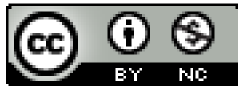
SOURCE-LEVEL DEBUGGING FOR HARDWARE GENERATOR FRAMEWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Keyi Zhang
December 2022

© 2022 by Keyi Zhang. All Rights Reserved.
Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.
http://creativecommons.org/licenses/by-nc/3.0/us/

This dissertation is online at: https://purl.stanford.edu/wb864cp0807

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Clark Barrett

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Zain Asgar

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Acknowledgments

Over my PhD career at Stanford, I have met many amazing and incredible people. Even though I have only had small interactions with most of them, I am deeply humbled by their passion and focus. Without their help, I would not have made this far. It is the accumulation of their kindness and support that helps me stay focused on my PhD. I am also fortunate enough to interact with some of them regularly who guided, taught, and mentored me.

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Mark Horowitz. Mark is always there for me when I need to discuss new research ideas or need insights from a different perspective. I have the tendency of getting distracted easily by random ideas that come to my mind, and frankly speaking, most of them are unrealistic and uninspiring. Mark convinced me that even though "99% of my ideas turn out to be utterly useless, some of them are bound to be incredible if I keep coming up with ideas". He was kind and patient enough to help me filter out ideas, allowed me to narrow down my thesis topic, and helped me concentrate on the project so that I can actually graduate. I am very grateful to him for putting up with my "whimsy". In addition, Mark also encouraged me to work with others and taught me how to become more collaborative, which I will continue to improve upon. He is a mentor and researcher that I forever look up to.

I would also like to express my deepest gratitude to my Doctoral Dissertation Reading Committee and my Orals Committee. I would like to thank Professor Clark Barrett for taking time to read my thesis and to be on my Orals Committee. I am grateful to Professor Priyanka Raina for her input on my research and serving on my Orals Committee. Many thanks to Professor Juan Alonso for agreeing to be my out-of-department Orals Committee Chair. Finally I would like to thank Zain Asgar for serving both on my Reading Committee and Orals Committee. Zain had shown early interests in my work and helped me build the very first prototype of the system. He and I also taught a class together multiple times and I have learned a lot from him.

The endeavor would not have been possible without the support from Apple Inc. I was fortunate enough to receive an Apple Fellowship after encouragement from Seye Ewedemi. In addition to the financial support, Apple also allowed me to intern there for two consecutive summers, during which I have learned a lot and made friends with many brilliant people. I am also grateful for the Apple engineers who provide feedback and suggestions on my research during the quarterly review. I would

also like to thank DARPA for their financial support.

I would also like to extend my sincere thanks to my colleagues and friends. The results of the thesis would not have been there were it not for their help. It is impossible to name everyone here, but I would like to highlight some of them here. Max Strange and Taeyoung Kong were the first adopters of this research project and gave me a lot of useful feedback on how to improve the system. They also used part of the systems in their research project to provide experimental data required for this thesis. Outside work, they are also fun friends/roommates to hang out with. Stephen Richardson introduced me to the AHA project and I really enjoyed deep and constructive conversation with him early in the morning every Wednesday.

Finally, I would be remiss in not mentioning my family, especially my parents and brother, for supporting me throughout my PhD career. Their belief in me has motivated me to continue my research despite setbacks and frustrations. They are always there when I need comfort and company and offer unrequited support. My eternal gratitude goes to them.

Contents

	iv
Acknowledgments	**v**
1 Introduction	**1**
2 Background	**3**
2.1 Hardware generator frameworks and their advantages	3
2.1.1 Execution order and debugging semantics	8
2.1.2 Compiler pass and optimization	9
2.2 You do not control the world	10
2.3 Cost of using hardware generators	11
2.4 How hardware designs are usually debugged	14
2.5 SystemVerilog and simulator verification features	14
2.5.1 How RTL simulator works	15
2.5.2 DPI and VPI	17
2.5.3 SystemVerilog assertions	19
2.6 Prior work on hardware debugging and verification	20
2.7 What to learn from software	21
2.8 Summary	23
3 Building a Debugging System for Hardware Generators	**24**
3.1 Debugger runtime design	25
3.1.1 Watchpoint emulation	36
3.1.2 Assertion handling	36
3.1.3 Reverse debugging	38
3.1.4 Locate generated IP	39
3.1.5 Summary of simulator primitives	39
3.2 Symbol table for hardware generators	40

3.2.1 Mapping source-level symbols to RTL . . . 40
3.2.2 Symbol table primitive design . . . 42
3.3 Debugger frontend interface . . . 44
3.4 Putting everything together . . . 45

4 Hgdb: One Framework to Debug Them All **56**
4.1 Simulator primitive implementation . . . 56
4.2 Instance mapping and concurrent semantics . . . 59
4.3 Waveform replay . . . 61
4.4 Simulation loop and performance optimization . . . 62
4.5 Symbol table design and implementation . . . 64
4.6 Debugger frontend design and implementation . . . 69
4.7 Case study on real world designs . . . 71
4.7.1 Bugs in specialized CGRA for sparse tensor algebra . . . 72
4.7.2 Bugs in the global buffer . . . 74
4.8 Extracting symbol tables from compilers . . . 76
4.8.1 Chisel/Firrtl compiler . . . 77
4.8.2 MLIR/Circt . . . 82
4.8.3 Vitis HLS . . . 84
4.9 Why it is so difficult to produce a symbol table? . . . 86

5 Design for Debugging **89**
5.1 Languages are not created equal . . . 89
5.1.1 Debugging primitives . . . 90
5.1.2 Language-specific implementations for embedded frameworks . . . 92
5.2 Maintaining a symbol table inside the compiler . . . 94
5.2.1 First class debugging support in the compiler . . . 95
5.2.2 Debug invariant transformations . . . 96
5.3 Kratos: an efficient working prototype . . . 98

6 Conclusions and Future Work **104**

Bibliography **108**

Appendices **113**

A hgdb User Manual **114**
A.1 Setting up hgdb . . . 114
A.1.1 Installing hgdb . . . 114
A.1.2 Adding hgdb to the simulators . . . 115

A.1.3 Runtime options . . . 116
A.1.4 Builtin tools . . . 117
A.2 Use a compatible debugger . . . 119
A.2.1 Visual Studio Code . . . 119
A.2.2 Console-based debugger . . . 119
A.3 hgdb runtime protocol . . . 120
A.4 Symbol table format . . . 126
A.5 Internals . . . 127
A.5.1 Simulator interface implementation . . . 127

B kratos user manual **129**
B.1 Features . . . 129
B.1.1 Python AST . . . 129
B.1.2 Finite state machine . . . 129
B.1.3 Compiler passes . . . 131
B.2 Internals . . . 131
B.2.1 Performance optimization . . . 132
B.2.2 AST transformation . . . 135

List of Tables

3.1 Debugging features and SystemVerilog specification compliance for popular commercial and open-source RTL simulators. ✓ indicates incomplete supports. 26

4.1 A list of VPI routines used by hgdb. The explanation is adapted from LRM [1]. Some VPI routines have only partial support, and we thus implemented some heuristic-based workarounds in the shim layer. 58

4.2 A list of Waveform interface primitives. It is designed to work with various waveform formats such as VCD and FSDB. Functions marked with * are unavailable for VCD format. 62

4.3 Example rules for signal flattening. The idea is to use `.` to flatten out any nested data structures for the source-level symbol name. Note that if the data structure is maintained from the source level to RTL, e.g., a source-level array is lowered to a RTL array, no flattening is required. 69

4.4 Source location and symbol tracking support in different generator frameworks at the time of writing. ✓ indicates incomplete support or workarounds required. 88

5.1 A list of debug primitives that are useful for implementing debugging features for hardware generators. 91

5.2 Language support for debug primitives. GSF stands for `get_stack_frames`, GLV stands for `get_local_variables`, GSL stands for `get_source_location`, and GVS stands for `get_variable_scope`. ✓ indicates incomplete support. 95

List of Figures

2.1 HLS memory layout optimization. Block turns the array into a collections of smaller continuous subarrays; cyclic interleaves the memory content; and complete changes the array into a flattened register array. 7
2.2 Simulation execution order is often orthogonal to that of source-level programs, which is visualized using breakpoint triggering order. There will be no confusion if we make it clear to the users. 9
2.3 A simplified simulation loop defined by the SystemVerilog standard [1]. 16

3.1 System design diagram, which is divided into multiple components. Each component communicates with the others via well-defined interfaces. 25
3.2 Performance slowdown by turning on/adding various features to a commercial simulator vendor. No actual breakpoint is inserted during the simulation. The slowdown is normalized to the same simulator without any features turned on. 27
3.3 Compiler passes to insert DPI-based breakpoint calls into the design. The first pass labels each statement that has corresponding source locations and instances. The second pass inserts DPI calls before these statements with proper caller arguments, the combination of which is used to identify the source locations. 28
3.4 Issues with memory writes due to multi-power memory updates. At the posedge of the clock, multi-port memory writes happen simultaneously, whereas memory is updated sequentially in software programming languages. 34
3.5 The write set is a shadow copy of the memory from the previous posedge. As a result, we have full visibility of the memory content before and after the memory update and thus are able to reconstruct the memory state faithfully. 34
3.6 Performance slowdown from clock-edge based breakpoint emulation. No actual breakpoint was inserted during the simulation. The slowdown was normalized to the same simulator without any features turned on. 35

3.7 When a watchpoint is inserted, we first search all the related breakpoint locations that write to the same variable. Then we insert all matched breakpoints into the system. At every posedge of the clock, we check the mapped target variable to see if the value differs. If a mapped RTL value is different from the target value, we trigger the watchpoint breakpoint and update the target value. Note that the value change builds on the existing breakpoint system; that is, the breakpoint must still be active for it to trigger. 37
3.8 Symbol table for example code shown in Listing 3.1. We use `sum.c` as the filename. Notice that four different breakpoints are created at the particular source location. . 41
3.9 We can emulate breakpoints in a loop where we fetch breakpoints in order and check if any of them are active. If so, we reconstruct the breakpoint and send the result to the user. Otherwise we loop back and fetch more breakpoints. The loop terminates when we exhaust all inserted breakpoints, and we then wait for the next posedge of the clock. 47

4.1 Performance improvement after optimizing expression tree data structures. Eight breakpoints were inserted, whose enable condition evaluates to false at runtime. Instead of querying handles during breakpoint evaluation, we tagged the vpiHandle to the expression node when users inserted a breakpoint. We also compared the performance where multiple-processing was introduced. Time was normalized to simulation time without any modification. 64
4.2 Simulation overhead breakdown with eight breakpoints inserted. The time is normalized to simulation time without any modification. Note that more than half of the overhead os from getting values from the simulator. The miscellaneous time is calculated by subtracting the overhead of known categories from the total observed wall clock overhead. Most of this time is due to CPU context switching and other tasks. 65
4.3 Simplified schema diagram for SQLite-based symbol table. Arrows in the figure illustrate relations, which can be used to improve the search performance and guarantee data integrity. 66
4.4 How `get_scope_info` is implemented over a network channel. The underlying transport layer should not matter, and hgdb supports both TCP and WebSocket. 70
4.5 Debuggers provided by hgdb. 71
4.6 CGRA architecture diagram. Processing element (PE) and memory (MEM) tiles are connected through a mesh-like interconnect. It also has a cache block called global buffer that controls configuration and streams in/out data. Blocks written in kratos are shown in □ 72

4.7 Bug fix for the sparse controller FSM to ensure we do not double count the block allocation command. We added additional conditions to control the FIFO pop signals (shown in green background with leading +) 74
4.8 Architecture diagram for the global buffer. 75
4.9 Illustration of debugging global buffer at source-level and corresponding bug fix. . . 76
4.10 Simplified illustration of how state information is associated with source code and IR. 85

5.1 Workarounds for handling variables declared outside the framework. Available breakpoints are shown in ❶ and ❷. 94
5.2 Illustration of how to preserve symbol mapping while inlining instances. 97
5.3 Illustration of how to preserve symbol mapping when removing pure combinational dead code. 98
5.4 Illustration of how to preserve symbol mapping when removing sequential dead code. Function SC returns the shadow copy of an expression. 98
5.5 AST transformation in kratos that keeps track of the source location. 100

Chapter 1

Introduction

Historically, hardware was specified manually in hardware description languages (HDLs) such as SystemVerilog and VHSIC Hardware Description Language (VHDL). Although these languages have evolved over the years, it is increasingly difficult for them to keep up with the trend of hardware customization and specialization, where a family of hardware is created with minor variants for different use cases. The parameterization and template language features offered in these HDLs can no longer satisfy the complex nature of hardware design, increasing the engineering cost [2]. In response to this problem, Ofer Shacham *et al.* [3] argued that we should rethink the traditional hardware design: instead of building a new customized chip for every application, we should reuse the design process to generate multiple new chips, which is done through a set of frameworks called hardware generators.

Following the new design paradigm shift, designers have seen various hardware generator frameworks developed and used in practice over the past decade. Many frameworks are embedded into other high-level programming languages, i.e., embedded domain-specific languages (eDSL), such as magma [4] (Python) and Chisel [5] (Scala). This kind of framework requires designers to write a program in the eDSL, which later is executed or compiled to produce Register-Transfer Level (RTL). In these systems, the designers usually need to fully specify the cycle-level timing behavior of the hardware in the eDSL. This constraint is relaxed in High-Level Synthesis (HLS), another form of hardware generator, where designers specify only the high-level logic and scheduling directives: the compiler is responsible for producing a valid schedule given timing and resource constraints.

Unfortunately, however, many features introduced in the frameworks to increase design productivity and synthesis quality make the generated RTL difficult for the designers to read and to understand. This is because the compiler transformations and optimizations often obfuscate the code. Since most intellectual property (IP) blocks are transferred at the RTL level, electronic design automation (EDA) engineers focus on optimizing RTL-level simulation and verification. As a result, any system-level verification has to be done at the RTL level, and hardware designers have to debug

the system-level issues with RTL, which contains the generated code. Consequently, this generated code makes the generator frameworks difficult to debug [6, 7, 8].

While it might be tempting to fix the debugging problem by removing hardware generators–after all, more time is spent on validation than design [9]–this would be a bad decision. In many eDSL-focused generator frameworks, the designer can use many host language features, such as object-oriented programming, functional programming, and complex data types, which offer stronger type safety guarantee. Because program execution involves many sequential stages, designers can instead use software languages to control the hardware generation process. These software programming techniques overcome the limitations of traditional HDLs and improve design productivity [5, 10, 6, 11].

To continue to leverage these advantages, the work presented in this thesis demonstrates how we can apply source-level software debugging techniques to create a powerful debugging system for hardware generators. Software debugging systems are extremely useful for this purpose because they deal with problems in software similar to that discussed above, namely, when the execution environment, such as CPU or virtual machine instructions, is drastically different from the source code.

This work is described in the following chapters. Chapter 2 first discusses the techniques used in hardware generator frameworks and how they affect the final RTL and debugging, and then reviews prior work on debugging systems. Chapter 3 proposes a component-based source-level hardware generator debugging infrastructure. Each component has a set of well-defined primitives and an interface to maximize compatibility among different simulators and frameworks. This chapter also presents algorithms by which critical debugging features, such as breakpoints and assertions, can be properly implemented. Chapter 4 presents hgdb, a system that realizes these primitives and offers efficient techniques to implement these components. It also discusses algorithms by which we can produce debugging collateral from different frameworks and then showcases how hgdb can be used to identify real-world design bugs. Based on our development experience with these frameworks, Chapter 5 looks ahead and provides guidelines for generator frameworks to improve the debugging experience. It also shows how a prototype framework called kratos is implemented as a debuggable hardware generator framework.

Chapter 2

Background

Unlike traditional hardware design, where the logic is specified explicitly by the designers and the simulator takes in the description as is, hardware generators allow users to specify the logic in a high-level programming language and produce the desired RTL. This process is usually done by introducing an intermediate representation (IR), where the high-level programming languages get lowered to the IR via compiler transformations and/or code execution before RTL code generation.

Before discussing our approach to bringing source-level debugging to hardware generators, we first need to better understand the benefits of and effects on traditional hardware debugging of hardware generator frameworks and their compilation and generation processes. This chapter first discusses this in depth, and then examines existing software debugging systems and well-established RTL simulators to find prior work upon which we drew to build our own debugging infrastructure.

2.1 Hardware generator frameworks and their advantages

Generally speaking, hardware generators can be classified into one of the following three categories: text-based template frameworks, embedded domain-specific languages (eDSL), and high-level synthesis (HLS). These frameworks were developed at different times and offer different benefits to designers.

Text-based template frameworks usually use a scripting language such as Perl and Python to perform string-based substitution. The scripting language functions as a preprocessor, which allows a designer to mix RTL logic with scripting languages, similar to Jinja-based templates in web designs [12]. During execution, the scripts are statically evaluated and produce the final RTL. Because the execution has little to do with the underlying RTL, such a framework does not need to understand the RTL, nor is it capable of altering the design through compiler optimizations. Genesis2 is one example of a text-based template framework; indeed, it pioneered the concept of hardware generators [13]. Listing 2.1 and Listing 2.2 show how code is transformed into final RTL

through Perl-based elaboration. The code sections nested inside SystemVerilog comments are in fact Perl code that uses the specialized token `//;`. During program execution, Genesis2 identifies these code blocks and executes the script while treating the RTL logic as plain text. For instance, `` in_`$` `` is nested inside a Perl loop and gets instantiated out `$num_tracks` (4 in the example) times with `` `$i` `` replaced with a concrete number in the final output.

```
//; my $width  = parameter(Name=>'width', val=> 16, doc=>'Bus width for CB');
//; my $num_tracks = parameter(Name=>'num_tracks', val=> 10,
//;                            doc=>'num of tracks for CB');
//; my $feedthrough_outputs = parameter(Name=>'feedthrough_outputs',
//;                                     val=> "1111101111",
//;                                     doc=>'binary vector muxing');
//; my @feedthrough = split('',$feedthrough_outputs);

module `mname` (
clk, reset,
//; for(my $i=0; $i<$num_tracks; $i++) {
//;  if ($feedthrough[($i%$num_tracks)]==1) {
in_`$i`,
//;   }
//; }
out,
config_addr,
config_data,
config_en,
read_data
);
```

Listing 2.1: Genesis2 code that describes the module input and output ports for a configurable connection box. The commented out code section is the inlined Perl script that controls the code generation. Notice that the module name and variable `` in_`$` `` are parameterized.

Embedded domain-specific languages are a class of programming languages that use the host environment, typically a high-level programming language such as Scala and Python, to define new programming semantics for a particular domain. Unlike text-based template frameworks, eDSL defines a set of hardware primitives, such as wires and registers, and designers have to write software code to construct the hardware using these primitives. The host language runtime then executes the software code and produces some form of IR, which is further transformed and optimized before turning into RTL.

Chisel is one of the most popular eDSL-based hardware generator frameworks, and is based on Scala [5]. Listing 2.3 shows a snippet of Chisel code that computes a lookup table for a sin function. Because the code itself is a program, upon execution, the program will compute the content of the lookup table and then use it to initialize the read-only memory (ROM) in the generated RTL. If the

```
module cb_unq1 (
clk, reset,
in_0,
in_1,
in_2,
in_3,
out,
config_addr,
config_data,
config_en,
read_data
);
```

Listing 2.2: SystemVerilog code that is generated from Listing2.1 when `\$num_tracks` is set to 4. After the code generation, the parameterized variable `in_`$i`` is instantiated multiple times with different names. The module is also set to `cb_unq1` to ensure a unique module definition given the parameters.

same ROM were designed in conventional SystemVerilog, the designer would first need to compute the table in a separate program and then copy and paste the values in code, which is error-prone and brittle since the size of ROM can change during the design iteration, and designers would have to redo the process repeatedly and manually.

Listing 2.4 shows another example of Chisel code that uses Scala's functional language feature. In the snippet, we want to ensure that all the input values are even. Scala allows users to write map-reduce functions where each element in the array gets mapped to a new value and then combined to produce a single value. We first map the inputs into a boolean array by checking the modulo result, then we reduce the boolean array into a single boolean, all in a single line. This programming feature makes the code more expressive and significantly reduces the number of lines required to implement the same logic in conventional RTL. In addition, using functional programming makes the code snippet work with any input array size since the host language (Scala in this case) tracks variable types automatically.

Unlike eDSL-based hardware generators, HLS abstracts away many micro-architecture aspects of hardware design, such as resource allocation and memory. It allows designers to focus on the actual functional specification. Instead of program execution, HLS tools parse the program directly using the compiler before turning the code into an IR through compilation. Then the HLS compiler transforms the IR into a control flow graph. HLS also builds a physical compute engine with different components such as memory devices and multipliers based on the resource and scheduling constraints provided by the users. The last step is the binding process, where the HLS compiler schedules the control flow graph nodes on to the physical engine such that each component can perform a different compute task at each time unit. The scheduling is typically done through constructing a finite state

```
import chisel3._

val Pi = math.Pi
def sinTable(amp: Double, n: Int) = {
  val times =
    (0 until n).map(i => (i*2*Pi)/(n.toDouble-1) - Pi)
  val inits =
    times.map(t => Math.round(amp * math.sin(t)).asSInt(32.W))
  VecInit(inits)
}
```

Listing 2.3: Chisel code to compute the content of a read only memory (ROM). Note that the values are computed during Scala code execution, and the RTL only contains computed values. Code snippet is taken from Chisel/Firrtl documentation [14].

```
13 class InputEven extends Module {
14   val io = IO(new Bundle {
15     val inputs = Input(Vec(4, UInt(32.W)))
16     val output = Output(Bool())
17   })
18   // Chisel one liner to check all input values are even
19   io.output := io.inputs.map(i => (i % 2.U(32.W)) === 0.U(32.W)).reduce((a, b) =>
   ↪  a & b)
20 }
```

Listing 2.4: A Chisel one-liner to compute if all the input values are even. Note that we use Scala's functional programming features such as map and reduce to make the code easier to read.

machine (FSM) that dictates the specific task for each hardware component at any given time unit. Note that adjusting the hardware resources and scheduling does not change the high-level functional logic specification, thus enabling rapid design space exploration. In addition, the same software unit tests that are used to verify the software program can also be used to test the generated RTL, allowing "test reuse."

Listing 2.5 shows a high-level C code that squares the input array. Note that there is no instantiation of any particular memory device: the user simply codes up memory size and access logic. Many field-programmable gate array (FPGA) HLS compilers are target-aware and can instantiate platform-specific BRAMs given a generic memory description. Users can also specify different memory layouts such as sequential or interleaved, independent of the implementation logic to explore performance trade-offs, as shown in Figure 2.1. If the same memory optimization is done in traditional HDLs, designers have to carefully consider the availability of underlying memory technology and adjust the logic accordingly. The designers also have to manually specify the memory access

logic to serialize the memory read. Such a manual process can be tedious and error prone if the memory access is irregular and requires a great deal of engineering effort. Furthermore, this advantage of HLS tools significantly reduces the effort required to migrate the same design to different platforms.

Another benefit of using HLS with C++ is to utilize the languages' rich meta-programming features, such as `template` and `constexpr`, by virtue of using a C++ compiler. During the program compilation, template values can be statically elaborated and different logic will be instantiated or simplified, allowing flexible parameterization similar to that of eDSL (C++ templates are Turing-complete [15]).

```
1  #define N 64
2
3  void square(short input[N], short output[N]) {
4      int i;
5      for (i = 0; i < N; i++) {
6          output[i] = input[i] * input[i];
7      }
8  }
```

Listing 2.5: C code squares an one-dimensional array.

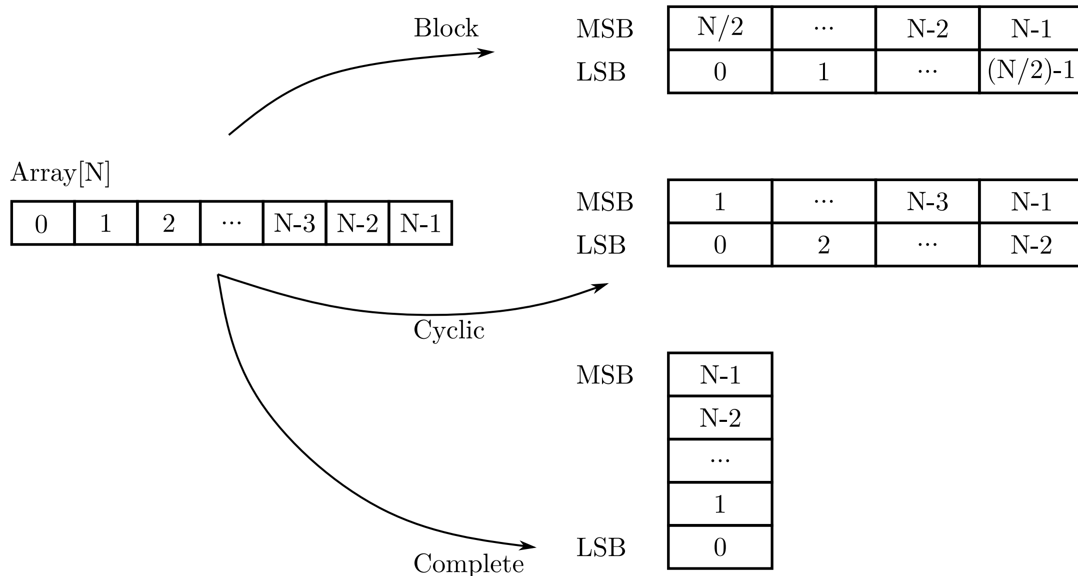


Figure 2.1: HLS memory layout optimization. Block turns the array into a collections of smaller continuous subarrays; cyclic interleaves the memory content; and complete changes the array into a flattened register array.

Since both eDSL and HLS employ some form of IRs and use a compiler to produce the final

RTL, we can design compiler passes to understand the hardware logic and extract debugging information. As a result, these frameworks are much easier to work with compared to text-based template frameworks. We focus on these two types of hardware generators in this thesis.

2.1.1 Execution order and debugging semantics

Generator frameworks typically employ multi-stage execution. Designers write a program and the execution or compilation of that program generates the RTL. As a result, the hardware construction obeys the semantics of the software programming languages. However, the RTL simulator uses different execution semantics. Most software languages use a sequential execution order, which implies that statements specified later in the same scope get executed later. Hardware, on the other hand, is intrinsically concurrent. For instance, thousands of registers can update their values at exactly the same time (if we disregard the clock skew). In addition, most testbenches are set up based on value changes, i.e., events, such as performing a group of actions when a signal goes high. Because of this intrinsic concurrency, RTL simulators are event-driven and employ some form of event loop to simulate the concurrent nature of hardware circuits. The event execution order is vendor dependent, but typically follows the data flow to avoid unnecessary loop iteration. In other words, the simulation order is different from the hardware construction order defined in the source language. A direct consequence of this is that a hardware block that is declared before a given block can execute after in the simulation.

Although such duality of execution orders may lead to confusion for designers, we think this is not an issue if the debugging framework can help designers distinguish these two execution orders. Fundamentally, these two orders are orthogonal: the source-level execution order specifies the sequence of hardware construction and the simulation order specifies the value update sequence. The former is syntactical based on the source code and the latter is temporal based on the simulation time. Given any time, i.e., when the simulation is paused for debugging, we present the simulated hardware as a pure software program with the host language execution semantics. If the program is written in a language that uses a sequential execution model, the simulated hardware should be debugged as a sequential program, even though it is parallel in reality. Stepping through code reflects the source-level order and the debugger only advances to the next time once we exhaust the debuggable locations. Figure 2.2 shows how we can simultaneously support two execution orders without confusing users. At time 1, users can set breakpoints and have them trigger in syntactical order; at time 2, they can have a different set of breakpoints, and they trigger in software semantics too.

Since during debugging we have access only to the simulator, we can only natively support the simulation order, i.e., debug as the time advances. In order to faithfully render the hardware in the source-level execution order, we need help from the framework to produce a symbol table that informs us about the source-level semantics. This is discussed in greater detail in Chapter 3.

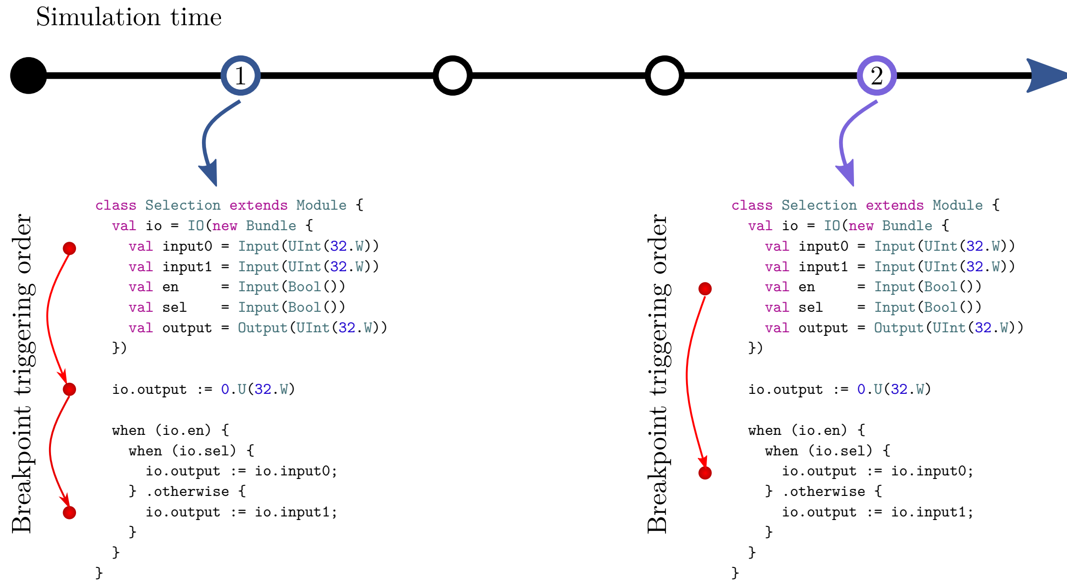


Figure 2.2: Simulation execution order is often orthogonal to that of source-level programs, which is visualized using breakpoint triggering order. There will be no confusion if we make it clear to the users.

2.1.2 Compiler pass and optimization

Because the hardware construction is now through a software program, we can apply standard compiler techniques to assist hardware construction or even optimize the generated hardware. This is accomplished by using an IR to represent the hardware constructed by the program. The IR can be close to SystemVerilog, as in Firrtl and Circt, or close to IR for software, such as Vitis HLS, which uses LLVM.

This separation of IR and frontend language design enables a more flexible frontend design and allows generator compiler developers to focus on the optimization and analysis. To do so, the IR can have different levels, such as Firrtl [16] used by Chisel and MLIR [17] used by Circt. The high-level IR can closely relate to the frontend language where the primitives can be high-level constructs such as FIFOs and ROMs. The low-level IR can be close to RTL, where the primitives are wires and flip-flops. The high-level IR makes the frontend tool easier to implement because it can express high-level logic without going through the low-level implementation details. The compiler can then transform the high-level IR into a low level one, a process called lowering. During the lowering, high-level constructs are replaced with more concrete low-level primitives. For instance, SRAM/BRAM and wrapper logic such as flip-flops are instantiated when lowering a FIFO primitive. The same process can be used to implement features described in Figure 2.1 in HLS tools.

In addition to lowering, compiler passes can be naturally used for hardware optimization. A common pass is dead code elimination, where dead logic is removed if it does not contribute to the

overall data output. This optimization is difficult to implement by hand since it requires additional information, such as test bench coverage reports, to discover the unnecessary logic. Another popular pass is Static-Single-Assignment (SSA), which forms the basis for many other optimizations such as constant propagation, loop unrolling, and dead code elimination. In addition, because the IR is typically higher than RTL and thus closer to the original source code, the compiler can make optimization decisions better than synthesis tools, which typically operate at RTL-level. Many optimization passes are generic and can be applied to all designs generated through the framework, allowing code reuse. Designers can also write design-specific passes to further optimize their designs using domain-specific knowledge.

As mentioned earlier, most HLS tools generate hardware based on a functional description, which is scheduled into underlying hardware. The functional description does not carry explicit timing information: it is realized through compiler transformations/operation scheduling passes and implemented in an FSM. The scheduling is usually guided by the scheduling directives provided by the designers. Complex operations such as memory fetch and dependent control logic are divided into multiple states where the hardware advances states at each clock edge. Because the HLS compiler implements a function specification and controls the overall system timing, it supports a form of optimization called automatic pipelining, which inserts pipeline registers into the design to increase the overall clock frequency. Unlike conventional hardware compiler optimization, where the end result is strictly equivalent, pipelining alters the exact timing for data waves without changing the overall function. Pipelining parameters are often provided by the users, and the compiler tries its best to meet the pipelining demand while satisfying resource constraints.

Although applying software compiler techniques to hardware generation improves the design productivity and final synthesis quality, it also introduces many issues, such as debugging, code coverage closure, and physical design. This thesis focuses on the debugging problem.

2.2 You do not control the world

While one might assume that designers would simulate and debug their designs in their source languages, three factors usually force designers to simulate the generated hardware at the RTL level: the usage of external IP blocks, the usage of popular testing environments, and the performance advantage of the RTL-based simulators.

When designing a large-scale hardware system, designers often reuse existing IP blocks to reduce turnaround time and engineering cost. This is more true in modern system-on-chip (SoC) design, where the sheer complexity forbids the hardware team from designing all components from scratch [2]. Using verified and configurable IP cores lets the designers focus on a subset of the SoC system where the innovation happens. However, these IP blocks are typically exchanged as SystemVerilog/VHDL and potentially encrypted, forcing designers to simulate the entire system at

the RTL level.

Because SystemVerilog and VHDL dominate the EDA market, most verification tools and frameworks build on top of these two languages. For instance, UVM is the de facto testbench library based on SystemVerilog and supported by all commercial RTL simulators [18]. It defines the boundary between a testing target and a testing environment and allows the test environment to be reused across different hardware designs. As a result, if the designers wish to use UVM to validate their design, they have to simulate their generated IP block at RTL level.

Although some framework developers provide verification tools within their own frameworks, these "home-made" simulators are orders of magnitude slower than commercial simulators and Verilator [19]. In addition, many framework-specific simulators support only a small subset of SystemVerilog semantics (typically synthesizable only) [20, 6]. Since existing simulators have nearly perfected their implementation on cycle-based simulation, it is difficult for hardware generator framework developers to compete without investing significant amounts of engineering efforts. As a result, designers often simulate the entire generated IP block in SystemVerilog since the RTL simulator typically runs faster. In addition, not supporting the powerful verification semantics offered by SystemVerilog fetters the verification engineers if the entire design is simulated using framework's simulators [8]. As a result, SystemVerilog-based verification is always preferred.

2.3 Cost of using hardware generators

Unfortunately, most generator frameworks do not produce readable RTL. Although the generator framework improves the frontend design productivity, it also makes the generated hardware challenging to understand and debug. Before addressing this difficulty, we need to understand why many hardware generators produce obfuscated RTL that has lost high-level design information, and why it is an issue for hardware debugging.

Generally speaking, the more software-centric features introduced in the frontend language, the greater the semantic gap that exists between the frontend language and generated RTL. This is because RTL is more rigid and less expressive than software programming languages. This gap is reflected in the difference between the frontend language and the IR, and between the IR and the RTL. These differences require lowering passes to transform the high-level semantics to a low-level RTL equivalent. These passes are necessary to produce valid RTL but also introduce unavoidable transformation artifacts. To illustrate this unfortunate consequence, let us revisit the code snippet in Listing 2.4, where a one-liner is used to ensure all input values are even. Listing 2.6 shows the generated RTL. Because SystemVerilog is not functional, we need some temporal variables to hold the partial result when performing map and reduce (Chisel uses the naming scheme `GEN_` for generated variables.) As a result, the generated RTL has many signals that make it difficult to discern the original functionality, which is critical for debugging. Even if the compiler eliminates

the temporal variables by inlining the expression tree, it is still difficult to debug since users cannot read out partial results during simulation if there is a bug in the map reduce logic. Although Chisel includes the source location in the generated RTL, e.g., `InputOdd.scala 19:38`, it offers little help since all of the code shares the same location.

```
module InputOdd(
  input         clock,
  input         reset,
  input  [31:0] io_inputs_0,
  input  [31:0] io_inputs_1,
  input  [31:0] io_inputs_2,
  input  [31:0] io_inputs_3,
  output        io_output
);
  wire [31:0] _GEN_0 = io_inputs_0 % 32'h2; // @[InputOdd.scala 19:38]
  wire [31:0] _io_output_T = _GEN_0[31:0]; // @[InputOdd.scala 19:38]
  wire [31:0] _GEN_1 = io_inputs_1 % 32'h2; // @[InputOdd.scala 19:38]
  wire [31:0] _io_output_T_2 = _GEN_1[31:0]; // @[InputOdd.scala 19:38]
  wire [31:0] _GEN_2 = io_inputs_2 % 32'h2; // @[InputOdd.scala 19:38]
  wire [31:0] _io_output_T_4 = _GEN_2[31:0]; // @[InputOdd.scala 19:38]
  wire [31:0] _GEN_3 = io_inputs_3 % 32'h2; // @[InputOdd.scala 19:38]
  wire [31:0] _io_output_T_6 = _GEN_3[31:0]; // @[InputOdd.scala 19:38]
  assign io_output = _io_output_T == 32'h0 & _io_output_T_2 == 32'h0 &
↪  _io_output_T_4 == 32'h0 & _io_output_T_6 == 32'h0
    ; // @[InputOdd.scala 19:85]
endmodule
```

Listing 2.6: Generated RTL for the one-liner that computes if all the even inputs are even. Default Firrtl passes are applied to the design. Note that although there are source locations in the comments, the generated code structure is very different from the original source code in Listing 2.4.

Since HLS schedules a high-level function onto a fixed-resource hardware, the transformation needs to generate extra logic to interface with the target hardware device. Listing 2.7 shows less than 1% of the RTL code that is generated from the array accessing logic in Listing 2.5. The memory fetch schedules are determined by the states. For example, `gmem_ARLEN` will have a proper value if `ap_CS_fsm_state73` or `ap_CS_fsm_state72` is high. In addition to scheduling, these compilers also need to generate boilerplate code to interface with bus protocols such as Advanced eXtensible Interface (AXI) to use the memory. This bootstrap code, e.g., `gmem_ARID` and `gmem_ARLEN` in Listing 2.7, has nothing to do with the user logic and only makes the code more difficult to comprehend.

Another source of code "obscurity" comes from compiler optimizations. Taking common subexpression elimination (CSE) as an example, the pass allows the compiler to search for instances of identical expressions and replace them with a single variable holding the computed value. It is commonly used to reduce the number of expensive operations used in the RTL, such as a divider

```
always @ (*) begin
    if (((1'b1 == ap_CS_fsm_state73) | (1'b1 == ap_CS_fsm_state72))) begin
        gmem_ARID = grp_square_Pipeline_VITIS_LOOP_6_1_fu_87_m_axi_gmem_ARID;
    end else begin
        gmem_ARID = 1'd0;
    end
end

always @ (*) begin
    if (((gmem_ARREADY == 1'b1) & (1'b1 == ap_CS_fsm_state2))) begin
        gmem_ARLEN = 32'd2;
    end else if (((1'b1 == ap_CS_fsm_state73) | (1'b1 == ap_CS_fsm_state72)))
 ↪  begin
        gmem_ARLEN = grp_square_Pipeline_VITIS_LOOP_6_1_fu_87_m_axi_gmem_ARLEN;
    end else begin
        gmem_ARLEN = 'bx;
    end
end
```

Listing 2.7: Generated RTL from Listing 2.5 using Vitis HLS. Only less than 1% of the code is shown here.

or a multiplier. As is the case with lowering functional programming primitives, CSE also needs to introduce a temporal variable to hold the subexpression. In addition, after the expression replacement, it is difficult for designers to trace back the expression logic as it introduces one extra level of indirection. Pipelining is another code optimization that makes source-level functionality difficult to locate in the generated hardware. It breaks the user's logic and state into multiple stages that execute at the same time, again introducing new logic and state with generated names.

Some compiler optimizations may even change the assumption of logical resource allocation presented in the source code. For instance, some HLS tools reduce the memory resources, either the size or number of ports, if it is safe to do so. Given a large register file, if the compiler detects sequential read and write, or the user specifies particular memory optimization directives, it can replace the array with a much smaller memory device with limited ports while streaming data in and out. As a result, the hardware waveform holds only a partial slice of the logical memory, which slides through the memory. This data slicing optimization makes it difficult for designers to reason about bugs. When a memory corruption happens, the designers have to locate the partial view held in the physical memory within the logical view of an array.

2.4 How hardware designs are usually debugged

When the hardware is manually written in RTL, designers often use waveforms for debugging because the waveform retains value changes for all the signals specified by the RTL. If the design is small, they can directly trace signal changes in the waveform produced from the simulation. If the design is large, such as an SoC, the designers usually use trackers or monitors to first localize the problem, then use waveforms once the problem scale becomes more manageable. In other words, using waveforms produced from the simulator is the de facto way to debug hardware. Without any doubt, this method requires the designers to locate the critical signals of interest in the RTL. Therefore, the readability of the RTL directly impacts verification productivity.

Manually tracing signals using waveforms can become unmanageable, and waveforms are often presented outside the source code. To address this issue, some more advanced debuggers, such as Cadence SimVision and Synopsys Verdi [21, 22], have been developed to translate waveform signals into source-level symbols, thus making signal tracing much easier. These tools have a tight integration with the simulators and allow designers to interactively debug the running simulation using debugging features such as breakpoints and watchpoints. They also offer causal analysis that allows users to trace the signal drivers in the RTL code. In addition to RTL-level debugging, the commercial debugging tools also offer transaction-level debugging. Once users annotate the source code about the transaction information, e.g., the packet payload and transaction boundary, the tools can visualize the transaction across the design in lieu of low-level waveforms. Overall, these features significantly reduce the time to localize the bug in the RTL code. Unfortunately, these features work only with conventional RTL code, e.g., handwritten SystemVerilog. Since the generated RTL is difficult to read, designers cannot use the tools directly.

Another useful set of tools for verification is logical equivalence checkers, such as Cadence Conformal and Synopsys Formality [23]. The checkers can formally prove the equivalence of two hardware designs, or identify the difference between them, if any, without going through the traditional functional simulation. Recent advances in formal logic allow the checker to take timing into account and examine the equivalence of two hardware designs over multiple clock cycles. The equivalence checkers significantly reduce the time to verify small modifications of the design and can also be used to prove the correctness of hardware transformation.

2.5 SystemVerilog and simulator verification features

In this section, we first explain how a RTL simulator simulates the hardware description in SystemVerilog; then we will cover advanced SystemVerilog features that are useful for us to build debugging tools.

2.5.1 How RTL simulator works

Most RTL simulators follow the SystemVerilog standard to model physical hardware. As a result, there are several properties that the RTL simulators should support in order to simulate the hardware in a realistic and deterministic manner:

- Four-state logic
  Every single "bit" is represented as a 4 state value: in addition to the traditional 0 and 1, we also have `x` for unknown value and `z` for high impedance. These two new states are introduced to faithfully represent the hardware response for different physical situations. The simulator uses `x` to model uninitialized memory and any logic outputs that depend on that memory's output. This value means the logical value could be either 0 or 1 and is essential to validate that a design will come out of reset and always work. For a net that is driven by multiple drivers, such as tri-state buffers, `z` is used. `z` indicates that this buffer is currently not driving the output, so it will not conflict with the other gates. The SystemVerilog standard also extends the Boolean logic to work with four-state values. For instance, 0 `AND x` is 0. For control logic such as `if`, SystemVerilog optimistically treats `x` as 0.

- Combinational and sequential logic
  Combinational logic is used to model hardware component thats do not involve states: output is computed immediately (a tiny delay in the physical world) whenever the input changes. Sequential logic is used to model memory devices such as flip-flops and memories. These components update their value on a clock edge, typically the positive edge (posedge) of the clock. To properly simulate this behavior, SystemVerilog introduces a semantics called non-blocking assignment (NBA), using the `<=` operator. The right hand side of the expression is evaluated at the edge of a signal change (usually positive edge of a clock signal) and the end result is stored in the simulator's memory. After every NBA expression is evaluated, the simulator then proceeds to update the left hand side.

- Event-driven execution
  Whenever a net or register changes its value, an event occurs and the simulator needs to ensure that the value change event is properly propagated through the hardware system. However, because hardware is concurrent and most simulators are single-threaded, the simulator must serialize the events in some order in order to process the events. To ensure a consistent and deterministic simulation behavior, the SystemVerilog specification defines a zero-delay simulation model by dividing the computation into different simulation regions, as shown in Figure 2.3 (simplified). After each simulation region, the simulator loops back to the beginning if there are any changes detected. For instance, after the execution of the NBA region, i.e., non-blocking assignments, if the simulator detects that some net values need to be updated, it will loop back to the active region to compute the value updates. This looping semantics

implies that the same code block in the active region might be executed multiple times. Once the simulator finishes the postponed region, the simulator enters a synchronized stage, i.e., stable, where no further updates are scheduled for this time slot. The simulator then moves to the next time slot. For a synchronous design, simulation time only advances when the clock toggles. This implies that clock value changes are the first set of signals the simulator evaluates after advancing the simulation time. In other words, at the clock edges, the simulation state is synchronized and can be used to infer current design state. We exploit this phenomenon in Section 3.1.

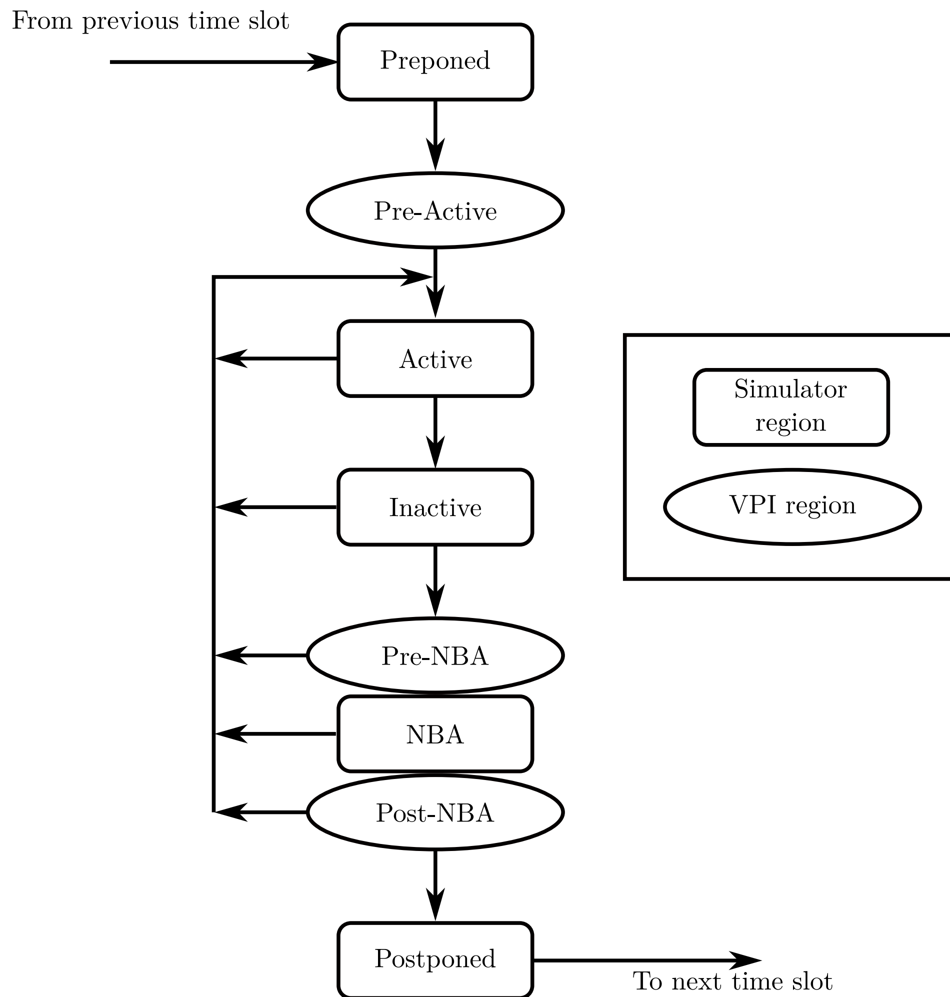


Figure 2.3: A simplified simulation loop defined by the SystemVerilog standard [1].

```c
int add(int a, int b) {
    return a + b;
}
```

```systemverilog
import "DPI-C" function int add(input int  a,
                                input int  b);

module test_add;
int a, b, c;
initial c = add(a, b);

endmodule
```

Listing 2.8: A simple DPI example. We first define an `add` function in C (left), then import it to SystemVerilog via DPI and use it to compute value `c`.

2.5.2 DPI and VPI

Many useful methods have been built to interact with SystemVerilog simulators. Two of the most commonly used software-oriented frameworks are Direct Programming Interface (DPI) and Verilog Procedural Interface (VPI), both of which are specified by the SystemVerilog language reference manual (LRM) [1].

DPI is an interface that allows foreign functions written in C/C++ to be called inside a simulator as if the function is written in SystemVerilog. It is commonly used as a bridge to bring high-level functional models written in C/C++ to the SystemVerilog verification environment. Another advantage of DPI is that it is supported by virtually every RTL simulator, making it easy for engineers to develop custom functions that interface with the simulator. Listing 2.8 shows how we can import a simple C function into SystemVerilog and call it as if it were a native function. All we need to do is to declare the DPI function prototype at the very top. Note that DPI only supports 2-state values, so conversion occurs if a four-state value is used with DPI.

VPI is also part of SystemVerilog standard. It defines a set of functions, called VPI routines, inside a C header file. The application uses the functions directly and then is compiled as a shared library. In order to get recognized by the simulator, the application also needs to register its initialization functions, either through a binary function table called `vlog_startup_routines` (specified by the LRM), or through an input file to the simulator. Before the simulation starts, the simulator loads up the shared library and initializes the application. Unlike DPI, where the simulator only provides a compatible layer to translate between SystemVerilog primitive types and C primitive types, VPI defines a complete model for all SystemVerilog constructs. As a result, applications can use VPI to access the design during simulation, such as querying the design hierarchy and obtaining signal types and values. SystemVerilog also defines a dedicated VPI region in the simulation model and allows users to place callbacks on various simulation events, as shown in Figure 2.3. An example is shown in Listing 2.9 (written in C) and Listing 2.10 (SystemVerilog test environment). In the C code, we first register our initialization function in the table. We then insert a custom callback to the simulation event `cbStartOfSimulation`, which allows the simulator to call the callback when the simulation starts. The callback function proceeds to query the design hierarchy and print out

instance definitions.

```c
#include <stdio.h>
#include <vpi_user.h>

PLI_INT32 print_hierarchy(p_cb_data cb_data) {
    // this function gets called when the simulator starts simulation
    vpiHandle it, inst, top;
    // use NULL to get top
    it = vpi_iterate(vpiModule, NULL);
    top = vpi_scan(it);
    // loop through top
    it = vpi_iterate(vpiModule, top);
    while ((inst = vpi_scan(it)) != NULL) {
        const char *def_name = vpi_get_str(vpiDefName, inst);
        vpi_printf("module: %s ", def_name);
        const char *inst_name = vpi_get_str(vpiName, inst);
        vpi_printf("name: %s\n", inst_name);
    }
    // we will see the following printout after the function runs
    // module: design1 name: inst1
    // module: design1 name: inst2
    // module: design2 name: inst3
    // module: design2 name: inst4
    return 0;
}

void initialize_callback() {
    // this function gets called the moment the VPI library is loaded
    // into the simulator
    s_cb_data cb_data = {.reason = cbStartOfSimulation,
                         .cb_rtn = &print_hierarchy,
                         .obj = NULL,
                         .time = NULL,
                         .value = NULL,
                         .user_data = NULL};
    vpi_register_cb(&cb_data);
}

// store the initialize function in a predefined VPI symbol
void (*vlog_startup_routines[])() = {initialize_callback, NULL};
```

Listing 2.9: VPI example to illustrate call back and module hierarchy query. Note that we insert a custom callback to the event when the simulation starts using the callback type `cbStartOfSimulation`. Once the custom callback is called, we query the design hierarchy from the very top and print out the module names of any instance defined under the top module.

```
module design1();
endmodule

module design2();
endmodule

module top;

design1 inst1();
design1 inst2();
design2 inst3();
design2 inst4();

endmodule
```

Listing 2.10: Example design for the VPI code in Listing 2.9 to query during simulation.

Since VPI only defines the function signatures, the simulator vendors need to provide their own implementation. As a result, leveraging simulators' internal implementation makes VPI applications portable across different simulators on the same type of operating system. Unfortunately, the support for VPI varies across different simulators due to the sheer complexity of the specification. We discuss a software engineering solution to this issue in Section 4.1.

2.5.3 SystemVerilog assertions

SystemVerilog assertions (SVA) is a standard that formally specifies properties of underlying hardware systems. During RTL simulation, the simulator checks the assertion condition, and, if the assertion fails, the simulator can pause the simulation immediately and enter the interactive mode for users to debug. Users can also specify a coping mechanism after assertion failure, e.g., call a custom function or simply log the incident. As a result, using SVA can reduce the time to localize the bug since the assertion is closer to the source of the fault as opposed to observing it only from the outputs. There are two kinds of assertions: immediate assertions and concurrent assertions. Immediate assertions behave as a normal statement (similar to assert functions in most software programming languages), whereas concurrent assertions are based on clocking semantics. The support for SVA varies across different RTL simulators, but most simulators support immediate assertions.

An example of an immediate assertion is shown in Listing 2.11. Note that the assertion check occurs whenever the `always_comb` block is evaluated and, if it fails, the simulator will print out `"FAIL"`.

```
module mod (
    input logic a
);

always_comb begin
    assert (a != 1'b1) else $display("FAIL");
end

endmodule
```

Listing 2.11: A simple example that prints out "FAIL" when assertion fails

2.6 Prior work on hardware debugging and verification

Most of the prior work on debugging the output of hardware generators focuses on how to get to the bug *faster*, not making it *easier* to debug. That is, the majority of previous works try to make the RTL simulation or emulation more performant using various compiler techniques, and only a few discuss how to build a debugging system for users to localize bugs quickly.

In the field of FPGA prototyping and emulation, there are many commercial tools and research projects related to debugging. Xilinx ChipScope [24] is a set of tools that allows designers to probe internal FPGA signals while the FPGA is running. Designers need to place a soft core in the FPGA design beforehand, which specifies the trigger conditions and sampling rate. Upon triggering, the IP core can collect signals of interest and send them to the host machine for users to inspect. Because the probe core is integrated into the design, ChipScope is limited by the memory resources available on the FGPA. In addition, since designers need to specify the signals ahead of time before realizing the entire design onto FPGA, localizing the bug can be a game of “cat and mouse”: multiple runs of synthesis and place and route are required before the bug is found, extending debugging time much more than an ideal RTL debugger would.

One solution for handling the limited scope of probing is to use a full state checkpoint on FPGA. The checkpoint captures the entire state of the design and can be loaded into traditional RTL simulators for much more detailed inspection. Using a checkpoint combines the benefits of FPGA prototyping and RTL simulation, but it is difficult to set up, and designers need to fine-tune the checkpoint interval to avoid creating a bottleneck in the performance due to memory bandwidth constraints [25]. DESSERT [26] is a research project based on the Chisel generator framework. It addresses this issue by using two identical FPGA instances spaced apart by simulation time. Once the head FPGA detects an error, it sends snapshot commands to the tail instance to capture the state. Using such a setup avoids manual snapshot controls and minimizes the snapshot intervals. The system presented in this thesis is somewhat orthogonal to checkpoint-based FPGA emulation; however, designers can still use our system to debug once the checkpoint is replayed in a simulator.

Treadle is a hardware simulator that works on top of the Chisel compiler stack [27]. Although treadle is slower than state-of-the-art RTL simulators [28], it offers many convenient debugging features for users. For instance, it automatically generates test harnesses for Chisel designs and allows users to peak and poke the circuit state during simulation. However, because there is no symbol mapping in the Chisel compiler, users have to use lowered RTL names rather than source-level symbols, when interacting with treadle.

Inspect [29] is an experimental system designed for debugging in the LegUp HLS. Inspect defines a symbol table which is used to reconstruct the source execution state so users can debug HLS programs at source-level. Because it is tightly integrated into the compiler tool chain, it has control over the code generation as well as various compiler passes. As a result, it can only work with a certain compiler and a simulator and cannot be generalized to all hardware generator frameworks. In addition, it suffers significant performance slowdown since performance is not its main focus. Our work on HLS-related debugging builds on this work to make the concept more general so it can be applied to different HLS compilers.

2.7 What to learn from software

In the early stages of software engineering, developers had to write their code in a high-level language such as C or COBOL and then debug the assembly code, which became difficult to do when compiler optimizations were introduced. David Ditzel and David Patterson [30] argued that the computer system should use high-level languages for both programming and debugging, and, moreover, that the system should report execution errors in terms of the high-level language source program.

Over the decades, software debugging frameworks have evolved, and many standards have been proposed and adopted to encourage information exchange among different components in the debugging ecosystem. In general, there are two main components in a source-level software debugging system: a symbol table and a debugger. The symbol table is a database that stores the source-level information and the debugger interacts with the execution platform and the user. For instance, DWARF [31] is a symbol table format used in many languages for Linux systems, such as C, C++ and Rust; it can also be consumed by debuggers such as gdb and lldb.

For a compiled language such as C and C++, the compiler needs to track the source file information and store it in a separate symbol table that maps variable names to actual memory addresses and vice versa. For a scripting language such as Python, the debugging information is stored within the interpreter runtime as the interpreter runs through the code. If the source-language is trans-compiled to another programming language, such as TypeScript to JavaScript, the symbol table primarily deals with symbol transformation since the target language compiler can handle the rest of the symbol table mapping. Regardless of the form of the symbol table, it has to store the information about source location and type information, which can be used to reconstruct source-level

variables during debugging. As a result, it is the compiler's responsibility to create a faithful symbol table so that users can debug the program at the source level correctly.

However, generating a correct symbol table is not always straightforward. Many transformations in software compilers, such as constant propagation, loop reordering, and function inlining, destroy the source structure, making it difficult to track symbols. In addition, many meta-programming techniques in software programming languages, such as C++, are known to be difficult to reason about. To solve this issue, the compiler encodes the template information into the symbol table so that the information can be reconstructed during debugging [31]. Since hardware generator frameworks have compiler and language structures similar to those of software languages, the frameworks face similar challenges when producing the symbol table and can apply similar solutions to these problems.

During program execution, the debugger can load the symbol table and display the values with their original variable names instead of memory addresses. For instance, if we are debugging a C program and need to inspect the value of a struct, the debugger first reads out the memory address of the struct from a register or a memory location based on the symbol table. Then it calculates the memory offset for each struct member, reads out the memory values, composites the members to form a struct, and renders the struct to the user. For a program where variables have different lifetimes, the variable binding, or scope, depends on the context of execution, and the debugging mapping must change accordingly. Another important feature the debugger provides is a breakpoint, which pauses the program execution and allows users to inspect the stack frame to understand the program state. Depending on the language execution environment, breakpoints are implemented differently. Breakpoints on a compiled program usually require assistance from the hardware. The debugger typically patches the target instruction specified by the breakpoint with a debug or trap instruction, which later alerts the debugger if executed. Upon the trap triggering, the debugger can then proceed to determine whether to pause the execution for inspection given the condition associated with the breakpoint or continue execution. Again, the mapping from a source location to target instruction is stored in the symbol table. For an interpreted language, the debugger inserts a flag to the interpreter, which pauses execution before running the particular instruction. This process is done solely in software.

Another useful debugging feature in software systems is reverse debugging. During normal execution, system information such as CPU and external interrupts are collected as logs. When the programmers want to reverse the execution flow, these logs can be used to reconstruct the previous execution state. The instrumentation of log collection can be done either through hardware, such as special trace registers [32, 33], or pure software. Reverse debugging has been shown to significantly reduce the time it takes to locate difficult-to-reproduce bugs, such as stack corruption and race conditions. However, collecting hardware events slows down the execution by at least a factor of two [34], making reverse debugging difficult to deploy in many systems. In addition, most of the

state-of-the-art trace collecting techniques require advanced hardware features which might not be available to the programmers [32].

2.8 Summary

The conflict between the need to verify and debug the hardware design at the RTL level and the illegible RTL code produced from hardware generators creates a debugging problem in hardware generator frameworks. As discussed in the following chapters, we addressed this issue by creating a hardware generator debugging infrastructure modelled after software debugging systems; that is, we defined a symbol table and built a debugger. In particular, since the hardware generator frameworks use some form of a compiler, we implemented compiler passes to track and extract debugging information to produce a valid symbol table. We then built a debugger to reconstruct the source-level information. Besides borrowing ideas from software systems, we also found unique opportunities in hardware systems that are difficult to come by in software. Most, if not all, simulators support waveform recording during simulation out of the box. This feature implies that designers can easily use reverse-debugging tools at the source-level, a feature difficult to implement in software without the help of special hardware support. In addition, unlike software compilers, where it is difficult to prove the correctness of an optimization, hardware designs can be easily verified for equivalence using standard formal tools [23]. This allows the framework to have a debug mode where most optimizations are off and still guarantee the logic equivalence between debug and optimized RTL.

The rest of the thesis details algorithms and implementation techniques to build such a system. Specifically, we will discuss how we can reuse the SystemVerilog features introduced in this chapter, such as VPI and SVA, to reduce our implementation effort.

Chapter 3

Building a Debugging System for Hardware Generators

This chapter describes one way to organize an efficient debugging system that integrates seamlessly with popular hardware generator frameworks and RTL simulators. Our system uses the same organization as software debugging systems. That is, it partitions the debugging system into three separate components, namely the symbol table, the simulator (execution platform), and the debugger. We divide the debugger into two sub-components: the debugger runtime and the debugger frontend. The debugger runtime interacts with the simulator and symbol table, and the debugger frontend handles user interaction, as shown in Figure 3.1. This separation of functionality enables the debugger frontend to be used on a different host machine, enabling remote debugging. This feature is highly useful since large-scale simulation typically runs on remote servers.

We believe the key to achieving our design goals is to define a clean interface between the three main components. Each interface consists of a small set of simple primitives:

- simulator interface: it controls the simulator, such as pausing and resuming the simulation; it also fetches and sets its data values.
- symbol table interface: it obtains mappings between source variables and simulation variables from the symbol table; it also informs the source location for the users to set breakpoints and reconstruct stack frames.
- debugger UI interface: it sends the user's commands to the runtime as well as renders source-level stack frames upon breakpoint hits.

The primitives chosen for the simulation interface have a fundamental impact on the simulation performance and the debugging semantics. As we demonstrate in this chapter, some "obvious"

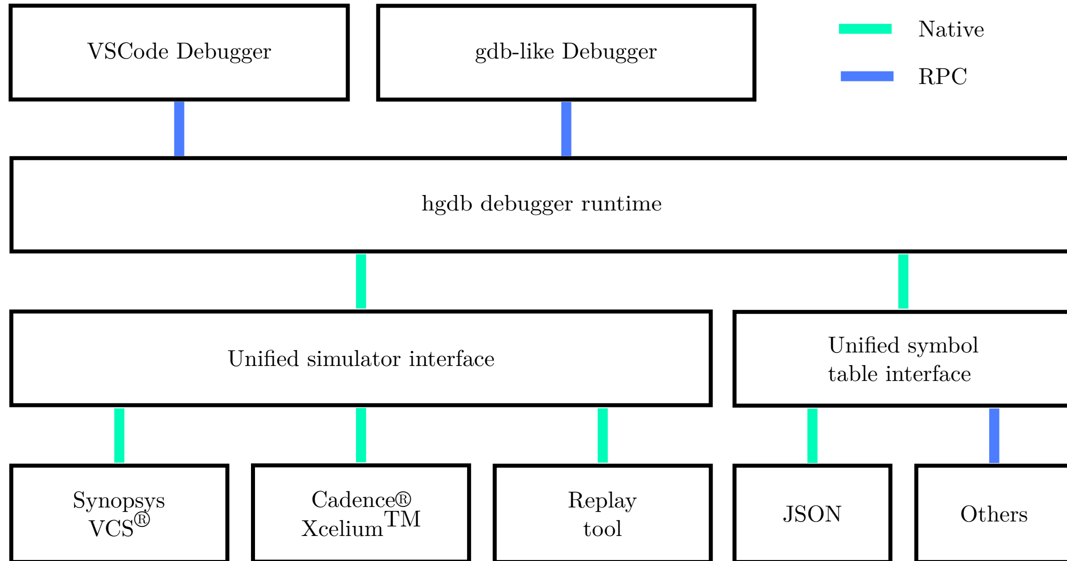


Figure 3.1: System design diagram, which is divided into multiple components. Each component communicates with the others via well-defined interfaces.

choices of primitives slowed down the simulation significantly, whereas others did not result in correct debugging semantics. We also strove to achieve orthogonality when creating interfaces to ensure the design choices for one interface primitive did not affect those of another interface. The capabilities of debugger runtimes strongly affect the debugging information that the symbol table provides, so we focus on the debugger runtime first.

3.1 Debugger runtime design

The biggest challenge of building an efficient debugger for hardware generators is implementing breakpoints, which forms the basis of debugging. We follow the same breakpoint semantics as that of software: the programmers expect the program to be paused precisely right before the target platform executes the line specified by the breakpoint. As we will demonstrate, providing such execution semantics is challenging because of the concurrent hardware execution semantics.

The most obvious solution is to use the breakpoint features offered by the simulator. In most simulators, users can set breakpoints—either through the graphic interface or through some Tcl scripts—by specifying the file name and line number in RTL. However, because the simulator knows only the RTL source structure, to use this approach, the symbol table needs to maintain location mappings. For every breakpoint capable source-level location, the debugger needs to know the corresponding RTL location. Whenever users set a breakpoint at the source level, the system translates it to a RTL location and sets the breakpoint in the simulator.

Features	Xcelium	VCS	Questa	Verilator	iverilog
DPI	✓	✓	✓	✓	✗
VPI	✓	✓	✓	✓	✓
Native breakpoint	✓	✓	✓	✗	✗
Tcl interface	✓	✓	✓	✗	✗
hgdb	✓	✓	✓	✓	✓

Table 3.1: Debugging features and SystemVerilog specification compliance for popular commercial and open-source RTL simulators. ✓ indicates incomplete supports.

However, this solution has several drawbacks. First, few simulators offer native breakpoint support, and their interfaces vary across the vendors. Some popular simulators, such as Verilator, do not even offer a way to set native breakpoints. Second, even for the same simulation vendor, different tools may require a different set of scripts. For instance, Verdi and VCS are both from Synopsys, but Verdi uses `srcTBAddBrkPnt` to set breakpoints, whereas VCS uses `break`. This issue is the hidden engineering cost of using proprietary interfaces. Table 3.1 shows the simulator support for setting breakpoints using native breakpoint or Tcl scripts. All the non-commercial simulators lack such features.

Third, breakpoints offered by the simulator significantly slow down the simulation speed and are probably too slow to be practical in large designs. This problem arises partially because enabling breakpoint support in the simulator disables certain compiler optimization passes since the simulator needs to maintain its internal symbol table. To illustrate the performance slow down, we use a popular commercial simulator on a RISC-V SoC called RocketChip [35]. RocketChip is written in Chisel and comes with a suite of benchmark applications that exercise various parts of the system, such as a floating point unit and a direct memory access (DMA) engine. Figure 3.2 shows the performance slowdown when we switched on breakpoint support in a commercial simulator. Note that no breakpoint was inserted, but the simulation was already about $2\times$ slower.

Given the limitations of native breakpoint support, another solution is to use DPI calls. The SystemVerilog LRM specifies that the statements inside a procedural block must be evaluated in sequence, regardless of the number of procedural blocks a module has. As a result, in a single-threaded simulation environment, if we place one DPI call statement before every relevant RTL statement, the DPI function can serve as a “trap” that pauses the simulation after the simulator makes the call. This behavior is identical to that of software breakpoints. For simplicity, we used a simple code example that computed the input sum based on the input array’s parity, as shown in Listing 3.1. Listing 3.2 shows the generated RTL without any compiler transformation. We discuss various aspects of primitive designs using this code snippet.

The call takes a unique ID to identify the instance, and another ID for the statement, as shown in Listing 3.3. The instance ID is a module parameter, which will be set with a unique number during instantiation by the generator compiler. The statement ID should be unique within a module

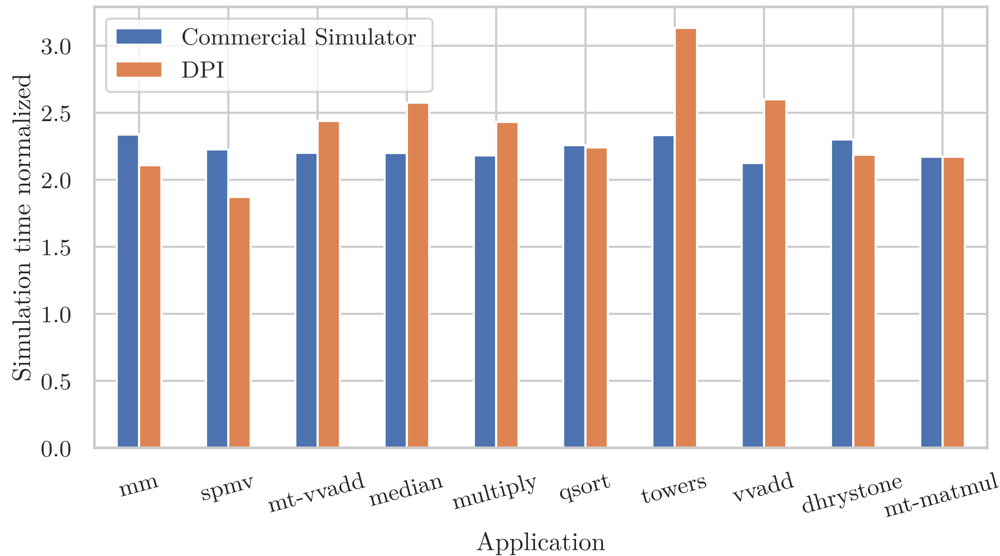


Figure 3.2: Performance slowdown by turning on/adding various features to a commercial simulator vendor. No actual breakpoint is inserted during the simulation. The slowdown is normalized to the same simulator without any features turned on.

```
int sum_odd(int input[4]) {
    int sum = 0;
    for (int i = 0; i < 4; i++) {
        if (input[i] % 2) {
            sum += input[i];
        }
    }
    return sum;
}
```

Listing 3.1: Simple C code to accumulate the input sums based on individual parity.

definition. We can use these two IDs to locate any module instance and its debugging information. Not all RTL statements correspond to locations in the original source code, e.g. line 10. In that case, no DPI function call is inserted.

The beauty of using DPI is its simplicity. If a DPI function is called, we are certain that the corresponding source statement is to be executed. For instance, if the DPI function at Line 15 is called, we know for sure that the parent `if` condition is true, i.e. `inputs[i]` is odd, without knowing the value of `inputs` and `i`. Another major benefit of using DPI to emulate breakpoints is its wide support. Most of the popular simulators including open-source Verilator support DPI function calls.

```
module sum_odd (
    input  logic[31:0] inputs[3:0],
    output logic[31:0] sum
);

always_comb begin
    int i;
    sum = 0;
    for (i = 0; i < 4; i++) begin
        if (inputs[i]) begin
            sum += inputs[i];
        end
    end
end

endmodule
```

Listing 3.2: RTL code that implements the same logic as in Listing 3.1.

Table 3.1 also shows the status of major simulator support for DPI.

One drawback of this approach is that it requires the compiler to insert DPI function calls. This can be accomplished by using a standard compiler pass after introducing the DPI function primitive into its IR, as shown in Figure 3.3. The compiler first uses an analysis pass to identify unique statements and instances and assign a unique ID to each identified primitives. Then during code generation, the compiler inserts DPI function calls in front of any primitive that has an ID. The compiler also ensures that each instance is assigned a unique ID. Another change related to code generation is continuous assignment. Because SystemVerilog LRM limits DPI function calls to procedural blocks, the compiler must transform continuous assignments into respective `always_comb` blocks, which have the same semantics during simulation.

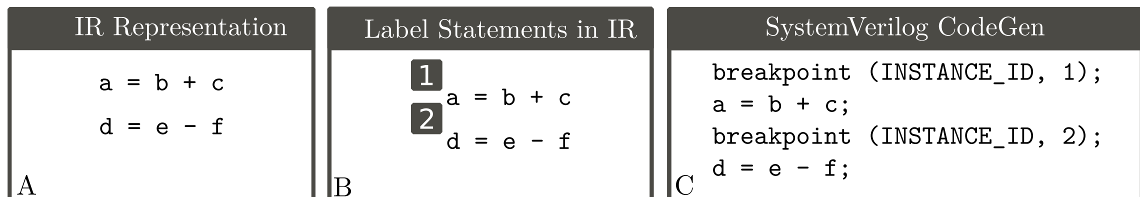


Figure 3.3: Compiler passes to insert DPI-based breakpoint calls into the design. The first pass labels each statement that has corresponding source locations and instances. The second pass inserts DPI calls before these statements with proper caller arguments, the combination of which is used to identify the source locations.

Similar to the approach that uses native breakpoint, DPI-based breakpoint emulation suffers from performance slowdowns. Fundamentally, there is no difference between native breakpoint and

```
import "DPI-C" function void breakpoint(input int  instance_id, input int  stmt_id);

module sum_odd #(parameter INSTANCE_ID = 32'h0)
(
    input  logic[31:0] inputs[3:0],
    output logic[31:0] sum
);

always_comb begin
    int i;
    breakpoint (INSTANCE_ID, 32'h0);
    sum = 0;
    breakpoint (INSTANCE_ID, 32'h1);
    for (i = 0; i < 4; i++) begin
        breakpoint (INSTANCE_ID, 32'h2);
        if (inputs[i]) begin
            breakpoint (INSTANCE_ID, 32'h3);
            sum += inputs[i];
        end
    end
end

endmodule
```

Listing 3.3: RTL code shows the breakpoint DPI function signature and simple use case using generated RTL from Listing 3.2.

DPI-based emulation. Both methods pause the simulation before a target statement is evaluated. Figure 3.2 shows the simulation performance using the RocketChip test suite with the same commercial simulator. When DPI calls are inserted into the RTL, they slow down the simulation by at least 2×. Note that for this test, the C function exits immediately when called, thus minimizing the DPI overhead. If the call includes the logic to check whether any breakpoints are inserted, the overhead increases significantly because thousands of DPI calls are made each clock cycle.

Even if the user tolerates this slowdown, there is another problem with the native breakpoint and DPI-based approaches. Because most simulators use an event-loop based on the simulation model described in Section 2.5.1, a combinational procedure block may be triggered multiple times before the loop stabilizes. These multiple triggers all occur at the same simulation time. Listing 3.4 shows some simple SystemVerilog code that has a procedural block triggered multiple times. In the initial block, `c` is changed, thus triggering the evaluation of `block2`. Furthermore, `block2` modifies both `b` and `d`, thus triggering `block1` and `block3`. While evaluating block3, we modify `e`, which triggers block1 again. As a result, `block1` is triggered twice before the simulation stabilizes. If we set a breakpoint at line 5, the users will see the breakpoint triggering twice. These simulation semantics

```
module top;
logic a, b, c, d, e, f;
always @(*) begin :block1
    $display("1");
    a = b;
    f = e;
end
always @(*) begin :block2
    $display("2");
    b = c;
    d = b;
end
always @(*) begin: block3
    $display("3");
    e = d;
end

initial begin
    // output 2 1 3 1
    c = 1;
end
endmodule
```

Listing 3.4: Multiple triggering during RTL simulation caused by simulator event loop. When the simulation runs, we get output `2 1 3 1`, where `block1` gets triggered twice.

conform to the SystemVerilog LRM and are not incorrect. The logic is indeed synthesizable according to commercial synthesis tools. Nonetheless, although correct, this double triggering does not match the designers' view of sequentially executing code.

Another implication of these event-loop based simulation semantics is that the source-level execution frame reconstruction may be incorrect while the simulation stabilizes. Regardless of whether the simulator is single-threaded, there is little guarantee that the simulator will schedule the code blocks in the same order as the source-level programs. Many simulators reorder statements to reduce the inter-code block dependencies. As a result, when we pause the simulator inside the simulation loop, the simulator is in a *transient* state. In other words, when the simulator iterates through the event queues, some signals have out-dated values when we pause the simulation. If we use these out-dated values to construct a source-level stack frame, users will see inconsistent results. Again, this point conforms to SystemVerilog LRM but is undesirable for hardware generator debugging.

The root cause of the two shared systematic flaws inherent in native breakpoint and DPI-based emulation, namely incorrect breakpoint semantics and performance slowdown, is that the simulator is paused for each RTL statement. We can solve both problems by pausing the simulator after it stabilizes. To accomplish this goal, we make two observations about modern digital designs:

- Most designs are synchronous, which implies that signals must be stable before the rising edge of the clock;
- Most RTL simulations use zero-delay logic models, since they are faster, and designers usually check the timing using static timing analysis (STA) tools.

The first point means our system only needs to check for potential breakpoints at the rising edge of the clock for that section of logic. The second point means that all pending simulation updates happen inside the same simulation time slot and the simulator only advances the time once it stabilizes. These two properties imply, first, that clock edges are the only events that advance time; second, at the rising edge of the clock, the simulation state is stable. These two properties solve the issue by allowing us to emulate breakpoints at the clock edge. [1]

We introduce two primitives to implement clock edge-based breakpoint. The first primitive is `place_callback_on_signal`, which allows for pausing the simulation whenever the designated signal changes its value, and we can then proceed to check the inserted breakpoints. The second primitive `get_signal_value` is used to obtain a signal's current value from the simulator. For example, if we only want to check the breakpoints at the posedge of the clock, we use the first primitive to insert a callback whenever the clock value changes; then, inside the callback, we use the second primitive to check the clock value and resume the execution if that value is low.

While evaluating breakpoints only on clock transitions has many benefits, this approach does not really correct the difference between the programmer's and simulator's view of the execution order. This difference creates subtle issues that we must address. One of the biggest challenges is that certain desirable intermediate values are overwritten while the state is converging. This typically happens when the same variable is reused inside the same code block to obtain the final result. Using our example code in Listing 3.1, a variable called `sum` can be overwritten multiple times inside a `for` loop for accumulation. If we inspect the design state at the next clock edge, we will see only the final result; all the intermediate partial sums will be lost.

Our solution is to leverage a widely used algorithm in hardware generator compilers, namely static single assignment (SSA) [36]. Because the symbol name aliasing problem happens only in combinational logic, SSA is the perfect solution. We first apply loop unrolling to remove any loop dependencies. This is feasible in hardware since all the loops must be finite to be synthesizable. During the SSA transform, conditional statements are flattened so that each variable is assigned exactly once. Listing 3.5 shows the C code after loop unrolling and Listing 3.6 shows the generated RTL. Note that Line 5 will be mapped to Lines 8, 9, 10, and 11 due to loop unrolling. Listing 3.7 shows the C code after the SSA transformation and Listing 3.8 shows the generated RTL.

Note that the transform creates several temporal variables to hold the value of `sum`, which implies that the variable mapping can differ depending on the context. We can use standard compiler

[1] This technique also works with gate-level simulation with delays, since the logic will have stabilized before the rising edge of the clock. If there is a timing violation, the simulator will have an incorrect value (as it should).

```c
int sum_odd(int input[4]) {
    int sum = 0;
    if (input[0] % 2) sum += input[0];
    if (input[1] % 2) sum += input[1];
    if (input[2] % 2) sum += input[2];
    if (input[3] % 2) sum += input[3];
    return sum;
}
```

Listing 3.5: C code after loop unrolling pass using Listing 3.1 as an input.

```systemverilog
module sum_odd (
    input  logic[31:0] inputs[3:0],
    output logic[31:0] sum
);

always_comb begin
    sum = 0;
    if (input[0] % 2) sum += input[0];
    if (input[1] % 2) sum += input[1];
    if (input[2] % 2) sum += input[2];
    if (input[3] % 2) sum += input[3];
end

endmodule
```

Listing 3.6: RTL code generated using Listing 3.5 as an input.

```c
int sum_odd(int input[4]) {
    int sum0, sum1, sum2, sum3, sum4, sum;
    sum0 = 0;
    sum1 = sum0 + (input[0] % 2? input[0]: 0);
    sum2 = sum1 + (input[1] % 2? input[1]: 0);
    sum3 = sum2 + (input[2] % 2? input[2]: 0);
    sum4 = sum3 + (input[3] % 2? input[3]: 0);
    sum = sum4;
    return sum;
}
```

Listing 3.7: C code after SSA transformation using the IR from Listing 3.1.

techniques to track the variable mapping. In this case, if the breakpoint hits Line 8 in Listing 3.8, we should fetch the value of `sum0` to represent `sum`, and `sum1` at Line 9.

Due to loop unrolling and SSA, if the user sets a breakpoint at Line 5 in Listing 3.1, we need to emulate four breakpoints in Listing 3.8. However, since these two mapped statements always execute regardless of the input condition, we need something called an *enable* condition that determines which

```
1  module sum_odd (
2      input  logic[31:0] inputs[3:0],
3      output logic[31:0] sum
4  );
5
6  logic [31:0] sum0, sum1, sum2, sum3, sum4;
7  assign sum0 = 0;
8  assign sum1 = sum0 + (inputs[0] % 2? inputs[0]: 0);
9  assign sum2 = sum1 + (inputs[1] % 2? inputs[1]: 0);
10 assign sum3 = sum2 + (inputs[2] % 2? inputs[2]: 0);
11 assign sum4 = sum3 + (inputs[3] % 2? inputs[3]: 0);
12 assign sum = sum4;
13
14 endmodule
```

Listing 3.8: RTL code after SSA transformation using Listing 3.7 as an input

line can be active during simulation. For instance, the enable condition for Line 8 in Listing 3.8 is `input[0] % 2`, which specifies that the potential breakpoint can be enabled only if `input[0]` is an odd number. The `enable` condition can be obtained by AND-reduction on the SSA transform condition stack. For conditional logic in sequential blocks, `enable` condition extraction is much easier, since we can simply walk up the hierarchy in the syntax tree and perform an AND-reduction on the conditions. The symbol table must store the enable condition along with the target breakpoint so that we can infer the condition when setting up breakpoints.

While SSA can solve the issues with scalar values, it cannot resolve the memory writes. Unlike scalar signals, memory devices such as register files and dual-port memories allow multiple value updates at the same time. Generator frameworks such as HLS abstract away memory writes from interfacing with the raw memory device. Users instead write zero-delay memory update code at the source level, e.g., in C++. As a result, the simulator memory update semantics conflict with that of source code, and the debugger cannot faithfully reconstruct the source-level execution state when users step over the memory device's writes (see Figure 3.4). If we set a breakpoint at Line 7, even though at source-level the memory write (`array[addr2]`) has not yet happened, the simulator already updates the value.

We can use a "shadow copy" of the memory device to solve this problem. The main reason for the issue is that the simulator losses unmodified memory state at the clock edge. The shadow copy of the memory simply pipelines the memory, i.e., it stores old content from the previous clock cycle. With the help of the symbol table, we can then reconstruct the frame properly, as shown in Figure 3.5. The symbol table informs the runtime about which memory element to keep track of. Then at Line 7, instead of pointing `array` to the RTL array, it can redirect it to `$previous.array`, which will be handled by the runtime.

```c
void multi_memory_write(int addr1,
                        int addr2,
                        int value1,
                        int value2) {
    static int array[16];
    array[addr1] = value1;
    array[addr2] = value2;
}
```

Listing 3.9: Sequential memory update in C.

Sequential to parallel execution

```systemverilog
module multi_port_memory (
    input logic        clk,
    input logic[31:0]  addr1,
    input logic[31:0]  addr2;
    input logic[31:0]  value1,
    input logic[31:0]  value2
);

logic[31:0] array[15:0];

always_ff @(posedge clk) begin
    array[addr1] <= value1;
    array[addr2] <= value2;
end

endmodule
```

Listing 3.10: simultaneous update for multi-port memory in RTL.

Figure 3.4: Issues with memory writes due to multi-power memory updates. At the posedge of the clock, multi-port memory writes happen simultaneously, whereas memory is updated sequentially in software programming languages.

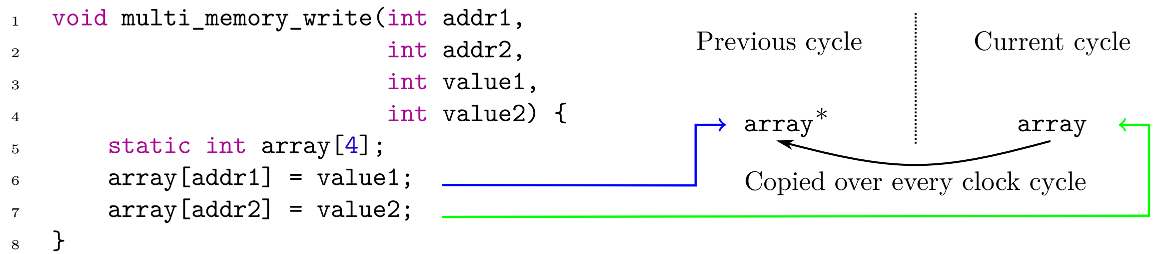


Figure 3.5: The write set is a shadow copy of the memory from the previous posedge. As a result, we have full visibility of the memory content before and after the memory update and thus are able to reconstruct the memory state faithfully.

Note that keeping a whole copy of the memory is not performant; at each clock cycle we must update the entire array. The memory content only changes when there is a write at the clock edge and the number of writes corresponds to the number of write ports. In addition, at the clock edge, we can read the value of the write port address to determine which address will be updated. As a result, we can reduce the shadow copy into a fixed-size write set. At each posedge of the clock, we first read out the write address for a memory port (if the write is to occur, i.e. write enable is high); then, we store the current memory value referred by the address to the write set. At the next posedge, the write set value is effectively the value from the "shadow copy". By adopting a much

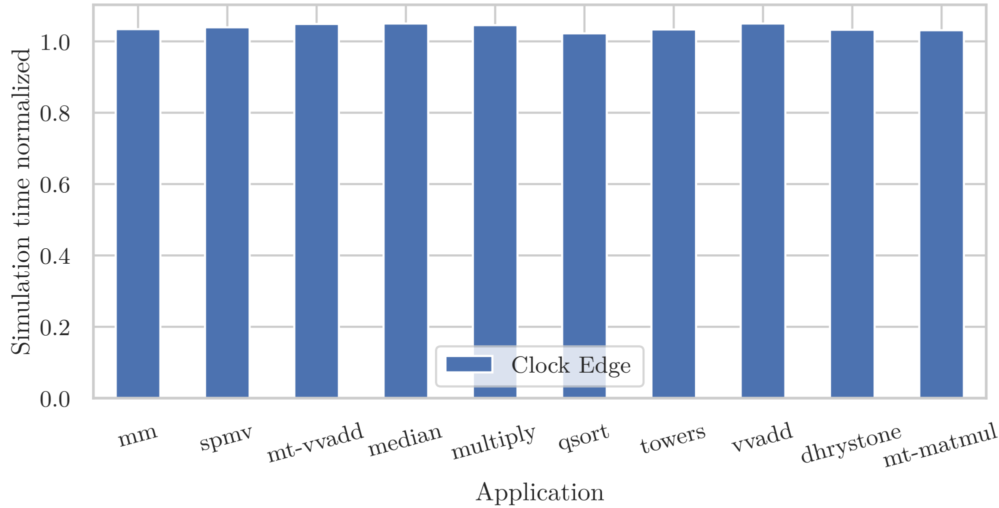


Figure 3.6: Performance slowdown from clock-edge based breakpoint emulation. No actual breakpoint was inserted during the simulation. The slowdown was normalized to the same simulator without any features turned on.

smaller write set, we reduce the memory updates to the number of write ports, which significantly improves performance if the memory device is large. The only change is having the symbol table store information about the write port address and write condition. In the actual implementation We can reuse the same breakpoint condition evaluation mechanism to implement this feature.

We need to keep track of the changes to memory devices in only one situation. First, the source-level programming model is sequential, and the users do not expect concurrent updates as is typically the case for HLS. Second, we keep track of the write set of a memory device if it is related to an inserted breakpoint. In other words, if no breakpoint is inserted or no memory device shows up in the context frame of an inserted breakpoint, there is no need to copy a target value over. In addition, the write set can be shared across different breakpoints if the same memory device is present, which means the overhead is relatively small.

Clock edge-based breakpoint emulation offers many advantages, with the most obvious one being performance. For a large-scale simulation, most of the computation time within a simulated clock cycle is consumed by evaluating the logic performed in the cycle. Compared to this time-intensive task, calling into a custom routine at each simulated clock cycle results in low overhead. We used the RocketChip benchmark suite shown in Figure 3.2 to evaluate the overhead, which was less than 5%, as shown in Figure 3.6.

Another advantage is the flexibility to extend the algorithm to other debugging features, such as watchpoints, assertions, and reverse debugging. These topics are discussed in detail in the following

sections. Other benefits include wide simulator support and no required modifications to the compiler code generation. Table 3.1 shows the support among popular RTL simulators (see the row named "hgdb"). Because many generator compilers already use SSA for analysis and optimization, adopting this breakpoint emulation algorithm does not alter the generated RTL.

3.1.1 Watchpoint emulation

A watchpoint, also called a data breakpoint, is a special type of breakpoint that pauses the execution whenever a new value is written to the target variable. In software debuggers such as gdb, the watchpoint can be based on either software or hardware [37]. A software watchpoint is very slow, because the debugger needs to single-step the program and test the value of the watch expression after each instruction. As a result, most watchpoints are implemented using hardware debug registers, with a fallback to software-based emulation if the former is unavailable or difficult to use.

Since we do not have hardware support inside the simulator, we have to emulate the watchpoint in our clock edge-based framework. As is the case with normal breakpoints, because the evaluation occurs only on the clock edge, the performance slowdown is rather modest. Because only an assignment statement can change a signal value, watchpoints share the same enable condition as normal breakpoints. When users place a watchpoint on a particular variable, the symbol table must provide all breakpoints that assign values to the target variable at the *source* level. This information can be obtained automatically through SSA transformation. Once the breakpoint condition is evaluated to `true`, the watchpoint is triggered.

Figure 3.7 demonstrates the process of watchpoint emulation using Listing 3.1 as a code example. When the breakpoint is inserted at Line 5, we first query the symbol table to identify any breakpoints that could potentially change the variable, that is, we identify any assignment whose left hand side is the target variable. The mapping can change at any time due to SSA transformation. In this case, there are five breakpoints, whose mapped RTL signals are `sum0-4`. We then internally set those breakpoints and create a tracker that tracks the value of `sum`. At posedge of the clock, we first ensure the watchpoint condition is satisfied; then, we compare the RTL signal value with the tracked `sum`. If they differ, we trigger the watchpoint and then update `sum`. The update is required because the following breakpoints could also trigger an assignment.

3.1.2 Assertion handling

"Failing fast" is one of the popular software techniques in agile software development [38]. It implies that the software developer can use assertions in the system early on to detect inconsistent changes or bugs. For instance, some feature upgrades may violate an early assumption in the system design, and by inserting assertions explicitly the programmer can catch the bug immediately. Since hardware generators are now software programs, designers are often encouraged to include assertions in their

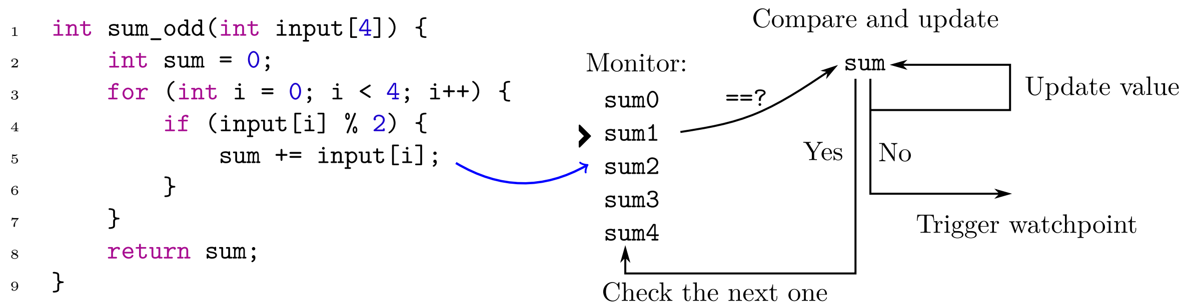


Figure 3.7: When a watchpoint is inserted, we first search all the related breakpoint locations that write to the same variable. Then we insert all matched breakpoints into the system. At every posedge of the clock, we check the mapped target variable to see if the value differs. If a mapped RTL value is different from the target value, we trigger the watchpoint breakpoint and update the target value. Note that the value change builds on the existing breakpoint system; that is, the breakpoint must still be active for it to trigger.

source code to check various conditions [39]. Because the assertion eventually gets lowered into RTL, we can apply similar techniques to render the assertion at source-level.

An easy approach is to implement the assertion in our existing debugging infrastructure, as the condition is no different from a breakpoint enable condition. However, doing so incurs two disadvantages. First, as shown in Section 4.4, obtaining RTL signal values from the simulator is expensive. If we evaluate many assertions at every clock edge, the performance slowdown can be significant. Second, if the assertion has clocking semantics, we must implement an FSM in our debugger runtime, unnecessarily complicating our system.

Most generator frameworks compile source-level assertions into SVA, which is supported by various simulators. In addition, SVA has a failure clause that executes any user-specified SystemVerilog statement if an assertion fails (see Section 2.5.3). Therefore, upon an assertion failure, the simulator can inform the debugger runtime which assertion failed. We can then easily render the failure to users, since fundamentally, there is no difference between a breakpoint triggering and an assertion failure. To facilitate the communication of assertion information between the simulator and the debugger runtime, we introduce a new simulator primitive called `trigger_assert`, which takes in the current instance and source location information. The runtime can then create an always-trigger breakpoint which contains assertion flags. Listing 3.11 shows the code example in SystemVerilog, where we call `$trigger_assert` and provide the source location (`example.py` and line 4). Once the primitive function is called, the runtime uses the first argument, the module handle, to determine which instance triggers the assertion. Then the debugger runtime uses the source location information to figure out which source-level breakpoint the assertion corresponds to. Finally the runtime inserts a breakpoint with an assertion flag on and informs the user of the assertion failure.

```
module mod (
    input logic a,
    output logic b
);

always_comb begin
    assert (a != 1'b1) else $trigger_assert(mod, "example.py", 4);
    b = a;
end

endmodule
```

Listing 3.11: VPI Custom system task to trigger assertion.

3.1.3 Reverse debugging

Reverse debugging, or "time-travel debugging", allows programmers to move back and forth in the execution flow. In software systems, reverse debugging significantly reduces debugging time [32, 40]. However, one of the main reasons this approach is not widely used in practice is that it requires specialized hardware. For instance, rr [33] requires the Intel CPU to be Nehalem or later, and if a virtual machine is used, the host must enable hardware performance counters virtualization. In addition to requiring hardware support, reverse debugging also significantly slows down the software execution. Therefore, race-condition related bugs are difficult to reproduce [32].

Hardware simulation does not incur similar problems. All SystemVerilog LRM compliant simulators can generate VCD waveforms, a type of trace file required for reverse debugging. In addition, although dumping waveforms during simulation incurs a performance penalty, it does not alter simulation ordering and thus does not affect the simulation result. Because traditional hardware debugging already relies heavily on waveforms, using waveforms for reverse debugging is a "free extra".

There are two levels of reverse debugging in our system, thanks to the clock edge breakpoint mechanism, namely the intra-cycle level and the inter-cycle level. "Intra-cycle reverse debugging" refers to the ability to reverse the execution flow within a clock cycle, whereas "inter-cycle reverse debugging" reverses the clock cycles. Because the runtime fetches the breakpoints in the order specified by the symbol table, if we reverse the fetch ordering during simulation, we can create an *illusion* of going back in time, even though the simulation is paused. Note that this type of "reversing" time does not need any support from the simulator.

The second type of reverse debugging requires other methods to reverse time. We introduce a primitive called `set_simulation_time_prev_value`, which is used to set the simulation time to the time of the previous value change. We thus revert to the time when the clock changed previously. We call this primitive twice, to reverse to the previous clock cycle, since we pause only on posedge.

Section 3.4 provides more details on how this primitive adequately implement reverse debugging.

3.1.4 Locate generated IP

A potential challenge with our debugging system is that developers rarely use a hardware generator framework to generate the entire design and test bench. Instead, IP blocks from different sources are composed together to form a larger system and then simulated under a complex testing environment. As a result, the hardware generator and the symbol table have only a partial view of the final design. From the simulator's perspective, the hardware we are debugging is just a subtree of the entire hierarchy. The RTL names in our symbol table are the names in the generated module, not the full name in the simulation.

To locate the generated IP within the entire simulation environment, we introduce a new primitive called `get_design_hierarchy`, which allows the debuggers to walk through the entire design hierarchy and obtain information such as signal names and module definition. We need only this primitive because although the design hierarchy presented in the symbol table is merely a subset of the final design, the relative hierarchy does not change. Section 3.4 explains how we locate the generated IP block using module definition. It also describes how to map source signals to the actual RTL hierarchical path inside the test bench by finding the block with matching module/signal names, should the module definitions not be available.

3.1.5 Summary of simulator primitives

The following list is a comprehensive description of all the primitives we have covered:

- `get_signal_value(rtl_signal_name)`. This primitive can be used to obtain any clock or RTL signal values. It is used to reconstruct the source-level execution frame and evaluate expressions.
- `get_design_hierarchy()`. This primitive walks the design hierarchy and returns instance information.
- `place_callback_on_signal(rtl_signal_name, callback)`. This primitive enables our custom function to be called whenever a specified signal changes its value.
- `trigger_assert`. This primitive provides the source location of the assertion that fails during simulation.
- `set_simulation_time_prev_value(rtl_signal_names)`. This primitive reverts the simulation state to the time when the specified signals change their values before the current time. It is only valid for offline replay.

In addition to the primitives critical for implementing breakpoint-related features, we have identified two primitives that are either informative or useful for debugging. The first primitive is `get_simulation_time()`, which displays the current simulation time. The second one is `set_signal_value(rtl_signal_name)`, which allows users to set a new value for an RTL signal. Setting the value is valid only for online interactive debugging, since replay tools typically cannot re-simulate the design once the state changes.

3.2 Symbol table for hardware generators

The symbol table stores critical debugging information about the source code and how the code relates to the generated output, in our case, the RTL code. Although there are many well-established symbol table standards, they are not suitable for hardware generators. For instance, DWARF [31] is a popular symbol table format for Linux systems, primarily targeted by languages such as C and C++. It specifies the correspondence between variables and memory/register address. As a result, we could not use DWARF for hardware generator symbol mapping directly, because the generated RTL code is addressable via symbol names (e.g., signal names) rather than memory addresses. This fact ruled out most of the symbol table formats. SourceMap [41] is another popular symbol table format for languages built on top of JavaScript, such as TypeScript. SourceMap is JavaScript-only and focuses on bidirectional mapping between JavaScript and the target language. Although in theory this type of bidirectional mapping would work in our case, we were not interested in mapping the generated SystemVerilog code back to the original source code. We also did not care about how to map source-level breakpoints to breakpoints in SystemVerilog, since all the breakpoints in our debugging framework are logical breakpoints. All things considered, we needed to design a new symbol table for hardware generators.

3.2.1 Mapping source-level symbols to RTL

The hardware is written as a software program before the compiler transformation. Therefore, its symbol table shares many characteristics with that of a pure software program. The minimum requirement is to store a symbol's name and what it has been mapped to in the RTL description. Since the symbol mapping changes across different source-level execution scopes, we also needed to store the scope information along with the symbol mappings.

The first set of source-level symbols have correspondence with RTL signals. These variables hold signal values at the source-level. Similar to software mapping, each variable must have a bijection to an RTL signal given an execution context. As discussed earlier, during the program execution, the same variable can refer to different RTL signals, such as `sum` in Listing 3.1. Since the mapping must be unique at any time, the compiler must perform certain transformations, such as SSA, to disambiguate symbols that share the same name in a different context. Note that the RTL signal

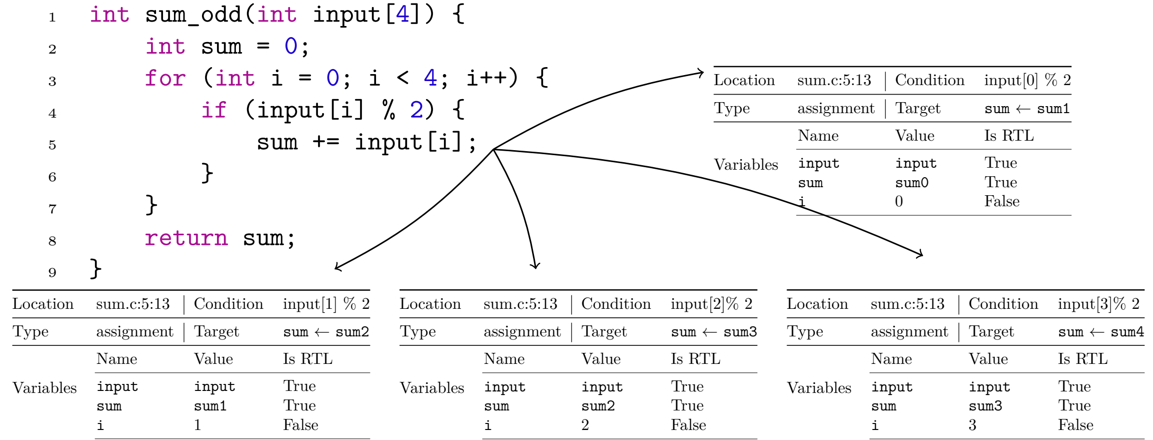


Figure 3.8: Symbol table for example code shown in Listing 3.1. We use `sum.c` as the filename. Notice that four different breakpoints are created at the particular source location.

only needs to be relative to the module itself, that is, the hierarchical name that starts from the current module definition. This point has two benefits: first, it makes the symbol table much more compact since there is no need to store duplicated names; second, we can address the child instances' symbols, which can be useful if the compiler creates implicit instances. As shown in Figure 3.8, there are four logical breakpoints at Line 5 and thus four mappings, all slightly different from each other. Variables `i` and `sum` have different mappings due to loop unrolling and SSA. They also have different enable conditions, despite sharing the same source location.

The second set of symbols is subtle yet important. These symbols are used to control the program execution and disappear in the RTL. We called this set of symbols "generator parameters", because they parameterize the hardware construction during the program execution. They can be as simple as boolean values or as complex as meta-programmed complex data types. If we treat the generator execution process as the "compile time" for RTL, the generator parameters are highly similar to C++ templates. Both of them are able to execute expressions, make decisions at compile time, and get the result checked at compile time. For instance, `static_assert` in C++ statically checks legality of certain template value combinations and aborts the compilation if it fails. In generator frameworks, assertions for parameter validity usually trigger exceptions or errors and abort the program execution. Modern C++ allows meta-programming constructs such as `if constexpr` to conditionally include certain code blocks in the output, which is similar to conditional hardware instantiation.

Because the generator parameters control the hardware compilation, recovering their values can be helpful to designers when debugging parameter related issues. As with C++ templates, although the generator parameter can be of a complex type, the nature of hardware construction requires all values of the parameter to be known at any elaboration stage. Hence, the parameters are static

values to the hardware compiler and can be stored in the symbol table, as shown in Figure 3.8, where the loop variable `i` is statically stored in the symbol table.

Regardless of the symbol type, it is necessary to store the type information. Because the high-level programming languages typically have more flexible typing systems than RTL, the mapping of type information must account for the lowering pass' outputs. For instance, Chisel allows aggregated bundle types that are difficult to express with Verilog. A variable defined by the bundle type can have many layers of nested bundles, which will all be flattened and lowered into individual wires. The type mapping should store information on how to reconstruct the original bundle value using flattened wires. Section 4.5 discusses implementation details on how to store type information effectively.

Another important detail to store is the source location information, i.e., filename and line and column number. A typical software symbol table stores mappings from source locations to sets of CPU instructions. It might seem possible to do something similar for hardware generators, e.g., map the source location to a generated RTL statement, but this would be difficult and unnecessary. As we have shown, breakpoints stop the entire simulation, so there is no need to associate a breakpoint with a specific RTL code fragment. Thus, the symbol table only needs to store the logical source location, without the RTL mapping. It should also store high-level semantics associated with each location, such as assignment and left-hand-side variables, which are needed to implement watchpoints, as shown in Figure 3.8. Although all breakpoints can have the same type, e.g., `assignment`, and target variable, they have different assignment RTL values. For instance, in Figure 3.8, `sum1` and `sum2` are used to assign to the same variable `sum` but they may have different values during simulation. As a result, users will see that the value of `sum` changes across these two breakpoints. If a watchpoint is set on variable `sum`, such a value change should cause the simulator to halt. Therefore, the assignment information is the key to implementing watchpoints, as discussed in Section 3.1.1.

Generating a correct symbol table is not an easy task. It requires the host language to mark the source location for each eDSL statement and the compiler to track the symbol changes throughout the transformation. In Section 4.8, we discuss implementation details on how to extract symbol tables from popular generator frameworks.

3.2.2 Symbol table primitive design

Our aim of supporting various hardware generator frameworks implies that one of the system design goals is to making it easier for compilers to generate the desired debug information. Some frameworks use a scripting language such as Python; some use a virtual machine-based language such as Scala, and others use a compiled language such as C++. It thus makes sense to have frameworks generate their own symbol tables, provided the tables implement a set of minimum primitives. This also reduces the amount of work required to change compiler internals to produce the symbol table.

Based on software debugging systems and our implementation experience, we have defined three categories of symbol table primitives: 1. breakpoint information, 2. variable mapping, and 3. design

mapping. Breakpoint information-related primitives query the breakpoint based on user provided source location. Variable mapping-related primitives translate a high-level symbol to RTL names and inform the systems about the data type. Design mapping-related primitives assist the simulator runtime to emulate the breakpoints properly. Below is a list of primitives that each symbol table system should implement, grouped by their categories:

Breakpoint information:

- `get_breakpoint_from_loc(filename, line_num, column_num)` → `List<bp>`.
- `get_breakpoint_info(bp)`.
- `get_breakpoints_ordering()` → `ordering`.

Variable mapping:

- `get_scope_info(bp)` → `scope`.
- `get_var_type(variable_name, scope)`.
- `resolve_scoped_var(variable_name, scope)` → `rtl_signal_name`.

Design mapping:

- `get_top_design_name()` → `name`.
- `get_clock_signals()` → `clock_signals`

The primitive `get_breakpoint_from_loc` is essential for the symbol table. During debugging, users attempt to insert breakpoints using the source location, either by clicking relevant lines in the source code or by using commands. This primitive allows the debugger to translate a source location to logical breakpoint(s). Note that a list of breakpoints is returned by this primitive. If no breakpoint is available at a given source location, an empty list is returned. A more subtle case is where multiple breakpoints are returned for two reasons. First, because of the compiler elaboration and transformations, the same source location can map to multiple breakpoints. For instance, Line 5 in Listing 3.1 is mapped to four breakpoints due to loop unrolling. Second, the same generator definition can be instantiated multiple times in the same design. For instance, generator `sum_odd` can be instantiated in different places in the design to accumulate input values. As a result, the same source location maps to more breakpoints, e.g. $4 \times 2 = 8$, if it is instantiated twice.

After obtaining the breakpoints, we use `get_breakpoint_info` to learn about the breakpoint, such as what type of statement it corresponds to. As discussed in the next section, this primitive should also retrieve other necessary information related to breakpoint emulation such as the enable condition. `get_breakpoints_ordering` returns the ordering of breakpoints as emulated by the debugger. This depends on the source-level execution semantics but is mostly lexical. We use

`get_scope_info` to learn about existing variables given a particular breakpoint. Variables can be created or removed across different breakpoints in the same scope. Once we know all the variables in the scope, we can query the type of variable, such as whether it is a static value (i.e., generator parameter), or a primitive type (net type in SystemVerilog jargon); if not, we can determine how to reconstruct the complex type using primitive types. `get_var_type` tells us the type of variable stored in the symbol table, i.e., whether it is a static value or an RTL signal. `resolve_scoped_var` returns the value of a particular variable with primitive type. If the variable is static, it should return the value stored in the symbol table; if not, it should give a mapped RTL signal name. `get_top_design_name` allows the debugger to locate generated IP within the test environment, which is critical for establishing the mapping from the symbol table name to the actual RTL hierarchical name. `get_clock_signals` returns a list of unique clocks in the design, each of which defines a clock domain.

Although these primitives strongly resemble those pf symbol tables used by software, there are subtle differences in the semantics. When a breakpoint is triggered in software, the current scope is computed based on register values, e.g. stack/frame pointer and program counter. As a result, the debugger can compute the complete call stack by reconstructing the stacks recursively. In our case, because we are interested only in debugging the generated hardware and any generation-related bugs, we are not concerned with the source-level stack frames. Provided we present all generator parameters that are used to control hardware generation and all the variables that have RTL correspondence in the execution context, users can debug the hardware at source-level.

3.3 Debugger frontend interface

The debugger frontend interface is the bridge between users and the rest of the debugging system. On the one hand, commands from the debugger frontend are decomposed into interface primitives and then remotely executed in the debugger runtime. On the other hand, information such as execution frame information must be rendered by the frontend in the form of interface primitives. Note that the interaction between the debugger frontend and the debugger runtime is part of the timing-critical path of the system. Whenever the interaction happens, the simulation is paused, and the user sends debugging commands or receives debugging information from the system. Therefore, we can separate the frontend from the debugger over the network. The primitives are naturally remote procedure calls (RPCs), as shown below:

- `set_breakpoint(filename, line_num, column_num?, condition?)`. This primitive sets the breakpoint given the source location (column number is optional). Users can also set an optional condition to the breakpoint. If the source location is invalid, the system will not insert any breakpoint and will instead return an error to user.
- `set_watchpoint(variable_name, filename, line_num, column_num?)`. Like the previous

primitive, this one adds a watchpoint to a source-level variable at a source location. The simulator runtime looks up the symbol table and identifies any relevant assignments related to the target variable, as discussed in Section 3.1.1.

- `render_breakpoint(breakpoint, variables)`. This primitive renders a given breakpoint to the user, i.e. it shows source-level variables. It is called by the runtime and is executed at the frontend.
- `evaluate_expression(expression, scope_id)`. This primitive allows the frontend to evaluate arbitrary source-level expressions. The `scope_id` is used to identify execution content to map source-level variables to the actual RTL signals and static values.
- `set_value(variable name, scope_id)`. This primitive allows the frontend to set variable values during interactive debugging. Similar to `evaluation_expression`, `scope_id` is used to locate execution context.

Except for `render_breakpoint`, all primitives are implemented in the debugger runtime and are callable by the frontend. The interface between the runtime and frontend can be implemented in any network stack, such as TCP/IP, HTTP, or WebSocket, and the serialization can be either binary or text-based. Section 4.6 describes the implementation details of the interface, using WebSocket and JSON.

3.4 Putting everything together

The previous sections covered the breakpoint emulation scheme and interface primitives. In this section, we sketch out the algorithms that use the interface primitives to implement the debugging system. We detail the algorithms in the order of how the RTL simulator interacts with our system. We first describe how to initialize the debugger runtime when the simulation starts; thereafter, we discuss how to emulate the breakpoint at each clock edge and what to do after a breakpoint triggers. We also discuss some algorithmic benefits of using the primitives in certain ways that make the system implementation easier and provide added benefits. For simplicity, the algorithms presented here assume there is only one instance of the generated IP core in the testing environment. The actual implementation can reuse the same algorithm and isolate each generated design in its own "process," as discussed in Section 4.2.

The first step is to initialize the debugger when the simulation starts. We must locate the generated IP block, determine the clock signals, and place a callback on the clock changes, as shown in Algorithm 3.1. We first query the symbol table to identify the top design and its module name. Then we walk through the entire design hierarchy to see if there is any object that is instantiated from the target module. If found, we remember the full hierarchy name and use it for any later RTL

signal mapping (defined as `MapToRTL`). If the underlying simulator does not support obtaining the instance definition name, we use string matching on signal and child instance names, as described in the next chapter (Section 4.2). Once we have obtained the IP mapping, we query the symbol table and obtain a list of unique clocks. We then place a callback on the clock signals using a custom debugger runtime function called `EvaluateBreakpoints`.

```
// Get the design name and map it to the RTL
top_name ← get_top_design_name()
design_hierarchy ← get_design_hierarchy()
foreach instance ← design_hierarchy do
    if instance.module_name = top_name then
        mapping ← instance.instance_name
        break
    end
end

Function MapToRTL(symbol_name):
    // Remove the top name from the symbol name
    internal_name ← symbol_name.left_strip(top_name)
    return mapping + internal_name

// Get clock signals and insert callback
clock_signals ← get_clock_signals()
foreach clock ∈ clock_signals do
    mapped_clock = MapToRTL(clock)
    place_callback_on_signal(mapped_clock, EvaluateBreakpoints)
end
```

Algorithm 3.1: Initialization steps to set up the debugging environment with a simulator. Function `EvaluateBreakpoints` is defined in Algorithm 3.2. Notice that we also define a helper function called `MapToRTL` that maps the symbol name to the instantiated RTL signal name.

The function `EvaluateBreakpoints` is called whenever the simulator stabilizes and there is a list of breakpoints to evaluate. Therefore we model `EvaluateBreakpoints` as an emulation loop to exhaust the inserted breakpoints in sequence, as shown in Figure 3.9 and Algorithm 3.2. At the beginning of each loop, we fetch a list of inserted breakpoints that share the same source location based on the pre-computed ordering. If there is no breakpoint left to select, we exit the loop and wait for the next posedge of the clock. Otherwise we evaluate each breakpoint condition using values from the simulator. Note that each breakpoint can contain conditional expressions specified by the user, and we need to take them into account as well. If this is a watchpoint, we also need to make sure that it satisfies the watchpoint triggering condition. (We can reuse the same breakpoint emulation for watchpoints, as discussed in Section 3.1.1.) The same logic goes for an assertion if the flag is set. If there is any breakpoint that satisfies all the criteria, we reconstruct the stack frame based on the symbol table and then send the result to the user by using the function `SendBreakpointPoints`.

Once the user responds with a command, we loop back to the beginning.

There are several benefits of using such loop-based breakpoint scheduling given our breakpoint emulation mechanism. First, we can exit the loop immediately if there is no breakpoint inserted, thus minimizing the runtime overhead. Second, the breakpoint fetching mechanics obey the breakpoint ordering set by the symbol table and also allow breakpoints to be inserted or removed when the simulator is paused. This ability gives users a smooth debugging experience since they can change the breakpoint behavior any time. Third, a more subtle advantage is that the emulation loop works with both forward and backward execution. Once there is no breakpoint left, if the users continue to reverse time, the runtime can call the function `ReverseTime` to ask the underlying simulator to roll back the simulation state. This is because users can only change the breakpoint mode inside the function `SendBreakpointPoints`, and the loop goes back to the beginning after the function `ReverseTime` returns. The function `FetchBreakpoints` handles the mode change properly.

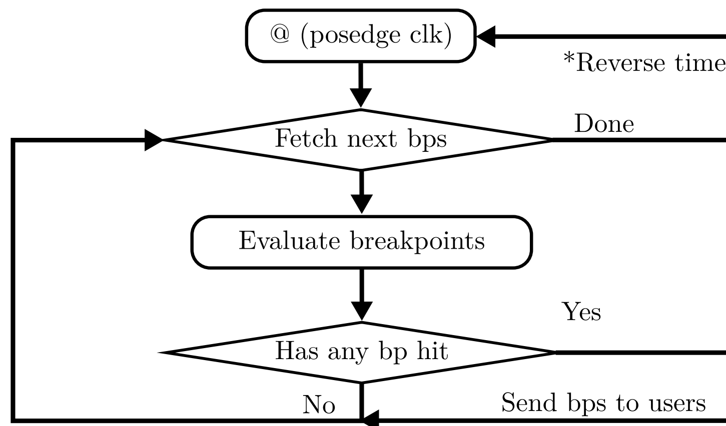


Figure 3.9: We can emulate breakpoints in a loop where we fetch breakpoints in order and check if any of them are active. If so, we reconstruct the breakpoint and send the result to the user. Otherwise we loop back and fetch more breakpoints. The loop terminates when we exhaust all inserted breakpoints, and we then wait for the next posedge of the clock.

The breakpoints are fetched through a function called `FetchBreakpoints`. This function keeps track of the list of breakpoints that have been evaluated and the current breakpoint. It behaves differently depending on the breakpoint evaluation mode, as shown in Algorithm 3.9. If there are no special debugging actions, the runtime is in the "normal" mode, meaning only the inserted breakpoints are of interest. To fetch normal breakpoints, we must first know the relative location of the current breakpoint within the inserted breakpoints, as shown in Algorithm 3.5. The current breakpoint might not belong to inserted breakpoints, due to user actions such as step over, which single-step through the source code. We use the `upper_bound` to ensure that the next breakpoint is *strictly after* the current breakpoint in the ordering specified by the symbol table. Once we have obtained a breakpoint candidate, we keep scanning the inserted breakpoints to see whether

```
// Static values maintained inside the debugger;
static breakpoint_mode ← Normal;
Function EvaluateBreakpoints():
    breakpoints ← FetchBreakpoints();
    if breakpoints.empty() then
        // We are done with this clock edge;
        if breakpoint_mode = ReverseNormal ∨ breakpoint_mode = ReverseStepOver then
            ReverseTime();
        end
        return;
    end
    active_breakpoints ← ∅;
    foreach breakpoint ← breakpoints do
        info ← get_breakpoint_info(bp);
        // Get breakpoint enable condition;
        enable_cond ← Evaluate(info.enable, breakpoint);
        // In case of conditional breakpoints;
        bp_condition ← Evaluate(info.condition, breakpoint);
        watchpoint ← Monitor(breakpoint);
        is_assertion ← breakpoint.assertion;
        if (enable_cond ∧ bp_condition ∧ watchpoint) ∨ is_assertion then
            // This breakpoint should be triggered;
            active_breakpoints ← active_breakpoints ∪ { breakpoint };
        end
    end
    if ¬ active_breakpoints.empty() then
        SendBreakpoints(active_breakpoints);
    end
```

Algorithm 3.2: Breakpoint emulation loop algorithm. The function `Evaluate` evaluates the expression based on the current simulator state and returns the expression value, defined in Algorithm 3.10. The function `SendBreakpointPoints` takes a list of breakpoints and constructs the stack frames before sending them to the user, which is defined in Algorithm 3.12. The function `ReverseTime` reverses simulation time, which is defined in Algorithm 3.3. Function Monitor tracks the target symbols and return `True` if the target symbol value changes, as defined in Algorithm 3.4.

any breakpoint matches the candidate's source location. Note that the same generator could be instantiated multiple times in a generated design. We are interested in rendering this kind of "concurrency" to the user, as explained in Section 4.2.

If the breakpoint mode is "step over," meaning the user wishes to examine every statement in the source code in the program execution order, we need to fetch the next breakpoint using the breakpoint ordering, i.e. `upper_bound`, as shown in Algorithm 3.6. Reverse normal breakpoints, shown in Algorithm 3.7, is the opposite of normal breakpoints. If the current inserted breakpoints are empty, we use the last breakpoint instead of the first one. We also use the reversed ordering

```
Function ReverseTime():
    clock_signals ← get_clock_signals()
    mapped_clock_signals ← ∅
    foreach clock ∈ clock_signals do
        mapped_clock = MapToRTL(clock)
        mapped_clock_signals ← mapped_clock_signals cup { mapped_clock }
    end
    success ← False
    while ¬ success do
        // Make sure we have at least one posedge
        set_simulation_time_prev_value(mapped_clock_signals)
        foreach clock ← mapped_clock_signals do
            value ← get_signal_value(clock)
            if value = HIGH then
                // This is the posedge clock
                success ← True
                break;
            end
        end
    end
```

Algorithm 3.3: Algorithm to reverse time. We first fetch the clock signals and then revert to a time when the signal previously changed.

of the breakpoints. Similar logic applies to fetching breakpoints in reverse step over, as shown in Algorithm 3.8.

The function `Evaluate` returns an integer value given an expression and breakpoint, as shown in Algorithm 3.10. It first parses the expression string into a tree. Because variables in the expression are context-dependent, we need the scope information to map it to RTL signals. The breakpoint is used to identify the source-level execution scope. Once we have obtained the variable mapping for a particular expression tree node, we first query the variable type, since it could be a static value. If not static, we map it to the full RTL hierarchy name and get its value from the simulator. We then use the values to evaluate the expression tree. The evaluation process is similar to that of an interpreter for scripting languages.

The function `Monitor` returns `true` if a tracked variable changes its value, as shown in Algorithm 3.4. If the statement is a watchpoint, we check whether the referenced value differs from the RTL signal the statement is assigning to. If they differ, we need to update the tracked value, since the next breakpoint may modify the same variable, and we want to ensure consistency. Note that in a certain case, called cold start, the value of the symbol of interest is not tracked. This only happens when the user first inserts the watchpoint and the system does not know the previous value. Hence, we allow the watchpoint to trigger unconditionally during cold start to prevent false negatives.

Once a breakpoint is triggered, we use the symbol table to reconstruct the execution frame and

```
static tracked_symbols
Function Monitor(breakpoint):
    if ¬ breakpoint.is_watchpopint then
        return True                          // Ignore any non-watchpoint breakpoints
    end
    scope ← get_scope_info(breakpoint)
    value ← Evaluate(breakpoint.assign_value, scope)
    if breakpoint.assign_symbol ∈ tracked_symbols then
        ref_value ← tracked_symbols.get(breakpoint.assign_symbol)
        if ref_value ≠ value then
            // This is a value change.  need to update the tracked value
            tracked_symbols.set(breakpoint.assign_symbol, value);
        end
        return ref_value = value
    end
    else
        // Cold start
        tracked_symbols.set(breakpoint.assign_symbol, value)
        return True
    end
```

Algorithm 3.4: Algorithm to monitor a particular expression, which returns True if a change is detected, or the expression is empty. The function `Evaluate` is defined in Algorithm 3.10.

then communicate the frames to the user, as shown in Algorithm 3.12. Depending on whether the scoped variable is static or not, we may query the simulator to obtain the signal value. We reuse the function `Evaluate`, because intrinsically there is no difference between an enable condition and symbol values, since they both involve expressions and get values from the symbol table and simulator. Reusing the function also enables complex symbol expressions in the symbol table mapping. After computing the stack frame, we use the debugger frontend primitives to render the result, and then wait for the user's action, such as continue or step over.

Now that we have discussed `EvaluateBreakpoints`, we can describe how to handle breakpoint insertion. The breakpoints can be added to the system through three different methods: conventional breakpoint insertion, watchpoint insertion, and assertions.

When the user sets a breakpoint at a particular source code location, we first query all breakpoints associated with that location; then we parse and validate the enable conditions and other related expressions. Finally we sort and store the breakpoints in the debugger, as shown in Algorithm 3.11. Sorting is necessary to ensure the correct ordering when fetching breakpoints. The actual system implementation can use data structures such as a red-black trees to speed up the insertion and `upper_bound()` performance.

To handle assertion, we first query breakpoints using the information provided by the VPI

```
// Static values maintained inside the debugger;
static inserted_breakpoints;
static current_breakpoint;
Function FetchBreakpointsNormal():
    breakpoints ← ∅;
    if inserted_breakpoints.empty() then
        return ∅;
    end
    if current_breakpoint = null then
        current_breakpoint ← inserted_breakpoints.first();
    end
    else
        current_breakpoint ← inserted_breakpoints.upper_bound(inserted_breakpoints);
    end
    next_breakpoint ← current_breakpoint;
    while next_breakpoint ≠ null do
        breakpoints ← breakpoints ∪ { current_breakpoint };
        if next_breakpoint.location ≠ current_breakpoint.location then
            // We have exhausted breakpoints that share the same source
            location;
            break
        end
        current_breakpoint ← next_breakpoints;
        next_breakpoint ← inserted_breakpoints.upper_bound(next_breakpoint);
    end
    return breakpoints;
```

Algorithm 3.5: Algorithm to fetch breakpoints in the normal order, i.e., the ordering specified by the symbol table.

```
// Static values maintained inside the debugger;
static inserted_breakpoints;
static current_breakpoint;
Function FetchBreakpointsStepOVer():
    breakpoints_ordering ← get_breakpoints_ordering();
    if current_breakpoint = null then
        current_breakpoint ← breakpoints_ordering.first();
    end
    current_breakpoint ← breakpoints_ordering.uppper_bound(current_breakpoint);
    breakpoints ← { current_breakpoint };
    return breakpoints;
```

Algorithm 3.6: Algorithm to fetch the next step over breakpoint. We need to read out the breakpoint ordering from the symbol table to determine which breakpoint is next.

function call; then, we use the same procedure that we used to handle breakpoints inserted by the users. We simply flag the breakpoint: `breakpoint.assertion` ← `true`. Note that in Algorithm 3.2,

```
// Static values maintained inside the debugger;
static inserted_breakpoints;
static current_breakpoint;
Function FetchBreakpointsReverseNormal():
    breakpoints ← ∅;
    if inserted_breakpoints.empty() then
        return ∅;
    end
    if current_breakpoint = null then
        current_breakpoint ← inserted_breakpoints.last();
    end
    else
        current_breakpoint ←
         reverse(inserted_breakpoints).upper_bound(inserted_breakpoints);
    end
    current_breakpoint ← next_breakpoint;
    while next_breakpoint ≠ null do
        breakpoints ← breakpoints ∪ { current_breakpoint };
        if next_breakpoint.location ≠ current_breakpoint.location then
            // We have exhausted breakpoints that share the same source
            location;
            break
        end
        current_breakpoint ← next_breakpoints;
        next_breakpoint ← reverse(inserted_breakpoints).upper_bound(next_breakpoint);
    end
    return breakpoints;
```

Algorithm 3.7: Algorithm to fetch breakpoints in the the reverse order. This is the complete opposite of Algorithm 3.5.

```
// Static values maintained inside the debugger;
static inserted_breakpoints;
static current_breakpoint;
Function FetchBreakpointsReverseStepOVer():
    breakpoints_ordering ← get_breakpoints_ordering();
    if current_breakpoint = null then
        current_breakpoint ← breakpoints_ordering.last();
    end
    current_breakpoint ← reverse(breakpoints_ordering).uppper_bound(current_breakpoint);
    breakpoints ← ∪ { current_breakpoint };
    return breakpoints;
```

Algorithm 3.8: Algorithm to fetch the next step over breakpoint in reverse order. This is the complete opposite of Algorithm 3.8.

we only need to trigger breakpoints if the flag is set regardless of the enable condition. This is

```
// Static values maintained inside the debugger;
static inserted_breakpoints;
static current_breakpoint_idx ← 0;
// By default we fetch breakpoint in the order of inserted breakpoints;
static breakpoint_mode ← Normal;
Function FetchBreakpoints():
    switch breakpoint_mode do
        case Normal do
            breakpoints ← FetchBreakpointsNormal();
            break
        end
        case StepOver do
            breakpoints ← FetchBreakpointsStepOVer();
            break
        end
        case ReverseNormal do
            breakpoints ← FetchBreakpointsReverseNormal();
            break
        end
        case ReverseStepOver do
            breakpoints ← FetchBreakpointsReverseStepOver();
            break
        end
    end
    return breakpoints;
```

Algorithm 3.9: Algorithm for `fetch_breakpoints` that changes the fetching order based on current breakpoint emulation mode.

because the VPI function is called only after an assertion fails, as shown in Listing 3.11. The fact that the VPI routine is called implies that the SVA has successfully executed to the assertion site, so no enable condition needed, as discussed in Section 3.1.2.

Watchpoint insertion is more complex. The difficulty comes from the fact that at the source location, the variable symbol that a user is interested in is usually a context variable, i.e., a variable defined prior to the breakpoint location. We must search the scope to find any relevant statements that assign to the target variable. Note that the assignments can also appear after the source location, which implies that the symbol table must track the variable mapping well within the scope. Once we have obtained all the assignments, we create breakpoints for those assignments and flag them accordingly. The rest is handled by the breakpoint loop and the `Monitor` function.

The algorithms presented here can easily be converted to the actual implementation. There are many global static variables defined and used by the algorithms. The actual implementation must garbage collect these variables to ensure that the variables do not grow indefinitely to avoid memory over-usage. In the next chapter, we describe how to implement these primitives using available frameworks and technologies to avoid duplicated efforts and to ensure maximum capability. We also

```
Function GetRTLValue(signal_name, scope):
    signal_name ← resolve_scoped_var(symbol.value, scope);
    mapped_signal_name ← MapToRTL(signal_name);
    value ← get_signal_value(mapped_signal_name);
    return value;

Function Evaluate(expression_str, breakpoint):
    expression ← ParseExpression(expression_str);
    scope ← get_scope_info(breakpoint);
    foreach node ← expression.nodes do
        symbol ← node.symbol;
        var_type ← get_var_type(symbol);
        if var_type.rtl then
            // This is a RTL signal;
            value ← GetRTLValue (symbol.value, scope);
        end
        else
            // This is a static value.  No need to query the simulator;
            value ← symbol.value
        end
        // Assign the value to the node;
        node.value ← value;
    end
    value ← expression.evaluate();
    return value;
```

Algorithm 3.10: Algorithm to evaluate the expression. We assume there is a parser function called `ParseExpression` which parses the expression string and returns an expression tree. We also assume that the expression tree data structure can interpret the expression once we have determined the values. Note that we have also defined a function called `GetValueFunction` that takes a symbol name and scope information and returns the RTL signal value.

offer further insights on optimizing the performance that were not covered in the current chapter.

```
static inserted_breakpoints;
Function SetBreakpoint(filename, line, column, condition, breakpoint_type, variable):
    breakpoints ← get_breakpoint_from_loc(filename, line, column);
    foreach breakpoint ← breakpoints do
        breakpoint.condition ← condition;
        breakpoint.type ← breakpoint_type;
        if breakpoint_type = WatchPoint then
            // Notice that we need to search the scope and see matched
            breakpoints;
            scope ← get_scope_info(breakpoint);
            target_breakpoints ← scope.get_assignments(symbol);
            foreach bp ← target_breakpoints do
                bp.is_watchpoint ← True;
                inserted_breakpoints ← inserted_breakpoints ∪ { bp };
            end
        end
        else
            inserted_breakpoints ← inserted_breakpoints ∪ { breakpoint };
        end
        if breakpoint_type = Assertion then
            // Assertion should always trigger by construction;
            breakpoint.assertion ← True;
        end
    end
    // Sort inserted breakpoints based on the breakpoint ordering;
    breakpoints_ordering ← get_breakpoints_ordering();
    sort(inserted_breakpoints, breakpoints_ordering);
```

Algorithm 3.11: Algorithm that handles breakpoint insertions.

```
Function SendBreakpoints(breakpoints):
    frames ← ∅;
    foreach breakpoint ← breakpoints do
        frame ← ∅;
        scope ← get_scope_info(breakpoint);
        foreach symbol ← scope.variables do
            value ← Evaluate (symbol.value, breakpoint);
            frame.set(symbol.name, value);
        end
        frames ← frames ∪ { frame };
    end
    RpcToClient(frames);
    WaitCommand();
```

Algorithm 3.12: Algorithm to reconstruct the stack frame once a breakpoint hits and send it to users.

Chapter 4

Hgdb: One Framework to Debug Them All

We built a prototype debugging infrastructure called Hardware Generator Debugger (hgdb) that implements the interface primitives described in Chapter 3. We leveraged many existing frameworks to help us reduce engineering effort, and created a high performance simulator runtime to reduce debugging overhead. It turned out that creating a high quality symbol table was the most difficult task in this project because many frameworks were not designed for debugging. Producing symbol tables required extensive modifications and workarounds in the generator compiler backends.

The first part of this chapter describes the construction and performance of hgdb. In particular, we discuss how hgdb realized the simulator interface primitives, why we made certain design choices in the symbol table interface implementation, and how the debugger interface was implemented. Thereafter, we explain how we extracted symbol tables from popular hardware generator frameworks such as Chisel, Circt, and Xilinx Vitis HLS. Finally we discuss the limitations of our approach.

4.1 Simulator primitive implementation

Hgdb implements the complete set of simulator interface primitives described in Section 3.1. Since we want to maximize compatibility among major simulators, the first challenge is to select which standard/API to interface with the simulator. As we show later in this section, regardless of what standards we choose, RTL simulators are complex software systems, and that means that there are always cases in which the standard implementation will vary. As a result, we need a system to handle standard implementation nuances. The second challenge was how to implement the system efficiently to avoid simulation overhead. Since some primitives are executed every clock cycle, even a tiny overhead for each primitive could result in significant simulation slow down.

After prototyping with different standards and API interfaces, we chose to use VPI. There were several benefits of using VPI. First, unlike DPI, where the simulator only provides a compatible layer to translate between SystemVerilog primitive types and C primitive types, VPI defines a complete model for all SystemVerilog constructs. As a result, applications can use VPI to access the design during simulation, such as querying the design hierarchy, obtaining signal types and values, and placing callbacks on various simulation events. Second, as shown in Table 3.1, all major simulators support some level of VPI routines. Another option would be using the Tcl interface provided by individual simulators. However, this interface is slow and not portable, and not all simulators have it, (Table 3.1). Based on our measurements, using the Tcl interface would slow down the simulation by a factor of 5 for those that support it.

We started by implementing primitives based on the SystemVerilog VPI specification, i.e., we used the VPI routines according to the semantics defined by the standard. We quickly found that some VPI routines were either not supported or had different semantics than the specification. For instance, at the time of writing, Verilator does not support obtaining module definition names and does not allow iteratively walking down the module hierarchy given any instance handle; it only supports walking down from the top. In addition, commercial simulator A and commercial simulator B returned different results on wire definition inside an interface because they had slightly different VPI object models.

To solve this problem, we implemented a shim, i.e. a compatibility layer that translates VPI requests into vendor-specific VPI routines. It abstracts away the minor difference between different simulators. For instance, commercial simulator A and Verilator often returns different types for a given RTL symbol, and the shim takes the difference into account. Another benefit of using a shim is allowing custom API functions to be introduced into the system without modifying the simulators. For instance, there is no officially defined VPI routine that reverts time, which means we have to introduce an API to the shim layer, called `vpi_rewind`. The function signature is shown in Listing 4.1. Instead of just taking target time, `vpi_rewind` also takes a set of signals. The reasoning is that time provided by users may be misaligned, that is, set between two clock edges. Using the provided signals allows the underlying simulator to rewind the simulation time to when the signals changed. We also set restrictions on the result time to be no greater than the current time. Therefore, calling this VPI routine with current time − 1 and clock signals yields the previous clock edge. If the underlying simulator does not support this VPI routine, the routine returns `false` and we would inform users that reverse debugging was not fully supported.

Through trial and error, we eventually determined a small subset of VPI routines that worked with all simulators we wanted to support. Table 4.1 shows the set of VPI routines used by hgdb, with most of them being self-explanatory. Note that we call function `vpi_get_vlog_info` to detect the type and version of the underlying simulator and use that information to create the basis of the sim layer.

```
struct rewind_data {
    // threshold time (the reversed timestamp has to be strictly < the given time)
    uint64_t time;
    // clock_handle
    std::vector<vpiHandle> clock_signals;
};

// return true only if the operation is successful;
// false if the underlying simulator does not support it
bool vpi_rewind(rewind_data *reverse_data);
```

Listing 4.1: Header definition for `vpi_rewind` primitive. This was our custom VPI routine and was defined in the VPI shim layer.

VPI routine name and syntax	Synopsis
`vpi_get(prop, obj)`	Get the value of an integer or Boolean property of an object.
`vpi_get_str(prop, obj)`	Get the value of a string property of an object.
`vpi_get_time(obj, time_p)`	Retrieve the current simulation time.
`vpi_get_value(obj, value_p)`	Retrieve the simulation value of an object.
`vpi_get_vlog_info(vlog_info_p)`	Retrieve information about SystemVerilog simulation execution.
`vpi_handle(type, ref)`	Obtain a handle to an object with a one-to-one relationship.
`vpi_handle_by_name(name, scope)`	Get a handle to an object with a specific name.
`vpi_iterate(type, ref)`	Obtain an iterator handle to objects with a one-to-many relationship.
`vpi_put_value(obj, value_p, time_p, flags)`	Set a value on an object.
`vpi_register_cb(cb_data_p)`	Register simulation-related callbacks.
`vpi_remove_cb(cb_obj)`	Remove a simulation-related callback registered with `vpi_register_cb()`.
`vpi_scan(itr)`	Scan the SystemVerilog hierarchy for objects with a one-to-many relationship.

Table 4.1: A list of VPI routines used by hgdb. The explanation is adapted from LRM [1]. Some VPI routines have only partial support, and we thus implemented some heuristic-based workarounds in the shim layer.

Algorithm 4.1 shows how we implemented `get_signal_value` using VPI routines. Note that most VPI routines only take in variables of the `vpiHandle` type, which is a pointer to a simulator's internal data structure. Since the symbol table only stores the signal path relative to the generated

IP, we map the signal path to the full hierarchical name. After obtaining the full hierarchical name, we call the function `vpi_handle_by_name` to obtain the handle and used it for subsequent calls. Appendix A shows the algorithms to implement other simulator primitives using VPI routines.

```
Function get_signal_value(sginal_name):
    handle ← vpi_handle_by_name(signal_name);
    if handle ≠ null then
        value_obj ← initialize_value();
        vpi_get_value(handle, value_obj);
        return value_obj.value;
    end
    else
        return null;
    end
```

Algorithm 4.1: How to implement the simulator interface primitive `get_signal_value` using VPI routines. `null` is returned if the signal name is invalid.

4.2 Instance mapping and concurrent semantics

The simulator interface covers the algorithms for mapping a generated IP to an instance inside the test environment. However, there are some challenges in practice. First, certain simulator backends, such as Verilator and some waveform formats, do not provide information about module definition. As a result, Algorithm 3.1 does not work in these environments. Second, the top-level design may contain multiple instances of the generated IP block. This implies that the framework needs to handle multiple symbol tables at once, each one mapping to a different instance.

To solve the first problem, we realized that the structure of most complex IP blocks are unique. That is, it is almost impossible to have exactly the same signal names and module hierarchy between two IP definitions. As a result, in the case of missing module definition names, we use the signal hierarchy to fingerprint the generated IP blocks. In addition, regardless of the location where the generated IP instantiates, its internal signal hierarchy remains the same. For instance, suppose we use a generator to produce a design called "GCD", and we have a signal with the path `GCD.inst1.inst2.signal_a`. Upon instantiation, say inside top.dut, the module instance will have path `top.dut.ip1`, and the signal has the path `top.dut.ip1.inst1.inst2.signal_a`. Note that the relative path `inst1.inst2.signal_a` is unchanged. If we find the identical signal structure in the complete design, we will have finished the mapping process, as shown in Algorithm 4.2. We first compute the most complex signal names, i.e., longest signal name and deepest hierarchy. The intuition behind this approach is that the deeper the hierarchy, the more complex the structure is. Thus, the more unique the design, the less likely it is for the algorithm to yield false matches. Then we walked through the design hierarchy to check each signal's full hierarchy. If our most complex

```
// Get the most complex signal names;
breakpoints ← get_breakpoints_ordering();
max_signal_size ← 0;
max_signal_name ← null;
foreach breakpoint ∈ breakpoints do
    scope ← get_scope_info(breakpoint);
    foreach signal ∈ scope.signals do
        // Complexity is computed from the full hierarchical name;
        size ← signal.complexity();
        if size > max_signal_size then
            max_signal_size ← size;
            max_signal_name ← signal.name;
        end
    end
end
// Use the most complex signal name to match with the test environment;
foreach instance ← get_design_hierarchy() do
    foreach signal ∈ instance.signals do
        if subset(signal.name, max_signal_name) then
            // We have found the match and obtain the mapping by subtracting
            common signal path;
            mapping = signal.name \ max_signal_name;
        end
    end
end
```

Algorithm 4.2: Algorithm to map generated IP block to the test environment hierarchy, where the module definition is not available. We exploited the fact that relative hierarchy remains the same regardless of the instantiation. Function `subset` returns `true` if a signal is subset of another signal.

signal (excluding the top module name) was a substring of a particular signal, we have found the mapping.

In a case where the same or multiple different IP blocks are instantiated multiple times, we need a method of managing all the mappings. This corresponds to the top level design using multiple instances of the same generated design. In this case, we really have a parallel execution situation, and we use a concept of processes to isolate the IP blocks with their own mapping schemes and symbol tables. This is similar to a modern operating system in which the virtual memory mapping is local to the process itself. During the instance mapping stage, we create a new process whenever we locate a matching instance. If users insert a single breakpoint whose source location is matched with multiple IP blocks, then hgdb will insert one logical breakpoint for each matched process. The breakpoint expression evaluation and the stack frame reconstruction follows naturally since the RTL signals are mapped differently across different processes.

Thanks to the process-based implementation, we introduce a limited "concurrency" to the source-level, even though the underlying simulator is single-threaded. This approach is similar to multi-threading in software. In the latter case, when a breakpoint pauses the execution, the debugger can display the execution states of all threads. This information is useful to debug race conditions and bugs in cooperative concurrent programming. However, in software, precise interrupt is difficult to implement in a multithreading environment, whereas it is straightforward in RTL simulation. This is because we pause only after the simulation stabilization, and all the processes' state information is correct. We assigned unique IDs to each process so that users can switch to different processes to find the instance of interest during debugging inside a GUI, as shown in Section 4.6.

An extension of process-based design is to support clock domains. Here, we introduce the concept of *namespace*, which consists of multiple processes that share the same clock. During debugger runtime initialization, we place the emulation loop callback on each namespace's clock. The namespace can be inferred automatically through VPI function calls, i.e., querying the wiring connection, without the input from the symbol table. Although the clock query can be expensive if the testing environment is complex, we only query once during initialization and thus it does not affect the overall simulation performance.

4.3 Waveform replay

Since all the simulator primitives were implemented with VPI, the easiest approach for enabling a waveform replay engine to interface with hgdb is to implement a VPI-based shim layer. The shim layer emulates the complete set of VPI routines listed in Table 4.1. As a result, the debugger runtime treats the waveform replay engine as another simulator.

We recognized that there are different formats of waveforms, and some of them have proprietary parsers and APIs, e.g., FSDB from Synopsys. In other words, each waveform format needs its own parser and adapter. As with the VPI shim layer that we provided to emulate VPI routines, we also define a waveform layer to abstract away the differences between various waveform formats. The interface primitives are defined in Table 4.2. Note that certain primitives are optional because some waveform formats might provide the relevant information.

Because the VPI specification treats vpiHandle as a pointer that can be allocated and freed, the VPI emulation layer manages vpiHandle allocation and thus abstracts away the VPI details from the waveform database. Signal and module instances are referred to by their unique IDs as defined by the waveform database, and the translation is performed at the interface layer. Algorithm 4.3 shows how to use waveform interface primitives to implement `vpi_get_value`. We first translates the vpiHandle into a unique signal ID that is used to refer to a particular signal in the waveform. Then we query the waveform database using the current time to get the desired value.

Waveform interface primitive name and syntax	Synopsis
get_instance_id(name)	Get the instance id given the hierarchical name of an instance
get_signal_id(name)	Get the signal id given the hierarchical name of an instance
get_instance_signals(instance_id)	Get a list of signal definitions given an instance
get_signal_value(signal_id, timestamp)	Get the signal value at a particular time
get_next_value_change_time(signal_id, base_time)	Get the time in the future when the signal changes its value
get_prev_value_change_time(signal_id, base_time)	Get the time in the past when the signal changes its value
get_instance_definition(instance_id)*	Get the instance definition from an instance id
top_instance_id()	Get the instance id of the top instance

Table 4.2: A list of Waveform interface primitives. It is designed to work with various waveform formats such as VCD and FSDB. Functions marked with * are unavailable for VCD format.

```
// Internal data structure that maps VPI handles to signal IDs;
static handle_to_signal;
static current_time;
Function vpi_get_value(handle, value_p):
    signal_id ← handle_to_signal.get(handle);
    value ← get_signal_value(signal_id, current_time);
    value_p.value ← value;
```

Algorithm 4.3: How to implement `vpi_get_value` using waveform interface primitives.

To implement callback related VPI routines, the VPI emulation layer keeps track of state information such as time and current signal values. Time flows “forward” as the emulation layer continues to read out the waveform data, and the layer triggers a callback if it detects signal value changes or enters simulation-related events.

4.4 Simulation loop and performance optimization

Hgdb implements the simulation loop, as described in Section 3.4. While the pseudocodes give a solid guideline for implementing various functions, there are many potential places where we can optimize for performance. Some optimizations might be critical, whereas other optimizations might be optional as they do not lie on the timing-critical path. Some optimizations might even be premature and potentially downgrade the performance.

Our debugging system introduces two types of overhead to the system: 1. overhead in the simulation loop due to evaluation of the breakpoint enable condition; and 2. overhead when interacting

with users, such as creating breakpoints and constructing stacks. These two overheads have different impacts on the system's performance. For instance, several milliseconds overhead per simulation cycle will be significant if the simulation runs hundreds of clock cycles per second. However, hundreds of milliseconds of delay when users set breakpoints or receive a reconstructed stack will not change the users' perception of the overall system's performance. The reason for this discrepancy is that the human's response time is on the order of hundreds of milliseconds. As a result, we should shift the performance overhead to the user interaction time whenever possible.

As shown in Algorithms 3.11 and 3.12, the debugging system translates signal names to vpiHandles when it interfaces with the simulator. An obvious performance optimization is to exploit the locality so that the same signals used previously can likely be used again in the future, such as clock signals and those used in breakpoint enable conditions. We could implement a cache layer in the runtime that maps signal names to vpiHandle, thus reducing the traffic to the simulator. However, vpiHandle translation only happens when the debugger is constructing the stack frame using the symbol table or when the breakpoint enable condition needs to be evaluated. Performance in the first situation matters little since users are interacting with the debugger. In the second situation, although implementing a cache does avoid querying the simulator for vpiHandles, we can do even much better.

One simple solution is using better data structures. When we parse the enable condition and generating the expression tree, we store the vpiHandle corresponding to the symbol directly in the expression tree node after expression validation. When we evaluate expressions during simulation, we directly fetch the vpiHandle and query its value, instead of going through the translation. In other words, we pay the one-time cost of translating vpiHandle when users set a breakpoint and thus guarantee optimal performance inside the simulation loop. As a result, there is no need to implement a cache layer. Such a layer is sub-optimal compared to the proposed approach since it requires paying the cost of the cache query. Figure 4.1 shows the improvement from using a cache layer (baseline) to using a better expression tree data structure (optimized).

Once we had optimized the most obvious aspects, we needed profile information to better understand any performance bottlenecks. Figure 4.2 shows the overhead breakdown in the debugger runtime on RocketChip benchmark. To profile different components properly, we inserted eight breakpoints in the system, whose enable condition was equality comparison of a one-bit signal to 2, which was always false. We chose such an expression so that it would not block the simulation; otherwise we would have needed a debugger frontend to send a continue command. The always-false expression also exercised all the code paths inside the runtime.

Based on the overhead breakdown, it does not make sense to implement advanced features such as Just-in-Time (JIT) to speed up the expression evaluation. Most of the time is spent querying values from the simulator. However, it does offer an opportunity for parallelism since obtaining values from the simulation and evaluating the expression are independent tasks, i.e. an embarrassing parallelism.

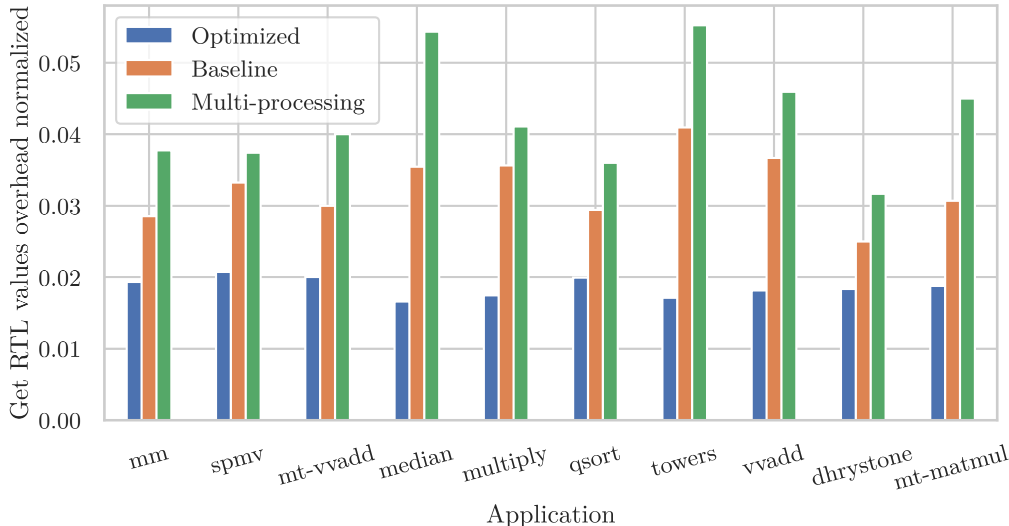


Figure 4.1: Performance improvement after optimizing expression tree data structures. Eight breakpoints were inserted, whose enable condition evaluates to false at runtime. Instead of querying handles during breakpoint evaluation, we tagged the vpiHandle to the expression node when users inserted a breakpoint. We also compared the performance where multiple-processing was introduced. Time was normalized to simulation time without any modification.

However, this is only beneficial if the underlying simulator does not use any mutable shared data structures inside the VPI implementation. Among the high-performance simulators, only Verilator is open-source and could be verified to conform to such a property. Other commercial simulators gave segmentation faults unless a mutex lock is used. Figure 4.1 shows the performance slowdown on a commercial simulator, where two hardware threads are used. In the commercial simulator, because getting values from the RTL is more expensive than evaluating expressions, the two threads compete with each other to obtain the mutex lock. Hence the mutex competition results in performance slowdown. In contrast. Verilator does not suffer from this issue, since its VPI implementation (the code path we care about) is free from side effects. Based on our preliminary experiments with RocketChip, we saw about 30% reduction in overhead when we adopted multi-threading without a mutex lock using Verilator.

4.5 Symbol table design and implementation

As mentioned in earlier sections, symbol table interaction is not on the timing-critical path of the runtime loop, because whenever the runtime queries the symbol table, the simulation is paused for users to interact. This implies that the performance of the symbol table is not the highest priority; rather, we should focus on making it easier for hardware generator frameworks to generate the

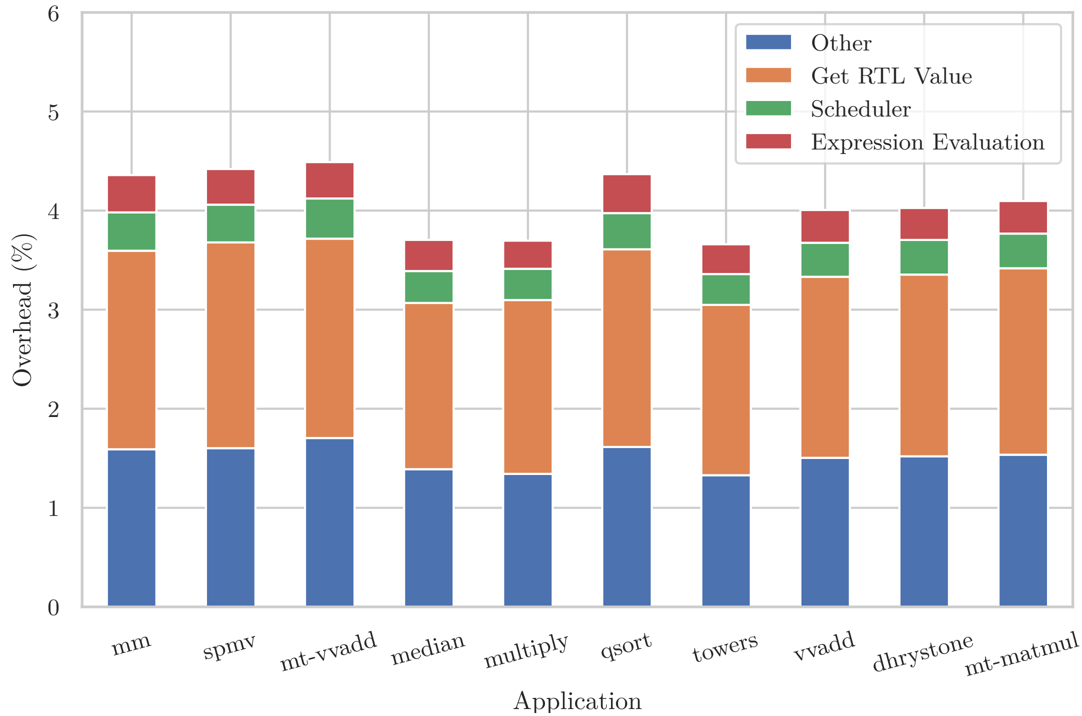


Figure 4.2: Simulation overhead breakdown with eight breakpoints inserted. The time is normalized to simulation time without any modification. Note that more than half of the overhead os from getting values from the simulator. The miscellaneous time is calculated by subtracting the overhead of known categories from the total observed wall clock overhead. Most of this time is due to CPU context switching and other tasks.

symbol table.

We focused on three different aspects of the symbol table design: 1. the structure of the symbol table, 2. the type of information stored in the symbol table, and 3. the communication interface of the symbol table. All the design decisions regarding these aspects were aimed at reducing the complexity of the symbol table implementation so that the generator compiler could easily produce the table.

The first aspect determines whether the symbol table will be structured or semi-structured. The benefit of a structured symbol table is that one can leverage many existing structured databases, e.g., relational databases. Our prototype started with SQLite, one of the most popular standalone relational databases [42]. A simplified SQLite schema is shown in Figure 4.3. The main idea behind the schema is to associate all the debug information with the breakpoint. Once a source location is provided for query, we follow the relations to obtain scope variable definitions as well. An example SQL query that implements the symbol table primitive `get_scope_info` is shown in Listing 4.2.

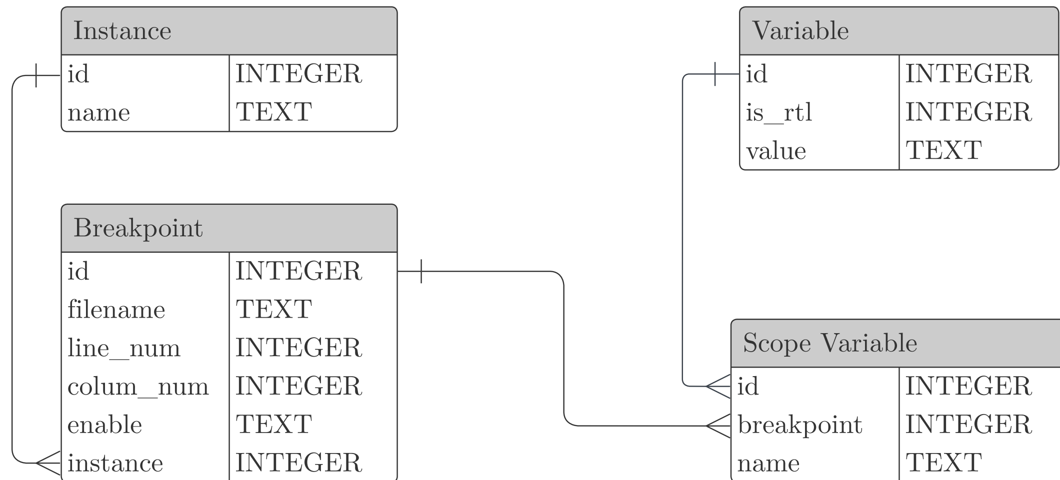


Figure 4.3: Simplified schema diagram for SQLite-based symbol table. Arrows in the figure illustrate relations, which can be used to improve the search performance and guarantee data integrity.

Note that we utilize the underlying relational database to handle a complex search. The SQLite engine followed the column relations, and breakpoints and instance IDs are naturally linked and searched.

```
SELECT
  "scope_variable"."id",
  "scope_variable"."name",
  "variable"."value",
  "variable"."is_rtl",
  "instance"."name"
FROM
  'breakpoint',
  'scope_variable',
  'instance',
  'variable'
WHERE
  "scope_variable"."breakpoint_id" = "breakpoint"."id"
  AND "scope_variable"."variable_id" = "variable"."id"
  AND "instance"."id" = "breakpoint"."instance_id"
  AND "breakpoint"."id" = ?
```

Listing 4.2: SQL query to implement `get_scope_info` given a breakpoint ID, using the table schema shown in Figure 4.3. We assumed the scope information was flattened and all the context variables were stored to every potential breakpoint.

A SQLite-based approach makes the symbol table query much easier to write, as the underlying database handle the majority of the implementation. However, we soon realize that it also makes the symbol table rather difficult to generate. For instance, when dealing with nested scopes in which variables had different lifetimes depending on their lexical location, we either have to flatten the scopes into a single scope for each statement or chain the scopes together through relations and recursively query. Flattening the scopes makes the symbol table much slower to generate and query, since it exponentially increases the table size. Recursively-defined scopes make the query slower, because it is challenging to fine-tune SQL-based databases to efficiently handle long recursive relations. We eventually implemented a JSON-based symbol table interface. Instead of using parent scope via relations, JSON-based symbol tables allows nested structures naturally, which is similar to the popular Linux native symbol table DWARF. Listing 4.3 shows the symbol table for Listing 3.1 following the JSON schema. Variable declarations and statements are stored inside the scope, which can also nest inside another scope. This design follows the source programming languages that frequently create new scopes, such as Python and C/C++. To generate such a JSON symbol table, the compiler walks the source-level abstract syntax tree (AST) and generates the JSON along the way. The obvious drawback is that we must implement our own indexing and search algorithms to implement the primitives. Algorithm 4.4 shows the pseudocode that implements the symbol table primitive `get_scope_info`. Note that we assume the scopes are already sorted by their lexical positions and variables defined later override the previous definition. During frame reconstruction, we walk the scope backward and ignore any duplicated variable declarations and assignments.

The second aspect determines how much information the compilers need to produce. In CPU-native symbol tables, the register size is fixed by the architecture and the symbol table has to store the actual primitive data type. In contrast, most simulators already store such information, because of the VPI object models defined by LRM [1]. As a result, we use VPI routines to obtain the signal types and the symbol table only needs to store the mapping. However, if the data types cannot be natively represented by SystemVerilog types, we define a way to translate from the source-level types to SystemVerilog types. We use the "dot notation" where the nested structures are flattened by joining the names with "`.`" between them. Table 4.3 shows examples of how the flattening is done. The dot notation works well with any text based representation, such as the JSON format, and it applies to all source-level data types.

The last aspect is the communication between the runtime and the symbol table. Both SQLite and JSON implementations are built into hgdb so the tables can be handled natively during debugging. However, since performance is not a major concern, we implement an RPC layer that allows any type of symbol table to communicate with the simulator runtime. Abstraction is again the key to achieving this goal. Hgdb defines an API that supports both TCP and WebSocket. The remote symbol table needs to communicate with the runtime using the predefined data format namely JSON. The communication flow is similar to that of a web application, which makes the symbol

```c
1 int sum_odd(int input[4]) {
2     int sum = 0;
3     for (int i = 0; i < 4; i++) {
4         if (input[i] % 2) {
5             sum += input[i];
6         }
7     }
8     return sum;
9 }
```

```json
{
  "type": "module",
  "name": "sum_odd",
  "scope": [{
      "type": "block",
      "filename": "sum.c",
      "scope": [{
          "type": "decl", "line": 1, "column": 17,
          "variable": {"name": "input", "value": "input", "rtl": true }
        }, {
          "type": "assign", "line": 2, "column": 9,
          "variable": {"name": "sum", "value": "sum0", "rtl": true}
        }, {
          "type": "block",
          "scope": [{
              "type": "decl", "line": 3, "column": 14,
              "variable": {"name": "i", "value": "0", "rtl": false}
            }, {
              "type": "assign", "line": 6, "column": 4,
              "variable": {"name": "sum", "value": "sum1", "rtl": true}
            }, {
              "type": "decl", "line": 3, "column": 14,
              "variable": {"name": "i", "value": "1", "rtl": false}
            }, {
              "type": "assign", "line": 6, "column": 4,
              "variable": {"name": "sum","value": "sum2","rtl": true}
          }]
      }]
  }]
}
```

Listing 4.3: Symbol table for Listing 3.1 (copied over here) represented in JSON (only two loop iterations are shown). Note that each statement is wrapped inside a scope entry, which can be arbitrarily nested inside other scope entries.

table relatively easy to implement. Figure 4.4 shows an example of how the symbol table primitive `get_scope_info` is implemented with RPC.

```
Function get_scope_info(scope):
    frame ← ∅;
    while scope ≠ null do
        if scope.type ∈ { "decl", "assign" } then
            var ← scope.variable;
            if var.name ∈ frame then
                continue;
            end
            frame.set(var.name, var);
        end
        next_scope ← scope.prev_scope();
        if next_scope = null then
            // We have reached the top of the current scope.  Need to go one
            level up;
            scope ← scope.parent();
        end
        else
            scope ← next_scope;
        end
    end
```

Algorithm 4.4: Algorithm to implement `get_scope_info` using a JSON-based table. Note that we optimize the data structure in the memory to quickly query previous entries given a scope entry.

Source-level symbol name	RTL name	Source-level type	RTL type
`array.0`	`array_0`	Array subindex	Net
`struct.a`	`struct_a`	Aggregated sub-field	Net
`array`	`array`	Array	Array
`structure`	`structure`	Aggregated	struct

Table 4.3: Example rules for signal flattening. The idea is to use `.` to flatten out any nested data structures for the source-level symbol name. Note that if the data structure is maintained from the source level to RTL, e.g., a source-level array is lowered to a RTL array, no flattening is required.

4.6 Debugger frontend design and implementation

Since hgdb supports remote debugging, we needed to define an interface and protocol layer so that the frontend described in Section 3.3 could communicate with the runtime. Instead of implementing the RPC channel using binary packets, we leveraged many existing web frameworks due to their popularity and wide support. For instance, each debugger frontend primitive is serialized to JSON and communicated via WebSocket, which is supported by many UI frameworks. Listing 4.4 shows how primitive `set_breakpoint` is serialized. In the example, we add a breakpoint at `example.py:10` with the condition that the clock must be high.

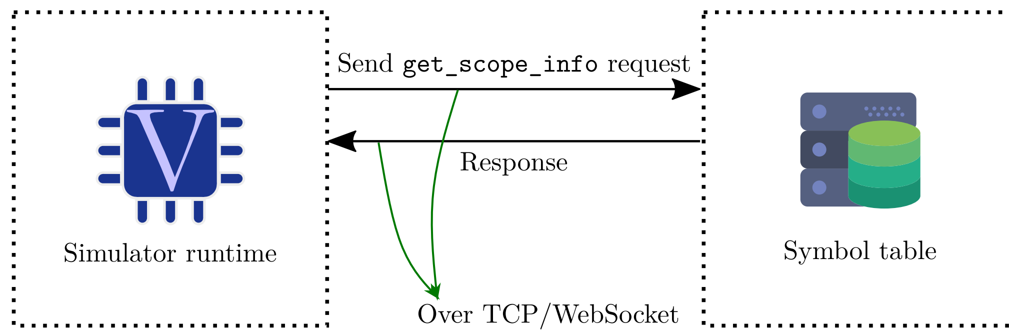


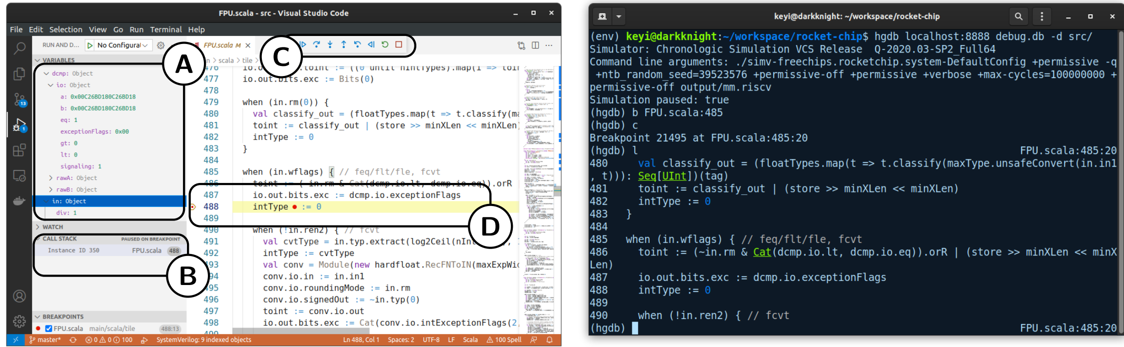
Figure 4.4: How get_scope_info is implemented over a network channel. The underlying transport layer should not matter, and hgdb supports both TCP and WebSocket.

```
{
    "request": true,
    "type": "breakpoint",
    "payload": {
        "filename": "example.py",
        "line_num": 10,
        "action": "add",
        "condition": "clk==1"
    }
}
```

Listing 4.4: An example of debugging protocols in JSON.

We implemented two debugger frontends, one based on Visual Studio Code (VSCode) and one based on the console, as shown in Figure 4.5a and Figure 4.5b respectively. VSCode is a popular open-source integrated development environment (IDE) that is extensible. It defines a protocol called Debug Adapter Protocol (DAP) [43] that allows developers to implement custom language debuggers inside the IDE. We leveraged the DAP interface and translated our remote debug protocols to DAP. We also mimicked the designs of gdb in our console-based debugger frontend. Text-based commands are parsed and translated into the remote debug protocols.

In retrospect, adopting web standards significantly reduced our implementation time and allowed us to reuse prior works. For instance, the console-based debugger has less than one thousand lines of code because of Python's many popular web server-based packages and asynchronous language features. The same approach also allowed us easily to test our web API using established frameworks.



(a) Visual Studio Code-based debugger. (A) Values of local and generator variables fetched from the simulator. (B) Concurrent hardware threads executing of the same line. (C) Simulation state control such as continue, step over, and reverse-step. (D) Source- and conditional-breakpoints.

(b) Console-based debugger.

Figure 4.5: Debuggers provided by hgdb.

4.7 Case study on real world designs

To quantify the gain in debugging productivity related to the use of source-level debugging, we compared the debugging experience with and without hgdb on a real world coarse-grained reconfigurable architecture (CGRA), as shown in Figure 4.6. Some IP blocks were designed in a prototype language, called kratos, that fully supports source-level debugging. Kratos itself is embedded in Python. We discuss the design philosophy and implementation details further in Section 5.3.

The CGRA we were interested in is a reconfigurable two-dimensional island-style fabric, in which the tiles are connected via a statically configured interconnect. In each tile, there are one or more cores, which can be configured to perform certain tasks. For instance, the processing element (PE) core has an arithmetic-logic unit (ALU) that is capable of performing arithmetic operations such as addition and multiplication; the memory (MEM) core was a memory controller with fixed-sized SRAMs that can store and fetch data given pre-configured access patterns. The interconnect is a reconfigurable mesh network that connects tiles together. In addition to the fabric, the CGRA also has a secondary-level cache called the "global buffer" that handles the data streaming between the fabric and the embedded ARM CPU. The IP blocks written in kratos are shown in □. There are multiple instantiations of different kratos IP blocks in the design, due to the semi-homogeneous nature of the CGRA design.

To run an application on the CGRA, we first compile the application graph into a configuration bitstream. In the application graph, each node corresponds to either a PE or MEM core; each node also contains instructions pertaining to the corresponding cores. The graph connection is realized via interconnect routing. In other words, the bitstream is essentially a collection of instructions that

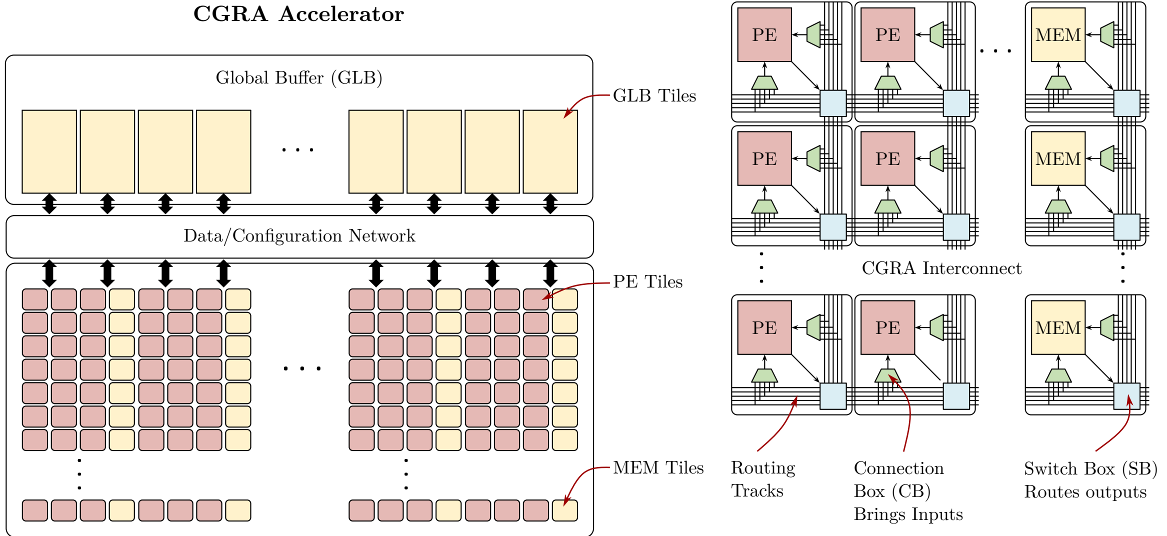


Figure 4.6: CGRA architecture diagram. Processing element (PE) and memory (MEM) tiles are connected through a mesh-like interconnect. It also has a cache block called global buffer that controls configuration and streams in/out data. Blocks written in kratos are shown in □ .

specifies the fabric computation logic. Once we load the bitstream and input data into the main memory, the global buffer streams out the configuration to the fabric configuration bus, then pushes the input data to the array and writes out the array outputs. We can then compare the result against a validated software model.

4.7.1 Bugs in specialized CGRA for sparse tensor algebra

A key advantage of the CGRA is that it allows architects to specialize the cores for specific application domains, i.e., designers can add specialized instructions to the core so that the overall execution of specific applications is more energy efficient. This process is called "hardening" in the context of reconfigurable architecture.

In one of the specialized CGRAs, the designers optimized the PE and MEM for the execution of generic sparse tensor algebra. The goal was to execute the entire space of expressible tensor algebra operations or kernels under many different schedules by adding specialized sparse primitives. Since the sparse tensor algebra uses a compressed tensor format, the exact computation schedule is input data dependent and can not be determined at compile time. To handle the dynamism the designers added a ready-valid protocol to the interconnection network.

To improve performance in these sparse applications, the designers tried to hide the memory latency by dividing the memory into smaller blocks and overlapping the memory loading with computation. In other words, we load the CGRA MEM tile data for the next input block while the

CGRA fabric is working on the the current input block. The memory controller thus requires different commands for different actions. To reduce the number of bits used in the control logic, the allocation and finalizing data block commands shares the same opcode. As a result, the action of the operation is interpreted based on the current controller state. That is, if the controller just allocated the block, the allocate/finalizing instruction finalizes it, and vice versa.

In an application test that involved the memory write and execution overlap, the designers noted an incorrect output from the CGRA. After more than 3 hours of reading waveforms and going back and forth with fix attempts, the designers finally identified the issue. There was a bug in the memory controller that managed the state to receive, interpret, and discard incoming memory write options. It did not properly discard the "finalize" instruction from a stream. Instead, the controller double-counted the finalize instruction and used the duplicated instruction to allocate a new block, since they shared the same opcode. When the real allocation instruction arrived, the memory controller believed it was finalizing a block of size 0, and all subsequent accesses to that new block were invalid and returned invalid data.

Two factors made this bug difficult to find. The simulator the designers used, namely Verilator, by default zeros out uninitialized memory instead of `x`. As a result, downstream controllers that read zeroed out data emitted additional bogus blocks of data that were indistinguishable from legitimate ones. The second factor was that the input data were mostly sparse or hyper-sparse tensors. It was highly likely that two input streams shared no common coordinates, and so the output was an empty stream. This meant that this bug manifesting valid, empty streams would not necessarily be detected by arbitrary data inputs. It would produce errors only with data inputs that have overlap with this error case. In other words, many data input patterns would not expose this bug at all. Furthermore, the lack of exposure meant designers had to examine the downstream logic slowly—and somewhat painstakingly—all the way back to the source unit.

To debug the same issue with hgdb, we first captured the waveforms and replayed them using the built-in replay tool. Since we did not know where or when the bug happened, we first set a watchpoint on the final output and set the triggering condition to be true only when the output was not zero. It never triggered as a result of empty streams. The next step was to set various conditional breakpoints on the stream interface controller where it controlled the memory operations. The conditions ensured the breakpoint was triggered only if the designer expected the control signal to be high but in the simulation it was not. Figure 4.7 shows an example of a conditional breakpoint that check the FSM output. We expected the FSM output logic, `self._pop_in_full[ID_idx]` to be low when we were writing to the FIFO. One unique advantage with hgdb is that although there are multiple instantiations and multiple variants of the same generator definition, we only need to set the breakpoint in one place, i.e., the source place. This is because hgdb models all the module instantiations as concurrent threads, as described in Section 4.2. We saw multiple breakpoints being triggered as we replayed the simulation back and forth. Whenever we encountered a false positive,

```
condition: !self._pop_in_full[ID_idx] && self._wr_ID_fifo_out_data == ID_idx

WRITING[ID_idx].output(self._pop_in_full[ID_idx],
                       (self._mem_acq[2 * ID_idx + 0] &
                        self._joined_in_fifo & (self._wr_data_fifo_out_op == 1) &
                        (self._wr_addr_fifo_out_data <
                        (self._buffet_capacity[ID_idx] -
                        ↪  self._curr_capacity_pre[ID_idx])) &
                        (self._wr_ID_fifo_out_data == ID_idx)) |
+                      (self._joined_in_fifo & (self._wr_data_fifo_out_op == 0) &
+                       ~self._blk_full[ID_idx] &
+                       (self._wr_ID_fifo_out_data == ID_idx)))
```

Figure 4.7: Bug fix for the sparse controller FSM to ensure we do not double count the block allocation command. We added additional conditions to control the FIFO pop signals (shown in green background with leading +)

we added more conditions to the breakpoint to avoid false alarms. After a couple of rounds of filtering, we identified the situation where the FIFO pop signal was not set to high, i.e., where the bug was. The highlighted lines in Figure 4.7 show the bug fix, where we added additional logic to ensure the pop signal was high. The whole process took less than 20 minutes, which was about 10× faster than the waveform method.

4.7.2 Bugs in the global buffer

The global buffer is the cache between the CGRA and the CPU. Unlike traditional CPU caches where the content is stored based on memory addresses, the global buffer store data based on tasks. For each compute task, the global buffer streams in input and configuration data and streams out output data from the CGRA. The architecture of the global buffer is shown in Figure 4.8. Note that there are two DMAs in each global buffer tile to transfer data between the global buffer and the CGRA compute fabric.

Since we had introduced sparse tensor algebra to the compute fabric, we needed to add features to the global buffer so that the CGRA could send dynamic memory read/write requests to the global buffer. To implement this feature, we added a ready-valid handshake mechanism to the the communication channel between the global buffer and the CGRA fabric. The adjustment was achieved partly by adding FIFOs to the communication channels to deal with the memory transfer. In other words, the global buffer had to be retrofitted to support the ready-valid interface with the CGRA fabric, which is error-prone.

During an integration test after making the changes, the designers noted failed transactions between the global buffer and the CGRA: the FIFO positioned between the global buffer and the

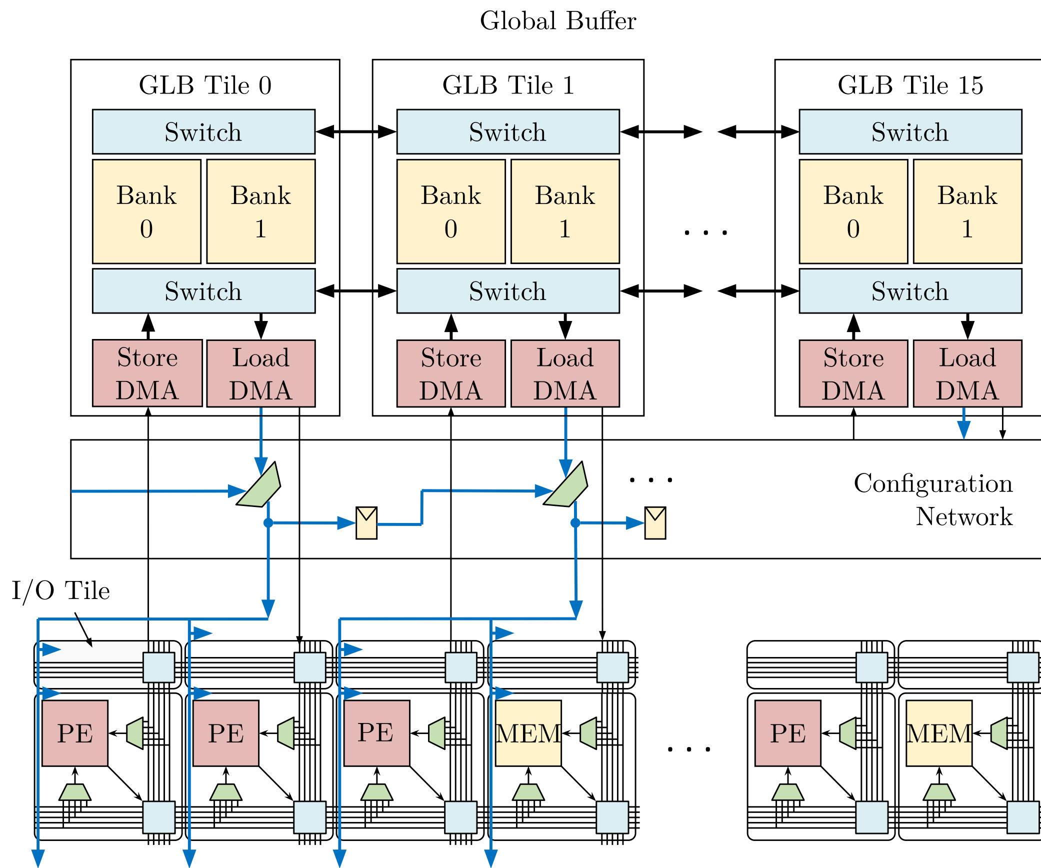


Figure 4.8: Architecture diagram for the global buffer.

CGRA was streaming garbage results to the CGRA. This is challenging to debug because the bug could either be in the address generator or the FIFO itself. For instance, the address generator could send a wrong memory request to SRAM or the FIFO could drop/corrupt data when the response data arrived at the FIFO.

Upon closer examination, the designers discovered that the address generator engine inside the load DMA that sent requests to the SRAM misbehaved when the FIFO was almost full. Ideally, when the FIFO was full, the address generator should stop sending requests to the SRAM. However, it could take multiple cycles for a memory request to arrive at the FIFO because of the chained GLB tiles. This meant that there could be a number of in-flight/outstanding requests even after the address generator stopped. To ensure that the FIFO has space for these requests, the address generator must stop before the FIFO is full, a situation called "almost full." The address generator was thus required to use the almost full signal instead of the full signal as a stop token, as shown at Line 5 in Figure 4.9. This was an interface bug that disrupted the communication channel and corrupted the input data, and the designers spent half an hour isolating the cause and issuing a fix.

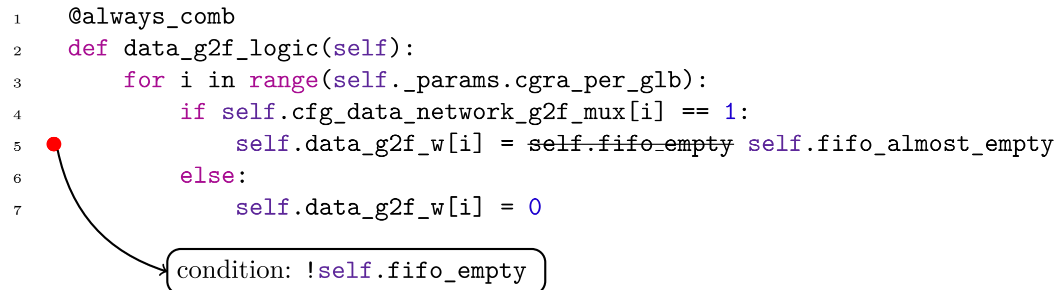


Figure 4.9: Illustration of debugging global buffer at source-level and corresponding bug fix.

The same bug was discovered within minutes when using hgdb. Since the result was incorrect, we set breakpoints in the communication channel that would trigger whenever there was a transaction, either from the global to the fabric or vice versa. After several transactions, we noted immediately that a transaction happened even though the load DMA should have blocked it. The bug was discovered by setting a conditional breakpoint on the transaction control signal, as shown in Figure 4.9.

Although these bugs required only a few changes to fix, they were difficult to find because they manifested late in the integration tests. Even if the RTL was designed by hand using SystemVerilog, it would still have taken significant effort to debug using waveforms, due to the system complexity. Such complexity is partly why designers typically use SystemVerilog debuggers such as Verdi [22] to debug complex designs. Hgdb essentially provides a state-of-the-art debugger that allows designers to debug in the context of the source code instead of the generated RTL. In other words, we expect designers to have similar productivity using hgdb with hardware generators as they would using Verdi for traditional RTL.

4.8 Extracting symbol tables from compilers

A debugging system is not complete without a symbol table, and a correct symbol table is difficult to extract from a system that was not built to create one. Indeed, many hardware generator frameworks were developed without debugging in mind, and we spent most of our engineering effort determining heuristics and implementing symbol table extraction algorithms. Even with such vast effort, the extracted symbol tables remained incomplete. This problem was further exacerbated by the fact that some compiler tool chains are not open-source. We were essentially reverse engineering the black boxes that are locked behind corporate “trade secrets”. These problems are not fundamental, and Chapter 5 provides suggestions on how to build generators with better support for debugging. Nonetheless, the algorithms described in this section work most of the time and can help designers with debugging and verification.

4.8.1 Chisel/Firrtl compiler

Chisel/Firrtl is arguably one of the most popular hardware generators [5, 16]. The frontend language, Chisel, is embedded into a high-level functional programming language called Scala. There are several benefits of using Scala. First, Scala has syntactic sugar that makes the language more expressive than conventional SystemVerilog. Its features include pattern matching and implicit classes. The second benefit is that Scala is functional and object-oriented, making the DSL more flexible and reusable. Using the static typing system, Scala allows users to create complex types through inheritance and apply functional programming techniques, such as map and reduce, to define the logic. Object-oriented programming encourages code reuse and generator parameterization.

Like many other generator frameworks, Chisel first elaborates the design and then produces outputs in a form of IR called Flexible Intermediate Representation for RTL (Firrtl). The benefit of separating the frontend language from the IR is similar to that of programming languages and LLVM: it allows independent development and optimization of the programming languages and compiler. Unlike LLVM, which has only one form, Firrtl has three levels, namely HighForm, MidForm, and LowForm. The Chisel compiler typically emits HighForm, which is later lowered to LowForm to produce RTL inside the Firrtl compiler. Listings 4.5, 4.6, and 4.7 show example Chisel code and its HighForm and LowForm Firrtl IR, respectively. In the example, we compute the output based on the selection and enable bit. Note that HighForm roughly resembles the original Scala code whereas the LowForm is closer to RTL.

Another advantage of using an IR is that it allows compiler passes to transform the logic. Widely used optimization passes, such as dead code elimination and CSE, are built into the Firrtl compiler [16]. Multiple lowering passes are used to transform HighForm into MidForm, and finally to LowForm before the RTL code generation. Information between passes is communicated through metadata called annotations, which are attached to the IR nodes. One compiler pass can attach information to a particular IR node, which is later consumed by another pass. Users can also add additional passes to analyze or transform the design. We implemented the symbol table extraction using Firrtl passes, as shown in Algorithm 4.5.

Algorithm 4.5 has two passes. The first pass annotates the IR, and the second collects the annotation and produces the symbol table. In the first pass, different types of information are included in annotations of the HighForm, such as “source-level” names, statement types, and their enable conditions. Note that Chisel does not maintain a symbol table when emitting Firrtl, which implies that the symbol tables we have in HighForm might not be the same as in the source code (this sometimes does happen). We use a series of string-matching heuristics to recover the original source name. We annotate the IR nodes with “source-level” name so we know the original name after the lowering passes.

For simplicity of implementation, we compute the breakpoint enable condition at HighForm. This is because the source location gets displaced after the `when` construct is lowered into a multiplexer

```
12 class Selection extends Module {
13   val io = IO(new Bundle {
14     val input0 = Input(UInt(32.W))
15     val input1 = Input(UInt(32.W))
16     val en     = Input(Bool())
17     val sel    = Input(Bool())
18     val output = Output(UInt(32.W))
19   })
20
21   io.output := 0.U(32.W)
22
23   when (io.en) {
24     when (io.sel) {
25       io.output := io.input0;
26     } .otherwise {
27       io.output := io.input1;
28     }
29   }
30 }
```

Similar to assignment in `always_comb`

Similar to `if`

Listing 4.5: A 2-input mux with enable bit written in Chisel.

```
circuit Selection :
  module Selection :
    input clock : Clock
    input reset : UInt<1>
    output io : { flip input0 : UInt<32>, flip input1 : UInt<32>, flip en :
     ↪  UInt<1>, flip sel : UInt<1>, output : UInt<32>}

    io.output <= UInt<32>("h0") @[Selection.scala 21:13]
    when io.en : @[Selection.scala 23:16]
      when io.sel : @[Selection.scala 24:19]
        io.output <= io.input0 @[Selection.scala 25:17]
      else :
        io.output <= io.input1 @[Selection.scala 27:17]
```

Listing 4.6: HighForm of Firrtl generated from Listing 4.5.

when lowering HighForm to LowForm. The LowForm in Listing 4.7 shows the source location of `when` is fused with other locations. This change occurs because the source location is attached to the node assign statement, not the mux itself. Keeping track of the source location would require reworking the Firrtl specification as well as the Chisel frontend, which would require significant engineering work and compiler internal changes. Section 4.8.2 details the algorithm to handle enable conditions

```
circuit Selection :
  module Selection :
    input clock : Clock
    input reset : UInt<1>
    input io_input0 : UInt<32>
    input io_input1 : UInt<32>
    input io_en : UInt<1>
    input io_sel : UInt<1>
    output io_output : UInt<32>

    node _GEN_0 = mux(io_sel, io_input0, io_input1) @[Selection.scala 24:19 25:17
    ↪  27:17]
    skip
    io_output <= mux(io_en, _GEN_0, UInt<32>("h0")) @[Selection.scala 21:13 23:16]
```

Listing 4.7: LowForm of Firrtl lowered from Listing 4.6

```
module Selection(
  input         clock,
  input         reset,
  input  [31:0] io_input0,
  input  [31:0] io_input1,
  input         io_en,
  input         io_sel,
  output [31:0] io_output
);
  wire [31:0] _GEN_0 = io_sel ? io_input0 : io_input1; // @[Selection.scala 24:19
↪  25:17 27:17]
  assign io_output = io_en ? _GEN_0 : 32'h0; // @[Selection.scala 21:13 23:16]
endmodule
```

Listing 4.8: Generated RTL from Listing 4.7

with multiplexers in a more mature compiler toolchain, where the source location is tracked properly.

In the HighForm pass, we obtain the enable conditions by descending into the IR nodes while maintaining a condition stack. Whenever we enter a conditional scope, we push in the condition and we pop it out when leaving. All the statements are annotated with expressions after ANDing the condition stack.

The second pass collects the annotations from the IR nodes in LowFrom after all the compiler passes. If a previous pass removes an IR node with our annotation, the second pass does not see it. As a result, the debug information is missing. Although inconvenient, this behavior is similar to that of software debugging, where aggressive compiler optimizations remove or inline debug-critical

```
Input: CircuitState
Output: Table
// First pass to annotate FIRRTL nodes;
Annotations ← ∅;
foreach node ∈ CircuitState do
    // Compute enable condition in HighForm;
    if node is statement then
        node.enable ← ComputeEnableCondition(node);
    end
    Annotation ← Annotations ∪ {node};
end
// Second pass after FIRRTL transformations;
IRNodes ← ∅;
foreach node ∈ Annotations do
    if node ∈ CircuitState then
        IRNodes ← IRNodes ∪ node
    end
end
Function ComputeEnableCondition(node):
    condition ← ∅;
    while node ≠ null do
        if node.type = when then
            condition ← condition ∪ node.condition;
        end
    end
    if condition.empty() then
        // This node is not enclosed by any when construct.  It can always be
        triggered.;
        return 1;
    end
    else
        // AND all conditions together.;
        return condition.reduce_and();
    end
Table ← ComputeSymbolTable(IRNodes);
```

Algorithm 4.5: Algorithm to extract symbol table from Chisel/Firrtl.

symbols. The second pass produces debug collateral, which is later used to construct the hgdb symbol table.

A noteworthy aspect of Algorithm 4.5 is the notion of "debug" build. Firrtl did not offer such a build option. However, we achieved similar results using an annotation called `DontTouchAnnotation`. Many of its built-in optimization passes recognize this annotation, since it is a built-in annotation that informs the pass to skip transformations whenever possible. By including this annotation for all relevant IR nodes, we create the equivalent of a debug mode in Firrtl.

```
{
  "type": "module", "name": "Selection",
  "scope": [{
    "type": "block", "filename": "Selection.scala",
    "scope": [{
        "type": "decl", "line": 0, "column": 0,
        "variable": {"name": "io.input0", "value": "io_input0", "rtl": true }
      }, {
        "type": "decl", "line": 0, "column": 0,
        "variable": {"name": "io.input1", "value": "io_input1", "rtl": true}
      }, {
        "type": "decl", "line": 0, "column": 0,
        "variable": {"name": "io.en", "value": "io_en", "rtl": true}
      }, {
        "type": "decl", "line": 0, "column": 0,
        "variable": {"name": "io.sel", "value": "io_sel", "rtl": true}
      }, {
        "type": "decl", "line": 0, "column": 0,
        "variable": {"name": "io.output", "value": "io_output", "rtl": true}
      }, {
        "type": "assign", "line": 21, "column": 13,
        "variable": {"name": "io.output", "value": "io_output", "rtl": true}
      }, {
        "type": "block", "line": 23, "column": 16, "scope": [{
          "type": "block", "line": 24, "column": 19, "scope": [{
            "type": "assign", "line": 25, "column": 17, "condition": "io_en &&
↪  io_sel",
            "variable": {"name": "io.output", "value": "io_output", "rtl": true}
          }]
        }, {
          "type": "block", "scope": [{
            "type": "assign", "line": 27, "column": 17, "condition": "io_en &&
↪  !io_sel",
            "variable": {"name": "io.output", "value": "io_output", "rtl": true}
            }]
        }]
    }]
  }]
}
```

Listing 4.9: Symbol table extracted from Listing 4.6.

Figure 4.9 shows the simplified view of the symbol table extracted from the design in Listing 4.6. Although our algorithm works well in many cases, it has several limitations. A key issue is with extracting source-level names. We relied on the naming convention of Chisel when computing the

enable condition at HighForm, a brittle heuristic that would break if Chisel decides to change it in the future. This can be solved by implementing some form of symbol table or mapping in Chisel and annotating the Firrtl IR. The second limitation is the lack of generator parameters in the symbol table. This information disappears in Chisel after elaboration, but it is crucial for debugging parameterization related bugs. A potential way to solve this challenge is to store the parameters in the symbol table. However, obtaining such information automatically is constrained by the Scala language runtime and can be difficult depending on the parameterization (see Chapter 5). The third problem concerns scope information. Hgdb needs to know about the current scope to construct the frame, e.g., where the variable is declared and what its lifetime is. The source information stored in Firrtl is not enough, since it only contains location information; thus this would require a future revision of the Firrtl standard.

In addition to these systematic limitations in the current systems, there are some issues related to implementation. At the time of writing, Chisel only stores the base name of the files as location information. This level of information introduces potential conflicts if two files share the same base name in the source tree. There are also some bugs in the optimizations, where the compiler crashes if `DontTouchAnnotation` is inserted. We had to turn these passes off during debug build. In terms of source location tracking, some source locations for variables are missing, e.g., `input0` in Listing 4.9. In addition, some are incorrectly tracked if the object is created else where. Because Chisel/Firrtl is still in an early stage, we believe the developers could iron out these implementation issues. However, most of the issues are fundamental and cannot be overcome by heuristics alone. We believe there should be a systematic redesign of the Chisel/Firrtl compiler stack, and relevant techniques are discussed in Section 5.1.

4.8.2 MLIR/Circt

Circt stands for Circuit IR Compilers and Tools, which aims to bring multi-level intermediate representation (MLIR) and LLVM development methodology to hardware design. Circct works by introducing several hardware-related representations into MLIR and leverages the mature MLIR/LLVM compiler infrastructure. It is not a hardware generator per se, but forms the basis for many hardware generator frameworks, whose frontend language compilers emit MLIR, which is then optimized and outputs SystemVerilog.

Circt utilizes the multi-level concepts in MLIR by introducing a concept of dialect. Different hardware generator frameworks, such as Firrtl [16] and Calyx [44], have their own dialects, which can be converted into each other. An IR in any of these dialects can then be lowered into HW dialect, which is the starting point for common Circt optimizations and lowering. At the time of writing, the most mature dialect is Firrtl and we use it as an example in this section.

To use Circt's compiler infrastructure, a high-level dialect needs to be lowered to HW dialect. In our implementation based on Firrtl, we keep track of the symbol changes during the lowering

process. To do so, we attach an attribute containing the original name to the Firrtl IR node when the node is lowered to HW dialect. This is similar to the first pass in Algorithm 4.5. However, we do not need to compute enable conditions at this level, since MLIR has first-class source location tracking ability. We no longer need to assume naming conventions since Circt automatically tracks attributes during transforms. The symbol table collection pass is identical to the second pass in Algorithm 4.5, because we only need to collect symbols that are not optimized away. Algorithm 4.6 shows how we compute enable conditions even though high-level conditional statements are lowered into multiplexers. We leverage the location tracking ability in MLIR and recursively descended into the multiplexer operands to obtain the source location for the conditional statements.

```
Function RecursiveMuxEnable(node, scope):
    if node.type = mux then
        condition ← node.condition;
        true_scope ← scope.add_scope();
        false_scope ← scope.add_scope();
        true_scope.condition ← condition;
        false_scope.condition ← ¬ condition;
        true_scope.location ← node.true.location;
        false_scope.location ← node.false.location;
        RecursiveMuxEnable (node.true, true_scope);
        RecursiveMuxEnable (node.false, false_scope);
    end
    else
        scope.add_node(node);
    end
```

Algorithm 4.6: Recursive algorithm to compute the enable condition for a multiplexer.

Although Circt is a more mature compiler toolchain than the Firrtl compiler, it is still lacking compared to LLVM. For instance, there is no first-class debugging metadata support in Circt. We implemented our debug information through generic attributes; however, metadata would make the implementation much easier if it were handled by the compiler itself. Another fundamental limitation of Circt is that it is merely a compiler backend. The quality of the debug information depends on the dialect that feeds into the system. Because we use Firrtl as an example, all the limitations of Chisel are reflected in Circt as well. However, given our experience with Circt, we believe Circt serves well as a common hardware compiler backend. When Circt adds first-class debugging support, as described in Section 5.2.1, we think most hardware generator frameworks should transition to use Circt as the backend compiler.

4.8.3 Vitis HLS

Vitis HLS is a commercial HLS tool that compiles C/C++ functions to a reconfigurable fabric and RAM/DSP blocks. It allows the same testbench to be used by both software-level execution and RTL simulation, which significantly reduces the time to debug logical bugs. In addition, Vitis HLS supports pipelining and timing/area-based optimization, which reduces the target area usage and increases system performance. Despite improving design productivity, Vitis HLS produces difficult-to-read RTL. Listing 2.7 shows a partial output of the code in Listing 2.5. Although Vitis HLS has a software co-simulation environment, during integration tests, engineers still need to run RTL simulation, which forces them to debug the generated RTL if errors manifest at this level.

To extract the symbol table from a Vitis HLS design, we leverage the fact that the framework frontend is LLVM-based and capable of producing standard LLVM bitcode objects with debug information. One might expect Vitis HLS to be the generator framework that required the least amount of engineering effort. As we show, the opposite is true: we had to work around many obstacles.

To compute the enable condition, we rely on the fact that Vitis HLS statically schedules the computation, i.e., it uses an FSM to specify the realization of a control flow graph, as mentioned in Section 2.1. Because there is a one-to-one correspondence between the control flow graph and source-level constructs, each source-level construct is assigned to a particular compute engine component at a particular time unit/state. As a result, any valid source location has an FSM state correspondence. If a user sets a breakpoint at a particular line, we just need to determine its FSM state and only trigger the breakpoint when the FSM state advances to that target state. If the location-to-state mapping is handled properly, it does not matter whether the design is pipelined or what initialization intervals the user sets. Figure 4.10 shows the algorithm, which was highly similar to prior work on HLS debugging [29]. However, as we started to apply the algorithm in practice, there were some other quirks from Vitis HLS compiler we had to attend to. For instance, due to loop unrolling and pipelining, the LLVM IR nodes are duplicated multiple times, and the new blocks' state information and execution context can not be inferred from the transformed IR, e.g., loop variable may disappear. We thus needed either to patch the loop unrolling pass in Vitis' LLVM frontend or to compare the unrolled block with the original block to infer the new state information. We chose the latter option since it was easier to implement.

Vitis HLS aggressively optimizes memory as it sees fit. For instance, top-level memory access usually turns into AXI streaming access. Internal memory can also be optimized based on the memory access pattern. If the compiler detects linear memory access, it can transform the large memory into a much smaller line buffer, which streams out data sequentially. If the design is pipelined, it can also change the single-port memory into dual-port, doubling the memory bandwidth, as described in Section 2.1. Because hgdb has the capability to monitor array value changes by registering a shadow copy, if we know the transformation semantics, we can annotate the memory

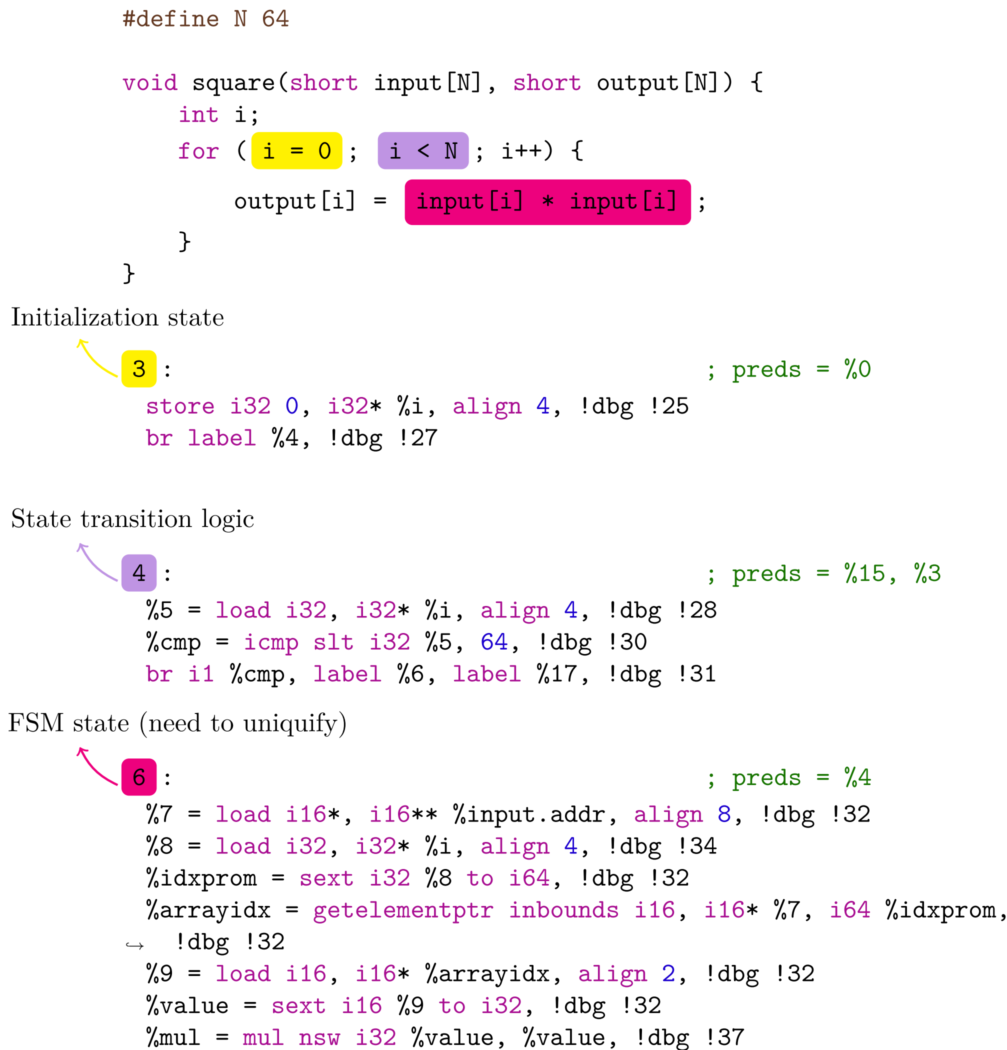


Figure 4.10: Simplified illustration of how state information is associated with source code and IR.

device in the symbol table so the debugger runtime can reconstruct the memory properly. However, Xilinx did not open source its Vitis backend and therefore we could not obtain the source code to extract out the memory transformation collaterals.

Similarly obtaining source-level symbols mapped to RTL is also challenging. Some LLVM IR nodes lose the debug information after HLS-specific transformation. To recover the information, we use the debug build from the C++ simulation and try to match the LLVM symbols with those from the HLS build. This allows us to map source-level symbols to most of the LLVM IR nodes. The next step was to map these IR nodes to RTL signals. Little is known about the Vitis HLS's signal transformation due to its proprietary nature. To solve this issue, we first parse the generated RTL,

then we use heuristics to match the LLVM symbol names to the RTL signal names, an approach that works very well. For instance, if we see an IR node in the C++ version of the LLVM bitcode that refers to the original source-level symbol, we first try to find the correspondence in the the HLS version of bitcode. Then we recursively look through its usage, i.e., any nodes that consume the target node. We then search through the RTL signals to identify any matches.

There are additional engineering-related challenges, such as storing elaboration artifacts in the LLVM IR nodes so that these parameters can be used for debugging. For instance, Vitis HLS unrolls the loops while discarding the loop variables. We annotate the unrolled loop body about the loop iteration so that we can recreate the source-level loop during debugging.

Although we are dealing with a proprietary HLS compiler with closed "trade secrets", our work demonstrates that we can still produce a working symbol table. Listing 4.10 shows the symbol table from the code in Listing 2.5. Note that variable `i` is mapped to a sub-instance register, and each statement is conditioned such that the FSM is not idle and not undergoing reset. Input and output arrays are converted to an AXI buffer, and we are able to detect indexed access to the buffer. Although it may seem obvious to HLS experts that the write buffer shifts in a new element every cycle, we are not able to infer this write pattern from the bitcode and HLS collateral. All we can compute is that the buffer holds 32 elements. As a result, the array reconstruction is neither complete nor correct, since the shift register semantics mismatch with the array semantics.

Although our algorithm to extract a symbol table from Vitis HLS is not perfect, we believe the same techniques can be applied to other open-source HLS compilers to produce a correct symbol table for debugging.

4.9 Why it is so difficult to produce a symbol table?

Although hgdb is very effective at helping designers to debug generated hardware, as shown in Section 4.7, the debugging experience is limited by the quality of the symbol table. In addition, as shown in the previous section, producing such a symbol table from existing generator frameworks is not an easy task. We had to use various heuristics to work around the limitation of the framework. We believe there are two main causes for the issue:

1. intrinsic limitation of the host programming language; and
2. immature compiler infrastructure.

We will discuss these two causes in more detail in the next chapter. Note that the immature compiler infrastructure is two-fold. Some of the limitation comes from the fact that there is no simple method to track debugging information. Table 4.4 shows the current status of various generator framework compiler backends that support symbol and source location tracking. Although certain frameworks support some forms of attributes/annotation tracking, none of these frameworks has

```
{
  "type": "module", "name": "square",
  "scope": [{
    "type": "block", "filename": "square.c",
    "scope": [{
        "type": "decl", "line": 3, "column": 19,
        "condition": "!ap_idle && ap_rst_n",
        "variable": {"name": "input", "value":
↪   "grp_square_Pipeline_VITIS_LOOP_6_1_fu_87.m_axi_gmem_RDATA", "rtl": true,
↪   "size": 32 }
      }, {
        "type": "decl", "line": 3, "column": 35,
        "condition": "!ap_idle && ap_rst_n",
        "variable": {"name": "output", "value":
↪   "grp_square_Pipeline_VITIS_LOOP_6_1_fu_87.m_axi_gmem_WDATA", "rtl": true,
↪   "size": 32}
      }, {
        "type": "decl", "line": 4, "column": 9,
        "condition": "!ap_idle && ap_rst_n",
        "variable": {"name": "i", "value":
↪   "grp_square_Pipeline_VITIS_LOOP_6_1_fu_87.ap_sig_allocacmp_i_1", "rtl": true}
      }, {
        "type": "block", "line": 5, "column": 4,
        "condition": "!ap_idle && ap_rst_n",
        "scope": [{
            "type": "assign", "line": 6, "column": 9, "condition": "!ap_idle &&
↪   ap_rst_n",
            "variable": {
              "name": "output", "value":
↪   "grp_square_Pipeline_VITIS_LOOP_6_1_fu_87.m_axi_gmem_WDATA",
              "rtl": true,
              "index": {
                "name": "i",
                "value":
↪   "grp_square_Pipeline_VITIS_LOOP_6_1_fu_87.ap_sig_allocacmp_i_1",
                "rtl": true
              }
            }
        }]
    }]
  }]
}
```

Listing 4.10: Symbol table extracted from Listing 2.5.

first-class support for debugging information. As shown in Section 4.8, if first-class debugging support is not available in the compiler, we have to implement ad-doc debugging attributes or workarounds to facilitate the process. The lack of debug support also makes it difficult for user-provided passes to track symbol changes, should they modify the logic.

Generator Framework Name	Location tracking		Symbol tracking
	Statement	Expression node	
Chisel	✓	✗	✗
Circt	✓	✓	✗
Vitis HLS	✓	✓	✗
PyMTL	✓	✗	✗
Magma	✓	✗	✓

Table 4.4: Source location and symbol tracking support in different generator frameworks at the time of writing. ✓ indicates incomplete support or workarounds required.

Chapter 5

Design for Debugging

As shown in the previous chapter, obtaining a complete and correct symbol table from existing hardware generators is a process that is technically challenging, heuristic-ridden, and implementation-intensive. This is because the frameworks are not designed to provide the information one needs for source-level debugging. No matter how comprehensive and efficient the debugging infrastructure, the overall source-level debugging experience is limited by the symbol tables that the hardware generator frameworks provide. In this chapter, we discuss how such frameworks should be designed to offer first-class support for source-level debugging. We use a prototype system called *kratos* to demonstrate these concepts.

5.1 Languages are not created equal

Most generator frameworks are embedded in high-level programming languages. This language choice in turn affects how difficult ot os fpr the generator to create the required symbol table information. One important lesson we learned from extracting symbol tables from existing hardware generator frameworks is that some programming language features, such as reflection and runtime inspection, are highly useful for creating a proper symbol table. As shown in the previous chapter, retroactively extracting the symbol table from an existing framework implementation requires significant engineering effort and suffers from various framework and language limitations. If the programming language or compiler lacks the ability to produce a high-quality symbol table, regardless of how expressive the frontend language becomes, the debugging experience will be counter-productive. Hence, when designing a hardware generator framework, developers should consider how the user will debug first, before becoming too heavily invested in their current compilation system and making debugging difficult.

```python
add_code: bool
var1 = input_port(4)

if add_code:
    var2 = output_port(4)
    wire(var1, var2)

# override symbol reference
var1 = input_port(4)
var2 = output_port(4)
```

Listing 5.1: An example hardware construction code using a simple Python eDSL generator framework.

5.1.1 Debugging primitives

Because eDSL-based generator frameworks use the host language's environment to construct hardware, the hardware generation is a result of partial program execution. In other words, the runtime environment executes the source code and produces hardware components along the way, as shown in Listing 5.1. Line 1 defines a Python variable called `add_code`. The variable does not have any RTL correspondence, but it controls the hardware generation, i.e., it is a generator parameter. During program execution, Line 4 tests `add_code` and only generates `var2` if `add_code` is true. As a result, the value of `add_code` is critical for debugging since it offers users information about the hardware generation process and whether it causes the hardware bug. Frameworks typically do not store generator parameters in the symbol because they do not correspond to any RTL construct. As mentioned in Section 3.2.1, we need to capture these generator parameters to help designer diagnose generation-related bugs.

Another consequence of partial evaluation during program execution is changes to the symbol mapping. In most programming languages, the variables only store the reference to an object on the stack or heap. Subsequent assignments can change the reference using the same variable. This implies that the variable reference is also execution-dependent. As shown in Line 9 and 10 in Listing 5.1, `var1` and `var2` can hold references to different wires depending on the execution context.

We thus need a means to introspect the execution context and record the generator parameter values and symbol mapping. Since the stack contains all variable references (except the global variables), regardless of where the variable is actually allocated, we can inspect the stack frame to obtain the required debugging information. We need to know the collection of variables declared so far, the source locations where the variables are declared, the object references stored by the variables, and the lifetimes of the variables. These requirements imply that the host programming languages should provide a rich set of tools for the generator framework to introspect the stack frame and obtain debugging information during their execution.

Because different programming languages have different features to introspect the stack frame and the obtainable information varies, we defined a set of debugging primitives that abstract away the programming language features. In this section, we first illustrate how one can obtain all necessary information using these primitives; thereafter, we discuss how each programming language should implement these primitives and provide mitigation approaches if the language lacks this ability.

Debug primitive name and syntax	Synopsis
`get_stack_frames()`	Get the stack frames at the caller site.
`get_local_variables(frame)`	Get local variables from a stack frame
`get_source_location(obj)`	Get source-level location from an object
`get_variable_scope(frame)`	Get the scope object from a stack frame.

Table 5.1: A list of debug primitives that are useful for implementing debugging features for hardware generators.

In the list of primitives shown in Table 5.1, some primitives are necessary whereas others are optional. However, the optional primitives either significantly reduce the amount of engineering work, or they improve the quality of the symbol table.

The most important primitive is `get_stack_frames`, which returns the call stack for the current execution. This primitive forms the basis of all subsequent primitives. Line 2 in Algorithm 5.1 shows how we use this primitive to obtain the caller frame. In the example, a register is created by calling an eDSL primitive function called `Register`. Note that we use the second stack entry to obtain the caller frame because the top frame belongs to the function `Register`.

The second primitive is `get_local_variables`, which can be used to determine currently declared variables and their values on the stack frame. The variable can either be static, i.e. generator parameters, or RTL signals. Line 6 in Algorithm 5.1 shows how to use this primitive to compute the local variables.

The third primitive is `get_source_location`, which takes an object and returns source-level information. This primitive can be used to obtain information about where the variable is declared. Line 4 in Algorithm 5.1 shows the algorithm that obtained signal declaration information, such as source location, and then attached the source location to the register.

The last primitive, called `get_variable_scope`, is used to obtain the scope information of a variable. Note that the third primitive does not indicate the lifetime of a variable. Although we can ignore the scopes and store all the local variables for every IR node, doing so is inefficient for storage and computation. This primitive allows us to obtain the scope information and store the local variable efficiently. Line 5 in Algorithm 5.1 demonstrates how to use `get_variable_scope` to obtain scope information and store local variables into the scope. Note that we only stored entries that are new to the scope because variables could be declared outside our framework in the host language. As a result, we capture these variables to reconstruct the state. Instead of storing the

```
// For simplicity, the Register constructor only takes one parameter;
Function Register(width):
    // Create the actual register;
1   reg ← CreateReg(width);
2   frames ← get_stack_frames();
    // We need caller site's stack frame, not current frame (index 0);
3   frame ← frames[1];
4   reg.loc ← get_source_location(frame);
5   scope ← get_variable_scope(frame);
6   foreach var ← get_local_variables(frame) do
7       AddContextVariable (scope, var);
8   end
9   reg.scope ← scope;
10  return reg;
```

Algorithm 5.1: Algorithm to obtain the debugging information for the register primitive by calling the primitive constructor `Register()`.

```
Function AddContextVariable(scope, variable):
    name ← variable.name;
    value ← variable.value;
    loc ← get_source_location(variable);
    if ¬ scope.has(name) ∨ scope.get(name).loc ≠ loc ∨ scope.get(name).value ≠ value then
        scope.set(name, variable);
    end
```

Algorithm 5.2: Algorithm to add static variables to debugging information.

entire local variables into the IR node, we only attach the scope reference to the IR node. The scope can be serialized into JSON format directly, as described in Section 4.5.

These primitives, however, are useful only if the generator frameworks are an eDSL. HLS needs a different approach because these frameworks directly digest the source code and generate the hardware through direct compiler elaboration and compilation. As a result, the compiler knows the exact source location and liveness of the variables. We refer to these types of generator frameworks as *compiled* and the eDSL-based as *embedded*. We disambiguate these two categories of frameworks because most of the chapter discusses the language implementation for embedded frameworks.

5.1.2 Language-specific implementations for embedded frameworks

The previous section discussed the primitives to introspect the stack frame to obtain debug information. In this section, we examine how to implement these primitives in various languages. Table 5.2 shows the difficulty level for various programming languages to implement these primitives. The difficulty levels are subjective, but generally reflect the amount of engineering work required for implementation. Note that the language paradigm plays little role in determining the feasibility of

primitive implementation: what matters more is how the language frontend is designed.

In general, it is much easier to implement these primitives with interpreted languages than compiled languages. This is because the debug primitives rely heavily on reflection, which gives the programming language access to the runtime objects. Because interpreted languages already have an interpreter that keeps track of all live objects, obtaining this information is straightforward. Compiled languages, in contrast, require additional effort to obtain local variables, since such information is typically not accessible by the source code once compiled into a binary.

This point does not mean it is impossible to implement the primitives while developing an embedded DSL. There are several ways to mitigate the issue. Java Virtual Machine (JVM), for instance, has an API that allows external programs to access its debug interface, called Java Debug Interface (JDI). The generator framework can essentially implement a debugger that talks JDI and controls the program execution. Through JDI, the framework can implement the primitives easily. However, this approach significantly reduces the runtime performance, since virtually all generation-related statements are treated as breakpoints for the JVM. It is also possible to parse the code at runtime and obtain the debugging information through partial program execution. However, this approach requires significant engineering effort if the language does not provide the means to parse itself and partially evaluate the syntax tree. As a result, if the framework developers wish to use a compiled language, it is more effective *not* to make the framework an embedded DSL. Instead, the frontend should be a language compiler that can parse and understand the source code structure, such as an HLS tool or a compiler like BlueSpec [45].

To demonstrate the easiness of primitive implementation in a interpreted language, we show how one can implement these primitives in Python in Listing 5.2 (the function `to_scope` is omitted for simplicity). Note that in Python, variables are scoped to the innermost function, class, or module; control blocks such as `if` and `while` do not start a new scope. As a result, `get_variable_scope` is essentially a mapping from the calling function stack to the scope object. A limitation of using Algorithm 5.1 is that we cannot obtain source locations created outside the framework, such as those created from Python native types. This is because we introspect the caller site stack frame to obtain the debugging information. Figure 5.1 shows an example of generator code that uses a `for` loop to construct wires. Variable `i` is created implicitly by Python's syntax at Line 2, and we need to store that result in our symbol table. The workaround is to assign the same source location where these variables are detected in the framework, e.g. Line 3 in the example. This is because hgdb strictly obeys the ordering of each statement when reconstructing the scope, as shown in Algorithm 3.12. When the debugger runtime walks up the scopes, these static variables will appear in the stack frame as if they were declared earlier. In addition, the strict ordering of symbol table entries means that while we walk upward through symbol table entries, variables declared later (visited first) cannot be overridden by those declared earlier (visited later). For instance, when we reconstruct the stack frame at breakpoint ❷, since we visit variable `i = 1` first, the subsequent visit to `i = 0` does not

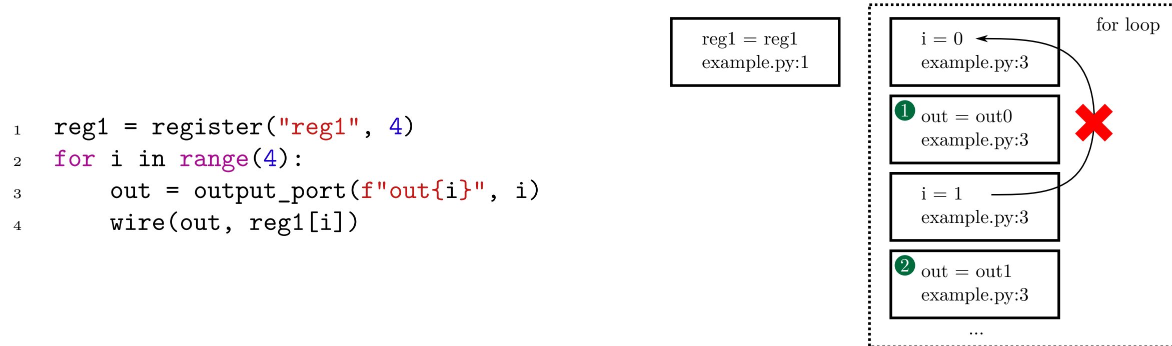


Figure 5.1: Workarounds for handling variables declared outside the framework. Available breakpoints are shown in ❶ and ❷.

```
import inspect

def get_stack_frames():
    return inspect.getouterframes(inspect.currentframe())


def get_local_variables(frame: inspect.FrameInfo):
    return frame.frame.f_locals


def get_source_location(obj):
    return inspect.getsourcefile(obj), inspect.getsourcelines(obj)[-1]


def get_variable_scope(frame: inspect.FrameInfo):
    # to_scope is a one-to-one mapping from frame to scope obj
    return to_scope(frame.frame)
```

Listing 5.2: Example code in Python to implement debug primitives.

change the variable's value.

5.2 Maintaining a symbol table inside the compiler

After obtaining the symbol table from the frontend, the compiler must keep track of symbol changes throughout the transformations and optimizations. Similar to software compilation, certain passes preserve the symbol table and some passes are more challenging than others with regard to tracking symbol changes properly.

Language	GSF	GLV	GSL	GVS	Overall difficulty
C/C++ (embedded)	Debugger	Debugger	✓	Debugger	Very difficult
C/C++ (compiled)	✓	✓	✓	✓	Easy
Scala (embedded)	Debugger	Debugger	✓	Debugger	Very difficult
Scala (compiled)	✓	✓	✓	✓	Easy
Python	✓	✓	✓	✓	Easy
JavaScript	✓	✓	✓	✓	Easy
Ruby	✓	✓	✓	✓ (incomplete)	Easy

Table 5.2: Language support for debug primitives. GSF stands for `get_stack_frames`, GLV stands for `get_local_variables`, GSL stands for `get_source_location`, and GVS stands for `get_variable_scope`. ✓ indicates incomplete support.

5.2.1 First class debugging support in the compiler

Although the option always exists to turn off the transformation passes, debug the unoptimized build, and use formal verification tools to verify the equivalence of unoptimized and optimized designs, there are nonetheless benefits to debugging optimized designs. First, optimization speeds up the simulation significantly. For a large-scale design, fast simulation shortens the engineering turnaround time (i.e., time spent on making a change, simulating, and debugging), which improves the verification productivity. Recall that once we turned off Firrtl compiler's optimization, including dead code elimination and instance inlining, the RocketChip simulation runtime increased by 30% with a commercial simulator. Second, some optimization passes alter the timing semantics of the design. For instance, auto pipelining inserts shift registers in the data path that changes the data wave timing. In many cases, such changes do not affect the functionality but may expose performance bugs within a larger system. If a downstream IP makes certain assumptions on the exact timing of data arrival, the bug will manifest only when the pipelining passes are turned on. Thirdly, some transformation passes are written by the designers to optimize the design. These passes are domain-specific and unfortunately may contain bugs. Designers have to turn these passes on to debug the design as well as the custom passes.

As a result, the generator frameworks should provide facilities to maintain the symbol table. In particular, they need to supply tools or APIs for transformation passes to update debugging information. As shown in Section 4.9, most generator framework compilers lack full support for tracking debugging information. Most modern software compilers have mature infrastructure for transformations to track debugging information, i.e., they automatically copy over an IR node's debugging information to a new IR node. As a result, the details on how to implement such an infrastructure are not described here. Hardware generator frameworks should borrow the ideas and reuse established implementation techniques from software compilers; or simply use a mature compiler backend such as Circt if it adds first class symbol tracking support in the future. We instead focus on how to *update* the debugging information during transformations.

5.2.2 Debug invariant transformations

Although hardware compiler passes share many similarities with those of software compilers, many minor differences still exist, which make it easier to manage debugging information in hardware. We introduce the concept of a "debug invariant" transformation which preserves debugging information for hardware debugging. To be classified as debug invariant, the transformation should transform the symbol table in such a way that the debugger can reconstruct the execution state as if the transformation never happened. We claim that all semantically equivalent transformations are "debug invariant." An incomplete list of these transformation is shown below:

- Loop unrolling
- SSA transformation
- Constant propagation
- Instance inlining
- Common subexpression elimination

Most of these passes, such as loop unrolling, SSA transformation, and constant propagation, share the same logic as that of software compilers. When performing loop unrolling and constant propagation, the compiler needs to store elaborated values in the symbol table so that the eliminated values can be reconstructed during debugging. SSA transformation, as discussed in Section 3.1, converts the nested control logic into continuous assignment. The compiler needs to keep track of SSA phi nodes inserted in the control flow graph, which are used to compute enable conditions. It must record the symbol mapping as temporary nodes inserted to satisfy the SSA property.

As for common subexpression elimination, since all the new expressions created have an RTL symbol handle, i.e., a named wire or register, we can use the new RTL symbol for variable mapping. Because this handle exists throughout the simulation and is not overwritten by other instructions, it is easier to handle the value reference than in software. Instance inlining is also much easier to manage in hardware. Function inlining in software is known to destroy debugging information and the debugger has to create an artificial stack to recover the information [46, 47]. In contrast, hardware instance inlining preserves the logic structures and thus also the debug information. This is because the hardware simulator keeps all the state information, whereas software states may destroy some states because certain instructions are not reversible. We need to rename the internal variable mapping to the new one and handle the instance port remapping, as shown in Figure 5.2. In the example, `inst.b` is inlined to `inst_b` and consequently the source level symbol `v_b` should map to the new inlined symbol. The compiler should also copy over any debugging information from internal statements (e.g., `assign b = a;`) while renaming symbols.

Other passes, such as dead code elimination, may or may not be debug invariant, depending on the completeness of a symbol table. The easiest approach would be to mark any removed IR node

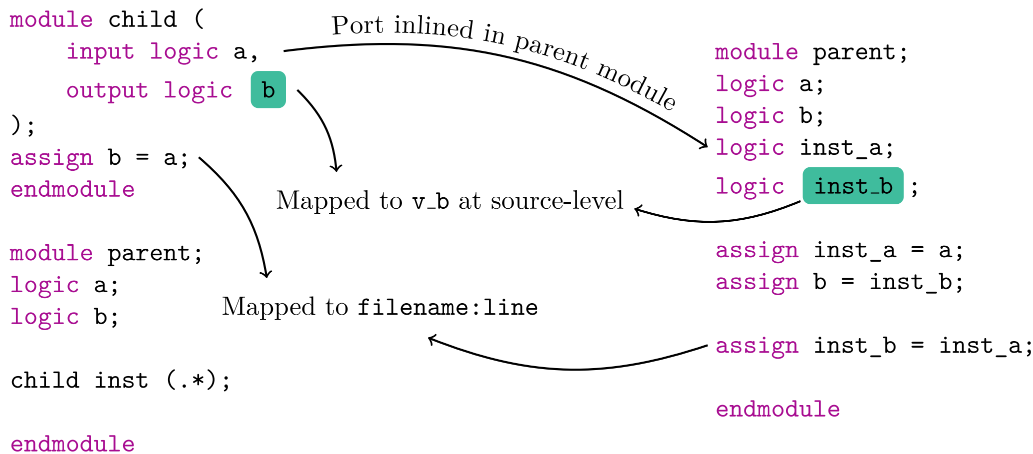


Figure 5.2: Illustration of how to preserve symbol mapping while inlining instances.

in the symbol table as "optimized out." During debugging, users will not see the actual value of the deleted node. This is the same behavior as in compiled software programming languages such as C/C++. However, the ability to inspect optimized-out logic is critical for debugging cases where the logic is unintentionally removed. This problem is typically caused by incorrect parameterization where the parameter that connects the target logic to the system is wrong. Allowing users to trace the hardware construction at the source level is crucial because it enables users to discover why the logic is disconnected. As a result, we think it is beneficial to preserve debugging information while removing dead logic for optimization. This is a potentially difficult task since the simulator does not contain the values of the deleted logic.

We realized that if the deleted nodes were combinational logic, i.e. pure functions that had no side effects, we could actually dynamically evaluate the node values inside the debugger during simulation. Figure 5.3 shows an example of how hgdb dynamically evaluates expressions to recover deleted variable information using the symbol table. Since logic `c` does not produce any output, we can safely remove it from the design. However, since it is referenced in another line's scope variable `v_c`, we must remap `v_c` to the expression that is assigned to `c`, namely. `!a`. During simulation, because the simulator knows the value of `a`, we can successfully compute the value of `v_c`.

When the deleted nodes contain sequential logic, i.e., are a function of time, however, the task is significantly more difficult. This is because the debugger needs to track the state changes and fill in the symbol table details in case there are inter-node dependencies in the logic. This is not the case for combinational logic since it is a pure function of the current simulator state. Figure 5.4 shows an example of how to use state updates from the existing nodes to compute the deleted sequential logic. When the breakpoint is inserted we create a shadow copy that tracks the previous value of `a`, similar to the shadow copy described in Figure 3.5. However, it is evident that this approach is essentially implementing a small logic simulator inside the debugger. We opted not to implement this feature

```
module example (
    input logic a,
    output logic b
)
              Will be removed after DCE
logic c;
assign c = !a ;

assign b = a;
              v_c (source) is mapped to RTL c
endmodule
```

```
module example (
    input logic a,
    output logic b
)
 v_c (source) is mapped to expression !a
assign b = a;

endmodule
```

Figure 5.3: Illustration of how to preserve symbol mapping when removing pure combinational dead code.

```
module example (
    input logic clk,
    input logic a,
    output logic b
);
  Will be removed after DCE
logic c;

always_ff @(posedge clk) begin
    c <= !a ;
    b <= a;
end      v_c (source) is mapped to RTL c

endmodule
```

```
module example (
    input logic clk,
    input logic a,
    output logic b
);
always_ff @(posedge clk) begin
    b <= a;
end
    v_c (source) is mapped to SC( !a )
endmodule
```

Figure 5.4: Illustration of how to preserve symbol mapping when removing sequential dead code. Function SC returns the shadow copy of an expression.

because it requires a significant engineering effort to make the evaluation performant. This requires the hardware compiler to keep sequential nodes so that the node values can be recovered properly. In other words, the hardware compiler should have a debug flag that keeps the dead code state, so that the debugger can reconstruct the complete execution state.

5.3 Kratos: an efficient working prototype

Previous sections have provided details on how to build a debuggable generator framework and issues to which the developers should pay attention. Nonetheless, it can remain challenging to build such a framework. [1] To demonstrate the feasibility of building such a generator framework without compromising the frontend language features, we built an efficient working prototype called

[1] The devil is in the details

kratos. Similar to software programming language debugging support, the debuggability of generator frameworks is orthogonal to language designs, if carefully planned. In this section, we describe the design choices, performance optimization techniques, and insights learned. We hope by offering detailed information about the prototype, we can help other framework developers redesign the future versions of hardware generators.

The first design choice was whether kratos should be an eDSL, and if so, which language to embed. We opted for a Python-based eDSL, for the following reasons:

1. eDSL makes the front language design much easier since we do not need to implement a new lexer or parser.

2. Python has excellent support for runtime introspection and code modification, as discussed in Section 5.1.2.

3. Python has a relatively flat learning curve compared to other languages, such as Scala and C++.

4. Python supports various programming paradigms, such as OOP, imperative programming, and functional programming. As a result, Python is a good candidate for proving the key points made in previous sections.

After choosing a host programming language, the next step was to implement the frontend features. We aimed to match and implement the frontend language features from other popular generator frameworks. For instance, we borrowed the structure of inheritance from Chisel [5]; each generator instance must inherit from the `Generator` class, which offers many helper functions to create wires and registers. We also supported functional programming using Python's functional language features. Listing 5.3 shows the kratos code for logic identical to that in Listing 2.4. At line 4, we create a generator that inherits from the base `Generator` class. At Line 7 and 8, we create a list of IO ports using functions `input()` and `output()`. Then we use Python's builtin `map()` and `reduce()` with the `lambda` function to compute the final result. In addition to copying object-oriented features from Chisel, we added complex data types, such as `Bundle` and `interface` to kratos.

Inspired by PyMTL, where complex logic can be specified by plain Python syntax, we introduced several primitives called `always_comb` and `always_ff` that take Python syntax and transform it into internal IR, as shown in Listing 5.4. In the example, we specify a simple ALU whose instruction selection logic is done via a series of Python `if` statements. Depending on the value of `instr`, the logic uses different arithmetic operations on the input data. The Python decorator `always_comb` encapsulates the syntax tree underneath as a function object. Inside the function call `add_always` in line 20, we introspect and transform each Python abstract syntax tree (AST) node into proper IR construction calls. The challenge with using Python AST nodes is that the methods described

```
1  from kratos import Generator
2  from functools import reduce
3
4  class InputEven(Generator):
5      def __init__(self):
6          Generator.__init__(self, "InputEven")
7          inputs = self.input("inputs", 32, size=4)
8          output = self.output("output", 1)
9
10         self.wire(output, reduce(lambda a, b: a & b), map(lambda i: i % 2 == 0,
           ↪  inputs))
```

Listing 5.3: Kratos implementation for the algorithm from Listing 2.4.

```
9  @always_comb
10 def alu_logic():
11     if instr == 0:
12         out = data0 + data1
13     elif instr == 1:
14         out = data0 - data1
15     elif instr == 2:
16         out = data0 * data1
17     elif instr == 3:
18         out = data0 / data1
```

```
always_block.add_stmt(
  If(instr == 0, Assign(out, data0 + data1,
  ↪  lineno=12), lineno=11).Else(
    If(instr == 1, Assign(out, data0 - data1,
    ↪  lineno=14), lineno=13).Else(
      If(instr == 2, Assign(out, data0 * data1,
      ↪  lineno=16), lineno=15).Else(
        If(instr == 3, Assign(out, data0 / data1,
        ↪  lineno=18), lineno=17)
        )
      )
    )
)
```

Figure 5.5: AST transformation in kratos that keeps track of the source location.

in Section 5.1.2 does not apply. This is because the transformed AST changes the original source code structure and thus yields incorrect source locations. We realize that Python encodes the source location in each AST node through the `ast` module. We therefore obtain the source location from the AST nodes, and then store the location during AST transformation, as shown in Figure 5.5. Note that every construct in the original Python code is transformed into a specific function call that constructs the corresponding IR node, e.g., `if` to `If`. We use keyword argument `lineno` to set the line number for newly constructed IR nodes. More details about the AST transformation and various techniques are shown in Appendix B.

Another challenge we faced while implementing debugging features in kratos was the performance overhead to obtain source locations. We initially implemented the algorithms described in Section 5.1.2. Obtaining previous frame objects from the current stack frame in Python code turned out to be very slow. This is because Python needs to initialize and populate all the fields in the frame object. Since we only need to obtain the source location most of the time, calling these functions

```python
class ALU(Generator):
    def __init__(self, width: int = 16):
        Generator.__init__(self, "ALU")
        data0 = self.input("data0", width)
        data1 = self.input("data1", width)
        out = self.output("out", width)
        instr = self.input("instr", 2)

        @always_comb
        def alu_logic():
            if instr == 0:
                out = data0 + data1
            elif instr == 1:
                out = data0 - data1
            elif instr == 2:
                out = data0 * data1
            elif instr == 3:
                out = data0 / data1

        self.add_always(alu_logic)
```

Listing 5.4: Kratos example code to implement an ALU using `always_comb` decorator.

in Python incurs undesirable overhead. To speed up the introspection, we implemented the same algorithm in a Python C extension. This allows us to bypass the field initialization and speed up the introspection process. Based on our experiments, switching to using the C extension sped up the RTL generation process by a factor of 50. Turning on debug flags only introduced about 25% overall runtime overhead.

Similar to Firrtl and Circt, kratos also adopts multi-level IRs where the high-level constructs like memory and FSM are transformed during lowering passes. The lower IR is similar to the HW dialect from Circt, where little transformation is required to generate SystemVerilog. During lowering passes, high-level IR primitives including FSM and FIFOs are transformed into lower IR primitives or technology-dependent modules. Listing B.4 in Appendix B shows some examples of the FSM code. Designs in Section 4.7.1 use the FSM primitive to construct the memory controller.

We implemented all commonly used compiler optimizations/transformations such as SSA, loop unrolling, constant propagation, strength reduction, instance inlining, and dead code elimination. In addition, we implemented optimizations that resulted in better simulation performance and synthesis quality. For instance, to reduce the waveform size, kratos optimizes away any pass-through wires. This is beneficial because most simulators dump the signal changes for the pass-through wires even though they can be aliased to the driver signals. Removing these pass-through wires reduces the disk IO and also speeds up the waveform loading time during debugging. To improve synthesis quality, we utilize many advanced SystemVerilog constructs to give hints to the synthesis tools. For

instance, kratos transforms nested `if` statements into a `case` statement, if all of their conditions are comparison of the same variable against some constants. Kratos automatically apply `unique` modifier if all the conditions are mutually exclusive. This allows the synthesis tools to ignore priority of the conditional statements and produce a better netlist. Note that conventionally this needs to be annotated by hand either in the RTL design phase or during physical design, e.g., using directives such as `full_case` and `parallel_case`. These directives are considered harmful but are still commonly used in practice [48]. Listing 5.5 shows the generated RTL from Listing 5.4, and Listing 5.6 shows the optimized result. Note that we only turned on SSA and other similar optimizations in a debug mode. This is because these transformations hinder the synthesis tool from optimizing the the logic based on SystemVerilog patterns. Kratos has a seamless flow to set up JasperGold [49] to verify the logical equivalence between normal and debug builds.

```
1  module ALU (
2    input logic [15:0] data0,
3    input logic [15:0] data1,
4    input logic [1:0] instr,
5    output logic [15:0] out
6  );
7
8  always_comb begin
9    if (instr == 2'h0)
10     out = data0 + data1;
11   else if (instr == 2'h1)
12     out = data0 - data1;
13   else if (instr == 2'h2)
14     out = data0 * data1;
15   else if (instr == 2'h3)
16     out = data0 / data1;
17 end
18 endmodule   // ALU
```

Listing 5.5: Generated RTL code from Listing 5.4 without `if` priority optimization.

Similar to Chisel/Firrtl, where users can implement their own passes to either transform or analyze the target design, kratos also allows users to write custom passes. In the sparse memory controller described in Section 4.7.1, the IP was configured externally; that is, the configuration register was instantiated outside the IP block. Configuration-related ports were annotated in the source code and were lifted to the top-level IP interface through a custom pass written by the designers. Through these custom passes, the designers can add configuration ports anywhere and anytime in the design. Traditionally, this port-lifting process has to be done manually and is tedious and error-prune since designers have to edit all the relevant files based on design hierarchy. More details about this pass are shown in Appendix B.

```
1  module ALU (
2    input logic [15:0] data0,
3    input logic [15:0] data1,
4    input logic [1:0] instr,
5    output logic [15:0] out
6  );
7
8  always_comb begin
9    unique case (instr)
10     2'h0: out = data0 + data1;
11     2'h1: out = data0 - data1;
12     2'h2: out = data0 * data1;
13     2'h3: out = data0 / data1;
14   endcase
15 end
16 endmodule   // ALU
```

Listing 5.6: Generated RTL code from Listing 5.4 with `if` priority optimization.

Unlike Circt, where we had to implement debugging information as generic attributes, kratos treats debugging information as a first-class entity. Every IR node in kratos, such as variables and expressions, has dedicated debugging attributes including source locations and scope information. The benefit is to reduce debug information access overhead. This is because we would have to query all the attributes associated with the IR node and filter out the debug information, if this feature were not implemented. Kratos also automatically tracks the debugging information during transformations. For instance, in Listing 5.6, even through the `if` is transformed into a case statement, kratos still tracks the original source location.

Although kratos is a prototype by nature, it has been used to design many different hardware IPs, for which have been taped out with several chips. As shown in Section 4.7, due to its first class support for debugging, designers can quickly identify the bugs, thus reducing the verification time. We also received positive feedback from users regarding design productivity. We have open-sourced kratos on Github and we hope that it can serve as a reference implementation for how to build a hardware generator that enables source-level debugging.

Chapter 6

Conclusions and Future Work

Hardware generator frameworks are known to improve design productivity but often produce illegible RTL code. Due to the nature of the established verification environments, the designers often have to debug with the generated RTL, which inevitably hurts verification productivity. This dissertation has presented a source-level debugging infrastructure called hgdb for hardware generator frameworks to address this issue. It bridges the semantic gap between source programming languages and generated RTL and enables designers to use modern debugging features at the source-level.

We had three design goals for our system: faithful source-level state reconstruction for debugging, minimal overhead introduced to RTL simulation, and maximum compatibility among existing RTL simulators and generator frameworks. To achieve these goals, we divided the system into multiple components, namely, the debugger runtime, the debugger frontend, and the symbol table. The debugger runtime interacts with the simulator and is responsible for emulating debugging features such as breakpoints as efficiently as possible; the frontend renders the debugging information at source-level; and the symbol table stores the source-level information. To decouple the dependencies among these components and target various entities such as RTL simulators and generator frameworks, we also defined three sets of interface primitives, namely, simulator, debugger frontend, and symbol table interface. These interface primitives, similar to the ISA for a CPU, abstract away the implementation details of different components and allow optimization for individual ones.

There were many challenges in building such a system that aims to achieve these ambitious goals. One of the biggest challenges was to select the types of interface primitives to use and determine how to implement these primitives. When working on the debugger runtime, we started off assuming unconstrained support from the simulator and chose simulator primitives that achieved the maximum simulation performance while preserving proper source-level debugging semantics. Then, in order to maximize compatibility among popular RTL simulators, we had to scale back and remove any features that were not widely supported. We eventually settled on a small subset of VPI routines, which are part of the SystemVerilog standard. As is the case with any software system that follows

a complex and obscure standard, different RTL simulators have different VPI implementations, and even their semantics vary slightly. We had to use a shim layer to abstract away the minor differences and implement heuristics when some VPI routines were not available.

The second challenge was how to emulate debugging features such as breakpoints in our system to make it as performant as possible. We learned two valuable lessons while trying out various techniques: the first was that we needed to interrupt the simulator as little as possible and shift the overhead to user interaction time. We proposed an algorithm to pause the simulator once every clock cycle and use the synchronized state to emulate breakpoints. This emulation technique also required additional compiler passes such as loop unrolling and SSA to preserve any partial computation results. Fortunately these passes were already widely used in many hardware generator frameworks. The second lesson is that we could offload the compute-intensive tasks to different system components to speed up timing-critical path. A few milliseconds of latency when interacting with machines is usually not an issue. As a result, we compute expensive tasks, such as translating symbols into simulator internal handles, whenever the user is interacting with the debugger. This allows us to get reasonable debugging performance without adding significant simulation overhead.

The greatest challenge was working with hardware generator compilers to produce a correct symbol table. Although we somewhat expected that this would be challenging, we were still surprised by the sheer amount of engineering effort that is needed to extract the symbol table. Several factors lead to this huge effort. First, debugging is an afterthought for these frameworks, whose primary goal is to increase frontend design productivity. As a result, many design choices made early on hinder the development of source-level debugging. Second, the compiler backend is not designed to track debugging information. Many transformation and optimization passes destroy source-level information because the IR is not structured to contain such information. Third, some generator frameworks are proprietary, and we had to reverse engineer the black box to recover the symbol mappings. This experience convinced us that the design for debug is an essential principle when developing a new hardware generator framework. Since verification usually takes a significant portion of the hardware development cycle, we believe verification productivity should also be the key metric when evaluating a generator framework. Our experience also convinced us that many debugging features in eDSLs are limited by their host programming languages. As a result, we systematically analyzed various software programming language features that affect symbol table creation. It turns out that some languages are better than others when used as a host language for eDSLs. Python, for instance, has the ability to introspect live stack frames during runtime, whereas Scala and C++ cannot easily do this. In addition to providing analysis for the choice of host language, we also identified a class of compiler transformation passes that preserve debugging information, provided the compiler tracks source-level information at the IR level.

To demonstrate the effectiveness of source-level debugging, we prototyped a Python-based hardware generator framework called kratos that satisfies all debugging requirements. It supports various

programming styles such as object-oriented programming and functional programming. It also implements common compiler passes such as dead code elimination and loop unrolling. We used kratos to develop some IP blocks in a reconfigurable hardware design called CGRA. Using the debugging infrastructure, we successfully shortened the time for localizing the bugs.

We learned two lessons from our prototyping experience. First, it is possible to build a generator debugger that can be debugged at source-level, regardless of the programming paradigm. Although it is true that additional engineering work occurs when developing the compiler backend, the effectiveness of source-level debugging outweighs the effort. Developers can reuse more mature compiler backends such as Circt to avoid reinventing the wheel. Second, our system essentially allows designers to use state-of-the-art debugging features at the source-level. Many commercial RTL debugging tools do offer similar capabilities that are widely used, but they cannot be used since they only operate at the RTL level. With hgdb, the designers can have similar verification productivity.

Looking forward, we have many potential options for extending our work. In our simulator interface implementation, we were essentially using VPI routines to hijack the simulator execution flow and obtain signal values. It might be more efficient if we can use the simulator's internal API to avoid VPI's overhead. This, however, requires collaboration with the simulator vendor. Since these vendors are less likely to open-source their internal APIs, we need to refactor our shim layer to handle proprietary simulator interfaces. Another future direction is to work with Circt developers to make symbol mapping information first class. This should greatly improve the symbol table quality, and given the increasing popularity of Circt, enable more generator frameworks to have source-level debugging support.

While this thesis has primarily covered the verification aspect of hardware design, another major difficulty of using hardware generator frameworks is physical design (PD). Unfortunately, PD tools share many conundrums with verification software. For instance, both categories only take SystemVerilog as input, and both are very well-established in the industry. Although we can apply the same ideas presented in this thesis to work with physical designs (in fact we even have a prototype reusing the same symbol table), it is unclear what the best approach is for handling the symbol mapping, and many open research questions remain. For instance, should the user set PD constraints in the source code and have the compiler generate proper Tcl scripts, or should the user write Tcl scripts instead and query the symbol table inside the script to obtain mapped RTL handles? If some PD-related issues are observed, such as routing congestion, and RTL-level changes are required, how should the designer trace the places of interest back to the source code and issue a change? Furthermore, if an engineering change order (ECO) is issued, how should the fix be propagated back to the source code so that it can be fixed in the future generation? We believe these research questions can be solved by making debugging the first priority when designing hardware generator frameworks. In other words, we think the root of most verification and physical design-related issues caused by using a hardware generator framework is the lack of interest in or consideration for debugging when

the frameworks are built. With the help of a mature compiler backend and a carefully designed frontend, the generator frameworks can overcome all the issues noted above and truly increase the overall productivity of hardware design.

Bibliography

[1] D. A. S. Committee, "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[2] D. Stow, I. Akgun, R. Barnes, P. Gu, and Y. Xie, "Cost analysis and cost-driven ip reuse methodology for soc design based on 2.5d/3d integration," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE Press, 2016, p. 1–6. [Online]. Available: https://doi.org/10.1145/2966986.2980095

[3] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. P. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian, "Rethinking digital design: Why design must change," *IEEE Micro*, vol. 30, no. 6, pp. 9–24, 2010.

[4] L. Truong and P. Hanrahan, "Magma circuits," 2022. [Online]. Available: https://github.com/phanrahan/magma

[5] J. B. *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *DAC*. USA: ACM/IEEE, 2012, pp. 1212–1221.

[6] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A unified framework for vertically integrated computer architecture research," in *Micro*, 2014.

[7] S. Chen, Y. Fisseha, J.-B. Jeannin, and T. Austin, "Twine: A chisel extension for component-level heterogeneous design," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 466–471.

[8] Google, "Experiences building edge tpu with chisel," 2018. [Online]. Available: https://www.youtube.com/watch?v=x85342Cny8c

[9] W. K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches.* USA: Prentice Hall PTR, 2005.

[10] SpinalHDL, "Spinalhdl," , 2022.

[11] W. Ecker and J. Schreiner, "Introducing model-of-things (mot) and model-of-design (mod) for simpler and more efficient hardware generators," in *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2016, pp. 1–6.

[12] A. Ronacher, "Jinja — jinja documentation," 2022. [Online]. Available: https://jinja.palletsprojects.com

[13] O. S. *et al.*, "Rethinking digital design: Why design must change," *IEEE micro*, vol. 30, no. 6, pp. 9–24, 2010.

[14] Chips Alliance, "Chisel/firrtl: Memories," 2022. [Online]. Available: https://www.chisel-lang.org/chisel3/docs/explanations/memories.html/

[15] T. L. Veldhuizen, "C++ templates are turing complete," 2003. [Online]. Available: https://rtraba.files.wordpress.com/2015/05/cppturing.pdf

[16] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 209–216.

[17] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[18] J. Bromley, "If systemverilog is so good, why do we need the uvm? sharing responsibilities between libraries and the core language," in *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*. IEEE, 2013, pp. 1–7.

[19] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, "Llhd: A multi-level intermediate representation for hardware description languages," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 258–271. [Online]. Available: https://doi.org/10.1145/3385412.3386024

[20] S. Beamer and D. Donofrio, "Efficiently exploiting low activity factors to accelerate rtl simulation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[21] Cadence, *SimVision Tcl Commands*, Cadence.

[22] Synopsys, *Verdi® and Siloti® Command Reference*, Snopsys.

[23] S. Sutherland, "Modeling with systemverilog in a synopsys synthesis design flow using leda, vcs, design compiler and formality," *SNUG Europe*, 2006.

[24] Xilinx, *ChipScope Pro Software and Cores User Guide*, Xilinx.

[25] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: Concepts, overhead analysis, and implementation," in *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, ser. FPGA '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 188–196. [Online]. Available: https://doi.org/10.1145/1216919.1216950

[26] D. Kim, C. Celio, S. Karandikar, D. Biancolin, J. Bachrach, and K. Asanović, "Dessert: Debugging rtl effectively with state snapshotting for error replays across trillions of cycles," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 76–764.

[27] C. Alliance, "Treadle – a chisel/firrtl execution engine," 2022. [Online]. Available: https://github.com/chipsalliance/treadle

[28] ——, "Chisel/firrtl:treadle," 2022. [Online]. Available: https://www.chisel-lang.org/treadle/

[29] N. Calagar, S. D. Brown, and J. H. Anderson, "Source-level debugging for fpga high-level synthesis," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[30] D. R. Ditzel and D. A. Patterson, "Retrospective on high-level language computer architecture," in *Proceedings of the 7th Annual Symposium on Computer Architecture*, ser. ISCA '80. New York, NY, USA: Association for Computing Machinery, 1980, p. 97–104. [Online]. Available: https://doi.org/10.1145/800053.801914

[31] T. D. committee, "The DWARF debugging standard," 2017.

[32] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, "REPT: Reverse debugging of failures in deployed software," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 17–32. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/weidong

[33] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 377–389. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[34] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: Parallelizing sequential logging and replay," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, feb 2012. [Online]. Available: https://doi.org/10.1145/2110356.2110359

[35] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, 2016.

[36] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th SIGPLAN-SIGACT symposium on Principles of programming languages.* USA: ACM, 1988, pp. 12–27.

[37] R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with gdb," *Free Software Foundation*, vol. 675, 1988.

[38] J. Shore, "Fail fast [software debugging]," *IEEE Software*, vol. 21, no. 5, pp. 21–25, 2004.

[39] D. Kim, C. Celio, S. Karandikar, D. Biancolin, J. Bachrach, and K. Asanovic, "Reverse debugging of kernel failures in deployed systems," in *Second Workshop on Computer Architecture Research with RISC-V.* CARRV, 2018.

[40] X. Ge, B. Niu, and W. Cui, "Reverse debugging of kernel failures in deployed systems," in *2020 USENIX Annual Technical Conference (USENIX ATC 20).* USENIX Association, Jul. 2020, pp. 281–292. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/ge

[41] Google and Mozilla, "Source map revision 3 proposal," 2011.

[42] R. Hipp, "Sqlite," 2022. [Online]. Available: https://www.sqlite.org/

[43] Microsoft, "Debug adapter protocol," 2022. [Online]. Available: https://microsoft.github.io/debug-adapter-protocol/

[44] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 804–817. [Online]. Available: https://doi.org/10.1145/3445814.3446712

[45] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, 2004, pp. 69–70.

[46] U. Hölzle, C. Chambers, and D. Ungar, "Debugging optimized code with dynamic deoptimization," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, ser. PLDI '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 32–43. [Online]. Available: https://doi.org/10.1145/143095.143114

[47] The LLDB Team, "The lldb debugger," 2022. [Online]. Available: https://lldb.llvm.org/

[48] C. E. Cummings, "" full_case parallel_case", the evil twins of verilog synthesis," in *Proc. SNUG Boston Meeting*. Citeseer, 1999.

[49] Cadence, *JasperGold Platform and Formal Property Verification App User Guide*, Cadence.

Appendices

Appendix A

hgdb User Manual

This chapter serves as a user guide for hgdb users and developers. The framework is designed to be plug-and-play, that is, users only need to add/change a few command line flags to most simulators without modifying any test environment. Although macOS works with hgdb, it is not the primary operating system on which most RTL simulators run. As a result, we focus on Linux operating system in this user guide.

A.1 Setting up hgdb

A.1.1 Installing hgdb

The easiest way to install hgdb into your system is through Python pip. Users can use the following command to install hgdb into the current Python environment:

```
pip install libhgdb
```

Then users can find the installed library path using the following command:

```
python -c "import pkgutil; print(pkgutil.get_loader('libhgdb').path)"
```

To install it from scratch, you need a C++20 compatible compiler as well as cmake. Here is a list of steps:

1. Clone the source from Git

   ```
   git clone --recurse-submodules https://github.com/Kuree/hgdb
   ```

2. Create a build directory:

   ```
   mkdir hgdb/build
   ```

3. Change the current directory to the build folder

```
cd hgdb/build
```

4. Run the cmake command with optimization on

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

5. Build everything. Use `-j` if you want to compile with multiple cores

```
make
```

6. Run the builtin tests to ensure everything is working

```
make test
```

7. The library is located at `build/src/libhgdb.so`

A.1.2 Adding hgdb to the simulators

For simulators that are SystemVerilog compliant, you can use hgdb without modifying any testbench code.

Assuming the hgdb library path is `$LIBHGDB_PATH`, we need to use the following flags for each simulator:

Cadence® Xcelium™:

```
xrun [commands] -access +r -loadvpi $LIBHGDB_PATH:initialize_hgdb_runtime
```

Note that you need to give at least read access to the simulator. If you want to set values through hgdb, a write access is required.

Synopsys VCS®

```
vcs [commands] -debug_acc+all -load $LIBHGDB_PATH
```

Note that `-debug_acc+all` slows down the simulation significantly. For most cases, `-debug_access+class` is sufficient.

Mentor Questa®

```
vsim [flags] -pli libghdb.so
```

Verilator Because Verilator is not SystemVerilog compliant, the set up is a little bit tedious. First, we need to generate the "verilated" files with extra VPI flags:

```
verilator [flags] --vpi $LIBHGDB_PATH
```

In addition, most signals should be labeled as public, otherwise breakpoints and frame inspection will not work. An easy way is to use `--public-flat-rw` flag when invoking `verilator`. In addition to the flags, we need add following code to the test bench:

- Forward declare the runtime call

  ```
  namespace hgdb {
      void initialize_hgdb_runtime_cxx();
  }
  ```

- At the beginning of the test bench code, call the following function

  ```
  hgdb::initialize_hgdb_runtime_cxx();
  ```

  Also make sure `argc` and `argv` are properly passed to verilator:

  ```
  Verilated::commandArgs(argc, argv);
  ```

- At each posedge of the clock, we need to call a specific callback:

  ```
  VerilatedVpi::callCbs(cbNextSimTime);
  ```

Icarus Verilog

Icarus Verilog only takes shared library with .vpi extension. As a result, it is a good idea to simply symbolic link libhgdb.so to libhgdb.vpi in the current working directory, using the following command

```
ln -s $LIBHGDB_PATH libhgdb.vpi
```

When you run the compiled circuit with `vvp`, use the following command:

```
vvp -M. -mlibhgdb [commands]
```

A.1.3 Runtime options

You can change the runtime settings using plus-args when invoking the simulator. Here is a short list of options you can change:

- `+DEBUG_PORT=num`, where `num` is the port number. By default, this is 8888
- `+DEBUG_LOG=1`, which enables the debugging log. Useful when debugging the behavior of the runtime

There are several predefined environment variables one can use to debug the runtime. It is not recommended for production usage:

- DEBUG_DISABLE_BLOCKING: when present, will disable the initial blocking. As a result, the simulator will start execution without user's explicit "start" or "continue" command.
- DEBUG_BREAKPOINT#=filename:line_num@[condition]: where # counts from 0. The runtime will query the predefined breakpoints starting from 0 and stop if corresponding environment variable name is not found. condition is optional.
- DEBUG_PERF_COUNT: when present, the system will collect performance information. This is valid only when the library is build with -DPERF_COUNT=ON when invoking cmake.
- DEBUG_PERF_COUNT_LOG: when set, the system will dump the performance data into the set value instead of stdout.

A.1.4 Builtin tools

hgdb also comes with a set of tools to make debugging and tooling development easier. Here is a list of tools that ship with the Python package:

- hgdb-replay: waveform replay tool
- hgdb-db: symbol table debugging tool
- hgdb-rewrite-vcd: VCD rewrite tool based on the symbol table

hgdb-replay is a very useful tool to support reverse debugging. Due to license constraints, the public distributed hgdb does not ship with FSDB support. As a result, only VCD is supported. However, if built from scratch and $VERDI_HOME is detected, hgdb-replay will support FSDB automatically. The usage of hgdb-replay is shown below:

```
Usage:  [options] filename

Positional arguments:
filename                Waveform file in either VCD or FSDB format [required]

Optional arguments:
-h --help               shows help message and exits
-v --version            prints version information and exits
--db                    [default: false]
-p --port               Debug port [default: 0]
--debug                 [default: false]
-s --start-time         When to start the replay [default: 0]
```

Once `hgdb-replay` is running, users can directly use debuggers to connect to it as if it is a RTL simulator. The only limitation is that setting signal value is not allowed.

`hgdb-db` is a tool for generator framework developers to debug generated symbol table. It mimic how the hgdb runtime queries the symbol table and allows developers to isolate malformed symbol table entries. The usage of `hgdb-db` is shown below:

```
$ hgdb-db
hgdb> help
Commands available:
    - help
    This help message
    - exit
    Quit the session
    - load <path/uri>
    Load symbol table
    - instance
    Instance data query
    - breakpoint
    Breakpoint data query
    - context
    Context variable query
    - generator
    Generator variable query
    - assign
    Assignment information query query
```

`hgdb-rewrite-vcd` can rewrite VCD using source-level symbols. For instance, Chisel's bundles are typically flattened into individual nets. `hgdb-rewrite-vcd` can rewrite them into proper hierarchy. The usage of `hgdb-=rewrite-vcd` is shown below:

```
hgdb-rewrite-vcd [options]

Optional arguments:
-h --help               shows help message and exits
-v --version            prints version information and exits
-i --input-vcd          Input VCD file [required]
-d --debug-db           Debug symbol table [required]
-o --output-vcd         Output VCD file [required]
```

A.2 Use a compatible debugger

A.2.1 Visual Studio Code

Visual Studio Code (VSCode) is the reference implementation of a hgdb-compatible debugger. It supports the following features:

- Set/remove breakpoints
- Read-Eval-Print Loop (REPL)
- Multiple instances view
- Complex data type rendering
- Reverse debugging
- Data breakpoint (watchpoints)

To install the debugger extension, simply run the following command in the VSCode console:

```
ext install keyiz.hgdb-vscode
```

Users should expect the same debugging experience as debugging any program in Visual Studio Code. This is because it implements the majority of the adapter protocol. An example of `launch.json` is shown in Listing A.1, which provides an example of debugger configuration. In the example, we use debugging port 8888, which is the one used by default in hgdb. We also use a macro to input the symbol table. You can also change it to an absolute path. Note that your current working directory must contain the source code, otherwise the extension will run into errors when trying to open up the file upon breakpoint hits.

A.2.2 Console-based debugger

The console version is implemented in Python and mimics the style of `gdb`. It uses built-in Python-bindings to communicate with the hgdb runtime. To install this debugger, simply do

```
pip install hgdb-debugger
```

After installing, `hgdb` will show up in your `$PATH` since it's installed as a Python's package.

Most of the commands are identical to those of gdb. Type `help` to see a list of commands.

Below is an example usage, where we connect to localhost with port number 8888. The symbol file is debug.db.

```
hgdb localhost:8888 debug.db
```

```json
{
    "version": "0.0.5",
    "configurations": [
        {
            "type": "hgdb",
            "request": "launch",
            "name": "HGDB Debugger",
            "program": "${workspaceFolder}/${command:AskForProgramName}",
            "runtimeIP": "0.0.0.0",
            "runtimePort": 8888
          }
    ]
}
```

Listing A.1: Example of `launch.json` to set debug configurations for VSCode

If the symbol table is compiled on a different machine where the simulator runs, but the source code is local, you need to specify the file mapping using `--map`. An example would be `--map /remote/dir:/local/dir`, where the first `dir` is remote source location and the second is the local source location.

Note that for Chisel users, you need to specify a working directory so that the debugger can locate the source files. This is because Chisel only encodes the basename of a file, which makes it impossible to resolve without a working directory as a reference. You can use `--dir [folder]` to specify it. You can also use –map to remap the filename, e.g. `--map :/absolute/dir` by providing an empty remote directory.

A.3 hgdb runtime protocol

This section describes the RPC protocol between the debugger runtime and the debugger frontend. All the communications are over WebSocket and serialized with JSON, which is in the format of

```json
{
    "request": true,
    "type": "breakpoint",
    "token": "TOKEN",
    "payload": {

    }
}
```

The `token` field is used to identify an asynchronous communication request-response pair. That is, if a request uses a particular token, then the corresponding response will use the same token as

well. The client can use the token to identify the the corresponding response for a given request. Most of the details are in the `payload` field. Each packet also has a unique type identifier.

`Connection request` The connection request is the first request sent from the debugger frontend. It sets up the debugger runtime by providing the symbol table path.

```
    {
    "request": true,
    "type": "connection",
    "payload": {
        "db_filename": "/tmp/abc.db",
        "path-mapping": {
            "a": "/tmp/a",
            "b": "/tmp/b"
        }
    }
}
```

Note that `db_filename` is the path accessible by the simulator. `path-mapping` is optional.

Breakpoint request

```
{
    "request": true,
    "type": "breakpoint",
    "payload": {
    "filename": "/tmp/abc",
        "line_num": 123,
        "action": "add",
        "column_num": 42,
        "condition": "a"
    }
}
```

Action can be `add` and `remove`. `column_num` and `condition` are optional.

Breakpoint location request

```
{
    "request": true,
    "type": "bp-location",
    "payload": {
        "filename": "/tmp/abc",
```

```
        "line_num": 42
    }
}
```

You can narrow down the search via specifying `line_num` and `column_num`.

Simulation command request

```
{
    "request": true,
    "type": "command",
    "payload": {
        "command": "continue"
    }
}
```

`command` can be `continue` and `step_over`, `step_back`, `stop`, `reverse_continue`, and `jump`. Extra field of `time` is required if the command type is `jump`.

Debugger information request

```
{
    "request": true,
    "type": "debugger-info",
    "payload": {
        "command": "breakpoints"
    }
}
```

`command` can be `breakpoints`, `status`, `options`, `design`, and `filename`.

Path mapping request This request sets the path mapping, if it is not set during the connection phase.

```
{
    "request": true,
    "type": "path-mapping",
    "payload": {
        "path-mapping": {
            "/tmp/a": "/workspace/a",
            "/tmp/b": "/workspace/b"
        }
    }
}
```

The key in the `path-mapping` corresponds to the path in the symbol table and the value corresponds to the actual path on disk.

Monitor request


```
{
    "request": true,
    "type": "monitor",
    "payload": {
        "action_type": "add",
        "monitor_type": "breakpoint",
        "var_name": "hgdb",
        "breakpoint_id": 42
    }
}
```


`breakpoint_id` can be obtained from the breakpoint location request. `var_name` is the source-level symbol name. `action_type` can be either `add` or `remove`. `monitor_type` can be `breakpoint`, `clock_edge`, `changed`, `data`, and `delay_clock_edge`. If the type is `breakpoint`, `clock_edge`, or `delay_clock_edge`, hgdb runtime checks the value during breakpoint emulation loop, but the response are sent out differently:

- `breakpoint` sends out value if there is any breakpoint hits.
- `clock_edge` sends out value at every clock cycle.
- `delay_clock_edge` sends out value after a certain amount of delay. This is useful to emulate window-based buffer update or multiple cycle memory update.

`data` specify the data breakpoint, that is, a data breakpoint will be triggered given the request information. This is mostly used internally.

Set value request


```
{
    "request": true,
    "type": "set-value",
    "payload": {
        "var_name": "a",
        "value": 42,
        "instance_id": 43,
        "breakpoint_id": 44
    }
}
```

breakpoint_id can be obtained from the breakpoint location request. var_name is the source-level symbol name. This request is only valid if the underlying simulator supports overwriting signal values and appropriate flags have been turned on.

Data breakpoint request

```
{
    "request": true,
    "type": "data-breakpoint",
    "payload": {
        "var_name": "a",
        "action": "add",
        "breakpoint-id": 42
    }
}
```

Generic response If not specified, all the requests are replied with generic response. If there is no error, the response looks something like this:

```
{
    "request": false,
    "type": "generic",
    "status": "success"
}
```

If there is an error, the response looks something like this:

```
{
    "request": false,
    "type": "generic",
    "status": "error",
    "payload": {
        "request-type": "type",
        "reason": "ERROR"
  }
}
```

Where the request type and error reason are stored in the payload field.

Breakpoint location response This is the response for breakpoint location request.

```
{
    "request": false,
```

```
    "type": "bp-location",
    "status": "success",
    "payload": [
        {
            "id": 0,
            "filename": "/tmp/a",
            "line_num": 0,
            "column_num": 0
        },
        {
            "id": 1,
            "filename": "/tmp/a",
            "line_num": 1,
            "column_num": 0
        }
    ]
}
```

The payload is an array of breakpoints, where the `id` field refers to the breakpoint ID, which is a unique identifier for the breakpoint.

Breakpoint Response This is a packet that is broadcasted to all connected clients. The format is shown below:

```
{
    "request": false,
    "type": "breakpoint",
    "status": "success",
    "payload": {
        "time": 1,
        "filename": "a",
        "line_num": 2,
        "column_num": 3,
        "instances": [
            {
                "instance_id": 42,
                "instance_name": "mod",
                "breakpoint_id": 43,
                "process_id": 0,
                "local": {
```

```
                    "d": "5"
                },
                "generator": {
                    "c": "4"
                }
            }
        ]
    }
}
```

Note that the instances are an array of objects. Each instance has unique instance name and process ID. The process ID is used to identify a generator top instance. If the same generator top is instantiated multiple times inside the test environment, different process ids will be assigned. Local and generator variable stores the symbol information. The key is source-level symbol name. The value can be either a number or `ERROR`.

A.4 Symbol table format

hgdb supports two forms of symbol table: SQLite and JSON. At the time of writing, SQLite-based symbol table is deprecated and should not be used unless absolutely necessary. hgdb uses JSON schema to validate the symbol, which is hosted on Github. The definition for module instance object is shown below.

```
"instance": {
    "type": "object",
    "properties": {
    "name": {
        "description": "Instance name",
        "type": "string"
    },
    "module": {
        "description": "Module definition name",
        "type": "string"
    }
    },
    "required": [
        "name",
        "module"
```

```
    ]
}
```

The schema defines that `name` and `module` fields are both of string type and are required for the instance object.

A.5 Internals

A.5.1 Simulator interface implementation

This section lists the complete VPI implementation of simulator interfaces.

Function `get_design_hierarchy():`
  Result ← ∅;
  queue ← Queue();
  queue.emplace(null);
  while *!queue.empty()* **do**
    handle ← queue.front();
    queue.pop();
    handle_iter ← **vpi_iterate**(vpiModule, handle);
    while *child_handle ← **vpi_scan**(handle_iter)* **do**
      Result ← Result ∪ {GetInstanceInfo(child_handle)};
      queue.emplace(child_handle);
    end
  end
  return Result;

Algorithm A.1: How to implement simulator interface primitive `get_design_hierarchy` using VPI routines.

Function `place_callback_on_signal(`*rtl_signal_name, callback*`):`
  handle ← **vpi_handle_by_name**(rtl_signal_name, null);
  cb_data ← s_cb_data{.reason = cbValueChange, .cb_rtn = callback, .obj = handle};
  vpi_register_cb(&cb_data);

Algorithm A.2: How to implement simulator interface primitive `place_callback_on_signal` using VPI routines.

Note that because there are some minor differences in how each simulator implements VPI, we have to implement a shim layer to ensure the compatibility. The list below shows some examples of how the shim layer works:

- VCS runs the VPI registration routine during compiler time. If we do not check this, VCS compilation will stuck since hgdb runtime will blocking the execution while waiting for debugger connections. As a result, we only need to register the tasks, not the callback on clock edges during the compilation phase.

- At the time of writing, VCS does not support querying SystemVerilog interface types. As a result, we need to terminate the interface type query when VCS is detected to avoid VPI errors.
- Verilator treats SystemVerilog packed array as `vpiMemory` or `vpiNetArray`. As a result, we need to detect Verilator and make sure proper type is used when iterating through array elements.
- Verilator treats SystemVerilog interface as `Module`. As a result, the shim layer needs to make sure proper type is used when iterating through interface members.
- Verilator does not support getting instance's full hierarchy name. As a result, we need to keep track of all the hierarchy when searching generator instance during initialization phase.

Appendix B

kratos user manual

Kratos is an experimental hardware generator that's focused on debugging and usability. It has been used in both academia and industry for hardware design.

B.1 Features

B.1.1 Python AST

Kratos supports using plain Python syntax to generate complex hardware logic, using `@always_comb` and `@always_ff`. Below is an example of pipelined ALU using pure Python syntax.

Parameterization is also supported in the decorated function. Listing B.2 and B.3 shows the example of parameterization. In Listing B.2, we can declare function arguments as parameter and pass these argument values when calling `add_always`. This allows the same logic to be instantiated with different parameters. Listing B.3 shows how local variable can also be used to parameterize the logic. `b` is a Python variable that can determine the value of `a` inside the `always_comb` function. Note that although the these kinds of the parameterization happen at the Python level, kratos will store the parameters in the symbol table so that users can see how parameterization works at the source-level.

B.1.2 Finite state machine

FSM is a first-class entity in kratos. During compilation the FSM will be lowered into proper SystemVerilog constructs using the canonical FSM 3-block style. Listing B.4 shows an example code of how to construct FSM using `add_fsm` function call. In each FSM, we need to define the states, their corresponding outputs, and the state transition logic. By default it constructs a Moore machine, but can be converted to a Mealy machine at any time. Listing B.5 shows the generated RTL code. The FSM coding style can be easily picked up by any verification tools such as JasperGold or VCS

```python
from kratos import Generator, always_ff, always_comb, posedge, negedge

class ALU(Generator):
    def __init__(self, width=16):
        super().__init__("ALU", debug=True)

        clk = self.clock("clk")
        rst_n = self.reset("rst_n")
        data0 = self.input("data0", width)
        data1 = self.input("data1", width)
        instr = self.input("instr", width)
        out = self.output("out", width)

        out_temp = self.var("out_temp", width)

        @always_comb
        def alu_logic():
            out_temp = 0
            if instr == 0:
                out_temp = data0 + data1
            elif instr == 1:
                out_temp = data0 - data1
            elif instr == 2:
                out_temp = data0 * data1
            elif instr == 3:
                out_temp = data0 / data1

        @always_ff((posedge, clk), (negedge, rst_n))
        def pipeline_logic():
            if ~rst_n:
                out = 0
            else:
                out = out_temp

        self.add_always(alu_logic)
        self.add_always(pipeline_logic)


from kratos import verilog
verilog(ALU())
```

Listing B.1: Constructing hardware logic use plain Python syntax

```python
a = self.var("a", 4)
@always_comb
def logic(num):
    a = num

self.add_always(logic, num=2)
```

Listing B.2: Parameterization of the always function

```python
b = False
a = self.var("a", 4)
@always_comb
def logic():
    if b:
        a = 1
    else: a = 2
```

Listing B.3: Using Python variables to parameterize the logic

to detect FSM coverage. It can also be synthesized automatically without any FSM annotation in the synthesis tools. This is one of the benefit of supporting FSM as a first-class entity.

B.1.3 Compiler passes

One unique design feature of kratos is to support multi-stage generation through a combination of hardware parameterization and compiler passes. Users can add continue to construct hardware in Python after a compiler transformation is applied. To illustrate how it works, let's take a look at how we can lift specific ports up based on attributes and then connect the new ports in the parent module, as shown in Listing B.6. `lift_port` is a pass that checks every single port of a generator and make sure that only those with attribute of `"lift-port"` can be lifted up to the parent generator. Once we have obtained the target port, we can create the a new port in the parent generator using the same definition, as shown in Line 18. After applying the pass to the entire generator hierarchy as shown in Line 38, we proceed forward to construct more logic, as shown in Line 41. The generated RTL is shown in Listing B.7. Note that users can apply any passes in any order they want to transform the logic while constructing the hardware. Based on the feedback from people who used kratos to design hardware, this feature significantly improved their design productivity.

B.2 Internals

This section describes some of the kratos internals that might be interesting to the readers. Kratos implements some of the novel and efficient techniques in Python to obtain source-level information

```python
from kratos import verilog, Generator, always_ff, posedge

class FSMExample(Generator):
    def __init__(self, green_duration, red_duration):
        super().__init__("FSM")
        clk = self.clock("clk")
        rst = self.reset("rst")
        # add a traffic light FSM
        fsm = self.add_fsm("TrafficLight")

        # should traffic stop or not
        stop = self.output("stop", 1)
        fsm.output(stop)

        # use a counter
        counter = self.var("counter", 16)
        @always_ff((posedge, clk), (posedge, rst))
        def counter_logic():
            if rst:
                counter = 0
            elif counter > (green_duration + red_duration):
                counter  = 0
            else:
                counter += 1
        self.add_always(counter_logic)


        red = fsm.add_state("red")
        green = fsm.add_state("green")
        fsm.set_start_state(red)

        # state transition
        red.next(green, counter > red_duration)
        green.next(red, counter <= red_duration)

        # state outputs
        red.output(stop, 1)
        green.output(stop, 0)
```

Listing B.4: FSM construct written in Kratos, which will get lowered during compilation.

for hgdb.

B.2.1 Performance optimization

Because we need to obtain the caller source location, the easiest thing would be using Python's `inspect` package. However, it is very slow since Python needs to populate every single field of the

```
module FSM (
  input logic clk,
  input logic rst,
  output logic stop
);
typedef enum logic {
  green = 1'h0,
  red = 1'h1
} TrafficLight_state;
TrafficLight_state TrafficLight_current_state;
TrafficLight_state TrafficLight_next_state;
logic [15:0] counter;
always_ff @(posedge clk, posedge rst) begin
  if (rst) begin
    counter <= 16'h0;
  end else if (counter > 16'hF) begin
    counter <= 16'h0;
  end else counter <= counter + 16'h1;
end
always_ff @(posedge clk, posedge rst) begin
  if (rst) TrafficLight_current_state <= red;
  else TrafficLight_current_state <= TrafficLight_next_state;
end
always_comb begin
  TrafficLight_next_state = TrafficLight_current_state;
  unique case (TrafficLight_current_state)
    green: begin
        if (counter <= 16'h5)
          TrafficLight_next_state = red;
      end
    red: begin
        if (counter > 16'h5)
          TrafficLight_next_state = green;
      end
  endcase
end
always_comb begin
  unique case (TrafficLight_current_state)
    green: begin :TrafficLight_green_Output
        stop = 1'h0;
      end :TrafficLight_green_Output
    red: begin :TrafficLight_red_Output
        stop = 1'h1;
      end :TrafficLight_red_Output
  endcase
end
endmodule   // FSM
```

Listing B.5: Generated SystemVerilog from Listing B.4.

```python
from kratos import Generator, IRVisitor, Attribute, Port

def lift_port(generator: Generator):
    class Visitor(IRVisitor):
        def __init__(self):
            IRVisitor.__init__(self)

        def visit(self, node):
            if isinstance(node, Port):
                # test if it has an attribute
                if not node.has_attribute("lift-port"):
                    return
                gen = node.generator
                parent_gen = gen.parent_generator()
                if parent_gen is None:
                    return
                # copy the definition over and create a port at parent generator
                new_port = parent_gen.port(node)
                parent_gen.wire(node, new_port)

    visitor = Visitor()
    visitor.visit_root(generator.internal_generator)


def construct_hardware():
    child = Generator("child")
    p1 = child.input("p1", 1)
    p2 = child.output("p2", 1)
    child.wire(p1, p2)
    # only lift p1
    p1.add_attribute("lift-port")

    parent = Generator("parent")
    var1 = parent.var("var1", 1)
    parent.add_child("inst", child, p2=var1)

    # apply the pas
    lift_port(parent)

    # connect more variables
    var2 = parent.var("var2", 1)
    # connecting them using the lifted p1 port
    parent.wire(var2, parent.ports.p1)

    return parent
```

Listing B.6: Example code in kratos that continues hardware construction after compiler passes

```
module child (
  input logic p1,
  output logic p2
);

assign p2 = p1;
endmodule   // child

module parent (
  input logic p1
);

logic var1;
logic var2;
assign var2 = p1;
child inst (
  .p1(p1),
  .p2(var1)
);

endmodule   // parent
```

Listing B.7: Generated RTL code from Listing B.6

frame object before passing it to the user. Kratos uses a Python C extension to bypass that and directly returns the location, as shown in Listing B.8. Note that we first obtain the current stack frame object, then loop backward to find target frame. Then we directly return the source location after resolving the filename. Based on our benchmark, this implementation is orders of magnitude faster than `inspect` module.

Similarly, we have also implemented a Python extension method to obtain the frame local variables, as shown in Listing. We only need to populate the `f_locals` field of the frame object and then return it immediately after casting it to a Python dictionary.

B.2.2 AST transformation

There were two challenges of using Python abstract syntax tree (AST) to construct kratos IR:

1. how to execute the Python AST,
2. and how to obtain the source location of the AST.

To solve the first issue, we rewrote the Python code decorated by the `@always_comb` through Python's `ast.NodeTransformer`. Below is a list of transformations done to the Python AST:

- Statically elaborate the conditions. This is to support using Python variables to control the

```
std::optional<std::pair<std::string, uint32_t>> get_fn_ln(uint32_t num_frame_back) {
    // get caller frame info
    PyFrameObject *frame = PyThreadState_Get()->frame;
    uint32_t i = 0;
    while (frame->f_back && (++i) < num_frame_back) {
        frame = frame->f_back;
    }
    if (frame) {
        uint32_t line_num = PyFrame_GetLineNumber(frame);
        struct py::detail::string_caster<std::string> repr;
        py::handle handle(frame->f_code->co_filename);
        repr.load(handle, true);
        if (repr) {
            // resolve full path
            std::string filename = kratos::fs::abspath(repr);
            return std::make_pair(filename, line_num);
        }
    }
    return std::nullopt;
}
```

Listing B.8: Python extension code written in C++ to obtain the source location given number of frames back.

```
py::dict get_frame_local(uint32_t num_frame_back) {
    PyFrameObject *frame = PyThreadState_Get()->frame;
    uint32_t i = 0;
    while (frame->f_back && (++i) < num_frame_back) {
        frame = frame->f_back;
    }
    if (frame) {
        // PyEval_GetLocals(void)
        PyFrame_FastToLocals(frame);
        auto local = frame->f_locals;
        if (local) {
            py::handle obj(local);
            return obj.cast<py::dict>();
        }
    }
    return py::dict();
}
```

```
def pipeline_logic(scope):
    scope.add_stmt(
    scope.if_(~rst_n,
    scope.assign(out, 0)).else_(
    scope.assign(out, out_temp)))
```

Listing B.9: Transformed Python code block of `pipeline_logic` from Listing B.1

```
def pipeline_logic(scope):
    scope.add_stmt(
    scope.if_(~rst_n,
    scope.assign(out, 0, 4), f_ln=3).else_(
    scope.assign(out, out_temp, 6), f_ln=7))
```

Listing B.10: Debugging information such as the line number is added to the transformed Python AST shown in Listing B.9

generation. For instance, if an `if` statement is false, we replace the statement node with the `else` clause node.

- Loop unrolling. This transformation only happens when the debug flag is on. Note that this transformation happens at the Python AST-level, instead of at the kratos IR level.
- Statement to function calls. This transformation changes every statement into corresponding function calls. For instance, assignment statement is turned into `lhs.assign(hls)` and `if` statement is turned into `if_(predicate, *statements).else_(*statements)`.

Listing B.9 shows the transformed Python AST from Listing B.1. Note that we have created a scope object that is essentially an `always_comb` block in kratos IR. After obtaining the transformed AST, we convert it back to text code and run Python interpreter to execute the code.

Because each AST node the transformed code may have different source location, we cannot use the Algorithm 5.1. Instead, we use the fact that Python AST node contains the source location and use them as additional arguments to the transformed AST nodes. Listing B.10 shows the transformed AST when the debug mode is on.

For function call that takes arbitrary number of arguments such as `if_`, a keyword argument of `f_ln` is used, otherwise the line number is provided as an additional argument. All the line numbers are offset from the top of the function, which can be used to compute final line number.