Advances in Robotics & Automation
ISSN: 2168-9695

Dossis, Adv Robot Autom 2013, 3:2
DOI: 10.4172/2168-9695.1000121

Review Article Open Access

Synthesis of Custom Hardware from ADA with Artificial Intelligence Techniques

Michael Dossis*

Department of Informatics Engineering, TEI of Western Macedonia, Kastoria Campus, Fourka Area, Kastoria, GR 52 100, Greece

Abstract

The advancing complexity of contemporary microelectronics has motivated research in high-level and system synthesis (HLS). Formal and intelligent HLS techniques are presented in this contribution, thus the generated implementation is correct-by-construction. These intelligent techniques include RDF (Resource Description Framework) and logic relations, along with automatic implementation options and they are employed for the transformations of a hardware compiler. The proposed toolset utilizes compiler-generators, RDF rules and logic programming in combination with XML validation of the internal state of the compiler. These intelligent and formal techniques make the whole transformation from source code to implementation, formal. The HLS tool is enhanced with the Parallel, Abstract Resource – Constrained Scheduler, which aggressively optimizes the initial state schedules, into maximally parallelized ones. A number of custom options are applied by the user of this toolset, in order to automatically compile selected testcases from real-world applications which prove the usability of the embedded scheduler and the formal compilation of the intelligent HLS compiler.

Keywords: Microelectronics design; High level synthesis; Formal languages; Electronic design automation; Scheduling; Compilers; RDF; Formal methods; Logic programming; Custom microarchitecture

Introduction

Digital microelectronics found in embedded, high-performance and portable computing systems have highly complex components, design hierarchy and interconnections. During the last couple of decades, commercial and academic organisations have invested in HLS and optimisation techniques, so as to achieve design automation, quality of implementations and short specification-to-product times [1,2]. However, existing HLS tools are not widely accepted by the engineering community because of their poor results, especially for large applications with complex module and control-flow hierarchy. Very often, the programming style of the source code has a severe impact on the quality of the synthesized implementation. For large-scale applications, the complexity of the synthesis transformations (front-end compilation, algorithmic transformations, optimizing scheduling, allocation and binding), increases exponentially, with a linear increase of the design size [3].

Existing HLS tools impose proprietary extensions or restrictions (e.g. exclusion of while loops) on the programming model of the specifications that they accept as input, and various heuristics on the HLS transformations that they utilize (e.g. guards, speculation, loop shifting, trailblazing). Most of them are suitable for linear, dataflow dominated (e.g. stream-based) designs, such as pipelined DSP, image processing and video/sound streaming.

The contribution of this work is an integrated HLS toolset which utilises intelligent and formal techniques so as to apply the source-to-implementation optimizing transformations, thus, the produced hardware implementations are correct-by-construction. Therefore, the design needs verification only at the top behavioral level, without spending days or even weeks, on lengthy RTL or annotated gate simulations. Moreover, various custom options can be applied by the user on the automatic HLS transformation, such as the type of the micro-architecture, the generated HDL code as well as the inclusion of custom (e.g. arithmetic) logic functions throughout the HLS compilation.

The author has designed and developed an intelligent HLS compiler [4] that includes a scheduler of operations into control steps, achieving the maximum functional parallelism in the synthesized implementation [5]. This HLS scheduler called PARCS, utilizes logic programming [6] and RDF subject-predicate-object relations [7], to formally achieve the maximum possible parallelism of operations. In this way, the functionality of the delivered implementations is correct-by-construction [3] explores various scheduling techniques.

Formal HLS techniques are analysed in the next section. Next, the intelligent approach of the prototype optimising CCC synthesizer is described, such as formal predicate logic [6], RDF relations and XML schema validation [7]. Then, the usability and correctness of the CCC HLS toolset are evaluated with a number of benchmarks. The last section draws useful conclusions and proposes future work.

Existing Work in intelligent HLS techniques

Established and well studied HLS tasks include scheduling, allocation and binding [3]. The front-end part of HLS tools include parts of software programming language compilers [8], such as parsing, semantic analysis, intermediate variable optimization, elimination of dead code, etc. The front-ends exchange information with the back-ends using intermediate formats, such as the Electronic Design Interchange Format (EDIF) [9,10], used by most E-CAD tools. Complex control flow optimization has been evaluated in [2,11,12], but for small parts of code and by no means complete application tests [13], discusses synthesis for distributed logic and memory [14] uses communicating processes as a system specification medium. HLS methods that include memory

***Corresponding author:** Michael Dossis, Department of Informatics Engineering, TEI of Western Macedonia, Kastoria Campus, Fourka Area, Kastoria, GR 52 100, Greece, Tel: 30-694-6154078; E-mail: mdossis@yahoo.gr

Received May 27, 2014; **Accepted** June 5, 2014; **Published** June 7, 2014

Citation: Dossis M (2014) Synthesis of Custom Hardware from ADA with Artificial Intelligence Techniques. Adv Robot Autom 3: 121. doi: 10.4172/2168-9695.1000121

access management are outlined in [1], where digital signal processing (DSP) and streaming applications are synthesized using performance constraints [15], analyses mutually exclusive scheduling on extended data-flow graphs (EDFG) [16] synthesizes behavioural descriptions with time constraints, where complex operations are decomposed into simpler ones, and a similar set of decomposed fragments of operators, with the same pattern, are scheduled in a clock cycle.

In [17] an actor, that is used to model every module or system process, communicates with other actors via communication channels. These actors are used by the System Co-Designer [17] to exercise electronic system level (ESL) design space exploration. In [18] the SURYA system utilises the Simplify theorem prover to prove that the RTL model generated by HLS tools is functionally-equivalent to the specification. SURYA found two bugs in the SPARK HLS tool [2], which were until then unknown. In [19] flip-flops are replaced with latches so as to improve implementation timing, since latches are inherently more tolerant to process variations than flip-flops. Nevertheless, latch-based design is more cumbersome than flip-flops.

The W3C Resource Description Framework

The Resource Description Framework (RDF) is a metadata model and is used to model the information of web resources [7]. RDF models include subject-predicate-object relations with explicit statements, called triplets. Triplets specify resources that store knowledge, information and data retrieval in large automated software tools. This is achieved due to the suitability of RDF to capture, store, exchange, and use machine-readable web information. Here, RDF is used to formally model and validate the internal data and state of the author's HLS tool.

The eXtensible Markup Language (XML) serialization format is used to define RDF. XML's syntax is formally specified [7] and is ideal to model simple triples such as subject-predicate-object (and other) relations. The following RDF relation:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:dc="http://purl.org/dc/elements/1.1/">
 <rdf:Description rdf:about="http://www.awl.com/Formal_Synthesis">
  <dc:title>High-level Synthesis</dc:title>
  <dc:publisher> Addison-Wesley Publishing</dc:publisher>
 </rdf:Description>
</rdf:RDF>
```

is the RDF knowledge model of the fact: "The title of this resource, published by the Addison-Wesley Publishing company, is High-level Synthesis". RDF and XML define object relations that also represent data attributes in large programs, and they can be used to validate internal data representations of E-CAD tools. XML files are easily readable by both humans and machines. An XML schema is a formal definition of an XML file, and it constraints the content and structure of such files. XML schema can be used to validate a particular XML instance. Well-known formalisms of the XML schema are the document type definition (DTD) language, the XML Schema and the Relax NG formats [7].

A valid XML instance can be checked against the rules of an XML schema. This is done by certain XML (commercial or free) parsers compatible with specific XML schema implementations such as the DTD or the Relax NG languages. The following XML instance is automatically produced by the author's HLS tool (see following paragraphs), and it defines a data type and two subprograms in the tool's source ADA programs:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Produced by CCC front-end compiler -->
<schemaxmlns=http://www.w3.org/2001/XMLSchema
 targetNamespace="http://www.w3.org/2001/XMLSchema">
<annotation>
 <documentation>
  XML schema for a hierarchical module of the source code
 </documentation>
</annotation>
<complexType name=" hierarchy_part">
 <sequence>
  <element name=" type_def_natural_2048"/>
   <sequence>
    <element name=" data_object_variable2 "/>
    <element name=" data_object_constant1_value_100 "/>
    <element name=" data_object_constant1_value_1000"/>
     </sequence>
      <element name=" function_convert3"/>
      <sequence>
       <element name=" input_parameter_my_input1 "/>
       <element name=" input_parameter_my_input2 "/>
       <element name=" input_parameter_my_input3 "/>
      </sequence>
       <element name=" procedure_differential2"/>
      <sequence>
       <element name=" input_parameter_my_input4 "/>
       <element name=" input_parameter_my_input5 "/>
       <element name=" output_parameter_my_output1 "/>
      </sequence>
   </sequence>
  </complexType>
</schema>
```

The above XML instance defines the data type "natural_2048" and two subprograms, the function "convert3" with three formal input parameters "my_input1", "my_input2" and "my_input3" as well as procedure (see ADA-95 definition) differential2 with two formal input parameters and one output.

These elements are of type "hierarchy_part", and each one of them includes a sequence of "leaf" sequences with the structure of all child elements with type defined above.

The graphical validation result of the above XML code is shown in Figure 1, and it was automatically produced by a DTD parser that helped to identify and correct some initial semantic issues in the tool development.

As widely documented, XML is a formal way to model internal information in design (E-CAD) tools. XML representation of C code is introduced in [20] to aid the CASE tool development. A hash table and a stream index are used in [21] to filter out invalid elements of an XML document. The IEEE std 1685-2009 IP_XACT, is based on XML schema, and is used as internal representation and exchange format between EDA tools (IEEE IP-XACT std, 2010). Such internal data include design units, interconnections, functional primitives, metadata, and IP blocks [22].

XML schema and the eXtensible Stylesheet Language Transformations (XSLT) language are used in [23] to enable the sharing, conversion, transfer and exchanging of healthcare database data. Many academic/commercial database and web tools utilise XML schemas and instantiations to represent internal information, such as primary data generated automatically and used by database, multi-media and web processing tools.

XML is combined with Web technology in [24] to structure, consult and share corporate data. Similarity algorithms for XML documents are analysed in [25]. The IBM DB2 query matching and compensation techniques are enhanced with XML functionality, to implement and evaluate query rewrite rules in [26]. An incremental approach called "T-Schema" (of XML-to-relational mapping storage), is proposed in [27], to address the strong dynamics of XML. The SMOQE tool generates the first regular XPath engine and provides answering queries technique, over recursively defined XML views [28] discusses the use of semantic web for EIS and databases. A XML metamodel captures NFRs and their relations in [29].

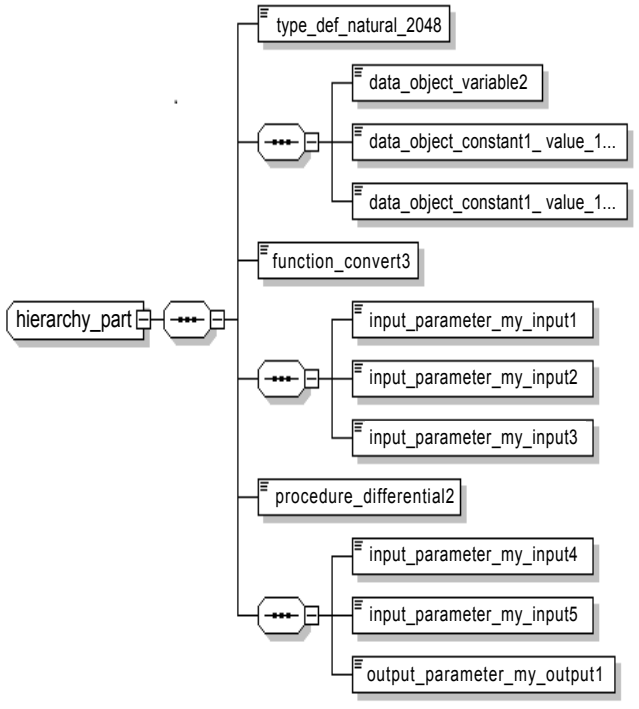

Figure 1: Validation of an Xml Document Instance against a Specific Xml Schema (Automatically Produced From Internal Compiler Data)

The Intermediate Predicate Format

The Intermediate Predicate Format (IPF) was invented by the author of this paper, to model the design and the HLS transformations in the CCC HLS tool [30] analyses the syntax and semantics of IPF, which uses the resolution of Horn clauses as formal object relations [6], to implement the HLS transformations. The front-end phase of the CCC compiler (see following paragraphs), generates the IPF database to capture all the algorithmic, structural and data typing attributes of the source program, as in the following Prolog fact:

fact_id(object1, object2, …, objectN) (form 1)

The Prolog predicate name fact_id relates the objects object1 to objectN in a formal manner. The identifier fact_id names the logical relation between the above objects. IPF facts represent program operations, data object descriptions, data types, operators, subprogram calls, etc. The back-end phase of the CCC compiler applies HLS transformations on these facts so as to produce the optimized design implementation. This is done in combination with the internal logical rules as a design knowledge-base in order to "conclude" and infer the RTL (register-transfer level) implementations. The IPF syntax facilitates declarative processing by Prolog predicates, as well as imperative processing. This is because IPF facts (e.g. data table facts) are referenced with their entry numbers in other IPF facts (e.g. program table facts).

The IPF XML schema view is used to validate the internal state of the front-end and back-end phases of the CCC HLS tools. The following XML instance models the program statement [30] of form 1:

```
<complexType name=" prog_stmt">
 <sequence>
    <element name=" module_subprogram2"/>
    <element name=" entry_number_3"/>
    <element name=" states_0"/>
    <element name=" operator_63"/>
    <element name=" left_operand_dx"/>
    <element name=" right_operand_dy"/>
    <element name=" result_operand_xc"/>
    <element name=" next_operation_5"/>
 </sequence>
</complexType>
```

The validation of this XML instance is shown in Figure 2.

The Intelligent CCC HLS Techniques

The synthesis design flow

The HLS flow includes two major steps: the front-end phase and the back-end phase. These two tools exchange data via the IPF database. XML instances of IPF are automatically generated to validate

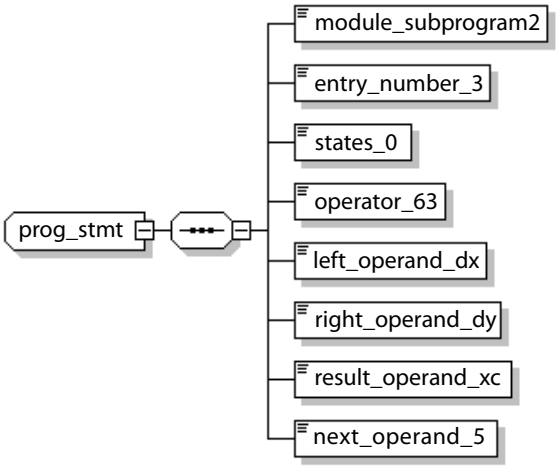

Figure 2: Graphical Validation of the Program Statement Xml Instance

the compilation process, using graphical or command-line based XML validators. The front-end phase performs the typical tasks of a software compiler, such as parsing, generation of abstract syntax trees, type-checking of data and program statements, optimization of auxiliary variables and constants, generation of syntactic and semantic error messages etc.

The XML schema view is also used by the front-end and back-end phases to formally validate the translation, since IPF constitutes a formal link between the two phases of the hardware synthesizer.

CCC analyses the IPF database and generates the initial FSM schedule, considering custom options (such as the location of large data objects on shared memory). Then, it optimises the initial schedules using the PARCS (Parallel Abstract Resoure-Constrained Scheduling) scheduler. PARCS works with, or without resource constraints, and generates the maximally parallel hardware implementation [4], while satisfying source code dependencies.

A custom options file can be used to mark certain program subroutines as "custom blocks", so that they are used as pre-optimized and static custom modules. Hardware arithmetic blocks, or complete data-flow systems such as DSP filter blocks and cryptographic mathematical functions, can be used as custom blocks as in the benchmarks analysed in later sections.

Other custom options of the back-end compiler are the targeting of either massively-parallel (with resource redundancy), or conventional FSM+datapath micro-architectures modelled in Hardware Description Language (HDL) RTL code. More options include the targeted language which (at the moment) include VHDL and Verilog HDL.

Formal Back-end HLS Transformations

The logical relations of the back-end compiler use definite clauses [6] such as follows:

$A0 \leftarrow A1 \wedge \ldots \wedge An$ (where $n \geq 0$) (form 2)

where $\leftarrow$ is the logical implication symbol ($A \leftarrow B$ means that if B applies then A applies), $\wedge$ is the logical conjunction symbol, and A0, …, An are atomic formulas (logic facts) of the form:

predicate_symbol(Var_1, Var_2, …, Var_N) (form 3)

Where the positional parameters Var_1,…,Var_N of the above predicate "predicate_symbol" are either variable names (such as in the back-end inference rules), or constants (such as in the IPF table statements) [6]. By combining these, the source code subroutines are transformed into optimized, provably-correct RTL hardware implementations.

Formal validation using XML schema

Formal logic rules (logic relations) as in form 2 construct the back-end inference engine. Hence, the IPF's facts "drive" the logic rules of the back-end compiler which infers provably-correct hardware implementations, which are technology-independent, free of any standard template and custom microarchitectures in synthesizable HDL code.

The PARCS optimizer works on the enhanced with the custom user options schedule, such as the shared memory access operations. The XML view is validated for the intermediate representations and processes, throughout the various phases of the compilation. Here follows the XML schema validation of the state (…) inference rule:

```
<complexType name="state">
 <sequence>
  <element name="Module_1"/>
   <element name="parcs"/>
   <element name="parcs_state_number"/>
   <element name="parcs_state_name"/>
   <element name="parcs_next_state"/>
   <element name="no_conditional_transition"/>
   <element name="scheduled_operation_list">
    <complexType>
     <sequence>
      <element name="Operation_1"/>
      <element name="Operation_2"/>
      <element name="Operation_3"/>
     </sequence>
    </complexType>
   </element>
   <element name="no_conditional_operations"/>
  </sequence>
 </complexType>
```

This XML instance models one PARCS state of design Module_1, with three scheduled operations in parallel and with no conditional operations or transitions. Both the logic and the XML views of IPF are extracted automatically by the front-end and back-end compilation phases, and they are validated in both logic programming and XML views. The graphical validation of the above XML PARCS state instance is shown in Figure 3.

The XML view of the HLS transformations and data of the back-end compiler consists of relations between predicate symbols. An example logic programming view of the relation transform1 (HLS

transformation) between objects operation1, operand_A, operand_B, result_C is the following:

transform1(operation1, operand_A, operand_B, result_C).

Here follows the XML view of the same relation:

```
<complexType name=" transform1">
 <sequence>
  <element name=" operation1"/>
  <element name=" left_operand_A "/>
  <element name=" right_operand_B "/>
 <element name=" result_data_C "/>
 </sequence>
</complexType>
```

The graphical XML validation of the same transformation is shown in Figure 4. The XML schema instances are validated using any available validators.

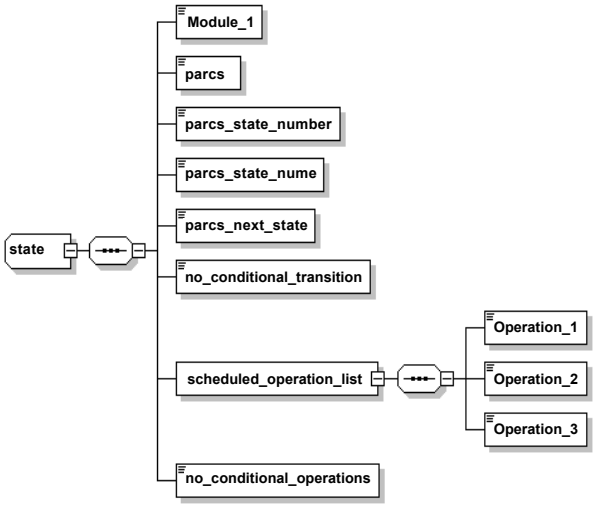

Figure 3: Xml Schema Validation of a PARCS State

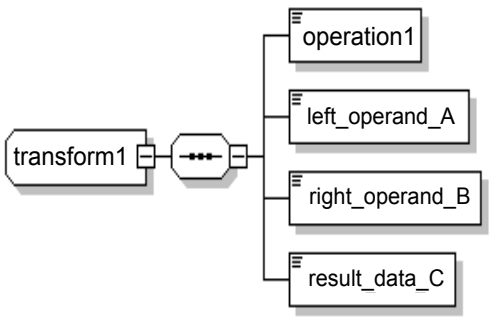

Figure 4: Xml Schema Validation of the Transform1 Back-End Compiler Predicate

Experimental Designs with Custom Options

A large number of ADA designs were synthesized with the CCC compiler and all of them were proven by simulations to be accurate and matching the behaviour of their source models. Here two of them have been analysed in order to demonstrate the usability of the CCC HLS custom options. These are a DSP FIR filter, and a RSA crypto-processor from cryptography applications.

In all tests, the intermediate form and the internal HLS transformations were validated against the respective XML schemas which were automatically extracted from critical points in the compilation flow. For the DSP filter, the two lower-level subroutines in ADA, which model the processing of one more incoming sample and the shifting of the filter history by one position, were chosen to be implemented as custom combinatorial blocks taking one clock cycle to implement. A top-level subroutine which contains calls to the above subroutines, was used to process a whole length of incoming sample arrays.

This subroutine was processed normally via the CCC compiler and produced VHDL code with an optimized FSM of 10 states. Within appropriate state descriptions of this FSM, the above custom block subroutines are called via VHDL call mechanism. The whole ADA coding and compilation of the DSP filter took less than half an hour to run. The hierarchy of the FIR ADA code and implementation is shown in Figure 5.

The experimental CCC environment is shown in Figure 6. The intended system is modelled in the ADA programming language. The initial ADA specification model is compiled with the GNU ADA and the generated binaries are verified with the ADA testbench and test vectors. This exhibits a rapid verification manner, due to extremely high compile-and-execute verification speed. Then the ADA subroutines that are intended for hardware (microelectronic) implementation are integrated in an autonomous ADA package (library module). The latter is compiled and synthesized into hardware using the prototype CCC tools. The generated hardware modules are downloaded in A Xilinx Virtex-2 FPGA was accommodating the synthesized hardware blocks, with the synthesis flow and the target architecture shown in Figure 6.

The initial state schedules are first extended with custom option-guided operations and the result is optimized with the PARCS scheduler as a new schedule. Statistics regarding the optimization rates of the PARCS scheduler are shown in Table 1.

Table 1 indicates that the states reduction rate reaches up to 41% for the FIR processor case. This is a very efficient hardware implementation of large designs with a few hundred states per module, such as large ASICs and complex IP modules being part of embedded system SoCs.

The code of the ADA subroutines can be either standalone or hierarchical. This means that a number of ADA subroutines can include function/procedure calls to other subroutines of the library that is synthesized.

This hierarchy is maintained through the CCC compilation. Thus, the CCC designer dictates the modularity of the generated hardware blocks. Additionally, all the necessary co-processor interfaces and (e.g. memory) communication protocols are automatically synthesized by the CCC compiler and inserted in the initial schedules derived directly from the input models. This is implemented via a memory custom options file. Moreover, selected ADA subroutines can be "marked" as custom arithmetic modules. These are usually complex Boolean

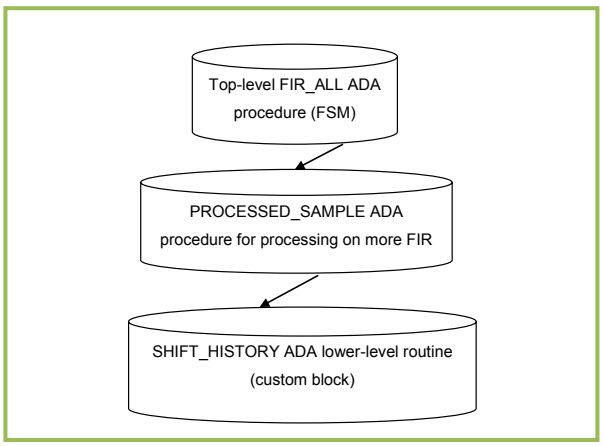

Figure 5: Fir Filter Ada Code Hierarchy

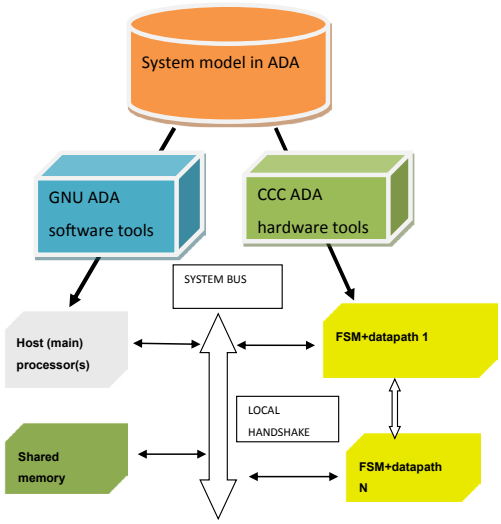

Figure 6: Synthesis Flow and Execution Architecture

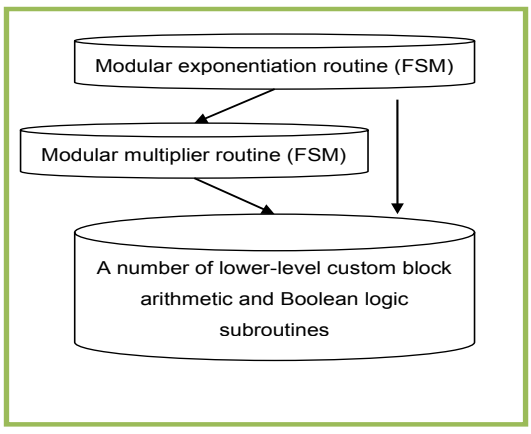

Figure 7: RSA Crypto processor ADA Code Hierarchy

Module Name	Initial Schedule States	PARCS Result States	State Reduction Percentage
DSP FIR filter processor	17	10	41%
RSA Crypto Processor	16	11	31%

Table 1: PARCS Optimization Statistics

functions and they are inserted as expanded VHDL procedure calls in the generated FSM states, executed in a clock cycle. The custom blocks option strategy was followed on the lower-level functions of the FIR DSP filter and RSA crypto-processor benchmarks.

Table 1 demonstrates that the number of hardware states is increasingly high, which constitutes the importance of the CCC framework contribution, in keeping the complexity of contemporary designs under control. It is a shared experience by experienced hardware designers, that development and verification of very complex FSMs with over 20-30 states is cumbersome and extremely prune to errors. Experienced programmers can use the CCC toolset, to implement very complex hardware designs in a few hours, whereas this takes usually more than 6 months of traditional development and verification time. This is due to the use of formal logic programming and RDF validation techniques embedded in the intelligence of the CCC synthesizer.

The ADA code hierarchy for the RSA public-key cryptography application is shown in Figure 7. The modular exponentiator and multiplier automatically produce optimized FSMs with 12 and 29 states respectively. The other 3 lower-level modules are transformed as custom blocks and their VHDL calls are instantiated into the higher-level FSM state descriptions, as shown in Figure 7. The multiplier subroutine is called within the exponentiator subroutine. This is a special case for the CCC translation and is being dealt as such by the compiler. In particular, this subroutine call is used to generate an “interface event” between the modular multiplier module and the exponentiator module. In general, when across both sides of a subroutine call, none of them is intended to be a custom block, then the subroutine calling is converted into a HDL module interface and data exchange mechanism. All of the VHDL modules for the FIR filter and the RSA cryptoprocessor have been simulated (although due to the formal nature of the CCC tool, not necessary) and the results coincided with the results of the ADA verification testbenches.

Conclusions and Future Work

The main contribution of this work is a formal, high-level hardware synthesis toolset and method developed by the author of this paper. The CCC HLS tool utilizes compiler-compiler and logic inference techniques to turn synthesis formal. The synthesis transformations are enhanced with XML schema validation as well as RDF logic relations to implement the execution and formal validation of the prototype hardware compiler. XML views of IPF as well as the formal CCC transformations are validated against their schema views.

Arbitrary and general input ADA code is synthesized into functionally-equivalent RTL VHDL/Verilog hardware implementation. Many applications were synthesized with the CCC toolset, two, very indicative ones being discussed in this paper. In any case, the functionality of the produced hardware accelerators (coprocessors) matched that of the input subprograms. The synthesized hardware can be used to accelerate complete hardware/software systems in their time-critical routines. The PARCS scheduler achieves high state-optimization rates which exceeded 36% in some cases (Table 1).

Future work includes more input programming languages (e.g. ANSI-C, C++) and a more globalized use of RDF techniques throughout the flow of the HLS toolset. Moreover, more diagrammatic system modelling formats are explored such as the UML diagrams to play the role of system hardware accelerator models.

References

1. Gal BL, Casseau E, Huet S (2008) Dynamic Memory Access Management for High-Performance DSP Applications Using High-Level Synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 16: 1454-1464.
2. Gupta S, Rajesh KG, Nikil DD, Nikolau A (2004) Coordinated Parallelizing Compiler Optimizations and High-Level Synthesis. ACM Transactions on Design Automation of Electronic Systems 9: 441-470.
3. Walker RA, Chaudhuri S (1995) Introduction to the scheduling problem. IEEE Design & Test of Computers 12: 60-69
4. Dossis MF (2011) A Formal Design Framework to Generate Coprocessors with Implementation Options. International Journal of Research and Reviews in Computer Science (IJRRCS) 2: 929-936.
5. Paulin PG, Knight JP (1989) Force-directed scheduling for the behavioral synthesis of ASICs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 8: 661-679.
6. Nilsson U, Maluszynski J (1995) Logic Programming and Prolog. John Wiley & Sons Ltd.
7. Allemang D, Hendler J (2011) Semantic Web for the Working Ontologist.
8. Holub A (1990) Compiler Design in C. Prentice-Hall Inc., New Jersey, USA
9. The Electronic Design Interchange Format
10. http://www.rulabinsky.com/cavd/text/chapd.html
11. Kountouris AA, Wolinski C (2002) Efficient Scheduling of Conditional Behaviors for High-Level Synthesis. ACM Transactions on Design Automation of Electronic Systems 7: 380-412.
12. Wang W, Tan TK, Luo J, Fei Y, Shang L, et al. (2003) A comprehensive high-level synthesis system for control-flow intensive behaviors. Proceedings of the 13th ACM Great Lakes symposium on VLSI (GLSVLSI '03).
13. Huang C, Ravi S, Raghunathan A, Jha NK (2007) Generation of Heterogeneous Distributed Architectures for Memory-Intensive Applications Through High-Level Synthesis. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 15: 1191-1204.
14. Huang W, Raghunathan A, Jha NK, Dey S (2003) High-level Synthesis of Multi-process Behavioral Descriptions". in Proceedings of the 16th IEEE International Conference on VLSI Design (VLSI'03).
15. Gupta S, Gupta RK, Nikil DD, Nikolau A (2003) Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits. In Proceedings of the IEEE Conference on Computers and Digital Technologies 150: 330-337.
16. Molina MC, Ruiz-Sautua R, Garcia-Repetto P, Hermida R (2009) Frequent-Pattern-Guided Multilevel Decomposition of Behavioral Specifications. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28: 60-73.
17. Keinert J, Streubuhr M, Schlichter T, Falk J, Gladigau J, et. al. (2009) SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. ACM Transactions on Design Automation of Electronic Systems (TODAES).
18. Kundu S, Lerner S, Gupta RK (2010) Translation Validation of High-Level Synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 29: 566-579.
19. Paik S, Insup S, Kim T, Youngsoo S (2010) HLS-I: A High-Level Synthesis framework for latch-based architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 29: 657-670.
20. Atsumi N, Kobayashi T, Yamamoto S, Agusa K (2011) An XML C Source Code Interchange Format for CASE Tools. In Proceedings of the IEEE 35th Annual Computer Software and Applications Conference (COMPSAC): 498-503.
21. Weijian X, Heji Z, Jiasheng Z (2011) The XML filtration based on hash table and stream index. In Proceedings of the International Conference on Mechatronic Science, Electric Engineering and Computer (MEC):1286-1290.
22. Jumaa H, Rubel P, Fayn J (2010) An XML-based framework for automating data exchange in healthcare. In Proceedings of the 12th IEEE International Conference on e-Health Networking Applications and Services (Healthcom).
23. Sanchez-Martinez LD, Medina-Ramirez RC (2010) An XML information management: a research team case. In Proceedings of the 20th International Conference on Electronics, Communications and Computer (CONIELECOMP):197-200.
24. Sun X, Cheng H, Wang X (2010) An XML schema-based similarity algorithm. In Proceedings of the 2nd International Conference on Future Computer and Communication (ICFCC) 1: 36-38.
25. Godfrey P, Gryz J, Hoppe A, Ma W, Zuzarte C (2009) Query Rewrites with Views for XML in DB2. In Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE '09).
26. Xu Y, Ma S, Yi S, Yan Y (2010) From XML Schema to Relations: A Incremental Approach to XML Storage. In Proceedings of the 2010 International Conference on Computational Intelligence and Software Engineering (CiSE):1-4.
27. Rajugan R, Chang E, Feng L, Dillon TS (2006) Modeling Dynamic Properties in the Layered View Model for XML Using XSemantic Nets. Advanced Web and Network Technologies, in Lecture Notes in Computer Science.
28. Kassab M, Ormandjieva O, Daneva M (2008) A Traceability Metamodel for Change Management of Non-Functional Requirements. In Proceedings of the Sixth International Conference on Software Engineering Research, Management and Applications IEEE: 245-254.
29. Dossis M (2010) Intermediate Predicate Format for design automation tools. Journal of Next Generation Information Technology (JNIT) 1: 100-117.
30. Wenfei F, Geerts F, Xibei J, Kementsietsidis A (2007) Rewriting Regular XPath Queries on XML Views. In Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE 2007): 666-675.