LG2PRP: A VERILOG TO PYROPE SOLUTION
A dissertation submitted in partial satisfaction of the requirements for the degree of
BACHELOR OF SCIENCE
in
COMPUTER ENGINEERING
by
Zachary Philip Potter
September 2018

The Dissertation of
Zachary Philip Potter is approved:

____________________________________
Professor Jose Renau, Chair

____________________________________
Professor Heiner Litz

____________________________________
Alexander Wolf
Dean of the Baskin School of Engineering
# Table of Contents

Abstract v

Acknowledgments vi

1 Introduction 1

2 Related Work 3
  2.1 Verilog .............................................. 3
  2.2 Pyrope ................................................. 4
    2.2.1 Pyrope Syntax .................................. 4
  2.3 Igraph ................................................. 7

3 Ig2prp Model 8
  3.1 Overall Flow ......................................... 8
  3.2 Handling Inputs and Outputs ............................. 12
  3.3 Handling Sequence of Statements ........................ 14
  3.4 Handling Control Flow ................................ 20

4 eprp Interface and Testing 27
  4.1 eprp Interface ......................................... 27
  4.2 Testing ................................................ 28

5 Conclusion and Future Work 29

Bibliography 31
# Listings

2.1 Inputs and Outputs Example .................................................. 4
2.2 Flip Flop Example .............................................................. 5
2.3 Function example ............................................................... 6
3.1 Main Pseudocode ............................................................... 9
3.2 ToOperation Pseudocode ................................................... 10
3.3 ToGraphIO Pseudocode .................................................... 11
3.4 ToSourceVar Pseudocode .................................................. 12
3.5 trivial.v ................................................................. 12
3.6 trivial.prp ............................................................... 13
3.7 add.v ................................................................. 14
3.8 add.prp ............................................................... 15
3.9 trivial1.v ............................................................ 16
3.10 trivial1.prp ............................................................ 16
3.11 operators.v ............................................................ 17
3.12 operators.prp ............................................................ 18
3.13 simpleFlop.v .............................................................. 19
3.14 simpleFlop.prp ............................................................. 19
3.15 latch.v ................................................................. 20
3.16 latch.prp .............................................................. 21
3.17 mux.v ................................................................. 22
3.18 mux.prp ............................................................... 23
3.19 submodule.v ............................................................ 24
3.20 submodule.prp ........................................................... 25
4.1 eprp Usage ................................................................. 28
Abstract

lg2prp: A Verilog to Pyrope Solution

by

Zachary Philip Potter

Verilog is currently the most prevalent hardware description language (HDL) and has been for some time. However, Verilog is becoming outdated, and has issues such as not being fully synthesizable, and having many hardware artifacts. Therefore, the UC Santa Cruz MASC lab has created Pyrope, a modern hardware description language. Once Pyrope development is complete, though, it will be necessary to find ways to increase the adoption rate of the language by the hardware design community.

In order to increase industry adoption of Pyrope, we propose lg2prp: a tool written in C++ that is able to convert Verilog code into Pyrope code. lg2prp is able to increase industry adoption of Pyrope by helping people learn the language. If someone is familiar with Verilog, then they will be able to convert their Verilog files into Pyrope files and see how to write Pyrope code. They will also notice that Pyrope is much more compact and efficient than Verilog, and thus will be inclined to use it for further circuit development. Large legacy programs written in Verilog can also be converted to Pyrope with lg2prp, so that it is easy for a company to switch from Verilog to Pyrope.
Acknowledgments

I would like to thank my thesis reading committee, Prof. Jose Renau and Prof. Heiner Litz, for taking the time to read my thesis and provide valuable feedback. In particular, I would like to thank Prof. Jose Renau for his continued support and guidance throughout this project. My labmate, Rafael, has also helped me countless times by being patient enough to answer all of my questions about lgraph and Pyrope. Finally, I want to thank my parents for all their support throughout my entire undergraduate career.
Chapter 1

Introduction

Pyrope is an HDL created by the UC Santa Cruz MASC lab that improves upon many other HDLs currently available [7]. However, as with any new language, the language must be widely adopted in order to be considered successful. Verilog is currently the most widely used HDL by the hardware development industry, so a Verilog to Pyrope converter tool could help improve the adoption rate of Pyrope by the hardware development community.

This work proposes the lg2prp tool that is able to convert Verilog code into Pyrope code. We describe the tool in more depth in following chapters, but in short, lg2prp is a tool that reads a Live Graph, or lgraph, that is generated from a Verilog file. The lgraph provides information about the base operations in the Verilog file, and with this information, lg2prp is able create an equal and valid Pyrope file. The main contributions of this work include:

- **The proposition of lg2prp** in order to increase industry adoption of Pyrope. We provide the model for lg2prp and describe the flow for converting Verilog files to Pyrope files with lg2prp.
• **Examples of lg2prp** run on multiple Verilog files in order to demonstrate the correctness of the tool.

• **A description of how to use lg2prp** with the eprp interface. We then describe our methodology for testing the tool with the eprp interface.

We first describe some pertinent related work in Chapter 2. Then, we provide the model for the proposed lg2prp tool in Chapter 3. Details on the user interface and our testing methods are provided in Chapter 4. Finally, we explain our conclusions and provide ideas for future work in Chapter 5.
Chapter 2

Related Work

This chapter will first give a short overview of Verilog and a couple of its drawbacks. Then, we will introduce Pyrope and describe its syntax. Finally, we will go over lgraph, which is used by lg2prp to help convert Verilog code into Pyrope code.

2.1 Verilog

Verilog is a commonly used hardware description language (HDL) that is utilized to design digital circuits. Verilog has been widely used since it’s inception in 1984 [4][11]. However, Verilog is becoming outdated and difficult to read compared to higher level programming languages (like Python and Go) that are becoming more popular in recent times. Verilog’s drawbacks include not being fully synthesizable, and containing hardware artifacts. Verilog is not fully synthesizable because it has constructs, such as delays, which cannot be synthesized on an FPGA [10]. Verilog’s nonblocking assignment is an example of a hardware artifact, because it exposes the hardware constructs underneath [8].
2.2 Pyrope

Pyrope is a modern HDL developed in the UCSC MASC lab which is easier to learn and has many improvements compared to Verilog and other popular HDLs like Chisel [2], PyMTL [5], or CLasH [1]. Some notable improvements are that Pyrope is a fully synthesizable language, and it does not have hardware artifacts. Since it is fully synthesizable, everything written in Pyrope can be synthesized on an FPGA. Because Verilog is currently the industry standard in HDLs, there are many programs which are still written in Verilog being used today. Therefore, it is necessary to have a Verilog to Pyrope translator, which compiles Verilog code into equal and valid Pyrope code. Such a tool could also help people learn Pyrope if they are already familiar with Verilog.

It is important to understand the syntax of Pyrope in order to understand how lg2prp works. This next section will delve into some of the relevant Pyrope syntax.

2.2.1 Pyrope Syntax

```plaintext
1 $a as __bits:1
2 $b as __bits:1
3 %c = $a xor $b
```

Listing 2.1: Inputs and Outputs Example

In Listing 2.1 we see that there are three variables: a, b, and c. When the $ symbol is attached to a variable, it denotes that the variable is an input, while the % symbol denotes that a variable is an output. Thus, we can see that a and b are inputs, while c is an output.
We also see some compiler directives in the above example. Two underscores denote that you are giving the compiler a directive. Thus, $a as _bits:1 means that the variable $a is one bit. It is unnecessary to declare how many bits $c is because the compiler is able to infer how many bits it should be based on how many bits $a and $b have.

```
1 if $enable {
2   @total := @total + 1
3 }
```

Listing 2.2: Flip Flop Example

Listing 2.2 shows a counter in Pyrope. If the enable input is high, then the total will be incremented. We also see the use of the @ symbol, which denotes that a variable is a flip flop or latch in Pyrope.
Listing 2.3: Function example

Listing 2.3 demonstrates a function call, which would be similar to a module in Verilog. Lines 1-6 show the definition of the function named `func`. In line 10, we see the function called, and the instance of that function is assigned to the variable `foo`. In line 10, we also see that `a` and `b` are used as inputs to the function and are assigned to variables `e` and `f`, respectively. Lines 11 and 12 assign the outputs of the function, `g` and `h`, to variables `c` and `d`, respectively. Line 14 demonstrates a comment, which is started with the `#` symbol. Line 14 shows an alternate, more compact, method of doing what lines 10-12 do, without creating the instance variable `foo` [7].
Most other Pyrope syntax works similarly to how you might expect it to work in C-like languages or Verilog. For example, comparison operators, arithmetic operators, and conditional statements will work how one might expect if they are familiar with programming.

2.3 lgraph

In order to translate Verilog into Pyrope, an intermediary is needed. The one that we use is lgraph, which is a graph representation of a digital circuit that was also developed by the MASC lab [12]. The lgraph is represented by a collection of nodes, where each node has an associated ID number, operation, and inputs and outputs of that operation. When the lgraph is created, unique ID numbers are assigned to each node in order to identify them. The operation assigned to the node is any operation that could be represented in a HDL (such as arithmetic operations, comparison operations, mux operations, latch operations, flip flop operations, or function operations, just to name a few). Each node also contains the IDs of the inputs and outputs of that operation; for example, if the operation was division, then the inputs would be the dividend and divisor, and the output would be the quotient. lgyosys is the tool that we use to convert Verilog code into lgraphs [14].

The lgraph can be traversed either forward or backwards, but lg2prp only uses the forward traversal since it is the more efficient operation.
Chapter 3

lg2prp Model

This chapter provides the structure of lg2prp and describes the inner workings of the tool. We first describe the overall flow of lg2prp reading an lgraph. Then, some of lg2prp’s important functions are explained. Finally, several examples are provided of Verilog files converted to Pyrope by lg2prp.

3.1 Overall Flow

In order to use lg2prp, it is necessary to first take the Verilog code, and use lgyosys in order to transform it into an lgraph. Once the Verilog code is represented as an lgraph, one can then use lg2prp in order to read the lgraph and translate it into valid Pyrope code. The naming of lg2prp comes from shortening “lgraph to Pyrope”. The lgraph is made up of many nodes, and each node represents an operation. Thus, lg2prp works by traversing the graph, reading the operation of each node, getting any inputs and outputs of that operation, and printing the corresponding operation and source variables in Pyrope syntax to the .prp output file.
In Listing 3.1 we see some pseudocode for the main functionality of lg2prp. This first block represents the main function. This first function reads the lgraph, iterating over each node. For each node, it reads the operation, and calls the to_op function on it. If the operation requires a destination variable (such as an addition or logical operation), then it prints the destination variable. Finally, it prints each output string to the .prp file.

```c
void to_pyrope(lgraph):
    open the lgraph
    for each node
        op = node.get_operation()
        dest = to_op(op, output string)
        if dest
            to_dest_var(node, output string)
    print subgraph output string
    print main output string
```

Listing 3.1: Main Pseudocode

Listing 3.2 shows the outline for the generalized "to_op" function. In the lg2prp source code, we have functions to print each type of operation that we encounter. However, since each function has a similar skeleton, we describe it here. The to_op function first iterates over all input edges. For the first input, it simply prints the source variable. For the second input, it prints the operation and then the second source variable. Thus, the output of this would be \( c = a + b \), if the operation was addition. The source variables would be \( a \) and \( b \), and the destination variable would be \( c \). It would be similar for any other logical or arithmetic
operation, but for something like a subgraph, latch, or flip flop, there would be some extra steps in order to ensure that the output was formatted correctly.

```python
1 bool to_op(op, output string):
2    bool first = true
3    for all node input edges
4        if first
5            first = false
6        else
7            print operation # (i.e. +, -, xor, etc.)
8            print source variable to output string
9
10 return true

Listing 3.2: ToOperation Pseudocode
```

Listing 3.3 shows the pseudocode for graph IO operations. A graph IO operation is how the lgraph defines whether a variable is an input or output, as well as the number of bits it has. The simple way to print the graph IO operation is check whether it is an input or output to print the correct symbol. If it is an output that has no inputs, then we can safely not print anything since it is an unused variable. Finally, we can print the variable name, compiler directive, and number of bits to the output string.
bool to_graphio(node, output string):

    if (node is input) {
        print '$'
    } else { # Node is output
        if (!node.has_inputs())
            return false # output is not driven/used
        print '%'
    }
    print variable name
    print " as __bits:
    print node.get_bits()
return false

Listing 3.3: ToGraphIO Pseudocode

Listing 3.4 shows the pseudocode for the to_src_var() function. This is called in most to_op type functions whenever a source variable needs to be printed. It first checks the simple cases, whether the variable is an input or output, and prints the necessary symbol followed by the variable name. Then, it checks if it is a constant, or numerical value. Finally, if it doesn’t fall into any of these categories, we can assume that the variable is a tmp variable and print ”tmp” followed by the node’s ID number. A variable is a tmp variable if yosys creates extra nodes in the graph by expanding complex operations in the Verilog source.

The to_dest_var() is similar to to_src_var(), except it includes some slightly different checks.
void to_src_var(node, output string):

if (node is input) {
    print '$_$
    print variable name
}
else if (node is output) {
    print '%'
    print variable name
}
else if (node is constant) {
    print constant value
}
else {
    print "tmp"
    print node.get_node_id()
}

Listing 3.4: ToSourceVar Pseudocode

In this next section, we see some examples of Verilog code that has been transformed into Pyrope code by lg2prp.

3.2 Handling Inputs and Outputs

module trivial( input a, input b, output c);
assign c = a ^ b;
endmodule

Listing 3.5: trivial.v
```
# trivial.prp file from lgraph_trivial

$b as __bits:1
$a as __bits:1
%c = $a xor $b
%c as __bits:1
```

**Listing 3.6: trivial.prp**

This first example demonstrates lg2prp working on a trivial graph. Listing 3.6 shows that lg2prp is able to assign the bit values to the inputs and outputs. It also is able to successfully convert the xor operation in Verilog to the correct Pyrope equivalent. lg2prp is able to make these conversions by reading the lgraph, which contains information such as how many bits each variable is, and the inputs and outputs of the xor operation.
3.3 Handling Sequence of Statements

```verilog
module add(input [7:0] a, input [7:0] b,
           output [7:0] c,
           output [7:0] d,
           output [7:0] e,
           output [7:0] e1,
           output signed [7:0] f,
           output signed [7:0] g,
           output signed [7:0] h,
           output signed [7:0] hi);

assign c = a + b;
assign d = a - b;
assign e = a + b - a;

signed wire [7:0] as = a;
signed wire [7:0] bs = b;

assign f = as + bs;
assign g = as - bs;
assign h = as + bs - as;
endmodule
```

Listing 3.7: add.v
# add.prp file from lgraph_add

$b as __bits:8
$\text{a as __bits:8}
$c = $\text{a + b}
$c as __bits:8
\text{tmp12 = a + b}
$e = \text{tmp12 - a}
$e as __bits:8
$f = a + b
$f as __bits:8
\text{tmp14 = a + b}
$\text{h = tmp14 - a}
$\text{h as __bits:8}
$\text{d = a - b}
$\text{d as __bits:8}
$\text{g = a - b}
$\text{g as __bits:8}
module trivial1(
  output reg y,
  input a, b, c,
);

always @(*) begin
  y = (((a & b) & c) | ((a & b) & ~c));
end
endmodule

Listing 3.9: trivial1.v

# trivial1.prp file from lgraph_trivial1
$c as __bits:1
tmp9 = not $c
$b as __bits:1
$a as __bits:1
tmp5 = $a and $b
tmp6 = tmp5 and $c
tmp7 = $a and $b
tmp8 = tmp7 and tmp9
%y = tmp6 or tmp8
%y as __bits:1

Listing 3.10: trivial1.prp

The add.v and trivial1.v examples in Listings 3.8 and 3.10 show lg2prp converting a variety of different operations. One will notice that when there is more than one operation...
occurring on the right side of an assignment operation in the .v file, the operation is decomposed into multiple lines in the .prp file. This is because the lgraph only supports one operation per node. Since we can only read one node at a time, we use tmp variables in order to have each operation on a separate line. The number of the tmp variable is taken from the node ID of the operation in the lgraph.

```
module operators (input [7:0] a, input [7:0] b,
output c, output d, output e, output f, output g, output h);

assign c = a == b;
assign d = !(a == b);
assign e = a > b;
assign f = a >= b;
assign g = a < b;
assign h = a <= b;
endmodule
```

Listing 3.11: operators.v
Listing 3.12: operators.prp

This example shows many different types of comparison operators. Listing 3.12 demonstrates that lg2prp is able to successfully convert each type of comparison operator.
module simple_flop(input c, input d, output reg q);

always @(posedge c)
  q <= d;

endmodule

Listing 3.13: simpleFlop.v

# simple_flop.prp file from lgraph_simple_flop
$d as __bits:1
$c as __bits:1
@q = $d

Listing 3.14: simpleFlop.prp

This example in Listing 3.14 shows how a flip flop is converted to Pyrope by lg2prp.

In Pyrope, the flip flop is always updated after each clock cycle, so it is unnecessary to include the always @ statement like Verilog.
3.4 Handling Control Flow

```verilog
module latch(input d, input c, output q);

always_latch begin
  if(c == 1) begin
    q <= d;
  end
end
endmodule
```

Listing 3.15: latch.v
Listing 3.16: latch.prp

Listing 3.16 shows how a latch is converted to Pyrope by lg2prp. The directive as __fflop:false means that q will be interpreted as a latch, rather than a flip flop. One will notice that there are many variables created that are not pertinent to the latch operation. This is because there are some rare instances where yosys is unable to completely optimize the lgraph and will generate nodes that are unnecessary. I would like to improve detection of this in the future in order to eliminate the extra lines of code.
modulemux(inputa, inputb, inputsel, outputf);

regf;

always @(sel or a or b)
beginMUX
    if (sel == 1'b0) begin
        f = a;
    end else begin
        f = b;
    end
end
endmodule
Listing 3.18: mux.prp

Listing 3.18 shows how a mux is translated to Pyrope by lg2prp. The inputs to the mux operation are the case and the variables being assigned to the output of the mux. In other words, tmp5, a, and b are inputs to the mux operation, while f is the output of the mux operation in Listing 3.18. The inputs are differentiated by their port IDs on the mux operation’s node. Therefore, the to_mux operation in the lg2prp source code iterates over each input and uses the port ID of each input to determine which variable should be used as the case, which should be used as the output in the true case, and which should be used as the output in the false case.
module inner(input e, input f, output g, output h);
  assign g = e & f;
  assign h = !(e & f);
endmodule

module submodule (input a, input b, output c, output d);
  inner foo(.e(a),.f(b),.g(c),.h(d));
endmodule

Listing 3.19: submodule.v
inner = :( $e $f %g %h):

$f as __bits:1
$e as __bits:1
%g = $e and $f
%g as __bits:1
tmp6 = $e and $f
%h = not tmp6
%h as __bits:1
}

# submodule.prp file from lgraph_submodule
$b as __bits:1
$a as __bits:1
foo = inner e:$a f:$b
%c = foo.g
%d = foo.h
%d as __bits:1
%c as __bits:1

Listing 3.20: submodule.prp

The submodule operation was interesting to consider because each module in a Ver-ilog file creates another lgraph. Thus, when lg2prp comes to a subgraph node, it must open another lgraph and start reading that one. The output of this new lgraph is saved into a new string which is printed before the main module output string is printed to the .prp file. To connect the inputs and outputs in the calling module to the submodule, the port IDs for the
submodule’s inputs and outputs must be saved while reading the open subgraph. Then, after we close the subgraph, the saved port IDs are compared to the port IDs of the calling module; if port IDs match, that means that the inputs and outputs can be connected. For example, \( c \) and \( g \) would have matching port IDs, which is why they are connected in line 15 of Listing 3.20.
Chapter 4

eprp Interface and Testing

This chapter describes usage of our interface, eprp, which was created by MASC in order to easily use our tools that create, read, or manipulate lgraphs [6]. This interface was created because we had so many separate binary files that it was becoming cumbersome to use the various tools. We had tried using open source interfaces, but none had fit our needs well. Thus, we created our own interface. After describing the interface, we will go over the testing method that we used.

4.1 eprp Interface

First, one must clone the github repository masc-ucsc/lgraph.git. Then, use the `bazel build //...` command in order to build the project. Finally, we can open up the eprp console by running the command `./bazel-bin/main/lgraph`

Once the console is open, run the command

```
inou.yosys2lg name:file_to_convert.v | @a
```

in order to generate an lgraph and
store it in the variable @a. Then, run @a |> inou.pyrope.fromlg in order to generate Pyrope code from that lgraph. Finally, exit the console and you will see the generated Pyrope code output.prp in your current directory.

Therefore, if someone was starting from scratch, the overall flow for using the interface would be:

```
1  >git clone git@github.com:masc-ucsc/lgraph.git
2  >cd lgraph
3  >bazel build //...
4  >./bazel-bin/main/lgraph
5  >inou.yosys2lg name:file\_to\_convert.v |> @a
6  >&a |> inou.pyrope.fromlg
7  >exit
8  >cat output.prp
```

Listing 4.1: eprp Usage

### 4.2 Testing

To test the tool, we would use the above flow to convert a variety of different test Verilog files. Then, the output of lg2prp was visually confirmed to be logically equivalent to the Verilog code. Next, the generated .prp file was compiled by the Pyrope compiler to ensure that it was correct Pyrope code. This testing method was suboptimal since it was not automated, but we will be improving it in the future.
Chapter 5

Conclusion and Future Work

The examples demonstrate that lg2prp is able to convert the operations in Verilog to Pyrope. These operations include all the examples shown in Chapter 3 as well as others that were not shown such as wire splitting and joining operations and logical shift operations. This tool can help boost industry adoption of Pyrope and allow for the conversion of legacy Verilog files into Pyrope files so that they will be easier to maintain.

As for future work, the first improvement to make would be removing the tmp variables in the generated Pyrope code. This would make the output Pyrope more compact and improve the readability of the code if each operation was printed on a single line like it is in the original Verilog code. This would also include detecting extra or unnecessary nodes when lg2prp is given an unoptimized lgraph. Another improvement would be more automated testing. One way to do this would be to create a Pyrope to Verilog converter. Then, one could convert Verilog code into Pyrope, and then convert the generated Pyrope back into Verilog. If the original Verilog code matched the final file, it could be confirmed to be correct. Also, there are many
analysis and debugging tools available for Verilog, such as Verilator [3], Icarus Verilog [13], and VCS [9]. Therefore, it would be useful to be able to convert your Pyrope file into Verilog, use tools to analyze and optimize your Verilog code, and then convert the file back to Pyrope for further editing.

The work presented in this thesis offers a reliable way to convert Verilog code into Pyrope code. The tool is very useful on its own, but there are still many interesting opportunities for future work and improvement upon this tool.
Bibliography


