SCAFFI: An intrachip FPGA asynchronous interface based on hard macros

Julian Pontes, Rafael Soares, Ewerson Carvalho, Fernando Moraes, Ney Calazans
Faculty of Informatics, PUCRS, Porto Alegre, Brazil
email: {jpontes, rsoares, ecarvalho, moraes, calazans}@inf.pucrs.br

Abstract

Building fully synchronous VLSI circuits is becoming less viable as circuit geometries evolve. However, before the adoption of purely asynchronous strategies in VLSI design, globally asynchronous, locally synchronous (GALS) design approaches should take over. The design of circuits using complex field programmable components like state of the art FPGAs follows this same trend. In GALS design, a critical step is the definition of asynchronous interfaces between synchronous regions. This paper proposes SCAFFI, a new asynchronous interface to interconnect modules inside FPGAs. The interface is based on clock stretching techniques to avoid metastability. Differently from other interfaces, it can use both logic levels for stretching and do not require the use of arbiters. Also, compactness of the implementation is enhanced by the use of dedicated FPGA hard macros. A GALS version implementation of an RSA cryptography core demonstrates the use of SCAFFI.

1 Introduction

It is undisputable that non-synchronous design methodologies for digital systems offer a series of advantages when compared to the widespread synchronous design style. Among the advantages is a potential for lower power consumption and average case performance. Nonetheless, the design simplification provided by synchronous design and the automated tools support have, for a long time, made the choice between synchronous and non-synchronous design easy in favor of the former. Nowadays, the picture is changing, due to increasing problems of clocking billion-gate designs at gigahertz operating frequencies. A chip can no longer be crossed by an electric signal in one clock period [1]. Also, the power implied by a global clock distribution tree dominates chip power dissipation [2].

The division of a digital system into modules controlled each by a different clock domain allows reducing the problems faced by nanoscale circuits.

Using multiple synchronous modules asynchronously connected is called Globally Asynchronous, Locally Synchronous (GALS) design [3]. Synchronous techniques are promptly applicable at module level, and the interface design becomes a new design task.

The development of GALS systems in modern commercial FPGAs is possible, since most devices count more than one clock domain. Indeed, high end devices contain several dozen clock domains. However, primitives necessary to build GALS asynchronous interfaces are not directly available in such devices. While some FPGA architectures were proposed to support asynchronous circuit design, most targeted a particular design style and so far none of these are viable commercial products. Previous works proposed techniques to implement asynchronous or GALS systems in commercial FPGAs, e.g. [4][5][6]. This work proposes *Stretchable Clock Asynchronous Flexible FPGA Interface* (SCAFFI), a flexible interface to support the construction of GALS systems in FPGAs.

The rest of this paper is organized as follows. Section 2 gives basic definitions and reviews proposals of asynchronous interfaces that can be implemented on commercial FPGAs. Section 3 describes SCAFFI design. Section 4 addresses how hard macros can be used to support SCAFFI implementation. An RSA core using SCAFFI is the subject of Section 5. Conclusions and directions for future work appear in Section 6.

2 Asynchronous interfaces in FPGAs

The design of asynchronous interfaces in FPGAs is complex, since these devices were conceived to support synchronous design only. Thus, FPGAs lack asynchronous design primitives such as arbiters and synchronizers and do not allow, as ASICs do, that these be constructed at the layout level.

The design of asynchronous circuits in general and asynchronous interfaces in particular assumes the respect of timing restrictions. The nature of these restrictions varies according to the adopted asynchronous design style. An asynchronous interface design style comprises choices of: (*i*) communication protocol, (*ii*) data encoding, and (*iii*) synchronization strategy.

The simplest communication protocol is the explicit handshake between a sender and a receiver. It can be implemented using edge or level signaling, respectively named *2-phase* and *4-phase handshake*.

A commonly used data encoding scheme in asynchronous systems is the same used in practically any synchronous system, the *binary* or *single track encoding*. Here, a wire represents exactly one bit of information. An asynchronous communication interface employing single track encoding associated to a handshake protocol defines a communication mechanism called *bundled data*, where data availability/validity is indicated by an explicit request signal (Req) and data reception is signaled by an explicit acknowledgement signal (Ack). This implies that data bus signals must be delayed by an amount of time smaller than the Req signal. Alternatively, the controller generating the Req must be designed so that Req only occurs after a stable data value is available at the input of the Receiver.

To eliminate/reduce bundle data constraints, schemes that carry validity information inside data can be used. These encodings allow developing communication protocols where it is not necessary to impose timing restrictions on control signals, apart from the isochronic fork restriction [7]. Such encoding schemes are called *delay insensitive* (DI) *encodings* or *codes*.

GALS systems need a synchronization mechanism. This encompasses defining a safe data sampling process for synchronous islands to acquire data from an asynchronous environment. Safeness is defined with regard to metastability avoidance or confinement. The most commonly employed synchronization strategy consists in using two series flip-flops clocked by the receiver. This does not eliminate metastability, but drastically reduces the probability that its occurrence result in synchronization failures. Ginosar [8] presents several variations of this strategy and analyzes the influence of each in the robustness and correctness of the interface. A problem with this approach is the increased data transfer latency.

Using pausible clocks can eliminate the risk of metastability at the interface. The clock of each communicating synchronous module is paused (or *stretched*) before the data transfer, and then restarted when data is stable. Clock stretching is the task of an arbiter or mutual exclusion (ME) element. ME devices usually employ RS latches and specially built filter devices. Muttersbach et al. [9] and Moore et al. [10] present GALS communication interfaces proposals for ASICs.

Few works proposed GALS communication interfaces for FPGAs [5][6]. The main problem here is the implementation of metastability-free mutual exclusion. Najibi et al. [6] implemented a GALS system using as ME device an RS latch. To avoid metastability, they propose that the clock stretching request signal cross a latch sensitive to the high logic level. However, this method only transfers the risk of metastability from the arbiter to the latch itself. In another work, Moore and Robinson [5] present an arbiter that can be implemented in commercial FPGAs. The arbiter has its own clock, and its structure is a variation of the series two flip-flops approach. Request and the system clock must be synchronized with the arbiter clock, so that arbitration succeeds. Interfaces based on this arbiter present reduced throughput compared to two flip-flops.

3 SCAFFI: a new interface proposal

This Section presents the architecture of a new interface, called SCAFFI, which is useful to overcome some disadvantages pointed out on previously proposed interfaces. SCAFFI is point-to-point, unidirectional and supports both bundled data and delay or quasi-delay insensitive (QDI) communication styles. Two instances of SCAFFI can provide a bidirectional communication interface. Sections 3.1 through 3.3 discuss this architecture. For more reliable data transmission, the basic design can be improved with additional modules. Section 3.4 explores this.

3.1 Basic architecture - bundled data style

The SCAFFI basic architecture is depicted in Fig. 1. Physically, it occupies part of a Sender and part of a Receiver sharing a data communication channel. SCAFFI employs clock stretching techniques.

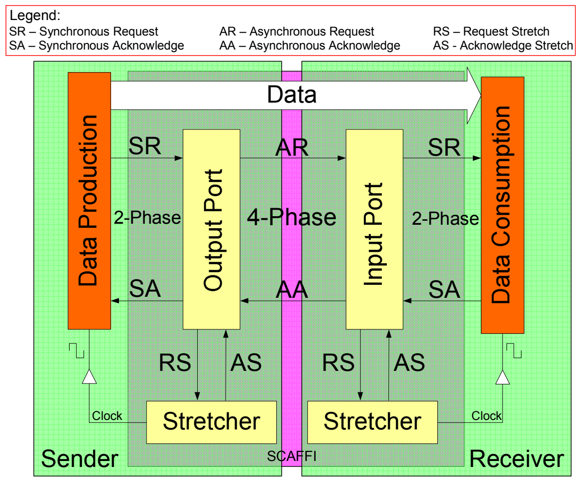

Fig. 1. Structure of SCAFFI, showing the Sender and Receiver sides of the interface.

The basic architecture supports bundled data communication, leading to a small footprint interface adapted to short-range connections. Output and Input Ports employ 2-phase handshake between them and the neighbor synchronous island (respectively, Sender and Receiver in Fig. 1), to improve local communication. These same ports communicate with one another using a 4-phase handshake, to improve robustness.

3.2 The stretcher

The clock stretching technique of the GALS interfaces proposed in [5] and [6] employs a ME mechanism to decide when executing clock stretching. If ME is used to control stretching, the clock can only be paused at one of its logic levels, either at 0 or at 1. If the requisition occurs when the clock is at the other logic level, stretching can only occur in the next half of a cycle, which adds unneeded delay to communication.

SCAFFI uses a stretcher to eliminate MEs, allowing stretching to occur at any logic level. The stretcher is depicted in Fig. 2. It has two control signals and a Clock output. The control signals are input Req and output Ack, respectively connected to SCAFFI RS and AS signals. The stretcher consists in a ring oscillator controlled by a multiplexer (mux).

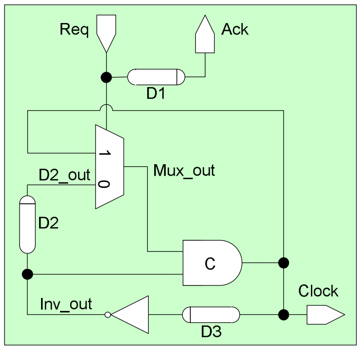

Fig. 2. Structure of the stretcher. The oscillator ring includes D3, the inverter, D2, the mux and the C-element.

The mux Req input controls the ring oscillator feedback path. Starting at the Clock output, this path crosses the delay element D3, the inverter, the delay element D2, the mux and the C-element, before returning to its starting point. The mux and C-element have hazard free implementations for fundamental mode operation.

Glitches may occur at the mux output, but D2 is dimensioned to ensure these glitches do not propagate to the Clock output. This occurs because every possible mux glitch is produced when either D2 or the C-element output transition near some assertion or deassertion of Req. However, these can only change when the lower input of the C-element is stable, due to D2 or C-element delays. A glitch may cause the C-element to transition, but only once. After that, the C-element works as a glitch filter. ASIC Spice simulations of this circuit and the FPGA implementation showed identical behaviors. Both, stretch high and low clock levels.

Whenever Req is unasserted (Req=0), the Clock output oscillates at a frequency controlled by the dimensioning of D3. When Req is asserted, the mux and the C-element keep the Clock output logic level stable (either at 0 or 1), performing the stretch. Finally, D1 is dimensioned to produce the Ack output only after the Clock output is stable, taking longer than the combined delay of the mux and the C-element.

3.3 SCAFFI input and output ports

SCAFFI Input and Output Ports have burst mode specifications depicted in Fig. 3. This specification was implemented as a hazard-free circuit using the MINIMALIST tool [11].

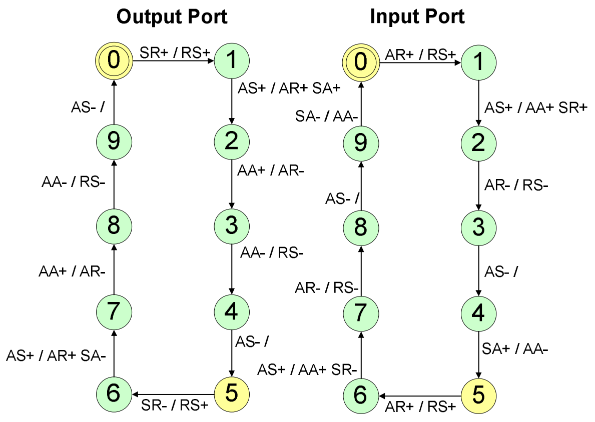

Fig. 3. Burst mode specification for SCAFFI Ports.

The resulting hazard-free logic equations for these controllers appear in Table 1. Signal Y0 corresponds to the internal feedback line of each controller.

Table 1. Equations for Input and Output Port behaviors.

Output Port	InPUT Port
$RS = AA + \overline{SR} * Y_0 + SR * \overline{Y_0}$	$RS = AR$
$AR = SR * AS * \overline{AA} * \overline{Y_0} + \overline{SR} * AS * \overline{AA} * Y_0$	$SR = \overline{SA} * AS + Y_0 + SA * \overline{AS} * AR$
$SA = SR * Y_0 + \overline{AS} * Y_0 + SR * AS$	$AA = \overline{SA} * Y_0 + AS + SA * \overline{AR} * \overline{Y_0}$
$Y_0 = SR * AA + SR * Y_0 + \overline{AA} * Y_0$	$Y_0 = \overline{SA} * AS * \overline{AR} + \overline{AR} * Y_0$

The behavior of SCAFFI is illustrated by the waveforms of the timing simulation of Fig. 4, which depicts a single data transmission taking place. To illustrate the local use of 2-phase protocols, the simulation shows communication with Ports initially at state 5. State 0 would be another possible starting point.

The simulation depicts the Input and Output Ports behavior. Signals are listed in ascending assertion order, illustrating the interface modules behavior. A data transmission starts when the Sender inserts information at the Data lines and asserts a synchronous requisition (SR-), asking the Output Port to start a communication. This action triggers the sequence of control signal assertions to asynchronously transmit data through SCAFFI. At the Sender side, the initial transitions are SR- → RS+ → AS+ → AR+ → SA-. After this sequence, the Sender clock is paused and data to transmit is available and stable. Actions on the Receiver side follow, where the sequence AR+=RS+→AS+→AA+→SR- takes place. Next, the Receiver clock is paused and a synchronous request is placed on the Receiver. This later cannot respond immediately to the request, since its clock is paused. However, transition AA+ triggers the Output port to produce AR- (Output port, transition 7 to 8), releasing the Receiver clock to restart oscillating (marked by Producer AS-). Thus, when the Receiver clock restarts, its synchronous request signal SR is assuredly stable. Follows data consumption at the Receiver side, controlled by the control sequence SA-→AA- and clock restart at the Sender side, with the sequence AA-→RS-→AS-. At the next rising edge of the Sender clock, a new transmission can already start.

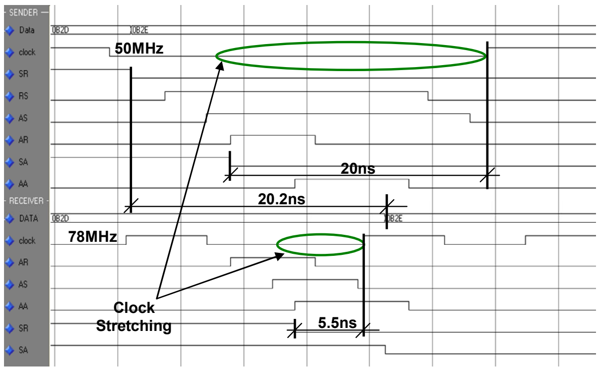

Fig. 4. Example timing simulation of a data transmission using SCAFFI. AR and AA signals are repeated for Sender and Receiver, to ease reference. RS is the same as Receiver signal AR (see Table 1). Ports start at state 5.

Fig. 4 also contains typical quantitative data for a SCAFFI implementation for a XC3S200-4 Spartan3 Xilinx FPGA. For a Sender working at 50MHz and a Receiver at 78MHz, Sender stretching is around 20ns, and Receiver stretching is only 5.5ns. The time to transmit a single data is 20.2ns from the start of the handshake (SR-) until data is available to the Receiver. This is sufficient to ensure that bundle data constraints are met. However, some time is needed until the receiver clock is restarted. Conceptually, less than twice the Sender clock period is enough for any transmission to take place, but this is influenced by the sender/receiver frequency relationship. For this case, SCAFFI achieves a throughput of 31 MegaWords/s, nearly 4 times the throughput achievable with series flip-flop synchronizers using 2-phase handshake.

3.4 Dual rail SCAFFI

Two modules that need to communicate asynchronously may be far apart. This reduces the feasibility of using bundled data interfaces, since it is difficult to guarantee that request signals always have a delay longer than all data lines. The situation is especially critical for wide channels expected in deep submicron technologies, due to delay dispersion caused by effects like crosstalk. One way to consider such effects is using DI interfaces. SCAFFI is based on a module library with components that allow implementing point to point delay insensitive interfaces by using dual rail data transmission. The basic SCAFFI can be enhanced with *Single to Dual* and *Dual to Single* rail converters inserted at Sender and Receiver sides, respectively. Fig. 5 depicts a dual rail SCAFFI interface.

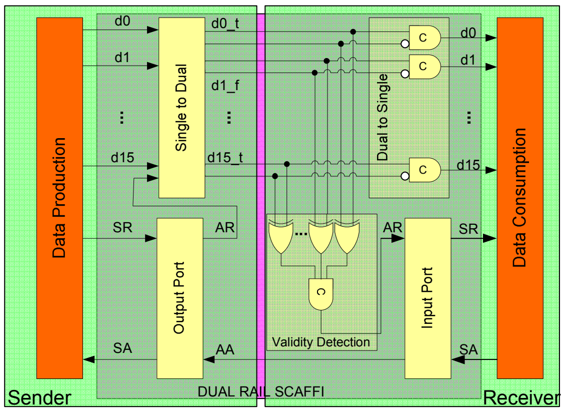

Fig. 5. Dual rail SCAFFI for distant Sender/Receiver pairs. Stretchers were omitted for clarity purposes.

The asynchronous request AR is embedded within dual rail data lines and *Validity Detection* generates the receiver side version of AR. For wide n data bundles, this can be a quite large macro, requiring n XOR gates and a tree of C-elements. The AA signal is connected as before. While the Single to Dual module is standard HDL, the Dual to Single converter and the Validity Detection module are hard macros.

A dual rail register in the library enables an additional enhancement to SCAFFI. It can be used to implement SCAFFI with an associated asynchronous FIFO. This is useful e.g. to create an efficient interface between a synchronous and a dual rail QDI module.

4 The hard macro library

Hard macros are a step further in controlling FPGA synthesis results, since a manually designed layout is produced for critical parts of the design. They imply enhanced control, and of course, increased design complexity. Hard macros are not new to FPGA design. They have been used, for example, by Martín-Langerwerf et al. in [12] to reduce FPGA chip count and synthesis runtime for video applications. The present work employs hard macros for implementing asynchronous primitives, enabling the use of non-synchronous design techniques in FPGAs in a compact way. To the knowledge of the Authors this is the first work proposing hard macros for asynchronous design.

In Xilinx FPGAs, a physical hard macro is a module created from FPGA primitive components like Look-Up Tables (LUTs), flip-flops and wires. Hard macros are specific for a given device of some family, but are independent of device speed grade, and can be placed in multiple positions and instantiated multiple times. Hard macros can be created using the graphical layout tool FPGA Editor, provided by Xilinx. In this work, experimental hard macro libraries to support asynchronous circuit design have been implemented for three Xilinx FPGAs: Spartan-3 XC3S200 and Virtex-II XC2V1000 and XC2V4000 devices.

Hard macros allow controlling net delays more precisely than the use of higher level constraints. For instance, constraints allow specifying the maximum delay of a net, but do not allow defining a delay relationship among wires composing a net. This is fundamental to safely implement both symmetric (as in SCAFFI Ports) and asymmetric (as in C-element feedback) isochronic forks. Once a hard macro layout respects an isochronic fork constraint, every instance of it has this characteristic. All hard macros in the library proposed have timing constraints respected. The SCAFFI Output Port is a hard macro built as depicted in Fig. 6.

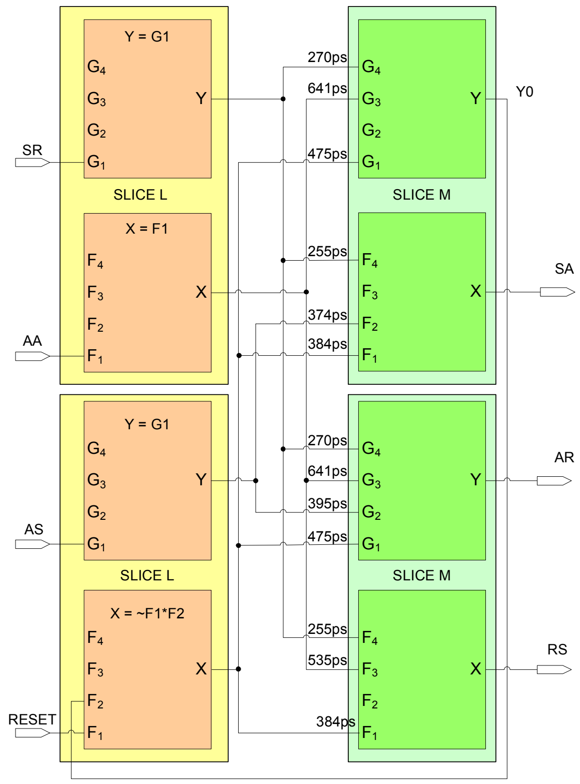

Fig. 6. Example of library hard macro implementation, the SCAFFI Output Port. For behavior, see Table 1.

The Output Port takes exactly one FPGA Configurable Logic Block (CLB) to implement. A CLB contains four Slices, each with two 4-input LUTs. To achieve the isochronic fork timing requirements in this macro, every signal originating such a fork must have a single entry point in the macro layout. This is obtained using transparent LUTs, i.e. a LUT performing the identity function (3 top LUTs at the left, in Fig. 6). The LUT output feeds the isochronic fork net. Each fork output delay can be computed within the FPGA Editor.

In Fig. 6 it is possible to note the four isochronic forks with individual output delays marked (between L and M slices). The way to define the best isochronic fork layout is unfortunately a trial and error process, because no method to compute these delays for all possibilities exists, but the amount of combinations is usually small. Also, the transparent LUTs add to the delay of the hard macro, but this loss is usually offset by the gains in the hard macro optimized layout.

5 RSA cryptography: a use case

The implementation of an RSA cryptography core served to validate SCAFFI. It demonstrates the potential of non-synchronous design to save power compared to synchronous implementations. The RSA core is an IP that executes modular exponentiation. This operation is computed as a control loop, where each step executes a modular multiplication. Multiplication operands and the result are 128-bit values.

Here, the proposed structure of the implementation is implementing RSA as two modules interconnected through SCAFFI: modular exponentiation (MX, the Sender), and modular multiplication (MM, the Receiver). The application is described in VHDL and SCAFFI is a just set of module instances written in VHDL and taken from the hard macro library. The synthesis environment just need to know which file contains the library.

Given the sequential behavior of RSA, the GALS implementation used a slightly modified version of SCAFFI, that keeps the exponentiation clock paused during multiplication. This has a major impact in lowering power. The SCAFFI interface follows loosely the basic architecture presented in Fig. 1. Due to the application characteristics, it is not necessary to use two instances of SCAFFI for sending operands and receiving the multiplication result. Instead, the data bundle includes two 128-bit buses from Sender to Receiver and one 128-bit bus from Receiver to Sender. Sender only generates an acknowledgement to the Receiver after multiplication is finished and the 128-bit result is stable on the Receiver to Sender bus.

The GALS RSA implementation was compared to a synchronous RSA with equivalent functionality. Timing simulation helped validating the designs. These were prototyped on the XC3S200 FPGA. Table 2 gives maximum operating frequencies and clock loads for both implementations, obtained from the post place and route timing report.

Table 2. Maximum operating frequencies for RSA.

	Frequency(MHz)	Clock Load
Synchronous	41.748	1368
Async. Exponentiation	157.953	694
Async. Multiplication	45.335	674

Table 3 shows area and power quantitative data for the synchronous and GALS versions of RSA.

Table 3. FPGA area and power figures for RSA.

	LUTs	Flip Flops	Gates	Power (mW)
Synchronous	1521	1367	21294	27.07
GALS	1562	1367	21549	14.48

Power figures were obtained using the method proposed in [13] to compute FPGA power consumption. To enable applying the method, the prototyping board was modified by adding a precision resistor in series with the FPGA core power source. The synchronous RSA uses a 40MHz operating frequency, close to its maximum value. In the GALS RSA the MM module operates at 40MHz, while MX operates at 72MHz. This last value was chosen because it is slightly higher than the frequency where the performance of the GALS version equals the performance of the synchronous version. For higher frequencies, the GALS RSA presents better performance than the synchronous RSA. However, this performance gain is not significant, because 98% of the RSA execution time is spent executing modular multiplications. Results show that the GALS RSA incurs in 12% area overhead and a reduction of 46.5% in power, compared to the synchronous RSA version.

6 Conclusions

This paper proposed a new asynchronous interface to enable GALS design style in FPGAs which is flexible enough to allow interconnecting a mix of synchronous and QDI modules. SCAFFI relies on a hard macro library providing FPGAs with efficient, compact and low power asynchronous devices. The library development process is complex and each implementation contemplates only a single device size in a given device family. However, a simple set of asynchronous devices could be offered as part of FPGA vendor libraries, enabling a large set of non-synchronous design styles to be implemented on ordinary commercial FPGAs with little effort other than writing HDL code.

Devised future works include the detailed comparison of previously proposed interfaces with SCAFFI and an automated, parameterizable process for generating specific SCAFFI interfaces. Improving the flexibility of SCAFFI to support 1 of 4 encodings is under way. This encoding takes as many wires per bit as the dual rail encoding, but uses only half of the transitions to convey the same information transfer, increasing power efficiency. An asynchronous network on chip (NoC) based on SCAFFI is also under implementation.

7 References

[1] R. Ho et al. “The future of wires”. Proceedings of the IEEE, 89(4), Apr. 2001, pp. 490-504.

[2] M. Amde et al. “Asynchronous on-chip networks”. IEE Proc. Comp. and Dig. Tech., 152(2), Mar. 2005, pp. 273-283.

[3] D. Chapiro. “Globally Asynchronous Locally Synchronous Systems”. PhD Thesis, Stanford University, 1984.

[4] Q. Ho et al. “Implementing asynchronous circuits on LUT based FPGAs”. In: 12th FPL, 2002, pp. 36-45.

[5] S. Moore, P. Robinson. “Rapid Prototyping of Self-timed Circuits”. In: 17th ICCD, 1998, pp. 360-365.

[6] M. Najibi et al. “Prototyping Globally Asynchronous Locally Synchronous Circuits on Commercial Synchronous FPGAs”. In: 16th RSP, 2005, pp. 63-69.

[7] A. Martin, M. Nystrom. “Asynchronous techniques for system-on-chip design”. Proc. of the IEEE, 94(6), Jun. 2006.

[8] R. Ginosar. “Fourteen Ways to Fool Your Synchronizer”. In: 9th ASYNC, 2003, pp. 89-96.

[9] J. Muttersbach et al. “Practical Design of Globally-Asynchronous Locally-Synchronous Systems”. In: 6th ASYNC, 2000, pp. 52-59.

[10] S. Moore et al. “Point to Point GALS Interconnect”. In: 8th ASYNC, 2002, pp. 69-75.

[11] M. Singh, S.M. Nowick. “Synthesis for Logical Initializability of Synchronous Finite-State Machines”. IEEE Trans. on VLSI Systems, 8(5), Oct. 2000, pp. 542-557.

[12] J. Martín-Langerwerf et al. “Benefits of Macro-based Multi-FPGA Partitioning for Video Processing Applications”. In: 13th RSP, 2002, pp. 60-65.

[13] J. Becker, et al. “Power estimation and power measurement of Xilinx Virtex FPGAs: trade-offs and limitations”. In: 16th SBCCI, 2003, pp. 283- 288.