A Verilog Compiler Proposal for *VerCPU* Simulator

Tze Sin Tan
Altera Corporation
Penang, 11900 Malaysia
tts12_eee107@student.usm.my, tstan@altera.com

Bakhtiar Affendi Rosdi
Universiti Sains Malaysia
Penang, 11800 Malaysia
eebakhtiar@usm.my

January 29, 2015

Abstract— Verilog is a widely used Hardware Description Language (HDL) for VLSI design and modeling. As a language developed with hardware execution concurrency in mind, Verilog can be mapped onto a dedicated processor for higher simulation throughput. The processor requires a compiler to transform Verilog netlist into compiled-code instructions. In this paper, we propose a data structure that adequately represents a Verilog model. Then, the Verilog compiler is developed to map Verilog netlist into this data structure. We also demonstrated that it is possible to construct a hardware simulator (*VerCPU*) utilizing this data structure.

I. Introduction

Verilog is a Hardware Description Language (HDL) used to model digital circuits. It accepts coding abstraction levels starting from behavioral, Register-Transfer-Level (RTL), gate level netlist (GLN), and transistor level in accordance to modeling requirements throughout design flow. At behavioral level, all syntactically correct codes are acceptable. RTL, on the other hand, accepts a subset of behavioral codes that can be transformed unambiguously to connections of common circuit components. This connection of fundamental components is represented by GLN. Transistor level is used where low level modeling is needed using MOSFETs.

All simulation platforms capable of consuming Verilog model and evaluating its behavior are collectively known as simulators. The IEEE standards defined simulation scheduling semantics [1, 2]. This allows consistent results being produced by all Verilog compliant simulators. Software based simulator running on a general purpose computer is the most widely adopted solution. However, software simulator suffers a few shortcomings where higher simulation throughput is sought after [3]. The main reason being that Verilog statements cannot be mapped directly onto a general purpose CPU instruction. For example, a general purpose CPU processes logic *0* and *1*, but cannot process X/Z and signal strength directly.

In this paper, we present a compiler to transform Verilog constructs into basic operations that are suitable for execution on a custom Verilog processor, dubbed *VerCPU*. We begin the discussion by explaining the application of *VerCPU* system in a multi-user development environment. This is followed by treatment of commonly used Verilog constructs into manageable basic operations. Then, an internal data structure representing a Verilog netlist is proposed. In the subsequent section, a simulator modeling *VerCPU* is developed to demonstrate the viability of running simulation traversing the proposed data structure. Lastly, we summarize the achievements and possible next steps that can be carried out to improve digital logic simulation.

II. *VerCPU* Architecture

Typically, software based Verilog simulator runs on a general purpose computing system. The compiler converts Verilog syntax into instructions supported by the computer. Often, there is no good match of native instruction corresponding to Verilog primitive. A single Verilog operation results in a sequence of CPU instructions. More on this will be elaborated in Section III B.

Hardware assisted simulator is another option, mainly synthesis-based FPGA platform and emulator [6]. FPGA platform requires a synthesis process to map onto the hardware. Not only that this is a complex process, synthesized circuit does not offer the same level of simulation fidelity. XZ and delay modeling are examples. Generally, emulator has similar hardware mapping flow and limitations compared to synthesis-based FPGA platform. However, its tool chain includes software development platform with in-circuit emulation capability. Architecture and usage model of modern general purpose emulator are vendor specific.

In order to improve simulation efficiency while retaining the fidelity of software simulator and usage flow, *VerCPU* is a custom processor designed to carry out HDL simulations. The processor has built-in instruction set to evaluate Verilog primitives directly. At such, the processor executes simulation in lesser clock cycles as compared to simulating the same Verilog statement on a general purpose CPU.

Before we proceed to describe the proposed Verilog

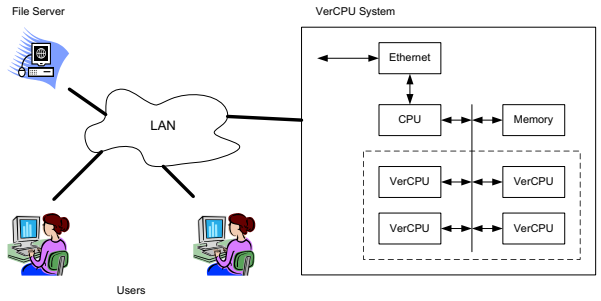

Fig. 1. Usage of VerCPU over Local Area Network (LAN)

data structure, it is imperative to have an understanding of where and how the data structure is used. An overview of Verilog simulator based on *VerCPU* is given here. This system is constructed using a pool of parallel running *VerCPU*s evaluating event-triggered Verilog statements. From user point of view, the usage is similar to a server hosted software based simulator. As depicted in Figure 1, a *VerCPU* system is connected to Local Area Network (LAN). Users initiate simulations through a file server, where compilation of Verilog models is carried out and sent to *VerCPU* system for simulation. The *VerCPU* system consists of a general purpose CPU for system management, an array of volatile memories to support runtime simulation storage, and a pool of parallel processing *VerCPU*s. The system communicates to LAN through Ethernet.

A more detail overview of the system is illustrated in Figure 2. *VerCPU*s operate on Verilog primitives from behavioral to transistor level. Instruction set is crafted to perform statement evaluation at the shortest time including X/Z signal state, signal strength, and delay. Multiple *VerCPU*s are instantiated for parallel processing. This is particularly to the advantage of Verilog which is already capable of supporting concurrent execution by design. Events scheduled to be evaluated in the same time step can be distributed to any processor for parallel evaluation without violating causality. The system takes in netlist model and runtime data from main memory (DDR). While the role of event list, runtime data, instruction, and netlist will be explained in later sections, these can be understood to be the sources of simulation system inputs for the moment. L1 and L2 cache memories are used to reduce memory access latency. Instruction and netlist cache can be sourced from a speed optimized L1 cache by skipping coherency checks because the content does not change during a simulation session.

III. Verilog Model

In this section, we discuss major components that make up Verilog language. Besides serving as a refreshment to readers, we explained considerations for simulator construction. In order to keep the discussion succinct, we focus on the main constructs of Verilog.

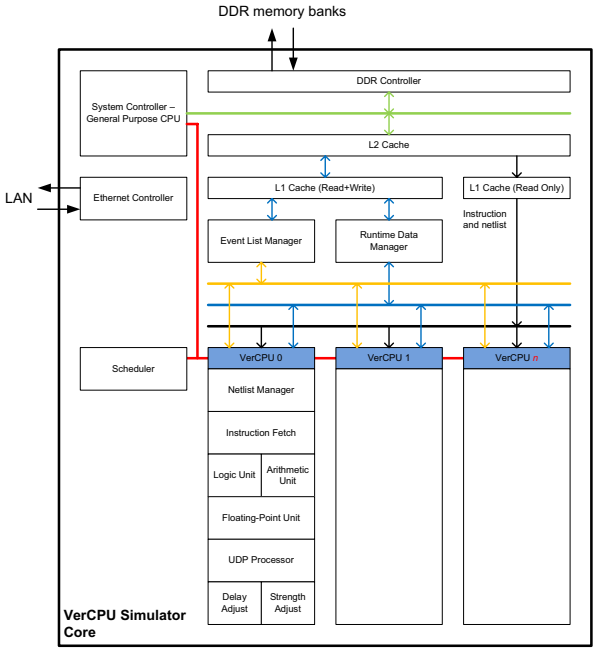

Fig. 2. Block level view of VerCPU system

A. Hierarchy

Verilog netlist is a collection of module instances. A module is a template of circuit entity that transforms *inputs* into *outputs*. Every module may consist of instantiations of smaller modules with connections in between these modules. In a module, statements that describe the functionality of a module are coded. For every physical node in an instantiated module, runtime storage must be allocated to store its electrical state in a simulator. Figure 3 summarizes using an example showing runtime storage allocation according to the modeled circuit. Assuming every node requires 1 unit of storage, *inst0* requires 5 storage units to store *clk* and *countout[3:0]*. Another instance, *inst1* requires 9 storage unit because the *WIDTH* is overridden to 8. Summing from all submodules, *TopModule* requires 14 units of runtime data storage.

B. Statements

In Verilog, all statements are viewed as being executed concurrently. A well written model always give consistent results on all Verilog compliant simulators. State convergence and race hazard do not result in uncertainty by adhering to design and modeling rules. There are two classes of statements, i.e. continuous assignments and procedural blocks.

Continuous assignment is used to describe simple logic and arithmetic behavior. Though meant to be simple for

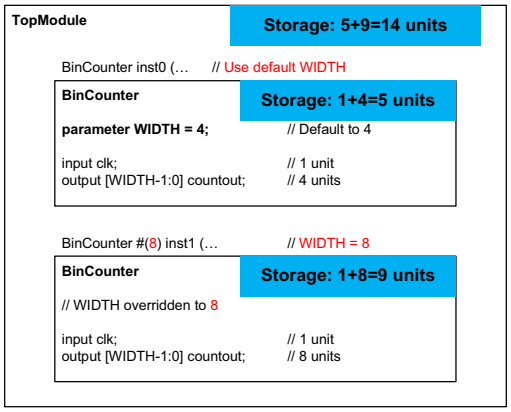

Fig. 3. Calculation of runtime storage for hierarchical instantiated modules

hand coded RTL, CAD tools can generate highly complex statements. Usually, a general purpose CPU has instruction set accepting two arguments [4, 5]. At such, equation evaluation has to be carried out in a sequence of two argument operations. For example, equation 1 is decomposed into two step operations as listed in equations 2.

$$assign\ A = B \mid C \,\&\, D; \tag{1}$$

$$Y = C\&D; A = B \mid Y \tag{2}$$

Note that a compiler must observe precedence of operations defined in Verilog. In the given example, logical OR has lower precedence than logical AND. Though logical operations are supported by most CPUs, Verilog circuit node is in one of four possible states (*0, 1, X, Z*) and seven strengths (*supply* to *hi-Z*). Therefore, equations 2 need to be further elaborated in order to evaluate X/Z state and signal strength.

Procedural statements, on the other hand, are a collection of logical, arithmetic, and conditional branching statements. Among commonly used procedural blocks include *initial*, *always*, and *table*. The blocks consist of sequentially evaluated statements. *Table* is the keyword used to code User Defined Primitive. It is a form of representing the transformation of *inputs* into an *output* using look-up table, either combinational or sequential.

IV. Data Structure and Compiler Proposal

In this paper, Verilog compiler is restricted to the context of simulation on *VerCPU* , a custom designed processor capable of evaluating Verilog statements using its native data types. Compilation involves the steps of transforming a Verilog netlist into machine instructions and the system's internal data structure. Here, we propose to represent the netlist using four data blocks, i.e. Netlist Map, Instruction Segment, Data Map, and Event List.

Netlist Map describes hierarchical relationship of flattened netlist nodes. All nodes are assigned a unique entry in the Netlist Map. Each entry points to the next entry which the node is connected to. As Verilog simulator is event triggered, the entry also points to a section of instructions that it triggers when the node changes state. Instruction Segment is a list of compiled statements in the modules. A module can be instantiated many times in the netlist. Though every instance has its own runtime storage, only one set of instructions is needed for the module instances. Therefore, the triggering event needs to pass the base address of the instance (index to the first data node) that triggers the instructions, similar to base pointer associated to a function call in software programming. A Data Map consists of runtime storage for every node in the netlist. This includes intermediate nodes created by the compiler after decomposing Verilog statements. The fourth data block, Event List, contains pointers to all active instructions pending execution. Upon simulation startup, the list is initialized with pointers to the beginning of all *initial* and *always* blocks. Event List is dynamically updated during runtime. Newly triggered events are inserted whereas evaluated events are removed. An example of compiled data blocks is shown in Figure 4.

The example is a 4-bit binary counter and its test bench. Verilog source code is on the left of the diagram. In module *counter*, output port *count* is reset to 0 by asynchronous *rst*. Upon de-assertion of *rst*, *count* is incremented at every clock edge. Module *counter_tb* is the test bench that instantiates a *counter*. The test bench also generates a free running clock and stimuli to the *dut*, i.e. the instance name of *counter*.

Every node requiring runtime storage results in a node entry in the Data Map. Referring to the bottom right block labelled Data Map, node 0–11 are directly derived from signals in the RTL. As explained in Section III B, complex statements are decomposed into simpler equations. Often, additional storage is needed to keep track of intermediate results. These nodes are indicated by "$" prefixed nodes in the Data Map.

Instruction Segment is shown on the upper right block in the figure. Segment of codes is color shaded to show the same portion of corresponding RTL. The instructions are shown in pseudo-code for ease of understanding. 31 instructions are generated in this example. Instructions 0–2 are generated from a free running clock source in the test bench. The first instruction is a *delay* to defer execution of the next instruction. During runtime, an event pointing to instruction 1 is created and inserted into the Event List for 10 time units after current simulation time. "[1] = ~[1]" means node 1 in Data Map is inverted and updated into the same location. Instructions run sequentially after triggered. A "(Last)" prefixed instruction indicates that the execution has reached the end of an instruction section. Upon encountering "(Last)", simulation proceeds to extract the next event instead of advancing to the next

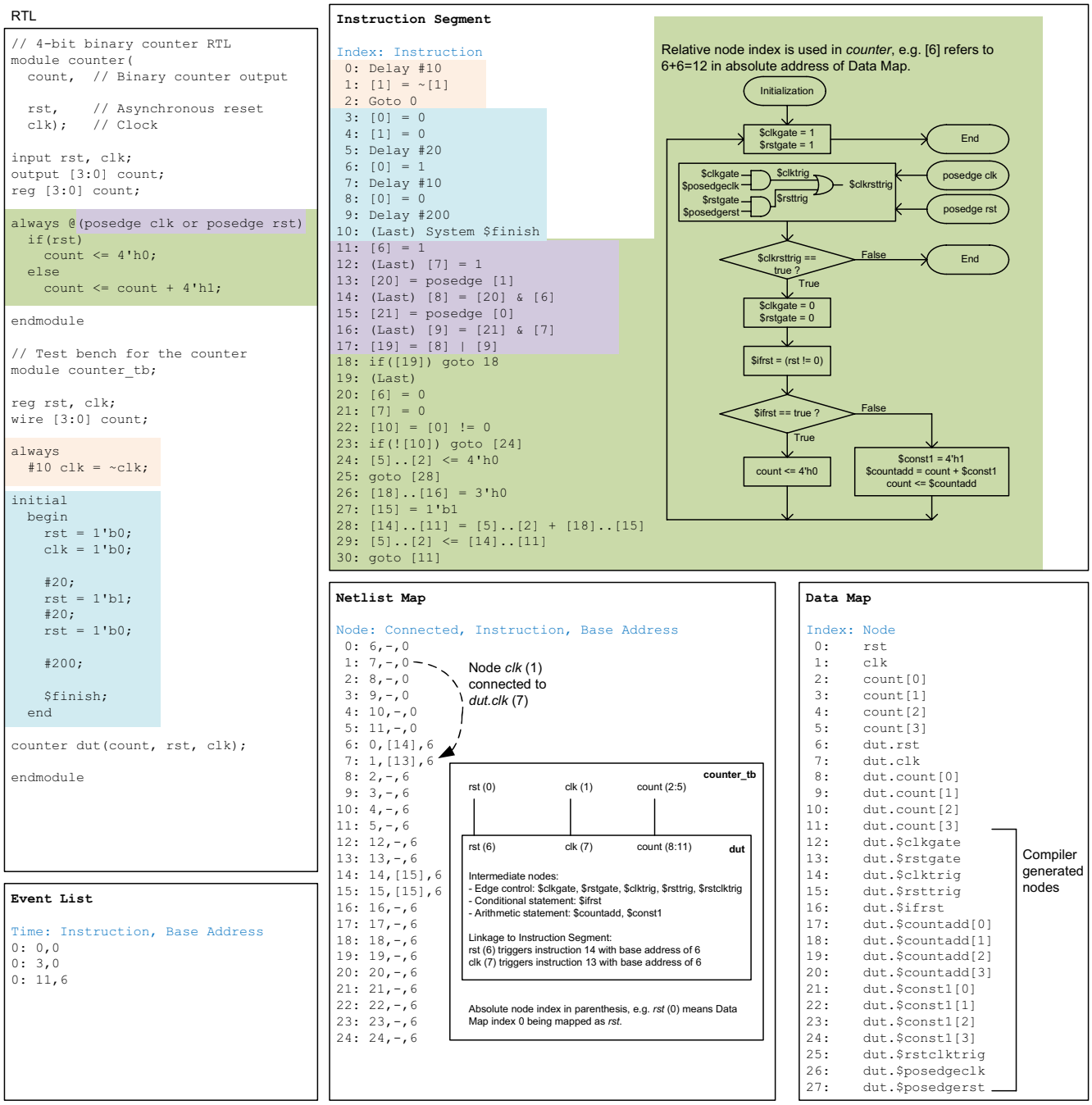

Fig. 4. Compiler output using 4-bit binary counter and its test bench

instruction address. A detailed breakdown of procedural block (*always*) with sensitivity list is expanded in the flowchart on the right. Gating nodes such as *$clkgate* and *$rstgate* are generated to prevent re-entry into the procedural block when multiple events are triggered pointing to the same procedural block. An overall gating signal representing the combined sensitizing event, *$clkrsttrig*, is used to gate subsequent statement evaluation until a sensitivity list event is detected. In more complex modeling where multiple event control statements (@*sensitizing event*) are used in the same procedural block, the gating controls are expanded to cover all instances of event control statements. This ensures correct sequential evaluation of the procedural block where an event control has been encountered.

At simulation startup, the Event List is initialized with

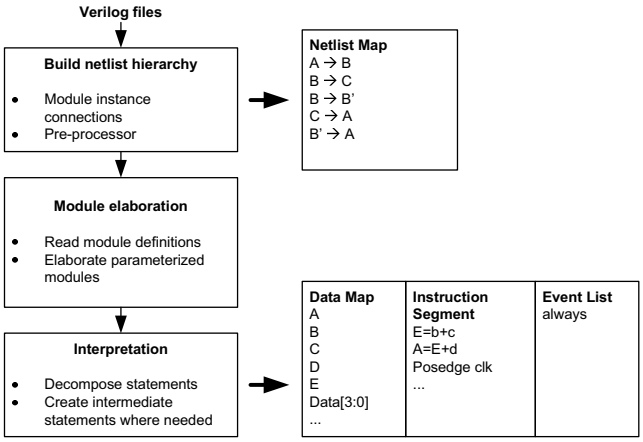

Fig. 5. Compilation flow

pointers to the instruction sections that are scheduled to be executed at time 0. In this example, there are three startup events resulted from procedural blocks (two *always* and one *initial*). This list will be updated dynamically during simulation depending on simulation events triggered.

Lastly, a Netlist Map lists the connection of circuit nodes. A circular list is maintained for each net, along with the instance's base address. Use of base address allows the sharing of instructions for all instances through relative address computation. The example emphasizes connection of node 1 (*clk* in test bench) and 7 (*clk* in the *dut*). Node 1 does not trigger new event creation, but node 7 triggers a new event creation pointing to instruction 13 with base address of 6.

A summary of compilation flow is shown in Figure 5. Compilation starts by reading in all Verilog models. Pre-processors, such as *`ifdef–`endif* and user defined macros, are interpreted at the same time. A skeletal netlist graph, which consists of connections and parental relationship, is created at the end of this phase. Next in the elaboration step, parameterized modules are generated. Only one module definition has to be stored for all instances of a module having the same parameter set. In the interpretation phase, statements are decomposed into sequence of operations. At the end of this phase, the netlist is fully interpreted and generated in the form of internal data structure presented earlier.

The data structure proposal is created with *VerCPU* in considerations. It is a custom processor built for Verilog simulation. The Netlist Map and Instruction Segment can be marked read-only for simpler cache design, whereas the Data Map and Event List can be optimized for multiprocessor accesses. *VerCPU* supports Verilog logic levels and strengths as its data type, i.e. one cycle instruction for logic operation involving *01XZ* in the inputs. In contrast to general purpose CPU that has to evaluate a Verilog statement in multiple steps as described in Section III B, the *VerCPU* is able to carry out the same evaluation using an instruction completing execution in lesser clock cycle.

V. Results

In this section, we demonstrate how a simulator can be constructed by consuming data blocks discussed in Section IV.

Simulation follows the flow illustrated in Figure 6. The global timer is reset to 0 upon simulation starts. Events that are scheduled to be executed at time step matching the global timer are evaluated. This is carried out by executing instructions pointed by the event. Upon evaluating a statement, the resultant may cause other nodes to toggle. This in turns triggers a new event. The new event is inserted into the Event List. The order of extracting events scheduled at the same time step is not important because they can be executed in concurrent. Models coded in accordance to HDL guidelines are immune to circuit hazard resulted from event evaluation sequencing uncertainties. However, it must be noted that blocking statement updates have higher precedence over non-blocking updates. This is automatically observed by assigning an additional time step precision. All non-blocking updates are scheduled at the end of current simulation time step.

After having exhausted all events in current time step, the global timer is advanced to the next time step as indicated by the scheduled event at the head of Event List. The same process of "extract event ⇒ evaluate ⇒ insert event" is repeated. A simulation session is complete when the Event List becomes empty. Most of the time, this is impractical because of the existence of free-running clocks which continuously schedule new events. Flow control task such as *$finish* is used to forcibly terminate simulation after all meaningful simulation data has been gathered.

A *VerCPU* model was written in C++ to simulate the algorithm shown in Figure 6. Simulation result was verified to be correct, as illustrated in Figure 7.

VI. Summary

In this paper, we presented a proposal to compile Verilog netlist into data blocks consisting of Event List, Data Map, Instruction Segment, and Netlist Map. These entities sufficiently represent the source of inputs for simulation on a custom processor as demonstrated through *VerCPU* model.

VII. Future work

The Verilog compilation process described in the paper paves the way for a hardware assisted simulation platform,

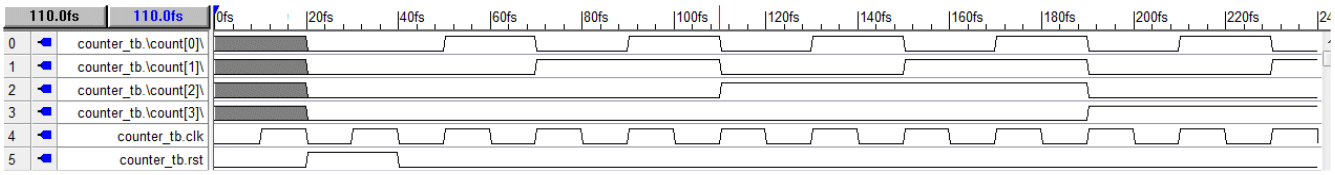

Fig. 7. Simulation output of the 4-bit binary counter

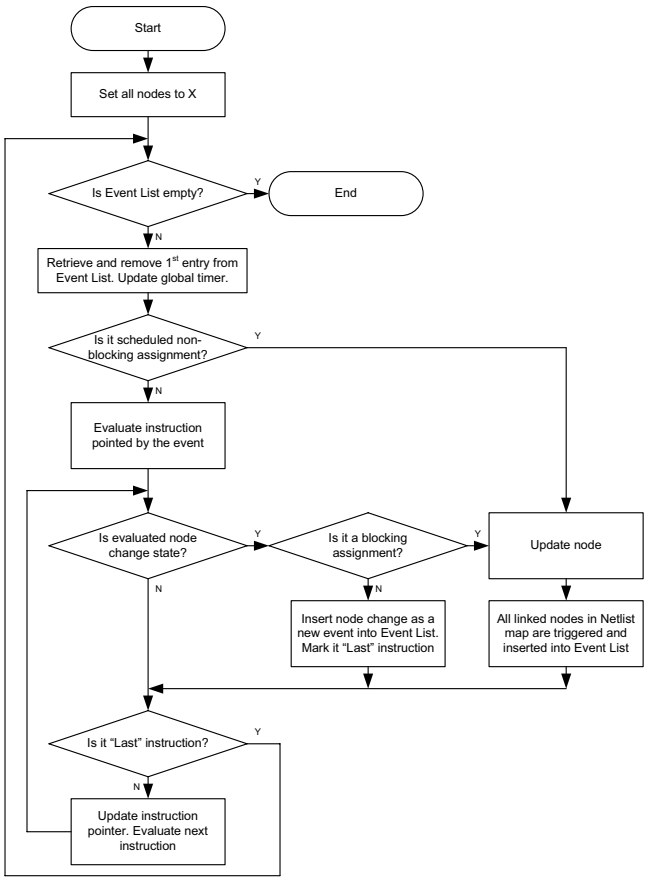

Fig. 6. Simulation flow chart

VerCPU . The proposed compilation outputs are structured with simulation platform scalability and parallelism in considerations. This solution fills the gap in between conventional software based Verilog simulator and proprietary vendor specific hardware assisted platforms. We look forward in presenting such a system in near future.

Acknowledgment

This research was a collaboration in between Universiti Sains Malaysia and Altera Corporation.

References

[1] IEEE Standards Board, "1364–1995 – IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language," *IEEE Std 1364–1995*, 1995.

[2] IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, "1800–2012 – IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800–2012*, 2012.

[3] V. Bertacco, D. Chatterjee, N. Bombieri, F. Fummi, S. Vinco, A.M. Kaushik, and H.D. Patel, "On the Use of GP-GPUs for Accelerating Compute-intensive EDA Applications," *Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1357-1366, 2013.

[4] R.K. Agarwal, "80x86 Architecture and Programming," *Prentice-Hall*, 1991.

[5] G. Kane and J. Heinrich, "MIPS RISC Architecture," *Prentice-Hall*, 1992.

[6] Samir Palnitkar, "Verilog HDL A Guide to Digital Design and Synthesis," *Prentice-Hall*, 2003.