Unified (A)Synchronous Circuit Development

Philipp Paulweber
University of Vienna
Faculty of Computer Science
Vienna, Austria
philipp.paulweber@univie.ac.at

Jürgen Maier*
TU Wien
Insitute of Computer Engineering
Vienna, Austria
juergen.maier@tuwien.ac.at

Jordi Cortadella
Universitat Politècnica de Catalunya
Department of Computer Science
Barcelona, Spain
jordi.cortadella@upc.edu

***Abstract*—Despite its development several decades ago and several very beneficial properties asynchronous logic design, which is data driven and runs as fast as possible in all situations, is rarely used nowadays. Reasons are of course its disadvantageous properties such as bad testability but also required sophisticated knowledge for designers and missing tools. In this paper we draw a path to tackle the latter points by suggesting a tool/way to generate multiple circuit implementations from a single description. We are aiming to convert specifications written in various input languages, e.g. C or VHDL, to an unified Internal Representation (IR). This IR is composed of building blocks (semantic vocabulary) specified through the Abstract State Machine (ASM) based formal method. The ASM artifact is then used to generate the circuit in the desired (a)synchronous design style. As short term goal we aim to train developers by reading synchronous descriptions and converting them to asynchronous designs however in the long run we hope to establish a unified path for circuit development, which only requires an abstract behavioral description.**

***Index Terms*—Abstract State Machines, Asynchronous Logic**

I. Introduction

In contrast to the broadly used synchronous approach, which uses a central clock to coordinate the single units, data-driven asynchronous logic utilizes dedicated signals to indicate when new data has arrived (request) and when the old data has been processed (acknowledge). This allows the circuit (1) to work as fast as possible and (2) to adapt much better to Process-Voltage-Temperature (PVT) variations. Where synchronous circuits fail due to timing violations their asynchronous counterparts still deliver correct results.

Despite these argument and being available for several decades now, asynchronous logic is still used only marginally in digital design [1]. Reasons for that are manifold: Since data is processed immediately after arrival, a very high level of concurrency is achieved. This leads however also to a huge amount of possible states and thus bad testability, as all of them have to verified. Furthermore, tool support is still lacking. At the moment this seems to be a chicken-egg problem: Companies are not ready to develop new tools until the demand is high, which does however not grow due to the lacking tool support. Please note that, albeit with increased effort, it is actually possible to design asynchronous logic with available tools assumed that the designer has sophisticated knowledge about the asynchronous design style, which differs in certain points significantly from the synchronous one. For example one has to make sure that the communication protocol (request and acknowledge) can be fully executed, which sometimes requires the introduction of additional memory elements [2]. Thus switching to asynchronous logic means retraining the designer which results in high risk for a company.

Generating asynchronous circuits systematically from Data Flow Graph (DFG)s, i.e., models that show how data is propagated from one operation to the next but neglects the inherent timings, is already possible. For this purpose operations and special constructs such as *merge* or *join* are simply replaced by corresponding asynchronous block as shown e.g. in [3]. In computer architectures similar concepts have already been implemented successfully, e.g. in data flow processors [4] or in out-of-order computational units, which essentially generate a DFG at run time.

In summary, we currently have a very good Intermediate Representation (IR) for asynchronous logic, namely DFGs, and various methods to describe the behavior on different abstraction levels (e.g. C, VHDL, or Verilog). The main issue is to properly close the gap between them. Of course, we are aware that available compiler tools, for example Low Level Virtual Machine (LLVM) [5], are capable to generate data flow models, which was for example used by Josipović et al. [3] to develop Elastic Pipelines [6] based on a algorithmic description in C. While the DFG in this paper could be converted to asynchronous logic in a straightforward way this is not valid in general, due to the synchronous or computer architectural assumptions that are utilized by the compilers. An example are memory references which are common in software however hard to convert to hardware. Therefore the biggest challenge currently in our opinion is the generation of an IR that can be used as starting point for multiple purposes, e.g., development of synchronous and asynchronous circuits.

Contribution: We therefore propose a tool that can convert circuit descriptions in various formats and abstraction levels, e.g. pseudo code, C, or VHDL, into a single common IR. The latter is based on the well-known formal method Abstract State Machine (ASM) [7] which allows exactly what we need: a unified specification of the circuit behavior independent of the desired implementation details. The latter are only fixed during the export to an implementation specific IR, for example DFG in the case of asynchronous circuits. Overall the task of the tool is to (a) extract valuable information from different descriptions and (b) export this information to the desired format and circuit style.

*The work presented in this paper is supported in part by the Austrian Science Fund (FWF) under project number I3485-N31.

II. Abstract State Machines

In 1995 Gurevich [7] described the ASM theory, which is a well-known formal method based on transition *rules*, *agents*, and mathematical *function* states that can be used to specify arbitrary algorithms, applications or even whole systems. The mathematical *function* state is defined with a corresponding type relation. *Agents* are able to execute *rules*, which enables the producing of *updates* resulting in a change of the global mathematical *function* state. Since the appearance of the ASM theory, several definitions and implementations were create of ASM-based languages, interpreter, and compilers. However, all of them focus mainly on the analyzes of ASM specifications for certain properties or on software-sided simulations. The Corinthian Abstract State Machine (CASM) language [8] and project[1] focuses on enabling not only to analyze and simulate ASM-based specifications. One of the main research objectives is to establish through a model-based transformation approach [9] a generic transformation of CASM specifications to various target languages and execution environments including the software and hardware domain. Important is that in contrast to already existing hardware-based languages or IRs (Chisel [10] or FIRRTL [11]), CASM does not include any assumptions of the resulting circuit style (synchronous design, clock etc.). Therefore, the same CASM input specification can be (re)used and (re)targeted. In order to achieve this goal multiple compiler IRs [12] were introduced to separate the languages' own run-time implementation of a certain target environment or language. A possible asynchronous hardware target environment could be the *link and joint model* by Roncken et al. [13].

III. Accelerating Asynchronous Logic

In the introduction we stated that asynchronous logic is only marginally used at the moment. The question is how we actually can improve this situation with the proposed tool. Specifying the new IR building blocks (semantic vocabulary) as well as the implementation of various front-ends and back-ends can only happen in steps. We are planning to start with specific front-ends for Verilog and VHDL and, naturally, with an asynchronous circuit style back-end. In the beginning this would allow designers to read existing (synchronous) circuit descriptions and generate asynchronous counterparts, which gives them the chance to (a) get a quick estimation what asynchronous logic is capable of and thus support future asynchronous implementation and (b) learn by comparison how to properly design asynchronous circuits on the fly.

The proposed tool is also supposed to enhance verification and validation as the transformation to a formal model enables the usage of automatic verification methods to proof specific properties and thus show correct behavior.

Currently, we are still at the very beginning of our research. We already have an ASM implementation, however the required IR building blocks (semantic vocabulary) are not specified yet. This is clearly one of the first steps to go. To find a proper abstraction, we have to analyze the structure of asynchronous circuits and systems and based on these create suitable data structures that are capable to model the data flow graph appropriately. From that it should be easy to generate asynchronous logic automatically by using the already available gates and building blocks.

[1]https://casm-lang.org

IV. Conclusion

Asynchronous circuits are far less popular than their synchronous counterparts. In this paper we have shown a roadmap of a tool that might change this, as it is capable to generate from a single (high-level) description both synchronous and asynchronous circuits. For that purpose we need an IR that represents through proper building blocks (semantic vocabulary) parsed input descriptions. This IR is specified using an ASM-based language. Our short term goal is to allow the user to describe the behavior of the desired circuit in different formats and automatically generate asynchronous implementations. Together with an automatic verification this hopefully makes asynchronous circuits more attractive.

References

[1] S. M. Nowick and M. Singh, "Asynchronous Design — Part 1: Overview and Recent Advances," *IEEE Design & Test*, vol. 32(3), pp. 5–18, 2015.

[2] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 657–662, July 2006.

[3] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically Scheduled High-Level Synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on FPGAs*, pp. 127–136, ACM, 2018.

[4] J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," in *ACM SIGARCH Computer Architecture News*, vol. 3, pp. 126–132, ACM, 1975.

[5] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Code Generation and Optimization*, pp. 75–86, IEEE, 2004.

[6] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1437–1455, 2009.

[7] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide - Specification and Validation Methods," pp. 9–36, New York, NY, USA: Oxford University Press, Inc., 1995.

[8] R. Lezuo, G. Barany, and A. Krall, "CASM: Implementing an Abstract State Machine based Programming Language," in *Software Engineering (Workshops)*, pp. 75–90, 2013.

[9] P. Paulweber and U. Zdun, "A Model-Based Transformation Approach to Reuse and Retarget CASM Specifications," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016*, Lecture Notes in Computer Science 9675, pp. 250–255, Springer, 2016.

[10] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, IEEE, 2012.

[11] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, *et al.*, "Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations," in *Proceedings of the 36th International Conference on Computer-Aided Design*, pp. 209–216, IEEE Press, 2017.

[12] P. Paulweber, E. Pescosta, and U. Zdun, "CASM-IR: Uniform ASM-Based Intermediate Representation for Model Specification, Execution, and Transformation," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018*, Lecture Notes in Computer Science 10817, pp. 39–54, Springer, 2018.

[13] M. Roncken, S. M. Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland, "Naturalized Communication and Testing," in *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 77–84, IEEE, 2015.