CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Repositório Científico do Instituto Politécnico de Lisboa

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Mestrado em Engenharia de Electrónica e Telecomunicações (Perfil de Telecomunicações)

Many-Core Approach to 2D-DCT Calculation Using an FPGA

Wilson Alexandre Borges Mália

(Licenciado em Engenharia Electrónica e Telecomunicações e de Computadores)

Trabalho Final de Mestrado para Obtenção do Grau de

Mestre em Engenharia de Electrónica e Telecomunicações

Orientador: Professor Doutor Mário Pereira Véstias

Júri:

Presidente: Professora Doutora Paula Maria Garcia Louro Antunes

Vogais: Professor Doutor José Manuel Peixoto do Nascimento

Professor Doutor Mário Pereira Véstias

December 2014

Abstract

Nowadays the need for more computing capacity has increased exponentially, requiring embedded systems to evolve and find new solutions. Due to technology limitation the single-core unavoidably was replaced by multi-core alternatives. Beside platforms like the Field-Programmable Gate Array(FPGA) provide great opportunities, it is often seen mathematical algorithms done by dedicated single-core solutions. This thesis introduces an embedded many-core architecture responsible for a 2D Discrete Cosine Transform(2D-DCT) calculation, with the goal of giving a viable alternative to the current implementations.

During this work it was necessary to develop a Network-on-a-chip, that creates the communication infrastructure responsible for connecting the dedicated cores. By analysing the 2D-DCT it was possible to implement a module that is flexible enough to enable algorithm parallelism. Each dedicated core is capable of calculating individual DCT coefficients, meaning that many-core architecture can be scaled in order to obtain different configurations, that vary in performance or resources consumption.

Resumo

Hoje em dia a necessidade computacional cresce exponencialmente, requerendo com que os sistemas embebidos estejam em constante evolução de forma a apresentar novas soluções. Devido a limitações tecnológicas o uso de um core simples foi inevitavelmente ultrapassado pelas alternativas que optam por implementações multi-core. Apesar de plataformas como a Field-Programmable Gate Array(FPGA) nos presentearem com grandes oportunidades, ainda se verifica a existência de resoluções de algoritmos matemáticos ainda recorrerem a solução dedicadas com apenas um core.

Neste documento vai-se introduzir um sistema embebido com arquitectura many-core para cálculo da Transformada discreta de cosseno bi-dimensional(2D-DCT), como alternativa viável às implementações actuais.

No decorrer deste trabalho foi necessário desenvolver uma Network-On-a-Chip(NoC), que vai criar a infraestrutura de comunicação responsável por ligar os vários módulos dedicados. Ao analise a 2D-DCT foi possível implementar um modulo suficientemente flexível que permita alcançar o paralelismo deste algoritmo. Cada core dedicado é capaz de calcular coeficientes individuais da DCT, fazendo com que a arquitectura many-core possa ser escalável com o objectivo de obter diferentes configurações, variando na performance e consumo de recursos.

Acknowledgments

I would like to thank my parents for providing me the opportunity and means to achieve my goals. They gave everything they could to ensure that I would thrive even facing difficulties and in the end here am I still proudly fighting.

I also would like to thank my brother for being a role model, because for how hard something hits, he always had the strength to keep pushing.

During my journey I've had the luck to find someone that filled my heart with passion and comfort boosting the towards fulfilling this goal. Even when I'm lazy, she had the patience to motivate me and I'm eternally grateful for that.

Last but not the least, I would like to remember every colleague, from student to professor, that accompanied me every step of the way, that challenged, teach and, in the end, made me a better person. Thank you all.

Acronyms

ASIC Application-specific integrated circuit

BRAM Block of RAM

DCT Discrete Cosine Transform

DSP Digital Signal Processing

FIFO First In First Out

FPGA Field-Programmable Gate Array

FSL Fast Simplex Link

LUT Look-Up Table

NoC Network-On-a-Chip

PLL Phased-Locked Loops

SoC System-on-a-chip

UART Universal asynchronous receiver/transmitter

Contents

Abstract i

Resumo (Portuguese) iii

Acknowledgments v

Acronyms vii

Contents xi

List of Figures xiv

List of Tables xv

1 Introduction 1

1.1 Problem Overview 2

1.2 Related Work and Motivation 3

1.3 Goals and Contributions 3

1.4 Thesis Outline 4

2 Mathematical Background 5

2.1 Discrete Cosine Transform 6

2.2 Multi Dimensional DCT 8

2.3 Decompose DCT for multi core approach . . . 13

3 System Architecture . . . 17

3.1 Overview . . . 18

3.2 Control . . . 19

3.3 Communication . . . 23

3.3.1 Router . . . 25

3.4 Calculus . . . 28

3.4.1 Coefficient Calculation . . . 29

3.4.2 Communication/Control . . . 34

4 Results . . . 37

4.1 Theoretic Analysis . . . 38

4.2 Experimental Results . . . 39

4.2.1 Performance . . . 40

4.2.2 Resources . . . 45

5 Discussion . . . 49

6 Conclusions . . . 53

7 Future work . . . 55

7.1 Processor . . . 55

7.2 Communication . . . 56

7.3 Dedicated Core . . . 56

A DCT Dedicated Core . . . 59

A.1 Moderator . . . 63

A.2 Coefficient . . . 67

A.3 Angle . . . 74

A.4 Cosine 75
A.5 Fractional Multiplier 78
A.6 Sum 79

B NoC 81
B.1 FSL Manager 88
B.2 Moderator 90

C Microblaze code 95
C.1 NoC 97
C.1.1 C file 97
C.1.2 H file 99
C.2 DCT 100
C.2.1 C file 100
C.2.2 H file 100

D System description 103

References 113

List of Figures

2.1 Cosine basis functions 7
2.2 DCT over a cosine with the same frequency as the second basic function 8
2.3 Multi dimensional basis functions 10
2.4 Multi dimensional DCT over an image resembling a basic function . 11
2.5 Bi-dimensional DCT over a realistic image 12
2.6 Image compression over a realistic image 13

3.1 Block diagram of the system 18
3.2 Microblaze peripherals diagram 20
3.3 Configuration of each dedicated 2D DCT core 21
3.4 Image sent in broadcast to every element in the network 21
3.5 Reception of the calculated 2D DCT coefficients 22
3.6 Point to point communication architecture 23
3.7 Shared bus communication architecture 23
3.8 Hierarchical bus communication architecture 24
3.9 Crossbar communication architecture 24
3.10 NoC communication architecture 25
3.11 Router block diagram 26
3.12 Router header composition 26

3.13 FSL Manager behaviour diagram 27
3.14 Moderator access sequence 28
3.15 DCT dedicated core block diagram 29
3.16 DCT core pipeline diagram 30
3.17 Cosine samples with absolute values 32
3.18 Representation of the angle's two most significant bits value 32
3.19 Representation of the number of bits required in order to keep precision(1/16) 33
3.20 Signal value according to the two MSB 33
3.21 DCT communication block's workflow 35

4.1 Test bench used for the tests 37
4.2 Atlys resources and interfaces 40

List of Tables

4.1 Single core results 41
4.2 Single core results with calculated time constraints 43
4.3 DCT calculation with different number of cores 43
4.4 Many cores results with calculated time constraints 44
4.5 NoC resources consumption 45
4.6 DCT resources consumption 46
4.7 Test bench resources consumption 47

5.1 Resource utilization of the Optimized Fast 2D-DCT hardware accelerator on the Xilinx XC2VP30 FPGA 50

1

Introduction

The computational applications greed for more performance and require new innovative solutions that are able to satisfy the goals.

The increase of throughput collided with hardware limitation, since it became harder to achieve higher frequencies using only one processor.

This originated new concurrent systems that would allow the load distribution throughout multiple processors, reducing the need of higher frequencies.

With the evolution of semiconductors it was possible to increase the resources density in embedded systems and technologies as Application-specific integrated circuit(ASIC) and Field-Programmable Gate Array(FPGA) allowed the implementation of system-on-a-chip(SoC) that comprises multi-cores, with less area involved.

Beside this evolution, there were several solutions that kept its implementation based in a single-core architecture.

This thesis presents a scalable many-core architecture with the purpose of showing the capacity and advantages that can be obtained by creating parallelism in tasks that still thrive as single-core approach. It is worth to mention that there is a distinction between a multi-core and a many-core architecture, the first one is an incremental approach of migrating designs that were already in small computers

into a single chip. In the second one, the key is the scalability, the designs tend to comprise a lot more cores that should be optimize for parallel processing[1].

There are many application that could be analysed, however as a prove of concept it was chosen the image processing area, in particular the study of the 2D-DCT algorithm.

1.1 Problem Overview

The final result of a 2D-DCT is cumulative and it depends in every computed operation. The reason why most of current implementations use a single-core architecture might be because there is still to be developed or proven that a many-core architecture can pay-off.

To implement a scalable many-core architecture it is important to decompose the 2D-DCT, in order to obtain parallelism. By doing so it is possible to develop a dedicated core that is going to be responsible for the calculus of each portion.

This component beside aiming towards performance, due to the fact that the idea is to disseminate this element in the system, it is important that it represents a low resource footprint.

The communication architecture determine in most cases, the performance of the whole system, the way every element interconnects to each other is relevant and will dictate how everything works, but this has to be done in a way that avoids the performance to be dominated by the communication. The Network-On-a-Chip(NoC) architecture provides several advantages, however creating one from its basis can represent a considerable challenge. There are various authors that investigated the phenomenons and issues around this mechanism and should be taken into consideration [2][3].

1.2 Related Work and Motivation

After analysed several works, it is possible to verify that most of them still rely in a single core approach.

A fast 2D-DCT hardware accelerator is studied in [4], consists in a single seven stages 1D-DCT pipeline able to alternate the computation for the even and odd coefficients. It also optimizes transpose operation by using special memories.

In work [5] authors have shown that by exploitation of parallelism and pipelining it is possible to develop a DCT implementation that operates at 25 frames per second.

Regarding approaches based in a many core architecture, the study [6] makes use of a programmable interconnected infrastructure, that offers scalability and deterministic communication between each core. One of the benchmarks used to validate this architecture was the bi-dimensional DCT.

The idea of breaking up the 2D-DCT and separate each coefficient in order to be calculated separately is somewhat different then the other approaches that were analysed. Using an FPGA as a blank canvas, it is possible to develop a scalable system that can be adjusted not only for solving this particular algorithm, but to be generic enough to be used for other purposes.

1.3 Goals and Contributions

The main goals and contribution of this work are:

- Implement an hardware architecture based on a many-core approach, appealing to a NoC communication architecture, that allows 2D-DCT calculation;
- Analyse the DCT algorithm and adapt it to a parallel calculus;
- Design the elementary processing elements and the router that will allow their interconnection;

- Profile the architecture to identify limitations and gather information regarding resources consumption, performance and flexibility;
- Implement a realistic scenario using a FPGA.

1.4 Thesis Outline

The remainder of this document is organized the following way:

- **Mathematical Background** introduces the mathematical operations behind DCT and demonstrates the steps required to achieve a parallel implementation
- **System Architecture** describes every module responsible that integrates in the system architecture. From the dedicated core, responsible for the DCT calculation, to the router that allows a network on chip, guaranteeing the connectivity to all cores
- **Results** present the achievements of the implemented architecture, it is analysed in performance and resource consumption
- **Discussion** compare and interpret the results, by analysing other solutions and pointing out their strengths and limitations
- **Conclusion** make an overview of the developed architecture, considering the expectations and results obtained
- **Future Work** analysis the limitations detected during the implementation and approach them with possible solution in order to reduce their effect or enhance the architecture

2

Mathematical Background

The Discrete Cosine Transform converts a signal from the spatial to the frequency domain.

This conversion is achieved by quantifying the importance of different cosine frequencies in a finite sequence of data. It means that it is possible to reconstruct the signal through the sum of the different cosines with each amplitude accordingly.

The DCT and the Discrete Fourier Transform are related algorithms, the only difference is that the first one uses only real numbers.

This transform has several applications from science to engineering. It allows the numerical solution of partial differential equations, as the reduction of data needed to describe an image, by acquiring the spatial frequency components and discarding the less important ones, that represent the higher frequencies of the image. Since the energy is more concentrated in the lower frequencies, it is achieved a compression without compromissing the data. Normally this technique is done by breaking the image in several contiguous blocks.

This document focused on the DCT over an image, not with the purpose of compressing it, however it is a possible application.

In order to understand what is behind a data compression, it is required to know the several steps to calculate a DCT. This chapter introduces the DCT, as

well as the multi dimensional DCT, and how to decompose it in a smaller form, that is going to be used in the multi core approach.

2.1 Discrete Cosine Transform

As mentioned before, the DCT can be described as the correlation of different cosine frequencies over a stream of data, $f_{(x)}$. The cosine samples are depend in the position of the data sample, x, the frequency number, u and the size of the stream, N, as seen in the following equation.

$$X_{(u)} = \sum_{x=0}^{N-1} f_{(x)} \cos(\frac{\pi(2x+1)u}{2N}), u = 0, ..., N-1 \tag{2.1}$$

Some authors further multiply the resultant matrix by $\alpha(u)$. This is done with the purpose of normalize the matrix, this way the inverse of the DCT can be computed by simply transpose the matrix.

Normally the value of the $\alpha(u)$ is define the following way:

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}}, u = 0 \\ \sqrt{\frac{2}{N}}, u \neq 0 \end{cases} \tag{2.2}$$

In order to simplify the computation of the DCT equation this factor will be omitted in the developed work.

Now lets focus in the main component of the function, in this case, the cosine. As a proof of concept lets assume we are analysing a signal with $N = 8$ samples. If we represent the several variations that occur, we obtain eight different cosine functions, shown in figure 2.1.

The cosines represented in the figure are all orthogonal to each other. This means that they are unique and if multiplied one another, the sum of all resulting samples would be zero.

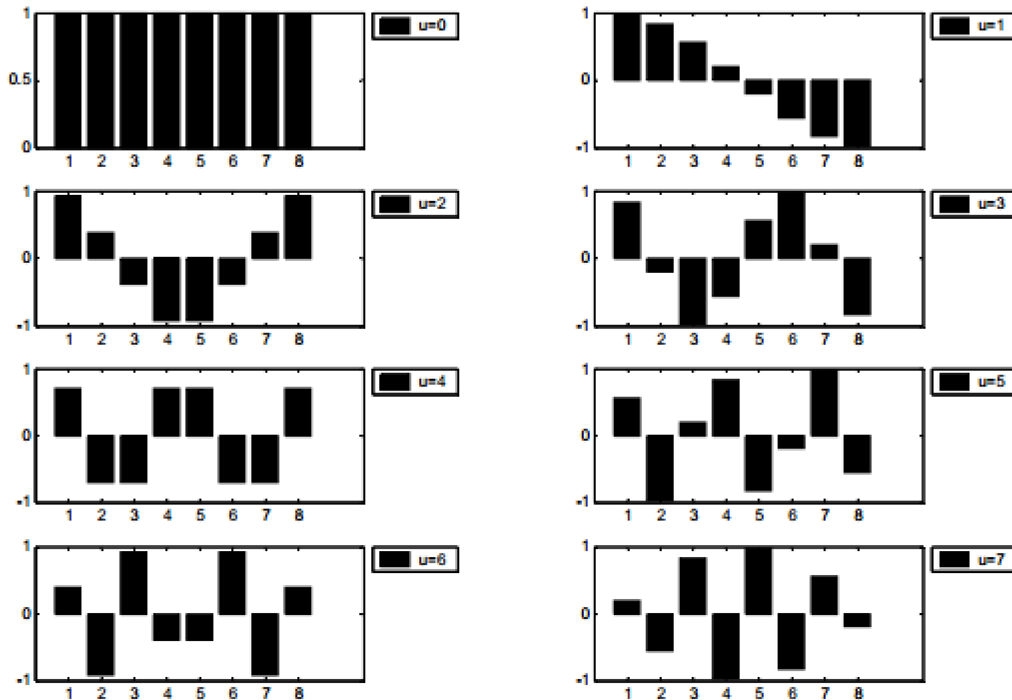

Figure 2.1. Cosine basis functions

This guarantees that none of the basis functions can be represented by combining others, making the relation between cosine frequency and final coefficient also unique.

Since this functions are immutable, to fasten calculation, they can be pre-computed, reducing the mathematical operations.

Until now we have analysed the basis functions, leading us to the conclusion that the only thing that changes is $f(x)$.

It's not transparent but the information given by the DCT is no more no less then a stream of coefficients that represent the level of correlation of the cosine basis functions with the input data. For a more visual example lets consider the figure 2.2, where the input signal is similar to the cosine given by $u = 2$.

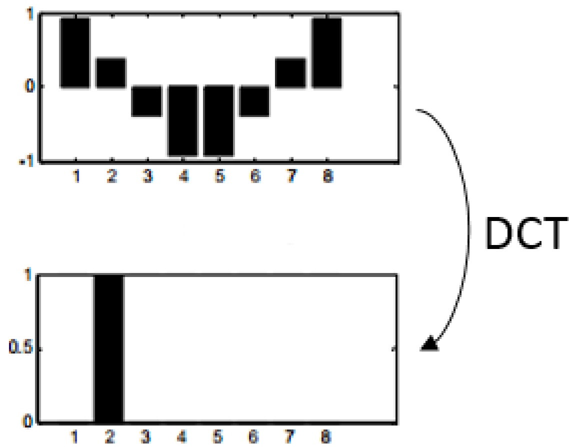

Figure 2.2. DCT over a cosine with the same frequency as the second basic function

The more similar the input data is with the cosine, the more perfect is the correlation between them. In this case, after the DCT is calculated, the only coefficient present is the one associated to the cosine which frequency is $u = 2$.

2.2 Multi Dimensional DCT

To be able to apply the same algorithm over a bi-dimensional data source(e.g. an image), it is necessary to study the 2D-DCT.

This technique is commonly used in image compression. By converting the image to its frequency spectra, it is possible to evaluate what components can be discarded, while maintaining a perceptible image. The lower coefficients have most of the image energy. So they are essential to maintain the content, while the higher frequency when discarded represents a loss of detail, resulting in an image compression.

Since the image is a bi-directional blocks of data, to obtain the resultant matrix it is necessary to multiply either horizontal and vertical cosine frequencies, with

respective u and v indexes. Each pixel $f_{(x,y)}$ can be index by its position marked by x and y values, as seen in the equation 2.3.

$$X_{(u,v)} = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f_{(x,y)} \cos(\frac{\pi(2x+1)u}{2N}) \cos(\frac{\pi(2y+1)v}{2N}) \quad (2.3)$$

For $u, v = 0, ..., N-1$.

To accomplished the 2D-DCT, first the cosine frequencies are correlated horizontally and the result is again processed vertically. The resultant matrix has the same size of the image that suffered the process.

The cosines used to perform the decomposition are known as basis functions and it is possible to calculate them previously.

This functions can be obtained with the following equation.

$$\beta_{(u,v,x,y)} = \cos(\frac{\pi(2x+1)u}{2N}) \cos(\frac{\pi(2y+1)v}{2N}) \quad (2.4)$$

It is very similar to the one presented for the single DCT. However, to perform the 2-D correlation requires the insertion of a vertical cosine.

This equation is going to generate basis function that resemble to striped figures with different tones. The figure 2.3 allows a visual interpretation of this functions.

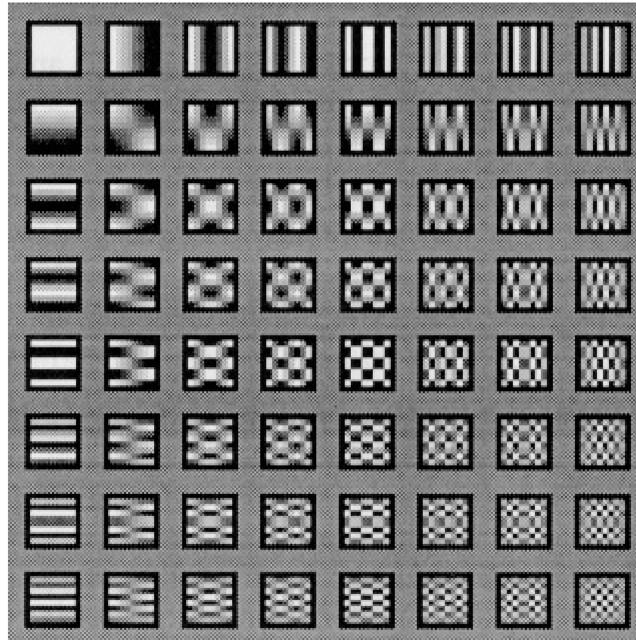

Figure 2.3. Multi dimensional basis functions

The stripes can help to understand each component, the horizontal frequencies produce a variation along the image length and the vertical component influence along the image height.

It is possible to see that the frequency increases along the coefficients, just like the single DCT, the right edge corner represents the multiplication of the higher coefficients. The first coefficient represent the DC component and the first row and column show the influence of a single cosine.

In figure 2.4 it is shown a 2D-DCT over an image resembling the basic function at $(1, 1)$ position.

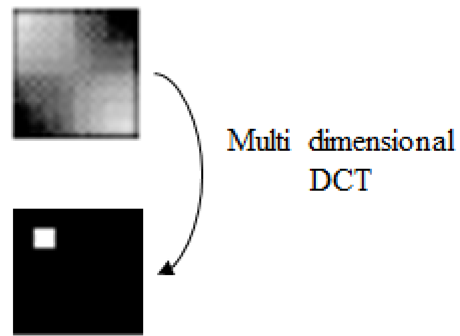

Figure 2.4. Multi dimensional DCT over an image resembling a basic function

This image has the same goal as the presented for the single DCT. If we try to correlate an image equal to a basic function, this is going to lead to a resultant 2D-DCT with only one coefficient. Considering a more realistic scenario, we have the following image and its DCT.

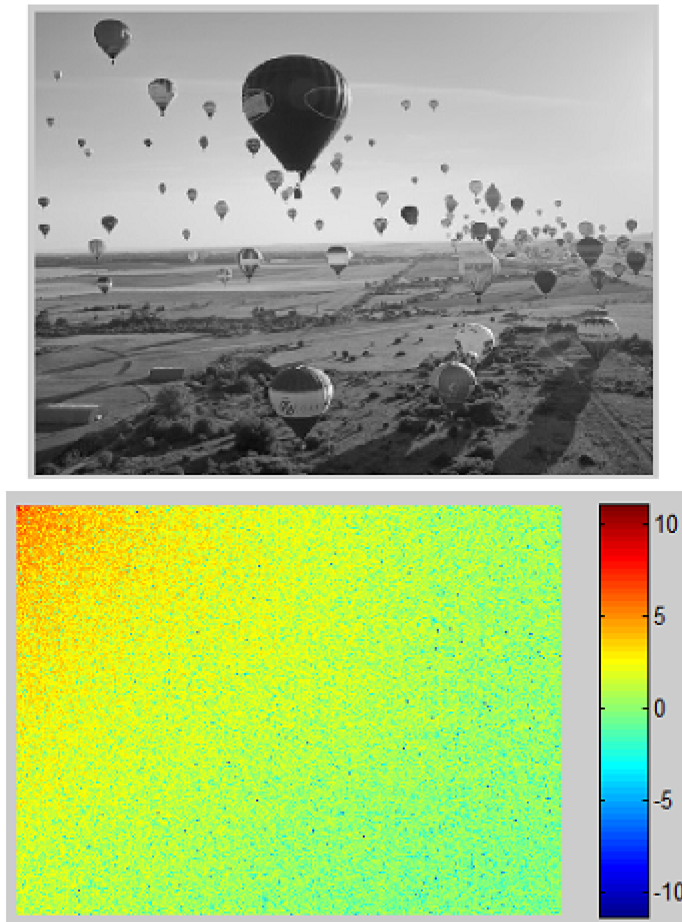

Figure 2.5. Bi-dimensional DCT over a realistic image

The most significant coefficients are the lower coefficients. As mentioned before, this is the basis for image compression, if a quantization of the DCT coefficients is applied to the DCT matrix, it would reduce the amount of information needed to represent a perceptible image. Figure 2.6 show the result of applying a threshold to the bi-dimensional DCT calculate, removing coefficients lower then a certain value.

Figure 2.6. Image compression over a realistic image

2.3 Decompose DCT for multi core approach

Before it's possible to conceive a hardware architecture, it is important to analyse the DCT algorithm and understand what can be done to improve or fasten the calculation performance.

The approach that is often used is known as the row column method. It is the same as mentioned before, where it's calculated an horizontal DCT, followed by a vertical one over the data acquired in the first one. It is a simple method, but it lacks of flexibility, because the only way to improve it, is by increasing the efficiency of the pipeline behind the calculus. A bi-dimensional DCT of an 8×8 image can be unfold into 4096 different operation, each one symbolizes the variations of pixel and frequencies, since there are 64 pixels and each coefficient is

obtained by one (u, v) combination. To simplify the future analysis, lets consider that the measuring unit is an 'operation' and the image size corresponds to a block with $N \times N$ pixels. This way the generic calculation method can be seen in the following equation.

$$
\begin{aligned}
Operations_{pixels} &= N * N \\
Operations_{frequencies} &= N * N \\
Total = Operations_{pixels} * Operations_{frequencies} &= N^4
\end{aligned}
\tag{2.5}
$$

The solution that is going to be addressed focus in the unique basis functions, represented in the equation 2.4. Instead of calculating the multi dimensional DCT in a row, we are going to split it by frequency. This is possible to achieve since the DCT result is the sum of the several frequency contribution.

It seems that we are degrading the performance, but when we split the equation, it open the possibility of parallel calculation. The latency to calculate a single frequency is exactly the same as before, however this way we can calculate simultaneous frequencies, lowering the number of iteration.

$$
\begin{aligned}
Operations_{pixels} &= N * N \\
Operations_{frequencies} &= \frac{N * N}{Number_{ofparallelfrequencies}} \\
Total = \frac{Operations_{pixels} * Operations_{frequencies}}{Number_{ofparallelfrequencies}} &= \frac{N^4}{Number_{ofparallelfrequencies}}
\end{aligned}
\tag{2.6}
$$

In the best case scenario, if we had all frequencies calculated simultaneously, it would take the same amount of time as calculation for only one frequency.

To achieve this goal, the frequencies are predefined or preconfigured, leading to the following example:

$$
\begin{aligned}
X_{(0,0)} &= \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f_{(x,y)} \\
X_{(1,1)} &= \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f_{(x,y)} \cos(\frac{\pi(2x+1)}{2N}) \cos(\frac{\pi(2y+1)}{2N}) \\
&\cdot \\
&\cdot \\
&\cdot \\
X_{(N-1,N-1)} &= \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f_{(x,y)} \cos(\frac{\pi(2x+1)N-1}{2N}) \cos(\frac{\pi(2y+1)*N-1}{2N})
\end{aligned}
\tag{2.7}
$$

Each equation seen in 2.7 is calculated simultaneously.

3

System Architecture

There are some questions that have to be answered before starting any implementation. How can we connect every element? How should the architecture be managed? How to integrate the DCT calculation in the system?

After analysing the DCT algorithm and the need of parallelism, the approach that is going to be used is a many-core architecture.

Unlike a multi-processor architecture, this one is going to be established with dedicated cores responsible for calculating DCT coefficients. By developing cores with a focused goal, it reduces the resources utilization and overhead compared with a generic processor or other generic core.

Each core gives the possibility of being parametrized, allowing them to calculate different coefficients by changing their designated frequencies. This makes possible to adapt the cores to calculate any frequency.

Taking this feature into consideration, the system can be scaled, increased or shortened, in order to achieve more efficiency or utilize less resources. Consider the following line of thoughts.

- One many-core system with eight dedicated cores;
- The DCT of an 8×8 image leads to 8×8 coefficients, in other words, it is necessary to re-utilize the cores eight times until achieving a full DCT;

- If the system had sixty-four cores, each one could be fully dedicated to one coefficient, avoiding the need of re-utilizing them, improving the system efficiency by sacrificing resources.

For how fast a dedicated core is, it relies in the communication architecture and its performance to fulfil what is promised. For this system it is going to be implemented a network on chip(NoC). This infrastructure allows the introduction of one or more router elements, that will work together in order to transport any information from one point to another. With this kind of architecture, it is easier to scale the system by integrating more routers. Since each router its limited to a number of connected cores, the network load is going to be dispersed by every element, maintaining the performance.

3.1 Overview

The architecture is going to be divided in three different blocks: communication, calculus and control(See figure 3.1).

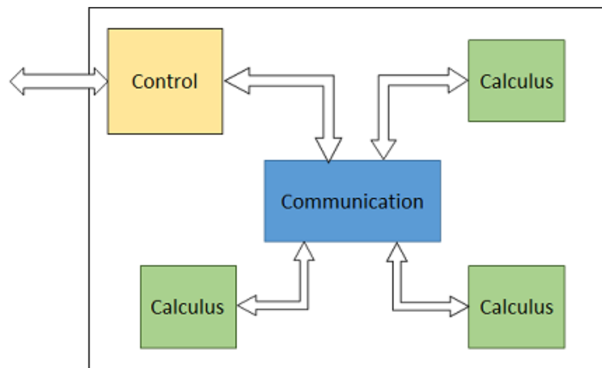

Figure 3.1. Block diagram of the system

In the figure, the main purpose of the control module is to distribute data to the dedicated cores, retrieve the respective results and compile them into one multidimensional DCT.

The communication block is composed by the NoC. It provides the ability to communicate data from the microblaze to the other elements and back. It allows flexibility to increase the network, since a router can be attached to another and expand the number of available module slots.

The third block is where the fragmented 2-D DCT is calculated. To be able to give the architecture a wide range of configurations, it is intended that this module enables a dynamic set of parameter, allowing to change the frequencies to be calculated and the limits of the image or a partial.

The following sections will explain with more detail each block and its implementation.

3.2 Control

The root of everything begins in the control module. This is where the architecture behaviour is going to be determined.

This element is implemented with a microblaze, a microprocessor provided by Xilinx. This choice was made only by convenience, since it was already integrated in the tools provided to develop the system. However there are many soft processors available that could attained the same purpose[7].

The microblaze is responsible for every management detail in this implementation. Its purpose is to assign a job to each DCT core in the network, acquire the several coefficients and gather everything in a coherent block.

In figure 3.2 it's possible to see how this module is integrated in the system.

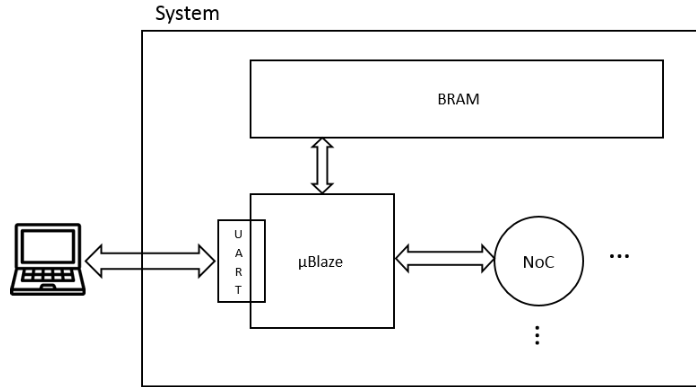

Figure 3.2. Microblaze peripherals diagram

The processor is connected to the Universal asynchronous receiver/transmitter(UART) and a block of RAM(BRAM) trough a common bus and the NoC communication is established with fast simplex links(FSLs). The FSL is a unidirectional point-to-point FIFO based communication.

The UART is the entry point of the system, allowing it to be used externally by a computer or another device that communicates via the same interface. This communication port is where the image data is acquired for further processing in the dedicated cores. While the data is received, it will be held in a dedicated memory block.

Afterwards, each DCT core is going to be configured with specific parameters, that determine the coefficient to be calculated, as seen in 3.3.

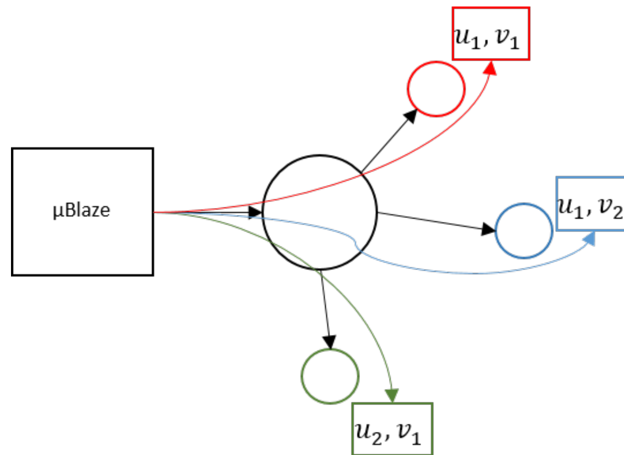

Figure 3.3. Configuration of each dedicated 2D DCT core

Since every core is properly configured with its respective frequency and image limits, the next step relies in sending a stream of data with the image information.

Instead of sending this data individually, similar to the configuration step, it's broadcasted to all elements. This mechanism reduces the latency of the communication since it is sent only one image to all cores in the network, instead of repeating the process for each one.

This phase is represented in figure 3.4.

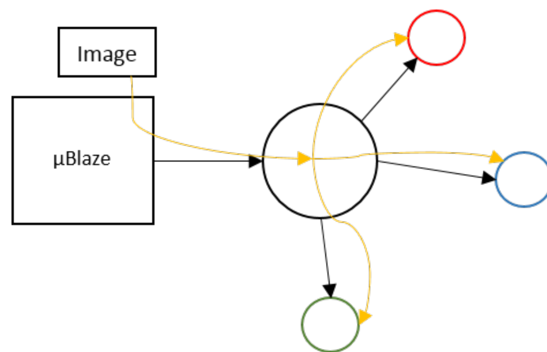

Figure 3.4. Image sent in broadcast to every element in the network

The cores receive the stream and treat it accordingly. Due to the fact that each DCT module has an internal pipeline that fastens the process, soon after everything is sent, they are ready to respond with the calculated coefficient(each one represented with a different color in the figure 3.5).

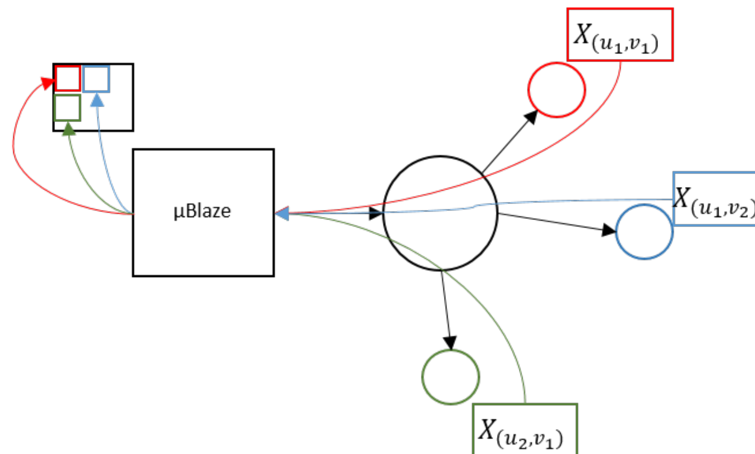

Figure 3.5. Reception of the calculated 2D DCT coefficients

Once the processor receives each coefficient, he is going to be responsible for the their organization in the final block.

As said before, it is possible to configure the dedicated cores to calculate a partial result of the image, in this case the Microblaze is going to ensure that the result is summed with the previous value that could be already in the same position in the final block.

To finalize, the calculated block is sent back to the computer or external device, ending a 2D DCT image calculation.

3.3 Communication

To backup every data transaction required by the processor, the decision regarding the communication architecture has great impact on the overall performance. Some of most used communication architectures are:

- Point-to-point - every module is directly connected to the others just like in figure 3.6. It represents the architecture with less latency. However, depending in the system size or core number, the complexity can become a bottleneck to this solution;

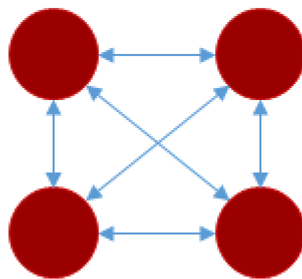

Figure 3.6. Point to point communication architecture

- Shared Bus - each core is connected to a bus, allowing them to communicate to any core connected to the bus, as seen in figure 3.7. It is a more flexible solution then the previous, but if the system complexity increases, it struggles with bandwidth efficiency, since the bus can only be used by one core at a time;

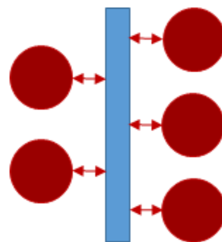

Figure 3.7. Shared bus communication architecture

- Hierarchical bus - Bus segments are connected via a bridge, as shown in figure 3.8. Protocol and structures can be varied in different bus segments and each segment may be dedicated to specific functions. The partitioning can further allow optimization of local bus architecture and the communication performance;

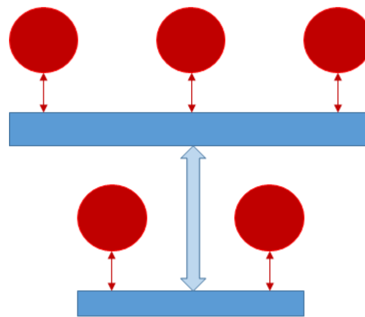

Figure 3.8. Hierarchical bus communication architecture

- Crossbar - with this architecture each source can send data to the target directly, similar to what is shown in figure 3.9. It implements a NoC stage in order to manage the connection from end-to-end.
  This approach allows more than one connection at a time, unless the same destination is already being used[8];

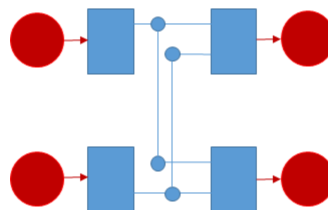

Figure 3.9. Crossbar communication architecture

- NoC - This communication architecture provides a more scalable solution, the complexity of a network can be split into several routers and it can be accessed by every core simultaneously. Figure 3.10 shows a diagram of this kind of architecture.

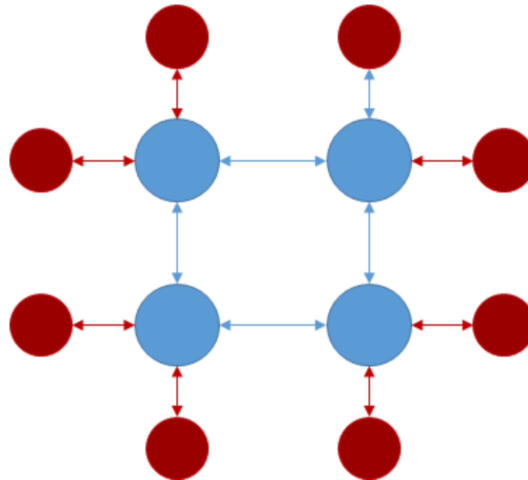

Figure 3.10. NoC communication architecture

The use of each architecture is not exclusive, they all can be used at the same time, but this approach is more likely to be seen in dedicated architectures.

For this architecture it is going to be implemented a NoC, allowing a flexible solution that will maximize the communication performance even if the system comprises many cores.

The first step towards a NoC is to implement a router able to manage all communication in one of the network node.

3.3.1 Router

This module behaviour is similar to a multiplexer, according to the information received in a port, the data is going to be redirected to the desired destination. Each router has four ports and each one can have another module or core connected via a Fast Simplex Link, FSL.

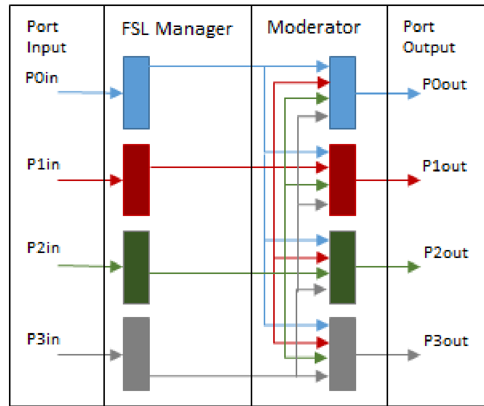

Figure 3.11. Router block diagram

To fully understand how this module works, it is important to take into consideration that, just like many communication systems, the data package is composed by a preamble that provides the required information to allow the router to send the data to the specified destination. When a package arrives in the router, it is composed by one or more headers.

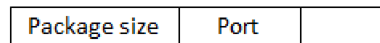

Figure 3.12. Router header composition

When a header reaches the router it is consumed. It gives the necessary information about how many more chunks of data are coming and to which port the data is going to be sent. Normally a message has one header for each router it is going to pass, this information needs to be provided by the core that originated the message.

The module responsible for dealing with the package and decompose the header is the FSL Manager.

FSL Manager

The FSL manager, just like the name indicates, is going to manage every data that arrives through the FSL. When the package arrives, the preamble is split from the message, then it is analysed and lead to a permission request to the moderator responsible for the desired destination.

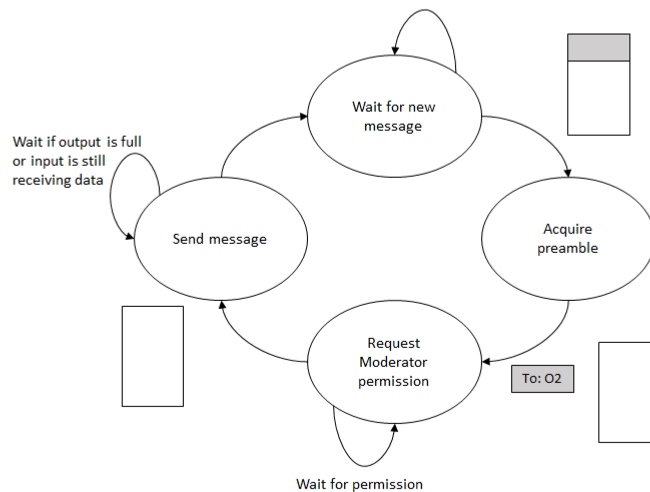

Figure 3.13. FSL Manager behaviour diagram

It is also responsible for every detail in the FSL communication, like the following:

- Wait for every portion of data that is associated to the current package;
- Ensure that the output FSL is not overrun, risking that it overflows the queue;
- Respect the read and write protocol of the FSL.

Arbiter

The router behaves like a multiplexer, connecting an input to the desired output. After the FSL manager decodes the preamble and make a request to a specific moderator, it is established the final link.

It may seem a simple task, but since it is a concurrent resource, this moderator is also responsible for controlling the access. A bad access implementation can lead to a consequence known as resource starvation.

Resource starvation often occurs in multi-tasking system, when one or more tasks keep the resource constantly occupied, while other tasks are unable to do so.

To avoid this to happen, it is implemented an access policy, based on a round robin pattern similar to the one in figure 3.14. This will guarantee that the same client, in this case a FSL manager, will only be able to use the same connection, after the moderator checks for other pending requests. Meaning that it will attend to every FSL managers one after another, avoiding that one as all the attention of this resource.

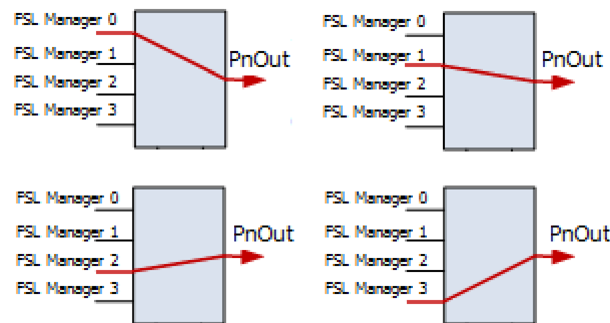

Figure 3.14. Moderator access sequence

3.4 Calculus

The main goal of this thesis is to calculate the 2D DCT in a multi core approach.

This could be done in several manners. One of the most simple ways to achieve this purpose would be using processors, programmed to calculate the desired algorithm.

It is a solution that would lead to a functional architecture, however it lacks in performance and resource consumption optimization. A processor is not made to

be a calculus dedicated core, leading to a higher latency compared to a dedicated one. With that in mind, it is necessary to implement a core that can be attached to the developed NoC and capable of giving the best performance possible towards calculating the DCT.

This core can be divided in two major blocks: communication and coefficient calculation, as seen in figure 3.15.

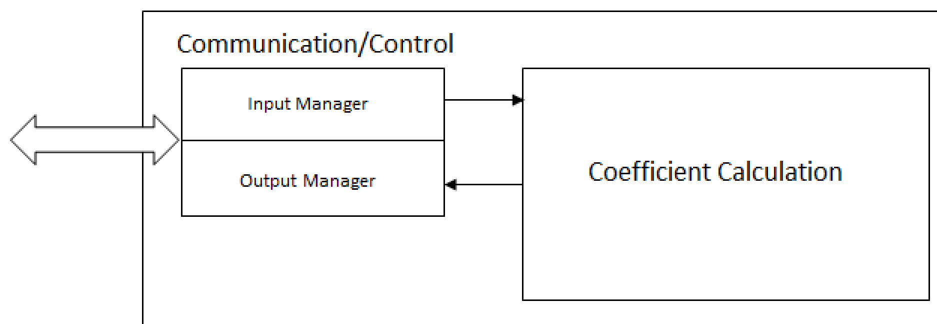

Figure 3.15. DCT dedicated core block diagram

3.4.1 Coefficient Calculation

As said before, this module is not responsible for the whole 2D DCT, but for one of its coefficients. That's the reason why it is required to be pre-configured, however it gives the flexibility to change the coefficient assigned, thus being able to be re-purposed.

This section was developed using a pipeline architecture in order to obtain a more proficient core. The several pipeline levels can be divided as follows:

- Angle;
- Cosine;
- Multiplication;
- Sum;

Each component is responsible for performing the operations needed to calculate the already mentioned 2D-DCT, equation 2.4.

It is easier to analyse the stages by taking into consideration the figure 3.16, that shows each level in more detail.

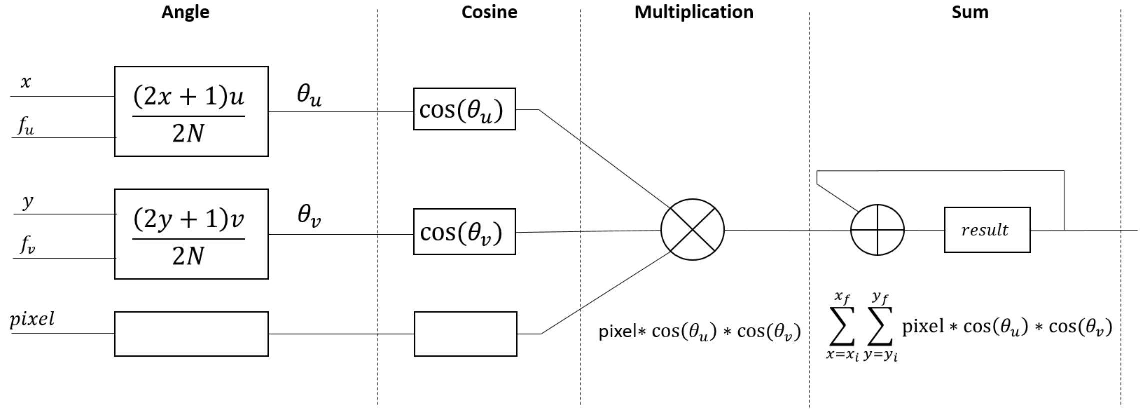

Figure 3.16. DCT core pipeline diagram

On each level it was taken into consideration several details to attain performance issues. Since the 2D DCT calculation is the main purpose of the core and the most demanding task, the latency should be reduced at maximum. Lets consider the two most prominent examples, where we can see the level of detail achieved to obtain better time constraints.

Angle Calculation

This pipe level was optimized by using numeric properties that reduce calculus, by converting them into a logic operation. By minimizing the complexity, the algorithm latency is reduced, in pair with resources consumption.

The multiplication of any number by its base, is the same as a left shift. Consider the following example in a base ten scenario.

$$2 * 10 = 20 \Leftrightarrow 2 << 1 = 20 \tag{3.1}$$

If the number is a power, the exponent of it, is equivalent to the number of shift.

$$2 * 1000 = 2 * 10^3 = 2000 \Leftrightarrow 2 << 3 = 2000 \tag{3.2}$$

The analysis was made considering a base ten, but the angle calculation is made in binary, base two. This way, the numerator in the angle calculation can be simplified by a shift, a sum by one and a multiplication. Since the first bit is zero due to the shift, it's the same as introducing a $'1'$ bit in the right side of the x value and multiplying u. In the denominator we have a similar case. To further simplify the division, it was sacrificed some flexibility and made the value N a number that is a power two related. Like with the multiplication, the division leads to a right shift.

Cosine

The cosine is a complex function and to acquire a value during a mathematical operation would simply increase the latency.

To avoid that latency and improve the performance, it was pre-calculated samples of the cosine between $[0; \pi]$.

The next step was to obtain the absolute value of the calculated samples, this way its easier to obtain the value of any angle, by adjusting the number signal according to the angle quadrant.

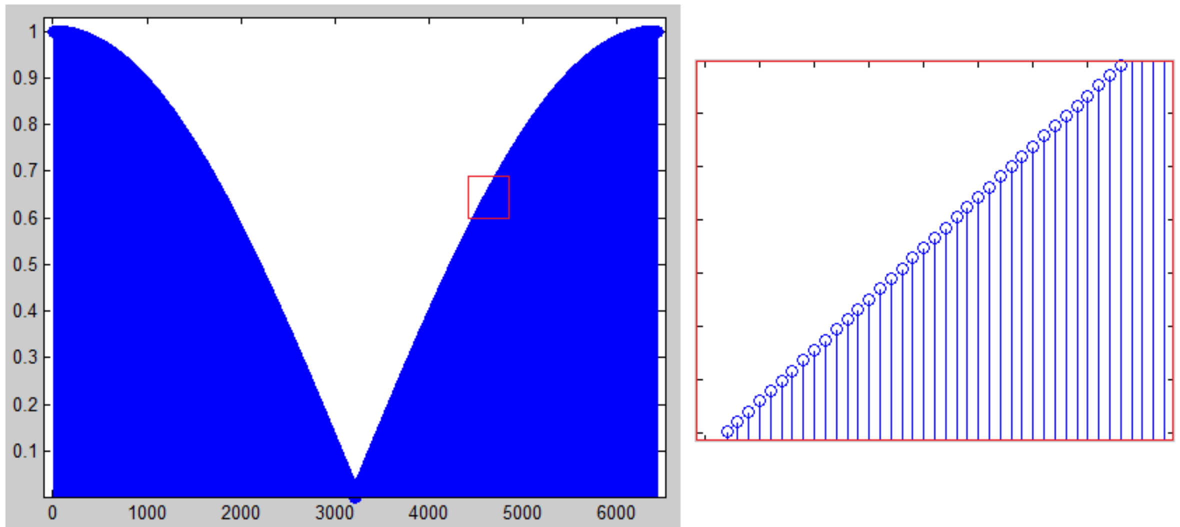

Figure 3.17. Cosine samples with absolute values

The calculated angle as a value that ranges from $[0; 2\pi[$, so the two most significant bits represent 180° and 90°.

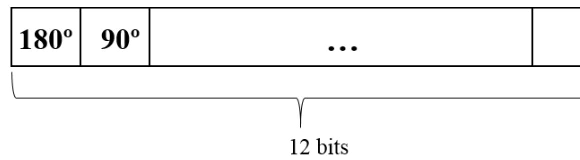

Figure 3.18. Representation of the angle's two most significant bits value

As said before, the lookup table ranges the value $[0; \pi]$, so it is required a conversion between the angle and the index value of the table. In order to perform this conversion without losing precision, the cosine expression should be evaluated for its lower value. This value is achieved when x or $y = 1$:

$$\cos(\frac{\pi(2x+1)u}{2N}) = \cos(\frac{\pi}{2N}) \leftrightarrow \theta = \frac{1}{2N} \tag{3.3}$$

Since a division of a power of 2 is similar to a right shift, the amount of bits used for the conversion should be equal to the same amount of bits used to represent

$\frac{1}{2N}$. Considering $N = 8$, the value is $\frac{1}{16} = 0.0625$, which requires $4bits$ to be represented(see figure 3.19).

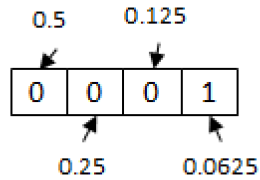

Figure 3.19. Representation of the number of bits required in order to keep precision(1/16)

In this case, the lookup table has 16 entries.

To achieve that specification, the MSB was discarded and used only for signal determination. This way we have an angle that complies with the index needed. Next step is to calculate the cosine signal. It can be obtained as represented in the following figure 3.20

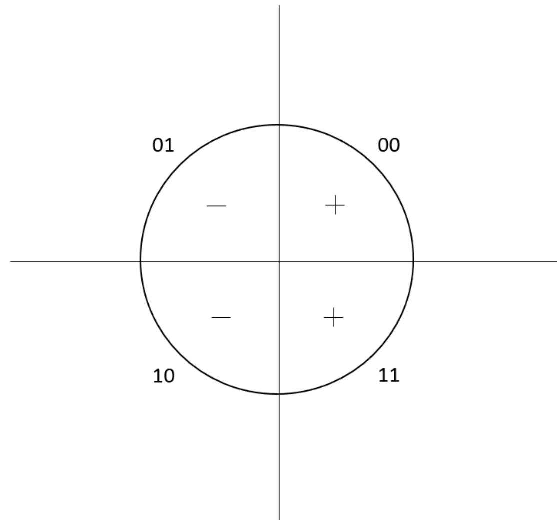

Figure 3.20. Signal value according to the two MSB

It's easier to understand the entire picture through the next equation.

$$(+/-) * cos(\theta_{[0;\pi]}) \Leftrightarrow (\theta_{180^o} \oplus \theta_{90^o}) * cos(\theta_{[0;\pi]}) \tag{3.4}$$

The final value is acquired using a lookup table to convert the angle to the absolute cosine value, afterwards, simultaneously the signal is calculated using the two MSB. This way it was possible to achieve a latency of one clock cycle.

3.4.2 Communication/Control

In every network there are required rules to allow every end device, in this case end core, to receive a message with viable information to be used. Since the DCT dedicated core is connected to a network moderator like the router, it needs to implemented an interface that complies with all the rules. The communication block is responsible to attend this specifications, dealing either with the input and the output from/to the router.

The data received is going to inform the module of what it should do, this packages can contain new configuration parameters(frequency, image limits) or pixels to enter the DCT calculation pipeline. In a normal work cycle, after configured every parameter in the core and every pixel of the image as been treated, the result is gathered, encapsulated in a network package format and sent to the control unit, in this case the Microblaze.

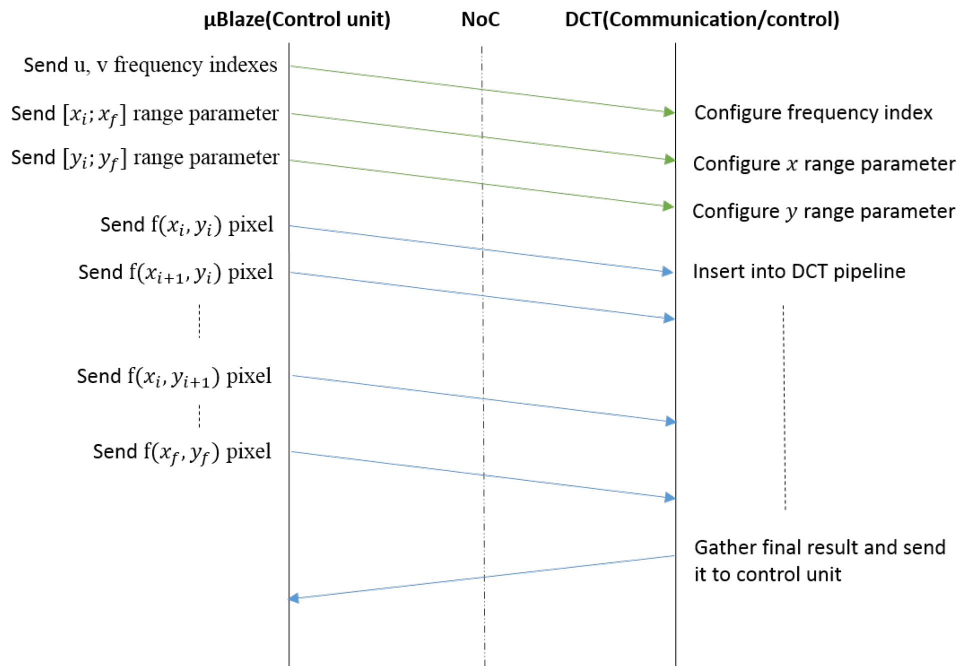

Figure 3.21. DCT communication block's workflow

4

Results

After implementing the system, it is tested on different scenarios, in order to obtain results that prove or refute the idea established around the parallelism in the DCT calculation.

The test bench that is going to be used can be seen in figure 4.1. The system is composed by a Microblaze connected to a router and three dedicated modules. All elements can communicate with each other via the NoC, using the router as the mediator.

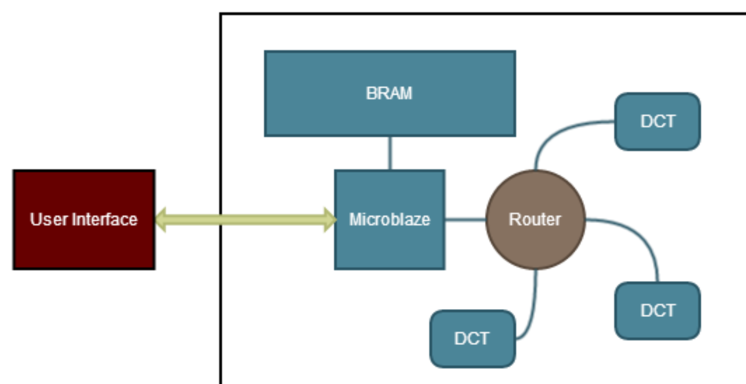

Figure 4.1. Test bench used for the tests

With this layout it is only possible to have four elements connected, but if it was necessary to increase the system size, that would be possible by connecting

more routers. This way it would exist more available slots for dedicated cores. Since each network mediator works independently from the others, the increase of network complexity wouldn't affect the node performance as much as in other communication architectures.

This chapter is split into two major section, first one intends to explain the preconceived expectations around the developed architecture, it's a baseline that allows a more critical interpretation of the experimental tests.

The second part is going to expose the results obtained by deploying the system in a real scenario. It will shows the performance benchmarks and resources consumption.

It is important to mention that beside the already mentioned Xilinx tools, used during the system development and deployment, it was also used Matlab. It is important to have a comparison method that can guarantee result verification.

4.1 Theoretic Analysis

Similar to what was told during the Mathematical background chapter, creating parallelism in the DCT algorithm, allows a system to outperform a single threaded architecture. By calculating coefficients simultaneously this is going to reduce the time required to achieve the final result.

This way it is expected to see that the final time spent calculating the DCT with one core is going to divide for each extra core existent in the system. The results can be seen in the following table, that connects the number of cores with the 2D-DCT calculation time, $\mathit{\Delta} DCT$.

$$\mathit{\Delta} DCT = \frac{\mathit{\Delta} DCT_{1core}}{N_{cores}} \tag{4.1}$$

The performance factor can be the most prominent, however it is important to analyse this solution by its resource consumption. It is expected to obtain an inverse result to what was seen in the previous table, the resources should increase with a multiplicity similar to the number of core in the system.

4.2 Experimental Results

The results depend on what hardware is used to deploy this architecture, since the main goal is to integrate it on a FPGA, it is important to give the full detail about that base.

The FPGA used was a Spartan-6 XC6SLX45, it is optimized for high-performance logic and offers the following characteristics:

- 6,822 slices each containing four 6-input LUTs and eight flip-flops;
- 2.1Mbits of fast block RAM;
- 4 clock tiles (8 DCMs and 4 Phased-Locked Loops(PLLs));
- 6 PLLs;
- 58 DSP slices;
- 500MHz+ clock speeds.

Since during the hardware implementation some components were limited to 100MHz clock frequency, it was decided to use this value for the entire system, instead of using the board's 500MHz clock, to avoid unforeseen synchronization problems with all elements.

This FPGA makes part of a Digilent board called Atlys, that provide a lot of other interfaces that can be useful for debug or user interface, such as USB-UART. This make possible the acquisition of data that is going to be used to analyse the system performance.

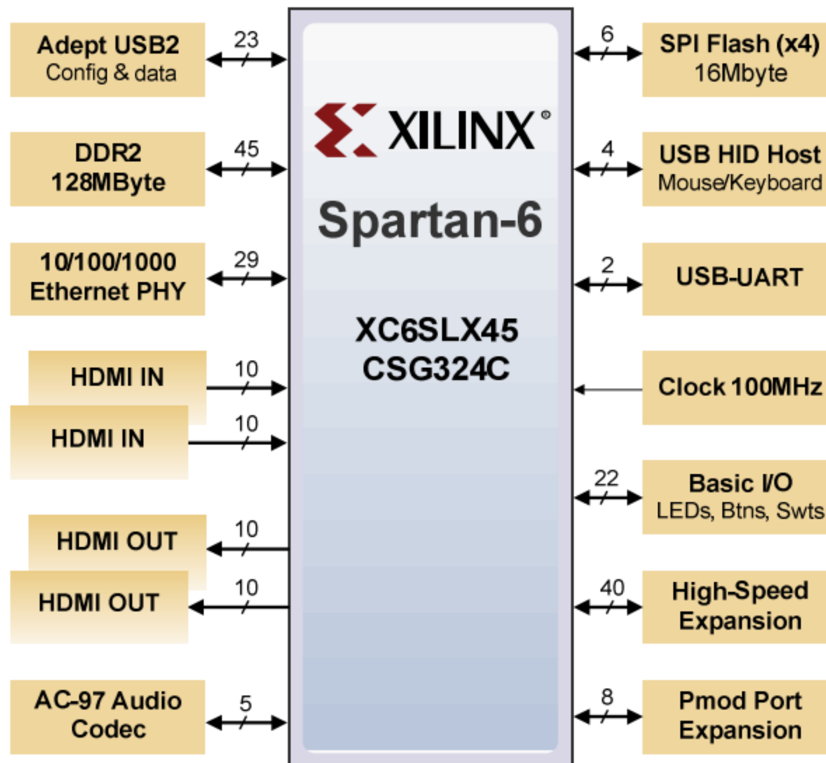

Figure 4.2. Atlys resources and interfaces

4.2.1 Performance

In order to obtain enough information to compare with the theoretic analysis and determine more accurately how the system performs, the results were divided in two steps, single and many-core.

The first one has all information regarding to the time constraints achieved by only one core. This measures were taken using different data sizes, this way it is possible to correlate the number of pixels with the time that took to acquire the result.

The second step consists in a many core analysis, where it is possible to determine if the DCT parallelism achieves the desired impact, reducing the time spent calculating for each extra core in the system.

For every following tests, consider a frame with an $8x8$ resolution.

Single Core

Each dedicated core can be defined to calculate one DCT coefficient, but to understand the singular performance of each element it is important to run several tests in order to obtain a more accurate benchmark.

The following tests consists in sending packets with different number of pixels and calculate the partial coefficient that is associated with the data sent. The results obtained can be seen in the following table:

Table 4.1. Single core results

Scenario	Number of pixels	Total time
1 Pixel	1	1,830μs
1 Line	8	2,6789μs
Half Frame	32	6,789μs
Full Frame	64	11,899μs

It is possible to see that there isn't a perfect multiplicity in the results, the relation between the first and last row should be $R_1 = \frac{R_4}{64}$, being R_n the row index in the table.

It's worth to notice that the result represent more than the calculation time, it takes into account the delay in data processing, transmission and reception. This can temper the values doesn't match with the expectations.

Due to the fact that almost every step that occurs inside the system is outside the monitoring range, the impact of each activity can't be acquired directly. However, considering the different measures, it is possible to estimate the impact of the Microblaze in the final result.

The latency introduced by the processor corresponds to the time that takes to build the configuration header for the dedicated module, with the definition of

each parameter. Afterwards, each pixel sent to the FSL is going to be simultaneously treated by the router. This data transmission offset can be considered as communication overhead instead of processing time.

Knowing this it is possible to assume that the latency introduced by the Microblaze is a constant, $\Delta_{\mu blaze}$, and the only variable in equation is the pixel number, N_{px}, that is going to directly affect the transmission and calculation time, $\Delta_{calc\&com}$.

Considering the described system architecture, the most important constraints regard to the communication and calculation time. The Microblaze is only responsible for supporting the architecture and could be replaced in order to reduce the impact of its latency in the final result.

In sum, the total coefficient time, Δ_{coeff}, can be represented as the following equation:

$$\Delta_{coeff} = \Delta_{\mu blaze} + N_{px} * \Delta_{calc\&com} \tag{4.2}$$

Taking the first and last line by example, it is possible to calculate the weight of each parcel.

$$\begin{cases} 1.830\mu s = \Delta_{\mu blaze} + \Delta_{calc\&com} \\ 11.899\mu s = \Delta_{\mu blaze} + 64\Delta_{calc\&com} \end{cases} \Leftrightarrow \begin{cases} \Delta_{\mu blaze} = 1.6702\mu s \\ \Delta_{calc\&com} = 0.1598\mu s \end{cases} \tag{4.3}$$

By isolating the Microblaze influence from the rest of the result, the values appear to be more like what was anticipated, as seen in the following table.

Table 4.2. Single core results with calculated time constraints

Scenario	N_{px}	$\Delta_{calc\&com}$	$\Delta_{\mu blaze} * N_{px}\Delta_{calc\&com}$
1 Pixel	1	$0.1598\mu s$	$1.830\mu s$
1 Line	8	$1.2784\mu s$	$2.9486\mu s$
Half Frame	32	$5.1136\mu s$	$6.7838\mu s$
Full Frame	64	$10.2288\mu s$	$11.899\mu s$

Many Core

After a more detailed analysis over the dedicated core, it's time to verify the system behaviour when calculating a full DCT. The following table shows the values obtained for each test, where the variable in question is the number of dedicated cores used.

Table 4.3. DCT calculation with different number of cores

N_{cores}	Δ_{DCT}	Δ_{coeff}
1	$758{,}719\mu s$	$11.84\mu s$
2	$422{,}690\mu s$	$6.6\mu s$
3	$335{,}890\mu s$	$5.23\mu s$

It is possible to see that the ratio $\frac{\Delta t}{coefficient}$ obtained with one core matches the result in the last line of the Table 4.1.

Similar to what was seen in the previous section, the result don't match with the expectations. With two and three cores should be, respectively, a half and one third of the time.

Even in a great scale, it is possible to see the influence of the time spent in package pre processing. In order to obtain more representative results, consider the calculated $\Delta_{\mu blaze}$ and $\Delta_{calc\&com}$ values. The Microblaze processing took about $1.6702\mu s$ for one coefficient. A full DCT is composed by 64 coefficients, leading to an accumulated offset of $106.89\mu s$.

Since every data is sent simultaneously to every core, the impact is the same even if the number of dedicated cores change. The time that takes to calculate the DCT can be represented the following way.

$$\Delta_{DCT} = N_{coeff}(\Delta_{\mu blaze} * \frac{N_{px}\Delta_{calc\&com}}{N_{cores}}) \tag{4.4}$$

In order to acquire the impact of each parcel, consider a full DCT scenario($N_{px} = N_{coeff} = 64$), with the Δ_{DCT} obtained in the first and third line of the Table 4.3.

$$\begin{cases} 758.719\mu s = 64(\Delta_{\mu blaze} * 64\Delta_{calc\&com}) \\ 335.89\mu s = 64(\Delta_{\mu blaze} * \frac{64\Delta_{calc\&com}}{3}) \end{cases} \Leftrightarrow \begin{cases} \Delta_{\mu blaze} = 1.998\mu s \\ \Delta_{calc\&com} = 0.154\mu s \end{cases} \tag{4.5}$$

It is possible to notice some similarities with the single core calculations for one coefficient. The communication time is approximately equal as it should, since every data that circulates in the network goes to each core simultaneously.

The time spent processing data in the Microblaze it is higher, however in this test each coefficient that arrives is verified from which core it came and gathered in the final frame that has the final DCT.

With this values it is possible to compile a new table, that shows the results with the calculated portions.

Table 4.4. Many cores results with calculated time constraints

N_{cores}	N_{coeff}	N_{px}	Δ_{DCT}	$\Delta_{DCT} - \Delta_{\mu blaze}$
1	64	64	$758.719\mu s$	$630.847\mu s$
2	64	64	$443.264\mu s$	$315.392\mu s$
3	64	64	$335.89\mu s$	$208.018\mu s$

It is merely symbolic, because the data acquired via calculation is not 100% accurate, however it shows that even with some approximation, the time that took to calculate the DCT without the influence of the processor matches the expectation.

$$\Delta_{DCT_{L_2}} - \Delta_{\mu blaze} \simeq \frac{\Delta_{DCT_{L_1}} - \Delta_{\mu blaze}}{2} \leftrightarrow \Delta_{DCT_{L_3}} - \Delta_{\mu blaze} \simeq \frac{\Delta_{DCT_{L_1}} - \Delta_{\mu blaze}}{3} \tag{4.6}$$

4.2.2 Resources

The developed architecture intends to be flexible enough to adjust the system size in order to match with the desired specifications. It is important to know the impact of each core in the overall resources. This evaluation is going to be made taking into consideration four key resources: Flip-flops, LUTs(Look-up tables), DSP(Digital signal processing) and RAM blocks(BRAM). It is going to be analysed the maximum frequency possible for each core.

Starting with the main element in the NoC communication architecture, the router, it is possible to see in the Table 4.5, the consumption of this core element in the Atlys board.

Table 4.5. NoC resources consumption

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	140	54576	0%
Number of Slice LUTs	394	27288	1%

The router works like a multiplexer and an encoder, it was expected to consume a small amount of resources, leaving a small footprint in the FPGA. Since each

router is able to connected four other cores, it's safe to assume that even in a large system, the impact would be reduced. After synthesised, this core presents a frequency constraint of 257.185 MHz. The signal transition within the circuit isn't instantaneous. In order to have every signal stabilized each clock cycle, the maximum propagation speed cannot exceed this frequency value.

The 2D-DCT dedicated core performs complex mathematical operations, requiring more resources in order to achieve its goal.

Table 4.6. DCT resources consumption

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	281	54576	0%
Number of Slice LUTs	226	27288	0%
Number of Block RAM/FIFO	1	116	0%
Number of DSP48A1s	2	58	3%

It is possible to see in Table 4.6 the impact caused in the FPGA. The main reason is the several multiplication that occur inside and the table that contains the pre-calculated cosine. The complexity also as an impact in the frequency constraint, resulting in a maximum of 105.702 MHz Even with a more considerable amount of resources consumed, it is possible to increase the system size with a great number of this cores until filling all the capacity.

During all the tests it was used the test bench seen in figure 4.1. This scenario has the weight of the Microblaze and auxiliary BRAM, leading to extra resources consumed as seen in the Table 4.7.

Table 4.7. Test bench resources consumption

Device Utilization Summary (actual values)			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	3,877	54,576	7%
Number of Slice LUTs	4,484	27,288	16%
Number of DSP48A1s	9	58	15%
Number of RAMB16BWERs	29	116	25%
Number of RAMB8BWERs	0	232	0%

As expected the higher percentage lies in the BRAM consumption, because the code that runs inside the processor is stored in local memory.

5

Discussion

The developed architecture intended to optimize an algorithm by splitting it in several calculation streams. Each stream can be parallelized in order to be executed simultaneously to others and achieve a faster result.

The algorithm used to prove this concept was the 2D-DCT, a complex mathematical algorithm that is often used in image processing.

It was possible to verify that by increasing the parallelism, the result acquisition occurred faster. Even achieving an increase of performance, it didn't directly met with the expectations. The overhead introduced by the processor and the initial configuration of each dedicated core tempered with the total time that takes to calculate the full 2D-DCT. By isolating the impact of each component, it was possible to see that the result was more similar then it appeared to be, resulting in a time multiplicity where two dedicated cores outperform one by two times and the same logic for three cores.

Since this architecture is versatile, there are many configurations possible that imply different use cases. The low-end system, composed with only one dedicated core, presents a small resource footprint, meaning that even low density embedded systems are able to use this architecture and achieve the same goal, in exchange of performance.

An high-end system is going to approach the number of dedicated cores in the network to the number of coefficients, improving performance by occupying more area.

There are several works around the 2D-DCT algorithm in order to achieve higher throughput, but besides the final algorithm solution there aren't much more similarities. Most of the implementation focus in a single core architecture, responsible for the complete calculus. This approach presents big advantages by comparison to the parallelism solution, since it concentrates the optimizations in a single core, avoiding the need of a communication infrastructure. However it reduces the flexibility and scalability of the solution and it becomes harder to overcome current architectures since most of the single core approaches have already been analysed.

For example, lets consider the pipelined Fast 2D-DCT Accelerator [4], mentioned during the introduction. By implementing a pipelined fast DCT and decomposing the algorithm in two single DCTs, they are able to achieve the final result in 80 clock cycles, with a 107 MHz clock, leading to a $\Delta_{DCT} = 0.75\mu s$. This mark is far better then the presented in the previous chapter, but in exchange of FPGA resources consumption. It is mentioned that they have an occupation 2.5 times higher then the IP core provided by Xilinx.

Table 5.1. Resource utilization of the Optimized Fast 2D-DCT hardware accelerator on the Xilinx XC2VP30 FPGA

Resource	Used	Available	Utilization
Slices	2823	13696	20%
Slice Flip Flop	3431	27392	12%
4 input LUTs	2618	27392	9%

Even using a different chip, it is possible to compare the resources consumption from the Table 4.6 and Table 5.1. There is a significant difference between this two implementations.

Considering the occupation versus performance, in comparison with the three dedicated core solution, their implementation presents better result, but less flexibility then being able to adjust amount of cores. If the multiplicity verifies it self, when implementing a 64 DCT core system, it would to be expected to achieve a mark of $9.86 \mu s$, without the Microblaze influence. With 64 elements in the system, they would overwhelm the FPGA resources, in specific the DSPs.

In sum, it is important to analyse each architecture and understand where they fit. When the 2D-DCT is decomposed and parallelized as described in this document, it presents a solution which base consists in a generic network that can be readjusted in order to achieve the requirements and reused with other algorithms.

It is important to mention a "hidden" feature that the 2D-DCT parallelism provides. The DCT algorithm is often used in JPEG image compression. This is done by removing less important coefficients, reducing the image resolution, therefore less data. The coefficients that are removed or ignored can be chosen through different techniques that imply a quantization over the resultant 2D-DCT.

In the last technique, calculating a full DCT with the purpose of discarding part of its information, it seems very inefficient compared to being able to only calculate certain coefficients.

With an 2D-DCT parallelism architecture it is possible to process an image and filter the desired coefficients, instead of performing the full algorithm.

6

Conclusions

The world became more demanding and required that the technology followed the same steps.

The processor evolved in a way that led to a physical and economical barrier, it became harder to achieve better performances due to hardware cost and limitation. This forced the introduction of new technologies and architectures based in cooperative systems with multiple processing units.

The idea behind this solution is logic, by creating parallelism, instead of using brute force with one high-end core, the task is split and distributed by several elements, achieving similar or better results then with the previous system.

Following this logic, this document intends to mirror this behaviour, but based in complex mathematical algorithms. The algorithm adopted was the 2D-DCT since it often use in image compression and it is significantly demanding to be calculated.

Most of the solutions presented are based in a dedicated single core, that normally has a great throughput in exchange of high resource consumption. The 2D-DCT can be calculated resorting to two 1D-DCTs, using this architecture demands that the task is fully performed by only one core.

The proposed solution intended to take this algorithm into the next level. Decomposing the 2D-DCT, it was possible to achieve the same result by calculating

every DCT coefficient in different dedicated cores. Since each core works independently, with this parallelism it is possible to acquire multiple coefficients simultaneously. This led to a scalable system that interconnects every endpoint via a NoC communication architecture.

This communication grid guarantees a growth sustainability, presenting the system with expandable and relatively low latency infrastructure.

When this architecture was introduced in a System-on-Chip using a FPGA it was possible to develop multiple scenarios with different number of dedicated cores. In every test, the main goal was to verify multiplicity between the total latency, but it appeared that the Microblaze that fed the network was introducing an undesirable offset. After some analysis it was possible to reach into a conclusion.

When the offset is removed from the equation, the calculation latency revealed to be proportionally decreasing when increasing the number of dedicated cores in the network.

It is possible to acknowledge that a single core solution can still retrieve a great performance, however parallelizing the tasks opens new possibilities, allowing a more flexible solution that can be tuned considering the application requirements. Beside the results obtained leaning to the continuous use of a single dedicated core, there is one aspect that this architecture overpower the other. By having the possibility of calculate specific coefficients, the task of performing image compression, resorting to a coefficient filter, can be done without the need of obtaining the complete 2D-DCT.

Overall, it was implemented an architecture that provides a scalable infrastructure, with the intend of parallelizing tasks or algorithms to boost performance and simultaneously giving a reduced resources footprint, depending on the system configuration.

7

Future work

During this thesis it was possible to achieve the main goal, however there are some points to take into consideration in order to improve this architecture in the future.

This chapter focus in describing what can been done to achieve better results with this architecture.

7.1 Processor

In order to calculate the DCT, it was required to have some entry point of the data and it was used the Microblaze, in order to reduced the time spent developing a core that could communicate with an external program or device.

During the tests this option revealed an unconsidered consequence, because it introduced an offset into the final results.

After some analysis, there were two different approaches that could solve or minimize this problem. The first one, and less intrusive, is to increase the processor complexity and improve the code quality in order to obtain better performance. This measure should reduce the impact, but this one would still considerable.

The second solution and more efficient would require the implementation of a dedicated core. This module would met every requirements and since it is dedicated to a specific task, every clock cycle would be optimized. Receive data to store in

memory, stream the image to the network, receive the coefficients and put them together to be sent back.

7.2 Communication

The glue to all the cores in the network is the NoC. The router can be used to communicate in several ways, from unicast, for core configuration and coefficient retrieval, to multicast or broadcast, for image sending.

The last feature is presented with a faulty condition, because it is only possible to perform a broadcast in a specific node/router, making impossible to create a bigger network and keep the same behaviour. It is important to give more intelligence to this core, in order to be able to identify a broadcast message and propagate it to the other router that it may be connected.

7.3 Dedicated Core

The heart of all calculation is made inside the DCT dedicated core and here every operation counts in order to achieve a compromise between performance and resources consumption. Inside each module there is a look-up table that associates the angle with the respective cosine value. However to acquire the angle to be converted it requires the use of several operations that include multiplications.

The cosine frequencies can be orthogonal to each others, however this doesn't that they can't be related. In fact there are several samples coincide in different frequencies.

The idea is to analyse this relations and verify the possibility of reducing the consumed memory in the look-up table by replacing the $\cos(\frac{\pi(2x+1)u}{2N})$ operation for a multiplexer able to relate (x, u) with a final value.

This wouldn't only reduce the computation cycles, but also the resources consumed.

A

DCT Dedicated Core

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity dct is
generic (
        ID                          : integer := 0;

        N                           : integer := 3;

        PIXEL_SIZE          : integer := 8;
        PARAM_SIZE          : integer := 10;
        CONFIG_SIZE     : integer := 4;

        LOAD_X_POS          : integer := 31;
        LOAD_Y_POS          : integer := 30;
        LOAD_FREQ_POS   : integer := 29;
        RESET_COEFF_POS : integer := 28
);
Port (
        -- NoC Bus
        FSL_Clk         : in  std_logic;
        FSL_Rst         : in  std_logic;
        FSL_S_Read      : out std_logic;
        FSL_S_Data      : in  std_logic_vector(0 to 31);
        FSL_S_Exists  : in    std_logic;
        FSL_M_Write     : out std_logic;
        FSL_M_Data      : out std_logic_vector(0 to 31);
        FSL_M_Full      : in  std_logic
);
```

```
end dct;

architecture Behavioral of dct is

component moderator
Generic(
        ID                : integer := 0;

        PIXEL_SIZE        : integer := 8;
        PARAM_SIZE        : integer := 10;
        CONFIG_SIZE       : integer := 4;

        LOAD_X_POS              : integer := 31;
        LOAD_Y_POS              : integer := 30;
        LOAD_FREQ_POS     : integer := 29;
        RESET_COEFF_POS : integer := 28
);
Port (
        clock    :        in       std_logic;
        reset    :        in       std_logic;

        -- NoC Bus
        FSL_S_Read          : out std_logic;
        FSL_S_Data          : in  std_logic_vector(0 to 31);
        FSL_S_Exists   : in       std_logic;

        FSL_M_Write         : out std_logic;
        FSL_M_Data          : out std_logic_vector(0 to 31);
        FSL_M_Full          : in  std_logic;

        -- Coefficient calculus
        coeff_ready       : in std_logic;
        coeff_enable  : out std_logic;
        coeff_reset       :        out std_logic;

        pixel             : out std_logic_vector(PIXEL_SIZE - 1 downto 0);
        parameterA        : out std_logic_vector(PARAM_SIZE - 1 downto 0);
        parameterB        : out std_logic_vector(PARAM_SIZE - 1 downto 0);

        load_x            : out std_logic;
        load_y            : out std_logic;
```

```
        load_freq        : out std_logic;

        coeff    : in std_logic_vector(23 downto 0)
);
end component;

component coefficient
Generic(
        N          : integer := 3;
        PIXEL_SIZE : integer := 8;
        PARAM_SIZE : integer := 10
);
Port (
        clock     :       in       std_logic;
        enable  :         in std_logic;
        reset     :       in       std_logic;

        pixel            : in std_logic_vector(PIXEL_SIZE - 1 downto 0);
        parameterA       : in std_logic_vector(PARAM_SIZE - 1 downto 0);
        parameterB       : in std_logic_vector(PARAM_SIZE - 1 downto 0);

        load_x           : in std_logic;
        load_y           : in std_logic;
        load_freq        : in std_logic;

        ready    : out std_logic;
        coeff    : out    std_logic_vector(23 downto 0)
);
end component;

signal coeff_clock : std_logic;
---------------- FSL BUS
signal FSL_S_Read_internal          : std_logic;
signal FSL_S_Data_internal          : std_logic_vector(31 downto 0);
signal FSL_S_Exists_internal  : std_logic;
signal FSL_M_Write_internal   : std_logic;
signal FSL_M_Data_internal          : std_logic_vector(31 downto 0);
signal FSL_M_Full_internal          : std_logic;
-------------------------------------------------------
signal enable                : std_logic;
signal ready                 : std_logic;
```

```
signal requestPixel : std_logic;
signal clear          : std_logic;
signal hasPixel       : std_logic;

signal parameterA     : std_logic_vector(PARAM_SIZE - 1 downto 0);
signal parameterB     : std_logic_vector(PARAM_SIZE - 1 downto 0);
signal load_x         : std_logic;
signal load_y         : std_logic;
signal load_freq      : std_logic;
signal high_low :std_logic;

signal pixel : std_logic_vector(7 downto 0);
signal coeff : std_logic_vector(23 downto 0);

begin

FSL_M_Data <= FSL_M_Data_internal;

moderator_unit : moderator
generic map(
        ID => ID
)
port map(
        clock => FSL_Clk,
        reset => FSL_Rst,

        -- NoC Bus
        FSL_S_Read      => FSL_S_Read,
        FSL_S_Data      => FSL_S_Data,--_internal,
        FSL_S_Exists => FSL_S_Exists,

        FSL_M_Write     => FSL_M_Write,
        FSL_M_Data      => FSL_M_Data_internal,
        FSL_M_Full      => FSL_M_Full,

        -- Coefficient calculus
        coeff_ready     => ready,
        coeff_enable => enable,
        coeff_reset     => clear,

        pixel           => pixel,
```

```
        parameterA => parameterA,
        parameterB => parameterB,

        load_x    => load_x,
        load_y    => load_y,
        load_freq => load_freq,

        coeff     => coeff
);

coeff_unit : coefficient
generic map(
        N => N
)
port map(
        clock     => FSL_Clk,
        enable => enable,
        reset     => clear,

        pixel     => pixel,
        parameterA => parameterA,
        parameterB => parameterB,

        load_x    => load_x,
        load_y    => load_y,
        load_freq => load_freq,

        ready     => ready,
        coeff     => coeff
);

end Behavioral;
```

A.1 Moderator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.all;

entity moderator is
Generic(
```

```
        ID                                    : integer := 0;

        PIXEL_SIZE      : integer := 8;
        PARAM_SIZE      : integer := 10;

        CONFIG_SIZE      : integer := 4;
        LOAD_X_POS              : integer := 31;
        LOAD_Y_POS              : integer := 30;
        LOAD_FREQ_POS    : integer := 29;
        RESET_COEFF_POS : integer := 28
);
Port (
        clock    :       in       std_logic;
        reset    :       in       std_logic;

        -- NoC Bus
        FSL_S_Read          : out std_logic;
        FSL_S_Data          : in  std_logic_vector(0 to 31);
        FSL_S_Exists   : in       std_logic;

        FSL_M_Write         : out std_logic;
        FSL_M_Data          : out std_logic_vector(0 to 31);
        FSL_M_Full          : in  std_logic;

        -- Coefficient calculus
        coeff_ready      : in std_logic;
        coeff_enable  : out std_logic;
        coeff_reset      :       out std_logic;

        pixel           : out std_logic_vector(PIXEL_SIZE - 1 downto 0);
        parameterA      : out std_logic_vector(PARAM_SIZE - 1 downto 0);
        parameterB      : out std_logic_vector(PARAM_SIZE - 1 downto 0);

        load_x          : out std_logic;
        load_y          : out std_logic;
        load_freq       : out std_logic;

        coeff    : in std_logic_vector(23 downto 0)
);
end moderator;
```

```
architecture Behavioral of moderator is

type STATE_TYPE is (Idle, Write_Header, Write_Data);
signal txState : STATE_TYPE;

signal newData    : std_logic;
signal fsl_buf        : std_logic_vector(0 to 31);
signal fsl_buf_inv    : std_logic_vector(31 downto 0);
signal coeffReg   : std_logic_vector(23 downto 0);

function reverse_any_vector (a: in std_logic_vector)
return std_logic_vector is
  variable result: std_logic_vector(a'RANGE);
  alias aa: std_logic_vector(a'REVERSE_RANGE) is a;
begin
  for i in aa'RANGE loop
    result(i) := aa(aa'high - i);
  end loop;
  return result;
end; -- function reverse_any_vector

begin

process (clock, reset)
begin
        if reset = '1' then
                newData <= '0';
                fsl_buf <= (others => '0');
        elsif clock'event and clock = '1' then
                newData <= FSL_S_Exists;
                if FSL_S_Exists = '1' then
                        fsl_buf <= FSL_S_Data;
                else
                        fsl_buf <= (others => '0');
                end if;
        end if;
end process;


fsl_buf_inv <= reverse_any_vector(fsl_buf);
```

```
load_x            <= fsl_buf_inv(LOAD_X_POS);
load_y            <= fsl_buf_inv(LOAD_Y_POS);
load_freq         <= fsl_buf_inv(LOAD_FREQ_POS);
coeff_reset   <= fsl_buf_inv(RESET_COEFF_POS);
coeff_enable <= newData and not (fsl_buf_inv(LOAD_X_POS)
                                 or fsl_buf_inv(LOAD_Y_POS)
                                 or fsl_buf_inv(LOAD_FREQ_POS)
                                 or fsl_buf_inv(RESET_COEFF_POS));

parameterA <= fsl_buf_inv((PARAM_SIZE - 1) downto 0);
parameterB <= fsl_buf_inv((2*PARAM_SIZE - 1) downto PARAM_SIZE);
pixel <= fsl_buf_inv((PIXEL_SIZE - 1) downto 0);

process (clock, reset)
begin
        if reset = '1' then
                -- Reset Signals
                txState <= Idle;
                coeffReg <= (others => '0');
                --
        elsif clock'event and clock = '1' then
                case txState is
                        when Idle =>
                                if coeff_ready = '1' then
                                        coeffReg <= coeff;
                                        txState <= Write_Header;
                                end if;

                        when Write_Header =>
                                if FSL_M_Full = '0' then
                                        txState <= Write_Data;
                                end if;

                        when Write_Data =>
                                if FSL_M_Full = '0' then
                                        txState <= Idle;
                                end if;

                end case;
        end if;
end process;
```

```
FSL_M_Data <= X"00010800" when (txState = Write_Header) else
                    CONV_STD_LOGIC_VECTOR(ID, 8) &
                        reverse_any_vector(coeffReg)
                        when (txState = Write_Data) else
                                (others => '0');

FSL_M_Write <= not FSL_M_Full when (txState = Write_Header OR
                                    txState = Write_Data) else
                                            '0';

FSL_S_Read <= FSL_S_Exists;

end Behavioral;
```

A.2 Coefficient

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity coefficient is
Generic(
        N                       : integer := 3;
        PIXEL_SIZE : integer := 8;
        PARAM_SIZE : integer := 10
);
Port (
        clock   :       in      std_logic;
        enable  :       in std_logic;
        reset   :       in      std_logic;

        pixel                   : in    std_logic_vector(PIXEL_SIZE - 1 downto 0);
        parameterA      : in std_logic_vector(PARAM_SIZE - 1 downto 0);
        parameterB      : in std_logic_vector(PARAM_SIZE - 1 downto 0);

        load_x          : in std_logic;
        load_y          : in std_logic;
        load_freq       : in std_logic;
```

```
        ready    : out std_logic;
        coeff    : out   std_logic_vector(23 downto 0)
);
end coefficient;

architecture Behavioral of coefficient is

component angle
        Generic(
                N                                                       : integer := 3
        );
        Port (
                clock   : in std_logic;
                enable : in std_logic;
                reset   : in std_logic;

                pos       : in   std_logic_vector(9 downto 0);
                freq      : in   std_logic_vector(9 downto 0);
                pixel  : in   std_logic_vector(7 downto 0);

                ready             : out std_logic;
                pixel_buf  : out std_logic_vector(7 downto 0);
                rad               : out std_logic_vector(N+1 downto 0)
        );
end component;

component cosine
        Generic(
                N                                                       : integer := 3
        );
        Port (
                clock            : in std_logic;
                enable    : in std_logic;
                reset    : in std_logic;

                angle_x          : in   std_logic_vector(N+1 downto 0);
                angle_y          : in   std_logic_vector(N+1 downto 0);
                pixel            : in   std_logic_vector(7 downto 0);

                ready             : out std_logic;
                cosine_x   : out std_logic_vector(9 downto 0);
```

```
            cosine_y   : out std_logic_vector(9 downto 0);
            pixel_buf  : out std_logic_vector(7 downto 0)
        );
end component;

component fractionalMultiplier
        Port (
            clock           : in std_logic;
            enable    : in std_logic;
            reset    : in std_logic;
            pixel    : in  std_logic_vector(7 downto 0);
            cosine_x  : in  std_logic_vector(9 downto 0);
            cosine_y  : in  std_logic_vector(9 downto 0);

            ready           : out std_logic;
            negative        : out std_logic;
            result   : out std_logic_vector(15 downto 0)
        );
end component;

component sum
        Port (
            clock                   : in std_logic;
            enable                  : in std_logic;
            reset           : in std_logic;

            plus                    : in  std_logic_vector(15 downto 0);
            operation       : in std_logic;

            ready                   : out std_logic;
            result          : out std_logic_vector(23 downto 0)
        );
end component;

--Internal
signal operation : std_logic;

signal x_in       :        std_logic_vector(9 downto 0) := (others => '0');
signal x_out :   std_logic_vector(9 downto 0) := (others => '0');

signal y_in       :        std_logic_vector(9 downto 0) := (others => '0');
```

```
signal y_out :  std_logic_vector(9 downto 0) := (others => '0');

signal f_w          :       std_logic_vector(9 downto 0) := (others => '0');
signal f_h          :       std_logic_vector(9 downto 0) := (others => '0');

--Count
signal idx_x        : std_logic_vector(9 downto 0) := (others => '0');
signal idx_y        : std_logic_vector(9 downto 0) := (others => '0');
signal count_ended : std_logic;
-- Ready bus
signal angle_enable : std_logic;
signal angle_ready  : std_logic;
signal cosine_ready : std_logic;
signal mult_ready   : std_logic;
signal sum_ready    : std_logic;
signal ready_buf    : std_logic;
--Pixel buffer
signal pixel_angle_buf  : std_logic_vector(7 downto 0);
signal pixel_cosine_buf : std_logic_vector(7 downto 0);
--Outputs
signal angle_x  : std_logic_vector(N+1 downto 0);
signal angle_y  : std_logic_vector(N+1 downto 0);
signal cosine_x : std_logic_vector(9 downto 0);
signal cosine_y : std_logic_vector(9 downto 0);
signal mult              : std_logic_vector(15 downto 0);
signal coeff_buf: std_logic_vector(23 downto 0);

begin

angle_enable <= enable and not count_ended;

angX : angle
generic map(
        N => N
)
port map (
        clock => clock,
        enable => angle_enable,
        reset => reset,

        pos => idx_x,
```

```
        freq => f_w,
        pixel => pixel,

        ready => angle_ready,
        rad => angle_x,
        pixel_buf => pixel_angle_buf
);

angY : angle
generic map(
        N => N
)
port map (
        clock => clock,
        enable => angle_enable,
        reset => reset,

        pos => idx_y,
        freq => f_h,
        pixel => pixel,

        ready => open,
        rad => angle_y,
        pixel_buf => open
);

cos : cosine
generic map(
        N => N
)
port map (
        clock => clock,
        enable => angle_ready,
        reset => reset,

        angle_x => angle_x,
        angle_y => angle_y,
        pixel => pixel_angle_buf,

        ready => cosine_ready,
        cosine_x => cosine_x,
```

```
        cosine_y => cosine_y,
        pixel_buf => pixel_cosine_buf
);

multiplier : fractionalMultiplier
port map (
        clock => clock,
        enable => cosine_ready,
        reset => reset,

        pixel => pixel_cosine_buf,
        cosine_x => cosine_x,
        cosine_y => cosine_y,

        ready => mult_ready,
        negative => operation,
        result => mult
);

sum_res : sum
port map (
        clock  => clock,
        enable => mult_ready,
        reset  => reset,

        plus => mult,
        operation => operation,

        ready => sum_ready,
        result => coeff_buf
);
-- load parameters
process(clock)
begin
        if clock'event and clock = '1' then
                if load_x = '1' then
                        x_in    <= parameterA;
                        x_out   <= parameterB;
                end if;

                if load_y = '1' then
```

```
                y_in      <= parameterA;
                y_out     <= parameterB;
            end if;

            if load_freq = '1' then
                f_w       <= parameterA;
                f_h       <= parameterB;
            end if;
        end if;
end process;

process(reset, clock)
begin
        if reset = '1' then
            idx_x <= x_in;
            idx_y <= y_in;
            count_ended <= '0';
            ready_buf <= '0';
            coeff <= (others => '0');
        elsif clock'event and clock = '1' then

            -- increment index
            if enable = '1' and count_ended = '0' then
                if idx_y = y_out and idx_x = x_out then
                                count_ended <= '1';
                elsif idx_y = y_out then
                        idx_x <= unsigned(idx_x) + 1;
                        idx_y <= y_in;
                else
                        idx_y <= unsigned(idx_y) + 1;
                end if;
            end if;

            -- end condition
            if (count_ended and sum_ready and
                not(angle_ready or cosine_ready or mult_ready)) = '1' then
                ready_buf <= '1';
                coeff <= coeff_buf;
            else
                ready_buf <= '0';
            end if;
```

```
        end if;

end process;

ready <= ready_buf;

end Behavioral;
```

A.3 Angle

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;


entity angle is
        Generic(
                N                                               : integer := 3
        );
        Port (
                clock  : in std_logic;
                enable : in std_logic;
                reset  : in std_logic;

                pos    : in  std_logic_vector(9 downto 0);
                freq   : in  std_logic_vector(9 downto 0);
                pixel  : in  std_logic_vector(7 downto 0);

                ready         : out std_logic;
                pixel_buf : out std_logic_vector(7 downto 0);
                rad           : out std_logic_vector(N + 1 downto 0)
        );
end angle;

architecture Behavioral of angle is

signal pos_aux   : std_logic_vector(9 downto 0);
signal freq_aux  : std_logic_vector(9 downto 0);
signal numerator : std_logic_vector(20 downto 0);
signal pixel_clk_edge : std_logic_vector(7 downto 0);
```

```
begin

process (clock, reset)
begin
        if reset = '1' then
                ready <= '0';
                pos_aux <= (others => '0');
                freq_aux <= (others => '0');
                pixel_clk_edge <= (others => '0');
        elsif clock'event and clock = '1' then
                ready <= enable;
                if enable = '1' then
                        pos_aux <= pos;
                        freq_aux <= freq;
                        pixel_clk_edge <= pixel;
                end if;
        end if;
end process;

numerator <= (pos_aux & '1') * (freq_aux);
rad <= numerator(N + 1 downto 0);
pixel_buf <= pixel_clk_edge;

end Behavioral;
```

A.4 Cosine

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_signed.all;



entity cosine is
        Generic(
                N                                                       : integer := 3
        );
        Port (
                clock           : in std_logic;
                enable    : in std_logic;
```

```
                reset    : in std_logic;

                angle_x          : in  std_logic_vector(N+1 downto 0);
                angle_y          : in  std_logic_vector(N+1 downto 0);
                pixel            : in  std_logic_vector(7 downto 0);

                ready             : out std_logic;
                cosine_x  : out std_logic_vector(9 downto 0);
                cosine_y  : out std_logic_vector(9 downto 0);
                pixel_buf : out std_logic_vector(7 downto 0)
        );
end cosine;

architecture Behavioral of cosine is

COMPONENT LUT
  PORT (
          clka : IN STD_LOGIC;
     rsta : IN STD_LOGIC;
     ena : IN STD_LOGIC;
     addra : IN STD_LOGIC_VECTOR(10 DOWNTO 0);
     douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
     clkb : IN STD_LOGIC;
     rstb : IN STD_LOGIC;
     enb : IN STD_LOGIC;
     addrb : IN STD_LOGIC_VECTOR(10 DOWNTO 0);
     doutb : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END COMPONENT;

signal angle_x_in : std_logic_vector(10 downto 0);
signal angle_y_in : std_logic_vector(10 downto 0);

signal angle_x_aux : std_logic_vector(1 downto 0);
signal angle_y_aux : std_logic_vector(1 downto 0);

signal douta : std_logic_vector(7 downto 0);
signal doutb : std_logic_vector(7 downto 0);

signal pixel_clk_edge : std_logic_vector(7 downto 0);
```

```
begin


angle_x_in(angle_x_in'high downto angle_x_in'high - N) <= angle_x(N downto 0);
angle_x_in((angle_x_in'high - (N + 1)) downto 0) <= (others => '0');

angle_y_in(angle_y_in'high downto angle_y_in'high - N) <= angle_y(N downto 0);
angle_y_in((angle_y_in'high - (N + 1)) downto 0) <= (others => '0');

cosineLUT : LUT
PORT MAP (
        clka  => clock,
        rsta => reset,
   ena => enable,
        addra => angle_x_in,
        douta => douta,
        clkb  => clock,
        rstb => reset,
   enb => enable,
        addrb => angle_y_in,
        doutb => doutb
);

cosine_x(9) <= angle_x_aux(1) xor angle_x_aux(0);
cosine_x(8 downto 0) <= "100000000" when (CONV_INTEGER(douta) = -1) else
                                                                   '0' & douta;

cosine_y(9) <= angle_y_aux(1) xor angle_y_aux(0);
cosine_y(8 downto 0) <= "100000000" when (CONV_INTEGER(doutb) = -1) else
                                                                   '0' & doutb;

process (clock, reset)
begin
        if reset = '1' then
                angle_x_aux <= (others => '0');
                angle_y_aux <= (others => '0');
                pixel_clk_edge <= (others => '0');
                ready <= '0';
        elsif clock'event and clock = '1' then
                ready <= enable;
                if enable = '1' then
```

```
                    pixel_clk_edge <= pixel;
                    angle_x_aux <= angle_x(N+1 downto N);
                    angle_y_aux <= angle_y(N+1 downto N);
                end if;
        end if;
end process;

pixel_buf <= pixel_clk_edge;

end Behavioral;
```

A.5 Fractional Multiplier

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity fractionalMultiplier is
Port (
        clock           : in std_logic;
        enable  : in std_logic;
        reset   : in std_logic;

        pixel   : in  std_logic_vector(7 downto 0);
        cosine_x        : in  std_logic_vector(9 downto 0);
        cosine_y        : in  std_logic_vector(9 downto 0);

        ready           : out std_logic;
        negative : out std_logic;
        result  : out std_logic_vector(15 downto 0)
);
end fractionalMultiplier;

architecture Behavioral of fractionalMultiplier is

signal pixel_aux          : std_logic_vector(7 downto 0);
signal cosine_x_aux : std_logic_vector(9 downto 0);
signal cosine_y_aux : std_logic_vector(9 downto 0);
signal pixel_fraction : std_logic_vector(31 downto 0);
signal cosines_multiplication : std_logic_vector(17 downto 0);
```

```
begin

process(clock, reset)
begin
        if reset = '1' then
                ready <= '0';
                pixel_aux <= (others => '0');
                cosine_x_aux <= (others => '0');
                cosine_y_aux <= (others => '0');
        elsif clock'event and clock = '1' then
                ready <= enable;
                if enable = '1' then
                        pixel_aux <= pixel;
                        cosine_x_aux <= cosine_x;
                        cosine_y_aux <= cosine_y;
                end if;
        end if;

end process;

cosines_multiplication <= cosine_x_aux(8 downto 0) * cosine_y_aux(8 downto 0);
pixel_fraction <= ("0000000" & cosines_multiplication(16 downto 8)) * (pixel_aux & "00000000");

negative <= cosine_x_aux(9) xor cosine_y_aux(9);
result <= pixel_fraction(23 downto 8);



end Behavioral;
```

A.6 Sum

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_signed.all;

entity sum is
        Port (
                clock           : in std_logic;
                enable  : in std_logic;
                reset   : in std_logic;
```

```
		plus	  : in   std_logic_vector(15 downto 0);
		operation : in std_logic;

		ready		  : out std_logic;
		result	  : out std_logic_vector(23 downto 0)
	);
end sum;

architecture Behavioral of sum is

signal result_buffer	: std_logic_vector(23 downto 0) := (others => '0');

begin

process(clock, reset)
begin
	if reset = '1' then
		ready <= '0';
		result_buffer <= (others => '0');
	elsif clock'event and clock = '1' then
		ready <= enable;
		if enable = '1' then
			if operation = '1' then
				result_buffer <= result_buffer - ("00000000" & plus);
			else
				result_buffer <= result_buffer + ("00000000" & plus);
			end if;
		end if;
	end if;
end process;

result <= result_buffer;

end Behavioral;
```

B

NoC

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity noc is
port
    (
        NOC_CLK     : in std_logic;
        NOC_RST     : in std_logic;
        -- DO NOT EDIT BELOW THIS LINE ---------------------
        -- Bus protocol ports, do not add or delete.
        FSLP0_S_Read     : out   std_logic;
        FSLP0_S_Data     : in    std_logic_vector(0 to 31);
        FSLP0_S_Exists   : in    std_logic;
        FSLP0_M_Write    : out   std_logic;
        FSLP0_M_Data     : out   std_logic_vector(0 to 31);
        FSLP0_M_Full     : in    std_logic;

        FSLP1_S_Read     : out   std_logic;
        FSLP1_S_Data     : in    std_logic_vector(0 to 31);
        FSLP1_S_Exists   : in    std_logic;
        FSLP1_M_Write    : out   std_logic;
        FSLP1_M_Data     : out   std_logic_vector(0 to 31);
        FSLP1_M_Full     : in    std_logic;

        FSLP2_S_Read     : out   std_logic;
        FSLP2_S_Data     : in    std_logic_vector(0 to 31);
        FSLP2_S_Exists   : in    std_logic;
        FSLP2_M_Write    : out   std_logic;
        FSLP2_M_Data     : out   std_logic_vector(0 to 31);
```

```
        FSLP2_M_Full      :  in      std_logic;

        FSLP3_S_Read      :  out     std_logic;
        FSLP3_S_Data      :  in      std_logic_vector(0 to 31);
        FSLP3_S_Exists    :  in      std_logic;
        FSLP3_M_Write     :  out     std_logic;
        FSLP3_M_Data      :  out     std_logic_vector(0 to 31);
        FSLP3_M_Full      :  in      std_logic;

        -- Debug interface --
        DBG               : out std_logic_vector(7 downto 0)
    );
end noc;

architecture Behavioral of noc is

COMPONENT fslManager
PORT(
    clock : IN std_logic;
    reset : IN std_logic;
    FSL_S_Read : OUT std_logic;
    FSL_S_Data : IN std_logic_vector(0 to 31);
    FSL_S_Exists : IN std_logic;
    FSL_M_Write : OUT std_logic;
    FSL_M_Data : OUT std_logic_vector(0 to 31);
    requestPortPermission : OUT std_logic_vector(0 to 3);
    replyPortPermission : IN std_logic_vector(0 to 3);
    DBG    : out std_logic_vector(3 downto 0)
);
END COMPONENT;

COMPONENT moderator
PORT(
    clock : IN std_logic;
    reset : IN std_logic;
    FSLP0_M_Write : IN std_logic;
    FSLP0_M_Data : IN std_logic_vector(0 to 31);
    FSLP1_M_Write : IN std_logic;
    FSLP1_M_Data : IN std_logic_vector(0 to 31);
    FSLP2_M_Write : IN std_logic;
    FSLP2_M_Data : IN std_logic_vector(0 to 31);
```

```
    FSLP3_M_Write : IN  std_logic;
    FSLP3_M_Data : IN  std_logic_vector(0 to 31);
    requestPermission : IN  std_logic_vector(0 to 3);
    replyPermission : OUT  std_logic_vector(0 to 3);
    FSLX_M_Write : OUT  std_logic;
    FSLX_M_Data : OUT  std_logic_vector(0 to 31)
);
END COMPONENT;

signal reset : std_logic;
signal clock : std_logic;

signal P0requestPermission : std_logic_vector(0 to 3);
signal P1requestPermission : std_logic_vector(0 to 3);
signal P2requestPermission : std_logic_vector(0 to 3);
signal P3requestPermission : std_logic_vector(0 to 3);

signal P0replyPermission : std_logic_vector(0 to 3);
signal P1replyPermission : std_logic_vector(0 to 3);
signal P2replyPermission : std_logic_vector(0 to 3);
signal P3replyPermission : std_logic_vector(0 to 3);

signal Mod0RequestedPermission : std_logic_vector(0 to 3);
signal Mod1RequestedPermission : std_logic_vector(0 to 3);
signal Mod2RequestedPermission : std_logic_vector(0 to 3);
signal Mod3RequestedPermission : std_logic_vector(0 to 3);

signal Mod0replyPermission : std_logic_vector(0 to 3);
signal Mod1replyPermission : std_logic_vector(0 to 3);
signal Mod2replyPermission : std_logic_vector(0 to 3);
signal Mod3replyPermission : std_logic_vector(0 to 3);

signal FSLP0_2_Mod_M_Write : std_logic;
signal FSLP0_2_Mod_M_Data     : std_logic_vector(0 to 31);

signal FSLP1_2_Mod_M_Write : std_logic;
signal FSLP1_2_Mod_M_Data     : std_logic_vector(0 to 31);

signal FSLP2_2_Mod_M_Write : std_logic;
signal FSLP2_2_Mod_M_Data     : std_logic_vector(0 to 31);
```

```
signal FSLP3_2_Mod_M_Write : std_logic;
signal FSLP3_2_Mod_M_Data     : std_logic_vector(0 to 31);

signal FSLManagerP0_DBG     : std_logic_vector(3 downto 0);

begin

reset <= NOC_RST;
clock <= NOC_CLK;


P0replyPermission <= (Mod0replyPermission(0) and not FSLP0_M_Full)
                     & (Mod1replyPermission(0) and not FSLP1_M_Full)
                     & (Mod2replyPermission(0) and not FSLP2_M_Full)
                     & (Mod3replyPermission(0) and not FSLP3_M_Full);

P1replyPermission <= (Mod0replyPermission(1) and not FSLP0_M_Full)
                         & (Mod1replyPermission(1) and not FSLP1_M_Full)
                         & (Mod2replyPermission(1) and not FSLP2_M_Full)
                         & (Mod3replyPermission(1) and not FSLP3_M_Full);

P2replyPermission <= (Mod0replyPermission(2) and not FSLP0_M_Full)
                         & (Mod1replyPermission(2) and not FSLP1_M_Full)
                         & (Mod2replyPermission(2) and not FSLP2_M_Full)
                         & (Mod3replyPermission(2) and not FSLP3_M_Full);

P3replyPermission <= (Mod0replyPermission(3) and not FSLP0_M_Full)
                         & (Mod1replyPermission(3) and not FSLP1_M_Full)
                         & (Mod2replyPermission(3) and not FSLP2_M_Full)
                         & (Mod3replyPermission(3) and not FSLP3_M_Full);

Mod0RequestedPermission <= P0requestPermission(0) & P1requestPermission(0)
                                    & P2requestPermission(0) & P3requestPermission(0);

Mod1RequestedPermission <= P0requestPermission(1) & P1requestPermission(1)
                                    & P2requestPermission(1) & P3requestPermission(1);

Mod2RequestedPermission <= P0requestPermission(2) & P1requestPermission(2)
                                    & P2requestPermission(2) & P3requestPermission(2);

Mod3RequestedPermission <= P0requestPermission(3) & P1requestPermission(3)
```

```
                                        & P2requestPermission(3) & P3requestPermission(3);

-- Instantiate the fslManagerP0 Unit
fslManagerP0: fslManager PORT MAP (
    clock => clock,
    reset => reset,
    FSL_S_Read      => FSLP0_S_Read,
    FSL_S_Data      => FSLP0_S_Data,
    FSL_S_Exists => FSLP0_S_Exists,
    FSL_M_Write     => FSLP0_2_Mod_M_Write,
    FSL_M_Data      => FSLP0_2_Mod_M_Data,
    requestPortPermission => P0requestPermission,
    replyPortPermission => P0replyPermission,
    DBG => FSLManagerP0_DBG
);

-- Instantiate the moderator0 Unit
moderator0: moderator PORT MAP (
    clock                    => clock,
    reset                    => reset,
    FSLP0_M_Write            => FSLP0_2_Mod_M_Write,
    FSLP0_M_Data            => FSLP0_2_Mod_M_Data,
    FSLP1_M_Write            => FSLP1_2_Mod_M_Write,
    FSLP1_M_Data            => FSLP1_2_Mod_M_Data,
    FSLP2_M_Write            => FSLP2_2_Mod_M_Write,
    FSLP2_M_Data            => FSLP2_2_Mod_M_Data,
    FSLP3_M_Write            => FSLP3_2_Mod_M_Write,
    FSLP3_M_Data            => FSLP3_2_Mod_M_Data,
    requestPermission      => Mod0RequestedPermission,
    replyPermission          => Mod0replyPermission,
    FSLX_M_Write            => FSLP0_M_Write,
    FSLX_M_Data            => FSLP0_M_Data
);

-- Instantiate the fslManagerP1 Unit
fslManagerP1: fslManager PORT MAP (
    clock => clock,
    reset => reset,
    FSL_S_Read      => FSLP1_S_Read,
    FSL_S_Data      => FSLP1_S_Data,
    FSL_S_Exists => FSLP1_S_Exists,
```

```
    FSL_M_Write        => FSLP1_2_Mod_M_Write,
    FSL_M_Data         => FSLP1_2_Mod_M_Data,
    requestPortPermission => P1requestPermission ,
    replyPortPermission => P1replyPermission ,
    DBG => open
);

-- Instantiate the moderator1 Unit
moderator1: moderator PORT MAP (
    clock                   => clock ,
    reset                   => reset ,
    FSLP0_M_Write           => FSLP0_2_Mod_M_Write,
    FSLP0_M_Data           => FSLP0_2_Mod_M_Data,
    FSLP1_M_Write           => FSLP1_2_Mod_M_Write,
    FSLP1_M_Data           => FSLP1_2_Mod_M_Data,
    FSLP2_M_Write           => FSLP2_2_Mod_M_Write,
    FSLP2_M_Data           => FSLP2_2_Mod_M_Data,
    FSLP3_M_Write           => FSLP3_2_Mod_M_Write,
    FSLP3_M_Data           => FSLP3_2_Mod_M_Data,
    requestPermission     => Mod1RequestedPermission ,
    replyPermission        => Mod1replyPermission ,
    FSLX_M_Write           => FSLP1_M_Write,
    FSLX_M_Data           => FSLP1_M_Data
);

-- Instantiate the fslManagerP2 Unit
fslManagerP2: fslManager PORT MAP (
    clock => clock ,
    reset => reset ,
    FSL_S_Read         => FSLP2_S_Read,
    FSL_S_Data         => FSLP2_S_Data,
    FSL_S_Exists => FSLP2_S_Exists,
    FSL_M_Write        => FSLP2_2_Mod_M_Write,
    FSL_M_Data         => FSLP2_2_Mod_M_Data,
    requestPortPermission => P2requestPermission ,
    replyPortPermission => P2replyPermission ,
    DBG => open
);

-- Instantiate the moderator2 Unit
moderator2: moderator PORT MAP (
```

```
    clock                     => clock,
    reset                     => reset,
    FSLP0_M_Write             => FSLP0_2_Mod_M_Write,
    FSLP0_M_Data              => FSLP0_2_Mod_M_Data,
    FSLP1_M_Write             => FSLP1_2_Mod_M_Write,
    FSLP1_M_Data              => FSLP1_2_Mod_M_Data,
    FSLP2_M_Write             => FSLP2_2_Mod_M_Write,
    FSLP2_M_Data              => FSLP2_2_Mod_M_Data,
    FSLP3_M_Write             => FSLP3_2_Mod_M_Write,
    FSLP3_M_Data              => FSLP3_2_Mod_M_Data,
    requestPermission         => Mod2RequestedPermission,
    replyPermission           => Mod2replyPermission,
    FSLX_M_Write              => FSLP2_M_Write,
    FSLX_M_Data               => FSLP2_M_Data
);
-- Instantiate the fslManagerP3 Unit
fslManagerP3: fslManager PORT MAP (
    clock => clock,
    reset => reset,
    FSL_S_Read       => FSLP3_S_Read,
    FSL_S_Data       => FSLP3_S_Data,
    FSL_S_Exists => FSLP3_S_Exists,
    FSL_M_Write      => FSLP3_2_Mod_M_Write,
    FSL_M_Data       => FSLP3_2_Mod_M_Data,
    requestPortPermission => P3requestPermission,
    replyPortPermission => P3replyPermission,
    DBG => open
);

-- Instantiate the moderator3 Unit
moderator3: moderator PORT MAP (
    clock                     => clock,
    reset                     => reset,
    FSLP0_M_Write             => FSLP0_2_Mod_M_Write,
    FSLP0_M_Data              => FSLP0_2_Mod_M_Data,
    FSLP1_M_Write             => FSLP1_2_Mod_M_Write,
    FSLP1_M_Data              => FSLP1_2_Mod_M_Data,
    FSLP2_M_Write             => FSLP2_2_Mod_M_Write,
    FSLP2_M_Data              => FSLP2_2_Mod_M_Data,
    FSLP3_M_Write             => FSLP3_2_Mod_M_Write,
    FSLP3_M_Data              => FSLP3_2_Mod_M_Data,
```

```
    requestPermission       => Mod3RequestedPermission,
    replyPermission          => Mod3replyPermission,
    FSLX_M_Write             => FSLP3_M_Write,
    FSLX_M_Data              => FSLP3_M_Data
);

DBG <= P0replyPermission & FSLManagerP0_DBG;

end Behavioral;
```

B.1 FSL Manager

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity fslManager is
port(
        clock  : in std_logic;
        reset  : in std_logic;

        FSL_S_Read      : out  std_logic;
        FSL_S_Data      : in   std_logic_vector(0 to 31);
        FSL_S_Exists  : in     std_logic;
        FSL_M_Write     : out  std_logic;
        FSL_M_Data      : out  std_logic_vector(0 to 31);

        -- Permissions
        requestPortPermission  : out std_logic_vector(0 to 3);
        replyPortPermission  : in std_logic_vector(0 to 3);

        -- DBG --
        DBG     : out std_logic_vector(3 downto 0)
);
end fslManager;

architecture Behavioral of fslManager is

type STATE_TYPE is (Idle, Wait_Header, Wait_Data, Write_Data, Read_FSL);
signal state        : STATE_TYPE;
```

```
signal msg_size   : std_logic_vector(0 to 15);

signal hasPermission      : std_logic;
signal requestPermission  : std_logic_vector(0 to 3);
signal replyPermission    : std_logic_vector(0 to 3);

begin

process (clock, reset)
begin  -- process The_SW_accelerator

        if reset = '1' then
                -- Reset Signals
                state   <= Idle;
                requestPermission <= (others => '0');
                msg_size <= (others => '0');
                --
        elsif clock'event and clock = '1' then     -- Rising clock edge
                case state is
                        when Idle =>
                                requestPermission <= (others => '0');
                                msg_size <= (others => '0');
                                state   <= Wait_Header;

                        when Wait_Header =>
                                if FSL_S_Exists = '1' then
                                        -- HEADER FORMAT
                                        -- 0_____15_16________19_20________23_24____________27_28_____________31
                                        -- | size | NOT USED | dst_port | noc_y_index | noc_x_index |
                                        -- |________|____________|____________|________________|________________|
                                        msg_size                        <= FSL_S_Data(0 to 15);
                                        requestPermission       <= FSL_S_Data(20 to 23);
                                        state   <= Read_FSL;
                                end if;

                        when Wait_Data =>
                                if FSL_S_Exists = '1' then
                                        state   <= Write_Data;
                                end if;
```

```vhdl
                    when Write_Data =>
                        if hasPermission = '1' then
                            msg_size <= msg_size - 1;

                            state    <= Read_FSL;
                        end if;

                    when Read_FSL =>
                        if msg_size = 0 then
                            state    <= Idle;
                        else
                            state <= Wait_Data;
                        end if;

                end case;
            end if;
end process;

requestPortPermission <= requestPermission;
replyPermission <= replyPortPermission;
hasPermission <= '1' when (replyPermission = requestPermission) else
                                                    '0';

FSL_S_Read <=     FSL_S_Exists when (state = Read_FSL) else
                                                  '0';

FSL_M_Write <= hasPermission when (state = Write_Data) else
                                                  '0';

FSL_M_Data <= FSL_S_Data;

DBG <= "1111" when (state = IDLE) else
                "0110" when (state = WAIT_HEADER) else
                "1001" when (state = WAIT_DATA) else
                "1010" when (state = WRITE_DATA) else
                "0000";

end Behavioral;
```

B.2 Moderator

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity moderator is
	port(
		-- Moderator primary clock
		clock    : in std_logic;
		reset  : in std_logic;

		-- START FSL IN DECLARATION (P0)
		FSLP0_M_Write    :  in      std_logic;
		FSLP0_M_Data     :  in      std_logic_vector(0 to 31);
		-- END FSL OUT DECLARATION

		-- START FSL IN DECLARATION (P1)
		FSLP1_M_Write    :  in      std_logic;
		FSLP1_M_Data     :  in      std_logic_vector(0 to 31);
		-- END FSL OUT DECLARATION

		-- START FSL IN DECLARATION (P2)
		FSLP2_M_Write    :  in      std_logic;
		FSLP2_M_Data     :  in      std_logic_vector(0 to 31);
		-- END FSL OUT DECLARATION

		-- START FSL IN DECLARATION (P3)
		FSLP3_M_Write    :  in      std_logic;
		FSLP3_M_Data     :  in      std_logic_vector(0 to 31);
		-- END FSL OUT DECLARATION

		-- START MODERATOR CONTROL SIGNALS DECLARATION
		requestPermission  : in std_logic_vector(0 to 3);
		replyPermission  : out std_logic_vector(0 to 3);
		-- END MODERATOR CONTROL SIGNALS DECLARATION

		-- START FSL OUT DECLARATION (out)
		FSLX_M_Write     :  out     std_logic;
		FSLX_M_Data              :  out     std_logic_vector(0 to 31)
		-- END FSL OUT DECLARATION
	);
end moderator;
```

```
architecture Behavioral of moderator is

signal Locked : std_logic := '0';

signal has_requests : std_logic;
signal permissions : std_logic_vector(0 to 3);
signal write_signal : std_logic_vector(0 to 3);
signal sweep_index : NATURAL range 0 to 3;

begin

has_requests <= '0' when (requestPermission = "0000") else
                                            '1';

write_signal <= FSLP0_M_Write & FSLP1_M_Write &
                                            FSLP2_M_Write & FSLP3_M_Write;

replyPermission <= permissions;

process (clock, reset)
begin
        if reset = '1' then
                permissions <= (others => '0');
        elsif clock'event and clock = '1' then      -- Rising clock edge
                permissions <= (others => '0');
                permissions(sweep_index) <= requestPermission(sweep_index);

                if Locked = '0' AND has_requests = '1' then
                        if requestPermission(sweep_index) = '1' then
                                Locked <= '1';
                        else
                                sweep_index <= sweep_index + 1;
                        end if;
                else
                        if requestPermission(sweep_index) = '0' then
                                Locked <= '0';
                        end if;
                end if;
        end if;

end process;
```

```
FSLX_M_Write <= write_signal(sweep_index) AND Locked;

FSLX_M_Data <= (others => '0') when Locked = '0' else
                                       FSLP0_M_Data when sweep_index = 0 else
                                       FSLP1_M_Data when sweep_index = 1 else
                                       FSLP2_M_Data when sweep_index = 2 else
                                       FSLP3_M_Data when sweep_index = 3;

end Behavioral;
```

C

Microblaze code

```
#include <stdio.h>
#include "platform.h"
#include "dct/dct.h"
#include "noc/noc.h"
#include "xparameters.h"
#include "xstatus.h"
#include "xtmrctr.h"

int timerOffset;
XTmrCtr TimerCounter; /* The instance of the Tmrctr Device */

void print(char *str);

void simultaneousDiffFreqTest(){
        u32 config[7];
        u32 dctData[3];

        xil_printf("Started test...\n\r\n\r");

        portNumber port = port1;

        config[0] = dctParameters(loadH, 0);
        config[1] = dctParameters(loadH | highLow, 0);
        config[2] = dctParameters(loadW, 0);
        config[3] = dctParameters(loadW | highLow, 0);
        config[4] = dctParameters(loadFreq, 0);
        config[5] = dctParameters(loadFreq | highLow, 0);
        config[6] = dctParameters(resetDct, 0);
        noc_SndPackage(0, 0, port, config, 7);
```

```
        u8 status;
        do {
                ngetfsl(dctData[0], 0);
                fsl_isinvalid(status);
                xil_printf("[dumping 0x%08x]\n\r", dctData[0]);
        } while (!status);


        u8 pixel = 1;
        config[1] = dctPixel(pixel);
        noc_SndPackage(0, 0, port, config, 1);
        dctData[0] = readFSL();
        xil_printf("[coefficient 0x%08x]\n\r", dctData[0]);
        dctData[1] = readFSL();
        xil_printf("[coefficient 0x%08x]\n\r", dctData[1]);
}

int main()
{
    init_platform();
        int status = XTmrCtr_Initialize(&TimerCounter, XPAR_MBTIMER_DEVICE_ID);
        if (status != XST_SUCCESS) {
                return XST_FAILURE;
        }
        XTmrCtr_SetResetValue(&TimerCounter, 0, 0);
        XTmrCtr_SetOptions(&TimerCounter, 0,
                                XTC_AUTO_RELOAD_OPTION);

        XTmrCtr_Reset(&TimerCounter, XPAR_TMRCTR_0_DEVICE_ID);
        timerOffset = XTmrCtr_GetValue(&TimerCounter, XPAR_TMRCTR_0_DEVICE_ID);
        XTmrCtr_Start(&TimerCounter, XPAR_TMRCTR_0_DEVICE_ID);
        timerOffset = XTmrCtr_GetValue(&TimerCounter, XPAR_TMRCTR_0_DEVICE_ID) - timerOffset;
        XTmrCtr_Stop(&TimerCounter, XPAR_TMRCTR_0_DEVICE_ID);
        xil_printf("Timer test: %d\n\r\n\r", timerOffset);

        simultaneousDiffFreqTest();

        xil_printf("Test ended...\n\r");
    cleanup_platform();
```

```
    while(1);

    return 0;
}
```

C.1 NoC

C.1.1 C file

```
/*
 * noc.c
 *
 *  Created on: 8 de Ago de 2012
 *      Author: WMalia
 */

#include "noc.h"

u8 nReadFSL(u32 * data) {
        u8 status;
        ngetfsl(*data, 0);
        fsl_isinvalid(status);

        return status;
}

u32 readFSL() {
        u8 status;
        u32 data;
        do {
                ngetfsl(data, 0);
                fsl_isinvalid(status);
        } while (status);

        return data;
}

void writeFSL(u32 data) {
        u8 status;
        do {
                nputfsl(data, 0);
```

```
                fsl_isinvalid(status);
        } while (status);
}

u32 conv_header_2_u32(u16 size, u8 not_used, portNumber dst_port,
                u8 noc_y_index, u8 noc_x_index) {
        Header header;

        header.size = size;
        header.not_used = not_used;
        header.dst_port = dst_port;
        header.noc_y_index = noc_y_index;
        header.noc_x_index = noc_x_index;

        return *((u32*) &header);
}

Header conv_u32_2_header(u32 header) {
        return *((Header*) &header);
}

int noc_RcvPackage(u32 * data, u32 size) {
        u16 index = 0;

        while (index < size) {
                data[index++] = readFSL();
        }
        return size;
}

void noc_SndPackage(u8 x, u8 y, portNumber port, u32 * data, u16 size) {
        u16 index = 0;
        u32 header = conv_header_2_u32(size, 0xFF, port, 0xFF, 0xFF);
        writeFSL(header);
        while (index < size) {
                writeFSL(data[index++]);
        }
}

void noc_OpenStream(u8 x, u8 y, portNumber port, u16 size){
        u32 header = conv_header_2_u32(size, 0xFF, port, 0xFF, 0xFF);
```

```
        writeFSL(header);
}

void noc_SendStreamPacket(u32 data){
        writeFSL(data);
}
```

C.1.2 H file

```
/*
 * noc.h
 *
 *  Created on: 8 de Ago de 2012
 *      Author: WMalia
 */

#ifndef NOC_H_
#define NOC_H_
/****************************************/
/*                INCLUDES              */
/****************************************/
#include "mb_interface.h"
/****************************************/
/*                 MACROS               */
/****************************************/
typedef enum portNumber{
        portAll = 0x0F,
        port0 = 0x08,
        port1 = 0x04,
        port2 = 0x02,
        port3 = 0x01
}portNumber;

#define START_CODE 0x0A
/****************************************/
/*               STRUCTURES             */
/****************************************/
//                                   HEADER FORMAT
// 0_____15_16________19_20________23_24___________27_28____________31
// | size | NOT USED | dst_port | noc_y_index | noc_x_index |
// |_______|____________|____________|_______________|_______________|
typedef struct msg_header{
```

```
        u8 noc_x_index : 4;
        u8 noc_y_index : 4;
        portNumber dst_port : 4;
        u8 not_used : 4;
        u16 size;
}Header;
/***************************************/
/*                        FUNCTIONS                        */
/***************************************/
u32 conv_header_2_u32(u16 size, u8 not_used, portNumber dst_port,
                      u8 noc_y_index, u8 noc_x_index);
Header conv_u32_2_header(u32 header);
int noc_RcvPackage(u32 * data, u32 size);
void noc_SndPackage(u8 x, u8 y, portNumber port, u32 * data, u16 size);

void noc_OpenStream(u8 x, u8 y, portNumber port, u16 size);
void noc_SendStreamPacket(u32 data);

#endif /* NOC_H_ */
```

C.2 DCT

C.2.1 C file

```
/*
 * dct.c
 *
 *  Created on: 29 de Mar de 2014
 *      Author: Patricia
 */
#include "dct.h"

u32 dctParameters(u32 operation, u16 parameter){
        return operation | (u32)parameter;
}

u32 dctPixel(u8 pixel){
        return (u32)pixel;
 }
```

C.2.2 H file

```c
/*
 * dct.h
 *
 *  Created on: 29 de Mar de 2014
 *      Author: Patricia
 */

#ifndef DCT_H_
#define DCT_H_

#include "../noc/noc.h"

typedef enum operation{

	loadW     = 0x80000000,
	loadH     = 0x40000000,
	loadFreq  = 0x20000000,
	highLow   = 0x10000000,
	resetDct  = 0x08000000,
	newCoeff  = 0x00000000
}operation;

typedef struct{
	u8 empty : 8;
	u32 param2 : 10;
	u32 param1 : 10;
	u8 op : 4;
}DCTConfig;

typedef struct{
	u32 empty : 20;
	u32 pixel : 8;
	u8 op : 4;
}DCTData;

u32 dctParameters(u32 operation, u16 parameter);
u32 dctPixel(u8 pixel);

#endif /* DCT_H_ */
```

D

System description

```
# ##############################################################################
# Created by Base System Builder Wizard for Xilinx EDK 14.2 Build EDK_P.28xd
# Mon Apr 21 21:30:16 2014
# Target Board:  digilent atlys Rev C
# Family:     spartan6
# Device:     xc6slx45
# Package:    csg324
# Speed Grade:  -3
# ##############################################################################
 PARAMETER VERSION = 2.1.0


 PORT zio = zio, DIR = IO
 PORT rzq = rzq, DIR = IO
 PORT mcbx_dram_we_n = mcbx_dram_we_n, DIR = O
 PORT mcbx_dram_udqs_n = mcbx_dram_udqs_n, DIR = IO
 PORT mcbx_dram_udqs = mcbx_dram_udqs, DIR = IO
 PORT mcbx_dram_udm = mcbx_dram_udm, DIR = O
 PORT mcbx_dram_ras_n = mcbx_dram_ras_n, DIR = O
 PORT mcbx_dram_odt = mcbx_dram_odt, DIR = O
 PORT mcbx_dram_ldm = mcbx_dram_ldm, DIR = O
 PORT mcbx_dram_dqs_n = mcbx_dram_dqs_n, DIR = IO
 PORT mcbx_dram_dqs = mcbx_dram_dqs, DIR = IO
 PORT mcbx_dram_dq = mcbx_dram_dq, DIR = IO, VEC = [15:0]
 PORT mcbx_dram_clk_n = mcbx_dram_clk_n, DIR = O, SIGIS = CLK
 PORT mcbx_dram_clk = mcbx_dram_clk, DIR = O, SIGIS = CLK
 PORT mcbx_dram_cke = mcbx_dram_cke, DIR = O
 PORT mcbx_dram_cas_n = mcbx_dram_cas_n, DIR = O
```

```
 PORT mcbx_dram_ba = mcbx_dram_ba, DIR = O, VEC = [2:0]
 PORT mcbx_dram_addr = mcbx_dram_addr, DIR = O, VEC = [12:0]
 PORT RS232_Uart_1_sout = RS232_Uart_1_sout, DIR = O
 PORT RS232_Uart_1_sin = RS232_Uart_1_sin, DIR = I
 PORT RESET = RESET, DIR = I, SIGIS = RST, RST_POLARITY = 0
 PORT GCLK = GCLK, DIR = I, SIGIS = CLK, CLK_FREQ = 100000000
 PORT dct_0_DBG_pin = dct_0_DBG, DIR = O, VEC = [7:0]


BEGIN proc_sys_reset
 PARAMETER INSTANCE = proc_sys_reset_0
 PARAMETER HW_VER = 3.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT MB_Debug_Sys_Rst = proc_sys_reset_0_MB_Debug_Sys_Rst
 PORT Dcm_locked = proc_sys_reset_0_Dcm_locked
 PORT MB_Reset = proc_sys_reset_0_MB_Reset
 PORT Slowest_sync_clk = clk_100_0000MHzPLL0
 PORT Interconnect_aresetn = proc_sys_reset_0_Interconnect_aresetn
 PORT Ext_Reset_In = RESET
 PORT BUS_STRUCT_RESET = proc_sys_reset_0_BUS_STRUCT_RESET
 PORT Peripheral_Reset = net_noc_0_NOC_RST_pin
END

BEGIN lmb_v10
 PARAMETER INSTANCE = microblaze_0_ilmb
 PARAMETER HW_VER = 2.00.b
 PORT SYS_RST = proc_sys_reset_0_BUS_STRUCT_RESET
 PORT LMB_CLK = clk_100_0000MHzPLL0
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = microblaze_0_i_bram_ctrl
 PARAMETER HW_VER = 3.10.a
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007FFF
 BUS_INTERFACE SLMB = microblaze_0_ilmb
 BUS_INTERFACE BRAM_PORT = microblaze_0_i_bram_ctrl_2_microblaze_0_bram_block
END

BEGIN lmb_v10
 PARAMETER INSTANCE = microblaze_0_dlmb
```

```
 PARAMETER HW_VER = 2.00.b
 PORT SYS_RST = proc_sys_reset_0_BUS_STRUCT_RESET
 PORT LMB_CLK = clk_100_0000MHzPLL0
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = microblaze_0_d_bram_ctrl
 PARAMETER HW_VER = 3.10.a
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007FFF
 BUS_INTERFACE SLMB = microblaze_0_dlmb
 BUS_INTERFACE BRAM_PORT = microblaze_0_d_bram_ctrl_2_microblaze_0_bram_block
END

BEGIN bram_block
 PARAMETER INSTANCE = microblaze_0_bram_block
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = microblaze_0_i_bram_ctrl_2_microblaze_0_bram_block
 BUS_INTERFACE PORTB = microblaze_0_d_bram_ctrl_2_microblaze_0_bram_block
END

BEGIN microblaze
 PARAMETER INSTANCE = microblaze_0
 PARAMETER HW_VER = 8.40.a
 PARAMETER C_INTERCONNECT = 2
 PARAMETER C_USE_BARREL = 1
 PARAMETER C_USE_FPU = 0
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_ICACHE_BASEADDR = 0xa8000000
 PARAMETER C_ICACHE_HIGHADDR = 0xafffffff
 PARAMETER C_USE_ICACHE = 1
 PARAMETER C_CACHE_BYTE_SIZE = 8192
 PARAMETER C_ICACHE_ALWAYS_USED = 1
 PARAMETER C_DCACHE_BASEADDR = 0xa8000000
 PARAMETER C_DCACHE_HIGHADDR = 0xafffffff
 PARAMETER C_USE_DCACHE = 1
 PARAMETER C_DCACHE_BYTE_SIZE = 8192
 PARAMETER C_DCACHE_ALWAYS_USED = 1
 PARAMETER C_FSL_LINKS = 1
 PARAMETER C_USE_DIV = 1
 BUS_INTERFACE SFSL0 = fsl_mbSlave
```

```
 BUS_INTERFACE M_AXI_DP = axi4lite_0
 BUS_INTERFACE M_AXI_DC = axi4_0
 BUS_INTERFACE M_AXI_IC = axi4_0
 BUS_INTERFACE DEBUG = microblaze_0_debug
 BUS_INTERFACE DLMB = microblaze_0_dlmb
 BUS_INTERFACE ILMB = microblaze_0_ilmb
 BUS_INTERFACE MFSL0 = fsl_mbMaster
 PORT MB_RESET = proc_sys_reset_0_MB_Reset
 PORT CLK = clk_100_0000MHzPLL0
END

BEGIN mdm
 PARAMETER INSTANCE = debug_module
 PARAMETER HW_VER = 2.10.a
 PARAMETER C_INTERCONNECT = 2
 PARAMETER C_USE_UART = 1
 PARAMETER C_BASEADDR = 0x41400000
 PARAMETER C_HIGHADDR = 0x4140ffff
 BUS_INTERFACE S_AXI = axi4lite_0
 BUS_INTERFACE MBDEBUG_0 = microblaze_0_debug
 PORT Debug_SYS_Rst = proc_sys_reset_0_MB_Debug_Sys_Rst
 PORT S_AXI_ACLK = clk_100_0000MHzPLL0
END

BEGIN clock_generator
 PARAMETER INSTANCE = clock_generator_0
 PARAMETER HW_VER = 4.03.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PARAMETER C_CLKIN_FREQ = 100000000
 PARAMETER C_CLKOUT0_FREQ = 600000000
 PARAMETER C_CLKOUT0_GROUP = PLL0
 PARAMETER C_CLKOUT0_BUF = FALSE
 PARAMETER C_CLKOUT1_FREQ = 600000000
 PARAMETER C_CLKOUT1_PHASE = 180
 PARAMETER C_CLKOUT1_GROUP = PLL0
 PARAMETER C_CLKOUT1_BUF = FALSE
 PARAMETER C_CLKOUT2_FREQ = 100000000
 PARAMETER C_CLKOUT2_GROUP = PLL0
 PARAMETER C_CLKOUT0_DUTY_CYCLE = 0.500000
 PARAMETER C_CLKOUT0_PHASE = 0
 PARAMETER C_CLKOUT1_DUTY_CYCLE = 0.500000
```

```
 PARAMETER C_CLKOUT2_BUF = TRUE
 PARAMETER C_CLKOUT2_DUTY_CYCLE = 0.500000
 PARAMETER C_CLKOUT2_PHASE = 0
 PORT LOCKED = proc_sys_reset_0_Dcm_locked
 PORT CLKOUT2 = clk_100_0000MHzPLL0
 PORT RST = RESET
 PORT CLKOUT0 = clk_600_0000MHzPLL0_nobuf
 PORT CLKOUT1 = clk_600_0000MHz180PLL0_nobuf
 PORT CLKIN = GCLK
END

BEGIN axi_interconnect
 PARAMETER INSTANCE = axi4lite_0
 PARAMETER HW_VER = 1.06.a
 PARAMETER C_INTERCONNECT_CONNECTIVITY_MODE = 0
 PORT INTERCONNECT_ARESETN = proc_sys_reset_0_Interconnect_aresetn
 PORT INTERCONNECT_ACLK = clk_100_0000MHzPLL0
END

BEGIN axi_interconnect
 PARAMETER INSTANCE = axi4_0
 PARAMETER HW_VER = 1.06.a
 PORT interconnect_aclk = clk_100_0000MHzPLL0
 PORT INTERCONNECT_ARESETN = proc_sys_reset_0_Interconnect_aresetn
END

BEGIN axi_uartlite
 PARAMETER INSTANCE = RS232_Uart_1
 PARAMETER HW_VER = 1.02.a
 PARAMETER C_BAUDRATE = 115200
 PARAMETER C_DATA_BITS = 8
 PARAMETER C_USE_PARITY = 0
 PARAMETER C_ODD_PARITY = 1
 PARAMETER C_BASEADDR = 0x40600000
 PARAMETER C_HIGHADDR = 0x4060ffff
 BUS_INTERFACE S_AXI = axi4lite_0
 PORT S_AXI_ACLK = clk_100_0000MHzPLL0
 PORT TX = RS232_Uart_1_sout
 PORT RX = RS232_Uart_1_sin
END
```

```
BEGIN axi_s6_ddrx
 PARAMETER INSTANCE = MCB_DDR2
 PARAMETER HW_VER = 1.06.a
 PARAMETER C_MCB_RZQ_LOC = L6
 PARAMETER C_MCB_ZIO_LOC = C2
 PARAMETER C_MEM_TYPE = DDR2
 PARAMETER C_MEM_PARTNO = EDE1116AXXX-8E
 PARAMETER C_MEM_BANKADDR_WIDTH = 3
 PARAMETER C_MEM_NUM_COL_BITS = 10
 PARAMETER C_SKIP_IN_TERM_CAL = 0
 PARAMETER C_S0_AXI_ENABLE = 1
 PARAMETER C_INTERCONNECT_S0_AXI_MASTERS = microblaze_0.M_AXI_DC & microblaze_0.M_AXI_IC
 PARAMETER C_MEM_DDR2_RTT = 50OHMS
 PARAMETER C_S0_AXI_STRICT_COHERENCY = 0
 PARAMETER C_INTERCONNECT_S0_AXI_AW_REGISTER = 8
 PARAMETER C_INTERCONNECT_S0_AXI_AR_REGISTER = 8
 PARAMETER C_INTERCONNECT_S0_AXI_W_REGISTER = 8
 PARAMETER C_INTERCONNECT_S0_AXI_R_REGISTER = 8
 PARAMETER C_INTERCONNECT_S0_AXI_B_REGISTER = 8
 PARAMETER C_S0_AXI_BASEADDR = 0xa8000000
 PARAMETER C_S0_AXI_HIGHADDR = 0xafffffff
 BUS_INTERFACE S0_AXI = axi4_0
 PORT zio = zio
 PORT rzq = rzq
 PORT s0_axi_aclk = clk_100_0000MHzPLL0
 PORT ui_clk = clk_100_0000MHzPLL0
 PORT mcbx_dram_we_n = mcbx_dram_we_n
 PORT mcbx_dram_udqs_n = mcbx_dram_udqs_n
 PORT mcbx_dram_udqs = mcbx_dram_udqs
 PORT mcbx_dram_udm = mcbx_dram_udm
 PORT mcbx_dram_ras_n = mcbx_dram_ras_n
 PORT mcbx_dram_odt = mcbx_dram_odt
 PORT mcbx_dram_ldm = mcbx_dram_ldm
 PORT mcbx_dram_dqs_n = mcbx_dram_dqs_n
 PORT mcbx_dram_dqs = mcbx_dram_dqs
 PORT mcbx_dram_dq = mcbx_dram_dq
 PORT mcbx_dram_clk_n = mcbx_dram_clk_n
 PORT mcbx_dram_clk = mcbx_dram_clk
 PORT mcbx_dram_cke = mcbx_dram_cke
 PORT mcbx_dram_cas_n = mcbx_dram_cas_n
 PORT mcbx_dram_ba = mcbx_dram_ba
```

```
 PORT mcbx_dram_addr = mcbx_dram_addr
 PORT sysclk_2x = clk_600_0000MHzPLL0_nobuf
 PORT sysclk_2x_180 = clk_600_0000MHz180PLL0_nobuf
 PORT SYS_RST = proc_sys_reset_0_BUS_STRUCT_RESET
 PORT PLL_LOCK = proc_sys_reset_0_Dcm_locked
END

BEGIN fsl_v20
 PARAMETER INSTANCE = fsl_mbSlave
 PARAMETER HW_VER = 2.11.e
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = clk_100_0000MHzPLL0
 PORT SYS_Rst = net_noc_0_NOC_RST_pin
 PORT FSL_M_Clk = clk_100_0000MHzPLL0
 PORT FSL_S_Clk = clk_100_0000MHzPLL0
END

BEGIN fsl_v20
 PARAMETER INSTANCE = fsl_mbMaster
 PARAMETER HW_VER = 2.11.e
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = clk_100_0000MHzPLL0
 PORT SYS_Rst = net_noc_0_NOC_RST_pin
 PORT FSL_M_Clk = clk_100_0000MHzPLL0
 PORT FSL_S_Clk = clk_100_0000MHzPLL0
END

BEGIN fsl_v20
 PARAMETER INSTANCE = fsl_dct0Master
 PARAMETER HW_VER = 2.11.e
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = clk_100_0000MHzPLL0
 PORT SYS_Rst = net_noc_0_NOC_RST_pin
 PORT FSL_M_Clk = clk_100_0000MHzPLL0
 PORT FSL_S_Clk = clk_100_0000MHzPLL0
END

BEGIN fsl_v20
 PARAMETER INSTANCE = fsl_dct0Slave
 PARAMETER HW_VER = 2.11.e
 PARAMETER C_USE_CONTROL = 0
```

```
 PORT FSL_Clk = clk_100_0000MHzPLL0
 PORT SYS_Rst = net_noc_0_NOC_RST_pin
 PORT FSL_M_Clk = clk_100_0000MHzPLL0
 PORT FSL_S_Clk = clk_100_0000MHzPLL0
END

BEGIN fsl_v20
 PARAMETER INSTANCE = fsl_dct1Master
 PARAMETER HW_VER = 2.11.e
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = clk_100_0000MHzPLL0
 PORT SYS_Rst = net_noc_0_NOC_RST_pin
 PORT FSL_M_Clk = clk_100_0000MHzPLL0
 PORT FSL_S_Clk = clk_100_0000MHzPLL0
END

BEGIN fsl_v20
 PARAMETER INSTANCE = fsl_dct1Slave
 PARAMETER HW_VER = 2.11.e
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = clk_100_0000MHzPLL0
 PORT SYS_Rst = net_noc_0_NOC_RST_pin
 PORT FSL_M_Clk = clk_100_0000MHzPLL0
 PORT FSL_S_Clk = clk_100_0000MHzPLL0
END

BEGIN fsl_v20
 PARAMETER INSTANCE = fsl_dct2Master
 PARAMETER HW_VER = 2.11.e
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = clk_100_0000MHzPLL0
 PORT SYS_Rst = net_noc_0_NOC_RST_pin
 PORT FSL_M_Clk = clk_100_0000MHzPLL0
 PORT FSL_S_Clk = clk_100_0000MHzPLL0
END

BEGIN fsl_v20
 PARAMETER INSTANCE = fsl_dct2Slave
 PARAMETER HW_VER = 2.11.e
 PARAMETER C_USE_CONTROL = 0
 PORT FSL_Clk = clk_100_0000MHzPLL0
```

```
 PORT SYS_Rst = net_noc_0_NOC_RST_pin
 PORT FSL_M_Clk = clk_100_0000MHzPLL0
 PORT FSL_S_Clk = clk_100_0000MHzPLL0
END

BEGIN noc
 PARAMETER INSTANCE = noc_0
 PARAMETER HW_VER = 1.00.b
 BUS_INTERFACE SFSLP0 = fsl_mbMaster
 BUS_INTERFACE MFSLP0 = fsl_mbSlave
 BUS_INTERFACE SFSLP1 = fsl_dct0Master
 BUS_INTERFACE MFSLP1 = fsl_dct0Slave
 BUS_INTERFACE SFSLP2 = fsl_dct1Master
 BUS_INTERFACE MFSLP2 = fsl_dct1Slave
 BUS_INTERFACE SFSLP3 = fsl_dct2Master
 BUS_INTERFACE MFSLP3 = fsl_dct2Slave
 PORT NOC_CLK = clk_100_0000MHzPLL0
 PORT NOC_RST = net_noc_0_NOC_RST_pin
 PORT DBG = noc_0_DBG
END

BEGIN dct
 PARAMETER INSTANCE = dct_0
 PARAMETER HW_VER = 4.01.a
 PARAMETER ID = 0
 PARAMETER N = 3
 BUS_INTERFACE SFSL = fsl_dct0Slave
 BUS_INTERFACE MFSL = fsl_dct0Master
 PORT FSL_Clk = clk_100_0000MHzPLL0
 PORT FSL_Rst = net_noc_0_NOC_RST_pin
END

BEGIN dct
 PARAMETER INSTANCE = dct_1
 PARAMETER HW_VER = 4.01.a
 PARAMETER ID = 1
 PARAMETER N = 3
 BUS_INTERFACE SFSL = fsl_dct1Slave
 BUS_INTERFACE MFSL = fsl_dct1Master
 PORT FSL_Clk = clk_100_0000MHzPLL0
 PORT FSL_Rst = net_noc_0_NOC_RST_pin
```

```
END

BEGIN dct
 PARAMETER INSTANCE = dct_2
 PARAMETER HW_VER = 4.01.a
 PARAMETER ID = 2
 PARAMETER N = 3
 BUS_INTERFACE SFSL = fsl_dct2Slave
 BUS_INTERFACE MFSL = fsl_dct2Master
 PORT FSL_Clk = clk_100_0000MHzPLL0
 PORT FSL_Rst = net_noc_0_NOC_RST_pin
END

BEGIN axi_timer
 PARAMETER INSTANCE = mbTimer
 PARAMETER HW_VER = 1.03.a
 PARAMETER C_ONE_TIMER_ONLY = 1
 PARAMETER C_BASEADDR = 0x41c00000
 PARAMETER C_HIGHADDR = 0x41c0ffff
 BUS_INTERFACE S_AXI = axi4lite_0
 PORT S_AXI_ACLK = clk_100_0000MHzPLL0
END
```

References

1. J. Reinders, “Ask james reinders: Multicore vs. manycore,” *[Online]. Available: http://goparallel.sourceforge.net/ask-james-reinders-multicore-vs-manycore/.*

2. R. Marculescu, J. Hu, and U. Ogras, “Key research problems in noc design: a holistic perspective,” in *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS ’05. Third IEEE/ACM/IFIP International Conference on*, pp. 69–74, Sept 2005.

3. S. Pasricha and N. Dutt, *On-chip communication architectures: system on chip interconnect.* Morgan Kaufmann, 2010.

4. A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, “A pipelined fast 2d-dct accelerator for fpga-based socs,” in *VLSI, 2007. ISVLSI’07. IEEE Computer Society Annual Symposium on*, pp. 331–336, IEEE, 2007.

5. D. Trainor, J. Heron, and R. Woods, “Implementation of the 2d dct using a xilinx xc6264 fpga,” in *Signal Processing Systems, 1997. SIPS 97-Design and Implementation., 1997 IEEE Workshop on*, pp. 541–550, IEEE, 1997.

6. V. Gunes and T. Givargis, “Xgrid: A scalable many-core embedded processor,” *Center for Embedded Computer Systems*, vol. 1, no. 2, p. 1, 2013.

7. P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of fpga-based soft processors," in *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pp. 202–212, ACM, 2005.

8. D. Bafumba-Lokilo, Y. Savaria, and J.-P. David, "Generic crossbar network on chip for fpga mpsocs," in *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, pp. 269–272, IEEE, 2008.

9. S. A. Khayam, "The discrete cosine transform: Theory and application," 2003.