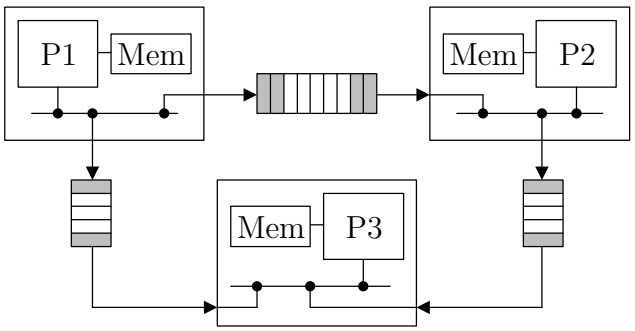
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Window Based FIFOs for Communication in On-Chip Multiprocessor Systems

Hardware Design, Testing Strategies and Design Flow Integration.




Master's Thesis by
David Grünert, 1 April – 30 Sept. 2006

Tutor: Kai Huang
Supervisor: Prof. Dr. Lothar Thiele

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

## Abstract

This thesis introduces a new communication model for on-chip multi-processor systems. The model is based on a high level interface that defines both the data transfer and the synchronisation of the connected processors. It was designed to simplify the reuse of hardware and software components and to support the automation of the design flow. On the one hand, these improvements help to minimise the time-to-market and the design costs, on the other hand it also leads to designs of higher quality because it avoids error-prone low level interfaces and ad-hoc synchronisations. The model of computation is based on an extension of Kahn process networks (KPN). It is therefore particularly suitable for the design of stream-based applications in signal processing.

The thesis includes the description of the new concept, the hardware implementation of the new buffer type for an FPGA architecture, the definition and implementation of the API for buffer access, the implementation of an automated testing environment, a tool for automated system design and the discussion of theoretical aspects of WFIFO buffers such as memory requirements and determinateness.

Zurich, 30 September 2006

David Grünert

# Contents

# 1 Introduction

Today's signal processing systems such as mobile-phones and set-top boxes have requirements that can not be satisfied with a single-processor architecture. The primary reasons for choosing a multiprocessor architecture are that a single-processor has not enough computing power or that it's power consumption is too high. But the performance requirements can not be the only reason for the trend to multiprocessor architectures. It can be observed that dedicated components are replaced by general purpose processors whenever this can be done with acceptable effort.[1] This trend is clearly not motivated by the need for more computing power, because dedicated hardware usually is more powerful than a general purpose processor of similar size.

Although the required computing power and power consumption are important properties for the system design, it emerged that other points in architecture and software design are at least equally important for the success of the designed product. Depending on the field of activity, these points are known under different key words. Where the management talks about time to market, design costs or market size, engineers understand reuse, short and automated design cycles and programmability. Although this thesis does not discuss marketing aspects, engineers can benefit from this similarity of goals when explaining the necessity for a development step to the management. Independently from the professional background, the concepts aim at a more flexible solution while minimising the design time. The following list summarises these trends and their motivation.

**Software Solutions:** Dedicated hardware is replaced by software running on processors. Programmable systems simplify upgrades, bug fixes and make it possible to provide a big array of products that are all based on the same hardware.

**Reuse:** The reuse of hardware leads to shorter design time. No time is used for the hardware design of the reused blocks and the verification is limited to tests that check whether they are properly connected and configured. It is not necessary to test the complete functionality of the reused block if it was designed properly. Reuse also has a software aspect. In most current designs, much more engineering power is used for software design than for hardware design. Changes in the hardware can have major impact on the software and cause high costs.

**Automation:** In the design cycles many steps are automated. Examples are automated testing environments and the automated system assembly from IP libraries. The shorter design cycles made possible by the automation allow multiple redesigns before the product is launched, which leads to better quality. On the other hand, it is possible to design a first version in a very short time. This version does not necessarily include the complete functionality but in many cases it is good enough for the customer to make first tests.

[1] For high performance parallel systems, a similar trend can be observed. Parallel systems with specialised hardware and architectures are replaced by clusters built from off-the-shelf computing systems.

In this thesis, a solution for the implementation of signal processing systems that follows these trends is presented. A special property of signal processing systems is that they often have a stream-based or pipelined structure. An input signal is processed in a number of subsequent steps, and the output of one step serves as input for the next step. A popular way for modelling signal processing systems is the Kahn process network (KPN). The KPN models the system with a set of processes that are connected with FIFO buffers. Each process follows a sequential program and uses only the FIFOs for data exchange. The used FIFOs have unlimited size and blocking read. The blocking read stops the calling process in case there are no data available until new data arrive. The model has become popular because all systems modelled with KPN are determinate. Furthermore, synchronisation is done with a simple blocking read semantic.

When implementing a KPN with a multiprocessor architecture, it is reasonable to treat the processes as the smallest unit and to map one or more of them on the same processor. In most cases, this is the best solution because it minimises the data dependencies between the processors and leads to a smaller load on the interconnection network. However, the common multiprocessor architectures are not very good for the realisation of such systems. The best one of them is the MIMD-architecture (Figure1.1).[2] The following list outlines the problems of this architecture.

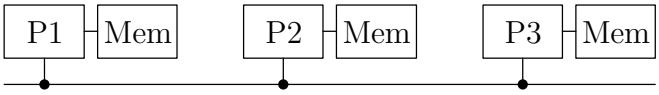



**Figure 1.1:** MIMD example architecture with processors P and instruction/data memories Mem.

- All communication modelled by the FIFO buffers has to share the same bus. This limits the data throughput and even if the interconnection network satisfies the data rates of the implementation, it limits the extendability of the system.
- A bus arbiter is required, which is equal to adding a global control structure to the system. This kind of global control is not included in the KPN model. It must therefore be shown that the bus arbiter does not violate the KPN rules and that the system delivers the expected results. For many situations, this is difficult and the automation of the design flow is very demanding.
- Synchronisation between the processes is based on low level mechanisms like interrupts and shared memory. This makes the behaviour of the system dependent on the used hardware and it is more difficult to migrate the design on a new processor type.

A solution for these problems is to use an architecture with a dedicated structure that is more similar to the KPN model. Figure 1.2 shows such an architecture. In this architecture,

[2] Multiple Instruction Multiple Data.

each processor has its own bus and a special hardware block is used to implement the FIFO channel. This channel implements the blocking read synchronisation in hardware.

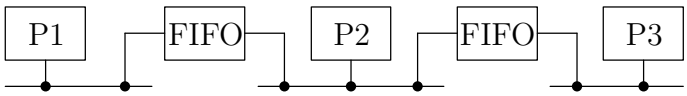



**Figure 1.2:** KPN based architecture with a dedicated structure.

Because the processor is the single master device, no bus arbitration or global scheduling is required. Each processor can use the full bus speed and this speed is not reduced when additional processing steps are added. The synchronisation is implemented in an isolated block and makes no use of low level mechanisms.

## 1.1 Problem Description: Limits of KPN

The KPN architecture offers distributed control and a simple interface for programming, but it also includes limitations that make the implementation of stream-based algorithms difficult or inefficient. The following three limitations that are of great importance for signal processing algorithms are addressed in this paper.

**Reordering:** The communication channel behaves in a strict first in first out manner, which does not allow to read data in another order as they were written.

**Multiple Read:** The communication channel does not allow to read the same data item more than once. After reading an item, it is deleted from the buffer and cannot be read a second time.

**Skipping:** It is not possible to remove an item from the channel buffer without reading it. If the buffer contains unwanted data, all of these data must be read.

The first two limitations can be handled with the FIFO architecture shown in Figure 1.2 if a local memory is used to implement the reordering and the multiple read. In most cases, the memory used to store the instructions of the processor can be used for this purpose. In such an implementation, the process has to implement the memory management for the reordering memory which is a task that was not modelled by the original process network. Furthermore it requires computing time. A known solution for this problem is to use a special address generation unit (AGU) ([11], [15]). Implementing skipping is not possible with an architecture using FIFOs. A new buffer type is required for this purpose.

## 1.2 Solution Approach: WFIFO Concept

The solution presented in this thesis defines a new type of process network, the WFIFO process network. It is based on the Kahn process network, but the FIFO buffers are replaced by a new buffer type called windowed FIFO (WFIFO). The WFIFO buffer is similar to the normal FIFO but it allows multiple read, reordering and skipping at the head and the tail of the buffer. These memory regions at the head and the tail are called windows. They have a user-defined size and are allocated and released at runtime by API instructions. The API follows a protocol with three subsequent steps: acquiring, data read/write and releasing. With acquiring, a new window is allocated, then its content can either be written or read. Before acquiring the next window, the previous must be released. Details of the WFIFO concept are presented in Chapter 2.

## 1.3 Research Contributions

- The WFIFO concept for the communication in on-chip multiprocessor systems is presented in detail. It is shown how data are transferred over a WFIFO buffer and the protocol used is explained.
- The WFIFO buffer is implemented in hardware for a Xilinx FPGA platform. The problems and requirements of the design are discussed and details of the implementation are shown.
- The API is defined in detail and it is realised for the WFIFO hardware implementation. Extra functionality is added to the API to simplify testing.
- Different strategies for IP and system testing are discussed and the solution implemented for the WFIFO design is presented. For testing purposes, two special IPs are designed. The implemented solution allows to perform both functional and performance tests.
- The testing procedure is automated with a SW tool. The program is controlled over a command line interface and can run multiple tests on the same test system. After test execution, a test report is generated.
- The design of WFIFO architecture is automated with a SW tool. The application takes an abstract specification of the system as input and generates the target architecture for the FPGA platform used.
- It is discussed whether a process network with WFIFO buffers is determinate or not. It is shown that with some restrictions, a WFIFO process network is a subset of Kahn process networks.
- The WFIFO concept is compared with two other concepts that are motivated by similar ideas like the WFIFO concept.
- Aspects concerning the optimal usage of the WFIFO concept are discussed.

## 1.4 Related Work

Discussions on the extension of the KPN model have a long tradition and many publications have been issued. Most of these publications can be assigned to one of the following two approaches. The first includes design-flow-centric approaches and the second includes interface-centric approaches. The WFIFO model was designed with both of theses aspects in mind. It includes the definition of a new interface and the automation of the hardware design.

The interface-centric approaches focus on the improvement of the interface that is used by the processes to communicate. An interface that is related to the interface used by the WFIFO was presented in [3]. In contrast to the WFIFO, this interface allows multiple acquiring before releasing and the transfer of data blocks. The proposed system has a MIMD architecture with distributed memory which is very different to the WFIFO architecture. Another interface-centric paper is [13]. It presents a set of seven different interface definitions. The interfaces named CB and RB (Combined and Relative Blocking) are the ones that are most similar to the WFIFO approach. In contrast to the WFIFO interface, CB and RB both allow multiple acquiring before releasing. This paper is based on ideas from [1], which also discusses the design-flow aspects.

The design-flow-centric approaches focus on the automation of the design flow. One of these approaches it the Compaan/Laura tool-chain [8]. This tool-chain takes Matlab code with a nested loop structure as input and converts it into a KPN model. In a second step, dedicated hardware is generated and the KPN is mapped on this architecture. A set of solutions to overcome the multiple-read and the reordering limitations were presented in [11] and [15]. Another approach was followed with the SHIM model [2]. SHIM takes a C-like description as input where both hardware and software is described and shared variables are used to model the communication. From this description, a dedicated hardware and the required software is generated. A buffer using a rendezvous protocol is used to implement the communication channel.

## 1.5 Content of this Report

### Introduction

**Chapter 2:** Introduces the concept of the WFIFO buffer, gives a first impression of the WFIFO API and shows how an algorithm can be modelled with a WFIFO process network. It also shows the structure of the target architecture and a simple example is made to give a first example of application.

## Practical Work

The practical work is described in four chapters which can not be completely separated. Figure 1.3 gives an overview of the dependencies. This figure is repeated at the beginning of each of the four chapters and the dependencies of the respective part are discussed. Each chapter starts with the general aspects and goes more and more into the details.

**Chapter 3:** Presents the hardware implementation of the WFIFO concept. The requirements and problems for the hardware implementation are listed and details of the implementation are presented. Furthermore, the FPGA platform used is indicated and the selection of the processor and bus types is explained.

**Chapter 4:** Defines the API in detail and shows how it was implemented for the designed WFIFO buffer. A concept to simplify testing is presented.

**Chapter 5:** Discusses the problem of hardware testing by explaining the solution implemented for the WFIFO project. It also shows how testing was automated with the design of a modular testing environment. The presented testing concept addresses the IP verification and the performance measurement of complete systems.

**Chapter 6:** Presents the automation of the hardware design. A software tool was developed that can generate a complete and synthesizeable implementation of a WFIFO process network from an abstract specification.

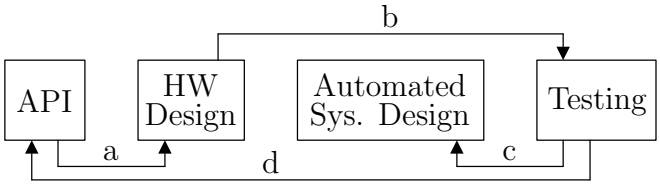



**Figure 1.3:** Chapter dependencies of the practical work.

## Theoretical Work

**Chapter 7:** Discusses whether a process network with WFIFO buffers is determinate or not. A concept for describing the data transport in communication channels is presented. This concept is used to calculate minimum memory requirements for WFIFO channels and for channels with reordering memory.

**Chapter 8:** Compares the WFIFO concept with two other approaches for implementing communication with multiple read and reordering in process networks.

**Chapter 9:** Contains the conclusion and the outlook. The conclusion refers to the beginning of my work, gives an impression of its scope and lists the completed tasks. The outlook shows how the WFIFO concept could be further developed.

# 2 WFIFO: The Concept

## 2.1 WFIFO – Windowed FIFO

The WFIFO buffer is related to the normal FIFO but it offers more functionality and flexibility. The normal FIFO behaves in a strict first-in first-out manner which does not allow to read data in another order than they were written to the FIFO. Furthermore, each data item written to the buffer must be read exactly once. It is not possible to read the same item more than once or to remove an item from the buffer without reading it. Unlike the FIFO buffer, the WFIFO buffer allows to skip unwanted data. It also supports data reordering and multiple read within a continuous data segment located at the FIFO's head or tail. These segments are called windows, which has lead to the name WFIFO for this buffer structure. Figure 2.1 shows the simplified schematic representation of a FIFO and a WFIFO buffer. It shows that the WFIFO has a more complex interface. Read and write windows are indicated in gray.

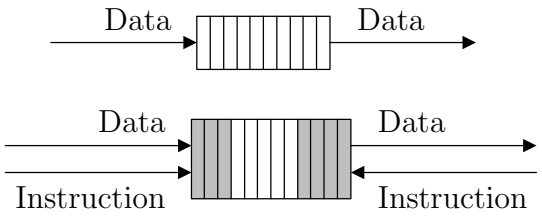



**Figure 2.1:** Simplified schematic of FIFO and WFIFO with write port on the left and read port on the right side.

While the FIFO buffer allows a single instruction at both ports, i. e. data-write at the input port and data-read at the output port, the WFIFO supports two additional instructions at each port. Table 2.1 lists these instructions. They can not be used an in arbitrary order. The next section explains the required protocol for the instructions and what they are used for. A detailed description of the API is given in Chapter 4.

| Write Port Instructions | Read Port Instructions |
|---|---|
| • Acquire-read-window(port, size) | • Acquire-write-window(port, size) |
| • Data-read(port, offset, data) | • Data-write(port, offset, data) |
| • Release-read-window(port) | • Release-write-window(port) |

**Table 2.1:** WFIFO instructions for read and write port.

### 2.1.1 WFIFO Protocol

Writing to and reading from a WFIFO are two independent transactions, which both follow a protocol. The protocol is defined by three subsequent steps and four WFIFO-internal states (Figure 2.2). The enumeration below explains the steps for the write port of the WFIFO. It is assumed that the write port is currently in the idle state and the process connected to the write port wants to write some data to the buffer.

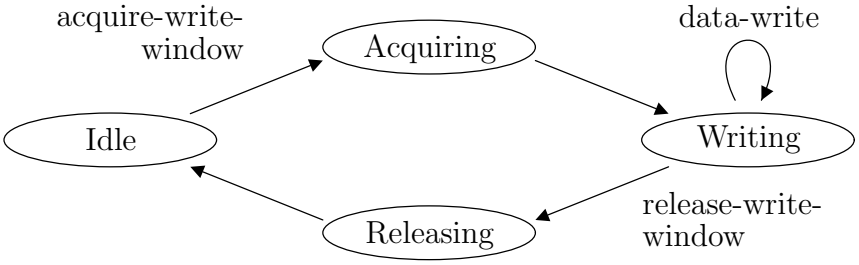



**Figure 2.2:** States of the WFIFO write port.

1. Before writing any data to the WFIFO buffer, a write window must be acquired, which is done by passing an acquire-write-window instruction to the WFIFO. This instruction takes the window size as argument. When receiving the instruction, the WFIFO port changes into the acquiring state. If there is enough free memory[1] in the WFIFO, it acquires the window and changes into the write state. From then on data can be written to the WFIFO (Step 2) or the window can be released (Step 3). If there is not enough memory, the WFIFO's behaviour depends on the instruction type used. There are two versions of the acquire instruction: a blocking and a non-blocking one. When using the blocking version, the WFIFO waits until enough memory becomes available and the calling process is blocked during this time. As soon as enough memory is available, the WFIFO changes into the write-state and wakes up the blocked process. If there is not enough memory for the non-blocking version, the WFIFO returns into the idle state and signals the calling process that acquiring failed.

2. As soon as the WFIFO reaches the writing-state, data can be written to the window. The write instruction takes the data to write and an offset position as arguments. The offset indicates to which position the data is written within the write window. The instruction can be repeated an unlimited number of times as long as the WFIFO is in the writing-state. It is possible to write the same offset position more than once, which results in overwriting the old value. If an offset position is never written, its value is undefined. The writing-state is left when executing the release instruction (Step 3).

3. The release instruction is executed to terminate writing, and it can only be executed if the WFIFO is in writing-state. When receiving the release instruction, the WFIFO

[1] The terms free-memory and internal FIFO used in this description are explained in 2.1.2.

changes into the releasing-state and closes the write window. After that, no further writing is possible until a new write window is acquired. After releasing is done, the WFIFO changes into the idle state.

The protocol for the read port is very similar to the protocol of the write port. It is also defined by three steps and four WFIFO-internal states (Figure 2.3). The enumeration below is focused on the difference between the read and write protocols.

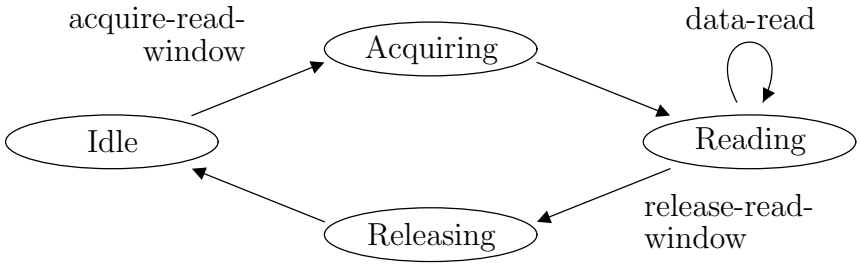



**Figure 2.3:** States of the WFIFO read port.

1. The read port also offers two acquiring functions – a blocking and a non-blocking one. Acquiring can be executed successfully if there are enough data available in the internal FIFO buffer.

2. Step two is the same as for the write port except that data are read from the window instead of written.

3. Releasing the read window is equal to deleting all of its content from the buffer. No further read operations on the data within the window are possible. The memory that was occupied by the read window is added to the free memory.

### 2.1.2 WFIFO Data Transport

From the perspective of data transport, the WFIFO buffer can be understood as a normal FIFO channel with a reordering memory implemented as read and write windows at its head and its tail. Such a model is shown in Figure 2.4. The logic block is used to implement the WFIFO protocol. This model does not show the internal structure of the WFIFO. It is only used to describe the data transport between the WFIFO write and read port. Differences between the model and the WFIFO implementation are discussed at the end of this section.

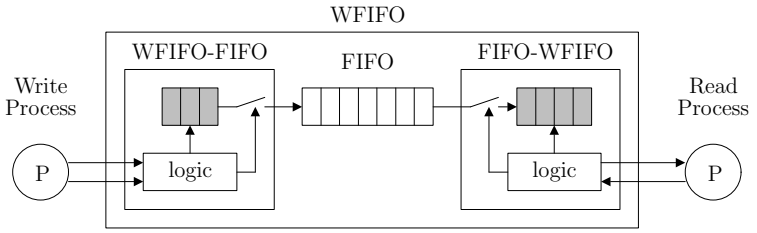



**Figure 2.4:** Models used to explain the data transport in WFIFO buffers. The reordering memories are indicated in gray.

When the logic at the write port receives an acquire-write instruction, it enables writing to the reordering memory. This memory has a continuous address range starting with zero and a size as big as the specified window. Subsequent write instructions are redirected to this memory at the position indicated by the offset. When receiving a release instruction, writing is disabled and the content of the memory segment is shifted to the FIFO without changing its order.

The logic at the read port starts reading data from the FIFO when receiving an acquire-read instruction. It reads as many items as specified by the window size and writes them to the reordering memory with the same addressing concept as the write window. Also here, the order of the data is not changed and the last item read has the memory address zero. After reading from the FIFO is finished, read operations are redirected to the memory position indicated by the offset. When receiving a release instruction, reading is disabled.

As indicated above, the realisation with FIFO buffer and reordering memory only models the WFIFO's functionality. This model is useful to explain the data transport but it does not describe all aspects of the WFIFO's behaviour correctly. In the WFIFO hardware implementation, a single memory block is used instead of the combination of FIFO and reordering memory. This memory block is segmented into four regions: free-memory, read-window, FIFO-buffer and write-window.[2] Because the memory block has a finite size, there is a maximum amount of memory that can be occupied by the windows and the FIFO-buffer. If a process tries to acquire a new write window when all memory is used, it is blocked. This kind of blocking is not modelled correctly by the WFIFO model from above because it never blocks on acquiring. But the model can block in another situation: if the FIFO buffer has limited size, the model blocks during the release instruction if there is not enough free memory left to write all data. The implementation with the single memory never blocks during release.

[2] A detailed description of the required memory management is given in Chapter 3.

## 2.2 System Modelling with WFIFO Process Networks

Implementing a signal processing system starts with modelling the system's behaviour as a WFIFO process network. This is similar to system modelling with Kahn process networks but windowed FIFOs are used instead of normal FIFOs. A system model includes the source code for the processes and the description of the topology of the underlying process network. For all communication between the processes, WFIFO buffers are used. They are accessed with the WFIFO API. A source code example is given in 2.4. Figure 2.5 shows a topology of a WFIFO process network with three processes.

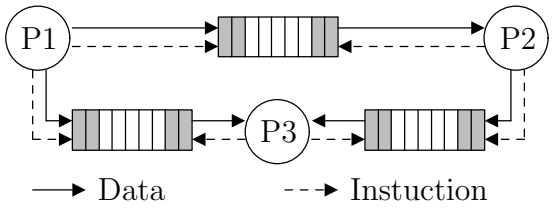



**Figure 2.5:** Example WFIFO process network with P1 as data source and P3 as data sink.

In order to benefit from the advantages of the WFIFO concept it is important to use the additional options in a reasonable way. With the read and write windows, a segmentation of the data is introduced that was not necessary for the FIFO buffers in KPN networks. For making a good WFIFO design, the definition of the data segmentation is essential. There are situations where a segmentation is contained in the signal processing algorithm. In other situations, the segmentation must be defined by the designer. In Chapter 7 mathematical concepts are presented for data segmentation.

The advantages of a windowed FIFO can be summarised under multiplicity, reordering and skipping. Multiplicity means that the same data item can be accessed more than once. On the read port, the same item can be read an unlimited number of times as long as it is in the read window. On the write port it is possible to update values that were already written to the buffer until the write window is released. Reordering allows data items to be consumed in an order other than produced. Both, the producer and the consumer can reorder the data within the acquired windows. If the consumer does not read all data within the read window this can be seen as skipping of unwanted data. Skipping makes sense if it is not possible for the producer to know if data will be required by the consumer or not.

## 2.3 WFIFO Target Architecture

The target system has a dedicated architecture and is assembled from two building blocks. One includes the WFIFO buffer, the other a processor with a bus and a memory for instructions and internal data (Figure 2.6).


**Figure 2.6:** Building blocks for the WFIFO architecture. Mem is the instruction and local data memory of processor P1.

To build the target architecture, the topology of the WFIFO process network and a mapping definition is used. The mapping defines which processes have to share a processor. No mapping is required for the WFIFO buffers because each buffer is mapped on one WFIFO hardware block. The connections of the WFIFOs are well-defined by the topology and the mapping defined for the processes. Figure 2.7 shows a WFIFO architecture of the process network shown in Figure 2.5 for a one-to-one mapping.

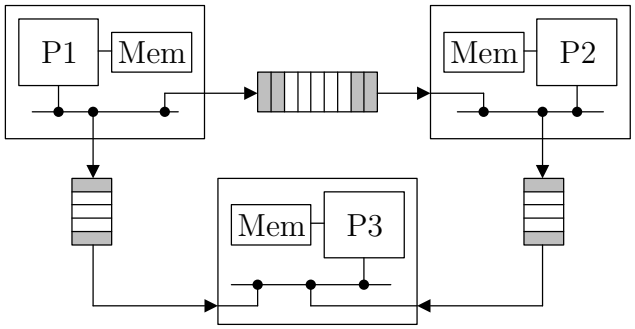



**Figure 2.7:** WFIFO example architecture.

Since the WFIFO architecture is a dedicated solution, it is important to automate the hardware design flow. In contrast to heterogeneous architectures, it is simple to implement such an automation for a WFIFO architecture. Advantages are:

- Only two building blocks are used.
- No bus arbitration is required because each bus has a single master device, the processor, and a set of equal slave devices, the WFIFOs.
- Bus address range segmentation is simple because the WFIFOs require a range of equal and fixed size.
- No access arbitration for a global memory is needed.
- No global scheduler with access to all processors is needed.

## 2.4 Example of Application

In this section a first simple example of application is shown. Details of the API are given in Chapter 4. The example below does not use all the options offered by the WFIFO. It is a first and simple code example. The algorithm is given as a C-code sequence (Figure 2.9). It contains a producer and a consumer section. In the producer section, the two dimensional array $A$ is written. In the consumer section it is read. The communication channel used when mapping the producer and the consumer section on two different processors must support multiple read and reordering. The code for the process network is given in Figure 2.10.


**Figure 2.8:** Topology of WFIFO process network for a consumer producer pair.

```
// P1: producer
for (i=0; i<4; i++) {
  for (j=0; j<3; j++) {
    write(A[i][j]);
  }
}
```

```
// P2: consumer
for (i=0; i<3; i+=2) {
  for (j=0; j<2; j++) {
    read(A[i][j]);
    read(A[i+1][j]);
    read(A[i][j+1]);
    read(A[i+1][j+1]);
  }
}
```

**Figure 2.9:** Source code of the algorithm.

```
// P1: producer
void main() {
  WFIFO_ACQUIRE_WRITE(0,6);
  WFIFO_WRITE(0,0,A[0][0]);
  WFIFO_WRITE(0,1,A[0][1]);
  WFIFO_WRITE(0,2,A[0][2]);
  WFIFO_WRITE(0,3,A[1][0]);
  WFIFO_WRITE(0,4,A[1][1]);
  WFIFO_WRITE(0,5,A[1][2]);
  WFIFO_RELEASE_WRITE(0);

  WFIFO_ACQUIRE_WRITE(0,6);
  WFIFO_WRITE(0,0,A[2][0]);
  WFIFO_WRITE(0,1,A[2][1]);
  WFIFO_WRITE(0,2,A[2][2]);
  WFIFO_WRITE(0,3,A[3][0]);
  WFIFO_WRITE(0,4,A[3][1]);
  WFIFO_WRITE(0,5,A[3][2]);
  WFIFO_RELEASE_WRITE(0);
}
```

```
// P:2 consumer
void main() {
  WFIFO_ACQUIRE_READ(0,6);
  WFIFO_READ(0,0,A[0][0]);
  WFIFO_READ(0,3,A[1][0]);
  WFIFO_READ(0,1,A[0][1]);
  WFIFO_READ(0,4,A[1][1]);
  WFIFO_READ(0,1,A[0][1]);
  WFIFO_READ(0,4,A[1][1]);
  WFIFO_READ(0,2,A[0][2]);
  WFIFO_READ(0,5,A[1][2]);
  WFIFO_RELEASE_READ(0);

  WFIFO_ACQUIRE_READ(0,6);
  WFIFO_READ(0,0,A[2][0]);
  WFIFO_READ(0,3,A[3][0]);
  WFIFO_READ(0,1,A[2][1]);
  WFIFO_READ(0,4,A[3][1]);
  WFIFO_READ(0,1,A[2][1]);
  WFIFO_READ(0,4,A[3][1]);
  WFIFO_READ(0,2,A[2][2]);
  WFIFO_READ(0,5,A[3][2]);
  WFIFO_RELEASE_READ(0);
}
```

**Figure 2.10:** Source code for the WFIFO process network.

# 3 The Hardware Design

This chapter shows details of the WFIFO hardware implementation. The implementation includes the design of new IPs, namely the WFIFO IP and the selection of the target platform and the IP cores. All components together make it possible to realise a complete WFIFO architecture.

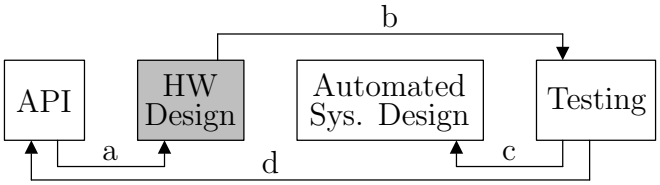



**Figure 3.1:** Overview of practical work.

Figure 3.1 shows the dependencies of the hardware design. The WFIFO hardware is used by the API and makes use of the testing infrastructure. Because the testing itself uses the API there is a closed loop including hardware design, testing and API. This means that the hardware design must be done in a way that allows to make tests. This fact is well known as DFT – design for testability. The interface between API and hardware is defined by the instruction format (See 3.5.1) and the bus driver. The interface offered by the testing environment is presented in Chapter 5.

## 3.1 Target Platform and Core Selection

### 3.1.1 FPGA Platform

The hardware implementation of the WFIFO concept was designed for an FPGA platform. It is also possible to use the WFIFO concept in an ASIC, but the fact that the WFIFO architecture depends on the implemented algorithm makes an FPGA implementation more reasonable. An other reason for selecting an FPGA is the available software. Most FPGA providers offer a good software bundle that also includes an IP library for very low costs. A disadvantage of using FPGAs is that the hardware design becomes dependant on the target platform. For example, when designing a new IP, it is not enough to write the VHDL code. The software used for system design and FPGA mapping needs additional data about the new IP. The format and the content of these data depends on the software used. Another point is that certain components from the IP library like the processors are only available in a precompiled version that can not be used for other FPGAs. All of this makes it difficult to migrate the design to a platform of a different provider.

The requirements when selecting the FPGA for the WFIFO design were that an IP library must be available that includes processor cores and that it is possible to map a couple of them on the FPGA. ML300 Evaluation Platform from Xilinx with a Virtex-II chip was selected. ISE and EDK version 7.1 were used as design software. The most important software was the EDK which includes a big library of IPs and the application XPS Studio. With XPS Studio all steps of the design flow can be done. First, the system can be assembled and configured using the IPs from the EDK library. Then, application code for the processors can be added and compiled. And finally, XPS can generate a simulation model for Modelsim or it can generate the bitmap used for programming the FPGA. A total of four IPs were designed for this project. All of them are compliant with the standard of the XPS IP library. This makes it possible to assemble and simulate a complete WFIFO architecture with XPS Studio.

### 3.1.2 Processor and Bus Selection

The first step in hardware design was to select those processors and buses from the IP library that are best for the WFIFO architecture. The library includes two processor types, the Micro Blaze and the Power PC. Table 3.1 shows the buses of these processors that can be used for attaching a WFIFO.

| Processor Type | Bus Name | Address Width | Data Width | Throughput |
|---|---|---|---|---|
| PPC | PLB | 32-bit | 64 or 32-bit | 1600MB/s |
| MB | LMB | 32-bit | 32-bit | 500MB/s |
| MB, PPC | OPB | 32-bit | 32-bit | 500MB/s |

**Table 3.1:** Interfaces of PPC and MB

The only differences of interest between MB and PPC are the computing power and the maximum data rates. Thats why it is not possible to make a decision between them without knowing the requirements of the implemented algorithm. But for the target platform used, the Micro Blaze is the only feasible solution because the ML300 board can only hold a single PPC. Since the WFIFO architecture uses several processors in most cases, PPC can not be used. However, it would be nice if the designed WFIFO IP could also be used together with Power PC processors as soon as a bigger FPGA is available.

The problem when designing an IP for both MB and PPC is that the only bus type supported by both processors is the OPB bus. This is a good solution for MB processors but with PPC processors it is not possible to benefit from the higher data transferrate that is supported. One could design the WFIFO IP for the PLB bus and use an OPB to PLB bridge to connect with the MB. This approach has the disadvantage that an extra bus and bridge is required for each MB processor which increases chip size. Such a bridge does also increase the latency and reduces the system's performance. This is why another concept was preferred. The implementation made uses an extra IP called IPIF to connect the WFIFO to the bus (See Figure 3.2). The IPIF offers a standardised IP interface at one side and a specific bus

interface at the other side. With such a design it is possible to connect the same IP to different bus types by using different IPIF implementations. Since the interface for the IP is the same for all bus types, it is not necessary to change it. The EDK library offers IPIF implementations for OPB and PLB bus. This was the reason to use the OPB bus instead of the LMB for the Micro Blaze. The LMB bus is used to connect the local instruction and data memory of the processor.


**Figure 3.2:** WFIFO implementation with IPIF bus adaptors.

## 3.2 WFIFO Overview

Figure 3.3 gives an overview of the WFIFO architecture. The IP consists of two IPIFs, a dual port BRAM and the state machine. The IPIF first was taken from the EDK library but a reimplementation was required because the original included too many design errors (See 3.8). The memory is built from a set of BRAMs. To simplify the usage, different memory configurations have been integrated in a BRAM entity. Section 3.4 gives a short description of the BRAM entity. The interface to the state machine and the access rules are described in 3.7.2. The state machine is the heart of the WFIFO implementation. Its description is divided up in several sections. First, the interface and the instruction format used for the bus protocol are defined in 3.5.1. Details of the state machine are presented in 3.3 and the structure of the VHDL implementation is discussed in 3.7.6.

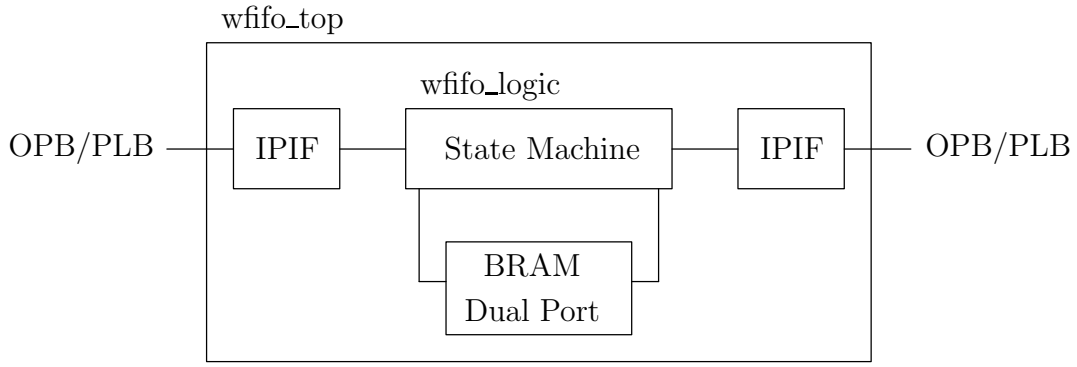



**Figure 3.3:** WFIFO architecture overview

### 3.2.1 Requirements and Problem Description

This section list the requirements and the most important points to discuss for the WFIFO implementation.

**Simultaneous R/W:** Die WFIFO IP is connected to two different buses. From one bus it receives the read instructions and from the other one the write instructions. The IP must be designed to support simultaneous read and write. At the same time it must guarantee that no conflicts in data access occur. This problem was solved with an adequate state machine architecture and memory management.

**Bus interface definition:** In the WFIFO concept, only the API is defined. For the IP, design a bus protocol and an instruction format must be defined that allow an efficient API implementation.

**Minimum latency:** The WFIFO IP must be designed for minimum latency because this has major impact on the performance of the entire system.

**Protocol violations:** In the WFIFO protocol it was not defined how the buffer reacts if the protocol is violated. Such violations can be treated by the API software or by the hardware. If implemented in hardware an error signalling concept must be defined.

**Easy to use:** The WFIFO IP should be easy to use but flexible in application. This was reached by integrating the entire WFIFO including the IPIF adaptors and the memory in a single IP. This IP can be configured with design parameters (See 3.6).

**DFT:** The WFIFO IP must offer an interface that allows the testing of its functionality in an efficient way. This was considered in the error signalling concept and a software reset was added to make testing more efficient.

## 3.3 WFIFO State Machine

### 3.3.1 Single and Dual Bus Application

The WFIFO has two different applications. Either it is used as a dual bus IP or as a single bus IP. Single bus means that both ports of the WFIFO are connected to the same bus. In

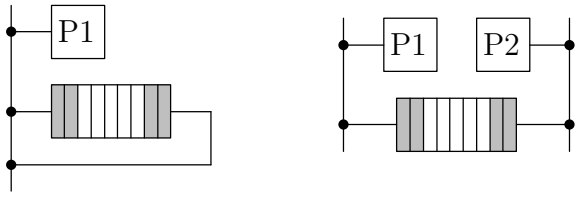



**Figure 3.4:** Single and dual bus WFIFO.

the dual bus case it is connected to two different buses (see Figure 3.4). This difference also has an impact on the design of the state machine. In the dual bus version it is possible that two requests arrive at the same time. In this case the state machine has to react to two commands at the same time. Furthermore it must be ensured that the two commands do not affect each other in a way not permitted. In the single bus version it depends on the bus type if two requests may arrive at the same time. The PLB bus allows simultaneous read and write, whereas the OPB does not. To keep things simple, an IP that is suitable for both cases was implemented. This results in an IP that is not optimal for the single bus application on the OPB bus but it can be used for all of the proposed applications. As will be seen later, this is not a disadvantage in terms of timing and throughput.

An implication of this architecture is that in the single bus application read and write access must be sent to different ports (different address ranges). The required address range of the single bus version is twice as large as for an IP optimised for the OPB bus. This is acceptable since the single bus application is assumed to be less important than the dual bus application.

### 3.3.2 WFIFO State Machine Architecture

For the architecture of the state machine, an implementation with two synchronised state machines is used (see Figure 3.5). One state machine is connected to the write port and handles all write commands, the other one does the same for the read port. Synchronisation is done over a set of shared states. From the functional point of view, an architecture with two synchronised state machines is not better than an architecture with a single state machine. But the version with two state machines is easier to specify and to implement. The reason for this is that the implementation with the single state machine not only has to encode all possible read and write states but also all possible combinations of them. Therefore it can even be assumed that the implementation with two state machines is more efficient in terms of chip size, because less bits are used to encode the states.

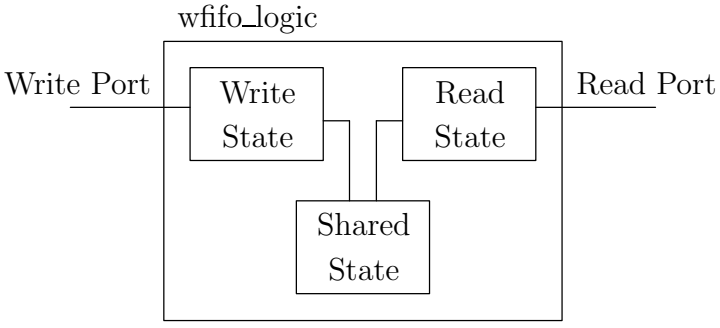



**Figure 3.5:** WFIFO state machine architecture

### 3.3.3 Protocol Violations

The WFIFO concept only defines the WFIFOs behaviour in case the used instructions follow the protocol correctly. Examples of protocol violations are writing before acquiring or reading with an offset bigger than the acquired window. In such situations it must be ensured that the buffer is not set into an invalid or unknown state. There are two possibilities to address this problem. The first is to make sure that the WFIFO hardware never receives instructions violating the protocol. This can be done with a checking mechanism in the API. The second one is to design the WFIFO IP in a robust way that checks for protocol violations and rejects incorrect instructions. For the WFIFO design, the second solution was selected because it reduces the latency. Checking within the API does always need additional clock cycles, whereas checking can be implemented more efficiently in hardware. On the other hand, the hardware solution must pay attention to the design for testability.

The testing infrastructure used for the WFIFO design can access the WFIFO hardware only via the API. To test whether the WFIFO rejects incorrect instructions, rejection must be signalled to the API. To submit this status information, it is possible to use the error signal line of the bus or an extra bus transaction. Although the solution with the error signal seems to be faster at first sight, the solution with the extra bus transaction was chosen. The problems for an implementation using the error signal line are presented in 4.3.

The concept selected to detect protocol violations has impact on the state machine design. First, the state machine must check all incoming requests whether they are compliant with the protocol, and second it must support an additional state to read the status information.

### 3.3.4 WFIFO States

Figure 3.6 shows the timing diagram for executing a WFIFO instruction. The incoming instruction is checked for correctness in the same clock cycle it arrives and is executed in the next clock period. With this solution, the buffer needs only two clock cycles to execute an instruction. Exceptions are the data read instruction that takes two clock cycles and the blocking acquiring in case that the WFIFO has to wait for data or free memory. For the total execution time, the time for bus transfer and pipelining in IPIF must be added. Table 4.5 gives an overview of the total execution times.

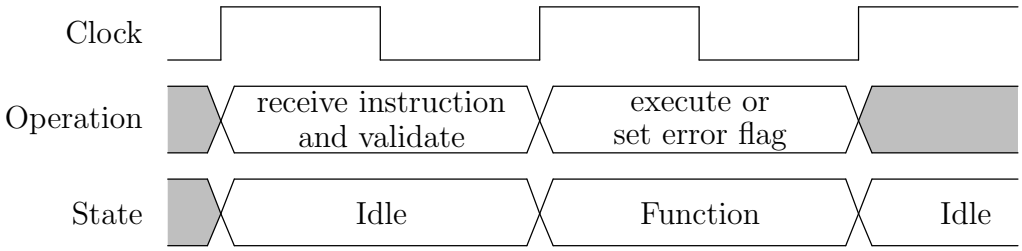



**Figure 3.6:** Timing diagram for WFIFO instructions. For the read data operation the function state takes two clock cycles.

This section presents the states of the read and write state machine and a simplified state transition function. A more detailed but also more abstract description of the state transitions is given in 3.7.4.

### WFIFO Write States and Transitions

| Write State | Shortcut | Function |
|---|---|---|
| Idle State | WrIdle | WFIFO is waiting for the next write function call. |
| Acquire Write Ack | AcWrAck | WFIFO executes an acquire write operation and signals acknowledge to the calling process. The write function status is set to OK. |
| Acquire Write Blocked | AcWrBlk | WFIFO is waiting until free memory is big enough for an acquire write operation. |
| Acquire Write Failed | AcWrFail | WFIFO is not able to execute a non-blocking acquiring operation because there is not enough free memory. Acknowledge is signalled to the calling process and the write function status is set to FAILED. |
| Release Write Ack | ReWrAck | WFIFO executes a release write operation and signals acknowledge to the calling process. The write function status is set to OK. |
| Write Data Acl | WrDaAck | WFIFO writes data to its memory and signals acknowledge to the calling process. The write function status is set to OK. |
| Error State | WrErr | The last instruction received was incorrect. The instruction is not executed and the write function status is set to ERROR. |
| Write Status Ack | WrStaAck | The current write function status can be read from the data bus and acknowledge is signalled to the calling process. |

**Table 3.2:** WFIFO write states. The write function status indicates the success of the last instruction. It can have the values OK, ERROR and FAILED.

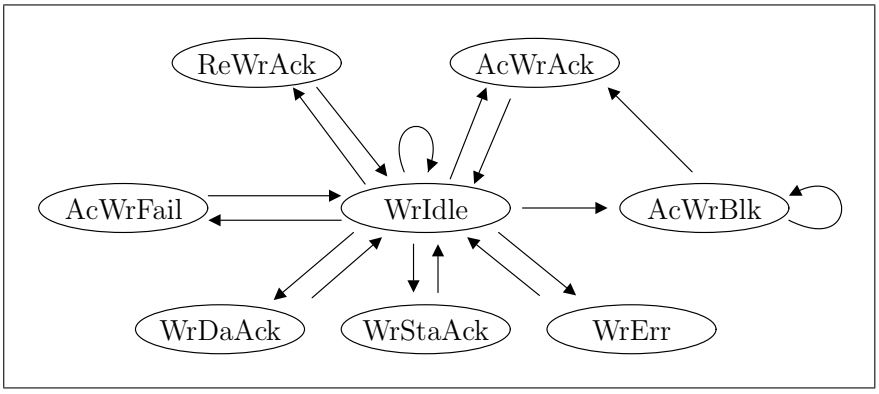



**Figure 3.7:** WFIFO state transition of the write state machine. The figure shows all possible transitions. The conditions for the transitions are listed in Table 3.3.

| Current State | Next State | Condition |
|---|---|---|
| WrIdle | WrIdle | No WFIFO write instruction received. |
| | WrErr | The received instruction violates the WFIFO protocol. |
| | AcWrAck | The WFIFO receives a correct acquire write instruction and the free memory is big enough for the acquiring. |
| | AcWrBlk | The WFIFO receives a correct and blocking acquire write instruction but the free memory is not big enough for the acquiring. |
| | AcWrFail | The WFIFO receives a correct and non-blocking acquire write instruction but the free memory is not big enough for the acquiring. |
| | ReWrAck | The WFIFO receives a correct release write window instruction. |
| | WrDaAck | The WFIFO receives a correct data write instruction. |
| | WrStaAck | The WFIFO receives a correct status signal report instruction. |
| WrErr<br>AcWrAck<br>AcWrFail<br>ReWrAck<br>WrDaAck<br>WrStaAck | WrIdle | Transition occurs in any case. |
| AcWrBlk | AcWrBlk | There is still not enough free memory for the acquire operation. |
| | AcWrAck | Now there is enough free memory. |

**Table 3.3:** WFIFO write state transitions.

### WFIFO Read States and Transitions

| Read State | Shortcut | Function |
|---|---|---|
| Idle State | RdIdle | WFIFO is waiting for the next read function call. |
| Acquire Read Ack | AcRdAck | WFIFO executes an acquire read operation and signals acknowledge to the calling process. The read function status is set to OK. |
| Acquire Read Blocked | AcRdBlk | WFIFO is waiting until enough data is available from the internal FIFO buffer. |
| Acquire Read Failed | AcRdFail | WFIFO is not able to execute a non-blocking acquiring operation because there are not enough data in the internal memory. Acknowledge is signalled to the calling process and the read function status is set to FAILED. |
| Release Read Ack | ReRdAck | WFIFO executes a release read operation and signals acknowledge to the calling process. The read function status is set to OK. |
| Read Data OK | RdDaOk | To execute the read instruction, the WFIFO needs two states. In the OK state it provides the memory with the read address. |
| Read Data Ack | RdDaAck | The data can now be read from the data bus. The WFIFO signals acknowledge to the calling process and the read function status is set to OK. |
| Error State | RdErr | The last instruction received was incorrect. The instruction is not executed and the write function status is set to ERROR. |
| Read Status Acknowledge | WrStaAck | The current read function status can be read from the data bus and acknowledge is signalled to the calling process. |

**Table 3.4:** WFIFO read state

.

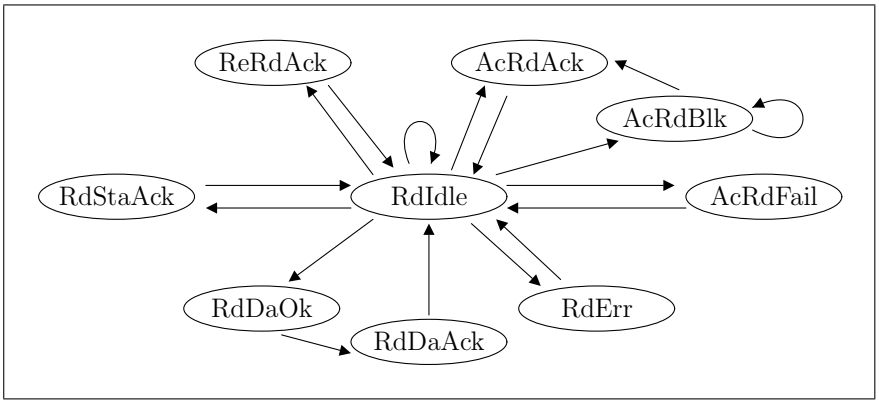



**Figure 3.8:** WFIFO state transition of the read state machine.

| Current State | Next State | Condition |
|---|---|---|
| RdIdle | RdIdle | No WFIFO read instruction received. |
| | RdErr | The received instruction violates the WFIFO protocol. |
| | AcRdAck | The WFIFO receives a correct acquire read instruction and there are enough data in the FIFO buffer for acquiring. |
| | AcRdBlk | The WFIFO receives a correct and blocking acquire read instruction but there are not enough data in the FIFO buffer for acquiring. |
| | AcRdFail | The WFIFO receives a correct and non-blocking acquire read instruction but there are not enough data in the FIFO buffer for acquiring. |
| | ReRdAck | The WFIFO receives a correct release read window instruction. |
| | RdDaOk | The WFIFO receives a correct data read instruction. |
| | WrStaAck | The WFIFO receives a correct status signal report instruction. |
| RdErr<br>AcRdAck<br>AcRdFail<br>ReRdAck<br>RdDaAck<br>WrStaAck | RdIdle | Transition occurs in any case. |
| RdDaOk | RdDaAck | Transition occurs in any case. |
| AcRdBlk | AcRdBlk | There are still not enough data in the FIFO buffer for acquiring. |
| | AcRdAck | Now there are enough data in the FIFO buffer. |

**Table 3.5:** WFIFO read state transitions.

### 3.3.5 Memory Management

Since the WFIFO is implemented with a single memory block, a memory management is required. In the BRAM memory, there are four distinguishable regions: read window, write window, FIFO data and free memory. The size of these regions changes dynamically and every memory cell belongs to one of these regions. There are different options to map the regions to the BRAM. In this implementation a circular concept is used. Figure 3.9 shows how the size and the position of the four regions change while executing an example code sequence.

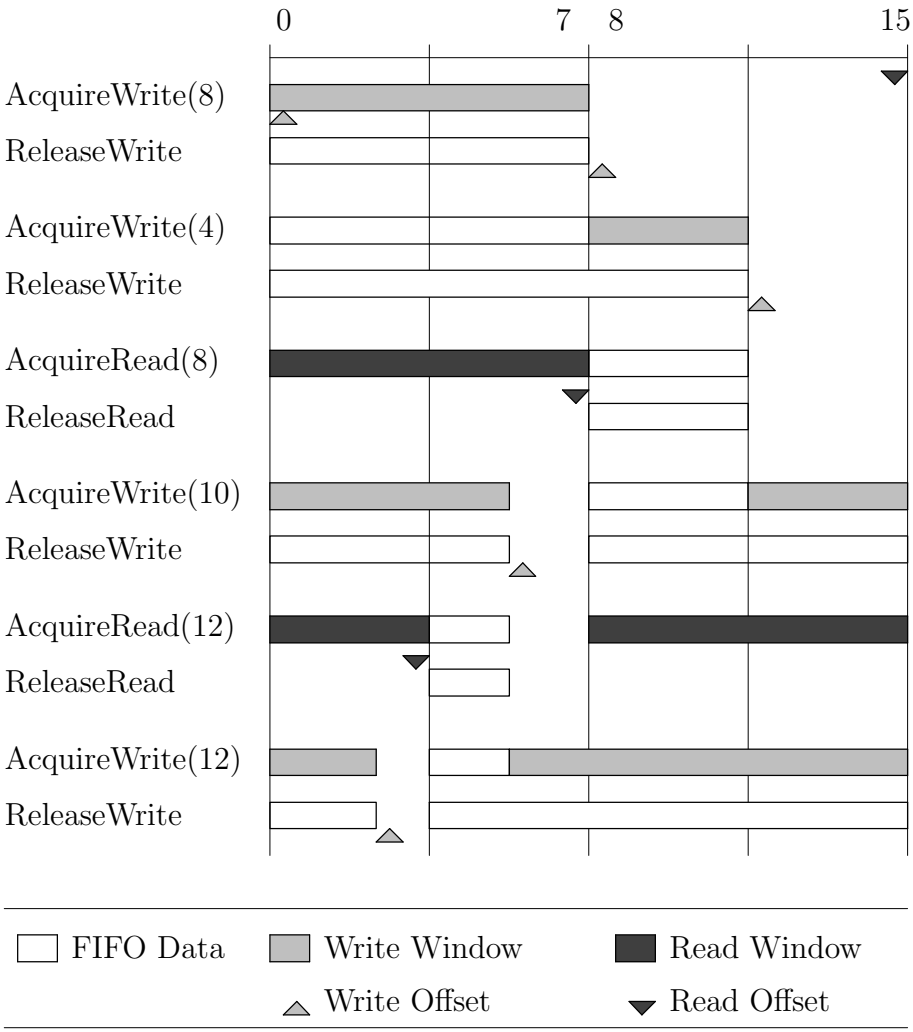



**Figure 3.9:** Memory management example for a memory of size 16. Read and write offsets are only shown if they change. The number in brackets indicates the window size.

### Address Translation

The address is calculated by adding the address passed by the instruction and the offset value. If the resulting value extends the address range defined by the BRAM size, the address is translated to the beginning of the BRAM memory by a modulo operator.

$$\text{BramWrAddrxD} = \text{mod}(\text{BramWrOffsetxD} + \text{WrAddrxDI}, \text{BramSize})$$
$$\text{BramRdAddrxD} = \text{mod}(\text{BramRdOffsetxD} + \text{RdAddrxDI} + 1, \text{BramSize})$$

### Offset Updating

The read and write offset are updated whenever a read or write window is released. When a write window is released, the occupied memory region is added to the FIFO data. When a read window is released, the memory region is added to the free memory. During read and write operations, the offset values are used to calculate the target address. During the rest of the time, the input address is set to zero and the offset serves as a dummy read address for the BRAM. The read and write BRAM address must never have the same value. This is ensured by setting the read offset one position before the next read window. In case there are no read or write windows but a maximum of FIFO data, two memory locations must be reserved for the offset pointers. Therefore, the maximum amount of data that can be written to the buffer is the BRAM size minus two. It is also possible to implement a memory management that uses only one empty address, but more complex state machines are required.

## 3.4 WFIFO Memory

The WFIFO memory is implemented with BRAM blocks. A BRAM is a parameterisable memory module that is available on all newer Xilinx FPGAs. The BRAM has two independent access ports. Simultaneous reading or writing on both ports is possible. The only limitation is that simultaneous access must address two different memory locations. Usually one BRAM memory is built up from a set of smaller BRAM blocks. The number of these sub-blocks and how they are connected depends on the required memory size. To hide this from the state machine, the different memory configurations are placed in a WFIFO BRAM entity that offers a memory-size-independent interface for the state machine. The IO signals of the BRAM entity and access rules are described in 3.7.2.

The total memory size is a parameter that is set during architecture design. Depending on the required data scheduling, there is a lower bound for this design parameter and it is the designer's responsibility to select this parameter properly. How the minimum memory size can be calculated is shown in section 7.5.

## 3.5 Bus Driver Interface

The physical interface between WFIFO IP and processor is a bus that includes an address and a data line and a set of transfer qualifier signals. This section describes the instruction format used on address and data bus. First the implemented format is shown, then it is discussed why this format was selected.

### 3.5.1 Implemented Instruction Format

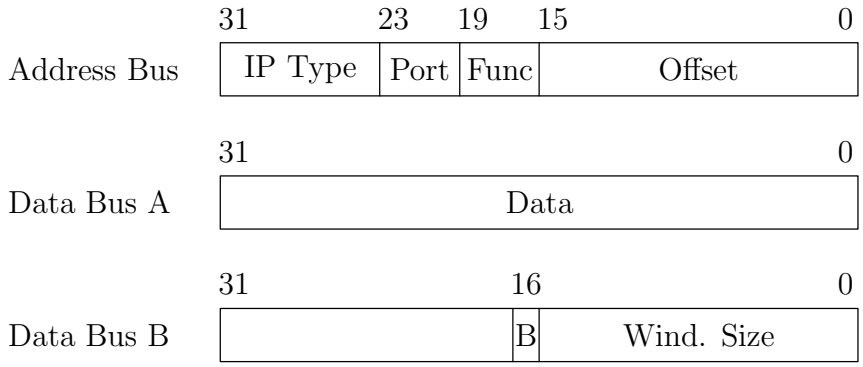



**Figure 3.10:** WFIFO bus interface instruction format. For the data bus, two different formats are used: A for data transfer and B for the other instructions.

| Name | Size (bit) | Range/Value | Function |
|---|---|---|---|
| IP Type | 8 | 0x01 | The IP type is used to separate the address range of the WFIFO IP from other IPs. IP type for WFIFO IPs is one. |
| Port | 4 | 0x0-0xF | The port number is used to identify the target WFIFO. If a WFIFO connects both read and write port to the same bus, two different port numbers must be used. |
| Inst. | 4 | 0x1-0x6, 0xF | Instruction ID identifies the instruction to execute.<br>1 Read<br>2 Write<br>3 Acquire Read<br>4 Acquire Write<br>5 Release Read<br>6 Release Write<br>7 Status<br>F Mir/Reset |
| Offset | 16 | 0x0000-0xFFFF | Offset address for reading and writing to the WFIFO window. |

**Table 3.6:** Instruction format address bus.

| Name | Size (bit) | Range | Format | Function |
|---|---|---|---|---|
| Data | 32 | all | A | Data for read and write. |
| Window Size | 16 | 0x0000-0xFFFF | B | Window size for acquiring. |
| Blocking | 1 | 0x10000 | B | One for blocking, zero for non blocking acquring. |

**Table 3.7:** Instruction format data bus

### 3.5.2 Instruction Format Evaluation

There are different possibilities for the bus transaction format. An implementation with a single instruction type format was selected because this simplifies the API implementation and the hardware design. To make a decision for the instruction format, Table 3.8 was used. This table lists all the information that must be transferred.

| Item Name | Size | Address | Data |
|---|---|---|---|
| Instruction ID | 4 bit | yes | yes |
| Port number | sizeP | yes | no |
| Window size | sizeW | yes | yes |
| Offset | sizeW | yes | yes |
| Data | 32 bit | no | yes |
| IP type | 4-8 bit | yes | no |

**Table 3.8:** Items to include in the instruction format with required size and possible bus selection.

For the size of the instruction ID, four bits are proposed. With four bits, sixteen different instructions are possible. Currently only eight are used and therewith it offers enough room for an extension of the API. The IP type is used to separate the address range of the WFIFO form other IPs connected to the same bus. Port number and IP type cannot be transfered by the data bus because they are used to identify the target WFIFO.

Table 3.9 shows mappings for the address bus that lead to an acceptable maximum port number and maximum window size[1]. Mapping for the data bus is not critical and needs no detailed discussion. The value of <Max Ports> in Table 3.9 is the maximum number of ports for a single processor. In the system there can be many more WFIFOs than this number. The port parameter must not be understood as a unique identifier of the IP. It defines the position of the WFIFO within the address range of the bus.

[1] One data item is 32-bit. This is the smallest block that can be addressed.

| IP Type | SizeP | Func Name | SizeW | Max Ports | Max Window Size |
|---|---|---|---|---|---|
| 4 | 10 | 4 | 14 | 1024 | 64 KB |
| 4 | 8 | 4 | 16 | 256 | 256 KB |
| 4 | 6 | 4 | 18 | 64 | 1 MB |
| 4 | 4 | 4 | 20 | 16 | 4 MB |
| 8 | 4 | 4 | 16 | 16 | 256 KB |

**Table 3.9:** Possible mappings for the address bus. Left: bits used per item. Right: Maximum number of ports and maximum window size.

For the WFIFO implementation the last format from Table 3.9 was used because it offered enough port numbers and a big enough window size for our requirements. At the same time, it uses the smallest address range. The API and and WFIFO IP have been designed to simplify the change of the instruction format. How this can be done for the API is shown in 4.2.

## 3.6 WFIFO Design Parameters

The WFIFO IP offers four design parameters. They must be set during hardware design. With XPS Studio, they are listed under Project-> Add/Edit Cores in the Parameter tab. If WAB is used, the parameters are defined in the platform file. Table 3.11 shows all design parameters. The default values are marked with an asterisk.

The IPIF has a configurable pipeline architecture (Figure 3.11). The pipeline mode design parameter defines which pipeline registers are inserted. Table 3.10 shows all possible configurations. More pipeline registers increase the latency but also the maximum clock speed.

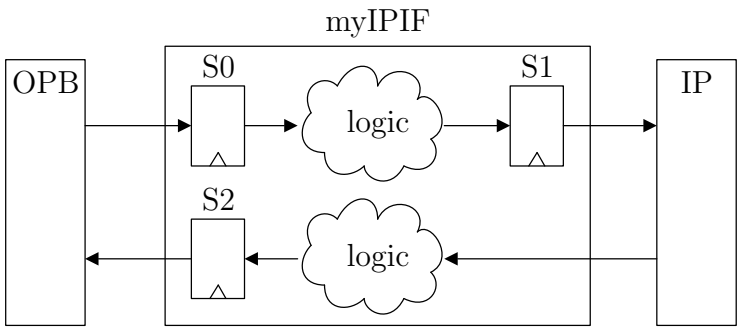



**Figure 3.11:** WFIFO pipeline architecture

| Pipeline Mode | S0 | S1 | S2 | Max Clk. |
|---|---|---|---|---|
| 7 | √ | √ | √ | 173.6 MHz |
| 5 | √ | | √ | 122.7 MHz |
| 3 | √ | √ | | 173.6 MHz |
| 2 | √ | | | 122.7 MHz |

**Table 3.10:** WFIFO pipeline modes

| Parameter | Values | Function |
|---|---|---|
| C_WFIFO_MEMSIZE | 1, 4, 8* | Defines the size of the used BRAM (in KB) and therewith the maximum number of items that can be stored in the WFIFO. Maximum window size is equal to the maximum item number. 8 KB = 254 items, 4 KB = 1022 items and 1 KB = 2046 items. |
| C_WFIFO_WINDWIDTH | 11*-1 | Defines the maximum window width in bits. A smaller value leads to a design with smaller chip size because smaller registers, adders and comparators can be used. |
| C_WFIFO_BRAMWIDTH | 11*, 10, 9 | Defines the width of the address bus used for the BRAM. This parameter cannot be chosen freely. It depends on the memory size: 11 for 8 KB, 10 for 4 KB and 9 for 1 KB. |
| C_WFIFO_PIPELINEMODE | 2, 3, 5*, 7 | The pipeline mode defines which pipeline registers are used in myIPIF. Details are shown in Figure 3.11 and Table 3.10. |

**Table 3.11:** WFIFO design parameters.

## 3.7 WFIFO Implementation Details

### 3.7.1 IPIF IO Signals

Both the read and the write port are connected to an IPIF IP. The IPIF offers many signals to connect the IP, but only few of them are used for the WFIFO IP. Tables 3.12 and 3.14 list all IO signals used by the WFIFO IP. It is important to note that all these IO signals exist twice – once at the write port and once at the read port. The signal names in the design are Bus2IP_W_Addr and Bus2IP_R_Addr for the input listed as Bus2IP_Addr.

| Signal Name | Size | Function |
|---|---|---|
| Bus2IP_Addr | 32 | Encodes the function name and the offset for read or write transaction. The address is valid at rising clock edge if Bus2IP_CS is high. |
| Bus2IP_RNW | 1 | Is a single bit indicating whether the next transfer is read (high) or write (low). |
| Bus2IP_CS | 1 | Indicates that an address was decoded that is in the IP's address range. If Bus2IP_CS is high at rising clock edge, there is a transfer waiting to be processed. |
| Bus2IP_WrCExDI | 1 | Write transaction is pending. |
| Bus2IP_RdCExDI | 1 | Read transaction is pending |
| Bus2IP_Data | 32 | WFIFO data input is used to transfer the data to the WFIFO write port. It is also used to communicate the read and write window sizes on the read and write port. The input is valid on rising clock edge if Bus2IP_CS is high and Bus2IP_RNW is low. |
| Bus2IP_Clk | 1 | Bus Clock. |
| Bus2IP_Reset | 1 | System Reset. |

**Table 3.12:** WFIFO inputs.

To simplify the description of the state machine, new signals are defined as sub-vectors of the input signals from Table 3.12.

| Signal | Range | Signal Name |
|---|---|---|
| IP2Bus_W_Addr | [15:0] | WrAddrxDI |
| | [19:16] | WrCommandxDI |
| IP2Bus_W_Data | [15:0] | WrWindSizexDI |
| | [16] | WrBlockingxDI |
| IP2Bus_R_Addr | [15:0] | RdAddrxDI |
| | [19:16] | RdCommandxDI |
| IP2Bus_R_Data | [15:0] | RdWindSizexDI |
| | [16] | RdBlockingxDI |

**Table 3.13:** Sub-vectors of IP2Bus_Data and IP2Bus_Addr.

Similar to the input signals, the output signals exist once per port. The two signals for the output listed as IP2Bus_Ack are IP2Bus_R_Ack and IP2Bus_W_Ack. The signal IP2Bus_R_Data is not used because no data are written to the read port. Table 3.14 lists the output signals.

| | |
|---|---|
| IP2Bus_Data | WFIFO data output is used for the read port only. It transfers the data read from the WFIFO. The data is valid on rising clock edge if Bus2IP_RNW and IP2Bus_Ack are high and IP2Bus_Error is low. |
| IP2Bus_Ack | For write operations, IP2Bus_Ack indicates that reading is finished. For read operations, IP2Bus_Ack indicates that the data is available. Acquire and release operations are treated as write commands. The acknowledge can be negated with IP2Bus_Error. |
| IP2Bus_Toutsup | Is set to one whenever the WFIFO needs more time than eight clock cycles to send and acknowledge. IP2Bus_Toutsup is used for blocking read and write. As long as the signal is high, the process keeps on waiting. |

**Table 3.14:** WFIFO outputs

### 3.7.2 BRAM IO Signals

The IO signals of the WFIFO BRAM entity are listed in Table 3.15.

| | |
|---|---|
| Bram_W_AddrxDI | Address of write port. |
| Bram_W_DataxDI | Data to write. |
| Bram_W_WrEnablexDI | Enable BRAM writing. Data Bram_W_DataxDI is written to address Bram_W_AddrxDI. |
| Bram_R_AddrxDI | Address of write port. |
| Bram_R_DataxD0 | Data at address Bram_W_AddrxDI. Data output is delayed by one clock cycle. |
| Wfifo_Bram_ClkxCI | BRAM Clock. |
| Wfifo_Bram_RstxRI | Bram Reset. |

**Table 3.15:** WFIFO BRAM IO signals

#### Accessing the BRAM

Accessing the BRAM is very similar to reading and writing a register. For writing, data and address must be available and write enable must be high during one clock cycle. For reading, the read address must be available during one clock cycle. The corresponding data can be read from the data port during the next clock cycle. This behaviour is shown by the timing diagrams 3.12 and 3.13.

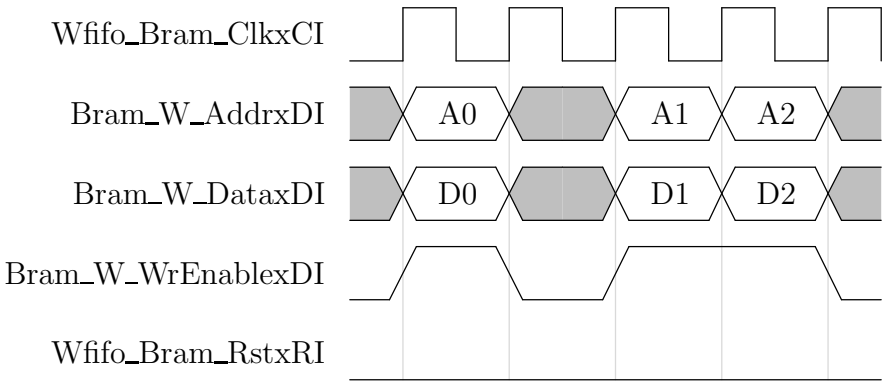



**Figure 3.12:** Timing diagram for WFIFO BRAM for three subsequent write operations.

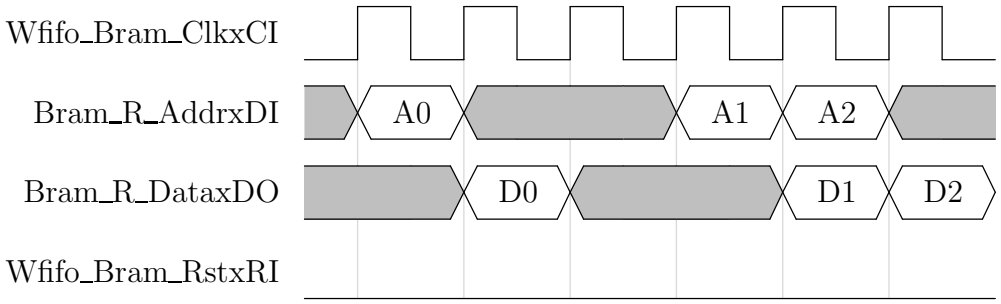



**Figure 3.13:** Timing diagram for WFIFO BRAM for three subsequent read operations.

It is important to note that the hardware does not offer a special write or read port. Both ports can be used for writing and reading and there is no parameter to configure a port for reading or writing only. This has an impact on the WFIFO BRAM write port. During the time the write port is not used, the write enable signal is low. This causes the write port to behave like a read port. The unwanted data delivered by the BRAM is not a problem since it can be ignored, but it must be ensured that the address on the write port differs from the address on the read port. How this is done is shown in 3.3.5.

### 3.7.3 States for Memory Management

Table 3.18 lists all states used for the memory management. The column named Type indicates the state machine that uses the respective state. The two states FreeMem and FifoSize are the shared states, used for the synchronisation of the two state machines. Tables 3.16 and 3.17 show the transitions of these states. Because there are situations where both state machines want to change the shared states in the same clock cycle, a special hardware structure with two adders is used (see Figure 3.17).

| Current FreeMem | Next FreeMem | Next read state | Next write state |
|---|---|---|---|
| FreeMem | FreeMem − WrWindSizexDI | ReRdAck | !AcWrAck |
| | FreeMem + RdWindSizexDI | !ReRdAck | AcWrAck |
| | FreeMem − WrWindSizexDI + RdWindSizexDI | ReRdAck | AcWrAck |
| | FreeMem | !ReRdAck | !AcWrAck |

**Table 3.16:** Free memory state transitions.

| Current FifoSize | Next FifoSize | Next read state | Next write state |
|---|---|---|---|
| FifoSize | FifoSize + WrWindSizexDI | ReWrAck | !AcRdAck |
| | FifoSize − RdWindSizexDI | !ReWrAck | AcRdAck |
| | FifoSize + WrWindSizexDI − RdWindSizexDI | ReWrAck | AcRdAck |
| | FifoSize | !ReWrAck | !AcRdAck |

**Table 3.17:** Fifo size state transitions

| Name | Type | Reset | Type | Function |
|---|---|---|---|---|
| BramWrOffset | int | 0 | W | Memory address of the first item in the write window. The value is changed if a write window is released by adding the size of the released window modulo to the total memory size. |
| BramRdOffset | int | 0 | R | Memory address of the first item in the read window minus one. The value is changed if a read window is released by adding the size of the window modulo to the total memory size. |
| RdWindowV | bool | false | R | Read-window-valid bit is true if a read window has been set up and was not released yet. |
| RdWindowSize | int | 0 | R | Size of the read window. Is only valid if RdWindowV=true. |
| WrWindowV | bool | false | W | Write-window-valid bit is true if a write window has been set up and was not released yet. |
| WrWindowSize | int | 0 | W | Size of the write window. Is only valid if WrWindowV=true. |
| FreeMem | int | Size-2 | R/W | Size of the memory that is currently not used. FreeMem is measured in items: FreeMem = 1 indicates that 32-bit are free. |
| FifoSize | int | 0 | R/W | Current FIFO size without writing windows. FifoSize = 2 indicates that 64-bit are used by the FIFO. |

**Table 3.18:** Variables for memory management.

### 3.7.4 Signal Level State Transitions

| Command Name | Encoding |
|---|---|
| ComAcRead | ”0011” |
| ComReRead | ”0101” |
| ComRead | ”0001” |

| | |
|---|---|
| ComAcWrite | ”0100” |
| ComReWrite | ”0110” |
| ComWrite | ”0010” |
| ComGetStatus | ”0111” |

| Transition | | Input Signals | | | Internal State |
|---|---|---|---|---|---|
| Current State | Next State | Address and Data bus | Rnw | Cs/Ce | WrWindowV |
| RdIdle | AcRdAck | RdCommandxDI = ComAcRead<br>0 < RdWindSizexDI ≤ FifoSize | 0 | 1 | false |
| | RdErr | RdCommandxDI = ComAcRead | 0 | 1 | true |
| | | RdCommandxDI = ComAcRead<br>RdWindSizexDI ≤ 0 | 0 | 1 | |
| | AcRdFail | RdCommandxDI = ComAcRead<br>RdWindSizexDI > FifoSize<br>RdBlockingxDI = 0 | 0 | 1 | |
| | AcRdBlk | RdCommandxDI = ComAcRead<br>RdWindSizexDI > FifoSize<br>RdBlockingxDI = 1 | 0 | 1 | false |
| | RdIdle | RdCommandxDI = ComAcRead | 1 | 1 | |
| | ReRdAck | RdCommandxDI = ComReRead | 0 | 1 | true |
| | RdErr | RdCommandxDI = ComReRead | 0 | 1 | false |
| | RdIdle | RdCommandxDI = ComReRead | 1 | 1 | |
| | RdDaOk | RdCommandxDI = ComRead<br>0 ≤ RdAddrxDI < RdWindSizexDI | 1 | 1 | true |
| | RdErr | RdCommandxDI = ComRead<br>RdAddrxDI ≥ RdWindSizexDI | 1 | 1 | |
| | | RdCommandxDI = ComRead<br>RdAddrxDI < 0 | 1 | 1 | |
| | | RdCommandxDI = ComRead | 1 | 1 | false |
| | RdIdle | RdCommandxDI = ComRead | 0 | 1 | |
| | | | | 0 | |
| | RdStaAck | RdCommandxDI = ComGetStatus | 0 | 1 | |
| *Err | RdIdle | | | | |
| *Ack | RdIdle | | | | |
| RdDaOk | RdDaAck | | | | |
| AcRdBlk | AcRdBlk | RdWindowSize > FifoSize | | | |
| | AcRdAck | RdWindowSize ≤ FifoSize | | | |

**Table 3.19:** WFIFO read state transitions.

| Transition | | Input Signals | | | Internal State |
|---|---|---|---|---|---|
| Current State | Next State | Addr and Data | Rnw | Cs/Ce | WrWindowV |
| WrIdle | AcWrAck | WrCommandxDI = ComAcWrite<br>0 < WrWindSizexDI ≤ FreeMem | 0 | 1 | false |
| | WrErr | WrCommandxDI = ComAcWrite | 0 | 1 | true |
| | | WrCommandxDI = ComAcWrite<br>WrWindSizexDI ≤ 0 | 0 | 1 | |
| | AcWrFail | WrCommandxDI = ComAcWrite<br>WrWindSizexDI > FreeMem<br>WrBlockingxDI = 0 | 0 | 1 | |
| | AcWrBlk | WrCommandxDI = ComAcWrite<br>WrWindSizexDI > FreeMem<br>WrBlockingxDI = 1 | 0 | 1 | false |
| | WrIdle | WrCommandxDI = ComAcWrite | 1 | 1 | |
| | ReWrAck | WrCommandxDI = ComReWrite | 0 | 1 | true |
| | WrErr | WrCommandxDI = ComReWrite | 0 | 1 | false |
| | RdIdle | WrCommandxDI = ComReWrite | 1 | 1 | |
| | WrDaAck | WrCommandxDI = Write Data<br>0 ≤ WrAddrxDI < WrWindowSize | 0 | 1 | true |
| | WrErr | WrCommandxDI = ComWrite<br>WrAddrxDI ≥ WrWindowSize | 0 | 1 | |
| | | WrCommandxDI = ComWrite<br>WrAddrxDI < 0 | 0 | 1 | |
| | | WrCommandxDI = ComWrite | 0 | 1 | false |
| | WrIdle | WrCommandxDI = ComWrite | 1 | 1 | |
| | | | | 0 | |
| | WrStaAck | WrCommandxDI = ComGetStatus | 0 | 1 | |
| *Err | WrIdle | | | | |
| *Ack | WrIdle | | | | |
| AcWrBlk | AcWrBlk | WrWindowSize > FreeMem | | | |
| | AcWrAck | WrWindowSize ≤ FreeMem | | | |

**Table 3.20:** WFIFO write state transitions.

### 3.7.5 Output Signal Values

The values of the output signals depend on the state of the read or write state machine (see Tables 3.21 and 3.22). They can be derived from the internal state by a combinatorial output function. Only the IP2Bus_R_Data signal is different. It comes directly from the WFIFO input.

| Write State | Output Signals | | |
|---|---|---|---|
| | Data | Ack | Toutsup |
| AcWrBlk | 0 | 0 | 1 |
| AcWrAck, ReWrAck, WrErr | 0 | 1 | 0 |
| WrDaAck | Valid Data | 1 | 0 |
| WrDaOk, WrIdle | 0 | 0 | 0 |

**Table 3.21:** Write state machine outputs. All output signal names have the prefix IP2Bus_W.

| Read State | Output Signals | |
|---|---|---|
| | Ack | Toutsup |
| AcRdBlk | 0 | 1 |
| AcRdAck, ReRdAck, RdDaAck RdErr | 1 | 0 |
| RdIdle | 0 | 0 |

**Table 3.22:** Read state machine outputs. Both output signal names have the prefix IP2Bus_R.

### 3.7.6 WFIFO Process Level

In this section, the structure of the VHDL code is explained. A hierarchical view of the processes, including the interconnection signals is shown and a short description of the important processes is given. For implementation details, the VHDL code can be found in Appendix C.1.

#### Overview

The wfifo_top instantiates the IPIFs, the WFIFO BRAM and the state machine and connects these entities. This organisation simplifies the hardware design because the WFIFO can be added in a single step and only the bus connections must be made. Figure 3.14 shows the top view of the WFIFO. In Figure 3.15, details of the WFIFO logic are shown. This block includes the WFIFO state machine and it can be seen that there is a single sequential

process. This process makes the state transitions for tread and write state machine. All other processes are combinatorial. In section 3.7.6, a description of these processes can be found. The IPIF and the WFIFO BRAM will not be discussed any further because they are mainly based on IPs included in the EDK.

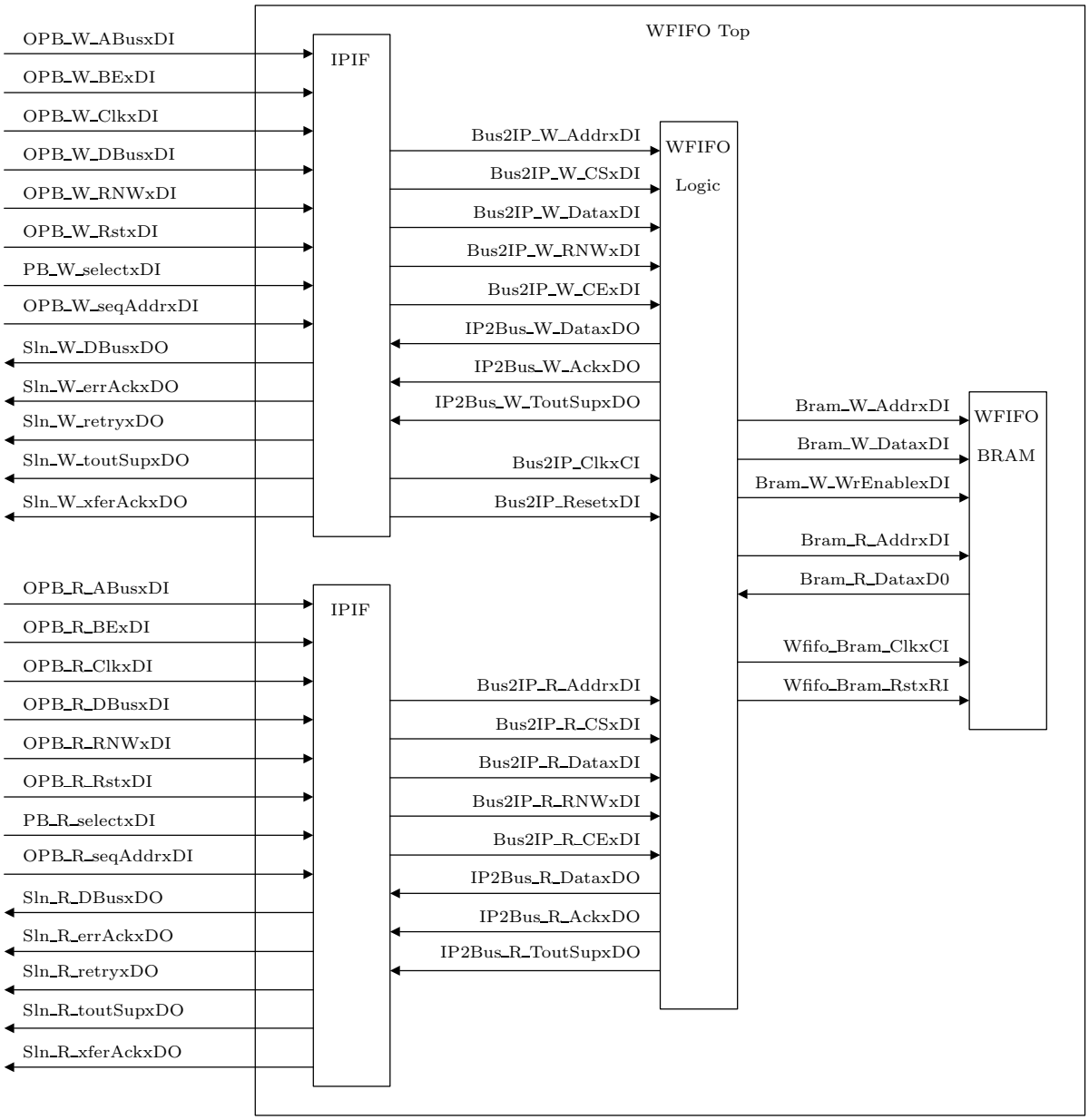



**Figure 3.14:** WFIFO top architecture

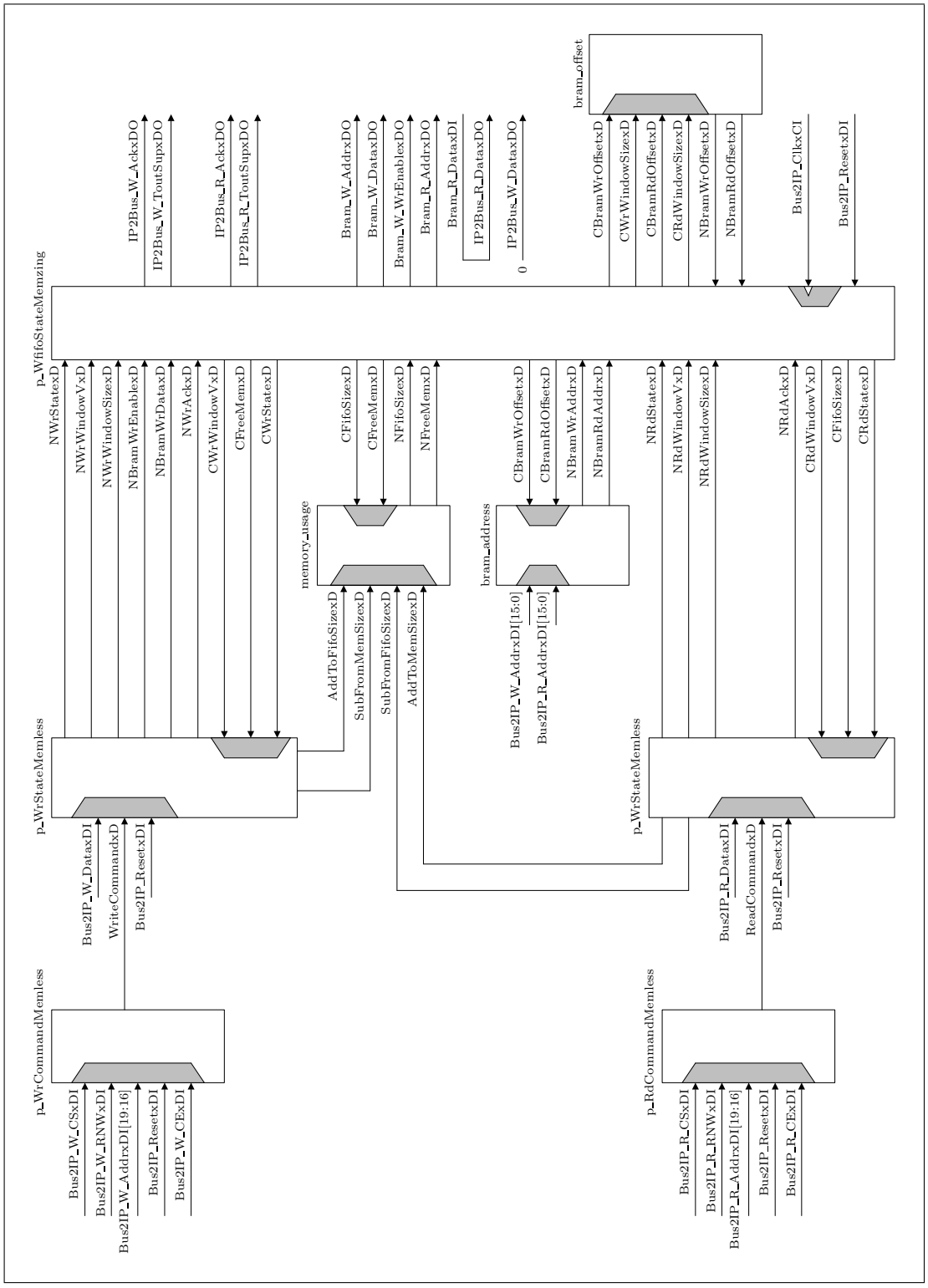



**Figure 3.15:** WFIFO-logic processes and interconnections. Figure 3.16 shows the symbols used.

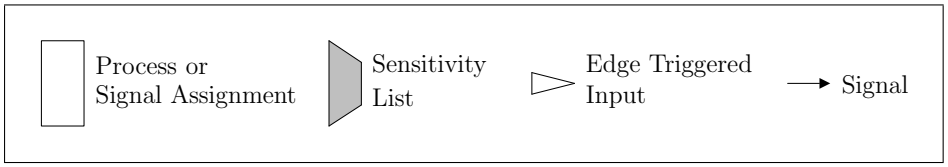



**Figure 3.16:** Legend for architecture drawings. Whenever a process has an edge-triggered input, it is sequential, otherwise it is combinatorial.

### WFIFO Process Description

This section gives a detailed description of the processes shown in Figure 3.15.

**p_WrCommandMemless** This process acts like an input filter for the write state machine. It outputs one of the states s_AcWr, s_ReWr or s_WrDa whenever a valid write instruction has been received (see Table 3.23). It does not check any internal states, it just reads the bus and checks whether the bus transactions are valid. If no command is waiting, the output is s_NoWr.

| WrCommandxDI | Bus2IP_W_CSxDI | Bus2IP_W_RNWxDI | WriteCommandxD |
|---|---|---|---|
| 0100 | 1 | 0 | s_AcWr |
| 0110 | 1 | 0 | s_ReWr |
| 0010 | 1 | 0 | s_WrDa |
| Default | | | s_NoWr |

**Table 3.23:** p_WrCommandMemless: Truth Table

**p_RdCommandMemless** This process does the same as the corresponding write process but for the read state machine (see Table 3.24).

| RdCommandxDI | Bus2IP_R_CSxDI | Bus2IP_R_RNWxDI | ReadCommandxD |
|---|---|---|---|
| 0011 | 1 | 0 | s_AcRd |
| 0101 | 1 | 0 | s_ReRd |
| 0001 | 1 | 1 | s_RdDa |
| Default | | | s_NoRd |

**Table 3.24:** p_RdCommandMemless: Truth Table

**memory_usage** The memory_usage process synchronises the read and the write state machines. The architecture with two adders and two subtracters enables the read and write state machines to change FifoSize and FreeMem in the same clock period.

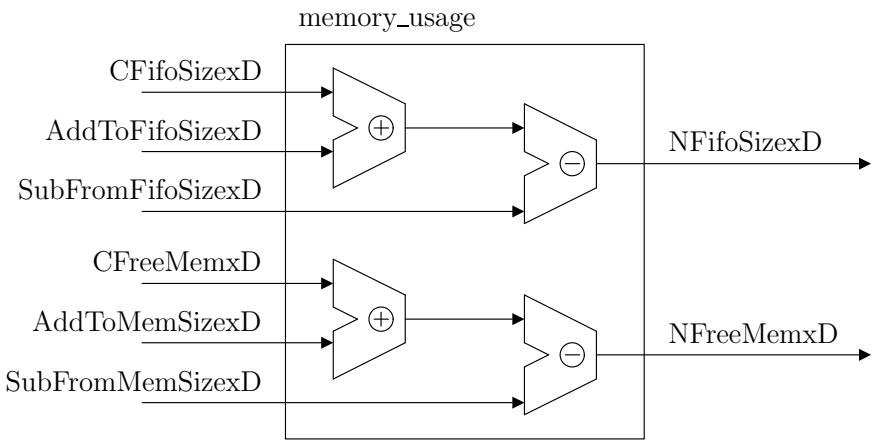



**Figure 3.17:** WFIFO memory usage calculation architecture

**bram_offset** Calculates the next read and write offset. The new value is stored if the current window is released (see Figure 3.19).

**bram_address** Calculates the next BRAM write address from the current offset and the address input. The result is not directly forwarded to the BRAM (see Figure 3.19).

### p_WfifoStateMemzing

For most signals, p_WfifoStateMemzing behaves like a flip flop that separates the next state from the current state. In addition, a multiplexer is used to set the reset value.

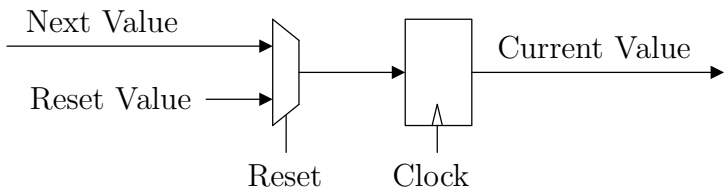



**Figure 3.18:** Not gated signal in p_WfifoStateMemzing

For some signals, a gate or switch was implemented. Figures 3.19 and 3.20 show these signals. All other signals are connected as shown in Figure 3.18.

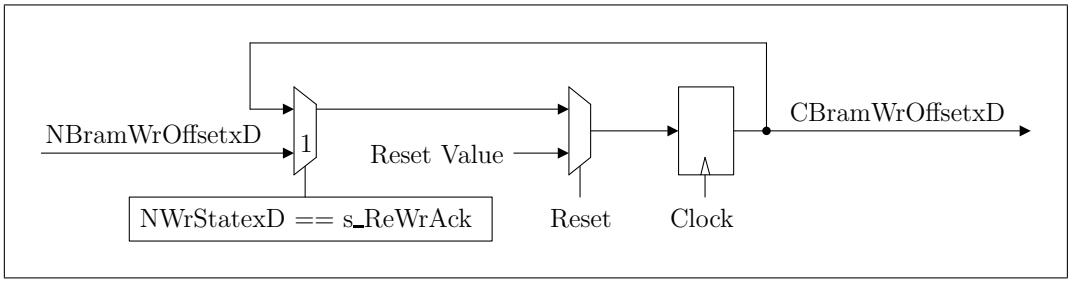



**Figure 3.19:** Gated signals in p_WfifoStateMemzing. The same structure exists for the read offset.

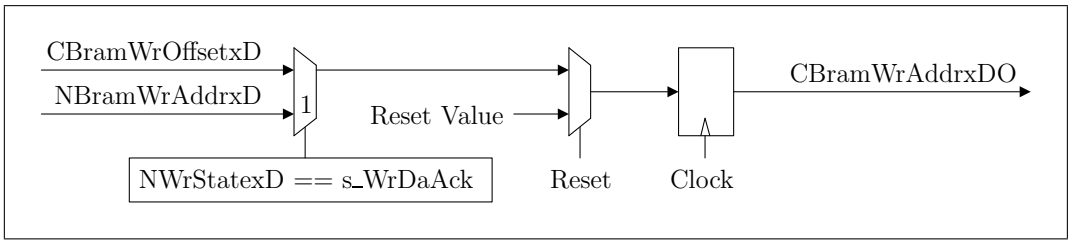



**Figure 3.20:** Switched signals in p_WfifoStateMemzing. The same structure exists for the read address.

#### p_WrStateMemless and p_RdStateMemless

Include the rest of the logic shown in tables 3.19 and 3.20 that is required for the next state transition.

## 3.8 myIPIF

During hardware design of the WFIFO IP, it turned out that the IPIF implementation of Xilinx includes a lot of errors. At the beginning, workarounds could be found, but in the end it turned out that it is better to make a new IPIF design. The main reason for the redesign was the incorrect handling of the Toutsup signal that is used for blocking acquiring instructions. Another reason for the redesign was the devious development of the Xilinx IPIF module. There are many different versions, and each newer version implements more functionality. Some errors were fixed in newer versions but new errors occured with the new functionality. However, I found it amazing to find this kind of errors in a commercial product. The reimplementation of IPIF was called myIPIF. It has the same interface to the IP as the original but offers only part of the functionality. If a better version of IPIF becomes available, then myIPIF will not be required any more. Configuring the myIPIF is complex but it does not differ from configuring the original IPIF. For details, the documentation of the IPIF can be consulted. The only difference of importance is that myIPIF only supports part of the pipeline modes offered by the original. They are listed in 3.6.

# 4 The WFIFO API

The WFIFO API is the link between the program running on the processor and the WFIFO IP. Most of the functionality offered by the WFIFO API is implemented in hardware and the API's job is to translate the WFIFO instructions into bus transactions. The requirements for the API design were that commands are executed as fast as possible in order to reduce the latency and that the API allows efficient IP testing. Figure 4.1 shows the API's position and its dependencies from hardware design and testing environment. The API does not access the hardware directly. It makes use of the bus driver interface that is part of the Xilinx design software (see Figure 4.2). The WFIFO API offers all the commands required by the WFIFO protocol. In addition, it offers commands and functionality that enable or simplify the IP testing.

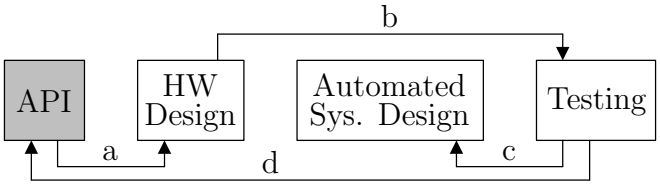



**Figure 4.1:** Overview of practical work.

This chapter starts with a detailed description of the API that includes the instructions for the WFIFO protocol and the instructions used for testing the IP. Then details of the API implementation are presented and it is shown how the need for minimum latency was met. Finally it is shown why error signalling by the error signal line that was proposed in the last chapter could not be implemented.

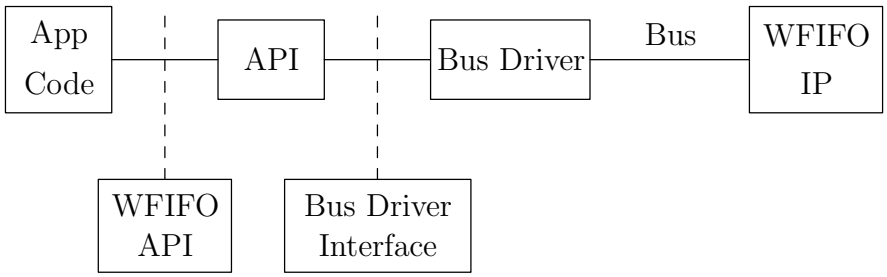



**Figure 4.2:** Communication path from application code to WFIFO IP

## 4.1 API Interface

The WFIFO API is used in application code to access the WFIFO buffer and in the testing environment to verify the IPs functionality. Although the API is a software interface, it cannot be separated from the target architecture. An application can only access the WFIFOs that are connected to the bus of the processor. Each of the WFIFOs has its own address range, which is represented in the API by the port number. For writing application code, these port numbers of the target WFIFOs must be known.

Table 4.1 summarises the instructions offered by the WFIFO API and the subsequent tables list the parameters used by the instructions and the return values. After that, a detailed description of each command is given. To simplify the IP testing, configuration parameters have been defined for the API. The problem description and the solution approach for this configuration are given in 4.1.2.

| Function Name and Parameter | | Description |
|---|---|---|
| WFIFO_ACQUIRE_WRITE<br>WFIFO_ACQUIRE_READ<br>WFIFO_ACQUIRE_WRITE_NONBLK<br>WFIFO_ACQUIRE_READ_NONBLK | size,<br>port | Acquires a read or write window of the specified size. If there are not enough data for a read window or not enough free memory for the write window, the instruction waits until data or memory becomes available. The non-blocking version does not wait. It returns immediately if acquiring is currently not possible. |
| WFIFO_RELEASE_WRITE<br>WFIFO_RELEASE_READ | port | Releases the current read or write window. |
| WFIFO_READ WFIFO_WRITE | port,<br>offset,<br>data | Reads or writes the window position indicated by the offset. |
| WFIFO_GETSTATUS | port | Returns the status of the last WFIFO instruction. |
| WFIFO_RESET | port | Resets the WFIFO IP. |
| WFIFO_GETMIR | port | Returns the IP identifier of the WFIFO. |

**Table 4.1:** Summary of WFIFO API instructions.

The instructions of the WFIFO API can return the values defined in Table 4.2. The return value FAILED is only used by the non-blocking acquiring instructions.

| | |
|---|---|
| WFIFO_OK | The API command was executed successfully. |
| WFIFO_ERROR | The API command was rejected because it violated the WFIFO protocol. The command has not changed the WFIFO's state. The API does only check for such violations if the ASSERT compile option is set. If ASSERT_NONBLK compile option is not set, the execution of the calling process is blocked after such a violation. See 4.1.2 for details. |
| WFIFO_FAILED | The API command could not be executed because there was not enough free memory or not enough data. |

**Table 4.2:** API Return Values.

The instructions of the WFIFO API take a subset of the parameters listed in Table 4.3.

| Name | Type | Description |
|---|---|---|
| port | uint | The port number is a unique identifier for all WFIFO buffers connected to the same bus. The port number is used by the API to calculate the address range of the target WFIFO IP. Setting the port number of a WFIFO IP is done during the design of the target architecture. |
| offset | uint | The offset specifies the read or write position within a window. It must be in the range $0 \leq$ offset $<$ window-size. |
| data | int | Data is a variable of type int. The variable is written on read operations and its content is read on write operations. |
| size | uint | Defines the size of a read or write window. The smallest possible value is one, the highest value depends on the memory size of the WFIFO IP (see Table 3.11). |

**Table 4.3:** API Parameters.

### 4.1.1 Command Details

This section gives a detailed description of all API commands and lists possible exceptions. One exception that all instructions have in common occurs if a read port instruction is sent to a WFIFO write port or vice versa. Such an instructions is treated by the WFIFO as a bus transaction outside of the WFIFO's address range. In the architecture with MB and OPB bus, the instruction call returns with the value zero after eight clock cycles. The same happens if the used port number does not exist. All other exceptions are listed in the instruction descriptions below.

#### WFIFO_ACQUIRE_WRITE(port, size)

**Return Values:** WFIFO_OK, WFIFO_ERROR

**Description:** A new write window with length `size` is created in the WFIFO connected to `port`. If there is enough free memory for the new window, it is created and WFIFO_OK is returned. If there is not enough memory, the calling process is blocked until enough memory becomes available. If the specified window size is bigger than the maximum value (see Table 3.11), the calling process is blocked forever.

**Exceptions:** The instruction returns WFIFO_ERROR if one or both of the following conditions is met.

- There is already a write window in the target WFIFO that was not yet released.
- The size parameter is zero.

#### WFIFO_ACQUIRE_WRITE_NONBLK(port, size)

**Return Values:** WFIFO_OK, WFIFO_ERROR, WFIFO_FAILED

**Description:** A new write window with length `size` is created in the WFIFO connected to `port`. If there is enough free memory for the new window, it is created and WFIFO_OK is returned. If there is not enough memory, the instruction returns WFIFO_FAILED. It does not matter whether `size` is bigger than the maximum value given in Table 3.11 or not.

**Exceptions:** The instruction returns WFIFO_ERROR if one or both of the following conditions is met.

- There is already a write window in the target WFIFO that was not yet released.
- The size parameter is zero.

#### WFIFO_ACQUIRE_READ(port, size)

**Return Values:** WFIFO_OK, WFIFO_ERROR

**Description:** A new read window with length `size` is created in the WFIFO connected to `port`. If there is enough data in internal FIFO buffer, it is created and WFIFO_OK is returned. If there is not enough data, the calling process is blocked until enough data are available in the FIFO. If the specified window size is bigger than the the maximum value (see Table 3.11), the calling process is blocked forever.

**Exceptions:** The instruction returns WFIFO_ERROR if one or both of the following conditions is met.

- There is already a read window in the target WFIFO that was not yet released.
- The size parameter is zero.

### WFIFO_ACQUIRE_READ_NONBLK(port, size)

**Return Values:** WFIFO_OK, WFIFO_ERROR

**Description:** A new write window with length `size` is created in the WFIFO connected to `port`. If there is enough data in internal FIFO buffer, it is created and WFIFO_OK is returned. If there is not enough data, the instruction returns WFIFO_FAILED. It does not matter whether `size` is bigger than the maximum value given in Table 3.11 or not.

**Exceptions:** The instruction returns WFIFO_ERROR if one or both of the following conditions is met.

- There is already a read window in the target WFIFO that was not yet released.
- The size parameter is zero.

### WFIFO_RELEASE_WRITE(port)

**Return Values:** WFIFO_OK, WFIFO_ERROR

**Description:** The current write window in the WFIFO connected to `port` is released and WFIFO_OK is returned.

**Exceptions:** The instruction returns WFIFO_ERROR if there is no write window to release.

### WFIFO_RELEASE_READ(port)

**Return Values:** WFIFO_OK, WFIFO_ERROR

**Description:** The current read window in the WFIFO connected to `port` is released and WFIFO_OK is returned.

**Exceptions:** The instruction returns WFIFO_ERROR if there is no read window to release.

### WFIFO_WRITE(port, offset, data)

**Return Values:** WFIFO_OK, WFIFO_ERROR

**Description:** The value of `data` is written to the write window in the WFIFO connected to `port` at position `offset`.

**Exceptions:** The instruction returns WFIFO_ERROR if one or both of the following conditions is met. After writing, the instruction returns WFIFO_OK.

- There is no write window in the target WFIFO.
- The offset is bigger or equal to the write window size.

### WFIFO_READ(port, offset, target)

**Return Values:** WFIFO_OK, WFIFO_ERROR

**Description:** The value of the read window position `offset` in the WFIFO connected to `port` is read and written to `data`. After reading, the instruction returns WFIFO_OK.

**Exceptions:** The instruction returns WFIFO_ERROR if one or both of the following conditions is met. After writing the instruction returns with WFIFO_OK.

- There is no read window in the target WFIFO.
- The offset is bigger or equal to the read window size.

### WFIFO_GETSTATUS(port)

**Return Values:** WFIFO_OK, WFIFO_ERROR, WFIFO_FAILED

**Description:** Returns the status information of the last instruction executed on the indicated port. The status is one of the return values listed above. Although it is possible to execute the `WFIFO_GETSTATUS` function in application code, this should not be done. This function must be used for API programming only.

### WFIFO_RESET(port)

**Return Values:** none

**Description:** The WFIFO IP connected to `port` is resetted. This means that all valid data in the WFIFO are lost and that address pointers and other internal values are set to their initial state. The implementation of the software reset is located in the IPIF module. When receiving the software reset command, the IPIF sets the reset signal for one clock cycle. The WFIFO IP never receives this command, it only sees the toggling of the reset line.

The software reset is very useful for IP testing. Without the software reset, a new simulation must be started for each test-case because the IP has to be in a defined start

state. With the software reset, all test-cases can be copied to the same application. The software reset that is performed after every test-case ensures that the different test-cases cannot influence each other.

#### WFIFO_GETMIR(port)

**Return Values:** mir-value

**Description:** The IP identification command returns an identification and a version number of the WFIFO connected to `port`. It can be used by the processor to test whether the WFIFO is properly connected and it allows to execute WFIFO version dependent code.

Again, this function is implemented in IPIF. When receiving the IP identification command, the IPIF returns the ID immediately. No interaction is done with the WFIFO IP. The ID is defined in the WFIFO IP as a constant and transferred to the IPIF at compile time. IPIF also defines the encoding for the ID (vector 31 down to 0):

| Bits | Function | Current Value |
|---|---|---|
| 31–28 | major version number | 1 |
| 27–21 | minor version number | 0 |
| 20–16 | minor version letter (a=0,b=1) | a |
| 15–8 | block id | 3 |
| 7–0 | block type | 1 |

**Table 4.4:** Encoding of WFIFO ID value

### 4.1.2 Error Handling and Compile Options

There are two different reasons why the execution of a WFIFO API command can fail. The first is the execution of a non-blocking acquiring operation in case that there is not enough memory or data. Such a situation is not an error, but still the acquiring is not executed. Such a situation is signalled with the return value WFIFO_FAILED. The second reason is a violation of the WFIFO protocol such as reading before acquiring. In such a situation the value WFIFO_ERROR is returned. It is important to distinguish between the two reasons because the first one happens during a correct operation, whereas the second one is caused by a programming error. Errors caused by the illegal use of the WFIFO API occur only during the software design phase but must be fixed for the final implementation. This situation causes two problems:

- The WFIFO IP always rejects instructions that violate the WFIFO protocol but extra computing time is required for the API to check if a command was rejected or not. An API implementation that always does this checking results in a lower system performance. It would be desirable if the checking could be enabled during software design and disabled for the final version.

- If errors are signalled by the return value, major changes of the source code are required between design phase and final version. The the source code listed below shows an example code sequence during software design and the corresponding final version. This process of code manipulation is error-prone and time consuming. For this situation it would be better if the errors were written to a log file and not signalled by a return value. But there are situations where error signalling by return value is the best solution. One such application is a test where an error is caused voluntarily in order to check if it is correctly rejected.

```
// example code during software design
nrOfErrors = 0;
if (WFIFO_ACQUIRE_WRITE(1,6,0)  != WFIFO_OK) nrOfErrors++;
if (WFIFO_WRITE(1,0,0x12345678) != WFIFO_OK) nrOfErrors++;
if (WFIFO_RELEASE_WRITE(1)      != WFIFO_OK) nrOfErrors++;

// example code for the final implementation
WFIFO_ACQUIRE_WRITE(1,6,0);
WFIFO_WRITE(1,0,0x12345678);
WFIFO_RELEASE_WRITE(1);
```

In the following, checking for errors caused by the illegal use of API commands is called assertion. The switching mechanism indicated above was implemented with preprocessor statements in the API implementation. If the flag `WFIFO_ASSERT` is set, the version with assertion is used, otherwise no assertion is done. The gcc-mb compiler offers the `-Wp,...` option to pass statements to the preprocessor. A compilation call with assertion has the following structure:

```
mb-gcc ...  -Wp,-DWFIFO_ASSERT
```

The API offers two types of assertion, a blocking and a non-blocking one. The blocking version writes an error message to standard output and stops the execution with an endless loop if an error occurs. The non-blocking version only prints the error message. The flag `ASSERT_NONBLK` must be set for the non-blocking version, the blocking version is the default. If the `WFIFO_ASSERT` flag is not set, then `ASSERT_NONBLK` is ignored. A compilation call with non-blocking assertion has the following structure:

```
mb-gcc ...  -Wp,-DWFIFO_ASSERT -Wp,-DASSERT_NONBLK
```

There is a menu in XPS where additional compiler flags can be set. It is located in Project->Software-Platform-Settings in the processor tab. For the modular testing environment, there are parameters to set the assertion and blocking compile options.

#### Discussion

The presented error handling concept satisfies the demands for safety, speed and debugging. The error handling is save because the WFIFO cannot be set to an invalid state by violating the WFIFO protocol. The API offers maximum performance for the final implementation and debugging information during the design phase. Furthermore, no changes of the program source are required for the final implementation.

### 4.1.3 Command Latency

The latency of the commands depends on the pipeline mode and on the assertion type used. Tables 4.5 and 4.6 summarise the results.

| Write | Read | Acquire | Acq. Non-Blocking | Release | Pipeline Mode |
|---|---|---|---|---|---|
| 5 | 6 | 5 | 12 | 5 | 2 |
| 6 | 7 | 6 | 13 | 6 | 3 |
| 6 | 7 | 6 | 13 | 6 | 5 |
| 7 | 8 | 7 | 14 | 7 | 7 |

**Table 4.5:** Number of required clock cycles for WFIFO instruction without assertion.

| Write | Read | Acquire | Acquire Nonblk | Release | Pipeline Mode |
|---|---|---|---|---|---|
| 12 | 13 | 12 | 12 | 12 | 2 |
| 13 | 14 | 13 | 13 | 13 | 3 |
| 13 | 14 | 13 | 13 | 13 | 3 |
| 14 | 15 | 14 | 14 | 14 | 7 |

**Table 4.6:** Number of required clock cycles for WFIFO instruction with assertion.

## 4.2 WFIFO API Implementation

This section presents details of the API implementation. The complete API source code is shown in Appendix B.1. The job of the API is to translate WFIFO instructions into

instructions for the bus driver. As already shown in Figure 4.2, the API uses a bus driver for this purpose. The XIO-Driver from Xilinx offers the following instructions:

```
XIo_Out32(Address, Data)                    XIo_In32(Address)
```

The driver implements the handling of the transfer qualifier signals and the WFIFO API only has to submit 32 bit values for data and address bus. The first instruction is used for writing to the bus, the second one for reading. The data are provided by the application. The API's job is to translate the port number into the bus address of the WFIFO. Since this translation is static, it is possible to implement the API with function macros instead of normal functions. This speeds up the execution because the function call is done at compile time. The preprocessor replaces the WFIFO API instructions with instructions for the bus driver. No API function call is done at run-time. The translation from port number to bus address depends on the selected instruction format. In order to simplify the adaptation of the API for new instruction formats, the format is defined in a separate section of the API. The code below shows this section.

```
//mask
#define WFIFO_IPTYP_MSK 0xFF000000 // 8 bit ip typ  (31 downto 24)
#define WFIFO_PORTN_MSK 0x00F00000 // 4 bit port no (23 downto 20)
#define WFIFO_FUNCN_MSK 0x000F0000 // 4 bit instr.  (19 downto 16)
#define WFIFO_OFFST_MSK 0x0000FFFF // 16 bit offset (15 downto 0)

// shift
#define WFIFO_PORTN_SFT 20
#define WFIFO_OFFST_SFT 0

// settings
#define WFIFO_READ_ID          0x00010000
#define WFIFO_WRITE_ID         0x00020000
#define WFIFO_ACQUIRE_READ_ID  0x00030000
#define WFIFO_ACQUIRE_WRITE_ID 0x00040000
#define WFIFO_RELEASE_READ_ID  0x00050000
#define WFIFO_RELEASE_WRITE_ID 0x00060000
#define WFIFO_GETSTATUS_ID     0x00070000
#define WFIFO_MIR_RESET_ID     0x000F0000
#define WFIFO_IPTYP_ID         0x01000000
```

The instruction format definition uses three types of values: MSK, SFT and ID. MSK stands for mask. It is a bit mask with value one at the bit positions that are valid for the corresponding parameter and zero elsewhere. SFT stands for shift and indicates the position of the LSB bit. The ID values define the instruction numbers and the IP type identifications. When changing the instruction format, the API can be updated by changing these values.

Because function macros are used, it is not possible to check parameters at run-time. Errors caused by protocol violations are caught by the WFIFO IP but the API has to guarantee that the instruction format is respected. For example, if there is a maximum of 16 ports, it must be ensured that writing to port 20 does not lead to illegal bus transaction possibly outside the IPs address range. As shown in the code below, the API makes a logical and operation with the parameters and the bit masks defined above. This deletes all bits outside the valid range of the parameter.

```
#define WFIFO_WRITE_I(port, offset, data) \
(XIo_Out32( \
   ( WFIFO_IPTYP_ID \
     | (WFIFO_PORTN_MSK & (port<<WFIFO_PORTN_SFT)) \
     | WFIFO_WRITE_ID \
     | (WFIFO_OFFST_MSK & (offset<<WFIFO_OFFST_SFT))),\
   data)\
)
```

The write function macros from above is called by one of the macros shown below. If the API is compiled with assertion, the first macro is used. This macro makes two bus transactions for each WFIFO instructions: one to execute the instruction and a second to read the status of the execution. If the API is compiled without assertion, the status checking is skipped.

```
#ifdef WFIFO_ASSERT

#define WFIFO_WRITE(port, offset, data)\
 (WFIFO_WRITE_I(port, offset, data), wfifoAssert(WFIFO_GETSTATUS(port)))

#else

#define WFIFO_WRITE(port, offset, data)\
 (WFIFO_WRITE_I(port, offset, data))

#endif
```

The wfifoAssert() function checks the status of the instruction. If the status is WFIFO_ERROR, it writes an error message to standard output and blocks the calling process if required. If the status is not equal to WFIFO_ERROR, the assert function does nothing. The source code of the assert function is given in Appendix B.1. More details of the assertion are explained in 4.1.2.

## 4.3 Alternative Status Signalling Concept

The implementation of the status signalling developed for the WFIFO is not the most obvious one and it seems that it could be implemented more efficiently. The main disadvantage of the used implementation is that it needs two bus transactions for a single instruction if the status of the instruction must be checked. Furthermore, this situation has led to the assert compile options, and the API became more complex.

The OPB bus offers a single bit line to signal an error. If this line is used, it is possible to implement WFIFO instructions with a single bus transaction that includes the error signalling and no function to read the status is used any more. In the first version of the WFIFO API and the IP, the approach with the error signal was implemented. This section describes the steps required and shows the problems that occurred. Finally, this concept was replaced by the signalling concept with the extra bus transaction because it could not be implemented with acceptable performance. An advantage of the version finally implemented is that it is less hardware dependant. It can also be used on a bus that has no error signal.

### 4.3.1 Accessing the Error Signal with MB

The MB-Processor has a set of general purpose registers and a set of special purpose registers. The special purpose registers of interest are the Machine Status Register (MSR) and the Exception Status Register (ESR). The incoming error signal is saved to the ESR register if error signalling is enabled in the MSR register. There is no API or driver to access the MSR and ESR registers. The only possibility for reading and writing is to use assembler instructions. Fortunately, the MB C-Compiler allows to use inline assembler. In the code below, the EE bit in the MSR register is set and ESR register is read to memory.

```
int esrRegister;

int main(void){
  // set ee bits in msr register
  asm( "msrset r12, 256" );

  // read esr register
  asm( "mfs r12, resr" );
  asm( "swi r12, r0, esrRegister" );
}
```

Configuring the MSR register can be done with `msrset`. The register r12 is used to store the content of the MSR register before setting the EE bit and is of no relevance here. The `msrset` instruction requires a single clock cycle. On the other hand, reading the ESR register needs two instructions and three clock cycles. In a first step, the content of the ESR register is written to the register r12, then r12 is written to the memory. By default, access for

MSR and ESR register is disabled by hardware. It must be enabled with the MB design parameters `C_USE_MSR_INSTR`, `C_IOPB_BUS_EXCEPTION` and `C_DOPB_BUS_EXCEPTION`.

### 4.3.2 IPIF Problems with Error Signal

To signal an error, the WFIFO IP has to reply by setting both IP2Bus_Error and IP2Bus_Ack to high for one clock cycle. This signalling worked correctly for read but not for write transactions. After spending a lot of time on tracking this problem, the error was located in the IPIF block. The IPIF implementation used allows error signalling only during the first clock period, which is in contradiction to the IPIF specification [14]. The WFIFO IP usually replies one clock cycle after receiving a new command (see Figure 3.6). To overcome the problem, the WFIFO implementation was changed to signal the error immediately. The disadvantage of this solution is the long data path from the OPB bus through the IPIF and all the logic in the WFIFO IP. This may have major impact on the timing for the entire design. The IPIF has a design option that breaks down the length of this path by adding additional pipelining registers. Therewith the timing problem is reduced at the cost of latency. Another option would be to fix the IPIF.

### 4.3.3 MB Exception Handling

The MB processor handles an incoming error on the OPB bus like a hardware exception or an interrupt. On incoming error, the MB branches to the hardware exception vector and from there to the label _hw_exception_handler. Then it does the following operations.

r17 ← PC
PC ← 0x00000020
MSR[EE] ← 0
MSR[EIP] ← 1
ESR[EC] ← exception specific value
ESR[ESS] ← exception specific value
EAR ← exception specific value
FSR ← exception specific value

Some of the operations listed above are of minor interest, but the list shows that many operations are involved. Many more instructions are used for context saving and restoring. The table below gives an overview of the required time. It ignores the time used for reading the ESR register.

| Command | Signal | Required Clk Cycles |
|---|---|---|
| Write | No Error | 8 |
| | Error | 100 |
| Read | No Error | 8 |
| | Error | 100 |

**Table 4.7:** Required time for error signalling

It might be possible to overwrite the exception handler in order to speed up the procedure, but no documentation is available. Furthermore this is not a good idea for compatibility reasons.

### 4.3.4 Summary

The solution using the error signal of the OPB bus requires a lot of processor specific instructions and is extremely slow. Furthermore, implementation is made more difficult by the errors in IPIF IP and the smart model. The preprocessor of the MB C-Compiler does not like assembler instructions. Calling assembler instructions from a function macro is possible only if it is hidden in a normal function. This slows down the operation because an extra function call must be done. In the Modelsim simulation, access to ESR and MSR register does not seem to work correctly. I assume that the problem is located in the smart model used by Modelsim.

# 5 Testing

This chapter deals with two different aspects of testing: verification and performance measurement. Although they address different problems, it will be shown that they demand a similar infrastructure. This chapter starts with a summary of the requirement of functional and performance tests. Section 5.1 points out the problems of testing the WFIFO IP by analysing some general aspects of hardware testing. In 5.2 the implemented testing environment is presented and in 5.3 it is shown how testing was automated.

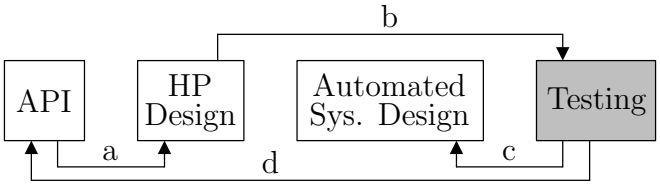



**Figure 5.1:** Overview of practical work.

**Verification:** Verification is focused on testing an IP or a system for functional correctness. These tests ignore timing or performance constraints. They are only concerned with the output generated for known input data or a known start state. Functional tests usually end with the result passed or failed. Because a single test can only verify a part of the functional behaviour, a big set of tests is used. Functional test are important during the design and redesign phase, where the same tests are frequently executed. For efficient testing, the execution of a set of tests for a given IP must be automated.

**Performance measurement:** Performance tests are used to estimate the performance of an IP or an entire system. These tests assume that the IP is working functionally correct and measures parameters like latency and data throughput. If the IP is not working functionally correct, a performance test does not necessarily provide meaningful results. Performance tests end with a score for a certain parameter. They can only pass or fail if a constraint to be met was defined. Performance test are used after a redesign to check whether a system meets the requirements, or to compare different systems. For a comparison, the same test is executed on different IP versions or different system architectures. To make design more efficient, the testing environment should automate this process.

## 5.1 Challenge of Testing – Design for Testability

Testing and verifying a hardware IP is very different from doing tests on a software implementation. Because hardware is designed in languages like VHDL and Verilog, that look very similar to programming languages, software engineers usually have difficulties to understand the difference. I will to point out the problems that appear when testing hardware by explaining the implementation alternatives for making hardware testable. Each of the following itemisations deals with one question and explains which decisions were made for the WFIFO IP. The concepts used are not based on textbook knowledge, as far as I know. They are based on my practical experience and they show my own approach to this subject.

- To perform tests on hardware, an execution environment is required. Such an environment can be a simulator like Modelsim or an FPGA system. The advantage of a simulator is that it offers maximum controllability and observability of internal signals, but the simulation speed is very low. For systems of medium complexity, a simulation of one second of operation usually requires a simulation time in the dimension of hours or days. In contrast to the simulator, the FPGA environment runs with high speed, but it does not allow access to internal signals. In order to observe or control such signals, additional hardware is required. When mapping the design on an FPGA, the hardware must be written in the subset of VHDL that allows synthesis.

  The WFIFO test environment allows to execute tests with the simulator Modelsim or with an FPGA board. This approach combines the advantages from simulator and FPGA execution environments.

- An isolated IP does not perform any operations. To test its behaviour, the IP must be stimulated and the IP's response must be analysed. For systems without processors, a test-bench is a popular solution for this purpose. Most test-benches read the stimulation data from a file and save the output of the IP for further analysis. Such test-benches can only run on a simulator, and mapping on an FPGA is not possible. For systems with processors, the stimulation, and for certain applications even the verification of the output, can be done with a test application running on the IC's processor. Instead of the software environment of the test-bench, the IP runs in the target hardware architecture which also allows FPGA mapping of the test system.

  The testing concepts presented for the WFIFO do not use a test-bench. Tests are performed by integrating the IP in the target architecture and by executing a special test code on the processors. The solution presented can be executed with a simulator or mapped on an FPGA.

- A further challenge of testing is to get enough information from the test to state if or how good something works. There are different locations that can serve as a data source. This can be seen when the system shown in Figure 5.2 is examined. This system shows that test data can be delivered from processor, memory, bus and IP. For a more complex system, the situation is the same except that there are more sources of the same type. All of these sources have partial information about the state of the

system and it must be decided which information is required. It must be considered that functional and performance tests require different data. For functional test, the data transferred and stored are of interest. In addition, performance tests need timing information.

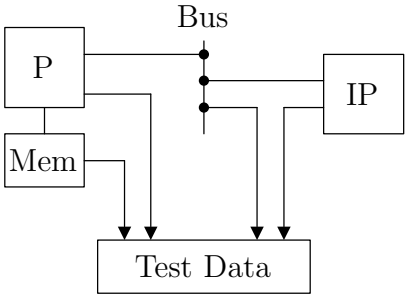



**Figure 5.2:** Possible sources for test data.

**Processor:** The processor source type is particularly suitable for functional tests because it is the only device that can stimulate and verify. For such a test, the processor executes a code sequence and checks whether the IP behaves as expected. Several processors even have special debugging interfaces that allow a step by step execution or additional data exchange. Performance tests are more difficult to accomplish with a processor because the testing itself can have an influence on the performance. Furthermore, a timer is required to measure the elapsed time.

**Memory:** The memory contains a lot of information of which usually only a small part is of interest. In order to find this part, the address map of the application must be known. A possible implementation for performance tests is to stop the execution after a known number of clock cycles and to make a memory dump. This gives all stored data for a specific point in time.

**Bus:** The bus implementation is usually taken from an IP library and it is difficult to change the hardware for testing purposes if it is not already integrated. In contrast to the processor, it is not possible to change the behaviour of the bus by software. This is why the data from the bus are more difficult to read. A possible approach is to use an extra hardware block connected to the bus that acts as a recorder of bus transactions. Such recorders are usually not taken from a library and it is possible to add a timer that allows to save a time stamp for the data observed. The data from the recorder can be used for functional and performance tests.

**IP:** The advantage of the IP data source is that it makes it possible to change the hardware. This allows to integrate additional testing hardware. If this testing infrastructure is added permanently, additional chip size is required. If it can be removed, it must be verified that removing the test hardware does not change the test results. Data produced by the IP can be used for functional and performance tests.

For the WFIFO design, the processor and the bus are used as sources for test data. Data from the processor are used for functional tests because this solution offers the best functionality. A bus recorder is used for performance tests. It was selected because it can be added and removed to a system without changing its behaviour for the test execution and because no changes of the tested IP are required.

- All the data collected during a test must be written out. The options depend on the execution environment:

  **Modelsim:** In a Modelsim simulation, data can be written out to a file with VHDL instructions. Such a kind of log-file writing can only be done if changing the corresponding VHDL code is possible. This is the case for self-written IPs but it is usually not possible for third-party components like processor, memory or bus, because they are taken from a protected library. There are two approaches to overcome this restriction. The first approach is to replace the protected IP with a simulation model that implements the functionality of the original IP *and* the logging. The second approach makes no changes on the protected IP. To write out the information, a sniffer is used that is connected to some of the IP's signal lines. From the information grabbed, the sniffer writes a log-file. Since the sniffer and the simulation model do not have to be synthesizeable, it is possible to use higher level languages like System C for this implementation. A popular solution is to connect a sniffer with a memory region.

  **FPGA:** For FPGA simulations, one can choose the output interface from the ports offered by the FPGA board (COM, LPT, PS2, SATA etc.). Figure 5.3 shows a simplified WFIFO architecture. In this architecture, only the IP and the processor P have access to the I/O device. For the WFIFO IP, the processor is the only device with I/O access because the WFIFO is a slave device that cannot initiate any bus transfers. During the simulation, the selected output port must be observed to store the data. This can be done by a computer or another storage device.

It would be best if the implementations for FPGA and Modelsim did not differ at all in order to run the same tests in both environments. But unfortunately this is difficult to achieve because the FPGA simulation inevitably needs a hardware interface like the COM port to write out the data. For the Modelsim simulation, it is difficult to read such an output because this would mean reading from a virtual COM port that exists only in a simulation.

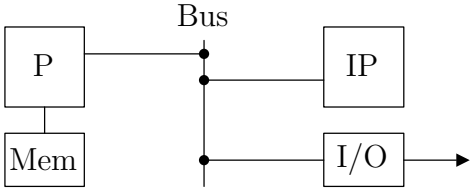



**Figure 5.3:** Architecture example for WFIFO IP

The WFIFO system uses a UART IP that is connected to a COM port to write out data during the FPGA execution. When running a Modelsim simulation, the UART is replaced by an UART-Logger that has the same bus interface as the UART, but writes out the received data to a text file. With this solution it is possible to use the same test software for FPGA and Modelsim but two versions of the same system are required: one with UART IPs and a second one with UART-Loggers. The advantage of this approach is that the changes between Modelsim and FPGA are limited to the output device. All of the central components like the IP, the processor and the software are unchanged because they do not include components that can be used for simulation only. The solution with a sniffer was not used because it can be used for simulation only. Furthermore, there is a major dependency from the version of the observed IP. Whenever a new version of the IP is used, the sniffer must be update.

## 5.2 Testing Architecture

Two systems were mainly used to test the WFIFO IP. They are called single- and dual-bus system (see Figure 5.4 and 5.5 respectively). In the single bus system, the WFIFO is connected with a single processor, in the dual bus system with two processors. With the two processor system it is possible to test simultaneous read and write, whereas in the single processor system read and write are executed alternately. For functional verification, a set of test cases was used. A short description of each test is given in Appendix D. Besides the test architectures, the present section contains a short description of the OPB recorder and the UART IPs and presents two example test cases, one for functional testing and one for performance testing. A more detailed description of OPB recorder and UART-Logger is given in 5.5 and 5.4 respectively.

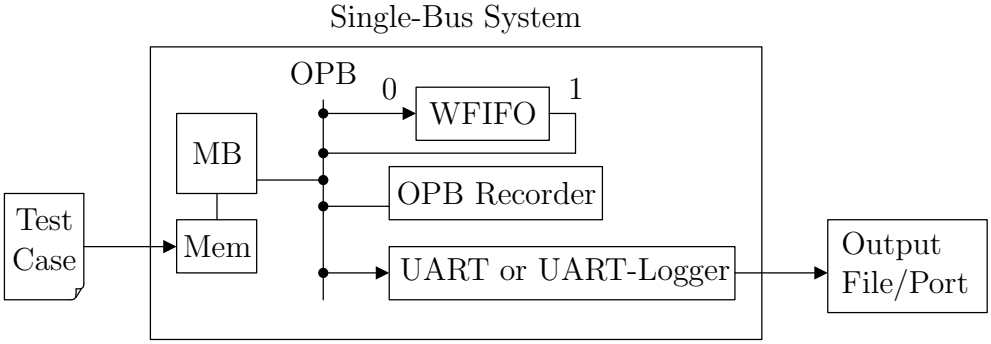



**Figure 5.4:** Single bus WFIFO IP test architecture. The digits next to the WFIFO are the port numbers.

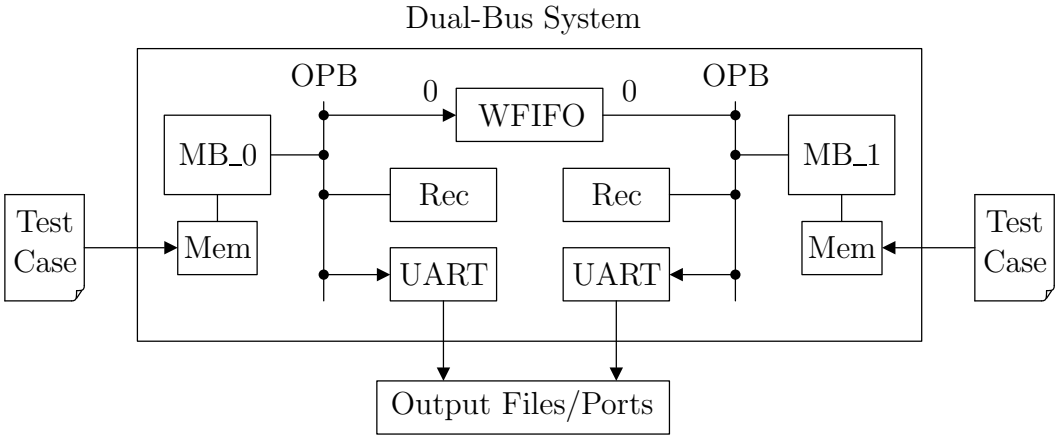



**Figure 5.5:** Dual bus WFIFO IP test architecture.

### 5.2.1 Data Output

All data are written out by the processor(s). As output device, the UART-Logger is used alternately with a normal UART IP. The normal UART IP writes out to a hardware port, whereas the UART-Logger writes out to a log-file. For FPGA simulations, the normal UART is used. For Modelsim simulations it is replaced by the UART-Logger. Since both IPs have same input interface, the rest of the IC cannot see any difference.

UART and UART-Logger belong to a special IP class, called the standard output peripherals. For each processor in the system, a standard output peripheral can be defined. This causes all write operations to stdout to be redirected to this peripheral.[1] Xilinx offers a slimmed version of the `printf` function named `xil_printf`. It only supports the format parameters %d, %l, %x, %s and %c but uses much less instruction memory than `printf`. Below, an example for data output is shown where `errNo` is a variable of type integer. If the test is running on the FPGA, the data can be read from the COM port. If the test is running on the simulator, the result is written to a file.

```
xil_printf("number of errors: %d\n",errNo);
```

The default UART speed of 9600 baud is acceptable for FPGA simulations but for Modelsim simulations it is too slow. To speed up, the baud rate can be changed to 921600 for Modelsim simulations. For both simulation types it must be considered that the UART is not suitable to write out big amounts of data. More information on the UART can be found in the Xilinx IP description.

[1] In XPS this setting is located in Project->Software Platform Settings->Library/OS Parameters.

### 5.2.2 Data Acquisition

Data are collected by the processor and the OPB recorder. For functional tests, only data from the processor are used. A typical functional test sends instructions to the WFIFO buffer and checks if it behaves as expected. After checking, the processor writes out whether the test passed or failed.

Performance tests are accomplished in two phases. In the first phase, data are collected by the OPB-Recorder, in the second phase, the recorded data are transferred to the processor and from there to the output device. For these two phases, the recorder operates in two modes, the record mode and the play mode. In record mode, it only records the bus transaction but it never writes anything to the bus. For all other IPs connected to the bus, the recorder is invisible and their behaviour is not affected by the recorder. If the recorder is switched to play mode, it writes out the data recorded. Table 5.1 lists the functions of the recorder's API. The source code is given in Appendix B.2.

| Name | Return | Function |
|---|---|---|
| OPB_RECORDER_INIT() | none | Clean the recorder memory and set it to record mode. |
| OPB_RECORDER_PLAY() | none | Set recorder to play mode. Output will start with the oldest item and end with the newest. Data output is triggered by the GETNEXT function. |
| OPB_RECORDER_HASMORE() | int | Returns one if there are items in the record buffer that have not been played yet and returns zero otherwise. Recorder must be in play mode for this function. |
| OPB_RECORDER_GETNEXT() | int | Returns the next unread data. Each data item requires two read operations because one item uses 36 bits. |
| replayOpbRecorder() | none | This function reads all of the recorder memory and writes it to standard output. |

**Table 5.1:** API of OPB Recorder

Below, a typical code sequence in a functional test is shown. First, the recorder is initialised, then the WFIFO is stimulated and finally the recorded results are written out.

```
OPB_RECORDER_INIT();  // init the recorder
...                   // execute some wfifo instructions
replayOpbRecorder();  // write out all recorded data
```

The OPB-Recorder does not record all bus transactions. It makes a filtering for WFIFO commands and records only these commands. The address range that is recorded is defined by design parameters (see 5.4). The OPB-Recorder saves the exact time and the WFIFO command type to the BRAM. The time is encoded with 32 bits, which is enough to record for 43 seconds at 100 MHz. The command is encoded with four bits.

### 5.2.3 Example of Functional Test

Functional tests make use of UART or UART-Logger but do not use the OPB-Recorder. As stated above, every functional test includes two steps. First, the processor executes a code sequence and checks whether the IP behaves as expected. Then it signals whether the test passed or failed. The UART is used to write out this simulation result. The example code below shows a test on the single-bus architecture. First, a write window of size six is acquired, then a correct write operation to position zero is made. Then it tries to write at position six, which is outside the window, and checks if an error is signalled. After that, the write window is released, and the data are read back and checked. Finally, the test result is written out. If the test passed, it ends with: `test ended with 0 error(s)`. The test must be complied with nonblocking assertion.

```
/* include files */

int temp, nrOfErrors;

int main(void){
  nrOfErrors = 0;
  temp = 0;

  //write
  if (WFIFO_ACQUIRE_WRITE(0,6)    != WFIFO_OK)    nrOfErrors++;
  if (WFIFO_WRITE(0,0,0x12345678) != WFIFO_OK)    nrOfErrors++;
  if (WFIFO_WRITE(0,6,0xFFFF0000) != WFIFO_ERROR) nrOfErrors++;
  if (WFIFO_RELEASE_WRITE(0)      != WFIFO_OK)    nrOfErrors++;

  //read
  if (WFIFO_ACQUIRE_READ(1,6 ) != WFIFO_OK) nrOfErrors++;
  if (WFIFO_READ(1,0,temp)     != WFIFO_OK) nrOfErrors++;
  if (temp != 0x12345678)                   nrOfErrors++;
  if (WFIFO_RELEASE_READ(1)    != WFIFO_OK) nrOfErrors++;

  xil_printf("test ended with %d error(s)\n",nrOfErrors);
}
```

### 5.2.4 Example of Performance Test

Performance tests make use of the UART IPs and the recorder IP. A test starts by setting the recorder IP to record mode. Then a code sequence that is of interest from the performance point of view is executed. After that, the recorder is set to play mode. The processor reads line by line from the recorder and writes it out over the UART. The example code below shows a test on the dual-bus architecture. Processor MB_0 writes two items to the WFIFO and MB_1 reads them. The output of the recorder shows the time elapsed. To get the best performance, the test must be complied without assertion.

Code for processor MB_0.

```
/* include files */

int main(void){
  xil_printf("start testcase on mb 0\n");
  OPB_RECORDER_INIT();                          // start recorder

  WFIFO_ACQUIRE_WRITE(0,2);                     // write data
  WFIFO_WRITE(0,0,0x12345678);
  WFIFO_WRITE(0,1,0xFFFFAAAA);
  WFIFO_RELEASE_WRITE(0);

  replayOpbRecorder();                          // write out recorded data
  xil_printf("test ended with 0 error(s)\n"); // signal end of test
}
```

Code for processor MB_1.

```
/* include files */

int main(void){
  int temp = 0;

  xil_printf("start testcase on mb 1\n");
  OPB_RECORDER_INIT();                           //start recorder

  WFIFO_ACQUIRE_READ(0,2);                      // write data
  WFIFO_READ(0,0,temp);
  WFIFO_READ(0,1,temp);
  WFIFO_RELEASE_READ(0);

  replayOpbRecorder();                          // write out recorded data
  xil_printf("test ended with 0 error(s)\n");  // signal end of test
}
```

Output

```
start testcase on mb 0
9  4  // acquire write
15 2  // write
21 2  // write
27 8  // release write
test ended with 0 error(s)
```

```
start testcase on mb 1
33 3  // acquire read
40 1  // read
47 1  // read
54 5  // release read
test ended with 0 error(s)
```

### 5.2.5 Summary

- The testing includes functional tests and performance tests.
- All tests can run on the FPGA or in a Modelsim simulation.
- All tests can run with the same version of the WFIFO IP.
- The results of performance tests are cycle true.

## 5.3 Modular Testing Environment

A single IP test can be done manually, but in practise many tests must be executed or the same test is executed repeatedly which makes testing boring and error-prone. With the modular testing environment that is presented in this section the testing is automated. As indicated by the name, the testing environment is designed with a modular concept. It uses three types of libraries: iplib, syslib and tclib. This concept is shown in the next figures. The testing environment was called iptester.

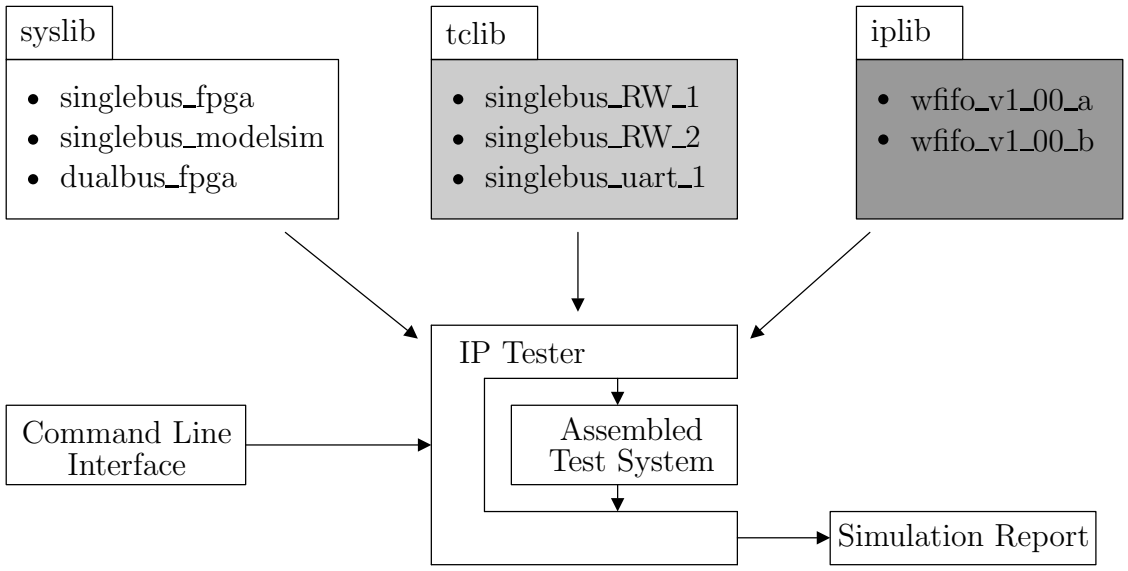



**Figure 5.6:** Libraries used in the iptester

The syslib contains XPS-compliant systems that do not include the test application and the implementation of the IP to test. The missing test application is located in the tclib library and the IP implementation is located in the iplib library. The iptester is started from a command line interface and the configuration for the test is specified with command line arguments. The testing includes the following three steps:

**System Preparation:** The system to test is assembled form the three libraries and the required compilation and generation steps are done. The command line arguments define the system and the test case that are used. It is possible to run multiple tests with one function call. This causes the iptester to repeat the required steps for each test case. For system assembly, the components of the libraries are copied together. For the compilation and generation, the command line interface of XPS is used. If the system is prepared for a Modelsim simulation, the simulation model is generated. If it will run on the FPGA, the bit-stream for FPGA programming is generated.

**Test Execution:** Depending on the simulation type, either Modelsim is started or the simulation is downloaded on the FPGA. During test execution, the iptester waits for the end-of-test signal. A process can signal that it has finished by writing out a line `test ended with .. error(s)`. A command line argument specifies which processors will signal end-of-test. The iptester runs the simulation until all of them have signalled completeness or until a timeout is reached.

**Simulation Report:** From the information read from the COM port and the files generated during execution and preparation, the iptester generates a simulation report. It is stored in a directory named after the current time. It contains one folder for each test case executed that contains detailed information about this test and a text file that summarises the results.

Figures 5.7 and 5.8 show assembled single-bus systems for an FPGA and a Modelsim simulation, respectively. The gray level indicates the source library.

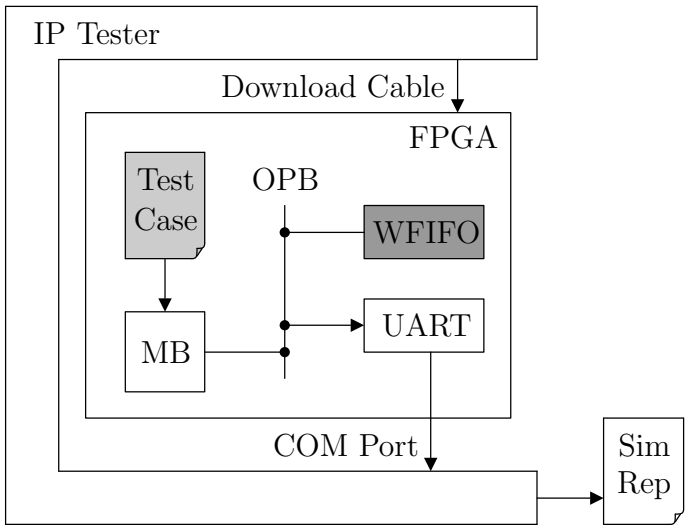



**Figure 5.7:** Assembled system for FPGA simulation and single bus architecture.

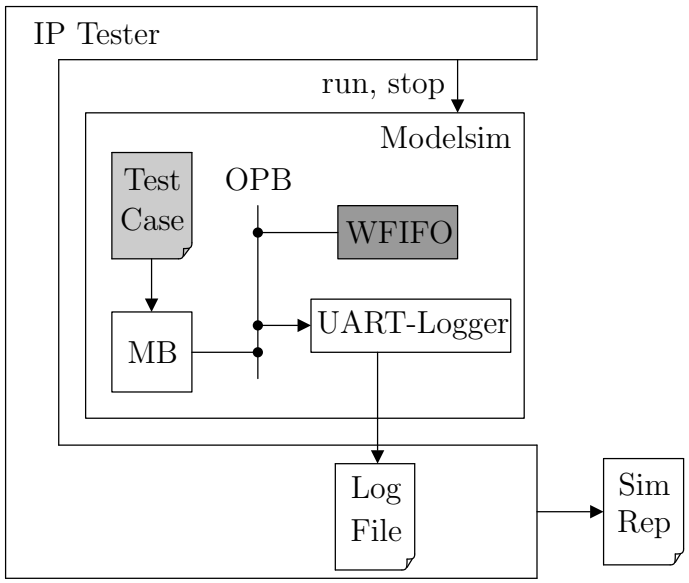



**Figure 5.8:** Assembled system for Modelsim simulation and single bus architecture.

### 5.3.1 Command Line Options

The iptester is implemented with TCL, which is the standard script language for most chip design applications. The script requires ISE, EDK and Modelsim. Modelsim is not required if only FPGA tests are executed. For Windows systems[2], the best way for running the script is to use the normal command line window and to call the TCL interpreter from there. The syntax for such calls is given below. If the TCL command line window is used, the script can be called directly, but all environmental variables of EDK and ISE must be set again.

```
>tclsh84 iptest.tcl parameters
```

#### Parameters

A single parameter consists of a parameter name that starts with a minus sign followed by a space and the value of the parameter.

| Parameter | Function |
|---|---|
| `-sys` | Select the system to simulate. The value is the directory name of the XPS built system. It is located in the syslib directory. |
| `-tc` | Select the test cases to run. The value is a pattern that consists of letters, numbers and stars. All test-cases in the tclib directory that match this pattern will be executed. |
| `-mode` | Set the simulation mode. The possible values are <clean> and <update>. update is used if only the test-cases have been updated and no hardware changes were done. clean must be set after hardware changes. clean takes much longer to run than the update mode for FPGA simulation. For Modelsim simulations the difference is smaller. |
| `-sim` | Set the simulation type. The two possible values are <modelsim> and <fpga>. If fpga is selected, the FPGA board must be connected with the parallel programmer cable and serial cable to read the simulation result. Furthermore, the board must be supplied and switched on ;-). |
| `-assert` | Set the assertion mode. <off> turns assertion off, <blk> enables assertion with blocking, <nonblk> enables non-blocking assertion. |
| `-waitfor` | List of log files or COM ports to read in order to detect the end of simulation. |

**Table 5.2:** Parameters for iptester function call

[2] ISE and EDK are available for Microsoft Windows only.

#### Examples

```
>tclsh84 iptest.tcl -sys singlebus_fpga -waitfor com_1
```

Simulate with single-bus system, use default mode (update), use default simulation type (Modelsim) and run all test-cases.

```
>tclsh84 iptest.tcl -sys dualbus_modelsim -mode update -sim modelsim
                    -tc *WErr* -assert off -waitfor log_0 log_1
```

Simulate with dual-bus system, use mode clean, run the simulation with Modelsim and run all test-cases that match the pattern *WErr* until both processors signal end of test. The listing below shows an example for the test report summary.

```
----------------------------------------------------------------
Summary for simulation run 2006-06-07-17-43-31
Simulation type: fpga
Simulation mode: clean
Used System    : singlebus_fpga
----------------------------------------------------------------
wfifo_WErr_1: test ended with 0 error(s)
wfifo_WErr_2: test ended with 0 error(s)
wfifo_WErr_3: test ended with 0 error(s)
wfifo_WErr_4: test ended with 0 error(s)
----------------------------------------------------------------
```

### 5.3.2 How to Write Test Cases

A single test case consists of one source file named `main.c` for each processor of the target system. They must be stored in the folder tclib as shown in Figure 5.9. The execution of the test case is stopped as soon as all processors specified by the `waitfor` command line option have written out `"test ended with x error(s)"` (see Table 5.2). When writing a new test case, it must be considered that the execution is not stopped before all steps of interest are executed. If none of the processors signals end-of-test, then the execution is stopped after a timeout is reached.

### 5.3.3 How to Add New Systems

New systems can be built with XPS or with the tool for automated system generation called WAB presented in Chapter 6. If the system is designed with XPS, the systems generated with WAB can be used as reference designs. At least one of the processors should be connected to an UART or an UART-Logger in order to signal end-of-test to the testing envrionment. After the design is finished, it must be copied to the syslib directory of the testing evironment. The complete directory structure is shown in Figure 5.9.

### 5.3.4 Directory Structure

| | |
|---|---|
| /tester | iptester home directory. |
| /syslib | Contains all systems used for simulation. |
| /singlebus_fpga | Contains an XPS built system. The directory name consists of the system name followed by the simulation type. |
| /iplib | Contains all IPs that are tested. |
| /wfifo_v1_00_a | WFIFO IP. |
| /opb_ipif_wfifo_v1_00_a | IPIF IP for WFIFO to OPB connection. |
| /tclib | Contains all test cases. |
| /singlebus_wr_1 | Contains a single test case. The directory name starts with the system name of the test case it is written for, followed by a test case name and a number. |
| /mb_no | Source code for MB processor with ID <no>. Smallest number is zero. |
| /scripts | Scripts used by iptester. |
| iptest.tcl | Tester script main file. |
| iptest-toolkit.tcl | Function definitions. |
| /logfiles | All log-files of the last simulation. |
| /simresult | Contains one folder for each simulation run. |
| /2006-06-07-17-43-31 | Contains all results for a test run that was started at the indicated time. |
| summary.log | Contains a summary of all test cases run in this session. |
| /singlebus_RW_1 | Contains all log-files of the indicated test case. |

**Figure 5.9:** Directory structure of the testing environment.

## 5.4 OPB Recorder

The VHDL source code of the OPB-Recorder is given in Appendix C.2. This section is not a complete documentation of the recorder but it explains some points of interest. The API of the OPB-Recorder is shown in Table 5.1.

When using the recorder, it must be configured with design parameters. Table 5.3 lists all parameters required.

| Parameter | Function |
|---|---|
| C_BASEADDR | Recorder base address. |
| C_HIGHADDR | The high address. Minimum address range is 512 bits (0x1FF). |
| C_WFIFO_R_BASE | Base address of the WFIFO read port to record. For the current instruction format, the base address is given by 0x01[read port no]00000. The default is set to 0xFFFFFFFF and must be changed. If only the write port is used, the read port base is set to the same value as the write port base. |
| C_WFIFO_R_HIGH | High address of the WFIFO read port to record. For the current instruction format, the base address is given by 0x01[read port no]FFFFF. The default is set to 0x00000000 and must be changed. If only the write port is used, the read port high address is set to the same value as the write port high address. |
| C_WFIFO_W_BASE | Base address of the WFIFO write port to record. For the current instruction format, the base address is given by 0x01[write port no]00000. The default is set to 0xFFFFFFFF and must be changed. If only the read port is used, the write port base is set to the same value as the read port base. |
| C_WFIFO_W_HIGH | High address of the WFIFO write port to record. For the current instruction format, the base address is given by 0x01[write port no]FFFFF. The default is set to 0x00000000 and must be changed. If only the read port is used, the write port high address is set to the same value as the read port high address. |

**Table 5.3:** OPB recorder design parameters.

The OPB recorder contains logic and a BRAM block. The most interesting part of the logic is the state machine. Figures 5.10 and 5.11 show the states and possible transitions.

The recorder has three system states.

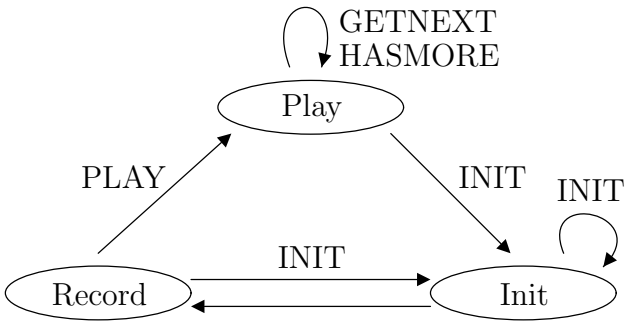



**Figure 5.10:** System states of the recorder

The play state has four sub-states.

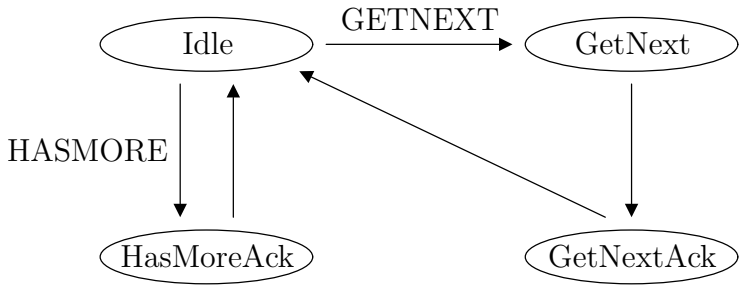



**Figure 5.11:** Play states of the recorder

The recorder encodes the bus transactions with four bits. Table 5.4 shows how the transaction types are encoded.

| Encoding | Command |
|---|---|
| 1 | Read |
| 2 | Write |
| 3 | Acquire Read Window |
| 4 | Acquire Write Window |
| 5 | Release Read Window |
| 6 | Release Write Window |
| 7 | Status OK |
| 8 | Status Error |
| 9 | Status Failed |

**Table 5.4:** Encoding of WFIFO commands by the OPB-Recorder

## 5.5 UART Logger

The UART-Logger is a simulation model for the Xilinx UART lite. It offers the same bus interface as the lite version but it writes the received data to a text file instead of a COM port. The UART-Logger can be used for simulation only. It cannot be synthesised and mapped on an FPGA. Compared to the UART lite, the UART-Logger offers one additional design parameter. It is shown in Table 5.5.

| Parameter | Function |
|---|---|
| C_LOG_FILE_NAME | This parameter takes a string as value that defines the name of the log file. It must be ensured that this name is unique in the system. The default name is uart_logger_1.out. |

**Table 5.5:** UART-Logger design parameter.

# 6 Design Flow Integration and Automation

This chapter shows how the WFIFO and the other IPs designed were integrated in the XPS design flow and how the system design was automated. Beside normal system design, automated design is very useful for generating different architectures for testing. This dependency between testing and design automation is shown in Figure 6.1.

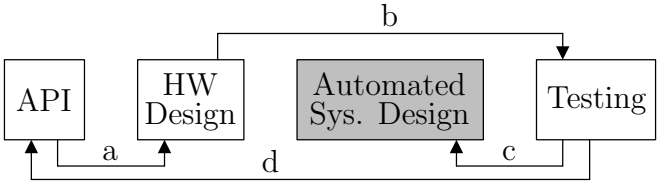



**Figure 6.1:** Overview of practical work.

## 6.1 XPS Design Flow Integration

The application Xilinx Platform Studio (XPS) is used to develop systems based on the Embedded Development Kit (EDK) from Xilinx. XPS allows to design the complete HW/SW system. It includes all the tools for compilation and can generate a simulation model for Modelsim or a bit-stream for FPGA programming. In order to integrate a user defined IP in XPS, two files are required in addition to the VHDL files.

**MPD:** The microprocessor peripheral description file defines the bus interfaces supported by the IP, the required design parameters and the port definitions. It also defines the type of the IP, i.e. normal peripheral or standard output peripheral.

**PAO:** The peripheral analyse order file lists all dependencies for the compilation of the VHDL files. It is required to compile all VHDL files in the right order.

A detailed description of the MPD and PAO file format is given in the Platform Specification Format Reference Manual. The MPD and PAO files of the WFIFO are shown in C.1.5 and C.1.4. How systems are designed with XPS is described in the Platform Studio User Guide. The systems single-bus and dual-bus presented for IP testing can be used as a reference design for WFIFO architectures.

## 6.2 Automated Design-Flow

System design with XPS for multiprocessor systems, as required for WFIFO architectures, is time consuming and error prone. This is why an automated system design was implemented. Very similar work was presented in [4]. The automated design flow presented in this section takes an abstract definition of the system as input and generates a complete XPS system. After the generation, the system can be opened with XPS or used to perform test with the modular testing environment presented in 5.3.

The automated system design is performed in two steps: mapping and architecture generation (see Figure 6.2). Currently, the mapping step is only partially integrated because it is limited to a one-to-one mapping. Its main purpose in the current implementation is to simplify the input file format. The topology file required for the mapping step has a simpler structure than the platform file. Sections 6.2.1 and 6.2.2 describe the mapping and the architecture generation and define the input file formats.

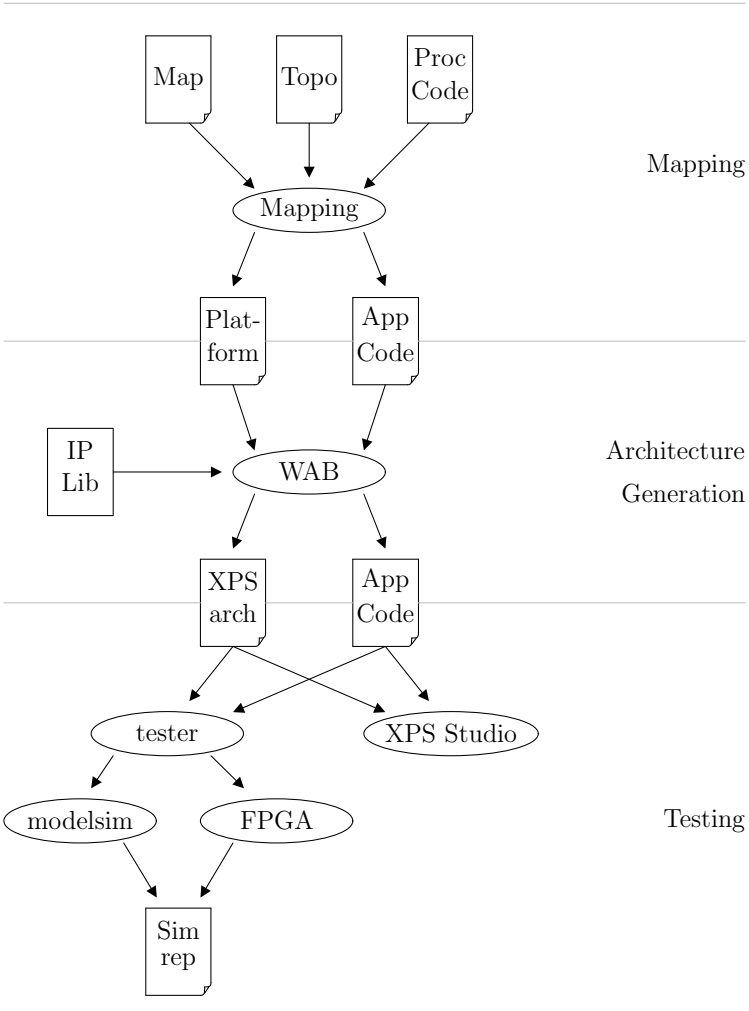



**Figure 6.2:** Overview of the WFIFO architecture design and testing flow.

### 6.2.1 Automated Mapping

The mapping step takes a topology file, a mapping file and one folder for each process containing its application as input. The topology file defines the structure of the WFIFO process network. It defines how many processes and WFIFOs are used and how they are connected and configured. It also defines a unique number for each process and each WFIFO. During the mapping step, port numbers for each WFIFO are generated. The format of the output file is defined in 6.2.2.

Before mapping is done, the port numbers of the WFIFOs are not known. Processes only know the ID of the WFIFO they are connected with. It is not possible to predefine the port numbers before the mapping because if two processes are mapped on the same processor, port number change. Therefore, the application code uses the WFIFO IDs instead of the port numbers. How this looks like is shown in the example code below. Mapping is implemented in a single TCL script named map.tcl.

#### Input Directory Structure

| | |
|---|---|
| topology.txt | Defines the topology oft the WFIFO process network. |
| mapping.txt | Defines the mapping for each process of the network. |
| p_0 | Contains all the source code for process zero. |
| main.c | Source file for process zero. WFIFO IDs are used instead of port numbers. |
| p_1 | Contains all the source code for process one. Similar directories exist for all other processors. |

#### Topology File Format

Each line in the topology file starts with a block name and is followed by a list of parameters and parameter values. The block name defines the parameters that can be used. Table 6.1 defines these parameters. The used WFIFO design parameters are explained in 3.6.

| Block | Parameter | Function and Values |
|---|---|---|
| `p_x` | `-in` | list of WFIFOs that are read by process $x$. If more than one WFIFO is read, they are separated by a space. |
| | `-out` | list of WFIFOs that are written by process $x$. |
| | `-dev` | IO devices used by the process. Possible values are `uartlite_x` and `uartlog_x`. |
| `wfifo_x` | `-mem` | memory size design parameter of wfifo_x. |
| | `-windsize` | window size design parameter of wfifo_x. |
| | `-bramsize` | bram address size design parameter of wfifo_x. |
| | `-pipeline` | pipeline mode design parameter of wfifo_x. |
| `rec_x` | `-read` | block name of the WFIFO read port recorded by rec_x. |
| | `-write` | block name of the WFIFO read port recorded by rec_x. |

**Table 6.1:** Topology file format. $x$ is the unique number of the corresponding block.

Example WFIFO process network.

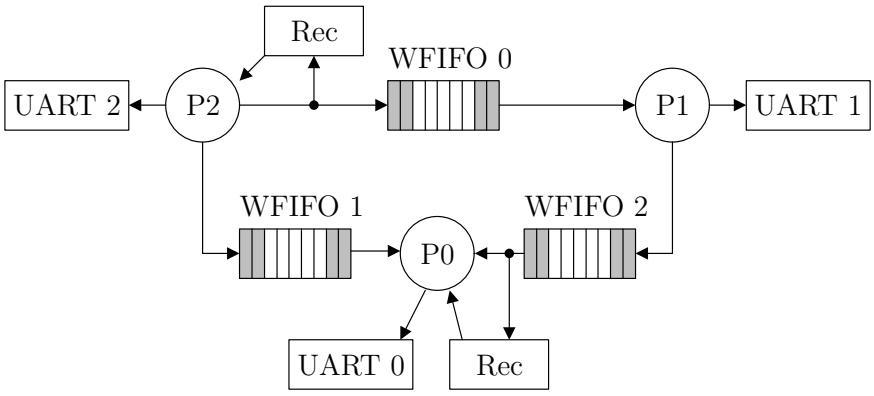



**Figure 6.3:** Example WFIFO process network with output devices and OPB recorders.

Topology file for the process network shown in Figure 6.3.

```
p_0 -in  wfifo_1 wfifo_2 -dev uartlog_0
p_1 -in  wfifo_0 -out wfifo_2 -dev uartlog_1
p_2 -out wfifo_0 wfifo_1 -dev uartlog_2
wfifo_0 -mem 8 -windsize 7 -bramsize 11 -pipeline 7
wfifo_1 -mem 4 -windsize 7 -bramsize 11 -pipeline 7
wfifo_2 -mem 4 -windsize 7 -bramsize 11 -pipeline 7
rec_0 -write wfifo_0
rec_1 -read wfifo_2
```

### Mapping File Format

| Block | Parameter | Function and Values |
|---|---|---|
| `mb_x` | `-proc` | List of all process names that are mapped on Micro Blaze number $x$. |

**Table 6.2:** Mapping file format.

Mapping file example for a one-to-one mapping of a network with three processes.

```
mb_0 -proc p_0
mb_1 -proc p_1
mb_2 -proc p_2
```

### Process Source Code

In the source code shown below, strings like WFIFO_W_2 and WFIFO_R_0 are used instead of the port numbers. The corresponding port numbers are generated during the mapping and saved to wfifo-portmap.h.

```
//*************************************************************/
#include "xparameters.h"    /* generated system parameters    */
#include "xbasic_types.h"   /* basic types for device drivers */
#include "xio.h"            /* bus access                     */
#include "wfifo-level1.h"   /* wfifo level API                */
#include "wfifo-portmap.h"  /* wfifo port to address mapping  */
//*************************************************************/
```

```
int temp, i;

int main(void){
  temp = 0;

  // acquire read and write window
  WFIFO_ACQUIRE_READ( WFIFO_R_0,5);
  WFIFO_ACQUIRE_WRITE(WFIFO_W_2,5);

  // read one item form WFIFO 0, add 10 and write it to WFIFO 2
  for (i=0; i<5; i++) {
    WFIFO_READ(WFIFO_R_0,i,temp);
    temp = temp + 10;
    WFIFO_WRITE(WFIFO_W_2,i,temp);
  }

  // release read and write window
  WFIFO_RELEASE_READ( WFIFO_R_0);
  WFIFO_RELEASE_WRITE(WFIFO_W_2);

  // signal end of test
  xil_printf("# test ended with 0 errors");
}
```

### Command Line Options

| Parameter | Function |
|---|---|
| `-input` | Path of the input directory. This director must include the mapping and topology files and the directories for the application codes as defined above. |
| `-output` | Path of the output directory. The generated output can be used as input of WAB. |

**Table 6.3:** Mapping command line options

```
map -input c:/mapin -output c:/mapout
```

#### Output

The mapping script generates the platform file used by WAB, changes the directory structure for the source files and generates port-map files. The platform file is shown in the next section and the port map file is listed below.

```
#define WFIFO_R_1 0
#define WFIFO_R_2 1

#define WFIFO_R_0 0
#define WFIFO_W_2 1

#define WFIFO_W_0 0
#define WFIFO_W_1 1
```

### 6.2.2 WAB – WFIFO Architecture Builder

The WFIFO Architecture Builder is an application that generates an XPS compliant architecture from an abstract specification. WAB takes as input parameter a directory path. This path must have the following content:

| | |
|---|---|
| platform.txt | Defines the architecture. The format is explained below. |
| /mb_0 | Contains all the source code for processor zero. |
| main.c | Source code file. |
| /mb_1 | Contains all the source code for processor one. Similar directories exist for all other processors. |

XPS stores the information about the architecture and the used software in four files. To automate the architecture generation, these files must be generated. Below, a short summary of these files is given. A detailed description is given in the XPS manual.

**MHS:** The Microprocessor Hardware Specification file defines all hardware components of the system with the required design parameters and signal connections.

**MSS:** The Microprocessor Software Specification file defines software parameter such as the standard output peripheral or the used compilers for the processors.

**XMP:** The Xilinx Microprocessor Project file contains all required paths such as the path to the application source files for each processor and some target platform and system dependant settings.

**UCF:** The User Constraint file defines timing constraints and pin connections for the FPGA board.

### Platform File Format

Each line in the platform file starts with a block name and is followed by a list of parameters and parameter values. The block defines which parameters can be used. Table 6.4 defines theses parameters.

| Block | Parameter | Function and Values |
|---|---|---|
| `mb_x` | -bus | name of the bus for Micro Blaze $x$ for example `opb_0`. |
| | -dev | devices connected to mb_x. Possible values are `uartlite_x` and `uartlog_x`. |
| `wfifo_x` | `-rport` | read port number. |
| | `-wport` | write port number. |
| | `-rbus` | read bus number. |
| | `-wbus` | write bus number. |
| | `-mem` | memory size design parameter. |
| | `-windsize` | window size design parameter. |
| | `-bramsize` | bram address size design parameter. |
| | `-pipeline` | pipeline mode design parameter. |
| `rec_x` | `-bus` | bus number. |
| | `-rport` | read port number. |
| | `-wport` | write port number. |

**Table 6.4:** Platform file format. $x$ is the unique block number.

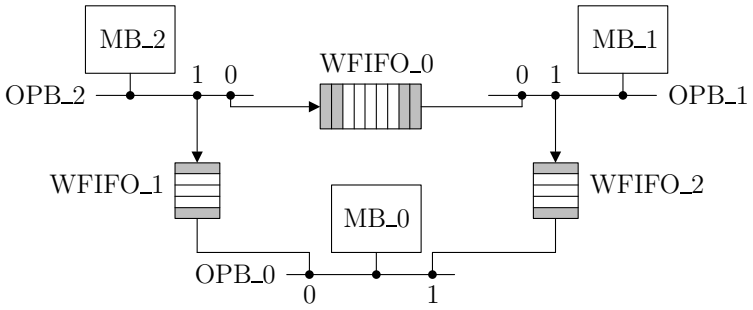



**Figure 6.4:** Example architecture for platform definition. Recorder and UART devices are not shown.

Platform file for the architecture shown in Figure 6.4.

```
mb_0 -bus opb_0 -dev uartlog_0
mb_1 -bus opb_1 -dev uartlog_1
mb_2 -bus opb_2 -dev uartlog_2
wfifo_1 -rport 0 -wport 1 -rbus 0 -wbus 2 -mem 4 -windsize 7
        -bramsize 11 -pipeline 7
wfifo_2 -rport 1 -wport 1 -rbus 0 -wbus 1 -mem 4 -windsize 7
        -bramsize 11 -pipeline 7
wfifo_0 -rport 0 -wport 0 -rbus 1 -wbus 2 -mem 8 -windsize 7
        -bramsize 11 -pipeline 7
rec_0 -bus 2 -rport 0 -wport 0
rec_1 -bus 0 -rport 1 -wport 1
```

### WAB Script Structure

The WAB application includes a TCL script and a set of templates. The templates are used to create the XPS files listed above. The code below shows the UART template for the MSS file. This template is sourced by the TCL script. The strings that start with a Dollar-sign are TCL variables and get replace with the correct value. This implementation with TCL script and templates makes it easy to update WAB for new IP versions.

```
set template "
BEGIN DRIVER
 PARAMETER DRIVER_NAME = uartlite
 PARAMETER DRIVER_VER = 1.00.b
 PARAMETER HW_INSTANCE = RS232_Uart_\$uartNo
END
"
```

### WAB Command Line Options

Table 6.5 lists the command line options for WAB. An example is given below.

| Parameter | Function |
|---|---|
| `-input` | Path of the input directory |
| `-output` | Path of the output directory |
| `-style` | Possible styles are <XST> and <tester>. With option XST, a normal XPS compliant system is created. With option tester, the source code and the architecture are separated in a syslib and a tclib directory. This is the file structure used by the modular testing environment (iptester). |
| `-arch` | This parameter is only used if style is set to tester. It defines the name of the resulting architecture (e.g. singlebus_modelsim_3). |
| `-code` | This parameter is only used if style is set to tester. It defines the name of the test-case. |

**Table 6.5:** WAB command line options.

Command Line Examples:

```
wab.tcl -input C:/wabin/test_1 -output C:/wabout/test_1 -style XST

wab.tcl -input C:/wabin/test_1 -output C:/wabout/test_1 -style tester
            -arch dualbus_fpga_1 -code dualbus_mir_1
```

# 7 WFIFO Theory

In this chapter, aspects of WFIFO theory are discussed. The first section shows that with certain restrictions, the WFIFO process network can be considered as a subset of KPN. The subsequent sections discuss the calculation of the minimum memory requirements for communication channels in general and for WFIFO channels in particular.

**Disclaimer:** Since the WFIFO is a new buffer type, most concepts presented in this chapter are based on my own ideas. It is possible that I have not chosen the best approaches to solve the problems addressed, but I think it is a first step for the theory on WFIFO channels. I think that the presented ideas are right but I was not able to verify them completely. I also present some problems that I could not solve completely but I show the reflections I made.

## 7.1 KPN Compliance

The reason why the KPN is so popular to model signal processing algorithms is that it offers a set of favourable properties. The most important one for signal processing application is that the KPN model is determinate, which means that the input/output relation is independent from the timing of the processes and therefore computation delay insensitive. These conditions make it interesting to investigate whether a process network with WFIFOs is compliant with the KPN model of computation. This section shows that with some restrictions, the WFIFO process network is a subset of Kahn process networks. A KPN compliant process network must meet the following conditions [5]:

1. The processes can only communicate over FIFO buffers with blocking read and unlimited size.
2. A communication line transmits information within an unpredictable but finite amount of time.
3. Every process follows a sequential program.
4. At any time, a process is either computing or waiting for information on one of its input ports.

In case a WFIFO network makes use of non-blocking acquiring operations, it is not KPN compliant. For non-blocking acquire-read operations this is obvious because it is possible to implement a process that waits for data on two or more of its input lines, which is a contradiction to condition four. For the non-blocking write operation, it is assumed that the used FIFOs are of limited size. With a process that uses the non-blocking write instruction to write data to the first buffer that is not full, it is possible to implement a non-determinate system. It will be shown later that it is possible to realise FIFO buffers of limited size from FIFO buffers of unlimited size.

The following statement will be shown:

> WFIFO process networks are a subset of Kahn process networks if all acquiring operations are executed with blocking.

To proof this it must be shown that it is possible to realise a WFIFO process network with a Kahn process network. Two statements must be shown to be true:

- It is possible to realise the WFIFO's functionality with a process network that uses only FIFO buffers of limited size.
- It is possible to realise FIFO buffers of limited size from FIFO buffers of unlimited size.

The second point addressing the limited buffer size was discussed in [2]. It was shown how a buffer with rendezvous protocol can be implemented with FIFOs of unlimited size. Here a similar proof is made for FIFOs of limited size that is not restricted to the rendezvous protocol. We start with a FIFO process network with buffers of limited size. Each of these buffers $B_i$ with memory sizes $S_i$ is replaced by two FIFOs with blocking read and unlimited size. The first one has the same direction as $B_i$, the second one goes in the opposite direction. The second FIFO initially contains $S_i$ data items if $B_i$ is empty at reset. If $B_i$ contains $x$ items at reset, the second FIFO is initialised with $S_i - x$ items.

The channel access is changed as follows: The writing process always reads an item from the backward channel before writing the data to the forward channel. The reading process writes an item to the backward channel immediately after reading an item from the forward channel. With the backward channel it is warranted that there are never more than $S_i$ data items in the forward channel, which is equivalent to a FIFO buffer of limited size.

For the first point from above, it must be shown that it is possible to implement the WFIFO's API using a FIFO buffer of limited size. When the API receives an acquire-write instruction, a local memory segment with the specified window size is allocated. Subsequent write instructions are redirected to this local memory. When receiving a release instruction, all data from the memory segment are written to the FIFO buffer. When receiving an acquire-read instruction, the API reads as many data as specified by the window size and writes them to a local memory. Subsequent read operations are redirected to this local memory. When receiving a release operation, the memory segment is removed. This description shows that the implementation is possible.

## 7.2 A Model for Data Transport in Communication Channels

This section presents a model to describe the data transport in a communication channel. In the following, a single channel is examined that is connected with two different processes.

The writing process is called producer, the reading process is called consumer. The channel is not restricted to FIFO channels. It allows multiple read of the same data and reordering. There are different possibilities to specify a data transport sequence on such a channel. A common option is to use a code sequence as shown in Example 7.1, where variable $A$ indicates the channel. In this section, a more general approach is presented.

To make a comparison of the order data is produced with the order it is consumed, rank values for each data item $i$ transferred over the channel are defined.[1] There are two kinds of rank values: production rank values $\text{rank}_P(i)$ and consumption rank values $\text{rank}_C(i)$. The value of $\text{rank}_P(i)$ expresses how many items have been produced up to item $i$, whereas $\text{rank}_C(i)$ expresses how many items have been consumed up to item $i$. It is assumed that all items are written once, and read once or more times. This results in a situation where each item $i$ has one $\text{rank}_P(i)$ value and one or more $\text{rank}_C(i)$ values.

For minimum memory calculations, only the relation from production to consumption is of interest and not the item itself. This is why a more general concept is introduced:

**Consumption Sequence:** The consumption sequence $C(t)$ is a function from $\text{rank}_C(\cdot)$ to $\text{rank}_P(\cdot)$. For each consumer rank value the producer rank value that refers to the same data item is returned.

**Production Sequence:** The production sequence $P(t)$ is a function from $\text{rank}_P(\cdot)$ to $\text{rank}_C(\cdot)$. For each production rank value the highest consumption rank value that refers to the same data item is returned.

The production and consumption sequences allow to describe the relation between production and consumption on an abstract level. It is not limited to program code-like channel definitions. Below, an example is given.

### Example 7.1

```
// producer
for (i=0; i<4; i++) {
  for (j=0; j<3; j++) {
    write(A[i][j]);
  }
}
```

```
// consumer
for (i=0; i<3; i+=2) {
  for (j=0; j<2; j++) {
    read(A[i][j]);
    read(A[i+1][j]);
    read(A[i][j+1]);
    read(A[i+1][j+1]);
  }
}
```

[1] A very similar concept was introduced in [9].

| $t$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $C(t)$ | 1 | 4 | 2 | 5 | 2 | 5 | 3 | 6 | 7 | 10 | 8 | 11 | 8 | 11 | 9 | 12 |

| $t$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P(t)$ | 1 | 5 | 7 | 2 | 6 | 8 | 9 | 13 | 15 | 10 | 14 | 16 |

### 7.2.1 Classifications of Communications Channels

The communication between a producer and a consumer process can be classified under the aspects of multiplicity and reordering. If the consumer wants to read the same data item more than once, it is said that the communication uses multiplicity. If the data are read by the consumer in a different order than written by the producer, then the communication needs reordering. Paper [10] introduces a classification of communication types based on the mapping matrix. The mapping matrix can be derived from polytopes defined by the read and write sequences. A simple example is given in [12].

The consumption sequence defined in this section simplifies the classification of communications channels. Table 7.1 summarises the naming convention introduced in [10] that is used to classify channels.

| | Without Multiplicity | With Multiplicity |
|---|---|---|
| Without Reordering | IOM– | IOM+ |
| With Reordering | OOM– | OOM+ |

**Table 7.1:** Naming convention for channel classification.

**Multiplicity:** A channel uses multiplicity if there are any two entries in the consumption sequence of equal value:

$$\exists\, m, n \,|\, C(m) = C(n), \quad m, n \in \mathrm{rank}_C(\cdot).$$

**Reordering:** A channel uses reordering if the consumption sequence is not monotonic increasing:

$$\exists\, t \,|\, C(t) > C(t+1), \quad t \in \mathrm{rank}_C(\cdot).$$

## 7.3 Classification of Implementation Alternatives

This section shows implementation alternatives for channels with multiple read and reordering. From the hardware point of view, there are three possibilities:

1. **Reordering Memory:** The communication channel is realised with a single FIFO and the local memory of the process is used to implement the reordering and the multiple

read. From the functional perspective, it does not matter if this memory is located in the producer or in the consumer. To accelerate memory access, special address generation units (AGU) can be used.

2. **Multiple FIFOs:** The communication channel is realised with a set of FIFOs. The reordering is implemented by a redirection of the read/write operations to one of the FIFOs. A local FIFO with both read and write ports connected to the consumer process is used to store items if they are used later on (multiplicity).

3. **Extended Semantics:** The FIFO buffer is replaced by a new buffer type that allows reordering and multiple reading. This approach is used for the implemented WFIFO.

Different implementations to solve the reordering and multiplicity problem were presented in the context of the Compaan/Laura tool-chain. Paper [11] gives an overview. Most of the presented solutions are of the Reordering-Memory type. A comparison is made in Chapter 8.

## 7.4 Minimum Memory Size for Channels with Reordering Memory

In this section, it is shown how the minimum memory requirements for a channel with reordering memory can be calculated from given production and consumption sequences. It is assumed that random access to the memory is possible and that the items transferred over the channel and the memory cells have equal size. What is addressed in this section is the minimum amount of total memory. It does not matter how this memory is distributed over the different locations such as channel memory or local memory. In the following, it is assumed that all data written to the channel are read at least once.

The calculation of the minimum memory size is based on the observation that a data item cannot be removed from the memory before it is read for the last time. As soon as it is read for the last time, it can be removed immediately. Equation 7.1 shows the calculation of the minimum memory size that is based on this idea. The calculation is made from the consumers point of view. For a given consumption rank value $k$, the function $L(k)$ defined in Equation 7.2 indicates if the item consumed at $k$ is consumed for the last time. If it is the last consumption, $L(k)$ is equal to one, else it is zero. If $k$ is not in $\text{rank}_C(\cdot)$, $L(k)$ is also zero. $L(k)$ is computed from $P(x)$, which by definition only contains the consumption rank values of the last consumption . The cumulated sum of $L(k)$ in equation 7.1 is the total number of items that can be removed before item $t$ is read. The minimum memory size is the maximum difference of the number of items produced to the number of items that can be removed. After $t$ read operations, at least $C(t)$ items have been produced because each item is produced only once. But it is possible that there is a $t_s < t$ with $C(t_s) > C(t)$. For this case, at least $C(t_s)$ items have been produced after $t$ read operations. Nevertheless, equation 7.1 is also right in this case. The reason is that if $C(t_s) > C(t)$ for $t_s < t$, then the maximum

difference is not at $t$ because the cumulated sum over $L(k)$ is monotonic increasing. For Example 7.1, the minimum memory size is three.

$$\mathrm{Mem}_{\min} = \max\left(C(t) - \sum_{k=0}^{t-1} L(k)\right), t \in \mathrm{rank}_C(\cdot) \tag{7.1}$$

$$L(k) = \begin{cases} 1 & \exists\, x : P(x) = k \\ 0 & \text{else} \end{cases} \tag{7.2}$$

## 7.5 Minimum Memory Size for WFIFO Channels

Calculation of the minimum memory requirement for WFIFO channels differs from the calculation for channels with reordering memory. The WFIFO channel does not allow to remove items selectively because only entire windows can be released. Furthermore, reordering is limited to the windows and cannot be done on the entire memory as is it the case for an implementation with reordering memory. For the following minimum memory calculation, it is assumed that the processes only use the WFIFO for storing data and that no local memory is used.

For calculating the memory requirements, the concept of minimum item-life-time is introduced. This minimum life-time shows how long an item has to stay in the memory after the production. What is known is that each item is written to the channel at production time. The question is how long this data item has to stay in memory after that. The channel used for this concept allows random access to the channel memory. There are two conditions to consider:

**Condition 1:** A data item cannot be removed from memory until it is consumed for the last time.

**Condition 2:** A data item cannot be removed from memory before all items that are read earlier have been produced.

Equation 7.3 shows how the item life-time can be calculated from consumption and production sequence. The value of $\mathrm{life}_n(t)$, $t, n \in \mathrm{rank}_C(\cdot)$ is equal to one during the time the $n$th item produced has to stay in the memory and zero elsewhere. At production time the item has to be in the memory, even if it is consumed immediately. This is taken into account with the condition $t = n$. After that, it stays in the memory until the two conditions from above are met. Condition 2 is represented by $C(j)$ and condition 1 is represented by $P(n)$.

$$\mathrm{life}_n(t) = \begin{cases} 1 & t = n \vee n < t \leq \max\big(C(j)\big), \\ & \quad j \in [1, 2, \ldots, (P(n) - 1)] \\ 0 & \text{else} \end{cases} \tag{7.3}$$

Figure 7.1 shows the life-time for the producer/consumer example from 7.2. At the positions indicated with a cross, the item can be removed and at positions with a dot, it is written to the memory. If there is no dot, the item can be removed immediately after production.

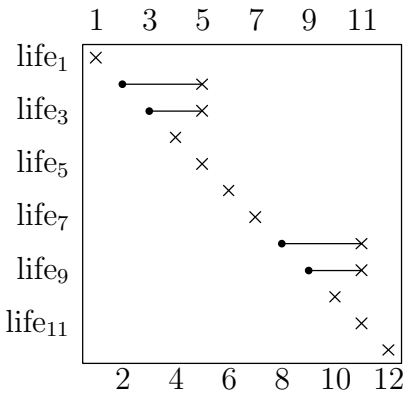



**Figure 7.1:** Life-time for the P/C pair from Example 7.1.

The item-life-time cannot be used directly for the calculation of the memory requirements of a WFIFO implementation, because the WFIFO does not allow random access to the channel memory. How this calculation can be done is shown in the next sections.

### 7.5.1 Channels with Equal Read and Write Windows

For the calculation of the minimum memory requirements, a step by step approach is chosen. First the problem is discussed for a situation where read and write windows are equal. Here equal means that read and write windows have the same position and size relating to the data. All items that are at the first position within a write window are also at the first position in a read window. The same thing holds for the last items. Figure 7.2 shows a data stream with read and write windows of equal size.

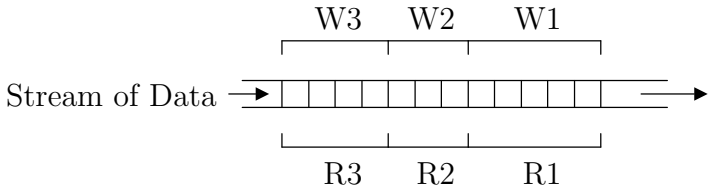



**Figure 7.2:** Data stream with read and write windows of equal size.

The minimum memory size is equal to the biggest read/write window. This can be seen from the fact that it is always possible to acquire the next read window after a write window was released. After the reading is finished and the read window is released, the channel is empty if no write window was acquired in the meantime. It would be possible to already acquire the next write window, but for a scheduling that requires the minimum memory, the producer

waits until all data are removed from the buffer before acquiring the next write window. This shows that boundaries of read and write windows must be in a position within the data stream where it is *possible* that the channel memory is empty. When setting boundaries at other locations, this will result in a loss of data. To identify positions where it is possible to have boundaries, the minimum fill level is defined:

$$\mathrm{Fill}(t) = \sum_{n=1}^{\max(\mathrm{rank}_P(\cdot))} \mathrm{life}_n(t),\ t \in \mathrm{rank}_P(\cdot) \tag{7.4}$$

Equation 7.4 defines the minimum fill level for the memory after $t$ write operations. It is calculated from the item-life-time by adding up all items that are in the buffer at the same time. Now it is possible to calculate the minimum memory requirements for the case of equal read and write windows: If a new read/write window is started whenever this is possible, the minimum amount of memory is used. The size of the required memory is equal to the size of the biggest window.

> A window can be started before the first item in the production sequence or before every production $t$ with $\mathrm{Fill}(t) = 1$. A window can end at the end of the production sequence or before every production $t$ with $\mathrm{Fill}(t) = 1$. The minimum memory size for a channel with equal read and write windows is given by the length of the longest sequence in $\mathrm{Fill}(t)$, $t = \{1, 2 \ldots \max(\mathrm{rank}_P(\cdot))\}$ without a one.

The fill level can also be used to calculate the minimum memory requirements for a channel with reordering memory. The required memory size is equal to the maximum fill level. For Example 7.1, the fill level is shown below. The minimum memory size for a WFIFO channel with equal read and write windows is four.

| $t$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathrm{Fill}(t)$ | 1 | 1 | 2 | 3 | 3 | 1 | 1 | 1 | 2 | 3 | 3 | 1 |

## 7.5.2 Channels with Non-Overlapping Windows

Figure 7.3 shows an example of non-overlapping windows. The minimum memory requirements of non-overlapping windows is equal to the requirements for the case of equal read-/write windows discussed above. When starting with the solution for equal read and write windows that uses the smallest windows possible, the non-overlapping structure allows to add additional boundaries to a read window *or* the corresponding write window but not to both of them. Adding new boundaries at different positions in both windows of a read-/write window pair violates the structure. This case is discussed in the next section. Adding new boundaries at the same position in a read/write window pair is not feasible because all possible boundaries of this type are already included. Because of this, a non-overlapping structure requires the same amount of memory as the corresponding structure with equal read and write windows. Adding new boundaries only in one window of a read/write window

pair does not change the minimum memory size because it does not changes the size of the biggest window.

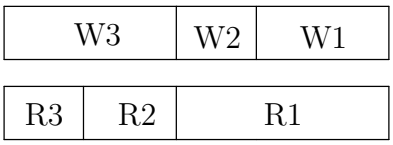



**Figure 7.3:** Non-overlapping read and write windows.

### 7.5.3 Channels with a Brick Wall Window Structure

**Definition:** The brick wall structure for read/write windows has the following properties: each read window must not use data from more than two write windows and in addition it must not contain more than one write window completely. Same thing holds for the write windows. Figure 7.4 shows an example of a brick wall window structure.

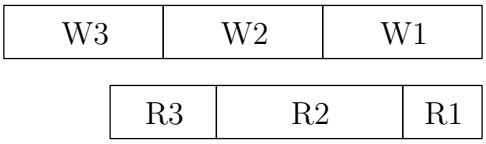



**Figure 7.4:** Read and write windows with a brick wall structure.

The brick wall structure is the most general structure for calculating the minimum memory size. All other structures of interest are a subset of the brick wall structure. This can be seen when examining the structure shown in Figure 7.5. In this example, R2 uses data from three different write windows, which is not allowed for brick wall structures. In the structure shown, R1 can be acquired after W1 is released. But before R2 can be acquired, both W2 and W3 must be released. This shows that W2 and W3 can be replaced with a single read window without changing the required memory size, which results in a correct brick wall structure.

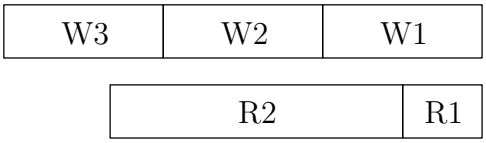



**Figure 7.5:** Other read and write window structure.

With the brick wall structure, it is possible to shift items in the data stream over a longer distance than the window size. This is shown in Figure 7.6. The longest distance an item

can be shifted is from the first position in W1 to the last position in R2. The number of items that can be shifted is limited by the size of the overlap.

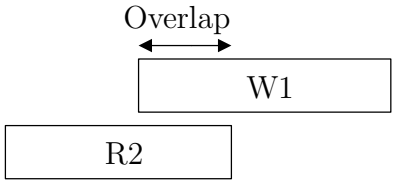



**Figure 7.6:** Data stream with read and write windows of equal size.

The following observations refer to Figure 7.6.

**Observation 1:** The overlap must contain all data that cannot be consumed as soon as W1 is released. This means that the overlap at $t$ contains all data that are in the memory at $t$ and that are not removed then. This can be calculated from the item life-time.

**Observation 2:** All items in the overlap must be consumed within R2. This defines the minimum length of R2 and also the minimum length of W2. For calculating the minimum length, the number of items that must be written to the channel until the overlap can be read completely must be known. How this can be calculated from the life-time is shown in Equation 7.5.

**Observation 3:** The minimum amount of memory that is required if a write window is released at $t$ is the sum of the size of the next write window plus the size of the overlap from the preceding write window. The biggest read window can have equal size but it cannot be bigger than the sum of overlap and write window.

The function next() returns the minimum size of the next write window if a write window is released after item $t$ was written to the memory. $n$ includes all items that are in the memory at $t$. From these items, the maximum function searches the item that is removed last. The distance between this last removal and $t$ is the minimum size of the next write window.

$$\text{next}(t) = t - \max\Big(t_i \,|\, \text{life}_n(t_i) = 1\Big), \quad \forall n \,|\, \text{life}_n(t) = 1 \tag{7.5}$$

With these observations, it is possible to calculate for each item written to the channel the cost for releasing a write window immediately after this item. The costs are calculated from the size of the overlap and from the minimum size of the next write window.

**Problem 1:** Only the minimum size for the next write window is known. In certain situations it is better to use a window bigger than the minimum size because this can lead to smaller windows later in the data transfer. An algorithm to calculate the minimum memory size also has to check windows with bigger size than the minimum value.

**Problem 2:** Multiple read was not considered so far. If an item is consumed multiple times, it is not possible to release a read window between the first and the last consumption.

## 7.6 Non-Blocking Acquiring and Determinism

The WFIFO API offers a blocking and a non-blocking version of the acquire window command. In section 7.1 it was shown that a WFIFO channel without blocking is not compliant with the Kahn model of computation. One of the main reasons to use this model is that all specified systems are determinate: for any fixed input sequence, the system always generates the same output sequence. But it is not a necessary condition to be KPN compliant in order to get a determinate system.

When using non-blocking acquiring, it is the designer's responsibility that the resulting system is determinate. In the example below, a determinate system is presented that uses non-blocking acquiring. With the non-blocking operations, data throughput could be increased considerably.

### Example 7.2

The system in Figure 7.7 has three input channels namely, $a$, $b$ and $c$, which receive data in an unsteady manner. Never, more data are written to the channels than the process P is able to consume. On average, each channel receives the same amount of data. The task of process P on channel $a$ is to read four items, calculate the sum of them and write the result to $a'$. The same must be done for the other channels.

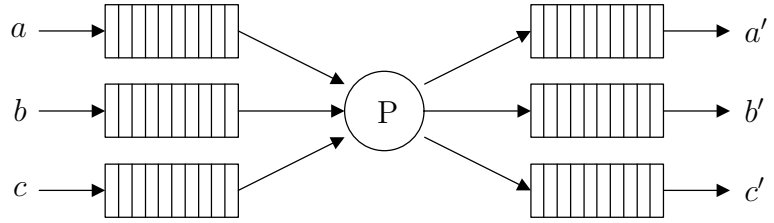



**Figure 7.7:** Example architecture for non-blocking read.

The advantage of using the non-blocking acquiring function is that process P can check if there are already four items in the input buffer. If not, it checks the next buffer. A blocking implementation has to decide in advance which channel to read, which leads to a success rate of $\frac{1}{3}$. The important thing is that both implementations are determinate.

# 8 WFIFO Compared with Other Approaches

In this chapter, the WFIFO concept is compared with two other approaches that address the limits of KPN architectures.

## 8.1 Compared Concepts

The concepts used for the comparison both use a reordering memory to implement multiple read and reordering. Various concepts have been presented for Compaan tool-chain. They mainly differ in the address generation concept and the used memory type. Two versions are discussed here: The Segment realisation and the CAM realisation (see [11] [15]). A short summary is given below.

### 8.1.1 CAM

CAM is a special memory type. A user defined key is used for data read access instead of an address. This key is defined during the write operation. The location where data are written is not known and the only way to recover the data is to make a read with the same key as used for writing. The procedure is as follows: The producer writes its data to a FIFO buffer. From there, they are written to the CAM memory by a storing process. Usually, the storing process is part of the consumer process and is executed whenever a read operation on the CAM fails. The key for storing the data is either provided by the producer or, in the case where the production sequence is known in previous, generate by the storing process itself. For reading, the consumer has to know the right keys. How data are removed from the CAM memory is discussed in 8.5.

### 8.1.2 Segment

The Segment realisation is similar to the CAM realisation. Also here data are written to a FIFO and transfered to a memory but Segment realisation uses normal memory instead of CAM. The producer and the consumer agree on the coordinates of data items within the iteration domain. In a write operation, the coordinates and data are both saved. A special memory is used to save the coordinates called the segment memory. This memory makes a simple compression of the coordinate data. For reading, the consumer calculates the memory address of the item it wants to read from the coordinates of the item and the information stored in the segment memory. No mechanism to remove data from the memory was presented.

## 8.2 Motivation and Origin

Although the concepts compared are all motivated by the limitations of KPN they have different origins. CAM and Segment realisation have been presented in connection with the Compaan tool-chain ([7],[8]). This tool-chain transforms sequential program code into a process network. The program code must be written in Matlab with a nested loop structure. This is a very powerful subset of the Matlab language and many algorithms can be written in this form. In the Matlab code, array variables of one or more dimension are used as a data buffer. Compaan replaces each of this variables by a FIFO buffer. In order to avoid problems caused by the reuse of the same variable the Matlab code is first translated in single assignment form ([6]). In this context, different realisations have been presented to implement algorithms that use out of order data transfer and multiplicity.

The WFIFO realisation was developed to offer a simple interface for the communication in on chip multiprocessor systems. Such architectures are used for data stream based systems in multimedia and other applications asking for high computing power. The interface was requested to support out of order communication, multiple read and skipping in a way that this is comprehensible from the program code. The focus was not to automate parallelisation. This is done manually during the design process.

## 8.3 Hardware

The difference between the compared concepts also manifests in the required hardware. The simplified block diagrams below give an impression of the variety. In the block diagram of the reordering memory, there is a connection between the producer and the logic. For a fixed production sequence, this connection can be removed because the logic can generate this information itself. It is also possible to implement the logic block in software, usually on the cost of speed.

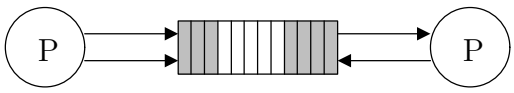



**Figure 8.1:** WFIFO

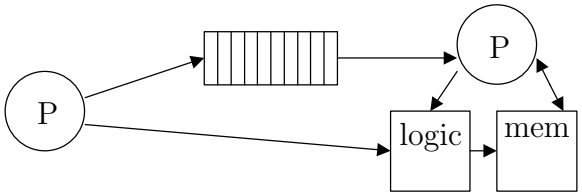



**Figure 8.2:** Reordering Memory

## 8.4 Memory Usage

In this section the memory requirements of the different channel implementations are compared. The comparison is done on an abstract level, independent from the implemented algorithm.

### CAM

The structure of the CAM memory allows to individually define the duration which a data item stays in the memory. All items that are not required any more can be removed immediately and selectively. Therefore it is possible to implement a channel with the smallest possible memory size as defined by Equation 7.1. However, there are situation where a memory bigger than the minimum size is chosen in order to optimise throughput or latency.

### Segment

The Segment realisation requires a memory as big as the iteration domain. The difference between the size of the iteration domain and the minimum memory size depends on the access sequence and can not be expressed by fixed ratio. If the iteration domain has size $N$, then the ratio to the minimum memory size is defined in the interval:

$$\frac{\text{size}_{\text{Itr}}}{\text{size}_{\text{Min}}} \in [1, \ldots, N] \tag{8.1}$$

In the best case, the Segment realisation needs as much memory as the CAM realisation. In the worst case, it uses $N$ times more.

### WFIFO

The memory size of the WFIFO solution is between the CAM and the Sequential solution. The reason why WFIFO uses more memory than the CAM realisation is that it is not possible remove items selectively from the memory. Deleting occurs when the read window is released and affects all data within this window.

## 8.5 API Concept

In this section the API concepts are compared. It must be stressed that the shown APIs are only correct from the conceptual point of view. An implemented API may look different. The syntax of an operation only shows which parameter are required in principle. It is possible that these parameters are not encoded in program code, but delivered from hardware. A parameter only means that the corresponding information must be known somehow by the channel. The parameter `port` identifies the buffer in case that there is more than one buffer connected to the same process.

### CAM

The producer and the consumer process can agree on an arbitrary key to identify a certain data item. The implementation presented in [15] uses the coordinates of the iteration point for key generation. This computation can be implemented in hardware and is therefore fast. CAM read and write commands:

```
write(Port,Data,Key,Multiplicity)
read(Port,Key)
```

Different versions of the CAM write function have been presented – once with a multiplicity parameter [11] and once without [15]. The version without this parameter does not support multiplicity at all and uses therefore less memory.

The consumer is free to choose the read sequence but the number of read operation on a data item is fixed. The consumer is responsible to read each item as many times as expected. If an item is read less often, it stays in memory forever which may lead to a lack of memory. Another thing that must be considered is that the calculation of the required memory size depends on the access sequence. If the consumer makes a read access that would require a bigger memory, the system will end in dead lock.

```
for (int i=1; i<=N; i++) {
  for (int j=M; j<0; j--) {
    tKeq = getKey(i,j);
    read(port,tKey);
}
```

### Segment

For writing and reading data to/from the reordering memory the coordinates of the data in the iteration domain must be known. Usually this information is located in the iteration variables of the loops.

```
write(Port,Data,Coordinates)
read(Port,Coordinates)
```

Since the memory is as big as the iteration domain, it is possible to read any data item written to the memory in an arbitrary order and as often as required. No limits for reordering and multiplicity exist. Data items are identified by their coordinates. The producer and the consumer have to agree on these coordinates. Usually this is done at compile time but it is also possible to implement the coordinate calculation data dependent. Then the agreement on the coordinates is made at runtime.

```
for (int i=1; i<=N; i++) {
  for (int j=M; j<0; j--) {
    read(port,i,j);
}
```

### WFIFO

The API of the WFIFO offers four extra instructions for acquiring and releasing a window.

```
acquireRead(Port,Size)
releaseRead(Port)
acquireWrite(Port,Size)
releaseWrite(Port)
write(Data,Port,Offset)
read(Port,Offset)
```

The commands can not be used in an arbitrary order. The way they can be used is defined by a protocol described in 2.1.1 The process has unlimited access to all items in a window. But when releasing a write window it is the responsibility of the process that no further write operations in this segment are required. The same holds for read operations. It is not possible to reacquire the same type of window over the same data after releasing. Therefore, the main difference is that the process is also responsible to acquire the windows and to choose the right window size.

```
for (int i=1; i<=N; i++) {
  acquireRead(port,M);
  for (int j=M; j<0; j--) {
    read(port,i);
  }
  releaseRead(port);
}
```

## 8.6 Latency

The latency addressed in this section is the time from writing a data item to the channel until it *can* be read. The analysis of the latency is done in a qualitative manner. It is not measured in clock cycles but in execution steps. The comparison is focused on the question what has to be done between read and write.

In CAM and Segment realisation, a data item can be read as soon as it is written to the memory. The latency between reading and writing depends on how the data is transfered from the FIFO to the memory. In the implementation presented in [11], data are transferred from the FIFO to the memory by the consumer processor. Transfer starts as soon as a read operation fails because the wanted data is not in the memory and ends when this data is written to the memory. With this implementation latency depends on the read sequence and is somewhere between zero and the minimum memory size.

For the CAM realisation, an implementation was proposed in [15] that works without a FIFO. For the Segment realisation it is possible to do something similar. In figure 8.3, the

schematic of this concept is shown. In such an implementation, the consumer process is no longer responsible for reading the FIFO. This task is done by a controller located in the channel. An implementation without a FIFO has less latency, because data are transfered to the CAM as soon as they are written to the channel.[1] The cost of this latency reduction is the increasing complexity of the CAM controller.

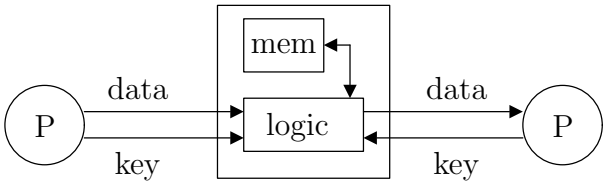



**Figure 8.3:** Reordering realisation without FIFO.

Latency calculation for a WFIFO is more complex than for the other concepts and depends on a lot of parameters. Here, not a complete calculation for the latency is given but the dependencies are shown. In the best case, a data item is written, and the write window is released immediately after that. Then the consumer acquires a read window that contains this item and reads it. The following questions must be answered:

- How many other items are written before the window is released and how long does this take?
- How long does it takes until the consumer can acquire a window that contains the respective item.

The first question can be answered by examining the producer and is not discussed any further. The more complex one is the second question because both consumer and producer process are involved. Following questions are of interest:

- Do read and write window have fixed sizes or do they change?
- Do read an write window have equal size? See Figure 7.2.
- Are the windows overlapping or not? See Figures 7.3 and 7.4.

A detailed discussion of the latency question for a WFIFO realisations is to long for this short comparison. It must treat all the combinations of possible answers to the questions listed above.

[1] In certain situations it is necessary to use a bigger CAM memory for such an architecture in order to benefit from the latency reduction.

## 8.7 Summary

In contrast to the FIFO buffer, CAM, Segment and WFIFO allow multiple read and reordering. For CAM, the number of read operations of an item is fixed in previous. The WFIFO allows multiple read and reordering only within the acquired windows. In Table 8.1 the complexity of the WFIFO API is classified as high because the process has to follow a protocol. CAM and Segment are both classified as medium because address generation is required.

| | Multiplicity | Reordering | Complexity |
|---|---|---|---|
| CAM | fixed | free | medium |
| Sequential | free | free | medium |
| WFIFO | window | window | high |

**Table 8.1:** Summary of API comparison.

The CAM is the only realisation that allows to delete items selectively from the buffer. Selectively means that it is possible to freely choose the next item to delete.

| | Operation | Selectively |
|---|---|---|
| CAM | number of reads = multiplicity counter | yes |
| Sequential | not defined | |
| WFIFO | release window | no |

**Table 8.2:** Removing data from the communication channel.

The CAM uses fewest memory of the concepts compared. The Segment realisation uses the most and the WFIFO is somewhere in between.

| | | |
|---|---|---|
| CAM | $\text{size}_{\text{Min}}$ | |
| Sequential | $\text{size}_{\text{Itr}}$ | $\text{size}_{\text{Itr}} \geq \text{size}_{\text{Min}}$ |
| WFIFO | $\text{size}_{\text{WFIFO}}$ | $\text{size}_{\text{Min}} \leq \text{size}_{\text{WFIFO}} \leq \text{size}_{\text{Itr}}$ |

**Table 8.3:** Memory requirement

To check if data are available, CAM and Segment realisation use the read operation. They can only check if a single item is available. With the WFIFO it is possible to check if a data sequence is available with a single instruction.

| | Command | Entity |
|---|---|---|
| CAM, Sequential | read | single item |
| WFIFO | acquireRead | data block |

**Table 8.4:** How to check disposability

# 9 Outlook and Conclusion

## 9.1 Conclusion

At the beginning of my thesis, I started with the idea of the windowed FIFO concept that was sketched out by my tutor Kai Huang. Together, we have defined the concept in more detail during the past six months. Although the WFIFO is a new concept and therefore this is the first work on this issue, I was able to make a complete and easy to use implementation that covers the complete functionality. With the automation of testing and system design and the theory on the WFIFO buffer, I have exceeded the objectives formulated at the beginning of my work. During my work I was able to benefit from my broad knowledge in digital IC design and could use my practical experience in testing of complex IC systems.

My implementation shows that the WFIFO concept is a feasible solution for the communication in on-chip multiprocessor systems. In addition, I have shown that the modularity of the concept enables automation of the design flow. For this purpose I designed a tool for automated system design and a tool for automated testing.

### 9.1.1 Summary of Completed Work

- The concept of the WFIFO buffer and the interface for buffer access were defined in detail. The interface was implemented with an API that includes features for buffer and system testing.

- The WFIFO concept was completely implemented for an FPGA platform. Processor and bus were taken from an IP library. A new IP was designed for the WFIFO buffer and two additional IPs were designed to simplify testing. The WFIFO IP is easy to use and no knowledge about WFIFO internals is required. It is realised in a single IP that can be configured over design parameters and it is compliant with the Xilinx IP library standard. Therefore it can be used in the Xilinx Platform Studio (XPS).

- An automated system design flow was implemented. The tool takes the application code for each process and a description of the process network topology as input and generates a synthesizeable and XPS compliant implementation of the system.

- A test concept was defined that is suited for testing the WFIFO IP and for complete WFIFO systems. It supports functional and performance tests and can be executed with a simulation environment (Modelsim) or it can be mapped on the FPGA.

- It was shown that with some restrictions the WFIFO concept is compliant with the KPN model of computation. Approaches to calculate the minimum memory requirements were presented.

## 9.2 Outlook

This section shows how the WFIFO concept could be further investigated and how it could be improved.

### 9.2.1 FIFO Read and FIFO Write

The FIFO read and write commands are a possible extension of the WFIFO concept. The format of these commands is shown below. They allow to transfer data to a WFIFO buffer without using windows but also without the possibilities of reordering, skipping and multiple read. FIFO read and write commands can only be executed if no windows are acquired.

```
FIFO_WRITE(port, data)
FIFO_READ(port, target)
```

The advantage of the new command is that the WFIFO can behave like a normal FIFO at the write port and like WFIFO at the read port or vice versa. If a producer or a consumer expects being connected to a FIFO, no changes are required. In many situation it is enough to use windows at one port. At the other port, all the acquire and release commands can be saved, which speeds up the execution. This approach can also reduce the latency. From the functional point of view, it is even possible to use FIFO and WFIFO commands at the same port. The only limitation is that windows must be closed before a code sequence with FIFO commands. A new window must not be opened before the code sequence of FIFO commands is finished. The question remains whether this is good practise.

### 9.2.2 Block Transfer

Block transfer is another option to extend the functionality of the WFIFO concept. The WFIFO read and write functions allow the transfer of single data items only. The new block read and write functions allow the transfer of continuous data blocks. A possible format is shown below. The parameter `start-offset` is used to position the first item to read or write within the window, `length` indicates the length of the data block and `address` is the start address of an array to read or write the data.

```
WFIFO_WRITE(port, start-offset, length, source-address)
WFIFO_READ(port, start-offset, length, target-address)
```

### 9.2.3 Multiple Processes on one Processor

The current implementation of the system generation tool (WAB) only allows a one-to-one mapping. This means that one processor is used for every process of network. An improved version of WAB allows to execute multiple processes on the same processor. The simplest solution for this is to use the operating system Xilkernel that was developed for the Xilinx FPGA processors and that allows to execute multiple processes. A problem of this OS is that it is non-preemptive. This means that the WFIFO API must be changed for the usage

on a system with Xilkernel. The blocking acquiring must be implemented with a polling mechanism that uses the non-blocking WFIFO instructions and that suspends the process if acquiring is not possible. Executing blocking instructions on a non-preemptive OS can cause a dead-lock.

### 9.2.4 Complete and Verify WFIFO Theory

The theory on the minimum memory requirements does not include a simple solution for the most general case of usage. The theory could be improved to cover also this case. Furthermore, Chapter 7 contains concepts and statements that I could not verify completely. The current version gives an idea of the solution approach but the concepts should be stated more precisely in order to prove their correctness.

### 9.2.5 Find Relevant Application Examples

In this thesis, only few examples of application are given and none of them is of great relevance. In order to show the advantages of the WFIFO concept, a problem must be found that benefits from the additional functionality. The chapter on the WFIFO theory may help to identify problems of interest.

From my point of view, a comparison must not be limited to properties like data throughput or latency. A major advantage of the WFIFO concept is that it can simplify hardware and software design and that the implementation is less hardware dependant. These advantages cannot be shown with a set of test data, they can only be discussed. I outlined some of these aspects in the comparison in Chapter 8.

# A Thesis Assignment

## Window-Based FIFOs for Communication in On-Chip Multiprocessor Systems

### 1 Introduction

We are looking for motivated students, willing to contribute to a large European project called SHAPES (http://www.shapes-p.org), which involves several partners from all over Europe. The project team includes such leading companies, universities and research labs as ST Microelectronics, ATMEL, THALES, TIMA, Target Compiler Technologies, Aachen University of Technology, Fraunhofer-Gesellschaft, and others. This project aims to find a scalable HW/SW design style for future CMOS technologies.

In TIK, a Distributed Operation Layer (DOL) is developing special for SHAPES. The purpose of the DOL is to significantly reduce the effort associated with the mapping of applications (from a restricted domain) onto multiprocessor SHAPES hardware platforms. Therefore, new communication model and interface are introduced for data transfer to fully decouple the application functionality from the hardware architecture and the mapping procedure.

In the DOL, we propose an extended read-write interface, instead of the traditional *read* and *write* functions. The proposed interface features a window-based access: Sliding windows are placed at both ends of a channel, one window at each end. Within a window the data in the channel can be accessed (i.e. be read and written) in a random order, while the window itself can move only in one direction (determined by the channel direction). This concept is illustrated in Figure 1.

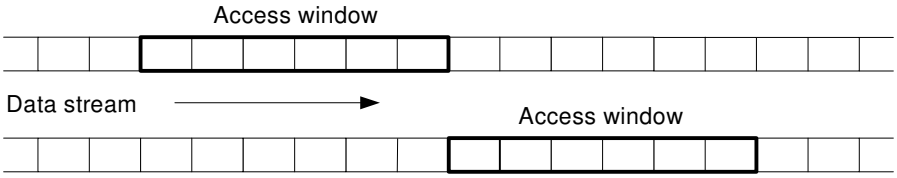



Figure 1: Window-based access mechanism.

### 2 Task

This diploma project includes several subtasks:

- Prototype this window-based access mechanism in FPGA and evaluate the trade-off of this approach.
- Explore FPGA-based test technique for both the Hardware WFIFO IPCore and target system which uses this IPCore.
- Probe possible ways for design flow and test flow automation for this approach.

**Kind of Work:** 20% theory, 50% design and implementation, 30% evaluation

# B Source Code

## B.1 WFIFO API

```
/************ wfifo-level1.h: WFIFO level 1 API **************/

/****************** Address Bus Settings ********************/

//mask
#define WFIFO_IPTYP_MSK 0xFF000000 // 8 bit ip typ (31 downto 24)
#define WFIFO_PORTN_MSK 0x00F00000 // 4 bit port no (23 downto 20)
#define WFIFO_FUNCN_MSK 0x000F0000 // 4 bit function (19 downto 16)
#define WFIFO_OFFST_MSK 0x0000FFFF // 16 bit offset (15 downto 0)

// shift
#define WFIFO_PORTN_SFT 20
#define WFIFO_OFFST_SFT 0

// settings
#define WFIFO_READ_ID             0x00010000
#define WFIFO_WRITE_ID            0x00020000
#define WFIFO_ACQUIRE_READ_ID     0x00030000
#define WFIFO_ACQUIRE_WRITE_ID    0x00040000
#define WFIFO_RELEASE_READ_ID     0x00050000
#define WFIFO_RELEASE_WRITE_ID    0x00060000
#define WFIFO_GETSTATUS_ID        0x00070000
#define WFIFO_MIR_RESET_ID        0x000F0000
#define WFIFO_IPTYP_ID            0x01000000

/******************** Data Bus Settings *********************/

// mask
#define WFIFO_WSIZE_MSK 0x0000FFFF  // 16 bit offset (15 downto 0)
#define WFIFO_BLOCK_MSK 0x00010000  // 1 bit blocking (16)

// shift
#define WFIFO_WSIZE_SFT 0
#define WFIFO_BLOCK_SFT  16

/*************** WFIFO API with assertion *****************/
#ifdef WFIFO_ASSERT

#define WFIFO_WRITE(port, offset, data)\
 (WFIFO_WRITE_I(port, offset, data), wfifoAssert(WFIFO_GETSTATUS(port)))

#define WFIFO_READ(port, offset, target) \
 (WFIFO_READ_I(port, offset, target), wfifoAssert(WFIFO_GETSTATUS(port)))

#define WFIFO_ACQUIRE_READ(port, size)\
 (WFIFO_ACQUIRE_READ_I(port, size, 1), wfifoAssert(WFIFO_GETSTATUS(port)))
```

```
#define WFIFO_ACQUIRE_READ_NONBLK(port, size)\
 (WFIFO_ACQUIRE_READ_I(port, size, 0), wfifoAssert(WFIFO_GETSTATUS(port)))

#define WFIFO_ACQUIRE_WRITE(port, size)\
 (WFIFO_ACQUIRE_WRITE_I(port, size, 1), wfifoAssert(WFIFO_GETSTATUS(port)))

#define WFIFO_ACQUIRE_WRITE_NONBLK(port, size)\
 (WFIFO_ACQUIRE_WRITE_I(port, size, 0), wfifoAssert(WFIFO_GETSTATUS(port)))

#define WFIFO_RELEASE_READ(port)\
 (WFIFO_RELEASE_READ_I(port), wfifoAssert(WFIFO_GETSTATUS(port)))

#define WFIFO_RELEASE_WRITE(port)\
 (WFIFO_RELEASE_WRITE_I(port), wfifoAssert(WFIFO_GETSTATUS(port)))

/*********** WFIFO API without assertion ***************/
#else

#define WFIFO_WRITE(port, offset, data)\
 (WFIFO_WRITE_I(port, offset, data))

#define WFIFO_READ(port, offset, target) \
 (WFIFO_READ_I(port, offset, target))

#define WFIFO_ACQUIRE_READ(port, size)\
 (WFIFO_ACQUIRE_READ_I(port, size, 1))

#define WFIFO_ACQUIRE_READ_NONBLK(port, size)\
 (WFIFO_ACQUIRE_READ_I(port, size, 0), WFIFO_GETSTATUS(port))

#define WFIFO_ACQUIRE_WRITE(port, size)\
 (WFIFO_ACQUIRE_WRITE_I(port, size, 1))

#define WFIFO_ACQUIRE_WRITE_NONBLK(port, size)\
 (WFIFO_ACQUIRE_WRITE_I(port, size, 0), WFIFO_GETSTATUS(port))

#define WFIFO_RELEASE_READ(port)\
 (WFIFO_RELEASE_READ_I(port))

#define WFIFO_RELEASE_WRITE(port)\
 (WFIFO_RELEASE_WRITE_I(port))

#endif

/**************** Assertion Functions ********************/

#ifdef WFIFO_ASSERT

#ifdef ASSERT_NONBLK
void wfifoWait() // do nothing
{}
```

```
#else
void wfifoWait()
{
  xil_printf("Execution STOPPED\n");
  int w=0;
  while (w==0) {}
}
#endif //ASSERT_NONBLK

int wfifoAssert(int status) {
  if (status==1) {
    xil_printf("ERROR: illegal operation\n");
    wfifoWait();
  }
  return status;
}

#endif //WFIFO_ASSERT


/***************** Reset/MIR Function ********************/

#define WFIFO_GETMIR(port) \
(XIo_In32( \
   ( WFIFO_IPTYP_ID \
     | (WFIFO_PORTN_MSK & (port<<WFIFO_PORTN_SFT)) \
     | WFIFO_MIR_RESET_ID ))\
)

#define WFIFO_RESET(port) \
(XIo_Out32( \
   ( WFIFO_IPTYP_ID \
     | (WFIFO_PORTN_MSK & (port<<WFIFO_PORTN_SFT)) \
     | WFIFO_MIR_RESET_ID ),0xA)\
)

/***************** Status Report Function ********************/

#define WFIFO_GETSTATUS(port) \
(XIo_In32( \
   ( WFIFO_IPTYP_ID \
     | (WFIFO_PORTN_MSK & (port<<WFIFO_PORTN_SFT)) \
     | WFIFO_GETSTATUS_ID ))\
)


/********** Function Macros Without Status Report ************/

/* WFIFO_READ */

#define WFIFO_READ_I(port, offset, target) \
(target = XIo_In32( \
```

```
    ( WFIFO_IPTYP_ID \
      | (WFIFO_PORTN_MSK & (port<<WFIFO_PORTN_SFT)) \
      | WFIFO_READ_ID \
      | (WFIFO_OFFST_MSK & (offset<<WFIFO_OFFST_SFT))))\
)

/* WFIFO_WRITE */

#define WFIFO_WRITE_I(port, offset, data) \
(XIo_Out32( \
    ( WFIFO_IPTYP_ID \
      | (WFIFO_PORTN_MSK & (port<<WFIFO_PORTN_SFT)) \
      | WFIFO_WRITE_ID \
      | (WFIFO_OFFST_MSK & (offset<<WFIFO_OFFST_SFT))),\
    data)\
)

/* acquire write window */

#define WFIFO_ACQUIRE_READ_I(port, size, blocking) \
(XIo_Out32( \
    ( WFIFO_IPTYP_ID \
      | (WFIFO_PORTN_MSK & (port<<WFIFO_PORTN_SFT)) \
      | WFIFO_ACQUIRE_READ_ID),\
      (WFIFO_BLOCK_MSK & (blocking<<WFIFO_BLOCK_SFT))\
      | (WFIFO_WSIZE_MSK & (size<<WFIFO_WSIZE_SFT)))\
)

/* acquire read window */

#define WFIFO_ACQUIRE_WRITE_I(port, size, blocking) \
(XIo_Out32( \
    ( WFIFO_IPTYP_ID \
      | (WFIFO_PORTN_MSK & (port<<WFIFO_PORTN_SFT)) \
      | WFIFO_ACQUIRE_WRITE_ID),\
      (WFIFO_BLOCK_MSK & (blocking<<WFIFO_BLOCK_SFT))\
      | (WFIFO_WSIZE_MSK & (size<<WFIFO_WSIZE_SFT)))\
)

/* release read window */

#define WFIFO_RELEASE_READ_I(port) \
(XIo_Out32( \
    ( WFIFO_IPTYP_ID \
      | (WFIFO_PORTN_MSK & (port<<WFIFO_PORTN_SFT)) \
      | WFIFO_RELEASE_READ_ID),\
      0)\
)

/* release write window */

#define WFIFO_RELEASE_WRITE_I(port) \
```

```
(XIo_Out32( \
   ( WFIFO_IPTYP_ID \
     | (WFIFO_PORTN_MSK & (port<<WFIFO_PORTN_SFT)) \
     | WFIFO_RELEASE_WRITE_ID),\
     0)\
)
```

## B.2 OPB Recorder API

```
/******* opb-recorder.h: Record WFIFO opb bus events **********/

/*************************************************************/
#include "xparameters.h"    /* generated system parameters    */
#include "xbasic_types.h"   /* basic types for device drivers */
#include "xio.h"            /* bus access                     */
/*********************** Settings ****************************/


//#define RECORDER_IPTYP_ID XPAR_OPB_RECORDER_0_BASEADDR
#define RECORDER_IPTYP_ID          0x02000000

// command encoding
#define OPB_RECORDER_INIT_ID       0x00000000
#define OPB_RECORDER_PLAY_ID       0x00000001
#define OPB_RECORDER_GETNEXT_ID    0x00000002
#define OPB_RECORDER_HASMORE_ID    0x00000003


/******************** Function Macros  ***********************/

#define OPB_RECORDER_INIT() \
(XIo_Out32(RECORDER_IPTYP_ID | OPB_RECORDER_INIT_ID,0x0))

#define OPB_RECORDER_PLAY() \
(XIo_Out32(RECORDER_IPTYP_ID | OPB_RECORDER_PLAY_ID,0x0))

#define OPB_RECORDER_HASMORE() \
(XIo_In32(RECORDER_IPTYP_ID | OPB_RECORDER_HASMORE_ID))

#define OPB_RECORDER_GETNEXT() \
(XIo_In32(RECORDER_IPTYP_ID | OPB_RECORDER_GETNEXT_ID))


/********************** Functions  ***************************/

void replayOpbRecorder()
{
  OPB_RECORDER_PLAY(); //switch to play mode

  int tmp1,tmp2;

  xil_printf("# start replay\n");
```

```
  while (OPB_RECORDER_HASMORE() == 1) //has more tokens
  {
    tmp1 = OPB_RECORDER_GETNEXT();
    tmp2 = OPB_RECORDER_GETNEXT();
    xil_printf("%d %d\n",tmp1,tmp2);
  }

  xil_printf("# end replay\n");

  OPB_RECORDER_INIT(); //restart recorder
}
```

# C VHDL Code

## C.1 WFIFO

### C.1.1 WFIFO Top

```
------------------------------------------------------------------------------
-- Filename:          wfifo.vhd
-- Version:           v1.00a
-- Description:       top of wfifo
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

library opb_ipif_wfifo_v1_00_a;
use opb_ipif_wfifo_v1_00_a.all;

library Unisim;
use Unisim.all;

library wfifo_v1_00_a;
use wfifo_v1_00_a.all;

------------------------------------------------------------------------------
-- Entity Section
------------------------------------------------------------------------------

entity wfifo is
  generic
  (
    C_W_BASEADDR          : std_logic_vector(0 to 31) := X"FFFFFFFF";
    C_W_HIGHADDR          : std_logic_vector(0 to 31) := X"00000000";
    C_R_BASEADDR          : std_logic_vector(0 to 31) := X"FFFFFFFF";
    C_R_HIGHADDR          : std_logic_vector(0 to 31) := X"00000000";
    C_OPB_AWIDTH          : integer                   := 32;
    C_OPB_DWIDTH          : integer                   := 32;
    C_WFIFO_MEMSIZE       : integer                   := 8;
    C_WFIFO_WINDWIDTH     : integer                   := 10;
    C_WFIFO_BRAMWIDTH     : integer                   := 10;
    C_WFIFO_PIPELINEMODE : integer                   := 5
  );
  port
  (
    -- write
    OPB_W_ABus      : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_W_BE        : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_W_Clk       : in  std_logic;
    OPB_W_DBus      : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_W_RNW       : in  std_logic;
    OPB_W_Rst       : in  std_logic;
    OPB_W_select    : in  std_logic;
    OPB_W_seqAddr   : in  std_logic;
    Sln_W_DBus      : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sln_W_errAck    : out std_logic;
    Sln_W_retry     : out std_logic;
    Sln_W_toutSup   : out std_logic;
    Sln_W_xferAck   : out std_logic;
    -- read
    OPB_R_ABus      : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_R_BE        : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
```

```
    OPB_R_Clk       : in  std_logic;
    OPB_R_DBus      : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_R_RNW       : in  std_logic;
    OPB_R_Rst       : in  std_logic;
    OPB_R_select    : in  std_logic;
    OPB_R_seqAddr   : in  std_logic;
    Sln_R_DBus      : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sln_R_errAck    : out std_logic;
    Sln_R_retry     : out std_logic;
    Sln_R_toutSup   : out std_logic;
    Sln_R_xferAck   : out std_logic
  );

end entity wfifo;

------------------------------------------------------------------------------
-- Architecture Section
------------------------------------------------------------------------------

architecture behavioral of wfifo is

------------------------------------------------------------------------------
-- Constant Declarations
------------------------------------------------------------------------------

-- wfifo ip identification (mir)
-- major version number
constant C_WFIFO_VER_MAJ : std_logic_vector(3 downto 0) := "0001";
-- minor version number
constant C_WFIFO_VER_MIN : std_logic_vector(6 downto 0) := "0000000";
-- minor version letter
constant C_WFIFO_VER_LET : std_logic_vector(4 downto 0) := "00000";
-- block id
constant C_WFIFO_BLK_ID  : std_logic_vector(7 downto 0) := "00000011";
-- block typ
constant C_WFIFO_BLK_TYP : std_logic_vector(7 downto 0) := "00000001";

constant C_WFIFO_MIR_CODE : integer  :=
  to_integer(unsigned(C_WFIFO_VER_MAJ & C_WFIFO_VER_MIN & C_WFIFO_VER_LET
                      & C_WFIFO_BLK_ID & C_WFIFO_BLK_TYP));

constant C_WFIFO_MIR_BAOFFSET : std_logic_vector(0 to 31)
  := "00000000000001111000000000000000";
constant C_WFIFO_MIR_HIOFFSET : std_logic_vector(0 to 31)
  := "00000000000011110000000000000001";

------------------------------------------------------------------------------
-- Signal and Type Declarations
------------------------------------------------------------------------------
signal Bus2IP_W_Addr         : std_logic_vector(0 to C_OPB_AWIDTH-1);
signal Bus2IP_W_Clk          : std_logic;
signal Bus2IP_W_CS           : std_logic;
signal Bus2IP_W_WrCE         : std_logic;
signal Bus2IP_W_RdCE         : std_logic;
signal Bus2IP_W_Data         : std_logic_vector(0 to C_OPB_DWIDTH-1);
signal Bus2IP_W_Reset        : std_logic;
signal Bus2IP_W_RNW          : std_logic;
signal IP2Bus_W_Ack          : std_logic;
signal IP2Bus_W_Data         : std_logic_vector(0 to C_OPB_DWIDTH-1);
signal IP2Bus_W_Error        : std_logic;
signal IP2Bus_W_PostedWrInh  : std_logic;
signal IP2Bus_W_Retry        : std_logic;
signal IP2Bus_W_ToutSup      : std_logic;

signal Bus2IP_R_Addr         : std_logic_vector(0 to C_OPB_AWIDTH-1);
signal Bus2IP_R_Clk          : std_logic;
```

```
signal Bus2IP_R_CS           : std_logic;
signal Bus2IP_R_WrCE         : std_logic;
signal Bus2IP_R_RdCE         : std_logic;
signal Bus2IP_R_Data         : std_logic_vector(0 to C_OPB_DWIDTH-1);
signal Bus2IP_R_Reset        : std_logic;
signal Bus2IP_R_RNW          : std_logic;
signal IP2Bus_R_Ack          : std_logic;
signal IP2Bus_R_Data         : std_logic_vector(0 to C_OPB_DWIDTH-1);
signal IP2Bus_R_Error        : std_logic;
signal IP2Bus_R_PostedWrInh  : std_logic;
signal IP2Bus_R_Retry        : std_logic;
signal IP2Bus_R_ToutSup      : std_logic;

signal Bram_W_AddrxD         : std_logic_vector(31 downto 0);
signal Bram_W_DataxD         : std_logic_vector(31 downto 0);
signal Bram_W_WrEnablexD     : std_logic;

signal Bram_R_AddrxD         : std_logic_vector(31 downto 0);
signal Bram_R_DataxD         : std_logic_vector(31 downto 0);

signal WfifoResetxD     : std_logic;
---------------------------------------------------------------------------
-- Component Declarations
---------------------------------------------------------------------------

 component wfifo_logic is
  generic
  (
    C_WFIFO_MEMSIZE      : integer := 8;   -- in KB
    C_WFIFO_WINDWIDTH    : integer := 11;  -- in bit
    C_WFIFO_BRAMWIDTH    : integer := 11   -- in bit
  );
   port
   (
     -- write port
     Bus2IP_W_AddrxDI     : in  std_logic_vector(0 to 31);
     Bus2IP_W_ClkxCI      : in  std_logic;
     Bus2IP_W_CSxDI       : in  std_logic;
     Bus2IP_W_WrCExDI     : in  std_logic;
     Bus2IP_W_RdCExDI     : in  std_logic;
     Bus2IP_W_DataxDI     : in  std_logic_vector(0 to 31);
     Bus2IP_W_ResetxDI    : in  std_logic;
     Bus2IP_W_RNWxDI      : in  std_logic;
     IP2Bus_W_DataxDO     : out std_logic_vector(0 to 31);
     IP2Bus_W_AckxDO      : out  std_logic;
     IP2Bus_W_ErrorxDO    : out  std_logic;
     IP2Bus_W_ToutSupxDO  : out  std_logic;
     Bram_W_AddrxDO       : out  std_logic_vector(31 downto 0);
     Bram_W_DataxDO       : out  std_logic_vector(31 downto 0);
     Bram_W_WrEnablexDO   : out  std_logic;
     -- read port
     Bus2IP_R_AddrxDI     : in  std_logic_vector(0 to 31);
     Bus2IP_R_ClkxCI      : in  std_logic;
     Bus2IP_R_CSxDI       : in  std_logic;
     Bus2IP_R_WrCExDI     : in  std_logic;
     Bus2IP_R_RdCExDI     : in  std_logic;
     Bus2IP_R_DataxDI     : in  std_logic_vector(0 to 31);
     Bus2IP_R_ResetxDI    : in  std_logic;
     Bus2IP_R_RNWxDI      : in  std_logic;
     IP2Bus_R_DataxDO     : out std_logic_vector(0 to 31);
     IP2Bus_R_AckxDO      : out  std_logic;
     IP2Bus_R_ErrorxDO    : out  std_logic;
     IP2Bus_R_ToutSupxDO  : out  std_logic;
     Bram_R_AddrxDO       : out  std_logic_vector(31 downto 0);
     Bram_R_DataxDI       : in  std_logic_vector(31 downto 0)
   );
```

```
 end component wfifo_logic;


component opb_ipif_wfifo is
  generic
  (
    C_BASEADDR          : std_logic_vector(0 to 31) := X"FFFFFFFF";
    C_HIGHADDR          : std_logic_vector(0 to 31) := X"00000000";
    C_MIR_BASEADDR      : std_logic_vector(0 to 31) := X"FFFFFFFF";
    C_MIR_HIGHADDR      : std_logic_vector(0 to 31) := X"00000000";
    C_WFIFO_MIR_CODE    : integer                   := 1;
    C_PIPELINE_MODEL    : integer                   := 5;
    C_OPB_AWIDTH        : integer                   := 32;
    C_OPB_DWIDTH        : integer                   := 32;
    C_FAMILY            : string                    := "virtex2"
  );
  port
  (
      OPB_ABus            : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
      OPB_BE              : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
      OPB_Clk             : in  std_logic;
      OPB_DBus            : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
      OPB_RNW             : in  std_logic;
      OPB_Rst             : in  std_logic;
      OPB_select          : in  std_logic;
      OPB_seqAddr         : in  std_logic;
      Sln_DBus            : out std_logic_vector(0 to C_OPB_DWIDTH-1);
      Sln_errAck          : out std_logic;
      Sln_retry           : out std_logic;
      Sln_toutSup         : out std_logic;
      Sln_xferAck         : out std_logic;
      Bus2IP_Addr         : out std_logic_vector(0 to C_OPB_AWIDTH-1);
      Bus2IP_BE           : out std_logic_vector(0 to C_OPB_DWIDTH/8-1);
      Bus2IP_Burst        : out std_logic;
      Bus2IP_Clk          : out std_logic;
      Bus2IP_CS           : out std_logic;
      Bus2IP_Data         : out std_logic_vector(0 to C_OPB_DWIDTH-1);
      Bus2IP_RdCE         : out std_logic;
      Bus2IP_Reset        : out std_logic;
      Bus2IP_RNW          : out std_logic;
      Bus2IP_WrCE         : out std_logic;
      IP2Bus_Ack          : in  std_logic;
      IP2Bus_Data         : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
      IP2Bus_Error        : in  std_logic;
      IP2Bus_PostedWrInh  : in  std_logic;
      IP2Bus_Retry        : in  std_logic;
      IP2Bus_ToutSup      : in  std_logic
  );
end component opb_ipif_wfifo;


component  wfifo_bram is
  generic
  (
    C_BRAM_SIZE    : integer := 8;        -- in KB
    C_BRAM_AWIDTH : integer := 32;
    C_BRAM_DWIDTH : integer := 32
  );
  port
  (
    Wfifo_Bram_ClkxCI     : in  std_logic;
    Wfifo_Bram_RstxRI     : in  std_logic;
    -- write
    Bram_W_AddrxDI       : in  std_logic_vector(C_BRAM_AWIDTH-1 downto 0);
    Bram_W_DataxDI       : in  std_logic_vector(C_BRAM_DWIDTH-1 downto 0);
    Bram_W_WrEnablexDI   : in  std_logic;
```

```
    -- read
    Bram_R_AddrxDI       : in  std_logic_vector(C_BRAM_AWIDTH-1 downto 0);
    Bram_R_DataxD0       : out std_logic_vector(C_BRAM_DWIDTH-1 downto 0)
  );
end component wfifo_bram;

-------------------------------------------------------------------------------
begin
-------------------------------------------------------------------------------


wfifo_bram_0 :  wfifo_bram
  generic map
  (
    C_BRAM_SIZE    =>  C_WFIFO_MEMSIZE,        -- in KB
    C_BRAM_AWIDTH => 32,
    C_BRAM_DWIDTH => 32
  )
  port map
  (
    Wfifo_Bram_ClkxCI      => Bus2IP_W_Clk,
    Wfifo_Bram_RstxRI      => WfifoResetxD,
    -- write
    Bram_W_AddrxDI         => Bram_W_AddrxD,
    Bram_W_DataxDI         => Bram_W_DataxD,
    Bram_W_WrEnablexDI     => Bram_W_WrEnablexD,
    -- read
    Bram_R_AddrxDI         => Bram_R_AddrxD,
    Bram_R_DataxD0         => Bram_R_DataxD
  );


OPB_IPIF_W_I : opb_ipif_wfifo
  generic map
  (
    C_BASEADDR             => C_W_BASEADDR,
    C_HIGHADDR             => C_W_HIGHADDR,
    C_MIR_BASEADDR         => C_W_BASEADDR or C_WFIFO_MIR_BAOFFSET,
    C_MIR_HIGHADDR         => C_W_BASEADDR or C_WFIFO_MIR_HIOFFSET,
    C_WFIFO_MIR_CODE       => C_WFIFO_MIR_CODE,
    C_PIPELINE_MODEL       => C_WFIFO_PIPELINEMODE,
    C_OPB_AWIDTH           => C_OPB_AWIDTH,
    C_OPB_DWIDTH           => C_OPB_DWIDTH
  )
  port map
  (
    OPB_ABus               => OPB_W_ABus,
    OPB_BE                 => OPB_W_BE,
    OPB_Clk                => OPB_W_Clk,
    OPB_DBus               => OPB_W_DBus,
    OPB_RNW                => OPB_W_RNW,
    OPB_Rst                => OPB_W_Rst,
    OPB_select             => OPB_W_select,
    OPB_seqAddr            => OPB_W_seqAddr,
    Sln_DBus               => Sln_W_DBus,
    Sln_errAck             => Sln_W_errAck,
    Sln_retry              => Sln_W_retry,
    Sln_toutSup            => Sln_W_toutSup,
    Sln_xferAck            => Sln_W_xferAck,
    Bus2IP_Addr            => Bus2IP_W_Addr,
    Bus2IP_BE              => open,
    Bus2IP_Burst           => open,
    Bus2IP_Clk             => Bus2IP_W_Clk,
    Bus2IP_CS              => Bus2IP_W_CS,
    Bus2IP_Data            => Bus2IP_W_Data,
```

```
    Bus2IP_RdCE          => Bus2IP_W_RdCE,
    Bus2IP_Reset         => Bus2IP_W_Reset,
    Bus2IP_RNW           => Bus2IP_W_RNW,
    Bus2IP_WrCE          => Bus2IP_W_WrCE,
    IP2Bus_Ack           => IP2Bus_W_Ack,
    IP2Bus_Data          => IP2Bus_W_Data,
    IP2Bus_Error         => IP2Bus_W_Error,
    IP2Bus_PostedWrInh   => IP2Bus_W_PostedWrInh,
    IP2Bus_Retry         => IP2Bus_W_Retry,
    IP2Bus_ToutSup       => IP2Bus_W_ToutSup
  );


OPB_IPIF_R_I : opb_ipif_wfifo
  generic map
  (
    C_BASEADDR           => C_R_BASEADDR,
    C_HIGHADDR           => C_R_HIGHADDR,
    C_MIR_BASEADDR       => C_R_BASEADDR or C_WFIFO_MIR_BAOFFSET,
    C_MIR_HIGHADDR       => C_R_BASEADDR or C_WFIFO_MIR_HIOFFSET,
    C_WFIFO_MIR_CODE     => C_WFIFO_MIR_CODE,
    C_PIPELINE_MODEL     => C_WFIFO_PIPELINEMODE,
    C_OPB_AWIDTH         => C_OPB_AWIDTH,
    C_OPB_DWIDTH         => C_OPB_DWIDTH
  )
  port map
  (
    OPB_ABus             => OPB_R_ABus,
    OPB_BE               => OPB_R_BE,
    OPB_Clk              => OPB_W_Clk,     -- same clock is used for both IPIF
    OPB_DBus             => OPB_R_DBus,
    OPB_RNW              => OPB_R_RNW,
    OPB_Rst              => OPB_R_Rst,
    OPB_select           => OPB_R_select,
    OPB_seqAddr          => OPB_R_seqAddr,
    Sln_DBus             => Sln_R_DBus,
    Sln_errAck           => Sln_R_errAck,
    Sln_retry            => Sln_R_retry,
    Sln_toutSup          => Sln_R_toutSup,
    Sln_xferAck          => Sln_R_xferAck,
    Bus2IP_Addr          => Bus2IP_R_Addr,
    Bus2IP_BE            => open,
    Bus2IP_Burst         => open,
    Bus2IP_Clk           => Bus2IP_R_Clk,
    Bus2IP_CS            => Bus2IP_R_CS,
    Bus2IP_Data          => Bus2IP_R_Data,
    Bus2IP_RdCE          => Bus2IP_R_RdCE,
    Bus2IP_Reset         => Bus2IP_R_Reset,
    Bus2IP_RNW           => Bus2IP_R_RNW,
    Bus2IP_WrCE          => Bus2IP_R_WrCE,
    IP2Bus_Ack           => IP2Bus_R_Ack,
    IP2Bus_Data          => IP2Bus_R_Data,
    IP2Bus_Error         => IP2Bus_R_Error,
    IP2Bus_PostedWrInh   => IP2Bus_R_PostedWrInh,
    IP2Bus_Retry         => IP2Bus_R_Retry,
    IP2Bus_ToutSup       => IP2Bus_R_ToutSup
  );


 WFIFO_LOGIC_I : wfifo_logic
   generic map
   (
     C_WFIFO_MEMSIZE => C_WFIFO_MEMSIZE,
     C_WFIFO_WINDWIDTH => C_WFIFO_WINDWIDTH,
     C_WFIFO_BRAMWIDTH => C_WFIFO_BRAMWIDTH
```

```
    )
    port map
    (
      -- write
      Bus2IP_W_AddrxDI      => Bus2IP_W_Addr,
      Bus2IP_W_ClkxCI       => Bus2IP_W_Clk,
      Bus2IP_W_CSxDI        => Bus2IP_W_CS,
      Bus2IP_W_WrCExDI      => Bus2IP_W_WrCE,
      Bus2IP_W_RdCExDI      => Bus2IP_W_RdCE,
      Bus2IP_W_DataxDI      => Bus2IP_W_Data,
      Bus2IP_W_ResetxDI     => Bus2IP_W_Reset,
      Bus2IP_W_RNWxDI       => Bus2IP_W_RNW,
      IP2Bus_W_DataxDO      => IP2Bus_W_Data,
      IP2Bus_W_AckxDO       => IP2Bus_W_Ack,
      IP2Bus_W_ErrorxDO     => IP2Bus_W_Error,
      IP2Bus_W_ToutSupxDO   => IP2Bus_W_ToutSup,
      Bram_W_AddrxDO        => Bram_W_AddrxD,
      Bram_W_DataxDO        => Bram_W_DataxD,
      Bram_W_WrEnablexDO    => Bram_W_WrEnablexD,
      -- read
      Bus2IP_R_AddrxDI      => Bus2IP_R_Addr,
      Bus2IP_R_ClkxCI       => Bus2IP_R_Clk,
      --Bus2IP_R_ClkxCI       => Bus2IP_W_Clk,
      Bus2IP_R_CSxDI        => Bus2IP_R_CS,
      Bus2IP_R_WrCExDI      => Bus2IP_R_WrCE,
      Bus2IP_R_RdCExDI      => Bus2IP_R_RdCE,
      Bus2IP_R_DataxDI      => Bus2IP_R_Data,
      Bus2IP_R_ResetxDI     => Bus2IP_R_Reset,
      Bus2IP_R_RNWxDI       => Bus2IP_R_RNW,
      IP2Bus_R_DataxDO      => IP2Bus_R_Data,
      IP2Bus_R_AckxDO       => IP2Bus_R_Ack,
      IP2Bus_R_ErrorxDO     => IP2Bus_R_Error,
      IP2Bus_R_ToutSupxDO   => IP2Bus_R_ToutSup,
      Bram_R_AddrxDO        => Bram_R_AddrxD,
      Bram_R_DataxDI        => Bram_R_DataxD
    );

IP2Bus_W_PostedWrInh <= '0'; -- do not inhibit posted write
IP2Bus_R_PostedWrInh <= '0'; -- do not inhibit posted write

IP2Bus_W_Retry <= '0';
IP2Bus_R_Retry <= '0';

WfifoResetxD <= Bus2IP_W_Reset or Bus2IP_R_Reset;


end architecture behavioral;
```

## C.1.2 WFIFO Logic

```
------------------------------------------------------------------------------
-- Filename:          wfifo_logic.vhd
-- Version:           v1.00a
-- Description:       WFIFO logic
------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library Unisim;
use Unisim.all;

entity wfifo_logic is
  generic
  (
```

```
    C_WFIFO_MEMSIZE      : integer := 8;  -- in KB
    C_WFIFO_WINDWIDTH    : integer := 11; -- in bit
    C_WFIFO_BRAMWIDTH    : integer := 11  -- in bit
  );
  port (
    -- write port (NU stands for not used)
    Bus2IP_W_AddrxDI     : in  std_logic_vector(0 to 31);     -- address
    Bus2IP_W_ClkxCI      : in  std_logic;                     -- clock
    Bus2IP_W_CSxDI       : in  std_logic;                     -- address in range
    Bus2IP_W_WrCExDI     : in  std_logic;                     -- write is pending
    Bus2IP_W_RdCExDI     : in  std_logic;                     -- read is pending
    Bus2IP_W_DataxDI     : in  std_logic_vector(0 to 31);     -- data in
    Bus2IP_W_ResetxDI    : in  std_logic;                     -- reset
    Bus2IP_W_RNWxDI      : in  std_logic;                     -- read not write
    IP2Bus_W_DataxDO     : out std_logic_vector(0 to 31);     -- data out
    IP2Bus_W_AckxDO      : out std_logic;                     -- output acknowledge
    IP2Bus_W_ErrorxDO    : out std_logic;                     -- error (NU)
    IP2Bus_W_ToutSupxDO  : out std_logic;                     -- timeout supress
    Bram_W_AddrxDO       : out std_logic_vector(31 downto 0); -- bram write address
    Bram_W_DataxDO       : out std_logic_vector(31 downto 0); -- bram data out
    Bram_W_WrEnablexDO   : out std_logic;                     -- bram write enable
    -- read port
    Bus2IP_R_AddrxDI     : in  std_logic_vector(0 to 31);     -- address
    Bus2IP_R_ClkxCI      : in  std_logic;                     -- clock
    Bus2IP_R_CSxDI       : in  std_logic;                     -- address in range
    Bus2IP_R_WrCExDI     : in  std_logic;                     -- write is pending
    Bus2IP_R_RdCExDI     : in  std_logic;                     -- read is pending
    Bus2IP_R_DataxDI     : in  std_logic_vector(0 to 31);     -- data in
    Bus2IP_R_ResetxDI    : in  std_logic;                     -- reset
    Bus2IP_R_RNWxDI      : in  std_logic;                     -- read not write
    IP2Bus_R_DataxDO     : out std_logic_vector(0 to 31);     -- data out
    IP2Bus_R_AckxDO      : out std_logic;                     -- output acknowledge
    IP2Bus_R_ErrorxDO    : out std_logic;                     -- error (NU)
    IP2Bus_R_ToutSupxDO  : out std_logic;                     -- timeout supress
    Bram_R_AddrxDO       : out std_logic_vector(31 downto 0); -- bram read address
    Bram_R_DataxDI       : in  std_logic_vector(31 downto 0)  -- bram data in
    );
end entity wfifo_logic;

---------------------------------------------------------------------------
-- Architecture section
---------------------------------------------------------------------------

architecture behavioral of wfifo_logic is

---------------------------------------------------------------------------
-- Constants
---------------------------------------------------------------------------
-- command size and position in Bus2IP_Addr (big endian)
constant CommandLsb : integer := 15;
constant CommandMsb : integer := 12;
constant CommandSiz : integer := 3;

-- blocking bit position in Bus2IP_Addr (big endian)
constant BlockLsb : integer := 15;

-- data size and position in Bus2IP_Data (big endian)
constant DataLsb : integer := 31;
constant DataMsb : integer := 0;
constant DataSiz : integer := 31;

--  address (=offset) size and position in Bus2IP_Addr (big endian)
constant AddrLsb : integer := 31;
constant AddrMsb : integer := 32-C_WFIFO_WINDWIDTH;
constant AddrSiz : integer := C_WFIFO_WINDWIDTH-1;    -- width-1
```

```
-- window size and position in Bus2IP_Data (big endian)
constant WindLsb : integer := 31;
constant WindMsb : integer := 32-C_WFIFO_WINDWIDTH;
constant WindSiz : integer := C_WFIFO_WINDWIDTH-1;     -- width-1

-- highest mem address
constant BramSizeAbs : integer := C_WFIFO_MEMSIZE*256;

-- Address size (required bits-1) depends on the used memory
-- Values: 9 (2K), 10 (4K), 11 (8K), 12 (16K)
constant BramAddrSize : integer := C_WFIFO_BRAMWIDTH-1;     -- width-1

-- command ids
constant ComRead       : std_logic_vector(CommandSiz downto 0) := "0001"; -- read
constant ComWrite      : std_logic_vector(CommandSiz downto 0) := "0010"; -- write
constant ComAcRead     : std_logic_vector(CommandSiz downto 0) := "0011"; -- Acquire Read
constant ComAcWrite    : std_logic_vector(CommandSiz downto 0) := "0100"; -- Acquire Write
constant ComReRead     : std_logic_vector(CommandSiz downto 0) := "0101"; -- Release Read
constant ComReWrite    : std_logic_vector(CommandSiz downto 0) := "0110"; -- Release Write
constant ComGetStatus  : std_logic_vector(CommandSiz downto 0) := "0111"; -- Get Status

------------------------------------------------------------------------------
-- Types
------------------------------------------------------------------------------

type WfifoWrState is (s_WrIdle, s_WrStAck, s_WrErr,
                      s_AcWrAck, s_AcWrBlk, s_ReWrAck, s_WrDaAck);

type WfifoRdState is (s_RdIdle, s_RdStAck, s_RdErr,
                      s_AcRdAck, s_AcRdBlk, s_ReRdAck, s_RdDaAck, s_RdDaOk);

type WfifoReadCommand is  (s_AcRd, s_RdDa, s_ReRd, s_NoRd, s_StRd);
type WfifoWriteCommand is (s_AcWr, s_WrDa, s_ReWr, s_NoWr, s_StWr);

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

-- inputs
signal WrAddrxDI      : std_logic_vector(AddrSiz downto 0);
signal WrCommandxDI   : std_logic_vector(CommandSiz downto 0);
signal WrDataxDI      : std_logic_vector(DataSiz downto 0);
signal WrWindSizexDI  : std_logic_vector(WindSiz downto 0);
signal WrBlockingxDI  : std_logic;

signal RdAddrxDI      : std_logic_vector(AddrSiz downto 0);
signal RdCommandxDI   : std_logic_vector(CommandSiz downto 0);
signal RdDataxDI      : std_logic_vector(DataSiz downto 0);
signal RdWindSizexDI  : std_logic_vector(WindSiz downto 0);
signal RdBlockingxDI  : std_logic;

signal LogicResetxD : std_logic;

-- outputs
signal CBramWrAddrxDO     : std_logic_vector(BramAddrSize downto 0);
signal NBramWrAddrxD      : std_logic_vector(BramAddrSize downto 0);
signal NBramWrAddrLongxD  : std_logic_vector(BramAddrSize+1 downto 0);

signal CBramRdAddrxDO     : std_logic_vector(BramAddrSize downto 0);
signal NBramRdAddrxD      : std_logic_vector(BramAddrSize downto 0);
signal NBramRdAddrLongxD  : std_logic_vector(BramAddrSize+1 downto 0);

signal CBramWrDataxDO     : std_logic_vector(DataSiz downto 0);
signal NBramWrDataxD      : std_logic_vector(DataSiz downto 0);
signal BramRdDataxD       : std_logic_vector(DataSiz downto 0);
```

```
signal NWrAckxD : std_logic;          -- acknowledge for write port
signal CWrAckxD : std_logic;
signal NRdAckxD : std_logic;          -- acknowledge for read port
signal CRdAckxD : std_logic;

signal NWrToutSupxD       : std_logic;
signal CWrToutSupxDO      : std_logic;
signal NRdToutSupxD       : std_logic;
signal CRdToutSupxDO      : std_logic;

-- processed inputs
signal WriteCommandxD : WfifoWriteCommand;  -- write command (derived from WrCommandxDI)
signal ReadCommandxD  : WfifoReadCommand;   -- read command (derived from RdCommandxDI)

-- states
signal NWrStatexD : WfifoWrState;           -- next write state
signal CWrStatexD : WfifoWrState;           -- current write state

signal NRdStatexD : WfifoRdState;           -- next read state
signal CRdStatexD : WfifoRdState;           -- current read state

signal NWrWindowVxD : std_logic;         -- write window valid
signal CWrWindowVxD : std_logic;

signal NRdWindowVxD : std_logic;         -- read window valid
signal CRdWindowVxD : std_logic;

signal NWrWindowSizexD : std_logic_vector(WindSiz downto 0);     -- write window size
signal CWrWindowSizexD : std_logic_vector(WindSiz downto 0);

signal NRdWindowSizexD : std_logic_vector(WindSiz downto 0);     -- read window size
signal CRdWindowSizexD : std_logic_vector(WindSiz downto 0);

-- status
signal NWrStatusxD  : std_logic_vector(1 downto 0);  -- status of last write operation
signal CWrStatusxDO : std_logic_vector(1 downto 0);
signal NRdStatusxD  : std_logic_vector(1 downto 0);  -- status of last read operation
signal CRdStatusxDO : std_logic_vector(1 downto 0);

-- current free memory. value does not include read and write windows.
signal CFreeMemD : unsigned(BramAddrSize downto 0);
signal NFreeMemD : unsigned(BramAddrSize downto 0);

-- add this value to mem size (used by read state machine)
signal CAddToMemSizexD   : unsigned(WindSiz downto 0);
signal NAddToMemSizexD   : unsigned(WindSiz downto 0);
-- subtract this value mem fifo size (used by write state machine)
signal CSubFromMemSizexD : unsigned(WindSiz downto 0);
signal NSubFromMemSizexD : unsigned(WindSiz downto 0);

-- current fifo size. value does not include read and write windows.
signal NFifoSizexD : unsigned(BramAddrSize downto 0);
signal CFifoSizexD : unsigned(BramAddrSize downto 0);

-- add this value to fifo size (used by write state machine)
signal CAddToFifoSizexD   : unsigned(WindSiz downto 0);
signal NAddToFifoSizexD   : unsigned(WindSiz downto 0);
-- subtract this value from fifo size (used by read state machine)
signal CSubFromFifoSizexD : unsigned(WindSiz downto 0);
signal NSubFromFifoSizexD : unsigned(WindSiz downto 0);

-- bram write offset
signal CBramWrOffsetxD     : unsigned(BramAddrSize downto 0);
signal NBramWrOffsetxD     : unsigned(BramAddrSize downto 0);
signal NBramWrOffsetLongxD: unsigned(BramAddrSize+1 downto 0);
```

```
-- bram read offset
signal CBramRdOffsetxD     : unsigned(BramAddrSize downto 0);
signal NBramRdOffsetxD     : unsigned(BramAddrSize downto 0);
signal NBramRdOffsetLongxD: unsigned(BramAddrSize+1 downto 0);

signal NBramWrEnablexD : std_logic;      -- write enable (for bram write port only)
signal CBramWrEnablexD : std_logic;

------------------------------------------------------------------------------
-- Begin architecture
------------------------------------------------------------------------------

begin -- architecture IMP

-- input signals
WrAddrxDI      <= Bus2IP_W_AddrxDI(AddrMsb to AddrLsb);
WrCommandxDI   <= Bus2IP_W_AddrxDI(CommandMsb to CommandLsb);
WrDataxDI      <= Bus2IP_W_DataxDI(DataMsb to DataLsb);
WrWindSizexDI <= Bus2IP_W_DataxDI(WindMsb to WindLsb);
WrBlockingxDI <= Bus2IP_W_DataxDI(BlockLsb);

RdAddrxDI      <= Bus2IP_R_AddrxDI(AddrMsb to AddrLsb);
RdCommandxDI   <= Bus2IP_R_AddrxDI(CommandMsb to CommandLsb);
RdDataxDI      <= Bus2IP_R_DataxDI(DataMsb to DataLsb);
RdWindSizexDI <= Bus2IP_R_DataxDI(WindMsb to WindLsb);
RdBlockingxDI <= Bus2IP_R_DataxDI(BlockLsb);

------------------------------------------------------------------------------
-- Function: calcualte WRITE command (filter non valid commands)
-- Typ:      combinatorial
------------------------------------------------------------------------------

 p_WrCommandMemless : process(WrCommandxDI, Bus2IP_W_CSxDI, Bus2IP_W_RNWxDI, LogicResetxD,
                              Bus2IP_W_WrCExDI, Bus2IP_W_RdCExDI) is
  begin

    if  Bus2IP_W_CSxDI = '1' and Bus2IP_W_RNWxDI = '0' and Bus2IP_W_WrCExDI = '1' then
      case WrCommandxDI is
        when ComAcWrite =>
          WriteCommandxD <= s_AcWr;
        when ComReWrite =>
          WriteCommandxD <= s_ReWr;
        when ComWrite   =>
          WriteCommandxD <= s_WrDa;
        when others     =>
          WriteCommandxD <= s_NoWr;        -- check!
      end case;
    elsif Bus2IP_W_CSxDI = '1' and Bus2IP_W_RNWxDI = '1' and Bus2IP_W_RdCExDI = '1' then
      if WrCommandxDI = ComGetStatus then
        WriteCommandxD <= s_StWr;
      else
        WriteCommandxD <= s_NoWr;
      end if;
    else
      WriteCommandxD <= s_NoWr;
    end if;
  end process p_WrCommandMemless;


------------------------------------------------------------------------------
-- Function: calcualte READ command (filter non valid commands)
-- Typ:      combinatorial
------------------------------------------------------------------------------

 p_RdCommandMemless : process(RdCommandxDI, Bus2IP_R_CSxDI, Bus2IP_R_RNWxDI,
                              LogicResetxD, Bus2IP_R_WrCExDI, Bus2IP_R_RdCExDI ) is
```

```
begin
  -- default

  if  Bus2IP_R_CSxDI = '1' then
    case RdCommandxDI is

      when ComAcRead =>
        if Bus2IP_R_RNWxDI = '0' and Bus2IP_R_WrCExDI = '1' then
          ReadCommandxD <= s_AcRd;
        else
          ReadCommandxD <=  s_NoRd;
        end if;

      when ComReRead =>
        if  Bus2IP_R_RNWxDI = '0' and Bus2IP_R_WrCExDI = '1' then
          ReadCommandxD <= s_ReRd;
        else
          ReadCommandxD <=  s_NoRd;
        end if;

      when ComRead   =>
        if  Bus2IP_R_RNWxDI = '1' and Bus2IP_R_RdCExDI = '1' then
          ReadCommandxD <= s_RdDa;
        else
          ReadCommandxD <=  s_NoRd;
        end if;

      when ComGetStatus  =>
        if  Bus2IP_R_RNWxDI = '1' and Bus2IP_R_RdCExDI = '1' then
          ReadCommandxD <=  s_StRd;
        else
          ReadCommandxD <=  s_NoRd;
        end if;

      when others      =>
        ReadCommandxD <= s_NoRd;        -- check!
    end case;
  else
    ReadCommandxD <=  s_NoRd;
  end if;
end process p_RdCommandMemless;


-------------------------------------------------------------------------------
-- Function: calcualte next WRITE state based on input and current states
-- Typ:      combinatorial
-- Sensitivity:
--  > WriteCommandxD: new command input (output of WrCommandMemless)
--  > CWrStatexD: internal write state was updated by p_WfifoStateMemzing
--  > LogicResetxD: initialize state variables such as NWrWindowVxD
--  > CWrWindowVxD, CFreeMemD: internal states used to determine the next state
--  > WrWindSizexDI, WrDataxDI, WrBlockingxDI, WrAddrxDI: direct inputs
-------------------------------------------------------------------------------

p_WrStateMemless : process(WriteCommandxD, CWrStatexD, LogicResetxD,
                           CWrWindowVxD, CFreeMemD, CWrStatusxDO, CWrWindowSizexD,
                           WrWindSizexDI, WrDataxDI, WrBlockingxDI, WrAddrxDI) is

begin
  -- default assignments
  NWrWindowVxD <= CWrWindowVxD;
  NWrWindowSizexD <= CWrWindowSizexD;
  NAddToFifoSizexD <= (others => '0');   -- no change of fifo size
  NSubFromMemSizexD <= (others => '0');  -- no change of free mem size
  NWrAckxD <= '0';                       -- no acknowledge
  NBramWrEnablexD  <= '0';               -- bram write disabled
```

```
NBramWrDataxD <= (others => '0');
NWrStatexD <= s_WrIdle;                 -- is not necessary
NWrStatusxD <= CWrStatusxDO;
NWrToutSupxD <= '0';                    -- no timeout supress


-- current state is idle
case CWrStatexD is
  when s_WrIdle  =>
    case WriteCommandxD is

      -- acquire write window
      when s_AcWr =>
        -- no write window (ok), and target size > 0
        if  CWrWindowVxD = '0' and unsigned(WrWindSizexDI) > 0 then
          -- enough free memory
          if CFreeMemD >= unsigned(WrWindSizexDI)  then
            NWrStatexD <= s_AcWrAck;
            NWrAckxD <= '1';
            NWrWindowVxD <= '1';
            NWrWindowSizexD <= WrWindSizexDI;
            NSubFromMemSizexD <= unsigned(WrWindSizexDI);
            NWrStatusxD <= "00";
          -- not enough free memory
          else
            -- blocking
            if WrBlockingxDI = '1' then
              NWrStatexD <= s_AcWrBlk;
              NWrToutSupxD <= '1';
              NWrWindowSizexD <= WrWindSizexDI;
            -- non blocking (ok)
            else
              NWrStatexD <= s_WrErr;
              NWrAckxD <= '1';
              NWrStatusxD <= "10";
            end if;
          end if;
        else                                -- there already is a write window (error)
          NWrStatexD <= s_WrErr;
          NWrAckxD <= '1';
          NWrStatusxD <= "01";
        end if;

      -- release write window
      when s_ReWr =>
        if  CWrWindowVxD = '1' then           -- write window available (ok)
          NWrStatexD <= s_ReWrAck;
          NWrAckxD <= '1';
          NWrWindowVxD <= '0';
          NWrStatusxD <= "00";
          NWrWindowSizexD <= (others => '0');
          -- add window to fifo size
          NAddToFifoSizexD <= unsigned(CWrWindowSizexD);
        else                                  -- no write window available (error)
          NWrStatexD <= s_WrErr;
          NWrAckxD <= '1';
          NWrStatusxD <= "01";
        end if;

      -- write data to window
      when s_WrDa =>
        -- write window available and address within valid range
        if  CWrWindowVxD = '1' and unsigned(WrAddrxDI) < unsigned(CWrWindowSizexD) then
          NWrStatexD <= s_WrDaAck;
          NWrAckxD <= '1';
          NWrStatusxD <= "00";
```

```
            -- enable write to bram
            NBramWrEnablexD <= '1';
            NBramWrDataxD <= WrDataxDI;
          else
            NWrStatexD <= s_WrErr;
            NWrAckxD <= '1';
            NWrStatusxD <= "01";
          end if;

        -- get write status
        when s_StWr =>
          NWrStatexD <= s_WrStAck;
          NWrAckxD <= '1';

        when s_NoWr =>
          NWrStatexD <= s_WrIdle;
        when others  =>
          NWrStatexD <= s_WrIdle;            -- check!
      end case;

    -- current state is blocked
    when s_AcWrBlk =>
      -- now enough space: acquire the window
      if CFreeMemD >= unsigned(CWrWindowSizexD) then
        NWrStatexD <= s_AcWrAck;
        NWrAckxD <= '1';
        NWrWindowVxD <= '1';
        NWrWindowSizexD <= CWrWindowSizexD;
        NSubFromMemSizexD <= unsigned(CWrWindowSizexD);
        NWrStatusxD <= "00";
        NWrToutSupxD <= '1';
      -- still not enough space: continue blocking
      else
        NWrStatexD <= s_AcWrBlk;
        NWrToutSupxD <= '1';
      end if;

    -- current state is not idle or blocked
    when others =>
      NWrStatexD <= s_WrIdle;
  end case;

end process p_WrStateMemless;

-------------------------------------------------------------------------------
-- Function: calcualte next READ state
-- Typ:      combinatorial
-- Sensitivity:
--  > ReadCommandxD: new command input (output of RdCommandMemless)
--  > CRdStatexD: internal read state was updated by p_WfifoStateMemzing
--  > LogicResetxD: initialize state variables such as NRdWindowVxD
--  > CRdWindowVxD, CFifoSizexD: internal states used to determine the next state
--  > RdWindSizexDI, RdDataxDI, RdBlockingxDI, RdAddrxDI: direct inputs
-------------------------------------------------------------------------------

p_RdStateMemless : process(ReadCommandxD, CRdStatexD, LogicResetxD,
                           CRdWindowVxD, CFifoSizexD, CRdWindowSizexD, CRdStatusxDO,
                           RdWindSizexDI, RdDataxDI, RdBlockingxDI, RdAddrxDI) is
begin
  -- default assignments
  NRdWindowVxD <= CRdWindowVxD;
  NRdWindowSizexD <= CRdWindowSizexD;
  NSubFromFifoSizexD <= (others => '0');  -- no change of fifo size
  NAddToMemSizexD <= (others => '0');     -- no change of free mem size
  NRdAckxD <= '0';                        -- no acknowledge
  NRdStatexD <= s_RdIdle;                 -- is not necessary
```

```
NRdStatusxD <= CRdStatusxDO;
NRdToutSupxD <= '0';                    -- no timeout supress

-- current state is idle
case CRdStatexD is
  when s_RdIdle  =>
    case ReadCommandxD is

      -- acquire read window
      when s_AcRd =>
        -- no read window available (ok) and window size > 0
        if  CRdWindowVxD = '0' and unsigned(RdWindSizexDI) > 0 then
          -- engough data for read window
          if CFifoSizexD >= unsigned(RdWindSizexDI)  then
            NRdStatexD <= s_AcRdAck;
            NRdAckxD <= '1';
            NRdWindowVxD <= '1';
            NRdStatusxD <= "00";
            NRdWindowSizexD <= RdWindSizexDI;
            -- sub form fifo size
            NSubFromFifoSizexD <= unsigned(RdWindSizexDI);

          -- not engough data for read window
          else
            -- blocking (ok, block)
            if RdBlockingxDI = '1' then
              NRdStatexD <= s_AcRdBlk;
              NRdToutSupxD <= '1';
              NRdWindowSizexD <= RdWindSizexDI;

            else
            -- non blocking (ok)
              NRdStatexD <= s_RdErr;        -- read window to large
              NRdAckxD <= '1';
              NRdStatusxD <= "10";
            end if;
          end if;
        -- there already is a read window (error) or window size <= 0
        else
          NRdStatexD <= s_RdErr;
          NRdAckxD <= '1';
          NRdStatusxD <= "01";
        end if;

      -- release read window
      when s_ReRd =>
        if  CRdWindowVxD = '1' then          -- read window available (ok)
          NRdStatexD <= s_ReRdAck;
          NRdAckxD <= '1';
          NRdWindowVxD <= '0';
          NRdStatusxD <= "00";
          NRdWindowSizexD <= (others => '0');
          NAddToMemSizexD <= unsigned(CRdWindowSizexD);
        else                                 -- no read window available (error)
          NRdStatexD <= s_RdErr;
          NRdAckxD <= '1';
          NRdStatusxD <= "01";
        end if;

      when s_RdDa =>
        if  CRdWindowVxD = '1' and unsigned(RdAddrxDI) < unsigned(CRdWindowSizexD) then
          NRdStatexD <= s_RdDaOk;
          -- bram read address
        else
          NRdStatexD <= s_RdErr;
          NRdAckxD <= '1';
```

```
            NRdStatusxD <= "01";
          end if;

        -- get read status
        when s_StRd =>
          NRdStatexD <= s_RdStAck;
          NRdAckxD <= '1';

        when s_NoRd =>
          NRdStatexD <= s_RdIdle;
        when others  =>
          NRdStatexD <= s_RdIdle;            -- check!
      end case;

    -- current state is Read OK
    when s_RdDaOk  =>
      NRdStatexD <= s_RdDaAck;
      NRdAckxD <= '1';
      NRdStatusxD <= "00";

    -- current state is blocked
    when s_AcRdBlk =>
      -- now enough data: acquire the window
      if CFifoSizexD >= unsigned(CRdWindowSizexD) then

        NRdStatexD <= s_AcRdAck;
        NRdAckxD <= '1';
        NRdWindowVxD <= '1';
        NRdStatusxD <= "00";
        NRdWindowSizexD <= CRdWindowSizexD;
        -- sub form fifo size
        NSubFromFifoSizexD <= unsigned(CRdWindowSizexD);
        NRdToutSupxD <= '1';

      -- still not enough space: continue blocking
      else
        NRdStatexD <= s_AcRdBlk;
        NRdToutSupxD <= '1';
      end if;

    -- current state is not idle read or blocked
    when others =>
      NRdStatexD <= s_RdIdle;
  end case;

end process p_RdStateMemless;


-------------------------------------------------------------------------------
-- Function: set next state, init states on reset
-- Typ:      sequential
-------------------------------------------------------------------------------

p_WfifoStateMemzing: process (Bus2IP_W_ClkxCI) is

begin
  if Bus2IP_W_ClkxCI'event and Bus2IP_W_ClkxCI = '1' then

    if LogicResetxD = '1' then  -- synchronous reset
      ---------------------------------------------------------------------
      -- Reset values
      ---------------------------------------------------------------------

      -- wfifo states
      CWrStatexD   <= s_WrIdle;
      CRdStatexD   <= s_RdIdle;
```

```
    -- wfifo internal states
    CWrWindowVxD     <= '0';                -- no valid write window
    CRdWindowVxD     <= '0';                -- no valid read window
    CWrWindowSizexD <= (others => '0');    -- initial write window length
    CRdWindowSizexD <= (others => '0');    -- initial read window length
    CFifoSizexD      <= (others => '0');    -- initial fifo size
    CWrStatusxDO     <= "00";               -- status ok
    CRdStatusxDO     <= "00";               -- status ok
    CFreeMemD <= to_unsigned(BramSizeAbs-2,BramAddrSize+1); -- two addresses are not used

    -- data transfer qualifiers
    CWrAckxD <= '0';          -- no acknowledge by default
    CRdAckxD <= '0';
    CWrToutSupxDO <= '0';     -- no timout supress
    CRdToutSupxDO <= '0';

    -- bram signals
    CBramWrAddrxDO <= (others => '0');
    CBramRdAddrxDO <= (others => '0');
    CBramWrOffsetxD <= (others => '0');
    CBramRdOffsetxD <= to_unsigned(BramSizeAbs-1,BramAddrSize+1);
    CBramWrEnablexD <= '0';            -- write disabled
    CBramWrDataxDO <= (others => '0');

    CAddToMemSizexD <= (others => '0');
    CSubFromMemSizexD <= (others => '0');
    CAddToFifoSizexD <= (others => '0');
    CSubFromFifoSizexD <= (others => '0');

  else
    ------------------------------------------------------------------------------
    -- transitions on rising clock edge without reset
    ------------------------------------------------------------------------------


    ------------------------------------------------------------------------------
    -- wfifo states
    ------------------------------------------------------------------------------
    CWrStatexD <= NWrStatexD;
    CRdStatexD <= NRdStatexD;

    CWrWindowVxD <= NWrWindowVxD;
    CRdWindowVxD <= NRdWindowVxD;

    ------------------------------------------------------------------------------
    -- window and fifo sizes
    ------------------------------------------------------------------------------
    CWrWindowSizexD <= NWrWindowSizexD;
    CRdWindowSizexD <= NRdWindowSizexD;

    CFifoSizexD <= NFifoSizexD;
    CFreeMemD <= NFreeMemD;

    CAddToMemSizexD <= NAddToMemSizexD;
    CSubFromMemSizexD <= NSubFromMemSizexD;
    CAddToFifoSizexD <= NAddToFifoSizexD;
    CSubFromFifoSizexD <= NSubFromFifoSizexD;

    ------------------------------------------------------------------------------
    -- BRAM: offset gating
    -- offset is updated whenever a write/read window is released
    ------------------------------------------------------------------------------
    if  NWrStatexD = s_ReWrAck then
      CBramWrOffsetxD <= NBramWrOffsetxD;
    else
```

```
        CBramWrOffsetxD <= CBramWrOffsetxD;
      end if;

      if NRdStatexD = s_ReRdAck then
        CBramRdOffsetxD <= NBramRdOffsetxD;
      else
        CBramRdOffsetxD <= CBramRdOffsetxD;
      end if;

      ------------------------------------------------------------------------
      -- BRAM: address switching
      -- set write/read address to offset while not writing/reading
      ------------------------------------------------------------------------
      if NWrStatexD = s_WrDaAck then
        CBramWrAddrxDO <= NBramWrAddrxD;
      else
        CBramWrAddrxDO <= std_logic_vector(CBramWrOffsetxD);
      end if;

      if NRdStatexD = s_RdDaOk then
        CBramRdAddrxDO <= NBramRdAddrxD;
      else
        CBramRdAddrxDO <= std_logic_vector(CBramRdOffsetxD);
      end if;

      CBramWrEnablexD <= NBramWrEnablexD;
      CBramWrDataxDO <= NBramWrDataxD;

      ------------------------------------------------------------------------
      -- output signals
      ------------------------------------------------------------------------
      CWrAckxD <= NWrAckxD;
      CRdAckxD <= NRdAckxD;

      CWrStatusxDO <= NWrStatusxD;
      CRdStatusxDO <= NRdStatusxD;

      CWrToutSupxDO <= NWrToutSupxD;
      CRdToutSupxDO <= NRdToutSupxD;

    end if;
  end if;
end process p_WfifoStateMemzing;

------------------------------------------------------------------------
-- concurrent and conditional signal assignments for internal usage
------------------------------------------------------------------------

-- next fifo size
NFifoSizexD <= CFifoSizexD + CAddToFifoSizexD - CSubFromFifoSizexD;
-- next mem size
NFreeMemD   <= CFreeMemD + CAddToMemSizexD - CSubFromMemSizexD;


-- next write offset
NBramWrOffsetLongxD <= (CBramWrOffsetxD + unsigned(CWrWindowSizexD))
                       mod to_unsigned(BramSizeAbs, BramAddrSize+2);
NBramWrOffsetxD <= NBramWrOffsetLongxD(BramAddrSize downto 0);

-- next read offset
NBramRdOffsetLongxD <= (CBramRdOffsetxD + unsigned(CRdWindowSizexD))
                       mod to_unsigned(BramSizeAbs, BramAddrSize+2);
NBramRdOffsetxD <= NBramRdOffsetLongxD(BramAddrSize downto 0);

-- next write address
NBramWrAddrLongxD <= std_logic_vector((CBramWrOffsetxD + unsigned(WrAddrxDI))
```

```
                                    mod to_unsigned(BramSizeAbs,BramAddrSize+2));
NBramWrAddrxD <= NBramWrAddrLongxD(BramAddrSize downto 0);

-- next read address (address offset lags by one and points to unused mem location)
NBramRdAddrLongxD <= std_logic_vector((CBramRdOffsetxD + unsigned(RdAddrxDI)+1)
                                    mod to_unsigned(BramSizeAbs,BramAddrSize+2));
NBramRdAddrxD <= NBramRdAddrLongxD(BramAddrSize downto 0);

------------------------------------------------------------------------------
-- output signal connections
------------------------------------------------------------------------------

-- connect signals to bram
Bram_W_WrEnablexDO <= CBramWrEnablexD;
Bram_W_AddrxDO(BramAddrSize downto 0) <= CBramWrAddrxDO(BramAddrSize downto 0);
Bram_R_AddrxDO(BramAddrSize downto 0) <= CBramRdAddrxDO(BramAddrSize downto 0);
Bram_W_AddrxDO(31 downto BramAddrSize+1) <= (others => '0');
Bram_R_AddrxDO(31 downto BramAddrSize+1) <= (others => '0');
Bram_W_DataxDO <= CBramWrDataxDO;

-- transfer qualifier signals
IP2Bus_W_AckxDO          <= CWrAckxD;
IP2Bus_W_ToutSupxDO      <= CWrToutSupxDO;
IP2Bus_R_AckxDO          <= CRdAckxD;
IP2Bus_R_ToutSupxDO      <= CRdToutSupxDO;

-- error signal is not used
IP2Bus_W_ErrorxDO        <= '0';
IP2Bus_R_ErrorxDO        <= '0';

-- data output: switch for get status state, and gate data output with ack state
IP2Bus_W_DataxDO(0 to 31) <= ("00000000000000000000000000000000" & CWrStatusxDO)
                             when CWrStatexD =  s_WrStAck
                             else  (others => '0');

IP2Bus_R_DataxDO(0 to 31) <= ("00000000000000000000000000000000" & CRdStatusxDO)
                             when CRdStatexD =  s_RdStAck
                             else BramRdDataxD;

BramRdDataxD(31 downto 0) <= Bram_R_DataxDI when CRdStatexD = s_RdDaAck
                          else (others => '0');

------------------------------------------------------------------------------
-- reset logic of WFIFO
------------------------------------------------------------------------------

LogicResetxD <= Bus2IP_W_ResetxDI or Bus2IP_R_ResetxDI;


end architecture behavioral;
```

## C.1.3 WFIFO BRAM

```
------------------------------------------------------------------------------
-- Filename:          wfifo_bram.vhd
-- Version:           v1.00a
-- Description:       wfifo BRAM
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

library Unisim;
use Unisim.all;

library wfifo_v1_00_a;
```

```
use wfifo_v1_00_a.all;

------------------------------------------------------------------------------
-- Entity Section
------------------------------------------------------------------------------

entity wfifo_bram is
  generic
  (
    C_BRAM_SIZE    : integer := 8;        -- in KB
    C_BRAM_AWIDTH : integer := 32;
    C_BRAM_DWIDTH : integer := 32
  );
  port
  (
    Wfifo_Bram_ClkxCI       : in  std_logic;
    Wfifo_Bram_RstxRI       : in  std_logic;
    -- write
    Bram_W_AddrxDI       : in  std_logic_vector(C_BRAM_AWIDTH-1 downto 0);
    Bram_W_DataxDI       : in  std_logic_vector(C_BRAM_DWIDTH-1 downto 0);
    Bram_W_WrEnablexDI   : in  std_logic;
    -- read
    Bram_R_AddrxDI       : in  std_logic_vector(C_BRAM_AWIDTH-1 downto 0);
    Bram_R_DataxD0       : out std_logic_vector(C_BRAM_DWIDTH-1 downto 0)
  );

end entity wfifo_bram;

------------------------------------------------------------------------------
-- Architecture Section
------------------------------------------------------------------------------

architecture imp of wfifo_bram is


signal alwaysZero : std_logic_vector(0 downto 0);

------------------------------------------------------------------------------
-- Component Declarations
------------------------------------------------------------------------------
  ------------------------------------------------------------------------------
  -- BRAM for 1 KB Memsize
  ------------------------------------------------------------------------------
  component RAMB16_S36_S36 is
    port
    (
      -- A
      ADDRA : in std_logic_vector(8 downto 0);
      CLKA : in std_logic;
      DIA : in std_logic_vector(31 downto 0);
      DIPA : in std_logic_vector(3 downto 0);
      DOA : out std_logic_vector(31 downto 0);
      DOPA : out std_logic_vector(3 downto 0);
      ENA : in std_logic;
      SSRA : in std_logic;
      WEA : in std_logic;
      -- B
      ADDRB : in std_logic_vector(8 downto 0);
      CLKB : in std_logic;
      DIB : in std_logic_vector(31 downto 0);
      DIPB : in std_logic_vector(3 downto 0);
      DOB : out std_logic_vector(31 downto 0);
      DOPB : out std_logic_vector(3 downto 0);
      ENB : in std_logic;
      SSRB : in std_logic;
      WEB : in std_logic
```

```
    );
end component;

-------------------------------------------------------------------------------
-- BRAM for 4 KB Memsize
-------------------------------------------------------------------------------
component RAMB16_S18_S18 is
  port
  (
    -- A
    ADDRA : in std_logic_vector(9 downto 0);
    CLKA : in std_logic;
    DIA : in std_logic_vector(15 downto 0);
    DIPA : in std_logic_vector(1 downto 0);
    DOA : out std_logic_vector(15 downto 0);
    DOPA : out std_logic_vector(1 downto 0);
    ENA : in std_logic;
    SSRA : in std_logic;
    WEA : in std_logic;
    -- B
    ADDRB : in std_logic_vector(9 downto 0);
    CLKB : in std_logic;
    DIB : in std_logic_vector(15 downto 0);
    DIPB : in std_logic_vector(1 downto 0);
    DOB : out std_logic_vector(15 downto 0);
    DOPB : out std_logic_vector(1 downto 0);
    ENB : in std_logic;
    SSRB : in std_logic;
    WEB : in std_logic
   );
end component;

-------------------------------------------------------------------------------
-- BRAM for 8 KB Memsize
-------------------------------------------------------------------------------

component RAMB16_S9_S9 is
  port
  (
    -- A
    ADDRA : in std_logic_vector(10 downto 0);
    CLKA : in std_logic;
    DIA : in std_logic_vector(7 downto 0);
    DIPA : in std_logic_vector(0 downto 0);
    DOA : out std_logic_vector(7 downto 0);
    DOPA : out std_logic_vector(0 downto 0);
    ENA : in std_logic;
    SSRA : in std_logic;
    WEA : in std_logic;
    -- B
    ADDRB : in std_logic_vector(10 downto 0);
    CLKB : in std_logic;
    DIB : in std_logic_vector(7 downto 0);
    DIPB : in std_logic_vector(0 downto 0);
    DOB : out std_logic_vector(7 downto 0);
    DOPB : out std_logic_vector(0 downto 0);
    ENB : in std_logic;
    SSRB : in std_logic;
    WEB : in std_logic
  );
end component;


-------------------------------------------------------------------------------
begin
-------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
-- 1 KB
-------------------------------------------------------------------------------

KB1 : if C_BRAM_SIZE = 1  generate

  ramb16_s36_s36_0 : RAMB16_S36_S36
   port map (
     -- A (write)
     ADDRA => Bram_W_AddrxDI(8 downto 0),
     CLKA  => Wfifo_Bram_ClkxCI,
     DIA   => Bram_W_DataxDI(31 downto 0),
     DIPA  => (others => '0'),
     DOA   => open,
     DOPA  => open,
     ENA   => '1',
     SSRA  => Wfifo_Bram_RstxRI,
     WEA   => Bram_W_WrEnablexDI,
     -- B (read)
     ADDRB => Bram_R_AddrxDI(8 downto 0),
     CLKB  => Wfifo_Bram_ClkxCI,
     DIB   => (others => '0'),
     DIPB  => (others => '0'),
     DOB   => Bram_R_DataxD0(31 downto 0),
     DOPB  => open,
     ENB   => '1',
     SSRB  => Wfifo_Bram_RstxRI,
     WEB   => '0'
     );

end generate;

-------------------------------------------------------------------------------
-- 4 KB
-------------------------------------------------------------------------------

KB4 : if C_BRAM_SIZE = 4  generate

  ramb16_s18_s18_0 : RAMB16_S18_S18
   port map (
     -- A (write)
     ADDRA => Bram_W_AddrxDI(9 downto 0),
     CLKA  => Wfifo_Bram_ClkxCI,
     DIA   => Bram_W_DataxDI(15 downto 0),
     DIPA  => (others => '0'),
     DOA   => open,
     DOPA  => open,
     ENA   => '1',
     SSRA  => Wfifo_Bram_RstxRI,
     WEA   => Bram_W_WrEnablexDI,
     -- B (read)
     ADDRB => Bram_R_AddrxDI(9 downto 0),
     CLKB  => Wfifo_Bram_ClkxCI,
     DIB   => (others => '0'),
     DIPB  => (others => '0'),
     DOB   => Bram_R_DataxD0(15 downto 0),
     DOPB  => open,
     ENB   => '1',
     SSRB  => Wfifo_Bram_RstxRI,
     WEB   => '0'
     );

  ramb16_s18_s18_1 : RAMB16_S18_S18
   port map (
     -- A (write)
     ADDRA => Bram_W_AddrxDI(9 downto 0),
```

```
      CLKA  => Wfifo_Bram_ClkxCI,
      DIA   => Bram_W_DataxDI(31 downto 16),
      DIPA  => (others => '0'),
      DOA   => open,
      DOPA  => open,
      ENA   => '1',
      SSRA  => Wfifo_Bram_RstxRI,
      WEA   => Bram_W_WrEnablexDI,
      -- B (read)
      ADDRB => Bram_R_AddrxDI(9 downto 0),
      CLKB  => Wfifo_Bram_ClkxCI,
      DIB   => (others => '0'),
      DIPB  => (others => '0'),
      DOB   => Bram_R_DataxD0(31 downto 16),
      DOPB  => open,
      ENB   => '1',
      SSRB  => Wfifo_Bram_RstxRI,
      WEB   => '0'
      );

end generate;

-------------------------------------------------------------------------------
-- 8 KB
-------------------------------------------------------------------------------

KB8 : if C_BRAM_SIZE = 8  generate

  ramb16_s9_s9_0 : RAMB16_S9_S9
    port map (
      -- A (write)
      ADDRA => Bram_W_AddrxDI(10 downto 0),
      CLKA  => Wfifo_Bram_ClkxCI,
      DIA   => Bram_W_DataxDI(7 downto 0),
      DIPA  => alwaysZero(0 downto 0),
      DOA   => open,
      DOPA  => open,
      ENA   => '1',
      SSRA  => Wfifo_Bram_RstxRI,
      WEA   => Bram_W_WrEnablexDI,
      -- B (read)
      ADDRB => Bram_R_AddrxDI(10 downto 0),
      CLKB  => Wfifo_Bram_ClkxCI,
      DIB   => "00000000",
      DIPB  => alwaysZero(0 downto 0),
      DOB   => Bram_R_DataxD0(7 downto 0),
      DOPB  => open,
      ENB   => '1',
      SSRB  => Wfifo_Bram_RstxRI,
      WEB   => '0'
      );

  ramb16_s9_s9_1 : RAMB16_S9_S9
    port map (
      -- A (write)
      ADDRA => Bram_W_AddrxDI(10 downto 0),
      CLKA  => Wfifo_Bram_ClkxCI,
      DIA   => Bram_W_DataxDI(15 downto 8),
      DIPA  => alwaysZero(0 downto 0),
      DOA   => open,
      DOPA  => open,
      ENA   => '1',
      SSRA  => Wfifo_Bram_RstxRI,
      WEA   => Bram_W_WrEnablexDI,
      -- B (read)
      ADDRB => Bram_R_AddrxDI(10 downto 0),
```

```
      CLKB  => Wfifo_Bram_ClkxCI,
      DIB   => "00000000",
      DIPB  => alwaysZero(0 downto 0),
      DOB   => Bram_R_DataxD0(15 downto 8),
      DOPB  => open,
      ENB   => '1',
      SSRB  => Wfifo_Bram_RstxRI,
      WEB   => '0'
      );

  ramb16_s9_s9_2 : RAMB16_S9_S9
    port map (
      -- A (write)
      ADDRA => Bram_W_AddrxDI(10 downto 0),
      CLKA  => Wfifo_Bram_ClkxCI,
      DIA   => Bram_W_DataxDI(23 downto 16),
      DIPA  => alwaysZero(0 downto 0),
      DOA   => open,
      DOPA  => open,
      ENA   => '1',
      SSRA  => Wfifo_Bram_RstxRI,
      WEA   => Bram_W_WrEnablexDI,
      -- B (read)
      ADDRB => Bram_R_AddrxDI(10 downto 0),
      CLKB  => Wfifo_Bram_ClkxCI,
      DIB   => "00000000",
      DIPB  => alwaysZero(0 downto 0),
      DOB   => Bram_R_DataxD0(23 downto 16),
      DOPB  => open,
      ENB   => '1',
      SSRB  => Wfifo_Bram_RstxRI,
      WEB   => '0'
      );

  ramb16_s9_s9_3 : RAMB16_S9_S9
    port map (
      -- A (write)
      ADDRA => Bram_W_AddrxDI(10 downto 0),
      CLKA  => Wfifo_Bram_ClkxCI,
      DIA   => Bram_W_DataxDI(31 downto 24),
      DIPA  => alwaysZero(0 downto 0),
      DOA   => open,
      DOPA  => open,
      ENA   => '1',
      SSRA  => Wfifo_Bram_RstxRI,
      WEA   => Bram_W_WrEnablexDI,
      -- B (read)
      ADDRB => Bram_R_AddrxDI(10 downto 0),
      CLKB  => Wfifo_Bram_ClkxCI,
      DIB   => "00000000",
      DIPB  => alwaysZero(0 downto 0),
      DOB   => Bram_R_DataxD0(31 downto 24),
      DOPB  => open,
      ENB   => '1',
      SSRB  => Wfifo_Bram_RstxRI,
      WEB   => '0'
      );

end generate;

alwaysZero <= (others => '0');


end architecture imp;
```

## C.1.4 MPD

```
#-------------------------------------------------------------------------------
# wfifo_v2_1_0.mpd
#-------------------------------------------------------------------------------

BEGIN wfifo

#-------------------------------------------------------------------------------
# Peripheral Options
#-------------------------------------------------------------------------------
OPTION IPTYPE = PERIPHERAL
OPTION EDIF=TRUE

#-------------------------------------------------------------------------------
# Bus Interfaces
#-------------------------------------------------------------------------------
BUS_INTERFACE BUS = WOPB, BUS_STD = OPB, BUS_TYPE = SLAVE
BUS_INTERFACE BUS = ROPB, BUS_STD = OPB, BUS_TYPE = SLAVE

#-------------------------------------------------------------------------------
# Parameters
#-------------------------------------------------------------------------------
PARAMETER c_w_baseaddr         = 0xFFFFFFFF, DT = std_logic_vector, BUS = WOPB,
 MIN_SIZE = 0x04
PARAMETER c_w_highaddr         = 0x00000000, DT = std_logic_vector, BUS = WOPB
PARAMETER c_r_baseaddr         = 0xFFFFFFFF, DT = std_logic_vector, BUS = ROPB,
 MIN_SIZE = 0x04
PARAMETER c_r_highaddr         = 0x00000000, DT = std_logic_vector, BUS = ROPB

PARAMETER c_opb_awidth       = 32,          DT = integer
PARAMETER c_opb_dwidth       = 32,          DT = integer

PARAMETER c_wfifo_memsize    = 8, DT = integer, VALUES = (1= 1  , 4= 4  , 8= 8)
PARAMETER c_wfifo_windwidth = 11,       DT = integer
PARAMETER c_wfifo_bramwidth = 11,       DT = integer
PARAMETER c_wfifo_pipelinemode  = 5, DT = integer, VALUES = (2= 2  , 3= 3  , 5= 5, 7= 7)

#-------------------------------------------------------------------------------
# Write WFIFO Port
#
#     entity            bus
#-------------------------------------------------------------------------------
PORT opb_W_abus     = OPB_ABus,     DIR = IN, VEC = [0:(c_opb_awidth-1)],      BUS = WOPB
PORT opb_W_be       = OPB_BE,       DIR = IN, VEC = [0:((c_opb_dwidth/8)-1)], BUS = WOPB
PORT opb_W_clk      = "",           DIR = IN,                                  BUS = WOPB,
 SIGIS = CLK
PORT opb_W_dbus     = OPB_DBus,     DIR = IN, VEC = [0:(c_opb_dwidth-1)],      BUS = WOPB
PORT opb_W_rnw      = OPB_RNW,      DIR = IN,                                  BUS = WOPB
PORT opb_W_rst      = OPB_Rst,      DIR = IN,                                  BUS = WOPB
PORT opb_W_select   = OPB_select,   DIR = IN,                                  BUS = WOPB
PORT opb_W_seqaddr  = OPB_seqAddr,  DIR = IN,                                  BUS = WOPB
PORT sln_W_dbus     = Sl_DBus,      DIR = OUT, VEC = [0:(c_opb_dwidth-1)],     BUS = WOPB
PORT sln_W_errack   = Sl_errAck,    DIR = OUT,                                 BUS = WOPB
PORT sln_W_retry    = Sl_retry,     DIR = OUT,                                 BUS = WOPB
PORT sln_W_toutsup  = Sl_toutSup,   DIR = OUT,                                 BUS = WOPB
PORT sln_W_xferack  = Sl_xferAck,   DIR = OUT,                                 BUS = WOPB

#-------------------------------------------------------------------------------
# Read WFIFO Port
#-------------------------------------------------------------------------------
PORT opb_R_abus     = OPB_ABus,     DIR = IN, VEC = [0:(c_opb_awidth-1)],      BUS = ROPB
PORT opb_R_be       = OPB_BE,       DIR = IN, VEC = [0:((c_opb_dwidth/8)-1)], BUS = ROPB
PORT opb_R_clk      = "",           DIR = IN,                                  BUS = ROPB,
 SIGIS = CLK
PORT opb_R_dbus     = OPB_DBus,     DIR = IN, VEC = [0:(c_opb_dwidth-1)],      BUS = ROPB
```

```
PORT opb_R_rnw      = OPB_RNW,      DIR = IN,                                    BUS = ROPB
PORT opb_R_rst      = OPB_Rst,      DIR = IN,                                    BUS = ROPB
PORT opb_R_select   = OPB_select,   DIR = IN,                                    BUS = ROPB
PORT opb_R_seqaddr  = OPB_seqAddr,  DIR = IN,                                    BUS = ROPB
PORT sln_R_dbus     = Sl_DBus,      DIR = OUT, VEC = [0:(c_opb_dwidth-1)],       BUS = ROPB
PORT sln_R_errack   = Sl_errAck,    DIR = OUT,                                   BUS = ROPB
PORT sln_R_retry    = Sl_retry,     DIR = OUT,                                   BUS = ROPB
PORT sln_R_toutsup  = Sl_toutSup,   DIR = OUT,                                   BUS = ROPB
PORT sln_R_xferack  = Sl_xferAck,   DIR = OUT,                                   BUS = ROPB

END
```

### C.1.5 PAO

```
#-----------------------------------------------------------------------------
# wfifo_v2_1_0.pao
#-----------------------------------------------------------------------------

lib proc_common_v1_00_b proc_common_pkg
lib proc_common_v1_00_b pselect
lib proc_common_v1_00_b or_muxcy

lib ipif_common_v1_00_c ipif_pkg
lib ipif_common_v1_00_c interrupt_control
lib ipif_common_v1_00_c ipif_steer

# lib opb_ipif_v3_00_a     reset_mir
# lib opb_ipif_v3_00_a     opb_bam
# lib opb_ipif_v3_00_a     opb_ipif

lib opb_myipif_v3_00_a     opb_myipif

lib opb_ipif_wfifo_v1_00_a   opb_ipif_wfifo

# wfifo

lib wfifo_v1_00_a wfifo_bram
lib wfifo_v1_00_a wfifo_logic
lib wfifo_v1_00_a wfifo
```

## C.2 OPB Recorder

```
-------------------------------------------------------------------------------
-- opb recorder: record WFIFO bus transactions for testing
-------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library unisim;
use unisim.all;

library opb_recorder_v1_00_a;
use opb_recorder_v1_00_a.all;

entity opb_recorder is
  generic (
    C_OPB_AWIDTH : integer                       := 32;
    C_OPB_DWIDTH : integer                       := 32;
    C_BASEADDR   : std_logic_vector(0 to 31)     := X"FFFF_8000";
    C_HIGHADDR   : std_logic_vector              := X"FFFF_80FF";
    C_WFIFO_R_BASE : std_logic_vector(0 to 31) := X"FFFF_FFFF";
    C_WFIFO_R_HIGH : std_logic_vector(0 to 31) := X"0000_0000";
```

```
    C_WFIFO_W_BASE : std_logic_vector(0 to 31) := X"FFFF_FFFF";
    C_WFIFO_W_HIGH : std_logic_vector(0 to 31) := X"0000_0000"
    );
  port (
    -- Global signals
    OPB_Clk : in std_logic;
    OPB_Rst : in std_logic;

    -- OPB signals
    OPB_ABus     : in  std_logic_vector(0 to 31);
    OPB_BE       : in  std_logic_vector(0 to 3);
    OPB_RNW      : in  std_logic;
    OPB_select   : in  std_logic;
    OPB_seqAddr  : in  std_logic;
    OPB_DBus     : in  std_logic_vector(0 to 31);

    Sln_DBus     : out std_logic_vector(0 to 31);
    Sln_errAck   : out std_logic;
    Sln_retry    : out std_logic;
    Sln_toutSup  : out std_logic;
    Sln_xferAck  : out std_logic
    );

end entity opb_recorder;


architecture IMP of opb_recorder is

  constant timerSize : integer := 31;       -- timer bit width-1
  constant CommandLsb : integer := 15;
  constant CommandMsb : integer := 12;
  constant timerStep : unsigned(timerSize downto 0) := to_unsigned(1,timerSize+1);

  constant CommandSiz : integer := 3;
  constant ComRead       : std_logic_vector(CommandSiz downto 0) := "0001";
  constant ComWrite      : std_logic_vector(CommandSiz downto 0) := "0010";
  constant ComAcRead     : std_logic_vector(CommandSiz downto 0) := "0011";
  constant ComAcWrite    : std_logic_vector(CommandSiz downto 0) := "0100";
  constant ComReRead     : std_logic_vector(CommandSiz downto 0) := "0101";
  constant ComReWrite    : std_logic_vector(CommandSiz downto 0) := "0110";
  constant ComGetStatus  : std_logic_vector(CommandSiz downto 0) := "0111";

  constant LogIdSiz        : integer                             := 3;
  constant ReadLogId       : std_logic_vector(LogIdSiz downto 0) := "0001";
  constant WriteLogId      : std_logic_vector(LogIdSiz downto 0) := "0010";
  constant AcReadLogId     : std_logic_vector(LogIdSiz downto 0) := "0011";
  constant AcWriteLogId    : std_logic_vector(LogIdSiz downto 0) := "0100";
  constant ReReadLogId     : std_logic_vector(LogIdSiz downto 0) := "0101";
  constant ReWriteLogId    : std_logic_vector(LogIdSiz downto 0) := "0110";
  constant StatusOkLogId   : std_logic_vector(LogIdSiz downto 0) := "0111";
  constant StatusErrLogId  : std_logic_vector(LogIdSiz downto 0) := "1000";

  signal CtimerxD : unsigned(timerSize downto 0);
  signal NtimerxD : unsigned(timerSize downto 0);

  signal CcommandxD : std_logic_vector(CommandSiz downto 0);

  signal AddrInRdRangexD  : std_logic;
  signal AddrInWrRangexD  : std_logic;
  signal AddrInMyRangexD  : std_logic;
  -- last value of AddrInMyRangexD
  signal LastAddrInMyRangexD  : std_logic;
  -- is high for one clock cycle if address in range was detected
  signal AddrCSxD  : std_logic;
```

```
  signal COpbABusxD   : std_logic_vector(0 to 31);
  signal COpbRNWxD    : std_logic;
  signal COpbDBusxD   : std_logic_vector(0 to 31);

  signal NBramAddrWrxD       : std_logic_vector(10 downto 0);
  signal CBramAddrWrxD       : std_logic_vector(10 downto 0);

  signal NBramAddrRdxD       : std_logic_vector(10 downto 0);
  signal CBramAddrRdxD       : std_logic_vector(10 downto 0);

  signal CBramRdItemCntxD  : std_logic;
  signal NBramRdItemCntxD  : std_logic;

  signal CBramAddrxD         : std_logic_vector(10 downto 0);

  signal BramDataInxD     : std_logic_vector(35 downto 0);
  signal BramDataOutxD    : std_logic_vector(35 downto 0);
  signal BramWrEnablexD   : std_logic;

  signal LogCommandxD     : std_logic_vector(LogIdSiz downto 0);

  type loggerState is  (s_record, s_play, s_init);
  type playerState is  (s_idle, s_hasMoreAck, s_getNextOk, s_getNextAck);

  signal CBusLoggerState : loggerState;
  signal NBusLoggerState : loggerState;

  signal CPlayerState : playerState;
  signal NPlayerState : playerState;

  constant ComInit       : std_logic_vector(1 downto 0) := "00";
  constant ComPlay       : std_logic_vector(1 downto 0) := "01";
  constant ComGetNext    : std_logic_vector(1 downto 0) := "10";
  constant ComHasMore    : std_logic_vector(1 downto 0) := "11";

  signal InputCommandxD : std_logic_vector(1 downto 0);
  signal InputCommandLongxD : std_logic_vector(31 downto 0);

  signal BramHasMoreTokensxD  : std_logic;

  signal CCommandAckxDO  : std_logic;
  signal NCommandAckxD   : std_logic;

  signal CDataOutputxDO  : std_logic_vector(31 downto 0);
  signal NDataOutputxD   : std_logic_vector(31 downto 0);

  signal alwaysZero : std_logic_vector(0 downto 0);

-------------------------------------------------------------------------------
-- Component Declarations
-------------------------------------------------------------------------------

component RAMB16_S36_S36 is
    port
    (
      -- A
      ADDRA : in std_logic_vector(8 downto 0);
      CLKA : in std_logic;
      DIA : in std_logic_vector(31 downto 0);
      DIPA : in std_logic_vector(3 downto 0);
      DOA : out std_logic_vector(31 downto 0);
      DOPA : out std_logic_vector(3 downto 0);
      ENA : in std_logic;
      SSRA : in std_logic;
      WEA : in std_logic;
      -- B
```

```
      ADDRB : in std_logic_vector(8 downto 0);
      CLKB : in std_logic;
      DIB : in std_logic_vector(31 downto 0);
      DIPB : in std_logic_vector(3 downto 0);
      DOB : out std_logic_vector(31 downto 0);
      DOPB : out std_logic_vector(3 downto 0);
      ENB : in std_logic;
      SSRB : in std_logic;
      WEB : in std_logic
    );
end component;

-------------------------------------------------------------------------------
-- Begin architecture
-------------------------------------------------------------------------------

begin

  -------------------------------------------------------------------------------
  -- connections
  -------------------------------------------------------------------------------

  CcommandxD  <= COpbABusxD(CommandMsb to CommandLsb);


  alwaysZero  <= (others => '0');

   ramb16_S36_S36_0 : RAMB16_S36_S36
   port map (
     -- A (used)
     ADDRA => CBramAddrxD(8 downto 0),
     CLKA  => OPB_Clk,
     DIA   => BramDataInxD(31 downto 0),
     DIPA  => BramDataInxD(35 downto 32),
     DOA   => BramDataOutxD(31 downto 0),
     DOPA  => BramDataOutxD(35 downto 32),
     ENA   => '1',
     SSRA  => OPB_Rst,
     WEA   => BramWrEnablexD,
     -- B (not used)
     ADDRB => (others => '0'),
     CLKB  => OPB_Clk,
     DIB   => (others => '0'),
     DIPB  => (others => '0'),
     DOB   => open,
     DOPB  => open,
     ENB   => '0',
     SSRB  => OPB_Rst,
     WEB   => '0'
   );


  -------------------------------------------------------------------------------
  -- command to log
  -------------------------------------------------------------------------------

  p_LogCommand: process (CcommandxD, COpbDBusxD) is
  begin
    case CcommandxD is

      when ComRead =>
        LogCommandxD <= ReadLogId;
      when ComWrite =>
        LogCommandxD <= WriteLogId;
      when ComAcRead =>
        LogCommandxD <= AcReadLogId;
```

```
    when ComAcWrite =>
      LogCommandxD <= AcWriteLogId;
    when ComReRead =>
      LogCommandxD <= ReReadLogId;
    when ComReWrite =>
      LogCommandxD <= ReWriteLogId;
    when ComGetStatus =>
      -- No pipeline register !
      if  OPB_DBus(31) = '0' then
        LogCommandxD <= StatusOkLogId;
      else
        LogCommandxD <= StatusErrLogId;
      end if;
    when others      =>
      LogCommandxD <= (others => '0');

  end case;

end process p_LogCommand;

-------------------------------------------------------------------------------
-- next internal state
-------------------------------------------------------------------------------

p_NextState: process (AddrCSxD, InputCommandxD, OPB_Rst, CBusLoggerState) is
begin

  -- command in my address range
  if AddrCSxD = '1' then
    case InputCommandxD is
      when ComInit =>
        NBusLoggerState <= s_init;
      when ComPlay =>
        if CBusLoggerState = s_record then
          NBusLoggerState <= s_play;
        else
          NBusLoggerState <= CBusLoggerState;
        end if;
      when others =>
        if CBusLoggerState = s_init then
          NBusLoggerState <= s_record;
        else
          NBusLoggerState <= CBusLoggerState;
        end if;
    end case;

  -- no command received
  else
    if CBusLoggerState = s_init then
      NBusLoggerState <= s_record;
    else
      NBusLoggerState <= CBusLoggerState;
    end if;
  end if;

end process p_NextState;

-------------------------------------------------------------------------------
-- next player state
-------------------------------------------------------------------------------
p_NextPlayState: process (AddrCSxD, InputCommandxD, OPB_Rst, CPlayerState) is
begin

  case CPlayerState is
    when s_hasMoreAck =>
      NPlayerState <= s_idle;
```

```
    when s_getNextOk =>
      NPlayerState <= s_getNextAck;
    when s_getNextAck =>
      NPlayerState <= s_idle;
    when s_idle =>

      if AddrCSxD = '1' then
        case InputCommandxD is
          when ComGetNext =>
            NPlayerState <= s_getNextOk;
          when ComHasMore  =>
            NPlayerState <= s_hasMoreAck;
          when others =>
            NPlayerState <= CPlayerState;
        end case;
      else
        NPlayerState <= CPlayerState;
      end if;

    when others =>
      NPlayerState <= CPlayerState;
  end case;
end process p_NextPlayState;


-------------------------------------------------------------------------------
-- memzing
-------------------------------------------------------------------------------

p_BusLogger: process (OPB_Clk) is
begin

    if OPB_Rst = '1' then
      -- reset timer
      CtimerxD <= (others => '0');

      -- reset write and address
      CBramAddrWrxD <= (others => '0');
      CBramAddrRdxD <= (others => '0');

      -- start in recording mode
      CBusLoggerState <= s_record;
      CPlayerState <= s_idle;

      LastAddrInMyRangexD <= '0';
      CCommandAckxDO <= '0';
      CDataOutputxDO <= (others => '0');

      CBramRdItemCntxD <= '0';

    else
      if OPB_Clk'event and OPB_Clk = '1' then

        -- save input
        COpbABusxD <= OPB_ABus;
        COpbRNWxD <= OPB_RNW;
        COpbDBusxD <= OPB_DBus;

        --  update timer
        if CBusLoggerState = s_init then
          CtimerxD <= (others => '0');
        else
          CtimerxD <= NtimerxD;
        end if;

        -------------------------------------------------------------------------------
```

```
        -- BRAM
        ------------------------------------------------------------------------
        -- Write enable
        if CBusLoggerState = s_record then
          BramWrEnablexD <= AddrInRdRangexD or AddrInWrRangexD;
        else
          BramWrEnablexD <= '0';
        end if;

        -- Write address
        if CBusLoggerState = s_record then
          CBramAddrWrxD <= NBramAddrWrxD;
        else
          CBramAddrWrxD <= CBramAddrWrxD;
        end if;

        -- Read address
        if CBusLoggerState = s_play then
          CBramAddrRdxD <= NBramAddrRdxD;
        else
          CBramAddrRdxD <= CBramAddrRdxD;
        end if;

        -- used address
        if CBusLoggerState = s_record then
          CBramAddrxD <= CBramAddrWrxD;
        else
          CBramAddrxD <= CBramAddrRdxD;
        end if;

        -- Data
        BramDataInxD(35 downto 32) <= LogCommandxD;
        BramDataInxD(31 downto 0)  <= std_logic_vector(NtimerxD);

        ------------------------------------------------------------------------
        -- opb output
        ------------------------------------------------------------------------
        CCommandAckxDO <= NCommandAckxD;

        ------------------------------------------------------------------------
        -- update internal state
        CBusLoggerState <= NBusLoggerState;
        CPlayerState <= NPlayerState;
        LastAddrInMyRangexD <= AddrInMyRangexD;
        CDataOutputxDO <= NDataOutputxD;
        CBramRdItemCntxD <= NBramRdItemCntxD;
      end if;
    end if;
end process p_BusLogger;


------------------------------------------------------------------------
-- recorder output
------------------------------------------------------------------------

p_DataOutput: process (NCommandAckxD, NBusLoggerState, NPlayerState, CBramRdItemCntxD) is
begin
  if NCommandAckxD = '0' then
    NDataOutputxD <= (others => '0');
  else
    if NBusLoggerState = s_play then
      case NPlayerState is
          when s_hasMoreAck =>
            NDataOutputxD <= "00000000000000000000000000000000" & BramHasMoreTokensxD;
          when s_getNextAck =>
            if CBramRdItemCntxD = '0' then
```

```
            NDataOutputxD <= BramDataOutxD(31 downto 0);
          else
            NDataOutputxD <= "00000000000000000000000000000000" & BramDataOutxD(35 downto 32);
          end if;
        when others =>
          NDataOutputxD <= (others => '0');
      end case;
    else
      NDataOutputxD <= (others => '0');
    end if;
  end if;
end process p_DataOutput;

-------------------------------------------------------------------------------
-- increment counter
-------------------------------------------------------------------------------
NtimerxD <= CtimerxD + timerStep;

-------------------------------------------------------------------------------
-- increment bram write address if wfifo addr range is left
-------------------------------------------------------------------------------

NBramAddrWrxD <=  std_logic_vector(unsigned(CBramAddrWrxD) + 1) when
                  BramWrEnablexD = '1' and (AddrInRdRangexD or AddrInWrRangexD) = '0'
                  else CBramAddrWrxD;

-------------------------------------------------------------------------------
-- increment bram read addres on getNext state if the current item
-- was read completly
-------------------------------------------------------------------------------
NBramAddrRdxD <=  std_logic_vector(unsigned(CBramAddrRdxD) + 1) when
                  CPlayerState = s_getNextAck and CBramRdItemCntxD = '1'
                  else CBramAddrRdxD;

-------------------------------------------------------------------------------
-- address decoding
-------------------------------------------------------------------------------
AddrInRdRangexD <=  '1' when unsigned(COpbABusxD) >= unsigned(C_WFIFO_R_BASE)
                        and unsigned(COpbABusxD) <= unsigned(C_WFIFO_R_HIGH)
                    else '0';

AddrInWrRangexD <=  '1' when unsigned(COpbABusxD) >= unsigned(C_WFIFO_W_BASE)
                        and unsigned(COpbABusxD) <= unsigned(C_WFIFO_W_HIGH)
                    else '0';

AddrInMyRangexD <=  '1' when unsigned(COpbABusxD) >= unsigned(C_BASEADDR)
                        and unsigned(COpbABusxD) <= unsigned(C_HIGHADDR)
                    else '0';

AddrCSxD <= '1' when LastAddrInMyRangexD = '0' and AddrInMyRangexD = '1'
            else '0';

-------------------------------------------------------------------------------
-- command decoding
-------------------------------------------------------------------------------
InputCommandLongxD <= std_logic_vector(unsigned(COpbABusxD) - unsigned(C_BASEADDR));
InputCommandxD <= InputCommandLongxD(1 downto 0);

-------------------------------------------------------------------------------
-- bram fill level
-------------------------------------------------------------------------------
BramHasMoreTokensxD <=  '1' when CBramAddrRdxD < CBramAddrWrxD
                        else '0';


-- toggle on every read
```

```
  NBramRdItemCntxD <=  (CBramRdItemCntxD xor '1')  when CPlayerState = s_getNextAck
                       else CBramRdItemCntxD;


  ------------------------------------------------------------------------------
  -- unused bus signals
  ------------------------------------------------------------------------------
  Sln_errAck  <= '0';
  Sln_retry   <= '0';
  Sln_toutSup <= '0';

  Sln_DBus <= CDataOutputxDO;


  -- signal acknowledge if a new address in my range was detected
  -- delay ack for get next by one cycle
  NCommandAckxD <=  '1' when (AddrCSxD = '1' and CBusLoggerState /= s_play)
                    or NPlayerState = s_getNextAck
                    or NPlayerState = s_hasMoreAck
                    or (AddrCSxD = '1' and CBusLoggerState = s_play and InputCommandxD = ComInit)
                   else '0';

  Sln_xferAck <= CCommandAckxDO;


end architecture IMP;
```

## C.2.1 MPD

```
BEGIN opb_recorder

## Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION SIM_MODELS = BEHAVIORAL : STRUCTURAL
OPTION USAGE_LEVEL = BASE_USER
OPTION CORE_STATE = ACTIVE
#OPTION IP_GROUP = MICROBLAZE:PPC:LOGICORE:SERIAL
OPTION IP_GROUP = MICROBLAZE:PPC:LOGICORE:LOGGER
OPTION ARCH_SUPPORT = qrvirtex2:qvirtex2:spartan2:spartan2e:spartan3:virtex:virtex2:
virtex2p:virtex4:virtexe

IO_INTERFACE IO_IF = uart_0, IO_TYPE = XIL_UART_V1

## Bus Interfaces
BUS_INTERFACE BUS = SOPB, BUS_STD = OPB, BUS_TYPE = SLAVE

## Generics for VHDL or Parameters for Verilog
PARAMETER C_BASEADDR = 0xFFFFFFFF, DT = std_logic_vector, MIN_SIZE = 0x100
PARAMETER C_HIGHADDR = 0x00000000, DT = std_logic_vector
PARAMETER C_OPB_DWIDTH = 32, DT = integer
PARAMETER C_OPB_AWIDTH = 32, DT = integer

PARAMETER C_WFIFO_R_BASE = 0xFFFFFFFF, DT = std_logic_vector, PERMIT = BASE_USER
PARAMETER C_WFIFO_R_HIGH = 0x00000000, DT = std_logic_vector, PERMIT = BASE_USER
PARAMETER C_WFIFO_W_BASE = 0xFFFFFFFF, DT = std_logic_vector, PERMIT = BASE_USER
PARAMETER C_WFIFO_W_HIGH = 0x00000000, DT = std_logic_vector, PERMIT = BASE_USER

## Ports
PORT OPB_Clk = "", DIR = IN, SIGIS = CLK, BUS = SOPB
PORT OPB_Rst = OPB_Rst, DIR = IN, BUS = SOPB
PORT OPB_ABus = OPB_ABus, DIR = IN, VEC = [0:C_OPB_AWIDTH-1], BUS = SOPB
PORT OPB_BE = OPB_BE, DIR = IN, VEC = [0:C_OPB_DWIDTH/8-1], BUS = SOPB
PORT OPB_RNW = OPB_RNW, DIR = IN, BUS = SOPB
PORT OPB_select = OPB_select, DIR = IN, BUS = SOPB
PORT OPB_seqAddr = OPB_seqAddr, DIR = IN, BUS = SOPB
```

```
PORT OPB_DBus = OPB_DBus, DIR = IN, VEC = [0:C_OPB_DWIDTH-1], BUS = SOPB

PORT Sln_DBus = Sl_DBus, DIR = OUT, VEC = [0:C_OPB_DWIDTH-1], BUS = SOPB
PORT Sln_ErrAck = Sl_errAck, DIR = OUT, BUS = SOPB
PORT Sln_Retry = Sl_retry, DIR = OUT, BUS = SOPB
PORT Sln_ToutSup = Sl_toutSup, DIR = OUT, BUS = SOPB
PORT Sln_XferAck = Sl_xferAck, DIR = OUT, BUS = SOPB

END
```

## C.3 UART Logger

```
-------------------------------------------------------------------------------
-- uartlogger: simulation model for uart lite ip
-------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

use STD.textio.all;

library unisim;
use unisim.all;

library opb_uartlogger_v1_00_a;
use opb_uartlogger_v1_00_a.all;

entity OPB_UARTLOGGER is
  generic (
    C_OPB_AWIDTH : integer                     := 32;
    C_OPB_DWIDTH : integer                     := 32;
    C_BASEADDR   : std_logic_vector(0 to 31) := X"FFFF_8000";
    C_HIGHADDR   : std_logic_vector          := X"FFFF_80FF";
    C_DATA_BITS  : integer range 5 to 8      := 8;
    C_CLK_FREQ   : integer                     := 125_000_000;
    C_BAUDRATE   : integer                     := 19_200;
    C_USE_PARITY : integer                     := 0;
    C_ODD_PARITY : integer                     := 1;
    C_LOG_FILE_NAME: string                    := "uart_logger_1.out"
    );
  port (
    -- Global signals
    OPB_Clk : in std_logic;
    OPB_Rst : in std_logic;

    -- OPB signals
    OPB_ABus     : in std_logic_vector(0 to 31);
    OPB_BE       : in std_logic_vector(0 to 3);
    OPB_RNW      : in std_logic;
    OPB_select   : in std_logic;
    OPB_seqAddr  : in std_logic;
    OPB_DBus     : in std_logic_vector(0 to 31);

    UART_DBus    : out std_logic_vector(0 to 31);
    UART_errAck  : out std_logic;
    UART_retry   : out std_logic;
    UART_toutSup : out std_logic;
    UART_xferAck : out std_logic;

    -- UART signals
    Interrupt : out std_logic;
    RX        : in  std_logic;
    TX        : out std_logic
    );
```

```
end entity OPB_UARTLOGGER;


architecture IMP of OPB_UARTLOGGER is

  signal UartXferAckxDO : std_logic;
  signal InputLetterxDI : std_logic_vector(7 downto 0);

begin
  InputLetterxDI <= OPB_DBus(24 to 31);

  UartXferAckxDO <= '1' when (OPB_ABus(0 to 15) = C_BASEADDR(0 to 15) and OPB_select = '1')
                    else   '0';

  UART_xferAck <= UartXferAckxDO;
  UART_DBus    <= (others => '0');
  UART_errAck  <= '0';
  UART_retry   <= '0';
  UART_toutSup <= '0';
  Interrupt    <= '0';
  TX           <= '0';

  p_UartLogger: process (OPB_Clk) is
      file my_output : TEXT open WRITE_MODE is C_LOG_FILE_NAME;
      variable my_line : LINE;
  begin

      if OPB_Rst = '1' then
        -- nothing
      else
        if OPB_Clk'event and OPB_Clk = '1' and UartXferAckxDO = '1' then

          -- ingnore inputs of value zero
          if (to_integer(signed(InputLetterxDI))/=0) then

            -- write line to file if its a \n
            if (to_integer(signed(InputLetterxDI))=10) then
              -- not suported by xst
              writeline(my_output, my_line);
            else
              -- convert to char and add to line buffer
              -- not suported by xst
              write(my_line, CHARACTER'VAL(to_integer(signed(InputLetterxDI))));
            end if;

          end if;

        end if;
      end if;
  end process p_UartLogger;


end architecture IMP;
```

# D WFIFO Test-Cases

## D.1 Single Bus Architecture

| | |
|---|---|
| wr_1 | Check if a single data item is transferred correctly: Open write window of size one, write 0x12345678 to the window and release the window. Open a read window of size one, read its content and and check if it equal to the value written. |
| wr_2 | Correct read and write of all positions in a window of size six: Open write window of size six, write to the six addresses the values 0xA to 0xF and release the write window. Open a read window of size six, read all elements and check if they equal to the value written |
| wr_3 | Do the same as for singlebus_wr_2 but do it twice. |
| w_err_1 | Error before acquiring: write before acquiring any window and check if error is signalled. |
| w_err_2 | Error after release: write after a write window has been correctly released and check if an error is signalled. |
| w_err_3 | Write outside a correctly acquired write window and check if an error is signalled. |
| w_err_4 | Acquire a new write window before the old one has been released and check if an error is signalled. |
| r_err_1 | Read before acquiering a read window: acquire a write window of size six, release the write window. Read from the WFIFO and check if an error is signalled. |
| r_err_2 | Read after releasing a read window: acquire a write window of size six, release the write window, acquire a read window of size six and release the read window. Read from the WFIFO and check if an error is signalled |
| r_err_3 | Read outside an acquired read window: acquire a write window of size six, release the write window. Then acquire a read window of size six and read address 7 and check if an error is signalled. |
| r_err_4 | Acquire a new read window before releasing the old one: acquire a write window of size six, release the write window, and acquire a read window of size two. Then acquire a read window of size two and check if an error is signalled |

| | |
|---|---|
| r_err_5 | Try to read before doing anything else and check if an error is signalled |
| r_err_6 | Acquire a read window and check if an error is signalled. Read address 0 and check if an error is signalled. Release the read window and check if an error is signalled. |

**Table D.1:** Single bus test cases

## D.2 Dual Bus Architecture

| Name | Function |
|---|---|
| mir_1 | Processor A and B both read the MIR register of the WFIFO and check if the value is correct. The MIR value is written to the uart log file. |
| wr_1 | - Processor A: acquire write window (nonblocking) of size one, write the value 0xAAAAFFFF to it and release the window.<br>- Processor B: acqure read window (blocking) of size one, read the data item and check if its equal to the value written. Release the read window. |
| wr_2 | - Processor A: Same as in wr_1.<br>- Processor B: acqure read window nonblocking of size one. Keep on acquiring until ok is returned. Then read the data item and check if its equal to the value written. Release the read window. Print the number of pollings. |
| wr_3 | - Processor A: acquire write window (nonblocking) of size two, write the value 0xA at position 0 and 0xB at 1 and release the window. Do the same thing again but write now 0xC and 0xD.<br>- Processor B: acqure read window (blocking) of size four and read it in the reverse order. Check if the sequence is equal to 0xD, 0xC, 0xB, 0xA. |
| wr_4 | - Processor A: acquire write window (nonblocking) of maximum size and write offset+7 to each position.<br>- Processor B: acqure read window (blocking) of maximum size and check all values. |
| werr_1 | - Processor A:<br>- Processor B: |

**Table D.2:** Dualbus test cases

# Bibliography


[1] E. A. de Kock, J.-Y. Brunel, K. A. Vissers, P. Lieverse, P. van der Wolf, W. M. Kruijtzer, W. J. M. Smits, and G. Essink. Yapi: Application modeling for signal processing systems. In *Proceedings of 37th Conference on Design Automation (DAC'00)*, pages 402–405.

[2] Stephen A. Edwards and Olivier Tardieu. Shim: A deterministic model for heterogeneous embedded systems. In *Proceedings of the ACM Conference on Embedded Software (Emsoft), Jersey City, NJ, September 2005.*

[3] Om Prakash Gangwal, André Nieuwland, and Paul Lippens. A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. In *Proceedings of ISSS, October 1-3, 2001, Montréal, Québec, Canada.*

[4] Kai Huang and Ji Gu. Automatic platform synthesis and application mapping for multiprocessor systems on-chip. Technical report, Leiden Embedded Research Center, LIACS, Netherlands, 2005.

[5] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress 74, Stockholm, Sweden, August 5-10 1974*, pages 471–475.

[6] Bart Kienhuis. Matparser: An array dataflow analysis compiler. Technical report, University of California at Berkeley, February 2000.

[7] Bart Kienhuis, Edwin Rijpkema, and Ed Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES 2000), May 3 – 5 2000, San Diego, CA, USA.*

[8] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Proceedings of the 7th Int. Conf. Design, Automation and Test in Europe (DATE 2004)*, pages 340–345.

[9] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A compile time based approach for solving out-of-order communication in Kahn Process Networks. In *Proceedings of International Conference on Application Specific Array Processors (ASAP 2002).*

[10] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A hierarchical classification scheme to derive interprocess communication in process networks. In *Proceedings of the 14th IEEE Int. Conf. on Application-specific Systems, Architectures, and Processors (ASAP 2004).*

[11] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Realizations of the extended linearization model in the compaan tool chain. In *Proceeding of the 2th Int. Workshop on Systems, Architectures, Modeling, and Simulation, (SAMOS 2002), July, Samos, Greece, 2002s.*

[12] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A technique to determine interprocess communication in the polyhedral model. In *Proceedings of the 10th Int. Workshop on Compilers for Parallel Computers, (CPC 2003), January, Amsterdam, The , Netherlands 2003.*

[13] Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, Wido Kruijtzer, and Gerben Essink. Design and programming of embedded multiprocessors: An interface-centric approach. In *Proceedings of CODES+ISSS 2004 Stockholm, Sweden.*

[14] Xilinx Inc. *OPB IPIF (v3.01a) Product Specification*, Octobre 2004.

[15] Claudiu Zissulescu-Ianculescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Solving out of order communication using CAM memory; an implementation. In *Proceedings of the 13th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2002), 28-29 November, Utrecht, Netherlands, 2002.*

# List of Figures

# List of Tables