International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 04 Issue14
November 2017

A Review on Source Code Error Detection in High-Level Synthesis Functional Verification

P.Usharani &Ms. Kapuluru Leelavathi
1 PG Student, Dept. Of VLSI and Embedded systems, SKR College Of Engineering & Technology, AP
2 Asst. Professor, Dept. Of VLSI and Embedded systems, SKR College Of Engineering &Technology,AP.

Abstract_*A dynamic functional verification method thatcompares untimed simulations versus timed simulations for synthesizable [high-level synthesis (HLS)] behavioral descriptions (ANSI-C) is presented in this paper. This paper proposes a method that automatically inserts a set of probes into the untimed behavioral description. These probes record the status of internal signals of the behavioral description during an initial untimed simulation. These simulation results are subsequently used as golden outputs for the verification of the internal signals during a timed simulation once the behavioral description has been synthesized using HLS. Our proposed method reports any simulation mismatches and accurately pinpoints any discrepancies between the functional Software (SW) simulation and the timed simulation at the original behavioral description (source code). Our method does not only determine where to place the probes, but is also able to insert different type of probes based on the specified HLS synthesis options in order not to interfere with the HLS process, minimizing the total number of probes and the size of the data to be stored in the trace file in order to minimize the running time. Results show that our proposed method is very effective and extremely simple to use as it is fully automated.*

INDEX TERMS—Domain-specific design, field-programmablegate array (FPGA), high-level synthesis (HLS), quality of results (QoR).

I. INTRODUCTION

High level synthesis (HLS) tools, which automate transla-tion of C/C++ algorithm implementations into register transfer level (RTL) descriptions, have seen significant improvements in recent years. HLS tools are large software systems, and thus verification and debugging of HLS tools is a significant portion of the design and development effort.Traditionally, large scale software development uses a va-riety of tools and techniques to support verification and debugging efforts, including debug tools (e.g. GDB), memory analysis tools (e.g. Valgrind), assertions and printing-based debugging, modularization of source code for unit-testing, and formal verification. However, although these techniques continue to play a role in debugging of HLS tools, they are not sufficient; these tools can help verify that an HLS tool executes without syntax errors and produces syntactically correct RTL, but final verification also requires that the produced RTL is functionally equivalent to the input C/C++ source.

Functional verification of RTL is performed through sim-ulation and comparison of output values. When an output mismatch is identified, the user must trace backwards through the simulation to discover the earliest incorrect internal value; this earliest symptom can then

be used to diagnose the cause of the problem in the HLS tool. This process may require de-tailed backtracing through hundreds of signals over the course of hundreds or thousands of cycles of simulated execution. Furthermore, HLS-produced RTL is typically not intended to be human-readable. This challenge of effectively verifying and debugging incorrect RTL can become a bottleneck in HLS tool development, hindering further improvement of the HLS tools.

The HLS process performs many transformations to paral-lelize and optimize execution; thus, we cannot validate appli-cation correctness by comparing the exact order of operations. However, we can fundamentally characterize correct execution with a few properties: input data received, output data pro-duced, conditional control transitions, correct propagation of data through data selection (PHI-node) operations, and forward progress in execution.

In this paper, we present a framework that supports HLS tool debug: we use just-in-time compilation and trace-generation to generate the set of expected values for all operations that characterize an application and automatically insert RTL ver-ification code for each operation and value pair, together with information about the correspondence between the RTL and operation in LLVM-IR. Using this framework, we demonstrate that we can detect bugs in the HLS core. Furthermore, we demonstrate that our RTL verification code detects the earliest instance of execution mismatch with low-latency; often zero cycles, and always 3 or fewer cycles of simulated execution.

This paper contributes to debugging and verification of HLS tools with:

A JIT based implementation that automatically gathers expected values for all characteristic operations.

A trace-based approach to automatically insert RTL ver-ification code for all operations that characterize correct application execution.

A demonstration that this technique detects mismatched execution with low-latency.

II.PROPOSED WORK

Our proposed method applies to the first category for HLS (presilicon), but uses concepts used in the second category (postsilicon). The notion of probes has been taken from typical VLSI postsilicon verification flows. For example, commercial Field Programmable Gate Array tool vendors provide on-chip support to allow the observability of internal signals (e.g., Chipscope in Xilinx [12] and SignalTap in Altera [13]). These tools insert probes to signals in the design to be tested and capture them using a sampling clock, while storing them in a buffer. The buffer content is transmitted to a PC and displayed graphically, once the buffer is full or certain number of samples taken. The designer can then manually verify the correctness of the design. ARM does also provide a similar technology to debug ARM-based systems-on-a-chip with the Advanced High-performance Bus (AHB) trace macrocell, which gives visibility on Advanced Microcontroller Bus Architecture AHB busses, offering visibility of accesses to memory areas [14]. To be able to root-cause design bugs, postsilicon validation requires to have full controllability and observability of the circuit under debug's (CUD) internal behavior. This can currently not be achieved due to the extremely larger number of signals that would

need to be traced. A more effective debug technique is to selectively monitor some of the internal signals. Designers typically select to tap a number of signals in the CUD, but only a subset of the tapped signals are traced concurrently during debug phase due to trace bandwidth limitation. This is achieved by inserting a mux tree that links the tapped signals to trace buffers or trace ports. These systems also include trigger units, which are used to determine when to start and stop signal tracing in order to further reduce trace bandwidth requirement [18]. The effectiveness of these trace-based debug systems, hence, rely considerably on the signals being traced. In current postsilicon validation flows designers usually manually select those signals that are important for analysis to trace, based on their own design experience. This ad hoc method, however, cannot guarantee the quality of debug process [17]. More importantly, bugs often occur in unexpected scenariosand it is very difficult, if not impossible, to predict which signals will be related to them during the design phase. Ko and Nicolici [19] first introduced an automated method identifying a small set of trace signals from which a large number of states can be restored using a compute-efficient algorithm. This enlarged set of data can then be used to aid the search of functional bugs in the fabricated circuit. Liu and Xu
[21] expanded this paper conducting circuit-level propagation of risibilities from traced signals to untraced ones achieving a more accurate visibility estimation. Although our work applies to a completely different VLSI design stage, its main objective is similar to the postsilicon validation techniques. This paper targets the verification at the synthesis level. A classification of synthesis verification is given in [25]. This paper classifies the synthesis verification into presynthesis verification of algorithm(s) to be synthesized typically using software verification
methods, formal methods using theorem provers and postsynthesis verification, where the synthesized results are verified against the input behavioral descriptions. This last category is the most
widely used today, to which this paper also belongs. This last category can be further classified into simulation based and formal based. Formal methods have been applied to verify the HLS process usng translation validation. For example, Ashar et al. [27] focused on the valid binding stage of HLS, while recently [26] focused on the scheduling and concurrent systems modeling communicating sequential processes. Formal methods have gained popularity because RTL simulations for larger designs, simulations are too slow and cannot detect corner cases. Other formal verification approaches include [30], where a fully automatic equivalence
verification of a design before and after the scheduling step of HLS is presented. This paper was extended in [31] by mapping the designs into virtual controllers and virtual datapaths. A more recent work [32] uses a finite-state machine (FSM) with datapath models to represent both behaviors (untimed and timed). Our work is fundamentally different from this previous works as it is simulation based. An early work on simulation-based HLS verification is presented in [28]. The advantages ofsimulationbased methods are that simulations are always needed for overall functional verification. Moreover, we apply our method to compare pure software (untimed) simulations and use cycleaccurate model simulations as the timed model instead of the synthesizable RTL generated by HLS. Cycle-accurate models have been reported to be 10–100× faster than RTL simulations [3], making our method fast enough

International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 04 Issue14
November 2017

to work for larger designs. We can define the problem to be solved as follows. Problem Definition: Find the signals to be traced and probe insertion points in a behavioral description for HLS in order to locate the operation in the source code where the error is first introduced, minimizing the simulation running time and trace file sizes, without distorting the intended HLS result. To the best of our knowledge, this is the first work investigating functional verification methods comparing untime versus timed simulations at the behavioral level. Section II-A describes a typical HLS verification flow, indicating where our proposed method fits in the overall VLSI verification flow, followed by a detailed description of our proposedmethod.

A. High-Level Synthesis Verification Flow
HLS takes as inputs a behavioral description, e.g., C, C++, or SystemC, and generates synthesizable RTL(Verilog or VHDL) by creating a control unit in the form of an FSM and a data path unit. The datapath unit mainly consists of a number of functional units (FUs) combined with registers and multiplexers. Most commercial HLS tools provide tools to verify and debug the design at the highest possible level of abstraction in order to facilitate the verification process. For this purpose, they normally include model generators that create different types of simulation models depending on the design stage

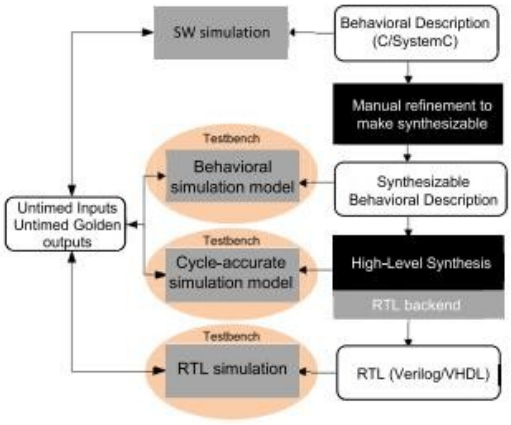


Fig. 1. HLS design flow and verification overview.

When using HLS, he first step designers need to take is to manually refine the original SW description in order to make it synthesizable. Some of the typical constructs that are not supported in HLS are dynamic memory allocation and recursion. At this stage, the designer also refines the data types in order to obtain the smallest possible and most efficient HW design. For this purpose, most HLS vendors extend the C syntax providing their own data types (e.g., CatapultC uses ac_types [5] and CyberWorkBenchvar_types
[6]). In the case of SystemC, the sc_types are used. For the verification at this state, some of the vendors include behavioral model generators. These model generators create a behavioral program that models the original behavioral description including its custom data types. After HLS, a cycle-accurate simulation can be performed for timing verification. These cycle-accurate models again generate a behavioral description in any high-level language, i.e., ANSI-C or C++/SystemC, and mimic the behavior of the RTL cycle accurately. The input to these cycle-accurate model generators are normally the result of the HLS scheduling phase (a byproduct of HLS). These models have been proven to be consistently faster than RTL by a factor of 100–1000× for the behavioral model and 10–100× faster for the cycle-accurate model [3]. A testbench generator that allows the reuse of the untimed SW inputs and outputs is typically part of these model generators.

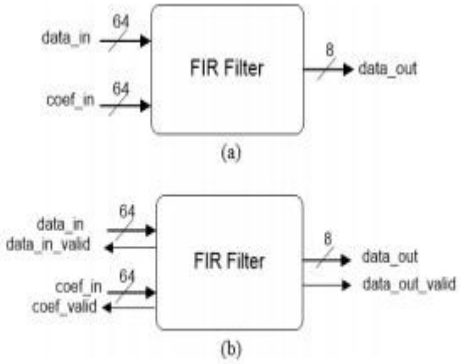
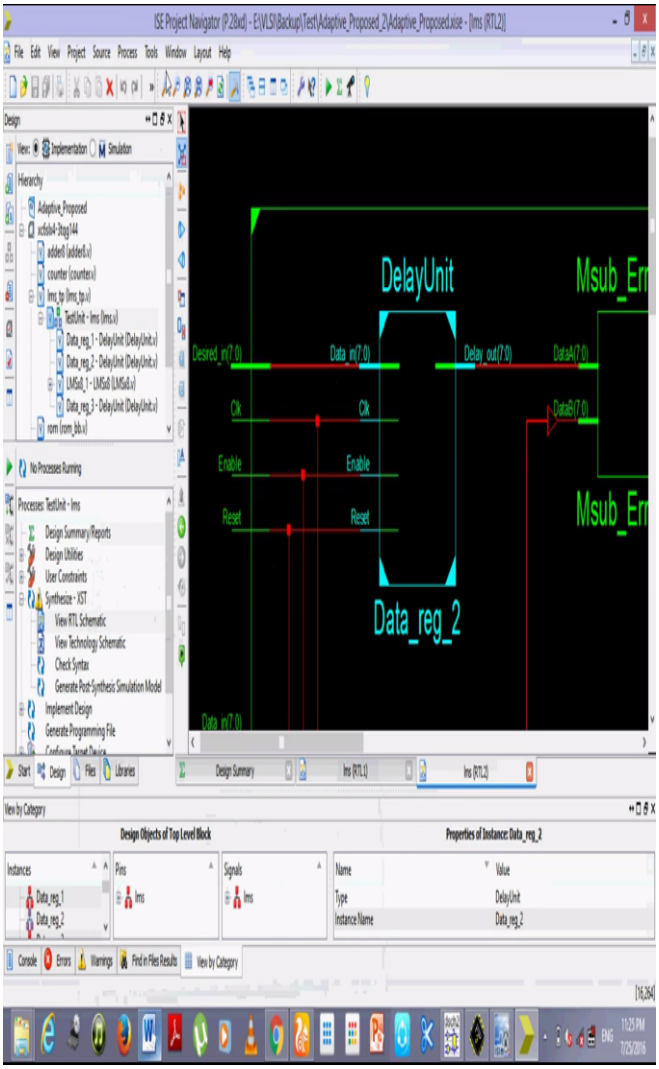
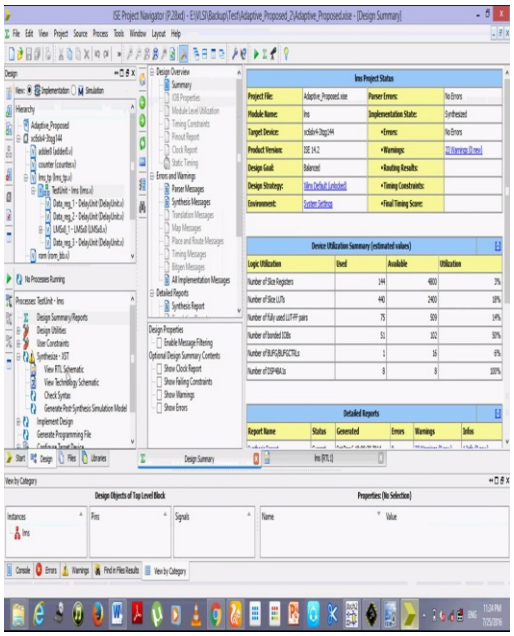
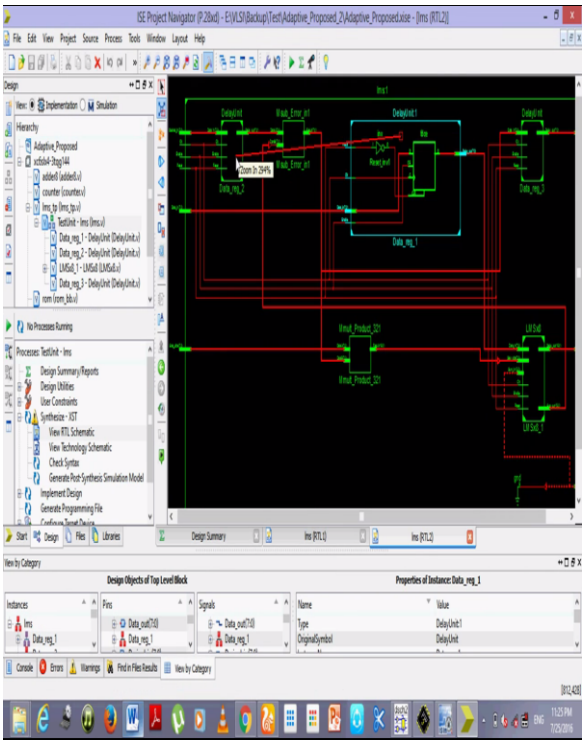


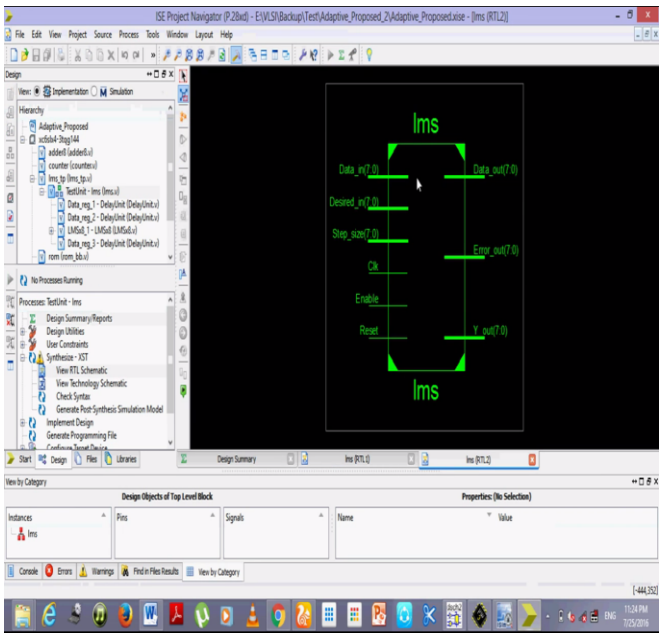
Fig. 2. (a) Regular FIR filter. (b) FIR filter with valid signals for DEC.

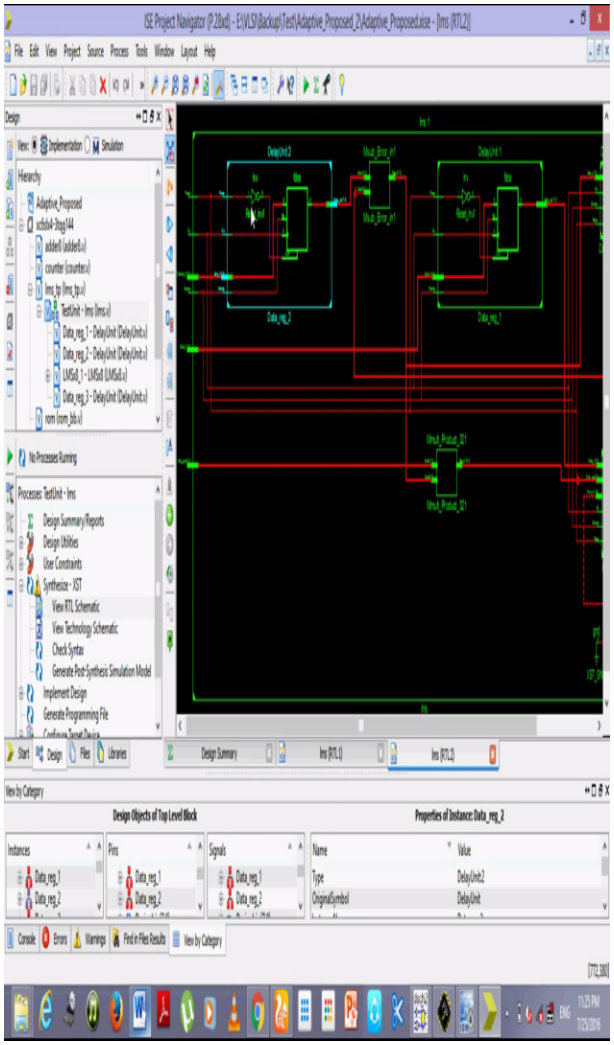
III.Simulation Results:

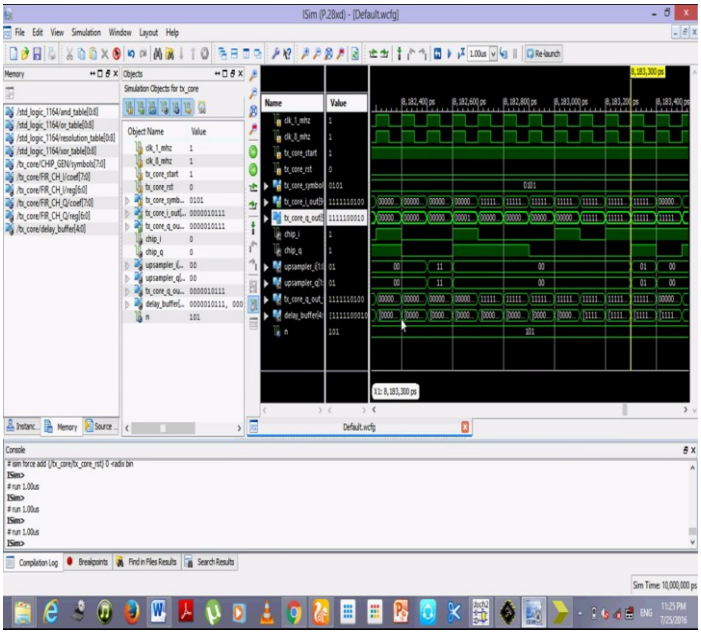


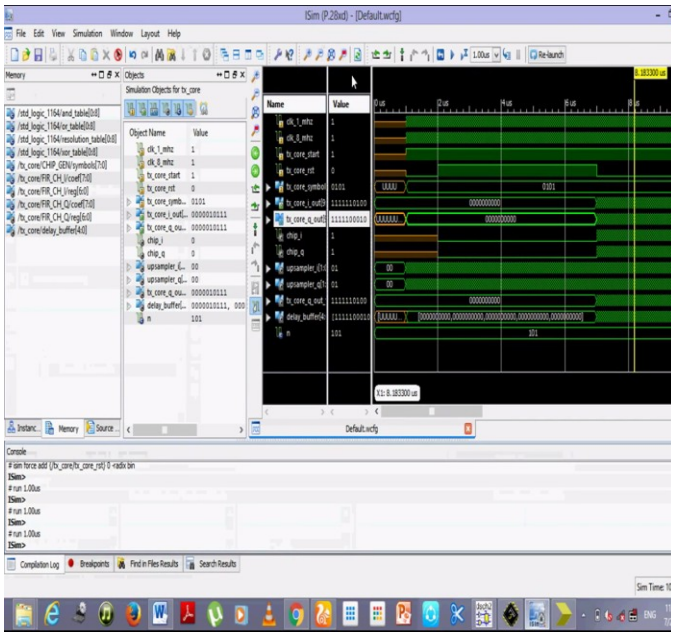












IV.Conclusion:

In this paper, we have presented a complete automated verification flow for synthesizable behavioral descriptions in order to detect where in the source code mismatches between the original untimed simulation and the timed synthesized design occur. Our proposed verification flow leverages the latest verification features of commercial HLS tools, which allow the reuse of transaction level test vectors for timed simulations. By automatically inserting a set of internal probes our method can efficiently detect mismatches between the untimed behavioral simulation and the synthesized circuit and locates where the error is introduced directly at the source code based on the distances between probes. This paper introduces the term SCED to determine the quality of our verification environment. The proposed method inserts different types of probes based on the synthesis directives for arrays and loops and makes use of synthetic operators in probes for arrays to avoid the probes interfering with the HLS results. Three different probe insertions methods are presented each with unique tradeoffs (SCED versus simulation runtime versus VCD file size). A set of experiments were conducted and

an error was found in one of the designs that would have taken much longer time to find using a manual approach, further validating our verification methodology. The probe library is currently being extended to include probes, e.g., partial loop unrolling.

References

[1] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formal verification of ssa-based optimizations for llvm," in ACM SIGPLAN Notices, vol. 48, no. 6. ACM, 2013, pp. 175–186.

[2] A. Mathur, M. Fujita, E. Clarke, and P. Urard, "Functional equivalence verification tools in high-level synthesis flows," Design Test of Comput-ers, IEEE, vol. 26, no. 4, pp. 88–95, July 2009.

[3] X. Feng and A. Hu, "Early outpoint insertion for high-level software vs. rtl formal combinational equivalence verification," in DAC, 2006, pp. 1063–1068.

[4] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson, "Source level debugger for the sea cucumber synthesizing compiler," in FCCM, April 2003, pp. 228–237.

[5] J. A. N. Calagar, S. Brown, "Source-level debugging for fpga high-level synthesis," in FPL, September 2014.

[6] J. Goeders and S. Wilton, "Effective fpga debug for high-level synthesis generated circuits," in FPL, Sept 2014, pp. 1–8.

[7] G. Jeffrey and W. Steven, "Using dynamic signal-tracing to debug compiler-optimized hls circuits on fpgas," in FCCM, 2015.

[8] K. A. Campbell, D. Lin, S. Mitra, and D. Chen, "Hybrid quick error detection (H-QED): Accelerator validation and debug using high-level synthesis principles," in DAC, 2015, pp. 53:1–53:6.

[9] H. Zheng, S. Gurumani, L. Yang, D. Chen, and K. Rupnow, "High-level synthesis with behavioral level multi-cycle path analysis," in FPL, 2013.

[10] H. Zheng, S. Gurumani, L. Yang, D. Chen, and K. Rupnow, "High-level synthesis with behavioral-level multicycle path analysis," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 33, no. 12, pp. 1832–1845, Dec 2014.

[11] Kaleidoscope: Adding JIT and Optimizer Support, http://llvm.org/docs/tutorial/LangImpl4.html.

[12] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in ISCAS, 2008, pp. 1192–1195.

Author's Profile

P.Usharani received B.Tech in Electronics and Communication Engineering from SKR College of Engineering, Nellore affiliated to the Jawaharlal Nehru technological university Anatapur in 2011, and pursing M. Tech in VLSI and Embedded systems from SKR College of Engineering affiliated to the Jawaharlal Nehru technological university Anantapur in 2017, respectively.

Ms. KAPULURU LEELAVATHI as Asst Professor Department of ECE.Qualification: M.Tech SKR College of Engineering & Technology
Email ID: leelavathi256@gmail.com