Periodica Polytechnica
Electrical Engineering
and Computer Science

60(2), pp. 107-124, 2016
DOI: 10.3311/PPee.8527
Creative Commons Attribution

Semi-Automatic RTL Methods for System-on-Chip IP Delivery in the Cyber-Physical System Era

Péter Horváth[1*], Gábor Hosszú[1], Ferenc Kovács[2]

RESEARCH ARTICLE

Received 26 August 2015; accepted after revision 20 December 2015

Abstract

With the dawn of Cyber-Physical Systems (CPS) the relevance of System-on-Chips equipped with run-time configurable, application-specific macrocells increases as numerous tasks has to be taken over from the microprocessors in order to cope with the real-time requirements typical of CPS applications. One of the largest challenges of macrocell design is to conform to the demand for increasing computation capacities while keeping the development effort low to handle time-to-market requirements and reduce design cost. This article presents a novel method for creating RTL models of SoCs' reusable macrocells. The proposed method is based on a novel modeling language (AMDL) offering a reduced design time while making the micro-architectural details fully accessible for the designer to ensure the required level of optimization. Beside the formal definitions of the language's semantic elements, the results of design efficiency investigations, moreover, a comparison of AMDL and the similar solutions are also presented.

Keywords

ARTL, AMDL, CPS, SoC, microarchitecture, RTL design

[1] Department of Electron Devices, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, H-1117 Budapest, P.O.B. 91., Hungary

[2] Faculty of Information Technology and Bionics
Pázmány Péter Catholic University
H-1083 Budapest, P.O.B. 278., Hungary

[*] Corresponding author, e-mail: horvathp@eet.bme.hu

1 Introduction

In the recent decade the most spectacular change in the field of digital system design has been the rise of the design entry's abstraction level and the usage of automated methods in the early stages of the design process. This is caused by the ever-increasing time-to-market and time-on-market pressure which has become one of the most important design objectives nowadays. With the dawn of the Cyber-Physical Systems (CPS) integrating physical processes and computation systems directly influencing each other via feedback loops, a huge amount of possible applications emerged and, since the development platforms and tools are very efficient and widely known, the development time has become the key factor on the common market [1, 2, 3].

Because of these characteristics of the backend application requirements, the central data processing hardware components of these systems [4, 5] mainly comprise pre-designed, optimized, and pre-verified macrocells and the focus of digital design shifted to the high level methods used for integrating these components into functionally complete, intelligent systems [6-9].

The foresaid demand for a fast development process in the application area implicitly places major demands upon the computation capacities of the underlying SoCs as well: The intelligence of the CPS applications is usually provided by the software environment running on an embedded microprocessor. These software applications tend to be more and more complex, therefore, the significance of the operating system-based runtime environments is also increasing, even in small designs. Similarly to the hardware development, the tool chains of the application software products are also based on the increasing abstraction (interpreted scripting languages, virtual machines etc.) requiring microprocessors with impressive computation capacities. Moreover, CPS applications are typically real-time systems necessitating predictable computation delays. This predictability is mainly handled by outsourcing the timing-critical tasks to application-specific, highly configurable Intellectual Property (IP) cores instead of performing them in software running on an Instruction-Set Processor (ISP) [10-13]. Since the overall characteristics of SoCs, such

as power-consumption, computation performance and cost significantly depend on the properties of the IPs constituting them; it is expectable that, beside the new approaches addressing system level integration and high level verification, the demand for fast design methods aiming high-quality and optimized hardware models of macrocells will arise as well. In this article an overview of the existing tools and design languages aiming automated Register-Transfer Level (RTL) model generation of application-specific IPs is provided. A novel design method for IP-core design is also proposed, which is developed to improve the efficiency of design process and decrease design time of SoCs' macrocells providing the required computation capacities in CPS applications.

This paper is organized as follows: Section 2 presents the existing methods used for automated RTL generation of SoCs' macrocells and it discusses their advantages and disadvantages. In Section 3 the concept of a novel abstraction level called Algorithmic RTL (ARTL) is introduced, which is intended to unite the above mentioned design methods. Section 4 presents a novel modeling language (AMDL) and synthesis process representing the ARTL abstraction and in Section 5 an alternative macrocell design flow is recommended involving the novel modeling language. Section 6 discusses the experiences gained during AMDL design efficiency investigations and in Section 7 a brief comparison of AMDL and the existing RTL modeling methods is provided. Section 8 draws conclusions.

2 Related work

2.1 The traditional SoC design flow

The traditional SoC design flow consists of numerous steps requiring different modeling means, formal languages, and Computer-Aided Design (CAD) tools. In this section a brief overview of this process is provided in order to make it possible to better understand the integration of our proposed method into the traditional SoC design approach.

Complex SoC design projects usually start with a textual specification directly created according to the user requirements. Based on this informal specification important decisions have to be made with regard to the hardware-software partitioning and the interfaces between these two elementary parts of the design. This partitioning step has a huge influence on the characteristics of the entire product; therefore, it is usually made by experienced system designers. Although, there are modern techniques which may be applied at this point (see Section 2.2), in the traditional design flow partitioning is mostly an intuitive task resulting in a series of simpler, more specific module specifications, which are still expressed in an informal manner. The submodules may be software or hardware components, but, since our paper is exclusively hardware-related, the subsequent steps of the SoC design flow are discussed from the viewpoint of a hardware designer.

Based on the informal specification a more reliable and unambiguous model has to be prepared using a formal language. The main objective of this model is to capture the behavior of the module and it is not concerned in the implementation details. It is basically used as a golden reference during the subsequent design steps. As the formal specification is complete, RTL modeling may start resulting in a synthesizable model of the module. In the traditional design flow RTL modeling is the last step, that is performed completely manually. The subsequent steps, such as RTL synthesis constructing gate-level representation of the circuit based on the RTL model, and physical design concerned in technology-dependent optimizations are mainly done by software tools implementing well-known algorithms and internal representations. These back-end steps are not part of our discussion. Figure 1 shows the relevant steps of the SoC design flow and the most widely used languages used at the different design steps.

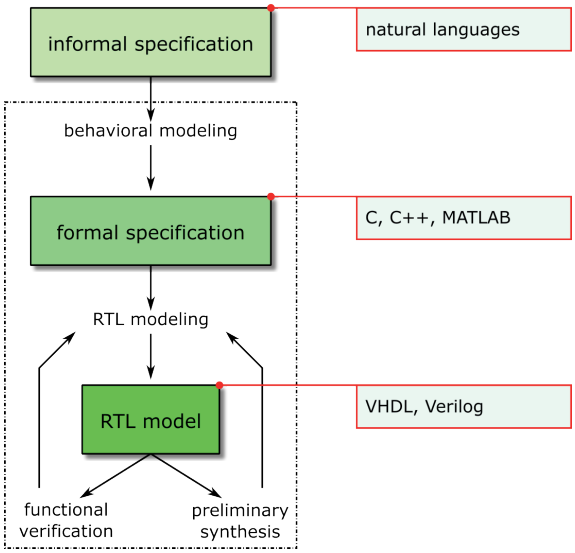

Fig. 1 The front-end steps of the traditional SoC design flow.

2.2 Model-based approaches

The most compelling drawback of the above described design process is that the fully automated, correct per construction steps are preceded by intuitive, hand-optimized tasks, which are prone to errors and inconsistencies. Therefore, in the last decade, significant effort has been made to improve the front-end of this design flow. The so-called model-based approach is a general term used to unite these modern solutions intended to handle the aforementioned difficulties. There are two main directions of this field of research. One of them makes the specification more formal and independent from all implementation details. In this case the ambiguities of natural-language specifications are absent, making time-consuming and costly redesign iterations caused by the misunderstanding of the specification unnecessary. The tool for making this complete independence possible is Unified Modeling Language

(UML), which is a set of graphical representations originally developed for describing the structure[1] and communication mechanisms in complex software systems. However, because of its general nature, it is able not only to model the structure of SoCs but its behavior-oriented diagram types make it possible to automatically generate executable specifications [14, 15], which may be used during the implementation and the verification phase of the SoC as well.

The other type of model-based approaches does not want to make the specification independent of the implementation but it uses front-end languages capable of describing the functionality and the structure of software and hardware parts of the design in the same time on a wide scale of abstraction. The most widely known solution is the hardware-related extension of C++ called SystemC [16]. SystemC is a class library making it possible to describe the hardware and software parts of the design in the exact same language environment (traditional C++ with additional pre-defined macros) and the SystemC simulation kernel is able to co-simulate them. Beside the improved flexibility ensured by the interchangeability of the hardware and software versions of the same module, SystemC hardware models are also capable of underlying an automated RTL synthesis process.

Both directions of the model-based approach tend to improve the design flow by raising the abstraction level of the design entry and by extending the scope of automated processes to these high level representations. The most spectacular advantage of these methods is that they significantly reduce design time but the increase in efficiency comes at the cost of more limited capabilities of modeling architectural details on a lower level of abstraction. Without the opportunity of low-level access to the microarchitecture, hardware designers may experience difficulties when the demand for detailed optimization is strong enough to throw the significance of reduced development time into the shade.

2.3 Methods for optimized architectural design

When the optimization level regarding the timing, power-consumption and/or resource requirement plays a primary role, hand-crafted RTL is the traditional means which IP designers can apply. Transaction level models and languages such as C++, SystemC, or SystemVerilog are used during the design space exploration but the final RTL implementations intended to be the starting point of automated RTL synthesis are usually created manually. However, there are existing solutions for generating RTL based on specific design languages and unique model generation procedures.

Bluespec SystemVerilog (BSV) is a multipurpose modeling language which can be used for describing executable specifications, transaction level behavior, virtual platforms for the software components of SoCs and even RTL models [17]. Moreover, a synthesis method capable of generating gate-level representations from the RTL descriptions written in BSV is also available. The most specific characteristics of the BSV language are that *(i)* the interfaces, beside the modules, are independent design units themselves, so they can be reused without any modifications performed on the modules using them and that *(ii)* the concurrencies of the subcircuits constituting the macrocells are described as atomic transactions. With the usage of the rule-semantics realizing atomic transactions the designer does not have to deal with the interferences between the submodules, the whole system can be handled as a set of small, independently designed pieces. Both of these solutions are intended to make the RTL modeling more efficient. The main drawback of BSV is that the output model is difficult to read and modify, however it is often necessary, since the synthesis procedure and the language elements used in the RTL model are inflexible and cannot take the implementation technology into consideration. Therefore, the generated RTL model is usually modified by hand before RTL synthesis to cope with the requirements of the coding style guides of the design team and the implications of the underlying technology (e.g. block RAM modeling in case of FPGAs) [18, 19].

Architecture Description Languages (ADLs), also known as Processor Description Languages (PDLs) are very specific design languages developed for describing instruction set processors. They are similar to BSV with regard to the level of abstraction. These languages have been developed to describe Application-Specific Instruction-Set Processors (ASIPs) in a more efficient way than traditional hardware description languages as e.g. Verilog and VHDL. ADLs and the associated synthesis tools always provide target architecture models determining some basic characteristics of the described system. The more rigid and detailed the target architecture model is, the more efficient the synthesis can be. Although the skeleton of the model is mainly pre-defined, the designer has some freedom to fit it to the needs of the specification (e.g. unique instructions and accelerators, additional pipeline stages etc.). However, there are two main drawbacks of ADL-based methods. *(i)* The limitations of the above mentioned target architecture models may inhibit the designer to accurately optimize the design [18-21] and *(ii)* they have a very limited scope, since only stored-program microprocessors can be described with them [22, 23].

HLS (High Level Synthesis) is a method used to directly generate synthesizable RTL models from algorithmic specifications, predominantly written in C-like languages [24, 25]. This approach has proved to be very efficient in terms of design time, since it automatizes a time-consuming and error-prone part of digital design; moreover, it is also favorable in terms

1 The structural modeling capabilities of UML are exploited in our work as well, although in a less elaborated form than in the aforementioned solutions. As it may be seen in Section 4, the target architecture models of our method are defined with UML class diagrams.

of reusability, since an algorithmic specification is, contrary to high quality RTL models, completely independent from the backend technology. Nevertheless, there are some major limitations which have inhibited HLS from completely replacing hand-optimized RTL in the IP design flow. The most important drawbacks are that *(i)* in case of designs demanding a high optimization level the designer interfaces of the HLS tools often seem to be unsatisfactory. Once the input model has been adequately prepared for a specific synthesis tool, the designer has only a limited set of constraints to "steer" the process toward the intended microarchitecture [24, 26, 27]. The other limitation which has to be taken into account is that *(ii)* HLS methods have been essentially developed for loop-and-array algorithms. That means that their capabilities can only be taken advantage of by DSP (Digital Signal Processing) applications comprising simple pipeline stages and inter-stage FIFO channels [28].

It is common to the above mentioned approaches that they lend some aspects of higher abstraction levels; the rule semantic of BSV makes hardware design very similar to object oriented programming, while HLS and ADLs use ANSI-C-like language constructs for describing the behavior.

2.4 Earlier work

Earlier stages of this work are reported in [29] and [30]. The concept of ARTL abstraction has been reconsidered ever since and an exact definition has been created which is presented in Section 3. In our earlier approach ARTL represented the abstraction level exclusively concerned about AMDL language while the definition presented in this paper describes ARTL as a common set of key properties of certain existing modeling languages including AMDL. The AMDL language itself has also been significantly improved and the underlying target architecture models have been comprehensively elaborated resulting in well-defined structural and behavioral models presented in Section 4.1 in this paper. Both of the earlier papers concentrated on the synthesis-related issues of AMDL including detailed discussions of the AMDL software tools chain, the synthesis algorithms, test systems' architectures and RTL synthesis results. In this paper we emphasize the formal definitions of the language elements (see Section 4.2) and, instead of paraphrasing the details of the synthesis, we show how AMDL is intended to be integrated into the traditional digital system design flow (see Section 5). However, for the sake of completeness, a brief overview of quantitative design efficiency investigations is also presented in Section 6.

3 The concept of Algorithmic RTL abstraction

The digital IP design flow recommends modeling languages for the abstraction levels. The abstraction levels themselves are traditionally illustrated with the Gajski-Kuhn Y-diagram (GK-diagram) which investigates them from three different points of view. The functional point of view shows how the specific abstraction level models the behavior, the structural point of view describes the typical components of a model on that level and the physical point of view deals with the physical appearance of the particular abstraction level's design units. Although the abstraction levels of the GK-diagram have their own widely used languages and tools, there are some modern modeling means which cannot be unambiguously fit into the diagram. E.g. the aforementioned BSV and the ADLs describe the behavior in a slightly more abstract form than traditional RTL while the structural elements of their formal models are equivalent to those used by traditional RTL models. In order to adequately integrate these special types of modeling means into the GK-diagram a novel level of abstraction called Algorithmic Register-Transfer Level (ARTL) is proposed. The main property of the proposed abstraction level is that it is based on the traditional RTL but from the functional point of view it has a moderate rise in abstraction (see Fig. 2).

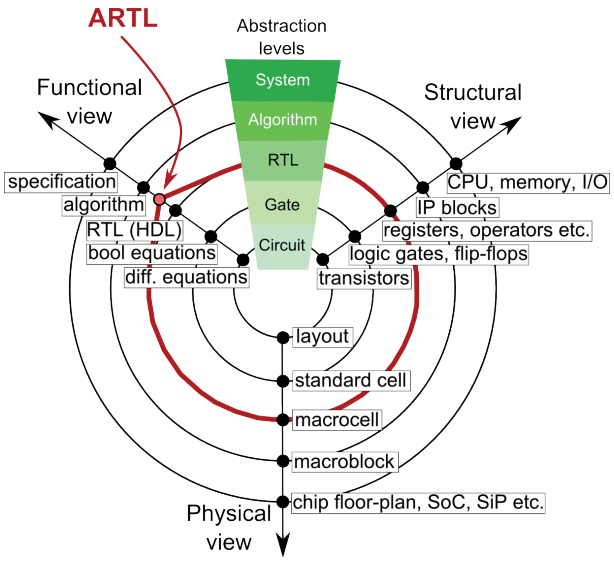

Fig. 2 The GK-diagram with the proposed abstraction level.

Beside the high level functional modeling style ARTL languages and methods have other common, more factual properties.

- Contrary to traditional RTL languages they do not use the concept of the clock cycle and clock signal.
- The pure structural modeling elements (such as components instantiations) are less frequently used or completely missing.
- The resources responsible for scheduling the operations do not appear in an explicit form in the formal language models, the control mechanisms are hidden into semantic rules of the specific language.
- A subset of the datapath resources used to describe the functionality is directly correlated to the traditional RTL models' components (e.g. state registers in BSV and REGISTER resources of LISA).

- They need a specific synthesis mechanism for converting the formal language models into other ones compatible with the subsequent tools (e.g. RTL synthesis tools) of the design flow.

The most important common properties may be summarized in the following definition of ARTL:

A formal language or method represents the ARTL abstraction, when a language-specific subset of the datapath resources used to describe the functionality can be directly mapped to the elements of the target RTL representation, while it describes the controlling mechanisms through language-specific semantic rules.

According to the above presented definition BSV and ADLs may be considered ARTL methods while HLS does not, since the datapath resources of an algorithmic model cannot be directly mapped to their RTL counterparts in the generated RTL model. This is caused by the automated scheduling and resource sharing mechanisms realized by the HLS tools. In an HLS process we cannot tell whether an internal variable in the C code will be a wire or a register in the generated RTL because it depends on the automatically determined pipeline stages. Moreover, we cannot tell how many actual arithmetic circuits will be our operator calls mapped to during the resource sharing which is also automated and can only be indirectly affected by the design constraints.

4 Algorithmic Microarchitecture Description Language (AMDL)

4.1 Target architecture models

It is a typical technique among ADLs and their underlying synthesis tools that they provide target architecture models defining the common properties of the systems that may be obtained with the methods themselves. E.g. the target architecture model of a SystemC-based ADL called ArchC [31] declares that the described system is an ISP with a single-issue pipeline. It also defines a strict interrupt-handling mechanism and it does not make the instruction pointer accessible for the designer. The resources responsible for scheduling the instruction execution are pre-defined and identical in every generated output model (note that ArchC may also be considered an ARTL modeling tool). Since the circuit structures implementing these common details are pre-defined and may be highly optimized, the target architecture models defining many implementation details implicitly mean more efficient, fast and reliable synthesis procedures on the cost of limited freedom with regard to architectural details.

The aim of AMDL's target architecture model is two-fold: *(i)* It defines the structural and behavioral characteristics thus the semantic view (also known as programmer's view in case of high level languages) of the language elements which are indispensable for the designer and *(ii)* it also defines the structure and the behavior which have to be implemented by the synthesis procedure's output model as well. Accordingly, the target architecture model is a common language for the designer and the underlying synthesis tool during the development process. AMDL's target architecture model describes the data processing macrocells as a set of so-called architecture elements which may be recursively integrated into each other. There are three architecture elements, namely the multicycle processor, the data stream processor, and the instruction stream processor. In order to ensure a wide applicability, the architecture elements do not limit the designer regarding the functionality, only minor scheduling and structural properties are pre-defined. During the discussion of the architecture elements' details the following notations are used:

- *ENV (environment)*: resources of the circuit environment
- *CU (control unit)*: resources of the control unit
- *DP (datapath)*: resources of the datapath
- *sig = [id, value]: id, value*: state of a signal (identifier and current value)
- $CS = \{sig_1 ... sig_n\}$: state of the control signals (CU → DP)
- $SS = \{sig_1 ... sig_n\}$: state of the status signals (DP → CU)
- $CO = \{sig_1 ... sig_n\}$: state of the control outputs (CU → ENV)
- $CI = \{sig_1 ... sig_n\}$: state of the control inputs (ENV → CU)
- $DI = \{sig_1 ... sig_n\}$: state of the data inputs (ENV → DP)
- $DO = \{sig_1 ... sig_n\}$: state of the data outputs (DP → ENV)
- *env(...)*: a function defined by the environment
- *cu(...)*: a function defined by the control unit
- *dp(...)*: a function defined by the datapath
- $SE \subseteq CS$: $sig.value \in \{active, inactive\} \ \forall \ sig \in SE$ *(storage enable)*: state of the storage enable signals
- $SE_{assertable} \subseteq SE$: set of the assertable storage enable signals. An SE signal can be activated only if it is in the set $SE_{assertable}$. The data storage resources assigned to the signals being in the set $SE_{assertable}$ work concurrently.
- $S = \{s_1 ... s_n\}$: internal states of the control unit
- $[CS, CO, s_{recent}, s_{next}, s_{init}]: s_{recent}, s_{next}, s_{init} \in S$: recent state of the control unit
- $exit \in \{false, true\}$: exit condition
- $bypass \in \{false, true\}$: bypass condition

Structural properties. The architecture elements comprise two sets of resources, namely control resources and datapath resources. The control structures implement the operation scheduling (see below) and they may not be directly accessed by the designer. The datapath resources implement data manipulation and internal storage. Every instance of this resource type is explicitly declared by the designer. The control structures and the datapath structures communicate with each other and with the environment via unidirectional signals. Figure 3a) shows the general structure of the architecture elements and Fig. 3b) presents an exemplary circuit with the above listed notations.

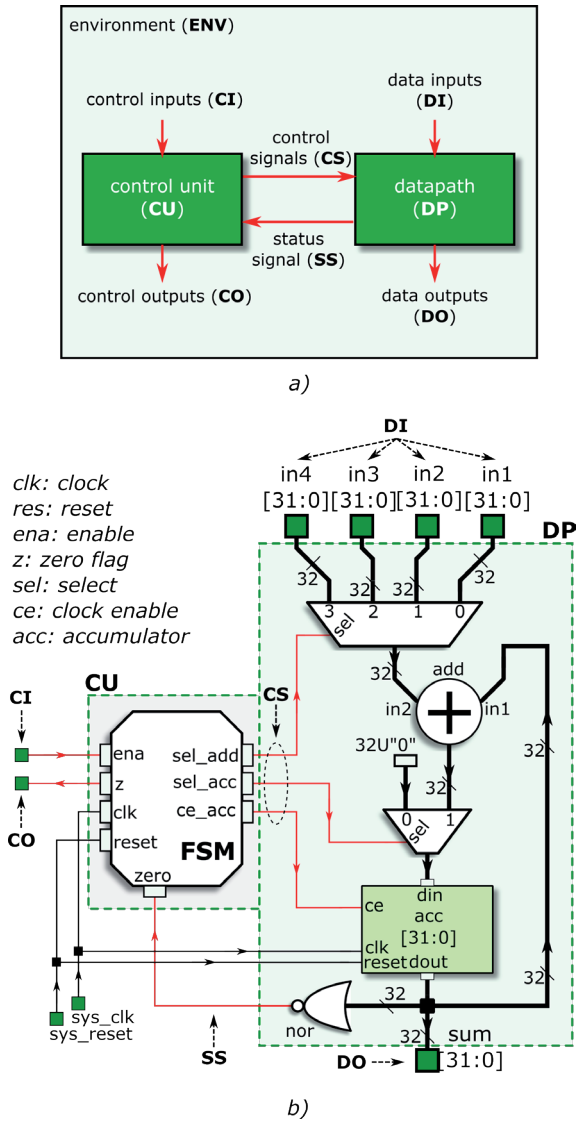

Fig. 3 The general structure of the architecture elements (a) and a circuit example (b).

The structural hierarchy of the architecture elements is shown by the UML [32] class diagram in Fig. 4. The association between the architecture element class and the instruction set class indicate that every architecture element is able to describe ISPs.

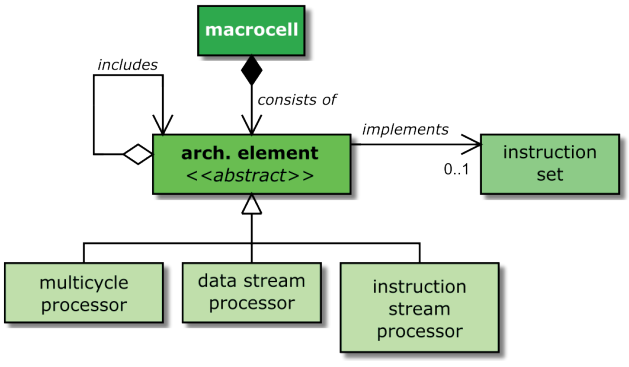

Fig. 4 Structural hierarchy of the architecture elements.

Behavioral properties. The behaviors of the architectural elements are described by specific algorithms. These algorithms are derived from the abstract Mealy Finite State Machine (FSM) model and they involve only minor inherent scheduling attributes. Using the notations defined above the behaviors of the multicycle processor, the data stream processor, and the instruction stream processor are shown in Fig. 5, Fig. 6, and Fig. 7, respectively.

1: $s_{recent} \leftarrow s_{init}$
2: $initialize\ [CO, CS, SS]$
3: $exit \leftarrow false$
4: **while** $(exit = false)$ **do**
5: $\quad sig.value \leftarrow inactive\ \forall\ sig \in SE$
6: $\quad SE_{assertable} \leftarrow cu(s_{recent})$
7: $\quad [CS \backslash SE, CO, s_{next}] \leftarrow cu(s_{recent}, CI, SS)$
8: $\quad SE \leftarrow cu(CI, SS)$
9: $\quad exit \leftarrow cu(s_{recent}, CI, SS)$
10: $\quad s_{recent} \leftarrow s_{next}$
11: $\quad CI \leftarrow env(CO)$
12: $\quad SS \leftarrow dp(CS)$
13: **end while**

Fig. 5 Behavior of the multicycle processor architecture element.

The most important properties of the multicycle processor are that the set $SE_{assertable}$ is a real subset of SE and it can be changed during the operation of the circuit. This means that a certain storage resource included in the datapath can be activated only in the control states it is assigned to.

1: $initialize\ [CO, CS, SS]$
2: $[exit, bypass] \leftarrow false$
3: $SE_{assertable} \leftarrow SE$
4: **while** $(exit = false)$ **do**
5: $\quad sig.value \leftarrow inactive\ \forall\ sig \in SE$
6: $\quad [CS \backslash SE, CO] \leftarrow cu(CI, SS)$
7: $\quad SE \leftarrow cu(CI, SS)$
8: $\quad exit \leftarrow cu(CI, SS)$
9: $\quad CI \leftarrow env(CO)$
10: $\quad SS \leftarrow dp(CS)$
11: **end while**

Fig. 6 Behavior of the data stream processor architecture element.

The difference between the multicycle processor and the data stream processor is that in the first case the set $SE_{assertable}$ is assigned to the recent control state but in the second case this set cannot be changed during the operation. In fact, the control unit of the data stream processor has only a single state and the values of the control signals and control outputs exclusively depend on the control inputs and the status signals.

The instruction stream processor is similar to the data stream processor but it can be switched into multicycle operation mode (bypass) depending on the values of the control inputs and the status signals. The bypass mechanism may be used to implement exception-handling typical of instruction set microprocessors.

```
1:  initialize [CO, CS, SS]
2:  CONCURRENT: [exit, bypass] ← false
3:  SE_assertable ← SE_assertable,concurrent
4:  while ( exit = false ) do
5:      sig.value ← inactive ∀ sig ∈ SE
6:      [CS\SE, CO] ← cu(CI, SS)
7:      SE ← cu(CI, SS)
8:      bypass ← cu(CI, SS)
9:      if ( bypass = true ) then
10:         goto BYPASS
11:     end if
12:     exit ← cu(CI, SS)
13:     CI ← env(CO)
14:     SS ← dp(CS)
15: end while
16:
17: BYPASS: s_recent ← cu(CI, SS)
18: while ( bypass = true ) do
19:     sig.value ← inactive ∀ sig ∈ SE
20:     SE_assertable ← cu(s_recent)
21:     [CS\SE, CO, s_next] ← cu(s_recent, CI, SS)
22:     SE ← cu(CI, SS)
23:     bypass ← cu(s_recent, CI, SS)
24:     s_recent ← s_next
25:     CI ← env(CO)
26:     SS ← dp(CS)
27: end while
28: goto CONCURRENT
```

Fig. 7 Behavior of the instruction stream processor architecture element.

4.2 Language overview

Algorithmic Microarchitecture Description Language (AMDL) is an ARTL modeling language inspired by the aforementioned special RTL modeling tools. The main goal of the language is to improve the efficiency of hand-crafted RTL design's most intuitive tasks such as resource allocation, scheduling, and resource sharing. To achieve this, AMDL borrows the key concepts of HLS and ADLs in the following manner:

- An AMDL model describes the behavior of the system with high-level language constructs, similar to those applied in the HLS environments. However, the basic expressions constituting the high-level language elements provide the designer with comprehensive low-level access to the microarchitectural details.
- Similarly to ADLs, AMDL also defines target architecture models (see Section 4.1) to improve the efficiency of the synthesis procedure. Contrary to those offered by ADLs, these architecture models are more widely applicable since they do not involve any architectural and functional limitations.

Because of the limited available space, instead of the formal definitions of AMDL syntax elements, the grammar of the language is presented in a form of characteristic exemplary expressions. The exact definition of the grammar is provided in Annex A as an Extended Backus-Naur Form (EBNF) description.

4.2.1 Resources

The definition of the ARTL abstraction declares that an ARTL model explicitly includes a set of resources which may be directly mapped to their traditional RTL counterparts in the output model generated by a specific synthesis process. In case of AMDL this set of resources appears in the resource declaration part of the description and it includes all datapath resources of the system, namely the data storing resources (registers and register files), the data manipulating resources (operators), and the interface signals (ports). These resources are functional units communicating via their own interfaces called terminals. In the model body the designer explicitly defines the interconnections of these terminals while expressing the behavior of the system as well. Table 1 summarizes the resources and the interfaces of AMDL resources.

Table 1 AMDL resource types.

Resource type	Resource subtype	Input terminals	Output terminals
interface signal (port)	control input	-	d_{out}: value written on the port
	control output	d_{in}: value to be sent	-
	data input	-	d_{out}: value written on the port
	data output	d_{in}: value to be sent	-
storage	register	d_{in}: value to be stored	d_{out}: stored value
	register file	a_{id}: address input of interface i $d_{in,id}$: value to be stored at address a_{id}	$d_{out,id}$: value stored at address a_{id}
operator	async	defined by the designer	
	sync		
	multicycle		

The interface signals represent the traditional RTL ports. They do not store any data. Table 2 shows the different interface signal types and their declaration syntax.

Table 2 Interface signal types and their declaration syntax.

Signal type	Declaration syntax	Declaration example
control input	`controlport <id>: input [<size>]`	`controlport load: input [1]`
control output	`controlport <id>: output [<size>]`	`controlport ready: output [1]`
data input	`dataport <id>: input [<size>]`	`dataport init: input [8]`
data output	`dataport <id>: output [<size>]`	`dataport cout: output [8]`

The registers are the basic storage elements of AMDL. They are able to store n-bit logic vectors and may be used as internal variables. Register files are sets of registers with unidirectional read and write interfaces. The number of their interfaces is declared by the designer. A register file interface comprises an address vector and a data input/output vector, depending on the direction of the interface. Table 3 shows the different storage resource types and their declaration syntax.

Table 3 Storage resource types and their declaration syntax.

Storage resource type	Declaration syntax	Declaration example
register	`storage <id>: reg [<size>]`	`storage ir: reg [32]`
register file	`storage <id>: regfile [<number of write interfaces>] [<number of write interfaces >] [<address size>] [<data size>]`	`storage rf: regfile [2][2][5][32]`

The data manipulations are performed by operators in AMDL. The operators are similar to functions of high level programming languages. They have an interface through which they can be "called" in the behavioral description of the circuit. Their interface (the number, directions and sizes of their terminals) is defined by the designer. Table 4 shows the different operator types and their declaration syntax.

Table 4 Operator types and their declaration syntax.

Operator type	Declaration syntax	Declaration example
asynchronous	`operator <id>: async (<inputs>) (<outputs>)`	`operator adder: async (opa[32],opb[32])(sum[33])`
synchronous	`operator <id>: sync (<inputs >) (<outputs >)`	`operator alu: sync (func[3],opa[32],opb[32]) (result[32],carry)`
multicycle	`operator <id>: multicycle (<inputs >) (<outputs >)`	`operator nth_root: multicycle (request,arg[32],n[32]) (ready,nth_root[32])`

The asynchronous operators describe combinatorial circuits; their latency is zero, which means that their outputs immediately change whenever their inputs change. The synchronous operators and the multicycle operators describe sequential circuitry. The difference between them is that the synchronous operators' latency is constant while the multicycle operators' may vary depending on the operands. Therefore, multicycle operators' interfaces usually implement hand-shaking.

4.2.2 Semantics of expressions and basic assignments

The key concept of AMDL is that its expressions used in the assignments explicitly describe the circuit structure realizing a particular data transfer. The expressions directly refer to the resources and their terminals not only in case of storage elements but in case of data manipulating resources, operators as well. Moreover, if a value of a register file is read out or a value is written into a register file, the designer can explicitly determine, which interface of the register file should be used to perform the read/write operation. So, contrary to HLS, the designer is responsible to the resource allocation, scheduling and binding tasks. This detailed nature of the basic expressions makes it possible to comprehensively optimize the design.

Figure 8 and Fig. 9 present examples of register, register file, and operator expressions and the intended RTL circuit structures. In case of registers the expressions do not explicitly refer to its terminals. If the expression is a right-value, the name of the register refers to its d_{out} terminal, otherwise the name represents the d_{in} terminal.

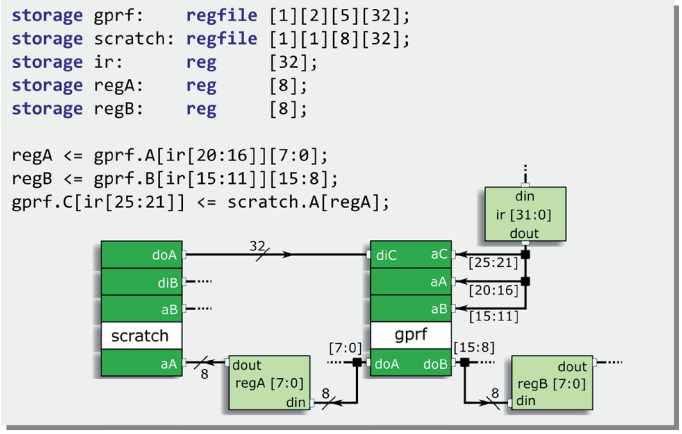

Fig. 8 Register and register file expressions.

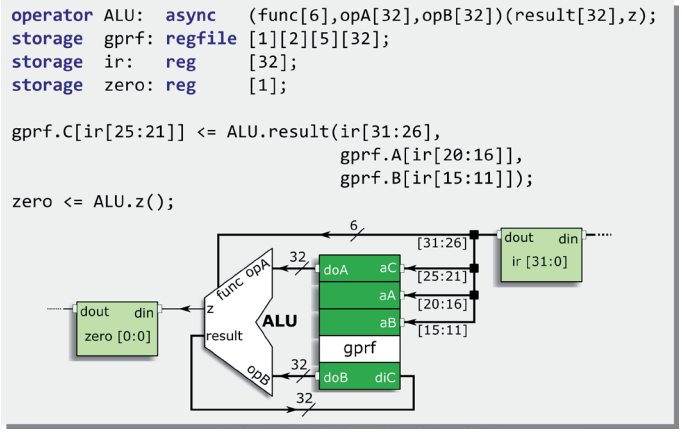

Fig. 9 Operator expressions.

In case of register files the interfaces are identified by letters. Every read and write operations are performed through the explicitly expressed interface.

In case of operator calls the expression defines the operator itself, its output terminal, and optionally a parameter list (this latter one is optional because the operator permanently has an active parameter set which may be omitted).

The semantics of the basic AMDL assignment is defined by a two-step (preparation and writeback), recursive procedure with two parameters. The parameters are used by the control structures using the basic assignments to slightly modify their behavior (see in Section 4.2.3). To formally describe the meaning of the basic assignment the following notations are used:

- *LOAD: [parent,type,id,value]* = {set of the input terminals}
  - *parent*: parent resource of the terminal
  - *type* ϵ {output, regInput, regfileWriteAddress, regfileReadAddress, regfileDataInput, operatorInput}: type of the terminal
  - *id*: identifier of the terminal
  - *value*: logic vector, the current value of the terminal
- *DRIVER: [parent,type,id,value]* = {set of the output terminals}
  - *parent*: parent resource of the terminal
  - *type* ϵ {input, regOutput, regfileOutput, operatorOutput}: type of the terminal
  - *id*: identifier of the terminal
  - *value*: logic vector, the current value of the terminal
- State of the system: $ST = \{ST_{connection} \cup ST_{storage}\}$
  - $ST_{connection}$ = {*dlink*: [*load*,*driver*,*evaluated*] | *load* ϵ *LOAD, driver* ϵ *DRIVER, evaluated* ϵ {true, false}}: active data connections of the system (data link)
  - $ST_{storage}$ = {*strg: [id,value]*}: current content of the registers and register files

The definition of the basic assignment is shown in Fig. 10.

The assignment comprises two main steps; the preparation phase and the writeback phase. In the preparation phase the connections between the resource terminals are established. These connections are realized by the controller FSM by determining the values of the routing resources' (multiplexers) control lines. This change in the control lines causes a chain of reactions throughout the datapath, which settles on the data inputs of storage resources. Figure 11 shows the complete AMDL model of an exemplary counter circuit and its intended architecture. In the preparation phase of the assignment in line 17 the FSM sets the control lines of multiplexers M1 and M2 establishing the emphasized local datapath.

The result of the preparation phase is a logic vector provided by the right-value expression. In the writeback phase this logic vector is stored into the storage element referred to by the left-value expression. If the left-value expression refers to a non-storage element (e.g. a data output port) or the assignment's behavior is modified by the embedding control structure (e.g. structure statement block, see in Section 4.2.3) then the writeback has no effect.

```
1:  procedure BasicAssignment(prepare, writeback)
2:      DLINK_asg = {connections concerned in the assignment}
3:      dlink_wb ∈ DLINK_asg = connection realizing the writeback
4:      if (prepare = true) then
5:          ST_connection ← DLINK_asg
6:          evaluated ← false ∀ dlink ∈ DLINK_asg
7:          for each dlink ∈ DLINK_asg do
8:              if (dlink.evaluated = false) then
9:                  call evaluate(dlink, DLINK_asg)
10:             end if
11:         end for
12:     end if
13:     if (writeback = true ∧ left-value type is storage) then
14:         storage ∈ ST_storage = left-value reg or regfile
15:         storage.value ← dlink_wb.load.value
16:     end if
17:     return
18: end procedure
19:
20: procedure Evaluate(dlink, DLINK_asg)
21:     if (dlink.driver.type ∈ {regfileOutput,operatorOutput}) then
22:         TMP = {tmp | tmp ∈ DLINK_asg :
23:         tmp.load.parent = dlink.driver.parent ∧
24:         tmp.load.type ∈ { operatorInput, regfileReadAddress }}
25:         for each tmp ∈ TMP do
26:             call evaluate(tmp, DLINK_asg)
27:         end for
28:     end if
29:     dlink.load.value ← dlink.driver.value
30:     dlink.evaluated ← true
31:     return
32: end procedure
```

Fig. 10 The formal definition of the AMDL assignment.

```
1  machine counter is
2    controlport load: input  [1];
3    controlport dec:  input  [1];
4    dataport    init: input  [8];
5    dataport    cout: output [8];
6    storage     acc:  reg    [8];
7    operator    add:  async  (opa[8],opb[8])
8                              (sum[8]);
9  begin
10   structure output_port:
11     cout <= acc;
12   end structure;
13   loop
14     if (load = "1") then acc <= init;
15     elsif (dec = "1") then
16       acc <= add.sum(acc,8s"-1");
17     else acc <= add.sum(acc,8u"1");
18     end if;
19   end loop;
20 end machine;
```

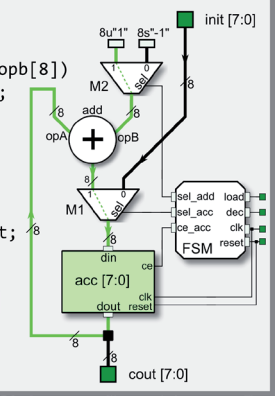

Fig. 11 The AMDL model of a counter circuit.

4.2.3 Control structures

As may be seen in Fig. 11 AMDL uses the elements of structured programming (statement sequences, loops, and decisions) for behavioral modeling. Additionally, it also has specific language structures improving the scheduling capabilities of the language. These language elements are statement blocks

incorporating each other and the basic assignments. The different statement blocks slightly modify the semantics of their included basic assignments by setting its parameters accurately before calling the procedure shown in Fig. 10. Figure 12 shows the definition of the concurrent statement block.

```
1: procedure CONCURRENT(prepare, writeback)
2:     S = {statements inside the block}
3:     for each s ∈ S do
4:         call s(true, false)
5:     end for
6:     for each s ∈ S do
7:         call s(false, true)
8:     end for
9:     return
10: end procedure
```

Fig. 12 Definition of the concurrent statement block.

In a concurrent block the preparation phase of all the assignments included in the block are performed before any write-backs. In other words the assignments inside the same concurrent block are evaluated concurrently. Figure 13 shows the definition of the structure statement block.

```
1: procedure STRUCTURE(prepare, writeback)
2:     S = {statements inside the block}
3:     for each s ∈ S do
4:         call s(true, false)
5:     end for
6:     return
7: end procedure
```

Fig. 13 Definition of the structure statement block.

The structure block is similar to the concurrent block except that it does not perform the writeback phase of its assignments. The structure block is used to define static connections between the resource terminals (e.g. the aforementioned permanent parameters of the operators).

Beside the machine design unit, such as that presented in Fig. 11, another design unit type is available for realizing the behavior of the data stream processor and the instruction stream processor target architecture models. The design unit called pipeline is syntactically and conceptually more than the statement blocks described before; however, its behavior can be discussed with the same formalism. Figure 14 shows the definition of the pipeline design unit.

All pipeline design units are assigned to a host machine, which enables/disables the pipeline based on the control inputs and internal status signals. Multiple pipelines may be assigned to the same host machine. This is a superscalar-like architecture making it possible for more than one instruction at a time to be executed. The pipeline itself comprises a set of concurrently performed assignments organized into stage blocks. During the operation of the pipeline the following subtasks are performed cyclically: After the preparation of the stage blocks' assignments, a so-called observer block monitors the datapath. If any exceptional event occurs, the execution is passed to a bypass-block outside the concurrently executed part of the pipeline. The exception detection is done by conditional statements reading the status signals and control inputs. A single bypass call instruction is assigned to every conditional statement inside the observer block. The bypass blocks' semantics is identical to that used in case of the basic machine; their statements are executed sequentially. There is a special instruction (return) to pass the execution back to the concurrent part. If no exceptional events occur, the writeback of the stage blocks' assignments is performed. Figure 15 shows a three-stage pipelined, unsigned multiplier implemented using the pipeline design unit.

```
1: procedure PIPELINE
2:     STRUCT: str = STRUCTURE block assigned to the pipeline
3:     call str(true, false)
4:     PIPE: STG = {STAGE blocks of the pipeline}
5:     for each stg ∈ STG do
6:         S = {statements inside of stg}
7:         for each s ∈ S do
8:             call s(true, false)
9:         end for
10:    end for
11:    OBSERV: C = {conditions of the OBSERVER block}
12:    for each c ∈ C do
13:        if (c = true) then
14:            goto BYPASS block assigned to c (BPS)
15:        end if
16:    end for
17:    for each stg ∈ STG do
18:        S = {statements of stg}
19:        for each s ∈ S do
20:            call s(false, true)
21:        end for
22:    end for
23:    goto STRUCT
24:    BPS: B = {statements of the BYPASS block}
25:    for each b ∈ B do
26:        call b(true, true)
27:    end for
28:    « stop execution »
29: end procedure
```

Fig. 14 Definition of the pipeline design unit.

4.3 Synthesis process

4.3.1 Implementation scheme-based model transformation

The existing tools generating RTL models usually implement a bottom-up synthesis which means that the low level constructs of the front-end language have generic circuit structures and their back-end language models. The whole output model consists of these relatively small subcircuits and their interconnections. E.g. conditional statements are transformed into multiplexers and priority encoders, HDL loops turn into combinational circuits during RTL synthesis, and during HLS the loops of high level languages may become FSMs with associated datapath element implementing the functionality of the loop body. The synthesis process developed to AMDL follows the top-down method, where an implementation frame is generated first based on the target architecture model used then this frame is complemented with low level implementation details. This synthesis process uses so-called implementation

schemes during model transformation. The implementation schemes define how the AMDL target architecture model elements should be implemented using VHDL. In fact, the implementation schemes are basically subsets of VHDL language elements, which are allowed to participate in constituting the output model and a set of design guidelines how to use them. Figure 16 and Fig. 17 illustrate this model generation method using the exemplary pipeline presented in Fig. 15. The possible structural elements and interconnections of the pipeline architectural model, which are not utilized in the example, are denoted by dashed lines and frames.

```
1  machine m1 is                                   m1 is a host
2    controlport ena: input [1];                   machine including
3  begin                                           a single pipeline
4    loop                                          called p1.
5      if ( ena = "1" ) then p1.start;             The host enables
6      else p1.stop; end if;                       and disables
7      if ( p1.overflow = "1" ) then               the pipeline.
8         p1.stop;
9      end if;
10   end loop;
11 end machine;
12
13 pipeline p1 of machine m1 is
14   controlport overflow: output  [1];
15   dataport    p_prod:   output  [32];
16   -- additional internal resource declarations...
17 begin
18   structure p_prod <= product[31:0]; end structure;
19   stage s1:                                     p1 consists of
20     overflow <= "0";                            concurrently executed
21     tmp1 <= mul1.prod(op_a,op_b[15:0]);         statements organized
22     tmp2 <= mul2.prod(op_a,op_b[31:16]);        into pipeline stages.
23   end stage;
24   stage s2:
25     tmp3 <= tmp1;
26     tmp4 <= shl16.result(tmp2);
27   end stage;
28   stage s3:
29     product <= add.sum(tmp3,tmp4);
30   end stage;
31   observer                                      The observer detects
32     if ( product[63:32] /= 32u"0" ) then        exceptional events
33       bypass(overflow);                         in the pipeline and
34     end if;                                     passes the control
35   end observer;                                 to a bypass block.
36   bypass overflow:
37     overflow <= "1";                            The sequential bypass block
38   end bypass;                                   performs exception handling.
39 end pipeline;
```

Fig. 15 A three-stage pipelined, unsigned multiplier.

Capturing overall architecture. The AMDL model of the multiplier implies that the so-called instruction stream processor abstract model should be used. The structural RTL implementation scheme of the instruction stream processor discussed later in this section could include a single host machine and multiple pipelines assigned to it. The host machine consists of a FSM and a datapath but the multiplier does not require any data processing resources in the host machine's datapath. Moreover, the abstract model of the pipelined multiplier only needs a single pipeline. The pipeline itself also comprises a FSM and a datapath. There are two dedicated signals between the FSMs, namely the start/stop control signal and the overflow status flag.

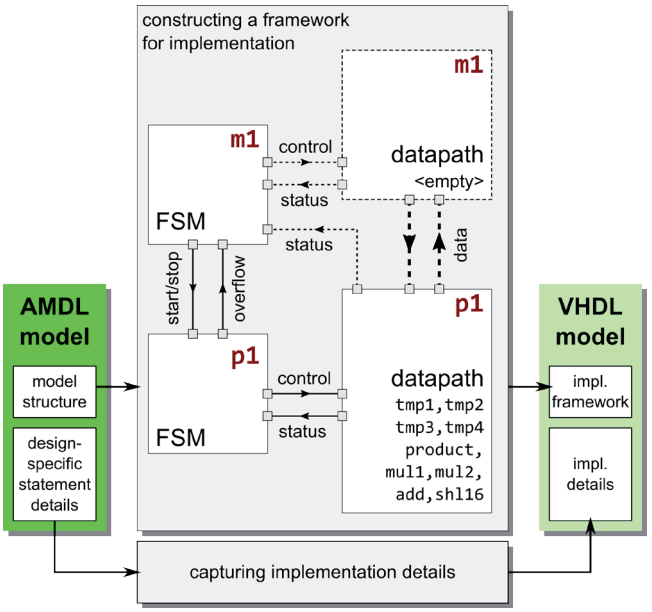

Fig. 16 Implementation framework construction.

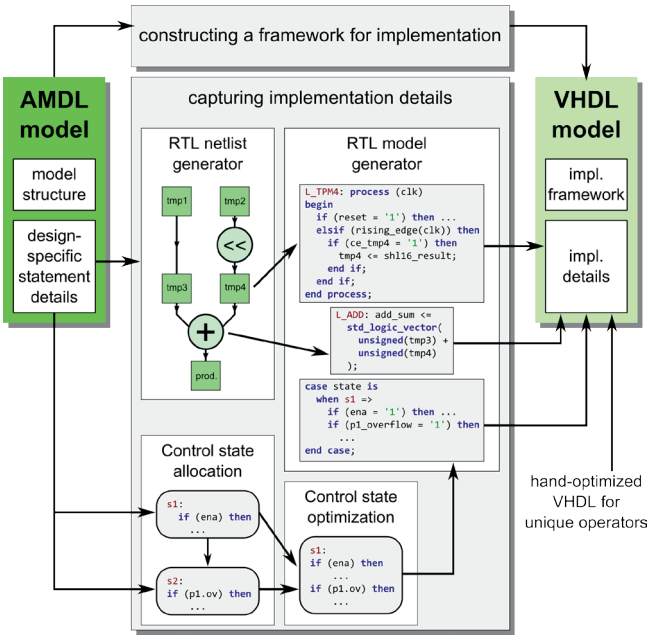

Fig. 17 Embedding the implementation details.

Capturing implementation details. The above described overall architecture is captured based on the AMDL model structure but the details, such as exact states of the FSMs, storage and arithmetic operators inside the datapath, are determined based on the statements included in the AMDL description. The datapath is generated from the assignment statements; the RTL netlist is constructed directly by the AMDL designer by explicitly expressing the connections between storage and arithmetic/logic elements.

The model transformation algorithm is responsible for resolving the multiply driven resource inputs by inserting the appropriate multiplexers into the netlist. Once the RTL netlist is complete, the model generator algorithm creates the HDL

models of the datapath resources and organizes them into language constructs defined by the implementation scheme. If the AMDL model includes design-specific operators then the datapath should be complemented by their VHDL models as well.

The control unit generation is based on specific mapping rules determining the required control states implementing the different AMDL control structures. The control unit generation includes two main steps. The first is the control state allocation step, in which the model transformation algorithm performs the above mentioned mapping between AMDL control structures and their VHDL implementations. In this step a hierarchical data structure (practically a tree) containing FSM-snippets is generated. In the second step the required clock cycles are minimized by detecting the possible concurrencies between the control states. The RTL model generator then creates the whole FSM implementation based on the optimized tree of control states.

4.3.2 VHDL implementation schemes

We have developed two implementation schemes called behavioral RTL and structural RTL models. Their naming reflects the fact that these two models represent slightly different levels of abstraction within RTL. Behavioral RTL is a compact, one-process description of FSMs, which includes the data storage resources as internal signals and the data manipulating resources as operator calls embedded into the FSM's behavioral description. Figure 18 shows an exemplary behavioral RTL model of an accumulator circuit.

```
architecture bhv_rtl of accumulator  is
  signal ...
begin
  CTRL: process (clk,reset)
  begin
    if (reset = '1') then state <= s1; z <= '0';
                          acc <= (others => '0');
    elsif (rising_edge(clk)) then
      case state is
        when s1 => if (ena = '1') then acc <= (others => '0');
                                       state <= s2;
                   end if;
        when s2 => acc <= in1 + in2; state <= s3;
        when s3 => acc <= acc + in3; state <= s4;
        when s4 => acc <= acc + in4; state <= s5;
        when s5 => if (acc = X"00") then z <= '1';
                   else z <= '0'; end if;
                   state <= s1;
        when others => report "unknown state" severity failure;
      end case;
    end if;
  end process;
  sum <= acc;
end architecture;
```

Fig. 18 Behavioral RTL model of an accumulator circuit.

The structural RTL model is a more detailed one with regard to the circuit structure. It describes the AMDL resources declared in the declaration part as independent entity-architecture pairs interconnected in a separate datapath model. In this case the FSM scheduling the operation of the datapath resources is implemented as a separate entity-architecture pair as well. Figure 19 shows certain details of the accumulator circuit's structural RTL implementation.

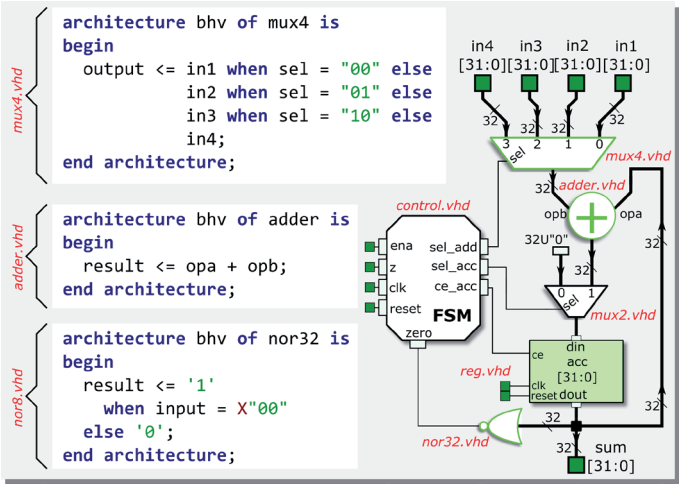

Fig. 19 Structural RTL model of an accumulator circuit.

The steps of the top-down AMDL to VHDL model transformation are the following:

1. An implementation frame is created which comprises the VHDL language elements describing the details which are identical in every model using the same target architecture model (e.g. general model structure and hierarchy, FSM templates including common states and signals, etc.)
2. The implementation frame is complemented with the design-specific details. The pre-defined VHDL counterparts of the AMDL resources are integrated into the datapath model and the control states realizing the applied control structures are embedded into the FSM templates.

5 A recommended AMDL-based design flow

Traditional RTL design includes several subtasks which may be classified based on the amount of required intuitive decisions. In our discussion resource allocation, operation scheduling, and binding are considered completely intuitive tasks, therefore they are called primary RTL subtasks. An RTL designer also has to deal with subtasks which mainly depend on chip-level decisions or may be performed mechanically. These secondary RTL subtasks include timing model selection (latch vs. flip-flop), phase signal generation and propagation, designing the reset mechanism and reset circuitry etc. AMDL-based design flow represents a middle course between HLS and hand-optimized RTL in the following manner:

- It provides the designer with high level language constructs in order to improve design time.
- It forces the designer to manually perform the primary RTL subtasks in order to ensure the possibility of comprehensive datapath and scheduling optimization.
- It automatizes the secondary RTL subtasks.

Figure 20 shows the intended AMDL-based design flow.

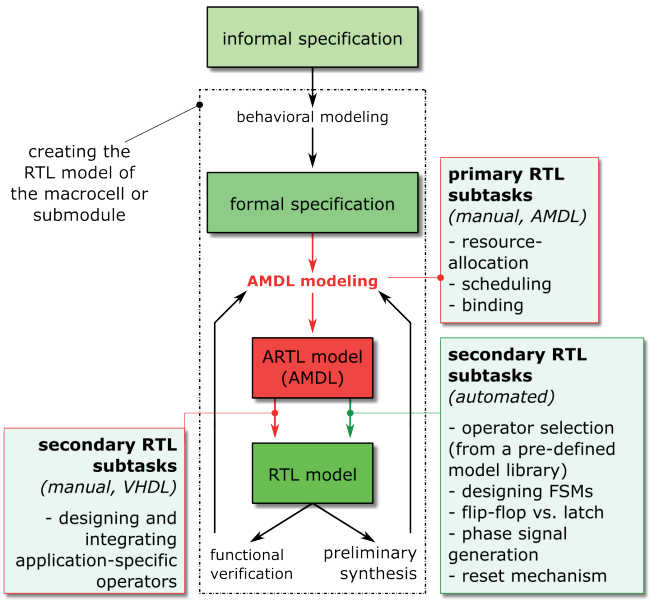

Fig. 20 Integrating AMDL into the traditional design flow.

The AMDL language is basically a very specific means for the RTL designer, therefore it incorporates the traditional design flow after the formal specification (usually an executable specification described with a high level programming language such as C or C++) is done. The primary RTL subtasks are then performed in the language environment provided by AMDL. Once the AMDL model is prepared, the secondary RTL subtasks may be performed using the synthesis method presented in Section 4.3. Note that the AMDL model itself is not functionally complete, since it does not describe the operators' behavior. There are two ways to complete the output model; *(i)* An RTL component library can be prepared and the required elements of it can be selected for complementing the output model, since the operators embedded in an AMDL model are typically widely used arithmetic/logic circuits whose VHDL representations are available in generic forms. *(ii)* If the operator's functionality is application-specific, its VHDL model may be prepared by hand. In extreme cases, when the area and/or timing requirements are very rigorous, the designer may want to implement all operators by hand. In this case the AMDL synthesis tool is used only for generating the VHDL model of the control unit and for creating a "shell" for the datapath resources which can be manually completed with handmade, highly optimized code.

6 Experimental results

Numerous test systems have been developed to ascertain the efficiency of the proposed modeling method and to ensure that the qualities of the generated RTL models are sufficient. Because of the specific synthesis method based on the implementation schemes discussed before, the following investigation could be performed: The implementation schemes define strictly how the elements of an algorithmic specification should be implemented in the RTL model. This means that the behavioral RTL and the structural RTL implementations can be created not only with the AMDL synthesis tool we developed but by hand-crafted RTL coding as well. To compare hand-optimized and AMDL-based design efficiencies, the traditional hand-optimized RTL and the above discussed AMDL-based design flow have been performed concurrently. The behavioral RTL and structural RTL implementation schemes were used in both design flows as guidelines for implementation details. Since the output RTL models obtained were practically identical, the development times and manually prepared code sizes could be directly compared. The test systems have the following features:

- MULT: 32×32-bit unsigned shift&add multiplier.
- PIEZO: Application-specific digital pre-processor developed for a piezoresistive MEMS force sensor system. (1) gives the realized equation. E, a, b, π, l, A, V_T, n, and V_{ref} are run-time configurable parameters characteristic of the MEMS structure and the analog read-out circuitry which the digital post-processor is connected to, and $V_{out,d}$ is the output of the A/D converter. It uses 32-bit fixed-point arithmetic, the precision is synthesis parameter.

$$F = \frac{Eab^2}{3\pi l\left(\frac{AV_T}{2} - \frac{V_{ref}V_{out,d}}{2^n}\right)} \times \frac{V_{ref}V_{out,d}}{2^n} \tag{1}$$

- FIR4: 4-channel FIR filter including an SPI interface circuitry. Through this programming interface the order (up to 255) and the coefficients of the channels may be configured in runtime. The circuit uses 32-bit fixed-point arithmetic with a precision of 2^{-20}.
- TYLR: An arithmetic unit capable of calculating the values of *sin*, *ln* and *exp* functions in a limited range based on their Taylor-polynomials. The required degree and the base of the polynomials are automatically determined based on the argument. The circuit uses 32-bit fixed-point arithmetic with a precision of 2^{-26}.
- FFT: Generic n-point Fast Fourier Transform unit implementing the Cooley-Tukey Radix-2 algorithm where n is a synthesis parameter. The circuit uses 32-bit fixed-point arithmetic with a precision of 2^{-20}. General-purpose input and an output FIFO interfaces are also provided. The circuit reads the samples in natural order (bit-reverse reordering is included).
- MINK: Arithmetic unit calculating the Minkowski-norm of vectors defined by (2),

$$d_{ij} = \sqrt[r]{\sum_{k=1}^{p}\left(w_k^r \times \left|x_{ik} - x_{jk}\right|^r\right)} \tag{2}$$

- where p and r are run-time configurable parameters. The circuit uses 32-bit fixed-point arithmetic with a precision of 2^{-20}. It implements the Newton-iteration method for r^{th} root calculation including a shift&subtract divisor.
- ISP_1: Programmable processor implementing a 3-address instruction set with a 3-stage 32-bit wide pipelined datapath including a 32-word register file, a DSP ALU and a 64-bit wide accumulator storing the results of the 5 DSP instructions. The core provides high-speed external input and output FIFO interfaces. To minimize the control hazard occurrence the core performs 2-bit dynamic branch prediction with a 256-entry branch history table. To prevent data hazards the microarchitecture implements data forwarding.
- ISP_2: Educational case study for hardware accelerator-based ASIPs. The 32-bit wide 5-stage pipelined datapath implements 64 instructions (based on MIPS and SPARC ISAs) including 9 DSP instructions. The data forwarding and branch prediction system is equivalent to that applied in ISP_1. Additionally, it includes a loosely-coupled accelerator with a 32-word input parameter table and a dedicated interface to the data cache controller. This general-purpose interface makes it possible to implement different accelerator functionalities for different application domains. In the default configuration the accelerator implements fixed-point and integer division algorithms (shift&subtract). The data cache interface is WISHBONE-compatible.

Table 5 shows the results of the development efficiency investigations in case of the above test systems.

Table 5 AMDL vs. hand-crafted VHDL development effort comparisons.

Test system	Dev. time (PH)		VHDL LOC (bhv / str)	AMDL LOC (AMDL + VHDL)
	VHDL	AMDL		
MULT	3	0.3	100 / 180	40 + 0
PIEZO	6	1.5	240 / 730	190 + 0
FIR4	20	6	390 / 730	210 + 0
TYLR	10.2	3.4	430 / 720	300 + 0
FFT	60-70	14.6	770 / 1300	440 + 0
MINK	17.7	4.2	430 / 620	420 + 0
ISP_1	280-300	50-60	600 / 2900	500 + 190
ISP_2	400-450	150-170	1900 / 3800	800 + 900

In Table 5 PH stands for Person-Hour and LOC stands for Lines of Code. In the column labeled VHDL LOC the numbers indicate the code sizes of the behavioral RTL and the structural RTL implementations respectively. In the last column the VHDL code sizes of the unique operators are also indicated, since these code snippets have to be prepared by hand in the AMDL-based design flow as well. Zero VHDL code size in this column means that the applied operators were available as reusable library elements. The conclusion can be drawn that the AMDL-based design flow ensures a much more effective development than tradition hand-crafted RTL coding, while it also ensures the quality of the results because of its synthesis process based on strictly defined implementation schemes.

7 A qualitative comparison of IP delivery methods

The aims and scopes of the design methods discussed in Section 2 are very similar to those addressed by AMDL, therefore a comparison of them is provided in this section.

BSV has been developed for describing parallelized data processing constructs. The designer can concentrate on simple pieces of the system without needing to explicitly handle concurrencies. The atomic transaction-based semantics of BSV has proved to be especially useful in case of flexible pipelines which are not directly supported in AMDL. Contrary to this, AMDL is effective in describing algorithmic content, which is favorable in case of SoCs underlying CPS applications, since the specifications of such systems, including the components intended to be realized in a hardware accelerator, are usually described with a high level algorithm that can be more easily transformed into AMDL than into BSV.

Similarly to AMDL, ADLs also provide target architecture models which their specific synthesis tools are based on. In case of AMDL the target architecture model is more widely applicable since there are no significant limitations regarding the microarchitecture. This advantage of AMDL over ADLs becomes a disadvantage when the specification requirements fit well with the target architecture model of a certain ADL. The more specific architecture model may result in a more optimized output model, unless the hand-written HDL inclusion of AMDL is exploited but in this case the increased development time has to be taken into consideration as well.

AMDL provides a solution for the best known disadvantage of C-based HLS procedures, namely that the microarchitectural details are hidden from the designer. In the AMDL-based design the resource-allocation, scheduling, and binding tasks are purposely given to the designer for hand-optimization, while in case of HLS tools these tasks are automatized. That means that the aforementioned design time improvement ensured by AMDL is absent in case of the main target applications of HLS (DSP algorithms with regular computation models). However, the hand-crafted, detailed access to scheduling properties of the AMDL-based design may lead to a better result in case of CPS applications' hardware accelerators which have to ensure predictable response times and latencies.

Table 6 shows a summary of the advantages and disadvantages of the design methods, which may be used for IP delivery of SoCs.

Table 6 Advantages and drawbacks of the discussed modeling methods.

Method	Advantages	Drawbacks
hand-written RTL	• no architectural limitations • high-quality hardware model	• time-consuming • error-prone
BSV	• high-quality hardware model • efficient means for modeling flexible pipelines	• algorithmic content is difficult to express • limited design space exploration capabilities
ADLs	• rapid prototyping • detailed microarchitectural model	• restricted applicability of the generated RTL model (post-processing needed) • limited target architecture model
HLS	• rapid prototyping • algorithmic design style • high-quality hardware model	• limited application domain • syntactic variance • limited RTL optimization capabilities
AMDL	• accessibility of microarchitectural details • high-quality hardware model • flexible target architecture model • decreased design time compared to more general design methods	• lack of means for modeling flexible pipelines • increased design time compared to more specific methods

8 Conclusions

The SoCs supporting CPS applications are facing a demand for constantly growing computational capacities while the significance of the development time is also increasing because of the rigorous time-to-market requirements. RTL design is a crucial step in the design flow from the viewpoint of both of these perspectives. This paper gives an overview of design techniques used to improve the efficiency of RTL design of complex SoCs' macrocells. To cope with the known problems of these techniques, a novel modeling language (called AMDL) and RTL model generation method has been developed. Contrary to the previous publications related to the proposed solution this article details on the formal definitions of the syntactic and semantic elements of the language. Based on the design efficiency investigations it may be concluded that the proposed modeling means is a promising candidate for fulfilling the gap between high level synthesis tools optimized for development effort and hand-optimized RTL used for comprehensive architectural optimization. However, further research in the direction of highly parallelized constructs and dynamically scheduled pipelined architectures should be performed to extend the presented method's scope, improving its applicability hereby.

Acknowledgement

The work was supported by the EuroCPS (No. 644090) Horizon 2020 Project of the EU.

References

[1] Ahmed, S. H., Gwanghyeon, K., Dongkyun, K. "Cyber Physical System: Architecture, applications and research challenges." In: Wireless Days (WD), Valencia, Spain, Nov. 13-15, 2013. pp. 1-5. DOI: 10.1109/wd.2013.6686528

[2] Taherkordi, A., Eliassen, F. "Towards Independent In-Cloud Evolution of Cyber-Physical Systems." In: IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA), Hong Kong, China, Aug. 25-26, 2014. pp. 19-24. DOI: 10.1109/cpsna.2014.12

[3] Wolf, M., Feron, E. "What don't we know about CPS architectures?" In: Design Automation Conference (DAC), San Francisco, CA, USA, June 8-12, 2015. pp. 1-4. DOI: 10.1145/2744769.2747950

[4] Sarma, S., Dutt, N., Gupta, P., Venkatasubramanian, N., Nicolau, A. "CyberPhysical-System-On-Chip (CPSoC): A self-aware MPSoC paradigm with cross-layer virtual sensing and actuation." In: Design, Automation & Test in Europe Conference and Exhibition (DATE), Grenoble, France, March 9-13, 2015. pp. 625-628. DOI: 10.7873/date.2015.0349

[5] Sarma, S., Dutt, N. "FPGA emulation and prototyping of a cyberphysical-system-on-chip (CPSoC)." In: 25th International Symposium on Rapid System Prototyping (RSP), New Delhi, India, Oct. 16-17, 2014. pp. 121-127. DOI: 10.1109/rsp.2014.6966902

[6] Hamalainen, T. D., Salminen, E. "Gamification of System-on-Chip design." In: International Symposium on System-on-Chip (SoC), Tampere, Finland, Oct. 28-29, 2014. pp. 1-8. DOI: 10.1109/issoc.2014.6972441

[7] Lichen, Z. "An integration approach to specify and model automotive cyber physical systems." In: International Conference on Connected Vehicles and Ecpo (ICCVE), Las Vegas, NY, USA, Dec. 2-6, 2013. pp. 568-573. DOI: 10.1109/iccve.2013.6799856

[8] Chengyuan, Y., Song, J., Xuan, L. "An Architecture of Cyber Physical System Based on Service." In: International Conference on Computer Science & Service System (CSSS), Nanjing, China, Aug. 11-13, 2012. pp. 1409-1412. DOI: 10.1109/csss.2012.355

[9] Lichen, Z. "Modeling large scale complex cyber physical control systems based on system of systems engineering approach." In: 20th International Conference on Automation and Computing (ICAC), Cranfield, England, Sept. 12-13, 2014. pp. 55-60. DOI: 10.1109/iconac.2014.6935460

[10] Berger, R., Chadwick, S., Chan, E., Ferguson, R., Fleming, P., Gilliam, J., Graziano, M., Hanley, M., Kelly, A., Lassa, M., Bin, L., Lapihuska, R., Marshall, J., Miller, H., Moser, D., Pirkl, D., Rickard, D., Ross, J., Saari, B., Stanley, D., Stevenson, J. "Quad-core radiation-hardened system-on-chip power architecture processor." In: IEEE Aerospace Conference, Big Sky, MT, USA, March 7-14, 2015. pp. 1-12. DOI: 10.1109/aero.2015.7119114

[11] Bogdap, P. "A cyber-physical systems approach to personalized medicine: Challenges and opportunities for NoC-based multicore platforms." In: Design, Automation & Test in Europe (DATE), Grenoble, France, March 9-13, 2015. pp. 253-258. DOI: 10.7873/date.2015.1127

[12] Wei, W., Aziz, M. K., Hantao, H., Hao, Y., Hoay, B. G. "A real-time cyber-physical energy management system for smart houses." In: IEEE PES Innovative Smart Grid Technologies Asia (ISGT), Perth, WA, Western Australia, Nov. 13-16, 2011. pp. 1-8. DOI: 10.1109/isgt-asia.2011.6167084

[13] Shoukry, Y., El-Kharashi, M. W., Hammad, S. "MPC-On-Chip: An Embedded GPC Coprocessor for Automotive Active Suspension Systems." *IEEE Embedded System Letters*. 2(2), pp. 31-34. 2010. DOI: 10.1109/les.2010.2051794

[14] Brackenbury, L. E. M., Plana, L. A., Pepper, J. "System-on-Chip Design and Implementation." *IEEE Transaction on Education*. 53(2), pp. 272-281. 2010. DOI: 10.1109/te.2009.2014858

[15] Qiang, Z., Oishi, R., Hasegawa, T., Nakata, T. "Integrating UML into SoC design process." In: Proceeding of Design, Automation and Test in Europe (DATE), Munich, Germany, March 7-11, 2005. Vol. 2. pp. 836-837. DOI: 10.1109/date.2005.186

[16] Schattkowsky, T. "UML 2.0 – overview and perspectives in SoC design." In: Proceeding of Design, Automation and Test in Europe (DATE), Munich, Germany, March 7-11, 2005. Vol. 2. pp. 832-833. DOI: 10.1109/date.2005.320

[17] Nikhil, R. "Bluespec System Verilog: efficient, correct RTL from high level specifications." In: IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), San Diego, CA, USA, June 22-25, 2004. pp. 69-70. DOI: 10.1109/memcod.2004.1459818

[18] Chung, E. S., Hoe, J. C. "High-Level Design and Validation of the BlueSPARC Multithreaded Processor." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 29(10), pp. 1459-1470. 2010. DOI: 10.1109/tcad.2010.2057870

[19] Nikolov, H., Rao, A., Deprettere, E. F., Nandy, S. K., Narayan, R. "A H.264 decoder: A design style comparison case study." In: Conference Record of the 43rd Asilomar Conference on Signals, Systems and Computers (ACSSC), Pacific Grove, CA, USA, Nov. 1-4, 2009. pp. 236-242. DOI: 10.1109/acssc.2009.5470115

[20] Tradowsky, C., Harbaum, T., Deyerle, S., Becker, J. "LImbiC: An adaptable architecture description language model for developing an application-specific image processor." In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Natal, South Africa, Aug. 5-7, 2013. pp. 34-39. DOI: 10.1109/isvlsi.2013.6654619

[21] Schliebusch, O., Chattopadhyay, A., Witte, E. M., Kammler, D., Ascheid, G., Leupers, R., Meyr, H. "Optimization techniques for ADL-driven RTL processor synthesis." In: 16th International Workshop on Rapid System Prototyping (RSP), Montreal, Canada, June 8-10, 2005. pp. 165-171. DOI: 10.1109/rsp.2005.36

[22] Schliebusch, O., Chattopadhyay, A., Leupers, R., Ascheid, G., Meyr, H., Steinert, M., Braun, G., Nohl, A. "RTL processor synthesis for architecture exploration and implementation." In: Proceedings of Design, Automation & Test in Europe Conference and Exhibition, Paris, France, Feb. 16-20, 2004. Vol. 3. pp. 156-160. DOI: 10.1109/date.2004.1269223

[23] Taghavi, T., Pimentel, A. D., Thompson, M. "System-level MP-SoC design space exploration using tree visualization." In: 7th Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), Grenoble, France, Oct. 15-16, 2009. pp. 80-88. DOI: 10.1109/estmed.2009.5336816

[24] Coussy, P., Gajski, D. D., Meredith, M., Takach, A. "An Introduction to High-Level Synthesis." *IEEE Design & Test of Computers*. 26(4), pp. 8-17. 2009. DOI: 10.1109/mdt.2009.69

[25] Martin, G., Smith, G. "High-Level Synthesis: Past, Present, and Future." *IEEE Design & Test of Computers*. 26(4), pp. 18-25. 2009. DOI: 10.1109/mdt.2009.83

[26] Arvind, Nikhil, R. S., Rosenband, D. L., Dave, N. "High-level synthesis: an essential ingredient for designing complex ASICs." In: International Conference on Computer Aided Design (ICCAD), San Jose, CA, USA, Nov. 7-11, 2004. pp. 775-782. DOI: 10.1109/iccad.2004.1382681

[27] "Achieving Timing Closure with Bluespec SystemVerilog." White Paper, Bluespec, Inc., 2004. URL: http://www.bluespec.com/forum/download.php?id=22

[28] Nikhil, R., Czeck, K. "*BSV by Example*." Chapter 1.3, Bluespec Inc., 2010.

[29] Horváth, P., Hosszú, G., Kovács, F. "A Proposed Synthesis Method for Application-Specific Instruction Set Processors" *Microelectronics Journal*. 46(3), pp. 237-247. 2015. DOI: 10.1016/j.mejo.2015.01.001

[30] Horváth, P., Hosszú, G. "ARTL-Based Hardware Synthesis to Non-Heterogeneous Standard Cell ASIC Technologies." *Journal of Low Power Electronics*. 11(3), pp. 278.289. 2015.

[31] Rigo, S., Araujo, G., Bartholomeu, M., Azavedo, R. "ArchC: A systemC-based architecture description language." In: 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Foz do Iguacu, Brazil, Oct. 27-29, 2004. pp. 66-73. DOI: 10.1109/sbac-pad.2004.8

[32] Unified Modeling Language™ (UML®) Resource Page. [Online] Available from: www.uml.org [Accessed: 22th August 2015]

Annex A – The EBNF description of AMDL

```
system_model ::= (machine_definition | pipeline_
definition | isa_definition)+

machine_definition ::= 'machine' id 'is' declaration*
'begin' functional_statement* 'end' 'machine' ';'

pipeline_definition ::= 'pipeline' id 'of' 'machine'
id 'is' ( ( declaration* 'begin' functional_
statement* 'end' 'pipeline' ';' ) | ( 'like' id 'of'
'machine' id ';' ) )

isa_definition ::= 'isa' 'of' ( ('machine' id) |
('pipeline' id '.' id) ) 'is' isa_address_length_
definition isa_word_length_definition isa_opcode_
length_definition 'begin' instruction_definition*
'end' 'isa' ';'

id ::= letter (letter | decimal_number)*

declaration ::= resource_declaration | constant_
declaration

functional_statement ::= label? ( assignment |
conditional_statement | loop | block | control_
statement ) ';'

isa_address_length_definition ::= 'address' 'length'
':' decimal_number+ ';'
```

isa_word_length_definition ::= 'word' 'length' ':' decimal_number+ ';'

isa_opcode_length_definition ::= 'opcode' 'length' ':' decimal_number+ ';'

instruction_definition ::= id ':' constant_literal instruction_parameter_list ';'

letter ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '_'

decimal_number ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

resource_declaration ::= controlport_declaration | dataport_declaration | reg_declaration | regfile_declaration | operator_declaration

constant_declaration ::= 'constant' id ':' constant_literal ';'

label ::= 'controlpoint' id ':'

assignment ::= left_expression '<=' right_expression

conditional_statement ::= 'if' condition 'then' functional_statement* elsif_statement* ('else' functional_statement*)? 'end' 'if'

loop ::= 'loop' functional_statement* 'end' 'loop'

block ::= concurrent_block | structure_block | stage_block | bypass_block | observer_block

control_statement ::= 'break' | 'continue' | 'stop' | ('wait' '(' (decimal_number)+ ') ') | ('redirect to' id) | (id '.' 'start') | (id '.' 'stop') | 'return' | ('bypass' '(' id ')')

constant_literal ::= (decimal_number+ ('B' | 'b' | 'H' | 'h' | 'U' | 'u' | 'S' | 's'))? '"' "-"? hexa_number+ '"'

instruction_parameter_list ::= instruction_word_fields ('+' instruction_word_fields)*

controlport_declaration ::= 'controlport' id ':' ('input' | 'output') width_definition ';'

dataport_declaration ::= 'dataport' id ':' ('input' | 'output') width_definition ';'

reg_declaration ::= 'storage' id ':' 'reg' width_definition (":=" constant_literal)? ';'

regfile_declaration ::= 'storage' id ':' 'regfile' width_definition width_definition width_definition width_definition ';'

operator_declaration ::= 'operator' id ':' ('async' | 'sync' | 'multicycle') operator_port_list ';'

left_expression ::= '-' | operator_expression | regfile_expression | simple_reference_expression

right_expression ::= '-' | operator_expression | regfile_expression | simple_reference_expression | constant_literal

condition ::= ('(' id '.' 'stopped' ')') | ('(' (arithmetic_relation_expression | condition) (logic_relation (arithmetic_relation_expression | condition))* ')')

elsif_statement ::= 'elsif' condition 'then' functional_statement*

concurrent_block ::= 'concurrent' (id ':')? functional_statement* 'end' 'concurrent'

structure_block ::= 'structure' (id ':')? functional_statement* 'end' 'structure'

stage_block ::= 'stage' (id ':')? functional_statement* 'end' 'stage'

bypass_block ::= 'bypass' id ':' functional_statement* 'end' 'bypass'

observer_block ::= 'observer' (id ':')? functional_statement* 'end' 'observer'

hexa_number ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'

simple_reference_expression ::= id bit_index?

instruction_word_fields ::= '(' (instruction_field_definition (',' instruction_field_definition)*)? ')'

width_definition ::= '[' decimal_number+ ']'

operator_port_list ::= '(' id width_definition? (',' id width_definition?)* ')' '(' id width_definition? (":=" constant_literal)? (',' id width_definition? (":=" constant_literal)?)* ')'

regfile_expression ::= id '.' regfile_port_id '[' (right_expression | decimal_number+)? ']' bit_index?

operator_expression ::= id '.' id bit_index? '(' (((id '=>')? right_expression) (',' ((id '=>')? right_expression))*)? ')'

arithmetic_relation_expression ::= right_expression arithmetic_relation right_expression

logic_relation ::= 'and' | 'or'

instruction_field_definition ::= decimal_number+ ':' decimal_number+

bit_index ::= '[' decimal_number+ (':' decimal_number+)? ']'

regfile_port_id ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'