DESIGN SPACE EXPLORATION FOR AUTOMATICALLY GENERATED CRYPTOGRAPHIC HARDWARE USING FUNCTIONAL LANGUAGES

Davy Wolfs [*], *Kris Aerts, Nele Mentens*

KU Leuven, ESAT-SCD/COSIC, Kasteelpark Arenberg 10, 3001 Leuven, Belgium
KHLim, FIIW, ACRO-ES&S, Agoralaan Gebouw B bus 3, 3590 Diepenbeek, Belgium
email: {davy.wolfs,kris.aerts,nele.mentens}@khlim.be

ABSTRACT

This paper presents an EDA (Electronic Design Automation) tool that generates basic building blocks for cryptographic hardware in VHDL. The purpose of the tool is to decrease the design time of cryptographic hardware and to allow designers to make abstraction of both the arithmetic and design complexity. The tool generates multiple implementations for one arithmetic description and then benchmarks the implementations to find the most optimal, based upon design space parameters. These parameters consist of area and speed requirements. We present datapath and control logic results for a Xilinx Virtex-5 FPGA.

The novelty in our approach lies in the fact that we exploit the higher-order features of functional languages to facilitate the design space exploration and that we take benefit from the strength of the third-party synthesis tool by generating VHDL code at an abstraction level that is higher than the gate level. Nevertheless, in this stage of the development of the tool, the different cryptographic architectures are hand-made and the selection of the most optimal solution, based upon user requirements, is done by exhaustive search. This means that the tool leaves room for improvement, but forms a solid base for further development.

1. MOTIVATION AND CONTRIBUTION

In storage and communication of digital data, security is an important issue. To provide confidentiality, authentication, secure key exchange, privacy, ... cryptographic algorithms are needed. When requirements such as high speed, low area, low power consumption and/or high security level are an issue, these cryptographic algorithms are implemented in hardware coprocessors. Most of the times cryptographic hardware needs to be designed with as little overhead as possible because it does not contribute to the core functionality of the application. However, to achieve a sufficient level of security, cryptographic algorithms and in particular public key algorithms consist of rather complicated mathematical operations on large numbers. Consequently, the challenge for the design of cryptographic hardware is to find a suitable architecture for a rather complicated mathematical operation on large numbers that results in a restricted overhead in time, area and/or power consumption.

Like almost all digital designs, a cryptographic hardware architecture consists of a datapath and control logic. In this paper, we explore the design space of both. We are specifically interested in the area and speed of the actual hardware after synthesis by both FPGA and ASIC (commercial) third-party tools. Therefore we make sure that the VHDL code generated by our Lava programs is platform-independent. We also shift from basic logic gates to formal FSMs for the description of the control path, as is common in VHDL programming. These state machines can be simulated in Haskell, and exported to native VHDL FSM code on the behavioral level, which gives more optimization possibilities to the synthesis tool. We apply a similar approach for the datapath, as far as it allows a behavioral description in VHDL.

The paper is organized as follows. An overview of the state-of-the-art is described in Sect. 2. The tool flow is presented in Sect. 3. A more detailed view on two parts of the tool flow, namely the VHDL generation phase and the synthesis phase, is given in Sects. 4 and 5, respectively. Finally, Sect. 6 concludes this paper and looks ahead at future work.

2. RELATED WORK

In the field of cryptography, many hardware implementations have been developed for high speed or low area [1, 2, 3, 4], but the solutions are either too specific to be interchangeable in different systems or too generic to be sufficiently efficient. This implies that for most applications custom cryptographic hardware must be designed. Our tool reduces the design time by using functional languages to automatically generate cryptographic hardware. The idea to use functional programming for hardware design is not widespread in the field of cryptography.

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

However, the idea to use functional programming for hardware design exists since the 80s: Sheeran gives an authoritative overview in [5]. Since 1998 a renewed interest has given birth to functional hardware description languages such as Lava [6] (with variants from Chalmers, Xilinx, York and in 2009 Kansas Lava [7, 8]) and ForSyDe [9]. These examples are usually capable of generating hardware for many application domains. Our work only focusses on the generation of cryptographic hardware, which allows us to make domain-specific optimizations. These optimizations are acquired by the integration of design space exploration in our tool. This is in contrast to older publications, that typically aim at generating 'a' solution, with little attention for non-functional requirements. However, current research starts focussing explicitly on low power consumption and/or high speed [10, 11, 12].

The programming approach in this paper has some similarities with Cryptol [13], "the language of cryptology". Both are based on Haskell and focus on stream programming for cryptographic hardware, but there are some differences. Whereas Cryptol is suitable for the description of almost any existing and new cryptographic function, we focus on a pre-defined subset of cryptographic functions. This allows us to do operation-specific optimizations for this subset. Users of our final tool do not need to program: based on the hardware requirements, the tool itself will explore the design space and will deliver the most suitable hardware. This paper discusses the programming model being used in the tool, not the actual usage of the tool.

3. TOOL FLOW

Fig. 1 shows the interaction of our tool with inputs from the user and third-party synthesis tools. Based upon the algorithmic description of the desired cryptographic functionality in a functional language, VHDL code is generated by our tool and the design space is explored by the generation of a set of different architectures in VHDL. Next, these architectures are synthesized by a third-party synthesis tool for FPGA or ASIC and the design space is explored by calling the third-party synthesis tool under different optimization options. Based upon the synthesis reports and user requirements, our tool selects the optimal architecture. For the design space exploration in both the generation and the synthesis phase, we use the higher-order listing features of Haskell.

4. DESIGN SPACE EXPLORATION IN THE GENERATION PHASE

We use a differentiated approach for the datapath and the control logic in order to allow a coverage of the design space that is as large as possible. This will be illustrated with use case examples in the following paragraphs.

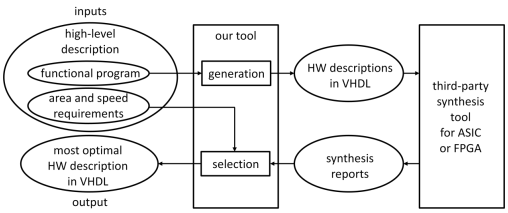

Fig. 1. Interaction of our tool with user inputs and third-party synthesis tools

4.1. Datapath

In our tool, we focus on public key datapaths, for which the first phase of design space exploration is done by generating different architectures in VHDL. The possibilities strongly depend on the type of mathematical unit that needs to be implemented. The second phase of design space exploration is done by calling the third-party synthesis tool for both speed and area optimized synthesis. As a design example, we generate a 192-bit adder, which is a common building block in elliptic curve cryptography. This is done by exporting a set of 192-bit adder architectures to VHDL with an internal datapath width of 8, 16, 32 and 64 bits and with different hardware architectures, namely a ripple carry adder, a carry select adder, a Sklansky adder and an adder for which the synthesis tool chooses the architecture. The three former architectures are described in York Lava and generate VHDL code that consists of basic logic gates. The latter architecture generates VHDL code at the behavioral level, which allows the third-party synthesis tool to employ its full optimization capabilities. This results in the following functional code:

```
buildCircDSE3 (writeSeqAdder "seqAdder")
  [8, 16, 32, 64]
  [192]
  [RippleCarry, CarrySelect, Sklansky, VhdlAdd]
```

This function results in 16 different architectures in VHDL for a 192-bit adder. The synthesis results of these architectures will be presented in Sect. 5.

4.2. Control Logic

The control logic is implemented as a Moore FSM. In our design space exploration, we consider two versions. The first one consists of a traditional architecture in which the transition function computes the next state based on the inputs and the current state. The state value is stored in the state register on each rising or falling edge of the incoming clock. The outputs are computed from the state by means of the output function. The transition function as well as the output function consist of combinatorial logic, while the state register consists of flip-flops. An alternative version

of a Moore FSM contains extra output flip-flops that result in glitch-free output signals and create a buffer between the output of the FSM and the logic driven by the FSM. In order not to introduce an extra clock delay, a so-called look-ahead output function is used that calculates the outputs of the FSM based on the inputs of the state register instead of the outputs of the state register.

In our tool, the first phase of design space exploration for control logic is done by translating a high-level FSM description into both aforementioned architectures of a Moore FSM. Our tool also facilitates the second phase of design space exploration, done by the third-party synthesis tool, by using a different approach than the one used in York and Chalmers Lava [14]. When exporting the FSM to VHDL, both Lavas generate a netlist of basic logic gates (*and*, *or* and *delay*). This is functionally correct, but VHDL synthesis tools are more efficient in optimizing an FSM described in VHDL without explicit state encoding than optimizing a gate level netlist in VHDL. Therefore we expand the Lava framework such that it can export a behavioral VHDL description of an FSM instead of a gate level description, giving more freedom to the third-party synthesis tool for the optimization of the FSM, consisting of a suitable state encoding style as well as the logic optimization of the transition and output functions. Our tool calls the third-party synthesis tool for one-hot, Gray and automatic encoding. For each of these options both area and speed optimized synthesis are done. This results in circuits that have a better area/speed tradeoffs than their York or Chalmers equivalent. As an example, we generate an FSM consisting of 17 states controlling an AES datapath. The type of a generic Moore FSM in our tool is as follows:

```
data MooreStateMachine state inp outp
  = MooreStateMachine
    { moStates     :: [state]
    , moInitial    :: [state]
    , moInputs     :: [inp]
    , moOutputs    :: [outp]
    , moTransition :: state -> inp -> [state]
    , moOutputFunction:: state -> [outp]
    }
```

5. DESIGN SPACE EXPLORATION IN THE SYNTHESIS PHASE

5.1. Interaction with Our Tool

The following code example illustrates the function in our tool that calls the synthesis tool:

```
synthXstCircDSE4 (synthAdder "seqAdder")
  [8, 16, 32, 64]
  [192]
  [RippleCarry, CarrySelect, Sklansky, VhdlAdd]
  [Area, Speed]
```

The example runs FPGA synthesis for the 192-bit adder for all parameter options in the generation phase, i.e. the first three dimensions in the function. The last dimension refers to the design space exploration in the synthesis phase, namely optimizations for low area and high speed.

5.2. FPGA Synthesis Results

Table 1 shows the FPGA synthesis results for the 192-bit adder and the AES FSM. The FPGA synthesis tool called by our tool is Xilinx XST Release 12.1 - M.53d (linux). The designs have been synthesized for a Virtex-5 5vlx20-2 FPGA.

For the 192-bit adders in Table 1, the general trend is that the maximal clock frequency decreases with an increasing internal datapath width. Since the number of clock cycles for one addition is equal to 192 divided by the datapath width, the adder with the widest datapath has the smallest latency. The maximal clock frequency also depends on the architecture. Further, the area of the adder increases with an increasing datapath width. However, because the adder consists of two 192-bit input shift registers and one 192-bit output shift register together with a counter that keeps track of the number of clock cycles, the internal datapath width has a relatively low influence on the total area. The adder described at the behavioral level in VHDL, using the operator '+', shows the best timing and area results. Here, we clearly see the advantage of our approach in including an architecture described at the behavioral level in VHDL. Note that the adders with a datapath of 16 bits described in behavioral VHDL code result in a faster maximal clock frequency than the 8-bit adders because the FPGA contains 16-bit long optimized carry chains.

For FPGA synthesis of the AES FSMs at the behavioral level, i.e. the Moore FSM (Comb) and the Moore FSM with synchronized outputs (Sync), both the encoding style and the optimization goal (speed or area) have an effect on the synthesis results. When the encoding style is chosen automatically by the synthesis tool, this leads to the best results in area and speed. FPGA synthesis of the AES FSM described in gate-level VHDL (Gate), as generated by Lava, does not allow any optimization in the synthesis phase, which makes its area larger and its maximal clock frequency lower. Again, the advantage of generating behavioral VHDL code is clear. Note that the second version of the Moore FSM (Sync) does not lead to any advantage in speed or area. The registers at the output will become important in a larger system when the propagation delay of the output function and the propagation delay of the logic driven by the FSM add up to a delay that is larger than the critical delay.

6. CONCLUSION AND FUTURE WORK

In this paper, we describe a tool for the automatic generation of cryptographic hardware based on functional languages. We do not only take algorithmic requirements into account,

Table 1. Xilinx XST pre-layout synthesis results for a 192-bit adder (left) and an AES FSM (right) on a Virtex-5 5vlx20-2 FPGA. In the vertical and horizontal direction, the different options for the design space exploration (DSE) in the generation phase and in the synthesis phase are given, respectively. Ripple-Carry Adder and Carry-Select Adder are denoted by RCA and CSA, respectively.

		Area		**Speed**	
		Slices	Max.clk [MHz]	Slices	Max.clk [MHz]
RCA	8-bit	594	300	598	341
	16-bit	600	194	626	314
	32-bit	613	103	687	265
	64-bit	646	56	805	220
CSA	8-bit	597	276	602	309
	16-bit	607	234	628	271
	32-bit	628	157	678	240
	64-bit	677	95	634	229
Sklansky	8-bit	594	284	602	334
	16-bit	605	191	630	314
	32-bit	627	130	677	267
	64-bit	702	92	780	191
Behavioral	8-bit	594	321	595	350
	16-bit	600	333	600	355
	32-bit	613	352	613	352
	64-bit	645	286	645	286

		Area		**Speed**	
		Slices	Max.clk [MHz]	Slices	Max.clk [MHz]
Gate	auto	37	857	37	857
	one-hot	37	857	37	857
	gray	37	857	37	857
Comb	auto	30	615	31	905
	one-hot	30	615	31	905
	gray	26	582	26	583
Sync	auto	30	453	34	507
	one-hot	30	453	34	507
	gray	27	362	42	332

but also speed and area requirements. To achieve this, our tool performs an exploration of the design space in order to produce a solution that fulfills the these requirements in an optimal way. We generate VHDL code at the behavioral level in order to allow third-party synthesis tools to use their full optimization capabilities. Results for FPGA synthesis show that this leads to better area and timing results compared to gate-level descriptions generated by Lava.

At the moment, our conclusions are only based on exemplary datapaths and control logic. In order to draw conclusions for cryptographic hardware in general, a more representative subset of cryptographic functions is necessary. Further, there is a lot of room for improvement by adding more intelligence to the generation process of the architectures and the selection of the optimal solution.

7. REFERENCES

[1] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede, "Public-Key Cryptography on the Top of a Needle," in *IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2007, pp. 1831–1834.

[2] T. Güneysu and C. Paar, "Ultra high performance ecc over nist primes on commercial fpgas," in *Cryptographic Hardware and Embedded Systems (CHES)*, ser. LNCS, E. Oswald and P. Rohatgi, Eds., no. 5154. Springer-Verlag, 2008, pp. 62–78.

[3] N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede, "Efficient Pipelining for Modular Multiplication Architectures in Prime Fields," in *Proceedings of the 17th ACM Great Lakes symposium on VLSI (GLSVLSI)*. ACM, 2007, pp. 534–539.

[4] E. Wenger, M. Feldhofer, and N. Felber, "Low-resource hardware design of an elliptic curve processor for contactless devices," in *Proceedings of the 11th Workshop on Information Security Applications (WISA)*, ser. LNCS, Y. Chung and M. Yung, Eds., no. 6513. Springer-Verlag, 2010, pp. 92 – 106.

[5] M. Sheeran, "Hardware design and functional programming: a perfect match," *Journal of Universal Computer Science*, vol. 11, no. 7, pp. 1135–1158, 2005.

[6] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ser. ICFP. ACM, 1998, pp. 174–184.

[7] A. Gill, "Declarative fpga circuit synthesis using kansas lava," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, T. P. Plaks, Ed. CSREA Press, 2011, pp. 55–64.

[8] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing kansas lava," in *Proceedings of the 21st International Symposium on Implementation and Application of Functional Languages (IFL)*, ser. LNCS, M. T. Morazán and S.-B. Scholz, Eds., no. 6041. Springer-Verlag, 2009, pp. 18–35.

[9] 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 (ISSS)*, 2002, pp. 86–91.

[10] K. Claessen, C. Seger, M. Sheeran, E. Shriver, and W. Swierstra, "High level architectural modelling for early estimation of power and performance," Talk at Workshop on Hardware Design and Functional Languages (HFL), http://www.cs.ru.nl/~wouters/Publications/HFL09.pdf, 2009.

[11] A. Gill and A. Farmer, "Deriving an efficient fpga implementation of a low density parity check forward error corrector," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, O. D. M. M. T. Chakravarty, Z. Hu, Ed. ACM, 2011, pp. 209–220.

[12] G. Wright, "Functions ot junctions: Ultra low power chip design with some help from haskell," Talk at Workshop on Hardware Design and Functional Languages (HFL), http://antiope.com/documents/cufp08_wright.pdf, 2009.

[13] J. Lewis, "Cryptol: specification, implementation and verification of high-grade cryptographic applications," in *Proceedings of the ACM workshop on Formal methods in security engineering (FMSE)*. ACM, 2007, pp. 41–41.

[14] K. Claessen, "A slightly revised tutorial on lava: A hardware description and verification system," www.cse.chalmers.se/edu/course/TDA956/Papers/lava-tutorial.ps, 2007.