Bi-Synchronous FIFO for Synchronous Circuit Communication Well Suited for Network-on-Chip in GALS Architectures

Ivan Miro-Panades, Alain Greiner

Bi-Synchronous FIFO for Synchronous Circuit Communication Well Suited for Network-on-Chip in GALS Architectures

Ivan MIRO PANADES
STMicroelectronics
17, avenue des Martyrs
38054 Grenoble, France
ivan.miro-panades@st.com

Alain GREINER
The University of Pierre and Marie Curie
4, Place Jussieu
75252 Paris, France
alain.greiner@lip6.fr

Abstract

The distribution of a synchronous clock in System-on-Chip (SoC) has become a problem, because of wire length and process variation. Novel approaches such as the Globally Asynchronous, Locally Synchronous try to solve this issue by partitioning the SoC into isolated synchronous islands. This paper describes the bi-synchronous FIFO used on the DSPIN Network-on-Chip capable to interface systems working with different clock signals (frequency and/or phase). Its interfaces are synchronous and its architecture is scalable and synthesizable in synchronous standard cells. The metastability situations and its latency are analyzed. Its throughput, maximum frequency, and area are evaluated in function of the FIFO depth.

1. Introduction

In deep sub-micron processes, the largest parts of the delays are related to the wires. In multi-million gates System-on-chip (SoC), achieving timing closure is difficult, as place & route tools have difficulty coping with long wires and balancing the clock tree.

The Globally Asynchronous, Locally Synchronous (GALS) [18,19] approach attempts to solve this problem by partitioning the SoC into isolated synchronous islands that have frequency and phase clock independency. With this approach, the timing constraints of the SoC can be bounded to the isochronous limit of each island. In this case, the communications between islands should be carried out by mixed-timing interfaces that adapt the clock frequency and phase discrepancy. Such interfaces are not trivial [7] since the synchronization failure (metastability) of the registers can corrupt the transferred data. Many architectures have been proposed to solve this issue, some of them restricted to study only skew [12, 13, 17], and jitter [17] correction. Others use pausible or stretchable clocks which pause [1, 6], or stretch [2, 3] the receiver clock to synchronize the data in a safe rising edge of the receiver clock.

A number of approaches interface systems with a rational clock relation [11, 14]. A recent published paper from The University of Pierre and Marie Curie proposes a novel synchronous↔asynchronous converter well suited for Network-on-Chip (NoC) in GALS architectures [20]. Also in [22] asynchronous to synchronous and synchronous to asynchronous interfaces for GALS NoC are implemented using Gray FIFOs (first-in, first-out). Finally, the most generic architecture where frequency and phase are unknown is resolved by robust designs where metastability is well analyzed. The basic architecture is the "two-flop synchronizer" [7] where only the request and acknowledge signals are synchronized. Fastest architectures use FIFOs [4] or a RAM bank [5], where write and read pointers are transferred between clock domains using the Gray code. J. Jex et al. [9] and Chelcea-Nowick [8] propose optimized architectures for mixed-timing systems. The J. Jex et al. solution was originally designed to be used on supercomputer interfaces. The solution can modify the number of registers in the synchronizer to increase the robustness against metastability, and is suited to high bandwidth communications since the control signals are optimized. The solution of Chelcea et al. [8] proposes a mixed-timing interface for mixed synchronous to synchronous, asynchronous to synchronous, and synchronous to asynchronous interfaces. Its architecture is modular and requires low area overhead. However, it requires full custom cells to implement the Full and Empty detectors, which becomes less attractive for synchronous standard cell flows.

The FIFO presented in this document is a bi-synchronous FIFO [10] able to interface two synchronous systems with independent clock frequencies and phases. A latency optimization of the architecture in which both interfaces have the same clock frequency but different clock phase (mesochronous) is also proposed. The main features of

the bi-synchronous FIFO are low latency, robustness to metastability, small area, scalability, maximized throughput, and synthesizability on synchronous standard cell flow. This bi-synchronous FIFO is been used on the DSPIN [21, 23] Network-on-Chip (NoC), where ten bi-synchronous FIFOs of 34 bits are used per router: Five have 8 words depth, and five have 4 words depth. Since a SoC can contain more than 30 routers, the area of this component have to be careful designed to minimize the total SoC area. Furthermore, the depth of the bi-synchronous FIFOs in this kind of NoC is no deeper than 10 words, therefore a register-base FIFO is more suited than a RAM-base FIFO due to its lower area and its simplicity of test. Thus, its test method can be the scan-chain method. Moreover, the bi-synchronous FIFO makes the DSPIN NoC well suited to the GALS approach as the DSPIN router communicates asynchronously with the local subsystem, and mesochronously with the neighbor routers.

The paper is organized as follows. Section 2 presents a novel encoding algorithm well suited to interface two clock systems. In Section 3, all the bi-synchronous FIFO modules are detailed. In Section 4, the analysis of latency, throughput, area, and maximum frequency are analyzed and compared with existing solutions.

2. Bubble encoding

In this section, a novel-encoding algorithm based on a token ring is demonstrated to be useful on the synchronization of pointers between two independent clock domains.

2.1. Token ring

A token ring is a succession of nodes interconnected in a circular manner that contain tokens. It can be described with N registers (with enable signal) interconnected in serial, like a cyclic shift-register. Figure 1 shows an example of a token ring with 5 registers.

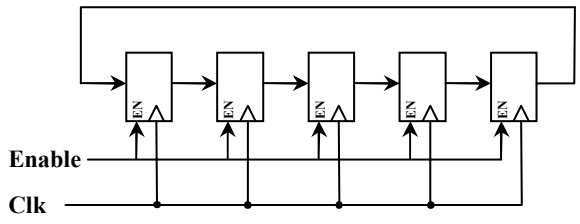

Figure 1. Token ring

If the enable signal is true, the content of the register is shifted (register *i* is shifted to register *i+1*, an register *N-1* to register *0*) at the rising edge of the clock, otherwise the register maintains the data. A token is represented by the logic state 1 of the register. The number of tokens in a token ring can be from one to N, where N is the number of registers.

A token ring with one token can be used as a state-machine. The position of the token defines the state of the state-machine. The enable signal of the token ring is equivalent to the next-state of the state-machine. If this condition is true, the token ring shifts and the state-machine changes. It is also possible to define a state-machine when the token ring contains two consecutives tokens, the state of the state-machine can be defined, for example, as the position of the first one.

2.2. Synchronizing the Token

Since the position of the tokens defines the state of the state-machine, the synchronization of the position can be exploited to interface the two clock domains. To synchronize the state of the state-machine, a parallel synchronizer (two registers per bit) can be used, as shown in Figure 2.

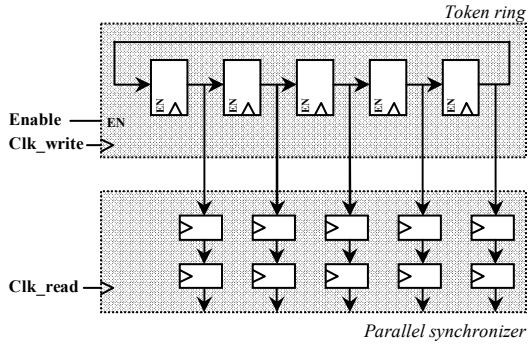

Figure 2. Synchronization of a token ring

However, as described by R. Ginosar, the parallel synchronizer [7] does not guarantee the correctness of the result. Figure 3 shows an example of synchronization. *Solution A*, *B*, *C* and *D* are all the possible solutions when the metastability of the register changes its content. *Solution A* and *B* are correct since the token is well determined. *Solution C* is exploitable using some logic but *Solution D* is useless due to absence of information. Because the token was moving, the metastability of the register can be resolved to a useless result. In that case, one clock cycle should be waited to attempt to obtain a useful data.

To solve this issue, we propose to use two consecutive tokens (bubble encoding) in the token ring. As the metastability affects the changing registers, the use of two consecutive tokens prevents some registers

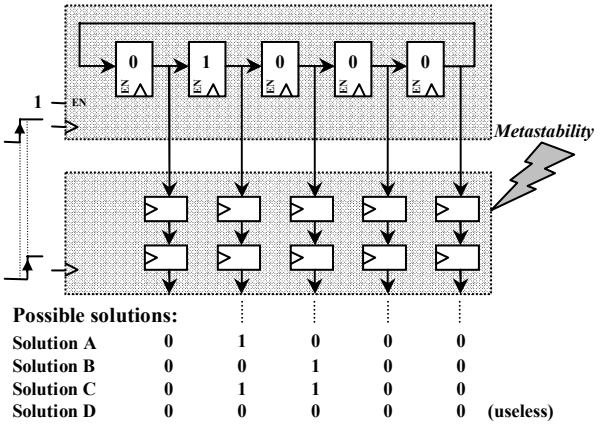

Figure 3. Possible solution in the synchronization of a single token ring containing one token

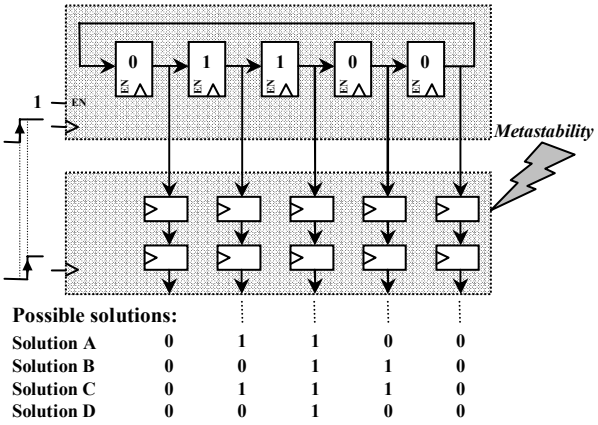

Figure 4. Possible solution in the synchronization of a token ring containing two successive tokens

from changing. Assuming that registers *i* and *i+1* have the tokens, if the token ring shifts, register *i+2* gets a token, register *i* loses its token, and register *i+1* does not change (it shifts its token and gets a token). In terms of logic value, register *i* and *i+2* change state but register *i+1* remains unchanged. Because there always exists a register that does not change state, it is always possible to detect a token. Figure 4 shows an example of synchronization. For example, we can define the position of the detected token by the position of the first logic 1 after a logic 0 (starting from the left). In this case, all *solutions A, B, C,* and *D* are correct because the token can be well defined; it is always possible to detect a transition between 0 and 1. This encoding algorithm does not avoid the metastability on the synchronizer. It just guarantees that the position of the token will be detected and a useless situation will never occur.

The token ring and the bubble encoding presented in this section are used on the definition of the state-machines of the bi-synchronous FIFO and will be detailed in next section.

3. Bi-synchronous FIFO

This section presents the architecture of the bi-synchronous FIFO and its application in multi-clock systems. The goal of this FIFO is to interface two synchronous systems having different clock signals. Each system is synchronous with its clock signal but can be asynchronous (frequency and/or phase) to the others. The challenge of this architecture is to hide all synchronization issues while respecting the FIFO protocol on each interface. Furthermore, this architecture is scalable and synthesizable in a synchronous standard-flow without using custom cells.

As shown in Figure 5, five modules compose the bi-synchronous FIFO architecture: *Write pointer*, *Read pointer*, *Data buffer*, *Full detector*, and *Empty* detector. The *Write* and *Read* pointers indicate the position to be written and to be read in the *Data buffer*, the *Data buffer* contains the buffered data of the FIFO, and the *Full* and *Empty* detectors signal the fullness and the emptiness of the FIFO.

To better understand the bi-synchronous FIFO, its interfaces and protocol are detailed.

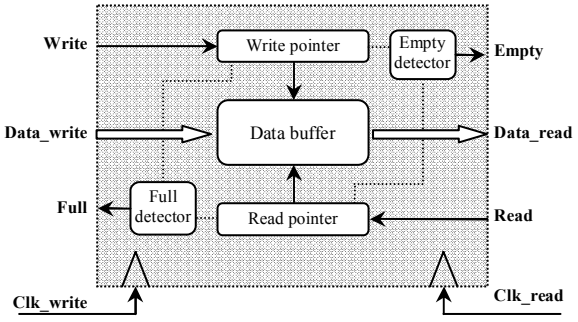

Figure 5. Bi-Synchronous FIFO architecture

3.1. Bi-synchronous FIFO interface and protocol

The bi-synchronous FIFO has a sender and a receiver interface. As shown in Table 1, each interface has its own clock signal, *Clk_write* for the sender and *Clk_read* for the receiver.

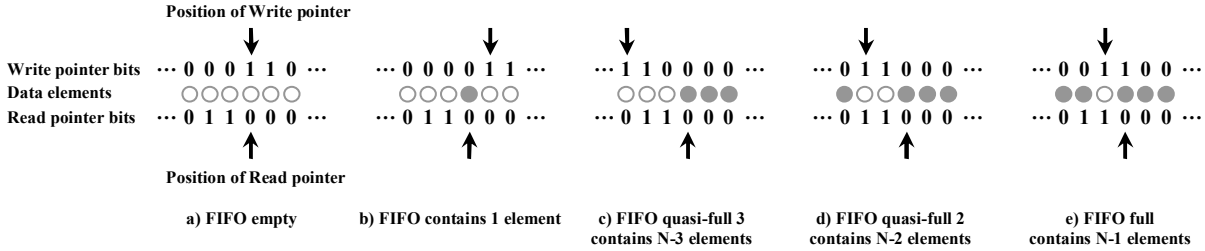

Figure 6. Write and Read pointer position definition and Full and Empty conditions in terms of tokens position

The FIFO protocol is synchronous; all input and output signals in the sender and receiver interfaces are synchronous to their clock signal *Clk_write* and *Clk_read,* respectively.

Table 1. Sender and receiver interface signals

	Signal	Description
Sender interface	Data_write	Data to be written into the FIFO
	Write	Input signal requesting a write into the FIFO
	Full	Output signal indicating the fullness of the FIFO
	Clk_write	Sender clock signal
Receiver interface	Data_read	Output data from the FIFO
	Read	Input signal requesting a read in the FIFO
	Empty	Output signal indicating the emptiness of the FIFO
	Clk_read	Receiver clock signal

The queuing and dequeuing of data elements in the FIFO follows the next synchronous protocol. The *Data_write* is queued into the FIFO, if and only if, the *Write* signal is true and the *Full* signal is false at the rising edge of *Clk_write*. Symmetrically, data is dequeued to *Data_read*, if and only if, the *Read* signal is true and the *Empty* signal is false at the rising edge of *Clk_read*.

The clear partitioning of the sender and receiver interfaces into synchronous and independent interfaces simplifies the timing constrains analysis for all the modules connected to the FIFO ports.

3.2. Write and read pointers

The *Write* and *Read* pointers are implemented using the described token rings with the bubble-encoding algorithm. The position of the tokens determines the position of the pointer. The position of the *Write_pointer* is defined by the position of the register *containing* the first token (starting from the left) as shown in Figure 6a. Likewise, the position of the *Read_pointer* is defined by the position of the register *after* the second token (starting from the left). The Full and Empty detectors exploit this particular definition of the pointers and will be explained hereinafter.

The *Write_pointer* shifts right when the FIFO is not full and the *Write* signal is true. Likewise, the *Read_pointer* shifts right when the FIFO is not empty and the *Read* signal is true.

As the write and read interfaces belong to different clock domains, the token rings are clocked by their clock signal, *Clk_write* and *Clk_read* respectively.

3.3. Data buffer

The *Data buffer* module is the storage unit of the FIFO. Its interfaces are: *Data_write*, *Data_read*, *Write_pointer*, *Read_pointer*, and *Clk_write*. It is composed by a collection of data-registers, AND gates, and tri-state buffers as shown in Figure 7.

The input data, *Data_write*, is stored into the data-register pointed by the *Write_pointer* at the rising edge of *Clk_write*. AND gates recode the *Write_pointer* into a one-hot encoding which controls the enable signals of the data-registers. Likewise, the *Read_pointer* is recoded into one-hot encoding which controls the tri-state buffer on each data-register. Finally, the *Data_read* signal collects the outputs of the tri-state buffers. It is also possible to replace the tri-state buffers with multiplexers to simplify the Design for Test (DfT) of the FIFO.

The width and number of data-registers determine the width and the depth of the FIFO. The depth also determines the range of the *Write* and *Read* pointers.

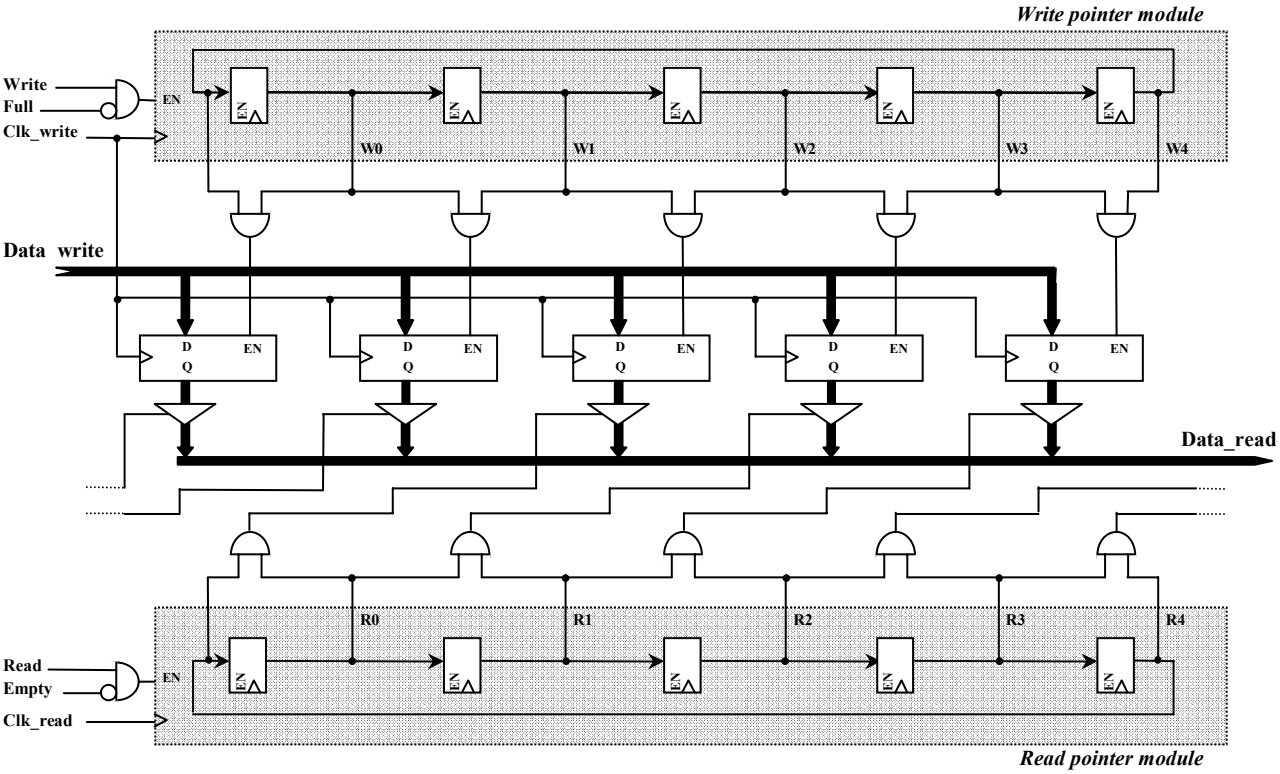

Figure 7. Write pointer, Read pointer, and Data buffer detail

3.4. Full detector

The Full detector computes the *Full* signal using the *Write_pointer* and *Read_pointer* contents. No status register is used as in the J. Jex et al. [9] or Chelcea-Nowick [8] solutions. The Full detector requires N two-input AND gates, one N-input OR gate, and one synchronizer, where N is the FIFO depth (Figure 8). The detector computes the logic AND operation between the *Write* and *Read* pointer and then collects it with an OR gate, obtaining logic value 1 if the FIFO is Full (Figure 6e) or quasi-Full (Figure 6c and 6d) otherwise is 0. This value is finally synchronized to the *Clk_write* clock domain into *Full_s*. Since the synchronization has a latency of one clock cycle and the situation on Figure 6c can potentially be metastable, the detector has to anticipate the detection of the Full condition. Is for this reason that the output of the OR gate detects the Full and the quasi-Full conditions.

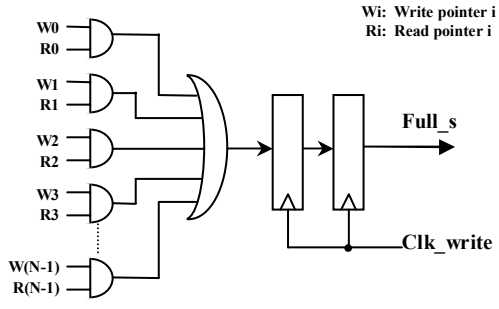

Figure 8. Full detector detail

The Full detector in Figure 8 can be optimized since the synchronization latency inhibits, in some cases, the FIFO from being completely filled. For example, if the FIFO is in the situation of Figure 6c, and the sender does not write any other data, the *Full_s* will be asserted even if the FIFO is not filled completely. As the fullness has to be anticipated, the detector is dimensioned to stop the sender before overflowing the FIFO, if the sender was writing continuously.

An improved Full detector implementation would be more complex (as the Empty detector), and would therefore require greater die area. However, a non-optimal Full detector does not penalize the throughput of the FIFO as much as a non-optimal Empty detector. For example, assuming an optimal Empty detector and a non-optimal Full detector, the Full condition occurs when the receiver is not able to consume all the data. In this case, even with a non-optimal Full detector, the receiver limits the throughput of the FIFO. Therefore, design effort and chip area should be devoted to improving the performance of the Empty detector.

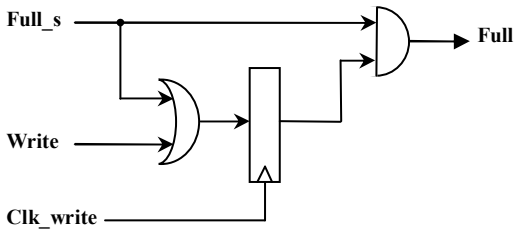

Figure 9. Full detector optimization

Even when using a non-optimized Full detector, a low cost optimization can improve its performance. Figure 9 shows an additional module connected to the *Full_s* signal, which improves the Full detector. The module's operation is as follows: if the writer was not writing before asserting the *Full_s* signal, the *Full* signal is delayed one clock cycle, giving a second chance to the writer to fill completely the FIFO.

3.5. Empty detector

The implementation of the Empty detector is similar to the Full detector because both employ the *Write* and *Read* pointer contents. As seen in the previous paragraph, the Full detector has to anticipate the detection of the Full condition to avoid FIFO overflow. As the Empty detector is correlated to the FIFO throughput, its detection has to be optimized, and no anticipation detector should be used.

Figure 10 shows the Empty detector for a five word FIFO. First, the *Write_pointer* is synchronized with the read clock into the *Synchronized_Write_pointer* (SW) using a parallel synchronizer. Next, the *Read_pointer* is recoded into the *AND_Read_pointer* (AR) using two-input AND gates, operation also done in the *Data_buffer* module. The output of AR is a one-hot encoded version of the *Read_pointer*. Finally, the Empty condition is detected comparing the SW and AR values using three-input AND gates. As the metastability can perturb some bits of the SW (as seen on Figure 4), each pair of consecutive bits is compared to find a transition between 0 and 1. Their analysis is as

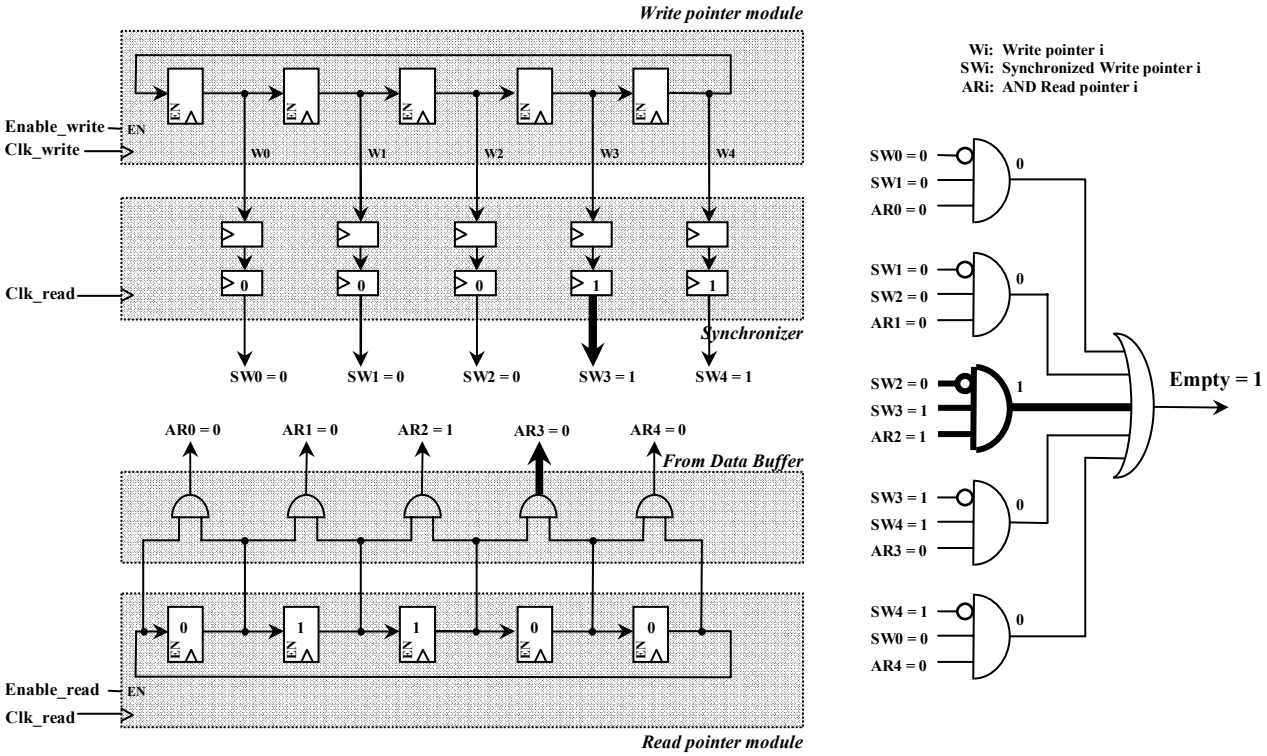

Figure 10. Empty detector detail

follows, if the values of $SW_i = 0$ and $SW_{i+1} = 1$ that means that the SW pointer is on position *i+1*. Furthermore, when $AR_i = 1$ that means that the AR pointer is on position *i+1* (Figure 10).

The FIFO is considered empty (see Figure 6a) when the *Write_pointer* points the same position of the *Read_pointer*. This can be detected when SW_i=0, SW_{i+1}=1 and AR_i=1 for any *i*. These comparisons are computed by means of the three-input AND gates. Finally, a N-input OR gate collects all the values of the three-input AND gates to generate the Empty signal. This N-input OR gate and the one on the Full detector can be decomposed with $\log_2 N$ levels of two-input OR gates.

The latency introduced by the synchronization of the *Write_pointer* cannot corrupt the FIFO, because a change in this pointer cannot underflow/overflow the FIFO, it just introduces latency into the detector.

The advantage of the bubble-encoding algorithm in this detector relies on the continuous detection of the *Write_pointer* position. Otherwise, as seen on Figure 3 Solution D, its position cannot be detected and the Empty condition should be asserted to avoid a possible underflow of the FIFO, thus, introducing one additional clock cycle latency to the FIFO.

3.6. Mesochronous adaptation

The FIFO architecture was originally designed to interface two independent clock domains, but can be adapted to interface mesochronous clock domains where the sender and the receiver have the same clock frequency but different phase. The difference of phase can be constant or slowly varying. We find examples interfacing mesochronous clock domains that employ the predictability of the rising edges to avoid the metastability situations [15, 16]. The proposed adaptation lowers the FIFO latency by reducing the number of registers on the synchronizer module. Since metastability can be avoided when the rising edges of the clock signals are predictable, the two rows of registers on the synchronizer can be reduced to a single row of registers as shown in Figure 11b. The remaining row of registers is clocked using a delayed version of the read clock. This delay must be chosen to exchange the data without metastable situations (Figure 11a). The delay can be a programmable delay, or any other metastability-free solution, as for example the Chakraborty-Greenstreet [11] architecture allowing the FIFO to work also on plesiochronous (small difference of frequency) clocks. Likewise, if the write and read clock are out of phase by 180º (clock-inverter), no

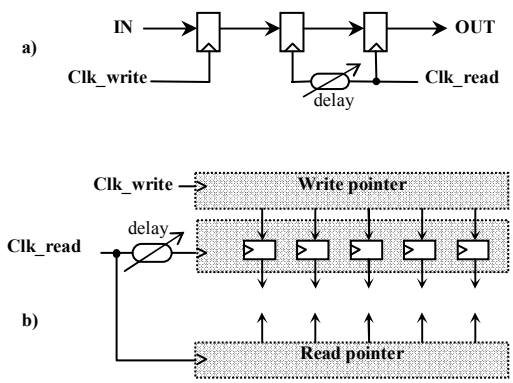

Figure 11. Mesochronous adaptation

programmable delay is needed because, by-construction, the communication is free of metastability. The interface is free of metastability also if the difference of phase varies between 90° and 270°. This type of implementation is done on the DSPIN NoC routers. The clock-tree of each DSPIN router is balanced with a 5% skew. Then, in the routers placed on the Y row and X colon position, where (Y+X) is an odd number, the first clock-buffer of the clock-tree have been replaced by a clock-inverter. Finally, the top pins of the clock-trees routers are balanced with a 50% skew, which is easier to design and less power consuming than a fully synchronous clock tree balance. With this procedure, neighbor routers are 180° out of phase with a tolerance of 50% skew.

This mesochronous adaptation of the bi-synchronous FIFO is simple and allows switching between mesochronous and asynchronous modes. This adaptation is interesting in the design of a multi-million gate SoC in deep sub-micron technology, where the delay of long wires can drastically vary with temperature, voltage, and process. In such a system, the mesochronous clock distribution could fluctuate to an undesirable metastable situation, making the FIFO data useless. By switching the bi-synchronous FIFO into the asynchronous mode, robustness against metastability is improved, preventing the SoC from requiring redesign.

4. Simulation and analysis

Both synthesizable VHDL models and cycle accurate SystemC models of the bi-synchronous have been designed. We have simulated the bi-synchronous FIFO to characterize its latency, throughput, frequency, and area.

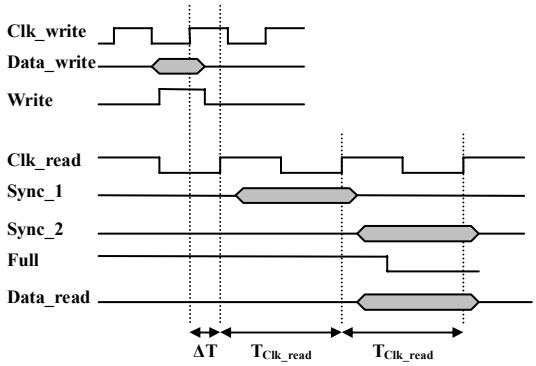

Figure 12. Latency analysis

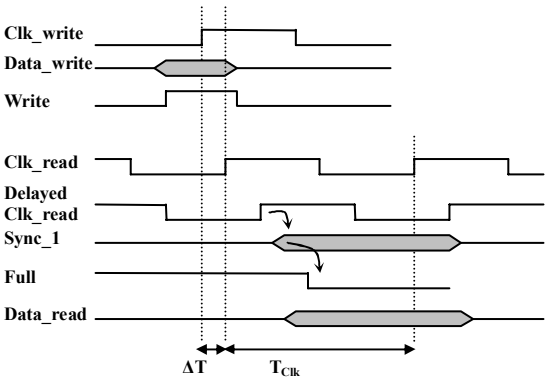

Figure 13. Latency analysis with mesochronous adaptation

4.1. Latency analysis

As the sender and the receiver have different clock signals, the latency of the FIFO depend on the relation between these two signals.

The latency of the FIFO can be decomposed in two parts: the state machine latency and the synchronization latency. As the state-machines are designed using Moore automates, its latency is one clock cycle. Two registers compose the synchronizers and its latency is ΔT plus one clock cycle. Where ΔT is the difference, in time, between the rising edges of sender and receiver clock. As this difference is between zero and one *Clk_read* clock cycle, the latency of the bi-synchronous FIFO is between two and three *Clk_read* clock cycles. Figure 12 shows the detail of the latency. Sync_1 and Sync_2 are the synchronization registers. The latency of the bi-synchronous FIFO is equivalent to the latency of the J. Jex et al. [9] solution. This latency can be lower, but the robustness to the metastability would be penalized [24, 7].

When the bi-synchronous FIFO is adapted to a mesochronous clock distribution, the latency of the FIFO is reduced, because a single register replaces the two-register synchronizer. In addition, the ΔT is constant as the difference of phase is constant. In that case, the latency of the FIFO is one clock cycle plus ΔT, as shown in Figure 13.

4.2. Throughput Analysis

The throughput of the bi-synchronous FIFO was analyzed in function of the FIFO depth. As the synchronizers add latency, the flow control of the FIFO is penalized and its performances are influenced. In case of deep FIFO, those latencies do not decrease the FIFO throughput since the buffered data compensate the latency of the flow control. Table 2 shows the minimum FIFO depth for 50% and 100% throughput in function of the clock relation. For FIFO depth of 6 or above, the synchronization latency has no influence on the flow control and the FIFO is able to deliver one word per cycle (100% throughput) even on asynchronous clock relation. For the asynchronous analysis, the write and read clock signals frequencies are similar, otherwise it is not possible to obtain 100% throughput.

Table 2. Minimum FIFO depth in function of the clock relation and required throughput

	Minimum depth for 50 % throughput	Minimum depth for 100 % throughput
Asynchronous	5	6
Mesochronous	4	5

4.3. Area and frequency estimation

The area and frequency estimation of the FIFO was computed once synthesized on CMOS 90nm GPLVT STMicroelectronics standard cells. Different FIFO depths are used to illustrate the scalability of the architecture and its performances in terms of maximum frequency. To minimize the power consumption, a clock gating technique is used. Two architectures were synthesized, one with the tri-state buffers and another with multiplexers.

Table 3 shows the area and frequency estimation of a 32-bit bi-synchronous FIFO in function of the FIFO depth and type of output port. Note that the maximum frequency of the *write* clock is greater than the one of the *read* clock. The limitation of the *read* clock is due to the Empty detector.

The architecture with tri-state buffers has greater area than the one with multiplexers. This phenomenon

is due to the large area of the tri-state buffers. Moreover, the maximum clock frequency of the read part with tri-states is greater than the one with multiplexers, since the multiplexers are decoded in a $\log_2 N$ manner rather than in parallel.

Table 3. Area and frequency in function of FIFO depth and type of output port

Type	FIFO Depth	Area (μm^2)	Max. Write Freq. (MHz)	Max. Read Freq. (MHz)
Mux	4	3304	2000	1110
	8	6581	2000	1000
	16	13384	2000	769
Tri-state	4	4082	2000	1428
	8	8032	2000	1250
	16	16101	2000	1110

4.4. Comparison with other existing designs

This architecture has been compared with similar architectures to analyze its area and latency. As its architecture is synthesizable with standard cells, the comparison with the others is done after Synopsys synthesis with the same timing constraints.

The selected architectures are a register-base Gray FIFO and the J. Jex et al. [9]. Their VHDL description has been written for different FIFO depths: 4, 8 and 16 words of 32bits. A clock-gating technique is applied but no tri-state buffer is used. Table 4 shows the estimated area and the area overhead percentage of these architectures compared to the presented solution.

Table 4. Area and overhead comparison between other existing designs

FIFO Depth	This Design μm^2	Register-base Gray FIFO μm^2 (%)	J. Jex et al. [9] μm^2 (%)
4	3304	5113 (+54%)	3364 (+1.8%)
8	6581	9702 (+47%)	6858 (+4.2%)
16	13384	20364 (+52%)	14362 (+7.3%)

The register-base Gray FIFO has a 50% bigger area than the presented architecture. Even if the number of registers and synchronizers is lower than our architecture, the Gray code algorithm adds complexity to the read and write state machines. Nevertheless, the Full and Empty detectors are fully optimized, as the write and read sides know the exact position of the read and write pointer (after synchronization).

The J. Jex et al. [9] architecture has similar complexity as ours, but its area increases more than ours when the FIFO depth increases. Moreover, its Full detector is not optimized and suffers the same problem of the non-optimized Full detector presented in Figure 9. To correct this issue, the optimization of Section 3.4 could be used, regardless the increase of the total area.

In terms of FIFO latency, all three have the same latency, 2-3 clock cycles, since all of them use Moore state-machines and two flip-flops synchronizers.

5. Conclusions

A new bi-synchronous FIFO has been presented and analyzed. It is well suited to interface different systems working with independent frequency and/or phase clock signals.

It uses a novel encoding algorithm combined with an astute definition of the FIFO pointers that avoids the utilization of status registers. Its *write* and *read* pointers are directly combined to obtain the Full and Empty detectors.

Both of its interfaces are synchronous to its relative clock signals. Moreover, its architecture is designed to be synthesized using a synchronous standard cell design flow. None of its modules requires custom cells.

A simple mesochronous adaptation is proposed which reduces the latency of the FIFO. Its latency is 2-3 clock cycles in asynchronous mode, and 1-2 clock cycles in mesochronous mode.

Both SystemC cycle accurate and VHDL models of the bi-synchronous FIFO has been designed. The FIFO throughput depends on the FIFO depth. Throughput is 100% when the FIFO depth is six or above.

Using CMOS 90nm GPLVT STMicroelectronics standard cells, we have synthesized and analyzed the FIFO area and maximum frequency for different FIFO depths. Two architectures are analyzed, one with tri-state buffers and another with multiplexers. A 32-bit bi-synchronous FIFO with eight words depth requires $6581 \mu m^2$ and its maximum clock frequency is 1GHz.

The comparison with existent synthesizable asynchronous FIFOs shows a better integration density at the same data latency.

References

[1] A. E. Sjogren and C. J. Myers, "Interfacing synchronous and asynchronous modules within a high-speed pipeline," in IEEE Trans. VLSI Syst., Vol 8, no. 5, pp 573-583, Oct. 2000.

[2] S.W. Moore, G.S. Taylor, P. A. Cunningham, R.D. Mullins, and P. Robinson, "Self calibrating clocks for globally asynchronous locally synchronous systems," in IEEE Proc. Int. Conf. on Computer Design, 2000.

[3] P. Zipf, H. Hinkelmann, A. Ashraf, and M. Glesner, "A switch architecture and signal synchronization for GALS System-on-Chip," in Proc. of 17th Symposium

on Integrated Circuits and Systems Design (SBCCI2004), pages 210-215, 2004.

[4] N. Rougnon Glasson, "Device for transferring data between two asynchronous subsystems having a buffer memory," Patent US2004230723, Nov. 2004.

[5] G. N. Pham and K. C. Schmitt, "A high throughput, asynchronous, dual port FIFO memory implemented in ASIC technology," in Proc. Annual IEEE Int. ASIC Conf. and Exhibition, 1989.

[6] R. R. Dobkin, R. Ginosar, and C. P. Sotiriou, "Data synchronization issues in GALS SoCs," in IEEE Proc. Int. Symp. on Asynchronous Circuits and Systems (ASYNC'04), 2004.

[7] R. Ginosar, "Fourteen ways to fool your synchronizer," in Proc. 9th IEEE Int. Symp. on Asynchronous Circuits and Systems (ASYNC'03), pp. 89-97, 2003.

[8] T. Chelcea and S. M. Nowick, "Robust interfaces for mixed-timing systems," in IEEE Trans. on VLSI Systems, Vol. 12, No. 8, Aug. 2004.

[9] J. Jex, C. Dike, and K. Self, "Fully asynchronous interface with programmable metastability settling time synchronizer," Patent 5 598 113, Jan. 1997.

[10] I. Miro Panades, "Buffer memory control device" (Dispositif de commande d'une memoire tampon), Patent pending.

[11] A. Chakraborty and M. R. Greenstreet, "Efficient self-timed interfaces for crossing clock domains," in Proc. 9th IEEE Int. Symp. Asynchronous Circuits Systems (ASYNC'03), pp. 78-88, 2003.

[12] A. Chakraborty and M. R. Greenstreet, "A minimal source-synchronous interface," Proc. 15th IEEE ASIC/SOC Conference, pp. 443-447, Sept.2002.

[13] Y. Elboim, A. Kolodny, and R. Ginosar, "A clock tuning circuit for System-on-Chip," in Proc. 28th European Solid-State Circuits Conference (ESSCIRC 2002), 2002.

[14] L.F.G. Sarmenta, G.A. Pratt, and S.A. Ward, "Rational clocking," in Proc. ICCD, pp. 217-228, 1995.

[15] F. Mu and C. Svensson, "Self-tested self-synchronization circuit for mesochronous clocking," in IEEE Transactions on Circuits and Systems-II, vol. 48, no. 2, pp. 129-140, Febr. 2001.

[16] B. Mesgarzadeh, C. Svensson, and A. Alvandpour, "A new mesochronous clocking scheme for synchronization in SoC," in IEEE Int. Symp. on Circuits and Systems, May 2004.

[17] R. Kol and R. Ginosar, "Adaptive synchronization for multi-synchronous systems," in IEEE Int. Conf. Computer Design (ICCD'98), pp. 188-189, Oct. 1998.

[18] D. M. Chapiro "Globally-Asynchronous Locally-Synchronous systems," PhD thesis, Stanford University, 1984.

[19] J. Muttersbach, T. Villiger, K. Kaeslin, N. Felber, and W. Fichtner "Globally-Asynchronous Locally-Synchronous Architectures to Simplify the Design of On-CHIP Systems," in Proc. 12th Int. ASIC/SOC Conference, pp. 317-321, Sept. 1999.

[20] A. Sheibanyrad and A. Greiner, "Two efficient synchronous↔asynchronous converters well-suited for network on chip in GALS architectures," in Int. workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS 2006), pp. 191-202, 2006.

[21] I. Miro Panades, A. Greiner, and A. Sheibanyrad, "A low cost Network-on-Chip with guaranteed service well suited to the GALS approach," in IEEE 1st Int. Conf. on Nano-Networks, 2006.

[22] E. Beigné and P. Vivet, "Design of On-chip and Off-chip interfaces for a GALS NoC architecture," in Proc. 12th IEEE Int. Symp. on Asynchronous Circuits and Systems (ASYNC'06), pp. 172-181, March 2006.

[23] A. Sheibanyrad, I. Miro Panades, and A. Greiner, "Systematic comparison between the asynchronous and the multi-synchronous implementations of a Network on Chip architecture," in Proc. IEEE Design, Automation and Test in Europe (DATE'07), April 2007.

[24] C. Dike and E. T. Burton, "Miller and noise effect in a synchronizing flip-flop," in IEEE Journal of Solid-State Circuits, vol. 34, no. 6, June 1999.