



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	Use case scenario	12

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 enables 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 performs 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

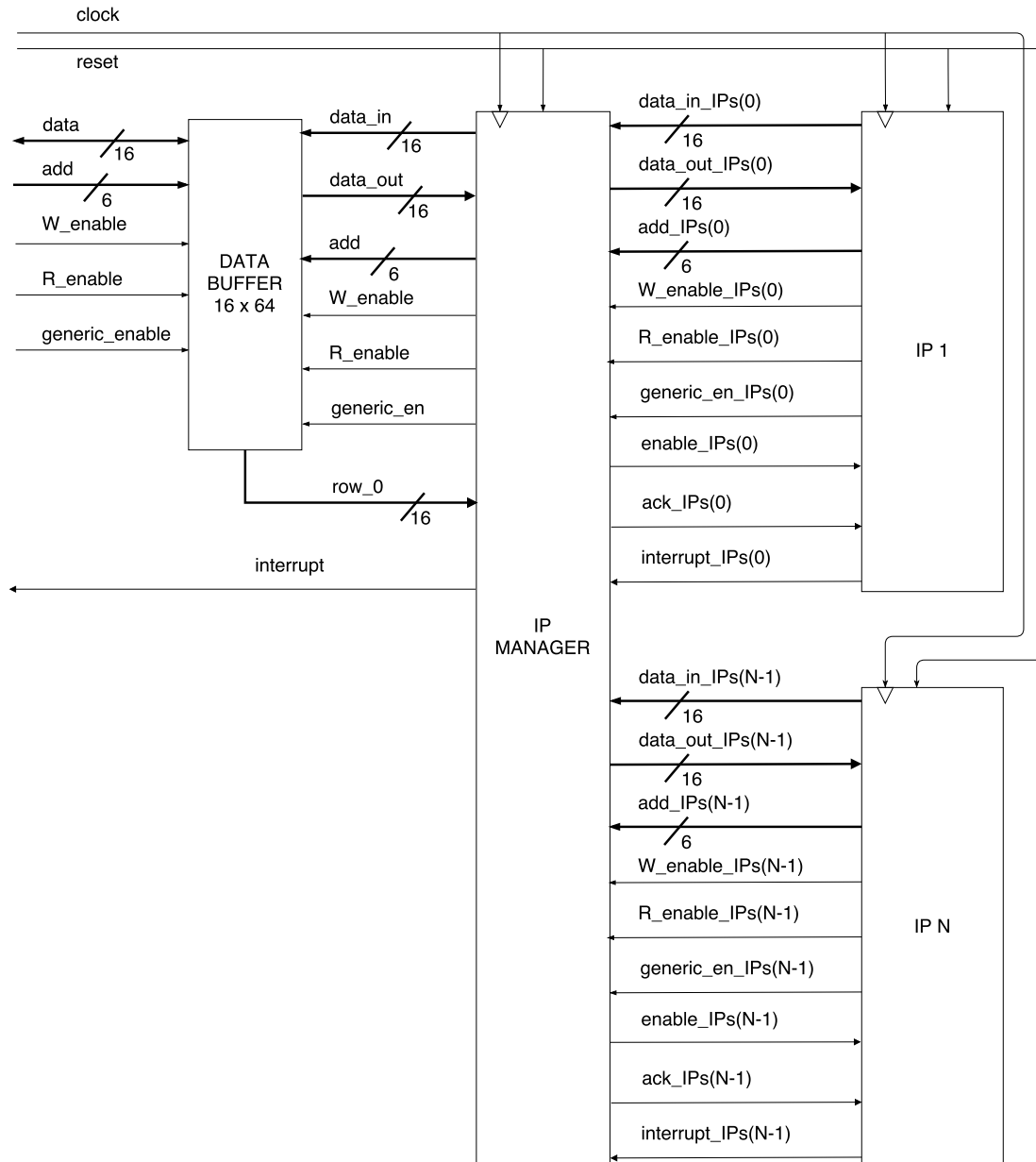


Figure 2.1: System architecture

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. This is the VHDL code that perform this task

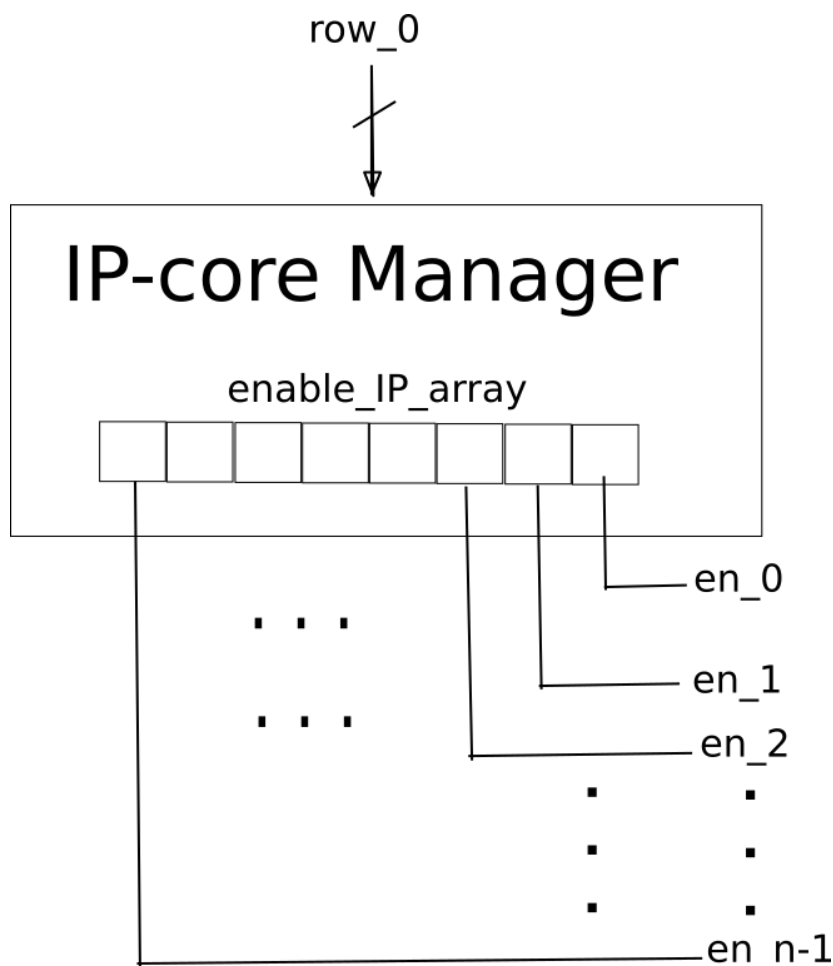


Figure 3.1: Enabling the right IP core

```

1 -- Begin ( or continue ) transaction:
2 if row_0(BE_POS) = '1' then
3   enable_IPs(conv_integer(row_0(IPADD_POS downto 0))-1) <= '1';
4   data_in <= data_in_IPs(conv_integer(row_0(IPADD_POS downto 0))-1);
5   data_out_IPs(conv_integer(row_0(IPADD_POS downto 0))-1) <= data_out ;
6   add <= add_IPs(conv_integer(row_0(IPADD_POS downto 0))-1);
7   W_enable <= W_enable_IPs(conv_integer(row_0(IPADD_POS downto 0))-1);
8   R_enable <= R_enable_IPs(conv_integer(row_0(IPADD_POS downto 0))-1);
9   generic_en <= generic_en_IPs(conv_integer(row_0(IPADD_POS downto 0))-1);

```

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.

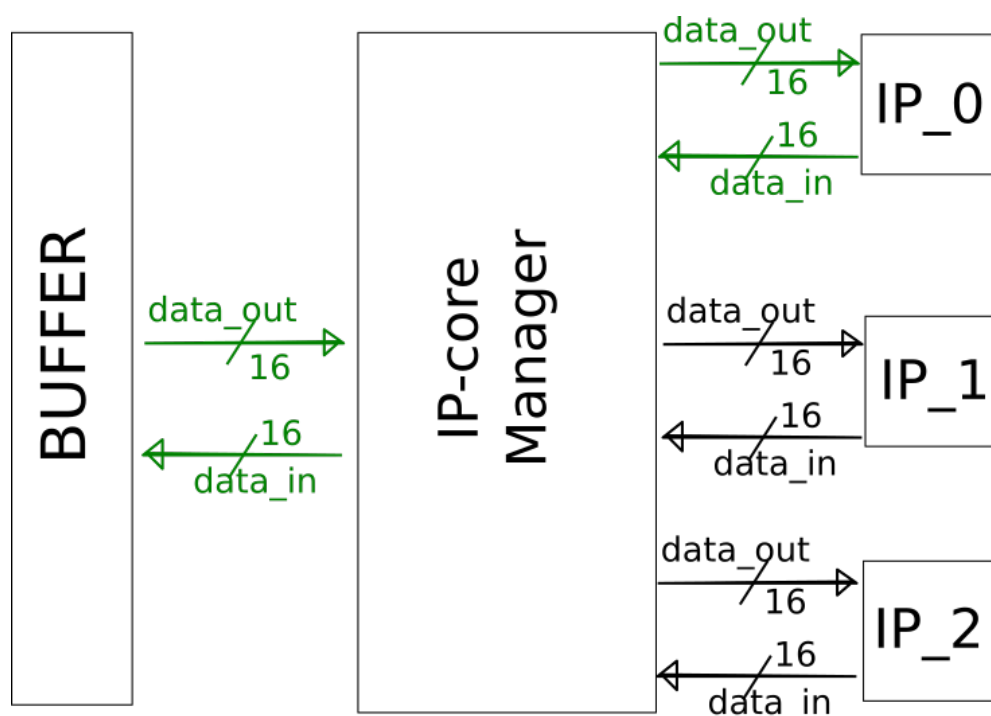


Figure 3.2: connecting the signal with IP_0 , supposing row_0 ask for the first IP core

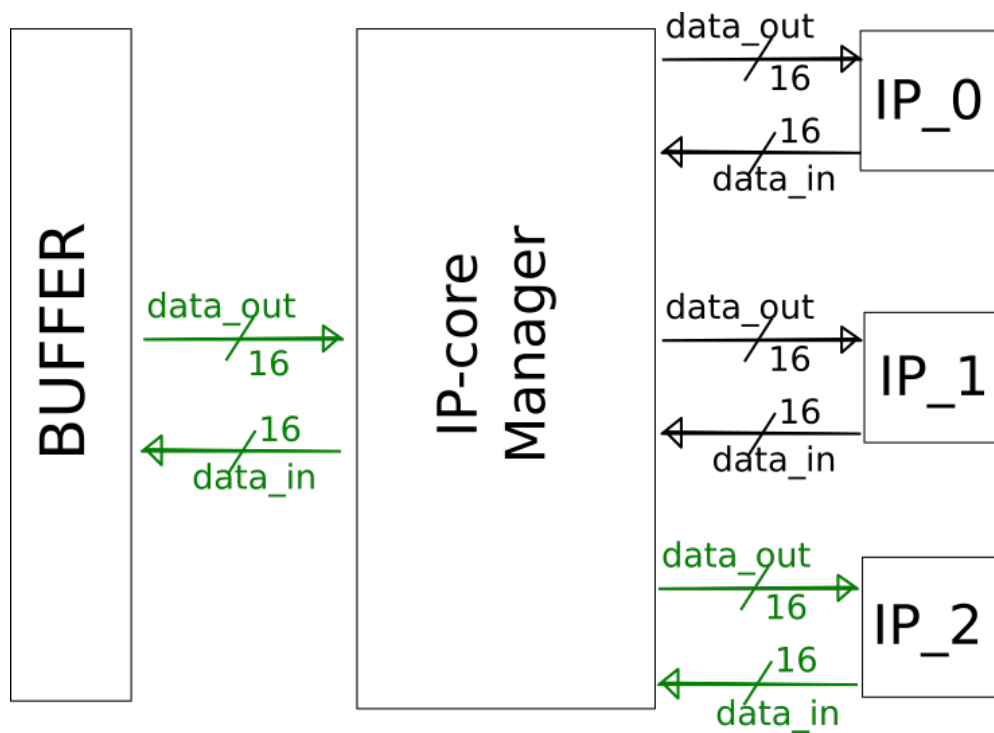


Figure 3.3: connecting the signal with IP_2 supposing row_0 ask for the third IP core

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.

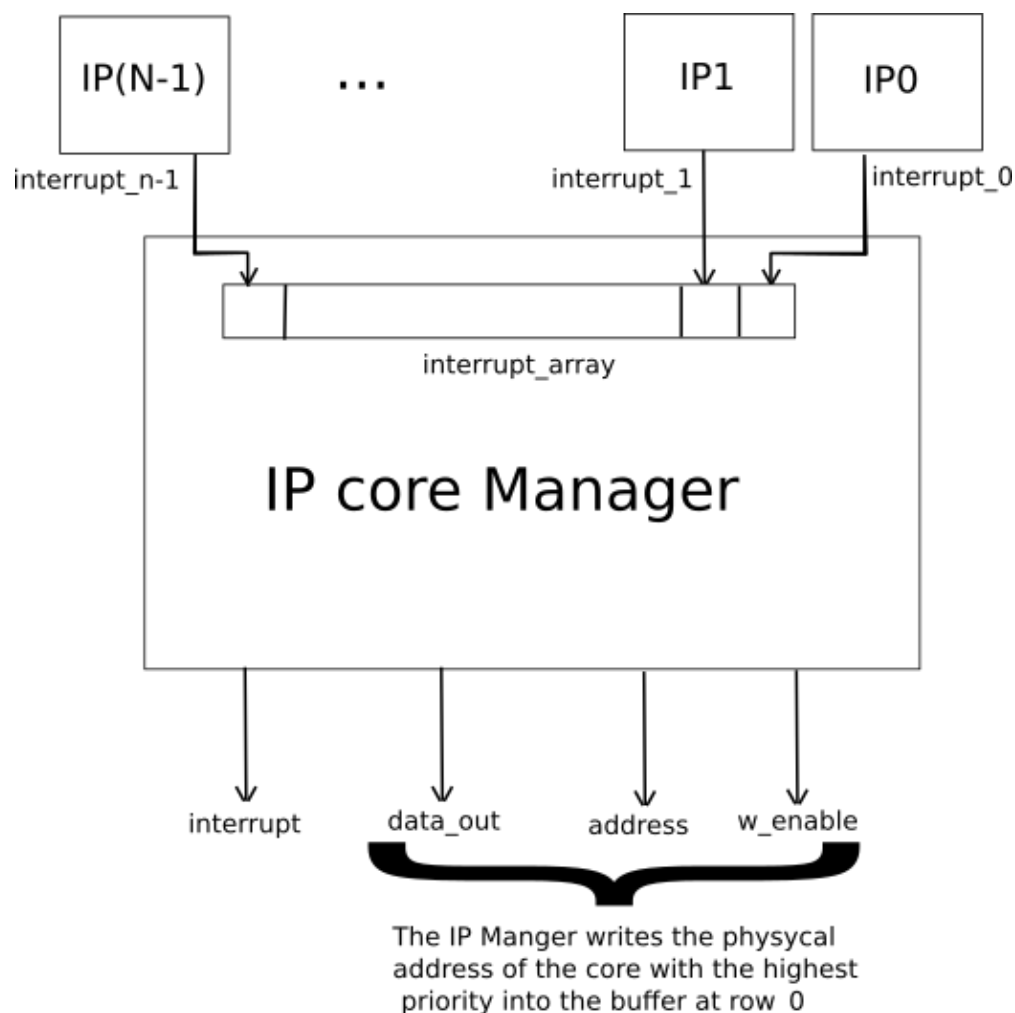


Figure 3.4: Interrupt Handler

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.

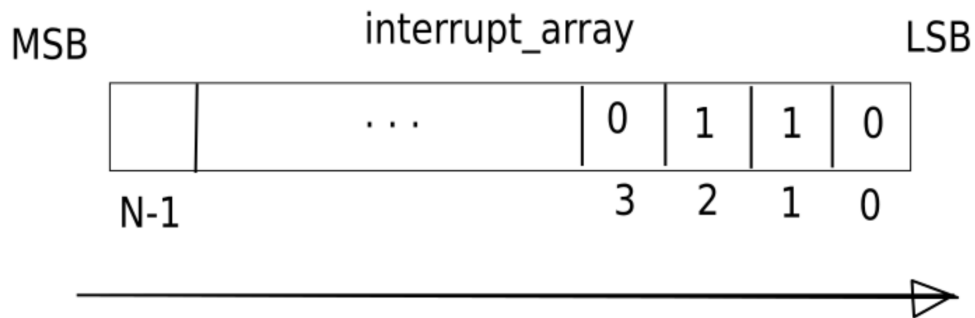


Figure 3.5: A possible interrupt vector

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

Use case scenario