UNIVERSITY OF CALIFORNIA
SANTA CRUZ

BRIDGING INTERMEDIATE REPRESENTATIONS TO ACHIEVE HARDWARE DESIGN LANGUAGE TRANSLATION

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Hunter James Coffman

September 2020

The Thesis of
Hunter James Coffman is approved:

Professor Jose Renau, Chair

Professor Scott Beamer

Professor Heiner Litz

Quentin Williams
Interim Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Bridging Intermediate Representations to Achieve Hardware Design Language Translation

by

Hunter James Coffman

The hardware industry is currently beginning a trend towards a more productive hardware design flow. Many designers have been noticing for years a lack of efficiency when it comes to the design process. In response, research groups have been proposing tools and languages to improve productivity in this domain. Two such examples are LiveHD and Chisel. LiveHD is a framework that seeks to perform live synthesis and simulation, where live means that results for these tasks are achieved in seconds or minutes rather than hours. Chisel is a new hardware description language that focuses on using software abstractions to improve design reuse. The compilers for both works rely upon intermediate representations to model a design for their compiler. Having LiveHD and Chisel interact is currently impossible since the way in which they model hardware designs is different. This paper proposes an implementation of a translator that would bridge the gap between these two research efforts, allowing designers to reap the benefits of both and thus greatly improve their productivity during the design process.

Acknowledgments

Working with the Micro-Architecture Santa Cruz lab has been nothing short of a wonderful experience. Professor Jose Renau and Sheng-Hong Wang have helped foster my education over the past two and a half years and have opened the doorway for me to pursue academic research in a way I never would've expected prior to joining the lab. Both of them have pulled me out of a great number of problems, and I owe a large portion of my success to their willingness to help me out even at the beginning of my journey.

I also want to thank Professor Scott Beamer and Professor Heiner Litz for serving as members of the committee for this paper. Both of them were instructors for me prior to serving on my committee and both of the classes I took with them were of the most enjoyable that I can remember. The accelerator-based knowledge and parallel-processing knowledge I learned from them was incredibly useful and played an integral role in getting a position at a prestigious company.

Finally, I would like to extend a general thanks to the Chisel, FIRRTL, and RISC-V community. Having talked and worked with so many people of that community, I think that they truly have a chance to change the hardware industry and I wholeheartedly support them in their endeavours.

Chapter 1

Introduction

The modern hardware design flow is inefficient. One reason that this is the case is because tasks such as simulation and synthesis take large amounts of time to complete when performed on massive designs, often on the scale of hours if not days. This leads to inefficiency because if a bug is caught after a physical design step is finished and a change has to be made in the design, the entire physical design process has to be restarted even if the changes are small. Another reason is that hardware designers primarily write their designs in Verilog or SystemVerilog, languages which lack the expressive power that many other modern languages thrive upon. While SystemVerilog does have some built-in software abstractions, many of them are not synthesizable meaning that they cannot be used in physical hardware design. In attempts to give designers the ability to use modern software abstractions like types, polymorphism, and parameterization, several research groups have started building their own hardware description languages (HDLs) [5, 6, 9, 14, 15, 20, 21]. More over, these languages feature these software abstractions as

synthesizable features unlike SystemVerilog.

Live Hardware Development (LiveHD) is an open-source framework that addresses these inefficiencies and facilitates a live hardware design flow [18, 22, 23]. In this context, live is used to indicate that designs with small changes will be able to return simulation or synthesis results in seconds rather than hours or days, assuming they were synthesized or simulated before the changes. This is achieved by using a technique known as incremental synthesis [7, 16, 17]. LiveHD also strives to be the LLVM of the open-source hardware community by supporting interfaces with many different HDLs, such as Verilog and Pyrope [20, 23]. This is enabled by one of the core components of LiveHD, known as LNAST. LNAST is a language-agnostic data model that is used to represent a hardware design. This data model focuses on the semantics of a each operation in the design rather than the syntax used to perform that operation. Using LNAST, it is possible to design an interface for any HDL and allow designs written in that language to gain the benefits provided by LiveHD.

One HDL that has recently been growing in popularity is Chisel [6]. Chisel is a language embedded in Scala which offers designers all the aforementioned software abstractions as well as the benefits of functional programming. Since Chisel has become quite prominent amongst the open-source community and even used by some industry groups, it makes sense that LiveHD should have an interface to support it. Another language designed alongside Chisel is FIRRTL [12, 13], which is a lowered form of Chisel where many of the software abstractions have been handled. For example, Chisel performs loop-unrolling when generating FIRRTL. Anything written in Chisel can be

emulated using FIRRTL. Thus, since FIRRTL will have already handled any software abstractions, it will be used as the language for this interface instead of Chisel. The relationship between Chisel and FIRRTL will be discussed more in the next chapter.

Creating a way for Chisel to interface with LiveHD comes with several challenges that will be discussed further in this thesis. The design of this interface is important exploratory research as it shows that LiveHD truly does have the ability to interface with a highly verbose and complex HDL such as Chisel. This research also helped find problems within LiveHD that lead to several changes internal to the framework. As was mentioned before, LiveHD plans to support live mechanisms in the future for faster generation of synthesis and simulation results. Chisel designers who use this interface will be able to use LiveHD's flow, increase their overall productivity by utilizing these live feedback mechanisms, and make changes without having to worry about massive delays in between receiving synthesis and simulation results.

The contributions of this thesis are summarized as follows. Most prominently, LiveHD is now able to interface with Chisel. More specifically, LiveHD can now map any Chisel design to LiveHD's internal design representation, via a lowered Chisel form known as FIRRTL. As well, LiveHD is now able to leverage its internal design representation to generate matching FIRRTL code. All of these contributions have been upstreamed into the LiveHD repository which is found on GitHub [1].

The rest of this thesis is broken down into sections that focus deeply into the details of this work. Chapter 2 discusses background knowledge that is required to

[1]The repository can be accessed at `https://github.com/masc-ucsc/livehd`. The implementation of the work mentioned in this thesis is found in the "inou/ firrtl " and "pass/lnast_fromlg" directories.

understand this work as well as the context of why this work is important. Chapter 3 details how Chisel/FIRRTL is translated to LNAST. Chapter 4 takes a step away from the interface to discuss the LGraph to LNAST translation, where LGraph is a netlist-like way to model a hardware design. Chapter 5 will return to the LNAST-FIRRTL interface and discuss how FIRRTL can be generated from an LNAST. Chapter 6 compares the speed of these translation steps against the FIRRTL compiler. Finally, the thesis ends with Chapter 7 which summarizes the contributions of the thesis and suggests areas for future research.

Chapter 2

Background Information

2.1 IRs & ASTs

Topics that will be discussed in this thesis will often revolve around the concepts of intermediate representations (IRs) and abstract sytnax trees (ASTs).

An AST is a tree-like data structure that is regularly used by compilers to represent a program. Each node in the tree denotes some construct that is found in the that original program. Often these constructs indicate variable names, integer values, operators, or control-flow blocks such as if-else statements. Usually an AST will strip away some of, if not all of, the syntax of the language and focus on what the semantics behind the syntax. This means that inessential punctuation, such as semicolons or parentheses, are not present. This data structure is used in compilers because it closely matches the original program when traversed through. Additionally, it's easy to quickly iterate over the tree and perform analysis to see if the program is correct.

An IR is a data structure used by a compiler to represent the semantics of a program, usually with the syntax from the original program language stripped away. An AST is an example of an IR, but an IR does not have to be an AST. IRs can take the form of whatever best suits the intent behind the compiler the IR is used in. Another common data-structure used as an IR is a graph since it can be used to show the dependency of certain operations on one another.

A good IR is incredibly important in making an efficient compiler. If one wanted to build a compiler that took in n source languages and wanted to compile for m unique architectures, a naive implementation would require building $n * m$ compiler passes where a pass would take one source language and translate it to one architecture. With an IR that has abstracted enough information away but preserved the semantics of the original program, a compiler would instead only need n passes that go from a language to the IR and m passes to go from the IR to an architecture. Beyond saving a programmer from the redundancy of making far more passes than necessary, the IR also means that each pass does not have to be cluttered with specific details of the source language or machine architecture.

In this thesis, three IRs will be looked at: language-neutral abstract syntax tree (LNAST), LGraph, and FIRRTL. LNAST and LGraph are both IRs used in the LiveHD framework to represent hardware design. The LNAST IR is similar to an AST, while the LGraph IR is a bi-directional graph. FIRRTL is both the name used by the language itself and the AST-like IR used to represent that language inside of its compiler. In Sections 2.2.1, 2.2.2, and 2.3 each of these IRs will be discussed.

2.2 LiveHD

LiveHD provides an efficient and responsive hardware development flow that integrates the many steps of development into one single framework. Though open-source tools exist for each of the steps involved in physical design, the largest challenge when integrating all of them together is that none of them model a design in the same way. This means that the result from one physical design step cannot simply be passed to the next step, since the way they model the design is different [8,10,19,24]. Additionally, the API is unique for each tool, which makes it difficult for a designer to manage the interface between each tool. LiveHD addresses this issue by using only two IRs: LNAST and LGraph [18,22]. By having LNAST as a way to interface with HDLs and LGraph as a way to interface with physical design tools, a designer wouldn't have to worry about interfacing their design with any of the tool-specific data models since LiveHD would take care of that for them.

2.2.1 LNAST

An LNAST is a tree-like data structure that can model the AST of a given hardware design in a syntax-free way. To generate an LNAST representation of a design, the source code for a design is passed into an interface that translates from that source code's language to LNAST. LNAST serves as the higher-level representation of a design inside of LiveHD. Each of the node types inside of an LNAST represent either a name, some part of the design's control flow, or some sort of operation. A Pyrope code

example and its associated LNAST can be found in Figures 2.1 and 2.2. The LNAST is constructed by performing the subtraction first and then the addition of the constants to the result of the subtraction. This is because unlike some other HDLs, a variable has to be set prior to its use in the LNAST. As well, expression nesting is not allowed in the LNAST so the subtraction node could not be a child of the addition node. Finally, the result of all of that is assigned to the left-hand side of the statement in Pyrope.

```
1  total = (x - 1) + 3 + 2
```

Figure 2.1: Sample Pyrope code to create the LNAST found in Figure 2.2.

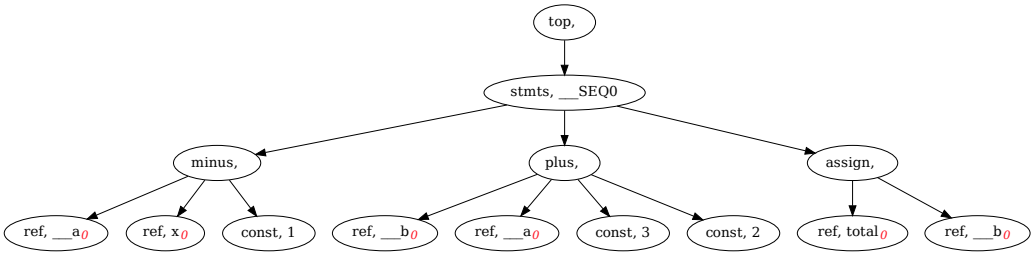

Figure 2.2: LNAST for a statement involving addition and subtraction.

LNAST is also useful for code-generation. As an example, one might want to translate Pyrope code to Verilog. So long as an interface between LNAST and the two languages exists, one could give a design written in Pyrope to LiveHD and turn it into its LNAST form. From the LNAST form, one could then use the interface to create matching Verilog.

Figure 2.3 shows the current design flow using LNAST. One other important

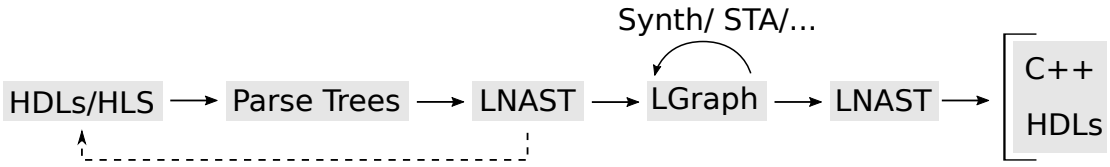

Figure 2.3: Current LiveHD flow, relying upon LNAST as the interface to go from a language to LGraph or for generating code from a given LGraph [23].

note about LNAST is that inputs are indicated with a `$` in front their name, while `%` is used to indicate an output and `#` indicates a register. More information regarding LNAST can be found in the next subsection as well as in [1, 23].

2.2.1.1 Tuples, Attributes, & Duck Typing

Inside of LNAST, tuples serve as the way in which data can be grouped together. These tuples are not statically defined like classes are in C++ or like bundles are in Chisel. Instead these tuples are dynamically defined which means that fields can be added or taken away from a tuple during run-time, on top of the usual expectation to be able to alter tuple values. This is often referred to as duck typing.

Some examples of ways to instantiate and modify a tuple in Pyrope can be found in Figure 2.4. Pyrope closely matches what can be done in LNAST so it serves as a useful language to write examples in. What can be noticed in that figure is that a tuple's subfields may or may not be named and can change as the program executes. A tuple can theoretically have an unlimited number of subfields. The subfields of a tuple are those elements of a tuple that have been instantiated. For tuple `foo`, `a` and `b` are fields but anything else would not be unless later specified in the design.

```
1  // Instantiate subfields a and b of tuple named 'foo'.
2  foo.a = 2
3  foo.b = 5
4
5  // Instantiate many subfields to tuple 'bar' all at once.
6  bar = (x = foo.a, y = foo.b, z = 7)
7  assert(bar.y = 5)
8
9  // Instantiate tuple 'qux' with unnamed subfields.
10 qux = (3, 2, 1)
11 assert(qux[0] == 3)
```

Figure 2.4: Sample Pyrope code to show different ways for tuples to be instantiated. Due to duck typing, the number of subfields of a given tuple can change as the program executes.

LNAST's foundation is deeply rooted in this typing scheme. Since there are no class definitions, trying to access a tuple's subfield is the only way to verify if that subfield is present. Besides allowing an LNAST to dynamically change what a tuple is, this duck typing scheme allows for the designer to choose whether or not they wish to specify certain attributes of circuit components. The most common application of this is specifying the `__bits` attribute for a component. This is one example of a compiler-based attribute but many others exist, such as setting a register's clock by setting its `__clk_pin` attribute. Compiler-based attributes are indicated with two underscores at the start of their name. These compiler-based attributes are handled differently than accesses to other tuple subfields. Those attributes specific to the compiler are resolved at compilation time whereas tuple subfields are resolved at runtime.

An access to a tuple's subfield is represented as a `dot` node in LNAST. This node type can be used for both setting as well as accessing an attribute of some circuit component. An LNAST which shows how to set the bitwidth of something can be seen

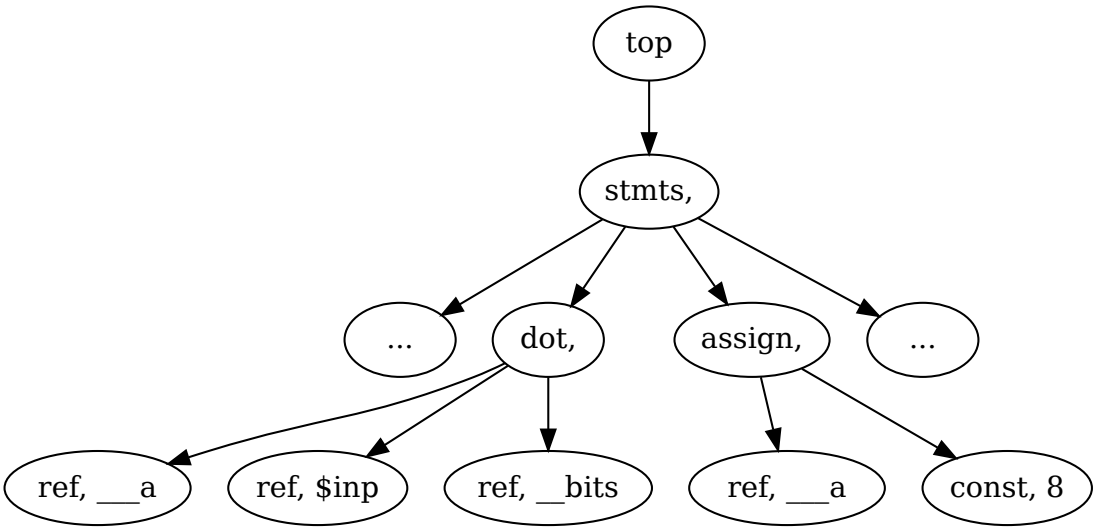

Figure 2.5: Example of how to set an attribute in an LNAST. A similar LNAST could be used to specify then set a field of a tuple.

in Figure 2.5.

There is a lot of power in this style of programming. Since everything can have attributes specified about it, this means everything can be a tuple. The definition of that tuple changes as more of these attributes are specified in the LNAST and more fields are potentially added to that tuple.

2.2.2 LGraph

LGraph is a graph-like structure that is meant to represent a design in a way similar to a netlist. An example of Verilog code and the LGraph generated from that code can be found in Figures 2.6 and 2.7, respectively.

LGraph is structured as a bidirectional graph where each node usually represents some sort of operation. Some of the most common node types can be found in

```
1  module simple_module(
2     input a,
3     input b,
4     input c,
5     output y
6  );
7
8     assign y = ((a & b) & !c) | ((a & b) & c);
9
10 endmodule
```

Figure 2.6: Sample Verilog code to create the LGraph found in Figure 2.7.

Sum_Op	Mult_Op	Equals_Op
Not_Op	And_Op	ShiftLeft_Op
SFlop_Op	AFlop_Op	Join_Op
Mux_Op	GraphIO_Op	SubGraph_Op

Table 2.1: A small selection of the node types found in LGraph.

Table 2.1. Each node has some number of input pins and output pins, specific to its node type. If one node's output pin has an edge connected to another node's input pin, that means the result of the first node's operation will be used as an operand in the second node's operation. In Figure 2.7, an example of this can be seen with node `n7_and`. That node has two edges connected to its input pins. The node's used to drive those output pins are `c` and `n8_and`. Since the output pin of `n8_and` represents `a & b`, this means that the two values being provided to `n7_and` are `a & b` and `c`. The output of that node would be `(a & b) & c`. If one wanted to know what the inputs to a given node are, they can iterate over each of the edges pointed to that node's input pins and see what each edge is pointing from.

Each node and its pins also have associated attributes. Some examples of attributes currently implemented are: a node's name, a pin's name, maximum and

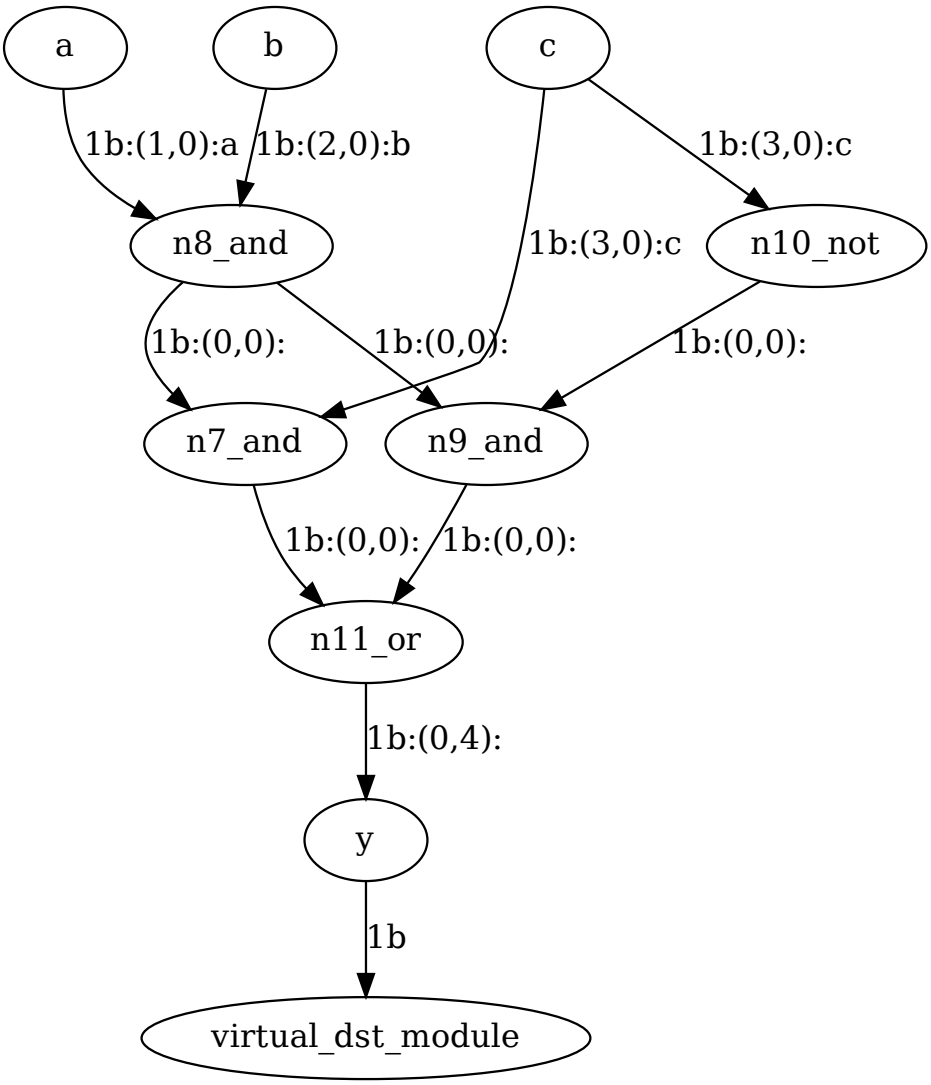

Figure 2.7: LGraph generated from Verilog code found in Figure 2.6. The information on each edge is for bitwidth, driver pin number, sink pin number, and finally name if the driver pin of the edge has one.

minimum values of a given register or wire, and hierarchy-specific information used when multiple modules of the same type are instantiated as submodules. Using much of this information, LGraph can perform a variety of different design optimizations such as bitwidth inference.

One shortcoming found in the open-source tools that LiveHD integrates with is that they require that an entire design is flattened before it can be dealt with, rather than allowing any sort of hierarchy. Flattening the entire design is problematic since it increases the complexity involved with physical synthesis. Industry tools often do not have this same limitation. To be able to support both forms, an LGraph can appear as hierarchical or as flattened based on which graph iterator API is invoked. Other major benefits of LGraph is that it reduces the amount of duplicated work necessary between physical design stages, it is implemented using a very fast memory-map library which can process information from large designs much faster than traditional storage techniques, and it has current support for some of the physical design stages already implemented.

More information can be obtained about LGraph by reading the most recent paper in which it was proposed [22].

2.2.3 Comparing LNAST & LGraph

LGraph and LNAST, despite both being a part of the same framework, serve very different purposes. Each IR provides certain benefits which was why they were chosen for their specific part of the framework.

LNAST is a powerful front-end IR because it completely focuses on semantics, like a good IR should, which means that any HDL can be translated from itself to LNAST form. In addition, the compilers for most languages already leverage an AST IR. This is convenient since being able to translate from one AST data structure to another AST data structure is much more straight-forward than translating from an AST to some other data structure.

LGraph is very similar to a netlist, which can make it very complex to parse and understand when compared to LNAST. Even more so, LGraph has many more intricacies and minor details associated with each of the nodes found in its graph-like structure. These complexities are not worthwhile to have on the front-end because it would make creating an interface a very difficult task. It serves as a suitable back-end however, since it is a bi-directional graph. This style of data structure is great because it shows how certain operations depend upon the results of other parts of a design. Often when making optimizations to a part of the design, that optimization may propagate to other parts of the design. Having edges between parts of the graph that are dependent upon one another allow for someone using LGraph to quickly identify what relevant parts of the IR need to be changed.

2.3 Chisel/FIRRTL

Chisel is a powerful HDL that was built with solving Verilog's inadequacies in mind. It focuses on introducing object-oriented programming and parameterization into

hardware design. The result is that it allows designers to construct circuit generators which greatly improve productivity and design reuse. A fantastic example of this is in creating a filter. One could design a filter with a form of parameterization that allows for control of something like input bitwidth similar to what can be done in SystemVerilog, but in Chisel one can also parameterize the filter condition to be some complex expression such as "filter out every value divisible by n". In SystemVerilog, this would not be possible since a designer can set parameters to be constant values, not expressions.

When a Chisel design is compiled, it transforms into a lower-level form known as FIRRTL [12, 13]. Many of the same ideas that inspired LGraph inspired FIRRTL. Since Chisel generates hardware but doesn't exactly describe it in the same way Verilog does, FIRRTL serves as the intermediary between the two. More precisely, it is a language that resembles Chisel after many of the software abstractions used in a design have been resolved. For these reasons, the interface between Chisel/FIRRTL and LiveHD will specifically be translating something written in FIRRTL into an LNAST form instead of handling any Chisel directly.

A FIRRTL user can pass their design to LiveHD by using Protobuf [11]. Protobuf is a tool that serializes structured data [2]. In this case, the FIRRTL compiler can parse generated FIRRTL then format it into a Protobuf message containing associated inputs, outputs, statements, and expressions.

Protobuf was chosen as the input form because it offers several benefits over other alternatives. The most prominent reason it was chosen is because Protobuf pro-

vides both serialization as well as deserialization. This means that no FIRRTL parser needed to be made in LiveHD and instead the FIRRTL design can be deconstructed by leveraging Protobuf's API. Another reason that Protobuf was used is because the way in which the FIRRTL compiler formats its Protobuf messages closely resembles that of an AST. This makes it a suitable data structure when translating to LNAST as they are very similar in structure, while also requiring less overall space compared to alternative approaches. Protobuf also supports backwards compatability and tracks versions, which is another useful benefit.

Chapter 3

Translating FIRRTL to LNAST

This section describes the process by which the FIRRTL-LNAST interface translates a FIRRTL design into LNAST form. It will start by presenting the structure of a FIRRTL design when represented as a Protobuf message. Then this section will describe how the interface translates FIRRTL statements to LNAST and will point out some cases where the process of translating is non-trivial.

3.1 FIRRTL-Protobuf Structure

A Protobuf message is structured in such a way that it is composed of sub-messages or primitive types, like strings or integers. This structure is nearly identical to classes and their definitions in C++. One of Protobuf's main draws over other serialization tools is that it allows for construction and deconstruction of messages in a straight-forward, object-oriented manner.

For FIRRTL, the Protobuf message their compiler creates is structured such

that it is composed of sub-messages that are elements like IO ports, types, statements, and expressions. In several cases, these message types are used as attributes of one another. For example, often a statement will have a left-hand side expression and a right-hand side expression. When trying to do some sort of operation, the right-hand side expression may be of a specific type that requires several sub-expression to represent each of its operands.

This way this information is structured is important to understand as it will serve as the foundation for this interface's implementation. However, for the rest of this chapter, FIRRTL will primarily be mentioned in language-form instead of in Protobuf-form for ease of readability. Recall that anything written in FIRRTL can be constructed in Protobuf form using the Chisel/FIRRTL compiler.

3.2 Translation Details

3.2.1 Inputs, Outputs, & Types

A feature of Chisel/FIRRTL that is lacking in Verilog is the ability to give circuit components types. Many types exist and users can implement their own, but the most commonly used types are UInt, SInt, Bool, Vector, and Bundle. Most of these types are what a programmer would expect, but the bundle type may not be so readily obvious. A bundle is a collection of signals, where each signal has its own type. The most common usage of this in Chisel is for wrapping module inputs and outputs into a single bundle. This means that to access any of the contents of the bundle, one must

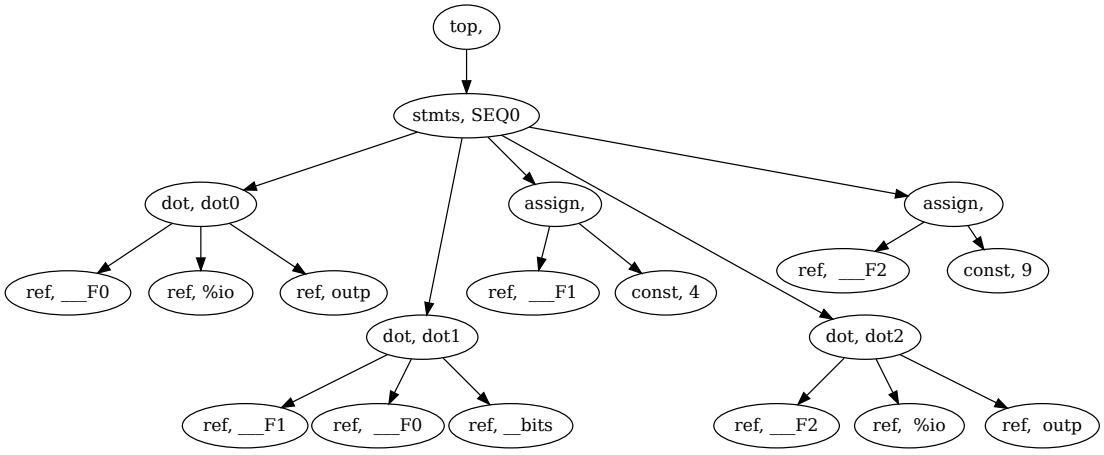

Figure 3.1: The LNAST for a design that has one output, `io.outp`, which has a bitwidth 4 and is assigned to the value of 9. Nodes dot0, dot1, and the first assign are used to set the bitwidth. Nodes dot2 and the second assign are used to set the value.

first access the bundle itself. In Chisel/FIRRTL, this is simple since the designers added syntax to easily do exactly that. Whenever one wants to access a member of a bundle, one would access it as `[bundle_name].[field_name]`. In LNAST, the ability to access these subfields requires the use of one or more `dot` nodes. Figure 3.1 shows what accessing a bundle element looks like in LNAST. As can be seen in that figure, LNAST can access bundle fields by creating a `dot` node where the second child is the bundle name, the third child is the field name, and the first child is the string one can use to access or set that field.

How LNAST handles inputs and outputs differs from FIRRTL or Verilog. In most HDLs, a designer has to explicitly define all of the inputs and outputs prior to use. LNAST is different since the only time inputs and outputs are specified is when they are used in a statement. As mentioned before, inputs and outputs are denoted

with either a `$` or `%` in front of their name. These symbols have to be applied to the name every time the signal is used. Since a FIRRTL design will not include these symbols, the interface has to add them in. This is accomplished by having two lists filled with names: one for inputs and the other for outputs. At the very beginning of the translation process for a module, all of that module's ports are looked at and their direction is determined. Once this information is gathered, the port name is added to one of the lists based on its direction. At this time, it is also checked to see if the port has its bitwidth specified in the FIRRTL design. If it is, that port's bitwidth is specified in LNAST by setting its `__bits` attribute using another `dot` node. Later, when looking at each of the statements that make up the FIRRTL design, all the signal names are checked to see if they are found on the input or output lists. If they are, then the appropriate symbol is appended to the front of their name.

One caveat about all of this is that LNAST does not support bundles that are bidirectional, though Chisel does. This means that Chisel/FIRRTL bundles used for module IO do not directly translate to a tuple of the same name in LNAST. Instead, that bundle has to be split into two bundles: one for inputs and another for outputs. If this is the case then, at the front of their names, all inputs will have `$inp_` and outputs will have `%out_`. Because it is split for all modules, this change will also be seen when dealing with signals serving as both inputs and outputs of submodules instances.

Expression Type	Description
Reference	a name that refers to some circuit component
[U/S]IntLiteral	an unsigned/signed integer which may include number of bits
Mux	a traditional two-to-one mux
SubField	a subelement of an expression that types as a bundle, ex. `io.a`
SubIndex	a static reference to a subelement of a vector, ex. `vec.2`
SubAccess	a dynamic reference to a subelement of a vector, ex. `vec[x]`
PrimOp	a primitive operation that takes in some arguments, ex. `add(a, b)`

Table 3.1: FIRRTL expression types.

3.2.2 Expressions

Several things qualify as an expression in FIRRTL so to help understand the different kinds of expressions, Table 3.1 has been provided. The PrimOp entry is special as there are many different types of primitive operations, such as binary operations, arithmetic operations, type conversions, bit selectors, along with many others.

What can make expressions tricky to deal with is that an expression in FIRRTL can be a sub-expression of some other expression. In FIRRTL, **and**(**not**(a), b) is a legal expression. To model this in LNAST, one cannot create one `AND` node and have its two children be **not**(a) and `b` since LNAST does not support expression nesting. An example of how this functionality would be modeled can be found in Figure 3.2. Since expression nesting is not permitted and LNAST requires that variables are specified prior to use, each sub-expression must be defined separately in the LNAST and then the parent expression can be defined using the results from each of the sub-expressions as needed. To handle any nested expressions, the interface recursively checks to see if a given expression has any sub-expressions as an argument. If it does, then it will handle

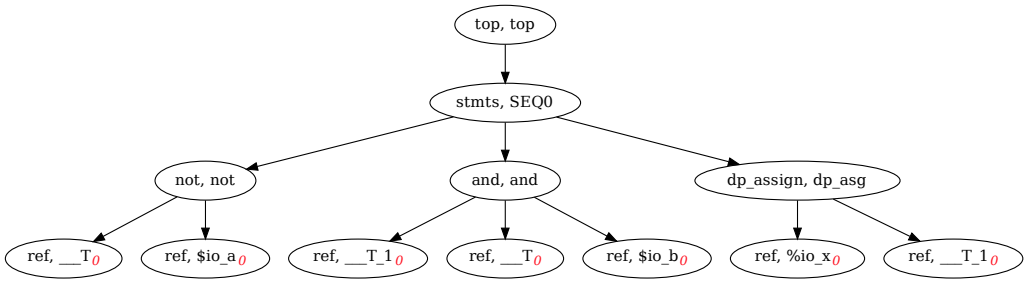

Figure 3.2: An LNAST showing how subexpressions are handled separately with intermediary names. Note that bitwidth information was removed from this graph to keep the figure condensed.

those in the same way. Once every sub-expression has been added into the LNAST, then the expression itself can be added to the LNAST. A high-level version of this traversal can be found in Algorithm 1.

Due to the variety of unique primitive operations, the interface includes several different handlers whose sole purpose is to translate one specific case of primitive operations. Some primitive operations have one-to-one mappings between FIRRTL and LNAST, like the `add` operator translating to a `plus` node in LNAST.

Other FIRRTL operations involve the use of multiple different LNAST node types to achieve the same functionality. One example of this is `tail`, which returns the n-least significant bits of some signal. This translation of this case, along with several others, is discussed in Section 3.3.

Algorithm 1 Add expression to LNAST

```
1: procedure InitialExprAdd(Lnast ln, FIR_Expr expr, Ln_node stmts, string lhs)
2:     if expr is a Reference or IntegerLiteral then
3:         asg_node ← create node of ASSIGN type as a child to stmts;
4:         Set asg_node children to be: lhs, expr's name;
5:     else if expr is a SubIndex then
6:         Get name to access SubIndex's expression (sie);
7:         sel_node ← create node of SELECT type as a child to stmts;
8:         Set sel_node children to be: lhs, sie, and the static index number;
9:     else if expr is a SubAccess then
10:        Get names to access SubAccess's expression (sae) and index (sai);
11:        sel_node ← create node of SELECT type as a child to stmts;
12:        Set sel_node children to be: lhs, sae, and sai;
13:    else if expr is a SubField then
14:        Create dot nodes to handle bundles, then get name to access SubField (sfn);
15:        asg_node ← create node of ASSIGN type as a child to stmts;
16:        Set asg_node children to be: lhs, sfgn;
17:    else// expr is of type Mux, ValidIf, or PrimOp
18:        Invoke specific handler based on expr's type to create many nodes;
19:    end if
20: end procedure
```

3.2.3 Statements

In a FIRRTL-Protobuf message, most statements consist of several sub-expressions. For example, the assign statement consists of a left-hand side expression and a right-hand side expression. Registers instantiation statements are also like this, where the clock, reset, and initial value are all just sub-expressions. What truly differs between these statements is how the interface uses these expressions. For the assign statement, an `assign` statement is made in the LNAST and the results of those two expressions act as the children to that node. A register instantiation statement would instead have many different `assign` statements that would be setting each of the register's attributes in the LNAST.

Another example to look at is FIRRTL's `when` statement type, which is semantically the same as an if-else block in most other languages. In this case, an **`if`** node has to be added to the LNAST. This node's children will be a string to access the condition, the statements to execute if true, and the statements to execute if false. All sub-statements will then be added as children to these new statement nodes. The same logic is effectively used for `mux` and `validif` statements.

Memory blocks can also be specified using FIRRTL and are treated as statements. The main task of translation in this case is to make sure that any specified attributes, like depth or number of read ports, get specified using `dot` nodes in the LNAST. Statements that specify register attributes work in a very similar way. Some attributes are mandatory to have defined, like depth for memory, but not all are required. Some things, like a register's clock, can be left undefined and will be set to a default value. For the cases where a mandatory attribute is not defined in the FIRRTL, then the interface will fail to translate and will present the user with a relevant error message.

Another statement type is a submodule instance. In FIRRTL, this statement specifies an instance name as well as what kind of submodule is being created. This statement does not connect any of the inputs or outputs; instead, it creates a submodule with none of its signals attached yet. The equivalent node type in LNAST is FN_CALL which requires three children: the bundle name for the output signals, the type of submodule being made, and the bundle name for the input signals. Since none of the input signals will have been attached yet, because FIRRTL specifies those after the

instance is made, the bundle name for the inputs must have the `__wire` attribute set to true. That bundle will be forced to act as a wire in LNAST, instead of as a software-like variable similar to everything else in LNAST.

3.3 Non-Trivial Translation Cases

To understand all of the translations that this interface does, a deeper understanding of how an LNAST is interpreted is required. To help the reader gain this intuition, two such cases will be discussed in this section.

3.3.1 Tail Operation

The `tail` operator in FIRRTL takes a signal and returns the n-least significant bits of that signal, where n is some specified argument. LNAST does not have a node type that does this, but that functionality can be achieved through a series of three nodes which can be seen in Figure 3.3. The first node performs a shift right based on the n value. This is done because when the LNAST is interpreted, the bitwidth of the signal being assigned to by the shift right will be n fewer bits than the thing being shifted. For example, if a six-bit signal is shifted right once then the result should only have five bits. This shift right does not provide the final value, but is instead done with the intention of setting the signal's bitwidth. The assign node is used as a way to implicitly set the number of bits, as it will evaluate the number of bits on the right-hand side and set the left-hand side's number of bits to be equivalent. Finally, a dp_assign is used as the actual assignment. A dp_assign drops overflow bits when assigning, if

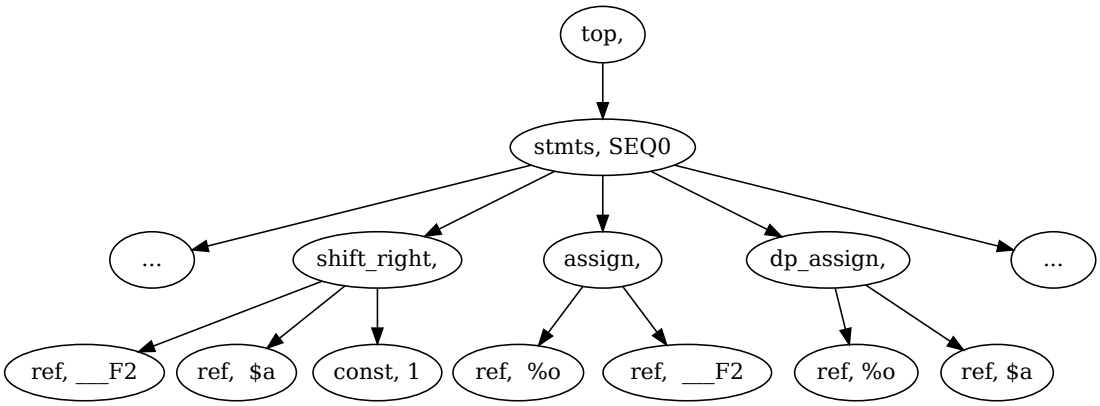

Figure 3.3: Translation of `o = tail(a, 1)` to LNAST, where `a` is an input and `o` is an output.

necessary, so this will guarantee that what is assigned to the left-hand side is only the n least significant bits of the right-hand since the previous assign already limited the number of bits.

3.3.2 Concatenation

At the time of writing this, LNAST does not support a concatenation operator. This is because one of LNAST's underlying goals is to have a very small number of unique node types. In most cases when some operation can be done using other nodes, then a new node type does not need to be made for that specific operation. Though there are exceptions to this rule, particularly when the operation is very common, the concatenation operator was deemed unnecessary to add as a node type.

Using only nodes present in LNAST, the concatenation of two signals could be implemented as follows: `(e1 << e2.__bits) | e2`. The nodes that would be required to

```
./bazel-bin/main/lgshell
```

Figure 3.4: Command to open LiveHD shell, lgshell.

```
//To create an LNAST from a Protobuf file
inou.firrtl.tolnast files:path/to/file.pb
//To create a .dot file to see LNAST visually
inou.firrtl.tolnast files:path/to/file.pb |> inou.graphviz.from
```

Figure 3.5: LiveHD shell commands to use FIRRTL to LNAST interface.

achieve this would be a dot node, a shift_left node, and an **or** node.

Similar logic can be extended to FIRRTL's pad operator which pads a signal with zeroes until it is at least of some bitwidth. Using the underlying attributes of each signal in a design allows the interface to mimic the functionality of just about everything in FIRRTL without needing unique nodes in LNAST for every single possibility.

3.4 Usage

This interface can be used after building the LiveHD repository. The interface can be accessed by using LiveHD's shell, which can be entering using the command in Figure 3.4. Using the commands in Figure 3.5, a provided FIRRTL-Protobuf message can be put through this interface and translated into an LNAST. The "—¿" operator found within the second command is functionally the same as using the pipe operator on the command line.

Chapter 4

Translating LGraph to LNAST

Before discussing the translation of LNAST back to FIRRTL, the entirety of this work should be put in context. In LiveHD, the design optimizations and physical design steps all interface with LGraph. Thus, getting an LNAST from FIRRTL is useful but would serve little purpose other than helping translate between HDLs if that LNAST could not be translated into an LGraph. Likewise, code generation would be much simpler from an LNAST instead of an LGraph due to its AST-like structure. One researcher has been working on the translation from LNAST to LGraph, but that will not be detailed in this paper. Instead, the implementation of an LGraph to LNAST translator will be discussed.

As mentioned in Chapter 2, LGraph is a graph structure used to represent a design where an LNAST is a tree structure used for a similar purpose. There will be many similarities between this work and the work detailed in Chapter 3. The critical difference lies in the structure that is being translated from.

4.1 LGraph Traversal Algorithm

In LGraph, individual nodes may have both input pins and output pins. Input pins receive a value, while output pins provide a value. To keep our wording concise, a node's input pin will be referred to as a sink pin and a node's output pin will be called a driver pin. An edge from one node's driver pin to another node's sink pin indicates that the node whose sink pin is a part of that edge depends on the result of the other node's driver pin. A driver pin may be labeled with a name to indicate if that pin is a wire, register, or input in the graph's design, but a sink pin will never have a name. Any driver pin that doesn't have a name at the beginning of this algorithm is given a name that starts with `___` to indicate it as a compiler generated variable.

The edges of the LGraph are essential to the translator. When traversing through an LGraph, the translator will visit each node in the graph and look at that node's type as well as the edges connected to that node's sink pins. This information is then used to add a statement to the LNAST that is being created. The order in which nodes are visited is important. As was mentioned before, LNAST requires that a signal must be specified with a value before that signal can be used on the right-hand side of an expression, unless it is an input or register.

What this means for LGraph is if a node has input edges connected to other nodes, those other nodes need to have added their statement to the LNAST prior to the translator adding the current node's statement. This can be achieved by traversing through the graph in a topological-manner, starting from the outputs. This can be seen

Algorithm 2 Start traversal at output nodes

```
1: procedure HandleOutputNode(LGraph lg, Lnast ln, Lnast_node stmt_nd)
2:     for each output (outp) of lg do
3:         input_edge ← outp's input edge
4:         HandleSourceNode(input_edge.driver, ln, stmt_nd);
5:         /* This next function takes a pin, looks at that pin's node's
6:         * input edges, then uses the name of each edge's driver pin
7:         * as a child of the LNAST node that will be created. */
8:         AttachToLNAST(ln, stmt_nd, outp.get_driver_pin());
9:     end for
10: end procedure
```

in Algorithm 2 which invokes Algorithm 3.

This graph traversal algorithm utilizes a coloring scheme where white means that a node has not yet been visited, grey indicates that the node has been visited but its inputs haven't all been added to the LNAST yet, and black means that a node's statement has already been added to the LNAST. This coloring scheme was used to prevent adding the same statement to the LNAST multiple times unnecessarily. It also guarantees that if a node is colored black, every node who is an input to that node must also be colored black.

The most challenging part of this algorithm comes with the fact that a graph-cycle can exist inside of an LGraph. Using Verilog, an example of this could be generated by having some register `x` and performing `x = x − 1`. A topological traversal normally requires that no cycles exist within the graph being traversed. An example LGraph where this is the case can be found in Figure 4.1.

Using Algorithm 3, node $n7$ would be visited and marked grey, and shortly after the same thing would happen to nodes $n8$ then $n11$. When the algorithm is looking

Algorithm 3 Traverse over non-output node

```
1: procedure HandleSourceNode(LGNode_pin pin, Lnast ln, Lnast_node stmt_nd)
2:     if pin.get_node().get_color() == black then
3:         return;
4:     else if pin.get_node().get_color() == grey then
5:         /* Node has already been visited in algorithm. If this function
6:         * is invoked on a node already colored grey, then a cycle exists
7:         * in the graph, caused by a loop from a register's Q to Din. */
8:         for each of the input edges of pin.get_node() do
9:             edge_driver = input_edge.driver;
10:            if edge_driver.get_node().get_color() == white or grey then
11:                if !(edge_driver.get_node().get_type().op == [A/S/F]Flop_Op) then
12:                    HandleSourceNode(g, edge_driver, stmt_nd);
13:                end if
14:            end if
15:        end for
16:        pin.get_node().set_color(black);
17:        AttachToLNAST(ln, stmt_nd, pin);
18:        return;
19:    end if
20:    // Node hasn't been visited yet by algorithm.
21:    pin.get_node().set_color(grey);
22:    for each of the input edges of pin.get_node() do
23:        edge_driver = input_edge.driver;
24:        if edge_driver.get_node().get_color() == white or grey then
25:            HandleSourceNode(g, edge_driver, stmt_nd);
26:        end if
27:    end for
28:    if !pin.get_node().get_color() == black then
29:        pin.get_node().set_color(black);
30:        AttachToLNAST(ln, stmt_nd, pin);
31:    end if
32: end procedure
```

at $n11$'s inputs, it would notice that one of the inputs, $n7$, is marked grey which means that the value represented by that node has not yet been added to the LNAST. If the algorithm treated that as already added to the LNAST then the mux represented by $n11$ would have been added to the LNAST with its condition being whatever name

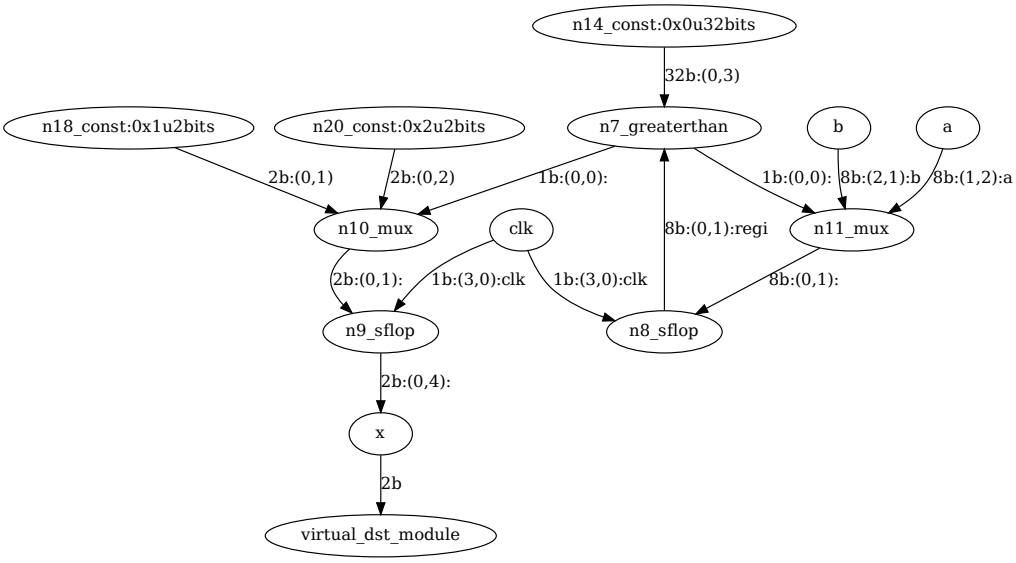

Figure 4.1: LGraph with a cycle consisting of **n7_greaterthan**, **n8_sflop**, and **n11_mux**.

is used for node $n7$. Problematically, node $n7$ would have not yet been added to the LNAST. This will not work since in LNAST a circuit component has to be set prior to use.

To get around this, the algorithm identifies that a graph-cycle is present and attempts to resolve that cycle. An observation can be made that a register, unlike other LNAST components, can be used before being set due to the persistent nature of registers. Based on this observation, the algorithm follows the cycle from where it began until reaching the register node that must necessarily exist in the cycle, represented by an SFlop_Op in this case. The nodes of that list are then added to the LNAST from most recently visited until reaching the start of the chain. Using Figure 4.1 as an example,

this would mean that only $n7$ would need to be added and then colored black from grey. The algorithm would then be able to resolve $n11$, $n8$, and finally would get back to $n7$ where that node would be ignored since it was already added to the LNAST.

4.2 Translation Explanations

To help the reader understand how the translation process works on the per-node level, this section provides explanations for how some specific translations work. The selected node types were chosen to highlight different aspects of LGraph and how these variants are handled.

4.2.1 Sum Operator

In order to describe how this translation works, more details regarding LGraph pins need to be known. Each node, in its class constructor, specifies certain sink pins and driver pins as well as what they represent. For instance, the node type for sums defines four sink pins: pin zero is for signed values that should be added, pin one is for unsigned values that should be added, and pins three and four are the same as the first two except they are meant for subtraction. If one wanted to add two unsigned values together, they would need to create two edges in a graph. Both edges would go from each of the node's driver pins to sink pin zero on the sum node. There is no limit on how many edges can have the same sink pin when discussing the sum operator.

This is relevant during translation because LNAST does not support having both addition and subtraction in one single node. Instead, the translator may have to

create both an addition node and a subtraction node in the LNAST, assuming there are edges connected to both the addition sink pins as well as the subtraction sink pins. Knowing which sink pins are used will also inform the translator whether the output needs to be signed. If only signed sink pins are used, then the output of this node should also be signed. This is done by creating a `dot` node in the LNAST which sets the result's `__signed` attribute to true.

4.2.2 And Operator

This operator is somewhat similar to the operator previously discussed, except that instead of having multiple sink pins this has multiple driver pins. Unlike the binary operation which only has two inputs, an `AND` node can have an infinite number of inputs. Driver pin zero performs the expected binary operation on every single input to the sink pin. This would be equivalent to `a & b & c & ...`. Driver pin one concatenates all of those connected inputs and performs an and-reduction upon all of these concatenated signals. This would be represented as `&{a,b,c}`. When handling operators that have multiple driver pins as outputs, the result for each driver pin has to be added to the LNAST since the LGraph node cannot be revisited later due to the fact that it will be colored black. Thus, every driver pin has to be given a name and have its associated statement added into the LNAST all at once.

For the specific case of an LGraph `AND` node, two groups of statements would have to be added to the LNAST in order to account for both driver pins. The first statement added to the LNAST would be a single `AND` that performs the binary

operation on every single input. Much like the LGraph, an LNAST allows for an AND node to have an infinite number of children. For an AND node in LNAST that has n children, the result would be performing a binary-and similar to what was mentioned for LGraph.

The second statement group has to perform concatenation, just like what was discussed in Section 3.3.2, and would have to check to see if this concatenated value was equal to negative one in twos complement. Checking if a value is equal to negative one is a clever way to do the same thing that an and-reduction operator would do: check to make sure all the bits are set to one.

One optimization can be made on top of this. Even though multiple driver pins exist, not all of them are necessarily used every time the node is present. Thus, before creating statements for a driver pin, it is best to only create those statements if that driver pin is used. This can be easily be checked by seeing if any edges connect to each driver pin. If there are none, then the statements for that driver pin should not be created.

4.2.3 Submodules

LNAST implements submodule instances as function calls, where these functions can have zero or more outputs. When creating one of these nodes, it will require three children: the first is the name of a tuple through which all the outputs can later be accessed, the second is the submodule name, and the third is a tuple that holds all the inputs to the submodule.

LGraph has a specific node type to handle submodules. The translator has to treat these submodule nodes like a black-box and ignore everything except the submodule's input and output names since there is no guarantee that a user will provide the submodule's LGraph. Thankfully, LGraph implements submodule nodes with this in mind. A submodule node has a number of sink pins equal to the number of inputs into the submodule, and the same relation exists between driver pins and submodule outputs. To create the tuple that stores all of the inputs for the function call, each of the node's input edges are looked at. The first edge looked at will be attached to sink pin zero, the next edge that will be looked at is attached to sink pin one, and this process will continue until finally all input edges have been looked at. When looking at these edges, the name of the driver pin will be concatenated onto the end of the input tuple. This will eventually mean that the translator has a single tuple with all of the inputs to the submodule.

The submodule name can be easily accessed using the submodule node and the output tuple name will be some temporary variable. Thus the function call node can be created in the LNAST, as seen in Figure 4.2. Finally, the names of each driver pin of the submodule have to be changed in the LGraph. Prior to this change, each driver pin will hold the name of the submodule output. For the translator, accessing the output signal using that name will not work because that signal is now a part of the output tuple. Thus, to access that signal, a `dot` node would have to be created. After the `dot` node is added to the LNAST, the driver pin name is replaced with the temporary variable name used for the `dot` node. Using Figure 4.2 as an example, the

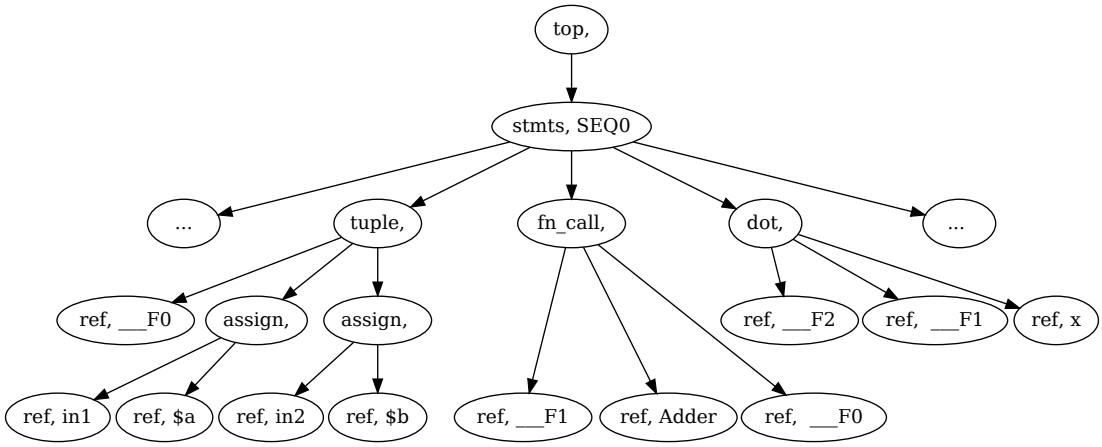

Figure 4.2: LNAST conversion when encountering some Adder submodule that has two inputs, `in1` and `in2`, and has an output port named `x`. `__F2` could later be used to access that submodule output.

submodule node's driver pin that was named `x` would be renamed to `__F2`.

4.3 Bitwidth Table

Looking ahead, the LNAST that is being generated from LGraph will later be used to generate code in some HDL. For many of these HDLs, knowing the bitwidths of circuit components is useful, if not mandatory. In FIRRTL, only the bitwidths for top-level module inputs need to be known. To accommodate for this, bitwidth information needs to be passed from LGraph to LNAST. It is fair to assume that the LGraph that is being translated from will have the bitwidths of everything specified, since there exists a part of the LiveHD compiler that determines the minimum bitwidths for all circuit components in an LGraph. For completeness, the translator tracks the bitwidths of everything in the entire circuit. This may seem unnecessary for FIRRTL, but the

decision to do so was made with generating other HDLs in mind. If one wanted to generate Verilog, then all these bitwidths would need to be known.

Two possibilities exist for tracking this information: creating a table that maps names to bitwidths or adding `dot` nodes in the LNAST to set every circuit component's `__bits` attribute. The latter option was originally used but was found to be cumbersome and incurring unnecessary performance penalties since all LNAST nodes had to be traversed over to find the bitwidth of some component. The former option had efficient lookup times, but wasted space when other attributes became important to track such as signedness of a component since most table entries wouldn't specify their signedness.

A hybrid of the two was eventually decided upon. A table is used to track only bitwidths for each circuit component, while properties such as signedness are specified using `dot` nodes in the LNAST since usually few things specify this attribute. The table is populated during the traversal discussed in Section 4.1 using the value of bits specified on each driver pin. The `dot` nodes used to specify other attributes are added immediately before that component's first use, if any of these nodes need to be added at all. The table and its usage will be discussed more in Chapter 5.

4.4 Usage

The commands found in Figure 4.3 can be used inside of LiveHD's shell to convert from LGraph to LNAST. The first shell command will use LiveHD's Verilog interface to translate from Verilog to LGraph, then LiveHD will translate that LGraph

```
1  // Convert from Verilog to LNAST
2  inou.yosys.tolg files:path/to/file.v |> pass.lnast_fromlg
3
4  // Convert an already optimized LGraph to LNAST
5  lgraph.open name:lgraph_name |> pass.lnast_fromlg
```

Figure 4.3: LiveHD shell commands to use LGraph to LNAST interface.

to LNAST. The second command will open an LGraph that already exists and then translate that LGraph to LNAST.

Chapter 5

Translating LNAST to FIRRTL

After having converted a FIRRTL design to LNAST, translating that LNAST to LGraph, performing design optimizations upon that LGraph, and finally generating a new LNAST from that improved LGraph, a designer may want to receive back the optimized design in FIRRTL form. To support this, an interface was made that translates an LNAST into the FIRRTL-Protobuf form discussed in Section 3.1. This output form was decided over a text format so that the output could be passed back to the FIRRTL to LNAST interface if one wanted to do that.

This section discusses the implementation details specific to the LNAST to FIRRTL interface and details the different phases of the translation process.

5.1 Resolving Circuit Components

The first phase of the translation process is dedicated to finding out what wires, registers, submodules, inputs, and outputs are used in the LNAST and defining

them in the FIRRTL design. LNAST does not support explicit declarations of these circuit components, unlike Verilog or FIRRTL which require a designer specify what a component is prior to its use. To identify all of these components, a pass over the LNAST must occur during which all of its nodes are visited. At any given time if the node being visited is used to represent the name of something in the circuit, then that name is checked to see if it matches certain patterns. As mentioned before, certain symbols indicate whether something is an input, output, or register. The way to know if something is a wire is if it does not begin with one of those symbols and the name does not begin with three underscores, as three underscores is used by the LiveHD compiler to represent an intermediate value in a circuit.

Each time one of these components are found, a Protobuf object is created to represent it. For inputs and outputs, a port is created and its name, type, and direction are all set. For registers and wires, a statement is added to the FIRRTL design to serve as their declaration. The type of the component and its name are also attached to this statement. The bitwidths of all of these things must be specified. This can be achieved through accesses to the bitwidth table discussed in Section 4.3. Using the name of the circuit component as the key into the table, the interface can then easily specify how many bits something should be.

Initially it is difficult to determine what type any of these components should be, so everything is set to the type of UInt with a bitwidth equivalent to what was specified in the bitwidth table. Since the claim that all circuit components will be UInts is not necessarily a correct one, a table is created that takes in the name of a circuit

component and returns a pointer to that component's type. This allows the translator to later alter what type something should be based off information that is seen during the next pass over the LNAST. For instance, it is possible that during the next pass a `dot` node is found which specifies that something should be signed. The translator can then look up the type of that component and alter it without having to make an entirely new statement or port. This will be used in the next phase when the translator refines types and will be referred to as the name-to-type map.

Submodules are handled slightly different than other circuit components. In LNAST, a submodule is represented as a `func_call` node. A node of this type can't be mapped directly to FIRRTL since this node will have a tuple for inputs and a tuple for outputs, whereas FIRRTL submodule instances use a single bundle through which all accesses to its IO must go through. Since only a single bundle can be used, the input and output tuples have to be combined. The simplest way to achieve this is by creating a submodule instance and using the name for that submodule instance as the single bundle name. The name for this submodule instance can be one of two things. If the names of the LNAST's two bundles are the same aside from one starting with `inp_` and the other starting with `out_`, then the name following these prefixes will be used as the submodule instance's name. Otherwise, the output tuple's name will be used. If the output tuple name is used, then all uses of the input tuple name must be renamed to the output tuple name so that they go through this single bundle.

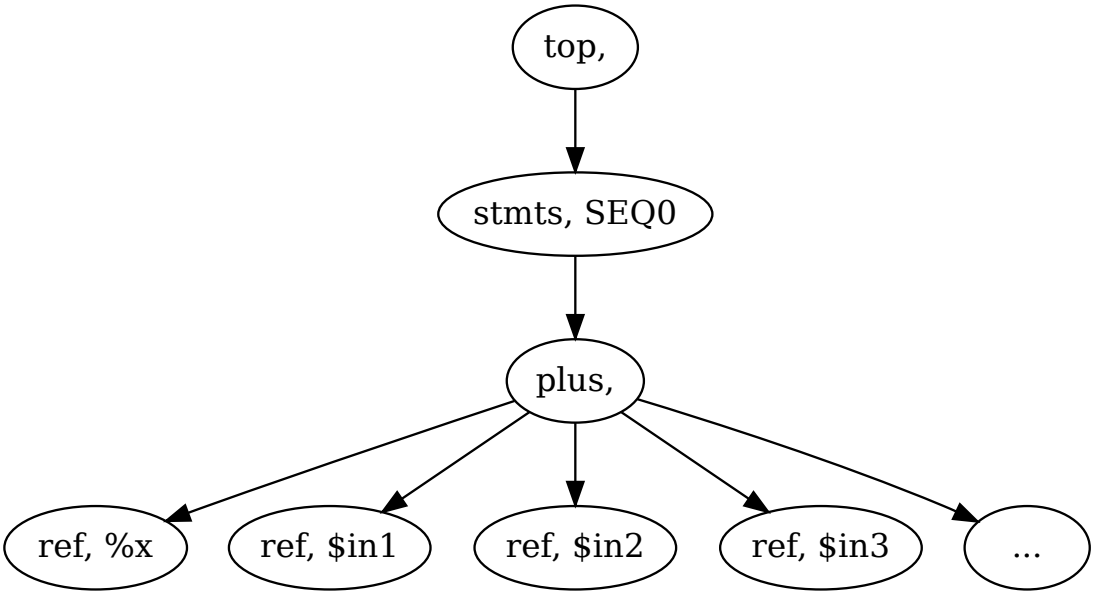

Figure 5.1: Example LNAST showing how n-ary operations can handle infinite number of children.

5.2 Statement Translation

The second, final phase of translation focuses on mapping LNAST statements to FIRRTL statements and refining the types created during the previous translation phase. Fortunately, the LNAST generated from LGraph uses only a subset of LNAST operations for which there exists a FIRRTL operation that does the same thing. FIRRTL's operations are usually stricter, however. For instance, a `plus` node in LNAST needs to have at least three nodes as children but in theory can have an infinite number. The `add` operation in FIRRTL only accepts two inputs. Having more inputs isn't much of a problem however since the translator can recursively add expressions inside of themselves, as can be seen in Figure 5.2.

```
1  x <= add(in1, add(in2, add(in3, ...)))
```

Figure 5.2: FIRRTL Translation from LNAST n-ary operation found in Figure 5.1.

Occasionally, when creating certain statements in FIRRTL, what is being created will require its arguments to be of a certain type. The translator uses these as guidance to help infer what the types of everything should be. For instance, if a signal is being used as a register's clock then that signal will be classified as a Clock type in FIRRTL. Using the name-to-type map discussed in the previous section, it is easy to update the type of a component whenever these hints are encountered.

The other way in which types can be updated is by encountering `dot` nodes in the LNAST. The `dot` node type is the only node type allowed to exist in the LNAST that is not supported by FIRRTL. These serve as hints as to what a component's type should be. An example of this is if a `dot` node is encountered and it is specifying the `__signed` attribute to be true, then it is inferred that the type of that component is SInt.

If it is ever encountered that a statement requires an argument to be of some type but that argument was previously inferred to be some other type and the two types are not interchangeable, then the translator will default to using FIRRTL type conversion statements. An example of this is the `asUInt()` operation.

5.3 Usage

The commands found in Figure 5.3 can be used inside of LiveHD's shell to convert from LGraph to LNAST to FIRRTL. The first command will generate an LGraph

```
1  // Convert from Verilog to FIRRTL
2  inou.yosys.tolg files:path/to/file.v
3        |> pass.lnast_fromlg |> inou.firrtl.tofirrtl
4
5  // Convert an already optimized LGraph to FIRRTL
6  lgraph.open name:lgraph_name |> pass.lnast_fromlg
7        |> inou.firrtl.tofirrtl
```

Figure 5.3: LiveHD shell commands to use LGraph to LNAST and LNAST to FIRRTL interfaces.

from Verilog, translate that LGraph to LNAST, then finally create a FIRRTL-Protobuf message. The second command will take a pre-existing LGraph then translate that to LNAST and then to a FIRRTL-Protobuf message.

Chapter 6

Benchmarks

To illustrate the speed in which LiveHD is able to transate from FIRRTL-Protobuf to Verilog, Figure 6.1 and Table 6.1 have been provided. In LiveHD, the entire process to obtain Verilog would be to translate from FIRRTL-Protobuf to LNAST then to LGraph and finally to Verilog. These tests show the time it takes for large FIRRTL designs filled with xors to complete that entire translation process. The size of the benchmarks ranged from five thousand xors in a row to five hundred thousand. This style of benchmarking was chosen because it allows for easy configurability and allows data points to be compared to see how the translation process scales when handling larger workloads.

The benchmark results for LiveHD are compared against the FIRRTL compiler since that compiler is currently the only alternative to LiveHD that can translate from FIRRTL-Protobuf to Verilog. These results can be obtained by building LiveHD with all optimizations active. To create a fair and accurate comparison, the time it takes for the

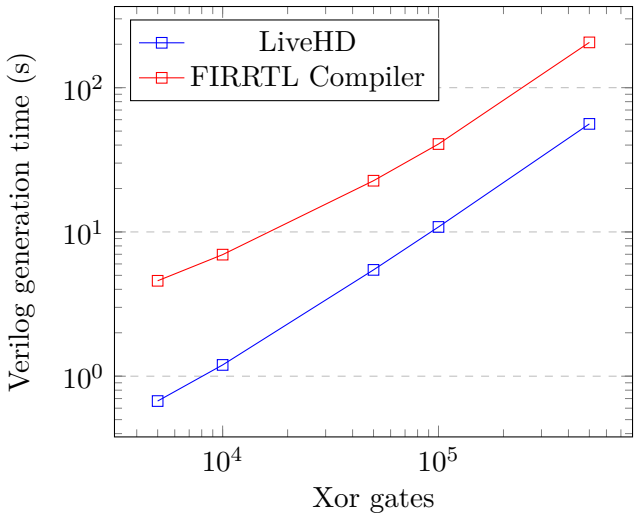

Figure 6.1: Visualization of the time it takes to go from a FIRRTL-Protobuf file to Verilog in the FIRRTL compiler versus the LiveHD compiler, using its new FIRRTL interface.

Time (s) required to convert xor-chain design from FIRRTL-Protobuf to Verilog		
Xor count (thousands)	In FIRRTL Compiler	In LiveHD
5	4.5847	0.6726
10	6.9597	1.1968
50	22.6463	5.4563
100	40.6753	10.8273
500	206.0043	56.091

Table 6.1: Data points used for Figure 6.1, comparing the time in seconds it takes for both the FIRRTL and LiveHD compiler to take in a FIRRTL-Protobuf file and to generate Verilog from it.

FIRRTL compiler to start up the Java virtual-machine (JVM) has not been counted in the results presented in this section. That portion of the run-time was removed since this work is more interested in comparing the translation time between different intermediate representations and HDLs. It is well-known that the JVM can have long startup times, thus it is not fair to include that as a part of the FIRRTL compiler's translation time.

In every case tested, LiveHD performed better than the FIRRTL compiler. Both showed an approximately linear growth with respect to the input size, with exception to some of the smaller designs when generating from the FIRRTL compiler. LiveHD's speed compared to the FIRRTL compiler ranges from approximately ten times faster, when handling smaller designs, to just shy of four times faster on larger input sizes. Such a speedup, especially when targetting large designs, is definitely a strong motivation for design groups to move towards using LiveHD.

In both cases, design optimizations were applied in a closely equal amount. For instance, both designs went through their respective bitwidth inference algorithm and constant propogation pass. Since LiveHD does not have a dead-code elimination pass at the time of these results, the dead-code eliminator in FIRRTL was turned off when benchmarking.

With much of LiveHD still in the development phase, there's a lot of room to grow with respect to performance. The step that dominates LiveHD's execution time is translating from LGraph to Verilog.

Chapter 7

Conclusion and Future Work

This paper presents several new techniques for translating between three different intermediate representations used for hardware designs: FIRRTL, LNAST, and LGraph. Notably, these translators show that it is possible for LiveHD to support an LNAST-FIRRTL interface that allows for a design to flow both ways. By extension, this work also shows that LiveHD is able to fulfill its promise of supporting interfaces with emerging HDLs coming from other research communities. Creating these interfaces also allows Chisel/FIRRTL designers to reap the benefits of what LiveHD is trying to achieve: live synthesis and simulation.

7.1 Future Work

Several areas that this paper touched upon would be worthy of future research. LNAST does not have a unified way to represent everything in the FIRRTL language. Because of this, the interface cannot support all FIRRTL designs yet. This is acceptable

right now as most of these features are either rarely used or experimental, but extending support to fixed point numbers, intervals, and analog wires would allow for a designer to unlock FIRRTL's full potential.

Another area for improvement can be found when going through the entire process of FIRRTL to LNAST to LGraph to LNAST back to FIRRTL. When translating from LNAST to LGraph, bundles and vectors get flattened. This means that when transforming that LGraph back to LNAST form, all of the bundle hierarchy will be gone except for submodule calls which had to be specifically accounted for. This becomes most apparent when generating FIRRTL from that LNAST. If the module had any bundles, which is incredibly common in FIRRTL, those bundles will be flattened. While not problematic by itself, this means that one cannot easily slot their generated FIRRTL design back into a larger design as the interface may have changed. Now, the designer can rewrite the interface but this can be potentially error-prone. One LiveHD team member is currently looking into ways to regain this hierarchy, specifically by storing hierarchy information when going from LNAST to LGraph. This would allow a hierarchical LNAST to be generated from an LGraph, while preserving much of the bundle and vector notation.

Finally, a random FIRRTL generator would allow for a more verbose test-suite for these interfaces. Exploring this would prove useful for both this work as well as anyone else wanting to design tools to support the Chisel/FIRRTL ecosystem. A suitable alternative that exists currently is testing using Rocket Chip [3] or BOOM [4], however it would be useful to have something paramaterizable that can help tool

designers target infrequent edge cases.

Bibliography

[1] Language Neutal Abstract Syntax Tree Documentation. `https://masc.soe.ucsc.edu/lnast-doc/`. Online; accessed on 10 April 2020.

[2] Protocol Buffers - Google's data interchange format. `https://github.com/protocolbuffers/protobuf`. Online; accessed on 10 April 2020.

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, and John Koenig. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Berkeley, CA, USA, Apr. 2016.

[4] Krste Asanović, David Patterson, and Christopher Celio. The Berkeley Out-of-Order Machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor. Technical Report UCB/EECS-2015-167, University of California, Berkeley, Berkeley, CA, USA, Jun. 2015.

[5] C.P.R. Baaij. *Digital circuit in CλaSH: functional specifications and type-directed*

synthesis. PhD thesis, University of Twente, Netherlands, 1 2015. eemcs-eprint-23939.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[7] Daniel Brand, Anthony Drumm, Sandip Kundu, and Prakash Narain. Incremental synthesis. In *Computer-aided Design, Proceedings of the 1994 IEEE/ACM International Conference on*, ICCAD'94, pages 14–18, Los Alamitos, CA, USA, Nov. 1994. IEEE Computer Society.

[8] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[9] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. A pythonic approach for rapid hardware prototyping and instrumentation. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–7. IEEE, 2017.

[10] Tsung-Wei Huang and Martin D. F. Wong. OpenTimer: A high-performance timing analysis tool. In *Computer-Aided Design, Proceedings of the IEEE/ACM Interna-*

tional Conference on, ICCAD'15, pages 895–902, Piscataway, NJ, USA, Nov. 2015. IEEE Press.

[11] Adam Izraelevitz. *Unlocking Design Reuse with Hardware Compiler Frameworks.* PhD thesis, EECS Department, University of California, Berkeley, Dec 2019.

[12] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim 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*, pages 209–216. IEEE Press, 2017.

[13] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. Specification for the firrtl language. Technical Report UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016.

[14] Derek Lockhart, Gary Zibrat, and Christopher Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–282. IEEE, 2014.

[15] Rishiyur Nikhil. Bluespec system verilog: efficient, correct RTL from high level specifications. *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, Jun. 2004.

[16] Rafael T. Possignolo, Nursultan Kabylkas, and Jose Renau. Anubis: A new benchmark for incremental synthesis. In *Logic Synthesis, Proceedings of the International Workshop on*, IWLS'17, Jun. 2017.

[17] Rafael T. Possignolo and Jose Renau. LiveSynth: towards an interactive synthesis flow. Poster at the 28th HotChips: A Symposium on High Performance Chips. Available at `https://users.soe.ucsc.edu/~rafaeltp/files/livesynth-hotchips2016.pdf`.

[18] Rafael T. Possignolo, Sheng H. Wang, Haven Skinner, and Jose Renau. LGraph: A multilanguage open-source database. In *Open-Source EDA Technology, Proceedings of the First Workshop on*, WOSET'18, Oct. 2018.

[19] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. Yosys+ nextpnr: an open source framework from verilog to bitstream for commercial fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–4. IEEE, 2019.

[20] Haven Skinner, Sheng-Hong Wang, Akash Sridhar, Rafael Trapani Possignolo, Kenneth Mayer, and Jose Renau. Pyrope. `https://masc.soe.ucsc.edu/pyrope.html`. Online; accessed on 16 December 2019.

[21] Jose I. Villar, Jorge Juan, Manual J. Bellido, Julian Viejo, David Guerrero, and

J. Decaluwe. Python as a hardware description language: A case study. In *2011 VII Southern Conference on Programmable Logic (SPL)*, pages 117–122, April 2011.

[22] Sheng-Hong Wang, Rafael Trapani Possignolo, Qian Chen, Rohan Ganpati, and Jose Renau. LGraph: a unified data model and api for productive open-source hardware design. In *Open-Source EDA Technology, Proceedings of the Second Workshop on*, WOSET'19, Nov. 2019.

[23] Sheng-Hong Wang, Akash Sridhar, and Jose Renau. LNAST: a language neutral intermediate representation for hardware description languages. In *Open-Source EDA Technology, Proceedings of the Second Workshop on*, WOSET'19, Nov. 2019.

[24] Clifford Wolf. Yosys Open SYnthesis Suite. `http://www.clifford.at/yosys/`. Online; accessed on 10 April 2020.