North Dakota State University
Graduate School

Title

Implementation of Bus-Based and NoC-Based

MP3 Decoders on FPGA

By

Kianoosh Karami

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

ABSTRACT

Karami, Kianoosh, M.S., Department of Electrical and Computer Engineering, College of Engineering and Architecture, North Dakota State University, November 2011. Implementation of Bus-Based and NoC-Based MP3 Decoders on FPGA. Major Professor: Dr. Cristinel Ababei.

The trend of modern System-on-Chip (SoC) design is increasing in size and number of Processing Elements (PE) for various and general purpose tasks. Emergence of Field Programmable Gate Array (FPGA) into the world of technology has lowered the limitations faced by Application Specific Integrated Circuit (ASIC) design. FPGA has a less time-to-market and is a perfect candidate for prototyping purposes due to the flexibility they create for the design and this is the key feature of the FPGA technology. Technology advancements have introduced reconfiguration concepts which increase the flexibility of FPGA designs more.

One method to improve SoC's performance is to adopt a sophisticated communication medium between PEs to achieve a high throughput. Bus architecture has been improved to meet the requirements of high-performance SoCs, however, its inherently poor scalability limits their enhancement. The Network-on-Chip (NoC) design paradigm has emerged to overcome the scalability limitations of point-to-point and bus communication.

This thesis presents an investigation towards NoC versus bus based implementation of an SoC. An MP3 decoder has been selected as an application to be implemented on the proposed design. The final design in the thesis demonstrated that the NoC based MP3 decoder achieves a 14% faster clock frequency and real time operation with the NoC based design decode an MP3 frame on average in 10% less time that the bus based MP3 decoder.

ACKNOWLEDGMENTS

I am very grateful to take this opportunity to thank Dr. Cristinel Abebei, Dr. Chao You, Dr. Benjamin Braaten and Dr. Chad Ulven for serving on my graduate committee. Especially, I would like to thank Dr. Abebei for his constant support and patience with me.

I would like to thank the Center For Nanoscale Science and Engineering, Mark Pavicic, and Mike Reich for believing in me and giving me the oppurtunity to learn and gain experience under their supervision.

Finally, I would like to thank my family for always believing and supporting me throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

1.1. Field Programmable Gate Arrays

An FPGA is a semiconductor device designed to be configured by the designer after fabrication. Hardware Description Language (HDL) is generally used to determine the software and behavior configuration of an FPGA. FPGAs are free of any previously installed hardware function, therefore allowing the designer to adapt the FPGA to any new standards and configure the FPGA for any desired application while maintaining the freedom to reconfigure the FPGA for any other specific applications later hence "field programmable". Reconfigurable hardware gained attention for its ability to prototype applications faster than other technologies. Reconfigurability is one of the many reasons why FPGA technology has replaced ASIC design's long and expensive time-to-market [1].

There is no "standard" structure for FPGAs, therefore many different architectures exist. The most simple structure of an FPGA cell is shown in Figure 1.1. A simple FPGA cell structure consist of I/O blocks, Configurable Logic Blocks (CLB) and routing channel. Most of the CLBs contain Look Up Tables (LUT) along with flip flop architecture in some determined architecture. LUT contains the configuration logic of the FPGA cell. The routing channel is responsible for routing a circuit based configuration of the CLBs. Due to the high count of FPGA cells that is continuously increasing with advancements in technology, Computer Aided Tools (CAD) are needed to properly implement a design on an FPGA chip. Xilinx [2] and Altera [3] are the two leading companies that have been impacting the FPGA technology with their product and offer many sorts of different FPGA chips. Xilinx provides Xilinx ISE as a CAD tool for their FPGA technology. Xilinx ISE provides designers with a set of tools to implement a design on an FPGA.

A bitstream is the configuration data downloaded into an FPGA chip to be configured into a new design. The conversion of HDL source into a bitstream consists of two operations: synthesis and implementation. Xilinx ISE provides Xilinx Synthesis Tools (XST)

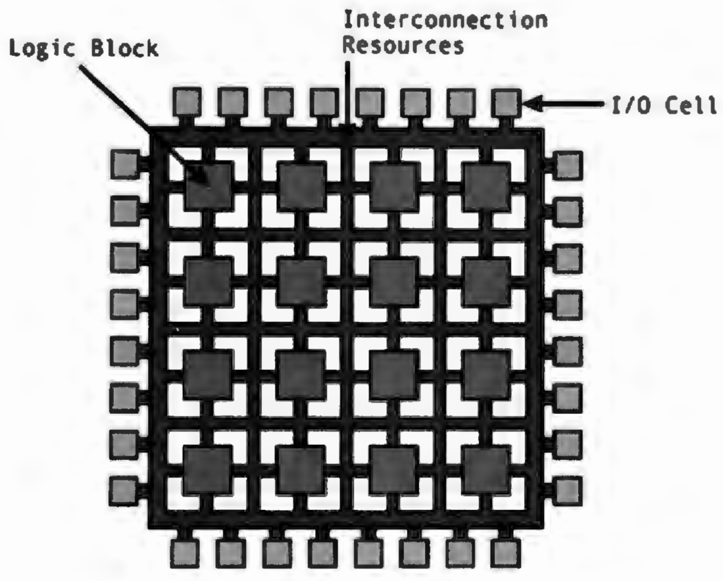


Figure 1.1. Simple FPGA structure with interconnection blocks serving the same purpose as the routing channel.

and other tools [2] to synthesize and generate bitstreams from the design for the targeted FPGA chip. The designer can use Xilinx ISim to verify and simulate the design or any other HDL or schematic based design. The designer could also use on chip testing such as ChipScope to troubleshoot the design.

1.2. Bus

The processing power of SoC has increased to embed data intensive applications, and more attention has been focused on the communication aspect of the SoC. SoC enables implementation of wide range of applications that employ parallel processing with some required to fulfill real-time requirements. During the early stages of the SoC concept, the SoCs had generally been employing buses and point-to-point links [4]. Bus is a set of address, data and control lines that are shared by the connected PEs that contend among themselves to transfer data through the bus to other connected PEs. Figure 1.2 shows a shared system bus with eight PEs with four masters and four slaves. In Figure 1.2, the arbiter periodically checks the requests from the master interfaces, and grants bus access to the master with the highest priority. The bus allows for the master to send string of data without having to request for bus access again.

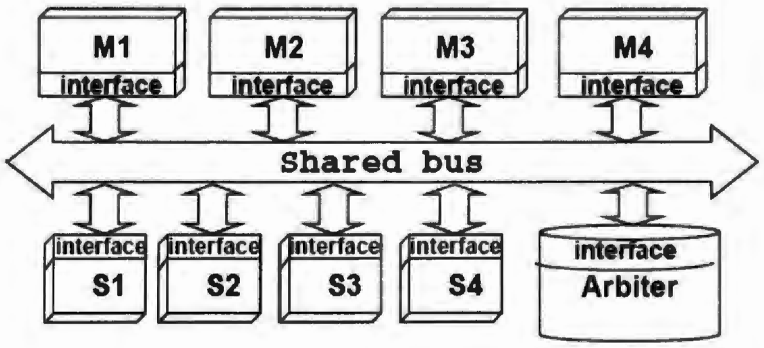


Figure 1.2. Shared system bus.

1.3. Network-on-Chip

As the number of PEs within an SoC increases, the complexity of connections between the PEs also increase. Point-to-point connection between the PEs is the most efficient connection [5], however it leads to inefficient use of resources, silicon area and diminished design flexibility. One cost factor when designing the system is measured by chip area and power dissipation.

The architecture of the bus has been improved to meet the requirements of high performance SoCs [6], however, its poor scalability limits the enhancement. Current state-of-the art bus architectures, such as the AMBA multi-layer, STBus, and SonicsMX enable the instantiation of multiple buses operating in parallel, thereby providing a crossbar architecture [7]. However, as all the PEs in the design need to be connected to the crossbar, such architecture is non-scalable for large number of PEs in the design.

The NoC architecture has been proposed to meet the communication needs and Quality of Service (QoS). QoS is associated primarily with latency and throughput. It is desired to lower the latency and increase the throughput to increase the performance of the SoC. A simple NoC connecting multiple PEs has been shown in Figure 1.3 [8].

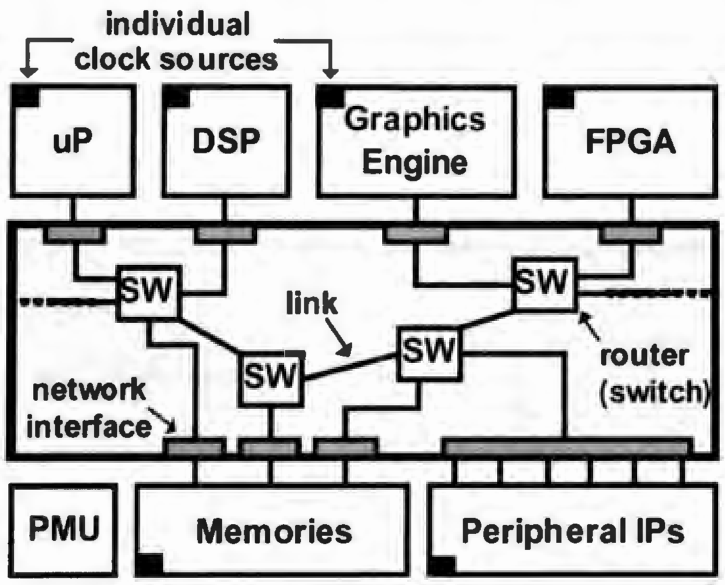


Figure 1.3. An example of a heterogeneous NoC architecture.

Buses have been vastly used in SoC due to their well understood concepts, compatibility with different processor interfaces, area taken on the chip and the zero latency after arbiter has granted control. The performance of the SoCs will be decided by the efficiency of the connections between PEs and customized modules [8]. Despite the advantages of the bus such as their simplicity to implement, their architecture will not meet the increased communication requirement because of the bandwidth bus shares with all the attached devices. Another disadvantage of the bus is that the clocking frequency of global wiring becomes tightly constrained by the electrical properties of deep sub micron processes [9].

NoC has emerged as a new paradigm for SoC communication infrastructure and as a replacement to buses and dedicated interconnections [10, 11]. The NoC architecture uses layered protocol and packet-switched networks with on chip routers, links and network interfaces with predetermined network topology.

1.4. Summary

Table 1.1 discusses the pros and cons of NoC and bus based communication infrastructure [12]. NoC achieves better scalability than bus and point-to-point with large numbers of PEs in an SoC [13]. Recent analysis [14] has shown linear increase in power and area used for NoC with addition of PEs to the SoC, whereas buses and point-to-point display super linear growth.

Table 1.1. Bus versus NoC arguments.

Bus Pros & Cons			NoC Pros & Cons
Electrical performance degrades with addition of units.	-	+	Only point-to-point one-way wires are used, for all network sizes, thus local performance is not degraded when scaling.
Bus timing is difficult in a deep sub micron process.	-	+	Network wires can be pipelined because links are point-to-point.
Bus arbitration can become a bottleneck. The arbitration delay grows with the number of masters.	-	+	Routing decisions are distributed, if the network protocol is made non-central.
The bus arbiter is instance-specific	-	+	The same router may be re instantiated, for all network sizes.
Bus testability is problematic and slow.	-	+	Locally placed dedicated built-in self test is fast and offers good test coverage.
Bandwidth is limited and shared by all units attached.	-	+	Aggregated bandwidth scales with the network size.
Bus latency is wire-speed once arbiter has granted control.	+	-	Internal network contention may cause a latency.
Any bus is almost directly compatible with most available IPs, including software running on CPUs.	+	-	Bus-oriented IPs need smart wrappers. Software needs clean synchronization in multiprocessor systems
The concepts are simple and well understood.	+	-	System designers need reeducation for new concepts

1.5. Thesis Outline

The proposed design is implementation of both bus and NoC based MP3 decoder on FPGA. The MP3 decoder's algorithm is discussed in Chapter 3. Bus and NoC communication will be discussed briefly in Sections 4 and 5 respectively. Implementation of the MP3 decoder on the FPGA chip using bus and NoC based communication is discussed in Section 6. Results of the implementation of both NoC and bus based MP3 decoder is reported in Chapter 7. Conclusion and future work is discussed in Chapter 8.

CHAPTER 2. STATE OF THE COMMUNICATION PARADIGMS FOR SOC DESIGNS

SoC is growing in complexity level and the number of independent applications on a single chip. There are a growing number of applications that require a system architecture to be scalable on the physical, architectural and functional level. Many applications have constraints on real-time performance, such as maximum time allowed for an application to be executing commands (e.g., MP3 decoder).

In the implementation of a multi-processor SoC, the bus architecture comes to the forefront. The performance of the system is not dependent only on the processors's speed but also on the bus's architecture, which may impact the system's performance. An efficient bus architecture and arbitration for reducing contention plays an important role in maximizing the performance of the system. AMBA [15] offers the simplest architecture compared to any other bus. AMBA and IBM CoreConnect [16] are two widely used on chip buses. AMBA and IBM CoreConnect both have a split bus architecture, with a low speed peripheral bus attached to the main high speed system bus through a bridge. Sonics MicroNetwork [17] is a time division multiple access based bus, which can be easily adapted to different needs while providing high bandwidth utilization. The architecture of the bus has been improved to meet the requirements of high performance SoCs [6], however, its inherently poor scalability limits the enhancement. Many NoC architectures have therefore been proposed to meet such demands. The advantages of spatial-reuse packet/wormhole switched networks were reported and explored in comparison with buses in [10, 18, 19].

It is claimed in [20] that different types of networks will be required for each application. The authors in [20] provide a step through methodology to provide a flexible network for an optimum solution to develop the communication infrastructure of a new system. Cost performance trade off is a major issue in NoC design and this constitutes the effectiveness of the NoC in a given design. There are five major key factors to determine the efficiency of NoC: silicon area, network saturation threshold, communication throughput, packet latency and power consumption [21]. The SPIN Micro Network [22, 23] was the first published attempt to solve the bandwidth bottleneck, when interconnecting a large number of PEs.

Growing system size with increasing performance variation has impacted globally synchronous operations. There have been NoCs presented that consider the synchronous performance issue such as Dally's NoC [18], ÆTHEREAL [24], XPIPES [25], NOSTRUM [26] with a global synchronous clock, which may be hard to implement with given time constraints [27]. Also, there have been some asynchronous NoCs presented such as CHAIN [28], MANGO [29], QNOC [30], ANOC [31]. Another drawback of globally synchronous operation is a large peak current at the clock edge leading to ground bounce and voltage drops, which in turn induce jitter in both clock and data, causing the system to operate in the unstable region. These drawbacks have lead to the concept of Globally Asynchronous Locally Synchronous (GALS), which suggests implementation of the SoC as synchronous islands that communicate asynchronously [32].

A drawback of GALS approach is the lack of design tools' availability along with complications that arise with ensuring QoS of the communication needs. Another approach that has been taken in [33, 34] has been a meso-synchronously clocked system, which employs a single clock across the entire system, but at arbitrary phases. Meso-synchronous NoC's power dissipation is much less than synchronous NoC, and Meso-synchronous NoCs are scalable due to the phase difference between the clock-phase of different regions.

There have been NoCs based on Time Division Multiplexing (TDM) [24]. TDM communication allows for transfer of two or more bitstreams simultaneously as sub channels in one communication channel. The ÆTHEREAL[24] NoC uses contention-free routing, or pipelined TDM circuit switching, to implement its guaranteed performance services. The focus of ÆTHEREAL NoC is to guarantee QoS specially ordered, lossless, uncorrupted data delivery, throughput and bounded latency. Such requirements of TDM NoC make the performance of NoC predictable, which can then accommodate applications in real-time requirements. TDM has the drawback of the connection latency being inversely proportional to the connection bandwidth.

An NoC emulation framework has been implemented onto an FPGA platform [35]. The authors in [35] cite utilization of the FPGA as an active element in the emulation control layer to speed up functional validation and to add flexibility to the NoC configuration exploration, instead of merely being the platform where the circuit is prototyped, as emulation is typically used. Ehliar et al. [36] have proposed an FPGA based NoC architecture similar to the NoC implemented in this project.

In [37], the authors employ the partial dynamic reconfiguration technology of the FPGAs in their proposed framework. The proposed partial and dynamic reconfigurable NoC can enhance the performance of the NoC by dynamically establishing or deleting a number of dedicated point-to-point connections between the routers. Another dynamic reconfigurable NoC is presented in [38], where the routers are controlled by some signals to prevent packets from queuing up in the internal buffer, which will decrease the size of the internal buffer. This NoC can be reconfigured to adjust the properties of the NoC such as routing scheme and buffer sizes at run-time. However, the changes dynamically made to NoC have to be already allocated for at design time.

2.1. Summary

During the early stages of SoC design, bus and point-to-point were vastly used as the communication infrastructures of the SoCs. Over the years, their poor scalability and slow performance rate have created a demand for a faster and more scalable communication paradigm such as NoC. Variations of NoC have been presented to meet demands of different applications with constraints on their performance implemented on an SoC.

CHAPTER 3. MP3 DECODER

MP3 is a codec formally known as Moving Pictures Experts Group (MPEG)-1 Audio Layer 3, and it is defined in the MPEG-1 standard. This standard defines three different audio codecs, where layer 1 is the simplest and has the worst compression ratio, and layer 3 is the most complex but has the highest compression ratio with the best audio quality per bit rate. The MP3 encoded files are made up of several “frames”, which are consecutive blocks of data. Each frame is consisted of two granules with two possible channels. Each granule contains 576 samples. While important for unpacking the bit stream, frames are not fundamental and cannot be decoded individually.

3.1. MP3 Decoder’s Algorithm

The MP3 decoder’s algorithm is made up of the following functions [39]:

1. **Find and Read Header**: The MP3 decoder has to align itself with the MP3 file bitstream, by identifying the MP3 frames within the MP3 file. This function finds and identifies series of information regarding the newly found frame.

2. **Get Side Information**: The MP3 decoder requires block of information for each channel in a granule to properly transform data into sample values. This function extracts the block of data needed, called *side information*.

3. **Get Scale Factors**: There are scale factor bands that span the frequency spectrum of the hearing bands of the human ear. Each scale factor band requires a unique factor for each channel in a granule, called *scale factor*. Get Scale Factors function extracts the scale factors.

4. **Huffman Decoder**: The MP3 encoder uses Huffman encoding technique to compress and reduce the size of the data. Huffman Decoder uses the reverse technique to decompress data using different Huffman tables.

5. **Sample Dequantization**: Huffman decoder's data are dequantized using the proper values of the scale factors to create a set of data called *sample values.*

6. **Stereo**: MP3 decoder supports four different channel modes. Stereo will properly adjust the dequantized samples based on the frame's channel mode. The adjusted samples are called *stereo samples.*

7. **Reorder**: Stereo samples that use the short time window setting must now be reordered.

8. **Alias Cancellation**: The MP3 decoder applies alias cancellation to stereo samples that use the long time window setting (long blocks) to compensate for the frequency overlap of the sub-band filter bank.

9. **Inverse Modified Discrete Cosine Transform (IMDCT)**: This function transforms each sub-band into the time domain.

10. **Frequency inversion**: After IMDCT function, every second sample in every second sub-band needs to be multiplied by -1 to correct for the frequency inversion of the sub-band filter bank.

11. **Sub-band Synthesis**: Finally, the 32 sub-bands are combined into time domain samples that cover the whole frequency spectrum. One sample is taken from each sub-band and transformed using a transform similar to the discrete cosine transform. The results are *pulse code modulation (PCM) samples*, which are calculated by means of a windowing operation. PCM values are the decoded audio samples for each frame.

3.2. Synchronization between the MP3 Decoder and the Bitstream

Any file with .mp3 extension indicates that the file is an MP3 file. The MP3 file may include some information about the audio properties of the .mp3 file such as the artist, author, name of the track and etc. Therefore, in order to start decoding any information, the decoder and the file must be aligned perfectly. The decoder and the MP3 file's bitstream are aligned by an entity called the *sync word.* This process is called *sync word detection.* The sync word is 12 consecutive 1 bits in a row with the first 8 bits in one byte, and the following 4 bits in the next byte. Decoding a frame starts after a successful sync word detection.

3.3. Layer and Side Information Block

Following a successful sync word detection, the next 20 bits represent the frame's layer information. These 20 bites are called*frame header bits.* Table 3.2 shows the relation between particular frame header bits in Figure 3.4 and the properties of the frame's layer.

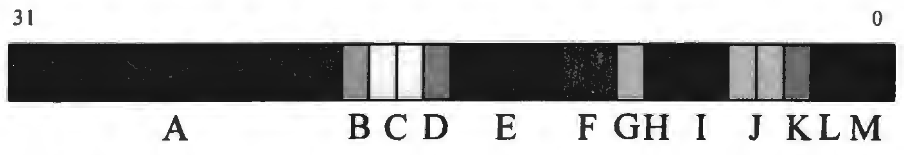


Figure 3.4. 32 bits of frame's layer info.

3.4. MP3 Decoding Block Diagram

The block diagram of the MP3 decoder is shown in Figure 3.5. Once synchronization was successful and side information block has been retrieved, the first loop will begin. The first loop will continue until both of the granules of the frame has been processed. The stereo will perform stereo decoding on two granules of the frame. Second loop will begin after stereo decoding is finished. After completion of the second loop, the PCM values are saved or streamed.

Table 3.2. Frame Header bits information.

Sign	Length	Position	Dec
A	12	(31-20)	Sync Word
B	1	(19)	MPEG Audio version ID.
C	2	(18-17)	Layer Description.
D	1	(16)	Protection bit.
E	4	(15-12)	Bit rate Index.
F	2	(11-10)	Sampling rate frequency index
G	1	(9)	Padding bit.
H	1	(8)	Private bit. This bit is only for informational purposes.
I	2	(7-6)	Channel Mode. If the channel mode bits are 00, the frame is stereo, if the bits are 01, the frame is joint stereo, if the bits are 10 the frame is Dual channel (2 mono channels)
J	2	(5-4)	Mode Extension.
K	1	(3)	MP3 file Copyright protection.
L	1	(2)	Originality of the MP3 file.
M	2	(1-0)	Emphasis.

3.5. Partitioning the MP3 Decoder's Components

There are total of 11 functions to the MP3 decoder's algorithm with an another function needed to handle the control of the MP3 file's bitstream and the operation of the functions in the proper manner, called **Manager**. Manager is very similar to the operating system used in [40]. For example, Manager must ensure Huffman Decoder is not accessing the MP3 file's bitstream prior to the Getting Scale Factors. Grouping some of the functions allows to simplify the MP3 decoder and reduce the need for more resources with consideration of implementing the MP3 decoder on an FPGA chip.

The Find and Reader Header function with the layer and side information block component are very much intertwined together and can be grouped together because their processes are sequentially grouped with following bitstreams. The new function is called

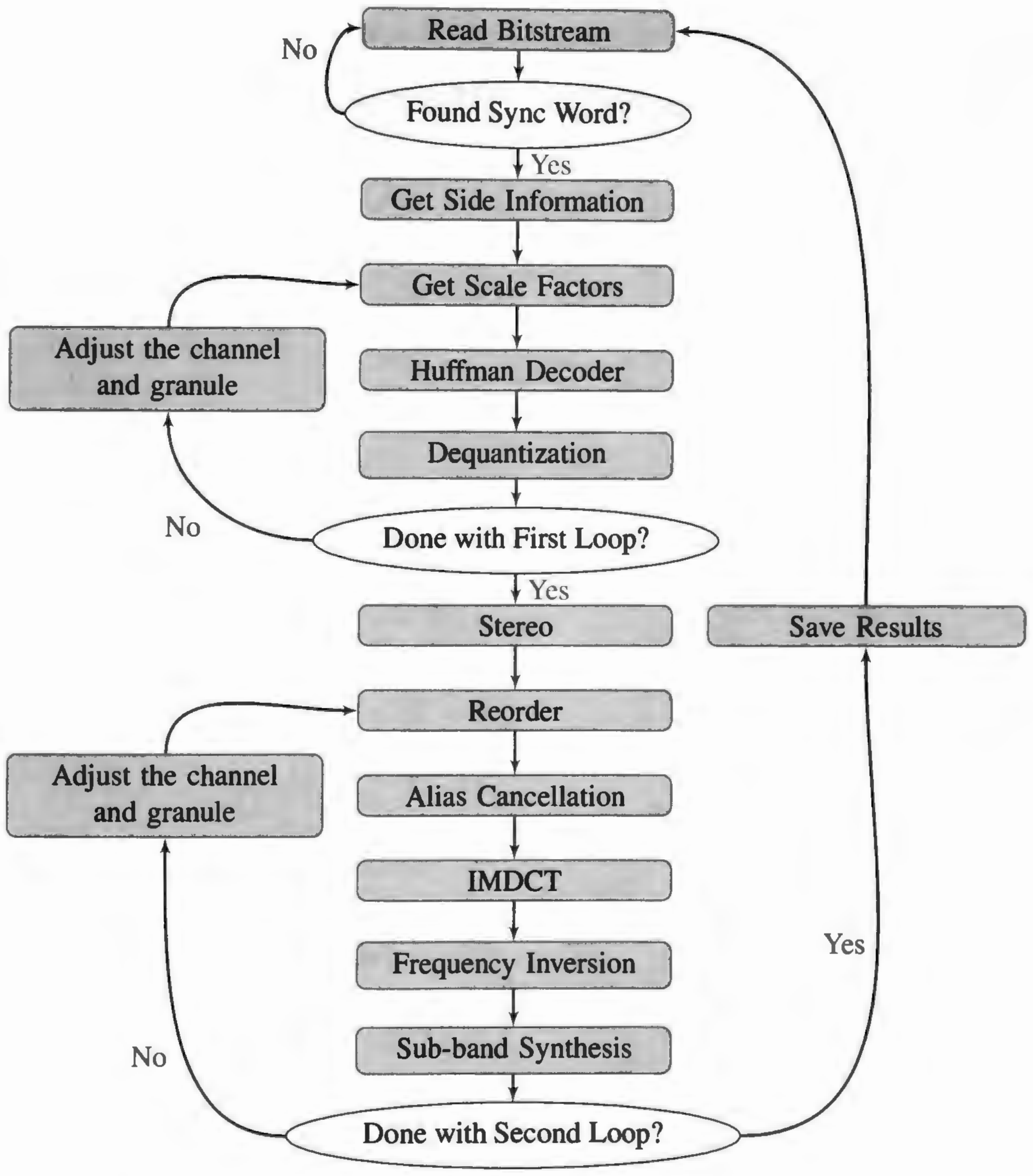


Figure 3.5. Block diagram of the MP3 decoder.

Sync. Frequency inversion is a very small task that can be done along with IMDCT. Grouping of Frequency inversion and IMDCT forms **Hybrid**. For a single channel in a granule, the decoder is either reordering or performing alias cancellation, therefore these two tasks can be combined. This function is called **Antialias**.

Stereo decoding, Huffman decoding, Dequantization, Sub-band synthesis and Getting Scale Factors will remain intact with the following names **Stereo**, **Huffman**, **Dequantization**, **Synthesis** and **Get Scale**.

CHAPTER 4. BUS BASED IMPLEMENTATION OF MP3 DECODER

4.1. Introduction

Different bus architecture have been discussed in [4]. The bus implemented in this design is a shared system bus. A system bus is composed of three separate components, Data Bus, Address Bus, Control Bus. Control bus contains an arbiter, which is a round-robin arbitration system, which allows the connected PEs to contend for equal rights to control the bus. Data and Address Bus contain the data and the address of the packet in the bus. Control Bus is used to control the flow of the packets between the PEs connected to the bus.

4.2. Architecture

Figure 4.6 shows high level abstraction of the implementation of the bus. Data bus is 16 bits, address bus is 8 bits and control bus is 2 bits to make a 26 bit bus. Each PE has three extra signal lines connected to the bus. First signal is called *Request flag*, which each PE uses to signal the arbiter for control of the bus. Once the arbiter grants any PE control of the bus, it will do so through the second signal called *Grant Control.* Also, the control line of the tri-state buffers used to allow the PE to insert data into the bus is active as shown in Figure 4.7. Third extra line participates in a flow of the data as an acknowledgment signal from the slave PE to the master PE.

4.3. Operation

Figure 4.8 shows a simple transfer of data between two PEs through the bus. Xilinx ISim was used to simulate and verify the transaction through the bus. During this simulation a PE located at port 2 will transfer data to PE located at port 1. The following steps take place in order for the transaction:

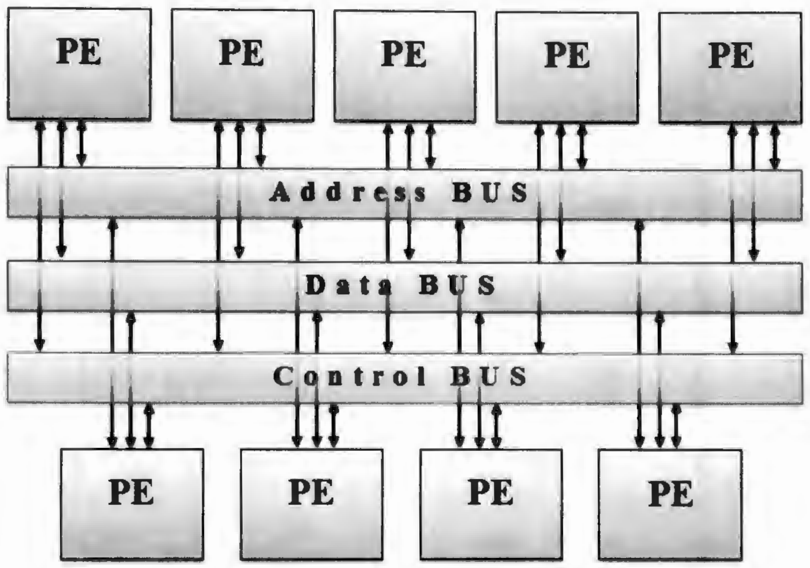


Figure 4.6. High level architecture of the bus implemented in the design.

1. Port 2 will raise its flag *PORT2_Req*, which is the request flag for the PE located at port 2.

2. Bus is not currently being used, this is done by checking the *Job_Req*. *Job_Req* is high when the bus is in use. Step 1 triggers the arbiter to determine if port 2 can control the bus.

3. Bus grants port 2 (*Grant_Access* equals to 1, which translates to port 2) the right to use the bus. The bus then lowers *Grant_PORT_Master[1]*. The bus will set the control line of the tri-states buffers used to allow port 2 to control the bus to active.

4. Once port 2 has noticed that it has been granted the right to control the bus, it will begin transferring data by setting data on *BUS_Data* and *BUS_Address*.

5. Port 2 will use the 4-phase handshake through the *BUS_Control* to control the flow of the data. *BUS_Address* is set to 0x10, which states that data is coming from port

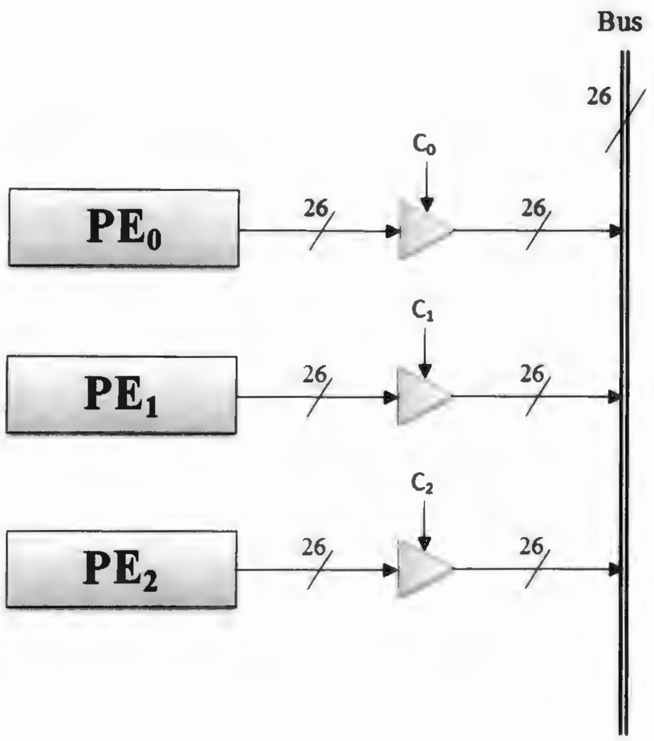


Figure 4.7. Tri-state switches used in the bus.

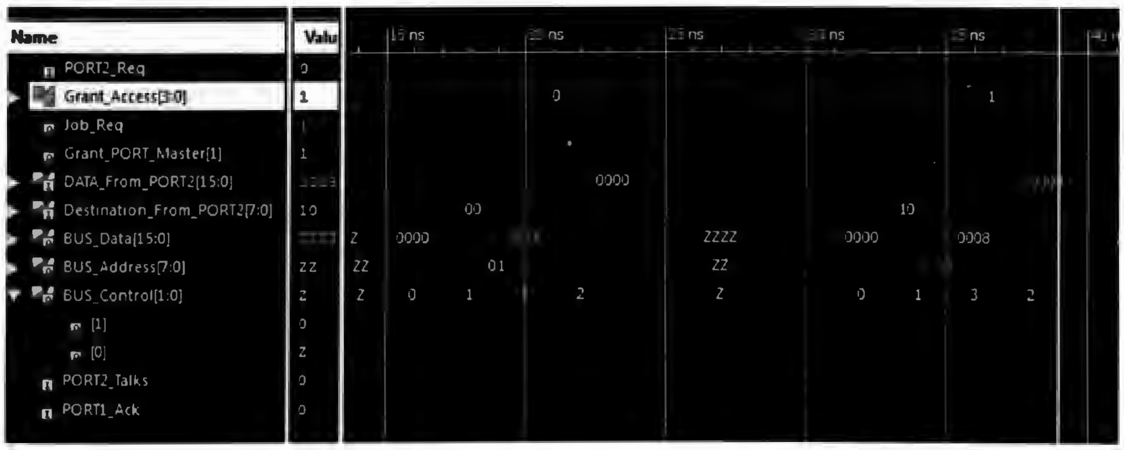


Figure 4.8. Transfer of data through the bus.

2, and is traveling to port 1. *BUS_Control[0]* is the request flag controlled by port 2, and the *BUS_Control[1]* is the acknowledgment signal from port 1.

6. Once port 2 has completed sending data, it will lower *PORT2_Req*, which releases the bus of its control.
7. Bus deactivates the control line of the tri-states for port 2 and the arbiter will allow other ports to control the bus.

CHAPTER 5. NETWORK-ON-CHIP

5.1. Introduction

The concept of NoC was introduced in Section 1.3. Main Components of the NoC are routers, network interface, and physical links between each router. NoC properties and the design methodology are explained in the following sections.

5.2. Architecture

NoC is a 3×3 homogeneous mesh network. Differences, advantages and requirements of each NoC topology have been explored in [41]. There are total of 9 routers. Each router has a location ID based on their location in the network.

Mesh topology has been known for its simplicity in the design phase and implementation. Due to the simple traffic within the MP3 decoder, and previously known destinations of each packet, mesh topology has been chosen. Mesh topology's downside is its long diameter which creates a larger communication latency. Application specific computation time may overshadow the longest latency in mesh topology but steps have to be taken to avoid long diameters between the origin and the destination of the data. Efficient mapping of the PEs on the NoC will help reduce the long latency. The mapping process is explained in Section 6.7.

High level abstraction of the implemented NoC is shown in Figure 5.9. The blue rectangular box in between the PE is the Network Interface (NI), which serves as the intermediate stage between a router and a PE.

In Figure 5.9, PE's location is identified by two numbers, where the first number is the x-axis, or horizontal axis position, and the second number is the y-axis, or the vertical axis position. The numbering starts from left to right for the x-axis and top to bottom for the y-axis. Physical links create the physical connection between the routers.

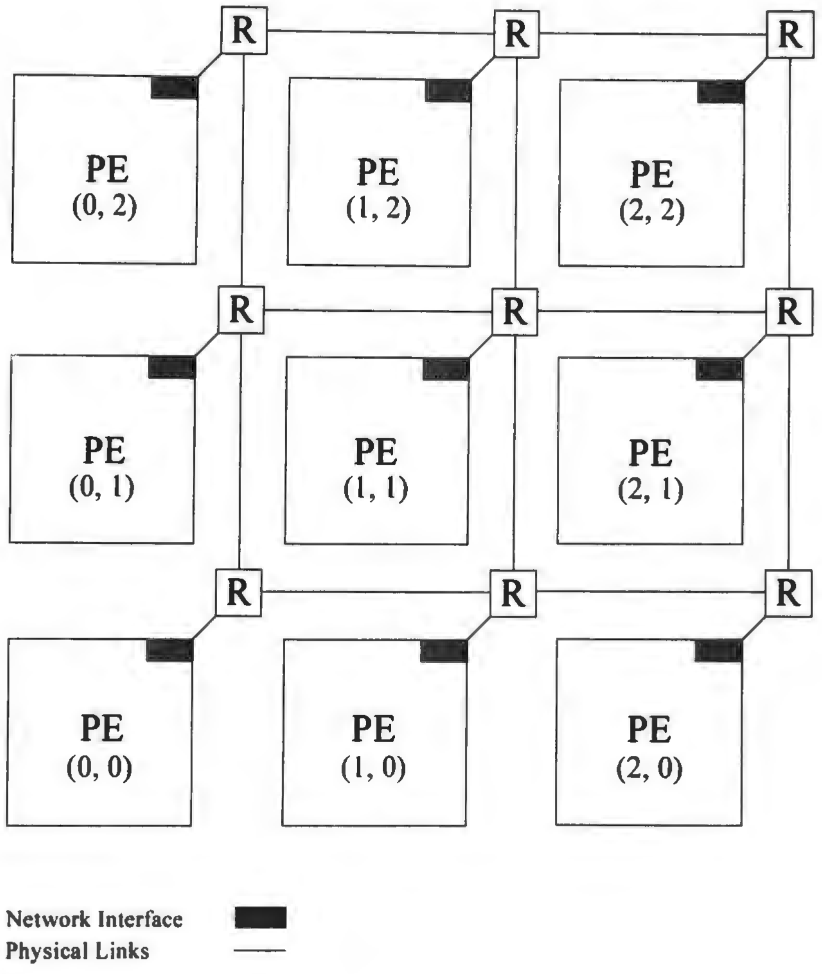


Figure 5.9. NoC Topology - 3 $\times$ 3 Homogeneous. R stands for router.

5.3. Router

The router is the main component of the NoC, which sends and receives packets from each port. Each router has five ports: north, east, south, west and PE. The names are a direct correlation of to where the ports are connected to. Each router has the following components: input buffers, switch box and an arbiter. Components of the routers are explored in the following sections.

The high level architecture of the router is shown in Figure 5.10. In a 3 $\times$ 3 homogenous network with mesh topology, some of the ports and components do not exist.

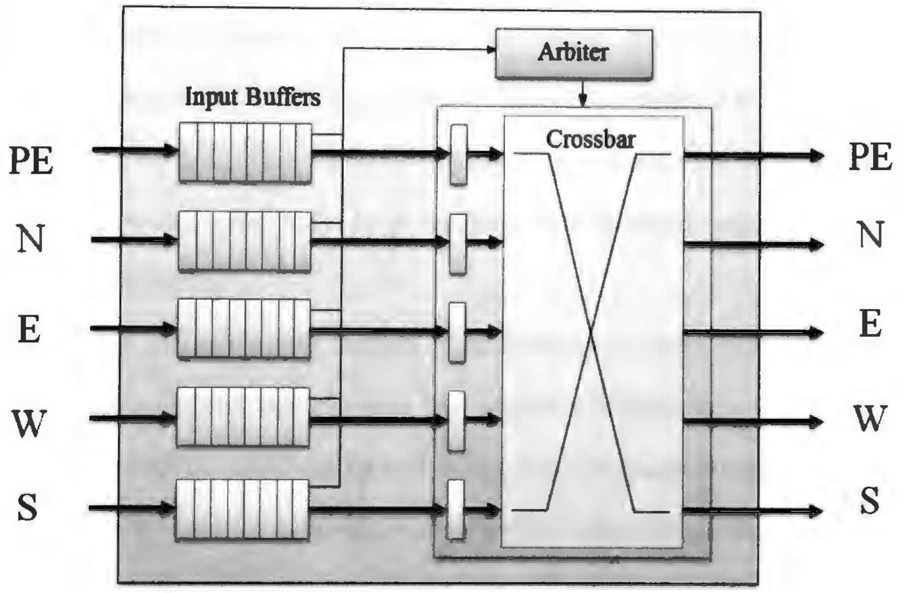


Figure 5.10. Architecture of the router implemented in the NoC.

5.3.1. Input Buffers

The input buffers are used to store data between stages of the NoC. The input buffers store data from the neighboring routers or the NIs. Once the input buffer contains an element, the input buffer flags the arbiter informing the availability of data in the input buffer. Input buffers have one channel, and can hold up to 8 elements.

The population of the input buffer is the number of elements present in the buffer. If the population is less than 8, then the buffer will set a flag as an indication of availability of the input buffer. Any storing component is required to check the availability flag before sending any data, this ensures that no data is mishandled. If the population of the buffer equals 8, then the buffer's availability flag is not set and the storing component will have to wait until the flag is set.

Input buffers are of a First In First Out (FIFO) system. Elements that arrive earlier will be sent first. One very general method is to order the elements in the buffer based on how early they arrived. If there is a transaction between the input buffer and another component, the method calls for re-ordering of the buffer. This method ensures index 0 is the next element to be sent at all times. Once the element in index 0 is sent, element 1 will then be shifted to index 0. This method is exhaustive and not very efficient. It also may take some clock cycles for the buffer to be ready to store and send data, as a result a latency is introduced to the NoC.

To overcome the exhaustive method of re-ordering, circular buffering is used. Circular buffering is a method that requires two registers to indicate which index the next incoming element will be stored and which index holds the next element that will be sent next. For example, if there are 4 elements stored in the buffer, the storage index for the next incoming element is 4. If the buffer has already sent two of the four elements, then the value of the sending index will be 2. This method does not require shifting of elements in the buffer, and will not create any extra delay by moving data around in the buffer internally. Also, circular buffering uses the dedicated RAM components of the FPGA chip.

5.3.2. Arbiter

The arbiter is the component that ensures all the buffers will be given equals right to send data through the switch box. The arbiter uses a round-robin ordering system. For example, if the input buffer for the PE has data and has the right over all the other input buffers, then it will be given the right to use the switch box. Once the input buffer for the PE has successfully used the switch box, it will then have the least priority between all the PEs.

5.3.3. Switch Box

The switch box or also known as the cross bar, is simply where the data from the input buffers are routed to the neighboring routers or the NIs. Once an input buffer has a population of one or more, it will flag the arbiter. Once the buffer is granted the right to use the switch box, the switch box will then retrieve its data and store it in a 1 element long register for each input buffer. Each element contains the destination of the element. In order to determine and check the availability of the destination, the element from the input buffer needs to be retrieved first and stored into a register. Once the element is stored into the temporary register, the destination of the element is determined. The switch box will then check the availability of the destination. If the destination is available, then the switch box will route the element to the destination. However if the destination is unavailable, the switch box will check again within the next clock. Once the destination becomes available, the element can be routed and allow for other elements from that particular input buffer to be routed.

The temporary registers allow for other input buffers to use the switch box. For example, suppose one router has elements coming in from the NI and traveling west, and one element coming from the south and traveling north. If west is busy, and the PE has the higher priority, the switch box will not have to wait for west to become available in order for the south elements to travel north. However, elements in the PE traveling somewhere else besides west will have to wait for the elements traveling west to exit the router before they can be routed. This problem may be solved by adding more virtual channels to the input buffers. Also, if output buffers are added to the router, only the elements traveling to the busy destinations will have to wait, however this is not a guaranteed solution.

5.4. Routing Algorithms

Routing algorithms can be either deterministic or adaptive [41]. For the simple communication needs of the MP3 decoder, the routing algorithm is deterministic. Odd-Even or XY routing algorithms are examples of deterministic algorithms. The XY routing algorithm is used in the NoC. Elements will travel to the east or west (x-axis) until the elements reach a point where their current position in the network is equal to their destination's x-axis position. Elements will then complete their y-axis route until they reach the destination. For example, suppose an element starts at position (1, 0) and will travel to its destination at (2, 2). The packet will travel from router (1, 0) to (2, 0) and it is now at the same column or the same x-axis position with the destination. The packet will then travel to (2, 1) and (2, 2) afterwards.

5.5. Packet Structure

Packet is an organized structure of data, as a message from one PE to another PE. The packet is made up of flits (flow control digits). Usually a packet can contain a head, body and tail flit, where the head and tail indicate the beginning and the end of the packet along with other information such as length of the packet or priority level which may influence the NoC to treat the packet differently than others. Since the NoC will not reserve any buffer slots for any packet and is independent of the packet's priority or information, there is not a need for head and tail flits. Body flit contains data and the destination of the packet. Alternatively, the body flit can only contain the data and leave the destination and origin location to the head and tail flits. However, this method requires extra logic where as the routers will have to have extra logic such as a counter to account for the size of the packet, and buffers. For simplicity, the packet will only contain body flits that will have the data along with the destination and origin of the packet.

A single flit's architecture is shown in Figure 5.11. Each flit is 26 bits long, the first two bits are the flit type bits which dictate whether the flit is a head, body or tail. The next four bits called the destination bits, inform the router location of flit's destination. The next four bits called the origin bits and have no effect in the operation of the NoC, but it is informative for the PEs. The next 16 bits are the data bits and contain the data portion of the packet.

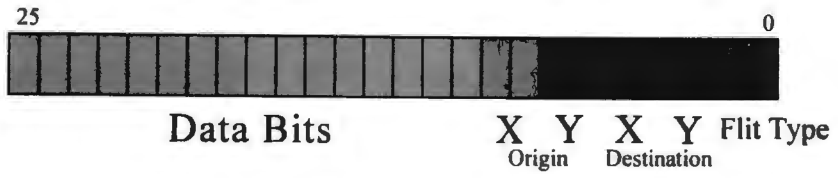


Figure 5.11. Body flit structure.

The size of the flit dictates the width of the input buffers and the physical links. It also influences operation of the PEs. For example, some PEs may require to send a data that is 32 bits long, but with the current settings of the packet, they are forced to send the 32 bit long in a two flit long message. There is a large amount of data transferred between PEs which are 8 and 32 bits long. 16 bits were presumably used for data width to account for both 8 and 32 bits long packets.

5.6. Network Interface

The NI or sometimes regarded as wrapper is the component between the router and the PE. The NI will wrap and unwrap the data bits into and out of a packet. The NI is regarded as the PE port to router in the NoC.

5.7. Simulation Example of Data Transfer in the NoC Based MP3 Decoder

Figure 5.12 shows example of data sent from a PE in the MP3 decoder to an another PE. All the data shown is in hexadecimal format. Data is originated at the location (1, 0) where it translates to binary value of 0100 for origin bits of the packet and is traveling to PE at location (1, 1) which translates to binary value of 0101 for the destination bits of the packet.

1. During the first step, the PE at location (1, 0) checks the wrapper's availability flag.

If the flag is set to high, then the wrapper has open slots. PE will set the data on *Output* and raise the *PE_Talks* flag.

2. The wrapper or the NI for the PE at location (1, 0) will be notified of new data by *PE_Talks* flag. The wrapper will then wrap the data on *Output_From_PE* register and store the packet into the buffer. Along with the data, there is another four bit register controlled by the PE to indicate where the data is traveling to.

3. The wrapper looks to send the newly stored packet in the buffer to the NoC. The wrapper starts by reading the NoC's availability flag. If the NoC is available, the wrapper will set the packet's data on the *DATA_To_NoC* and raise the *Tell_NoC* flag.

4. The PE input buffer in router at location (1, 0) will receive and store the data on *PE_1_0_To_R_1_0* into the buffer. The PE input buffer will then notify the arbiter in router at location (1, 0).

5. The arbiter in the router at location (1, 0) grants the switch box the permission to retrieve the data from the PE input buffer. The switch box determines that the data is traveling north. The switch box will then prompt the south input buffer of the router at location (1, 1) by setting the flag *Tell_N* high.

6. The south input buffer in the router at location (1, 1) stores the data on the *DATA_From_S* into the buffer. The south input buffer in the router at location (1, 1) now has data available and will notify the arbiter in the router at location (1, 1).

7. The arbiter in the router at location (1, 1) grants the switch box to retrieve the data from the south input buffer. Once the data is retrieved, it can be determined where the data is traveling to. The switch box will then prompt the wrapper at location (1, 1) by setting the flag *Tell_PE* high.

8. The wrapper at location (1, 1) position will unwrap the data bits and the origin bits of the packet and store them into a buffer. The buffer in the wrapper at location (1, 1) contains data for the PE, and notifies the PE at (1, 1) position by raising *Alert_PE* flag and setting the data on *Input_To_PE.*

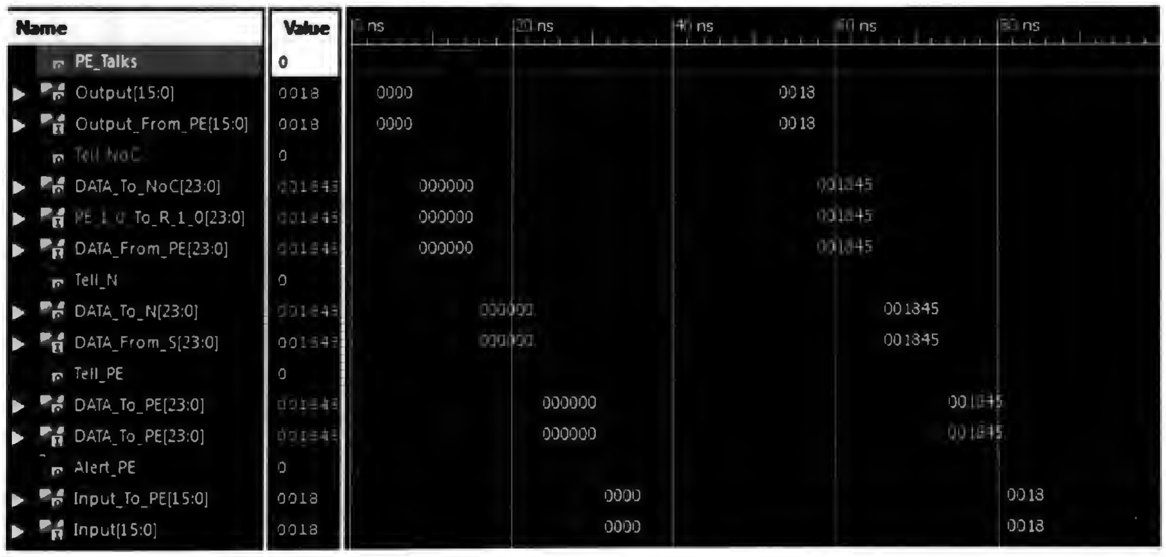

Name
Value
0 ns
20 ns
40 ns
60 ns
80 ns
PE_Talks 0
Output[15:0] 0018 0000 0018
Output_From_PE[15:0] 0018 0000 0018
Tell_NoC 0
DATA_To_NoC[23:0] 001845 000000 001845
PE_1_0_To_R_1_0[23:0] 001845 000000 001845
DATA_From_PE[23:0] 001845 000000 001845
Tell_N 0
DATA_To_N[23:0] 001845 000000 001845
DATA_From_S[23:0] 001845 000000 001845
Tell_PE 0
DATA_To_PE[23:0] 001845 000000 001845
DATA_To_PE[23:0] 001845 000000 001845
Alert_PE 0
Input_To_PE[15:0] 0018 0000 0018
Input[15:0] 0018 0000 0018

Figure 5.12. Simulation example of data transferred between two of the PEs in the NoC based MP3 decoder.

CHAPTER 6. IMPLEMENTATION OF THE MP3 DECODER ON FPGA

Previous section introduced the high level of abstraction of the MP3 decoder. The MP3 decoder is partitioned into nine blocks implemented on different PEs. Many MP3 decoder open sources are publicly available. One open source MP3 decoder library has been used as a reference to develop HDL code for each PE of the MP3 decoder, as explained in Section 6.1. While developing HDL sources, I realized that there are factors that must be considered and are different from the open source library. For example, certain data types are required by multiple PEs explained in Section 6.2. HDL code is written to take advantage of the platform such as parallel processing, which is explained in Section6.3. Once the HDL sources have been written, I realized that there are common aspects in the both NoC and Bus based implementations of the MP3 decoder. These common aspects include floating point representation of data, parallel processing, interface between the FPGA and a computer host, audio processing and some common FPGA requirements. However, mapping the MP3 decoder PEs is different and will be discussed.

6.1. Development of HDL Sources for Each PE

The Fraunhofer Institute has been the main developer of MPEG audio Layer-3 and the MP3 standard that has been approved is mainly based on their work. There are many open sources available written in different programming languages. The Fraunhofer Institute has provided an open source library written in C that decodes any MP3 file and outputs the result in a Audio Interchange File Format (AIFF). In order to gain more insight about the operation of the MP3 decoder I wrote a Python version based on the open source library provided by Fraunhofer Institute. The replication allowed me to gain more insight in the relation between each function, the block diagram, and the operation of the MP3 decoder.

The Python module does not assume the MP3 decoder is implemented on the FPGA .

Hence, developing the HDL source will be a separate effort. The differences are explained in Section 6.1.1.

6.1.1. Differences in Transfer of Data Between the PEs in Python and Actual HDL Implementation

Even though the Python code provides a very easy understanding of the MP3 decoder, there are differences, which change the writing the HDL sources particularly Verilog. The NoC and bus based verilog codes create an isolation between each PE, which means no register or data is shared between PEs unless sent to one another through the NoC or bus. In Python a list or a buffer can be shared throughout the whole code or certain classes containing data such as a side information block. Each processor has access to the memory elements of the buffer, which is not true in the case for the proposed design. Figure 6.13 shows that the **Side Info** function will write data to a shared buffer between multiple functions. This shared buffer is accessed by seven different functions. The proposed implementation does not provide a global buffer that can be accessed by multiple PEs. It is possible to designate a PE to be a memory module connected to the NoC and bus in the design. Adding a memory module is not an effective solution since PEs can contain large blocks of memory which can store data for their own purposes, this is further explained in Section 6.6. Also, with the addition of a memory module, the traffic in the design will increase. The number of PEs will also decrease from 9 to 8 which then requires for two more PEs to be grouped together.

The option of having a global buffer may be more efficient in different applications. For example, for an application where a lot of data is required by multiple PEs, it will be required that each PE to contain a large memory to fit that data. This replication of the memory in each PE will then waste silicon area, however, the design would be more efficient if one global buffer existed for each PE.

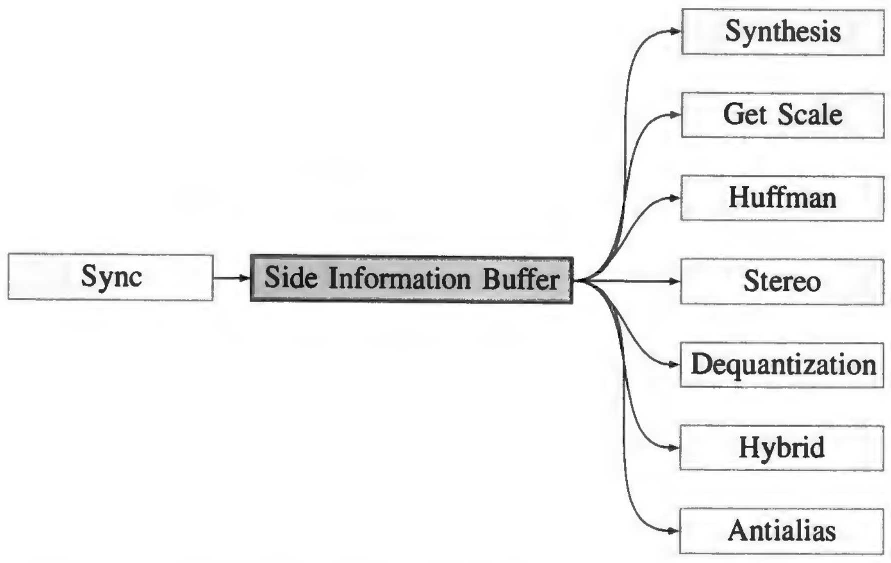


Figure 6.13. Storing and reading side information by different PEs in the MP3 decoder in the Python.

My solution is to send the data to the PEs directly. For example, the PEs will have to save the "side information" block locally. Figure 6.14 shows the different processes that handle side information. To reduce congestion in the NoC and bus and to reduce the size of local buffers, **Sync** block will send only the required part of the side information to rest of the PEs. For example, *global gain value* used for dequantizing the Huffman decoded data for each channel is a subset of side information block. The only PE that requires global gain is the Dequantization block. Therefore, it is unnecessary to send the complete side information block to all the PEs that need subsets of it. This method reduces buffer size in the PEs and lowers the traffic in the design.

6.2. Floating Point Representation

Most of the PE's input and output data are floating point real numbers. Antialias, Dequantiztion, Hybrid, Synthesis and Stereo are the PEs that compute multiplications and/or additions of floating point data.

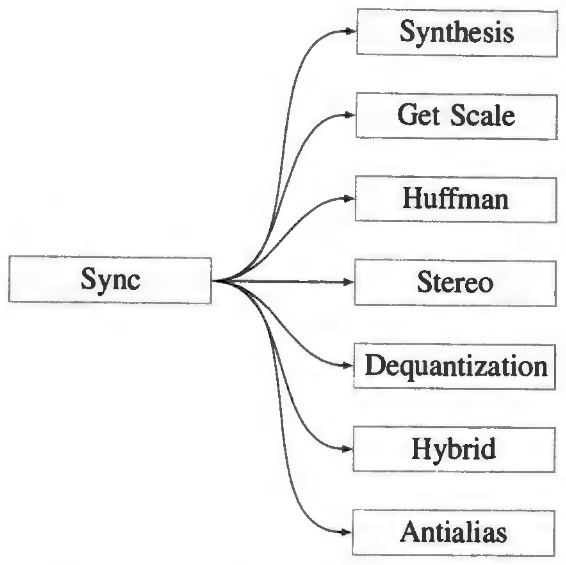


Figure 6.14. Sync sends the side information block directly to the other PEs.

6.2.1. Floating Point Format

Floating point numbers are represented in IEEE-754 format [42]. The single precision format uses 32 bits and it has one sign bit, 8 bits for the exponent width, and 23 bits for the significant precision. Figure 6.15 shows the bits in a single precision representation. If the width is increased, the accuracy of the representation increases, however number of the logic and flip flops will rise with a more accurate floating point representation.

The value of a real number represented in floating point in IEEE-754 formation is given by equation 6.1, where *sb* is the sign bit value, *eb* is the exponent bits value and b_i

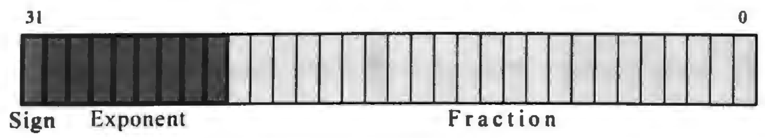


Figure 6.15. IEEE 754 single precision standard.

represents the bit value of ith bit of the floating point.

$$(-1)^{sb} * (2^{eb-127}) * (1 + \sum_{i=0}^{22} \frac{b_i}{2^{(23-i)}}) \tag{6.1}$$

6.2.2. Conversion from Fixed Point to Floating Point

At some stage during the decoding procedure, the decoded value needs to be translated from fixed point to the floating point representation. The conversion happens in Dequantization where multiplication of series of numbers takes place. To dequantize numbers, a constant value over a single channel in a single granule is used as a base value. During the operation of Dequantization, different numbers will be multiplied to the base value. At the end of the operation of Dequantization, value of the each 18 Huffman decoded value for each sub-band is raised to the power of $\frac{4}{3}$. Assume G_g to be the global gain value from side information block for a single channel in a single granule, then base global gain value is given by equation 6.2

$$2^{(\frac{G_g-210}{4})} \tag{6.2}$$

It is obvious that getting the floating point representation of the base global gain may be lengthy and may require complicated logic. Since the base is two, a simple shift arithmetic could solve this issue. Furthermore, there is a division by four which is not an integer division, therefore an exact value of the exponent is needed. Hence, an arithmetic shift will not work. A subtraction is needed along with division to compute the exponent value of the global gain base value. Beyond that, it is necessary to do a power operation to compute the result. As a last step, an IP core is required to translate the fixed point, or any other representation of the result, to the floating point representation. This process can introduce a large delay and also be wasting resources available.

I addressed this issue by using a look up table where as for each G_g, the floating point representation of the base global gain value is stored. Since the global gain value from the side information is 8 bits long, a look up table of 256 $\times$ 32 bits table is required. This approach eliminates the any need for a non integer division and a power arithmetic but still requires a lot of of resources. The size of the look up table is 8192 bits. Look up table solutions seem to take a large amount of resources and therefore increase the size of the design. The FPGA chip may not contain enough logic space for the design with the current size of the look up table. Therefore, the look up table solution fine tuned to decrease its size. For example, patterns can be detected which, can be explored to lower the size of the look up table.

Assume G_n to be the G_g subtracted by 210. Table 6.3 show the hexadecimal floating point representation of the base global gain value in the IEEE-754 single precision format.
Table 6.3. Look up table for floating point representation of the global gain.

G_n	Global Gain	Floating Point Representation
...	...	...
0	1	0x3F800000
1	0.840896415	0x3F5744FD
2	0.707106781	0x3F3504F3
3	0.594603558	0x3F1837F0
4	0.5	0x3F000000
5	0.420448208	0x3ED744FD
6	0.353553391	0x3EB504F3
7	0.297301779	0x3E9837F0
8	0.25	0x3E800000
9	0.210224104	0x3E5744FD
10	0.176776695	0x3E3504F3
11	0.148650889	0x3E1837F0
12	0.125	0x3E000000
13	0.105112052	0x3DD744FD
...	...	...

Pattern 1: It is not possible to get a base global gain value which is negative, hence, bit 31 or the sign bit will always be 0. This reduces the look up table to from 256 $\times$ 32 bits

to 256 $\times$ 31 bits.

Pattern 2: Bits 23 to 0 will always be one of the eight options. Table 6.4 shows the eight options and the relationship between G_n and bits 23 to 0.

Furthermore, the pattern across bits 23 to 0 reduces the table to be 256 $\times$ 7 bits plus an 8 $\times$ 23 bit look up table. The size of the table is therefore decreased from 8192 to 1976 bits.

Pattern 3: Another pattern can be recognized in Table 6.3. Bits 30 to 24 seem to only change for every 8 G_n. Table 6.5 displays the relationship between G_n and the values of bits 30 to 24.

It can be noted that bits 30 to 24 are equal to $63 - \frac{n+3}{8}$. The division is in integer division so no special IP core is needed to implement it. Hence, the 256x7 bit look up table is eliminated, and the final size of the look up table is now 184 bits, which has been in this way decreased by 98%.

IEEE 754 single precision format allows for the numbers that are the result of integers divided by factors of two exponents to base power of two to be translated with a small look up table for certain bits and a simple equation for the rest.

This technique is used for the Dequantization procedure since the values that are required to be shifted to the IEEE 754 format are all created by raising number two to a linear function of subsets of the side information block. This technique requires no additional IP core for the shift to floating point representation and only a small sized look up table and a simple equation is required.

The values that are decoded by Huffman block do not follow the same order as all the other values in Dequantization. The values from Huffman are raised to the power of $\frac{4}{3}$. A semi-large look up table was implemented to translate the decoded data to IEEE 754 single precision. There are other methods involving around implementation of the power arithmetic along with optimization approaches in [39].

Table 6.4. Bits 23 to 0 of the floating point representation with relationship to G_n.

G_n mod 8	bits 23 to 0
0	0x800000
1	0x5744FD
2	0x3504F3
3	0x1837F0
4	0x000000
5	0xD744FD
6	0xB504F3
7	0x9873F0

Table 6.5. Bits 30 to 24 of the floating point representation with relationship to G_n.

G_n	bits 30 to 24
...	...
-3,-2,...,3,4	63
5,6,...,11,12	62
13,14,...,19,20	61
...	...

6.2.3. Addition and Multiplication

To do addition, subtraction or multiplication on the floating point numbers, LogiCORE Floating-Point IP core from Xilinx tools [43] was used to create a 32 $\times$ 32 bit multiplier and adders.

6.2.4. Resource Usage of Multipliers and Adders

If the multiplier or adder uses DSP48E slices, then the maximum latency and number of flip flops and look up tables needed to implement the multiplier or the adder are reduced. However, the number of DSP48E slices is limited. A DSP48E slice is a digital signal processing logic element included on certain FPGA device families that allows designers to implement multiple slower operations in a single DSP48E slice using time-multiplexing methods. Figure 6.16 shows the relationship between latency and speed, flip flops and LUTs if DSP48E slice is fully used for the multipliers. Figure 6.17 shows the relationship if DSP48E slices are not used in creating the multiplier.

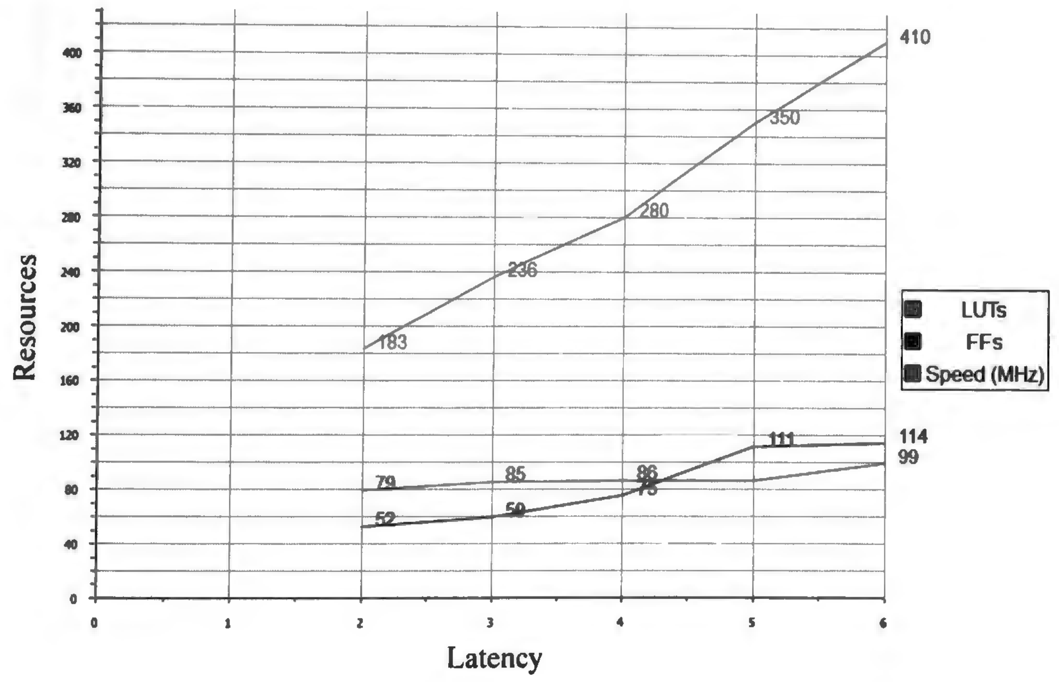


Figure 6.16. 32 Bit floating point multiplier resource estimation using DSP48E slices.

I conclude that it is better to use DSP48E slices if possible, since the number of LUTs and flip flops used will decrease and the maximum clock frequency will be higher than not using DSP48E slices.

6.2.5. Conversion From Float Point to Fixed Point

Sub-band synthesis output 576 of 32 bit of data for each channel in a granule. Each one of the 32 bit needs to be shifted from IEEE 754 floating representation to a 20 bit fixed point. The DAC within driver requires data to be 20 integer bits and zero fraction bits in the fixed representation of the PCM data. The process of this shift is done through

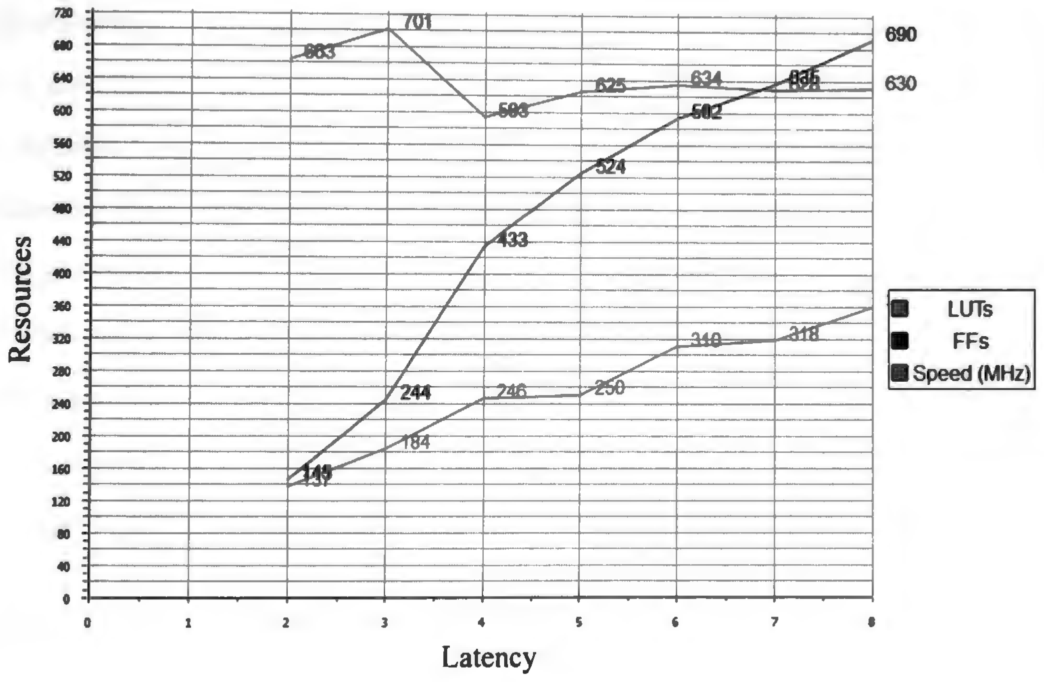


Figure 6.17. 32 Bit floating point multiplier resource estimation without using DSP48E slices.

LogiCORE Floating-Point IP core from Xilinx tools [43]. Sub-band Synthesis core will shift the data into this pipelined core and the audio driver will store the results into a FIFO buffer. Sub-band Synthesis core could also send the floating points to an off chip audio decoder or media storage where it may need to be shifted back to fixed point value.

6.3. Parallel Processing

Parallel processing is the ability to process multiple operations or tasks simultaneously. Such ability can speed up the execution of an application. For example, Huffman block processes 32 sub-bands, each containing 18 samples, creating a large output called Huffman Decoded Table (HDT). HDT is then treated as an input by Dequantization. Dequantization processes HDT one sub-band at a time and creates a large output of its own called Dequantized Samples Table (DST). DST is then treated as an input by Stereo. For any sub-band that Dequantization is processing, Dequantization does not require any data from any other sub-band, hence creating an opportunity of employing parallelism. Instead of Huffman computing complete HDT, Huffman can decode one sub-band at a time before processing again. Huffman can then proceed to send the 18 samples of that particular sub-band to Dequantization. Once the transaction of sending the 18 samples of the particular sub-band to the PE is finished, Huffman proceeds to decode the next sub-band. Meanwhile, Dequantization receives the 18 samples of that particular sub-band and starts dequantizing the 18 samples. Dequantization then sends the 18 dequantized samples of that particular sub-band to Stereo. Much like Huffman, once Dequantization is done sending the row of data, it can begin to process the next sub-band if the 18 Huffman decoded samples of that next sub-band is available.

Parallel processing can be found in three different solutions. Assume t_h to be the time it takes for Huffman to decode 18 samples in a single sub-band, and t_d to be the time for Dequantization to dequantize 18 samples of data in a single sub-band, and t_n to be the time for a 18 samples elements to travel from one PE to another.

Solution One: If t_h is less than t_d, then Huffman is not be able to decode 18 samples in a single sub-band until the Dequantization has finished dequantizing with the previous sub-band. Dequantization dictates when Huffman can start. Figure 6.18 shows the behavior of this case. The total time it takes for the 32 sub-band in one channel to be dequantized in this solution is $32 * t_d + 64 * t_n$.

Solution Two: If t_d is less than t_h, then Huffman does not depend on Dequantization. As soon as Huffman has decoded 18 samples of data in one sub-band, Huffman can proceed to sub-band and produce 18 elements. The total time it takes for the 32 sub-band in one channel to be dequantized in case two is $32 * t_h + 64 * t_n$.

Solution Three: If Dequantization is unable to empty its output buffer by sending the 18 elements to Stereo, then both Huffman and Dequantization must wait for the previous

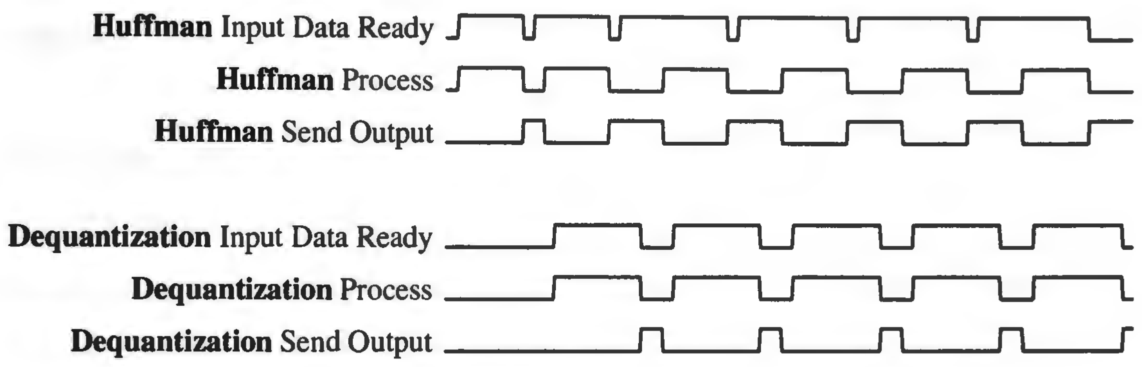

Huffman Input Data Ready
Huffman Process
Huffman Send Output
Dequantization Input Data Ready
Dequantization Process
Dequantization Send Output

Figure 6.18. Parallel Processing between Huffman and Dequantization.

PEs in the application to finish their task. The total time it takes for the 32 sub-band in one channel to be dequantized depends on other factors. If the network is congested, it can be argued that parallelism may not be useful. If Dequantization stays idle until the next PE allow it to send data, it is then preferred for dequantization to complete dequantizing complete by increasing the buffers in Dequantization and Huffman.

It should be noted that if parallel processing was not employed, the time for 32 sub-bands to be dequantized would be $32 * t_h + 32 * t_d + 32 * t_n$, which is much larger than any of the discussed cases earlier.

The faster the data gets processed by the decoder, the faster the response of the decoder is. If parallel processing is not employed between Huffman and Dequantization, then in order for Dequantization to begin its process, it needs to completely receive the complete HDT. The HDT which is stored in a $32 \times 18 \times 32$ bits buffer has to empty its data before allowing Huffman to start processing again. Another buffer with the same size has to exist to store the data for Dequantization since there are no shared buffer or registers between different PEs. This method seems to employ a three separate 18,432 bits long buffer, which is very large. However with parallel processing, we need only three separate

$1 \times 18 \times 32$ buffers which are 576 bit long and also $\frac{1}{32}$ the size of the serial processing method.

6.4. Interface Between FPGA and the Host Computer

The MP3 decoder requires data from the host computer to decode. Also, the MP3 decoder could send the data back to the host computer instead of playing it directly through the speakers. The decoded data is the audio data in an AIFF file. Standard AIFF is a leading format (along with SDII and WAV) used by professional-level audio and video applications, and unlike the better-known lossy MP3 format, it is non-compressed (which aids rapid streaming of multiple audio files from disk to the application), and lossless.

In my design setup, the connection is through a virtual serial port that uses USB connection on the computer side and serial communication port on the FPGA development board side. Simple implementation of a receiver and transmitter on FPGA was the prominent reason to use the serial communication. Another reason for using serial communication is the easy implementation of sending and handling data in the computer side by using modules precompiled to handle the adversities of serial communication. However, this approach has the disadvantage that it cannot handle high bandwidths.

6.4.1. Virtual Serial Port

With serial ports not being used in the computers anymore, one way to establish a serial communication with another core is to emulate the serial communication using hardware kit and software on the computer side. Virtual serial ports fully emulate real ones, so that an application communicating with a virtual port never suspects the difference. Moreover, using virtual ports is often more convenient than using real ones since virtual serial port connection offers better stability.

6.4.2. Communication Wrapper

Just like the Network Interface in the Router of the NoC, an interface module or a wrapper is needed between the MP3 decoder and the host, called Communication Wrapper. Communication Wrapper is responsible for wrapping and unwrapping data in a different format than that used in the Network Interface. Each time the host or the Communication Wrapper communicate with one another, each party sends a single byte at a time.

A header format is needed so both of the devices are aware of what type of message they are receiving. The header format has four bytes; the first three bytes represent the length of the message. Since only 8 bits can be transferred at a time, the range is limited to range of 0 to 255. Some of the required data by the MP3 decoder may exceed 256 bytes, like for example the length of the incoming batch of MP3file. It is known that the length cannot exceed 70,000 bytes which is the dedicated RAM's size to hold the MP3 file. Therefore, the first three bytes will determine the length of the packet.

The fourth byte is the message type. Depending on the previously agreed byte value between the both parties, the type of message or command can be determined.

Let us assume the host is sending 25 bytes of data to the MP3 decoder, the contents of the packet is described in Table 6.6. The first four bytes indicate that the PC is sending 29 bytes and 25 bytes of the bitstream. The communication wrapper stores the 25 data bytes into the RAM.

However since the Host is unaware how many bytes are available in the RAM, it will send a 4 byte message that prompts the MP3 decoder to send back a packet containing the number of free slots in the RAM.

6.4.3. Host Command Interface

It is required that a command interface exists on the host that will interpret the commands from both the user of the MP3 decoder and the MP3 decoder itself. Once the

Table 6.6. 25 bytes of bitstream is sent from the host computer to the MP3 decoder.

Byte counter	Decimal value	Description
0	0	low byte of the length
1	0	middle byte of the length
2	25	high byte of the value
3	55	predefined value for MP3 file contents
4	bitstream byte 0	byte 0 of the bitstream
4+n	...	byte n of the bitstream
29	...	byte 25 of the bitstream

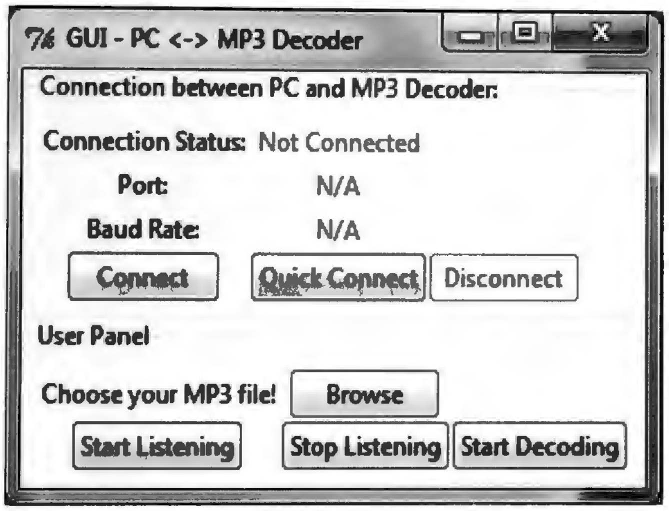
user has selected a MP3 file to decode, it will then be sent through the serial communication to the FPGA. FPGA will save the incoming file contents in a large RAM that can contain up to 70,000 bytes. In this case the MP3 file that can be decoded needs to be less than 70,000 bytes in size. Once the MP3 file is completely sent to the FPGA, another command is sent to the MP3 decoder to start decoding.

The Python language was used to create the command interface. PySerial, which is a Python module, was used to establish a communication between the host and MP3 decoder. The Python IO module selects the desired MP3 file and allows access to the file.

To provide a more user friendly command interface, Tcl/Tk framework was used to create a Graphical User Interface(GUI). Figure 6.19 shows the GUI when it is not connected to the MP3 decoder. Many of the features are not available because of no connectivity between the two cores. When the GUI and MP3 decoder establish a connection, the GUI will look like the Figure 6.20. The user can select an MP3 file by pressing the Browse button, once the User presses Start Decoding, the selected file is sent to the FPGA.

6.5. Audio Processing

The ML501 development board has an the Analog Devices AD1981 Audio Codec which supports stereo 16-bit audio with up to 48-kHz sampling. The AD1981B contains


GUI - PC <-> MP3 Decoder

Connection between PC and MP3 Decoder:

Connection Status: Not Connected

Port: N/A

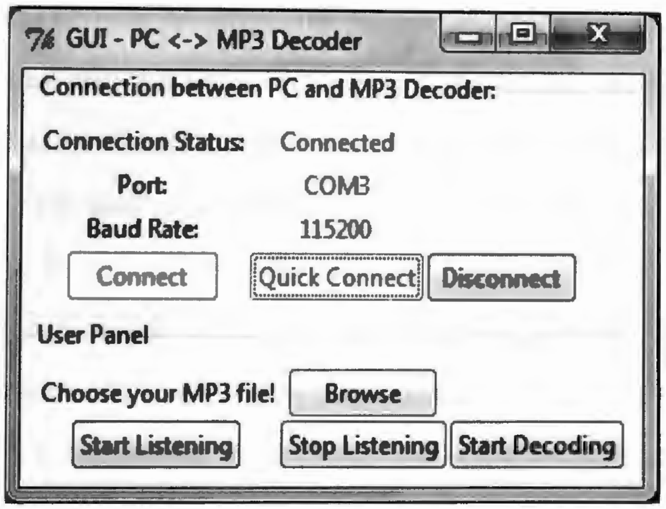
Baud Rate: N/A

Figure 6.19. Graphical User Interface when the host computer is not connected to the MP3 decoder.

an Audio Codec 97 (AC 97) Architecture and Digital Interface (AC-link) designed to implement audio functionality in a given system. The AC 97 Codec performs DAC and ADC conversions, mixing, and analog I/O for audio (or modem), and always functions as slave to an AC 97 Digital Controller. The digital link is bi-directional, five wire time domain based interface that connects the AC 97 Digital Controller to the AC 97 Codec.

AC-link 5 wires are BIT_CLK, RESET, SDATA_IN, SDATA_OUT and SYNC. When the AC 97 is configured as a primary codec, BIT_CLK is driven by the codec and clocks the stream of the serial data. RESET is the reset line of the codec and is active low. SDATA IN and SDATA OUT are used to transmit serial data in and out of the codec. SYNC is used for synchronization of each frame. The interface is shown in the Figure 6.21.

AC-link handles multiple input and output PCM audio streams, as well as control register accesses employing a time division multiplexed (TDM) scheme that divides each audio frame into 12 outgoing and 12 incoming data streams, each with 20-bit sample resolution. With a minimum required DAC and ADC resolution of 16-bits, AC 97 could also be implemented with 18 or 20-bit DAC/ADC resolution, given the headroom that the



Connection between PC and MP3 Decoder:

Connection Status: Connected

Port: COM3

Baud Rate: 115200

User Panel

Choose your MP3 file!

Figure 6.20. Graphical User Interface when the host computer is connected to the MP3 decoder.

AC-link architecture provides.

The AC 97 Controller signals synchronization of all AC-link data transactions. The AC 97 Codec drives the serial bit clock onto AC-link, which the AC 97 Controller then qualifies with a synchronization signal to construct audio frames. SYNC, fixed at 48 kHz, is derived by dividing down the serial bit clock (BIT_CLK). BIT_CLK, fixed at 12.288 MHz, provides the necessary clocking granularity to support 12 20-bit outgoing and incoming time slots. AC-link serial data is transitioned on each rising edge of BIT_CLK. The Codec

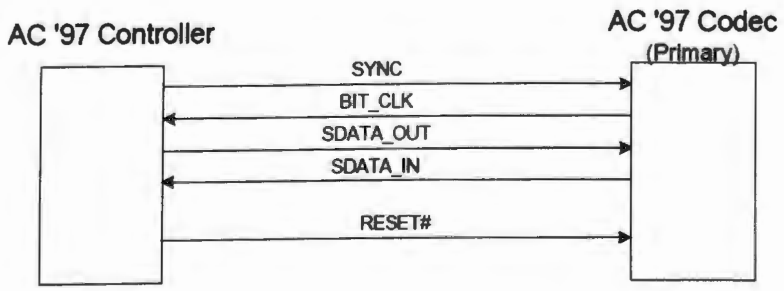


Figure 6.21. AC Link.

samples at the falling edges of BIT_CLK. The AC-link protocol uses a 16-bit time slot (Slot 0) to validity the time slots use for the current audio frame being received. SYNC is set high by the controller for the slot 0. The 13 time slots is shown in Figure 6.22.

Slot 1 and 2 are used to set some of the registers within the codec in this case AD1981B. Some of the registers are volume control where within power up sequence, the audio output are muted and require modification. Along with volume settings, the settings of the DAC included in the codec require to be changed and modified to accommodate the current MP3 file's settings such as sampling frequency of the PCM samples. AD1981B provides sampling frequency range between 7kHz and 48kHz in steps of 1Hz, which covers every possible sampling frequency used for encoding a MP3 file. Slot 3 and 4 are 20 bit PCM data for Left and Right channels. Rest of the slots are not used. Figure 6.23 illustrates the time slot based AC-link protocol.

The ML501 development board contains five separate jacks, three for audio out including headphones and analog line out. The PCM results of the sub-band synthesis is transition from floating point representation into a 20 bit fixed point value and stored into a FIFO RAM. The FIFO RAM is divided into two different banks, where each banks contain PCM data for each granule in an MP3 file frame. Each bank also contains data for both of the channels within the granule of the audio frame. Once a bank is read, the bank is empty to be written to by the sub-band synthesis. The driver will then read data from the

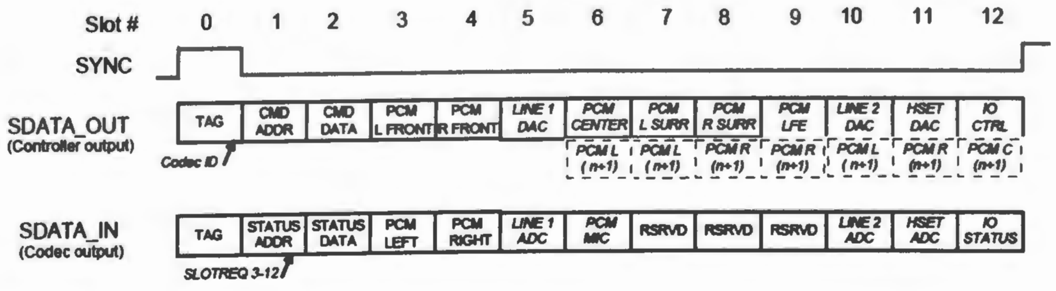


Figure 6.22. Bi-directional AC-link frame with slot assignments.

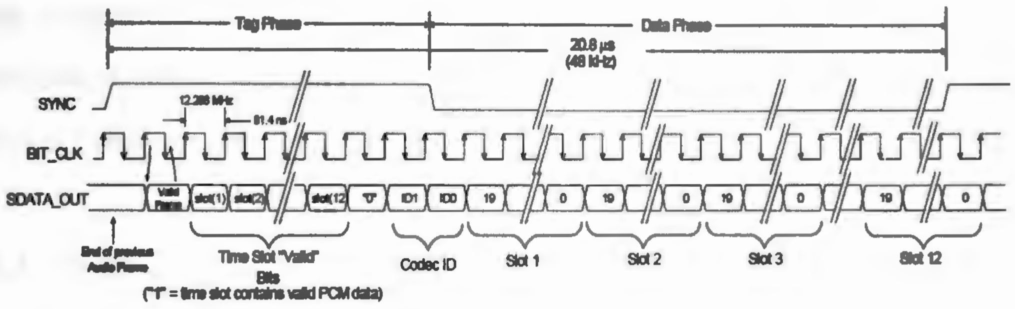


Figure 6.23. AC-link output frame.

next bank while the sub-band synthesis fills up the recently emptied bank. One method to validate liveliness and process of the MP3 decoder is to ensure when the AC 97 driver has read a bank, the next bank has been filled up.

6.6. FPGA Implementation: Common Implementation Elements

This design is implemented on ML501 development board housing a XC5VLX50 FPGA. The XC5VLX50 FPGA has 7200 slices, in which each slice contains four LUTs and four flip-flops. The XC5VLX50 also has 48 DSP48E slices.

6.6.1. Block Random Access Memory

Block Random Access Memory (BRAM) is dedicated, configurable memory with address, data and control ports. For designs where a lot of data is stored, BRAM is the most efficient method, instead of exhausting the flip flops available on the chip.

The BRAM in Virtex-5 FPGAs stores up to 36K bits of data and can be configured as either two independent 18 Kb RAMs, or one 36 Kb RAM. Each 36 Kb block RAM can be configured as a 64K × 1 (when cascaded with an adjacent 36 Kb block RAM), 32K × 1, 16K × 2, 8K × 4, 4K × 9, 2K × 18, or 1K × 36 memory. Each 18 Kb block RAM can be configured as a 16K × 1, 8K × 2 , 4K × 4, 2K × 9, or 1K × 18 memory. Write and read are synchronous operations in the Block RAMs, two ports are symmetrical and totally independent, sharing only the stored data. BRAM is used for all the PEs with large buffers. Table 6.7 shows the PEs that use BRAM.

6.6.2. Clocking

After the design is synthesized, mapped and routed onto the FPGA, the highest clocking frequency is found. It may vary based on the design. The clock generator on the ML501 board provides several clock frequencies. One of the frequencies is 100 MHz single ended which is used as an input into Phase Locked Loop (PLL) in the FPGA. The Phase Locked Loop is also configured at different configurations to output different clocks. Figure 6.24 shows the PLL configuration for generating 87.5 MHz clock.

6.6.3. Handshake

An important characteristic of the transfer of data is the handshake protocol. In this section, 4-phase handshake protocol is described as the handshake protocol used for every instance where data is transferred.

A 4-phase handshake is defined by four steps as shown in Figure 6.25.

1. assert request
2. assert acknowledge
3. de-assert request
4. de-assert acknowledge

Data can be transferred during the first or second step. When the switch box in the NoC is transferring data, the data is transferred at first step. However, when switch box is retrieving data from one of the input buffers, first step requires the data to be transferred on

Table 6.7. BRAM usage in the PEs.

PE	Size (rows × bits)	Purpose
Manager	4098 × 8	Known as hbuf in the MP3 decoder algorithm
Huffman	2308 × 8	Huffman Table contents
Antialias	576 × 32	Input data Dequantized and stereo jointed Data
Hybrid	18 × 32	Input data from Antialias
Hybrid	1152 × 32	Previous channel data
Sub-band Synthesis	576 × 32	Total hybrid output for one channel and one granule
Sub-band Synthesis	2048 × 32	Used for Sub-band Synthesis process
Sub-band Synthesis	2304 × 16	Decoded PCM values for a frame

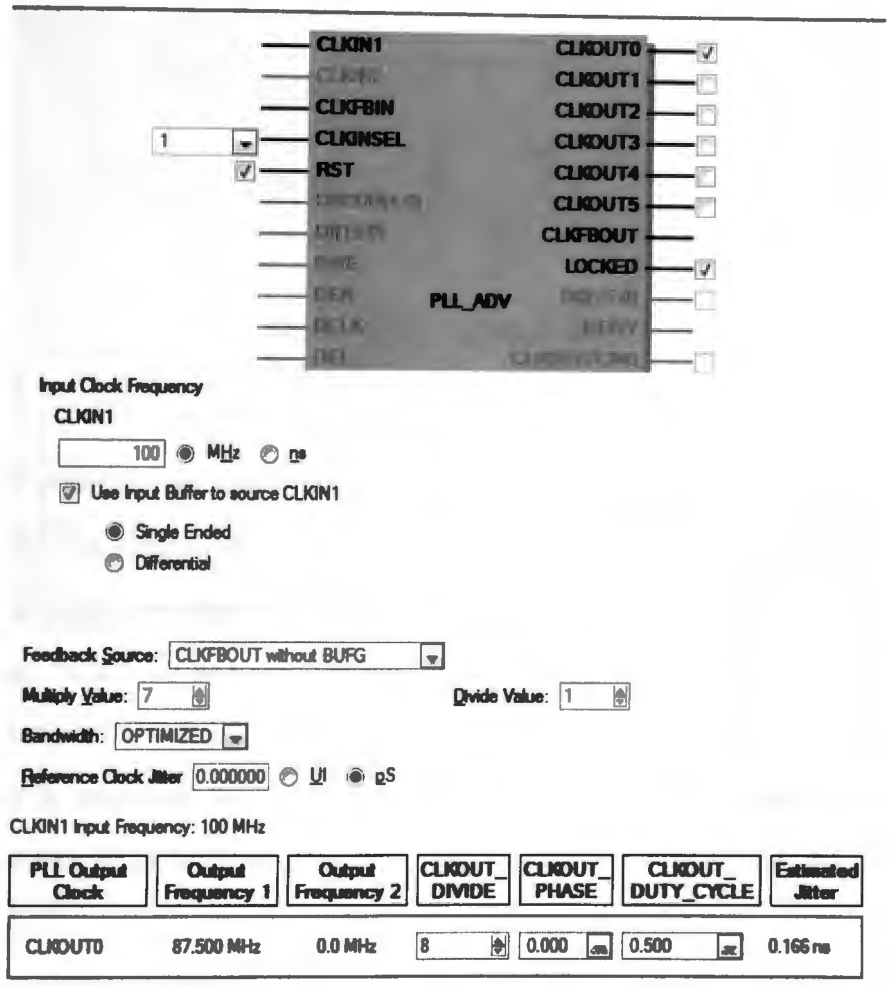
the second step.

6.7. Manual Mapping of MP3 Decoder on 3x3NoC: Different Design Aspects

One of the most crucial factors in the performance of the system is placing or mapping the PEs on the NoC architecture. MP3 decoder was partitioned into 9 PEs in Section 3.5, and each PE needs to have a location on the NoC . A shared system bus does not carry a notion of location since each connected component to the bus is treated equally. Placement of PEs on the NoC affects the performance. The following sections explain the process of placing or mapping the PEs on the NoC, which i did manually. However, this mapping process can be automated too.

6.7.1. Interactions between the Processes

Replicating the Fraunhofer Institute's open library helped me to understand the flow chart of the MP3 decoder as shown in Figure 3.5. The flow chart helped develop an interaction map between the tasks of the MP3 decoder. Figure 6.26 shows the interactions between PEs, which helps in the process of placing the PEs.



CLKINSEL: 1
RST: ✓
CLKOUT0: ✓
LOCKED: ✓

Input Clock Frequency
CLKIN1
100 [✓ MHz] ns
✓ Use Input Buffer to source CLKIN1
[✓ Single Ended]
Differential

Feedback Source: CLKFBOUT without BUFG
Multiply Value: 7
Divide Value: 1
Bandwidth: OPTIMIZED
Reference Clock Jitter 0.000000 UI [✓ pS]

CLKIN1 Input Frequency: 100 MHz

PLL Output Clock	Output Frequency 1	Output Frequency 2	CLKOUT_ DIVIDE	CLKOUT_ PHASE	CLKOUT_ DUTY_CYCLE	Estimated Jitter
CLKOUT0	87.500 MHz	0.0 MHz	8	0.000	0.500	0.166 ns

Figure 6.24. Configuration of PLL used to generate 87.5 MHz clock frequency.

6.7.2. Mapping of PEs on NoC

Besides the interactions between the PEs, there are other factors that influence the placement such as the number of packets transferred between the PEs. For example, Sync will transfer a lot of bitstream bytes to the Manager, However Hybrid interacts with Manager four times during each frame and each time only a single flit long long packet is transferred. The importance of placing Sync and Manager close is higher than that of placing Hybrid and Manager close.

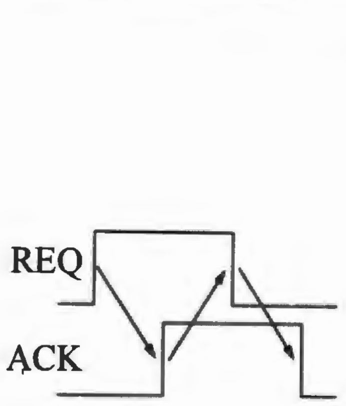


Figure 6.25. 4-phase handshake.

One of the factors involved in increasing speed of the application in the NoC [44] is the optimal placement of the PEs. If Synthesis is placed at location (0,0) and Hybrid at location (2,2), then the data from the Hybrid has to travel four links to arrive to the destination. In a 3 x 3 homogenous NoC using X-Y routing algorithm, the longest path has four links. The longer the path, the longer it takes for a packet to travel from point A to point B. Therefore, the best placement of two PEs where large amounts of data are transferred between them, is to place them where the path between them is only one link if possible.

As a starting point, the flow of the MP3 bitstream and the results are observed. Bitstream is transferred to Sync through Manager with Sync requesting more bits of the MP3 file from Manager. Sync and Manager should then be placed close to each other. Synthesis is the last PE in the loop. The output from the Synthesis are the result that can be sent back to the PC or played through an audio codec, Therefore the Synthesis is best placed close to the Hybrid.

Manager block is the most important component of the MP3 decoder since a large number of data transferred occurs in Manager. Manager establishes the communication with the host computer. Hence Manager must be placed at a location on the outside border. Manager can be placed on any PE that is not PE (1,1).

Manager interacts with four PEs. Therefore, Manager is placed on an inside bor-

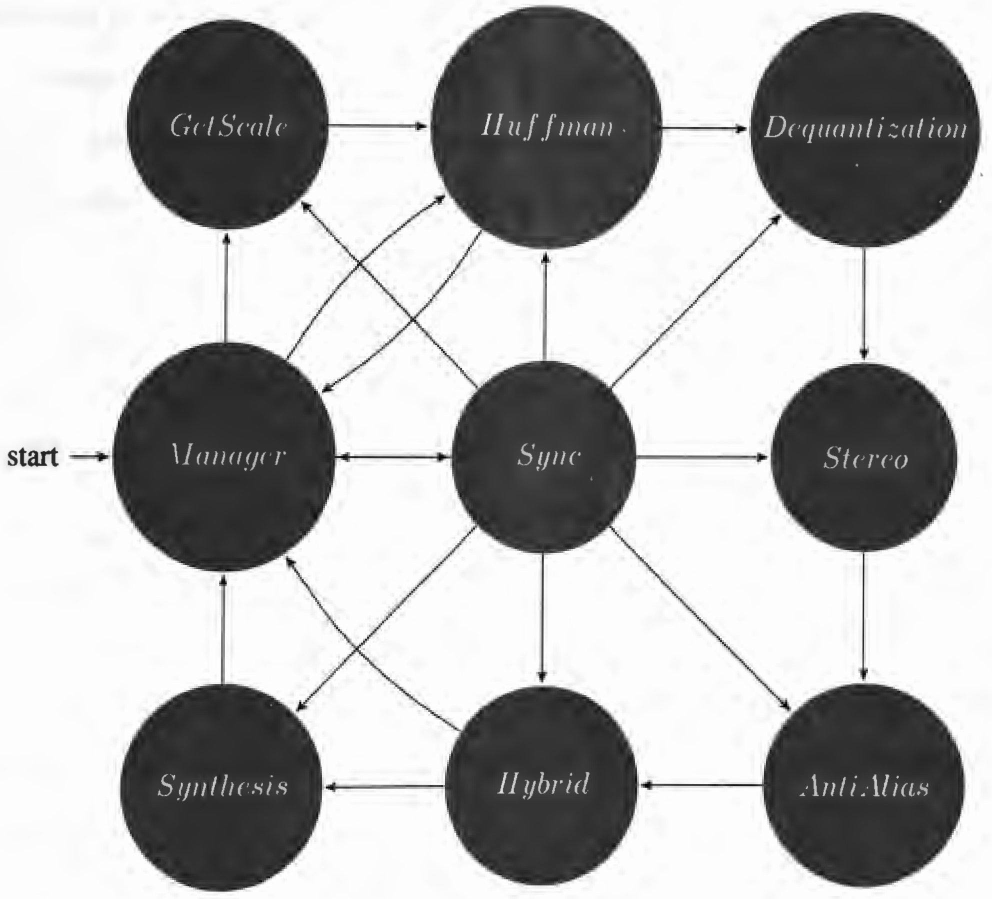


Figure 6.26. Interactions between the PEs in the NoC.

dering PE with three links. Manager can be placed at any of the four possible locations (1,0), (0,1), (1,2) or (2,1) in the platform, and Manager was placed at position (1,0). The neighboring positions to Manager are then (0,0), (1,1) and (2,0). Sync is required to be a neighboring PE of Manager. Since Manager handles communication with PC or other component, Synthesis is added as a neighbor of Manager.

Sync interacts with the all the PEs. Therefore, its best placement is at location (1,1) where it has four 1 link neighboring PEs and the other four PEs are only two links away. Since Sync is now placed at (1,1), Synthesis can be placed at (0,0) or (2,0). Due to the symmetry created by placing Manager at position (1,0), Synthesis can be placed at either position and produce the same results. So for this case, Synthesis is placed at (2,0).

Manager also interacts with two more PEs: Huffman and Get Scale. As shown in Figure 3.5, after synchronization and extraction of the side information block, Get Scale is the next process. After that, Huffman can start processing. The only interaction between Hybrid and Manager is a one element long packet for each channel in a single granule that informs Manager of completion of a channel in a granule of a frame. The small traffic between Hybrid and Manager does not require for Hybrid to be placed close to Manager.

The current placement of PEs is shown in Figure 6.27.

Figure 6.27 shows that Synthesis has one PE neighbor that can be explored. Figure 6.26 shows that Hybrid is the only PE besides Manager and Sync which are already placed, to interact with Synthesis and the magnitude of the interaction between Hybrid and Synthesis is a large percentage of the traffic. So it is better for Hybrid to be placed very close to Synthesis so the distance between them is only one link. Hybrid is consequently placed at location (2, 1).

After placing Hybrid at location (2,1), Hybrid has one vacant neighbor left. In Figure 6.26, Antialias sends its output which is a 4608 element long packet for each frame to Hybrid. Therefore, it is better for Antialias to be placed at location (2,2). This creates another tight constraint on the placement of the PE at location (1,2). Figure 6.26 shows that Stereo sends its output to Antialias, hence another strong reason why Stereo should be placed at (1,2). Such dependencies require to place Dequantization at location (0,2).

Two processes and locations now remain to be decided; Get Scale and Huffman and two locations (0,0), (0,1). Huffman can be placed at (0, 1). This results in Huffman being placed at two links away from Manager which is not very desirable. Instead if Huffman is placed at location (0,0), Huffman would be two links away from Dequantization, and Get Scale would be also two links away from Manager. Hence, it is better if if Huffman is placed at (0, 1). On the contrary, Sync and Get Scale are idle when Huffman is in execution,

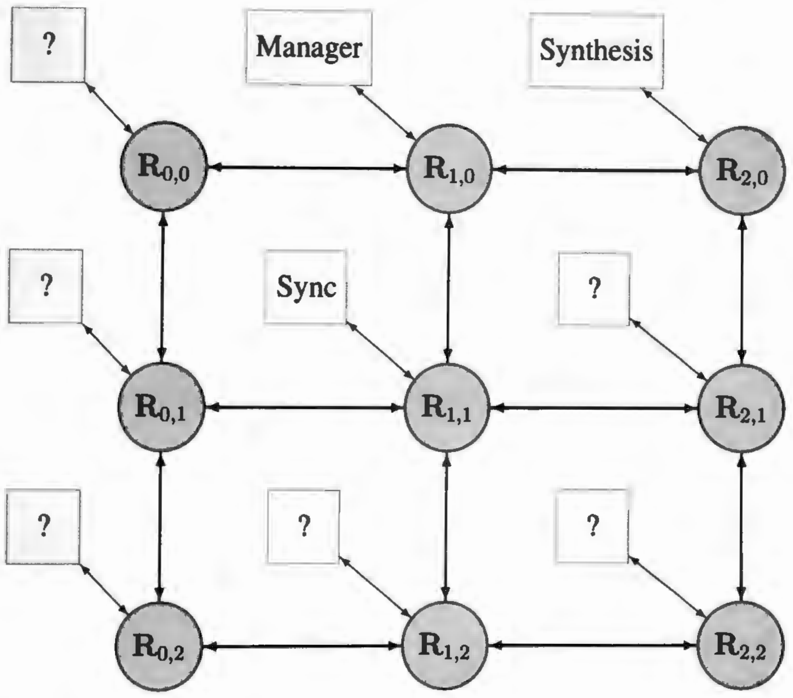


Figure 6.27. First phase of placement of PEs in the NoC.

so the only traffic between Huffman and Manager is the data between the Huffman and Manager. Get Scale is therefore placed at location at (0,0).

The final placement is shown in Figure 6.28.

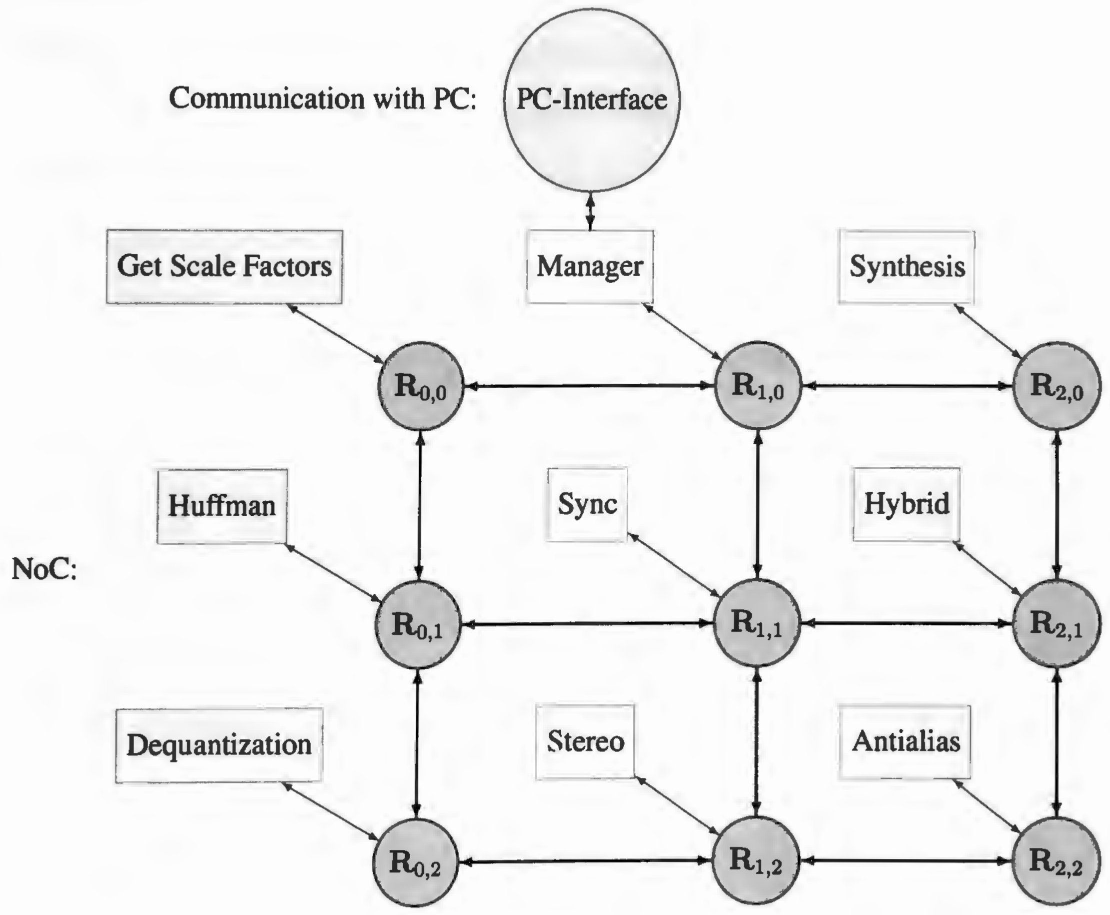


Figure 6.28. Final phase of placement of PEs in the NoC

CHAPTER 7. RESULTS

In this Chapter, I discuss the results achieved by the two implementations.

7.1. Device Utilization Summary

Table 7.8 shows the comparison between bus based and NoC based MP3 decoder. The results were calculated after synthesis of the design using Xilinx Synthesis Technology (XST), mapping, placing and routing using Xilinx tools.

It was expected that NoC would use more resources than the bus, due to buffering of packets between each router and the network interfaces. This was confirmed with more slice LUTs being used for memory, and more of the Block RAM and FIFO resources were used for the NoC based implementation.

Table 7.8. Device utilization summary for NoC and bus based MP3 decoder.

Slice Logic Utilization	Bus	NoC	Available	Bus %	NoC %
Number of Slice Registers	5,748	10,419	28,800	19%	36%
Number of Slice LUTs	10,355	13,562	28,800	35%	47%
Number of Slice LUTs used as logic	10,051	12,439	28,800	34%	43%
Number of Slice LUTs used as Memory	233	830	7,680	3%	10%
Number of occupied Slices	3,785	5,382	7,200	52%	74%
Number of BlockRAM/FIFO	35	44	48	72%	91%
Number of DSP48Es	42	42	48	87%	87%

7.2. Layout

XST synthesizes the design and Xilinx ISE maps, places and routes the design on to the target FPGA architecture. Figures 7.29 and 7.30 shows the layout of the MP3 decoder divided into PEs and the routers for the NoC based MP3 decoder. Some of the PEs are placed in a scattered fashion. This can be explained by some PEs requirement to use the DSP slices and RAM blocks in the targeted FPGA. Because DSP slices and RAM blocks are limited, the design has to be placed and routed around the locations of the pre-placed blocks. This which leads to the placement of the PEs in a scattered fashion. Scattered placement of the PEs leads to heterogeneous placement of the routers as shown in Figure 7.30.

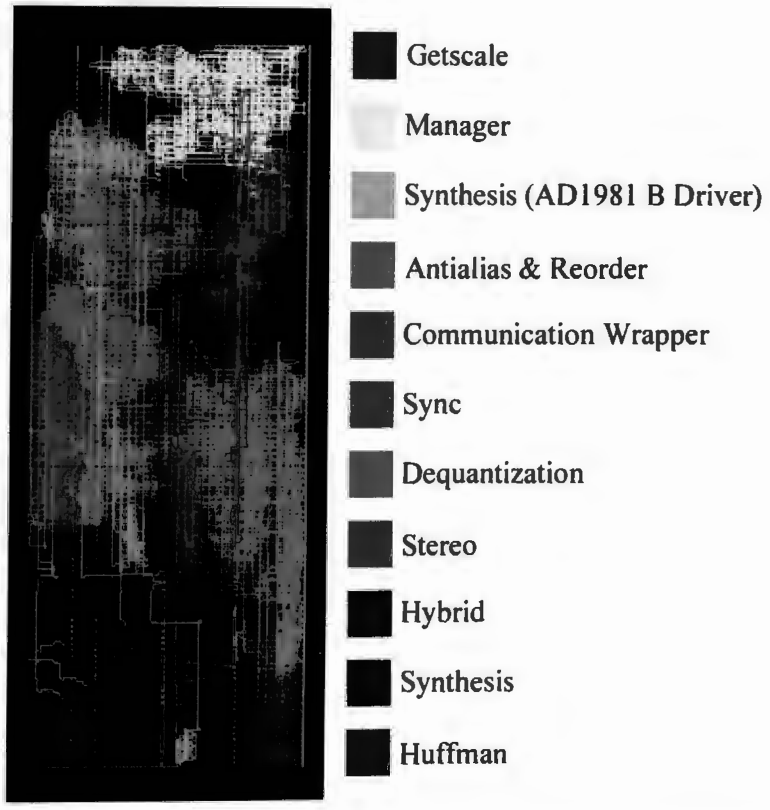


Figure 7.29. Layout of the NoC based MP3 decoder showing the PEs.

Figures 7.31 and 7.32 illustrate the layout of the bus based MP3 decoder. It can be observed that the bus has long nets which run along the whole FPGA chip.

7.3. Critical Path

Critical path dictates the maximum clock frequency for both of the designs. Critical path of NoC and bus based MP3 decoder were determined after the design was placed and routed into the FPGA architecture using Xilinx tools. Critical path over view for NoC is shown in Figure 7.33 and the data path of the critical path is shown in Figure 7.34. Critical

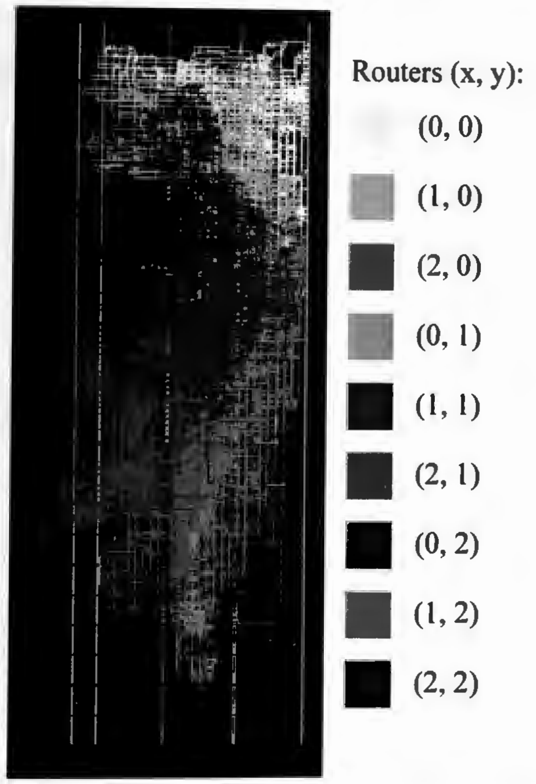


Figure 7.30. Layout of the NoC based MP3 decoder showing the Routers.

path over view for bus is shown in Figure 7.35 and the data path of the critical path is shown in Figure 7.36.

The best clock rate achievable by NoC based MP3 decoder is 91.7 MHz. The best clock rate achievable by the bus based MP3 decoder is 80.45MHz.

7.4. Performance

MP3 player requires for a frame to be decoded into PCM samples in less than 24 ms in order to avoid gaps between frames and maintain real time performance. Each frame is composed of two granules and each granule is composed of two channels if the MP3 file supports stereo mode. Therefore, each channel needs to be decoded in less than 6ms. An IO pin is toggled upon completion of a channel in a granule for each frame to ensure real

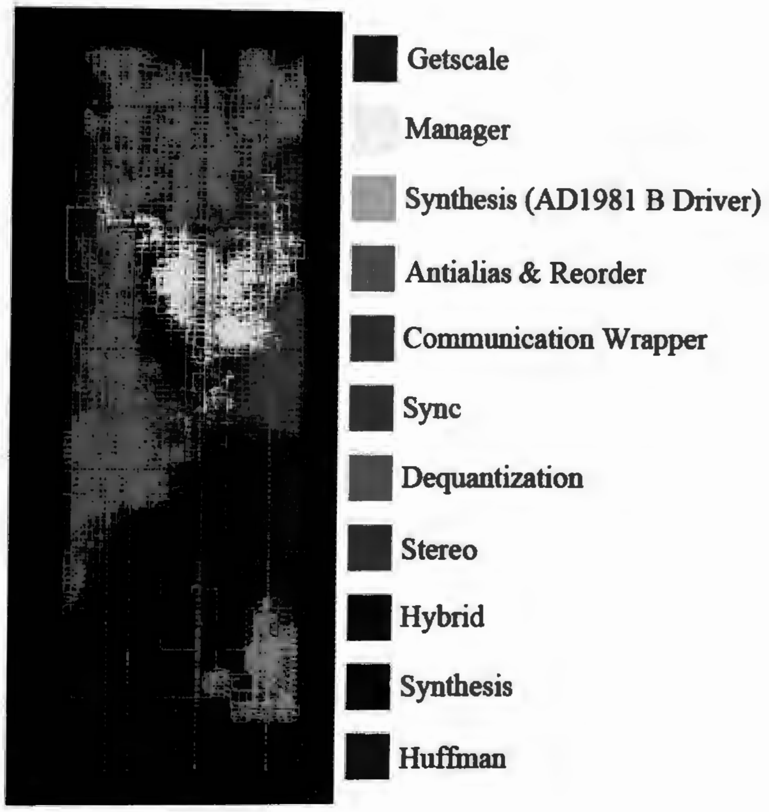


Figure 7.31. Layout of the bus based MP3 decoder showing the PEs.

time performance for both NoC and bus based MP3 decoder. The test was performed using the same MP3 file along with both cases settings described in Table 7.9. The dedicated IO pin was sampled by Agilent Logic Analyzer 1681A at 100ns sampling rate.

It is clear that NoC based MP3 decoder performs faster than the bus based MP3 decoder.

7.5. Power Consumption

Power estimation was done by Xilinx XPower Analyzer. Table 7.10 reports the power estimation results for Bus and NoC based MP3 decoder. It was expected NoC to consume more power due to larger amount of resources used for NoC based MP3 decoder which is

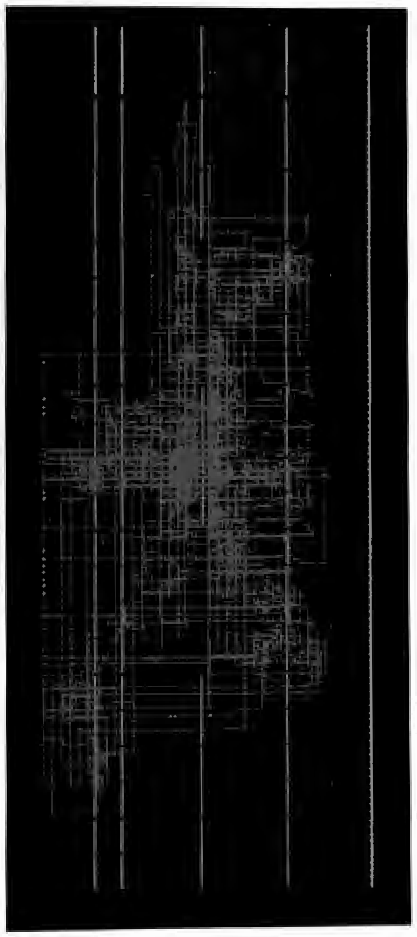


Figure 7.32. Layout of the bus based MP3 decoder showing the Bus.

distinguishable on Clocks, Logic and BRAMs section of Table 7.10.

7.6. Experimental Setup

Figure 7.37 shows the ML501 development board connected to the computer along with speakers. The virtual communication port driver which is used to send data to the FPGA is shown along with the Xilinx Platform Cable USB II used to send bitstream of the design through the USB in order to configure the FPGA. Figure 7.38 shows a close up of the ML501 and its external connections to the speakers, programmer and the serial

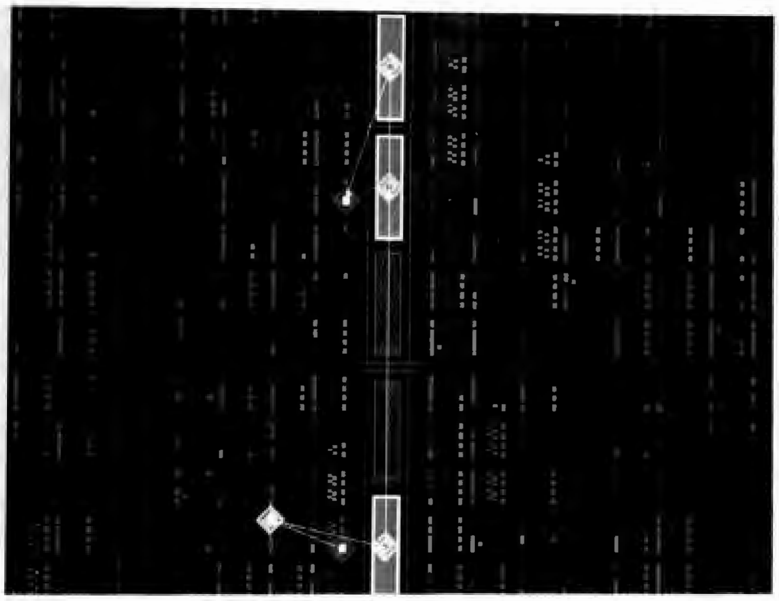


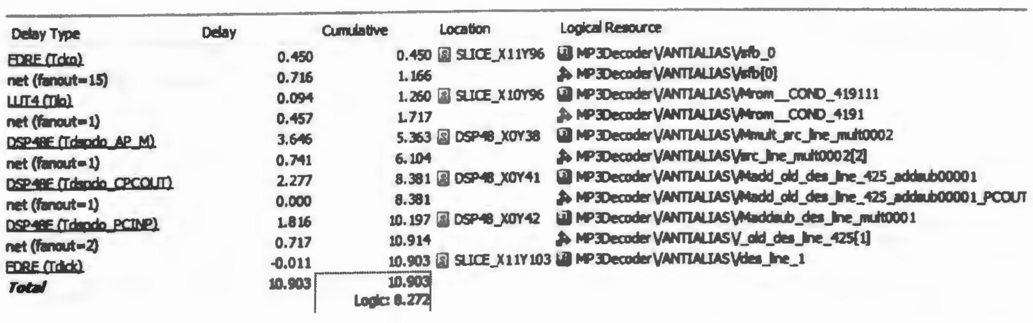
Figure 7.33. Critical Path shown in the Device for a NOC based MP3 decoder.

communication.

7.7. Challenges Faced during Working on this Project

Some of the challenges that I faced during the project are:

1. Partitioning the MP3 decoder: In order to save resources and increase the perfor-



Delay Type	Delay	Cumulative	Location	Logical Resource
FDRE (Tcko)	0.450	0.450	SLICE_X11Y96	MP3Decoder\/ANTIALIAS\/sfb_0
net (fanout=15)	0.716	1.166		MP3Decoder\/ANTIALIAS\/sfb[0]
LUT4 (Tilo)	0.094	1.260	SLICE_X10Y96	MP3Decoder\/ANTIALIAS\/Mrom__COND_419111
net (fanout=1)	0.457	1.717		MP3Decoder\/ANTIALIAS\/Mrom__COND_4191
DSP48E (Tdspdo_AP_M)	3.646	5.363	DSP48_X0Y38	MP3Decoder\/ANTIALIAS\/Mmult_src_line_mult0002
net (fanout=1)	0.741	6.104		MP3Decoder\/ANTIALIAS\/src_line_mult0002[2]
DSP48E (Tdspdo_CPCOUT)	2.277	8.381	DSP48_X0Y41	MP3Decoder\/ANTIALIAS\/Madd_old_des_line_425_addsub00001
net (fanout=1)	0.000	8.381		MP3Decoder\/ANTIALIAS\/Madd_old_des_line_425_addsub00001_PCOUT
DSP48E (Tdspdo_PCINP)	1.816	10.197	DSP48_X0Y42	MP3Decoder\/ANTIALIAS\/Maddsub_des_line_mult0001
net (fanout=2)	0.717	10.914		MP3Decoder\/ANTIALIAS\/_old_des_line_425[1]
FDRE (Tdick)	-0.011	10.903	SLICE_X11Y103	MP3Decoder\/ANTIALIAS\/des_line_1
Total	10.903	10.903 Logic: 8.272		

Figure 7.34. Description of the critical path shown in the device for the NoC based MP3 decoder.

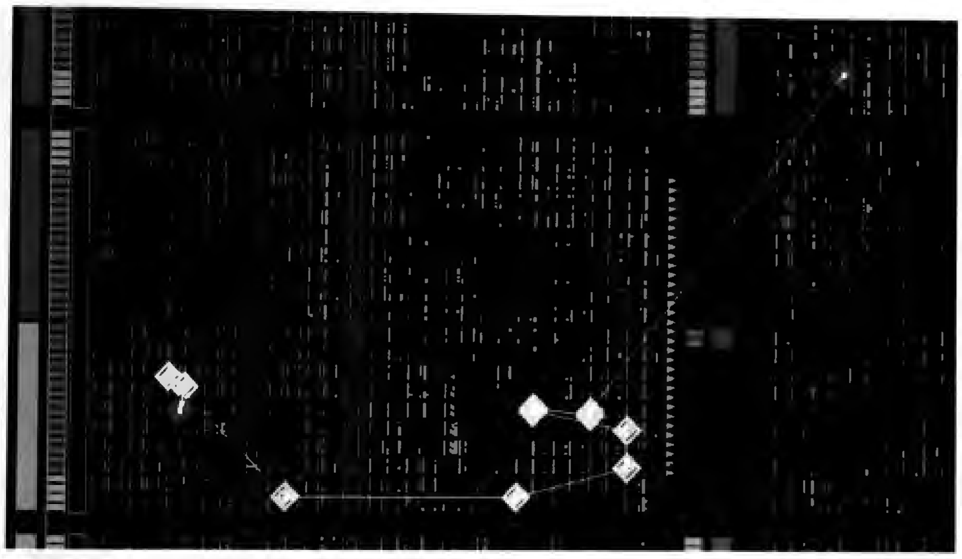


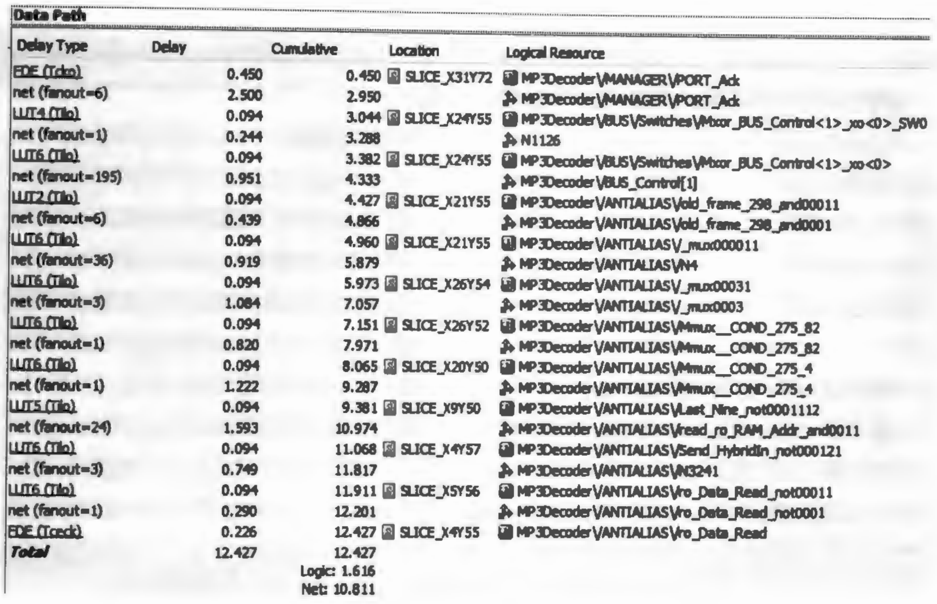
Figure 7.35. Critical path shown in the device for the Bus based MP3 decoder.

Table 7.9. Performance results.

Communication Method	clock frequency (MHz)	Computation time for a channel (ms)
NoC	87.5	4.85
Bus	80	5.342

mance of the decoder, the MP3 algorithm must be well understood. It is necessary to know that certain processes such as Reorder and Anti aliasing will not work together in the same channel. They can be grouped into one block in order to use fewer floating point arithmetic cores, RAM size, and other resources.

2. Differences between software and hardware implementations of the MP3 decoder: besides limitations on available resources, some arithmetic operations require more than a simple addition. When MP3 decoder performs IMDCT, buffer index values are determined by division of values by numbers that are not power of two. The FPGA offers core dividers with different settings that can be used to carry such divisions, but they require resources and introduce a new delay to the system. However, if the



Data Path

Delay Type	Delay	Cumulative	Location	Logical Resource
FDE (Tcko)	0.450	0.450	SLICE_X31Y72	MP3Decoder/MANAGER/PORT_Ack
net (fanout=6)	2.500	2.950		MP3Decoder/MANAGER/PORT_Ack
LUT4 (Tilo)	0.094	3.044	SLICE_X24Y55	MP3Decoder/BUS/Switches/Mxor_BUS_Control<1>_xo<0>_SW0
net (fanout=1)	0.244	3.288		N1126
LUT6 (Tilo)	0.094	3.382	SLICE_X24Y55	MP3Decoder/BUS/Switches/Mxor_BUS_Control<1>_xo<0>
net (fanout=195)	0.951	4.333		MP3Decoder/BUS_Control[1]
LUT2 (Tilo)	0.094	4.427	SLICE_X21Y55	MP3Decoder/ANTIALIAS/old_frame_298_and00011
net (fanout=6)	0.439	4.866		MP3Decoder/ANTIALIAS/old_frame_298_and0001
LUT6 (Tilo)	0.094	4.960	SLICE_X21Y55	MP3Decoder/ANTIALIAS/_mux000011
net (fanout=36)	0.919	5.879		MP3Decoder/ANTIALIAS/N4
LUT6 (Tilo)	0.094	5.973	SLICE_X26Y54	MP3Decoder/ANTIALIAS/_mux00031
net (fanout=3)	1.084	7.057		MP3Decoder/ANTIALIAS/_mux0003
LUT6 (Tilo)	0.094	7.151	SLICE_X26Y52	MP3Decoder/ANTIALIAS/Mmux__COND_275_82
net (fanout=1)	0.820	7.971		MP3Decoder/ANTIALIAS/Mmux__COND_275_82
LUT6 (Tilo)	0.094	8.065	SLICE_X20Y50	MP3Decoder/ANTIALIAS/Mmux__COND_275_4
net (fanout=1)	1.222	9.287		MP3Decoder/ANTIALIAS/Mmux__COND_275_4
LUT5 (Tilo)	0.094	9.381	SLICE_X9Y50	MP3Decoder/ANTIALIAS/Last_Nine_not0001112
net (fanout=24)	1.593	10.974		MP3Decoder/ANTIALIAS/read_ro_RAM_Addr_and0011
LUT6 (Tilo)	0.094	11.068	SLICE_X4Y57	MP3Decoder/ANTIALIAS/Send_HybridIn_not000121
net (fanout=3)	0.749	11.817		MP3Decoder/ANTIALIAS/N3241
LUT6 (Tilo)	0.094	11.911	SLICE_X5Y56	MP3Decoder/ANTIALIAS/ro_Data_Read_not00011
net (fanout=1)	0.290	12.201		MP3Decoder/ANTIALIAS/ro_Data_Read_not0001
FDE (Tceck)	0.226	12.427	SLICE_X4Y55	MP3Decoder/ANTIALIAS/ro_Data_Read
Total	12.427	12.427		
		Logic: 1.616		
		Net: 10.811		

Figure 7.36. Description of the critical path shown in the device for the Bus based MP3 decoder.

pattern of the indices is carefully observed, it can be realized that the indices can be determined by certain pattern operations based on the previous indices. Such patterns can simplify the process further. However, certain processes cannot benefit from such technique. During Dequantization, an integer number is raised to the power of $\frac{4}{3}$, which require an IP core.

3. Proper coding: In order to use certain features of the FPGA chip, a certain style to write verilog code must be used. For example, if an array is accessed in certain format, it is then translated to a RAM when the design is synthesized using XST. However, if the method of accessing the array was not carefully used, the code could be translated to just sets of flip-flops for the array. This is not using the FPGA resources properly.

4. Huffman table contents: During the initial stage of the MP3 decoder, the Huffman table contents are requested from the Host, this method introduced a huge delay in

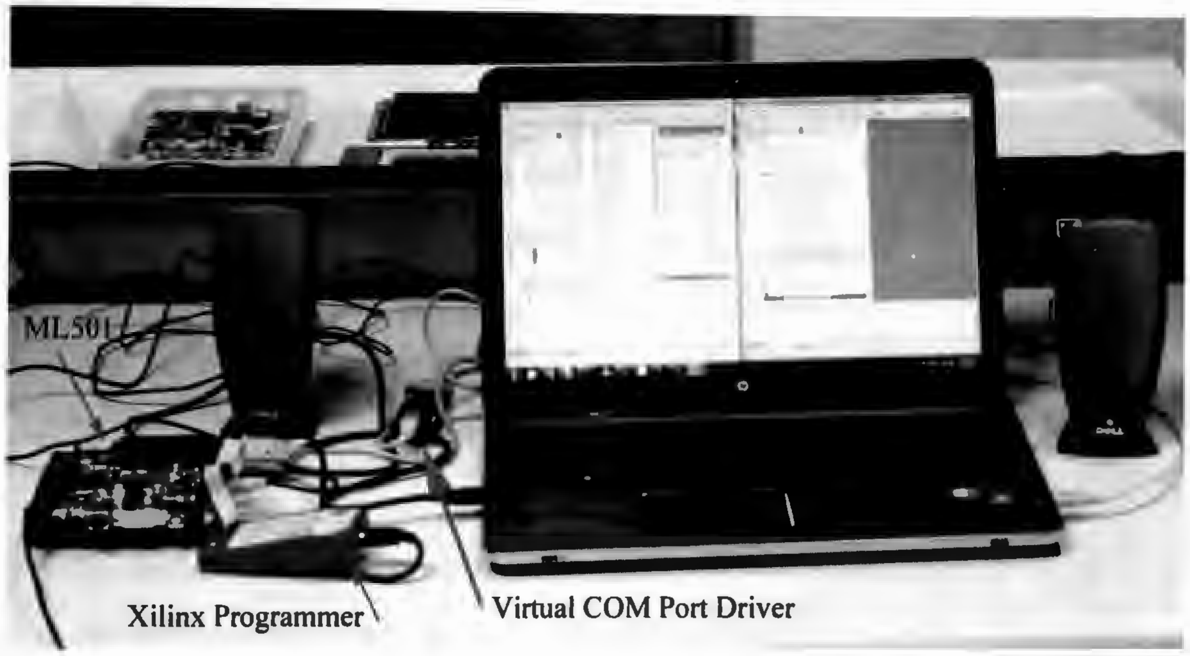


Figure 7.37. MP3 decoder setup shown with a computer host and speakers.

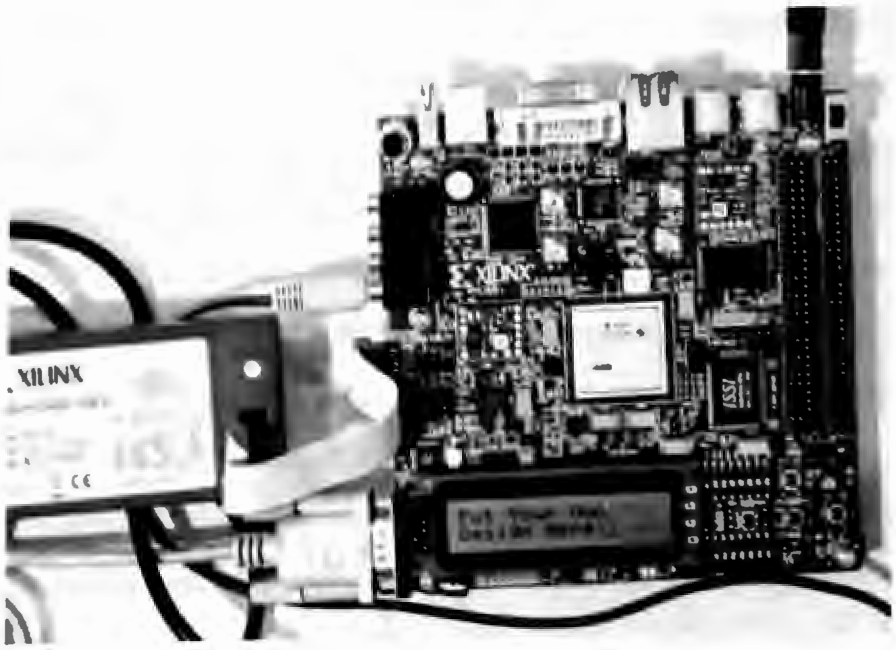


Figure 7.38. ML501 development board shown with auxiliary connections.

Table 7.10. Power estimation for NoC and bus based MP3 decoder.

On-Chip	Power Bus (W)	Power NoC (W)
Clocks	0.100	0.155
Logic	0.004	0.008
Signals	0.011	0.016
BRAMs	0.057	0.069
DSPs	0.001	0.001
PLLs	0.114	0.104
IOs	0	0
Leakage	0.423	0.424
Total	0.711	0.777

the decoder. A block RAM was instead implemented with its initial contents set as the Huffman table contents as a result. This eliminated the need to retrieve the huge amount of data for each frame.

5. Differences in PEs when using different communication: Besides differences in sending and receiving data mechanisms when using different communication methods, there are other differences that need to be addressed. For example, when the Huffman has decoded a set of numbers, Huffman needs to send the data one byte at a time to the Dequantization PE. In the NoC based decoder, the Huffman sends the data if the wrapper is available, and the wrapper then forwards the data to the Dequantization. In the bus based decoder, if the Huffman has data to send, Huffman has to wait to get the control of the bus. Once Huffman has control of the bus, it tries to send data to the Dequantization. However, if Dequantization is busy processing data, the Huffman holds the bus until the Dequantization is ready to receive. While Huffman is holding the bus, other data can be transferred. One solution to this problem was to design a timeout scenario where the PE controlling the bus releases the bus if the receiver is not responding.

CHAPTER 8. CONCLUSION AND FUTURE WORK

8.1. Conclusion

It was expected to achieve a faster clock frequency for the NoC based MP3 decoder since the nets for the NoC are shorter compared to the bus based MP3 decoder. Also, it was expected to achieve a real time operation of for both the NoC and bus based MP3 decoder. The final design in the thesis demonstrated that the NoC based MP3 decoder achieves a 14% faster clock frequency. Both implementations achieve real time operation with the NoC based design decode an MP3 frame on average in 10% less time that the bus based MP3 decoder. Also, the design parameters achieved by Xilinx tools such as number of logic blocks used to implement both NoC and bus based MP3 decoder verified the expectation that the NoC based MP3 decoder requires more resources.

8.2. Future Work

Serial communication is a simple and easy communication protocol to implement, but not the fastest. To improve the design, serial communication needs to be eliminated and alternative methods to be considered such as Ethernet or USB.

PEs need to be optimized in order to increase the performance of the system. The whole design uses a global clock signal, and if one of the PEs contains a path that forces the clock signal to be slowed down, then the path must be eliminated by re-designing the PE to achieve better performance.

To achieve a better understanding of differences between bus and NoC implementations a bigger or more complex application such as MPEG decoder should be implemented.

Dynamic and Partial Reconfiguration is an important design technique, which could be utilized. By using partial reconfiguration, different audio decoders could be stored in a memory and loaded on the FPGA platform depending on the audio decoder needed for the file.

BIBLIOGRAPHY

[1] S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial," *IEEE Design and Test of Computers*, vol. 13, no. 2, pp. 42–56, 1996.

[2] Xilinx inc. [Online]. Available: www.xilinx.com

[3] Altera. Altera Inc. [Online]. Available: www.altera.com

[4] K. Lahiri, S. Dey, and A. Raghunathan, "Evaluation of the Traffic-Performance Characteristics of System-on-Chip Communication Architectures," *Proceedings of the 14th International Conference on VLSI Design*, 2001.

[5] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli, "Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design," *Proceedings of the 38th Annual Design Automation Conference*, 2001.

[6] L. Benini and G. D. Micheli, "Powering Networks on Chips: Energy-Efficient and Reliable Interconnect Design for socs," *Proceedings of the 14th International Symposium on Systems Synthesis*, 2001.

[7] S. Murali and G. D. Micheli, "An Application-Specific Design Methodology for STbus Crossbar Generation," *Proceedings of the Conference on Design, Automation and Test in Europe*, vol. 2, 2005.

[8] K. Lee, S.-J. Lee, and H.-J. Yoo, "Low-power Network-on-Chip for High-Performance SoC Design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, February 2006.

[9] J. Duato, S. Yalamanchili, and N. Lionel, *Interconnection Networks, An Engineering Approach.* Morgan Kaufmann, 2002.

[10] L. Benini and G. D. Micheli, "Networks on Chips: a New SoC Paradigm," *Computer*, vol. 35, January 2002.

[11] W. J. Dally, *Principles and Practices of Interconnection Networks.* Morgan Kaufmann, 2004.

[12] T. Bjerregaard and S. Mahadevan, "A Survey of Research and Practices of Network-on-Chip," *ACM Computing Surveys*, vol. 38, 2006.

[13] I. Cidon and I. Keidar, "Zooming in on Network-on-Chip Architectures," Department of Electrical Engineering, Technion, Israel Institute of Technology, Tech. Rep., 2005.

[14] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "Cost Considerations in Network on Chip," *Integration, the VLSI Journal - Special Issue: Networks on chip and reconfigurable fabrics*, vol. 38, October 2004.

[15] CoreLink System IP & Design Tools for AMBA. ARM Ltd. [Online]. Available: http://www.arm.com/products/system-ip/amba/index.php

[16] IBM CoreConnect Bus Cores. IBM.

[17] D. Wingard, "Micro-Network Based Integration for SOCs," *Proceedings of the 38th Annual Design Automation Conference*, 2001.

[18] W. J. Dally and B. Towles, "Route Packets, not Wires: On-Chip Interconnection Networks," *Proceedings of the 38th annual Design Automation Conference*, 2001.

[19] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist, "Network on Chip: An Architecture for Billion Transistor Era," *Proceedings of the IEEE NorChip Conference*, 2000.

[20] T. Bartic, J.-Y. Minolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins, "Highly Scalable Network on Chip for Reconfigurable Systems," *Proceedings of International Symposium System-on-Chip*, November 2003.

[21] A. Sheibanyrad, I. M. Panades, and A. Greiner, "Systematic Comparison between the Asynchronous and the Multi-Synchronous Implementations of a Network on Chip Architecture," *Proceedings of the Conference on Design, Automation and Test in Europe*, 2007.

[22] P. Guerrier and A. Greiner, "A Generic Architecture for on Chip Packet-Switched Interconnections," *Proceedings of the Conference on Design, Automation and Test in Europe*, 2000.

[23] A. Andriahantenaina and A. Greiner, "Micro-network for SoC: Implementation of a 32-port SPIN Network," *Proceedings of the Conference on Design, Automation and Test in Europe*, vol. 1, 2003.

[24] K. Goossens, J. Dielissen, and A. Radulescu, "Æ thereal Network on Chip: Concepts, Architectures, and Implementations," *IEEE Design and Test of Computers*, vol. 22, 2005.

[25] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini, "Xpipes: A Latency Insensitive Parameterized Network-on-Chip Architecture For Multi-Processor SoCs," *Proceedings of the 21st International Conference on Computer Design*, 2003.

[26] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed Bandwidth Using Looped Containers in Temporally Disjoint Networks within the Nostrum Network on Chip," *Proceedings of the Conference on Design, Automation and Test in Europe*, vol. 2, 2004.

[27] "System drivers," International Technology Road map for Semiconductors, Tech. Rep., 2007.

[28] J. Bainbridge and S. Furber, "Chain: A Delay-Insensitive Chip Area Interconnect," *IEEE M*, vol. 22, September 2002.

[29] T. Bjerregaard and J. Sparso, "A Router Architecture for Connection-Oriented Service Guarantees in the MANGO Clockless Network-on-Chip," *Proceedings of the Conference on Design, Au*, vol. 2, 2005.

[30] D. Rostislav, V. Vishnyakov, E. Friedman, and R. Ginosar, "An Asynchronous Router for Multiple Service Levels Networks on Chip," *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, 2005.

[31] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin, "An Asynchronous NOC Architecture Providing Low Latency Service and Its Multi-Level Design Framework," *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, 2005.

[32] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner, "Globally-Asynchronous Locally-Synchronous Architectures to Simplify the Design of On-Chip Systems," *ASIC/SOC Conference*, 1999.

[33] T. Bjerregaard, M. B. Stensgaard, and J. Sparso, "A Scalable, Timing-safe, Network-on-Chip Architecture with an Integrated Clock Distribution Method," *Proceedings of the Conference on Design, Automation and Test in Europe*, 2007.

[34] S. Kubisch, E. Heinrich, and D. Timmermann, "A Mesochronous Network-on-Chip for an FPGA," University of Rostock, Department of CS and EE, Institute of Applied Microelectronics and Computer Engineering, Rostock, Germany, Tech. Rep., 2009.

[35] N. Genko, D. Atienza, G. D. Micheli, and L. Benini, "NoC Emulation: A Tool and Design Flow for MPSoC," *Circuits and Systems Magazine, IEEE*, vol. 7, 2007.

[36] R. Gindin, I. Cidon, and I. Keidar, "NoC-Based FPGA: Architecture and Routing," *Proceedings of the First International Symposium on Networks-on-Chip*, 2007.

[37] V. Rana, D. Atienza, M. D. Santambrogio, D. Sciuto, and G. D. Micheli, "A Reconfigurable Network-on-Chip Architecture for Optimal Multi-Processor SoC Communication," *Proceedings of the first NASA/ESA conference on Adaptive Hardware and Systems*, 2006.

[38] B. Ahmad, A. T. Erdogan, and S. Khawam, "Architecture of a Dynamically Reconfigurable NoC for Adaptive Reconfigurable MPSoC," *Proceedings of the first NASA/ESA conference on Adaptive Hardware and Systems*, 2006.

[39] A. Ehliar and J. Eilert, "A Hardware MP3 Decoder with Low Precision Floating Point Intermediate Storage," Linoping University, Tech. Rep., 2003.

[40] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Run-time Support for Heterogeneous Multitasking on Reconfigurable SoCs," *Integration, the VLSI Journal - Special Issue: Networks on chip and reconfigurable fabrics*, vol. 38, October 2004.

[41] M. Mirza-Aghatabar, S. Koohi, S. Hessabi, and M. Pedram, "An Empirical Investigation of Mesh and Torus NoC Topologies Under Different Routing Algorithms and Traffic Models," *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, 2007.

[42] "IEEE Standard for Binary Floating-Point Arithmetic," IEEE Standard Board, Tech. Rep., 1985.

[43] "LogiCORE IP Floating-Point Operator v5.0," Xilinx, Tech. Rep., March 2011.

[44] S. Manolache, P. Eles, and Z. Peng, "Task Mapping and Priority Assignment for Soft Real-Time Applications under Deadline Miss Ratio Constraints," *ACM Transactions on Embedded Computing Systems*, vol. 7, February 2008.