Abstract Communication Model and Automatic Interface generation for IP integration in Hardware/Software Co-design

C. Araujo and E. Barros
Centro de Informática, UFPE

Abstract: The use of standard languages like VHDL and C for the description of hardware and software IP has became a common practice. Despite this, these languages, specially the hardware description languages lack constructs that allow the IP designer to develop highly re-usable IP blocks. In this paper is described an abstract communication mechanism that uses extensions to the VHDL language, communication library for software and automatic interface generation for the easy integration of IP modules.

1. INTRODUCTION

Current processes for IC fabrication allow the easy integration of millions of transistors in a single chip. Despite this evolution in the fabrication processes, designer are unable to fulfill this enormous capacity respecting a more constrained time-to-market. This phenomena is known as "Design Productivity Gap". One of the most promising solutions to this problem is the IP reuse. Where known, reliable system modules are integrated in digital designs.

One problem arises here, how to build and integrate these IP modules to designs. Languages currently used for hardware description, like VHDL and Verilog don't have the abstract communication mechanisms needed for an easy "plug and play" integration of IP modules. Use of mixed hardware/software modules is a task even harder.

In this paper is described the mechanisms used in the PISH Co-design system for an easy integration of IP modules. The solution uses a

---

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35597-9_40
M. Robert et al. (eds.), *SOC Design Methodologies*

combination of automatic interface generation, extension of the VHDL language with the introduction of abstract communication constructors for hardware modules and communication library for the software ones.

The rest of this paper is organized as follows: in section 2 is given an overview of the PISH Co-design system. In next section some of the related works in this area are shown. Section 4 describes the proposed system InterfPISH, which allows automatic interface generation as well as code generation for hardware and software synthesis. A case study and results obtained are given in section 5 and finally a conclusion section is given.

2. THE PISH CO-DESIGN SYSTEM

An overview of the PISH Co-design system can be seen in *Figure 1*. It can be divided into three main stages: specification and partitioning, co-synthesis and prototyping.

The first stage, specification and partitioning taking an initial specification of the digital system to be implemented and partition it into processes to be implemented in hardware and software components. This system uses occam as specification mechanism [4]. The main reason to use occam is that, being based on CSP [5] occam has a simple and a elegant semantics, given in terms of algebraic laws, which allows the partitioning be performed by applying a series of algebraic transformations into the initial occam description in order to preserve the semantics. These transformations change the initial specification and as result new concurrent processes and communication are introduced. Despite being a new description, the transformed one is guaranteed to have the same semantic as the original specification. The set of transformation rules is applied according to the results of a cost analysis obtained by using a Petri net based estimator and clustering techniques [3]. The interface generation depends on the number of concurrent processes of different nature (hardware/software) that communicate, the type of the data being transferred between the processes and also the target architecture taken into account. Most co-design systems considers a very simple architecture composed of one software component. In order to have a pre-defined protocol some systems consider the hardware running as a co-processor, i.e. hardware and software do not execute concurrently [1][2].

In this work, software and hardware can run concurrently and for that device drivers must be generated at the software side, as well as specific hardware to make transparent for the hardware side which processor is being used. The interface between hardware modules must also be synthesized.

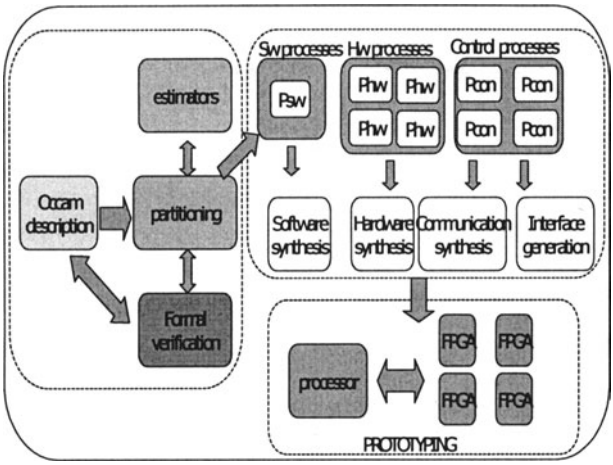

Figure 1: PISH Co-design system

3. THE PROPOSED APPROACH

In this section is described an approach for co-synthesis that allows the generation of code for hardware and software synthesis, including the automatic interface generation between the hardware and software parts of the digital system.

The flow for co-synthesis and interface generation is shown in *Figure 1*. It can be divided in four parts: translation into an internal format representation, threads and communication extraction, interface generation and the last phase code generation. In the first part of the flow, *Figure 1*a, a description of the partitioned system representing the software, hardware and communication processes is given as input.. These processes are described using the occam language. This is the result of the automatic partitioning tool of the PISH system. These descriptions are translated to Petri Net representing the control and a graph representing the data flow.

The second part, *Figure 1*b, performs the capture of the concurrent threads existing in the digital system, the extraction of the communication among the concurrent threads and for the insertion of IO modules in order to allow for interaction with the outside world. Initially are identified in the Petri Net representation all the concurrent threads. These threads can only execute sequential statements. Concurrency in the system is obtained by the simultaneous execution of several threads. Threads can activate others threads in the system and also communicate with other threads through communication channels. After the threads identification, the communication among them is extracted. This information is used for the

implementation of communication channels in hardware and software, depending on the nature of the communicating threads. In the last part of this phase IO elements are inserted in the digital system. These IO elements have a dual mean. They act by one side as an execution thread and can communicate with others threads through the use of communication channels and can also control IO devices, at the other side.

The following part, *Figure 1*c, is responsible for the automatic interface generation between the hardware and software parts of the digital system. In this step the target architecture must be taken into account. The designer can choose a particular target architecture from a library and code for interface between hardware and software is automatic generated. The system may be composed of several concurrent threads and most may want to communicate simultaneously. To handle this situation the generated interface is able to schedule its use of the shared resources by concurrent threads. The data transferred in the communications can also be of any data length. So the interface is also responsible for the transference of data independent of its length.

The last step in *Figure 1* is the code generation phase. VHDL code is automatically generated for the hardware parts of the system while standard C code is generated for the software parts of the digital system. Both codes are standard and can be synthesized by most of the VHDL and C tools.

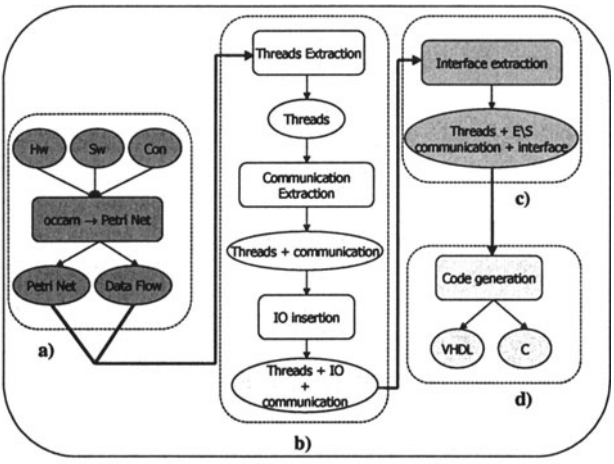

Figure 1: InterPISH: co-synthesis + interface generation

3.1 System Model

In this approach the defined model for the digital system is seen in *Figure 2*. As can be seen it is completely symmetrical, what means that the software and hardware parts of the system are treated in the same way and have basically the same elements. The digital system is composed of concurrent executing processes or threads that communicate through communication channels. These channels implement the synchronous CSP communication semantics [6]. The IO processes are responsible for the control and transfer of data to/from the IO devices. This way the IO processes or IO threads have a dual behavior, they can communicate through communication channels and control IO devices. Between the hardware and software parts of the digital system is the interface component. This component is also symmetrical, with the same parts implemented both in hardware and software. This component performs the communication among threads of different nature and has some important characteristics: is transparent to the communicating threads, it can transfer different data lengths and can also schedule the use of the shared resources among several concurrent threads communicating through the interface.

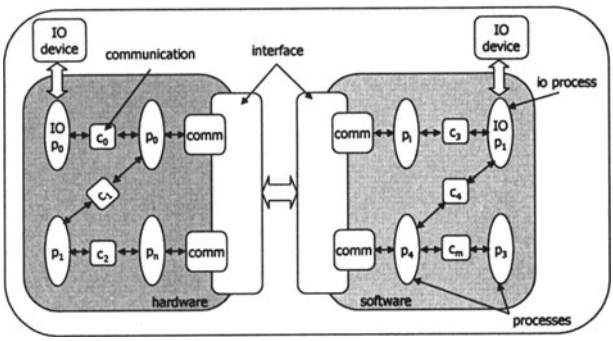

Figure 2: System Model

3.2 Communication Model

Two communication models are used in the implementation of the digital system. Direct communication and synchronous communication by channel.

Direct communication is used in two occasions. The first is used when one thread activates another one. When this happens data may be need by the activated thread, so there is a direct transfer of data from the parent thread (the one that activates) and the son thread (the one that is activated). The second situation happens when one son thread finishes its execution and then returns to the parent thread the data previously transferred. These data

values are returned updated. In *Figure 3* a parent thread activates a son thread. The set {Vp0, ..., Vpn} represents the variables used by the parent thread and {Vf0,..., Vfm} represents the set of variables used by the son thread. The values of the shared variables are transferred from the parent thread to the son thread by using the activation block shown in *Figure 3*. This block is also responsible for sending the activation signal from the parent thread to the son one. The opposite happens with the finalization block. This one, shown in *Figure 4*, is responsible for returning the updated values of shared variables and also to indicate to the parent thread that the son thread has finished its execution.

The second communication type models the synchronous communication by channel. This communication model implements the occam communication semantics for concurrent processes that cannot share variables[7]. This model is shown in *Figure 5* where two concurrent threads p0 and p1 transfer data synchronously through the channel c0.

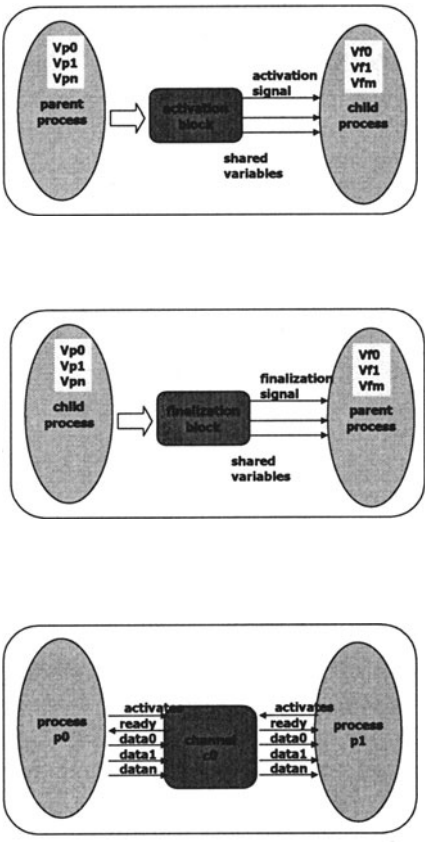

Figure 5: Synchronous channel communication

3.3 Automatic Interface Generation

This section details the automatic interface generation done in the InterfPISH tool. The communication models shown in the previous section are independent on the nature of the two communicating threads. It doesn't matter whether the two threads are implemented in hardware, software or one is in hardware and the other in software. If the two threads have the same nature the communication component (activation block, finalization block and communication channel) is implemented in the same technology (hardware or software). For instance if the two concurrent threads are in hardware and communicate using communication channel, the channel is implemented as a hardware component. In the software case the channel is implemented as a data structure and functions.

When the threads have different natures an interface is built. The model of the interface can be seen in *Figure 6*. The interface model is completely symmetrical and layered. The interface has three layers: prcs_unit(*Figure 6*a), comm_unit(*Figure 6*b) and io_unit(*Figure 6*c).

The prcs_unit layer is responsible for implementing the communication components (activation block, finalization block and communication channel). This makes the communication transparent for the threads, once they only communicate through these components. This layer is responsible for rebuilding the data so the threads can used them.

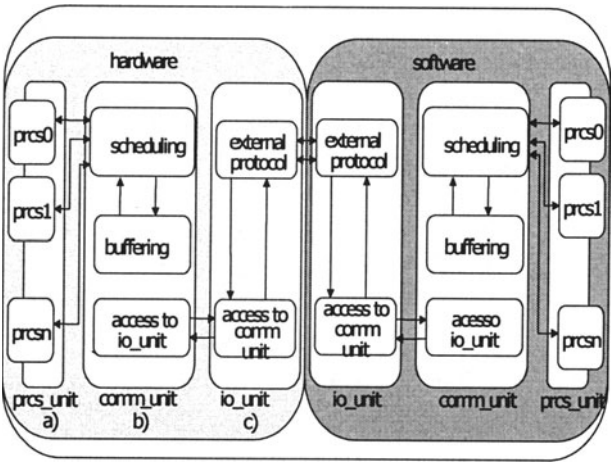

Figure 6: Interface

The second layer, comm_unit, is responsible for the scheduling of the interface and work as a buffer. The scheduling is need because the interface is a limited resource that can be used by several concurrent threads. The buffering function allows that several communications take place while the buffers are not full. This can make the communication faster.

The last layer, io_unit, conects the software component (processor) to the hardware component (FPGA). This layer is the only one that depends on the target architecture. The tool InterfPISH allows the user for choosing one from several io_unit stored in a library depending on the target architecture. This layer is not fixed as the previous one.

3.4 IO Threads

The concurrent threads that compose the system may need some data from the outside world. But these threads are not able to access the IO devices directly. The access is performed by the use of special IO threads. These threads are stored in libraries and the designer can choose the IO thread depending on the device to be controlled and on the data type (length) to be transferred. A process, or thread, p0 gets data from the input device using the channel to receive data from the IO thread. It can include three distinct blocks: a device control block, a type composition block and channel control. The device control block is responsible for getting data from/to the IO device. It is able to activate the control and data lines of the IO device when some data is requested or must be sent to the device. The second block, type composition, is responsible for adapting data types with different lengths to be transferred through the channel. Channels are only able to transfer specific data lengths. This means a channel that transfer a 8 bit integer type cannot transfer a 16 bit word type and it must be handled as two 8 bits data. So the composition block is responsible for composing the data coming from the device control, where the data is seen as a stream of bits, to the channel specific type. The last block is responsible for controlling the communication channel. This makes the IO thread to be seen as an ordinary thread by the communication channel.

3.5 Hardware/Software Co-synthesis

As seen before the digital system is modeled as a set of concurrent threads. Each of these threads executes sequentially. Another characteristic of the threads is that they can activate other threads that run concurrently. The threads must also be responsible for informing its parent threads that they have finished their execution. This characteristic is necessary because once a parent thread activates several concurrent son threads it must wait until all the son threads finish execution. This implements the occam semantics for concurrent processes, where one processes can activate concurrent processes and waits until they all complete [7].

In our case each thread is represented as a FSM (Finite State Machine). This representation is consistent with the model for the digital system because the FSM can execute tasks sequentially and can implement control constructs like decision and loops. In this model the state transition can represent a condition or an action.

As said before the thread is able for activating son threads. This is done by a state transition that signals to the sons that they must leave the initial state. An activation transition can activate any number of son threads. After this transition, the FSM goes to a state where it waits for all the sons to indicate their finalization.

The other state transitions can indicate an action. An action can be one of the following: logical operation, arithmetic operation, decision, null action and stop action, which represents a deadlock of the thread that does nothing and stay forever in this state.

A decision allows a changing in the sequence of states depending on conditions. The decisions can be a conditional or loop.

4. CASE STUDY: ATM SWITCH

In this section it is shown some results by applying the proposed methodology and the tool interfPISH to the interface generation of an ATM switch controller proposed in [8] whose partitioning is described in detail in [9]. This ATM switch controller must decide whether a cell must be sent or not based on four policy algorithms. The aim of discarding a cell is to reduce the traffic on an ATM network.

The ATM switch executes four types of policy: high peak, high average, low peak and low average. The peak policies are related to the maximum rate that is established during connection negotiation. The average policies observe the average number of cell arrivals during the connection. The cells can have two types of priority: high and low. A high priority cell is submitted to the high peak and to the high average policies. If approved in both of them the cell is accepted. In the case the cell is rejected in some of the policies it is submitted to the respective low policy. If reproved in any of the low policies the cell is rejected, on the contrary, if approved in both the low policies the cell is accepted and its priority is verified. When the cell has a high priority, this priority is changed to low priority. A low priority cell is submitted to both the low peak and low average policies and is rejected if not approved in both of them.

In the example all the policies are based on the leaky bucket algorithm. The different values for the parameters X, LCT, I and L determine which policy is being used.

In figure 7 can be seen the partitioning result in occam for the ATM switch. It is composed of the protocol and channel declarations that define the communication among the processes and with the outside world. The partitioned system is composed of several concurrent processes that are under the first PAR constructor. In the figure are highlighted the four policies processes that must be implemented in hardware while all the other processes are implemented in software. From this occam representation of the partitioned system is generated a Petri Net representation. This Petri Net is composed of 156 places and 154 transitions.

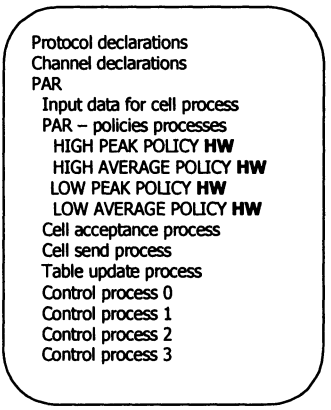

figure 7: Hw/Sw partitioning in occam

The tool extracts the concurrent threads from the Petri Net representation of the partitioned system. In this case 14 concurrent threads are generated and the results are summarised in the *Table 1*. The table gives the number of places and transitions for each thread and also its nature that can be hardware or software thread. The hardware threads are the 4 policy processes and can be noted that they all have the same number of places and transitions. This comes from the fact that only the parameters are different, the policy processes are equal.

Thread type	Number
Software thread	10
Hardware threads	4

Table 1: Thread results

In *Table 2* the results for the IO threads selected by the system designer are shown. In this example all the IO threads are implemented in hardware. For each IO thread two files are generated, an VHDL* file and a VHDL file. The VHDL* language is an extension of VHDL that has constructs for

synchronous communication. From this extension is generated standard VHDL code for synthesis.

Number of IO Threads	Type
6	Hardware

Table 2: IO Threads

As mentioned before the interface is implemented in layers. The last layer is responsible for implementing the 3 types of communication schemes between threads in hardware and software. One file is generated for each policy thread, resulting in four files. In table 4 are shown the results for the interface in hardware.

Block name	Type	Number of lines
Communication	Hardware	1089
Activation	Hardware	944
Finalisation	Hardware	932

Table 3: interface in hardware

The software implementation is simpler than the hardware one. In this case header and C files are generated for the parts of the system to be implemented in software. In *Table 4* are summarized the software results. The first file represents the whole system in software. It contains the main function. The second file, processos.c, implements the threads in software. The next file, comunicacao.c, implements the communication in software. As there are no IO threads to be implemented in software, no files are generated. The last three lines of the table contain the three layers of the interface. As in the case of the hardware interface, the io_unit.c file is generated based on a description of the target architecture while the others are generated automatically.

C file	Lines	H file	Lines
atm_protocolo.c	58	-	-
processos.c	573	processos.h	11
comunicacao.c	1997	comunicacao.h	791
e_s.c	-	-	-
io_unit.c	40	io_unit.h	2
Comm_unit.c	230	comm_unit.h	17
prcs_unit.c	808'	Prcs_unit.h	182

Table 4: software results

5. CONCLUSIONS

In this paper has been described the characteristics and mechanisms of the PISH Co-design system that makes easier the integration of IP modules in a design, independently whether the IP must be implemented in hardware or software. The problem has been approached in two ways. Firstly making easier IO devices integration and secondly through the automatic interface generation. This allows the designer migrate from hardware to software IP´s and vice-versa.

This work uses ideas of HDL extension where abstract communication mechanisms are used for the VHDL language. It also allows the reuse of modules stored in library, both for interface and also for IO devices.

The adopted implementation clearly separates the system in concurrent threads, communication, interface and IO components and uses an strategy based on the use of library components, IO threads and inner part of the interface, and automatic generation of code.

The tool can taken into account different architectures, since new target architectures and new IO threads can be added into the library. This way the designer can have more choices for the implementation of the digital system.

6. REFERENCES

[1] Daniel D. Gajski, Rainer Dömer, Jianwen Zhu *IP-centric Methodology and Design with the SpecC Language* Contribution to NATO-ASI workshop on System Level Synthesis, Il Ciocco, Lucca, Italy, August 1998.

[2] R. Ernst , J. Henkel, T. Benner, *Hardware-Software Co-Synthesis for Microcontrollers*– IEEE Design and Test of Computers, pp. 64-75, December 1993

[3] E.Barros and W. Rosenstiel A Clustering Approach to Support Hardware/Software Partitioning". In: K. Buchenrieder, and J. Rozenblit (eds.), Computer Aided Software/Hardware Engineering. Chapter 11- IEEE Press, 1994.

[4] D. Pountain and D. May, *A Tutorial Introduction to OCCAM Programming.* Inmos BSP Professional Books, (1987).

[5] C. A. R. Hoare, *Communicating Sequential Processes* Prentice-Hall, 1985

[6] C. A. R. Hoare, *Communicating Sequential Processes* Prentice-Hall, 1985

[7] Geof Barret *occam 3 reference manual*, INMOS, 1992.

[8] J.A. G. Lima "Um controlador microprogramado para comutadores ATM". PhD. Thesis, Universidade Federal da Paraiba, Brasil, 1999.

[9] J. Yioda *ParTS – Uma Ferramenta de Suporte ao Particionamento Hardware/Software.* Recife: Universidade Federal de Pernambuco, 1999. Dissertação Mestrado.