INTERFACING AN IP WITH PAUSIBLE CLOCKING.
A case study for connecting IP to a Network on Chip.

K.M.M. van Kaam

Supervisor/Coach: prof.dr.ir. J.L. van Meerbergen
Date: June 2004
Abstract: In nowadays digital IC designs multiple processors, memories and peripherals are integrated and have become System on Chips (SoC). These SoCs typically have multiple clock domains. As a result there is an increasing interest in Globally Asynchronous Locally Synchronous (GALS) systems. The different modules in such a system are clocked locally and the communication between modules is asynchronous. Therefore synchronization techniques must be applied, which can be data or clock based. Furthermore due to the complexity of SoCs, the use of Networks on Chip (NoC) are also being proposed. NoC provide a flexible communication architecture and guarantee of services. They also reduce the need for long bus wiring.

Pausible clocking is a technique in which the clock generation is paused to synchronize data. The goal is to connect modules that use pausible clocking to the Åthereal Network on Chip. An obvious solution would be to use pausible clocking for the NoC as well. However this causes a performance degradation of the network and the guarantee of services might be lost. A solution must be found that keeps the benefits of using a NoC and enables the usage of pausible clocked modules interfaced to the Åthereal network.

The module that used pausible clocking is wrapped with an asynchronous shell consisting of a pausible clock generator and Input/Output controllers. Every I/O controller is interfaced to the pausible clock generator and to a ripple-through FIFO that uses transition signalling. These FIFOs are already used in the network as data queues. The asynchronous signals that arrive at the clock boundary of the network can be synchronized with data based synchronization techniques.

Signal Transition Graphs (STGs) are here used to formally proof the correct behaviour of the pausible clock generator and to reason about its timing. The I/O controllers are defined with STGs, which can be synthesized to speed independent Asynchronous Finite State Machines.

The pausible clock generator and the I/O controllers are implemented with standard cells in a 0.13 μm technology. A Mutual Exclusion element is designed as a standard cell, which is used in the pausible clock generator. The different asynchronous components are interfaced to make a simple communication channel, which is simulated with back-annotation from the layout.
Conclusions: The main conclusions are:

- Interfacing pausable clocked IP to a NoC without pausing the network clock is proven (with STGs) to be feasible.
- In the pausable clocked IP module metastability can be prevented during synchronization.
- The frequency of the pausable clock must be lower than the frequency of the network.
- The clock is paused when the IP module writes data to a full FIFO or when it reads data from an empty FIFO. If the control sequence of an I/O controller takes longer than a clock cycle the clock is paused with every write or read action.
- The data that is received by the IP module must be stored in an extra latch stage to keep the data stable for sampling. An efficient solution is adding a stage in the ripple-through FIFO for this purpose.
- To make the concept mature for products the concept must be incorporated in the design flow; possibilities for production testing must be added and full-custom implementation of the I/O controllers can be considered for better performance.
Contents

Glossary ................................................................. 9

Acknowledgements .................................................... 10

1. Introduction ....................................................... 11
   1.1. Problem description ........................................... 11
   1.2. Contents ..................................................... 12
      1.2.1. For the reader ............................................ 13

2. Asynchronous design ............................................. 15
   2.1. Handshaking .................................................. 15
   2.2. Modelling and synthesis techniques ........................ 16
      2.2.1. Signal Transition Graphs ............................... 16
      2.2.2. Generalized Signal Transition Graphs .................. 18
      2.2.3. Petriify ............................................... 18
   2.3. Asynchronous components ................................... 18
      2.3.1. Muller C-element ...................................... 18
      2.3.2. Generalized C-element ................................ 19
      2.3.3. Mutual Exclusion element .............................. 20

3. Synchronization .................................................. 22
   3.1. Metastability ............................................... 22
      3.1.1. Finding $T_m$ and $T_f$ for a flip-flop ............... 23
      3.1.2. Mean Time Before Failure ............................ 24
   3.2. Classification of signal to clock synchronization ........ 25
   3.3. Clock generation versus synchronization ............... 26
   3.4. Pausible clocking in GALS systems ....................... 27
   3.5. Dual flip-flop synchronizer ............................... 28
      3.5.1. MTBF for dual flip-flop synchronizers ............... 29

4. Top-level architecture ......................................... 32
   4.1. Network Interface .......................................... 32
      4.1.1. Kernel ............................................... 32
      4.1.2. Queues .............................................. 33
   4.2. MOUSETRAP FIFOs ......................................... 34
      4.2.1. MOUSETRAP FIFO in a synchronous environment ....... 35
   4.3. Interfacing pausible clocked IP modules ................. 36
   4.4. Synchronizers ............................................. 38

© Koninklijke Philips Electronics N.V. 2004
9.1. Consequences ................................................................. 77
9.2. Implementation............................................................. 78
9.3. Future work ................................................................. 78

References ............................................................................. 79

A Example Petrify: Muller C-element ....................................... 82
B Synthesising an output controller ........................................ 85
C Synthesising an input controller with auxiliary latches ............. 87
D Synthesising an input controller without auxiliary latches ........... 89
Glossary

AFSM - Asynchronous Finite State Machine
ASIC - Application Specific Integrated Circuit
BE - Best Effort
CAD - Computer Aided Design
CDC - Clock Domain Crossing
CMOS - Complementary Metal Oxide Semiconductor
CSC - Complete State Coding
DCO - Digitally Controlled Oscillator
DFT - Design for Test
DI - Delay Insensitive
DTL - Device Transaction Level
FIFO - First In First Out
FSM - Finite State Machine
GALA - Globally Asynchronous Locally Asynchronous
GALS - Globally Asynchronous Locally Synchronous
GALDS - Globally Asynchronous Locally Dynamic Systems
GS - Globally Synchronous
GT - Guaranteed Throughput
HPU - Header Parsing Unit
IC - Integrated Circuit
icfb - ic front to back
I/O - Input/Output
IoS - Islands of Synchronicity
IP - Intellectual Property
ITRS - International Technology Roadmap for Semiconductors
LS - Local Synchronous
ME - Mutual Exclusion (element)
MOS - Metal Oxide Semiconductor
MTBF - Mean Time Between Failures
NI - Network Interface
nMOS - n-channel MOS
NoC - Network on Chip
PCC - Pausible Clocking Control
pMOS - p-channel MOS
SAT - SoC Architecture and Technology
SB - Synchronous Block
SoC - System on Chip
SoR - Skeleton of ReUse
SS - Source Synchronous
STG - Signal Transition Graphs
TUBE - Tuning for Best Execution
VLSI - Very Large Scale Integration
Acknowledgements

This Report is the result of the work during my graduation project, which has been carried out within the group Digital Design and Test at Philips Research in Eindhoven.

I would like to thank prof. dr. ir. J.L. van Meerbergen of the Eindhoven University of Technology, dr. K.G.W. Goossens and dr. ir. A.A. Abbo (both with Philips Research Eindhoven) for supervising my graduation project. Also I would like to thank ir. John Dielissen for his support and fruitful discussions about the Network Interface of the Æthereal Network on Chip. Furthermore I would like to thank ir. Paul Wielage for his discussions about asynchronous FIFOs and metastability issues and Dr. Francesco Pessolano for sharing his knowledge about Signal Transition Graphs. And of course I would like to thank all other colleagues not mentioned here, who supported me with my graduation.

Kees van Kaam,

Eindhoven, May 2004
1. Introduction

The research carried out in the group Digital Design and Test (part of the IC design sector at the Philips High-Tech Campus) aims to enable the integration of System on Chips (SoC) in state of the art and future CMOS technologies. MoSAIC, which stands for Methodology for Silicon Architecting of ICs, aims improving the re-usability of IP modules and hence design productivity by means of a methodology that makes IP modules easier to integrate. An essential part of such a methodology is a flexible communication architecture. The goal of the Æthereal project [6][22] is to provide a Network on Chip (NoC) for this purpose. The graduation project described in this report is part of the MoSAIC and of the Æthereal projects.

1.1. Problem description

Figure 1.1 gives a schematic architecture of a SoC with several IP modules, which communicate via such a Æthereal network. The network is composed of routers and every IP module is interfaced to a router via a Network Interface (NI).

![Figure 1.1: SoC with several IP modules and a Æthereal network for on-chip communication](image)

In a SoC as given in Figure 1.1 there typically exist multiple clock domains, which can be completely independent. These multiple clock domains in a SoC can be caused by one or more of the following trends in IC design:

- In modern technology it is difficult to distribute a global high-speed low-skew clock signal for the complete chip.
- IP modules can be firm cores that can not be changed and therefore the clock frequency is predefined.
- IP modules might differ sensibly in clock frequency.
- The SoC uses dynamic frequency scaling for every IP module separately for dynamic power management.
The emerging of SoC with multiple clock domains makes Globally Asynchronous Locally Synchronous (GALS) systems a viable and interesting design choice. In GALS systems every IP module keeps its own local clock domain and the communication between the different IP modules is asynchronous. This communication is only possible when the necessary synchronization between these clock domains is achieved. Several synchronization solutions are available, which could be applied to the architecture in Figure 1.1. All synchronization techniques are either clock based or data based. One clock based technique, which is called pausible clocking is very interesting. The concept of synchronization by pausing the clock will be explained in Paragraph 3.4. This synchronization technique has several advantages over more conservative synchronization techniques:

- The chance of failure ($P_f$) due to metastability (see Paragraph 3.1) is zero, making the complete on-chip communication very robust. However non-determinism is introduced at system-level, which can cause problems for guarantees in the design.

- Asynchronous handshaking is used for data communication between IP modules, which have their own local (independent) clock domain. If frequencies are not predictable (e.g. if dynamic frequency scaling is used) the communication remains robust.

- Pausible clocking can be added to a Locally Synchronous (LS) module as a shell, making integration of soft cores as well as firm cores possible.

- Pausible clocking simplifies the use of dynamic frequency scaling.

For the architecture in Figure 1.1 the network must have a common clock frequency, because the Æthereal network uses time-division-multiplex circuit-switching, which can provide guaranteed services. All IP modules can have an independent pausible clock domain. However the network can not use pausible clocking, because the global communication of the SoC is stopped when the network clock is paused for synchronization with a certain IP module. This will result in a performance degradation of the network, and the guarantee of services is lost.

Thus the question (or problem) is how to connect the Network Interface of the Æthereal network to an IP module, when this IP module uses pausible clocking for synchronization and the clock of the network can not be paused. When a solution is found further questions are what the consequences are for a SoC and how this solution can be implemented.

### 1.2. Contents

After this general introduction the following two Chapters give more background information. Chapter 2 first gives background information about asynchronous design. The two main topics in this Chapter are Signal Transition Graphs and asynchronous components, which are used for the implementation. Chapter 3 gives background information about synchronization in general. A thorough explanation of metastability is given, and synchronization techniques using pausible clocking or dual flip-flop synchronizers are described.

In Chapter 4 the Network Interface (NI) of the Æthereal network is introduced. The ripple-through FIFOs, which are used in this NI are described in here as well. A top-level architecture for interfacing an IP module with a pausible clock domain to the Network Interface that has a non-pausible clock domain is introduced. The synchronizers that are necessary for this top-level architecture are also discussed in Chapter 4. In the following Chapters the components of this architecture are described.
Chapter 5 introduces a pausible clock generator. The timing of the basic circuit of the pausible clock generator is described and a solution for interfacing multiple I/O controllers to the pausible clock generator is given. Chapter 6 and Chapter 7 describe solutions for the output and input controllers respectively. The interfacing, the behaviour and the timing specification of these I/O controllers are described. The behaviour of the I/O controllers is specified with Signal Transition Graphs, which can be mapped to Asynchronous Finite State Machines. Chapter 8 gives the implementation of a Mutual Exclusion element, the pausible clock generator and the I/O controllers in a CMOS12 process. The report ends with conclusions in Chapter 9.

1.2.1. For the reader

Throughout this report signal names and special names for components are given in italics. For example when the signal with the name "accept" is used in a sentence it is written as: accept.

When Boolean functions are used throughout this report then the following syntax for these functions is used:

- Negation (NOT): the variable that is negated has an apostrophe behind its variable name (e.g. var').
- AND: there is no special character or an asterisk (*) between variables (e.g. var1 var2).
- OR: there is a plus sign between variables (e.g. var1 + var2).

The next Figures give the meaning of the symbols of the logic gates and sequential elements that are used in the schematics in this report.

![Input port and Output port](image)

*Figure 1.2: input port (left) and output port (right)*

![Inverter and Buffer](image)

*Figure 1.3: inverter (left) and buffer (right)*

![OR gate and NOR gate](image)

*Figure 1.4: OR gate (left) and NOR gate (right)*

![XOR gate and XNOR gate](image)

*Figure 1.5: Exclusive OR gate (left) and Exclusive NOR gate (right)*
Figure 1.6: AND gate (left) and NAND gate (right)

Figure 1.7: Muller C-element (left) and Muller C-element with inverted output (right)

Figure 1.8: delay-line with delay $\tau$

Figure 1.9: latch with enable signal and data input and output

Figure 1.10: D flip-flop with clock input (clk), data input (D) and data output (Q); optional are the inverted data output ($Q'$) and the asynchronous reset (clear)
2. Asynchronous design

This chapter gives a brief summary of asynchronous design techniques, asynchronous components and asynchronous circuit modelling with Signal Transition Graphs (STGs) that are used later on in this report. An overview of different asynchronous design techniques is given in [28] and [9].

Asynchronous design does not use a clock signal as is used in synchronous design. This gives a number of advantages of which the most important advantages are given below:

- Lower power consumption: in synchronous design every memory element switches every clock cycle, thus also when the data is not changed. In asynchronous design the memory element only switches when new data is received.
- Less emission of electro-magnetic noise: the switching of the logic and memory elements of asynchronous designs is more spread out than in synchronous designs.
- No clock distribution and clock skew problems, because there is no global clock signal.
- High operation speed: the operation speed is determined by the delays in the circuit. This is the same as in synchronous design, however in synchronous design the longest path (or the longest delay) determines the maximum clock frequency and with this the speed. In asynchronous design every delay is local, which means that there is never more time used then the delay of the local path itself.

Despite these advantages, asynchronous design is normally adapted. The main reason is that the available tools for making asynchronous systems is limited.

2.1. Handshaking

To pass data in an asynchronous system, handshaking is used. Different protocols exist, but here only the bundled-data protocol is discussed. The bundled-data protocol refers to a situation where the data signals use normal Boolean levels to encode information. Bundled-data protocols need a single request signal (Req) and a single acknowledge signal (Ack). These two signals are bundled with a number of data signals. When data is send there is always an initiator and a target and there is a sender and a receiver. When the initiator is the sender and the target is the receiver the channel is called a push channel. When the initiator is the receiver and the target is the sender the channel is called a pull channel. Both channels are shown in Figure 2.1, a black dot indicates the initiator.

![Push channel and pull channel for bundled-data protocols](image)

*Figure 2.1: push channel and pull channel for bundled-data protocols*
Figure 2.2 shows the behaviour of a 4-phase and a 2-phase bundled-data protocol (both push channels). The numbers 4 and 2 indicate the number of communication actions necessary for one cycle of the protocol. The four steps of the 4-phase protocol are:

- **Sender**: make the data valid and make the *Req* signal true.
- **Receiver**: when the request signal is received (and thus the data), store the data and make the *Ack* signal true to acknowledge the request.
- **Sender**: when the acknowledge signal is received, make the *Req* signal false.
- **Receiver**: when the request signal is received, make the *Ack* signal false.

The 2-phase bundled-data protocol uses transition signalling and therefore only 2 communication actions are necessary for one cycle of the protocol. The 2-phase protocol is therefore more efficient in time then the 4-phase protocol.

![Diagram showing 4-phase and 2-phase protocol](image)

**Figure 2.2: 4-phase and 2-phase bundled data protocol**

### 2.2. Modelling and synthesis techniques

Several asynchronous design methodologies exist and the most important ones are given in [9]. In this report the modelling of the asynchronous components is done with Signal Transitions Graphs (STGs). STGs make reasoning about the behaviour and timing of the circuit easy. Furthermore STGs (if they comply to certain rules) can be synthesised into equations with the tool *Petrify*, which is developed at the *Universitat Politècnica de Catalunya*.

#### 2.2.1. Signal Transition Graphs

The formal representation of a STG is the 5-tuple $\text{STG} = (A, P, T, F, m_0)$, with:

- **A**: the set of all signals in the STG.
- **P**: the set of all places in the STG.
- **T**: the set of all transitions in the STG.
- **F**: the flow relation of the STG; $F \subseteq (P \times T) \cup (T \times P)$
- **$m_0$**: the set of tokens that indicate the initial net marking.

Figure 2.3 shows a simple example of a STG with the corresponding 5-tuple. The transitions and the places (in the example there is only one place $p_1$) are the vertices in the graph and the arcs are described with the flow relation $F$. Possible transitions are a falling transition (−), a rising transition (+) or either (*). Place $p_1$ holds a token (the black dot) and the initial net marking is
therefore \{p_1\}. The net marking can be changed by executing actions (or firing transitions) according to the following basic rules:

- A transition is enabled in a marking if each of its inputs has one token, this is denoted by \text{m}(t) (with \text{t} \in T).
- Any enabled transition can occur, and its firing is represented by removing a token from each of the corresponding inputs and inserting a new token in each of its outputs. The transformation of \text{m} into \text{m}' by firing a transition \text{t} is denoted by \text{m}(t)\text{m}'.
- Enabled transitions can occur concurrently as long as they are independent.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{stg_example.png}
\caption{Example of a STG.}
\end{figure}

Instead of using the denotation \text{m}(t)\text{m}', the denotation \text{m}' = \delta(m, t) can be used. With \delta a partial function \text{M} \times \text{T} \rightarrow \text{M}, where \text{M} denotes the set of all markings. These two notations can be executed by using a sequence of transitions \text{\sigma} instead of a single transition \text{t}. Thus the denotation \text{m}(\text{\sigma})\text{m}' or \text{m}' = \delta(m, \text{\sigma}) is the transformation of \text{m} into \text{m}' by firing the sequence of transitions \text{\sigma}.

STGs are 1-bounded (safety), which means that there is only one token allowed in a place. And in STGs the selection among alternatives must only be controlled by mutually exclusive inputs. Both constraints implicate that every STG is deterministic by nature. If a circuit must be generated out of a STG then one or more of the following restrictions must be met as well:

- Liveness: from every reachable marking, every transition can be fired.
- Persistency: for all arcs \text{a*} \rightarrow \text{b*} in the STG (* means + or -), there must be other arcs that ensure that \text{b*} fires before the opposite transition of \text{a*}.
- Consistent state assignment: every transition of a signal should alternate between rising and falling transitions.
- Unique state assignment: no two different markings of the STG have identical values for all signals.
- Single-cycle transitions: each signal in the STG appears in exactly one rising and one falling transition.
2.2.2. Generalized Signal Transition Graphs

A generalization of STGs is introduced in [29], because some applications are hard to be described with STGs (e.g. arbiters). Two extensions in this generalized STG model are used in this report for STGs that do not have to be synthesized. These two extensions are:

- The use of non-free-choice nets in the STG (see Figure 2.4). Thus allowing arcs departing from the same place to different transitions and these transitions have other arcs arriving at these transitions.
- The use of Boolean Guards at the arcs. These Boolean Guards are used as an extra mechanism for the firing of transitions.

![Diagram of a STG, which is a non-free-choice](image)

*Figure 2.4: part of a STG, which is a non-free-choice*

2.2.3. Petrify

As previously mentioned Petrify is developed at the Universitat Politècnica de Catalunya and it can be used to design asynchronous circuits. The input of Petrify is a Signal Transition Graph. Petrify can generate State Graphs and can solve the Complete State Coding (CSC) problem. CSC means that there is no state with the same state coding and a different next state. To explain how to work with Petrify a small example with a Muller C-element (this component is discussed in Paragraph 2.3.1) is given in Appendix A. The example has as input the STG of a Muller C-element and has as output an Equation file and State Graphs. Petrify can generate equations for speed-independent circuits with generalized C-elements (these components are discussed in Paragraph 2.3.2). A speed-independent (SI) circuit is a circuit that operates correctly assuming positive, bounded but unknown delays in gates and ideal zero-delay in wires. A complete tutorial of Petrify can be found in [4].

2.3. Asynchronous components

Some components used in asynchronous design are rarely used in synchronous design and are therefore also not commonly known in the world of synchronous IC design. The components used further on in this report are for completeness given in the next paragraphs. Starting with the description of the standard Muller C-element followed by the generalized C-element. The third and last component described is the mutual exclusion element.

2.3.1. Muller C-element

The Muller C-element is an important component in asynchronous design. It is used for data holding and synchronization purposes. The symbol and the schematic of a two input C-element is shown in Figure 2.5. The behaviour of the C-element in words is as follows:

- If both inputs $a$ and $b$ are true then output $z$ is true.
• If both inputs $a$ and $b$ are false then output $z$ is false.
• If inputs $a$ and $b$ are different then output $z$ holds its last value.

![Figure 2.5: symbol for a Muller C-element and the schematic for a dynamic implementation](image)

The behaviour can be easily matched with the schematic of the dynamic implementation of the C-element. When inputs $a$ and $b$ are both true the intermediate node $i$ becomes false and the output $z$ becomes true. When inputs $a$ and $b$ are both false the intermediate node $i$ becomes true and the output $z$ becomes false. When inputs $a$ and $b$ are different the node $i$ does not change and so does $z$. The two inverters (keeper element) makes sure that the value of $z$ keeps stable. The behaviour is also given in a STG in Figure 2.6. There is assumed in the STG that inputs $a$ and $b$ do not fall or rise after its transition until the output falls or rises. This assumption is not necessary for the inputs of the C-element, but this assumption makes the STG synthesisable to a Muller C-element.

![Figure 2.6: STG for a Muller C-element](image)

### 2.3.2. Generalized C-element

The behaviour of a Muller C-element can be defined in a more general manner. The output of a Muller C-element is dependent of two functions: a set and a reset function. For the Muller C-element the functions are:

- Set = $a \cdot b$
- Reset = $a' \cdot b'$

Any Boolean function can be used for the set and reset function. The components that use more general set and reset functions are called generalized C-elements. The schematic of these
elements with an example are shown in Figure 2.7. In the example the output \( z \) becomes true if inputs \( a \) and \( b \) are true. The output \( z \) becomes false if inputs \( b \) and \( c \) are false. Of course the set and the reset function are not allowed to be true for the same input values, because otherwise there will be a short circuit between the power supply and the ground. The operation of the circuit of generalized C-elements is similar to the Muller C-element.

\[
\begin{align*}
\text{set}(z) &= ab \\
\text{reset}(z) &= b'c'
\end{align*}
\]

*Figure 2.7: the common circuit and an example of a generalized C-element*

### 2.3.3. Mutual Exclusion element

A Mutual Exclusion element is mostly used for arbitration purposes. A schematic and the symbol of a Mutual Exclusion element (also called Mutex) is given in Figure 2.8. The Mutex has two inputs, in the schematic called \( R1 \) and \( R2 \), and two outputs, in the schematic called \( G1 \) and \( G2 \). The behaviour in words is (see also the timing diagram in Figure 2.9):

- When \( R1 \) rises and \( R2 \) is false the output \( G1 \) rises after a certain propagation delay.
- When \( R2 \) rises and \( R1 \) is false the output \( G2 \) rises after a certain propagation delay.
- When \( R1 \) and \( R2 \) rise at (nearly) the same moment, then the Mutex decides which output will rise.

*Figure 2.8: schematic and symbol of a Mutual Exclusion element (Mutex)*

The circuit of Figure 2.8 uses two cross-coupled NAND gates to set or reset one of the nodes \( x1 \) and \( x2 \). When \( R1 \) becomes true and \( R2 \) is false then node \( x1 \) becomes false and \( x2 \) remains true. Now node \( x1 \) will open the pMOS transistor at the bottom of the circuit and output port \( G1 \) will rise because node \( x2 \) is true. Furthermore node \( x2 \) will open the nMOS transistor at the top and
will pull output \( G2 \) to the ground, thus \( G2 \) is false. The inverse behaviour happens when \( R2 \) becomes true and \( R1 \) is false.

When input \( R1 \) and \( R2 \) rise at (nearly) the same moment in time then there will occur metastability (see Paragraph 3.1) at the outputs of the cross-coupled NAND gates. When metastability occurs the nodes \( x1 \) and \( x2 \) will have an undefined value, typically the voltage of the nodes will be half of the supply voltage. To prevent that the outputs of the Mutex also have undefined values the 4 transistors “filter” out the undefined values of \( x1 \) and \( x2 \). The metastability will end when the voltage difference of the nodes \( x1 \) and \( x2 \) reaches the thresholds of the transistors in the filter circuit. When this happens depends on the noise in the circuit and the arrival times of the input signals \( R1 \) and \( R2 \).

![Figure 2.9: timing diagram of a Mutual Exclusion element (including internal signals)](image)

To illustrate the behaviour of the Mutex somewhat further, the (generalized) STG for a Mutex is given in Figure 2.10. The STG uses a non-free choice net for the place in the middle of the STG. The token in this place will be “stolen” by transition \( R1+ \) or \( R2+ \), dependent on which transitions fires first. For example if \( R1+ \) fires, the token of the place in the middle is taken. Next if now transition \( R2+ \) fires then the transition \( G2+ \) can only fire when the token is put back in the place in the middle (when transition \( G1- \) has fired). The token is taken again when transition \( G2+ \) fires.

![Figure 2.10: STG of a Mutual Exclusion element](image)
3. Synchronization

This chapter first gives a thorough description of metastability. In synchronous design metastability is prevented by using worst case timing constraints on the design. Furthermore static timing analysis is applied when the design is finished and if problems are reported these should be fixed in the design. Synchronizers can be used to synchronize asynchronous signals, which are received in a synchronous design. A single flip-flop can be used as a synchronizer, but as is described in this Chapter a single flip-flop is not sufficient for a robust synchronization. When asynchronous signals must be sampled by synchronizers, such that the sampled signals can be used in a synchronous design, metastability can occur in these synchronizers. If this happens there is a chance that the device malfunctions.

A classification of signal to clock synchronization is given along with a matrix that gives the different clock generation techniques versus different synchronization techniques. Two important synchronization techniques are described in this chapter: synchronization with pausable clocking and the usage of dual flip-flop synchronizers.

3.1. Metastability

Basically if a sequential gate with a data and a sampling input has a transition on both these inputs at the same time, metastability occurs. “At the same time” is not really a good definition. The time window in which metastability occurs (Metastability Window: $T_w$) is a characteristic of the design of the sequential gate. When metastability occurs the sequential gate may remain in an in-between state, called the "metastable state", for an indeterminate time interval. Eventually, the output of the sequential element settles to a logical zero or a logical one. While it is settling, its output may glitch, oscillate, sit at an intermediate voltage, or merely show an increased clock-to-output delay. The time needed to resolve from the metastable state is called $\tau$ (Resolve Time Constant) and this constant is a characteristic of the implementation. The constants $T_w$ and $\tau$ are difficult to determine, the best way is by doing measurements on the sequential gate itself and determine the statistical properties [7]. Another approach is by doing simulations of the sequential gate, but this is less accurate. Figure 3.1 shows a simulation of two cross-coupled NAND gates in a 0.13 $\mu$m process technology (CMOS12) forming a Set/Reset latch (this circuit is already discussed in Paragraph 2.3.3 for the Mutual Exclusion element).

![Figure 3.1: simulation of metastability for two cross-coupled NAND gates](image)
Both data inputs (the Set and Reset) get a rising transition close to each other. It shows that the closer the transitions are the longer it takes to get a logical one or a logical zero at the output.

The probability of a metastable event lasting longer than some time, $t$, is defined as:

$$P = e^{-t/\tau}$$  \hfill (1)

### 3.1.1. Finding $T_w$ and $\tau$ for a flip-flop

A D-flip-flop with asynchronous reset from the CMOS12 standard cell library (type df2sqx05) is used to find the values for $T_w$ and $\tau$ for this flip-flop. To find the values the $P$star simulator is used on the netlist of this flip-flop with all parasitic components (abstracted netlist of the layout of the flip-flop). During the simulation the data arrival time is swept around the setup time interval. Figure 3.2 shows a simulation result for some different input data signals. The data signals with delay values -80 ps, -68 ps and -66 ps are still early enough to be clocked into the flip-flop, the others are too late and do not change the output $Q$. The data signals that can change the output introduce a larger propagation delay, which is caused by the metastability occurring in the first latch stage of the flip-flop.

![Figure 3.2: setup metastability simulation for a flip-flop](image)

The used flip-flop has a load of only 6 fF in the simulation, which makes the propagation delay of the flip-flop as small as possible. The propagation delay for the data to output transition for a low to high transition found in the datasheet is 171 ps. Furthermore the setup time of the flip-flop for a logic one at the data input is 72.1 ps, which matches with the simulation results shown in Figure 3.2.

The different arrival times of the data are plotted versus the propagation delay of the flip-flop in Figure 3.3. The plot is only made for the setup interval and not for the hold interval. The hold interval looks the same as the setup interval, but it is flipped vertically [7]. Now an estimate of the Metastability Window $T_w$ and the Resolve Time Constant $\tau$ can be made. The normal propagation time of the flip-flop should be about 171 ps. Take the maximum propagation time without metastability to happen to be 200 ps, then the Metastability Window for the setup
interval is about 25 ps. For the complete Metastability Window (including the hold interval) this value is doubled, which equals 50 ps. The Resolve Time Constant is estimated to be 150 ps. This value is determined by finding the maximum additional propagation delay for the Metastability Window. To get better values for the Metastability Window and the Resolve Time Constant the flip-flop should be measured.

![Propagation delay vs. data arrival time](image)

**Figure 3.3: propagation delay of a flip-flop versus the data arrival time**

### 3.1.2. Mean Time Before Failure

Eventually metastability can cause malfunctioning of the device. To give a prediction for the probability of failures, Mean Time Before Failure (MTBF) is used. For a single-stage synchronizer (D flip-flop) and a simple model of a design (see Figure 3.4), the MTBF is defined as follows:

\[
MTBF_1 = \frac{\frac{t_{r1}}{\tau}}{f_c \cdot f_d \cdot T_w} - \frac{\frac{t_{r2}}{\tau}}{f_c \cdot f_d \cdot T_w}
\]

(2)

![Single stage synchronizer](image)

**Figure 3.4: single stage synchronizer (D flip-flop)**

With the constants \(\tau\) and \(T_w\) defined as described in the previous Paragraph. And furthermore:

- \(f_c\): the clock frequency of the flip-flops (A).
• $f_d$: the number of data events per second (B).
• $t_{r1}$: the time available for metastability to be resolved for $logic_1$.
• $t_{r2}$: the time available for metastability to be resolved for $logic_2$.

This formula calculates the number of faults that can happen due to data inconsistency in the sampled values after $logic_1$ and $logic_2$. For example if metastability does not propagate through $logic_2$, but it propagates through $logic_1$, there is an inconsistency between the two data values sampled by the flip-flops in the receiver.

The variables $t_{r1}$ and $t_{r2}$ are dependent of a number of timing parameters. First of all the clock frequency of the flip-flop determines the maximum time available before new data is clocked into the next flip-flop. This maximum time is reduced by the setup time of the flip-flop in the receiver, the propagation delay of the flip-flop $FF$, the propagation delay of the wires and the propagation delay of the logic between two flip-flop stages. Thus the formulas for the resolving time $t_{r1}$ and $t_{r2}$ are:

\[
tr_1 = \frac{1}{f_c} - t_{p\_ff} - t_{p\_logic1} - t_{p\_wire1} - t_{setup}
\]  \hspace{1cm} (3)

\[
tr_2 = \frac{1}{f_c} - t_{p\_ff} - t_{p\_logic1} - t_{p\_wire2} - t_{setup}
\]  \hspace{1cm} (4)

With:
• $t_{p\_ff}$: the propagation delay of flip-flop $FF$.
• $t_{p\_logic1}$: the propagation delay of $logic_1$.
• $t_{p\_wire1}$: the wire delay through $logic_1$.
• $t_{p\_logic2}$: the propagation delay of $logic_2$.
• $t_{p\_wire2}$: the wire delay through $logic_2$.
• $t_{setup}$: the setup time of the flip-flops in the receiver.

### 3.2. Classification of signal to clock synchronization

The different signal-to-clock synchronizations can be classified according to the characteristics of the phase and frequency dependencies. These dependencies are given in Table 3.1 and the descriptions of the different types of signal-to-clock synchronization are given below:

- Synchronous: data is sampled at the same frequency and with the same phase for the sending and receiving clock domain. (Typically the sending and the receiving clock domain are the same clock domain.)
- Mesochronous: data is sampled at the same frequency, but the clock phase between the sending and receiving clock domains has a constant phase difference.
- Multi-synchronous: the same as mesochronous, but the phase differences is variable.
• Plesiochronous: the sending and the receiving clock domains have nearly the same frequency. If the sending domain sends data to the receiving domain the phase of the data varies with the receiving clock domain.

• Periodic: a periodic signal is synchronized to any periodic clock.

• Asynchronous: the frequency and the phase of the receiving and sending clock domain are completely independent.

More detailed information about this classification can be found in [5].

Table 3.1: classification of signal to clock synchronisation

<table>
<thead>
<tr>
<th>Classification</th>
<th>Periodic</th>
<th>( \Delta f )</th>
<th>( \Delta \phi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>Yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mesochronous</td>
<td>Yes</td>
<td>( \phi _m )</td>
<td>0</td>
</tr>
<tr>
<td>Multi-synchronous</td>
<td>Yes</td>
<td>Varies</td>
<td>0</td>
</tr>
<tr>
<td>Plesiochronous</td>
<td>Yes</td>
<td>Varies</td>
<td>( f_g &lt; \varepsilon )</td>
</tr>
<tr>
<td>Periodic</td>
<td>Yes</td>
<td>Periodic variation</td>
<td>( f_g &gt; \varepsilon )</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

3.3. Clock generation versus synchronization

The matrix shown in Figure 3.5 illustrates the different clock generation techniques versus the different data synchronization techniques used in SoC design. The y-axis defines the different clock generation techniques and consists of the following set:

• Local clock generation with ring oscillators.

• Local clock signals with different frequency derived from a master clock.

• Local clock signals with the same frequency but different phase derived from a master clock.

• Single global clock signal.

The x-axis defines the different synchronization techniques and consists of the following set:

• Synchronizers (data based).

• Synchronizers (clock based).

• Handshake.

• Arbitration based.

• Controlled delay based.

The arbitration-based synchronization is introduced to indicate that the pausable clock synchronization technique is not only using handshaking, but also clock arbitration. In the next Paragraphs the pausable clocking scheme and the dual flip-flop synchronizer are described. Details about the different synchronization techniques can be found in [5][19][20], more specialized information about periodic synchronizers can be found in [9]. The synchronous Waterfall clocking scheme is covered in [12]. The other synchronous clocking schemes: using
H/X clock trees, using PLLs and using DLLs are standard approaches for distributing a global clock signal in the design.

![Diagram of clock generation and synchronization techniques](image)

**3.4. Pausible clocking in GALS systems**

The concept of Globally Asynchronous Locally Synchronous (GALS) is introduced by Chapiro [3], but the usage of pausible clocking for systems is first introduced by Yun [34][35]. The usage of pausible clocking in GALS evolved further and Borman [1] introduced an asynchronous wrapper for pausible clocking to be used in GALS systems. Although the concept of pausible clocking seems to be very promising, the usage in real products is limited. Muttersbach et. al. implemented pausible clocking in several designs [16][17][32]. An overview of different GALS systems and the universities, research institutes and companies that work on GALS systems with pausible clocking can be found in [12].

In general the approach for GALS systems with pausible clocking is wrapping a Local Synchronous (LS) module with asynchronous components including the pausible clock generator (see Figure 3.6). The LS module is then first designed in a standard synchronous manner. A pausible clock generator is added for every Local Synchronous (LS) module, which can pause the clock signal for synchronizing data, which is exchanged with another LS module. The controllers that control the data transfer are asynchronous and use handshaking between different LS modules (every type of handshaking can be used). The implementation of the I/O controllers and the pausible clock generator can be done in several manners, dependent of the chosen asynchronous design techniques and handshaking techniques. A thorough description about the behaviour and timing of a pausible clock generator is given in Chapter 5.

Next an example of communicating data in a push channel with using 4-phase handshaking (with bundled data) is described.

© Koninklijke Philips Electronics N.V. 2004
When a LS module wants to send data (the initiator) to another LS module (the target), the initiator requests to send data to its asynchronous output controller. Then the output controller requests to pause the generation of the clock signal and sends a request signal to the target. This request signal is received by the target at the input controller and this controller requests to pause the generation of the clock signal of the target. This is acknowledged back to the initiator and the initiator sends the data and an acknowledgement of this action. When this acknowledgement is received by the target the clock generation of the target can be started again and the data is sampled. An acknowledge signal is send back to the initiator, which can then safely start the clock generation again.

![Figure 3.6: asynchronous wrapper for GALS systems with the usage of pausable clocking](image)

### 3.5. Dual flip-flop synchronizer

For a single flip-flop synchronizer the formula for the MTBF was presented in Paragraph 3.1.2. With the values found for the Metastability Window (50 ps) and the Resolve Time Constant (150 ps) the MTBF can be calculated for different values of the clock frequency of the receiver. A fixed data event frequency of 10 Mhz is taken and the setup time of the flip-flop used is 70 ps. The simple model for the design is taken as shown in Figure 3.4. The delay through logic_1 (including wires) is taken as 50% of the clock cycle and the delay through logic_2 (including wires) is taken as 60% of the clock cycle. Furthermore there is assumed that the data input signal is asynchronous, thus this signal has no relation with the clock signal of the receiver and is not periodic either. The resulting MTBF values for these parameters and the simple model of the design are given in Table 3.2. With increasing clock frequencies the MTBF gets worse, for example at 200 MHz clock frequency a failure happens every 29 seconds. For the 500 MHz clock frequency there are about 2000 failures every second.

<table>
<thead>
<tr>
<th>f [MHz]</th>
<th>f [MHz]</th>
<th>t_1 [ns]</th>
<th>t_2 [ns]</th>
<th>MTBF [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>10</td>
<td>9.73</td>
<td>7.73</td>
<td>5.93E+23</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>4.73</td>
<td>3.73</td>
<td>9.90E+08</td>
</tr>
<tr>
<td>200</td>
<td>10</td>
<td>2.23</td>
<td>1.73</td>
<td>2.86E+01</td>
</tr>
<tr>
<td>300</td>
<td>10</td>
<td>1.40</td>
<td>1.06</td>
<td>7.24E-02</td>
</tr>
<tr>
<td>400</td>
<td>10</td>
<td>0.980</td>
<td>7.30</td>
<td>3.25E-03</td>
</tr>
<tr>
<td>500</td>
<td>10</td>
<td>0.730</td>
<td>5.30</td>
<td>4.63E-04</td>
</tr>
<tr>
<td>600</td>
<td>10</td>
<td>0.563</td>
<td>3.97</td>
<td>1.18E-04</td>
</tr>
<tr>
<td>700</td>
<td>10</td>
<td>0.444</td>
<td>3.01</td>
<td>4.23E-05</td>
</tr>
</tbody>
</table>
Obviously a single flip-flop cannot be used as a synchronizer between two independent clock domains. Therefore a dual flip-flop synchronizer is normally used in multiple clock systems, see Figure 3.7. Two disadvantages of this synchronizer are:

- There is still a chance of metastability.
- An additional latency for synchronization is paid.

![Figure 3.7: dual flip-flop synchronizer](image)

The chance of metastability can be reduced further by using more flip-flops in serial, but at the expense of latency. There is always a trade-off between robustness and the latency of the synchronizer. Sometimes the design of the dual flip-flop is altered by designers to decrease the latency, but this has a negative effect on the robustness of the synchronizer (the chance of metastability is increased). An example of this is given in [10].

### 3.5.1. MTBF for dual flip-flop synchronizers

The MTBF for a dual flip-flop synchronizer and the simple model of a design (see Figure 3.7) is defined similarly as for a single flip-flop synchronizer:

\[
MTBF = \left| \frac{t_{f1} / e_{r1}}{f_e \cdot f_{a1} \cdot T_w} - \frac{t_{f2} / e_{r2}}{f_e \cdot f_{a2} \cdot T_w} \right|
\]  

(5)

However the parameters \(f_{a1}\) and \(f_{a2}\) are not known in advance, but it is assumed that they give the probability that the first flip-flop has not settled by one setup time before the clock of the second flip-flop. This means that \(f_{a1}, f_{a2}, t_{r1}\) and \(t_{r2}\) can be rewritten to:

\[
f_{a1} = \frac{1}{MTBF_1}
\]  

(6)

\[
f_{a2} = \frac{1}{MTBF_2}
\]  

(7)

\[
t_{r1} = t_{r2} = \frac{1}{f_e} - t_{smp} \cdot f_{a2}
\]  

(8)

© Koninklijke Philips Electronics N.V. 2004
Now the resulting formula for the MTBF of a dual flip-flop synchronizer can be derived:

\[
MTBF = \left[ \frac{1}{e} \frac{1}{t_{\text{setup},ff}} \right]^2 - \left[ \frac{1}{e} \frac{1}{t_{\text{setup},ff}} \right]^2 = \left[ \frac{1}{f_c} \frac{1}{t_{\text{setup},ff}} \right]^2 - \left[ \frac{1}{f_c} \frac{1}{t_{\text{setup},ff}} \right]^2
\]

With \( t_{r,\text{logic}1} \) and \( t_{r,\text{logic}2} \) defined as:

\[
t_{r,\text{logic}1} = 2 \cdot \frac{2}{f_c} \cdot \frac{1}{t_{\text{setup},ff} - t_{p,ff} - t_{p,\text{wire}1} - t_{p,\text{logic}1}}
\]

\[
t_{r,\text{logic}2} = 2 \cdot \frac{2}{f_c} \cdot \frac{1}{t_{\text{setup},ff} - t_{p,ff} - t_{p,\text{wire}2} - t_{p,\text{logic}2}}
\]

This method can be repeated for every depth (number of flip-flops in serial) of the synchronizer. In general if the synchronizer has \( N \) flip-flops in serial, the MTBF can be formulated as:

\[
MTBF(N) = \left[ \frac{1}{e} \frac{1}{t_{\text{setup},ff}} \right]^2 - \left[ \frac{1}{e} \frac{1}{t_{\text{setup},ff}} \right]^2 = \left[ \frac{1}{f_c} \frac{1}{t_{\text{setup},ff}} \right]^2 - \left[ \frac{1}{f_c} \frac{1}{t_{\text{setup},ff}} \right]^2
\]

With:

\[
t_{r,\text{logic}1}(N) = \frac{N}{f_c} - t_{p,ff} - t_{p,\text{wire}1} - t_{p,\text{logic}1} - \sum_{i=1}^{N} t_{\text{setup},i}
\]

\[
t_{r,\text{logic}2}(N) = \frac{N}{f_c} - t_{p,ff} - t_{p,\text{wire}2} - t_{p,\text{logic}2} - \sum_{i=1}^{N} t_{\text{setup},i}
\]

By increasing the number of flip-flop stages in the synchronizer the MTBF increases, but also the area and the latency of the synchronizer increases. The MTBF for different clock frequencies for the dual flip-flop synchronizer is given in Table 3.3. The values are calculated with the same parameter values as for the single flip-flop synchronizer. Remember that the calculated values for the MTBF are just an indication, because the values for the Metastability Window and the Resolve Time Constant are estimates and a simple model of a design taken.

Compared to the single flip-flop synchronizer the MTBF of the dual flip-flop synchronizer is improved quite a lot, but if the receiving clock domain is 500 Mhz or larger the MTBF is not good enough anymore. Then more flip-flop stages are necessary which makes the silicon area and the latency for synchronization larger.

Figure 3.8 gives the trend of the MTBF (vertical axis) versus the clock frequency (horizontal axis) of the receiver for different synchronizers. The synchronizers differ in the number of flip-flops used in serial: \( mbf1 \) uses a single flip-flop, \( mbf2 \) uses two flip-flops, etc... The Figure shows indeed that a synchronizer with more flip-flops in serial has a better MTBF. Also the MTBF decreases enormously when the clock frequency of the receiver is increases.
Table 3.3: MTBF values for a dual flip-flop synchronizer (df2sqx05)

<table>
<thead>
<tr>
<th>$f_c$ [MHz]</th>
<th>$f_s$ [MHz]</th>
<th>$t_{\text{max}}$ [ns]</th>
<th>$t_{\text{max}2}$ [ns]</th>
<th>MTBF [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>10</td>
<td>29.5</td>
<td>27.5</td>
<td>2,52E+84</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>14.5</td>
<td>13.5</td>
<td>4,68E+40</td>
</tr>
<tr>
<td>200</td>
<td>10</td>
<td>7.03</td>
<td>6.53</td>
<td>4,36E+18</td>
</tr>
<tr>
<td>300</td>
<td>10</td>
<td>4.53</td>
<td>4.20</td>
<td>1,55E+11</td>
</tr>
<tr>
<td>400</td>
<td>10</td>
<td>3.28</td>
<td>3.03</td>
<td>2,54E+07</td>
</tr>
<tr>
<td>500</td>
<td>10</td>
<td>2.53</td>
<td>2.33</td>
<td>1,25E+05</td>
</tr>
<tr>
<td>600</td>
<td>10</td>
<td>2.03</td>
<td>1.86</td>
<td>3,37E+03</td>
</tr>
<tr>
<td>700</td>
<td>10</td>
<td>1.72</td>
<td>1.57</td>
<td>3,41E+02</td>
</tr>
</tbody>
</table>

Figure 3.8: the MTBF of different synchronizers for different clock frequencies
4. Top-level architecture

This chapter introduces the top-level architecture for interfacing the Network Interface (NI) to IP modules that use pausable clocking for synchronization. First a short introduction of the NI of \textit{Ethereal} is given, followed by the description of MOUSETRAP FIFOs. The different components in the top-level architecture will be described in detail in the next Chapters (including the implementation). The synchronizers that are also necessary in the top-level architecture are however described in this Chapter.

Thus the first Paragraphs give background information about the Network Interface. Paragraph 4.3 gives the first step towards a solution by introducing the top-level architecture.

4.1. Network Interface

The full description of the Network Interface can be found in [22][23][6]. Here a short introduction and the most important properties are given.

The NI can be split up into two main parts, these are the Kernel and the Shells. The Kernel executes the following tasks:

- Implements the channels: a channel is a connection between a master and a slave (point to point), a connection between a master and multiple slaves of which each executes a transaction (multicast) or a connection between a master and multiple slaves of which only one executes a transaction (narrowcast).

- Packetizes and schedules messages to the routers: the messages that are received at the Kernel are packetized. The scheduling depends on the properties of the channels. If a channel is configured as Best Effort (BE) there are no constraints on the throughput and a round robin arbitration is used between BE channels. A channel can also be configured as Guaranteed Throughput (GT), then sendable data is always directly scheduled (no arbitration).

- Implements end-to-end flow control: this ensures that no data is sent to a receiver in which there is no space available in the destination buffer of the particular channel.

- Implements the clock-domain crossing between the IP module and the network.

The Shells implement the connections (e.g. narrowcast, multicast), transaction ordering for connections, and other higher-level issues. The shells translate the protocol (e.g. DTL), which is used by the IP module, to messages that are send to the Kernel. A message is typical a sequentialized version of the commands, addresses, and write data for a request message or the read data for a response message.

4.1.1. Kernel

As shown in Figure 4.1 the Kernel of the NI consist of several components. The Kernel of Figure 4.1 has 3 IP ports. Two of the ports have a single source queue (at the top) and a single destination queue (at the bottom). One of the ports has two source and destination queues. The source queue holds messages that are send to the router and the destination queue holds messages that are received from the router. The source queues have a counter that holds the number of words in the queue. For the destination queues the number of words is also counted, this is done...
in the Local Credit. The end-to-end flow control is implemented with credits. The Remote Space contains the number of empty data words of the remote destination queue for every connection (all connections are stored in a Connection Table). When data is send to the destination queue the Remote Space is decreased. When the data is consumed by the IP module of the destination the Local Credit in the NI of the destination is increased. This Local Credit is send to the producer of the data and the value of the Local Credit that is received is added to the Remote Space of the producer.

The term sendable data is already used in the previous Paragraph, the term is defined as the minimum between the data items in the source queue and the value in the Remote Space. The Request Generator issues a signal specifying that the queue can be scheduled if the queue contains sendable data. The scheduling is further handled by the GT Scheduler and the BE Scheduler, dependent of the configuration of the specific channel.

One of the mechanisms that is used to optimise the NoC utilization is the usage of a threshold mechanism to use longer packets. These thresholds are stored in the Limit Table. A threshold can be overridden by the IP module with a so-called Flush signal. This is important, because it can happen that the threshold is not reached while there is no new data send by the IP module and there is data waiting to be send to the router. Then the data can remain in the queue indefinitely, because the IP module is waiting for an acknowledge from the destination.

4.1.2. Queues

The FIFOs in the Kernel are used as source and destination queues for the messages to and from the IP module. When the IP module has a clock domain that is not related to the clock domain of the Network Interface (or network) there is a clock domain crossing in the accompanying queue. In this Paragraph the architecture of the source queue with its counter is described. The approach for the destination queues is similar.
The source queue with its counter for counting the number of valid words in the FIFO is shown in Figure 4.2. For the source queues the write clock (wr_clk) is connected to the IP clock (IP_clk) and the read clock (rd_clk) is connected to the network clock (clk). The count register holds the number of valid data words in the FIFO and is used by the Request Generator in the Kernel. The rd_valid of the FIFO indicates if it is possible to read from the FIFO. Now this signal is not necessary anymore, because the counter holds the information of the filling of the FIFO.

![Figure 4.2: the source queue (rdt_ni_fifo) with counter (rdt_ni_cnt)](image)

The value in the counter is incremented with one if the wr_valid and the wr_accept signals are both true. And the value is decremented with the value of the send_words signal (from the Request Generator) when the Flit Controller in the Kernel requests data from the selected channel (cnt_read_pulse and selected_channel are both true).

### 4.2. MOUSETRAP FIFOs

MOUSETRAP is an abbreviation for Minimal-Overhead Ultrahigh-SpEed Transition-signalling Asynchronous Pipeline and is introduced by [27]. This naming already gives a good idea what kind of FIFO the MOUSETRAP is: a ripple-through FIFO. It uses an asynchronous control scheme with transition signalling and therefore it can be implemented into a small silicon area and gives a good performance for speed and power consumption. A “special” MOUSETRAP FIFO, which has two synchronous interfaces instead of asynchronous interfaces is used for the queues in the Kernel [33]. The plain MOUSETRAP FIFO is described in this Paragraph and the one with the synchronous interfaces is described in Paragraph 4.2.1.

A MOUSETRAP FIFO is build up with a number of cells. Each cell has a control and a memory part (see Figure 4.3) and the number of cells in serial gives the memory depth of the FIFO.

![Figure 4.3: a single MOUSETRAP cell consisting of a control and a memory part](image)
The control part of a cell is built up with a single bit latch and an Exclusive-NOR gate. The memory part is a number of latches storing the data, of course the width of the data is equal to the number of latches in the memory part of the cell. To illustrate the behaviour of a single MOUSETRAP cell its STG is given in Figure 4.4. When the \( Wreq \) signal makes a transition and the cell is empty (signal \( empty \) is true) then the write request action will be acknowledged with a transition of \( Wack \). Now a data word has been written to the cell and a read request will be passed to the next MOUSETRAP cell with a transition of the \( Rreq \) signal. The \( empty \) signal will be false until the read request is acknowledged with a transition of the \( Rack \) signal. The transition of the \( Rack \) signal means that the next MOUSETRAP cell has stored the data and the sending cell is "made" empty by this acknowledgement. This is the first sequence in the STG (started from the initial net marking). The second sequence behaves exactly the same, but now the transitions for the request and acknowledge signals are inverted. The reason for this is that transition signalling is used in the MOUSETRAP cell.

\[
\begin{align*}
&\text{Wack}^+ \\
&\text{empty}^+ \\
&\text{Wreq}^+ \\
&\text{empty}^- \\
&\text{Wack}^- \\
&\text{Wreq}^- \\
&\text{empty}^- \\
\end{align*}
\]

Figure 4.4: STG for a single MOUSETRAP cell

4.2.1. MOUSETRAP FIFO in a synchronous environment

To make use of the MOUSETRAP FIFO in a fully synchronous environment the input and the output stage of the asynchronous FIFO are adapted to the synchronous environment. Figure 4.5 shows the connections between a MOUSETRAP FIFO and the adapters to interface a write clock and a read clock. In [33] two different implementations are presented for the adaptors.

\[
\begin{align*}
&\text{Wdata} \\
&\text{wr_valid} \\
&\text{wr_accept} \\
&\text{wr_clk} \\
\end{align*}
\]

\[
\begin{align*}
&\text{Rdata} \\
&\text{rd_valid} \\
&\text{rd_accept} \\
&\text{rd_clk} \\
\end{align*}
\]

Figure 4.5: using adaptors for using a MOUSETRAP FIFO in a synchronous environment
Data is written to the FIFO when both signals \textit{wr\_accept} and \textit{wr\_valid} are true and there is a rising clock edge on \textit{wr\_clk}. Data is read from the FIFO when both signals \textit{rd\_accept} and \textit{rd\_valid} are true. Figure 4.6 shows the timing arcs of the FIFO when it is implemented with these adaptors. The arcs from the write clock (\textit{wr\_clk}) to the \textit{rd\_data} and \textit{rd\_valid} signals at the read side are only active if the FIFO is empty. The arc from the read clock (\textit{rd\_clk}) to the \textit{wr\_accept} signal is only active if the FIFO is full. This makes the FIFO without further hardware not suitable to be used in designs where the read clock and the write clock are independent of each other. If the FIFO is empty the \textit{rd\_data} and the \textit{rd\_valid} signals have to be synchronized to the clock of the receiver, because these signals are asynchronous to this clock signal. If the FIFO is full the \textit{wr\_accept} signal has to be synchronized to the clock of the sender, because this signal is asynchronous to this clock signal.

With addition of synchronizers for the signals that cross the two independent clock domains a Clock Domain Crossing (CDC) FIFO can be made. This kind of FIFO can be used safely in the Kernel when the IP module has a clock domain that is independent from the clock domain of the network.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{fig4.6}
\caption{timing arcs of the MOUSETRAP FIFO with synchronous interfaces, source Wielage [33]}
\end{figure}

4.3. Interfacing pausable clocked IP modules

A top-level architecture is chosen such that it is possible to synchronize an IP module with a pausable clock domain to the Network Interface, which is clocked by the network clock. Thus the clock domain of the IP module and the clock domain of the network are assumed to be completely independent of each other. Therefore different (independent) clock frequencies can be used between the two clock domains. The top-level architecture that is chosen is shown in Figure 4.7. In the top-level architecture the complete IP module consist of the following blocks:

- Pausable clock generator.
- The original IP module.
- Translation Shells from the Network Interface (e.g. for DTL).
- One or more output controllers.
- One or more input controllers.

The clock generated with the pausable clock generator is distributed to the original IP module and the Translation Shells. So the Translation Shells become part of the IP module. Every port between the Shells and the Kernel get an output controller and an input controller. The output controller is interfaced to the source queue and the input controller is interfaced to the destination.
queue of the Kernel. The source and destination queues are implemented with asynchronous FIFOs that use transition signalling. The MOUSETRAP FIFOs can be used here (just every asynchronous FIFO that uses transition signalling would do). For the source as well as for the destination queue there is a push channel. For the source queue the output controller is the initiator and therefore the request signal \((Req)\) goes from the output controller to the FIFO. For the destination queue the FIFO is the initiator and therefore the request signal \((Req)\) goes from the FIFO to the input controller.

![Figure 4.7: top-level architecture for interfacing pausible clocked IP modules to the NI](image)

Synchronizers are necessary for the interfacing of the asynchronous FIFO to the Kernel of the NI. Possible solutions for these synchronizers are introduced in [13]. Furthermore there is synchronization necessary for the counters that hold the number of valid data words in the queues. (The counter for the destination queue is implemented in the Credit Controller.) The \(ack\) signals from the FIFOs at the IP side can be used to update the counters in the Kernel. When synchronizing these signals there is a certain latency introduced on these signals and it is possible that the number of valid words in the queue is higher than the value in the counter. When this happens the threshold calculations in the Kernel will not be correct. However this is no problem for the functioning of the Kernel, because the counter will eventually be updated and a new threshold calculation is done. Thus the delay introduced by the counter comes back in the threshold calculations and therefore also in the scheduling of packets in the Kernel. Next to the synchronization of the signals for counting data words there is a \(Flush\) signal per source and destination queue. All \(Flush\) signals have to be synchronized at the Kernel as well. More details about these synchronizers are given in the next Paragraph.
4.4. Synchronizers

Eventually with the current top-level architecture synchronizers are necessary at the Kernel of the Network Interface, three different kind of synchronizers are identified (as mentioned in the previous Paragraph):

- Synchronizers for the asynchronous FIFOs.
- Synchronizers for the signals that can change the value of the counters, which count the number of valid words in the queues.
- Synchronizers for the Flush signals.

The use of synchronizers constraints the maximum clock frequency of the IP module. The maximum frequency of the IP module must be lower than the frequency of the network clock. If this is not the case data events from the IP module can not be sampled by the synchronizers in the Kernel.

For synchronization of the Flush signal dual flip-flop synchronizers can be used. With these synchronizers a delay is introduced on the Flush signals and furthermore there is a chance of metastability to occur (see Paragraph 3.5.1). The signals interfaced to the counters for the queues in the Kernel can be synchronized with the synchronizer as shown in Figure 4.8. This synchronizer is actually a dual flip-flop synchronizer with a 2-phase to 4-phase converter. The converter is necessary, because the Ack signal of the asynchronous FIFO is used to indicate that a data word is written to a source queue or read from a destination queue. This Ack signal uses transition signalling and is converted to a level sensitive signal, which can directly be used for updating the counters in the Kernel.

![Figure 4.8: dual flip-flop synchronizer with 2-phase to 4-phase converter](image)

It is important to mention that due to the delay for the Flush signals and the delay of the signals that are used for updating the counter values there can be a problem with the Flush functionality. When a Flush is issued by the IP module the number of data words written to the queue at that moment should be flushed. If due to the synchronization the Flush signal is received earlier than that the counter value is updated then not all valid words will be flushed. To prevent this problem the Flush signal must be received after the counter is updated. If this is not the case extra cycles can be inserted for the Flush signals by using more flip-flops in the synchronizer for the Flush signal. For example use three flip-flops in serial instead of two. This increases the delay, but the delay is already present for the updating of the counter values. There is also an advantage if more flip-flops are used in serial: the chance of metastability is going down (see Paragraph 3.5.1).

For the source queue the request signal to read data from the asynchronous FIFO is received at the Kernel. This request signal must be synchronized and when this signal is synchronized the data can be sampled safely as well. For the destination queue the acknowledge signal, which indicates that data is read by the asynchronous FIFO, is received at the Kernel. This acknowledge signal must be synchronized at the Kernel. As mention previously the synchronizers introduced
in [13] can be used for the asynchronous signals of the FIFOs. These synchronizers use so-called WAIT components. For completeness these WAIT components are described with STGs in the next Paragraph.

### 4.4.1. WAIT components

WAIT components can be used to synchronize handshake signals [13]. The basic WAIT component is the *WAIT4*. This component is a Mutual Exclusion element with only one output. The inverted clock is connected to the first input of the Mutual Exclusion element and the signal synchronized to the phase of the clock signal is connected to the second input. Only the second output (corresponding to the signal that must be synchronized) is available. Figure 4.9 shows the *WAIT4* component with the corresponding STG. The internal signal *dummy* is used in the STG to separate the *clk-* and the *clk+* transitions. A request (low-to-high transition of *Req*) will only propagate through the *WAIT4* component when the clock signal has a high phase. When the clock signal has a low phase and a request arrives, then this request is blocked until the clock signal rises.

![Figure 4.9: WAIT4 component with STG](image)

The *WAIT4* component is suitable to synchronize handshake signals that use 4-phase handshaking, but when 2-phase handshaking is used (e.g. MOUSETRAP) a 2-phase WAIT component must be used. Figure 4.10 shows a 2-phase WAIT component, also called *WAIT2*. It is build up with a *WAIT4* component, a latch and an Exclusive OR-gate. The behaviour of the WAIT2 component is described with a STG (see Figure 4.11).

![Figure 4.10: WAIT2 component](image)

In the right part of the STG the behaviour of the *WAIT4* component is found again. Instead of the *Req* and *Ack* signals it now has the signals *d* and *e*. The signal that is synchronized is named *Req*.
again, the synchronized result is the signal Ack. Now a rising as well as a falling transition of the Req signal will be blocked if the clock signal has a low phase. A transition is blocked until the clock has a high phase again.

The WAIT4 and the WAIT2 component both use synchronization by the phase of the clock signal, but it is also possible to synchronize by the edge of the clock signal with WAIT components. With edge synchronization a better performance can be achieved. A chance of metastability exists when WAIT components are used for synchronization, because metastability occurs in the Mutual Exclusion element when both inputs arrive at the same moment in time.
5. Pausible clock generator

The pausible clock generator is an essential component when clock pausing is used for synchronization purposes. The circuit that is used in the top-level architecture to generate a pausible clock signal for the IP module is described in this Chapter. First the basic circuit of this pausible clock generator is described. Then extensions to this circuit are discussed. These extensions are necessary for multiple request signals from I/O controllers that request to pause the clock signal. Two approaches to accommodate multiple request signals are discussed. Only one of the two is working correctly under all conditions of the input signals and is therefore the preferred circuit. Furthermore the timing of the pausible clock generator is described. This timing behaviour is important for the interfacing of the I/O controllers to the IP module and the pausible clock generator.

5.1. Basic circuit

The basic circuit of the pausible clock generator that is used for the pausible clocked IP module is shown in Figure 5.1. The basic circuit is actually a ring oscillator with an arbiter in the loop. The ring oscillator is build up with a delay line and an inverter. The delay line can be programmable, such that different frequencies can be generated. This is a necessity if dynamic frequency scaling is used [14][25][26]. A Mutual Exclusion element is used for the arbitration. The circuit also has an asynchronous reset, which is controlled by the \texttt{rst_an} signal. When extensions of the basic circuit are discussed later on in this Chapter this reset is not shown for keeping things a bit simpler. Every pausible clock generator should have a reset to start the clock generation properly. Three intermediate signals are also named in the basic circuit of Figure 5.1., these are:

- \texttt{clk\_n}: the output signal of the ring oscillator. This signal is inverted to produce the actual clock signal (\texttt{clk}), thus \texttt{clk\_n} is the inverted clock signal.
- \texttt{clk\_d}: this is the delayed \texttt{clk\_n} signal.
- \texttt{clk\_m}: the signal \texttt{clk\_d} after the propagation through the Mutual Exclusion element.

With the inputs, outputs and intermediate signals of the basic circuit for the pausible clock generator a STG can be made that describes the behaviour of the circuit. This STG is shown in Figure 5.2. The reset behaviour of the basic circuit is also modelled in the STG. The result of this is that the STG does not comply to the \textit{liveness} property and the \textit{single-cycle-transitions} property of STGs. The former is not met, because transition \texttt{rst\_an}- can not be fired from every reachable
net-marking of the STG. The latter is not met, because the signal \( \text{rst}_\text{an} \) has only a falling transition and there is no rising transition in the STG.

The circuit is reset with a falling transition of \( \text{rst}_\text{an} \). When \( \text{rst}_\text{an} \) is logic zero the output \( \text{clk} \) is logic zero as well, until \( \text{rst}_\text{an} \) becomes logic one then the clock generation is started. The clock frequency is determined by the delay between the two transitions \( \text{clk} \; \text{n+} \) and \( \text{clk} \; \text{n-} \) (low phase of the clock signal) and the two transitions \( \text{clk} \; \text{n-} \) and \( \text{clk} \; \text{n+} \) (high phase of the clock signal).

![Figure 5.2: the STG of the basic circuit for the pausable clock generator](image)

The clock generation can be paused immediately when the \( \text{clk}_\text{d} \) signal has a low phase. If during this phase the \( \text{Req} \) signal is logic one then the clock generation is paused until the \( \text{Gnt-} \) transition has fired. The transition \( \text{Req+} \) can fire as soon as there is a token at its input arc. However the clock is paused only when the token in the place in the middle of the STG has returned and this happens after transition \( \text{clk}_\text{m-} \) has fired (low phase \( \text{clk}_\text{d} \)).

5.2. Timing of the basic circuit

The timing of the basic circuit of the pausable clock generator is dependent from the arrival time of the \( \text{Req} \) signal. The following situations are possible for the circuit:

1. The circuit is reset and the \( \text{Req} \) signal is false.
2. The \( \text{Req} \) signal rises at the low phase of the \( \text{clk}_\text{d} \) signal.
3. The \( \text{Req} \) signal rises at the high phase of the \( \text{clk}_\text{d} \) signal.
4. There is metastability in the Mutex, because the \( \text{Req} \) signal rises at the rising edge of the \( \text{clk}_\text{d} \) signal and the \( \text{Req} \) signal "wins".
5. There is metastability in the Mutex, because the \( \text{Req} \) signal rises at the rising edge of the \( \text{clk}_\text{d} \) signal and the \( \text{clk}_\text{d} \) signal "wins".
5.2.1. Situation 1

The reset of the circuit itself needs a constraint for the time the reset signal has to be logic zero. When the circuit is reset the \( \text{Req} \) signal must remain logic zero to make sure the clock generation starts correctly after the reset is done. Figure 5.3 shows the timing diagram for the basic circuit during a reset and after the reset when \( \text{Req} \) remains logic zero (situation 1; the other four situations are covered in the next Paragraphs). The shaded area in the timing diagram indicates that the signal is undefined. The minimal time that the reset must be logic zero to reset the pausable clock generator is equal to:

\[
T_{\text{reset\_min}} = \tau_{\text{AND}} + \tau_{\text{delay\_line}} + \tau_{\text{mutex}} + \tau_{\text{inv\_ring}} + \tau_{\text{inv}} + \tau_{\text{clockskew}}
\]  

(15)

With:

- \( \tau_{\text{AND}} \): the propagation delay of the AND gate.
- \( \tau_{\text{delay\_line}} \): the propagation delay of the (programmable) delay line.
- \( \tau_{\text{mutex}} \): the propagation delay of the Mutual Exclusion element.
- \( \tau_{\text{inv\_ring}} \): the propagation delay of the inverter in the ring oscillator.
- \( \tau_{\text{inv}} \): the propagation delay of the inverter.
- \( \tau_{\text{clockskew}} \): the maximum value for the clock skew in the clock tree of the design.

![Figure 5.3: timing diagram for the reset of the pausable clock generator and the clock generation after a reset when signal Req is logic zero](image)

5.2.2. Situation 2

The second situation occurs when the input signal \( \text{Req} \) rises during the low phase of the intermediate signal \( \text{clk\_d} \). Figure 5.4 shows the timing diagram for situation 2. In this case the request to pause the clock is immediately granted by the Mutual Exclusion element (see also the STG of Figure 5.2). Thus the clock generation is directly paused and there is no rising edge of the clock signal after the request to pause the clock. The rising edge of the clock signal is generated again when the \( \text{Gnt} \) signal falls.

Related to the generated clock signal the request to pause the clock is allowed to arrive (\( t_{\text{req\_arrival}} \)) at the following time intervals (for situation 2):

© Koninklijke Philips Electronics N.V. 2004
With the assumptions that at moment zero the clock signal rises, the period of the clock signal is $T$ and that the duty cycle of the clock signal is 50%.

\[
\frac{T}{2} - \tau_{\text{mutex}} - \tau_{\text{inv, ring}} - \tau_{\text{inv}} < t_{\text{req, arrival}} < \frac{T}{2} - \tau_{\text{mutex}} - \tau_{\text{inv, ring}} - \tau_{\text{inv}} \tag{16}
\]

5.2.3. Situation 3

Figure 5.5 shows the timing diagram for situation 3. The request to pause the clock generation arrives during the high phase of $clk_d$. Now the request to pause the clock is not granted immediately as is the case for situation 2, because the Mutual Exclusion element is blocking the $Req$ signal until the $clk_d$ signal becomes false again. It is important to note that for situation 3 there is the possibility that there is a rising edge of the clock signal within the request/grant sequence. This happens when the $Req$ signal arrives in the shaded area of the high phase of signal $clk_d$ (see Figure 5.5). This shaded area starts at the rising edge of $clk_d$ and has the following width:

\[
\tau = \tau_{\text{inv}} + \tau_{\text{mutex}} + \tau_{\text{inv, ring}} \tag{17}
\]
Related to the generated clock signal the request to pause the clock is allowed to arrive at the following time intervals (for situation 3):

\[ 0 < t_{\text{req\_arrival}} < T/2 - \tau_{\text{mutex}} - \tau_{\text{inv\_ring}} - \tau_{\text{inv}} \]  

(18)

With the assumptions that at moment zero the clock signal rises, the period of the clock signal is \( T \) and that the duty cycle of the clock signal is 50%. Furthermore here is assumed that there is no rising clock edge allowed during the request/grant sequence to pause the clock signal. Together with the allowed arrival time intervals for the request in situation 2 this gives the complete interval in which a request is allowed to arrive (with the same aforementioned assumptions):

\[ 0 < t_{\text{req\_arrival}} < T - \tau_{\text{mutex}} - \tau_{\text{inv\_ring}} - \tau_{\text{inv}} \]  

(19)

\[ 5.2.4. \text{ Situation 4 and 5} \]

Situation 4 and 5 are the cases in which metastability occurs due to a rising edge of the \( \text{Req} \) signal when the \( \text{clk\_d} \) signal rises. Situation 4 defines the case in which signal \( \text{Req} \) wins the arbitration and situation 5 defines the case in which signal \( \text{clk\_d} \) wins the arbitration. Figure 5.6 shows the timing diagram for situation 4.

It shows that if the \( \text{Req} \) signal wins the arbitration, the clock signal is paused until the request/grant sequence is complete. However there is additional delay added for handling the request/grant sequence, because the metastability in the Mutual Exclusion element must be resolved. The shaded area is the time interval in which metastability occurs and after this interval the metastability has ended. During the metastability the Mutual Exclusion element makes both outputs false.

[Figure 5.6: timing diagram for situation 4]

The timing diagram for situation 5 is shown in Figure 5.7. When the metastability is resolved and (in this case) signal \( \text{clk\_d} \) wins the arbitration, the \( \text{Req} \) signal is blocked until \( \text{clk\_d} \) falls again. Next to the additional delay for the request/grant sequence there is now also a rising edge of the clock signal during the request/grant sequence. It is better to prevent metastability to get a better performance for pausing the clock signal. If the clock is not allowed to rise during the request/grant sequence then metastability must even be prevented in the Mutual Exclusion element (e.g. by setting the proper timing constraints for the \( \text{Req} \) signal).
5.3. Multiple request signals with one Mutual Exclusion element

The basic circuit of the pausible clock generator can only be used with one I/O controller, because there is only one request/grant (Req and Gnt signals) pair available. Figure 5.8 shows the circuit of a pausible clock generator that has two request/grant pairs. The circuit can be extended with more than two request/grant pairs by increasing the number of Muller C-elements and the number of input ports for the OR gate. The Muller C-elements are necessary to make sure that the grant signals fall after the clock generation has started again. If an AND gate is used the grant signals immediately fall when the corresponding request signal falls. If this is the case and if the request has not propagated through the Mutual Exclusion element the clock generation is not started yet, while the grant signal is already false.

A nice feature of this circuit is that there is only one Mutual Exclusion element used in the circuit. This makes the necessary silicon area smaller to implement the circuit compared to the circuit discussed in the next Paragraph. That circuit uses more Mutual Exclusion elements for enabling multiple request signals.

The part that handles the request and grant signals is shown in Figure 5.9. The STG for this part of the circuit is drawn, which is shown in Figure 5.10. The Mutual Exclusion element is taken out of the circuit as well, so that there is a separate output signal me_req and a separate input signal me_ack. The initial net marking of the left STG in Figure 5.10. The STG can actually be split into
two separate STGs working in parallel, one for the \( \text{Req1/Gnt1} \) pair and one for the \( \text{Req2/Gnt2} \) pair. Thus if these two are combined in a single STG to describe the behaviour of the circuit of Figure 5.9 there are two initial tokens in the STG. These two tokens can then fire transitions independent of each other. A problem with the circuit can occurs when the net marking of the right STG in Figure 5.10 is considered. The net marking can cause a glitch on the \text{me\_req} or \text{me\_ack} signal. For example if the transition \text{me\_req+} fires the transition \text{me\_req-} can fire immediately after this because of the second token in the STG. The possibility of these glitches to occur in the circuit makes the circuit not suitable to be used as a pausable clock generator, because it will make the clock pausing not robust.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5_9.png}
\caption{the request/grant part of the pausable clock generator with two request/grant pairs and only one Mutual Exclusion element}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5_10.png}
\caption{the STG of the request/grant part of the pausable clock generator; left STG: initial net marking; right STG: net marking, which can cause glitches}
\end{figure}
5.4. Multiple request signals with multiple Mutual Exclusion elements

An alternative for accommodating more than one request/grant pair for pausing the clock signal is shown in Figure 5.11. The circuit uses a Mutual Exclusion element for every request/grant pair. In the circuit there are two request/grant pairs, thus there are two Mutual Exclusion elements used. The grant signals, \textit{me\_gnt1} and \textit{me\_gnt2}, are used for the clock generation and are combined with the AND gate. If one of the two is not granted, because a request to pause the clock generation is received, then the output of the AND gate becomes false and the clock generation is paused. This pausable clock generator can always be extended with more request/grant pairs. This is achieved by adding a Mutual Exclusion element for every request/grant pair and adding an input port for the AND gate. The Mutual Exclusion elements are connected in parallel and therefore there is no additional delay when there are more Mutual Exclusion elements inserted. However the chance that the clock generation will be paused is larger with more request/grant pairs (this is a system level issue). The behaviour of this pausable clock generator is specified completely with a STG, which is shown in Figure 5.12. In contrary to the circuit introduced in Paragraph 5.3, no glitches can occur in this circuit.

![Figure 5.11: the pausable clock generator with two request/grant pairs and two Mutual Exclusion elements](image)

© Koninklijke Philips Electronics N.V. 2004
Figure 5.12: the STG for the pausible clock generator with two request/grant pairs and two Mutual Exclusion elements
6. Output controller

One output controller is developed. This chapter describes the interfacing, the behaviour and the timing specification of this output controller. The output controller is specified with a Signal Transition Graph (STG). The tool Petrify is used to synthesize the STG in equations for generalized C-elements. The implementation of this output controller in standard cell design is described in Chapter 8.

6.1. Interface

The interfacing of the output controller to the IP module, the pausible clock generator and the asynchronous FIFO is shown in Figure 6.1. The complete interface of the output controller consists of the following inputs and outputs:

- **Inputs:** \( \text{clk}, \text{wr\_valid}, \text{Gntl}, \text{ack} \).
- **Outputs:** \( \text{Req1}, \text{req} \).

The \( \text{clk} \) signal is also interfaced to the output controller, but the clock signal is only used to resample the \( \text{wr\_valid} \) signal as is explained in the timing specification of the output controller (see Paragraph 6.3). The \( \text{wr\_valid} \) signal indicates when data is send to the asynchronous FIFO. The output controller controls the handshaking with the asynchronous FIFO and can request to pause the generation of the clock signal with the \( \text{Req1} \) signal. The request to pause the clock is acknowledged with the \( \text{Gntl} \) signal. The handshaking with the pausible clock generator is level sensitive (also called 4-phase handshaking) and for the communication with the asynchronous FIFO transition signalling (also called 2-phase handshaking) is used. The data signals from the IP module are directly connected to the asynchronous FIFO.

![Figure 6.1: interfacing of the output controller to the IP module, the pausible clock generator and the asynchronous FIFO](image-url)
6.2. Behaviour

The output controller is an Asynchronous Finite State Machine (AFSM) with some additional logic. The STG for the AFSM is shown in Figure 6.2 and describes the behaviour of the output controller. There is a small difference with the interface and the STG for the AFSM: the \textit{wr\_valid} signal at the interface is not used in the STG, instead the signal \textit{valid} is used. There is only a difference in timing, which becomes clear in Paragraph 6.3 where the timing specification of the output controller is defined. The \textit{valid} signal has the same meaning as the \textit{wr\_valid} signal, which is the indication that new data must be send to the asynchronous FIFO. The STG also contains an internal signal called \textit{dummy}. This signal is inserted to prevent that the transition \textit{valid-} is followed by the transition \textit{valid+}. Furthermore the STG has two almost identical parts which are connected via the \textit{valid+} transition. The only difference between the two parts is the rising transitions for the signals \textit{req} and \textit{ack} in one part and the falling transition for these signals in the other part. The reason for this is that these signals use transition signalling with the asynchronous FIFO.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{stg.png}
\caption{STG for the output controller}
\end{figure}
The behaviour of the output controller is then as follows (see also the timing diagram in Figure 6.3):

- After the firing of transition valid+ the Req1 signal will rise (transition Req1+) to request to pause the clk signal.

- When this is granted with the Gnt1 signal from the pausable clock generator, the transition req+ fires to request writing to the asynchronous FIFO. The transition ack+ will fire when the FIFO consumed the data.

- The clk signal can now be activated again by making the Req1 signal false (transition Req1-) and when the Gnt1- transition fires the output controller waits for a new rising edge of the valid signal. Before the valid+ transition can fire again the valid- transition must fire first (including the transitions for the dummy signal), as is shown in the parallel branch in Figure 6.2.

- The sequence of transitions is repeated, but now with a high-to-low transition for the req and ack signals, because the asynchronous FIFO uses transition signalling. When this sequence is complete the STG begins at the starting point again.

![Figure 6.3: timing diagram for the output controller](image)

The complete sequence of writing a data word to the asynchronous FIFO should be done within a single clock cycle. Then the clock signal is never paused when there is space in the FIFO. If the FIFO is full, the clk signal will be paused and the IP module must wait until there is space available in the FIFO. The first data word written in Figure 6.3 shows that writing a data word to the asynchronous FIFO can be done within one clock cycle, because here the clock signal is not paused. However the second word that is written pauses the clock signal, because the FIFO is full. When the transition of the ack signal is received the data is consumed by the FIFO and the clock is started again.

The input file for Petrify and the resulting equations for the output controller can be found in Appendix B.

### 6.3. Timing specification

The wr_valid signal from the IP module is interfaced to the output controller and the wr_data signals are interfaced to the asynchronous FIFO (see Figure 6.1 and Figure 6.4). Figure 6.4 shows the output controller with the AFSM and the surrounded logic. The logic is necessary to translate the wr_valid signal to the valid signal that is connected to the AFSM. It must be possible to write
a data word to the asynchronous FIFO every clock cycle, so that there is a high throughput for the data. Therefore timing constraints for the interfacing of the output controller are necessary:

1. The valid signal must rise and fall in a single clock cycle (as the defined in the STG).
2. When the valid signal rises the wr_data must be stable.
3. When a request to pause the clock signal is send to the pausable clock generator, no rising edge of the clock signal is allowed until the request is withdrawn.

![Diagram of the IP module and output controller](image)

**Figure 6.4: The paths for the valid and the wr_data signals from the IP module to the output controller**

The wr_valid signal does not have a rise and fall transition in a single clock cycle. To generate this behaviour for the valid signal an extra flip-flop is used (FF2), which has an asynchronous reset port. This port is controlled by the normal reset signal (rst_an), but also with the Req1 signal. The latter becomes true after the valid signal becomes true (see STG) and is received by the AFMS. Now the valid signal will always fall after its rising transition and when the Req1 signal is false again a new valid+ transition can be produced with FF2. This as a whole solves constraint 1.

FF2 samples the input at a different phase than the phase of the clk signal, which is used in the IP module. This phase difference is used to insert extra delay for the valid signal to make a rising transition, because when the valid+ transition occurs the data that is send to the asynchronous FIFO must be stable. The minimal value for the delay $\tau_i$ can be calculated with (constraint 2):

$$
\tau_i = \max \left\{ \left( t_{p,ff} + t_{p,logic} + t_{p.wire} + t_{su,ff} \right), \left( t_{p,ff} + t_{p,wire} + t_{p,logic} \right) \right\}
$$

With:
- $t_{p,ff}$: the propagation delay of FF1.
- $t_{p,logic}$: the propagation delay of logic1.
- $t_{p,wire}$: the propagation delay of the wires between FF1 and FF2.
- $t_{su,ff}$: the setup time of FF2.
- $t_{p,ff}$: the propagation delay of FFi.
- $t_{p\text{-wire}2}$: the propagation delay of the wires between \textit{FF}$_3$ and the asynchronous FIFO.
- $t_{p\text{-logic}2}$: the propagation delay of \textit{logic}$_2$.

For constraint 3 the used pausable clock generator must guarantee that there is no rising edge of the clock signal when a request to pause the clock generation is send. When the pausable clock generator of Paragraph 5.4 is used, then putting the proper timing constraints on the \textit{Reql} (and \textit{Req2}) signals answers constraint 3.

The time at which the \textit{Reql} signal rises after the last rising clock edge can be determined with:

$$\tau_2 = \tau_1 + t_{p\text{-ff}2} + t_{p\text{-controller}}$$  \hspace{1cm} (21)

With:
- $t_{p\text{-controller}}$: the propagation delay for the output controller to make a low-to-high transition for signal \textit{Reql}.
- $t_{p\text{-ff}2}$: the propagation delay of \textit{FF}$_2$.

To apply these timing constraints in a design flow there is the problem that not all delays are known during design time (e.g. the delay of wires). A solution for this is that a fixed value for delay $\tau_1$ is taken. A good choice is to take $\tau_1$ equal to half the clock period. Then \textit{FF}$_2$ can simply be a negative-edge-triggered flip-flop and this gives the timing constraint (with $T$ the clock period and assuming a duty cycle of 50%):

$$\max \{(t_{p\text{-ff}1} + t_{p\text{-logic}1} + t_{p\text{-wire}1} + t_{p\text{-controller}}), (t_{p\text{-ff}2} + t_{p\text{-wire}2} + t_{p\text{-logic}2})\} \leq T/2$$ \hspace{1cm} (22)

This constraint is set during design time and can be checked after layout with static timing analysis. The timing constraint for all request signals to pause the clock is then:

$$t_{p\text{-controller}} < T/2 - t_{p\text{-ff}2} - \tau_{mutex} - \tau_{inv\text{-ring}} - \tau_{inv}$$ \hspace{1cm} (23)

With the following delays in the pausable clock generator (remember the timing of the pausable clock generator):
- $\tau_{mutex}$: the propagation delay of the Mutual Exclusion element.
- $\tau_{inv\text{-ring}}$: the propagation delay of the inverter in the ring oscillator.
- $\tau_{inv}$: the propagation delay of the inverter.
7. **Input controller**

Two input controllers are developed. This Chapter describes the interfacing, the behaviour and the timing specification of these input controllers. The same procedure as for the output controller is followed. Thus both input controllers are defined with STGs and synthesized to equations. The input controller that is described first uses an extra latch stage to keep the input data stable such that the data can be sampled by the IP module. The other input controller uses the asynchronous FIFO to keep the data stable. Only the input controller, which does not use an auxiliary latch stage is implemented and this implementation is described in Chapter 8.

7.1. **Input Controller with auxiliary latches**

The input controller is first designed with the use of an extra latch stage that keeps the data stable when this data is sampled by the IP module. The input controller can then directly control these latches. The advantage of using auxiliary latches is that the control of these latches is completely decoupled from the 2-phase handshaking of the FIFO via the input controller. However the use of auxiliary latches makes the AFSM in the input controller more complex and also the timing for the interface is more complex.

The latches can be implemented with standard cell latches with a single enable signal (a reset is not necessary). However if standard cells are used this will consume quite some silicon area. It would be better to design the latches together with the asynchronous FIFO, so that both are designed full custom.

7.1.1. **Interface**

The interfacing of the input controller to the IP module, the pausible clock generator, the auxiliary latch stage and the asynchronous FIFO is shown in Figure 7.1. The complete interface of the input controller consists of the following inputs and outputs:

- **Inputs:** \( clk, rd\_accept, Gnt2, req. \)
- **Outputs:** \( Req2, ack, en. \)

For the \( rd\_valid \) signal a similar approach is taken as for the \( wr\_valid \) signal of the output controller. The \( clk \) signal is used to resample the \( rd\_valid \) signal as is explained further in the timing specification of the input controller. The \( rd\_valid \) signal indicates that the IP module wants to read a data word from the FIFO. Furthermore the 4-phase handshaking with the pausible clock generator is the same as for the output controller. The \( Req2 \) signal is used to send a request to pause the clock and this is granted by the pausible clock generator with the \( Gnt2 \) signal. The 2-phase handshaking with the asynchronous FIFO is different compared to the output controller. Now the \( req \) signal is send by the asynchronous FIFO, which is therefore the initiator and the sender of the data. The input controller will acknowledge to the asynchronous FIFO with the \( ack \) signal that the data has been consumed by the IP module. As mentioned before the latches are used to sample the data from the asynchronous FIFO (\( rd\_data \) signals). This is done by setting the latches transparent by making the enable signal (\( en \)) logical one. When the data is sampled the latches are made opaque by making the enable signal (\( en \)) logical zero.
Figure 7.1: interfacing of the input controller with auxiliary latches to the IP module, the pausable clock generator and the asynchronous FIFO

7.1.2. Behaviour

The STG of the input controller with auxiliary latches is shown in Figure 7.2. The following three starting places are present in the STG:

- The place before transition req+.
- The place before transition Req2+.
- The place before transition accept+.

This STG also has two parts which are the same accept for the transitions of the handshaking signals for the asynchronous FIFO (due to the 2-phase handshaking). The accept signal in the STG has the same function as the rd_accept signal as described before. The behaviour of the input controller with auxiliary latches is then as follows (see also the timing diagram of Figure 7.3):

- After the firing of transition accept+ a request to the pausable clock generator is send to pause the clock (transition Req2+).
- In the meantime a request from the asynchronous FIFO can be received (or is already received earlier) and when the low-to-high transition of the Gnt2 signal is received by the input controller the transition en+ fires to set the latches transparent.
- The enable signal is delayed (as explained in the timing specification in the next Paragraph) and after this delay (firing of transition en_d+) the pausing of the clock is stopped (firing of transitions Req2- and Gnt2-).
- When the latch enable signal (en) and the delayed version of this signal (en_d) are made false, then the acknowledge is send to the asynchronous FIFO with the ack signal. This acknowledge indicates that the data is consumed by the latches.
• Parallel to the sequence of steps above the \textit{accept} signal must have a high-to-low transition. When the transition \textit{accept}+ fires again the complete sequence is repeated, but now with the inverted transitions of the request (\textit{req}) and acknowledge (\textit{ack}) signals of the asynchronous FIFO.

• After the second sequence the STG starts over again from the starting point(s).

\textbf{Figure 7.2: STG for the input controller with auxiliary latches}
The complete sequence of reading a data word from the asynchronous FIFO should be done within a single clock cycle. The clock signal is never paused when there is valid data in the FIFO. However if the FIFO is empty then the clk signal is paused and the IP module must wait until there is valid data available in the FIFO. Figure 7.3 shows that the first data word that is read from the asynchronous FIFO can be read within a single clock cycle, so the clock signal is not paused. When the second data word is read the clock signal is paused, because the transition accept+ fires when the FIFO is empty. There is data available when the req signal makes a transition again and then the read sequence is continued.

The input file for Petrify and the resulting equations for this input controller can be found in Appendix C.

7.1.3. Timing specification

Next the timing specification is given for the input controller with an auxiliary latch stage. The rd_accept signal is interfaced to the input controller and the rd_data signals are interfaced from the asynchronous FIFO to the extra latch stage (see Figure 7.4). The latch stage is controlled by the input controller with the en signal and the data from the latch stage is interfaced to the IP module.

The en signal is delayed with $t_3$ when the en signal has a low-to-high transition. The hereby generated signal is called en_d and is send back to the AFSM of the input controller. The en_d signal is generated by using an AND gate with as input the en signal and the delayed en signal. The following timing constraints are necessary for this input controller:

1. The accept signal must rise and fall in a single clock cycle (as defined in the STG).
2. When a request is received from the asynchronous FIFO (transition of the req signal) the data must be kept stable long enough to be sampled correctly by the receiving flip-flops of the IP module (setup and hold constraints).
3. When a request to pause the clock signal is send to the pausible clock generator, no rising edge of the clock signal is allowed until the request is withdrawn.
Constraint 1 is similar to constraint 1 of the output controller and therefore an extra flip-flop ($FF_2$) is used to generate the $accept$ signal. This flip-flop can be reset with the normal reset signal ($rst\_an$) and the $Req2$ signal from the AFSM. The latter makes sure that the falling transition of the $accept$ signal automatically occurs after its rising transition is received by the AFSM. The minimal value for the delay $\tau_1$ can be calculated with:

$$\tau_1 = t_p_{ff1} + t_p_{logic1} + t_p_{wire1} + t_{su_{ff2}}$$

With:

- $t_p_{ff1}$: the propagation delay of $FF_1$.
- $t_p_{logic1}$: the propagation delay of the $logic_1$.
- $t_p_{wire1}$: the propagation delay of the wires between $FF_1$ and $FF_2$.
- $t_{su_{ff2}}$: the setup time of $FF_2$.

The signal $en\_d$ determines when the clock can be enabled again. The delay $\tau_3$ is then used to make sure that the data from the FIFO can arrive at $FF_3$ before the rising clock edge. The minimal value for delay $\tau_3$ can be calculated with (constraint 2):

$$\tau_3 = t_p_{latch} + t_p_{logic2} + t_p_{wire2} + t_{su_{ff3}}$$

With:

- $t_p_{logic2}$: the propagation delay of $logic_2$.
- $t_p_{latch}$: the propagation delay of the latch.
- $t_p_{wire2}$: the propagation delay of the wires between $FF_3$ and the asynchronous FIFO.
- $t_{su_{ff3}}$: the setup time of $FF_3$. 

Figure 7.4: the paths for the $accept$ and the $rd\_data$ signals to and from the input controller when auxiliary latches are used
The hold constraint for the *rd_data* is automatically met if (constraint 2):

\[ \tau_1 > t_{h,FF_i} \] (26)

With:

- \( t_{h,FF_i} \): the hold time of FF_i.

Constraint 3 is similar as for the output controller. Thus, the time at which the Req2 signal rises after the last rising clock edge can be determined with:

\[ \tau_2 = \tau_1 + t_{p,FF_2} + t_{p,controller} \] (27)

With:

- \( t_{p,controller} \): the propagation delay for the input controller to make a low-to-high transition for signal Req2.
- \( t_{p,FF_2} \): the propagation delay of FF_2.

For practical reasons it is wise to take a fixed value for delay \( \tau_1 \) as was mentioned for the output controller. The easiest solution is to take a half clock period for delay \( \tau_1 \). Then FF_2 can be a negative-edge-triggered flip-flop and this gives the timing constraint (with \( T \) the clock period and a duty cycle of 50%):

\[ t_{p,FF_2} + t_{p,logic} + t_{p,reset} + t_{sw,FF_2} \leq T/2 \] (28)

This constraint is set during design time and can be checked after layout with static timing analysis. The timing constraint for the Req2 signal is then (similar as for the output controller):

\[ t_{p,controller} < T/2 - t_{p,FF_2} - \tau_{max} - \tau_{inv,ring} - \tau_{inv} \] (29)

Delay \( \tau_3 \) can be inserted with an estimated worse case value first and can then be checked with static timing analysis. However this is not a very nice approach, because if the estimated value was not good enough a new delay must be inserted into the design and the layout phase must be restarted. Another option is to use a programmable delay line, which can be programmed during a system reset. This has the advantage that there is no danger of longer design time due to a bad choice of the value of the delay. The disadvantage is that more silicon area is necessary with this approach.

### 7.2. Input Controller without auxiliary latches

This input controller does not use an extra latch stage to keep the data stable that has to be sampled by the IP module. Instead the last (latch) stage of the asynchronous FIFO is used for this purpose. The result is that the effective depth of the asynchronous FIFO is reduced with one. This is explained further when the behaviour of this input controller is discussed. Eventually this means that still extra latches are needed, but these can now be implemented in the asynchronous FIFO directly. This makes the design easier and also more efficient in silicon area and speed. Furthermore the AFSM for this input controller becomes simpler than the one for the input controller with auxiliary latches.
7.2.1. Interface

The interfacing of the input controller to the IP module, the pausable clock generator and the asynchronous FIFO is shown in Figure 7.5. The complete interface of this input controller consists of the following inputs and outputs:

- **Inputs:** \( clk, \text{rd\_accept}, Gnt2, \text{req}. \)
- **Outputs:** \( \text{Req2}, \text{ack}. \)

The interface of this input controller is the same as for the input controller with auxiliary latches with the difference that there are no latches in the data path \( \text{rd\_data} \) and therefore the enable signal \( \text{en} \) is missing as well.

![Figure 7.5: interfacing of the input controller without auxiliary latches to the IP module, the pausable clock generator and the asynchronous FIFO](image)

7.2.2. Behaviour

For the input controller without auxiliary latches the STG is shown in Figure 7.6. There are three starting places:

- The place before transition \( \text{ack}- \).
- The place before transition \( \text{Req2}+ \).
- The place before transition \( \text{accept}+ \).

The behaviour of the input controller without auxiliary latches is as follows (see also the timing diagram in Figure 7.7):

- At start up the controller waits for the \( \text{accept}+ \) transition to fire. If this transition arrives a request to pause the clock is given to the pausable clock generator (transition \( \text{Req2}+ \)) and an acknowledge is given to the asynchronous FIFO (transition \( \text{ack}- \)) to make it possible for the asynchronous FIFO to write to the last stage of this FIFO.
Now the controller waits for a request signal from the asynchronous FIFO. If this request is received then the data is available in the last stage of the FIFO. Therefore pausing of the clock signal can be stopped by firing the transition \( \text{Req2-} \).

In the meantime the \( \text{accept} \) signal must have a high-to-low transition and when a new rising clock edge occurs transition \( \text{accept+} \) can fire again.

After the firing of the second \( \text{accept+} \) transition the same sequence of events is repeated, but now with the inverted transitions for the request \( \text{(req)} \) and acknowledge \( \text{(ack)} \) signals of the asynchronous FIFO. After this second sequence the STG begins at the starting point(s) again.

\[ \text{accept+} \]
\[ \text{ack-} \]
\[ \text{req+} \]
\[ \text{ack+} \]
\[ \text{req-} \]
\[ \text{Req2-} \]
\[ \text{Gnt2-} \]
\[ \text{Req2+} \]
\[ \text{Gnt2+} \]
\[ \text{dummy-} \]
\[ \text{accept-} \]
\[ \text{dummy+} \]
\[ \text{accept+} \]
\[ \text{Req2+} \]
\[ \text{Gnt2+} \]
\[ \text{dummy-} \]
\[ \text{accept-} \]
\[ \text{dummy+} \]

\[ \text{Gnt2-} \]

\[ \text{accept+} \]
\[ \text{ack-} \]
\[ \text{req+} \]
\[ \text{ack+} \]
\[ \text{req-} \]
\[ \text{Req2-} \]
\[ \text{Gnt2-} \]

\[ \text{accept-} \]
\[ \text{dummy+} \]

\[ \text{Gnt2+} \]

\[ \text{Req2+} \]

\[ \text{Gnt2+} \]

\[ \text{dummy-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2-} \]

\[ \text{accept+} \]

\[ \text{ack-} \]

\[ \text{req+} \]

\[ \text{ack+} \]

\[ \text{req-} \]

\[ \text{Req2-} \]

\[ \text{Gnt2-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2+} \]

\[ \text{Req2+} \]

\[ \text{Gnt2+} \]

\[ \text{dummy-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2-} \]

\[ \text{accept+} \]

\[ \text{ack-} \]

\[ \text{req+} \]

\[ \text{ack+} \]

\[ \text{req-} \]

\[ \text{Req2-} \]

\[ \text{Gnt2-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2+} \]

\[ \text{ Req2+} \]

\[ \text{ Gnt2+} \]

\[ \text{ dummy-} \]

\[ \text{ accept-} \]

\[ \text{dummy+} \]

\[ \text{ Gnt2-} \]

\[ \text{accept+} \]

\[ \text{ack-} \]

\[ \text{req+} \]

\[ \text{ack+} \]

\[ \text{req-} \]

\[ \text{Req2-} \]

\[ \text{Gnt2-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2+} \]

\[ \text{Req2+} \]

\[ \text{Gnt2+} \]

\[ \text{dummy-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2-} \]

\[ \text{accept+} \]

\[ \text{ack-} \]

\[ \text{req+} \]

\[ \text{ack+} \]

\[ \text{req-} \]

\[ \text{Req2-} \]

\[ \text{Gnt2-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2+} \]

\[ \text{Req2+} \]

\[ \text{Gnt2+} \]

\[ \text{dummy-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2-} \]

\[ \text{accept+} \]

\[ \text{ack-} \]

\[ \text{req+} \]

\[ \text{ack+} \]

\[ \text{req-} \]

\[ \text{Req2-} \]

\[ \text{Gnt2-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2+} \]

\[ \text{Req2+} \]

\[ \text{Gnt2+} \]

\[ \text{dummy-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2-} \]

\[ \text{accept+} \]

\[ \text{ack-} \]

\[ \text{req+} \]

\[ \text{ack+} \]

\[ \text{req-} \]

\[ \text{Req2-} \]

\[ \text{Gnt2-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2+} \]

\[ \text{Req2+} \]

\[ \text{Gnt2+} \]

\[ \text{dummy-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2-} \]

\[ \text{accept+} \]

\[ \text{ack-} \]

\[ \text{req+} \]

\[ \text{ack+} \]

\[ \text{req-} \]

\[ \text{Req2-} \]

\[ \text{Gnt2-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2+} \]

\[ \text{Req2+} \]

\[ \text{Gnt2+} \]

\[ \text{dummy-} \]

\[ \text{accept-} \]

\[ \text{dummy+} \]

\[ \text{Gnt2-} \]

\[ \text{accept+} \]

\[ \text{ack-} \]

\[ \text{req+} \]

\[ \text{ack+} \]

\[ \text{req-} \]

\[ \text{Req2-} \]
It is important to note that the data, which is sampled by the IP module remains in the last stage of the asynchronous FIFO until a new accept+ transition is received. Figure 7.7 shows the reading of two data words. The first data word is read without pausing the clock signal, because the sequence of reading a data word from a non empty FIFO can be done within a clock cycle (in this case). If there is more time needed then a single clock cycle the clock signal is paused. The clock signal is always paused when the FIFO is empty. This is the case for the second data word that is read as shown in Figure 7.7.

![Figure 7.7: timing diagram for the input controller without auxiliary latches](image)

### 7.2.3. Timing specification

Here follows the timing specification of the input controller if no auxiliary latch stage is used. The rd_accept signal is interfaced from the IP module to the input controller and the rd_data signals are interfaced from the asynchronous FIFO to the IP module (see Figure 7.8). The following timing constraints are necessary for the input controller:

1. The accept signal must rise and fall in a single clock cycle (as defined in the STG).
2. When a req transition is received from the asynchronous FIFO the data must be kept stable long enough to be sampled correctly by the receiving flip-flops in the IP module (setup and hold constraints).
3. When the clock signal is paused, no rising edge of the clock signal is allowed after a request for pausing the clock is given to the pausable clock generator.
4. A transition of the ack signal must be given (to empty the last stage of the FIFO) after the rising clock edge in which the data is sampled by the IP module.

Constraint 1 is exactly the same as for the input controller with auxiliary latches and therefore the minimal value for the delay $\tau_1$ can be calculated with:

$$\tau_1 = t_{D,FF_1} + t_{P,logic} + t_{P,wires} + t_{SU,FF_2}$$  \hspace{1cm} (30)

With:

- $t_{D,FF_1}$: the propagation delay of $FF_1$.
- $t_{P,logic}$: the propagation delay of the $logic_1$.
- $t_{P,wires}$: the propagation delay of the wires between $FF_1$ and $FF_2$.
- $t_{SU,FF_2}$: the setup time of $FF_2$.
The reset for $FF_2$ is the same as for the input controller with auxiliary latches.

Timing constraint 2 can be met by delaying the $req$ signal from the asynchronous FIFO to the input controller. When the input controller receives the transition of the $req$ signal, the clock will be enabled again. The delay must be set such that the $rd\_data$ is stable at the sample moment of $FF_3$. Or in other words: the delay must be set such that the $rd\_data$ is stable when the clock is enabled again by the input controller. The minimal delay for $\tau_3$ is calculated with:

$$\tau_3 = t_{su\_f3} + t_{p\_wire} + t_{p\_logic}$$

With:

- $t_{su\_f3}$: the setup time of $FF_3$.
- $t_{p\_wire}$: the propagation delay of the wires between $FF_3$ and the asynchronous FIFO.
- $t_{p\_logic}$: the propagation delay of $logic_2$.

The hold constraint for $rd\_data$ is automatically met if:

$$\tau_3 > h_{f3}$$

With:

- $h_{f3}$: the hold time of $FF_3$.

![Figure 7.8: the paths for the accept and the rd_data signals to and from the input controller if no auxiliary latches are used](image)

For constraint 3 the used pausable clock generator must guarantee that the clock generation is paused when a pause request is received. The time at which the $Req2$ signal rises after the last rising clock edge can be determined with:

$$\tau_2 = \tau_1 + t_{p\_ff2} + t_{p\_controller}$$
With:

- $t_{p\text{-controller}}$: the propagation delay for the input controller to make a low-to-high transition for signal $Req2$.
- $t_{p\text{-} FF2}$: the propagation delay of $FF2$.

Finally constraint 4 is guaranteed by the specification in the STG. Thus the transition of the $ack$ signal is only allowed when a new $accept+$ transition is received.

The practically considerations are the same as for the input controller with auxiliary latches, but now there is another formula for delay $\tau_3$. 

© Koninklijke Philips Electronics N.V. 2004 65
8. Implementation

This Chapter describes the design flow and the tools and libraries that are used to implement the different components used in the top-level architecture. The following components are implemented: a Mutual Exclusion element, a pausible clock generator, an input controller and an output controller. These components are implemented in a 0.13 μm process technology and back-annotated simulations are done.

8.1. Tools and libraries

The tools used for designing the components described in this Chapter are all available within the Philips CAD environment. An exception is the tool Petrify which is freeware. The most important tools used are listed in Table 8.1. The layouts are made with Silicon Ensemble (from Cadence). For Layout Versus Schematic (LVS) checking the tool Calibre (from Mentor Graphics) is used. This tool is also used for Design Rule Checks (DRC) and for checking antennas in the layout.

Table 8.1: versions of the used tool packages

<table>
<thead>
<tr>
<th>VERSION</th>
<th>TOOL PACKAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>01.03.00</td>
<td>PeAssura</td>
</tr>
<tr>
<td>01.05.01</td>
<td>PeCalibre</td>
</tr>
<tr>
<td>3.2</td>
<td>QDF</td>
</tr>
<tr>
<td>3.0.446</td>
<td>cadence assura</td>
</tr>
<tr>
<td>5.0.33.500.0.6</td>
<td>cadence ic</td>
</tr>
<tr>
<td>2003.2</td>
<td>cadence philips</td>
</tr>
<tr>
<td>05.40-S130</td>
<td>cadence sedsm</td>
</tr>
<tr>
<td>9.1 8.2</td>
<td>Calibre</td>
</tr>
<tr>
<td>3.0</td>
<td>Diesel</td>
</tr>
<tr>
<td>04.05.00</td>
<td>gds2util</td>
</tr>
<tr>
<td>4.2</td>
<td>Petrify</td>
</tr>
<tr>
<td>4.6</td>
<td>pstar</td>
</tr>
<tr>
<td>1.1.1 4.4.6</td>
<td>simkit</td>
</tr>
</tbody>
</table>

The designs are implemented in a CMOS 0.13 μm process technology (Philips' CMOS12 technology). The Mutual Exclusion element is designed full custom as standard cell, because this cell is not available in the standard cell library. The I/O controllers and the pausible clock generator are implemented with standard library cells. The used libraries are given in Table 8.2. The Philips Technology Package (PTP) that is used has 6 metal layers (H) and is General Purpose (G). The nominal supply voltage for this technology is 1.2 Volt.

Table 8.2: versions of the used library packages

<table>
<thead>
<tr>
<th>VERSION</th>
<th>LIBRARY PACKAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.1</td>
<td>PcCL013HGPTP</td>
</tr>
<tr>
<td>2.1.1</td>
<td>PcCMOS12corelib</td>
</tr>
</tbody>
</table>
8.2. Design flow

Figure 8.1 shows the design flow that is used to implement the different components. The first part of the flow is used for the I/O controllers only. This part is the specification of a STG and the synthesis with Petrify to equations, these equations are implemented into a gate level verilog netlist (standard cells). The verilog netlist is simulated first to check the behaviour of the design. If this is correct the netlist together with the standard cell libraries is read in by Silicon Ensemble, in which the floorplan is created, the placement is done and the routing is made of the layout. The result is an GDSII file which contains the properties of the layout, but not the layout of the used standard cells. The next step is to collect the GDSII of the design and the GDSII of the standard cells together in a final GDSII file, this is done with the gds2util toolbox.

When the final GDSII is present this file must be verified, which is done with the tool Calibre. The checks that are done are: Layout versus Schematic, Design Rule Check and Antennas checking. When no problems are found during the verification the parasitic components can be extracted with the tool HyperExtract. Furthermore the delays in the design are calculated (for the nominal case only (25 °C and 1.2 Volt power supply) with the tool Pearl. The parasitics are used with the tool Diesel to calculate the power consumption of the design. The delays are used to redo the simulations, but now with timing values from the layout (back-annotated simulations).

The designed components must be reused in real designs and therefore an abstract must be created from the layout of these components. This abstract is created with the tool icfb from Cadence (icfb stands for “ic front to back”). This is done by streaming in the final GDSII file into the tool icfb. Then the pinning must be redrawn in the layout and an abstract is generated. This abstract is exported to a LEF file, which can be read in with Cadence layout tools.

8.3. Mutual Exclusion element

For the implementation of the Mutual Exclusion element the schematic of Figure 2.8 is used. The schematic is implemented as a standard cell layout with the tool icfb (not with the design flow described in the previous Paragraph). The Mutual Exclusion element is “fair” if the layout is symmetric. Fair means that the layout does not influence the decision of which output is granted.
by the Mutual Exclusion element in case both request signals rise at the same moment in time. Figure 8.2 shows the standard cell layout of the Mutual Exclusion element. As can be seen in the layout (and in the schematic of course) the Mutual Exclusion element is build up out of 12 MOS transistors. The standard cell has a width of 4.92 \( \mu \text{m} \) and a height of 4.92 \( \mu \text{m} \). This makes a total cell area of 24.2064 \( \mu \text{m}^2 \). To verify the behaviour of the standard cell \( F_{\text{star}} \) simulations are done with the extracted layout.

![Figure 8.2: standard cell layout of the Mutual Exclusion element](image)

If this Mutual Exclusion element is used in other designs an abstract of the layout must be available and a timing characterisation of the standard cell must be available. The abstract can be generated with \textit{icfb} as described in Paragraph 8.2. The timing characterisation is defined in a TLF file, this file can not be generated automatically. Instead first the characterization of the Mutual Exclusion element (for the nominal case) is done with the tool \textit{CHARLI} [21]. This tool performs simulations with different capacitive loads for the outputs and different slopes for the input signals. The results of this tool are pasted in a template TLF file. Some specific properties for the Mutual Exclusion element must be put in this TLF file as well.

### 8.4. Pausible clock generator

The pausible clock generator is implemented according to the schematic of Figure 5.11. Thus two I/O controllers can be interfaced to this clock generator, because it has two request/grant pairs. Therefore this component needs two Mutual Exclusion elements. The netlist of the pausible clock generator is made by hand and contains only standard library cells and the two Mutual Exclusion Elements. The layout of the pausible clock generator is shown in Figure 8.3. The floorplan has 4 standard cell rows and there is no power ring used. The advantage of this layout is that it can be placed in a standard cell row (with a height of four rows) when it is used in a new design. Thus automatic placement can be used with this component. The two Mutual Exclusion elements are placed in the lowest row to the left side. In the layout decoupling capacitances are inserted to prevent power supply drops. The layout has a width of 40.8 \( \mu \text{m} \) and a height of 19.68 \( \mu \text{m} \), which makes a total area of 790.742 \( \mu \text{m}^2 \).

In Figure 8.4 a back-annotated simulation of the pausible clock generator is shown. The clock generation is first paused with the \( \text{Req1} \) signal and after that again with the \( \text{Req2} \) signal. The simulation shows that the clock is paused during the low phase of the clock period. Furthermore the simulation shows that \( \text{Req1} \) rises during the high phase of the internal signal \( \text{clk}_d \). As a result the \( \text{Gnt1} \) signal rises later, because the \( \text{clk}_d \) signal must fall first. The \( \text{Req2} \) signal rises during the low phase of \( \text{clk}_d \) and now the \( \text{Gnt2} \) signal rises immediately.
The delay line of the ring oscillator is implemented by concatenating NAND gates and the used clock period is 2.95 ns, about 339 MHz (three different frequencies can be programmed with the delayline). What is important to note is that due to the usage of NAND gates the duty cycle of the generated clock cycle is not 50%. This is caused by the different fall and rise time of the output of the NAND gate. The generated clock signal has a longer high phase than a low phase. For setting timing constraints and timing verification it is better to design a delay line that has the duty cycle of 50%.

8.5. Output controller

The output controller that is implemented is described in Chapter 6. The AFSM of the output controller is synthesised to equations and these equations are converted by hand to a gate level netlist. Next to the AFSM some surrounding logic must be added as well. The resulting layout after finishing the design flow is shown in Figure 8.5. The same approach as for the pausable clock generator is taken for making the layout. So there is no power ring and decoupling capacitances are inserted. The layout has a width of 30.34 μm and a height of 19.68 μm, which makes a total area of 597.091 μm².

Figure 8.6 shows a back-annotated simulation of the output controller. The simulation shows two control sequences of the output controller. The control signals behave as specified in Chapter 6 and the second control sequences shows that the transitioning signalling with the asynchronous FIFO behaves correct as well.
With the simulation timing values for the output controller can be measured. The values for the first sequence (with $req^+$ and $ack^+$ transitions, see STG in Paragraph 6.2) are:

- $wr\_valid^+ \rightarrow valid^+$ takes 240 ps.
- $valid^+ \rightarrow Req1^+$ takes 370 ps.
- $Gnt1^+ \rightarrow req^+$ takes 470 ps.
- $ack^+ \rightarrow Req1^-$ takes 580 ps.

And the timing for the second sequence (with $req^-$ and $ack^-$ transitions) are:

- $wr\_valid^+ \rightarrow valid^+$ takes 240 ps.
- $valid^+ \rightarrow Req1^+$ takes 370 ps.
- $Gnt1^+ \rightarrow req^-$ takes 510 ps.
- $Ack^- \rightarrow Req1^-$ takes 710 ps.

The second sequence is slower than the first sequence with a cycle time of 1.83 ns. The time consumed for handshaking with the pausable clock generator and for the handshaking with the asynchronous FIFO is not included in the cycle time. The cycle time is not bad, but one must take into account that there is only a half clock cycle available for the complete sequence (see Paragraph 6.3). Thus if a worst case value of say 3 ns is taken for the cycle time (including all
handshaking) a maximum clock frequency of 167 Mhz can be used for the IP module so that the
clock generation is not paused with write actions. Of course the clock generation is always
paused when there is written to a full FIFO. If a higher clock frequency is used the clock will be
paused with every write action. The speed of the output controller can be improved by designing
it full custom instead of using standard library cells. It is obvious that the performance of the IP
module is improved by this when higher frequencies must be used.

8.6. Input Controller

The input controller that is implemented is the one described in Paragraph 7.2, which is the input
controller without auxiliary latches. The input controller is implemented with the same design
flow and approach as the output controller. The STG for the AFSM of the input controller is
complexer than the one of the output controller. Therefore the input controller needs a larger area
than the output controller. The layout is shown in Figure 8.7 and has a width of 40.8 μm and a
heigth of 19.68 μm, which makes an area of 790.742 μm².

Figure 8.7: layout of the input controller

Figure 8.8 shows the back-annotated simulation of the input controller. In the simulation two
control sequences are shown. Both sequences are as defined in the behaviour and the timing of
the input controller. With the simulation timing values for the input controller can be measured.
The values for the first sequence (with req+ and ack- transitions, see STG in Paragraph 7.2.2)
are:

- \(rd_{\text{accept}}+ \rightarrow \text{accept}+\) takes 230 ps.
- \(\text{accept}+ \rightarrow \text{Req}2+\) takes 370 ps.
- \(\text{accept}+ \rightarrow \text{ack}{}-\) takes 420 ps.
- \(\text{req}+ \rightarrow \text{Req}2-\) takes 710 ps.

And the timing for the second sequence (with req- and ack+ transitions) are:

- \(rd_{\text{accept}}+ \rightarrow \text{accept}+\) takes 230 ps.
- \(\text{accept}+ \rightarrow \text{Req}2+\) takes 370 ps.

©Koninklijke Philips Electronics N.V. 2004
• accept+ → ack+ takes 370 ps.
• Req- → Req2- takes 750 ps.

The first control sequence is the slowest with a cycle time of 1.73 ns. Again the time necessary for handshaking with the pausable clock generator and the asynchronous FIFO are not taken into account. This means that the input controller is 100 ps faster than the output controller. The considerations for clock frequencies higher than 167 Mhz that are given for the output controller can be applied to this input controller as well (if the cycle time of 3 ns is taken). Now the clock generation is always paused when the IP module reads a data word when the FIFO is empty. If the control sequence takes longer than the clock cycle of the IP module the clock is paused for every read action.

![Figure 8.8: back-annotated simulation of the input controller](image)

8.7. Simulation of a channel

To simulate the interaction of the different components a simple channel is made, which consists of an input and an output port. The channel, which is embedded in a testbench, is shown in Figure 8.9. The two FIFOs that are used are implemented with standard cells and no back-annotation is used for these components. Both FIFOs have only two MOUSETRAP stages. All components in the channel need an active low reset (not shown in the Figure). The signals, which must be interfaced to the IP module are: clk, wr_valid, wr_data, rd_accept and rd_data.

![Figure 8.9: the channel and the used testbench](image)

The signals that must be interfaced to the Kernel of the Network Interface are:

• F_rreq: the request signal from the source queue to the Kernel.

© Koninklijke Philips Electronics N.V. 2004
• $F_{\text{rack}}$: the acknowledge signal from the Kernel to the source queue.
• $F_{\text{rddata}}$: the data signals from the source queue to the Kernel.
• $F_{\text{wreq}}$: the request signal from the Kernel to the destination queue.
• $F_{\text{wack}}$: the acknowledge signal from the destination queue to the Kernel.
• $F_{\text{wrdata}}$: the data signals from the Kernel to the destination queue.

Different simulations are done with this testbench by using different stimuli for the inputs and observing the outputs of the testbench. These simulations are described in the next Paragraphs.

8.7.1. IP module writes data

Figure 8.10 shows the back-annotated simulation of the IP module writing data to the source queue. The complete channel is reset first with the $rst_{\text{an}}$ signal. When the reset is done the pausable clock generator starts the clock with a clock frequency of 339 MHz. Furthermore the asynchronous control signals are reset.

![Figure 8.10: IP writes data to the source queue](image)

The simulation is zoomed in as shown in Figure 8.11. The different steps in the simulation after the reset are as follows (including the responses):

• The IP module writes three data words to the source queue. The values of the data words are $1234_{\text{hex}}$, $5678_{\text{hex}}$ and $9ABC_{\text{hex}}$ respectively.

• The first two data words are written immediately to the source queue. During these two write actions the clock is paused for a certain time interval, because the write sequence takes longer than a clock cycle.

• The third data word can not be written to the source queue, because the queue is full. As a result the clock is paused until there is space in the source queue again.

• When the Kernel reads the first data word from the source queue, the IP module can write the third data word ($9ABC_{\text{hex}}$) to the source queue and the clock generation is started again. As shown in Figure 8.11 the first word read by the Kernel is (as expected) $1234_{\text{hex}}$.

• The second and third data word read by the Kernel are $5678_{\text{hex}}$ and $9ABC_{\text{hex}}$ respectively.

© Koninklijke Philips Electronics N.V. 2004
8.7.2. IP module reads data

Figure 8.12 shows the back-annotated simulation of the IP module reading data from the destination queue. First the channel is reset and the clock generation is started. The \textit{ack} signal is reset to logical one, which makes the last stage of the FIFO (destination queue) full. Of course the data in this last stage is useless after a reset, but now the input controller has control over the last stage of the FIFO. When the IP module wants to receive data the \textit{ack} signal will toggle to empty the last stage of the destination queue. Now valid data can be shifted into this last stage of the FIFO, which can be sampled by the IP module.

The simulation is zoomed in as shown in Figure 8.13. The different steps in the simulation after the reset are as follows (including the responses):

- Two data words are written by the Kernel to the destination queue. These words have the values: $1234_{\text{hex}}$ and $5678_{\text{hex}}$ respectively. The second data word can not be written immediately, because the input controller blocks the last stage of the destination queue. The second data word can be written when the first data word is read by the IP module.
- The IP module wants to read three data words from the destination queue.
- The first two data words are read immediately. The read-sequence causes the clock generation to be paused for a certain time interval. As shown in Figure 8.13 the data words that are read are correct ($1234_{\text{hex}}$ and $5678_{\text{hex}}$ respectively). When data word $1234_{\text{hex}}$ is read, data word $5678_{\text{hex}}$ can be written to the destination queue by the Kernel.
The third data word cannot be read, because it is not yet in the destination queue. Therefore the clock generation is paused until the data word is written to the destination queue.

The data word $9\text{ABC}_{\text{hex}}$ is written to the destination queue by the Kernel. As a result this data word is read by the IP module and the clock generation is started again.

Figure 8.13: IP reads data from the destination queue (zoomed in)

8.7.3. IP module writes and reads in the same clock cycle

The IP module must also be able to write and read to the Kernel of the Network Interface in the same clock cycle. Figure 8.14 shows a back-annotated simulation in which the IP module writes and reads in the same clock cycle.

Figure 8.14: IP writes and read data in the same clock cycle

The steps during this simulation are as follows:

- Two data words ($1\text{234}_{\text{hex}}$ and $5\text{678}_{\text{hex}}$) are written to the destination queue by the Kernel. The second data word can only be written to the destination queue if the first data word is read by the IP module.

- The IP module reads the first data word from the destination queue ($1\text{234}_{\text{hex}}$) and at the same time the IP module writes a data word to the source queue ($1\text{234}_{\text{hex}}$). During these two actions the clock generation is paused for a certain time interval.

- In the next clock cycle the IP module reads a data word from the destination queue and writes a data word to the source queue. Both data words have the value $5\text{678}_{\text{hex}}$. 
- The third data word is read from the destination queue, but at the moment of reading the destination queue is empty. Therefore the clock signal is paused until a data word is written by the Kernel. In this case the data word 9ABC_{hex} is written by the Kernel and this data word is read by the IP module. As a result the clock generation is started again.

- The two data words that are written to the source queue are read by the Kernel. The values of these data words are: 1234_{hex} and 5678_{hex} respectively.
9. Conclusions

The problem of interfacing an IP module that uses pausable clocking for synchronization to an \textit{Æ}thereal network of which the clock generation cannot be paused, is studied and a solution is provided. First a top-level architecture is chosen in which asynchronous as well as synchronous components are used. The protocol translations shells in the Network Interface (NI) are shifted towards the IP module, because they use the same clock domain as the IP module. The IP module gets a wrapper with a pausable clock generator, per input port an asynchronous input controller and per output port an asynchronous output controller. Every input and output port is interfaced to an asynchronous FIFO that uses 2-phase handshaking. At the NI side synchronizers are used for synchronization, in total three different synchronizers are necessary for a single queue. These synchronizers are used to:

- Interface and synchronize the asynchronous FIFOs to the NI.
- Synchronize the signals necessary to count the number of valid data words in the FIFOs.
- Synchronize the Flush signals.

The asynchronous I/O controllers are described with STGs to specify the behaviour of these components. For the pausable clock generator of the IP modules a standard solution is chosen, which is proven via a STG, that describes the behaviour of this pausable clock generator.

For the I/O controllers timing constraints must be set to allow a proper timing between the IP module and the asynchronous FIFO. Furthermore timing constraints must be set for the signals that request to pause the clock generation.

9.1. Consequences

If the solution is used it has several consequences for the design of a SoC. The consequences that are known at this point of the research are:

- At the IP side metastability can be prevented completely. However timing constraints for the request signals for pausing the local clock generation must be set to achieve this. In a design these timing constraints can be applied, because the delay for the request signals is known and the rising of the request signals is related to the clock signal.

- At the NI side there is still metastability possible, because synchronizers are used here. Per channel there are 3 synchronizers necessary: one for the asynchronous FIFO, one for the counting of the number of valid words in the FIFO and one for the flush signal.

- The clock frequency of the IP module must be lower than the clock frequency of the network. This is necessary, because otherwise the synchronizers at the NI side can miss events from the IP module.

- The performance of the IP module will degrade if the control sequence for writing or reading data to or from the asynchronous FIFO takes longer than a clock cycle. If this is the case the local clock generation will pause until the write or read cycle is complete.

- The current solution is less suitable for IP modules that can use multi-threading, because the clock will be paused when the IP module wants to send data to a FIFO that is full or when the IP module wants to read data from an empty FIFO. This means that a multi-
threading IP can not continue with threads that are not waiting for data or waiting to send data.

- The original FIFOs in the NI are replaced with asynchronous FIFOs that use transition signalling. The FIFOs for the destination queues should be made one stage larger. This extra stage is necessary to keep the data that has to be read by the IP module stable such that it can be sampled correctly.

### 9.2. Implementation

The following components are implemented: a Mutual Exclusion element, a pausible clock generator, an input controller and an output controller. These components are implemented in a 0.13 μm process technology with standard library cells (Philips' CMOS12 technology). With the netlists and the layouts back-annotated simulations are done. The Mutual Exclusion element is designed as standard cell, because this cell is not available in the standard cell library. The Mutual Exclusion element is used in the pausible clock generator.

The area necessary to implement the components is low. For a pausible clock generator, an input controller and an output controller an area of 0.002 mm² is necessary. The sequence of writing a data word to the FIFO (source queue) takes 1.83 ns and the sequence of reading a data word from a FIFO (destination queue) takes 1.73 ns. Both values are without the time necessary to handshake with the FIFO and the time necessary to arbitrate the pausing of the clock generation.

### 9.3. Future work

Future work can be done on improving the current solution, doing more research on the impact at system level and work on implementation issues of the solution. Currently the solution is not mature enough to be transferred directly to SoC design. The following future work can be considered:

- Doing research about the performance of the IP module related to the number of communication channels per IP module and the depth of the FIFOs of these communication channels (system level issues).
- Doing research about the robustness of the synchronizers and how to minimize the metastability or if possible remove the metastability at the NI side by using more advanced synchronization techniques.
- Improve the design of the asynchronous controllers by implementing them full custom instead of using standard library cells.
- Adding functional and structural test to the solution.
- Incorporate the solution in the design flow.
References

Asynchronous wrapper for heterogeneous systems.

Combining asynchronous and synchronous circuits using stretchable clocks.

[3] Chapiro, D.M.
Globally-Asynchronous Locally-Synchronous Systems.
Stanford University, October, 1984.
Doctoral dissertation.

Petriify: a tutorial for the designer of asynchronous circuits.
Universitat Politècnica de Catalunya, Department of Software.

[5] Dally, W. J. and J.W. Poulton
Digital Systems Engineering.

Concepts and Implementation of the Philips Network-on-Chip.

Data Synchronization Issues in GALS SoCs.

[8] Foley, C.
Characterizing Metastability: Practical Measurement Techniques to accurately determine "device dependent coefficients" used to predict synchronizer MTBF.

A Predictive Synchronizer for Periodic Clock Domains.
http://www-ee.technion.ac.il/

[10] Ginosar, R.
Fourteen Ways to Fool Your Synchronizer.

Asynchronous Design Methodologies: An Overview.

[12] Kaam, K.M.M. van
Globally Asynchronous Locally Synchronous and Networks on Chip: A literature search.
Kessels, J. and A. Peeters, S.J. Kim
Bridging Clock Domains by synchronizing the mice in the mousetrap.

Meijer, M. and F. Pessolano, J. Pineda de Gyvez
Technology Exploration for Adaptive Power and Frequency Scaling in 90nm CMOS.

Moore, S. and G. Taylor, R. Mullins, P. Robinson
Point to point GALS interconnect.

Muttersbach, J. and T. Villiger, W. Fichtner
Practical design of globally-asynchronous locally-synchronous systems.

Muttersbach, J. and T. Villiger, H. Kaeslin, N. Felber, W. Fichtner

Öberg, J.
Clocking Strategies for Networks-on-Chip.

Philips Semiconductors, Chief Technology Office, System ASIC Technology
Inter-Clock Domain Communication Techniques.

Philips Semiconductors, Chief Technology Office, System ASIC Technology
Bus Communication between Islands of Synchronicity.

Philips Semiconductors
CHARLI User Guide; A system to CHARACTERise Libraries.

Rădulescu A. and J. Dielissen, K. Goossens, E. Riijkema, P. Wielage

Rădulescu A. and J. Dielissen, K. Goossens, E. Riijkema, P. Wielage
Ethereal: Network Interface Design.

Seizovic, J.N.
Pipeline synchronization.
Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture.

[26] Semeraro, G. and G. Magklis, R. Balasubramonian et al
Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling.

[27] Singh, M. and S.M. Nowick
MOUSETRAP: Ultra-high-speed transition-signalling asynchronous pipelines.


[29] Vanbekbergen, P. and C. Ykman-Couvreur, B. Lin, H. De Man
A Generalized Signal Transition Graph Model for Specification of Complex Interfaces.

Specification and Analysis of Timing Constraints in Signal Transition Graphs.

[31] Vanbekbergen, P. and G. Goossens, B. Lin
Modeling and Synthesis of Timed Asynchronous Circuits.

Self-timed Ring of Globally-Asynchronous Locally-Synchronous Systems.

[33] Wielage P. and R. Salters, J. Dielissen, K. Goossens
An area-efficient asynchronous pipeline FIFO and its use in a synchronous environment.

[34] Yun, K.Y. and R.P. Donohue
Pausible clocking: A first step towards heterogeneous systems.

Pausible clocking based heterogeneous systems.
A Example Petrify: Muller C-element

```
.model muller_c
# Declaration of signals
.inputs a b
.outputs z
# Petri net
.graph
b+ z+
a+ z+
z+ b- a-
b- z-
a- z-
z- b+ a+
#initial marking
.marking {<z-,b+> <z-,a+>}
.end
```

Box 9.1: input file for Petrify with the description of the STG

```
INPUTS: a, b
OUTPUTS: z
```

Figure 9.1: generated STG with petrify

The plot in Figure 9.1 is generated with executing the following command:
draw_astg -nofold muller_c.g -o muller_c.g.ps

With the following three commands the State Graphs are created:
write_sg muller_c.g -o muller_c.sg
write_sg muller_c.g | draw_astg -sg -o muller_c.sg.ps
write_sg muller_c.g -bin | draw_astg -sg -o muller_c_bin.sg.ps
The first command will generate the State Graph in a text file and the second command will generate a plot of the State Graph as shown in Figure 9.2. The last command also generates a plot of the State Graph, but now the state decoding is also included (see Figure 9.3).

Finally the Equations of the Muller C-element can be generated with the command:

```
petrify -csc muller_c.g -eqn muller_c.eqn
```

Note the option `-csc`, which is used to automatically solve the CSC problem. In this case there is no CSC problem, but at least it is checked. The resulting equations are shown in Box 9.2. From this file it is not immediately clear that the equation behaves as a Muller C-element. If the output \( z \) is rewritten to the equation:

\[
Z = b \ a + b \ z + a \ z
\]
With the variable $z$ on the right hand side the previous value of $Z$ is somewhat easier to understand. If it is not clear enough one can write down the truth table.

\[
\begin{align*}
\text{INORDER} &= a \ b \ z; \\
\text{OUTORDER} &= [z]; \\
[z] &= b \ (a + z) + a \ z;
\end{align*}
\]

*Box 9.2: resulting equation file*
B Synthesising an output controller

```
.model output_port
# Declaration of signals
.inputs valid Gnt1 ack
.outputs Req1 req
.internal dummy

# Petri net
.graph
valid+/1 Req1+/1
Req1+/1 Gnt1+/1
Gnt1+/1 req+/1
req+/1 ack+/1
ack+/1 Req1-/1
valid+/1 dummy+/-1
dummy+/-1 valid+/-1
Gnt1+/-1 valid+/-1
Req1+/-1 Gnt1+/-1
Gnt1+/-1 valid+/-1
Req1+/-1 dummy+/-1
Req1+/-1 Gnt1+/-1
Req1+/-1 dummy+/-1
req+/2 Req1+/-2
Req1+/-2 Gnt1+/-2
Req1+/-2 dummy+/-2
dummy+/-2 valid+/-2
Gnt1+/-2 req+/2
req+/2 ack+/2
ack+/2 Gnt1+/-2
valid+/-2 dummy+/-2
dummy+/-2 valid+/-1
Gnt1+/-2 valid+/-1

#initial marking
.marking { <Gnt1-/2.valid+/1> <dummy+/2.valid+/1> }
.end
```

Box 9.3: input file for Petrify for the output controller

```
# EQN file for model output_port
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
INORDER = valid Gnt1 ack Req1 req dummy csc0 csc1;
OUTORDER = [Req1] [req] [dummy] [csc0] [csc1];
[0] = valid (ack' csc0' csc1' + ack csc0 csc1);
[1] = ack csc0' + ack' csc0;
[Req1] = [1]' ([0] + Req1) + Req1 [0];          # mappable onto gC
[3] = Gnt1 csc0';
[4] = Gnt1 csc0;
[req] = [4]' ([3] + req + req [3]);          # mappable onto gC
[6] = csc1 (Req1 csc0' + Req1' req) + csc1' (Req1' req' + Req1 csc0);
[7] = valid [req csc1' + req' csc1 + Req1];
[dummy] = [7]' ([6] + dummy) + dummy [6];    # mappable onto gC
[9] = Gnt1' ack;
[10] = Gnt1' ack';
[csc0] = [10]' ([9] + csc0) + csc0 [9];      # mappable onto gC
[12] = valid' (Req1' req + Req1 csc0');
[13] = valid' (Req1' req + Req1 csc0);
[csc1] = [13]' ([12] + csc1) + csc1 [12];    # mappable onto gC

# Set/reset pins: reset(Req1) reset(req) set([6])
```

Box 9.4: equation results file for the output controller
SET(Reql') = ack csc0' + ack' csc0
RESET(Reql') = valid (ack' csc0 csc1' + ack csc0 csc1)

[Reql] = Req1 (output inverter)
  > triggers(SET): ack+/2 -> Req1-/2
  > triggers(RESET): (valid+/2,csc0-) -> Req1+/2
  > 11 transistors (7 n, 4 p) + 4 inverters
  > Estimated delay: rising = 43.33, falling = 28.25

SET(req') = Gntl csc0
RESET(req') = Gntl csc0'

[req] = req' (output inverter)
  > triggers(SET): Gntl+/2 -> req-/2
  > triggers(RESET): Gntl+/l -> req+/l
  > 4 transistors (2 n, 2 p) + 3 inverters
  > Estimated delay: rising = 23.46, falling = 26.00

SET(dummy') = valid (req csc1' + req' csc1 + Req1)
RESET(dummy') = csc1 (Req1' req + Req1 csc0') + csc1' (Req1' req' + Req1 csc0)

[dummy] = dummy' (output inverter)
  > triggers(SET): Req1+/2 -> dummy-/1 Req1+/2 -> dummy+/2
  > triggers(RESET): csc1+ -> dummy+/1 csc1- -> dummy+/2
  > 16 transistors (10 n, 6 p) + 6 inverters
  > Estimated delay: rising = 38.88, falling = 30.33

SET(csc0') = Gntl' ack'
RESET(csc0') = Gntl' ack

[csc0] = csc0' (output inverter)
  > triggers(SET): Gntl-/2 -> csc0-
  > triggers(RESET): Gntl-/l -> csc0+
  > 4 transistors (2 n, 2 p) + 2 inverters
  > Estimated delay: rising = 27.83, falling = 22.88

SET(csc1') = valid' (Req1' req' + Req1 csc0)
RESET(csc1') = valid' (Req1' req + Req1 csc0')

[csc1] = csc1 (output inverter)
  > triggers(SET): valid-/2 -> csc1-
  > triggers(RESET): valid-/l -> csc1+
  > 10 transistors (5 n, 5 p) + 4 inverters
  > Estimated delay: rising = 31.96, falling = 29.75

Box 9.5: summary of gates for the output controller
C  Synthesising an input controller with auxiliary latches

```
.model input_port
#Declaration of signals
.inputs req Gnt2 accept en_d
.outputs Req2 ack en
.internal dummy

# Petri net
.graph accept+/2 Req2+/1
ack- req+
Req2+/1 Gnt2+/1
Req2+/1 dummy-/1
dummy-/1 accept-1
accept-1 dummy+/1
dummy+/1 accept+/1
accept+/1 Req2+/2
Gnt2+/1 en+1
en+/1 en_d+/1
en_d+/1 Req2-1
req+ en+/1
Req2-1 Gnt2-1
Gnt2-1 en-1
en-1 en_d-/1
en_d-/1 ack+
ack- req-
req- en+/2
en+/2 en_d+/2
Gnt2-/2 Req2-2
Req2-2 dummy-/2
dummy-/2 accept-/2
accept-/2 dummy+/2
dummy+/2 accept+/2
Req2+/2 Gnt2+/2
Gnt2+/2 en+/2
en_d+/2 Req2+/2
Gnt2+/2 en-/2
Gnt2-2 Req2+/1
en-2 en_d-/2
en_d-/2 ack-

#initial marking
.marking { <ack-,req> <Gnt2-1,Req2+/2> <dummy+/2,accept+/2> }
.end

Box 9.6: input file for Petrify for the input controller with auxiliary latches

```

```
# EQN file for model input_port
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)

INORDER = req Gnt2 accept en_d Req2 ack en csc0 csc1;
OUTORDER = [Req2] [ack] [en] [csc0] [csc1];
0 = accept (csc0 csc1 (en_d' + req) + csc0' csc1' (req' + en_d'));
1 = en_d (req csc1' + req' csc1);
[Req2] = [1]' ([0] + Req2) + Req2 [0];
[3] = en_d' csc1;
[4] = en_d' csc1';
[ack] = [4]' ([3] + ack) + ack [3];
[6] = Gnt2 (req csc1' + req' csc1);
[7] = req' csc1' + req csc1;
[en] = [7]' ([6] + en) + en [6];
[9] = accept' (Req2 (en_d req + csc1' en_d') + Req2 csc1);
[csc0] = [10]' ([9] + csc0) + csc0 [9];
```

© Koninklijke Philips Electronics N.V. 2004
Box 9.7: equation results file for Petrify for the input controller with auxiliary latches

Box 9.8: summary of gates for the input controller with auxiliary latches
D Synthesising an input controller without auxiliary latches

```
.. model input_port
   # Declaration of signals
   .. inputs req Gnt2 accept
   .. outputs Req2 ack
   .. internal dummy

   # Petri net
   .. graph
   accept+/2 ack-
   accept+/2 Req2+/1
   ack- req-
   Req2+/1 Gnt2+/1
   Req2+/1 dummy+/1
   dummy-/1 accept-/1
   accept-/1 dummy+/1
   dummy+/1 accept+/1
   accept+/1 ack+
   ack- req-
   Req2-/2
   # Gnt2+/1 Req2+/1 req+ Req2+/1
   Req2-/1 Gnt2-/1
   accept+/1 Req2+/2
   Gnt2-/1 Req2+/2
   Req2+/2 dummy-/2
   dummy-/2 accept+/2
   accept-/2 dummy+/2
   dummy+/2 accept+/2
   Req2+/2 Gnt2+/2
   Gnt2+/2 Req2+/2
   Req2-/2 Gnt2-/2
   Gnt2-/2 Req2+/1
   req- ack-
   req+ ack+

   # Initial marking
   .. marking { <req-.ack-> <Gnt2-/2,Req2+/1> <dummy+/2,accept+/2> }
   .. end
```

Box 9.9: input file for Petrify for the input controller without auxiliary latches

```
# EQN file for model input_port
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)

INORDER = req Gnt2 accept Req2 ack dummy csc0 csc1 csc2 csc3;
OUTORDER = [Req2] tack[ dummy] [csc0] [csc1] [csc2] [csc3];
[0] = accept dummy (csc0' csc3' + csc0 csc3);
[1] = Gnt2 (csc0' csc3' + req') + csc0 csc1' (csc2 + req);
[Req2] = [1]' ([0] + Req2) + Req2 [0];
[ack] = csc2;
[4] = accept' (csc1' csc3' + csc1 csc3);
[5] = csc1 csc3' + csc1' csc3;
[dummy] = [5]' ([4] + dummy) + dummy [4];
[7] = Gnt2' Req2' csc1;
[8] = Gnt2' Req2' csc1;
[csc0] = [8]' ([7] + csc0) + csc0 [7];
[10] = dummy' csc3;
[11] = dummy' csc2;
[12] = req' csc1' dummy;
[csc1] = [11]' ([10] + csc1) + csc1 [10];
[13] = req csc1' dummy accept;
[14] = req' csc1 dummy accept;
[csc2] = [14]' ([13] + csc2) + csc2 [13];
[Req2] = csc0' csc2;
[16] = Req2 csc0 csc2;
[csc3] = [17]' ([16] + csc3) + csc3 [16];

# Set/reset pins: reset(Req2) set(csc1) set(csc2) set(csc3)
```

Box 9.10: equation results file for Petrify for the input controller without auxiliary latches
Box 9.11: Summary of gates for the input controller without auxiliary latches

```
SET(Req2') = Gnt2 (csc0' csc3 (csc2' + req') + csc0 csc3)  
RESET(Req2') = dummy accept (csc0' csc3' + csc0 csc3)  
[Req2] = Req2'  

> triggers (SET):  
(req-,Gnt2+/2,csc3+) -> Req2-/2  
(req+,Gnt2+/1,csc3-) -> Req2-/1  

> triggers (RESET):  
(accept+/2,csc0+) -> Req2+/1  
(accept+/1,csc0-) -> Req2+/2  

> 15 transistors (6 n, 9 p) + 6 inverters  
> Estimated delay: rising = 41.58, falling = 50.75

ack = csc2  

> triggers (SET):  
csc2+ -> ack+  
csc2- -> ack-  
> 2 transistors (1 n, 1 p) + 1 inverters  
> Estimated delay: rising = 16.00

SET(dummy') = csc1 csc3' + csc1' csc3  
RESET(dummy') = accept' (csc1' csc3' + csc1 csc3)  
[dummy] = dummy'  

> triggers (SET):  
triggers (RESET):  
(csc3- -> dummy-/1 csc3+ -> dummy-/2  
(accept-/1,csc1-) -> dummy+/1 (accept-/2,csc1+) -> 

dummy+/2  

> 9 transistors (5 n, 4 p) + 4 inverters  
> Estimated delay: rising = 37.08, falling = 29.50

SET(csc0') = Gnt2 Req2' csc1'  
RESET(csc0') = Gnt2' Req2 csc1  
[csc0] = csc0'  

> triggers (SET):  
(Gnt2-/2,csc1-) -> csc0-  
> triggers (RESET):  
(Gnt2-/2,csc1+) -> csc0+  
> 6 transistors (3 n, 3 p) + 3 inverters  
> Estimated delay: rising = 33.46, falling = 33.50

SET(csc1') = dummy' csc2'  
RESET(csc1') = dummy' csc3  
[csc1] = csc1'  

> triggers (SET):  
(dummy-/1 -> csc1-  
> triggers (RESET):  
dummy-/2 -> csc1+  
> 4 transistors (2 n, 2 p) + 2 inverters  
> Estimated delay: rising = 27.83, falling = 22.88

SET(csc2') = req' csc1 dummy accept  
RESET(csc2') = req csc1' dummy accept  
[csc2] = csc2'  

> triggers (SET):  
(accept+/2,req-) -> csc2-  
> triggers (RESET):  
(req+,accept+/1) -> csc2+  
> 8 transistors (4 n, 4 p) + 4 inverters  
> Estimated delay: rising = 34.71, falling = 47.25

SET(csc3') = Req2 csc0 csc2'  
RESET(csc3') = Req2 csc0' csc2  
[csc3] = csc3'  

> triggers (SET):  
(Req2+/2,csc2-) -> csc3-  
> triggers (RESET):  
(Req2+/2,csc2+) -> csc3+  
> 6 transistors (3 n, 3 p) + 3 inverters  
> Estimated delay: rising = 29.08, falling = 36.62

© Koninklijke Philips Electronics N.V. 2004
```