RC21951 (98679) 6 February 2001 **Computer Science**

IBM Research Report

Automating the Design of Systems-on-Chip Using Cores

Reinaldo A. Bergamaschi[1], Subhrajit Bhattacharya[1], Ronaldo Wagner[+], Colleen Fellenz[+], William R. Lee[++], Foster White[**], Michael Muhlada[++], Jean-marc Daveau[*]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[+]IBM EDA Lab
Fishkill, NY

**Nortel Networks
Raleigh, NC

*ST. Microelectronics
Grenoble, France

[++]Cisco Sytems
Raleigh, NC

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

Automating the Design of Systems-on-Chip Using Cores

Reinaldo A. Bergamaschi, Subhrajit Bhattacharya,
Ronaldo Wagner, Colleen Fellenz,
William R. Lee, Foster White,
Michael Muhlada, Jean-Marc Daveau

IBM Corporation[1]

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 predesigned and preverified, 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 researched 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 predesigned 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 preverified, it does not mean the whole

1. R. Bergamaschi and S. Bhattacharya are with IBM T. J. Watson Research Center, Yorktown Heights, NY,
R. Wagner and C. Fellenz are with IBM EDA Lab., Fishkill, NY,
W. Lee and Muhlada were with IBM Microelectronics, Raleigh, NC, and are now with Cisco Systems, Raleigh, NC,
F. White was with IBM Microelectornics, Raleigh, NC, and is now with Nortel Networks, Raleigh, NC,
J-M. Daveau was with IBM T. J. Watson Research Center, Yorktown Heights, NY, and is now with ST Microelectronics, Grenoble, France.

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. The Micon system [3] was the first attempt to automatically interconnect discrete components on a system board. It used an expert system to guide the interconnections based on rules for each component. Although it was meant for boards, the problem bears strong similarities to today's core-based designs. More recently, modeling approaches using variations of high-level languages [2][6] 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 [10] 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 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 [7]. 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™[9] 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[4], 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 [9]:

- 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, if interconnected wrongly, can cause weeks of schedule delays, since the error may be found 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.

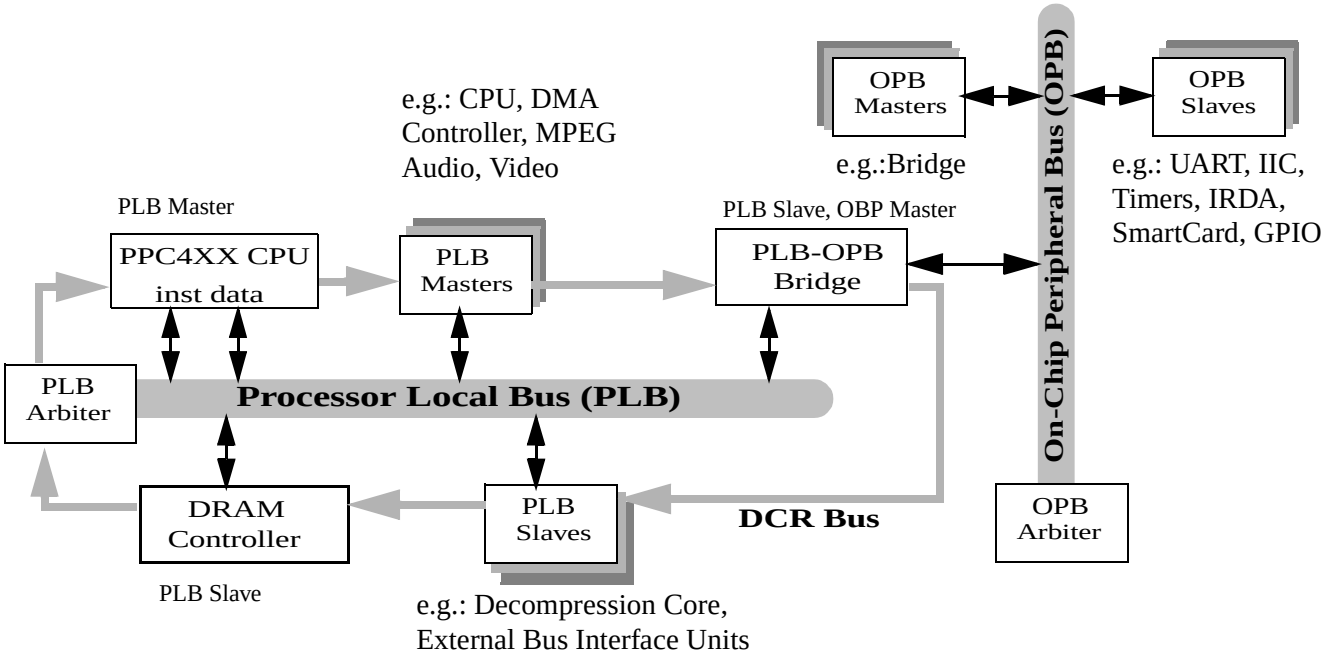


Figure 1. System-on-chip using the CoreConnect bus architecture

- 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, ensuring that an address conflict is not created between any two cores. When bus bridges are present, the checks for address conflicts should also take into account the address ranges of slave devices accessed through the bridge.
- 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) Configuration Engine, and (F) Virtual to Real Synthesis Engine. 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.

```
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

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. Note that a virtual interface is more powerful than just grouping a set of real pins in a complex type such as a record type. First of all, in a record type all fields (or bits) need to have the same direction (all inputs or outputs or bidis), and secondly when connecting two interfaces of the same record type, all fields of the record are connected, which is not the case when connecting two virtual interfaces.

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 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 on 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.

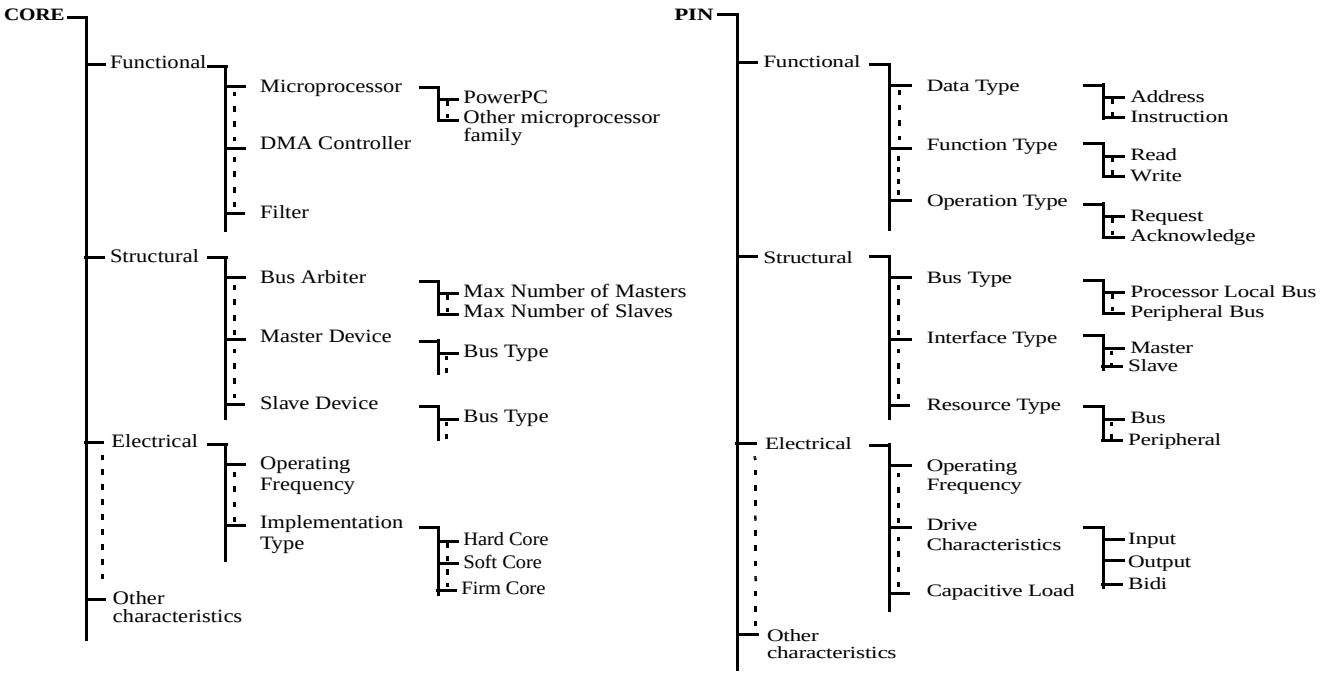


Figure 3. Example of classification tree for cores and pins

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, functional and electrical 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.

Based on our analysis of hundreds of cores and pins we have identified several characteristics which can be used for their classification. Fig. 3 illustrates an example of such classification tree for cores and pins.

Properties are also grouped according to specific purposes. Based on the IBM Blue Logic™ Core Library utilizing the CoreConnect™ bus architecture and other external cores, we have defined a number of property groups which encapsulate all the information required for interconnecting pins, or for generating interface logic, to name a few.

Our analysis of several pins and cores revealed that most pins can be classified for interconnection purposes according to a limited number of functional and structural characteristics, which were grouped into a pin property group called INTERFACE_DEF. The properties in this group and their meaning are:

- 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^{TM}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.

The approach in [8] 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 [8] does not give any algorithm for searching and reasoning about the properties, which is an integral part of Coral (see Section 3.4).

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: property encoding using Boolean functions, and property comparison and matching using logical operations on Boolean functions.

3.4.1. Property Encoding as Boolean Functions Using Binary Decision Diagrams

All core and pin properties used in Coral are encoded as Boolean functions represented as Binary Decision Diagrams [5] (BDDs). Boolean functions were chosen because they can be easily manipulated and compared, and the use of BDDs provide an efficient data-structure for their manipulation. An important characteristic of BDDs is

that they are canonical representations. That is, given an ordered set of Boolean variables and two different Boolean expressions representing the same Boolean function, the BDDs for these Boolean expressions will be exactly the same.

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 Boolean variables is denoted $B(PG) = \{b_1, b_2, ..., b_n\}$. The Boolean function 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 Boolean function *F(PG)* is denoted as $F(T)|_{PG}$, or the property function *F* of pin *T* with respect to property group *PG*.

Given the canonical qualities of BDDs, if two property groups (e.g., in two pins) contain the same property/value pairs, their BDDs will be exactly the same, even if the orders of the property/value pairs in the both groups differ. This implies that in order to check if two pins have exactly the same properties, it is sufficient to build the BDDs for the properties in each pin and check if the two BDDs are the same.

DCU_plbRequest		***M0_Request***		
Property Type	***Property Value***	***Property Type***	***Property Value***	***Boolean Variable***
BUS_TYPE	***PLB***	***BUS_TYPE***	***PLB***	***A***
INTERFACE_TYPE	***MASTER***	***INTERFACE_TYPE***	***MASTER***	***B***
FUNCTION_TYPE	***READ_OR_WRITE***			***C***
OPERATION_TYPE	***REQUEST***	***OPERATION_TYPE***	***REQUEST***	***D***
DATA_TYPE	***DATA***			***E***
RESOURCE_TYPE	***BUS***	***RESOURCE_TYPE***	***BUS***	***F***
F(DCU_plbRequest) = A.B.C.D.E.F		***F(M0_Request) = A.B.D.F***		

Figure 4. Example of pin properties and their corresponding Boolean Functions

Fig. 4 shows the interconnection properties and the corresponding Boolean functions for two pins, namely *DCU_plbRequest* in the PowerPC core and *M0_Request* in the PLB Arbiter core. Based on the datasheets, these pins will typically be connected together. The property functions on these pins are not exactly the same, but they are *compatible*. This illustrates the fact that pins may be connected together even though their properties do not match exactly.

The Interconnection Engine contains specialized algorithms for comparing properties and deciding whether two pins should be connected, namely:

1. Property Compatibility Check

   Given two pins and a property group, this check decides whether the pins are compatible with respect to the property group. For this check to return true it is not necessary for the property/value pairs to match exactly; it is sufficient for them to contain each other. More formally this can be stated as follows. Let *S* and *T* be two pins in different cores and let $F(S)|_{PG}$ and $F(T)|_{PG}$ be the corresponding property functions for pins *S* and *T* with respect to the same property group *PG*. Pin *S* is compatible with pin *T* if and only if $F(S)|_{PG} \supseteq F(T)|_{PG}$ or $F(T)|_{PG} \supseteq F(S)|_{PG}$, that is one must be fully contained in the other. The containment operator " $\supseteq$ " is computed using BDD operations. In logic terms, *F* contains *G* if $F \vee G \equiv F$.

   Using this algorithm it can be shown that pins *DCU_plbRequest* and *M0_Request* are compatible with respect to the interconnection properties shown in Fig. 4 because *F(M0_Request) = A.B.D.F* contains *F(DCU_plbRequest) = A.B.C.D.E.F*, hence they can be connected together.

2. Property Matching Check

   For certain types of properties, it is important to determine if the property/value pairs in two pins are exactly the same. This is a more strict check than compatibility and it can be used for determining whether two pins have the same electrical properties. For example, when comparing the operating frequencies of two pins, it is

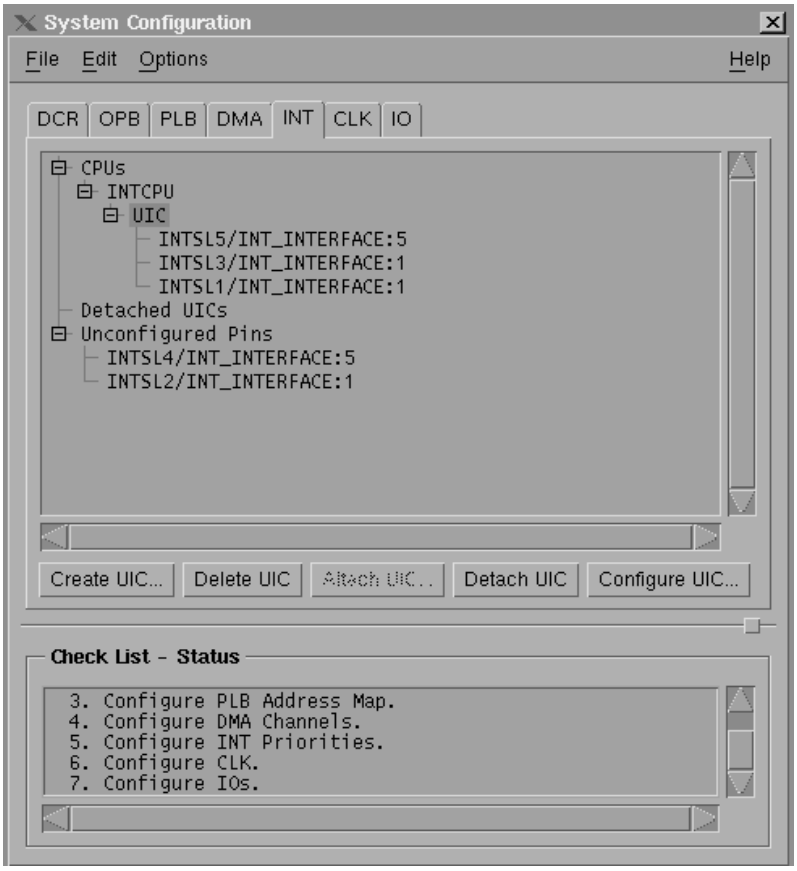


Figure 5. Main system configuration panel, showing interrrupt configuration panel in detail

necessary to check for an exact match.

Due to the canonical properties of BDDs, two Boolean expressions representing the same Boolean function will be mapped to the same BDD. Hence, in order to check the if two property functions for two pins are the same, it is sufficient to check if their BDD representations are the same. This can be done simply by performing an equality check on the BDD pointers.

More formally this check can be stated as follows. Let *S* and *T* be two pins in different cores and let $F(S)|_{PG}$ and $F(T)|_{PG}$ be the corresponding property functions for pins *S* and *T* with respect to the same property group *PG*. Pin *S* matches pin *T* with respect to property group *PG* if and only if $F(S)|_{PG} \equiv F(T)|_{PG}$.

3.5 Configuration Engine

Coral provides the designer various system configuration menus which define much of the overall SoC operation. These menus permit programming of the real 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: bus and address map definition for DCR, OPB and PLB busses, DMA channel assignment, interrupt map definition, clock and I/O specification and generation. The main system configuration graphical user interface is shown in Fig. 5. The top of this figure shows the tabs for the different configuration folders. The interrupt configuration folder is selected and its components and interfaces are shown in the center of the figure. The bottom of this figure shows a configuration checklist - each item receives a check mark as the corresponding configuration is completed by the designer.

An important aspect of the configuration engine is that it works on the virtual design. That means that even for specifying low level information such as address maps, priorities, etc., the designer is shielded from the complexities of the real design. Moreover, the configuration engine helps build the virtual design itself.

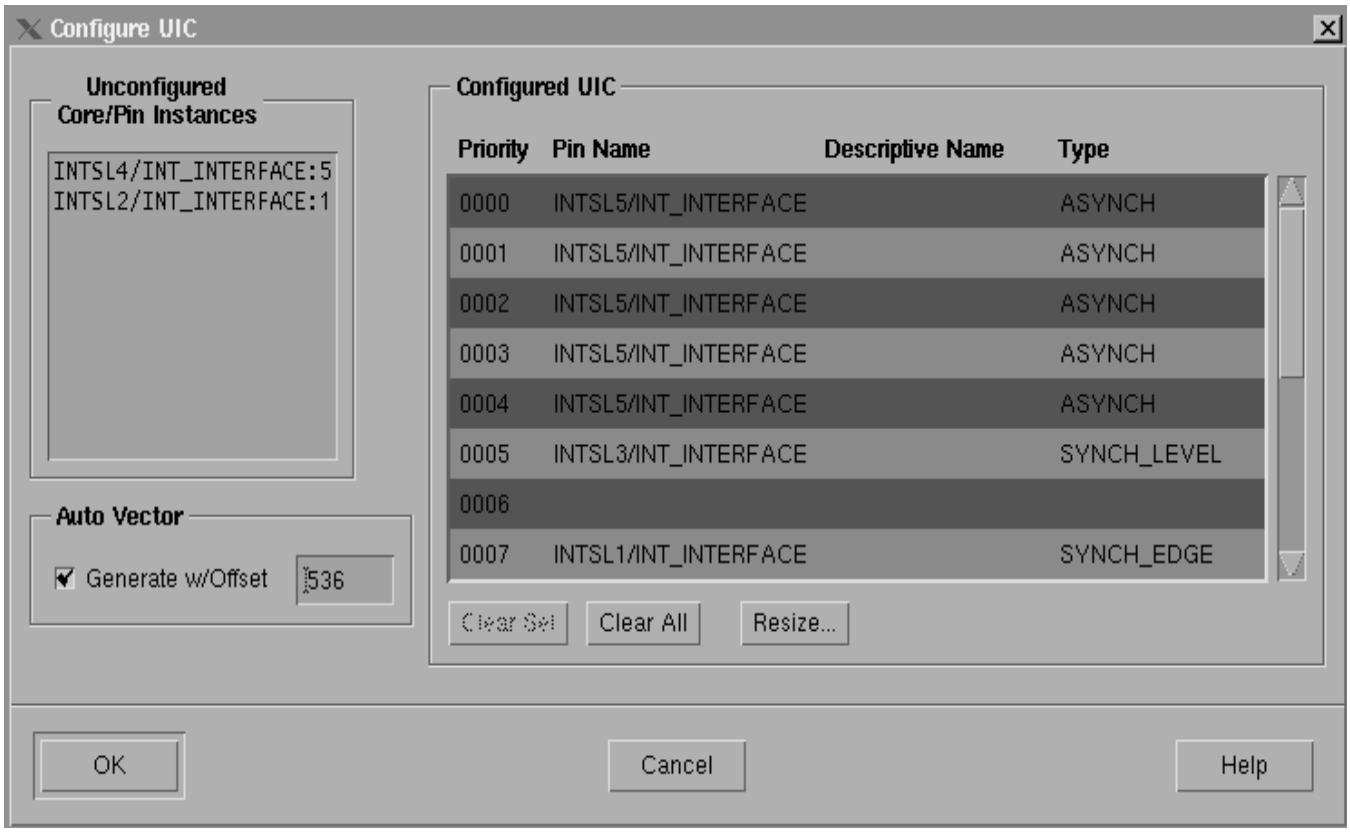


Figure 6. Interrupt Controller configuration panel

By checking properties Coral can automatically extract and provide the designer with the necessary system information for configuring the system. For example, given a list of virtual cores present in the design, the configuration engine can provide the designer with all virtual interfaces/cores that can initiate interrupt requests. Those interfaces will have property INTERFACE_TYPE = INTERRUPT on cores with property INTERRUPT_TYPE = INTERRUPT_SLAVE. It can also list all cores which are interrupt controllers, by checking for property INTERRUPT_TYPE = INTERRUPT_CONTROLLER. Then through the interrupt configuration graphical user interface (shown in Fig. 6), the user can drag and drop interrupt interfaces into the interrupt controller of choice with a given priority. Based on this input, CORAL can create the virtual connections automatically as well.

Another example of this configuration capability is the configuration of masters and slaves devices and address maps. Each master device needs to be associated with a bus and assigned a priority. Each slave device in the design needs to be associated with a given address map in its own bus domain. Again, given a list or virtual cores in the design, the configuration engine automatically identifies all cores which are bus arbiters, bus masters and slaves by checking the appropriate properties. It displays the lists of such cores to the designer, who then selects a bus arbiter to configure. In the case of a PLB arbiter, the configuration engine graphical user interface is shown in Fig. 7. This figure lists on top left the master cores and interfaces, which can be dragged to the right-hand side and dropped in the desired priority position of the PLB Arbiter. On the bottom left, it lists the slave cores which can be dragged to the right-hand side and dropped in the desired address map value. The user can also type in a specific address map value. Based on this information, the configuration engine can automatically create the interconnections between masters, slaves and arbiters in the virtual design.

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.

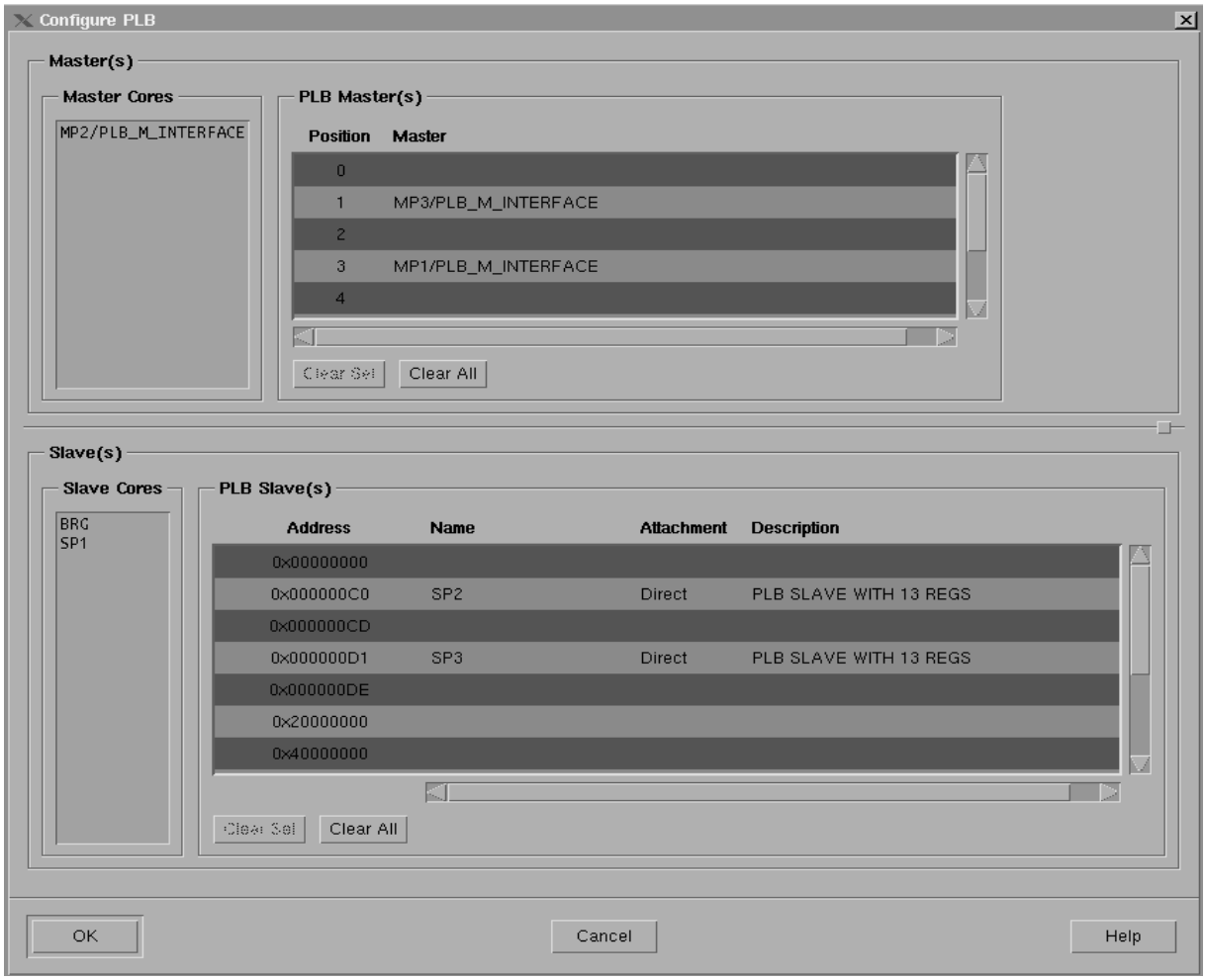


Figure 7. PLB Arbiter/Masters/Slaves configuration panel

3.6 Virtual to Real Synthesis

The virtual-to-real synthesis engine (VRSE) is responsible for synthesizing a real design from a virtual design, as illustrated in Fig. 8. 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 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 *RC* (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*.

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.

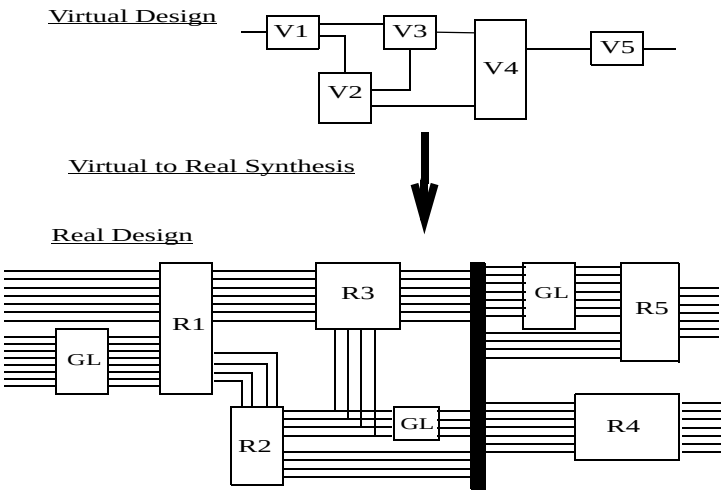


Figure 8. Virtual to real synthesis (GL = glue logic)

The VRSE works as follows:

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. Fig. 9 shows two virtual components *VC1* and *VC2* and pins *v1* and *v2* connected by a virtual net. *VC1* and *VC2* are mapped to real components *RC1* and *RC2* respectively.

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 described earlier). This is illustrated in comparison step 1 shown in Fig. 9. Given a virtual pin *v1* in virtual component VC1, the VRSE compares 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 comparison step 2 shown in Fig. 9. The VRSE automatically determines the direction of the real ports (in, out, inout) and connects them correctly.

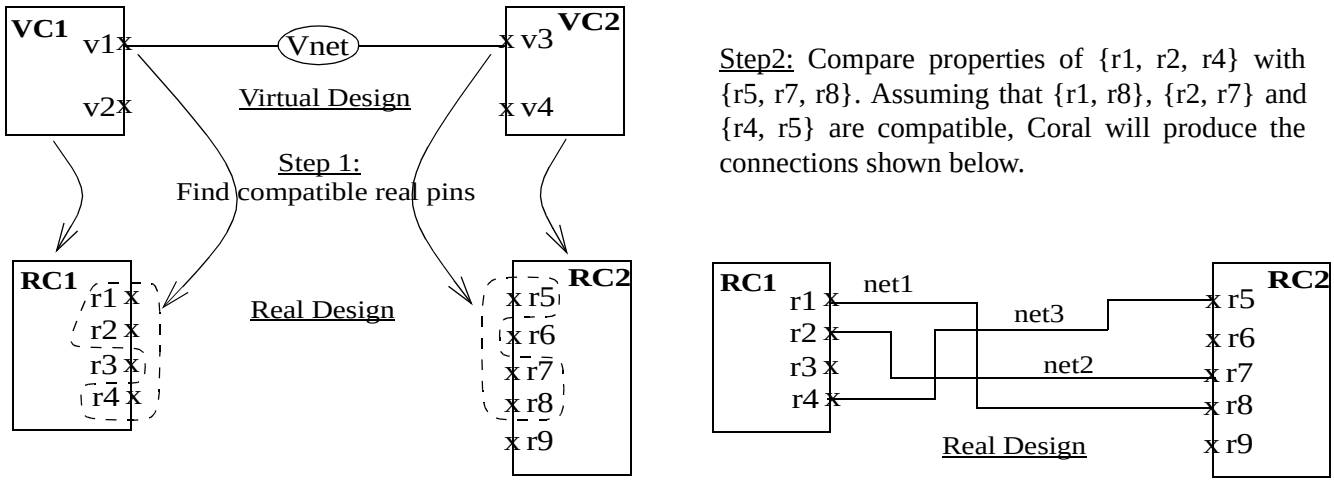


Figure 9. Steps during virtual to real synthesis

Fig. 9 shows simple cases of straight connections. The algorithm also 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.

Fig. 10 shows a more complex example of the virtual to real synthesis process. In this case there is a virtual net connected to three virtual ports in different virtual components. The virtual components and ports get mapped into real cores and ports (which may be inputs or outputs). Then by comparing properties on the real ports, it determines which real ports should be connected together. In this example, it found 3 sets of ports with compatible properties (thus should be connected together): {*r1*, *r6*, *r12*}, {*r2*, *r11*} and {*r3*, *r7*, *r10*}. The VRSE checks whether the ports are drivers or receivers (outputs, inputs, or bidis) and connects them appropriately.

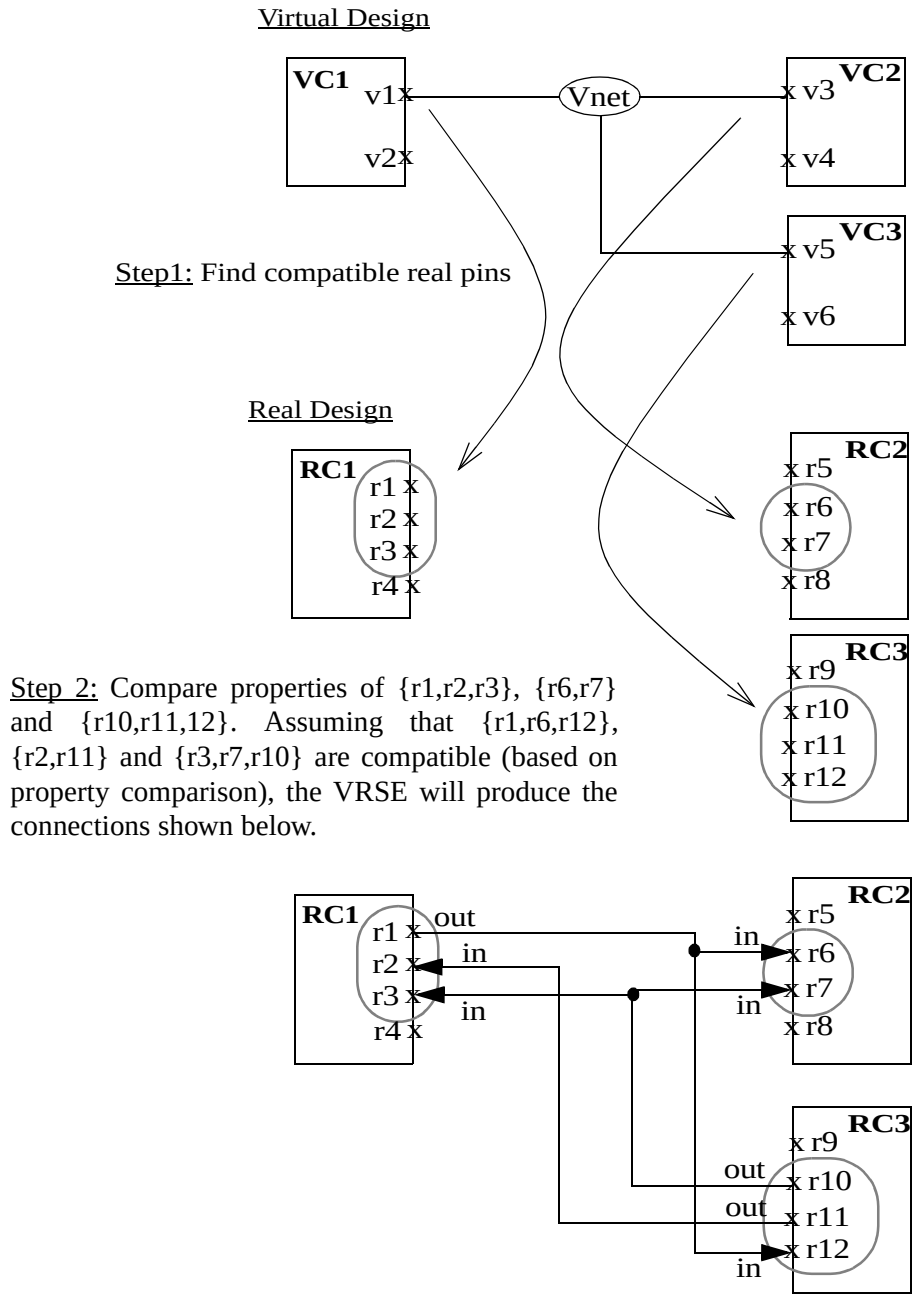


Figure 10. Virtual to real synthesis example with multi-point virtual net

3.6.1. Interconnection Logic

The VRSE is also capable of automatically creating a logic network for interconnecting components. This capability is not aimed at creating full interface logic between components (such as the automatic synthesis of protocols), but primarily targeted at creating simple glue logic which can be described as a simple Boolean expression. Some examples of this capability are described below.

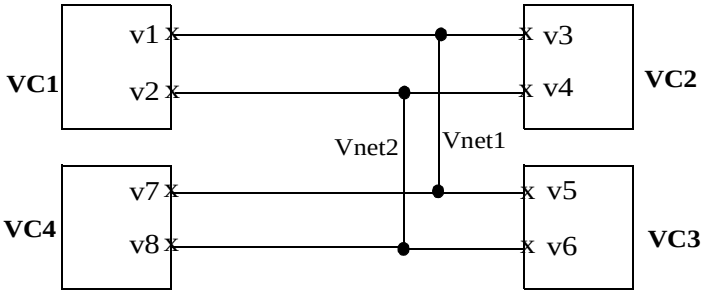
1. Resolving multiple drivers: if two real ports which are deemed compatible are drivers, there may be a contention problem (for example, two outputs and one input being connected together), which is resolved based on a special property called CONNECTION_LOGIC attached to the input port (real port). This property is used for specifying a simple Boolean expression that takes as sources all the drivers and whose sink is connected to the input port. The VRSE creates the gates required for implementing this Boolean expression in the real design and connects the sources and sinks appropriately.
2. Grouping of bits into a bundle: sometimes it is necessary to group multiple source nets and connect all of them (as a bundle) to a sink pin of larger size (number of bits). This situation is implemented by the VRSE in a similar manner to case 1 of multiple drivers. If the sink pin has property CONNECTION_LOGIC with value "CONCAT", then the VRSE groups all source nets into a bundle and connects the whole bundle to the sink pin. Checks are made to ensure that the size of the bundle matches the size of the sink pin, otherwise padding or truncation may occur.
3. Ordering bits in a bundle according to priority: in certain types of connections (e.g., interrupt pins) it is necessary to order the bits in a bundle according to their priority. For example, the interrupt controller core (real component) has a multi-bit input for the interrupt requests. All single-bit interrupt request pins from the cores get bundled in a multi-bit net which is then connected to the interrupt request pin in the interrupt controller. Moreover, each single-bit interrupt request has an interrupt priority and it must be added to the bundle in the bit-position corresponding to its priority. This case is implemented by the VRSE as follows: it first groups all single-bit interrupt request pins into a bundle (as in case 3), and then it reorders the bits in the bundle according to their relative priorities. The priorities should be non-overlapping; it two interrupt requests have the same priority, they are positioned one after the other.
4. Connecting different bit-widths: whenever two real ports are connected together, the VRSE checks if their bit-widths are the same. If they are different, it may be necessary to truncate or pad the source net in order for its width to match the sink port. If the sink width is smaller than the source width, truncation takes place, which can be on the most-significant or least significant side. If the sink width is larger than the source width then padding takes place, which can be done in many ways, e.g., with zeros, sign-extend, little-endian, big-endian.

Fig. 11 presents examples of these four cases. In this case the virtual design contains four virtual components and two virtual nets connected to all four virtual components. Each virtual pin gets expanded into one or two real pins. Fig. 11 shows the virtual to real interface mappings as well as the sizes and direction of all real pins. For example, virtual pin *v1* in *VC1* gets expanded into real pins *r1* and *r2* in real component *RC1*. Pin *r1* is a 1-bit output (bounds 0 to 0), whereas pin *r2* is a 16-bit input (bounds 0 to 15). Fig. 11 shows also the compatible sets of real pins which should be connected together. Using the properties attached to each virtual and real pin, the Interconnection Engine (used by the VRSE) determined for example, that pins {*r1*, *r5*, *r10*, *r13*} should be connected together.

When connecting pins together, the VRSE checks if they are drivers or receivers and their properties as explained earlier in this section. For example, the compatible set {*r1*, *r5*, *r10*, *r13*} has 3 output pins (*r1*, *r10*, *r13*) and 1 input pin (*r5*), hence this is a situation of multiple drivers. Pin *r5* has property CONNECTION_LOGIC = XOR, which specifies the Boolean expression that must be built to implement the interface to pin *r5*. The VRSE creates an XOR gate and connects all driver (output) pins in the compatible set to this XOR gate, and connects its output to the receiver pin *r5*. There may be more than 1 receiver pin in each compatible set.

In the case of compatible set {*r2*, *r6*, *r9*, *r12*}, there is 1 receiver pin *r2* of 16 bits and 3 driver pins *r6*, *r9* and *r12* of sizes 4, 7 and 5 bits respectively. Since pin *r2* has property CONNECTION_LOGIC = CONCAT, the 3 driver pins are concatenated into a 16-bit bundle which is then connected to *r2*. The bundling of these 3 pins is done

Virtual Design



Real Design

Virtual to Real Interface mappings:

Virtual pin v1	maps to	real pin r1(0:0),	output	
		real pin r2(0:15),	input,	CONNECTION_LOGIC = CONCAT
Virtual pin v2	maps to	real pin r3(0:15),	output	
		real pin r4(0:0),	output,	PRIORITY = 2
Virtual pin v3	maps to	real pin r5(0:0),	input,	CONNECTION_LOGIC = XOR
		real pin r6(0:3),	output	
Virtual pin v4	maps to	real pin r7(0:7),	input	
		real pin r8(0:0),	output,	PRIORITY = 0
Virtual pin v5	maps to	real pin r9(5:11),	output	
		real pin r10(0:0),	output	
Virtual pin v6	maps to	real pin r11(0:3),	input,	CONNECTION_LOGIC = CONCAT
Virtual pin v7	maps to	real pin r12(4:0),	output	
		real pin r13(0:0),	output	
Virtual pin v8	maps to	real pin r14(0:0),	output,	PRIORITY = 1

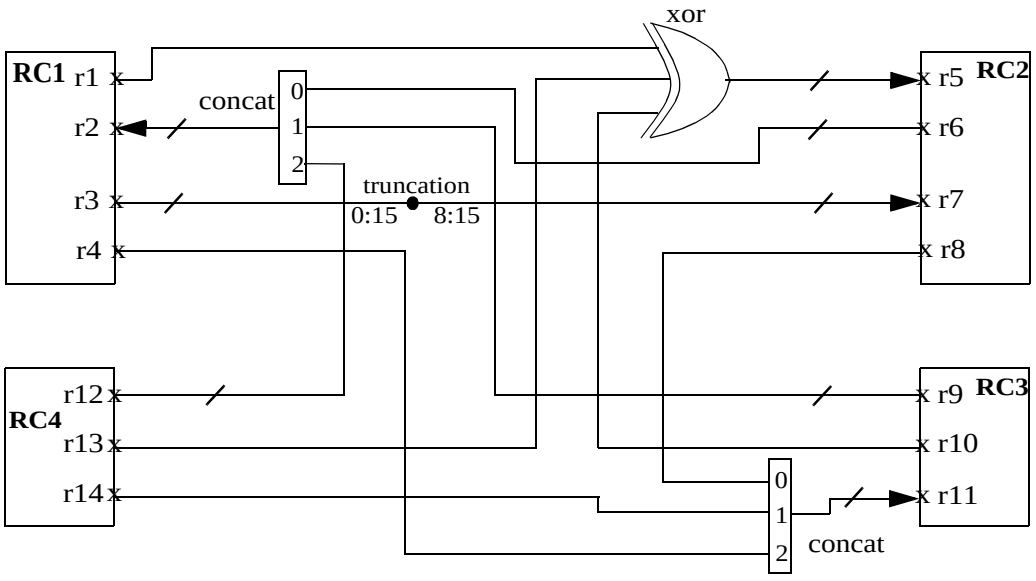


Figure 11. Virtual to real synthesis example with interface logic insertion

without regard to their relative order. Similarly, compatible set {*r4*, *r8*, *r11*, *r14*} also has 3 drivers and 1 receiver (*r11*) with property CONNECTION_LOGIC = CONCAT, however, the driver pins *r4*, *r8* and *r14* have property PRIORITY equals 2, 0 and 1 respectively. Hence, the VRSE when grouping the connections from *r4*, *r8* and *r14* into a 3-bit bundle, it orders the bits in the bundle according to their relative PRIORITY, i.e, pin *r8* is connected to bit 0, pin *r14* to bit 1 and pin *r4* to bit 2.

Compatible set {*r3*, *r7*} illustrates the case where two pins of different sizes need to be connected, therefore requiring a size adjustment. In this case the 16-bit driver pin *r3* (bounds 0 to 15) is being connected to the 8-bit receiver pin *r7* (bounds 0 to 7). The VRSE truncates the net connected to *r3* to 8 bits and connects only those 8 bits to *r7*. The truncation can happen on the most-significant or least-significant side. In this case, the 8 most significant bits of *r4* (bits 0 to 7) were truncated and the remaining bits (bits 8 to 15) were connected to pin *r7*.

3.6.2. Broadcast Connection

Any Soc will contains several connections which are generic, i.e., when a net needs to be connected to a large number of pins. This is the case, for example, for clock and test nets which usually go to almost all cores in a design. In order to make it easier for the designer to implement these connections, the VRSE has a special capability called *broadcast connection*. Using this capability, it is not necessary to connect in the virtual design those virtual interfaces which usually should be connected to most cores. In other words, the designer does not need to worry about connecting the virtual clock, test and other generic interfaces even in the virtual design.

The broadcast connection capability takes place after the VRSE creates the real design from the virtual design. It searches for all unconnected real pins in the real components in the real design, and selects all those that have property BROADCAST_CONNECTION = TRUE. It then compares the properties amongst all selected real pins to generate possible compatible sets. Finally, it connects the real pins in each compatible set together. This scheme is used mostly for clocks and test pins, and it can handle even complex clocking schemes, such as multiple clock frequencies.

4. Summary and Conclusions

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.

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. The virtual design can also be created in schematic form only (without an HDL).

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.

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] W.P. Birmingham, A. Gupta and D.P. Siewiorek, “Micon: Automated Design of Computer Systems”, in “High-Level VLSI Synthesis”, edited by R. Camposano and W. Wolf, Kluwer Academic Press, 1991.

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

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

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

[7] 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.

[8] 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.

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

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