A Novel Interconnection Approach for Globally Asynchronous Locally Synchronous Circuits

Wolfgang Forster
Vienna University of Technology
Distributed Systems Group E184-1
Argentinierstr. 8, 1040 Vienna, Austria
w.forster@infosys.tuwien.ac.at

Eric Armengaud
Vienna University of Technology
Embedded Computing Systems Group E182-2
Treitlstr. 3, 1040 Vienna, Austria
armengaud@ecs.tuwien.ac.at

Abstract

This paper introduces a new methodology to solve the interfacing problem in Globally Asynchronous Locally Synchronous *(GALS) design approaches. We present a generic high-speed and delay-insensitive connector based on asynchronous* four state logic *(FSL). The advantages of this approach are following: First, it provides flexibility in the time domain since the data transfer is based on local handshakes and does not depend anymore on a global clock signal. Consequently, it removes timing constraints and even enables local optimizations. Second, this approach only requires a small number of interfacing signals, thus reducing the routing resources needed between data source and sink. Furthermore, the proposed architecture does not require customized delay lines and thus suits well for both ASIC and FPGA platforms. A modified FlexRay bus analyzer tool has been used to illustrate the advantages of our approach: High-speed and delay-insensitive data communication between different clock domains.*

1 Introduction

System-on-chip (SoC) is state-of-the-art in today's digital design technologies. Several functional modules from different vendors — so called intellectual property (IP) module — are integrated to a whole system into a single ASIC or FPGA. This trend is similar to purchase several IC's and integrate them on a printed circuit board (PCB) — just one level lower. In both cases one of the most challenging problems is the interconnection of the independent modules. Hence, the different interface specifications require translators, the different clock domains require synchronization stages, and consequently the rapidly growing number of integrated modules are requiring a rapidly growing number of individual solutions to properly handle the communication.

Globally Asynchronous Locally Synchronous (GALS) design techniques [2, 8, 11] have been introduced in this context. Those enable the integration of several synchronous islands which are communicating using asynchronous connectors. This approach combines the advantages of both design methodologies: Each module is a synchronous module on its own that can be efficiently synthesized with existing design tools and design flows, while asynchronous communication techniques [6] are used for the long distance communication between the individual clock domains. The synchronization problem at the interface of the IP modules is usually solved by a two-stage synchronizer. An alternative approach to minimize the transfer duration is to stretch the clock of the interfacing clock domains. The latter approach has been already discussed in detail in [12, 13, 16] and requires the following system properties on each clock domain: (a) The clock must be generated from an internal delay line, and (b) it must be possible to stretch the clock cycle — so called a gated clock. Both requirements cannot be fulfilled if a module in the system depends on clock quality properties, like clock drift rate or duty cycle. Furthermore, the process of system constraining, synthesis, and place and route for FPGA targets is complex since customized gates such as delay lines are required.

With these limitations in mind, we propose a novel methodology based on the principle of asynchronous pipelines for the asynchronous communication between synchronous modules. Our motivation is to provide a generic connector with a simple and flexible interface in order to minimize the connection effort between any two given synchronous modules. Moreover, our approach is independent from the different clock domains and provides a delay-insensitive communication scheme, thus removing any timing constraints and enabling high-speed intra- and inter-chip communication. Furthermore, we limit the number of signals that have to be routed over the chip, thus saving routing resources. Finally, we propose an architecture that can be easily implemented for both ASIC and FPGA platforms and does not need dedicated gates such as customized delay lines.

The theoretical discussion is illustrated with a practical use case. We will show how our existing FlexRay bus analyzer [7] can be enhanced with the SPEAR microcontroller [4], using our asynchronous connector for data exchange. The motivation for this enhancement is to give our bus analyzer the capability to compute *online* complex scenarios and more especially to interact with the distributed FlexRay clock synchronization mechanism [1].

This document is organized as follows: Section 2 provides the basics on asynchronous logic design. The novel concept for asynchronous interaction in GALS systems is introduced in Section 3. Section 4 describes the architecture of our tester node. The evaluation of our solution is presented in Section 5. Section 6 concludes this work.

2 Asynchronous logic design

2.1 Clock-gating schemes for GALS

Nowadays, almost every digital system is built based on synchronous design techniques. All activities are triggered by one (or a few) global control signal — a so called global clock. The designer can concentrate on the logic function of the application and automatic tools determine the maximum acceptable clock frequency. Hardware components in synchronous schemes are developed to fulfill certain timing requirements and thus designed to work with a specific clock frequency. The interaction of such components results in synchronization problems, which become very extensive in case of arbitrary clock frequencies between the different components.

The main concern of the Globally Asynchronous Locally Synchronous (GALS) approach is to solve the synchronization problem between different locally synchronous modules. In accordance to the work of [2, 8, 12, 13] we have identified the following requirements:

1. Delay-insensitive interconnection between subsystems to support long distance communication in SoC architectures

2. Simple interfaces between a synchronous sub-system and the asynchronous interconnection logic to avoid additional interface wrappers

3. Minimization of the number of interconnection signals in order to reduce the synthesis effort and to save routing resources

4. Avoidance of metastability during data takeover from one clock domain to another one

Existing GALS approaches are an adequate solution to meet the first and second requirement. The third requirement highlights the trade-off between data transfer rate and costs in terms of routing resources. Hence, increasing the number of interconnection signals improves the data transfer rate but requires more routing resources. The problem of metastability during data takeover is usually prevented by stretching the clock cycle of both involved clocks. This gated-clock approach, however, cannot be applied for modules with high demands concerning clock accuracy and clock duty cycle.

A promising solution for optimizing the connectors are asynchronous design techniques, which are based on the principle of local handshakes between sink and source pairs. The absence of a global clock eliminates the problem of frequency based data synchronization and allows all components to operate at their specific clock frequency. It exist different realizations of asynchronous design principles: Delay-Insensitive (DI) [14], Quasi-Delay-Insensitive (QDI) [9], Speed-Independent (SI) and Self-Timed (ST) circuits [17]. Notice that asynchronous approaches are not mandatorily free from timing constraints. QDI, SI and ST approaches all require (local) timing assumptions and can fail if these assumptions are violated.

FSL logical state	φ_0	φ_1
LOW ('0')	(0,0)	(0,1)
HIGH ('1')	(1,1)	(1,0)

Table 1. FSL encoding scheme [5]

2.2 Four state logic

Delay-insensitive circuits seem to be the most suited for asynchronous connectors since they do not require timing information and thus provide the most flexibility on the time domain. We have focused here on Four State Logic (FSL) [5, 10], where the information when the data is consistent and valid is encoded in the data itself. The sink indicates the consumption of data to the source by changing the value of an acknowledge signal (FSL is defined as a two-phase-protocol).

In FSL a dual rail coding is used, which means that two signal rails are necessary to represent a logical '1' or '0'. This resource overhead is also used to add a phase information. A logical '1' or '0' has two representations, one in phase 0 (φ_0) and another one in phase 1 (φ_1). Table 1 illustrates this FSL encoding scheme. During data transmission a new bit is indicated by a change of the coding phase. Notice that exactly one transition is required to change from one phase to the other, independent from the data information. This removes the risk of glitches within the asynchronous circuit. If all bits of a data word are in φ_0, the whole data word is valid and consistent and can be consumed by a sink. The acknowledgement of the consumption triggers the source to change the phase to φ_1 and to transmit the new data word. Only when all bits of the data word have changed to φ_1 at the sink the new data can be consumed again. An example of such data waves is given in Figure 1.

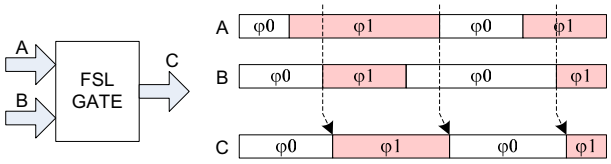

Figure 1. Flow of data-waves in FSL [5]

The most relevant problem of implementing FSL in state-of-the-art FPGA technologies is the increase of standard logic elements while increasing the size of a data width. A phase detector of a 16 bit vector needs to detect and compare the phase of each bit. Only when the phases of all bits are equal the data element is consistent and can be consumed. Consequently the resource consumption increases proportional to $n \cdot log(n)$ whereas n represents the size of the vector to decode.

3 The asynchronous connector

3.1 The connector interface

Figure 2 shows the basic concept of our asynchronous connector, which is separated into three major parts: The asynchronous source pipeline, the asynchronous sink pipeline and the serial connection, which could be reduced to solely three wires.

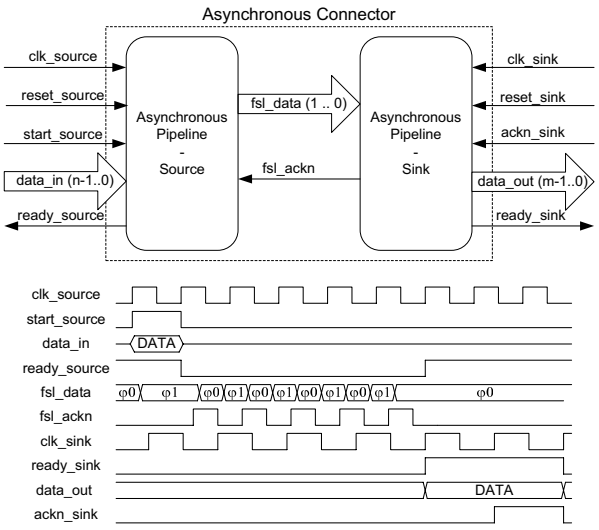

Figure 2. Asynchronous connector block diagram

The waveform illustrates the interface: Each communication partner has to provide the individual clock and reset signals. A data transfer can be initiated by setting *start_source* and providing the data to *data_in* for only one clock cycle of *clock_source*. The end of the transmission will be automatically signalled by *ready_source* until a new transfer is initiated. The corresponding receiver of the data transfer will be notified by *ready_sink*, while *data_out* will be valid as long as *ready_sink* is set. Setting *ackn_sink* for only one clock cycle on the receiver's side will reinitialize the sink pipeline for the next transfer. A two-stage synchronizer is used to synchronize *ready_source* and *ready_sink* to the corresponding clock domain.

Concerning the interface design, any arbitrary data width can be selected if *data_in* and *data_out* have the same dimension. Otherwise, the data width of *data_in* can be selected independently from the data width of *data_out* with the only restriction that the width of both must be an even number. Nevertheless it is possible to simply convert from one typical interface to another without additional design effort (e.g. from 32 bit to 16 bit or vice versa). The adaption of the interface width is implicitly handled by the *start_source* - *ready_source* and *ready_sink* - *ackn_sink* mechanisms.

3.2 Four-state-logic latch

An asynchronous pipeline is a simple serial composition of several Four State Logic (FSL) latches. This element is used to store the state of an asynchronous FSL coded bit — which is the counterpart to a register in synchronous circuits. The structure of such an FSL latch is shown in Figure 3. *Data_in* and *data_out* are the FSL-coded input and output signals of the latch. The information about the phase currently stored in the FSL latch is indicated via the *done* output. The *ackn* input represents the phase information of the following asynchronous element. Notice that for our asynchronous pipeline *data_out* of latch (*n*) is connected to *data_in* of latch (*n+1*) and the phase information *done* of latch (*n+1*) directly drives the control signal *ackn* of latch (*n*). Takeover of new data is

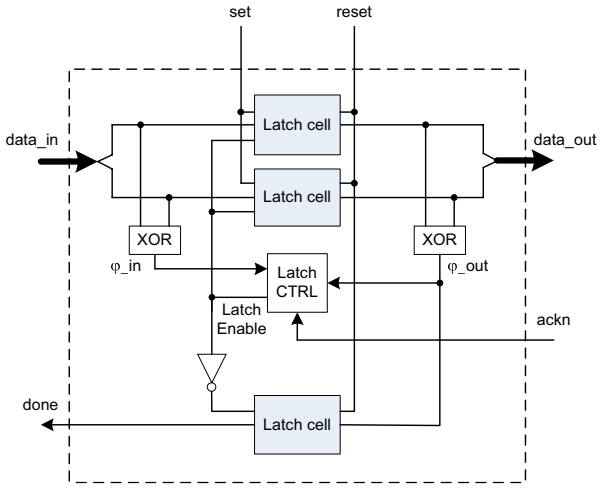

Figure 3. FSL latch [5]

based on the principle of consecutive source-sink pairs. New data can be stored in an FSL latch if two conditions are satisfied:

1. The actual stored data is different from the input data (φ_out unequal φ_in)

2. The consecutive asynchronous element has already consumed the actual data (φ_out equal *ackn*)

3.3 Asynchronous pipeline

The core function of our concept is the asynchronous pipeline — as used in our source and sink pipeline. An illustration of operation with four stages is presented in Figure 4 to picture the data flow.

At the top of the figure, *FSL Latch 0, 1, 2* have an alternative phase information while *FSL Latch 2* and *3* have the same phase information. This indicates that latch 3 has already took over the data from latch 2. Since new data is provided on the input of *FSL Latch 2* — this is indicated by phase information φ_1 — the latch condition for *FSL Latch 2* is fulfilled and the data from *FSL Latch 1* will be taken over. This is shown in the second part of the figure. Now, *FSL Latch 1* and *2* have the same phase information. Consequently, *FSL Latch 1* will take over the information of *FSL Latch 0*, and so on.

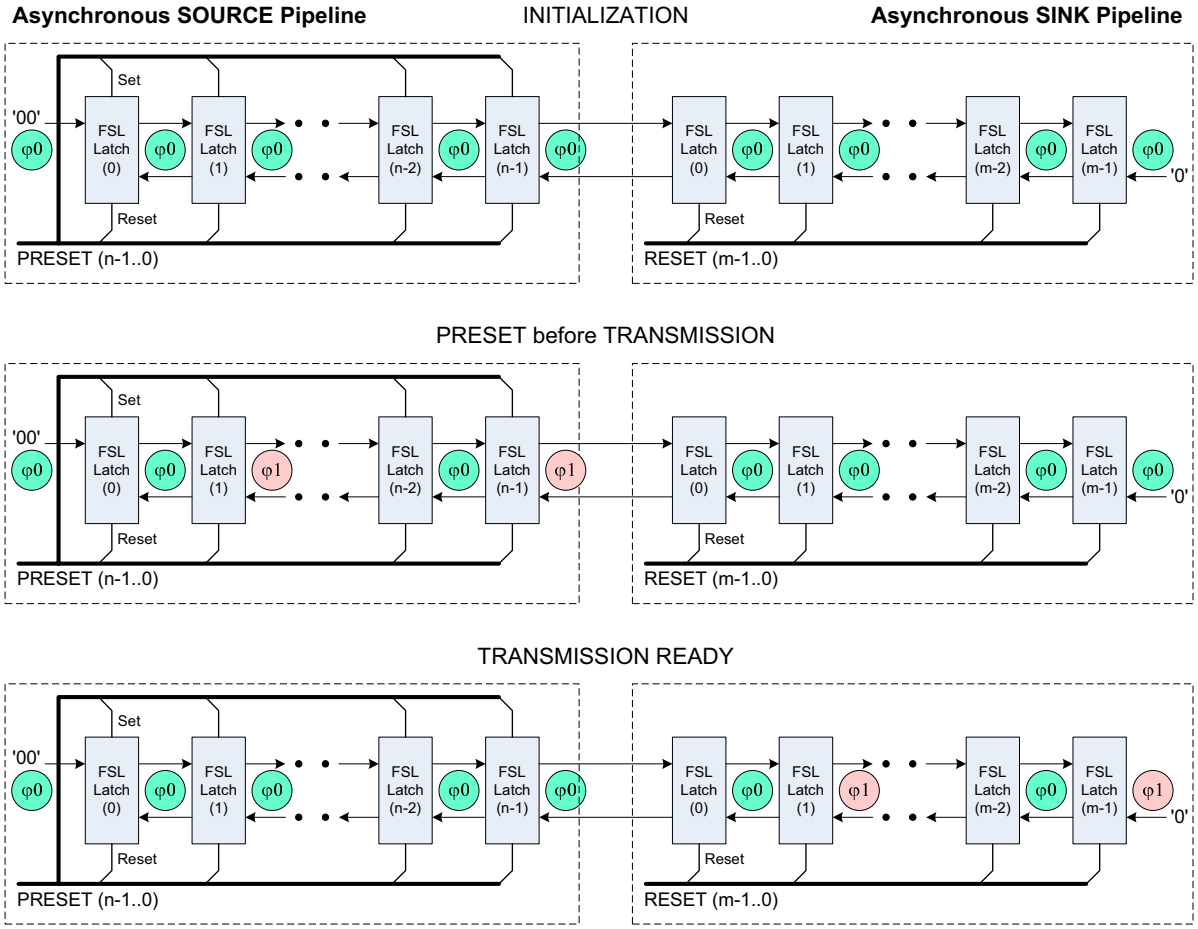

Figure 5. Asynchronous data transfer

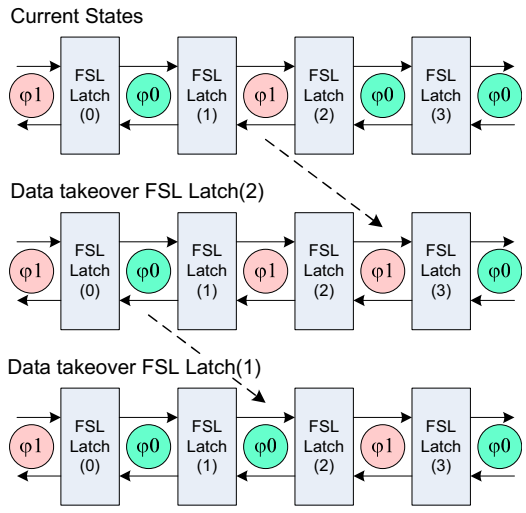

Figure 4. Elastic pipeline

This principle is called elastic pipeline [15] and is used as fundamental principle for our delay-insensitive asynchronous connector. Notice that system progression is triggered by the local handshake and does not rely on any timing assumption. Consequently, the effective asynchronous data transmission can be even faster than only one bit per cycle of source or sink clock in case of a favorable routing. Moreover, correct data transmission is also not invalidated in case of long or varying delays. Thus this mechanism is not only suited for intra-chip communication but for inter-chip communication as well, and more generally for systems where delays can not be guaranteed.

3.4 Initialization and data transfer

We have seen that system progression is based on the different FSL phases. Therefore, attention has to be paid to the initialization of the pipeline before each transfer. Figure 5 illustrates the data transfer flow between source and sink pipeline. During power-up both pipelines have to be initialized using *reset_source* and *reset_sink* (Figure 5, initialization). To initiate a data transfer the source pipeline is initialized with alternating FSL phase information (φ_1 - φ_0 - φ_1 - φ_0 - ..) and the corresponding data that was provided from the synchronous sub-system (Figure 5, preset before transmission). A prerequisite for the operation is that the first latch — *FSL Latch (n-1)* — is initialized with φ_1 and the whole sink pipeline is initialized

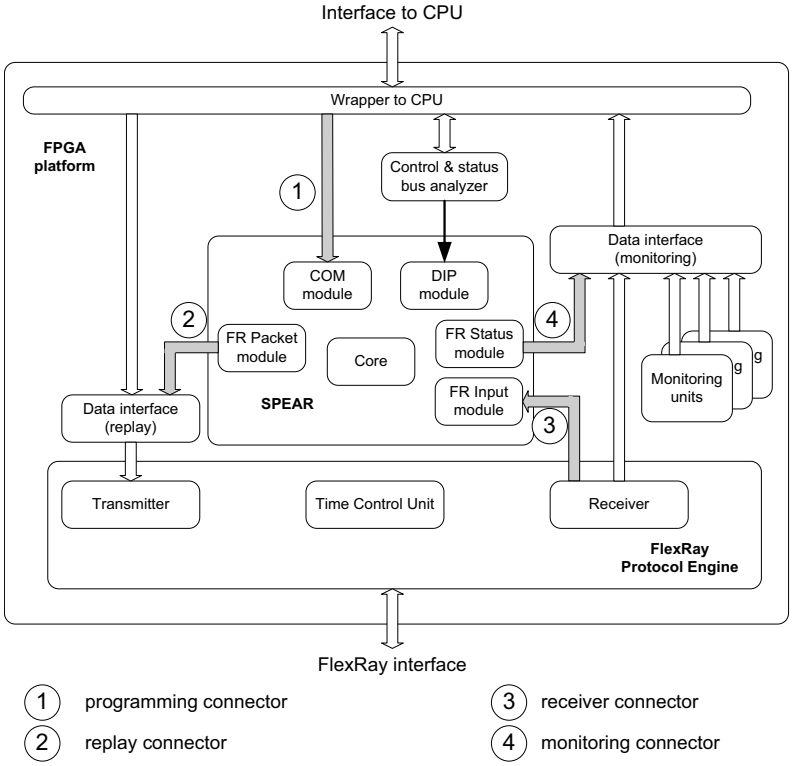

Figure 6. Tester node's architecture

with φ_0.

The alternating phases of the source pipeline will be transferred to the sink on the principle that has been already shown in Figure 4. The *ackn* signal of the last sink pipeline latch — *FSL Latch (m-1)* — is fixed to phase information φ_0, which stops the data transfer if the first phase φ_1 reaches this latch. The sink pipeline will recognize the transfer as completed if all sink latches have alternating phase information (starting with φ_1, Figure 5: transmission ready). If the transferred data has been consumed by the receiving synchronous sub-system the sink pipeline will be re-initialized — via *ackn_sink* — for the next transfer by setting all sink latches to φ_0. On the other hand, the source pipeline will recognize the completed transfer if all latches of the source pipeline have phase information φ_0. The next transfer can again be initiated by initialization of the source pipeline with alternating phase information.

3.5 Interface requirements

The physical connection between the source and the sink pipeline is using a delay-insensitive communication scheme. Nevertheless, to guarantee correct operation of our asynchronous connector a few local timing constraints have to be met:

Source pipeline preset: The interconnection delay between the source pipeline latches has to be less than one *clk_source* cycle to guarantee the correct preset of new data.

Sink pipeline reset: The interconnection delay between the sink pipeline latches has to be less than one *clk_sink* cycle to guarantee the correct re-initialization of the sink pipeline.

Sink pipeline ready: The interconnection delay between a sink pipeline latch and the corresponding *data_out(m-1..0)* port has to be less than one *clk_sink* cycle to guarantee the correct data takeover into the sink clock domain.

These requirements describe the interface between the asynchronous connector and the synchronous modules. It physically binds the source pipeline to the initiator of the transfer and the sink pipeline to the corresponding receiving sub-system. Notice that these requirements are local and related to the corresponding clock domain. There are no timing requirements between the different clock domains or for the asynchronous pipeline itself.

4 Tester node's system architecture

In order to illustrate the benefits of our approach, we have instantiated our asynchronous connector into a state-of-the-art FlexRay bus analyzer tool. This section provides an overview of the resulting system architecture (see Figure 6). It basically consists of different *monitoring units* delivering data from the different sources (e.g. CAN, LIN, digital or analog input). These data are ordered in the *data interface (monitoring)* and further sent to the host

through the *wrapper*. An opposite architecture (*data interface (replay)*) is provided for the transmission of data. The interface to the FlexRay bus uses a *FlexRay protocol engine*. More information concerning our prototype is available in [7].

The tester node enhancement is achieved by the integration of the SPEAR microcontroller for the processing of complex stimulus. The motivation for this IP is first to be platform independent (in contrary to Altera's NIOS[1] or Xilinx's Microblaze[2]). Moreover, we need real-time properties to guarantee the computation of the next stimulus iteration before the next communication cycle starts. This is provided by using an instruction set of constant execution time. Finally, we are in a prototyping phase and require a flexible, extendable computing unit. One main concept of SPEAR is the support of *Hardware Extension Modules*, which provides an environment to easily add new modules to this core. The integration is supported in hardware by well defined interfaces and in software by simple module access using memory mapping.

The SPEAR core requires four dedicated connections to our bus analyzer (in gray in Figure 6). The first one (*programming connector*) goes from the CPU to the COM module to load the program to execute. The second one, the *replay connector*, transfers the generated FlexRay packets together with a 25 ns accurate timestamp to the replay unit. This timestamp precisely defines the transmission time of the frame. The third dedicated interface, the *receiver connector*, is concerned with gathering the time differences between the tester node's time base and the received frames. This input is required to further compute the stimulus for the next communication cycle. Finally, the *monitoring connector* is used to notify the user from the SPEAR status.

In order to minimize the programming time of the SPEAR core the programming connector is implemented as a dual clock FIFO. The replay connector uses our asynchronous concept with a 16 bits to 16 bits interface. Since the packets to transmit are larger than 16 bits (between 128 bits and 2176 bits), several successive accesses are required. The receiver and monitoring connectors are implemented as 32 bits to 16 bits (respectively 16 bits to 16 bits) asynchronous connectors.

5 Evaluation

5.1 Setup

The quality of a connection can be principally measured in terms of data transfer rate versus resources required. Some other "soft skills" such as development and integration effort, or presence of (timing) constraints can be taken into account, too. The aim of this section is to compare our asynchronous connector to other synchronous approaches. To that aim, we have developed a serial and a parallel connector based on the synchronous development scheme. The three modules provide the same interface (see Section 3) in order to facilitate the comparison. The implementation of a serial and a parallel connector highlights the trade-off between high-performance parallel and resource saving serial interfaces.

The synchronous serial connector requires two control signals (data available and acknowledge) and one data signal for transmission. It requires a two-stage synchronizer for each control signal to properly cross the clock domains. Consequently, the transmission time for a N bit vector is:

$$T_{sync\_serial} = N \cdot [2 \cdot T_{C_{source}} + 2 \cdot T_{C_{sink}} + 2 \cdot T_{ic}] + T_{preset} + T_{readout} \quad (1)$$

$T_{C_{source}}$ and $T_{C_{sink}}$ are the clock periods from source and sink. T_{ic} is the interconnect delay between these two clock domains, and finally T_{preset} and $T_{readout}$ represents the additional delay to store (respectively readout) the data in the connector. The term $[2 \cdot T_{C_{source}} + 2 \cdot T_{C_{sink}} + 2 \cdot T_{ic}]$ represents the duration to transmit a bit from one clock domain to another. Two clock periods of each clock domain are required to synchronize the control signals (data available and acknowledge). Additionally the interconnect delay in both directions is necessary to transmit these control signals. The advantage of this serial connector is the low interconnection resource that is required (only three wires between source and sink part of the connector). Moreover, it provides a flexible interface definition since the source and sink part might implement different data width (the width of a transmitted signal can be adapted using the *start_source – ready_source* and *ready_sink – ackn_sink* control signals).

The synchronous parallel connector (equivalent to a dual clock FIFO) uses the same transmission scheme as the serial one with the one difference that N bits (instead of one) are transmitted in parallel. This naturally increases the required interconnect resources to $N+2$ wires and decreases the transmission time to:

$$T_{sync\_parallel} = 2 \cdot T_{C_{source}} + 2 \cdot T_{C_{sink}} + 2 \cdot T_{ic} + T_{preset} + T_{readout} \quad (2)$$

Our asynchronous serial interface combines the advantages of serial connection (low interconnect resources: only three interconnect wires) and asynchronous design (delay-insensitivity). The latter property is particularly interesting for interconnections. Hence, proper operation does not depend on any timing assumptions or any global clock signal. The transmission time only depends on the capacity of our elastic pipeline to transmit a bit, which in turn depends on the FSL latch delay (T_{cell}) and the maximum interconnect delay between two successive FSL latches (T_{mic}). The transmission time then is:

$$T_{async\_serial} = N \cdot [T_{cell} + T_{mic}] + T_{preset} + T_{readout} \quad (3)$$

In synchronous circuits the maximum clock frequency is determined by the longest path between any two registers of the whole module. In contrary, the transmission time of our asynchronous pipeline approach depends

[1] http://www.altera.com/
[2] http://www.xilinx.com

Connector	Asynchronous Serial				Synchronous Serial				Synchronous Parallel			
Data Width	LC	Duration	Signals	TR	LC	Duration	Signals	TR	LC	Duration	Signals	TR
[bit]	[]	[ns]	[]	[MBit/s]	[]	[ns]	[]	[MBit/s]	[]	[ns]	[]	[MBit/s]
8	131	177	3	**15.07**	49	1200	3	**2.22**	27	230	10	**3.48**
16	261	285	3	**18.71**	87	2300	3	**2.32**	43	230	18	**3.86**
32	519	505	3	**21.12**	162	4480	3	**2.38**	75	230	34	**4.09**
64	1036	880	3	**24.24**	312	9050	3	**2.36**	139	230	64	**4.35**

Table 2. Connector implementation properties

only on the effective path between two successive FSL latches. Consequently asynchronous transmission can be faster than one bit per clock period (as it is in serial synchronous schemes). Moreover, the pipeline architecture can be efficiently used by the place and route tool to minimize the average interconnect delay. Furthermore, the delay insensitivity provides higher robustness against long or variable delays. It makes then our approach very interesting for long distance intra-chip as well as inter-chip communication.

5.2 Results

This section deals with the comparison of our proposed asynchronous connector and the standard synchronous approaches previously described. We focus here on the resource usage and data transfer rate. For that, we have implemented the three connectors using an Altera CycloneII EP2C70 FPGA[3]. Each of these connectors has been designed with different interface data width (8, 16, 32 and 64 bits). We have identified the following properties as relevant measures for our comparison:

- Usage of FPGA logic cells of each connector (LC)
- Duration of the whole data transfer from source to sink (Duration)
- Number of interconnection signals (Signals)

Further we have introduced a new metric that considers these properties: *Transfer rate per interconnect signal* (TR). This metric represents the maximum number of bits per second that can be transferred via a single wire of a given connector. The unit of the TR is bit per second [Bit/s].

$$TR = \frac{Data\_width}{Duration \cdot Signals} \tag{4}$$

Table 2 shows the results of our evaluation regarding the different data widths and types of connectors. The clock frequencies of source and sink have been set to 30 and 66 MHz. The evaluation points out that our proposed asynchronous connector is the most effective concerning the transfer rate per interconnection signal.

Concerning the asynchronous connector, an increasing data width leads to an increasing TR because the pure asynchronous transmission gets more impact on the whole transmission. The delay overhead consisting of synchronizing the data at the interface is proportionally reduced with an increasing data width. Hence, the source and sink clocks only influence the preset and readout of the transferred data.

As expected, the synchronous parallel and serial connectors provide approximately the same TR. This can be explained by the communication scheme: the transmission of one bit is triggered by the clock signal and can not be locally optimized such as our asynchronous connector. Then, the principal difference between synchronous serial or parallel scheme is the number of wires used between source and sink. The number of bits transmitted per wire stays approximately the same. Notice that the delay overhead for a serial communication is higher since the control signals have to be activated for each bit and not only once for the whole transfer (parallel scheme). This explains the slight difference between parallel and serial TR.

The outstanding deficit of the asynchronous connector is the high number of logic cells required. This is caused by the implementation into a standard FPGA that does not provide optimized cells for FSL design. Nevertheless, if the usage of logic cells is taken into account in our comparison a very interesting fact becomes observable. Thus we have related TR to the corresponding number of logic cells (LC) for each connector, and this relation $\frac{TR}{LC}$ has always nearly the same value for each connector per dedicated data width. This highlights the fact that the three properties are tightly related, and the designer has to decide which attribute is more relevant for its implementation. We actually believe that the actual trend is set to the minimization of transfer duration and number of interconnection signals, thus making our approach particularly interesting. The number of logic elements is not playing a significant role due to the fast increasing number of transistors per chip. Even low cost FPGAs are already available with an huge number of logic cells (e.g. 70k or 120k logic cells).

5.3 The FlexRay bus analyzer

The implementation of the asynchronous connector in our tester node has permitted the efficient connection between the SPEAR core and the bus analyzer. First, efficient in term of integration and testing effort since the different clock domains and data width are naturally supported by our connector. Second, efficient in terms of interconnect resources and on resulting timings. We could not notice any worsening of maximal frequency achievable. Finally, efficient in terms of data transfer rate since the requested data arrived on time.

[3]http://www.altera.com

6 Conclusion

The decision of using a technology or a design method instead of another one mainly depends on the parameter(s) we want to optimize. In case of connectors, the focus is usually set to the transmission quality. This term addresses transmission rate and also resource requirements. Additional (non measurable) parameters such as presence of timing constraints or flexibility in the interface implementation can be taken into account, too.

In this paper we have presented an asynchronous, delay-insensitive approach for connecting two independent synchronous modules. The main advantages of this approach are (i) the maximization of the data transfer rate per interconnect wire and (ii) the absence of timing constraints with respect to the clock domains. The proposed asynchronous connector thus well suits both for high-speed intra-chip communication and for inter-chip communication.

Future work is going towards the development of a generic configurable asynchronous connector that supports both serial and parallel transmission. This mixed approach should provide a new configuration opportunity for a system designer to adjust the trade-off between data transfer rate and resource requirements.

7 Acknowledgements

This work has been partially funded by the FIT-IT [embedded systems initiative of the Austrian Federal Ministry of Transport, Innovation, and Technology] and managed by Eutema and the Austrian Research Agency FFG within projects COMPASS [3] (contract 809444) and ExTraCT (contract 810834).

We would especially like to thank Martin Delvai and Peter Tummeltshammer for valuable support and interesting discussions during integration of the SPEAR microcontroller.

References

[1] Flexray Communications Systems – Protocol Specification Version 2.1, available at http://www.flexray.com. FlexRay Consortium, 2005.

[2] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[3] COMPASS. *Component Based Automotive System Software*. http://www.infosys.tuwien.ac.at/compass.

[4] Martin Delvai. SPEAR Handbuch. Technical report, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2002.

[5] Martin Delvai and Andreas Steininger. Asynchronous logic design - from concepts to implementation. In *The 3rd International Conference on Cybernetics and Information Technologies, Systems and Applications*, pages 81–86, 2006.

[6] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[7] M. Horauer, F. Rothensteiner, M. Zauner, E. Armengaud, A. Steininger, H. Friedl, and R. Pallierer. An FPGA based SoC Design for Testing Embedded Automotive Communication Systems employing the FlexRay Protocol. In *Proceedings of the Austrochip 2004 Conference*, pages 119–125, September 2004.

[8] Xin Jia and Ranga Vemuri. Using GALS architecture to reduce the impact of long wire delay on FPGA performance. In Ting-Ao Tang, editor, *ASP-DAC*, pages 1260–1263. ACM Press, 2005.

[9] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. Technical report, California Institute of Technology, 1990.

[10] Anthony J. McAuley. Four state asynchronous architectures. *IEEE Trans. Computers*, 41(2):129–142, 1992.

[11] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 52–59, April 2000.

[12] Jens Muttersbach, Thomas Villiger, Hubert Kaeslin, Norbert Felber, and Wolfgang Fichtner. Globally-asynchronous locally-synchronous architectures to simplify the design of on-CHIP systems. In *Proc. 12th International ASIC/SOC Conference*, pages 317–321, September 1999.

[13] Mehrdad Najibi, Kamran Saleh, Mohsen Naderi, Hossein Pedram, and Mehdi Sedighi. Prototyping globally asynchronous locally synchronous circuits on commercial synchronous FPGAs. In *IEEE International Workshop on Rapid System Prototyping*, pages 63–69. IEEE Computer Society, 2005.

[14] Jens Sparso and Steve Furber. *Principles of Asynchronous Circuit Design - A Systems Perspective*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[15] I. E. Sutherland. Micropipelines (the turing award lecture). *Comm.A.C.M.*, 32(6):720–738, June 1989.

[16] George S. Taylor, Simon W. Moore, Robert D. Mullins, and Peter Robinson. Point to point GALS interconnect. In *ASYNC*, pages 69–75. IEEE Computer Society, 2002.

[17] Hiroaki Terada, Souichi Miyata, and Makoto Iwata. DDMP's: Self-timed super-pipelined data-driven multimedia processors. *Proceedings of the IEEE*, 87(2):282–296, February 1999.