Amelia Dobis (ad4048@princeton.edu)

DVRTL
A Hardware Language for Deductive Verification

DVRTL Language Specification

May 6th 2025

Contents

1 Introduction

With the increasing demand in domain-specific accelerators and ever-tightening production times, more efficient tools for implementing hardware designs, in the form of high-level hardware languages, have been created [3, 5, 17, 20, 21, 23]. While these greatly improve the efficiency of implementing a design, verifying these designs still remains a tight bottleneck.

Throughout my research, I have been focusing on improving the efficiency of verification in high level hardware languages [7–10]. To do so, I have introduced the notion of first-class verification constructs to the CIRCT (pronounced 'circuit') core IR [11], a unified compiler that is used for most high-level hardware languages. One of the specific problems that were addressed was the lack of modularity in hardware verification systems. To solve this problem, I introduced the concept of hardware contracts, for formal verification, into the CIRCT compiler. This solution has since been fully implemented and integrated into the Chisel [2] frontend language.

However, while this method has been shown to work through various tests, it has yet to be formalized and proven to be a valid abstraction. In this project, I propose to do just that: to formalize hardware contracts by proposing exact semantics for the operation and proving the legality of the contract replacement within the context of bounded model checking.

In order to correctly formalize contracts, we must first define the semantics language in which we our formalizing it. We will thus start by proposing a minimal RTL language (*miniRTL*), following a similar level of abstraction as the core dialects but with simpler language constructs, and will incrementally expand that language to include assertions, modularity, and finally hardware contracts. We will then enable bounded model checking of our designs by defining a compilation to BTOR2 [16], which is a popular model checking format supported by several open-source solvers [15], and the verification target of choice for CIRCT [7].

2 Background

In this section, we will present the necessary background to understand how hardware languages work, and how we should go about designing them. I will then follow by presenting the concept of hardware contracts in the context of Chisel and the CIRCT compiler.

2.1 Hardware Languages and their Requirements

Hardware Languages differ in several ways from software languages, and those differences will have to be taken into account in our minimal RTL design.

First, we must define what an RTL language is. While being the most popular paradigm for writing hardware, Register Transfer Level (RTL) languages are surprisingly unclearly defined [1]. In my understanding, RTL refers to a language paradigm that requires the user to describe a circuit as structural declarations of stateful elements and logic that defines how those states are updated across a single time step, i.e. a clock cycle. This is the paradigm used in most hardware languages. The core unit of state in this paradigm is the register, which typically has a single output, i.e. the value it holds at the start of a clock cycle, and takes in 4 inputs: a reset signal, a reset value, a clock, and a next expression. It is common in RTL languages for clocks and reset signals to be

[1] This triggered a funny and lengthy debate between most of the folks currently researching hardware language design, during the language design session of the most recent ASPLOS conference, where nobody could come up with a clear definition of what RTL truly was, so we all just agreed that it is just vaguely defined.

implicit, as most designs are single clock, single reset designs. Other paradigms also exits, such as rule-based languages, e.g. BlueSpec [18], or High-Level Synthesis (HLS) [6, 19, 21, 22]. These are at a slightly higher-level than RTL, but are not as commonly used in practice.

Hardware languages are inherently structural rather than behavioral, meaning that the core idea of what we are describing is the structure of a digital circuit rather than the behavior of a program. This means that concepts like "scope" and "mutability" don't really apply to hardware, as we are essentially describing the placement and connection of logical components. As of result, hardware languages are also implicitly parallel and operations are interpreted following a dataflow ordering. While this won't necessarily strongly impact our semantics, it will impact the implementation of the language, as now all compiler passes must proceed in a Depth-First-Search ordering, rather than following the program order.

Some languages, such as SystemVerilog [1] or Chisel [2] attempt to give the user the feeling of writing a behavioral program, by allowing for constructs such as conditionals, events, or even functions. However, these are then elaborated into structural components, leading to elements such as registers (the core stateful element of an RTL language) to be inferred from the behavioral, usually event-driven, description.

2.2 Hardware Contracts

Hardware contracts describe an operation (in the MLIR [14] sense) that can be tied to a block of operations to describe a Hoare triple. This Hoare triple is formed, as in software, by wrapping a block with pre-conditions, which define constraints on the inputs of the block, and post-conditions, which define expectations that the outputs of the block should meet. This is implemented in CIRCT as follows:

```
verif.contract (<inputs>) : <input_types> { body }
```

where the inputs represent the block of operations that the contract will replace and the body represents the logic defining the pre- and post-conditions of that block. This definition is purely used as an IR by the compiler, the user-facing syntax is a lot more intuitive, e.g. in Chisel:

```
class A extends Module {
    val in = IO(Input(UInt(32.W)))
        val out = IO(Output(UInt(32.W)))
        contract {
            requires in >= 0.U
            ensures out === in + 42.U
        }
        // ... Module body ...
}
```

In most open-source verification tools, module instances cause the instanced module to be inlined in its place during verification [7, 12, 24]. This makes bounded model checking problems very difficult to parallelize, and causes them to scale quite poorly, leading to most verification tasks being limited to small units at a time. With the use of contracts, these tasks can be split into smaller parallel problems. This greatly speeds up verification and makes it far more scalable, as verifying modules instances no longer incurs any additional cost. To ensure correctness, a separate proof obligation is generated for each module, where the goal of these obligations is to prove each module's contract using the body of the module.

3 A Minimal RTL Language (*miniRTL*)

Let us begin our formalization by defining our minimal RTL language. Let's start with the simplest possible RTL language that supports 1-bit sequential circuits defined as a series of nested combinational logic expressions used to update the value of a register.

3.1 Syntax

```
Registers r
Value v          ::= 0 | 1
Expression e     ::= e xor e | e and e | e or e | mux e e e | v | r
Circuit C        ::= [] | C ; [r -> v, e]
```

Figure 1: Syntax of the *miniRTL* language.

Figure 1 defines the syntax for our minimal RTL language. Register definitions can be mutually recursive. Referring to a register by its name `r` reads the value it has at the beginning of a cycle. We chose not to encode the `not` operation, as `not e` $\equiv$ `e xor 1`[2]. In order to simplify writing example programs, we will also allow for a line break to be use in place of the delimiter `;`.

3.2 Semantics

We will define the semantics of our language using *denotational semantics*. The semantics of *miniRTL* are simple enough to be denoted in boolean algebra with the only subtlety being that the denotations of registers must allow for mutual recursion.

$$
\begin{array}{lcl}
[\![\ 0\ ]\!] & = & 0 \\
[\![\ 1\ ]\!] & = & 1 \\
[\![\ e_1\ \texttt{xor}\ 1\ ]\!] & = & \neg\, [\![\ e_1\ ]\!] \\
[\![\ e_1\ \texttt{xor}\ e_2\ ]\!] & = & [\![\ e_1\ ]\!] \oplus [\![\ e_2\ ]\!] \\
[\![\ e_1\ \texttt{or}\ e_2\ ]\!] & = & [\![\ e_1\ ]\!] \vee [\![\ e_2\ ]\!] \\
[\![\ e_1\ \texttt{and}\ e_2\ ]\!] & = & [\![\ e_1\ ]\!] \wedge [\![\ e_2\ ]\!] \\
[\![\ \texttt{mux}\ e_1\ e_2\ e_3\ ]\!] & = & ([\![\ e_1\ ]\!] \wedge [\![\ e_2\ ]\!]) \vee (\neg\, [\![\ e_1\ ]\!] \wedge [\![\ e_3\ ]\!]) \\
[\![\ r \to v, e\ ]\!]_0 & = & [\![\ v\ ]\!] \\
[\![\ r \to v, e\ ]\!]_1 & = & [\![\ e\,[v/r]\ ]\!] \\
[\![\ r \to v, e\ ]\!]_k & = & [\![\ e\,[[\![\ r \to v, e\ ]\!]_{k-1}/r]\ ]\!] \\
[\![\ r_0 \to v_0, e_0; ...; r_i \to v_i, e_i\ ]\!]_0 & = & [\![\ v_0\ ]\!]; ...; [\![\ v_i\ ]\!] \\
[\![\ r_0 \to v_0, e_0; ...; r_i \to v_i, e_i\ ]\!]_1 & = & [\![\ e_0\,[v_0/r_0] ... [v_i/r_i]\ ]\!]; ...; [\![\ e_i\,[v_0/r_0] ... [v_i/r_i]\ ]\!] \\
[\![\ r_0 \to v_0, e_0; ...; r_i \to v_i, e_i\ ]\!]_k & = & [\![\ e_0\,[[\![\ r_0 \to v_0, e_0\ ]\!]_{k-1}/r_0] ... [[\![\ r_i \to v_i, e_i\ ]\!]_{k-1}/r_i]\ ]\!]; \\
 & & ...; [\![\ e_i\,[[\![\ r_0 \to v_0, e_0\ ]\!]_{k-1}/r_0] ... [[\![\ r_i \to v_i, e_i\ ]\!]_{k-1}/r_i]\ ]\!]
\end{array}
$$

Figure 2: Denotational Semantics of *miniRTL*, using boolean algebra. The notation $[\![\ e\,[v/r]\ ]\!]$ is used to denote basic name substitution of r by the value v in expression e, and $[\![\ e\ ]\!]_i$ is used to denote an expression whose behavior depends on the cycle at which we are interpreting it, e.g. $[\![\ e\ ]\!]_0$ means that e is interpreted at cycle 0, $[\![\ e\ ]\!]_1$ at cycle 1, etc...

[2]This follows from First Order Logic (FOL), the same decision was made in the design of the CIRCT Core IR.

3.3 Example

Now that we have defined the meaning of a circuit written in our language, we can show an example of a real design implemented using *miniRTL*, which is illustrated in Figure 3. This circuit implements a 1-bit accumulating adder, that handles overflows. This shows that we can already encore interesting circuits using only a single bit, without any external inputs. In the following sections, we will extend *miniRTL* to make our adder design even more useful.

```
A -> 0, C xor (a xor b)
B -> 1, B
C -> 0, (A and B) or (C and (A xor B))
```

Figure 3: Example of a simple 1-bit adder circuit that repeatedly adds 1 to a starting value and stores that value in one of the operands. A and B are the operands of our addition, and C is the carry bit. This implements: $A = C \oplus (A \oplus B)$ and $C = (A \wedge B) \vee (C \wedge (A \oplus B))$.

4 A Minimal RTL Language with Assertions (*assertRTL)*

Our proposed minimal RTL language can express various 1-bit designs. However, our goal is to use this language to support formally verifying designs using bounded model checking. In order to do so, we need to at least be able to verify some properties about our design. In this section, we will thus add assertions and additional logic to our language.

4.1 Syntax

```
Registers r
Value v         ::= 0 | 1
Order o         ::= skip | fail
Expression e    ::= e eq e | e xor e | e and e | e or e | mux e e e | v | r
Circuit C       ::= [] | C ; [r -> v, e] | C ; [assert e]
```

Figure 4: Syntax of the *assertRTL* language.

Figure 4 shows the syntax of our assertion enable language. This is simply an extension of *miniRTL* where now a program can not only be a register definition but also an assertion. In order to simplify the process of writing assertions, we added an equality operator. We chose to leave out more complex arithmetic operations from our language, as those would not be very useful in a 1-bit setting and would require being opinionated about various elements, e.g. how overflows are handled. We also added the concept of orders (`skip` and `fail`), which are statements that can be produced by an assertion. These impact how the design behaves during verification but are not proper expressions and do not produce any additional hardware.

4.2 Semantics

The semantics of most of *assertRTL* are identical to *miniRTL*, so we will only be defining the additional elements that were added, i.e. equality, assertions, and the behavior of `skip` and `fail`. We will be defining the behavior of `assert`, `skip`, and `fail` using small step operational semantics. The meaning of equality will be kept in denotational semantics.

$$[\![e_1 \texttt{ eq } e_2]\!] \quad = \quad \neg([\![e_1]\!] \oplus [\![e_2]\!])$$

$$\frac{i \in \mathbb{N} \quad [\![e]\!]_i = 0}{\texttt{assert } e \to_i \texttt{fail}} \qquad \frac{i \in \mathbb{N} \quad [\![e]\!]_i = 1}{\texttt{assert } e \to_i \texttt{skip}}$$

$$\frac{i \in \mathbb{N} \quad \texttt{assert } e \to_i \texttt{fail}}{C_1;\ \texttt{assert } e;\ C_2 \to_i \texttt{fail}} \qquad \frac{i \in \mathbb{N} \quad \texttt{assert } e \to_i \texttt{skip}}{C_1;\ \texttt{assert } e;\ C_2 \to_i C_1;\ C_2}$$

$$\textit{equiv}\ \frac{C \to_i C'}{[\![C]\!]_i = [\![C']\!]_i} \qquad \textit{fail}\ \frac{C \to_i \texttt{fail}}{[\![C]\!]_i = 0?}$$

Figure 5: Semantics of the additional elements introduced in *assertRTL*.

Figure 5 defines the semantics of the unique elements of *assertRTL*. Equality is defined using denotational semantics while assertions, being solely a verification construct that does not generate any additional hardware, can only be defined using operational semantics. The *equiv* and *fail* rules define the equivalence between our denotational and operational semantics. To define verification failure, we extend our denotation logic to boolean algebra with tests, following the same notion of tests defined in Kleene Algebra with Tests (KAT) [13]. These semantics also define how to simulate our designs for verification purposes, where $C \to_i C'$ means that a circuit C steps to an evaluated circuit C' after i cycles of simulation.

4.3 Example

To illustrate our new features, let's extend our adder circuit from Figure 3 to add a simple assertion that checks that A always alternates between 1 and 0.

```
A  -> 0, C xor (a xor b)
Ap -> A
B  -> 1, B
C  -> 0, (A and B) or (C and (A xor B))
assert A xor Ap
```

Figure 6: Extension of our *miniRTL* adder that asserts that the accumulator `A` toggles between cycles. This is done by recording the delayed value of `A` in `Ap` and then asserting that the two values are always different.

5 A Modular RTL Language (*modRTL*)

While adding assertions allowed us to check basic properties about our designs, the designs themselves remain very basic and have no re-usability of expressions. In order to improve the re-usability of things, we will extend *assertRTL* to support name bindings and modules (basically structural versions of functions).

5.1 Syntax

```
Registers r     Variables x
Value v         ::= 0 | 1
Order o         ::= skip | fail
Expression e    ::= e eq e | e xor e | e and e | e or e | mux e e e |
                        v | r | x | m (e,...,e) | x (e,...,e)
Module m        ::= mod(x, ...,x){b}
Statement s     ::= r -> v, e | x = e | x = m | assert e | m
Body b          ::= [s]* ; out e
Circuit c       ::= [s]*
```

Figure 7: Syntax of the *modRTL* language.

Figure 7 shows the syntax of the *modRTL* language. There is a lot more going on in this syntax than the two previous ones. First, we introduce the new concept of modules, which function similarly to multi-argument functions with a single return value. Given that modules can be bound to a variable name `x`, we need to define *instantiating*, i.e. calling, for both modules and variables. The body of a module is defined as an arbitrary series of statements followed by an output expression. Finally, we extract the definition of top-level statements and define a circuit as an arbitrary sequence of these statements.

5.2 Semantics

As with our previous extension, the semantics of *modRTL* build on top of *assertRTL* by adding modules, module instancing (i.e. calling a module with concrete arguments), and name bindings. The use of name bindings requires that we keep track of a binding context Γ. Many of these rules are inspired by one of the few other formalized hardware languages, Kôika [4].

$$read\frac{\Gamma[x] = e}{\Gamma \vdash [\![x]\!] = [\![e]\!]} \qquad bind\frac{\Gamma[x] = e}{\Gamma \vdash [\![x = e]\!] = \texttt{skip}}$$

$$bsubst\frac{\begin{array}{c} i \in \mathbb{N} \\ b = s_0; ...; s_j;\ \texttt{out}\ e \end{array}}{\Gamma \vdash [\![b\,[e_0/x_0] ... [e_k/x_k]]\!]_i = [\![s_0\,[e_0/x_0] ... [e_k/x_k]\,; ...; s_j\,[e_0/x_0] ... [e_k/x_k]\,; e\,[e_0/x_0] ... [e_k/x_k]]\!]_i}$$

$$body\frac{\begin{array}{c} i \in \mathbb{N} \\ b = s_0; ...; s_j;\ \texttt{out}\ e \end{array}}{\Gamma; s_0\,[e_0/x_0] ... [e_k/x_k]\,; ...; s_j\,[e_0/x_0] ... [e_k/x_k] \vdash [\![b\,[e_0/x_0] ... [e_k/x_k]]\!]_i = [\![e\,[e_0/x_0] ... [e_k/x_k]]\!]_i}$$

$$callm\frac{i \in \mathbb{N}}{\Gamma \vdash [\![\texttt{mod}(x_0, ..., x_k)\{b\}(e_0, ..., e_k)]\!]_i = [\![b\,[e_0/x_0] ... [e_k/x_k]]\!]_i}$$

$$callx\frac{\Gamma[x] = \texttt{mod}(x_0, ..., x_k)\{b\}}{\Gamma \vdash [\![x(e_0, ..., e_k)]\!]_i = [\![\texttt{mod}(x_0, ..., x_k)\{b\}(e_0, ..., e_k)]\!]_i}$$

Figure 8: Semantics of the additional components introduced in *modRTL*.

Figure 8 defines the denotational semantics of *modRTL*. Rule *bind* defines the behavior of name

bindings, *bsubst* and *body* define the meaning of substitution over the body of a modules as well as how to evaluate it, and *callm* and *callx* define the meaning of instantiation using the module directly or a variable bound to a module. All cases that are not explicitly defined in the semantics, e.g. calling a variable bound to an arbitrary expression, result in a compilation error.

5.3 Example

We can now extend our adder example to implement a 2-bit adder using modules. Given that our modules only support a single output (in order to avoid dealing with tuples and projections), we will need to define separate modules for computing the carry bit and the sum.

```
sum = mod (a_in, b_in, c_in) {
    axb = a_in xor b_in
    out c_in xor axb
}
carry = mod (a_in, b_in, c_in) {
    axb = a_in xor b_in
    anb = a_in and b_in
    out anb or (c_in and axb)
}
add2_0 = mod (a1, a0, b1, b0) {
    out sum(a0, b0, 0)
}
add2_1 = mod (a1, a0, b1, b0) {
    c_0 = carry(a0, b0, 0)
    out sum(a1, b1, c_0)
}
carry2 = mod (a1, a0, b1, b0) {
    carry0 = carry(a0, b0, 0)
    out carry(a1, b1, carry0)
}
bit0 = add2_0(0,1,0,1)
bit1 = add2_1(0,1,0,1)
overflow = carry2(0,1,0,1)

assert (bit0 eq 0) and (bit1 eq 1) and (overflow eq 0)
```

Figure 9: Implementation of a full 2-bit adder in *modRTL*. Each bit is computed separately, as well as the final carry bit (overflow). The final assertion tests our 2-bit adder by checking that $1+1=2$, or in binary `0b01 + 0b01 = 0b10` without overflow.

Figure 9 demonstrates the capabilities of *modRTL* by defining a full 2-bit adder. The addition of modules and bindings allows for a lot more flexibility, and reduces repetition, when writing out circuits. Given that this is still 1 bit logic, each bit of our 2-bit component must be computed separately, leading to a slightly larger amount of similar modules. However, this shows that our language can be used to implement practical designs and test their functionality.

6 A Modular RTL Language with Contracts (*dvRTL*)

For the final iteration of our language, we will be focusing on verification. While *modRTL* can describe designs in a modular way, our assertion remain quite simple and rooted in synthesizable logic, i.e. logic that is identical to one we use to describe physical hardware. Additionally, our specification power remains limited to writing assertions all over our design, making it a very manual process to check that inputs to a module instance are valid or that our module behaves the way we want it to. For these reasons, we will extend *modRTL* to add basic arithmetic expressions for writing our assertions, as well as the concept of contracts introduced in Section 2.2. In order to support the instantiation of a hardware contract, we will also be adding assumptions to our language.

6.1 Syntax

```
Registers r     Variables x
Value v         ::= 0 | 1
Order o         ::= skip | fail
Expression e    ::= e xor e | e and e | e or e | mux e e e |
                        v | r | x | m (e,...,e) | x (e,...,e)
Arithmetic a    ::= a impl a | a + a | a - a | a eq a |
                        a xor a | a and a | a or a | e
Contract h      ::= res | a
Module m        ::= mod(x, ...,x)[req a; ens a]{b} | mod(x, ...,x){b}
Statement s     ::= r -> v, e | x = e | x = m | assert a | assume a | m
Body b          ::= [s]* ; out e
Circuit c       ::= [s]*
```

Figure 10: Syntax of the *dvRTL* language.

Figure 10 defines the syntax of the *dvRTL* language, which is the final iteration of our minimal RTL language for verification. The main introductions here are an added precondition `req a` and postcondition `ens a` to our module definition. For simplicity, we are only allowing for a single precondition and postcondition to be defined per modules. We are still allowing for modules to be defined without a contract, which will impact how they are elaborated. We also extended our assertion language with basic arithmetic operations like addition and subtraction. Given that these do not directly generate hardware, they must only be used in verification contexts. We also add the keyword `res` to refer to the output of a module, such that we can reason about it easily in the contract without needing a specific binding for it.

6.2 Semantics

As with our previous extensions, the semantics of most of *dvRTL* are identical to *modRTL*, with the addition of contracts, assumptions and basic arithmetic operations.

Figure 11 defines the denotational semantics of our two arithmetic operations. These lower to 1-bit unsigned arithmetic and the two edge cases causing overflows and underflows were defined explicitly for clarity. We now must define how assumptions work, in this case we introduce a knowledge context A, which keeps track of all of the assumptions made in the circuit. Similar to our name bindings, assumptions are global as all statements are interpreted in parallel.

$$
\begin{array}{lcl}
[\![a_0 + a_1]\!] & = & [\![a_0]\!] + [\![a_1]\!] \\
[\![1 + 1]\!] & = & 0 \\
[\![a_0 - a_1]\!] & = & [\![a_0]\!] - [\![a_1]\!] \\
[\![0 - 1]\!] & = & 1 \\
[\![a_0 \texttt{ implies } a_1]\!] & = & \neg[\![a_0]\!] \vee [\![a_1]\!]
\end{array}
$$

Figure 11: Semantics of our arithmetic operations, where + and − function the same way as 1-bit unsigned addition and subtraction. The edge cases where defined explicitly for clarity.

$$
\textit{assume}\frac{a_0 \in A}{\Gamma; A \vdash [\![\texttt{assume } a_0]\!] = \texttt{skip}}
$$

$$
\textit{assert}\frac{\{a_0, .., a_k\} \in A}{\Gamma; A \vdash [\![\texttt{assert } a]\!] = [\![\texttt{assert } ((a_0 \texttt{ and...and } a_k) \texttt{ impl } a)]\!]}
$$

$$
\textit{definec}\frac{\begin{array}{c} i \in \mathbb{N} \\ a_r \in A \end{array}}{\Gamma; A \vdash [\![\texttt{mod}(x_0, ..., x_k)\,[\texttt{req } a_r; \texttt{ens } a_e]\,\{b\}]\!]_i = [\![\texttt{assume } a_r;\ b;\ \texttt{assert } a_e]\!]_i}
$$

$$
\textit{callc}\frac{\begin{array}{c} i \in \mathbb{N} \\ a_{esub} \in A \\ a_{rsub} = a_e\,[e_0/x_0] \ldots [e_k/x_k] \\ a_{esub} = a_e\,[e_0/x_0] \ldots [e_k/x_k]\,[e_{out}/\texttt{res}] \end{array}}{\Gamma; A \vdash [\![\texttt{mod}(x_0, ..., x_k)\,[\texttt{req } a_r; \texttt{ens } a_e]\,\{b\}(e_0, ..., e_k)]\!]_i = [\![\texttt{assert } a_{rsub};\ \texttt{assume } a_{esub}]\!]_i}
$$

Figure 12: Semantics of the additional elements of *dvRTL*. We define A as our global knowledge context which accumulates the set of assumed arithmetic expressions.

Figure 12 defines the denotational semantics of *dvRTL*. Given that we have introduced assumptions, we must track our knowledge context across all of our rules. Note that module contracts do no give any guarantees about the strength of the post conditions, only that the preconditions, along with the statements in the module body, are enough to prove the postconditions (following the *weakest precondition* techniques used in deductive verification).

6.3 Example

We can now extend our 2-bit adder example from Figure 9 to include more generalized verification using our contracts.

Figure 13 illustrates the use of contracts in the extension of our 2-bit adder example. Most of the preconditions are trivial, and we leave out a contract for `carry` as our assertion language is not powerful enough to specify `carry` without duplicating the implementation.

7 Compiling *dvRTL* for Verification

To close out our formalization, we propose a compilation of *dvRTL* down to BTOR2, a popular bounded model checking format. BTOR2 does not support any modularity of notion of contracts, and is at a similar abstraction as *assertRTL*, if add assumptions and our additional arithmetic

```
sum = mod (a_in, b_in, c_in) [
    req 1
    ens res eq (a_in + b_in + c_in)
]{
    axb = a_in xor b_in
    out c_in xor axb
}
carry = mod (a_in, b_in, c_in){
    axb = a_in xor b_in
    anb = a_in and b_in
    out anb or (c_in and axb)
}
add2_0 = mod (a1, a0, b1, b0) [
    req 1
    ens res eq (a0 + b0)
] {
    out sum(a0, b0, 0)
}
add2_1 = mod (a1, a0, b1, b0) [
    req 1
    ens res eq ((a0 and b0) + (a1 + b1))
]{
    c_0 = carry(a0, b0, 0)
    out sum(a1, b1, c_0)
}
carry2 = mod (a1, a0, b1, b0) [
    req 1
    ens res eq ((a0 and b0) + (a1 and b1))
]{
    carry0 = carry(a0, b0, 0)
    out carry(a1, b1, carry0)
}
bit0 = add2_0(0,1,0,1)
bit1 = add2_1(0,1,0,1)
overflow = carry2(0,1,0,1)

assert (bit0 eq 0) and (bit1 eq 1) and (overflow eq 0)
```

Figure 13: 2-bit adder implemented in *dvRTL*. In this specific case, we only have trivial preconditions, but in more complex designs, various other preconditions would be useful.

assertion language to it. We thus define *verifRTL* as an extension of *assertRTL* that contains the arithmetic expressions, assumptions, and name bindings of *dvRTL* as well as a way to define free variables. The exact syntax of *verifRTL* is defined in Figure 14. The compilation of *dvRTL* to BTOR2 will involve 2 steps: an elaboration down to our intermediate representation called *verifRTL*, and a compilation from *verifRTL* to BTOR2.

```
Registers r
Value v          ::= 0 | 1
Order o          ::= skip | fail
Expression e     ::= e xor e | e and e | e or e | mux e e e | v | r
Arithmetic a     ::= a impl a | a + a | a - a | a eq a |
                         a xor a | a and a | a or a | e
Statement s      ::= r -> v, e | x = a | assert a | assume a | in x
Circuit C        ::= [s]*
```

Figure 14: Syntax of the *verifRTL* language.

7.1 Elaborating *dvRTL* to *verifRTL*

Elaborating *dvRTL* is straightforward and involves inlining modules following the rules defining the semantics of *modRTL* (for non-contract modules) from Figure 8, and the rules defining the semantics of *dvRTL* (for contract modules) from Figure 12. We will only mention here the small subtleties that need to be cared for in order to correctly elaborate a *dvRTL* design. The name bindings in *verifRTL* function as Single-Static Assignment (SSA) names that are used to simplify the conversion to BTOR2. Therefore during elaboration, all nested expressions need to be separated into a series of SSA statements, as is done in Figure 18. Our *verifRTL* will thus be assumed to be written in SSA style during compilation, where each expression or arithmetic operation is on its own line. *verifRTL* allows for the definition of free variables using `in x`, as to allow for generic module tests to be correctly elaborated. All inputs from a module with a contract will be converted to `in x`, in order to generate the module test *checkmod*, described in Section 7.2. The special `res` expression referenced in postconditions will also be converted to a free variable.

7.2 An *Informal* Argument for Correctness

In order to verify our designs, we need to have some notion of what correctness guarantees our verification method is providing. The idea is to trade-of scalability for strength of guarantees. This means that choosing to write a contract for a modules allows the verification task to be more modular, making it simpler to verify large designs that have several instances of some of the modules. However, our elaboration follows the approach of focusing on guaranteeing a weakest precondition assumption, i.e. that the given preconditions are stronger than the weakest precondition required to verify the given postcondition. This means that we can only guarantee that the postcondition can be proven given the body of the design, but not that the postcondition itself is the strongest it can be. In more specific terms we simply follow the *modus ponens* rule to use our contract to generate a single proof for the module that allows every instance to be abstracted to simply checking that the preconditons hold for a given set of concrete inputs. Basically we use the following rule for a triple `[req p; ens q]{b}`:

$$\frac{\begin{array}{c}\textit{checkmod: } p \rightarrow (b \wedge q) \\ \textit{checkinst: } p\,[v_0/in_0] \ldots [v_k/in_k]\end{array}}{q\,[v_0/in_0] \ldots [v_k/in_k]}$$

7.3 Compiling *verifRTL* to BTOR2

The general form of a BTOR2 statement is: `<lid> <op> <sort> <operand_ids>...`, where `<lid>` refers to the line number and the operand identifiers refer to the line numbers at which the operands were defined. Registers are typically encoded using multiple statements: one to declare the regis-

ter `<lid> state <sort> <name>`, one to define the initial value `<lid> init <sort> <state_id> <val_id>`, and one to define the next state expression `<lid> next <sort> <state_id> <next_id>`. One last detail, is that assertion are encoded as "bad" states that should not be reached, therefore `assert` $a \equiv$ `bad` $\neg a$ (a more formal translation rule is given in Figure 15).

To compile *verifRTL* down to BTOR2, we will follow the following notation: $((s, id))$ produces the BTOR2 string that encodes the `s` statement at the current line id, while $((s))$.`lid` returns the last line number produced by the translation process of that particular statement, which will often be `id + 1`, but may be larger in cases like assertion or register translations. Given that *verifRTL* is in SSA form, all of our lines will begin with a name binding, which we will omit in our compilation rules for readability (i.e. $((x = a, id))$ will be written $((a, id))$). Every BTOR2 program starts with a series of *sort* declarations, and given that we only support 1-bit logic, we will start all compilation processes by emitting a top-level 1-bit bitvector sort as follows:

```
1 sort bitvector 1
```

$$
\begin{array}{lcl}
((0, \texttt{id})) & = & \texttt{<id> zero 1} \\
((1, \texttt{id})) & = & \texttt{<id> one 1} \\
((a_0\ \texttt{impl}\ a_1, \texttt{id})) & = & \texttt{<id> implies 1 } ((a_0)).\texttt{lid } ((a_1)).\texttt{lid} \\
((a_0 + a_1, \texttt{id})) & = & \texttt{<id> add 1 } ((a_0)).\texttt{lid } ((a_1)).\texttt{lid} \\
((a_0 - a_1, \texttt{id})) & = & \texttt{<id> sub 1 } ((a_0)).\texttt{lid } ((a_1)).\texttt{lid} \\
((a_0\ \texttt{eq}\ a_1, \texttt{id})) & = & \texttt{<id> eq 1 } ((a_0)).\texttt{lid } ((a_1)).\texttt{lid} \\
((a_0\ \texttt{xor}\ a_1, \texttt{id})) & = & \texttt{<id> xor 1 } ((a_0)).\texttt{lid } ((a_1)).\texttt{lid} \\
((a_0\ \texttt{and}\ a_1, \texttt{id})) & = & \texttt{<id> and 1 } ((a_0)).\texttt{lid } ((a_1)).\texttt{lid} \\
((a_0\ \texttt{or}\ a_1, \texttt{id})) & = & \texttt{<id> or 1 } ((a_0)).\texttt{lid } ((a_1)).\texttt{lid} \\
((\texttt{mux}\ e_0\ e_1\ e_2, \texttt{id})) & = & \texttt{<id> ite 1 } ((e_0)).\texttt{lid } ((e_1)).\texttt{lid } ((e_2)).\texttt{lid} \\
(((r \to v, e), \texttt{id}))_0 & = & \texttt{<id> state 1 r} \\
(((r \to v, e), \texttt{id}))_1 & = & \texttt{<id> init 1 } (((r \to v, e)))_0.\texttt{lid } ((v)).\texttt{lid} \\
(((r \to v, e), \texttt{id}))_2 & = & \texttt{<id> next 1 } (((r \to v, e)))_0.\texttt{lid } ((e)).\texttt{lid} \\
((\texttt{assert}\ a, \texttt{id}))_0 & = & \texttt{<id> not 1 } ((a)).\texttt{lid} \\
((\texttt{assert}\ a, \texttt{id}))_1 & = & \texttt{<id> bad } ((\texttt{assert}\ a))_0.\texttt{lid} \\
((\texttt{assert}\ 1, \texttt{id})) & = & \emptyset \\
((\texttt{assume}\ a, \texttt{id})) & = & \texttt{<id> constraint } ((a)).\texttt{lid} \\
((\texttt{assume}\ 1, \texttt{id})) & = & \emptyset \\
((\texttt{in}\ x, \texttt{id})) & = & \texttt{<id> input 1 } x
\end{array}
$$

Figure 15: Compilation rules from an SSA encoded *verifRTL* to BTOR2. A couple of basic optimizations such as skipping trivial assertions and assumptions have been added to simplify the translation.

Figure 15 shows the full set of compilation rules from *verifRTL* to BTOR2. Note that since *verifRTL* is specifically encoded as SSA statements, we do not have to deal with nested expressions. The compilation is done in parallel where lids are resolved once they are made available. Given that BTOR2 is strictly ordered (i.e. does not allow def-after-use like our language), we statements need to be serialized in a DFS order. These rules define the compilation of a single statement. A pseudocode description of the specific implementation details required to functionally compile an entire circuit is given in Figure 16.

```
def compile(c) :=
    def rec(cur_s, id, BTOR2, Compiled) :=
        if c.empty then
            return BTOR2
        end if
        s = next_dfs(c, cur_s, Compiled)
        (inst, lid) = ((s, id))
        rec(s, lid + 1, BTOR2 ++ inst, Compiled ++ s)
    end rec
    sort = "1 sort bitvector 1"
    return rec(None, 2, [sort], [])
end compile
```

Figure 16: Pseudocode describing the implementation of a *verifRTL* to BTOR2 compiler. C is encoded as a dataflow tree where the head is the final operation (typically an assertion) and the children are its operands. The `next_dfs` takes as input the tree, the current node, and a list of the explored elements and returns the next node to be explored in DFS order.

7.4 End-to-End Verification Example

For this last part we will show a quick end-to-end example of our language. In order to not produce an absurdly large final example we will be using a simple 1-bit adder to illustrate the end-to-end process (although it doesn't benefit as much from modularity as the two bit adder). Our final

```
add = mod(a, b) [
    req 1
    ens res eq (a + b)
] {
    out a xor b
}
carry = mod(a, b) {
    axb = a xor b
    anb = a and b
    out anb or axb
}
s0 = add (1, 1)
s1 = carry (1, 1)
assert s1 and s0
```

Figure 17: Example of a 1-bit adder implemented in *dvRTL*.

result, shown in Figure 19, is now in a formatted that is widely supported in open-source bounded model checkers such as `btormc` [16], which can be used to perform our verification task.

8 Conclusion

In this work, we proposed and formalized a hardware language specifically designed for formal verification. We did so by building up our feature set from a minimal language called *miniRTL*, through 2 other extensions *assertRTL* and *modRTL* to finally end up will a modular 1-bit hardware

```
in a
in b
assume 1
out0 = a xor b
prec0 = a + b
assert out0 eq prec0

assert 1
cont0 = 1 + 1
in s0
assume s0 eq cont0

axb = 1 xor 1
anb = 1 and 1
s1 = anb or axb

assert s1 and s0
```

Figure 18: Example from Figure 17 elaborated to *verifRTL*.

language that supports assertions, assumptions, and contracts, called *dvRTL*. We illustrated the flexibility of this language by designing a 2-bit full adder with contracts. We then formalized a compilation from *dvRTL* down to BTOR2 using an intermediate representation similar to *assertRTL*. We then argued for the correctness of our verification process.

Throughout this work, I learned how to formalized and design a domain-specific language from scratch using denotational semantics. While the language is currently very simple, it was designed to be a subset the the CIRCT Core IR, in the hopes that this project can be used as a launching point to fully formalize the entirety of the Core IR. To do so, future work would be to first extend the language to include multi-bit designs by adding a simple type-system that keeps track of bit-widths and enforces the correct handling of overflows and underflows. This would also allow us to extend our language with additional bit-manipulation expressions that are present in the Core IR. Afterwards, our current implicit clock and reset signals would have to be made explicit, allowing for multi-clock designs to be implemented. Finally, our assertion language would have to extended to support temporal logics, as this would allow us to specify sequential designs. The main difficulty would then be defining a lowering from this added temporal logic to the BTOR2 format that only supports First-Order Logic and state-transition systems, but not modal logics like LTL.

With this project we have hopefully laid the foundation to fully formalize the core IR and hopefully one day create a true formalized foundation for all of hardware compilation.

```
1 sort bitvector 1
2 input 1 a
3 input 1 b
4 xor 1 2 3
5 add 1 2 3
6 eq 1 4 5
7 not 1 6
8 bad 7

9 const 1 1
10 add 1 9 9
11 input 1 s0
12 eq 1 11 10
13 constraint 12

14 xor 1 9 9
15 and 1 9 9
16 or 1 14 15

17 and 1 16 11
18 not 1 17
19 bad 18
```

Figure 19: Example from Figure 18 compiled to BTOR2.

References

[1] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, pages 1–1354, 2024.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery.

[3] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. Creating an agile hardware design flow. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20. IEEE Press, 2020.

[4] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The essence of bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 243–257, New York, NY, USA, 2020. Association for Computing Machinery.

[5] chipsalliance. Flexible intermediate representation for rtl.

[6] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Trans. Parallel Distrib. Syst.*, 32(5):1014–1029, May 2021.

[7] Amelia Dobis. Formal verification of hardware using mlir. Master thesis, ETH Zurich, Zurich, 2024.

[8] Amelia Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. Verification of chisel hardware designs with chiselverify. *Microprocessors and Microsystems*, 96:104737, 2023.

[9] Amelia Dobis, Tjark Petersen, and Martin Schoeberl. Towards functional coverage-driven fuzzing for chisel designs. In *Workshop on Open-Source EDA Technology (WOSET 2021)*, 2021.

[10] Amelia Dobis, Fabian Schuiki, and Mae Milano. Incremental conversion of sva properties to synthesizable hardware. In *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'25)*, 2025.

[11] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. Mlir as hardware compiler infrastructure.

[12] Martin Erhart, Fabian Schuiki, Zachary Yedidia, Bea Healy, and Tobias Grosser. Arcilator: Fast and cycle-accurate hardware simulation in CIRCT. `https://llvm.org/devmtg/2023-10/slides/techtalks/Erhart-Arcilator-FastAndCycleAccurateHardwareSimulationInCIRCT.pdf`.

[13] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, May 1997.

[14] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 2–14. IEEE Press, 2021.

[15] Aina Niemetz and Mathias Preiner. Bitwuzla. In *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II*, page 3–17, Berlin, Heidelberg, 2023. Springer-Verlag.

[16] Aina Niemetz, Mathias Preiner, Claire Wolf, and Armin Biere. Btor2 , btormc and boolector 3.0. In *International Conference on Computer Aided Verification*, 2018.

[17] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 804–817, New York, NY, USA, 2021. Association for Computing Machinery.

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

[19] Morten Borup Petersen. A dynamically scheduled hls flow in mlir, jan 2022.

[20] Lenny Truong and Pat Hanrahan. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, 2019.

[21] Christian Ulmann. Multi-level rewriting for stream processing to rtl compilation. Master thesis, ETH Zurich, Zurich, 2022.

[22] Mike Urbach and Morten B Petersen. HLS from PyTorch to System Verilog with MLIR and CIRCT. 2022.

[23] whitequark. amaranth. `https://github.com/amaranth-lang/amaranth`, 2022.

[24] Claire Wolf. Yosys open synthesis suite. `https://yosyshq.net/yosys/`.