Interface Definition and Instruction Manual.

Date: April 8th, 2017
Version: 1.0

Table of contents

1. Document Control

1.1. Purpose and Scope

This document contains the definition, requirements and functions of a common *interface* integrated with an *IP-core* defined by the user, for any project that may involve multiple IP cores.

1.2. Audience

This document should be read and used by any of the members of the teams that need to implement an IP block while maintaining a common interface.

1.3. Document Organization

Section 1	It contains introductory information including purpose and scope of this document, intended audience, terms definition, version control and a description of every section.
Section 2	It contains the interface definition.
Section 3	It contains the description of how to use the signals established in the interface and the expected response from output ports.
Section 4	It contains Verilog codes for all interface's modules
Section 5	It contains the hardware simulation of the interface using ModelSim Tool.

2. The Interface architecture

2.1. Introduction

This document contains the definition, requirements and functions of the proposed Interface integrated into an IP module (Intellectual Property). The Interface main functionality is to guarantee that standard interface, common to any of the IP blocks developed within a given project, is the same. This allow for an easy management of any of the developed blocks. Therefore, the Interface is integrated with an **IP-core** which is designed and implemented by hardware engineers in order to accomplish specific purposes. Figure 2.1 shows an example of network with connections between some NIcs and IP modules. This could be only possible if all developed IPs share a same interface.

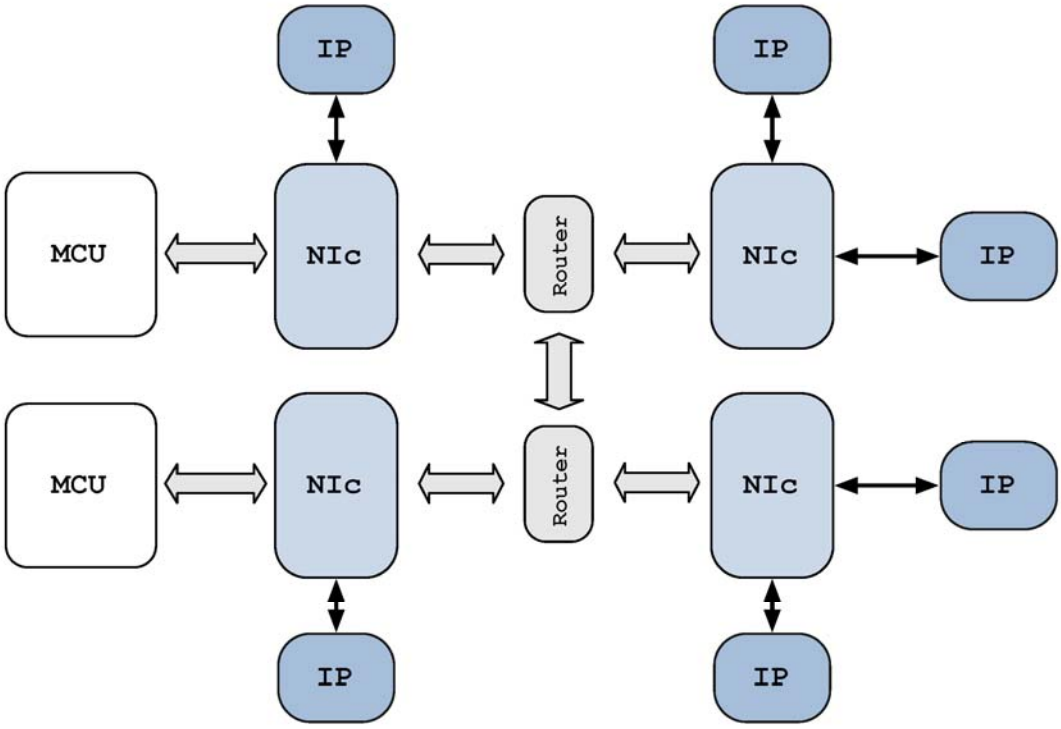

Figure 2.1 Example of a network with connections between several NIc and IP modules

The Interface design establishes a communication protocol between multiples IP modules which were designed to be interconnected in a network. This Interface possesses in its design enough parameterization and reconfigurability, allowing to designers change/update the functional requirements, design, etc., of the IP-core without impact in the communication between IP modules, accessibility and configuration of each IP module as well as starting/stopping of the IP-cores.

As a consequence, these considerations offer several advantages such as the re-use of an IP module by several tasks, algorithms, processes, etc., as well as the design and validation time of a complete system is reduced due to each IP-core that comprise such system, was already validated.

Figure 2.2 shows the black box diagram for an **IP module** and its corresponding inputs and outputs signals. All the IP modules designed by hardware engineers must be provided with the signals denoted as *Mandatory* (colored in black). The signals showed in blue and denoted as *Optional* are given as an example for the hardware engineers in order to connect the IP module with external modules such as Digital/analogue converters, DRAMs, video controllers, analogue mixers or any other peripheral. Therefore, it is important to remark that the optional signals are defined as desired the hardware engineers, however, the mandatory signals remain fixed and cannot be modified.

Table 2.1 describes basic control signals for any IP module. In addition, Table 2.2 describes the mandatory interface inputs and outputs and Table 2.3 describes the optional interface inputs and outputs.

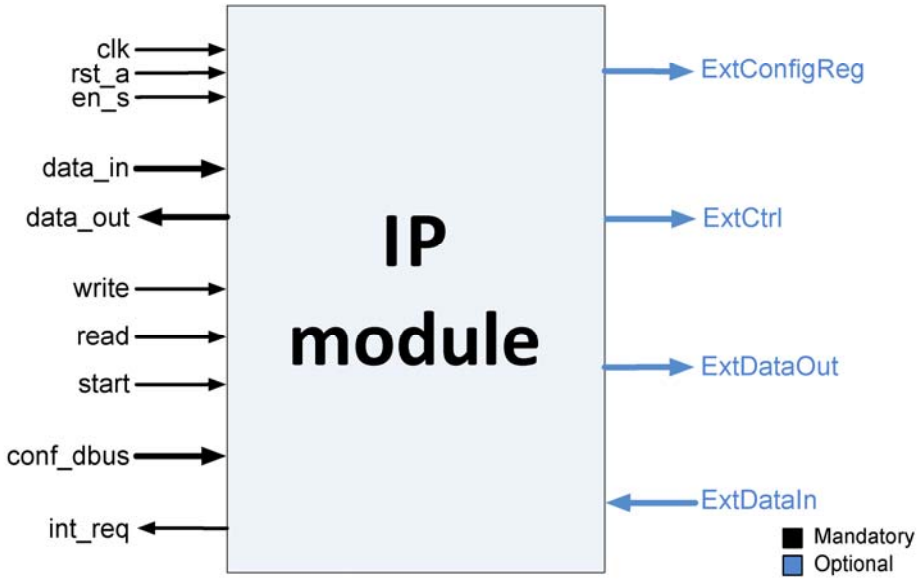

Figure 2.2 Black box diagram with inputs and outputs signals

Bus Name	Direction	Bus	Description
clk	Input	1	Signal Clock of the system
rst_a	Input	1	Reset signal which is active low.
en_s	Input	1	It enables the IP functionality while it is active high. It is a synchronous signal

Table 2.1 Basic control signals

Bus Name	Direction	Bus	Description
data_in	Input	32	Data input for configuration or parameterization of the IP module, and/ or data input for being processed by the IP module.
data_out	Output	32	Data output of the results provided by the IP module, the IP module status or the IP module's ID number.

conf_dbus	Input	5	Selects the internal buses configuration in order to determine the information flow from/to the IP module.
read	Input	1	Enables the reading process from the ***data_out*** port according to the ***conf_dbus*** value.
write	Input	1	Enables the writing process through ***data_in*** por according to the ***conf_dbus*** *value*.
start	Input	1	Initiates the IP module process
int_req	Output	1	Notifies the interruption activation of a IP module's event.

Table 2.2 Generic interface inputs and outputs description

Bus Name	Direction	Bus	Description
ExtConfigReg	Output	UD	Used to configure an external module/system
ExtCtrl	Output	UD	Used to control an external module/system
ExtData	Output	UD	Used to send information to an external module/system
ExtData	Input	UD	Used to obtain information from an external module/system

Table 2.3 Optional inputs and outputs defined by the user (UD).

In Figure 2.3 it is shown an example of a diagram of connection between the Interface and the IP-core. As was mentioned above, some signals are optional and their description dependents on the IP-core definition. In the following sections it is explained how an IP-core is connected to the Interface in accordance to the signals described in Table 2.4.

For a well understanding, in the Section 3 the proposed Interface will be explained by a dummy core example whose function is to transfer input data to an output port after some configurable delay. It is worth to mention that some configuration signals can vary in every IP module. Some variable features are the number of memories, memories sizes, configuration registers, etc. In addition, Appendix A, B and C provide other example of an IP-Core connected to the Interface.

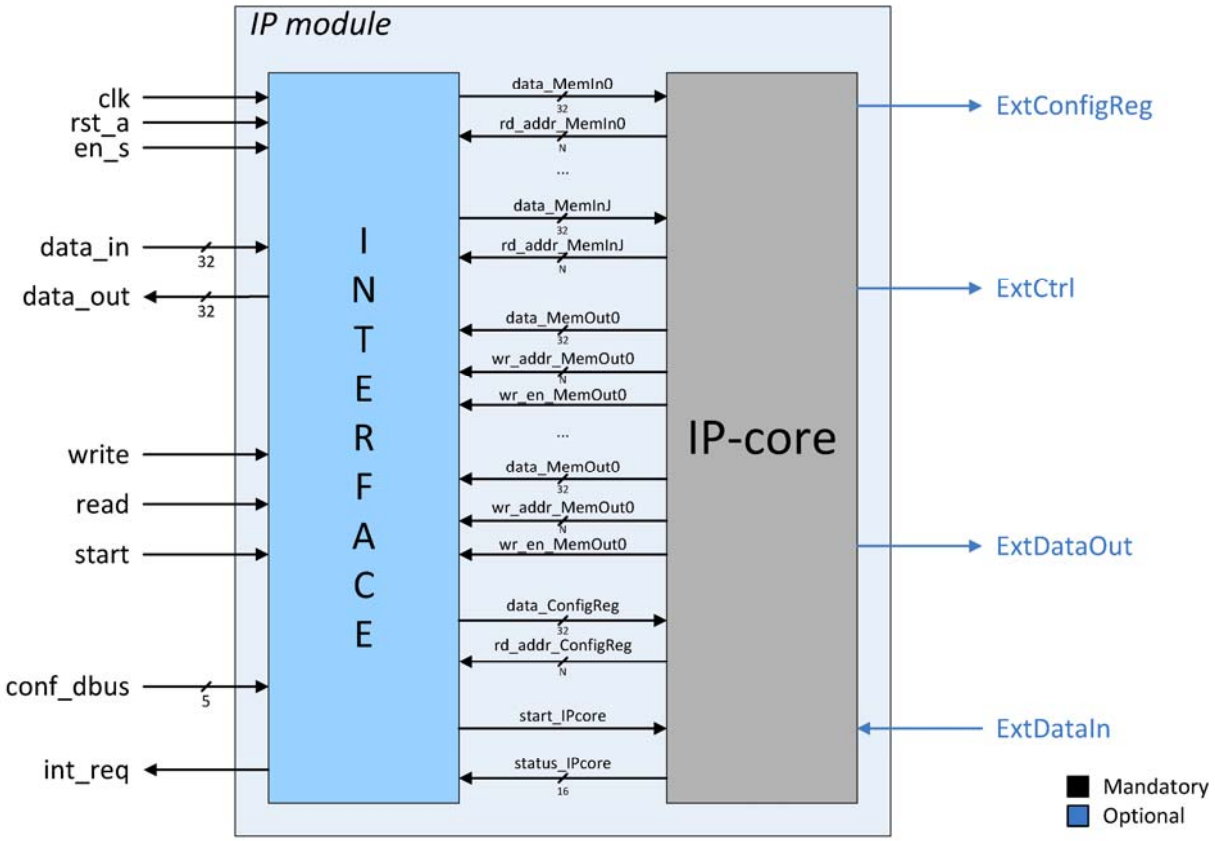

Figure 2.3 Interface and IP-core interconnection example

Bus Name	Bus	Description
data_MemIn0	32	Data input for being processed by the IP-core
rd_addr_MemIn0	6 (UD)	Address bus to read the data from an input memory through the data port ***data_MemIn0***
data_ConfigReg	32	Provides the data of the Configuration Register for the IP-core module
rd_addr_ConfigReg	2 (UD)	Addressing bus for reading the Configuration Register
data_MemOut0	32	Data output which will be used for storing the data provided by the IP-core

wr_addr_MemOut0	6 (UD)	Addressing bus for writing the data provided by the IP-core
wr_en_MemOut0	1	Enables the write operation for storing the data provided by the IP-core
start_IPcore	1	Initiates the IP-core's process.
status_IPcore	15	Contains the IP-core's status.

Table 2.4 Interface and IP-core interconnections signals

2.2. The Interface: Functional Description of the General Architecture

In Figure 2.4, it is shown the general architecture of the proposed Interface. The Interface is composed by a set of Input memories **MemIn0**, **MemIn1**, ..., **MemInJ** and a set of output memories **MemOut0**, **MemOut1**, …, **MemOutK**. Likewise, the Interface has a **Status** module, an Identification module **ID**, a configuration register **ConfigRegister** and control module **Ctrl Interface**.

Basically, the module **Ctrl Interface** controls the data flow of the ports *data_in* and *data_out* according to the signals *write, read* and *conf_dbus.* If the user requires to transfer data to the **IP module**, the **Ctrl Interface** will configure the internal logic in accordance to a value at *conf_dbus,* in order to enable the data flow to the Input memories **MemIn0**, **MemIn1**, ..., **MemInJ**. On the other hand, If the user requires to read data from the **IP module**, the **Ctrl Interface** will configure the internal logic according to value at *conf_dbus,* in order to enable the output data flow from the output memories **MemOut0**, **MemOut1**, …, **MemOutK**.

The aim of the Interface design is to offer a versatile hardware interface in order to rapidly connect any type of IP-core within the hardware Network. This requires to provide an identification number for every **IP module** which is established in the **ID** module. Moreover, in order to take the Interface's advantage of accessibility, it is recommended to use the input memories **MemIn0**, **MemIn1**,..., **MemInJ** and the output memories **MemOut0**, **MemOut1**, …, **MemOutK**, as the main memory blocks of the IP-core. The aim of this recommendation is to access to the IP-cores memory resources at any time for validation and easy parameterization purposes as well as to reduce the amount memory resources of the IP-module.

Furthermore, in the proposed architecture there are memories which can vary in number of blocks used and depth for each block, according to every IP-core requirements.

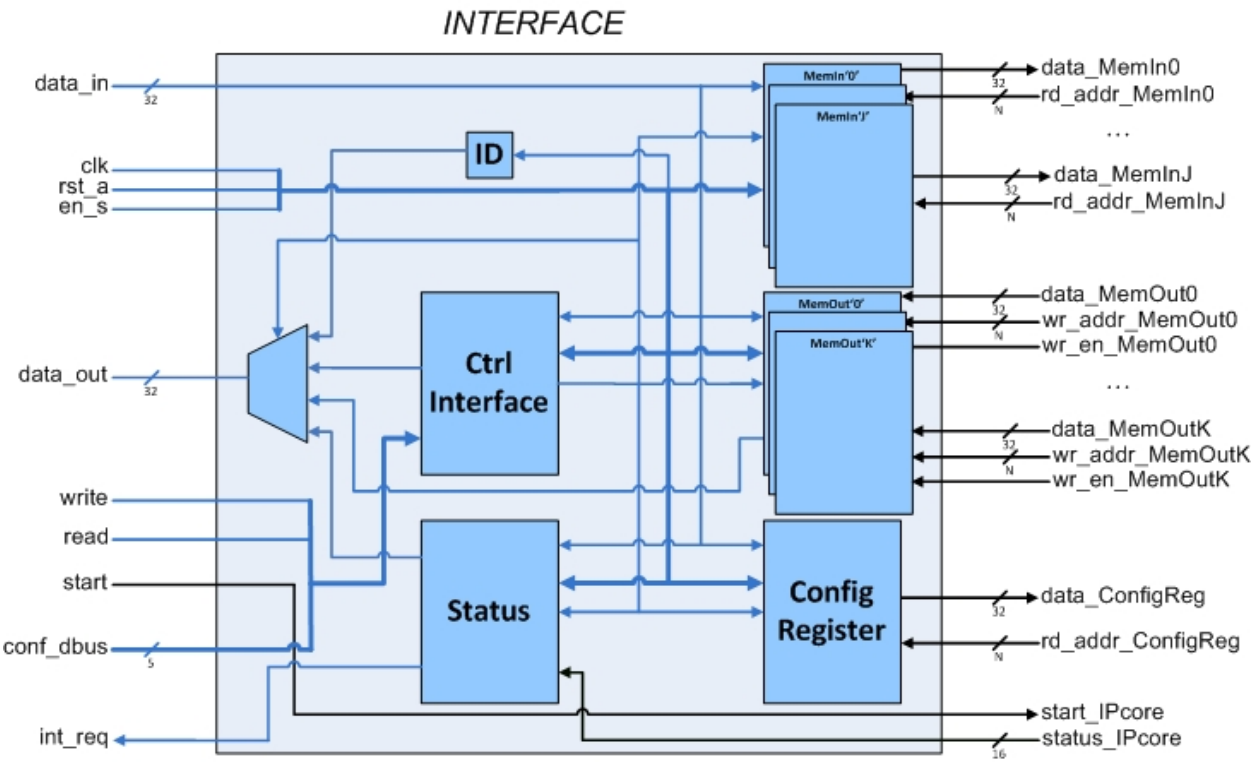

Figure 2.4 General Architecture of the Interface

2.3. Functional Description Ctrl Interface module

This module is responsible for the data flow control of the Interface, i.e., the writing process for input memories and the configuration register as well as the reading process for output memories. This control unit works as a decoder which considers the input signals: *conf_dbus*, *read* and *write* to achieve the data flow control task. The main functional description of **Ctrl Interface** module is already defined, however the IP-core designer should modified the corresponding signals of the input/output memories and the configuration register. These modifications should correspond to the data flow selection provided by the *conf_dbus* values. Nevertheless, there are two default *conf_dbus* values for ID and IP-core Status request which cannot be modified. Figure 2.5 shows **Ctrl Interface** block and Table 2.5 describes the **Ctrl Interface** input and output signals.

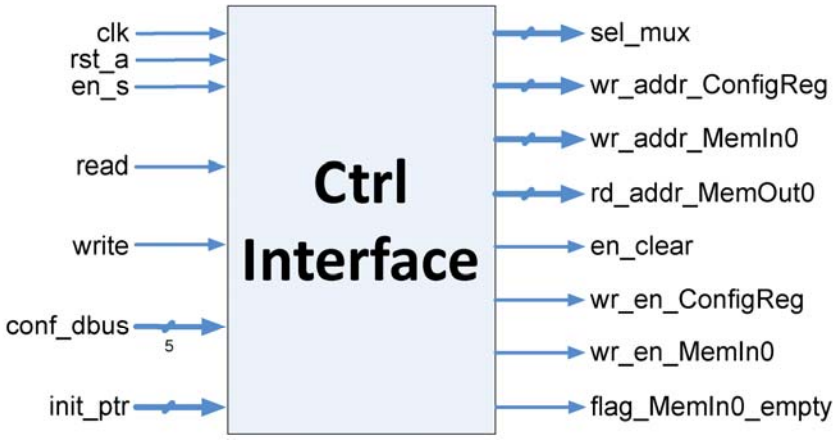

Figure 2.5 Diagram of the Ctrl Interface module

Bus Name	Direction	Bus	Description
clk	Input	1	Clock defined by the core.
rst_a	Input	1	Reset signal. Active low.
en_s	Input	1	It enables the IP functionality. It is a synchronous signal and it is active high
read	Input	1	Enables the reading process from a data source selected according to the *conf_dbus* value.
write	Input	1	Enables the writing process to a resource selected according to the *conf_dbus* value.
conf_dbus	Input	5	It selects the corresponding module to send/receive data by the interface using the write and read signals.
init_ptr	Input	6 (UD)	Set the pointer's initial value for the configuration register and memories according to the *conf_dbus* value.
sel_mux	Output	4 (UD)	It selects the data output according to the *conf_dbus* value.
wr_addr_ConfigReg	Output	1 (UD)	It contains the configuration register pointer
Wr_addr_MemIn0	Output	6 (UD)	It contains the input memory pointer with index equal to '0'.

rd_addr_MemOut0	Output	6 (UD)	It contains the output memory pointer with index equal to '0'.
en_clear	Output	1	enables the interruption flags' cleaning
wr_en_ConfigReg	Output	1	It enables the writing process of configuration data
wr_en_MemIn0	Output	1	It enables the writing process of data for the input memory with index equal to '0'.

Table 2.5 Ctrl Interface inputs and outputs description

2.4. Functional Description ConfigRegister module

The **ConfigRegister** module is a storing resource whose main purpose is to provide special configuration instructions, parameters or any 32-bit word that can enhanced the IP-core operation.

The designer can define the configuration register depth that best suits its needs. The registers of the **ConfigRegister** have a 32-bit width and have some dependency with the data bus *dataIn_ConfigReg* and *dataOut_ConfigReg* from the Interface. The depth is dependent of each IP-core, having as maximum of 16 locations. Figure 2.6 shows Configuration Registers block. Table 2.6 describes the Configuration Registers inputs and outputs.

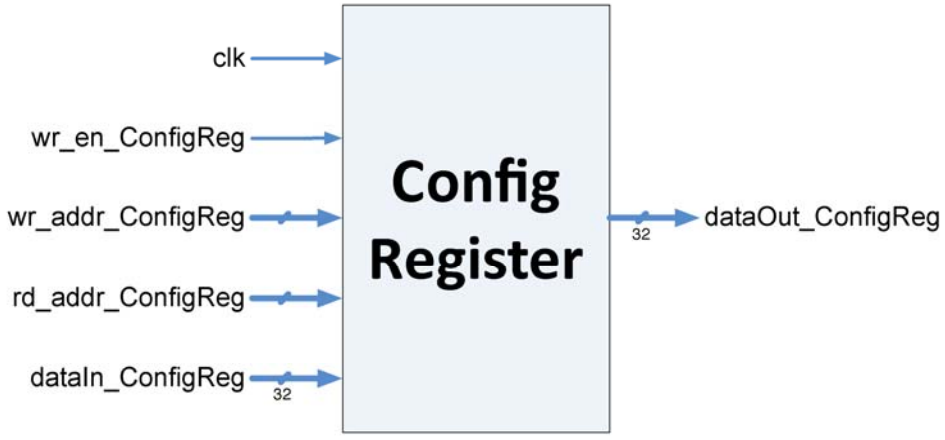

Figure 2.6 Configuration registers block

Bus Name	Direction	Bus	Description

clk	Input	1	It is the clock signal defined by the core.
wr_en_ConfigReg	Input	1	It enables the valid data to be written.
wr_addr_ConfigReg	Input	1 (UD)	It contains the address to be written.
rd_addr_ConfigReg	Input	1 (UD)	It contains the address to be read.
Datain_ConfigReg	Input	32	It contains the data to be written.
dataOut_ConfigReg	Output	32	It contains the data to be read.

Table 2.6 Configuration Registers inputs and outputs description

2.5. The Status Register and Interruption Generation

The purpose of the **Status** module is to register the indicators or any notification of the IP-core operation in order to provide flags and interruptions according to the necessities of the user. Hence, the hardware designer should connect the IP-core indicators of interest, to the *status_IPcore* signal. It is important to note that the IP-core designer should interact only with the *status_IPcore* signal.

Figure 2.7 shows **Status** register module which is a 32-bit width register. The user of the IP module is able to read the status of the IP-core via the *data_status,* and if an interruption is previously configured according to a mask, then it is attended via the *int_req* signal. The *int_req* signal notifies to the user about an IP module request or state change. This functionality helps to determine which of the status flags are active. Table 2.7 describes the Status register inputs and outputs.

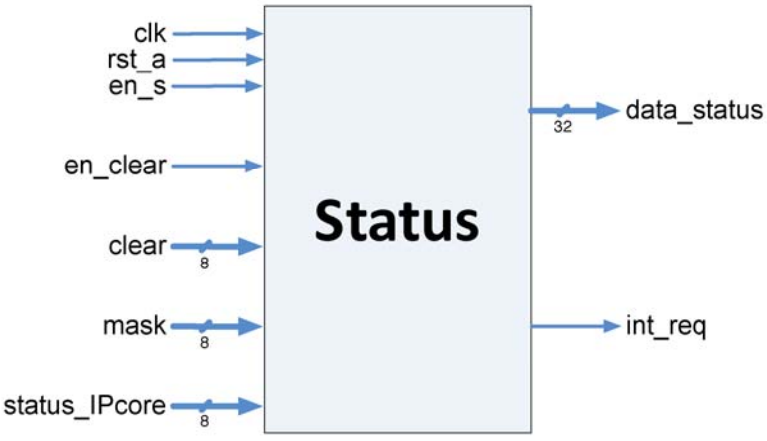

Figure 2.7 Status register block

Bus Name	Direction	Bus	Description
clk	Input	1	Clock defined by the IP-core.
rst_a	Input	1	Reset signal. Active low.
en_s	Input	1	It enables the IP functionality. It is a synchronous signal and it is active high
en_clear	Input	1	It enables the attended flags clearing
clear	Input	16	It contains the flags to be cleared
mask	Input	16	It contains the mask to be applied to the interruptions
status_IPcore	Input	16	It contains the flags' value from the IP-core
data_status	Output	32	It contains the mask and the flags' value of the IP-core
int_req	Output	1	It enables an interruption signal

Table 2.7 The **Status** register inputs and outputs description

Table 2.8 shows the fields of the 32-bit **Status** register which are **Mask** and **Flag**. The **Mask** field is defined by the range 23 to 16 which is determined for being applied to the **Flag** value defined as the 8 lowest significant bits. Likewise, the definition of the **Flag** field is divided by two bytes, the most significant byte, i.e., the 15-8 bits of the **Status** register, determines the *Notifications* which aim is to register any indicator or notification of the IP-core operation that it is not attractive for being enabled as an interruption. On the other hand, the lowest significant byte, i.e., the 7-0 bits of the **Status** register, determines the *Interruptions* which aim is to register any indicator or notification of the IP-core operation that it is critical and should be enabled as an interruption. Both type of flags, *Notifications and Interruptions,* are active high and they are the only fields of the **Status** register in which the IP-core designer should be interacting with, i.e., the IP-core designer should connect the IP-core with the *status_IPcore* signal.

Info	Mask	Flag	
Bits	23-16	15-8 Notifications	7-0 Interruptions

Table 2.8 The arrangement of information of the mask and the flag status

Moreover, during the reading process of the **Status** register via the *data_status* port, the active mask filed and the flag's status are read by the user. Likewise, when the status register is written through the *mask* input signal, the new mask is set with the aim to determine which the interruption flags can generate an interruption. The procedure is as follows: active high the bits of the flags that need to generate an

interruption and active low the bits of the flags that not need to generate an interrupt. Additionally, when the status register is written through the clear input signal and the *en_clear*, the attended flags been cleared. The procedure is as follows: active high the bits of the flags that need to be cleared and active low the bits of the flags that not need to be cleared.

The **Status** register has 16 Flags, four are maintained and the rest flags are defined by the User (DU). As mentioned before, there are interruption flags and notification flags. The first ones are flags that could generate an interruption to be attended for the user, the essential ones are: Done, helps to know when the process of an IP-core has been ended; Data Out Rdy, the user is notified when the data created by IP-core is ready to be read; MemIn Rd Rdy, the user receives a notification when the data to be processed has been read by the IP-core; the designer can create another five interruption flags. On the other hand, there are flags that notify some possible internal IP-core process in an indicative manner, there is only one essential, Busy, used to indicate that the IP-core is performing some process; the designer can create another seven notification flags. Table 2.9 shows the fields of the 16-bit flags registers.

DU	…	Busy	UD	…	MemIn Rd Rdy	Data Out Rdy	Done
15	…	8	7	…	2	1	0

Table 2.9 Structure of Status Register

It is important to clarify that the designer is responsible for activating and deactivating the notification flags. As for the interruption flags, the designer only takes charge the activation, the user is responsible for deactivating them by using the interface.

2.6. Identification module ID

This is a 32 bit-width register. Its functionality is identified cores types in the network. It has two fields: Group and IP Number. Group field contains the 20 MSB (Most Significant Bits) and it makes the difference between Work Groups or IP types in the same branch. The tool developer assigns the Group number. The IP Number is defined with the 12 LSB (Less Significant Bits) and it represents each IP in a group. The workgroup can assign this value. Figure 2.8 shows the diagram of an ID block and Table 2.10 describes the ID inputs and outputs.

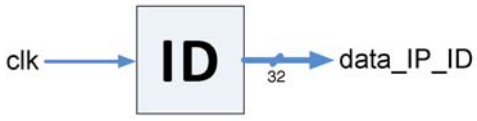

Figure 2.8 ID register block

Bus Name	Direction	Bus	Description
clk	Input	1	Clock defined by the core.

data_IP_ID	Output	32	It shows the core ID value

Table 2.10 ID register inputs and outputs description

Info	Group	IP Number
Bits	31-12	11-0

Table 2.11 The arrangement of information of the ID register

The ID value is a parameter to integrate the core and the interface. Once assigned the ID for a specific IP no changes are allowed. Table 2.11 shows the fields of the 32-bit ID register.

2.7. Input and Output Memories

The designers handle two types of memories: asynchronous and synchronous. To handle large memories blocks the synchronous one are used and with small sizes the asynchronous are used. This interface allows to designer choose the best one for their conventions.

The memory has a 32-bit word width and is related to the data bus size *dataIn_Mem* and *dataOut_Mem*; the width and depth depend on each IP module, having as limit 512 locations. In some cases, is necessary to know the state of the memory during the injection and the reading of the memory, this information could be handled by the Status Register. Figure 2.9 shows the diagram of a Memory block and Table 2.12 describes the Memory inputs and outputs. For more technical details about the Input and Output memories implementation, please see the document named as "***Technical Design Specifications of interface random-access memories***".

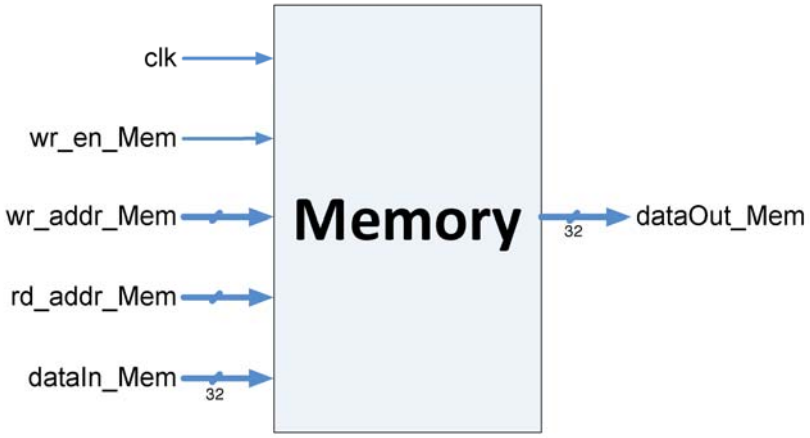

Figure 2.9 Memory block

Bus Name	Direction	Bus	Description
Clk	Input	1	It is the clock signal defined by the core.
wr_en_Mem	Input	1	It enables the valid data to be written.
wr_addr_Mem	Input	6 (UD)	It contains the address to be written.
rd_addr_Mem	Input	6 (UD)	It contains the address to be read.
datain_Mem	Input	32	It contains the data to be written.
dataOut_Mem	Output	32	It contains the data to be read.

Table 2.12 Memory inputs and outputs description

3. How to connect the Interface - The Dummy IP-Core

As was mentioned before, the Interface module offers several advantages like versatility and reconfigurability allowing being adapted to different core needs or requirements. For a well understanding of the interface, this section explains the integration of the Interface with an IP-core of low complexity. The Dummy IP-core is an example whose function is to transfer input data to an output port. Such data transfer can be configured with a certain delay before starting the transferring process.

3.1. The Interface architecture Compliant - Interfacing Considerations

In order to ensure the performance and functionality of the Interface, the IP designers must adapt the Interface design with their IP-cores following the connecting the IP-core to the Interface connections described in Figure 2.4. Moreover, the modules that require the IP designers from the Interface should be implemented following the recommendations of size and signals width.

The IP designers must provide an identification number for every **IP module** which is established in the **ID** module. Moreover, in order to take the Interface's advantage of accessibility, it is recommended to use the input memories **MemIn0**, **MemIn1**,..., **MemInJ** and the output memories **MemOut0**, **MemOut1**, …, **MemOutK**, as the main memory blocks of the IP-core. The aim of this recommendation is to access to the IP-cores memory resources at any time for validation and easy parameterization purposes as well as to reduce the amount memory resources of the IP-module.

It is important to remark that all IP modules designed by hardware engineers must be provided with the signals denoted as *Mandatory* (colored in black). The signals showed in blue and denoted as *Optional* are given as an example for the hardware engineers in order to connect the IP module with external modules such as Digital/analogue converters, DRAMs, video controllers, analogue mixers or any other peripheral. Therefore, it is important to note that the optional signals are defined as desired the hardware engineers, however, the mandatory signals remain fixed and cannot be modified.

3.2. Analyzing the IP-Core

As was mentioned before, the Dummy IP-core is an example whose function is to transfer the input data, stored in a memory, to another block of memory. Figure 3.1 shows the architecture of the Dummy IP-core which is composed by four modules: an input memory **MemoryIn**, an output memory **MemoryOut**, a module for **Configuration** and a control module **IPCtrl**.

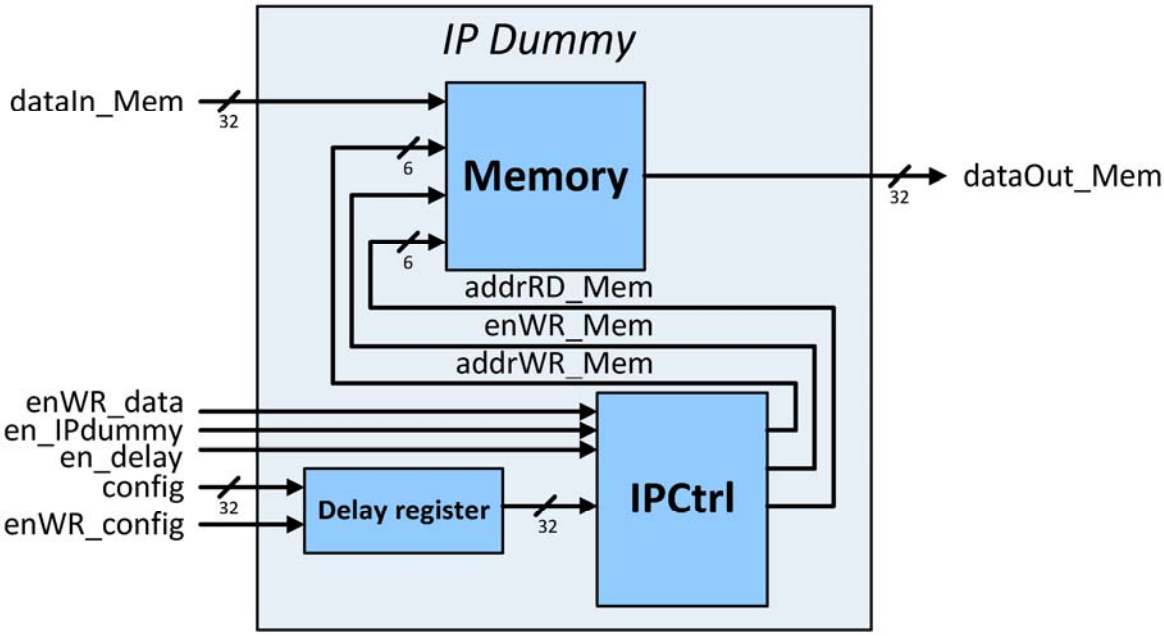

Figure 3.1 The Dummy IP-core interface architecture

The Dummy IP-core has a data memory that allows the information being transferred to the output during the module's enablement or after a certain amount of time. Also, there is a 32-bit register in the **configuration** module, which stores the delay value in milliseconds. Finally, the control block **IPCtrl** is responsible for performing the transferring data functionality.

Based on the above, it will begin to form the type of interface to use. The IP-dummy requires to receive data, therefore at least one input memory is needed. According to the type and amount of receiving data, it is needed only one input memory. In addition, at least one output memory is necessary to extract information, under the same considerations of type and amount of data for the input memory; the conclusion is to use a single output memory. Last, there is a delay register and a delay enabling signal,

these parameters can been set in the configuration register with the same functionality. As result of this analysis, Figure 3.2 shows the signals needed to interact with the interface and the IP-dummy.

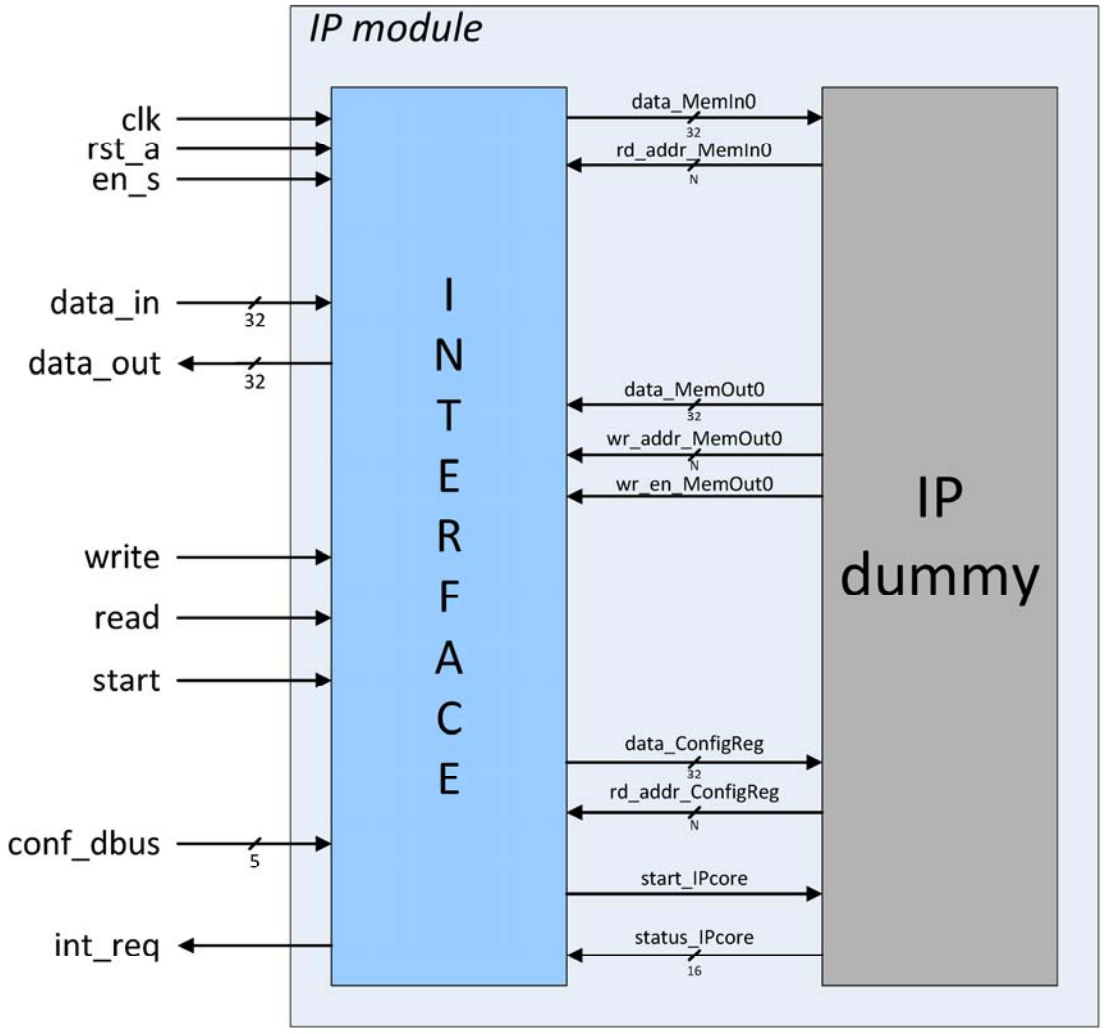

Figure 3.2 Diagram of the Dummy IP-core connection with the interface architecture

3.3. Interfacing the Input and Output Memories with the IP-core

Within the interface will be allocated the memory modules and the configuration register. Therefore, the *ctrl_interface* module has to be modified in order to consider the memories addressing, enabling signals for writing and reading as well as selection signals of the *data_out* multiplexor. Figure 3.3 shows the Interface architecture defined for IP-dummy.

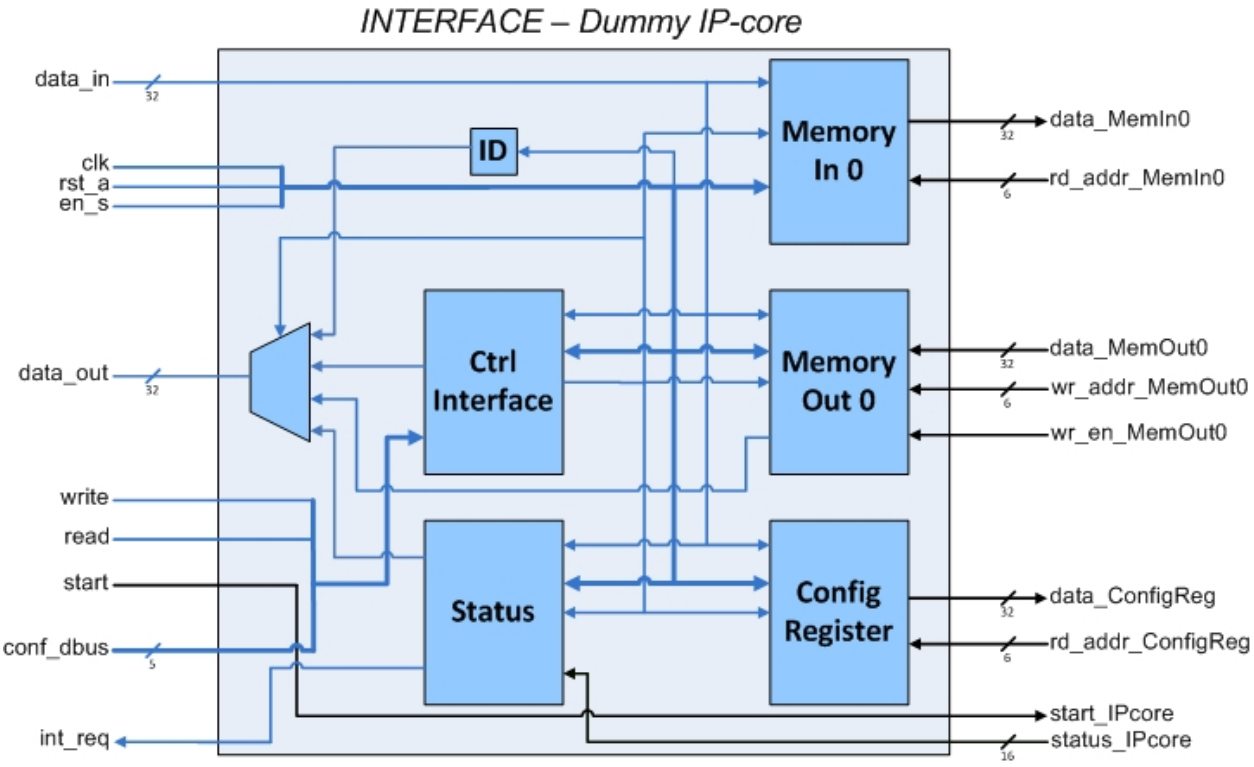

Figure 3.3 The Dummy IP-core Interface architecture

For this case, only one control module will remain on the IP-core side, which will control the reading access of the input memory **MemoryIn0** and the writing access of the output memory **MemoryOut0** as well as the configuration register **ConfigRegister** to perform the required functionality. Furthermore, it is responsible for establishing the IP status indicators, this will be discussed later. Figure 3.4 shows the final IP-core architecture connected to the interface for IP-dummy.

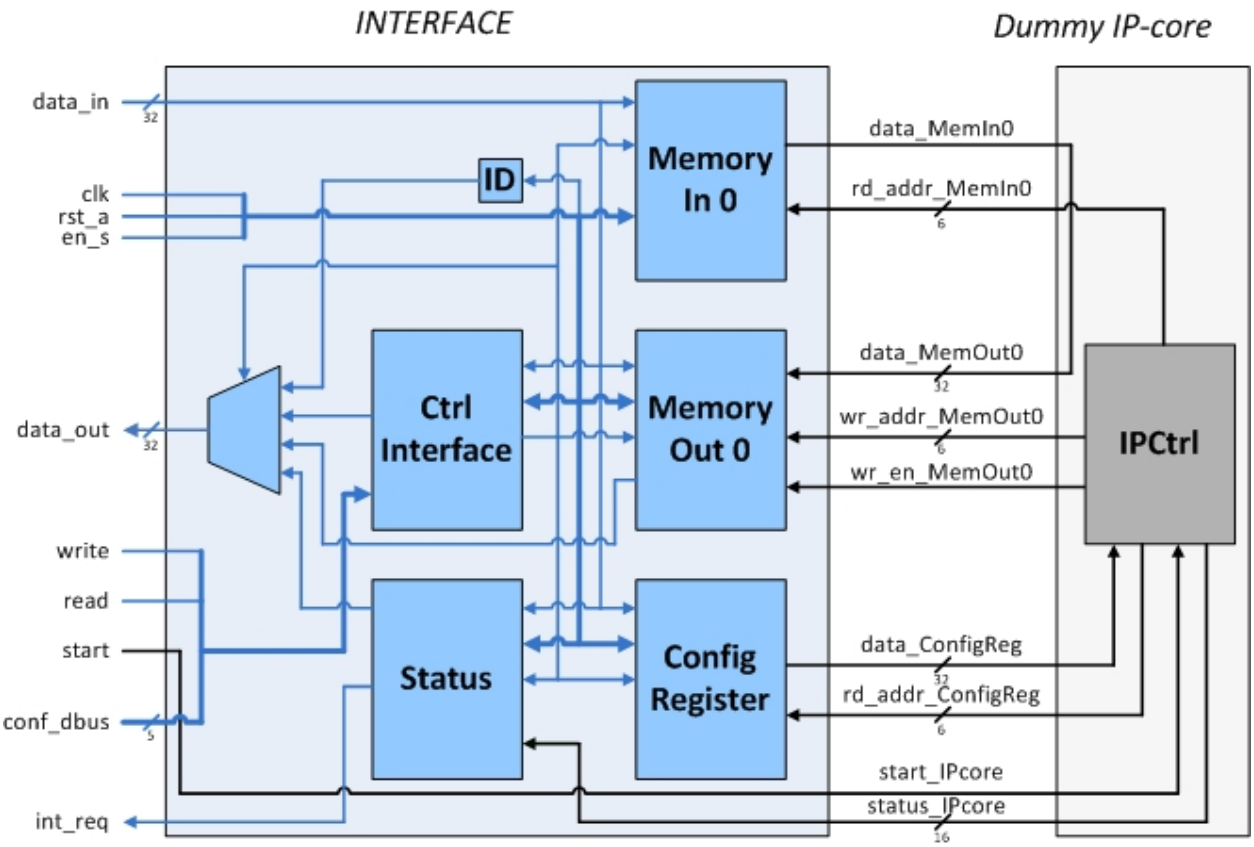

Figure 3.4 Final architecture of the Dummy IP-core with the Interface

3.4. Using the Configuration Register

For this example, it is used two registers, the first one enables the use of a delay and the second one determines the delay time in milliseconds that will be used. In case of having more working modes, it can be used more registers from the **ConfigRegister** module.

The Input Memories, Output Memories and Configuration Registers have the same operating mode for reading and writing process. The procedure for writing mode is: to define the data to be written, consequently, to establish the pointer value and finally, to set the writing enable. On the other hand, in the reading mode the order to handle the signal is: to read the pointer value and then, to read the data.

As a need to use a single data out bus (32 bits) for different sources, a multiplexer is used to achieve this goal. The ports zero and one are used for ID value and IP-core status respectively. Therefore, the designer is able to define the remaining ports. It is important to remark that the **Ctrl Interface** module should be modified properly with respect to the mux-selector in order to avoid errors in data transmission.

3.5. Porting the IP-core indicators and interruptions to the Status Register

During the process of adapting the IP-core to the interface, it is necessary to notify certain state indications in which the IP-core is operating; or to send interruptions, due to some IP-core's event, for being attended by the user.

The indicators and interruptions are grouped in the *status_IPcore* signal as mentioned in the **Status** module. Every IP-core must have the *Busy* indicator in the state flags, the signals *Done*, *DataRdy* and *MemRdRdy* must be found inside the interruption signals; in case of need more indicators or interruptions, the remaining seven flags and the other five for interruptions can be used.

3.6. Assembling the final IP Module and generating the CSV Description file

Finally, it is important to define the user interaction with the IP-dummy interface. The user will be able to perform several actions such as *reading* or *writing* the memories and the configuration register through the *conf_dbus* signal. Other actions that the user can carry out is set the pointer of the memories or the configuration register, either to make changes only to certain values or to read values in specific addresses. Moreover, it can be accessed to read the **Status** registers and the IP identifier. Hence, the IP identifier value is unique for any type of IP-core.

Depending on the final characteristics of the Interface connected with an IP-core, it is necessary to establish the connection between the configuration of the *conf_dbus* signal with the value corresponding to the *reading* or *writing* the memories, *reading* or *writing* memory pointers and the configuration register. Table 3.1 shows the *conf_dbus* definition for IP-dummy case study.

Value	Selection
00000	Configuration Register
00001	Memory In 0
00010	Memory Out 0
00011	Address Config Reg pointer
00100	Address Mem In 0 pointer
00101	Address Mem Out 0 pointer
...	Defined by the IP-core owner
11110	Status register
11111	Identification register

Table 3.1 Commands to access the IP Dummy module through the ***conf_dbus*** signal

In order to let the user know how the IP module is used with the Interface, the designer must generate a CSV file which defines all the *conf_dbus* values and their meaning, in addition to the IP identifier value. This type of file stores tabular data in plain text where each line is a data record. A record consists of one or more fields separated by commas. The following definition lines are the IP-dummy CSV.

```
1. type,alphaID,numID,size,mode,description
```

```
2. IPID,IPDummySPS,0x00001001,Behavioral IP Dummy Model
3. CFG,CONFREG,0b00000,2,W,IP Configuration Register
4. CFG,MEMIN,0b00001,64,W,IP Data In Memory
5. CFG,MEMOUT,0b00010,64,R,IP Data Out Memory
6. CFG,CRADDRSWR,0b00011,1,W,Set Write Address Pointer of Configuration Register
7. CFG,MINADDRSWR,0b00100,1,W,Set Write Address Pointer of Data In Memory
8. CFG,MOUTADDRSRD,0b00101,1,W,Set Read Address Pointer of Data Out Memory
```

The first line represents the label of each column; in the second line are the assigned ID and the IPcore description. The following lines establish the *conf_dbus* values defined above; First, the type is defined as CFG, an alphanumeric ID is assigned, followed by the *conf_dbus* value in binary format, the depth of the element is defined (for example memories of 64 locations are used), the operation mode is determined (W for writing and R for reading) and finally a description of the element.

3.7. Simulation all the system

Once the IP-core and interface have been integrated, it is necessary to test the system operation. It is important to know the operation of each interface signals, in order to perform test vectors that behave as wish during the tests.

3.7.1. Reading and Writing by the user

The Input Memories, Output Memories and Configuration Registers have the same operating mode for read and write process.

Write mode, the value to be written it is defined, as well as the *conf_dbus* value that defines where will be written the data. The Write signal is where the data will be written. It is very important to notice that every transition 0 to 1 makes that the pointer increases it value by one. As a note only input memories and configuration register have Write Mode. The procedure to use this operational mode is first define the data to be written, then enables the pointer value and at the end set the write enable.

Read mode, *conf_dbus* signal defines which is the data to be read after this the data is sent through the Read signal. Every signal transition increases the read pointer by one. Output memories have this mode as mandatory, in case of input memories and configuration registers this operational mode is optional.

Earlier, was mentioned that two types of memories can be used, as note their signals should maintain a standardized, regardless of the type of memory. It is necessary to ensure that the write, read and start signals

stay active only one clock cycle.

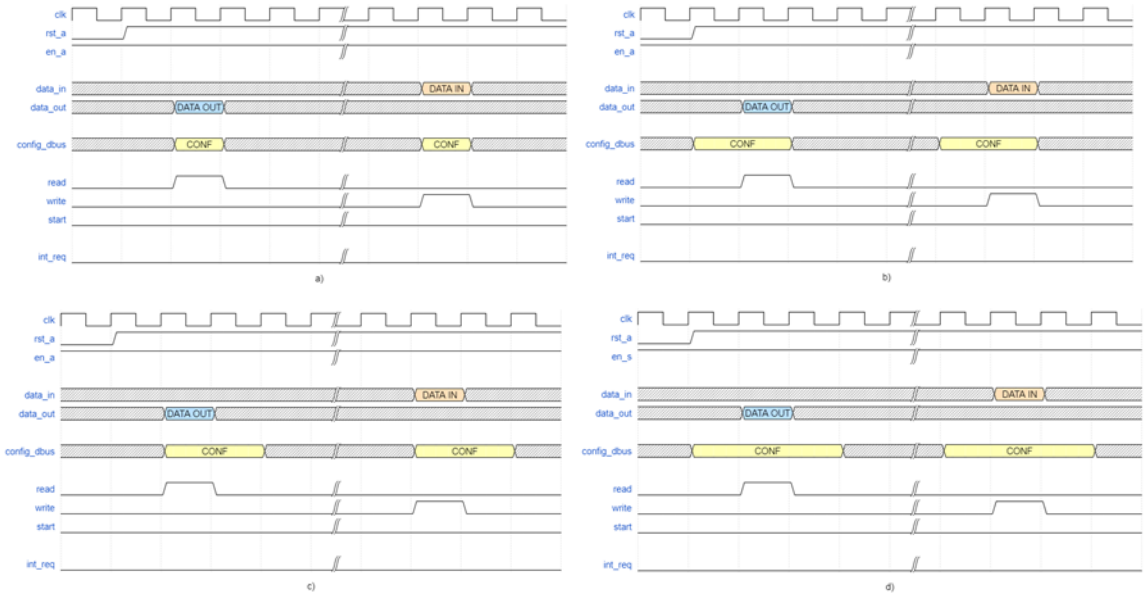

Figure 3.5 shows an example of the above. In

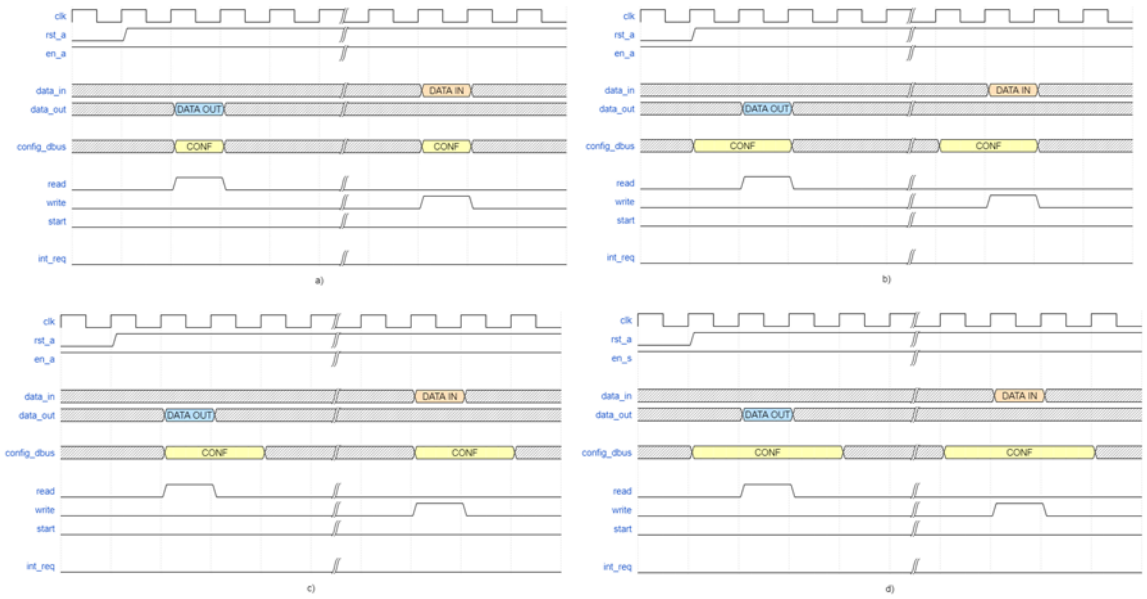

Figure 3.5.a the *config_dbus* value, read and write signals last a clock cycle, in order to perform the required action. In

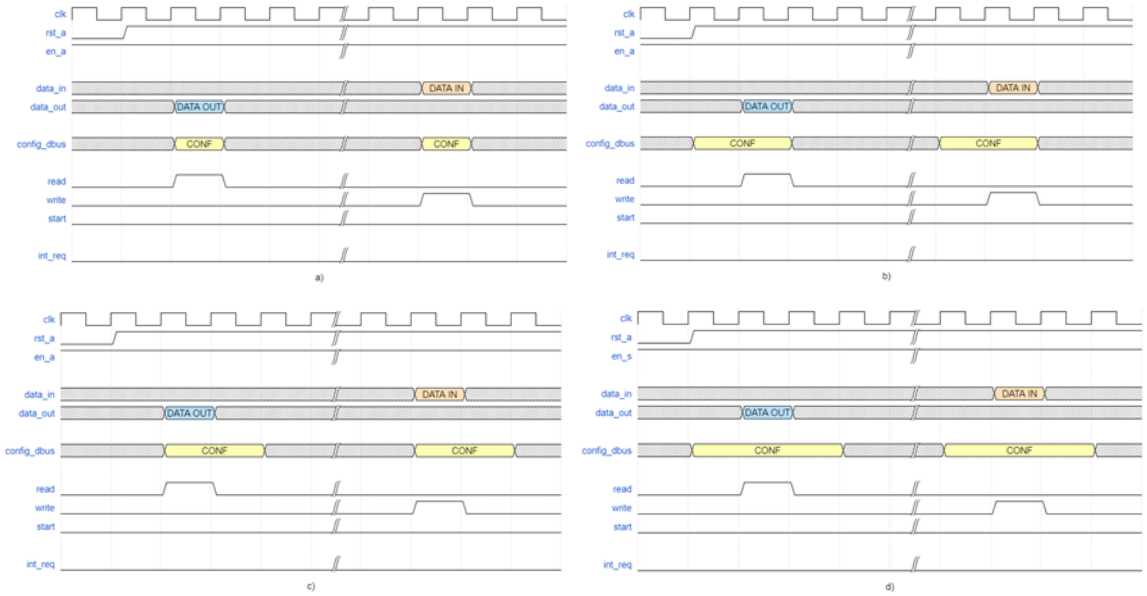

Figure 3.5.b the *config_dbus* value is established one cycle before read and write signal. For

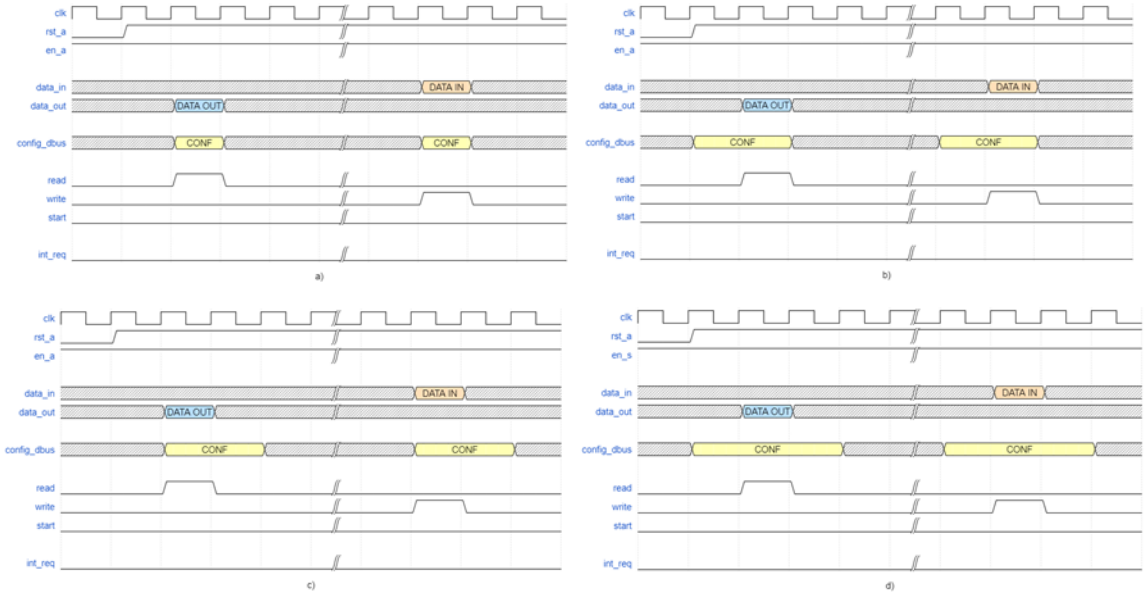

Figure 3.5.c the value of config dbus, read and write signals are established at the same time, but the conf_dbus still remain after that clock cycle. Finally,

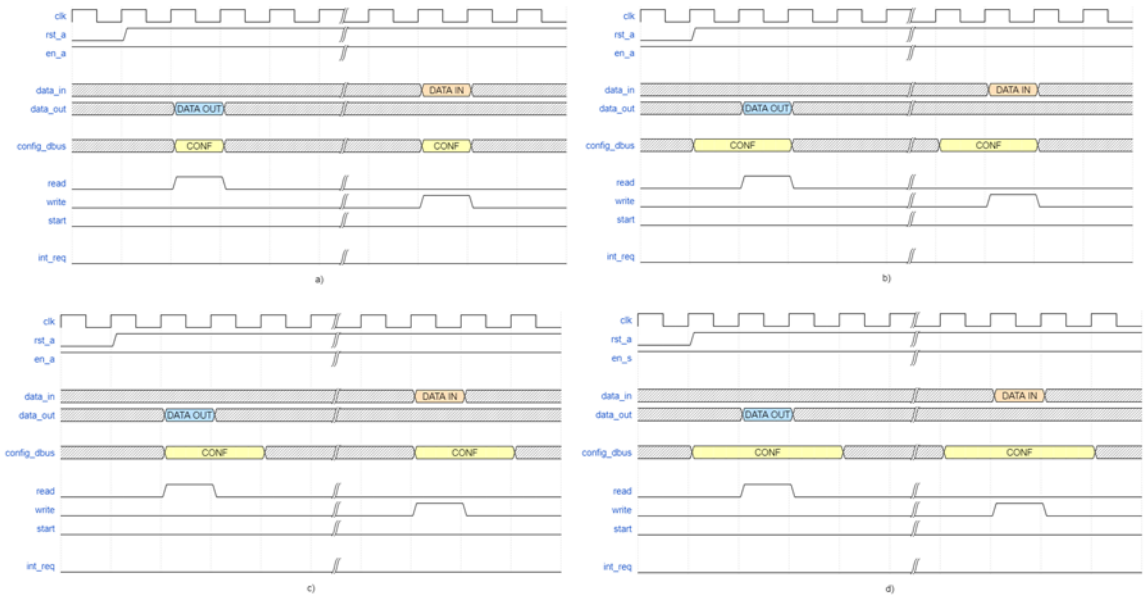

Figure 3.5.d the *config_dbus* value is set one cycle before read and write signal, and the *conf_dbus* remains after the clock cycle lasting read and write signals.

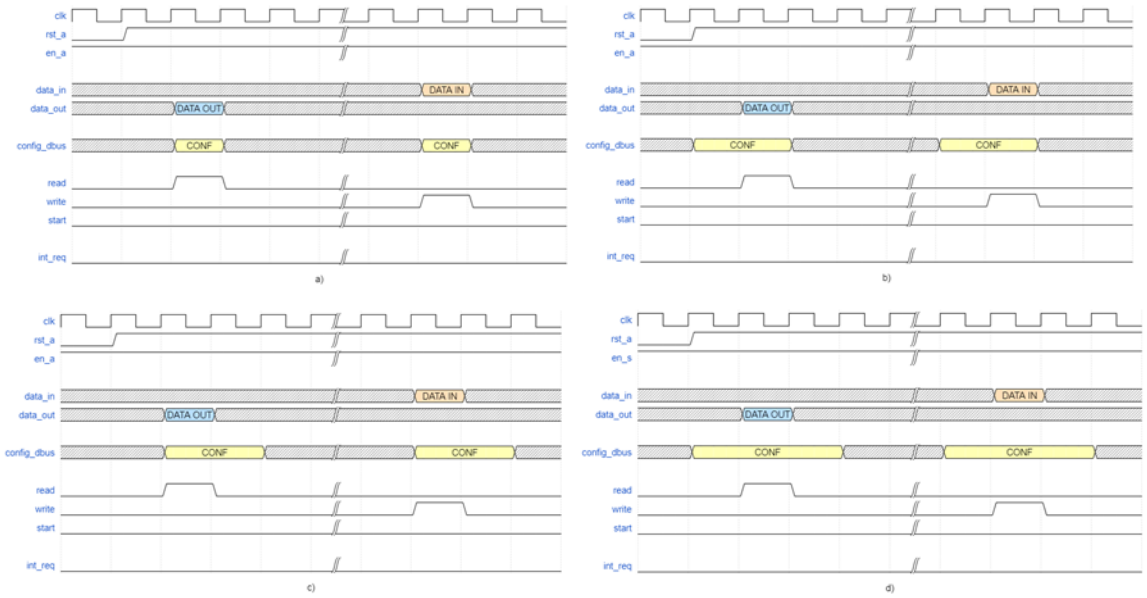

Figure 3.5 Simulation of the signals

3.7.2. Read module identifier

To read the ID register, the *conf_dbus* be set with the value 11111b as shown in Table 3.1. Once the *read* signal set to a high value for one cycle a 32-bit width word is obtained. Figure 3.6 show a waveform with this procedure.

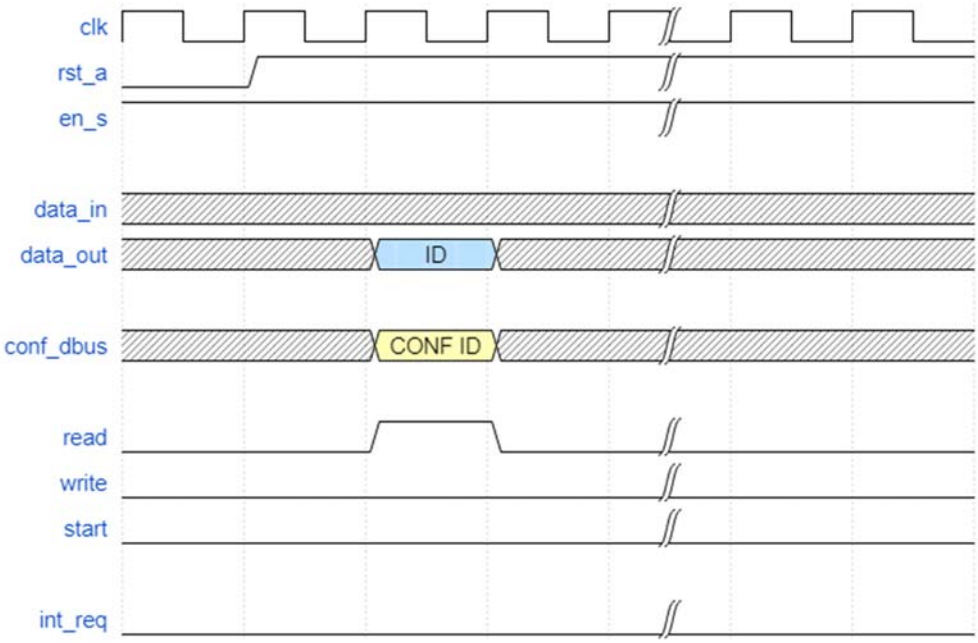

Figure 3.6 Simulation of the signals used for reading module identifier

3.7.3. Status Register

To read or write the status register, *conf_dbus* be set with the value 11110b to select the status register as shown in Table 3.1. Once the *read* signal is set to a high value for one cycle, *data_out* bus obtain the flags value and the enabled mask; only the 24 LSB are considered, see the Section 2.5. Figure 3.7 shows a waveform with this procedure.

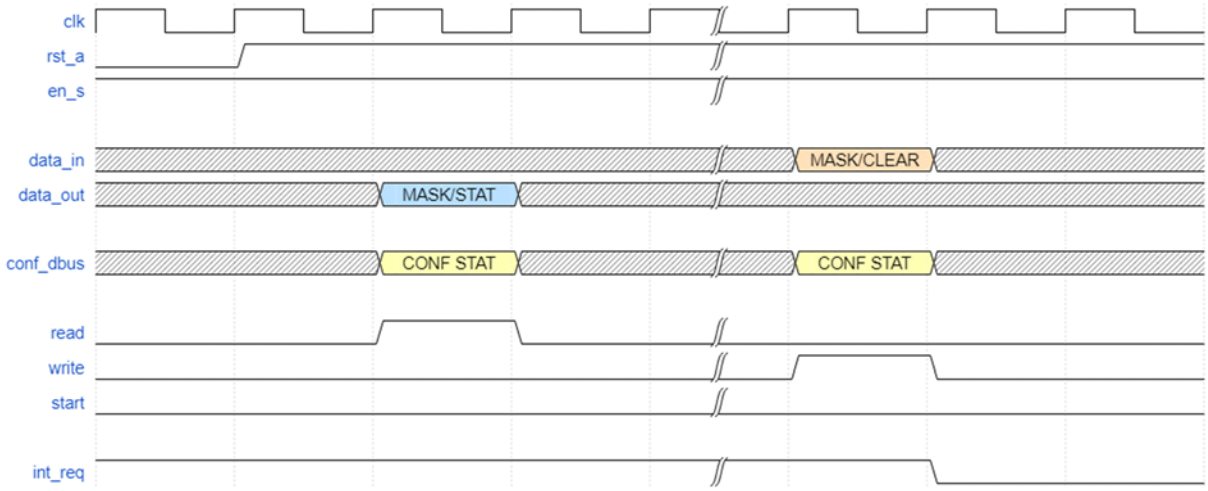

Figure 3.7 Simulation of the signals used for reading and writing status register

Once the *write* control signal is set to a high value for one cycle, the *data_In* value be written in the status register as Figure 3.7 shows in the waveform, see the Section 2.5.

3.7.4. Write data to process

For writing data to the IP Dummy module, *conf_dbus* be set with the value 00001b to select the memory in as shown in Table 3.1. The information is set in the *data_in* bus before enabling the writing process; the *write* signal is set to a high value for one cycle. Each rising edge indicates new writing data. Figure 3.8 shows the behavior of the *write* signal.

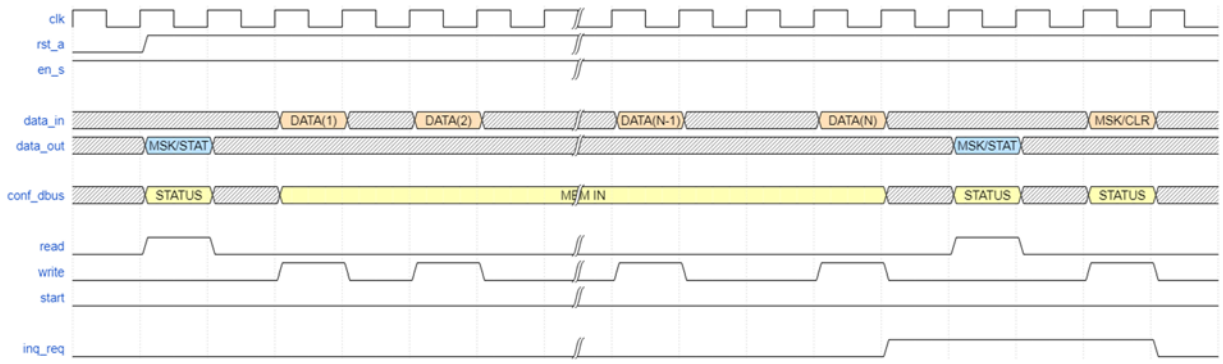

Figure 3.8 Simulation of the signals used for writing data to process

3.7.5. Start IP-core

First, must be read the status register and verify it is not busy, it to determines that the IP-core is not processing information and is able to start a new process. To initiate a new process the *start* signal is set to a high value for one cycle; at that moment the IP-core pass to the busy state and start the IP-dummy process; finally, the done flag, from status register, is active to indicate the process is complete and an interruption can be sent. Figure 3.9 shows the waveform for this procedure.

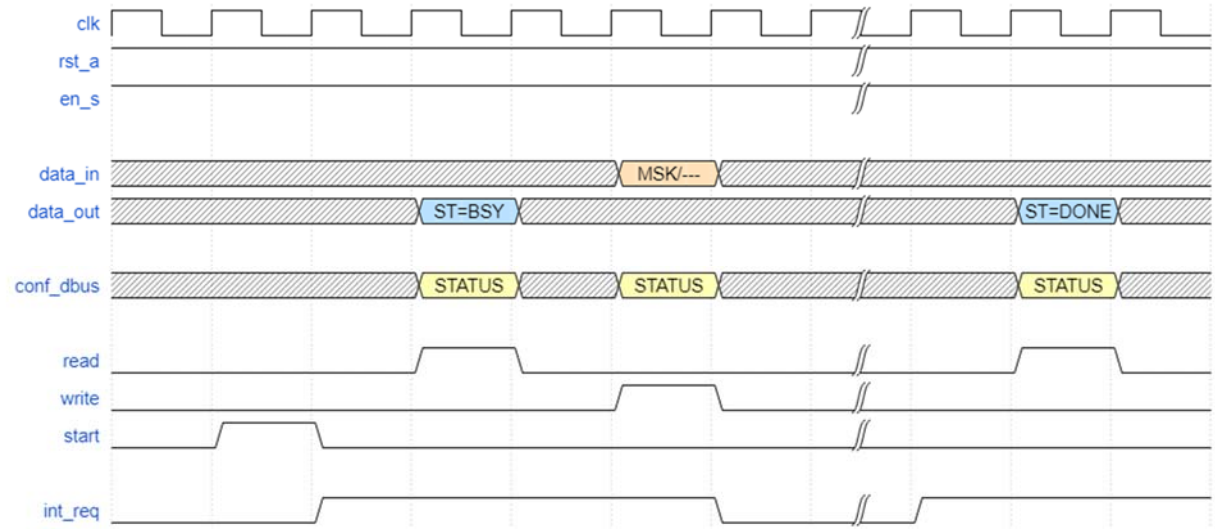

Figure 3.9 Simulation of the signals used for sends data to process

3.7.6. Reading data processed

Under the interruptions, are found the *data_rdy* flag interruption; when the *int_req* is detected and the *data_rdy* flag is active it means that the processed data can be read.

For reading data from the IP Dummy module, *conf_dbus* be set with the value 00010b to select the memory out as shown in Table 3.1. The information is in the *data_out* bus during the enable the reading process; the *read* signal is set to a high value for one cycle. Each rising edge indicates new reading data. Figure 3.10 shows the behavior of the *read* signal. Finally the interruption must be cleared. See Section 2.5 to clear the Status register.

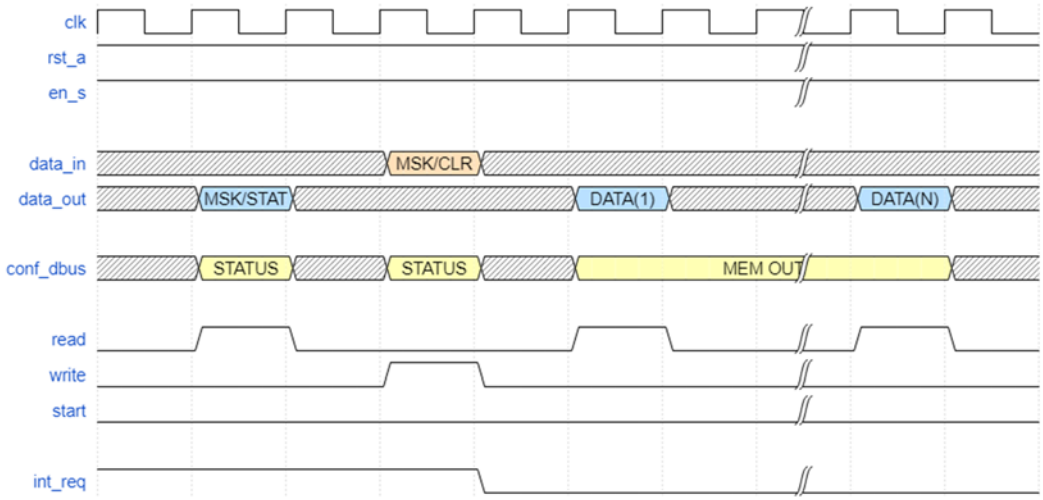

Figure 3.10 Simulation of the signals used for reading data to processed

3.7.7. Set the address reading or writing of the memories

To establish the memories address the *conf_dbus* will load the 00100b; to read and write all memories addressees the signal *data_in* should be used. Figure 3.11 shows the corresponding signals with some values and the behaviors that describe this module.

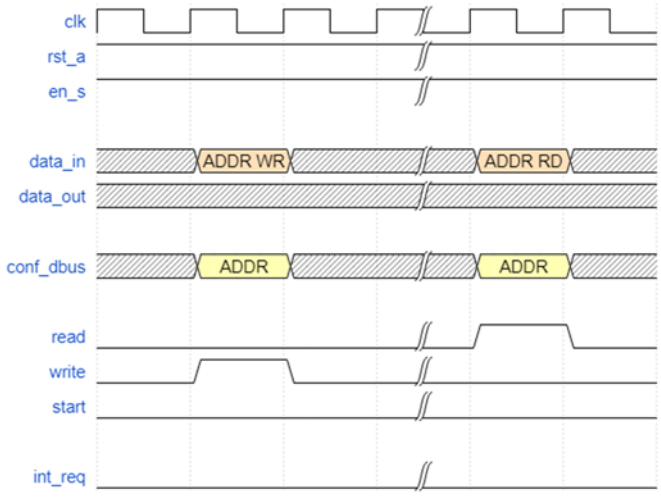

Figure 3.11 Simulation of the signals used for the addressing the memories

4. Appendix A “Code of the Modules”

This section shows previously described modules with a Core dummy example. Table 4.1 presents the files names and his hierarchy.

Hierarchy	File Name
1	IP_dummy_tb.v
1.1	IP_dummy.v
1.1.1	NoC_interface.v
1.1.1.1	simple_dual_port_ram_single_clock.v
1.1.1.2	configuration_register.v
1.1.1.3	ctrl_interface.v
1.1.1.4	ID.v
1.1.1.5	status.v
1.1.1.6	mux4g.v
1.1.2	IPcore.v
1.1.2.1	ctrlIP.v

Table 4.1 Files name and hierarchy

4.1. IP module testbench

```
`timescale 1ns/1ns
module IP_dummy_tb();

parameter      CYCLE          =  'd20;       // Define the clock work cycle
localparam     CONF           = 5'd0, //Value conf_dbus for ...
                       MEMI           = 5'd1,
                       MEMO           = 5'd2,
                       CONF_PTR       = 5'd3,
                       MEMI_PTR       = 5'd4,
                       MEMO_PTR       = 5'd5;

reg                            read;
reg                            write;
reg                            start;
reg            [4:0]   conf_dbus;
reg            [31:0]  data_in;

wire                           int_req;
wire    [31:0] data_out;

reg                            clk, rst_a, en_s;
```

```
always #(CYCLE/2) clk = !clk;

IP_dummy
IP_dummy
(
    .clk              (clk),
    .rst_a            (rst_a),
    .en_s             (en_s),
    .data_in   (data_in),        //different data in information types
    .data_out  (data_out),       //different data out information types
    .write            (write),         //Used for protocol to write different information types
    .read             (read),          //Used for protocol to read different information types
    .start            (start),         //Used to start the IP-core
    .conf_dbus (conf_dbus),      //Used for protocol to determine different actions types
    .int_req   (int_req)         //Interruption request
);

initial
begin
       //display($time, " << Start Simulation >>");
       clk                  = 1'b1;
       en_s          = 1'b1;
       read          = 1'b0;
       write         = 1'b0;
       start         = 1'b0;
       conf_dbus     = 5'd0;
       data_in       = 32'd0;
       rst_a         = 1'b0;// reset is active
       #3 rst_a      = 1'b1;// at time #n release reset
       #37
       task_dummy();
       $stop;
       //display($time, " << End Simulation >>");

end

// ----------- Task composed by basic interface tasks for dummy example ----------
task task_dummy;
begin
       read_ID();
       read_STATUS();
       write_STATUS(16'h0000,16'h0000);//Mask-Clear
       write_data('d0,CONF);
       write_data('d0,CONF);
       write_data_rand('d64,MEMI);
       start_procces;
       #(CYCLE*128)
       read_data('d64,MEMO);
end
endtask

// --------- Basic Tasks to use Interface -----------
task read_ID;  //Read the IP ID
begin
       conf_dbus     = 5'h1F;
       read          = 1'b1;
       #(CYCLE)
       read          = 1'b0;
       #(CYCLE);
end
endtask

task read_STATUS;     //Read the IP status
begin
       conf_dbus     = 5'h1E;
       read          = 1'b1;
       #(CYCLE)
       read          = 1'b0;
       #(CYCLE);
end
endtask
```

```
task write_STATUS;    //Write the mask and clear flags for status (2 input parameters)
input [15:0]   mask;
input [15:0]   clear;
begin
       conf_dbus      = 5'h1E;
       data_in        = {mask,clear};
       write          = 1'b1;
       #(CYCLE)
       write          = 1'b0;
       data_in        = 32'd0;
       #(CYCLE);
end
endtask

task start_procces;   //Send one cycle start signal
begin
       start   = 1'b1;
       #(CYCLE)
       start   = 1'b0;
       #(CYCLE);
end
endtask

task set_addr; //Write the memory in, memory out or config register  pointer, input parameters
pointer and conf_dbus value (2 input parameter)
input  [31:0] num_data;
input  [4:0]  conf_ptr;
begin
       conf_dbus      = conf_ptr;
       data_in= num_data;
       write  = 1'b1;
       #(CYCLE)
       write  = 1'b0;
       data_in= 32'd0;
       #(CYCLE);
end
endtask

task write_data;      //Write one data in the memory in or config register, input parameters data
and conf_dbus value (2 input parameter)
input [31:0]   data;
input  [4:0]   conf_ptr;
begin
       conf_dbus      = conf_ptr;
       data_in= data;
       write  = 1'b1;
       #(CYCLE)
       write  = 1'b0;
       data_in= 32'd0;
       #(CYCLE);
end
endtask

task write_data_rand; //Write n rand data in the memory in or config register pointer, input
parameters data and conf_dbus value (2 input parameter)
input  [31:0] num_data;
input  [4:0]  conf_ptr;
begin
       conf_dbus      = conf_ptr;
       repeat (num_data) begin
              data_in= $urandom;
              write  = 1'b1;
              #(CYCLE)
              write  = 1'b0;
              data_in= 32'd0;
              #(CYCLE);
       end
end
endtask
```

```
task read_data;       //Read one data from memory out or pointer value, input parameters data and
conf_dbus value (2 input parameter)
input   [31:0] num_data;
input   [4:0]  conf_ptr;
begin
       conf_dbus      = conf_ptr;
       #(CYCLE)
       repeat (num_data) begin
              read    = 1'b1;
              #(CYCLE)
              read    = 1'b0;
              #(CYCLE);
       end
end
endtask

endmodule
```

4.2. IP module

```
/*   Filename : IP_dummy.v*/
module IP_dummy
#(
    parameter   DATA_WIDTH      =   'd32,       //define data length
    parameter   ADDR_WIDTH_MEMI =   'd6,        //define Memory In depth
    parameter   ADDR_WIDTH_MEMO =   'd6,        //define Memory Out depth
    parameter   ADDR_WIDTH_CR   =   'd1,        //define Configuration Register depth
    parameter   STAT_WIDTH      =   'd16,        //define status length
    parameter   IP_ID           = 32'h00001001  //define IP-core ID value
)
(
    input   wire                        clk,
    input   wire                        rst_a,
    input   wire                        en_s,
    input   wire    [DATA_WIDTH-1:0]    data_in,        //different data in information types
    output  wire    [DATA_WIDTH-1:0]    data_out,       //different data out information types
    input   wire                        write,          //Used for protocol to write different
information types
    input   wire                        read,           //Used for protocol to read different
information types
    input   wire                        start,          //Used to start the IP-core
    input   wire    [4:0]               conf_dbus,      //Used for protocol to determine
different actions types
    output  wire                        int_req         //Interruption request
//-------------------------- Test points --------------------------//
//  output wire     [DATA_WIDTH-1:0]    dataOutMemB
);

    wire    [DATA_WIDTH-1:0]        data_MemIn0;        //data readed for memory in 0
    wire    [ADDR_WIDTH_MEMI-1:0]   rd_addr_MemIn0;     //address read for memory in 0

    wire    [DATA_WIDTH-1:0]        data_ConfigReg;     //data readed for configuration register
    wire    [ADDR_WIDTH_CR-1:0]     rd_addr_ConfigReg;  //address read for configuration register

    wire    [DATA_WIDTH-1:0]        data_MemOut0;       //data to write for memory out 0
    wire    [ADDR_WIDTH_MEMO-1:0]   wr_addr_MemOut0;    //address write for memory out 0
    wire                            wr_en_MemOut0;      //enable write for memory out 0

    wire                            start_IPcore;       //Used to start the IP-core

    wire    [STAT_WIDTH-1:0]        status_IPcore;      //data of IP-core to set the flags value

    NoC_interface
    #(
        .DATA_WIDTH     (DATA_WIDTH),       //define data length
        .ADDR_WIDTH_MEMI(ADDR_WIDTH_MEMI),  //define Memory In depth
        .ADDR_WIDTH_MEMO(ADDR_WIDTH_MEMO),  //define Memory Out depth
        .ADDR_WIDTH_CR  (ADDR_WIDTH_CR),    //define Configuration Register depth
        .STAT_WIDTH     (STAT_WIDTH),       //define status length
```

```
        .IP_ID          (IP_ID)             //define IP-core ID value
    )
    interface_dummy
    (
        .clk                (clk),
        .rst_a              (rst_a),
        .en_s               (en_s),
    //-------------------------- To/From NIc --------------------------//
        .conf_dbus          (conf_dbus),        //I: Used for protocol to determine different
actions types
        .read               (read),             //I: Used for protocol to read different
information types
        .write              (write),            //I: Used for protocol to write different
information types
        .start              (start),            //I: Used to start the IP-core
        .data_in            (data_in),          //I: different data in information types
        .int_req            (int_req),          //O: Interruption request
        .data_out           (data_out),         //O: different data out information types
    //------------------------ To/From IP-core -----------------------//
        .data_MemIn0        (data_MemIn0),      //O: data readed for memory in 0
        .rd_addr_MemIn0     (rd_addr_MemIn0),   //I: address read for memory in 0
        .data_ConfigReg     (data_ConfigReg),   //O: data readed for configuration register
        .rd_addr_ConfigReg  (rd_addr_ConfigReg),//I: address read for configuration register
        .data_MemOut0       (data_MemOut0),     //I: data to write for memory out 0
        .wr_addr_MemOut0    (wr_addr_MemOut0),  //I: address write for memory out 0
        .wr_en_MemOut0      (wr_en_MemOut0),    //I: enable write for memory out 0
        .start_IPcore       (start_IPcore),     //O: Used to start the IP-core
        .status_IPcore      (status_IPcore)     //I: data of IP-core to set the flags value
    //-------------------------- Test points --------------------------//
    //  .dataOutMemB(dataOutMemB)
    );

    IPcore
    #(
        .DATA_WIDTH     (DATA_WIDTH),       //define data length
        .ADDR_WIDTH_MEMI(ADDR_WIDTH_MEMI),  //define Memory In depth
        .ADDR_WIDTH_MEMO(ADDR_WIDTH_MEMO),  //define Memory Out depth
        .STAT_WIDTH     (STAT_WIDTH),       //define status length
        .ADDR_WIDTH_CR  (ADDR_WIDTH_CR)     //define Configuration Register depth
    )
    IPcore_dummy
    (
        .clk                (clk),
        .rst_a              (rst_a),
        .en_s               (en_s),
        .start_IPcore       (start_IPcore),     //I: Used to start the IP-core
        .data_MemIn0        (data_MemIn0),      //I: data readed for memory in 0
        .rd_addr_MemIn0     (rd_addr_MemIn0),   //O: address read for memory in 0
        .data_ConfigReg     (data_ConfigReg),   //I: data readed for configuration register
        .rd_addr_ConfigReg  (rd_addr_ConfigReg),//O: address read for configuration register
        .data_MemOut0       (data_MemOut0),     //O: data to write for memory out 0
        .wr_addr_MemOut0    (wr_addr_MemOut0),  //O: address write for memory out 0
        .wr_en_MemOut0      (wr_en_MemOut0),    //O: enable write for memory out 0
        .status_IPcore      (status_IPcore )    //O: data of IP-core to set the flags value
    //-------------------------- Test points --------------------------//
    );

endmodule
```

4.3. Interface

4.3.1. Top Level Interface

```
/*   Filename : Noc_interface.v*/
module NoC_interface
#(
    parameter   DATA_WIDTH     =    'd32,           //define data length
    parameter   ADDR_WIDTH_MEMI =   'd6,           //define Memory In depth
```

```
    parameter   ADDR_WIDTH_MEMO =   'd6,            //define Memory Out depth
    parameter   ADDR_WIDTH_CR  =    'd1,            //define Configuration Register depth
    parameter   STAT_WIDTH      =   'd16,            //define status length
    parameter   IP_ID          =    32'h00001001    //define IP-core ID value
)
(
    input   wire                            clk,
    input   wire                            rst_a,
    input   wire                            en_s,
//-------------------------- To/From NIc -------------------------//
    input   wire    [4:0]                   conf_dbus,          //Used for protocol to determine
different actions types
    input   wire                            read,               //Used for protocol to read
different information types
    input   wire                            write,              //Used for protocol to write
different information types
    input   wire                            start,              //Used to start the IP-core
    input   wire    [DATA_WIDTH-1:0]        data_in,            //different data in information
types
    output  wire                            int_req,            //Interruption request
    output  wire    [DATA_WIDTH-1:0]        data_out,           //different data out information
types
//------------------------ To/From IP-core ------------------------//
    output  wire    [DATA_WIDTH-1:0]        data_MemIn0,        //data readed for memory in 0
    input   wire    [ADDR_WIDTH_MEMI-1:0]   rd_addr_MemIn0,     //address read for memory in 0
    output  wire    [DATA_WIDTH-1:0]        data_ConfigReg,     //data readed for configuration
register
    input   wire    [ADDR_WIDTH_CR-1:0]     rd_addr_ConfigReg,  //address read for configuration
register
    input   wire    [DATA_WIDTH-1:0]        data_MemOut0,       //data to write for memory out 0
    input   wire                            wr_en_MemOut0,      //enable write for memory out 0
    input   wire    [ADDR_WIDTH_MEMO-1:0]   wr_addr_MemOut0,    //address write for memory out 0
    output  wire                            start_IPcore,       //Used to start the IP-core
    input   wire    [STAT_WIDTH-1:0]        status_IPcore       //data of IP-core to set the
flags value
);

    wire    [ADDR_WIDTH_MEMI-1:0]   wr_addr_MemIn0;     //address write for memory in 0
    wire                            wr_en_MemIn0;       //enable write for memory in 0

    wire    [ADDR_WIDTH_CR-1:0]     wr_addr_ConfigReg;  //address write for configuration
register
    wire                            wr_en_ConfigReg;    //enable write for configuration register

    wire    [DATA_WIDTH-1:0]        dataOut_MemOut0;    //data readed for memory out 0
    wire    [ADDR_WIDTH_MEMO-1:0]   rd_addr_MemOut0;    //address read for memory out 0

    wire    [3:0]                   sel_mux;            //address read for memory out 0
    wire                            en_clear;           //enable to clear the flags

    wire    [DATA_WIDTH-1:0]        data_IP_ID;         //ID IP-core value

    wire    [31:0]                  data_status;        //Status IP-core value

    assign  start_IPcore = start;                       //Start bypass
//-------------- Memories modules --------------//
    simple_dual_port_ram_single_clock
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH_MEMI)
    )
    MemIn0
    (
        .Write_clock__i(clk),
        .Write_enable_i(wr_en_MemIn0),      //I:enable write for memory in 0
        .Write_addres_i(wr_addr_MemIn0),    //I:address write for memory in 0
        .Read_address_i(rd_addr_MemIn0),    //I:address read for memory in 0
        .data_input___i(data_in),           //I:data to write for memory in 0
        .data_output__o(data_MemIn0)        //O:data readed for memory in 0
    );
```

```
    simple_dual_port_ram_single_clock
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH_MEMO)
    )
    MemOut0
    (
        .Write_clock__i(clk),
        .Write_enable_i(wr_en_MemOut0),     //I:enable write for memory out 0
        .Write_addres_i(wr_addr_MemOut0),   //I:address write for memory out 0
        .Read_address_i(rd_addr_MemOut0),   //I:address read for memory out 0
        .data_input___i(data_MemOut0),      //I:data to write for memory out 0
        .data_output__o(dataOut_MemOut0)    //O:data readed for memory out 0
    );
//------------ End memories modules ------------//

//-------------- Conf Reg modules --------------//
    configuration_register
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH_CR)
    )
    ConfigReg
    (
        .Write_clock__i(clk),
        .Write_enable_i(wr_en_ConfigReg),   //I:enable write for configuration register
        .Write_addres_i(wr_addr_ConfigReg), //I:address write for configuration register
        .Read_address_i(rd_addr_ConfigReg), //I:address read for configuration register
        .data_input___i(data_in),           //I:data to write for configuration register
        .data_output__o(data_ConfigReg)     //O:data readed for configuration register
    );
//------------ End Conf Reg modules ------------//

//  Control of the memories and the addresses
    ctrl_interface
    #(
        .ADDR_WIDTH_PTR  (ADDR_WIDTH_MEMI),//This value must be the biggest value of
ADDR_WIDTH_MEMI or ADDR_WIDTH_MEMO
        .ADDR_WIDTH_MEMI(ADDR_WIDTH_MEMI),  //define Memory In depth
        .ADDR_WIDTH_MEMO(ADDR_WIDTH_MEMO),  //define Memory Out depth
        .ADDR_WIDTH_CR   (ADDR_WIDTH_CR)
    )
    ctrl_interface
    (
        .clk               (clk),
        .rst_a             (rst_a),
        .en_s              (en_s),
        .read              (read),                          //I: Used for protocol to read
different information types
        .write             (write),                         //I: Used for protocol to write
different information types
        .conf_dbus         (conf_dbus),                     //I: Used for protocol to determine
different actions types
        .init_ptr          (data_in[ADDR_WIDTH_MEMI-1:0]), //I: Set the new pointer value for
memories or configuration register
        .sel_mux           (sel_mux),                       //O: Is a data out selecter
        .wr_addr_ConfigReg (wr_addr_ConfigReg),             //O: Address to write in
Configuration Registers
        .wr_addr_MemIn0    (wr_addr_MemIn0),                //O: Address to write in Memory In 0
        .rd_addr_MemOut0   (rd_addr_MemOut0),               //O: Address to read from Memory Out
0
        .en_clear          (en_clear),                      //O: enable to clear the flags
        .wr_en_ConfigReg   (wr_en_ConfigReg),               //O: Enable the write in
configuration_register
        .wr_en_MemIn0      (wr_en_MemIn0)                   //O: Enable the write in Memory In 0
    );

    mux4g
    #(
        .SIZE   (DATA_WIDTH )
    )
```

```
    mux
    (
        .a  (data_IP_ID),                                                   //I: ID IP-core value
        .b  (data_status),                                                  //I: Status IP-core value
        .c  (32'd0),
        .d  (dataOut_MemOut0),                                              //I: Memory Out 0 data
value
        .e  (32'd0),
        .f  ({{(DATA_WIDTH-ADDR_WIDTH_CR){1'b0}},wr_addr_ConfigReg}),       //I: write address
pointer for configuration register
        .g  ({{(DATA_WIDTH-ADDR_WIDTH_MEMI){1'b0}},wr_addr_MemIn0}),        //I: write address
pointer for Memory In 0
        .h  ({{(DATA_WIDTH-ADDR_WIDTH_MEMO){1'b0}},rd_addr_MemOut0}),       //I: read address pointer
for Memory Out 0
        .s  (sel_mux),                                                      //I: Is a data out
selecter
        .y  (data_out)                                                      //O: data out selected
    );

//  IP-core Identifier register to be sended to NIc
    ID
    #(
        .SIZE_REG   (DATA_WIDTH),
        .ID         (IP_ID)
    )
    IDreg
    (
        .clk        (clk),
        .data_IP_ID (data_IP_ID)    //O: ID value
    );

//  Status register to be sended to NIc
    status
    #(
        .SIZE_REG   (DATA_WIDTH),
        .STAT_WIDTH (STAT_WIDTH)
    )
    status_reg
    (
        .clk            (clk),
        .rst_a          (rst_a),
        .en_clear       (en_clear),                     //I: enable to clear the flags
        .clear          (data_in[STAT_WIDTH-1:0]),      //I: set in 1 all flags to clear
        .mask           (data_in[STAT_WIDTH+15:16]),    //I: set in 1 all flags can send a
interruption
        .status_IPcore  (status_IPcore),                //I: data of IP-core to set the flags
value
        .data_status    (data_status),                  //O: data IP-core status value
        .int_req        (int_req)                       //O: interruption request
    );

endmodule
```

4.3.1.1. Asynchronous memory

If is necessary to use asynchronous memories the next code is used in the space for memories modules.

```
    //-------------- Asynchronous --------------//
    simple_dual_port_ram
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH_MEMI)
    )
    MemIn0
    (
        .Write_enable_i(wr_en_MemIn0),      //I:enable write for memory in 0
        .Write_addres_i(wr_addr_MemIn0),    //I:address write for memory in 0
        .Read_address_i(rd_addr_MemIn0),    //I:address read for memory in 0
        .data_input___i(data_in),           //I:data to write for memory in 0
        .data_output__o(data_MemIn0)        //O:data readed for memory in 0
```

```
    );

    simple_dual_port_ram
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH_MEMO)
    )
    MemOut0
    (
        .Write_enable_i(wr_en_MemOut0),     //I:enable write for memory out 0
        .Write_addres_i(wr_addr_MemOut0),   //I:address write for memory out 0
        .Read_address_i(rd_addr_MemOut0),   //I:address read for memory out 0
        .data_input___i(data_MemOut0),      //I:data to write for memory out 0
        .data_output__o(dataOut_MemOut0)    //O:data readed for memory out 0
    );
```

4.3.1.2. *Synchronous memory*

If is necessary to use synchronous memories the next code is used in the space for memories modules.

```
    //-------------- Synchronous --------------//
    simple_dual_port_ram_single_clock
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH_MEMI)
    )
    MemIn0
    (
        .Write_clock__i(clk),
        .Write_enable_i(wr_en_MemIn0),      //I:enable write for memory in 0
        .Write_addres_i(wr_addr_MemIn0),    //I:address write for memory in 0
        .Read_address_i(rd_addr_MemIn0),    //I:address read for memory in 0
        .data_input___i(data_in),           //I:data to write for memory in 0
        .data_output__o(data_MemIn0)        //O:data readed for memory in 0
    );

    simple_dual_port_ram_single_clock
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .ADDR_WIDTH(ADDR_WIDTH_MEMO)
    )
    MemOut0
    (
        .Write_clock__i(clk),
        .Write_enable_i(wr_en_MemOut0),     //I:enable write for memory out 0
        .Write_addres_i(wr_addr_MemOut0),   //I:address write for memory out 0
        .Read_address_i(rd_addr_MemOut0),   //I:address read for memory out 0
        .data_input___i(data_MemOut0),      //I:data to write for memory out 0
        .data_output__o(dataOut_MemOut0)    //O:data readed for memory out 0
    );
```

4.3.2. Control interface

```
/*   Filename : ctrl_interface.v*/
module ctrl_interface
#(
    parameter ADDR_WIDTH_PTR  = 'd6,    //This value must be the bigest value of ADDR_WIDTH_MEMI
or ADDR_WIDTH_MEMO
    parameter ADDR_WIDTH_MEMI =   'd6,
    parameter ADDR_WIDTH_MEMO =   'd6,
    parameter ADDR_WIDTH_CR   = 'd1
)
(
    input   wire                            clk,
    input   wire                            rst_a,
    input   wire                            en_s,
    input   wire                            read,               //Used for protocol to read
different information types
```

```
    input   wire                            write,              //Used for protocol to write
different information types
    input   wire    [4:0]                   conf_dbus,          //Used for protocol to determine
different actions types
    input   wire    [ADDR_WIDTH_PTR-1:0]    init_ptr,           //Set the new pointer value for
memories or configuration register
    output  wire    [3:0]                   sel_mux,            //Data out selecter
    output  reg     [ADDR_WIDTH_CR-1:0]     wr_addr_ConfigReg,  //Address to write in
Configuration Registers
    output  reg     [ADDR_WIDTH_MEMI-1:0]   wr_addr_MemIn0,     //Address to write in Memory In 0
    output  reg     [ADDR_WIDTH_MEMO-1:0]   rd_addr_MemOut0,    //Address to read from Memory Out
0
    output  wire                            en_clear,           //enable to clear the flags
    output  wire                            wr_en_ConfigReg,    //Enable the write in
configuration_register
    output  wire                            wr_en_MemIn0        //Enable the write in Memory In 0
);

    reg wr_en;
    reg wr_en_cr;

// Config definition
    localparam  CONF_REG    = 5'b00000,
                MEM_IN      = 5'b00001,
                MEM_OUT     = 5'b00010,
                CR_WR_PTR   = 5'b00011,
                MI_WR_PTR   = 5'b00100,
                MO_RD_PTR   = 5'b00101,
                STAT_REG    = 5'b11110,
                ID_REG      = 5'b11111;

// Mux out definition.
    localparam  MUX_ID          = 4'b0000,
                MUX_STAT        = 4'b0001,
                MUX_PARM        = 4'b0010,
                MUX_DATAo       = 4'b0011,
                MUX_DATAi       = 4'b0100,
                MUX_PTR_WRcr    = 4'b0101,
                MUX_PTR_WRi     = 4'b0110,
                MUX_PTR_RDo     = 4'b0111,
                MUX_DEFT        = 4'b1111;

// Write and read controlled by config. Add more signals for every memory in
    assign  en_clear        =   write & (conf_dbus == STAT_REG);
    assign  wr_en_MemIn0    =   write & (conf_dbus == MEM_IN);  // this signal is for memory in
    assign  wr_en_ConfigReg =   write & (conf_dbus == CONF_REG);

// Data out controlled by config. To increase number of data outs, you should also change the
mux4g.v file.
    assign  sel_mux  =  (conf_dbus == CONF_REG)    ?   MUX_PARM    :
                        (conf_dbus == MEM_OUT)     ?   MUX_DATAo   :
                        (conf_dbus == MEM_IN)      ?   MUX_DATAi   :
                        (conf_dbus == STAT_REG)    ?   MUX_STAT    :
                        (conf_dbus == CR_WR_PTR)   ?   MUX_PTR_WRcr:
                        (conf_dbus == MI_WR_PTR)   ?   MUX_PTR_WRi :
                        (conf_dbus == MO_RD_PTR)   ?   MUX_PTR_RDo :
                        (conf_dbus == ID_REG)      ?   MUX_ID      :   MUX_DEFT;

// Control of address memory. Add more signals for every memory in, memory out or configuration
register
    always @(posedge clk) begin
        wr_en      <= (write && conf_dbus == MEM_IN);
        wr_en_cr   <= (write && conf_dbus == CONF_REG);
    end

    always @(posedge clk or negedge rst_a) begin
        if (!rst_a) begin
            wr_addr_ConfigReg   <= 'd0;
            wr_addr_MemIn0      <= 'd0;
            rd_addr_MemOut0     <= 'd0;
        end
```

```
        else begin
            if (en_s) begin
                if (write && conf_dbus == CR_WR_PTR)
                    wr_addr_ConfigReg   <= init_ptr[ADDR_WIDTH_CR-1:0];
                else if (wr_en_cr)
                    wr_addr_ConfigReg   <= wr_addr_ConfigReg + 1'b1;
                else if (write && conf_dbus == MI_WR_PTR)
                    wr_addr_MemIn0      <= init_ptr[ADDR_WIDTH_MEMI-1:0];
                else if (wr_en)
                    wr_addr_MemIn0      <= wr_addr_MemIn0 + 1'b1;
                else if (write && conf_dbus == MO_RD_PTR)
                    rd_addr_MemOut0     <= init_ptr[ADDR_WIDTH_MEMO-1:0];
                else if (read && conf_dbus == MEM_OUT)
                    rd_addr_MemOut0     <= rd_addr_MemOut0 + 1'b1;
            end
        end
    end

endmodule
```

4.3.3. Identifier register

```
/*  Filename : ID.v*/
module ID
#(
   parameter SIZE_REG   =   'd32,
   parameter ID         = 32'h00001000
)
(
    input wire                  clk,
    output reg [SIZE_REG-1:0]   data_IP_ID  //ID value
);

always @(posedge clk)
    data_IP_ID <= ID;

endmodule
```

4.3.4. Status register

```
/*  Filename : status.v*/
module status
#(
    parameter SIZE_REG      = 'd32,
    parameter STAT_WIDTH    = 'd16
)
(
    input   wire                        clk,
    input   wire                        rst_a,
    input   wire                        en_clear,       //enable to clear the flags
    input   wire    [STAT_WIDTH-1:0]    clear,          //set in 1 all flags to clear
    input   wire    [STAT_WIDTH-1:0]    mask,           //set in 1 all flags can send a
interruption
    input   wire    [STAT_WIDTH-1:0]    status_IPcore,  //data of IP-core to set the flags value
    output  wire    [SIZE_REG-1:0]      data_status,    //data IP-core status value
    output  wire                        int_req         //interruption request
);

    reg [STAT_WIDTH-1:0]  regStatus;
    reg [(STAT_WIDTH/2)-1:0]  maskStatus;

    assign data_status  = {maskStatus,regStatus};
    assign int_req      = ~|(regStatus[7:0] & maskStatus[7:0]);

    genvar i;

    generate
        for ( i = 0; i < STAT_WIDTH; i = i+1 ) begin : buff
```

```
            if (i >= 8) begin       // Status flags
                always @ (posedge clk or negedge rst_a) begin
                    if(!rst_a)
                        regStatus[i] <= 1'b0;
                    else begin
                        regStatus[i] <= status_IPcore[i];
                    end
                end
            end
            else begin              // Interruption flags
                always @ (posedge clk or negedge rst_a) begin
                    if(!rst_a)
                        regStatus[i] <= 1'b0;
                    else begin
                        if (clear[i] & en_clear)
                            regStatus[i] <= 1'b0;
                        else if (status_IPcore[i])
                            regStatus[i] <= 1'b1;
                        else
                            regStatus[i] <= regStatus[i];
                    end
                end
            end
        end
    endgenerate

    always @(posedge clk or negedge rst_a) begin
        if(!rst_a)
            maskStatus      <=  {(STAT_WIDTH/2){1'b0}};
        else if(en_clear)
            maskStatus      <=  mask[7:0];
    end

endmodule
```

4.3.5. Data out mux

```
/*  Filename : mux4g.v*/
module mux4g
#(
    parameter SIZE = 7
)
(
    input   wire    [SIZE-1:0]   a,
    input   wire    [SIZE-1:0]   b,
    input   wire    [SIZE-1:0]   c,
    input   wire    [SIZE-1:0]   d,
    input   wire    [SIZE-1:0]   e,
    input   wire    [SIZE-1:0]   f,
    input   wire    [SIZE-1:0]   g,
    input   wire    [SIZE-1:0]   h,
    input   wire    [3: 0]       s,
    output  reg     [SIZE-1:0]   y
);

    always @(*) begin
        case (s)
            3'd0: y = a;
            3'd1: y = b;
            3'd2: y = c;
            3'd3: y = d;
            3'd4: y = e;
            3'd5: y = f;
            3'd6: y = g;
            3'd7: y = h;
            default: y = 'd0;
        endcase
    end
```

```
endmodule
```

4.3.6. Configuration Registers

```
// Module Name:    configuration_register
module configuration_register
#(
    parameter DATA_WIDTH    =   12,     // Datawidth of data
    parameter ADDR_WIDTH    =   6       // Address bits
)
(
    input                       Write_clock__i,
    input                       Write_enable_i,
    input   [(ADDR_WIDTH-1):0]  Write_addres_i,
    input   [(ADDR_WIDTH-1):0]  Read_address_i,
    input   [(DATA_WIDTH-1):0]  data_input___i,
    output  [(DATA_WIDTH-1):0]  data_output__o
);

    reg [(DATA_WIDTH-1):0] reg_Structure [2**ADDR_WIDTH-1:0];

    assign  data_output__o = reg_Structure[Read_address_i];

    always @(posedge Write_clock__i) begin
        if (Write_enable_i) begin
            reg_Structure[Write_addres_i] = data_input___i;
        end
    end

endmodule
```

4.4. Core Dummy

4.4.1. Top Level core dummy

```
/*  Filename : IPcore.v*/
module IPcore
#(
    parameter   DATA_WIDTH      =   'd32,
    parameter   ADDR_WIDTH_MEMI =   'd6,
    parameter   ADDR_WIDTH_MEMO =   'd6,
    parameter   STAT_WIDTH      =   'd16,
    parameter   ADDR_WIDTH_CR   =   'd1
)
(
    input   wire                            clk,
    input   wire                            rst_a,
    input   wire                            en_s,
    input   wire                            start_IPcore,
    input   wire    [DATA_WIDTH-1:0]        data_MemIn0,
    output  wire    [ADDR_WIDTH_MEMI-1:0]   rd_addr_MemIn0,
    input   wire    [DATA_WIDTH-1:0]        data_ConfigReg,
    output  wire    [ADDR_WIDTH_CR-1:0]     rd_addr_ConfigReg,
    output  wire    [DATA_WIDTH-1:0]        data_MemOut0,
    output  wire    [ADDR_WIDTH_MEMO-1:0]   wr_addr_MemOut0,
    output  wire                            wr_en_MemOut0,
    output  wire    [STAT_WIDTH-1:0]        status_IPcore
);


    wire    done_f;
    wire    data_rdy;
    wire    data_read;
    wire    bsy_f;
```

```
    assign  status_IPcore    = {{((STAT_WIDTH/2)-1){1'b0}},bsy_f,{((STAT_WIDTH/2)-
3){1'b0}},data_read,data_rdy,done_f};
    assign  data_MemOut0     = data_MemIn0;

    ctrlIP
    #(
        .DATA_WIDTH     (DATA_WIDTH     ),
        .ADDR_WIDTH_MEMI(ADDR_WIDTH_MEMI),
        .ADDR_WIDTH_MEMO(ADDR_WIDTH_MEMO),
        .ADDR_WIDTH_CR  (ADDR_WIDTH_CR  )
    )
    ctrlIP
    (
        .clk        (clk),
        .rst_a      (rst_a),
        .en_s       (en_s),
        .start      (start_IPcore),
        .confReg    (data_ConfigReg),
        .addrCR     (rd_addr_ConfigReg),
        .addrWR     (wr_addr_MemOut0),
        .addrRD     (rd_addr_MemIn0),
        .enWR       (wr_en_MemOut0),
        .busy_f     (bsy_f),
        .done_f     (done_f),
        .data_rdy   (data_rdy),
        .data_read  (data_read)
    );

endmodule
```

4.4.2. Control IP dummy

```
/*  Filename : ctrlIP.v*/
module ctrlIP
#(
    parameter   DATA_WIDTH      =   'd32,
    parameter   ADDR_WIDTH_MEMI =   'd6,
    parameter   ADDR_WIDTH_MEMO =   'd6,
    parameter   ADDR_WIDTH_CR   =   'd1
)
(   input   wire                            clk,
    input   wire                            rst_a,
    input   wire                            en_s,
    input   wire                            start,
    input   wire    [DATA_WIDTH-1:0]        confReg,
    output  reg     [ADDR_WIDTH_CR-1:0]     addrCR,
    output  reg     [ADDR_WIDTH_MEMO-1 : 0] addrWR,
    output  reg     [ADDR_WIDTH_MEMI-1 : 0] addrRD,
    output  reg                             enWR,
    output  reg                             busy_f,
    output  reg                             done_f,
    output  reg                             data_rdy,
    output  reg                             data_read
);

    localparam  STANDBY     =   'd0,
                CONFIG      =   'd1,
                DELAY       =   'd2,
                IP_PROC     =   'd3,
                WAIT_B      =   'd4,
                WAIT_A      =   'd5,
                EN_DELAY    =   32'h0A0A0A0A,
                WAIT        =   4'hF;

    reg [2:0]   state;
    reg [15:0]  count_b;
    reg [31:0]  count;
    reg         en_count;

    always @(posedge clk or negedge rst_a) begin
```

```
    if(!rst_a) begin
        count    <= 'd0;
        count_b  <= 'd0;
    end
    else begin
        if(en_count) begin
            if (count_b == 16'hC350) begin
                count    <= count + 1'b1;
                count_b  <= 'd0;
            end
            else begin
                count_b  <= count_b + 1'b1;
            end
        end
        else begin
            count    <= 'd0;
            count_b  <= 'd0;
        end
    end
end

always @(posedge clk or negedge rst_a) begin
    if (!rst_a) begin
        state       <= STANDBY;
        addrCR      <=  'd0;
        addrRD      <=  'd0;
        addrWR      <=  'd0;
        enWR        <= 1'b0;
        busy_f      <= 1'b0;
        done_f      <= 1'b0;
        data_rdy    <= 1'b0;
        data_read   <= 1'b0;
        en_count    <=  'd0;
    end
    else begin
        if (en_s) begin
            case (state)
                STANDBY:
                    begin
                        if (start) begin
                            state       <= CONFIG;
                            busy_f      <= 1'b1;
                        end
                        done_f      <= 1'b0;
                        data_rdy    <= 1'b0;
                        data_read   <= 1'b0;
                    end
                CONFIG:
                    if (confReg == EN_DELAY) begin
                        state       <= DELAY;
                        addrCR      <= 1'b1;
                        en_count    <= 1'b1;
                    end
                    else begin
                        state       <= WAIT_B;
                    end
                DELAY:
                    if (count == confReg) begin //delay in milliseconds
                        state       <= WAIT_B;
                        addrCR      <= 'd0;
                        en_count    <= 1'b0;
                    end
                    else begin
                        state       <= DELAY;
                    end
                WAIT_B:
                    begin
                        state       <= IP_PROC;
                        enWR        <= 1'b1;
                    end
                IP_PROC:
```

```
                    begin
                        state       <= WAIT_A;
                        enWR        <= 1'b0;
                    end
                WAIT_A:
                    if (addrWR == {ADDR_WIDTH_MEMO{1'b1}}) begin
                        state       <= STANDBY;
                        addrRD      <=  'd0;
                        addrWR      <=  'd0;
                        enWR        <= 1'b0;
                        busy_f      <= 1'b0;
                        done_f      <= 1'b1;
                        data_read   <= 1'b1;
                    end
                    else begin
                        if (addrWR == 'd8)
                            data_rdy    <= 1'b1;
                        state       <= WAIT_B;
                        addrRD      <= addrRD + 1'b1;
                        addrWR      <= addrWR + 1'b1;
                        enWR        <= 1'b0;
                    end
                default:
                    begin
                        state       <= STANDBY;
                        addrCR      <=  'd0;
                        addrRD      <=  'd0;
                        addrWR      <=  'd0;
                        enWR        <= 1'b0;
                        busy_f      <= 1'b0;
                        done_f      <= 1'b0;
                        data_rdy    <= 1'b0;
                        data_read   <= 1'b0;
                        en_count    <=  'd0;
                    end
            endcase
        end
    end
end

endmodule
```

4.5. Memories

4.5.1. Synchronous memory

```
// simple_dual_port_ram_single_clock
module simple_dual_port_ram_single_clock
#(
    parameter DATA_WIDTH    =   12,     // Datawidth of data
    parameter ADDR_WIDTH    =   6       // Address bits
)
(
    input                           Write_clock__i,
    input                           Write_enable_i,
    input       [(ADDR_WIDTH-1):0]  Write_addres_i,
    input       [(ADDR_WIDTH-1):0]  Read_address_i,
    input       [(DATA_WIDTH-1):0]  data_input___i,
    output  reg [(DATA_WIDTH-1):0]  data_output__o
);

    reg [(DATA_WIDTH-1):0] RAM_Structure [2**ADDR_WIDTH-1:0];

    always @(posedge Write_clock__i) begin
        if (Write_enable_i) begin
            RAM_Structure[Write_addres_i] = data_input___i;
        end

        data_output__o <= RAM_Structure[Read_address_i];
    end
```

```
endmodule
```

4.5.2. Asynchronous memory

```
// simple_dual_port_ram
module simple_dual_port_ram
#(
    parameter DATA_WIDTH    =   12,     // Datawidth of data
    parameter ADDR_WIDTH    =   6       // Address bits
)
(
    input                       Write_clock__i,
    input                       Write_enable_i,
    input   [(ADDR_WIDTH-1):0]  Write_addres_i,
    input   [(ADDR_WIDTH-1):0]  Read_address_i,
    input   [(DATA_WIDTH-1):0]  data_input___i,
    output  [(DATA_WIDTH-1):0]  data_output__o
);

    reg [(DATA_WIDTH-1):0] reg_Structure [2**ADDR_WIDTH-1:0];

    assign  data_output__o = reg_Structure[Read_address_i];

    always @(posedge Write_clock__i) begin
        if (Write_enable_i) begin
            reg_Structure[Write_addres_i] = data_input___i;
        end
    end

endmodule
```

5. Appendix B “Important implementation notes”

In order to avoid conflicts during the development and implementation of the IP modules, the Verilog files that have to be modified as the functional requirements given by the HW-designer, must be renamed according to the IP-core.

As was said before, the IP designers must adapt the Interface design with their IP-cores following the connecting the IP-core to the Interface connections described in Figure 2.4. Hence, the hardware description files for the IP Dummy model were provide as a guidance for an easy adaptation with others IP-cores. However, if the IP designer modifies any of the Verilog files listed in Table 4.1, then he has to create other version of those files. The Verilog files listed in Table 4.1 corresponds exactly to the functional requirements in accordance to the IP-Dummy.

As an example, if the IP-designer decides to take out the output memory **MemOut0** from the interface, then besides to the corresponding hardware modifications, the file name `NoC_interface.v` may change the name to `NoC_interface_NewIPCore.v` as a suggestion.

1.1.1	NoC_interface_NewIPCore.v

```
module NoC_interface_NewIPCore
#(
```

```
    parameter   DATA_WIDTH     =    'd32,           //define data length
    parameter   ADDR_WIDTH_MEMI =   'd6,            //define Memory In depth
    parameter   ADDR_WIDTH_MEMO =   'd6,            //define Memory Out depth
    parameter   ADDR_WIDTH_CR  =    'd1,            //define Configuration Register depth
    parameter   STAT_WIDTH      =   'd16,            //define status length
    parameter   IP_ID          =    32'h00001001    //define IP-core ID value
)
(
    input   wire                            clk,
    input   wire                            rst_a,

       ...

       ..

       .
```

The consideration of change the files names and entities, has to be applied to all the Verilog files.

6. Appendix c “Simulation of the IP module”

The file IP_dummy_tb.v contains basic tasks that are used to make use of the interface, the conf_dbus value can use the values defined in localparam, those values are related to the definition found in CVS file.

The *read_ID* task is used to read the IP ID, lasts a two clock cycles, it does not use input parameters.

The *read_STATUS* task is used to read the status of the IP, lasts a two clock cycles, it does not use input parameters.

The *write_STATUS* task is used to write the status mask and clean the IP interrupts, it lasts two clock cycles, it does use two input parameters, *mask* and *clear*.

The *start_procces* task is used to send a start signal to the IP, it lasts two clock cycles, it does not use input parameters.

The *set_addr* task is used to initialize the address pointer for the input memory, the output memory or the configuration register, lasts two clock cycles, uses the input parameters *data* (pointer value) and *conf_ptr* (pointer to select MEMI_PTR, MEMO_PTR or CONF_PTR).

The *write_data* task is used to write a data in the input memory or the configuration register, it lasts two clock cycles, it uses the input parameters *data* (information to write) and *conf_ptr* (value to select MEMI or CONF).

The *write_data_rand* task is used to write *n* data in the input memory or the configuration register randomly, lasts *n* * two clock cycles, uses the input parameters *num_data* (number of data to be written) and *conf_ptr* (value to select MEMI or CONF).

The *read_data* task is used to read *n* data in the output memory, lasts *n* * two clock cycles, uses the input parameters *num_data* (number of data to write) and *conf_ptr* (value to select MEMO).

This section shows the simulation for an IP, this is exemplified with an IP-core dummy.

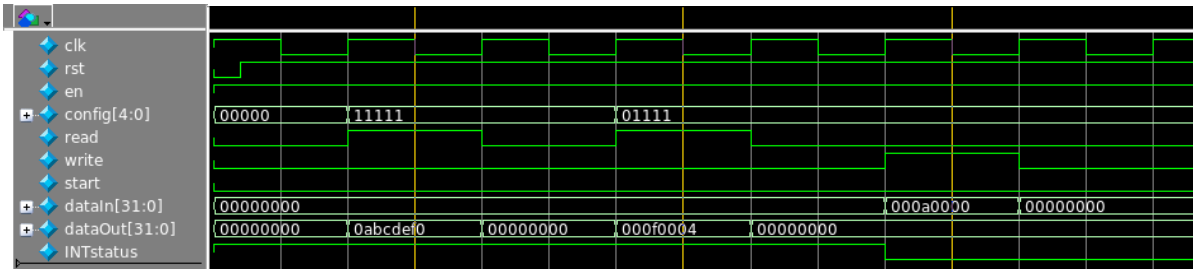

Figure 6.1 Simulation for read the ID and status, write the mask for the status register

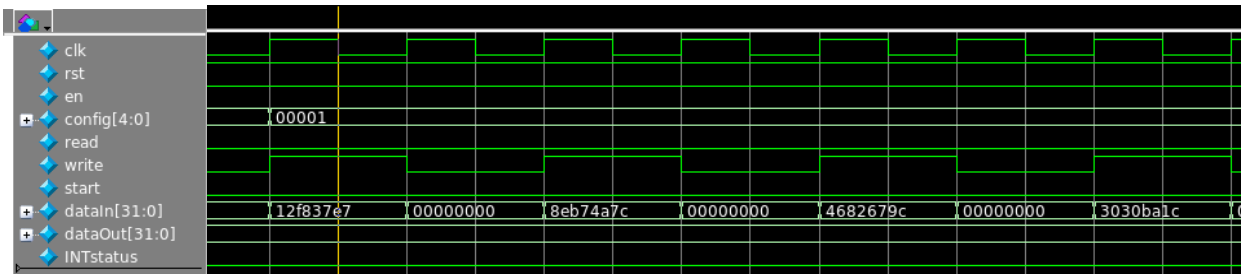

Figure 6.2 Simulation for writes data

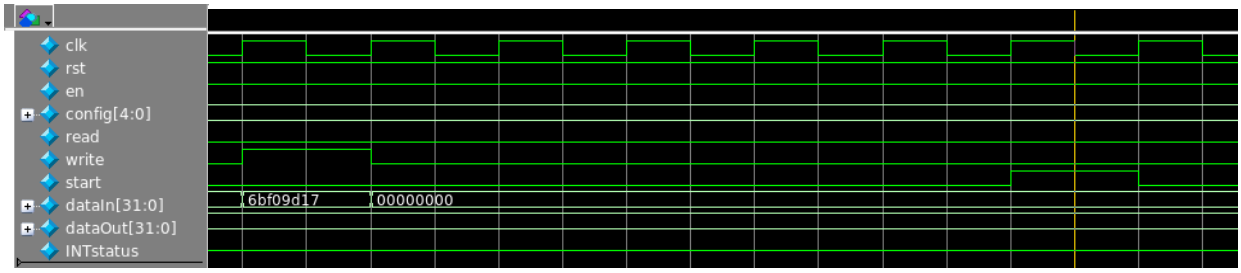

Figure 6.3 Simulation for starts the process of the IP

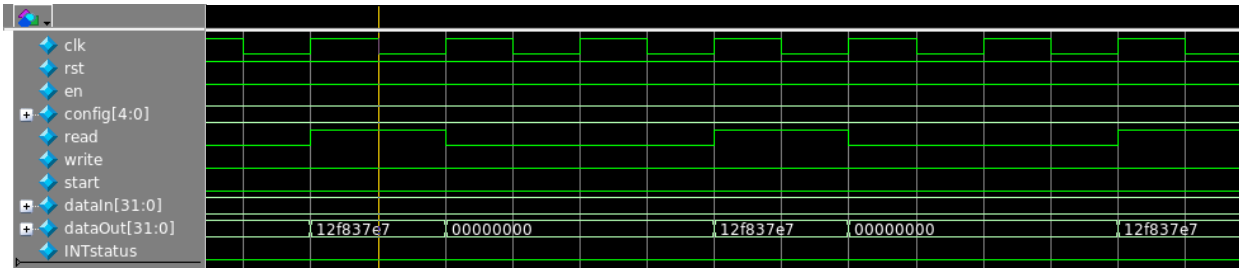

Figure 6.4 Simulation for read data

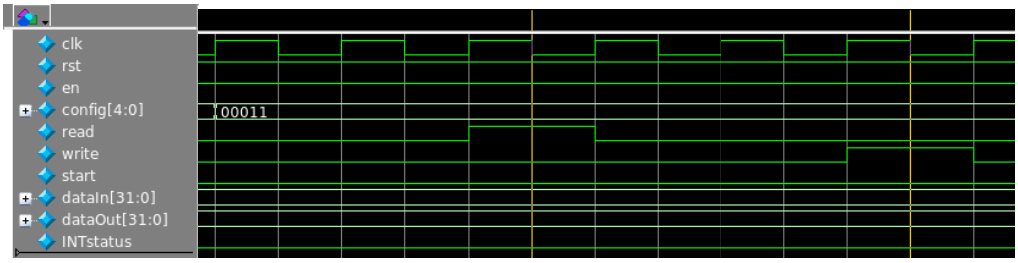

Figure 6.5 Simulation for set the address of memories