Eindhoven University of Technology
Department of Electrical Engineering

DESIGN OF COMPONENTS FOR A NoC-BASED MPSoC PLATFORM

Adding a shared memory node to the mNoC

José C. Prats Ortiz

June 30, 2005

Master of Science thesis

Project period: October 2004 – June 2004

Department of Electrical Engineering
Capacity group: Information and Communication Systems
Chair: Electronic Systems

Supervisors:
Prof. Dr. H. Corporaal
Dr. Ir. B. Theelen

To my parents, Pepe and Piluca;

and my two sisters, Patricia and Mª Elena.

Thank you because I would not be writing

these words without your support.

List of Contents

List of Figures

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

List of Tables

Abstract

Nowadays, technology is introducing new multimedia applications that should run concurrently on a platform with limited resources into market. Decoding JPEG pictures, MJPEG video, or MP3 music or executing computer games require a large computational load, and therefore a specific Hardware. Until recently, multimedia systems were implemented in fixed ASICs (*Application Specific Integrated Circuit*).

The design process of such systems has to cope with continuously changing multimedia standards in a short period of time: the Hardware should be flexible to changes. For this reason, research is focusing on reconfigurable architectures. Computer architectures capable of running this kind of applications usually have several components such processors, memories, video processing units or I/O nodes. One of the most important things in these platforms is the communication between these components.

A solution to flexibility and reconfigurability of interconnects are *Network on Chip* (NoC). These systems consist of a number of nodes (processors, memories, etc.) which are directly connected with each other trough an on-chip network. The network is composed of a set of routers that handle communication between nodes.

In this Master Thesis a Network on Chip based system is studied and improved to run a *multi-processor Motion JPEG decoder* on it. The processor nodes have been improved to reduce simulation and execution time. Two extensions were made on the NoC: the network was enlarged from 4 nodes to 6 and a new shared memory node was added. Finally, a parallelized Motion JPEG Decoder has been constructed from a JPEG decoder.

This report will give an overview of old and new features of the NoC as well as results and evaluation of them with different test applications.

Chapter 1

Introduction

1.1 What is a SoC

Trying to find a simple definition, System on Chip (SoC) could be defined as *an IC, designed by switching together multiple stand-alone VLSI designs to provide full functionality for an application* [1][21]. A SoC consist of modules that perform certain complex functions. They are known as *cores* and they are the essential SoC components. Therefore, a SoC could be composed by several different types of *cores,* such as microprocessors, large memory arrays, graphics controllers, digital signal processing (DSP) units, etc. All these *cores* could be defined using synthesizable high-level description language (e.g. VHDL, Verilog) or with optimized-transistor level layout. It is possible to classify *cores* as follows:

- *Soft cores*: These are reusable blocks in the form of synthesizable RTL description or a netlist of generic library elements. This implies that the user of a *soft core* is responsible for the actual implementation and layout.
- *Firm cores*: These are reusable blocks that have been structurally and topologically optimized for performance and area through floorplanning and placement techniques. They are available as synthesized code or a netlist of generic library elements. In these *firm cores*, the design can be customized by the end user.
- *Hard cores*: These are reusable blocks that have been optimized for performance, power, and size; and then mapped to a specific process technology. They have a predefined layout that cannot be modified by the system designer. In this case, the *hard core* is used as a library element during the design cycle.

The differences between the three kinds of *cores* could be measured in terms of *reusability, flexibility, optimized performance, cost* and *time to market*. The following figure shows the classification in those terms:

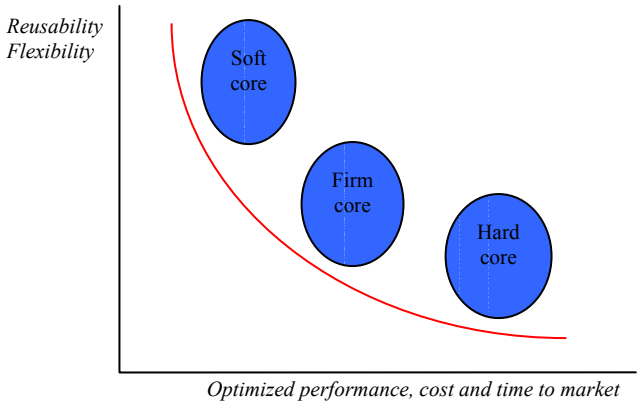

Figure 1.1: SoC classification in terms of reusability, flexibility, optimized performance, cost and time to market [2]

Because of the fact that this kind of systems could be composed of different *cores*, the SoC design may contain a very high level of integration complexity, interfacing and synchronization issues, data management issues, design verification, and test, architectural, and system-level issues.

1.2 Current SoC design trends

Current technology is focused to embedded systems, and they represent the major market for microprocessors [17]. One of the reasons because this is happening is that intense competition among manufacturers has decreased the life cycles for products that use embedded systems.

Together with this fact, product functionality and hence the complexity of embedded systems are rapidly increasing. For instance, mobile telephones are continuously including more and more applications on them, such as Internet browsing, MP3 and video players and even games.

Shorter product life and increased functionality makes that silicon technology will allow chip complexities up to 1 billion transistors on a single chip within this decade [3]. The embedded systems will make profit of this improvement just if it is possible to create a full system (with processing elements, memories, communication infrastructure, etc.) on a single chip (SoC).

Early embedded systems SoCs were low complexity systems and just with one or two programmable processing units together with custom hardware, peripherals and memories. In current SoCs, the new amount of transistors is used to perform embedded memories, but the major part of them will be used for increasing the number of on-chip processor units in order to increase system performance. Using all the available transistors it could be possible to fit several hundreds or even thousands of embedded processors in a single chip. So, the current design trend is increasing the number of processing units replacing old custom logic blocks [18].

Nowadays complex SoCs feature 10-15 programmable processing units on a single chip. Unfortunately, it is not possible to exploit the maximum possible amount of transistors per chip yet. The number of transistors requirements of new SoCs is clearly behind of the current capabilities of silicon technologies. This fact is shown in *Figure 1.2*:

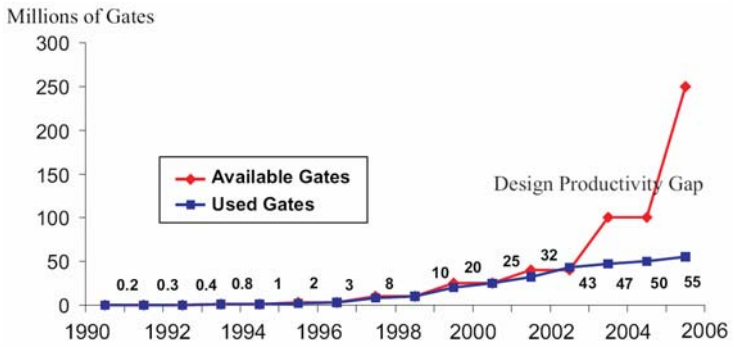

Figure 1.2: Predicted design productivity gap [4]

Figure above shows the predicted design productivity gap. It is measured as the number of available gates per chip for a given silicon technology (*red line*) in one side and the number of actually number of used gates per chip for a given silicon technology (*blue line*) on the other side.

However, application areas for 1 billion transistors SoC are becoming more used. Security systems, control systems, individual health systems, entertainment or personal computing are only some examples where a tendency to use larger functionality and more complex SoCs is observed.

This trend will significantly change the way SoCs are designed, both from a design methodology point of view as well as from an architectural point of view: hundreds or thousands of processors will be integrated on a single chip and they will need to be connected by an efficient communication infrastructure.

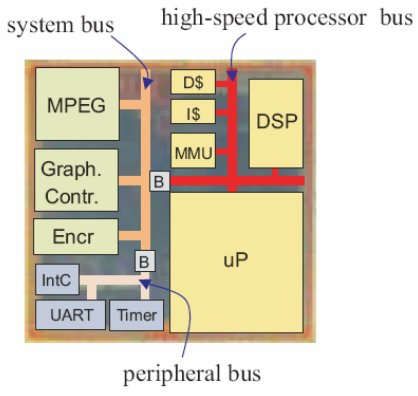

Figure 1.3: Example for a typical bus-based SoC

Bus-based communications, even the improved ones using hierarchies of buses (see *Figure 1.3*) will not be enough for several reasons:

- A single bus can not have more than one access at the same time, which means that concurrent transactions are not allowed. Depending on the arbitration algorithm, one transaction will have access to the bus at a certain moment, and the rest will have to be postponed even when it could be possible to run in parallel.
- Clock skew is the main problem in large bus-based SoCs. Combination of high clock frequency and large SoC designs is an untenable problem in a bus-based SoC.

Trying to find solutions to buses problems, some researchers have borrowed ideas from the Internet communication due to its scalability and success. Switch-based (routers) networks and packet-based communication for on-chip communications represents a good alternative to bus-based systems. Most of the basics and terminology used for on-chip packed-switched communication are extracted from computers networking area. This kind of communication provides scalability and allows the possibility of standardization and reuse of the communication architecture. These features give lower design effort and chances to meet time to market constraints to designers.

1.3 Network on Chip (NoC)

1.3.1 OVERVIEW

Network on Chip (NoC) is becoming a solution to nowadays bus-based communication infrastructure limitations. NoC is a set of routers that compose a network, which allow all the nodes connected to it (containing system resources, *cores*) to communicate ones with each others [17][20].

Functionality of a NoC like the one shown in *Figure 1.4* could be the following: data split in packets could be sent from an origin *IP* to the input channel of a router via *links*. These packets are *switched* inside the router to the output channel and driven to the next router. When a packet reaches its destination address, it is not switched to the next router, but it is switched to the *IP* attached to this router.

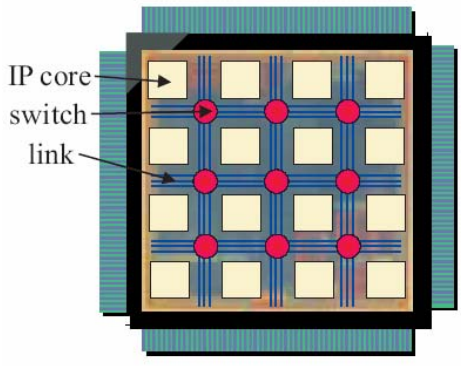

Figure 1.4: Example of NoC with IP cores links and switches.

Some features make NoC a good solution approach to the bus-based SoC problems:

- It transmits packets instead of words. Depending on the routing algorithm, packets can travel through the network via several paths because the destination address is part of the packet. Therefore, dedicated address lines like in bus-based systems are not necessary.
- In this case communication infrastructure does not block the system. Using a NoC, transactions could be made in parallel if the network provides more than one transmission channel between two different nodes of the network.
- Routers in the Network solve the clock skew problem when large buses and high clock frequency are used.

1.3.2 ON-CHIP INTERCONNECTION NETWORKS

On-chip communications could be studied attending to several aspects such as *interconnection network topology*, *switching*, *routing*, *flow control*, *queuing*, and *scheduling* [5][13]. All these aspects could be deeply explained, but only some of them will be introduced in this section in order to understand some terms used in further chapters. For this Master Thesis purposes, it will be useful to introduce network topologies like *n-dimensional meshes* and *k-ary n-cube*. Also flow control mechanisms like the one used in *Wormhole routing* will be discussed later on, and finally, the system presented in *Chapter 3* tries to avoid routing problems and that is why they will be explained as well.

1.3.2.1 Interconnection network topologies

This term defines how the nodes are interconnected via channels to other nodes inside the network. Usually these topologies are represented as a graph. In *Figure 1.5* it could be observed a classification tree of the most popular interconnection topologies for multiprocessor architectures. The ones related with this project are highlighted.

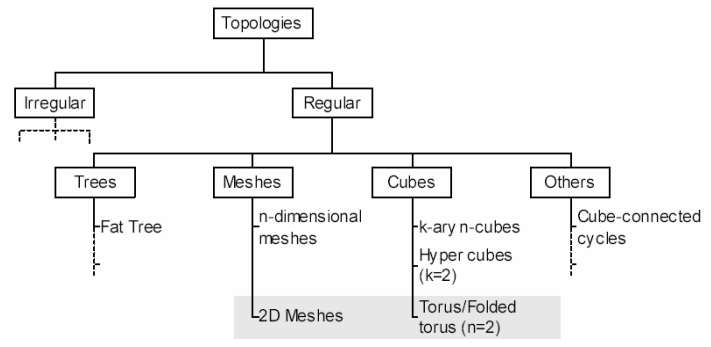

Figure 1.5: Topologies classification

n-dimensional Meshes

N-dimensional Meshes together with *k-ary n-cubes* are the most used interconnection choices because their regular topologies simplify routing.

In *n-dimensional Mesh* the nodes are distributed in n-dimensional array. Along each dimension there are k_i nodes ($k_i \geq 2$), where $0 \leq i \leq n-1$ is the dimension number. So the total number of nodes is $k_0 \times k_1 \times \ldots \times k_{n-2} \times k_{n-1}$. Each node x is defined by n coordinates (which are the joints between nodes in *Figure 1.6*), $\sigma_{n-1}(x)$, $\sigma_{n-2}(x)$, …, $\sigma_1(x)$, $\sigma_0(x)$, where $0 \leq \sigma_i(x) \leq k_i - 1$ for $0 \leq i \leq n-1$. Two nodes x and y are neighbours if $\sigma_i(x) = \sigma_i(y)$ for every i, $0 \leq i \leq n-1$, except one, j, where $\sigma_j(x) = \sigma_j(y) \pm 1$. Thus, nodes have from n to $2n$ neighbours, depending on their location in the mesh. In the mesh there is only direct connection between neighbours.

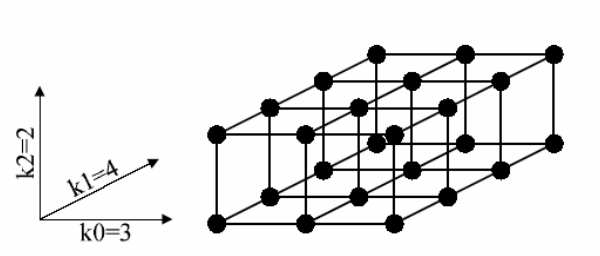

Figure 1.6: Example of 3-dimmensional mesh

A problem with the mesh networks is that assuming uniform traffic between nodes, the channels near the centre of the mesh tend to have higher traffic density than channels in the periphery.

k-ary n-cube

In *k-ary n-cubes* the nodes are arranged in n-dimensional array as well, but here, in each dimension there is an equal number of nodes k, so $k_0 = k_1 = \ldots = k_{n-2} = k_{n-1} = k$. The total number of nodes is k^n. Another difference with the *n-dimensional meshes* is the definition of neighbour nodes. Here, two nodes x and y are neighbours if $\sigma_i(x) = \sigma_i(y)$ for every i, $0 \leq i \leq n-1$, except one, j, where $\sigma_j(x) = (\sigma_j(y) \pm 1) \bmod k$. The use of modular arithmetic in the definition results in wrap around channels. They double the bisection width[1] and reduce the network diameter[2] in respect to the mesh. If $k > 2$, then every node has $2n$ neighbours, if $k = 2$, then every node has n neighbours

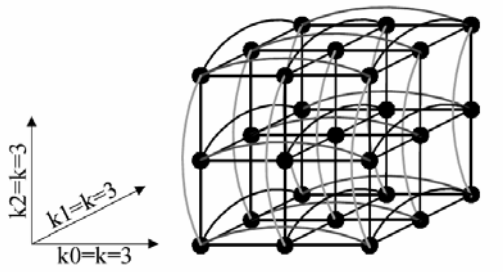

Figure 1.7: Example of 3-ary 3-cube

[1] *Bisection width of the network*: the minimum number of channels that must be removed, or cut, to partition the network into two sub networks, each containing half the nodes in the network.
[2] *Network diameter*: the maximum distance over all pair of nodes in the network.

Many graphs (e.g. mesh among others) can be used to create a *k-ary n-cube* (because of its recursive structure) and because of this reason, there is a large class of algorithms used in those graphs that can be applied in this topology.

Unfortunately, 2D implementation of this topology could give as a result an inefficient layout with a high wire density.

Low dimensional networks, topology used in this project

When *n=2, n-dimensional mesh* reduces to 2D mesh, and *k-ary n-cube* reduces to 2D torus. This dimension is convenient for planar implementations because they fulfil requirements such as scalability, efficient layout and energy-efficiency comparing with other topologies.

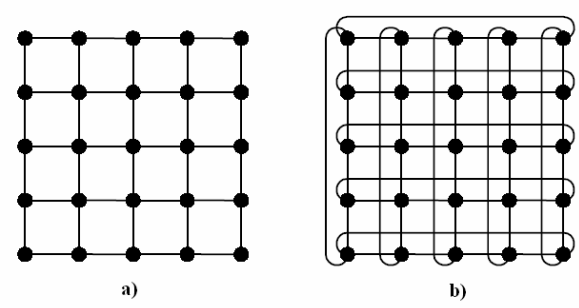

Figure 1.8: a)Example 2D mesh topology, b)Example torus topology

As it will be introduced in section *3.2*, the network topology used in this project is a 2D *torus network* with two unidirectional virtual channels on each physical link. Refer to *Chapter 3* to know more details about architecture and functionality of the network.

1.3.2.2 Flow control mechanisms

Flow control manages the channel and buffer resources availability for a certain packet that travel through the network from a source to a destination. One of those resources could be used for any other packet and it is task of the flow control to block it, store this packet in some buffer or try to find an alternative path for it.

There are several flow control mechanisms, but only *Store and Forward* and the one used in this project, *Wormhole routing*, will be presented. Both mechanisms will be explained in order to be able to compare them and knowing which are the improvements introduced by the second one.

Store and Forward

In this case, a message is fully received in one of the routers before it starts to forward the message to the next router. If L is the message length, W is the channel width and T_C is the channel cycle time, then the transmission time for single message through n consecutive channels is:

$$T = (n \cdot \frac{L}{W}) \cdot T_c$$

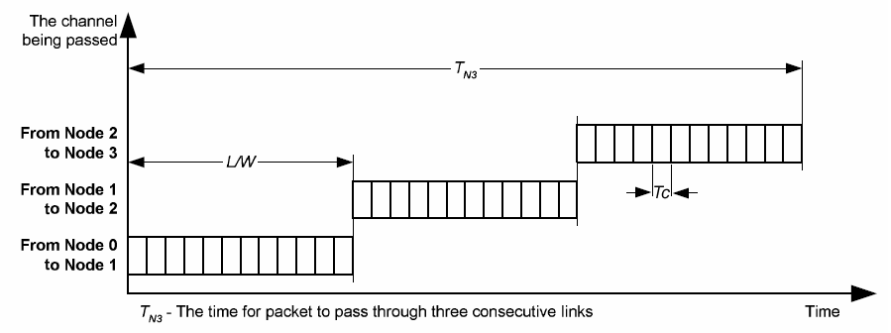

Figure 1.9: Store and Forward latency

Latency with this method is proportional to the packet size; therefore the main inconvenient of this method is that it implies important buffer size requirements.

Wormhole

In this method the message is divided into small parts called *flits* (flow control digits). Unlike the previous method, now only one or some flits are stored and forwarded, so it is not necessary to wait until all the message have arrived to start sending it to the next router. The first flit *opens* a path, which all the rest will follow. Just the last one will close the path.

Flits are smaller than a packet, which leads in improvements in bandwidth and storage allocation even for large packets.

Using the same terms than in *Store and forward* routing, the transmission time for single message through *n* links will be:

$$T = ((n-1) + \frac{L}{W}) \cdot T_c$$

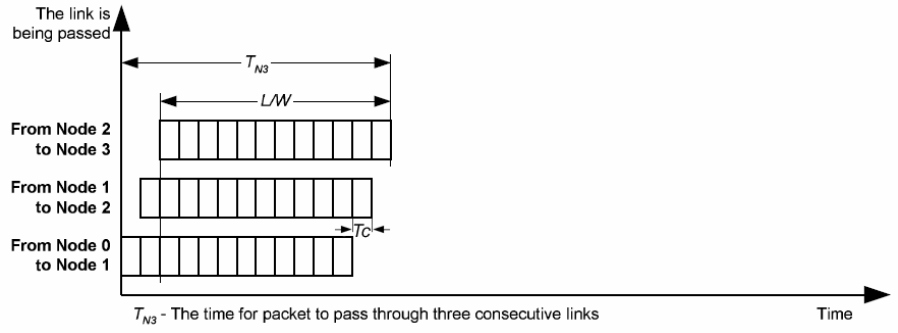

Figure 1.10: Wormhole latency

The main inconvenient of this method is that since the first flit open a new path, all these resources will be used until the last flit releases them. Therefore, if one flit is blocked those physical channels will be idle and they can not be used for others.

On the other hand, it is only necessary to send destination information in the first flit because all the rest will use the path already created. For this reason, *Wormhole* permits to send *just data* in the flits that follow the first one, making more profit on the bandwidth on this way.

1.3.2.3 Routing

All the nodes connected to a network should be able to communicate with each other. The routing algorithm has to find a path for every packet that travels from one node to another of the network.

During the routing process some problematic situations might occur:

- *Deadlock*: it occurs when a packet is waiting for an event that can not happen (usually when a packet is waiting for a network resource to be released).
- *Livelock*: it occurs when a packet never reaches its destination and stay indefinitely inside the network.
- *Indefinite postponement*: it occurs when a packet waits for an even that could happen but never does.

It should be mentioned that routing algorithm used in this project, *e-cube routing* (see section *3.2*), guarantees deadlock free in a *2D torus network*.

1.4 Thesis organization

This Master Thesis is divided in four parts:

- A brief *introduction* of many terms that will be used later on has been presented in this first chapter. To finish with the first part, the *objectives* of this project will be explained in the second chapter.
- The second part is a deep explanation of the system that was the basis of this project, and it is presented in *Chapter 3*.
- From *Chapter 4* to Chapter 7 the new system created and the results obtained with it are presented, composing the third part of the project.
- Finally, the fourth part will come with Chapter 8, which contains conclusions, perspectives and recommendations formulated attending to the obtained results.

Chapter 2

Objectives

2.1 Introduction

The main objective of this Master Thesis is to extend the *mNoC* (mini NoC) (see *Chapter 3*) [6] platform such that a number of (parallelized) MJPEG (Motion JPEG) decoders can be run in parallel. To this end, both the on-chip network component and the mMIPS [16] processor cores need to be extended as well as memory and I/O nodes have to be added to the platform (*Figure 2.1*).

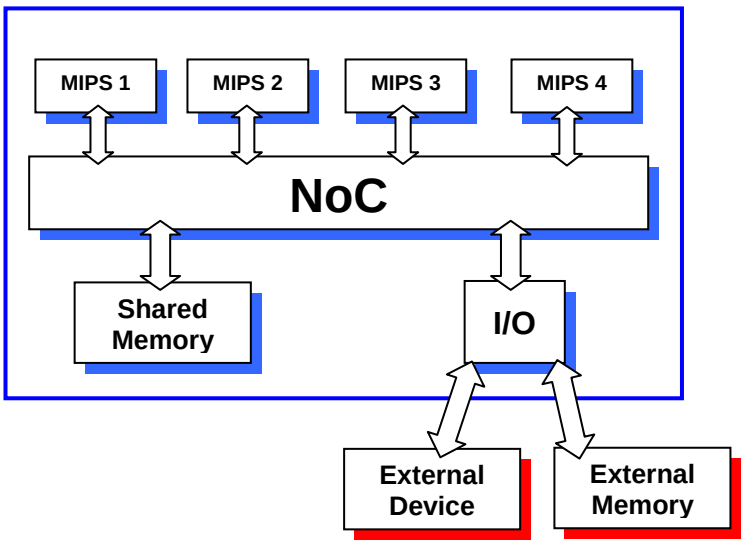

Figure 2.1: Pretended mNoC

An additional objective is to improve the simulation speed and debugging facilities of the used tools in order to make these tools more accessible for users.

In this chapter, the reasons why some changes in the original platform have been made are explained, however for a full description of *mNoC* platform refer to *Chapter 3,* where its features, architecture and functionality are deeply presented.

2.2 Work description

Previous studies, changes, extensions and simulations have been done over the *mNoC* platform in order to reach the main objective presented above. These intermediate objectives will be explained in this section.

2.2.1 EXTENDING THE mMIPS

The *mMIPS* (mini MIPS) processor is a simple MIPS version with a reduce instruction set. Multiply function is not among this set, and the lack of it implies restrictions on the applications that can be simulated for and run on the mNoC. In the *original mMIPS* the multiply function is done in software and therefore a lot of function calls are executed. This fact implies large code size and a longer run time.

For this reason, one of the objectives for this project in to investigate the mMIPS processor architecture and then extend it with the necessary instructions (including a multiply) to improve these aspects before mentioned.

Once this implementation is achieved, an interesting result is simulating the *JPEG decoder application* (one of the test applications for the *mNoC*, refer to *Chapter 3*) with and without multiply function on the mMIPS in order to measure the processor speed-up.

2.2.2 EXTENDING THE mNoC

2.2.2.1 Extending the Network

The mNoC used in this project is composed for four nodes with one mMIPS processor on each connected using a *torus* network with *E-cube routing* (see section *3.2*).

In order to achieve the main objective exposed before, some extensions on the mNoC must to be done. As it will be explained in the following sections, it will be necessary to add two *extra* nodes allowing the processors to handle more data on this way. For this reason, the network must to be extended from 4 nodes to 6 to support these extra nodes.

As well as changes on the SystemC description of the network, changes in the simulator should be performed to check the mNoC functionality before the physical implementation on the FPGA.

Finally, it is also necessary to upload the FPGA with the new network and run a test application. It will be possible to compare the results from the compiler with the FPGA ones.

2.2.2.2 Adding shared memory node

The current memory space of the *mNoC* is the one available on the local memory of each mMIPS node. In the *original mNoC* this space is reduced to 4 Kbytes, therefore the amount of data that fits in the system to be processed could not be enough for certain applications.

It is an objective of this Master Thesis to redesign the original configuration and research the procedure to add a new shared memory node in the system (*Figure 2.1*). Investigating which kind of memories is necessary, how much memory is needed and possible (because of area restrictions), and decides the communication protocol between memory nodes and processor ones through the network are important aspects to decide.

Changes in the SystemC description of the mNoC must be done attending to the previous study. After this updating, a *model* of the memory needs to be added to the simulator, as well as it is necessary to reflect on it all the changes made on the mMIPS processor core (exposed in section *2.2.1*). The user will be able to check the correct operation of the device before the physical implementation.

Last task to achieve this objective is uploading the FPGA with the new design and run one of the test applications available for the mNoC in order to check its functionality.

2.2.2.3 Adding I/O nodes

It could be possible that memory space might not be enough even having the new memory node due to physical limitations on the FPGA. More data space will be available adding I/O nodes to the mNoC, being able to have access to external memory where area constraints will not be a problem. Moreover, with this kind of node, external devices could be connected to the mNoC as a bigger data source.

For this reason, other objective of this project is researching the possibility of adding I/O nodes to the mNoC and describing which kind of messages are needed to communicate to external devices.

Just like the procedure done with the memory node, changes in the SystemC description of the platform should be performed, as well as updates in the simulator attending to those changes. For this reason, a *model* of the I/O node and also for the external memory needs to be added to the simulator.

Finally, uploading the FPGA with the new design will be the last step to achieve this objective. Results with simulator and FPGA should be evaluated.

2.2.3 CONVERT EXISTING JPEG DECODER TO A MJPEG DECODER

As it was exposed in the introduction of the chapter, the main objective of this project is to run some MJPEG decoders in parallel over an extended mNoC platform. The decoding of a Motion JPEG will be done decoding each one of its JPEG frames, so

the transition from a JPEG decoder to a MJPEG one should not entail too much problems.

With the *original mNoC* it could not be possible to store such amount of data just in the mMIPS local memory, but with the extensions that have been presented it should be. Processors will handle more data from the new shared memory node and from external data sources connected to the I/O node.

To achieve this purpose, a static mapping of tasks (they will be introduced in section *3.3.2*) on each processor node will be done; therefore a parallelized application will be achieved obtaining a pipelined task decoder.

Chapter 3

mMIPS Network on Chip (mNoC)

This chapter is mainly based on the information extracted from the *mNoC webpage* and it was the starting point of this project [6].

The Network on Chip (NoC) which this project is based on consists of four mMIPS processors [16] connected using a torus network with E-cube routing. The implementation of the NoC is in C++ using SystemC libraries, and all the example applications to test the functionality of the system have been written in C. The original sources of the NoC and example applications are for a 2-by-2 (4 nodes) variant of the NoC, but it is one of the objectives for this project to extend the platform to a 3-by-2 (6 nodes) one.

Figure 3.1 shows the elements that compose the mNoC:

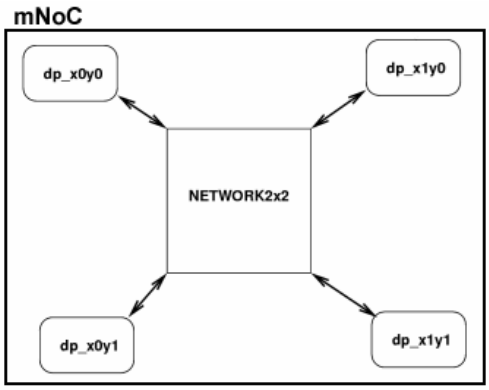

Figure 3.1: mNoC configuration

The list below gives a short description of components shown in the previous figure:

- *dp_xXyY*: The four *mMIPS processors* with their *Network Interfaces* are instantiated as dp_x0y0, dp_x1y0, dp_x0y1 and dp_x1y1 (dp stands for "data processor").

- *NETWORK2X2*: The module *NETWORK2x2* encapsulates the routers that interconnect the network interfaces of the *mMIPS processors*.

3.1 The mMIPS

3.1.1 INTRODUCTION

The mMIPS (mini MIPS) is a simplified version of the MIPS processor. It is a pipelined system performed in synthesizable SystemC code. Compared to the MIPS it has a reduced instruction set, which means that some operations need to be done in software and therefore more execution time is needed. It is also an objective for this project to extend this instruction set. The mMIPS processor without any extension will be called in this text *original mMIPS*. A block diagram is presented in *Figure 3.2*:

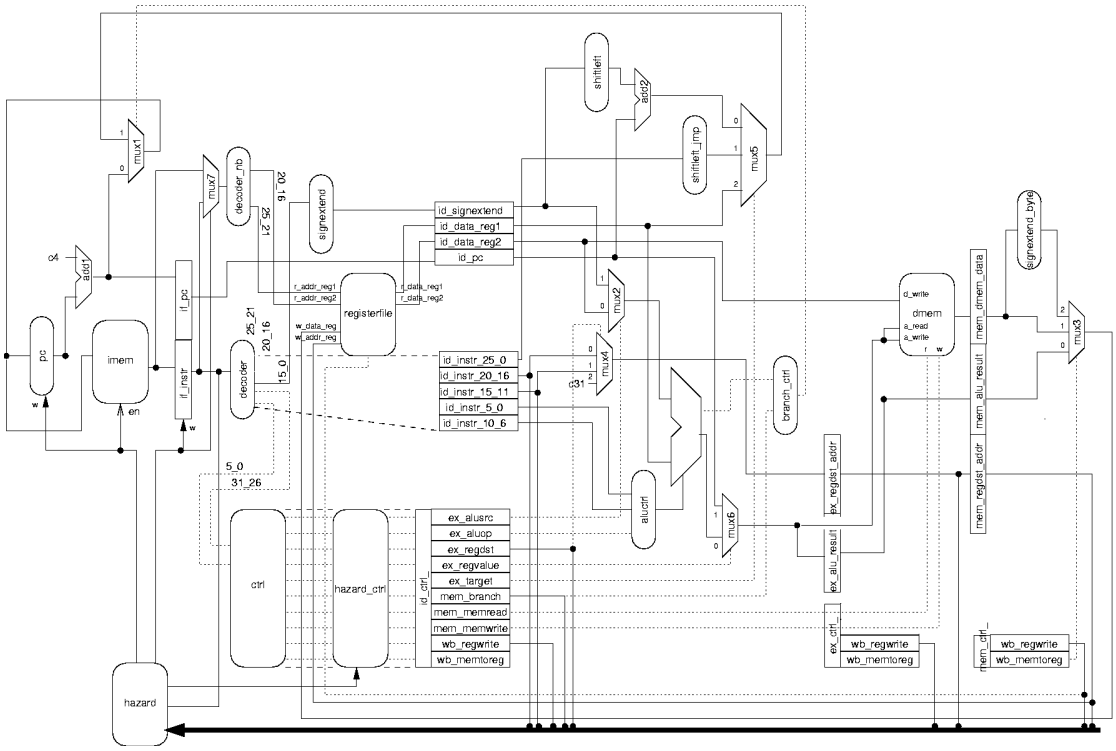

Figure 3.2: Oiginal mMIPS block diagram

These instructions are supported by the original mMIPS and the compiler (*lcc*) that is used in this project:

- addiu addu subu
- and andi or ori xor xori
- beq bne
- jal jalr jr j
- lb lw sb sw
- lui
- slti sltiu slt sltu
- sll sra srl (1, 2, 8 bits)

Below is a summary of the sizes in bytes of the standard data types in lcc and the operations that are performed in software due to the lack of a complete instruction set.

- The sizes of the standard C data types are 8 bits for char and 32 bits for the other types: short, int, long, float and double.
- Due to the limited instruction set the following operations are done in software:
  - All floating point operations
  - Multiply, divide, modulo
  - Variable distance shifts
  - Partial-word operations

3.1.2 MEMORY LAYOUT

The mMIPS implementation used in this project contains two separate memories of 16 kilobytes each. One of them is for instructions/code (ROM memory) and the other one for data structures (RAM memory). The compiler will not be able to use the whole memory space because the system is keeping 4 kilobytes from the RAM memory for debugging purposes and user data storage. The memory sizes have been hard coded in the SystemC sources of the mMIPS, in the compiler and in the test applications that run on them.

This memory mapping is shown in the figure below:

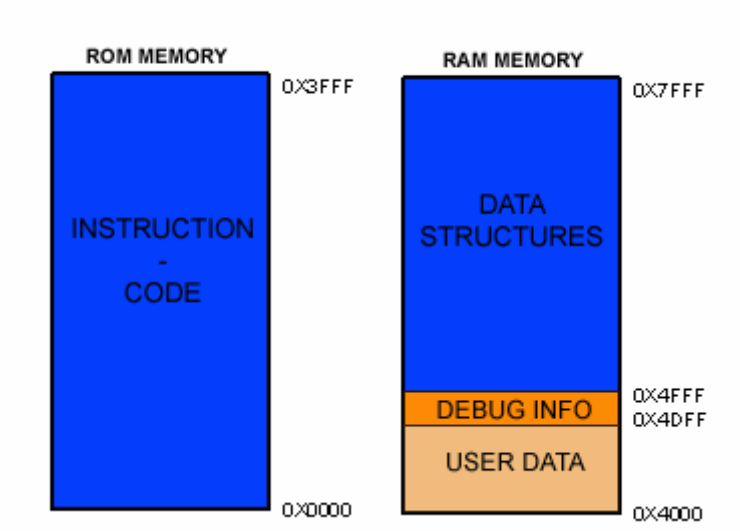

Figure 3.3: Memory layout

This memory structure tells the compiler that the 16 kilobytes up to 0x4000 should be used for instructions, that the 4 kilobytes from 0x4000 up to 0x5000 should not be used and that 12 kilobytes from 0x5000 through 0x7FFF should be used for data structures. The bytes from 0x4000 through 0x4DFF and 0x4E00 through 0x4FFF are used to store user data and debug information respectively.

The debug information is generated by the mMIPS when the function *mprintf()* is called. This function operates similar to the C function *printf()*, but in this case *mprintf()* outputs the resulting string to a memory space pre-defined (*DEBUG INFO* space in *Figure 3.3*). The user will be able in this way to follow the execution of the applications running on the mMIPS.

In the *USER DATA* memory space it could be stored any data necessary for the running application. In the *original mMIPS* this amount of memory is not enough for some applications tested in the *mNoC* (they will be introduced further on), and this is the reason why another objective of this project is solve this problem by adding new memory to the *mNoC*.

3.1.3 NETWORK INTERFACE

All the mMIPS communicates with other nodes through the network using the Network Interface (NI). Since this module is memory mapped it can be accessed through specific memory locations. The NI is controlled in mMIPS assembler by appropriate stores/loads to/from these memory locations. Another module, MEMDEV, replaces the data memory of the mMIPS and, based on the requested memory address either read/writes the RAM memory (for regular addresses) or performs appropriate communications with network interface (for device addresses).

This configuration is described in the following figure:

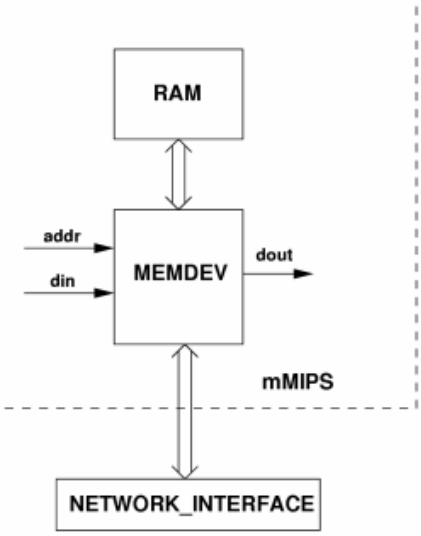

Figure 3.4: The MEMDEV module accesses RAM or NI depending on the address.

The way to communicate between two different nodes through the network is using two software functions: *sc_send()* and *sc_receive()*. These functions are used in the C code of the applications running on the mNoC and they are interpreted by the compiler giving the proper assembly code to the mMIPS to access the NI. *sc_send()* has three parameters: relative destination address, a pointer to the buffer where the information to be send is stored, and the number of bytes to be sent. In the case of *sc_receive()* only two parameters are necessary: a pointer to the buffer where the data should be stored and the size in bytes to receive.

The MEMDEV module recognizes two addresses assigned to the NI: 0x80000000 and 0x80000004. The first address (data word address) is associated with NI data while the second word (control word address) is used for NI control. Reading and writing from the data word results in the reading/writing from internal buffers of the

NI. The read/write operations into the data word are always non-blocking, which means that regardless the state of the NI they read/write the NI buffers. However, depending on the state of NI the read data may be invalid, or the written data can overwrite the packet in the send buffer, which was not sent yet. To avoid this kind of problems there are control signals in the NI which are asserted when the data is ready to read or when the buffer is free to write. To monitor the status of the NI, the control word of the NI can be accessed by the memory address 0x80000004. The meaning of the bits in the word is explained in the figure above:

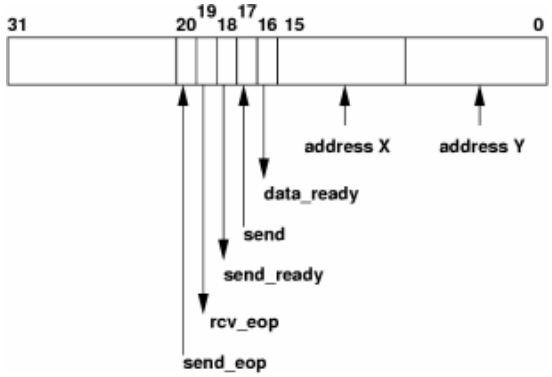

Figure 3.5: Meaning of the bits in the NI control word (device address: 0x80000004).

Attending to *Figure 3.5* and to the arrows on it note that some of the bits in the control word can only be written (0-15, 17 and 20 - they control the behaviour of NI) while some can only be read (16, 18 and 19 - they report the status of NI). Also note that bit from 21 to 31 are free and could be used for other purposes.

The status bits include:

- *data ready* (bit 16) - new data has arrived and is ready for reading. This bit will be automatically cleared after the data word has been read.
- *send ready* (bit 18) - previous data has been sent. NI is ready to send, data word can be safely written.
- *received end of packet* (bit 19) - multiword packets can be read from NI by reading consecutive 32-bit words. If this bit is active together with data ready, it means that the read is the last word of the packet.

The control bits:

- *address bits* (bits 0-15) - used to write the destination address of the packet (X distance - bits [15:8], Y distance - bits [7:0])
- *send bit* (bit 17) -writing 1 to send bit triggers sending the data previously written to the data word to the address present at the address bits of the control word.
- *send end of packet* (bit 20) - asserting bit 20 together with the send bit (17) means that the word written to data word is the last word of the packet and instructs NI to close the packet.

The NI is a module used to send and receive packets on the network. It is capable of sending and receiving packets with lengths being an arbitrary multiple of 32

bits. When such a packet is sent over the network it is split in smaller parts called *flits* before it is sent. On the other side, the arriving packet is reconstructed by collecting three flits. For any additional 32-bit word within the packet, additional two data-flits need to be added to the packet. The two actions of sending and receiving the packet are performed by two independent processes within the network interface, which means that NI is able to receive and send simultaneously. The interface of the module is shown in *Figure 3.6*.

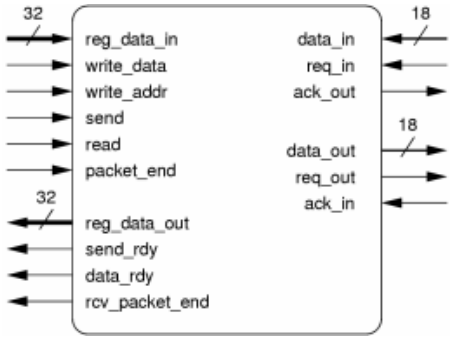

Figure 3.6: The Network Interface module

On the network side, the interface is compatible with the routers which compose the network: it has two sets of *data/req/ack* signals, one in each direction.

On the processor side, NI provides a set of signals necessary to write destination address and the packet word data (*reg_data_in* with *write_addr* and *write_data*), read received packet word (*reg_data_out*), trigger packet sending (*send*) and confirm packet reading (*read*) and the signals reporting communication status (*send_rdy* and *data_rdy*). In addition to that, two signals are used to mark last words of a given packet*: packet_end* asserted together with *send* means that a last word of the packet is being sent, while *rcv_packet_end* active together with *data_rdy* means that the last word of the packet has arrived.

The sending of a packet is shown in *Figure 3.7*:

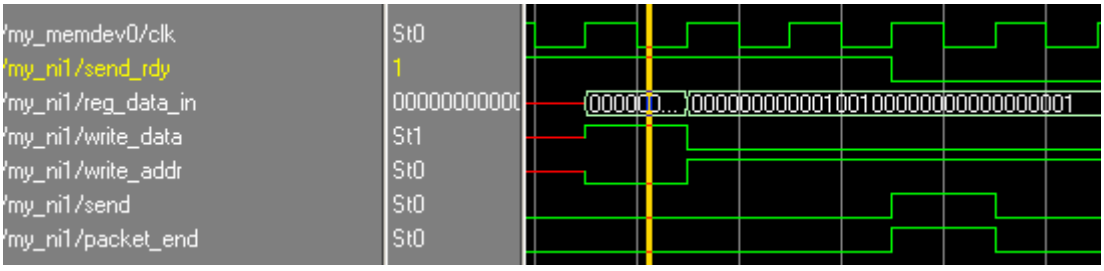

Figure 3.7: Sending of a packet

1. Wait for *send_rdy* to become high.
2. Output packet word on the *reg_data_in* bus and assert *write_data* signal.
3. Output destination address (relative) on the bottom 16 bits of the *reg_data_in* bus (X distance - bits [15:8], Y distance - bits [7:0]) and assert *write_addr* signal.

4. Assert *send* signal to trigger word sending, *send_rdy* will be deasserted. If this is the last word of the packet, assert *packet_end* as well.
5. If more packet words remain to send, wait for *send_rdy* to become high and goto step 2.

To read a received packet:

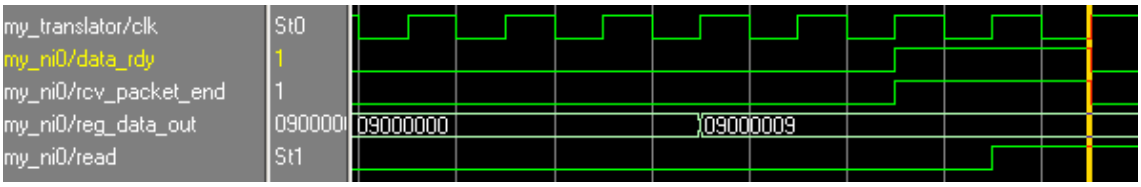

Figure 3.8: Receiving a packet

1. Wait for *data_rdy* signal to become high
2. Read data present at *reg_data_out*
3. Assert *read* signal to free the buffer and allow receiving of new packet words. If *rcv_packet_end* is high, this is the last word of the packet.

3.2 NETWORK2X2

The network that connects the mMIPS processors on the NOC is a *torus network* with *E-cube* routing [7]. The network used in the *original mMIPS* is two nodes wide and two nodes high.

In the *torus* topology, a network node is connected to its immediate neighbours in both dimensions. At edges of the network, the connections wrap around and connect the last router in the given dimension with the first.

In the E-cube routing each packet in the network is first routed along the X dimension, until it reaches a router with the X address equal to the packet's destination X address. Then, it starts to move in the Y dimension until it reaches the destination router. Since connections in the network are unidirectional, the packets can only travel in the direction of increasing addresses, if necessary wrapping at the edge of the network.

To implement deadlock-free communication, two virtual channels, numbered 0 and 1, share each physical link. Each packet sent in the network travels on the channel 0 until it reaches destination or wraps-around. In the later case, it moves to channel 1 and continues on this channel. This switch breaks the circular dependencies within the network and therefore prevents deadlocks.

The physical links are realized as 18-bit wide busses (16 data bits + 2 flit-type-bits). Each link is accompanied by 2 sets of request/acknowledge signals, each of which implements a single virtual channel (*Figure 3.9*).

The mMIPS is connected to a router by a bidirectional link realized as two unidirectional 18-bit wide data lines with associated req/ack lines.

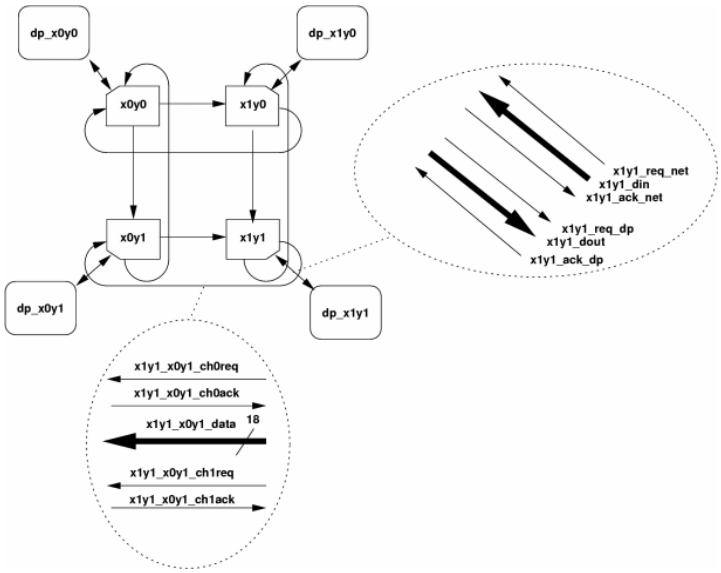

Figure 3.9: Four routers (xYyY), four mMIPS (dp_xXyY) and the data lines that connect them.

The communication in the network is based on a synchronous request/acknowledge protocol. To send data, sender outputs the data and asserts the request signal. The data is stored by the receiver at the rising clock edge and acknowledged by the asserted acknowledge signal. Then, the sender withdraws the request signal and the receiver withdraws the acknowledge signal.

3.2.1 THE ROUTER

The symmetry of both dimensions permits realization of a single router as a composition of two identical 1-dimensional sub-routers (see *Figure 3.10*). The X sub-router receives data from the NI of mMIPS on input data bus, *din*. This data is forwarded to the x output, which is connected to the X sub-router of the neighbouring router. The data travels through X sub-routers until it reaches the destination "column". Then, it is forwarded to the d output of X sub-router, which is connected to the d input of Y sub-router. In the process, the X address is stripped off the packet and replaced with the Y address. Further, the packet travels to the destination along the Y dimension. When it reaches the destination address, it is again forwarded to the data output bus, *dout*, which in the case of Y sub-router is connected to the input of the data processor.

In the mMIPS relative addressing is used. A mMIPS node provides initial packet with the address containing X and Y distance to the destination (including possible wrap-around). During the journey along one dimension, the first address component is decremented upon leaving a router. When it reaches 0, it is replaced by the address in the second dimension and forwarded to the d output of sub-router (which may be connected to Y router or mMIPS).

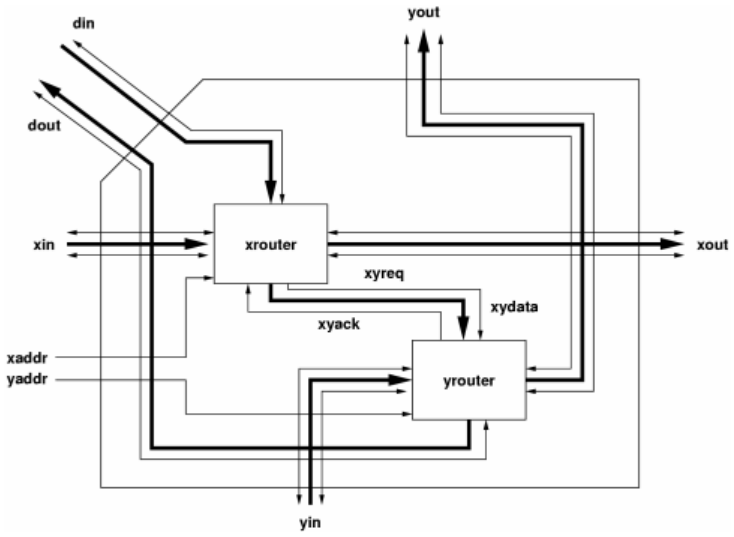

Figure 3.10: Router with the two sub-routers

As mentioned earlier, the links in the network are 16-bit wide (actually 18-bit, but the top 2 bits are used for control purposes). To transfer larger packets of data, packets need to be split into several 16-bit wide flits. The network allows arbitrary length packets. First flit of the packet is marked as *header* on the top 2 bits of the 18-bit wide data link (these are the control bits). The 16-bit data of the header flit contains two 8-bit destination addresses (for X and Y dimensions). Upon receiving the header flit and based on the destination address, a router sets up the connection between the input link and an appropriate output link (x output on channel 0, x output on channel 1 or d output). An arbitrary number of the following flits are forwarded along the same route. The packet is closed by a flit marked as *trailer*. The data in the trailer flit is also forwarded along the route and the route is closed. This method of routing is called *wormhole routing* [13]. *Figure 3.11* illustrates this for a packet with 32-bits of data (0x12345678) sent from address (1,1) to (0,0) (relative address 1,1). This package is split in three flits: a *header* flit with the relative destination address, a *data* flit with 16 bits of data and a data flit marked as *trailer* with the remaining 16 bits of data.

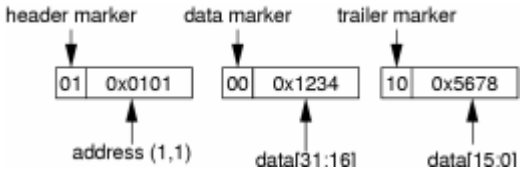

Figure 3.11: A 32-bit packet sent from address (1,1) to (0,0) is split in three flits: relative address (0x0101) and data (0x1234 and 0x5678)

3.2.2 SUB-ROUTER

A block diagram of a single sub-router is presented in the following picture:

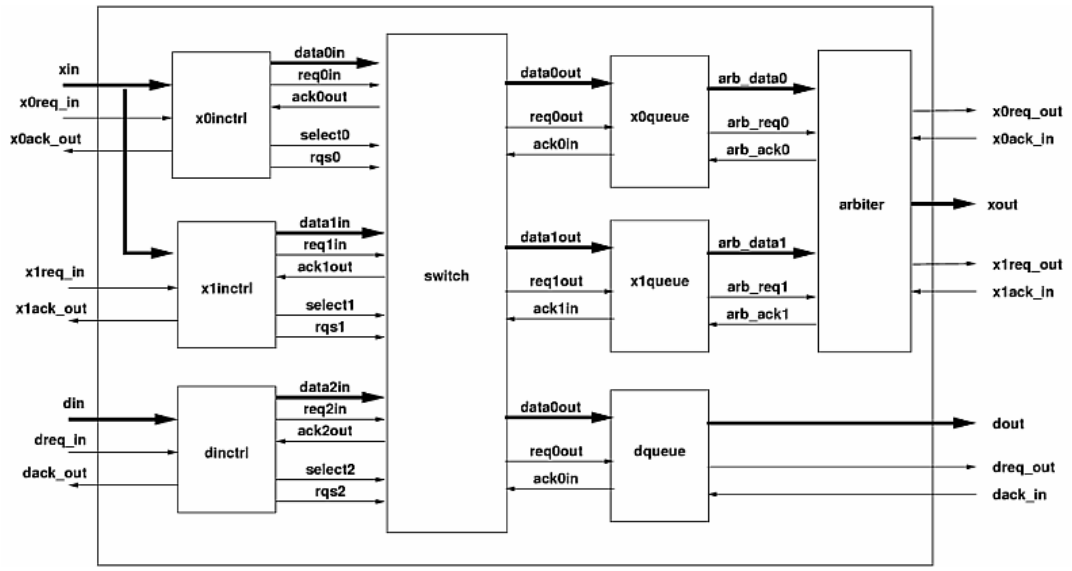

Figure 3.12: Sub-router

The sub-router includes three identical input controllers: one for the data input to the router and one for each virtual input channel to the router (note how *x0inctrl* and *x1inctrl* share single physical data input link, but have separate control inputs for separate virtual channels). Upon detecting active request line, input controller registers and examines the input data. If it is a *packet header*, input controller requests from the switch a route to an appropriate output queue (signals *select* and *rqs*).

The requested output queue depends on the destination address of the packet. The switch arbitrates the route requests and connects winner's *data*, *req* and *ack* signals to the corresponding signals of the requested output queue. If the output queue (in the *original mMIPS* it is just 1 element) is not full, it stores the data and issues *ack* signal to the input controller, which forwards it to the data source. The following flits delivered to the input controller are forwarded along the route established through the switch until the *trailer flit* is detected. After the *trailer flit* has been acknowledged by the output queue, input controller withdraws *rqs* signal, thereby freeing the route through the switch.

The output queue, upon receiving from the input controller and acknowledging a flit, forwards it to its output. Once the flit has been acknowledged, the queue clears buffer and is ready to accept new flits.

Note that while the *d* output queue is connected directly to the physical output channel, the two *x* output queues associated with virtual channels 0 and 1 need to compete for the access to the physical output link. It is necessary to introduce an arbiter to decide which virtual channel will get this access.

3.2.3 OUTPUT ARBITER

The arbitration is performed by the *Output Arbiter* module (*Figure 3.13*). Whenever a virtual channel controller needs access to the link, it asserts a request signal to the arbiter (*arb_req*) and waits for *grant* signal. The request signal 0 has a higher priority than 1, but after each transmission (1 clock cycle) the requesting channel

controller has to remove the request signal, allowing the other waiting channel to access the link. This way, the arbiter is giving priority to channel 0, but channel 1 will not be starved.

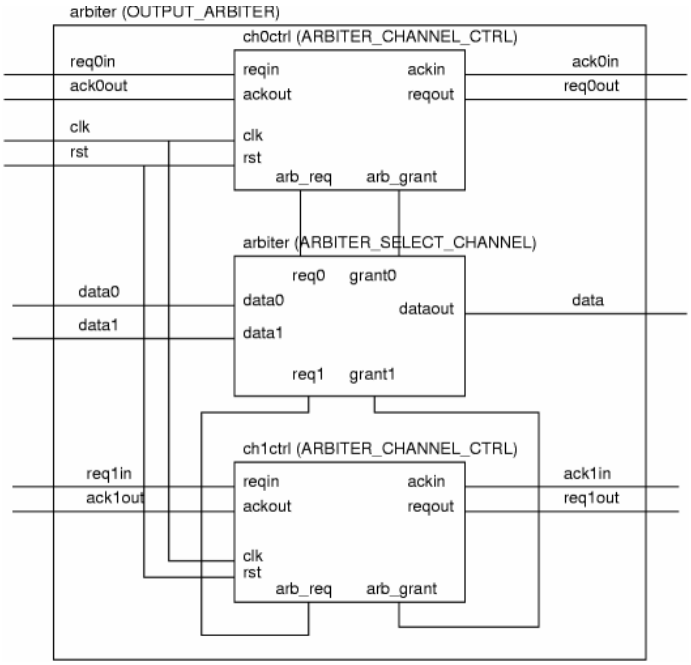

Figure 3.13: Output Arbiter

3.3 Applications

There are two test applications which are being used to test the functionality of the original mMIPS. The first one, Gossip, is a very simple program that sends a short message across the network. The second one, *JPEG decoder*, takes a JPEG image and converts it to an (uncompressed) bitmap using a decoding method called *the baseline decoding process*.

3.3.1 TEST APPLICATION GOSSIP

Gossip is an application to test if the whole system has been set up correctly, simply sending a short message across the network (see *Figure 3.14*).

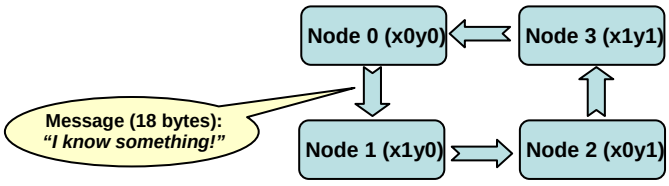

Figure 3.14: Gossip for 2x2 network

Node x0y0 send the text "*I know something!*" to node x1y0 and then listens for a return message from any node. When it has received that message, it quits. All other nodes wait for an incoming message, forward it to the next node and then exit.

3.3.2 MULTI-PROCESSOR JPEG DECODER

The JPEG decoder takes a JPEG image and converts it to a (uncompressed) bitmap using a decoding method called *the baseline decoding process* [8]. The multi-processor decoder is based on a single-processor decoder, and its functionality is presented in *Figure 3.15* that shows the process steps (blue) of the JPEG decoding process:

- VLD - Variable length decoding,
- ZZ - Zigzag scan,
- DQ - Dequantization,
- IDCT - Inverse discrete cosine transform,
- Color conversion (YUV to RGB) and reorder.

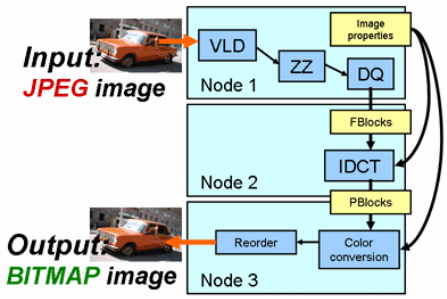

Figure 3.15: JPEG decoding process

The yellow blocks from the previous figure describe the data that is sent between nodes (mMIPS processors).

The JPEG decoder process has been divided up into three nodes as depicted in *Figure 3.15*. Each step runs on a separate node of the mMIPS network: node1 is at (X,Y) = (0,0), node2 is at (1,0), node3 is at (0,1) and the node (1,1) is not used. This partition was chosen because a quick investigation of the sources of the original JPEG decoder revealed that partitioning just before and after IDCT-function was the easiest to realize. This choice also has the advantage that the Huffman decoding and dequantization tables required by the VLD and DQ units respectively do not need to be sent over the network. This partition leads to the distribution of the workload as shown in *Table 3.1*.

Node	Steps + workload %	Total workload %
1	VLD (35%) + ZZ (5%) + DQ (10%)	50%
2	IDCT (20%)	20%
3	Color conversion (15%) + Reorder (15%)	30%

Table 3.1: Workload for nodes 1, 2 and 3 in the JPEG decoder.

This application has been tested with a 32x24 pixels color JPEG image, and the decoding on a *2x2 mNOC* using the hardware simulator took 29 hours on a Pentium III 1GHz processor running GNU/Linux 2.4.20 with 2048 MB of RAM (co2.ics.ele.tue.nl). Because of this long simulation time, another objective of this project is developing the mMIPS processor adding new Hardware instructions in order to decrease this time.

Chapter 4

Extensions on mMIPS and NETWORK

4.1 Introduction

Trying to achieve the main objective proposed for this Master Thesis, some changes in the system presented in *Chapter 3* have been done. Intermediate objectives have already been explained in *Chapter 2*; which are prior steps to achieve the purpose of this project.

This chapter introduces the changes that are made regarding the mMIPS and the NETWORK. A description of changes performed on the system as well as results and evaluations of them are part of the following sections.

Implementation of the new shared memory node and the conversion of a JPEG decoder to a MJPEG decoder will be presented in *Chapter 5* and *Chapter 6* respectively.

4.2 Extended mMIPS

4.2.1 IMPLEMENTATION

As it was explained in section *3.1*, the *mMIPS* (mini MIPS) processor is a simple MIPS version with a reduce instruction set. Multiply function is one of the instructions that are not among this set, and for this reason some applications could have restrictions running on the mNoC.

Due to the lack of this function all multiplies are done in software, which means that a lot of function calls are executed. This fact implies large instruction code size and long run time.

An extension in the mMIPS has been performed in order to solve this problem. The *new mMIPS* has a multiply function among its instruction set, thus multiplies are now done in hardware avoiding software function calls on this way. With this extension an increment of the mNoC speed-up is expected.

It is important to mention that the multiply instruction added to the mMIPS set is not the one that is implemented in a regular MIPS, but in the case of the *new mMIPS* is a reduced version of the one in a MIPS. It is an operation that handles two 32-bits

numbers obtaining another 32-bits number. It does not take issues such as overflow into account. The reason why a simple multiplication has been chosen is that for the applications used in the mNoC (see *3.3*) more complex operations are not needed.

Adding a new operation to the mMIPS implies changes in the compiler (*lcc* in this case) and also in the SystemC definition of the mMIPS [19]. The following figures try to explain the reason of changes in both sides.

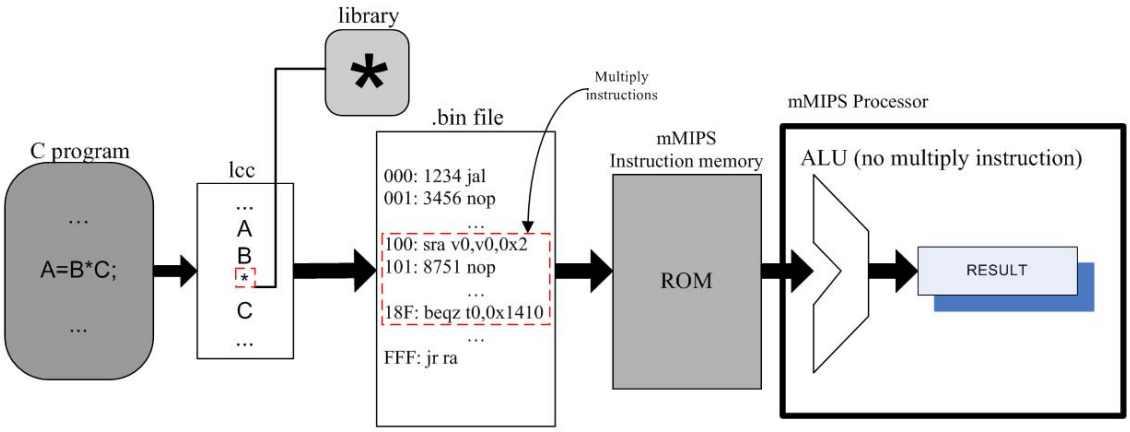

Figure 4.1: mMIPS without multiply instruction

When an application wants to be run in the mNoC, a C program must to be compiled with *lcc* obtaining a binary file which contains the assembly code that should be uploaded into the instruction memory of the mMIPS. If the *original compiler* finds a multiplication in the code ('*' symbol) it will not translate it to a single assembly instruction, but to multiple calls that means several assembly instructions for the processor. In *Figure 4.1* the compiler is performing a multiplication with several simple arithmetic instructions known by the processor, so the mMIPS will take more time executing it.

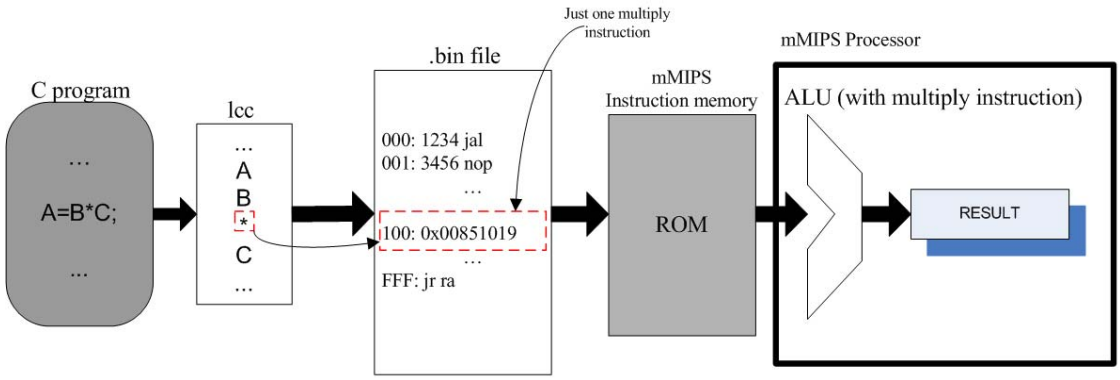

Figure 4.2: mMIPS with multiply instruction

Different is the case of a mNoC provided with a multiply instruction, which is the case of the *new mNoC* created for this project. The first change to be done is in the compiler. It should now recognise the multiplication symbol and assign a specific assembly instruction to it. On this way, the compiler is avoiding library calls that mean

extra instruction code. Once the compiler has assigned a single instruction to a multiplication, the ALU inside the processor should interpret it and perform the proper arithmetic operation. Therefore, the second change is in the processor. A new instruction, multiply, has been implemented in hardware on it; being able to spend less time calculating the multiplication result.

4.2.2 RESULTS

In order to measure the improvements when a multiply instruction has been added to the mMIPS, some applications were run on the simulator. Those programs have been simulated in both mNoC with and without multiply instruction and then compared one with each other obtaining the mMIPS speed-up on this way.

The first application used to compare original and new mNoC is a loop where a thousand of multiplications are done. As said before, with this simple program the compiler has to assign an assembly instruction to the multiply symbol, saving several call instructions.

The second application is the *JPEG decoder application,* already explained in section *3.3.2*. Here is where one of the objectives of this project could be checked: the improvements in the mNoC simulation time.

Table 4.1 shows the results of these simulations:

Application	Simulation time without multiply instruction (min)	Simulation time with multiply instruction (min)	Improvement (%)
1000 multiplications	39.62	35.53	**10.3**
JPEG decoder	1727.17	1628.76	**5.7**

Table 4.1: Comparative simulation time table between original and new mNoC

As it could be seen in the previous table, the speed-up improvement when a new multiply instruction is added to the mNoC is not that big as it was expected. Even forcing the compiler to use the new instruction, a 10% of speed-up has been achieved. Even though a 5.7% on the JPEG decoder means saving almost 2 hours of simulation time, a bigger percentage was expected.

Attending to these results, it seems necessary to research another solution for the low simulation speed problem. Implementing in hardware other instructions that are executed in software doing library calls, like it was done with the multiply function, could be one of these solutions. Even with these new instructions, a big speed-up improvement can not be expected. The main problem is the simulator itself, which abstraction level is too low. Therefore, raising the abstraction level of the simulator from cycle-accurate RTL level simulation to non-synthesizable cycle-accurate or instruction set simulation could yield better results.

Changing simulator abstraction level could take a lot of time and it is out of the scope of this project (it could be a project itself). For this reason, this solution will be proposed for future improvements of the mNoC.

4.3 Extended NETWORK

4.3.1 IMPLEMENTATION

The network used in the mNoC is composed for four nodes with one mMIPS processor on each connected using a *torus* network with *E-cube routing*. In order to achieve the main objective for this project, it was necessary to add two *extra* nodes allowing the processors to handle more data in this way. For this reason, the network has been extended from 4 nodes to 6 to support these extra nodes, which means a transition from a NETWORK2x2 to either NETWORK3x2 or NETWORK2x3. The first option was the one chosen.

Further on, these two nodes will be hosting a memory node and an I/O node respectively, but the network functionality should be tested first. In order to do it, the new two nodes have been filled with mMIPS processors as shown in *Figure 4.3*:

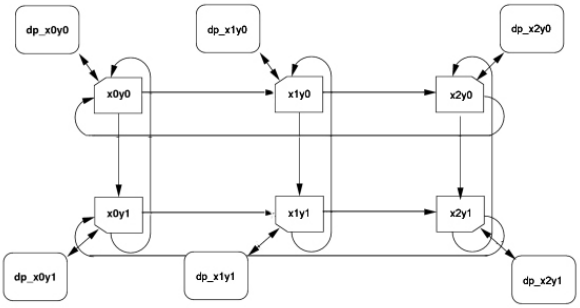

Figure 4.3: mNoC with NETWORK3x2

To be able to achieve the diagram shown in the previous figure, the first change was made in the SystemC description of the network. Two new routers were added to it (*x2y0* and *x2y1*) obtaining a NETWORK3x2 on this way.

The second change was performed in the SystemC description of the mNoC. It was in this point where the two *extra* nodes were added to the system (*dp_x2y0* and *dp_x2y1*). For the time being these two nodes contain a mMIPS processor each.

Last change concerns with the simulator, which was also extended. All extensions made in the SystemC description of the network have been reflected on the simulation in order to test its functionality before hardware implementation. Also new trace signals have been added to the simulator to follow the network behaviour after executing an application.

4.3.2 RESULTS

It was necessary to run an application on the simulator in order to test the new network. *Gossip application* (see *3.3.1*) has been extended to handle 6 nodes instead of 4, although the main purpose of the new application is the same as the original one: send a message along the whole network.

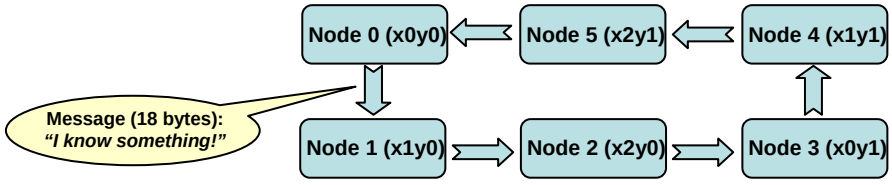

Figure 4.4: Gossip for 3x2 network

A message is sent from *Node 0* to *Node 1*, and then this one will forward it to the next one. All remaining nodes will do the same procedure until the message reaches *Node 0* again, where the application finishes. With this application, communication between nodes through the network could be tested.

After executing this e*xtended Gossip*, the content of the RAM memory should be examined in order to check whether network functionality is correct or not. The *Debug Info* area of this memory (see *3.1.2*) contains information about the application execution. In the case of a NETWORK3x2, the result was the following:

```
NODE x0y0:                      NODE x1y0:                      NODE x2y0:
=========                       =========                       =========
Node 0 up and running!          Node 1 up and running!          Node 2 up and running!
I sent: "I know something!“     I received: "I know something!“ I received:"I know something!“
I received:"I know something!“  Let's call it a day!           Let's call it a day!
Let's call it a day!            Successfully fw the message.    Successfully fw the message.

NODE x0y1:                      NODE x1y1:                      NODE x2y1:
=========                       =========                       =========
Node 3 up and running!          Node 4 up and running!          Node 5 up and running!
I received: "I know something!”  I received: "I know something!“ I received:"I know something!“
Let's call it a day!            Let's call it a day!            Let's call it a day!
Successfully fw the message.    Successfully fw the message.     Successfully fw the message.
```

Figure 4.5: Extended Gossip debug info with a 3x2 network

Looking at this result where the message is successfully forwarded around the whole network, it could be said that the NETWORK3x2 works and is ready to handle communications between 6 nodes.

Therefore, another objective has been achieved performing the transition from a NETWORK2x2 to a NETWORK3x2. The next step then is replacing the two mMIPSs located in the *extra* nodes for a memory node and an I/O node.

Chapter 5

Shared memory node

5.1 Introduction

It was already mentioned that the memory space in the mNoC is the one available on the local memory of each mMIPS node. In the *original mNoC* this space could not be enough when certain applications are run on it.

Trying to avoid this memory constraint, the system should be changed in order to be able to handle more data than the one located on the mMIPS local memory. For this reason a remote shared memory node has been added to the mNoC.

Once the network has been extended from 4 to 6 nodes, one of the two new nodes was used to host the remote shared memory as it could be observed in *Figure 5.1*:

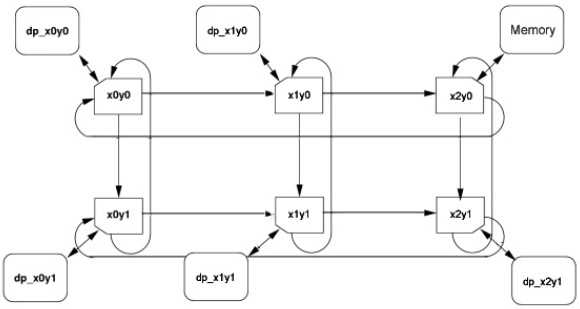

Figure 5.1: mNoC with remote memory node using a 3x2 network

5.2 Choosing memory implementation

The first aspect under study when a new memory wants to be implemented is which type of memory is needed and possible. Attending to particular FPGA features, different ways to implement a memory could be performed. The FPGA used in this project is a *Xilinx Virtex-II XC2V3000*, and looking to its data sheet [9][10] some features should be considered in order to choose the optimum implementation.

In *Virtex-II* FPGAs there are two ways to implement memories:

- Configurable Logic Blocks (CLBs)

They are used to build combinatorial and synchronous logic designs; which are 4-input look-up tables (LUT), 16-bit variable-tap shift register element, or 16-bits of distributed SelectRAM memory.

Distributed SelectRAM memory modules are synchronous (write) resources. The combinatorial read access time is extremely fast, while the synchronous write simplifies high-speed designs.

- Block SelectRAM memory

  The block SelectRAM memory resources are 18 Kb of dual-port RAM programmable from 16K x 1 bit to 512 x 36 bits in various depth and width configurations. Each port is totally synchronous and independent. Block SelectRAM memory is cascadable to implement large embedded storage blocks.

The availability of these resources on the FPGA may vary depending on the family. In the case of *XC2V3000*, these resources are shown in the next table:

Device	**System Gates**	**CLB (1 CLB = 4 slices = Max 128 bits)**			**SelectRAM Blocks**	
		Array Row x Col.	**Slices**	**Maximum Distributed RAM Kbits**	**18 kbit Blocks**	**Max RAM (Kbits)**
XC2V3000	3M	64 x 56	14336	448	96	1728

Table 5.1: CLB and SelectRAM avalibility in a Virtex-II XC2V3000

There are other aspects that should be considered, like how much memory is needed or how much memory is possible to implement. In the case of *JPEG decoder application*, the memory space required depends on the amount and size of images to be decoded; and the amount of memory that could be implemented depends on the resources available in every node of the mNoC.

In particular case of mNoC, each node is using 18 SelectRAM Blocks; where 8 of them are for RAM memory, another 8 for ROM memory and finally 2 of them for registers. In a network with 4 nodes 72 SelectRAM Blocks are used. Considering that enlarging the network there will be 6 nodes, 108 SelectRAM Blocks are needed. Here was where a restriction on the FPGA was detected; therefore just a NETWORK2x2 fits in the available FPGA.

Because of this reason the mNoC could be enlarged just in the simulator, but only 4 nodes could be used in the FPGA. Knowing this, one of the nodes of NETWORK2x2 will be the memory node. Since there will be only RAM memory on it, just 10 SelectRAM Blocks are used on the memory node; therefore 32 more blocks are free to be used to build RAM memory.

Doing this calculation there are 40 (8+32) SelectRAM Blocks available to build a RAM memory in the memory node. Since each block can store 16kbits, the amount of memory that could be implemented in the memory node is 80 Kbytes.

Considering all these aspects, the memory chosen to be implemented as a shared remote memory was a *16Kbytes dual-port RAM* because of the following reasons:

- There was RAM memory code available in the SystemC description of the mNoC which could be reused. This memory is the one used for the mMIPS RAM memory.
- This RAM is based on Virtex dual-port SelectRAM blocks, therefore it seems the most suitable memory design to be used in the remote shared memory node.

Another aspect to be considered is that *dual-port* functionality of this memory is not used to perform two memory accesses at the same time. In this case, the mNoC configuration uses the second port to connect memories with outside world, and being able to upload them when running applications on a FPGA.

As it was explained before 80 Kbytes could be used, but just a 16 Kbytes memory was implemented because of code simplicity (reusing code). Nevertheless up to 80 Kbytes could be used if it is necessary.

5.2.1 16 Kbytes dual-port RAM

The implementation of the *16 Kbytes dual-port RAM* memory is based on Virtex II Block SelectRAM memories. Specifically, the ones used for this memory are *RAMB16_Sm_Sn*; which are dual-ported memory blocks with synchronous write capability. Each block SelectRAM port has 16384 bits of data memory with port width (*m* or *n*) configured to 1, 2, 4, 9, 18 or 36 bits. Each port is independent of the other accessing the same set of 16384 data memory cells.

Table 5.2 shows architectures supported with these Blocks SelectRAM [11], where the FPGA used in this project (Virtex-II) is among of them:

RAMB16_Sm_Sn	
Spartan II, Spartan IIE	No
Spartan-3	Primitive
Virtex, Virtex-E	No
Virtex-II, Virtex-II Pro, Virtex-II Pro X	Primitive
XC9500, XC9500XV, XC9500XL	No
CoolRunner XPLA3	No
CoolRunner-II	No

Table 5.2: Architectures Supported with RAMB16_Sm_Sn

In the case of *16 Kbytes dual-port RAM* the Blocks SelectRAM were configured with *m=n=4*, and the features of this configuration is shown in the following table:

Component	**Port A**			**Port B**		
	Data cells (Depth *x* Width)	**Address Bus**	**Data Bus**	**Data cells** (Depth *x* Width)	**Address Bus**	**Data Bus**
RAMB16_S4_S4	4096x4	(11:0)	(3:0)	4096x4	(11:0)	(3:0)

Table 5.3: RAMB16_Sm_Sn features with m=n=4

The aspect of this block is presented in *Figure 5.2*:

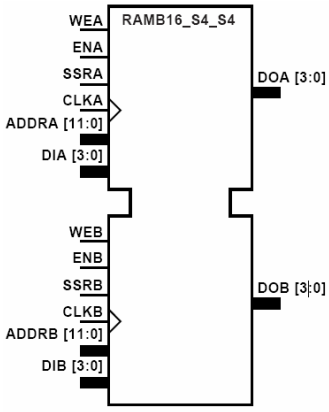

Figure 5.2: RAMB16_S4_S4 Block SelectRAM

Table 5.4 shows the truth table of *RAMB16_S4_S4* port A. The truth table for port B is the same.

INPUTS							OUTPUTS	
GSR	**ENA**	**SSR A**	**WE A**	**CLK A**	**ADDRA**	**DIA**	**DOA**	**RAM Contents**
1	X	X	X	X	X	X	INIT_A	No Chg
0	0	X	X	X	X	X	No Chg	No Chg
0	1	1	0	↑	X	X	SRVAL_A	No Chg
0	1	1	1	↑	addr	data	SRVAL_A	RAM(addr) => data
0	1	0	0	↑	addr	X	RAM(addr)	No Chg
0	1	0	1	↑	addr	data	No Chg[1] RAM(addr)[2] Data[3]	RAM(addr) => data

Table 5.4: Truth table of RAMB16_S4_S4 ports A and B

- GSR=Global Set Reset
- INIT_A=Value specified by the INIT_A attribute for output register. Default is all zeros.
- SRVAL_A=register value
- addr=RAM address
- RAM(addr)=RAM contents at address ADDR
- data=RAM input data

Depending on the value of WRITE_MODE_A (attribute of the memory):

- [1]WRITE_MODE_A=NO_CHANGE
- [2]WRITE_MODE_A=READ_FIRST
- [3]WRITE_MODE_A=WRITE_FIRST (default)

16 Kbytes dual-port RAM is composed of 8 *RAMB16_S4_S4*. On this way, it is able to store 4096*x*4*x*8 bits = 131072 bits. Since 1 byte = 8 bits, *16 Kbytes dual-port RAM* can store 16384 bytes = 16Kbytes on each port.

This RAM memory uses both ports for different purposes. Port A is used for regular memory transactions, while Port B is used for debugging purposes (upload memory before executing applications on a FPGA and read back memory contents after

the execution). All 8 Blocks are administered by two *wrappers*, one for each port. *Figure 5.3* shows a simple diagram of a *16 Kbytes dual-port RAM*:

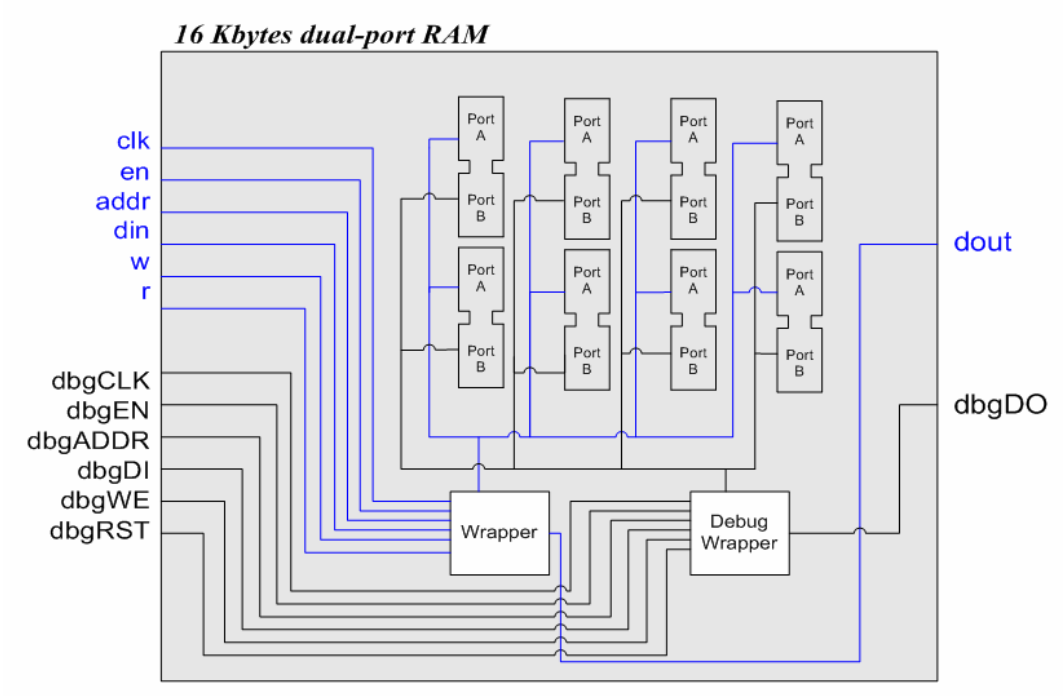

Figure 5.3: 16 Kbytes dual-port RAM simple diagram

5.3 Changes on the mMIPS

It has been already explained that the memory space in the *original mNoC* is reduced to the one available in the local RAM memory of the mMIPS. In order to handle more data, changes on the SystemC description of the mMIPS should be preformed to use the new remote shared memory node.

Having a look to section *3.1.2* it could be observed that 512 bytes from RAM memory are reserved for debugging purposes and 3.5 Kbytes are used for user data storage. The first idea was relocate this data to the remote memory, obtaining on this way a 16K RAM memory only for mMIPS data structures. Data used by the compiler (located in local RAM) and debug and user data (located in remote RAM) could be separated if this change was performed. *Figure 5.4* tries to explain this idea:

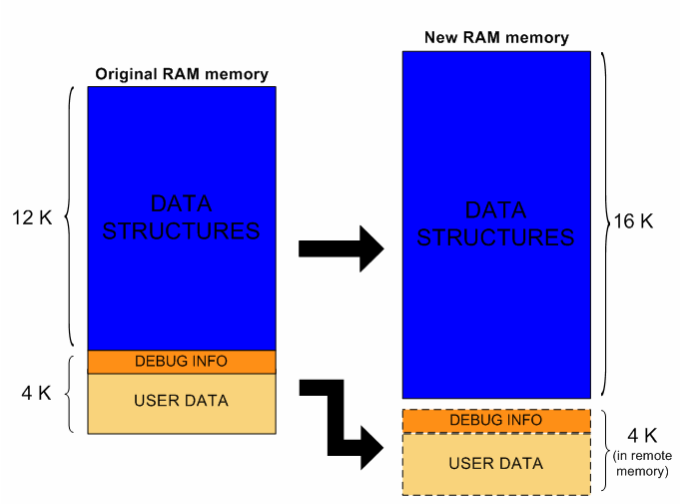

Figure 5.4: Relocation of memory in the mMIPS

Even though it could be a good implementation proposal, two main reasons made discard it:

- Storing debug information in remote memory introduces more traffic in the network, having slower communication. This data will not be shared with other nodes because it is only useful to follow the application behaviour after an execution.
- It could be useful having some memory space in local memory for certain applications (for example *Gossip application*). Access to this data will be faster than the one done in the remote memory.

Therefore the RAM memory used by the mMIPS did not undergo any change.

5.4 Proposals for communication with memory node

Next question to answer was how the mMIPS nodes will communicate with the remote memory node when an access to the shared memory is required. It was necessary to find a way of communication between those nodes that converts the regular *load* and *store* used when accessing to local RAM into *sc_send()* and *sc_receive()* through the network to the remote memory node.

Two possibilities were considered in order to find an answer to this question: either changing the compiler or introducing a new module called TRANSLATOR.

5.4.1 SOFTWARE SOLUTION: CHANGING THE COMPILER

In section *3.1.3* it was explained how the communication between mMIPS nodes works. MEMDEV (see *Figure 3.4*) module is the one that make use of the network depending on the incoming address. This proposal uses this functionality.

If the compiler is able to perform a *sc_send()* or *sc_receive()* when *load/store* from/to certain addresses in a C program are detected, there is no need to change

mMIPS nodes. Since MEMDEV will receive control and data words created by the compiler in the proper addresses, a regular communication between nodes will be performed. The compiler should have a new memory map in order to detect which addresses belong to the local RAM memory and which ones do to the remote one.

The main advantage of this solution is that no Hardware changes in the mMIPS nodes have to be done, only in the compiler. This means that this is a pure Software proposal.

But there are some disadvantages which do not make suitable this solution:

- Make changes in the compiler like the one necessary for this solution could be a very complex task.
- This is a software solution. The compiler will generate extra code, which means more assembly instructions for the mMIPS and therefore long execution time.
- What happen in the remote memory node? There should be something that manages data transactions in the remote memory. Just with *sc_send()* and *sc_receive()* is not enough to handle a communication with the remote memory. Therefore, changes in the MEMDEV located in the remote memory are necessary to solve this problem.

5.4.2 ADDING A NEW HARDWARE MODULE: TRANSLATOR

Another possibility that could avoid software solutions and is able to manage data transactions in the remote memory was proposed.

In this case it is not necessary to perform changes in the compiler or in the mMIPS, because a new module called TRANSLATOR is introduced into the system. In the mMIPS side this module detects the addresses that belong to the remote memory and it is only in this case when it starts to work. It encapsulates all necessary information in flits to be sent as a request to the remote memory. It also manages the communication with the network creating proper control signals and sending the flits already assembled. Therefore, TRANSLATOR module is the one that translates regular load and store to *sc_send()* and *sc_receive()* through the network. In the memory node side this module receives requests from other nodes and handles the access to the remote memory. Remote memory node should be able to send data back to mMIPS nodes, for this reason in the memory side TRANSLATOR also manages the communication with the network.

This proposal has some advantages:

- This is a Hardware solution. No extra assembly code will be added because generation of new control and data words will be performed in Hardware, which means faster execution time.
- It is not necessary to perform big changes in the original system because TRANSLATOR module will manage all the communication with the remote memory. Compiler and mMIPS will have their original functionality.

- With TRANSLATOR module the memory node will be able to manage accesses to the shared memory and also send data back to the mMIPSs.

The main disadvantage is that this one could be a more difficult solution because of the necessity of new Hardware and new communication protocol with the remote memory.

Since no changes are performed in the compiler, this solution could introduce a problem: *memory consistency*. Nodes may access to the same shared memory address at the same time if the compiler does not know that it can not access certain address ranges. Due to this potential problem, in section *5.8* both the problem and a solution are introduced.

Considering advantages and disadvantages of both proposals, the second one seemed the most suitable to implement. TRANSLATOR module is *transparent* to the system; no changes on it need to be performed. This means that introducing the new module into the system the only change is the location of the data (local or remote memory), because the rest of the system will have the original functionality.

5.5 TRANSLATOR: New communication with the shared memory node

As said before both nodes the mMIPS and memory ones need a TRANSLATOR module, although necessities are different on each node. In mMIPS nodes it is necessary to detect an address range that belongs to remote memory, create new control and data words and pass them to NI. Finally NI will split those words into flits and it will send them to the remote memory node. In memory node it is necessary to receive those flits, perform access to the shared memory and also send data back to mMIPS nodes. For this reason it is possible to distinguish two kinds of TRANSLATOR modules: TRANSLATOR_MMIPS and TRANSLATOR_MEM. *Figure 5.5* shows a simple diagram of the new system:

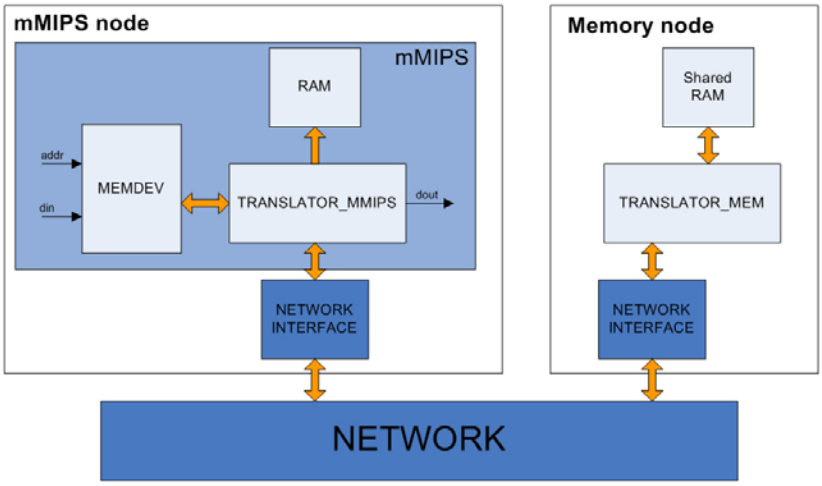

Figure 5.5: mMIPS and memory nodes with TANSLATOR

5.5.1 TRANSLATOR_MMIPS

In the communication between nodes of the *original mNoC* two modules are involved: MEMDEV and Network Interface (NI). *Figure 3.4* shows a diagram of the original configuration. MEMDEV module receives certain input data and address, and depending on the address it could perform two kinds of actions: access to the local RAM memory or use the network to communicate with another mMIPS node.

To be able to communicate with the shared memory node, in the mMIPS side of the *new mNoC* three modules are now involved: MEMDEV, NI and TRANSLATOR_MMIPS. *Figure 5.5* shows the new system. In this case three kinds of actions are possible to perform depending on the input address: access to the local RAM memory, use the network to communicate with another mMIPS node, or perform a communication with the remote memory node using the network. As said before all the mNoC modules will not change their behaviour, therefore MEMDEV and NI have the same functionality even when TRANSLATOR_MMIPS is inside the system. TRANSLATOR_MMIPS is the one that handle the new communication with the remote memory.

System memory map had to change in order to detect whether the data transactions are in local memory, shared memory or between nodes. The new memory map is in *Figure 5.6*:

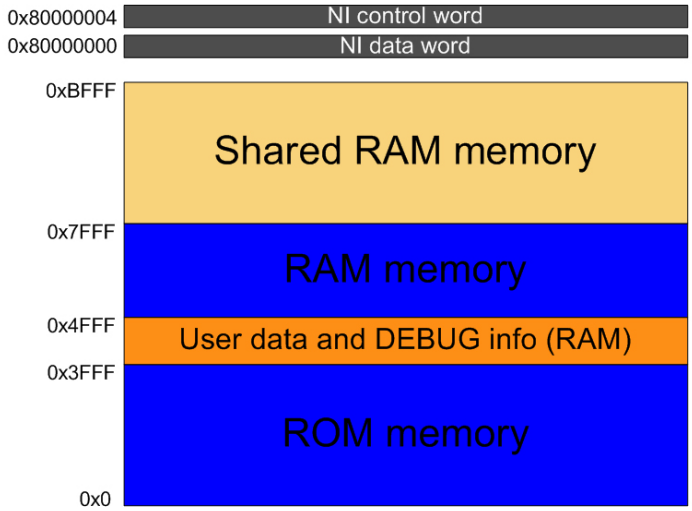

Figure 5.6: New memory map

5.5.1.1 Possible communication actions using TRANSLATOR_MMIPS

In order to understand how the 3 modules involved in communication work, each possible communication action will be explained separately:

- Accessing to local RAM memory.

  Among the whole memory map, MEMDEV can only recognize NI addresses (*0x80000000* and *0x80000004*). In this case MEMDEV sends control signals to the NI in order to use it, otherwise no control signals are

sent and NI is idle. MEMDEV considers the rest addresses as local memory addresses. It will be work for TRANSLATOR_MMIPS to decide whether the addresses belong to local or remote memory. This is the reason why it was said that TRANSLATOR_MMIPS is *transparent* to the *original mNoC*. MEMDEV and NI keep their original functionality and the new module will decide where the data should be located.

Looking to the new memory map above, accessing to local RAM memory happen when *0x3FFF<address<0x8000*. In case of reading or writing in local RAM memory MEMDEV will receive an *address* and a *read* or *write* signal from the processor. Because it is not a NI address MEMDEV considers it as a local memory address, therefore no control data for NI is sent. TRANSLATOR_MMIPS will receive *address* and *read* or *write* signals detecting that the query is for the local RAM memory and not for the remote one. TRANSLATOR_MMIPS will only forward those signals. Consequently an access to local memory will be performed obtaining the data that the processor was requiring.

- Using the network to communicate with other mMIPS node.

NI control and data words (*0x80000004* and *0x80000000* respectively) should be sent to MEMDEV in order communicate with other mMIPS node using the network. MEMDEV will receive an *address* and a *read* or *write* signals on its input buses. In this case MEMDEV recognizes the *address* as a NI address (*0x80000000* or *0x80000004*), therefore control signals for NI are sent from MEMDEV to TRANSLATOR_MMIPS. The new module will receive *address* detecting that it is for NI, and then control signals will be forwarded to NI performing regular *sc_send()* or *sc_receive()* between mMIPS nodes.

- Performing communication with remote memory node.

In this case MEMDEV behaves in the same way as it does in the first case. Shared memory address range is between *0x8000* and *0xBFFF*, therefore MEMDEV will consider the *address* as a local memory one and no control signals for NI will be created. TRANSLATOR_MMIPS will receive *address* and *read* or *write* signals detecting in this case that the query is for the remote memory and not for the local one. Is in this moment when the new module starts to work. Now the data transaction will not be in the local memory and for this reason *read* or *write* signals will not be forwarded to it. No access to local memory will be performed on this way. TRANSLATOR_MMIPS encapsulates necessary data for the remote memory in Data Words, creates Control Words and proper control signals to the NI. Finally NI will split those words into flits and send them using the network. TRANSLATOR_MMIPS also has to wait to receive data from remote memory node in case of reading from shared memory, obtaining the data that the processor was requiring. All this process is done with a Finite State Machine (FSM) inside TRANSLATOR_MMIPS.

The new module translates on this way the regular *load* and *store* in local memory to *sc_send()* and *sc_receive()* through the network in order to be able to access to the shared memory.

Observing to these three possibilities it could be noticed that the new module TANSLATOR_MMIPS will be only active when the incoming address is among the remote memory range. In other cases it only forwards signals keeping the original mNoC functionality.

Table 5.5 shows the first and third actions presented above. Access to local and shared memories are shown, where the behaviour of the different modules involved in the communication could be observed.

Address range	C code	Assembly code	TRANSLATOR_ MMIPS action	Local memory access	NI control signals
0x3FFF<addr<0x8000	*addr=data;	sw $s0,0($t0)	Forward signals to local memory	write(addr,data)	None
	register=*addr;	lw $t0,0($s0)		read(addr)	
0x8000<addr<0xBFFF	*addr=data;	sw $s0,0($t0)	send(addr,data)	None	Generated by TANSLATOR_ MMIPS
	register=*addr;	lw $t0,0($s0)	send(addr) receive(data)		

Table 5.5: mNoC modules behaviour in local and shared memory access

5.5.1.2 Implementation

Once an overall idea of how the new system works has been presented, the question is how TRANSLATOR_MMIPS is able to perform *sc_send()* and *sc_receive()* from *load* and *store*. A detailed diagram of the new module is in *Figure 5.7*:

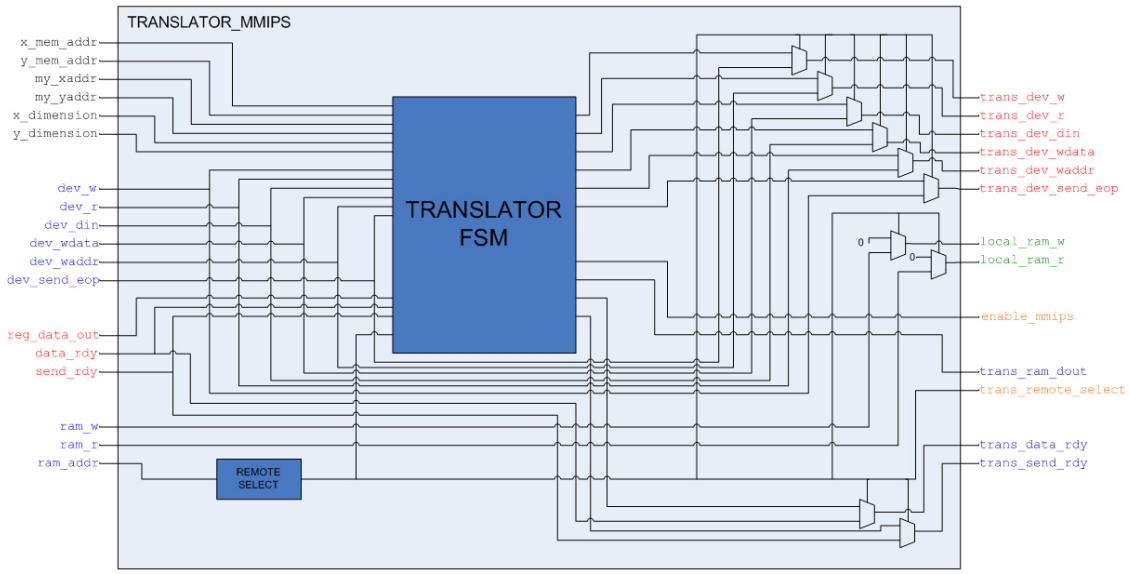

Figure 5.7: Detailed TRANSLATOR_MMIPS block diagram

In this figure, signals in *black* are coming from outside of mMIPS node. The ones in *blue* are for MEMDEV module, *red* for NI, *orange* to control the mMIPS and finally *green* for local RAM memory.

As it could be seen TRANSLATOR_MMIPS is mainly a FSM, which starts to work when *remote_select* signal is active. This signal is active when *ram_address* is among the shared memory address range (see *Figure 5.6*). Signal *remote_select* also has the control of every multiplexer in the module.

Using *remote_select* is how the rest of modules in a mMIPS node keep their original functionality. When it is not active TRANSLATOR_MMIPS just forward all signals to other modules, while when *remote_select* is asserted the output signals for other modules will be the ones generated by the FSM. This selection is made by multiplexers controlled by *remote_select*.

On the left side of the previous figure TRANSLATOR_MMIPS is connected with MEMDEV and NI, while in the right side it is connected MEMDEV, NI and RAM local memory. The interface with those modules is the original one. Only 8 signals have been added to the system. The 6 new signals located on the left side are used to create relative addresses necessary in Control Words sent to remote memory. Signals *trans_remote_select* and *enable_mmips* are used to control mMIPS processor. All of them will be described further on.

5.5.1.3 Functionality

The one that manage the communication with the shared memory in the mMIPS side is TANSLATOR_FSM, which is a finite state machine composed for 9 states as shown in *Figure 5.8*.

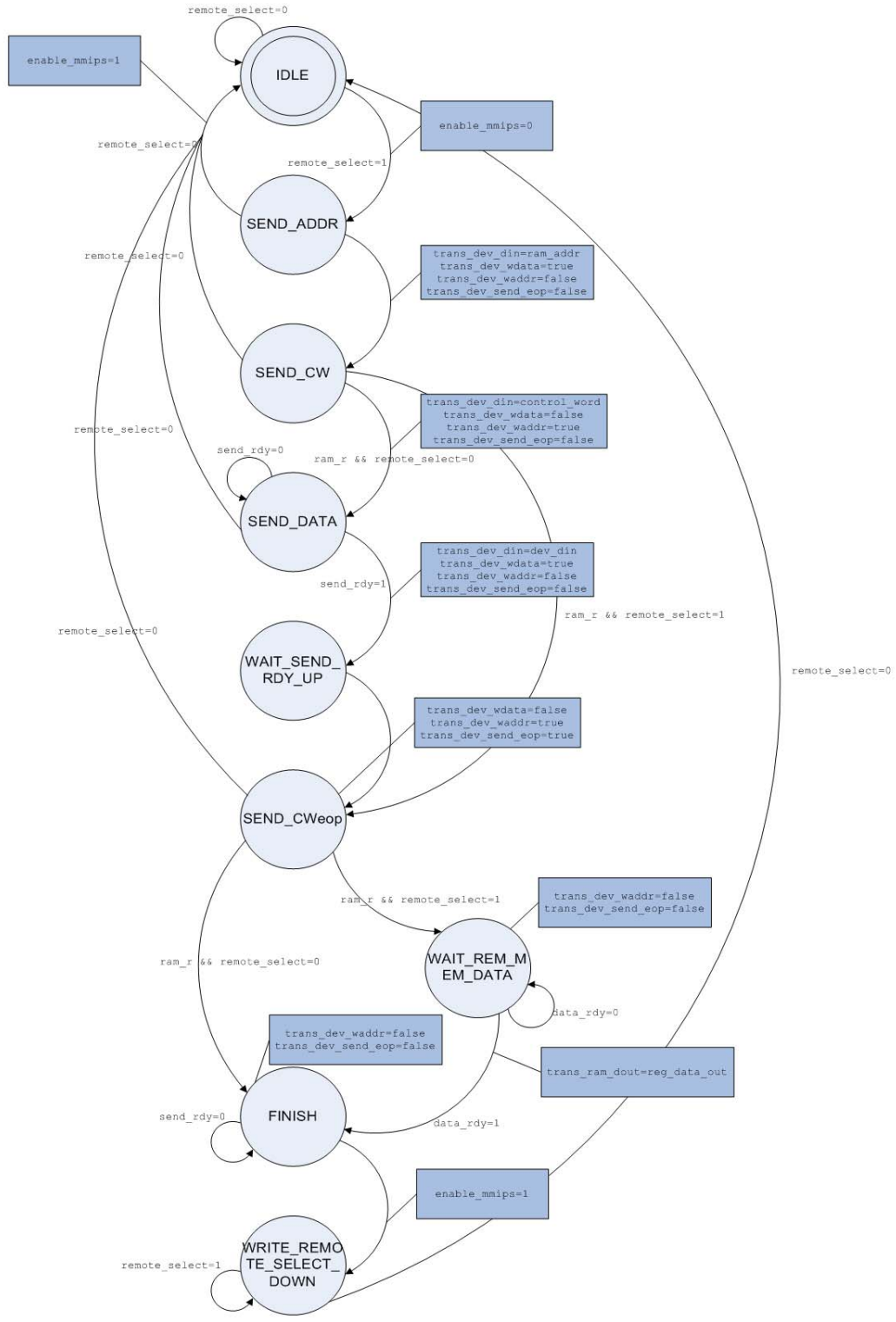

Figure 5.8: TRANSLATOR_FSM state diagram

TRANSLATOR_FSM starts to work when *remote_select* is asserted. Then this machine should be able to distinguish two cases: writing in remote memory and reading from remote memory.

- **Writing in shared memory**

In order to perform a write access in any memory three signals are necessary: *address, data* and *write*. Therefore, TRANSLATOR_FSM should encapsulate those signals in Data Words (DW) and create Control Words (CW); then NI will split those words into flits in order to communicate with the shared memory node.

Looking to *Figure 5.8* it could be noticed where the two necessary DW containing *address* and *data* are created (*SEND_ADDR* and *SEND_DATA* states) as well as their respective CW (*SEND_CW* and *SEND_CWeop* states).

CW and DW created by TRANSLATOR_FSM are the same used by *sc_send()* and *sc_receive()*, which have been presented in section *3.1.3*. Unlike these software functions, TANLATOR_FSM creates CW and DW in hardware.

When a *write* in remote memory is required, flits are created from CW and DW by NI in the same way as explained in section *3.2.1*. The aspect of these flits is shown in *Figure 5.9*:

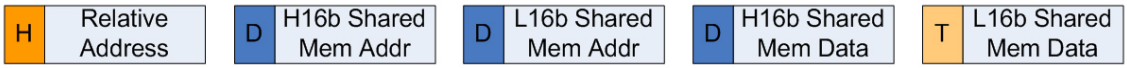

Figure 5.9: Flits in a remote write

First flit labelled with *H* (Header) contains the relative address from the sender node to shared memory node. In the *original mNoC* this address was introduced in software by *sc_send()* function. In *new mNoC* this address is created in hardware using the 6 new signals on the left side of the module presented in *Figure 5.7*. Having sender node address inside the network, destination node address (shared memory node in this case) and network size, calculating relative address to another node could be obtained with the following calculation:

$$sndr\_xaddr \leq dst\_xaddr \Rightarrow rel\_xaddr = dst\_xaddr - sndr\_xaddr$$
$$sndr\_xaddr > dst\_xaddr \Rightarrow rel\_xaddr = n - (sndr\_xaddr - dst\_xaddr)$$

$$sndr\_yaddr \leq dst\_yaddr \Rightarrow rel\_yaddr = dst\_yaddr - sndr\_yaddr$$
$$sndr\_yaddr > dst\_yaddr \Rightarrow rel\_yaddr = m - (sndr\_yaddr - dst\_yaddr)$$

where:

sndr_xaddr, sndr_yaddr: *x* or *y* address of the sender node inside the network.
dst_xaddr, dst_yaddr: *x* or *y* address of the destination node inside the network.
rel_xaddr, rel_yaddr: relative *x* or *y* address from sender node to destination node.
n,m: size of the network. In NETWORK3x2, *n*=3 and *m*=2.

Following three nodes labelled with *D* (Data) and the last one with *T* (Trailer) are Address in shared memory and Data to be stored in this address.

No extra information is necessary in these flits because of the following assumptions [12]:

- Guaranteed delivery.
- Deadlock-free.
- Wormhole routing in the network: flits reach destination in the same order that they were sent.

With these assumptions some bits inside the flits could be saved. Information like *write acknowledge* or *flit numbering* are not necessary in this case.

Another aspect should be considered by TRANSLATOR_FSM when a *write* in shared memory is required. Accessing to local memory just takes 1 clock cycle, therefore when mMIPS processor performs a *write* data is ready in memory the next cycle after *write* signal is asserted. When this happen in remote memory, accessing to it takes more than one cycle because logical delays when creating flits and sending them trough the network.

In order to be able to keep the original functionality of all modules in the mNoC, TANSLATOR_FSM should be able to encapsulate data in Data Words and create Control Words to NI; which will split those words into flits and send them through the network. All this process should be done in only one clock cycle; which is obviously impossible. For this reason a new signal was created: *enable_mmips* (*Figure 5.7*). When TANSLATOR_MMIPS detects an address among shared memory address range the FSM starts to work (*remote_select* is asserted). In this moment signal *enable_mmips* is immediately deasserted, therefore mMIPS processor will be stalled until the *write* action has finished and *enable_mmips* is asserted again. On this way mMIPS processor *will not notice* that a remote access has been performed in the shared memory. This action could be seen in the following figure:

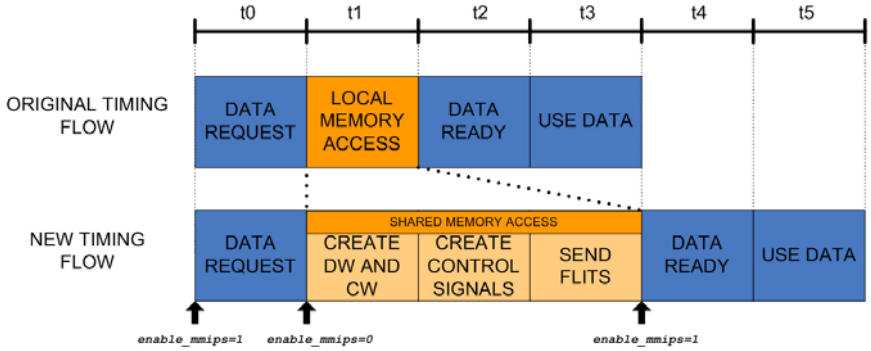

Figure 5.10: Original and new timing flow using enable_mmips

- **Reading from shared memory**

When performing a read in any memory two signals are necessary: *address* and *read*. Therefore, TRANSLATOR_FSM should encapsulate those signals in DW and creates CW; then NI will split those words into flits in order to communicate with the shared memory node.

Looking again to *Figure 5.8* it could be seen that only one CW and one DW containing the *address* to be read are necessary when reading from shared memory. The DW is created in *SEND_ADDR* state, and because a *read* is detected the FSM skips three states creating the CW in *SEND_CWeop* state.

Unlike the writing case now it is necessary to wait for data sent from shared memory node (*WAIT_REM_MEM_DATA* state), for this reason TANSLATOR_FSM has to let the shared memory know that a *read* wants to be performed. The FSM has to encapsulate more data then; the address of the sender should also be among the data sent to shared memory in order to receive data back.

For this reason CW has been changed now. Looking at *Figure 3.5* it could be observed that 11 bits of the Control Word are unused; therefore one of them could be used to inform the remote memory that a *read* wants to be performed. The new CW is shown in *Figure 5.11*:

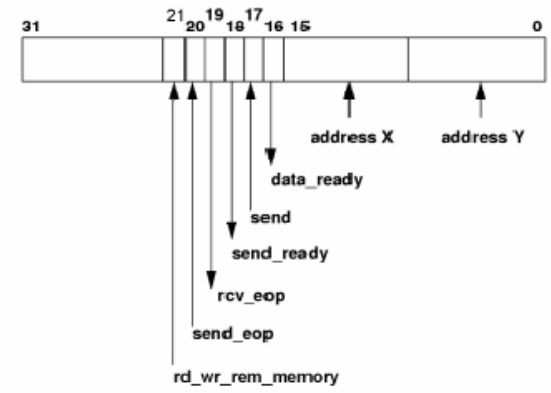

Figure 5.11: Meaning of the bits in the NI New control word (device address: 0x80000004).

Only when TRANSLATOR_FSM detects a read from shared memory bit 21 from CW will be set to 1. In any other case it will be set to 0.

Changes in the CW imply changes in the NI, which is the one that creates the flits that will arrive to shared memory node. If NI detects this bit set to 1 in the CW it will create a new flit labelled with *R* (Read), which contains the relative address from the sender to the shared memory node. On this way, memory node will be able to send data back after a *read* request.

The aspect of the flits created by NI from CW and DW is the one presented in Figure 5.12:

Figure 5.12: Flits in a remote read

It has been possible to create a new flit because there was a flit marker available to encode. Two bits are used to encode four kinds of markers now: Header, Data, Trailer and Read.

First flit labelled with *H* (Header) contains the relative address from the sender node to shared memory node. This relative address is calculated in the same way as did in *writing* case.

Second and third flits are the new ones created when bit 21 of CW is set to 1. Sending this information to memory node, it will be able to calculate the relative address to send data back. One of the flits is empty because using the NI original implementation it is only possible to send multiples of 32 bits after the Header; therefore since each flit is 16 bits it is necessary to send couples of them. Changing this implementation could be one proposal for future improvements of the mNoC.

Finally, last two bits are the ones that contain the address in shared memory to be read.

It should be considered that the original functionality of the *original mNoC* nodes still the same even with these changes presented above in CW and NI. The new CW will be only used in case of remote reading; original bits have not been changed. NI will create a new flit only when bit 21 of CW is set to 1, otherwise only original flits will travel along the network.

Same assumptions about routing adopted in case of *writing* in shared memory are used here. For this reason several bits could be saved in the communication with remote memory.

Signal *enable_mmips* is also used in this case. As well as in previous case, data after reading from local memory will be ready 1 cycle after the *read* signal is asserted. When a *read* from remote memory is required, TANSLATOR_FSM should create new DW and CW, create control signals to NI, send flits to remote memory and finally wait for data from remote memory node. Then data required by mMIPS processor will be ready. All these actions could not be done in only one cycle unless the mMIPS processor is stall since the moment the *remote read* is required until the data from shared memory is ready.

Last aspect to be considered when a reading from remote memory wants to be performed is how TRANSLATOR_MMIPS could give the data received from memory node to MEMDEV. This module is waiting for data from local RAM memory in one of its input buses. I order to keep the original functionality, a multiplexor controlled by *trans_remote_select* (presented in *Figure 5.7*) was introduced in this bus. Proceeding on this way two sources will be connected to the same input bus: output data from local RAM memory and output remote data from TANLATOR_MMIPS. Only when reading from remote memory (*trans_remote_select* = 1) remote data will go to MEMDEV, otherwise local RAM will be the one that pass data to MEMDEV. This idea is shown in *Figure 5.13*.

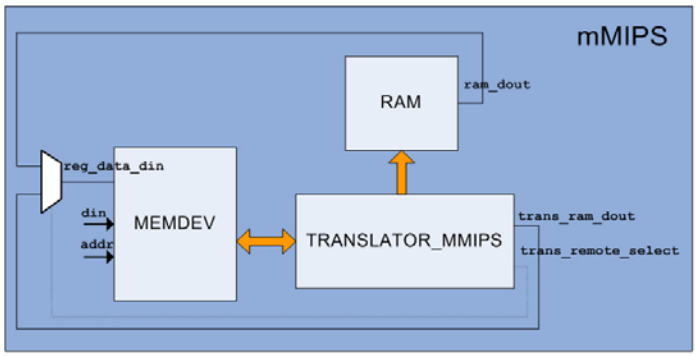

Figure 5.13: Memory data selection

5.5.2 TRANSLATOR_MEM

TANSLATOR_MEM will be the one that handle all memory access queries from all mMIPS nodes to memory node. In memory node side just two modules are involved in communication with the other nodes: NI and TANSLATOR_MEM. *Figure 5.5* shows a diagram of this system.

Shared memory node is a completely new design, and for this reason it was not necessary to keep any original functionality as happened with mMIPS nodes.

In this case two kinds of actions are possible to perform depending on the queries of mMIPS nodes: write in shared memory and read from it. Because of the read action, TRANSLATOR_MEM should be able to send data from shared memory to mMIPS nodes.

5.5.2.1 Implementation

A detailed diagram of TANSLATOR_MEM is shown in *Figure 5.14*:

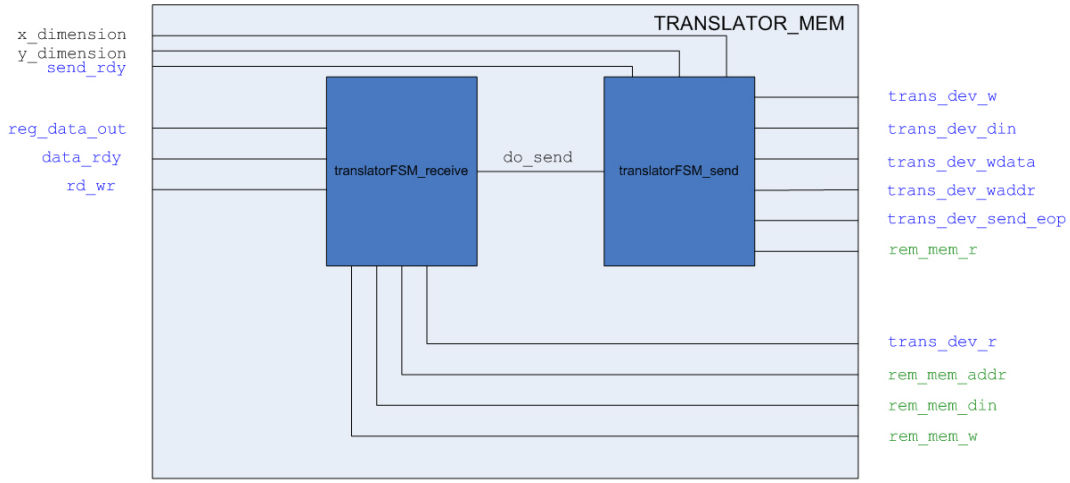

Figure 5.14: Detailed TRANSLATOR_MEM block diagram

In this figure, signals in *black* are coming from outside of memory node. The ones in *blue* are for NI module and *green* ones for shared memory.

Unlike TRANSLATOR_MMIPS this module is mainly composed for 2 Finite State Machines (FSM). TRANSLATOR_MEM is listening in every moment to queries from mMIPS nodes; and only in reading from shared memory cases the new module should send data back to mMIPS nodes creating Data Words (DW) and Control Words (CW). For this reason two independent FSM have been created, which can be working at the same time.

translatorFSM_receive is the one that will listen to mMIPS requests in every moment. It should be able to interpret data given by NI from mMIPS nodes and perform *write* accesses to shared memory. *translatorFSM_send* is the one that waits for a reading request and only in this case it starts to work. It should be able to perform *read* accesses to shared memory and create DW and CW to send data back to mMIPS nodes.

The way to detect a *read* or *write* in shared memory is by means of *rd_wr* signal; which is activated by NI. Internal signal *do_send* is sent by *transltorFSM_receive* to *translatorFSM_send* when a reading request is detected; and it is in this moment when the second FSM starts to work.

5.5.2.2 Functionality

The ones that manage the communication with the shared memory in the memory node side are the two FSM above mentioned. Both of them are composed for 6 states as shown in *Figure 5.15* and *Figure 5.16*.

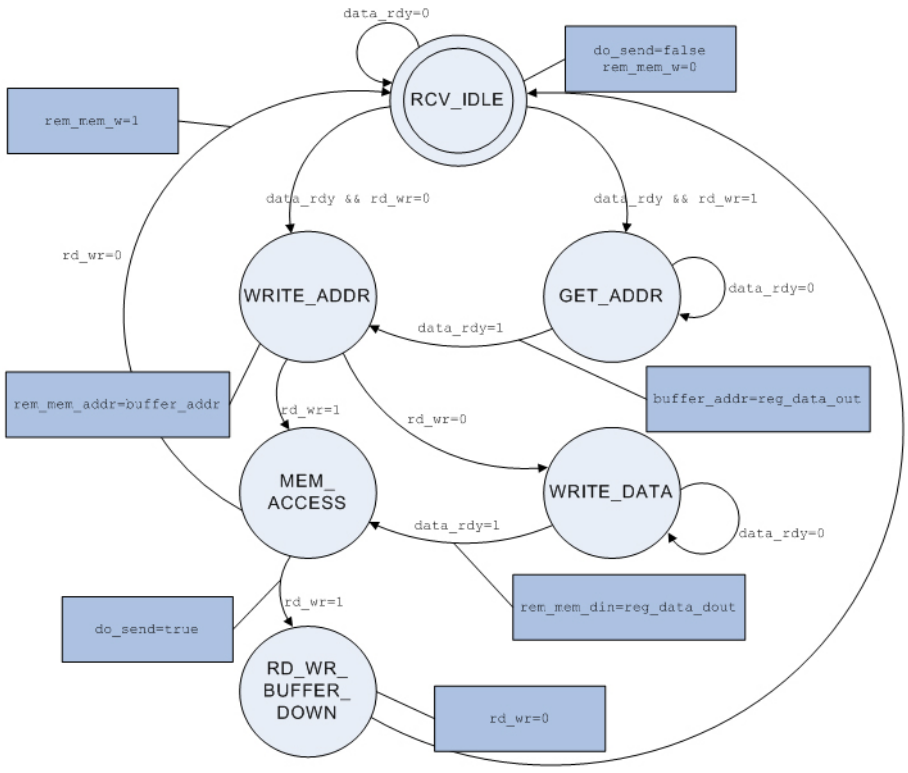

Figure 5.15: translatorFSM_receive state diagram

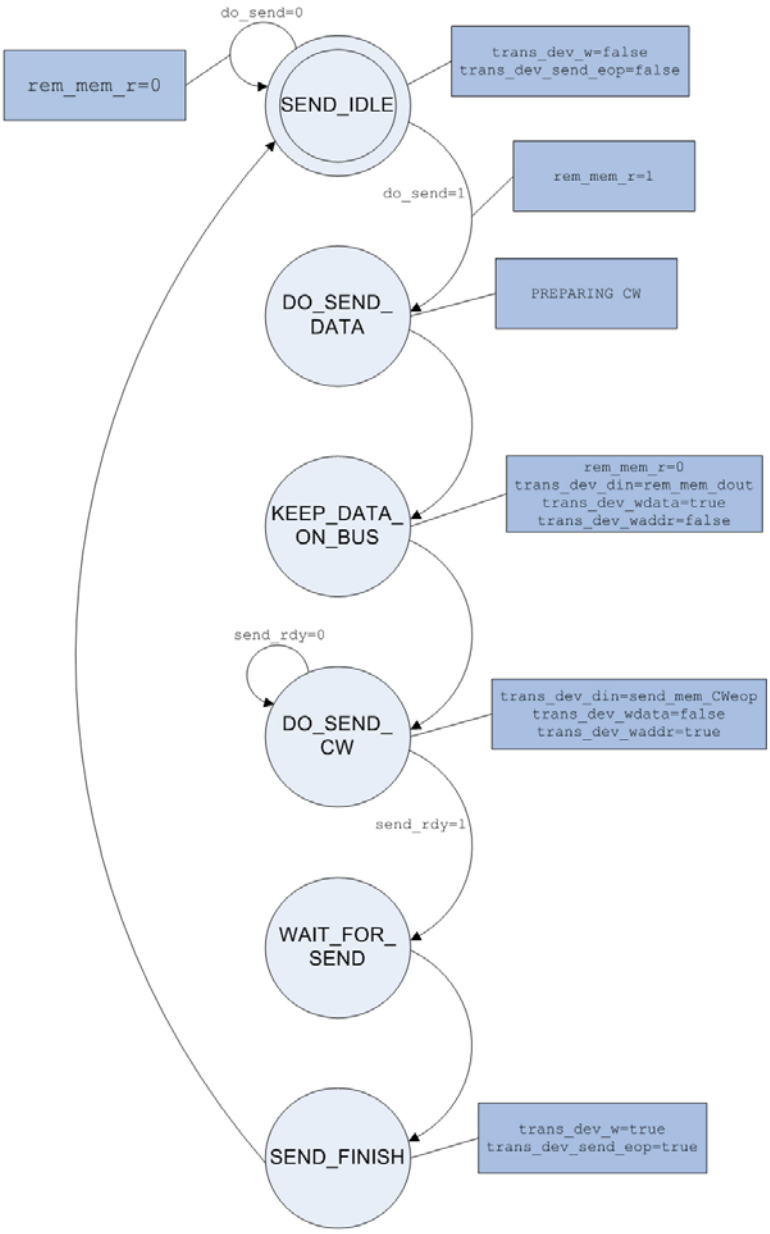

Figure 5.16: translatorFSM_send state diagram

As it was mentioned before, two kinds of actions are possible to perform in shared memory node depending on the request from mMIPS nodes. Each one of them is related with one FSM. In order to understand the behaviour of those State Machines in both cases they will be presented separately:

- **Writing in shared memory**

  translatorFSM_receive is the State Machine used to perform write access to shared memory. It will be always waiting for a request from mMIPS nodes; and it starts to work when *data_ready* signal is asserted. This signal is activated by NI, and it means that a request from a mMIPS node is ready to be read.

As it was presented in section *5.5.1.3* when writing in shared memory was explained, TRANSLATOR_MMIPS will send address and data to shared memory node. *translatorFSM_receive* will receive this information writing the received data in the requested shared memory address (*WRITE_ADDR, WRITE_DATA* and *MEM_ACCESS* states in *Figure 5.15*). Once writing has been done, the FSM is ready to handle another request.

In this case *translatorFSM_send* is idle waiting for a reading request.

- **Reading from shared memory**

The State Machine used to perform read access to shared memory is *translatorFSM_send*. It will be always waiting for a write request from mMIPS nodes; and it starts to work when *do_send* signal is asserted. This signal is activated *by translatorFSM_receive*, and it means that a write request from a mMIPS node is ready to be performed.

How TRANSLATOR_MEM could ascertain whether the request is a read or a write in shared memory? Signal *rd_wr* is the one that answers this question. It has been explained in section *5.5.1.3* that changes in NI were performed in order to create a new flit when a read from shared memory is requested. Signal *rd_wr* was added to NI, which will be asserted when *Read* flit is detected on input bus of NI. A value of 0 in *rd_wr* signal means writing in shared memory; while when 1 is the read value it means reading from shared memory.

Since *rd_wr* is connected to *translatorFSM_receive*, when it is asserted the FSM machine will detect a reading from shared memory request. In this case *translatorFSM_receive* will not perform any access to shared memory. In *Figure 5.15* it could be observed that *translatorFSM_receive* skips *WRITE_DATA* state when *rd_wr* signal is asserted.

When writing in shared memory, TRANSLATOR_MMIPS sends relative address from sender node to memory node, and also the address in shared memory that wants to be read. *translatorFSM_receive* will receive this information storing relative address in a register in order to be used by *translatorFSM_send* afterwards (*GET_ADDR* state), and then putting the address in the shared memory address bus (*WRITE_ADDR* state).

It is in this moment when *do_send* signal is asserted by *translatorFSM_receive* (*MEM_ACCESS* state with *rd_wr* signal active) and *translatorFSM_send* starts to work. This FSM will perform a read in shared memory obtaining the data requested by mMIPS node. Then it should encapsulate this data in Data Words, create Control Words and also control signals to NI. Flits will travel now in the opposite direction containing data from shared memory to mMIPS nodes. This functionality could be easily followed in *translatorFSM_send* state diagram in *Figure 5.16*.

translatorFSM_send should be able to create the relative address necessary to send data from memory node to mMIPS ones. Reading the

relative address stored by *translatorFSM_receive* and using *x_dimension* and *y_dimension* signals (*Figure 5.14*), the relative address to the proper mMIPS node could be easily calculated as follows:

$$rel\_xaddr\_origin = 0 \Rightarrow rel\_xaddr\_dest = 0$$
$$rel\_xaddr\_origin \neq 0 \Rightarrow rel\_xaddr\_dest = n - rel\_xaddr\_origin$$

$$rel\_yaddr\_origin = 0 \Rightarrow rel\_yaddr\_dest = 0$$
$$rel\_yaddr\_origin \neq 0 \Rightarrow rel\_yaddr\_dest = m - rel\_yaddr\_origin$$

where:

rel_xaddr_origin, rel_yaddr_origin: relative *x* or *y* address from sender node to destination node.
rel_xaddr_dest, rel_yaddr_dest: relative *x* or *y* address from destination node to sender node.
n,m: size of the network. In NETWORK3x2, *n*=3 and *m*=2.

Once *translatorFSM_send* has finished sending data to the proper mMIPS node, it is ready to handle another *read* in shared memory.

5.6 Changes on the simulator

As it happened when extending the network from 4 nodes to 6 nodes, all changes done in the SystemC description of the mNoC in order to add a memory node to it have been reflected on the simulator.

A *model* of the new node was added to the simulator in order to test its functionality before hardware implementation, where new trace signals were defined to follow the new node behaviour after running an application.

But the most important change when adding this new node was the simulator access to the new shared memory. In order to check the functionality of the new node, the content of the memory should be read. This is only possible if after an application execution the simulator *dumps* the content of the memory into a file. The simulator is proceeding on this way as it does when *dumping* the content of the others RAM and ROM memories.

As it will be explained in section *5.9.2*, the *dump* file with the content of the shared memory created by the simulator is where the input and output images for *JPEG Decoder application* could be read.

5.7 TRANSLATOR timing characteristics

As it was said in previous sections, two kinds of actions could be performed in shared memory: *read* from and *write* on it. When a memory access request is performed in local memory data is ready 1 clock cycle after asserting *read* or *write* signals; but in shared memory case this does not happen. Due to actions in TANSLATOR_MMIPS, TRANSLATOR_MEM and also because of time spent sending and receiving data using the network; data will not be ready in only 1 clock cycle.

Using the following figures, TRANSLATOR timing characteristics will be explained. The aim of this explanation is ascertain how many clock cycles a *write* or a *read* in shared memory takes.

5.7.1 WRITING IN SHARED MEMORY

A *write* in shared memory is shown in *Figure 5.17*:

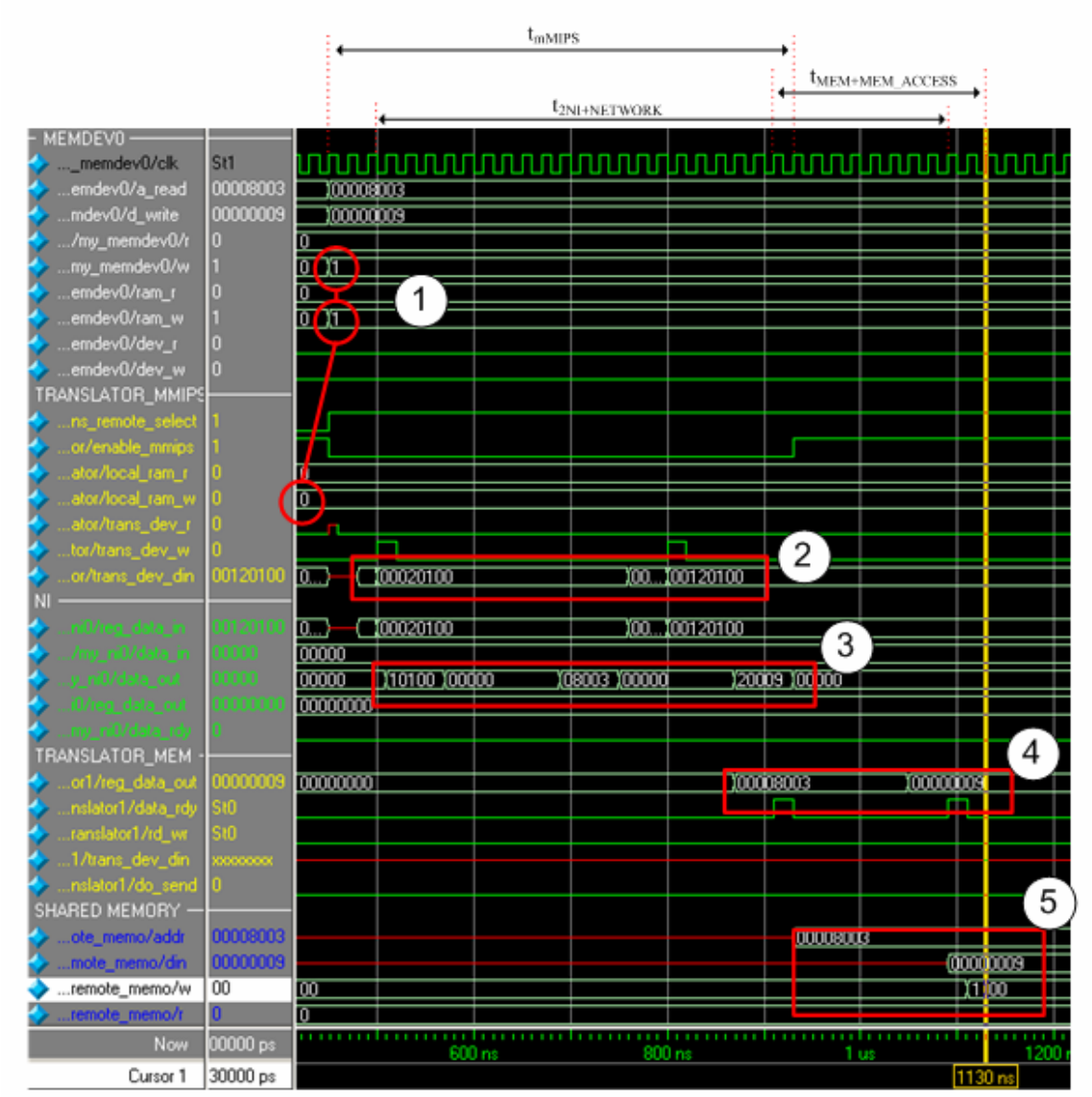

Figure 5.17: Writing in shared memory timing diagram

In point 1 MEMDEV is receiving an address (*0x8003*), data (*0x9*) and a *write* signal. These signals will be forwarded to TRANSLATOR_MMIPS. It detects that this address is among shared memory range; therefore *remote_select* is asserted and *enable_mmips* is set to 0 as well as *local_ram_w*. Here is when TRANSLTOR_MMIPS begins the new functionality not performing an access to local memory and starting to create Data and Control Words.

Creation of CW and DW could be seen in point 2. First DW is put in the output bus to NI (*tras_dev_din*). This data will be stored in a register inside NI waiting for the CW. This is the next step for TANSLATOR_MMIPS; which will assert *trans_dev_w* when this happen. When asserting this signal NI starts to split CW and DW into flits to be sent through the network.

In point 3 those flits could be seen. First *Header flit* (which marker is *1*) contains the relative address from mMIPS node to memory node (*0x0100* in this case). Then the address (*0x00008003*) is sent in two *Data flits* (with *0* as a marker). Finally the data that wants to be stored (*0x00000009*) is also sent in two flits. One of them a *Data flit* containing the higher 16 bits; and the other in a *Trailer flit* (marker is *2*) with the lower 16 bits of the data.

These flits should travel along the network until they reach the destination NI; which has to handle those flits and give the data to TRANSLATOR_MEM. This happens in point 4 where TRANSLATOR_MEM is receiving the address and the data sent by a mMIPS node. NI is asserting signal *data_rdy* when data is ready to be read by TANSLATOR_MEM.

TANSLATOR_MEM should perform a memory access with the data received. Point 5 shows how received data is put on shared memory address and data buses. Finally write signal is activated and write access has been done. After this moment data sent by mMIPS node is ready in shared memory.

Counting how many clock cycles have passed since MEMDEV received a write request to this data was ready in shared memory, is how a *remote write* could be measured. In *Figure 5.17* three times are marked. t_{mMIPS} is the time spent since TRANSLATOR_MMIPS receives a query from MEMDEV until it finishes (*enable_mmips=1*). This process takes *24 clock cycles*.

$t_{2NI+NETWOK}$ is the time that NI and NETWORK spend to delivery data from TRANSLATOR_MMIPS to TANLATOR_MEM. NI splits CW and DW into flits and sends them through the network. Another NI in memory node has to assemble those flits to have data ready. This process takes *21 clock cycles*, where 2 routers inside the network and 2 NI are involved on it.

Finally $t_{MEM+MEM\_ACCESS}$ takes the time spent since TRANSLATOR_MEM receives data from a mMIPS node and performs an access to shared memory having data ready. TANSLATOR_MEM takes *11 clock cycles* to complete this action.

Because of the fact that many actions are done in parallel, time spent to perform a *remote write* is not the sum of t_{mMIPS}, $t_{2NI+NETWOK}$ and $t_{MEM+MEM\_ACCESS}$. The number of clock cycles passed since a write request is received by MEMDEV until data is ready in shared memory is *34*.

5.7.2 READING FROM SHARED MEMORY

Figure 5.18 shows a timing diagram of reading from shared memory:

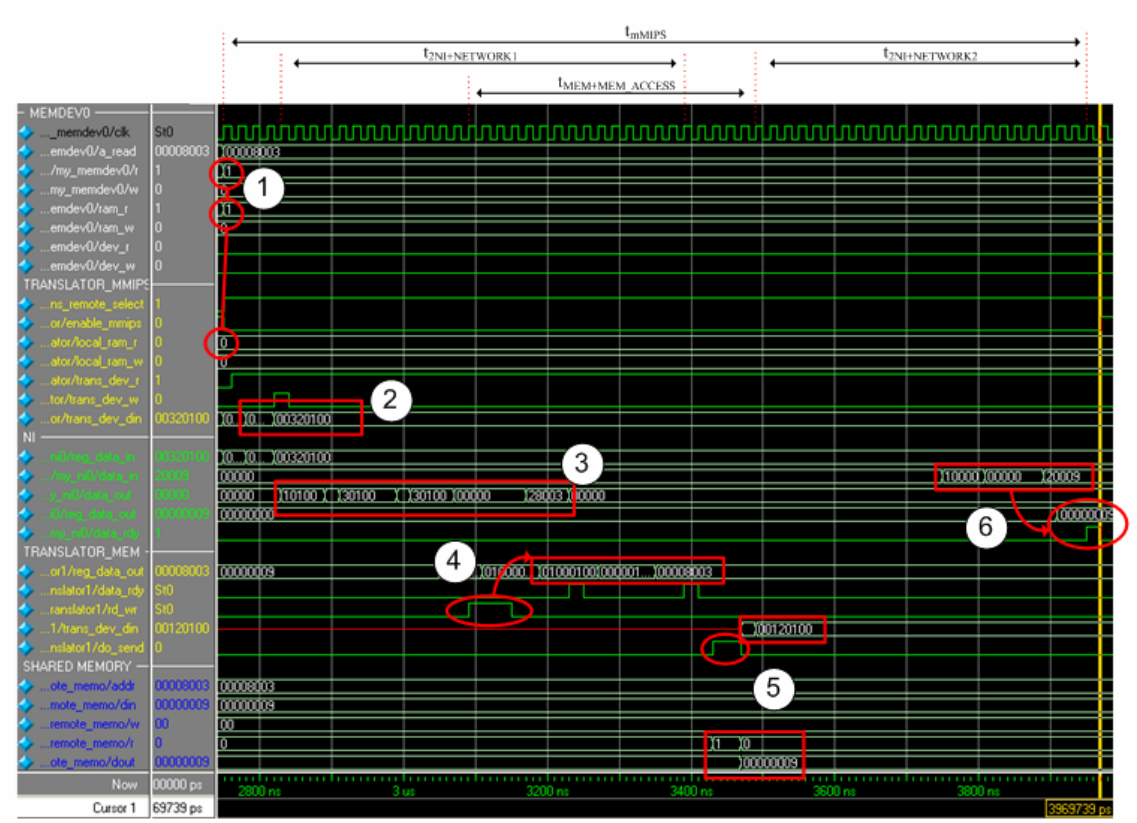

Figure 5.18: Reading from shared memory timing diagram

In point 1 MEMDEV is receiving an address (*0x8003*) and a *read* signal. These signals will be forwarded to TRANSLATOR_MMIPS. It detects that this address is among shared memory range; therefore *remote_select* is asserted and *enable_mmips* is set to 0 as well as *local_ram_r*. TRANSLATOR_MMIPS does not perform any access to local memory, and it starts to create Data and Control Words.

This could be seen in point 2, where TRANSLATOR_MMIPS is sending just one CW and one DW with the address that wants to be read.

NI detects a read from remote memory checking bit 21 of the CW. In this case the new *Read flit* is used. First *Header flit* (which marker is *1*) contains the relative address from mMIPS node to memory node (*0x0100* in this case). Relative address is also introduced in the new *Read flit* (labelled with *3*). Finally the address (*0x00008003*) is sent in a *Data flit* (with *0* as a marker) containing the higher 16 bits; and the remaining 16 bits in a *Trailer flit* (marker is *2*).

Point 4 shows how this data arrives to TRANSLATOR_MEM. NI in memory node detects the new *Read flit* asserting *rd_wr* signal. Now TRANSLATOR_MEM knows that a *read* wants to be done in shared memory. First data arriving to memory node is relative address from mMIPS node to memory node; the second one is the address to be read.

TRANSLATOR_MEM has to perform a read access to shared memory obtaining the requested data. Here is when signal *do_send* is asserted in order to start to create DW and CW; sending this data after to mMIPS node. This action is shown in point 5.

Finally data from shared memory has to travel to mMIPS node, and it is in point 6 where this happen. NI in mMIPS node is receiving data asserting *data_rdy* signal when it is ready to be read by TANSLATOR_MMIPS. After this moment data from shared memory is ready for mMIPS processor and the remote reading is finished.

Four times are marked in *Figure 5.18* in order to know how many clock cycles a *remote read* takes. Meaning of times highlighted in this figure is the same explained in *remote write* case. It is important to notice that in this case flits are travelling along the network twice: from mMIPS node to memory node and also in the opposite direction. These times are:

t_{mMIPS}: 60 clock cycles.
$t_{2NI+NETWOK1}$: 28 clock cycles.
$t_{2NI+NETWOK2}$: 23 clock cycles.
$t_{MEM+MEM\_ACCESS}$: 20 clock cycles.

In this case the number of clock cycles to perform a *remote read* coincides with t_{mMIPS}, because in this case TRANSLATOR_MMIPS is starting the process and it is over when the same module receives data from shared memory.

5.8 Memory consistency

A very important aspect when talking about shared memory between different processors is *memory consistency* [14]. *Memory consistency models* determine when a written value by a processor will be returned by a read performed by other processor.

It is not possible to pretend that a read in a shared memory address X could immediately returns the value written by another processor in the same address X. For example, a processor A could try to read an address X only a few moments after processor B writes some data in X; and it could happen that this data is not ready in memory when A tries to read X. Therefore, the question that *consistency* tries to answer is: when a processor should see a value updated by another processor?

Processors in a multiprocessor system communicate ones with each others through variables in shared memory. Since consistency problems could happen when reading or writing by different processors in those variables, there should be some rules in *read* and *write* actions in order to avoid those problems.

The main problem has been exposed above, and it is a problem of order of reading and writing actions between processors. A simple example of this situation is the following:

```
P1:   A=0;                    P2:   B=0;
      .....                         .....
      A=1;                          B=1;
L1:   if(B==0) ...            L2:   if(A==0) ...
```

Previous code corresponds to two processes running in two different processors. Variables A and B are originally stored in a shared memory with initial value 0. If writing takes immediately effect in the variable values, it is impossible that both *if* statements (in labels *L1* and *L2*) are evaluated as *true*. Because both processes are related sharing variables in shared memory, *if* statements should not be evaluated until writings in condition variables (*A* and *B*) have been achieved.

The most obvious consistency model is *sequential consistency*. With this model the result of any execution should be the same one as if operations were executed in some sequential order keeping internal order of each program. With this model the potential problem explained above is eliminated because in order to continue executing the processes, memory accesses should be completed. Therefore both *if* statements will never be evaluated as *true*.

The way to implement this model is delaying all memory access queries until other actions related with those accesses are finished. In the example explained above read accesses (*A= = 0* or *B= = 0*) must be delayed until write accesses (*A=1* and *B=1*) have been completed.

Although this model present the advantage of simplicity, *sequential consistency* turns a not suitable model when a large number of processors; which means a potential performance reduction.

Another model that can present more efficiency is a *synchronized model*. A program is synchronized when accessing to some shared value that could produce a consistency problem is controlled by a couple of synchronization operations. These operations could happen after a write of one processor in a shared variable and before a read of other processor of the same variable. On this way consistency problems could be avoided.

A good example of this model could be a read and write of a shared value performed by two different processors. Each processor should handle those actions over the shared variable with *lock* and *unlock* operations in order to ensure mutual exclusion for the write access, and obtain a consistent read access. As said above synchronization operations should be done after a write (unlock) and before a read (lock).

It could be said that a program is synchronized when an execution of a write in shared memory variable followed by accesses to this variable contains the following events:

```
write(value)
....
unlock(processor)
....
lock(processor)
....
access(x)
```

Although memory consistency problem does not happen in applications that want to be executed in the *mNoC* (*gossip application* or *jpeg decoder application*), a solution to those potential problems has been considered when developing this project. A *synchronization model* has been the one chosen to solve consistency problems: *Dekker algorithm* [15].

This algorithm tries to solve *deadlock* or *lockout* problems when solving mutual exclusion accessing to shared values. It uses two flags in order to *lock* or *unlock* memory resources which could be accessed by two different processors, and also a *turn* value which give relative priority to those processors.

Even using this solution to solve memory consistency problems, there could be a problem. When compiling two different applications for two different nodes, the only way to synchronize them and avoid these consistency problems is with those flags used by Dekker algorithm stored in shared memory. The problem could come if the compiler *does not know* that certain addresses contain some flags used to solve memory problems. It could happen that the compiler uses these addresses to store some data from one node, overwriting the value of the flags and ruining all synchronization between nodes in this way. For this reason the system should reserve some addresses to store the flags telling the compiler not to use them.

Besides using this algorithm to solve mutual exclusion problems, another issue had to be considered. When writing in remote memory data could be ready in memory a few moments after the writing action has been performed. For this reason, before leaving the critical section the processor should check that data in remote memory is ready to be read. The way proposed to do it was performing a *readback* after every write in critical section.

Considering these two aspects (using *Dekker* algorithm and performing *readbacks* after writes in critical section), two test were performed in the *new mNoC* with shared memory node. In both experiments consistency problems were forced because they do not happen in *mNoC* applications.

First experiment is testing the example presented at the beginning of this section. Two processes are running in two different processors, where two shared variables A and B are condition of two *if* statements. The value of A or B is set in one of the processors and used as *if* condition in the other one. Looking to the code of this example, the *if* statements should be always evaluated as *false*, therefore the program should never be inside of the condition. When solutions exposed above about consistency problems are not considered both processors were inside the *if* statements; and they were working properly when those solutions are used.

The second experiment was a *producer-consumer* application between two processor nodes. The *consumer* was implemented slower then *producer* in order to force the system to fail. In this case *producer* was generating data and *consumer* could not handle it; therefore some data was lost. Using considerations before mentioned any data was lost because using synchronization actions.

All solutions proposed above have been implemented in software, although they could be performed in hardware. Because of time constraints to finish this project,

hardware implementation of these solutions will be proposed for future improvements of the *mNoC*.

5.9 Results

In order to test the functionality of the new mNoC with shared memory node two test applications have been executed in the new system. The first one is a new application created for this aim called *New Gossip*; and the second one is the *JPEG Decoder application* which underwent some changes in order to use shared memory.

5.9.1 NEW GOSSIP

The main aim of this application is to test the communication between mMIPS nodes and remote memory node. It was said in section *5.8* that memory consistency problems will not happen in application running on *mNoC*; therefore with *New Gossip* just communication with shared memory will be checked. Nevertheless, as explained in section *5.8* two particular cases were tested forcing consistency problems in order to test software solutions in the *new mNoC*.

Due to some FPGA restrictions that were explained in section *5.2*, this application will use a NETWORK2x2, where there are three mMIPS nodes and one memory node. *Figure 5.19* shows this system:

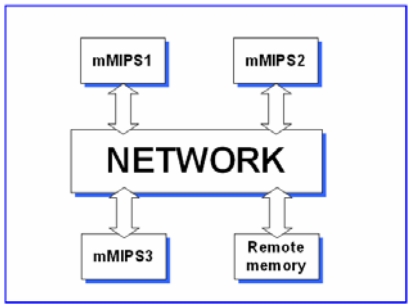

Figure 5.19: NETWORK2x2 with shared memory node

Three mMIPS nodes will start doing the same actions in the system in order to have concurrently access to shared memory. They will read some data from their local memories, and then the three of them will read a remote data from the same location in the shared memory. When reading from local memory having TANSLATOR_MMIPS in the system, original mMIPS functionality could be tested. When reading from shared memory, memory node is receiving three reading queries at the same time. It could be seen in *Figure 5.20 a)*:

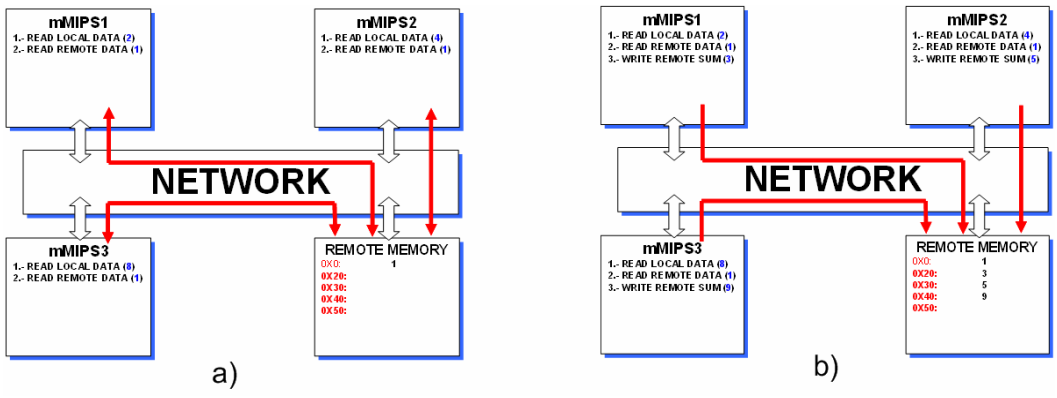

Figure 5.20: New Gossip reading (a) and writing (b) accesses

Once getting remote data from shared memory they will perform a sum between local and remote data. Each processor will write the result on different positions in shared memory. Remember that consistency cases are not tested with this application, therefore mMIPS nodes will write in different memory locations. *Figure 5.20 b)* shows how remote memory will receive three writing requests at the same time.

Next step is testing communication between nodes when TANSLATOR_MMIPS is in the system. In order to do it, the first node will send its sum to the second one; which will add data received to its own sum. The second node will also send this result to the third one that will perform a global sum of the three local results. Once node 3 has the global result, it will write it in shared memory as it could be seen in *Figure 5.21 a)*:

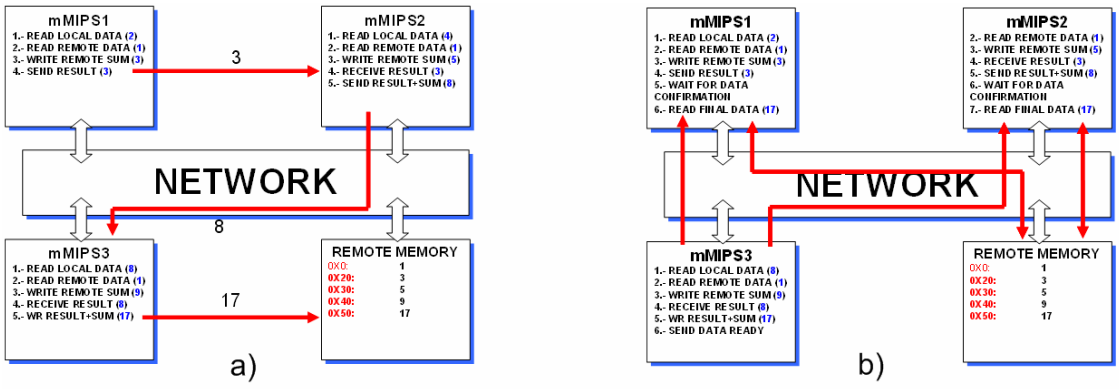

Figure 5.21: New Gossip nodes communication (a) and final result (b)

Nodes 1 and 2 will be waiting for a confirmation from node 3 saying that global sum is already in shared memory. When the third node sends this confirmation, nodes 1 and 2 will read the global result from shared memory. This last step is shown in *Figure 5.21 b)*.

Finally all nodes have been communicating through shared memory, and the three of them have the global sum on their local memories.

Once this application was running on the new system and results obtained were successfully checked it could be said that *new mNoC* was ready to handle communications between mMIPS nodes and memory nodes.

5.9.2 RUNNING JPEG DECODER USING SHARED MEMORY NODE

Running JPEG decoder on the *new mNoC* is how to achieve one of the objectives of this Master Thesis: being able to handle bigger amount of data, which means bigger images for JPEG Decoder.

Once memory node functionality was tested with *new Gossip application*, the new system was ready to test *JPEG Decoder application*. In order to run this application using the new shared memory some changes were made on it.

Originally the input image to be decoded was stored in the local memory of the first node. Then all this data was processed by three tasks mapped in mMIPS nodes, and finally output decoded image could be read from local memory of third node (detailed description of the application in section *3.3.2*).

The purpose now is to have input and output images in shared memory, having more data space to store bigger images than the ones stored in the local memory of the mMIPS. In order to do it, it was necessary to change the application to upload the input image in shared memory. New pointers to the input image on its new location were added to the application, as well as the location for the output decoded image was also changed to shared memory. On this way the content of new shared memory was the one shown on *Figure 5.22*:

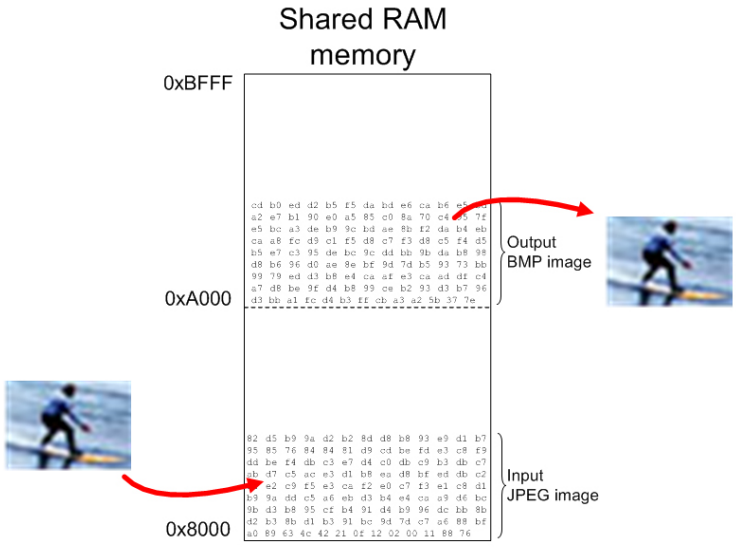

Figure 5.22: Shared memory content after JPEG Decoder execution

In the figure above, it could be seen why it was said that in *JPEG Decoder application* consistency problems do not happen. Node 1 is reading from some location the input image, while node 3 is writing in other location the decoded image. Just

because the implementation of this application, consistency problems do not have to be considered. Nevertheless, it was explained in section *5.8* which software solutions have been proposed in order to solve potential consistency problems with the new system.

A comparison between results obtained from different mNoC configurations wanted to be performed. Because of this reason, the input image chosen to be decoded by JPEG decoder application using the *new mNoC* was the same that the one used by the *original mNoC*. In this case a 32x24 pixels color JPEG image was decoded, but thanks to the shared memory extension, a bigger image could be used.

Four results when running *JPEG Decoder application* in four different mNoC configurations wanted to be compared: *original mNoC* with and without multiply instruction (already exposed in section *4.2.2*), and *new mNoC* using shared memory with and without multiply instruction. *Table 5.6* shows these results:

Configuration	Simulation time without multiply instruction (min)	Simulation time with multiply instruction (min)	Improvement with/without multiply instruction (%)
Original mNoC	1727.17	1628.76	**5.7**
New mNoC	1899.33	1804.37	**5**
Deterioration with/without shared memory (%)	**9**	**9.7**	

Table 5.6: Simulation time comparison between Original and New mNoC running JPEG Decoder

From the four results exposed above, two comparisons could be done. First the improvements with the new multiply instruction when the application is run on the new system. Again, it could be seen that a similar result than the one obtained with the original system was achieved; which means saving almost two hours of simulation.

Another result could be extracted from this table. It is interesting to know how the system is deteriorated when performing communications with the memory node, both with and without multiply instruction cases. Obviously, the new system is slower than the original one because TRANSLATOR processes and also delays sending and receiving information from shared memory. In both with and without multiply instruction cases the deterioration was 9.7 and 9 respectively.

Looking to the simulation time of the original mNoC without multiply instruction and the new one with this instruction, it could be noticed that the deterioration because of the shared memory could be compensated with the time saved with the multiply instruction. Therefore, considering the advantages of the memory node, the disadvantages because of the delay in communications and the final simulation time; it could be said that the shared memory simulation time is a good result.

Chapter 6

Motion JPEG decoder application

6.1 Introduction

Previous chapters have presented necessary steps to reach the main objective of this project: running some MJPEG decoders in parallel over an extended mNoC platform. In this chapter the achievement of this main objective is exposed.

Extending the mMIPS a faster simulation and execution time have been achieved. Adding a shared memory node gives the possibility to mMIPS processors to handle more data; which in case of a JPEG Decoder means handle bigger images.

Once those extensions have been successfully performed and tested, last task is changing *JPEG Decoder application* in order to get a *Motion JPEG Decoder application*. Since Motion JPEG is a sequence of JPEG frames, transition from original decoder to a Motion JPEG one will consist in a continuously decoding of each frame.

6.2 Implementation

As it has been exposed in section *3.3.2 JPEG Decoder application* is divided in three tasks; which are mapped in three mMIPS nodes. First task is reading data from its own mMIPS local memory processing it in some way; and results are sent to the second node. Second task will proceed in the same way sending resulting data to third task; which finish this process obtaining the decoded image and storing it on its local mMIPS memory.

During the decoding of a single image those nodes could be idle in certain moments. For example, at the beginning of the process third node is waiting for data processed by node two; which means inefficiency using mNoC resources. Something similar happens at the end of the process; where first node has finished but still waiting for node three to finish.

Implementing a *MJPEG Decoder* this aspect is improved. In this new case a new image is immediately decoded after a previous one has finished; therefore nodes are really working in parallel decoding different images at the same time. When task 1 finish processing data from *image 1*, the third one still working on it. Is in this moment when *image 2* starts to be decoded by node 1; therefore two images are now in the system and nodes are not idle.

Of course at the beginning and the end of the process the problem mentioned before still existing because there is not more images, but a significant improvement is achieved in the middle of the process when decoding more that one image.

For this reason, it could be tested that decoding two images with *MJPEG Decoder* is faster than decoding two separate images with a *JPEG Decoder*. This will be presented in *Results* section. *Figure 6.1* tries to show this fact. Figure *a)* is the execution time of *JPEG Decoder application*, where total execution time is *t1+t2*. Figure *b)* is execution time of *MJPEG Decoder application* where total execution time is lower than *t1+t2*.

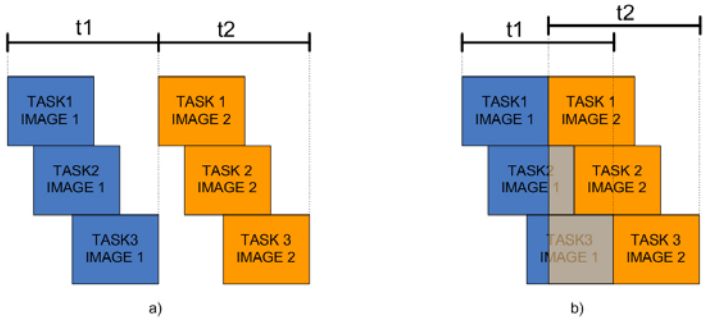

Figure 6.1: JPEG Decoder (a) and MJPEG Decoder (b) execution times

Besides changes already done in *JPEG Decoder application* when using shared memory node, more changes have been done on it in order to obtain a *MJPEG Decoder application*.

Now more images have to be uploaded to shared memory; therefore new pointers to input images have been added to the application, as well as changes in the location of the decoded images. Each node has to continue working until all images in shared memory have been decoded; therefore a loop with parameters like *current image* or *number of images* were added to the application. On this way the content of shared memory was the one shown on *Figure 6.2*:

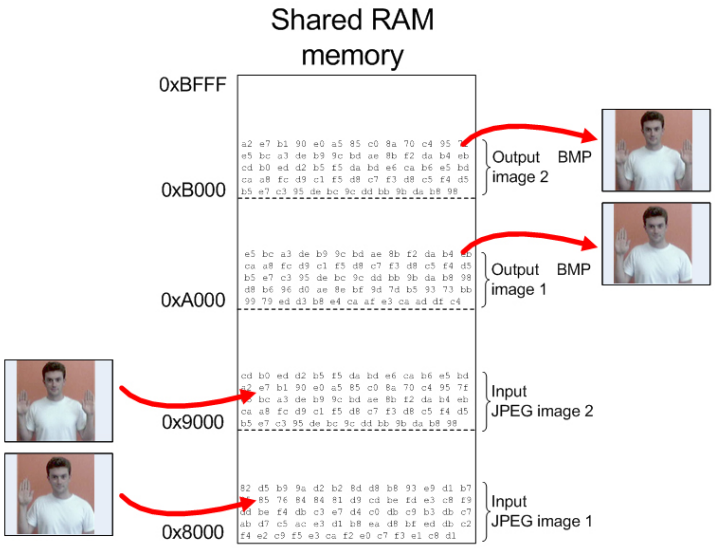

Figure 6.2: Shared memory content after MJPEG Decoder execution

6.3 Results

Two new images (*Figure 6.2*) were created to be decoded by *MJPEG Decoder application*, being able to test its functionality on this way. In this case both of them were 32x24 pixels color JPEG images. Just two images were chosen because simulation time reasons, but the new system is able to handle as many images as shared memory can fit.

In order to measure the improvements of *MJPEG Decoder* compared with *JPEG Decoder*, the images were decoded by both systems. The simulation was also made with and without multiply instruction. *Table 6.1* shows these results:

	Simulation time of two separately images using *JPEG decoder* (min)	**Simulation time of two images using *MJPEG Decoder* (min)**	**Improvement *MJPEG-JPEG* (%)**
Without multiply instruction	3798.66	3627.8	**4.5**
With multiply instruction	3608.74	3484.5	**3.5**
Improvement multiply instruction (%)	**5**	**4**	

Table 6.1: Simulation time comparison between JPEG and MJPEG decoders

As well as results obtained in section *5.9.2*, two comparisons could be made with these four results. First the improvements when using the multiply instruction in both JPEG and MJPEG decoders. Again this result is around 5%; which means saving more than three hours of simulation time.

Another result could be extracted comparing the improvement when using MJPEG with using JPEG decoder. When decoding two images the new system with and without multiply instruction is saving 3.5 and 4.5% of the time respectively; this means saving around 3 hours of simulation time.

It has to be considered that this result is just with two images, and using more ones the improvements will be bigger because of the reasons explained in *Figure 6.1*.

Last result that could be seen from this table and *Table 3.1* is an overall balance between simulation time of two images with the original mNoC without multiply instruction and simulation time of two images with the new system and multiply instruction. In this case the times are almost the same: 3454.34 and 3484.5 minutes respectively. For this reason it could be said that even introducing delays in communication using the network, all advantages of a memory node can be used without a simulation time penalty.

Chapter 7

Hardware implementation

7.1 Original mNoC implementation

The *original mNoC* introduced in *Chapter 3* is a fully synthesizable design, which can be uploaded into a FPGA. The one used for the mNoC is a Xilinx Virtex II FPGA which is on a BenOne development board from Nallatech.

The NOC itself cannot be placed on a FPGA, because it lacks an interface to the outside world. It is necessary such an interface for things like starting the processors and uploading or retrieving their memories. Two modules provide this interface. The first module, Comm.Core, is connected to the pins of the FPGA on one side and to the second module on the other. The Comm.Core module implements a register or DMA based communication scheme. The second module, BENIF_NET_WRAPPER, uses the register communication scheme of Comm.Core. Register addresses and their data have special meanings to the BENIF_NET_WRAPPER module and can be used for the desired processor control functionality and memory access.

An extension of *Figure 3.1* is in *Figure 7.1*, where the whole system is shown:

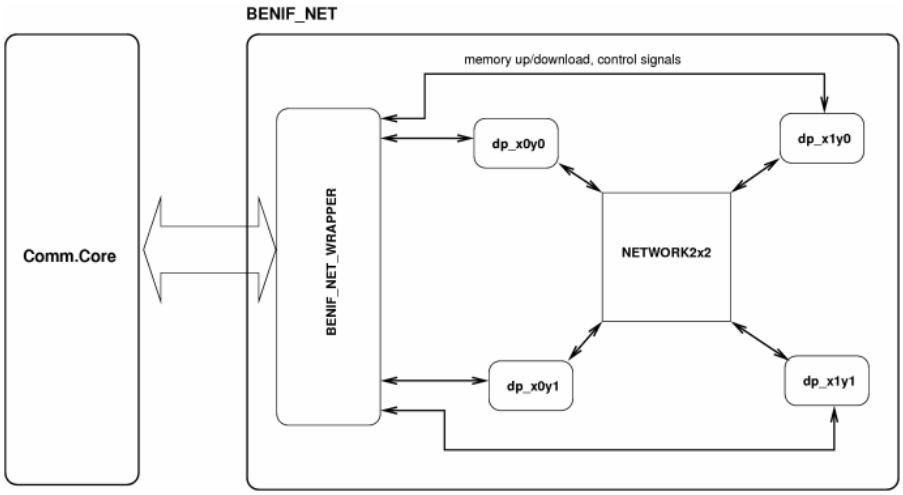

Figure 7.1: Top level view of the contents of the FPGA with the NoC and extra modules

- *Comm.Core*: The Comm.Core is supplied by development board manufacturer Nallatech. It provides a register or DMA based communication scheme between the PC that runs

Nallatech software on the one side and the interface module BENIF_NET_WRAPPER of BENIF_NET design on the other side.

- *BENIF_NET*: This is the top level module which encapsulates the mNoC including its interface to the Comm.Core of Nallatech, BENIF_NET_WRAPPER.

7.1.1 DESIGN FLOW

Figure 7.2 shows the mNoC design flow used to run Software applications in both the Hardware simulator and the FPGA board:

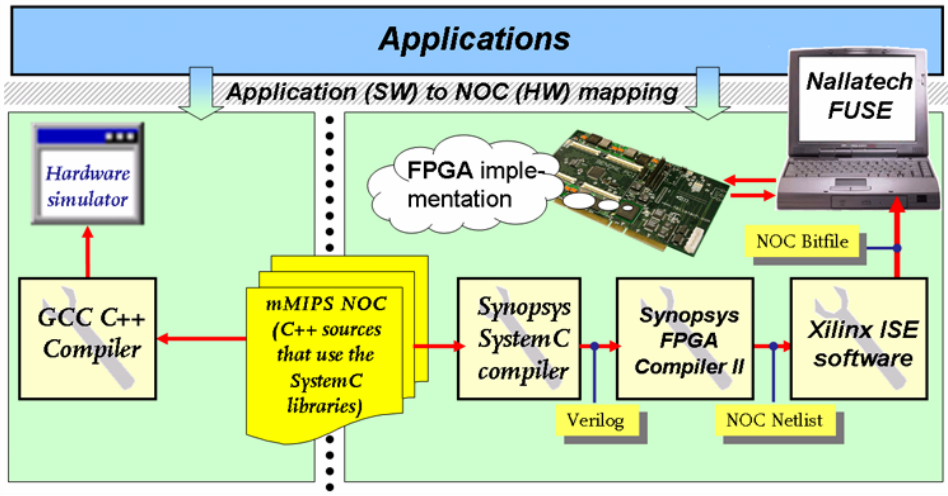

Figure 7.2: mNoC design flow

The left side of the image represents the flow to obtain the Hardware simulator of the system. All results explained in chapters 4, 5 and 6 were obtained following this flow. Compiling all mNoC SystemC descriptions with *GCC compiler* was enough to use the mNoC Hardware simulator.

On the right side of *Figure 7.2* the steps to upload the FPGA with the mNoC design are shown. First a Verilog file is obtained after compiling the SystemC description of the mNoC with *Synopsys SystemC compiler*. *Synopsys FPGA Compiler II* gives the mNoC Netlist, which will be used by *Xilinx ISE* to create the mNoC Bitfile to upload the FPGA with the mNoC design.

7.1.2 HARDWARE FEATURES

Attending to the text reports created after running *Xilinx ISE* and obtaining the bitfile that will be uploaded to the FPGA, the Hardware features of the mNoC can be observed.

- Maximum frequency: **33.247 MHz** (Minimum period: 30.078 ns)
- FPGA resources usage:
  - Number of RAMB16s: 72 out of 96 (75%)
  - Number of SLICEs: 4951 out of 14336 (34%)

7.2 New mNoC implementation

7.2.1 HARDWARE FEATURES

Following the same design flow performed for the *original mNoC*, the following Hardware features were obtained from the *new mNoC* synthesizable SystemC description:

- Maximum frequency: **31.899 MHz** (Minimum period: 31.349 ns)
- FPGA resources usage:
  - Number of RAMB16s: 62 out of 96 (64%)
  - Number of SLICEs: 4730 out of 14336 (32%)

7.3 Problems with the FPGA

Although the *new mNoC* is a complete synthesizable design, some problems appeared when running test applications on the FPGA. The system was successfully executed in the simulator obtaining expected results; but incoherent results were detected after the Hardware execution finished. Apparently, a bad synchronization in the mNoC memories could be the reason of this strange behaviour.

Due to the fact that even the original design was not working on the FPGA, three actions were made trying to find the solution to the problem:

- Deep inspection of the *original mNoC* SystemC description.
- Running FPGA with alternative simple test applications in order to identify where the problem was.
- Single simulation of mNoC modules Verilog files (MEMDEV, NI, NETWORK2x2...).

Unfortunately no solutions were found and the new mNoC was tested just with the simulator obtaining successful results.

Trying to find the solution to this FPGA problem was out of the scope of this project because the starting point of it, which is the *original mNoC,* should be working and it was not. Therefore, it could be proposed for future work to find the problem and solutions to the FPGA implementation of the mNoC.

The time spent trying to find solutions to this FPGA problem, the impossibility of test the Hardware design of the *new mNoC* with external devices connected to the board, together with the time constraints to finish this project are come of the reasons

why one of the objectives of this project was not achieved: the I/O nodes for the mNoC were not implemented.

Nevertheless, once the communication with the new shared memory node was achieved, a new communication with an external device using an I/O node should not entail too many difficulties. Buffers to store input and output data in order to synchronize external devices and mMIPS nodes behaviours, using the new network communication protocol or doing some changes on it, and finally creating some custom interface with external devices could be enough to perform external communication.

Chapter 8

Conclusion and recommendations

In this chapter, the conclusions for this project are presented as well as some recommendations

8.1 Conclusions

As it has been said in this report, many changes had to be done in the *original mNoC* in order to achieve the main objective proposed for this project. Therefore, some conclusions can be extracted from each change.

Extending the mMIPS processors when introducing new Hardware instructions improved functionality of the mNoC, but a big speed-up of execution time was not produced. In order to have a faster simulation, a new simulator with a higher abstraction level is necessary.

Due to the limitations of the current FPGA exposed in section *5.2*, the enlarged *new mNoC* was only run in the simulator where the communication between six nodes was tested. In order to upload the FPGA with this new design it is necessary to use a new FPGA with more resources available.

The new shared memory node created for this project has a good functionality, and it does not introduces too much execution delay penalty because of the fact that now remote communications are performed. This timing penalty could be compensated with the improvements introduced by the new Hardware multiply instruction. Considering these two aspects, and as it was explained in section *6.3*, the *new mNoC* has the advantages of a shared memory node with the same simulation time.

One aspect that should be highlighted about the new module TANSLATOR is that it keeps the *original mNoC* functionality. Just data location changes when using the new module or not, the original modules behaviour is the same. When TRANSLATOR is not inside the system, all data goes to local memory. When the new module is inside of the mNoC data could go to local or remote memory, but it is TANSLATOR the one that decides where the data should go.

And finally, the most important conclusion about this project is that the main objective proposed at the beginning of this Master Thesis has been achieved obtaining successful results: running on the mNoC several Motion JPEG decoders in parallel using a shared memory node.

8.2 Recommendations

For a possible future research continuing the present work, the following suggestions are made:

- Raise the simulator abstraction level from cycle-accurate RTL level to non-synthesizable cycle-accurate or instruction set simulation. This improvement of the simulator could give faster simulation times.
- Spend more time and effort trying to find FPGA problems in order to upload the available FPGA with the original and new designs, and run test application in Hardware.
- Implement an acknowledge write, which could give a more general solution to communication with shared memory node or even I/O nodes.
- It could be interesting to implement in Hardware the Software proposals presented in this project about memory consistency problems.

Bibliography

[1] Yi-Ran Sun. "Simulation and Performance Evaluation for Networks on Chip". *Master of Science Thesis*. December 2001.

[2] Rochit Rajsuman. "System-on-a-chip: design and test". *Artech House Publishers*, 2000.

[3] ITRS, International Technology Roadmap for semiconductors, Update 2002.

[4] G. Smith. DAC Panel Presentation. *40th Design Automation Conference (DAC)*. Anaheim. June 2003.

[5] Nikolay Kavaldjiev, Gerard J. M. Smit. "A survey of Efficient On-Chip Communications for SoC". University of Twente. Enschede, The Netherlands, 2003.

[6] A. Slusarczyk, S. Stuijk, M. Visser. "mMIPS Network-on-chip website". *http://www.es.ele.tue.nl/~mininoc*.

[7] W.Dally. "A VLSI Architecture for Concurrent Data Structures". *Kluwer Academic Publishers*, September 1987.

[8] S. Stuijk. "Design and implementation of a JPEG decoder". *Practical Training Report*. Technical University of Eindhoven. Eindhoven, The Netherlands. December 2001.

[9] Xilinx website. *http://www.xilinx.com*.

[10] Xilinx. "Virtex-II Platform FGPAs: Complete Data Sheet". June 2004.

[11] Xilinx. "Libraries Guide, ISE 6.3i". 2004.

[12] W. J. Dally, C. L. Seitz. "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks". *IEEE Transactions on Computers*. 1987.

[13] W.J. Dally, B. Towles. "Principles and Practices of Interconnection Networks". *Morgan Kaufmann Publishers*. 2004.

[14] J. L. Hennesy, D. A. Patterson. "Computer Architecture a Quantitative Approach". Second Edition. *Morgan Kaufmann Publishers*. 1996.

[15] W. Stallings. "Operating Systems: Internals and design Principles". Fourth Edition. *Prentice Hall Publishers*. 2001.

[16] D. A. Patterson, J. L. Hennesy. “Computer Organization & Design”. Second Edition. *Morgan Kaufmann Publishers*. 1998.

[17] J. Henkel, W. Wolf, S. Chakradhar. “On-Chip networks: A scalable, communication-centric embedded system design paradigm”. *17th International Conference on VLSI Design (VLSID’04)*. Princeton University. Princeton, USA. 2004.

[18] J. Henkel. “Closing the SoC Design Gap”. *Computer Magazine*. September 2003.

[19] Synopsys, Inc. “SystemC v2.0 User´s Guide”. 2002.

[20] A. Jantsch, H. Tenhunen. “Networks on Chip”. *Kluwer Academic Publishers*. February 2003.

[21] A. Shickova. “Architecture Exploration of Interconnection Networks as a Communication Layer for Reconfigurable Systems”. *Master of Science Thesis*. Leuven, Belgium. July 2003.