



Politecnico di Torino

System design project

Project8

IP-core Manager for FPGA-based Designs (RT level)

Final Report

Master degree in Computer Engineering
Specialization in Embedded Systems

Referents:

Prof. Paolo Ernesto Prinetto
PhD. Student Giuseppe Airò Farulla

Authors:

Emanuele Garolla
Gina Jiang
Evelina Forno
Francesco Buttafuoco
Salvatore Bellino

June, 2017

Acknowledgement

We would like to express our special thanks of gratitude to our class mates involved in a similar project (project 14, 7 and especially 13) who collaborated with us on defining the CPU On Board IP-core Manager protocol.

Contents

1	Introduction	1
1.1	Multi IP-core systems and the IP-core Manager	1
2	System's architecture	2
2.1	Components	2
2.2	Interfaces	2
2.2.1	Data Buffer interface	3
2.2.2	IP core interface	3
2.3	CPU Control bits	5
3	IP-core Manager functionalities	6
3.1	Enabling the right IP	6
3.2	Connecting the signals	8
3.3	Interrupt Handler	10
4	User Manual	12
4.1	Introduction	12
4.2	Interface to the CPU	12
4.2.1	Initiating a transaction	12
4.2.2	Writing data to the IP core	13
4.2.3	Reading data from the IP core	14
4.2.4	Ending a transaction	14
4.2.5	Servicing interrupts	14
4.3	Interface to the IP cores	14
4.3.1	Reading data from the cpu	15
4.3.2	Writing data to the cpu	15
4.3.3	Requesting an interrupt	15
4.4	Use case scenario	16
4.4.1	example 1: adder IP	16
4.4.2	example 2: accumulator IP	17
5	Guidelines	19

6	Future developments	20
6.1	Asynchronous is the new synchronous	20
6.2	Configurable manager	20
6.3	Interface for the IP cores	20

CHAPTER 1

Introduction

1.1 Multi IP-core systems and the IP-core Manager

A multi IP-core system is a component with two or more independent unit each one designed for a different purpose. These cores are deployed into the FPGA and are used for enhancing performance, implementing functions, and simultaneous processing of multiple tasks. However due to the increasing complexity, a IP core manager is required to handle context switching, scheduling, and interrupt handling.

The first and basic task for this core manager is to enable the connection between the CPU and the selected core. More in particular this means exchanging the data in the required time as described in the protocol. The CPU can talk to one and only one core, therefore the core manager will disregard other cores whether they finished or not their task.

The exchanging of data is done by a means of a dual-port buffer (64x16). The buffer is the component that communicate the physical address of the interested core from the CPU to the IP-core manager (and viceversa) and exchange the data between the CPU and the IP-cores.

Even for the data buffer we have some constraints: only one core at a time can talk to the buffer. This requirement is satisfied thanks to the IP manager.

As for the priority of the cores, we put the most priority core at $port_0$ and the least one to the last port, i.e. $port_{n-1}$.

CHAPTER 2

System's architecture

2.1 Components

For this project, 3 main components are required (fig. 2.1):

1. **IP-core Manager**: it's the main element of this architecture and its main goal is to handle the complexity of this system. It has to provide the correct exchange of data between the target IP-core and the CPU. Moreover it has to handle the interrupt service routine.
2. **Data Buffer**: it's a dual port buffer that stores the data coming both from the CPU and the selected IP-core. It also enable the communication of the desired transaction from the CPU to the IP-core Manager by means of the *row0* which is always read from the IP-core Manager
3. **IP-core**: a core that perform a specific task or function.

2.2 Interfaces

A multi IP-core system requires a IP core manager to handle the complexity of this system. Main tasks of the IP Manager are the correct exchange of data between the target IP and the CPU. Another critical task is the interrupt handling. Figure 2.1 shows the overall architecture of the whole system adopted.

As shown in the figure 2.1, the IP manager indirectly communicates with the CPU through a dual port data buffer. This buffer has a standard interface and it is used to redirect data to/from the IP core selected. Whenever the CPU wants to start a transaction, it has to write some control bits (explained in section 2.3) at address 0 (*row0*) of the data buffer. The data in this address is always read from the IP Manager

to speed-up the routing process whenever a new transaction begins.

2.2.1 Data Buffer interface

This data buffer has a standard interface. The signals for this components (whether they come from the CPU or from from the IP core Manager or from both of them) has the following function:

- **data**: input/output port, used for transferring data from/to the CPU to/from the buffer.
- **add**: input port, used for selecting the right row within the buffer to write/read data.
- **w_enable**: input port, must be asserted in order to enable write operation.
- **r_enable**: input port, must be asserted in order to enable read operation.
- **generic_enable**: input port, must be asserted in order to start any operation on that port.
- **reset**: input port, it brings the buffer to the reset state.
- **row_0**: output port, it reflects any change in the row 0 of the buffer.

2.2.2 IP core interface

As for a generic x IP core interface, we have the following signals:

- **data_in_IPs(x)**: data from the x IP-core to the CPU/Data buffer.
- **data_out_IPs(x)**: data to the x IP-core from the CPU/Data buffer.
- **add_IPs(x)**: address from the x IP-core to the Data buffer.
- **W_enable_IPs(x)**: when the x IP-core wants to write to the buffer
- **R_enable_IPs(x)**: when the x IP-core wants to read from the buffer
- **Generic_en_IPs(x)**: when the x IP-core wants to communicate with the buffer
- **enable_IPs(x)**: when the CPU wants to communicate with the x IP-core
- **ack_IPs(x)**: the IP-core manager sent this signal to the x IP-core to tell it that its interrupt request will be served
- **interrupt_IPs(x)**: when the x IP-core raises an interrupt request

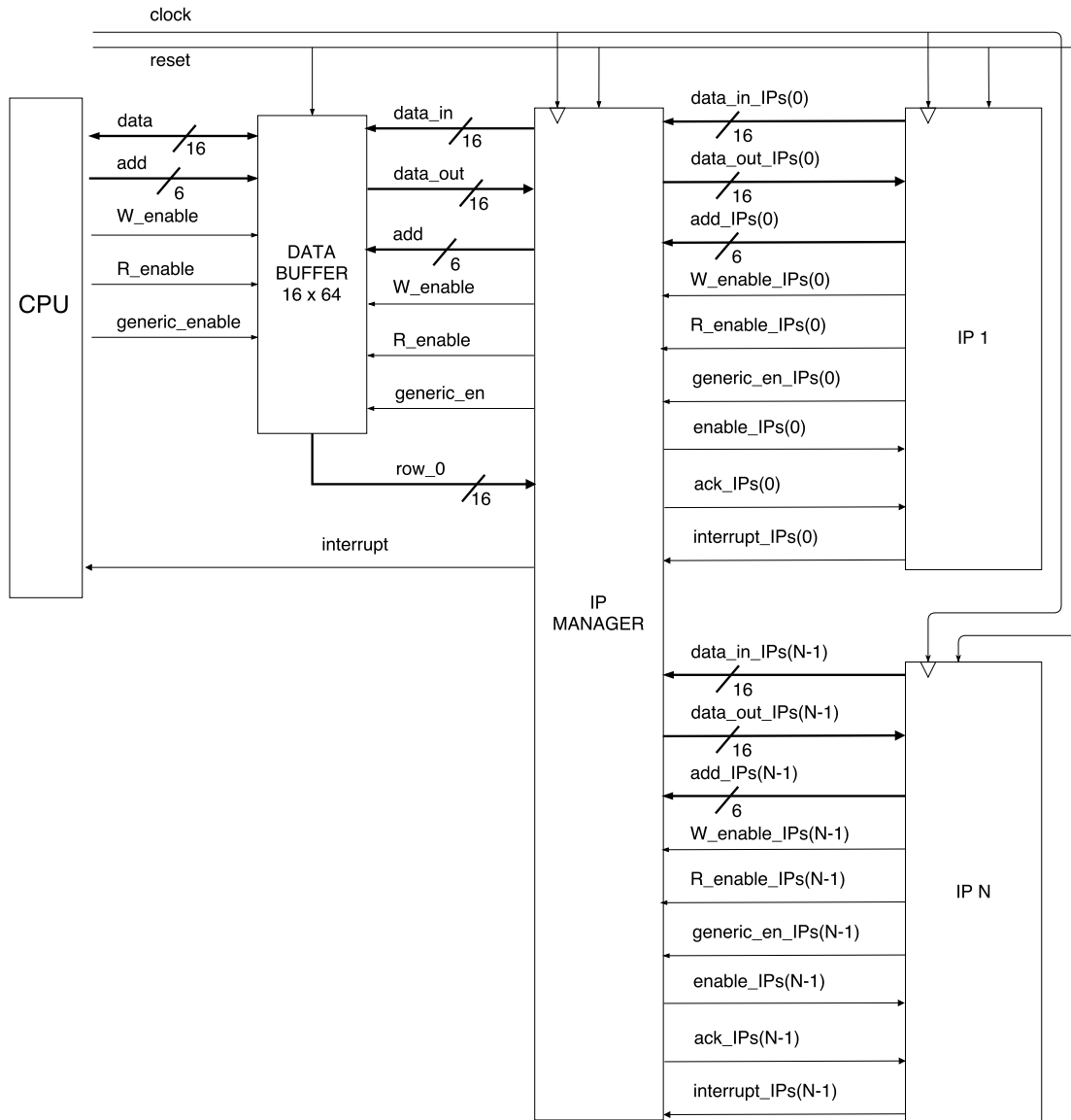


Figure 2.1: System architecture

2.3 CPU Control bits

When the CPU wants to start a transaction, it has to write a data packet with the following structure

15	14	13	12	11	0
UNUSED		INT	B/E	IP ADDR	

Bit(s)	Purpose	Value(s)
Bit 15	unused	unused
Bit 14	unused	unused
Bit 13	Interrupt ACK from the CPU	Normal = 0, Interrupt = 1
Bit 12	Signals the begin/end of a transaction	Begin = 1, End = 0
Bit 11-0	The physical address of the target IP	From 0 up to N-1

For a better understanding of the transaction mechanism, please see [chapter 4](#).

CHAPTER 3

IP-core Manager functionalities

3.1 Enabling the right IP

The first and basic task for this core manager is to enable the connection between the CPU and the selected core. More in particular this means exchanging the data in the required time as described in the protocol. The CPU can talk to one and only one core, therefore the core manager will disregard other cores whether they finished or not their task. We know that the CPU writes at row_0 what it wants to do. Here we can read the physical address of the chosen IP core. If we know that the 1st core is selected, than the IP manager has to enable the 1st core (i.e. $port_0$) and disable all the other ones, thus having the following signal:

$$\begin{aligned} enable_0 &= '1' \\ enable_1 &= '0' \\ enable_2 &= '0' \\ &\vdots \\ enable_{n-1} &= '0' \end{aligned}$$

We can put all these signal together as if they were an array.

To make this kind of operation, a conversion is required.

We can clearly see that only one bit is at '1', while the others are at '0', this is due because the CPU can talk at most to only one IP-core.

The feature of having at most one bit at '1' is the well known *one hot encoding*.

We implicitly implemented a "kind" of *binary to one hot encoding* converter.

At row_0 we read the physical address, and we raise the right enable signal, keeping the other ones at '0'.

Figure 3.1 shows this functionality.

A small adjustment has been done, since the physical address 0 is reserved to the IP

manager, whilst the *IP core 0* has as physical address *1*. This component is active during the rising edge of the clock and when the CPU wants to start a transaction.

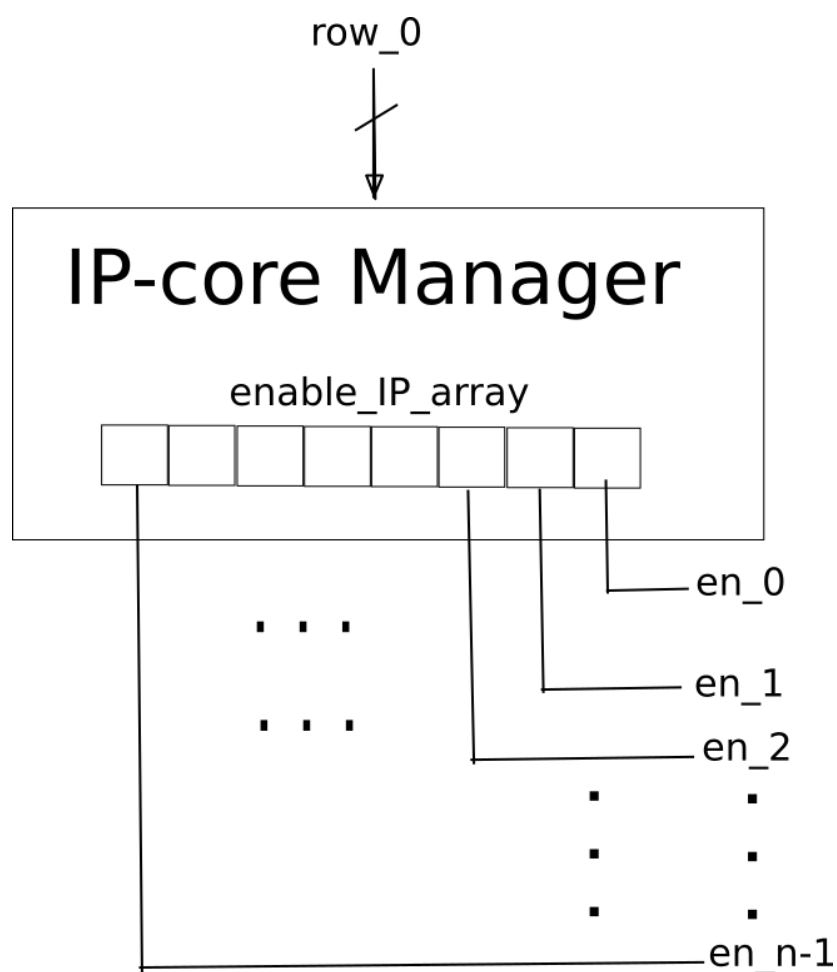


Figure 3.1: Enabling the right IP core

3.2 Connecting the signals

The IP core Manager has the duty to make possible the exchanging of data between the CPU (buffer) and the selected IP core, in both direction i.e. from CPU to the IP core and from the IP core to CPU. In order to select the right input among the N ones of the IP cores, we can just look at row_0 , bits $[11 : 0]$, because here there is the physical address of the selected core.

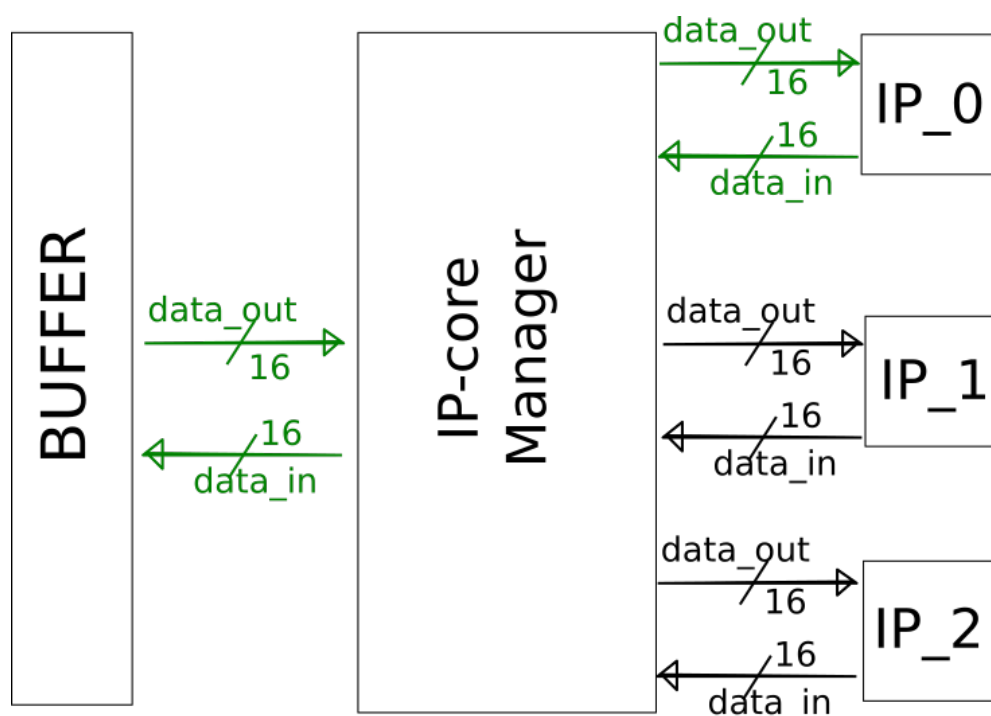


Figure 3.2: connecting the signal with IP_0 , supposing row_0 ask for the first IP core

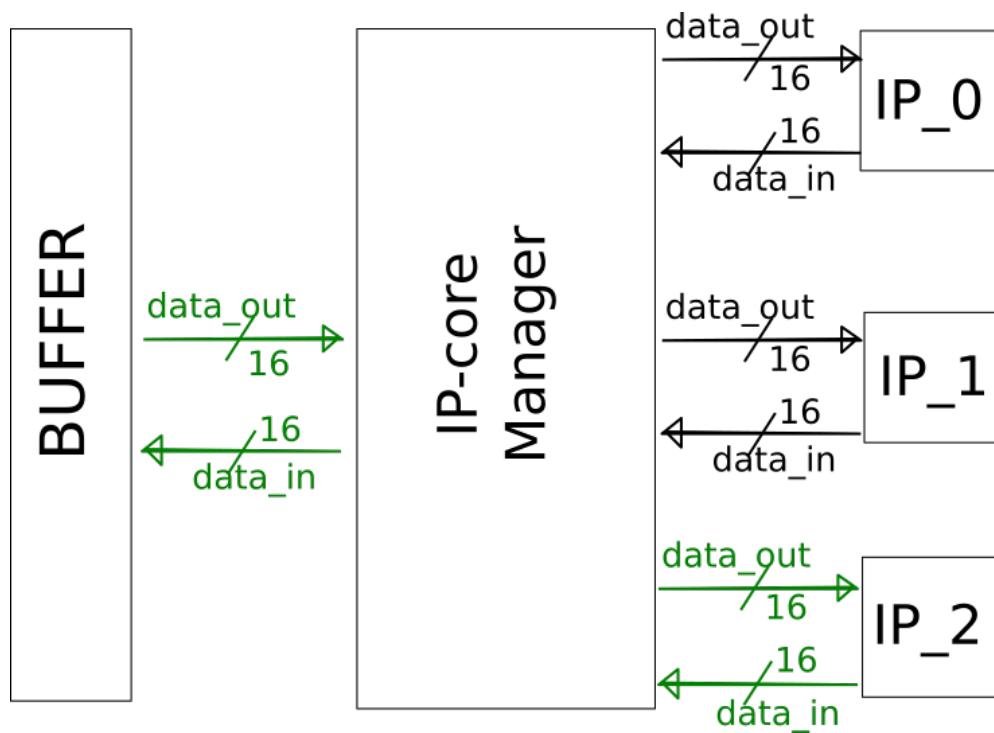


Figure 3.3: connecting the signal with IP_2 supposing row_0 ask for the third IP core

3.3 Interrupt Handler

The IP-core manager has to manage the case when a single or multiple core raise an interrupt request. However since the architecture is a master (CPU) slave (FPGA) architecture, the IP manager must guarantee that an interrupt request from one or more of IPs do not interrupt a transaction started by the master. In other words, the interrupt handler will be active at the end of a transaction.

In the case when multiple core raise the interrupt, the IP core manager will give priority to the core with the most priority level which is connected to the lowest port i.e. $port_0$.

In order to do so, we collect all the $interrupt_x$ signals of the cores into an array. Then we search for the LSB at '1'. Finally we convert the position of this bit into the physical address of the core requesting the interrupt.

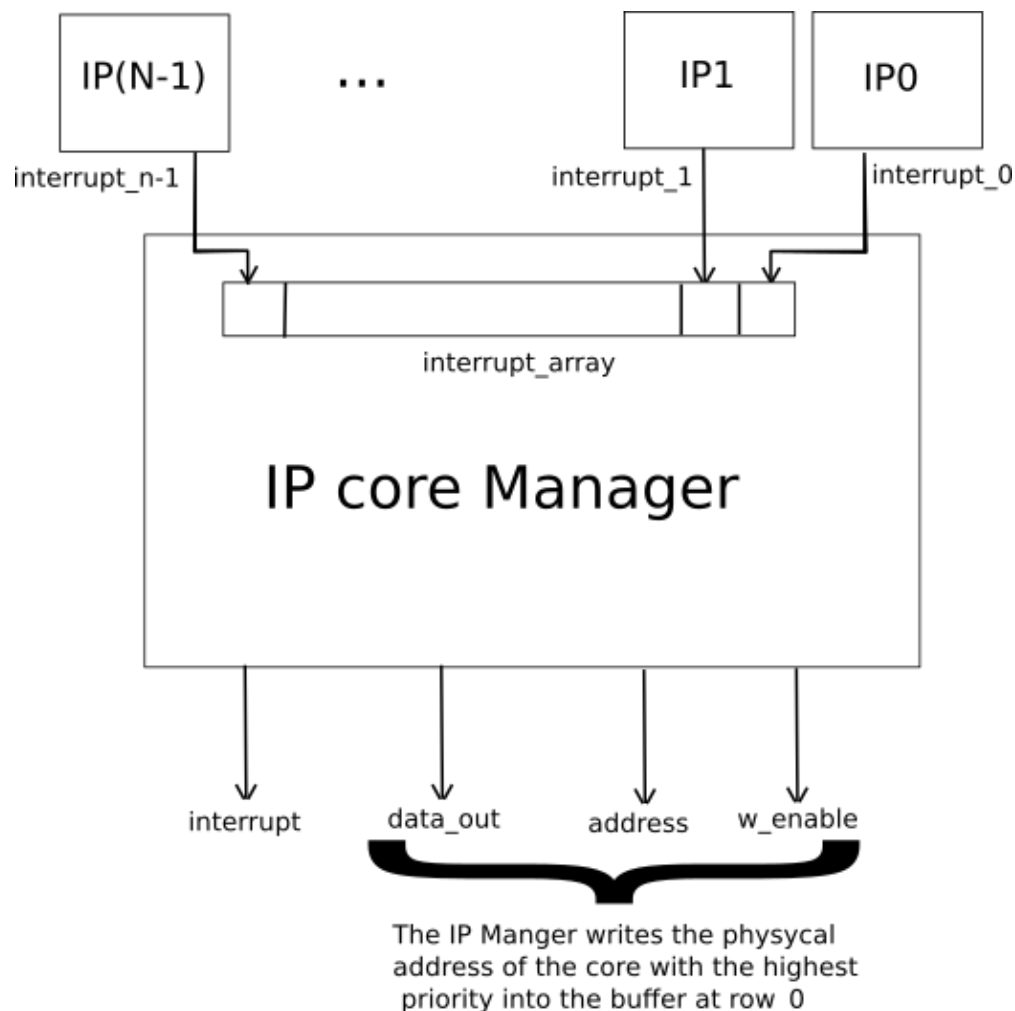


Figure 3.4: Interrupt Handler

The work of the IP Manager is first to check if there is any interrupt, this can be done by comparing the value of the interrupt vector. If the value is zero, that means that nobody raised the interrupt.

Otherwise it has to find the core with the highest priority.

In order to do so, it can iterate N times, i.e. from $i = N - 1$ to $i = 0$. In this loop it checks the bit in $Interrupt_array(i)$, if it is '1', then the variable i is the physical address of the IP core requesting an interrupt. However, we keep the loop, because it is possible to have an IPcore with an higher priority.

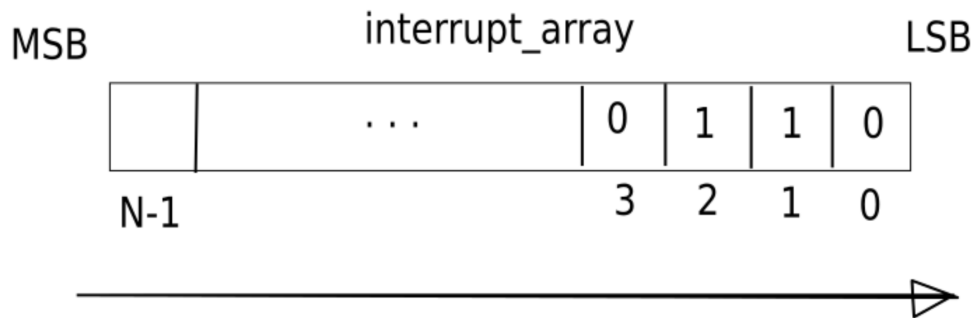


Figure 3.5: A possible interrupt vector

For instance, let's take the interrupt vector shown in fig. 3.5.

When iterating with $i = 2$, we want to send the address of the third core, but when we decrease the counter $i = 1$, we have to update this address. When $i = 0$ we leave everything unchanged.

CHAPTER 4

User Manual

4.1 Introduction

The IP-core manager is an interface for the integration of multiple custom IP designs in the secube platform. It handles context switching, interrupt handling and scheduling, guaranteeing the correct exchange of data between the target IP and the CPU.

This guide describes the interface of the IP core manager, and is meant to assist developers and hardware designers in correctly implementing custom IP cores as well as the software that interacts with them.

4.2 Interface to the CPU

The CPU talks to the IP-Core manager by the means of the data buffer. In the picture [2.1](#), you can see the interface of the Data buffer connected to the CPU.

After the cores are synthesized and running, the CPU can enable or disable the cores, load input data into them and read their output through the IP manager.

To do this, the CPU must initiate a transaction when it is ready to exchange data with a chosen core.

4.2.1 Initiating a transaction

Initiate or end a transaction by writing a 16-bit data packet at address row 0.

To begin the transaction, bit 12 must be set (=1); to end the transaction, it must be unset (=0).

15	14	13	12	11	0
UNUSED		INT	B/E	IP ADDR	

Bit(s)	Purpose	Value(s)
Bit 15	unused	unused
Bit 14	unused	unused
Bit 13	Interrupt ACK from the CPU	Normal = 0, Interrupt = 1
Bit 12	Signals the begin/end of a transaction	Begin = 1, End = 0
Bit 11-0	The physical address of the target IP	From 0 up to N-1

Specify the address of the target IP. Addresses are assigned sequentially from 1 to N.

To write the data packet, set the signals as follows:

Data: your instruction packet in the above format

Address: 0

W_enable: 1

R_enable: 0

Generic_en: 1

NOTE: *generic_en* must be set to 1 whenever communication with the IP manager is in progress.

NOTE: only one IP can be enabled at any time. If a new IP is enabled, all other IPs will be disabled, even if their transaction was not explicitly closed.

4.2.2 Writing data to the IP core

Once a core is selected, the IP manager becomes transparent to read/write operations between core and CPU. Each core has access to a range of addresses which is freely assigned by the IP core's designer. Then, it is simply a matter of reading and writing to these addresses according to the specifications of the selected core.

To write data, set the signals as follows:

Data: the 16-bit data to be sent

Address: the IP's read address (refer to the IP core's documentation)

W_enable: 1

R_enable: 0

Generic_en: 1

4.2.3 Reading data from the IP core

To write data, set the signals as follows:

Data: don't care

Address: the IP's write address (refer to the IP core's documentation)

W_enable: 0

R_enable: 1

Generic_en: 1

4.2.4 Ending a transaction

To end a transaction, follow the steps in 4.2.1, and set bit 12 of the instruction packet to 0. The IP core will be unselected and disabled.

4.2.5 Servicing interrupts

The CPU can receive interrupt requests from the cores. In this event, the *interrupt* signal will be raised. The IP manager will only forward interrupts when no transactions initiated by the CPU are currently active. The address of the IP requesting the interrupt is written in address row 0 by the IP manager.

When the CPU is ready to service an interrupt, perform the following steps:

1. Read address row 0 to find out the address of the requesting core.
2. Start a new transaction, selecting the requesting IP. Follow the steps in 4.2.1, with the only difference being that bit 13 of the control word must be set to 1. This is the ACK bit.
3. The IP core will now perform its routines. Refer to the IP core's documentation.
4. The transaction can be closed as in 4.2.4.

4.3 Interface to the IP cores

IP cores designed for the multi-IP core system must comply to the interface described in section 2.2.2 (see figure 2.1).

Note that the maximum number of IPs that can be managed by the system is 2^{12} . Addresses are assigned sequentially from 1 to $N = 2^{12}$.

The IP cores have at their disposal the system's clock and reset signals. It is recommended that the IP's processes are made sensitive to the enable signal and that a

suitable idle routine is written for when the core is disabled.

When the core is enabled, it can perform read and write operations. When the core is disabled, it can send interrupts to the CPU to request to be enabled.

4.3.1 Reading data from the cpu

When it is enabled, IP can receive data from the CPU by requesting a read from the IP manager. The range of addresses that the IP is expecting to read from can be chosen freely. However care must be taken to assign different r/w addresses to each IP in use, to avoid race conditions. The chosen addresses must be documented and communicated to the software designer.

In order to read data, set signals as follows:

address: the IP's read address
W_enable: 0
R_enable: 1
Generic_en: 1

NOTE: the IP manager inserts a 1-stage delay between the IP core and the CPU. The requested data will effectively appear on *data_out* with a delay of 1 clock cycle. The IP developer is invited to pipeline read requests or otherwise insert idle cycles to keep this delay into account.

4.3.2 Writing data to the cpu

The write address for the core can also be freely assigned, with the same caveats as in [4.3.1](#).

To write data, set the signals as follows:

Data_in: the data to be written
Address: the IP's write address
W_enable: 1
R_enable: 0
Generic_en: 1

4.3.3 Requesting an interrupt

When the core is disabled, it is not able to read or write to the buffer. Then, it can send interrupts to the CPU to request to be enabled. The IP manager will only consider

interrupt requests one at a time, and it assigns a static priority to the IPs according to their address; the lowest address has the highest priority.

To request an interrupt, raise the "interrupt" signal. The signal should remain high until the ACK signal is raised by the IP manager.

Once the CPU is ready to service the interrupt and no interrupts with higher priority are active, the IP manager will raise the *ack* signal. At the same time, the IP's select signal becomes high. When the IP receives the ACK, it should unset the interrupt signal; then it can perform its operations as during a normal transaction.

4.4 Use case scenario

4.4.1 example 1: adder IP

A user develops a simple adder. The adder receives two operands from the CPU, adds them, and delivers the result back to CPU.

The following buffer positions are reserved for the adder:

- Address row 1: OP1
- Address row 2: OP2
- Address row 3: Result

The CPU writes the operands to rows 1 and 2, then enables the core, selecting IP address 1.

The adder core is an FSM cycling through a few states:

1. **IDLE**: This state is entered at reset, and the adder stays in *IDLE* whenever *enable* = 0. In this state, the core continuously requests to read OP1 from address row 1; this way one clock cycle is gained when starting operation since one of the operands has already been requested.
2. **READ_OPERAND2** : This state is entered as soon as *enable* becomes 1. The core requests to read OP2 from address row 2.
3. **WRITE_OPERAND1**: During this cycle, OP1 is received on *data_out* and stored in a local register.
4. **WRITE_RESULT**: OP2 is received on *data_out*. The adder directly computes the results and writes it to the buffer's *data_in* at address row 3. After this state, the core returns to *IDLE*.

In order to complete the sum, the CPU keeps the transaction open for 4 clock cycles. After this time is elapsed, the CPU can read address row 3 and retrieve the result.

4.4.2 example 2: accumulator IP

A user develops a second core that accumulates the value of the data found in memory for 12 clock cycles. The accumulator should work while the transaction is off and send an interrupt when the result is ready; this operation is similar to the behavior of a sensor. This IP is used at the same time as the adder IP from example 1.

In order to avoid conflicts with the adder, buffer positions are allocated as follows:

- Address row 4: OP1
- Address row 5: Result

Since this core is the second to be loaded in the FPGA, it is assigned IP address 2. The CPU writes the first value to be accumulated into address 4, then selects and enables the core.

The accumulator core is an FSM cycling through states:

1. **IDLE**: This state is entered at reset, and the adder stays in IDLE whenever `enable = 0`. In this state, the core continuously requests to read OP1 from address row 4.
2. **OP_START**: This state is entered as soon as `enable` becomes 1. The request to read OP1 has already been sent, but the `data_out` signal is not yet ready. The IP initializes a counter to 12.
3. **ACCUMULATE**: `data_out` becomes valid, and OP1 is saved in a local register and in the accumulate register. The CPU can close the transaction. While in this state, the IP continuously adds OP1 to the ACC register and decrements the counter every clock cycle. Only once the counter rolls down to 0, it sets interrupt to 1 and continues to state 4.
4. **WAIT_ACK**: The interrupt has been sent on the last iteration of state 3. This state polls the `ack` signal every clock cycle. When the CPU is ready to read the result, it opens a transaction with the accumulator core while setting the ACK bit in the control packet. Once `ack` becomes 1, the IP sets interrupt to 0 and proceeds to state 5.
5. **WRITE_RES**: The result is written to address row 5. After this state, the IP returns to IDLE.

In order to initiate operation, the CPU keeps the transaction open for 2 clock cycles to allow for the read of OP1. When reading the result, the transaction stays open for 3 clock cycles, after which the result appears in the buffer.

CHAPTER 5

Guidelines

- To reduce the latency, it's better if the synchronous IP cores start with the interface signals “loaded” with the address of the first data to read.
- Another way to reduce the latency is to have the IP cores working on the falling edge of the clock instead of the rising edge like does the IP manager.
- Synthesizer like Vivado and Lattice don't like very much asynchronous memories like our Data Buffer. Usually they automatically introduce latches. Be aware that these can introduce differences between the simulation behaviour and the behaviour on FPGA.

CHAPTER 6

Future developments

6.1 Asynchronous is the new synchronous

As we have seen in the previous chapters a synchronous IP manager introduces several clock cycle of latency in the overall transaction between CPU and the cores.

A possible future development to remove this latency is to change from the current clock based, sequential implementation to a full combinational one.

6.2 Configurable manager

We have reserved the address 0 for the IP manager, in case the CPU wants to *configure* it. Which possible configurations are available? None! That's up to you to implement it.

6.3 Interface for the IP cores

Every IP in the world is designed with a custom interface, and they are usually all different.

A possible future development is to analyze if it is possible to realize a hardware component capable of adapting whatever IP interface to the interface compliant with this architecture and described in this document.