ChiselTrace

Typed Behavioural Debugging in Modern Typed HDLs Through Signal Dependency Tracing

Jarl Brand

Master Thesis
MSc Computer & Embedded Systems Engineering

TU Delft

ChiselTrace

Typed Behavioural Debugging in Modern Typed HDLs Through Signal Dependency Tracing

by

Jarl Brand

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday, August 27th at 14:00.

Student:	J. Y. K. Brand: 5367980	
Project duration:	November 14, 2024 – August 27, 2025	
Thesis committee:	Prof. Dr. H. Peter Hofstee	TU Delft, IBM (supervisor)
	Dr. Charlotte Frenkel	TU Delft
	Dr. Zaid Al-Ars	Trinilytics (external)

TU Delft

Abstract

Debugging modern HDLs such as Chisel (Constructing Hardware In a Scala Embedded Language) remains challenging due to the lack of debugging tools operating on the source-language level. Furthermore, due to a lack of tooling, engineers often resort to manual waveform debugging, undermining productivity gains promised by such a language.

This thesis presents ChiselTrace, an open-source tool for Chisel that is capable of automatic signal dependency tracing at the Chisel source level, allowing faults to be more easily traced back to their root cause. The contributions of this work include the following. Modifications are made to the Chisel library to extract program dependence graphs and control flow graphs, and add instrumentation probes to the circuit, enabling post-simulation analysis. Furthermore, a library capable of dynamic program slicing and program dependence graph generation is introduced that is based on reconstruction from intermediate representation-level analysis. Finally, a front-end dependency graph viewer is presented, along with a method to automatically start a ChiselTrace session from failed ChiselSim unit tests.

The debugging capabilities of ChiselTrace are presented using a variety of test cases, including a real-world example, where an injected fault in the ChiselWatt processor is traced back to the source.

Preface

Throughout my years at TU Delft, I have had numerous encounters with hardware design, both during my undergraduate degree in electrical engineering and during my master's in computer engineering and embedded systems. One common theme that emerged while working on these projects was the great difficulty in debugging the hardware designs. Often, the only option to quickly find a bug in a hardware design is to write a test and inspect the waveform. More advanced tools are often proprietary and therefore not available.

When I approached Zaid for a master's thesis project, he introduced me to the Tywaves project: an open-source waveform viewer to improve debugging of Chisel hardware designs via source-level types in the waveform viewer. I had never used Chisel, but the promise of easier, open-source hardware debugging had me intrigued. The Chisel language brings many useful features from software development to hardware development; however, debugging still often happens at a lower level. With this in mind, my original goal was to improve the debugging experience at the source level. This eventually grew into ChiselTrace, an automated signal dependency analysis tool.

Throughout my time working on this thesis, I have had weekly meetings with the research group that I really enjoyed. I would like to give my thanks to my supervisor, Prof. Dr. Peter Hofstee, and Dr. Zaid Al-Ars for their excellent advice and the opportunities to present my work outside of the research group. Furthermore, I want to thank ir. Casper Cromjongh for the conversations we had about where to take the tool, and for presenting the work at various venues. Lastly, I would like to thank my parents and friends for lending me their ear to talk about my work.

Jarl Brand
Delft, June 2025

Glossary

ALU Arithmetic Logic Unit. 3, 40, 42

API Application Programming Interface. 35

ASIC Application-Specific Integrated Circuit. 1

AST Abstract Syntax Tree. 13, 14

CFG Control Flow Graph. 2, 4, 15–17, 19, 22–25, 28

CLI Command Line Interface. 39

DAG Directed Acyclic Graph. 33

DFS Depth-First Search. 30

DPDG Dynamic Program Dependency Graph. 2, 3, 11, 12, 15, 26–34, 37–40, 46, 48–51, 59

DUT Device Under Test. 9, 48

FIRRTL Flexible Intermediate Representation for RTL. 2–10, 15–28, 30–32, 37, 40, 48–51

FPGA Field-Programmable Gate Array. 1

GUI Graphical User Interface. 26, 33–35

HDL Hardware Description Language. 1, 2, 4, 6, 7, 13, 14, 50

HGL Hardware Generator Language. 1, 2, 4, 12, 13, 50, 51

HLS High-Level Synthesis. 1

IR Intermediate Representation. 2, 4–6, 16, 51

ISA Instruction Set Architecture. 4, 45, 56

JSON JavaScript Object Notation. 25, 27

LHS Left-Hand Side. 8, 22, 24

MUX Multiplexer. 18, 21, 22, 37, 39–41

PDG Program Dependency Graph. 2, 4, 11, 12, 15–20, 22–28, 30, 32, 39, 46, 48–50

RHS Right-Hand Side. 7, 8, 22, 23

RTL Register-Transfer Level. 1

VCD Value Change Dump. 10, 27, 28, 31, 32, 35

Contents

1 Introduction

1.1. Context

Since the end of Moore's law, it is no longer possible to rely on expected gains in raw processor performance and transistor density. To remedy this, the industry is turning to new device technologies, computing architectures, and accelerator design [2]–[4]. The rise of accelerators means that the complexity of and demand for hardware designs have risen in recent years. Accelerators are implemented either on field-programmable gate arrays (FPGA) or application-specific integrated circuits (ASIC). This means that they are often designed at the register-transfer level (RTL) using hardware description languages (HDL) such as Verilog and VHDL. These languages offer hardware description at a low level, without many abstractions, making hardware design time-consuming.

This has led to design methods that offer higher abstraction levels, such as high-level synthesis (HLS) and new HDLs, which are also known as hardware generator languages (HGL) [5]. HLS allows hardware designers to specify the functionality of a design in a high-level language such as C and then compile it into a RTL design.

HGLs, on the other hand, embed hardware construction primitives in a modern, general-purpose, host language. Instead of hand-coding RTL modules, designers write generator code that produces hardware descriptions. Examples of such languages include Clash [6], MyHDL [7], SpinalHDL [8], and Chisel [9]. These languages offer high-level abstractions that are often only found in software languages, such as functional programming, meta-programming, and polymorphism [5]. Furthermore, they typically compile to traditional HDLs, bringing productivity gains while retaining compatibility with existing tool chains. Among these languages, Chisel has been identified as the most popular [10].

Traditional HDLs benefit from a long history of debugging tools. Commercial tools, such as Synopsys Verdi [11], provide advanced tools for debugging. For example, it includes a debugger which is capable of pausing the simulation when a breakpoint is hit, much like a software debugger. Furthermore, it is capable of stack-frame inspection and reverse debugging. Other tools of note that are provided by vendor tooling are related to assisting a developer in finding the root cause of an observed erroneous value. These include active signal driver tracing and "temporal flow" views, capable of visualising data-flow between registers. In the open-source space, there has been less development in such tools. The tools include waveform viewers, such as GTKWave [12] and Surfer [13]. For testing and verification, there are tools such as SymbiYosys [14], VUnit [15], and cocotb [16].

Applying these tools to HGL-generated code brings many challenges. Generated HDL is often not easily human-readable and obfuscates the high-level abstractions of the source language. Flattened signals, optimised logic, and the lack of high-level constructs make it difficult to relate bugs in a design back to the source language. For this reason, debugging tools that operate at the source level have been a topic of active research. For example, HGDB [17] is a breakpoint debugger for Chisel. Additionally, the Tywaves [18] project makes an effort to reduce the gap between the Chisel source code and signals in the waveform viewer by displaying signals in their source type.

This thesis aims to further address this gap in debugging solutions by introducing ChiselTrace. ChiselTrace is an open-source debugging tool that enables automated signal dependency tracing through time and program slicing of behavioural designs at the Chisel source level. A method is introduced to construct dynamic program dependence graphs (DPDG) of Chisel circuits. Furthermore, by creating an interactive user interface to explore simulation-time signal dependencies, ChiselTrace seeks to reduce the time spent manually debugging by tracing wrong values back to their source in the waveform viewer, thereby reducing the gap in debugging tools, compared to proprietary tooling for classic HDLs.

1.2. Challenges

ChiselTrace aims to provide a way for hardware developers to debug their Chisel designs by automatically tracing back signal dependencies that occur during a simulation. This brings several challenges. First, Chisel is a complex hardware design language. Because a large portion of hardware may be generated using Scala code, it would be difficult to perform the required dependency analysis directly at the source level. In this thesis, the concept of mapping analysis performed on a lower-level intermediate representation back to a Chisel representation is explored.

A challenge that is associated with this is how a dynamic program dependence graph can be constructed at the intermediate representation (IR) level (namely, FIRRTL, Flexible Intermediate Representation for RTL). This requires knowledge of signal dependencies and a way to know which statements are active at a given time. A follow-up challenge is to map the graph in FIRRTL representation back to the Chisel representation.

Finally, it may prove difficult to visualise the generated graph in a way that benefits developers' debugging experience.

1.3. Problem Statement

The problem that this thesis addresses is the lack of source-level debugging tools for Chisel. This makes design debugging a time-consuming task, because designers have to either work with a limited amount of debugging tools or resort to debugging at the generated HDL level. Where Tywaves addressed the mismatch in signal representation between the waveform viewer and the hardware design, no academic effort has been made to reduce the amount of time spent in the waveform viewer tracing erroneous values back to their source statement, a process that is time-consuming and error-prone (see Section 2.2.1). This problem is what informs the main research question: How can debugging via manual waveform inspection be enhanced using automated dependency analysis at the HGL source level, bringing debugging closer to the source level and reducing the mental overhead for circuit debugging?

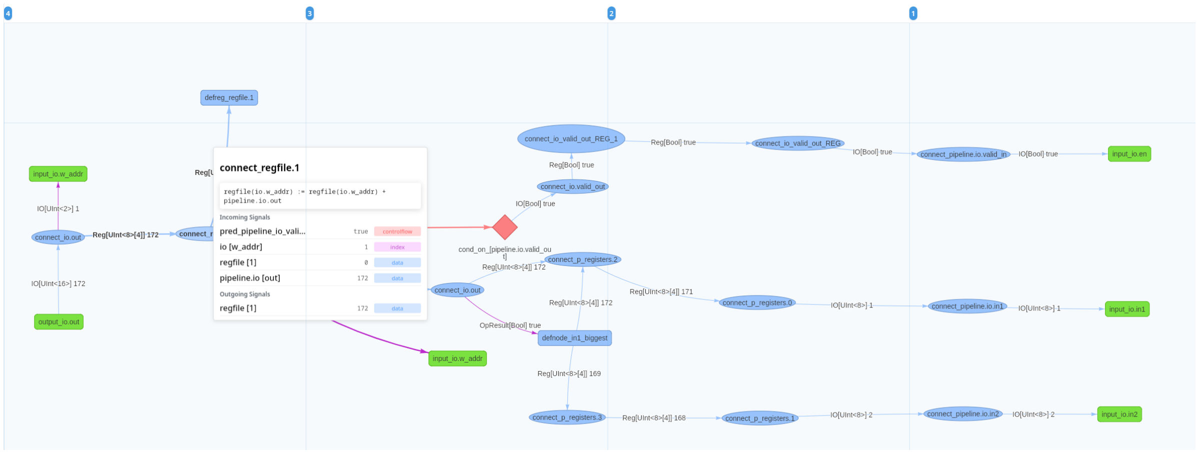
This overarching question is broken down into the following sub-questions that are answered in this thesis:

1. How can dependencies between Chisel statements be found?
2. How can dynamic data- and control-flow dependencies be resolved using simulation data?
3. How can a dynamic program dependence graph of a FIRRTL circuit be created?
4. Which steps are required to convert program dependence graphs from a FIRRTL representation to a Chisel representation?
5. What is an effective way to visualise a dynamic dependence graph for debugging purposes?

1.4. Contributions

The contributions of this thesis are as follows:

1. The Chisel library is modified such that it emits the program dependence graph (PDG) and control flow graph (CFG) of the FIRRTL circuit and inserts probes that are used for reconstructing control-flow and dynamic data-flow by downstream analysis.
2. A Rust library, `chiseltrace-rs`, that performs dynamic program dependence graph building and program slicing. The library is also capable of reconstructing a Chisel representation of the FIRRTL DPDG. Furthermore, it integrates with the `tywaves-rs` library to annotate the nodes of the DPDG with typed simulation values.



connect_regfile.1

```
regfile(io.w_addr) := regfile(io.w_addr) +
pipeline.io.out
```

Incoming Signals

pred_pipeline_io_vali...	true	controlflow
io [w_addr]	1	index
regfile [1]	0	data
pipeline.io [out]	172	data

Outgoing Signals

regfile [1]	172	data

Figure 1.1: Example of a ChiselTrace view of a circuit simulation.

3. A graph viewer that can be integrated with the aforementioned library. The viewer visualises the different types of nodes (connection, control flow, I/O) in the DPDG and the different types of dependencies between them (data, control flow, index). The graph viewer is interactive and shows Tywaves simulation data on graph edges to indicate data flow between nodes. Furthermore, source-code lines corresponding to the graph nodes can be inspected.

4. Modifications to ChiselSim and the tywaves-chisel library to integrate ChiselTrace with ChiselSim, allowing users to launch a ChiselTrace session automatically when an assertion fails.

An example of ChiselTrace is given in Figure 1.1. This example shows how ChiselTrace is capable of tracing various kinds of signal dependencies across clock cycles.

1.5. Thesis Outline

This document is organised in the following way:

- Chapter 2 provides background information about the Chisel language, its compilation chain, and the FIRRTL intermediate representation. Furthermore, the workings of Tywaves are detailed, and information is provided about the use of program slicing for debugging purposes. The chapter concludes with a discussion of work related to source-level debugging and dependency back-tracing.

- Chapter 3 Presents the changes that are made to the Chisel library to facilitate ChiselTrace functionality. In particular, a Chisel extension is introduced that allows for probe insertion and extraction of a program dependence graphs and control flow graphs at the FIRRTL level.

- Chapter 4 details the main implementation of the dynamic program dependence graph building and program slicing. Details are given on how information from the Chisel extension, the simulator, and compiler are synthesised into the graph that is used by the ChiselTrace front-end.

- Chapter 5 introduces a custom graph viewer front-end for ChiselTrace. Design choices regarding the presentation of the dynamic program dependence graph are discussed.

- Chapter 6 shows the results of this thesis. Various ChiselTrace functionalities are presented using example circuits. Then, these are put together in a small, but realistic arithmetic logic unit (ALU) circuit demonstration. Several scenarios for debugging using ChiselTrace are presented. Furthermore, a real-world example is given, where an injected fault in the ChiselWatt processor is traced back to the source using ChiselTrace. Finally, the performance scaling of the various components of ChiselTrace is discussed.

- Chapter 7 concludes this work by summarising the findings and presenting recommendations for future work.

2 Background and related work

This chapter presents background information about the Chisel HGL and Tywaves, a typed waveform viewer for Chisel. Details about the compilation and simulation chains, and FIRRTL, the IR that Chisel-Trace uses for PDG and CFG extraction, are provided. Furthermore, information on concepts related to the main topic, such as program slicing, and dependency back-tracing, is provided.

2.1. Chisel

Chisel [9] (Constructing Hardware In a Scala Embedded Language) is an HGL built on the Scala programming language. Where traditional HDLs focus on low-level modelling of circuits, Chisel aims to bring higher-level software concepts such as Functional / Object Oriented Programming, parametrisation, and type safety to hardware design. Furthermore, being embedded in Scala, it fosters the re-use of components, as would be the case for software libraries. Chisel has been used for numerous real-world designs, such as the Rocket Chip Generator [19], and the Berkeley Out-of-order Machine [20]. In this thesis, ChiselWatt [21], an OpenPower POWER v3.0 [22] instruction set architecture (ISA) processor design is used for evaluation purposes.

Chisel is implemented in Scala as a library and compiler plugin. It defines Scala objects that a designer can instantiate or derive from. For example, hierarchical modules can be defined by extending the `Module` class. An important feature of Chisel is the capability to represent complex, nested data types. For this purpose, the `Bundle` and `Vec` are used in combination with enums and ground types (such as `UInt`). The first two are akin to structs and arrays in other languages. Module I/O is also defined using these constructs, along with `Input` and `Output` objects. Chisel provides the `Wire`, `Reg` and `Mem` constructs to provide combinational and sequential data-flow, respectively. Furthermore, connections between (parts of) signals can be made if the types are equivalent. Chisel additionally provides many built-in data-flow and control-flow constructs, such as decoders and switch statements.

At its core, a Chisel circuit is written as a regular Scala class. By using the objects and functions defined in the Chisel library, Chisel can construct a hardware graph during runtime. This means that the user may use any Scala functionality to generate Chisel hardware objects at runtime, effectively making Scala a meta-programming layer for hardware generation. For example, one may make use of Scala's generic type system to generate variants of circuits. This also significantly complicates signal dependency tracking at the Chisel level.

2.1.1. Compilation chain

In this section, the compilation chain of Chisel is detailed. This is done by presenting a simple circuit in Listing 2.1 and seeing which transformations are performed to arrive at a Verilog version of the circuit that can be simulated and synthesised. An overview of the Chisel compilation chain is shown in Figure 2.1. Because Chisel is a library, the first step is to compile the Scala code [23]. The hardware graph is generated by the library at runtime.

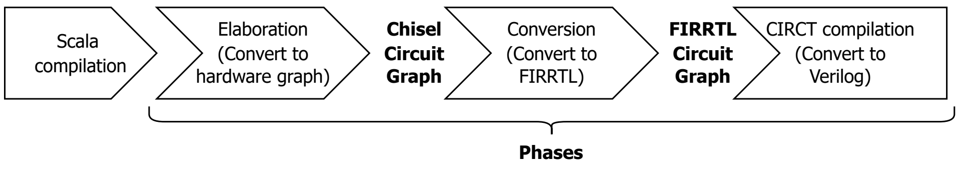

Scala compilation
Elaboration (Convert to hardware graph)
Chisel Circuit Graph
Conversion (Convert to FIRRTL)
FIRRTL Circuit Graph
CIRCT compilation (Convert to Verilog)
Phases

Figure 2.1: Chisel compilation stages. First, the Scala project is compiled. Afterwards, the circuit is compiled to Verilog in three main phases, of which CIRCT is an external compiler.

```scala
class AddOne extends Module {
    val io = IO(new Bundle {
        val in  = Input(UInt(32.W))
        val out = Output(UInt(32.W))
    })

    io.out := io.in + 1.U
}
```

Listing 2.1: Example Chisel circuit that adds the value 1 to the input and presents the result at the output.

This compilation of the Chisel circuit to Verilog is done in *phases*. The most important phases are the `Elaborate`, `Convert`, and `CIRCT` phases. Besides these phases, there exist phases for I/O and runtime sanity checks, etc. These phases are executed using the `PhaseManager` of the `Chisel-Stage` object. This phase manager ensures that all phases and their prerequisites are executed. Phases operate on sequences of annotations. Each phase takes in an annotation sequence and produces a new sequence of annotations. This is known as a transformation.

During the `Elaborate` phase, the Chisel `Module` being compiled is translated into a Chisel circuit graph (a graph that consists of primitive circuit elements), which is added to the annotations for the next phase as a `ChiselCircuitAnnotation`. It is also at this point that all hardware-generating Scala code is taken into account.

The Chisel circuit graph is then used by the `Convert` phase to produce a FIRRTL circuit graph. FIRRTL is an intermediate representation (IR) (see Section 2.1.2). This has the goal of simplifying and standardising the input for the `CIRCT` compilation phase. The FIRRTL version of the example circuit can be seen in Listing 2.2. The original Chisel circuit has been converted into a hardware graph, then converted into the FIRRTL IR.

```
public module AddOne : @[addone.scala 1:7]
    input clock : Clock @[addone.scala 1:7]
    input reset : UInt<1> @[addone.scala 1:7]
    output io : { flip in : UInt<32>, out : UInt<32>} @[addone.scala 2:16]

    node _io_out_T = add(io.in, UInt<1>(0h1)) @[addone.scala 7:21]
    node _io_out_T_1 = tail(_io_out_T, 1) @[addone.scala 7:21]
    connect io.out, _io_out_T_1 @[addone.scala 7:12]
```

Listing 2.2: FIRRTL version of the `AddOne` circuit. Temporary nodes are added to compute the intermediate results and source locators map the statements back to the Chisel source.

Finally, the FIRRTL circuit is converted to Verilog using the CIRCT [24] compiler, which is invoked via a phase. The result of this can be seen in Listing 2.3, where the example circuit is shown in Verilog representation. In addition to the generated Verilog, the compiler may generate debug information in the form of HGLDD files. These files contain structured JSON-like information about source-language mappings for the generated Verilog. This is not shown in the listing. As can be seen from the various representations, the signal naming and representation are not consistent across representations, which hinders the use of existing debugging tools for classic HDLs on Chisel circuits. The emitted debug information aids in reconstructing source-level data representations from the generated Verilog.

```verilog
module AddOne( // addone.scala 1:7
  input         clock, // addone.scala 1:7
                reset, // addone.scala 1:7
  input  [31:0] io_in, // addone.scala 2:16
  output [31:0] io_out // addone.scala 2:16
);

```

```
  wire [31:0] io_in_0 = io_in; // addone.scala 1:7
  wire [31:0] io_out_0; // addone.scala 1:7
  assign io_out_0 = io_in_0 + 32'h1; // addone.scala 1:7, :7:21
  assign io_out = io_out_0; // addone.scala 1:7
endmodule
```

Listing 2.3: Verilog version of the `AddOne` circuit. Source mappings are propagated.

2.1.2. FIRRTL

FIRRTL [25] is an IR that Chisel code is translated to before being passed to the CIRCT compiler for further processing to a Verilog circuit. In essence, it is a highly simplified HDL that can represent any Chisel circuit. In particular, all high-level constructs from Chisel are lowered into primitive operations. This makes FIRRTL a good target for circuit analysis. In this section, the structure of a FIRRTL circuit is explored. Furthermore, circuit statements of interest to this thesis are presented.

General circuit structure

Internally in the Chisel library, the FIRRTL circuit is represented as a circuit graph and is only converted to a text format when invoking the `CIRCT` compiler. This makes potential analysis on the FIRRTL circuit easier. Figure 2.2 shows how the FIRRTL circuit is represented internally in Chisel.

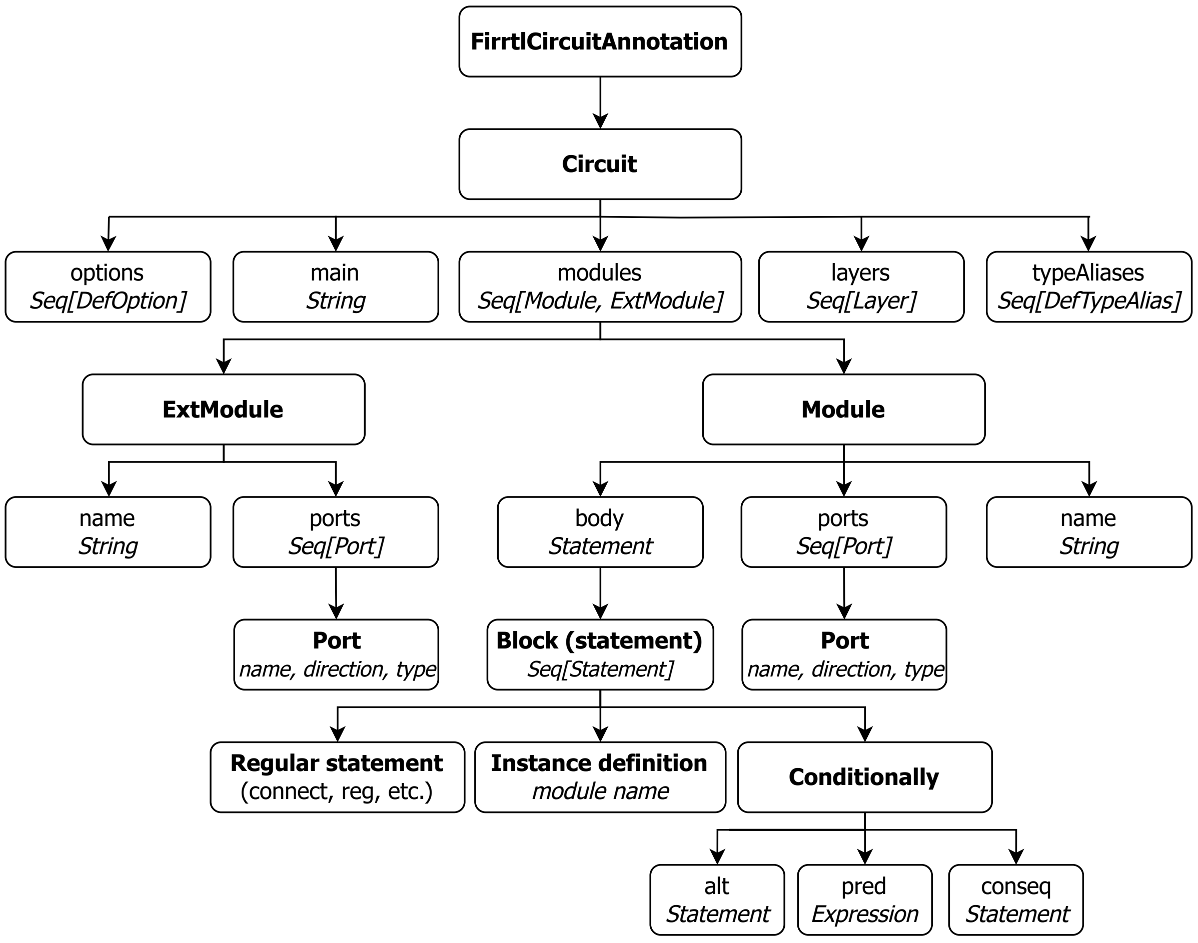


Figure 2.2: Structure of the FIRRTL circuit graph as it is available in the Chisel library. A selection of fields is shown. **Objects** are shown in bold.

A circuit in FIRRTL consists of modules. Modules are akin to modules in traditional HDLs, such as Verilog. They consist of circuit elements and can be combined in a hierarchical way to form a circuit. One module is the top-level module. The rest of the modules are used by instantiation. Every module can instantiate an arbitrary number of sub-modules.

There are two types of modules: regular FIRRTL modules and external modules. External modules only define port maps and are used to integrate existing Verilog code with a Chisel design. Regular modules consist of a body that is made up of statements. These statements are what define the circuit behaviour. In the figure, it can be seen that the body of a module is defined using a single `Block` statement, which contains any number of sub-statements. Modules can also contain layers that are meant to turn certain parts of the module on or off. These are defined at the circuit level in the Chisel library.

Annotations can be added to the circuit to add information about components, which can be used by the CIRCT compiler. In Chisel, these are represented as annotations separate from the `FirrtlCircuitAnnotation`. For example, these annotations can turn off optimisations for a particular statement, or add information regarding its type (see Section 2.2).

Data types

Before considering the various statements that are supported by FIRRTL, the data types of signals need to be considered. In FIRRTL, the distinction can be made between ground types and aggregate (or compound) types. Ground types, such as the `UInt`, represent a single value of zero or more bits. Compound types, on the other hand, consist of one or more ground- or compound-type sub-values. This allows FIRRTL to define arbitrarily complex data types, which is shown in Figure 2.3, where an example of a compound data type is given. There are three types of compound types: `Bundle`, `Vector`, and `Enumeration`. These correspond with the concepts of structs, arrays, and tagged unions, respectively. `Bundle` types may additionally mark a field as `flipped` to invert the data-flow direction. In the figure, flipped fields are indicated by red nodes.

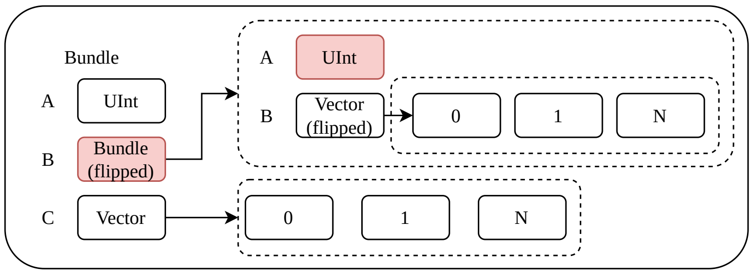


Figure 2.3: Example of nested data types in FIRRTL. Red nodes indicate that the direction of the signal is flipped.

Additionally, special types are defined for `Clock` and `Reset` signals, and `Analog` signals. The latter indicates that a port/wire may be connected to multiple drivers, for example, for data buses. Upon inspection, it can be observed that FIRRTL data types closely mirror the data types found in Chisel, resulting in a near one-to-one mapping of signals between the two (see Table 2.1).

Data-flow

FIRRTL supports combinational and sequential data flow through wires, registers, and memories. `Wire`s, like in other HDLs, represent memory-less data flow. They are defined with an explicit type and can be reassigned. `Node`s, on the other hand, are also memory-less, but are assigned once to an expression (`Expression`s are defined by operations, such as `add`, on operands). Their type is inferred by the right-hand side (RHS) expression. Module I/O also behaves as wires and can be defined using `input` and `output` statements. I/O statements must have an explicit type, and using a `flipped` field of a `Bundle` will flip the direction of the port. In the circuit graph, module I/O is defined separately from the module body.

Registers (`reg` and `regreset`) are used for sequential logic. Both of these are defined with an explicit type. Memories can be used to store data with an explicit type. There are two ways to define a memory. A short notation (`cmem` or `smem`, depending on whether the read behaviour is synchronous) can be used to create a memory element. Memories are accessed by creating ports using the `mport` statement. The direction of a port can be explicit (`read`, `write`, `readwrite`) or inferred. Another method of creating memories is by explicitly defining a memory. This allows for more configuration (such as read/write latencies, multiple read/write ports).

Data-flow is handled using `connect` statements. These statements assign to a left-hand side (LHS) expression the value of a RHS expression. The only constraint is that the two expressions must match in type. This also means that partial compound signals may be used in either expression. Furthermore, a RHS expression can be the result of an operation (such as `add`). One such operation of particular interest is the `mux` expression, which selects one of two signals based on a predicate value. This is a case where the data flow can only be determined at run-time. Another occasion where this happens is if either the LHS or RHS contains dynamically indexed vector expressions.

Multiple connect statements may assign to the same symbol. In this case, the last connect statement that did becomes the driving statement. Wires must be driven by a connect statement. Furthermore, combinational loops always result in an invalid circuit. Registers and memories update their value on a positive clock edge to the value to which they were connected. In case there was no connection to the component, they keep their value.

Control-flow

Control flow is handled almost exclusively through the `when` statement, which is a simple `if`-conditional, which operates on a 1-bit `UInt`. These conditionals always include a true-branch, and may or may not include a false-branch (else).

Another method of control-flow is the `match` statement, which is a pattern-matching statement that is meant to be used with the `Enumeration` data-type (similar to how Scala handles match statements). This construct is not used often in translated Chisel code.

In Figure 2.2, it can be seen that in Chisel, conditionals are statements that take in two branches. These can be `Block`s, allowing for more than one statement per branch. This effectively encodes the control flow graph of the circuit.

Other statements & features of interest

There are a number of statements that are of interest to ChiselTrace that have not been discussed yet. Firstly, modules are instantiated using an `inst` statement. This gives the symbol instance the type of the I/O ports of the module being instantiated. There are also statements available that allow for type aliasing, setting constants, or calling intrinsics. Intrinsics make use of specific functionality that is implemented in the CIRCT compiler.

Aside from these, FIRRTL implements Probes and Properties. Probes are constructs mainly meant for verification and allow read/write access to signals from outside of the module they are used in. Properties provide additional information about the circuit, but do not affect the functionality.

Relations to Chisel source code

In this section, we have seen how a FIRRTL circuit is built. In some aspects, the FIRRTL version of a circuit closely mirrors the Chisel version. This is further shown in Table 2.1. Here, a selection of types and key components is shown in both Chisel and FIRRTL representation. It can be observed that the core types and circuit components have a near one-to-one mapping between Chisel and FIRRTL. Primitive operations (e.g., `add`, `eq`, etc.) have similar mappings, but are not shown for brevity. Aside from the shown core components, Chisel provides standard library components. These are usually hardware generators that generate more primitive components, so these do not have a direct mapping in FIRRTL.

FIRRTL provides an additional link to the Chisel source in the form of source-locators. Namely, each statement contains information about the corresponding location in the Chisel source code in the form of a file, line number, and character number. This can be seen in Listing 2.2.

2.1.3. Chisel simulations

Traditionally, the functionality of hardware designs is tested in simulations of testbenches. The same is true for Chisel circuits. Testbenches are written in the source language (Scala + Chisel) and are executed in the form of unit tests. Using this approach, the hardware can be tested in a way that is similar to a software testing workflow. Furthermore, it enables integration with existing Scala testing tools and frameworks, such as the SBT [26] and scala-cli [27] `test` commands and the ScalaTest [28] framework.

Unit-testing is carried out using ChiselSim, which is a testing library that is integrated into Chisel. It targets Verilator [29] and VCS [30] as simulation back-ends.

Chisel	**FIRRTL**
Data types	
`UInt(n.W) / SInt(n.W)`	`UInt<n> / SInt<n>`
`Bool()`	`UInt<1>`
`ChiselEnum` (n options)	`UInt<log2(n)>`
`Bundle {val a = UInt(x.W) val b = UInt(y.W) }`	`{a: UInt<x>, b: UInt<y>}`
`Vec(x, UInt(n.W))`	`UInt<n>[x]`
Signal types	
`val a = Wire(UInt(n.W))`	`wire a: UInt<n>`
`val a = WireDefault(val<T>)` (value of type T)	`wire a: T` `connect a, val`
`val a = Reg(UInt(n.W))`	`reg a: UInt<n>, clock`
`val a = RegInit(UInt(n.W), 0.U)`	`regreset a: UInt<n>, clock, reset, UInt<n>(0h0)`
`val a = RegNext(val<T>)`	`reg a: T` `connect a, val`
`val a = Mem(x, UInt(n.W))`	`cmem a: UInt<n> [x]`
`val a = SyncReadMem(x, UInt(n.W))`	`smem a: UInt<n> [x]`
Control flow	
`when(condition)`	`when condition:`
`switch(value)`	series of `when` statements
Data flow	
`a := b`	`connect a, b`
`Mux(pred, a, b)`	`mux(pred, a, b)`
`vec(x)` (indexing `vec` with `x`)	`vec(x)`
Design hierarchy	
`class x extends Module` (module definition)	`module x:`
`val m = Module(new X())` (instantiation)	`inst m of X`

Table 2.1: A selection of the similarities between Chisel source code and the corresponding FIRRTL circuit.

Figure 2.4 shows the various components that are involved in a simulation with ChiselSim. The user writes a unit test that takes in a Chisel design. The simulation is started by calling a `Simulator`'s `simulate` method. This instantiates a `Simulator` object. A `Simulator` may perform actions such as setting additional simulation settings or exporting artifacts after the simulation has concluded. Chisel itself provides the `EphemeralSimulator`. This simulator does not have any additional functionality.

At the start of a simulation, any `Simulator` will set up a `Workspace`, which is responsible for generating the simulation environment. This includes compilation of the device under test (DUT), generating Verilog DPI definitions to interact with the DUT ports, and setting up the testing harness (a C++ simulation driver application). Finally, it creates a `Simulation`, which sets up the simulation process and a `Controller` object to interact with the simulation in the form of commands.

The `Controller` executes the commands defined in the unit test body. These commands are implemented using the `PeekPokeAPI`. The commands allow DUT ports to be read (peek) and written to (poke). Furthermore, they provide methods to advance a simulation by ticking a clock signal and performing assertions on the port values.

2.2. Tydi & Tywaves

The work presented in this thesis originates from the Tydi ecosystem (typed streaming interfaces) and Tywaves (typed waveform viewer). Tydi [31] is an open specification that defines a type system for variable-sized structures with compound data types and provides a way to transport them over hardware streams. Tydi-Lang [32] is a language for defining Tydi components, which is compiled to Tydi-IR [33], before VHDL is generated. Tydi-Chisel ports the Tydi concept to Chisel by [34] introducing a transpiler that converts Tydi-Lang into Chisel.

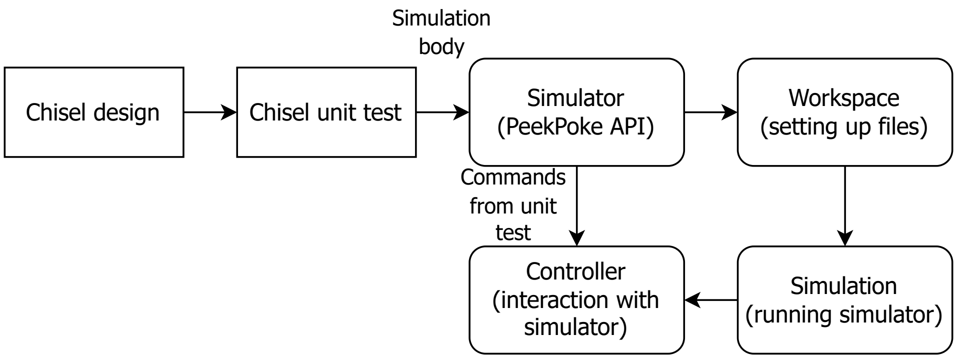

Simulation body
Chisel design
Chisel unit test
Simulator (PeekPoke API)
Workspace (setting up files)
Commands from unit test
Controller (interaction with simulator)
Simulation (running simulator)

Figure 2.4: The various components involved in running a ChiselSim simulation. Rectangular blocks are user-provided Chisel code, while rounded blocks are Chisel library components.

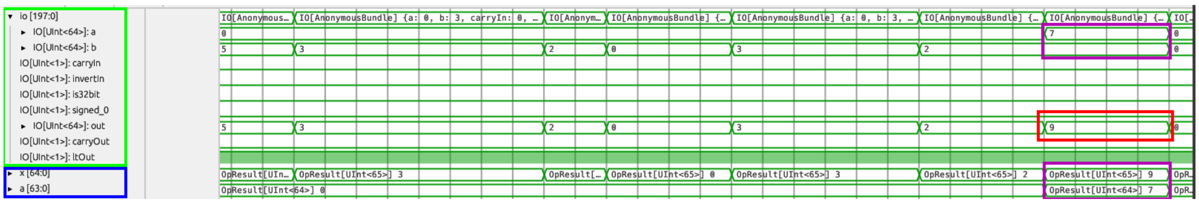


Figure 2.5: Tywaves waveform of debugging example. The fault is first observed in `io.out` after adding the `io` signal. After inspecting the source code, signals `x` and `a` are added. Here, we observe that the erroneous value comes from any of the value marked purple.

Tywaves [18] is a typed waveform viewer that originated from the Tydi ecosystem. Tywaves allows simulation data generated by Chisel simulations to be viewed in a waveform viewer in the source representation, instead of as Verilog signals (e.g., bundles and vectors are shown as compound signals, instead of separate signals; enums are shown by their name, not a numerical representation). This can be seen as a form of source-level debugging, which is further explored in this thesis in the form of ChiselTrace. An example of the Tywaves waveform viewer can be seen in Section 2.2.1.

Tywaves consists of three main parts: Chisel / CIRCT modifications, `tywaves-rs`, a Rust library to reconstruct the source-level representation of signals, and a translator for the Surfer waveform viewer. The Tywaves simulation- and compilation-flow is shown in Figure 2.6. During the circuit compilation process, an optional Chisel phase between `Elaborate` and `Convert` adds custom FIRRTL annotations to the circuit graph elements that contain the corresponding Chisel type. These annotations are then lowered by CIRCT, which associates the original type information with the generated Verilog signals and outputs this in the HGLDD file (debug information file).

The generated debug information is used by `tywaves-rs` to convert simulation data in the form of a value change dump (VCD) file from a Verilog representation to a Chisel source representation. It achieves this in two stages. First, the VCD file is rewritten to combine the changes of all individual signals belonging to a single compound signal into one bit vector. Then, this data is used to reconstruct the values of the original Chisel signal. This is what is displayed in the Surfer-Tywaves waveform viewer.

2.2.1. The current state of debugging with Tywaves

Debugging a design using Tywaves still involves manual actions. An example of this is provided in Figure 2.5. First, the observed fault is located in the waveform viewer (`io` is added to the viewer, and the value 9 is observed for `io.out`). This is done either by manual observation of the waveform or by inspecting the error from failed unit-test assertions. To proceed, the designer has to switch to the source code presented in Listing 2.4, where it can be seen that signals `a`, `x`, and `io.b` are relevant. The designer then has to look up the signals in the waveform viewer, add them to the view, inspect the values to decide which signals to debug next, and repeat the entire process for those signals. This method of debugging can quickly become time-consuming, especially if the fault-path crosses multiple hierarchical levels.

```scala
class Adder(n: Int) extends Module {
  val io = IO(new Bundle {
    val a          = Input(UInt(n.W))
    val b          = Input(UInt(n.W))
    ....
    val out        = Output(UInt(n.W))
    ....
  })

  val a = Mux(io.invertIn.asBool, ~io.a, io.a)
  val x = a +& io.b + io.carryIn

  ....

  io.out := x((n-1), 0)
  ....
}
```

Listing 2.4: Excerpt of code from the adder module of ChiselWatt [21]. Dots indicate code that has been removed for this example.

2.3. Program slicing

Program slicing [35] is a technique, used primarily in software, that allows for finding a subset of statements in a program, called a slice, based on some criterion. The criterion, C, consists of a statement and a list of variables. A program slice should produce the same values for these variables at the given statement as the original program containing all statements.

Using this definition, it can be seen why program slicing is of importance to debugging in general. Often, when debugging software or hardware, a problem is found where some variable or signal does not have the correct value after a certain statement. This can either be observed by tests or via manual inspection of the waveform (in the case of hardware). The error might occur in any statement that the criterion statement directly depends on. By reducing the number of possible erroneous statements to a slice of the program, finding the bug becomes easier.

This section discusses two main methods of program slicing. One requires only compile-time analysis, while the other also uses simulation data to further narrow the slice. Both techniques rely on the use of program dependence graphs (PDGs). We also discuss how the dynamic PDG (DPDG) that is used in ChiselTrace can be used on its own for debugging purposes.

2.3.1. Static & dynamic slicing

Using only information that is available at compile-time, a static program slice can be constructed. In order to do this, first a PDG is constructed. A PDG contains nodes for every statement in a program. The edges in the PDG signify dependencies between statements. An edge between two nodes is formed

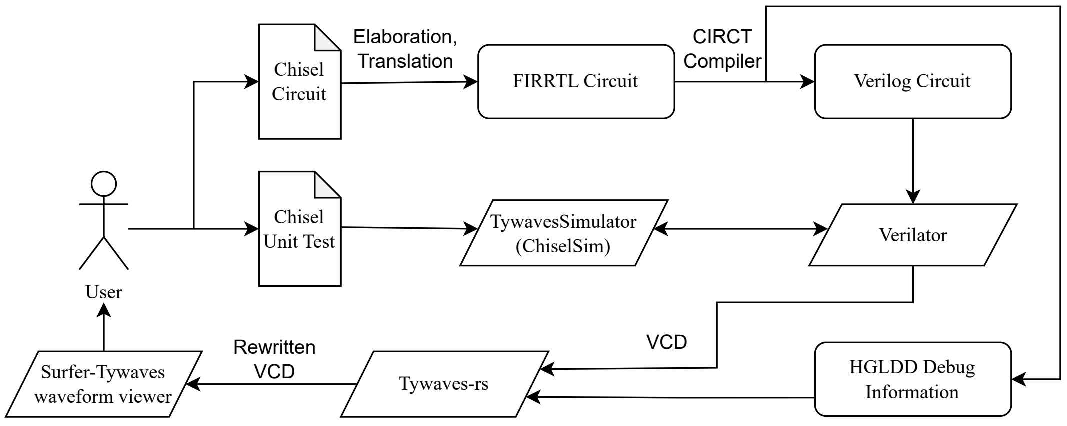
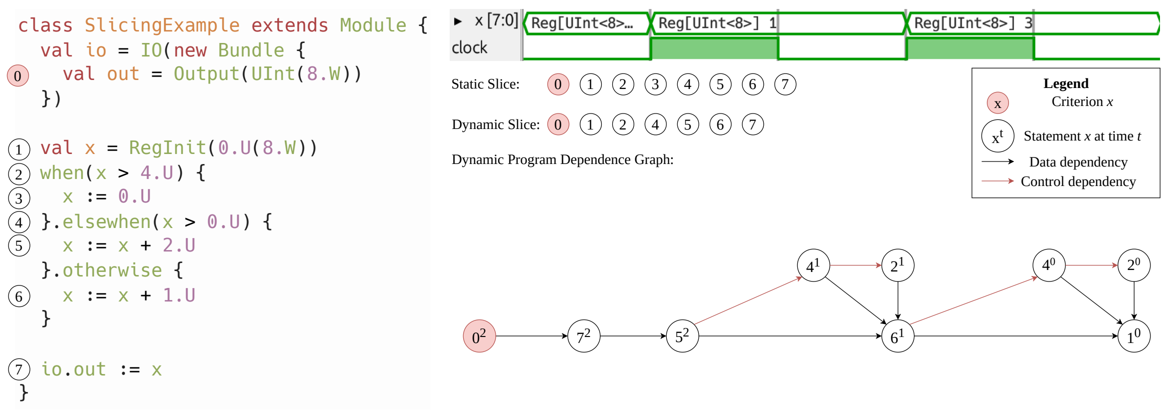


Figure 2.6: Tywaves simulation and compilation flow. The user first starts a simulation with Tywaves using its `Simulator`. After various stages, the user can inspect a waveform in the source representation.



```
  class SlicingExample extends Module {
    val io = IO(new Bundle {
0     val out = Output(UInt(8.W))
    })

1   val x = RegInit(0.U(8.W))
2   when(x > 4.U) {
3     x := 0.U
4   }.elsewhen(x > 0.U) {
5     x := x + 2.U
    }.otherwise {
6     x := x + 1.U
    }

7   io.out := x
  }
```

Figure 2.7: Program slicing applied to a Chisel circuit. The criterion is chosen to be the output port. The static slice contains all statements, while the dynamic slice only contains the statements in the DPDG.

when there is a data dependency or a control dependency. The data dependencies of a node N consist of all the nodes that provide a variable assignment that N depends on. The control dependency of a node consists of the control statement that directly influences node N (i.e., there is a maximum of one control dependency). When the PDG is formed, a slice can be constructed by listing all reachable vertices in the PDG from the criterion statement.

It is evident that the fewer statements there are in the slice, the more useful the slice becomes for debugging purposes. The technique mentioned above results in very broad slices, because every statement that could influence the criterion has to be included. By using runtime data, the slice can be reduced.

Dynamic program slicing [36] does exactly this. There are multiple ways to achieve this. One method still relies on static slicing. Namely, it first takes an intersection of the set of all statements in the PDG and the set of executed statements, then performs static slicing.

This method still has the issue that it does not take into account that separate occurrences may not have the same dependencies. In the context of hardware, some statement might depend on the register value x. However, the value of x may change over time, such that only the last assignment to x is relevant to the dynamic slice. This problem may be solved by constructing a DPDG based on the execution history. In this graph, each occurrence of a statement gets its own node and edges are made only to the dependencies of statements at a specific time.

Figure 2.7 shows static and dynamic program slicing, applied to a Chisel circuit. The criterion is chosen to be the output port `io.out`. Following the circuit's logic, the value of x starts at 0, then, after the first rising clock edge, gets updated to 1 by statement 6. One clock edge later, the value becomes 3 due to statement 5. At this point, the simulation is stopped, and the program slices are calculated. The static slice, which is based only on the PDG, shows all statements as having influenced the criterion. In reality, however, this is not what happened. We can observe that statement 3 was never executed during the simulation. This is reflected by the DPDG and the dynamic slice, which does not include the statement.

If the scenario above would have been a circuit in need of debugging, it would be clear to see why the dynamic slice is more useful than the static slice: it only includes statements that contribute to the criterion under a specific test case.

The objective of ChiselTrace is to improve source-level debugging by making use of automatic signal dependency analysis at the Chisel level. An observation that is made is that the DPDG itself is also a valuable representation for this purpose, as it shows which statements are executed per time step and their dependencies to previous time steps.

2.4. Related work

Tooling that functions on the HGL level is an active topic of research. This section explores works related to source-level debugging and other development tooling at the source level. In terms of debug tooling, two projects are HGDB [17], a GDB-like debugger for Chisel, and Tywaves [18], a typed waveform viewer. For Chisel, there also exists source-level tooling that is for the purpose of verification instead of debugging. Namely, ChiselVerify [37] aims to introduce formal verification for Chisel and introduces several coverage methods.

In addition to source-level tooling, works are presented that relate to the use of program slicing for debugging purposes in classic HDLs. Finally, we conclude by investigating existing tools for dynamic signal dependency tracing.

2.4.1. Source-level debugging

Two projects that can be classified as source-level debugging tools are the Hardware Generator Debugger, a debugger for HGLs such as Chisel, and Tywaves, a waveform viewer capable of reconstructing the source-level type of signals.

HGDB

HGDB (Hardware Generator Debugger) [17] is a debugger for HGLs. Specifically, a debugger implementation is presented for Chisel. Breakpoints are emulated, making the debugger simulator-agnostic. This works by evaluating breakpoints after each clock cycle. When a breakpoint is hit, the debugger is capable of reconstructing a “stack frame” (signals within the same scope as the breakpoint). Furthermore, the debugger gives insight into intermediate signal values (delta cycles) that would usually not be visible in an output simulation file. Moreover, it is also capable of reverse debugging within a clock cycle. HDGB provides two interfaces to interact with the debugger: a command line tool similar to GDB [38], and a Visual Studio Code extension.

Tywaves

Tywaves [18], as mentioned earlier, is a typed waveform viewer. Traditionally, Chisel circuits needed to be debugged at the Verilog level using tools such as GTKWave [12]. This resulted in a significant disconnect between the representation of the user-defined circuit and the circuit that was debugged. This issue was addressed by Tywaves, which is capable of reconstructing a view of Chisel signals in their original type from Verilog-level simulation data.

A source-level debugging research gap

Despite the efforts made in source-level debugging, finding faults in a hardware design remains a time-consuming manual effort. Whether a debugger is being used or a waveform viewer, the designer still has to diagnose the bug by manually tracing back the observed fault to the source.

Vendor tooling for classical HDLs contains tools to assist in this process, such as finding the active signal driver at a point in the simulation; however, similar tools have not yet been implemented for HGLs. This creates the gap that ChiselTrace aims to address using automatic dynamic dependence graph building at the source level.

2.4.2. Slicing & signal tracing in HDLs

In [39] and [40], static program slicing methods for VHDL and Verilog have been proposed, respectively. Furthermore, there has been an academic effort to use program slicing for hardware debugging purposes. Alizadeh et al. [41] introduce a debugging method with automatic error correction for HDLs that uses static program slicing as one method to reduce the search space. Dynamic program slicing on HDLs is explored in [42], where dynamic slices are calculated by combining static slices with coverage information and are used to accelerate fault injection. Lastly, program slicing and dynamic coverage analysis have been used for automatic fault localisation [43].

Commercial simulation tools such as Synopsys Verdi [11], Cadence Xcelium [44], and Siemens Questa Sim [45] offer capabilities to trace signal changes back to the active drivers through time. Furthermore, tools exist that allow for the visualisation of both sequential and combinational data flow at a hardware level. These tools also allow for automatic tracing of erroneous values to a root cause. ChiselTrace differs from these tools by enabling dependency analysis at the Chisel source level. This is an important distinction because the generated Verilog that these tools operate on is often dissimilar to the Chisel source code, making it difficult to relate these tools’ output to the original representation.

Ho et al. [46] introduce a method for abstract syntax tree (AST)-based back-tracing of erroneous signals. Nodes in the AST have an `LVALUE` and an `RVALUE`. These are expressions that get assigned to, and are dependencies, respectively. Based on the erroneous signal, an `RVALUE` tree is constructed, essentially forming a dependency tree for that specific signal. This dependency tree is used by a large language model agent to automatically diagnose bugs in a Verilog circuit. This AST-based method, however, is limited to static analysis.

All of the aforementioned works, however, are built for traditional HDLs, such as Verilog, and are therefore ill-suited for debugging Chisel designs due to their lack of source mappings. In addition, they lack the high-level type system that is used by Tydi and Tywaves.

3 Chisel modifications

This chapter introduces an extension to the Chisel library that enables the extraction of statement dependencies and the insertion of control- and data-flow probes at the FIRRTL level. Downstream ChiselTrace libraries require this information to reconstruct execution history from simulation data, and generate DPDGs and (dynamic) program slices. In this chapter, the choice is made to implement the circuit analysis that is required to do this in a Chisel phase. First, high-level design choices are presented that relate to where the circuit analysis is performed and the various alternatives that are available to implement it. Afterwards, the implementation details of the chosen methods are discussed.

3.1. High-level design choices

This section details the high-level design choices that are made during the design of ChiselTrace. In this section, various concepts relating to circuit dependency analysis in ChiselTrace are introduced, and the rationale behind them is explained. These concepts are described in more detail in the remaining sections of this chapter.

One of the challenges in the design of ChiselTrace is finding dependencies between statements at the Chisel source level. As we saw in Section 2.1, this would be difficult to do using static analysis of the Chisel code due to high-level constructs and the Scala meta-programming layer. However, Chisel first gets translated to FIRRTL, after which it is compiled to Verilog (see Section 2.1.1). FIRRTL contains source mappings, and during the compilation, these are mapped to the generated Verilog code. Furthermore, the different representations of the circuit are functionally equivalent. This means that the required analysis can be performed at either the FIRRTL level or Verilog level and can later be mapped back to a Chisel representation.

For Verilog, existing methods are available to build a PDG [47]. Execution history could be obtained by using a simulator that produces coverage information, such as Verilator [29]. However, situations could occur where the CIRCT compiler optimises parts of the input FIRRTL circuit, resulting in incomplete signal flow reconstructions of the original Chisel circuit.

FIRRTL, on the other hand, is much closer to the original Chisel circuit than the compiled Verilog circuit (see Section 2.1.2 for a comparison between Chisel and FIRRTL types and components). Furthermore, execution history can be reconstructed by inserting probe signals (note that these are simply FIRRTL nodes and are not the same as FIRRTL probes, which are meant for verification purposes) into the circuit for predicates of conditional statements. Combined with a CFG, this would allow for statement-level reconstruction of execution history (more information about this process is presented in Sections 3.2.2 and 4.3).

For these reasons, the FIRRTL level is chosen to perform the necessary analysis. This can be performed at any point in the Chisel compilation chain between the conversion of the Chisel circuit to FIRRTL and the compilation of FIRRTL to Verilog. This leaves three main options:

1. Integration as a Chisel phase after conversion to FIRRTL.
2. An external tool that operates between the Chisel and CIRCT phases.
3. Integration as a compilation pass within CIRCT.

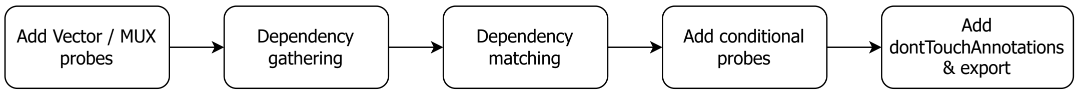


Figure 3.1: The stages of the PDG / CFG generation pipeline. Probes are inserted to enable dynamic analysis. ID-based matching is used to link nodes in the PDG.

Of these options, the second one can be immediately eliminated. Both integration in Chisel and CIRCT allow for the re-use of internal IR structures (such as the one presented in Figure 2.2). An external tool, on the other hand, would have to implement parts such as a parser and code generator from scratch, making this option impractical. An implementation in the Chisel library was chosen over CIRCT because this allows for faster prototyping and avoids the need to maintain a fork of the CIRCT compiler. This means that the information extraction back-end is written in Scala.

To build a PDG from a FIRRTL circuit graph, the FIRRTL specification[25] can be followed to determine dependencies, because the assumption can be made that the circuit is valid at this point (otherwise, it would not compile and the generated PDG would not be needed).

An overview of the PDG generation pipeline is shown in Figure 3.1. As can be seen, there are four main parts: probe insertion, dependency gathering, dependency matching, and exporting. The choice is made to use ID-based matching (generation of string IDs for symbolic statement dependencies and provided values) of FIRRTL statements to link statements with each other. The IDs are based on the symbols that refer to signals (e.g., if a `node` is named `x`, the ID is `x`).

As seen in section 2.1.2, FIRRTL supports compound datatypes, which can be arbitrarily complex due to the `Bundle`, `Vector`, and `Enumeration` types. To analyse dependencies between symbols of these types, the symbols are split into atomic symbols. This means that the dependency analysis is reduced to symbols of only a ground type. This is the same technique that FIRRTL compilers use and is referred to as scalarisation [25]. The IDs are used in a matching step to generate the required PDG. Furthermore, a CFG is generated from the circuit graph.

The choice is made not to track the dependencies of clock and reset signals. Instead, the assumption is made that all circuits are synchronous and have a global reset signal. This assumption will later simplify dependency tracking (see Chapter 4).

To supplement the static dependency analysis with dynamic data and allow reconstruction of execution history, probe signals are used. This is shown in Figure 3.2. The idea is that by inserting probe signals and relating them to branches in a CFG of the entire circuit and nodes/edges in the PDG, downstream analysis will be able to perform dynamic dependency analysis. Compiler optimisations are turned off on these probe signals. This idea is shown in the figure using two statements that assign to some wire `x`. Depending on whether the predicate for the conditional is true, the blue or green statement could drive `x` at a given time. By inserting the probe and associating it with the conditional in the CFG, it is possible to reconstruct which of the two statements provided `x` after the simulation is finished.

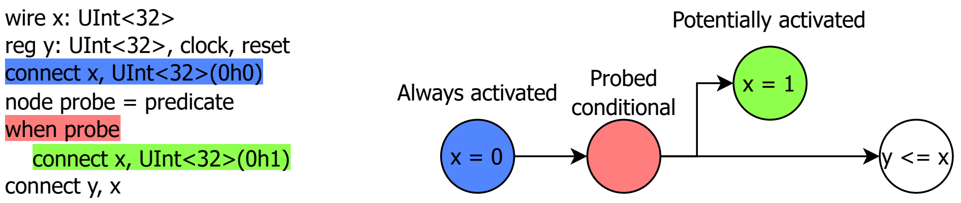

```
wire x: UInt<32>
reg y: UInt<32>, clock, reset
connect x, UInt<32>(0h0)
node probe = predicate
when probe
  connect x, UInt<32>(0h1)
connect y, x
```

Figure 3.2: An example FIRRTL circuit on the right with the control flow graph on the left. Register y gets a value from wire `x`. Depending on the control flow, `y` gets a value from the blue or green statement. Probed conditionals allow reconstruction using simulation data.

Note that reconstruction of control flow and execution history seems similar to line/statement coverage. There are existing methods for obtaining line coverage on the FIRRTL level [48] as well as multiplexer path coverage [37]. These methods, however, rely on custom FIRRTL cover statements. These statements then get converted to Verilog cover statements by CIRCT, which are tracked by

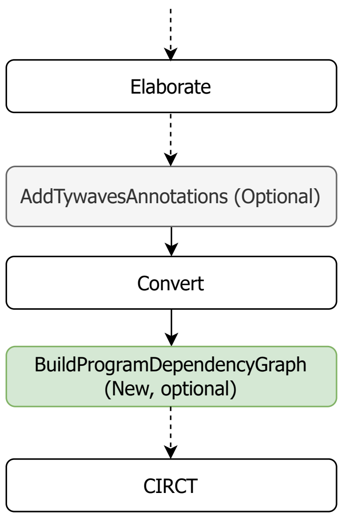


Figure 3.3: New `BuildProgramDependencyGraph` phase in the Chisel compilation pipeline. The most important phases are shown. Dotted arrows indicate one or more phases.

simulators such as Verilator [29]. The coverage, which only includes hit counts, is then exported by the simulator. Because only total hit counts are tracked, it is impossible to determine coverage on a per-cycle basis, hence the need for probe signals.

3.2. Implementation details

In the previous section, we saw that the PDG and CFG extraction, and probe insertion are performed in a Chisel phase at the FIRRTL level. In this section, more details are provided about the implementation of these parts. First, it is discussed how a Chisel phase can be added. Then, details are provided on how probes can be inserted into the FIRRTL circuit graph. After this, the generation of nodes and edges in the PDG is presented.

3.2.1. Adding a Chisel phase

As seen in Section 2.1.1, the Chisel compilation chain is carried out in phases. ChiselTrace adds a new phase to the compilation chain: `BuildProgramDependencyGraph`. This is shown in Figure 3.3. After the `Convert` phase, the Chisel circuit has been converted to a FIRRTL representation. This means that after this phase, a `FirrtlCircuitAnnotation` is available that contains the circuit, which can be accessed by subsequent phases. The CIRCT compiler is invoked during one of these phases to compile the FIRRTL circuit into a Verilog circuit.

With this in mind, the best place to insert the `BuildProgramDependencyGraph` phase would be between the `Convert` and `CIRCT` phases, because that would allow this phase to also still make changes to the `FIRRTL` circuit graph, which is required for adding probe signals. Furthermore, similar to the Tywaves phase, the `BuildProgramDependencyGraph` is made optional. This means that the phase only gets executed whenever the PDG / CFG debug information is required, preventing overhead during compilations where it is not required. The phase follows the aforementioned structure of Figure 3.1.

First, the entire FIRRTL circuit graph is scanned for nodes that contain dynamic vector indexing and multiplexor predicates. These are replaced with probe signals. After this, a PDG is built of the modified circuit graph. This part gathers dependencies for each statement, then matches them to form a PDG. This process also produces a CFG. After this, all conditional predicates are also replaced with probe signals. Finally, both the PDG and CFG are made serializable, and annotations for the PDG/CFG and the modified `FirrtlCircuitAnnotation` are returned. The following sections will go into more detail about each of these steps.

3.2.2. Probe signal insertion

To allow for downstream reconstruction of execution history and data flow of dynamically indexed vectors and multiplexors, probe signals are added to the circuit. The probe signal insertion happens at two locations in the phase. Before the PDG is generated, probe signals are inserted into the circuit for dynamic vector indexing and multiplexer (MUX) predicates. After the PDG generation, probes are added for conditionals. This behaviour is shown in Listing 3.1 and Listing 3.2. The former shows an example FIRRTL circuit that contains a control flow statement and dynamic vector indexing into a 2D vector. The latter shows the circuit after the probe insertion process. The expressions for the predicate and the vector indices are replaced with references to probe signals.

```
output io { flip addr1: UInt<4>, flip addr2: UInt<4>, flip en: UInt<1>, result: UInt<32>}
reg x: UInt<32>[16][16], clock, reset
connect io.result, UInt<32>(0h0)
when io.en
    connect io.result, x[io.addr1][io.addr2]
```

Listing 3.1: FIRRTL circuit with dynamic vector indexing and a control flow statement.

```
output io { flip addr1: UInt<4>, flip addr2: UInt<4>, flip en: UInt<1>, result: UInt<32>}
reg x: UInt<32>[16][16], clock, reset
connect io.result, UInt<32>(0h0)
node pred_io_en = io.en
when pred_io_en
    node probe_a = io.addr1
    node probe_b = io.addr2
    connect io.result, x[probe_a][probe_b]
```

Listing 3.2: The example circuit with probe signals added for control flow and dynamic indexing.

To insert the probes into the circuit, the body of each module in the FIRRTL circuit needs to be replaced with a version that has the probes added. This is achieved by mapping each module body using two functions. The objective of these functions is to recursively process the circuit graph, adding probe signals where necessary.

Processing of one FIRRTL statement is shown in Figure 3.4 (for the case of a dynamic vector index). Two helper functions are used. The first identifies any statements that may have an `Expression` associated with it (denoted with a blue node in the circuit graph in the figure). These are statements that could contain multiplexed assignments or dynamic indexing. When such a statement is found, the related expressions are processed by the second function. This function recursively processes the expression and produces a new expression containing references to new probe signals, if applicable, and a map containing the generated probe signals. For vector indexing, it searches for `SubAccess` expressions, while during MUX probing, it searches for `mux` expressions. Whenever such an expression is found, a random ID is generated that will serve as the probe signal name. Then, the original expression for the `SubAccess` index or MUX predicate is replaced with a reference to the newly generated probe signal. This is shown by the red node in the expression tree in the figure. Lastly, the old expression is added to the output map with the probe name as key. Note that an expression tree may contain multiple sub-expressions that need probing.

The map of generated probes is used by the first helper function to generate a `node` for each probe signal. These nodes are themselves recursively processed to remove all MUX predicates and dynamic vector indexing. An exception is the `mport` statement. This statement translates to a memory port and takes in an index expression without a `SubAccess` expression. To treat the memory ports similarly to how vectors are treated, the index expression of these statements is extracted and replaced by a probe signal.

Using the previously defined helper functions, one statement can be translated to a probed statement and a number of probe node statements. This means that adding probe nodes to all statements can now be achieved using a flatmap over all the module statements. Probe generation for conditional predicates is handled slightly differently. Namely, it takes place after the PDG generation. This way, the predicate probes do not end up in the PDG.

To ensure deterministic behaviour, a constant seed is used for the probe name generation. There is one last issue, however. Due to the way the probes are created, they can easily be optimised away by the CIRCT compiler. To prevent this, a `DontTouchAnnotation` is added to the FIRRTL annotations for each probe signal. This indicates to the compiler that the signals should not be optimised, allowing ChiselTrace functionality, even when compiler optimisations are otherwise enabled.

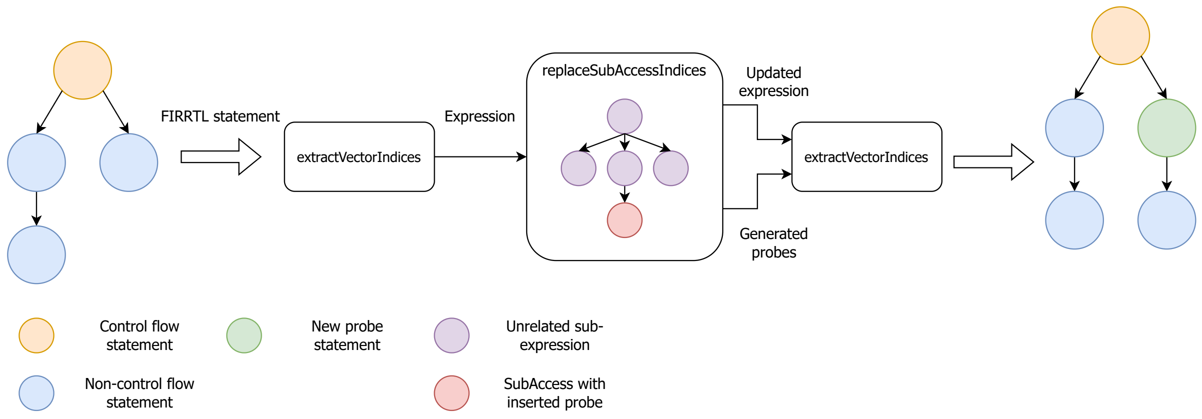


Figure 3.4: Probe insertion flow for one statement. The expressions in one of the statements are extracted. Then, the expression tree is scanned for sub-expressions that require probes. If they are found, probe names are generated, and the sub-expression is replaced with a reference. This is then used to generate new probe nodes in the circuit.

3.2.3. PDG node generation

Next, the PDG of the previously modified circuit is generated. First, a mapping is created between module names and the modules themselves. This will later be required for handling module instantiations. After this, two phases are executed: node generation and edge generation. During these phases, dependencies are gathered and matched, respectively. This subsection presents the former. Before describing the dependency gathering process, considerations regarding internal data structures and dependency scalarisation are discussed.

Data structures

In Section 2.1.2, we saw that the FIRRTL circuit graph is essentially structured like a CFG. It consists of modules that have a `Block` as body, which in turn contain `Statement`s. Conditional statements also contain `Block`s. The objective of the Chisel phase is to create a PDG and a CFG of the entire circuit. Keeping this in mind, we need to map the original circuit graph onto a structure that represents a CFG and allows for easy generation of a PDG.

This results in the following data structures: the `CFGStatement` and `CFGFork`. Both of these nodes extend from a `CFGNode` sealed trait. The data structures are shown in Listing 3.3. It can be seen that both types of `CFGNode`s contain a related statement. This is further elaborated on later. Additionally, the fork node contains information relating to the probe signal. This will later be used by `chiseltrace-rs` to determine the execution history. Furthermore, a fork node has two branches that contain statements. Together, these data structures allow for a CFG representation using a sequence of `CFGNode`s.

```scala
sealed trait CFGNode

case class CFGStatement(
    stmt: ConnectableStatement // Contains statement information
) extends CFGNode

case class CFGFork(
    stmt: ConnectableStatement,
    predSignalName: String, // Contains the signal name for the predicate probe
    hierPrefix: String, // Hierarchical prefix for the predicate probe
    left: Seq[CFGNode], // True branch
    right: Seq[CFGNode], // False branch
) extends CFGNode
```

Listing 3.3: CFG datastructures

The `ConnectableStatement` structure contains information about a FIRRTL statement. It is shown in Listing 3.4. Each `ConnectableStatement` contains the PDG node that is associated with the statement. The exact contents of this data structure are elaborated on in Table 3.1, where an overview of the exported data is presented. Furthermore, we can see that each statement can have dependencies and provide dependencies for other statements. Information about whether the statement is a register is required for determining if the connected PDG edges are clocked. Lastly, if the statement is a memory port, the direction of the port might be inferred. If that is the case, a “guess” is associated with the statement that is later resolved at the PDG edge generation stage.

```scala
case class ConnectableStatement(
    vertex: PDGVertex, // The PDG node that will be exported
    sourceModule: String, // The module that the statement belongs to
    dependencies: Seq[PDGDependency], // The symbolic dependencies of the statement
    provides: Seq[PDGDependency], // The symbolic dependencies the statement provides
    clocked: Boolean, // Indicates whether the statement is a register
    isInferredMemoryDirection: Option[InferDirection] // Used for inferred memory port
    directions
)
```

Listing 3.4: The `ConnectableStatement` data structure

There is one more critical data structure. Namely, the `PDGDependency`. Two types of dependencies can occur: the `RegularDependency` and the `ConditionalDependency` (see Listing 3.5). The former signifies a dependency that is always valid, while the latter indicates that a dependency is only valid if certain conditions on probe signals are met. This mechanism will be useful later on, when dealing with dynamic dependencies. Each dependency has a name. This is the dependency ID, which is used to perform the dependency matching. The name of the root symbol is also collected. This is the hierarchical path to the root of a compound symbol (e.g. if `io.a` is the name of the dependency, the root name would be `io`). Furthermore, the dependency should indicate whether it is flipped or not. A flipped dependency will cause a reverse connection direction and thus a reverse dependency direction.

The `connectID` is essential for connecting compound signals. Because all dependencies are scalarised, there must be some way to only give a `ConnectableStatement` the related dependencies. For example, if vector `x` of length N is assigned to vector y, scalarisation will lead to N `ConnectableStatement`s. However, we only want `x.n` to depend on `y.n` for every n if we assign `x` to `y`. This is achieved using the `connectID`. During scalarisation, each dependency is given an ID (e.g., vector index 0 will receive the ID “0”). Only if a dependency has the same `connectID` as the dependency being provided by the `ConnectableStatement` will the dependency be valid.

```scala
  case class RegularDependency(
    override val name: String, // Dependency ID
    override val rootName: String, // Path to the root symbol
    override val flipped: Boolean, // If a connection is flipped, the signal flow direction
    is reversed.
    override val connectID: String = ””, // Used for connecting nodes
    isIndex: Boolean = false
  ) extends PDGDependency

  case class ConditionalDependency(
    override val name: String,
    override val rootName: String,
    override val flipped: Boolean,
    override val connectID: String,
    conditionSignals: Seq[String], // Probes that impose a condition
    conditionValues: Seq[Int] // Condition values
  ) extends PDGDependency
```

Listing 3.5: The dependency data structures

Dependency scalarisation & symbol extraction

So far, dependency scalarisation (i.e., splitting compound signals into atomic dependencies) has been mentioned a few times. This section will go into more detail about how this is implemented. To see why dependency scalarisation is needed and which problems may arise, we can take a look at Listing 3.6, where a simple FIRRTL circuit is presented.

In the circuit, we can see that the I/O wire is of a `Bundle` type. The fields `sel` and `addr` are marked `flip` to invert the data flow direction. Furthermore, the registers `x` and `y` are bundles containing vectors. The value of node z is selected by the I/O, and ultimately, the result output is connected to a dynamically indexed vector.

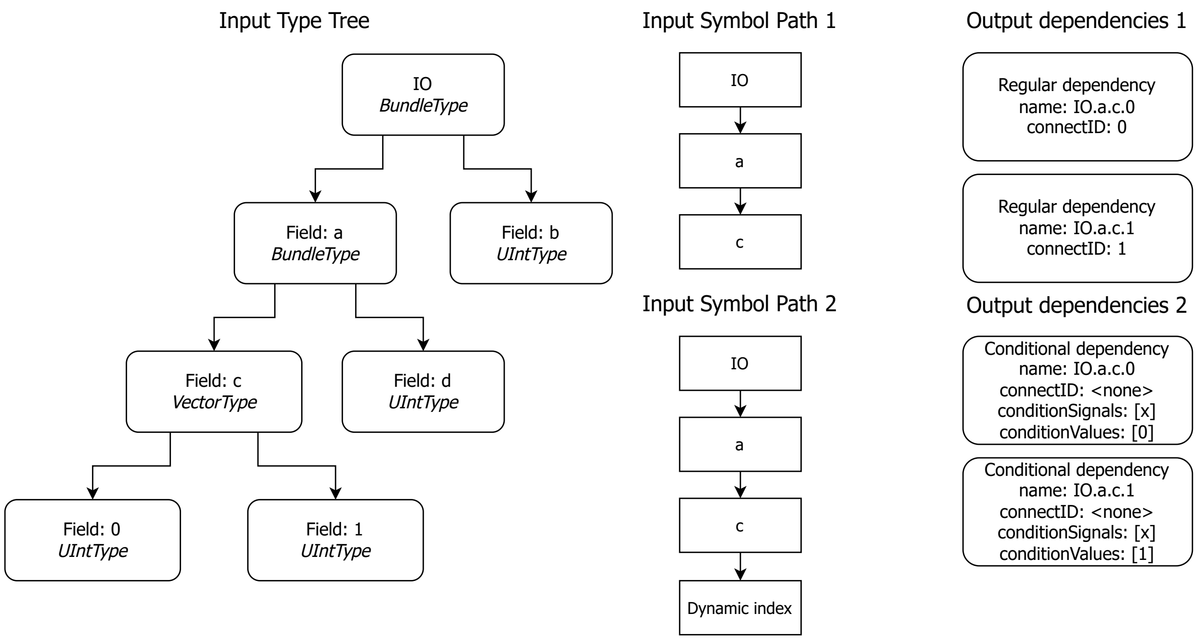

Input Type Tree
IO BundleType
Field: a BundleType
Field: b UIntType
Field: c VectorType
Field: d UIntType
Field: 0 UIntType
Field: 1 UIntType
Input Symbol Path 1
IO
a
c
Input Symbol Path 2
IO
a
c
Dynamic index
Output dependencies 1
Regular dependency name: IO.a.c.0 connectID: 0
Regular dependency name: IO.a.c.1 connectID: 1
Output dependencies 2
Conditional dependency name: IO.a.c.0 connectID: <none> conditionSignals: [x] conditionValues: [0]
Conditional dependency name: IO.a.c.1 connectID: <none> conditionSignals: [x] conditionValues: [1]

Figure 3.5: Dependency scalarisation using type tree traversal. Two example paths are shown with their desired outputs. When selecting an entire vector, dependencies for all elements are generated. When dynamically indexing a vector, the same happens, but now the dependencies have conditions imposed that can be resolved at simulation-time.

From this example, it is clear why dependency scalarisation is required. Ultimately, the result depends on a single element of `z`. That one element, in turn, depends on only one element of `x` or `y`. Without scalarising the dependencies, it would be challenging to represent this relation between the signals. However, there are multiple complications. First of all, the node `z` only depends on a subset of the signals from either `x` or `y`. Secondly, from the FIRRTL IR available in Chisel, it cannot directly be seen what the type of node `z` is, which complicates dependency scalarisation. Lastly, `io.sel` and `io.addr` introduce dependencies that can only properly be resolved if simulation data is available.

To remedy the first issue, while processing the statements, if a compound signal is encountered, it is stored in a map that associates the hierarchical path to the root symbol with the data type. Then, when this symbol is used in an expression, the type is known. Figure 3.5 shows how a list of dependencies can be generated by traversing the type tree using the type and local path (i.e., `.a.c` after `IO`). Note that the figure displays a different type tree for `IO` than the one used in Listing 3.6. The figure shows two examples of symbol paths being used to generate dependencies from the type tree. In the first example, the symbol that is used in the FIRRTL circuit would be `IO.a.c`. From the type tree, we can see that this field corresponds with a vector type. Therefore, two dependencies are generated with different `connectID`s. When these dependencies are then used while processing (for example) a connect statement, only field 0 of vector 1 is connected to field 0 of vector 2. In the second example, the symbol path `IO.a.c[x]` is used. This dynamically indexes the vector `c`. Therefore, two dependencies are generated with conditions. The first is only valid when the probe signal `x` has the value 0, and the second when the value is 1. During the type tree traversal, we also resolve whether the individual dependencies have a flipped direction. This is not shown in the figure.

The second issue is solved using type inference. FIRRTL has strict rules about type inference that allow the analysis code to know, based on the right-hand expression, what the type of node `z` must be.

The last issue is solved using the aforementioned `ConditionalDependency`, allowing for downstream insertion of dynamic data. Using the dependency scalarisation, the analysis simply adds all dependencies that could be a dependency for each node. Downstream processing can then invalidate these dependencies based on simulation data of the associated probe signals.

To extract symbols in the form of `PDGDependency` objects from an expression, logic is executed based on the expression type. If the expression refers to (part of) a compound signal, dependency scalarisation is applied. The expression may also contain an operation. This can be a primitive, MUX, or intrinsic operation. In these cases, the arguments to the operation are expressions that are recursively processed to extract dependencies. In the special case of the MUX, the dependencies from the two branches are made conditionally dependent on the MUX predicate.

```
output io { flip sel: UInt<1>, flip addr: UInt<2>, result: UInt<32>}
reg x: {a: UInt<32>[4], b: UInt<32>[4]}, clock, reset
reg y: {a: UInt<32>[4], b: UInt<32>[4]}, clock, reset
node z = mux(io.sel, x.a, y.b)
connect io.result, z[io.addr]
```

Listing 3.6: Example FIRRTL circuit. Dependency scalarisation is used to track individual signal dependencies within compound signals

Dependency gathering

During the dependency gathering stage, the entire circuit graph (see Figure 2.2) is recursively traversed, starting at the main module. First, the module's I/O is added to the compound data type map and if the module is the main module, PDG nodes are generated for the input and output wires. The clock and reset signals are ignored. If the module is an external module (e.g., a custom Verilog module), dependency analysis within this module is impossible, and the module is treated as a black box. For these modules, only I/O nodes are added to the PDG. If that is not the case, the body of the module (i.e., the statements that make up the module) is be further processed into a sequence of `CFGNode`s. This will be referred to as statement extraction. Before statement extraction, a list is made of all symbols that are registers.

The statement extraction has the goal of transforming a part of the FIRRTL circuit graph into the previously discussed `CFGNode` format. As seen in the previous section, every statement may provide multiple atomic dependencies. This potentially causes multiple nodes in the CFG for every processed statement. With this in mind, the most suitable method of converting a series of statements into `CFGNode`s is a flatmap over all of them with every statement mapping to a list of `CFGNode`s.

The dependency extraction works differently for all kinds of statements. This is summarised in Figure 3.6. Furthermore, a more detailed list is compiled of all the supported statements and how their dependencies are extracted:

- `Connect` statements are used to to connect a LHS sink expression with a RHS source expression. This is shown in Figure 3.7. First, the symbols on both sides of the connection are extracted from the expressions. After this, it is determined whether the connection is clocked or not, based on whether one of the LHS symbols is in the aforementioned list of clocked elements. For every LHS symbol, a `CFGStatement` is created. This means that there will also be one PDG node for every connection. Not every LHS symbol, however, depends on every RHS symbol (e.g., LHS element 0 of a vector only depends on element 0 of the RHS vector). The previously mentioned `connectID`s are used to solve this issue. Only if the `connectID` of the RHS symbol is the same as that of the LHS symbol, is it marked as a dependency. Furthermore, if a dependency is flipped, the RHS and LHS symbols will switch places.

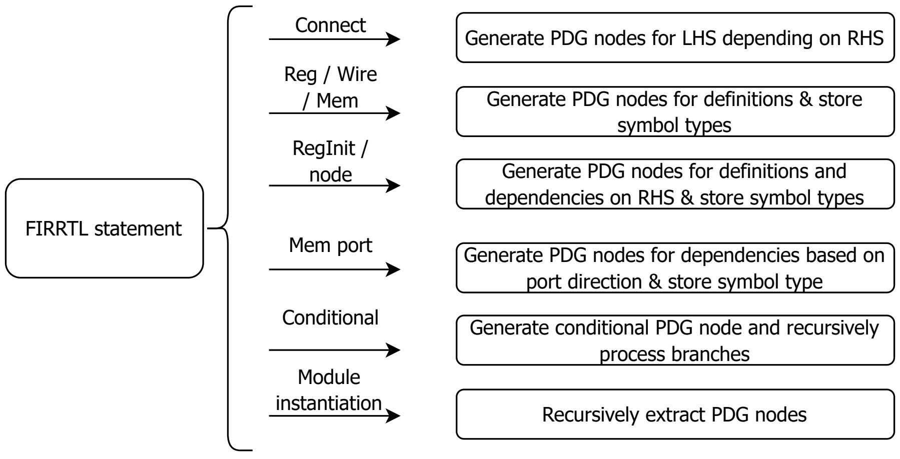

FIRRTL statement
Connect
Generate PDG nodes for LHS depending on RHS
Reg / Wire / Mem
Generate PDG nodes for definitions & store symbol types
RegInit / node
Generate PDG nodes for definitions and dependencies on RHS & store symbol types
Mem port
Generate PDG nodes for dependencies based on port direction & store symbol type
Conditional
Generate conditional PDG node and recursively process branches
Module instantiation
Recursively extract PDG nodes

Figure 3.6: Generation of PDG nodes for different kinds of FIRRTL statements.

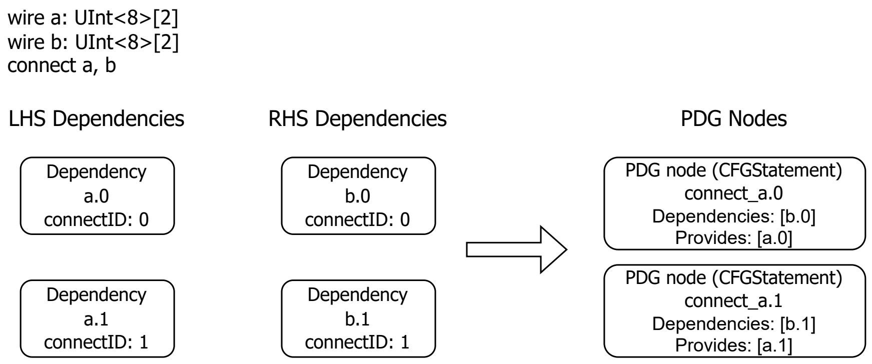

wire a: UInt<8>[2]
wire b: UInt<8>[2]
connect a, b
LHS Dependencies
RHS Dependencies
PDG Nodes
Dependency a.0 connectID: 0
Dependency b.0 connectID: 0
PDG node (CFGStatement) connect_a.0 Dependencies: [b.0] Provides: [a.0]
Dependency a.1 connectID: 1
Dependency b.1 connectID: 1
PDG node (CFGStatement) connect_a.1 Dependencies: [b.1] Provides: [a.1]

Figure 3.7: Conversion of FIRRTL connect statement to PDG nodes.

- Upon reaching a register or wire definition statement (`reg`, `reginit`, `wire`, `node`), the defined signal is added to the map of known compound signals, if applicable. CFG / PDG nodes are then created for the definition of each atomic signal. If a register has a reset value, the dependencies are handled similarly to how the connect statement works. In FIRRTL, nodes do not have an explicit type associated with them. Instead, they take on the type of the RHS value. This is problematic for compound signals because, without a known type, it is assumed that signals are of a ground type. This will result in too few PDG nodes being created and wrong dependencies when they are connected. For this reason, type inference is performed for nodes. If the node is determined to be of a compound type, the node itself is also registered as a compound signal.

- Memories are also supported by ChiselTrace. As seen in Section 2.1.2, memories work differently from registers in FIRRTL. Namely, there are memory definitions and memory ports. Memories can have either synchronous (i.e., SRAM) or asynchronous reading (i.e., a register file). During statement extraction, memories are treated the same as FIRRTL Vectors. FIRRTL also supports custom SRAM blocks. These blocks are treated as black boxes.

  Memory ports define a wire that is connected to a memory at a specific address. This is treated the same as vector indexing. Moreover, memory ports have a direction. This can be one of the following: `read`, `write`, `readwrite`, `infer`. When the port has a read or write direction, it is clear in which direction the dependencies are. Reading means that the port has a dependency on the memory, while writing means that the memory has a dependency on the port. With `read-write`, both directions are added to the PDG. Direction inference is more complicated, however. At the time that the memory port is processed, the memory direction is unknown. This is why both read and write nodes are added to the PDG. These nodes are marked with an `InferDir-ection`. This means that later, after the statement extraction is finished, the correct direction can be inferred, and the unneeded node will be discarded. Lastly, if the memory port is to a sequential memory, an assign delay is added to any read port. This will indicate to the downstream dependency reconstruction that the read value will be available one clock cycle later.

- Conditionals (`when` statements) are the main control flow statements. This statement will translate to a `CFGFork` in the CFG. Both branches of the conditional are recursively processed for statement extraction. Furthermore, a statement is generated for the predicate. This allows other nodes in the PDG to depend on the conditional using a special dependency. Lastly, the name of the probe signal is embedded in the newly created `CFGNode`.

- Modules in FIRRTL are also instantiated using statements. When a module instantiation statement is encountered, statements are recursively extracted.

- Lastly, there are minor FIRRTL features supported. Intrinsic statements are simply translated to a PDG node that depends on the input arguments. Layers are not handled, since it is expected that this happens at the Verilog level, where ChiselTrace cannot perform analysis.

From this list, it can be seen that certain FIRRTL features are missing, such as `enum`s, `match` statements, probe types, property types, and layers. The first two are omitted because Chisel constructs

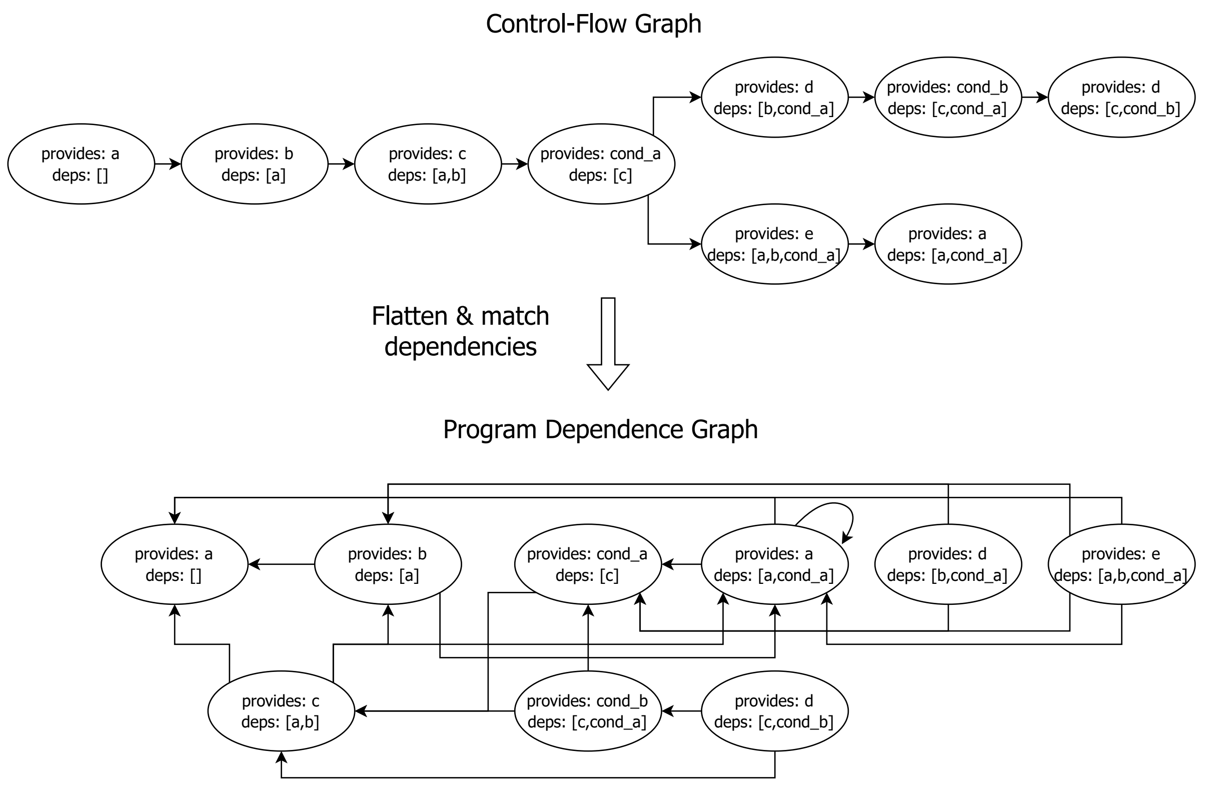

Control-Flow Graph
provides: a deps: []
provides: b deps: [a]
provides: c deps: [a,b]
provides: cond_a deps: [c]
provides: d deps: [b,cond_a]
provides: cond_b deps: [c,cond_a]
provides: d deps: [c,cond_b]
provides: e deps: [a,b,cond_a]
provides: a deps: [a,cond_a]
Flatten & match dependencies
Program Dependence Graph
provides: a deps: []
provides: b deps: [a]
provides: cond_a deps: [c]
provides: a deps: [a,cond_a]
provides: d deps: [b,cond_a]
provides: e deps: [a,b,cond_a]
provides: c deps: [a,b]
provides: cond_b deps: [c,cond_a]
provides: d deps: [c,cond_b]

Figure 3.8: PDG generation process. The generated CFG contains the nodes for the PDG. The nodes are first flattened, then, based on each node's dependencies and provided symbols, edges are formed between the statements, resulting in a PDG. Edges point in the direction of the dependencies. In this case, `a` is a registered signal.

such as the `ChiselEnum` do not map to FIRRTL enums, but to `UInt`s instead. Probe types are not supported because they are meant for verification and are therefore not part of the circuit. Similarly, property types do not affect the functionality of the circuit and are therefore not supported. Layers, which correspond with Chisel layers, are not supported because the functionality inside of them may not affect the rest of the circuit [49], making this a less interesting target for ChiselTrace.

3.2.4. PDG edge generation

After the PDG node generation, a CFG is available that contains PDG nodes. The first step in generating the PDG edges is to flatten the CFG structure to produce a list of nodes.

In the previous section, it was stated that memory ports may have an inferred direction. Now that all dependencies are known, it is possible to perform this inference. For each statement that was given a `write` inferred direction, there must be a statement in the PDG that connects the memory port as a LHS expression. If not, the statement is removed. For reading, we look for a statement that depends on the memory port. If there is none, it is removed. By eliminating unconnected statements, the type of the memory port is effectively inferred. This results in a new list of PDG statements.

Using the list of PDG statements, ID-based dependency matching is used to generate edges between them. This is shown in Figure 3.8. First, a list is made of all the generated PDG nodes. Then, a map is created that contains keys of all dependencies that are provided by PDG nodes (these are encoded in the `PDGDependency` objects associated with nodes). These keys map to lists of PDG nodes that provide the dependency indicated by the key. By constructing this map, finding all providing statements for a statement with dependencies can be performed with constant time complexity.

All the PDG nodes are iterated over. For every node, the dependencies are used to index the statement map. Based on the statement being processed, edges are created between it and the defining statements of its dependencies. Furthermore, if the dependency is a `ConditionalDependency`, the probe signal names and their required values are embedded in the edge. This is not shown in the figure.

The generated data is of the format shown in Table 3.1. The data for each vertex is the node data that is collected during the PDG node generation stage. The edge data is generated during the edge generation stage. The exported CFG is the same as the output of the node generation stage.

Field	Description
`vertices`	A list of PDG nodes. The following information is stored per node: • **file**, **line**, **char**: FIRRTL source locator information. • **name**: Name given to the node by the node generation stage based on the statement type. • **kind**: Node kind. One of: `Definition`; `DataDefinition` (for FIRRTL `node` statements); `IO`; `Connection`; `ControlFlow`. • **clocked**: Whether the node is a register. • **relatedSignal** (optional): Contains sub-fields `signalPath` (hierarchical path to root signal) and `fieldPath` (local path within compound signal). • **assignsTo** (optional): Full path that the node assigns a value to. • **isChiselStatement**: Indicates if node directly corresponds to a Chisel statement. • **condition** (optional): List of `probeName` and `probeValue` pairs imposing conditions on whether the node is valid. • **assignDelay**: Used when value assignment should be delayed (for memories).
`edges`	A list of PDG edges. The following information is stored per edge: • **from**: Index of dependency-having node. • **to**: Index of dependency-providing node. • **kind**: Edge kind. One of: `Data`; `Conditional`; `Declaration`; `Index`. • **clocked**: Whether the connection is a clocked dependency. • **condition** (optional): List of `probeName` and `probeValue` pairs imposing conditions on whether the edge is valid.
`predicates`	A list of special nodes that correspond with predicate probe signals.
`cfg`	A recursive data structure that contains the CFG. Each node can have the following data: • **stmtRef**: Index of the corresponding node. • **predStmtRef** (optional): Index of the related predicate signal node. • **trueBranch** (optional): List of CFG nodes. • **falseBranch** (optional): List of CFG nodes.

Table 3.1: Data generated by the `BuildProgramDependencyGraph` Chisel phase.

3.2.5. Data exporting

The generated PDG and CFG are based on references. To export these graphs, they are converted to a representation where all nodes are contained in a list and the references are replaced with indices into this list. The graphs are exported in a JSON format. This is integrated into the Chisel library by adding a custom annotation that implements `CustomFileEmission`. This will export the JSON file as a compilation artifact, which can be ingested by downstream applications. The exported data follows the structure as displayed in Table 3.1.

4 Back-end

The information that is produced during compilation and simulation must be processed to provide the required functionality of (dynamic) program slicing and DPDG generation. The functionality is implemented in a library that is separate from the compilation / simulation chain and any user interface. Therefore, it serves as a bridge between the raw data and front-ends. This chapter presents the design approach and implementation decisions that were made to create the `chiseltrace-rs` library.

4.1. Building a slicing library

ChiselTrace is capable of generating (dynamic) program slices and DPDGs for a Chisel circuit. After the compilation and simulation steps, the following information is available: VCD, HGLDD, and PDG/CFG files. A library, `chiseltrace-rs`, is created for synthesizing this information into the aforementioned formats.

`chiseltrace-rs` is designed as a library to serve as a bridge between the information generation stages and applications (such as user interfaces). This makes it easier to integrate into tools, as is shown in Chapter 5, where a graphical user interface (GUI) is presented. Since simulation data for real-world designs are often many gigabytes in size, a performant language is required. Rust is the language of choice for ChiselTrace, due to its performance and compatibility with the `tywaves-rs` library

Figure 4.1 shows how `chiseltrace-rs` fits into the toolchain and shows the different parts of the library. `chiseltrace-rs` consists of several parts. Namely, it contains methods for static slicing of PDGs, DPDG construction, mapping FIRRTL graphs back to a Chisel representation, dynamic slicing, and simulation data injection. The following sections will describe the implementation and the design choices behind the implementation of these parts.

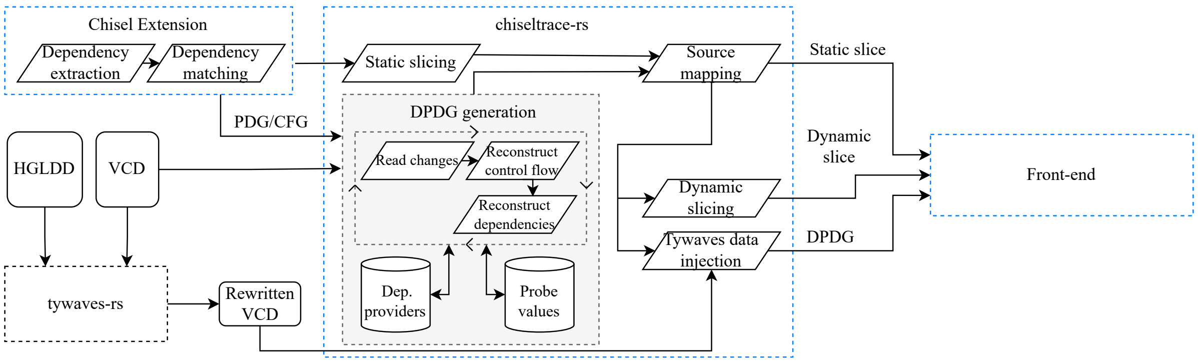


Figure 4.1: An overview of the components involved in `chiseltrace-rs`. It can be seen that `chiseltrace-rs` serves as as the interface between data-generation stages and front-ends. Novel contributions are marked in blue.

4.2. Static slicing

Before any program slicing can be performed, the PDG that is exported from the Chisel phase described in Table 3.1 has to be be ingested. To ensure ease of use and type safety, the `serde-json` crate is used for this purpose. `chiseltrace-rs` defines a schema in the form of Rust structs. By deriving a `Deserialize` trait on the structs belonging to the PDG specification, `serde-json`[50] is able to decode the incoming JSON files into the correct format. The rest of the library is then able to process the data in the PDG, just like any other struct. This has the further benefit that potentially malformed incoming JSON data is automatically validated and subsequently rejected.

Static slicing of the FIRRTL PDG is performed according to the algorithm presented in Section 2.3. First, the criterion is found in the PDG. Afterwards, the graph is traversed, marking all visited edges as present in the static slice. To facilitate the graph traversal, the PDG is converted into a linked format. The incoming PDG is provided in a two-list format: nodes and edges. The edges contain a “to” and “from” field that contain indices into the nodes array. All nodes are converted to reference-counted pointers. Then, using the list of edges, each node is given a list of pointers to its dependency nodes.

After the criterion is found, the graph is traversed from that node onward. Every visited node is part of the static slice. Using this approach, a reduced version of the PDG is produced, which can be exported in JSON using `serde-json`.

Note that, while slicing produces the lines of the included statements, the produced slice is not a functional slice (a slice that can be executed). For example, for some multi-line statements, only the start line is included in the slice. Furthermore, module definitions and syntactical lines (such as curly braces) are missing from the slice. To make the slice functional, additional processing is required. Since this is not the focus of this thesis, it is omitted.

4.3. DPDG generation and dynamic slicing

DPDG generation is more complicated, since it involves using execution history to build a graph of statement occurrences and their inter-cycle dependencies. To reconstruct execution history, simulation data is required. This is provided by Verilator in the form of a VCD file. To read this information, `chiseltrace-rs` makes use of the `vcd`[51] crate. This library enables parsing of VCD headers and reading signal changes from the VCD, one-by-one, using a buffered file reader, allowing ChiselTrace to process simulation data without loading the entirety of a possibly large VCD file into memory.

Two structs are defined that are responsible for the graph building process: the `GraphBuilder` and the `VcdReader`. An instance of the latter is a member of the former. The `VcdReader` is discussed first.

In order to make execution history reconstruction possible, a number of assumptions are made about the Chisel circuit. First of all, since Chisel circuits are often synchronous, the circuit is assumed to be sequential and of a synchronous nature. Furthermore, the assumption is made that the circuit only contains a single global clock and reset signal. The latter assumption rules out circuits with multiple clock domains. Based on these assumptions, it is possible to track statement dependencies by reconstructing control flow at each clock cycle. Here, the assumption is made that clocked elements update their value on the rising edge of the clock, after which wire signals are updated.

With this in mind, the `VcdReader` must be able to process the VCD file in chunks of one clock cycle, accumulating the probe signal changes between rising clock edges. An issue can occur here. In the VCD file, all delta cycles of the simulator have been resolved. Furthermore, the order in which the changes occur within the same timestamp is not necessarily the actual order in which the signals changed. For this reason, all signal changes that happen at the same time as a rising clock edge are buffered and counted as changes for the next cycle. The probe signal values are stored in a hash map, allowing the values to be read based on the probe signal name. This takes into account that multiple probes may share the same ID code in a VCD file.

To process the VCD into a DPDG, the `GraphBuilder` is instantiated. This, in turn, creates a `VcdReader`. Before proceeding to the DPDG building, the PDG is converted to a similar linked memory format as for static slicing. Furthermore, two essential data structures are set up: the dependency state and predicate values maps (see the part of Figure 4.1 marked in grey). The predicate value map stores the state of all predicate probe signals in the circuit. The dependency state map tracks based on the hierarchical symbol ID, which node (statement) of the PDG last assigned a value to that particular symbol. This makes inter-cycle dependency tracking possible.

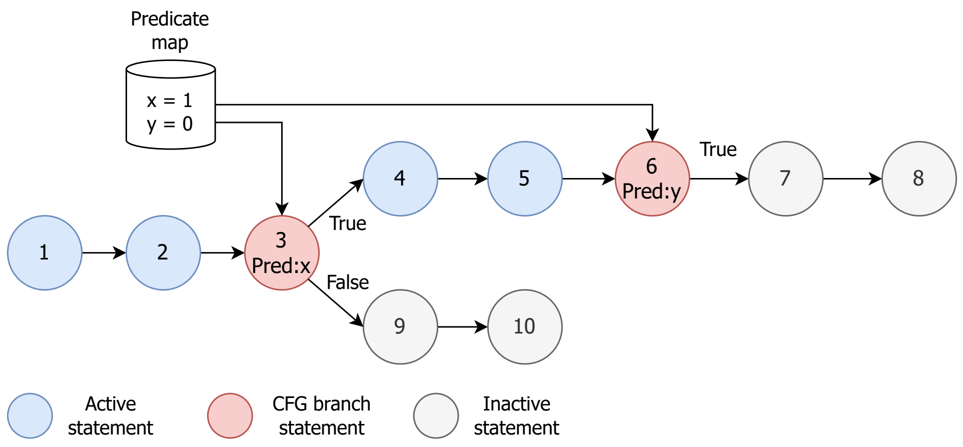


Figure 4.2: FIRRTL coverage calculation on an example CFG. In this example, the set of activated statements is $A = \{1, 2, 3, 4, 5, 6\}$.

After initialisation, the DPDG is built up from the start of the simulation data (i.e., the VCD file is processed in chronological order) until a maximum amount of clock cycles or the end of the data is reached. While this approach simplifies implementation, it may not scale well to larger simulation files. In the context of this thesis, this is deemed acceptable, as scaling is not the primary focus.

For each cycle, the value changes are requested from the `VcdReader`. These are then used to update the predicate probe values map. Using this map and the imported CFG, coverage is calculated on a FIRRTL statement level (i.e., a list of indices of the activated statements in the current clock cycle is produced). Figure 4.2 shows this process. As seen in Chapter 3, the CFG has the form of a list of nodes. Each node in the CFG contains an index to a corresponding PDG node. Furthermore, branch nodes contain two lists of statements for the true and false branches and an index to the related predicate probe. The CFG is traversed, marking each PDG node that is encountered as activated. When a branch is reached, the name of the predicate probe signal corresponding to the branch node is used to query the aforementioned predicate probe map for the related signal value. This value is then used to determine which branch to proceed traversing. The end result is a list of activated statements (execution history, or coverage) for each clock cycle. In Section 3.2.3, it was stated that some statements may have an “assign delay” (i.e., the result of the statement is delayed by some number of clock cycles). The list of activated statements is combined with a list of scheduled statements. New statements with a delay are scheduled to be handled at a later time step. Furthermore, a snapshot of the dependency map is taken to correctly assign dependencies to delayed statements.

New DPDG nodes are created in two steps: updating the dependency map and node creation. This is shown in Algorithm 1. First, the dependency map is updated. Each node in the PDG may have a field populated that indicates to which symbol it assigns a value. Updating the dependency map can now be achieved as follows. Each activated statement is iterated over. For each statement, a DPDG node is created with the current timestep, a reference to the underlying PDG node, and an empty dependency list. If the PDG node assigns to a symbol, the value at that key in the dependency map is overwritten with a reference to the newly created DPDG node. This can be done because the activated statements are generated from the CFG, which has the same statement order as the original FIRRTL circuit. This means that the FIRRTL last-connect semantics are being followed.

It is at this point that probe conditions are imposed on PDG nodes. If the PDG node has conditions on probe signals, they are checked, and only when all conditions are met will the new DPDG node be added to the dependency map. This has the effect of discarding all dynamically indexed assign statements that do not correspond with the correct index. Lastly, wire signals are immediately updated in the dependency map. However, registers only update their value at the next rising edge. Therefore, these updates are delayed by one cycle.

Using the updated dependency map, all activated new DPDG nodes are processed to fill their dependency lists. Symbolic dependencies are gathered by finding the dependencies of the corresponding PDG node. These are then looked up in the dependency map. At this point, the probe conditions are imposed on the edges of the PDG in a similar way as before. This filters out statement dependencies

Algorithm 1 DPDG construction

```
function CheckConditions($C$)                                    // Check probe conditions
    for each $c \in C$ do
        if $probeValues[c.probe] \neq c.value$ then
            return false
        end if
    end for
    return true
end function

function UpdateDPDG($S$)                                         // Update dynamic PDG
    $A \leftarrow$ getActivatedStatements()
    $time \leftarrow$ current cycle $- 1$
    $controlFlowProviders \leftarrow \emptyset$
    $newNodes \leftarrow \emptyset$
    $regBuffer \leftarrow \emptyset$                             // Buffer for register assignments

    for each $stmt \in A$ do                                     // Creation of new DPDG nodes
        $n \leftarrow$ PDGNode for $stmt$
        $dnode \leftarrow$ new DPDGNode($n, time$)
        $newNodes \leftarrow newNodes \cup \{(n, dnode)\}$
        if CheckConditions($n.conditions$) then
            if $n$ assigns to variable $v$ then
                if $n$ is clocked then
                    if reset condition then
                        $S[v] \leftarrow dnode$          // Immediately update the state for reset registers
                    else
                        $regBuffer[v] \leftarrow dnode$                  // Delayed assignment
                    end if
                else
                    $S[v] \leftarrow dnode$                              // Immediate state update
                end if
            end if
            if n.kind = Conditional then
                $controlFlowProviders[n] \leftarrow dnode$
            end if
        end if
    end for

    for each $(node, dnode) \in newNodes$ do                     // Dependency resolution
        if node has assignDelay then
            loadStateSnapshot()
        end if
        for each $(dep\_node, dep\_edge) \in$ dependencies of $node$ do
            if CheckConditions($dep.conditions$) then
                Add dependency to dnode edge based on dep_edge type
                    Data/Index: Use current state $S$
                    Conditional: Use controlFlowProviders
                    Declaration: Make new DPDGNode from node      // Only for dynamic slicing
            end if
        end for
    end for

    Update criterion
    Commit $reg\_buffer$ to $S$                                  // Register state update
end function
```

for indices other than the current one.

Lastly, we keep track of the criterion node. After all new DPDG nodes for a cycle are created, the criterion node is updated. Once all the simulation data is processed, the criterion is returned. If the criterion is a statement, it is the latest occurrence of that statement in the DPDG. If the criterion is a signal, it is taken from the dependency map.

Calculating a dynamic program slice from the DPDG is now done by traversing the DPDG from the criterion onward. Each unique underlying PDG node is included in the dynamic slice. For dynamic slicing, defining statements (such as `wire`, `reg`) are included. Together with the source mappings, all relevant lines in the circuit source code can be determined. Note that this dynamic slice has similar limitations to the static slice.

4.4. Source mapping

All analysis until this point has been done at the FIRRTL level. To fulfil the promise of source-level debugging, the generated DPDG must be converted to a Chisel representation. In Section 2.1.2, we saw that each FIRRTL statement has source mappings in the form of a file, line, and character mapping to the Chisel circuit.

Because the source mapping relies on source-locators, the reconstruction can only be as accurate as the available information in the FIRRTL statements allows it to be. Most FIRRTL statements contain accurate source locators; however, in certain cases, such as `I/O` definitions, all nodes in the PDG point only to one defining line of the Chisel circuit. In this case, that means that a DPDG node of some member `x` of the `I/O` of a module points at the defining line of the higher-level `IO` definition.

Another example would be certain high-level Chisel constructs (e.g., decoders). These produce statements that map to lines of the standard library. This section presents the grouping method and simple graph heuristics that ChiselTrace uses to reconstruct a reasonable reconstruction of the source-level language. The two examples given above would require more advanced heuristics to properly resolve. These are not implemented.

Algorithm 2 DPDG source mapping

```
function dpdgConvertToSource(pdg)
    $groups$ ←groupNodes($pdg.nodes$)                          // Simple HashMap based grouping
    $indexFiltered$ ←filterIndexProbes($pdg.edges$)             // Redirect probe edges
    $groups$ ←splitGroups($groups, indexFiltered$)              // Vertex reachability test
    $mappedEdges$ ←mapEdges($indexFiltered, groups$)            // Map edges to new group indices
    $newNodes$ ←convertGroupsToNodes($groups$)                  // Convert each group to a new node
    $newPDG$ ←deduplicateNodes($newNodes, mappedEdges$)
    return $newPDG$
end function
```

The entire conversion process is shown in Algorithm 2. Furthermore, Figure 4.3 shows a graphical representation of the grouping process. Here, the coloured nodes are nodes with the same source locator information, except for the purple node, which is an indexing probe. It can be seen that connected nodes are based on the file, line number, and timestamp. The directedness of the graph can be ignored for the grouping process. The initial grouping is implemented using a HashMap. At this point, no attention is paid to the graph nature of the data. Because of this, a situation can occur where different symbols are defined/assigned at the same source line (e.g., compound signals) and are therefore erroneously grouped. This problem is solved by checking vertex reachability within a group for each node using depth-first search (DFS, see Algorithm 3).

If the circuit contained any dynamic vector indexing or multiplexers, probes would have been inserted. Since these are not present in the original circuit, they are filtered out. Intra-group edges are also filtered out. Furthermore, edges are de-duplicated because multiple nodes in a group may depend on the same group.

Next, the groups and edges are converted into a (D)PDG. If there is a node in the group that is marked as corresponding with a Chisel statement, this node is chosen as the representative node for the group, making the group-representing node carry this node's kind and name. If such a node is absent, the node name is derived from the FIRRTL source locator information and the node kind is

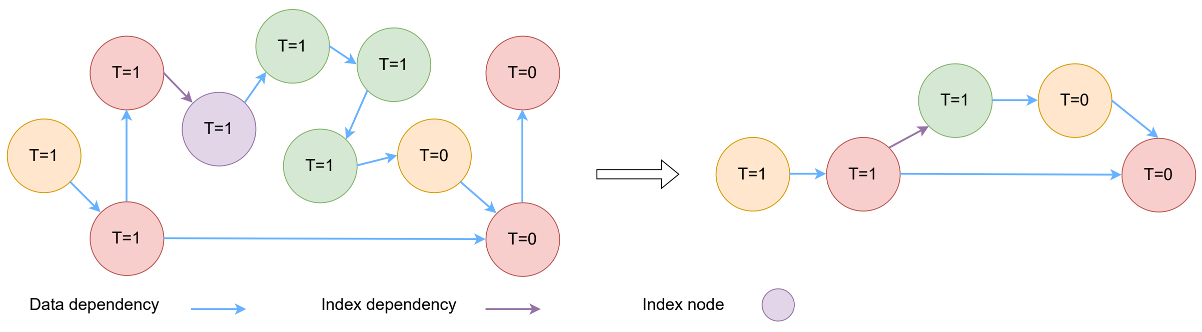


Figure 4.3: Conversion process of the DPDG from FIRRTL to Chisel representation. Different coloured nodes represent statements with the same FIRRTL source locator information. Purple nodes and edges represent index probes and dependencies, respectively.

determined by the node kinds of the individual nodes in the group. The order of importance is shown in Table 4.1.

Algorithm 3 Group splitting algorithm

```
function splitGroups(groups, edges)
    $(edgesByFrom, edgesByTo) \leftarrow createEdgeMaps(edges)$        // Create lookup tables for edges
    $newGroups \leftarrow \emptyset$
    for each $group \in groups$ do
        for each $currentNode \in group$ do
            $stack \leftarrow [currentNode]$        // DFS using a stack, starting at the current node
            $currentGroup \leftarrow \emptyset$
            for each $(node, idx) \in stack$ do
                $connections \leftarrow edgesByFrom[idx] \cup edgesByTo[idx]$
                for each $edge \in connections$ do
                    // Get the nodes in the current group that are connected to the edge
                    $connectedNodes \leftarrow getConnectedNodesInGroup(edge)$
                    $group \leftarrow group \setminus connectedNodes$        // Prevent re-processing nodes
                    $stack \leftarrow stack \cup connectedNodes$
                end for
                $currentGroup \leftarrow currentGroup \cup node$
            end for
            $newGroups \leftarrow newGroups \cup currentGroup$
        end for
    end for
end function
```

Finally, a heuristic is applied to reduce the number of nodes. It is observed that certain Chisel constructs, such as lookup tables, translate to many statements that are not efficiently grouped. These nodes share the same dependencies and are marked as not corresponding directly with a Chisel statement. This makes these nodes easy to detect and remove duplicates of.

To ensure scaling to large DPDGs, all stages of the source mapping that require filtering edges are optimised using lookup tables that contain pre-computed lists of edges, indexed by the source or destination node indices.

4.5. Injecting Tywaves simulation data

ChiselTrace integrates with the `tywaves-rs` library to annotate the DPDG with typed simulation data, where available. To this end, the VCD rewriting and variable translation of `tywaves-surfer` are partly reused. The entire value-annotation process is shown in Figure 4.4. `chiseltrace-rs` first uses `tywaves-rs` to rewrite the VCD file into the format mentioned in Section 2.2. The `tywaves-rs` library can then be used to translate a bit string into the typed Chisel representation of the corres-

Kind	Rationale
Connection	When high-level Chisel functionality is used, connect statements may be generated without a direct corresponding statement in the Chisel source code. The group may provide data dependencies to other nodes if a connection is present. It would be unusual for the node to be of another type than `Connection`, therefore it is placed above `ControlFlow` in importance.
Control Flow	If a node only contains a control flow node without connections, it likely corresponds with a control flow construct in the Chisel source. Note that `ControlFlow` is given higher priority than `DataDefinition` (which correspond with FIRRTL `node`s). This is because `node`s are often used to determine the predicate for a control flow statement.
Definition & DataDefinition	Definitions only exist in regular PDGs and are therefore not of high importance. For the `DataDefinition`, see the explanation above.

Table 4.1: Order of importance for determining the node kind in absence of a primary node.

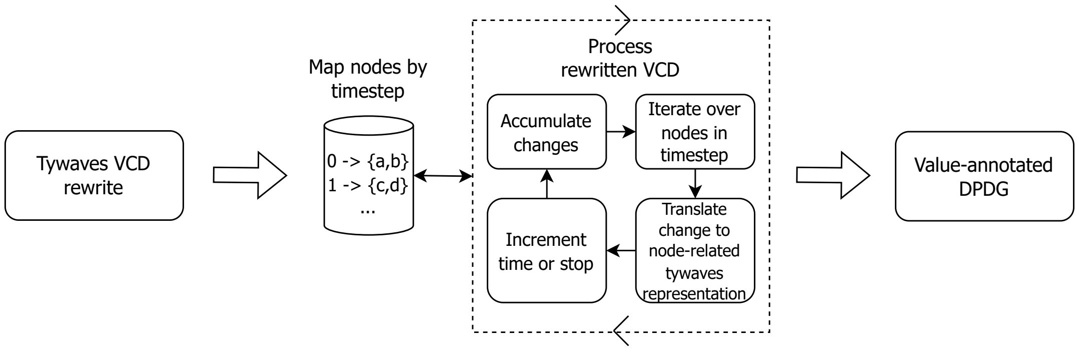


Figure 4.4: Simulation data value-annotation of the DPDG. The VCD file is rewritten by `tywaves-rs`, then processed in time steps of one clock cycle.

ponding compound signal. A minor contribution is made to the `tywaves-rs` library that speeds up this VCD rewriting phase. Instead of writing a change to the VCD file every time one of the sub-fields of a compound signal changes, all changes are merged in memory for each time step, reducing I/O, significantly decreasing file size, and improving subsequent VCD processing speed. Furthermore, a lookup table is computed that maps VCD IDs to relevant tywaves variables. This prevents the need to perform a linear search over all tywaves variables for every signal change.

Signal changes are accumulated per clock cycle while reading the rewritten VCD file. Every clock cycle, all DPDG nodes with the current timestamp are iterated over. If the node assigns to a symbol, the full hierarchical path of the symbol (including the local path within the compound type) is used to look up the symbol in the Tywaves data. If simulation data is found, it is associated with the DPDG node. After all the simulation data is processed, every node that corresponds with a symbol that exists in the simulation data has a typed value annotation.

5 Front-end & ChiselSim integration

ChiselTrace is presented as a debugging tool for Chisel circuit designs. The aim is to provide better source-level debugging capabilities, allowing designers to diagnose problems directly at the Chisel level. For this reason, user interaction is an important part of the project. So far, ChiselTrace can produce a DPDG for a circuit, given a criterion. This alone is not useful; it needs to be visualised in some way for the user. Further integration into the Chisel simulation pipeline would also make ChiselTrace much more useful in the real world.

The modular nature of `chiseltrace-rs` allows any front-end user interface to hook into the library. In this chapter, a reference implementation of a GUI is presented for ChiselTrace. Furthermore, modifications to ChiselSim are presented that enhance the ChiselTrace user experience for test-driven design approaches.

5.1. ChiselTrace front-end

The output produced by `chiseltrace-rs` is a DPDG, of which the nodes contain timestamps, information about source mappings, and simulation values. One observation about the DPDG is that it is a directed, acyclic graph (DAG). The directed edges indicate dependencies between statements. Furthermore, it can be observed that the general flow of the graph is in one direction: from nodes with a high timestamp value to nodes with a low value. A suitable visual representation for the DPDG is therefore a timeline visualisation, where nodes are grouped into distinct groups that correspond with their timestamp (clock cycle number). This closely mirrors the timeline representation of waveform viewers.

5.1.1. Integration vs custom tooling

Open-source tooling exists for creating graph visualisations. For example, GraphViz [52] is a widely used tool for graph visualisation. It can create visualisations from a graph represented in the `DOT` language. It also has library bindings. GraphViz, however, is meant for static rendering of graphs. This would limit user interaction and does not scale well to large DPDGs. An alternative could be Gephi [53], which is software focused on interactive graph visualisations. It has a lot of built-in graph analysis functionality that is unrelated to ChiselTrace. While Gephi does support extensions in the form of Java plugins, significant changes to the program would have to be made to create the desired timeline graph representation and DPDG node information. This means that a custom solution is created.

For the design of the ChiselTrace front-end, a web-based solution is chosen for a fast development cycle. Integration with the Rust back-end is achieved by using the Tauri [54] framework. Tauri is a GUI framework that uses the OS's native web renderer to display a web front-end that communicates over inter-process communication with a Rust back-end. This enables the use of well-established web frameworks and graph visualisation libraries, while keeping compatibility with `chiseltrace-rs`.

The Svelte [55] JavaScript framework is used in the front-end. Svelte is a compiler-based framework that converts components (written in HTML, CSS, and JavaScript) into optimised JavaScript. For ChiselTrace, the reactivity (i.e., when a change to a variable is made, the rendered `GUI` updates) simplifies dynamic graph loading and displaying node information upon user interaction. The graph

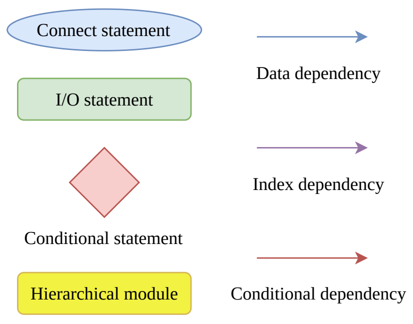


Figure 5.1: Legend for the GUI nodes and edges.

visualisation library of choice is `vis.js` [56]. This library was chosen over others, such as `D3.js` or `cytoscape.js`, due to its ease of use and inherent support for user-interaction.

5.1.2. Building a graph visualiser

The ChiselTrace front-end is shown in Figure 5.2. It consists of a scrollable collection of time slots. At the top of the interface, the time slot numbers can be seen (see area $\boxed{B}$ of the figure). These correspond with the full clock cycles (these are the time steps from Chapter 4). It can be seen that, contrary to how waveform viewers display simulation data, the scroll direction is from high to low time steps. This is done because the criterion is always in the latest shown time step.

Node placement is handled using the physics engine inside the `vis.js` library using a Barnes-Hut solver. Since the library does not natively support the type of visualisation that is required, the node positions are artificially constrained to their corresponding time slot before being drawn to the screen. After a short stabilisation period, the nodes are frozen and can be moved manually.

An issue with this approach is that at most a handful of timeslots can be displayed at a time. This is dealt with by dedicating a small part of each time slot to dependencies that would fall off-screen. The idea is that the majority of the dependencies will be at most a couple of clock cycles away from each other. This means that relatively few dependencies will have to be placed in this dedicated area. This is shown in area $\boxed{C}$.

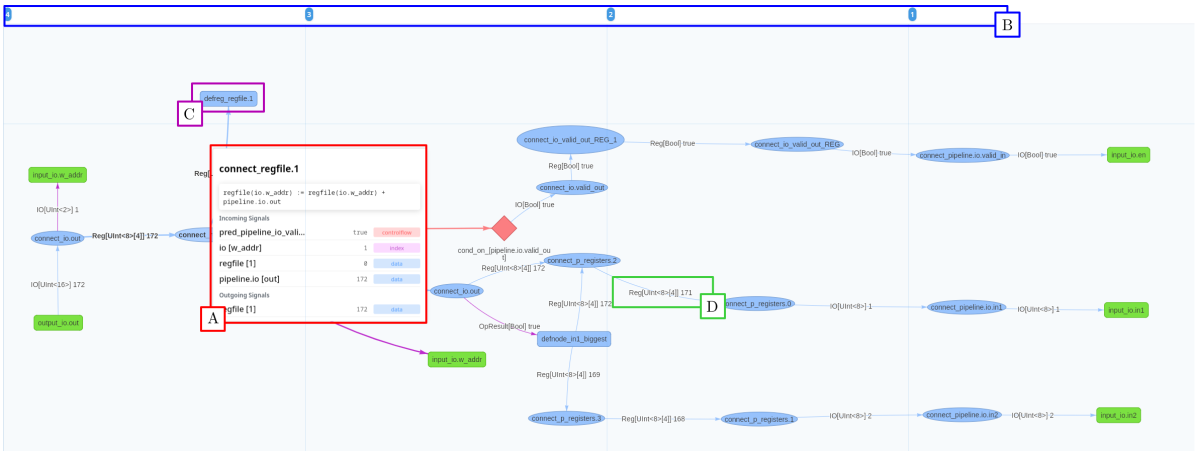
The different kinds of nodes are given distinct colours and shapes. Dependencies between nodes are shown as directed edges, with the arrow being in the direction of the dependency. A legend of the different nodes and edges is shown in Figure 5.1. Furthermore, node information, such as the source code line and all dependencies, is shown when the user hovers over the node (shown in $\boxed{A}$). Furthermore, the Tywaves simulation data is shown as annotations to the edges (see area $\boxed{D}$). Tywaves simulation data either contains an already translated value (e.g., enums are translated to their string representation by Tywaves) or a bit string with a type annotation. Data of the latter format is automatically interpreted based on the type, if it is a(n) (un)signed integer or a boolean value.

One potential issue with the approach, as it is stated in Chapter 4, is that DPDGs may grow quite large (many thousands of nodes). This is more than what the `vis.js` library is able to handle. To alleviate this problem, nodes are loaded dynamically based on the timeslots that are drawn on screen. The Rust back-end creates a mapping of which nodes belong to which time slots. The front-end then requests a partial graph based on a time slot range whenever the user scrolls the timeline. Updating the graph on screen is done by updating an array, making use of Svelte's reactive nature to update the graph on screen.

To enable a more interactive and clearer presentation of the DPDG, the front-end supports hierarchical node grouping (i.e., node clustering based on the instanced module they belong to in the circuit) and selecting a new graph head (and thereby eliminating paths).

5.2. Integrating with ChiselSim

To facilitate an easy debugging pipeline for end users, ChiselTrace is integrated with ChiselSim. As seen in Section 2.1.3, to simulate a Chisel module using ChiselSim, the user provides a unit test using



connect_regfile.1

```
regfile(io.w_addr) := regfile(io.w_addr) +
pipeline.io.out
```

Incoming Signals

pred_pipeline_io_vali...	true	controlflow
io [w_addr]	1	index
regfile [1]	0	data
pipeline.io [out]	172	data

Outgoing Signals

regfile [1]	172	data

Figure 5.2: ChiselTrace GUI of an example circuit. Different parts are highlighted to show the functionality of the interface.

a simulator object that uses the `PeekPokeAPI`. Chisel itself only includes an `EphemeralSimulator`, which does not produce any artifacts. The Tywaves project later added the `tywaves-chisel` API, which brought two new kinds of simulators: the `ParametricSimulator` and the `TywavesSimulator`, enabling simulations with artifacts (VCD and HGLDD files required for Tywaves) and a way to launch Tywaves using ChiselSim, respectively. ChiselTrace extends upon this work by introducing the `ChiselTraceDebugger` simulator.

This new simulator aims to provide an interactive debugging session within ChiselSim. This means that a simulation needs to be able to be paused during a unit test. Several modifications are made to facilitate this.

- The Tywaves-Surfer waveform viewer is launched before the simulation starts. Furthermore, modifications are made that allow previously unsupported hot-reloading of VCD files with Tywaves translation.
- Changes are made to the ChiselSim testing harness to add commands for flushing simulation data to a file.
- The simulation controller is changed such that a debug function may be registered. This allows the controller or a simulator to yield control to the debugger UI when the simulation needs to be paused. ChiselTrace implements pausing on a failed assertion.

A debug handler is registered with the simulation controller through a modified `ParametricSimulator`. In Figure 2.4, this is indicated by the communication between the `Simulator` and the `Controller`. Notably, the debug handler is attached before the commands from the unit test are run. If an assertion in the `PeekPokeAPI` fails and a debug handler is attached, the simulation data is flushed to disk and information regarding the failed assertion is passed to the debug handler. The user may then continue/stop the simulation or invoke ChiselTrace using the asserted signal as a criterion.

In Figure 5.3 and Figure 5.4, the process of running a unit test on an example `GCD` circuit is shown for the `TywavesSimulator` and `ChiselTraceDebugger`, respectively. It can be seen that the new `ChiselTraceDebugger` can be used as a drop-in replacement for the regular `TywavesSimulator`.

```scala
import tywaves.simulator._
import tywaves.simulator.ParametricSimulator._
import tywaves.simulator.simulatorSettings._
// Use the tywaves simulator
import TywavesSimulator._

describe("TywavesSimulator") {
  it("runs GCD correctly") {
    simulate(new GCD(), Seq(VcdTrace, WithTywavesWaveforms(true)), simName = "runs_GCD_correctly_chiseltrace") {
      gcd =>
        gcd.io.a.poke(24.U)
        gcd.io.b.poke(36.U)
        gcd.io.loadValues.poke(1.B)
        gcd.clock.step()
        gcd.io.loadValues.poke(0.B)
        gcd.clock.stepUntil(sentinelPort = gcd.io.resultIsValid, sentinelValue = 1, maxCycles = 10)
        // Will crash and fail the unit test when assertion fails
        gcd.io.resultIsValid.expect(true.B)
        gcd.io.result.expect(12)
    }
  }
}

```

Figure 5.3: Running a simulation using the `TywavesSimulator`

```scala
import tywaves.simulator._
import tywaves.simulator.ParametricSimulator._
import tywaves.simulator.simulatorSettings._
// Now, the debugger is imported instead of just the simulator
import ChiselTraceDebugger._

describe("ChiselTraceDebuggerr") {
  it("runs GCD correctly") {
    simulate(new GCD(), Seq(VcdTrace, WithTywavesWaveforms(true)), simName = "runs_GCD_correctly_tywaves") {
      gcd =>
        gcd.io.a.poke(24.U)
        gcd.io.b.poke(36.U)
        gcd.io.loadValues.poke(1.B)
        gcd.clock.step()
        gcd.io.loadValues.poke(0.B)
        gcd.clock.stepUntil(sentinelPort = gcd.io.resultIsValid, sentinelValue = 1, maxCycles = 10)
        // When either of the assertions fail, the simulation will pause and the user will receive a prompt to invoke ChiselTrace on the fault signal.
        gcd.io.resultIsValid.expect(true.B)
        gcd.io.result.expect(12)
    }
  }
}

```

Figure 5.4: Using ChiselTrace through the `ChiselTraceDebugger` interface is a drop-in replacement.

6 Results & discussion

In this chapter, the results for ChiselTrace are presented. Because of the nature of this thesis, ChiselTrace is mainly qualitatively evaluated. First, all basic functionality of ChiselTrace is verified using basic testing circuits that include various constructs built into Chisel. A comparison with the more traditional Tywaves waveform debugging experience, and several debugging scenarios are presented. Afterwards, ChiselTrace will be used on a real-world design, ChiselWatt, to debug an injected fault. Then, the performance and scaling characteristics of ChiselTrace are investigated. Lastly, limitations of the current implementation are discussed.

6.1. Evaluation of capabilities

In this section, the functionality of ChiselTrace, as presented in Chapters 3 to 5 is verified using a series of isolated examples and a simple example circuit. Attention is paid to the major features of ChiselTrace. This includes:

- Correct dependency finding of FIRRTL statements, including compound data type signals and cross-hierarchical connections.
- Correct dependency tracking through time during the DPDG building stage (both registers and memories).
- Reconstruction of activated statements per clock cycle.
- Dynamic vector indexing and MUX path reconstruction.
- Reconstruction of the source-level Chisel representation.
- Generation of static and dynamic slices of the Chisel design.

Furthermore, the ChiselTrace view of the example circuit is compared to the more traditional Tywaves waveform of the same circuit. Lastly, an example is given on how ChiselTrace can be used to quickly find bugs in the design.

6.1.1. Testing features

First, all capabilities of ChiselTrace are evaluated. This section will go over a series of simple circuits that each display one or more of the capabilities mentioned above.

Tracking compound signal assignments across clock cycles

Firstly, the capability of ChiselTrace to track connections of compound data type signals through time is shown. The circuit that is used for this is shown in Listing 6.1. This circuit emulates a (small part of a) data structure that may be used for communication purposes. The input signal has a nested compound data structure. Furthermore, the input signal is delayed by 3 clock cycles by registers. Finally, all parts of the signal are connected to one output signal. If this signal is chosen as the criterion for ChiselTrace,

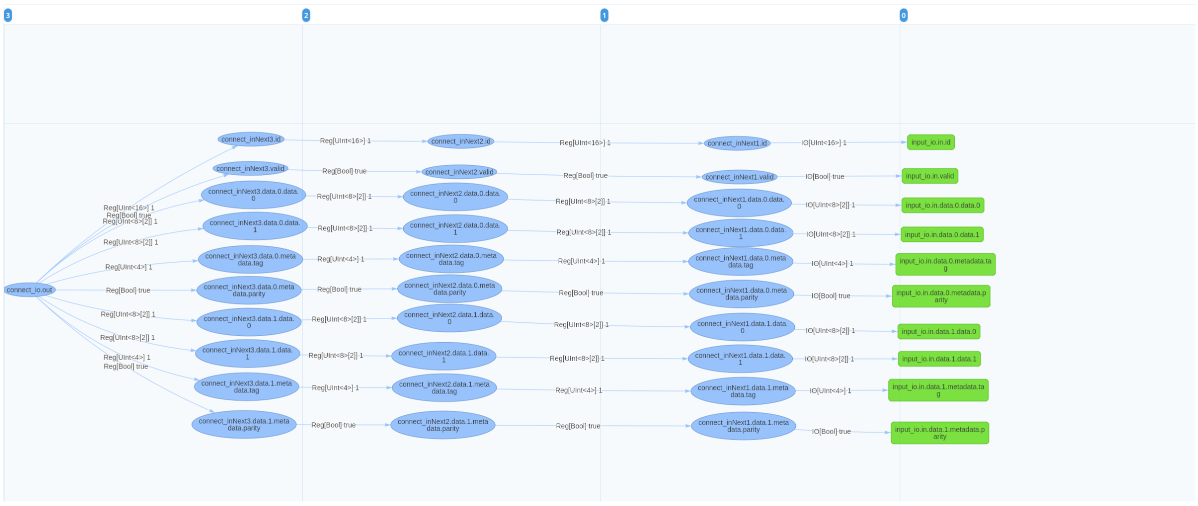

3
2
1
0
connect_io.out
connect_inNext3.id
connect_inNext3.valid
connect_inNext3.data.0.data.0
connect_inNext3.data.0.data.1
connect_inNext3.data.0.metadata.tag
connect_inNext3.data.0.metadata.parity
connect_inNext3.data.1.data.0
connect_inNext3.data.1.data.1
connect_inNext3.data.1.metadata.tag
connect_inNext3.data.1.metadata.parity
connect_inNext2.id
connect_inNext2.valid
connect_inNext2.data.0.data.0
connect_inNext2.data.0.data.1
connect_inNext2.data.0.metadata.tag
connect_inNext2.data.0.metadata.parity
connect_inNext2.data.1.data.0
connect_inNext2.data.1.data.1
connect_inNext2.data.1.metadata.tag
connect_inNext2.data.1.metadata.parity
connect_inNext1.id
connect_inNext1.valid
connect_inNext1.data.0.data.0
connect_inNext1.data.0.data.1
connect_inNext1.data.0.metadata.tag
connect_inNext1.data.0.metadata.parity
connect_inNext1.data.1.data.0
connect_inNext1.data.1.data.1
connect_inNext1.data.1.metadata.tag
connect_inNext1.data.1.metadata.parity
input_io.in.id
input_io.in.valid
input_io.in.data.0.data.0
input_io.in.data.0.data.1
input_io.in.data.0.metadata.tag
input_io.in.data.0.metadata.parity
input_io.in.data.1.data.0
input_io.in.data.1.data.1
input_io.in.data.1.metadata.tag
input_io.in.data.1.metadata.parity
Reg[UInt<16>] 1
Reg[Bool] true
Reg[UInt<8>[2]] 1
Reg[UInt<4>] 1
IO[UInt<16>] 1
IO[Bool] true
IO[UInt<8>[2]] 1
IO[UInt<4>] 1

Figure 6.1: ChiselTrace view of the circuit presented in Listing 6.1. All sub-signal inputs are set to 1. In this DPDG, it is visible that ChiselTrace tracks the individual sub-signals of compound signals through time.

the expected behaviour is that there will be connected nodes at each time step for each individual sub-signal. Finally, after three clock cycles, the signals should converge at the node for the output signal. The resulting DPDG for this circuit is shown in Figure 6.1. In the figure, it is visible that ChiselTrace is capable of tracking sub-signals of nested compound signals through time.

```scala
// A simplistic, nested compound datatype
class DataPair extends Bundle {
  val data = Vec(2, UInt(8.W))
  val metadata = new Bundle {
    val parity = Bool()
    val tag = UInt(4.W)
  }
}

class ExampleData extends Bundle {
  val data = Vec(2, new DataPair())
  val valid = Bool()
  val id = UInt(16.W)
}

// This is a simple circuit that allows for a visualisation of complex compound datatype
    signal tracking through time.
class ConnectionExampleTop extends Module {
  val io = IO(new Bundle {
    val in = Input(new ExampleData())
    val out = Output(UInt(32.W))
  })

  val inNext1 = RegNext(io.in)
  val inNext2 = RegNext(inNext1)
  val inNext3 = RegNext(inNext2)

  io.out := inNext3.data(0).data(0) + inNext3.data(0).data(1) + inNext3.data(0).metadata.
    parity.asUInt + inNext3.data(0).metadata.tag + inNext3.data(1).data(0) + inNext3.data(1).
    data(1) + inNext3.data(1).metadata.parity.asUInt + inNext3.data(1).metadata.tag + inNext3
    .valid.asUInt + inNext3.id
}
```

Listing 6.1: A simple circuit that delays a compound input signal by three cycles and connects all sub-fields to the output.

Dynamic reconstruction of control flow and data flow.
The following example will demonstrate the capability to reconstruct control flow and dynamic data flow. The circuit shown in Listing 6.2 shows multiple branches, dynamic data dependencies (reading/writing

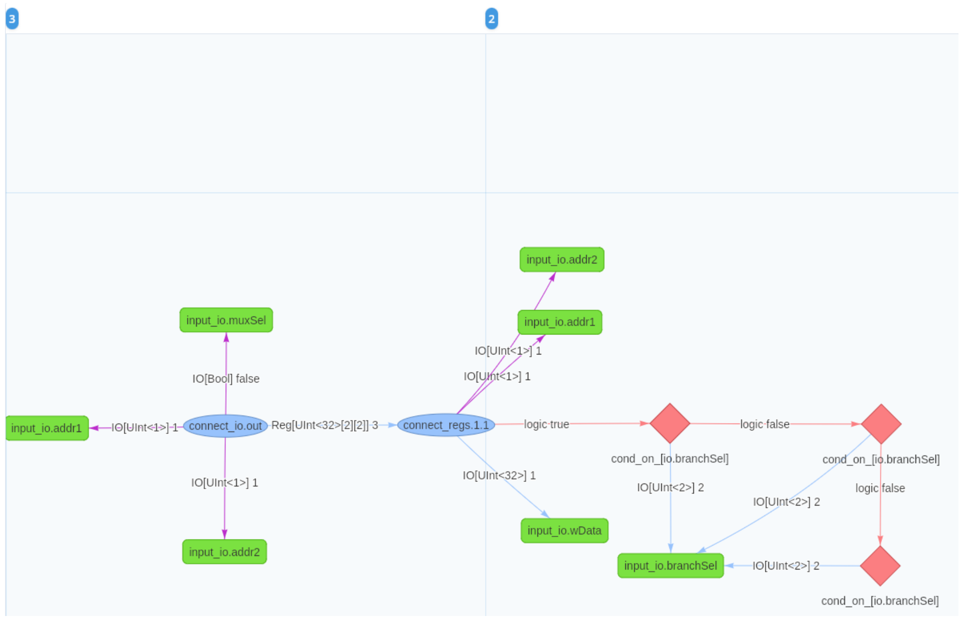


Figure 6.2: The ChiselTrace view of the circuit presented in Listing 6.2. The output DPDG matches expectations.

to a nested `Vector` type register and a MUX). Furthermore, slices generated from the (D)PDG are shown. The following inputs are chosen:

- `branchSel = 2`
- `muxSel = false`
- `addr1 = 1`
- `addr2 = 1`
- `wData = 1`

The expected behaviour is that a single connect node is generated for register 1.1. This node should have index dependencies to both address lines, a data dependency to `wData`, and a control flow dependency to the third conditional statement. Furthermore, the node that is generated for the output connection should have index dependencies on the address lines and the `muxSel` signal. It should have a data dependency on the register that was previously written to. Figure 6.2 shows the ChiselTrace reconstruction. Here, it can be seen that this view matches expectations.

In the listing, the observation is made that the static slice (shown in red + green) contains all statements that could influence the output, while the dynamic slice (red) only contains the statements that are present in the DPDG. However, it must be noted that the slices produced by the ChiselTrace CLI are not executable slices (i.e., not all lines that are required for a functional circuit are included). This limitation will be further explored in Section 6.4.

```
// This is a simple circuit that allows for a visualisation of control flow reconstruction
// and dynamic data flow reconstruction
class ControlFlowExampleTop extends Module {
  val io = IO(new Bundle {
    val addr1 = Input(UInt(1.W))
    val addr2 = Input(UInt(1.W))
    val muxSel = Input(Bool())
    val branchSel = Input(UInt(2.W))
    val wData = Input(UInt(32.W))
    val out = Output(UInt(32.W))
  })

  val regs = Reg(Vec(2, Vec(2, UInt(32.W))))
  val altOutput = WireDefault(0.U)

  when (io.branchSel === 0.U) {
  regs(io.addr1)(io.addr2) := io.wData
  } .elsewhen (io.branchSel === 1.U) {
  regs(io.addr1)(io.addr2) := io.wData + 1.U
  } .elsewhen (io.branchSel === 2.U) {
  regs(io.addr1)(io.addr2) := io.wData + 2.U
  } .otherwise {
  regs(io.addr1)(io.addr2) := io.wData + 3.U
  }

  io.out := Mux(io.muxSel, altOutput, regs(io.addr1)(io.addr2))
}
```

Listing 6.2: An example circuit containing multiple branches and dynamic data flow. Static and dynamic slicing have been applied to the circuit. Red lines are present in both slices while green lines are only in the static slice.

6.1.2. Putting it together: an example circuit

The main objective of this section is to provide a simple circuit that allows us to demonstrate most of the aforementioned functionality in a real circuit. Along with this, the ChiselTrace view is compared with the more traditional Tywaves view. Furthermore, examples are shown of how ChiselTrace can be used to enhance the debugging experience.

The example circuit is presented in Listing 6.3. It consists of a simple 8-bit ALU and a control circuit. The ALU implements addition, subtraction, OR, XOR, AND, left shift, and right shift operations. These operations are encoded as values of a `ChiselEnum`. Furthermore, the I/O of the ALU module consists of nested compound data types.

The ALU takes two 8-bit unsigned integers as input, along with a carry signal and an operation to be performed on the data. Using this, it produces one 8-bit output value and zero, carry, and sign flags. The main logic of the ALU is implemented using a high-level Chisel construct, namely the `MuxCase`. This is a construct that translates to a 2-input MUX for every case in FIRRTL.

The controller circuit is a state machine that calculates the 16-bit sum of two 16-bit inputs. It first loads in two values from the testbench into an 8-bit register file in little-endian format. It then computes partial sums in subsequent states. Finally, the result is presented at the output.

For the first example, the input values for the circuit are `0xFFFF` and `0x0001`. The criterion for ChiselTrace is chosen to be `io.out.data` of the `Controller` module. The resulting DPDG can be seen in Figure 6.3. In this figure, the previously demonstrated functionality is shown in the context of the example ALU circuit. Furthermore, node grouping based on a module hierarchy is shown.

Figure 6.4 shows the same view without hierarchical grouping. It demonstrates how ChiselTrace treats FIRRTL memories similar to vector types (see $\boxed{A}$). $\boxed{B}$ corresponds with the switch statement of the controller state machine. $\boxed{C}$ shows dynamic MUX path reconstruction. A non-optimal source-level reconstruction can be seen in $\boxed{D}$ (statements from the standard library are shown in the DPDG). This is touched upon in Section 6.4.

Comparing ChiselTrace with Tywaves

The previous example is now compared to the Tywaves view of the same circuit, which is shown in Figure 6.5. Between Figures 6.4 and 6.5, the following comparisons and observations are be made:

- On the more traditional Tywaves waveform viewer, relevant signals must be added manually. In contrast, ChiselTrace automatically determines the relevant statements and signals. In the waveform viewer, however, more signals over more timesteps can fit in view. This suggests that a combination of both tools would provide an optimal experience. First, the general location of the to-be-inspected behaviour can be located using the waveform viewer. Afterwards, fine-grained inspection can be performed using ChiselTrace.

- Some signals, such as the register file in this example, do not show up on the waveform viewer. Such cases might also occur due to compiler optimisation. On the ChiselTrace view, however, it is shown that the statement and data-flow reconstruction is unaffected by this (see $\boxed{E}$ in

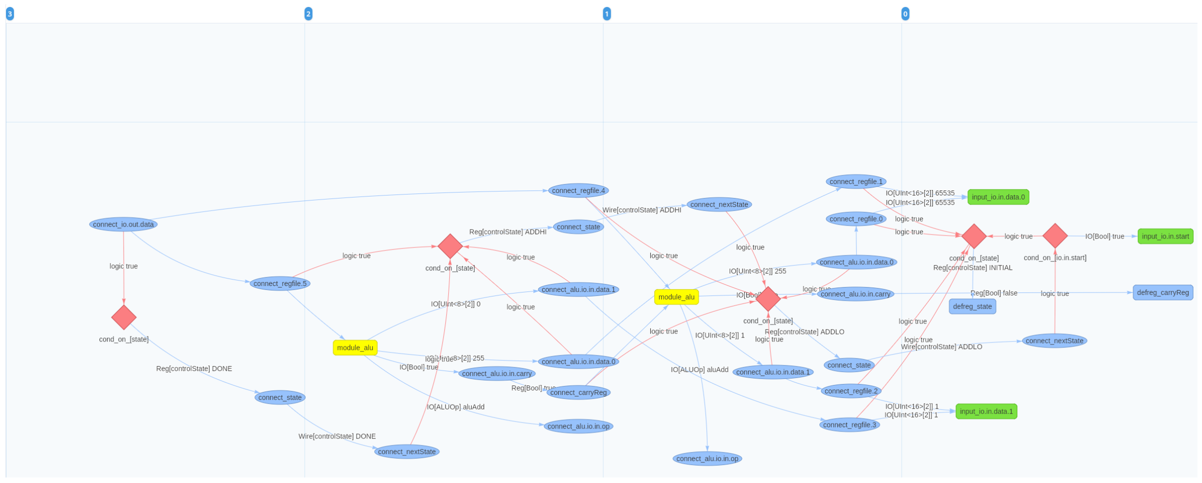


Figure 6.3: The ChiselTrace view of the ALU circuit simulation. The criterion is `io.out.data`. Hierarchical module grouping is enabled.

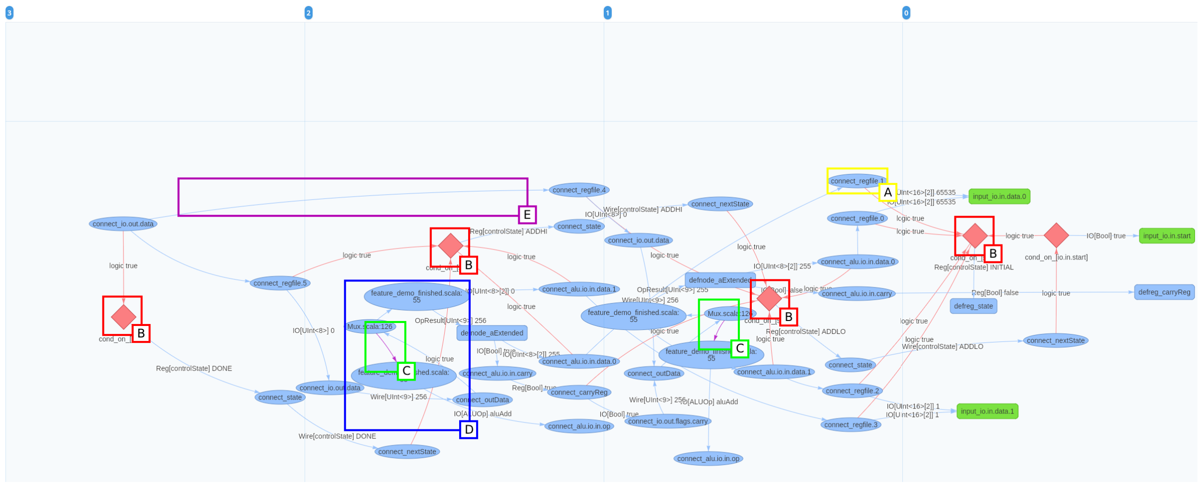


Figure 6.4: Non-grouped version of Figure 6.3. A demonstrates handling of memories. B shows a switch statement conditional. C shows dynamic MUX reconstruction. D indicates a non-optimal source-level reconstruction, and E shows ChiselTrace data-flow reconstruction in the absence of simulation data.

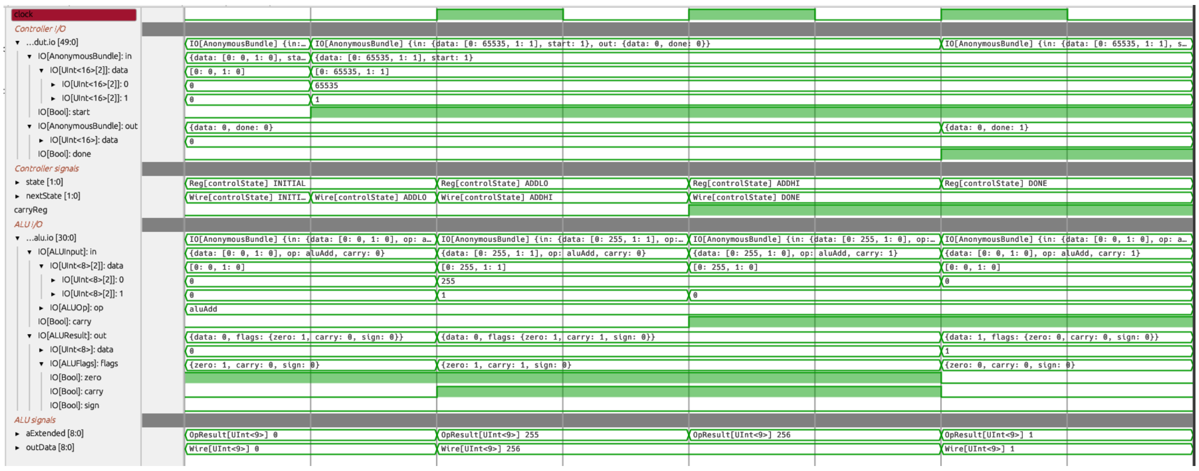


Figure 6.5: Surfer-Tywaves view of the ALU + Controller circuit.

Figure 6.4, where no Tywaves data-annotations are visible). This can be explained by the fact that the control- and data-flow reconstruction is based on probe signals that are never optimised out of a design.

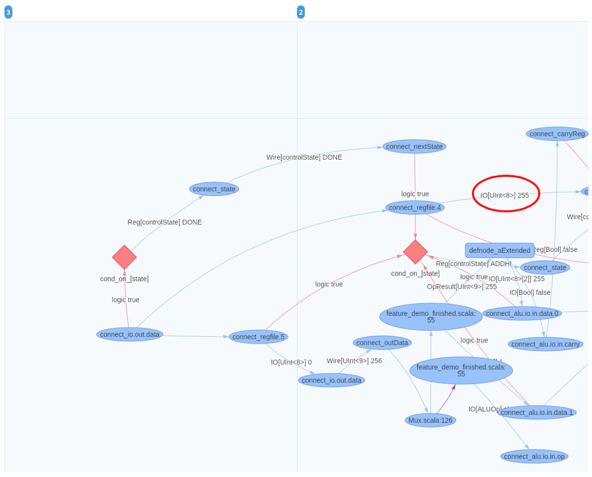
- Where a waveform viewer excels at displaying many signals at once, it does not show data-, index-, or control-flow. This means that hardware designers must infer these connections based on their design knowledge. ChiselTrace, on the other hand, clearly shows these connections.

ChiselTrace debugging scenarios

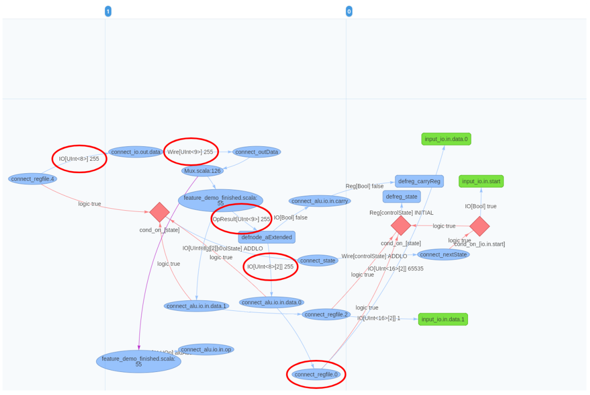
To illustrate how ChiselTrace can be useful for debugging, two behavioural debugging scenarios are considered. First, it is shown how ChiselTrace can be applied to trace erroneous data through the example circuit. This process can often be tedious for larger designs (see Section 6.2), because it requires the designer to select all relevant signals, while cross-referencing the simulation data with the source code.

To emulate a plausible bug, the data loading in the `INITIAL` state is modified to be big-endian instead of little-endian. This causes the propagation of wrong data from the input to the output signals. The bug is now investigated using ChiselTrace. Figure 6.6 shows how this can be achieved. First, the value we wish to trace back to the source is located in Figure 6.6a. Then, the data dependencies are followed to the defining statement. At this point, we can make use of the graph viewer's ability to reset the graph head to any node. This eliminates unrelated paths. The bug can now be traced back to the data loading by following data dependencies (see Figure 6.6b).

Another type of bug that can be found using ChiselTrace is caused by wrong dependencies. For example, at the `DONE` state, the output is set using the values at addresses 4 and 5 of the register file. If by accident, this were to be two times the value at address 5, it would be immediately visible in ChiselTrace. This is shown in Figure 6.7. Here, the only data dependency of the output signal is the value at address 5. In a waveform viewer, this type of bug is more difficult to find because it only shows the value of a signal, not how this value was formed.

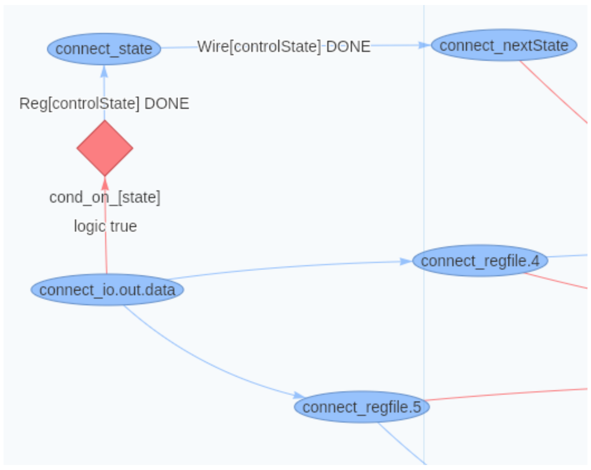


(a) Initial ChiselTrace view. The erroneous value can be spotted using the Tywaves data annotations.

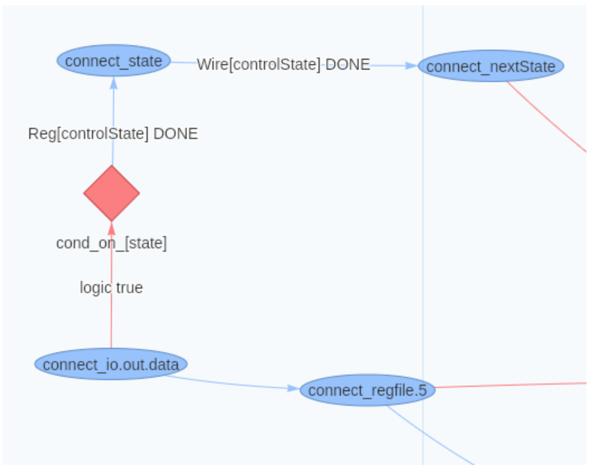


(b) By resetting the graph head, the search space is reduced. Now the value can be traced back to the data loading.

Figure 6.6: Debugging an erroneous value propagation in ChiselTrace. The traced signal path is indicated with red circles, before finally finding the root cause (`connect_regfile.0`).



(a) The expected dependencies of the output node are registers 4 and 5.



(b) The actual dependencies only include register 5, which can easily be seen in ChiselTrace.

Figure 6.7: A mismatch between actual dependencies and the mental model of the design can be found by looking at the statement dependencies in the graph. Here, only a dependency to `regfile.5` is visible, while this should be `regfile.4` and `regfile.5`.

```
class ALUFlags extends Bundle {
  val zero = Bool()
  val carry = Bool()
  val sign = Bool()
}

class ALUResult extends Bundle {
  val data = UInt(8.W)
  val flags = new ALUFlags()
}

object ALUOp extends ChiselEnum {
  val aluAdd, aluSub, aluOr, aluXor, aluAnd, aluLSL, aluLSR = Value
}

class ALUInput extends Bundle {
  val data = Vec(2, UInt(8.W))
  val op = ALUOp()
  val carry = Bool()
}

class ALU extends Module {
  val io = IO(new Bundle {
    val in  = Input(new ALUInput())
    val out = Output(new ALUResult())
  })

  // Allow for any overflow in the addition
  val outData = Wire(UInt(9.W))

  // Add carry to operand 1
  val aExtended = Cat(0.U(1.W), io.in.data(0)) + io.in.carry.asUInt

  // We use a high-level Chisel construct that will translate to many MUXes
  outData := MuxCase(0.U, Seq(
    (io.in.op === ALUOp.aluAdd) -> (aExtended + io.in.data(1).zext.asUInt),
    (io.in.op === ALUOp.aluSub) -> (aExtended - io.in.data(1).zext.asUInt),
    (io.in.op === ALUOp.aluOr)  -> (aExtended(7, 0) | io.in.data(1)),
    (io.in.op === ALUOp.aluXor) -> (aExtended(7, 0) ^ io.in.data(1)),
    (io.in.op === ALUOp.aluAnd) -> (aExtended(7, 0) & io.in.data(1)),
    (io.in.op === ALUOp.aluLSL) -> (aExtended(7, 0) << io.in.data(1)(2,0))(8,0),
    (io.in.op === ALUOp.aluLSR) -> (aExtended(7, 0) >> io.in.data(1)(2,0)).zext.asUInt
  ))

  io.out.data := outData(7, 0)  // Truncate
  io.out.flags.zero  := io.out.data === 0.U
  io.out.flags.carry := outData(8)  // 9th bit indicates carry
  io.out.flags.sign  := io.out.data(7)  // MSB
}

class Controller extends Module {
  val io = IO(new Bundle {
    val in = Input(new Bundle {
      val data = Vec(2, UInt(16.W))
      val start = Bool()
    })
    val out = Output(new Bundle {
      val data = UInt(16.W)
      val done = Bool()
    })
  })
  object controlState extends ChiselEnum { val INITIAL, ADDLO, ADDHI, DONE = Value }

  // Byte-addressed register file with a word length of 8 bits.
  val regfile = Mem(6, UInt(8.W))

  val nextState = WireDefault(controlState.INITIAL)
  val state = RegNext(nextState)
  val carryReg = RegInit(false.B)

  val alu = Module(new ALU())

  // Set defaults
  io.out.data := 0.U
  io.out.done := false.B
  alu.io.in.data(0) := 0.U
  alu.io.in.data(1) := 0.U

  switch (state) {
    is (controlState.INITIAL) {
      // Store the two in a little-endian fashion
      regfile(0) := io.in.data(0)(7, 0)
      regfile(1) := io.in.data(0)(15, 8)
      regfile(2) := io.in.data(1)(7, 0)
      regfile(3) := io.in.data(1)(15, 8)
      when (io.in.start) {
        nextState := controlState.ADDLO
      }
    }
    is (controlState.ADDLO) {
      alu.io.in.data(0) := regfile(0)
      alu.io.in.data(1) := regfile(2)
      regfile(4) := alu.io.out.data
      carryReg := alu.io.out.flags.carry
      nextState := controlState.ADDHI
    }
    is (controlState.ADDHI) {
      alu.io.in.data(0) := regfile(1)
      alu.io.in.data(1) := regfile(3)
      regfile(5) := alu.io.out.data
      nextState := controlState.DONE
    }
    is (controlState.DONE) {
      io.out.data := Cat(regfile(5), regfile(4))
      io.out.done := true.B
      nextState := controlState.DONE
    }
  }

  alu.io.in.op := ALUOp.aluAdd // The op is always add
  // Handle the carry signal
  alu.io.in.carry := carryReg
  state := nextState
}
```

Listing 6.3: An example 8-bit ALU circuit with a controller for 16-bit addition.

6.2. ChiselWatt

In this section, a real-world example of ChiselTrace is presented. Namely, the tool is used to debug an existing processor design, ChiselWatt. ChiselTrace is compared to the more traditional Tywaves waveform viewer, and proxy metrics for required debugging effort are presented.

ChiselWatt is a soft-core processor written in Chisel that partially implements the OpenPower POWER v3.0 ISA [22]. It consists of a pipeline that processes one instruction at a time. Furthermore, it has hardware support for various arithmetic and logic operations (see Appendix A for more details on ChiselWatt). In this thesis, a version of ChiselWatt is used that has been modified to work with recent Chisel versions and Tywaves [57].

To present the debugging capabilities of ChiselTrace and the advantages compared to a more traditional waveform viewer, a fault is injected into the logic unit of the processor. Namely, the XOR operation is changed to an OR operation. This will result in an erroneous value at the output of the logic module that will be propagated through the circuit (much like debugging scenario 1 from the previous section).

A test program is used to demonstrate how an incorrect observed output can be traced back to the source statement containing a bug. This program is shown in Listing 6.4. It first loads the values 5 and 3 into registers 3 and 4. Then, an XOR and ADDI 2 (add an immediate value of 2) are performed on this data. A correctly functioning processor should produce the value 8 in register 6, but value 9 is observed due to the injected fault.

```
li 3, 0b0101
li 4, 0b0011
xor 5, 3, 4 # Fault will cause 7 in register 5
addi 6, 5, 2 # Will result in 9 in register 6
blr
```

Listing 6.4: ChiselWatt assembly test program

Figures 6.8 and 6.9 show the more conventional Surfer-Tywaves waveform viewer and the ChiselTrace view, respectively. In the example, control flow and index flow have been disabled for clarity (for the full debugging session with ChiselTrace, see Appendix B). The criterion is the output of the adder module. By looking at the Tywaves data annotations on the graph edges, the initial fault path is determined. Namely, it can be seen that the erroneous value 9 is produced using a 2 and a 7. Knowing that the 2 comes from the immediate value of the ADDI instruction, the data path of the 7 is investigated further. The injected fault in `logical.scala` at line 29 can now be located by following the data dependencies that ChiselTrace traced through multiple hierarchical levels.

In Figure 6.8, all signals that need inspection need to be chosen manually from all signals in the design (in the figure, this has already been done). Then, the designer needs to follow the fault propagation path back to the bug (shown with numbering). Both of these steps require significant design understanding. Furthermore, not all Chisel statements may have a corresponding signal, making this analysis more difficult. ChiselTrace, on the other hand, is able to automatically determine the Chisel statements that are related to the criterion and find their dependencies.

Lastly, two proxy metrics are investigated to provide an anecdotal quantification of the savings in effort required to debug the fault in the example.

- We first look at the amount of "debugging actions" saved by using ChiselTrace. Each manual source statement inspection requires: locating the signal in the source code; locating the signal in the waveform, where applicable; and determining which dependencies to analyse next. Furthermore, an action is defined for dependency following for ChiselTrace. This last action includes making the decision to further follow the trace in the graph. Given that these actions roughly represent an equal amount of effort, we can compare the required debugging effort in both cases. For manual waveform debugging, 81 actions were required, as opposed to 19 actions using ChiselTrace (a reduction of 76.5%).

- The number of lines of code in the DPDG relevant to the fault path was analysed in comparison with the total number of lines in relevant modules calculated from the PDG. Here, we define lines relevant to the fault path as corresponding with a node for the DPDG (up until the timestep of the fault). For the PDG, the number of relevant lines is determined by taking all modules present in the graph up until the timestep of the fault, then enumerating all lines. This resulted in 298 lines for the PDG. ChiselTrace reduced this amount to 62 lines using the DPDG (a reduction of 79.2%), of which 17 were in the fault path.

6.3. Performance & scaling

So far, ChiselTrace functionality has been demonstrated on synthetic and real-world examples. An important part of usability in practice, however, has not yet been discussed. Namely, the performance and the scaling thereof with circuit size and simulation length. Below, the performance of the various stages of ChiselTrace is presented.

To measure performance, a synthetic test is used, which consists of an adder tree circuit. This circuit can be scaled up in size (amount of nodes in the PDG) in a consistent manner. For the measurements, adder trees with 2^n inputs are used. The value for n ranges between 2 and 12, resulting in 2 to 4096 inputs. The scaling with respect to the amount of PDG nodes in the circuit is investigated for the PDG building stage and the DPDG building stage. The performance of the other stages is measured with respect to the number of nodes in the DPDG. Lastly, the scaling with respect to simulation time is investigated for the DPDG generation stage. This is not done for the Tywaves simulation data injection stage, because it only heavily depends on the amount of DPDG nodes. For the scaling measurements with respect to the circuit size, a simulation length of 200 timesteps is used. For the measurement with respect to simulation time, an adder circuit with 1024 inputs is used.

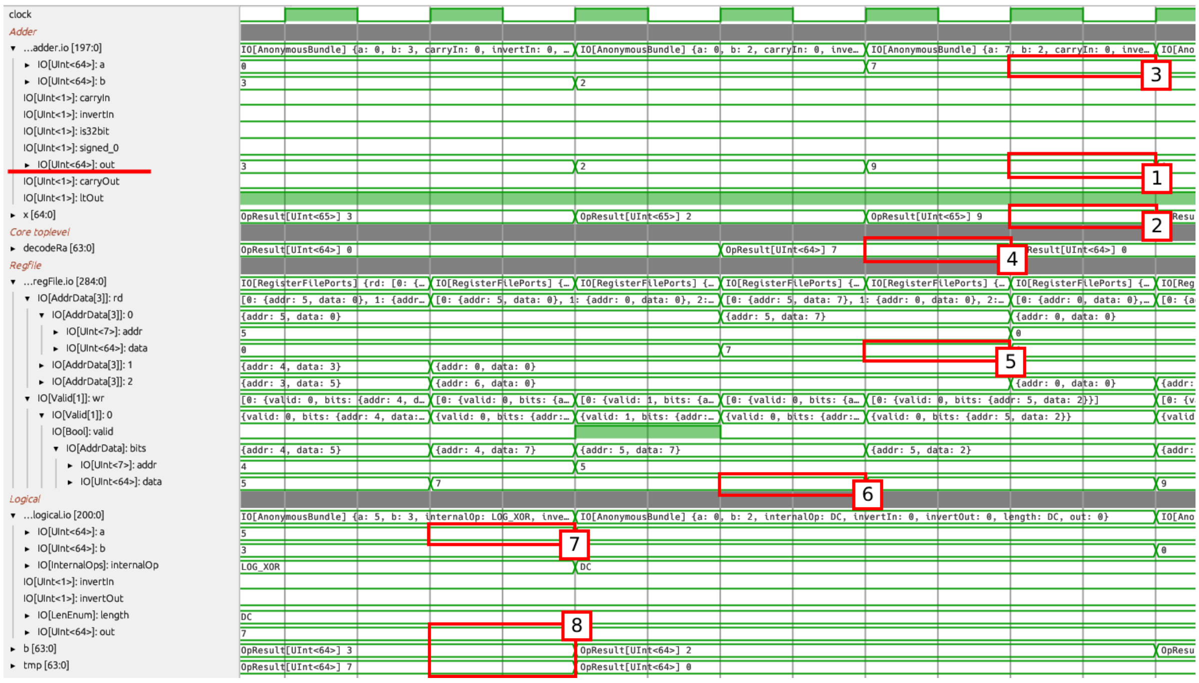
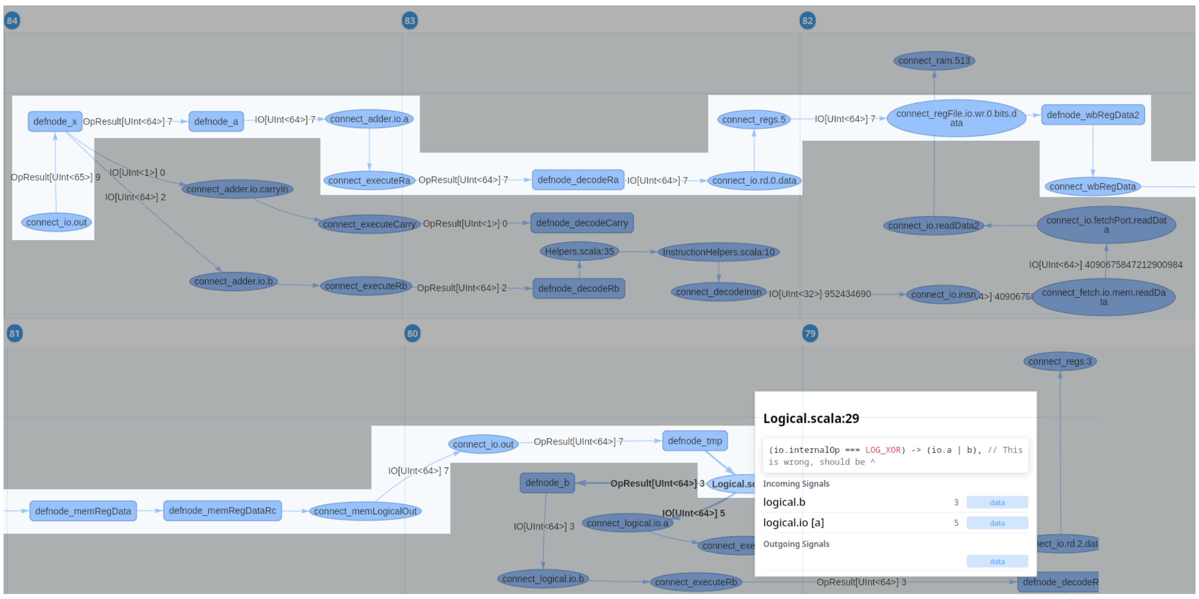


Figure 6.8: Tywaves waveform of the ChiselWatt simulation. Relevant signals have been highlighted. The criterion signal is underlined. The numbers indicate the order of manual signal inspection required to locate the fault.



```
(io.internalOp === LOG_XOR) -> (io.a | b), // This
is wrong, should be ^
```

Figure 6.9: ChiselTrace view of the ChiselWatt simulation. Only data flow (no control flow or index flow) is shown. Post-processing has been applied to highlight the relevant data path.

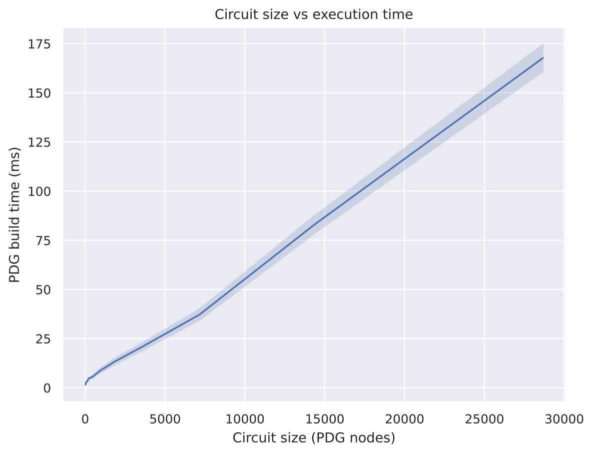


Figure 6.10: Performance scaling of the PDG generation stage in the Chisel extension.

The scaling of the PDG building stage is presented in Figure 6.10. It is observed that this stage displays near-linear scaling. Next, the performance of the various parts of `chiseltrace-rs` is inspected in Figure 6.11. It is observed that the DPDG generation scales linearly with simulation size (see Figure 6.11b). Furthermore, all stages scale super-linearly with respect to (D)PDG size.

For comparison, the ChiselWatt example from Section 6.2 consisted of 19663 PDG nodes, 1188 DPDG nodes and was run for 168 timesteps. On average, computing the DPDG for this example took 846.4 ms. This is roughly in line with the expectations.

A further observation is made that the addition of probe signals to the circuit slows down the simulation. However, since the simulation speed depends on the circuit being simulated and the simulator being used, no performance measurements are shown for this part. As an indication, the ChiselWatt compilation/simulation takes 8.2s (50%) longer.

6.4. Identifying limitations

Throughout this chapter, the capabilities of ChiselTrace have been demonstrated. It was shown that ChiselTrace functions on both small examples and real-world designs. Furthermore, debugging scenarios were presented where the value of ChiselTrace over a traditional waveform viewer was shown. There are, however, limitations to ChiselTrace, some of which were already touched upon. This section discusses the current limitations of ChiselTrace.

- It was shown in area $\boxed{\text{D}}$ of Figure 6.4 that situations can occur where the nodes in the generated DPDG do not directly correspond with Chisel statements. These cases occur most often when working with higher-level constructs in Chisel, such as the `MuxCase` that is implemented by the standard library. A similar situation would occur if a statement spans multiple lines of source code.

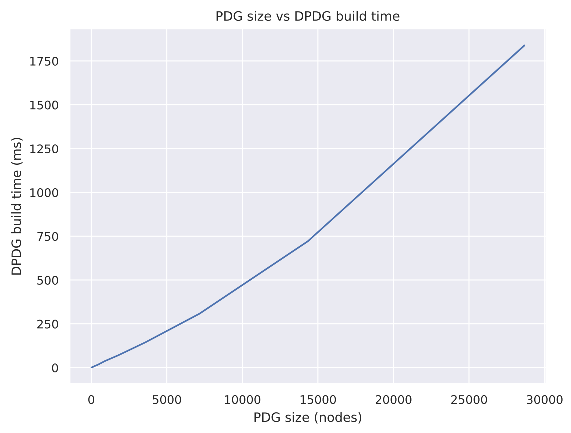
  The issue lies with the way FIRRTL source mappings are generated. When constructs from the standard library are used, the FIRRTL statements that are generated have mappings to files in the standard library. Because the FIRRTL to Chisel representation conversion in `chiseltrace-rs` relies on grouping by source mappings, nodes can be generated that do not correspond with statements in the DUT. Similarly, if a statement spans multiple lines, multiple nodes may be generated for the same Chisel statement.

  This issue could be addressed in multiple ways. Firstly, more heuristics could be added to the conversion stage of `chiseltrace-rs` that would make the conversion more accurate. Additionally, switching to a statement-based source-mapping would also address this issue, but this would require modifying more Chisel internals.

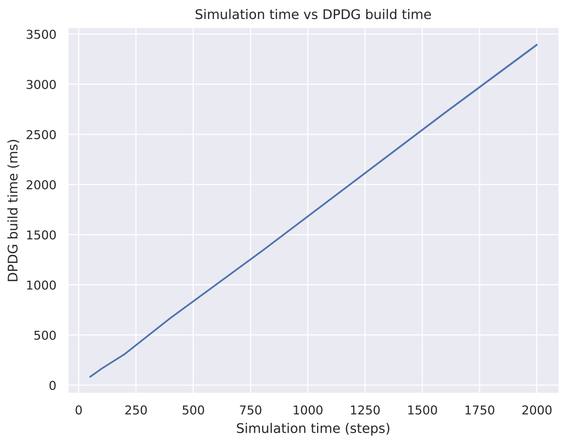
- Listing 6.2 showed (dynamic) program slicing. In the example, it can be seen that the produced slices are not executable (i.e., not all lines required for an equivalent circuit are present). Module and type definitions are not included in the slice, and some statements are not included entirely.

A further issue with the program slices produced by ChiselTrace is that under certain circumstances, statements are missing from the slice (e.g., when expressions are inlined during translation to FIRRTL). These issues mostly stem from the non-optimal FIRRTL source-mappings. By addressing this and adding post-processing to the slices, they could be made more accurate. Since program slicing was not the main focus of this thesis, this was not explored.

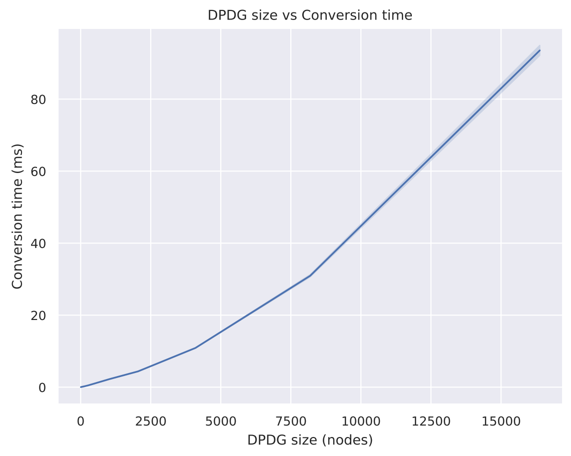
- Despite using hierarchical node grouping methods, the DPDGs that ChiselTrace produces are still large for larger designs, hindering swift debugging. More aggressive graph size reduction heuristics could be used to address this issue.
- While it was demonstrated that the PDG building and DPDG building scaled linearly with circuit size and simulation time, respectively, other parts showed superlinear scaling. On larger real-world designs, ChiselTrace may become prohibitively slow.



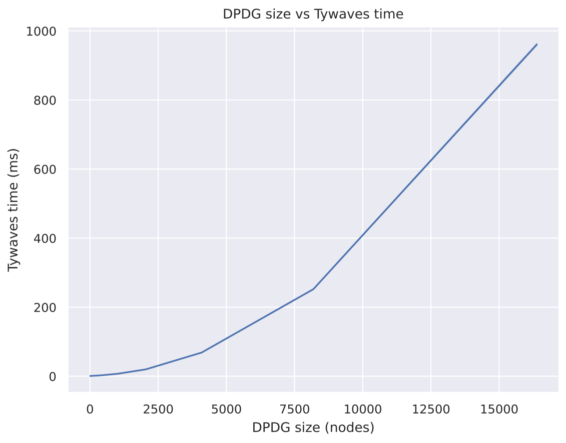
(a) Performance scaling of DPDG generation with respect to PDG size.



(b) Performance scaling of DPDG generation with respect to simulation length.



(c) Performance scaling of FIRRTL to Chisel representation conversion with respect to DPDG size.



(d) Performance scaling of Tywaves simulation data injection with respect to DPDG size.

Figure 6.11: Scaling behaviour of the various stages of chiseltrace-rs.

7 Conclusion

7.1. Summary of findings

Despite advances in debug tooling, modern HGLs still lack methods to debug hardware designs at the source level. This forces engineers to spend more time debugging their designs with a limited amount of tools, or to debug at the generated HDL level. The main research question that was answered in this thesis is:

> *How can debugging via manual waveform inspection be enhanced using automated dependency analysis at the HGL source level, bringing debugging closer to the source level and reducing the mental overhead for circuit debugging?*

To answer this question, ChiselTrace was introduced. ChiselTrace is an open-source, source-level debugging tool for Chisel, building on the Tywaves project, which is capable of creating and visualising DPDGs of Chisel designs. The design of ChiselTrace was informed by the following supporting research questions:

1. *How can dependencies between Chisel statements be found?*
    - In Chapter 2, we saw that directly finding dependencies between Chisel statements is difficult, due to Scala meta-programming. To find these dependencies, in Chapter 3, a method was introduced to build a PDG on the FIRRTL level. Using the FIRRTL source-mappings and the reconstruction method presented in Chapter 4, we could find dependence graphs at the Chisel source level.
2. *How can dynamic data- and control-flow dependencies be resolved using simulation data?*
    - To build a DPDG, only active statements should be included. This was achieved by the use of probe signals. In the FIRRTL circuit, predicate and dynamic data-flow expressions were replaced by probe signals (see Chapter 3). The values of these signals were then used to reconstruct control- and data-flow for each clock cycle of the simulation (Chapter 4).
3. *How can a dynamic program dependence graph of a FIRRTL circuit be created?*
    - A DPDG is created by reconstructing the active FIRRTL statements for each clock cycle of a simulation, then forming a graph based on the dependencies and provided symbols of the active statements (see Chapter 4).
4. *Which steps are required to convert program dependence graphs from a FIRRTL representation to a Chisel representation?*
    - A method to convert FIRRTL DPDGs to a Chisel representation is to group the graph nodes by source-mapping, then converting the groups to single nodes, based on the nodes contained in a group (see Chapter 4).

5. *What is an effective way to visualise a dynamic dependence graph for debugging purposes?*
    - Debugging using the DPDG can be achieved using a timeline-based, interactive, graph viewer that is capable of showing different types of dependencies between statements, and contains typed simulation data (see Chapter 5).

The functionality and applicability of ChiselTrace to debugging were demonstrated in Chapter 6. ChiselTrace was used on a real-world design, ChiselWatt, to trace an erroneous value back to an injected fault. Waveform debugging required significant design understanding and effort to select the relevant signals, while ChiselTrace was able to automatically determine the relevant signals. Furthermore, ChiselTrace was capable of showing statement dependencies, even in the absence of simulation data, due to its data-flow reconstruction. Two proxy metrics for debugging effort were presented to provide anecdotal evidence on the effectiveness of ChiselTrace. Namely, on a representative example, ChiselTrace reduced the amount of debugging actions by 76.5% and the relevant number of lines of code by 79.2%. Finally, it was shown that, while the performance of parts of ChiselTrace scales superlinearly, ChiselTrace is still capable of being used on real-world designs.

7.2. Recommendations for future work

This work has presented a method that enables dynamic building of program dependence graphs of signals at the Chisel source level. There are several recommendations for further improvement of ChiselTrace:

- Improve the Chisel reconstruction from DPDGs in FIRRTL representation. In Section 6.4, it was seen that the reconstruction stage of ChiselTrace sometimes fails to reconstruct an accurate Chisel view for nodes, especially if higher-level standard library constructs are involved. Methods that could solve this issue could consist of more heuristic graph processing in the reconstruction stage, and switching to different FIRRTL source-mappings that map to a specific Chisel statement instead of a location in the source code.
- Extend the functionality to work on multiple clock-domains and allow registers to use any signal as clock or reset. This would involve changes in the information that needs to be collected from the FIRRTL circuit and changes in the DPDG building.
- Integrate the ChiselTrace library with the Tywaves-Surfer waveform viewer to enable automatically adding signals to the waveform viewer that are dependencies of a particular transition. Furthermore, this could enable jumping to the location of an active driver of a signal in the source code.
- Add more graph processing to the front-end. One improvement could be automatically tracing a value to its root cause. This is a feature that also exists in Verdi's Temporal Flow View. Another interesting addition would be automatically calculating the difference between the DPDG under a test that produces wrong results versus a test that executes correctly.
- Create alternative front-ends for ChiselTrace, such as a Visual Studio Code extension. Such an extension could enable a user to jump to the active driver of a signal, similar to how a jump to definition works.
- Enable user-defined abstractions for data-flow. An example of this could be a communication transaction between components. An abstracted view could show only one data dependency for the entire transaction.
- Extend ChiselTrace to other HGLs. The ChiselTrace front-end and `chiseltrace-rs` could be partially reused for an implementation for another language. One big challenge would be to come up with an alternative way to processing at the FIRRTL level, as not all languages translate to a similar IR before compilation.

Bibliography

[1] Google DeepMind. 'Pexels.' (2025), [Online]. Available: https://www.pexels.com/photo/an-artist-s-illustration-of-artificial-intelligence-ai-this-image-was-inspired-neural-networks-used-in-deep-learning-it-was-created-by-novoto-studio-as-part-of-the-visualising-ai-proje-17483870/ (visited on 14/08/2025).

[2] R. S. Williams, 'What's next? [the end of moore's law],' *Computing in Science & Engineering*, vol. 19, no. 2, pp. 7–13, 2017. doi: 10.1109/MCSE.2017.31.

[3] T. N. Theis and H.-S. P. Wong, 'The end of moore's law: A new beginning for information technology,' *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, 2017. doi: 10.1109/MCSE.2017.29.

[4] W. J. Dally, Y. Turakhia and S. Han, 'Domain-specific hardware accelerators,' *Commun. ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020, issn: 0001-0782. doi: 10.1145/3361682. [Online]. Available: https://doi.org/10.1145/3361682.

[5] E. D. Sozzo, D. Conficconi, A. Zeni, M. Salaris, D. Sciuto and M. D. Santambrogio, 'Pushing the level of abstraction of digital system design: A survey on how to program fpgas,' *ACM Comput. Surv.*, vol. 55, no. 5, Dec. 2022, issn: 0360-0300. doi: 10.1145/3532989. [Online]. Available: https://doi.org/10.1145/3532989.

[6] C. Baaij, 'Clash: From haskell to hardware,' University of Twente, Tech. Rep., Dec. 2009.

[7] J. Decaluwe, 'Myhdl: A python-based hardware description language,' *Linux J.*, vol. 2004, no. 127, p. 5, Nov. 2004, issn: 1075-3583.

[8] C. Papon. 'Spinalhdl: An alternative hardware description langage presentation.' FOSDEM 2017 presentation, FOSDEM. (2017), [Online]. Available: https://archive.fosdem.org/2017/schedule/event/spinal_hdl/ (visited on 01/05/2025).

[9] J. Bachrach, H. Vo, B. Richards *et al.*, 'Chisel: Constructing hardware in a scala embedded language,' in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221. doi: 10.1145/2228360.2228584.

[10] M. Käyrä and T. D. Hämäläinen, 'A survey on system-on-a-chip design using chisel hw construction language,' in *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society*, 2021, pp. 1–6. doi: 10.1109/IECON48115.2021.9589614.

[11] Synopsys, *Verdi*. [Online]. Available: https://www.synopsys.com/verification/debug/verdi.html.

[12] GTKWave contributors. 'Gtkwave.' (2025), [Online]. Available: https://github.com/gtkwave/gtkwave (visited on 20/07/2025).

[13] F. Skarman, L. Klemmer, K. Laeufer and O. Gustafsson, *Surfer 0.3.0*, version 0.3.0, Dec. 2024. doi: 10.5281/zenodo.14536893. [Online]. Available: https://doi.org/10.5281/zenodo.14536893.

[14] SymbiYosys contributors. 'Symbiyosys.' (2025), [Online]. Available: https://github.com/YosysHQ/sby (visited on 20/07/2025).

[15] VUnit contributors. 'Vunit.' (2025), [Online]. Available: https://vunit.github.io/ (visited on 20/07/2025).

[16] FOSSi Foundation and cocotb contributors. 'Cocotb.' (2025), [Online]. Available: https://www.cocotb.org/ (visited on 20/07/2025).

[17] K. Zhang, Z. Asgar and M. Horowitz, 'Bringing source-level debugging frameworks to hardware generators,' in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22, San Francisco, California: Association for Computing Machinery, 2022, pp. 1171–1176, isbn: 9781450391429. doi: `10.1145/3489517.3530603`. [Online]. Available: `https://doi.org/10.1145/3489517.3530603`.

[18] R. Meloni, H. P. Hofstee and Z. Al-Ars, 'Tywaves: A typed waveform viewer for chisel,' in *2024 IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2024, pp. 1–6. doi: `10.1109/NorCAS64408.2024.10752465`.

[19] K. Asanović, R. Avizienis, J. Bachrach *et al.*, 'The rocket chip generator,' UC Berkeley, Tech. Rep., Apr. 2016. [Online]. Available: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[20] C. Celio, 'A highly productive implementation of an out-of-order processor generator,' UC Berkeley, Tech. Rep., Dec. 2018. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-151.html`.

[21] A. Blanchard. 'Chiselwatt.' (), [Online]. Available: `https://github.com/antonblanchard/chiselwatt` (visited on 06/05/2025).

[22] 'Power isa (version 3.0c),' OpenPOWER Foundation, Tech. Rep., 2020. [Online]. Available: `www.openpowerfoundation.org`.

[23] M. Schoeberl, *Digital Design with Chisel*. Kindle Direct Publishing, 2019. [Online]. Available: `https://github.com/schoeberl/chisel-book`.

[24] S. Eldridge, P. Barua, A. Chapyzhenka *et al.*, 'Mlir as hardware compiler infrastructure,' in *Workshop on Open-Source EDA Technology (WOSET)*, Oct. 2021.

[25] P. S. Li, A. M. Izraelevitz and J. Bachrach, 'Specification for the firrtl language,' Tech. Rep. UCB/EECS-2016-9, Feb. 2016. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html`.

[26] SBT contributors. 'Sbt build tool.' (2025), [Online]. Available: `https://www.scala-sbt.org/` (visited on 20/07/2025).

[27] VirtusLab & Scala-CLI contributors. 'Scala-cli.' (2025), [Online]. Available: `https://scala-cli.virtuslab.org/` (visited on 20/07/2025).

[28] ScalaTest contributors. 'Scalatest.' (2025), [Online]. Available: `https://www.scalatest.org/` (visited on 20/07/2025).

[29] W. Snyder, P. Wasson, D. Galbi *et al.*, *Verilator*, `https://verilator.org`, 2025.

[30] Synopsys, *Vcs*. [Online]. Available: `https://www.synopsys.com/verification/simulation/vcs.html`.

[31] J. Peltenburg, J. Van Straten, M. Brobbel, Z. Al-Ars and H. P. Hofstee, 'Tydi: An open specification for complex data structures over hardware streams,' *IEEE Micro*, vol. 40, no. 4, pp. 120–130, 2020. doi: `10.1109/MM.2020.2996373`.

[32] Y. Tian, M. Reukers, Z. Al-Ars *et al.*, 'Tydi-lang: A language for typed streaming hardware,' in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23, Denver, CO, USA: Association for Computing Machinery, 2023, pp. 521–529, isbn: 9798400707858. doi: `10.1145/3624062.3624539`. [Online]. Available: `https://doi.org/10.1145/3624062.3624539`.

[33] M. Reukers, Y. Tian, Z. Al-Ars *et al.*, 'An intermediate representation for composable typed streaming dataflow designs,' English, *CEUR Workshop Proceedings*, vol. 3462, 2023, Joint Workshops at the 49th International Conference on Very Large Data Bases (VLDBW'23), issn: 1613-0073.

[34] C. Cromjongh, Y. Tian, P. Hofstee and Z. Al-Ars, 'Tydi-chisel: Collaborative and interface-driven data-streaming accelerators,' in *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2023, pp. 1–7. doi: `10.1109/NorCAS58970.2023.10305451`.

[35] M. Weiser, 'Program slicing,' *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984. doi: `10.1109/TSE.1984.5010248`.

[36] H. Agrawal and J. R. Horgan, 'Dynamic program slicing,' *SIGPLAN Not.*, vol. 25, no. 6, pp. 246–256, Jun. 1990, issn: 0362-1340. doi: 10.1145/93548.93576. [Online]. Available: https://doi.org/10.1145/93548.93576.

[37] A. Dobis, T. Petersen, K. J. H. Rasmussen *et al.*, *Open-source verification with chisel and scala*, 2021. arXiv: 2102.13460 [cs.PL]. [Online]. Available: https://arxiv.org/abs/2102.13460.

[38] 'Gdb: The gnu project debugger.' (2025), [Online]. Available: https://sourceware.org/gdb/ (visited on 20/07/2025).

[39] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar and T. Teitelbaum, 'Program slicing for vhdl,' *International Journal on Software Tools for Technology Transfer*, vol. 4, pp. 125–137, 1 Oct. 2002, issn: 14332779. doi: 10.1007/s100090100069.

[40] J.-c. Ou, D. G. Saab and J. A. Abraham, 'Hdl program slicing to reduce bounded model checking search overhead,' in *2006 IEEE International Test Conference*, 2006, pp. 1–7. doi: 10.1109/TEST.2006.297665.

[41] B. Alizadeh, P. Behnam and S. Sadeghi-Kohan, 'A scalable formal debugging approach with auto-correction capability based on static slicing and dynamic ranking for rtl datapath designs,' *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1564–1578, 2015. doi: 10.1109/TC.2014.2329687.

[42] A. C. Bagbaba, M. Jenihhin, J. Raik and C. Sauer, 'Accelerating transient fault injection campaigns by using dynamic hdl slicing,' in *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, 2019, pp. 1–7. doi: 10.1109/NORCHIP.2019.8906932.

[43] J. Hu and Z. Liu, 'Context aware deep learning-based fault localization for hardware design code,' *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2025. doi: 10.1109/TCAD.2025.3543426.

[44] Cadence, *Simvision*. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html.

[45] Siemens, *Questa sim*. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/.

[46] C.-T. Ho, H. Ren and B. Khailany, *Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool*, 2025. arXiv: 2408.08927 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2408.08927.

[47] S. Takamaeda-Yamazaki, 'Pyverilog: A python-based hardware design processing toolkit for verilog hdl,' in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040, Springer International Publishing, Apr. 2015, pp. 451–460. doi: 10.1007/978-3-319-16214-0_42. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16214-0_42.

[48] K. Laeufer, V. Iyer, D. Biancolin, J. Bachrach, B. Nikolić and K. Sen, 'Simulator independent coverage for rtl hardware languages,' in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 606–615, isbn: 9781450399180. doi: 10.1145/3582016.3582019. [Online]. Available: https://doi.org/10.1145/3582016.3582019.

[49] Chisel contributors. 'Chisel documentation.' (2025), [Online]. Available: https://www.chisel-lang.org/ (visited on 20/07/2025).

[50] serde contributors. 'Serde.' (2025), [Online]. Available: https://serde.rs/ (visited on 06/08/2025).

[51] rust-vcd contributors. 'Rust-vcd.' (2025), [Online]. Available: https://github.com/kevinmehall/rust-vcd (visited on 06/08/2025).

[52] GraphViz contributors. 'Graphviz.' (2025), [Online]. Available: https://graphviz.org/ (visited on 20/07/2025).

[53] Gephi contributors. ‘Gephi.’ (2025), [Online]. Available: `https://gephi.org/` (visited on 20/07/2025).

[54] Tauri contributors. ‘Tauri.’ (2025), [Online]. Available: `https://tauri.app/` (visited on 20/07/2025).

[55] Svelte contributors. ‘Svelte.’ (2025), [Online]. Available: `https://svelte.dev/` (visited on 20/07/2025).

[56] vis.js contributors. ‘Vis.js.’ (2025), [Online]. Available: `https://visjs.org/` (visited on 20/07/2025).

[57] R. Meloni and A. Blanchard. ‘Tywaves version of chiselwatt.’ (2025), [Online]. Available: `https://github.com/rameloni/chiselwatt/tree/migrate-to-chiselsim` (visited on 21/07/2025).

A
ChiselWatt

ChiselWatt [21] is an open-source soft-core processor that partially implements the OpenPower POWER v3.0 ISA [22]. The processor is implemented in Chisel and follows a simple design. Namely, it is a single-issue processor without branch prediction or a cache subsystem. Even though the design contains a pipeline, bypassing is not implemented, and only one instruction is in the pipeline at a given time. The pipeline itself follows a conventional structure (fetch, decode, execute, memory, write back).

Figure A.1 shows the various components that ChiselWatt comprises. There is a top-level module, called Core, which instantiates all other modules and is responsible for connecting them in a pipelined fashion. The NIA (next instruction address) module contains the program counter. It increments by 4 when a ready signal is asserted (using `Decoupled` I/O). It can also jump by setting a redirect signal. The Fetch unit is responsible for fetching the instruction pointed to by the program counter from memory.

The interface between memory and other components is modelled using a `Bundle` with inputs and outputs (`MemoryPort`), allowing for bidirectional communication between the modules. In the control unit, the fetched instruction is decoded into control signals for the various modules. In Chisel, this is achieved by using a `ListLookup` with `BitPattern`s for all instructions. The register file is parametrised to take in any number of read and write ports. Chiselwatt uses 3 read ports and 1 write port.

The LoadStore unit interfaces with memory for load and store instructions. Just like the Fetch unit, this happens over a `MemoryPort`. Furthermore, the UART module is memory-mapped in this module. UART is not used in the examples for this thesis. The memory module itself uses Chisel's `Mem`, which compiles to a register file. The memory is initialised at the start of a simulation using an external file. ChiselWatt contains various logic and arithmetic modules. These are controlled using decoded signals from the control unit.

The Chiselwatt repository includes an example program written in C. The C example includes a Docker container to cross-compile to PowerPC and create binaries. This can be used to generate binaries for custom assembly programs as well, allowing for the creation of custom tests for specific instructions. In this thesis, this is used to construct the example shown in Section 6.2. In this example, it can be seen that tracing an erroneous signal from the adder to a bug in the logic unit requires crossing modules (adder, core, register file, core, logical).

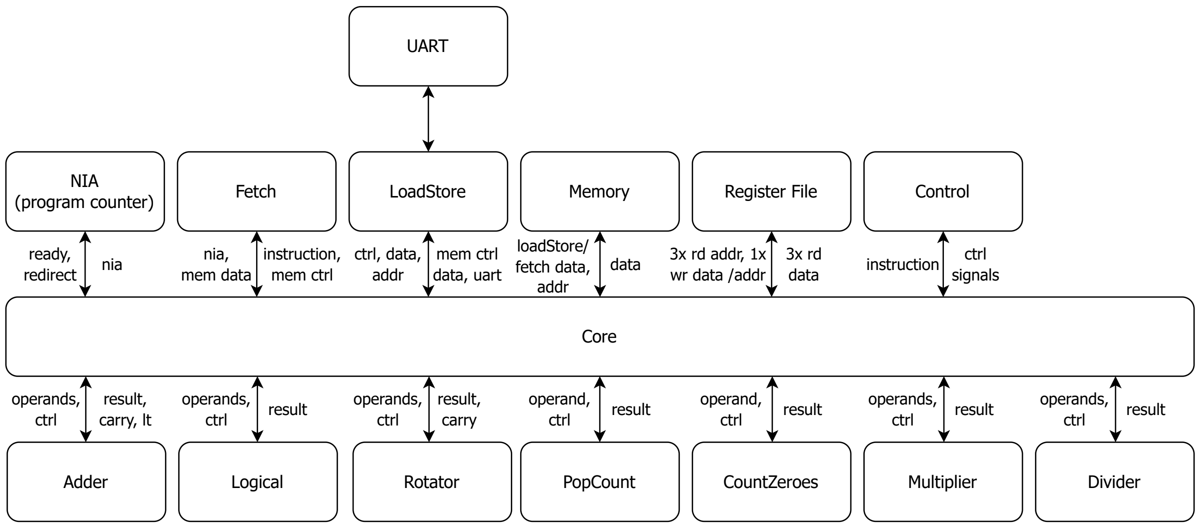


Figure A.1: Module diagram of ChiselWatt. The processor follows a conventional design. The toplevel Core is responsible for the pipeline, and all modules are connected to it. On the left of the arrows, signals coming into the module are shown, while on the right, signals going out of the module are shown.

B
Debugging with ChiselTrace

This appendix provides additional details on how ChiselTrace can be used to debug a hardware design. This is done by elaborating further on the steps required to find the fault of the ChiselWatt example in Section 6.2. Here, we show the end-to-end process, including a full dependency trace instead of a data-dependency-only trace.

As is the case in Section 6.2, a fault is inserted into the logic module of the ChiselWatt processor. Namely, the XOR operation is changed into an OR operation. The resulting logic module is shown in Listing B.1.

```scala
import chisel3._
import chisel3.util.{MuxCase, MuxLookup}
import circt.stage.ChiselStage

import Control.LenEnum._
import Control.InternalOps._
import Helpers._

class Logical(bits: Int) extends Module {
    val io         = IO(new Bundle {
    val a          = Input(UInt(bits.W))
    val b          = Input(UInt(bits.W))
    val internalOp = Input(Control.InternalOps())
    val invertIn   = Input(UInt(1.W))
    val invertOut  = Input(UInt(1.W))
    val length     = Input(Control.LenEnum())
    val out        = Output(UInt(bits.W))
  })

  val b = Mux(io.invertIn.asBool, ~io.b, io.b)

  val ext = MuxLookup(io.length, io.a.signExtend(8, bits))(Array(
              LEN_2B -> io.a.signExtend(16, bits),
              LEN_4B -> io.a.signExtend(32, bits)))

  val tmp = MuxCase(io.a, Seq(
      (io.internalOp === LOG_AND) -> (io.a & b),
      (io.internalOp === LOG_OR) -> (io.a | b),
      (io.internalOp === LOG_XOR) -> (io.a | b), // This is wrong, should be ^
      (io.internalOp === LOG_EXTS) -> ext
      ))

  io.out := Mux(io.invertOut.asBool, ~tmp, tmp)
}
```

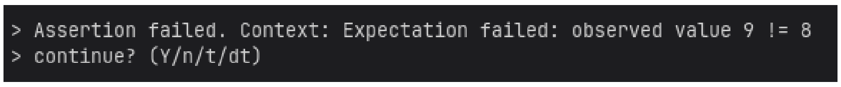
Listing B.1: Injected fault in the logic module of ChiselWatt [21]. The injected fault is highlighted in red.

Furthermore, a simple test program is used to trigger erroneous behaviour. Namely, it will result in a 9 at the output of the adder module, instead of the expected 8 (see Section 6.2 for the full details on the test program). The erroneous output is observed by running a unit test using the `ChiselTraceDebugger` introduced in Section 5.2. This is shown in Listing B.2. Here, first the program is loaded into memory, then it is executed, and at the end, we assert that the output of the adder module should have the value 8. Due to the injected fault, this assertion will fail, and ChiselTrace will pause the simulation and prompt the user to automatically trace back the dependencies of the asserted signal (see Figure B.1). Here, the options are a full trace or a data-dependency-only trace. In this case, we will select the full trace (t).

```scala
simulate(new Core(bits, words, filename, resetAddr, frequency), Seq(VcdTrace,
    WithTywavesWaveforms(true), SaveWorkdirFile("workDir")))
  { c =>
    // Write the test program to memory
    c.clock.step()
    c.reset.poke(true.B)
    c.clock.step()
    c.reset.poke(false.B)
    c.clock.step(83)
    // Assert the output value of the adder module
    c.io.adderOut.expect(8.U)
}
```

Listing B.2: Unit test used to find the injected fault. Adapted from [21], [57]. The assertion on the adder output will fail.

The `ChiselTraceDebugger` now opens a ChiselTrace session, which can be seen in Figure B.2. Here, we can see that there are many nodes influencing the adder output. First, we can identify the adder output. This is the node in the bottom left of the figure. Now, the erroneous value 9 can be traced to the node `x`, which takes in a 2 and a 7 to produce the 9. Knowing that the 2 is correct, we decide to proceed with the trace in the direction of the 7 (encircled in red in the figure). We do this by resetting the graph head to the 7-producing node in Figure B.3. This eliminates all nodes that are unrelated to the newly selected graph head, simplifying the DPDG. Figures B.4 and B.5 now show the same process of following dependencies and reducing the graph. Eventually, the data path enters the logic module. When the module is collapsed, all nodes of the module become visible. Based on this information (Figure B.6), the faulty statement in the source code can be located. The faulty line of code can be seen directly in ChiselTrace. Additionally, ChiselTrace provides the option to jump to the source location in the text editor.



```
> Assertion failed. Context: Expectation failed: observed value 9 != 8
> continue? (Y/n/t/dt)
```

Figure B.1: The assertion fails. The `ChiselTraceDebugger` now gives the option to continue the simulation or automatically trace back the dependencies of the asserted signal.

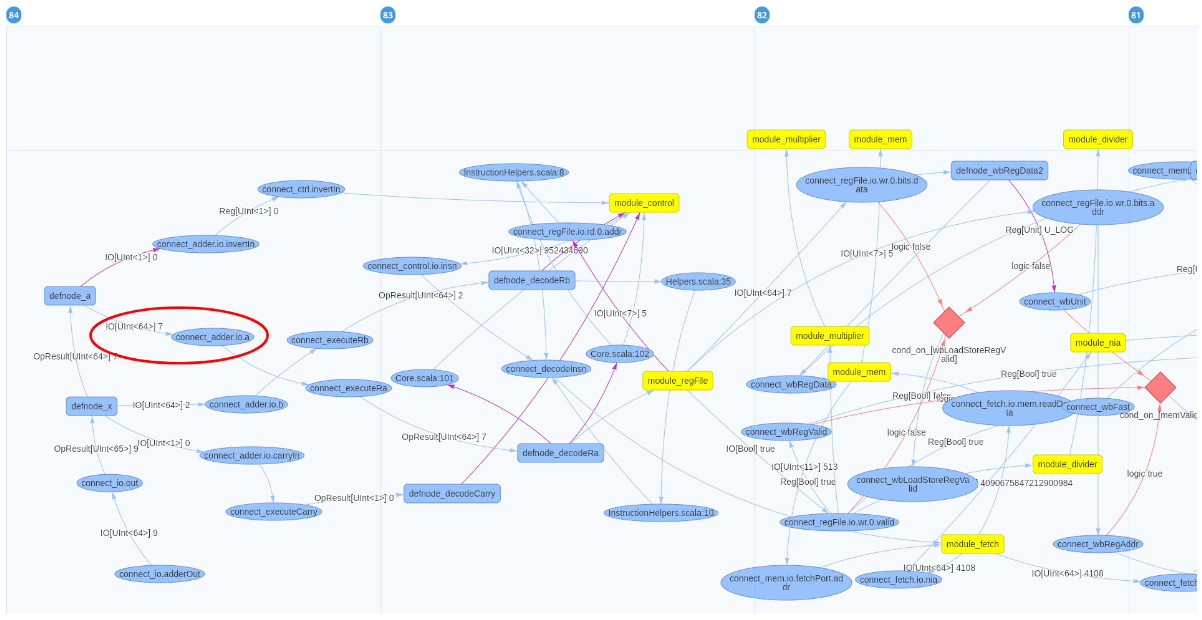


Figure B.2: The full ChiselTrace view of the ChiselWatt simulation. This view shows all types of dependencies (data, control flow, index). Furthermore, hierarchical node grouping is enabled for clarity. To find the fault, we start from the bottom (`connect_io.adderOut`). Then, we follow data dependencies. We decide to further investigate the red encircled path.

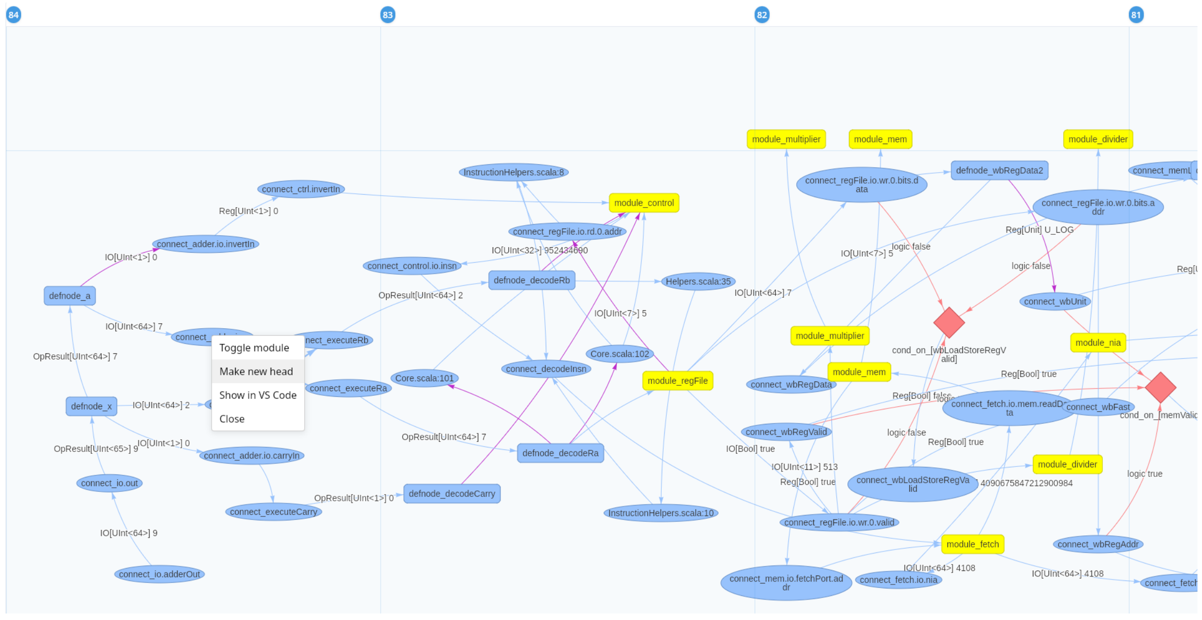


Figure B.3: Using the capability to reset the graph head, we can reduce the number of nodes in our search space.

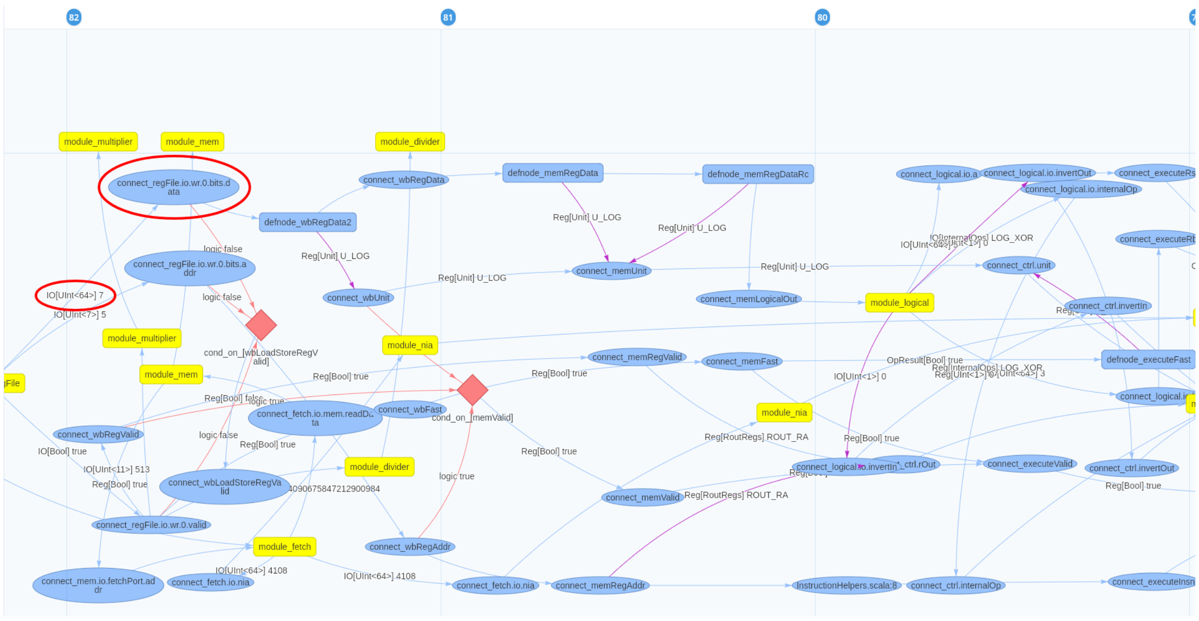


Figure B.4: A node is encircled that produces the value that is under investigation.

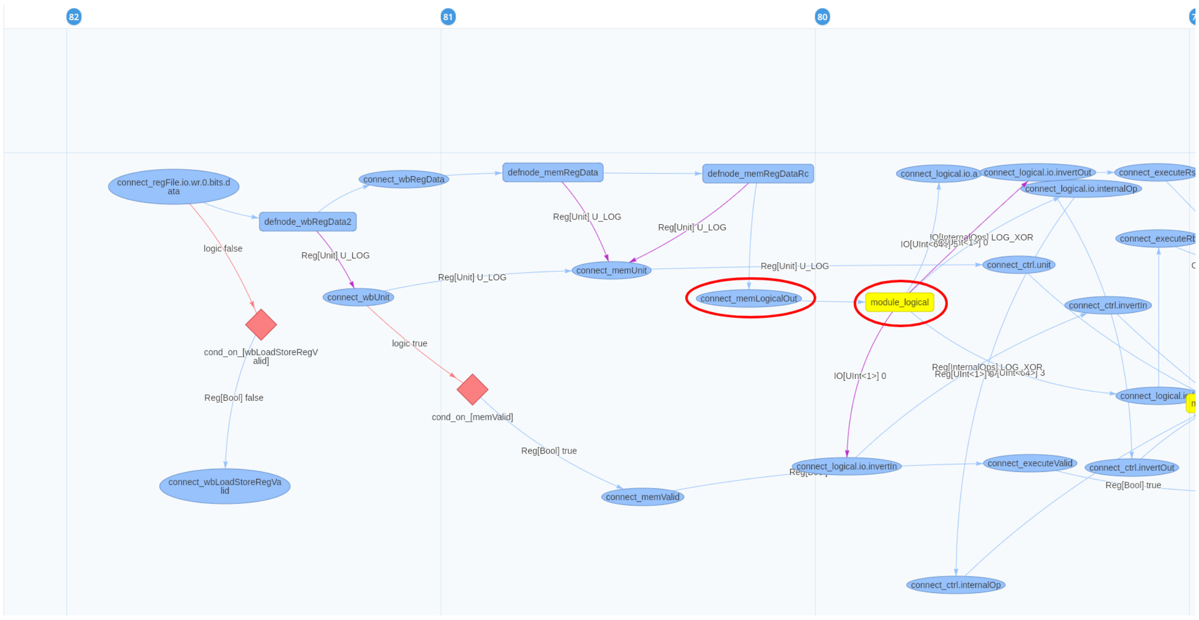
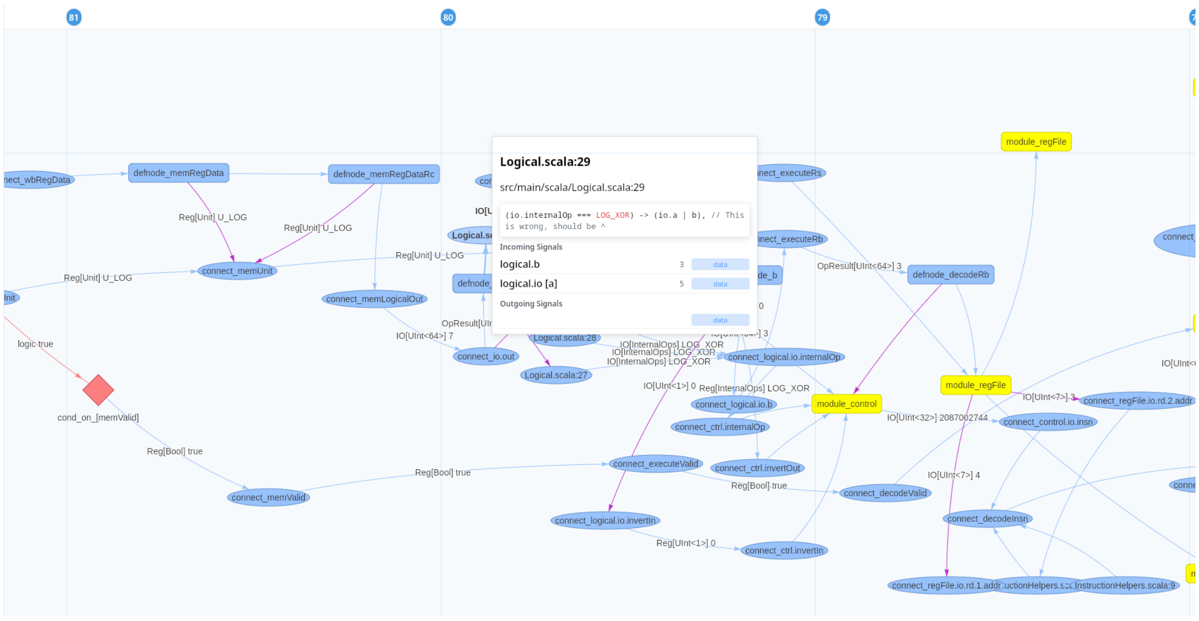


Figure B.5: The graph is once more reduced by resetting the head. The fault is now traced to the logic module.



Logical.scala:29

src/main/scala/Logical.scala:29

```
(io.internalOp === LOG_XOR) -> (io.a | b), // This
is wrong, should be ^
```

Incoming Signals

logical.b	3	data
logical.io [a]	5	data

Outgoing Signals

		data

Figure B.6: The logic module is collapsed and the faulty statement is located.