UNIVERSITY OF CALIFORNIA
SANTA CRUZ

DESIGN AND IMPLEMENTATION OF THE PYROPE HARDWARE LANGUAGE

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

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Jing-Hsiang Huang

December 2022

The thesis of
Jing-Hsiang Huang is approved:

Professor Jose Renau, Chair

Professor Tyler Sorensen

Professor Heiner Litz

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Design and Implementation of the Pyrope Hardware Language

by

Jing-Hsiang Huang

This thesis presents the design and implementation of a hardware language, Pyrope. It first describes Pyrope's distinct language features with their design rationales and compares Pyrope with the mainstream language, SystemVerilog. It then shows the implementation of the Pyrope compilation flow that translates the language specification into low-level intermediate representation in a compiler framework, LiveHD. Several improvements to LiveHD as part of this work are also presented. Finally, the thesis shows the result of compiling an example design written in Pyrope.

Acknowledgments

Thanks to Professor Jose Renau for guiding and collaborating on the project and Sheng-Hong Wang for building the LiveHD compiler framework.

Chapter 1

Introduction

Hardware description languages (HDLs) are the primary tool for implementing digital hardware designs. Hardware engineers implement circuit logics and structures using HDLs, where the compiler translates and synthesizes source code into physical structures, such as logic gates and wires. The design of HDLs, as the main interface connecting human designers and products, is critical to the overall productivity of hardware development. Improving the efficiency of HDLs is highly desirable in the face of diminishing returns from transistor scaling [21] and rising costs in leading-edge hardware bring-ups [17].

Traditional HDLs, such as SystemVerilog [10] and VHDL [11], have evolved through a series of revisions, getting packed with more expressive language constructs and higher-level abstractions. Many programming paradigms that have long been a part of software programming, including object-oriented programming and meta-programming, have been introduced to the hardware realm in the hope that hardware

designers will write cleaner and more efficient code. While the new set of features does improve the quality and compactness of code, it comes with the cost of a heavily loaded language standard. Building parsers that can read the complete language space is already very challenging, not to say simulating and synthesizing the code correctly per the standard. Even so, traditional HDLs sometimes still lack certain features needed in some use cases, but these languages do not support any user language extensions.

New HDLs and compilation frameworks have sought to tackle the shortcomings of traditional HDLs. Many of them are fully open-source and welcome contributions from the community, lowering the barriers for end users to develop productive language features and tools. One of the many examples is the Live Hardware Development (LiveHD) compiler infrastructure [23] and the Pyrope HDL developed by the Micro Architecture Santa Cruz (MASC) group. LiveHD is a compiler framework that compiles multiple HDLs, such as Verilog and Pyrope. Similar to LLVM [14], LiveHD comes a compiler pass framework where users can construct reusable parsing, transformation, analysis, and code generation passes. A pass may interface with two types of intermediate representation (IR) models, which are LNAST, a tree IR for behavioral modeling, and LGraph, a graph IR for netlist-level modeling. Pyrope has been developed as an experimental language for testing LiveHD's full capabilities. Pyrope also targets beginners in hardware design by providing software-programming-like syntax and high-level abstrations of hardware structures. The contributions of this work are listed below:

- Modernize the Pyrope language by adding more language features, such as variable typing, attribute fields, pipestaging, and more expression types.

- Formalize Pyrope's grammar.
- Support Pyrope compilation in LiveHD.
- Improve LiveHD's debug and testing infrastructure.

The rest of the thesis is organized as follows. Chapter 2 discusses background knowledge and related work that will help understand concepts, ideas, and considerations behind hardware language design. Chapter 3 presents the structure of the Pyrope language and compares Pyrope with other languages. Chapter 4 shows the implementation of the Pyrope compilation flow with several infrastructure improvements in LiveHD. Chapter 5 shows examples of Pyrope designs supported by the current compiler implementation. Finally, Chapter 6 summarizes the contributions of this work and discusses future research opportunities.

Chapter 2

Background and Related Work

This chapter is broken into two parts. The first part discusses the functions of HDLs, and the second part covers several existing HDLs and compares their features and flaws.

2.1 Designing Hardware through HDLs

Modern integrated circuit (IC) chips are made from a large number of transistors, up to the scale of a trillion of them in a single package [1]. It is impossible to handcraft the schematic connecting hundreds of billions of transistors while expecting it to perform all the desired functions correctly. Throughout the history of semiconductor engineering, several abstraction levels have been introduced to reduce human efforts involved in building the mega-chips we have seen today. The hierarchy of abstraction levels starts with how humans can describe the function of a circuit and ends with how tools can map the function to a physical structure that performs the function. Design-

ers specify circuit functions with the top abstraction language, and then automation tools transform the specification into lower abstraction levels. Figure 2.1 illustrates the hierarchy of common abstraction levels in hardware development.

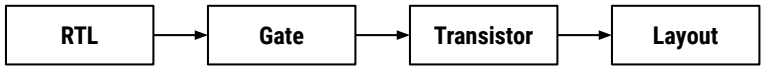

Figure 2.1: Different hardware abstraction levels.

There are two categories of mainstream top-level abstraction languages in the current industry, HDLs and high-level synthesis (HLS) languages [16]. HDLs model synchronous digital circuit behaviors at the register transfer level (RTL), which can be seen as fully-parallel lock-step programs. HLS languages are derived from software languages that are traditionally used to program von Neumann machines with one processing unit and one memory unit and must pair with a special compiler that maps a program using unbounded resources into a circuit with limited resources. While HLS languages require less input specification, the generated circuits are less predictable and often suboptimal to the manually written HDL counterpart. As a result, HLS languages are mostly used in industries that value a shorter product development cycle or less development cost over the optimality of the product. The majority of highly optimized products, such as CPUs and GPUs, are still designed with HDLs primarily.

2.2 Existing Languages

The requirement for a language to be considered an HDL is that any accepted code in that language must represent a circuit except for the non-synthesizable portion of the code. An HDL is often paired with a simulator that simulates functions of the represented circuits and a synthesizer that maps the described logic into actual circuits and physical layouts. The concept of HDL started with the design of the Verilog HDL or Verilog in short. Verilog was initially designed to be a language for constructing circuit simulators, where later a set of automation tools that can turn the simulated designs into physical implementations was developed to facilitate the physical design process. Derived from Verilog, SystemVerilog is widely adopted in the current semiconductor industry as the primary hardware design language. [1] Almost all new hardware products involve some use of SystemVerilog (or the subset language, Verilog) in their development process. As the industry has become very profitable and competitive, a large investment has been poured into constructing a complete and robust ecosystem around the language, including simulators, verification tools, and physical design compilers. Besides the development of tools, the demands for better readability and reusability have also driven multiple revisions of the language itself throughout the years. The current SystemVerilog language has adopted object-oriented programming and meta-programming concepts that have been a key part of software engineering for

[1] While VHDL has a notable share of users, its popularity has been declining throughout the last 20 years. See [12].

decades. While SystemVerilog also features many non-synthesizable constructs that are only used in simulations for verification, the following discussion will focus only on the synthesizable part of the language.

Since the SystemVerilog standard does not define any native language extensions, nor do most commercial tool implementations, end users often have to find workaround solutions to implement custom features, such as automated wiring, syntactic sugars for common structures, and annotating physical design attributes. One simpler solution is to utilize the code preprocessor, either to define a macro library or a comment template for pragmas and directives. Figure 2.2 shows a simple example use of define macros. Some notable macro libraries are the Open Verification Library (OVL) [4], the Universal Verification Methodology (UVM) [9], and attribute pragmas reserved in many synthesis tools [13]. A popular productivity script, Verilog-Mode [18], also defines a set of pragmas and uses the preprocessor and a lightweight parser to automate wire connections and variable declarations.

Due to the high complexity of the language specification, the initial investment in building complex tools that directly interface traditional HDLs overshadows the potential returns, discouraging the end users from experimenting with productivity-improving hacks. Even if the users do decide to implement custom tools with the aforementioned methods, these tools often have to use the language itself as the interface to other tools, as opposed to a much more efficient, universal intermediate representation (IR) format in most software compiler toolchains.

```
// Definition
`define INCREMENT_BY_ONE(IN, OUT, SIZE) \
  for (genvar i = 0; i < SIZE; i++) begin \
    assign OUT[i] = IN[i] + 1; \
  end
```

```
`INCREMENT_BY_ONE(in_array, out_array, 4)
```

Figure 2.2: SystemVerilog define macro.

When the preprocessing method is no longer sufficient for the desired features, users may want to implement code generators that automate the process of transforming higher-level or more compact specifications or parameters into cumbersome low-level HDL code. In [22], these software languages that generate HDL code from software runtime are described as meta-programmed host languages. A meta-programmed host language is usually a software language that supports dynamic data structures and sometimes dynamic typing, such as Python and Perl. Instead of specifying just one or a narrow subset of parameterized modules allowed in native HDLs, the user writes an automation script that takes parameters as inputs and outputs a valid HDL code. To reduce duplicate code across multiple automation scripts, the scripts can import utility libraries that encapsulate common HDL data structures and code generation functions. One example is a Python library called Magma [3] where circuit structures are represented by Python class instances.

```
def make_HandshakeData(data_type):
    in_type = m.Product.from_fields("HandshakeData", {
        "data": m.In(data_type),
        "valid": m.In(m.Bit),
        "ready": m.Out(m.Bit)
    })
    out_type = m.Flip(in_type)
    return in_type, out_type
```

Figure 2.3: An example code snippet from Magma's repository.

Beyond common functions and data structures, some libraries overwrite the built-in semantics of the host language with ones that suit the hardware behavior and map language constructs in traditional HDLs into simpler forms in the host language. This improvement makes an automation script itself structured just like an HDL without any code-generation artifacts. These languages are called embedded domain-specific languages (DSLs), including PyMTL [5], CHISEL [2], and PyRTL [6]. Figure 2.2 shows an example design written in CHISEL. Even more, as the source code can be translated into a data structure, some of these languages come with a set of standard compiler passes that operate on the internal data structures, such as constant folding and bitwidth checking, and even their own simulator and logic synthesis tools, which effectively cuts dependencies on tools that only interfaces with traditional HDLs. Furthermore, an important consensus made by developers of these languages is to adopt an open-source intermediate representation in the compiler infrastructures. The interoperability makes compiler passes much more reusable without the long yet error-prone HDL code generation and parsing. For example, CHISEL is paired with a compiler framework that

uses an IR called FIRRTL and a set of compiler passes that all interface with FIRRTL.

```
package fifo

import chisel3._
import chisel3.util._

/**
  * FIFO IO with enqueue and dequeue ports using the ready/valid interface.
  */
class FifoIO[T <: Data](private val gen: T) extends Bundle {
  val enq = Flipped(new DecoupledIO(gen))
  val deq = new DecoupledIO(gen)
}

/**
  * Base class for all FIFOs.
  */
abstract class Fifo[T <: Data](gen: T, depth: Int) extends Module {
  val io = IO(new FifoIO(gen))

  assert(depth > 0, "Number of buffer elements needs to be larger than 0")
}
```

Figure 2.4: An example code snippet from a CHISEL repository.

Though embedded DSLs like CHISEL have solved many issues in traditional HDLs, they still have several downsides that stop the industry from migrating to these languages. First, building HDLs as embedded DSLs on top of a host language requires an extra translation layer written in the host language, but usually, the host language is a scripting language that is processed by a less performant just-in-time (JIT) compiler, making the overall compile time longer. This is more critical in an industry setting as a reasonably sized system-on-chip design often consists of hundreds of thousands of lines of code, and compilation of the whole design often takes over an hour. Another

disadvantage of embedded DSLs is that the language itself must be a subset of the host language, and consequently, language designers often have to sacrifice the readability and the compactness of the language. The dependency on the host language also makes the HDL itself vulnerable to bugs in the host language compiler. A native HDL is desired as a result.

Pyrope is an example of a native HDL that does not rely on a host language. It is paired with the LiveHD compiler infrastructure that has its own IRs and compiler passes and also provides interfaces with FIRRTL and Verilog. Pyrope is now in version 0.3, and this work contributes to the specification of the revised language. The next chapter presents the Pyrope language with a focus on its unique features.

Chapter 3

Pyrope

This chapter presents the structures of the Pyrope language and compares Pyrope with SystemVerilog. The complete and most updated specification of Pyrope can be found on the website. The contribution of this work includes new language features listed below.

- Function definition statement
- Match expression
- Pipestage
- Type
- Attribute

3.1 Overview

This section gives a brief overview of the Pyrope language.

A design can be described with one or more Pyrope source files with `.prp` extension. Each file contains a description and forms a library scope, which can be referenced by another description in another file through library import. A hierarchical design can be constructed by specifying a source file as the top-level instance and elaborating dependent source files.

Comments can be added to the source code file for documentation and annotation purposes. Single-line comments start with `//`, where any character after which and before the first newline character will be ignored. The language does not support multi-line comment blocks enclosed to make sure that code can be split at newlines without peeking at neighboring lines. This limitation makes parallel parsing easier.

```
// This is a single-line comment.
```

A description consists of a sequence of statements, which are separated by one or more newline characters or semicolons. A long statement can be broken down into multiple lines for better readability with the limitation that the second line and after must start with a punctuation mark. This limitation ensures a consistent style for lists and long expressions and again will benefit parallel parsing.

```
a = 1                            // newline separator
b = a_very_long_variable_name
  + another_long_variable_name   // multi-line expression
c = (first_item_in_the_tuple
    ,second_item_in_the_tuple
    ,last_item_in_the_tuple)     // multi-line tuple list
d = 4; let e = 5                 // semicolon separator
```

Statements describe circuit structures, logic functions, or properties of the design. The most common statement type is the assignment statement, which binds a value to a variable. The value can be either a constant, the value of another variable, or the result of an expression. Common statements like conditional and loop statements are also defined in Pyrope.

```
a = 0
b = a + 2
if (b > 5) { c = 5 }
for i in 0..<4 { d[i] = c }
```

Variables are either mutable or immutable specified by a `var` or a `let` qualifier respectively. In the RTL context, immutable does not mean the value is constant, the true meaning is that the variable is only assigned once in its scope, and the assigned value may still change over time.

```
var a = 0
let b = a
a = 1
let c = a + 1
```

All variables in Pyrope are typed, and to be specific, strongly typed. This is a requirement for hardware languages as circuits must be static. Types can be implicitly inferred or explicitly specified. There are two categories of types, which are primitive

types, including sized or unsized integer, string, boolean, and composite types that are built from primitive types. One special feature of Pyrope is that type itself can be a kind of primitive type, while nesting it in composite types is prohibited, making it equivalent to type definition or aliasing in other languages.

```
a:u8     = 1 + 2     // type specification at declaration
b        = a:u9      // type check in an expression
T:type   = (:s8,:s8) // type aliasing or type declaration
```

A module can be specified either as a `fun` if the logic is purely combinational or as a `proc` if the logic contains sequential components. Sequential components in a `proc` are triggered by an implicit clock signal that can be specified through an attribute field or inherited from the parent scope at instantiation.

```
let add = fun(a, b)->(c) { c = a + b }
let flop = proc(d0)->(d1) { d1 =#[1] d0 }
let add_d1::[clock=sysclk] = proc(a, b)->(c) {
  x = add(a, b) // function call style 1
  flop(x)->(c)  // function call style 2 (with inherited clock)
}
```

Pyrope provides a pipeline abstraction construct called `pipestage` where staging flops can be implicitly synthesized.

```
let decode_pipe = proc(key, ref cam, ref dec)->(out) {
  val = cam.lookup(key)
} #> {
  val_dec = dec.decode(val) // `val` is auto-flopped
} #> {
  out = val_dec             // `val_dec` is auto-flopped
}
```

Lastly, Pyrope defines an attribute system that allows extensions to the predefined type system and semantics. The following code shows an example use of built-in

attributes, including `wrap`, `comptime`, and `debug`.

```
a:u4 = (20)::[wrap]  // 20 is wrapped into 4 bits = 4
b::[comptime] = 4    // b is a compile-time constant
c::[debug] = a * b   // c can only be propagated to another `debug` scope
d = c                // error - d is not a debug scope
```

The details of each language structure are presented in the following sections.

3.2 Constants

A constant can be an integer, a string, or a boolean value.

Integers have unlimited precision and are signed by default. Any token starting with a digit is an integer constant. Pyrope also accepts common base-2 multipliers like `K,M,G,T` to specify `KiB, MiB, GiB, TiB`.

```
123       // 123 (dec)
0b101     // 101 (bin) = 5 (dec)
0sb1110   // 110 (signed, 2's complement) = -2 (dec)
0xFF      // FF (hex) = 255 (dec)
0o31      // 31 (oct) = 25 (dec)
1K        // 1024 (dec)
1M        // 1K * 1K
```

String constants are enclosed by either a pair of double quotes `"` or a pair of single quotes `'`. Strings with double quotes further accepts escape characters.

```
'abc'
"first line\nsecond line"
```

Finally, a boolean constant can be either `true` or `false`, which has the type equivalent to a one-bit unsigned integer.

3.3 Variables

A variable is an instance with a name binding. A value can be assigned to a variable, where the value of the variable can be used by the variable name. A Variable can be either mutable or immutable by a specified qualifier, `var` or `let` at its declaration. Immutable variables cannot be assigned values more than one time in their scopes. Scopes define the code region where the variable's name binding and hence its value can be accessed. The detailed scoping rules are described in a later section.

```
var a = 1
let b = a + 3
a = 2
let c = a + 3
c = 4         // Error: assigning a new value to an immutable variable
```

3.4 Statements

A statement represents an action to be performed. In Pyrope, there are several categories of statement types, including assignment, conditional, loop, function call, and scope.

3.4.1 Assignment

Assignment statements can be found in many of the previous code snippets. The semantic of an assignment statement is to copy the value of the right-hand side expression to the left-hand side variable except when the variable is declared with a `ref` qualifier. In that case, aliasing or reference semantics will be applied. A new variable

must be declared with its qualifier at the beginning of the assignment statements. Since functions are also treated as a first-class type, function definitions are also specified with assignments.

```
let a = 1     // declaring an immutable variable `a`
var b = 2     // declaring a mutable variable `b`
b = 3         // variable `b` may be assigned twice
let c = a + b // copying the value of an expression to `c`
ref d = b     // `d` is now an alias of `b`
d = 3         // b == 3
// declaring a function definition
let incr = fun(a)->(b) {
  b = a + 1
}
```

3.4.2 Conditional

A conditional statement specifies a set of conditional behaviors that is optionally performed based on a conditional variable. Pyrope provides two types of conditional statements, which are `if` and `match`.

An `if` statement evaluates a sequence of one or more boolean variables and executes the code block that corresponds to the first true condition. When it is guaranteed that only one condition would be matched, a `unique` qualifier should be added to guide the logic synthesizer to synthesize a more optimized parallel matching structure instead of a cascaded matching structure.

```
if cond1 {
  a = 1
  b = 2
} elif cond2 {
  a = 2
  b = 3
} else {
  a = 4
  b = 4
}
unique if a == 2 {
  c = 4
} elif a == 3 {
  c = 5
} else {
  c = 6
}
```

A `match` statement is a unique parallel conditional statement that is equivalent to a `unique if` with an assertion in the `else` branch that verifies exactly one of the branches must be true.

```
match x {
  in 1..<4 { y = 0 }
  >= 4     { y = 1 }
}
// equivalent to
unique if x in 1..<4 {
  y = 0
} elif x >= 4 {
  y = 1
} else {
  assert false
}
```

At synthesis, the compiler performs a static single assignment (SSA) analysis on the branches to make the circuit structure static.

3.4.3 Loop

Pyrope defines two types of loop statements, including `for` and `while`.

A `for` statement consists of an array or a range with an iterator variable and a code block. An index variable can be optionally added after the iterator variable to retrieve the loop index starting from zero. The statements in the code block are executed once per iterator value from the iterator variable. At synthesis, the loop is fully unrolled first, and the compiler also does an SSA for each unrolled block.

```
for i in 0..<100 {
  a[i] = f(i)
}
for n,index in (1,1,2,3,5) {
  b[index] = n
}
```

`while` statements are typically not recommended to use in a hardware language because the compiler has to perform a bound analysis before unrolling the loop. A `while` statement has a condition expression and a code block, where the code block is repeatedly executed while the condition is true.

```
while n > 0 {
  n = n - a[n]
}
```

3.4.4 Function Call

In a hardware language context, every function call is considered inline, meaning that the logic defined in the function has to be instantiated in every function call. This particular semantic is very different from its counterpart in dynamic programming

languages where a function is represented by an instruction pointer and can be reused by different function calls. Furthermore, functions that represent sequential logics, or `proc`-typed functions in Pyrope, are stateful and involve the concept of the clock. Output arguments are not always immediately set when the input arguments are fed. Input arguments may also arrive at the function call in different clock cycles. Adding a `proc` call statement means that the function is called every clock cycle even if it is placed in a conditional branch because the output of the function may take more than one clock cycle to resolve, but the evaluation of the condition is immediate.

The following example illustrates how a function can be implemented with different hardware structures. Every `proc` will be synthesized regardless of the condition. The pipelining syntax like `#[1]` will be discussed later in this chapter.

```
let sub = proc(a,b)->(y) {
  y =# a - b
}
// We want a proc that computes the absolute difference of two values
// waveform:
//    clk |      0       1       2
//      a |      1       x       x
//      b |      4       x       x
//      y |      x       3       x

// do two parallel subtractions, then pick one based on condition
let abs_two_subs_1 = proc(a,b)->(y) {
  // store the value of `a` and `b` and compute condition at cycle 1
  if a#[1] > b#[1] {
    y = sub(a,b)
  } else {
    y = sub(b,a)
  }
}
let abs_two_subs_2 = proc(a,b)->(y) {
  // compute condition at cycle 0 and store the result
  if (a > b)#[1] {
    y = sub(a,b)
  } else {
    y = sub(b,a)
  }
}
// pick inputs of subtraction, then do substraction
let abs_one_sub = proc(a,b)->(y) {
  let l,r = if a > b { (a,b) } else { (b,a) }
  y = sub(l,r)
}
```

In SystemVerilog, only combinational logic with a single output value can be encapsulated as a `function`. Modules with sequential logic or with multiple outputs must be defined with the `module` syntax and called with a static module instantiation. Pyrope further allows a sequential function with one output value, often referred to as a pipeline function, to be called with a regular function call syntax.

```
module add_pipe(
  input  logic        clk,
  input  logic [31:0] a,
  input  logic [31:0] b,
  output logic [31:0] c
);
always_ff @(posedge clk) c <= a + b;
endmodule : add_pipe

module mul_pipe(
  input  logic        clk,
  input  logic [31:0] a,
  input  logic [31:0] b,
  output logic [31:0] c
);
always_ff @(posedge clk) c <= a * b;
endmodule : mul_pipe

module top(
  input  logic        clk,
  input  logic        mode,
  input  logic [31:0] a,
  input  logic [31:0] b,
  output logic [31:0] c
);
logic [31:0] c_add, c_mul;
logic mode_d1;
always_ff @(posedge clk) mode_d1 <= mode;
add_pipe i_add_pipe(clk, a, b, c_add); //  \
mul_pipe i_mul_pipe(clk, a, b, c_mul); //   split across multiple lines
assign c = mode_d1 ? c_add : c_mul;    //  /
endmodule : top
```

```
// Pyrope - module instantiation style
let add_pipe = proc(a:u32, b:u32, ref c:u32) { c::[wrap] =# a+b }
let mul_pipe = proc(a:u32, b:u32, ref c:u32) { c::[wrap] =# a*b }
let top = proc(mode:boolean, a:u32, b:u32, ref c:u32) {
  var c_add:u32, c_mul:u32
  c = if mode#[-1] { c_add } else { c_mul }
  add_pipe(a,b,c_add)
  mul_pipe(a,b,c_mul)
}
```

```
// Pyrope - simple function output style
let add_pipe = proc(a:u32, b:u32)->(c:u32) { c::[wrap] =# a+b }
let mul_pipe = proc(a:u32, b:u32)->(c:u32) { c::[wrap] =# a*b }
let top = proc(mode:boolean, a:u32, b:u32)->(c:u32) {
  c = if mode#[-1] { add_pipe(a,b) } else { mul_pipe(a,b) }
}
```

3.5 Expressions

An expression is a sequence of variables, constants, and operators that produces a single output value. The value of an expression may be assigned to a variable, used in another expression or statement, or treated as the return value of a code block. Expressions are the basis of an HDL as they form the majority the core functions of a hardware design, such as numeric computation, data processing, etc.

The most common expressions are math expressions that operate on integers and booleans.

```
let ty_cmpl = (real:s16, imag:s16)
let a:ty_comp = (1,2)
let b:ty_comp = (3,4)
let c:ty_comp = (a.real * b.real - a.imag * b.imag
                ,a.real * b.imag + a.imag * b.real)
let is_first_quadrant = (c.real >= 0) and (c.imag >= 0)
```

Another group of expressions operates on selection and concatenation over bits, arrays, and tuples.

```
// bit slice
let a = 0xABCD
let b = a@[0..=7]  // b == 0xCD
// member selection
let c = (x=0, y=1)
let d = c.x        // d == 0
// array member selection
let e = (0,1,2,3)
let f = e[1]       // e == 1
// tuple concatenation
let g = (1,2)
let h = (3,4)
let i = g ++ h     // i == (1,2,3,4)
```

In Pyrope, a `if`, `match`, or `for` statement can be used as an expression with the return value of the last statement in each code block as its computed value. This ensures a uniform style to construct statements and expressions, which makes code easier to comprehend.

```
// Inconsistent with if-else statement style
assign x_abs = (x > 0) ? x : -x;
// Nested ternary operators are hard to read
assign y_cutoff = (y > param.MIN) ? (
  (y < param.MAX) ? y : param.MAX) : param.MIN;
// Alternatie style - too wordful
always_comb begin
  unique if (y < param.MIN)
    y_cutoff = param.MIN;
  else if (y > param.MAX)
    y_cutoff = param.MAX;
  else
    y_cutoff = y;
end
// The semantic is to compute `z_normalized`, however
// the final assignment is hidden in the for loop
for (genvar i = 0; i < Z_RAW_SIZE; i++) begin
  NTYPE n_scaled, n_offset, n_quantized;
  assign n_scaled = z_raw[i] * param.SCALE;
  assign n_offset = n_scaled + param.OFFSET;
  assign n_quantized = quantize(n_offset);
  assign z_normalized[i] = n_quantized; // Here
end
```

```
// Consistent style
let x_abs = if (x > 0) { x } else { -x }
if (x > 0) { x_abs = x } else { x_abs = -x }
// Unique parallel match instead of long if-else chain
let y_cutoff = match y {
  < param.MIN { param.MIN }
  > param.MAX { param.MAX }
  else        { y }
}
// Cleaner code
let z_normalized = for n in z_raw {
  let n_scaled = n * param.SCALE
  let n_offset = n_scaled + param.OFFSET
  let n_quantized = quantize(n_offset)
  n_quantized
}
```

Figure 3.1: Pyrope unifies statement and expression styles for conditional assignments and array computations.

3.6 Scopes

A scope defines the region of source code where a variable name binding is available. Scopes can be either unnamed or named, where unnamed scopes are given an auto-incremented integer identifier as the reference name. There are three types of structures that can create a new scope, which are source files, code blocks, and tuples. Scopes can be nested. A local scope has all the name binding of its upper scopes. However, variable shadowing is not allowed except in tuple scopes. Each scoping rule is described as follows.

3.6.1 Source Files

Each source file forms a library scope with its filename as the scope name, which implies the compiler can concurrently process each file and elaborate at a later stage. This compilation framework is different from Verilog's single compilation unit structure, where module definitions across all the source files share the same global scope. SystemVerilog further introduces the `package` construct that allows a separate compilation of function or task definitions, parameters, and type definitions. SystemVerilog packages cannot contain any design modules, whereas Pyrope's library may enclose any variable declarations and definitions, including function variables representing design modules. The `import` function imports a library as a tuple where members are variables declared at the root level of the imported file.

```
// arith.prp
let add = fun(a,b)->(c) { a + b }
let mul = fun(a,b)->(c) { a * b }
```

```
// mac.prp
let arithlib = import("arith.prp")
let mac = fun(a,b,c)->(d) {
  arithlib.add(arithlib.mul(a,b),c)
}
```

Figure 3.2: Library scopes and library imports.

3.6.2 Code Blocks

A code block encloses a sequence of statements with `{ ... }`. A code block itself can act as a regular statement, which can be quite useful when the user wants to define temporary local variables but wants to avoid conflicting variable names. A code block can even serve as an expression with the last statement's return value as the code block's value. This feature promotes cleaner code where intermediate computations are annotated, as shown in Figure 3.6.2.

Code blocks are used in conditional statements and loop statements. For example, any branch of an `if` statement can have a declaration of a variable with a name that is already defined in the upper scope, which may be referenced within the scope and its lower scopes. Function definitions also use code blocks to enclose logic and local variables. The scope of the top-level code block of a function definition also covers the input and output port variables. Variable name shadowing is not allowed in

```
// One-liner code
logic [31:0] result, another;
assign result = (a * b) + c;
assign another = (a[15:0] * b[15:0]) + c;
```

```
// No declaration allowed in a `always_comb` block
logic [31:0] result, result_mult, another, another_mult;
always_comb begin
  result_mult = a * b;
  result = result_mult + c;
  another = a[15:0] * b[15:0]
  anothe_mult = another + c;
end
```

```
// Self-explanatory one-liner code
let result:u32:[wrap] =
  {result_mul:u32:[wrap] = a * b; (result_mul + c) }
// Variable name binding is contained in the code block
let another:u16:[wrap] =
  {result_mul:u16:[wrap] = a[0..15] * b[0..15]; (result_mul + c)}
```

Figure 3.3: Code blocks as expressions.

either regular statements or function definitions, as shown in Figure 3.6.2.

```
let f = fun(x,y)->(z) {
  x_adj = x + 1
  x_adj = y + 1
  if (x_adj + y_adj > 16) {
    // let x_adj = x << 2  // Error: variable shadowing
    // let y_adj = y << 2  // Error: variable shadowing
    let x_shr = x << 2
    let x_shr = y << 2
    z = x_shr + y_shr
  } else {
    z = x_adj + y_adj
  }
}
```

Figure 3.4: Variable shadowing is not allowed.

3.6.3 Tuples

A tuple defines a scope named `self` that can only be referenced by its own members. The members are ordered, and optionally named. If a tuple has a member named $Name$ and in position P, other members can reference this member by eithe `self.`$Name$ or `self.`P. This is similar to C++'s `this` and Python's `self` pointer. In a hardware language context, pointers are pure name aliases with no dynamic memory layout implication.

```
let t = (a = 8, b = 8, c = self.a + self.b)
let s = (x = 1, y = self.0 + 1)
```

Figure 3.5: Tuple scope.

3.7 Pipeline

Pipelines are sequential circuits commonly used in hardware design. A pipeline is built from a set of pipeline stages. A pipeline stage in the middle of a pipeline construct stores computed or delayed values in a pipeline register, and the next pipeline stage can read the value from that pipeline register. Pyrope defines the cycle select expression `<variable>#[<cycle>]` that automates the instantiation and assignment of pipeline registers. The semantic meaning of `<variable>#[<cycle>]` is to generate a chain of `<cycle>-1` of registers and take the value of the last one.

```
module fir(
  input          clk,
  input  integer x,
  input  integer w[0:3],
  output integer y
);
  integer x_pipe[1:3];          //
  always_ff @(posedge cl) begin //
    x_pipe[1] <= x;             // boilerplate code for generating
    x_pipe[2] <= x_pipe[1];     // pipeline registers
    x_pipe[3] <= x_pipe[2];     //
  end                           //
  assign y = w[0] * x
           + w[1] * x_pipe[1]
           + w[2] * x_pipe[2]
           + w[3] * x_pipe[3];
endmodule
```

```
let fir = proc(x,w) -> (y) {
  y =# w[0] * x#[0]
     + w[1] * x#[1]
     + w[2] * x#[2]
     + w[3] * x#[3]
}
```

Another special Pyrope syntax for pipelines is called `pipestage`, which uses `#>` to chain logics in different pipeline stages together. Each `#>` adds an implicit cycle delay for variables referenced from the previous scope. The following example shows two equivalent pipeline designs in Pyrope, where the second one utilizes the `pipestage` syntax.

```
let decode_pipe = proc(key, ref cam, ref dec)->(out) {
  let val = cam.lookup(key)
  let val_dec = dec.decode(val#[1])
  out = val_dec#[1]
}
```

```
let decode_pipe = proc(key, ref cam, ref dec)->(out) {
  {
    let val = cam.lookup(key)
  } #> {
    let val_dec = dec.decode(val) // `val` is auto-flopped
  } #> {
    out = val_dec             // `val_dec` is auto-flopped
  }
}
```

3.8 Types

All variables in Pyrope are typed. A type in Pyrope defines the physical attribute and the logical structure of a variable and can be either primitive or composite. In Pyrope, there are two ways of using types, type specification or type check. When a type is assigned to a variable at the declaration or the left-hand side of an assignment, it is treated as a type specification. When a type is added to a variable at the right-hand side of an assignment or to an expression, it acts as a type check.

```
// type specification
//     |   type check
//     |       |
//     V       V
var a:u32 = b:u32
var c:u32 = (d + e):u32
```

Table 3.8 lists all the available types in Pyrope, including primitive types and composite types that are built on top of primitive or other composite types.

Class	Name	Grammar	Example
primitive	unsized signed integer	`int` or `sint`	`a:sint = -1`
	unsized unsigned integer	`uint`	`a:uint = 1`
	sized signed integer	`s`*Num*	`a:s8 = 127`
	sized unsigned integer	`u`*Num*	`a:u8 = 255`
	boolean	`boolean`	`a:boolean = true`
	string	`string`	`a:string = "abc"`
	type	`type`	`a:type = boolean`
composite	unsized array	*Type*	`a:[u8] = (0,1,2,3)`
	sized array	*Type*`[`*Num*`]`	`a:u8[4] = (0,1,2,3)`
	tuple	`(:`*Type*`,:`*Type*`,...)`	`a:(:u8,:u4) = (255, 15)`
	mixin	*Type* `++`*Type*	`a:proto_1 ++ proto_2`
	lambda (combinational)	`fun(:`*Type*`...)`	`a:fun(:u8)->(:u8) =`
		`->(:`*Type*`,...)`	`fun(x)->(y) {y = x}`
	lambda (sequential)	`proc(:`*Type*`...)`	`a:proc(:u8)->(:u8) =`
		`->(:`*Type*`,...)`	`proc(x)->(y) {y =# x}`

Table 3.1: List of supported types in Pyrope.

3.8.1 Integer Types

Variables associated with circuit structures, like wires and flip-flop arrays, must have their bitwidths resolved before synthesis. Therefore bitwidths must be encoded in the type system. The bitwidth information is also useful in synthesizing arithmetic or

logical circuits where the compiler may optimize the datapath based on the bitwidths of input variables. The compiler must be capable of inferring the implicit bitwidth of any variable and checking if user-specified bitwidths match the inferred bitwidths.

Integer types are the basis of a hardware type system, as every variable needs to be converted into integers and eventually split into their binary forms. Unlike software programming languages, there are no constraints on word-aligned datatypes like `uint32_t`. As a minimum requirement, an integer-type system of a hardware language should be able to encode any bitwidth value. Moreover, integer types may encode signedness and bit index ordering for the ease of implementing arithmetic and bitwise operations respectively. SystemVerilog defines both signedness and bit index range and ordering, while Pyrope has only signedness and bitwidth, and bit indices default to ascending from zero. The reason why Pyrope leaves out bit index ordering (ascending and descending) and range (MSB index and LSB index) is to avoid ambiguity when operands have different bit orderings.

```
logic unsigned [31:0]  a;
logic signed   [31:0]  b;
logic          [63:32] c;
```

```
let a:u32 = 0 // unsigned sized integer
let a:s32 = 0 // signed sized integer
let c:u32 = 0 // unsigned sized integer
```

A good integer-type system should also be able to check for bitwidth mismatches and arithmetic overflows in the code. This feature requires a compiler pass called bitwidth propagation, where the compiler would compute the bitwidth of the

value of each expression. To further take advantage of the bitwidth propagation pass, Pyrope allows unsized integer types that do not specify the bitwidth in the code directly but rather take the bitwidth inferred by the pass.

```
// `c` should be `u33`
let c:uint = a:u32 + b:u32
// `d` should be `u16`
let d:uint = if c > 100 { e:u4 } else { e:u16 }
```

3.8.2 Logical Types

Logical structures can also be represented by types, such as enums, arrays, and tuples. Logical types make a language more expressive, succinct, and safe in many ways. With enum types, users do not have to memorize the encoding of a particular state, and values assigned to enum types must be one of the defined values. With array types, repetitive structures can be created without duplicate code, and loop statements and expressions make code more compact and cleaner. Tuple types can be seen as `struct` types in C++, which group members of mixed types together, and members can be accessed by either their positions or names if specified.

```
let state:enum(reset,init,busy,done)
let seq:[u32] = (0,1,2,3)
let complex = (real:u32 = 0, imag:u32 = 0, valid:boolean = false)
```

3.8.3 Lambda Types

One distinct feature of Pyrope is to treat functions as first-class types, meaning that a function can be assigned to a variable and passed as an argument to another

function. A function type is a composite type that is made up of input argument types and output argument types. The function definition is not encoded into the function type.

```
let ty:type = fun(:u32)->(:u32)
let incr = fun(a:u32)->(b:u32) { b::[wrap] = a+1 }
let decr = fun(a:u32)->(b:u32) { b::[wrap] = a-1 }
let do_twice = fun(a:u32,f:ty)->(b:u32) {
  b = f(f(a))
}
let x = do_twice(4,incr) // x == 6
let y = do_twice(4,decr) // y == 2
```

3.9 Attributes

One of the main targets of Pyrope is to lower the barriers to customizing and extending the language and the compiler. Instead of giving a long list of pre-defined types and their semantics, Pyrope defines a flexible attribute system that augments the type system and allows users to define the semantic meanings of each attribute field through the compiler directly. LiveHD provides the compiler infrastructure, including a pass manager and a pass template so that the user can easily build the pass required to process their custom attributes. In addition, the µPass framework, which will be introduced in the next chapter, is designed to reduce the effort in building a new pass and improve the performance of custom compilation pipelines.

An attribute can be assigned to any variable with the type specification syntax `<variable>:<type>:[<attributes>]`. The attribute field can be seen as a dictionary that maps an attribute name to its value. When the value is not specified in the attribute

field, it defaults to be `true`. There are three types of operations on attributes: attribute set, attribute check, and attribute get. The example below shows how to use each attribute operation.

```
let a:u32:[attr=2]  // atrtibute set
let b = a::[attr>1] // attribute check
let c = a.::[attr]  // attribute get
let d::[attr]       // equivalent to `let d::[attr=true]`
```

The following subsections present some interesting examples of attributes in verification and physical design.

3.9.1 Attributes for Verification

In production code, many of the verification components have to be placed within a design code that will be synthesized into circuits. However, the verification code should never be synthesized, and the design code should never use any data computed in the verification logic. A common solution is to put the verification code within a define guard that is set to be disabled during synthesis.

In Pyrope, it is possible to replace define guards with a `debug` attribute. When a variable is specified with a `debug` attribute, the variable and all the expressions using this variable would not be synthesized. Using this variable in a scope that is not declared as `debug` will trigger a compile error.

```
let is_between_r1_and_r2::[debug] = fun(x, r1, r2) -> (_) {
  ret (r1 <= x <= r2)
}
// Error: Assigning a `debug` variable to a `nodebug` variable
let design_variable = is_between_r1_and_r2(x, 0, 7)

// Correct: Using `debug` variable in a `debug` scope
let verifier::[debug] = fun(x) {
  cassert is_between_r1_and_r2(x, 0, 7)
}
```

Figure 3.6: An example using `debug` attributes to avoid mixing design logics and verification logics

Another example of attributes for verification is the `rand` attribute, which indicates that the value of the variable is uniformly randomized. The randomized value can be generated from the simulator's host machine, or be synthesized as a pseudo-random number generator in an FPGA emulation environment.

```
let rand_seq_item = fun {(
  ,cfg:enum(A,B):[rand,comptime] // compile-time random
  ,address:u46:[rand]            // simulation-time random
)}
```

Figure 3.7: An example using `rand` to generate random data.

3.9.2 Attributes for Physical Design

While SystemVerilog supports many netlist-level constructs, such as primitive gates and gate delay specification, these primitives are often only used in intermediate products in the physical design flow. Physical design attributes are often not specified in

RTL code, but instead defined in synthesis scripts and place-and-route configurations. One may argue that abstracting logical structures out of the physical implementation is better for reusing the same code with different technologies and libraries. The reality is that sometimes a designer wants to guide the synthesis of the code and obtain a more predictable result from the physical design flow. One common method to specify synthesis attributes is by writing a comment in a specific format for the synthesis tool to locate and read. As shown in the code below, these attributes can be directly assigned to the target variables, which eliminates any ambiguity.

```
reg r /* synthesis maxfan = 16 */;
/* synthesis syn_encoding = "one-hot" */
reg [1:0] fsm_state;
```

```
reg r::[maxfan=16];
reg fsm_state:int:[syn_encoding="one-hot"];
```

Figure 3.8: Comparing custom synthesis pragmas in SystemVerilog (Top) with custom attribute fields in Pyrope (Bottom)

Clock and reset signals can be specified in attribute fields instead of being passed around as variables. The code below shows an example use of `clock` and `reset` attributes.

```
let subsystem::[clock=clk_L2, reset=glb_rst] = proc(inp)->(out) {
  out.pkt.data::[cell=ultrafast]
    = normalize(inp.pkt.a * inp.pkt.b + inp.pkt.c)::[critical]
}
```

Chapter 4

Compiler Implementation

This work contributes to several parts of the Pyrope compiler. First, a language parser is constructed to parse the language as specified in the previous chapter. Second, a translation pass that converts a parse tree into the IR format is partially completed. Third, the IR used in the compiler framework is formally specified and documented to avoid any ambiguity across the development team. Fourth, a new pass framework is prototyped to allow future compiler developers to write efficient passes. Lastly, the debugging and testing infrastructure of the compiler framework is greatly improved with the introduction of a formal textual IR format and a regression testing utility.

4.1 Parser

The first step to build a compiler for a language is to define the complete grammar and build a parser that accepts all valid code sequences and rejects all invalid ones. The parser should also translate the code sequence into an equivalent parse tree

that can be further processed by the subsequent compiler passes.

The previous Pyrope parser was hand-written with top-down parsing that directly generates an abstract syntax tree. However, manual parser construction would be very challenging as there are close to a hundred production rules with possible rule conflicts. Manual parsing is also less flexible if we want to experiment with different syntax styles. To timely bring up a much heavier language, the new parser of the Pyrope language is implemented using a parser generator called Tree-sitter [8]. Tree-sitter uses a grammar data structure in JavaScript as the input specification and generates a C parser with generalized LR (GLR) parsing. The generated parser checks if a string matches the grammar and converts it into a parse tree, where the intermediate nodes can be traversed and retrieved via Tree-sitter's library functions.

The main reason to use Tree-sitter other than the alternatives, such as Lex/Y-acc or ANTLR, is that Tree-sitter can do incremental parsing with a well-established error recovery mechanism. Even when the input string violates the grammar rule, the generated parser can still produce a parse tree with error nodes nearby the grammar violation sites. Furthermore, many syntax highlighters use Tree-sitter as the core due to its fast response time. The latency to generate a new parse tree is often shorter than the delay between keystrokes with a reasonably sized source file. This aligns with the target of LiveHD where most of the compiler passes need to support incremental and parallel processing to shorten the Read-Eval-Print loop (REPL).

The complete grammar in Tree-sitter's JavaScript specification form can be

found in our repository[1]. The default lexer that comes with the Tree-sitter parser was extended to simplify the newline-separated statement list grammar by adding semicolons at proper line ends, similar to JavaScript's automatic semicolon insertion (ASI). The grammar has been refined several times to shrink the size of the specification and reduce the number of conflicting rules. Moreover, to make sure the parser always implements the most updated grammar rules, it is tested against the full collection of code snippets grabbed from Pyrope's documentation webpage.

4.2 IR Construction

The step after parsing is to lower the parse tree into a lower-level IR. There are two levels of IRs in LiveHD, which are LNAST, which stands for language-neutral abstract syntax tree, and LGraph, a graph IR that can encode netlist structures. In Pyrope compilation, the parse tree is first lowered into LNAST, which is subsequently processed by LNAST transformation passes that are already in LiveHD. Once all the required LNAST passes are completed, an IR-lowering pass converts the LNAST to a Lgraph netlist. Several Lgraph passes further optimize the netlist, and finally, a code generation pass serializes the netlist to a target language or IR, such as Verilog or FIRRTL. Figure 4.1 illustrates the compilation process.

The rest of the section presents the structure of an LNAST and shows how the translation pass lowers the parse tree to an equivalent LNAST.

[1] `https://github.com/masc-ucsc/tree-sitter-pyrope`

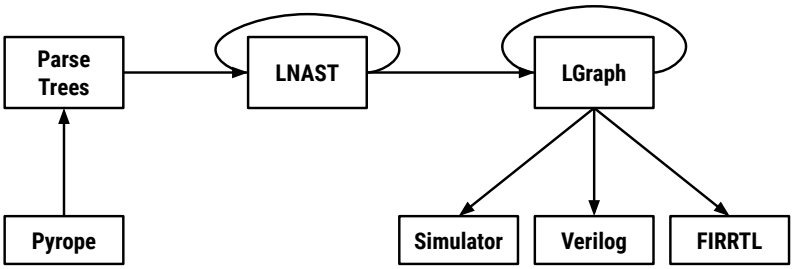

Figure 4.1: Pyrope compilation flow in LiveHD.

4.2.1 LNAST

LNAST is a tree IR that can represent behavioral-level logic. The underlying data structure is a vector n-ary tree optimized for small numbers of branches at each node. Each tree node is a `Lnast_node` data structure that stores the node type, line number, token position, and an optional string value. The contribution of this work includes documentation of the LNAST format and several additions to the set of node types for accommodating Pyrope's type system.

The tree structure of an LNAST is kept mostly flattened to avoid recursions while being processed by a compiler pass. Only code blocks and composite types are allowed to branch. The following describes the specification of each LNAST node type.

Reference and constant nodes are always leaf nodes, as well as most of the primitive type nodes except for any integer type node where it may have one child as the bitwidth. Range nodes have exactly two child nodes representing the lower and the upper bound.

```
<ref>               : string value is the reference name
<const>             : string value is the numerical/string value
<prim_type_boolean> : leaf primitive type node
<prim_type_uint> --| <ref/const> : integer type node with a child node
<range>          --| <ref/const> : lower bound of range
                   | <ref/const> : upper bound of range
```

A `stmts` node represents a code block made up of a sequence of statements. It can be placed under `top` node, which is the root node of an LNAST instance, or a statement node that requires a code block in its structure. `stmts` nodes are annotated as C-1 to C-N when used in semantic description.

```
<stmts> --| <assign> --| ...
          | <assign> --| ...
          | <if>     --| <ref>
          |             | <stmts> --| ...
          | ...
```

As a general node positioning rule for nodes with assignment semantics, the left-hand-side (LHS) references are always put in the first child, whereas the right-hand-side (RHS) values are put after all the LHS references. LHS nodes and RHS nodes are annotated as L-1 to L-N and R-1 to R-N when used in the semantic description.

In simple value assignment nodes, the LHS reference consists of a single reference node and the RHS can be either a single constant or a reference node. For type assignments, the RHS must be a type node. Note that it is not allowed to have nested computations at the RHS, meaning that intermediate results of an expression must be assigned to temporary reference nodes.

```
<assign>    --| <ref>               : L-1
              | <ref/const>         : R-1
<type_spec> --| <ref>               : L-1
              | <prim_type_boolean> : R-1
<type_def>  --| <ref>               : L-1
              | <prim_type_uint>    : R-1
```

In tuple and attribute member assignment nodes, an LHS reference consists of a sequence of mixed reference or constant nodes accessing members hierarchically.

```
<tuple_set> --| <ref>       : L-1
              | <ref/const> : L-2
              | <ref/const> : L-3
              | ...
              | <ref/const> : R-N
```

RHS values can also be a sequence of mixed references and constants. This pattern is used in tuple and attribute gets, binary/n-ary operations, and bit manipulations.

```
<tuple_get> --| <ref>       : L-1
              | <ref>       : R-1
              | <ref/const> : R-2
              | ...
              | <ref/const> : R-N
```

The RHS values represent the operands of the operation in math operator nodes. The semantics of each math operator node are listed in Table 4.1.

```
<plus> --| <ref>       : L-1
         | <ref/const> : R-1
         | <ref/const> : R-2
         | ...
         | <ref/const> : R-N
```

A function definition or a for-loop statement node has one `stmts` node representing the main code block of the node. An if-elif-else statement may have multiple

code blocks, and multiple branches of condition and `stmts` pairs are allowed under the `if` LNAST node.

```
<func_def> --| <ref>    : L-1
             | <ref>    : R-1
             | <ref>    : R-2
             | <stmts>  : C-1
```

```
<for> --| <ref>   : R-1
        | <ref>   : R-2
        | <stmts> : C-1
```

```
<if> --| <ref/const> : R-1
       | <stmts>     : C-1
       | <ref/const> : R-2
       | <stmts>     : C-2
       | ...
       | <ref/const> : R-(N-1)
       | <stmts>     : C-(N-1)
       | <stmts>     : C-N
```

A composite type node can have either primitive type nodes or another nested composite type node as its child.

```
<comp_type_lambda> --| <comp_type_tuple> --| <prim_type_uint>
                     |                     | <prim_type_uint>
                     | <comp_type_tuple> --| <prim_type_uint>
<comp_type_array>  --| <comp_type_array> --| <prim_type_uint>
                     |                     | <const>
                     | <const>
```

Table 4.1 summarizes the list of `Lnast_node` types and their semantics.

Table 4.1: `Lnast_node` types

Group	Type	Description
Scope	`top`	root node
	`stmts`	a sequence of statements
Primitive	`ref`	a variable with string value as the name

Table 4.1: `Lnast_node` types

Group	Type	Description
	`const`	A constant value with string value as the constant value
	`ranges`	A range bounded by R-1 and R-2
Statement	`if`	if-elif-else statement
	`for`	for statement
	`func_def`	function definition
	`func_call`	function call
	`assign`	variable assignments
Unary Expression	`bit_not`	bitwise not of R-1 (flip all bits)
	`reduce_or`	R-1 contain one or more set bits
	`logical_not`	logical not of R-1 (R-1 must be a `boolean`)
Binary Expression	`mod`	R-1 modulo R-2
	`shl`	R-1 left shifted by R-2
	`sra`	R-1 right shifted by R-2
	`ne`	R-1 not equal to R-2
	`eq`	R-1 equal to R2
	`lt`	R-1 less than R-2
	`le`	R-1 less than or equal to R-2
	`gt`	R-1 greater than R-2
	`ge`	R-1 greater than or equal to R-2
N-ary Expression	`bit_and`	bitwise and of R-1 to R-N
	`bit_or`	bitwise or of R-1 to R-N
	`bit_xor`	bitwise xor of R-1 to R-N
	`logical_and`	logical and of R-1 to R-N
	`logical_or`	logical or of R-1 to R-N
	`plus`	summation of R-1 to R-N
	`minus`	R-1 minus summation of R-2 to R-N
	`mult`	product of R-1 to R-N
	`div`	R-1 divided by product of R-2 to R-N
Bit manipulation	`sext`	Sign extension of R-1
	`set_mask`	set the bits of R-1 to L-1 where the mapped bit in mask R-2 is set

Table 4.1: `Lnast_node` types

Group	Type	Description
	`get_mask`	get the bits of R-1 where the mapped bit in mask R-2 is set
	`mask_and`	get the bitwise and of R-1 with mask R-2
	`mask_popcount`	get the number of ones in R-1 with mask R-2
	`mask_xor`	get the bitwise xor of R-1 with mask R-2
Tuple	`tuple_concat`	concatenation of R-1 to R-N (must all be tuples)
	`tuple_add`	tuple construction where R-1 to R-N can be a single reference or constant, or an assignment
	`tuple_set`	set the value of tuple L-1's member L-2's ...member L-N to R-1
	`tuple_get`	get the value of tuple R-1's member R-2's ...member R-N and assign it to L-1
Attribute	`attr_set`	set the value of tuple L-1's member L-2's ...attribute L-N to R-1
	`attr_get`	get the value of tuple R-1's member R-2's ...attribute R-N and assign it to L-1
Type	`type_spec`	assign L-2 as L-1's type
	`type_def`	name type R-1 as L-1
	`none_type`	untyped
	`prim_type_uint`	unsigned integer type
	`prim_type_sint`	signed integer type
	`prim_type_range`	range type
	`prim_type_string`	string type
	`prim_type_boolean`	boolean type
	`prim_type_type`	type-kind type
	`prim_type_ref`	type reference
	`comp_type_tuple`	tuple type with ordered type sequence R-1 to R-N
	`comp_type_array`	array type with base type R-1 and optional size R-2
	`comp_type_mixin`	mixin type with unordered type set R-1 to R-N
	`comp_type_lambda`	lambda type with input type R-1 and output type R-2
	`comp_type_enum`	enum type with ordered member name sequence R-1 to R-N

Table 4.1: `Lnast_node` types

Group	Type	Description
	`unknown_type`	unknown type

4.2.2 Translating Pyrope to LNAST

The front-end pass that translates Pyrope to LNAST is implemented in LiveHD as `inou.prp`. The input parse tree is traversed in preorder, and the pass appends appropriate LNAST nodes to the output LNAST data structure throughout the process. A required technique to convert a hierarchical structure of a parse tree to a flattened structure of LNAST is the use of temporary variables. In the example below, to follow the rules of LNAST, the intermediate result of a sub-expression has to be assigned to a temporary variable and later used in another expression.

```
// Pyrope
if a+b > 0 {
  x = 1
} else {
  x = 2
}
```

```
// LNAST (ln format)
{
  %{___t0} = add(%{a}, %{b})
  %{___t1} = gt(%{___t0}, 0)
  if (%{___t1}) {
    %{x} = 1
  } else {
    %{x} = 2
  }
}
```

The translation pass is currently in active development and only supports a subset of the Pyrope language. An example code supported by the translation pass is shown in the next chapter.

4.3 µPass

A µPass (pronounced micro-pass) or `upass` is a lightweight compiler pass that only operates within certain scopes and node types of an AST. A µPass is similar to an operation pass in MLIR [15] that is tied to a specific operator and only invoked when the operator is being processed. This concept of a modular pass framework and fusing tree traversal passes is also studied in [19] and [20]. One of the benefits of implementing a pass as a µPass instead of a full compiler pass is that multiple µPasses may operate on the same node during a single LNAST traversal instead of multiple LNAST traversals, which exploits both spatial and temporal locality, leading to better runtime. The caveat is that µPasses may have dependencies among each other. The current implementation of the µPass manager does not resolve the dependencies itself but rather relies on the user's own scheduling. In the future, a dependency resolution step will be added to the µPass manager to automate this process.

There are currently two µPasses implemented in LiveHD, which are `upass.verifier` and `upass.constprop`. `upass.verifier` checks if an LNAST node's structure satisfies the rules discussed in the LNAST section. `upass.constprop` propagates constant values through expressions if the RHS values are all constants.

4.4 Debug & Testing Infrastructure

A good compiler infrastructure needs to provide a robust and complete debug and testing environment so that compiler developers can quickly triage the root cause of each issue and fix it timely. The debug and testing infrastructure is even more critical to the Pyrope toolchain as it is designed to be compiler-centric. Learned from several compiler projects, three feature improvements have been implemented into the LiveHD compiler, including a textual IR format, a regression test suite library, and a serialization/deserialization interface.

4.4.1 Textual IR for LNAST

A textual representation of the IR is critical to the productivity of compiler development. It provides a human-friendly interface for compiler developers to look into the input and the output of the compiler pass in development. Furthermore, unit tests can also be written in the textual format by humans to test any compiler pass without going through a front-end language or constructing the IR by code.

LiveHD compiler already has an LNAST dumping function that can serialize the tree structure and each tree node from an LNAST instance. However, debugging using tree dumps is not very efficient in several ways. First, LiveHD cannot deserialize the dumped text back to an LNAST data structure, which means that compiler passes are mostly tested by a corpus of front-end languages and rely on front-end parsers' correctness and stability. The ideal way to conduct unit testing of a compiler pass is

by feeding the input directly in a textual IR format and checking the output, again in a textual IR format. Second, a tree dump takes up many lines as each child of a subtree occupies a line. For example, a simple assignment takes three lines while the information can be encoded in one line with no loss in clarity.

A textual IR format named `ln` is created to provide a much more compact visual representation of the IR. Figure 4.4.1 compares the output of an LNAST tree dump and the equivalent `ln` code.

Two interface passes were added to LiveHD, a writer pass (`pass.lnast_write`) and a parser pass (`pass.lnast_read`). The writer recursively traverses through each node in an LNAST structure and prints out text specified in the grammar either to a file or to the terminal. Printed code on a terminal further provides simple syntax highlights for better readability as shown in 4.3 The parser pass is implemented with a custom lexer and a hand-written top-down parser that reads `ln` code and converts it into the in-memory LNAST data structures. The reason to use a custom parser instead of a generated parser is that the grammar is fairly simple. The serialization and deserialization performance is also critical when a larger design is being actively worked on.

```
{
  %{___t0} = %{b}."debug"
  cassert(%{___t0})
  %{b} : #u(8)
  %{___t1} = add(%{a}, %{b})
  %{x} = %{___t1}
}
```

Figure 4.3: `ln` syntax highlighting in terminal.

```
x = a + b:u8:[debug]
```

```
0-0  (0,0    )      top     :
0-0  (1,0    )          stmts   :
0-0  (2,0    )              attr_get:
0-0  (3,0    )                  ref     : ___t0
0-0  (3,1    )                  ref     : b___0
0-0  (3,2    )                  const   : debug
0-0  (2,1    )              cassert :
0-0  (3,4    )                  ref     : ___t0
0-0  (2,2    )              type_spec:
0-0  (3,8    )                  ref     : b___0
0-0  (3,9    )                  prim_type_uint:
0-0  (4,0    )                      const   : 8
0-0  (2,3    )              plus    :
0-0  (3,12   )                  ref     : ___t1
0-0  (3,13   )                  ref     : a___0
0-0  (3,14   )                  ref     : b___0
0-0  (2,4    )              assign  :
0-0  (3,16   )                  ref     : x___0
0-0  (3,17   )                  ref     : ___t1
```

```
{
  %{___t0} = %{b}."debug"
  cassert(%{___t0})
  %{b} : #u(8)
  %{___t1} = add(%{a}, %{b})
  %{x} = %{___t1}
}
```

Figure 4.2: Comparing direct LNAST tree dump and `ln` format
(Top) Pyrope source code (Middle) Direct LNAST tree dump (Bottom) `ln` format dump

4.4.2 Diagnostic Tests

Changing any source code that falls within the compilation pipeline may break the pipeline, which makes the compiler less stable. One way to avoid the introduction of a new functional bug is to create a corpus of diagnostic tests and run regressions on them. However, during the development process, some tests may only be valid for a part

```
/*
:name: attributes
:type: parsing
*/
let top = fun(a::[comptime], b::[comptime], c) {
  x::[attr=(1,2)] = _
  c = (a::[comptime] = 1, b::[comptime], c)
  cassert c::[comptime=0]
}
```

Figure 4.4: A diagnostic test in `ln` that should pass the `parsing` test.

of the compilation pipeline. For example, while building a translation pass, we might want to add tests that can only be processed by the translation pass, but not by the optimization passes yet. To classify tests based on the set of supported passes, a header format that specifies the properties of a test is defined and can be read by a Python test runner library. This testing infrastructure is inspired by [7], which implements a similar structure for testing SystemVerilog tools.

4.4.3 Serializing and Deserializing LNAST with HIF

One of the main targets of LiveHD is to support incremental compilation. Across incremental runs, the intermediate product throughout the compilation flow needs to be serialized to a file and deserialized when needed by a later run. While a textual IR improves the productivity in testing and debugging compiler, it is suboptimal in space efficiency and memory performance. A compact binary encoding is more desirable.

The hardware interchange format (HIF) was adopted as the binary format for

LNAST, and two interface passes, `pass.lnast_save` and `pass.lnast_load`, are implemented to translate between LNAST and HIF.

Chapter 5

Experimental Result

5.1 Language Improvement

There are several differences between the old language and the revised language. First, bitwidths are now encoded into the type system, which has to be assigned to a reserved attribute in the old language. Second, the old language uses dollar signs and percent signs to identify function input variables and output variables, which are removed in the new language for a better readability. Third, the revised language provides a cleaner function definition and call syntax. Lastly, the if statement is the only conditional statement in the old language, whereas the new language provides the match construct and allows conditional statements to be used as expressions. Figure 5.1 compares code written in the revised language and the old language.

```
// Old
f = ||{ %y = !$x }
$a.__ubits = 63
$b.__ubits = 63
c = $a ^ $b
d = (x = t) |> f
if (d > 0) {
  %y = -1
} elif (d == 0) {
  %y = 0
} else {
  %y = 1
}
```

```
// New
let top = fun(a:u63, b:u63) -> (y) {
  let f = fun(x)->(y) { y = !x }
  c = a + b
  d = f(t)
  y = match d {
    > 0  { -1 }
    == 0 {  0 }
    else {  1 }
  }
}
```

Figure 5.1: Comparing the revised Pyrope language (Bottom) with the old Pyrope language (Top).

5.2 Compiler Execution

The Pyrope design below demonstrates the supported language features of the translation pass, including type specifications, assignments, conditional statements, tuples, and attributes.

```
let a:s32 = 10
let b:s32 = 20
var y = 0
if mode::[comptime] {
  let st = (1,2)
  let x = a + b
  match x {
    <  10 { y = st[0] }
    >= 10 { y = st[1] }
  }
} else {
  let t = (a=0,b=1)
  let x = a - b
  if x == 10 {
    y = t.a
  } elif x > 10 {
    y = t.b
  }
}
```

The following LiveHD shell command compiles the Pyrope design into LNAST, and prints out the LNAST in `ln` format.

```
> inou.prp files:example.prp |> pass.lnast_print
/*
  :name: example
*/
{
  %{a} : #s(32)
  %{a} = 10
  %{b} : #s(32)
  %{b} = 20
  %{y} = 0
  %{___t0} = %{mode}."comptime"
  cassert(%{___t0})
  if (%{mode}) {
    %{___t1} = (1, 2)
    %{st} = %{___t1}
    %{___t2} = add(%{a}, %{b})
    %{x} = %{___t2}
    %{___t3} = lt(%{x}, 10)
    %{___t4} = ge(%{x}, 10)
    if (%{___t3}) {
      %{___t5} = %{st}[0]
      %{y} = %{___t5}
    } elif (%{___t4}) {
      %{___t6} = %{st}[1]
      %{y} = %{___t6}
    }
  } else {
    %{___t8} = (%{a} = 0, %{b} = 1)
    %{t} = %{___t8}
    %{___t9} = sub(%{a}, %{b})
    %{x} = %{___t9}
    %{___t10} = eq(%{x}, 10)
    %{___t11} = gt(%{x}, 10)
    if (%{___t10}) {
      %{___t12} = %{t}["a"]
      %{y} = %{___t12}
    } elif (%{___t11}) {
      %{___t13} = %{t}["b"]
      %{y} = %{___t13}
    }
  }
}
```

Chapter 6

Conclusion and Future Work

This thesis shows the design of the Pyrope language and the implementation of a compiler that partially realizes the language's capability. As demonstrated in the thesis, Pyrope is superior to the mainstream language, SystemVerilog, in many aspects. Furthermore, The language allows the user to extend the language using the compiler framework. A subset of Pyrope can be correctly compiled with the current compiler implementation in LiveHD. While the implementation of the language is still incomplete, the compiler infrastructure has been significantly improved to facilitate future compiler development.

6.1 Future Work

In the future, both the design of the language and the compiler implementation can be further explored and improved.

For language design, several topics can be explored. First, the current type

system is loosely defined and requires a set of formal type rules to make the language more robust. Second, lambda types with sequantial logics may encode more than just input and output datatypes, but also temporal relations among arguments. For example, a temporal type system may encode the cycle delay of a pipeline, such as `(a#[0], b#[0])->(c#[3])` for a pipeline with a latency of three cycles. However, the temporal relations are usually much more complicated than simple delays between input and output pairs and require a more well-defined temporal type theory. Lastly, the current language only defines a small set of verification constructs for dynamic verification, like logging and testing statements. Constructs like linear-temporal-logic (LTL) properties can be added for formal verification.

For the compiler implementation, the translation pass should be able to convert the complete language specification into LNAST, and the subsequent compiler passes should process the LNAST correctly. There are multiple ongoing projects in LiveHD, which include a physical synthesis pipeline and a more complete constant propagation µPass. Furthermore, the compiler infrastructure for LNAST needs a major update to support parallel and incremental parsing of the language. This change potentially requires an LNAST manager for dispatching parsing tasks and scheduling compiler passes to operate on the new LNASTs. Finally, type inference and checking pass are currently absent and will be needed to correctly synthesize all Pyrope code.

6.2 Final Thoughts

The modern hardware development flow has been evolving over the past decades. Tools and algorithms involved in the flow have been extensively studied and optimized on their performance numbers to the limits. However, there are still many opportunities that lie in optimizing how humans interact with these tools, especially in languages where humans design, verify and synthesize the products. This work explores those opportunities. The design of the Pyrope language focuses on the efficiency of how humans learn and design with the language and how tools process it. On the human end, this work contributes to the revised language specification that is much more expressive and easier to comprehend. On the tool end, this work contributes to a functional Pyrope compilation pipeline with some major improvements to the compiler framework. In combination, this work not only realizes the design and implementation of the Pyrope hardware language but also showcases the process of creating a new hardware language for future language researchers.

Bibliography

[1] Cerebras cs-2. https://f.hubspotusercontent30.net/hubfs/8968533/CS-2%20Data%20Sheet.pdf.

[2] Chisel. https://www.chisel-lang.org/.

[3] Magma. https://github.com/phanrahan/magma.

[4] Open verification library. https://www.accellera.org/activities/working-groups/ovl.

[5] Pymtl3. https://github.com/pymtl/pymtl3.

[6] Pyrtl. https://ucsbarchlab.github.io/PyRTL/.

[7] sv-tests. https://github.com/chipsalliance/sv-tests.

[8] Tree-sitter. https://tree-sitter.github.io/tree-sitter/.

[9] Universal verification methodology. https://www.accellera.org/activities/working-groups/uvm.

[10] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[11] Ieee standard for vhdl language reference manual. *IEEE Std 1076-2019*, pages 1–673, 2019.

[12] Steve Golson and Leah Clark. Language wars in the 21st century: verilog versus vhdl–revisited. *Synopsys Users Group (SNUG)*, 2016.

[13] Intel. Verilog hdl synthesis attributes and directives. `https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/hdl/vlog/vlog_file_dir.htm`.

[14] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[15] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[16] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[17] McKinsey and Company. Semiconductor design and manufacturing: Achieving leading-edge capabilities.

[18] John McNamara and Wilson Snyder. Verilog-mode. `https://veripool.org/verilog-mode/`.

[19] Dmitry Petrashko, Ondřej Lhoták, and Martin Odersky. Miniphases: Compilation using modular and efficient tree transformations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 201–216, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. Treefuser: a framework for analyzing and fusing general recursive tree traversals. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[21] Thomas N. Theis and H.-S. Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science and Engineering*, 19(2):41–50, 2017.

[22] Lenny Truong and Pat Hanrahan. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[23] Sheng-Hong Wang, Rafael Trapani Possignolo, Haven Blake Skinner, and Jose Re-

nau. Livehd: A productive live hardware development flow. *IEEE Micro*, 40(4):67–75, 2020.