μ Systems Research Group

School of Electrical and Electronic Engineering



Design of Control and Datapath of Scenario-based Hardware Systems

Alessandro de Gennaro

Technical Report Series NCL-EEE-MICRO-TR-2016-203

December 2016

Contact: a.de-gennaro@ncl.ac.uk

Supported by EPSRC grant EP/L025507/1 (A4A: Asynchronous design for analogue electronics)

NCL-EEE-MICRO-TR-2016-203 Copyright © 2016 Newcastle University

μSystems Research Group School of Electrical and Electronic Engineering Merz Court Newcastle University Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

Design of Control and Datapath of Scenario-based Hardware Systems

Alessandro de Gennaro

December 2016

Abstract

One of the possible approach to the design of hardware systems is to start with the description of their behaviours. Each of them represents a task that the system is supposed to execute. A task, also known as scenario, is nothing else but the set of operations which need to be performed in a certain order for the achievement of the requested answer.

This has been validated with the design, fabrication and verification of an asynchronous reconfigurable pipeline (using a miniASIC with TSMC 90nm technology). The models and tools used are available in the open source Workcraft framework. Other models and theories for the synthesis of scenario-based hardware systems are also considered and briefly examined, in order to picture the benefits that the presented design-flow comes with.

1 Introduction

Over the years, hardware architectures are increasingly getting more complex. Processors gain new instructions and system-level features, hardware parallelism also grows due to the working frequency limit given by the slower transistors down-scaling. Formal specification and modelling is becoming an important area of research not only in academic environments, but also in leading processor companies like ARM [1]. Various are the reasons why using a systematic yet formal approach to the design of hardware systems is becoming appealing. Firstly, documenting the specification of hardware structures via mathematical models helps designers, with different backgrounds, develop a good understanding of the system [2]. Secondly, abstracting the system leads to a reduction in the design time, since the majority of the mistakes can be captured and solved earlier. Furthermore, using a systematic approach can be beneficial either for optimising the whole system and for obtaining a scalable entity which supports yearly architectural updates.

The higher hardware complexity brings a progressively longer delay in the design phase. To deal with this problem, a task is usually divided into multiple sub-tasks. Each sub-task is assigned to a different engineer/team which is in charge of solving that particular problem respecting the specification assigned. The latter is useful to let all the sub-answers be compatible to each other. This further fosters the importance of having a formal specification. The systems, which can be divided into independent sub-systems, (also known as scenarios) are named *scenario-based systems*. This approach can be applied to a wide range of applications: processor instruction sets [9], process-mining traces [10] and analogue system specifications [11]. The aim of this article

is to show that, from scenario-based specifications, it is possible to obtain hardware architectures able to satisfy the scenarios that the system is composed of.

In the literature, different attempts to build a hardware system starting from its scenarios are present [3][4][5]. There are models and theories such as Live Sequence Charts (LSCs) [6], Message Sequence Charts (MSCs) [7] and UML Sequence Diagrams(UML SDs) [8]. The above methods are not supported by industry-strength EDA software. The approach presented in this article relies on two graph-based models which have been validated over sound and extensive case studies: *Conditional Partial Order Graphs* (CPOG) [13][14] and *Dataflow Structures* (DFS) [15][16]. The former representation can be used for the design of the control side of the architecture, whereas the latter for the datapath. These two models are implemented as tool-plugins in the Workcraft framework [17][18].

The case study used is an algorithm for the ordinal pattern encoding (OPE) [19]. The article characterises a rationale for the computation of the ordinal analysis, meant for the examination and the prediction of various data-streams. The architecture is a pipeline of modules which, trough various operations, analyses chains of numbers. The length of the chain that the structure can analyse is given by the length of the pipeline. My goal was building an asynchronous reconfigurable pipeline for the analysis of different stream-lengths. The structure has been first implemented into a FPGA-based board (Maxeler [21]), and then into an ASIC using TSMC 90nm technology with Europractice facilities [20]. Itself, this technique finds different applications: from stock market prediction to some medical data analysis.

The article is divided as follows: Section 2 introduces the CPOG and DFS models, presenting how they can be adopted together for the design of hardware systems. Section 3 reviews the case study used for this work (OPE), highlighting the steps which brought the rationale to FPGA-based design. In Section 4 the ASIC fabricated for the validation of the design-flow is presented and evaluated. Section 5 concludes the article.

2 Scenario-based design with CPOG & DFS

Several hardware structures can be clustered into two parts: the *control* and the *datapath*. The former reads the current state of the system, by looking at the input stimuli coming from the external environment and from the datapath, activates the modules within the datapath for the current state execution, and eventually move the system into a next state. The datapath, instead, is in charge of executing the current state. They both are very important, and due to their interaction a malfunctioning on one of the two may cause the system to give a wrong answer.

Processors fit this view: the control unit is indeed in charge of the activation of the right modules, structurally defined within the datapath, to use for the execution of an instruction. Let us consider the execution of an addition, for instance. When the opcode associated of such an operation is recognised, the modules that are needed to perform the addition are activated. The register file for fetching the operands, the adder for the sum computation and the memory for storing the final result. The implementation of these modules is not important for the control unit. The latter does not need to know if the sum will be given by a simple ripple carry adder, or by a more sophisticated and faster sparse tree adder. What really matters is how these modules interact to each other. The control unit should, indeed, be able to manage them in the proper way. When the addition is done, the control unit moves to the next instruction. Due to the different nature of these two units, it makes sense describing them by using different models. In this article, I use Conditional Partial Order Graphs for the

description of the control unit, and Dataflow Structures for the description of the datapath.

The case study used for this research is an algorithm for the computation of the ordinal pattern encoding [19]. A pipeline composed by n stages is needed for analysing streams of n numbers. By modelling the control unit via CPOG, and the datapath via DFS we aim at building a pipeline which can be adjusted from 4 to 18 stages. For sake of brevity, in this section the design flow will be applied to the design of a pipeline for the computation of algebraic polynomial of different degrees. The principle is the same since both the two structures are composed by the same pipeline stage repeated as many times as the length of the streams to support.

2.1 Control unit description via CPOG

Composability is the most important concept when dealing with control unit description. A control unit can be defined as the set of possible behaviours that may happen inside a system. Each behaviour can be fired by a certain key, this activates a number of building blocks that, connected together, guide the Datapath to give a certain answer. All the behaviours, or *scenarios*, that compose a system share the same building blocks, or *operations*. If a scenario does not include some operations within itself, it does not mean that the system does not have to deal with those blocks when executing that particular scenario. Conversely, those block must be turned off at any time throughout the execution of the scenario. The notion of sharing is important because the control unit is the composition of all its internal scenarios, and their shared operations. When dealing with the design of a control unit, the operations (basic blocks) must first be identified. Afterwards, their dependencies must be enclosed into different scenarios which will be eventually enclosed into the highest level entity, the control unit. In Figure 1 the hierarchy between these three blocks is depicted. The operations set only contains the blocks which will implemented in the Datapath, Scenarios set contains also the dependencies between operations and the control unit set associates the scenarios with an unique key.



Figure 1: Control unit hierarchy blocks.

Conditional Partial Order Graphs [13] is a graph-based formalism convenient for representing scenariobased systems. The scenarios, their keys, operations and dependencies can be drawn in a graphical-friendly form with a sound math basis. In Figure 2 a CPOG composed of two scenarios is depicted. $\{a, b, c, d\}$ are the operations which the graphs share, the arcs represent the dependencies between them and the variable *x* encloses the key associated to the behaviours: if x = 1 the scenario on the left-hand side is activated, if x = 0 the one on the right is activated instead.

The conditional partial order graph at the top of the figure can be used for the synthesis of a hardware microcontroller. The latter, connected to a datapath that contains the operations defined $\{a, b, c, d\}$, that the control unit captured, can be eventually turned out into a working hardware system.

The main case study of the article is a self-timed reconfigurable pipeline that implements the OPE (see Section 3). For sake of brevity, I will focus on the design of a reconfigurable asynchronous pipeline that, given



Figure 2: CPOG composed of two scenarios.

a stream of values x_i computes corresponding polynomials $p_i = a_n x_i^n + ... + a_1 x_i + a_0$ and possibly aggregates the values on the fly, that is, computing $q_i = p_0 + ... + p_i$. The degree *n* and coefficients a_i of the polynomial, as well as the aggregation function, are free parameters that can be specified during the reconfiguration stage [23]. The design of a reconfigurable pipeline for the computation of polynomials of degree 5, 4 and 3 will be shown.

The pipeline is composed by the same operation repeated as many times as the degree of the algebraic polynomial that we achieve to compute. The operation, named *mad* which stands for *multiply and add*, is described in the next section with the Dataflow structure. This is instantiated *n* times, according to the length of the stream to process. This approach for the polynomial computation is based on the Horner's method [22]. The polynomial is computed by subsequent multiplications and additions according to the formula below.

$$p(x) = a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-1} + a_n x)))$$

The goal is building a pipeline able to selectively compute the three functions below:

$$f(x) = a[g]x^5 + a[h]x^4 + a[i]x^3 + a[j]x^2 + a[k]x + a[0]$$
(1)

$$f(x) = a[g]x^4 + a[h]x^3 + a[j]x^2 + a[k]x + a[0]$$
(2)

$$f(x) = a[g]x^{3} + a[h]x^{2} + a[k]x + a[0]$$
(3)

Mad should be therefore instantiated 5 times inside the datapath, these will be shared between three scenarios, represented by the three functions above. The dependencies between the operations are straightforward: the data stream should simply go through the pipeline of multiply and add modules one by one. The three scenarios are depicted in Figure 3. The partial order 1 models the behaviour of the Equation 1, the remaining ones of the Equations 2 and 3 respectively. The three scenarios can be composed into the CPOG at the bottom of the Figure 3.

Two variables are needed the three scenarios selectively: $\{C_0, C_1\}$. In this case, the first partial order is



Figure 3: CPOG & scenarios for the computation of algebraic polynomials from third to fifth degree.

encoded with the code $\{1,1\}$, the second one by the code $\{0,1\}$ and the third one by $\{1,0\}$. When the latter code is selected, for instance, the operations mad2 and mad3 are off and the polynomial computed has degree 3 (Figure 4). The control signals that this CPOG generates will feed the Datapath, described in the next section.



Figure 4: CPOG when code $\{1,0\}$ is selected.

The hardware synthesis from the CPOG, the scenarios encoding and graph visualisation plugins are available in the Workcraft framework. The latter also supports algebraic notation for entering CPOGs [12]. They have been extensively tested over several benchmarks [9][10][11], this guarantees the reliability of the design.

2.2 Datapath description via DFS

On the other side of the architecture there is the datapath, where the main computational units reside. Each unit can be described singularly via the DFS model, which, together with its dynamic extension, can be used for the description of more units interconnected together. This is due to the conditional activations of some nodes brought by the dynamic DFS extension. The Dataflow representation contains two different kinds of elements: the *static* and the *dynamic* nodes (Figure 5). The former can be either register or combinational nodes. These are used for representing single units as could be an adder, a linear feedback shift register (LFSR), etc. The latter, instead, is composed of elements (control, push, pop) which are meant to control the data paths, dynamically connecting or disconnecting some particular parts of the architectures. A *token* models a data item which goes through these elements. The structure fits well to the representation of asynchronous circuits, as the token game naturally follows the rules of 4 phases dual-rail self-timed protocol [24]. The latter has been used for our main case study (Section 3). Readers can find more information about the model and its math basis in [15].



Figure 6: Reconfigurable pipeline for polynomial computation.

In Figure 7(a) a 4-bit linear feedback shift register is depicted as an example. The second and fourth registers are connected back to the first through a xor gate. This circuit can be used for the pseudo random generation of cyclic set of *n* values, where *n* depends on the number of the registers composing the unit. This component can be easily modelled via DFS (see Figure 7(b)) resulting in an asynchronous 4-bit LFSR. Each register is doubled due to the need for a master and a slave register of the 4-phases dual rail protocol. The connections to the *XOR_GATE* are similar to the one in Figure 7(a), even though in the self-timed counterpart the slave register are the one in charge of propagating the data item back to the left-most register via *r0*. The \bullet symbol contained inside the master registers model the data tokens. DFS is indeed very convenient for the initialisation of asynchronous components, a simulation can be run in Workcraft in order to make sure that the structure works finely.

In order to show how a reconfigurable pipeline can be modelled I will refer to the polynomial example started in the previous section. It has been shown how the control unit can be described via the CPOG model. The actual computational units should now be described and then connected each other, in such a way to come up with a pipeline where some of the stages can be activated and deactivated at times. In Figure 8 the template of one stage of the pipeline is depicted. The stage is composed by four main elements: the input/output register which are in charge of propagating the values in and out the pipeline stage. The combinational logic block which implements the operation that needs to be performed (MAD in our case). The sequential logic register that simply stores the result computed. And finally the demultiplexer and multiplexer. These latter elements abstract the dynamic nodes shown in Figure 5. When the path addressed by a logic 1 is selected by the control coefficients, which also control the CPOG-based control unit, the modules enclosed between these elements are



Figure 7: 4-bit LFSR modelled via DFS.

activated. In Figure 8(b) the dynamic components hidden by the *demux* and the *mux* are shown. The *control* register activates the bottom path (including the combinational node) if controller by a logic 1, and the top path if by a logic 0.



Figure 8: One stage template of polynomial reconfigurable self-timed pipeline.

Once that the template of a pipeline stage has been described, let us focus on the description of the polynomial example. First of all, the MAD unit has to be modelled (Figure 6(b)), this has to perform the multiplication and the addition by a constant value for the computation of px + a. This module is placed 5 times inside the *Polynomial calculator* Figure 6(c). The latter implements the reconfigurable pipeline. For sake of simplification the dynamic nodes in Figure 5 have been abstracted by multiplexers and demultiplexers. The *Polynomial calculator* is eventually integrated within the *Function calculator e/o aggregator*, in Figure 6(a), which is in charge of the aggregations of the values on the fly: $qi = p_0 + ... + p_i$.

The flow described in this section can be used for the design of reconfigurable asynchronous pipelines, and of all the structures where the control and the datapath sides can be described separately. In the next section, the OPE case study will be presented and used for the validation of this design approach presented.

3 Theory and Maxeler-based implementation

Ordinal pattern encoding (OPE) is an operation that can be applied for the analysis of data streams. In [19], a reconfigurable pipelined accelerator for the execution of this operation is presented. This represents the main case study of this research. The accelerator, accordingly modified to fit our needs, has been first implemented into a FPGA-based board [21], and then into a an ASIC.

OPE definition [19]. Given a sequence of *n* distinct values $b = (b_1, ..., b_n)$, the ordinal pattern of *b* is mathematically described by a permutation $\pi = (k_1, ..., k_n)$ such that $b' = (b_{k_1}, ..., b_{k_n})$ is in ascending order.

For instance, let us consider the data stream b = (58, 20, 56, 10, 22) for subsequences of length 3. Three subsequences have to be considered: $b_1 = (58, 20, 56)$, $b_2 = (20, 56, 10)$ and $b_3 = (56, 10, 22)$. The result for the three streams are: $\mathscr{OP}_1 = (2, 3, 1)$, $\mathscr{OP}_2 = (3, 1, 2)$ and $\mathscr{OP}_3 = (2, 3, 1)$. The ordinal pattern results contain information about the regularity of the stream b, that can be used for some analysis. The number of *s*-length subsequences that can be found in a data stream composed by *n* uncorrelated values is equal to n-s+1. The OP requires the streams to be sorted, which has a negative impact either in terms of area or speed in hardware [25][26]. The algorithm described in [19] converts data streams into *Lehmer code*, which is in turn compressed via a factorial number representation to provide the result.

Lehmer code definition [19]. Let $x = (x_1, ..., x_n)$ be a sequence of length *n*, the Lehmer code of *x* is also a sequence with length *n* in the form of $\mathscr{L}(x) = (l_1, ..., l_n)$ where:

$$l_i = #\{x_j : j < i, x_j < x_i\}$$

For instance, below is shown how to compute the Lehmer code of x = (25, 35, 12, 89, 2):

$$\begin{aligned} \mathscr{L}(x_1) &= 0, \quad \Rightarrow \quad \nexists x_j : j < i, x_j < x_i \\ \mathscr{L}(x_2) &= 1, \quad \Rightarrow \quad x_j : j < i, x_j < x_i = x_1 \\ \mathscr{L}(x_3) &= 0, \quad \Rightarrow \quad \nexists x_j : j < i, x_j < x_i \\ \mathscr{L}(x_4) &= 3, \quad \Rightarrow \quad x_j : j < i, x_j < x_i = x_1, x_2, x_3 \\ \mathscr{L}(x_5) &= 0, \quad \Rightarrow \quad \nexists x_j : j < i, x_j < x_i \end{aligned}$$

Assuming, at first, the $\mathscr{L}(x) = (0,0,0,0,0)$ and the first value of the stream to be fed into the pipeline, 2, not to be in yet. The below Formula explains how each digit of *mathcalL* is computed at each iteration:

$$\mathscr{L}(l_i) = \begin{cases} l_i + 1 & \text{if } x_i > x_0 \\ l_i & \text{otherwise} \end{cases} \Rightarrow \mathscr{L}(x) = (0, 1, 0, 3, 0)$$

Lehmer code can be eventually used for the computation of the OPE. In this section, the implementation in the Maxeler desktop [21] (based on a FPGA-based board) is described.

3.1 Maxeler-based desktop implementation

The final goal is the design of an asynchronous reconfigurable pipeline, willing to support different stream lengths for the OPE. The reconfigurable accelerator has been first implemented in the Maxeler workstation [21]. This is a desktop which contains either a FPGA-based board, where digital circuits can be synthesised, and a CPU, able to run software. The hardware description of the circuits supposed to be implemented into the

FPGA is abstracted by a friendly java-based language, easy to learn and use. The CPU, connected toward the internal FPGA-board by default, provides a quick and way to test the hardware implementation. These reasons encouraged me to the usage of this architecture as a test environment before going for the final ASIC implementation. The pipeline is synchronous in this implementation, and reconfigurable upon code compilation.

The design is divided into two separated parts. The C-code run in the CPU and the hardware implementation of the pipeline synthesised into the FPGA. The software compiled for the CPU has a couple of purposes. Firstly, it is meant to randomly generate data streams to feed the pipeline. Secondly, it collects the final result generated by the FPGA for checking the correctness. The test vectors, indeed, are also used by the CPU for computing the OPE in software (a C-based implementation of the algorithm is indeed present in the CPU for the computation of the OPE). The equality of the two results represents a good proof that the value generated by the pipeline is correct. Either the pipeline and the random stream length can be adjusted upon complication.



Figure 9: OPE HW structure for streams of length 4. Figure taken from [19] and modified.

The FPGA, instead, contains the hardware structure for computing the Lehmer code, which can be used for the computation of the OPE. The structure of the pipeline is shown in Figure 9. The original accelerator described in [19] contains an array of multipliers for compacting the final result using the factorial representation. This simplifies the computation of the OPE. In my implementation, instead, the *Packed result* is obtained with the concatenation of all the *Lehmer buffers*. The size either of the buffers and the results is given by the length of the stream to analyse. This implementation requires an increased number of output pins but less area, we also used it into the ASIC presented in the next section.

4 ASIC implementation & evaluation

We apply the proposed method to the design of an asynchronous dataflow accelerator for reconfigurable ordinal pattern encoder (OPE) [19]. The first stage is always included in the pipeline, the remaining stages are reconfigurable. Using the developed WORKCRAFT plugin, we could visually simulate and formally verify the reconfigurable OPE pipeline at the abstract technology-independent level and with data represented by abstract tokens. Several cases of deadlock and non-persistent behaviour (mostly due to incorrect initialisation of control registers) were identified, analysed and corrected during the design process. The top-level schematic of the chip designed is shown in Figure 10a. It comprises two implementations of OPE pipeline, *static* and *reconfigurable*, that are activated by the config input. The former is implemented as a 18-stage pipeline and computes a permutation entropy parameter for 18 last numbers in a stream of input data. The latter can be configured to have from 3 to 18 stages.



Figure 10: Ordinal pattern encoding chip [15].

The chip can be used either in *normal* or *random* mode, as selected by the mode input. In normal mode a stream of input data supplied via in port are processed and the results are produced at the out port at every iteration.



Figure 11: Computation time and energy consumption under different voltages [15].

In random mode a series of count random numbers are produced by a linear-feedback shift register (LFSR) based on user-defined seed. A checksum of the output stream is calculated in accumulator and a single data item is produced when all the generated data is processed. This mode is convenient for testing both the chip performance and functionality, as there is no overheads on interfacing the chip to the testbench environment. The produced checksum can be validated against the output of OPE behavioural model initialised with the same

seed and count parameters.



Figure 12: Power consumption at changing supply voltage [15].

The chip floorplan and its main components are shown in Figure 10b. It has been fabricated using Europaractice facilities in TSMC 90nm CMOS gate family for low-power applications [20].

A custom PCB was developed to interface the packaged chip with a XILINX VIRTEX 7 FPGA to run the tests and check the correctness of the results. A series of experiments was run, each in random mode for a stream of 16M LFSR-generated numbers, at supply voltages from 1.6V down to 0.3V. The computation time was measured by the FPGA with 1ms precision. The power consumption was monitored using KEITHLEY 2612B SYSTEM source meter [27], with 1nW accuracy. The testbench setup is shown in Figure 10c.

Being completely asynchronous, the chip can operate in a wide range of voltages, dynamically adapting its speed. The computation time and energy consumption are characterised in Figure 11 for supply voltages from 0.5V to 1.6V. The length of reconfigurable pipeline (dashed lines) is set to the maximum value and matches that of the 18-stage static pipeline (solid lines). Both the computation time and the consumed energy are normalised to the corresponding measurements of static pipeline at the nominal voltage of 1.2V (the reference values are 1.22s and 2.74mJ, respectively). As expected, the lower the voltage the slower, but at the same time more energy-efficient, is the circuit. The energy consumption of reconfigurable implementation is slightly higher (5% overhead) due to the additional control logic for managing pipeline configuration. The high computation time of reconfigurable pipeline (36% overhead) is due to inefficient implementation of synchronisation between the stages using a daisy-chain C-element structure. This can be significantly improved (by our estimates below 10% overhead for configuration logic) using a tree-like C-element implementation (the same as in static OPE pipeline).

All configurations of reconfigurable OPE pipeline (from 3 to 18 stages) were exercised for 0.5-1.6V supply voltages. The experiments show that both the computation time and the energy consumption increase linearly with the length of the pipeline; the slope of increment is reverse-proportional to the supply voltage.

Another experiment demonstrates the capability of asynchronous pipelines to operate at fluctuating voltage supply down to the near-threshold values. Figure 12 shows the power consumption of the reconfigurable OPE pipeline (all 18 stages activated) during a single LFSR-generated experiment. At the very beginning (left side of the graph), the voltage is set to 0.5V. Here the circuit does nothing and the power consumption is due to

the leakage current of the cells. Then, the up spike represents the beginning of computation – the cells start switching and consume more power. Throughout the experiment, we gradually decreased the supply voltage down to 0.34V (the circuit starts malfunction at lower voltage). At this voltage the chip operation is frozen (the chip can be left at this voltage for hours with no progress being made. When the voltage is raised up again the circuit completes the remaining part of computation (down spike) without errors – the produced result is as expected. Note that in order for the chip core to communicate with the IO cells, the core voltage must be at least 0.5V (the threshold voltage of the IO cells).

In Figure 13 the variability of the computation time, power consumption and the consumed energy of the test is depicted. The reconfigurable OPE pipeline with 18 stages is taken into account under different voltages. The test is started at the instant 0 of the x-axis, and finishes at a different time which depends on the voltage which powers the core of the chip. The energy consumption (depicted under each curve) decreases with the voltage.



Figure 13: Power, energy and time absolute values of reconfig. OPE 18 stages under different voltages.

5 Conclusion

This article shows how to systematically design a hardware scenario-based system via two graph-based models: CPOG and DFS. The former is used for the design of the control unit, while the latter for the datapath. Some room for improvement is still present: new encoding techniques targeting specific parameter (area, power, latency) optimisation can be developed, for instance. However, the tool-chain is mature enough and can be therefore used. Conditional Partial Order Graphs provides a friendly graphical user interface for the development of scenarios, their composition, encoding, synthesis and technology mapping processes. Dataflow Structures can whereas be used to model hardware components, for the simulation and verification [15]. The tools are embedded in Workcraft [17][18].

The case study chosen for this work is an asynchronous reconfigurable pipeline. The chip has been tested and verified in Section 4. The tests demonstrate the high degree of flexibility and reliability that the manufactured self-timed chip has with respect to a range of supply voltages. This affects its performance and energy consumption, which can be tuned meeting the user needs. Designing reconfigurable pipelines is proved to be feasible also with asynchronous protocols. The simplification in the design of reconfigurable pipelines, that this DFS-based design approach fosters, helps reducing the performance/flexibility trade-off between ASIC and flexible architectures (FPGA-like). Furthermore, the correct behaviour of the chip provides a good proof that the DFS model tailors well the 4-phases protocol used for the pipeline. Components such as the LFSR (Figure 7), and the accumulator match exactly the representation via DFS.

CPOGs verification [28] represents one of the research direction we would like to inspect. The compositional graph derived by the single scenarios can be only simulated through the generation of a circuit, up to this point. Having a tool for checking graph consistency (only deadlock-free graphs can be valid), or the Boolean equations generated would give one more reason to adopt the presented design-flow. We also would like to perform some electromagnetic measurements over the chip fabricated. These will hopefully demonstrate the wide frequency variability that self-timed circuits produce. One of the reasons which might bring them to be used for highly secure applications (bank card chip). Having low electromagnetic interference chips is particularly useful for increasing the reliability of hardware systems. Peaks in the frequency spectrum of a generic circuit may indeed cause errors in other close electronic circuits.

Acknowledgment I would like to thank all the μ Systems Research Group members at Newcastle University for supporting us and this research. In particular: Andrey Mokhov, Danil Sokolov, Alex Yakovlev, Paulius Stankaitis, Reza Ramezani and Maxim Rykunov.

References

- A. Reid. "ARM's Architecture Specification Language". Github blog post: https://alastairreid. github.io/specification_languages/
- [2] A. de Gennaro, P. Stankaitis. "Towards a Systematic and Automated Approach in the Design of Processor Instruction Set". Technical Report Series NCL-EEE-MICRO-TR-2015-200, December 2015.
- [3] Weidenhaupt K., Pohl K., Jarke M., Haumer P. "Scenarios in system development: Current practice". IEEE Software 15. (1998). Pages: 34 45.
- [4] Wenrui Li, Zhijian Wang. "Comparing the Local or Global Synthesis of State-Based Specifications from Scenario-Based Specifications". International Conference on Computational Intelligence and Software Engineering, 11-13 Dec. 2009. Page(s): 1 - 4. DOI: 10.1109/CISE.2009.5363678. Publisher:IEEE.
- [5] M. Lora, F. Martinelli, Franco Fummi. "Hardware Synthesis from Software-oriented UML Description".
 15th International Microprocessor Test and Verification Workshop (MTV). 15-16 Dec. 2014.
- [6] Damm W., Harel D. "LSCs: Breathing Life into Message Sequence Charts". Formal Methods in System Design 19. (2001). Pages: 45 - 80.
- [7] Telecommunication Standardization Sector of ITU. "Message Sequence Chart (MSC)". International Telecommunication Union, 1999.
- [8] Object Management Group (OMG). "UML 2.1.1 superstructure". http://www.omg.org/spec/UML/2.
 1.1/Infrastructure/PDF/ 2007.

- [9] A. Mokhov, A. Iliasov, D. Sokolov, M. Rykunov, A. Yakovlev, A. Romanovsky. "Synthesis of processor instruction sets from high-level ISA specifications". IEEE Transactions on Computers, volume 63, issue 6, pages 1552 - 1566, 2014.
- [10] A. Mokhov, J. Carmona, J. Beaumont. "Mining Conditional Partial Order Graphs from Event Logs". Transactions on Petri Nets and Other Models of Concurrency XI. Pages 114 - 136. 2016.
- [11] D. Sokolov, A. Mokhov, A. Yakovlev, D. Lloyd. "Towards asynchronous power management". IEEE Faible Tension Faible Consommation (FTFC) Pages 1 - 4. 2014.
- [12] A. Mokhov, V. Khomenko. "Algebra of Parameterised Graphs". ACM Transactions on Embedded Computing Systems, Volume 13, Issue 4s. 2014.
- [13] A. Mokhov, A. Yakovlev. "Conditional partial order graphs: Model, synthesis, and application". IEEE Transactions on Computers, Volume 59, Pages 1480-1493, November 2010.
- [14] A. Mokhov. "Conditional Partial Order Graphs". Ph.D. Thesis, Newcastle University, September 2009.
- [15] D. Sokolov, A. de Gennaro, A. Mokhov. "*Reconfigurable Asynchronous Pipelines: from Formal Models to Silicon*". Submitted for DAC'17.
- [16] D. Sokolov, I. Poliakov, A. Yakovlev. "Analysis of static data flow structures". Fundamenta Informaticae, volume 88,issue 4, pages 581 - 610. Publisher IOS Press. 2008.
- [17] I. Poliakov, D. Sokolov, A. Mokhov. "Workcraft: A static data flow structure editing, visualisation and analysis tool". Petri Nets and Other Models of Concurrency - ICATPN 2007. Pages 505-514, 2007.
- [18] WORKCRAFT homepage. http://www.workcraft.org/.
- [19] C. Guo, W. Luk, S. Weston: "Pipelined reconfigurable accelerator for ordinal pattern encoding". Proc. Application-specific Systems, Architectures and Processors (ASAP), pp. 194–201, 2014.
- [20] Europractice IC website: "TSMC 90nm technology overview". http://www.europractice-ic.com/ technologies_TSMC.php?tech_id=90nm.
- [21] MaxWorkstation. https://www.maxeler.com/products/desktop/
- [22] Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., Stein Clifford (2009). "Introduction to Algorithms" (3rd ed.). MIT Press.
- [23] M.Rykunov, A.Mokhov, D.Sokolov, A.Yakovlev. "Dataflow Computation a la Carte". Poster presented at the IEEE International conference on Application-Specific Systems, Architectures and Processors (ASAP), June 2013.
- [24] J. Sparsø, S. Furber: "Principles of asynchronous circuit design: a systems perspective". Kluwer Academic Publishers, 2001.

- [25] J.Martinez, R.Cumplido, C.Feregrino. "An FPGA-based parallel sorting architecture for the Burrows Wheeler transform". Proceedings of International Conference on Field-Programmable Technology, 2005, pp. 295-296.
- [26] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. "*Parallel FPGA-based implementation of recursive sorting algorithms*". International Conference of Reconfigurable Computing and FPGAs, 2010.
- [27] Keithley 2612B system source meter data sheet. http://www.testequipmentdepot.com/keithley/ pdfs/2600b_datasheet.pdf
- [28] A. Mokhov, A. Yakovlev. "Verification of conditional partial order graphs". Application of Concurrency to System Design, pages 128 137. 2008.