LNAST: A Language Neutral Intermediate Representation for Hardware Description Languages

Sheng-Hong Wang, Akash Sridhar, Jose Renau
Dept. of Computer Science and Engineering,
University of California Santa Cruz.
http://masc.cse.ucsc.edu

ABSTRACT

In this paper, we extend the ideas from LGraph to propose a new intermediate representation(IR) called Language Neutral Abstract Syntax Tree(LNAST) to support the Live Hardware Development(LiveHD) framework. The LiveHD framework focuses on providing live feedback to small changes made in the hardware design using LGraph as the unified VLSI data model. LNAST offers a three-fold benefit to the LiveHD flow. First, it acts as a bridge for the LiveHD flow to interface with different HDLs at the front end. Second, it generates multiple HDL/C++ code from LGraph at the backend. When it is complete, LNAST will support Verilog, SystemVerlilog and other modern HDLs like Chisel/FIRRTL and Pyrope, we also launch a plan to support HLS. Third, the custom IR helps to accelerate the front-end translation from HDLs to LGraph.

1. INTRODUCTION

The hardware development cycle is prolonged and tedious, compared to the faster design turnaround time of the software development model. In industry, there is a long-standing problem of hardware designers spending hours or even days to wait for the entire design flow to complete. Inspired by the state-of-art incremental synthesis techniques [15, 16], we started to build a "live" hardware development flow called LiveHD. The ultimate goal of LiveHD is to provide the design feedback in a few seconds.

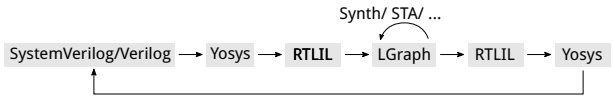

Figure 1: Current LiveHD flow.

LGraph [14] provides the foundation to achieve the fast design feedback goal of LiveHD. It is a unified VLSI data model with a focus on building agile infrastructures for LiveHD. Figure 1 depicts the current LiveHD flow. LiveHD integrates LGraph with Yosys [20] for design elaboration, with ABC [4] and Mockturtle [19] for technology mapping and synthesis, and with OpenTimer [8] for timing analysis. LGraph uses memory maps to support rapid loading/unloading of the design from the disk. Thus, it has a smaller read and write times, and this avoids the need to re-parse the netlist repeatedly between individual stages in LiveHD.

In the current LiveHD framework, we found three bottlenecks which could potentially hinder our goal in achieving live feedback. First, LiveHD uses Yosy to interface System Verilog and genereate an internal graph-like representation called RTLIL [20]. A translation layer converts the RTLIL to LGraph. Besides traditional HDLs like Verilog, SystemVerilog, and VHDL, there are many modern, open-source, active, and upcoming HDLs in the hardware design community like Chisel/FIRRTL [3, 9] and Pyrope [17, 18]. The translation pass must convert these modern HDLs into LGraph to extend the usability of LGraph beyond the traditional HDLs. The reality is that the translations have many similarities across HDLs, and care must be taken to avoid code replications.

Second, the current LGraph-Yosys interface takes several minutes to elaborate a large design with millions of gates. This long translation time impedes our goal of getting live feedback. The multi-layer translation in the LGraph-Yosys interface contributes to this delay. To begin with, the Verilog/SystemVerilog AST translates to Yosys RTLIL, followed by the Yosys RTLIL *proc* command [1] and the translation ends with generation of LGraph. A translation interface with minimal overhead is needed to accelerate the front end.

Third, LiveHD aims to support both synthesis and simulation. To do so, it has to convert the high-level HDL code to support synthesis and generate fast C++ for simulation(similar to what Verilator [2] does). The current LGraph model is a low-level graph representation for hardware design. The semantic gap between the high-level HDLs and the low-level LGraph IR must be bridged to ease the generation of human-readable C++, Verilog, and other HDLs.

We introduce LNAST, a high-level IR to bridge the gap between LGraph and multiple high-level HDLs. The combination of LNAST and LGraph in the LiveHD framework addresses the concerns raised earlier in this section.

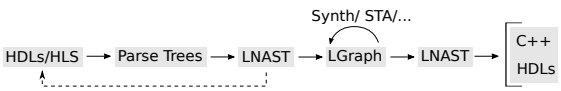

Figure 2: The new LiveHD flow with LNAST.

The IR design is a critical component in compiler de-

[1]Yosys proc is the command to translate high-level RTLIL to flops, muxes...

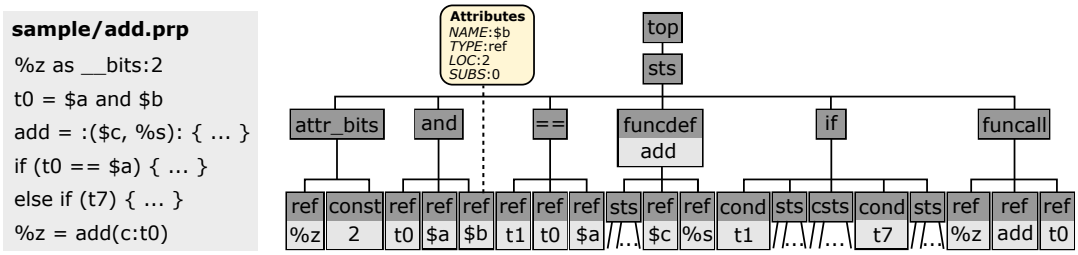

Figure 3: A sample Pyrope code and its LNAST representation. The code contains a bit width declaration of the top output %z, an *AND* operation of two top inputs $a and $b, an *add*() function definition with submodule IO $c and %s, a function call on *add*() with input assignment t0 to $c, and an if-else block.

velopment [10,11]. It is common practice for a compiler tool stack to have multiple layers of IR, ranging from high-level to low-level representations. A good high-level IR has simple semantics to express the high-level source language, and it hides details about the language syntax to the back-end compiler stack. It has to be independent of the programming language used and leverage common code optimization. In the new LiveHD model, we can view LNAST as a tree-like high-level IR, and LGraph as a graph-like low-level IR.

LNAST plus LGraph model is similar to Truffle and GraalVM [7], which also maintain a language-neutral AST for dynamically typed languages such as Python and JavaScript. GraalVM also has a low-level IR based on the LLVM IR.

Figure 2 depicts the new LiveHD framework. LNAST replaces the original LGraph-Yosys interface, and this eliminates multi-translations from the previous model. The language-agnostic nature of LNAST IR helps LiveHD to target both traditional Verilog/SystemVerilog and modern HDLs. The simple but expressive node-type definition in LNAST provides a clear representation of modern HDL semantics. LNAST has Static Single Assignment(SSA) [6] transformation to enable efficient conversion to LGraph. LNAST also facilitates the code-generation back from LGraph to the high-level source programs.

The rest of this paper is organized as follow: Section 2 describes the LNAST model and its internals. Section 3 reports the results, and Section 4 compares other similar works. Section 5 concludes the paper.

2. LNAST

In this section, we explain the LNAST model and its internal structure.

2.1 LNAST Internal Structure

2.1.1 Tree Structure and Node Attributes

Figure 3 illustrates the tree-like structure of LNAST. The LNAST tree represents every module in the circuit. The *top* and the *statements*(denoted as *sts* in Figure 3) node make up the root of the tree. The children of *statements* are generated in source code order. A primitive operation generates a sub-tree with the operator as the parent and the source and destination operands as leaves. The *and* operation in Figure 3 is an example of primitive. Another case of sub-tree generation results from source code hierarchy. For instance, an if-else or a function definition code block generates a new hierarchical sub-tree and has its own *statements* node.

Each LNAST node has four attributes: *NAME*, *TYPE*, *LOC*, and *SUBS*. *NAME* points to the variable name in the source code, *TYPE* is the node type in LNAST definition, and *LOC* denotes the line of code. *SUBS* is the subscript of a *NAME*, and it aids in SSA transformations.

2.1.2 Neutral Node Types for HDLs

LNAST node types are intentionally designed to capture the shared properties across different HDLs and aims to maximize expressibility. Node types are categorized into four groups: *structural*, *variable*, *attribute*, and *primitive_op*.

The *structural* group form the skeleton of LNAST and represent the program control flow. They define major node types like *if*, *for*, and *while*. LNAST to LGraph transformation flattens the sub-trees generated by loops.

The variables in the source code are classified as constants or references by the *variable* group. Notice that there are no specific type definitions for circuit input, output, and register. Special-characters prefix these circuit components. For example, $\$x$ denotes that x is a module input, $\%y$ is a module output, and $@z$ is a register.

The *attribute* group helps LNAST to express the attributes of a variable. In the first line of the source code in Figure 3, the variable's bit width attribute is expressed as an *attr_bits*. The *primitive_op* group includes common operations across different HDLs. For instance, many HDLs have similar logical, arithmetic, and comparison operations. A specific example is the addition operation.

2.2 LNAST Live Support

2.2.1 Persistence

We laid heavy emphasis on the speed of LNAST data persistence. LNAST uses memory-mapping like LGraph IR to speed up reads and writes. In LNAST, the source code is memory-mapped on to virtual memory. Lexing is done to tokenize the source code, and these tokens are stored in a memory-mapped vector. To record the token of the *name* field in LNAST nodes, we store the index in the memory-mapped vector instead of the plain string. It ensures that the strings are not manipulated directly and avoids additional memory operations.

2.2.2 *Fast Split*

Fast design split is a critical LiveHD infrastructure. During parsing, it breaks a monolithic hierarchical design into sub-module Verilog files. The parsed tokens are stored in persistent vectors. If a source code change is detected, all LNASTs are not rebuilt from scratch. We first compare the old and new persistent token vectors and only rebuild the LNASTs whose associated source code is modified.

2.3 LNAST Transformations in LiveHD

2.3.1 *From HDLs to LNAST*

In compiler design, a parser is used to scan the source code tokens and generate the parse tree. In LiveHD, different HDLs' source code is parsed into a language-specific parse tree, and translated into LNAST IR. The difference between the LNAST and a parse tree is that LNAST focuses more on representing useful abstract information from the components of the source code such as conditional loop blocks, whereas a parse tree captures low-level details of syntax, for instance, brackets and parentheses.

There are different front end parsers to its corresponding HDL in LiveHD. To avoid code replications, these parsers and LNAST share a common tree library for building the tree data structures.

There is a working prototype flow from Pyrope HDL to LNAST. Currently, we are building Verilog2005 and SystemVerilog to LNAST flow. The goal is to make LNAST represent fully synthesizable Verilog code and not just the netlist syntax.

2.3.2 *From LNAST to HDLs*

A key motivation for conceiving the idea of LNAST is to support multi-HDL code generation. This pass is still under development. In Figure 2, notice that LNAST generates HDLs, but it is doesn't have to go through the LGraph IR to achieve this. As shown by the dotted arrow, there could be a closed-loop transformation from the HDL to a language-specific parse tree, and then to an LNAST, and back to the HDL directly without entering the LGraph stage. This shortcut transformation loop is useful to verify the correctness between LNAST and HDLs quickly.

2.3.3 *From LNAST to LGraph*

The graph-based representation of LGraph IR makes it virtually the same as an SSA. The SSA is a widely used compiler optimization technique. It assures that every variable in the IR is assigned exactly once and describes the use-def chains explicitly, which in turn helps the LGraph optimization passes. To bridge LNAST to LGraph easily, we perform the SSA transformation on LNAST after building the primitive LNAST. Different SSA definitions from the same variable are represented in the SUBS (subscript) field in different LNAST nodes.

2.3.4 *From LGraph to LNAST*

We are currently working on translating LGraph IR back to LNAST IR. It would be the stepping stone to perform HDL code generation from the synthesized LGraph. However, when the lower-level LGraph IR expands from higher-level LNAST IR, the structural information could be lost. Take the conditional loop as an example. The loop node in LNAST is flattened in the LGraph Data Flow Graph(DFG) analysis. Therefore, the potential challenges would be identifying the corresponding LGraph region and tag the associated LGraph nodes, which would be useful when folding the loop region back to an LNAST node.

3. EVALUATION

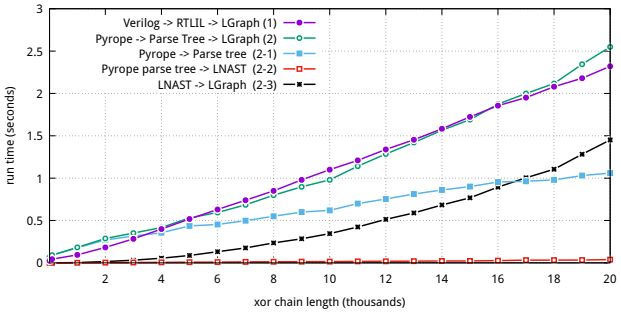

Figure 4: New LiveHD flow is competitive to old one for tested circuits.

3.1 Setup

We compared the original LiveHD flow in Verilog and the new LiveHD flow in Pyrope. The old flow uses Yosys to elaborate a Verilog file into a RTLIL, then converts the RTLIL to LGraph. The new flow uses a Pyrope parser in JavaScript to create the parse tree text, converts the parse tree text to LNAST, and transforms the LNAST to LGraph. The target circuits are simple xor logic chains ranging from 1 to 20,000 gates. Each xor operation takes one line of code. We ran our experiments on an Intel Core i7-6700K CPU @ 4.20 GHz with 16 GB of memory, running Manjaro v5.2.8-1 and used gcc v9.1.0.

3.2 Results

Figure 4 presents the runtime comparison of the new and the old LiveHD flow. For a 20k xor chain circuit, the former takes 2.54s in total, and the latter completes in 2.32s. On further observation, we notice that the Pyrope parser takes 1.01s to generate the parse tree text. This parser is written in JavaScript for easy prototyping with a performance tradeoff. We are currently implementing a new Pyrope parser in C++ to speed up the parsing phase. As memory mapping is used to construct LNAST from the parse tree, only the pointer of tokens are stored in memory instead of copying the entire string. Therefore, this conversion speed is very rapid, and it only takes 0.034s to generate LNAST.

For the LNAST to LGraph transformation, we notice an 80% performance overhead due to vector copy operations. We are currently working on its optimization, and we expect the LNAST to LGraph conversion to have a performance boost.

4. RELATED WORK

Hardware IR: FIRRTL and RTLIL are two open-source hardware IRs that target RTL and netlist representation. Similar to LNAST, their standardized intermediate representation of elaborated circuit makes it possible for circuit designers to convert the design into the IR and then continue to perform simulation and synthesis using the supported toolchain. Nevertheless, both FIRRTL and RTLIL are tightly knit to their target language and difficult to use as general-purpose IRs. Ross Daly proposed CoreIR [12] to interact with different HDLs, but the work focused more on formal verification support. Another hardware IR is netlistDB [1] which has a similar goal as LNAST to target different HDLs, but LNAST leverages the back-end infrastructures and all tools of LiveHD.

Software IR: Truffle is a framework to bridge different languages with Graal Java virtual machine. In this framework, the AST of different languages is mapped to the common Truffle AST. A series of optimization techniques like tree rewriting and just-in-time(JIT) compiling in the back-end GraalVM are applied to the common Truffle AST. Click and Paleczny [5] present a graph-based SSA intermediate representation to express optimization elegantly. The Common Intermediate Language (CIL) [13] is used in the .NET system and is also an IR designed for multiple languages such as C# and Basic.

5. CONCLUSIONS

We present LNAST, a high-level tree-based IR to bridge the gap between LiveHD and modern HDLs. LNAST helps different HDLs to interface with LiveHD flow and supports the code generation pass to perform LiveHD simulation and verification. The tree-like structure of LNAST facilitates rebuilding only the modified sub-trees, and this is useful to develop future live techniques.

The experimental result shows that the prototype of new LiveHD flow is competitive to the old one for a 20k lines source code. As a part of future work, we plan to optimize the conversion flow from LNAST to LGraph. The integration of LNAST in LiveHD flow is still in a preliminary phase. Future projects include interfacing the FIRRTL and Verilog/SystemVerilog front-end, designing a new fast C++ Pyrope parser, generating C++ code for simulation, generating HDLs for verification, optimizing tree traversal speed, and integrating live techniques like LiveSynth/SMatch [15, 16] into LiveHD framework.

6. ACKNOWLEDGMENTS

This work has been supported by the Center for Research in Open-Source Software (CROSS) at UC Santa Cruz, which is funded by a donation from Sage Weil and industry memberships.

7. REFERENCES

[1] "netlistDB," https://github.com/HardwareIR/netlistDB, online; accessed on 5 September 2019.

[2] Verilator 4.018, open source tool for verilog hdl simulation. Online Webpage. https://www.veripool.org/wiki/verilator.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[4] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.

[5] C. Click and M. Paleczny, "A simple graph-based intermediate representation," *ACM Sigplan Notices*, vol. 30, no. 3, pp. 35–49, 1995.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[7] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck, "High-performance cross-language interoperability in a multi-language runtime," in *ACM SIGPLAN Notices*, vol. 51, no. 2. ACM, 2015, pp. 78–90.

[8] T.-W. Huang and M. D. F. Wong, "OpenTimer: A high-performance timing analysis tool," in *Computer-Aided Design, Proceedings of the IEEE/ACM International Conference on*, ser. ICCAD'15. Piscataway, NJ, USA: IEEE Press, Nov. 2015, pp. 895–902.

[9] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson *et al.*, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 209–216.

[10] R. Koschke, J.-F. Girard, and M. Wurthner, "An intermediate representation for integrating reverse engineering analyses," in *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No. 98TB100261)*. IEEE, 1998, pp. 241–250.

[11] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[12] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan, "Cosa: Integrated verification for agile hardware design," in *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018, pp. 2–5.

[13] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of c programs," in *International Conference on Compiler Construction*. Springer, 2002, pp. 213–228.

[14] R. T. Possignolo, S. H. Wang, H. Skinner, and J. Renau, "LGraph: A multilanguage open-source database," in *Open-Source EDA Technology, Proceedings of the First Workshop on*, ser. WOSET'18, Oct. 2018.

[15] R. T. Possignolo and J. Renau, "LiveSynth: Towards an interactive synthesis flow," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 74.

[16] ——, "SMatch: Structural matching for fast resynthesis in fpgas," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 75.

[17] H. Skinner, R. T. Possignolo, and J. Renau, "Liam: an actor based programming model for hdls." in *MEMOCODE*, 2017, pp. 185–188.

[18] H. Skinner, S. H. Wang, A. Sridhar, R. T. Possignolo, K. Mayer, and J. Renau, "Pyrope," https://masc.soe.ucsc.edu/pyrope.html, online; accessed on 5 September 2019.

[19] M. Soeken, H. Riener, W. Haaswijk, and G. De Micheli, "The EPFL logic synthesis libraries," May 2018, arXiv:1805.05121.

[20] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/, 2019, online; accessed on 5 September 2019.