Automatic generation of Coprocessor program from VHDL description

Emna Kallel, Yassine Aoudni, Mohamed Abid
Computer & Embedded Systems Laboratory, Univ. Sfax, ENIS School
BP 1173, Sfax 3038, Tunisia

***Abstract*—The design and implementation of complex embedded systems including custom hardware and software is still to a large degree based on a collaboration of semi-manual and often poorly interconnected design methods and tools. This usually results in repetitive and longer development cycles. This paper describes an intelligent method to automate and accelerate the hardware generation process. Indeed, a VHDL parser is developed to provide an automated path from VHDL entry to Coprocessor design. To prove the correctness of our method, a Java source code framework named Automatic Custom Architecture generator (ACAgen) is developed. Experimental results on 3D sample application show that the proposed framework can rapidly and easily generate coprocessor. It leads to the design of large and complex systems-on-chip with less costs and higher performances.**

Keywords-Automatic generation; Coprocessor;VHDL parser; ACAgen.

I. Introduction

Technological advances in Field Program Gate Array (FPGAs) have opened the possibility for research in creating optimized custom hardware to run many algorithms at much greater speeds than possible on a standard computer processor [1]. Indeed, hardware accelerators or coprocessors are often used to provide efficient implementations of application-specific functions and to provide enhanced performance in systems-on-chip (SoC).

Nevertheless, currently, the development of applications for such platforms needs both in-depth software and hardware design knowledge due to its increasing fabrication cost and design complexity. Therefore, dedicated tools and frameworks that make the hardware details transparent to the software designer are necessary. One component of such framework is an automated hardware design generator that takes its input from a high level language (HLL) such as C. In this generation process there are multiple challenges to be overcome. These challenges range from mapping the high-level (HL) constructs to the hardware to finding the proper set of optimizations and transformations that would lead to optimal hardware design. Significant researches have been done in automating the creation of this hardware such that users do not need a high proficiency in hardware design to be able to benefit from. For example, projects, like Handel-C [3], remained oriented towards the hardware designers. There, the C syntax is extended with constructs, exposing all hardware details to the designer. Other project, like SA-C [5], tried to hide the hardware details for the software designers. They used C variations, excluding problematic construct and introducing language extensions that facilitated the optimizations and the hardware mapping. Those extensions however meant that existing applications had to be rewritten. Therefore, later projects like DEFACTO [6], SPARK [7] and ROCCC [8] considered unmodified C as input. Those projects emphasize parallelizing transformations and some also address memory access optimizations. Moreover, several commercial tools that generate hardware from HLL input also appeared like Catapult-C [9], Impulse-C[10]. Xtensa [4] allows an efficient ASIP rapid prototyping but it don't targets others processor models like LEON, MIPS, Microblaze and NIOS-II. Despite all those efforts, the automated hardware generation is yet to become a widely adopted industrial practice [2]. One of the reasons is the lack of automation of the research projects generation process which implicitly requires hardware-design knowledge.

To address this challenge, we are currently working on a toolset called "Automatic Custom Architecture Generator" (ACAgen) that automatically generates a hardware system (in VHDL) and the software (in C) that executes on that hardware without need of specialized hardware development knowledge. The work discussed here for the ACAgen toolset develops a module for the automatic implementation of the coprocessor interface from a HLL (VHDL in our case). A VHDL parser is developed analyzing the VHDL description input to find the proper set of optimizations and transformations that would lead to optimal coprocessor design. The presented design flow is a parser-based method that hides unnecessary details from high-level design phases and provides an automated path from HLL entry to hardware design. So, it can be easily used by non-HW experts of on-chip systems implementation.

The remaining of this paper is organized as follows. Section 1 describes the overall ACAgen custom instruction integration flow. Section 2 presents the coprocessor generation process. Section 3 presents the FPGA based synthesis results of a 3D synthesis application and discusses the experimental results showing the performance of the ACAgen framework. Finally, we end up with a conclusion.

II. ACAGEN CUSTOM INSTRUCTION INTEGRATION FLOW

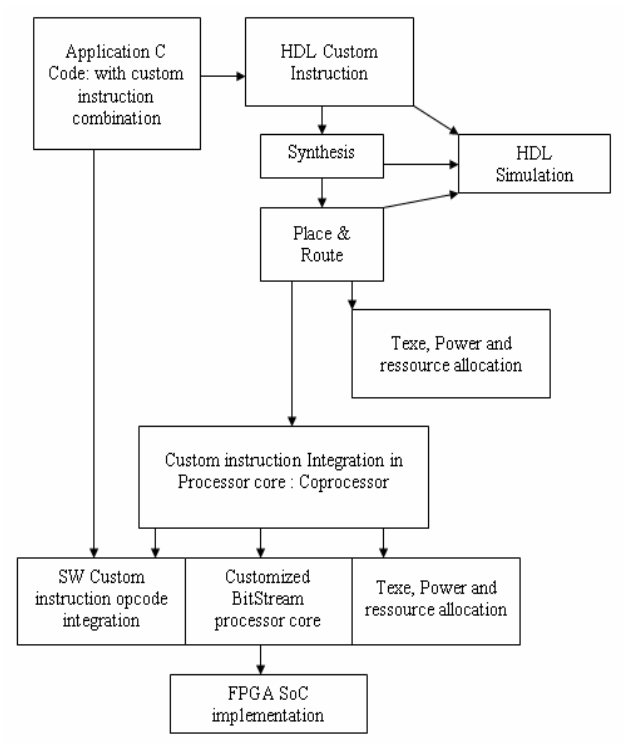
Application C Code: with custom instruction combination
HDL Custom Instruction
Synthesis
HDL Simulation
Place & Route
Texe, Power and ressource allocation
Custom instruction Integration in Processor core : Coprocessor
SW Custom instruction opcode integration
Customized BitStream processor core
Texe, Power and ressource allocation
FPGA SoC implementation

Figure 1. ACAgen custom instruction integration flow

Using the ACAgen toolset, we aim to automate the process of Custom Instruction Integration within Reconfigurable SoC and FPGA Devices. This can be successfully accomplished via three important steps.

Firstly, as described in previous work [13], a VHDL description of custom instruction is automatically generated from a high abstraction level. The generated VHDL code will be then verified using Quartus environment. This step will help the designer to rapidly test and verify the functionality of the custom instruction before integration in the NIOSII core and to get performance information about the hardware module (resource allocation, execution time and power dissipation).

The second step consists in updating the initial application C code with the custom instruction opcode. This step is successfully performed by developing a specific C parser [14].

In third step, the rapid integration of custom instruction in NIOSII core consists in automatically generating a specific coprocessor interface to adapt the communication between the custom hardware module and the ALU of NIOSII core. The coprocessor interface must respect:

- data size and nomination for inputs and outputs
- control signals for sequential and combinatory hardware module

In our project, we propose to use the NIOSII prototyping platform [18]. Indeed, NIOSII is a soft core that offers the possibility to integrate 5 custom instructions using three main registers: (dataa[0..31], datab[0..31]) as inputs and (result[0..31]) as output.

After these three steps, SOPC Builder and Quartus environment can be used to integrate custom instruction opcode in NIOSII compiler and to generate the customized NIOSII bitstream. In addition, performance information can be rapidly collected to specify the custom prototype by resources usage, execution time and power dissipation.

This work focuses on the third step of ACAgen design flow describing an intelligent method to automatically generate a coprocessor interface identified by the SOPC Builder from a simple VHDL description. The proposed method is based on the implementation of specific VHDL parser able to analyze the VHDL code to be converted to a coprocessor form. Below, we will detail the design of the developed parser.

III. COPROCESSOR GENERATION PROCESS

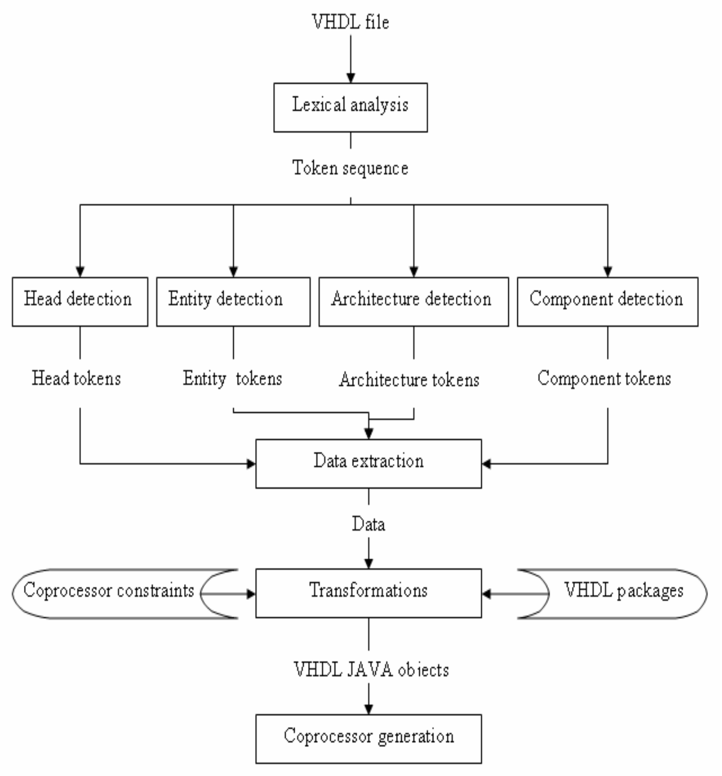

Figure 2. from VHDL file to Coprocessor interface

This module of ACAgen toolset takes as input a simple VHDL file and creates the corresponding coprocessor program (figure 2). The coprocessor integration within the NIOS processor demands some constraints. So, a VHDL parser able to know and extract the different parts of a VHDL file is needed.

In our design flow, a token-based VHDL analysis is adopted. Indeed, each line of VHDL input source files is divided into tokens corresponding to a lexical rule of the programming language. The tokens of all source files are concatenated into a single token sequence. At this step, the white spaces (including \n and \t and comments) between

Algorithm 1. Entity detection

Input: token sequence

Output: compName {component Name}

Input/output: an initial tokens vector $\vec{v} \leftarrow \phi$

```
1: for each token do
2:    if token = "ENTITY"
3:       compName ← nextToken
4:       if nextToken = "IS"
5:          do
6:          $t$ ← nextToken {a temporary states}
7:          $\vec{v} \leftarrow \vec{v} \bigcup t$
8:          while (nextToken = "END")
9:          end do while
10:      end if
11:   end if
12: end for
```

TABLE I. COPROCESSOR CONSTRAINTS

Name port	size	type
Dataa	32	input
Datab	32	input
Result	32	output
Start	1	input
Reset	1	input
Done	1	output
Clk	1	input

tokens are removed from the token sequence. To automatically detect the different components of VHDL file, several detection algorithms analyzing tokens are developed. All detection processes are handled in the same way as described in Algorithm 1. The ENTITY detection Algorithm traverses all tokens to identify the token sequence of the VHDL entity which is then saved in a vector $\vec{v}$. In that case, data is automatically extracted from each generated vector. For example, the names, sizes and types of the different Entity inputs/outputs are identified from the extracted Entity token sequence.After data extraction, the different identified token sequences are transformed, i.e., tokens are added, removed, or changed based on the coprocessor constraints as shown in table 1. These coprocessor rules are mapped to JAVA objects (from VJP and LPM Package described in [13]) which are then instantiated using the new coprocessor data (figure 3). Finally, the VHDL code for coprocessor following the specifications for the NIOS II processor is generated. The coprocessor 32-bits input ports are named dataa and datab and the 32-bits output port is named result. The generated coprocessor uses the clk, clk_en, reset, and start inputs along with the done output.

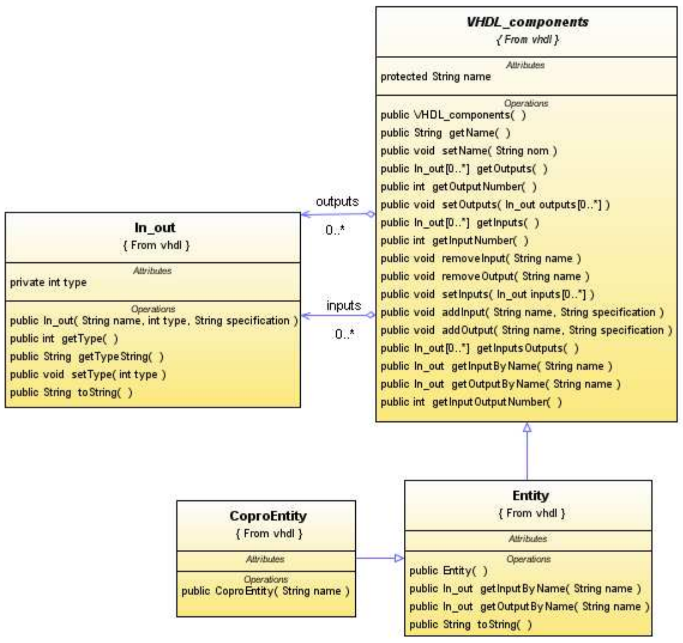
VHDL_components
{ From vhdl }
Attributes
protected String name
Operations
public VHDL_components( )
public String getName( )
public void setName(String nom)
public In_out[0..*] getOutputs( )
public int getOutputNumber( )
public void setOutputs(In_out outputs[0..*])
public In_out[0..*] getInputs( )
public int getInputNumber( )
public void removeInput(String name)
public void removeOutput(String name)
public void setInputs(In_out inputs[0..*])
public void addInput(String name, String specification)
public void addOutput(String name, String specification)
public In_out[0..*] getInputsOutputs( )
public In_out getInputByName(String name)
public In_out getOutputByName(String name)
public int getInputOutputNumber( )
outputs
0..*
inputs
0..*
In_out
{ From vhdl }
Attributes
private int type
Operations
public In_out(String name, int type, String specification)
public int getType( )
public String getTypeString( )
public void setType(int type)
public String toString( )
CoproEntity
{ From vhdl }
Attributes
Operations
public CoproEntity(String name)
Entity
{ From vhdl }
Attributes
Operations
public Entity( )
public In_out getInputByName(String name)
public In_out getOutputByName(String name)
public String toString( )

Figure 3. Part of VJP Package

IV. RESULTS

To evaluate the described ACAgen module, we first check its efficiency in terms of complexity and used memory. Experimentations show that ACAgen coprocessor generation module illustrates two original features. First, it has a small size – 300 lines (low complexity). Second, it quickly achieves good code coverage: in 1 ms it automatically updates and generates the new VHDL code for coprocessor .

TABLE II. PERFORMANCE COMPARISION

	ACAgen coprocessor generator Module	**Framework to Generate coprocessor [12]**
Lines of source code	300	552
Used memory	21 KB	33 KB
Runtime	3s	10s
Time generation	1 ms	10 ms
Total number of Generated VHDL lines	1000	1264

Executing its low-complexity algorithms, ACAgen coprocessor generator module is capable of rapidly generating 1000 lines of VHDL code with only 300 lines of source code. If we compare the numbers presented in table II with other Coprocessor generator tool, the results are still very good. Indeed, our framework improves its flexibility on Netbeans 5.5 version and centrino duo Core 2GHz with 21 Kbytes and 3 seconds average runtime while the other

TABLE III. SYNTHESIS RESULTS

	Logic utilization (additional cost %)	Power dissipation per module (mW) (additional cost en %)	Time execution (ticks number)
Scalar product	113 (9.6%)	2.34 (4.7%)	24537
Vectorial product	103 (8.8%)	1.37 (2.8%)	38236
Mult_Mat	193 (8.8%)	3.66 (7.4%)	23543
Projection	370 (31.6%)	3.64 (7.4%)	35900

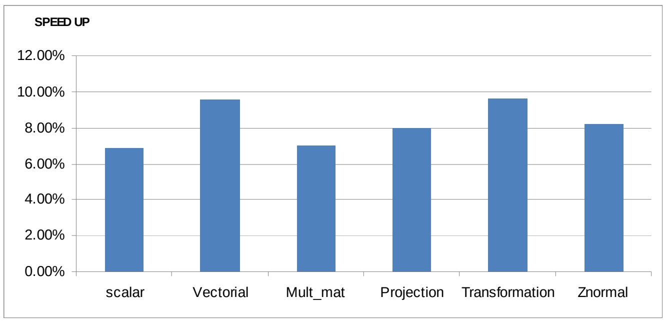

Figure 4. Speedup offred by our solution

framework [12] works with 33 Kbytes and 10 seconds average runtime.

Moreover, using the ACAgen module, it was easy to implement different applications within NIOSII processor core and the FPGA Stratix 2S180 device which includes 143520 ALUTs for hardware logic [11]. The performance results are given by table III. We note that the mult_mat application attain the maximum gain with 23543 ticks and additional costs of 16.5% in Logic utilization. On the other hand, it results in power consumption increase with 7.4%.

Also, as shown in figure 4, the results indicate that our approach can quickly (in several seconds to a minute) generate coprocessor for realistic programs. The coprocessor design can achieve an average speedup of 6.78X and peak speedup of 9,6X over the manual solution.

This analysis demonstrates the efficiency of our ACAgen module to generate hardware from a high level language (HLL). Through the VHDL parsing method presented in this work, a designer can automatically generate its implementation at RT level. The designer can easily and rapidly generates different SoC configurations to look for the best alternative for a given application.

V. CONCLUSION

In this paper, we proposed an approach to coprocessor generation from a VHDL description. Our automatic code generation approach is based on VHDL parsing by developing token-based algorithms able to know and extract the different parts of the entered VHDL file. Experimental results show that the proposed ACAgen module can rapidly and easily generate coprocessor for realistic programs. It leads to the design of large and complex systems-on-chip with less costs and higher performances.

REFERENCES

[1] Philip I. Necsulescu, Voicu Groza, “Automatic Generation of VHDL Hardware Code from Data Flow Graphs”, IEEE International S ymposium on Applied Computational Intelligence and Informatics • May 2011 • Timi úoara, Romania.

[2] Y. D. Yankova, K. Bertels, S. Vassiliadis, R. J. Meeuws, and A. Virginia, “Automated hdl generation: Comparative evaluation,” in Proceedings of International Symposium on Circuits and Systems (ISCAS2007), May 2007.

[3] Handel-c language reference, http://www.celoxica.com/, Celoxica web site 2012.

[4] Quickly Create Customized Functional Blocks, http://www.tensilica.com/products/xtensa-customizable.htm , Tensilica web site 2012.

[5] W. A. Najjar, A. P. W. B¨ ohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross, “High-level language abstraction for reconfigurable computing.” IEEE Computer, vol. 36, no. 8, pp. 63–69, 2003.

[6] P. C. Diniz, M. W. Hall, J. Park, B. So, and H. E. Ziegler, “Bridging the gap between compilation and synthesis in the defacto system.” in Proceedings of 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC’01), 2001, pp. 52–70.

[7] Spark: A parallelizing approach to the high-level synthesis of digital circuits, http://mesl.ucsd.edu/spark/, SPARK project web site 2012.

[8] ZHI GUO, WALID NAJJAR, BETUL BUYUKKURT, “Efficient hardware code generation for FPGAs”, ACM Transactions on Architecture and Code Optimization, Vol 5 Issue 1, May 2008.

[9] Catapult C Synthesis Overview , http://www.mentor.com/esl/catapult/overview, 2012.

[10] Impulse accelerated technologies, http://www.impulseaccelerated.com/, 2012

[11] Altera Corporation, Stratix II Device Handbook, 2004.

[12] Michael F. Dossis, “A Formal Design Framework to Generate Coprocessors with Implementation Options”, International Journal of Research and Reviews in Computer Science (IJRRCS) Vol. 2, No. 4, August 2011, ISSN: 2079-2557.

[13] E. Kallel, Y. Aoudni, M. Abid “Object-oriented approach to Custom Instruction design”, accepted for publication in FTFC 6-7 paris 2012.

[14] E. Kallel, Y. Aoudni and M. Abid, “Parser-based automatic code generation approach for embedded systems” in ICCA2011, Ryadh, Arabie Saudite, 2011, in press.