XML specification and tools for Automatic SoC Generation

Màrius Montón[1], Oriol Font[1], Jaime Joven[1], Pere Garcia[2], Lluís Terés[3], Jordi Carrabina [1]

[1]Universitat Autonoma de Barcelona. [2]EPSON Electronics Europe. [3]Centro Nacional de Microelectronica.

{marius.monton, jordi.carrabina}@uab.es. {joseporiol.font, jaime.joven}@campus.uab.es. pere.garcia@epson-electronics.de. lluis.teres@cnm.es

***Abstract* — This paper presents a methodology for building SoC systems starting from XML specifications oriented to system on chip platforms with virtual component (IPs) integration.**

This methodology includes the implementation of a set of tools, written in Java, to generate the whole set of Hardware (Verilog) and Software (C) files required to synthesize and simulate an entire AMBA-based SoC.

This methodology has been validated for a set of examples using a prototyping system that contains one FPGA and one ARM processor.

***Index Terms* — XML, SoC, FPGA, ARM, AMBA.**

I. Introduction

Advances in silicon technology allow integrating complete systems on a single chip, introducing the Systems-on-a-chip (SoC) architecture concept.

SoC architectures were initially proposed as a central processor, a on-chip bus (OCB), standard components like memory, peripherals, interrupt units, etc. plus some application specific components. Among the advantages of this approach we can consider reduction of power consumption, higher integration density, lower systems costs, easier procurement, etc.

Among the different approaches to the SoC concept, Virtual socket initiative (VSIA) [1], merits special remark for the effect in standardizing criteria for concepts, methods and allowed interoperability.

Perhaps, the most popular SoC approach has been the ARM processor strategy [2]. ARM disposes of a complete family of evolving RICS microprocessors, with an OCB architecture based on the AMBA [3] bus.

AMBA (Advanced Microcontroller Bus Architecture) became one of the most widely used systems bus architectures for SoC applications (even for processors other than ARM) .SoC approach together with component reuse, through virtual components (or IPs) management, coming from different design teams (either in-house or third party), is greatly increasing the design productivity and therefore pushing the creation of new low-power high-performance applications.

This works starts from the SoC concept and tries to establish a design methodology for building SoC architectures embedding IPs, for bus-centric systems.

II. SoC Platforms

SoC concept requires support of different levels of abstraction in order to map system level specifications into chips, either ASICs, ASSPs, ASIPs o CPLDs...

A. SoC Architecture

Computer architecture has been for years constrained by manufacturability physical restriction (i.e. pin count of ICs and PCBs). This lets to structural techniques that characterized the architecture itself (i.e. 3-trate buffers to manage buses). Silicon integration evolution together with increasing number of metal layers let to architectures without theses restrictions, what lead to new criteria for designing systems (separation of address and data buses, format of control signals, clock and reset management, etc.) most of them included in the VSIA recommendations.

One of the consequences is that SoCs have multiplexed OCBs, so that every shared signal is routed from masters to slaves with help of multiplexers and decoders. These Bus Modules must be customized in order to take into account the number of master and slave modules, and their base addresses. Like in more complex computer systems, a SoC can be implement than one bus, being the typical architecture composed of one high-performance bus for memory access and high speed peripherals and one low-speed bus for low-speed peripherals like UARTs or PIOs.

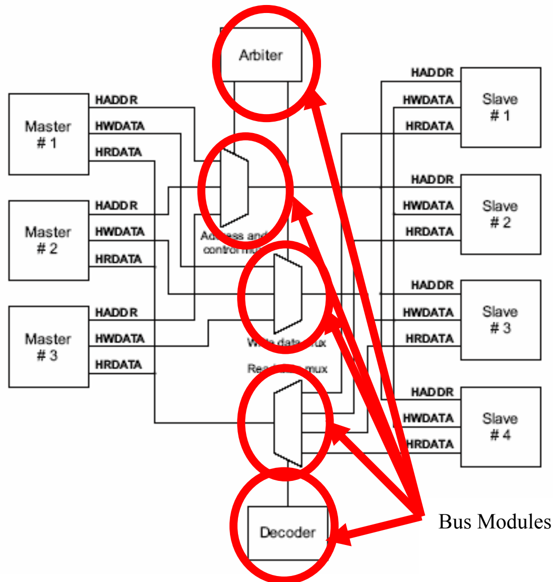

Fig. 1. Multiplexed OCB.

1) *AMBA based SoC*

The AMBA bus has been used widely in SoC development. Three distinct buses are defined within the AMBA specification:

1) Advanced High-performance Bus (AHB)
2) Advanced System Bus (ASB)
3) Advanced Peripheral Bus (APB)

AHB bus is devoted to high-performance communication, for system modules requiring high clock frequencies. This bus acts as the backbone bus. It is intended for connection of processors, on-chip memories and off-chip external memory interfaces.

The ASB is mainly deprecated and it has been substituted by AHB. The APB bus is devoted to low-speed peripherals and it is optimized to minimize power consumption and to reduce interface complexity. APB is designed to be used in conjunction with a system bus (AHB/ASB).

B. Processor IPs and OCBs

There are different approaches for building SoCs according to the application purposes. Our main interest is to help the system specification in order to quickly build any used defined architecture. This will help architectural exploration and verification either at simulation levels (at functional or structural level) or using prototyping platforms. Due to the fact that we are dealing with SoC design reaching prototyping platforms, we will validate our approach with examples mapping to ARM-FPGA solutions.

Among those solutions we analyzed several approaches:

1) *Altera SOPC [4]*

The Nios® embedded processor is a user-configurable, general-purpose RISC embedded processor. It was designed to be a flexible and powerful processor solution. Nios is a Soft-core (or Soft Ips), what means that RTL code is generated according to user specifications. Designer can choose data path width (16 or 32 bits), multiplier type (software based, one-cycle, n-cycles), cache size, memory map, interrupt priorities, etc.

Altera offers another SoC solution, based on the ARM processor. This system is not as flexible as Nios systems from the architectural point of view, but it is more powerful. ARM, Interrupt Controller and single and dual-port RAM are implemented as Hard-cores (or Hard IPs), so user cannot choose all options available on Nios systems.

For both systems Altera brings also complementary IPs (either proprietary or third party) ready to be used.

2) *Xilinx [5]*

MicroBlaze™ is the 32-bit soft processor from Xilinx. It is a standard RISC-based engine with a 32 register of 32 bits each, LUT RAM-based Register File, with separate instructions for data and memory access.

Smaller and less performing, PicoBlaze and its derivatives are fully customizable 8-bit soft microcontroller macros that provide 49 different 16- to 18-bit instructions, 8 to 32 general-purpose 8-bit registers, 256 directly and indirectly addressable ports, reset, and a maskable interrupt.

Xilinx disposes also of a solution with the IBM PowerPC 405 core integrated into Virtex-II Pro devices using the IP-Immersion architecture, which allows hard IP cores to be diffused at any location deep inside the FPGA fabric.

C. Prototyping Systems

To validate and test the entire system, we use a specific prototyping board. In this prototyping system, there is an ARM720T CPU connected to a Xilinx Virtex 2-6000 FPGA, SDRAM and FLASH and a set of ICs for physical interfaces to Ethernet, USB and PCMCIA/CF. Entire SoC is built inside FPGA, and any bus configuration is possible. This bus configuration flexibility corresponds with flexibility that our methodology and tools (UltraWizard) offers at design time.

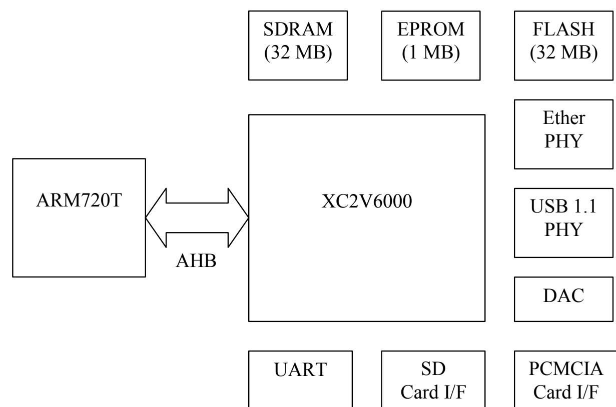

Fig. 2. ARM + FPGA Prototyping platform

This platform is configured, through a process in which ARM program memories are filled downloading code from PC to SDRAM through one UART running in special mode and, when then a reset sequence is generated and to allow all download-related peripherals running in normal mode of operation.

D. *Hierarchical Bus-centric SoCs*

As extension for traditional SoC architectures, hierarchical bus-based systems stand for systems with no limitations about number of buses and its hierarchy. These systems are more flexible and powerful than processor-centric systems (they allow any number of CPUs). To achieve that goal, we need to deal with any kind of bridges between buses (AHB-AHB, AHB-APB, APB-APB).

III. XML FOR ELECTRONIC CIRCUIT SPECIFICATION

During last years, XML has been proposed and progressively used for electronic design specification, becoming a de-facto standard for interoperability en IP management [6], [7], [8], [9].

A. *IP specification*

In order to describe an IP, it's necessary to specify it initially as a black-box, so without any functional or structural description, in our proposal, XML file describing components are divided in two parts, one for HW information and one for SW information. This means that the XML file corresponding to a given IP describes following items:

1) HW information :
   - Ports: name, direction (input, output or bidirectional), width and type (bus function, irq, or extern).
   - Bus type that IP is attached to (AHB or APB).
   - Behavior: Master or Slave.
   - IRQ related information: enabled or not, default assignment.
   - Address base and width.
   - Files needed to synthesize that IP.
2) SW information
   - Name of C structure.
   - Name of C header file.
   - Libraries directory.

B. *System specification*

To describe the entire SoC, we propose to use also an XML description. In this case, the XML file contains only buses, and references to IPs connected to them. At this level, designer can select special ports connection for some IP. For example, to allow connection of test signals to the bus in test stages that will later be disconnected for ASIC synthesis.

Bridging buses is supported, so it is necessary to instantiate the corresponding bridge in master bus and to indicate which one is the slave bus. Following this strategy, we can build any kind of bus hierarchy with no limitations in number of buses or bridges.

It is important to note that in XML, system specification contains only IPs and buses, avoiding explicit declaration of other necessary components of AMBA buses (Muxes, Arbiters, Decoder). These components are automatically generated by our system level tools (UltraWizard).

C. *Software Technology: JAVA*

Java technology includes both a programming language and a platform. Advantages of Java include mainly its portability, so using it as the programming language for our tools and methodology, will not determine the operating system it should run on.

Another advantage of Java is that it offers a whole set of facilities to parse XML documents giving a very simple interface to work with them.

IV. ULTRAWIZARD

An essential part of our methodology is composed of a set of tools that will build the correct system according to user specifications. We named this tool UltraWizard, and it is briefly explained in this chapter.

A. *EDA Tool Structure*

UltraWizard SW is composed of two pieces, front-end, to interact with the designer with a GUI, and back-end to generate the complete system. The GUI allows the designer to graphically build the entire system. Once done, front-end generates the XML file containing all the system information stored on it (system.xml). With this information, back-end tools generate all files needed for synthesis and simulation.

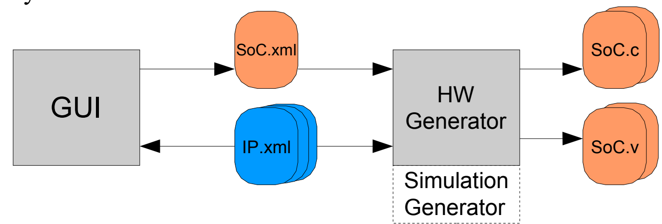

Fig. 3. UltraWizard Structure.

1) *Front-end*

With the GUI, designer describes the system to be implemented, selecting IPs from a catalog, and the way to connect them (assign IRQs, base address, etc).

As graphical interface with the designer, front-end UW manages the verification of correctness of the system description. (Unique IRQ, address range conflict ...).

2) *Back-end*

We call back-end UW, the code that generates the whole set of files starting from the information generated by front-end and designer. This information is archived in another XML file.

Files created at this stage to build the system specified by system XML file are:

1) HDL files (Verilog) to describe complete system. To achieve this goal, it necessary to create all bus interconnection modules for all buses present in the system, instantiate all modules and properly connect them.
2) C files to give a set of Software libraries in order to access registers of each module. At this time Back-end creates a single header file (system.h) that contains the definition of the set of pointers to each module or

structure that has registers visible by processor.

B. XML modeling

XML is a mark-up language for documents containing structured information [10]. Structured information contains both content (words, pictures, and in our case the more relevant characteristics of hardware IPs of our system) and some indication of what role those contents play (we use this indications to save the attributes we need for each characteristic of our Hardware IPs). Data can be identified using tags.

A mark-up language is a mechanism to identify structures in a document, whereas *XML* specification describes a standard way to add mark-up to documents.

XML seems to be a good choice because it is a very easy language to describe all kind of information and, because of its flexibility, allows building the right architecture to describe our hardware IPs and the top view of our SoC (it is a hierarchical language). These descriptions will be easily created, modified and parsed

All *XML* documents must follow certain rules in order to build *Well Formed* document. First of all, each document needs a root element (this element includes all information on other elements of the document), all labels have their closing tag, all elements should be correctly nested and all attribute values should go between quotation marks.

Also, all documents may accomplish all restrictions described in the document specification. This specification is known as *Document Type Definition* (*DTD*) that can be viewed as a dictionary for the document (also described in *XML*). It specifies the possible elements, attributes, entities, its properties and relations between them.

The first thing to do in order to build a design structure, is to write a *DTD* file describing the structure needed to save the information of the *hardware IPs* in our system, and also to describe the system itself.

The structure used for our systems is the following:

1) One *XML* file for each *hardware IP*
2) One XML file for the system
3) A DTD file defining all information on each module in the system
4) A DTD file defining all information on our system

C. Java Classes

SoCs build following our methodology and tools, will be composed of several interconnected buses and, at last in one of them, there will be a master module. The simplest example we can think about, is composed of one AHB with an ARM processor and a memory controller. We can make our systems grow in complexity to get a more complex bus structure, since our tool is being built trying to be as general and flexible as possible.

To represent the system we use the following classes:

1) *ModuleScanner*: this class reads and parses the XML files that describe stand-alone modules, and builds a class named *HWModule* containing all information needed to connect that module to the system.
2) *TopSysScanner*: this class reads and parses the XML file that describes the system structure (buses forming it and modules connected to each one of them). It returns a vector with all buses that the system is containing. To do it, several *AMBABus* classes are built containing all bus information (here we need to call *ModuleScanner* to get the *HWModule* classes).
3) *HWModule*: this class stores all information of each IP (HW module). To fill this information we need *ModuleScanner* class and its XML file. We can think in this XML file as a black box, so we only need to know know the module interface (inputs, outputs and some generic parameters).
4) *SharedHW*: this class is based on *HWModule* class, and specialized for modules that are simultaneously connected to more than one bus (modules shared by different buses). The only difference with *HWModule* class is that we added a special purpose vector, as class's attribute, to store information about which buses are sharing it. This is useful to store bridge information and to store modules that belong to more than one bus.
5) *AMBABus*: this class builds an *HWModule* vector for all buses in the system (we will be able to have spare vector for master and slave modules). This class also stores bus type (AHB or APB) and some other information needed to generate the system. Once we have an AMBABus vector we can generate all Verilog files for the system using *generate_from_template* class.
6) *Generate_from_template*: the main goal of this class is the automatic generation of code based in the use of templates and all the information contained in the classes structure.

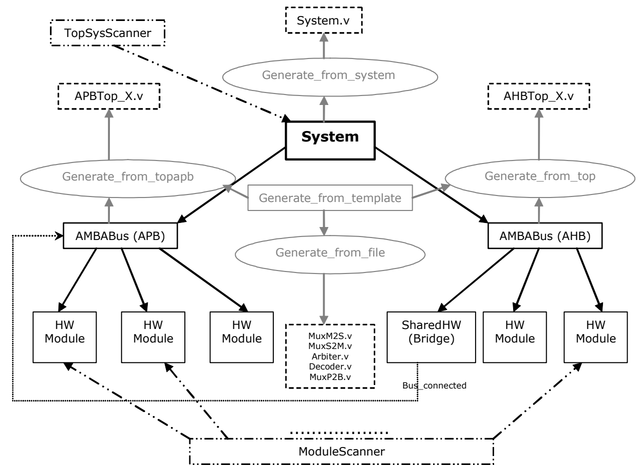

Fig. 4. Back-End class diagram.

For the automatic generation, we implemented the following methods:

1) *generate_from_file*: it uses templates with static structures and replaces all tags for the right information of each module (most of cases we will only need module's *name*). We use this method to generate Verilog code for multiplexers, decoders, arbiters… depending on the template received.
2) *generate_from_top*: it uses a template with a more rigid structure than the previous one, and generates different code depending on the tag found (explained later in this document). The main goal is to generate the Verilog

code for each AHB bus structure (modules attached to it, bridges and lines that are external to the bus). It builds a file for each AHB bus, known as *AHBTop_X*. It has two well differentiated parts: (1) inputs and outputs definition of the top (the external lines) and (2) the port-mapping between all ports of the modules with the internal lines of the bus (wires) and external lines for the top (inputs and outputs). In the second part, we may take special care for some modules that are always connected to an AHB bus (such as multiplexers between masters and slaves, the arbiter and the decoder).

3) *generate_from_topapb*: it is very similar to *generate_from_top* method, but in this case the main goal is to generate the Verilog code for each APB bus structure (modules attached to it and lines that are external to the bus). It builds a file for each APB bus, known as *APBTop_X*. In this case the template has a larger static part, because there are several modules that are always parts of an APB bus (such as the interrupt controller, slave to bridge multiplexer...).
4) *generate_from_system*: the main goal of this method is the automated Verilog code generation for the highest level in the SoC system. In this case we instantiate every bus using the *AHBTop_X* and *APBTop_X* modules, described in the files generated by the previous methods, we also interconnect them and describe the inputs and outputs for our system. We take special care with the port-mapping between each top, since we need to connect the bridge and the interrupt signals. We also use a template to do it.

D. Templates

Verilog files needed to synthesize the system are created using templates. Templates must follow the *AMBA* architecture specification again. The main goal for these templates is to be generic (they should work in any system that copes the restrictions of an *AMBA* system). This means enabling any number of buses (*AHB* or *APB*), any permitted interconnection between them and any number of modules attached to them. These templates also need to be created and oriented to build Verilog HDL descriptions (we may follow a certain structure and respect some syntax rules).

The main idea is create static Verilog sentences, with tags on the parts we want to be generated automatically. The tags are basically oriented to be substituted for the specific ports of each *IP* and to do correct *port-mapping* when we instantiate each *IP* in the top system.

V. EXAMPLE

In order to shown the powerfulness of our methodology, an example has been designed that allow its mapping into different architectural solutions that will be generated by UltraWizard. These variations will be later characterized from high level simulation up to rapid prototyping allowing a real architectural exploration of solutions.

A. Target Architecture

Test problem consists in a system that produces a sinusoidal waveform through a DAC with a FIFO interface to the SoC architecture. FIFO basically allows different frequencies at data producer and consumer sides. FIFO size can be selected by the designer. Samples are generated by SW code running on an ARM processor and then written to target device.

IP repository required for architectural exploration has a Memory Interface Controller (MIC), an AHB-APB Bridge, a DMA, the wrapper (containing the FIFO) to the DAC block, and all other wrappers to peripherals for the prototyping board.

We explore three different architectures considering that DAC is placed on the APB:

1) A single AHB + APB approach: ARM processor is continuously polling the DAC status. When ready, ARM calculates sample data and writes them directly to the DAC.
2) AHB + APB + IC: By adding an Interrupt Controller, CPU load is decreased, and samples will be sent when DAC is ready. With this configuration, HW resources are increased but bus occupancy decreases.
3) AHB + APB + DMA. Adding a Direct Memory Access (DMA) needs a buffer capability on DAC side. We implement a FIFO inside DAC. In this case, both DAC and DMA can interrupt processor, and pre-calculated and pre-buffered data is sent by DMA directly to DAC freeing ARM.

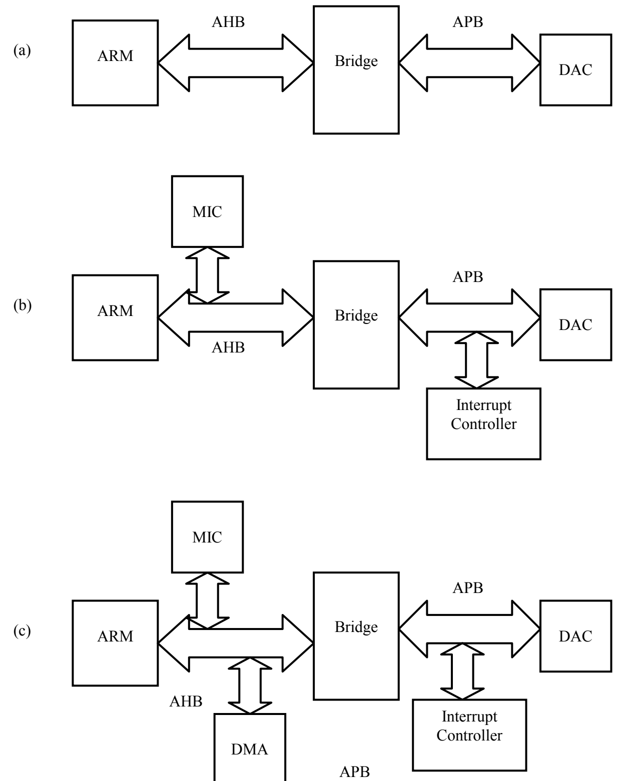

Fig. 5. (a) Simple design with ARM and DAC only. (b) Same design with DMA. (c) Using bridgeDMA.

B. *Synthesis and Simulation*

All different approaches have been generated by Ultrawizard. The obtained Verilog code was synthesized using Leonardo Spectrum, simulated with Verilog XL and later downloaded to our prototyping board.

Following table shows results.

ARCHITECTURE TYPE	HW RESOURCES (XILINX CLB)	BUS OCCUPATION (%)	PROCESSOR OCCUPATION (%)
Polling	1103	100%[1]	100%[1]
Interrupt	1341	7%	7%
DMA	2236	5%	3%

A quick evaluation of different architectural solutions can be carried out in a short period of time, using simulation or downloading into a prototyping platform.

Thus, the system architect can made performance measurements for different architectural choices in very short time.

VI. CONCLUSIONS

We believe complex SoC design will cope with a large diversity of architectures according to application domains and performance requirements. Application domains will drive the use of 3rd party IPs for which we dispose only of a restricted set of models. Performance requirements will let to complex architectural exploration analyses.

Ultrawizard is oriented to help designers deal with both aspects for AMBA systems. IP management is supported through XML de-facto standard specification. Detailed results for architectural exploration at any level will be accelerated by its capabilities for building complete systems with complex bus architectures.

REFERENCES

[1] VSI Alliance. http://www.vsia.com/
[2] ARM Limited. http://www.arm.com
[3] AMBAtm Specification (Rev 2.0) ARM Limited 1999. http://www.arm.com
[4] Altera SoPC Builder. http://www.altera.com/products/ip/certifications/sopc/ip-sopc.html
[5] Xilinx Embedded Development Kit, http://www.xilinx.com/ise/embedded/edk.htm
[6] Development System Reference Guide http://toolbox.xilinx.com/docsan/xilinx6/books/docs/dev/dev.pdf
[7] EDAXML DTD. http://www.e-tools.com/xml/dtds/EdaXML.dtd
[8] XML and Electronic Design Automation (EDA). http://xml.coverpages.org/xmlAndEDA.html
[9] An XML-based Meta-model for the Design of Multiprocessor Embedded Systems. W.O. Cesário, L. Gauthier, D. Lyonnard, G. Nicolescu and A.A. Jerraya.
[10] Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/xmlbase/.

[1] This percentage will depend on SW