maintains the expected condition over the entire window of time. The Restricted expect (`@@@`) checks that a signal changes to expected value on the first transition. Finally, any one of the expects can be combined using logical conjunction (AND) or disjunction (OR). Here are some examples of the three types of Expects:

```
@1 bus.data == 5;                          // Simple: assert that data equals 5 in 1 cycle
@0,10 bus.data == 4'b00x1;                 // Assert data equals 0001 or 0011
                                           //     within 10 cycles (x is a wildcard)
@2,20 bus.data == 5 or bus.addr != 10;     // Assert data is 5 and addr is not 10
                                           //     within 2 to 20 cycles
@@ 5,100 bus.data != 33;                   // Full: Assert addr is not 33
                                           //     in the 5 to 100 cycles interval
@@@ 0, 100 bus.addr == 1;                  // Restricted: assert the first addr change is to 1
                                           //     in the next 100 cycles
```

The *Synchronization* statement resembles the Verilog `event` operator and waits until the next synchronization point or designated transition occurs. Unlike the Drive, Sample, and Expect operations, which move simulation time only when necessary, the synchronization operation always blocks until the next designated transition (synchronization point). Here are some examples of the synchronization statement:

```
@( bus.data );              // wait until the next change of signal data
@( posedge bus.request );   // wait until the next rising transition of signal request
```

These four interface signal operations enable simple testbenches to be written very concisely. For example, to drive a signal `request` in an interface `channel` to 1, and then check that the signal `ack` in the same interface becomes 1 (in response to the request) within 100 cycles, can be written in just two lines:

```
bus.request = 1 ;         // drive request to 1
@0,100 bus.ack == 1 ;     // ensure ack becomes 1 within 100 cycles
```

7.3 Threads and Concurrency Control

Unlike Verilog, Vera does not provide structural hierarchy or static processes (for example, `always` blocks); instead, each testbench is completely contained within a single program construct. The `program` block defines a namespace and creates a single thread of execution that is intended to

create multiple threads of execution procedurally. Earlier versions of Vera were limited to a single program per testbench, but this restriction was relaxed, allowing multiple independent programs to coexist in the same test—all types and variables declared in a program are strictly local to that program's scope; direct program-to-program communication is disallowed. An example of a complete program that asserts a `reset` signal and then waits for up to 100 cycles for the `strobe` signal to be asserted is shown below:

```
program check_reset_strobe  {
   interface bus {
      output  reset PHOLD ;
      input  strobe PSAMPLE ;
   }
   { // Program entry point
      bus.reset = 1 ;           // drive reset to bus
      @0,100 bus.strobe == 1 ;  // expect strobe to become one
                                //   within 100 clock cycles
   }
}
```

Vera was largely motivated to facilitate the creation of highly concurrent but well-controlled verification programs. To that end, Vera supports a flexible mechanism for the creation of concurrent threads along with a powerful set of intra-thread synchronization and communication primitives. Vera builds on the Verilog `fork...join` construct (described above in Section 4.9) by adding two new thread joining mechanisms: `any` and `none`. The Verilog `fork...join` compels the forking (or parent) thread to stop executing until all its spawned threads terminate. Vera optionally allows the parent thread to either wait for any one of the threads to terminate (`join any`) or to not wait at all (`join none`). The `join none` variant is very useful to create an arbitrary number of background threads anywhere in the code, including tasks. Consider the example below that shows a task that spawns two threads: one that waits for signal `bus.ack` to be asserted within 1 to 10 cycles, and one that drives value `n` onto signal `bus.req`. Both threads execute concurrently with the parent thread, which drives the value 24 onto signal `bus.data`.

```
task spawn(integer n) {
   fork
     @1,10 bus.ack == 1 ;  // spawned thread
     @5 bus.req = n ;      // another spawned thread
   join none
   @4 bus.data = 24;       // parent thread
}
```

In addition to the dynamic thread creation capability, Vera supports several thread control, synchronization, and communication primitives: `event`, `semaphore`, and `mailbox`. The Vera `event` is conceptually similar to Verilog's named `event` (described above in Section 4.9) but provides a more powerful wait mechanism. Verilog uses the event control operator to wait for a single event to be triggered, while Vera provides a `sync` construct that can wait on multiple events to be triggered in different ways: `ALL` (all events are triggered), `ANY` (any one event is triggered), or `ORDER` (events are triggered in a specific order). Here are some examples:

```
sync(ANY,ev1,ev2); // wait until either ev1 or ev2 are triggered
sync(ORDER,a,b,c); // wait for events a,b,c to trigger in order (left to right)
```

Vera provides built-in functions to wait for one or more variables to change value (`wait_var`), to wait for one or more threads to terminate (`wait_child`), and to forcibly terminate a thread (`terminate`).

The *semaphore* is a synchronization mechanism that maintains an access tally and is often used to guarantee mutual exclusion to shared data using a consumer-producer scheme. The semaphore supports two operations: `semaphore_put`, which adds a designated number to the tally, and `semaphore_get`, which attempts to deduct a designated number from the tally while ensuring the tally never becomes negative. A get operation can either wait until the tally becomes large enough for the operation to succeed, or optionally check the success of the operation without waiting.

The *mailbox* implements a FIFO (first in, first out) message queue of arbitrary message types that supports two operations: `mailbox_put`, which adds a message to the queue, and `mailbox_get`, which removes a message from the (nonempty) queue. If the queue is empty, a thread can either wait until a message arrives or optionally check the empty status without waiting.

7.4 Classes

A major enhancement to Vera was the addition of an object oriented programming (OOP) framework. Classes and Objects are basic concepts in OOP and will not be covered here in detail, only the most salient characteristics and other novel Vera features built into the class system. Vera classes are very heavily influenced by Java [Arnold and Gosling 1998]. Vera implements the same single-inheritance mechanism as Java; it uses the same keywords and similar syntax for class declaration and extension. Vera supports static class methods and variables in the same manner as Java, and Vera supports the same access specifier for methods and variables: `public`, `protected`, and `private`. Vera also uses the same constructor operator as Java (`new`), and implements the same hierarchical object initialization as Java. Finally, Vera supports an automatic memory management scheme that is functionally the same as in Java. Vera's automatic memory reclamation was a deliberate decision and is used throughout the language for all dynamically allocated data, not just class objects. Without automatic memory management, Vera's multithreaded, reentrant, cosimulation environment creates many opportunities for users to run into problems. The one salient difference between Vera and Java is the polymorphism implementation: Vera distinguishes between virtual and non-virtual methods, hence, methods not explicitly declared as virtual are non-virtual and instead use hierarchical method override. This decision was influenced by C++ in order to allow Vera to generate more efficient code for non-virtual methods. Below is an example of a very simple base and derived class pair that implement an integer container.

```
class Base {
   integer count;
   task Set(integer value) { count = value; }
}
class Derived extends Base {
   function integer Get() { Get = count; }
}
```

7.5 Constrained Random Stimulus

One of the most important Vera innovations was the addition of constraint-based randomization to the language. Randomizing constraints are declarative and not procedural, which is a notable departure from the rest of the language. Users declare the constraints that delimit the solution space of the intended stimulus—typically the correct stimuli for a hardware block—and Vera generates

random solutions that satisfy the constraints. The key observation is that writing the constraints of what represent valid stimuli, and then using a computer to generate myriad valid stimuli, is much faster and more effective than a human being handcrafting all that stimuli. Hence, we trade compute time for human authoring time, which is much more valuable. Vera's constraint framework is built onto the class system thereby providing an OOP mechanism for assigning random values to the member variables of an object subject to user-defined constraints. An additional type specifier is used to declare a member variable as a random variable; variables not explicitly declared as random are considered state variables, that is, they can participate in constraints but their values are only read. Vera supports two types of random specifiers: `rand` and `randc`. The latter designates the variable as random cyclical and will cycle through all values in a random permutation before repeating any value—regular random variables have no such limit. An object is randomized by calling a built-in method called `randomize`. Consider the example below which shows a class that models a simplified bus with two random variables: `address` and `data`, representing the address and data values on a bus. The `word_align` constraint specifies that the random values for `address` must be word-aligned (the two low-order bits are 0). The declaration is followed by the class creation and a `repeat` loop that will generate 50 random values satisfying the address constraint. Note that the other variable, `data`, is unconstrained so it is randomized to an arbitrary value.

```
class Bus {
   rand bit[15:0] address;
   rand bit[31:0] data;
   constraint word_align { address[1:0] == 2'b0; }
}
Bus b = new;
repeat(50)  success = b.randomize();
```

OOP inheritance is used to allow the creation of a layered constraint system. The example below builds on the previous one by extending the class to include a new `range` random variable and a constraint that further constrains the address to two ranges depending on the value of `range`. Note that both constraints are active when an object of type `Bus2` is randomized.

```
class Bus2 extends Bus {
   rand bit range;
   constraint addr_range { ( range == 0 ) => address in {   0 : 255 };
                           ( range == 1 ) => address in { 256 : 512 }; }
}
```

An important concern when generating random stimuli is the distribution of the solution. By default, any bit in the solution space has the same probability of being 0 or 1, hence, some useful constrained solution may become very unlikely without shifting the distribution of certain variables. Vera includes a `dist` operator that allows users to specify a weight for a particular solution. When this operator is used, and absent any other constraints, the probability that the expression matches a value is proportional to its specified weight. The constraint below shows how to skew the distribution of the address variable above so that the value 0 occurs with a probability of 8.3%, 100 occurs with a probability of 50% (60/120) and 200 occurs with a probability of 41.6% (50/120):

```
constraint adr_dist  {  address dist {0:=10, 100:=60, 200:= 50}; }
```

Vera provides another semi-declarative construct for random stimulus generator based on a different premise. The Vera Stream Generator uses the `randseq` construct that allows users to specify a random sequence using a Backus-Naur Form (BNF). When the `randseq` executes, random production definitions are selected and streamed together to generate a random stream. BNF is

often used to define the syntax of programming languages; compiler generators such as Yacc [Johnson 1975] and Bison [FSF 2000] use BNF to describe the language to be parsed. Vera's stream generator uses BNF to define the language productions that represent all valid stimuli. But, unlike a compiler that uses BNF to check whether some code is a valid utterance in the language, the stream generator uses the BNF to randomly generate valid stimuli that satisfy the BNF productions. Vera's BNF allows the specification of a distribution weight to every production that specifies a choice. For example, the BNF below will generate one of the four possible sequences (idealized opcodes) shown on the right, and in which the add opcode will be selected with a probability of 90% (9/10), and dec with a probability of 10% (1/10).

```
randseq(main) {
   main :    top middle bottom;                  add   pop   mov
   top :     &(9) add | &(1) dec;       ⟶       add   push  mov
   middle :  pop | push;                         dec   pop   mov
   bottom :  mov;                                dec   push  mov
 }
```

The ranseq example above omits the details of the code blocks to clearly illustrate the construct. To complete the example and generate the stimuli (the add, dec, mov, push, and pop opcodes), users would need to add the procedural code blocks (enclosed by {}) that cause the action blocks to be executed when a particular terminal is selected. For example, to just print the push and mov opcodes requires the following revision to the code:

```
bottom : mov { printf("mov"); };
middle : pop | push { printf("push"); };
```

7.6 Functional Coverage

One final noteworthy Vera feature is the incorporation of a functional coverage system that is able to monitor states and state transitions as well as changes to variables and expressions. By setting up a number of monitor bins that correspond to states, transitions, and expression changes, Vera is able to track relevant activity during execution. Each time a user-specified activity occurs, a counter associated with the bin is incremented. By establishing a bin for each state, state transition, and variable change that represents some functional aspect of the DUT, users can track how much of the functionality has been exercised and compute the degree of completeness of the test. The main role of the functional coverage specification is to convert a typically very large state space into a manageable and actionable set of bins.

Vera's functional coverage specification is built onto the OOP class system and is contained inside the coverage_group construct declaration within the class. This construct encapsulates the specification of the coverage model or monitor, and it must be explicitly constructed (using the new operator) to activate it. A coverage_group specifies a sampling event and designates coverage-points, cross-products and a set of state and/or transition bins. The sampling event specifies when the coverage data is to be sampled and it can be any synchronization mechanism, not just an event operator. Coverage-points are declared using the sample construct that designates the variable, DUT signal, or expression of variables and signals to be sampled. Cross-products are declared using the cross construct and designate the coverage of all combinations of the bins of the associated coverage-points, that is, the Cartesian product of the sets of coverage-point bins. The following example shows a coverage group embedded within class A, which designates two 4-bit member variables x and y, and an interface signal ifc.sig as coverage points. This is followed by the definition of two cross-products cxy as the cross-product of coverage-points x and y, and sx as the

cross-product of coverage-points `x` and `ifc.sig`. Finally, the `class` constructor shows the creation of the coverage group.

```
class A {
   bit [3:0] x, y;
   coverage_group cg {
      sample_event = @(posedge CLOCK); // sampling event
      sample x, y, ifc.sig;           // 3 coverage points x, y and ifc.sig
      cross cxy ( x, y);              // cross product of x and y
      cross sx ( x, ifc.sig );        // cross product of x and ifc.sig
 }
 new() { cg = new(); }                // construct the coverage-group
}
```

8 ASSERTIONS

In general programming languages, an assertion is a statement that a (Boolean) predicate is expected to be true. If the predicate is false, a message is generated, and optionally an exception is thrown. The evaluation of the predicate expression happens when the statement is executed in the normal control flow.

In hardware description languages, an *assertion* is a statement that a predicate is expected to be true, but rather than relying on an execution context for its evaluation, the assertion may be evaluated continuously or at specific time points (clocked). Assertions may be simple expressions containing Boolean, relational, and arithmetic operators; or they can be complex sequences specifying temporal operators. These temporal logic sequences describe *properties* that are most useful for describing communication protocols. During the 1990s tools were developed that could formally check such properties, given a gate-level HDL description [Fix 2008]. An important aspect of formal property verification tools is that their proofs are exhaustive. They do not rely on the simulation of the HDL code, and thus require no testbench.

In 2002, Accellera set up a committee[20] to devise a property language standard. There were four candidates submitted by IBM, Intel, Motorola, and Verisity, and a set of requirements including many example properties. IBM's Sugar [Beer et al. 2001] won the evaluation, and as usual, the committee did not copy its language directly but included features and options from the other candidates. In particular, it allowed its predicate expressions to be written in the underlying hardware description language used, either Verilog or VHDL. The Property Specification Language standard [Accellera 2004a] was published in 2004 and donated to the IEEE, where it became standard 1850-2005 [IEEE 1850-2005], later revised to 1850-2010 [IEEE 1850-2010].

Also in 2002, Synopsys donated a language called OpenVera Assertion [Dudani and Cerny 2003] to Accellera. The technology was driven by Intel's ForSpec [Armoni et al. 2002], which used enhanced LTL (linear temporal logic) [Vardi 1995], rather than the CTL (computation tree logic) of IBM's Sugar.

[20] Authors of this paper Simon Davidmann and Peter Flake were on the committee.

A simple OpenVera Assertion (OVA) example shows a trap for an 8-bit counter overflow:

```
module counter_8bit {      // links to Verilog context containing cnt and clk
   clock negedge (clk) {    // defines clock
    event e_overflow :  (cnt == 8'hff) #1 (cnt == 8'h00);  // simple sequence
   }
   assert a_overflow : forbid(e_overflow,"cnt overflow");
}
```

The #1 indicates the next clock cycle, so the sequence is one Boolean expression followed by another. The assertion is evaluated each clock cycle on the negative edge (the counter is incremented on the positive edge), and the assertion fails and prints the message when the sequence occurs. The `forbid` directive fails when the counter changes from all ones `8'hff` to all zeros `8'h00`.

To facilitate reuse of OpenVera assertions, complete definitions may be encapsulated in templates. These are similar to parameterized macro definitions that can be instantiated in the appropriate scope and passed actual arguments. A generalized template for the overflow check is shown below:

```
template overflow(clk, exp, min, max, msg = "Overflow Failure") : {
   clock clk {
     event e_overflow : (exp >= max) #1 (exp <= min);
   }
   assert c_overflow : forbid(e_overflow, msg);
}
```

A template can then be embedded in the Verilog design using OVA pragmas (structured comments).[21]

```
module counter_8bit(rst, clk, cnt);
  input rst, clk;
  output [7:0] cnt;
  reg    [7:0] counter;
  /* ova overflow(negedge clk, cnt, 8'hff, 8'h00, "cnt overflow"); */
  always @(rst or posedge clk)
    if (rst) counter <= 8'b0;
    else counter <= counter + 1;
  assign cnt = counter;
endmodule
```

Templates can be packaged in libraries that check a particular communications protocol, and these can be supplied as reusable verification blocks.

9 SYSTEMVERILOG

9.1 Toward a Unified Language

In 2002 all the pieces of SystemVerilog were in place. Following the publication by Accellera of the SystemVerilog 3.0 standard in June 2002, Synopsys donated OpenVera and OpenVera Assertion to Accellera as a further enhancement. A few months later, in September 2002, Synopsys acquired Co-Design Automation.

[21] SystemVerilog allowed an even tighter integration between hardware description and assertions that does not need pragmas.

There were benefits in a tighter integration between assertions, hardware description and testbench. This could best be achieved by making property specifications and assertions part of the same design and verification language. The Accellera SystemVerilog Committee, chaired by Vassilios Gerousis, began work on merging the donated languages. There were four subcommittees: Basic/Design, Enhancement, Assertions, and C API.

The Basic subcommittee worked on enhancing the SystemVerilog with features such as packages and separate compilation (from VHDL), as well as tagged unions (a donation from Bluespec [Nikhil and Arvind 2009]).

The Enhancement subcommittee worked on merging Vera into SystemVerilog 3.0. Some features were distinct and therefore it was just a question of agreeing on a syntax that was consistent with SystemVerilog. However, there was a fierce argument over heap memory: explicit deallocation as in Superlog or garbage collection as in Vera? Pointers as in Superlog or handles as in Vera? In other words, C or Java style? Vera won the battle on the grounds of fewer memory bugs and larger user base. Vera syntax was also selected for dynamic process creation: `fork...join none`.

The Assertions subcommittee worked on merging the OpenVera Assertion (OVA) language into SystemVerilog. Syntax changes included changing #1 to ##1 to denote a clock cycle delay, changing `event` to `sequence`, and changing the assertion message to a more general procedural statement. A useful addition was that a Boolean expression in a sequence can include an assignment to a local variable.

The C API subcommittee worked on the Direct Programming Interface (DPI). This new C API had to be implementation-neutral, so it could not easily replicate everything in Superlog, where the implementation data structures matched those of the C compiler. However, it allowed most scalar and static compound data types to be passed as arguments, and it retained Superlog's ability to call tasks in addition to functions.

One of the most difficult discussions was about the scheduling during a time slot [Moorby et al. 2003]. Verilog already had immediate assignments (Active region), `#0` delays (Inactive region), nonblocking assignments (NBA region), and end-of-slot PLI calls (Postponed region). To this it was necessary to add the evaluation of assertions (Observe region) and execution of testbench (Reactive region), as well as a beginning-of-slot PLI call (Preponed region).

The following were the major technology donations that made up SystemVerilog [Sutherland et al. 2006]:

- Superlog Extended Synthesizable Subset from Co-Design Automation
- OpenVera verification language from Synopsys
- PSL assertions (which began as a donation of Sugar assertions from IBM)
- OpenVera Assertion (OVA) language from Synopsys
- DirectC and coverage Application Programming Interfaces (APIs) from Synopsys
- Separate compilation and `$readmem` extensions from Mentor Graphics
- Tagged unions and high-level language features from Bluespec

By 2004 Accellera had produced two new versions of the standard, SystemVerilog 3.1 [Accellera 2003] and 3.1a [Accellera 2004b], and the SystemVerilog language assumed more or less its present form.

Accellera donated the 3.1a version to IEEE, and many members of the Accellera subcommittees became members of the IEEE Verilog and SystemVerilog working groups. Each of these working groups produced a standard in 2005: a Verilog standard [IEEE 1364-2005], and a SystemVerilog standard [IEEE 1800-2005] that defined only enhancements compatible with Verilog. IEEE then undertook a major, four-year task to merge these two standards into a single SystemVerilog standard

[IEEE 1800-2009]. Subsequently, IEEE published two SystemVerilog standards, one in 2012 [IEEE 1800-2012] and one in 2017 [IEEE 1800-2017].

9.2 Universal Verification Methodology (UVM)

SystemVerilog is able to model digital hardware at various levels of abstraction; moreover, it also aims to provide a complete object-oriented programming framework that includes domain-specific features to support digital hardware verification. Notable among such features are constrained random generation, temporal assertions, and functional coverage constructs. Notwithstanding the richness of the SystemVerilog language, early adopters realized that the language alone was insufficient to enable widespread adoption of the best-practice verification techniques that inspired its development. While the language provides all the constructs that skilled developers need to implement complex verification environments, it does not guide developers on how to build reusable, collaborative, layered testbenches needed to verify complex contemporary systems. These cultural, technical, and practical challenges limited the utility of the language and encouraged development of additional libraries, toolkits, and methodology guides.

The Universal Verification Methodology (UVM) aims to provide a comprehensive framework for the creation of sophisticated SystemVerilog functional verification environments that encourage the development and deployment of re-usable verification components. UVM provides a set of prescriptive guidelines and a comprehensive SystemVerilog class library that enforces its conventions, and supports the development and deployment of reusable verification components. UVM improves engineering productivity by reusing code common to all environments, providing a consistent look and feel across multiple projects, and facilitating the deployment of pre-built Verification Intellectual Property (VIP) components.

UVM is the result of a multi-year evolution (see Fig. 18). The Verification Methodology Manual (VMM) [Bergeron et al. 2005] published in 2005 was the first manual to provide a comprehensive verification methodology and a SystemVerilog class library. VMM was based on the earlier Reference

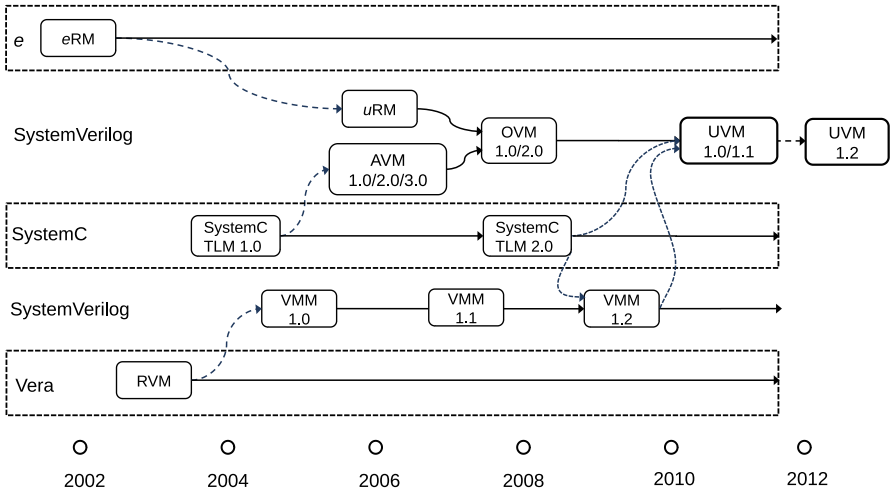


Fig. 18. Family tree for UVM.

Verification Methodology (RVM) developed in 2002 for the Vera verification language (see Section 7). VMM prescribed a specific architecture for organizing a testbench in SystemVerilog, plus a complete base class library supporting that architecture. VMM's class library was originally vendor-proprietary but was eventually released as open-source. Shortly after the release of VMM, another company published a competing verification methodology guide for SystemVerilog: Advanced Verification Methodology (AVM) [Mentor Graphics 2006]. The AVM class library was released as open-source from the outset. 2008 saw the release of yet another open-source SystemVerilog methodology: Open Verification Methodology (OVM) [Cadence and Mentor Graphics 2008]. The OVM methodology and class library was jointly created by two companies merging their guides: AVM and the Universal Reuse Methodology (URM) [Stellfox 2010] released in 2006; URM was a SystemVerilog version of the *e* Reuse Methodology (eRM) [Verisity Design 2002] developed in 2001 for the *e* Verification Language [Iman and Joshi 2004]. The plethora of SystemVerilog methodology guides and inherent incompatibilities led to the creation of an Accellera committee charged with the creation of a SystemVerilog methodology that would become an industry standard [Accellera 2011]. The committee began by combining the remaining two SystemVerilog methodologies, VMM and OVM, into one "unified" verification methodology or UVM—eventually the committee changed the name to "Universal Verification Methodology". In 2017 UVM became an IEEE standard [IEEE 1800.2-2017].

The SystemVerilog verification methodologies, including UVM, were motivated by several factors. First, earlier implementations of SystemVerilog supported different subsets of the language. Consequently, users worried that software developed for one implementation would not work with other implementations. Second, SystemVerilog's powerful domain-specific features enable the creation of sophisticated functional verification environments; however, the language alone does not prescribe best practices that promote reuse across projects or organizations. Third, digital hardware connects to its environment through interfaces that are often standard protocols (such as Ethernet or USB). However, creating the VIP containing verification infrastructure (such as stimulus generation and protocol validation) for any one interface requires significant effort. Consequently, there is a demand for pre-built, reusable VIPs that model particular interfaces and are readily connected to the hardware design. For such a VIP-based environment to work correctly, all VIPs must coexist and interoperate; this requires VIPs to adhere to a set of conventions and provide a consistent use-model. Finally, all verification projects present a recurring set of challenges; hence, valuable time and effort are saved by reusing code common to all environments. This is achieved with a software library that provides verification facilities such as error reporting and communications handshaking.

UVM's prescribed methodology along with its object-oriented class library impacted SystemVerilog and its use for verification in several important ways.

- The common interconnect mechanism created an ecosystem that enabled a marketplace for interoperable VIPs. The ensuing reuse enables larger verification subsystems to migrate across projects.
- The prescriptive architectural conventions and documentation of best-practices help guide engineers, thus, saving time and effort.
- The existence of working techniques provided the foundation for more sophisticated projects.
- The class library itself demonstrated that existing SystemVerilog implementations were complete and powerful enough to address the existing verification challenges.

The UVM library defines a set of base classes whose behavior can be customized by extending or overriding their functionality using object-oriented techniques. The class library is organized as a hierarchy; all classes are derived from a root class, `uvm_object`. This arrangement provides

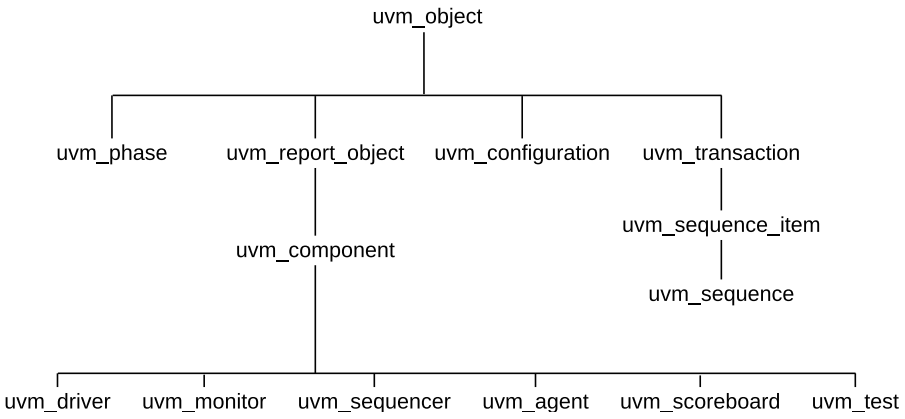


Fig. 19. Partial UVM class hierarchy.

a common framework for key facilities such as printing, comparing, and managing resources. Figure 19 shows the class hierarchy of some of the key UVM classes.

There are two categories of classes in the UVM library. The first category is derived from `uvm_component` and designates common verification components such as drivers and monitors. The second category is derived from `uvm_transaction` and defines the data objects that are manipulated by the verification components.

The UVM component classes correspond to various activities that a verification environment or testbench performs to interact with a hardware design and verify its functionality. Producing stimulus for the hardware requires a block that generates sequences of bits to be transmitted to its interface: a UVM *sequencer*. Normally, sequencers are unaware of the hardware's communication mechanism; their only responsibility is to produce generic sequences of data. A different component handles the communication with the hardware: a UVM *driver*. The driver directs the hardware's activity by feeding it data generated by sequencers, but it does not validate the responses to its stimuli. A different block oversees the hardware-driver communication and evaluates the responses from the hardware: a UVM *monitor*. A monitor samples the data sent to and received from the hardware, predicts the expected result, and sends the actual response along with the prediction to another block for comparison and evaluation: a UVM *scoreboard*. All these UVM components constitute a typical verification test (see Fig. 20).

A very important feature of UVM is "phasing." The `uvm_phase` class enforces a set of phase conventions that define the life-cycle of verification components. Each prescribed phase is automatically launched by UVM, triggering a call to the corresponding phase callback (virtual method) in each verification component object. Users merely override the phase callbacks to customize the functionality of a verification component. This class thus imposes an ordered set of steps (or phases) that all components follow; they include phases such as build, connect, run, and report.

The UVM library includes several other classes to manage the creation, synchronization, and configuration of components as well as classes to manage policies, recordings, and reports. It also includes classes that model important abstractions such as a Register Abstraction Layer (RAL) and a Transaction Level Modeling (TLM) mechanism. The Register Abstraction Layer provides an abstracted view of the actual registers and memories in the hardware. The specific protocols for accessing hardware registers can be complex and varied; RAL simplifies this access by abstracting all the detail so that for example applying stimulus to the abstract registers will cause the actual

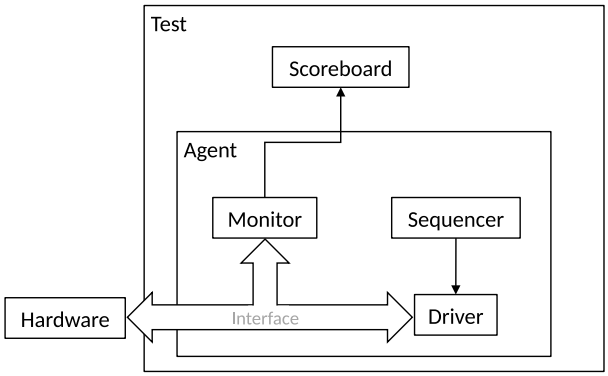


Fig. 20. Typical UVM testbench organization.

hardware registers to exhibit the changes applied by the stimulus. The TLM layer enables a highly abstract mechanism for modelling data transfers across the hardware as transactions. The TLM scheme represents data as transactions that flow in and out of different components via special ports called TLM interfaces. Transactions significantly raise the level of abstraction for verification environments that often struggle to manage the large number of individual bits associated with modern protocols. UVM's transaction-level interfaces enable different components to communicate by transferring data packets, which provides component isolation and promotes flexibility and reuse.

UVM is one of the most effective catalysts for the widespread adoption of SystemVerilog. Indeed, UVM has become so prevalent that for many engineers "verification with SystemVerilog" and "verification with UVM" are broadly synonymous. UVM is now supported by major vendors, maintained by an IEEE industry standard and promoted by numerous publications [Hunter 2016; Salemi 2013; Vasudevan 2016]. Presently, many companies leverage the UVM infrastructure to provide tools for linting or debug as well as broad portfolios of VIPs covering numerous standard protocols. UVM is now used to verify over 70% of design projects [Foster 2019].

10 OTHER HARDWARE LANGUAGES

10.1 Other HDLs

The US Department of Defense sponsored the development of the VHSIC Hardware Description Language (VHDL) for modeling digital components in the Very High Speed Integrated Circuit (VHSIC) project. This language was donated to the IEEE in 1986, and the first IEEE standard issued in 1987 [IEEE 1076-1987].

The design of VHDL was inspired by Ada, and VHDL shares Ada's verbosity and strict type checking. VHDL has three phases of implementation: analysis, elaboration (building the code/data structure) and execution (simulation). It is a complex language, and the first commercial implementations focused on language subsets. Many involved translation to proprietary languages, and the results were often unsatisfactory due to differences in semantics. The original demonstration simulator was too slow and memory intensive for general use, and it became widely used only after Model Technology released a low-cost full-language simulator, and Synopsys released a synthesis tool.

Unlike HILO and Verilog, the wire values and gate types are not built into the VHDL language, but must be specified in libraries. VHDL users commonly model hardware types via a multi-valued logic package known as `std_logic` [IEEE 1164-1993]. This package uses the following nine-valued logic:

Symbol	Meaning	Symbol	Meaning
U	uninitialized	W	weak drive, unknown logic
X	strong drive, unknown logic	L	weak drive, logic zero
0	strong drive, logic zero	H	weak drive, logic one
1	strong drive, logic one	-	don't care
Z	high impedance		

In contrast to Verilog's four-valued logic, VHDL's multi-valued logic uses three different values to represent nondeterminism: U, W, and X. Also, while Verilog keeps the (drive) strength separate from the logic value, `std_logic` uses different logic values to model strength (weak and strong). The final value (`-`) is not a modeling logic value, but it is reserved as a hint for synthesis optimizations (see Section 4.11).

The lack of standard libraries in the early days inhibited portability between tools. Nevertheless, it was possible to use VHDL for pre-synthesis simulation (design and testbench), capture of simulation traces, and use a different simulator for the post-synthesis netlist driven by the captured traces. Eventually EDA vendors developed mixed-language simulators for Verilog and VHDL, so that components described in VHDL could be embedded in Verilog designs and vice-versa. Similarly the design could be written in one language and the testbench in another. The fact that signal types and operator types were in libraries made performance optimization harder than Verilog, as did the VHDL (two list) simulation algorithm. Simulation performance was still important because design sizes grew as fast as computing power, and runs could take many hours.

In 1999, a new HDL called SystemC [IEEE 1666-2011] was launched. Rather than a language, it is a dialect of C++ that uses class libraries, templates, and macros. All hardware objects are coded as classes, and their connections are coded as an object constructor. Consequently, connection errors become run-time errors and may crash. This is in contrast to Verilog and VHDL, where there is an elaboration phase that builds the connections, and errors lead to meaningful error messages. For writing the testbench, SystemC provides methods that can "wait" a simulation time or an event. SystemC has had limited success, even though it has an open source simulator and some SystemC synthesis tools have been developed.

Verilog-AMS is a dialect of Verilog that provides extensions for modeling analog and mixed-signal systems. Most notably, it extends Verilog's event-based simulator with a time-continuous simulator capable of solving analog equations. Verilog-AMS was first standardized in 1996 by OVI under the name Verilog-A, Analog Extensions to Verilog HDL. This standard added basic time-continuous constructs needed to describe analog circuitry and, despite its name, was not intended to work with Verilog—it only used syntax similar to Verilog. In 1998 Verilog-A evolved into Verilog-AMS, and since 2000 the standard has been maintained by Accellera. The last version of Verilog-AMS [Accellera 2014] released in 2014 includes constructs that couple the discrete-time (digital) and the continuous-time (analog) domains: digital events can trigger actions in the analog domain and vice-versa. The Verilog-AMS committee intended to create a single Verilog language for analog and digital design, but this has not yet happened: Verilog evolved into SystemVerilog in the IEEE while Verilog-AMS remains a separate Accellera standard incompatible with SystemVerilog.

Bluespec, Inc., a company founded in 2003, developed an HDL based on the Haskell language. Subsequently the syntax was changed to have a Verilog style, and the language is called Bluespec

SystemVerilog (BSV). Bluespec introduced several novel features intended to reduce errors and encourage re-usability. A notable innovation is the interface construct used to specify interconnects. BSV interfaces are composed of methods (which act as inputs or outputs) and other sub-interfaces which may contain their own methods and sub-interfaces. Another key feature of the language is the formalization of the timing semantics that enforces correct access to any signal. For example, the language enforces that a data bus is read only when the "ready" signal is active. Bluespec provides a synthesis engine as well as a simulator, but it has not been widely adopted possibly due to designer's unfamiliarity with its functional programming model. A common complaint is that their simulator supports only the BSV language and cannot directly simulate mixed-language designs. A related complaint is that BSV's capability for interfacing to other HDLs such as VHDL or Verilog is extremely limited.

In 2012 UC Berkeley released Chisel, a hardware description dialect based on the Scala language. Chisel is not an HDL per se, but rather a hardware generator whose input is a structural view of the design. Like SystemC, it uses class constructors to provide highly parameterizable hardware descriptions. The Chisel3 Scala library generates an intermediate representation called FIRRTL that allows designers to programmatically modify and optimize the structure and organization of the design. After the design is generated, a converter turns FIRRTL into Verilog for use with standard synthesis and simulation tools. Currently, Chisel provides no support for writing testbenches; they must be written in another language like SystemVerilog. Debugging a Chisel simulation is harder because engineers work on the generator output, not the source code. Generally, Chisel has been successfully used to create many variants of the same type of design, such as the RISC-V project that produces different CPUs for the same instruction set architecture (ISA).

The X-propagation problems described in Section 4.11 have persisted for years, but in the last few years several tools have introduced novel simulation semantics that eliminate both X-pessimism and X-optimism. These tools are not languages per se, nevertheless, they address real problems and complement the semantics of existing HDLs. The SimXACT [Chang and Browy 2012] tool runs in tandem with a conventional gate-level simulator and monitors the results; when it detects an X, it uses a symbolic simulator to examine the logic and determine if the X is due to X-pessimism—if it is not, the Boolean value computed by the symbolic simulation is propagated. SimXACT is most useful to eliminate pessimistic Xs due to reconvergent signals as in Fig. 9. VCS Xprop [Salz et al. 2012] is a full-fledged SystemVerilog simulator that eliminates X-optimism by emulating the don't-care synthesis semantics. Xprop considers the effect of all possible Boolean values for every X-controlled assignment and then merges all possible outcomes into a single result. If the merging finds a contradictory outcome, the result becomes X. VCS Xprop also provides a mechanism for users to configure the degree of pessimism from three possible levels.

10.2 Other HVLs

In addition to Vera, other hardware verification languages appeared in the mid to late 1990s. Verisity, a company founded in 1995, developed the HVL *e* [Iman and Joshi 2004] for its Specman software system. In 2005 Verisity was acquired by Cadence, and in 2006 the *e* language became an IEEE standard [IEEE 1647-2006]. Like Vera, the *e* language includes first-order constructs to generate random and constrained random stimulus, constructs for defining and collecting functional coverage, and a temporal language that can be used to write assertions. Vera and *e* became the main contemporaneous competing verification languages at the time, hence, they both influenced one another. Due to this conceptual cross-pollination between Vera and *e*, the two languages are conceptually similar—and it is the reason Fig. 1 shows their relationship using a bidirectional arrow. However, the two languages do differ in the underlying framework used to achieve code reusability: Vera proposed a strict object-oriented programming (OOP) framework in which inheritance is

the only class extension mechanism whereas *e* advocated an aspect-oriented programming (AOP) framework in which extension is done via aspects. AOP [Kiczales et al. 1997] allows users to insert additional functionality to existing code in a non-invasive manner. SystemVerilog incorporates Vera's OOP approach—a subsequent proposal to add AOP extensions to SystemVerilog was rejected by the IEEE committee.

In 2000, Cadence released TestBuilder [Cadence 2000], an open-source testbench library written in C++. TestBuilder is a C++ class library that provides testbench-authoring capabilities similar to those available in an HVL. TestBuilder included classes to create reusable random and random constraint-driven tests as well as checker classes to write self-checking tests. Despite being open source, TestBuilder had very limited success and was eventually discontinued.

11 CURRENT & FUTURE

SystemVerilog is really three languages in one: Verilog, the Superlog extensions, and Vera. It is difficult if not impossible for any one engineer to be fluent in the complete language. Yet the simple syntax and ease of use that has attracted hardware designers since 1985 is still there—many hardware designers rely on the (very powerful) synthesizable subset of SystemVerilog, never using the sophisticated features that are so attractive to verification engineers [Sutherland 2017]. As of 2018 about 80% of integrated circuit designs worldwide use Verilog and SystemVerilog [Foster 2019]. Use of classic Verilog is declining, while use of SystemVerilog is increasing.

For IC verification SystemVerilog is significantly increasing its market share, and has now been adopted by close to 80% of all design projects [Foster 2019]. It appears that SystemVerilog is being used for testbench development even when the RTL being tested is a non-Verilog language, for example VHDL. Perhaps as teams become more familiar using SystemVerilog for verification, they will begin using it for their RTL design as well.

A further indication of SystemVerilog's strength in verification is the acceptance of UVM. Of all the various methodologies and class libraries used for verification, only UVM continues to grow in market share and is now used by over 70% of design projects [Foster 2019].

Research continues into new HDLs, both in academia and industry. Invariably though, these experimental tools always output Verilog (or SystemVerilog) as an intermediate language,[22] thus giving the researchers easy access to the huge and growing community of SystemVerilog tools and users.

SystemVerilog, with its roots stretching back over 40 years, is the predominant language for large-scale digital hardware design and verification, today and into the forseeable future.

ACKNOWLEDGMENTS

We acknowledge the help of many people, too numerous to list, in developing and promoting the languages described here.

[22] For example, BSV and Chisel (see Section 10.1). Initially, Co-Design provided a translation tool that converted Superlog into synthesizable Verilog.

A TIMELINE

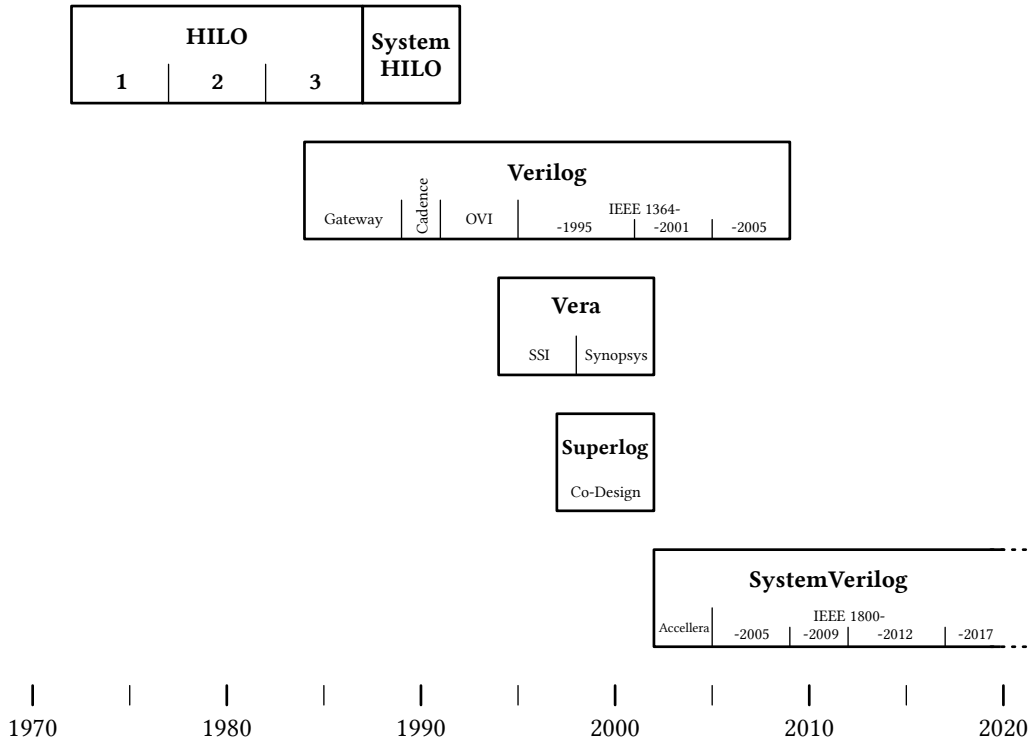


Fig. 21. Overall timeline for Verilog and related languages.

1972
Start of HILO 1 at University of Bradford.

1973
HILO project moves to Brunel University.

1974
First HILO paper presented.

1975
Phil Moorby joins HILO project at Brunel.

1976
End of HILO 1 development, start of HILO 2.

1979
US Department of Defense begins its Very High Speed Integrated Circuit (VHSIC) program. Discussion begins on VHSIC Hardware Description Language (VHDL).

1981
First HILO 2 paper.
First workshop held on VHDL.

1982
HILO 2 project moves to Cirrus Computers. Office remains at Brunel.
First HILO 2 customer.
Start of HITEST project at Cirrus Computers office in Fareham.
Simon Davidmann moves to USA to support HILO 2 early customers (including first US customer Wang).
Start of HILO 3 project. Continued maintenance of HILO 2.

1983
Peter Flake leaves Cirrus.
Phil Moorby leaves Cirrus.
GenRad acquires Cirrus.

Winter of 1983–1984
Phil Moorby joins Gateway Design Automation and defines first version of the Verilog language.

1984
GenRad Santa Clara office sets up HILO marketing, sales, and support.
Peter Flake joins GenRad.

1985
HILO R&D moves to Fareham.
Spring Verilog appears in the Sun Catalyst Program catalog. This is the first public mention of Verilog.
Spring First full Verilog language simulator released to customers by Gateway.
Phil Moorby begins development work on Verilog-XL.

1987
Verilog-XL, a full Verilog simulator featuring very fast gate-level simulation times, released to customers by Gateway.
VHDL is standardized as IEEE 1076-1987.
Launch of System HILO and HITEST.

1988
Synopsys adopts Verilog language for its early Logic Compiler synthesis tool, and later for its Design Compiler synthesis tool.
Simon Davidmann joins Gateway to drive Verilog adoption in Europe.

1989
October Gateway announces acquisition by Cadence.

1990
Peter Flake moves from GenRad to Cadence.
Open Verilog International (OVI) formed. Cadence releases the Verilog language (but not their simulator) to OVI.
December Publication of Thomas and Moorby's book *The Verilog Hardware Description Language*.

1991
VHDL International (VI) founded. Begins holding annual VHDL International Users Forum (VIUF).
November First version of Verilog LRM released by OVI language committee.

1992
OVI begins working toward IEEE standardization of Verilog.
OVI sponsors first International Verilog Conference (IVC).

1993
Chronologic releases its Verilog Compiled Simulator (VCS).
Simon Davidmann joins Chronologic to open market for VCS in Europe.

1994
Chronologic acquired by Viewlogic.

1995
March Initial version of the Vera language.
April First release of Vera (version 0.3) internal to Sun Microsystems.
April Joint review of Vera language by Sun Microsystems and Systems Science, Inc. (SSI).
May Vera is transferred to SSI.
June First official Vera release by SSI.
December Verilog standardized as IEEE 1364-1995. This version is referred to as either Verilog 1.0 or Verilog-95.
Simon Davidmann and Peter Flake start collaborating on evolving Verilog HDL.

1996
Two user language conferences IVC and VIUF are co-located.
Cadence rewrites their Verilog simulator and calls it NC-Verilog.

1997
Synopsys acquires Viewlogic and the VCS Verilog simulator.
Co-Design Automation, Inc. registered in California, and Co-Design Automation Ltd, registered in England, founded by Simon Davidmann and Peter Flake. Business plan, sales pitch, and early language specification for Superlog.

1998
IVC and VIUF hold a joint conference.
June Co-Design raises $600k in seed capital.
August Synopsys acquires SSI.

1999
Co-Design raises $1.2M in series B round.
The VIUF and IVC joint language conference is renamed HDL Conference and Exhibition (HDLCon).
Phil Moorby joins Co-Design.

2000
February Two language groups OVI and VI merge to form Accellera.
Co-Design holds series C round.
Co-Design first customers.
Superlog presented at ASP-DAC.

2001
GenRad acquired by Teradyne.
Synopsys creates www.open-vera.com.
Co-Design holds series D round.
Mentor releases its ModelSim Verilog simulator.
May Synopsys launches the OpenVera initiative that makes the Vera LRM an open source standard.
June Co-Design donates design subset of Superlog to Accellera.
September Updated Verilog standard IEEE 1364-2001. This version is referred to as either Verilog 2.0 or Verilog-2001.
November Synopsys integrates Vera and VCS.

2002
June Accellera standardizes SystemVerilog 3.0 based on Co-Design donation.
June Synopsys donates Vera to Accellera.
September Co-Design acquired by Synopsys.
Synopsys donates OpenVera Assertion (OVA) language to Accellera.

2003
HDLCon renamed to Design and Verification Conference (DVCon).
May Accellera standardizes SystemVerilog 3.1.

2004
May Accellera standardizes SystemVerilog 3.1a and in June donates it to IEEE.

2005
November IEEE 1800-2005 standardizes SystemVerilog as a superset of Verilog. This version is referred to as SystemVerilog-2005.
November Phil Moorby receives the Phil Kaufman Award presented by the Electronic Design Automation (EDA) Consortium.

2006
April IEEE 1364-2005 updates Verilog standard. This version is called Verilog-2005.

2009
December IEEE standard 1800-2009 merges the Verilog and SystemVerilog standards. This version is referred to as SystemVerilog-2009.

2013
February IEEE standard 1800-2012 released. This version is referred to as SystemVerilog-2012.

2016
April Phil Moorby receives a Fellow Award from the Computer History Museum.

2017
December IEEE standard 1800-2017 released. This version is referred to as SystemVerilog-2017.

B PEOPLE

The following is a list of people mentioned in this paper. By no means is this a comprehensive list of everyone involved in the development of Verilog and SystemVerilog.

Bechtolsheim, Andy Co-founder of Sun Microsystems and developer of the Sun workstations. Engineering VP at Cisco and founder (chairman) of Arista. In 1988 provided Google founders, Sergey Brin and Larry Page, with their first round of funding. Lead investor in seed funding round for Co-Design Automation.

Brophy, Dennis Chairman of the Accellera board during the Superlog donation and its standardization as SystemVerilog. Helped drive SystemVerilog's evolution and continued success.

Burisch, Christian Application engineer at Co-Design, then Synopsys. Worked on Superlog and created training material.

Davenport, Rich Sales director at Gateway. Seed round investor in Co-Design. Founder and CEO of Simulation Technologies Corp.

Davidmann, Simon Involved in the Verilog evolution from HILO to SystemVerilog. Worked on HILO 2 as a Fellow at Brunel University and at Cirrus. As application manager at Cirrus-USA supported early HILO customers including Gateway founders, Prabhu Goel and Chi-lai Huang, at Wang Labs. Drove European adoption of Verilog as Technical Manager at Gateway and later as European VP at Chronologic. Founder and CEO of Co-Design. Drove standardization of SystemVerilog as VP at Synopsys. Founder and CEO of Imperas.

Fitzpatrick, Tom Technical marketing manager at Co-Design responsible for Superlog and SystemVerilog. Verification evangelist at Mentor Graphics driving verification methodologies based on SystemVerilog. Significant contributor to several industry standards, including Verilog (1364), SystemVerilog (1800) and UVM (1800.2).

Flake, Peter Researcher at Bradford University, then Brunel University, Technical Manager at Cirrus Computers, then Director of Technology at GenRad. Architect at Cadence, Chief Technical Officer at Co-Design Automation, Scientist at Synopsys. Worked on all HILO projects, Superlog and SystemVerilog.

Gerousis, Vassilios Early adopter of Gateway's Verilog-XL and Co-Design's Superlog. Chairman of Accellera's technical committee during the standardization of SystemVerilog.

Goel, Prabhu Researcher, entrepreneur and businessman. In 1973, he joined IBM's EDA organization and developed the PODEM algorithm. In 1981 he joined Wang Labs where he was exposed to HILO. In 1982 he founded Gateway Design Automation which developed Verilog. Later became a private venture capitalist.

Golson, Steve Early user of Verilog and Design Compiler. Consultant in IC design and flow development. Co-designer of *Ms. Pac-Man* arcade video game.

Hall, Matthew Developer of HILO-based test product at GenRad. Developer of Superlog parser at Co-Design, then Synopsys.

Harding, Martin Manager of ASIC Business Group at Gateway, where he made Verilog-XL the de-facto standard sign-off simulator with many ASIC vendors.

Harris, Robert "Bob" Research Fellow at Brunel University, then engineer at Cirrus, then GenRad. Worked on HILO 2 and 3.

Huang, Chi-lai Synthesis expert. Collaborator on first version of Verilog language at Gateway.

Kasuya, Atsushi Verification engineer at Sun Microsystems in the 1990s where he worked on the Sunfire SMP server. Conceived and co-developed the first version of the Vera HVL.

Kelf, Dave VP marketing at Co-Design, then Director at Synopsys. Supported Verilog as an application engineer at Cadence, then worked on Superlog and SystemVerilog.

Kenney, James Co-founder of Co-Design Automation. Developed Superlog interpreter at Co-Design, then Synopsys.

Madhavan, Rajeev Seed round investor in Co-Design. Co-founder of multiple EDA companies including LogicVision (acquired by Mentor Graphics), Ambit Design Systems (acquired by Cadence), and Magma Design Automation (acquired by Synopsys).

Moorby, Philip Inventor of Verilog HDL. Inventor of Verilog-XL simulator. In 2005 received the Phil Kaufman Award [Aycinena 2005; EDAC 2005; Goering 2005; Newton 2005] presented by the EDA Consortium (now the ESD Alliance) for "creat[ing] and help[ing] to popularize the Verilog Hardware Description Language." In 2016 received a Fellow Award [CHM 2015, 2016] from the Computer History Museum: "For his invention and promotion of the Verilog hardware description language."

Moore, Lee Application engineer at Co-Design, then Synopsys. Worked on Superlog then SystemVerilog. Developed regression test system.

Musgrave, Gerry Professor at University of Bradford, UK then at Brunel University, West London. Director at Cirrus. Supervised HILO 1 and 2.

Rich, Dave Application engineer, first at Gateway supporting Verilog, then at Co-Design where he drove the evolution of Superlog into SystemVerilog. Persuaded Co-Design to make Superlog a strict superset of Verilog. At Mentor's Verification Academy supports SystemVerilog. Co-Chair of the Technical Champions committee of the SystemVerilog IEEE Working Group. Key contributor to AVM and OVM.

Salz, Arturo Researcher at Stanford University and developer of the IRSIM timing simulator. Co-founder of Systems Science where he developed the SimWave debug tool and oversaw the development of parts of Vera. As a Scientist at Synopsys he marshalled the standardization of Vera's verification features into SystemVerilog, and later developed the VCS Xprop simulator. Currently a Synopsys Fellow active in several IEEE standards committees including SystemVerilog (1800), UPF (1801), and UVM (1800.2).

Sanguinetti, John Verilog verification specialist at several computer manufacturers: DEC, Amdahl, ELXSI, Ardent, and NeXT. Founder of Chronologic Simulation in 1991. He was the Principal Architect of VCS. Early investor in Co-Design.

Shorland, Mike Research student at Brunel University. Worked on HILO 1.

Shukla, Venk Strategic marketing director at Cadence who initiated the formation of OVI to open up the Verilog language. Early investor in Co-Design.

Superlog Working Group IEEE Verilog contributors and early supporters of Superlog and the concept of evolving Verilog: Stuart Sutherland, Cliff Cummings, Stefen Boyd, Mike McNamara, Anders Nordstrom, Bob Beckwith, Kurt Baty, Warren Stapleton, Phil Moorby, and Don Thomas.

Thomas, Don Professor at CMU. Early pioneer in HDL methodologies. Wrote the first book on Verilog with Phil Moorby.

White, Ian PhD student at University of Bradford, then engineer at Smiths Industries. Worked on HILO 1 simulator.

Wilson, Richard Research Fellow at Brunel University. Worked on HILO 2.

Zhang, Eugene Verification engineer at Sun Microsystems in the 1990s where he worked on the the Sunfire SMP server. Co-developed the first version of the Vera HVL.

C GLOSSARY

Accellera A standards organization that supports standards and open interfaces for the EDA and integrated circuit industries. Formed by a merger of OVI and VI in 2000.

active low A signal which is active (asserted) when it has a low voltage or low value (0).

active high A signal which is active (asserted) when it has a high voltage or high value (1).

AMS Analog and mixed-signal

annotation Applying timing information and constraints to a netlist, to enable timing-accurate simulation. *see* **back-annotation**

ASIC Application-specific integrated circuit

assertion A predicate that must always be true. An invariant property of a system.

asynchronous An **asynchronous signal** has no timing relationship with a given clock, and may transition at any time in the clock period. An **asynchronous circuit** is an unclocked circuit element.

ATPG Automatic test pattern generation

back-annotation Annotation of a netlist using post-route timing values, rather than pre-route estimated timing values. This enables accurate sign-off simulation. *see* **annotation**

behavioral description Level of abstraction used by high-level programming languages to describe systems from the algorithm level to the functional level. Often implemented using procedural or functional languages.

Bluespec, Inc. Company founded in 2003 by Arvind Mithal. Developed a Haskell-based HDL called Bluespec, later renamed Bluespec SystemVerilog (BSV). In 2003 donated tagged unions and pattern matching technology to Accellera, which incorporated the features into SystemVerilog 3.1a.

BSV Bluespec SystemVerilog

Cadence Design Systems A major EDA company headquartered in San Jose, California. Created by the merger of two earlier companies, Solomon Design Automation and ECAD. In 1989 it acquired Gateway Design Automation, makers of Verilog.

Catalyst Program A co-marketing mechanism created by Sun Microsystems to partner with independent software developers, value-added resellers, and systems integrators in order to jointly develop and market products.

cell library A collection of pre-designed gates that implement elemental logic functions such as AND, OR and flip-flops.

CHDL International Conference on Computer Hardware Description Languages. A series of thirteen conferences held between 1973 and 1997 [Hartenstein 2018]. The first conference was called *Workshop on Computer Description Languages* but is now commonly referred to as CHDL '73.

Chronologic Simulation Company founded in 1991 to apply compiled code techniques to simulation. The first release of its VCS simulator was roughly 10 times faster than existing simulators and quickly became the performance leader. Acquired by Viewlogic Systems in 1994, which was later acquired by Synopsys in 1997.

Cirrus Computers Ltd. Small company in Fareham, UK, writing test programs for GenRad testers. Founded by ex-ICL people.

clock A periodic synchronization signal.

Co-Design Automation Company founded in 1997 by Simon Davidmann and Peter Flake to develop Superlog, a new combined hardware description and verification language. Superlog was donated to Accellera in 2001 and it evolved into SystemVerilog. Acquired by Synopsys in 2002.

combinational logic A hardware block that implements a Boolean function whose output is a pure function of its present inputs. This is sometimes referred to as time-independent logic. A circuit with no clock.

continuous assignment A statement of the form *variable = expression* where the expression is evaluated and the variable updated whenever an operand of the expression changes.

continuously driven A signal that is continuously updated by its logic function. *see* **continuous assignment**

DAC Design Automation Conference. Sponsored annually by ACM and IEEE, DAC is the premier conference for design and automation of electronic systems.

DDL Digital system Design Language [Duley and Dietmeyer 1968]

design kit A set of design rules, cell libraries, simulation models, scripts, and documentation provided by a semiconductor foundry to their design customers. *also* **process design kit**, **PDK**

Design Compiler First commercially successful logic synthesis tool to compile Verilog into a netlist. Developed by Synopsys in 1987 and grew to dominate the RTL synthesis market.

DFT Design for test, or design for testability.

DPI Direct Programming Interface

DRC Design rule checking

DUT Design under test

DVCon Design and Verification Conference. Sponsored annually by Accellera since 2003, DVCon is a leading conference covering languages, tools, and intellectual property for the design and verification of electronic systems.

e *e* Hardware Verification Language, IEEE Std 1647-2006.

EDA Electronic design automation

edge sensitive Dependent on a change or transition in input value.

EDIF Electronic Design Interchange Format

EE Times, Electronic Engineering Times Major trade publication for the electronics industry. During the 1980s–90s published as a weekly newspaper.

elaboration The process of building and connecting the complete design prior to simulation or synthesis. Commonly this requires software to instantiate modules, build the hierarchy, compute parameters, resolve hierarchical references, and establish the connectivity.

equivalence checker A verification tool that formally proves whether two representations of a design exhibit exactly the same behavior. Typically used to compare an RTL description to its synthesized netlist.

event queue A time-ordered set of events. Pending events are organized as a priority queue, sorted by time such that events are processed in strict chronological order regardless of the order in which events are added to the queue.

fab Fabrication of an IC, also short for **semiconductor foundry**.

fault conditions A system tested by fault injection techniques on the actual device or its simulated model.

fault injection A technique that introduces faults into a design in order to measure the effectiveness of the test environment or to improve test coverage. Typically used with stress testing for robustness of the design.

fault simulation A logic simulation that models the effect of one or more hardware faults.

flip-flop A clocked circuit that stores a bit. *see* **gate**

flow All the files, tools, and infrastructure required to implement an IC design methodology.

foundry *see* **semiconductor foundry**

four-valued Boolean representation extended to four values. In Verilog the values are: 1 (true), 0 (false), Z (high-impedance) and X (unknown).

FSM Finite state machine

gate An elemental circuit that performs a Boolean function such as AND, OR, NOT. Also, an elemental circuit to store data. *see* **flip-flop**

gate array A silicon chip with a collection of devices (transistors) having no predetermined function. Using custom metal layers, the devices can be arranged as specific logic gates that are then further interconnected into a complete circuit. Gate arrays are a type of ASIC.

gate-level representation A netlist of gates.

Gateway Design Automation Company founded by Prabhu Goel in 1982 (originally called Automated Integrated Design Systems) where Phil Moorby designed the Verilog HDL. Gateway Design Automation grew rapidly with the success of the Verilog-XL simulator. It was acquired by Cadence Design Systems in 1989.

GDS, GDSII Graphic Database System. An IC layout format.

GenRad, Inc. Manufacturer of automatic test equipment (ATE) based in Concord, Massachusetts, but with facilities in Santa Clara, California. Later acquired by Teradyne.

GHDL GenRad HDL. Language used by the HILO 3 simulator.

golden simulator A reference simulator that is certified by a semiconductor foundry for sign-off verification. Sometimes capitalized as Golden Simulator. *see* **sign-off simulator**

hardware accelerator Specialized hardware able to perform a function more efficiently than is possible in software running on a general-purpose computer. For IC designers, this usually means accelerating gate-level simulation.

hazard A race condition that may result in different behaviors.

HDL Hardware description language

HDLCon International HDL Conference. Held between 1999–2002 and then renamed DVCon.

HDVL Hardware design and verification language

high impedance The state of an undriven output signal. An open circuit.

HVL Hardware verification language

IC Integrated circuit. A silicon chip.

ICL International Computers Limited, a British computer company later acquired by Fujitsu.

Imperas Company that develops virtual platforms for RISC-V verification using SystemVerilog.

inertial delay The time that a signal must persist at an input of a device in order for a change to appear at an output.

inference Recognition by synthesis of various design elements such as flip-flops, latches, or combinational logic.

IVC International Verilog Conference. Sponsored by OVI from 1992–1998, when it was replaced by HDLCon.

level sensitive Dependent only on the steady-state input value.

LHS Left-hand side, referring to the target of an assignment in an equation or statement. The left side of an = operator. *see* **RHS**

logic verification Ensuring the correct behavior of a digital design against its specification.

logic simulation Dynamic modeling of a digital circuit using discrete values and discrete time steps.

logic synthesis The process of converting a high-level description of a design into an optimized gate-level representation.

LRM Language Reference Manual. A document describing the syntax, semantics, and usage of a programming language.

LSSD Level-sensitive scan design. A DFT technique developed at IBM [Eichelberger and Williams 1977].

LVS Layout versus schematic. Verifies that the physical layout correctly implements the desired netlist.

Mentor *see* **Mentor Graphics**

Mentor Graphics A major EDA company headquartered in Wilsonville, Oregon. Founded in 1981 and acquired by Siemens in 2017.

mixed-language A tool that supports multiple inter-operating languages such as Verilog, VHDL, or SystemC.

Model Technology Company founded in 1990 and headquartered in Beaverton, Oregon with the mission to provide digital simulation and verification tools. It is best known for creating ModelTech, a VHDL simulator, which was the precursor to ModelSim. Acquired by Mentor Graphics in 1994.

ModelSim Mentor Graphic's HDL simulator. It began in 1987 as a VHDL simulator (ModelTech) with Verilog support added in 2001. ModelSim is a leading simulator for FPGA design.

NAND Not-AND. Negation of the AND Boolean function (conjunction).

NBA Nonblocking assignment

netlist A textual description of the electronic components in a circuit and their interconnections. *see* **structural description**

nonblocking assignment A statement of the form *variable* <= *expression* where the expression is evaluated when the statement executes, but the variable update is scheduled for a later time.

NOR Not-OR. Negation of the OR Boolean function (disjunction).

OVA OpenVera Assertion

OVI Open Verilog International. A standards body and users group. Merged with VI to form Accellera.

PCB Printed circuit board

physical design tools Tools that implement the **physical layout**.

physical layout The representation of an integrated circuit in terms of planar geometric shapes that correspond to the patterns of conducting, insulating, or semiconductor layers that make up the components of an integrated circuit. Also known as integrated circuit layout or IC mask layout.

place and route A tool that creates the physical layout of an integrated circuit in two steps: The first step *places* all components in the allotted space. The second step creates the interconnections by *routing* all the wires needed to connect the placed components.

PLI Programming Language Interface

PNR, P&R Place and route

PODEM Path-oriented decision making. A novel test generation algorithm invented at IBM by Prabhu Goel [Goel 1980].

procedural assignment A statement of the form *variable* = *expression* where the expression is evaluated and the variable updated as the statement executes.

process An independent instance of the execution of a program. In the context of operating systems, each process is started with a single thread of execution, often called the primary thread, but can create additional threads from any of its threads. In Verilog, the terms *process* and *thread* are used interchangeably. *see* **thread**

Property Specification Language An assertion language developed by Accellera. Eventually became IEEE Std 1850.

property checker A verification tool that proves whether a design meets its specification by proving or disproving the specified design properties. The checker may be formal or dynamic (simulation-based).

PSL Property Specification Language

RAL Register abstraction layer. An abstract model of the registers and memories present in a hardware design.

register A memory element that stores one or more bits.

RHS Right-hand side, referring to an equation or expression. The right side of an = operator. *see* **LHS**

RTL Register transfer level. An abstraction that models digital circuits in terms of the flow of data between (storage) registers and the operations performed on the data. Sometimes **register transfer language**.

SDF Standard Delay Format [IEEE 1497-2001]

semiconductor A material whose electrical conductivity varies between a conductor and an insulator.

semiconductor foundry A fabrication facility that manufactures integrated circuits.

sign-off simulator A simulator used as part of the sign-off process. *see* **golden simulator**

sign-off The series of verification steps that a design must pass before it can be manufactured. Originally, the actual task of signing a document approving and releasing a design to be manufactured.

signal An abstraction of a wire, or group of wires, that may use discrete values and discrete time steps.

Smiths Industries Avionics Division The aviation division of Smiths Industries, a British multinational engineering business headquartered in London, UK. Founded in 1851 by Samuel Smith as a jewelry shop selling precision watches. In 2000 it was renamed Smiths Group.

SMP Symmetric multi-processing

Specman Tool developed by Veristy to support the *e* HVL.

SSI Systems Science, Inc.

STA Static timing analysis

state machine A model of computation. An abstract machine that can be in exactly one of a number of states at any given time. *see* **FSM**

static timing analysis A method to compute the expected timing of a digital circuit without having to simulate the circuit or specify a testbench (test vectors). *see* **timing constraints**

strength A model of the drive strength of a digital gate.

structural description A textual description of the electronic components in a circuit and their interconnections. *see* **netlist**

stuck-at A fault model used to mimic manufacturing defects by assuming a signal to be *stuck* at some logical value (0,1,Z,X).

Sugar An assertion language developed by IBM.

Sun Microsystems Computer and software company founded in 1982 by Scott McNealy, Andy Bechtolsheim, Vinod Khosla and Bill Joy. Notable products include computers and workstations built on its own RISC-based SPARC processor, the Solaris and SunOS operating system, and the Java language. Headquartered in Menlo Park, California, Sun was an extremely influential presence in Silicon Valley. In 2010 Oracle Corporation acquired Sun.

SV SystemVerilog

switch-level representation A netlist of switches. A model that abstracts transistors as ON or OFF switches.

synchronous A **synchronous signal** has a specific timing relationship with a given clock. A **synchronous circuit** is a clocked circuit element.

Synopsys A major EDA company headquartered in Mountain View, California. Founded in 1986 by Aart de Geus and David Gregory. Developed Design Compiler, a very successful commercial synthesis tool. In 1998 Synopsys acquired Systems Science, Inc., the creator of the Vera HVL. In 2002 Synopsys acquired Co-Design Automation, the creator of Superlog.

synthesis *see* **logic synthesis**

SystemC A C++ dialect with classes and macros that provide an event-driven simulation interface similar to that of an HDL. Early versions added models for hardware constructs; later versions focus on communication abstraction, transaction-level models, and virtual platforms.

Systems Science, Inc. Private company headquartered in Palo Alto, California. Founded in 1987 by Daniel Chapiro and Richard Kolb to provide advanced tools for the design verification and test of electronic systems. Reincorporated in 1989 when Arturo Salz joined as Director of Engineering. Developed Vera, the first commercial HVL. Acquired by Synopsys in 1998.

SystemVerilog 3.0 First SystemVerilog standardized by Accellera in June 2002.

SystemVerilog 3.1a Last SystemVerilog standardized by Accellera in May 2004.

Teradyne Massachusets company that develops automatic test equipment.

testing Confirming that a hardware component has been manufactured correctly. *see* **verification**

test vector A set of inputs provided to a system to test correct operation of that system. *see* **test pattern**

test pattern A sequence of inputs applied to a digital circuit to enable test equipment to distinguish between correct and faulty behaviors of the DUT. ATPG tools are able to generate highly-effective patterns measured by the number of detectable defects and the size of the patterns.

test insertion A tool that implements DFT by adding testability features to a hardware design, for example scan paths.

testbench A description of the input values (stimulus) to be applied to the design under test (DUT) and optionally the output values expected (response). The environment surrounding a DUT.

thread A subprogram (or sequence of instructions) within a process that can be scheduled for execution. In the context of operating systems, all threads (of the same process) share the same memory space and system resources whereas processes run in separate memory spaces. *see* **process**

three-state A signal that can be 0, 1, or high impedance (not driven). Also called **tristate** or **tri-state**.

timing check A property that ensures correct operation of the timing of the design.

timing hazard When the transient behavior of a logic circuit differs from its steady-state, or when an output changes more than once as the result of a single input transition.

timing constraints A series of limits applied to circuit paths that dictate the desired operating speed of a design. Used by logic synthesis tools, which minimize delays (and thus maximize circuit speed) by considering the arrival times at the primary inputs and the required times at the primary outputs. Used by static timing analysis tools.

timing analysis A method to compute the expected timing behavior of a digital circuit.

TLM Transaction level modeling. A highly-abstract modeling style that represents the transfer of data as transactions. Also **transaction level model**.

UVM Universal Verification Methodology

VCS Verilog Compiled Simulator developed by Chronologic Simulation. The first compiled-code Verilog simulator and the most successful commercial competitor to Gateway's Verilog-XL simulator.

verification The task of checking that a design behaves correctly. In hardware development, **verification** means checking that the design is correct, like **testing** in software development. The term **testing** is used for actual hardware, which may have manufacturing faults. *see* **testing**

Verilog 1.0 Short name for Verilog-95. Verilog standard IEEE Std 1364-1995.

Verilog 2.0 Short name for Verilog-2001. Verilog standard IEEE Std 1364-2001.

Verilog-2001 Verilog standard IEEE Std 1364-2001.

Verilog 3.0 *see* **SystemVerilog 3.0**

Verilog 3.1a *see* **SystemVerilog 3.1a**

Verilog-95 Verilog standard IEEE Std 1364-1995.

Verilog-XL Gateway Design's Verilog simulator created by Phil Moorby in 1987. Verilog-XL featured very fast gate-level simulation speeds, and was the first Verilog simulator to be certified for ASIC sign-off.

Verisity Company (originally called InSpec) founded in 1995 by Yoav Hollander to create tools to support the *e* HVL he had previously developed. The Specman tool supporting *e* was released in 1996 and in 2006 *e* became an IEEE standard. Acquired by Cadence In 2005.

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit, a 1980s US government research program.

VI VHDL International. User group promoting VHDL. Merged with OVI to form Accellera.

Viewlogic Systems An EDA company founded in 1984 and headquartered in Mountain View, California. In 1994 it acquired Chronologic Simulation, makers of the VCS Verilog simulator. Viewlogic was acquired by Synopsys in 1997. In 1998 Synopsys spun out the PCB and System units and retained the IC design unit that included VCS.

VIP Verification intellectual property. A verification component that is connected to a hardware design to validate the operation of an interface or protocol.

VIUF VHDL International Users Forum. Sponsored by VI from 1991–1998, when it was replaced by HDLCon.

VLSI Very large-scale integration. Initially this referred to IC chips incorporating more than 10,000 devices.

wired-OR A set of gate outputs that have an OR function when wired together.

wired-AND A set of gate outputs that have an AND function when wired together.

XL Verilog-XL simulator.

XOR Exclusive OR Boolean function. True when only one of its two inputs is true.

zero-extend Setting the high bits of a multi-bit signal to zero.

Zycad Company founded in 1981 to develop and market a hardware accelerator for gate-level simulation.

D ANNOTATED IMAGES

2	Phil Moorby	Verilog engineering team mgr.	47	Asad Khan	Verifault engineering team mgr.
3	Dave Grandin	Verilog engineering team	53	Eric Leavitt	Engineering mgr.
4	Jason Campbell	Verilog applications engineer	55	Kathleen Leavitt	Apps manager
6	Bill Fletcher	Sales	57	Charlie Loegering	Sales midwest
8	Steve Caplow	Verilog marketing	58	Ronna Alintuck	Verilog marketing
10	Pete Johnson	Verilog marketing	60	Stu Sutherland	Training mgr.
13	Dave Rich	Verilog applications engineer	61	Joel Paston	Verilog engineering team
14	Bob Sullivan	VP Marketing	72	Andy Stein	Applications engineer
15	George Indaco	Sales	81	Martin Harding	Director, ASIC business unit
18	Prabhu Goel	Founder and CEO of Gateway	84	Chris Browy	Applications engineer
23	Manny Correia	VP Applications	88	Simon Davidmann	European technical manager
32	Rich Davenport	Director, central sales	89	Craig Robbins	VP Sales
33	Dan Keshian	CFO	94	Tom Meyer	Verilog engineering team
34	Scott Sandler	Applications engineering mgr.	96	Leigh Brady	Verilog engineering team
36	George Bakewell	Applications engineer	101	Gary Leive	VP Engineering
39	Victor Berman	VHDL specialist	102	Bill Hoolhurst	Sales
42	Chi-lai Huang	Veritime engineering team	104	Patrick Beauvillard	Applications engineer
46	Phil Mason	Verilog engineering team			

Fig. 22. Gateway Design Automation corporate party in late 1989, around the time of the Cadence acquisition. Annotated to show many of the names and their positions in the company. Figure 15 on Page 38 shows the original image. Additional information can be found in People on Page 73.

1	Tom Fitzpatrick	7	Leigh Brady	15	Simon Davidmann
2	Christian Burisch	8	Joan Frazer	16	Dave Rich
3	Phil Moorby	9	Nanette Collins	17	Matthew Hall
4	Lee Moore	10	Stephanie Waters	18	Peter Flake
5	Andy Stein	12	Henry Cox	19	Stu Sutherland
6	Veronique Hermans	14	Dave Kelf	21	Chris Podger

Fig. 23. Most of the Co-Design staff, along with a few friends and consultants, attends Design Automation Conference (DAC) in June 2002 in New Orleans. Annotated to show many of the names. Figure 17 on Page 51 shows the original image. Additional information can be found in People on Page 73.

REFERENCES

Accellera. 2002a. *SystemVerilog 3.0 Accellera's Extensions to Verilog.* Standard. Accellera Organization, Napa, CA, USA. NON-ARCHIVAL http://www.eda-twiki.org/sv-ec/3.0_LRM.pdf.

Accellera. 2002b. *VHDL Users Group / VHDL International Users' Forum (VUG/VIUF) Collected Conference Proceedings.* Accellera. Archived at https://web.archive.org/web/20020803031109/http://eda.org/VIUF_proc/.

Accellera. 2003. *SystemVerilog 3.1, Accellera's Extensions to Verilog.* Standard. Accellera Organization, Napa, CA, USA. NON-ARCHIVAL http://www.eda-twiki.org/sv-ec/SystemVerilog_3.1_final.pdf (also at Internet Archive 7 April 2020 14:23:40).

Accellera. 2004a. *Property Specification Language Reference Manual Version 1.1.* Standard. Accellera Organization, Napa, CA, USA. Archived at https://web.archive.org/web/20040913230716/http://www.eda.org/vfv/docs/PSL-v1.1.pdf.

Accellera. 2004b. *SystemVerilog 3.1a Language Reference Manual, Accellera's Extensions to Verilog.* Standard. Accellera Organization, Napa, CA, USA. Archived at https://web.archive.org/web/20040626122728/http://www.eda.org/sv/SystemVerilog_3.1a.pdf.

Accellera. 2011. *Universal Verification Methodology (UVM) 1.0 Class Reference.* Standard. Accellera Organization, Napa, CA, USA. NON-ARCHIVAL https://www.accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.0.pdf.

Accellera. 2014. *Verilog-AMS Language Reference Manual 2.4.0.* Standard. Accellera Systems Initiative, Napa, CA, USA. NON-ARCHIVAL https://www.accellera.org/images/downloads/standards/v-ams/VAMS-LRM-2-4.pdf.

Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. 2002. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Grenoble, France, 2002-04-08/2002-04-12) (*TACAS '02*). Springer-Verlag, Berlin, Heidelberg, Germany, 296–311. https://dl.acm.org/doi/abs/10.5555/646486.694616. Volume 2280 of Lecture Notes in Computer Science.

Ken Arnold and James Gosling. 1998. *The Java Programming Language* (second ed.). Addison-Wesley, New York, NY, USA.

Peggy Aycinena. 2005. *Phil Moorby - 2005 Kaufman Award.* EDA Confidential (8 nov). Archived at https://web.archive.org/web/20060325053055/http://www.aycinena.com/index2/index3/archive/phil%20moorby.html.

Ayciena gives a report on the awards dinner, and includes an interview with Phil Moorby.

Mike Baird. 1998. *IVC/VIUF HDL Conference.* OVI & VI. Archived at https://web.archive.org/web/19980202191343/http://www.hdlcon.org/toppage.html.

Mike Baird. 1999. *The International HDL Conference and Exhibition.* HDLCon. Archived at https://web.archive.org/web/19990208004605/http://hdlcon.org/.

Mario Barbacci, C. Gordon Bell, and Allen Newell. 1972. ISP: A Language To Describe Instruction Sets and Other Register Transfer Systems. In *IEEE Computer Society International Conference* (San Francisco, CA, USA, 1972-09-12/1972-09-14) (*COMPCON '72*). IEEE, New York, NY, USA, 219–222. An expanded version, dating from July 1972, can be found at https://web.archive.org/web/20170323143040/https://gordonbell.azurewebsites.net/CGB%20Files/ISP%20Language%20to%20Describe%20Instruction%20Sets%207207%20c.pdf.

Thomas C. Bartee. 1960. *Digital computer fundamentals.* McGraw-Hill, New York, NY, USA.

The original 1960 edition is a classic early text on computer hardware. At the time, Bartee was at MIT Lincoln Laboratory.

Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. 2001. The Temporal Logic Sugar. In *Proceedings of the 13th International Conference on Computer Aided Verification* (Paris, France, 2001-07-18/2001-07-22) (*CAV '01*). Springer-Verlag, Berlin, Heidelberg, Germany, 363–367. https://dl.acm.org/doi/abs/10.5555/647770.734116.

C. Gordon Bell. 1991. *High-Tech Ventures.* Addison-Wesley, Reading, MA, USA. Scan available at NON-ARCHIVAL https://gordonbell.azurewebsites.net/CGB%20Files/High-Tech%20Ventures%201991%20c.pdf (also at Internet Archive 31 March 2020 15:17:55).

Pages 289–299 discuss Gateway Design Automation.

Nuel D. Belnap, Jr. 1977. A Useful Four-Valued Logic. In *Modern Uses of Multiple-Valued Logic*, J. Michael Dunn and George Epstein (Eds.). Springer Netherlands, Dordrecht, 5–37. https://doi.org/10.1007/978-94-010-1161-7

Michael J. Bending. 1984. Hitest: A Knowledge-Based Test Generation System. *IEEE Design Test of Computers* 1, 2 (may), 83–92. https://doi.org/10.1109/MDT.1984.5005617

Janick Bergeron, Eduard Cerny, Alan Hunter, and Andy Nightingale. 2005. *Verification Methodology Manual for SystemVerilog.* Springer-Verlag, Berlin, Heidelberg, Germany.

Neil C. Berglund. 1979. Level-Sensitive Scan Design Tests Chips, Boards, System. *Electronics* 52, 6 (March), 108–110.

Dominique Borrione, Robert Piloty, Dwight Hill, Karl J. Lieberherr, and Philip Moorby. 1992. Three Decades of HDLs, Part II: Conlan Through Verilog. *IEEE Design & Test of Computers* 9, 3 (sep), 54–63. https://doi.org/10.1109/54.156158

Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. 1984. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, MA, USA.

Cadence. 2000. *TestBuilder*. Cadence Design Systems, San Jose, CA, USA. Archived at https://web.archive.org/web/20001018010606/http://www.testbuilder.net/.

Cadence and Mentor Graphics. 2008. *Open Verification Methodology*. Cadence and Mentor Graphics (9 Jan.). Archived at https://web.archive.org/web/20080115101515/http://www.ovmworld.org/.

Michael Carroll. 1993. VHDL—panacea or hype? *IEEE Spectrum* 30, 6 (jun), 34–37. https://doi.org/10.1109/6.214581

Kai-Hui Chang and Chris Browy. 2012. Improving Gate-Level Simulation Accuracy When Unknowns Exist. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, CA, USA, 2012-06-03/2012-06-07) (*DAC '12*). Association for Computing Machinery, New York, NY, USA, 936–940. https://doi.org/10.1145/2228360.2228528

CHM. 2015. *Computer History Museum Announces its 2016 Fellow Award Honorees*. Computer History Museum (17 nov). Archived at https://web.archive.org/web/20200327134657/https://computerhistory.org/press-releases/fellows-2016/.

CHM. 2016. *Philip Moorby*. Computer History Museum. Archived at https://web.archive.org/web/20200331001046/https://computerhistory.org/profile/philip-moorby/. Video at https://www.youtube.com/watch?v=pUmCdQVukrg. See earlier version at https://web.archive.org/web/20161003015054/https://www.computerhistory.org/fellowawards/hall/Philip-Moorby/. Also see this page about the 2016 Fellow Awards https://web.archive.org/web/20191003110953/https://computerhistory.org/blog/2016-chm-fellow-awards/.

Peter Clarke. 1999. Startup to field next-generation design language. *Electronic Engineering Times* (1 June). Archived at https://web.archive.org/web/20001211142400/http://www.edtn.com:80/story/tech/OEG19990531S0003-R.

An informative article that provides a very good summary of Co-Design and its vision for Superlog. Also mentions Co-Design's early investors.

Ronald Collett. 1993. VHDL by TKO. *Electronic Engineering Times* (4 Jan.), 42. Scan available at https://web.archive.org/web/20200331183910/https://trilobyte.com/pdf/VHDL_by_TKO_Collett_1993-01-04.pdf.

John Cooley. 1995. *INDUSTRY GADFLY: The Fall(ing) VIUF '95*. Usenet, comp.arch.fpga (1 Dec.). Available from Google Groups at NON-ARCHIVAL https://groups.google.com/d/msg/comp.arch.fpga/AHzIcNVTG9E/ah9nIyAHHhwJ.

John Cooley wrote for a number of US trade publications, and afterwards he would often post his columns on Usenet. This column begins:

After attending the Fall meeting of the VHDL International Users Forum (VIUF) last week in Boston, it was painfully obvious that the six year old VIUF was in trouble.

Usenet header: Date: 1995/12/01 ; Message-ID: <DIwF8t.MCC@world.std.com> ; newsgroups: comp.arch.fpga, comp.lsi, comp.lsi.cad, comp.lsi.testing, sci.electronics, sci.engr.semiconductors, comp.cad.compass.

John Cooley. 1996. *INDUSTRY GADFLY: "From Beirut To Bosnia" + Reader Response*. Usenet, comp.lang.vhdl (13 March). This column originally published in *Electronic Engineering Times* (5 Feb 1996). Available from Google Groups at NON-ARCHIVAL https://groups.google.com/d/msg/comp.lang.vhdl/Lfynspa9cnM/r6Bd5KwXejQJ.

John Cooley wrote for a number of US trade publications, and afterwards he would often post his columns on Usenet. This column begins:

More than once I've thought of the Verilog/VHDL Wars as being the EDA industry's equivalent of Beirut. What has made this war particularly interesting to watch was the proxy propaganda war being fought via industry analysts. Ron Collett of Collett International made big news for predicting that VHDL was going to beat Verilog some time around '92. Gary Smith of Dataquest also caused a minor uproar when he claimed at the '94 DAC that VHDL revenues were just then beating Verilog revenues and projected it would only get better for VHDL. Bill Fuchs, the never shy Chairman of Open Verilog International (OVI), howled over the past four years that OVI's surveys found that Verilog's revenues were easily *twice* to *four times* what these analysts were seeing.

Usenet header: Date: 1996/03/13 ; Message-ID: <Do859p.FFJ@world.std.com> ; newsgroups: comp.cad.cadence, comp.sys.mentor, comp.lang.vhdl, comp.lang.verilog, comp.lsi, comp.cad.synthesis, comp.arch, comp.arch.fpga.

John A. Darringer. 1968. A Language for the Description of Digital Computer Processors. In *Proceedings of the 5th Annual Design Automation Workshop* (Washington, DC, USA, 1968-07-15/1968-07-18) (*DAC '68*). Association for Computing Machinery, New York, NY, USA, 15.1–15.18. https://doi.org/10.1145/800167.805400

Roger Dettmer. 2004. The HILO inheritance. *IEE Review* 50, 8 (aug), 22–26. https://doi.org/10.1049/ir:20040803

Interview with Peter Flake.

Antun Domic. 2015. The Evolution of Logic Synthesis: Dots and Dashes… Zeros and Ones. In *EPFL Workshop on Logic Synthesis & Verification* (Lausanne, Switzerland, 2015-12-11). EPFL, Lausanne, Switzerland, 1–49. Archived at https://web.archive.org/web/20200327132626/https://si2.epfl.ch/~demichel/si.epfl.ch/files/content/sites/si/files/shared/Logic%20Synthesis/slides/EPFL%20LSV%20Workshop%202015_Antun%20Domic.pdf.

This is the Closing Keynote from the conference. Slides 14–15 show vintage documents from the earliest days of Synopsys (then called Optimal Solutions), including the phrase "What can LOGIC COMPILER do? Convert equations into working circuits".

Surrendra Dudani and Eduard Cerny. 2003. *Authoring assertion IP using OpenVera assertion language.* Design And Reuse (7 Feb.). https://www.design-reuse.com/articles/4835/authoring-assertion-ip-using-openvera-assertion-language.html (also at Internet Archive 8 Sept. 2009 20:33:49).

James R. Duley and Donald L. Dietmeyer. 1968. A Digital System Design Language (DDL). *IEEE Trans. Comput.* C-17, 9 (sep), 850–861. https://doi.org/10.1109/TC.1968.229145

EDAC. 2005. *Phil Moorby Selected to Receive EDA Industry's Kaufman Award.* The EDA Consortium (10 oct). Archived at https://web.archive.org/web/20200330224958/http://esd-alliance.org/wp-content/uploads/PDFs/PhilKaufmanAward/05-10-10_PK_Award_FINAL.pdf.

Press release from EDA Consortium.

Edward B. Eichelberger. 1965. Hazard Detection in Combinational and Sequential Switching Circuits. *IBM Journal of Research and Development* 9, 2 (March), 90–99. https://doi.org/10.1147/rd.92.0090

Edward B. Eichelberger and Thomas W. Williams. 1977. A Logic Design Structure for LSI Testability. In *Proceedings of the 14th Design Automation Conference* (New Orleans, LA, USA, 1977-06-20/1977-06-22) (*DAC '77*). IEEE Press, New York, NY, USA, 462–468. https://dl.acm.org/doi/abs/10.5555/800262.809170

Limor Fix. 2008. Fifteen Years of Formal Property Verification in Intel. In *25 Years of Model Checking*, Orna Grumberg and Helmut Veith (Eds.). Lecture Notes in Computer Science, Vol. 5000. Springer, Berlin, Heidelberg, Germany, 139–144. https://doi.org/10.1007/978-3-540-69850-0_8

Peter L. Flake and Simon J. Davidmann. 2000. Superlog, a Unified Design Language for System-on-Chip. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference* (Yokohama, Japan, 2000-01-25/2000-01-28) (*ASP-DAC '00*). Association for Computing Machinery, New York, NY, USA, 583–586. https://doi.org/10.1145/368434.368814

Peter L. Flake, Philip R. Moorby, and Gerry Musgrave. 1980. Logic Simulation of Bi-directional Tri-state Gates. In *IEEE International Conference on Circuits and Computers* (Port Chester, NY, USA, 1980-10-01/1980-10-03) (*ICCC 80*). IEEE, New York, NY, USA, 594–600.

Peter L. Flake, Philip R. Moorby, and Gerry Musgrave. 1981. HILO Mark 2 Hardware Description Language. In *Computer Hardware Description Languages and Their Applications*, Melvin A. Breuer and Reiner Hartenstein (Eds.). North-Holland, Amsterdam, Netherlands, 95–108.

This is *Proceedings of the IFIP TC-10 Fifth International Conference on Computer Hardware Description Languages and Their Applications (Kaiserslautern, Germany) (CHDL '81)*, 7–9 Sep 1981.

Peter L. Flake, Philip R. Moorby, and Gerry Musgrave. 1983. An Algebra for Logic Strength Simulation. In *Proceedings of the 20th Design Automation Conference* (Miami Beach, FL, USA, 1983-06-27/1983-06-29) (*DAC '83*). IEEE Press, New York, NY, USA, 615–618. https://dl.acm.org/doi/abs/10.5555/800032.800733.

Peter L. Flake, Gerry Musgrave, and Mike Shorland. 1975a. The HILO logic simulation language. In *International Symposium on Computer Hardware Description Languages and Their Applications* (New York, NY, USA, 1975-09-03/1975-09-05) (*CHDL '75*). IEEE, New York, NY, USA, 134–142. IEEE catalog 75CH1010-8C.

Peter L. Flake, Gerry Musgrave, and Ian J. White. 1974. HILO—A Logic System Simulator. In *International Conference on Computer Aided Design* (The University Of Southampton, UK, 1974-04-08/1974-04-11). Institution of Electrical Engineers, London, UK, 130–136. IEE Conference Publication 111.

Peter L. Flake, Gerry Musgrave, and Ian J. White. 1975b. A Digital Systems Simulator—HILO. *Digital Processes* 1, 1, 39–53. Georgi Publishing Co.

Harry Foster. 2019. *Part 10: The 2018 Wilson Research Group Functional Verification Study: IC/ASIC Language and Library Adoption Trends.* Mentor (14 Feb.). Archived at https://web.archive.org/web/20190406084950/https://blogs.mentor.com/verificationhorizons/blog/2019/02/14/part-10-the-2018-wilson-research-group-functional-verification-study/. Also see the introduction to the study at https://web.archive.org/web/20200330191521/https://blogs.mentor.com/verificationhorizons/blog/2018/11/14/prologue-the-2018-wilson-research-group-functional-verification-study/.

This post discusses the usage of design, verification, and assertion languages in ASIC design teams. This study is the latest in a series of industry studies on functional verification, commissioned by Mentor in 2007, 2010, 2012, 2014, 2016, and 2018. The study covers both FPGA and IC/ASIC users.

FSF. 2000. *Bison.* Free Software Foundation. Archived at https://web.archive.org/web/20000831181346/http://www.gnu.org/software/bison/.

Prabhakar Goel. 1980. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. In *Digest of Papers, The 10th International Symposium on Fault-Tolerant Computing* (Kyoto, Japan, 1980-10-01/1980-10-03) (*FTCS-10*). IEEE, New York, NY, USA, 145–151. Reprinted in *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, 'Highlights from Twenty-Five Years'.* (Pasadena, CA, USA, 1995-06-27/1995-06-30). (*FTCS-25*). IEEE Computer Society Press, Los Alamitos, CA, USA, 337–343. https://doi.org/10.1109/FTCSH.1995.532656.

Prabhu Goel. 2017. Prabhu Goel, Silicon Valley. In *The $8 Man*, Brenda H. Christensen (Ed.). Redtop Publishing, Woodside, CA, USA, 51–76. Excerpt available at NON-ARCHIVAL https://www.the8dollarman.com/bios-8-men (also at Internet Archive 19 March 2019 14:29:36).

Oral histories of Indian immigrants to the United States in 1960s–1970s.

Richard Goering. 2005. Verilog's inventor nabs EDA's Kaufman award. *Electronic Engineering Times* (7 Nov.). NON-ARCHIVAL https://www.eetimes.com/verilogs-inventor-nabs-edas-kaufman-award/. Also see this Usenet posting in comp.lang.verilog NON-ARCHIVAL https://groups.google.com/forum/#!searchin/comp.lang.verilog/inventor\protect\T1\textdollar20nabs\protect\T1\textdollar20kaufman/comp.lang.verilog/BND2Y24RSrI/d7AwwvmrQW4J.

Robert L. Harris, Simon J. Davidmann, and Gerry Musgrave. 1984a. HILO-2 - a system to build on. In *6th International Conference and Exhibition on Computers in Design Engineering* (Brighton, UK, 1984-04-03/1984-04-05) (*CAD84*). Butterworths, Guildford, Surrey, UK, 48–60. NON-ARCHIVAL https://www.elsevier.com/books/cad84/wexler/978-0-408-01440-3. Scanned version available at NON-ARCHIVAL https://books.google.com/books?id=zgL9BAAAQBAJ&pg=PA48.

Robert L. Harris, Gerry Musgrave, and Simon J. Davidmann. 1984b. HILO-2 - a system to build on. *Computer-Aided Design* 16, 2 (mar), 104.

Reiner Hartenstein. 2018. *The History of CHDL Conferences.* Archived at https://web.archive.org/web/20180829003543/http://hartenstein.de/CHDL/

Chi-lai Huang. 1981. *Computer-aided logic synthesis based on a new multi-level hardware design language—LALSD II.* Ph.D. Dissertation. State University of New York, Binghamton, NY, USA. Advisor(s) Stephen Y. H. Su. NON-ARCHIVAL https://suny-bin.primo.exlibrisgroup.com/permalink/01SUNY_BIN/1igql2k/proquest303100438.

Brian Hunter. 2016. *Advanced UVM* (second ed.). CreateSpace, Scotts Valley, CA, USA.

IEEE. 1974. Hardware Description Languages. *IEEE Computer* 7, 12 (dec), 1–85. https://doi.org/10.1109/MC.1974.6323396 Special issue devoted to HDLs.

IEEE 1076-1987. *IEEE Standard VHDL Language Reference Manual.* IEEE, New York, NY, USA. https://standards.ieee.org/standard/1076-1987.html. IEEE Std 1076-1987.

IEEE 1076-1993. *IEEE Standard VHDL Language Reference Manual.* IEEE, New York, NY, USA. https://standards.ieee.org/standard/1076-1993.html. IEEE Std 1076-1993.

IEEE 1076.4-1995. *IEEE Standard VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification.* IEEE, New York, NY, USA. https://standards.ieee.org/standard/1076_4-1995.html. IEEE Std 1076.4-1995.

IEEE 1164-1993. *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164).* IEEE, New York, NY, USA. http://standards.ieee.org/standard/1164-1993.html. IEEE Std 1164-1993.

IEEE 1364-1995. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language.* IEEE, New York, NY, USA. https://standards.ieee.org/standard/1364-1995.html. IEEE Std 1364-1995.

IEEE 1364-2001. *IEEE Standard Verilog Hardware Description Language.* IEEE, New York, NY, USA. https://standards.ieee.org/standard/1364-2001.html. IEEE Std 1364-2001.

IEEE 1364-2002. *IEEE Standard for Verilog Register Transfer Level Synthesis.* IEEE, New York, NY, USA. https://standards.ieee.org/standard/1364_1-2002.html. IEEE Std 1364.1-2002.

IEEE 1364-2005. *IEEE Standard Verilog Hardware Description Language.* IEEE, New York, NY, USA. https://standards.ieee.org/standard/1364-2005.html. IEEE Std 1364-2005.

IEEE 1497-2001. *IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process.* IEEE, New York, NY, USA. https://standards.ieee.org/standard/1497-2001.html. IEEE Std 1497-2001.

IEEE 1647-2006. *IEEE Standard for the Functional Verification Language 'e'.* IEEE, New York, NY, USA. https://standards.ieee.org/standard/1647-2006.html. IEEE Std 1647-2006.

IEEE 1666-2011. *IEEE Standard for Standard SystemC Language Reference Manual.* IEEE, New York, NY, USA. http://standards.ieee.org/standard/1666-2011.html. IEEE Std 1666-2011.

IEEE 1800-2005. *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.* IEEE, New York, NY, USA. http://standards.ieee.org/standard/1800-2005.html. IEEE Std 1800-2005.

IEEE 1800-2009. *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.* IEEE, New York, NY, USA. http://standards.ieee.org/standard/1800-2009.html. IEEE Std 1800-2009.

IEEE 1800-2012. *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.* IEEE, New York, NY, USA. http://standards.ieee.org/standard/1800-2012.html. IEEE Std 1800-2012.

IEEE 1800-2017. *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.* IEEE, New York, NY, USA. http://standards.ieee.org/standard/1800-2017.html. IEEE Std 1800-2017.

IEEE 1800.2-2017. *IEEE Standard for Universal Verification Methodology Language Reference Manual.* IEEE, New York, NY, USA. https://standards.ieee.org/standard/1800_2-2017.html. IEEE Std 1800.2-2017.

IEEE 1850-2005. *IEEE Standard for Property Specification Language (PSL).* IEEE, New York, NY, USA. http://standards.ieee.org/standard/1850-2005.html. IEEE Std 1850-2005.

IEEE 1850-2010. *IEEE Standard for Property Specification Language (PSL).* IEEE, New York, NY, USA. http://standards.ieee.org/standard/1850-2010.html. IEEE Std 1850-2010.

Sasan Iman and Sunita Joshi. 2004. *The e Hardware Verification Language.* Springer, New York, NY, USA. https://doi.org/10.1007/b117092

Mahendra Jain. 1993. The VHDL forecast. *IEEE Spectrum* 30, 6 (jun), 36. Full-page sidebar in: Michael Carroll. 1993. VHDL—panacea or hype? *IEEE Spectrum* 30, 6 (jun), 34–37. https://doi.org/10.1109/6.214581.

Stephen C. Johnson. 1975. *Yacc - Yet another compiler-compiler.* Computer Science Technical Report 32. Bell Telephone Laboratories, Murray Hill, NJ, USA.

Harvey Jones. 2009. *Oral History of Harvey Jones.* Computer History Museum (6 Nov.). http://www.computerhistory.org/collections/catalog/102702217 (also at Internet Archive 3 Oct. 2013 02:33:32). Catalog number 102702217.

Harry F. Jordan and Burton J. Smith. 1973. Structure of Digital System Description Languages. In *Proceedings of the 1st Annual Symposium on Computer Architecture* (Gainesville, FL, USA, 1973-12) (*ISCA '73*). Association for Computing Machinery, New York, NY, USA, 31–34. https://doi.org/10.1145/800123.803968

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming* (Jyväskylä, Finland, 1997-06-09/1997-06-13) (*ECOOP '97*). Springer, Berlin, Germany, 220–242. https://www.springer.com/us/book/9783540630890. Volume 1241 of Lecture Notes in Computer Science.

Lindsay Kleeman and Antonio Cantoni. 1987. Metastable Behavior In Digital Systems. *IEEE Design and Test of Computers* 4, 6 (dec), 4–19. https://doi.org/10.1109/MDT.1987.295189

Jan Łukasiewicz. 1920. O logice trójwartościowej. *Ruch Filozoficzny* 5, 170–171. Translation: On three-valued logic. Also see: Peter Simons. 2017. Jan Łukasiewicz. In *The Stanford Encyclopedia of Philosophy* (spring 2017 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University. Archived at https://plato.stanford.edu/archives/spr2017/entries/lukasiewicz/.

Rajeev Madhavan. 1997. *About IVC.* OVI. Archived at https://web.archive.org/web/19970414023820/http://www.hdlcon.org/aboutivc.html.

Mentor Graphics. 2006. *Mentor Graphics Delivers the Next Generation of Functional Verification.* Mentor Graphics, Inc. (6 may). Archived at https://web.archive.org/web/20100523143319/https://www.mentor.com/company/news/questa_avm.

MIL-STD-454L. 1988. *Standard General Requirements for Electronic Equipment.* US Department of Defense, Washington, DC, USA (20 Sep).

Don Mills and Clifford E. Cummings. 1999. RTL Coding Styles That Yield Simulation and Synthesis Mismatches. In *Synopsys Users Group Conference* (San Jose, CA, USA, 1999-03-29/1999-03-30) (*SNUG San Jose 1999*). Synopsys, Mountain View, CA, USA, 1–16. NON-ARCHIVAL http://www.sunburst-design.com/papers/CummingsSNUG1999SJ_SynthMismatch.pdf (also at Internet Archive 12 Jan. 2006 04:11:53).

Philip Moorby. 2013. *Oral History of Philip Raymond "Phil" Moorby.* Computer History Museum (22 April). https://www.computerhistory.org/collections/catalog/102746653 (also at Internet Archive 4 Jan. 2014 12:35:24). Catalog number 102746653.

Phil Moorby, Arturo Salz, Peter Flake, Surrendra Dudani, and Tom Fitzpatrick. 2003. Achieving Determinism in SystemVerilog 3.1 Scheduling Semantics. In *Design and Verification Conference* (San Jose, CA, USA, 2003-02-24/2003-02-25) (*DVCon 2003*). Accellera, Napa, CA, USA, 1–7. Archived at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.385.1556&rep=rep1&type=pdf.

Philip R. Moorby. 1983. Fault Simulation using Parallel Value Lists. In *IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, USA, 1983-09-12/1983-09-15) (*ICCAD '83*). IEEE, New York, NY, USA, 101–102.

Gabe Moretti. 2015. *DVCon is the Primary Design and Verification Conference.* Chip Design Magazine (20 Feb.). Archived at https://web.archive.org/web/20150304150425/http://chipdesignmag.com/sld/moretti/2015/02/20/dvcon-is-the-primary-design-and-verification-conference.

Daniel Nenni and Paul McLellan. 2019. *Fabless: The Transformation of the Semiconductor Industry.* SemiWiki.com, USA. NON-ARCHIVAL https://semiwiki.com/forum/images/BookCovers/Fabless%202019%20Version%20PDF.pdf (also at Internet Archive 5 Nov. 2019 22:05:09).

A. Richard Newton. 2005. *Presentation of the 2005 Phil Kaufman Award to Phil Moorby.* University of California, Berkeley, CA, USA (1 nov). Archived at https://web.archive.org/web/20200330233003/https://people.eecs.berkeley.edu/~newton/Presentations/Kaufman/PMPresent.html.

Richard Newton's speech introducing Moorby at the awards ceremony.

Rishiyur S. Nikhil and Arvind. 2009. What is Bluespec? *ACM SIGDA Newsletter* 39, 1 (jan), 1. https://doi.org/10.1145/1862876.1862877

Irving S. Reed. 1952. Symbolic Synthesis of Digital Computers. In *Proceedings of the 1952 ACM National Meeting (Toronto)* (Toronto, Ontario, Canada, 1952-06) (*ACM '52*). Association for Computing Machinery, New York, NY, USA, 90–94. https://doi.org/10.1145/800259.809004

Irving S. Reed. 1956. *Symbolic design techniques applied to a generalized computer.* Technical Report TR-141. MIT Lincoln Laboratory, Cambridge, MA, USA (3 jan).

Writing in 1972, Reed told this story about the use of his register transfer language:

Perhaps its greatest impact was felt in the design of CG-24, an all solid state machine, completed and operating early in 1957. To illustrate how well the register transfer language was developed at that time, Miss B. Jensen, a programmer with little knowledge of computer hardware had previously taken the register transfer language design of the proposed machine and programmed the computer TX-0 to almost completely simulate the structure and operations of the, then, not completed CG-24.

Republished and discussed in: Irving S. Reed. 1972. Symbolic design techniques applied to a generalized computer. *IEEE Computer* 5, 3 (may/jun), 46–52. https://doi.org/10.1109/C-M.1972.216916.

Ray Salemi. 2013. *The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology.* Boston Light Press, Boston, MA, USA.

Arturo Salz, Bruce Greene, and Robert Booth. 2012. X-Optimism Elimination during RTL Verification. In *Synopsys Users Group Conference* (Santa Clara, CA, USA, 2012-03-26/2012-03-28) (*SNUG Silicon Valley 2012*). Synopsys, Mountain View, CA, USA, 1–16. Archived at https://web.archive.org/web/20200331183948/https://trilobyte.com/pdf/x-optimism_elimination_SNUG_2012.pdf.

Claude Elwood Shannon. 1940. *A symbolic analysis of relay and switching circuits.* Master's thesis. MIT, Dept. of Electrical Engineering, Cambridge, MA, USA. Archived at http://hdl.handle.net/1721.1/11173.

SIGARCH. 1973. Workshop on Computer Description Languages. *SIGARCH Comput. Archit. News* 2, 1, 8.

An announcement of the upcoming workshop:

On May 29 and 30, 1973 a workshop will be held at Rutgers University in New Brunswick, New Jersey with Professor Saul Levy of the Computer Science Department at Rutgers who will be the Chairman.. . . This is a workshop to examine the state of the art in computer description languages and the relative merits of the various approaches.

Note the workshop was actually held on 6–7 Sep 1973.

SIGARCH. 1974. Meetings. *SIGARCH Comput. Archit. News* 3, 2, 40.

Call for papers and announcement of the upcoming workshop:

July 31, Aug. 1, 1974 WORKSHOP ON COMPUTER HARDWARE DESCRIPTION LANGUAGES, Techische Hochschule Darmstadt, Federal Republik Germany.

After the first workshop on the topic of computer hardware description languages at Ruthers [*sic*] University in September 1973, a second will be held at Technische Hochschule Darmstadt immediately before the IFIP Conference in Stockholm. It serves to review the progress on hardware description language definition and application toward the development of a proposal for an international language standard.

Michael John Sebastian Smith. 1993. *Application-specific integrated circuits.* Addison-Wesley, Reading, MA, USA. Archived at https://archive.org/details/ApplicationSpecificIntegratedCircuitsAddisonWesleyMichaelJohnSebastianSmith/mode/2up.

Mike Stellfox. 2010. *UVM - 10 Years in the Making ...* Cadence Design Systems (17 may). NON-ARCHIVAL https://community.cadence.com/cadence_blogs_8/b/fv/posts/uvm-methodology-comes-first-circle.

This post mentions the history of URM:

When Cadence acquired Verisity in 2005, SystemVerilog was starting to gain attention as a second standard verification language, so several of the original eRM experts decided to adapt the eRM concepts to build a SystemVerilog verification methodology, which was released as the URM in 2006. The URM was a single verification methodology for both SystemVerilog and e, including support for multi-language environments.

Stuart Sutherland. 2000. The IEEE Verilog 1364-2001 Standard: What's New, and Why You Need It. In *9th Annual International HDL Conference and Exhibition* (San Jose, CA, USA, 2000-03-08/2000-03-10) (*HDLCon '00*). OVI & VI, USA, 1–8. Archived at https://web.archive.org/web/20030403025314/https://sutherland-hdl.com/papers/2000-HDLCon-paper_Verilog-2000.pdf. Slides at https://web.archive.org/web/20030403102459/https://sutherland-hdl.com/papers/2000-HDLCon-presentation_Verilog-2000.pdf.

Stuart Sutherland. 2017. *RTL Modeling With SystemVerilog for Simulation and Synthesis.* Sutherland HDL, Tualatin, OR, USA.

Stuart Sutherland, Simon Davidmann, and Peter Flake. 2006. *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling* (second ed.). Springer Science+Business Media, New York, NY, USA. https://doi.org/10.1007/0-387-36495-1

Donald Thomas and Philip Moorby. 1991. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, Dordrecht, Netherlands.

This book is the first public description of the Verilog language. There have been five editions: 1991, 1995, 1996, 1998, and 2002. The second edition included a DOS diskette containing examples from the book and a Verilog simulator. The third through fifth editions included a CD-ROM with examples and a simulator.

Donald Thomas and Philip Moorby. 1995. *The Verilog Hardware Description Language* (second ed.). Kluwer Academic Publishers, Dordrecht, Netherlands.

The Foreward by C. Gordon Bell gives a nice summary of the Verilog market as of 1995.

Mike Turpin. 2003. The Dangers of Living with an X (bugs hidden in your Verilog). In *Synopsys Users Group Conference* (San Jose, CA, USA, 2003-03-17/2003-03-19) (*SNUG San Jose 2003*). Synopsys, Mountain View, CA, USA, 1–34. Archived at https://web.archive.org/web/20150510162606/http://infocenter.arm.com/help/topic/com.arm.doc.arp0009a/Verilog_X_Bugs.pdf.

Ernst G. Ulrich. 1969. Exclusive Simulation of Activity in Digital Networks. *Commun. ACM* 12, 2 (Feb.), 102–110. https://doi.org/10.1145/362848.362870

Moshe Y. Vardi. 1995. An Automata-Theoretic Approach to Linear Temporal Logic. In *Logics for Concurrency : Structure versus Automata* (Banff, Canada, 1995-08-27/1995-09-03) (*VIII Banff Higher Order Workshop*). Springer-Verlag, Berlin, Heidelberg, Germany, 238–266. https://dl.acm.org/doi/10.5555/239519.239527. Archived at https://web.archive.org/web/20160512210711/https://www.cs.rice.edu/~vardi/papers/banff94rj.pdf. Volume 1043 of Lecture Notes in Computer Science.

Srivatsa Vasudevan. 2016. *Practical UVM.* CreateSpace, Scotts Valley, CA, USA.

Verisity Design. 2002. *eRM : e Reuse Methodology.* Verisity Design. Archived at https://web.archive.org/web/20021009225113/http://www.verisity.com/products/erm.html.

Frank Weiler. 2003. *DVCon.* Accellera, Napa, CA, USA. Archived at https://web.archive.org/web/20030408023416/http://www.hdlcon.org/geninfo.html.

Ian John White. 1975. *A digital systems simulator.* Ph.D. Dissertation. University of Bradford, Bradford, West Yorkshire, UK. Library record at NON-ARCHIVAL https://catalogue.brad.ac.uk/record=b1220417~S1.

Dyson Wilkes and M. M. Kamal Hashmi. 1999. Application of High Level Interface-Based Design to Telecommunications System Hardware. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference* (New Orleans, Louisiana, USA, 1999-06) (*DAC '99*). Association for Computing Machinery, New York, NY, USA, 778–783. https://doi.org/10.1145/309847.310057