Design automation for cryptographic hardware using functional languages

Davy Wolfs[1,3,*] Kris Aerts[2,3] Joren Moelans[1,3] Nele Mentens[1,3]

[1] Katholieke Universiteit Leuven, ESAT, SCD/COSIC
Kasteelpark Arenberg 10 - bus 2446, 3001 Leuven, Belgium
[2] Katholieke Universiteit Leuven, CS, DTAI
Celestijnenlaan 200a - bus 2402, 3001 Leuven, Belgium
[3] Katholieke Hogeschool Limburg, IW, ACRO/ES&S
Agoralaan Gebouw B bus 3, 3590 Diepenbeek, Belgium
{davy.wolfs,kris.aerts,joren.moelans,nele.mentens}@khlim.be

Abstract

This paper presents the results of the first phase in developing a user-friendly cryptographic hardware generation tool. The tool has to decrease the design time of cryptographic co-processors and allow designers to make abstraction of both the arithmetic and design complexity. Our approach is unique as we will generate a set of multiple solutions for one design case and then benchmark them to select the most optimal. It is our conjecture that hardware description languages based on a functional programming language, such as Lava offer the best results. We have written a very limited subset of the tool directly in VHDL, in SystemC and in Lava. We show that the currently developed framework with limited functionality is extensible, allowing hardware designers to make abstraction of both the arithmetic and design complexity.

1 Introduction

Developing cryptographic hardware is considered a tough job because of the mathematical complexity of the algorithms and the fact that one needs to address the resistance against implementation attacks as an extra design dimension [1], [2]. Moreover, cryptographic hardware often needs to be designed with as little overhead as possible because it does not contribute to the core functionality of the application. Because of the complexity of the algorithms and the design space, the automated design of cryptographic hardware is a research domain that is greatly beneficial to hardware builders. This paper presents the first results in the development of a tool for the automatic generation of cryptographic hardware through the use of functional programming.

Although there have been various attempts in the last decades for the automatic generation of hardware through functional programming, the level of efficiency of the resulting hardware was the main reason that most attempts remained unsuccessful. Our solution solves this problem by taking into account design dimensions such as area, speed and power consumption from the first step in the design flow. This approach is feasible because we focus on a domain-specific solution, namely public-key cryptography based on finite field arithmetic. Another key issue is that we generate different solutions for the same problem and benchmark them through a third party synthesis tool.

* Davy Wolfs was funded by BOF-project CREA/09/016 of the Katholieke Universiteit Leuven.

In the first phase we have developed a very limited subset of the tool, where, simply speaking, only modular adders can be generated. The interesting part of this approach is that we have done this directly in VHDL, in SystemC and in Lava, a functional hardware description language based on Haskell. The comparison of these variants shows that Lava provides better results with more opportunities for abstraction and generalization.

The paper is organized as follows: Section 2 poses the research question we want to address within this paper, with Section 3 giving an overview of related work. Section 4 and Section 5 describe the methodology and the results of the first phase. Finally, Section 6 lists our conclusions.

2 Research question

How does the implementation of a parameterizable modular adder in VHDL, SystemC and Lava, its generated code and its synthesized hardware compare to each other?

In the introduction we explained that our longer term goal is to develop an automated system for design exploration of cryptographic hardware. We believe that functional programming can be an important success factor in such a project. However, before we proceed, we want to establish a solid understanding of both the chances and the pitfalls. Therefore we have implemented a parameterizable modular adder in Lava and compared it with the implementation in two popular, main stream hardware description languages, VHDL and SystemC.

In this comparison both the implementation effort (the time spent, the expressiveness of the language, the flexibility in parametrization and the supported level of abstraction) and the resulting hardware (opportunities for verification and quantitative results such as speed and silicon area occupied) are taken into account. This paper reports our findings and serves as the basis for a further exploration of hardware description using functional languages.

3 Related work

VHDL and Verilog are probably the two most used languages for describing digital hardware, but as explained above, developing cryptographic hardware is considered hard in these languages. Therefore alternatives have been developed where a 'standard' programming language is augmented with hardware description primitives, or where the language or a subset of it, can be transformed to VHDL, Verilog or directly to a netlist. Probably because it is already known for its hardware capabilities (especially for micro controller programming) C/C++ is often used as the basis, e.g. in SystemC, HandleC or SpecC ([3], [4]). This has been successfully applied in education for designing embedded systems [5], [6]. Also Java, the popular object oriented language, has been the source language, but the best known effort, JHDL [7], already dates from 1998.

In this paper we are also exploring the path of using a functional programming language. This idea has already been popular in the 80s but did not have a lot of influence in actual practice. [8] gives an authorative overview of this. In the last decade a renewed interest has given birth to functional hardware description languages such as Lava (with variants from Chalmers [9], Xilinx [10], York [11] and in 2009 Kansas Lava [12]), Hawk [13], ForSyDe [14], aimed at generating 'a' solution.

Currently non-functional requirements are being researched in languages such as Reflect [15] where reflection features enable verification and theorem proving, or Wired [16] where resulting circuit descriptions capture detailed layout, including the size and positions of wires.

A related and interesting, but nevertheless fundamentally different approach is Cryptol [17], "the language of cryptology", that can generate reference implementations

in a variety of industrial languages, such as Java, C# and the functional programming language Haskell for software implementations. Cryptol can also generate VHDL for hardware, with generated code more designed for readability instead of efficiency. So where Cryptol can output to Haskell, we have a different approach, i.e. we use Lava, which is embedded in Haskell, to generate architectures with optimizations towards a number of implementation parameters.

4 Methodology

In the first phase of the project, we have developed a very limited subset of the tool, where, simply speaking, only modular adders with one design space variable can be generated. The modular adders are implemented for the NIST-recommended elliptic curve over prime field P-192. In this case, modular reduction is performed using the modulus P = 2^{192} - 2^{64} - 1. As the design space variable we have selected the width of the data path. We wanted to be able to implement the hardware for a data path width of 8, 16, 32 and 64 bit. The hardware should implement Algorithm 1.

Algorithm 1 Algorithm for modular addition/subtraction on a w-bit data path

Require: a; b; prime p; $0 \leq a, b < p$; data path width w; $s = \left(\frac{\#bits\, p}{w}\right)$
Ensure: $t = a \pm b \pmod{p}$
  $C = 0$; $x = a$; $y = b$
  for $i = 0$ to s **do**
    $(C; S) = x[i] \pm y[i] + C$
    $t[i] = S$
  end for
  $q = t$; $x = t$; $y = p$
  for $i = 0$ to s **do**
    $(C; S) = x[i] \mp y[i] + C$
    $t[i] = S$
  end for
  if $t < 0$ **then**
    return q
  else
    return t
  end if

The system has been written directly in VHDL, in SystemC and in Lava. VHDL is, along with Verilog, one of the established hardware design languages. SystemC and Lava are both an extension of their base language C++ and Haskell respectively. They both offer the means to describe a hardware circuit. We have chosen Chalmers Lava as our functional language, based upon the fact that they offer the most extensive tutorial [19].

As the modular adder that we have implemented is a rather basic structure, we intentionally used only those (basic) features of the language that can be mastered within one month of learning. This period is long enough to get past the initial learning curve and to be able to draw relevant conclusions, but still short enough not to spend too much time on development that will not be used in the final tool.

We make a distinction between the **producing code**, written in the previously mentioned languages (*Can we code the algorithm in a fast and flexible way?*), and the **produced code** (*How efficient is the generated hardware?*). For a fair comparison, the produced code has two requirements:

1. It has to be technology independent as we do not want to be limited to a specific device vendor or technology library, and we want to evaluate the properties of the generated hardware, not of the underlying technology.

2. It has to be the same target language for all producing codes, to prevent us from comparing apples with oranges.

The last requirement is the main reason we have chosen VHDL above Verilog as the produced language as Lava only exports directly to VHDL. We have chosen SystemCrafter SC as compiler to translate the SystemC code to VHDL. To benchmark the produced code, we needed a synthesis tool to translate our design into a netlist. We have opted for Synopsis Design Compiler and synthesized the code for On-Semi C035M-D, the 0.35 μm digital standard cell technology of On-Semi. As the synthesis tool adds an interpretation level to the design, we parsed all produced code from the three languages into the same tool.

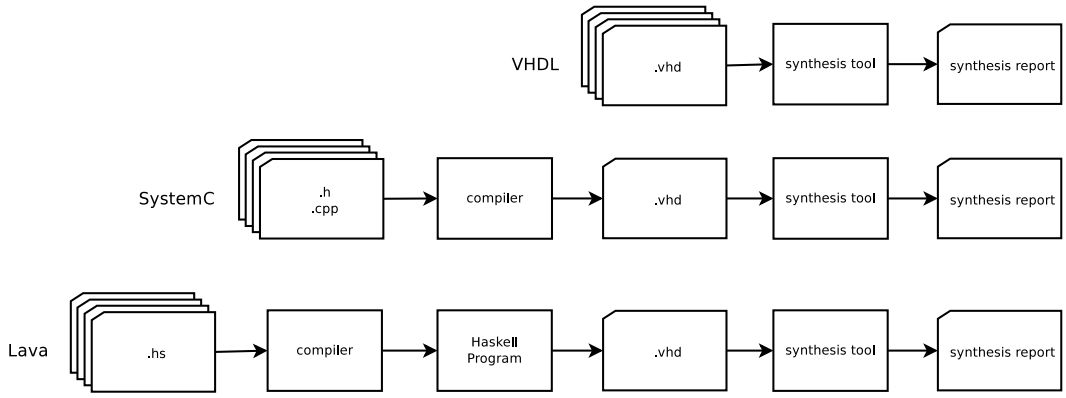


Figure 1: Design flow for VHDL, SystemC and Lava implementations

Fig. 1 shows the design flow for each language. VHDL offers us the simplest design flow. We simply present our written .vhd files to the synthesis tool and generate the netlist. The SystemC code has to be translated into VHDL first, adding an additional interpretation level to the design. Lava offers the export to VHDL as part of the code set. This means that, unlike SystemC, we do not need a special compiler to convert our design into VHDL. After we have called the writeVhdl function in Lava, we feed the produced VHDL code into the synthesis tool for evaluation.

5 Comparison

5.1 Expressiveness

In VHDL and SystemC hardware can be designed both in a RTL (register transfer level) and behavioral model, and in SystemC this nicely blends with the well known C++ features such as modules, constructors and methods. However only a subset of the language can actually be synthesized. Unfortunately, the standard for this subset is still in draft [18]. This results in compilers that often only partially support the synthesis subset draft, and in developers who only scarcely use the expressive features of SystemC, because doing too much might result in code that can not be synthesized.

Lava starts at the RTL level because the export function to VHDL only understands low level gates and registers, but it is very easy to reach a functional behavioral model in just a few steps by using connection patterns and recursive functions, resulting in

compact, but very concise and readable code. SystemC also tends to be very readable, but not as compact as in Lava, whereas in VHDL the readability and complexity depends on the qualities of the designer. When creating a RTL design, VHDL and Lava are both readable, while SystemC adds more complexity as every gate has to be within a process.

VHDL and Lava are strongly typed, giving early errors during development and ensuring a more predictable hardware implementation than in SystemC.

5.2 Dynamic adaptation of generic capabilities

The main requirement for doing design space exploration is the code's capability to handle generic definitions, i.e. if we need to change the data path width, it is imperative that we only have to do this at one line in the code. Furthermore, the dynamic adaptation of these generic capabilities is a must.

All three languages have generic capabilities. VHDL uses the keyword **generic** in the entity declaration while SystemC can rely on the **template classes** that are built into C++. However, SystemC is limited in generating a generic number of hardware components. SystemC demands a unique name for every instance of hardware added to the design. This name can only be given when calling the instance constructor. If we need a generic array of hardware instances, we need to create an array of pointers, of the generic size, to these instances. When invoking the top instance constructor, we can then create the number of instances defined by the generic. However, pointers cannot be synthesized, which makes this perfect compilable syntax useless for hardware generation. Lava handles everything as a function, meaning that the generic capability is simply another parameter for the function. The main difference is that for VHDL and SystemC, the generic parameter has to be defined prior to the hardware description while with Lava you have to define the generic parameter when you actually export to VHDL. The dynamic adaption of generic capabilities is impossible in VHDL. The generic parameters need to be changed externally.

SystemC demands that the template classes contain constants, making the direct adaptation of the generic parameter also impossible. It is possible to write a C++ routine that modifies the template within the file. This is not a direct approach, but it is feasible. We do need two compilers to achieve this, one to compile the regular C++ into a program that modifies the design files and a compiler to generate VHDL from these design files.

In Lava we can also generate a list of hardware designs instead of just one design, and using the strong list processing features of Haskell, we can combine these hardware designs into a long list of options where the best option can be selected. Although we have not used this in our limited setup, we explicitly mention it here because of the opportunities this offers for dynamic adaptation.

5.3 Abstraction level of the control path

Where the data path does the actual computation, the control path has to ensure that the data is presented and written back at the correct time. The control path is dependent of the data path and of the desired functionality. We want to make abstraction of the hardware implementation of the control path and focus more on the desired behavior. In our case we have implemented a Moore finite state machine. With VHDL and SystemC the mainstream approach is to write a next state process followed by a sequential process to update the current state with the next one. To make abstraction of the gate level, the states are type enumerated and for each state, the next state is computed based upon the inputs. This process is encapsulated within a case (VHDL) or switch (SystemC) language construct. Within Lava we make abstraction of the language construct and define the states, inputs and transitions in one data

structure. Especially the transition function is very declarative. It takes the current state and the actual input as parameters, and picks the corresponding next state from a list of transition tuples `(state, input, nextState)`

```
transition state input =
  [ nextState | (state', input', nextState) <-
              [ ("sIdle", "start", "sAB")
              , ("sIdle", "skip" , "sIdle")
              , ("sAB", "stepDone", "sABStore")
              , ("sAB", "skip" , "sAB")
              , ("sABStore", "skip" , "sSP")
              , ("sSP", "stepDone", "sCheckSign")
              , ("sSP", "skip" , "sSP")
              , ("sCheckSign", "skip" , "sCheckSign")
              ]
              , state == state'
              , input == input']
```

This data structure is translated to a hardware structure by a generic function provided in [19]. The control signals for the data path are generated purely on the current state of the finite state machine and are written in a separate process or function.

5.4 Verification

With VHDL, verification is done by coding a test bench and using an external simulator such as ModelSim. SystemC offers the means to write your test benches in SystemC and to compile and simulate them as a part of the program. It also offers the possibility to record your simulation as a wave form in .vcd (Value Change Dump) trace files. Lava cannot export such trace files, but offers us a set of tools to do not only simulation, but also check functional equality between different implementations. Lava has safety properties, induction and time transformation functions to verify different implementations on equal functionality. Time transformations offer the means to compare across multiple time domains i.e. pure combinatorial and sequential adders. These features enable us to check if, within our dynamical adapted design space, an implementation is functionally equivalent and has to be exported to VHDL. There have been some initial problems to make the synthesis tool accept the Lava generated VHDL. We have modified the Lava export function to solve this issue. But with the verification mechanism of Lava, we were able to verify correctness prior to synthesis. Please note that the produced code from SystemC and Lava, which is in VHDL, is also verified with ModelSim using the test bench of the VHDL design.

5.5 Pre-layout synthesis

Table 1 shows a summary of our pre-layout synthesis results. We make a distinction between the data path and the modular adder, which includes the control path. The 64-bit implementations were synthesized with a Synopsis DC optimization option for speed, while the 8-bit equivalents were optimized for area. The table lists for each implementation the silicon area it will occupy, the maximum clock frequency and the throughput. The throughput is determined by the number of clock cycles for the computation of one modular addition.

		VHDL	SystemC	Lava	[unit]
Modular adder 64-bit (speed)	area	190117	184145	113479	[μm^2]
	max. clock	100.00	74.07	80.52	[MHz]
	throughput	11.11	8.23	8.95	[MOps/s]
Modular adder 8-bit (area)	area	80679	67701	81377	[μm^2]
	max. clock	73.96	77.10	80.58	[MHz]
	throughput	1.45	1.51	1.58	[MOps/s]
Datapath only 64-bit (speed)	area	183600	123690	104689	[μm^2]
	max. clock	277.78	285.71	217.39	[MHz]
	throughput	30.86	31.75	24.15	[MOps/s]
Datapath only 8-bit (area)	area	74827	60111	76644	[μm^2]
	max. clock	168.07	147.06	185.53	[MHz]
	throughput	3.30	2.88	3.64	[MOps/s]

Table 1: Pre-layout synthesis results

6 Conclusion

On the criterium of the **produced code**, i.e. VHDL in this experiment, we can conclude that they are almost equivalent.

However for the **producing code** there are essential differences. Where VHDL simply does its job, with little opportunities for dynamic design space exploration, SystemC is the only language that offers full abstraction of the hardware, although for complex designs, knowledge of the target technology and desired hardware architecture is imperative. The lack of a synthesis subset standard is a major drawback. The code written might not work with another compiler. More problematic for our goals is the fact that the generic features have to be instantiated at compile time.

Lava offers the most flexible and readable way to program control paths and to do dynamic adaptation of generic capabilities. The latter is a must if one wishes to do design space exploration. Once we have defined our gate level logic, abstraction can quickly be gained by connection patterns and recursive functions. Using the list features a set of functionally equivalent hardware designs can be generated with different non-functional features. The strong type dependency of Lava requires a careful design, ruling out functional faults caused by erroneous type castings. As a surplus, we have full control on how the hardware will be generated. The export functions can be modified to suit ones needs, and actually we are already working on this. With SystemC, the way the hardware will be generated depends on the employed compiler.

References

[1] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems", In N. Koblitz, editor, Advances in Cryptology - Proceedings of CRYPTO, number 1109 in Lecture Notes in Computer Science, p. 104-113, Springer-Verlag, 1996.

[2] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis", In M. Wiener, editor, Advances in Cryptology - Proceedings of CRYPTO, number 1666 in Lecture Notes in Computer Science, p. 388-397, Springer-Verlag, 1999.

[3] http://www.systemc.org/

[4] http://www.ics.uci.edu/ specc/

[5] I. Verbauwhede, and P. Schaumont, "Skiing the embedded systems mountain", ACM Transactions on Embedded Computing Systems 4(3), p. 529-548, 2005.

[6] K. Sakiyama, P. Schaumont, D. Hwang, and I. Verbauwhede, "Teaching trade-offs in system-level design methodologies", In Proceedings of IEEE International Conference on Microelectronics Systems Education (MSE 2003), p. 62-63, 2003.

[7] P. Bellows, and B. Hutchings, "JHDL - An HDL for Reconfigurable Systems", In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, p. 175-184, 1998.

[8] M. Sheeran, "Hardware design and functional programming: a perfect match" Journal of Universal Computer Science 11(7), p. 1135-1158, 2005.

[9] P. Bjesse, K. Claessen, and M. Sheeran, "Lava: Hardware Design in Haskell", In Proceedings of the ACM International Conference on Functional Programming, 1998.

[10] S. Singh, "The Lava Hardware Description Language", http://raintown.org/lava/, 2009.

[11] M. Naylor, C. Runciman, and J. Reich, "Reduceron home page", http://www.cs.york.ac.uk/fp/reduceron/, 2009.

[12] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing Kansas", In Proceedings of the 21st International Symposium on Implementation and Application of Functional Languages, 2009.

[13] J. Launchbury, J. R. Lewis, and B. Cook, "On embedding a microarchitectural design language within Haskell", In Proceedings of the fourth ACM SIGPLAN international conference on Functional programming, p. 60-69, 1999.

[14] Z. Lu, I. Sander, and A. Jantsch, "A case study of hardware and software synthesis in ForSyDe", In Proceedings of the 15th international symposium on System Synthesis, p. 86-91, 2002.

[15] J. Grundy, T. Melham, and J. O'Leary, "A reflective functional language for hardware design and theorem proving" Journal of Functional Programming 16(2), p. 157-196, 2006.

[16] E. Axelsson, K. Claessen, and M. Sheeran, "Wired: Wire-aware circuit design", In Proceedings of the Conference on Correct Hardware Design and Verification Methods, p. 5-19, 2005.

[17] J. Lewis, "Cryptol: specification, implementation and verification of high-grade cryptographic applications", In Proceedings of the 2007 ACM workshop on Formal methods in security engineering, 2007.

[18] M. Meredith, B. C. Schafer, A. P. Su, A. Takach, and J. Verhaegh, "SystemC Synthesizable Subset 1.3 draft", The OSCI TLM-2.0 Standard and Synthesis Subset, 2009.

[19] K. Claessen, and M. Sheeran, "A Slightly Revised Tutorial on Lava: A Hardware Description and Verification System", http://www.cse.chalmers.se/edu/course/TDA956/Papers/lava-tutorial.ps, 2007.