EXPLOITING SYNCHRONOUS PLACEMENT FOR ASYNCHRONOUS CIRCUITS ONTO COMMERCIAL FPGAS

Maurizio Tranchero and Leonardo M. Reyneri

Dipartimento di Elettronica — Politecnico di Torino
C.so Duca degli Abruzzi, 24 — 10129 Torino — Italy
Email: maurizio.tranchero@polito.it

ABSTRACT

This paper describes an approach to the placement of self-timed circuits onto commercial FPGAs, using only conventional synchronous tools available on the market. Different parts of the design are constrained in order to maintain the timing relationship required for guaranteeing the correct circuit functionality and to keep the wiring influence on system delays bounded and fixed across the different iterations. This work is part of the extension to the CodeSimulink co-design environment we made in order to allow the synthesis of asynchronous circuits from Simulink specifications.

1. INTRODUCTION

FPGA-based systems are more and more complex and pure-performance is not their main goal. Indeed other second-order issues, like electromagnetic emissions and power consumption, are nowadays important.

These two problems can be faced using self-timed systems [1], which can achieve a reduction in both energy consumption and electromagnetic emissions (EME) [2].

We extended CodeSimulink [3], a Simulink-based co-design environment originally developed for synchronous systems, to clock-less logic [4], in order to fill the lack of high level design for asynchronous design [5]. In this paper we present how we faced issues related to the placement and routing of self-timed circuits onto commercial FPGAs.

This paper is organized as follows: Sec. 2 introduces to the CodeSimulink co-design environment detailing the specification used, the advantages and the drawbacks; Sec. 3 lists issues present in the placement of asynchronous circuits onto standard programmable devices, while in Sec. 4 it is described our proposal. In Sec. 5 a proof of concepts is presented. At the end we conclude the paper with several considerations and future works.

2. CODESIMULINK ENVIRONMENT

In commercial field, to reduce time-to-market, designers often use Simulink-based development [6], due to the environment flexibility in describing and simulating heterogeneous systems. In order to help engineers to save time, there are many tools available in commerce able to automatically convert a Simulink model into a target application. For software implementation The Mathworks provides a well-known Simulink-to-C compiler: Real-Time Workshop. For hardware implementation, instead, there are many possibilities [7, 8, 9], able to generate synthesizable HDL code from a Simulink model.

In this panorama there is also CodeSimulink, a tool developed by our group and offers with respect to the other environments: the ability to describe, implement and simulate heterogeneous systems composed of hardware and software in a transparent way. This is achieved through the data-flow paradigm [10] implementation used, and platform independence.

2.1. Structure and Design Flow

Thanks to the adherence to data-flow model [10], CodeSimulink is an environment which can generate VHDL-code for digital hardware, using an internal compiler, and C-code for software (through The Mathworks' Real-Time Workshop compiler) starting from a Simulink model. It is composed of: i) an ensemble of Simulink-compatible block libraries characterized by implementation-specific parameters (i.e., data representation, number of pipeline stages, physical addresses...); ii) a VHDL implementation of each of them; iii) a set of Matlab scripts for the synthesis process; iv) the needed interfaces between blocks with different implementations; v) other target-specific files. CodeSimulink uses the flow depicted in Fig. 1 for systems based on digital hardware and software components: everything starts with system description, made using Simulink environment and several components libraries. Once the description is ready, we proceed through the partitioning step and the system is divided in hardware and software. Following the hardware branch, it continues with logical synthesis and "Placement & Routing" phases, both performed using commercial tools depending on the actual target device. Software, instead, is

978-1-4244-3892-1/09/$25.00 ©2009 IEEE

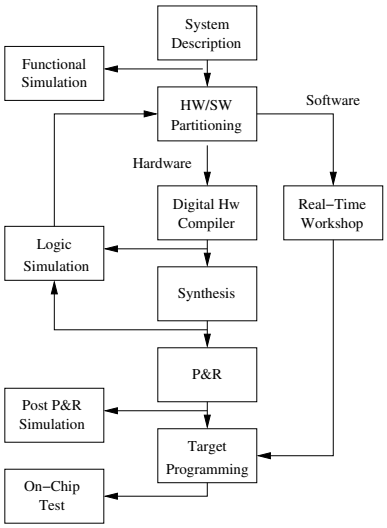

Fig. 1. CodeSimulink co-design flow.

handled by The Real Time Workshop compiler.

Every step can be checked using simulation which is fully integrated in the environment and is automatically generated by CodeSimulink tool, in order to keep consistency between the simulation performed at the highest level and the one used at every lowest level.

2.2. Architecture of a CodeSimulink Block

CodeSimulink was born for generating synchronous systems, but the dataflow implementation of each block leaded to a simple implementation also of asynchronous circuits [4]. Each block translated in VHDL is composed of three parts: one implementing the combinational data-path, one managing the communication protocol and a bank of registers for storing valid data (see Fig. 2). This logical partition allows to different implementation only by changing the protocol manager block. This means that once a CodeSimulink block has been written in VHDL it can be implemented both as synchronous and asynchronous only selecting the desired protocol manager. This results in a great saving of time in developing libraries which is crucial for allowing a short time-to-market. Further details on the asynchronous implementation can be found in [4].

3. TIMING ANALYSIS AND PLACEMENT OF ASYNCHRONOUS SYSTEMS

3.1. Asymmetric Delay Chains

In a bundled-data [1] implementation of an asynchronous circuit, each combinational path needs to be matched by an appropriate delay element. This is required in order to

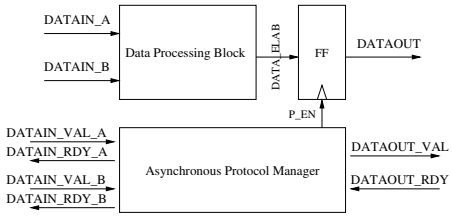

Fig. 2. The structure of an asynchronous block synthesized by CodeSimulink

maintain the correct timing relationship between the path followed by the data, through the combinational elements, and the path followed by the handshaking signals. For the protocol used in our implementation (4-phase), this delay is necessary only on rising edge of the validation signal; instead on the falling one any delay is unwanted, since it limits system speed. For these reasons we need to use an asymmetric delay chain [11].

Since such chains are very important for correct behavior of the whole system, they have to be threaten carefully both during synthesis and placement phase. Description of possible synthesis issues and how we fought them have been presented in our previous paper [12]; solutions to placement problematics are presented here in following sections.

3.2. Timing Analysis and Placement Issues

In order to synthesize a delay chain with the correct number of stages, it is necessary to know accurately the delay introduced by a combinational block. To obtain such precise information it is necessary to rely on the *timing analysis* (TA) process performed by tools present on the market.

3.2.1. Placement

The problem of correctly estimate the delay of a logic block is related on where and how it is physically placed. Moreover the disposition of the delay chain can cause some issues, since it modifies the overall placement structure. For this reason inserting delay chains can lead to wrong results.

The approach proposed here is based on an approach commonly used in literature [13]: the block under test is placed and analyzed in order to have an estimation of area and timings; such information are used to generate both i) a delay chain long enough to at least match the logic delay, and ii) some constraints for the placer tool in order to put the block and the delay chain in the same area, reducing the effect introduced by wiring on circuits timings. The difference is the usage of a mixed process, synchronous and asynchronous to reduce the number of operations and to lead to quick and accurate results using only commercial tools.

3.2.2. *Timing Analysis*

To obtain correct timing information TimeQuest by Altera has been used [14]. This tool provides a TCL shell to automate operation related to timing analysis. Unfortunately there are some issues that make this process not easy. First, since during the synthesis phase most of the signals loose their name (actually it depends on how the HDL code is interpreted and synthesized) it is difficult to isolate each block from the others present in the design to get information only on it. Another issue is related to the asynchronous controllers: they are not easy to analyze for standard tools since they contain combinational loops which are tricky for conventional TA tools.

A typical solution is to break combinational loops and perform the TA on the obtained circuit. Even if this approach is quite easy, there is still the problem related to block isolation. The solution proposed addresses both issues and is described in following section.

4. SYNCHRONOUS PLACEMENT OF ASYNCHRONOUS BLOCKS

4.1. The Idea

As presented in Sec. 2, CodeSimulink is based on a modular structure, which separates the computation part from the protocol management.

We exploit such modularity in order to use the commercially available P&R tools. Indeed for them it is straightforward to place and route a synchronous design, so we synthesize and place a synchronous version of our system. After this operation we retrieve all the informations we need (i.e., area, position, and delays) for each block in our design. Then, using ad hoc constraints, we can substitute the synchronous controllers with the asynchronous ones and start an iterative loop of placement in order to maintain the timing consistency between the data paths and the delay chains.

4.2. Placement Constraints

Commercial placement tools for FPGAs offer interfaces to constrain this phase as desired. Since the overall work has been based on Altera tools, also in this case it has been used the environment provided by Altera: the "Logic Lock" regions (LLRs).

Logic Lock regions are design partitions used by Quartus II to guide the incremental compilation process and to guide design placement. Such regions are rectangular areas in which the design, or a part of it, has to be placed. They can have different characteristics: i) fixed area and fixed position; ii) fixed area and floating position; iii) auto-sized area and floating position. Auto-sized, floating regions are very powerful since they allow the placer to perform small changes, without changing the locality of a design part.

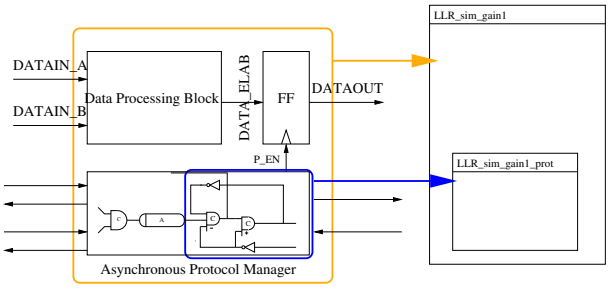

Fig. 3. Logic Lock regions structure for a given CodeSimulink block.

LLs can be used to place *equipotential regions* [11] in a CodeSimulink design. Equipotential regions are those in which signal delays can be considered negligible and if delays can be neglected, forks can be considered isochronic. Using such structures it is possible to bind each delay element to the combinational path it belongs to, and to avoid strance placement for the asynchronous controller. Figure 3 shows the LL structure used. The overall placement procedure follows these steps:

1. The model is copied and synthesized as synchronous, declaring each block function as an auto-sized, floating logic lock region.
2. After the synthesis procedure, TA is performed on the model in order to get a starting point for the delay chain synthesis.
3. Area constraints are obtained by the same model, logic lock are fixed, and passed to the asynchronous one.
4. The asynchronous model is updated with the number of delay stages in each chain and synthesized.
5. After the synthesis, the obtained circuit is analyzed in order to retrieve information on the delay chains.
6. if the delay introduced by the delay chain does not match the one introduced by the combinational path, the algorithm continues from step 4, otherwise it is finished.

4.3. Device Characterization

The standard approach is to run the placement and routing process, to perform timing analysis on the resulted design, and to correct the number of delay chain stages. Unfortunately changing the number of stages even in a small part of the design can change its placement, its routing, and consequently its delay. It is obvious to see that this approach needs to be iterative and for this reason can be quite time consuming.

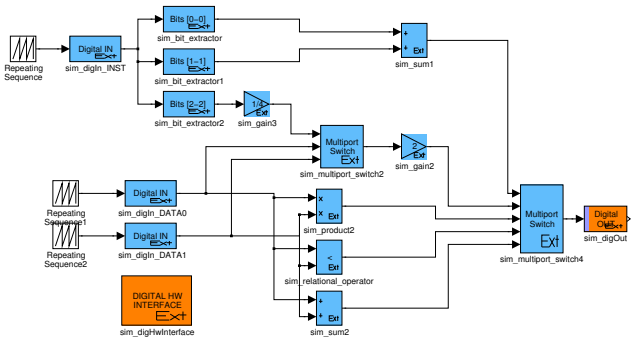

Fig. 4. CodeSimulink asynchronous model of simple datapath.

Table 1. Synthesis overhead of the two case studies used.

	Cells	Regs	Speed
Datapath (Synch)	380	179	132 MSa/s
Datapath (Asynch)	394	187	$45 \div 80$ MSa/s

To mitigate this drawback is to characterize automatically the device once for a great variety of different design (namely for each block and for different bit-size of them) and to use such information to initially bias the iterative placement process described in Sec. 4.2.

5. RESULTS

To test our methodology we developed a small model including different blocks of the available library. This model, called datapath, is a design in which depending on one input several operations are performed on the others two. Operations available are bit-inversion, sum, multiplications, right-shift. Operands are on 16 bit words and results are generated on 32 bits. This design has been synthesized and in Tab. 1 are presented area and timing results and those are compared with synchronous implementation of the same system.

If we look at the timing results shown in Tab. 1, we can notice there is a strong difference between the synchronous and the asynchronous version. The reason of that is due to two main reasons. First the 4-phase handshaking protocol, which includes extra delay on the circuit operation; they are mainly related to the falling transition of both validation and readiness signals, which is still quite long even implementing asymmetric delay chains. Second the conservative rules of placement introduced by us, which are including an extra delay on the delay chain to guarantee the functionality.

6. CONCLUSIONS AND FUTURE WORKS

This paper has shown a way to exploit synchronous placement tools for the placement of asynchronous circuits onto commercial FPGAs. It overcome the problematics related to the timing analysis of asynchronous circuits using a synchronous approach and uses hierarchical logic regions to maintain the logic blocks and handshake controllers as tight as possible. This results in simpler delay chain insertion and matching with combinational paths. At the end a small example has been used to test the methodology.

Next steps will focus on the test of the overall design environment with a real test case.

7. REFERENCES

[1] J. Sparso and S. Furber, *Principles of Asynchronous Circuits Design*. Kluwer Academic Publishers, 2001.

[2] C. Van Berkel, M. Josephs, and S. Nowick, "Scanning the technology: Applications of asynchronous circuits," *Proceedings of the IEEE*, vol. 87, no. 2, February 1999.

[3] L. Reyneri, F. Cucinotta, A. Serra, and L. Lavagno, "A hardware/software co-design flow and ip library based on Simulink," *DAC*, June 2001.

[4] M. Tranchero and L. Reyneri, "Automatic generation of self-timed circuits from Simulink specifications," *International Conference on Electronics, Circuits and Systems*, December 2007.

[5] A. Taubin, J. Cortadella, and L. Lavagno, *Design Automation Of Real-Life Asynchronous Devices And Systems*. United States: Now Publishers Inc, 2007.

[6] The Mathworks. Simulink on-line documentation. [Online]: http://www.mathworks.com/products/simulink/

[7] Xilinx Co. System generator. [Online]: http://www.xilinx.com/ise/optional_prod/system_generator.htm

[8] Altera. DSP-Builder. [Online]: http://www.altera.com/products/software/products/dsp/dsp-builder.html

[9] The Mathworks. HDL-Coder. [Online]: http://www.mathworks.com/products/slhdlcoder/

[10] A. H. Veen, "Dataflow machine architecture," *ACM Computer Surveys*, vol. 18, no. 4, Decembrer 1986.

[11] C. L. Seitz, "System timing," in *Introduction to (VLSI) Systems*, C. A. Mead and L. A. Conway, Eds. Addison Wesley, 1980, ch. 7.

[12] M. Tranchero and L. Reyneri, "Implementation of self-timed circuits onto fpgas using commercial tools," *11th Euromicro Conference on Digital System Design*, September 2008.

[13] C. Sotiriou, "Implementing asynchronous circuits using a conventional EDA tool flow," in *Proceedings of the Design Automation Conference (DAC)*. New Orleans: IEEE Computer Society, June 10–14 2002.

[14] Altera, *TimeQuest Timing Analyzer*, 2007. [Online]: http://www.altera.com/literature/hb/qts/qts_qii53018.pdf