Synthesis of Decoder Tables using Formal Verification Tools

Keerthikumara Devarajegowda[1,2], Johannes Schreiner[1,3], Wolfgang Ecker[1,3]
Infineon Technologies AG[1] - Technical University of Kaiserslautern[2] - Technical University of Munich[3]
Mail: Infineon Technologies AG, Am Campeon 1-12, Neubiberg - 85579, Germany
$Email$:<firstname>.<lastname>@infineon.com

***Abstract*— This paper discusses a use case of formal verification tools in automating the implementation of RTL modules. In detail, classical hardware design tasks involve constructing a micro-architecture and a control unit that makes the design functional. Traditionally, these control units are built by analyzing the micro-architecture implementation and the intended function. In this work, we introduce a novel approach for automatic synthesis of control signals for a given micro-architecture. Our approach uses formal methods to automatically derive a working control unit and/or decoder for a certain micro-architecture. For this purpose, the properties (SystemVerilog Assertions) developed for design verification are re-used with minor refinement. To show the applicability of our approach for real-life designs, we develop a RISC five-stage pipeline micro-architecture and automatically derive the instruction decoder. Our method saves a significant amount of work (i.e. manually tailoring units), allows to focus on the micro-architecture design and makes it easy to analyze various ISA alternatives, accelerators and architecture alternatives.**

I. Introduction

Even though the effectiveness of formal verification tools for design analysis is well-known, to the best of our knowledge, formal verification tools are not widely applied for the same. In addition to design verification tasks (full-proof, bug-hunting, coverage analysis), formal verification tools are leveraged for several design related tasks such as automatic linting, reachability analysis, etc. In addition, the tools can be utilized for examining the design behavior. In this paper, we discuss the usage of formal tools for generation of control unit.

The first step in a hardware implementation is the definition of a micro-architecture as an implementation template (e.g. pipelines for CPUs, FIR filters, (programmable) FSMs). These templates are, in a second implementation step, equipped with necessary control signals so that the overall design provides the required functionality (i.e. the correct behavior of specific signals at specific points in time). Since the method involves trial and error approach, this is an elaborate and error prone task. Hence, we developed an approach to automatically derive decoder tables by utilizing formal verification tools.

For simple control-data-path architectures, the generation of both data-path and control unit can be automated by using high-level synthesis tools. This approach however maps to simple data-path control architecture and does not support other architectures. The technique proposed in this paper can be used for synthesizing control signals for all types of control-data-path micro-architectures.

For explaining and proving our approach, we synthesize a control unit, i.e. an instruction decoder, of a processor implementing the RiscV [1] open source Instruction Set Architecture (ISA). RiscV is becoming widely visible and is being labeled as the next standard open architecture for industry implementations. Hence, we consider a pipeline implementation of RiscV ISA as the test vehicle. RiscV ISA specifies various extensions spanning from RV32I base integer instruction set to standard extension for decimal floating-point and many other. The part that must be adopted for different extensions of the ISAs is mainly the look-up-table[1] in the instruction decoder, which drives the control signals. These control signals are different for different instruction encoding and provide the required functionality of each instruction. The interface of the decoder look-up-table is shown in Fig. 1 illustrating the complexity of the table.

We first generated the RTL micro architecture design of the 5-stage pipeline CPU of RiscV ISA with an empty look-up-table. Next, we generate a set of properties for each instruction of the selected ISA that capture required micro-architecture behavior. The RTL and properties are generated using a meta-modeling framework following Object Management Group's (OMG) Model-Driven-Architecture (MDA) idea for code generation [8].

[1]Look-up-tables are generally used to replace run-time computation with simpler array indexing operations and provide significant performance benefits.

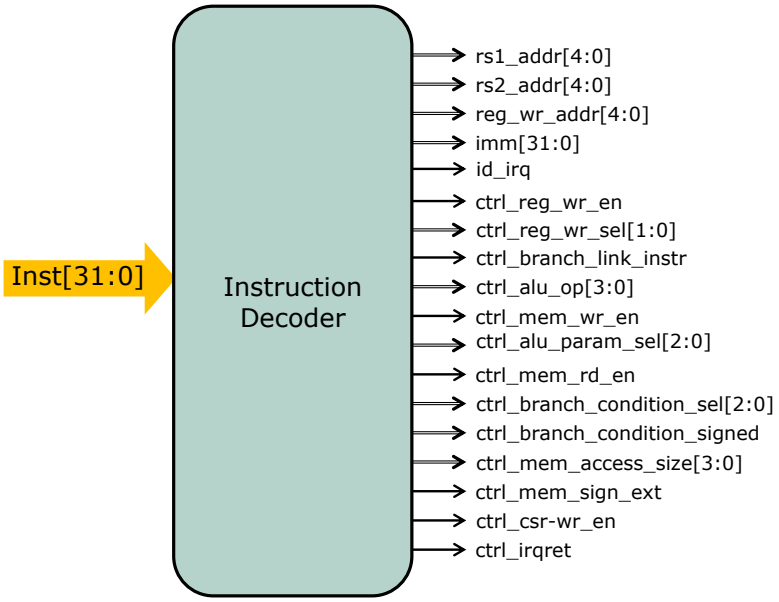

Figure 1: Block view of Instruction Decoder

These generated properties are then used with cover directive in a formal verification tool to generate an execution trace and to extract the control signals hereof.

One of the major advantage our technique is reduction in turn-around time for late changes in the decoders due to extension and changes in the specification. The properties generated for extracting control signals can be re-used for design verification with minimal changes. As a result, our approach reduces considerable efforts for design verification. In addition, our approach provides maximum benefit when used in a design flow methodology proposed in [10]. The methodology in [10] follows the principle of correct-by-construction and proposes parallel development of both verification IP and the RTL design.

The rest of the paper is organized as follows. Section II provides a discussion on deriving control signals using simulation techniques and manual analysis. Section III elaborates our approach of automatic derivation of control signals. In section IV, we provide an overview of the generation framework, property generation and their suitability to the proposed approach. A section on brief summary of the work completes the paper.

II. State of the Art

A. Manual Coding

For typical digital design problems, a set of micro-architectural patterns is available. Usually, one of these patterns is implemented and some control signals are provided to program or make the micro-architecture configurable. Functional verification then automatically checks that the micro-architecture meets the specified requirements.

In order to derive the control or configuration signals, the design engineer manually combines his understanding of the requirements and his knowledge of the implemented micro-architecture. For example, given the understanding of a certain CPU instruction and the knowledge of the micro-architecture, the control signals are manually set. However, this step is often repetitive and requires deep knowledge of the micro-architecture.

B. Synthesis using Simulation Technique

While generating a 5-stage implementation of the RiscV ([4]), a simulation-based method was developed to generate the look-up-table. This method identifies the necessary control signals for each instruction using a trial-and-error approach. For this purpose, we generated a test-bench to check the required functional behavior from a formal ISA definition. For each instruction, the set of possible control signals was exhaustively simulated against the test-bench. When the test-bench did not report an error, a control vector was found so that our micro-architecture behaves correctly. Even for the most minimalist micro-architectures and simple instruction sets, repeating this process for all instructions takes many hours to complete. This can be accelerated by parallel simulations (e.g. one for each instruction). The effort required however still increases exponentially with the number of control bits required for a data-path architecture. A simulation-based technique for complex, real-world micro-architectures is thus not feasible.

III. Automated Generation of Decoder tables

A. Approach

The proposed approach for utilizing formal verification tools to derive control signals in an automated manner is depicted in Fig. 2. Starting point is the micro-architecture template of the intended hardware system and the set of properties that are drawn from the specification (ex: ISA). A property is written/generated to verify the intended behavior of a particular operation (ex: ADD instruction) such that necessary signals of the micro-architecture template are captured. The properties are encoded in SVA following a coding style that should be supported by almost all commercial formal verification tools.

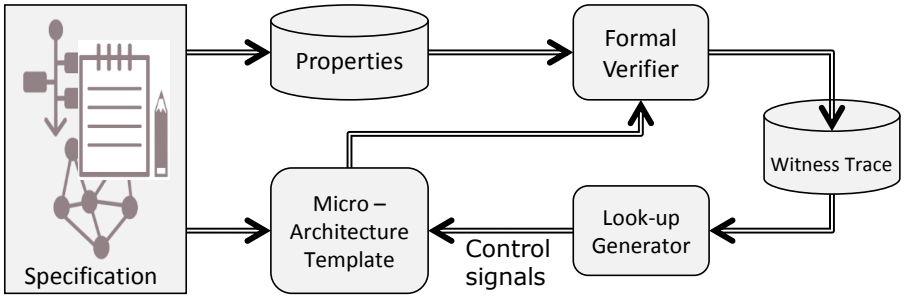

Figure 2: Block diagram of automatic look-up-table generation

The set of properties and the micro-architecture with the empty look-up-table (micro-architecture template) are then applied to the formal tool. Due to the nature of formal verification[2], the tool is expected to provide a counter-example (CEX) if the properties are evaluated with *ASSERT* directive. This is due to the empty look-up-table in the micro-architecture. However, formal tools also provide a witness trace that demonstrates a scenario in which all signals attain the required values. Hence, the properties are evaluated with the *COVER* directive[3]. *COVER* witness traces are an optimistic approximation of all possible scenarios satisfying the property sequence. Due to this, the generated property for a specific operation must be encoded to instruct the formal tool to prohibit other operations which partly have the same results. A detailed example is provided in section IV-B. The witness traces are investigated to extract necessary control signals and fed into the look-up-table generator. This task is automated by setting up scripts that parse the log files and generate control signal values required by the look-up generator. The look-up-generator then fills the micro-architecture template with necessary control signals.

The mechanism of encoding properties to select and exclude operations requires more effort if the properties are manually implemented. However, the complexity increased in coding properties to select and exclude operations can be mitigated by adopting automatic property generation approach. In addition, automatic code generation flow is simplified by transforming informal specifications into formal specifications. Generating properties from formal specification avoids ambiguity and enables simpler code generators. The approach taken for automating property generation is briefly outlined in section IV-A. Nevertheless, the proposed method for automatic derivation of control signals is independent of any property generation flow and the method is applicable to nearly any control decoder design.

IV. Application

We applied the technique described in section III to the automated generation of instruction decoder look-up-tables of various micro-architectures supporting the RiscV RV32I Base Integer Instruction Set [1, page 27]. Fig. 3 shows a well known template of a 5-stage CPU with the below part depicting our approach for generating control signals. As already mentioned in the previous section, in order to implement the properties with mechanism to exclude operations, we employed an automated code generation approach. The following subsections describe our code generation framework, property generation and derivation of control signals for an instruction decoder using cover properties.

[2]Formal methods provide exhaustive proof spanning all (legal) input combinations i.e., a property is verified against the design for all possible input combinations.

[3]The *COVER* directive instructs the verification tool to search for an example trace obeying the property sequence during analysis.

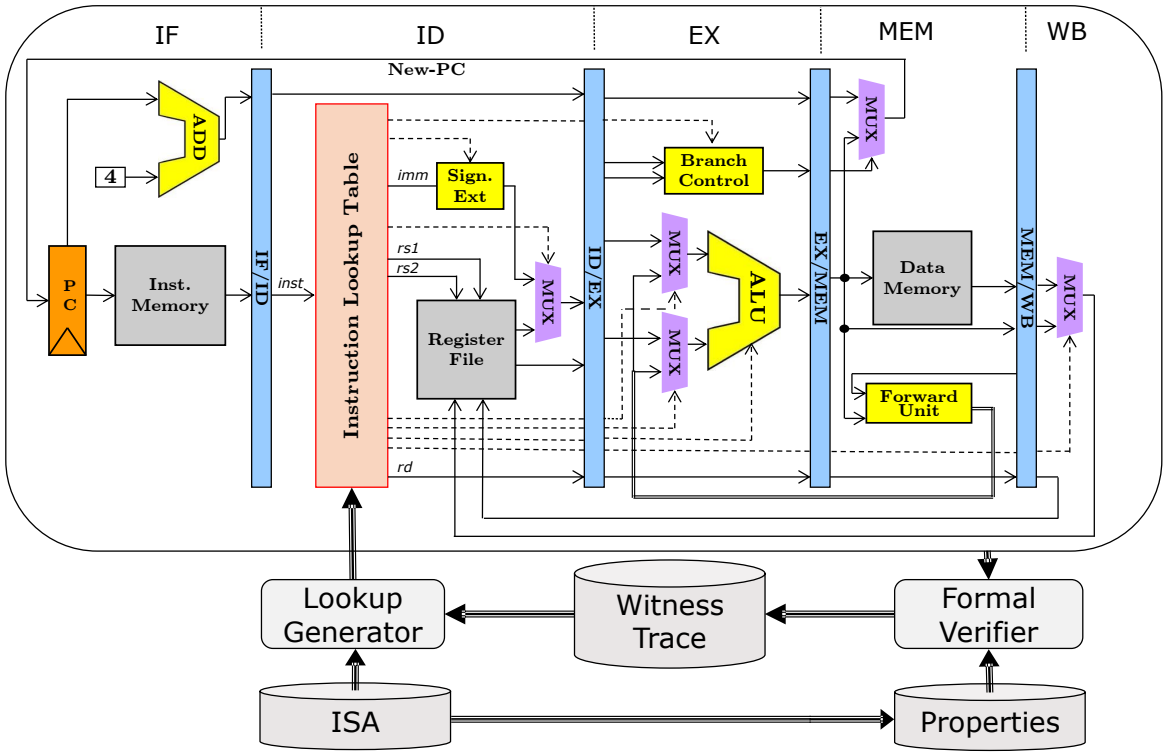

Figure 3: 5-stage pipeline template according to [7] and look-up-table generation

A. *Meta-modeling framework for code generation*

At Infineon, a meta-modeling framework based on Python and Mako templates is widely used for code generation. This meta-modeling framework for efficient code generation is deployed for more than 100 applications and is the source of high productivity benefit [6], [5]. The framework is currently being enhanced by splitting the generation flow into different steps/layers following the vision of OMG'S Model Driven Architecture approach for code generation [5]. The flow involves creation of a series of models in which the preceding model is an abstracted version of the current model. The code generation flow is split into following steps:

- In the first step, the specifications and requirements of the intended hardware design are transformed into an abstract model called Model-of-Things (MoTs). The MoT corresponds to computational-independent model (CIM) in the original MDA definition [9]. The implementation details are intentionally left-out from these abstract models to allow architectural alternative exploration. That is, the MoTs contain information of "what shall be implemented?" and abstracts away the information of "how shall be implemented?". For example, an abstract model is built to represent the RiscV ISA.
- Next, the abstract descriptions are transformed into an intermediate model that contains hardware implementation (micro-architecture) details. This layer corresponds to the platform-independent model (PIM) in the original MDA definition [9]. For property generation, these concrete models hold the intended property trace.
- Finally, these intermediate models are mapped onto the corresponding target code using Mako templates or using an additional intermediate model, which we call model-of-view (MoV). This MoV corresponds to the platform-specific model (PSM) in the original MDA definition [9].

The aforementioned meta-modeling framework is used for the generation of both, the CPU implementation and the properties. We first built the 5-stage pipeline RiscV processor using a model-driven flow mentioned above and elaborated in [3].

B. *Generation of Properties*

The generation of properties follows the MDA approach and is outlined in [2]. To facilitate the understanding of this paper, we provide a brief summary in the following.

Fig.4 shows the property generation flow following the MDA approach. As mentioned earlier, the first step is to convert specifications into abstract models which are called as Model-of-Things. These abstract models are a

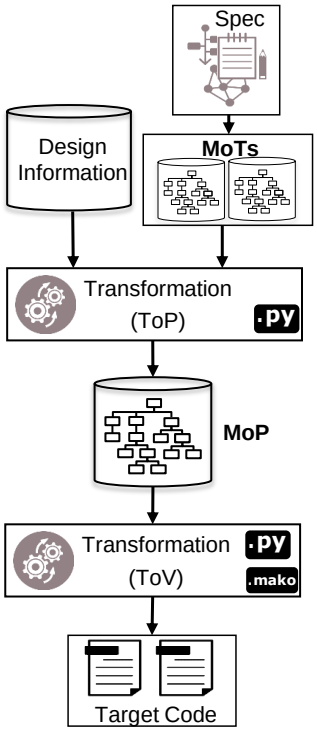

Figure 4: Property generation flow

formalized version of informal specifications and do not include implementation details. For RiscV ISA, we create an abstract model called 'MetaRisc' that contains instruction encoding, abstract instruction behavior, and several objects such as register files, program counter, memory, etc..

The translation from abstract models (MoTs) into a more concrete model is performed in Templates-of-Property (ToP). These Templates-of-Properties are used to aggregate the information from MoTs and define the trace for the properties. The property trace information is captured in a model called Model-of-Property (MoP). The definition of the property trace is aided by the underlying automation framework. In addition, ToPs also need information of the design implementation (hierarchical and input/output port names of blocks/sub-blocks). This information is either derived directly from the RTL files or from the Model-of-Design (MoD) if the design is generated following the MDA approach outlined in [3].

Model-of-Property is a platform independent way of specifying property traces and is the main model of our property generation flow. Finally, the MoP is mapped to Templates-of-View (ToV) to provide the properties in a property language of available tools. The generation framework is built to address multiple languages such as SVA, ITL or PSL.

C. Cover Properties

For the 5-stage pipeline CPU of RiscV ISA, we generate the properties using the MDA flow described in previous paragraphs. ToPs are used to unparse the abstract RiscV ISA model (MetaRisc) and construct the property trace for all instruction encoding types (R-Type, I-Type…). Hence for each instruction encoding type, a property trace is defined and re-used for all instructions of the particular encoding type. A property is generated for each instruction, such that it captures the required values[4] for several signals along the pipeline over a period of 4 - 5 clock cycles.

The property suite and the CPU design with an empty look-up-table in the instruction decoder are applied to the formal verification tool. It is important to make sure that while searching for control signals of a specific instruction, other operations are excluded. That is, the property generated to capture the control signals for one instruction performing a 'specific operation' must exclude all 'other operations'. This is because, when a formal verification tool is asked to provide a witness trace for a property sequence, the witness trace provided by the tool is one of several possible traces satisfying the property sequence. For example, consider the below ADD instruction:

[4]We cover only those values needed to verify correct behavior

ADD R3, R1, R2

The ADD instruction performs the addition of contents of $R1$ and $R2$ source registers and stores the result in the destination register $R3$ [1]. Now, consider the SLL instruction:

SLL R3, R1, R2

The SLL instruction performs the logical left shift operation on the contents of the source register $R1$ by the shift amount stored in the lower 5 bits of source register $R2$ and stores the result in destination register $R3$ [1]. For both instructions, following trace is true:

$$\text{if R1 = 1 \& R2 = 1} \implies \text{R3 = 2}$$

Hence, while searching for witness trace for ADD instruction, the formal tool may provide the witness trace for an SLL instruction. As a consequence, the property must instruct the formal verification tool to provide a witness trace valid for the specific instruction only. As mentioned earlier, our property generation framework allows to specify the property trace in Python language. Selecting and excluding operations for each instruction is hence a simple and straightforward task.

```
1 //Property for R-type, 'ADD' instruction with forwarding disabled and with exclusion mechanism
2 property _ADD;
3 @(posedge clk)
4 disable iff(reset)
5     $changed(instr)                                 &&
6     program_counter  == ($past(program_counter) + 4) &&
7     instr[6:0]       == R_TYPE                      &&
8     instr[14:12]     == FUCNT3                      &&
9     instr[31:25]     == FUNCT7                      &&
10    !forwarding_en                                  &&
11    !branch_en                                      &&
12    |->
13    ##0
14    rs1_addr         == instr[19:15]                          &&
15    rs2_addr         == instr[24:20]                          &&
16    reg_wr_addr      == instr[11:7]                           &&
17    ##1
18    alu_in1             == $past(rs1_data)                    &&
19    alu_in2             == $past(rs2_data)                    &&
20    alu_result          == alu_in_1 + alu_in2                 &&
21    alu_result          != 0                                  &&
22    alu_result          != (alu_in1 >> alu_in2[4:0])          &&
23    alu_result          != (alu_in1 & alu_in2))               &&
24    alu_result          != (alu_in1 << (alu_in2[4:0])))       &&
25    alu_result          != (alu_in1 - alu_in2))               &&
26    alu_result          != (alu_in1 < alu_in2))               &&
27    $signed(alu_result) != ($signed(alu_in1) >>> (alu_in2[4:0]))&&
28    alu_result          != (alu_in1 | alu_in2))               &&
29    alu_result          != (alu_in1 ^ alu_in2))               &&
30    $signed(alu_result) != ($signed(alu_in1) < $signed(alu_in2))
31    ##1
32    !data_mem_wr_en
33    ##1
34    reg_wr_data ==  $past(alu_result,2)                       &&
35    reg_wr_en                                                 &&
36    reg_wr_addr ==  $past(instr[11:7],3);
37 endproperty
38
39 //Assertion Directive
40 COVER_add_instruction: cover property(_ADD);
```

Figure 5: Property (in SVA) to generate control signals for R-Type, ADD instruction (simplified)

A generated property that captures the required signal behaviour in the pipeline for 'R-type, ADD instruction' ([1, page 15]) is shown in Fig. 5. Lines 3 and 4 define the trigger and reset sequence respectively. Lines 5-11 (antecedent/enabling sequence) assume values for signals required for the ADD instruction in decode phase. Lines 12-36 (consequent/satisfying sequence) capture the expected values for various signals in decode (ID), execute (EX), memory-access (MEM) and write-back (WB) phases of the pipeline. Lines 14-16 capture correct decoding of source/destination register addresses in the decode phase. Lines 18-20 capture the ADD instruction behavior

during execute phase. Lines 21-30 exclude the operations as already explained in previous paragraphs. Lines 32-36 reflect the pipeline behavior during mem-access and write-back phases.

If no witness trace can be generated, the given architecture is not capable to fully support the ISA[5] and must be enhanced. If a witness trace is generated by the tool, the expected control signals for each instruction are extracted and the instruction decoder is generated.

D. Results and Discussion

We used for our work the formal verification tool Onespin 360 (Version: 2017_06(38)). The witness computation time for each property is around one second. The tool (generally a feature of all commercial formal tools) allows to dump the witness trace information into a text file. From this text file specific signal values at specific time-points can be extracted. We set-up a "Lookup Generator" to parse the text log file and extract the control signals in a dictionary format. The overall witness generation time for all instructions (RISC-V RV32I instruction set) is less than a minute. Table I shows the generated control signals for R-type instructions.

Table I: Generated control signals using the approach depicted in Fig. 2 (Control signals for R-Type instructions are shown here).

Instruction	reg_wr _en	reg_wr _sel	branch _link _instr	alu_op	mem_wr _en	alu _param _sel	mem_rd _en	branch _condition _sel	branch _condition _signed	mem _access _size	mem_sign _ext	csr _wr	irqret
AND (R-type)	1	00	0	0110	0	000	0	000	0	0000	0	0	0
OR (R-type)	1	00	0	0111	0	000	0	011	0	0000	0	0	0
SLL (R-type)	1	00	0	0010	0	000	0	011	0	0000	0	0	0
SUB (R-type)	1	00	0	0001	0	000	0	010	0	0000	0	0	0
ADD (R-type)	1	00	0	0000	0	000	0	000	0	0000	0	0	0
SRL (R-type)	1	00	0	0100	0	000	0	011	0	0000	0	0	0
SLTU (R-type)	1	00	0	1010	0	000	0	011	0	0000	0	0	0
XOR (R-type)	1	00	0	1000	0	000	0	000	0	0000	0	0	0
SRA (R-type)	1	00	0	0101	0	000	0	000	0	0000	0	0	0
SLT (R-type)	1	00	0	1001	0	000	0	011	0	0000	0	0	0

In addition to the automatic derivation of control signals, one major advantage of our approach is that the same kind of properties can be used to synthesize the control signals for the instruction decoder and also to validate the micro-architecture. The properties for extracting control signals are modified in order to exclude operations not specified in the instruction. The aforementioned generator framework allows flexibility to easily exclude unintended behavior in the property trace. Existing languages such as SVA and existing tools can be used to provide the required signals and in turn synthesize the decoder. We used the same generated properties but without excluding mechanism for verifying the functional behavior of the pipeline for all instructions in RISC-V RV32I instruction set.

Verification is also done with constrained random simulation which does not cause big overhead, since such a simulation must be setup for use case verification anyhow. Later is needed since some behavior can be verified in conjunction with software - e.g. interrupt behavior - only. Nevertheless, we do not expect to find any bug in relation to ISA definition, since the formal verification tool guarantees that the architecture provides the expected behavior if the computed control signals are applied. In comparison to an exhaustive approach based on simulation, the method scales and is applicable in real-life designs.

[5]Assuming the verification environment is free of over-constraining

Also, as in traditional design flows the properties are developed once the RTL is ready for verification. However in [10], Urdahl et al. show that considerable verification efforts can be reduced by following a "Properties first" approach. The design flow proposes systematic development of a verification IP concurrently with the design process. The properties are first designed according to system-level specifications and are refined later in the design process. Our approach of automatic generation of control signals suits a similar design flow, where verification and design tasks are carried out concurrently.

V. Summary

In this paper, we have introduced a novel approach for generating control signals for decoder tables using formal verification tools. We also showed how the generated properties for control signals synthesis can be re-used for functional verification of the design with minimal modifications. The technique provides quick turn-around time for changes that relate to extensions or changes in the decoding structures. To show the applicability of our approach for real-life designs, we successfully synthesized the control unit for 3-stage, 5-stage pipelined RiscV CPU. In addition to the instruction decoder table synthesis, the technique can be used to derive the control signals for micro-architectures that involve a decoding operation to select various computations/operations based on certain input signals. Further, the technique unfolds a new direction of application for formal verification tools.

References

[1] Krste Asanovi Andrew Waterman. The risc-v instruction set manual volume i: User-level isa document version 2.2. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, May 2017.

[2] Keerthikumara Devarajegowda and Wolfgang Ecker. On Generation of Properties from Specification. In *IEEE International High Level Design Validation and Test Workshop, HLDVT 2017, Santa Cruz, CA, USA, October 5-6, 2017*, pages 95–98, 2017.

[3] W. Ecker and J. Schreiner. Introducing Model-of-Things (MoT) and Model-of-Design (MoD) for simpler and more efficient hardware generators. In *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, Sept 2016.

[4] Wolfgang Ecker and Johannes Schreiner. Introducing Model-of-Things (MoT) and Model-of-Design (MoD) for simpler and more efficient Hardware Generators. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 24, 2016.

[5] Wolfgang Ecker and Johannes Schreiner. Metamodeling. In Soonhoi Ha and Jrgen Teich, editors, *Handbook of Hardware/Software Codesign*, chapter 10, pages 266–290. Springer, 2017.

[6] Wolfgang Ecker, Michael Velten, Leily Zafari, and Ajay Goyal. Metamodeling and code generation - the infineon approach. In Wolfgang Mueller and Wolfgang Ecker, editors, *MeCoES - Metamodelling and Code Generation for Embedded Systems: Workshop with ESWEEK*, pages 1–4. http://adt.cs.upb.de/mecoes/MeCoES2012-Proceedings.pdf, 2012.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[8] "OMG". MDA - The Architecture of Choice for a Changing World, 2016.

[9] F. Truyen. The fast Guide to Model Driven Architecture.

[10] Joakim Urdahl, Shrinidhi Udupi, Tobias Ludwig, Dominik Stoffel, and Wolfgang Kunz. Properties First? A New Design Methodology for Hardware, and Its Perspectives in Safety Analysis. In *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, pages 84:1–84:8, New York, NY, USA, 2016. ACM.