J Electron Test
DOI 10.1007/s10836-014-5449-5

Verilog HDL Simulator Technology: A Survey

Tze Sin Tan · Bakhtiar Affendi Rosdi

Received: 27 August 2013 / Accepted: 5 May 2014
© Springer Science+Business Media New York 2014

Abstract Digital logic design has become increasingly complex in multi-billion transistor VLSI. In order to model such a circuit, Verilog has emerged to be one of the most widely used Hardware Description Languages in digital VLSI design. Verilog's ability to model a circuit at different abstraction levels makes it a preferred mode of design entry starting from the conceptual stage till obtaining the final gate level netlist. At every stage along the development, simulations are inevitably needed to verify circuit correctness. Various forms of software and hardware assisted digital logic simulators were developed ever since the adoption of Verilog in the industry. With increasing circuit design density and complexity, major simulator vendors are struggling to improve simulation throughput in order to maintain quality and engineering cost of product development. Researchers are actively studying new improvements and approaches to keep simulators relevant. This paper surveys the research activities in Verilog simulator technologies as well as comparing every technology's strengths and weaknesses.

Keywords Verilog HDL · Digital logic simulator · Hardware assisted simulator · Verilog compiler

Responsible Editors: V. D. Agrawal and A. P. Ambler

T. S. Tan (✉)
Altera Corporation (M) Sdn Bhd, Penang, Malaysia
e-mail: tts12_eee107@student.usm.my

B. A. Rosdi
Universiti Sains Malaysia, Penang, Malaysia
e-mail: eebakhtiar@usm.my

1 Introduction

The introduction of Moore's Law since the early 1970s has driven electronic circuits to evolve from only a handful of transistors in a system to many billions of transistors in an Integrated Circuit (IC) [8]. This advancement comes with skyrocketing engineering cost from generations of transistor miniaturization. Therefore, iterations of design change must be minimized. Electronic circuit design involves component selection and making the connections in between them. In the early days, the entire circuit was drawn on paper or a whiteboard with calculations executed manually. The complexity of digital ICs nowadays renders this practice irrelevant. Instead, circuit design is highly dependent on Computer Aided Design (CAD) tools in ensuring an IC is correctly designed before manufacturing commences.

Design specification is the first step in an IC design cycle where the feature set is defined. Then, behavioral models can be created to perform simulations to determine the best design parameters. This is the earliest Hardware Description Language (HDL) entry point in an IC development flow. At this conceptual stage, HDL is written at behavioral level representing architectural data flow. Models written in HDL are expected to be simulated on HDL compliant simulators. The intention at this stage is to prove design viability. The next stage is where individual modules are coded at Register Transfer Level (RTL). This is synthesized later into connections of available components offered in the technology library. The library includes commonly used primitive components such as flip-flop, logic gate, memory, phase lock loop (PLL), etc. The conversion output is known as gate level (GL) equivalent circuits. Timing specifications are provided as Synthesis Design Constraint (SDC) to the tool in order to optimize the GL based on target performance. HDL simulation using the GL netlist is carried out to verify GL equivalency by comparison with behavioral and RTL models.

The IC fabrication process is not flawless across dice. Imperfection results in random defects that are statistically distributed across manufactured parts. These parts are non-functional, unreliable, and operate outside of specifications. Detecting and preventing bad part shipment are important to meet quality requirements. For Very-Large-Scale Integration (VLSI) digital logics, Design For Test (DFT) is a necessary step in synthesis flow to insert DFT circuitries to enable efficient production testing. A DFT inserted GL netlist is used to generate production test programs. Components in an IC are tested structurally through Automatic Test Pattern Generation (ATPG) and Built-In Self Test (BIST) [17]. Similar to functional verification, production test program simulations are carried out using HDL simulator. Figure 1 summarizes the design processes discussed thus far in a typical VLSI development cycle. Note that some processes are iterative where they have to be repeated to make corrections and perform fine-tuning. Behavioral verification, functional verification and testing, logical verification and testing, and layout verification are major processes where simulations are carried out. The same circuit is represented by different abstraction levels at these stages. Simulations are carried out to ensure circuit behavior consistency as it passes through the design cycle [65].

Verilog is one of the dominant HDLs. Selecting a suitable simulator improves verification efficiency. However, terminology and the underlying simulator construction remain little known to even seasoned Verilog users. In this paper, we provide a survey of technologies adopted in modern digital logic simulator constructions. Focus is given to those that support Verilog as their primary input. A few choices of simulator are available, often from the same vendor, for circuit designer selection in order to meet quality, performance, and cost requirements. We also look at active research that drives current simulator direction. In the next section, we begin by giving a brief introduction to Verilog HDL. This is followed by an explanation of simulators and their evolution. In the subsequent two sections, we focus on discussing software-based simulators and hardware assisted simulators. A comparison between these simulators is made next. Lastly, we conclude the discussion in this paper and make a suggestion for future work to extend simulator usefulness.

2 Verilog HDL

2.1 Introduction

Before HDL was introduced in the mid-1980s, circuit design was mainly carried out using schematic entry. In order to verify circuit behavior, there was a need to model electronic circuits. There was no single widely adopted solution until the emergence of HDL, such as Verilog.

Verilog was created in 1984 by Phillip Moorby for Gateway Design Automation [12]. Various alternatives were available, such as Very High-Speed Description Language (VHDL) [22, 28], which was widely adopted during the 1990s. Introduction to VHDL can be found in books and papers [27]. The first formalization of Verilog in the public domain appeared as IEEE Standard 1364–1995, commonly

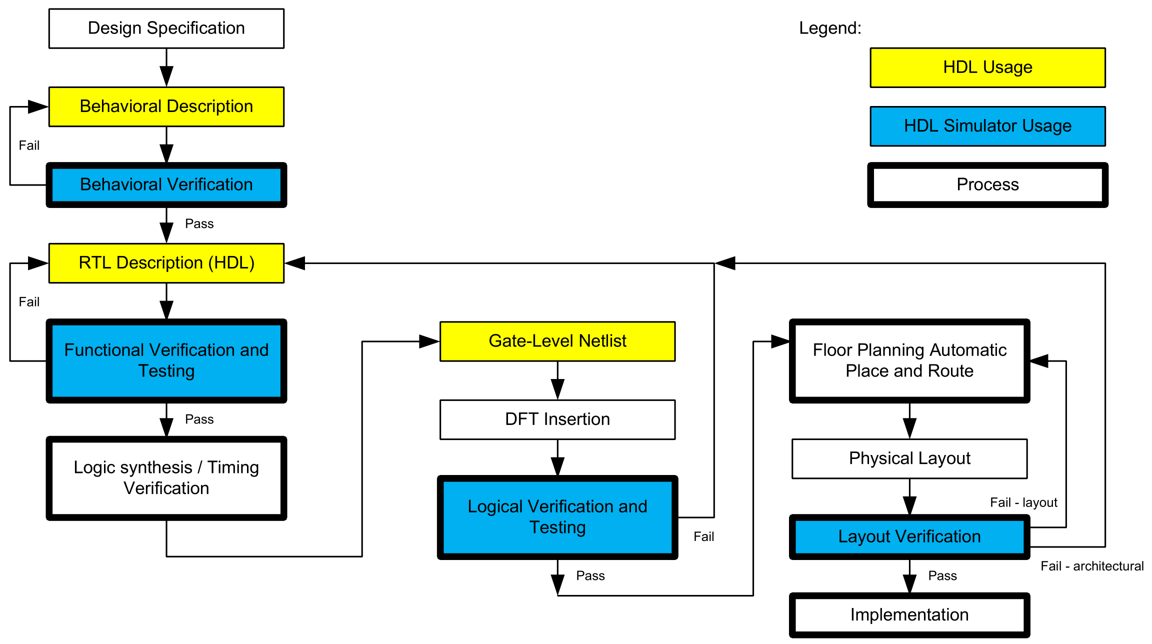

Fig. 1 Typical VLSI design flow [51]

known as Verilog–95 [30]. A major extension was added as Verilog-2001 [31, 60] to address a few shortcomings. These include VHDL style *generate-endgenerate* for multiple instance generation and syntactical simplifications. The last update, Verilog-2005 [32] was introduced to make minor corrections. From then onwards, Verilog has become part of System Verilog defined in IEEE Standard 1800 [29].

Starting as a modeling language, Electronic Design Automation (EDA) vendors were quick to realize HDL's usefulness as a primary design entry method. Thus, a subset of the language from which equivalent hardware can be easily inferred is identified as the Register Transfer Level (RTL) of the language. An automated RTL to GL synthesis tool was born. As VLSI design became increasingly complex, HDL gained a foothold as a primary design entry method from the mid-1990s.

2.2 Verilog Structure

It is not the intention of this paper to provide full details of Verilog capabilities. A few key features and examples are given so that readers can gain a general understanding of the essence of Verilog capabilities. Readers are encouraged to refer to IEEE sources [29–32] and books [12, 51] for more information.

Verilog is created to model digital circuits. Syntactically, Verilog superficially resembles C Language which was widely used in computer programming in the last century. Structurally, Verilog has a hierarchy following typical circuit design practice. A top level module encompasses sub-modules, which in turn consist of instances of less complex sub-modules till a level where the modules can be easily managed.

Circuits are encapsulated in Verilog modules. In each module, a circuit model is represented by multiple concurrent statements. The statements can be coded in different abstractions, such as built-in gate primitives, look-up tables, continuous assignments, and procedural blocks. Built-in gate primitives consist of commonly used pre-defined logic gates (*and*, *or*, *xor*, ...), transistors (*nmos*, *bufif*, ...) and their combinations. Look-up tables are constructed using User Defined Primitives (UDP) in the Verilog context. UDP has the capability of modeling both combinatorial and sequential logics. Gates and UDPs are primarily used in the GL netlist. Continuous assignments are used to describe simple combinatorial logics, both Boolean and arithmetic equations. Finally, procedural blocks are used to describe the functional behavior of a sub-circuit at a higher level of abstraction. Figure 2 illustrates an overview of the Verilog modeling structure. The block diagram on the left shows the hierarchical relationship of modules, whereas different types of coding abstraction choices are depicted on the right. When interpreting a Verilog model to understand circuit behavior, it is important to remember that every statement runs in parallel.

3 Simulators

3.1 Introduction

Before a new IC design is sent for fabrication, thorough verification is necessary to ensure design correctness and manufacturability. Digital circuit verification is mainly carried out on logic simulators. There can be many ways to model a circuit. The ability of Verilog to represent a circuit from behavioral to transistor level makes it an ideal input format for logic simulators.

3.2 Function of Simulator

A simulator is flexible to emulate a wide range of circuits. Though functionally accurate, a simulator executes at a few orders of magnitude slower compared to the actual circuit.

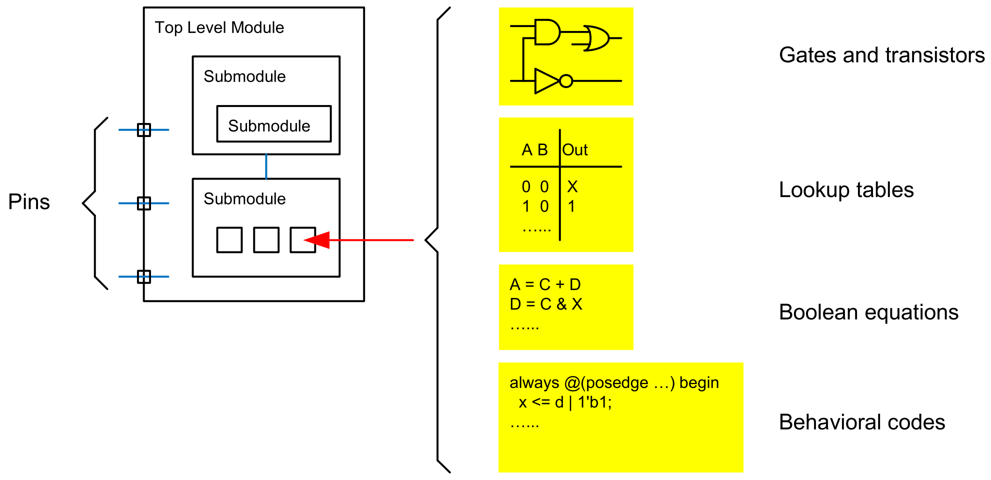

Fig. 2 Verilog modeling structure

Thus, in simulations, test cases are carefully selected to represent the extreme usages of circuits in final applications.

A typical simulation session has the circuit model encapsulated in a test bench. A verification framework is shown in Fig. 3. The test bench consists of stimuli generator, golden response generator, and response analyzer. The stimuli generator provides stimuli to the circuit model. The golden response generator infers the expected circuit response based on the stimuli received. While simulating, the modeled circuit's response is compared with the golden response by the response analyzer. Any mismatch is logged and flagged for the designer's action.

Figure 4 elaborates the verification using a 4-bit binary counter as an example. In order to keep this discussion succinct, supporting statements such as signal declarations are omitted. The module, *mycounter* on the right of the figure, consists of a *clk*, an asynchronous *rst*, and a bus of 4-bit binary counter output. The counter is reset to 0000_b by asserting *rst*. Upon de-assertion of *rst*, counter output increments by 1 at every rising *clk* edge. Listed in the same figure on the left, *my_testbench* is the test bench.

With reference to the text block on the left in Fig. 4, circuit model, *mycounter*, is instantiated as *dut*. In this example, stimuli generator, golden response generator, and response analyzer are embedded within a single *initial* block. The *dut* response is examined exhaustively from 0000_b to 1111_b. As soon as a mismatch is detected, simulation stops with an error message displayed indicating the cause of failure. On the other hand, the verification passes if the *dut* responses match all golden responses.

3.3 Evolution of the Digital Logic Simulator

A few classes of simulators were developed over the past decades to simulate digital logics [26]. The earliest adoption of a circuit simulator in general purpose computers was the Simulation Program with Integrated Circuit Emphasis (SPICE) created in the 1970s. SPICE was originally designed as a general purpose circuit simulator. Simulation semantics are time driven. At each time step, every circuit node is numerically evaluated for convergence. Then, the simulator advances to the next time step and repeats the evaluation cycle. This is a computation-intensive process. As digital circuits gained acceptance and dominated VLSI designs, SPICE's shortcomings surfaced. Throughput is exponentially affected when simulating large circuits. Often the numerical details of voltage and current at every time step are unnecessary for digital circuit functionality verification. Enhancements had been developed on top of SPICE by various vendors in order to improve simulation efficiency [56]. Nevertheless, the numerical method of evaluation ran out of steam at high gate count digital ICs.

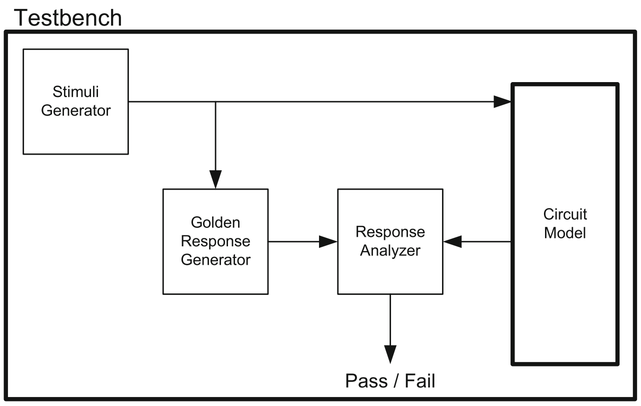

Fig. 3 Test bench-based verification framework

Digital circuit design called for a different level of focus as opposed to SPICE. Early digital logic simulators [21, 46] were developed to answer these needs. Some simulators offered more advanced features, such as delay modeling [16]. As HDL gained a foothold in VLSI design, new HDL simulator architecture was proposed. Before Verilog published its formal scheduling semantics, proposals were made to simulate digital logics by iterations of evaluation till the circuit stabilizes [23, 58], reminiscent of numerical method origins. Event driven simulators represented a major improvement in digital logic simulation efficiency [10]. Instead of evaluating every circuit node at every time step, only circuit nodes that have their inputs changed are evaluated. Variants of simulators have been created subsequently.

The simulation scheduling semantics of Verilog are described as a stratified event queue as summarized in [20, 31]. All modern simulators conform to the standard semantics. A flattened Verilog netlist is essentially a collection of concurrent statements. As an event triggered simulator, a statement is only evaluated when triggered, i.e. one or more of its inputs change. There is a global timer keeping track of the current time step, and an event queue points to the statements scheduled to be evaluated. The queue is sorted chronologically according to which statement is scheduled to have its output updated. After evaluation, the event is removed from the queue list whereas new events are inserted if the resultant output triggers another statement into action. Having evaluated all statements scheduled for the current time step, non-blocking assignments are updated last. The global timer is advanced when the head of the event queue points to a statement scheduled for a later time step. The simulation ends when the event queue becomes exhausted or is forcibly terminated upon encountering a system command such as "*$finish*."

Modern digital logic simulators are pre-dominantly event-driven. It is important to distinguish a HDL as a

Fig. 4 A test bench and a circuit model of 4-bit binary counter

```
module my_testbench;

…

// Instantiate the circuit model
mycounter dut(clk, rst, counter);

initial begin
  // Stimuli generator
  clk = 1'b0;
  rst = 1'b1;
  #10 rst = 1'b0;

  // “i” is stimuli pointer as well as golden response
  // generator
  for(i = 0; i < 16; i = i + 1) begin
     #10 clk = 1'b0;

     // Response analyzer
     if(counter != i)  begin
        $display(“FAIL: Expect %d, get %d”, i, counter);
        $finish;
      end
     #10 clk = 1'b1;
   end

  $display(“PASS”);
end
```

```
module mycounter(clk, rst, counter);

...

always @(posedge clk or posedge rst)
  begin
    if(rst)
      counter <= 4'h0;
    else
      counter <= counter + 4'h1;
  end

...
```

language used to describe a circuit, versus digital logic simulator which mimics circuit behavior in a computing platform. In subsequent sections, we look at simulator technologies with Verilog in mind. However, the same technique is applicable for other HDLs. We will discuss each class of simulator based on the classification shown in Fig. 5. Digital logic simulators are separated into software-based simulators and hardware assisted simulators. This classification is made to yield minimum overlaps in between classes, so that a few real world simulators falling in each class are available for comparison. Software-based simulators are further broken down into interpreter, compiled-code, gate level, and parallel computing. Parallel computing is not strictly a class of simulator on its own, but rather a feature that can be associated with other software-based simulators for ongoing improvements. It is examined more closely later in this paper. As for hardware assisted simulators, FPGA synthesis and emulators are two options.

The classification of simulators may not be perfect because some hybrid simulators are constructed combining different technologies in order to achieve the best performance metrics in the field. For clarity, discussions in this paper look at the main differentiator that sets common Verilog simulators apart.

4 Software Simulators

4.1 Introduction

As the name implies, a software simulator is software running on general purpose computing infrastructure, either on a workstation or a server. A few sign-off grade simulators are offered by major EDA vendors, such as VCS from Synopsys®, Questa from Mentor Graphics®, and Incisive from Cadence®. Key quality metrics that differentiate a simulator from others include speed, runtime memory footprint, circuit size, and supporting features. Depending on the construction of the simulator engine, simulation speed varies. They can be separated into four groups, i.e. interpreter, compiled-code, gate level, and parallel computing simulator.

Interpreter-based simulators load in a circuit model into an internal data structure at the beginning of a simulation session. Simulation is carried out by the source program mimicking the modeled circuit behavior. A compiled-code simulator, on the other hand, goes through a compilation phase [33]. The Verilog model is translated into standard programming source code, such as C++. This is followed by a software compilation stage that converts the source code into an executable representing the simulator. Simulation commences upon initiating the compiled executable. A gate level simulator translates the HDL into a GL netlist before moving into the simulation phase. The

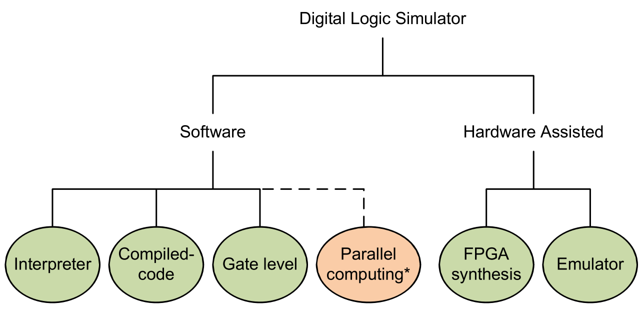

Fig. 5 Simulator classification. * Note: Parallel computing is not a simulator but a key technique that improves simulator performance

synthesis requirement limits usage of this simulator to generate synthesizable codes. Both interpreter and compiled-code simulators are available. Parallel computing, on the other hand, is an enhancement currently the subject of research by all software simulators to improve simulation throughput. Large simulations are separated into smaller simulation tasks running on multiple processors.

A HDL does not dictate how a simulator should be constructed aside from suggesting a formal simulation event scheduling mechanism. The standardized event scheduling ensures consistent results can be obtained from different simulators. Simulators created by different vendors are different in their simulation efficiency. One example is the runtime memory requirement. Some simulators need more runtime memory than others when simulating the same circuit. Some impose restrictions on circuit density, such as the largest size of bus, arrays, etc. A simulator is incomplete without features to help circuit designers identify circuit bugs. Internal circuit node tracing and source code stepping are some common tools supplied along with the simulators.

Despite the differences among these simulators, they are fundamentally software-based. General purpose computing infrastructure is all that is needed to carry out simulations. In the next sub-sections, we look at active research into enhancing this class of simulators.

4.2 Relevant Works

4.2.1 Interpreter Simulator

In order to simulate a piece of digital logic, the HDL which describes the circuit has to be loaded and understood by a simulator. An interpreter simulator carries out these steps. It is a software program running on general purpose computers. Aside from custom made circuit entry software [52], the same technique has been used by EDA companies for HDL-based digital simulators. The flow starts by loading, interpreting, and transforming HDL models into the internal data structure [4, 51]. Simulation is carried out by referencing the internal data. In this regard, the flow is similar to its SPICE ancestor.

An interpreter simulator is a pre-compiled software program. HDL source files are always loaded upon initiating a simulation session. This is similar to scripting language and analogous to software programming, such as Java [24], TCL, and VBScript. There is no compilation step involved to transform source codes into machine codes, but rather software that interprets source codes during runtime.

Despite being the oldest digital logic simulator class, it is still the dominant HDL simulator class. Similar to its software scripting counterpart [45], this interpreter is inefficient in computation. Lewis [36] pointed out that frequent traversing of internal data structure is the main contributor to slowdown. However, many simulator vendors have extended interpreter simulator usefulness by offering parallel computing. This will be discussed in Section 4.2.4.

4.2.2 Compiled-Code Simulator

Computer programs are written in text-based source codes. Then some are executed through interpreter programs during runtime, such as Perl and TCL, whereas others like C++ and Pascal are compiled into machine codes before being executed, independent of the compiler and source files, as a standalone program. Compiled-code software offers better runtime efficiency in general.

The compiled-code software approach is adopted in logic simulator construction. Instead of loading HDL source files during runtime, the circuit along with its test bench is first translated into common programming source code such as C++. This is then compiled by a standard software compiler in order to generate a simulator executable. The source files are no longer referenced in simulation runs. This class of simulator is known as a compiled-code simulator. As discussed in [36, 37], a compiled-code simulator has the advantage that the netlist and its data structure do not change during simulation. Machine codes are generated to access the data directly. Compared to frequent traverses of internal data structure in an interpreter simulator, a compiled-code simulator spends much time on actual model evaluation. The resultant simulator can be made reusable for multiple test benches by reading files for stimuli and golden results during runtime. The one-time effort expended on compilation is negligible. Upon formalization of Verilog simulation semantics, compliant compiled-code simulators were developed. Improvements have been proposed for this class of simulator to enhance its efficiency. [47] transforms logics into a look-up table before compiling them into a simulator. By only evaluating standard look-up tables, a highly optimized evaluation engine can be created to simulate the model.

A compiled-code simulator does not necessarily run on general purpose microprocessors. Co-processors are common-place nowadays to support other functions in a computing infrastructure. A Graphics Processing Unit (GPU) is a high performance microprocessor that can be used to support HDL simulation. Qian and Deng [53] generates a compiled-code simulator running on GPU. The presence of GPU as a co-processor in a general purpose computing system can also be treated as hardware assisted computation. This will be looked at in Section 5.

4.2.3 Gate Level Simulator

There are a few categories of coding abstractions supported by Verilog as described in Section 2. Having a simulator supporting full Verilog constructs may not be necessary in all scenarios. In most cases, HDL is primarily used as RTL in the design step and subsequently transformed into a GL netlist. There is only a small subset of Verilog needed in order to support GL simulation. Components in the form of logic gates can be easily evaluated in a well-defined duration using native instructions of the CPU.

Some simulators, therefore, are created to simulate a GL netlist with better efficiency as it is possible on a fully compliant Verilog simulator. The construction of the simulator can be either an interpreter or compiled-code [47, 54]. However, different from both simulators discussed in previous sections that support full Verilog compliancy, this class of simulator is restricted to the GL netlist. This makes it a class on its own. Gu and Vishkin [25, 42, 43] are examples that require conversion of behavioral codes into the GL netlist before distributing simulation tasks across multiple processors. Similarities in evaluating GL components allow these proposals to focus on tackling more difficult problems of workload balancing. This will be discussed in the following section.

4.2.4 Parallel Computing

The Verilog model is constructed from concurrent statements. Every statement executes in parallel. Early computers could only run single-threaded applications. Therefore, though Verilog statements are concurrent, they are evaluated one at a time. Nonetheless, the scheduling mechanism of a simulator presents a virtual parallel evaluation result to circuit designers. Modern general purpose computers are built to support multi-threaded applications in a multi-processor (including multi-core in this paper) system. This capability maps fairly well to the concurrent nature of Verilog constructs [13–15]. In fact, parallel logic simulation was conceptualized before HDL formalization [64]. A more recent application using Bluespec System Verilog [49] looks at extending Verilog into higher levels of abstraction. It addresses the need to carry out modeling at system level by improving HDL capabilities.

On a general purpose computer, multi-core computers are designed to run independent software threads. For example, an Operating System (OS) may allocate a word processor to run on one processor core, while a calculator is run on the other. These applications do not interact with each other. Different from a word processor or calculator, simulation threads are inter-dependent. Synchronization is needed when data needs to be transferred across threads. Any need to synchronize the threads results in inefficiency. It is unlikely that different threads would take the same amount of execution time to reach a checkpoint requiring synchronization. Thus, the faster executed thread will need to be put on hold while waiting for its sister thread to complete its execution.

In the case of Verilog modeling, the number of statements is usually many times more than processor cores available to simulate them. Though statements are concurrent in behavior, some statements consist of a collection of sequential statements, such as those in procedural blocks (*always*, *initial*). Every collection of code blocks is known as a *Logical Process* (LP), which is the smallest unit that can be distributed across processors. The order of execution is important so they should be run in the same thread. Therefore, for parallel processing, researchers focus on higher-level abstraction, such as subsystem level closer to the top-level module. Every subsystem simulation is allocated as a standalone thread that runs independently from others. A general overview of simulation flow is illustrated in Fig. 6. This is typical of a *Parallel Discrete Event Simulator* (PDES). The main loop is entered at every time step until all queued events are executed or stopped forcibly (such as via *$finish*). In the main evaluation loop, each parallel thread processes the assigned events. The inner loop is repeated if there are more events at the same time step. By splitting the subsystems closer to the top-level module, the need for information transfer across subsystems is minimized. There are proposals to allow individual threads to execute using local time steps [39, 57]. However, there can be connections among the subsystems. Whenever a signal propagates to the subsystem in another thread, the destination subsystem must have its input updated. There is a possibility of causality break because every thread is not exactly balanced in the execution time. The destination subsystem could have been simulating beyond the originating subsystem's time step. The latter simply takes longer to complete heavier simulation load. A usual solution is to rewind the former thread's time step so that both subsystems are resynchronized, which is a time warp mechanism.

4.3 Discussion and Perspectives

A software-based simulator requires a host to execute simulations. This is either a general purpose computing workstation or a server. A microprocessor serving this purpose is not optimized for HDL simulations. Every Verilog statement has to be decomposed into fundamental operation steps offered by the microprocessor. The elaboration results in steps that require tenths to hundredths of machine cycles to completely evaluate an equation. Note that Verilog supports four signal states (*0, 1, X, Z*). This needs to be at least two bits wide to represent each node state. A general purpose CPU has native logical operations restricted to single bit data types *0* and *1*. For a Verilog equation such as $A = B\&C$, it has to be evaluated in steps. $A = 0$ if either B or C is *0*. Otherwise, when both B and C are not Z, $A = B$

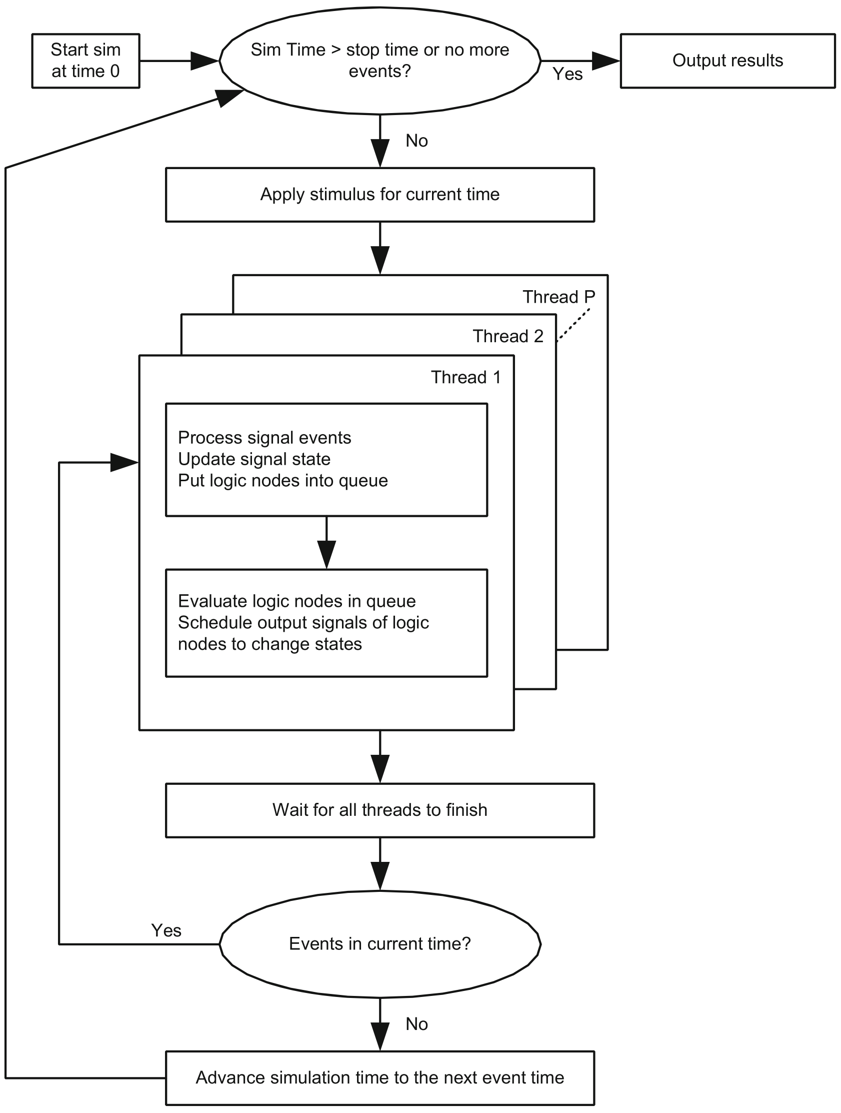
Start sim at time 0
Sim Time > stop time or no more events?
Yes
Output results
No
Apply stimulus for current time
Thread P
Thread 2
Thread 1
Process signal events
Update signal state
Put logic nodes into queue
Evaluate logic nodes in queue
Schedule output signals of logic nodes to change states
Wait for all threads to finish
Yes
Events in current time?
No
Advance simulation time to the next event time

Fig. 6 Multi-threaded simulation mechanism [13]

if C is *1*, or $A = C$ if B is *1*. If all previous conditions are not met, $A = X$. There is not much improvement that can be achieved on a single-threaded fully compliant interpreter and compiled-code simulators. Both are limited by the capabilities of the host CPU in carrying out model evaluations. Decades of algorithm optimizations have exhausted the known software optimization techniques.

Advancements in computing infrastructure allow multi-threaded capabilities. High-speed networking makes tightly coupled computer systems available as a server pool. It is possible to partition the simulation of Verilog models into segments for multi-processor systems such as discussed in Section 4.2.4. However, this requires a highly customized HDL compiler to predict the best partitioning during compilation. The fundamental of partitioning involves segregating sub-modules or LPs at a lower level, and assigning them to different processors. The assignment can be unbalanced. During simulations, the worst case happens when there are frequent signal transfers across sub-modules. This results in rewinds (or rollbacks) that invalidate already completed portions of simulation. The frequency of cross partition signal transfer not only depends on the circuit itself, the stimuli driven from test bench plays equal important role. Thus, inefficiency of a poorly partitioned design potentially offsets the benefits of parallel computing. Several initiatives are trying to tackle this problem [34, 63, 66].

The challenge lies in creating intelligent partitioning software that can predict simulation workload before simulation commences. Whenever causality is violated due to a cross module signal transfer, simulation of the destination module rewinds to the time step of the originating module. This implies the need to store a history of simulation status. The runtime memory requirement is at least a few times larger than a single-threaded simulator. Moreover, it is a waste of computing power to simulate incorrect predictions which eventually are discarded. There is a proposal to reduce rollbacks by hierarchically partitioning a circuit into units [66]. Each unit has its own local structure of input queue while eliminating output queue. These units communicate with each other through time-stamped messages. Instead of rolling back the entire circuit whenever a causality violation is detected, each unit rolls back locally if this is all that is needed. Nonetheless, this is a complex simulator architecture. Kim et al. [34] addresses the difficulty by allowing an LP to commence evaluation on partial information. This relieves the scheduler from stalling all processors trying to identify suitable LP for evaluation. Wang et al. [63] takes a different approach by partitioning long simulations into time-based segments. It focuses on low level simulation such as the GL. Each segment runs independently from the others. A corresponding reference simulation snapshot at a higher level of abstraction, hopefully simulating faster, must exist to supply initialization to each time partitioned segment.

One variant of parallel simulation requires synthesis prior to simulation [43]. Different from [63], a reference simulation snapshot is not needed. The process aligns with the typical design flow where the GL is always generated anyway. The method improves simulation speed. However, synthesis is an additional step. It is likely that a VLSI consists of behavioral models such as PLLs and memories. In these cases, GL simulation would have to resort to co-simulation of a mixed simulator type, which is often tricky. Besides being restricted to RTL, it is inconvenient to frequently go through the full synthesis process only for simulations, especially during the early design phase. This implies the need to use different simulators at different design stages. Users are required to become familiar with multiple simulation tools in order to compare simulation results. Loss of mixed abstraction modeling in the GL is an additional barrier in debugging.

5 Hardware Assisted Simulators

5.1 Introduction

Before the creation of the HDL, the idea of complementing a software-based circuit simulator with specialized hardware was introduced for digital circuit verification [5–7]. Soon after the creation of Verilog, it was quickly realized that a subset of the language was synthesizable. The use of RTL in IC design flow is now a main reason for adopting Verilog. As discussed in Section 4, software simulators are a few orders of magnitude slower compared to the real chip. Thus, transferring part of the model into configurable hardware is a natural process. Models running on hardware have the same logical behavior as the netlist. There are two main classes of hardware assisted simulators, i.e. FPGA synthesis simulators, and emulators.

A FPGA synthesis simulator has part of the netlist synthesized on a FPGA. The behavioral portion of the codes including the test bench is executed on a general purpose computing host. The host interfaces with the target FPGA to supply stimuli and retrieve computation results according to test bench instructions. The emulator, on the other hand, has the netlist synthesized on a hardware platform, but leaves out the Verilog test bench. Instead of relying on the test bench, the emulator is a virtual IC of the modeled circuit running applications it is designed to serve. These two classes of simulator are discussed in the following sections.

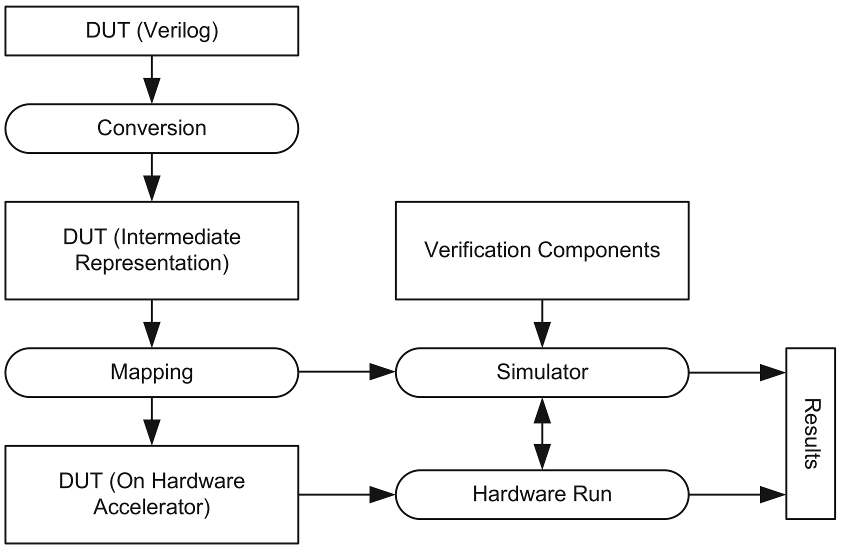

Fig. 7 Simulation flow of FPGA synthesis simulator [51]

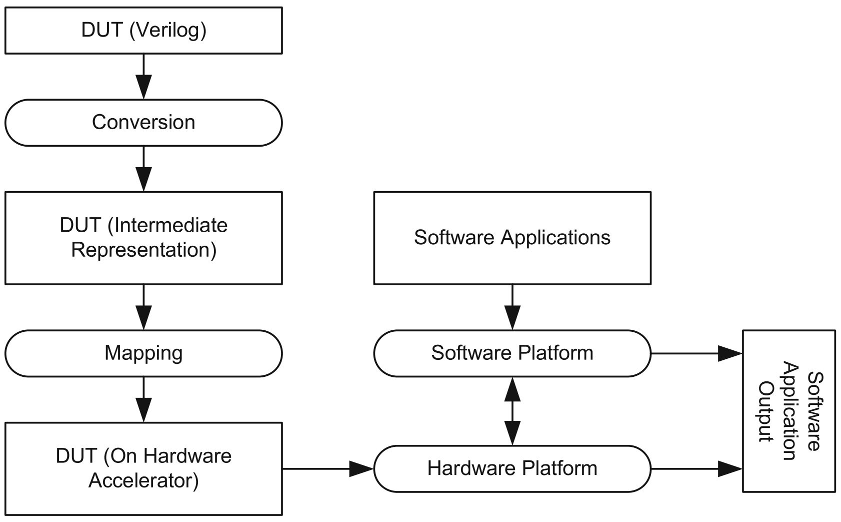
DUT (Verilog)
Conversion
DUT (Intermediate Representation)
Mapping
DUT (On Hardware Accelerator)
Software Applications
Software Platform
Hardware Platform
Software Application Output

Fig. 8 Simulation flow of emulator [51]

5.2 Relevant Works

5.2.1 FPGA Synthesis Simulators

A Field Programmable Gate Array (FPGA) is an IC consisting of customizable lookup-tables, flip-flops, SRAMs, multipliers, phase lock loops (PLL), and other common digital System-on-Chip (SoC) building blocks. RTL codes can be synthesized as connections of these primitive building blocks and loaded into a FPGA. An example of Arithmetic Logic Unit construction is demonstrated in [68]. It is functionally equivalent to an ASIC designed using the same RTL source. FPGAs are also widely used as a vehicle to prototype complex experimental designs, such as artificial intelligence processors [18, 48] where computation-intensive algorithms can be implemented. Lewis et al. [38, 46] gives an explanation where software-based algorithms are translated into a FPGA for accelerated execution.

This technology is proposed as the test platform for the Verilog netlist under verification. Some solutions are more effective than others. All of them have to go through a synthesis step as shown in Fig. 7. First, the Verilog source is transformed into an intermediate representation that is target platform independent. Based on the target platform, the synthesizable portion of the codes (RTL) is mapped onto the hardware. Non-synthesizable behavioral codes are run as software on a computing host. A library of verification components is referenced during simulation. Working as a tightly coupled system, a simulation result is generated. A more straightforward method accepts only Verilog UDP instead of RTL [67]. This is especially suitable for a GL netlist because commonly used logic gates can be easily represented by UDPs.

Most of the time it is impossible to synthesize everything into the GL and subsequently fit it into a FPGA. In the verification setup, there are always behavioral codes, mainly within the test bench as verification components. In this co-simulation environment, communication in between a hardware accelerated simulator and its behavioral code hosting computer becomes the bottleneck that limits

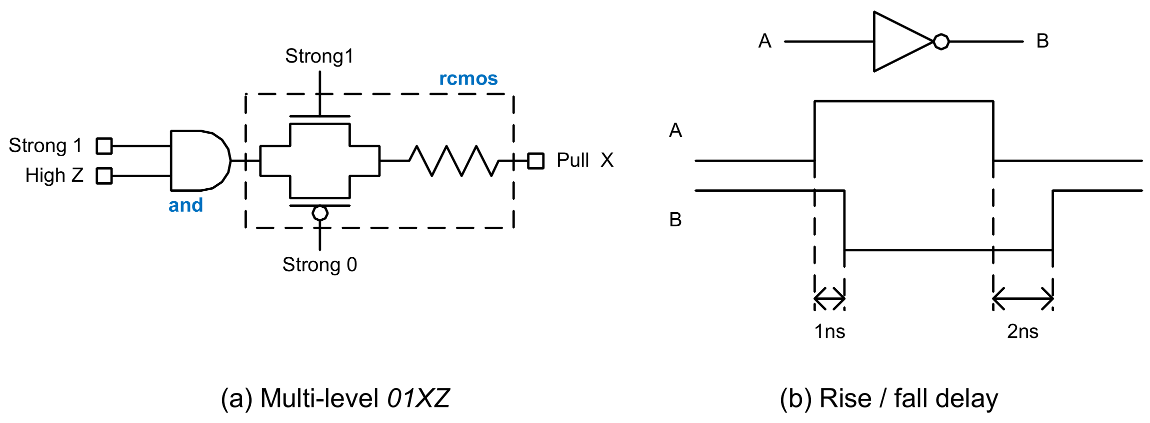

Fig. 9 Multi-level signal and delay

simulation efficiency. Research proposals were put forward to improve the efficiency of co-simulation bottlenecks. All simulators in this class go through a synthesis step to transform the netlist of the DUT into a FPGA. Wageeh et al. [62] retains the test bench on the host computer. Frequent interfacing in between the DUT and host computer is necessary. Maili et al. [40] has an embedded microprocessor on the platform that takes care of behavior models simulation. Instead of the host system, this embedded microprocessor simulates non-synthesizable codes, thereby reducing the need to communicate with the host system. Kim and Kyung [35] translates test bench into RTL for synthesis along with the DUT.

5.2.2 Emulators

Emulators were the first alternative introduced to replace software simulators. In the early days, they were specialized hardware offered by ASIC vendors. Customization through parameters is possible, but limited to the usage offered by the corresponding ASIC.

Early general purpose Verilog simulator architecture was proposed in [11]. Instead of going through synthesis, it has the ability to map Verilog into a dedicated GL and behavioral processors for simulation. After going through the pipelined processor, outputs are generated. This method requires a hardware platform to emulate the circuit under verification. A FPGA is a likely hardware choice [55]. The emulator is cycle accurate but runs at a much lower clock rate than the actual circuit.

There could be confusion over differentiating this class of simulator from FPGA synthesis simulators discussed in Section 5.2.1. Perhaps the ability of an emulator to operate as the modeled circuit at cycle accuracy is its main characteristic. Some emulators can substitute the actual hardware in a system, albeit at a slower clock rate. Figure 8 illustrates simulation flow involving an emulator. Compilation steps till *mapping* is the same as the FPGA synthesis simulator. Note that after *mapping*, the DUT is mapped into hardware. The Verilog source must use only resources that are available on the hardware for the mapping to be successful. Different from a synthesis-based simulator, test bench and other behavioral codes must be separated out as software applications run on the host platform, similar to how a real DUT is debugged.

5.3 Discussion and Perspectives

A hardware assisted simulator requires proprietary hardware from the vendor as a platform to carry out simulations. The platform is programmable according to the circuit netlist. A setup stage to map the circuit into the target platform is non-trivial. Not only does the process vary among platforms, the ability to map the circuit depends on primitive resources available on the hardware. In short, there is a steep learning curve for a user to master hardware assisted simulators.

Emulators are able to bring verification platform closest to the real circuit under development. Its capability of replacing the emulated circuit in the final system enables verification in the field. However, this verification method is different from the test bench-based approach that most hardware circuit designers are accustomed to.

Early emulators were available for specialized systems serving specific niches. This was usually a microprocessor system with common peripherals. Thus, its usefulness was restricted to simulating systems that were pre-defined for the emulators. Works in general purpose logic simulator try to solve the emulators' shortcomings [11, 44]. They called for specialized processors not only for the GL netlist but also for behavioral codes. Building such a system is complex and thus no follow-up research is found. Nonetheless, commercial solutions for this class of simulator are offered by major EDA vendors. Exact implementation is proprietary and obscure to the general public.

On the other hand, a FPGA system offers configurability via the synthesis of RTL. This is much more flexible than emulator systems, but has its own restrictions. It is obvious that a large design that requires more gate count cannot be fitted in a small FPGA. Though it is possible to construct a target simulator using a pool of FPGAs, this is not an easy way to grow according to netlist size. Moreover, native circuit blocks used in the modeled system must be available in the target FPGA. For instance, a FPGA without a dedicated multiplier cannot map such logic effectively. Synthesizing RTL into a FPGA is a specialized field in its own right. Users need to be familiar with logic synthesis in order to map a system successfully and satisfy timing requirements. Circuits that involve unpredictable runtime behaviors, such as cross domain signals, cannot generate consistent results in different simulation runs.

It is understandable that emulators can be easily confused with FPGA synthesized simulators at times. Furthermore, some emulators are constructed by FPGAs internally [1]. The main differences being,

- An emulator is cycle accurate with the capability to substitute the actual hardware in a system, albeit at a slower clock rate
- An emulator does not necessarily require a test bench. Or, the test bench is decoupled from the emulated netlist
- A FPGA synthesized simulator is able to absorb behavioral codes though it might not be physically synthesized into hardware. Software/hardware co-simulation is adopted

Table 1 Summary of logic simulator types

Class	Platform	Technique
Interpreter	General purpose computer	Simulator loads HDL, interpret and simulate [52]
Compiled Code	General purpose computer	HDL is converted to C++, compiled, and executed [36, 37, 47, 53]
Gate Level	General purpose computer	Synthesized to a GL for efficient simulation primitives [25, 42, 43, 47, 54]
Parallel Computing	Multi-core computer	Simulation is distributed to processors or computers [13–15, 49, 61, 64]
FPGA Synthesis	Proprietary hardware	Synthesized into hardware. Accept behavioral code [35, 40, 62, 67]
Emulator	Proprietary hardware	Synthesized into hardware [11, 55]

Besides being difficult to use, and there being restrictions in supported logics, these simulation platforms do not provide the same level of internal node visibility. Verilog supports logic states of *0*, *1*, *X*, and *Z* with eight levels of strength. These simulators report only *0* and *1*. *X/Z* behavior is inconsistent if allowed. A similar thing happens to delay models [19] that are common in post-layout GL netlists supplied using the back-propagated Standard Delay Format (SDF) file [50]. Two examples are shown in Fig. 9. In (a), the *AND* gate output is an *X*. The *X* is reduced in strength after passing through a resistive CMOS pass gate. Not only are emulators and synthesized simulators incapable of *X* modeling, signal strength information is lost. It is also impossible to synthesize arbitrary delays as shown in (b). Delay modeling is useful in verifying SDF back-annotated GL netlists, but there is no practical way to implement variable and controllable delay elements in hardware for every signal path.

Visibility and controllability over internal nodes offer great debugging advantage in software simulators. Providing such capability is costly in emulator and synthesis-based platforms. A scan-based DFT [17] technique was proposed as a mean to provide visibility and control [41]. Not only is it an expensive alternative, a full scan design cannot be easily achieved, especially in designs with multiple clock domains. It is a complex process to ensure successful loading of internal nodes and resume simulation afterwards. At least this is not a straightforward process in most designs as what engineers would expect from simulators.

6 Comparison of Simulators

Table 1 gives a summary of the comparison of different logic simulators. The classification is based on the discussion in Section 3.3. Table 2 lists speed comparison and examples of commercial simulators based on each technology. Speed is normalized to emulator speed. If a design takes 1s to complete a simulation session on an emulator, the same simulation would take 2000s on a compiled-code simulator. The numbers in this column are inferred from a

Table 2 Speed comparison of logic simulator types

Class	Speed[1]	Resources[2]	Examples
Interpreter	4000	Medium	Cadence Incisive Mentor Graphics Questa
Compiled Code	2000	Medium	Synopsys VCS
Gate Level	< 2000	Low	GPU
Parallel Computing	50	High	Mentor Graphics Questa Synopsys VCS
FPGA Synthesis	2	High	Cadence Palladium Mentor Graphics Veloce
Emulator	1	Low	Cadence Palladium Synopsys Zebu

[1]Speed normalization is inferred from multiple sources [2–4], aimed at giving readers an idea of speed differences in general. It is not to be taken as real numbers for simulators listed in examples.

[2]Resources refer to memory footprint and logic density, for software and hardware assisted simulators respectively.

Table 3 Feature comparison of logic simulator types

Class	Behavioral Support	Delay, *XZ*, Signal Strength Support	UDP	Internal Node Observe & Control	Ease of Use
Interpreter	+	+	+	+	+
Compiled Code	+	+	+	+	+
Gate Level	-	+	+	+	-
Parallel Computing	+	+	+	+	-
FPGA Synthesis	-	-	r	r	-
Emulator	-	-	r	-	-

+ – Supported, r – Partial supported, - – Not supported

few sources [2–4]. They are meant to give the reader an idea of speed difference based on simulator type in general. It does not mean all simulators in a particular category have the same speed ratio compared to other categories regardless of circuit model in the simulation. Every EDA vendor optimizes their simulator design differently. In the same table, the resource column lists relative resource utilization when these simulators are simulating the same circuit.

In general, software-based simulators tend to support all HDL constructs available in the language. As more parts of the simulations are transferred to a hardware platform, it becomes increasingly difficult to retain full HDL support. Emulators, which are closest in architecture to the final circuit, offer almost no debugging capability in preference for speed of verification. This is summarized in Table 3. It is a tradeoff between flexibility and speed.

7 Conclusion

Verilog is one of the major HDLs used in VLSI modeling. In order to ensure design correctness throughout the design stages, logic simulations are inevitably the most important verification method. Mainstream designs rely on software simulators from EDA companies. As the designs are becoming more complex, techniques to reduce simulation test cases have been proposed [59]. At the same time, limitations of single threaded software simulators have started to be studied.

The maturity of multi-processor systems allows natural evolution towards multi-threaded simulators. However, partitioning simulation threads is a complex process. All proposals focus on making the best prediction in the partitioning of a design for parallel simulations. Activity count in a module cannot be predicted with high accuracy before simulation commences. Imbalanced assignment does not help in simulation speedup. Though there are successes in the area under specific conditions, the method has yet to reach general use with expected speed improvements.

Hardware assisted simulators have been developed to improve Verilog simulation throughput. There were technological restrictions on what could be achieved in the early days when off-the-shelf components were limited. However, low-cost prototyping of circuit solutions are readily available through FPGA advancements now. There are drawbacks from this approach that limit its adoption other than in specific industries. The need to synthesize RTL into FPGAs is a huge barrier in addition to the loss of fidelity compared with software simulators.

8 Future Work

Verilog will continue to be among the dominant HDLs in the near future. A few active and recent research projects focus on high-level parallelism to achieve better simulation throughput, be it software or hardware assisted. From the software simulator perspective, [63] is one PDES example that tries to solve LP activation overhead through systematic initialization instead of shuffling through stalled LPs. It is acknowledged that it will take more work for a PDES to be ready for general use. Kim et al. [34] takes a different approach by exploiting the fact that computation-intensive GL simulation has been verified at the behavioral level already. It is possible to utilize behavioral simulation snapshots as references for time partitioned simulation fragments. Each fragment simulates independently starting from a checkpoint saved in the behavioral simulator. Moving onto hardware assisted simulators, Bertacco et al. [9] studies applying GPUs for the task. Cones of logic are partitioned at equal depth to achieve the most balanced loading across all processor cores.

Researchers in the field of Verilog simulators have not produced a widely adopted solution. There are either limitations in acceptable Verilog constructs, or there is a call for unconventional treatment of netlists before simulations. An innovative hardware assisted solution that addresses the existing weaknesses would be of high value for future VLSI designs. The solution must be nearly identical in terms of usage and capabilities to software simulators that verification engineers are accustomed to. This includes the ability to simulate all abstraction levels of Verilog codes from behavioral, RTL, to GL netlists, including delay models, *XZ*, and signal strengths. Standard Verilog semantics must be preserved to yield consistent results compared to software simulators. The ability to grow in response to larger designs can be of great advantage. And, of course, the new solution must run at orders of magnitude faster than software simulators.

Acknowledgments This research was a collaboration in between Universiti Sains Malaysia and Altera, sponsored in part by Malaysian Ministry of Higher Education under Industrial PhD program.

References

1. (2013). http://www.synopsys.com/Tools/Verification/hardware-verification/emulation/Pages/zebu-server-asic-emulator.aspx
2. (2013). http://www.veripool.org/wiki/veripool/Verilog_Simulator_Benchmarks
3. (2013). http://www.mentor.com/products/fv/emulation-systems
4. (2014). http://www.bawankule.com/verilogcenter/simspeed.html
5. Abramovici M, Levendel YH, Menon PR (1983) A logic simulation machine. IEEE Trans Comput-Aided Des Integr Circ Syst 2(2):82–94
6. Agrawal P (1990) Mixed behavior-logic simulation in a hardware accelerator. In: Proceedings of the IEEE custom integrated circuits conference, pp 9.2/1–9.2/4
7. Agrawal P, Tutundjlan R, Dally W (1989) Algorithms for accuracy enhancement in a hardware logic simulator. In: Proceedings of the IEEE 26th Conference on design automation, pp 645–648
8. Arden W, Brillouët M, Cogez P, Graef M, Huizing B, Mahnkopf R (2010) More-than-Moore white paper. International Technology Roadmap for Semiconductors
9. Bertacco V, Chatterjee D, Bombieri N, Fummi F, Vinco S, Kaushik A, Patel HD (2013) On the use of GP-GPUs for accelerating compute-intensive EDA applications. In: Design, automation test in europe conference exhibition (DATE), 2013, pp 1357–1366
10. Brown AD, Nichols KG, Zwolinski M (1995) Issues in the design of a logic simulator: an improved caching technique for event-queue management. IEEE Proc Circ Devices Syst 142(5):293–298
11. Burns C (1996) An architecture for a Verilog hardware accelerator. In: Proceedings of the IEEE international Verilog HDL conference, pp 2–11
12. Cavanagh J (2007) Verilog HDL digital design and modeling. CRC Press, Taylow & Francis Group. ISBN 1–4200–5154–7
13. Chan T (2004) A Multithreaded HDL simulator for deep Submicron SoC designs. In: Proceedings of the IEEE Asia Pacific conference on circuits and systems (APCCAS), vol 1, pp 77–80
14. Chan T (2010) Race logic synthesis for a multithreaded HDL/ESL simulator for SoC designs. In: Proceedings of the IEEE Asia Pacific conference on circuits and systems (APCCAS), pp 1179–1182
15. Chan T (2012) A robust multithreaded HDL/ESL simulator for deep submicron integrated circuit designs. In: Proceedings of the IEEE Asia Pacific conference on circuits and systems (APCCAS), pp 416–419
16. Charlton C, Jackson D, Leng PH, Russell PC (1990) Modelling circuit delays in a demand driven simulator. Comput Electr Eng 20(4):309–318
17. Cheng WT (2000) Current status and future trend on CAD tools for VLSI testing. In: Proceedings of the 9th Asian test symposium (ATS), pp 10–11
18. Chhedaiya N, Moyal V (2012) Implementation of back propagation algorithm in Verilog. Int J Comput Technol Appl 3(1)
19. Cummings CE (1999) Correct methods for adding delays to Verilog behavioral models. 8th International HDL Conference (HDLCon)
20. Cummings CE (2000) Nonblocking assignments in Verilog Synthesis, coding styles that kill! Synopsys User Group, San Jose (SNUG)
21. Glazier M, Ambler A (1984) ULTIMATE: a hardware logic simulation engine. In: Proceedings of the 21st conference on design automation, pp 336–342
22. Gopalakrishnan GC, Fujimoto RM, Akella V, Mani NS (1989) HOP: a process model for synchronous hardware; semantics and experiments in process composition. Integr, VLSI J 8(3):209–247
23. Gordon M (1995) The semantic challenge of Verilog HDL 1995. In: Proceedings of the 10th annual IEEE symposium on logic in computer science (LICS), pp 136–145
24. Gregg D, Ertl M, Krall A (2001) Implementing an efficient java interpreter. In: Hertzberger B, Hoekstra A, Williams R (eds) High-performance computing and networking, lecture notes in Computer Science, vol 2110. Springer, Berlin, pp 613–620
25. Gu P, Vishkin U (2006) Case study of gate-level logic simulation on an extremely fine-grained chip multiprocessor. J Embed Comp 2
26. Gunes M, Thornton MA, Kocan F, Szygenda SA (2005) A survey and comparison of digital logic simulators. In: Proceedings of the IEEE 48th midwest symposium on circuits and systems, vol 1, pp 744–749
27. Hands JP (1990) What is VHDL? Comput Aided Des 22(4):246–249
28. IEEE: 1164–1993 – IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Stdlogic1164), ieee std 1164–1993 edn. (1993)
29. IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group: 1800–2012 – IEEE Standard for System Verilog–Unified Hardware Design, Specification, and Verification Language, ieee std 1800–2012 edn. (2012)
30. IEEE Standards Board: 1364–1995 – IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language, ieee std 1364–1995 edn. (1995)
31. IEEE Standards Board: 1364–2001 – IEEE Standard Verilog Hardware Description Language, ieee std 1364–2001 edn. (2001)
32. IEEE Standards Board: 1364–2005 – IEEE Standard for Verilog Hardware Description Language, ieee std 1364–2005 edn. (2005)
33. Keller BL, Carlson DP, Maloney WB (1991) The compiled logic simulator. IEEE Des Test Comput 8(1):21–34
34. Kim D, Ciesielski M, Yang S (2013) MULTES: multilevel temporal-parallel event-driven simulation. IEEE Trans Comput Aided Des Integr Circ Syst 32(6):845–857
35. Kim YI, Kyung CM (2004) Automatic translation of behavioral Testbench for fully accelerated simulation. In: Proceedings of the IEEE/ACM international conference on computer aided design, pp 218–221
36. Lewis DM (1991) A Hierarchical compiled code event-driven logic simulator. IEEE Trans Comput Aided Des Integr Circ Syst 10(6):726–737
37. Lewis DM (1992) A compiled-code hardware accelerator for circuit simulation. IEEE Trans Comput Aided Des Integr Circ Syst 11(5):555–565

38. Lewis DM, Ierssel MH, Wong DH (1993) A field programmable accelerator for compiled-code applications. In: Proceedings of the IEEE workshop on FPGAs for custom computing machines, pp 60–67
39. Li T, Guo Y, Li SK (2004) Design and implementation of a parallel Verilog simulator: PVSim. In: Proceedings of the 17th international conference on VLSI design (VLSID), pp 329–334
40. Maili A, Steger C, WeiB R, Quigley R, Dalton D (2005) Reducing the communication bottleneck via on-chip cosimulation of gate-level HDL and c-models on a hardware accelerator. In: Proceedings of the IEEE computer society annual symposium on VLSI, pp 290–291
41. Mavroidis I, Papaefstathiou I (2009) Accelerating emulation and providing full chip observability and controllability. IEEE Des Test Comput 26(6):84–94
42. Meraji S, Tropper C (2012) Optimizing techniques for parallel digital logic simulation. IEEE Trans Parallel Distrib Syst 23(6):1135–1146
43. Meraji S, Zhang W, Tropper C (2009) On the scalability of parallel Verilog simulation. In: Proceedings of the international conference on parallel processing, pp 365–370
44. Ming L, Lingyu S (2012) Research on verification and implementation of RTL-based VHDL simulator. Energy Procedia 16 P(art A):522–527
45. Muller G, Moura B, Bellard F, Consel C (1997) Harrisa: a flexible and efficient java environment mixing bytecode and compiled code. In: Proceedings of the 3rd USENIX conference on object-oriented technologies and systems
46. Najjar WA (2007) Compiling code accelerators for FPGAs. In: Proceedings of the 5th IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis (CODES+ISSS), p 2
47. Nakahara H, Sasao T, Matsuura M (2006) A fast logic simulator using a look up table cascade emulator. Asia and South Pacific Conference on Design Automation
48. Nestor JA (2005) L3: an FPGA-based multilayer maze routing accelerator. Microprocess Microsyst 29(2–3):87–97
49. Nikhil R (2013) Types, functional programming and atomic transactions in hardware design. In: Tannen V, Wong L, Libkin L, Fan W, Tan WC, Fourman M (eds) In search of elegance in the theory and practice of computation, lecture notes in computer science, vol 8000. Springer, Berlin, pp 418–431
50. Open Verilog International: Standard Delay Format Specification 3.0 (1995)
51. Palnitkar S (2003) Verilog HDL a guide to digital design and synthesis. Prentice Hall. ISBN 0–13–044911–3
52. Pavlović V, Stanisavljević Ž, Nikolić B, Ðor̄ðević J. (2011) Digital logic simulator. In: Proceedings of the IEEE 2nd eastern European regional conference on the engineering of computer based systems (ECBSEERC), pp 155–156
53. Qian H, Deng Y (2011) Accelerating RTL simulation with GPUs. In: Proceedings of the IEEE/ACM international conference on computer aided design, pp 687–693
54. Riepe MA, Marques Silva JP, Sakallah KA, Brown RB (1993) Ravel-XL: a hardware accelerator for assigned-delay compiled-code logic gate simulation. In: Proceedings of the IEEE international conference on computer design: VLSI in computers and processors (ICCD), pp 361–364
55. Rose J (1993) Panel: logic emulation: a niche or a future standard for design verification? In: IEEE 30th conference on design automation, p 164
56. Sitkowski M (1990) The macro-modeling of logic functions for the spice simulator. IEEE Circ Devices Mag 6(5):11–13
57. Sköld S, Ayani R (1995) Fast simulation of HDL models. IEEE Potentials 14(5):14–17
58. Stigali PD, Shiv K (1987) Development of a user friendly gate-level logic simulator. Comput Electr Eng 13(3–4):147–167
59. Sunkari S, Chakraborty S, Vedula V, Maneparambil K (2007) A scalable symbolic simulator for Verilog RTL. In: Proceedings of the 8th international workshop on microprocessor test and verification (MTV), pp. 51–59
60. Sutherland S (2000) The IEEE Verilog 1364–2001 standard what's new, and why you need it. 9th Internatioinal HDL Conference (HDLCon)
61. Vaidya P, Lee JJ (2007) Simulation of hybrid computer architectures: simulators, methodologies and recommendations. In: Proceedings of the IEEE IFIP international conference on very large scale integration (VLSISoC), pp 157–162
62. Wageeh MN, Wahba AM, Salem AM, Sheirah MA (2004) FPGA based accelerator for functional simulation. In: Proceedings of the international symposium on circuits and systems (ISCAS), vol 5, pp V-317–V-320
63. Wang L, Chen H, Deng Y (2013) Robust conservative parallel HDL simulation on multi-core CPUs. In: 2013 international conference on, high performance computing and simulation (HPCS), pp 413420
64. Wong K, Franklin MA (1989) Performance analysis of a parallel logic simulation machine. J Parallel Distrib Comput 7(3):416–440
65. Wu J (2010) Functional verification methodology of complex electronics system based modeling and simulation. J Comput 5(9):1343–1347
66. Xu Q, Tropper C (2005) XTW: a parallel and distributed logic simulator. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC), vol 2, pp 1064–1069
67. Yu Y, Hoare RR (2003) A hardware acceleration simulation for user-defined-primitives. In: Proceedings of 5th international conference on ASIC, vol 1, pp 199–202
68. Zhang H, Wang ZQ, Liu W, Tan Z (2012) The design of arithmetic logic unit based on ALM. Procedia Eng 29:1969–1973

Tze Sin Tan received his Bachelor degree in Electrical Engineering from Universiti Teknologi Malaysia (2000) and his Masters degree in Solid State Physics from Universiti Sains Malaysia (2003). He is presently a Senior Member of Technical Staff at Test Development Department of Altera. He is engaged in defining DFT and test generation for FPGAs. He is also a student with Universiti Sains Malaysia under the Industrial Ph.D. program.

Bakhtiar Affendi Rosdi received his B.Eng, M.Eng, and D.Eng degrees in electrical and electronic engineering from Tokyo Institute of Technology, Tokyo, Japan in 1999, 2004, and 2007, respectively. He is currently a senior lecturer of School of Electrical and Electronic Engineering in Universiti Sains Malaysia. His research interest is the hardware architecture of pattern recognition algorithm and Biometrics.