Designing Systems-on-Chip Using Cores

Reinaldo A. Bergamaschi[1], William R. Lee[2]

[1]IBM T. J. Watson Research Center, Yorktown Heights, NY, [2]IBM Microelectronics, Raleigh, NC
rab@watson.ibm.com, brlee@us.ibm.com

Abstract

Leading-edge systems-on-chip (SoC) being designed today could reach 20 Million gates and 0.5 to 1 GHz operating frequency. In order to implement such systems, designers are increasingly relying on reuse of intellectual property (IP) blocks. Since IP blocks are pre-designed and pre-verified, the designer can concentrate on the complete system without having to worry about the correctness or performance of the individual components. That is the goal, in theory. In practice, assembling an SoC using IP blocks is still an error-prone, labor-intensive and time-consuming process. This paper discusses the main challenges in SoC designs using IP blocks and elaborates on the methodology and tools being put in place at IBM for addressing the problem. It explains IBM's SoC architecture and gives algorithmic details on the high-level tools being developed for SoC design.

1. Introduction

Today's market reality in VLSI design is characterized by: short time-to-market, large gate count and high-performance. Moreover, while time-to-market is paramount, complexity and performance cannot be compromised, at the risk of reaching the market with an uncompetitive product. This demanding environment is forcing fundamental changes in the way VLSI systems are designed. The use of pre-designed IP blocks (henceforth called *cores*) for SoC design has become essential in order to build the required complexity in a short time-to-market.

In practice, however, the vision of quickly assembling an SoC using cores has not yet become reality for various reasons, including the following:

- Architecting the system is a complex task, which requires designers to answer questions such as, (1) what cpu should be used, (2) what functions should be done in hardware or software, (3) what MIPS rate will the system achieve and is that enough for the target applications, etc. The answers to these questions lead to the cores to be used, however, in many cases they can only be confirmed later on in the design process.
- The integration of cores into an SoC is largely a manual and error-prone process because it requires designers to fully understand the functionality, interfaces and electrical characteristics of complex cores, such as microprocessors, memory controllers, bus arbiters, etc.
- Achieving full timing closure is very difficult due to the sheer complexity of the system. In many cases it requires cores to be tweaked which affects their reusability.
- Physical design of such large systems is a significant problem. Even if the layout of each core is predefined, putting them together with routing may cause unforeseen effects such as noise and coupled capacitances which degrade performance.
- System verification is one of the major bottlenecks. Even if the cores are pre-verified, it does not mean the whole system will work when they are put together. Various interface and timing issues can cause systems to fail even though the individual cores are correct. Current formal verification as well as software simulation techniques do not have the necessary capacity or speed to handle large systems in short run times.
- The lack of established standards industry-wide and/or the lack of efficient interface synthesis tools make it difficult for IPs from different providers to be integrated into the same SoC.
- Hardware-Software integration is another major problem which directly affects time-to-market because it is usually done later in the design process, when the hardware part is more stable.

Computer-aided-design tools have traditionally focused on low-level design issues, such as synthesis, timing, layout and simulation. More recently, modeling approaches using variations of high-level languages [2][5] have been developed. While they help with modeling and simulation speed-up, they are not directly targeted to systems using cores. The work being done by the VSI Alliance [9] is oriented towards facilitating integration of cores, however, it does not lessen the burden on the designer in understanding the complexities of the cores.

After the main architectural decisions have been made, the very first task in building an SoC is the integration of the cores into a top-level design, which can then be simulated, synthesized, floorplanned and used for early software development. This integration task today is largely a manual and error-prone process because it requires the designer to understand the functionality of hundreds of pins in various cores and determine which pins should be connected together. Moreover, cores are usually parameterized and need to be configured according to their use in the SoC. These tedious and manual tasks can insert errors in the design which may not be caught until much later in the process. This top-level design is the main driver for all follow-up tasks, hence it is important to be able to implement, configure and change it easily and efficiently.

There are almost no tools in industry today which help the designer to build an SoC by integrating and configuring cores easily. The complexity of current SoCs and the lack of appropriate high-level tools make the reuse and plug-and-play goal still unattainable.

This paper addresses these issues, and presents novel techniques

DAC 2000, Los Angeles, California

for SoC design using cores, which are very different from the normal ASIC design tools. These techniques and accompanying methodology have been implemented in a tool called “Coral”. Coral allows SoCs to be designed at a high-level abstraction called “virtual design” consisting of instantiations of virtual components, connected using virtual nets. This virtual design is much more concise and easier to create and configure than the real design. Coral also contains algorithms for mapping this virtual design onto a real design consisting of real cores from a library, interconnections and glue logic.

The paper is organized as follows: Section 2 describes the target architecture to be used and the main architectural issues involved in creating an SoC. Section 3 presents in detail the algorithms and techniques being developed for core-based SoCs. Section 4 presents a summary of the main contributions.

2. SoC Target Architecture

In the early stages of SoC design, cores were designed with many different interfaces and communication protocols. Integrating such cores in an SoC often required suboptimal glue logic to be inserted [6]. In order to avoid this problem, standards for on-chip bus structures were developed. Currently there are a few publicly available bus architectures from leading manufactures, such as the CoreConnect™[8] from IBM and the AMBA[1] from ARM. These bus architectures are usually tied to a processor architecture, such as the PowerPC or the ARM. The cores provided by these manufacturers are optimized to work with such bus architectures, thus requiring minimal extra interface logic.

IBM’s SoC framework consists of a core library called IBM Blue Logic™ Core Library[3], and a fixed bus architecture called the CoreConnect™ Architecture. The cores are predesigned and preverified to work with the CoreConnect bus architecture and protocols, thus allowing for reuse from chip to chip.

The IBM CoreConnect architecture provides three buses for interconnecting cores and custom logic [8]:

- Processor Local Bus (PLB): used for interconnecting high-performance, high-bandwidth cores, such as the PowerPC, DMA controllers and external memory interfaces.
- On-Chip Peripheral Bus (OPB): used for interconnecting peripherals which require lower data rates, such as serial ports, parallel ports, UARTs, and other low-bandwidth cores.
- Device Control Register Bus (DCR): low speed data-path used for passing configuration and status information between the processor core and other cores.

Fig. 1 illustrates a CoreConnect-based SoC. Although the cores are designed to interface with the buses almost directly, the designer still has to connect hundreds of pins and define the parameters for all cores. In order to create a correct top level description/schematic of the SoC a designer has to go through several steps, including the following:

- Define all the cores needed to implement the desired functionality. This process is a combination of identifying predesigned cores to be used either with or without modification and identifying new cores to be designed. These choices are typically made within the constraints of a given price/performance target. The designer must choose between 32, 64 or 128-bit buses, processor characteristics, hardware and software trade-offs, etc. In many cases, the choices can only be validated later on, after simulation and performance analysis.
- Understand the functionality of all pins on all cores and determine which pins should be connected together. Although this problem is alleviated with the use of predefined bus architectures, it is still a labor intensive manual process. It requires designers to read through lengthy documentation in order to understand the function of all pins in all cores. Even a single standard bus pin on just one core can cause weeks of schedule delays if it is named inconsistently with the specifications. The

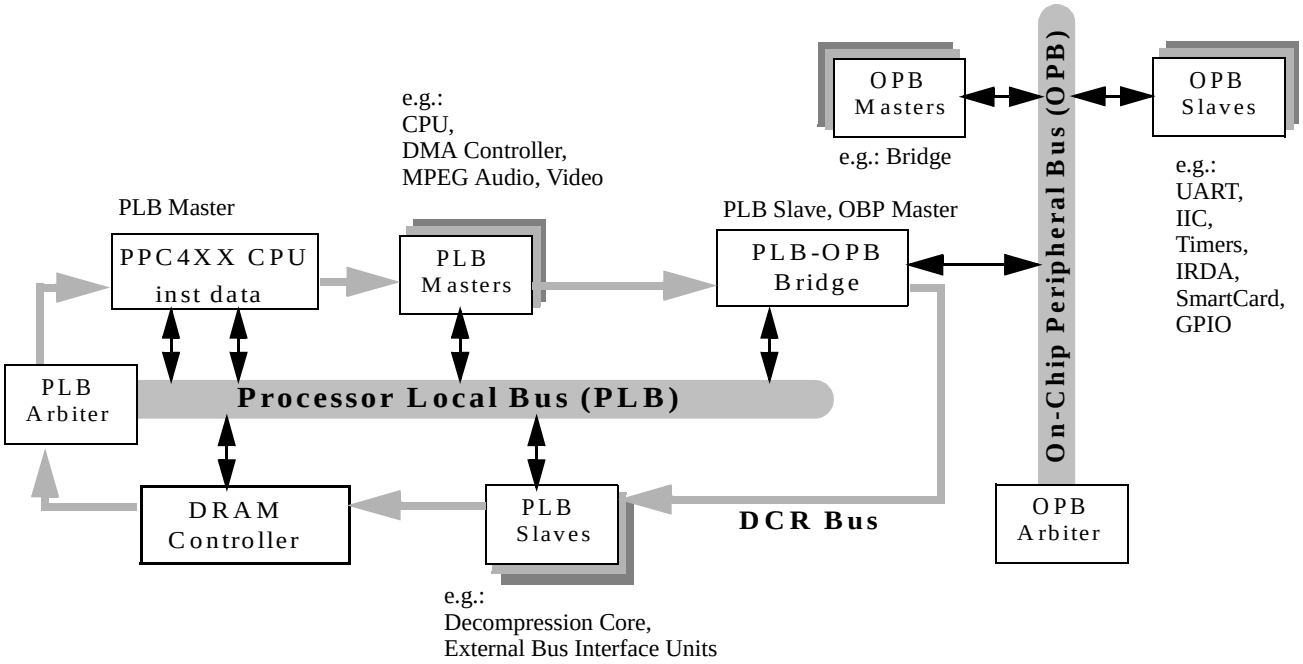


Figure 1. System-on-chip using the CoreConnect bus architecture

SoC integrator will either call the core support group for information, or worse, incorrectly connect this pin and find this problem only later during simulation and debugging.

- Define the request priorities for the masters on the buses and the processor interrupt request priorities. Priorities are application specific and may dramatically affect the system performance. Designers must analyze the application and determine the priorities among devices. On the master side, for example, in a set-top box chip, the video update from the MPEG decoder may be assigned higher priority than the processor itself. Similarly, for the slave interrupts, the designer must choose their relative priorities for the software to take advantage of the priority encoded interrupt vector generation.
- Interconnect pins according to their priorities, while possibly leaving room in the design for last minute changes (e.g., it may be decided at a later stage that interrupt pins should change priority, or a new interrupt be added).
- Define which cores may access memory through a DMA controller and perform the channel assignment according to the priority of the requesting devices. When the number of DMA requestors exceeds the number of DMA channels, channel sharing can be utilized and/or adding an additional DMA controller.
- Define address maps for all cores and pass the values as parameters to each core, insuring that an address conflict is not created between any two cores.
- Define the clock domains valid in the chip and connect the right clocks to each core, as well as the appropriate clock control logic.
- Insert any required glue logic between cores
- Define all the chip I/Os and design the I/O logic including any sharing of pins and manufacturer required test control logic.
- Check that the cores being used are compatible with respect to operating frequency, bit-width, version number, etc.
- Document the system (e.g., address maps, interrupt priorities, DMA channels, chip I/Os, etc.) for future use by software and printed circuit board developers.

This non-exhaustive list is sufficient to show that, despite using a fixed bus architecture, there still is a large number of complex tasks that need to be performed. These tasks are performed manually in today's methodologies and tools, which is inefficient and error-prone.

3. Automating SoC Integration

In order to automate many of the manual tasks described in the previous section, a new tool called "Coral" was developed, which contains new algorithms and methodologies for SoC design using cores based on the concept of a synthesizable "virtual design".

Coral increases productivity by raising the level of abstraction in which SoC designs are performed. By enabling the designer to work at the virtual level, it hides all the unnecessary complexity associated with the cores, which decreases errors and increases productivity.

Coral and its associated methodology are based on the following elements: (A) Virtual design, (B) Interface encapsulation and Glueless interfaces, (C) Core and Pin Properties, (D) Interconnection Engine, (E) Virtual to Real Synthesis Engine, and (F) Configuration Engines. Details on these elements are given in the following sections.

3.1 Virtual Design

In the traditional ASIC design flow, there is a high-level of abstraction represented by the register-transfer level (RTL) language description, using hardware description languages such as VHDL or Verilog. Most designs are written at the RT level and mapped to a gate-level netlist by logic synthesis tools. In current SoC design flows, there is no similar high-level abstraction. SoC designs are described directly at the core level (similar to gate-level) by manually instantiating the cores and the interconnections among their pins. In other words, core-based SoC design today is at a similar stage that ASIC design was prior to the widespread use of hardware description languages and logic synthesis tools.

Coral's synthesizable virtual design concept changes this picture. The virtual design is a structural and functional encapsulation of the real design consisting of virtual components, virtual interfaces and virtual nets. The virtual design can be created using a schematic editor or any hardware description language.

A virtual component is a representation of a class of real components. For example, the PowerPC virtual component (PPC_VC) represents all real PowerPC cores (e.g., 401, 405). For the same virtual design, the user can at any moment select a different real component mapping for a virtual component and Coral will automatically regenerate the necessary interconnections and glue logic to use the new real component.

The inputs/outputs of a virtual component are called virtual interfaces. Virtual interfaces are connected using virtual nets. A virtual interface represents a grouping of the real interface pins which are functionally related. For example, the PowerPC virtual component contains a single virtual pin PLB_M_DCU_interface representing all real pins (in the real PowerPC cores) which are responsible for the interface between the internal data cache unit and the master side of the external processor bus.

Because the number of virtual ports ranges between 4 to 15 for a virtual component vs. 50 to 300 for a real component, the task of creating a virtual design is much simplified and roughly equivalent to drawing the system block diagram for the SoC. The virtual design represents both a synthesizable description of the SoC as well as the documentation describing the function of the SoC.

In order to illustrate the degree of encapsulation of a virtual component, let us consider the PowerPC virtual component (PPC_VC) and the real PowerPC 401 (PowerPC401). The VHDL component declaration for PPC_VC, shown in Fig. 2, has 10 pins, or virtual interfaces, whereas the real component PPC401 has approximately 160 pins. The 10 virtual pins describe functional interfaces such as PLB_M_DCU_interface (master data cache unit

```
ENTITY PPC_VC IS
PORT (    PLB_M_ICU_interface:      in STD_LOGIC;
          PLB_M_DCU_interface:      in STD_LOGIC;
          ISOCM_interface:          in STD_LOGIC;
          DSOCM_interface:          in STD_LOGIC;
          APU_interface:            in STD_LOGIC;
          RESET_interface:          in STD_LOGIC;
          INTERRUPT_interface:      in STD_LOGIC;
          CLOCK_interface:          in STD_LOGIC;
          DCR_interface:            in STD_LOGIC;
          JTAG_interface:           in STD_LOGIC
);
END PPC_VC;
```

Figure 2. PowerPC Virtual Component interface definition

bus interface), or INTERRUPT_interface, or APU_interface (Auxiliary processor unit interface), as well as other required interfaces for clocking and testing. Each virtual interface may correspond to several real pins. For example the PLB_M_DCU_interface virtual pin corresponds to 18 real pins (including inputs and outputs).

3.2 From Interface Encapsulation to Glueless Interfaces

One of the primary concepts in Coral is that the system designer never has to worry about or create any interface logic between cores.

The use of a target bus architecture and cores predesigned to interface to the bus eliminates the need for protocol synthesis and reduces considerably the amount of interface logic. However, it does not eliminate the glue logic completely, as it is sometimes dependent on the complete system. For example, all "acknowledgment" signals from the slave devices are OR'ed and the output connected to the bus arbiter. The number of inputs to this OR gate depends on the number of slave devices.

In order to allow for fully automatic synthesis of a virtual design into a real design, Coral relies two levels of glue logic encapsulation. First, each core is designed to contain all the static and parameterizable protocol/interface logic. This can be done by means of generics ports in VHDL or parameters in Verilog. Secondly, Coral is able to create automatically a limited amount of glue logic between cores. This logic is described as a property of input pins. Such property can describe simple Boolean functions that represent the glue logic from all source nets to a given input pin. By means of these two levels, the designer is completely spared from having to create any interface logic explicitly.

For legacy cores and third party cores which were not originally designed to contain the interface logic, one can create *core wrappers* in VHDL or Verilog which contain the necessary parameterizable logic to interface the cores to the adopted bus architecture.

3.3 Core and Pin Properties

In order to automatically generate interconnections among cores, it is necessary to encode the structural and functional characteristics of a component and its pins, in a manner that can be algorithmically processed by a computer program. In current design methodologies, the designer has to spend a large amount of time reading and understanding specification manuals just to find out how pins in different components need to be connected.

In Coral, this information is encoded into *properties* attached to all components and their pins. Coral contains algorithms which can efficiently compare these properties and decide whether two pins should be connected. Properties associated with a pin define the functionality and taxonomy of that pin. By assigning unique properties to all pins in all cores, it is possible to compare those properties and determine if the pins are compatible.

The approach in [7] also mentions classifying IPs using properties. However, it differs completely from our work first because it only applies to IP properties (there is no mention of pin properties), and secondly because its goal was to be able to query a database for IP blocks satisfying a set of properties, whereas in our work properties are used in a much broader sense to help in the automatic synthesis of SoCs. Moreover, the approach in [7] does not give any algorithm for searching and reasoning about the properties, which is an integral part of Coral (see Section 3.4).

Based on the IBM Blue Logic™ Core Library utilizing the CoreConnect™ bus architecture and other external cores, it was determined that most pins can be classified for interconnection purposes according to the following functional and structural properties:

- BUS_TYPE: the type of bus that the pin interfaces to. This can assume values such as, PLB (processor local bus), OPB (on-chip peripheral bus), ASB (AMBA system bus), APB (AMBA peripheral bus), etc.
- INTERFACE TYPE: the type of interface represented by the pin, e.g., MASTER, SLAVE.
- FUNCTION_TYPE: the function implemented by the pin, e.g., READ, WRITE, INTERRUPT. This pin could be one of several pins responsible for implementing the function.
- OPERATION_TYPE: the operation performed by the pin as part of the function specified in FUNCTION_TYPE, e.g., REQUEST, ACKNOWLEDGE.
- DATA_TYPE: the type of data manipulated by the function, e.g., ADDRESS, INSTRUCTION, DATA.
- RESOURCE_TYPE: the system resource used when the function specified by FUNCTION_TYPE is executed, e.g., BUS, PERIPHERAL.
- PIN_GROUP: property used to indicate grouping of pins in the same interface.

For example, pin DCU_plbRequest on the PowerPC™401 is asserted by the Data Cache Unit (DCU) inside the PowerPC, to request a data read or write between external cacheable memory and the general purpose registers in the execution unit across the read or write data bus. The PowerPC acts as a master device on the processor local bus (PLB). Given this information, we derived the following properties for this pin:

- BUS_TYPE = PLB
- INTERFACE_TYPE = MASTER
- FUNCTION_TYPE = READ_OR_WRITE
- OPERATION_TYPE = REQUEST
- DATA_TYPE = DATA
- RESOURCE_TYPE = BUS
- PIN_GROUP = DCU

Coral uses a specialized language for specifying properties on cores and pins. For any given core to be usable by Coral, it needs to have a corresponding virtual component and properties associated with all its pins. Once that is available, that core can be used by Coral and automatically connected to other cores. This approach makes Coral the one of the first tools for plug-and-play use and reuse of cores in any architecture.

3.4 Interconnection Engine

Properties are used for establishing correspondence between a virtual pin and the real pins with similar functionality, as well as for matching up real pins in different components. By comparing properties on pins the tool can decide whether the functionality of a real pin falls within the functionality of a virtual pin. In addition, by comparing properties on real pins from different components, Coral can decide whether they should be connected.

Since the complete SoC may have hundreds to thousands of internal pins, these comparisons need to be done very efficiently and in a general manner. Moreover, the algorithms needs to be able to handle not only exact matches but also overlapping sets (not exact match). This is achieved by means of two novel techniques:

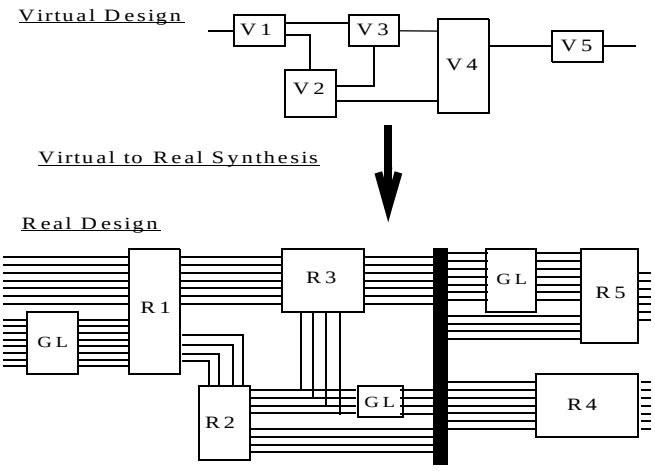


Figure 3. Virtual to real synthesis (GL = glue logic)

(1) property encoding using Binary Decision Diagrams[4] (BDDs), and (2) property comparison and matching using logical operations on BDDs.

Each property/value pair PV_i = <property_type Ti == property_value Pi> is mapped to a Boolean variable (e.g., a BDD variable), and a group of property/value pairs is mapped to the AND function of all individual Boolean variables. More specifically, given a group of property/value pairs PG = {PV_1, PV_2,..., PV_n}, the corresponding set of BDD variables is denoted B(PG) = {b_1, b_2,..., b_n}. The BDD for the complete group is given by: $F(PG) = b_1 \wedge b_2 \wedge \ldots \wedge b_n$. When the property group PG is attached to a pin T, the complete BDD function *F(PG)* is denoted as $F(T)|_{PG}$, or the property function F of pin T with respect to property group PG.

The virtual to real synthesis process requires two steps of property comparisons. First, given a virtual pin V in a virtual component VC, the tool needs to determine the compatible set of real pins in the corresponding real component, with respect to their interconnection property group. This is achieved using the following algorithm. Let $F(V)|_{PG}$ and $F(R)|_{PG}$ be the Boolean functions representing the interconnection property groups in virtual pin V and real pin R respectively, where V belongs to virtual component VC and R belongs to real component VR (which is a valid mapping of VC). R is compatible with V *iff* $F(V)|_{PG} \supseteq F(R)|_{PG}$, that is, the property function for V contains the one for R. The containment operator "$\supseteq$" is computed using BDD operations. In logic terms, A contains B if $A \vee B \equiv A$.

Secondly, given two real pins in two different components, the tool needs to determine if they are compatible and can be connected together. This is computed as follows. Let $F(Ra)|_{PG}$ and $F(Rx)|_{PG}$ be the Boolean functions representing the interconnection property groups in real pins Ra and Rx respectively, in two real components. Ra is compatible with Rx *iff* $F(Ra)|_{PG} \supseteq F(Rx)|_{PG}$ or $F(Rx)|_{PG} \supseteq F(Ra)|_{PG}$, that is one must be fully contained in the other. This again can be computed efficiently using BDD operations.

3.5 Virtual to Real Synthesis

The virtual-to-real synthesis engine (VRSE) is responsible for synthesizing a real design from a virtual design. Fig. 3 illustrates this synthesis process. The expansion of virtual interfaces and virtual nets into real interfaces and real nets relies on *properties* attached to both virtual and real components and virtual and real pins, and on the algorithms presented in Section 3.4.The VRSE is comprised of the following steps:

1. For each virtual component, the VRSE instantiates a real component in the real design. The exact real component is fully selectable by the designer based on a given virtual component library which lists all available real components for a given virtual component.
2. The VRSE traverses every virtual net and the virtual pins connected to it. For each virtual pin visited, it determines the corresponding real pins that have compatible functionality (using the property comparison algorithms explained in the previous sections). This is illustrated in the first comparison step shown in Fig. 4. Given a virtual pin *v1* in virtual component VC1, the VRSE compare the properties of *v1* with those of all real pins in real component *RC1*. From this comparison it can establish that real pins *r1*, *r2* and *r4* are compatible with virtual pin *v1*. Similarly, it can derive that, for example, real pins *r5*, *r7* and *r8* in real component *RC2* are compatible with virtual pin *v3* in virtual component *VC2*.
3. Given two groups of real pins in two real components (corresponding to two virtual pins connected by a virtual net), the VRSE compares the properties on the real pins (across these two groups) and determine which real pins should be connected together. This is the second comparison step shown in Fig. 4. The VRSE automatically determines the direction of the real ports (in, out, inout) and connects them correctly.

Fig. 4 shows simple cases of straight connections, however, the algorithm supports the specification of simple logic functions associated with real pins, and when the connection is made those logic functions are created as well. This is used for automatically inserting any required glue logic between components.

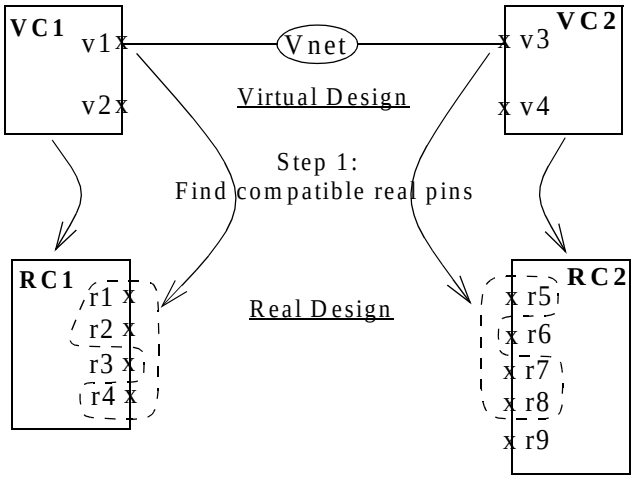
3.6 Configuration Engine

Coral provides the designer various system configuration menus which define much of the overall SoC operation. These menus permit programming of the virtual component parameters in an error free method and enable the generation of the system documentation that is not found as part of a stand-alone core specification. The configuration menus include clocking, address map definition, interrupt map definition, DMA channel assignment and the I/O specification and generation. The configuration information becomes part of the virtual design, and passed to the real design as parameters to the cores during virtual to real synthesis.

As an example of such configuration capability, consider the configuration of address maps. Each core in the design needs to be associated with a given address map in its own bus domain. IBM's CoreConnect™ SoCs have three address domains: DCR (device control register), PLB (processor local bus) and OPB (on-chip peripheral bus). Each domain is configured through the Coral user interface to generate all the address component parameterization. This process defines all register and memory address space, which can then be used to generate the documentation and keep the design and documentation consistent.

4. Summary

This paper presented the main issues involved in designing an SoC using cores and an approach for automating the design and synthesis of the top-level description of the system.



Step2: Compare properties or {r1, r2, r4} with {r5, r7, r8}. Assuming that {r1, r8}, {r2, r7} and {r4, r5} are compatible, Coral will produce the connections shown below.

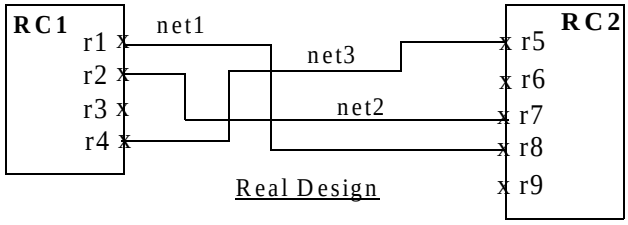


Figure 4. Steps during virtual to real synthesis

The algorithms and methodology presented in this paper have been implemented in C++ and tested using the IBM Blue Logic Core Library and the CoreConnect bus architecture. The approach is general and can be extended to any target bus architecture.

The main characteristics of Coral are: (1) a unique encapsulation of the structural and functional information of the cores in virtual representations and properties, (2) a synthesizable virtual design representation which is a high-level abstraction of the SoC, (3) Core encapsulation and glueless interfaces which free the designer from having to create any interface logic, (4) algorithms for mapping a virtual design into a real design with all interconnections and glue logic, and (5) special configuration menus which allow the designer to specify parameters to the SoC at the virtual design level.

As proof of concept, several virtual designs have been created and automatically synthesized to real designs. Significant reductions in design size and time have been accomplished. For example, a reference SoC design composed of 41 instantiated cores and macros was reduced from 6900 lines of Verilog for the real design to around 600 lines for the virtual design.

Coral effectively helps designers to automate most of the manual and error-prone tasks involved with designing a top-level SoC using cores. Moreover, by encapsulating cores with virtual descriptions and properties, it brings a high-level of abstraction to SoC design which allows for easy reuse of predesigned components.

Coral represents one of the first synthesis tools in industry that can effectively realize the promise of plug-and-play of cores.

ACKNOWLEDGMENTS

We would like to thank the members of the Coral team, namely, Subhrajit Bhattacharya, Foster White, Mike Muhlada, Duane Richardson and Ronaldo Wagner for many interesting discussions and suggestions which helped shape this paper.

REFERENCES

[1] “AMBA Specification Overview”, ARM, http://www.arm.com/Pro+Peripherals/AMBA.

[2] G. Arnout, “SystemC Standard”, Proceedings of the ASP-DAC 2000, January 2000.

[3] “Blue Logic Technology”, IBM, http://www.chips.ibm.com/bluelogic.

[4] R.E. Bryant, “Graph Based Algorithms for Boolean Function Manipulation”, IEEE Transactions on Computers, Vol.35, No.8, August, 1986.

[5] P. Flake and S. Davidmann, “Superlog, a Unified Design Language for System-on-Chip”, Proceedings of the ASP-DAC 2000, January 2000.

[6] A. Rincon, W. Lee and M. Slatery, “The Changing Landscape of System-on-a-Chip Design”, IBM MicroNews, 3rd Quarter 1999, Vol.5, No.3, IBM Microelectronics.

[7] P. Schindler, K. Weidenbacher and T. Zimmermann, “IP Repository, A Web based IP Reuse Infrastructure”, Proceedings of IEEE 1999 Custom Integrated Circuits Conference, May 1999.

[8] “The CoreConnect™ Bus Architecture” IBM, 1999, http://www.chips.ibm.com/product/coreconnect/docs/crcon_wp.pdf

[9] VSI Alliance™ Architecture Document, Version 1.0, VSI Alliance, 1997, http://www.vsi.com/the_rest.html.