Globally-asynchronous, Locally-synchronous Wrapper Configurations For

Akarsh Ravi
University of Central Florida

Part of the Electrical and Electronics Commons
Find similar works at: https://stars.library.ucf.edu/etd
University of Central Florida Libraries http://library.ucf.edu

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation
https://stars.library.ucf.edu/etd/227
GLOBALLY-ASYNCHRONOUS, LOCALLY-SYNCHRONOUS WRAPPER
CONFIGURATIONS FOR POINT-TO-POINT AND MULTI-POINT DATA
COMMUNICATION

by

AKARSH REDDY RAVI

B.E. Bangalore University, 2001

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2004
ABSTRACT

Globally-Asynchronous, Locally-Synchronous (GALS) design techniques employ the finer points of synchronous and asynchronous design methods to eliminate problems arising due to clock distribution, power dissipation, and large area overhead. With the recent rise in the demand for System-on-a-Chip (SoC) designs, global clock distribution and power dissipation due to clock distribution are inevitable. In order to reduce/eliminate the effects of the global clock in synchronous designs and large area overhead in asynchronous designs, an alternative approach would be to utilize GALS design techniques. Not only do GALS designs eliminate the issue of using a global clock, they also have smaller area overhead when compared to purely asynchronous designs.

Among the various GALS design approaches proposed till date, this thesis focuses on the working and implementation of Asynchronous Wrapper designs proposed by Muttersbach et al., in [1, 2]. This thesis specifically addresses different approaches to incorporate the wrappers in VLSI circuits, rather than discussing the efficiency and viability of GALS design techniques over purely synchronous or asynchronous approaches. It has been proven by researchers [3] that GALS design approaches bring down power consumption due to the elimination of the global clock by small amounts, but there is also a drop in performance. Since the goal of this thesis is to introduce the reader to GALS design techniques and not prove their efficiency, it is out of the scope of this thesis to validate the results shown in [3].

In our aim to introduce the reader to GALS design techniques, we first provide a comparison of synchronous and asynchronous design approaches, and then discuss the need for
GALS design approaches. We will then address issues affecting GALS such as metastability, latency, flow control, and local clock alteration.

After familiarizing the reader with the issues affecting GALS, we will then discuss various GALS design techniques proposed till date. We show the use of asynchronous FIFOs and asynchronous wrappers to realize GALS modules. Two wrapper design approaches are discussed: one being the asynchronous wrapper design proposed by Carlsson et al., in [4], and the other being the asynchronous wrapper design proposed in [1, 2].

An in-depth discussion and analysis of the wrapper design approach proposed in [1, 2] is provided based on the state transition graphs (STGs) that characterize the port-controller AFSMs. Various data transfer channel configurations that incorporate the wrapper port-controllers are designed and realized through VHDL codes, with their functioning verified through simulation results. Design examples showing the working of asynchronous wrappers to achieve point-to-point, synchronous-synchronous and synchronous-asynchronous data communication are provided. Finally, a design example to achieve multi-point data communication is realized. This example incorporates a previously proposed idea. We provide a modification to this idea by designing an arbiter that arbitrates between two separate requests coming into a multi-input port.

Through the above design examples, the functionality and working of GALS asynchronous wrappers are verified, and recommendations for modifications are made to achieve flexible multi-point data communication.
ACKNOWLEDGEMENTS

I am honored to have had the opportunity to work with Dr. Jiann. S. Yuan as my thesis advisor. Without his technical and personal advice, this research work would not have been successful. I am very grateful to Dr. Abdel Ejnioui and Dr. Taskin Kocak for the help provided at various instances during my research and for their time and support in serving as my thesis committee members. I would like to thank all UCF faculty and staff for their time and help. I would also like to thank friends and colleagues at UCF for their good-natured advice and support throughout the period of my research. Finally, I would like to thank my parents and family members for all their blessings and well wishes.
# TABLE OF CONTENTS

LIST OF FIGURES ..................................................................................................................... viii

CHAPTER ONE: INTRODUCTION ............................................................................................. 1
  Objective ............................................................................................................................. 1
  Introduction to VLSI Design Methods ............................................................................. 1
  Synchronous Vs Asynchronous Design Methodologies ................................................. 3
  Clocking Issues in VLSI Systems ................................................................................... 4
  Power Consumption due to Global Clock Distribution ................................................. 5
  Research Challenges ........................................................................................................... 6
  Thesis Overview ................................................................................................................. 8

CHAPTER TWO: GALS DESIGN ISSUES AND TECHNIQUES ................................................. 9
  Issues Affecting GALS Design ......................................................................................... 10
    Metastability Avoidance ............................................................................................... 10
    Latency .......................................................................................................................... 12
    Flow Control .................................................................................................................. 13
    Local Clock Alteration ................................................................................................. 13
  Handshake Protocols ........................................................................................................ 14
  GALS Design Techniques ................................................................................................ 15
    Asynchronous FIFO ....................................................................................................... 16
      Low-Latency FIFOs for Single and Mixed-Clock Systems ..................................... 17
    Asynchronous Wrappers ............................................................................................ 20
      Carlsson Wrapper Design ......................................................................................... 23
      Muttersbach Wrapper Design ................................................................................. 27

CHAPTER THREE: MUTTERSBACH ASYNCHRONOUS WRAPPER DESIGN ......................... 29
  Port-Ty pes ......................................................................................................................... 29
    Input-Port and Output-Port ........................................................................................... 29
    Poll-Type and Demand-Type ....................................................................................... 31
  Port Controller AFSMs .................................................................................................... 31
    Demand-Type Input-Port Controller ........................................................................... 31
LIST OF FIGURES

Figure 1: Globally-Asynchronous Communication between Units [1] ................. 2
Figure 2: Hierarchical Clock Distribution Network [3] ........................................ 4
Figure 3: Power break-down in a High Performance CPU [9] ............................. 6
Figure 4: Energy Minima and Maximum .......................................................... 10
Figure 5: Occurrence of Metastability ............................................................... 11
Figure 6: Two-stage Synchronizing Unit ............................................................ 12
Figure 7: Reduction in Probability of Metastability ............................................ 13
Figure 8: Two-Phase Protocol ......................................................................... 14
Figure 9: Four-Phase Protocol .......................................................................... 15
Figure 10: Basic Asynchronous FIFO ............................................................... 16
Figure 11: Mixed-Timing Interface [12] .............................................................. 17
Figure 12: Simple Asynchronous FIFO Interface [12] ....................................... 18
Figure 13: Four-Phase Single-Clock Domain Asynchronous FIFO [12] .............. 19
Figure 14: Mixed-Clock Domain Asynchronous FIFO [12] ............................... 20
Figure 15: Point-to-Point Data Communication ............................................... 21
Figure 16: Multi-Point Data Communication ..................................................... 21
Figure 17: Asynchronous Wrapper design proposed by Muttersbach et al. [1] .. 22
Figure 18: Asynchronous Wrapper design proposed by Carlsson et al. [4] .... 22
Figure 19: Carlsson Wrapper [4] ..................................................................... 23
Figure 20: STG of a Demand-Active Port [4] .................................................... 24
Figure 21: Realization of Demand-Active Port [4] ............................................ 25
Figure 22: Realization of Demand-Passive Port [4] .......................................... 25
Figure 23: Stretchable Clock Controller - Carlsson Wrapper [4] ..................... 26
Figure 24: STG for the Carlsson SCC [4] .......................................................... 27
Figure 25: Muttersbach Wrapper [1] ................................................................. 27
Figure 26: Input Port of Muttersbach Wrapper ............................................... 30
Figure 27: Output Port of Muttersbach Wrapper ............................................. 30
Figure 61: Synchronous-Synchronous, Point-to-Point Data Transfer Channel Example........70
Figure 62: Simulation results for Point-to-Point Data Transfer Channel Example........74
Figure 63: Multi-Point Data Communication Model.................................................75
Figure 64: Synchronous-Synchronous, Multi-Point Data Transfer Channel Example........76
Figure 65: Simulation results for Multi-Point Data Transfer Channel Example............79
Figure 66: Arbiter for Multi-Port Data Communication.............................................81
Figure 67: Diagrammatic waveforms showing the functioning of the Multi-Point Arbiter ....83
Figure 68: Simulation results for the Multi-Point Arbiter ...........................................83
Figure 69: Synchronous-Asynchronous Data Transfer Channel Example.....................84
Figure 70: Simulation results for Synchronous-Asynchronous Data Transfer Channel........86
CHAPTER ONE: INTRODUCTION

Objective

This M.S. thesis is intended to familiarize the reader with the syntax and semantics of the Globally-Asynchronous Locally-Synchronous (GALS) design methodology to design Very Large Scale Integrated (VLSI) systems, various approaches to design and optimize GALS systems, and to discuss and analyze the experimental results of various implementations. The main focus will be towards verifying the functioning of Asynchronous Wrappers in various modes of operation.

Introduction to VLSI Design Methods

VLSI design has so far adhered only to the ‘globally-synchronous’ design paradigm. Also, with recent rise in the System-on-a-Chip (SoC) design approach, problems arising due to issues such as clock skew, and power consumption caused by global cock distribution have become more significant. The distributions of low-skew, high frequency clock signals are responsible for a considerable share of power consumption and die area. Additionally, the implementation of multitude clock frequencies in SoC’s, require for each multi-clock interface to incorporate synchronizers to reduce the possibility of metastability, thus leading to widespread control overhead.

A theoretically viable but practically non-viable design strategy to overcome the above mentioned problems is the use of Asynchronous design methodologies. In asynchronous designs,
data transactions are controlled by local handshake signals. If designed properly asynchronous systems consume less power, are robust, and may be faster than their synchronous counterparts. But the price to pay is the inclusion of widespread control overhead. This often increases die size significantly, and leads to complications in the design. Also, the lack of familiar synthesis CAD tools makes it even more unacceptable for asynchronous designs to substitute the current synchronous VLSI design methods.

An alternative approach to these design methods is the ‘Globally-Asynchronous Locally-Synchronous’ (GALS) design methodology, shown in Figure 1 – an integration of both, synchronous and asynchronous design methods.

Figure 1: Globally-Asynchronous Communication between Units [1]

In GALS design, asynchronous interfaces are added to locally synchronous (LS) modules, where-in the interiors of a specific module are encapsulated by the interfaces; and it is possible for each module to use its own clock, power supply voltage, or even to collapse the power supply for the synchronous part without affecting any other parts of the chip. In GALS designs, each distinct module can thus generate its own clock signal, and communication between modules is asynchronous. Therefore, the problem of power consumption due to global-
clocking, and clock skew are avoided; and these issues are limited to the local synchronous modules. Hence, GALS design methods possess many advantages – not only is there a mitigation in the clock distribution problems such as increased power consumption, and the problem of clock skew arising due to large chip area– there is also a simplification in the reuse of modules.

**Synchronous Vs Asynchronous Design Methodologies**

Synchronous systems are those in which a single clock is used to synchronize all events. The advantages of pursuing synchronous design techniques are:

- Simplest way to implement sequencing
- Widely taught and readily understood
- Easy availability of components

The main disadvantages of synchronous design techniques are:

- Clock distribution, clock skew
- Not modular
- High power consumption

On the other hand, asynchronous systems are those which do not incorporate any clock; all communication takes place through the utilization of handshaking signals. The main advantages of asynchronous design techniques are:

- Elimination of problems arising due to clock
- Component modularity
- Lower system power requirements

The main disadvantages of pursuing asynchronous design are:

- Lack of mature CAD tools
Clocking Issues in VLSI Systems

In VLSI systems, clock skew is a major factor influencing the efficiency of the system. Clock skew can be tolerable to certain extents. But due to factors such as interconnect delays (mismatching wire lengths), gated clocks, and non-similar buffering methods, clock skew becomes uncontrollable.

In most systems, a phase lock loop (PLL) generates a high frequency clock signal from a slower external clock. Trees of buffers are used to distribute the high frequency clock throughout the chip. These trees work well if clock loading is uniform across the chip area; unfortunately, most systems have widely varying clock loads. Figure 2, shows an example of a clock distribution network. Several major clocks are derived from the global clock grid, and local clocks are in turn derived from the major clocks.

![Hierarchical Clock Distribution Network](image)

Figure 2: Hierarchical Clock Distribution Network [3]
Restle et al., [5] argue that clock skew arises mainly due to process variations in the tree of buffers driving the clock. Since device geometries will continue to shrink, and clock frequencies and die sizes will continue to increase, global clock skew induced by such variations only get worse.

As per the information stated in [6], CMOS feature sizes are decreasing – from 130 nm in 2001 to a predicted 22 nm in 2016 – the outcome of which will be an increase in the number of transistors per chip.

Also, CMOS local clock frequencies are increasing, from 1.7 GHz in 2001 to a predicted 28.7 GHz in 2016. In addition to this, earlier, transistors used to limit the speed of operation, but in the future, wires will be the limiting factor. Thus interconnect speeds will not keep up to transistor speeds.

It is evident from all of these current facts and future requirements that clock skew is and will be a major factor affecting clock period. Hence, VLSI circuits will eventually need to be designed without the ‘global-clock’.

**Power Consumption due to Global Clock Distribution**

VLSI design has reached a point where systems are increasing in size and are being clocked at high frequencies. This has led to an overhead in the form of power consumption, which is unacceptable for efficient working of VLSI systems. This is confirmed by [8] for high performance microprocessors. Figure 3 shows a power consumption breakdown in a high-performance CPU.
VLSI systems divided into smaller autonomous sub-modules can contain this overhead caused by global clock distribution – a completely asynchronous system being the extreme case. But due to control overhead and lack of CAD tools for synthesis purposes, asynchronous design methods have not yet matured in terms of widespread acceptability.

Since GALS architecture is composed of large LS blocks that communicate with each other asynchronously, the global clock can be eliminated. By doing so, a major source of power consumption is eliminated. Also, since LS modules operate asynchronously with respect to each other, the frequency at which a LS module operates can be tailored according to the local needs, thus reducing the average frequency and the overall power consumption. Hence, adopting a GALS design methodology is advisable to achieve efficient power consumption in VLSI systems.

**Research Challenges**

GALS is still conceptually young; there is no formal method for designing optimal GALS systems. One of the main issue that was to be considered while pursuing this thesis was the selection of the GALS design approach that best suited our objective. Among the various GALS...
design approaches proposed till date, the asynchronous wrapper design method proposed in [1] (we shall hence forth call it the ‘Muttersbach et al. wrapper’) is most suitable to verify the concept of GALS.

The Muttersbach, et al. wrapper is the most modular of all GALS design approaches. Also, it incorporates a clock stretching mechanism which is adaptable to any number of input clock-stretch requests, without the problem of metastability arising. Metastability is minimized by the use of mutual exclusive (ME) elements. Since this design approach is highly flexible with regards to the use of varied interface controllers, it gives us the opportunity to better understand the working of GALS wrappers in different scenarios.

Our second goal is to thoroughly understand the working of all the different types of port controllers proposed in [1], which are incorporated in various data transfer channel configurations. After having understood the individual and combined functioning of the port controllers, it is required to test their functionality in hypothetical examples.

So far, GALS design approaches have been researched in-depth with regards to point-to-point data communication. A model for multi-point data communication has been proposed in [10]. The third goal is to incorporate the multi-point data communication model in a RAM example, and verify its functioning through simulation results.

A limitation with the above proposed idea is the lack of flexibility to adapt to independent data communication between the modules. To overcome this limitation, we propose to use an arbiter designed to arbitrate between two separate port-request signals within one clock cycle of the multi-point module.

GALS design methodology is not limited to only synchronous-synchronous module communication, but can also be applied to achieve synchronous-asynchronous module
communication. As our final goal, a verification of this is provided through the example of an asynchronous ROM communicating with a synchronous module through a demand-type port asynchronous wrapper interface.

**Thesis Overview**

This thesis is organized into five chapters. Chapter Two discusses about various design issues that affect GALS designs. It also takes a look at different GALS design approaches, them being the asynchronous FIFO and asynchronous wrapper design approaches.

In Chapter Three, an in-depth understanding of the various blocks (ports/port-controllers) constituting the Muttersbach wrapper is achieved, along with an understanding of the flow of control within these blocks. Extended burst-mode specifications (XBMS) are used to define the various port-controller AFMS. Diagrammatic waveforms are provided to illustrate the functioning of each of the individual port-controllers.

Chapter Four concentrates on the application of the Muttersbach wrapper design approach to achieve different wrapper configurations of the port-controllers. Four combinations of data transfer channels are delved into. Simulation results through VHDL coding verify the working of these wrapper configurations. Design examples to understand the working of asynchronous wrappers to achieve point-to-point and multi-point data communication between modules in a sub-system are provided. A new arbiter design is proposed to achieve more flexible multi-point data communication.

Chapter Five highlights the contributions of this thesis and provides directions for future research.
CHAPTER TWO: GALS DESIGN ISSUES AND TECHNIQUES

As mentioned earlier, GALS design approaches incorporate a combination of the advantages of synchronous and asynchronous design methodologies while avoiding their disadvantages.

The major reasons for pursuing the GALS design methodology are:

- Easy to design systems with multiple clock domains.
- No global clock signal to distribute.
- Power efficient: System operates only when data are available. A form of clock-gating.
- Limited, standard asynchronous circuitry.
- Comparable performances.

The issues to be addressed during the design of a GALS system are:

- Metastability avoidance.
- Latency.
- Flow control.
- Local clock alteration.

The following section reviews these issues, providing an implicit relation between metastability avoidance, latency, flow control, and local clock alteration.
Issues Affecting GALS Design

Metastability Avoidance

Metastability occurs due to the presence of an unstable equilibrium point between two stable equilibria (Logic 1 and Logic 0). Metastability avoidance means to avoid any change in value of a control signal (asynchronous control signals in GALS) just before the signal is clocked into a synchronous logic block.

Metastability occurs whenever there is setup and hold time violations in flip-flops. Flip-flops are used as synchronizers when asynchronous signals are interacting with synchronous modules. The problem to be considered is that flip-flops, like all bi-stable elements, have two potential energy minima and a maximum separating the minima, as shown in Figure 4. Due to the occurrence of setup and hold time violations, the flip-flop enters into a meta-stable (quasi-stable) state at the maximum, and will eventually settle down to Logic 0 or Logic 1, as shown in Figure 5.

Figure 4: Energy Minima and Maximum
In reality, one cannot avoid metastability while synchronizing asynchronous signals without the use of complicated self-timed circuits. In order to bring down metastability to tolerable levels, we can, as stated in [16]:

- Make sure the clock period is long enough to allow for the resolution of quasi (meta)-stable states as well as whatever logic that may be in the path of the next flip-flop. This approach, while simple, is rarely practical given the performance requirements of most modern designs. A solution to this is the use of Pausible clocks, proposed by [12].

- By adding one or more successive synchronizing flip-flops to the synchronizer, for example, a two-stage synchronizing unit as shown in Figure 6. This approach provides for an entire clock period for meta-stable events in the first synchronizing flip-flop to themselves resolve. This does, however increase the latency in the synchronous logic’s observation of input changes. This latency

Figure 5: Occurrence of Metastability
issue is discussed in the following section, and a method to decrease latency, as proposed by Chelcea and Nowick in [12], is briefed upon later on in this document.

![Two-stage Synchronizing Unit](image)

Figure 6: Two-stage Synchronizing Unit

**Latency**

Latency is defined as the delay from the time of input of a data item until the corresponding output of data is produced.

Latency is an issue to be considered during the design of a GALS system due to the effect it has while implementing techniques to avoid metastability. As mentioned previously, the probability of metastability can be reduced by using a chain of flip-flops as shown in Figure 7. But this method has largely been rejected due to excessive latency. However, low-latency FIFOs have been designed to interface mixed-clock systems, and are further discussed in this document.
Flow Control

Flow control deals with the flow of data in a multi-clocked system. If the sending module’s frequency is higher than the receiving module’s frequency, there will be a pile up of signals, thus requiring buffering.

Flow control can be achieved by incorporating a local clock alteration method as discussed in the following section.

Local Clock Alteration

Local clock alteration is a process through which metastability can be avoided efficiently. If an asynchronous signal is to arrive at a time when there is an imminent danger of metastability occurring, the interface unit is designed to alter the local clock. Local clocks can be paused until all input data are available, thereby ensuring the presence of stable data. Also, flow control is achieved by stopping the local clock.

Various circuits to alter local clocks have been proposed by researchers, and are used within asynchronous wrappers in conjunction with port controllers. Some of these local clock alteration circuits are discussed further in this chapter.
**Handshake Protocols**

In GALS systems, data communication between modules is not dependent on clock signals as is the case in completely synchronous systems; but rather, asynchronous handshake signals are used to indicate the availability and arrival of data. This is also known as bundled data (data + handshake signals). The two most common asynchronous handshake protocols are the two- and four-phase protocols. Both protocols use a request and an acknowledge signal.

In the case of a two-phase protocol, Figure 8, the transmitting module indicates to the receiving module the availability of data by a transition on the request signal. After the receiving module utilizes the data, it sends a transition on the acknowledge signal. This completes the transmission, and the next data exchange will start with a new transition on the request signal. This protocol is also known as transition or non-return-to-zero (NRZ) signaling. The data valid periods are as shown in Figure 8.

![Figure 8: Two-Phase Protocol](image)

In a four-phase protocol scenario, Figure 9, the transmitting module indicates the start of a data transfer by a transition on the request signal. The receiving module acknowledges this by a
transition on the acknowledge signal. In response to which, the transmitting module then performs another transition on the request signal restoring the value to its initial value. The receiving module restores the acknowledge signal after accepting the data, but only after the request signal has been restored. This is also known as level or return-to-zero signaling. The data valid period can be divided into three schemes [13], the early, broad, and late schemes as shown in Figure 9.

![Four-Phase Protocol](image)

Figure 9: Four-Phase Protocol

**GALS Design Techniques**

Communication in GALS systems can be achieved by the use of FIFOs to synchronize communicating modules or by implementing asynchronous wrappers that incorporate handshake signals (port controllers) and local clock generators.

The following sections give a brief introduction to some of the synchronization procedures in use.
Asynchronous FIFO

Asynchronous FIFO architectures usually have two interfaces: One for writing data into the FIFO, and the other for reading out data from the FIFO. Asynchronous FIFOs also have two clocks, one for writing and the other for reading. A basic block diagram of an asynchronous FIFO is shown in Figure 10.

Asynchronous FIFOs also have certain control signals incorporated in their architecture to facilitate error free operation. The importance of these control signals is explained in the discussion of Low-Latency FIFOs for Mixed-Clock systems proposed by Chelcea and Nowick in [12]. Asynchronous FIFOs are used when the performance of the system is of importance, when one does not want to waste clock cycles by using handshake signals. But however, incorporating FIFOs leads to latency.
**Low-Latency FIFOs for Single and Mixed-Clock Systems**

The use of asynchronous FIFOs in GALS systems in order to provide synchronization between modules leads to latency. Even with this known fault, the use of FIFOs is acceptable because there is a non-occurrence of meta-stability, unlike in asynchronous wrapper architectures. Hence, with the development of GALS design methodologies, a need to design low-latency FIFOs for mixed-clock systems arises. Chelcea and Nowick have proposed such a design in [12]. They first define an asynchronous FIFO design for single clock domains and then adapt the same design technique for mixed-clock systems. These low-latency designs are adaptable to scenarios where communication can be between mixed-clock synchronous modules, asynchronous-synchronous modules, or just between asynchronous-asynchronous modules as shown in Figure 11.

![Figure 11: Mixed-Timing Interface [12]](image)

Let us first review the single-clock domain asynchronous FIFO. The basic asynchronous FIFO interface is as shown in Figure 12.
As shown above, the asynchronous FIFO interfaces two subsystems: a *sender*, which produces data items and a *receiver* which consumes data items. The “put interface” consists of a data bus \( \text{data}_{\text{put}} \) (used to send data items), a \( \text{req}_{\text{put}} \) input (requests to en-queue a data item) and a ‘full’ output (signals when the FIFO is full). The “get interface” consists of a data bus \( \text{data}_{\text{get}} \) (used to receive a data item), a \( \text{req}_{\text{get}} \) input (request to de-queue a data item), a \( \text{valid}_{\text{get}} \) output (used to signal the validity of a data item) and an empty output (indicates when the FIFO is empty). In this design, the two clocks are identical \( (CLK_{\text{put}} = CLK_{\text{get}} = CLK) \).

Figure 13 shows the basic architecture of a 4-phase single-clock domain asynchronous FIFO. It consists of a circular array of identical cells, a *full detector* and an *empty detector*, and control logic for the ‘put’ operation and ‘get’ operation. The full empty detectors observe the state of the FIFO and determine whether the FIFO is full or empty. The input and output behavior of the FIFO is controlled by the flow of two tokens: *put token* (used to en-queue data items) and a *get token* (to de-queue data items). Once a data item is input, it is almost immediately available at the output.
Let us now look at the mixed-clock domain asynchronous FIFO. For this mixed-clock FIFO, Chelcea and Nowick [12] have incorporated the same cell architecture as the one used for single-clock FIFO. But also, since two different clocks control the operation of the FIFO cells, additional synchronizers are used and modifications to the external logic have been made.

The solution to this matter is to introduce synchronization on the global control signals - full and empty, as shown in Figure 14. These global signals are now synchronized to their individual clocks, $CLCK_{put}$ and $CLK_{get}$, respectively. This synchronization however introduces additional latencies that results in data inconsistencies. Due to this, the authors re-define the terms ‘full’ and ‘empty.’ The main concern now is to detect when the FIFO is heading towards a full or empty state, and stop the respective interface in time. Also, because of the early detection of empty, deadlock can occur. The possibility of stalling the receiver when there is a single valid data item in the FIFO should be avoided. Hence a deadlock detector is introduced which introduces a dummy (invalid) data item into the queue so that the receiver does not view the queue as being empty.

![Figure 13: Four-Phase Single-Clock Domain Asynchronous FIFO [12]](image-url)
Chelcea and Nowick have also proposed similar low-latency FIFO designs compatible for use in synchronous-asynchronous and asynchronous-asynchronous interfaces.

**Asynchronous Wrappers**

Asynchronous wrappers, as mentioned before, are used to completely encapsulate the synchronous modules in GALS systems. In order to achieve this, these wrappers should be capable of generating a clock signal to the LS module. Also, to-and-fro data transfer between modules, through wrappers, must be accompanied by request-acknowledge handshake signals. Both two and four-phase handshake signals can be used to achieve handshaking. Normally, a four-phase protocol is less efficient in terms of signal transitions, but is easier to implement in terms of control circuitry. The four-phase protocol is preferred in the case of GALS design since it is possible to efficiently use the request and acknowledge signals to control data latches that input data into the receiving module. Data communication between modules can be achieved
either through *point-to-point* or *multi-point* configurations, as shown in Figures 15 and 16, respectively.

![Figure 15: Point-to-Point Data Communication](image1)

![Figure 16: Multi-Point Data Communication](image2)

Asynchronous wrappers consist of port controllers characterized by Asynchronous Finite State Machines (AFSM) that control the I/O ports of a module. Additionally, wrappers provide for these controllers to be able to request for the stretching of local clock signals. Block diagrams
of asynchronous wrappers proposed by Muttersbach et al. [1] and Carlsson et al. [4] are as shown in Figure 17 and Figure 18, respectively.

Figure 17: Asynchronous Wrapper design proposed by Muttersbach et al. [1]

![Asynchronous Wrapper design proposed by Muttersbach et al.](image1)

Figure 18: Asynchronous Wrapper design proposed by Carlsson et al. [4]

![Asynchronous Wrapper design proposed by Carlsson et al.](image2)

Both of the above wrapper designs comprise of almost similar components: a local clock generator, I/O ports, handshake signal lines, control signals, data bus lines, and a synchronous module. The above two wrappers differ in their AFSM states and method of generating the local clock.
We will next, briefly discuss the design of the wrapper proposed by Carlsson et al [4], including the state transition graphs (STG), the ports, and the local clock circuit proposed in [4], followed by an introduction to the wrapper design proposed by [1] in the following sections. We will then follow up with an in-depth discussion of the wrapper designs proposed by Muttersbach et al. in Chapter 3.

For the sake of simplicity, we will hence forth refer to the wrapper design proposed by Carlsson et al. in [4] as the ‘Carlsson wrapper’, and the one proposed by Muttersbach et al. in [1] as the ‘Muttersbach Wrapper’.

**Carlsson Wrapper Design**

The Carlsson wrapper as shown in Figure 19 consists of a locally synchronous module, a clock generator, and I/O ports encapsulated within the wrapper. This wrapper is extremely modular in that it adapts the interface circuit to changes in voltage requirements.

![Figure 19: Carlsson Wrapper](image)

A signal RATE REQUIREMENTi is used to scale the local power supply voltage so that the synchronous module can process data at the rate at which they arrive. Speed of the circuit is
given by \( f^*(V_{dd}) \), where \( f \) is the frequency of operation and \( V_{dd} \) is the supply voltage. Hence we can control data rate by varying \( V_{dd} \). According to this design, a synchronous module can be inserted into the wrapper, and then the wrapper will convert from/to the voltage outside the wrapper to/from the right voltage for the synchronous module.

Data communication can be divided based on several factors. When considering how the local clock is handled, there are two possible types, *Demand* or *Poll*; based on handshake, the ports could either be *Active* or *Passive*.

Demand ports stop the internal clock as soon as the synchronous module tries to send/receive data. The internal clock is stopped under the handshake process in a Poll port configuration only if necessary. An active port begins the handshake process, while a passive port ends the handshake process.

The signal transfer graph (STG) of a demand-active port is as shown in Figure 20. This STG defines the demand active port with \( En \) and \( Ack \) signals being inputs to the port, and \( Req \) and \( Str \) signals being the outputs of the port. The realization of this STG is shown in Figure 21. Similarly, the realization of a demand-passive port is as shown in Figure 22.

![Figure 20: STG of a Demand-Active Port [4]](image_url)
The functioning of the active and passive demand ports is very much similar. Let us consider the passive port in Figure 22. As explained in [4], the handshake process starts with a transition (either positive or negative) on \textit{Den} which will generate a positive pulse on \textit{Xa}. This pulse sets \textit{Stretch} ‘high’ as long as necessary. When \textit{Stretch} is ‘high’, the AND gate allows \textit{Ack} to follow \textit{Req}, thus enabling the handshaking process. When handshaking ends, i.e., \textit{Ack} goes ‘low’, it will generate a pulse on \textit{Xb} which will reset \textit{Stretch} back to zero. \textit{Ack} will stay ‘low’ until a new handshake is enabled by a transition on \textit{Den}. 

Figure 21: Realization of Demand-Active Port [4]

Figure 22: Realization of Demand-Passive Port [4]
As in all GALS systems, each synchronous module is dictated by a local clock. These local clocks are designed to be stretchable so that proper synchronization between communicating modules can occur. The clock alteration process in the Carlsson wrapper is achieved through the use of the *Stretchable Clock Controller* (SCC), designed by Carlsson *et al.* as shown in Figure 23.

![Figure 23: Stretchable Clock Controller - Carlsson Wrapper [4]](image)

The SCC consists of a ring oscillator made up of a number of inverters, a Muller C element and simple gates. As per the explanation in [4], the stretch signal from the port controller *Stretch* arrives at the input of the SCC and is XORed with a feedback signal from the SCC. According to the properties of the Muller C element, if *Stretch* is asserted ‘low’, the output of the C element will follow the inputs and a clock signal is generated. On the other hand, if *Stretch* is asserted ‘high’, the input $X_a$ is set to ‘low’, the output of the C element could either be ‘low’ or ‘high’. However, the output will eventually be maintained ‘low’. Thus the next rising edge of the clock signal is postponed by *Stretch*+. Figure 24 is the STG for the SCC. It clearly indicates the flow/transition of signals in the SCC. Also, provision for multiple requests of stretching is provided by connecting all the inputs, *Stretch* (1-i) to an OR gate; as soon as one stretch signal goes ‘high’, the clock will be stretched.
**Muttersbach Wrapper Design**

The wrapper architecture proposed by Muttersbach, *et al.* [1], as shown in Figure 25, employs the four-phase protocol for handshaking, where each handshaking cycle comprises of four sequential events, \( \text{Req}^+, \text{Ack}^+, \text{Req}^-, \text{Ack} \). The wrappers consist of I/O ports dictated by port controllers. Also all ports within a wrapper can request for clock stretching.

The Muttersbach wrapper, just like the Carlsson wrapper, has ports defined based on the way the local clock is handled: they could be either *Demand*- or *Poll-type* ports. As mentioned
previously, demand-type ports stretch the local clock when data transfer is immediately required. Poll-type ports are however used when data transfer is required, but not immediately. In Chapter 4, we will discuss in detail all combinations of ports that are used in the Muttersbach wrapper: demand-type input port, demand-type output port, poll-type input port, and poll-type output port.

Each type of port controller is characterized by an AFSM, which is defined by an extended burst-mode specification [14]. Muttersbach et al. [1, 2] use a 3-D tool set, through which a direct synthesis of a hazard-free two-level AND-OR circuit from the burst-mode specification was achieved. Since it is out of the scope of this thesis to use a 3-D tool set, VHDL programming language is used to realize the ports from the extended burst-mode specifications.

The Muttersbach wrapper incorporates a local clock generation method based upon the Pausible Clock Controller (PCC) technique proposed in [11]. This local clock generator is however a simpler variation of the one in [11], and is easier to implement. This clock generator can easily be modified to support multiple clock alteration requests.
CHAPTER THREE: MUTTERSBACH ASYNCHRONOUS WRAPPER DESIGN

In this chapter we discuss in detail the different types of ports, the port-controller AFSMs that characterize the ports of the Muttersbach Wrapper, and the clock generation techniques proposed by Muttersbach et al in [1, 2]. We begin by defining the structure of an Input port, followed by that of an Output port. Then we provide definitions for Demand- and Poll-type ports.

After defining the type of ports, an in-depth discussion of the different port-controller AFSMs is provided, beginning with the extended burst-mode specification of each AFSM, followed by an explanation of the working of the port-controller with the aid of diagrammatic waveforms.

Port-Types

A GALS port consists mainly of a port-controller defined by an AFSM and a flip-flop for signaling the completion of a transfer. Input ports additionally consist of latch register banks. The use of latches to latch transmitted data vectors between communicating modules helps prevent data misses and ensures data correctness.

Input-Port and Output-Port

The block diagram of the input port is as shown in Figure 26. It consists of a port-controller, either a D-type or a P-type, a flip-flop to indicate the transfer of data, and a latch to
hold onto incoming data. The architecture of an *output-port*, Figure 27, is similar to that of an input port but for the exclusion of the latch.

Figure 26: Input Port of Muttersbach Wrapper

Figure 27: Output Port of Muttersbach Wrapper

All data transfers on a particular port in a GALS system are managed by port-controllers. The enable signal triggered by the LS module uses transition signaling, while all the signal links
between the port-controller and the clock generator and between the two communicating port-controllers employ 4-phase handshaking.

**Poll-Type and Demand-Type**

A *poll-type* (P-type) port issues requests for clock stretching exclusively to prevent metastability thus ensuring correctness of the data being transferred. The clock is influenced as infrequently as possible [15]. A P-type port is used whenever a data transfer is possible but is not necessarily required to happen immediately. The locally synchronous module continues to work normally, while the P-type port controls the data transfer.

A *Demand-type* (D-type) port also ensures data integrity on the transfer channel but incorporates a feature similar to clock-gating [15]. When a D-type port is enabled, it immediately stretches/pauses the local clock and does not release it until the data transfer has taken place. A D-type port is used whenever data transfer is required immediately – when no further computations can be performed by the locally synchronous island without the data being transferred.

**Port Controller AFMUs**

**Demand-Type Input-Port Controller**

The I/O ports of the demand-type input port controller are shown in Figure 28. Immediately upon activation by a switching event *Pen*, it issues for a clock stretch request by setting *Ri* ‘high’, which is then acknowledged by a ‘high’ on *Ai*. With the clock remaining ‘low’, the external handshake cycle on *Rp/Ap* gets processed and subsequently, the clock may resume again.
The extended burst-mode specification shown in Figure 29, gives us the state graph (STG) to describe the behavior of an asynchronous FSM that satisfies the requirements of the demand port controller.

- As you can see, the port is idle in state ‘0’. A transition from state ‘0’ to state ‘1’ is performed upon activation of the enable signal $Pen^+$. As incoming request $Rp^+$ can occur at any time, it is allocated the don’t-care status during this transition period using the * symbol. This enable signal causes an output of $Ri^+$ which requests a clock pause.

- The STG transitions from state ‘1’ to state ‘2’ due to the activation of the $Ai^+$ signal which is the acknowledge generated by the PCC in response to the $Ri^+$ signal. Also, an activation of $Rp^+$ can be the input during this period too. The
output during this stage is the activation of the acknowledge signal $Ap^+$. Hence by stage ‘2’, handshake signals have been processed for data transfer.

- Stage ‘3’ is achieved when the $Rp$- signal gets activated as an input, the output to which would the activation of the request to resume clock generation $Ri^-$. This would lead to the generation of the acknowledge signal $Ai^-$, the response to which would be the generation of the $Ap^-$ signal.

- Thus at stage ‘4’, the FSM has executed the whole transfer cycle and is idle again. Since $Pen$ is a transition signal, the FSM is duplicated again, upon the activation of $Pen^-$. The diagrammatic waveforms of a D-type input port are shown in Figure 30.

![Diagrammatic waveforms for Demand-type Input AFSM](image)

A common characteristic of demand-type ports is that they issue a clock request, immediately after an enabling event on the controller. To be sure that the next rising clock edge
gets delayed, this event \( Ri^+ \) has to reach the ME before the \( rclk^+ \). This constraint can be met by the use of the negative clock edge to generate the event on \( Pen \).

**Demand-Type Output-Port Controller**

The I/O ports of the demand-type output port controller are shown in Figure 31.

![Demand-type output AFSM](image)

Figure 31: I/O Ports of Demand-type Output AFSM [1]

The extended-burst-mode specification shown in Figure 32 gives us the state graph (STG) to describe the behavior of an asynchronous FSM that satisfies the requirements of the demand port controller.

![XBSM of Demand-type Output AFSM](image)

Figure 32: XBSM of Demand-type Output AFSM [1]
• The port is idle in state ‘0’. A transition from state ‘0’ to state ‘1’ is performed upon activation of the enable signal Pen+. This enable signal causes an output of Ri+ which requests a clock pause.

• The STG transitions from state ‘1’ to state ‘2’ due to the activation of the Ai+ signal which is the acknowledge generated by the PCC in response to the Ri+ signal. The output during this stage is the activation of the request signal Rp+.

• Stage ‘3’ is achieved when the port gets Ap+ as input, the output response to which would the de-activation of the request signal to Rp-.

• A transition from stage ‘3’ to stage ‘4’ occurs; the controller generates the output Ri- in response to the input Ap-. Ri- is the request sent to the PCC to de-activate clock stretching thus resuming the generation of the local clock signal.

• Thus at stage ‘4’, the FSM has executed the whole transfer cycle. Since Pen is a transition signal, the FSM is duplicated again, upon the activation of Pen-, along with the acknowledge signal Ai- from the PCC.

The diagrammatic waveforms of a D-type output port are shown in Figure 33.
Poll-Type Input-Port Controller

Poll-type controllers differ from demand-type in the way they influence the clock generation process. After being activated by a transition on the Pen signal, a P-port polls the handshake lines it is attached to. If its communicating partner reacts, Ri is set only during the processing of the handshake cycle. Thus in most cases, the local clock is not affected at all. The I/O ports of the poll-type input port controller are shown in Figure 34.
The extended burst-mode specification for the p-type input port controller is shown in Figure 35.

- As it can be seen, the main difference between D-type and P-type is the need for the occurrence of an event on the handshake lines, i.e. an event on signal \( Rp \). Only when both \( Rp \) and \( Pen \) are activated, will there be an event on clock control signal \( Ri \).
- A transition from stage ‘1’ to ‘2’ occurs when the clock acknowledge signal \( Ai \) is set. This will lead to the setting of signal \( Ap \).
- The setting of acknowledge signal \( Ap \) causes data transfer to be synchronized, after which, the output port resets signal \( Rp \). This causes the transition from stage ‘2’ to ‘3’, the output for which is denoted by the resetting of signal \( Ri \) to the PCC and \( Ap \) to the other port.
- Stage transition from ‘3’ to ‘4’ is characterized by the reaction of signal \( Ai^- \) to \( Ri^- \). Also, request signal \( Rp \) is a ‘don’t care’ signal in this stage.
- Just as in the case of a D-type port, stages ‘4’ to ‘8’ are a replication of stages ‘0’ to ‘4’, due to the transition of signal \( Pen \), but along with \( Rp^+ \).
The diagrammatic waveforms of a P-type input port are shown in Figure 36.

Figure 36: Diagrammatic waveforms for Poll-type Input AFSM

**Poll-Type Output-Port Controller**

The I/O ports of the poll-type output port controller are shown in Figure 37.

Figure 37: I/O Ports of Poll-type Output AFSM [1]
Figure 38: XBSM of Poll-type Output AFSM [1]

The extended burst-mode application for the p-type input port controller is shown in Figure 38.

- A transition from stage ‘0’ to ‘1’ occurs when there is an event on input signal \( Pen \). This leads to generation of \( Rp^+ \).
- A transition from stage ‘1’ to ‘2’ occurs when the acknowledging port sets the acknowledge signal \( Ap \), leading to setting of the clock stretch request signal \( Ri \).
- In reaction to the acknowledge \( Ai^+ \) from the PCC, the p-type output port controller resets the request signal \( Rp \). This characterizes the transition in the STG from stage ‘2’ to stage ‘3’.
- Stage transition from ‘3’ to ‘4’ occurs when the acknowledge signal \( Ap \) is reset by the other port. In response to this, the P-type output port controller requests for de-stretching of the clock through \( Ri^- \).
- Stages ‘4’ to ‘8’ are a replication of stages ‘0’ to ‘4’, and the process is triggered due to the transition of signal \( Pen \), but along with \( Ai^- \).

The diagrammatic waveforms of a P-type input port are shown in Figure 39.
Local Clock Generation Circuits

Local clock generation circuits are contained within the asynchronous wrappers. They basically consist of ring oscillators and mutual exclusion elements. Locally synchronous modules within the wrappers can thus utilize local clock signals of frequencies suitable to their needs.

Whenever asynchronous data transfer is to take place at a particular port, the associated port-controller sets the clock-stretch request signal \( R_i \), in response to which the clock generation circuit stretches/pauses the negative cycle of the clock and acknowledges by setting \( A_i \).

Clock generation circuits can either receive stretch-request signals from just one port-controller, or they could receive the ‘request’ signals from more than one port-controller contained within a wrapper.
Single Request Clock Generation Circuit

The clock generation circuit for handling a single clock-stretch request from the port-controller in a Muttersbach wrapper is as shown in Figure 40.

![Figure 40: Single-Request Clock Generation Circuit [1]](image)

This clock generation circuit consists of a ring oscillator and a mutual exclusion element (MUTEX). The ring oscillator generates a continuous clock signal, whose frequency can be adjusted by varying the delay in the feedback loop. The MUTEX is used to arbitrate between the rising-edge of the incoming $R_{clk}$ signal and the rising edge of the clock-stretch request signal $R_i$.

The rising edge that appears first at the input of the MUTEX will go on to pass through the MUTEX to its respective output, and the rising edge on the other signal will pass through only after the first signal has gone low. If both signals appear at the inputs of the MUTEX at the same time, it utilizes a ‘coin-tossing’ strategy to let just one of the rising edges to pass through. Figure 41 shows the internal structure of the MUTEX. The schematic is implemented in Cadence and the simulation results are shown in Figure 42.
The diagrammatic waveform shown in Figure 43 provides us with a better understanding of the working of the single clock-stretch request clock generator.
Cadence simulation results for the single-request clock generator are shown below in Figure 44.
Multi Request Clock Generation Circuit

An asynchronous wrapper can have more than just one port contained within. The port-controllers of each of these ports have the provision to request for clock-stretching whenever required.

In order to facilitate this, the wrapper has to have a local clock generation circuit which can accept multiple clock-stretch requests from all the port-controllers within it. The multi-request clock generation circuit shown in Figure 45, proposed in [1, 2], provides for this feature in the wrapper.

![Multi-Request Clock Generation Circuit](image)

Figure 45: Multi-Request Clock Generation Circuit [1]

This multi-request circuit safely arbitrates between incoming request signals, \( Ri1 - Rin \) and the rising edge of \( Rclk \). It can be seen that even if just one request for clock-stretch is present, the \( Rclk \) signal will not pass through to the output due to the presence of the AND gate.
The diagrammatic waveform given in Figure 46 illustrates the functioning of this multi-request clock generation circuit. Figure 47 provides us with the Cadence simulation results.

Figure 46: Diagrammatic waveforms for Multi-Request Clock Generation Circuit
Chapter Overview

We defined the various port-types used in the Muttersbach wrapper. A thorough discussion of the AFSMs, and an explanation of the functioning of each of the port-controllers were given. These AFSMs are implemented using the VHDL programming language, and the codes are given in Appendix A. In depth understanding of local clock generation circuits for both single- and multi-request stretching was achieved. Cadence simulation results show the functioning of these clock generation circuits.

Proper functioning of a port-controller can be verified only when incorporated in a data transfer channel. Chapter 4 discusses about the various data transfer channel configurations incorporating the port-controllers discussed above. Also, we provide examples to verify the
functioning of the Muttersbach wrapper in a synchronous-synchronous, point-to-point and multi-
point data transfer configurations. An arbiter design is proposed for use in the multi-point data
communication scenario. We also verify the functioning of the wrapper in a synchronous-
asynchronous configuration.
CHAPTER FOUR: GALS COMMUNICATION CHANNEL DESIGNS

In this chapter, we first discuss the various data transfer channel configurations that can be implemented using the D-type and P-type, input and output ports. We first look at the architecture showing a Demand-type Input, Demand-type Output data transfer channel. We will explain the architecture of the data transfer channel, and then the working. A detailed explanation of the flow of control signals and data is presented, as well as simulation results to verify the working. Similarly, we will show the functioning of Poll-type Input, Demand-type Output data transfer channel; Poll-type Input, Poll-type Output data transfer channel; and finally, the Demand-type Input, Poll-type Output data transfer channel.

We will then provide design examples incorporating the various wrapper configurations. We will show how the asynchronous wrapper GALS design technique can be used to realize point-to-point and multi-point communication between synchronous modules; and we will also show how data communication between synchronous-asynchronous modules can be realized. For each of these design examples, we look at the architecture, which is then followed by the description of the architecture and a detailed explanation of the working of the architecture. The working of the architectures is then verified through simulation results.
Asynchronous Wrapper Configurations

Demand-Type Input, Demand-Type Output Data Transfer Channel

To show the functioning of the D-type output and D-type input port wrapper interface, we designed a data transfer channel as shown in Figure 48. In this design, we assume that data is to be transferred from locally synchronous module - 1 (LS1) to locally synchronous module - 2 (LS2). LS1 is encapsulated in a self-time asynchronous wrapper, which is controlled by a D-type output port controller. Similarly, LS2 is encapsulated in another self-time asynchronous wrapper, which is controlled by a D-type input port controller. Handshaking between the ports is achieved through the port-request (Rp) and port-acknowledge (Ap) signals. Also, the latch on the input-port side is enabled by Ap. So, every time there is a positive acknowledgement from the input-port in response to a request from the output-port, the latch is enabled and becomes transparent. Hence, data available at the input of the latch (Data 1) is now transparent and available at the output of the latch as Data 2. The local clock generator circuits (as discussed in the previous chapter) are dictated by the control signals Ri_op and Ai_op for the output-port, and Ri_ip and Ai_ip for the input-port. D-ff’s (Data flip-flop) are used in each wrapper to generate the transfer acknowledge signal Ta, which is used to indicate the completion of transfer to the LS modules. The transfer-acknowledge signal in each wrapper is gated with a locally generated signal(s) to trigger the T-ff (Toggle flip-flop) on the negative edge of the clock, which then generates the port-enable signal Pen. Every time the Pen signal makes an edge transition (either rising or falling), it initiates the port-controller AFSM, and thus begins the hand-shaking process.
Let us now understand the working of the architecture shown in Figure 48.

- To begin with, let us consider that the two different local clock generators (LCGs) generate two unique clocks of different frequencies. Let these clock signals be $Lclk_{op}$ and $Lclk_{ip}$, for the output port and input port, respectively.

- The input to the T-ff in LS1 is set ‘high’ when the Data Available signal becomes active ‘high’, with the $Ta_{op}$ signal being active ‘low’. Thus, on the negative edge of the clock, the output of the T-ff ($Pen_{op}$) toggles, thus triggering the output port-controller AFSM.
• The D-type output port-controller immediately sets the clock stretch request signal \( R_{i\_op} \), and the clock generator circuit responds by setting the acknowledge signal \( A_{i\_op} \). At this point of time, the local clock \( Lclk_{\_op} \) is paused (the negative clock cycle is paused/stretched until data transfer is complete). This then sets the port request signal \( R_{p} \), and the D-type output port waits for a response from the other communicating port.

• In the meanwhile, the input to the T-ff in the receiving LS module becomes active ‘high’ when the Data Accept signal is set by the LS module, with the transfer acknowledge signal \( T_{a\_ip} \) being ‘low’.

• When the input to the T-ff is set, its output (\( P_{en\_ip} \)) toggles on the negative clock edge, thereby triggering the D-type input port-controller AFSM.

• The D-type input port-controller immediately sets the clock stretch request signal \( R_{i\_ip} \), and the clock generator circuit responds by setting the acknowledge signal \( A_{i\_ip} \).

• The local clock \( Lclk_{\_ip} \) is paused (the negative clock cycle is paused/stretched until data transfer is complete). Also, setting of \( A_{i\_ip} \) sets the transfer-acknowledge signal \( T_{a\_ip} \), indicating the beginning of the data transfer cycle.

• At this point of time, if the port request signal \( R_{p} \) is set, then the input-port acknowledges by setting \( A_{p} \) ‘high’. This handshaking process then starts the data transfer, by enabling the latch and making it transparent. Also, a ‘high’ on \( A_{p} \) sets the transfer-acknowledge signal \( T_{a\_op} \) for the output-port.

• In response to the setting of the acknowledge signal \( A_{p} \), the D-type output port-controller AFSM responds by making \( R_{p} \) ‘low’.
• This transition in $Rp$ causes the D-type input port-controller to reset its clock stretch request signal $Ri_{ip}$, in response to which, the clock generator makes $Ai_{ip}$ go ‘low’. At this point of time, clock pausing/stretching is deactivated, and the local clock signal $Lclk_{ip}$ resumes to function normally.

• The input port-controller will then reset port acknowledge signal $Ap$, thus disabling the latch and ending the transfer of data. Also, the transfer-acknowledge signal $Ta_{ip}$ goes ‘low’, indicating an end to the data transfer.

• Finally, the D-type output port-controller responds to the transition in $Ap$ by resetting the clock stretch request signal $Ri_{op}$. The clock generator in LS1 acknowledges this change by making $Ai_{op}$ go ‘low’. This also resets $Ta_{op}$ indicating the end of the data transfer process at the output-port. The clock generator resumes normal operation, and data computation in the locally synchronous modules can now take place.
VHDL codes were written for all of the individual modules using behavioral modeling, and the final circuit implemented using RTL modeling. The simulation results are shown in Figure 50. The flow of control and handshake signals, and data transfer between the modules as explained above can be verified through the simulation results.
Poll-Type Input, Demand-Type Output Data Transfer Channel

To show the functioning of the D-type output and P-type input port wrapper interface, we designed a data transfer channel as shown in Figure 51. LS1 is encapsulated in a self-time asynchronous wrapper, which is controlled by a D-type output port controller. Similarly, LS2 is encapsulated in another self-time asynchronous wrapper, which is controlled by a P-type input port controller. The architecture of this data transfer channel is similar to the architecture in Figure 49. All handshake and control signals perform the same function. Data is assumed to be transferred from LS1 to LS2. The only point to be considered is that unlike in the architecture shown in Figure 48, the LS2 module in Figure 51 consists of a P-type port-controller. A P-type input port-controller stretches the clock when data transfer is possible, but only if there is a request for data transfer from the output-port.
Let us now understand the working of the architecture shown in Figure 51.

- To begin with, let us consider that the two different local clock generators (LCGs) generate two unique clocks of different frequencies. Let these clock signals be $Lclk_{op}$ and $Lclk_{ip}$, for the output port and input port, respectively.

- The input to the T-ff in LS1 is set ‘high’ when the Data Available signal becomes active ‘high’, with the $Ta_{op}$ signal being active ‘low’. Thus, on the negative edge of the clock, the output of the T-ff ($Pen_{op}$) toggles, thus triggering the output port-controller AFSM.
The D-type output port-controller immediately sets the clock stretch request
signal \( Ri_{op} \), and the clock generator circuit responds by setting the acknowledge
signal \( Ai_{op} \). At this point of time, the local clock \( Lclk_{op} \) is paused (the negative
clock cycle is paused/stretched until data transfer is complete). This then sets the
port request signal \( Rp \), and the D-type output port waits for a response from the
other communicating port.

In the meanwhile, the input to the T-ff in the receiving LS module becomes active
‘high’ when the \( Data Accept \) signal is set by the LS module, with the transfer
acknowledge signal \( Ta_{ip} \) being ‘low’.

When the input to the T-ff is set, its output \( (Pen_{ip}) \) toggles on the negative clock
edge. Unlike in the case of a D-type input port-controller where the AFSM would
get triggered by a transition of \( Pen_{ip} \), in the case of the P-type input port-
controller, the AFSM will be triggered if and only if there is a transition on
\( Pen_{ip} \), along with port request signal \( Rp \) being ‘high’.

The P-type input port-controller immediately sets the clock stretch request signal
\( Ri_{ip} \), and the clock generator circuit responds by setting the acknowledge signal
\( Ai_{ip} \).

The local clock \( Lclk_{ip} \) is paused (the negative clock cycle is paused/stretched
until data transfer is complete). Also, setting of \( Ai_{ip} \) sets the transfer-
acknowledge signal \( Ta_{ip} \), indicating the beginning of the data transfer cycle.

At this point of time, the input-port acknowledges by setting \( Ap \) ‘high’. This starts
the data transfer, by enabling the latch and making it transparent. Also, a ‘high’
on \( Ap \) sets the transfer-acknowledge signal \( Ta_{op} \) for the output-port.
• In response to the setting of the acknowledge signal \( Ap \), the D-type output port-controller AFSM responds by making \( Rp \) ‘low’.

• This transition in \( Rp \) causes the P-type input port-controller to reset its clock stretch request signal \( Ri_{ip} \), and also makes \( Ap \) go ‘low’. When \( Ap \) goes ‘low’, the latch is disabled, thus ending the transfer of data. This is also indicated by a ‘low’ on \( Ta_{ip} \).

• In response to a ‘low’ on \( Ri_{ip} \), the clock generator makes \( Ai_{ip} \) go ‘low’. The clock pausing/stretching is deactivated, and the local clock signal \( Lclk_{ip} \) resumes to function normally.

• Finally, the D-type output port-controller responds to the transition in \( Ap \) by resetting the clock stretch request signal \( Ri_{op} \). The clock generator in LS1 acknowledges this change by making \( Ai_{op} \) go ‘low’. This also resets \( Ta_{op} \) indicating the end of the data transfer process at the output-port. The clock generator resumes normal operation, and data computation in the locally synchronous modules can now take place.
VHDL codes were written for all of the individual modules using behavioral modeling, and the final circuit implemented using RTL modeling. The simulation results are shown in Figure 53. The flow of control and handshake signals, and data transfer between the modules as explained above can be verified through the simulation results.
Figure 53: Simulation results for PIP_DOP Data Transfer Channel

Poll-Type Input, Poll-Type Output Data Transfer Channel

To show the functioning of the P-type output and P-type input port wrapper interface, we designed a data transfer channel as shown in Figure 54. LS1 is encapsulated in a self-time asynchronous wrapper, which is controlled by a P-type output port controller. Similarly, LS2 is encapsulated in another self-time asynchronous wrapper, which is controlled by a P-type input port controller. Module LS1 is controlled by a P-type output port-controller, which generates a ‘high’ on the port request signal $R_p$ when there is a transition on port enable signal, $P_{en}$. It also sets the clock stretch request signal $R_i$ ‘high’ only when it receives an acknowledge $A_p$ from the communicating input module LS2. Module LS2 is controlled by a P-type input port-controller. This port-controller stretches the clock when data transfer is possible, but only if there is a request for data transfer from the output-port.
Let us now understand the working of the architecture shown in Figure 54.

- To begin with, let us consider that the two different local clock generators (LCGs) generate two unique clocks of different frequencies. Let these clock signals be $Lclk_{op}$ and $Lclk_{ip}$, for the output port and input port, respectively.

- The input to the T-ff in LS1 is set ‘high’ when the $Data\ Available$ signal becomes active ‘high’, with the $Ta_{op}$ signal being active ‘low’. Thus, on the negative edge of the clock, the output of the T-ff ($Pen_{op}$) toggles, thus triggering the output port-controller AFSM.
• The P-type output port-controller immediately sets the port request signal $Rp$ ‘high’.

• In the meanwhile, the input to the T-ff in the receiving LS module becomes active ‘high’ when the $Data\ Accept$ signal is set by the LS module, with the transfer acknowledge signal $Ta_{ip}$ being ‘low’.

• When the input to the T-ff is set, its output ($Pen_{ip}$) toggles on the negative clock edge. Unlike in the case of a D-type input port-controller where the AFSM would get triggered by a transition of $Pen_{ip}$, in the case of the P-type input port-controller, the AFSM will be triggered if and only if there is a transition on $Pen_{ip}$, along with port request signal $Rp$ being ‘high’.

• The P-type input port-controller immediately sets the clock stretch request signal $Ri_{ip}$, and the clock generator circuit responds by setting the acknowledge signal $Ai_{ip}$.

• The local clock $Lclk_{ip}$ is paused (the negative clock cycle is paused/stretched until data transfer is complete). Also, setting of $Ai_{ip}$ sets the transfer-acknowledge signal $Ta_{ip}$, indicating the beginning of the data transfer cycle.

• At this point of time, the input-port acknowledges by setting $Ap$ ‘high’. This starts the data transfer, by enabling the latch and making it transparent. Also, a ‘high’ on $Ap$ sets the transfer-acknowledge signal $Ta_{op}$ for the output-port.

• In response to the setting of the acknowledge signal $Ap$, the P-type output port-controller AFSM responds by making clock stretch request signal $Ri_{op}$ ‘high’.

• The LCG responds by acknowledging with a ‘high’ on $Ai_{op}$. This causes the output port-controller to reset the port request signal $Rp$. 
• This transition in $Rp$ causes the P-type input port-controller to reset its clock stretch request signal $Ri_ip$, and also makes $Ap$ go ‘low’. When $Ap$ goes ‘low’, the latch is disabled, thus ending the transfer of data. This is also indicated by a ‘low’ on $Ta_ip$.

• In response to a ‘low’ on $Ri_ip$, the clock generator makes $Ai_ip$ go ‘low’. The clock pausing/stretching is deactivated, and the local clock signal $Lclk_ip$ resumes to function normally.

• Finally, the P-type output port-controller responds to the transition in $Ap$ by resetting the clock stretch request signal $Ri_op$. The clock generator in LS1 acknowledges this change by making $Ai_op$ go ‘low’. This also resets $Ta_op$ indicating the end of the data transfer process at the output-port. The clock generator resumes normal operation, and data computation in the locally synchronous modules can now take place.
VHDL codes were written for all of the individual modules using behavioral modeling, and the final circuit implemented using RTL modeling. The simulation results are shown in Figure 56. The flow of control and handshake signals, and data transfer between the modules as explained above can be verified through the simulation results.
To show the functioning of the P-type output and D-type input port wrapper interface, we designed a data transfer channel as shown in Figure 57. LS1 is encapsulated in a self-time asynchronous wrapper, which is controlled by a P-type output port controller. Similarly, LS2 is encapsulated in another self-time asynchronous wrapper, which is controlled by a D-type input port controller. Module LS1 is controlled by a P-type output port-controller, which generates a ‘high’ on the port request signal $R_p$ when there is a transition on port enable signal, $Pen$. It also sets the clock stretch request signal $R_i$ ‘high’ only when it receives an acknowledge $A_p$ from the communicating input module LS2. Module LS2 is controlled by a D-type input port-controller. This port-controller stretches the clock immediately upon receiving a transition on the port enable signal $Pen$, irrespective of the presence of the port request signal $R_p$. 
Let us now understand the working of the architecture shown in Figure 57.

- To begin with, let us consider that the two different local clock generators (LCGs) generate two unique clocks of different frequencies. Let these clock signals be $L_{clk_{op}}$ and $L_{clk_{ip}}$, for the output port and input port, respectively.
- The input to the T-ff in LS1 is set ‘high’ when the Data Available signal becomes active ‘high’, with the $Ta_{op}$ signal being active ‘low’. Thus, on the negative edge of the clock, the output of the T-ff ($Pen_{op}$) toggles, thus triggering the output port-controller AFSM.
• The P-type output port-controller immediately sets the port request signal \( Rp \) ‘high’.

• In the meanwhile, the input to the T-ff in the receiving LS module becomes active ‘high’ when the Data Accept signal is set by the LS module, with the transfer acknowledge signal \( Ta_{ip} \) being ‘low’.

• When the input to the T-ff is set, its output \( Pen_{ip} \) toggles on the negative clock edge. The D-type input port-controller immediately sets the clock stretch request signal \( Ri_{ip} \), and the clock generator circuit responds by setting the acknowledge signal \( Ai_{ip} \). With \( Ai_{ip} \) ‘high’, and if \( Rp \) is ‘high’, the port-controller generates a ‘high’ on \( Ap \). This starts the data transfer, by enabling the latch and making it transparent.

• The local clock \( Lclk_{ip} \) is paused (the negative clock cycle is paused/stretched until data transfer is complete). Also, setting of \( Ai_{ip} \) sets the transfer-acknowledge signal \( Ta_{ip} \), indicating the beginning of the data transfer cycle.

• In response to the setting of the acknowledge signal \( Ap \), the P-type output port-controller AFSM responds by making clock stretch request signal \( Ri_{op} \) ‘high’. Also, a ‘high’ on \( Ap \) sets the transfer-acknowledge signal \( Ta_{op} \) for the output-port

• The LCG responds by acknowledging with a ‘high’ on \( Ai_{op} \). This causes the output port-controller to reset the port request signal \( Rp \).

• This transition in \( Rp \) causes the D-type input port-controller to reset its clock stretch request signal \( Ri_{ip} \).
• In response to a ‘low’ on $Ri_ip$, the clock generator makes $Ai_ip$ go ‘low’. This makes $Ap$ go ‘low’. When $Ap$ goes ‘low’, the latch is disabled, thus ending the transfer of data. This is also indicated by a ‘low’ on $Ta_ip$.

• The clock pausing/stretching is deactivated, and the local clock signal $Lclk_ip$ resumes to function normally.

• Finally, the P-type output port-controller responds to the transition in $Ap$ by resetting the clock stretch request signal $Ri_op$. The clock generator in LS1 acknowledges this change by making $Ai_op$ go ‘low’. This also resets $Ta_op$ indicating the end of the data transfer process at the output-port. The clock generator resumes normal operation, and data computation in the locally synchronous modules can now take place.

Figure 58: Diagrammatic waveforms for DIP_POP Data Transfer Channel
VHDL codes were written for all of the individual modules using behavioral modeling, and the final circuit implemented using RTL modeling. The simulation results are shown in Figure 59. The flow of control and handshake signals, and data transfer between the modules as explained above can be verified through the simulation results.

Figure 59: Simulation results for DIP_POP Data Transfer Channel

**Design Examples Incorporating Various Wrapper Configurations**

**Synchronous – Synchronous, Point-to-Point Data Communication**

Data communication between modules is said to be point-to-point, when a port of one LS module interacts with just one other port of another LS module. LS modules can be simply classified into three classes in terms of data flow: source LS modules with only output ports, sink LS module with only input ports, and intermediate modules having both input and output ports. A typical GALS sub-system with point-to-point data communication is as shown in Figure 60.
In our first design example, shown in Figure 61, we consider two synchronous modules, LS1 and LS2. Each module is locally clocked at different frequencies. This example shows the functioning of asynchronous wrappers in a synchronous-synchronous, point-to-point data-transfer channel configuration.

For the sake of simplicity, we have considered to design a sub-system consisting of two LS modules. One module consists of a ROM (Read-only-Memory); and the other LS module generates the control- and address-signals to the ROM, and also accepts the output data from the ROM. We have two data-transfer channels. One channel consisting of a D-type output port in LS1, and a D-type input port in LS2. This channel is used to transfer the address- and control-signals from LS1 to LS2. The second data-transfer channel consists of a D-type input port in LS1 and a D-type output port in LS2. This channel is used to transfer the data that is output from the ROM (LS2) to the synchronous module LS1. Each of the output ports have a latch which is used to latch the data being transferred over the channel. These latches become transparent when the port-acknowledge signals \((Ap)\) go ‘high’.
LS1 consists of two data flip-flops (D-ff). One of the D-ffs is used for input data (address signals + control signals), and the other is used for the output data (data from ROM). This module also has two toggle flip-flops (T-ff). The T-ffs are used to generate the port enable signals $Pen_{1\_op}$ and $Pen_{1\_ip}$, to the output and input ports, respectively. The inputs to the T-ffs are the ‘enable’ and ‘read’ control signals. Thus, during every instance when the ‘read’ and ‘enable’ control signals are ‘high’ at the negative edge of the clock, the T-ffs toggle the port enable signals, thereby triggering the port-controller AFSMs in LS1.
LS2 consists of two T-ffs, both to generate the port enable signals $Pen2_{ip}$ and $Pen2_{op}$. LS2 also consists of a D-ff, the input to which is the AND of the ‘read’ and ‘enable’ control signals, and its output is used to toggle the T-ff. The T-ff used to generate $Pen2_{ip}$ always has its input set to ‘1’. Hence, irrespective of whether the ROM is enabled and being read from or not, the D-type input port in LS2 will keep the local clock stretched.

The local clock generators in both the modules receive clock stretch request signals $Ri$, from two ports. Hence, we use a modified version of the multi-request clock generator as described in Chapter 3. The LCGs are modified to accept clock stretch request signals $Ri$ from two ports within the same synchronous module, and will generate corresponding acknowledge signals ($Ai$) to the $Ri$ signals.

Let us understand the proper working of the above architecture by looking at the step-by-step flow of control, handshake, and data signals through the architecture.

- To begin with, let us consider that the two different local clock generators (LCGs) generate two unique clocks of different frequencies.
- The input to the T-ff in LS1 which generates $Pen1_{op}$ is AND of the control signals ‘read’ and ‘enable’. The T-ff toggles when the input is ‘1’ on the negative clock edge.
- An event on $Pen1_{op}$ triggers the AFSM of the D-type output port-controller associated with LS1. This stretches the local clock by making $Ri1_{op}$ ‘high’, in response to which, $Ai1_{op}$ also become ‘high’. The port-controller also generates a ‘high’ on the port request signal $Rp$.
- The T-ff that generates $Pen2_{ip}$ for the input port-controller in LS2 is always set to ‘1’. Hence at every negative clock edge, the output of this flip-flop toggles,
causing an event on $Pen2_{ip}$. This triggers the AFSM for the D-type input port-controller associated with LS2.

- The clock stretch request signal $Ri2_{ip}$ is set ‘high’, in response to which the clock generator makes $Ai2_{ip}$ ‘high’. This causes the stretching of the local clock (pausing of the negative clock cycle). Also, with $Rp$ ‘high’, the port-controller acknowledges by setting $Ap$. This causes the latch to be enabled, and the data $(Address + Control\ Signals – 1)$ available at the input of the latch is now available over the channel $(Address + Control\ Signals – 2)$.

- The handshake process is completed by resetting the clock control signals $Ri1_{op}$, $Ai1_{op}$, $Ri2_{ip}$, and $Ai2_{ip}$. Also, the handshake signals $Rp$ and $Ap$ are reset, thus indicating the end of the transfer of address and control signals over the channel.

- The local clock generator in LS1 however still keeps the clock stretched because the D-type input port associated with LS1 had also request for a clock stretch ($Ri1_{ip}$), and is waiting for the D-type output port of LS2 to begin the handshake process.

- Now that the clock associated with LS2 resumes to work normally, the ROM is enabled for a ‘read’ operation on the positive edge of the clock. At the same time, the control signals are ANDed and fed to the D-ff, the output of which is delayed by one clock cycle. Thus, by the time data output is read from the ROM (1 positive clock cycle), an event occurs on $Pen2_{op}$ on the negative edge of the clock. This then triggers the D-type output port of LS2.
• Now, $Ri2_{\text{op}}$ is made ‘high’, in response to which the LCG makes $Ai2_{\text{op}}$ ‘high’.

Thus the clock of LS2 is paused. Also, the port-controller generates $Rp$ (makes it ‘high’).

• The communicating input port in LS1, responds to a ‘high’ on $Rp$ by setting the acknowledge signal $Ap$. This causes the latch to be enabled and become transparent to the data read from the ROM.

• The transfer of data ends with the handshake signals being reset, along with the clock control signals.

• Thus, both the clocks are now de-stretched and resume functioning, normally.

• It can be seen that the clock in LS1 remains stretched until data has been read from the ROM and is available at the output port of LS2. On the other hand, in the same period of time, the clock in LS1 gets stretched twice, once during the transfer of address and control signals, and once more during the transfer of the output data.

VHDL codes were written for all of the individual modules using behavioral modeling, and the final circuit implemented using RTL modeling. The simulation results are shown in Figure 62. The flow of control and handshake signals, and data transfer between the modules as explained above can be verified through the simulation results.
The above example incorporates only D-type port-controllers. We can substitute D-type controllers with P-type port-controllers. The example provided is hypothetical in nature. But the same techniques can be applied in a real system to realize GALS communication between modules. In this example we considered only point-to-point communication between ports. The GALS design technique can also be applied to realize communication between a multi-output port and various single-input ports, or between a multi-input port and various single-output ports. An example of this is shown in the next section.
Synchronous – Synchronous, Multi-Point Data Communication

Data communication between modules is said to be of type multi-point whenever a port of one LS module (multi-input or multi-output port) interacts with more than one other port. There are two possible multi-point configurations: either a multi-output port driving various input-ports, or various output ports driving a multi-input port, as shown in Figure 63.

Figure 63: Multi-Point Data Communication Model

Realizing multi-point communication is essential in the following scenarios:

- A multi-input port is used in an LS module where simultaneous computations are to be performed on data arriving from various output ports.
- A multi-output port is used whenever data is to be sent out to more than one input-port.
Basic Multi-Point Design Approach

In our second design example, shown in Figure 64, we consider five synchronous modules, LS1, LS2, LS3, LS4, and LS5. Each module is locally clocked at different frequencies. This example shows the functioning of asynchronous wrappers in a synchronous-synchronous, multi-point data-transfer channel configuration.

Figure 64: Synchronous-Synchronous, Multi-Point Data Transfer Channel Example
For the sake of simplicity, we have considered a hypothetical sub-system consisting of five LS modules. LS1 being the module that generates the Address and Control signals; LS2 being the module that provides the data (Data_In) to be written into the RAM; LS5 consisting of the RAM; and finally, LS3 and LS4 being the LS units that read out the data from the RAM, Data_Out_1 and Data_out_2, respectively. Other than LS5, which consist of a 2-port LCG, all other units have single-port LCGs. There are 4 latches in the sub-system, through which the 4 separate data-paths flow. Units LS1 (D-type output port), LS2 (D-type output port), LS3 (D-type input port), and LS4 (P-type input port) consist of ports that perform point-to-point communication; whereas unit LS5 consists of two ports: a D-type input port that accepts multiple inputs from various units, and a D-type output port that outputs to multiple units, simultaneously.

To provide for proper synchronization of the handshake signals between the multi-input port and the single-output ports, and the multi-output port and single-input ports, we use AND gates [10]. To properly synchronize the reception of the control and address signals, and Data_In by the D-type multi-input port, we AND the port request signals Rp_1 and Rp_2. Thus, only when both Rp_1 and Rp_2 are ‘1’, will Rp_RAM_ip be ‘1’. The acknowledge signal Ap_RAM_ip will be passed onto the output ports of both LS1 and LS2, thus enabling both the latches, as shown in Figure 64. This causes the latches to become transparent, hence making the control and address signals, and Data_In available to LS5.

Assuming that the data output from the RAM is to be simultaneously made available to both LS3 and LS4, we have to use another AND gate to synchronize the ‘acknowledge’ signals Ap_3 and Ap_4. The D-type multi-output port of LS5 sends out the port request signal Rp_RAM_op to both LS3 and LS4 as Rp_3 and Rp_4, respectively. In response to these request
signals, modules LS3 and LS4 generate Ap_3 and Ap_4, which are then ANDed to generate Ap_RAM_op. This signal is then used to enable the two latches, and thus make the latches transparent to the output data.

The port-controller AFSMs for LS2, LS3, and LS4 are enabled during every negative clock edge, since the inputs to the T-ffs are tied to ‘1’. The port-controller AFSM in LS1 is controlled by the enable signal. There is an event on port enable signal to the multi-input port of LS5 during the negative edge of every clock cycle. However, the port enable signal for the multi-output port of LS5 is triggered only if the read signal is enabled.

VHDL codes were written for all of the individual modules using behavioral modeling, and the final circuit implemented using RTL modeling. The simulation results are shown in Figure 65. The flow of control and handshake signals, and data transfer between the modules as explained above can be verified through the simulation results.
Figure 65: Simulation results for Multi-Point Data Transfer Channel Example
A disadvantage with the above multi-point design approach, where AND gates are used to synchronize the handshake signals between modules is that there is no flexibility to adapt to independent handshaking between the modules. For instance, in the above setup, request signals from separate output ports to the multi-input port are ANDed. Hence, irrespective of anything, both of the LS modules that contain the output ports are interdependent. That is, the duration for which the clocks are stretched in these modules is dependent on the slower module. Thus, even if LS1 and LS5 are ready to begin and end handshaking, they will have to wait until LS2 also sends out a request to begin handshaking. Hence, LS1 is slowed down to the clock speed of LS2.

In order to overcome this limitation, we have designed an arbiter that individually arbitrates between the two request-signals going into the multi-input port, with both the output-ports being able to achieve handshaking within the same clock cycle of the multi-input port.

**Arbiter for Multi-Point Design Approach**

To overcome the limitation mentioned above, we designed an arbiter that lets one of the two request signals from the individual output ports to pass through to the multi-input port at any given time, and also makes sure that both of the output ports achieve the required handshaking within the same clock cycle of the module containing the multi-input port. The designed arbiter is shown in Figure 66
Figure 66: Arbiter for Multi-Port Data Communication

At the region AA’-BB’, the arbiter contains Muller C-elements that prevent more than one ‘HIGH’ on the individual request signals R1 and R2 from passing through the arbiter to the multi-input port during a single handshake process. It ensures that during a handshake process, just one request from each of the output ports pass on to the multi-input port. Request signals R1 and R2 are inputs in this region, and the acknowledge signals A1 and A2 are outputs in this region.

The arbiter has a MUTEX (Mutual Exclusion element) in the region BB’-CC’, which allows just one of the two request signals, R1 and R2, to go through at any given time. Also, if both of the requests arrive simultaneously, the MUTEX utilizes a ‘coin-toss’ strategy to let just one request signal to pass through. The NAND gates in the region BB’-CC’ are used to gate the request signals that are output from the MUTEX with the acknowledge signals - R1 gated with A2, and R2 gated with A1.

Muller C-elements are used in region CC’-DD’ to limit the number of requests passing through the arbiter from the same output port in a single clock cycle of the multi-input port to
just one. Also, AND gates are used in this region to drive the acknowledge signals ‘LOW’ even when only the request signals go ‘LOW’. The acknowledge signals however go ‘HIGH’ only if both the input signals to the AND gates are ‘HIGH’.

The Muller C-element in region DD’-EE’ is used to control the 3-input C-elements in region EE’-FF’. This C-element controls the order of the requests going through to the output of the arbiter. First, a ‘HIGH’ on one of the request signals will pass through, then it allows the ‘HIGH’ on the second request signal to pass through, after which it allows the ‘LOWs’ on the first and second request signals to pass through.

The 3-input C-elements in region EE’-FF’ are used to further control the flow of the request signals, making sure that ‘LOWs’ on the request signal pass through only after the ‘HIGHs’ on both the request signals have passed through. The 2-input C-elements are used to control the ‘HIGH’ phase of the acknowledge signal A1 and A2.

Finally, an OR gate is used in region FF’-GG’ to let the request signals to pass through to the multi-input port as R0. The acknowledge signal from the multi-input port A0 is an input in this region.

The diagrammatic waveforms given below in Figure 67 show the functioning of the arbiter. It can be clearly verified that R0 (request signal going into the multi-input port from the arbiter) will remain ‘HIGH’ until both the output ports have sent out requests on R1 and R2. Also, we can notice in the waveform that though Req1-C comes before Req2-B at the input of the arbiter, it is not allowed to go through until ReqB at the output of the arbiter goes ‘LOW’.
Cadence Schematic Editor was used to create the schematic for this arbiter. The simulation results for the arbiter are shown in Figure 68.
GALS Data Communication in a Synchronous-Asynchronous Interface

So far, we have seen GALS data communication between locally synchronous modules. Let us now consider the case of synchronous-asynchronous interface based GALS communication. In our example, shown in Figure 69, we consider a locally synchronous module that sends address and control signals to an asynchronous ROM, and then receives the output that is read out of the asynchronous ROM.

Figure 69: Synchronous-Asynchronous Data Transfer Channel Example

For our example, only a D-type output port-controller can be used. The D-type port-controller stretches the local clock immediately, to access the ROM for the period of the 'ROM-
read’ action. If we substitute the D-type with a P-type port controller, then the local clock will get stretched only after the ‘ROM-read’ action, which is redundant. Hence, by using D-type port-controllers in such scenarios, we utilize the actual performance of memories, and only those clock cycles requiring memory access are adapted to memory speed.

In Figure 69, the synchronous module consists of a D-type output-port; a local clock generator; a T-ff to generate the port enable signal \( Pen_{Dop} \); two D-ffs to register the input address and control signals, and the output \( Data_{Out} \). Also, two latches are used, one for the address and control signals, the other for the output data. The latches are enabled by \( Rp \) and \( Ap \), respectively. Unlike in previous scenarios (synchronous-synchronous communication) where the latches used to be enabled only by the port acknowledge signal \( Ap \), in this synchronous-asynchronous scenario the latches are individually enabled by \( Rp \) and \( Ap \).

Every time the LS module wants to access the asynchronous ROM (dependent on the control signals), the output port-controller AFSM is triggered. Since it is of demand type, the output port controller immediately sets the clock stretch request signal ‘high’. In response to this, the clock generator acknowledges with a ‘high’ on \( Ai \). This causes the port-controller to set the port request signal \( Rp \). This enables the address latch, making it transparent to the address that is now available to the asynchronous ROM.

The availability of a new address and a ‘high’ on \( Rp \) at the inputs of the asynchronous ROM will cause a ‘ROM-read’ action to occur, and the data stored at that particular address will be available at the output of the asynchronous ROM. When the data is available at its output, the ROM will generate a ‘high’ on the acknowledge signal \( Ap \). This then enables the data latch, thus causing it to become transparent to \( Data_{Out} \).
When the port-controller detects a ‘high’ on \( Ap \), it resets \( Rp \). The asynchronous ROM responds by resetting \( Ap \). This then leads to the deactivation of the clock stretching, and the LS module now resumes normal operation. Thus, it can be seen that the locally synchronous clock is stretched only for the period of the ROM data access, making efficient use of the memory’s performance.

The VHDL codes for the above architecture are shown in appendix [], and the simulation waveforms shown in Figure 70 verify the functioning of the synchronous-asynchronous data transfer channel.

![Figure 70: Simulation results for Synchronous-Asynchronous Data Transfer Channel](image-url)
CHAPTER FIVE: CONCLUSION AND FUTURE WORK

Conclusion

While much remains to be learned with regards to performance and efficiency issues of the various GALS design approaches, the discussions provided and techniques developed herein provide a basis for better understanding the concept of the GALS design methodology.

This has been achieved by first realizing the need to pursue the GALS design methodology to design VLSI systems. Based on discussions of the various design issues that are to be considered while implementing the GALS design methodology, we were able to understand the motivation behind the different GALS design approaches. The ‘asynchronous FIFO’ approach is pursued when meta-stability avoidance is the major factor. On the other hand, the ‘asynchronous wrapper’ approach is pursued when both meta-stability avoidance and latency avoidance are of importance. With GALS being in its nascent stage, it is too early to take sides while choosing a particular design approach. It may eventually be possible to utilize all the different GALS design approaches while designing VLSI systems.

The Muttersbach design approach was chosen to study and validate the concept of GALS primarily because of the various possible wrapper configurations that could be implemented in realizing data communication between units within a system. We were able to realize all wrapper configurations and provide simulation results to verify their functioning.

Design examples to further understand the working of the various wrapper configurations and the concept of mixed-clock systems were realized. We are able to achieve point-to-point and multi-point data communication between synchronous-synchronous and synchronous-
asynchronous modules. Though the examples considered are hypothetical, the achieved results are valid enough to prove the concept of GALS.

**Future Work**

The main motivation for adapting GALS design techniques is to achieve reduced power dissipation due to global clock distribution. A few researchers have shown that though there is a slight drop in power dissipation by using GALS design approaches in realizing VLSI systems, there is also a drop in performance. It is therefore required to further research on the efficiency of the various GALS design approaches based on issues such as performance and power consumption.

Muttersbach *et al.* used a 3-D synthesis tool to synthesize the XBMSs and realize two-level AND-OR implementations for all the port-controllers. With the availability of this tool, one can easily realize the asynchronous wrapper port-controllers at the gate level using CAD tools, thus making it possible to better understand various performance issues.

The arbiter design proposed in this thesis is applicable only where a multi-input port is of concern. An interesting topic to research on would be the utilization of this arbiter in a scenario where the data paths are tri-state buses. Also, with further research, the need to realize more flexible approaches in implementing multi-point data communication at the multi-output port may arise.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- Begin entity rom_dip_dop
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
entity rom_dip_dop is
port (------ I/P Port Signals --------------
    address : in std_logic_vector (1 downto 0);
    enable  : in std_logic;
    read    : in std_logic;
    reset   : in std_logic;
------ O/P Port Signals --------------
    clk_ctrl_fin        : out std_logic;
    pen_op_add_ctrl_fin : out std_logic;
    ri_op_add_ctrl_fin  : out std_logic;
    ai_op_add_ctrl_fin  : out std_logic;
    rp_add_fin   : out std_logic;
    ap_add_fin   : out std_logic;
    latch_in_add_fin  : out std_logic_vector (3 downto 0);
    latch_out_add_fin : out std_logic_vector (3 downto 0);
    pen_ip_add_rom_fin  : out std_logic;
    ri_ip_add_rom_fin  : out std_logic;
    ai_ip_add_rom_fin  : out std_logic;
    clk_rom_fin       : out std_logic;
    pen_op_data_rom_fin : out std_logic;
    ri_op_data_rom_fin : out std_logic;
    ai_op_data_rom_fin : out std_logic;
    rp_data_fin   : out std_logic;
    ap_data_fin   : out std_logic;
    latch_in_data_fin: out std_logic_vector (3 downto 0);
    latch_out_data_fin: out std_logic_vector (3 downto 0);
    pen_ip_data_ctrl_fin : out std_logic;
    ri_ip_data_ctrl_fin : out std_logic;
    ai_ip_data_ctrl_fin : out std_logic;
    data_out       : out std_logic_vector (3 downto 0)
    );
end rom_dip_dop;
architecture rtl of rom_dip_dop is

-- Component Declarations

component rom
port (clock      : in std_logic;
reset     : in std_logic;
enable  : in std_logic;
read     : in std_logic;
address : in std_logic_vector(1 downto 0);
data_out  : out std_logic_vector(3 downto 0)
); end component;

component d_ip
port (reset   : IN std_logic;
pen     : IN std_logic;
rp      : IN std_logic;
ai      : IN std_logic;
ri      : OUT std_logic;
ap      : OUT std_logic
); end component;

component d_op
PORT (reset   : IN std_logic;
pen     : IN std_logic;
ap      : IN std_logic;
ai      : IN std_logic;
ri      : OUT std_logic;
rp      : OUT std_logic
); end component;

COMPONENT clk_gen_2port
PORT( reset : IN std_logic;
sel : IN std_logic;
ri1 : IN std_logic;
ri2 : IN std_logic;
);
component t_ff
port (
reset     : in std_logic;
t_in      : in std_logic;
neg_clk  : in std_logic;
t_out     : out std_logic
);
end component;

component data_latch
port (reset     : in std_logic;
data_in   : in std_logic_vector(3 downto 0);
enable    : in std_logic;
data_out  : out std_logic_vector (3 downto 0)
);
end component;

component d_ff_1
port (reset     : in std_logic;
d_in      : in std_logic;
clk        : in std_logic;
d_out     : out std_logic
);
end component;

component d_ff_4
port (reset     : in std_logic;
d_in      : in std_logic_vector (3 downto 0);
clk        : in std_logic;
d_out     : out std_logic_vector (3 downto 0)
);
end component;

-- -- -- -- -- -- -- --
-- Signal declarations
-- -- -- -- -- -- -- --
signal enable_s, read_s       : std_logic;
signal rom_address_s : std_logic_vector (1 downto 0);
signal latch_in_add_s, latch_out_add_s : std_logic_vector (3 downto 0);
signal rom_data_s, latch_in_data_s, latch_out_data_s : std_logic_vector (3 downto 0);
signal d_ff_in_data_ctrl_s, d_ff_out_data_ctrl_s : std_logic_vector (3 downto 0);
signal d_ff_in_add_ctrl_s, d_ff_out_add_ctrl_s : std_logic_vector (3 downto 0);
signal d_ff_in_read_en, d_ff_in_1, d_ff_out_1 : std_logic;
signal t_ff_in_dop_data_rom_s, t_ff_ctrl_ip_s : std_logic;
signal t_ff_in_dop_add_ctrl_s, t_ff_in_dip_data_ctrl_s : std_logic;
signal t_ff_in_dip_add_rom_s : std_logic := '1';
signal t_ff_out_dip_add_rom_s, t_ff_out_dop_add_ctrl_s : std_logic;
signal t_ff_out_dip_data_ctrl_s, t_ff_out_dop_data_rom_s : std_logic;
signal pen_ip_add_rom_s, pen_op_add_ctrl_s : std_logic;
signal pen_ip_data_ctrl_s, pen_op_data_rom_s : std_logic;
signal lclk_rom_s, lclk_ctrl_s : std_logic;
signal ap_add_s, rp_add_s, ap_data_s, rp_data_s : std_logic;
signal ai_op_add_ctrl_s, ri_op_add_ctrl_s : std_logic;
signal ai_op_data_rom_s, ri_op_data_rom_s : std_logic;
signal ai_ip_add_ctrl_s, ri_ip_add_ctrl_s : std_logic;
signal ai_ip_data_ctrl_s, ri_ip_data_ctrl_s : std_logic;
signal ctrl_sel : std_logic := '0';
signal rom_sel : std_logic := '1';

-- Component port mapping

begin

uut_rom : rom
port map (  
clock => lclk_rom_s,  
reset => reset,  
enable => enable_s,  
read => read_s,  
address => rom_address_s,  
data_out => rom_data_s  );

uut_dop_add_ctrl : d_op
port map (  
reset => reset,  
pen => pen_op_add_ctrl_s,  
ap => ap_add_s,  
ai => ai_op_add_ctrl_s,  
ri => ri_op_add_ctrl_s,  
rp => rp_add_s  );
uut_dip_add_rom : d_ip
port map (reset => reset,
         pen => pen_ip_add_rom_s,
         rp => rp_add_s,
         ai => ai_ip_add_rom_s,
         ri => ri_ip_add_rom_s,
         ap => ap_add_s);

uut_dop_data_rom : d_op
port map (reset => reset,
         pen => pen_op_data_rom_s,
         ap => ap_data_s,
         ai => ai_op_data_rom_s,
         ri => ri_op_data_rom_s,
         rp => rp_data_s);

uut_dip_data_ctrl : d_ip
port map (reset => reset,
         pen => pen_ip_data_ctrl_s,
         rp => rp_data_s,
         ai => ai_ip_data_ctrl_s,
         ri => ri_ip_data_ctrl_s,
         ap => ap_data_s);

uut_clk_ctrl : clk_gen_2port
port map (reset => reset,
         sel => ctrl_sel,
         ri1 => ri_op_add_ctrl_s,
         ri2 => ri_ip_data_ctrl_s,
         ai1 => ai_op_add_ctrl_s,
         ai2 => ai_ip_data_ctrl_s,
         lclk => lclk_ctrl_s);

uut_clk_rom : clk_gen_2port
port map (reset => reset,
         sel => rom_sel,
         ri1 => ri_ip_add_rom_s,
uut_t_ff_dop_add_ctrl : t_ff
port map (reset => reset,
          t_in => t_ff_in_dop_add_ctrl_s,
          neg_clk => lclk_ctrl_s,
          t_out => t_ff_out_dop_add_ctrl_s);

uut_t_ff_dip_add_rom : t_ff
port map (reset => reset,
          t_in => t_ff_in_dip_add_rom_s,
          neg_clk => lclk_rom_s,
          t_out => t_ff_out_dip_add_rom_s);

uut_t_ff_dop_data_rom : t_ff
port map (reset => reset,
          t_in => t_ff_in_dop_data_rom_s,
          neg_clk => lclk_rom_s,
          t_out => t_ff_out_dop_data_rom_s);

uut_t_ff_dip_data_ctrl : t_ff
port map (reset => reset,
          t_in => t_ff_in_dip_data_ctrl_s,
          neg_clk => lclk_ctrl_s,
          t_out => t_ff_out_dip_data_ctrl_s);

uut_data_latch_add : data_latch
port map (reset => reset,
          data_in => latch_in_add_s,
          enable => ap_add_s,
          data_out => latch_out_add_s)
uut_data_latch_data : data_latch
port map (
reset => reset,
data_in => latch_in_data_s,
enable => ap_data_s,
data_out => latch_out_data_s
);

uut_rom_read : d_ff_1
port map (
reset => reset,
d_in => d_ff_in_1,
clk => lclk_rom_s,
d_out => d_ff_out_1
);

uut_ip_add_ctrl : d_ff_4
port map (
reset => reset,
d_in => d_ff_in_add_ctrl_s,
clk => lclk_ctrl_s,
d_out => d_ff_out_add_ctrl_s
);

uut_op_data_ctrl : d_ff_4
port map (
reset => reset,
d_in => d_ff_in_data_ctrl_s,
clk => lclk_ctrl_s,
d_out => d_ff_out_data_ctrl_s
);

-- concurrent signal assignment statements

enable_s <= latch_out_add_s(0);
read_s <= latch_out_add_s(1);
rom_address_s <= latch_out_add_s(3 downto 2);
latch_in_add_s <= d_ff_out_add_ctrl_s;
latch_in_data_s <= rom_data_s;
d_ff_in_data_ctrl_s <= latch_out_data_s;
data_out <= d_ff_out_data_ctrl_s;
d_ff_in_add_ctrl_s(3 downto 2) <= address;
d_ff_in_add_ctrl_s(1) <= read;
d_ff_in_add_ctrl_s(0) <= enable;
d_ff_in_read_en <= latch_out_add_s(0) and latch_out_add_s(1);
d_ff_in_1 <= d_ff_in_read_en;
t_ff_in_dop_data_rom_s <= d_ff_out_1;
t_ff_ctrl_ip_s <= d_ff_out_add_ctrl_s(0) and d_ff_out_add_ctrl_s(1);
t_ff_in_dop_add_ctrl_s <= t_ff_ctrl_ip_s;
t_ff_in_dip_data_ctrl_s <= t_ff_ctrl_ip_s;
pen_ip_add_rom_s <= t_ff_out_dip_add_rom_s;
pen_op_data_rom_s <= t_ff_out_dop_data_rom_s;
pen_op_add_ctrl_s <= t_ff_out_dip_data_ctrl_s;
pen_ip_data_ctrl_s <= pen_ip_add_rom_s;
pen_ip_data_ctrl_fin <= pen_ip_data_ctrl_s;
pen_op_data_rom_fin <= pen_op_data_rom_s;
ap_data_fin <= ap_data_s;
rp_data_fin <= rp_data_s;
ap_add_fin <= ap_add_s;
rp_add_fin <= rp_add_s;
ri_op_add_ctrl_fin <= ri_op_add_ctrl_s;
ai_op_add_ctrl_fin <= ai_op_add_ctrl_s;
ri_ip_add_rom_fin <= ri_ip_add_rom_s;
ai_ip_add_rom_fin <= ai_ip_add_rom_s;
ri_op_data_rom_fin <= ri_op_data_rom_s;
ai_op_data_rom_fin <= ai_op_data_rom_s;
clk_ctrl_fin <= lclk_ctrl_s;
clk_rom_fin <= lclk_rom_s;
latch_in_add_fin <= latch_in_add_s;
latch_out_add_fin <= latch_out_add_s;
latch_in_data_fin <= latch_in_data_s;
latch_out_data_fin <= latch_out_data_s;

end rtl;
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- 
-- End architecture of entity rom_dip_dop
-----------------------------------------------------------

Synchronous-Synchronous Multi-Point Example [RTL – multi_port_ram]

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- Begin Entity multi_port_ram

entity multi_port_ram is
port (             
------ I/P Port Signals ------
reset                    : in std_logic;
enable     : in std_logic;
write     : in std_logic;
read     : in std_logic;
write_addr    : in std_logic_vector (1 downto 0);
read_addr   : in std_logic_vector (1 downto 0);
data_in    : in std_logic_vector (3 downto 0);
             
------ O/P Port Signals ------
clk_add_ctrl   : out std_logic;
pen_add_ctrl   : out std_logic;
ri_add_ctrl    : out std_logic;
ai_add_ctrl    : out std_logic;
rp_add_ctrl    : out std_logic;
ap_add_ctrl    : out std_logic;
add_ctrl_latch_in   : out std_logic_vector (6 downto 0);
add_ctrl_latch_out       : out std_logic_vector (6 downto 0);
clk_data_in    : out std_logic;
pen_data_in    : out std_logic;
ri_data_in    : out std_logic;
ai_data_in    : out std_logic;
rp_data_in    : out std_logic;
ap_data_in    : out std_logic;
data_in_latch_in  : out std_logic_vector (3 downto 0);
data_in_latch_out   : out std_logic_vector (3 downto 0);
clk_ram    : out std_logic;
pen_ram_ip    : out std_logic;
ri_ram_ip    : out std_logic;
ai_ram_ip    : out std_logic;
rp_ram_ip    : out std_logic;
ap_ram_ip    : out std_logic;
pen_ram_op    : out std_logic;
ri_ram_op    : out std_logic;
ai_ram_op    : out std_logic;
rp_ram_op    : out std_logic;
ap_ram_op    : out std_logic;
clk_data_out_1   : out std_logic;
pen_data_out_1   : out std_logic;
ri_data_out_1    : out std_logic;"

98
ai_data_out_1 : out std_logic;
rp_data_out_1 : out std_logic;
ap_data_out_1 : out std_logic;
data_out_1_latch_in : out std_logic_vector (3 downto 0);
data_out_1_latch_out : out std_logic_vector (3 downto 0);

clk_data_out_2 : out std_logic;
pen_data_out_2 : out std_logic;
ri_data_out_2 : out std_logic;
ai_data_out_2 : out std_logic;
rp_data_out_2 : out std_logic;
ap_data_out_2 : out std_logic;
data_out_2_latch_in : out std_logic_vector (3 downto 0);
data_out_2_latch_out : out std_logic_vector (3 downto 0);

data_out_2 : out std_logic_vector (3 downto 0);
data_out_1 : out std_logic_vector (3 downto 0);

end multi_port_ram;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- Begin Architecture rtl of Entity multi_port_ram
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
architecture rtl of multi_port_ram is
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- Component Declarations
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

component sram
port (reset : in std_logic;
clock : in std_logic;
enable : in std_logic;
read : in std_logic;
write : in std_logic;
read_addr : in std_logic_vector(1 downto 0);
write_addr : in std_logic_vector(1 downto 0);
data_in : in std_logic_vector(3 downto 0);
data_out : out std_logic_vector(3 downto 0)
);
end component;

component d_op
port (reset : IN std_logic;
pen : IN std_logic;

99
ap : IN std_logic;
ai : IN std_logic;
ri : OUT std_logic;
rp : OUT std_logic);
end component;

component d_ip
port (reset : IN std_logic;
pen : IN std_logic;
rp : IN std_logic;
ai : IN std_logic;
ri : OUT std_logic;
ap : OUT std_logic);
end component;

component p_ip
port (reset : IN std_logic;
pen : IN std_logic;
rp : IN std_logic;
ai : IN std_logic;
ri : OUT std_logic;
ap : OUT std_logic);
end component;

component t_ff
port (reset : in std_logic;
t_in : in std_logic;
neg_clk : in std_logic;
t_out : out std_logic);
end component;

component data_latch
port (reset : in std_logic;
data_in : in std_logic_vector(3 downto 0);
enable : in std_logic;
data_out : out std_logic_vector (3 downto 0));
end component;
component data_latch_7
port (reset : in std_logic;
data_in : in std_logic_vector(6 downto 0);
enable : in std_logic;
data_out : out std_logic_vector (6 downto 0)
);
end component;

component d_ff_1
port (reset : in std_logic;
d_in : in std_logic;
clk : in std_logic;
d_out : out std_logic
);
end component;

component d_ff_4
port (reset : in std_logic;
d_in : in std_logic_vector (3 downto 0);
clk : in std_logic;
d_out : out std_logic_vector (3 downto 0)
);
end component;

component d_ff_7
port (reset : in std_logic;
d_in : in std_logic_vector (6 downto 0);
clk : in std_logic;
d_out : out std_logic_vector (6 downto 0)
);
end component;

component clk_gen
port (reset : in std_logic;
sel : in std_logic;
ri : in std_logic;
ai : out std_logic;
lclk : out std_logic
);
end component;

component clk_gen_2port
port ( 
  reset   : IN std_logic;
  sel     : IN std_logic;
  ri1     : IN std_logic;
  ri2     : IN std_logic;
  ai1     : OUT std_logic;
  ai2     : OUT std_logic;
  lclk    : OUT std_logic
);
end component;

-- Signal Declarations

-------- misc signals --------
signal reset_s             : std_logic;
signal set                  : std_logic := '1';
signal clk_gen_1ns, clk_gen_2port_5ns : std_logic := '1';
signal clk_gen_2ns, clk_gen_2port_2_5ns : std_logic := '0';

-------- add_ctrl signals ------
signal pen_add_ctrl_s, ri_add_ctrl_s, ai_add_ctrl_s : std_logic;
signal ap_add_ctrl_s, rp_add_ctrl_s, clk_add_ctrl_s : std_logic;
signal add_ctrl_d_in_s, add_ctrl_d_out_s, add_ctrl_latch_out_s : std_logic_vector(6 downto 0);

-------- data_in signals --------
signal pen_data_in_s, ri_data_in_s, ai_data_in_s : std_logic;
signal ap_data_in_s, rp_data_in_s, clk_data_in_s : std_logic;
signal data_in_d_in_s, data_in_d_out_s, data_in_latch_out_s : std_logic_vector(3 downto 0);

-------- ram signals -------
signal pen_ram_ip_s, ai_ram_ip_s, ri_ram_ip_s : std_logic;
signal clk_ram_s, rp_ram_ip_s, ap_ram_ip_s : std_logic;
signal ri_ram_op_s, ai_ram_op_s, ap_ram_op_s, rp_ram_op_s : std_logic;
signal data_out_ram_s : std_logic_vector(3 downto 0);
signal t_ff_ram_op_s, pen_ram_op_s : std_logic;

-------- data_out_1 signals ------
signal pen_data_out_1_s, ri_data_out_1_s, ai_data_out_1_s : std_logic;
signal clk_data_out_1_s, rp_data_out_1_s, ap_data_out_1_s : std_logic;
signal data_out_1_latch_out_s, data_out_1_s : std_logic_vector(3 downto 0);

-------- data_out_2 signals ------
signal pen_data_out_2_s, ri_data_out_2_s, ai_data_out_2_s : std_logic;
signal clk_data_out_2_s, rp_data_out_2_s, ap_data_out_2_s : std_logic;
signal data_out_2_latch_out_s, data_out_2_s : std_logic_vector(3 downto 0);
begin
-- ------------------------
-- Component Port Mapping
-- ------------------------
------ ADD_CTRL Locally Synchronous Unit ------

uut_d_ff_add_ctrl : d_ff_7
port map (reset => reset_s, d_in => add_ctrl_d_in_s, clk => clk_add_ctrl_s, d_out => add_ctrl_d_out_s);

uut_t_ff_add_ctrl : t_ff
port map (reset => reset_s, t_in => add_ctrl_d_in_s(0), neg_clk => clk_add_ctrl_s, t_out => pen_add_ctrl_s);

uut_clk_gen_add_ctrl : clk_gen
port map (reset => reset_s, sel => clk_gen_2ns, ri => ri_add_ctrl_s, ai => ai_add_ctrl_s, lclk => clk_add_ctrl_s);

uut_d_op_add_ctrl : d_op
port map (reset => reset_s, pen => pen_add_ctrl_s, ap => ap_add_ctrl_s, ai => ai_add_ctrl_s, ri => ri_add_ctrl_s, rp => rp_add_ctrl_s);

uut_data_latch_7_add_ctrl : data_latch_7
port map (reset => reset_s, data_in => add_ctrl_d_out_s, enable => ap_add_ctrl_s);
data_out => add_ctrl_latch_out_s
);

------ DATA_IN Locally Synchronous Unit ------

uut_d_ff_data_in : d_ff_4
port map (
reset => reset_s,
d_in => data_in_d_in_s,
clk => clk_data_in_s,
d_out => data_in_d_out_s
);

uut_t_ff_data_in : t_ff
port map (
reset => reset_s,
t_in => set,
neg_clk => clk_data_in_s,
t_out => pen_data_in_s
);

uut_clk_gen_data_in : clk_gen
port map (
reset => reset_s,
.sel => clk_gen_2ns,
ri => ri_data_in_s,
ai => ai_data_in_s,
lclk => clk_data_in_s
);

uut_d_op_data_in : d_op
port map (
reset => reset_s,
pen => pen_data_in_s,
ap => ap_data_in_s,
ai => ai_data_in_s,
ri => ri_data_in_s,
rp => rp_data_in_s
);

uut_data_latch_data_in : data_latch
port map (
reset => reset_s,
data_in => data_in_d_out_s,
.enable => ap_data_in_s,
data_out => data_in_latch_out_s
);
); 

------ SRAM Loacally Synchronous Unit ------

uut_t_ff_ram_ip : t_ff
port map ( 
  reset       => reset_s,
  t_in        => set,
  neg_clk     => clk_ram_s,
  t_out       => pen_ram_ip_s
);

uut_d_ip_ram : d_ip
port map ( 
  reset       => reset_s,
  pen         => pen_ram_ip_s,
  rp          => rp_ram_ip_s,
  ai          => ai_ram_ip_s,
  ri          => ri_ram_ip_s,
  ap          => ap_ram_ip_s
);

uut_ram : sram
port map ( 
  reset       => reset_s,
  clock       => clk_ram_s,
  enable      => add_ctrl_latch_out_s(0),
  read        => add_ctrl_latch_out_s(2),
  write       => add_ctrl_latch_out_s(1),
  read_addr   => add_ctrl_latch_out_s(6 downto 5),
  write_addr  => add_ctrl_latch_out_s(4 downto 3),
  data_in     => data_in_latch_out_s,
  data_out    => data_out_ram_s
);

uut_clk_gen_2port_ram : clk_gen_2port
port map ( 
  reset       => reset_s,
  sel         => clk_gen_2port_2_5ns,
  ri1         => ri_ram_ip_s,
  ri2         => ri_ram_op_s,
  ai1         => ai_ram_ip_s,
  ai2         => ai_ram_op_s,
  lclk        => clk_ram_s
);
uut_d_ff_ram : d_ff_1
port map(
reset => reset_s,
d_in => add_ctrl_latch_out_s(2),
clk => clk_ram_s,
d_out => t_ff_ram_op_s);

uut_t_ff_ram_op : t_ff
port map(
reset => reset_s,
t_in => t_ff_ram_op_s,
neg_clk => clk_ram_s,
t_out => pen_ram_op_s);

uut_d_op_ram : d_op
port map(
reset => reset_s,
pen => pen_ram_op_s,
ap => ap_ram_op_s,
ai => ai_ram_op_s,
ri => ri_ram_op_s,
rp => rp_ram_op_s);

------ DATA_OUT_1 Loacally Synchronous Unit - 1 -------

uut_data_latch_data_out_1 : data_latch
port map(
reset => reset_s,
data_in => data_out_ram_s,
enable => ap_ram_op_s,
data_out => data_out_1_latch_out_s);

uut_t_ff_data_out_1 : t_ff
port map(
reset => reset_s,
t_in => set,
neg_clk => clk_data_out_1_s,
t_out => pen_data_out_1_s);

uut_clk_gen_data_out_1 : clk_gen
port map(
reset => reset_s,
    sel => clk_gen_1ns,
    ri => ri_data_out_1_s,
    ai => ai_data_out_1_s,
    lclk => clk_data_out_1_s);

uut_d_ip_data_out_1 : d_ip
port map (reset => reset_s,
    pen => pen_data_out_1_s,
    rp => rp_data_out_1_s,
    ai => ai_data_out_1_s,
    ri => ri_data_out_1_s,
    ap => ap_data_out_1_s);

uut_d_ff_data_out_1 : d_ff_4
port map (reset => reset_s,
    d_in => data_out_1_latch_out_s,
    clk => clk_data_out_1_s,
    d_out => data_out_1_s);

-------- DATA_OUT_2 Locally Synchronous Unit - 2 --------

uut_data_latch_data_out_2 : data_latch
port map (reset => reset_s,
    data_in => data_out_ram_s,
    enable => ap_ram_op_s,
    data_out => data_out_2_latch_out_s);

uut_t_ff_data_out_2 : t_ff
port map (reset => reset_s,
    t_in => set,
    neg_clk => clk_data_out_2_s,
    t_out => pen_data_out_2_s);

uut_clk_gen_data_out_2 : clk_gen
port map (reset => reset_s,
sel     => clk_gen_2ns,
ri      => ri_data_out_2_s,
ai      => ai_data_out_2_s,
lchk    => clk_data_out_2_s
);

uut_p_ip_data_out_2 : p_ip
port map (
reset   => reset_s,
pen     => pen_data_out_2_s,
rp      => rp_data_out_2_s,
ai      => ai_data_out_2_s,
ri      => ri_data_out_2_s,
ap      => ap_data_out_2_s
);

uut_d_ff_data_out_2 : d_ff_4
port map (reset   => reset_s,
d_in    => data_out_2_latch_out_s,
clk     => clk_data_out_2_s,
d_out   => data_out_2_s
);

-- -- -- -- -- -- -- -- -- -- --
-- Concurrent Signal Assignments
-- -- -- -- -- -- -- -- -- -- --

------ Entity input port signals being assigned to internal signals ------
reset_s                        <= reset;
add_ctrl_d_in_s(0)    <= enable;
add_ctrl_d_in_s(1)    <= write;
add_ctrl_d_in_s(2)    <= read;
add_ctrl_d_in_s(4 downto 3)   <= write_addr;
add_ctrl_d_in_s(6 downto 5)   <= read_addr;
data_in_d_in_s                <= data_in;
------ Internal signal assignment statements ------
set      < = '1';
ap_add_ctrl_s     <= ap_ram_ip_s;
ap_data_in_s     <= ap_ram_ip_s;
rp_ram_ip_s     <= rp_add_ctrl_s and rp_data_in_s;
ap_ram_op_s     <= ap_data_out_1_s and ap_data_out_2_s;
rp_data_out_1_s    <= rp_ram_op_s;
rp_data_out_2_s    <= rp_ram_op_s;
data_out_1     <= data_out_1_s;
data_out_2     <= data_out_2_s;
------ Signal assignments to entity output port signals ------
clk_add_ctrl <= clk_add_ctrl_s;
pen_add_ctrl <= pen_add_ctrl_s;
ri_add_ctrl <= ri_add_ctrl_s;
ai_add_ctrl <= ai_add_ctrl_s;
rp_add_ctrl <= rp_add_ctrl_s;
ap_add_ctrl <= ap_add_ctrl_s;
add_ctrl_latch_in <= add_ctrl_latch_in_s;
add_ctrl_latch_out <= add_ctrl_latch_out_s;

clk_data_in <= clk_data_in_s;
pen_data_in <= pen_data_in_s;
ri_data_in <= ri_data_in_s;
ai_data_in <= ai_data_in_s;
rp_data_in <= rp_data_in_s;
ap_data_in <= ap_data_in_s;
data_in_latch_in <= data_in_latch_in_s;
data_in_latch_out <= data_in_latch_out_s;

clk_ram <= clk_ram_s;
pen_ram_ip <= pen_ram_ip_s;
ri_ram_ip <= ri_ram_ip_s;
ai_ram_ip <= ai_ram_ip_s;
rp_ram_ip <= rp_ram_ip_s;
ap_ram_ip <= ap_ram_ip_s;
pen_ram_op <= pen_ram_op_s;
ri_ram_op <= ri_ram_op_s;
ai_ram_op <= ai_ram_op_s;
rp_ram_op <= rp_ram_op_s;
ap_ram_op <= ap_ram_op_s;

clk_data_out_1 <= clk_data_out_1_s;
pen_data_out_1 <= pen_data_out_1_s;
ri_data_out_1 <= ri_data_out_1_s;
ai_data_out_1 <= ai_data_out_1_s;
ap_data_out_1 <= ap_data_out_1_s;
rp_data_out_1 <= rp_data_out_1_s;
data_out_1_latch_in <= data_out_1_latch_in_s;
data_out_1_latch_out <= data_out_1_latch_out_s;
data_out_1 <= data_out_1_s;

clk_data_out_2 <= clk_data_out_2_s;
pen_data_out_2 <= pen_data_out_2_s;
ri_data_out_2 <= ri_data_out_2_s;
ai_data_out_2 <= ai_data_out_2_s;
ap_data_out_2 <= ap_data_out_2_s;
rp_data_out_2 <= rp_data_out_2_s;
Synchronous-Asynchronous Data Communication Example [RTL – async_dop]

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity async_dop is
  port (reset : in std_logic;
        address : in std_logic_vector(1 downto 0);
        add_valid : in std_logic;
        clk : out std_logic;
        pen : out std_logic;
        ri : out std_logic;
        ai : out std_logic;
        rp : out std_logic;
        add_latch_in : out std_logic_vector (1 downto 0);
        add_latch_out : out std_logic_vector (1 downto 0);
        ap : out std_logic;
        data_latch_in : out std_logic_vector (3 downto 0);
        data_latch_out : out std_logic_vector (3 downto 0);
        data_out : out std_logic_vector(3 downto 0)));
end async_dop;

architecture rtl of async_dop is

component d_op
  port (reset : IN std_logic;
        pen : IN std_logic;
        ap : IN std_logic;
        ai : IN std_logic;
        ri : OUT std_logic;
        rp : OUT std_logic
        ...)
component t_ff
port (reset : in std_logic;
t_in : in std_logic;
neg_clk : in std_logic;
t_out : out std_logic);
end component;

component data_latch_2
port (reset : in std_logic;
data_in : in std_logic_vector(1 downto 0);
enable : in std_logic;
data_out : out std_logic_vector (1 downto 0)));
end component;

component data_latch
port (reset : in std_logic;
data_in : in std_logic_vector(3 downto 0);
enable : in std_logic;
data_out : out std_logic_vector (3 downto 0)));
end component;

component async_rom
port (reset : in std_logic;
req_in : in std_logic;
address : in std_logic_vector(1 downto 0);
data_out : out std_logic_vector(3 downto 0);
ack_out : out std_logic)
);
end component;

component clk_gen
port (reset : in std_logic;
  sel : in std_logic;
  ri : in std_logic;
  ai : out std_logic;

);
lclk    : out std_logic
);
end component;

component d_ff_3
port (reset  : in std_logic;
d_in    : in std_logic_vector (2 downto 0);
clk     : in std_logic;
d_out   : out std_logic_vector (2 downto 0)
);
end component;

component d_ff_4
port (reset  : in std_logic;
d_in    : in std_logic_vector (3 downto 0);
clk     : in std_logic;
d_out   : out std_logic_vector (3 downto 0)
);
end component;

signal d_ff_in, d_ff_out     : std_logic_vector (2 downto 0);
signal address_latch_out    : std_logic_vector (1 downto 0);
signal data_latch_in_s, data_latch_out_s : std_logic_vector (3 downto 0);
signal clk_dop, pen_dop, ri_dop, rp_dop, ai_dop, ap_dop  : std_logic;
signal sel_1ns                                             : std_logic := '1';
signal sel_2ns                                             : std_logic := '0';

begin

uut_d_ff_3 : d_ff_3
port map (reset  => reset,
d_in    => d_ff_in,
clk     => clk_dop,
d_out   => d_ff_out
);

uut_t_ff : t_ff
port map (reset  => reset,
t_in    => d_ff_out(0),
reset   => clk_dop,
t_out   => pen_dop
);
uut_d_op : d_op
port map (
    reset => reset,
    pen   => pen_dop,
    ap    => ap_dop,
    ai    => ai_dop,
    ri    => ri_dop,
    rp    => rp_dop)
);

uut_clk_gen : clk_gen
port map (
    reset => reset,
    sel   => sel_1ns,
    ri    => ri_dop,
    ai    => ai_dop,
    lclk  => clk_dop
);

uut_addLatch : data_latch_2
port map (
    reset => reset,
    data_in => d_ff_out(2 downto 1),
    enable => rp_dop,
    data_out => address_latch_out
);

uut_data_latch : data_latch
port map (
    reset => reset,
    data_in => data_latch_in_s,
    enable => ap_dop,
    data_out => data_latch_out_s
);

uut_async_rom : async_rom
port map (
    reset => reset,
    req_in => rp_dop,
    address => address_latch_out,
    data_out => data_latch_in_s,
    ack_out => ap_dop
);

uut_d_ff_out : d_ff_4
port map (
    reset  => reset,
    d_in   => data_latch_out_s,
    clk    => clk_dop,
    d_out  => data_out
);

d_ff_in(0)    <= add_valid;
d_ff_in(2 downto 1)   <= address;
clk           <= clk_dop;
pen           <= pen_dop;
ri              <= ri_dop;
ai              <= ai_dop;
rp            <= rp_dop;
add_latch_in     <= d_ff_out(2 downto 1);
add_latch_out     <= address_latch_out;
ap            <= ap_dop;
data_latch_in     <= data_latch_in_s;
data_latch_out        <= data_latch_out_s;
end rtl;

Demand-Type Input Demand-Type Output [RTL – dip_dop]

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-----------------------------------------------------
--  Begin entity dip_dop
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
entity dip_dop is
port (]
    ------------ i/p port signals --------------
    reset        : in std_logic;
data_in_fin    : in std_logic_vector (3 downto 0);
data_avail     : in std_logic;
data_accept    : in std_logic;
    ------------ o/p port signals --------------
data_out_fin   : out std_logic_vector (3 downto 0);
    pen_ip_fin    : out std_logic;
    pen_op_fin    : out std_logic;
    ap_fin        : out std_logic;
    rp_fin        : out std_logic;
end rtl;
ai_ip_fin : out std_logic;
ri_ip_fin : out std_logic;
ai_op_fin : out std_logic;
ri_op_fin : out std_logic;
lclk_ip_fin : out std_logic;
lclk_op_fin : out std_logic;
ta_op_fin : out std_logic;
ta_ip_fin : out std_logic
);
end dip_dop;

--------------------------------------------------------
--  Begin architecture of entity dip_dop
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
architecture rtl of dip_dop is
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- Component instantiation
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
component d_ip
PORT (reset : IN std_logic;
   pen : IN std_logic;
   rp : IN std_logic;
   ai : IN std_logic;
   ri : OUT std_logic;
   ap : OUT std_logic);
end component;

component d_op
PORT (reset : IN std_logic;
   pen : IN std_logic;
   ap : IN std_logic;
   ai : IN std_logic;
   ri : OUT std_logic;
   rp : OUT std_logic);
end component;

component clk_gen
port (reset : in std_logic;
   sel : in std_logic;
   ri : in std_logic;
   ai : out std_logic;
   ai : out std_logic;

115
lclk : out std_logic
);
end component;

cOMPONENT t_ff
PORT (reset : in std_logic;
t_in : in std_logic;
neg_clk : in std_logic;
t_out : out std_logic)
);
end component;

COMPONENT data_latch
PORT (reset : in std_logic;
data_in : in std_logic_vector(3 downto 0);
enable : in std_logic;
data_out : out std_logic_vector(3 downto 0))
);
end component;

COMPONENT ff_ta IS
PORT (reset : in std_logic;
set : in std_logic;
clk : in std_logic;
ff_out : out std_logic)
);
end component;

-- Signal declarations

signal pen_ip_s, pen_op_s, rp_s : std_logic;
signal ap_s, ri_ip_s, ai_ip_s : std_logic;
signal ri_op_s, ai_op_s : std_logic;
signal data_in_s : std_logic_vector(3 downto 0);
signal data_out_s : std_logic_vector(3 downto 0);
signal lclk_ip_s, lclk_op_s : std_logic;
signal t_ff_ip_s, t_ff_op_s : std_logic;
signal dip_sel : std_logic := '1';
signal dop_sel : std_logic := '0';
signal ta_ip_s, ta_op_s : std_logic;
signal data_avail_s, data_accept_s : std_logic;
begin

d_ip:

reset  => reset,
pen    => pen_ip_s,
rp     => rp_s,
ai     => ai_ip_s,
ri     => ri_ip_s,
ap     => ap_s
);

d_op:

reset  => reset,
pen    => pen_op_s,
ap     => ap_s,
ai     => ai_op_s,
ri     => ri_op_s,
rp     => rp_s
);

dip:

reset  => reset,
sel    => dip_sel,
ri     => ri_ip_s,
ai     => ai_ip_s,
lclk   => lclk_ip_s
);

dop:

reset  => reset,
sel    => dop_sel,
ri     => ri_op_s,
ai     => ai_op_s,
lclk   => lclk_op_s
);

t_ff_dop:

reset     => reset,
117
t_in => t_ff_op_s,
neg_clk => lclk_op_s,
t_out => pen_op_s
);

uut_t_ff_dip : t_ff
port map (reset => reset,
t_in => t_ff_ip_s,
neg_clk => lclk_ip_s,
t_out => pen_ip_s);

uut_data_latch : data_latch
port map (reset => reset,
data_in => data_in_s,
enable => ap_s,
data_out => data_out_s);

uut_ta_op : ff_ta
port map (reset => reset,
set => ap_s,
clk => lclk_op_s,
ff_out => ta_op_s);

uut_ta_ip : ff_ta
port map (reset => reset,
set => ai_ip_s,
clk => lclk_ip_s,
ff_out => ta_ip_s);

-- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- concurrent signal assignment statements
-- -- -- -- -- -- -- -- -- -- -- -- -- -- --

t_ff_op_s <= (data_avail_s) and (not(ta_op_s));
t_ff_ip_s <= (data_accept_s) and (not(ta_ip_s));
data_in_s <= data_in_fin;
data_avail_s <= data_avail;
data_accept_s <= data_accept;
data_out_fin <= data_out_s;
pen_ip_fin <= pen_ip_s;
pen_op_fin <= pen_op_s;
ap_fin <= ap_s;
rp_fin <= rp_s;
ai_ip_fin <= ai_ip_s;
ri_ip_fin <= ri_ip_s;
ai_op_fin <= ai_op_s;
ri_op_fin <= ri_op_s;
lclk_ip_fin <= lclk_ip_s;
lclk_op_fin <= lclk_op_s;
ta_ip_fin <= ta_ip_s;
ta_op_fin <= ta_op_s;
end rtl;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- End architecture of entity dip_dop
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

Demand-Type Input Poll-Type Output [RTL – dip_pop]

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity dip_pop is
port (reset : in std_logic;
data_in_fin : in std_logic_vector (3 downto 0);
data_avail : in std_logic;
data_accept : in std_logic;
----------------- o/p port signals ----------------
data_out_fin : out std_logic_vector (3 downto 0);
pen_ip_fin : out std_logic;
pen_op_fin : out std_logic;
ap_fin : out std_logic;
rp_fin : out std_logic;
ai_ip_fin : out std_logic;
ri_ip_fin : out std_logic;
ai_op_fin : out std_logic;
ri_op_fin : out std_logic;
lclk_ip_fin : out std_logic;
lclk_op_fin : out std_logic;
lclk_op_fin : out std_logic;
);


ta_ip_fin : out std_logic;
ta_op_fin : out std_logic);
end dip_pop;

--------------------------------------------------------
--  Begin architecture of entity pip_dop
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
architecture rtl of dip_pop is
-- -- -- -- -- -- -- -- --
-- Component instantiation
-- -- -- -- -- -- -- -- --
component d_ip
PORT ( reset : IN std_logic;
pen : IN std_logic;
rp : IN std_logic;
ai : IN std_logic;
ri : OUT std_logic;
ap : OUT std_logic
);
end component;

compartment p_op
PORT ( reset : IN std_logic;
pen : IN std_logic;
ap : IN std_logic;
ai : IN std_logic;
ri : OUT std_logic;
rp : OUT std_logic
);
end component;

compartment clk_gen
port ( reset : in std_logic;
sel : in std_logic;
ri : in std_logic;
ai : out std_logic;
lclk : out std_logic
);
end component;

compartment t_ff
port ( 120
reset : in std_logic;
t_in : in std_logic;
neg_clk : in std_logic;
t_out : out std_logic
);
end component;

component data_latch
port (reset : in std_logic;
data_in : in std_logic_vector(3 downto 0);
enable : in std_logic;
data_out : out std_logic_vector (3 downto 0)
);
end component;

component ff_ta is
port (reset : in std_logic;
set : in std_logic;
clk : in std_logic;
ff_out : out std_logic
);
end component;

-- Signal declarations

-- Component port mapping

begin

uut_dip: d_ip
port map (  
reset  => reset,
pen    => pen_ip_s,
rp     => rp_s,
ai     => ai_ip_s,
ri     => ri_ip_s,
ap     => ap_s 
);

uut_pop : p_op  
port map (  
reset  => reset,
pen    => pen_op_s,
ap    => ap_s,
ai     => ai_op_s,
ri     => ri_op_s,
rp     => rp_s
);

uut_clkgen_dip : clk_gen  
port map (  
reset  => reset,
.sel    => pop_sel,
ri     => ri_ip_s,
ai     => ai_ip_s,
lclk   => lclk_ip_s
);

uut_clk_gen_pop : clk_gen  
port map (  
reset  => reset,
.sel    => dip_sel,
ri     => ri_op_s,
ai     => ai_op_s,
lclk   => lclk_op_s
);

uut_t_ff_pop : t_ff  
port map (  
reset     => reset,
t_in      => t_ff_op_s,
neg_clk   => lclk_op_s,
t_out     => pen_op_s 
);

uut_t_ff_dip : t_ff

port map (
    reset     => reset,
    t_in      => t_ff_ip_s,
    neg_clk   => lclk_ip_s,
    t_out     => pen_ip_s
);

uut_data_latch : data_latch
port map (
    reset      => reset,
    data_in   => data_in_s,
    enable    => ap_s,
    data_out  => data_out_s
);

uut_ta_op : ff_ta
port map (
    reset   => reset,
    set     => ap_s,
    clk     => lclk_op_s,
    ff_out  => ta_op_s
);

uut_ta_ip : ff_ta
port map (
    reset   => reset,
    set     => ai_ip_s,
    clk     => lclk_ip_s,
    ff_out  => ta_ip_s
);

-- concurrent signal assignment statements

-- -- -- -- -- -- -- -- -- -- -- -- -- -- --
t_ff_op_s  <= (data_avail_s) and (not(ta_op_s));
t_ff_ip_s  <= (data_accept_s) and (not(ta_ip_s));
data_out_fin <= data_out_s;
pen_ip_fin  <= pen_ip_s;
pen_op_fin  <= pen_op_s;
ap_fin     <= ap_s;
rp_fin     <= rp_s;
ai_ip_fin  <= ai_ip_s;
ri_ip_fin  <= ri_ip_s;
ai_op_fin  <= ai_op_s;
ri_op_fin  <= ri_op_s;
lclk_ip_fin <= lclk_ip_s;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pip_dop is
port (
          reset         : in std_logic;
          data_in_fin   : in std_logic_vector (3 downto 0);
          data_avail    : in std_logic;
          data_accept   : in std_logic;
          -------------- o/p port signals ---------------
          data_out_fin  : out std_logic_vector (3 downto 0);
          pen_ip_fin    : out std_logic;
          pen_op_fin    : out std_logic;
          ap_fin        : out std_logic;
          rp_fin        : out std_logic;
          ai_ip_fin     : out std_logic;
          ri_ip_fin     : out std_logic;
          ai_op_fin     : out std_logic;
          ri_op_fin     : out std_logic;
          lclk_ip_fin   : out std_logic;
          lclk_op_fin   : out std_logic;
          ta_ip_fin     : out std_logic;
          ta_op_fin     : out std_logic
        );
end pip_dop;

-- Begin architecture of entity pip_dop
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
architecture rtl of pip_dop is

lclk_op_fin <= lclk_op_s;
ta_ip_fin <= ta_ip_s;
ta_op_fin <= ta_op_s;
end rtl;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- End architecture of entity dip_pop
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

Poll-Type Input Demand-Type Output [RTL – pip_dop]
-- Component instantiation

component p_ip
PORT (
reset  : IN std_logic;
pen    : IN std_logic;
rp     : IN std_logic;
ai     : IN std_logic;
ri     : OUT std_logic;
ap     : OUT std_logic
);
end component;

component d_op
PORT (reset  : IN std_logic;
pen    : IN std_logic;
ap     : IN std_logic;
ai     : IN std_logic;
ri     : OUT std_logic;
rp     : OUT std_logic
);
end component;

component clk_gen
port (reset  : in std_logic;
sel    : in std_logic;
ri     : in std_logic;
ai     : out std_logic;
lclk   : out std_logic
);
end component;

component t_ff
port (reset     : in std_logic;
t_in      : in std_logic;
neg_clk  : in std_logic;
t_out     : out std_logic
);
end component;

component data_latch
port (  
reset      : in std_logic;  
data_in   : in std_logic_vector(3 downto 0);  
enable    : in std_logic;  
data_out  : out std_logic_vector (3 downto 0)  
);  
end component;

component ff_ta is  
port (  
reset   : in std_logic;  
set     : in std_logic;  
clk     : in std_logic;  
ff_out  : out std_logic  
);  
end component;

-- -- -- -- -- -- -- --  
-- Signal declarations  
-- -- -- -- -- -- -- --

signal pen_ip_s, pen_op_s, rp_s      : std_logic;  
Signal ap_s, ri_ip_s, ai_ip_s       : std_logic;  
signal ri_op_s, ai_op_s             : std_logic;  
signal data_in_s : std_logic_vector (3 downto 0);  
signal data_out_s : std_logic_vector (3 downto 0);  
signal lclk_ip_s, lclk_op_s         : std_logic;  
signal t_ff_ip_s, t_ff_op_s         : std_logic;  
signal pip_sel                    : std_logic := '1';  
signal dop_sel                     : std_logic := '0';  
signal ta_ip_s, ta_op_s             : std_logic;  
signal data_avail_s, data_accept_s  : std_logic;

-- -- -- -- -- -- -- ----  
-- Component port mapping  
-- -- -- -- -- -- -- ----

begin

uut_pip: p_ip
port map (  
reset  => reset,  
pen    => pen_ip_s,  
rp     => rp_s,  
ai     => ai_ip_s,  
ri     => ri_ip_s,  
ap     => ap_s  
);
uut_dop : d_op
port map (  
  reset  => reset,
  pen    => pen_op_s,
  ap     => ap_s,
  ai     => ai_op_s,
  ri     => ri_op_s,
  rp     => rp_s
);

uut_clkgen_pip : clk_gen
port map (  
  reset  => reset,
  sel    => pip_sel,
  ri     => ri_ip_s,
  ai     => ai_ip_s,
  lclk   => lclk_ip_s
);

uut_clk_gen_dop : clk_gen
port map (  
  reset  => reset,
  sel    => dop_sel,
  ri     => ri_op_s,
  ai     => ai_op_s,
  lclk   => lclk_op_s
);

uut_t_ff_dop : t_ff
port map (  
  reset  => reset,
  t_in   => t_ff_op_s,
  neg_clk => lclk_op_s,
  t_out  => pen_op_s
);

uut_t_ff_dip : t_ff
port map (  
  reset  => reset,
  t_in   => t_ff_ip_s,
  neg_clk => lclk_ip_s,
  t_out  => pen_ip_s
);

uut_data_latch : data_latch
port map (reset => reset,
data_in  => data_in_s,
enable   => ap_s,
data_out => data_out_s);

uut_ta_op : ff_ta
port map (reset   => reset,
set     => ap_s,
clk     => lclk_op_s,
ff_out  => ta_op_s);

uut_ta_ip : ff_ta
port map (reset   => reset,
set     => ai_ip_s,
clk     => lclk_ip_s,
ff_out  => ta_ip_s);

-- concurrent signal assignment statements
--

--

end rtl;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pip_pop is
port (------- i/p port signals -------------
reset         : in std_logic;
data_in_fin   : in std_logic_vector (3 downto 0);
data_avail    : in std_logic;
data_accept   : in std_logic;

------- o/p port signals -------------
data_out_fin  : out std_logic_vector (3 downto 0);
pen_ip_fin    : out std_logic;
pen_op_fin    : out std_logic;
ap_fin        : out std_logic;
rp_fin        : out std_logic;
ai_ip_fin     : out std_logic;
ri_ip_fin     : out std_logic;
ai_op_fin     : out std_logic;
ri_op_fin     : out std_logic;
lclk_ip_fin   : out std_logic;
lclk_op_fin   : out std_logic;
ta_op_fin     : out std_logic;
ta_ip_fin     : out std_logic
);
end pip_pop;

-- Begin architecture of entity dip_dop
architecture rtl of pip_pop is
-- Component instantiation
component p_ip
PORT (reset : IN std_logic;
pen : IN std_logic;
rp : IN std_logic;
ai : IN std_logic;
ri : OUT std_logic;
ap : OUT std_logic;
);
component p_op
PORT (
reset : IN std_logic;
pen   : IN std_logic;
ap   : IN std_logic;
ai    : IN std_logic;
ri    : OUT std_logic;
rp    : OUT std_logic
);
end component;

component clk_gen
port (
reset  : in std_logic;
sel    : in std_logic;
ri     : in std_logic;
ai     : out std_logic;
lclk   : out std_logic
);
end component;

component t_ff
port (
reset     : in std_logic;
t_in      : in std_logic;
neg_clk  : in std_logic;
t_out     : out std_logic
);
end component;

component data_latch
port (reset      : in std_logic;
data_in     : in std_logic_vector(3 downto 0);
enable     : in std_logic;
data_out   : out std_logic_vector (3 downto 0)
);
end component;

component ff_ta is
port (reset : in std_logic;
set    : in std_logic;
);
clk : in std_logic;
ff_out : out std_logic
);
end component;

-- Signal declarations

signal pen_ip_s, pen_op_s, rp_s : std_logic;
Signal ap_s, ri_ip_s, ai_ip_s : std_logic;
signal ri_op_s, ai_op_s : std_logic;
signal data_in_s : std_logic_vector (3 downto 0);
signal data_out_s : std_logic_vector (3 downto 0);
signal lclk_ip_s, lclk_op_s : std_logic;
signal t_ff_ip_s, t_ff_op_s : std_logic;
signal pip_sel : std_logic := '0';
signal pop_sel : std_logic := '1';
signal ta_ip_s, ta_op_s : std_logic;
signal data_avail_s, data_accept_s : std_logic;

-- Component port mapping

begin

uut_pip: p_ip
port map ( reset => reset,
pen => pen_ip_s,
rp => rp_s,
ai => ai_ip_s,
ri => ri_ip_s,
ap => ap_s
);

uut_pop : p_op
port map ( reset => reset,
pen => pen_op_s,
ap => ap_s,
ai => ai_op_s,
ri => ri_op_s,
rp => rp_s
);

uut_clkgen_pip : clk_gen
port map (  
reset    => reset,  
sel      => pip_sel,  
ri       => ri_ip_s,  
ai       => ai_ip_s,  
lclk     => lclk_ip_s  
);  

uut_clk_gen_pop : clk_gen  
port map (  
reset    => reset,  
sel      => pip_sel,  
ri       => ri_op_s,  
ai       => ai_op_s,  
lclk     => lclk_op_s  
);  

uut_t_ff_pop : t_ff  
port map (  
reset       => reset,  
t_in        => t_ff_op_s,  
neg_clk     => lclk_op_s,  
t_out       => pen_op_s  
);  

uut_t_ff_pip : t_ff  
port map (  
reset       => reset,  
t_in        => t_ff_ip_s,  
neg_clk     => lclk_ip_s,  
t_out       => pen_ip_s  
);  

uut_data_latch : data_latch  
port map (  
reset      => reset,  
data_in    => data_in_s,  
enable     => ap_s,  
data_out   => data_out_s  
);  

uut_ta_op : ff_ta  
port map (  
reset    => reset,  
set       => ap_s,  
clk       => lclk_op_s,  
ff_out    => ta_op_s  
)
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY d_ip IS
PORT (  
  reset : IN std_logic;  
  pen : IN std_logic;  
  rp : IN std_logic;  
  ai : IN std_logic;  
  ri : OUT std_logic;  
  ap : OUT std_logic  
);  
END d_ip;  

ARCHITECTURE behavioral of d_ip IS  

TYPE afsm_state IS (s0,s1,s2,s3,s4,s5,s6,s7);  
SIGNAL curr_state, next_state : afsm_state := s0;  
SIGNAL ri_s, ap_s : std_logic := '0';  

BEGIN  

dip : PROCESS (reset, pen, rp, ai, curr_state, next_state)  
BEGIN  

IF (reset /= '1') THEN  
  ri_s <= '0';  
  ap_s <= '0';  
ELSE  
  CASE curr_state IS  
  WHEN s0 =>  
    IF (pen'EVENT AND pen = '1') AND (rp = '1' OR rp = '0') THEN  
      ri_s <= '1' after 0.1 ns;  
      next_state <= s1;  
    END IF;  
  WHEN s1 =>  
    IF (ai = '1') AND (rp = '1') THEN  
      ap_s <= '1' after 0.1 ns;  
      next_state <= s2;  
    END IF;  
  WHEN s2 =>  
    IF (rp= '0') THEN  
      ri_s <= '0' after 0.1 ns;  
      next_state <= s3;  
    END IF;  
  WHEN s3 =>  
    IF (ai = '0') THEN  
      ap_s <= '0' after 0.1 ns;  
      next_state <= s4;  
    END IF;  
  END CASE;  
END IF;  
END PROCESS;  

END behavioral;
IF (pen'EVENT AND pen = '0') AND (rp = '1' OR rp = '0') THEN
    ri_s       <= '1' after 0.1 ns;
    next_state <= s5;
END IF;
WHEN s5 =>
    IF (ai = '1') AND (rp = '1') THEN
        ap_s       <= '1' after 0.1 ns;
        next_state <= s6;
    END IF;
WHEN s6 =>
    IF (rp = '0') THEN
        ri_s       <= '0' after 0.1 ns;
        next_state <= s7;
    END IF;
WHEN s7 =>
    IF (ai = '0') THEN
        ap_s       <= '0' after 0.1 ns;
        next_state <= s0;
    END IF;
END CASE;
END IF;
END PROCESS dip;

curr_state <= next_state;
ri       <= ri_s;
ap       <= ap_s;

END behavioral;

Demand-Type Output Port Controller [Behavioral – d_op]

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY d_op IS
PORT ( reset : IN std_logic;
       pen  : IN std_logic;
       ap   : IN std_logic;
       ai   : IN std_logic;
       ri   : OUT std_logic;
       rp   : OUT std_logic ) ;
ARCHITECTURE behavioral of d_op IS

TYPE afsm_state IS (s0, s1, s2, s3, s4, s5, s6, s7, s8);
SIGNAL curr_state, next_state : afsm_state := s0;
SIGNAL ri_s, rp_s : std_logic := '0';
BEGIN

dop : PROCESS (pen, ap, ai, curr_state, next_state, reset)
BEGIN
IF (reset /= '1') THEN
rp_s <= '0';
ri_s <= '0';
else
CASE curr_state IS
  WHEN s0 =>
    IF (pen'EVENT AND pen = '1') THEN
      ri_s <= '1' after 0.1 ns;
      next_state <= s1;
    END IF;
  WHEN s1 =>
    IF (ai = '1') THEN
      rp_s <= '1' after 0.1 ns;
      next_state <= s2;
    END IF;
  WHEN s2 =>
    IF (ap = '1') THEN
      rp_s <= '0' after 0.1 ns;
      next_state <= s3;
    END IF;
  WHEN s3 =>
    IF (ap = '0') THEN
      ri_s <= '0' after 0.1 ns;
      next_state <= s4;
    END IF;
  WHEN s4 =>
    IF (pen'EVENT AND pen = '0') AND (ai = '0') THEN
      ri_s <= '1' after 0.1 ns;
      next_state <= s5;
    END IF;
  WHEN s5 =>
    IF (ai = '1') THEN
      rp_s <= '1' after 0.1 ns;
      next_state <= s6;
    END IF;
END CASE;
END IF;
END PROCESS dop;
END d_op;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity p_ip is
  PORT (
    reset   : IN std_logic;
    pen     : IN std_logic;
    rp      : IN std_logic;
    ai      : IN std_logic;
    ri      : OUT std_logic;
    ap      : OUT std_logic;
  );
END p_ip;

ARCHITECTURE rtl of p_ip IS
  TYPE afsm_state IS (s0,s1,s2,s3,s4,s5,s6,s7);
  SIGNAL dop : afsm_state;
  SIGNAL ri_s : std_logic;
  SIGNAL rp_s : std_logic;
BEGIN
  PROCESS dop
  BEGIN
    CASE dop IS
    WHEN s0 =>
      IF (ap= '1') THEN
        rp_s <= '0' after 0.1 ns;
        next_state <= s7;
      END IF;
    WHEN s6 =>
      IF (ap= '1') THEN
        rp_s <= '0' after 0.1 ns;
        next_state <= s7;
      END IF;
    WHEN s7 =>
      IF (ap = '0') THEN
        ri_s <= '0' after 0.1 ns;
        next_state <= s8;
      END IF;
    WHEN s8 =>
      IF (ai = '0') AND (pen'EVENT AND pen = '1') THEN
        ri_s <= '1' after 0.1 ns;
        next_state <= s1;
      END IF;
    END CASE;
  END PROCESS dop;
  curr_state <= next_state;
  ri     <= ri_s;
  rp     <= rp_s;
END behavioral;

Poll-Type Input Port Controller [Behavioral – p_ip]
BEGIN

if reset /= '1' then
  ri_s  <= '0';
  ap_s  <= '0';
else
  CASE curr_state IS
    WHEN s0 =>
      IF (pen'EVENT AND pen = '1') AND (rp = '1') THEN
        ri_s <= '1' after 0.1 ns;
        next_state <= s1;
      END IF;
      WHEN s1 =>
        IF (ai = '1') THEN
          ap_s   <= '1' after 0.1 ns;
          next_state  <= s2;
        END IF;
        WHEN s2 =>
          IF (rp= '0') THEN
            ri_s   <= '0' after 0.1 ns;
            ap_s   <= '0' after 0.1 ns;
            next_state <= s3;
          END IF;
          WHEN s3 =>
            IF (ai = '0') AND (rp = '1' or rp = '0') THEN
              next_state  <= s4;
            END IF;
            WHEN s4 =>
              IF (pen'EVENT AND pen = '0') AND (rp = '1') THEN
                ri_s <= '1' after 0.1 ns;
                next_state <= s5;
              END IF;
              WHEN s5 =>
                IF (ai = '1') THEN
                  ap_s   <= '1' after 0.1 ns;
                  next_state  <= s6;
                END IF;
                WHEN s6 =>
                  IF (rp= '0') THEN
                    ri_s  <= '0' after 0.1 ns;
                    ap_s  <= '0' after 0.1 ns;
                  END IF;
  END CASE;
END IF;
next_state <= s7;
END IF;
WHEN s7 =>
IF (ai = '0') and (rp = '0' or rp = '1') THEN
next_state <= s0;
END IF;
END CASE;
END IF;
END PROCESS pip;

ri          <= ri_s;
ap  <= ap_s;
curr_state  <= next_state;

END rtl;

Poll-Type Output Port Controller [Behavioral – p_op]

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity p_op is
PORT (
  reset  : IN std_logic;
  pen    : IN std_logic;
  ap     : IN std_logic;
  ai     : IN std_logic;
  ri     : OUT std_logic;
  rp     : OUT std_logic
);
END p_op;

ARCHITECTURE rtl of p_op IS

TYPE afsm_state IS (s0,s1,s2,s3,s4,s5,s6,s7,s8);
SIGNAL curr_state, next_state : afsm_state := s0;
SIGNAL ri_s, rp_s              : std_logic := '0';

BEGIN

pop : PROCESS (pen, ap, ai, curr_state, next_state, reset)
BEGIN
IF reset /= '1' then
ri_s <= '0';
rp_s <= '0';
ELSE
CASE curr_state IS
  WHEN s0 =>
    IF (pen'EVENT AND pen = '1') THEN
      rp_s <= '1' after 0.1 ns;
      curr_state <= s1;
    END IF;
  WHEN s1 =>
    IF (ap = '1') THEN
      ri_s <= '1';
      curr_state <= s2;
    END IF;
  WHEN s2 =>
    IF (ai = '1') THEN
      rp_s <= '0' after 0.1 ns;
      curr_state <= s3;
    END IF;
  WHEN s3 =>
    IF (ap = '0') THEN
      ri_s <= '0' after 0.1 ns;
      curr_state <= s4;
    END IF;
  WHEN s4 =>
    IF (pen'EVENT AND pen = '0') AND (ai = '0') THEN
      rp_s <= '1' after 0.1 ns;
      curr_state <= s5;
    END IF;
  WHEN s5 =>
    IF (ap = '1') THEN
      ri_s <= '1';
      curr_state <= s6;
    END IF;
  WHEN s6 =>
    IF (ai = '1') THEN
      rp_s <= '0' after 0.1 ns;
      curr_state <= s7;
    END IF;
  WHEN s7 =>
    IF (ap = '0') THEN
      ri_s <= '0' after 0.1 ns;
      curr_state <= s8;
    END IF;
  WHEN s8 =>
    IF (ai = '0') AND (pen'EVENT AND pen = '1') THEN
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity async_rom is
port (reset      : in std_logic;
      req_in    : in std_logic;
      address   : in std_logic_vector(1 downto 0);
      data_out  : out std_logic_vector(3 downto 0);
      ack_out   : out std_logic);
end async_rom;

architecture behavioral of async_rom is

type mem_array is array(0 to 3) of std_logic_vector(3 downto 0);
signal mem: mem_array;

begin
read: process(address, mem, reset, req_in)
   -- index is an index into the array
   variable index: integer range 0 to 3 := 0; -- initialize index to 0
begin
   -- initialize the ROM contents
   mem(0) <= "0001";
   mem(1) <= "0010";
   mem(2) <= "0100";
   mem(3) <= "1000";
   if reset /= '1' then
      data_out <= (data_out'range => 'Z');
ack_out <= '0';
elsif (req_in = '1') then
index := conv_integer(address(1 downto 0));
data_out <= mem(index)after 2 ns;
ack_out <= '1' after 2 ns;
elsif (req_in = '0')then
ack_out <= '0' after 1 ns;
end if;
end process;
end behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity clk_gen is
port (reset  : in std_logic;
      sel    : in std_logic;
      ri     : in std_logic;
      ai     : out std_logic;
      lclk   : out std_logic);
end clk_gen;

architecture rtl of clk_gen is

component mutual_ex
port (reset  : in std_logic;
      rs_in   : in std_logic;
      rr_in   : in std_logic;
      as_out  : out std_logic;
      ar_out  : out std_logic);
end component;

signal ri_s, ai_s, rclk_s, lclk_s : std_logic;

begin

uut_me : mutual_ex


port map (  
reset   => reset,  
rs_in   => ri_s,  
rr_in   => rclk_s,  
as_out => ai_s,  
ar_out => lclk_s  
);

ring_clk : process (lclk_s, sel)  
begn  
if sel = '1' then  
rclk_s <= not (lclk_s) after 1 ns; -- for multi-point  
--     rclk_s <= not (lclk_s) after 0.5 ns; -- for point-to-point  
elsif sel = '0' then  
rclk_s <= not (lclk_s) after 2 ns; -- for multi-port  
--    rclk_s <= not (lclk_s) after 0.25 ns; -- for point-to-point  
else  
rclk_s <= '0';  
end if;  
end process ring_clk;

ri_s   <= ri;  
ai     <= ai_s;  
lclk   <= lclk_s;
end rtl;

Multi Request Clock Generator [Behavioral – clk_gen_2port]

library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.ALL;  
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity clk_gen_2port is  
port (  
reset   : in std_logic;  
sel     : in std_logic;  
ri1     : in std_logic;  
ri2     : in std_logic;  
ai1     : out std_logic;  
ai2     : out std_logic;  
lclk    : out std_logic  
);  
end clk_gen_2port;
architecture rtl of clk_gen_2port is

component mutual_ex
port (  
reset   : in std_logic;
rs_in   : in std_logic;
rr_in   : in std_logic;
as_out : out std_logic;
ar_out : out std_logic
);
end component;

signal ri1a_s, ai1a_s, ri2a_s, ai2a_s : std_logic;
signal ai1b_s, ai2b_s : std_logic;
signal relk_s : std_logic;
signal lclk_s : std_logic := '0';
signal sel_s : std_logic;
signal relk_s1, clk_me : std_logic;

begin

uut_me1 : mutual_ex
port map (  
reset  => reset,
rs_in  => ri1a_s,
rr_in  => relk_s,
as_out => ai1a_s,
ar_out => ai1b_s
);

uut_me2 : mutual_ex
port map (  
reset  => reset,
rs_in  => ri2a_s,
rr_in  => relk_s,
as_out => ai2a_s,
ar_out => ai2b_s
);

ring_clk : process (lclk_s, sel_s, reset)
begin
  -- if reset /= '1' then
  --  rclk_s <= '0';
  --  lclk_s <= '0';
  --  sel_s <= '0';
  --  ai1a_s <= '0';
end process ring_clk;

end architecture;
--     ai1b_s <= '0';
--     ai2   <= '0';
--     ai1   <= '0';
if sel_s = '1' then
  rclk_s <= not (lclk_s) after 5 ns;  -- For multi-port
--     rclk_s <= not (lclk_s) after 1 ns; -- for rom point-topoint
elsif sel_s = '0' then
  rclk_s <= not (lclk_s) after 2.5 ns;  -- for multi-port
--     rclk_s <= not (lclk_s) after 0.75 ns; -- for rom point-to-point
else
  rclk_s <= '0';
end if;
end process ring_clk;

process (clk_me, rclk_s1 )
begin
if clk_me = '1' and rclk_s1 = '1' then
  lclk_s <= rclk_s1;
elsif clk_me = '0' and rclk_s1 = '0' then
  lclk_s <= '0';
else
  lclk_s <= lclk_s;
end if;
end process;

--   process (rclk_s, ai1b_s, ai2b_s)
--     begin
--       if (ai1b_s = '1') and (ai2b_s = '1') then
--         lclk_s <= rclk_s;
--       else
--         lclk_s <= '0';
--       end if;
--   end process;

clk_me <= '1' when (ai1b_s = '1' and ai2b_s = '1') else '0';

rclk_s1 <= rclk_s;
--     lclk_s <= rclk_s when (ai1b_s = '1' and ai2b_s = '1') else lclk_s;

sel_s  <= sel;
--     ri1b_s <= rclk_s;
--     ri2b_s <= rclk_s;
ri1a_s <= ri1;
ri2a_s <= ri2;
ai1    <= ai1a_s;
ai2    <= ai2a_s;
lclk   <= lclk_s;
--     ai1clk <= ai1b_s;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mutual_ex is
port (reset : in std_logic; rs_in : in std_logic; rr_in : in std_logic; as_out : out std_logic; ar_out : out std_logic);
end mutual_ex;

architecture behavioral of mutual_ex is

begin

process (rs_in, rr_in, reset)
begin
if reset /= '1' then
as_out <= '0';
ar_out <= '0';
elif rs_in = '0' and rr_in = '0' then
as_out <= '0';
ar_out <= '0';
elif rs_in = '1' and rr_in = '0' then
as_out <= '1';
ar_out <= '0';
elif rs_in = '0' and rr_in = '1' then
as_out <= '0';
ar_out <= '1';
elif rs_in = '1' and rr_in = '1' then
as_out <= '1';
ar_out <= '0';
elif rs_in = '1' and rr_in = '1' then
as_out <= '1';
ar_out <= '0';
else
as_out <= '0';
end if;
end process;

end mutual_ex;
ar_out <= '0';
end if;
end behavioral;

**VHDL Test Benches**

**Synchronous-Synchronous Point-to-Point Communication [TB – rom_dip_dop_tb]**

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY rom_dip_dop_tb_rom_vhd_tb IS
END rom_dip_dop_tb_rom_vhd_tb;

ARCHITECTURE behavior OF rom_dip_dop_tb_rom_vhd_tb IS

COMPONENT rom_dip_dop
PORT(
    ----------- I/P Port Signals -------------
    address : in std_logic_vector (1 downto 0);
    enable  : in std_logic;
    read    : in std_logic;
    reset   : in std_logic;
    ----------- O/P Port Signals -------------
    clk_ctrl_fin : out std_logic;
    pen_op_add_ctrl_fin : out std_logic;
    ri_op_add_ctrl_fin : out std_logic;
    ai_op_add_ctrl_fin : out std_logic;
    rp_add_fin : out std_logic;
    ap_add_fin : out std_logic;
    latch_in_add_fin : out std_logic_vector (3 downto 0);
    latch_out_add_fin : out std_logic_vector (3 downto 0);
    pen_ip_add_rom_fin : out std_logic;
    ri_ip_add_rom_fin : out std_logic;
    ai_ip_add_rom_fin : out std_logic;
    clk_rom_fin : out std_logic;
    pen_op_data_rom_fin : out std_logic;
    ri_op_data_rom_fin : out std_logic;
    ai_op_data_rom_fin : out std_logic;
    rp_data_fin : out std_logic;
    ap_data_fin : out std_logic;
    latch_in_data_fin : out std_logic_vector (3 downto 0);
    latch_out_data_fin : out std_logic_vector (3 downto 0);
)
pen_ip_data_ctrl_fin : out std_logic;
ri_ip_data_ctrl_fin : out std_logic;
ai_ip_data_ctrl_fin : out std_logic;
data_out : out std_logic_vector (3 downto 0);
);
END COMPONENT;

BEGIN

uut: rom_dip_dop PORT MAP(
address => address,
enable => enable,
read => read,
reset => reset,
clk_ctrl_fin => clk_ctrl_fin,
pen_op_add_ctrl_fin => pen_op_add_ctrl_fin,
ri_op_add_ctrl_fin => ri_op_add_ctrl_fin,
ai_op_add_ctrl_fin => ai_op_add_ctrl_fin,
pen_ip_data_ctrl_fin => pen_ip_data_ctrl_fin,
ri_ip_data_ctrl_fin => ri_ip_data_ctrl_fin,
ai_ip_data_ctrl_fin => ai_ip_data_ctrl_fin,
);
rp_add_fin => rp_add_fin,
ap_add_fin  => ap_add_fin,
latch_in_add_fin => latch_in_add_fin,
latch_out_add_fin => latch_out_add_fin,
pen_ip_add_rom_fin => pen_ip_add_rom_fin,
ri_ip_add_rom_fin => ri_ip_add_rom_fin,
ai_ip_add_rom_fin => ai_ip_add_rom_fin,
clk_rom_fin  => clk_rom_fin,
pen_op_data_rom_fin => pen_op_data_rom_fin,
ri_op_data_rom_fin => ri_op_data_rom_fin,
ai_op_data_rom_fin => ai_op_data_rom_fin,
rp_data_fin  => rp_data_fin,
ap_data_fin  => ap_data_fin,
latch_in_data_fin => latch_in_data_fin,
latch_out_data_fin => latch_out_data_fin,
pen_ip_data_ctrl_fin => pen_ip_data_ctrl_fin,
ri_ip_data_ctrl_fin => ri_ip_data_ctrl_fin,
ai_ip_data_ctrl_fin => ai_ip_data_ctrl_fin,
data_out    => data_out
);

th : PROCESS
BEGIN
reset      <= '0', '1' after 5 ns;

------ Test bench below to show normal functioning of DIP_DOP interface
address    <= "11", "00" after 3 ns, "01" after 5 ns, "10" after 8 ns,
            "11" after 12 ns, "00" after 16 ns, "01" after 18 ns;
enable     <= '0', '1' after 3 ns, '0' after 10 ns, '1' after 14 ns, '0' after 20 ns;
read       <= '0', '1' after 3 ns, '0' after 15 ns, '1' after 16 ns, '0' after 20 ns;

------ Test bench below to prove the working of a demand type I/P port
----- (lclk_rom is gated/stretched irrespective of the other port)
----- Use to compare D-type I/P with P-type I/P
-- address <= "11", "00" after 5 ns, "10" after 15 ns, "01" after 20 ns,
-- "11" after 27 ns, "00" after 36 ns;
-- enable <= '0', '1' after 5 ns, '0' after 15 ns, '1' after 25 ns;
-- read   <= '0', '1' after 5 ns, '0' after 60 ns;

wait;
END PROCESS;

END;
Synchronous-Synchronous Multi-Point Communication [TB – multi_port_tb]

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY multi_port_ram_tb_ram_vhd_tb IS
END multi_port_ram_tb_ram_vhd_tb;

ARCHITECTURE behavior OF multi_port_ram_tb_ram_vhd_tb IS

COMPONENT multi_port_ram
PORT(
reset : IN std_logic;
enable : IN std_logic;
write : IN std_logic;
read : IN std_logic;
write_addr : IN std_logic_vector(1 downto 0);
read_addr : IN std_logic_vector(1 downto 0);
data_in : IN std_logic_vector(3 downto 0);
clk_add_ctrl : OUT std_logic;
pen_add_ctrl : OUT std_logic;
ri_add_ctrl : OUT std_logic;
ai_add_ctrl : OUT std_logic;
rp_add_ctrl : OUT std_logic;
ap_add_ctrl : OUT std_logic;
add_ctrl_latch_in : OUT std_logic_vector(6 downto 0);
add_ctrl_latch_out : OUT std_logic_vector(6 downto 0);
clk_data_in : OUT std_logic;
pen_data_in : OUT std_logic;
ri_data_in : OUT std_logic;
ai_data_in : OUT std_logic;
rp_data_in : OUT std_logic;
ap_data_in : OUT std_logic;
data_in_latch_in : OUT std_logic_vector(3 downto 0);
data_in_latch_out : OUT std_logic_vector(3 downto 0);
clk_ram : OUT std_logic;
pen_ram_ip : OUT std_logic;
ri_ram_ip : OUT std_logic;
ai_ram_ip : OUT std_logic;
rp_ram_ip : OUT std_logic;
ap_ram_ip : OUT std_logic;
pen_ram_op : OUT std_logic;
ri_ram_op : OUT std_logic;
ai_ram_op : OUT std_logic;
rp_ram_op : OUT std_logic;
);
ap_ram_op   : OUT std_logic;
clk_data_out_1  : OUT std_logic;
pen_data_out_1  : OUT std_logic;
ri_data_out_1   : OUT std_logic;
ai_data_out_1   : OUT std_logic;
rp_data_out_1  : OUT std_logic;
ap_data_out_1  : OUT std_logic;
data_out_1_latch_in  : OUT std_logic_vector(3 downto 0);
data_out_1_latch_out : OUT std_logic_vector(3 downto 0);
clk_data_out_2  : OUT std_logic;
pen_data_out_2  : OUT std_logic;
ri_data_out_2   : OUT std_logic;
ai_data_out_2   : OUT std_logic;
rp_data_out_2  : OUT std_logic;
ap_data_out_2  : OUT std_logic;
data_out_2_latch_in  : OUT std_logic_vector(3 downto 0);
data_out_2_latch_out : OUT std_logic_vector(3 downto 0);
data_out_2   : OUT std_logic_vector(3 downto 0);
data_out_1   : OUT std_logic_vector(3 downto 0));
END COMPONENT;

SIGNAL reset   : std_logic;
SIGNAL enable   :  std_logic;
SIGNAL write   :  std_logic;
SIGNAL read    :  std_logic;
SIGNAL write_addr   :  std_logic_vector(1 downto 0);
SIGNAL read_addr   :  std_logic_vector(1 downto 0);
SIGNAL data_in   :  std_logic_vector(3 downto 0);
SIGNAL clk_add_ctrl  :  std_logic;
SIGNAL pen_add_ctrl  :  std_logic;
SIGNAL ri_add_ctrl   :  std_logic;
SIGNAL ai_add_ctrl   :  std_logic;
SIGNAL rp_add_ctrl   :  std_logic;
SIGNAL ap_add_ctrl   :  std_logic;
SIGNAL add_ctrl_latch_in  :  std_logic_vector(6 downto 0);
SIGNAL add_ctrl_latch_out  :  std_logic_vector(6 downto 0);
SIGNAL clk_data_in   :  std_logic;
SIGNAL pen_data_in  :  std_logic;
SIGNAL ri_data_in   :  std_logic;
SIGNAL ai_data_in   :  std_logic;
SIGNAL rp_data_in   :  std_logic;
SIGNAL ap_data_in   :  std_logic;
SIGNAL data_in_latch_in  :  std_logic_vector(3 downto 0);
SIGNAL data_in_latch_out  :  std_logic_vector(3 downto 0);
SIGNAL clk_ram   :  std_logic;
BEGIN

uut: multi_port_ram PORT MAP(
reset    => reset,
enable    => enable,
write    => write,
read    => read,
write_addr    => write_addr,
read_addr    => read_addr,
data_in    => data_in,
clk_add_ctrl    => clk_add_ctrl,
pen_add_ctrl    => pen_add_ctrl,
ri_add_ctrl    => ri_add_ctrl,
ai_add_ctrl    => ai_add_ctrl,
rp_add_ctrl    => rp_add_ctrl,
ap_add_ctrl    => ap_add_ctrl,
add_ctrl_latch_in    => add_ctrl_latch_in,
add_ctrl_latch_out  => add_ctrl_latch_out,
clk_data_in         => clk_data_in,
pen_data_in         => pen_data_in,
ri_data_in          => ri_data_in,
ai_data_in          => ai_data_in,
rp_data_in          => rp_data_in,
ap_data_in          => ap_data_in,
data_in_latch_in    => data_in_latch_in,
data_in_latch_out   => data_in_latch_out,
clk_ram             => clk_ram,
pen_ram_ip          => pen_ram_ip,
ri_ram_ip           => ri_ram_ip,
ai_ram_ip           => ai_ram_ip,
rp_ram_ip           => rp_ram_ip,
ap_ram_ip           => ap_ram_ip,
pen_ram_op          => pen_ram_op,
ri_ram_op           => ri_ram_op,
ai_ram_op           => ai_ram_op,
rp_ram_op           => rp_ram_op,
ap_ram_op           => ap_ram_op,
clk_data_out_1      => clk_data_out_1,
pen_data_out_1      => pen_data_out_1,
ri_data_out_1       => ri_data_out_1,
ai_data_out_1       => ai_data_out_1,
rp_data_out_1       => rp_data_out_1,
ap_data_out_1       => ap_data_out_1,
data_out_1_latch_in => data_out_1_latch_in,
data_out_1_latch_out=> data_out_1_latch_out,
clk_data_out_2      => clk_data_out_2,
pen_data_out_2      => pen_data_out_2,
ri_data_out_2       => ri_data_out_2,
ai_data_out_2       => ai_data_out_2,
rp_data_out_2       => rp_data_out_2,
ap_data_out_2       => ap_data_out_2,
data_out_2_latch_in => data_out_2_latch_in,
data_out_2_latch_out=> data_out_2_latch_out,
data_out_2           => data_out_2,
data_out_1           => data_out_1;

tb : PROCESS
BEGIN
reset      <= '0', '1' after 1 ns;
enable     <= '0', '1' after 5 ns, '0' after 30 ns;
write      <= '0', '1' after 5 ns, '0' after 20 ns;
             -- '1' after 25 ns, '0' after 30 ns;
write_addr <= "00", "11" after 5 ns, "10" after 10 ns, "01" after 15 ns, "00" after 20 ns;
  -- "11" after 25 ns, "10" after 30 ns;
  -- "01" after 50 ns, "00" after 55 ns;
data_in  <= "0000", "0001" after 5 ns, "0010" after 10 ns, "0100" after 15 ns,
  "1000" after 20 ns;
  -- "1001" after 25 ns;
  --, "1010" after 35 ns, "1011" after 45 ns, "1100" after 50 ns,
  -- "1101" after 55 ns, "1110" after 60 ns;
read    <= '0', '1' after 10 ns, '0' after 30 ns;
  --, '1' after 50 ns, '0' after 65 ns;
read_addr <= "00", "11" after 10 ns, "10" after 15 ns, "01" after 20 ns, "00" after 25 ns;
  -- "11" after 45 ns, "01" after 50 ns, "10" after 55 ns, "00" after 60 ns;
wait;
END PROCESS;
END;

Synchronous-Asynchronous Data Communication [TB – async_dop_tb]

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY async_dop_tb_async_dop_vhd_tb IS
END async_dop_tb_async_dop_vhd_tb;

ARCHITECTURE behavior OF async_dop_tb_async_dop_vhd_tb IS

COMPONENT async_dop
PORT(reset     : IN std_logic;
address     : IN std_logic_vector(1 downto 0);
add_valid   : IN std_logic;
clk         : OUT std_logic;
pen         : OUT std_logic;
ri          : OUT std_logic;
ai          : OUT std_logic;
rp          : OUT std_logic;
ad_latch_in : OUT std_logic_vector(1 downto 0);
ad_latch_out: OUT std_logic_vector(1 downto 0);
ap          : OUT std_logic;
data_latch_in : OUT std_logic_vector(3 downto 0);
data_latch_out: OUT std_logic_vector (3 downto 0);
data_out : OUT std_logic_vector(3 downto 0);
END COMPONENT;

SIGNAL reset : std_logic;
SIGNAL address : std_logic_vector(1 downto 0);
SIGNAL add_valid : std_logic;
SIGNAL clk : std_logic;
SIGNAL pen : std_logic;
SIGNAL ri : std_logic;
SIGNAL ai : std_logic;
SIGNAL rp : std_logic;
SIGNAL add_latch_in : std_logic_vector(1 downto 0);
SIGNAL add_latch_out : std_logic_vector(1 downto 0);
SIGNAL ap : std_logic;
SIGNAL data_latch_in : std_logic_vector(3 downto 0);
SIGNAL data_latch_out : std_logic_vector(3 downto 0);
SIGNAL data_out : std_logic_vector(3 downto 0);

BEGIN
  uut: async_dop PORT MAP(
    reset => reset,
    address => address,
    add_valid => add_valid,
    clk => clk,
    pen => pen,
    ri => ri,
    ai => ai,
    rp => rp,
    add_latch_in => add_latch_in,
    add_latch_out => add_latch_out,
    ap => ap,
    data_latch_in => data_latch_in,
    data_latch_out => data_latch_out,
    data_out => data_out
  );

  tb : PROCESS
  BEGIN
    reset <= '0', '1' after 2 ns;
    add_valid <= '0', '1' after 3 ns, '0' after 7 ns, '1' after 10 ns, '0' after 13 ns;
    address <= "00", "01" after 5 ns, "10" after 9 ns, "11" after 12 ns;
    wait; -- will wait forever
  END PROCESS;

END;
LIST OF REFERENCES


