FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Efficient Interfacing Between Timing Domains

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Robert Kutschera

Matrikelnummer 0426125

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Andreas Steininger
Mitwirkung: Univ.-Ass. Dipl.-Ing. Dr. techn. Thomas Polzer
Univ.-Ass. Dipl.-Ing. Robert Najvirt

Wien, 02.06.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Efficient Interfacing Between Timing Domains

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Robert Kutschera
Registration Number 0426125

to the Faculty of Informatics
at the Vienna University of Technology

Advisor:	Ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Andreas Steininger
Assistance:	Univ.-Ass. Dipl.-Ing. Dr. techn. Thomas Polzer
	Univ.-Ass. Dipl.-Ing. Robert Najvirt

Vienna, 02.06.2014 ________________________ ________________________

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Robert Kutschera
Untere Weißgerberstraße 53-59/4/15, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

When interacting within a closed timing domain the timing requirements (esp. setup/hold requirements) of all stateful elements (memory) in different systems or components can be safely met. Examples are a globally synchronous clock domain or an asynchronous handshake domain. Often it is, however, necessary to exchange signals between two (or more) such timing domains, as within a GALS system (globally asynchronous locally synchronous) each component/subsystem has its own timing domain. This exchange of data inevitably leads to metastability problems at the interfaces. To prevent or handle the problems with metastability of stateful elements at the interfaces of a system synchronizers are needed. In this work a survey of the different existing interfacing solutions of synchronous and asynchronous systems, including all levels of synchrony (mesochronous, plesiochronous, heterochronous, rational) and the different handshake-protocols (2-phase, 4-phase) of asynchronous system design is created. In the survey the interfacing solutions are compared according to criteria such as MTBF (Mean Time Between Failures), throughput, latency, assumptions on their functionality. From this a suitable and efficient solution is chosen for each possible system combination between synchronous and asynchronous systems, to further make a general reference book about interfacing solutions.

Kurzfassung

Die Kommunikation zwischen Elementen mit deren Setup/Hold- Anforderungen (Speicherelemente) kann innerhalb einer geschlossenen Timing Domain (Bereich mit einheitlichem Zeitverständnis, z.B.: Systemtakt) sicher bewerkstelligt werden. Oft ist es aber notwendig Signale zwischen verschiedenen Timing Domains auszutauschen, wie im Falle eines GALS-Systems (**G**lobally **A**synchronous **L**ocally **S**ynchronous) bei dem die verschiedenen Systemkomponenten jeweils unterschiedliche Timing Domains besitzen. Diese Übertragungen von Signalen können zu Metastabilitätsproblemen an den Schnittstellen führen. Um diesen Metastabilitätsproblemen entgegenzuwirken müssen die Signale an der Empfängerseite in die dort herrschende Timing Domain synchronisiert werden. Diese Aufgabe bewerkstelligen Synchronizer.

In der vorliegenden Arbeit wird als erster Schritt ein Überblick über bestehende Interfacing-Lösungen erstellt um diese durch analytische Modelle vergleichen zu können. Dazu werden verschiedene Vergleichskriterien wie MTBF (Mean Time Between Failures), Durchsatz und Verzögerung herangezogen. Dabei werden die Interface-Typen von synchronen Systemen (z.B. mesochronous, plesiochronous, heterochronous) und asynchronen Systemen (verschiedene Handshake-Protokolle) betrachtet. Nach dem Vergleich der gesammelten Lösungen kann eine Identifizierung geeigneter Lösungen für die unterschiedlichen Kategorien vorgenommen werden und so ein Nachschlagewerk verfasst werden.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Most systems are not self-contained, thus communicate with their environment, which can be either another system or an external sensor or else. Often new incoming data occurs at arbitrary points in time (seen from receiver's point of view), so asynchronously to the local timing (clock or step in computation cycle). If a change on the data signal occurs within a time interval T_w around a sampling clock edge that triggers a memory element, it may latch an undefined voltage level and thus cannot resolve to one of the defined states within bounded time.

This effect is called *metastability*. Inside a memory element the signal wobbles around half V_{DD} or oscillates during metastability [28]. The logical value has not settled yet inside the memory element, a little disturbance (e.g. noise) is sufficent for the memory element to resolve to a defined but arbitrary value (this may cause erroneous or byzantine behaviour). It is safe to say, that if this takes longer than the clock-to-output delay for the memory element to resolve to the expected value one can conclude that it became metastable. To avoid metastable states within memory elements like flip-flops or latches, the incoming data has to be synchronized to the local timing (clocked or self-timed). Hence synchronizers are required at borders of timing domains. The design of synchronizers for different application demands has long been, and still is, an area of active research, and a wealth of approaches have been published. The focus of this thesis is to produce a guide that helps one to make the right decision on the choice of synchronizer that is perfectly suitable for a particular interface between different timing domains.

In some degree synchronizers are also needed in self-timed systems, because metastability is also present in asynchronous (self-timed) systems in case of external failure or interfacing to other timing domains. In contrast to clocked systems metastable states are based upon different reasons and apply to arbiters and Muller C-Elements. Arbiters are used to synchronize incoming data to the computational cycle of a self-timed system, they grant access to the system to an external request or prevent interrupts due to incoming requests. The arbiter may suffer from the *arbiter problem*, where it can not decide on a correct order of access to the single source within bounded time if the requests are not sufficiently separated in time.

The Muller C-Element forms the basic building block of a control path in an asynchronous circuits, e.g. as element of an elastic pipeline, its storage element creates a potential for metastability, so its setup/hold time window must be respected. To rapidly resolve metastability and suppress metastable outputs for positive transistions as well as for negative transistions a Schmitt-Trigger at the output of a Muller C-Element can be employed [40].

Metastability has several effects on a self-timed system. Interpretation of metastable output voltage depends on actual threshold of the input stage, thus different stages have different interpretations, which may lead to Byzantine behaviour. Metastable voltages may propagate to the output stage and further to the next element. Oscillations with multiple periods may emerge in storage loops, whose length exceeds a critical length relative to the transistion times. From this we conclude that **Metastability** is defined as the state where signal voltage within a stateful circuit element is inbetween thresholds, thus the affected component is in an undefined state, a small perturbation (e.g. noise) is sufficient to let the output converge to one of the two stable states [12]. This process may take an arbitrary amount of time.

1.2 Problem Statement

Meeting the timing requirements of stateful circuit elements within a system gets difficult as soon as it is necessary to exchange signals with another system or the environment.
Communication within *completely synchronous* systems is conceptually simple, but lacks modularity due to the constraints that are applied to the signalling. In completely synchronous systems careful clock distribution, i.e. a balanced clock tree, is required to ensure that each part or module of the system has exactly the same clock (in frequency and phase). Thus if a component is changed a complete redesign of the system (especially the clock tree) may be required. Other communication approaches have been proposed, they circumvent the constraints of completely sychronous systems and achieve an improvement in efficiency, but open the issue of crossing clock domain boundaries and thus require synchronizers to safely transmit data. These approaches may use *separate clock domains* with either related (e.g. rationally related) or uncorrelated clocks (e.g. a GALS system), where the system is divided in several independent modules, that operate fully independent/parallel (loosely coupled), each with its own clock. It is also possible to operate systems *completely asynchronous/self-timed* (i.e. no clocks). The modules in these systems operate in lock-step and require request-driven synchronization approaches (strongly/tightly coupled). Further these approaches can be mixed, hence a clocked system can *sample asynchronous signals* from its environment (e.g. a sensor) or a self-timed system and vice versa. Further there are several different possible implementations of the data link and the used communication protocol between the modules. A simple wire (single bit) or a bus structure or an asynchronous interconnect can be used to interface different modules/systems between each other or their environment. With these interconnects various communication protocol approaches come along. For instance when using a bus structure a *handshake* protocol (2-/4-phase) can be implemented. When using an asynchronous interconnect a *Bundled Data* approach (serial communication protocol) or a *Level Encoded Two Phase Dual Rail* protocol [14] or a *Null Convention Logic* protocol can be employed.
We see that synchronizing synchronous as well as asynchronous systems is a broad field of op-

tions, many solutions are available. Thus we need to categorize our system in terms of the timing domain of our system and of our communication partner and the used link and communication protocol between them. After evaluating these parameters we want to choose an appropriate synchronizer for our system, but which one is best to be used for which system? To answer this question a survey is done to gather existing synchronizer solutions and summarize and compare them, leading us to a general reference book about synchronizer solutions and interfacing techniques between different timing domains.

1.3 Methodological Approach

In this thesis an extensive literature survey shall lead us to a general reference book about synchronizer solutions and interfacing techniques between different timing domains. We will start with a literature survey to discover possible existing synchronizers and interfacing solutions. To find the most adequate solution for each possible system combination, the discovered synchronizers are categorized (by their concept) and we compare them among each other. The comparison is done in terms of *Mean Time Between Failures* (MTBF), throughput and latency. The MTBF is the time between two consecutive sychronization failures within a system and is defined for systems with unrelated clocks as

$$MTBF = \frac{e^{\frac{t_{res}}{\tau_c}}}{T_0 \cdot f_{clk} \cdot f_{dat}},$$

where t_{res} is the resolution time, a duration in which the output has time to settle. Further τ_c, time constant, and T_0, width of decision window are empirically determined flip-flop parameters. At last we have the frequency of the sampling clock f_{clk} and the data rate f_{dat}. Note that this formula is not applying to every relationship of receiver and transmitter frequency, only in case if the clocks are unrelated (see [3] for further details on this restriction). *Latency* denotes the time the synchronization process takes to synchronize a data item to the receiver's timing domain. The *Throughput* specifies the number of transmitted data items per time unit through a synchronizer.

1.4 Structure of the work

This thesis is structured as follows, in Chapter 2, the survey of existing synchronizer solutions is started by a brief description of the classification of system synchronization, for each class a table is presented in Section 9 giving an overview of the result of the survey. In Chapter 3 the *Mesochronous Synchronizers* are described in detail. The *Plesiochronous Sychronizers* are described in Chapter 5. Chapter 6 holds the detailed descriptions of synchronizers that are used between systems with uncorrelated clocks. The *Ratiochronous Synchronizers* are described in detail in Chapter 4. Interfacing solutions for self-timed systems are described and analyzed in detail in Chapter 7. Different types of interconnects are described in Chapter 8. The thesis concludes in Section 10.

CHAPTER 2

Levels of Synchrony

2.1 Classification

Synchronizers have to cope with different amount of clock and delay variations. There are four types of clock and delay variations [26]. First of all the *skew* that is constant over time and emerges from delay inside a chip and communication. The second one is *jitter* that refers to delay that varies between clock cycles. The next variation principle is called *drift* which is similar to jitter but refers to slower variations of delay. The last mentioned source of delay variations are *fast switching* and *clock harmonics*. The differences between transmitter and receiver clock in terms of the stated variations denotes the class of a system. The following list presents a classification of system synchronization [12] [28], which forms the base for the survey below.

Classification of System Synchronization:

→ *Synchronous*:

Transmitting as well as receiving systems have same clock frequency and are in phase. Safe communication is possible with out the need of explicit synchronization.

→ *Mesochronous*:

Both communication partners employ clocks with the same frequency with an arbitrary but fixed phase shift. The clock (local or foreign) or incoming data signal has to be delayed by a constant time for safe sampling.

→ *Multisynchronous*:

A multisynchronous system is composed of different modules which are using a globally generated clock that is distributed without underlying balanced clock tree, thus the clock of each module has an arbitary relative phase drift. Such a system has to cope with delay variations and correlation which evolve from such a clock distribution, such that the frequency of its clock is the same at each module but its relative phase is apriori unknown and may change. Within a system with multisynchronous clocking each module mostly generates its local clock by reshaping the global multisynchronous clock.

$\rightarrow$ *Ratiochronous (Rational Related)*:

The clock frequencies of two systems are related by a ratio that is known at design time e.g. 1:3, it is simple to determine dangerous cycles, where metastability may occur and thus one can directly add a delay at this point of time. Thus $\frac{f_m}{f_n} = \frac{M}{N}$ describes the ratio, where f_m is the transmitter's clock frequency and f_n is the receiver's clock frequency and M and N are integers. Hence a special case of *Heterochronous/Periodic* system synchronization.

$\rightarrow$ *Plesiochronous*:

Both systems have the same or nearly the same clock frequency. There is an arbitrary phase shift between the system clocks which is drifting slowly. To safely sample an incoming signal a continuously adjustable delay is needed for clock or data signals.

$\rightarrow$ *Heterochronous/Periodic*:

Systems are both clocked (periodic) at arbitrary nominally different frequencies. Synchronizers take advantage of the periodic nature of clock and data signals, to predict dangerous events which would lead to metastable states.

$\rightarrow$ *Asynchronous/Clocked*:

Communicating systems are in the same relationship as at *Heterochronous/Periodic*, but the synchronizer will not or cannot take advantage of the periodicity of the clocks, thus events occur at arbitrary times, which further requires a synchronizer that is capable of every frequency relationship (assume unrelated).

$\rightarrow$ *Asynchronous/Self-Timed*:

One or both communication partners are self-timed, means these systems do not employ clock signals and thus are event-triggered using either a handshake protocol (2-/4-phase) and/or a dual-rail implementation of the data path to maintain the delay-insensitivity property.

The major part of the above presented classes uses *seperate clock domains* (e.g. a GALS system), where the system is divided in several independent modules each with its own clock. The local clock of each module can either be generated within the module or derived from a central source that distributes a system wide clock to every module (i.e. unbalanced clock tree). Thus changes of modules are independent to the system and the other modules. Due to the modularity and the involved simplified signalling this approach is most commonly used in practice. A drawback is that the initial synchronization adds latency to the data transmission. The *asynchronous/self-timed* approach has several advantages compared to the clocked ones like less power dissipation, further the system remains in a quiescent state until triggered for a new data transmission. Although an asynchronous system is obviously event triggered in some degree there is synchronization or arbitration required, to synchronize incoming events to the current computation cycle (run) and to arbitrate between asynchronous requests, basically to determine which signal was asserted first, as mentioned before. Several of the synchronizers presented in this thesis employ an asynchronous interconnect to interface different modules/systems between each other or their environment. The advantages of such an asynchronous communication path (and asynchronous logic in general) are less power consumption, average case performance, adaption to changing environmental conditions and reduced electromagnetic radiation, which make an asynchronous interconnect more attractive and valuable. Interfacing asynchronous systems needs distinct design methodologies [23] [24]:

First there is the *Bundled Data* (**BD**) approach, where data transmission is parallelized. With this approach a 2(NRZ)-/4(RTZ)-phase handshaking (request and acknowledgement) is introduced. To avoid race conditions between the data stream and handshake signals the gate as well as wire delays are bounded by assumption. The main problems with this approach are glitches, and therefore masking is needed.

Further the *Level Encoded Two Phase Dual Rail* (**LEDR**) protocol [14], which is a delay-insensitive (no clocking) protocol that uses a 2-phase handshaking (with NRZ). Gate and wire delays are unbounded and unknown but finite. The data transmission needs a completion detection logic. A dual-rail approach is needed to determine “no transition” from a late transition, that is two physical wires are employed to represent a single signal. Data in this phased logic [24] is represented in two phases, φ_0 and φ_1, each includes both logical values. The phase of the signal changes with each transition, this is derived by a XOR operation on the two lines. Synchronization is easily done by waiting for all input signals to be in the same phase.

At last the *Null Convention Logic* (**NCL**) protocol, as the LEDR protocol it is a dual-rail approach, but uses a 4-phase handshaking (with RTZ). The protocol extends the boolean logic by a NULL value. The data stream consists of alternating waves of data and NULL (empty) words. These methodologies use a handshaking protocol, a *4-phase handshake* involves the following steps [27] (see Figure 2.1): At first a new data is available at the transmitter. From this the transmitter rises the request signal and keeps it up to signal a starting data transmission. The request is synchronized at the receiver and the bundled data stream from the transmitter is latched. After data transfer has finished the receiver rises its acknowledgement signal, which is synchronized at the transmitter. Data now may be removed from the output at the transmitter and the request signal is de-asserted and synchronized at the receiver. The receiver then de-asserts its acknowledgement signal which is further synchronized at transmitter. After these steps the transmitter is allowed to start this procedure all over again. A *2-phase handshake* on the other hand involves similar steps, but in contrast to the 4-phase handshake each transition on the request and acknowledgement lines signal a new request or acknowledgement.

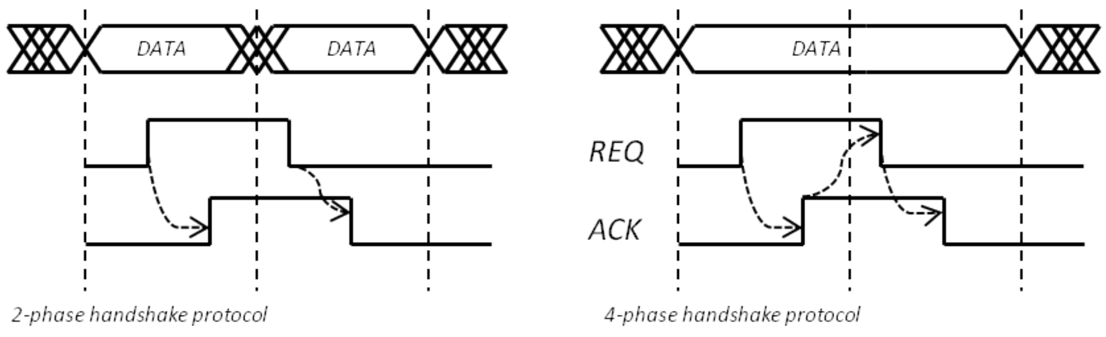


Figure 2.1: Handshaking signals (2-/4-phase)

CHAPTER 3

Mesochronous Synchronizers

A mesochronous system consists of communication partners that employ clocks with the same frequency with an arbitrary but fixed phase shift. Synchronizers can exploit this relationship between the two clocks to efficiently synchronize data communication.
Thus *Mesochronous synchronizers* need two mechanisms to safely interface two systems. First they need a **phase estimation** mechanism, to determine by which amount the phase of the receiver clock is shifted in respect to the transmitter clock, there are different approaches given in sections 3.2.2, 3.2.5 and 3.2.6 (see Figures 3.2, 3.8 and 3.10 respectively) that employ a phase estimation circuit. From the result of the phase estimation the synchronizer educe how to adjust a delay in the data path or on the clock or control lines, that is formed either by an adjustable delay element (see 3.2.2 and 3.2.5) or alternative data paths(see 3.2.3, 3.2.4 and 3.10), to adjust the arbitrary phase shift between clock reference and data signal and keep signal transistions away from unsafe regions of the clock.

3.1 Basic Concepts

The mesochronous synchronizers presented in this chapter can be classified in three basic concepts:

- First the *Brute-Force Synchronizer*, the *Delay-Line Synchronizer* and the *Adaptive Synchronization* approach directly **modify the timing in the data stream** either by a cascade of flip-flops or a variable delay element in the data channel.

- The *Three-Element FIFO Synchronizer* and the *Four-Stage Mesochronous Synchronizer* both use a **ring buffer** sampling incoming data every cycle of receiver's clock to one of its latches in spite of possible metastable states and let it rest for one turn (the other latches are written in sequence) and thus let a possible metastability decay.

- The *Two-Register Synchronizer* and the *Low-Latency and Low-Overhead Mesochronous Synchronizer* **control the input latches** at the receiver module to safely sample new data, either by delaying the control input of the latch relative to the receiver clock or using a *strobe* signal, sent by the transmitter (synchronized by a *Brute-Force Synchronizer*) to enable the latch.

From each set of synchronizers associated with the different basic concepts one solution is picked as a representative in the following general discussion of properties. Details on the implementation and function of all solutions can be found in the subsequent subsections. Note that a calculation of the MTBF using the formula given in the introduction does not work for mesochronous system due to the correlated clocks [3], but if the synchronizer is properly configured it achieves an infinite MTBF in the data path.

From the set of synchronizers that directly modify the timing in the data stream the *Adaptive Synchronization* approach is chosen. This approach employs a learning phase to dynamically adjust the delay line on the data channel. Thus is able to react to changes in phase drift (also capable of plesiochronous timing relations) when employing a continuous learning phase, thus achieving an infinite MTBF and a latency of half a clock cycle maximum (variable delay element). A throughput of m bit per clock cycle can be attained with m parallel input data paths each employing an adapative synchronization block and a variable delay.

From the set of parallel staged synchronizers the *Four-Stage Mesochronous Synchronizer* is chosen for discussion. It is an enhancement of the *Three-Element FIFO Synchronizer*, with a fourth stage and a FIFO at the input of the receiver to enable data burst from the transmitter and a mechanism to support back-pressure. The *Four-Stage Mesochronous Synchronizer* is chosen as the reference of its concept class, due to its enhanced functionality and MTBF compared to the *Three-Element FIFO Synchronizer*. In detail the *Four-Stage Mesochronous Synchronizer* provides a infinite MTBF, a latency of three clock cycles and one word per cycle throughput (see table in [17]).

From the last group the *Low-Latency and Low-Overhead Mesochronous Synchronizer* is picked as a representative. It uses a *strobe* signal, that is generated in the transmitter timing domain and synchronized by a *Brute-Force Synchronizer* to the receiver timing domain, to determine whether it is safe to sample incoming data on either the rising or falling clock edge (or if only dummy data was sent). This mechanism has the advantage that synchronization is done completely off the data path, hence metastabilities can only occur at the *strobe* signal input. The MTBF in the *strobe* signal path is very high due to the used cascade of latches. The latency of the *Low-Latency and Low-Overhead Mesochronous Synchronizer* is in best case T_w, equal to the delay in the *strobe* signal path, and in average case $\frac{T}{2} + T_w$ or in worst case $T + T_w$ (see [7]). It supports "maximal throughput" [7] of one data token per clock cycle. For a detailed description and references of the comparative parameters of the chosen prototypes see the correspondening section below (*Adaptive Synchronization* 3.2.5, *Four-Stage Mesochronous Synchronizer* 3.2.7, *Low-Latency and Low-Overhead Mesochronous Synchronizer* 3.2.6).

3.2 Detailed Descriptions

3.2.1 Brute-Force Synchronizer

The *Brute-Force Synchronizer* or *Waiting Synchronizer* [12] employs two or more (like in the conservative version, see 6.2.1) flip-flops. A system employing a *Brute-Force Synchronizer* is depicted in Figure 3.1. At first the flip-flop samples the input signal (asynchronous) and waits (one clock cycle) for any metastable state to decay, then in the next clock cycle the (possibly) stable output of the first flip-flop is sampled by a second flip-flop adding another cycle of resolving time to the synchronization process and providing the data item to the receiver. This synchronizer does not take advantage of knowledge about periodicity of clocks. Disadvantages involved with this approach are enhanced delay and a non-zero probability of synchronization failure. Further if the phases of transmitter and receiver clock almost match and the synchronizer runs into “bad” timing the input data may be corrupted by a metastability event leading to inconsistency in the communication which introduces a criterion for exclusion and thus it should not be used. Its average delay is $t_z = t_w + 2t_{dCQ} + t_{cy}/2$ (see [12]), where t_{dCQ} is the clock-to-output delay of the synchronizing latches and t_{cy} is the cycle time and t_w, the waiting time denotes the time a system is configured to wait until the output of synchronizer is sampled, typically about one clock cycle. In summary, the synchronizer has a non-zero probability of synchronization failure (thus a non-infinite MTBF), similar to the *Two-Flip-Flop Synchronizer*. The *Brute-Force Synchronizer* has an average latency of t_z and a very low throughput hence it synchronizes only one bit every three clock cycles.

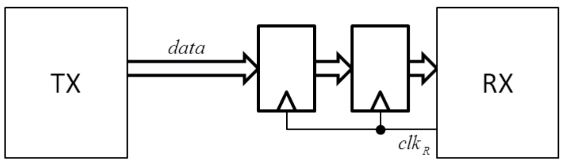


Figure 3.1: Digital Systems Engineering [12]: Brute-Force Synchronizer

3.2.2 Delay-line synchronizer

The *Delay-line Synchronizer* presented in [12] is used to synchronize mesochronous systems, a block diagram is depicted in Figure 3.2. It employs a variable delay-line on the data path with an adjustment range of a clock period. During its learning phase two flip-flops capture the variable delayed input with two different delayed versions of the clock to measure the relative phase of the delayed input and further adjust the variable input delay on the data path. If the delayed input signal changes during the keep-out region of the clock the two flip-flops will capture different values (the lower one will sample a different value) and signalling the FSM to adjust the delay so that it keeps the input out of the forbidden region. In detail the second and third flip-flop span a window of $\pm\, t_d$ around the rising edge of the original clock to simply define the *keep-out region* (see timing diagram in Figure 3.3). With a well chosen value for delay element t_d the setup and hold constraints can be reflected, hence t_d at the second flip-flop matches t_{hold} and $t_{cy} - t_d$ at the third flip-flop is equal to t_{su}, iff $t_d < t_{cy}/2$. The synchronizer employs an average delay of $t_z = t_{v(min)} + t_d + t_{dCQ} + t_{cy}/2$, where $t_{v(min)}$ is the minimum variable delay, t_d is the value of the delay element at the control input of the second and third flip-flop, t_{dCQ} is the clock-to-output delay of a flip-flop and t_{cy} denotes the cycle time. In summary, the *Delay-line Synchronizer* has a infinite MTBF, provides a data item per clock cycle with an average latency of t_z.

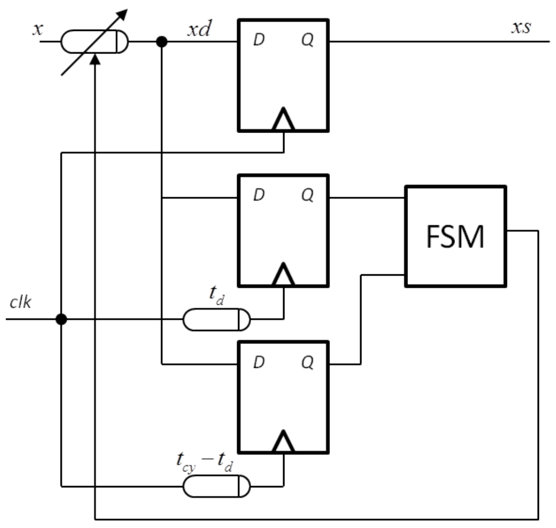


Figure 3.2: DSE [12]: Delay Line Synchronizer

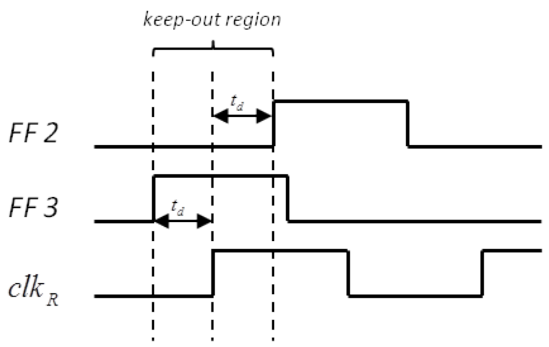


Figure 3.3: DSE [12]: Delay Line Synchronizer - Timing Diagram

3.2.3 Two-Register Synchronizer

The *Two-Register Synchronizer* [12] is similarily constructed as the *Delay-Line Synchronizer* but it delays the clock line instead of the data line, as one can see in Figure 3.4. There are two flip-flops or registers (for multiple data lines), one samples data on the rising edge of the clock, and the other one is triggered by a delayed version of the same clock edge, thus sampling data delayed at least by the width of the forbidden (keep-out) region of the flip-flop. A *phase comparator* measures the relative phase between the local clock and the transmitter clock of the incoming data stream once after reset and thus determines which of the flip-flops is safe to be selected by the multiplexer. The measurement phase may be repeated during operation to support variable phase shifts. Usually the upper (not delayed one) flip-flop is used unless there is an event on the data line during the keep-out region, in this case the lower one is chosen by the multiplexer. The average delay of the *Two-Register Synchronizer* is $t_z = t_{ko}^2/t_{cy} + t_d + t_{dCQ} + t_{cy}/2$, where the term t_{ko}^2/t_{cy} is the duration of the keep-out window multiplied with the probability that the lower flip-flop is chosen by the comparator, t_d is the delay of the flip-flop, t_{dCQ} the clock-to-output delay and $t_{cy}/2$ the half of the cycle time, which is the inherent synchronization delay of rounding up to the next cycle (see [12]). In summary, the *Two-Register Synchronizer* achieves an infinite MTBF in the data path, if the comparator has chosen correctly and the phase shift remains constant. On the other hand the phase comparator itself may get metastable, the MTBF in this case depends on the used technology. Further the synchronizer employs an average latency of t_z and a throughput of a data item each cycle.

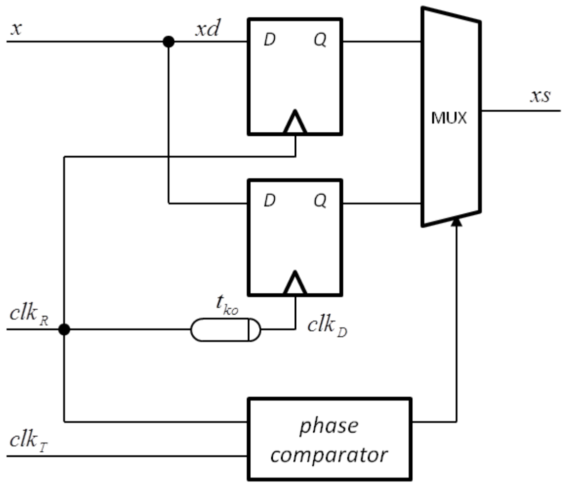


Figure 3.4: DSE [12]: Two Register Synchronizer

3.2.4 Three Element FIFO Synchronizer

The *Three Element FIFO Synchronizer* [12] [28] uses a small ring-buffer to decouple transmitter and receiver timing. The synchronizer employed with three stages is depicted in Figure 3.5.

Incoming data items are written alternately into a couple of flip-flops. The flip-flop is chosen by a *transmit* pointer (*xp*) which is generated by a counter in case of a three-element (or higher) synchronizer. A further ring counter, which is driven by the local clock of the receiver, is controlling the multiplexer to select the flip-flop with the oldest sampled value for read. The pointer of the receiver (*rp*) should lag the pointer of the transmitter (*xp*) by at least one clock cycle. In fact this lag represents the available resolution time and should therefore be maximized. Obviously more stages allow for higher resolution time. Note that the pointers *xp* and *rp* need to stay in sync to avoid pointing to the same flip-flop. This is guaranteed by the assumption of a mesochronous environment. In this respect a higher number of stages allows for a longer distance between the pointers and hence better tolerance of long phase variations. To accommodate for the combinational delay introduced by the multiplexer another flip-flop stage is added to relax the contraints. On the other hand is also possible to reduced the number of stages to two. In this two-element FIFO synchronizer only two toggle flip-flops are needed to generate the *xp* and *rp*, instead of counters. With only two flip-flops the data latency is reduced, but problems may occur in case the two clocks are nearly in phase, because it is not guaranteed that the chosen value will be stable for an entire clock cycle at the multiplexer (may be overwritten before read). Hence the three-element (or higher) version might be the preferred choice. In summary, a possible metastable state in one of the flip-flops has *n* clock cycles to decay, thus the MTBF is clearly finite for the *Three Element FIFO Synchronizer*, where *n* is 3. Data is passed every clock cycle after the three stages are filled initially with a data latency equal to the number of stages (i.e. a data item stays in the FIFO for n cycles of the receiver clock).

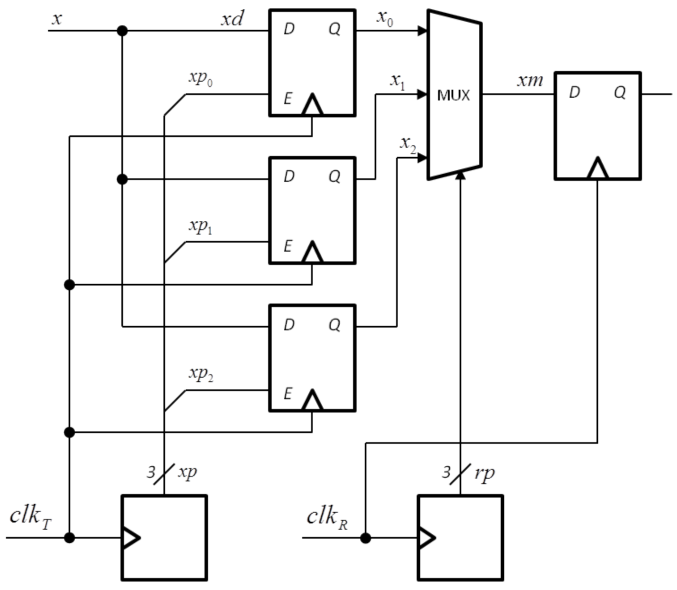


Figure 3.5: DSE [12]: Three Element FIFO Synchronizer

3.2.5 Adaptive Synchronization

Adaptive Synchronization [26] is a proposed method to synchronize multi-synchronous modules within a system. Delay variations and correlation evolve from unbalanced clock distribution, such that the frequency of the system clock is the same at each module but its relative phase is apriori unknown and may change. In [26] the relative phase is supposed to be "stationary" which means that the relative phase can be seen as fixed over long periods of time. The transmitter module provides additionally to the data a ready signal to the receiver module. The ready signal and the local clock are provided to a conflict detector which activates a counter as long as a detected conflict lasts (see Figures 3.7 and 3.8). The counter is reset at the start of the adaption cycle by the controller. The counter value (one-hot encoding) directly increases the number of inverters within the digital delay line (see Figure 3.9) to delay the incoming data. The conflict detector employs four mutual exclusion elements, each two to detect conflicts at rising and falling edges. A conflict is detected if the time interval between a transition of the ready signal and transition of the local clock is shorter than the employed delay of the conflict detector (see Figure 3.3). In [26] there are five modes (learning phases) proposed in which adaption is achieved. There are "one time adaption" (after manufacture), "power-up adaption" (once after power is applied), "periodic training sessions", "triggered training sessions" and "continuous tracking" to accommodate a different amount of drift and different electrical schemes. In summary, the synchronizer employs a very high or up to infinite MTBF (depending on which mode of learning phase is used), a latency of maximal a half clock cycle (variable delay) plus a learning phase and thus passes data each clock cycle.

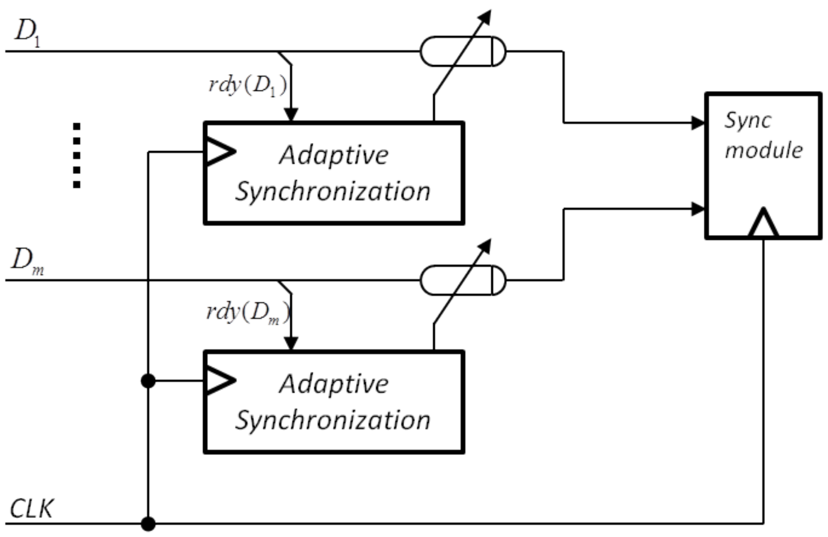


Figure 3.6: Adaptive Synchronization [26]: Structure

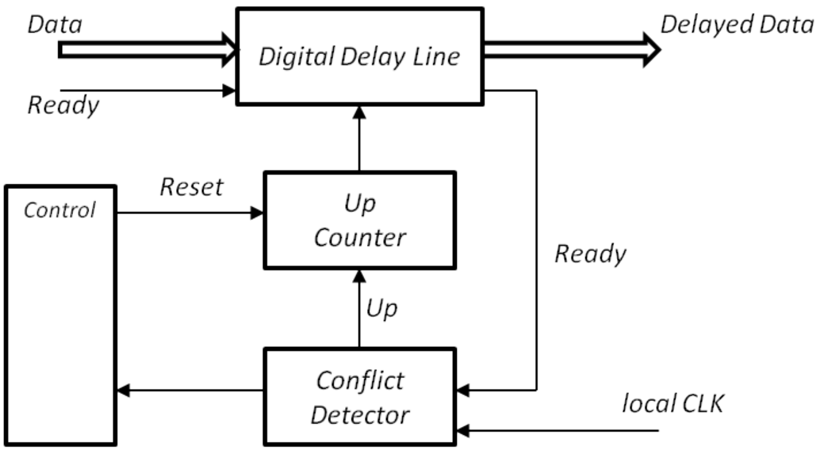


Figure 3.7: Adaptive Synchronization [26]: Adaptive Sensitivity Implementation

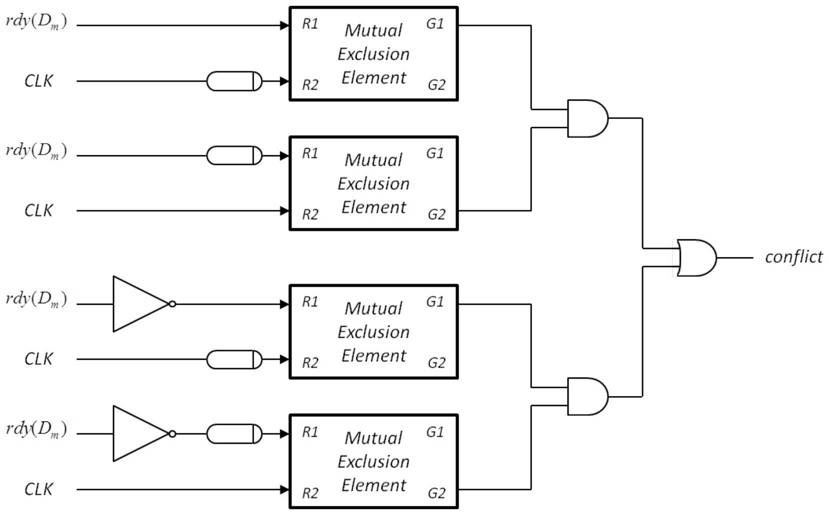


Figure 3.8: Adaptive Synchronization [26]: Conflict Detector

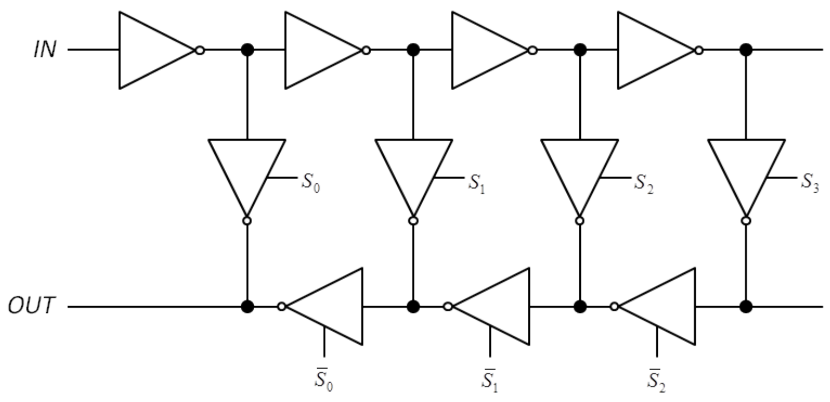


Figure 3.9: Adaptive Synchronization [26]: Digital Delay Line

3.2.6 Low-Latency and Low-Overhead Mesochronous Synchronizer

A low-latency and low-overhead mesochronous synchronizer is presented in [7]. It can be used in so-called globally non-synchronous system as GALS or systems with unbalanced clock trees, which bring performance enhancements and power savings in contrast to globally clock systems (with a balanced clock tree). Due to such an unbalanced clock tree each module of the system run with the same nominal frequency indeed, but with an unknown constant phase shift between them. The mesochronous receiver interface comes with an over-head of three flip-flops per data line, see Figure 3.10. The synchronizer does not employ backpressure or a handshaking mechanism and the transmitter is assumed to send data every clock cycle. It decides upon a learning phase when it is safe to sample further whether data should be sampled with rising or falling clock edge of receiver clock. A *strobe* signal that is generated by the transmitter, which ideally toggles with all data lines, is used to learn about when it is safe to sample. Note that the learning phase is necessary only once after reset in the mesochronous synchronizer. The strobe signal (bundled with data lines) changes from low to high and is sampled every rising and falling edge during the learning phase by the receiver. A delay T_w of a quarter of the cycle period time is inserted to ensure that a change of the strobe signal is first sampled on the falling edge. An additional flip-flop in this path is used to “move” the sample to be passed on the same rising edge as the sample taken a half clock cycle later (i.e. actually on the rising edge). The delayed signal is called *strobed*, that is used by the receiver to control the multiplexer that switches between the data input that is sampled on the falling and the other one that is sampled on the rising clock edge. During the learning phase two consecutive samples of the *strobed* signal are compared. These samples results in a sequence $s_0, s_1, \ldots, s_i$, where s_i is the first sample of the strobe signal that is high, thus s_0 to s_{i-1} are low samples. If s_i is sampled by a rising edge of the receiver clock the rising edge is further used to sample data. The learning phase thus takes several clock cycles, in this phase only dummy data is received. Clearly the “strobed” signal may become metastable, thus must be synchronized by a cascade of flip-flops. The use of an additional control signal as *strobe* moves the synchronization issue and thus the probability of metastabilities away from the data path. The total number of flip-flops needed for mesochronous interface is $3\cdot$*(number of data lines)* $+ 6 + 2\cdot$*(number of FF for strobed signal)*. Since the learning phase happens only once after reset it does not effect the data latency. In summary, the *Low-Latency and Low-Overhead Mesochronous Synchronizer* avoids metastabilities at the data input by using a *strobe* signal to control the input flip-flop, at this path a very high MTBF is achievable (as mentioned before the MTBF cannot be calculated with the given formula and thus only an estimate is given). The initial latency of the synchronizer consists of a single learning phase after reset and in best case only T_w (delay of the *strobe* signal), average case $T/2 + T_w$ or worst case is $T + T_w$ [7]. The synchronizer operates with maximal throughput (a data item per cycle), an example in [7] shows data transfer at $1GHz$.

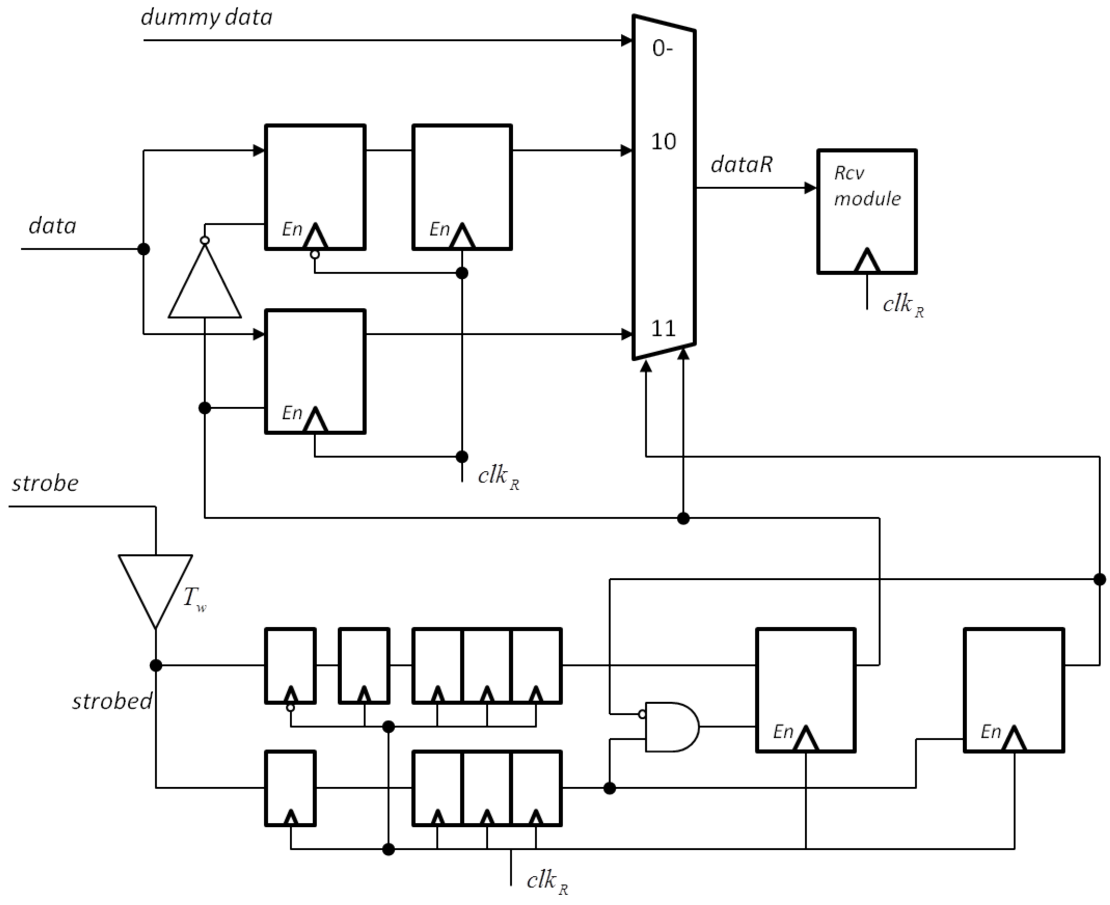


Figure 3.10: Low-Latency and Low-Overhead Mesochronous and Plesiochronous Synchronizer [7]: Mesochronous Receiver

3.2.7 Four-Stage Mesochronous Synchronizer

The *Four-Stage Mesochronous Synchronizer* (FSMS) presented in [17] provides full backpressure and buffering for data bursts, thus the receiver clock and the read/write pointers are never stopped or modified. Further it can be used for long range communication. If a *FSMS* is considered to interface between systems the most important design considerations are first the inital spread of read and write pointers and further the number of the used synchronizer stages (minimum of 4 stages). The *FSMS* employs two cyclic buffers, one for the transmitter and one for the receiver with each 4 parallel stages (registers), one can see a block diagram of the *FSMS* in Figure 3.11. The transmitter buffer uses four stages as data buffer for the data transfer and another four stages to buffer *forward* token, which indicate the validity of data items. Since a *forward* token shows data validity to the receiver it is used to push new data into the FIFO buffer, thus is linked with empty status line of the FIFO buffer and the *pop* signal. At the transmitter side the data transfer is done by writing data and *forward* token cyclically in one of the four stages, the stage is chosen by a cyclic counter, that is never stopped and then data and *forward* token will be transmitted to the receiver. Due to initial spread (chosen by design) of the read pointer data can settle and metastability decay before it is read from a transmitter stage into the receiver FIFO. The *forward* token indicates that the data item is valid in the corresponding data register (transmitter stage). Since the receiver clock is never stopped or modified data can arrive every cycles at the receiver side. The receiver can either read data directly from transmitter stage and process it or buffer it firstly into its FIFO. Further for the receiver to be capable of data bursts from the transmitter its FIFO needs to be deeper than the minimum four stages. The receiver may assert backpressure by sending a *backward* token containing logical 0 for “do not send”, which is produced by de-asserting the pop signal. By this the data input is directed to the FIFO to buffer incoming data items which were currently pending in the transmitter buffer when the *backward* token was sent. Therefore the FIFO threshold must be $L - 4$, where L is the size of the FIFO, to prevent data loss, thus its minimal depth is 4 or equal to the transmitter buffer. One *backwards* token is sent by one stage of the receiver buffer and blocks the corresponding stage in the transmitter buffer sending new data. For long wire communication (e.g. mesochronous NoC) the four stage synchronizer can either be employed at only one communication partner (transmitter or receiver) or in a split fashion. In the former case no modifications are necessary, in the latter an extension on the transmitter buffer (stages) depth will be necessary to compensate long delays. The *FSMS* can be further used to interface between *multi-synchronous* modules. Note that a system is *multi-synchronous* when a common clock is provided to its local modules without taking care of a balanced distribution. In this case six FIFO stages are needed to handle the occuring phase drifts during operation. Note that an additional phase shift by T adds two synchronizer stages, this results in a total number of stage of $4 + 2k$, where k is an integer that numbers the phase drift $[0, \pm kT]$. The asynchronous reset must be synchronized by a 2 flip-flop brute force synchronizer to the transmitter as well as to the receiver domain to synchronously release the reset signal. To prevent metastability the read and write pointer of the used FIFOs must not be initialized to the same value, there has to be an initial spread of at least 2 stages. Forward latency varies in the range of $(T, 3T)$ and depends on the initial pointer spread and relative phase difference. Throughput is maximized, one data item can be transferred on each clock cycle, but backpressure affects the throughput latency. A comparison table between the four-stage mesochronous synchronizer and the 2-Clock FIFO is provided in [17]. In summary, the *Four-Stage Mesochronous Synchronizer* has an infinite MTBF, a average latency of three clock cycles and a throughput of one word per cycle [17].

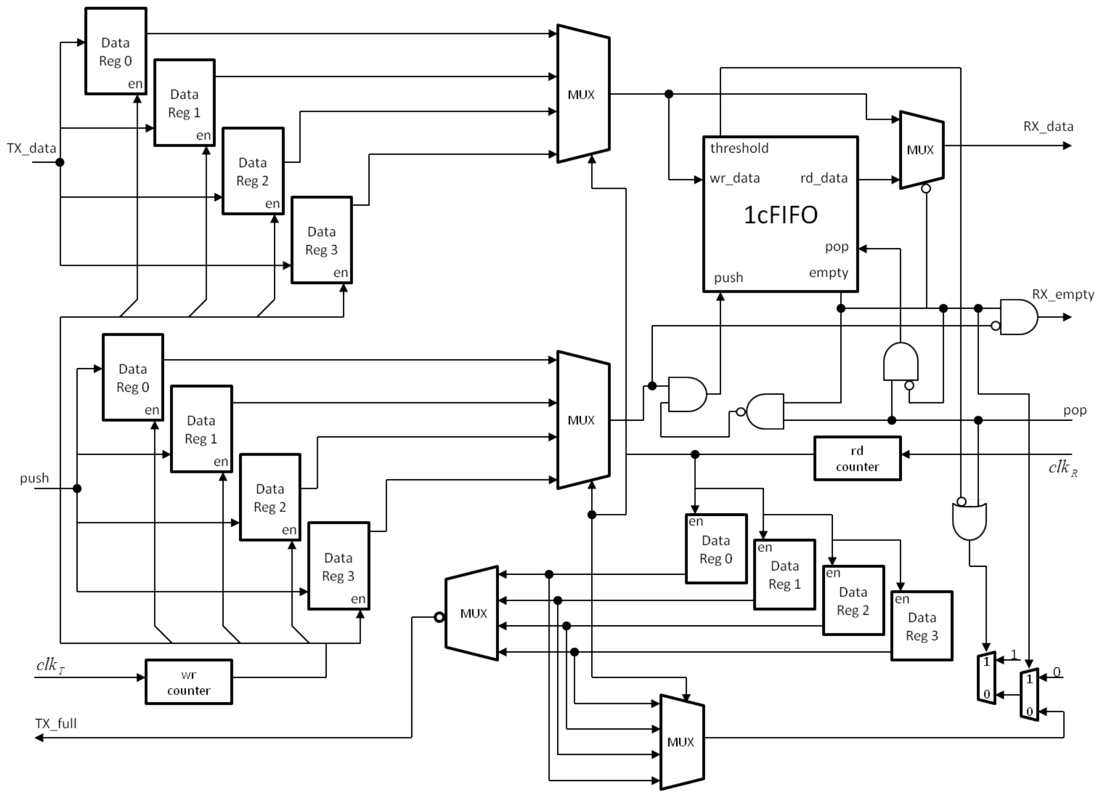


Figure 3.11: Four-Stage Mesochronous Synchronizer [17]: Structure

CHAPTER 4

Ratiochronous Synchronizers

Ratiochronous Synchronizers are used to interface between clocked systems with rationally related frequencies.

4.1 Basic Concepts

There are two **Ratiochronous** Synchronizers presented in this chapter where the clocks of the communicating systems have to be rationally related, one from *Rational Clocking* that employs a fixed communication schedule placed in a LUT and furthermore the *GRLS* approach, which takes advantage of the periodic nature of the rational relation and uses a *strobe* signal to regulate data latching at the receiver.
From the two ratiochronous synchronizers presented in this thesis, the *GRLS* approach is chosen as the prototype for this category. The main advantage of the *GRLS* approach over the *Rational Clocking* technique is that it is able to cope with phase shifts in the rationally related clocks. Further the *Rational Clocking* technique implements a *lazy* algorithm, hence introduces a higher latency than the *GRLS* approach (length of input delay in best case). The *GRLS* synchronizer is based on the *Low-Latency and Low-Overhead Mesochronous/Plesiochronous Synchronizer* presented before, thus moves the synchronization away from the data channel and uses a *strobe* signal to control the input at the receiver. Both approaches provide optimal throughput (i.e. one data item per clock cycle).
The following section holds the detailed descriptions of these synchronizers, the parameters that are used for comparison are taken from the original resources.

4.2 Detailed Descriptions

4.2.1 Rational Clocking

The proposed solution acts on the assumption that the frequencies of the clocks of two systems which tend to communicate are related by the ratio of two integers and that the phase relationship of these clocks is known. This can be used to exploit the predictability of points in time where it is not safe for the receiver to sample data and further create a schedule to time all communications and data transfers. This is called "rational clocking" [43] and relies on a given frequency ratio between the clocks of two modules f_1 and f_2. Thus $\frac{f_m}{f_n} = \frac{M}{N}$ describes the ratio, where f_m is the transmitter's clock frequency and f_n is the receiver's clock frequency. *M* and *N* are (small) integers that represent the same ratio as f_m and f_n. The communication schedule is generated upon the prediction of the relative position of the systems clock edges within *M* cycles on system *m* (employs clock with frequency f_m) and *N* cycles on system *n* (with clock frequency f_n) and shows safe and unsafe cycles for transmission. It can be derived in two ways either statically in a schedule diagram written to a LUT or dynamically at run time using graphical algorithms and a schedule plot. A schedule diagram contains the sequence of cycles and the corresponding values that state if it is safe to transmit/receive in this cycle. Within a cycle of the transmitter there is a transition window that defines when the transmitter latches data into the transmit register. At the receiver there is a decision window or setup-hold window defined around each clock edge during which the data must be stable at the input of the receiver to be sampled safely. If these windows overlap it is not safe to transmit data. So starting at the transmitter a sequence is derived where alternatingly transition windows and the next following decision window at the receiver (must not overlap) form a communication schedule (see Figure 4.1 and 4.4). This schedule is created at boot-up or is programmed into a ROM. On the other hand it is possible to generate a schedule at run-time. The advantage of this is that it has an *O(log(n))* need in memory in contrast to the previous mentioned method which has a cubic growth rate in needs of memory bits, *O*(N^3) for a supporting ratio range of $1 \ldots N$. Run-Time Scheduling uses a graphical algorithm similar to Bresenham's algorithm to generate a schedule plot. The schedule plot is a function *f(t)* defined by $f(t) = t - (P_m + S_n)$ that is equal to the time the transmitter has to provide data stablely to the receiver. P_m is the end of the transition window and S_n the start of the decision window. The resulting straight line is drawn on a grid of $M \times N$ lines. For each of the *N* vertical line a dot is set beneath *f(t)* on the nearest crossing point with one of the *M* horizontal lines of the grid. At these dots, which define points in time, it is safe to sample data. If two nearby dots are on the same horizontal line it is not safe to sample data on the later dot or no new data is available yet. For a detailed algorithm and implementation in hardware see [43]. The interface hardware employs a phase-locked loop to derive the clock of system *N* from the clock of system *M*. Two frequency-dividing counters are employed to indicate the current clock cycle of each system and form the index for the LUT. Each of the two LUTs generate a control signal for transmitter and receiver register, to enable the register when it is safe to sample or latch new data for transmission. Each system employs a *transmit* and a *receive register* (see Figure 4.2). The LUTs can be "generic" to take *M* and *N* as inputs and produce the signals for the appropriate ratio or they can be "programmable" where a LUT is loaded at boot-up from a ROM providing different frequency ratios. When creating a communication schedule the starting transition window affects the throughput efficiency, so one needs to make a schedule for each transition window as starting point or uses the double-buffering technique, which guarantees 100% throughput efficiency. The double-buffering technique uses multiple registers to alternately sample data. When using average timing parameters (of components) two registers are sufficient, but [43] provides a formula to derive the appropriate number of registers for one's design. In the double-buffering technique the transmit register is doubled and further selected by an additional control signal generated by the LUT during operation (see Figure 4.3). If the transmission is not safe data is placed in the second register. This gives new possibilities for communication schedules (see Figure 4.4). The Run Time Scheduling introduces a delay which can be critical in fast, high performance systems.

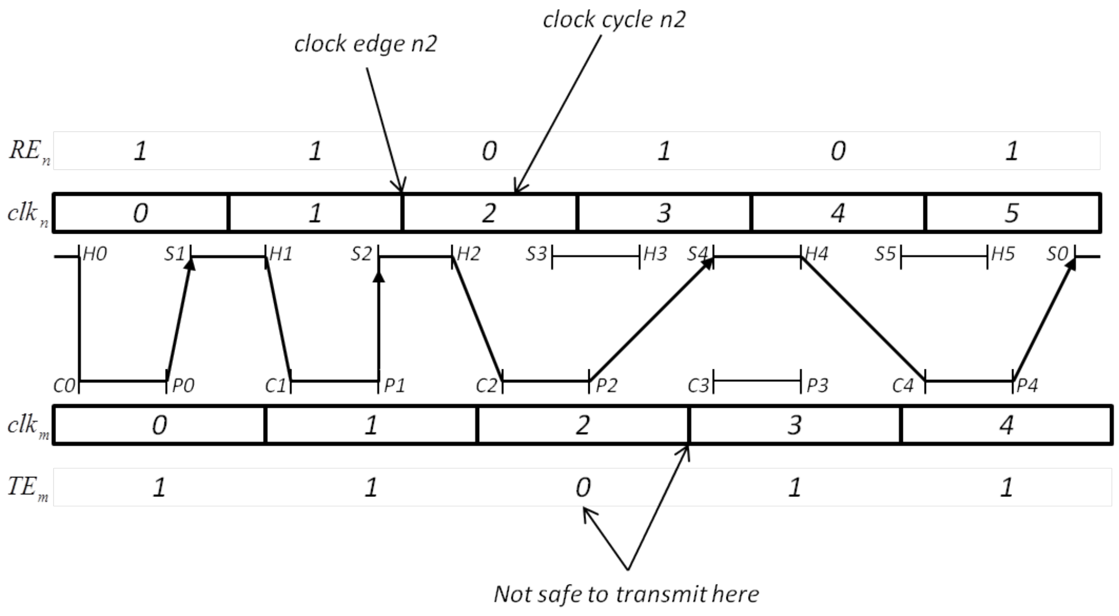


Figure 4.1: Rational Clocking [43]: Schedule Diagram 1

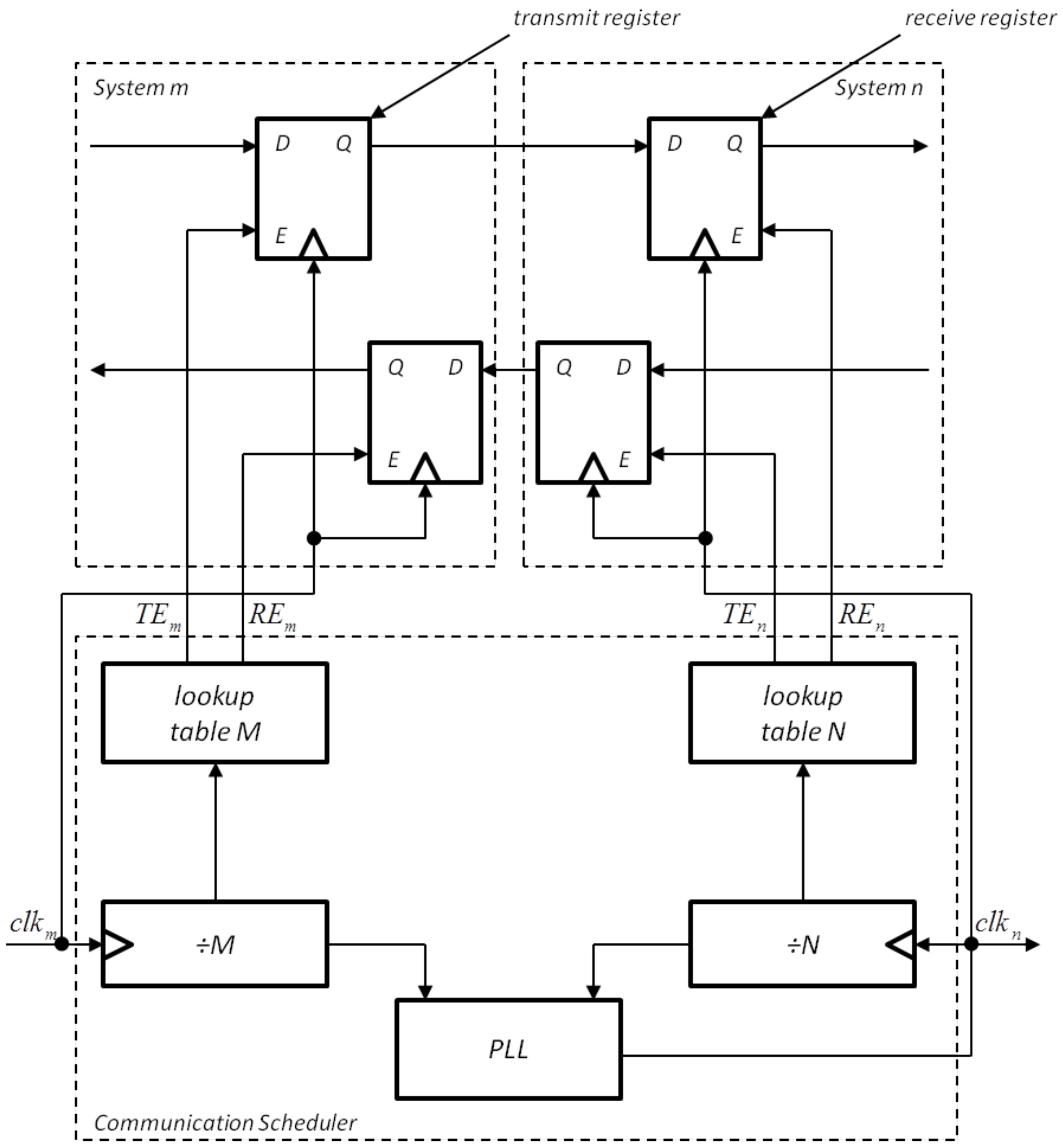


Figure 4.2: Rational Clocking [43]: Interface Hardware

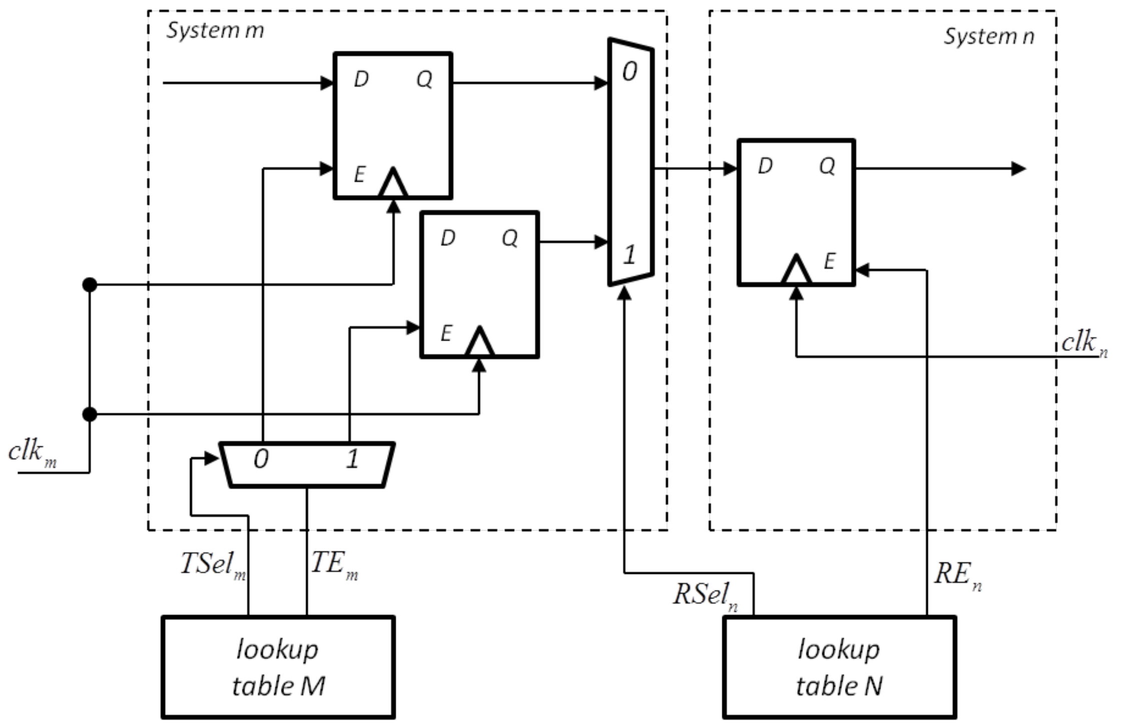


Figure 4.3: Rational Clocking [43]: Double Buffered Hardware (direction $m \to n$ shown only)

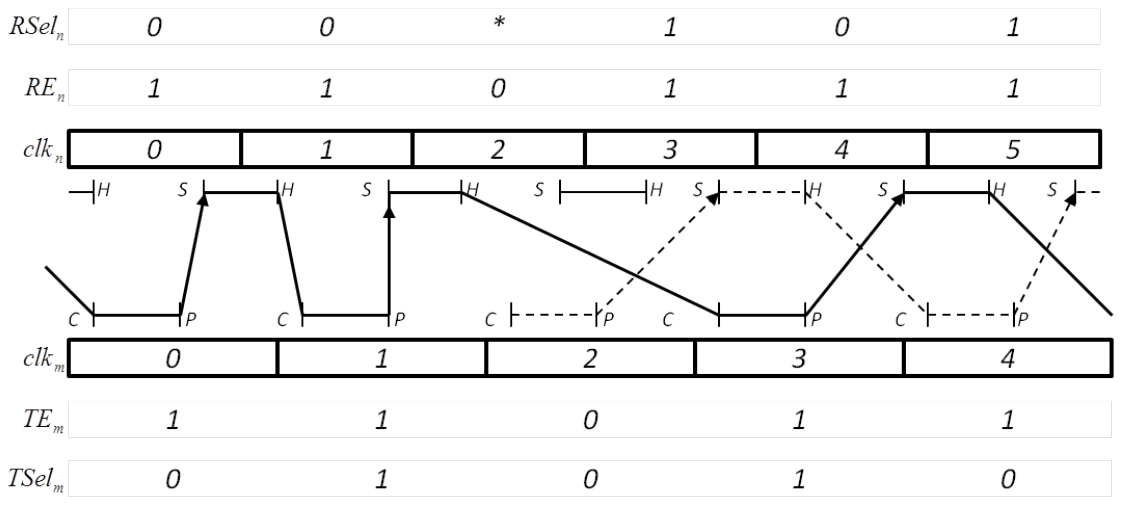


Figure 4.4: Rational Clocking [43]: Schedule Diagram 2

4.2.2 A Flexible Communication Scheme for Rational-Related Clock Frequencies

In [5] and [6] a design style is introduced called *Globally-Rationchronous Locally-Synchronous* (GRLS). It is located between the mesochronous and the GALS design style. Basically it constrains the frequencies used by the communicating modules to be rationally related, such that there is a global clock with frequency f_H that is determined by the least common multiple (lcm) of the frequencies of the receiver and the transmitter, where $f_H = N_T \cdot f_R = N_R \cdot f_R$. In contrast to the rational clocking technique from [43] presented in section 4.2.1 the GRLS interface can cope with unknown phase differences (skew) between the receiver and transmitter clock. The GRLS interface exploits the periodic nature of the rationally related systems/modules to safely interface them. The transfer rate is dictated by the slower communication partner. Note that the clock edges (of receiver and transmitter clock) align with period $PC = N_R \cdot T_T = N_T \cdot T_R$, where *PC* denotes the periodicity cycle. The interface operates as follows; the transmitter module passes a data item to its local GRLS transmitter including information if this data item is valid or not. The data item is placed in a FIFO preliminarily. The regulator circuit based on a regulation algorithm given in [6] generates a *send* signal, that is passed to toggle flip-flop and the read input of the FIFO where the data item is placed. The regulation algorithm generates a periodic signal to control the data flow at a rate of $min(f_T, f_R)$. Thus if the receiver is the faster module, the GRLS transmitter sends with a rate of its local clock. The toggle flip-flop is enabled by the periodic *send* signal and generates a *strobe* signal that is sent to the GRLS receiver besides the data item and the *valid* signal. The receiver samples all incoming data (and signals) at both the rising as well as falling clock edge. Thus the GRLS receiver may sample two valid data items in one clock cycle. But the receiver module cannot processes more than one data item per cycle hence a FIFO buffer is necessary. The FIFO only needs one cell due to the *average rate* property of the regulation algorithm, where the number of data items is limited to $d \leqslant K + 1$ in a time $K \cdot T_R$ with K as an integer. There are also three other properties given in [6] that were used to formally prove the regulation algorithm (for further details refer to [6]). The sampling of new data is enabled by the result of the analysis of the incoming *strobe* signal, which determines if it is safe to sample. It is safe to sample if the sample of the *strobe* signal at time t_i is different to the sample analysed a half clock cycle before. First the *strobe* signal has to be synchronized to the receiver's timing domain. After a delay T_w that is defined as $t_{su} + t_{ho} < T_w < \frac{T_H}{2} - (t_{su} + t_{ho})$ the *strobe* signal is synchronized by multistaged synchronizer, several cascaded flip-flops. The *strobed* signal (the delayed *strobe* signal) is sampled on both clock edges, the current sample is then (after synchronization) compared with the previous sample (arrived a half cycle earlier) by XOR gates. After comparison the signal is further delayed by a programmable delay line using a cascade of flip-flops. The length of the delay line is determined by $K \cdot N_T - N_S - 1$, where N_T is the multiplier of the transmitters frequency to match the global common clock and N_S is the number of stages at the synchronizer for the *strobed* signal and K is an integer. This forms the selector value for the multiplexer to determine at which point in the delay line the analysis result is grabbed and passed to the data sampler as enable signal (either for rising or falling edge). The sampled data item is then either directly passed to the receiver module or buffered in the 1-cell FIFO depending on which clock edge the previous data item was sampled (i.e. two data items are sampled in the same clock cycle by two consecutive clock edges). One can see the interface in Figure 4.5, it is based on the circuit from *Low-Latency and Low-Overhead Mesochronous and Plesiochronous Synchronizer* presented in 3.2.6 and 5.2.4 from [7] (see also Figure 3.10 and 5.6). So we conclude that the data flow is free from unwanted latency and metastability events. Metastable events only may take place at the input of the *strobe* signal, to ensure a high MTBF a *Brute-Force Synchronizer* with N_S stages is used. Notice that is might be problematic using a *Brute-Force Synchronizer*, because the clocks and thus the incoming events are not uncorrelated [3]. Further the data latency is not affected by the length of the learning phase (*strobe* analysis). Although the *strobe* signal shows if it is safe to sample the current data item is sampled with the *strobe* signal synchronized and analysed $K \cdot PC$ before, this is guaranteed by the *periodicity* property of the regulation algorithm. There is only an initial latency that affects the data transmission once after startup, which is defined as $t_s = t_i + K \cdot PC$, where t_i is the point in time when the *strobe* signal is analysed. The transmitter then sends every cycle so that the analysis flow (and data flow) is not interrupted, also in cycles in which the transmitter has no new data to send, in this case the *valid* signal is simply de-asserted and thus the receiver module does not process the incoming data. [5] also provides implementation details and a latency analysis. In the worst case using 90 nm technology the GRLS approach works with at most 1 GHz as global frequency (f_H). The latency is given as T_W in best case, and $(\lceil \frac{N_R}{N_T} - 1 \rceil) T_T + T_R + T_W$ in worst case.

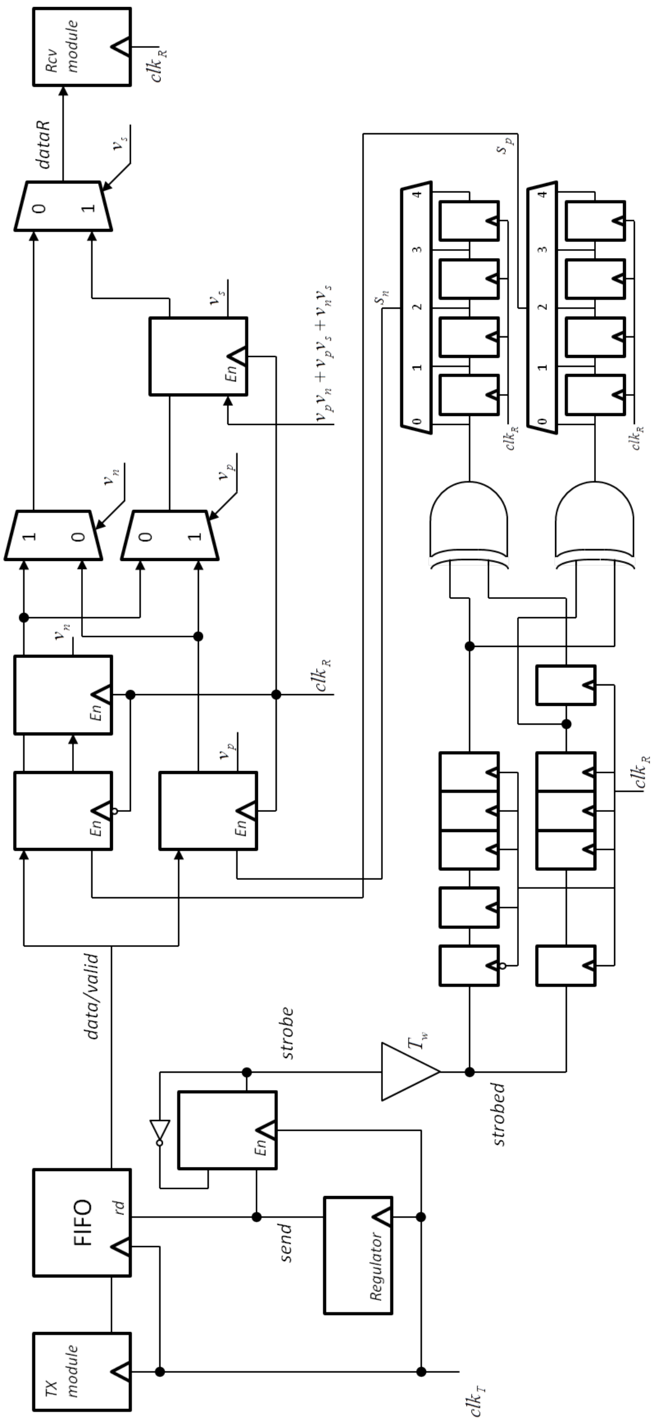


Figure 4.5: GRLS Interface Structure [5] [6]

CHAPTER 5

Plesiochronous Synchronizers

In contrast to the mesochronous synchronizers, treated in Chapter 3, *Plesiochronous Synchronizers* have to handle slight drifts in clock phases. That involves that the delay within the synchronizer needs to be adapted during operation to cope with changing relation in phase and to avoid duplicated or dropped data items. Within heterochronous/periodic systems, where the communication partners are clocked at different frequencies, synchronizers take advantage of the periodic nature of clock signals, to predict points in time where it is unsafe to transmit data. Due to different system clocks there is a need of flow control at the interfaces either at the faster or the slower clocked module, in order to avoid duplicated or dropped data items. There are different ways to accomplish flow control for example a backpressure mechanism (like in 3.2.7) or a common sending speed equal to the speed of the slowest module. Some mesochronous synchronizers can be used, but mostly require modifications like update circuits, null insertion or explicit flow control.

5.1 Basic Concepts

All the plesiochronous synchronizers presented in this chapter basically use the same concept. They synchronize a signal (transmitter's clock, resynchronization signal or strobe signal) from the transmitter to control a multiplexer that switches between a latch with the directly latched (unmodified) data item and a delayed version of it. The *Plesiochronous FIFO Synchronizer* uses a resynchronization mechanism to align the "read pointer" of the receiver to the "write pointer" of the transmitter, so that the receiver always samples the oldest value (more time for metastability resolution). The *Periodic Asynchronous Synchronizer* and the *Low-Latency Plesiochronous Data Retiming* approach directly influence the data stream by adding a delay upon the detection of a change of data within the exclusion region of the clock. At last the *Low-Latency and Low-Overhead Plesiochronous Synchronizer* works like the Low-Latency and Low-Overhead Mesochronous Synchronizer, but employs a continuous learning phase for the *strobe* signal that controls the input data latches (enable input) and further the used multiplexers.

As representative solution for plesiochronous synchronizers the *Low-Latency and Low-Overhead Plesiochronous Synchronizer* is chosen. As mentioned before, it is similar in construction to the *Low-Latency and Low-Overhead Mesochronous Synchronizer* with a difference in the strobe signal path and the continuously performed learning phase. Due to the continuous learning phase the synchronizer employs an infinite MTBF, in contrast to the other solutions. Note that the *Low-Latency Plesiochronous Data Retiming* approach uses a fixed delay and the *Plesiochronous FIFO Synchronizer* and the *Periodic Asynchronous Synchronizer* (only for predicted clock) employ a resolution time of one cycle only. Further the *Plesiochronous FIFO Synchronizer* has to wait for a cycle when a resynchronization of the read pointer it active, thus the throughput is limited, the *Low-Latency and Low-Overhead Plesiochronous Synchronizer* on the other hand supports maximal throughput. The latency of the chosen synchronizer is equal its mesochronous version, with the difference that it employs a continuous learning phase. Only the *Low-Latency Plesiochronous Data Retiming* approach employs a lower latency of three quarters of the cycle time, but on average the *Low-Latency and Low-Overhead Plesiochronous Synchronizer* achieves the best overall performance. Detailed descriptions of the prototypes and references for their comparative parameters can be found in the sections below.

5.2 Detailed Descriptions

5.2.1 Plesiochronous FIFO Synchronizer

The *Plesiochronous FIFO Synchronizer* [12] [28] is an extended version of the *Two Element FIFO Synchronizer* 3.2.4. The synchronizer is depicted in Figure 5.1. A reset logic for the pointer of the receiver (*rp*) is added to the two element (mesochronous) FIFO synchronizer. The resynchronization signal (*resync*) either lets the receiver pointer toggle with the local clock (inverter loop) or resets the pointer and resynchronizes it with the transmitter pointer (*xp*). In the latter case the current value of the transmitter pointer, delayed by an adequate margin, is set as new receiver pointer. This is neccessary to prevent that the multiplexer in the data path switches to the flip-flop input that is currently in its decay phase (the latest written flip-flop). When the receiver pointer is about to reset it is set to the current value of the transmitter pointer delayed by an adequate margin.

When using a FIFO synchronizer with more than two stages a phase-slip detector is required to determine the point in time at which the phase of the receiver pointer has to be adjusted. In the *phase-slip detection* logic (see Figure 5.2) the receiver pointer is controlled in such a way that it lags the transmitter pointer by a sufficient margin (in the case shown approximately two cycle plus the t_m to allow appropriate resolution time for the data within the *Brute-Force Synchronizer* formed by the FIFO. The resynchronization signal (*resync*) also controls a multiplexer within the *phase-slip detector* which switches between an incrementer loop and the delayed value of the transmitter pointer (*xp*). A comparator compares the inputs of the multiplexer, if the incrementer value of the receiver pointer is greater than the value of the transmitter pointer the comparator will signal that a duplicate will occur and the receiver pointer will not be incremented if the resynchronization signal is asserted. On the other hand if the receiver pointer is lower than the transmitter pointer it will skip a state and the comparator will signal a drop. Additionally it is possible to handle duplicates and drops by adding either *null* symbols to signal the absence of data or enable open/closed loop flow control.

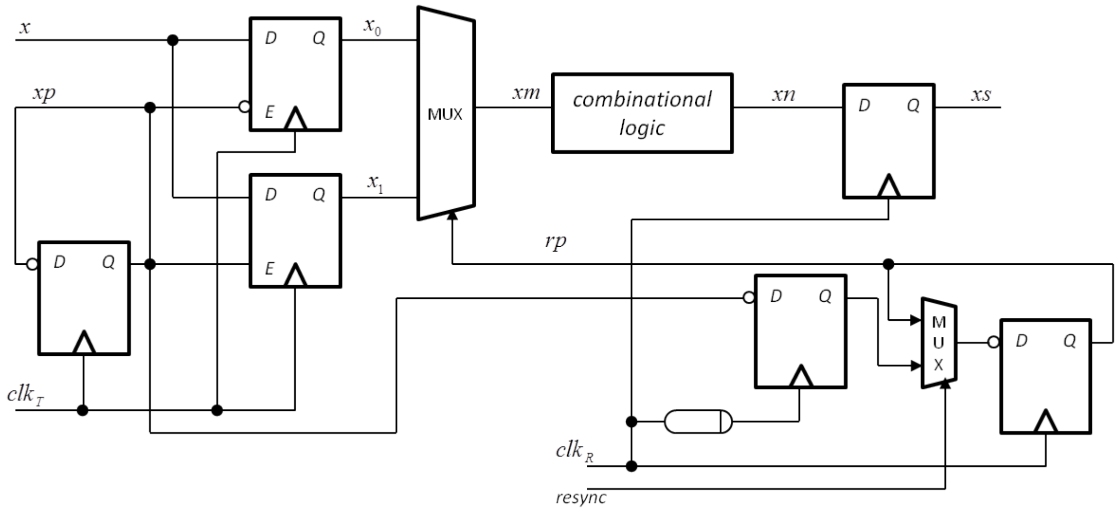


Figure 5.1: DSE [12]: Plesiochronous FIFO Synchronizer

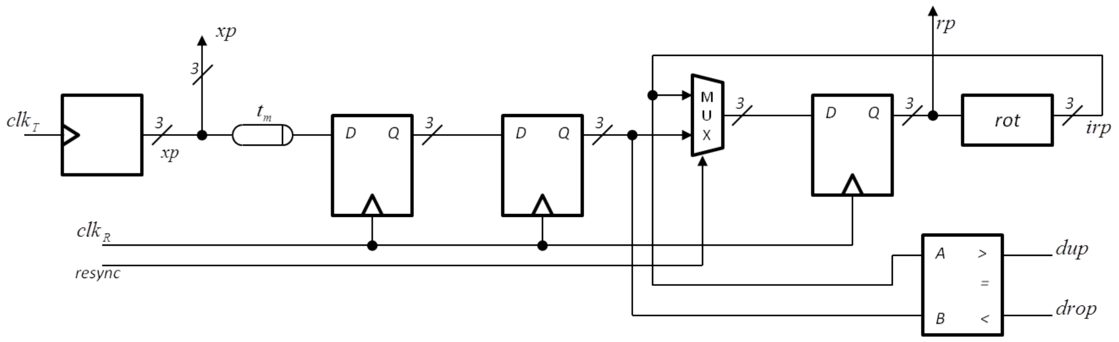


Figure 5.2: DSE [12]: Phase Slip Detector

5.2.2 Periodic Asynchronous Synchronizer

The *Periodic Asynchronous Synchronizer* [12] predicts the transmitter clock at receiver's interface and takes advantage of periodic nature of the used clocks to avoid synchronization failure. Some type of flow control will be required due to the difference in the clock frequency. A circuit to predict external clock is employed to "foresee" the transmitter's clock one clock cycle of the local clock ahead and to further generate a *keep-out* signal for the receiver pointer, one can see the synchronizer's structure in Figure 5.3. The clock predictor is based on a phase comparator, which sets a variable delay on the transmitter clock line upon the comparison of two consecutive clock edges of the transmitter clock. The predicted clock (*pxclk*) is interpreted like in a *Delay-line Synchronizer*, latched with a delayed local clock and synchronized with a second stage of latches, to let a possible metastable state decay. These two samples are then merged by an XOR gate to form a *keep-out* signal, which switches between the unmodified data item and a delayed version, i.e. it controls the multiplexer of a *Two-Register Synchronizer*). The interpreter has two paths, one with a higher delay than the other to determine if the input event (predicted clock) lies in the *keep-out* region of the local clock, which is indicated if the flip-flops latch different values. If the values are different and merged by the XOR gate the *keep-out* signal is asserted and the delayed data is passed.

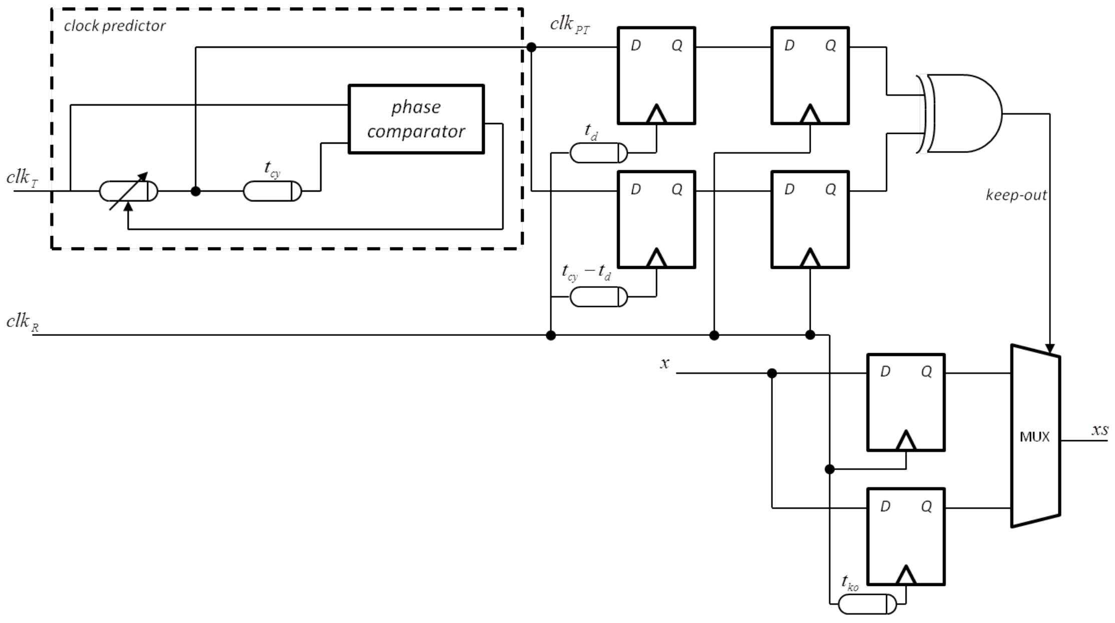


Figure 5.3: DSE [12]: Periodic Asynchronous Synchronizer

5.2.3 Low-Latency Plesiochronous Data Retiming

Data retiming is a synchronizer-avoidance method presented in [15] to interface data transmission within mesochronous and plesiochronous systems. More specifically, *data retiming* is the transmission of a data item from one timing domain to another. Each data item is sent within a so-called 'cell', that consists of an exclusion region where the data item changes or is not stable yet and a valid region where a data or non-data item can be safely sampled. It is used in communication networks as in network repeaters, parallel systems and so on, where several modules operate with the same frequency from different sources (this removes a single point of failure). With data retiming no handshake mechanism is required. Main focus is to keep up the one-by-one transmission of data items between the different timing domains. In a plesiochronous system the frequencies are nearly the same with a little deviation of $\pm\,\Delta f$. To avoid underruns (when receiver clock is faster) and overflows (when transmitter clock is faster) the transmitter sends data items at a lower rate, lower than the computation rate of the receiver. Furthermore the items in the stream produced by the transmitter can either contain data or non-data, the receiver distinguishes between them, and will only process data items. Thus the transmitter produces a stream at its clock rate (filled with both data and non-data), but data items occur only at receiver clock rate (operational rate) in the stream. The receiver (if it is faster than another module or finished computation early) must tolerate occasional delays waiting for the next data cell. Note that, in a communication network with several systems a transmitter must send data at the rate of the slowest receiver. So the transmitter always presumes that its clock is the fastest and send data at rate $f_d = f_t \dfrac{f_0 - \Delta f}{f_0 + \Delta f}$. If the sampling clock edge occurs within the exclusion region of a cell within a mesochronous system, the transmitted signal is simply delayed by half a clock period to get the edge to the valid region. A detection circuit is employed to detect if a sampling edge occurs within the exclusion region, if so a multiplexer switches between the original signal and the delayed signal.

Within a plesiochronous system the method is similar to the one from the mesochronous system with the difference that switching between original signal and its delayed version is allowed dynamically. The retiming circuit can be seen in Figure 5.4. To avoid missed or duplicated data items the switching is controlled by an automaton within the transmitter timing domain. The automaton only switches between original and delayed signal after both became valid and have the same non-data item. Since over- and undersampling only occurs when switching the input signal only non-data items are missed or duplicated. An extensive case distinction is given in [15] that shows the different scenarios what happens when the signal is switched (in both directions) once with a faster transmitter and once with a faster receiver. This approach has an average latency of three quarters a cycle time. When implementing a circuit for this method the used cell has to be divided into four pieces, each piece is related to a transmitter clock edge, the even numbered pieces to the rising edges and odd numbered pieces to falling edges. This is necessary to locate the exclusion region that straddles around a transmitter clock edge and further to construct the delayed version of the signal from the other three pieces (clock edges) and control the switching point between the two signals. Further the exclusion regions are sampled by the receiver clock to determine if switching is needed. These sampling outputs are sent back and synchronized into the transmitter domain to control the automaton. Note that the used synchronizer is off the critical path thus has no impact on the data movement. See Figure 5.5 for a FSM diagram of the signal switching. Further the data retiming can be used between system with frequencies that are integral multiples of each other, such that either $f_t \approx i \cdot f_r$ or $f_r \approx i \cdot f_t$, where i is an integer, must hold (for details on necessary adaptations see [15]). Either if the receiver is faster with additional sampling and occasionally creating a non-data cell (if sampling is not possible) or if the transmitter is faster by reducing the send rate and occasionally inserting a non-data cell. The achievements of this method are low latency, no synchronizers in the data path and true unidirectional retiming/signalling, that is communication without the need of any flow control.

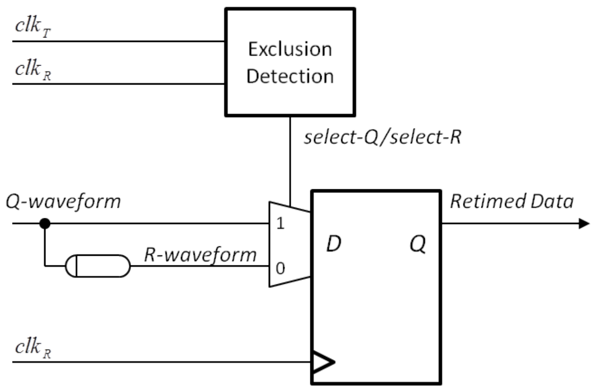
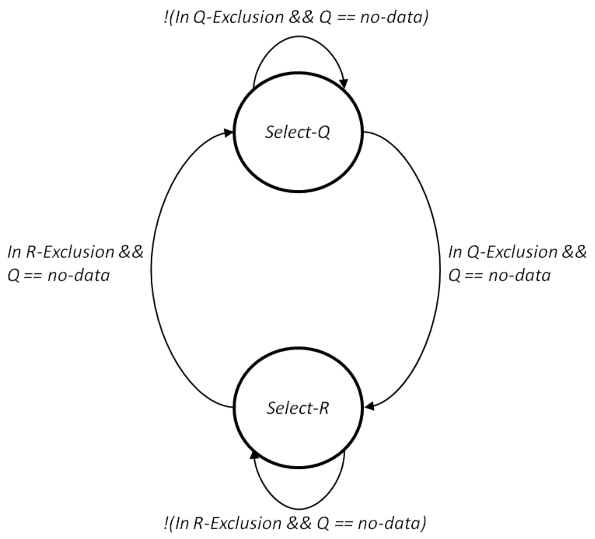


Figure 5.4: Low-Latency Plesichronous Data Retiming [15]: Mesochronous Retiming Circuit



!(In Q-Exclusion && Q == no-data)
Select-Q
In R-Exclusion && Q == no-data
In Q-Exclusion && Q == no-data
Select-R
!(In R-Exclusion && Q == no-data)

Figure 5.5: Low-Latency Plesichronous Data Retiming [15]: Select $Q \leftrightarrow R$ FSM

5.2.4 Low-Latency and Low-Overhead Plesiochronous Synchronizer

The low-latency and low-overhead plesiochronous synchronizer [7] is based on the low-latency and low-overhead mesochronous synchronizer, see 3.2.6. But it can tolerate drifts in clock rate ($\pm\Delta f$). To cope with these drifts an additional flip-flop for the plesiochronous synchronizer is required, thus it has four flip-flops over-head per data path. In contrast to the mesochronous interface it employs a *continuous learning phase*. Thus the *strobe* signal is sampled not only once after reset but continuously during operation (every cycle at the transmitter) to cope with the dynamically changing clock relation between the transmitter and the receiver. Hence it can also be used for mesochronous modules. The circuit is a little different from the mesochronous interface as one can see in Figure 5.6. In [7] formulas are given to determine the maximal frequency of the modules clocks.

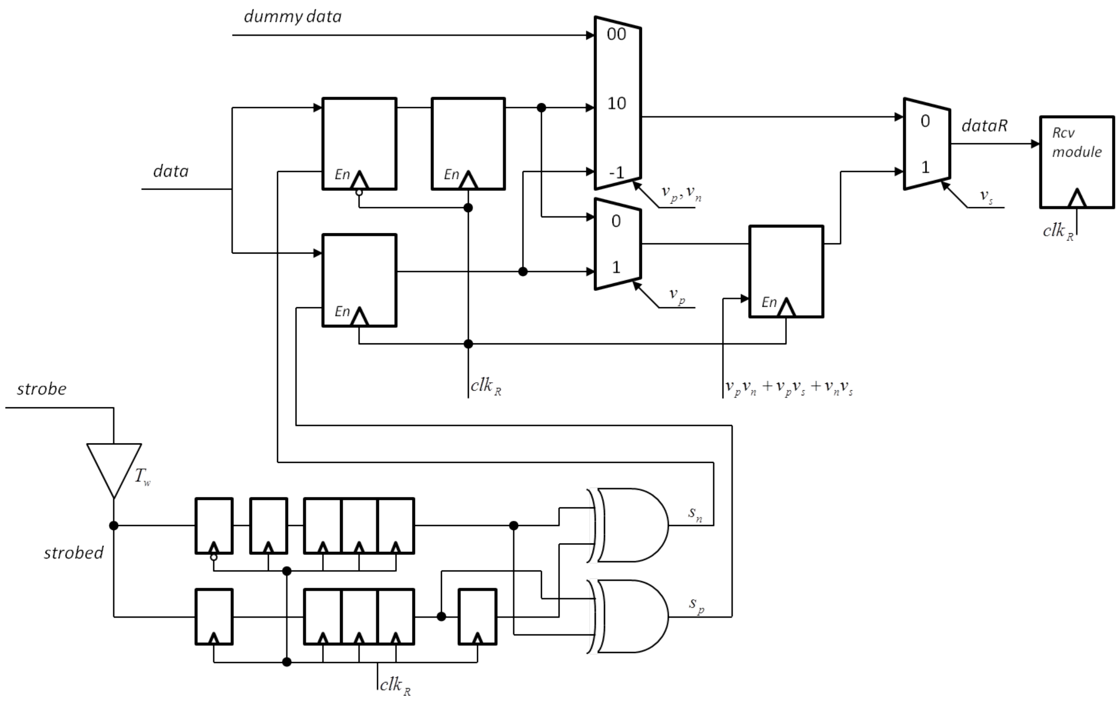


Figure 5.6: Low-Latency and Low-Overhead Mesochronous and Plesiochronous Synchronizer [7]: Plesiochronous Receiver

CHAPTER 6

Synchronizers for Systems with Uncorrelated Clocks

General Purpose Asynchronous Synchronizers are used to interface between clocked systems with unrelated frequencies, and also to synchronize sporadic incoming events into a clocked system (e.g. measurement readings from sensors).

6.1 Basic Concepts

The synchronizers discribed in this chapter use six different basic concepts. Starting with a group that synchronizes **handshake signals** like requests and acknowledgements (2- or 4-phase) between two mutually asynchronous systems. The *Two Flip-Flop Synchronizer*, the *Robust Synchronizer*, the synchronizer used for *Register Communication*, the *Fast Universal Synchronizer* and the *Micropipeline* belong to this group. The second concept is based on **pausible, stoppable clocking** or data-driven clocking or locally delayed latching (partial pausible clocking). The synchronizer in *High Rate Data Synchronization in GALS*, the *Stoppable Clock Synchronizer*, the *Asynchronous Interlocked Pipelined CMOS Circuit*, the *Asynchronous Wrapper for Heterogeneous Systems*, the synchronizer from *Pausible Clocking: A First Step Towards Heterogeneous Systems*, *SCAFFI*, the *Point to Point Interconnect*, the synchronizer in *Interfacing Synchronous and Asynchronous Modules Witin a Highspeed Pipeline* and *Using Stoppable Clocks to Safely Interface Asynchronous and Synchronous Subsystems* use the pausible clocking technique. The *Wagging Synchronizer* and the *BiNMOS Synchronizer* use **parallel paths** that are alternately written, either within a component or built from Jamb Latches, respectively (like the *Three Element FIFO Synchronizer*). Some of the presented synchronizers use a **FIFO** as interconnect and explicitly synchronize the write and read pointer of the FIFO to ensure that a data item is not read before entirely written. The *Asynchronous FIFO Synchronizer*, the *Robust Interfaces for Mixed-Timing Systems*, the *Pipeline Synchronization* mechanism, the *Even/Odd Synchronizer*, the *Four-Slot ACM* and the self-timed circuit version of the four-slot ACM, and the *STARI* approach (and its improvment) use FIFOs for communication. The *Two-Way Adaptive Prediction Synchronizer* and the *Periodic Synchronizer* from *A Solution to a Special Case of the Synchronization Problem* use a combination of **clock prediction** and **conflict detection** to generate a *keep-out* signal that controls the input latches at the receiver (by either delaying its clock or simply enabling it). The last concept forms a self-timed bus that implements handshaking and uses arbiter and is presented as *On-Chip Segmented Bus*. The synchronizers of each category are compared among each other in terms of MTBF, latency and throughput to find a prototype for each set. In the first group of synchronizers, which use handshaking to enable safe communication, the *Micropipeline* as bounded delay approach is picked. In general all synchronizers of this category are not very fast in terms of throughput, due to the FIFO structure of the *Micropipeline* the throughput depends on the fill level of the FIFO, but with a consistent data stream the throughput is higher than the other solutions. The only drawback is that the *Micropipeline* suffers from a high initial latency (as every other approach using FIFOs). The synchronizers of the next category use the *pausible clocking* technique to interface between mutual asynchronous systems. Among these various solutions the *Point to Point GALS Interconnect* is elected as prototype to represent its basic concept. The *Point to Point GALS Interconnect* employs a classical clock pausing mechanism using an arbiter that halts the clock (in a low phase) upon receiving a request signal from the transmitter. A possible metastable state at this point is quietly resolved by the arbiter. The solution from *Using Stoppable Clocks to Safely Interface Asynchronous and Synchronous Subsystems* is quite similar to the *Point to Point GALS Interconnect*. The *Point to Point GALS Interconnect* employs an infinite MTBF (due to the arbiter), and transmits a data item per handshake (FIFO buffering). The latency of the interfaces in this category is depending on the duration of the handshaking and thus fairly equal. Summing up the synchronizers using plausible clocking employ quite similar characteristics, the *Point to Point GALS Interconnect* is picked due to its simplicity of concept (represents the basic idea). In the category of parallel staged synchronizers the *Wagging Synchronizer* is chosen. It employs a significantly higher MTBF than other synchronizers, in [1] an example is given, while the *Wagging Synchronizer* employs a MTBF of 2.66 years at a rate of $2.5\ GHz$ (and $511\ ps$ resolution time) a *Two Flip-Flop Synchronizer* employs a MTBF of only 49.6 mins using edge-triggered flip-flops. The *BiNMOS Synchronizer* employs a MTBF of $6 \cdot 10^{34}$ years with 3 cycles settling time at $100\ MHz$. The *Even/Odd Synchronizer* is picked as prototype for synchronizers that use a FIFO as interconnect and synchronize the access pointers. It employs an arbitrarily small probability of synchronization failure and a very low latency of $0.5 + x$ cycles in average case, where x is the keep-out region of the registers. The throughput widely depends on the used FIFO. Since these solutions each employs a FIFO as interconnect the throughput can be seen as equal, the more crucial factor here is the latency. In these terms the *Even/Odd Synchronizer* employs the lowest average latency. From the basic concept of using collision detection the *Periodic Synchronizer* is taken as prototype. Either of the solutions move the synchronization away from the data path to the employed conflict detector. In contrast to the *Two-Way Adaptive Prediction Synchronizer* the *Periodic Synchronizer* employs a shift register (i.e. Brute-Force Synchronizer) and thus guarantees a resolution time of three clock cycles, while the former employs merely a half clock cycle which results in a MTBF of only 10^{16} years at a rate of $125\ MHz$, hence too little for modern system using fast clocks. On the other hand the used shift register in the *Periodic Synchronizer* employs a MTBF of 10^{204} years using a clock rate of 200 MHz [27]. For further details on the functionality and specifications of the chosen prototypes see the descriptions in the following section (*Micropipeline*, *SCAFFI*, *Point to Point GALS Interconnect*, *Wagging Synchronizer*, *Even/Odd Synchronizer*, *Periodic Synchronizer*, *GRLS*, *Segmented Bus*). Notice that the results for the comparative parameters are gathered from the original resources (see the detailed sections for references).

6.2 Detailed Descriptions

6.2.1 Two Flip-Flop Synchronizer

The *Two-Flip-Flop Synchronizer* [27] does not synchronize the data stream, instead it synchronizes *request* and *acknowledgement* signals and implements a handshake protocol. So the transmitter and receiver can safely agree on one point in time to transmit a *bundled data* item (namely a point where the transmitter holds new data at its output and the receiver is ready to latch it). A block diagram of the *Two Flip-Flop Synchronizer* can be seen in Figure 6.1, where each communication partner employs two flip-flops, that are clocked with its local clock, as input interface. The *Two Flip-Flop Synchronizer* can be used with various principles [27]. First as "push" synchronizer, where the request is sent from transmitter to receiver to signal that new data is available, further as "pull" synchronizer, where the request for new data is transmitted from receiver to transmitter, or in either way as "push-pull" synchronizer. Further it can be used as a "control-only synchronizer". In [27] the MTBF for the *Two Flip-Flop Synchronizer* is calculated, based on the assumption that at an input flip-flop in a system without synchronizer that is running at 200 MHz with new data on every tenth clock cycle 2 metastability events occur within a millisecond. Two flip-flops in cascade enable a MTBF of 10^{204} years, with a third flip-flop the *conservative* synchronizer is formed and increases the MTBF to 10^{420} years.

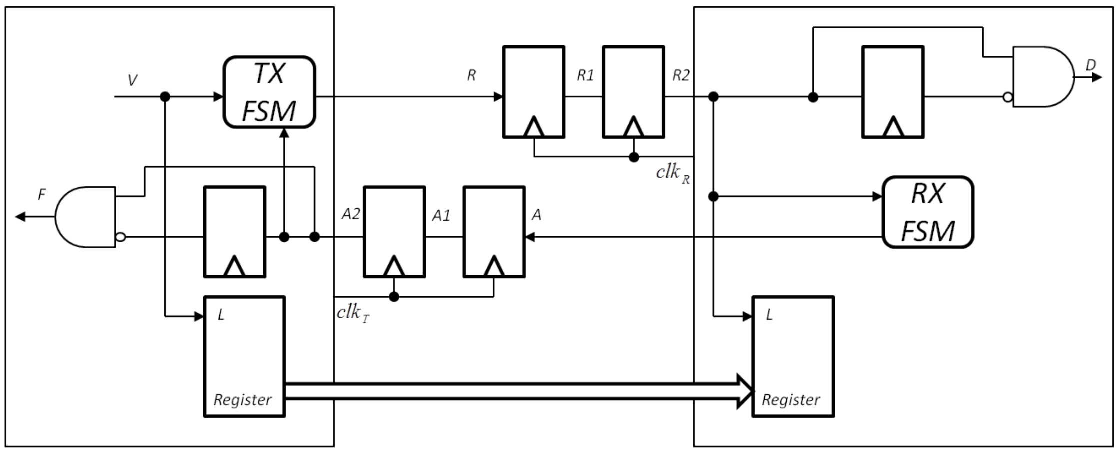


Figure 6.1: Two Flip-Flop Push Synchronizer [27]

6.2.2 A Robust Synchronizer

The Robust Synchronizer in [59] uses the Jamb latch synchronizer [16] and improves it to provide a stable value of the metastability time constant τ without increasing the power consumption. The Jamb latch synchronizer is simply a *Brute-Force Synchronizer* or *Two-Flip-Flop Synchronizer* that uses Jamb latches. The circuit of a Jamb latch can be seen in Figure 6.2. Using Jamb latches for a synchronizer results in a reduction of supply voltage of about 50%, which further results in an increase of 100% of the resolution time τ [59]. The Jamb latch is extended by two p-type load transistors (A) to maintain the current supply during metastability and by two feedback transistors (B) to hold the state of the latch, in Figure 6.3 one can see the circuit of the robust synchronizer. By using a metastability filter between the two nodes within the latch it is possible that the current is only increased during metastability.

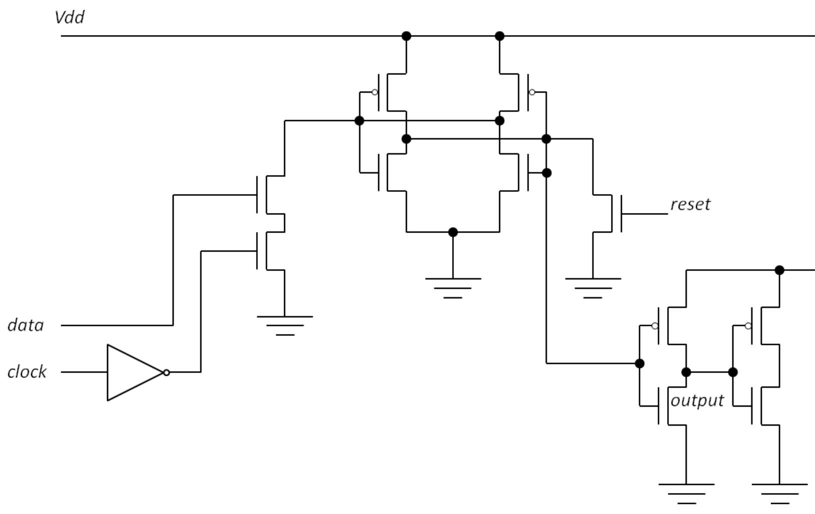


Figure 6.2: Robust Synchronizer [59]: Jamb Latch Circuit

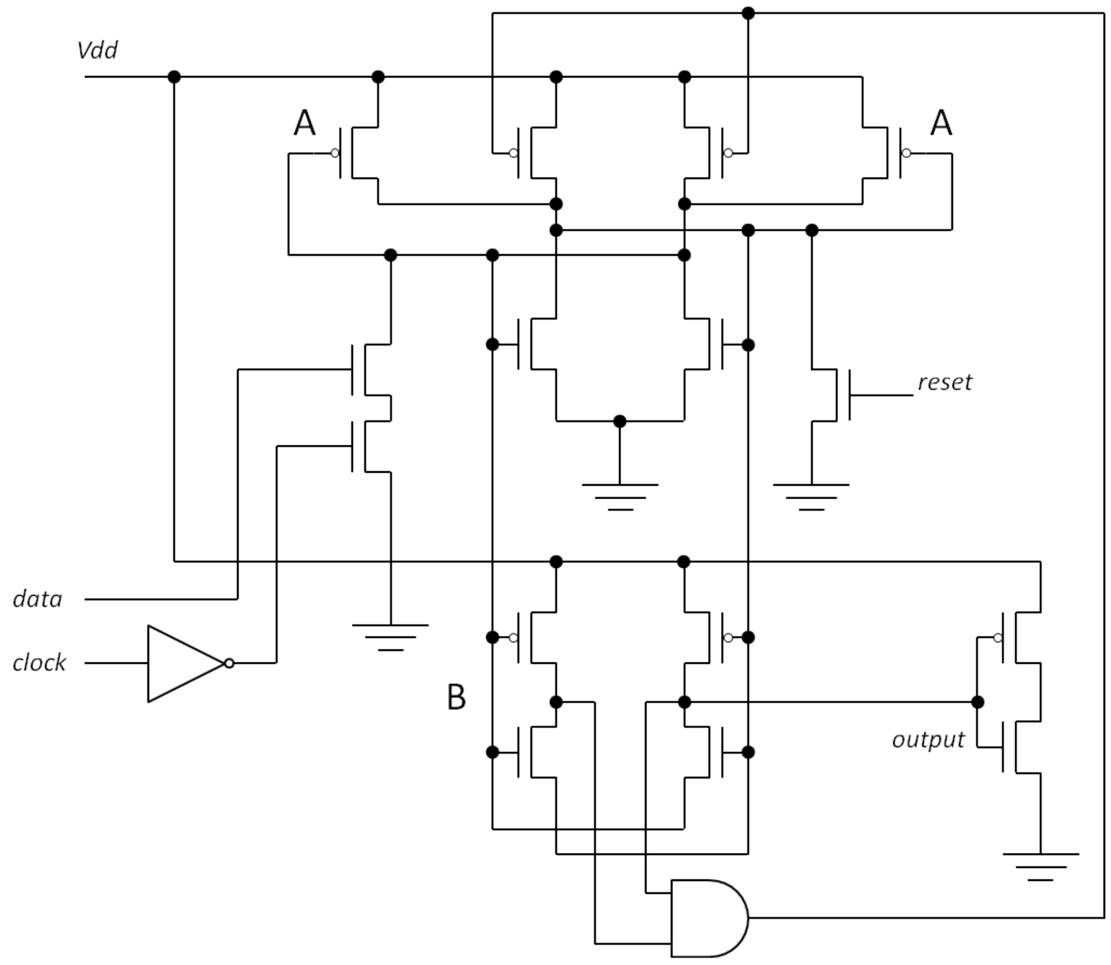


Figure 6.3: Robust Synchronizer [59]: Robust (Improved) Synchronizer

6.2.3 Register-Communication Between Mutually Asynchronous Domains

In [32] a communication register is presented that supports non-blocking communication between two mutually asynchronous modules (clocked or self-timed). It is used as a wrapper circuit around modules to enable communication between the clocked module and the asynchronous bus. The write and read accesses to the register are never paused or held up, hence data may get duplicated or lost. Note that a write access is destructive and on the other side a read access is non-destructive. Register communication is usually used in terms of command and status exchange, in combination with a completion indication (interrupt or status register). The concept is based on the two-flip-flop synchronizer, but extended to support more than single-bit communication. A formal description and verification of the functional requirements (correctness and liveness) are given in [32], but omitted in this context. Since this approach supports clocked as well as self-timed modules there are four types of communication registers, where self-timed modules use a 4-phase handshake. The clock signals of the synchronous modules are only used as event generators by the communication register to synchronize the handshake signals. The ports of the communication register (write, read) are passive, thus the module or environment connecting to a port initiates the activity (write, read) using a request signal. Each register has data and request inputs and data and acknowledgement outputs. The data at the input must be valid when the request is asserted. When the acknowledgement signal is asserted the output data is valid and stable until the next rising edge of the request signal. Such a register can be written in two different ways, which differ in the duration of validity of the input data. First with an *early-write* register the input data is only valid until the acknowledgement is asserted, on the other hand the input data of a *broad-write* register is valid until the acknowledgement is de-asserted again. Within an early-write register flip-flops clocked by the request signal are used. The broad-write register uses latches that are transparent with the request asserted. Figure 6.4 shows the communication register that can be used to interface two modules which use a handshaking protocol. It consists of a broad-write register and an arbiter. The arbiter controls the access to the register by arbitrating the request signals (write, read). This register guarantees that a read value is always the latest written value.

If the communication register interconnects a module within a clocked domain with a module within a handshaking domain there are two cases to distinguish. On the one hand there is the case where the module within the handshake domain is the transmitter and on the other hand where the clocked module is the transmitter. Figure 6.5 shows these two communication registers. Both employ an 'up-edge' component instead of the simple arbiter. The 'up-edge' component consists of two sequential wait components, see Figure 6.6. A 'wait' component is either a Muller C-Element or an arbiter. Both have an enable and a request input and an acknowledgement output. Note that the output of the arbiter that is related to the enable input is ignored, and that the enable input is inverted. The first 'wait' component within the 'up-edge' component is provided with the inverted clock signal, used as enable input, and the second one with the original clock signal from the clocked module. The 'up-edge' component delays the request input within the handshake domain only for a 4-phase handshake until a rising clock edge occurs, thus is synchronized with the clocked module. This is done by the two sequential 'wait' components. If the input request is asserted the first wait component stalls until the clock is low, then asserts an intermediate acknowledgement signal that is passed to the second 'wait' compo-

nent, which further asserts the acknowledgement output on the next rising edge of the clocking signal. If the clock frequency is very slow the handshake response time has to be bounded. This is done by adding another register to the interface and a repeater that involves the handshake for the passive ports (write, read). For further details refer to [32]. Additionally [32] provides low-power registers, here the input and output of an employed register is compared. The output of the comparator controls a 'wait' component between the repeater and the 'up-edge' component. The 'wait' component delays the request, generated by the repeater, until the compared values are different. For a detailed description and block diagram see [32].
The last type of communication register connects two clocked modules. This communication register can be realized in two ways. Either with one control loop or two symmetric control loops, where the first one supports higher clock frequencies and the second one provides a smaller latency. Both employ two registers, the one control loop approach an early-write register at the sender side and a broad-write register at receiver side. The solution using two control loops employs an early-write register on either side. Further both employ each two 'up-edge' components, to synchronize the request into clock domain of the respective side. The circuit with one control loop employs one repeater that generates the request signal for the register of the write port. Note that the request signal for the read register is the acknowledgement output of the write register. The approach with two control loops employ two repeaters that generate the request for the write and read register. The employed arbiter only passes one request into the system at a time. The approach of using communication registers for mutually asynchronous modules is not very fast in general in terms of throughput [32].

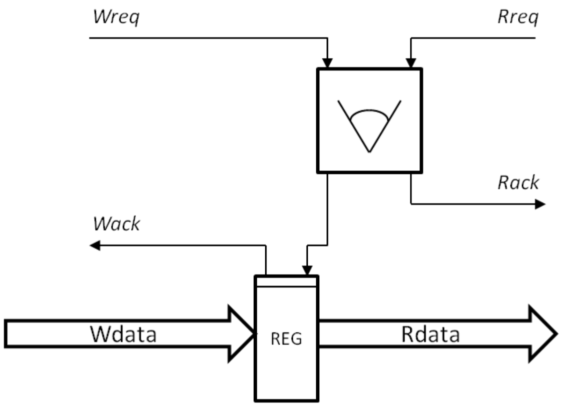


Figure 6.4: Register Communication [32]: Handshake-to-handshake Register

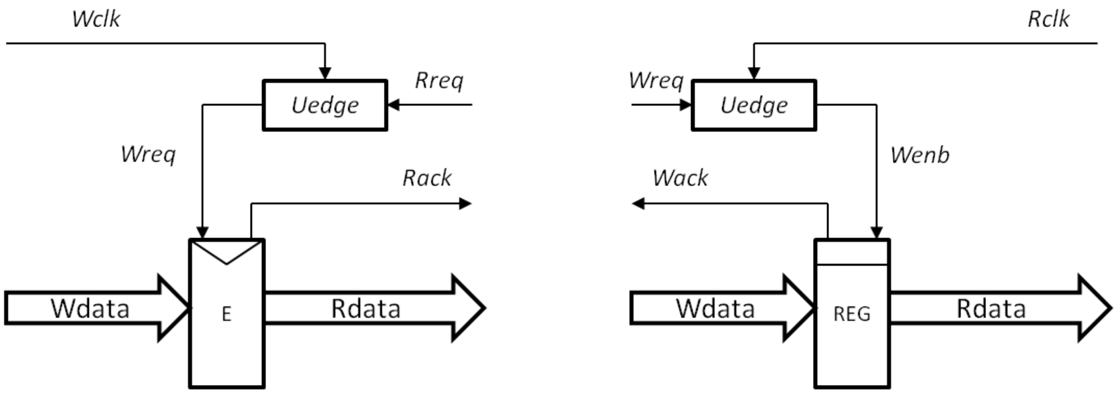


Figure 6.5: Register Communication [32]: Basic Design

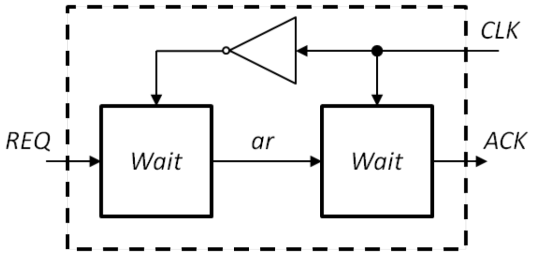


Figure 6.6: Register Communication [32]: Four-Phase Up-Edge Component

6.2.4 Fast Universal Synchronizer

In [20] the *Fast 4-Phase Synchronizer* is presented, which is based on the *Simple 4-Phase Synchronizer*, which is basically a *Brute Force Synchronizer*. The *Simple 4-Phase Synchronizer* employs two flip-flops to sample asynchronous 4-phase handshake signals (acknowledgement and request), see Figure 6.7. These two flip-flops enable one clock cycle for metastability resolution. To stretch resolution time for metastability additional flip-flops in row can be inserted. On the other hand to shorten the resolution time, i.e. shorter than the duration of a half clock cycle, a falling edge flip-flop can be employed in first place. Such an universal synchronizer (between mutually asynchronous domains) does not take advantage of knowledge about the clock relationship, i.e. it is not optimized for a specific system. The *Simple 4-Phase Synchronizer* forms the base for the *Fast 4-Phase Synchronizer*, which simply increases the number of used flip-flops to extend the resolution time and further support faster clocks. A 2-phase handshake protocol improves the synchronization data rate significantly. This requires addtional control logic, see Figure 6.8. The time needed for metastability resolution is reduced by the delay of an XOR gate while maintaining the same MTBF [20], compared to the *Fast 4-Phase Synchronizer*. The *Fast 2-Phase Synchronizer* operates as follows: it waits for data, whose arrival is indicated by rising *VI* and *SNT* signal. A request is sent to the receiver (event on *REQ* signal), and if the preceeding handshake is done the transmitter output registers (*REGD/REGV*) are enabled and will send out data on the next rising edge of transmitter clock. When the receiver is ready to latch new data it rises the *READY* signal. This generates a falling edge on the *VO* signal that indicates that new data is arriving and needs to be processed. Further the *REGR* register is enabled latching the incoming data (falling edge of *RXE* signal). The falling edge causes *VO* to rise again and with the rising edge of the *RXE* signal (indicating that latching is finished) the acknowledgement is sent to the transmitter (event on *ACK* signal). Note that the *READY* signal may be delayed by an extra clock cycle due to metastability resolution. The acknowledgement disables the transmitter's output registers (REGD/REGV) and asynchronously resets request and stays disabled until the handshake is finished (when both *VI* and *SNT* fall again). The data cycle depends largely on the slower clock if the fast universal synchronizer is employed between mutually asynchronous systems, i.e. two cycles from each domain with the *Fast 2-Phase Synchronizer* and three cycles from each domain with the *Fast 4-Phase Synchronizer*. Between mesochronous systems the minimal data cycle is four clock cycles in the worst case [20], thus a maximal throughput of 0.3 words per transmitter cycle is achievable.

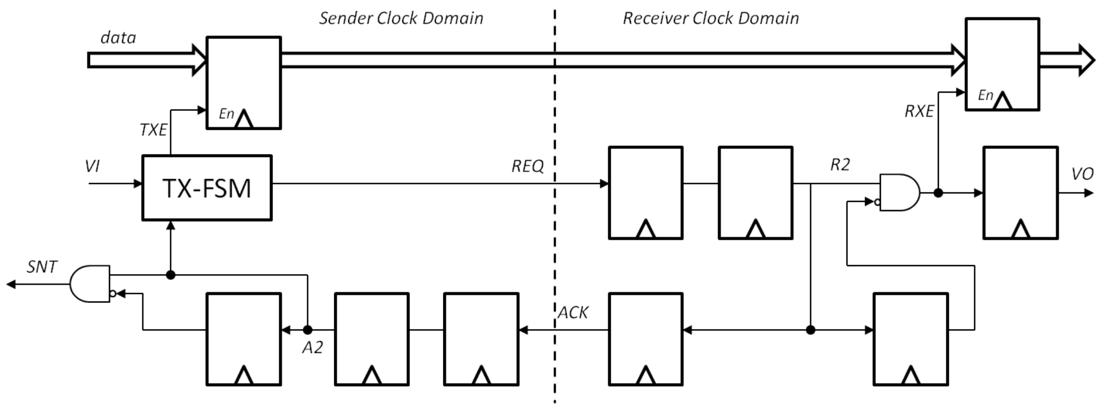


Figure 6.7: Fast Universal Synchronizer [20]: Simple Four Phase Synchronizer

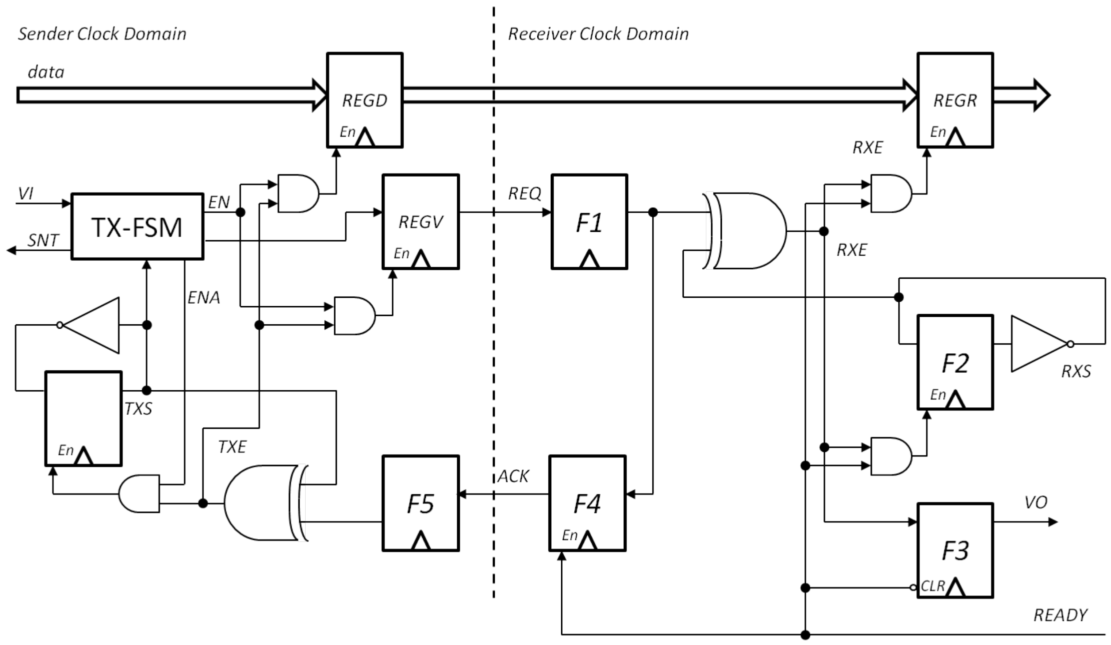


Figure 6.8: Fast Universal Synchronizer [20]: Fast Two Phase Synchronizer

6.2.5 Micropipelines

A *micropipeline* [53] implements a request/acknowledgement handshake mechanism using an elastic pipeline. It forms a closed-loop control for the data stream between transmitter and receiver and uses a 2-phase handshake (transistion signalling (events), non-return-to-zero (NRZ)). A bundled data approach is used in combination with the micropipeline where the data wires and request signalling are treated as a bundle. In Figure 6.9 a string of Muller C-Elements employed with inverters (in reverse signal path) and delay elements (to avoid race conditions between data and request) forming a micropipeline with four stages and computional logic between them is depicted. Each stage consists of a Muller C-Element with one input as request input and the other one as acknowledgement input, latter is the inverted request of the successor stage or the acknowledgement output of the receiver. Each request line employs a delay element that correspondes to the propagation delay of the logical cloud of each stage to avoid a race between request and data (requests should be forwarded faster than a data item). This bounded delay is employed to match the transmission of request and data, note that the data line delay has to be smaller than request line delay. The logic is guided by event-controlled registers in each stage, which are controlled by the signals *capture* and *pass*. Data is latched by a register when the *capture* signal is changed, the *capture* signal is the output of the Muller C-Element. When a register finished latching data the *capture-done* signal is asserted, which is routed back to the register of the previous stage as the *pass* signal which is equivalent to the acknowledgement. An empty pipeline is indicated by all stages having the same state, within a full pipeline the states of the stages are alternating. Initally the micropipeline is empty, all stages have the same state. When the transmitter asserts *R(in)*, *A(1)* is low (due to an empty pipeline) hence the output of the Muller C-Element rises, propagating the request through the delay to the next stage and further triggers the register to latch *D(in)*. Note that each request (and data item) propagate through the entire pipeline to the last (empty) stage. The state of *A(1)* changes, thus allowing the first stage only a falling transistion of *R(in)*. When the register has finished capturing data the *capture-done* signal is asserted and the next stage activated, in the meanwhile the logic of the stage 1 processes the new data. The procedure repeats in each stage until the receiver samples the processed data or the pipeline is full.
The concept of the micropipeline is further reused for other functionalities. First a micropipeline can be used with asynchronous stages forming a FIFO ring, as presented in [22]. Such a FIFO ring can be used for clock recovery and clock generation and distribution. Further the micropipeline is used for a ripple-through FIFO that is presented in [55], that can be used to interface the modules of a GALS system. FIFOs are commonly used within GALS systems in numbers, thus seize a large part of chip area. To minimize the chip area used by the interfacing FIFOs the design is changed from using SRAM stages to an asynchronous micropipeline [53]. Thus each stage employs a latch for exactly one data item and a control circuit.

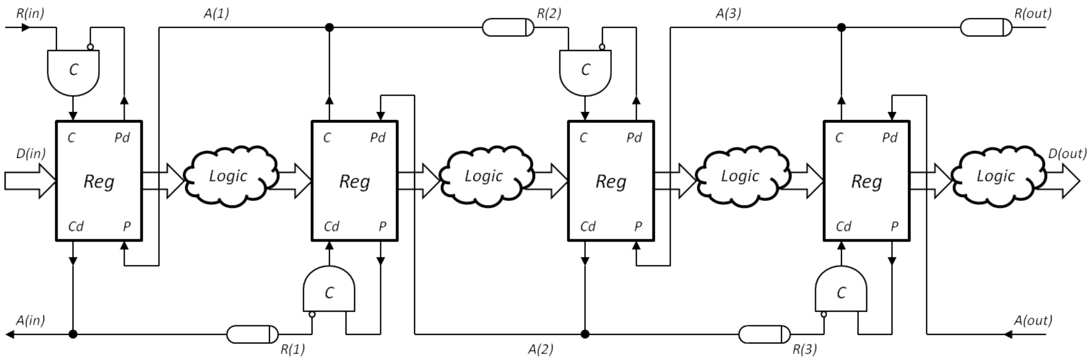


Figure 6.9: Micropipeline [53]: Structure

6.2.6 High Rate Data Synchronization in GALS SoCs

A clock stretching approach for a synchronizer used for a GALS system is presented in [18]. The presented synchronizer uses locally delayed latching (LDL) to interface mutually asynchronous modules within a GALS system. In contrast to other synchronizers used within GALS system this LDL approach does not require pausible clocking for the entire module (restricted to a single input latch only). This eliminates the potential threat of failure caused by delays from the clock distribution network in systems that uses pausible clocking [18]. One can see in Figure 6.10 that a LDL input port synchronizer employs an asynchronous controller that controls the input port latch and how the high phase of the local clock of the locally synchronous module is shortened. In the LDL approach the clock is not paused but the rising edge is delayed (note that the falling edge is unaffected by the delay). The rising clock is thus “moved” in case a conflict between an incoming request and a rising edge is imminent, the conflict detection is done by a mutual exclusion element within the asynchronous controller. To maintain a sufficient amount of time for metastable state to resolve, even if the edge gets moved, the controller must assure a high phase that is long enough to clock the registers in the module. In [18] a high phase of 43 gate delays (FO-4 inverters) and a period time of $T = 100\tau$ (where $10^{-11} < \tau < 10^{-10}$ seconds) for a MTBF of 10^4 years is the minimum length (too short for modern systems), hence at least one half of a symmetric clock cycle should be intended for resolution.
Three possible implementations are shown in [18]. First the ’Decoupled Input Port’, shown in Figure 6.11 synchronizes an asynchronous incoming request. The incoming request is forwarded to the Muller C-Element and as soon as the previous transmitted data is processed (valid signal is low) the request asserts the mutual exclusion element input to delay the rising edge of the local clock. This is possible because the clock forms the second input to the mutual exclusion element. If the request wins an imminent conflict it is forwarded/looped back to the asynchronous input control and directed through the latch-matched delay. After the delay a signal is asserted that activates the latch (gather new data) and sets the valid signal. The valid signal is used to signal that a new data item has arrived and thus prevents *write-after-read* hazards. Then an acknowledgement is generated and sent back to the sender, which in turn de-asserts the request. This leads the Muller C-Element output to fall and the clock is forwarded to the register in the module to sample new data from the latch. As second implementation a ’Decoupled Output Port’ is presented. A detailed block diagram of the circuit is given in [18], the difference to the ’Decoupled Input Port’ is that instead of the incoming request an incoming acknowledgement must be synchronized. The last implementation variant is a simplification of the ’Decoupled Input Port’. It removes the asynchronous input control and moves the delay element to the acknowledgement ouput. Further the input request is directly connected to the Muller C-Element. The ’Simple Input Port’ is depicted in Figure 6.12. Further details on simulation results and verification results are given in [18], as well as a comparison to *classical* GALS interfacing solutions.

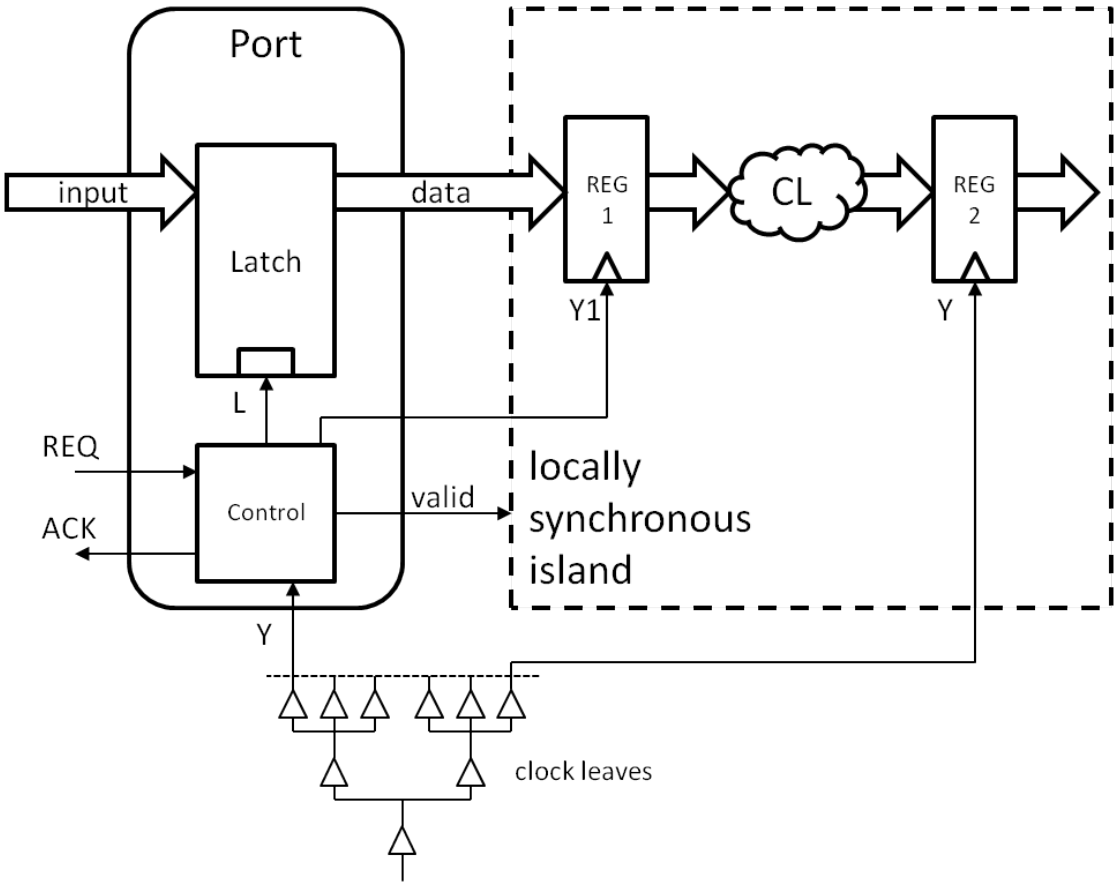


Figure 6.10: High Rate Data Synchronization in GALS SoCs [18]: LDL Circuit

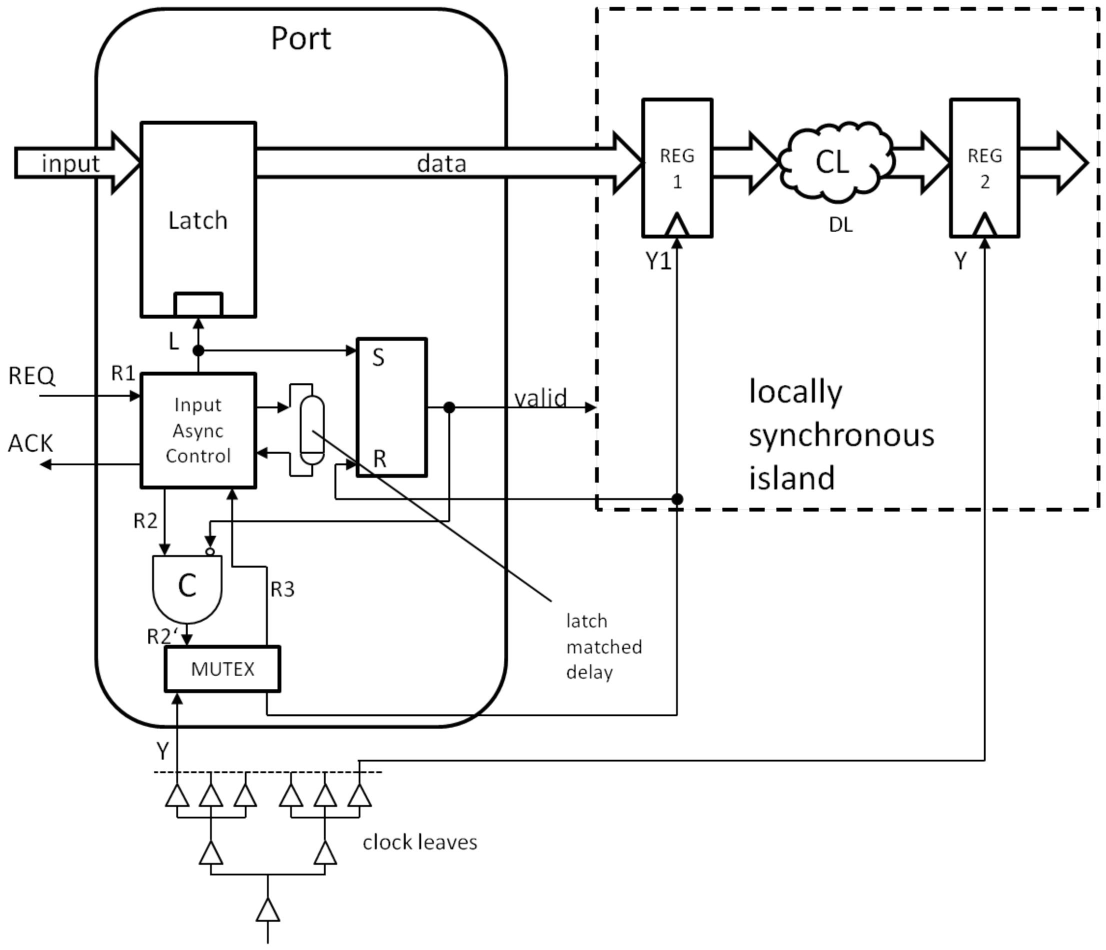


Figure 6.11: High Rate Data Synchronization in GALS SoCs [18]: GALS decoupled input port

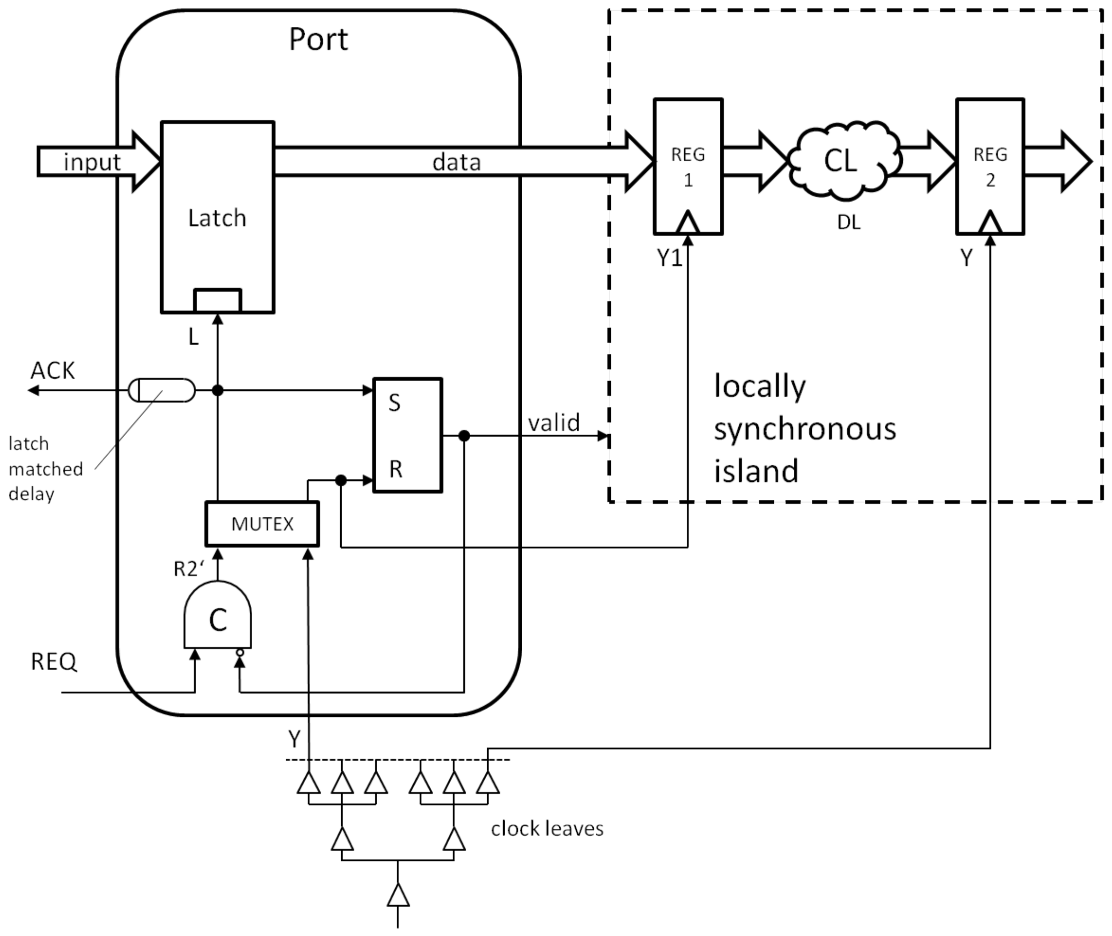


Figure 6.12: High Rate Data Synchronization in GALS SoCs [18]: Simple input port

6.2.7 Module with Stoppable Clocks

A *Module with Stoppable Clocks* is presented in [12] and is shown in Figure 6.13. On the arrival of a new asynchronous event a clock signal is generated and stopped while waiting for the next event, this is called data-driven clocking. The module using such a mechanism is clocked, only the used interface is asynchronous. This eliminates the synchronization delay and probability of synchronization failure. A data transmission starts with a request (*rin*) that is sent bundled with the input data to the receiver's interface. The asynchronous state machine (ASM) asserts the *go* signal enabling the latch to sample new data at the input (*datain*). Further the rise of the *go* signal causes the circuit (a ring oscillator) to generate the stoppable clock. The *done* signal is de-asserted to continue the operation for another clock cycle. Additionally the acknowledgement for the transmitter is sent (rising *ain*), thus the transmitter de-asserts the request (*rin* falls). After the input is sampled and the computation finished the *done* signal rises again causing the *go* signal to fall, which further stops the clock signal. Note that the delay of stopping the clock signal has to be shorter than the delay of the delay-element in order to avoid glitches on the clock line. The request signal (*rout*) for the receiver module is now asserted to signal new data at the output (*dataout*), which causes the acknowledgement signal (*ain*) for the transmitter to fall again. From this point the transmitter is allowed to initate a new data transmission (assert *rin*). Note that if a new request is sent to the receiver after a data transmission, the *go* signal is delayed as long as the request (*rout*) for the receiver has not be acknowledged (rising *aout*). Thus the data transmission is finished with an acknowledgement from the receiver module. Further notice that the request signal for the receiver (*rout*) and the acknowledgement signal for the transmitter (*ain*) has to be explicitly synchronized to the local timing, respectively, to avoid metastable states at these inputs (e.g. a *Brute-Force Synchronizer* may be used).

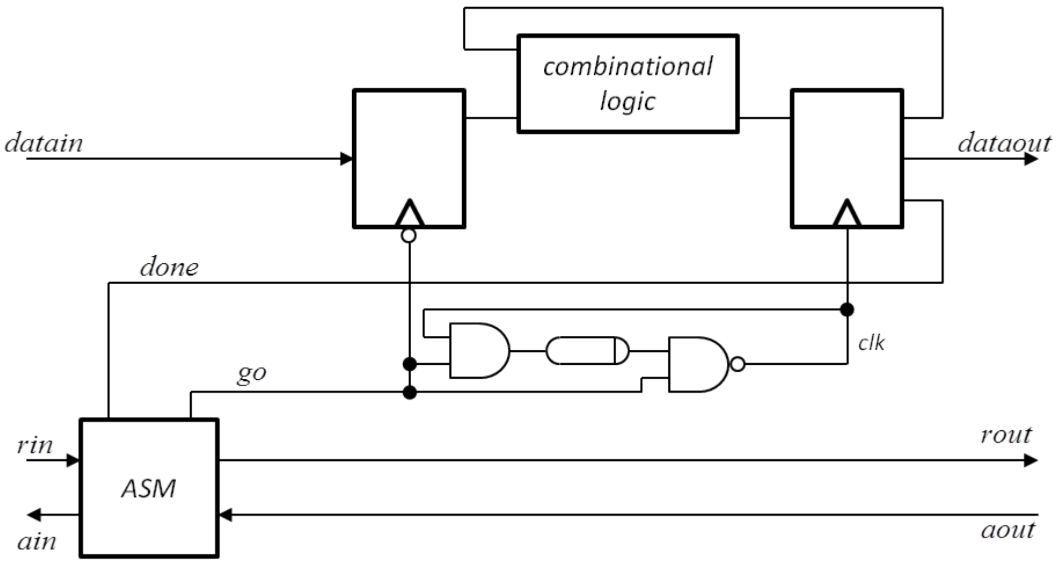


Figure 6.13: Stoppable Clock Synchronizer [12]

6.2.8 Asynchronous Interlocked Pipelined CMOS Circuits

The interlocked pipelined CMOS (IP CMOS) approach [44] is a further solution for an asynchronous, data-driven clocking technique. Each module that is interlocked uses an acknowledgement input and data and valid output to send data. A strobe circuit (see Figure 6.15 and further 6.16) is an input control circuit that is used to generate a clock-enable signal such that a data-driven clock is generated only upon receiving new data (closing the switch). In particular after a strobe was generated all valid signal inputs are low, as well as the clock-enable output (see Figure 6.14), when the first valid signal input rises the clock-enable output rises too, and falls again when all risen valid signal inputs have fallen again. Note that the clock generation is done in the module (receiver) and is not further mentioned in [44]. The acknowledgement signal rises with the valid signal input and falls again upon the rising edge of the clock-enable output. The generated clock is delayed due to the used handshaking mechanism, but running at $4.5\ GHz$ in best-case conditions ($3.3\ GHz$ under typical conditions). For further details on measurement results refer to [44].

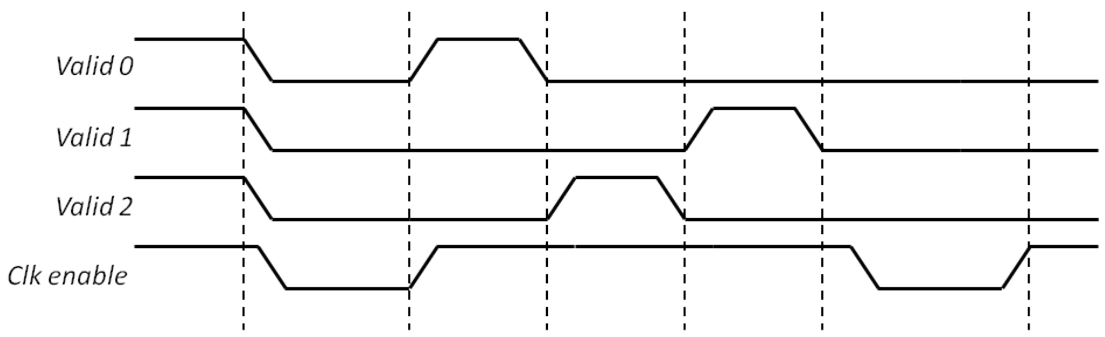


Figure 6.14: Asynchronous Interlocked Pipelined CMOS Circuits [44]: Strobe circuit unique "AND" function

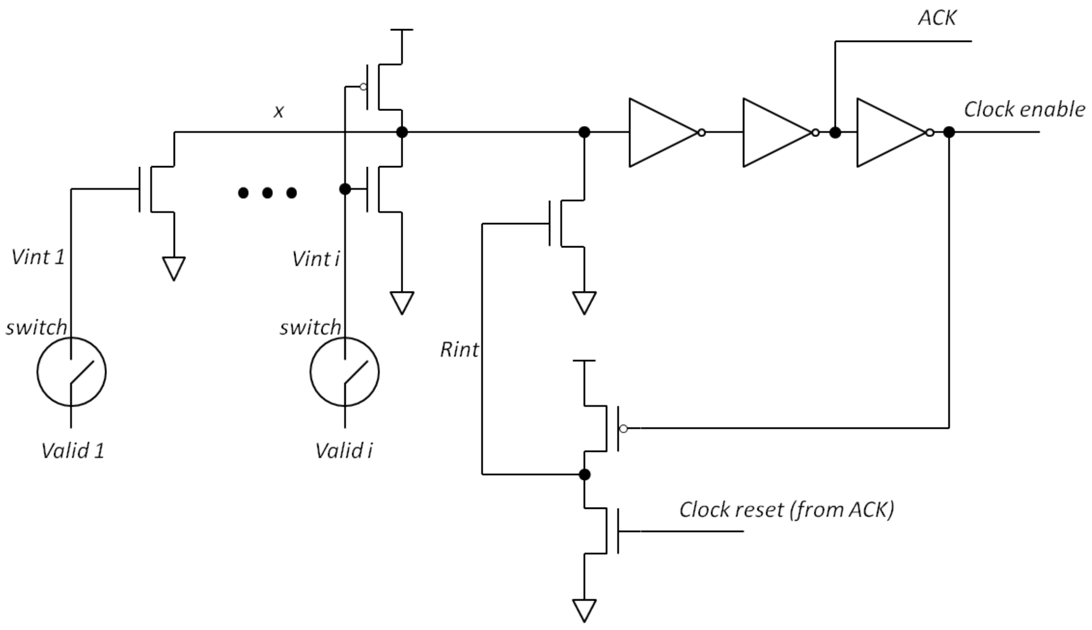


Figure 6.15: Asynchronous Interlocked Pipelined CMOS Circuits [44]: Local clock strobe circuit

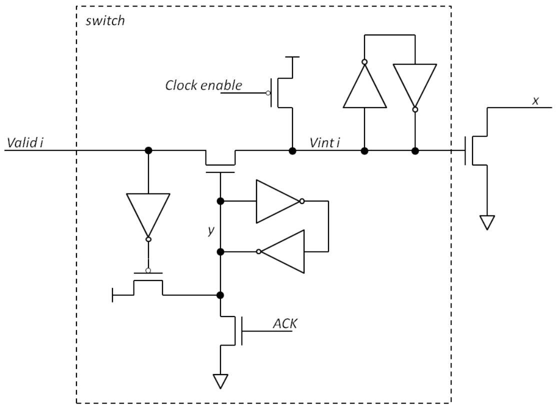


Figure 6.16: Asynchronous Interlocked Pipelined CMOS Circuits [44]: Strobe circuit switch

6.2.9 Asynchronous Wrapper for Heterogeneous Systems

An asynchronous wrapper is presented in [4] that provides an asynchronous interface to a locally synchronous module. This input and output interface is wrapped around a locally synchronous module within a GALS system. The wrapper uses an asynchronous finite state machine (AFSM) to do the handshaking, which further pauses or stretches the local clock of the synchronous module. This asynchronous handshake circuit (AFSM) is built up by using Extended-Burst-Mode specifications, for further details on these see [57]. In contrast to the original GALS implementation this approach requires only one clock period to complete a full 4-phase handshake due to its used AFSM. This enables a data exchange every clock cycle. Each locally synchronous module uses its own variable-length stretchable clock, independent from other modules in the system. Figure 6.17 shows a locally synchronous module surrounded by an asynchronous wrapper. As one can see the wrapper provides an input port, an ouput port and a stretchable clock circuit to the module (see Figure 6.18). Each port has a request, acknowledgement and data line. The request and acknowledgement lines are used to perform a 4-phase bundled data handshaking for each data item entering or leaving the module. The port module generates a stretch signal to signal that new data is provided to the module and the handshake is not completed now. The stretch signal also causes the stretchable clock circuit to pause the local clock and thus synchronizing the asynchronous data, until the handshake is done. In particular the next rising edge of the local clock is prevented as long as the stretch signal is asserted. [4] provides detailed specifications of active and passive input and ouput ports. As in every other bundled data approach that uses pausible clocking the value of used delay element that stretches the clock has to be consistently higher than the worst case delay of the combinational logic within the synchronous module. In [4] the clock buffer delay (realized through inverter chain) is about at least 10 inverter delays for 32 internal registers. This number grows approximately proportional to the natural logarithm of the number of loads. Due to the use of stretchable clocks the MTBF in terms of metastability grows to infinity [4]. A further advantage of this approach is that the module stops the clock after computation has finished and sleeps until new data arrives. The asynchronous wrapper however suffers from the same problems as every approach that uses pausible clocks, namely the fact that with each request the clock is paused. With several inputs from different modules and "bad" timing the computation is badly delayed.

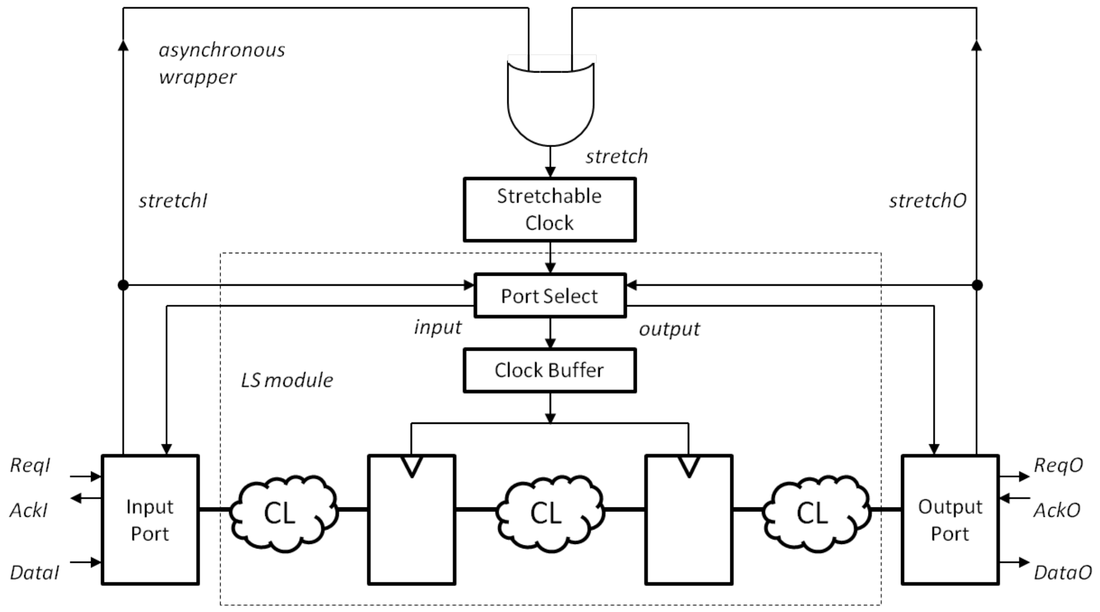


Figure 6.17: Asynchronous Wrapper for Heterogeneous Systems [4]: Locally Synchronous Module with Asynchronous Wrapper

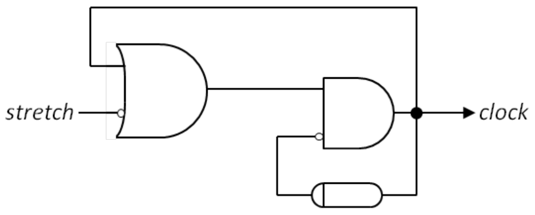


Figure 6.18: Asynchronous Wrapper for Heterogeneous Systems [4]: Stretchable Clock Circuit

6.2.10 Pausible Clocking: A First Step Toward Heterogeneous Systems

Another approach that uses pausible clocking mechanism for communication between differently clocked modules is presented in [58]. The modules communicate via an asynchronous FIFO. Further a module uses a 2-phase handshaking protocol to interact with its FIFO. Each module employs a pausible clocking control (PCC) unit as interface to a FIFO. A detailed circuit schematic of the PCC unit is given in [58]. The PCC unit adjusts the local clock to avoid synchronization failure by using a mutual exclusion element, which arbitrates between an incoming request from the FIFO and the local clock generated by a ring oscillator. Note that the used ring oscillator [58] limits the maximum frequency of the local clock to 22 MHz. We know from [40] that mutual exclusion elements become metastable in case that both inputs are asserted at nearly the same time. In this case the mutual exclusion element needs an arbitrary amount of time to decide on one input, once decided the PPC either immediately pauses the local clock or stalls the incoming request for another half cycle. So there is no malicious behaviour in case of metastability, except for the indefinite delay. Figure 6.19 shows a block diagram of a receiver PCC unit for one-way communication. One sees the request R_p from the FIFO is provided to the asynchronous FSM, that converts the 2-phase handshake request to a 4-phase signal. The AFSM provides the request to the mutual exclusion element. When the request input is selected by the mutual exclusion element it generates a grant signal which is looped back to the AFSM. The AFSM de-asserts the request output after having seen the rising edge at the grant signal. Furthermore the grant signal triggers the generation of the synchronized request (SR_p) that is forwarded to the FSM (synchronous), which generates the acknowledgement for the transmitter (A_p). The grant signal is converted from a 4-phase to a 2-phase protocol signal. The local clock that is generated by the ring oscillator is provided to the other input of the mutual exclusion element. It is paused or stretched if the mutual exclusion element has chosen the request input. In the bidirectional version of the PCC unit (see Figure 6.20) the FSM and AFSM are doubled, one as input and one as output interface to a FIFO. The input and output interface are switched by an additional arbiter between the AFSM and the mutual exclusion element. This approach allows the mixture of asynchronous and synchronous modules within a system and proposes to use the communication mechanism in a ring configuration to keep the system simple.

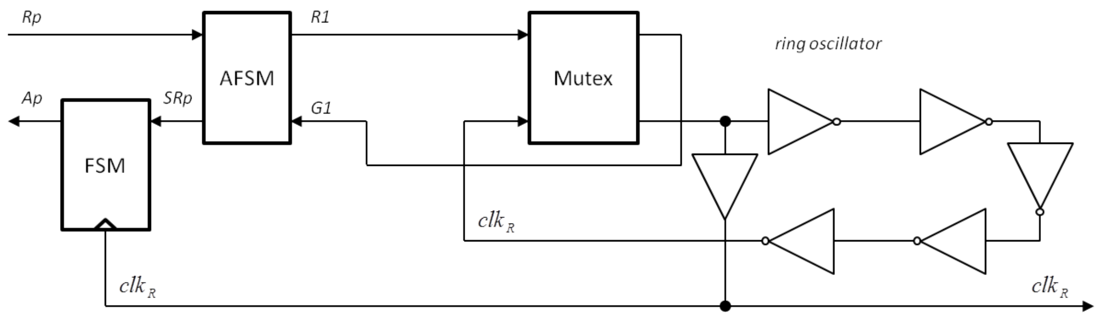


Figure 6.19: Pausible Clocking - A First Step Toward Heterogeneous Systems [58]: Receiver PCC for unidirectional communication

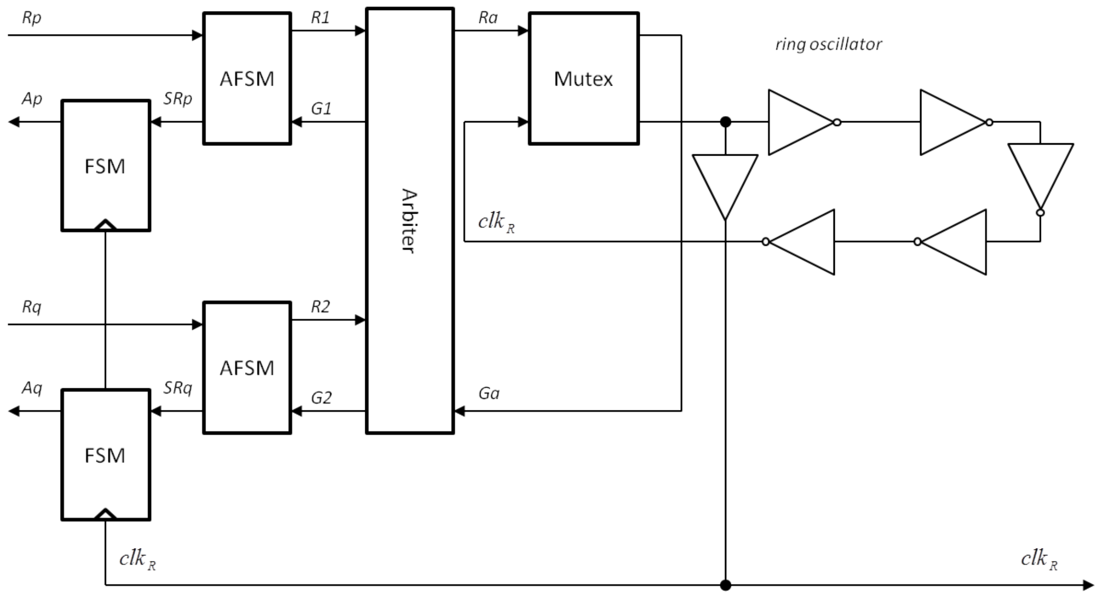


Figure 6.20: Pausible Clocking - A First Step Toward Heterogeneous Systems [58]: Receiver PCC for bidirectional communication

6.2.11 SCAFFI: An Intrachip FPGA Asynchronous Interface Based on Hard Macros

The *Stretchable Clock Asynchronous Flexible FPGA Interface*, SCAFFI [41], is an approach of an asynchronous interconnect between differently clocked modules that is based on pausible clocking. SCAFFI does not use an arbiter like other solutions and thus circumvents possible metastabilities caused by the employed mutual exclusion element and involved indefinite delays. On the other hand SCAFFI creates problems with metastabilities by using Muller C-Elements [40]. Further the interface can stretch the clock using both logical levels. In the basic architecture it supports bundled data communication using a 4-phase handshake between the transmitter and the receiver. Between the data production block of the transmitter and its output port as well as between the data consumption and the input port of the receiver a 2-phase handshake is used. The structure of SCAFFI is depicted in Figure 6.21. Further in Figure 6.22 one can see a detailed circuit of the stretcher module. The circuit employs two feedback loops which are switched by a multiplexer controlled by the input request. If a request is asserted the current value of the clock signal is kept (blocked by the Muller C-Element), else the clock is generated by the lower loop (and its inverter). The blocking Muller C-Element allows stretching of the clock at any logical level. An acknowledgement is generated after a delay that is longer than the delay of the multiplexer and the Muller C-Element combined. The delay *D3* controls the frequencies (duty cycle) of the pausible clock. To prevent possible glitches at the output of the multiplexer from propagating to the clock output the delay element *D2* is inserted.

In operation the data production of the sender asserts a synchronous request to signal the output port (of the transmitter) that it has new data available for the receiver. The output port asserts the request line of the stretcher and thus pauses the clock. Upon the acknowledgement from the stretcher the output port generates an asynchronous request for the input port of the receiver. This asynchronous request is forwarded to the stretcher of the receiver, thus the clock of the receiver is paused. A synchronous request is placed on the data consumption of the receiver, but since the clock is paused it cannot gather data immediately. Next an asynchronous acknowledgement is sent to the transmitter and further the clock of the receiver is released, hence the receiver samples the new data on deassertion of the acknowledgement from the stretcher. On reception of the acknowledgement from the receiver the request input of the stretcher is deasserted and thus its acknowledgement output. From this the synchronous acknowledgement for the data production is generated signalling that data transmission is completed. Note that the transmitter's clock is paused more than twice as long as the receiver's clock. The bundled data approach is only used if short-range connections are employed, for long wire interconnections SCAFFI is used as a point-to-point delay-insensitive interface implementing dual-rail data lines. Figure 6.23 shows the concept of the dual-rail SCAFFI. The dual-rail SCAFFI employs two addtional modules a single-to-dual-rail converter at the sender and a dual-to-single-rail converter and validity detection (XOR gate) at the receiver. Note that the Muller C-Element forms the asynchronous request. With these extensions SCAFFI can be used to create an interface between synchronous and dual-rail quasi-delay-insensitive modules. In summary, SCAFFI employs 4 times the throughput of a series of 2-phase handshake flip-flops with a transmitter clocked at $50MHz$ and a receiver at $78MHz$ a throughput of 31 MegaWords/s is achieved.

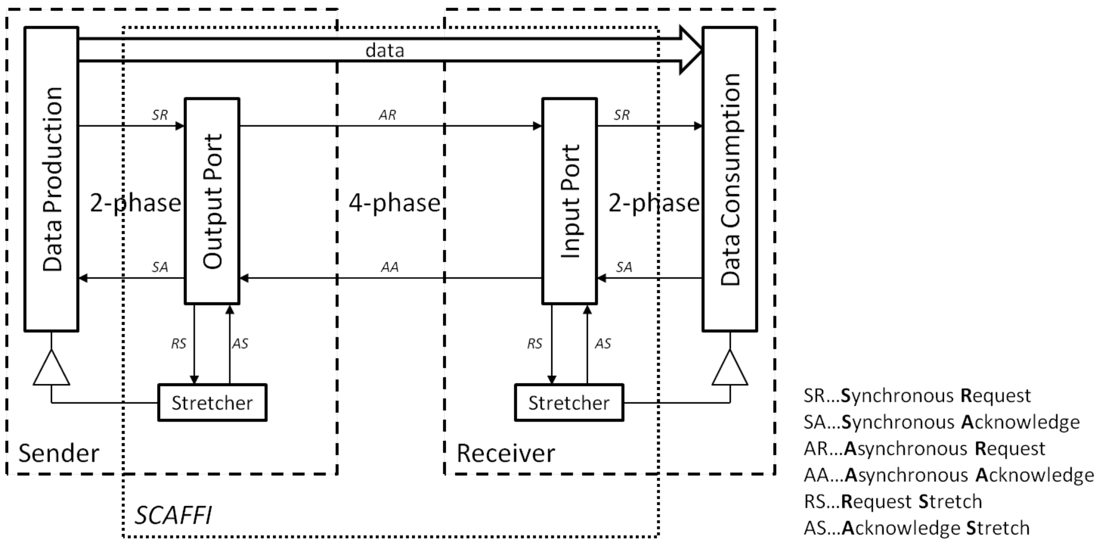


Figure 6.21: SCAFFI [41]: Structure of SCAFFI, including transmitter and receiver sides of the interface

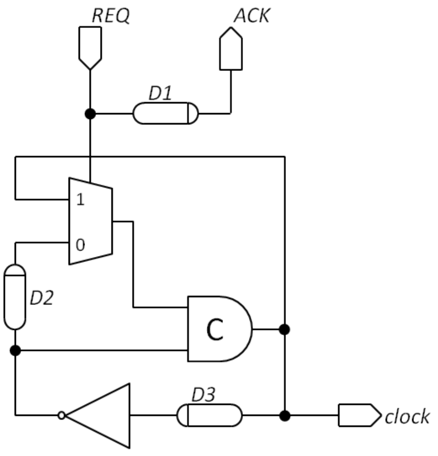


Figure 6.22: SCAFFI [41]: Structure of the Stretcher

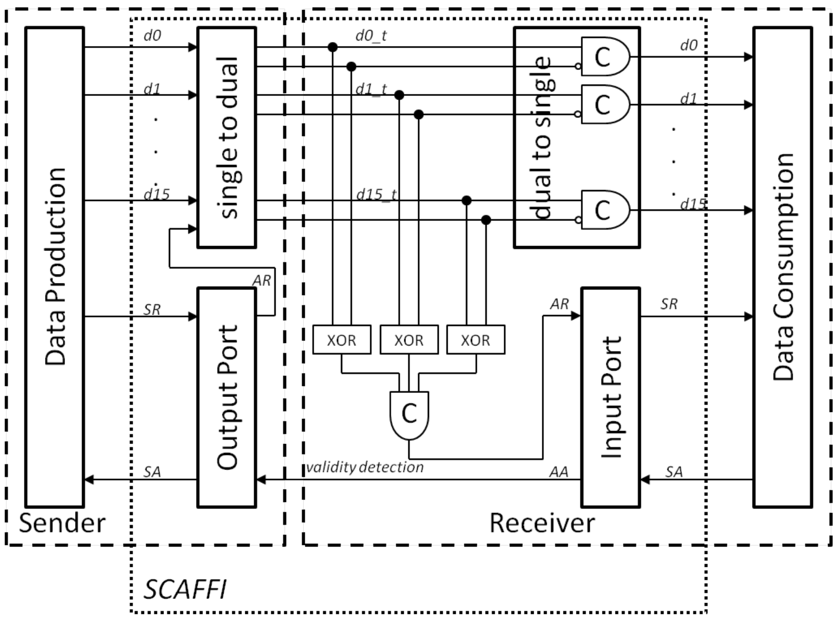


Figure 6.23: SCAFFI [41]: Dual Rail SCAFFI for long distance between transmitter and receiver (stretchers omitted for clarity)

6.2.12 Point to Point GALS Interconnect

The point to point GALS interconnect presented in [37] is basically a combination of a clock pausing mechanism, a self-calibrating delay line and an asynchronous interconnect using a 2-phase signalling technique. Since the interconnect is asynchronous it can be placed between independently clocked domains and enables communication between asynchronous and synchronous modules. [37] provides two circuits, a synchronous receiver and a synchronous transmitter, to receive and send asynchronous data, respectively. The synchronous receiver (see Figure 6.24) employs a looped (clock) oscillatory process, that is looped through an arbiter and paused if the clock input is currently low and a request is incoming (connected to other port of the arbiter). When the arbiter grants access for the request it activates the first latch to sample new data, and further signals that request is received, thus the request is dropped (XOR gate is deactivated) which unpauses the clock (arbiter grants access to the clock input). If the clock falls (arbiter input goes high) simultaneously to the rising request signal the arbiter may become metastable. This is safe since the arbiter does not propagate request or generate clock until metastability has resolved, but the decision making may take an arbitrary amount of time. When the clock is running again and data is latched in the system the request signal is synchronized and used as acknowledgement for the transmitter. The second provided circuit forms the synchronous transmitter (see Figure 6.25), it uses a pausible clocking mechanism as the receiver. For data transmission it requires a control signal from the (asynchronous) receiver that shows whether the receiver is ready to receive new data. This is done by an acknowledgement signal, that is generated as response to the request signal from the transmitter, showing that new data is ready. The acknowledgement signal pauses the clock at the transmitter and further de-asserts the request signal and initiates the transmission of new data. After enabling the latches the acknowledgement is looped back (*ACK received*), which releases the arbiter and further unpauses the clock. For both modules the clock generating delay is matched to the worst case path of the combinational logic within the system (GALS module). Using only these modules limits the bandwidth to only one data item per clock cycle at maximum. The average bandwidth is even worse, the synchronous transmitter has to wait at least one cycle between sending data and receiving the corresponding synchronized acknowledgement. To circumvent this bottleneck the channel communication is enhanced with buffering, in particular an asynchronous FIFO, which decouples the path of the request signal to the synchronized acknowledgement signal and allows the transmitter to send a data item every clock cycle, see Figure 6.26. The buffering FIFO shows that a data element is empty by a *consumed* signal which is used as acknowledgement for the transmitter. If the FIFO is full, no consumed signal will be sent to the transmitter, thus no acknowledgement arrives and the transmitter clock is paused. The synchronizer module can be extended to interface more than one input or output, therefore for each interface an additional arbiter is required, all the parallel used arbiters of the interface are merged by an AND gate. A simulation in [37] shows that a FIFO with more than two slots does not improve performance further. To provide the modules with a sleep mode extension an additional arbiter is required that permits the clock to be paused.

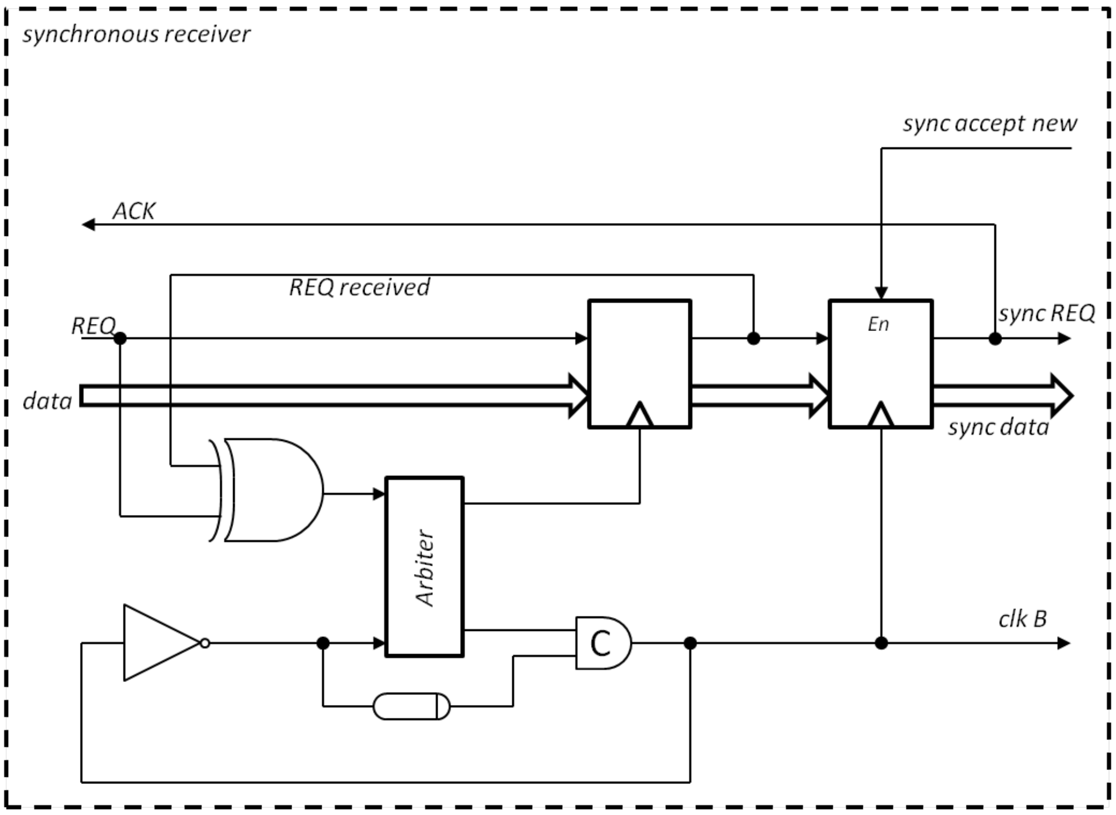


Figure 6.24: Point to Point GALS [37]: Interface between an asynchronous transmitter and a synchronous receiver

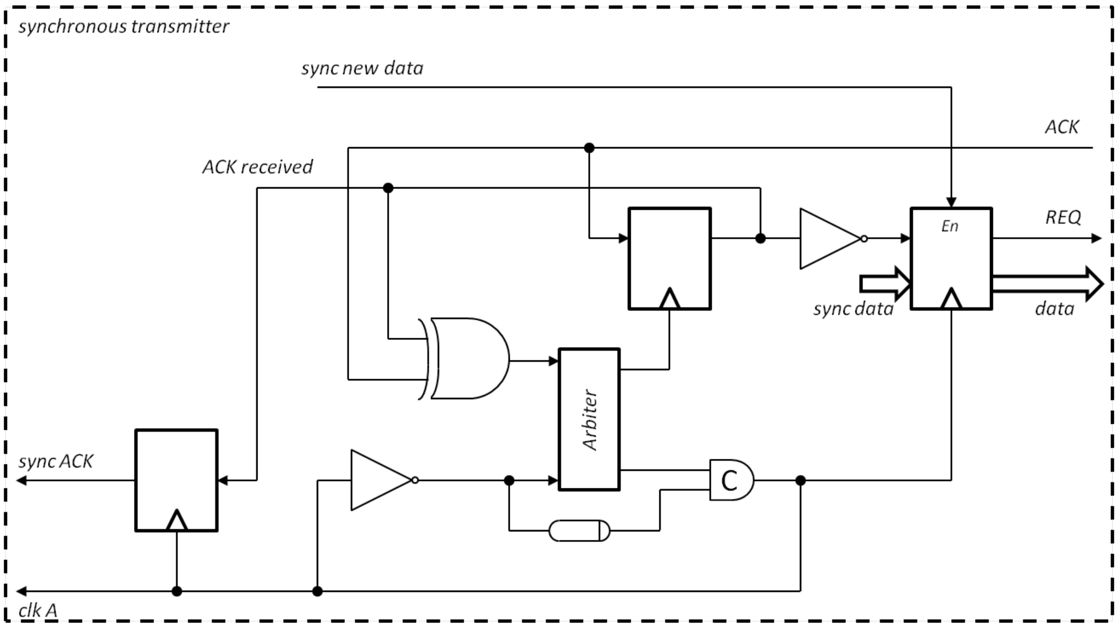


Figure 6.25: Point to Point GALS [37]: Interface between a synchronous transmitter and an asynchronous receiver

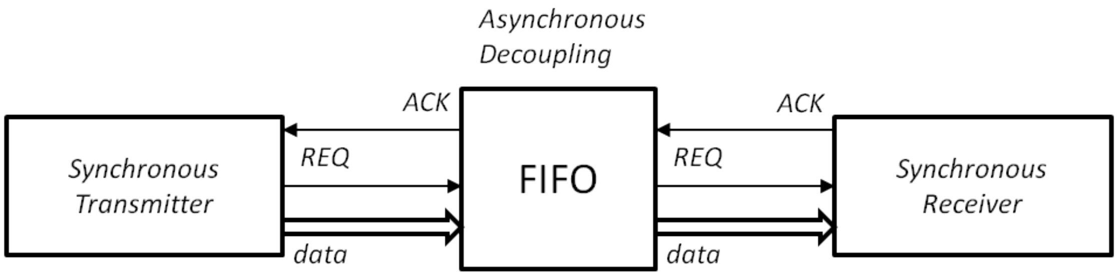


Figure 6.26: Point to Point GALS [37]: Channel communication with FIFO buffering

6.2.13 Interfacing Synchronous and Asynchronous Modules within a Highspeed Pipeline

A method to interface synchronous as well as asynchronous modules with a highspeed synchronous pipeline is shown in [50]. In opposite to a typical GALS system it combines a globally synchronous pipeline with locally asynchronous modules. This is done by using a synchronously stoppable clock signal generated by a ring oscillator. Additionally the clock signal is used as handshake signal for controlling the asynchronous modules. In contrast to typical GALS interfacing solutions the proposed pipeline interface does not need a mutual exclusion element for clock generation due to the assumption that the pipeline determines when to move data to an asynchronous module and thus the transmission is synchronized to the internal clock. Hence data is moved every clock cycle into and from an asynchronous module. The pipeline employs an interface controller that generates the stoppable clock as well as the handshake signal for the asynchronous modules. Therefore the interface controller employs a stoppable clock generator and a handshake controller module. Figure 6.27 shows the asynchronous-synchronous interface controller. The interface controller generates a rising clock edge and then de-asserts the *run* signal. When all asynchronous modules finished the handshake procedure, the *run* signal is asserted again. Note that the *run* signal needs to be explicitly synchronized to the interface controller clock, otherwise bad timing can cause metastability in the synchronous interface controller. The handshake controller circuit is shown in Figure 6.29. The interface controller employs one handshake controller for each asynchronous module to generate the request signal for them. Further it generates the *run* signal for the stoppable clock generator. The request signal is generated on reset or falling edge of the acknowledgement from asynchronous module. During low phase of the request signal the asynchronous logic is precharged (domino dual-rail logic) and thus results from previous computations are removed. The request signal is de-asserted when clock and acknowledgement signals from the asynchronous module are high. A state holding inverter loop is used to keep the request signal during low phases of the running clock. So the handshake controller is used to generate a 4-phase handshake from the stoppable clock signal, the request signal is asserted to signal that new data is available and the acknowledgement signal to signal that the computation is finished and data lies ready at the output. The acknowledgement signals of all employed handshake controllers are merged into an AND gate and forwarded as *run* signal to the stoppable clock generator. The circuit of the stoppable clock generator is depicted in Figure 6.28. It employs a state-holding gate to save the state of the clock in case of the run signal is de-asserted from one of the handshake controllers. An inverter chain is employed whose length matches the time of the worst case computation path of the slowest synchronous module to generate a clock signal that is suitable for the system. After the acknowledgement falls, thus the precharge is completed data is forwarded on the rising edge of the clock signal. To compensate the large control overhead the delay between the de-assertion of the acknowledgement signal to that of the *run* signal is covered by the computation. Further the time between the assertion of acknowledgement and that of the *run* signal can be masked by adding another pipeline register to cover the delay with the precharge delay. The additional register is needed to preserve the computation results, recall that during the precharge phase all results get lost. A lot of measurements was done on the technique, for results see [50].

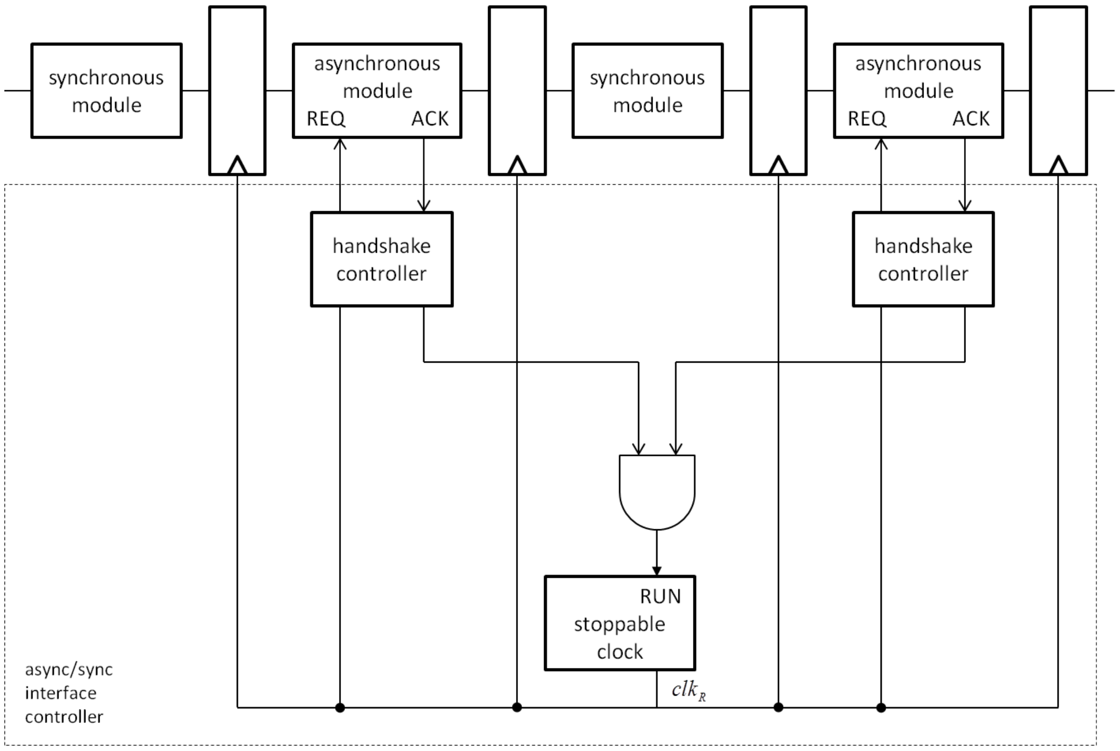


Figure 6.27: Interfacing Synchronous and Asynchronous Modules within a Highspeed Pipeline [50]: Interface Structure

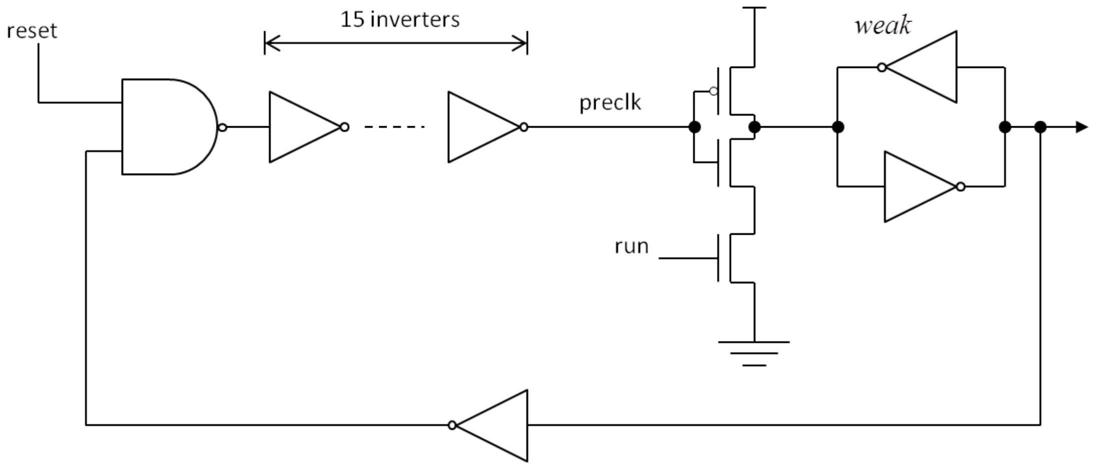


Figure 6.28: Interfacing Synchronous and Asynchronous Modules within a Highspeed Pipeline [50]: Basic Stoppable Ring Oscillator Clock Circuit

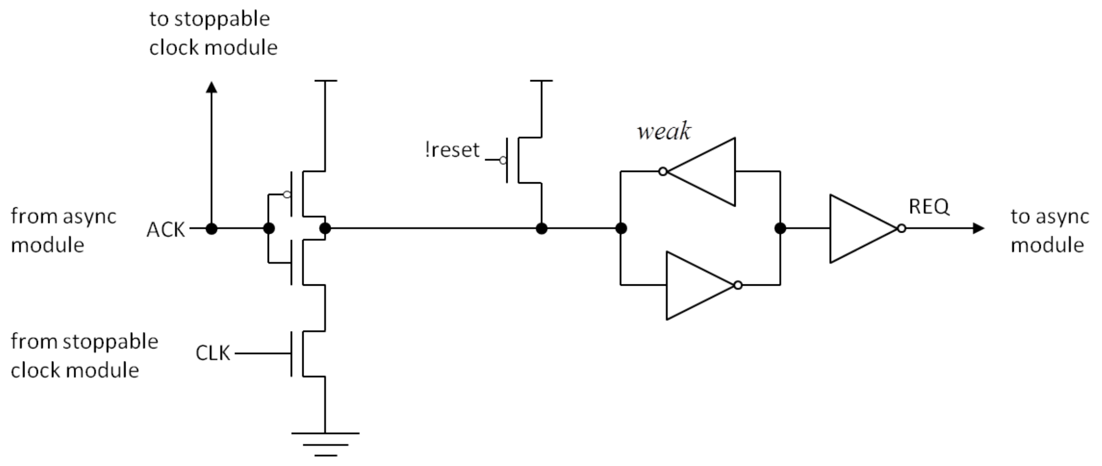


Figure 6.29: Interfacing Synchronous and Asynchronous Modules within a Highspeed Pipeline [50]: Handshake Control Circuit

6.2.14 Using Stoppable Clocks to Safely Interface Asynchronous and Synchronous Subsystems

A *Simple Clock Stretching Circuit* that is used to safely interface asynchronous and synchronous systems is presented in [38] and can be seen in Figure 6.30. It is used in combination with a bundled data approach and is employed at the receiver side. Instead of resolving metastability it is prevented by stretching the clock, this is achieved by a digital delay line. A 4-phase handshake is used on the one hand to signal that new data is available and on the other hand that data is latched. The stretching circuit employs two data latches and a digital delay line that is looped through an arbiter to generate the pausible clock. Besides the pausible clock line a request input line is an input to the arbiter. When the request input rises and clock is low data will be latched by the first register, thus the request enables the first latch directly (data must be stable when request is asserted). As long as the request is asserted the clock is paused. Once the arbiter grants access to the pausible clock line data is moved forward, which will generate the acknowledgement for the transmitter. Once a request is de-asserted the clock starts again and the second register latches the data and forwards it to the combinational logic of the receiver. The performance of the receiver is reduced due to this arbitration process. Thus the minimum latency on data latching is one half clock cycle, if a data request arrives slightly after the clock goes low. On the other hand the maximum latency of the interface is one full clock cycle, if a request arrives slightly after the clock rises. In [38] there are some improvements to the basic stretching circuit, first of all the *clock prediction* circuit that predicts when the next rising clock edge occurs and starts arbitration early to give the synchronous system (receiver) a higher priority. It uses the inverted clock to lock out the asynchronous system (transmitter). There are two possible ways to use the interface for more than one input, first by serializing the data from different transmitters, thus the transmitters have to take turns and second adding arbiters for each additional input. By serializing the data input the latency of the interface is increased. As previously mentioned it is possible to receive data from more than one transmitter by adding arbiters per input. This is realized by the *parallel clock pause circuit* (see Figure 6.31) that consists of a number of arbiters, one for each input. Each arbiter switches between request input and the inverted clock, outputting either an acknowledgement for the transmitter or producing a *clock-allowed* signal when all requests are low, respectively. The inverted clock locks the arbitration process at each input and is merged by an AND gate to result in a *clock-allowed* signal. When clock is high, the inverted clock releases the arbiter and data can be forwarded to the system. The timing of this window can be adjusted by the variable delay to converge as near as possible to the rising edge of the receiver clock. The *clock-allowed* signal is the merged product (by AND gates) of all inverted clock signals, and prevents operation in presence of metastability. Further each of these input interfaces requires a *decoupler circuit* that guarantuees that the input locks on its arbiter for the minimum time that is required by the receiver to latch data and thus ensure correct operation. The decoupler circuit, see Figure 6.32, operates as follows, when a *new-data* indication signal arrives, a request is generated and sent to the clock pause circuit (see Figure 6.31) which respond with an acknowledgement when it is safe to forward new data. The acknowledgement sets the RS-flip-flop which further generates an *enable* signal that controls the data latches. When data is consumed, i.e. clock is rising and acknowledgement is still high the *enable* signal is de-asserted and a *consumed* signal is generated. The *consumed* signal is forwarded to the transmitter and allows it to de-assert the *new-data* signal. A simulation in [38] shows that there is sufficient time between clock going low and new data signal rising such that the arbiters does not go metastable. Note that the enable signal of the data latches forms the most critical delay.

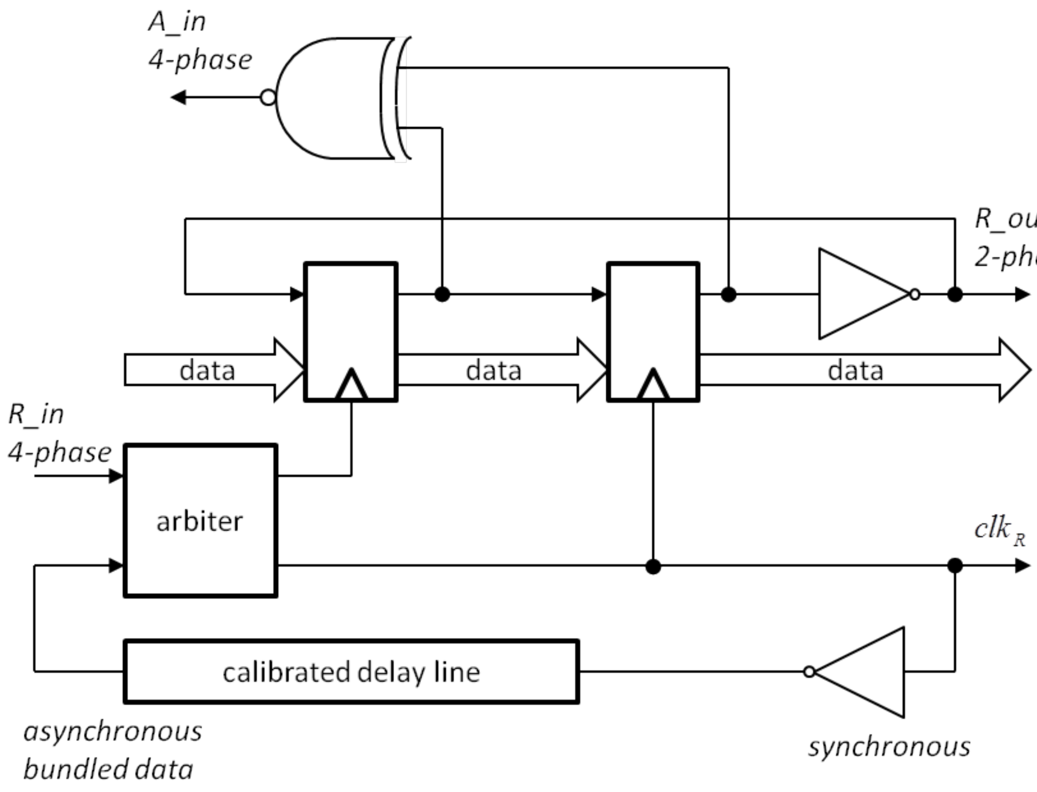


Figure 6.30: Using Stoppable Clocks to Safely Interface Asynchronous and Synchronous Subsystems [38]: Simple Clock Stretching Circuit

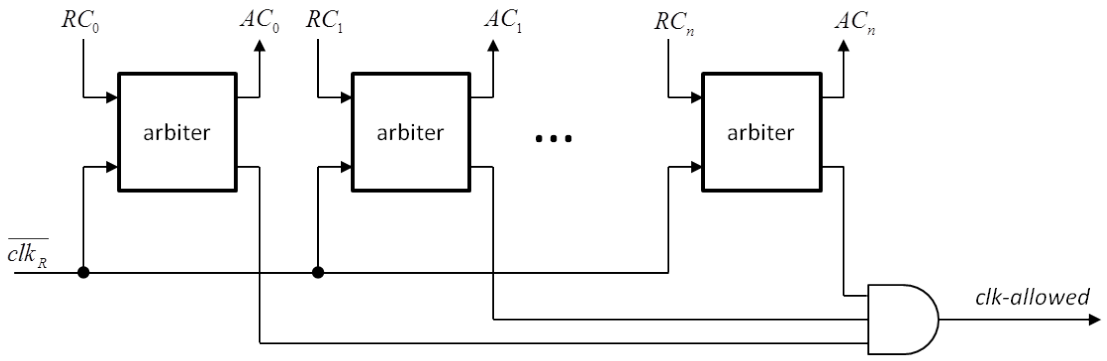


Figure 6.31: Using Stoppable Clocks to Safely Interface Asynchronous and Synchronous Subsystems [38]: Parallel Clock Pause Circuit

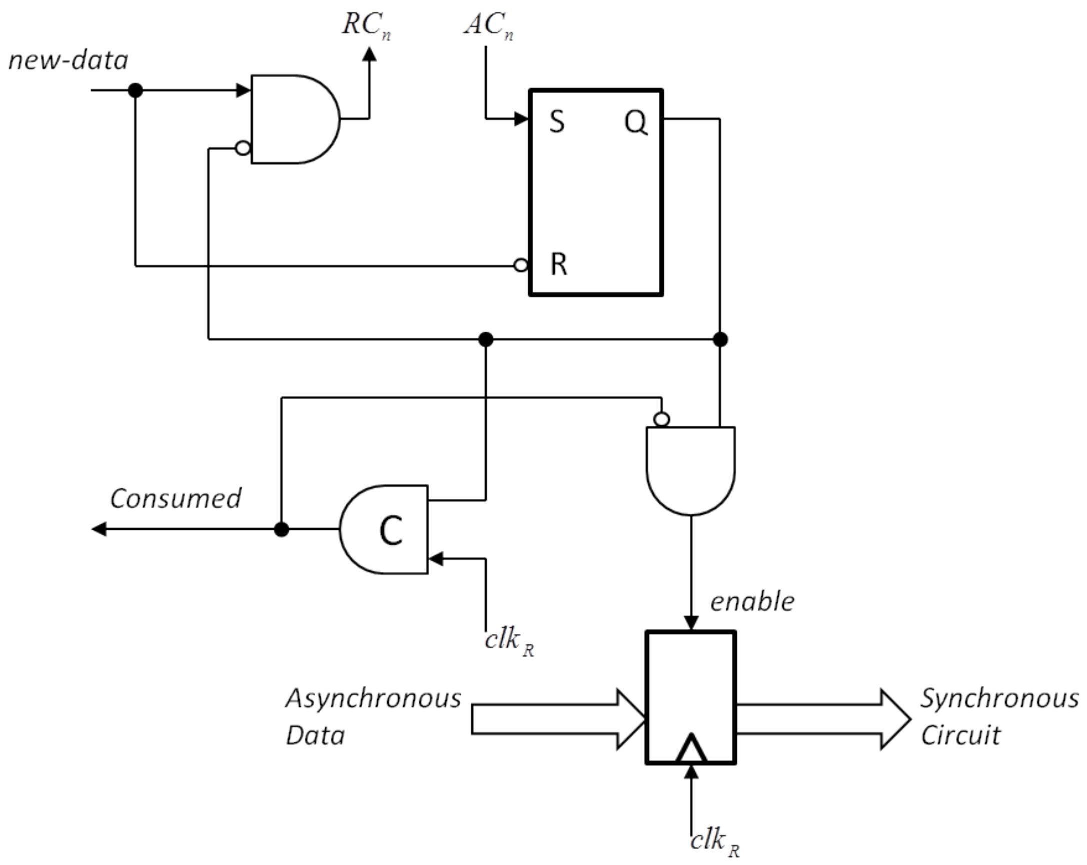


Figure 6.32: Using Stoppable Clocks to Safely Interface Asynchronous and Synchronous Subsystems [38]: Decoupler Circuit

6.2.15 A Synchronizer Design Based on Wagging

The wagging synchronizer is proposed in [1]. This synchronizer uses an extended version of dual-edge triggered D-flip flops (DETFF) whose operation is based on wagging. The DETFF consists of two latches each with an input gate and an output gate. In the wagging synchronizer the DETFF is extended by a third latch. Furthermore the input and output stage of the latches are clocked by two different clock signals based on the same clock signal but different in phase (same duty-cycle). For the entire wagging synchronizer the clock signal is used with three different phases (see signals clk_0, clk_1 and clk_2 in Figure 6.33). The circuit of the wagging synchronizer can be seen in Figure 6.34. The upper latch (*latch*$_0$) samples the input on an edge of *clk*$_0$ and the lower latch (*latch*$_2$) outputs the value (on the same edge). In this time segment the latch in the middle (*latch*$_1$) holds the current value to let a possible metastable state decay. Next, on an edge of *clk*$_1$ the latch in the middle outputs the value and the lower latch samples a new one, while the upper latch holds its value for metastability resolution. An edge of *clk*$_2$ triggers the upper latch to output its value and the latch in middle to sample a new one. The lower latch holds its value to recover from metastability. From this point onward the procedure repeats again. The wagging synchronizer is compared to two 2-flip flop synchronizers in [1] in detail, it exceeds them in MTBF and latency and area. The MTBF is calculated for a system using a $2.5\ GHz$ system clock and is about 2.66 years, with a latency of $511ps$ for a resolution time of 40τ, where τ is the resolution time and $\tau = 10.66ps$. Further is employs a throughput as the Two-Flip-Flop Synchronizer of one word per cycle.

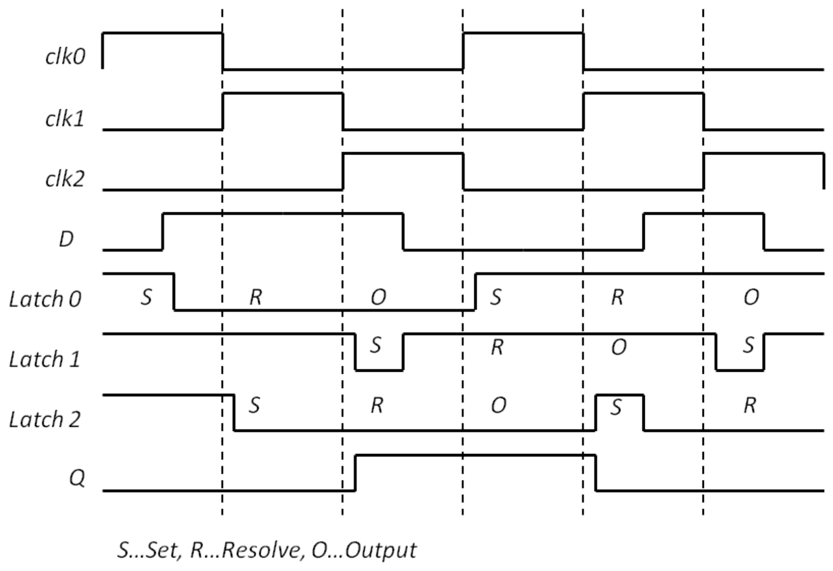


Figure 6.33: A Synchronizer Based on Wagging [1]: Signals of Wagging Synchronizer Operation

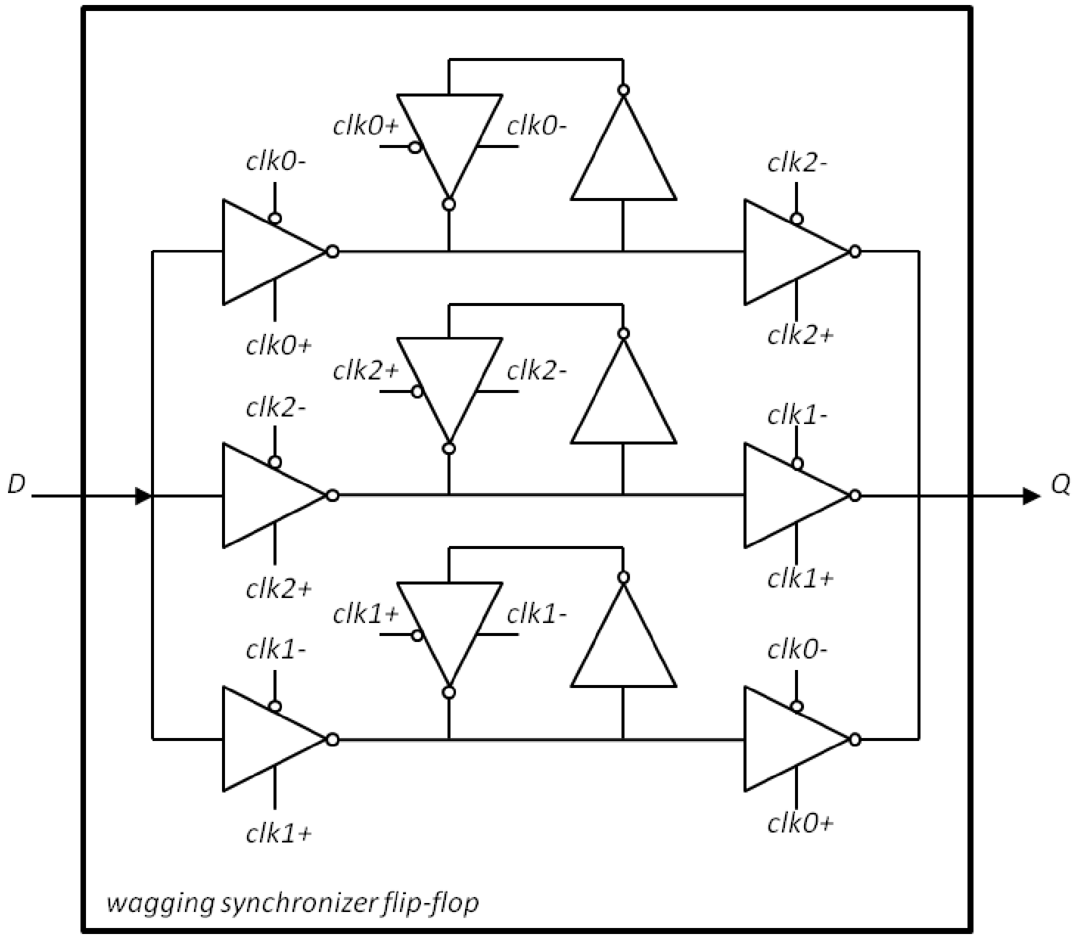


Figure 6.34: A Synchronizer Based on Wagging [1]: Wagging Synchronizer Flip-Flop

6.2.16 A Fast Resolving BiNMOS Synchronizer for Parallel Processor Interconnect

The BiNMOS metastability resolving synchronizer is presented in [31]. This synchronizer enables communication between two synchronous systems with unknown phase and frequency relationship, thus the interconnected systems are truly asynchronous in respect to each other. The BiNMOS synchronizer is built up as parallel staged synchronizer. Metastable immune circuitry is omitted in its structure, because this usually does not accelerate the output of a stable value nor enhances the settling time of the synchronizer. The parallel staged BiNMOS synchronizer is shown in Figure 6.35. It employs three Jamb latches and a multiplexer. The Jamb latch is a cross-coupled inverter latch and its schematic can be seen in Figure 6.36. Each Jamb latch is alternately enabled, thus only one is enabled at a time to latch incoming data. A ring counter generates a special enable signal for each synchronizer stage (Jamb latch), which on the one hand enables a latch to sample data and on the other controls the multiplexer and connects the output to the following latch and forwards the previously latched data. The benefit of parallelization of the synchronizer stages is that there is only one latch in the data path at a time, thus high throughput, and the settling time is enhanced without reducing the sampling rate. The Figure shows three synchronizer stages but it can be extended and with each additional stage the MTBF increases. The benefit of using Jamb latches is the reduced RC loading in the synchronizer and the resulting optimization of the gain-bandwidth product [31]. This behaviour results from the structure and operation principle of the Jamb latch. Note that the Jamb latch switches by overpowering an NMOS that uses a bipolar emitter follower, from there originates the name of the synchronizer 'BiNMOS'. In [31] an extensive testing is done and a table with MTBF results for different parameters (settling time and clock frequency) is given. The BiNMOS synchronizer provides a settling time of $8.5\ ns$ at a clock frequency of 200 MHz and exponential time constant of the decay rate of metastability τ of around 70 ps (simulated results, a table is given in [31]).

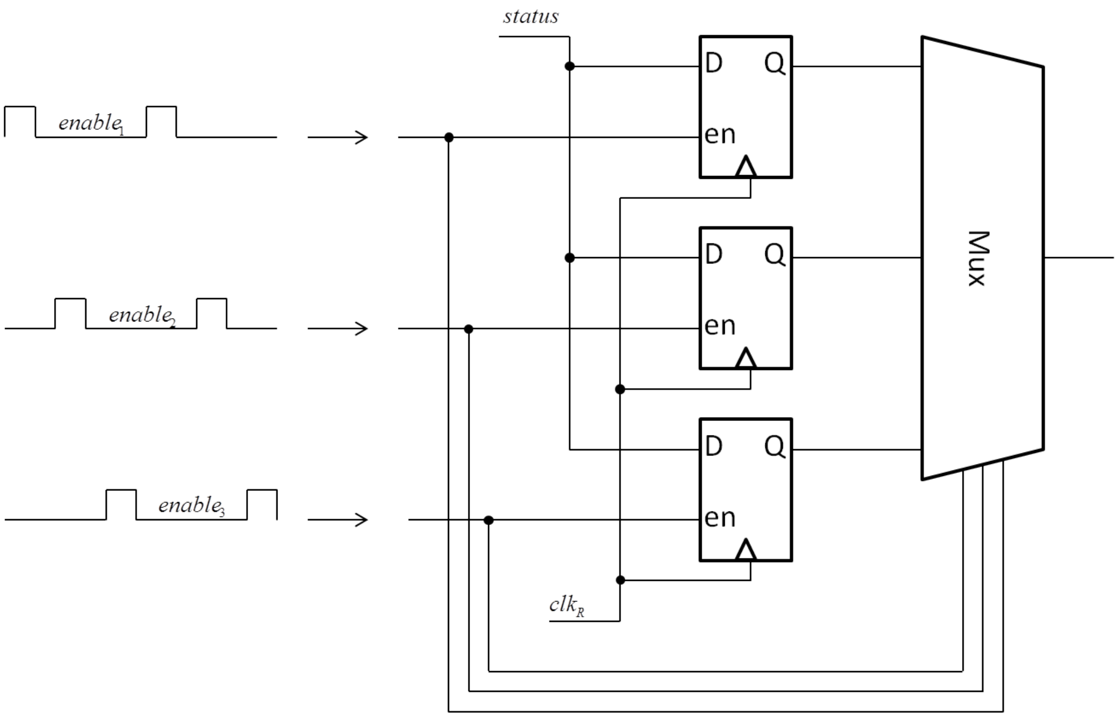


Figure 6.35: BiNMOS Synchronizer [31]: Parallel Synchronizer Stages

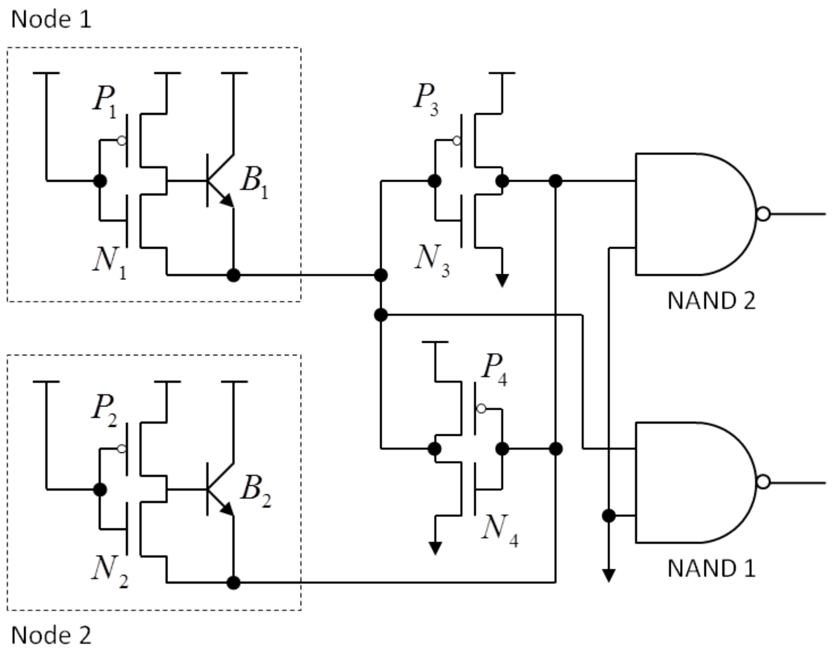


Figure 6.36: BiNMOS Synchronizer [31]: Basic Jamb Latch

6.2.17 Asynchronous FIFO Synchronizer

The *Asynchronous FIFO Synchronizer* [12] [28] can be seen in Figure 6.37. A FIFO is used to synchronize the data stream by explicitly synchronizing the transmitter and receiver pointer to generate a signalization for a full or empty FIFO. The data is shifted into the FIFO according to the transmitter clock and consumed with receiver clock, employing a latency of $3-6$ cycles [17] and a throughput of one data word per cycle [17]. A flow control is implicitly implemented via the *full* and *empty* signalization of the FIFO. As synchronizer for the pointers, a two-flip-flop or conservative synchronizer as outlined in the previous section can be used.

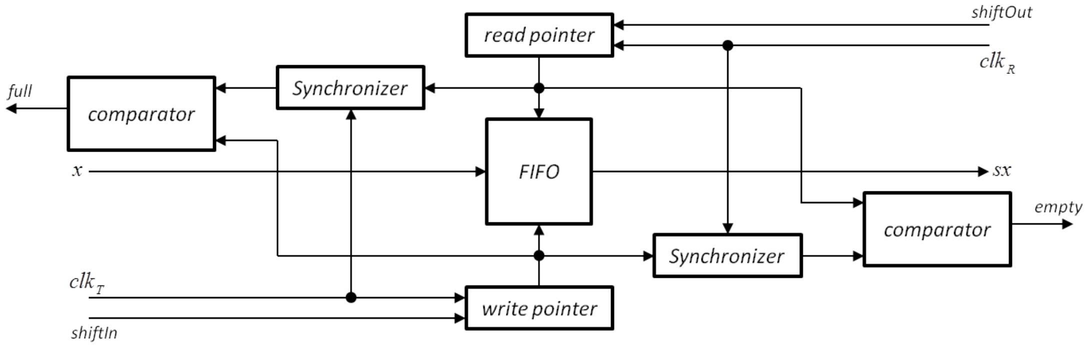


Figure 6.37: Asynchronous FIFO Synchronizer [12]

6.2.18 Robust Interfaces for Mixed-Timing Systems

Another approach using FIFOs as interfacing solution is presented in [10]. Four 'mixed-timing' FIFOs are presented, first one interfaces between synchronous (clocked) modules with arbitrary clock frequencies, second one between asynchronous (self-timed) modules and the last two FIFOs either synchronize the communication from a synchronous to an asynchronous module or vice versa. Each FIFO consists of a variable number of cells that are circular arranged, each cell further consists of a get interface (receiver side), a put interface (transmitter side), a register (part of get and put interface) for data latching and a data validity controller. Due to this modular structure of the mixed-timing FIFO approach, the different variations are easily accomplished by freely combining the put and get interfaces. Note that the validity controller needs a special implementation for each of the interface types. In the synchronous version of the FIFO the put/get interfaces employ a full/empty detector, respectively, that monitors the current state of the FIFO (cells). Additionally a put/get controller regulates the access to the FIFO according to the state of the FIFO (full/empty). These two function blocks are not required within the asynchronous version. The asynchronous interfaces use a 4-phase communication protocol in combination with a single-rail bundled data approach. In both cases data transmission is initiated by a request either from the transmitter (put request) or the receiver (get request). In the synchronous version the request is blocked if the FIFO is full/empty by the put/get controller. In the asynchronous case the acknowledgement for the handshake is simply held back if the FIFO is full or empty.

Once a data item is placed in a cell of the FIFO it does not change the spot until it is removed from the FIFO.
In Figure 6.38 one sees the implementation of the mixed-clock FIFO, that interfaces two synchronous modules. The cell that is next to be written or read is identified by a put or get token, respectively. The tokens are passed to the left cell after an operation. In the mixed-clock FIFO the data validity controller consists of a SR-Latch that is set when a data item is written to the cell (a put token is received and the interface is enabled) and reset when a data item is read (a get token is forwarded and the interface enabled). The control signals (full/empty) of the FIFO have to be synchronized to the respectively domain to assure that a changing cell state is not falsely read. Simple two flip-flop synchronizers are used in combination with a modified definition of the cell states (full/empty). So 'full' is asserted when there are either zero or one empty cells left. To avoid deadlocks in the FIFO and assure robustness the definition of the 'empty' signal is split into two signals the so-called "new empty" that is asserted when there either non or one cell is filled and the 'true empty' that shows that no full cells are left. Both employed signals are synchronized into the receiver domain and merged with an AND gate into a global empty signal. Still it is possible that the control signals become metastable and are read incorrect. The metastabilities can only occur when changing from "full"/"empty" to "not full"/"not empty", thus in case of a wrong control signal an interface is stalled for one clock cycle in worst case. The latency of the mixed-clock FIFO is minimum $L_{min} = 0.5\ T_{put}\ +\ 2.5\ T_{get}$ and maximum $L_{max} = 0.5\ T_{put}\ +\ 3\ T_{get}$, where T_{put} is the period time of the transmitter clock and T_{get} the period time of the receiver clock. This depends on the synchronizers used to synchronizes the control signals. The mixed-clock FIFO can be extended by a frequency comparator to easily shortcut the synchronizer if both clocks (receiver and transmitter) are equal. This is called a dual-mode FIFO and reduces the latency to one clock cycle. The FIFO that interfaces two asynchronous (self-timed) systems simply consists of two asymmetric Muller C-Element for token control and a register for latching data items.
Figure 6.39 shows the FIFO (asynchronous to asynchronous). The cell on the right passes the token by asserting a write enable signal. The put token is merged with the put request and valid signal into the Muller C-Element and form the write enable signal for the next cell on the left. For an asymmetric Muller C-Element all inputs must be asserted to assert the output but not every input has to be de-asserted to de-assert the output (only designated inputs are required). For the Muller C-Element of the put interface the valid signal input may not be low to de-assert the output. In the get interface the read enable is generated as output of the asymmetric Muller C-Element upon asserted request signal, valid signal and read acknowledgement signal from the register. With the assertion of the read enable signal the get token is passed. The data validity controller generates the valid signal from the write and read enable signals.
For the mixed asynchronous-synchronous FIFO: the put and get interfaces from above can be reused, only a new data validity controller that indicates when a cell contains a data item is required. The STG specifications of the data validity controller for asynchronous-synchronous and synchronous-asynchronous FIFOs can be found in [10]. In summary, the throughput is decreased to two data items in three cycles due to the constraints on the empty and full cell, the basic interface moves a data item every cycle, further a table with simulation results is given in [10]. The latency can be calculated by the given formulas and further a table with simulation

results is given in [10]. To get a sufficient value for the MTBF (at the FIFO control signals) for fast clock speeds more latches are necessary.

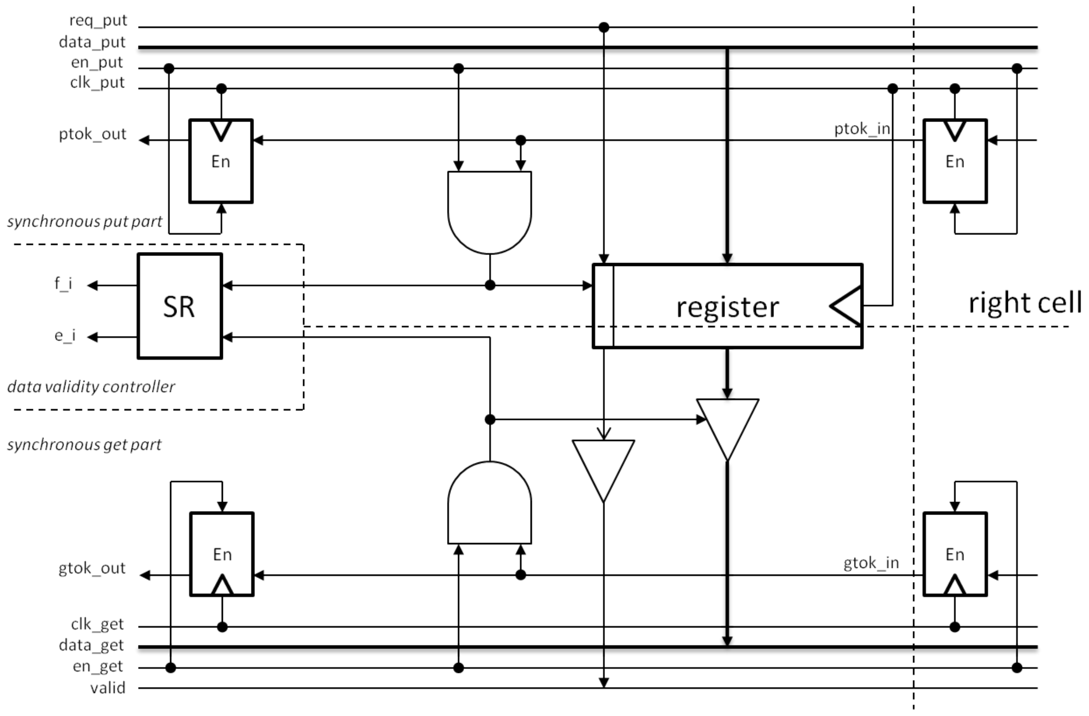


Figure 6.38: Robust Interfaces for Mixed-Timing Systems [10]: Mixed-Clocked FIFO cell implementation

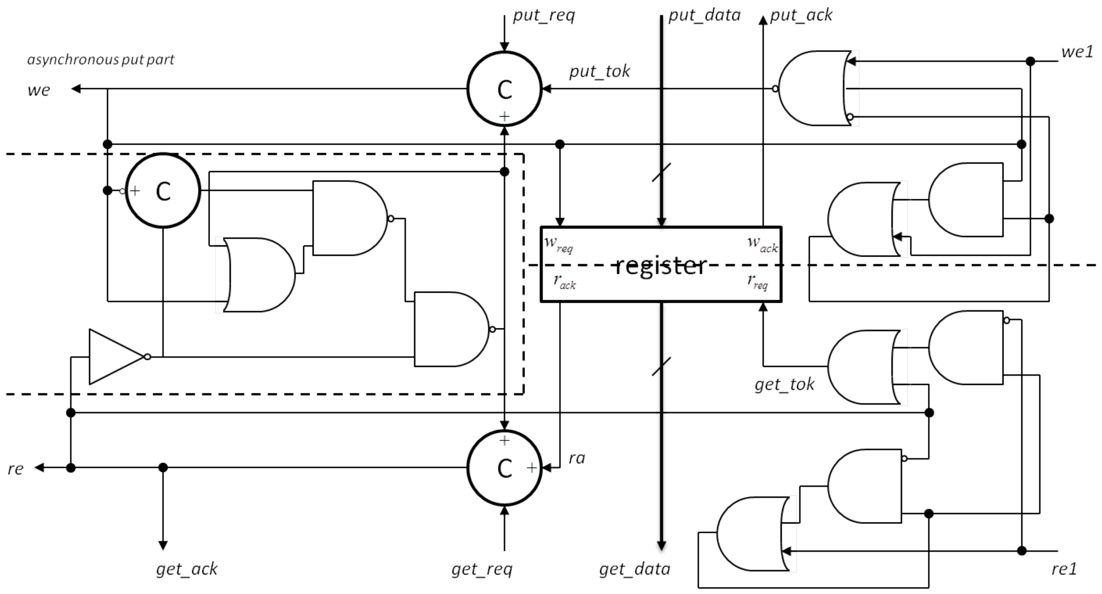


Figure 6.39: Robust Interfaces for Mixed-Timing Systems [10]: Asynchronous-asynchronous FIFO cell implementation

6.2.19 Pipeline Synchronization

The pipeline synchronization mechanism [47] is a solution to interface asynchronous signal into synchronous systems or two synchronous modules running two mutually different clock frequencies. It can be implemented as 2-phase or 4-phase handshake protocol both using bundled data approach. In Figure 6.40 one can see the structure of the pipeline synchronizer. It employs *k* stages, each consisting of a 2- or 4-phase protocol FIFO and an asymmetric or symmetric synchronizer element, respectively. A 2-phase FIFO behaves as follows: it waits for an incoming request, then concurrently generates an outgoing request for the successor stage and an acknowledgement for the stage that sent the original request and then waits for the acknowledgement from the successor stage to arrive. The request input of each stage is synchronized by a symmetrical mutual exclusion element to the clock input ($\varphi_{(k-1)} mod$ 2). Note that the clock inputs φ_0 and φ_1 ($\varphi_{(k-1)} mod$ 2) are 2-phase non-overlapping clocks [47]. In [47] a CMOS circuit is provided for the symmetric synchronizer. In a pipeline synchronizer the data stream synchronization is done along with the data flow to decrease the probability of synchronization failure exponentially with the number of pipeline stages. A correctness proof is given for this in [47]. For further details on the 4-phase version of the pipeline synchronizer see [47]. The design was implemented and tested with a measurement result of a latency of four clock cycles when using an 8-stages pipeline synchronizer resulting in a probability of metastability failure of less than 10^{-53}. The throughput of the design is throttled by the synchronous side to one word per cycle.

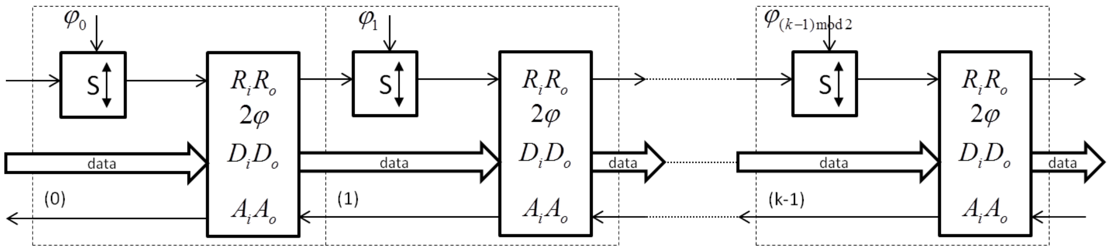


Figure 6.40: Pipeline Synchronization [47]: Pipeline Synchronizer with asynchronous input and synchronous output

6.2.20 The Even/Odd Synchronizer

The Even/Odd synchronizer [13] represents a synchronizer for periodic or heterochronous timing domains. In such a timing domain the transmitter and receiver clock are totally unrelated, there are no constraints on the used clock frequency other than it must be constant. The basic idea of the Even/Odd synchronizer is that the transmitter writes alternately to two registers. During even clock cycles it writes to the 'E' register and during odd cycles to the 'O' register (see Figure 6.41), the alternate write access is realized by an *even*-enable signal. At the receiver domain a selection signal that controls the multiplexer which selects between the 'E' or 'O' register is generated based upon the phase estimation of the transmitter clock (signal p). The phase estimation is calculated by interval arithmetic and comes with an estimation error of $\varepsilon = |\varphi - p|$ where φ is the actual phase of the transmitter clock and p the estimation of it. The phase φ of the transmitter clock is given as a real number within the interval $[0, 2[$, where 0 and 1 depicts a transition of the transmitter clock. The time around 0 and 1 is handled as a keep-out window of wide $\pm x$ (e.g. $60\ ps$ like the duration of the sample and hold window of a typical flip-flop), where the registers are not safe to sample. The transmitter writes the 'O' register within $]x, 1 + x]$, within this phase it is only safe for the receiver to sample the 'E' register. On the other hand the transmitter writes the 'E' register within $]1 + x, x]$, where it is only safe for the receiver to sample the 'O' register. So the rule is that the register that is most recently written (in the last half cycle) is safe to sample by the receiver. This mechanism does not ensure that each value written to the registers of the transmitter is exactly sampled once. It is possible to drop data if the transmitter clock is faster than the receiver clock and also to sample it more than once if the receiver clock is the faster one. This further requires a flow control mechanism, as in the proposed FIFO synchronizer where the write and read pointers (head/tail) are synchronized via the Even/Odd synchronizer. For the receiver it is necessary to estimate the phase of the transmitters clock. To do this it is required to measure its frequency in a first step. This is done by the fraction of two counter values, first counter (f_R) in the receiver timing domain increments until a fixed value (i.e. 2^b) is reached, at this point the counter (f_T) in the transmitter timing domain is stopped and relative frequency f can be derived by $f = \frac{f_T}{f_R}$. The counter in the transmitter timing domain is stopped by a signal from the receiver which is synchronized by Brute Force Synchronizer to the transmitter timing domain. Further to estimate transmitter's clock phase even clock cycles are sampled with receiver's clock twice, once early once late. If the samples are different, a transition obviously occured in the observed interval. To determine the detection interval of the phase a bound value for the delay used for the late sample and the cycle time of transmitter's clock is necessary. All this happens off the critical path of the synchronization process, thus has no effect on the delay of the synchronized data, which allows metastability to decay. The Even/Odd synchronizer has a low latency of $0.5 + x$ cycles on average, where x is the keep-out region of the registers, and an arbitrarily small probability of synchronization failure. The idea of the Even/Odd synchronizer has similarities to the *Two-Register-Synchronizer* of [12] without the constraints of mesochronous clocking (no phase drift, same nominal frequency) with a fixed delay of half a clock cycle.

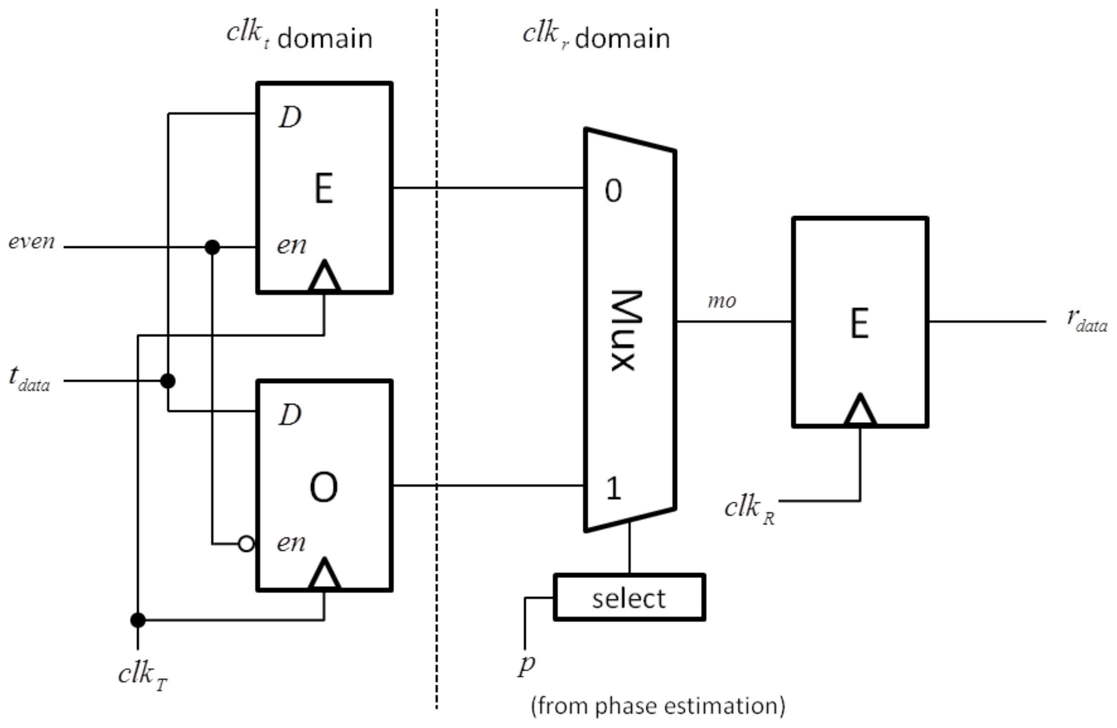


Figure 6.41: Even/Odd Synchronizer [13]: Structure

6.2.21 Four-slot Fully Asynchronous Communication Mechanism

A four-slot fully asynchronous communication mechanism is presented in [49]. This mechanism is provided as algorithm for software and hardware implementation. The basic idea is to use a shared memory communication mechanism without mutual exclusion to provide fully asynchronous data transfer. To avoid any timing interference between the communication partners the mechanism must avoid usage of control variables in conditional statements, like in use with arbiters or busy waiting to provide fully asynchronous communication. Several approaches are shown in [49] but only a four-slot solution is truly capable of asynchronous communication. Listing 6.42 shows the algorithm for the four-slot mechanism. There are two procedures, one for write access and one for read access to the shared memory. The control variables are bit variables (latest, reading, slot array). The data array consists of two fields with each two slots, hence four slots in total. The 'write' procedure sets the 'pair' variable to the complement of 'reading', the currently or latest read data pair, in the first step. Then it looks for a free slot in that pair and after that writes a data item to the memory. As a last step it sets the index of the written pair and slot for the reader. The 'read' procedure firstly gets the latest written pair and slot and then gathers the data item. Hence these procedures are forced to avoid conflicts on the shared memory (two dimensional data array) and so the four-slot mechanism forms an orthogonal avoidance strategy. This strategy directs the 'write' procedure to write the pair that is not currently read and the 'read' procedure to read the latest pair that is not currently written. Further a performance improvement to the algorithm above is given in [49], where the algorithm works with pointers instead of the data items directly. Paper [49] provides a hardware implementation of the given algorithm. A schematic of the hardware design is given in Figure 6.43. The hardware design reflects the mechanism using registers, bistables and switches. A register is a flip-flop with enable input. A bistable is a latch with enable input and two complementary outputs. Note that the negative ouput is looped back into the input to switch the binary state of the bistable on activity at the enable input. The read and write of the data items are executed simultaneously in hardware. Latching pulses guide the data item into the right register and out of them. A number of timing constraints are summarized in [49], but omitted here. Two errors can occur in the hardware implementation. First a so-called 'flicker' can occur if the reader is much faster than the writer and two read accesses can occur between consecutive write accesses. The author [49] states that this error is bounded by the write duration and is not of practical concern. The second error, called 'dither', is in contrast an unbounded failure and is caused by metastability in bistables. Thus the reader may receive an unstable or erroneous value. Unfortunately the author omits the details of an effective solution to this problem and only gives the recommendation to lower the speed and increase the length of the paths. The design is experimentally validated in detail in [49].

```
mechanism four slot;
  var data:array[bit, bit] of data := ((null,null),(null,null));
      slot:array[bit] of bit := (0,0);
      latest, reading : bit := 0,0;

  procedure write (item : data);
  var pair, index : bit;
  begin
      pair := not reading;
      index := not slot[pair];
      data[pair,index] := item;
      slot[pair] := index;
      latest := pair;
  end;

  function read : data;
  var pair,index : bit;
  begin
      pair := latest;
      reading := pair;
      index := slot[pair];
      read := data[pair,index];
  end;
end;
```

Listing 6.1: Four-slot Fully Asynchronous Communication Mechanism [49]: Four-slot algorithm

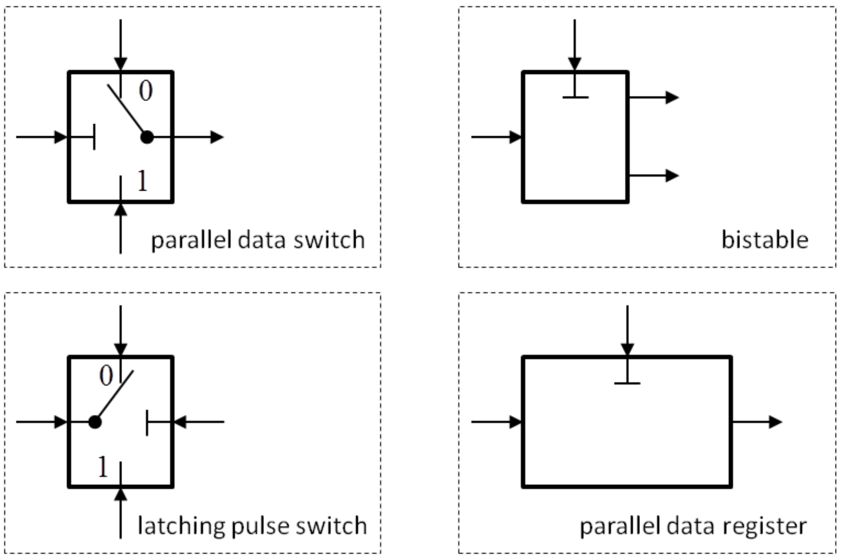


Figure 6.42: Four-slot Fully Asynchronous Communication Mechanism [49]: Basic Building Blocks

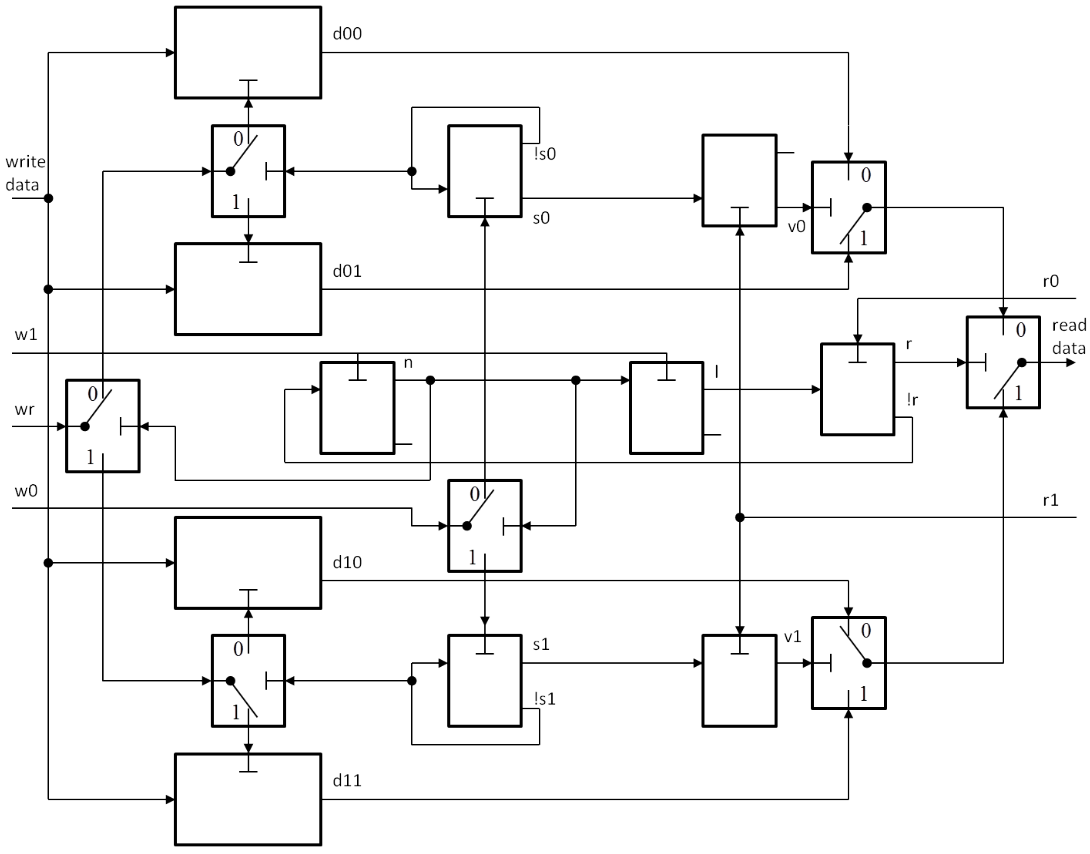


Figure 6.43: Four-slot Fully Asynchronous Communication Mechanism [49]: Schematic hardware design

6.2.22 Asynchronous Communication Mechanisms Using Self-timed Circuits

The asynchronous data communication mechanisms (ACMs) presented in [56] are based on the three-slot and four-slot fully ACMs in [49]. The algorithms (see Listing 6.1) from [49] are picked up and for each statement a circuit is designed. Additionally the accesses (write/read) to the ACM are controlled by arbiters to improved handling of the possible metastability events and constrain them to these particular points, away from data path to control logic. The developed speed independent statement circuits use a 4-phase handshake bundled data protocol for write/read access to the ACM control part as well as in the data path (see Figure 6.44). In Figure 6.44 one can see the writer starts its access by handshake request, when the ACM control part grants access, data is forwarded to the ACM data path. On reader side a similar procedure takes place. The ACM control part steers the data to the next slot. The used arbiters are enhanced with metastability detectors, or also called metastability resolvers, and are taken from [46]. As long as this arbiter detects a metastable state its outputs will not change until metastability has resolved. In the ACM the arbiter controls the execution of two statements, one from each the writer and the reader. The statement of the writer sets the index of the last written data item and the one of the reader gets the index of the lastest data item. This ensures full data coherence over temporal independence. Due to the arbiter as metastability detector the full asynchronism of the base design from [49] is no longer achieved ("theoretically"), due to the required waiting for the input that lost arbitration (such an temporal relation causes the loss of full asynchronism [56]); although the arbitration only takes place outside the ACM data path, hence in the control part. To maintain a certain degree of temporal independence in this approach a more flexible handshake protocol is employed. After the writer/reader initiates a data transmission, it waits for the acknowledgement from the control part, but the writer only waits for a predefined time period, by this point in time the writer continues its operation. This maximum waiting period is obtained by simulation of the algorithm and measuring the execution time of each statement and an additional amount of time for metastability resolution (about 5 ns in modern CMOS systems). In [56] a circuit for each statement is provided, details are omitted here, since the single statements only involve latching data. For this, transparent latches are introduced and further handshake decoupling elements from [54] are provided which use David's Cells to control handshakes. Each David's Cell consists of a flip-flop and a NOR gate. In contrast to the four-slot ACM the three-slot ACM only provides full asynchronism, full data coherence and data freshness if the statements involving the index of the last written data item are atomic to each other. This is easily achieved by the used arbiters. The benefit of the three-slot ACM is the maximization of operation speed. The design of the three-slot ACM is given in Figure 6.45. The access of the write and read control processes the data movement from writer to the reader, their accesses are controlled by the arbiter. The 'differ' block reads the indices of latest written and read values and negate them to provide a new indices pair for the next write access. Additionally simulation results and analysis of the presented solution are given in detail in [56].

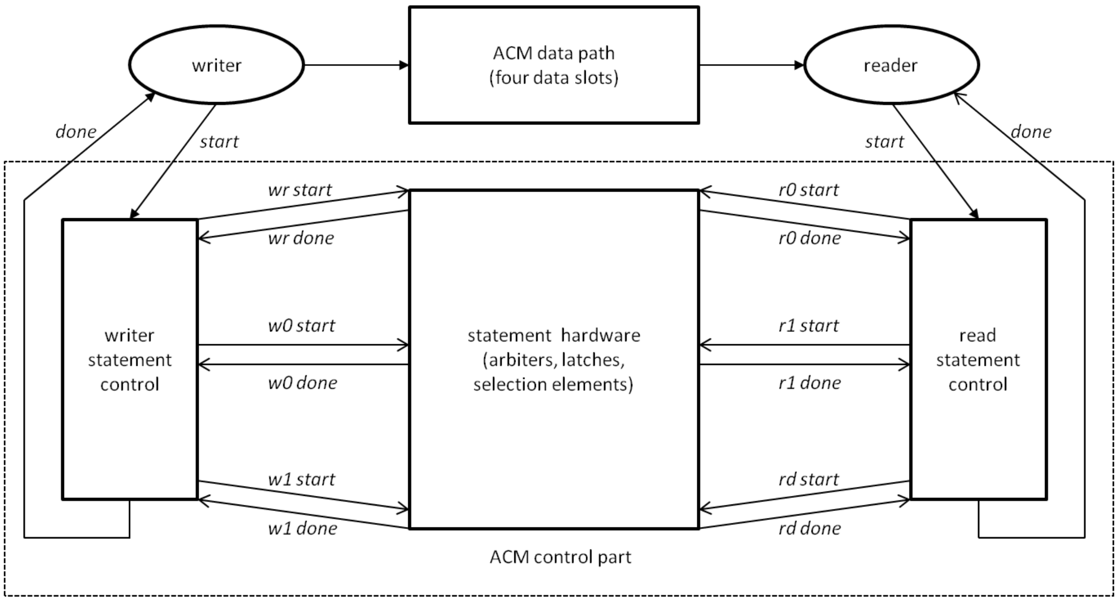


Figure 6.44: Asynchronous Communication Mechanism Using Self-timed Circuits [56]: Basic structure of modified 4-slot ACM with SI circuits

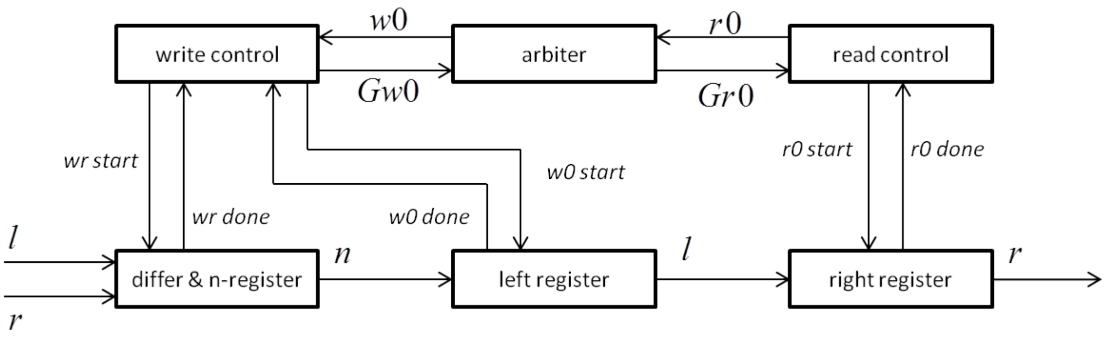


Figure 6.45: Asynchronous Communication Mechanism Using Self-timed Circuits [56]: Basic structure of 3-slot ACM design

6.2.23 Implementing a STARI Chip

The *STARI* (Self-timed at Receiver's Input) [29] interface uses synchronous as well as self-timed (asynchronous) circuits. The transmitter and receiver operate with a global clock and communicate via a self-timed FIFO. *STARI* does not use low-level flow control. This technique is able to compensate clock skew (drifts in relative phase) and variations in delay on data items, it tolerates up to 1.5 cycle lasting jitter [7] and is often used with mutually asynchronous clocks. The intent of using a self-timed FIFO is to avoid an explicit handshake mechanism between FIFO and transmitter and get rid of the round-trip delay that comes along and thus the limitation of the bandwidth of the channel. To assure this the FIFO has to appear synchronous to the transmitter and the receiver. To appear synchronous the FIFO must complete a write as well as a read operation within one cycle of the global clock. Furthermore the transmitter and the receiver appear as self-timed communication partners to the FIFO using the *STARI* signaling technique. There is only one synchronization process, at the initialization of the FIFO when the first data packet arrives. The receipt of the first data packet can occur at an arbitrary point in the clock cycle of the receiver and thus hold the possibility of metastability. To avoid this a cascade of synchronizers is employed. Each new data packet sets the latch (see Figure 6.46), its output increments the write address of the FIFO synchronized to the receivers clock. The receiver only needs to know the position of the first data item in the FIFO to have a point where to start from, this happens during initialization. After a reset the FIFO is empty, when the transmitter starts its transmission it signals the receiver for an appropriate number of clock cycles (equal to synchronizer depth) that the first data packet is sent. The receiver waits until the FIFO is half-filled $\frac{(n+1)}{2}$, where *n* is the number of stages within the FIFO, before reading data from it. The half-full FIFO is needed to avoid an overflow or underrun and is used to maximize throughput and robustness, this is guaranteed through static timing analysis. So with a *STARI* interface the receiver has to wait $S + \frac{(n+1)}{2}$ cycles until receiving first data, after this initialization the receiver accepts a data item every clock cycle, resulting a throughput of 120 Mb/s [29]. *STARI* avoids double-buffering (as in [43]) and synchronization and thus is free from timing hazards during operation. The FIFO in detail uses inverted dual-rail data encoding with a four-phase handshake protocol. In [29] a detailed comparison is provided between different implementation variations.

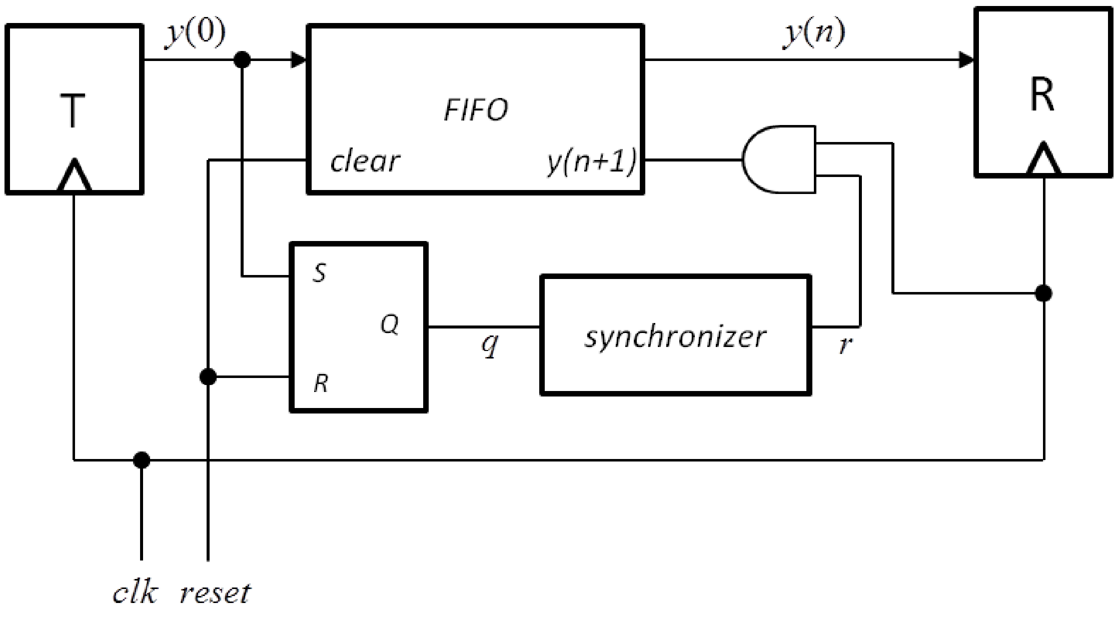


Figure 6.46: STARI [29]: Structure

6.2.24 Efficient Self-Timed Interfaces for Crossing Clock Domains

The source-synchronous communication interface presented in [29] called STARI, short for *Self-Timed At Receivers Input* is improved and enhanced in [9] and [8]. There are solutions for mesochronous, plesiochronous systems as well as systems with rational clocking or arbitrary clocking. First, STARI itself is a mesochronous interfacing technique, there is one global clock and a fixed phase drift between receiver and transmitter clock. In [9] and [8] the STARI approach is first improved to only using a special latch instead of a FIFO. The special latch in this minimal source-synchronous interface employs a generic latch (single-stage FIFO) and a latch controller that generates the clock for the latch upon the product of the clocks from transmitter and receiver. The latch controller is basically a Muller C-Element, after both, receiver and transmitter clock events, the latch controller generates the corresponding event (falling or rising edge) to clock the latch (see Figure 6.47). To satisfy setup and hold requirements of the transmitter and receiver latches their input clocks are delayed within the latch controller. For further details on timing, circuit and its correctness see [9]. The interface can be initialized for maximum robustness or minimum latency. To employ the latter the "transmitter-last" mode is used. The "transmitter-last" mode describes the scenario that the last clock event before the generated clock event from the latch controller occurs is from the transmitter. The "receiver-last" mode refers to the other case. To achieve maximum robustness to skew variations the interface must run in the mode that tolerates the largest skew change in either direction. If the time between a rising edge of the transmitter clock to the next rising edge of the receiver clock is greater than the time between a rising edge of the receiver clock and the next rising edge of the transmitter clock then the "transmitter-last" mode is optimal, the "receiver-last" mode is used in the opposite case. To achieve these modes an adjustable delay is required within the latch-controller (self-reset cycle), which is gradually decreased during initialization until a point is reached where one of the two modes is feasible. During this process metastability can occur. To support rationally related clock frequencies the previous single stage FIFO interface has to be extended by another module, the *rate-multiplier*. The *rate-multiplier*, based on an algorithm provided in [9], is only employed at the communication partner that operates at the higher clock rate. It generates a derived clock from the source clock that is provided to the latch-controller. The *rate-multiplier* outputs n pulses of the slower clock for every pulse of the faster clock to the latch controller. This clock is an estimate of the clock of the slower running communication partner. Further the derived clock signal is used as a valid signal for the faster module to indicate when it is safe to latch new data. The latch controller in the rational approach can miss an event (either transmitter or receiver clock) during its self-resetting phase. To avoid this a miss-detector is employed that indicates the misses or near-misses and reports them to the receiver and transmitter. Furthermore a plesiochronous interface can be realized, where the frequency of receiver and transmitter clocks are closely matched, but the relative phase shift between them is slowly drifting. The interface uses a modified miss detector that indicates when either a rising edge from the transmitter clock or the receiver clock occurs shortly after the self-reset of the latch controller. To detect near-misses a delayed version of the generated clock from the latch controller is used. Near-misses indicate misses thousands of cycles in advance, giving the module that causes the near-miss a chance to react, hence to skip clocking the latch controller for the next cycle. To implement such a mechanism "stuffed bytes" are used, these are extra bytes at the end of a data item. Additionally the interface for the rational clocking and the plesiochronous interface can be combined to support communication between systems with arbitrary clock frequency. First of all the communication partners forward their clock to each other to derive an estimate using a counter and further provide a rational approximation. This is necessary if the clocks are unknown in advance. The communication partner with the faster clock employs a rate-multiplier module to create an approximation of the slower clock (as described above). Due to the fact that the used frequency values are only approximations the used FIFO may underflow of overflow. Near-miss signals are generated and forwarded to the rate-multiplier module to update the frequency estimation. The interfaces for rational clocking, plesiochronous and arbitrary clocks do not transfer exactly one data item on every clock cycle thus an additional receiver FIFO (synchronous FIFO clocked by the receiver) is required. If the FIFO is not empty new data is placed in the receiver FIFO, if the FIFO is empty it is bypassed by a multiplexer directly providing the new data item to the receiver.

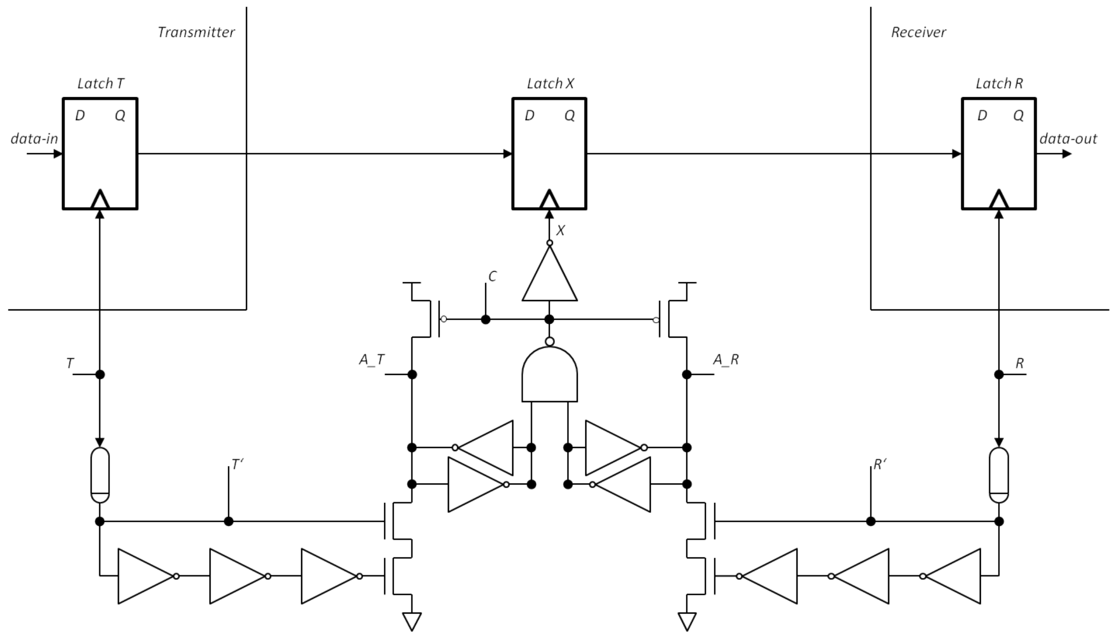


Figure 6.47: Efficient Self-Timed Interfaces for Crossing Clock Domains [9]: Simple Single Stage FIFO Latch Controller

6.2.25 A Predictive Synchronizer for Periodic Clock Domains

A **Two-Way Adaptive Predictive Synchronizer** is presented in [25], which synchronizes bi-directional communication between systems within periodic clock domains by taking advantage of the periodic nature of their clocks. The synchronizer assumes that a conflict occurs when an integral number of cycles of the local (receiver) clock span the same time as the number of cycles of the external (transmitter) clock. This assumption is used to predict conflicts and to control the send/receive output that signals when it is safe to send/receive to avoid duplicates and misses in data transfer. More precisely the sampling of the input is delayed by keep-out time T_{ko}, which is controlled by the keep-out controller to signal whether it is safe to receive data or not. Conflict prediction is achieved by creating a *predicted clock* (by Δ delayed external (transmitter) clock). Thus the predicted and local (receiver) clock will conflict one T_{local} cycle ahead from a "real" conflict between transmitter and receiver clock. A term is introduced in [25], *d-conflict*, to define the occurence of two clock events of different clocks within an interval of length *d*. The structure of the predictive synchronizer (see Figure 6.48) consists of a *d-conflict* detector (see Figure 6.50), a programmable delay line (T_{local}-delay, part of Figure 6.51), an adaptive clock predictor (see Figure 6.51) and a conflict prevention circuit (keep-out controller and clock select circuit, see Figure 6.49). There are three different versions of the conflict detector, in this context the *d-conflict* detector is presented. As mentioned before the *d-conflict* detector determines that two clock events occur within an interval of *d* or that one of these clock events precedes the other. The circuit operates as follows, two flip-flops sample the clk_2 input *d* time after and before the rising clock event of clk_1 input. To cope with susceptibility to metastability there are two additional flip-flops (FF3 and FF4) employed for metastability resolution, this results in a half clock cycle for metastability to decay. If the rising edge event at the clk_2 input occurs within the sampled *2d* interval a conflict is detected one clock cycle in advance (due to usage of the delayed external (transmitter) clock). The adaptive clock predictor circuit employs two adaptive delay lines (T_{local} and Δ) which are programmable via digitally tapped inverter chains and dynamically tuned. Each delay line starts with a zero (minimal) delay and in- or decreases in steps of size *q*, where *q* is the adjustment step size and must be smaller than the time resolution of the conflict detector. The Δ-delay line is adapted if the conflict detector predicts a conflict or one of the compared clock events precedes the other ($clk_1 \gtrless clk_2$). Further the adaptive clock predictor employs a *rate reducer*, that reduces the output rate to a slower clock (either external or local), the employed flip-flops are doubled as at the conflict detector for metastability resolution. A formula for the total tuning time of the clock prediction is given in [25]. The adaptive clock predictor generates a delayed version of the external clock that periodically precedes its original version by T_{local}. The conflict prevention circuits, keep-out controller and clock select, employ a *d-conflict* detector which generates a keep-out signal indicating a conflict between local and predicted clock, which is further taken to select either local clock or a delayed version of the local clock in case of a conflict to finally generate the receiver clock. The *d-conflict* detector may become metastable if two rising events on the input clocks occur about *d* time from each other. The additional flip-flop enables one half clock cycle for metastability to decay (latency), this results in a MTBF of 10^{16} years. To add more resolution time for metastable states (in case of fast clock frequencies) further flip-flops can be added (which results in an extended prediction time). Also the rate reducer may become metastable, there is one full cycle available for metastability to decay, thus the MTBF is higher than MTBF of the detector. If metastability resolves either to logical zero or one the only effects are extension of convergence time or small variations in the timing of the predicted clock, thus no malicious behaviour results from a wrong decision after a metastable state. A formal proof and verification of the two-way predictive synchronizer is given in [25].

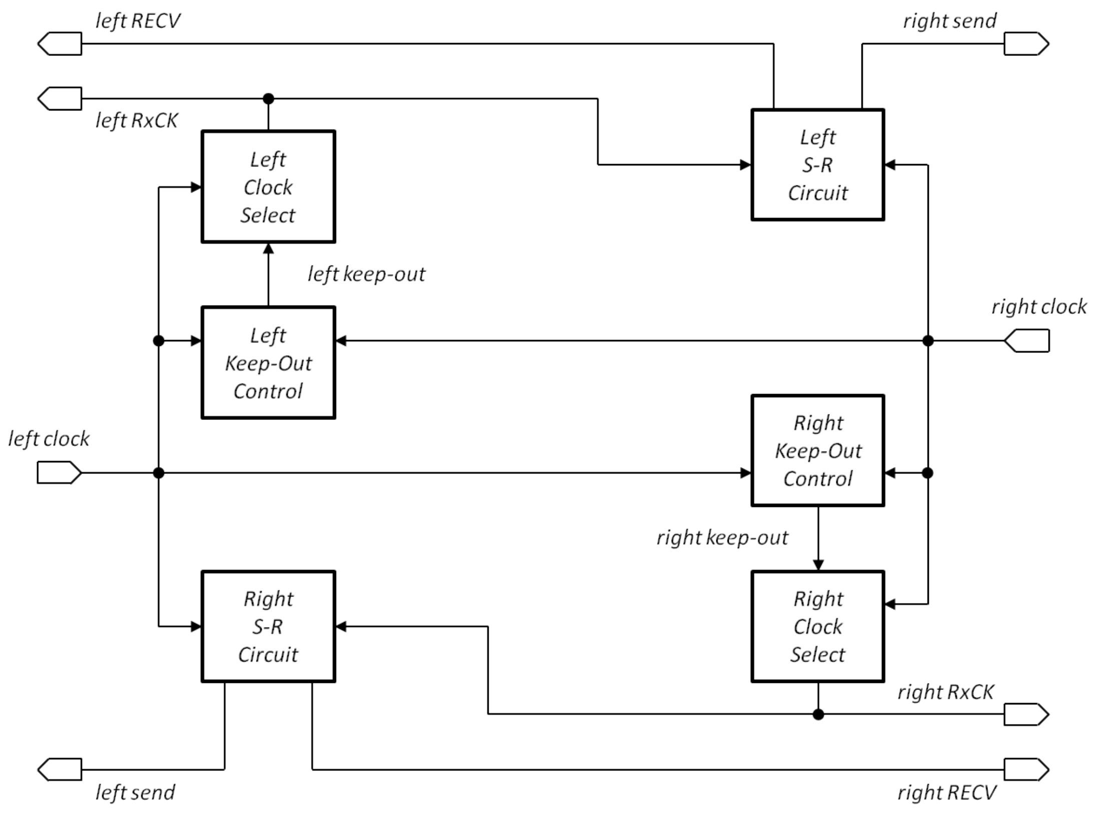


Figure 6.48: A Predictive Synchronizer for Periodic Clock Domains [25]: Architecture of Two Way Predictive Synchronizer

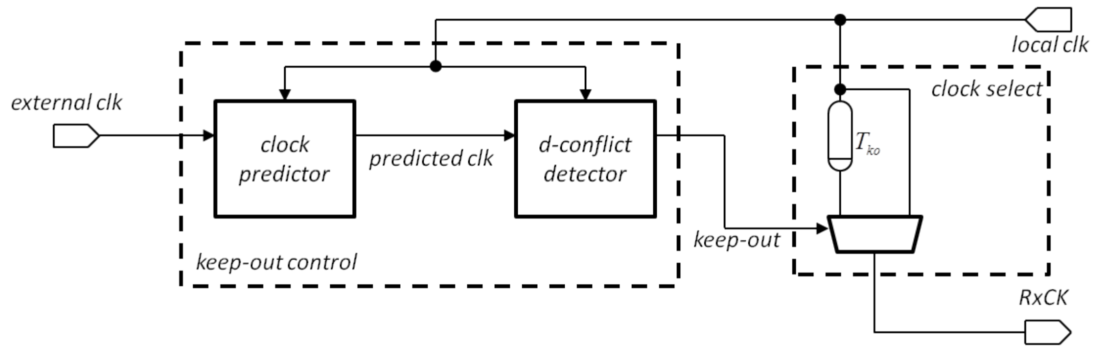


Figure 6.49: A Predictive Synchronizer for Periodic Clock Domains [25]: Keep-Out/Clock-Select Circuit

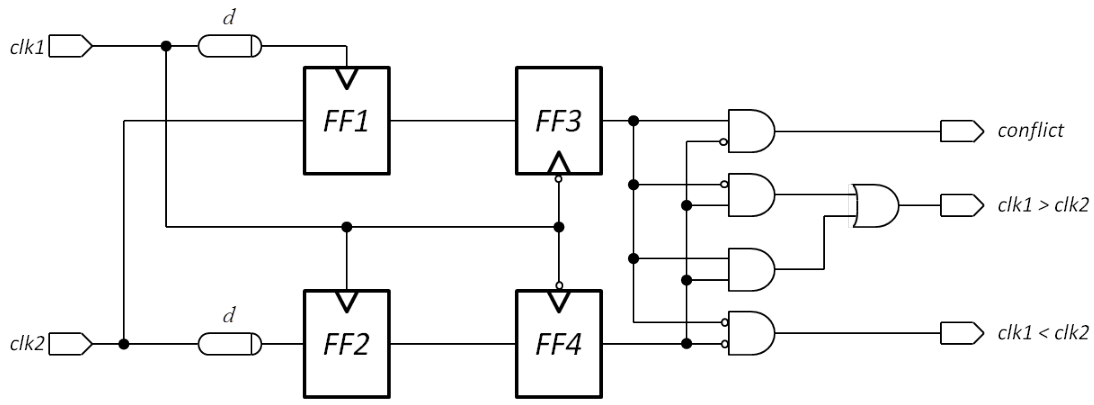


Figure 6.50: A Predictive Synchronizer for Periodic Clock Domains [25]: d-Conflict Detector

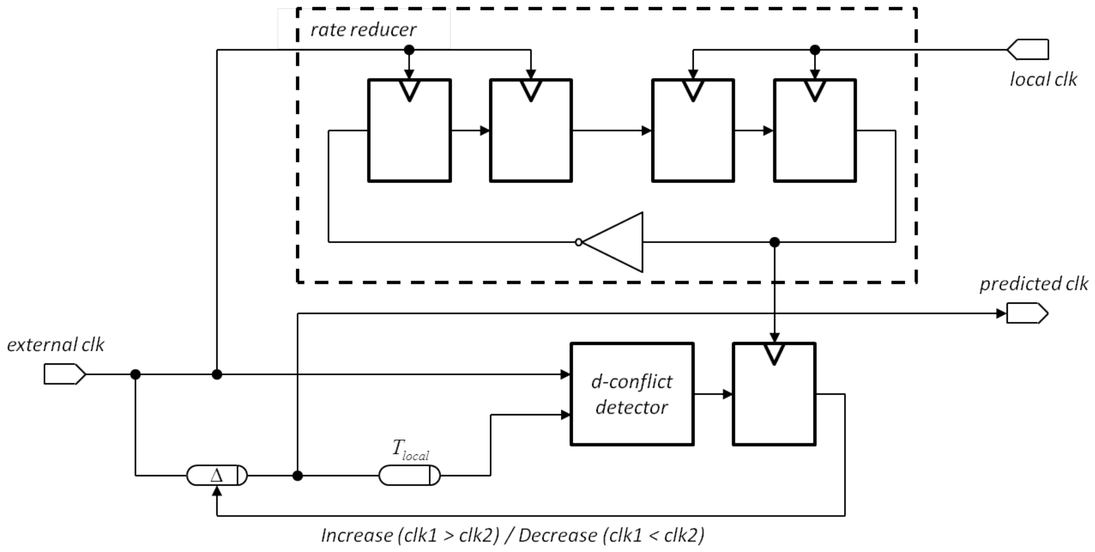


Figure 6.51: A Predictive Synchronizer for Periodic Clock Domains [25]: Adaptive Clock Predictor Circuit

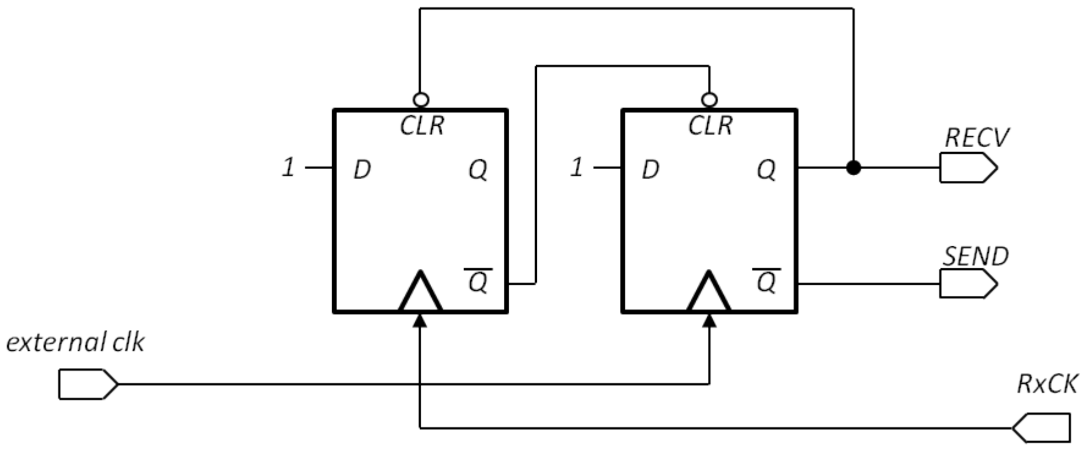


Figure 6.52: A Predictive Synchronizer for Periodic Clock Domains [25]: SR-Circuit (Duplicate/Miss Controller)

6.2.26 A Solution to a Special Case of the Synchronization Problem

In [51] a *Brute Force Synchronizer* is called a "shift-register" synchronizer and is described as a cascade of *n* D-flip-flops clocked by the receiver's local clock. A periodic synchronizer is proposed, it exploits the periodicity of the receiver's and transmitter's clock and their predictability of near-conflicts and performs the synchronization in advance. The clocks are not related, but must be known in advance. The proposed synchronizer has an improved synchronization delay compared to the shift-register synchronizer, such that the synchronized signal lags never more than a fraction of a clock period of the receiver's local clock.

A system employing this periodic synchronizer operates as follows: the transmitter generates a signal *x* which should be transmitted to the receiver and synchronized to result in a signal x_{safe} that is safe to sample. The synchronization is done off the critical path by creating a window signal within the local clock domain which is a chain of pulses with a duration τ_l and a period T_l (same as local clock) that shows that x_{safe} must not be changed to stay safe. Further a second window signal is created but within transmitter (remote) clock domain. The latter signal is a chain of pulses with a duration of τ_f and a period of T_f (same as remote clock) and each pulse shows that the signal *x* may change such that it cannot be sampled safely. These two window signals are merged by an AND gate and only if they overlap a synchronization failure can occur. The resulting signal is monitored by a pulse detector, namely a D-flip-flop that is cleared on overlapping window signals. Due to the asynchrony of the two window signals runt pulses may occur at the output of the AND gate. Thus a shift-register synchronizer to decay a metastable state within the pulse detector may be needed. When the two signals do not overlap, the pulse detector is set statically to logical one providing a safe signal that shows that it is safe to sample. Additionally an enable signal is generated by the local clock that consists of pulses of duration τ_e and period of the local clock. The safe and enable signals are merged by an OR gate to control a D-flip-flop that samples signal *x* into the receivers domain (resulting in x_{safe}). The mentioned pulse widths (τ_l,τ_f,τ_e) are important parameters and depend on the frequency of local/remote clock. τ_l states how long the data signal *x* must be stable before the rising edge of the local clock. τ_f states the last point in time after a rising edge of the transmitter clock when a transition of signal *x* can occur. The pulse width τ_e depends on τ_l and τ_f and is defined as $\tau_e < T_l - (\tau_l + \tau_f)$. Notice that the window signal of the transmitter domain is delayed by τ_d to adjust the phase between the two window signals and model the relationship between the receiver and transmitter clock *d* periods in advanced. The delay τ_d is fixed and defined by $\tau_d = nT_f - (dT_l - \tau_l)$ where *n* is chosen to be the smallest positive integer that results in $\tau_d > 0$. For details of the derivation see [51]. A drawback is that the delays must be exactly adjusted to the used clocks, and that the frequency must be known in advance. A next step would be an adaptive periodic synchronizer where the frequencies of the clocks are permitted to move within a given range.

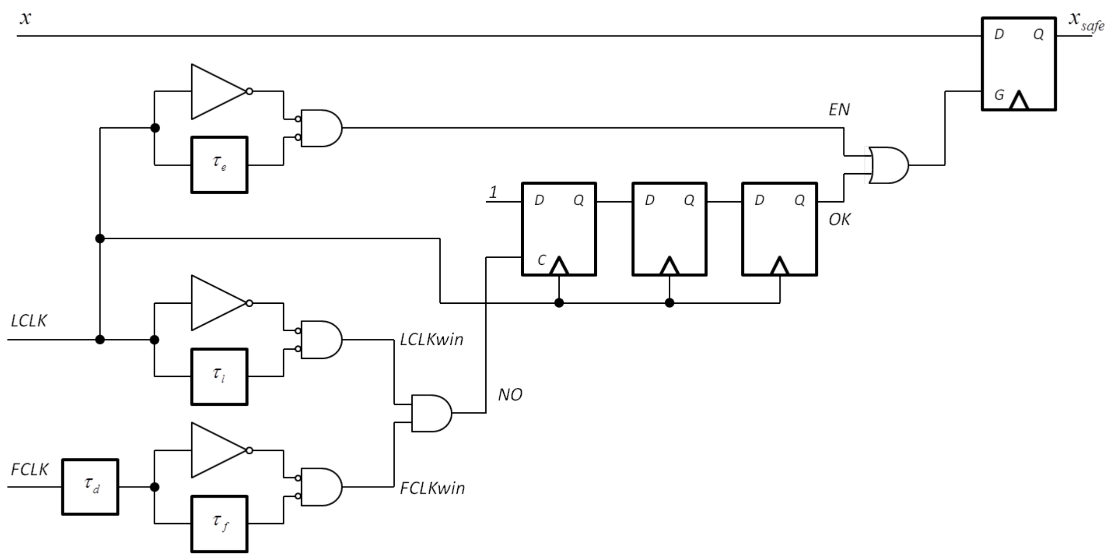


Figure 6.53: A Solution to a Special Case of the Synchronization Problem [51]: Periodic Synchronizer Circuit

6.2.27 On-Chip Segmented Bus: A Self-timed Approach

Another solution to realize the interconnect/communication interfaces between the modules of a GALS system is presented in [45]. A segmented bus architecture designed in a self-timed fashion is presented. The presented bus is split in several (at least two) segments, which are dynamically interconnected such that every module of a segment can reach every other module in the system independent from its location (same or different segment). The different bus segments operate concurrently (concurrent communication). Each bus segment contains three component groups, masters, slaves and arbiters and an inter-segment bridge. A master requests a connection to a slave to send or receive data. The arbiter grants mutual exclusive access (read or write) to a slave. There are two arbiter types employed in the segmented bus, the local arbiter and the central arbiter. The local arbiter controls the access within a bus segment and the central arbiter controls the access inbetween the bus segments. The modules that frequently communicate are mounted into the same bus segment to provide the optimal solution for a high communication probability. The segmented bus uses a communication protocol based on request and acknowledgement handshaking. The request line from each master module are sequentially polled by the local arbiter. A master asks for the ownership of its segment by asserting its request line. The request is forwarded to the local arbiter, if no other master requests the ownership or it has the highest priority the local arbiter sends an acknowledgement back to the requesting master and grants access to the segment. The protocol to obtain ownership of another segment is quite similar. The master sends its request to the local arbiter but with an address of a foreign segment. The local arbiter forwards such a request to the central arbiter, which further moves the request to the desired bus segment and its local arbiter, if the target segment is currently not busy, this information is obtained from the *Segment Register* of the central arbiter. An external request has always the highest priority, currently active connections will be finished first. If the foreign bus segment has acknowledged the request for ownership the connection is built up between the inter-segment bridge and the central arbiter at each communication partner. The inter-segment bridge is composed by tri-state buffers and logical building blocks and is used to dynamically build up and close the physical connection to the central arbiter. Measurement results of the self-timed bus are given in [45], the maximum polling frequency is about $425\ MHz$ but there is no given lower bound, thus no timing constraints can be met. The segment bus provides about 20% improvement in comparison to a single bus communication mechanism. The presented solution is similar to a Network-on-Chip (NoC), thus focuses more on the communication scheme than on the interfacing technique.

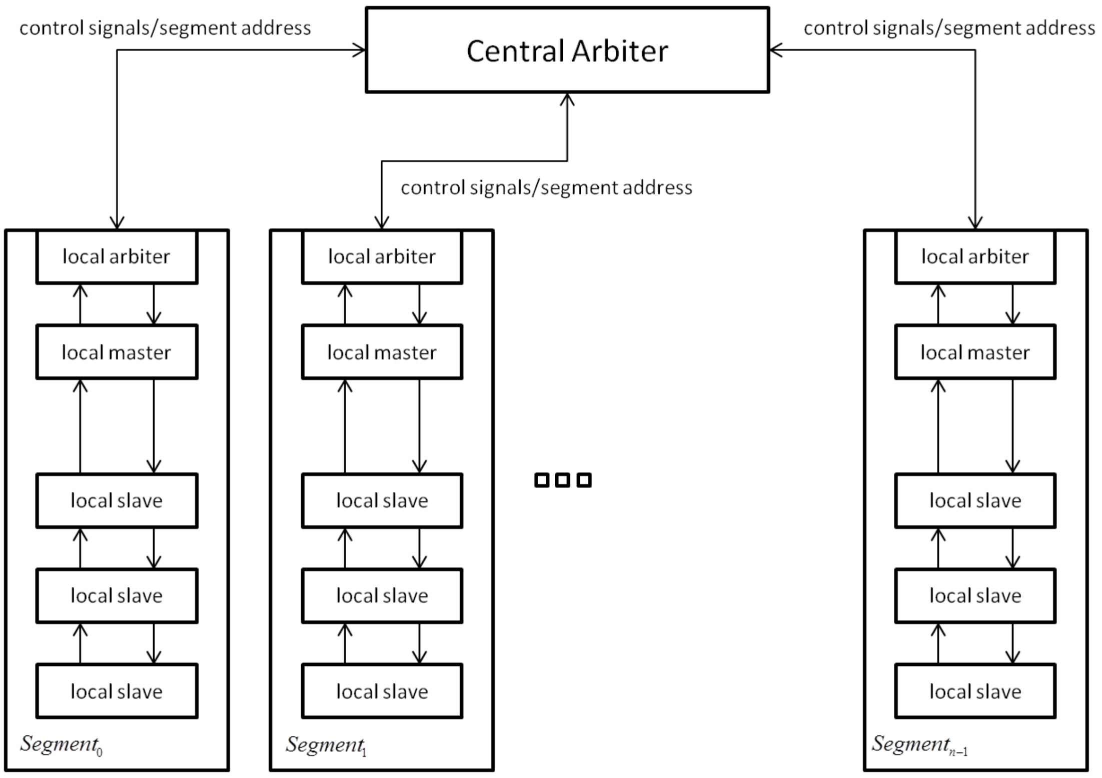


Figure 6.54: On-Chip Segmented Bus - A Self-timed Approach [45]: Bus Structure

CHAPTER 7

Synchronizers for Self-Timed Systems

This section describes synchronizers for *self-timed systems*. This refers to systems where computation and communication is not timed by a clock reference, but defined by a schedule of events.

7.1 Basic Concepts

The interfaces presented in this chapter are used to synchronize signals, originated in clocked as well as self-timed systems, into the working cycle of a self-timed system. These interfaces can be classified by four basic concepts. The most basic one is the **Arbiter**, it grants mutually exclusive access to a shared resource, such that an external request can not interfer with another input or the working cycle of the target system itself (assuming that it was granted access first). The *Synchronization and Conversion Circuits* and the *Glitching Synchronizer* are both used to convert incoming signals to a self-timed protocol and from single-rail to dual-rail, but there are no special mechanisms to synchronize incoming data to the working cycle of the receiver system. Thus when these **protocol converters** are needed, e.g. when interfacing a clocked module and a self-timed module using LEDR [14], they should be combined with an additional synchronizer or at least an arbiter. The delay-insensitive chip area interconnect called *Chain* provides a special self-timed *pipeline latch*, that is used as a basic building block to construct network components such as router, multiplexer or arbiter for a **self-timed data bus**. At last the *Modular Synchronizing FIFO for NoCs*, the *FIFO Ring Performance Experiment*, the *GasP* approach and the *Doubly-Latched Asynchronous Pipeline* use a FIFO controlled by a **handshake pipeline** (e.g. a *Micropipeline*) to form a communication channel between self-timed systems. As in the chapters before the synchronizers are compared amoung each other within the categories to derive a prototype for each category, with the difference that the MTBF equation from section 1.3 cannot directly be applied, since there is no periodic clock anymore. Further more most of the self-timed systems are delay-insensitive or quasi-delay-insensitive and therefore latency is not as valueable for the comparison. Note that the latency of self-timed interfaces is often omitted in literature. So the synchronizers are only compared in terms of throughput and elegance of the

solution.
The most basic concept is the *Arbiter*, it is used to grant mutually exclusive access to a system or shared resource. It can be used to synchronize either self-timed or clocked signals to a self-timed system. Although its design is optimized to avoid metastability as far as possible there is inevitably a residual risk of getting metastable. For the arbiter this happens when both request inputs are asserted at the same time. However, even when the arbiter is internally metastable, this metastable state is not exposed at the output. In that case the decision of whom to give the grant is simply delayed [40]. It has no direct opponent and thus is noncompetitive. In the category of synchronizers which use a FIFO that is controlled by a handshake pipeline as self-timed interconnect the *Modular Synchronizing FIFO for NoCs* provides the most modular design with the highest throughput of $2 - 2.5\ Gtps$. It provides *get* and *put* interfaces for self-timed as well as clocked systems to use the self-timed *asP** FIFO pipeline. The following sections provide detailed descriptions and references for the comparative parameters of the chosen prototypes.

7.2 Detailed Descriptions

7.2.1 Arbiter

An arbiter as described in [12] and [33] is used to grant mutually exclusive access to a resource shared among different clients upon an incoming request. No other input request is granted until a preceeding (granted) request is dropped again. A request may be enhanced with a priority value to signal the arbiter the urgency of the request (needs additional logic), or it is simply first-come-first-serve principle in use. Thus in self-timed systems an arbiter can be used as interface, to synchronize incoming (asynchronous) signals to the internal operation cycle. Arbiters can be part of different interfaces and building blocks as e.g. synchronizers using *pausible clocking* and *completion detection* circuits [12]. The circuit and electronic symbol of an arbiter are depicted in Figure 7.1, the shown arbiter is a cross-coupled NAND-Arbiter.

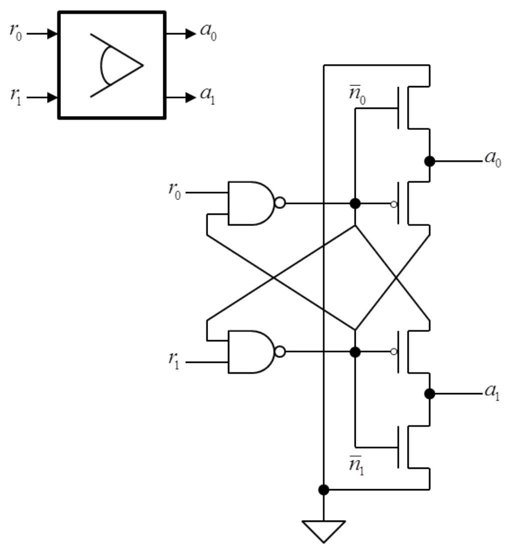


Figure 7.1: DSE [12]: Arbiter Circuit

7.2.2 Glitching Synchronizer

The *Glitching Synchronizer* [24] is a phased logic interface that synchronizes incoming (clocked) data to the internal phase of a self-timed module and generates a valid phased logic dual-rail signal. The circuit of the *Glitching Synchronizer* can be seen in Figure 7.3. It acts as an interface to the self-timed module (see Figure 7.2, the handshaking path is omitted in this figure). Note that the system shown in Figure 7.2 with only one data line is often used for simple communication protocols, further the synchronous system does not forward its clock to the self-timed system (in System-on-Chips (SoCs) and for serial communication protocols the clock would be required at the receiving self-timed module). The incoming data is latched by two registers in a row that are alternately triggered by the internal *phase* signal. Data is provided to the system via two lines, *PL.Line0* and *PL.Line1*. The *PL.Line0* forms the XOR product of the current phase and the sampled data, while *PL.Line1* directly shows the sampled data. The used XOR gate always will produce glitches. The author in [24] states that in practice these glitches have enough time to decay because they occur at the beginning of a new phase where the phased logic module is not ready to latch the new data. The output must be glitch-free to maintain the delay-insensitivity property. The *Glitching Synchronizer* can be extended to definitely eliminate these glitches. The *Double-Edge Flip-Flop Synchronizer* takes the *Glitching Synchronizer* as basis and extends it by a third register after the XOR gate, one can see the circuit in Figure 7.4. The third register makes it necessary using double-edge triggered flip-flops, because the interface output needs to change with each transition on the *phase* signal. Further it is crucial that the XOR gate introduces a delay that is long enough to avoid a change of the phase output (*PL.Line0*) of the third flip-flop within the same phase transition. Only one rail is permitted to change its value at a time when a LEDR protocol is employed. This also can be handled by pre-calculated phase output values that are selected via a multiplexer that is controlled by the phase signal.

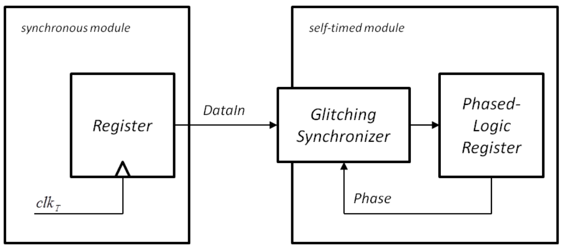


Figure 7.2: Coupling Asynchronous Signals into Asynchronous Logic [24]: System Overview (Synchronous Module → Self-Timed Module, without clock)

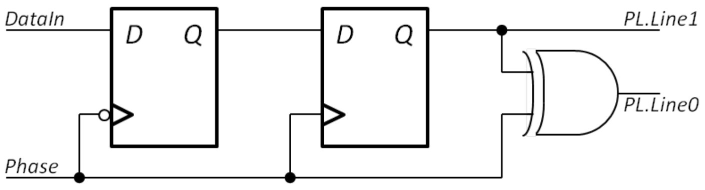


Figure 7.3: Coupling Asynchronous Signals into Asynchronous Logic [24]: Glitching Synchronizer

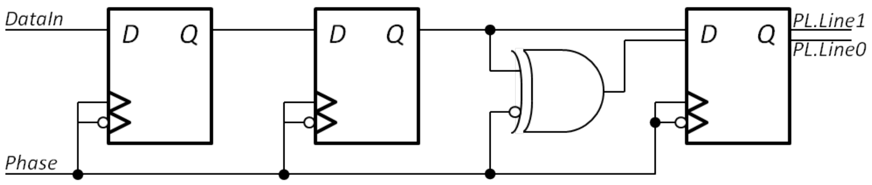


Figure 7.4: Coupling Asynchronous Signals into Asynchronous Logic [24]: Double Edge Flip-Flop Synchronizer

7.2.3 Synchronization and Conversion Circuit

The *Synchronization and Conversion Circuits* presented in [24] form an interface that synchronizes a single-rail (synchronous or asynchronous) signal and converts it to a dual-rail signal. It is based on the *Glitching Synchronizer* presented in the previous section and extends its circuit to derive a delay-insensitive implementation. The circuit can be seen in Figure 7.5, the handshaking path is omitted as at the *Glitching Synchronizer*. It is used as an interface for a self-timed module like the *Glitching Synchronizer*, see Figure 7.2. It employs several building blocks, firstly the *sync* block, it consists of two cascaded D-flip-flops, one triggered by the falling and one triggered by the rising edge of the phase input signal. It synchronizes the incoming data to the internal phase and passes it to the *data* block and the *phase calculation* (φ_0/φ_1) block. The *data* block stores the data value for the data output line (*PL.Line1*) and the phase calculation circuit, it uses a D-flip-flop triggered by the rising edge of phase input signal. Note that the data is delayed by a full cycle of the phase input signal. The *phase calculation* block calculates φ_0 and φ_1 signal for phase output line (*PL.Line0*). It consists of two D-flip-flops, one samples the data (φ_0) from the *data* block and the other one samples the inverted data stream directly from the *sync* block (φ_1). The outputs are provided once to the multiplexer block as inputs and also to the *sel* block. Note that in phase φ_0 the values of both output lines are equal in contrast to phase φ_1 where the value must be different, further the phases must alternate thus only one output line changes at a time. The *sel* block generates the selection signal to control the multiplexer which selects between (precalculated) φ_0 and φ_1. The phase input signals from the *phase calculation* block are merged by an XOR gate and latched into a D-flip-flop. The output of the flip-flop is merged with the phase input signal by an AND gate forming the selection signal that controls the multiplexer. At the end of the interface the *MUX* block, controlled by the signal from the *sel* block, switches between the two precalculated phase values for the phase output line (*PL.Line0*). (redundant logic should by employed to avoid static-1-hazards at the multiplexer)

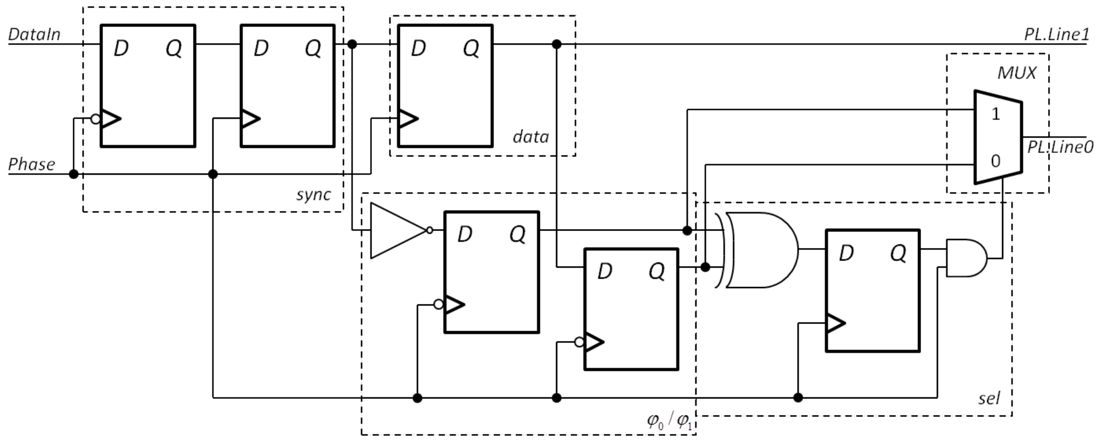


Figure 7.5: Coupling Asynchronous Signals into Asynchronous Logic [24]: Synchronization/-Conversion Circuit

7.2.4 A FIFO Ring Performance Experiment

In [35] and [36] an experiment shows that the performance of an asynchronous FIFO that uses a pulse-like protocol to advance data (*asP**) is equal to or greater than that of a clocked shift register. For the experiment a FIFO ring of 17 stages with a 4-bit wide data path was constructed. A control circuit containing three NAND gates per stage is presented that controls the move operation of a data item through its stage and shows the control stages (full or empty), see Figure 7.6. The **a**synchronous **s**ymmetric **p**ersistent **p**ulse **p**rotocol, short *asP**, a pulse-like protocol is used to efficiently move data items from one stage to the next. In Figure 7.7 the structure of the first and last stage of the FIFO ring is depicted. Inbetween these two stages an interface to insert and read data to and from the ring is employed. Thus to do the experiment and the associated measurements each stage has a control block and a 4-bit D-Latch that holds the data. This solution achieves a maximum throughput of 930 to 1126 million data items per second. The FIFO ring performance experiment is enhanced in [36], where a two-rail implementation of a micropipeline is shown and a comparison between it and a synchronous FIFO ring is given, with the result that the two-rail micropipeline operates faster than the conservative solution.

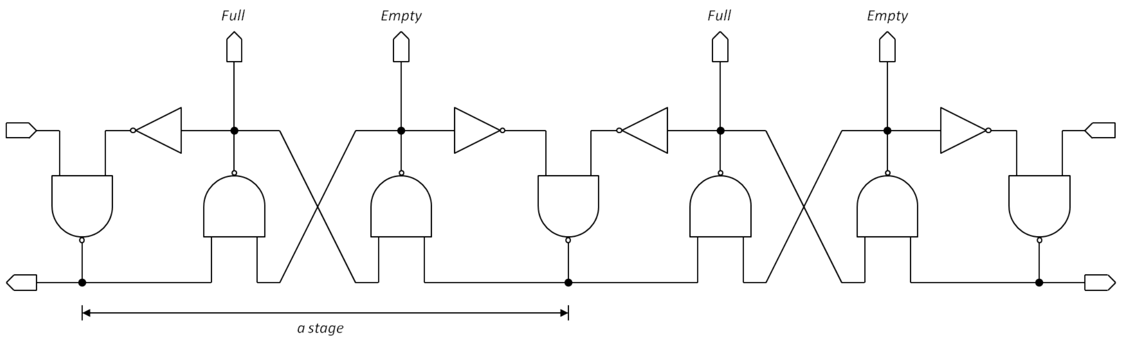


Figure 7.6: A FIFO Ring Performance Experiment [35]: Alternative Control Circuit

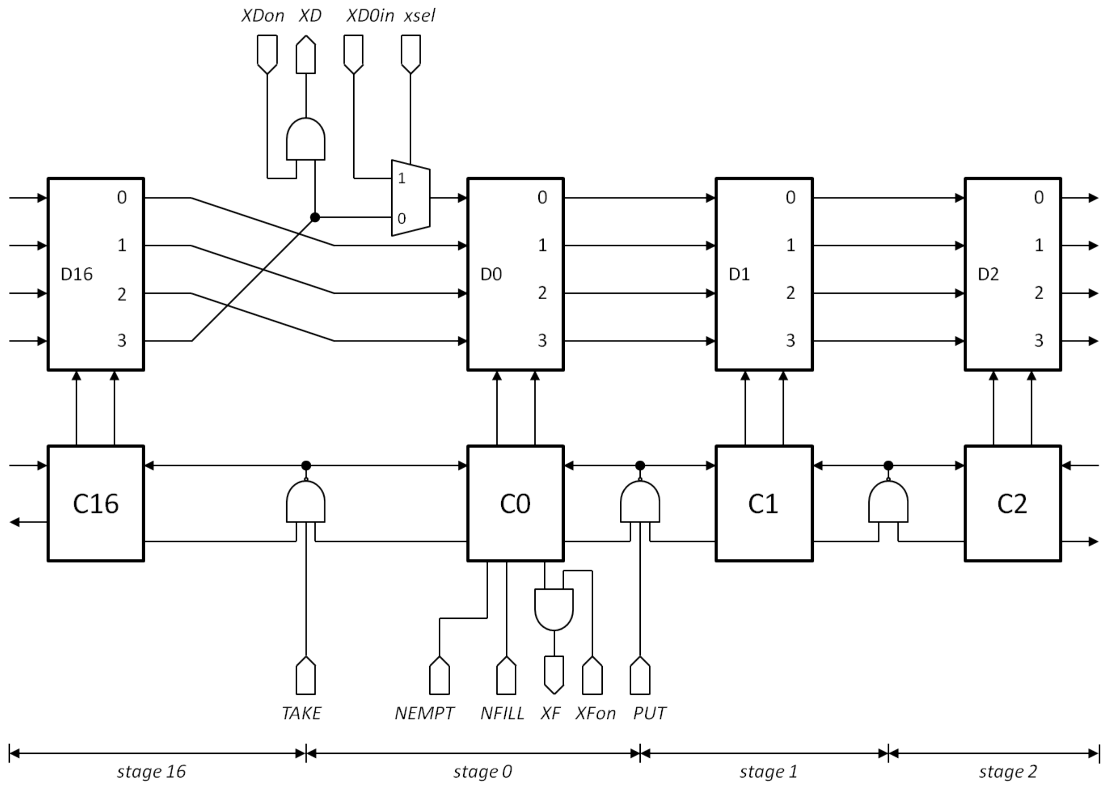


Figure 7.7: A FIFO Ring Performance Experiment [35]: Four stages of FIFO ring

7.2.5 A Modular Synchronizing FIFO for NoCs

A modular synchronizing FIFO is presented in [39] that forms an interface between either synchronous (clocked) and asynchronous (self-timed) modules and an asynchronous Network-on-Chip (NoC). This is realized through interchangeable put (write) and get (read) interfaces, which form the input and output interface of the FIFO, respectively. In Figure 7.8 one can see the basic structure of the modular FIFO with n stages. The sender interacts with the put interface (dashed area in Figure 7.8), which is further divided into cells (see detailed structure in Figure 7.10), one for each FIFO stage. To put data in the FIFO the data storage (at least one stage) has to be 'not full', which is determined by the Full-Empty controller. When the sender inputs data to the FIFO, the data is passed through the stages as a token to the last empty stage which is realized as a ring counter within the put interface cell of each stage. On the other side the FIFO is read out by the get interface (dashed area in Figure 7.8) that triggers the receiver with a request signal. Like in the put interface the get interface is also organized in cells (see detailed structure in Figure 7.10), the cell that is ready for the next read operation is marked with a token generated by a ring counter. The modular synchronizing FIFO employs the *asP** handshaking protocol [35]. Each FIFO stage employs a data latch, a SR-Latch as Full-Empty controller and the handshake logic (in the simplest way only an AND gate), see Figure 7.9. The scheme can be used for asynchronous (self-timed) modules with short interconnects. Due to the fact that *asP** is not delay-insensitive, only short interconnects can be used because the minimum pulse width that is required by the SR-Latch for a set and reset operation must be less than the minimum of the input-to-output delay of the latch. This requirement can easily be met in systems with short interconnects. For systems where long interconnects are employed a truly delay-insensitive protocol like LEDR is used. To use LEDR with the presented modular synchronizing FIFO an asynchronous protocol converter is required, one before the put interface and one after the get interface. Within the the *LEDR-to-asP** protocol converter each data line (2-wire) employs a completion detection (XOR gate), the received data is stored into a latch. This first stage converts the transmission into the micropipeline protocol [53]. Then the second stage transfers the transmission from the micropipeline protocol to the *asP** protocol. Note that one can use the second stage alone as a protocol converter to a micropipeline module. The *asP*-to-LEDR* converter employs as first stage an *asP*-to-micropipeline* converter and then a *micropipeline-to-LEDR* converter as final stage. Further the put and get interfaces can be employed in a clocked version. The put interface as well as the get interface employ a FIFO put/get control block that synchronizes the full/empty signal and write/read signal of the FIFO into the transmitter/receiver clock domain. The synchronizer depth can be chosen according to the requirements of the modules. The depth is freely chosen due to the fact that the FIFO controller is seperated from the synchronizer. As synchronizing elements flip-flops (full-cycle synchronization stage) or transparent latches (half-cycle synchronization stage) can be used and combined (see Figure 7.10). As well as the clockless FIFO the clocked version employs a Full-Empty controller and data latches to control operations and store data. The achievements of this approach are a high-throughput at $2 - 2.5$ Giga-transfers per seconds (at $90\ nm$ process) at a FIFO size of $\lceil 2 \cdot (n+1) \rceil$ stages, where *n* is the number of cycles that the synchronizer needs (its depth) and that both modules with synchronous and asynchronous timing can be used.

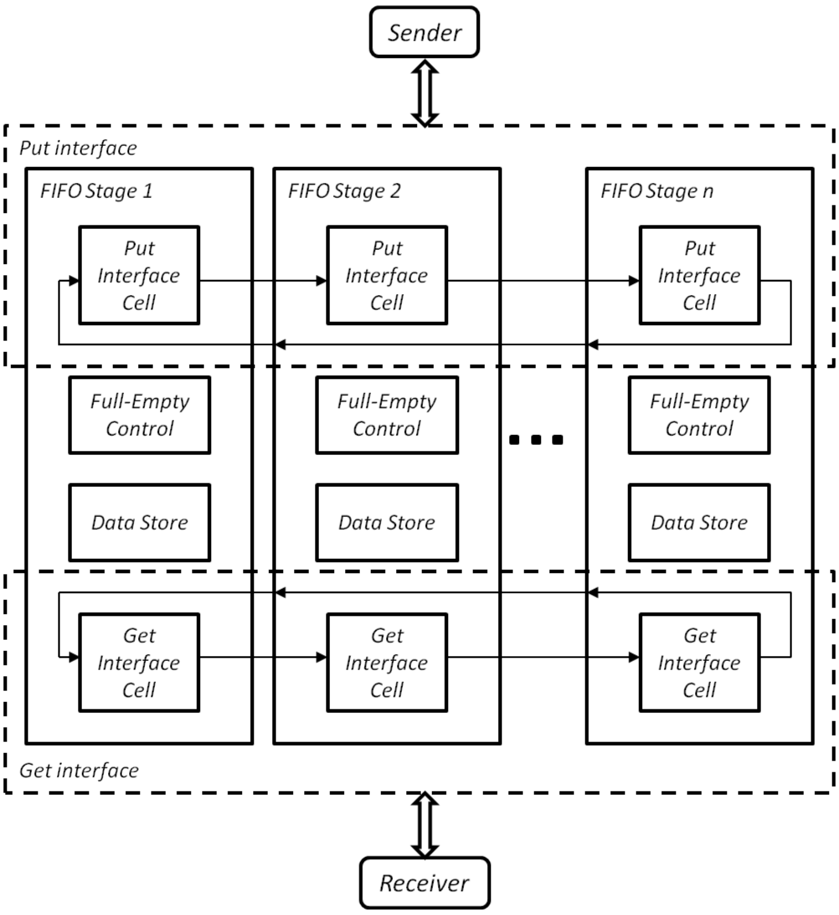


Figure 7.8: A Modular Synchronizing FIFO for NoCs [39]: FIFO Structure

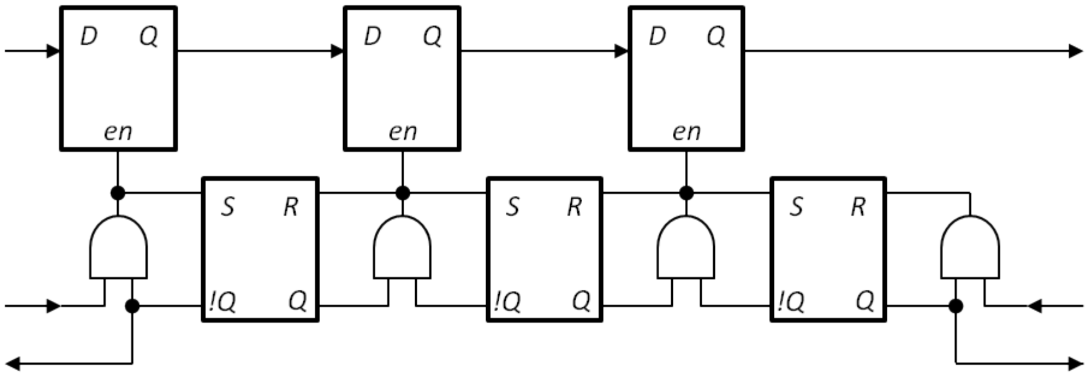


Figure 7.9: A Modular Synchronizing FIFO for NoCs [39]: asP* FIFO

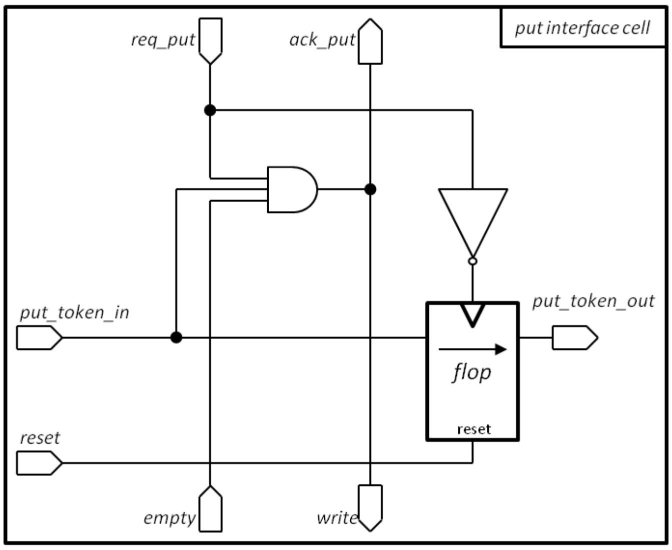


Figure 7.10: A Modular Synchronizing FIFO for NoCs [39]: *put* Interface Cell

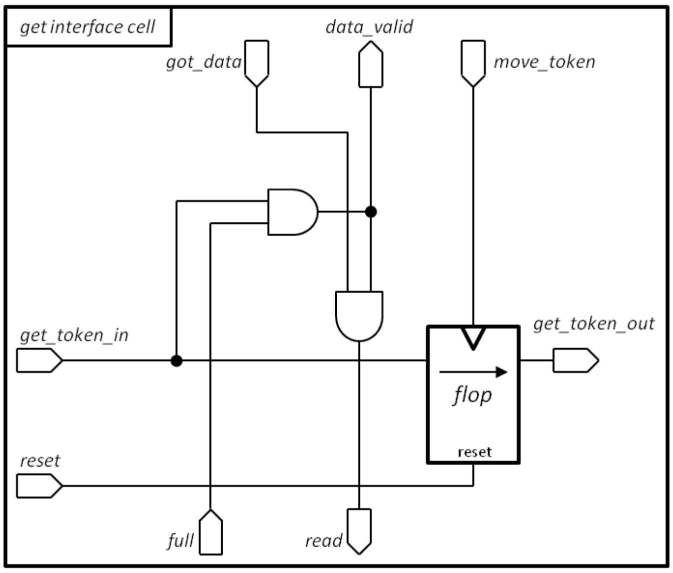


Figure 7.11: A Modular Synchronizing FIFO for NoCs [39]: *get* Interface Cell

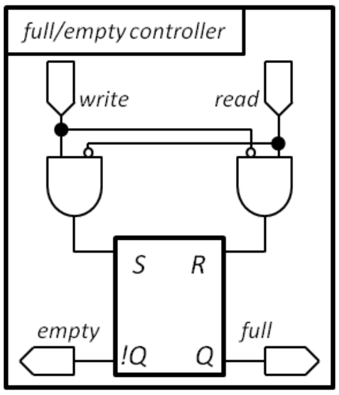


Figure 7.12: A Modular Synchronizing FIFO for NoCs [39]: Full/Empty Controller

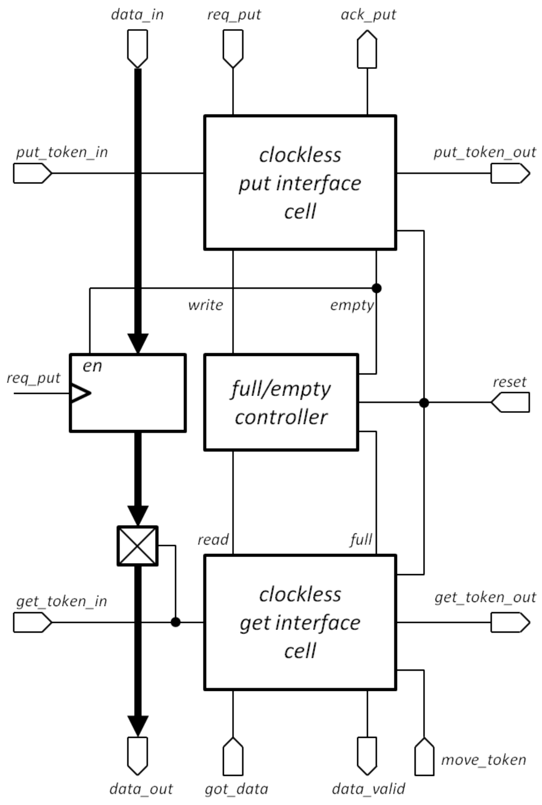


Figure 7.13: A Modular Synchronizing FIFO for NoCs [39]: FIFO Stage with asP* (asynchronous) *put* and *get* interface

7.2.6 Chain: A Delay-Insensitive Chip-Area Interconnect

Chain [2] is a network-on-a-chip approach that connects asynchronous and synchronous modules using a delay-insensitive encoding and a return-to-zero signaling technique. The link is employed as an one-of-five data encoding using five communication lines and one acknowledgement line, only one of these communication lines is allowed to send at a time, thus be at logic one. These five communication lines are further divided into one end-of-packet (EOP) signal and four data line. When one of these data lines is driven high a binary code is signaled which is interpreted as a two-bit data value, e.g. for a logical one data line 2 has to be asserted, for the two-bit data value '10'. The EOP signal seperates two consecutive data packets of variable length. This asynchronous interconnect uses pipeline latches for the self-timed latch stage for the one-hot links. Between consecutive pipeline latch stages a loop via Muller C-Elements is built up, which can be thought of as a ring oscillator, to latch data for a minimum oscillation period, that is determined by the Muller C-Elements, the OR gate, the inverter (acknowledgement) and the length of line between the stages. The pipeline latch (or *pipe latch*) consists of 5 Muller C-Elements, one for each data line. The outputs of the Muller C-Elements are forwarded either to the next stage or the receiver and are merged by a 5-input OR gate to form the acknowledgement for the predecessor stage. The acknowledgement forms the second input to each C-Element. An advantage of using return-to-zero signaling and one-hot coding is the minimized crosstalk. With *Chain* one can create networks of any topology. To achieve this several additional modules (circuits) are needed, like routers, arbiters and multiplexers. These can be built by adapting and extending the pipeline latch, as one can see in Figure 7.16, 7.17, 7.18. Note that the multiplexer (Figure 7.16) does not employ a *select* input, thus either the environment has to enable mutually exclusive communication or an arbiter is needed at its input (as seen in the collage in Figure 7.14), to ensure uninterrupted forwarding of data from one of the two inputs to the output. The figure shows two versions how a transmitter can be connected to a receiver, first through a router and second through an arbiter. The transmitter determines the routing by using its full knowledge of the network topology, further it spreads this information via encoded symbols at the start of each data package. Performance is limited by the path, because it is self-regulating in the asynchronous approach. By increasing the width of the data path, thus ganging links together increases the throughput (if 700 Mb/s of a single wire are too slow). In [2] five message types are defined to be used with *Chain*, one command message and four different responses. In *Chain* these two message types are transmitted via two seperately switched networks. Additional multimessage packets are provided to enable semaphore operations. In [2] a prototype is implemented using around 20 gates for each network fabric component and providing a maximum worst case performance (read memory via greatest network distance) of 20 million transfers per second. For further measurement data refer to [2].

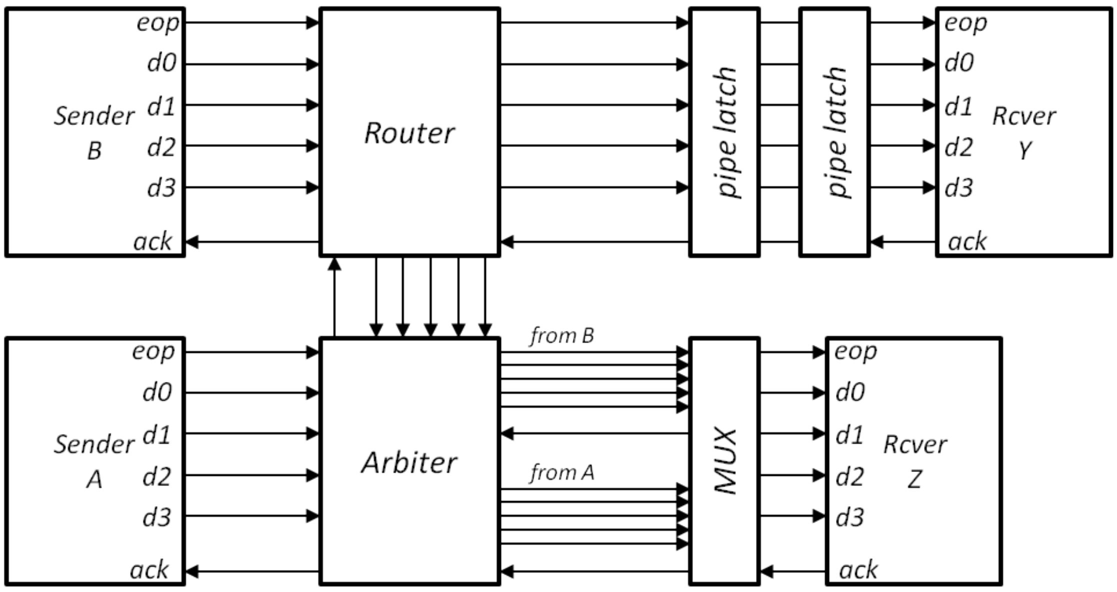


Figure 7.14: Chain - A DI chip area interconnect [2]: Collage

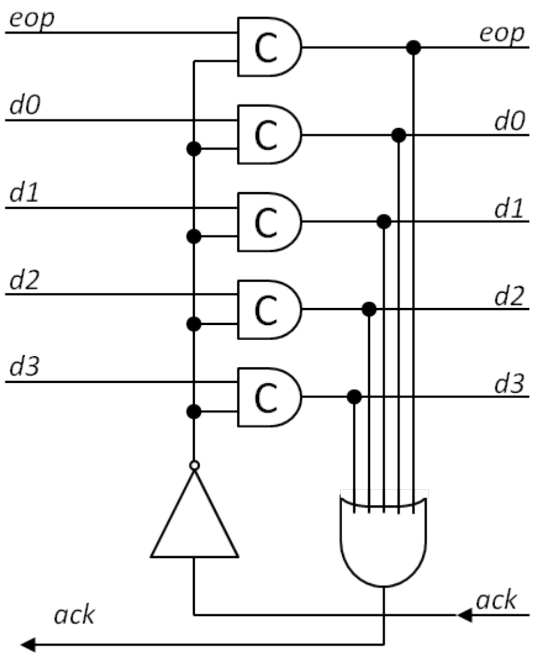


Figure 7.15: Chain - A DI chip area interconnect [2]: Pipe Latch

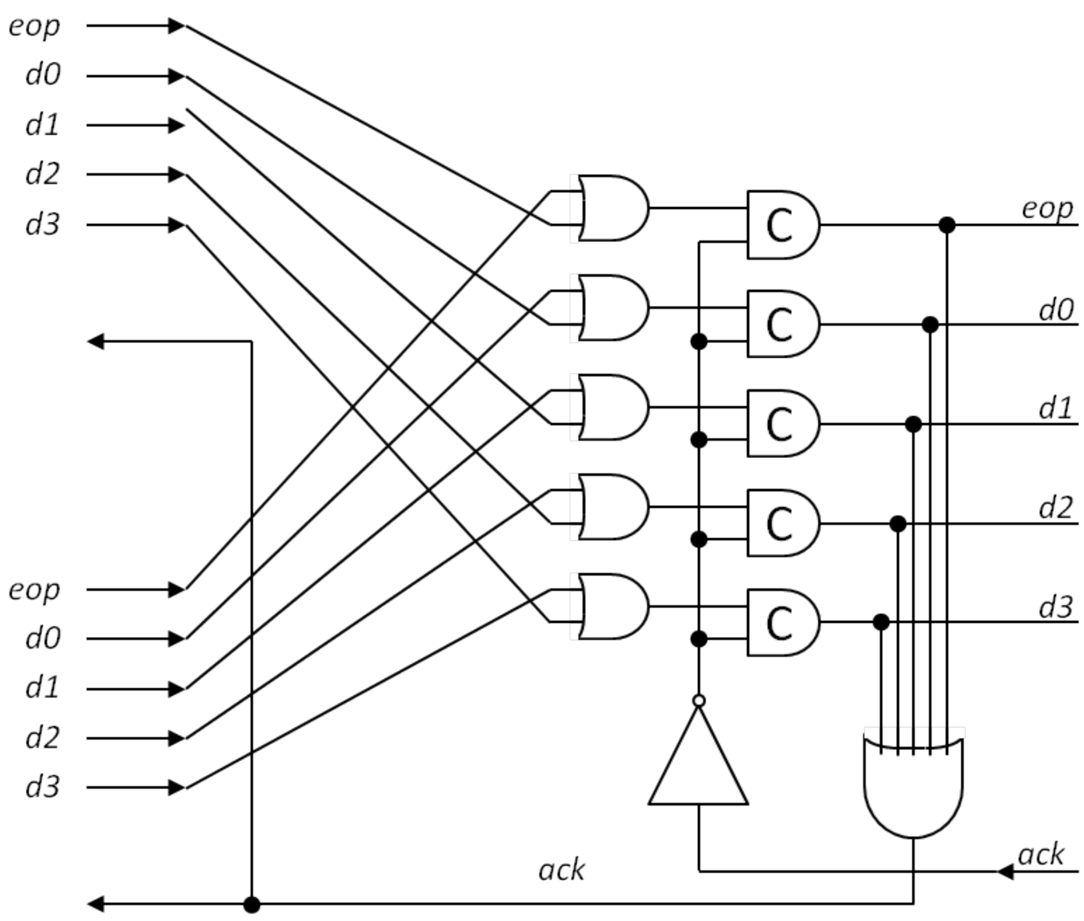


Figure 7.16: Chain - A DI chip area interconnect [2]: Multiplexer

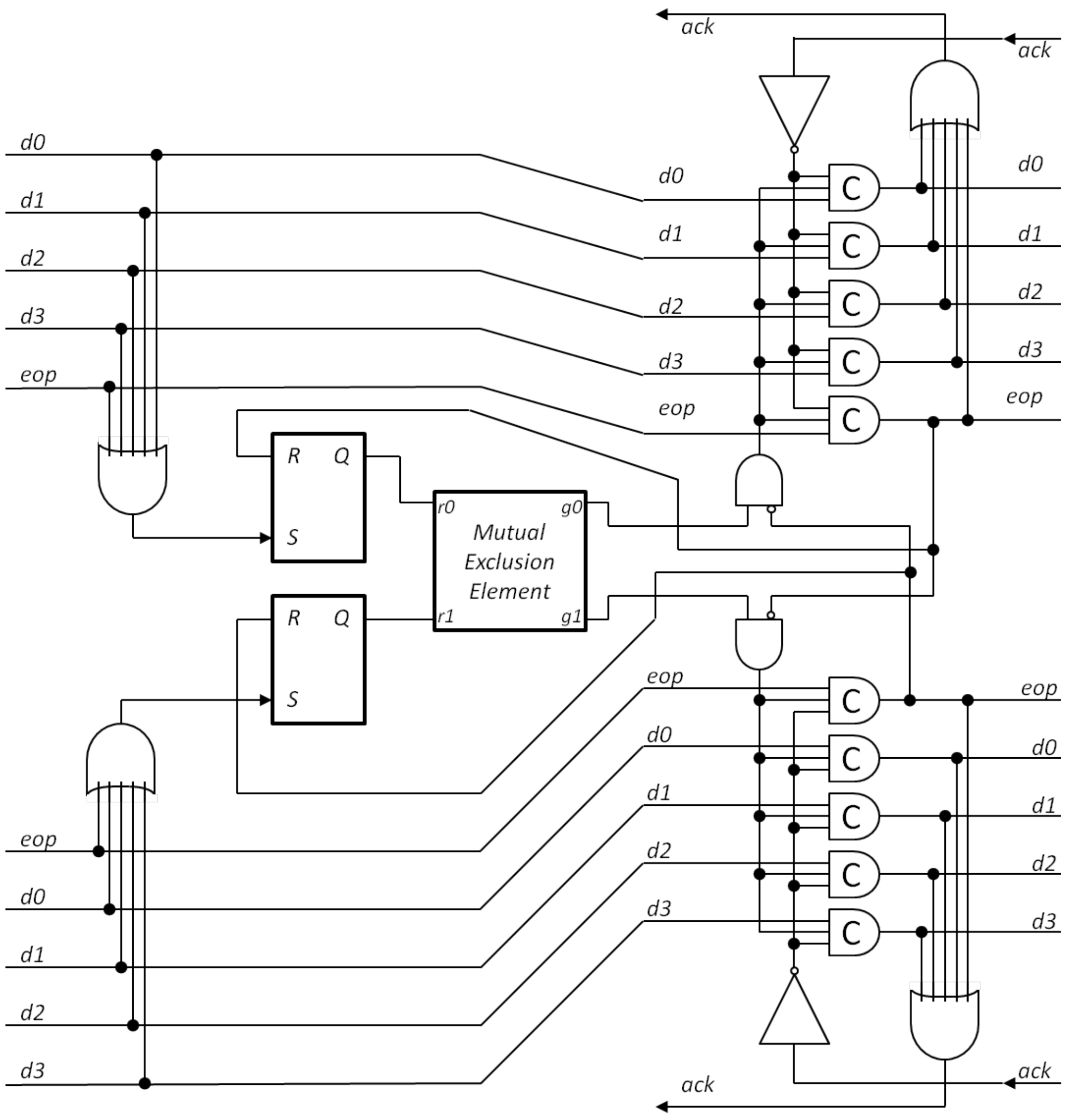


Figure 7.17: Chain - A DI chip area interconnect [2]: Arbiter

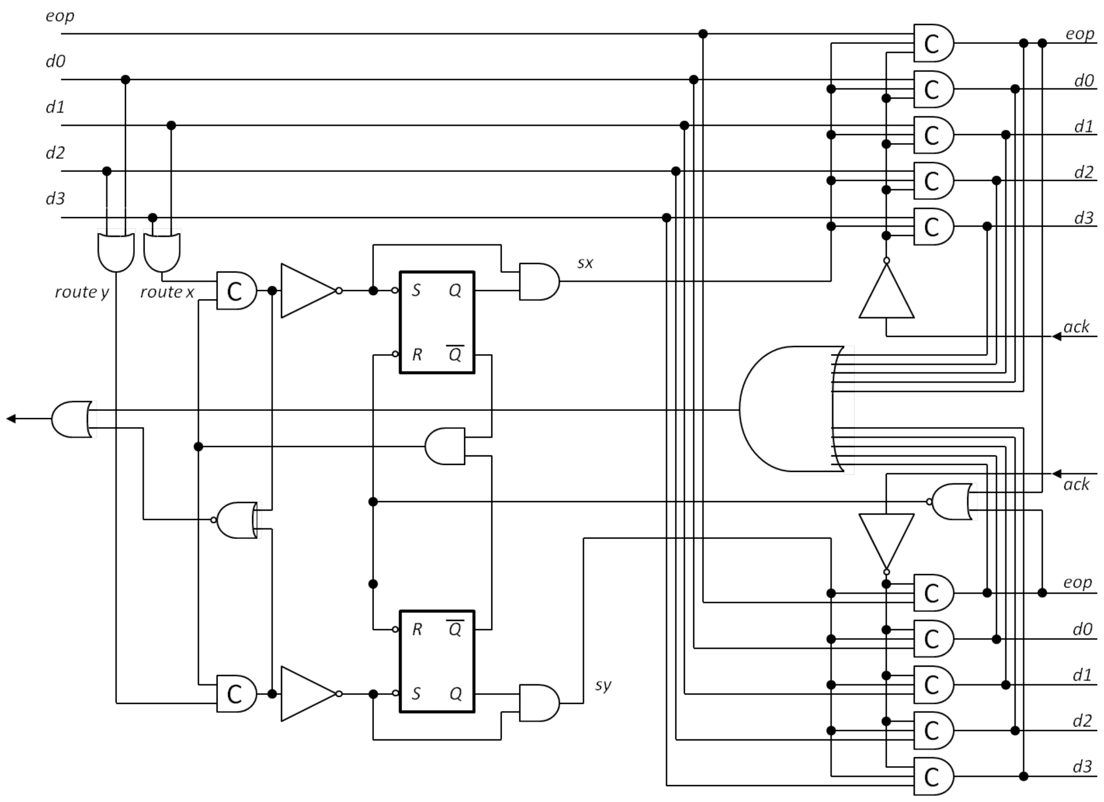


Figure 7.18: Chain - A DI chip area interconnect [2]: Router

7.2.7 GasP: A Minimal FIFO Control

The GasP approach [52] is an asynchronous control circuit for simple (handshake) pipelines. GasP is based on Molnar's *asP** control circuits, and is short for "Gigabit asynchronous symmetric persistent pulse control". The idea was to lower the reverse path latency as improvement to Molnar's *asP** (where forward and reverse path have the same latency), because it is easy and fast to move nothing (no data) backwards the pipeline. In GasP the forward path has a latency of four gate delays and the reverse path a latency of two gate delays, thus a cycle time of six gate delays (like a three-inverter ring oscillator). GasP employs two different circuit modules called PATH and PLACE, where the PATH is to perceive as a gate between the pipeline stages, the PLACEs. The PATH gates the data from its full predecessor stage to its empty successor stage. The PLACE or stage consists of a data latch (inverter loop) and a state conductor. The state conductor provides the state of the PLACE, either full or empty, to the surrounding PATH modules. The PATH module simply consists of a self-resetting NAND gate, whose inputs are both state conductor outputs from the predecessor and successor PLACEs. The output of the self-resetting NAND gate is provided to a pass transistor that "opens the gate" for data between two successive stages. The PLACE only gates data iff the predecessor stage is full and its successor stage is empty. Note that the state encoding is logical high for an empty stage and logical low for a full stage. In [52] several implementation approaches are shown for GasP. The first one, described previously, employs self-resetting NAND gates and a single state conductor. A further implementation employs two seperate state conductors, which is proven to be useful in [21]. In this approach the PLACE module provides both state conductors to each surrounding PATH, but the PATH module only monitors one of them. Additional a (unconditional) BRANCH module is presented where data of a full stage is provided to two other stages, both must be empty. A JOIN circuit is mentioned too. A data conditional circuit is described that can be used to prevent or permit driving the successor stage depending on the data input value. Furthermore GasP can be changed using another state encoding where logical high means full, which is preferabel for several reasons (refer to [52]), but has a slightly higher logical effort. The last presented implementation employs an additional mutual exclusion element for arbitration and a metastability guard (two inverters at the outputs) for it. The arbitration is used to stop the data flow without damaging the data. GasP is a fast approach, it provides a throughput of 1.5 Giga data items per second. To make it work it is crucial that the transistor widths are carefully balanced to match the delays.

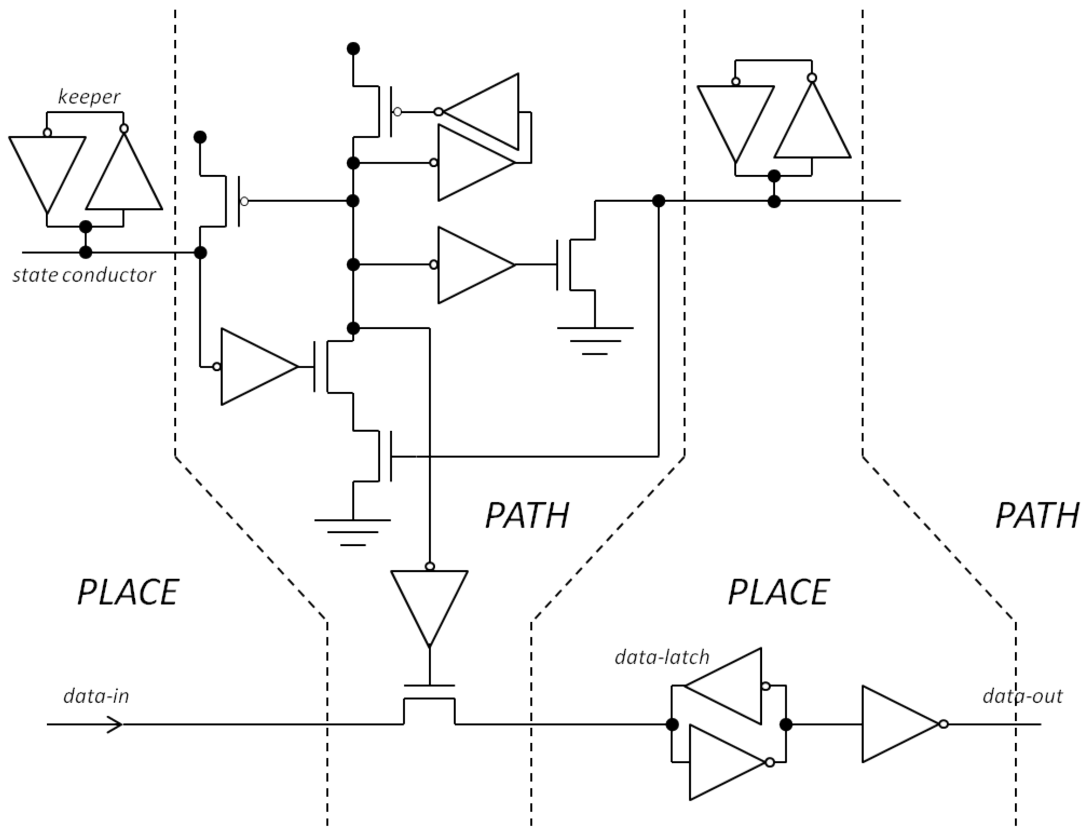


Figure 7.19: GasP - A Minimal FIFO Control [52]: GasP Self-Resetting NAND

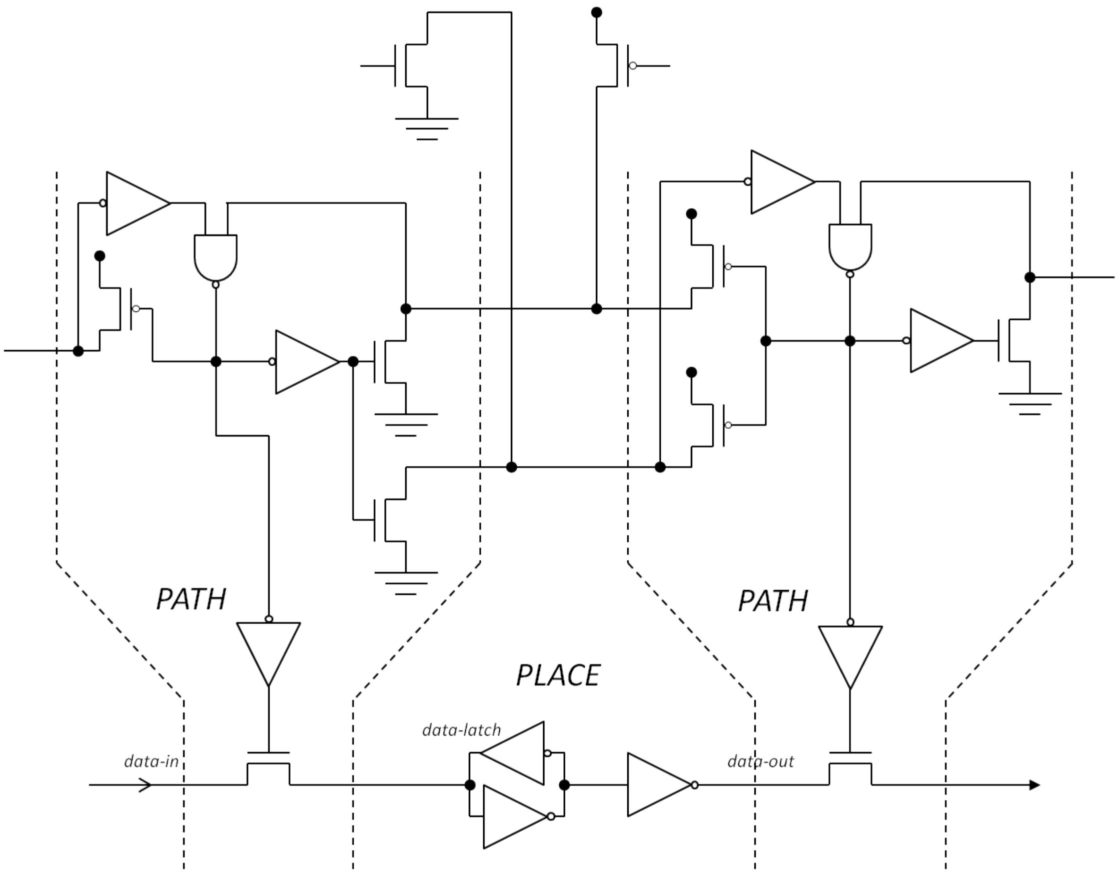


Figure 7.20: GasP - A Minimal FIFO Control [52]: GasP Twin State Conductors

7.2.8 A Doubly-Latched Asynchronous Pipeline

The DLAP (double-latched asynchronous pipeline) [34] is a single-rail, 4-phase protocol approach for an asynchronous pipeline that employs two registers per stage in a master-slave hierarchy. The registers are either edge-triggered or transparent latches. The DLAP operates similar to a synchronous pipeline by imitating a synchronous master-slave pipeline and can also be used for synchronous-to-asynchronous conversion. Due to the employed dual registers DLAP operates truly decoupled thus data is shifted simultaneously through all stages. This is done by alternately latching data to master register and slave register. While the master register latches new data from a predecessor stage (from its slave register), the slave register retains the previous data and provides it to the successor stage (to its master register). The communication between the stages is achieved by a stage controller and ready and acknowledgement signals. The stage controller has different implementations, depending on which type of register is used, either edge-triggered or transparent latches. First the stage controller for edge-triggered register is realized by three Muller C-Elements. One merges the done signals of either the master and slave register, whose output further is used by either of the two Muller C-Elements to alternately activate either the master or slave register in combination with the ready signal of the predecessor stage or the acknowledgement signal of the successor stage, respectively (see Figure 7.21). The advantage of edge-triggered registers is the low response time compared to transparent latches. On the other hand transparent latches are simpler than edge-triggered registers. The transparent latches are kept closed when no data must be latched. Thus the master and slave register can not be open at the same time, thus the controller has to be more complex than the one for the edge-triggered DLAP. There has to be an extra signal within the controller that signals which of the registers has been opened last. The controller for transparent latches employs six Muller C-Elements and several logic gates to achieve a mutual exclusion element for the activation signals for the latches. For further details of the circuit see [34]. Further non-linear DLAP data path can be realized by using *fork* and *join* interconnection modules. A *fork* splits the ready output signal and merges the incoming acknowledgement tokens. A *join* module merges two input stages by merging their ready input signals. Additionally with these modules a DLAP ring can be constructed. (Note that a ring must contain an extra register to prevent the possibility of a deadlock.) In summary, the throughput of the DLAP widely depends on the width of data lines and the register size, a SPICE simulation indicates that a full handshake needs about $24.7\ ns$ (see [34]), which results in a throughput of 40.5 million data items per second.

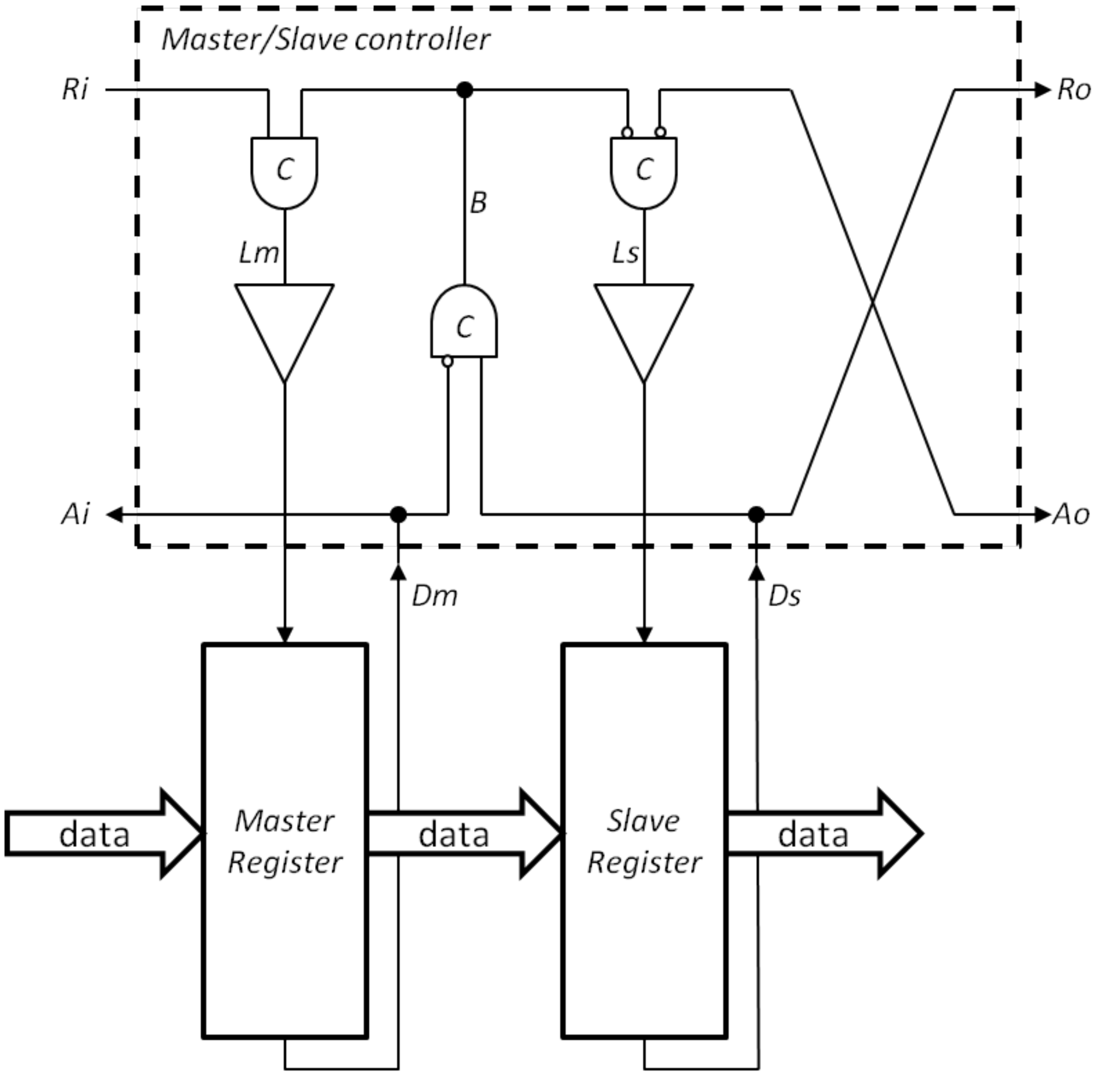


Figure 7.21: A Doubly-Latched Asynchronous Pipeline [34]: DLAP Stage Structure

CHAPTER 8

Interconnects

In this section the focus lies on the connection between modules rather than their interfaces. *Interconnects* are tailored to particular needs, e.g. to enable long-wire communication, build up networks or connect building blocks within FPGAs. The presented interconnects in this chapter are slightly off the focus of this thesis and therefore not further compared to each other, but they are, however, worth to mention.

8.1 Detailed Descriptions

8.1.1 Surfing Interconnect

The *Surfing Interconnect* [30] is an approach for long-wire signaling by dividing long wires into several segments connected by so-called soft-latches or surfing-latches as buffers. This interconnect pipelining is used to regulate the data transfer along long-wires between either asynchronous and synchronous modules. A surfing link can be seen in Figure 8.1. The surfing link consists of two lines, the timing chain and the data path. The timing chain further consists of a repeater, a simple inverter and a 'edge-to-pulse' converter. The 'edge-to-pulse' converter generates a timing pulse called FAST at every logic segment of the pipeline that controls the soft-latch, more precisely its delay, on the data path. The FAST signal when asserted decreases the delay within the soft latch. Hence the soft-latch consists of an ordinary inverter and a tristate inverter, when FAST is asserted the tristate inverter provides extra drive capability and thus reduces the delay. On the other hand when FAST is deasserted only the ordinary inverter drives the output and thus the delay increases. With this mechanism two timing constraints hold, that neither the data cannot keep up with the timing events in FAST mode nor that the data move faster than the timing events in slow mode. The timing path itself does not employ a handshaking mechanism (not needed for synchronous communication partners), thus an interface circuit is needed at the receiver. So all data items can be kept in close relationship to their related timing event. Thus surfing interconnect is achieved by changing the delay of the stages or varying the output driver of the repeater and its strength. To enable proper communication using the *Surfing Interconnect* receivers need a FIFO for buffering. However, if the communication partners are running different clock frequencies an acknowledgment signal back to the transmitter is required. A possible implementation of a 2-phase handshaking surfing interconnect is shown Figure 8.2. In this case the size of the buffering FIFO has to be large enough to compensate a handshake round-trip delay back to the sender.

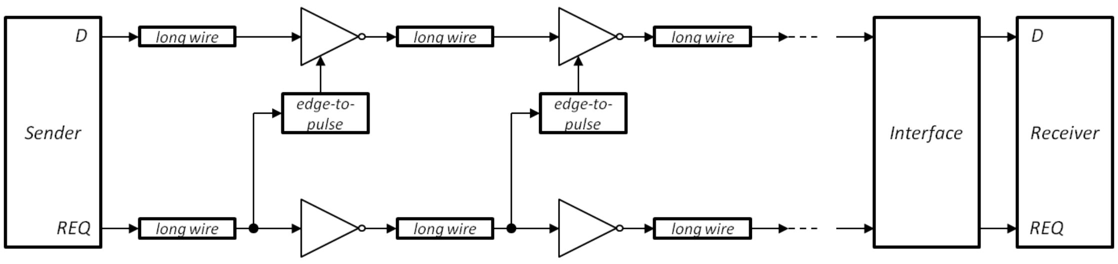


Figure 8.1: Surfing interconnect [30]: A Surfing Link

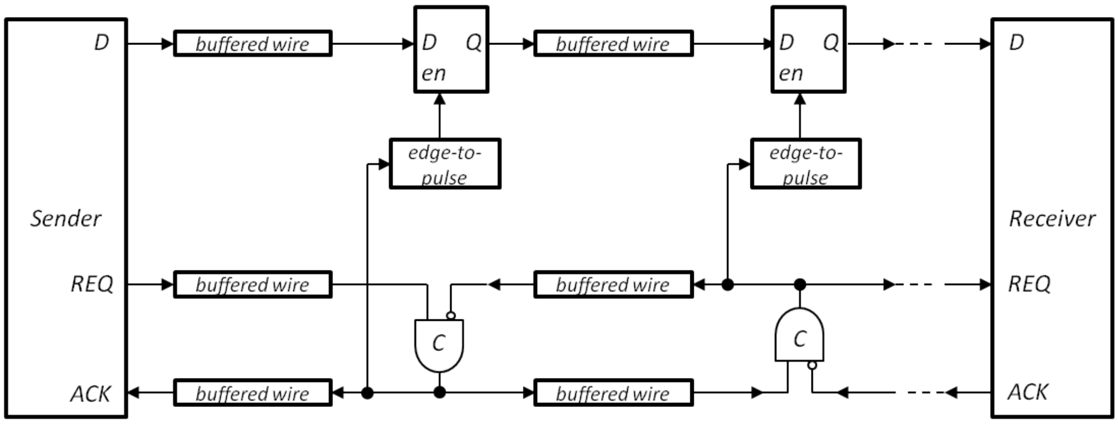


Figure 8.2: Surfing interconnect [30]: 2-phase Handshaking Surfing Link

8.1.2 Practical Asynchronous Interconnect Network Design

An asynchronous interconnect design is presented in [42] that uses a 2-phase handshaking technique and dual-rail encoding. The intent of the authors was to create a solution to interconnect a large number of IP (intellectual property) blocks with different clock rates to a single PLC (programmable logic core) in order to provide the possibility of a post-silicon debug mechanism. As one can see in Figure 8.3, the asynchronous interconnect approach is organized in stages. Each stage employs two flip-flops, one for each rail. Further a clock generation circuit is employed that generates the clock-like signal upon the incoming data on the dual-rail line. More specifically the clock generator generates a rising edge on the output if new incoming data is different in encoding to the current one or the current encoding of the data is equal to the one of the following stage. Note that each stage returns its current encoding of data to its predecessor stage as acknowledgement signal. On the other hand a falling edge is produced when the flip-flops have latched new data and thus generate a new current encoding of data. The generated clock is used to control or activate the data flip-flops. The delays of the XOR gates in the stage and further the clock generator have the most impact on the latency and cycle time of the design (approximately $8.75\ ns$ at a die width of $12090\ \mu m$). This solution can be used by any module with 2-phase encoding, it also can easily be adapted to fit 4-phase encoding.

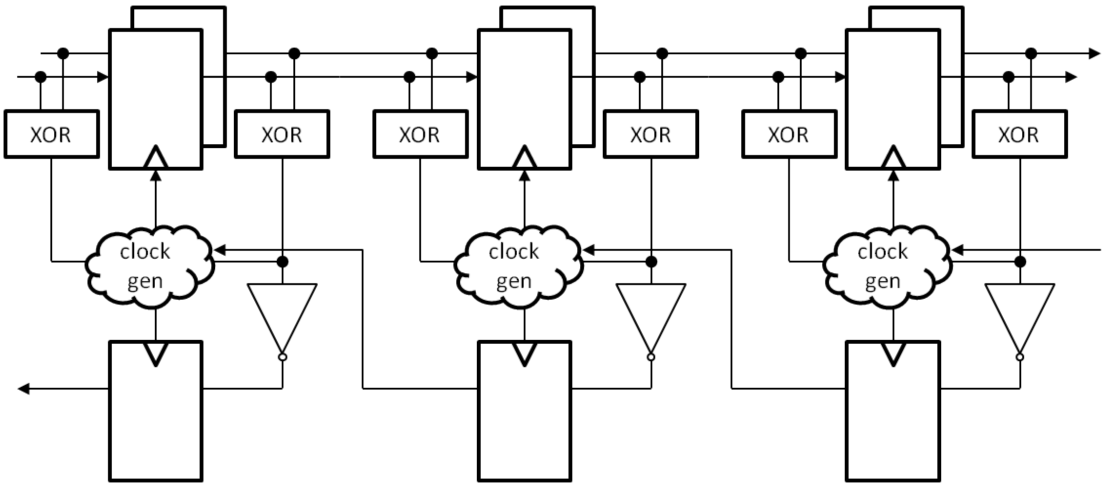


Figure 8.3: Practical Asynchronous Interconnect Network [42]: Dual Rail, Two-Phase Implementation

8.1.3 Asynchronous Current Mode Serial Communication

An approach for on-chip long-wire interconnects is presented in [19]. The approach uses an asynchronous wave-pipelined serial link. The link is realized as dual-rail and uses the LEDR 2-phase protocol in order to avoid explicit handshaking and synchronizing of each bit. Typically serial links are used for off-chip communication due to pin-out limitations, but they achieve high throughput at long distances (about 7 mm in a 65 nm-process chip) within a chip and need less area overhead. To be independent from the voltage swing of long-wire interconnects genuine current-mode sense-amplifiers are used for communication, and further they enable a data cycle of a single FO-4 (fan out of 4) gate delay. In Figure 8.4 one can see the serial communication link, it employs two synchronizers for either communication partner, a serializer and LEDR encoder for the sender, and a de-serializer and LEDR decoder for the receiver. The receiver acknowledges every transmitted word. The LEDR encoding and decoding is done on-the-fly and uses only an XOR and some transmission gates. The serializer and de-serializer employ fast shift-registers. Each wave-pipeline shift-register consists either of a split stage (at the receiver) or a merge stage (at the transmitter), to either split or merge the two seperate data paths, respectively. These two seperate data lines are passed from/to the transition latches (see Figure 8.5). These transition latches consists of an inverter and a weak keeper for each data line and are controlled by a differential signal from the split stage (receiver) or the transmitter directly. The keeper is off when a data bit is shifted. Data is shifted and sampled at rising as well as falling edge of the differential control signal. For the technique of current mode asynchronous signaling several differential current-mode drivers are available, in the presented approach both differential wires pull currents of two slightly different values (i.e. $I_1 > I_2 > 0$ or $I_2 > I_1 > 0$). By this, the current swing is minimized in the solution of [19]. The benefit of current sensing is to reduce the voltage swing that comes with long-wire interconnects. The presented approach was simulated in [19] and achieves a throughput of 67 Gb/s over a 7 mm distance for 65 nm-technology.

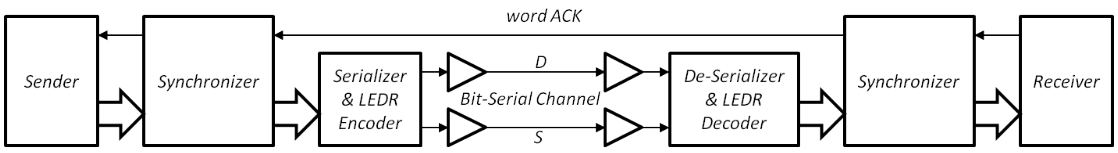


Figure 8.4: Asynchronous Current Mode Serial Communication [19]: Serial Communication Link

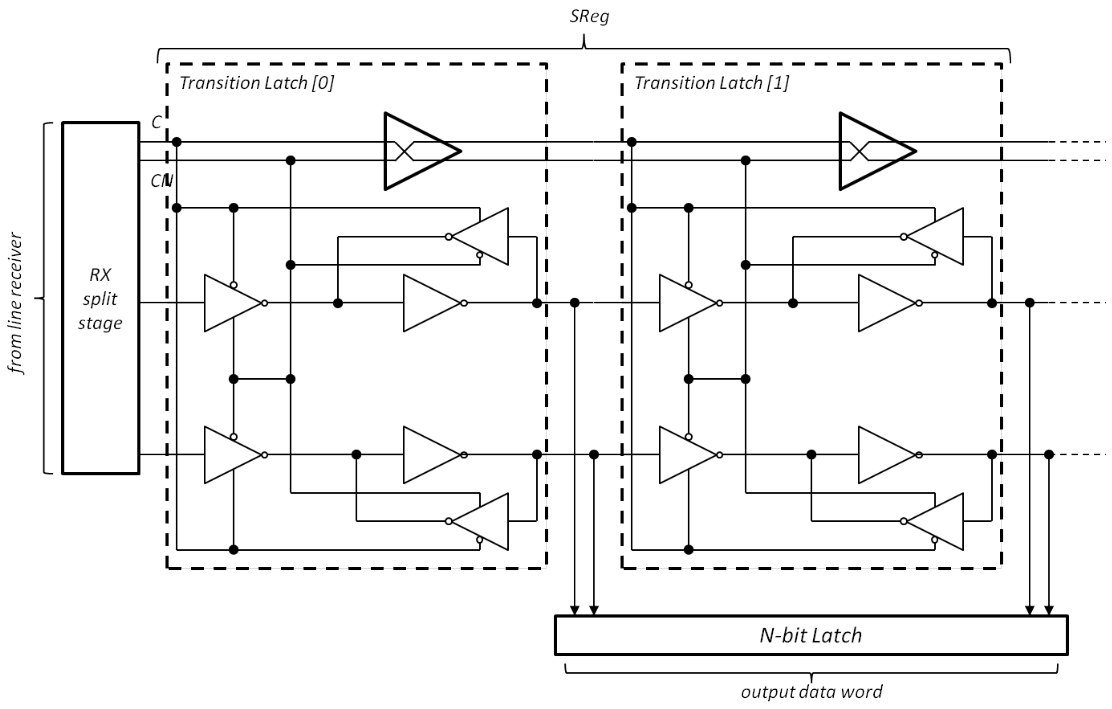


Figure 8.5: Asynchronous Current Mode Serial Communication [19]: One FO4 Gate Delay Shift Register (De-Serializer

8.1.4 Asynchronous FPGA Architecture with Distributed Control

An approach for an asynchronous delay-insensitive FPGA module interconnect is presented in [48]. To keep the existing FPGA function blocks a wrapper is developed based on David's Cells to implement distributed control logic and provide asynchronous routing to the system. A David's Cell forms a distributed control circuit and is shown in Figure 8.6 in its simplest form. It basically consists of three NAND gates, a backward line and a forward line, and a set and a reset signal. The David's Cell is set up when the set signal goes low and further the backward line is de-asserted. These signals form a handshake protocol for the sender. A control token is forwarded to the next stage by setting the forward line to low, after the set signal is high again. The forward line together with the reset signal form the handshake protocol for the following stage, the receiver (the reset signal has to go low to initiate the handshake). In [48] a more complex example is shown as well as a block diagram of the general definition of the David's Cell. This mechanism is used to guide data items from one configurable logic block to the next. As one can see in Figure 8.7 data items are delivered via dual-rail data lines towards a logic block. A completion detection logic block detects the complete arrival of the data and triggers the David's Cell. The David's Cell generates a signal that initiates the computation of the newly arrived data item in the computational logic block. This signal is forwarded, through a programmable delay, to the single-to-dual-rail converter (the FPGA module works with single-rail signaling). After the conversion is completed the handshake is done (handshake signal), which signals the predecessor stage to provide new data.

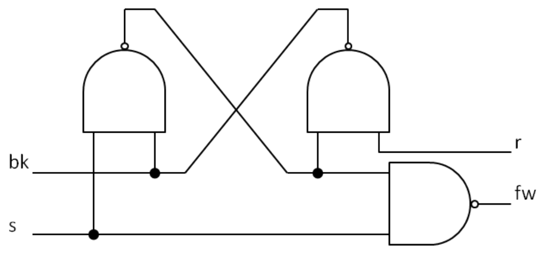


Figure 8.6: Asynchronous FPGA Architecture with Distributed Control [48]: Simple David Cell

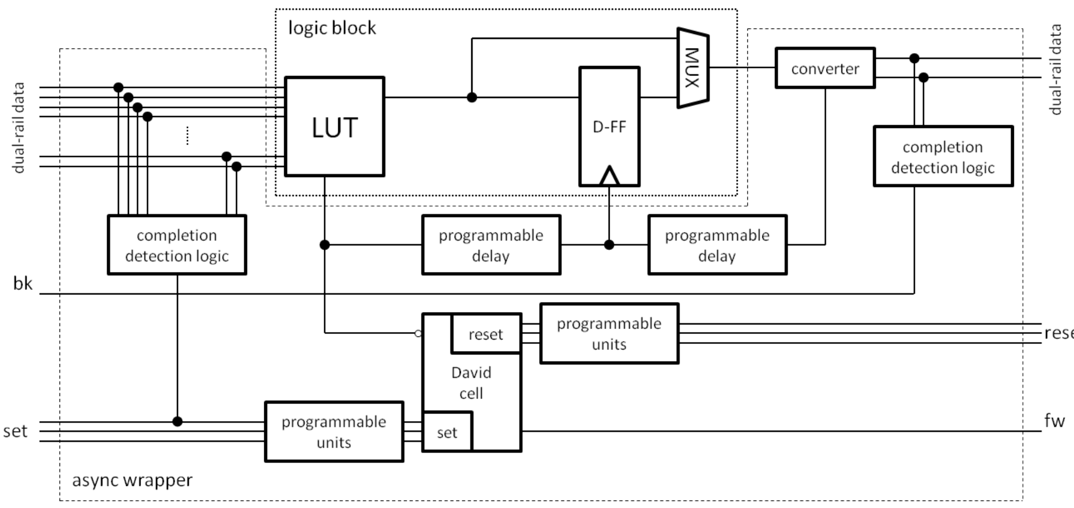


Figure 8.7: Asynchronous FPGA Architecture with Distributed Control [48]: Wrapper Structure

8.1.5 FLEETzero: An Asynchronous Switching Experiment

FLEETzero [11] is a chip developed for testing an asynchronous switch-fabric based on GasP control circuits [52]. The chip tests the transport of data item from eight different sources (inputs) to eight different destinations (outputs) where every source can address every destination. The sources/destinations are called ships (modules). This is done by special data-controlled branch and merge circuits. FLEETzero is a so-called "transport-triggered architecture" where the focus of chip design is moved from 'operation-centric' to 'communication-centric' to improve system performance in communication. In a 'communication-centric' system data transmission is done by "move" instructions, that are executed in sequential order and include their source and destination address for routing. The used GasP circuits provide asynchronous data-controlled branching and merging modules that offer data every six gate-delays, that results a throughput of 1.2 $Giga\ data\ items\ per\ seconds$ for the FLEETzero chip. The merge switch-fabric called *funnel* and the branch switch-fabric called *horn* are used to build up a network between source and destination to ensure that every source can reach every destination (for data transfer). One can see the switch-fabrics in Figure 8.8. The function of the basic branch switch-fabric is similar to a single multiplexer with two ouputs. The modified branch switch-fabric multiplexes the data like the basic branch switch-fabric with the difference that the control signal is included in the data signal on the input of the branch and seperated afterwards for a distinct order signal for the switching control. This order output line is typically forwarded to a merge switch-fabric as control signal. The merge switch-fabric on the other side takes two input data lines and an order signal, latter is used as control signal for the merge process and is merged with the data at the output. The merge switch-fabric is similar to a demultiplexer with two inputs. A merge component can also be employed with an arbiter at the data input, so the first data item that arrives will activate the merge operation instead of the order input. This merge process activated on-demand entails a first-come-first-serve order on data transfer, this implies a non-deterministic order. When building a network with these modules a bottleneck is generated at the point where the funnel meets the horn, this is called the *trunk*. A network with four sources and destinations and the trunk is shown in Figure 8.9. The network is a balanced binary tree built up with asynchronous pipeline components. Note that between each branch and merge component data and order FIFOs are employed. The merge component reconstructs the data order from the information placed in the order FIFO. The performance and throughput of the network can be improved by employing multiple trunks between source and destination. The implementation details and simulation results of FLEETzero experiment are omitted in this thesis because we are only interested in the used interface (asynchronous pipeline network) but not in the resulting test chip.

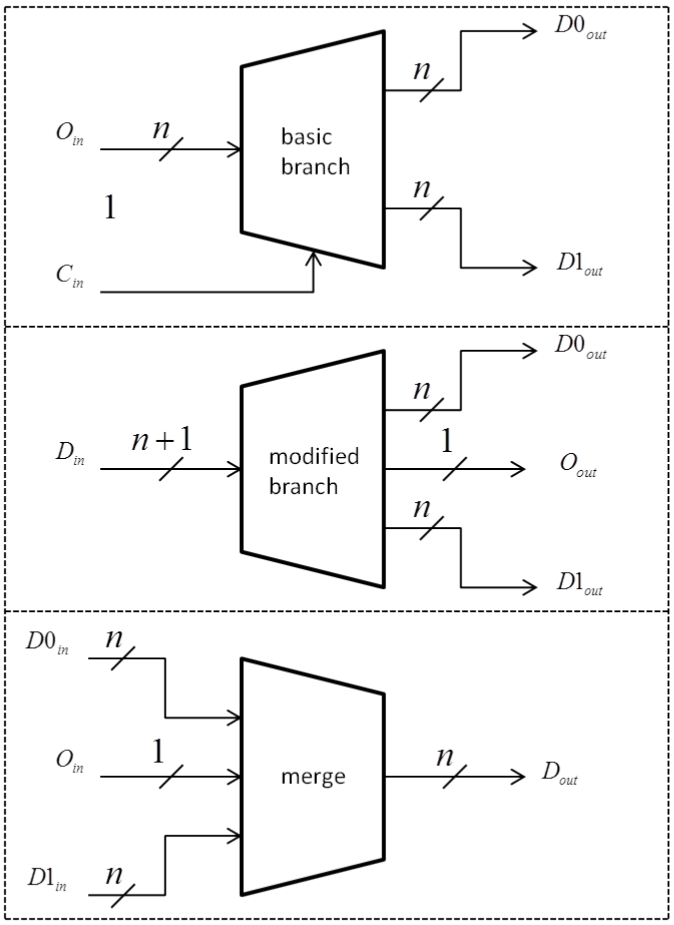


Figure 8.8: FLEETzero [11]: Basic Switching Primitives

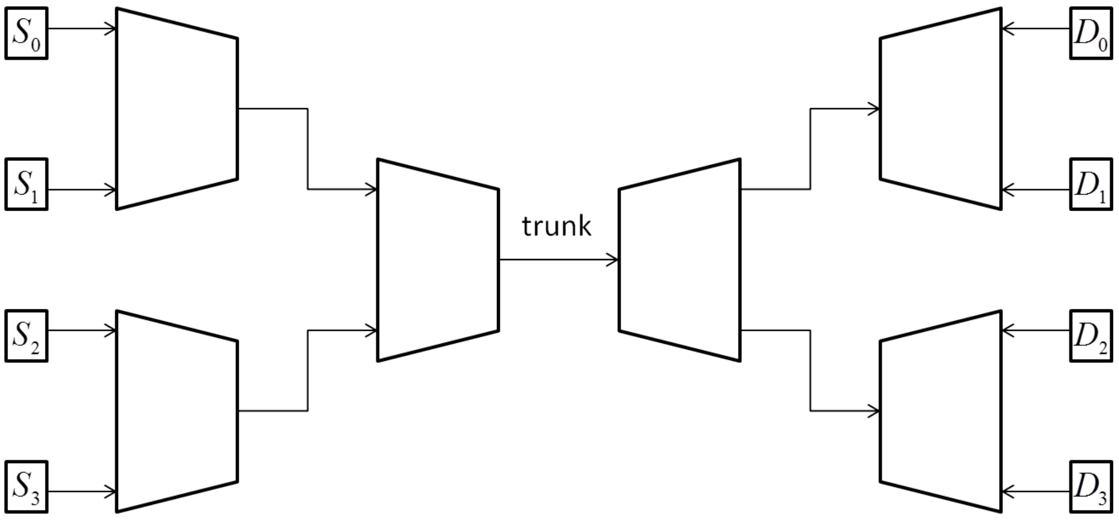


Figure 8.9: FLEETzero [11]: Horn-and-Funnel Network

CHAPTER 9

Quick Reference Guide

9.1 Tables of Categories

The following five tables list the over sixty different synchronizers and interconnect approaches that have been found and classified into five different categories (thus a table for each category, see Chapter 2.1).

9.1.1 Table of Mesochronous Synchronizers

Mesochronous Synchronizer			
Name		**Chapter**	**Reference**
Brute-Force Synchronizer		3.2.1	[12]
Description	Directly synchronizing an incoming data signal from sender by latching it with a Flip-Flop, which is clocked by receiver clock. A second FF, that is also clocked by receiver clock, is set in row to ensure that a possible metastable state of the first FF decay until the receiver reads the new data item. (Not recommended)		
Delay-Line Synchronizer		3.2.2	[12]
Description	Similar to the *Brute Force Synchronizer*, but with a variable delay on the data lines, needed for each bit (line). Employs a learning phase during which the FSM determines how to adjust the digital delay.		

Mesochronous Synchronizer - continued

Name		Chapter	Reference
Two-Register Synchronizer		3.2.3	[12]
Description	Acts like a Delay-Line Synchronizer, but the delay element is inserted on clock-lines of the registers, only one delay element required for multi-bit data paths.		
Two/Three Element FIFO Synchronizer, Mesochronous Synchronizer		3.2.4	[12] [28]
Description	Small ring-buffer to decouple transmitter and receiver timing, new data is alternately sampled into a couple (2 *or* 3) of flip-flops to rest for a round. The oldest value is chosen by the read pointer and transmitted to the receiver.		
Adaptive Synchronization		3.2.5	[26]
Description	Used to sychronize modules that operate with a clock that is distributed through the system with an unbalanced clock tree. A conflict detector unit checks the *ready* signal and the (reshaped) local clock, depending on the detected conflicts a digital delay line is configured with less or more stages to appropriately align the data to the clock.		
Low-Latency and Low-Overhead Mesochronous Synchronizer		4.2.2	[7]
Description	Receiver interface samples incoming data either on the falling or rising clock edge, depending on the *strobe* signal sent by the transmitter. The *strobe* signal toggles with the data lines and is delayed and synchronized at the receiver, and further used to control input latches. Metastabilities may arise only during learning phase (only once after reset), when sampling *strobe* signal and thus off the data path.		
Four-Stage Mesochronous Synchronizer		3.2.7	[17]
Description	Employs four stages (ring buffer) at the transmitter for data transmission, which are alternately written to provide resolution time and new data to the receiver every cycle. The receiver either directly latches data from the transmitter's stages or buffers it to a FIFO (data burst). A token mechanism is used to indicate validity of data items in one direction and for backpessure in the other direction. Also can be extended to support transmission between *multi-synchronous* modules.		

9.1.2 Table of Plesiochronous Synchronizers

Plesiochronous/Periodic Synchronizer

Name		**Chapter**	**Reference**
Plesiochronous FIFO Synchronizer		5.2.1	[12]
Description	Extended *Two Element FIFO Synchronizer*, a resynchronization signal switches between pointer of transmitter and pointer of receiver if the phase drifts over an entire cycle.		
Periodic Asynchronous Synchronizer		5.2.2	[12]
Description	Clock prediction mechanism; generates a keep-out signal that determines if the transition of the predicted clock cycles lies in the keep-out region of the local clock and controls the multiplexer of a two-register synchronizer.		
Adaptive Synchronization		3.2.5	[26]
Description	Basically a mesochronous synchronizer, but in conjunction with a certain learning phase (“continuous tracking” mode) it is possible to use it as plesiochronous synchronizer too.		
Low Latency Plesiochronous Data Retiming		5.2.3	[15]
Description	A synchronizer-avoidance method for mesochronous and plesiochronous systems. Data items are sent one-by-one in so-called *cells*. To regulate the data rate, cells can also contain non-data items (the receiver destinguishes). In a plesiochronous system data is retimed (delayed) by a variable amount of time (to cope with the variable phase drifts), if the receiver would sample data during the exclusion region.		
Low-Latency and Low-Overhead Plesiochronous Synchronizer		5.2.4	[7]
Description	Based on Low-Latency and Low-Overhead Mesochronous Synchronizer 3.2.6. Employs a *continuous learning phase* (samples *strobe* signal not only once after reset). Cope with plesiochronous systems, modules with same clock frequency but slowly drifting phase shift.		
Four-Stage Mesochronous Synchronizer		3.2.7	[17]
Description	The Four-Stage Mesochronous Synchronizer (see Table 9.1.1) can be extended (adding more stages to the transmission buffer) to support *plesiochronous* (*multi-synchronous*) modules.		

9.1.3 Table of Synchronizers for Systems with Uncorrelated Clocks

Synchronizer for systems with uncorrelated clocks			
Name		**Chapter**	**Reference**
Two-Flip-Flop Synchronizer, Waiting Synchronizer		3.2.1 6.2.1	[12] [27] [28]
Description	It has the same structure as the Brute-Force Synchronizer, but synchronizes handshake signals instead of directly modifying the data stream. Hence requests from the transmitter are synchronized by a cascade of flip-flops. Provides a MTBF of 10^{204} years at a data rate of 200 MHz.		
Conservative Synchronizer			[27]
Description	Extends the flip-flop cascade of the Two-Flip-Flop Synchronizer by a third flip-flop to enhance resolution time and increase the MTBF.		
Asynchronous FIFO Synchronizer, Two-clock FIFO Synchronizer		6.2.17	[12] [28]
Description	Synchronization is performed on transmit/receive pointers (used to detect full/empty conditions of FIFO). New data is shifted into FIFO from the transmitter clock domain and pulled out of FIFO into the receiver clock domain. Flow-control is enabled via full and empty signals.		
Stoppable Clocks Module		6.2.7	[12]
Description	Local clock is only generated on incoming events (requests). The receiver employs an asynchronous interface, but works internally synchronous. The event-driven clock eliminates synchronization delay and the associated probability of synchronization failure.		
Asynchronous Interlocked Pipelined CMOS Circuits		6.2.8	[44]
Description	Asynchronous data-driven clocking technique. Provides a strobe circuit that generates a clock signal upon data valid signals. Can be used at 4.5 GHz in best case conditions (3.3 GHz in average case).		

Synchronizer for systems with uncorrelated clocks

Name		Chapter	Reference
Asynchronous Wrapper for Heterogeneous Systems		6.2.9	[4]
Description	Provides asynchronous input and output interfaces for (locally) synchronous modules of a GALS system. The wrapper employs a 4-phase handshake (performed within one clock period by an AFSM) and stretches the clock of the wrapped module.		
A Robust Synchronizer		6.2.2	[59]
Description	An extension of a Jamb-Latch Synchronizer (basically a Two-Flip-Flop Synchronizer using Jamb-latches), to reduce the supply voltage and increase the metastability resolution time.		
Pausible Clocking		6.2.10	[58]
Description	A FIFO-based synchronizer that uses pausible clocking technique and a 2-phase handshaking protocol to interface differently clocked modules. Each module employs a PCC (pausible clocking control) unit to control the access to the asynchronous FIFO.		
A Synchronizer Design Based on Wagging		6.2.15	[1]
Description	Employs an extended version of a dual-edge triggered flip-flop (DETFF). The DETFF employs three paths (each including an inverter loop), which alternately sample the input and output the sampled data according to another clock with a different phase than the sampling clock.		
SCAFFI: Asynchronous Interface		6.2.11	[41]
Description	*Stretchable Clock Asynchronous Flexible FPGA Interface*, SCAFFI, is an asynchronous interconnect, that connects mutually asynchronous modules and uses a pausible clocking mechanism. Is capable of stretching the clock using both logical levels. Provides a 4-phase handshake and bundled data mechanism between communication partners. Between SCAFFI and the module (receiver, transmitter) a 2-phase handshaking mechanism is used. A dual-rail version of SCAFFI is provided for long-wire interconnects.		

Synchronizer for systems with uncorrelated clocks

Name		Chapter	Reference
Register-Communication		6.2.3	[32]
Description	Introduces a communication register that supports non-blocking data transfer (write and read accesses are never paused) between mutually asynchronous domains (clocked or self-timed). The four different types of communication register are used as wrapper circuits for modules.		
High Rate Data Synchronization in GALS SoCs		6.2.6	[18]
Description	Locally delayed latching (LDL) is used to synchronize data between mutually asychronous modules within a GALS system. Employs an asynchronous controller that assures that the high time of the clock of the module is long enough to resolve possible metastabilities at the input latch.		
Robust Interfaces for Mixed-Timing Systems		6.2.18	[10]
Description	The mixed-timing FIFOs provide a variable number of cells, each with a put (transmit) and a get (receive) interface; these cells are circular arranged. The modular structure of the mixed-timing FIFO enables several combinations and hence allow communication between clocked and/or self-timed modules. The clocked version employs a full/empty controller, the self-timed version uses a 4-phase handshake and single-rail bundled data communication channel instead.		
Pipeline Synchronization		6.2.19	[47]
Description	A pipeline using a k-staged 2- or 4-phase protocol FIFO (with bundled data channel) with an asymmetric or symmetric synchronizer element, respectively, per stage to synchronize mutually asynchronous modules or asynchronous input signals. Data stream synchronization is done along with the data flow.		

Synchronizer for systems with uncorrelated clocks

Name		Chapter	Reference
Interfacing Synchronous and Asynchronous Modules within a Highspeed Pipeline		6.2.13	[50]
Description	Combines a globally synchronous pipeline with locally asynchronous modules using a synchronously stoppable clock for the highspeed pipeline. The pipeline interface controller generates the stoppable clock (ring oscillator) without a mutual exclusion element. Further it handles the handshaking and saves the state of the clock in the moment when it is stopped. Uses bundled data approach (including inverter chain that matches the worst case computation path).		
Four-slot Fully Asynchronous Communication Mechanism		6.2.21	[49]
Description	An Asynchronous Communication Mechanism (ACM) that uses a shared memory communication mechanism without mutually exclusive access to provide fully asynchronous data transfer. A four-slot algorithm (see Listing 6.1) that avoids control variables in conditional statements is used to guarantee that no timing interferences between the communication partners will occur.		
A Fast Resolving BiNMOS Synchronizer for Parallel Processor Interconnect		6.2.16	[31]
Description	Connects two mutually asynchronous modules/systems using parallel staged Jamb-latches and a multiplexer for switching between the stages. To latch incoming data the Jamb-latches are alternately activated by an *enable* signal, that is generated by a ring counter. The multiplexer is also controlled by the *enable* signal, thus enables data forwarding of the successor stage (successor to the currently enabled latch). There is only one latch in the data path at a moment in time. Additional stages provide an increase in MTBF.		

Synchronizer for systems with uncorrelated clocks

Name		Chapter	Reference
Asynchronous Communication Mechanisms Using Self-Timed Circuits		6.2.22	[56]
Description	Another ACM based on the four-slot algorithm in Listing 6.1. A self-timed circuit with 4-phase handshake protocol and bundled data transmission approach. The ACM constrains possible metastability events to particular points in the control logic by using additional arbiters with metastability resolvers for write/read accesses. But loses the full asynchronism due to used arbiters in contrast to the ACM from 6.2.21. Note that transparent latches are used which employ David's cells.		
On-Chip Segmented Bus: A Self-timed Approach		6.2.27	[45]
Description	A self-timed segmented bus architecture to interface modules of a GALS system. Modules that are closely related in terms of computation are connected to the same bus segment. A master (transmitter) requests a connection to a slave (receiver), i.e. ownership of slave's segment, either in the same segment or in another segment via the inter-segment bridge, to communicate.		
The Even/Odd Synchronizer		6.2.20	[13]
Description	Synchronizes mutually asynchronous (clocked) systems by alternately (even and odd clock cycle) writing two registers. At the receiver a multiplexer picks data from the non last recently written register, to allow a possible metastability to resolve. The synchronizer is similar to the *Two-Register-Synchronizer* but with a fixed delay of half a clock cycle. Is used to synchronize FIFO read and write pointers, in direct use duplicates and drops may occur.		
Implementing a STARI Chip		6.2.23	[29]
Description	*Self-Timed At Receiver's Input* (STARI) uses self-timed as well as synchronous circuits. Communication is done between clocked modules (appear self-timed to the FIFO) via a self-timed FIFO (appears clocked to the modules). Receiver gets the position of the first data packet, then waits until FIFO is half-filled before it receives data items. STARI uses inverted dual-rail data encoding with a four-phase handshake protocol.		

Synchronizer for systems with uncorrelated clocks

Name		Chapter	Reference
A Solution to a Special Case of the Synchronization Problem		6.2.26	[51]
Description	Periodic Synchronizer, it uses a shift-register as basis and exploits the periodicity and predictability of the clocks (must be known in advanced). The synchronizer generates a window signal based upon the clock to signal if it is safe to sample or not. This window signal is synchronized to the receiver's clock domain by the shift-register. The synchronized signal never lags more than a fraction of receiver's clock.		
Efficient Self-Timed Interfaces for Crossing Clock Domains		6.2.24	[9]
Description	Improvement of the STARI approach to support mesochronous, plesiochronous, rational and mutually asynchronous clocking. Mesochronous clocking is improved by using a special latch that employs a generic latch (single-stage FIFO) and a latch controller (Muller C-Element) that generates the clock for the latch upon the transmitter's and receiver's clocks. For rational clocking an additional rate-multiplier has to be employed. For plesiochronous clocking a near-miss detector is needed, that detects if a rising clock edge occurs near to the self-reset of the latch controller. The latter both can be combined to support mutually asynchronous clocking.		
A Predictive Synchronizer for Periodic Clock Domains		6.2.25	[25]
Description	Synchronizes systems within periodic (mutually asychronous) clock domains by exploiting the periodic nature of the clocks. Detects conflicts between two clocks one clock cycle ahead and delays the sampling of the input in case of a predicted conflict.		

Synchronizer for systems with uncorrelated clocks

Name		Chapter	Reference
Fast Universal Synchronizer		6.2.4	[20]
Description	Synchronizes a 2-/4-phase handshake protocol between mutually asynchronous domains, using a bundled data channel.		
Point to Point GALS Interconnect		6.2.12	[37]
Description	Provides synchronous receiver and transmitter interface circuits to use an asynchronous interconnect within a GALS system. Uses a pausible clocking technique to transmit data. Can be extended by a buffering FIFO.		
Using Stoppable Clocks to Safely Interface Asynchronous and Synchronous Subsystems		6.2.14	[38]
Description	Using stoppable clock mechanism to synchronize incoming asynchronous signals to the local clock domain. Employs a 4-phase handshake to the asynchronous transmitter.		
Micropipeline		6.2.5	[53] [22] [55]
Description	Implements a 2-phase handshaking mechanism for a bundled data approach as a pipeline using Muller C-Elements. Namely a closed-loop handshaking pipeline/interface, using bounded delays. Also used for a ripple-through FIFO to interface modules of a GALS system, or for clock recovery, generation and distribution.		

9.1.4 Table of Ratiochronous Synchronizers

Ratiochronous Synchronizer			
Name		**Chapter**	**Reference**
Rational Clocking		4.2.1	[43]
Description	Based on the phase relationship of two clocks that is given as a rational number (frequency ratio). This is used to predict when it is safe to sample data and further to generate a communication schedule (e.g. written to a LUT). The schedule is used to enable or 'pause' a latch on the transmitter as well as on the receiver side. Clock may not be known in advance, therefore a Run Time Scheduling can be introduced.		
A Flexible Communication Scheme for Rational-Related Clock Frequencies		4.2.2	[5] [6]
Description	Based on the Low-Latency and Low-Overhead Mesochronous/-Plesiochronous Synchronizer [7]. Introduces a design style called *Globally-Rationchronous Locally-Synchronous* (GRLS), the basically frequencies of the communication partners has to be rationally related. The synchronizer can to cope with unknown phase differences (skew) in contrast to the rational clocking technique from [43]. Receiver interface samples incoming data either on the falling or rising clock edge, depending on the *strobe* signal sent by the transmitter. The *strobe* signal is generated by a regulator circuit at the transmitter toggles with the new data item at the output FIFO of the transmitter and is delayed and synchronized at the receiver, and further used to control input latches. Metastabilities may arise only during learning phase, when sampling *strobe* signal and thus off the data path.		

9.1.5 Table of Asynchronous (Self-timed) Synchronizer

Synchronizer for Self-Timed Systems

Name	Chapter	Reference
Glitching Synchronizer	7.2.2	[24]
Description	A Delay-insensitive interface circuit that employs a LEDR protocol (dual-rail) that synchronizes incoming data to the internal phase by using two flip-flops that are alternately activated based upon the transitions on the phase signal.	
Double-Edge Flip-Flop Synchronizer	7.2.2	[24]
Description	Extends the *Glitching Synchronizer* by a third flip-flop to avoid introduced glitches. Double-edge triggered flip-flops are required to change the output with each phase signal transition. Like the *Glitching Synchronizer* it synchronizes incoming data to the internal phase.	
Synchronization/Conversion Circuit	7.2.3	[24]
Description	A circuit that converts incoming signals from single-rail to dual-rail. It is used at input ports of delay-insensitive modules. Further employs a *Glitching Synchronizer* to synchronize the incoming data to the internal phase and calculates the phase line for a dual-rail signal.	
Arbiter	7.2.1	[12]
Description	The arbiter basically grants mutually exclusive access to a shared resource. This feature is used to synchronize asynchronous input signals into the work flow of a asynchronous (self-timed) system, by determining which of its inputs was asserted first.	
Micropipeline using Muller C-Elements with Schmitt-Trigger Masking	6.2.5	[40]
Description	Schmitt-Trigger masking is employed at the output of Muller C-Elements to prevent a possible metastable state to be propagated to successor stages. The enhanced Muller C-Element is thus used to build micropipelines. Due to the increased propagation delay such a micropipeline is only used in self-timed systems.	

Synchronizer for Self-Timed Systems - continued

Name		**Chapter**	**Reference**
A Modular Synchronizing FIFO for NoCs		7.2.5	[39]
Description	Synchronizes clocked as well as self-timed modules. The FIFO employs modular put (write) and get (read) interfaces, and uses the *asP** handshaking protocol. For long interconnects the LEDR protocol can be used.		
Chain: A Delay-Insensitive Chip-Area Interconnect		7.2.6	[2]
Description	A delay-insensitive NoC interconnect (RTZ), that enables data exchange between synchronous and asynchronous modules using a one-of-five data encoding. *Chain* provides several NoC building blocks as router and arbiters based on the presented pipeline latch (self-timed latch).		
A FIFO Ring Performance Experiment		7.2.4	[35]
Description	An asynchronous FIFO ring with 4-bit wide data path using the *asP** pulse-like protocol. In [35] a 17 stage FIFO ring is compared to a clocked shift register in terms of performance.		
GasP: A Minimal FIFO Control		7.2.7	[52]
Description	Asynchronous control circuit for simple (handshake) pipelines. Employs two circuit modules (PATH and PLACE), where the PATH (self-resetting NAND gate) functions as a gate between the pipeline stages, the PLACEs (inverter loop).		
A Doubly-Latched Asynchronous Pipeline		7.2.8	[34]
Description	A single-rail, 4-phase protocol approach for an asynchronous pipeline using two registers per stage in a master-slave hierarchy. The stages are decoupled by alternately latching data to the master and slave register (slave forwards to the successor stage master, while master latches from predecessor stage slave).		

Synchronizer for Self-Timed Systems - continued			
Name		**Chapter**	**Reference**
Implementing a STARI Chip		6.2.23	[29]
Description	The *Self-Timed At Receiver's Input* (STARI) approach uses a self-timed FIFO as interconnect. Since the communication partners appear self-timed to the self-timed FIFO it can also be used to interface real asynchronous (self-timed) systems, using an inverted dual-rail data encoding with a 4-phase handshake protocol.		

9.1.6 Table of Interconnects

Types of System Interconnects			
Name		**Chapter**	**Reference**
Surfing Interconnect		8.1.1	[30]
Description	The long wire interconnect consists of two lines (data line and timing chain). A request forwarded on the timing chain pulses inverters on the data line to refresh the data and thus keep the data signal stable on long communication channels. Surfing interconnect can be used between all types of system.		
Practical Asynchronous Interconnect Network Design		8.1.2	[42]
Description	An asynchronous staged interconnect using 2-phase handshaking technique and dual-rail encoding to connect a large number of differently clocked IP cores. Each IP core needs a 2-phase dual-rail interface or converter.		
Asynchronous Current Mode Serial Communication		8.1.3	[19]
Description	An asynchronous long-wire interconnect that is unaffected from voltage swing. This serial link is a dual-rail link and uses LEDR protocol for communication, further provides on-the-fly LEDR encoding. Employs a serializer that uses special fast (one gate delay) shift registers.		

Types of System Interconnects - continued

Name		Chapter	Reference
Asynchronous FPGA Architecture with Distributed Control		8.1.4	[48]
Description	A wrapper circuit for an asynchronous interconnect for FPGA modules (e.g. IP cores) based on David's cells. The controller circuit uses David's cells to implement the handshake protocol and signals the arrival of new data for computation.		
FLEETzero: An Asynchronous Switching Experiment		8.1.5	[11]
Description	FLEETzero is a chip developed for testing an asynchronous switch fabric based on GasP control circuits, that introduces an asynchronous pipeline network. It provides branching and merging components to enable connections between e.g. eight sources and eight destinations and offers new data every six gate-delays. A so-called *transport-triggered architecture* with a *communication-centric* focus to improve system performance in communication.		

CHAPTER 10

Conclusion

The purpose of this thesis was to gather information around synchronizers and interfacing solutions and methods to arrange a reference guide for technical engineers. The information was collected by an extensive literature survey. The results of the survey were refined and summarized in the tables of section 9.1 and described in detail in sections 3, 5, 6, 4 and 7 above to create a guide for developers and others. The synchronizers presented in this thesis are grouped in five rough categories, *Mesochronous*, *Plesiochronous*, *Asynchronous*, *Ratiochronous* and *Self-timed*. The basic concepts of the synchronizers were identified and described in these five categories. The solutions of each basic concept group were compared amoung each other in terms of MTBF, throughput and latency to further determine a prototype that is representive for its group. The resulting reference guide should support engineers in their decision which synchronizer or interface should be used between different timing domains. The following table (Table 10.1) shows the identified prototypes as the results of the comparisons between the synchronizers of each category, that were investigated during the survey (see Chapter 2).

Recommendation		
Classification	*Basic Concept*	*Prototype*
Mesochronous		
	data stream modification	Adaptive Synchronizer 3.2.5 [26]
	ring buffer	Four-Stage Mesochronous Synchronizer 3.2.7 [17]
	input latch control	Mesochronous Strobe Synchronizer 3.2.6 [7]
Plesiochronous		
	read pointer synchronization	Plesiochronous Strobe Synchronizer 5.2.4 [7]

Recommendation		
Classification	*Basic Concept*	*Prototype*
Asynchronous		
	handshaking	Micropipeline 6.2.5 [53]
	pausible clocking	Point to Point GALS Interconnect 6.2.12 [37]
	parallel staged	Wagging Synchronizer 6.2.15 [1]
	FIFO pointer synchronization	Even/Odd Synchronizer 6.2.20 [13]
	conflict detection	Periodic Synchronizer 6.2.26 [51]
Ratiochronous		
	GRLS	GRLS Strobe Synchronizer 4.2.2 [5] [6]
Self-timed		
	access control	Arbiter 7.2.1 [12] [33]
	self-timed data bus	Chain 7.2.6 [2]
	handshake pipeline	Modular Synchronizing FIFO for NoCs 7.2.5 [39]

Table 10.1: Synchronizer Prototypes

Bibliography

[1] M. Alshaikh, D. Kinniment, and A. Yakovlev. A synchronizer design based on wagging. In *Microelectronics (ICM), 2010 International Conference on*, pages 415–418, 2010.

[2] J. Bainbridge and S. Furber. Chain: a delay-insensitive chip area interconnect. *Micro, IEEE*, 22(5):16–23, 2002.

[3] S. Beer, R. Ginosar, R. Dobkin, and Y. Weizman. Mtbf estimation in coherent clock domains. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 166–173, May 2013.

[4] D.S. Bormann and P. Y K Cheung. Asynchronous wrapper for heterogeneous systems. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, pages 307–314, 1997.

[5] J. Chabloz and A. Hemani. A flexible communication scheme for rationally-related clock frequencies. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 109–116, 2009.

[6] J.-M. Chabloz and A. Hemani. Low-latency maximal-throughput communication interfaces for rationally related clock domains. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1–1, 2013.

[7] J.M. Chabloz and A. Hemani. Low-latency and low-overhead mesochronous and plesiochronous synchronizers. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 157–164, 2011.

[8] A. Chakraborty and M.R. Greenstreet. A minimal source-synchronous interface. In *ASIC/-SOC Conference, 2002. 15th Annual IEEE International*, pages 443–447, 2002.

[9] A. Chakraborty and M.R. Greenstreet. Efficient self-timed interfaces for crossing clock domains. In *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 78–88, 2003.

[10] T. Chelcea and S.M. Nowick. Robust interfaces for mixed-timing systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(8):857–873, 2004.

[11] W.S. Coates, J.K. Lexau, I.W. Jones, S.M. Fairbanks, and I.E. Sutherland. Fleetzero: an asynchronous switching experiment. In *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 173–182, 2001.

[12] William J. Dally and John W. Poulton. *Digital Systems Engineering*. Cambridge University Press, New York, NY, USA, 1998.

[13] W.J. Dally and S.G. Tell. The even/odd synchronizer: A fast, all-digital, periodic synchronizer. In *Asynchronous Circuits and Systems (ASYNC), 2010 IEEE Symposium on*, pages 75–84, 2010.

[14] Mark E. Dean, Ted E. Williams, and David L. Dill. Efficient self-timing with level-encoded 2-phase dual-rail (ledr). In *Proceedings of the 1991 University of California/Santa Cruz Conference on Advanced Research in VLSI*, pages 55–70, Cambridge, MA, USA, 1991. MIT Press.

[15] L.R. Dennison, W.J. Dally, and D. Xanthopoulos. Low-latency plesiochronous data retiming. In *Advanced Research in VLSI, 1995. Proceedings., Sixteenth Conference on*, pages 304–315, 1995.

[16] C. Dike and E. Burton. Miller and noise effects in a synchronizing flip-flop. *Solid-State Circuits, IEEE Journal of*, 34(6):849–855, 1999.

[17] Ran Ginosar Dmitry Verbitsky, Rostislay (Reuven) Dobkin. A four-stage mesochronous synchronizer with back-pressure and buffering for short and long range communications. Technical report, VLSI Systems Research Center, EE Dept., Technion, Haifa Israel, 2011.

[18] R. Dobkin, R. Ginosar, and C.P. Sotiriou. High rate data synchronization in gals socs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(10):1063–1074, 2006.

[19] R. Dobkin, M. Moyal, A. Kolodny, and R. Ginosar. Asynchronous current mode serial communication. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(7):1107–1117, 2010.

[20] Rostislav (Reuven) Dobkin and Ran Ginosar. Integrated circuit and system design. power and timing modeling, optimization and simulation. chapter Fast Universal Synchronizers, pages 199–208. Springer-Verlag, Berlin, Heidelberg, 2009.

[21] J. Ebergen. Squaring the fifo in gasp. In *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 194–205, 2001.

[22] S. Fairbanks and S. Moore. Analog micropipeline rings for high precision timing. In *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, pages 41–50, 2004.

[23] M. Ferringer. Conversion and interfacing techniques for asynchronous circuits. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 11–16, 2011.

[24] Markus Ferringer. Coupling asynchronous signals into asynchronous logic. In *Austrochip*, pages 97–102. Institut für Elektronik - TU Graz, 2009. poster presentation: Austrochip, Graz, Austria; 2009-09-07 – 2009-09-08.

[25] Uri Frank, Tsachy Kapshitz, and Ran Ginosar. A predictive synchronizer for periodic clock domains. *Formal Methods in System Design*, 28(2):171–186, 2006.

[26] R. Ginosar and R. Kol. Adaptive synchronization. In *Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings. International Conference on*, pages 188–189, 1998.

[27] Ran Ginosar. Fourteen ways to fool your synchronizer. In *Proc. 9th IEEE Int. Symp. on Asynchronous Circuits and Systems (ASYNC03*, 2003.

[28] Ran Ginosar. Metastability and synchronizers: A tutorial. *IEEE Design & Test of Computers*, 28(5):23–35, 2011.

[29] M.R. Greenstreet. Implementing a stari chip. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD '95. Proceedings., 1995 IEEE International Conference on*, pages 38–43, 1995.

[30] M.R. Greenstreet and J. Ren. Surfing interconnect. In *Asynchronous Circuits and Systems, 2006. 12th IEEE International Symposium on*, pages 9 pp.–106, 2006.

[31] J. Jex and C. Dike. A fast resolving binmos synchronizer for parallel processor interconnect. *Solid-State Circuits, IEEE Journal of*, 30(2):133–139, 1995.

[32] J. Kessels. Register-communication between mutually asynchronous domains. In *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, pages 66–75, 2005.

[33] David J. Kinniment. *Synchronization and Arbitration in Digital Systems*. 2008.

[34] R. Kol and R. Ginosar. A doubly-latched asynchronous pipeline. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, pages 706–711, 1997.

[35] C.E. Molnar, I.W. Jones, W.S. Coates, and J.K. Lexau. A fifo ring performance experiment. In *Advanced Research in Asynchronous Circuits and Systems, 1997. Proceedings., Third International Symposium on*, pages 279–289, 1997.

[36] C.E. Molnar, I.W. Jones, W.S. Coates, J.K. Lexau, S.M. Fairbanks, and I.E. Sutherland. Two fifo ring performance experiments. *Proceedings of the IEEE*, 87(2):297–307, 1999.

[37] S. Moore, G. Taylor, R. Mullins, and P. Robinson. Point to point gals interconnect. In *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pages 69–75, 2002.

[38] S. W. Moore, G. S. Taylor, P. A. Cunningham, R. D. Mullins, and P. Robinson. Using Stoppable Clocks to Safely Interface Asynchronous and Synchronous Subsystems. In *Proceedings of the AINT (Asynchronous INTerfaces) Workshop*, July 2000.

[39] T. Ono and M. Greenstreet. A modular synchronizing fifo for nocs. In *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, pages 224–233, 2009.

[40] Thomas Polzer, Andreas Steininger, and Jakob Lechner. Muller c-element metastability containment. In JoséL. Ayala, Delong Shang, and Alex Yakovlev, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 7606 of *Lecture Notes in Computer Science*, pages 103–112. Springer Berlin Heidelberg, 2013.

[41] J. Pontes, R. Soares, E. Carvalho, F. Moraes, and N. Calazans. Scaffi: An intrachip fpga asynchronous interface based on hard macros. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 541–546, 2007.

[42] B.R. Quinton, M.R. Greenstreet, and S. J E Wilton. Practical asynchronous interconnect network design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(5):579–588, 2008.

[43] L.F.G. Sarmenta, G.A. Pratt, and S.A. Ward. Rational clocking [digital systems design]. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD '95. Proceedings., 1995 IEEE International Conference on*, pages 271–278, 1995.

[44] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous interlocked pipelined cmos circuits operating at 3.3-4.5 ghz. In *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International*, pages 292–293, 2000.

[45] T. Seceleanu, J. Plosila, and P. Lijeberg. On-chip segmented bus: a self-timed approach [soc]. In *ASIC/SOC Conference, 2002. 15th Annual IEEE International*, pages 216–220, 2002.

[46] Ch. Seitz. Ideas about arbiters. *Lambda*, 1:10–14, 1980.

[47] J.N. Seizovic. Pipeline synchronization. In *Advanced Research in Asynchronous Circuits and Systems, 1994., Proceedings of the International Symposium on*, pages 87–96, 1994.

[48] Delong Shang, F. Xia, and A. Yakovlev. Asynchronous fpga architecture with distributed control. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 1436–1439, 2010.

[49] H.R. Simpson. Four-slot fully asynchronous communication mechanism. *Computers and Digital Techniques, IEE Proceedings E*, 137(1):17–30, 1990.

[50] A.E. Sjogren and C.J. Myers. Interfacing synchronous and asynchronous modules within a high-speed pipeline. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(5):573–583, 2000.

[51] W.K. Stewart and S.A. Ward. A solution to a special case of the synchronization problem. *Computers, IEEE Transactions on*, 37(1):123–125, 1988.

[52] I. Sutherland and S. Fairbanks. Gasp: a minimal fifo control. In *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 46–53, 2001.

[53] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, June 1989.

[54] V. Varshavsky. Self-timed control of concurrent processes. 1990.

[55] P. Wielage, E.J. Marinissen, M. Altheimer, and C. Wouters. Design and dft of a high-speed area-efficient embedded asynchronous fifo. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, 2007.

[56] F. Xia, A. Yakovlev, D. Shang, A. Bystrov, A. Koelmans, and D.J. Kinniment. Asynchronous communication mechanisms using self-timed circuits. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 150–159, 2000.

[57] K.Y. Yun and D.L. Dill. Unifying synchronous/asynchronous state machine synthesis. In *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on*, pages 255–260, 1993.

[58] K.Y. Yun and R.P. Donohue. Pausible clocking: a first step toward heterogeneous systems. In *Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on*, pages 118–123, 1996.

[59] Jun Zhou, D. Kinniment, G. Russell, and A. Yakovlev. A robust synchronizer. In *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, volume 00, pages 2 pp.–, 2006.