(System)Verilog to Chisel Translation for Faster Hardware Design

Jean Bruant, Pierre-Henri Horrein, Olivier Muller, Tristan Groleat, Frédéric Pétrot

(System)Verilog to Chisel Translation for Faster Hardware Design

Jean Bruant[*,†], Pierre-Henri Horrein[‡], Olivier Muller[†], Tristan Groléat[§] and Frédéric Pétrot[†]
OVHcloud, [*]Paris, [‡]Lyon, [§]Brest, France
[†]Univ. Grenoble Alpes, CNRS, Grenoble INP[1], TIMA, Grenoble, France

***Abstract*—Bringing agility to hardware developments has been a long-running goal for hardware communities struggling with limitations of current hardware description languages such as (System)Verilog or VHDL. The numerous recent Hardware Construction Languages such as Chisel are providing enhanced ways to design complex hardware architectures with notable academic and industrial successes. While the latter environments are now mature and perfectly suited for brand new projects, migrating partially or entirely existing Verilog code-base proves to be a challenging and very time-consuming process. Successful migrations need to be able to leverage finely tuned existing hardware descriptions as a basis to build complex systems through simple iterations. This article introduces *sv2chisel*, an open-source automated (System)Verilog to Chisel translator as entry point for this iterative migration processes. Our tool achieved the proper translation, with on-par resource usage of a real-world production FPGA design at OVHcloud as well as two independent open-source Verilog projects: a MIPS core and the size-optimized 32-bit RISC-V core *PicoRV32*.**

***Index Terms*—Hardware description languages, Agile development, EDA tool, Source code translation**

I. Introduction

VHDL and Verilog have reigned unchallenged for two decades as reference hardware description languages. A first breach was opened by the introduction of SystemVerilog in the early 2000's, rapidly followed by higher abstraction level proposals allowing more efficient hardware developments [1] [2]. Despite embedding many useful modern programming paradigms, SystemVerilog advanced concepts such as object-oriented programming are indeed partially supported in simulation context but have not been integrated into main synthesis tools.

With the end of Dennard scaling leading to an increasing demand for specialized hardware, the adoption of these languages has risen, although mainly in niche areas. These Hardware Construction Languages (HCL) can be seen as domain specific languages that leverage the power of high-level languages like Python, Scala, etc, to build efficient hardware generators that produce, in the end, synthesizable VHDL or Verilog code.

In this work, we focus on the Chisel [3] HCL, which is quite mature and has been proven successful in the development of complex circuits, notably *rocket-chip*, the original RISC-V core generator [4] and Google Edge TPU (Tensor Processing Unit) [5]. Chisel was initially designed to gain agility in RISC-V microprocessors tape-out process [6]. Despite this initial target, we also successfully use it into our production FPGA-based network functions at OVHcloud.

Chisel introduces many features and concepts intended to improve hardware design efficiency which are especially useful for the design of complex IPs and in large projects. However most existing large projects are made of thousands of lines of (System)Verilog or VHDL code. Similarly, complex pieces of hardware are often reused in larger projects along with many dependencies. In order to leverage the power of Chisel on such legacy code-bases, the first step is to translate the existing HDL into Chisel. When performed manually, this process is long, tedious and error-prone.

To address this issue, we propose in this paper to translate legacy (System)Verilog projects into Chisel ones, ready to be inserted within a larger Chisel hierarchy, to undergo deep refactoring or to be integrated with existing Chisel generators. Note that we limit ourselves to the synthesizable subset of SystemVerilog that is in practice a limited extension of Verilog [7]. We believe this will be beneficial to the hardware designers by giving them higher level constructs to refactor the entire code after translation, and limits the burden of creating signal level interfaces for instantiation, which takes a lot of time but adds nothing to the design. The source code of our translator sv2chisel is available on GitHub [8].

These languages aim at specifying the same hardware concepts but are based on different programming paradigms: (System)Verilog relies on an event-based semantic to depict underlying hardware while Chisel is based on explicit hardware objects assembled through object-oriented and functional generators. In addition Chisel is strongly typed while (System)Verilog provides very few different types. Performing the translation from (System)Verilog to Chisel thus requires a sequence of analysis to perform the appropriate abstraction. This includes *type analysis*, to infer the proper high-level types from possibly less structured information, and *data-flow and control-flow analysis* to abstract the expressions, functions and their parameters, loops and conditionals, etc.

Our work does not intend to abstract the original code into a high-level refactored functional object-oriented code. The process is thus more straightforward than, for example, decompilation of optimized assembly into C [9], because both languages have control structures and share many constructs such as function and module hierarchy. It is however not trivial, as we shall explain in the following sections.

II. Related Work

Translating algorithms or structured data from one representation to another is a recurrent task in most computer-aided fields. Simplest cases can be solved through stateless processing with tools such as `awk` or `perl`. However when control flow, data flow or semantic issues become more complex, advanced analysis and transformations on an intermediate representation (IR) are required. Due to the substantial differences between (System)Verilog and Chisel, our translator follows this second scheme.

A first work exhibiting similarities, as (System)Verilog is of quite lower abstraction than Chisel, is related to decompilation [9]. The principle here is to abstract assembly code into structured C, to target other architectures, but mainly to build a reusable, understandable code, to ease maintenance or reverse engineer algorithms. The authors develop data and control flow analyses to build C statements from abstract sequential (non-vliw) optimized assembly. The former recovers high-level language expressions while the latter captures control statements (loops, conditionals). They traverse dependency graphs and either use generic patterns to recognize the control structures, or study the registers assignments to build the higher-level expressions. They also perform simple data type recovery. In our case, control statement and assignments are in general sufficiently similar for not being an issue. However, registers identification and types recovery is at the core of our work.

An other work that relates to ours is [10] that focuses on the discovery of memorizing elements in VHDL. This issue was very important before the standardization of the descriptions for the hardware synthesis tools (IEEE Std 1076.6-1999), as the simulation semantic of the language made this discovery particularly tricky. The authors build a Petri Net that represents the behavior of the circuit. Using semantic preserving reduction techniques, they achieve a minimal form from which they extract a set of equations. They perform a formal analysis on all cyclic symbol assignments from which they derive if the circuit is a flip-flop, a latch, or a multiplexer, and the conditions to command it. Given the fact that we assume synthesizable (System)Verilog as input, this kind of analysis is useful only to determine how to compute the reset, clock and write enable signals.

Besides the work of [11] proposes a FPGA-specific synthesis optimization consisting of removing useless reset subcircuits in favor of FPGA power-on resets. It notably points out that inferring reset signals from the Verilog source code or at the equivalent Abstract Syntax Tree (AST) level is a complex tasks due to the numerous available ways of describing a reset [12]. To solve this issue, they leverage the synthesized netlist to implement a powerful reset detection algorithm. As we are not in a synthesis context and that reset inference is not crucial for the translation, we decided to infer only some well known reset patterns and translate transparently more advanced patterns.

Finally, we base our hardware Intermediate Representation on the work done in the FIRRTL compiler [13]. It has been a very valuable starting point – most notably for expression representation, above which we introduced all the generative subset of Verilog.

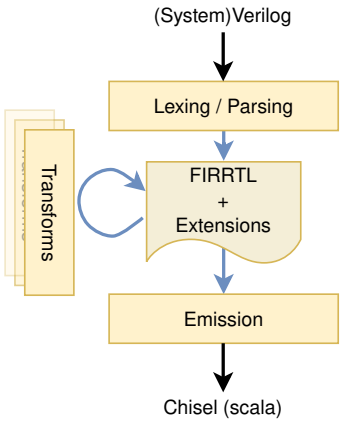

Fig. 1. Main processing steps of sv2chisel

III. Translation

A. sv2chisel Translator Structure

Figure 1 presents the structure of our translator with the classical stages of any translator or compiler. We parse a synthesizable (System)Verilog file using ANTLR 4 [14], which builds an abstract syntax tree that we eventually map on our custom IR. It is based on FIRRTL which has been extended most notably to support Verilog generative statements and syntactic-sugars requiring a few transformations to be mapped to Chisel constructs. Four kinds of analysis and transformations, main contributions of this work, are then performed on this IR: 1) clock inference, 2) reset inference, 3) types inference, 4) (System)Verilog syntactic-sugar translation.

Finally, the emitter outputs Chisel code, relocating comments and retaining original layout as much as possible. To enable the comments relocation, every node of the IR retains its initial position in the token stream of the ANTLR lexer. Transformations then have to carefully take these indexes into account whenever inserting or removing nodes, in order not to disturb comments relocation To achieve the final relocation, the emission goes through two steps. First, every IR node is converted to its Chisel equivalent string and the IR is flattened as a stream of those Chisel strings, each associated to its position in the original (System)Verilog token stream. Then, this stream is merged, based on the token indexes, with the original token stream which retains comments and spaces in special sub-streams.

B. Main Transformations

1) Clock Inference: Clock inference is a crucial part in the success of a proper translation from (System)Verilog to Chisel. As a direct consequence of (System)Verilog event-driven paradigm, the distinction between wires and registers does not

come from their declaration but from their assignments either in a clocked process or as a continuous assignment outside a clocked process.

```
module clock_example(
    input clock,
    input rst,
    input i,
    output o_w,
    output o_r
  );
  /* event-driven behavioral description of
   * register r with reset value '0 */
  logic r;
  always @(posedge clock) begin
    if (rst) begin
      r <= '0;
    end else begin
      r <= i;
    end
  end
  /* behavioral description of a wire */
  logic w;
  assign w = i;
  /* output connections */
  assign o_r = r;
  assign o_w = w;
endmodule
```

Listing 1: Verilog for Wire and Synchronous-reset-register

Listing 1 illustrates how signals `w` and `r` are to be described in the standard synthesizable (System)Verilog such that a wire is inferred for `w` and a register for `r`.

As a side note, we thoughtfully chose the use of **`logic`** keyword in our Verilog example for both declaration of `w` and `r`, instead of the misleading **`reg`** or **`wire`** keywords. Despite providing those keywords (System)Verilog does not enforce any restriction in their usage, consequently a register can be inferred from a signal declared as **`wire`** but then sequentially assigned and conversely a wire can be inferred from a signal declared as **`reg`** but then combinationally assigned. Some tools raise warning on such misusage however the rule remains the same for all of them: behavioral inference is preferred over user **`wire`** or **`reg`** hint.

Whereas wire and register are inferred from signal assignment behavior in Verilog, Chisel provides concrete objects **`Reg`** and **`Wire`** for signal declaration. These objects are self-sufficient when it comes to assignments: both **`Reg`** and **`Wire`** are assigned with the same operator, in the same context as shown on Listing 2.

To infer the main clock of a module, sv2chisel walks the IR a first time, looking for **`ClockRegion`** blocks corresponding to (System)Verilog **`always @(posedge clock)`** blocks. For each assignment inside such a block, left hand side references are recorded as assigned within a process clocked by `clock`. The declaration of theses references will then be converted as a **`Reg`** object during a second IR walk that will also remove **`ClockRegion`** blocks in favor of simple collections of statements.

```
class ImplicitClockOnly() extends MultiIOModule {
  // implicit input clock, connected to all `Reg`
  val rst = IO( Input  ( Bool() ))
  val i   = IO( Input  ( Bool() ))
  val o_w = IO( Output ( Bool() ))
  val o_r = IO( Output ( Bool() ))
  /* declaration of a register object */
  val r = Reg(Bool()) // type only
  when(rst){ // explicit synchronous reset
    r := false.B
  } .otherwise {
    r := i // sequential assignment @(clock)
  }
  /* declaration of a wire object */
  val w = Wire(Bool())
  w := i // combinational assignment
  /* output connections */
  o_w := w
  o_r := r
}
```

Listing 2: Translated Chisel for Listing 1 with clock inference

Clock might also be inferred from submodules instantiations whenever the actual module declaration can be processed recursively by sv2chisel.

At the end of this transformation, zero, one or several clocks might have been discovered. When zero or one clocks are discovered, the clock might be fully implicit within the module as Listing 2 shows. When several distinct clocks have been inferred, no implicit clock is used for the module and an explicit clock area must surround the register declarations of a same clock region, leveraging Chisel syntax `withClockAndReset` as Listing 3 introduces.[1]

This syntax might also be required to deal with some constrained board clock and reset naming–for example at design top level. Listing 3 illustrates this syntax to explicit the implicit constructs provided by **`MultiIOModule`** in the previous examples.

```
class ExplicitClockExample() extends RawModule {
  val reset = IO(Input(Reset()))
  val clock = IO(Output(Clock()))
  // inputs as shown Listing 2

  withClockAndReset(clock, reset){
    // statements as shown Listing 2
  }
}
```

Listing 3: Explicit clock and reset area in Chisel

2) Reset Inference: Several kinds of reset methods can be found in Verilog descriptions:

- Synchronous Reset
- Asynchronous Reset
- FPGA Power-on Reset
- Simulation **`initial`** Reset

[1]Multi-clock modules support is a work in progress at the time of writing.

Synchronous resets are hard to gracefully infer from a design as they might be hidden in deep and intricate **if/else** hierarchy [11]. For the simplest and most common Verilog pattern we perform synchronous reset inference, leveraging **RegInit** object as shown Listing 4. In more complex cases, the code is translated literally, keeping the conditional assignment tree as shown Listing 2. This is less idiomatic but still perfectly acceptable.

```
class ImplicitClockReset() extends MultiIOModule {
  // implicit inputs clock & reset
  val i   = IO( Input  ( Bool() ))
  val o_w = IO( Output ( Bool() ))
  val o_r = IO( Output ( Bool() ))
  /* declaration of a register object with reset */
  val r = RegInit(false.B) // reset value and type
  r := i // sequential assignment @(clock)
  /* declaration of a wire object */
  val w = Wire(Bool())
  w := i // combinational assignment
  /* output connections */
  o_w := w
  o_r := r
}
```

Listing 4: Translated Chisel for Listing 1 with reset inference

On the other hand, asynchronous resets are easily understood with a method equivalent to the one used for clock inference (see III-B1), based on the Verilog **always** blocks events `@(posedge(rst))` or `@(negedge(rst))`.

Leveraging power-on resets in a FPGA design can save a lot of hardware resources used by reset trees. This kind of reset is transparently associated to the signal declaration in Verilog and can hence be straightforwardly associated to Chisel **RegInit** declaration.

(System)Verilog **initial** keyword is part of the non-synthesizable subset, intended for simulation usage only and hence unsupported by our translator.

At the time of writing the sv2chisel support for mixed reset methods within the same design is currently a work in progress.

3) Types Inferences: While (System)Verilog requires explicit clocks and reset signal, it is very permissive towards data types: there is no typing or explicit cast in most cases. As a base principle every signal is an array of bits and if it needs to be used in an arithmetic computation, it is automatically cast as an unsigned, unless explicitly specified by the designer with the `$signed` cast.

This is very different in Chisel which comes embedded in the strongly typed language Scala. Chisel has several distinct basic hardware types such as arrays of hardware booleans (**Vec**[**Bool**]) and hardware (un)signed integers (**UInt** and **SInt**).

While both Chisel **Vec**[**Bool**] and **UInt** types would be a valid translation of any (System)Verilog signal, Chisel enforces very constraining rules for the usage of these distinct types. Chisel's **Vec**[**Bool**] are meant for individual bit manipulations and do not provide arithmetic operations. On the other hand **UInt** are intended for such arithmetic operations and are to be considered as a whole, they hence do not support subrange or subindex assignments.

To provide the most sensible translation – or at least the less verbose – sv2chisel counts the usage of each reference in arithmetic or bit operations to choose the proper declaration types.

Then for each expression, appropriate casting is inserted wherever necessary to guarantee type consistency.

```
module type_example #(param en)(
    input  clock,
    input  reset,
    output [31:0] counter,
    output [3:0]  sign
  );
  logic [31:0] cnt;
  always @ (posedge clock) begin
    cnt <= &cnt ? sign : cnt + 1;
  end
  assign counter = cnt;
  for( i=0 ; i < 4 ; i++) begin
    assign sign[i] = en ? ^cnt[(i+1)*8 - 1:i*8] : '0;
  end
endmodule
```

Listing 5: Untyped Verilog Input

Listing 5 shows a simple Verilog snippet illustrating the absence of typing: both `sign` and `cnt` signals are declared as packed arrays of bits. However while `cnt` is used in an arithmetic expression, `sign` is assigned one bit at a time. Listing 6 is the result of sv2chisel translation for Listing 5 and illustrates type inference of the signals with respect to their usage: `cnt` is now declared as a **UInt** while `sign` is declared as **Vec**[**Bool**]. Moreover, as `sign` is declared as **Vec**[**Bool**] but used to assign the **UInt** signal `cnt`, an explicit `asUInt` cast is inserted accordingly.

```
class TypeExample(en: Int) extends MultiIOModule {
  // implicit inputs clock & reset
  val counter = IO( Output ( UInt(32.W)    ))
  val sign    = IO( Output ( Vec(4,Bool()) ))

  val cnt = Reg(UInt(32.W))
  cnt := Mux(cnt.andR =/= 0.U, sign.asUInt, cnt + 1.U)
  counter := cnt

  for(i <- 0 until 4){
    sign(i) := if(en != 0) cnt((i+1)*8,i*8).xorR
               else        false.B
  }
}
```

Listing 6: Raw Chisel Translation of Listing 5

Although sv2chisel is designed to produce ready-to-use code, resulting code is mainly intended to serve as a transition step between (System)Verilog and handcrafted Chisel code, after manual inspection and refactoring. Listing 7 illustrates a small code refactoring, using a more Chisel-ish way of

expressing the same computation. Instead of a not so obvious iteration on subranges, one can do a simple cast of the 32 bits counter `cnt` to the equivalent array of 4 bytes. Then Scala functional paradigm can be leveraged to apply a xor reduction on every byte thanks to the `map` function. The special object **`Zeroes`** is a custom object that is equivalent to `O.U.asTypeOf(sign)` here.

```
class TypeExample(en: Boolean) extends MultiIOModule {
  /* see statement above */

  val bytes = cnt.asTypeOf(Vec(4, UInt(8.W)))
  sign := if(en) VecInit(bytes.map(_.xorR)) else Zeroes
}
```

Listing 7: Chisel for Listing 5 after manual refactoring

This kind of refactoring which makes the code clearer cannot be inferred by the tool but greatly helps with code maintainability. It confirms that both automated translation and manual expertise are essential to produce a finely-tuned piece of hardware. The former makes the conversion task realistic even for large code base while the latter ensures simplicity and readability of resulting code.

4) Syntactic Sugar Removal: (System)Verilog is also very permissive with connections of signals with mismatching widths. Implicitly wherever required it will automatically infer left-padding, with sign extension for signed and with zeroes otherwise. This implicit padding is then widely used as a feature, enabling some concise syntactic-sugars.

For example, SystemVerilog *bit patterns* `'0` and `'1` are such context-dependent width, equivalent to a value of inferred width with all bits set to 0, respectively all bits set to 1.

```
wire [31:0] w;
assign w = '0;
assign w = '1;
assign w = '{4{8b'010111}};
```

When translating to Chisel, actual width is required for all ones bit pattern as follow :

```
val w = Wire(UInt(32.W));
w := 0.U
w := ~0.U(32.W)
w := VecInit.tabulate(4)(_ => "b010111".U(8.W))
```

A second example of (System)Verilog concision is the left-hand-side assignment on a concatenation, which allows straightforward expression of bit unpacking.

```
wire [15:0] a;
wire [7:0]  b;
wire        op;
wire [15:0] sum;
assign {op, b, a} = signal;
assign sum = op ? a + b : a - b
```

Chisel does not offer such a syntax out of the box, however casting the right-hand-side into a **`Bundle`** does the trick as shown below:

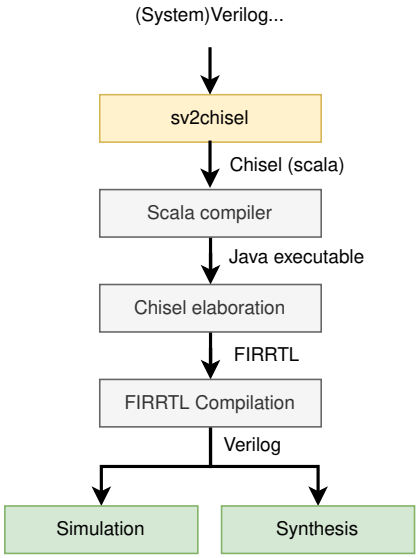

Fig. 2. Validation process

```
val bdl = Wire(new Bundle {
  val a   = UInt(16.W)
  val b   = UInt(8.W)
  val op  = Bool()
})
val sum = UInt(16.W)
bdl := signal.asTypeOf(bdl)
sum := Mux(bdl.op, bdl.a + bdl.b, bdl.a - bdl.b)
```

Translating (System)Verilog syntactic-sugars into Chisel code is not always as concise as the original code but thanks to the previously introduced type inference system, correct translations are achievable at the cost of a few additional lines. These verbose translations are then good starting points for manual code refactoring, as the power of Scala and Chisel constructs bring functional and object-oriented refactoring opportunities.

IV. Experiments

A. Methodology

Figure 2 illustrates all the steps taken to validate a translated design from its original description to the synthesized design.

The first step is to run sv2chisel tool to translate the original (System)Verilog into Chisel. The tool revealed itself to be a powerful stateful linter, catching wrongly declared and undeclared signals. Some manual adjustment might hence be required on the original description to fix caught issues and to overcome the current limitations of the tool as detailed in Section V.

The resulting Scala must then go through the entire Chisel generation flow, each step coming with its own correctness checks:

1) Scala compilation catches types inconsistencies
2) Chisel elaboration complains on mismatching widths

3) FIRRTL compilation stops on uninitialized references and detects combinational loops

While the two first steps are expected to pass flawlessly as long as sv2chisel does not complain during translation, FIRRTL compilation step raises errors that might only be manually corrected or ignored depending on the intended behavior of the result. Unexpected combinational loops combined with warnings of sv2chisel about unsupported blocking assignments might indicate an inaccurate translation and require further inspection of the original source code. Regarding uninitialized references, among classical examples stand big arrays, only half-connected, used to implement binary reduction operations. Such partial initialization can easily be fixed by adding the explicit default connections either in the original source or in the translated one. However a rather simple manual refactoring operation could leverage Chisel power to implement such reduction operation in a functional way while getting rid of this partially initialized arrays.

To validate the resulting Verilog and hence the correctness of the translation, we integrate it into the existing test system, leveraging usual simulators.

Last but not least, we compare the resource usage after synthesis on FPGA.

B. Results

We successfully ran our tool on three different projects to guarantee a consistent coverage of usual Verilog synthesizable constructs: 1) An internal module, currently used in production at OVHcloud, 2) A simple MIPS core implementation [15], functional but not actually suited for synthesis, 3) The size-optimized RISC-V Core *PicoRV32* [16].

Translated code for our internal module and the MIPS core passed their respective tests with minor manual modifications of the input source code. To pass PicoRV32 RISC-V core tests, some structural code refactoring was necessary to translate the concept of variable and their blocking assignments, as this feature does not find a straightforward Chisel equivalent.

Table below shows a resource usage comparison between original and translated versions for the two synthesizable projects. The SimpleCPU MIPS Core is indeed not intended for synthesis, which results in similar but inconsistent resources evaluation for both versions.

		Verilog	Chisel
OVHcloud Module	LUTs	1681	+7
	FFs	2733	=
	BRAMs	0	=
	DSPs	0	=
PicoRV32	LUTs	2349	+27
	FFs	1276	=
	BRAMs	0	=
	DSPs	0	=

Reported resource usage after synthesis of the translated Chisel designs are on-par with resource usage reported for their respective original (System)Verilog implementation. The 27 additional LUTs for the Chisel PicoRV32 are inferred in a quickly refactored submodule which is relying on Verilog's blocking assignments in its original implementation. Moreover as FIRRTL compilation to Verilog is flattening complex structures and expressions, some slight differences between the two resource counts can be expected.

V. Future Works

The development of sv2chisel followed a test-driven design methodology, starting with a reduced IR and limited syntactic-sugars support that were sufficient for our first internal SystemVerilog module. Applying the tool on external open-source examples ensured to enlarge the diversity of syntax and concepts properly translated up to a decent subset of (System)Verilog constructs. Each supported input construct comes with a corresponding unit-test while the presented real world examples serve as integration tests to validate the correctness of the translation. As a next step, further semantic analysis of Verilog and Chisel could be leveraged to formally design and prove the correctness of the transformation rules. At the time of writing our tool empirically covers the (System)Verilog synthesizable subset except for compiler directives (Verilog pre-processor) and blocking assignments.

Among the manual refactoring still required the input files, some are due to currently missing concepts in Chisel while others could be supported and properly translated without designer's help. Such currently under development features include the support of multi-resets and multi-clocks modules, synthesizable functions and partial pre-processor construct integration within the IR.

Some other translation issues reflect currently missing concepts in Chisel. The first issue is the support of semi-formal constructs equivalent to SystemVerilog `assert` or `property` directives, which is currently a work in progress, already integrated in FIRRTL backend but yet to be released in Chisel front-end. The second issue is the concept of (System)Verilog variables associated with blocking assignments for which EDA tools automatically infer multiple intermediate signals. Such a concept could actually be integrated into Chisel but this is rather unlikely as recursive functions or other functional constructs can be leveraged to implement the same hardware architectures.

Last but not least, we are currently investigating the feasibility of extending language support to VHDL that shares many concepts with (System)Verilog while being very strongly typed, like Chisel. Other HCLs translation targets, such as MyHDL [17], migen [18] or SpinalHDL [19], could also be considered to enable hardware designer to pick the language that fit the best their needs for a given project, as it can been observed for software languages.

VI. Conclusion

The power of Hardware Construction Languages is promising but their adoption remains for now limited to new projects

as the cohabitation between HCLs and usual HDLs requires substantial engineering work.

In this paper we introduced sv2chisel, a synthesizable (System)Verilog to Chisel translator. Producing Chisel code close to a word-for-word translation of the original source, this tool is intended as a first step in migrating valuable legacy Verilog code-bases into Chisel.

Although this tool will not fully replace the fine manual analysis required to achieve a correct translation in the most advanced cases, we showed through three pre-existing real-world Verilog examples that our tool can produce a very decent translation draft. These drafts are then intended to be manually refactored to leverage the power of HCLs, generalizing the use of object-oriented and functional constructs. We believe the adoption of HCLs within existing HDL projects could greatly benefit from this translation approach, enabling a smooth and agile migration of existing code-bases and rapid new HCLs system prototyping based on finely-tuned existing hardware libraries.

REFERENCES

[1] D. F. Bacon *et al.*, "Fpga programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

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

[3] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proc. of the 49th Design Automation Conference*. IEEE, 2012, pp. 1212–1221.

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

[5] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[6] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic *et al.*, "An agile approach to building risc-v microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, 2016.

[7] S. Sutherland and D. Mills, "Synthesizing SystemVerilog: busting the myth that SystemVerilog is only for verification," *SNUG Silicon Valley*, p. 24, 2013.

[8] J. Bruant *et al.*, "sv2chisel (System)Verilog to Chisel translator," Retrieved August 2020. [Online]. Available: https://github.com/ovh/sv2chisel

[9] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to high-level language translation," in *Proc. of the International Conference on Software Maintenance*. IEEE, 1998, pp. 228–237.

[10] L. Jacomme, F. Pétrot, and R. K. Bawa, "Formal analysis of single WAIT VHDL processes for semantic based synthesis," in *Proc. of the 12th International Conference on VLSI Design*. IEEE, 1999, pp. 151–156.

[11] P. Patros and K. B. Kent, "Automatic detection and elision of reset sub-circuits," in *Proc. of the 27th International Symposium on Rapid System Prototyping*, 2016, pp. 26–32.

[12] C. E. Cummings, D. Mills, and S. Golson, "Asynchronous & synchronous reset design techniques-part deux," *SNUG Boston*, vol. 9, 2003.

[13] A. Izraelevitz *et al.*, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *Proc. of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 209–216.

[14] T. Parr and K. Fisher, "LL(*) the foundation of the ANTLR parser generator," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 425–436, 2011.

[15] R. Behl, "SimpleCPU," Retrieved July 2020. [Online]. Available: https://github.com/SimpleCPU/SimpleCPU

[16] C. Wolf, "PicoRV32 - a size-optimized RISC-V CPU," Retrieved July 2020. [Online]. Available: https://github.com/cliffordwolf/picorv32

[17] J. Decaluwe, "MyHDL: a python-based hardware description language." *Linux journal*, no. 127, pp. 84–87, 2004.

[18] S. Bourdeauducq, "Migen: A python toolbox for building complex digital hardware," Retrieved July 2020, 2013. [Online]. Available: https://m-labs.hk/migen/manual/introduction.html

[19] C. Papon, "SpinalHDL: An alternative hardware description language," FOSDEM, 2017.