TECNOLÓGICO DE MONTERREY®

# VLSI ARCHITECTURES FOR A VIDEO CO-PROCESSOR TOWARD MOBILE APPLICATIONS BASED ON RECONFIGURABLE PLATFORMS.

TESIS QUE PARA OPTAR POR EL GRADO DE
DOCTOR EN CIENCIAS DE INGENIERÍA
PRESENTA

**FRANCISCO JAVIER ORTIZ CERECEDO**

Asesor: Dr. ANDRÉS DAVID GARCÍA GARCÍA

| | | |
|---|---|---|
| Jurado: | Dr. RENÉ ARMANDO CUMPLIDO PARRA, | Presidente |
| | Dr. MIGUEL GONZÁLEZ MENDOZA, | Secretario |
| | Dra. CLAUDIA FEREGRINO URIBE, | Vocal |
| | Dr. JAIME MORA VARGAS, | Vocal |
| | Dr. ANDRÉS DAVID GARCÍA GARCÍA | Vocal |

Atizapán de Zaragoza, Edo. Méx., Noviembre de 2011.

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY
CAMPUS ESTADO DE MÉXICO

TECNOLÓGICO
DE MONTERREY®

# VLSI Architectures for a Video Co-Processor toward mobile applications based on reconfigurable platforms

DOCTORAL DISSERTATION
Presented by


## FRANCISCO JAVIER ORTIZ CERECEDO


To obtain the degree of

## DOCTOR IN ENGINEERING SCIENCES

| | | |
|---|---|---|
| Advisor: | DR. ANDRÉS DAVID GARCÍA GARCÍA | |
| Thesis Commitee: | DR. RENÉ ARMANDO CUMPLIDO PARRA | President |
| | DR. MIGUEL GONZÁLEZ MENDOZA | Secretary |
| | DRA. CLAUDIA FEREGRINO URIBE | Examiner |
| | DR. JAIME MORA VARGAS | Examiner |

Atizapán de Zaragoza, Estado de México
November, 2011

# Dedicatoria

A mis padres, Francisco y Reyna por su apoyo incondicional en todos los proyectos que he emprendido a lo largo de mi vida sin importar lo descabellados y complicados que éstos hayan parecido, sin su apoyo y sin su ejemplo no habría podido lograr nada, MUCHAS GRACIAS!

A mis hermanos Fernando e Hiram, gracias por estar conmigo en las buenas y en las malas, sin ustedes la vida sería muy aburrida.

# Agradecimientos

Quiero agradecer a mi director de tesis, Dr. Andrés García por su ayuda, sus consejos y sus valiosos comentarios que me ayudaron a llevar a cabo este trabajo de tesis.

Agradezco profundamente a los doctores René Cumplido, Claudia Feregrino, Miguel González y Jaime Mora por haberme hecho el honor de participar en el comité de evaluación de este trabajo.

Agradezco a todos mis compañeros del doctorado por el tiempo que hemos convivido y sobre todo por su amistad. Gracias Carmen, Diana, Aarón, Azzael, Danny, Jaime y Pedro.

Gracias al Tecnológico de Monterrey, Campus Estado de México por la beca que me fue otorgada para la realización de mis estudios de grado y por todas las facilidades prestadas durante mi estancia como estudiante.

Por último, quiero hacer un agradecimiento especial al Consejo Nacional de Ciencia y Tecnología (CONACYT) por la beca de manutención que me fue otorgada durante mis estudios doctorales.

# Resumen

El procesamiento de video es un campo que ha tenido un crecimiento notable en los últimos años, ha pasado de ser un área de investigación académica pura a ser un área de desarrollo para aplicaciones de seguridad y electrónica de consumo; actualmente hay una tendencia de mercado orientada a eliminar los reproductores ópticos para reemplazarlos con discos duros y así incrementar la capacidad de almacenamiento al mismo tiempo que se reducen los costos de operación. Un nicho de mercado emergente es el procesamiento de imágenes para aplicaciones médicas, en donde se requieren dispositivos especializados para auxiliar en el diagnóstico de enfermedades incluso si el médico se encuentra en otro lugar.

Un sistema de codificación de video requiere de varias etapas de procesamiento para adquirir, comprimir y codificar la información para después transmitirla a algún otro dispositivo o medio para su posterior reproducción. En términos computacionales, cada etapa es muy demandante, por lo que debemos escoger entre dos enfoques: hardware y software; las implementaciones en software son muy eficientes en términos del volumen de procesamiento de datos, pero el problema radica en que se desperdician recursos del sistema al estar ligados a una longitud de palabra fija dependiente de la arquitectura del procesador, adicionalmente, las operaciones de acceso y escritura a memoria tienden a disminuir el desempeño del sistema.

Cuando el volumen de procesamiento, el consumo de potencia y el desempeño son restricciones de diseño, los procesadores en hardware son la mejor opción para implementar tareas de codificación de video; existen varias arquitecturas de hardware apropiadas para la implementación de procesamiento de video como los Procesadores Digitales de Señales (DSP), Unidades de Procesamiento Gráfico (GPU), Arreglos de Compuertas Programables (FPGA) y Circuitos Integrados de Aplicación Específica. Cada uno de estos procesadores está especializado en tareas específicas, pero solo los FPGA han demostrado tener la capacidad de acelerar cualquier algoritmo o proceso debido a su alto grado de flexibilidad, permitiendo que el diseñador pruebe la arquitectura cuantas veces sean necesarias hasta lograr un diseño eficiente en términos de complejidad material, consumo de potencia y frecuencia de operación.

En la primera parte de esta tesis discutimos las características fundamentales de la representación numérica de la información visual y las bases de la compresión; los métodos de compresión con pérdida y sin pérdida son analizados y comparados; posteriormente se presenta un breve resumen de los estándares de codificación y al final de la sección se

presenta un comparativo entre los procesadores anteriormente mencionados.

En la siguiente etapa de este trabajo se presenta un resumen de las arquitecturas de hardware utilizadas comúnmente para procesamiento de video; Las arquitecturas para calcular la Transformada de Coseno Discreta tienen la característica de ser regulares y apropiadas para la implementacin en alta escala de integración (VLSI); la mayoría de estas arquitecturas está basada en la descomposición Renglón-Columna que permite una implementación directa en hardware; los algoritmos rápidos se explican a profundidad, posteriormente se presenta un estudio de las arquitecturas basadas en Aritmética Distribuida y se establece un comparativo entre las dos familias en términos de la viabilidad para su implementación en FPGA.

La codificación entrópica y la codificación de Huffman se explican en esta sección; existen varios esquemas para la codificación de Huffman que son apropiados para la implementación en ASIC, el problema principal radica en que son arquitecturas muy grandes que requieren de accesos constantes a memoria; posteriormente se explican los Algoritmos de Ordenamiento serial y paralelo así como su implementación en hardware. Las arquitecturas paralelas permiten ordenar arreglos de datos en unos cuantos ciclos de reloj, por lo que se explora la posibilidad de estimar la función de densidad de probabilidad de la imagen con una red de ordenamiento. Al final de esta sección se presentan las arquitecturas para la Estimación de Movimiento.

En la siguiente etapa de este trabajo se presentan las implementaciones de los procesos requeridos para la codificación de video, se explica brevemente en qué consiste la adecuación algoritmo-arquitectura y se presentan los códigos de descripción material de los bloques aritméticos necesarios. Decidimos utilizar plataformas reconfigurables por sus características de flexibilidad y la posibilidad de probar diferentes configuraciones sin necesidad de cambiar completamente el diseño. En esta tesis se utiliza únicamente la librería ieee standard logic para reducir la utilización de recursos.

Finalmente, los resultados de las implementaciones arquitecturales son discutidos y comparados en términos de complejidad material, ya que si ésta es reducida, podemos suponer que habrá un consumo de potencia reducido. Se presentan algunos resultados de síntesis lógica y simulación para determinar la viabilidad de trasladar el sistema a un Circuito Integrado de Aplicación Específica.

# Abstract

Video processing has been a fast growing field in the recent years, it has evolved from a purely academic research area to a research and development area for consumer electronics, medical and security applications; actually there is a market trend oriented to remove all optical media reproducers and replace them with hard drives to increase media storage and to reduce operation costs; an upcoming niche market is identified as medical image processing, where dedicated devices are required to help in the diagnosis of medical conditions even if the physician is in other geographic location.

Conventionally a video coding system involves several processing stages to acquire, compress and code data to convey it efficiently to another media for a later reproduction. Each stage is very demanding in terms of computational complexity and oftenly we have to choose between a hardware or a software solution; software implementations of video coders are very efficients in terms of throughput, but they tend to underutilize the system resources as they work with fixed wordlengths and are tied to an specific processor architecture with external memory devices; read/write operations required to fetch data tend to diminish the system performance.

When system performance, power consumption and throughput are hard constraints a hardware processor is the best choice to implement video coding tasks; there are many hardware architectures suitable for implementing video processing like Digital Signal Processors (DSP), Graphic Processing Units (GPU), Field Programmable Gate Arrays (FPGA) and Application Specific Integrated Circuits (ASIC). Each processor is specialized for certain task, but only FPGAs have proven to be capable of accelerating any algorithm or process as they offer a high degree of flexibility, allowing the system designer to test the architecture over and over again until an efficient design in terms of material complexity, power consumption and maximum operating frequency is achieved.

In the first part of this thesis we discuss the fundamental characteristics of the numerical representation of visual information and the necessity of compresion; both lossy and lossless compression methods are analyzed and compared; then a brief survey on video coding standards is presented, and a comparison between dedicated processors for image processing and their application is carried out.

In the next stage of this work, we survey hardware architectures for video processing tasks; Discrete Cosine Transform (DCT) architectures have the characteristic of being highly regular and suitable for Very Large Scale of Integration (VLSI) implementation, most of them are based in a Row-Column decomposition that allows to implement the algorithm directly in hardware; Fast algorithms are thoroughly explained, later, Distributed arithmetic architectures are studied and compared to fast algorithms in terms of the feasibility for FPGA implementation.

Entropy coding is explained along with Huffman Codes, some variable length coding architectures are surveyed and explained in terms of their material complexity, there are many Huffman Coding Schemes that are suitable for ASIC implementation, the main drawback is that all of them are too large and require external memories to store the code words; next we explain Sorting algorithms; this architectures are useful to sort ascendingly or descendingly arrays of numbers in a few clock cycles, there are two families of sorters, parallel and serial, in the former a random array is sorted in a comparison-exchange network, meanwhile in the latter the incoming value is inserted into the corresponding position of the array, in any case, we study the possibility of estimating the probability density function of the image with a sorting network, at the end of this section Motion Estimation architectures are presented and the representative block searching algorithms and hierarchical search algorithms are explained in detail.

In the next part of this work the architectural implementations of the above mentioned processes is presented, algorithm-architecture adequations are discussed and the HDL coding process of the required arithmetic modules is explained; we decided to work over reconfigurable platforms because they are the best developing tools because of their characteristics of flexibility and testability; FPGAs were the chosen platform to target HDL designs. Only ieee standard logic library was employed to code the architectures, eventhough there are IP cores and libraries that have many functions or blocks already build we decided to implement the algorithms directly into hardware to reduce resource utilization.

Finally, the results of the architectural implementations are discussed and compared in terms of material complexity, if the complexity is low, then we can expect a low power consumption, nevertheless sinthesis and simulation results are presented in order to determine the feasibility to translate the system design to an ASIC.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Image and Video Processing is a field that has changed exponentially through recent years; it is one of the most successful and used technologies and has evolved from an academic research area to a commercial application research and development area; nowadays is common to hear that certain device has real-time video processing capabilities, or that supports many image and video standards; why do we talk about real time processing? Is it really necessary? Why do we need to process such large amounts of data in short periods of time? Why are we so concerned about giving real time processing capabilities to consumer electronics?

Real-time image and video processing is one of the fastest growing technologies in the communications field as High Definition television, streaming and videoconferencing applications reach everyday larger markets, but also is of great importance in security surveillance systems for large facilities like airports, bus and train stations, schools and even state buildings, video processing is a critical tool for medical applications where an accurate visual representation of images is required to help physicians diagnose a number of diseases or medical conditions, video processing could also help monitoring patients with infectious diseases without endangering medical personnel.

To process an image or a sequence of images we must perform various processes consecutively to compress, quantize, code, and convey information to an storage device or to another processor that reverses the process; compression is understood as the process of eliminating information redundancies within a frame, as we know, there are some light components that the eye cannot see, therefore their presence within the image is irrelevant; if the information of the frame is highly correlated we can assume that there is more information than required, the same principle applies to consecutive frames where a pixel has a high probability of staying as it is in the next frame; The standardized image compression is achieved by applying a Fourier based orthogonal transform known as Discrete Cosine Transform, this transform has the characteristic of concentrating the energy in the lowest frequencies of the cosine function, letting the frame with a huge amount of irrelevant information that can be removed before sending data to the next process in the chain.

The quantization process consists of assigning binary values within a range to the DCT coefficients to that the next process, known as entropy coding can be performed easily and smoothly; unlike DCT, entropy coding is a lossless coding scheme that helps to reduce the amount of information that has to be stored or conveyed, the greatest challenge in entropy coding is to make efficient a computationally exhaustive method known as Huffman Coding. Finally, the most demanding task in terms of computational operations must be performed, Motion Estimation is a method used to identify the portions of the frame that change through adjacent frames, this process is required to increase the maximum DCT compression and to increase the maximum operating frequency.

Images and Video can be processed either in hardware or software, software video processing algorithms are very fast and accurate, but they rely on a computer processor, and an operating system to handle memory access and peripheral devices; hardware video processing can be performed by a number of specialized devices like DSPs, GPUs, FPGAs and ASICs; each signal processor is suitable for some task, for example DSPs are a good option if we need to compute filter functions of convolutional operations between vectors, regularly they are programmed in C language; GPUs are generic devices optimized for video processing tasks, they are very close to CPUs as they only have cache memory and require external memories with random accesses to store the code that is going to be executed, they are popular because they can be programmed with high level languages and their processing rate is usually high.

FPGAs represent along with ASICs the most efficient hardware option to process video; FPGAs have the advantage of being extremely flexible; as a designer, one can test an architecture over and over again until a satisfactory result is achieved, their configuration is usually coded in Hardware Description Languages like Very High Speed Integrated Circuits Hardware Description Language (VHDL) or Verilog HDL allowing the designer to code a complex system in a modular fashion, an FPGA is capable of performing any task of an ASIC, the main difference between them is that FPGAs need to be configured before the process starts and ASICs are already configured to perform an specific task and cannot be used for any other application.

This thesis is concerned with the design of a pure VLSI architecture of a video co-processor that does not rely on a CPU or external memory devices to compress video frames; most of video processing tasks are computationally exhaustive, therefore it is necessary to translate the algorithms into hardware to reduce the computational complexity employing parallel and pipelining techniques by accelerating the processes and reducing the critical path. VLSI design over reconfigurable platforms will allow us to compare hardware implementations targeted for power consumption, die area and operating frequency to establish a triple commitment between design constraints to find the optimum architecture.

The methodology of this thesis consisted in analyzing the algorithms of every video processing stage in the main layer, then a preliminary translation of algorithm to hardware is performed in order to begin testing the design constraints; after analyzing the algorithm's

computational complexity, a hardware version is coded in VHDL and then is synthesized with Altera Quartus II software to determine the material complexity, silicon area and maximum operating frequency of the module.

## Thesis Outline

This thesis is organized as follows. In Chapter 2 the basic concepts of image processing are explained and taken as the basis for compression; Lossy and Lossless compression methods are explained in detail and some recomendations for chosing a compression scheme were presented; A video coding standard survey is presented to understand the different processes that might be follower in order to process video in an efficient way, also a brief summary of video processing devices is presented, later in this chapter a comparative table between DSPs, CPUs, GPUs, FPGAs and ASICs is presented.

Chapter 3 presents a study of the available VLSI architectures for DCT, Quantization, Entropy Coding, Sorting Algorithms and Motion Estimation; the performance of each architecture is thoroughly discussed and a feasibility analysis between existing architectures for a determined processing stage is presented; at the end of the chapter a novel video coder is presented and compared to a generic coder in terms of the number of clock cycles required to process video through all the stages.

In chapter 4 is reported every architecture implemented in the FPGA in terms of how many logic elements are required, how many flip-flops and what amount of memory is required for the proposed architectures, we look forward establishing an approximate measure of the material complexity of the Co-Processor; Chapter 5 presents the results of the architecture synthesis for DCT and sorting networks, a comparison is made between synthesis results optimized for area and speed and the feasibility of implementation in a single reconfigurable device is studied.

Finally, concluding remarks are given and future work is highlited.

# Chapter 2

# State of the Art

In recent years there has been a significant evolution in algorithms and architectures for audio, image and video processing [1]. On the algorithm field robust methods for size reduction of audio, image and video data have been developed in order to make easier the data manipulation, storage and transmission; on the architecture field nowadays it is possible to implement complex compression processes on a relatively low cost integrated circuit, in fact this has incited a great deal of activity in developing multimedia systems for the consumer market [2].

The importance of these advancements is that audio, image and video information have the potential to become another data type, this implies that multimedia data can be digitally encoded so that it can be stored and transmitted in the same media or channel with other digital data types [3]. Data encoding standardization can lead the industry to the development of low cost implementation that will promote the generalized use of multimedia information [4].

## 2.1 Image Processing

Sight is one of the senses that allow us to perceive and assimilate an incredible amount of information in a short time interval [5], the variety of information that goes through the eye to be interpreted by the brain can be consider infinite as we never stop receiving visual information (unless of course we close our eyes). Throughout the years we have increased sight capability by creating devices capable to detect electromagnetic radiation even if the wavelengths are outside the normal vision range. By the use of sound waves or X-rays we are able to "see" inside objects and into locations far beyond our scope, high speed videocameras allow us to see the slightest details of a moving object like a flying humming bird.

Visual information is a term that cannot be accurately defined as it involves everything, there is no way to explain the term unless we introduce some restrictions; first of all, we

shall assume that information is enclosed in a finite image size, that is, the viewer receives the visual information as if looking through a rectangular window of finite dimensions as shown in figure 2.1[1]; every image or video capturing device works under the same principle of handling finite amounts of information. Second restriction is based on the assumption that the viewer does not have the capability of depth perception, so he cannot tell how distant objects are by changing the focus of his eyes, in exchange, he can infer whether the object is in the foreground or in the background based on the position of the object with respect to others.

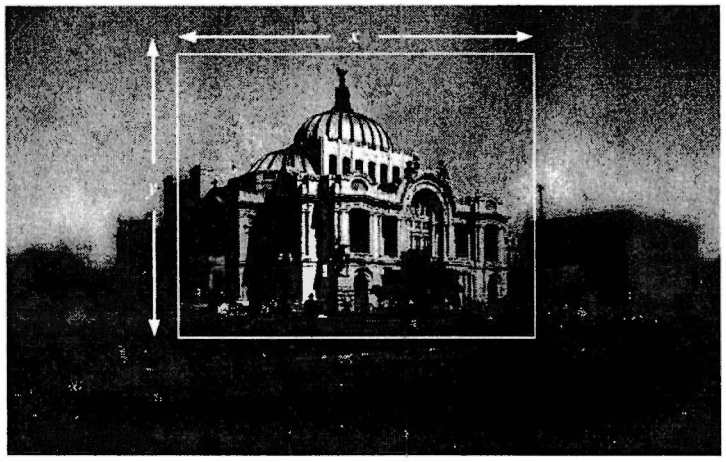


Figure 2.1: Finite size window

Having these two restrictions in mind, visual information is determined by the wavelengths and light intensities that passes through each point of the window and reach the viewer eyes, hence, the problem of representing visual information in a numerical form is reduced as we only have to represent the intensity distribution of the finite size window. If a cartesian plane is imposed on the window we can represent the perceived intensity at any point $(x, y)$. Thus $I(x, y)$ represents the visual information or *image* at the instant of consideration, later we will add the parameter $t$ for images that vary with time.

There are certain images $I(x, y)$ that can be exactly specified, for example a light square over a dark background could be described as indicated in equation 2.1, the main problem is that no such exact specification is possible for real life situations with a limited amount of numerical data, hence an aproximation of $I$ must be done if it is to be processed in a practical system.

$$I(x,y) = \begin{cases} 1 & a \leq x \leq b,\ c \leq y \leq d \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

As the majority of images must be stored, transmitted or processed a binary representation of data is required; $I(x, y)$ must be represented by a finite number of bits, the main

[1]Picture taken from www.bellasartes.gob.mx

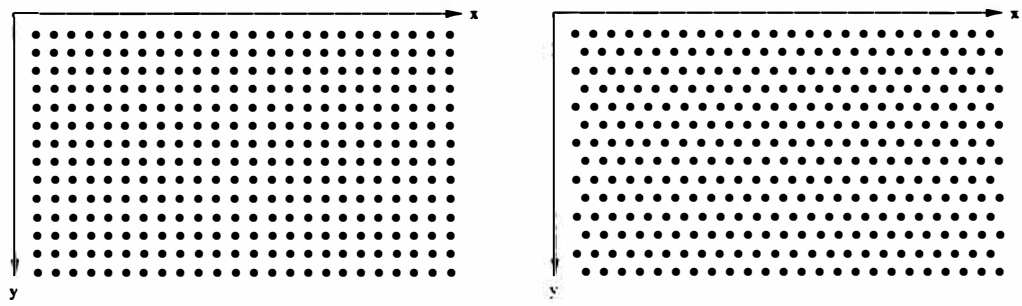
problem relies in the wordlength that must be chosen; as we know, bit representation is an aproximation to the closest level of intensity, so, to increase the accuracy of the representation we must increase the number of bits per word; this decision must be seriously considered as it might lead to an increase in material complexity, making the system too expensive in terms of silicon area and can affect the maximum operating frequency and power consumption of the processor, so it is necessary to estimate the *minimum* representational fidelity required for the particular application to design the processing system accordingly [5].

Deciding what degree of accuracy is needed in a system is not an easy task, starting with $I(x, y)$ we must find a digitization scheme that comply with computational constraints and then construct the corresponding representation of data; from this data we must reconstruct an approximate replica $\tilde{I}(x, y)$ of the original image. Next we must assume that a distortion measure $D(I, \tilde{I}) \geq 0$ exists to indicate how accurate is $\tilde{I}(x, y)$ with respect to $I(x, y)$; although this coding issues are very specific, there are several difficulties associated to it, first of all, image processing systems are designed to handle a variety of image formats, this leads to different distortion measurements per format, so it could be possible to have image with high distortion for all coding schemes, thus making impractical to work with. The next problem is that the distortion measure can be out of the desireable constraints making the system unfeasible; even though this issues need to be considered, there are other aspects that can be used to evaluate a coding system like the complexity, robustness, resilience, compatibility and scalability [6].

Now that is clear that we cannot describe $I(x, y)$ with absolute accuracy it is evident that we require a finite number of bits to represent the intensity levels of $I$ with an acceptable degree of accuracy; if time is considered as a parameter, then the whole process of sampling and quantizing is performed in a repeating cycle known as PCM coding; the set of sampling points of the image can be considered as a bidimensional array where samples are placed both on vertical and horizontal directions as shown in figure 2.2a, a slight variation of the sampling array is shown in figure 2.2b, notice that some rows are shifted by half sampling period; this samples are commonly known as *picture elemets* or ***pixels***, in this way, $pixel(x, y)$ is exactly the same than $I(x, y)$.

Pixels should be located close together to measure the intensity variations of the area of interest, but, if we locate them too close we will have an unnecessary amount of information, so, *the required sampling density depends on the intensity variations that the processor must accomodate*, this sampling density is closely related to *resolution*; there are some techniques to measure resolution, one is the minimum distance between two lines or two pixels within the image, another metric is related to frequency contents of the image, this is especially useful when working with Fourier transform of any other Fourier related orthogonal transform; keeping this in mind, we can say that sampling density is chosen depending on either resolution or processing costs.

Once the sampling density has been determined, there might be some fast variations in




(a) Rectangular Array (b) Offset Array

Figure 2.2: Pixel Sampling Points

the intensity levels within $pixel(x, y)$ making difficult to measure accurately with the selected sampling rate, this happens often when optical devices have a larger resolution than needed, this fast transitions tend to distort the visual information, having as a result an image that does not represent the real data; in signal processing this phenomenon is known as pre-alias. To prevent this effect we require to take an average of samples surrounding each $(x, y)$ pixel; even if intensity transitions are too fast or too small they would not change the final representation of visual information.

An easy method is to form a weighted average over the pixel neighborhood; for this we need a weighting function $h_c(u, v)$ that peaks only at $(0, 0)$ and falls in any other case, the average or filtered image is then expressed as shown in equation 2.2; $I(w, z)$ is calculated as the instantaneous value at pixel (w,z) (see 2.3. The intensity at each point of the image is replaced by a weighted average, this process is known as pre-filtering since $I$ has not been sampled.

$$\bar{I}(x, y) = \int_{y-\delta}^{y+\delta} \int_{x-\delta}^{x+\delta} I(w, z) h_c(x - w, y - z) dw dz \tag{2.2}$$

$$I(w, z) = \delta(w)\delta(z) \tag{2.3}$$

Pre-filtering helps to reduce the amount of information as only a finite number of samples is represented, fast variations of intensity are removed and intensity values are quentized to finite accuracy. Having the quantized pixels of the filtered image we might need to reconstruct a replica $\tilde{I}(x, y)$ of the original image, the main problem is deciding what intensity values are going to be assigned at $(x, y)$ points that do not correspond to a sampling point; we might choose either to assign a zero or a near value; in both cases an artificial pattern will appear in $\tilde{I}(x, y)$ causing a post-aliasing effect, this problem can be diminished if we interpolate between pixels to obtain smooth intensity transitions. Let $h_d(u, v)$ be an interpolation function, so that the reconstructed replica is given by equation 2.4, if the sampled replica is defined using

Dirac delta functions, then equation 2.4 can be re-written as shown in equation 2.5 and the reconstructed image is given by equation 2.6, the sequence of sampling and reconstruction of an image is shown in figure 2.3.

$$\tilde{I}(x,y) = \sum_{k} \bar{I}(x_k, y_k) h_d(x - x_k, y - y_k) \tag{2.4}$$

$$I_s(x,y) = \sum_{k}^{N} \bar{I}(x_k, y_k) \delta(x - x_k) \delta(y - y_k) \tag{2.5}$$

$$\tilde{I}(x,y) = \int \int I_s(w,z) h_d(x - w, y - z) dw dz \tag{2.6}$$

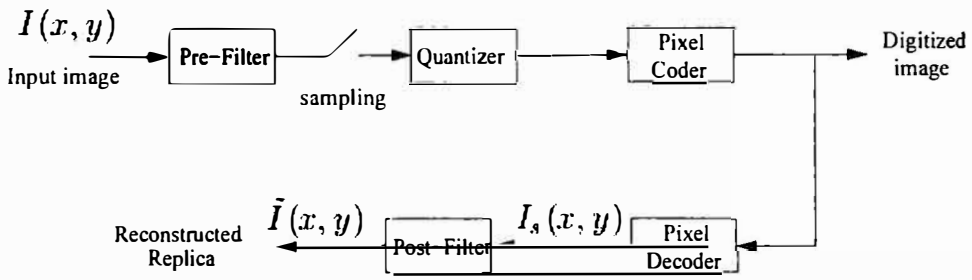



Figure 2.3: Sampling and Reconstruction sequence

## 2.2 Compression

Compression is a process intended to produce a compact *digital representation* of a signal [2] [3]; when the signal is defined as a video stream or an audio segment the real problem of compression is to minimize the bit rate of the digital representation; it is important to state that without compression, many applications would not be feasible as shown in table 2.1.

Audio, Image and Video signals hace repeated or irrelevant information known as redundant data. If we define $b$ and $b'$ as the number of bits in two different representations of the same information, then we can define $C$ as the *Compression Ratio*

$$C = \frac{b}{b'} \tag{2.7}$$

And from the compression ratio we can calculate the relative redundancy of data:

$$R = 1 - \frac{1}{C} \tag{2.8}$$

If we consider a compression ratio $C = 10$ $(10:1)$, means that 10 bits of data can be represented using one single bit in compressed form, when we calculate the relative redundancy of data $R = 0.9$, we notice that 90% of data is redundant and can be removed.

| Application | Uncompressed Data Rate | Compressed Data Rate |
|---|---|---|
| Voice<br>*8 ksamples/s, 8 bits/sample* | 64 Kbps | 2 – 4 Kbps |
| Audio conference<br>*8 ksamples/s, 16 bits/sample* | 128 Kbps | 6 – 64 Kbps |
| Video conference (15 fps)<br>*framesize* 352 × 240, *24 bits/pixel* | 30.41 Mbps | 64 – 768 Kbps |
| Digital audio<br>*44.1 ksamples/s, 16 bits/sample* | 1.5 Mbps | 128 – 768 Kbps |
| Digital video (30 fps)<br>*framesize* 352 × 240,*24 bits/pixel* | 60.83 Mbps | 1.5–4 Mbps |
| Broadcast video (30 fps)<br>*framesize* 720 × 480, *24 bits/pixel* | 248.83 Mbps | 3–8 Mbps |
| HDTV (59.94 fps)<br>*framesize* 1280 × 720, *24 bits/pixel* | 1.33 Gbps | 20 Mbps |

Table 2.1: Applications for Audio, Image and Video Compression

As stated before, compression refers to the process of reducing the amount of data required to represent a given quantity of information [3] and there are many reasons for using compression, the most relevant are [4]:

1. Compression extents reproduction time of an storage device.
2. Allows electronic component miniaturization, with less data to store we get the same result with increasingly smaller hardware.
3. Reduces bandwidth, thus it is useful for lowering costs.
4. If a fixed bandwidth is assigned, compression allows sending a better quality signal over the same space.

In digital image compression, $b$ is the number of bits required to represent an image as a two-dimensional intensity array [5], that is a matrix $f(x, y)$ of M columns and N rows where $(x, y)$ are discrete coordinates; although this representation is suitable for human eye because it makes easier the interpretation of data it is not the best or the most efficient way to represent it. In general, we can say that two-dimensional arrays have five different types of data redundancy, each of these must be identified and later we must analize if we can exploit them or not.

**Code Redundancy:** When the word-length of the 2D array utilizes more bits than the required to represent the luminance intensities we have a Code Redundancy, this type of data redundancy is presented when the codes assigned to a group of values do not represent accurately each value probability. This implies that there are intensities more

probable than others, therefore a binary code is required to represent both the most and the least recurring intensity, but we must assignt codes to intermediate values that might not be present in the image, so the code is redundant.

**Spatial Redundancy:** Also known as *Spatial Correlation* refers to the fact that within an image or a video frame exists a significant correlation among neighbor samples and information is unnecesarily replicated. Most images are composed by objects that have regular shape and regular reflectance.

**Spectral Correlation:** When data is acquired from multiple sources there exists significant correlation among samples from these sensors.

**Temporal Redundancy:** For temporal data such as video there is significant correlation between samples in different segments of time.

**Irrelevant Information:** Most 2D arrays have information that is ignored and is even imperceptible to human eye, this kind of information is considered redundant because it is not used.

*Image and Video Compression is achieved when one or more redundancies are reduced or eliminated.* [3]

Normally we choose only one type of redundancy to remove; therefore there are many ways to achieve data compression, but, the increased commercial interests have ignited the efforts of international standardization of image and video coding [1].

A block diagram of the compression process is shown in figure 2.4, the source coder performs a compression process to reduce the input data rate to a level that can be conveyed by the transmission medium; after the source coding a second level of codification is required to translate the compressed bit stream into a signal suitable for either transmission or storage, in most systems source and channel coding are different processes, but in recent years have been developed methods to perform combined source and channel coding. To reconstruct the signal one simply needs to reverse the codification process.

Standardization enables image and video material from different sources to be:

1. Processed on different hardware platforms
2. Stored on different storage devices
3. Transmitted on different communication networks

The achieved interoperability opens a huge market for image and video acquiring-reproducing equipments at the same time that provides the consumers a wide selection of services, so standardization gives manufactures the great opportunity to have large scale productions at considerably low costs. The main issue of compression is that there are many methods to compress a given signal, some are Model-Based methods while others are part of a great

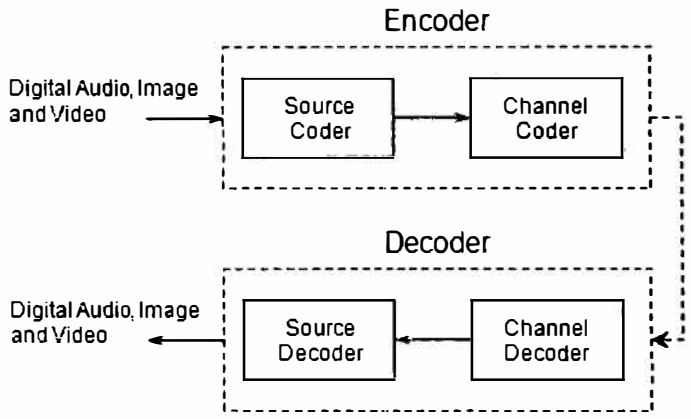



Figure 2.4: Generic compression system

family of Waveform-Based methods that include lossy and lossless compression. A common classification of compression methods is shown in figure 2.5 [2].

### 2.2.1 Lossless Compression

There are many applications where the decoder has to reconstruct the original data without any loss; lossless compression scheme is by definition a reversible process where the reconstructed data is an exact replica (sample by sample) of the original data. This compression method can be used for any kind of data and is achieved by removing redundant information; in general there are two lossless techniques:

- Run-Length Encoding: Data normally presents long strings with the same value, to reduce the string lenghts we simply count the values and construct a new one with this number and the value that represents, for example, the string:

  $$GGGGGGGGGGGGGBBBRRRRRRRRGGGGGGGGGGGBBBRRRRR$$

  Can be coded as:

  $$14G3B7R11G4B5R$$

- Entropy Coding: Every sample is represented by a unique value called *Symbol*, equal samples will be represente with the same symbol, this technique has the advantage that symbols can have different lengths, so small-length symbols are used to represent frequent samples (or data) and larger symbols are used to represent non-common data, an example of this coding technique is shown in table 2.2.

The choice of a lossless compression technique involves a triple tradeoff between Coding Efficiency, Coding Delay and Coder Complexity as shown in figure 2.6

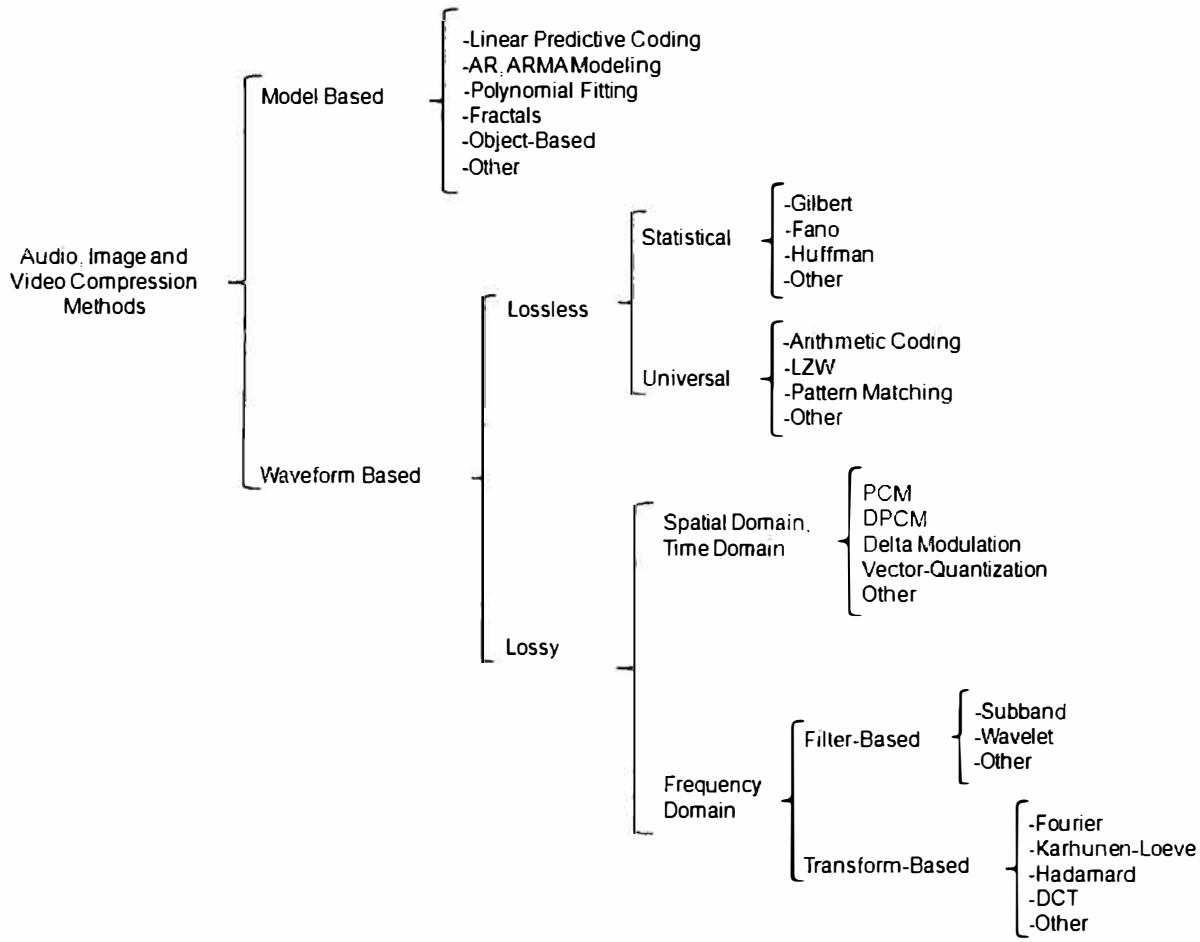



Figure 2.5: Taxonomy of Compression Methods

| Symbol | Probability | Binary Code |
|---|---|---|
| A | 0.5 | 0 |
| B | 0.3 | 10 |
| C | 0.1 | 110 |
| D | 0.1 | 111 |

Table 2.2: Entropy Coding with Variable Length Symbols

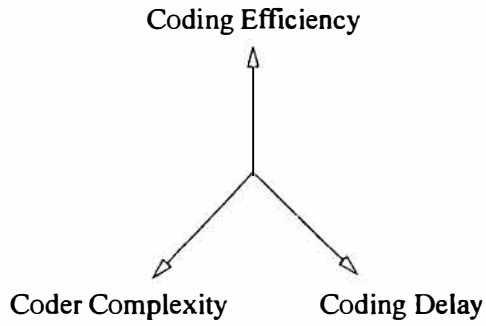



Figure 2.6: Lossless Compression Tradeoffs

**Coding Efficiency:** This parameter is usually measured in bits per second (bps) and is limited by the entropy of the source signal; if the source has a large entropy it will be difficult to compress (e.g White Noise); when data source is redundant, coding efficiency increases as the entropy or randomness of the source is low.

**Coding Complexity:** Refers to the computational requirements that must be met in order to compress a signal, the most important are:

- Memory consumption
- Power consumption
- Number of operations per second
- Hardware Implementation

**Coding Delay:** This parameter measures the time required to code or decode a signal, it is possible to accelerate the compression-decompression process implementing parallel or pipeline processing techniques, but this may be impractical in terms of coding complexity.

### 2.2.2 Lossy Compression

Many applications in image and video processing do not require the reconstructed data to be identical to the original data, hence some amount of loss in the reconstruction is allowed; any compression method that results in an imperfect reconstruction is by definition a Lossy Compression Method; These methods are irreversible, so the signal quality is the most important factor as it rapidly degrades as data goes through different the compression processes. The selection of an specific lossy compression method is not a trivial task and involves a four-way tradeoff along Signal Quality, Coding Efficiency, Coding Complexity and Coding Delay (Figure 2.7). The signal quality is the key to lossy compression methods because there is no way to measure the quality of a compression system, in fact these compression methods are also known as *Perceptual Coding Systems* and the only thing that matters is the subjective effect achieved in the receiver. This degree of freedom allows lossy compression methods to

reach higher compression ratios than lossless compression schemes.

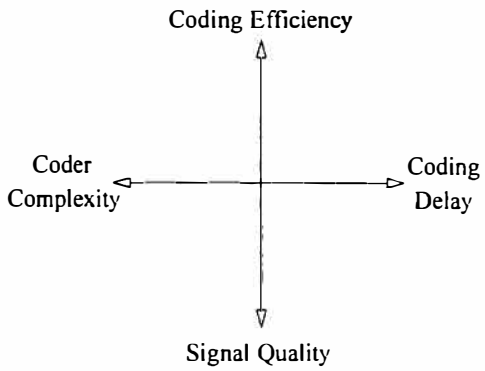



Figure 2.7: Lossy Compression Tradeoffs

### 2.2.3 Considerations in Compression Method selection

When choosing a compression method, many parameters have to be considered to be sure that the selected method is compliant with the application requirements; the first decision is whether a lossy or lossless scheme is needed, this is usually dictated by the coding efficiency requirements, we must take into account that even in lossy compression processes the desired coding efficiency might not be achieved, this is a common case when there are specific constraints on output signal quality.

As mentioned before, complexity tradeoffs between encoder and decoder must be considered, there are some applications that require symmetrical encoding while others require only low complexity in the decoder; Also we must consider that some compression methods are more robust than others, therefore resilience to transmission errors needs to be considered in terms of the additional material complexity in the decoder. Data representation must also be considered as many compression standards code data through various stages, so we must ensure that the original can be restored no matter how many coding stages were between the coding and encoding or even if the encoding-decoding process is performed in tandem as it is used in video editing tasks.

All issues in compression method selection must be considered but we must pay special attention to interplay with other data modalities and internetworking with other systems; in the former we must ensure that several data modalities are supported so that the compression methods should have common elements [2], in the latter, transcoding between compression methods might be required because depending on the storage or transmission media we will need to exploit different types of redundancies at different times.

## 2.3 Video Standards

In the previous section we stated some of the most relevant reasons for standardizing, but it is necessary to have clear that standards do not necessarily represent the best technical solutions, but rather attempt to achieve a compromise between flexibility, complexity and compression efficiency achieved [7]; Video coding standardization activities began around 1980's as an initiative of the International Telegraph and Telephone Consultative Comittee (now the ITU-T); this efforts were followed by the CCIR, the International Standards Organization(ISO) and the International Electrotechnical Commission (IEC), since then many standards have appeared among the years [1]; table 2.3 summarizes the most relevant characteristics of the most important video standards.

| Standard | Throughput | Image Size | Chroma Format |
|---|---|---|---|
| H.120 | 1.544 Mbps<br>2.048 Mbps | - | - |
| H.261 | 64 Kbps<br>384 Kbps | CIF | 4:2:0 |
| MPEG-1 | 1.5 Mbps | SIF | 4:2:0 |
| MPEG-2 | 10 Mbps | QCIF<br>CIF<br>4CIF<br>16CIF | 4:2:0<br>4:2:2<br>4:4:4 |
| H.263 | 64 Kbps | SQCIF<br>QCIF<br>CIF<br>4CIF<br>16CIF | 4:2:0<br>4:2:2 |
| MPEG-4 | 5Kbps to<br>10Mbps | SQCIF to<br>HDTV | 4:2:0<br>4:2:2<br>4:4:4 |

Table 2.3: Video Coding Standards Summary

### 2.3.1 H.120

Study Group XV of CCITT was responsible of the first international effort towards video coding standardization; during the first study period (1980–1984) the first recomendation was issued as the recommendation H.120, later on 1988 the group issued the second and definitive version. This standard targeted videoconference applications at primary rates of 1.544 Mbps and 2.048 Mbps; this standard has three parts or sections, part 1 was intended for 625 lines at $50Hz$ at 2.040 Mbps. Part 2 was designed for international use, so it was

suitable for 625 and 525 lines at 50 or 60 $Hz$ an 1.544 Mbps, finally, part 3 was designed for 525 lines at 60 $Hz$ at 1.544 Mbps. Parts 1 and 2 use conditional replenishment with intrafield Differential Pulse Code Modulation for changed regions while part 3 uses intrafield prediction, backward prediction and motion compensated interfield prediction. This coding difference between the three parts of H.120 was the reason why it never became a commercial success.

### 2.3.2 H.261

After the release of the first recommendation of H.120, the study group XV of the CCITT decided to define a video standard for videoconferencing applications over ISDN at transmission rates lower or equal to 2Mbps ($\leq$ 2 Mbps). The first version of H.261 was released in 1989 and was meant to provide audio and video services at variable data rates: $p \times 64$ Kbps ($p = 1 \ldots 30$); this first version became an international standard in 1991 because the same algorithm could be applied to a variety of data rates, becoming the first video coding standard that succeded worldwide. H.261's hybrid Motion Compensation based on Differential Pulse Code Modulation and Discrete Cosine Transform technique, zigzag scanning, Run-Length Encoding and Variable Length Coding are now necessary elements in most video coding standards.

### 2.3.3 MPEG-1

The Moving Picture Experts Group was created in 1988 under the Subcomitee 2 of ISO, later it was renamed as the Work Group 11 (WG11) of the subcomitee 29 under the Joint Technical Comitee of ISO/IEC, so, the official name of MPEG is ISO/IEC JTC1/SC29/WG11. The main task of this group was to develop a video coding standard for digital storage applications at up to 1.5 Mbps [1]. In 1992 the draft ISO/IEC 11172 became an international standard, MPEG's algorithm is very similar to H.261 but it has more sofisticated techniques like bidirectional prediction and Half-pel Motion Compensation; although the standard was developed for storage applications, the coder is not defined, so the manufacturers can use their own coding algorithms as long as they are compliant with the bitstream syntax. MPEG is an asymetric system (shown in figure 2.8) where the coder can be based on algorithms or be adaptive, this is, the coder is able to perform different tasks depending of the nature of input data; the decoder only performs preestablished tasks.

There have been defined some boundaries that differenciate MPEG data streams from Non-MPEG streams; any data flow compliant with the defined MPEG bitstream syntax can be decoded, if any of the parameters is outside the boundaries, then the data flow can not be decoded. Table 2.4 shows the limits of some MPEG parameters.

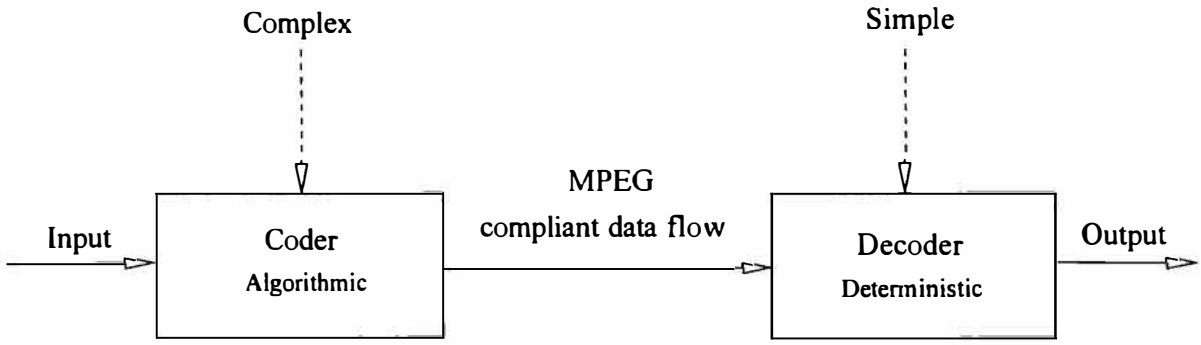



Figure 2.8: Asymmetric Compression System

| Parameter | Limit |
|---|---|
| Horizontal Image Size | $\leq$ 768 pixels |
| Vertical Image Size | $\leq$ 576 lines |
| Number of Macroblocks | $\leq$ 396 |
| Number of Macroblocks per second | $\leq 396 \times 25 = 9900$ |
| Frame Rate | $\leq$ 30fps |
| Video Buffer Size | $\geq 2,621,440$bits |
| Bit Rate | $\leq 1,856,000$bps |

Table 2.4: MPEG-1 Limitations

#### 2.3.3.1 Hierarchy

The highest level of MPEG is a sequence or succession of images, this succession can have an arbitrary length and might represent a video clip, a complete program, a series of programs or even a movie; inside the sequence the next level in MPEG hierarchy is defined as the Group of Pictures (GOP); typical MPEG-1 data flow consists in a repetitive structure of GOPs. In the simplest codification (that is without temporal compression) the GOP can be a single image, nevertheless in practice we consider a GOP as a sequence of 10 to 30 frames that can also be considered as the interval between intra frames. GOP length is a determinant factor in compression efficiency, short GOPs imply a bad compression and long GOPs tend to diminish the quality of reproduction; Group of Pictures structure is shown in figure 2.9.

The next element in MPEG hierarchy is the frame; unlike other video coding standards MPEG-1 does not consider the interlacing concept, so every image is treated as a complete frame even if it is one field, so it is common that in this case the achieved compression is inefficient. The following element is the macroblock (figure 2.10), this entities are used to represent small areas of 16 $\times$ 16 pixels of luminance, they are numbered in the normal scanning order (top to bottom, from left to right); each macroblock is divided into 8 $\times$ 8 pixel blocks, a block is the minimum unit that a discrete cosine transform processor can work with.

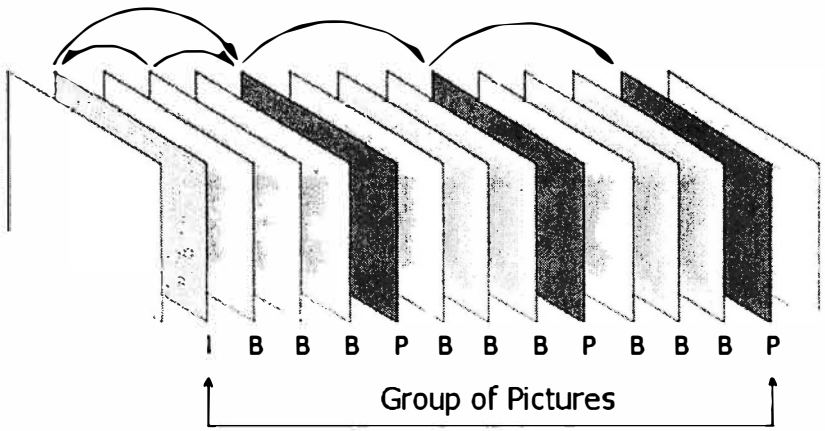



Figure 2.9: Group of Pictures

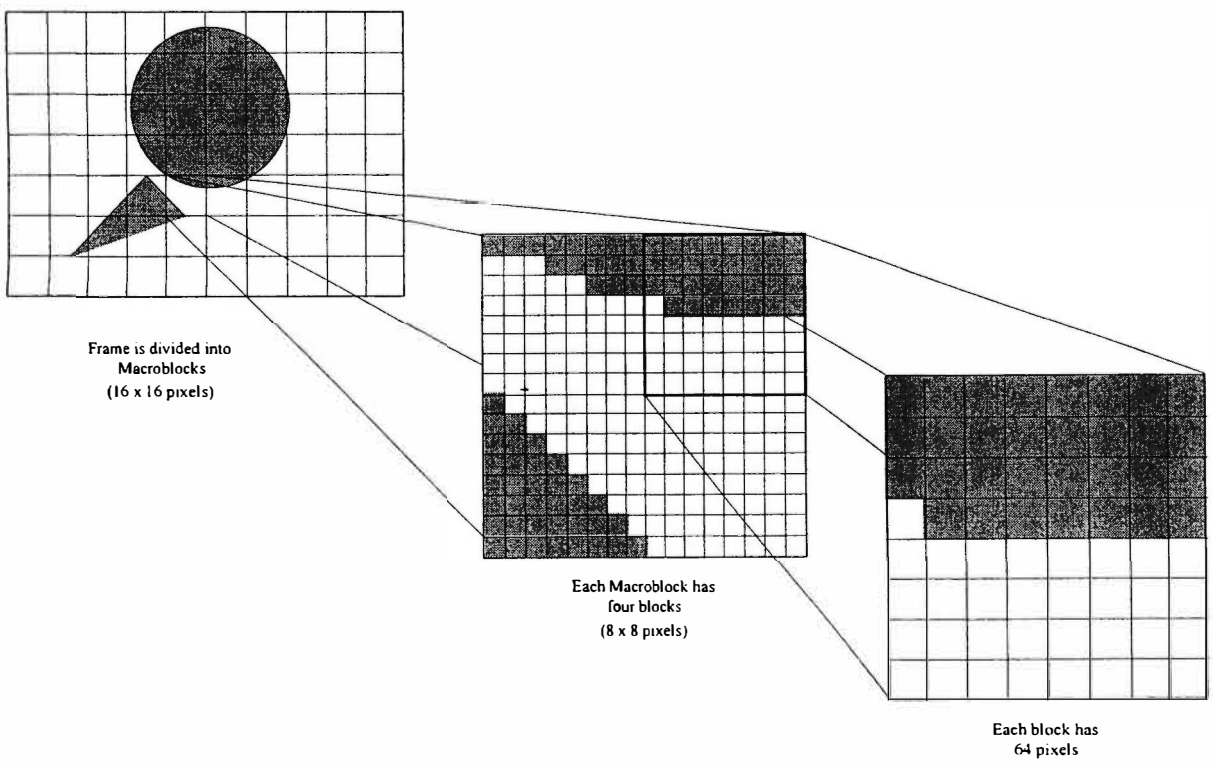



Figure 2.10: Macroblock

Macroblocks are used to reduce the complexity of bidirectional frames generation, also they are useful to find areas with the same color and are useful to identify the redundancies within intra and bidirectional frames. When we group macroblocks depending on their average intensity we have a slice, this entity helps the decoder to recover in case of synchronization errors.

There are two types of frame in MPEG, so, to clarify the function of each frame their definition is:

**Intra frame:** This frames are coded using only the information within the frame, this means that intra frames are spatially coded

**Non-Intra frame:** Also known as inter frames, this type is subdivided into Bidirectional and Predicted frames, both of them are coded taking advantage of temporal redundancies, so it is obvious that they use information of previously coded frames.

- Bidirectional Frame: are coded using information of the previous and the next frame, in fact, the "future" reference is the next I or P frame.
- Predicted Frame: are coded using information of the previous I or P frame

The relationship between I, B and P frames is shown in figure 2.11.

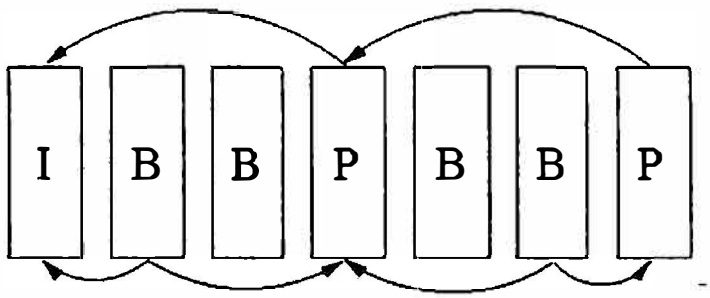



Figure 2.11: MPEG frames

#### 2.3.3.2 Group of Pictures

Images are categorized in a different way, I and P frames are known as anchors because they are used as a reference for B frame codification using motion compensation; a GOP starts with an I frame that must be coded first to start de sequence, if there is no I frame there is no previous information and it is impossible to have a reference for motion compensation. It is possible to have many bidirectional frames after de intra frame because B frames are coded and transmitted immediately after I frames.

The first P frame is coded using the previous I frame to have a temporal reference, the following P frames are coded using the first P frame as a reference, this implies a serious disadvantage when an error occurs in this frames because it propagates indefinitely as the bad P frame will be taken as a reference in the future P frames. B frames are coded using the previous anchor frame as a reference for the forward prediction and they use the following anchor frame for the bakwards prediction, this means that B frames are never use as a reference for prediction; in figure 2.12 is shown a typical closed GOP and the way it is unfolded.

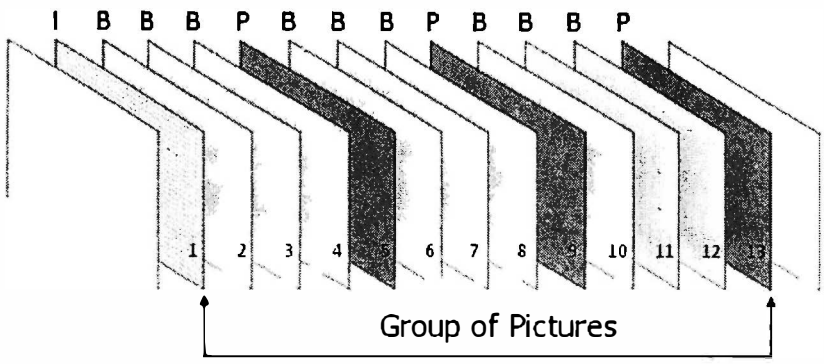



Figure 2.12: Group of Pictures deployment

A GOP is considered closed when all predictions occur within a block, there are also open GOPs with an $I - B - I - B - I \ldots$ structure that are usually very efficient, the main problem of this structure is that there is no way to separate the data flow; Closed GOPs are also known as regular GOP, this is because there is a fixed patter of P and B frames between I frames; a regular GOP can be characterized in terms of two parameters N and M, where N is the distance between P frames an M is the distance between I frames. B frames can be decoded only if the previous and next anchor frame have been sent to decoder, the figure 2.13 shows the transmission order of a GOP.

### 2.3.4 MPEG-2

In 1990 began the efforts to generate a new standard to cover the applications that could not be attended with MPEG-1, in particular, this standard was foreseen to video qualities similar to NTSC/PAL, this implied data rates up to 10 Mbps; it was called MPEG-2 because it was seen as the second phase of MPEG-1. In 1992 the study group XV of ITU-T joined the group ISO/IEC JTC1/SC29/WG11 to design the video coder for ATM networks. In 1993 the study group realized that the scope of MPEG-2 was extended beyond HDTV, so the efforts to generate MPEG-3 where dropped.

Like MPEG-1, MPEG-2 coding standard is flexible, compatible and generic, but it has some aditional characteristics that were not covered with the previous standard, the most

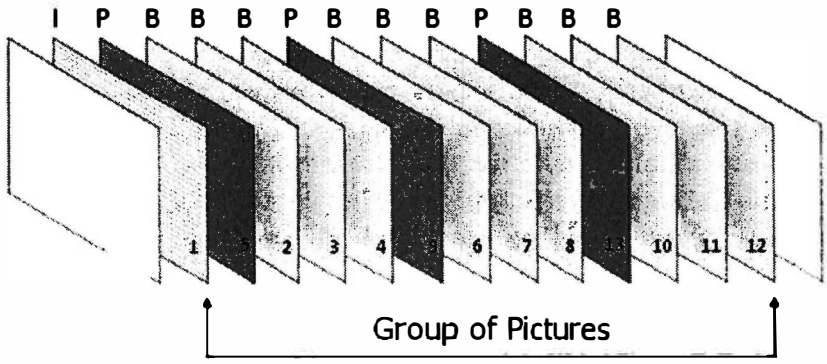



Figure 2.13: Transmission of a GOP

important of them is the capability to process interlaced video; in MPEG-2 were defined some profiles to describe the functionality of the standard, also were defined the levels to describe the resolutions that can be used. This characteristic allowed MPEG-2 to replace MPEG-1 in areas like cable tv, ATM networks and Broadcast TV.

#### 2.3.4.1 Macroblock Structure

Every profile and level of MPEG-2 supports 4:2:0 codification, but there is a slight difference than MPEG-1: the position of the color samples was redefined; Some profiles suppor 4:2:2 and 4:4:4 codifications, the macroblock structures and position of color samples is shown in figure 2.14 [4].

#### 2.3.4.2 Slice Structure

A slice is a group of macroblocks placed in scanning order and can be decoded without a reference of other slice; in MPEG-1 there was no restriction in the slice size, it could be a macroblock or an entire image. In MPEG-2 the maximum length of a slice is defined as a complete row of an image or less, but it cannot surpass this length.

#### 2.3.4.3 Quantization

MPEG-1 allowed an 8 bit precision for the DC coefficients of the Discrete Cosine Transform; somo profiles of MPEG-2 allow 9 or 10 bits of precision for the same coefficients.

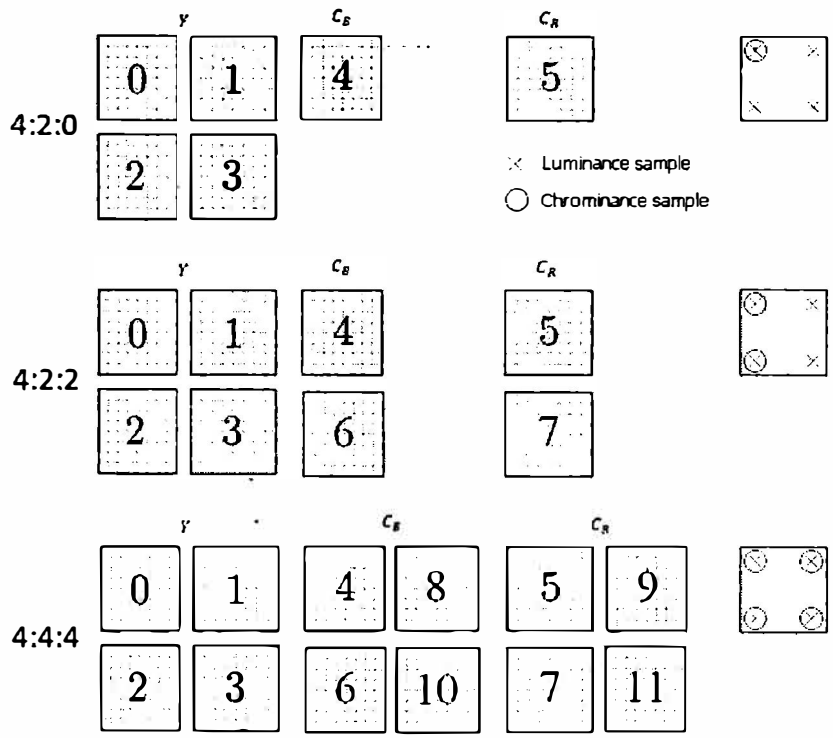



Figure 2.14: MPEG-2 Macroblocks and sample positions

#### 2.3.4.4 Calculated Motion Vectors

This tool was designed for MPEG streams transmitted over channels with high probabilities of macroblock loss. This is a problem, especially for I and P frames because in the worst case scenario, errors will propagate indefinetely. MPEG-2 has hidden motion vectors that are transmitted with intra coded macroblocks, if any macroblock is lost, the previous Calculated Motion Vector will point to a similar macroblock that will substitute it in the decoder.

#### 2.3.4.5 Profiles and Levels of MPEG-2

The characteristic that made MPEG a video coding success was the fact that the decoder was able to translate any signal as long as the bit rate was compliant to MPEG specifications, for MPEG-2 the situation was far more complicated because it was too expensive to have a decoder that supported the new syntax elements for all allowed data rates; to solution this problem, the standard was divided into two categories. The tools or syntax elements required to decode are defined in MPEG-2 *profiles*, *levels* define the range of allowed data rates, table 2.5 shows MPEG-2 profile constraints; table 2.6 shows different MPEG-2 levels [8].

| Constraint | Profile | | | | | | |
|---|---|---|---|---|---|---|---|
| | Nonscalable | | | | Scalable | | |
| | Simple | Mail | Multiview | 4:2:2 | SNR | Spatial | High |
| chroma format | 4:2:0 | 4:2:0 | 4:2:0 | 4:2:0 or 4:2:2 | 4:2:0 | 4:2:0 | 4:2:0 or 4:2:2 |
| picture types | I, P | I, P, B | I, P, B | I, P, B | I, P, B | I, P, B | I, P, B |
| scalable modes | - | - | Temporal | - | SNR | SNR or Spatial | SNR or Spatial |
| intra precision (bits) | 8, 9, 10 | 8, 9, 10 | 8, 9, 10 | 8, 9, 10, 11 | 8, 9, 10 | 8, 9, 10 | 8, 9, 10, 11 |
| sequence scalable extension | no | no | yes | no | yes | yes | yes |
| picture spatial scalable extension | no | no | no | no | no | yes | yes |
| picture temporal scalable extension | no | no | yes | no | no | no | no |
| repeat first field | constrained | | unconstrained | constrained | unconstrained | | |

Table 2.5: MPEG-2 Profile limits

| Constraint | Profile | | | | | | |
|---|---|---|---|---|---|---|---|
| | Nonscalable | | | | Scalable | | |
| | Simple | Mail | Multiview | 4:2:2 | SNR | Spatial | High |
| high | - | yes | - | yes | - | - | yes |
| high 1440 | - | yes | - | - | - | yes | yes |
| main | yes | yes | yes | yes | yes | - | yes |
| low | - | yes | - | - | yes | - | - |

Table 2.6: MPEG-2 Levels

#### 2.3.4.6 Tools for Interlacing

Interlacing is an image scanning system that has been used since the origins of television, this technique divides an image or frame into two fields, each field contains the half of lines that compose a field. This process is necessary to increment de flicker rate of the image so that it will be imperceptible without affecting the TV bandwidth. In 525 line systems like NTSC, scanned is performed in $\frac{1}{30}$ of a second and it is displayed at $60Hz$; This displaying method has the inconvenience of a temporal overlap of vertical and temporal spectres; Transcoding interlaced scanning video to progressive scanning is not a trivial task, it can only be acomplished using Motion Estimation techniques.

As mentioned before, MPEG-1 does not consider interlaced video, so if a field is to be coded, every line will be duplicated to complete a frame, and of course this is an issue that affects coding efficiency. MPEG-2 has the necessary tools to process interlaced images and the combination is very efficient, it is nearly impossible to differenciate them from a progressive scanned image.

### 2.3.5 H.263

H.263 arose in response to the need to transmit digital video over circuit switched networks and mobile networks, study group XV of ITU-T began the standardization process whose objective was to develop a video coding standards for low data rates (as low as 64 Kbps) using the coding structure of H.261. In 1996 the H.263 recommendation was approved.

If H.261 and H.263 are compared, we will find that H.263 provides the same subjective quality than its predecessor, the difference is that H.263 requires less than the half of the transmission rate of H.261, this performance improvement was possible thanks to coding technique optimization and the optional use of advanced coding techniques.

Some improvements of H.263 over H.261 are: support of different image formats, motion compensation with half-pel resolution, three dimensional run-length encoding, optimized variable length coding tables, extra headers to increase the capability of error recovery, optimized addressing of macroblocks, arithmetic coding (optional), four motion vectors per macroblock and bidirectional prediction (optional).

#### 2.3.5.1 H.263+

It is considered the second version of H.263, was developed by the Advanced Video Experts Group ITU-T/SG16/Q15 (previously known as ITU-T/SG15) and was published in 1998. H.263+ added twelve optional characteristics that allowed among others to: have custom image sizes, have custom clock rates, increase compression efficiency, improve error adjust-

ment in wireless networks and ensure backward compatibility.

### 2.3.6 MPEG-4

In 1993 the MPEG group began a new standardization activity under the name of MPEG-4, the goal of the group was to achieve higher compression efficiency using the available coding techniques; in 1994 the group determined that there were to few improvements compared to H.263 and H.263+ standards. The goal was then redefined and a particular attention was given to the convergence of three industries that have been treated as separate applications: communications, computation and TV. This scope helped the study group to determine that MPEG-4 should be functional in future applications that for sure would not be supported by existing standards; eight new features were added and classified into three categories.

1. Content based interaction
    - Tools for multimedia data
    - Content manipulation and data flow edition
    - Synthetic and hybrid data coding
    - improved temporal random access
2. Compression
    - Coding efficiency
    - Simultaneous dataflow codification
3. Universal Access
    - Content based scalability
    - Robustness in error susceptible environments

The first version of MPEG-4 was issued in 1998 and the second in 1999, this standard is officially known as ISO/IEC 14496 and entitled as Generic Audio Visual Object Coder, as its name states, it is a generic standard designed to cover a wide range of:

- Bit rates: typically from 5 Kbps to 10 Mbps.
- Image format: Interlaced or progressive.
- Frame rates: From still images to video sequences.
- Communication networks: LAN, WLAN, etc.
- Input information: Synthesized or Natural.

MPEG-4 utilizes an object-based representation model, this means that every scene is coded and manipulated as a single audiovisual object, this standard is an evolution of the previous block-based standards as it integrates second generation coding techniques as the pyramid coding and segment coding.

### 2.3.7 H.264/AVC

The Joint Video Team (JVT) of ISO/IEC and the Video Coding Experts Group (VCEG) of ITU-T developed a new video coding standard known officially as recommendation ITU-T H.264, ISO/IEC MPEG-4 part 10, Advanced Video Coding or simply H.264/AVC. Work started on 1998 when the VCEG called for proposals for a proyect known as H.26L. The main goal was to significantly improve the coding efficiency of existent standards; the first version of H.26L was issued in 1999. At the end of 2001 the JVT was created and had as a mission to conclude the new coding standard that was finally published as H.264/AVC in 2003.This standard has a high coding efficiency and is suitable for applications like broadcast transmission over different media or video storage in optical and magnetic devices.

H.264/AVC is composed by two layers, the first one is the Video Coding Layer (VCL) designed to compress video efficiently; the second layer is the Network Abstraction Layer (NAL), the NAL is used to give format to the video representation of the VCL and to provide the header information needed to handle different transport layers or storage media. Compared to other video coding standards, H.264/AVC is much more efficient in data coding, is very robust and is very flexible as it can operate over a great variety of network environments.

There are two strategies that can be used to achieve a higher coding efficiency:

1. Increase the accuracy in image prediction
2. Use entropy coding to reduce the average bits per symbol

Many tools to improve inter and intra frame prediction have been developed, some of them are the following:

- Variable Block Size for motion compensation: There are seven different block sizes (the smallest one is $4 \times 4$), this strategy is based in the fact that precision accuracy is improved using small blocks .
- Multiple references frame for motion compensation: P and B frames prediction requires multiple reference frames, it is obvious that a better prediction is achieved if we have more than one reference the precision of the prediction is upgraded.
- Spatial Prediction: Is a directional intra frame prediction that uses as a reference previously coded areas whithin the same frame
- Skip mode: For predicted frames

- Direct mode: For bidirectional frames [2]

## 2.4 Video Processing Architectures

In 1997, Trimberger, Carberry and Johnson [9] developed a Time Multiplexed FPGA; the main idea is that one single FPGA can have one active configuration and eight inactive configurations stored in on-chip memory (although it can be stored in external memory devices). These inactive configurations can be viewed as configuration planes; every plane is a very large word of memory, when the device is reconfigured every bit in the logic and interconnection array is updated from one of the memory planes in 25 ns.

A rapid-reconfigurable device is nothing but a curiosity without an operating model:

1. *Logic Engine Mode*: Designs are modeled as Mealy state Machines emulated as a single large design, FPGA can split combinational logic into pieces and look-up tables, FPGA is reconfigured many times per user cycle.

2. *Time Share Mode*: FPGA emulates several independent FPGAs, the device remains in a single configuration for multiple user clock cycles before switching to another configuration.

3. *Static Mode*: FPGA does not appear to be reconfiguring, hence the logic must always be resident and active.

4. *Mixed Mode*: A single application may have few memory planes and the logic part of the array will be split between the previous modes

From the above we can infer that memory access and location (on-chip/off-chip) are strong constraints that we will have to deal with in video processing tasks (very demanding in terms of hardware and memory usage).

To make this scenario even more difficult, there are many hardware platforms and algorithms to process video and images; in [10] the first comparison between GPUs and FPGAs and their use for video processing is made, both are capable of parallel processing but their architectures and features are compared exhaustedly in order to determine whether GPUs have surpassed FPGAs or not; In table 2.7 the most relevant features of FPGAs, ASICs, GPUs, DSPs and CPUs are compared.

Work presented in [10] is fundamental for this study because it establishes the guidelines to further comparison between programmable devices and the application; later on 2007 Wayne Luk revisited his previous work and established a new comparison betweeen GPUs and FPGAs under multi-processor architectures [11], to be as objective as possible they

[2] In both, Skip and Direct Modes, the reconstructed signal is obtained directly from the reference frame

| Feature | FPGA | ASIC | GPU | DSP | CPU |
|---|---|---|---|---|---|
| Architecture | Parallel Processing and pipelining | Parallel Processing and pipelining | Pipeline | Pipeline | Parallel Processing |
| Computation | Data path without instr. fetch and decode cycle, CLB | Standard Cell | High memory bandwidth, many floating point units | Multiple MAC units | Integer units, floating point units, vector units. |
| Memory Access | Can be minimized though data streaming and reuse | Custom | Less efficient than FPGAs | DMA, modified Von Neumann Architecture | Memory access instructions |
| Clock Rate | Slow | Custom | Fast | Fast | Fast |
| Number representation | Fixed point (floating point can be implemented at a very high HW cost) | Fixed point (floating point can be implemented at a very high HW cost) | Floating point | Fixed / Floating point | Floating Point |
| Throughput Rate | Is the clock rate | Custom | # of parallel pipelines multiplied by clock rate, divided by # of cycles taken by the task | Is the Clock rate | At least two instruction cycles |

Table 2.7: Comparison between FPGAs, ASICs, GPUs, DSPs and CPUs

tested five algorithms, all of these were used to test throughput in both FPGA and GPU platforms. In bicubic interpolation, GPUs showed superior performance over FPGAs, for 2-D Convolution as window size grows, the FPGA overcomes the GPU, in Histogram Equalization FPGAs outperforms GPUs by over three times; for Non-Full Search Motion Vector Estimation GPUs there is not a clear domination because the architecture was targeted at a desired throughput, for Primary Color Correction, despite the results obtained in [10], the GPU has a higher throughput than FPGA, this demonstrates that GPUs have been under constant evolution. From the comparison we can emphasize that GPUs are limited when the designer wants to get data from an entire frame but *are well suited to algorithms with arbitrary reuse patterns*, on the other hand, FPGAs have proven ability for parallel processing and true real time processing can be obtained with them [12].

FPGAs and DSPs are often compared in terms of the implementation of a certain algorithm, DSPs are optimized for external memory usage, so they are meant for on-line processing rather than real-time, on the other hand, FPGAs are suited for real-time and on-line processing, in fact, any DSP routine can be implemented in FPGAs, of course it is easier to implement algorithms written in high level languages directly on a DSP rather than describing the algorithm in HDL for an FPGA implementation. As stated in [13] FPGAs are uniquely suited to repetitive DSP tasks, like MAC operations because they can perform this operations in parallel, so as a result FPGAs vastly outperform DSPs. The main problem of DSP implementations over FPGAs is that DSP programmers are forced to re-formulate the algorithms in terms of logic gates and flip-flops rather than high level codes.

FPGAs are faster and more efficient in processing arithmetic operations than CPUs or GPUs. Compared to CPUs and GPUs, FPGAs are more efficient in terms of memory usage and memory access, the on-chip memory and parallel processing are a combination that neither CPUs nor GPUs are capable to overcome. Even number representation is a factor to compare, CPUs, GPUs and some DSPs use floating point representation, in the other hand FPGAs use fixed point [10], and this implies a faster and more efficient implementation because the same length is used to represent all numbers, of course floating point can be implemented in FPGAs but the hardware costs are very high.

In fact to determine which device should be used we must compare different factors:

- Power Consumption: CPUs and GPUs require more power than DSPs or FPGAs [11], as FPGAs have a lower clock speed there is no need to use heat sinks or fans.

- Design Time: Obviously this fact depends on whether we choose to work with Hardware Description Languages (HDL) or high level languages.

- Redundancy: both CPUs and GPUs are already inside a computer so redundancy is not a problem, redundancy implementation on FPGAs and DSPs implies an extra hardware cost.

- Multiple Pipelines: DSPs and FPGAs can compute multiple data in the same clock cycle, this capability helps both devices to overcome the redundancy naturally held by CPUs and GPUs

- Multiple cores: if a device is not capable to reach the target throughput, a number of them can be used to compute high amounts of data.

Even if we select a platform based on the recommendations issued in [10, 11], we must be aware that video processing algorithms are very demanding tasks and there is not a unique way to overcome this situation, sometimes it is necessary to divide the algorithms into a hardware and software part [14], if any portion of the algorithm is going to stay as it is, we should implement it into hardware to save time, in this way the fixed part of the algorithm is synthesized only once; dynamic parts or rapidly changing methods stay as software to give the developer the necessary flexibility to update them every time it is required.

### 2.4.1 Hardware-Software Applications

In [15] Dubois et al. propose a hardware accelerator for MPEG-4 based on Motion Estimation (ME) algorithm. ME is known to be the most computationally expensive stage of video processing, there are different families of approaches developed to solve the problem both in algorithmic and architectural point of view, the first family is the Reduced Search Algorithms (RSA), the goal of these algorithms is to reduce complexity by identifying possible candidate vectors within a search window; the second family consists of Multi-Resolution Searches, the third uses simplified matching criteria instead of Maximum Absolute Difference (MAD), finally, the fourth family consists on pre-processing the images in order to reduce them to binary data and then use a XOR to evaluate the MAD.

The main contribution of Dubois's work is the modular programmable coprocessor architecture for Reduced Search Motion Estimation, where user is supposed to define both window and macroblock size, inside the coprocessor all pixels are stored in the FPGA to reduce external memory access, therefore increasing processing speed, the coprocessor is constituted of an internal buffer, the external memory controller and a MAD estimation core; the memory controller is in charge of downloading a pattern and its search window into the internal buffer memory.

The motion estimation core has three elements, the first one controls the input throughput (provided by the external memory), the second provides two rows to the MAD estimator and the third one temporarily stores the results before transferring them to the external memory, the main idea of the architecture is to achieve the desired level of flexibility and reprogrammability for the search in order to separate the data access from the marching criterion algorithm.

Another demanding video processing algorithm is the Variable Length Coding (VLC), VLC is a lossless compression scheme and its current implementations are specific to a particular codec, so we can conclude that this particular algorithm is not suitable for multi-codec environments.

In [16] is presented a design for encoding variable length codes in a multi-codec environment, two flexible and low cost implementations are presented, the first consists of various codec's implemented on the same reconfigurable array, so, as the same array is shared for multiple codec's high flexibility and low cost is achieved. The second implementation is a multi-codec processor where several parts are implemented in hardware and many memories are used to cover multiple standards.

Vitabile [17] retakes the fact that image and video processing tasks are confined to large workstations or custom designed hardware because CPUs are too slow for the high speed required for real time video processing, but also they remind us that in most cases, image and video processing systems are designed in high level languages because developers are used to programming in C/C++ rather than any HDL, to solve this problem, algorithmic like hardware programming languages such as System-C or Handel-C are used, actually, the main contribution of the research was the guideline establishment to translate C code into Handel-C; the code synthesis and optimization regards on the software development tool. Despite the promising characteristics of this kind of algorithmic-like HDLs, we must not forget that they are translators between C and hardware, so their resulting netlists are far from optimal because they are used as a tool for software designers that need to implement their algorithms into hardware.

Through [18], [19], [20] and [21] Lawal, Thornberg and O'Nils explore Real Time Video Processing Systems (RTVPS) and their constraints in hardware implementation, as we know, in RTVPS the operations performed over a pixel are neighborhood oriented, so a large amount of data is required to be buffered (data size grows depending on the video frame and the operation window), therefore the main problem of RTVPS is memory access, FPGAs have proven to be effective implementation architectures for systems with high throughput requirements; in this case, memory structure and memory access become a priority, taking advantage or FPGA architecture, actually, the closest the memory is located, the fastest the FPGA will operate.

### 2.4.2 Hardware-Software Co-Design

As stated by De Michelli and Gupta [22] the methodology of Hardware-Software Co-design is a strategy for meeting system-level objectives by exploiting the existent synergism between hardware and software through their concurrent design. The major challenge in hardware-software co-design consists on identifying the critical segments of the software programs and compiling them efficiently to run on the programmable architecture, so, *co-design problems*

*relate to how software programs are used to configure and program a hardware circuit as well as to how the reconfigurable architecture is organized* [22].

Designing hardware-software systems involves three stages, the first one is the process of conceptualizing the specifications and the construction of a hardware and software models, the second stage looks towards achieving a reasonable level of confidence that the system will meet the model requirements, the third and last stage is simply the physical realization of the synthesized hardware and the compiled software.

As mentioned before, the main problem relies on choosing the elements of the system that will be designed as hardware or software; it is a real problem because any partitioning decision must consider the properties of the resulting blocks, if we consider general purpose processors, a partition represents a logical division of system functionality, but for FPGAs partitioning is not a simple issue, if we have systems consisting on arrays of FPGAs partitioning is equivalent to mapping; for systems consisting of processors and FPGAs, partitioning includes both a physical partition of system functionality and mapping [22].

For embedded systems, an architectural assumption is needed: since these systems are implemented by processors and application-specific hardware, they can be characterized as coprocessors. This kind of architectures is often chosen to improve system performance in executing specific algorithms, but there are applications like real time systems where speedup is not as important as the satisfaction of size and timing constraints. From the above, we can say that coprocessors are easier to design and implement using embedded processors like Xilinx's MicroBlaze [23] or Altera's Nios-II [24].

### 2.4.3 Applications

We must consider that mobile convergence, portable multimedia applications and many different devices have led us to a world were multi-codec is needed [18], also, as mobile terminals become increasingly popular, cost effectiveness and low power consumption are very important issues [25], as video processing requires special architectures its usual to find dedicated hardware engines, DSPs or FPGAs to bridge the gap, or process independent techniques are much more desirable for increasing reusability, so the evident solution must be an hybrid architecture consisting on a DSP or FPGA plus a dedicated hardware engine.

The work presented in [25] proposes a hardware efficient and low power architecture for an MPEG-4 decoder consisting in a memory block, a DSP, a Decoding Engine Unit (DEU) and an Interface Unit; the DEU consists of four different functions, including a variable length decoder, a motion compensation (MC) unit, an inverse discrete cosine transform and a post noise reduction filter. As mentioned in [16] Variable Length Encoding/Decoding cannot be implemented on a DSP because of high density control flow and large memory requirements, so both VLC and MC are considered as co-processors with local memory and independent from the DSP, the other two units are incorporated in the DSPs vector pipeline to reduce

the number of operations and power consumption.

The work presented in [26] is concerned about the trade-off between performance and power consumption for the portable electronic device market, the objective is to minimize power dissipation depending on the required throughput, it is mentioned that the fast evolution of mobile devices is forcing the designers to consider image acquisition and a better quality display in devices that were not originally meant to deal with this tasks, this fact implies that still image acquisition will require higher sensor resolution to comply the video codec standards (high performance algorithms are used to achieve the greatest compression ratio).

Power is important for us because when processing image and video a lot of memory is used and a lot of power is consumed during memory accesses, this is when FPGA architectures can be used to reduce external memory accesses and to exploit spatial and temporal redundancies to efficiently encode video.

In [27] the design flow for ASICs and FPGAs is studied, of course the most significant difference between ASIC and FPGA platform-based design is that ASIC derivative designs are fixed at design time whereas FPGAs allow the derivative design at every stage, even at run-time, this important characteristic of FPGAs was termed by Cheung et al as late integration [27]. The main advantage of late integration is that it enables an instance of a system (and the system itself) to be customized depending on the environment in which it is deployed making it adaptable to changes.

| Stage | ASIC | FPGA |
|---|---|---|
| Platform | Hardware Kernel | Hardware Kernel<br>I/O, Clocks, Test structures<br>Floor Planning |
| -<br><br>Derivative | System Design<br>Functional verification<br>Clocks, Test Structures<br>Power Distribution<br>Floor Planning<br>Block implementation<br>Assembly, I/O | Subsystem Design<br>Functional verification<br><br><br><br><br>Block implementation<br>Pre-assembly processing |
| Run-time | | Environment Analysis<br>Assembly |

Table 2.8: Tasks in Platform-Based Design

In table 2.8 the stages for both ASIC and Reconfigurable platforms design are compared; we can observe that ASIC design stages are clearly separated, the hardware kernel is designed to meet functional requirements that will be needed in the derivative stage like buses,

specialized component blocks, interface ports (for attaching the derivative designs) and test functions; in reconfigurable architectures, the platform stage covers the same tasks than ASIC design, but also includes tasks that would normally be carried out in the derivative development stage like the clock three and the floor planning (place and route) of the kernel.

The derivative design stage for ASIC development involves the selection of component modules that the system requires for full functionality, the verification of the functionality, implementation of all components and final assembly, meanwhile in the derivative design stage for FPGAs a library of subsystems is validated and implemented instead of designing an validating the entire system.

Finally, for reconfigurable platform architectures, at Run-time stage, information about the environment is gathered, systems are selected and assembled together and programming parameters are set.

Now that we know the representative stages in ASIC and FPGA design, their coincidences and differences, there is one issue left to deal with: we must choose one of these to implement a model or system, but we will make our selection based on design and production times and costs.

In figure 2.15 a comparison between FPGA and ASIC time to market analysis is shown [28]; as we can see the specification stage is exactly the same in both technologies, in the previously discussed design stage we start to notice the difference between FPGAs and ASICs, while ASIC designers take a long time in designing the platform and the derivative systems, FPGAs designers are capable of covering all three design stages and make the system integration.

The next stage in ASIC design is the creation of the first silicon prototype meanwhile in FPGA design the production stage is started just before the system integration, the main advantage of FPGAs over ASICs is that the system design can be modified even in the production stage, but ASICs are restricted to testing the silicon prototypes and have to wait for a new one if a change is needed; when the prototype is accepted the system integration stage begins and after that the production begins.

So we can say that FPGAs have the advantage of reducing design time besides reducing the production costs, and they are the obvious choice if we are looking towards fast design and implementation.

When we speak of FPGAs the most recurrent term is reconfigurability, but what exactly does it mean? According to De Michelli and Gupta, is the factor that increases the usability of a digital system, but not its performance, so if performance is the parameter of interest, we must know that for general purpose computing, top performance is achieved using superscalar RISC architectures, meanwhile for dedicated applications, ASICs achieve

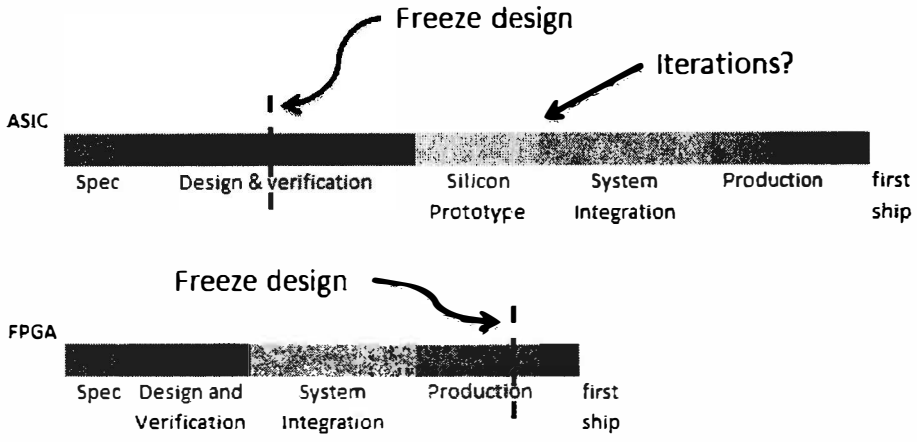



Figure 2.15: Time to market ASIC vs. FPGA

the best performance and the lowest power consumption; Another important fact to consider is that ASIC-based coprocessors can accelerate specific functions, but FPGA-based ones can be applied to speedup arbitrary software programs, especially those with parallelizable operations [22].

## 2.5 Justification

Video coding is a task that can be implemented both in hardware and software; the main problem with software implementations is that they tend to underutilize system resources as they work with long fixed wordlengths (i.e. 32 or 64 bits) and are tied to external memory read/write operations to fetch data; they also depend on an specific purpose processor capable of performing arithmetic and logic operations.

Hardware implementation of algorithms have the advantage that memory requirements are drastically reduced because the algorithm is directly translated into VLSI components that perform the same task; another interesting fact is that the algorithm can be decomposed in hardware modules that perform separate tasks, allowing the designer to parallelize operations and reduce propagation delays, making possible high throughput rates. As the algorithm is already translated into transistors no software is required to operate the system. We must remember that any algorithm can be solved in software but it will never be as efficient as a VLSI architecture solution approach.

We decided to work with reconfigurable architectures because they are suitable for modular implementation of systems, that is, we can code many different algorithms and then test and optimize each one individually prior to putting them together in a complex system, this

design methodology is possible with Hardware Description Languages; even though there are IP Cores, libraries and parametrizable modules, we chose to work with direct VHDL coding because we want to reduce the computational complexity and at the same time achieve the smallest material complexity as possible using custom wordlengths and dedicated interconnection paths; once the modules are coded in VHDL we can optimize the architectures in terms of speed, area or power consumption.

CPUs, GPUs, DSPs, ASICs and FPGAs are suitable for some specific tasks required in image and video processing like domain transformations, quantization, codification or filtering, but only FPGAs are useful to accelerate any algorithm or process as they offer a high degree of flexibility that allows system designers to test over and over again until an efficient architecture is achieved. For this reason we consider that FPGAs are the best option for real-time image and video processing tasks; their remarkable capabilities of parallelism, pipelining and dynamic reconfigurability make them especially useful when dealing with highly demanding algorithms where high throughput is required. The great advantage of working with FPGAs comes when the system prototype is completely optimized: we can choose to scale the architecture or to transfer the design to an application specific integrated circuit.

There are many applications where a dedicated video processor is required rather than a complex computational system; a co-processor could be used to enhance video surveillance applications, instead of having a CD/DVD/Blu-Ray or tape recorder we could acquire visual information, code it and transmit it over a digital channel using only an integrated circuit; a dedicated video co-processor can increase the competitive advantages of many devices of consumer electronics like car entertaining systems, we could simply replace the CD/DVD player with a hard drive, so the consumer can transfer any media to the drive, and reproduce it whenever he wants, this system can even be implemented on airplanes or buses without representing an excessive cost for the airline/line. The medicine field also requires low cost and reliable video processors as there are many tools that require image processing like Magnetic Resonance, X-rays, Ultrasound and Computarized Tomography, all these applications are critical to diagnose a large number of diseases or medical conditions, so a dedicated circuit can be used to acquire data, process it depending on the application and send it directly to a hard drive or to a network streaming device making possible to establish virtual consults between pairs that are located far away in a remote location without compromising visual quality.

We want to develop a pure VLSI architecture of a video co-processor that does not rely on a CPU or external memory devices; most of video processing tasks are computationally exhaustive, therefore it is necessary to translate the algorithms into hardware to reduce the computational complexity employing parallelism and pipelining to accelerate the processes and reduce the critical path. VLSI design will allow us to compare between implementations targeted for power consumption, die area and operating frequency; once the comparison is done we can establish a triple commitment to find the optimum architecture.

# Chapter 3

# VLSI Architectures

## 3.1 Discrete Cosine Transform

The energy of a discrete time signal x(n) can be represented as the summation of the square values of each pixel as shown in equation 3.1, we can translate the expression of energy to the frequency domain (equation 3.2); the energy distribution of the signal as a function of frequency is known as Energy Density Spectrum (EDS) it is simply the magnitude of the DTFT of $x(n)$ (equation 3.3).

$$E_{x(n)} = \sum_{n=-\infty}^{\infty} |x(n)|^2 \tag{3.1}$$

$$E_{x(n)} = \frac{1}{2\pi} \int_{-\pi}^{\pi} |X(\omega)|^2 \, d\omega \tag{3.2}$$

$$EDS = |X(\omega)|^2 \tag{3.3}$$

Using the EDS we are able to identify the frequency components that possess the lowest amount of energy and then we can discard them and still have a good representation of the original signal when we return to time domain, hence a lossy compression scheme can be used to process the image in order to reduce the amount of data that has to be stored or transmitted.

The Discrete Cosine Transform is a Fourier related transform that uses only real numbers (equation 3.4); this orthogonal transform has the characteristic of concentrating all high energy components in the lowest frequencies of the cosine [29] making the quantization and entropy coding processes easier to achieve. DCT has been widely studied since its introduction in 1974 and is probably the most common orthogonal transform in image and video processing, therefore exist many hardware architectures to compute it, a taxonomy of DCT Architectures is shown in figure 3.1 [30].

$$S(u,v) = \sqrt{\frac{2}{NM}}\, C(u)C(v) \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} x(n,m) \cos\left[\frac{(2n+1)u\pi}{2N}\right] \cos\left[\frac{(2m+1)u\pi}{2M}\right] \qquad (3.4)$$

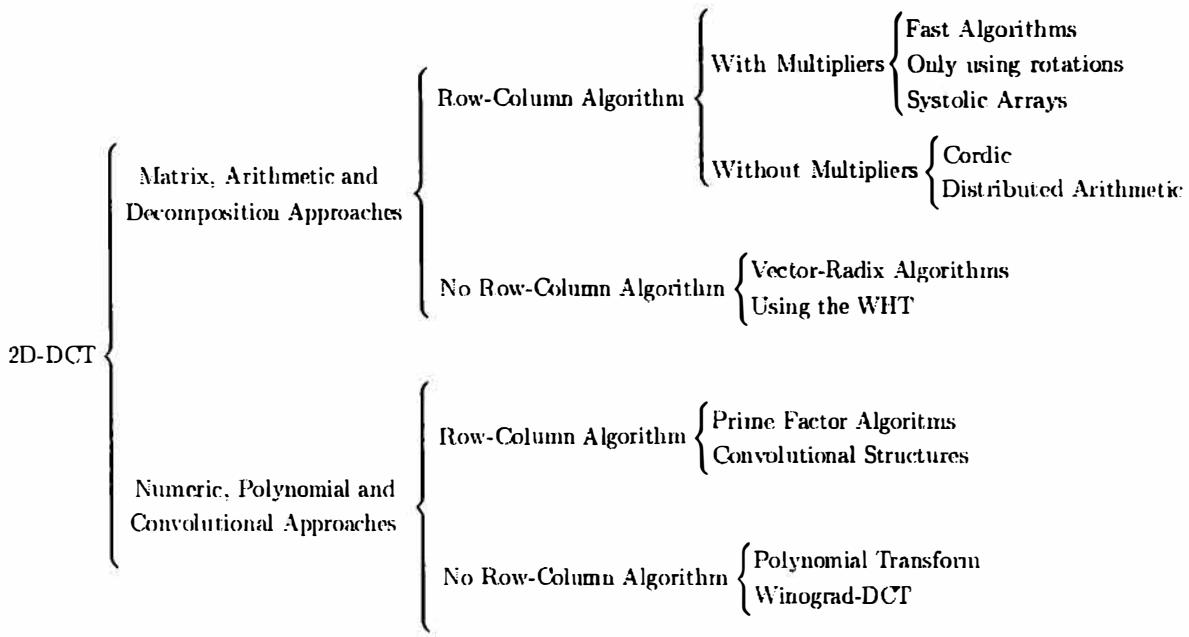



Figure 3.1: DCT Architectures Classification

The methods used to compute the DCT can be divided into two categories, the first one groups matrix analysis, matrix decomposition and arithmetic approaches, the second category groups polynomial transforms numeric approaches and convolutional structures: each category can be divided into Row-Column Algorithms (RCA) and No Row-Column Algorithms (NRCA) [30]; an overwhelming majority of implementations can be found in the RCA group of the first category, many of them are Fast Algorithms [31], [32], [33], [34], [35], [36] and the rest are Distributed Arithmetic implementations [37], [38], [39]; of course there are polynomial transform proposals like [40], [41], [42], [43], [44] but most of them are particular cases of Row-Column decomposition algorithms or direct algorithm derivations.

In general, hardware costs of any architecture are usually determined by the number of processing elements (PE), the interconection paths required to wire those PE and the amount of memory required to store data; NRCAs family requires the least number of multiplications but their interconnection structure is often complicated and the implementation is not trivial. RCA family of architectures offer easy implementations; many 1D-DCT architectures have been implemented using fast algorithms; Parallel DCT algorithms are difficult to implement because they require a dedicated communication structure and their material complexity is

too high, Systolic Array or Distributed Arithmetic implementations offer a high degree of regularity and help to reduce multiplications and additions.

The goal of any DCT architecture is to reduce the number of operations required to perform the orthogonal transform; It has been reported in [35] that direct or "brute force" implementation of the DCT-II definition requires $N^4$ multiplications and additions to compute one 8 x 8 block; One way to reduce computational complexity to $2N^3$ multiplications and additions per block is taking advantage of the separability property of the DCT (3.5); as the DCT is orthogonal, implementing a multidimensional DCT implies aplying the 1-D DCT along each dimension [45], for 2D-DCT we first compute the row transformation first and then the column transformation of the transposed DCT-1D result as shown in figure 3.2; to implement a 2-D DCT of an $M \times N$ image M $N-$ point DCTs and N $M-$ point DCTs are required as shown in figure 3.3.

$$S(u,v) = k\ C(u)C(v) \overbrace{\sum_{n=0}^{N-1} \cos\left[\frac{(2n+1)u\pi}{2N}\right] \underbrace{\left\{\sum_{m=0}^{M-1} x(n,m) \cos\left[\frac{(2m+1)v\pi}{2M}\right]\right\}}_{\text{1D DCT}}}^{\text{1D DCT}} \tag{3.5}$$

$$\text{where} k = \sqrt{\frac{2}{NM}}$$

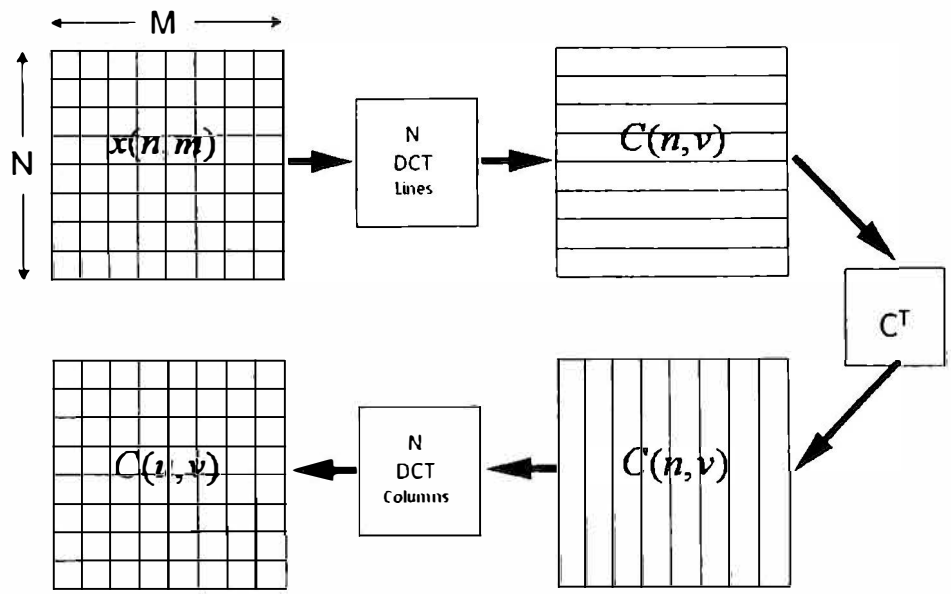



Figure 3.2: DCT-2D General Architecture

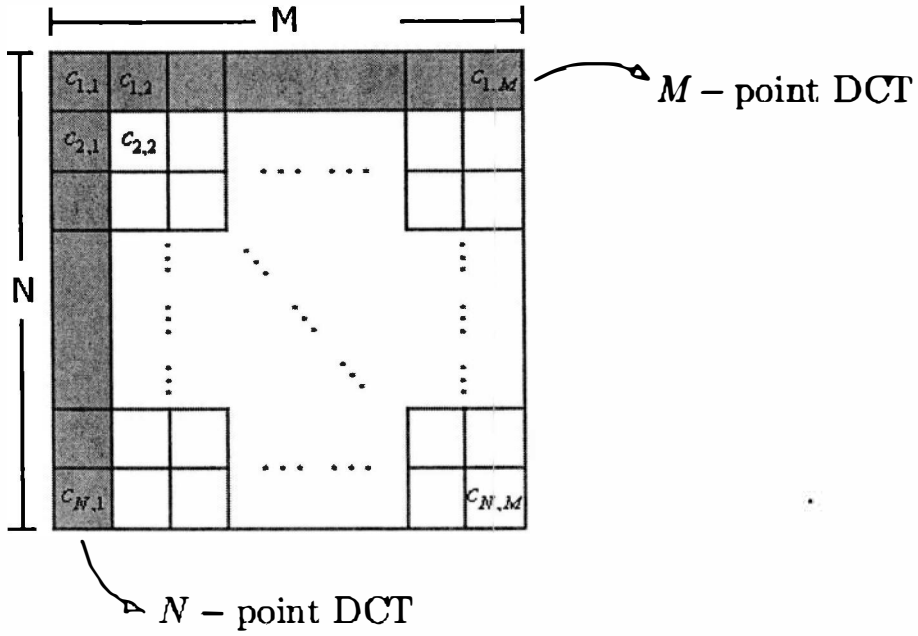



Figure 3.3: DCTs required for an $M \times N$ image

### 3.1.1 1D DCT

#### 3.1.1.1 Fast algorithms

This family of VLSI architectures is very efficient in terms of the number of multiplications and additions required to perform the orthogonal transform and represent the best option for ASIC (Application Specific Integrated Circuit) implementation even though large silicon areas might be required.

The work presented in [32] was the first algorithm to report a meaningful reduction of computational complexity of DCT, the algorithm was derived in the form of four different types of matrices that were translated into a directed graph suitable for any desired value $N = 2^n \geq 2$, this graph requires less than $\frac{1}{6}$ of the required operations of a conventional DCT algorithm using Fast Fourier Transform. Chen's Graph is shown in figure 3.4, we can notice that the algorithm has a recursive structure, therefore it is suitable for any macroblock size, the coefficients $a$, $b$, $c$, $d$, $e$, $f$ and $g$ are calculated as shown in equations (3.6).

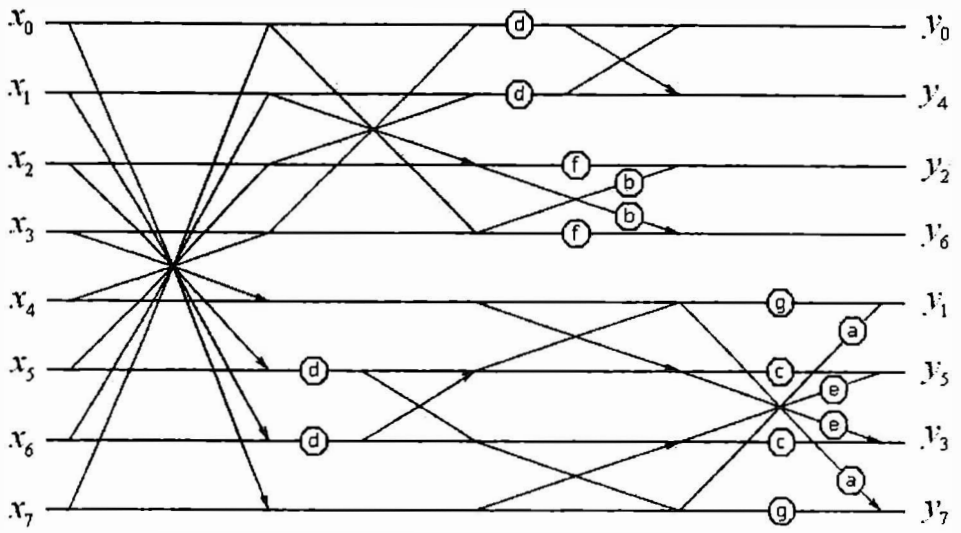



Figure 3.4: Chen's Graph

$$
\begin{aligned}
a &= \cos\frac{\pi}{16} & b &= \sin\frac{3\pi}{8} & c &= \cos\frac{3\pi}{16} \\
& & d &= \cos\frac{\pi}{4} & & \\
e &= \sin\frac{3\pi}{16} & f &= \sin\frac{\pi}{8} & g &= \cos\frac{7\pi}{16}
\end{aligned}
\tag{3.6}
$$

Rao and Kamangar present two types of algorithms for fast implementation of the 2-D DCT in [45], the first one is a recursive structure that can be extended for different block sizes, the second algorithm is nonrecursive and is tied with an specific block size; both algorithms have the same amount of additions than previous works, so the main contribution is to reduce the number of multiplications in order to reduce computational complexity. The fixed block size algorithm was proposed because when implementing a DCT is assumed that the processor will work only for a previously defined $M \times N$ blocks. Both recursive and nonrecursive algorithms work with all the coefficients at the same time, so, for a $4 \times 4$ block all 16 "boxes" are used at the same time to obtain the DCT coefficients.

In [33] a VLSI implementation of Lee's Fast Cosine Transform (FCT) [34] is presented, the FCT is similar to Fast Fourier Transform (FFT) in terms or the number of multiplications required as the $N$ point DCT is decomposed into two $\frac{N}{2}$ point DCTs; it is clear that this is not the most efficient way to implement the DCT, but it is suitable for hardware implementation; As the DCT is an orthogonoal transform, is clear that it is completely reversible, this means that we only need to invert inputs and outputs of the graph to compute the Inverse Discrete Cosine Transform (IDCT) and for that, bidirectional operators must be used. The graph is shown in figure 3.5, notice that there are three butterfly stages that involve all the

coefficients, the remaining stages consist on coefficient multiplication and reordering.

The $C_i$ factors are calculated using a simple formula:

$$C_i = \frac{1}{2\cos\frac{i\pi}{16}} \tag{3.7}$$

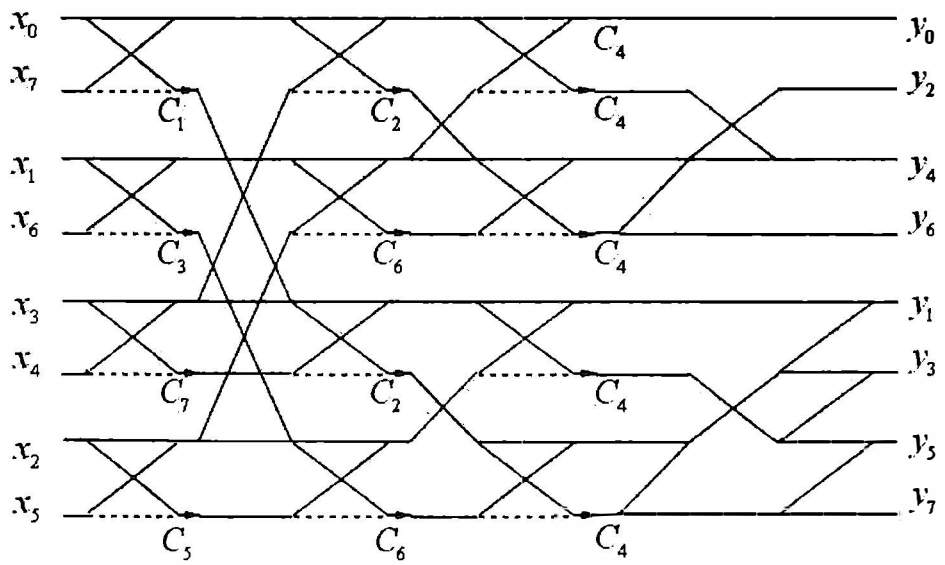



Figure 3.5: Lee's Graph

In [35] an operation efficient graph is presented, it requires 11 multiplications and 29 additions to perform 1D-DCT; the graph structure is shown in figure 3.6; this graph was designed by Loeffler et al. based on even-odd decomposition, the algorithm consists in four calculation stages. First stage is commonly known as butterfly stage and is basically an addition-substraction block, second stage is by far the most complicated stage in this algorithm, in the upper half of data lines a new butterfly needs to be performed, the remaining four data lines need to be processed as a rotation as shown in (3.8), this graph considers three rotation that are translated into nine multiplications and nine additions; $I_n$ represent the inputs of the rotator while $O_n$ represent the outputs.

$$\begin{aligned} O_0 &= I_0 \cdot k \cdot \cos\frac{n\pi}{2N} + I_1 \cdot k \cdot \sin\frac{n\pi}{2N} \\ O_1 &= -I_0 \cdot k \cdot \sin\frac{n\pi}{2N} + I_1 \cdot k \cdot \cos\frac{n\pi}{2N} \end{aligned} \tag{3.8}$$

In the third stage of the graph, a butterfly is applied to odd coefficients, meanwhile a new rotation process needs to be applied to data lines 2 and 3; In stage four the even coefficients

are left with no change and the remaining multiplications and additions are applied to odd coefficients, finally after the four stage transformation, data is rearranged and sent to the next process.

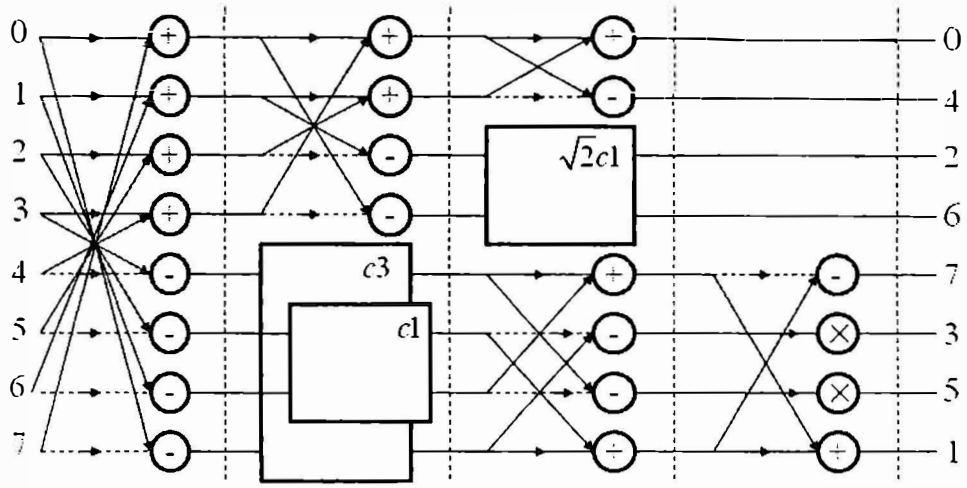



Figure 3.6: Loeffler's Graph

Many other algorithms have been proposed throughout the years, even though many of them are very efficient in terms of the number of operations required to compute the DCT, we can affirm that Fast Algorithms are not well suited for FPGA implementation because large routing and interconnection butterflies are required and this leads to extremely long propagation delays; This group of architectures also has problems with finite precision arithmetic because several rounding and truncation operations are required and this causes a serious degradation with finite precision accuracy at the same time that material complexity increases [46].

### 3.1.1.2 Polynomial Transforms

The Polynomial Transform (PT) approach to compute the Discrete Cosine Transform is based in the fact that any $N-$dimensional DCT can be splitted into a series of 1D-DCTs [43], this family of algorithms achieve considerable savings on the number of operations compared with the RCAs; many polynomial transform architectures require $\frac{1}{N}$ of the multiplications needed in RCAs, the number of additions is also reduced. Some PT algorithms are very efficient in terms of the number of operations but have a very high material complexity, some others have better computational structures and flexibility in the choice of dimensional sizes.

In [44] a 2D-DCT implementation on FPGA using Polynomial Transformation is reported; as we know, the direct implementation of the DCT definition requires $N^4$ multiplications and additions for each $8 \times 8$ macroblock; the solution proposal consists in a group of $N$ 1D-DCT

and a butterfly array to avoid memory transposition as shown in figure 3.7; one problem of this architecture is the regularity, but in exchange is a suitable option if computational speed is a prime.

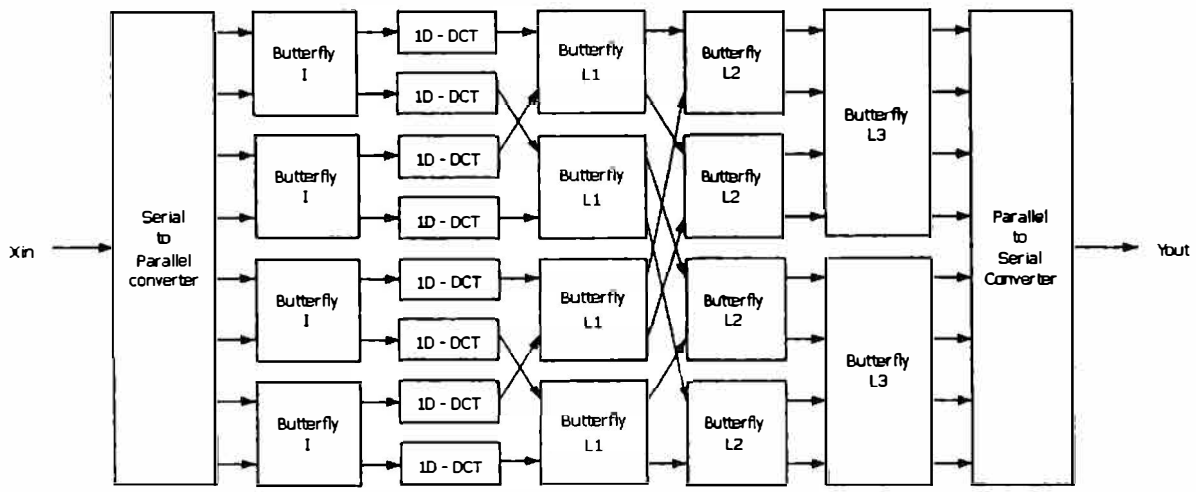



Figure 3.7: 2D-DCT by Polinomial Transform

In [47] an algorithm for splitting a $2^N$ point 1D-DCT into a set of $2^k$ point type-IV DCTs is presented, $k$ is assumed to be in $1 \leq k \leq (N-1)$. Figure 3.8 shows both decomposition methods, in figure 3.8a the first approach is shown, we can observe that an 8 point DCT-II is decomposed into one 4 point, one 2 point and one 1 point DCTs type IV; the first stage of this decomposition method rearranges input data in order to determine the multiplication and addition operands in the paired transform stage; it has been reported that the 8 point paired transform stage requires 14 additions, the 4 point DCT IV requires eight multiplications and twelve additions, the 2 point DCT-IV requires three multiplications and three additions. Therefore, this paired transform requires 12 multiplications and 29 additions.

Figure 3.8b shows the decomposition of an 8 point DCT-II into a 4 point DCT-II and a 4 point DCT-IV, the required multiplications and additions to perform this transformation are exactly the same than the required for 3.8a; both methods have large butterfly stages and then we must assume long propagation delays, the main contribution of [47] is the paired transform-based approach for splitting 1D-DCT into a number of short 1D transforms with the least number of additions and multiplications.

#### 3.1.1.3 Distributed Arithmetic

Distributed Arithmetic (DA) is an efficient method to compute inner products when one of the input vectors is fixed [48], [49], [38], the main characteristic of DA is that it uses lookup

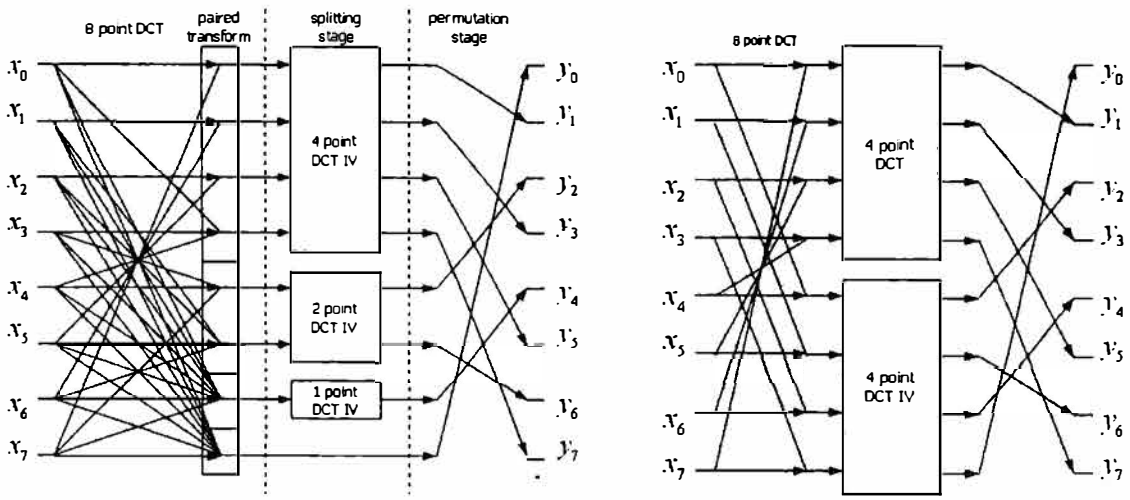



Figure 3.8: 8-point DCT II decomposition

tables (LUT) to store precomputed multiplication results that must be accumulated in a separate stage to obtain the final result, this LUT-Accumulator couple is known as ROM Accumulator (RAC); in this way the use of multipliers is avoided and as a result the material complexity of the VLSI architectures is reduced. DA-based DCT implementations may use the original DCT algorithm, the even-odd frequency decomposition or the recursive DCT algoritm to perform the orthogonal transform.

Any inner product can be represented as

$$Y = C \cdot X = \sum_{i=0}^{N-1} c_i x_i \tag{3.9}$$

where $C = \{c_0, c_1, \ldots, c_{N-1}\}$ is a fixed coefficient vector, and $X = \{x_0, x_1, \ldots, x_{N-1}\}$ is an input vector; assuming that $x_i$ is represented in $B$-bit 2's complement as shown in 3.10

$$x_i = -x_{i,B-1}2^{B-1} + \sum_{p=0}^{B-2} x_{i,p}2^p \tag{3.10}$$

then, we substitute $x_i$ in 3.9 and rearrange the expression as shown in 3.11

$$\begin{aligned} Y &= \sum_{i=0}^{N-1} c_i \left( -x_{i,B-1}2^{B-1} + \sum_{p=0}^{B-2} x_{i,p}2^p \right) \\ Y &= -\sum_{i=0}^{N-1} c_i x_{i,B-1}2^{B-1} + \sum_{p=0}^{B-2} \left( \sum_{i=0}^{N-1} c_i x_{i,p} \right) 2^p \end{aligned} \tag{3.11}$$

let

$$C_p = \sum_{i=0}^{N-1} c_i x_{i,p}$$
$$C_{B-1} = -\sum_{i=0}^{N-1} c_i x_{i,B-1} \tag{3.12}$$

Notice that when we exchange the summation order, the initial multiplications are distributed in a different calculus pattern (equation 3.13) as shown in figure 3.9.

$$Y = \sum_{p=0}^{B-1} C_p 2^p \tag{3.13}$$

The $C_j$ terms depend on the $X_{ij}$ values that have $2^N$ possible values, so it is posible to precompute the results and store them in a ROM whose address port is driven by the set $(x_{0,j}, x_{i,j}, \cdots, x_{N-1,j})$ to retrieve the corresponding $C_j$ value; All the intermediate results are accumulated in B clock cycles to produce the desired $Y$ value. This methodology leads to a multiplier-free architecture, in figure 3.10a the basic ROM Accumulator architecture is shown, notice that each retrieved value from the ROM is accumulated and shifted every clock cycle to compute the final $Y$ output. The shift accumulator unit is a ripple carry adder that adds de ROM content to the previous accumulated result, table 3.1 is an example of the ROM content when $N = 4$.

In figure 3.10b the architecture of Distributed Arithmetic with Offset Binary Coding (OBC) is shown; OBC is a reordering technique used in Distributed Arithmetic to halve the ROM size, input bits are now represented in the range $[-1, 1]$ instead of $[0, 1]$; any input data $x_i$ can be re-written as

$$x_i = \frac{1}{2}[x_i - (-x_i)] \tag{3.14}$$

The representation of $-x_i$ in 2's complement is

$$-x_i = \overline{-x_{i,B-1}} 2^{B-1} + \sum_{p=0}^{B-2} \overline{x_{i,p}} 2^p + 1 \tag{3.15}$$

then

$$x_i = \frac{1}{2}\left[-(x_{i,B-1} - \overline{x_{i,B-1}})2^{B-1} + \sum_{p=0}^{B-2}(x_{i,p} - \overline{x_{i,p}})2^p - 1\right] \tag{3.16}$$

now we define

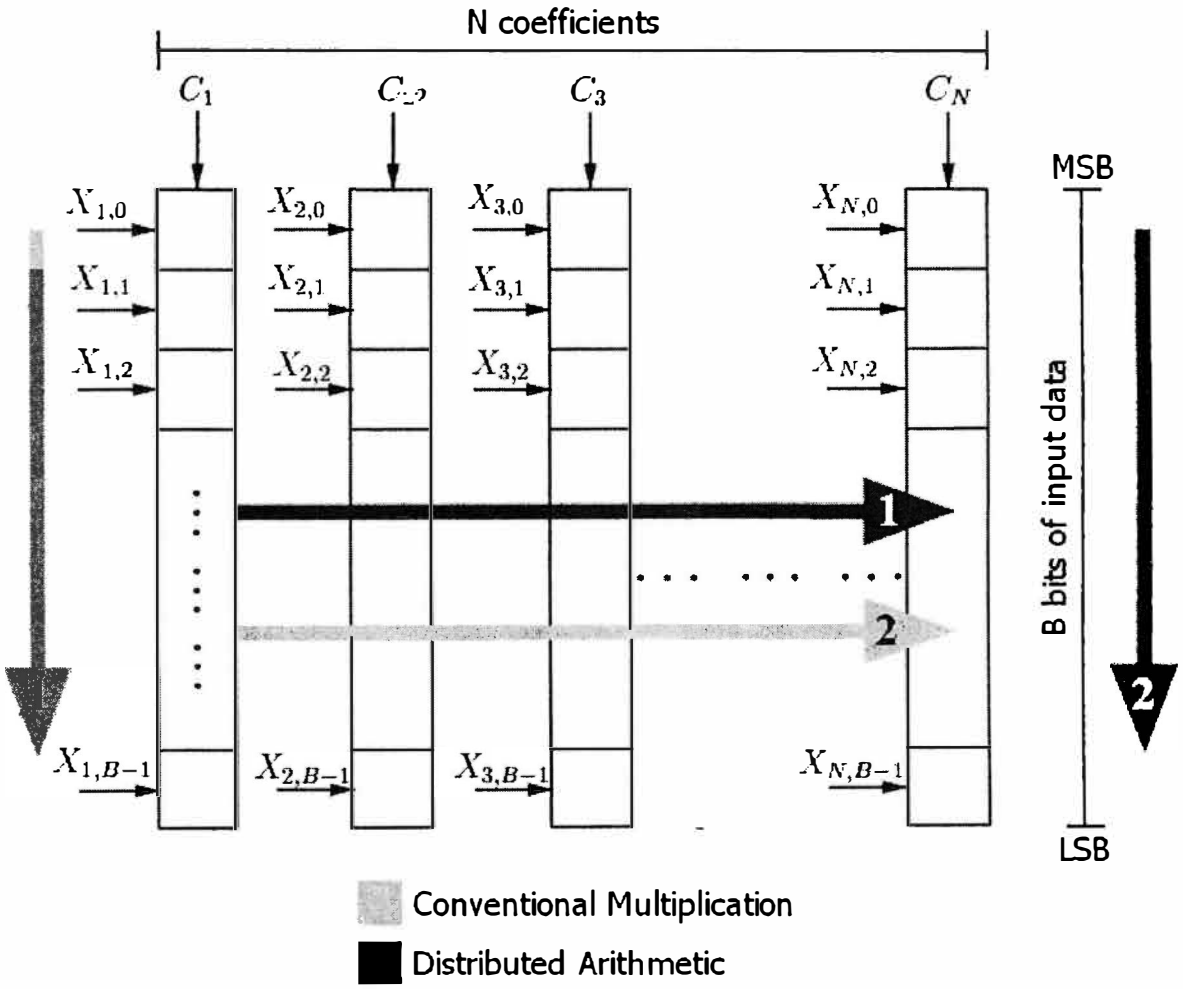
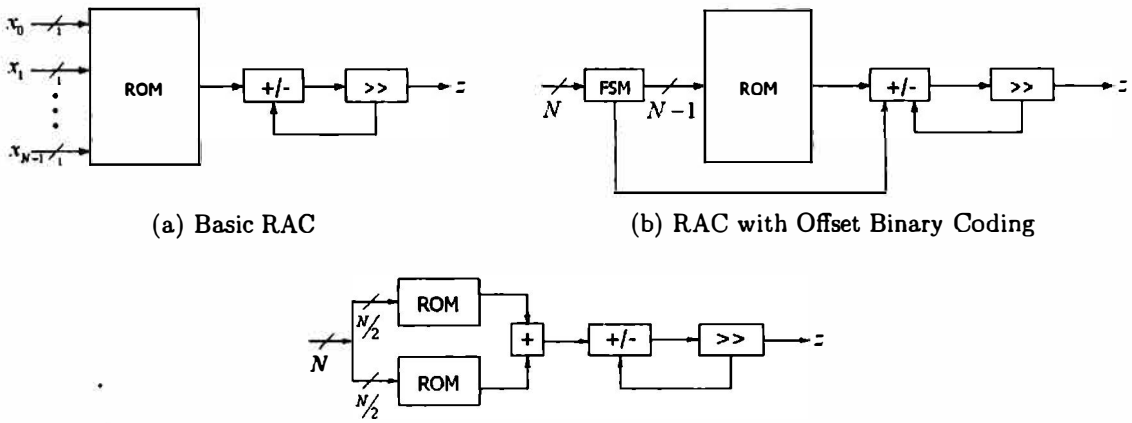



Figure 3.9: Conventional Multiplication vs Distributed Arithmetic




(a) Basic RAC

(b) RAC with Offset Binary Coding

(c) Partial Sum RAC

Figure 3.10: ROM-Accumulator Architectures

| $x_{0,j}$ | $x_{1,j}$ | $x_{2,j}$ | $x_{3,j}$ | Precomputed Result |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | $c_3$ |
| 0 | 0 | 1 | 0 | $c_2$ |
| 0 | 0 | 1 | 1 | $c_2 + c_3$ |
| 0 | 1 | 0 | 0 | $c_1$ |
| 0 | 1 | 0 | 1 | $c_1 + c_3$ |
| 0 | 1 | 1 | 0 | $c_1 + c_2$ |
| 0 | 1 | 1 | 1 | $c_1 + c_2 + c_3$ |
| 1 | 0 | 0 | 0 | $c_0$ |
| 1 | 0 | 0 | 1 | $c_0 + c_3$ |
| 1 | 0 | 1 | 0 | $c_0 + c_2$ |
| 1 | 0 | 1 | 1 | $c_0 + c_2 + c_3$ |
| 1 | 1 | 0 | 0 | $c_0 + c_1$ |
| 1 | 1 | 0 | 1 | $c_0 + c_1 + c_3$ |
| 1 | 1 | 1 | 0 | $c_0 + c_1 + c_2$ |
| 1 | 1 | 1 | 1 | $c_0 + c_1 + c_2 + c_3$ |

Table 3.1: ROM content for $N = 4$

$$d_{i,j} = \begin{cases} x_{i,j} - \overline{x_{i,j}} & \text{for } j \neq B-1; \\ -x_{i,B-1} - \overline{x_{i,B-1}} & \text{for } j = B-1; \end{cases} \tag{3.17}$$

then

$$x_i = \frac{1}{2}\left[\sum_{p=0}^{B-1} d_{i,p}2^p - 1\right] \tag{3.18}$$

$$d_{i,p} \in \{-1, +1\}$$

substituting 3.18 in 3.9

$$Y = \sum_{i=0}^{N-1} \frac{1}{2}c_i \left[\sum_{p=0}^{B-1} d_{i,p}x^p - 1\right] \tag{3.19}$$

$$Y = \sum_{p=0}^{B-1} \left[\sum_{i=0}^{N-1} \frac{1}{2}c_i d_{i,p}s^p - \frac{1}{2}\sum_{i=0}^{N-1} c_i\right] \tag{3.20}$$

now we define

$$D_p = \sum_{i=0}^{N-1} \frac{1}{2}c_i d_{i,p} \qquad \text{for } 0 \leq j \leq B-1 \tag{3.21}$$

$$D_{extra} = -\frac{1}{2}\sum_{i=0}^{N-1} c_i \tag{3.22}$$

finally, the Distributed Arithmetic with Offset Binary Coding is defined as

$$Y = \sum_{p=0}^{B-1} D_p 2^p + D_{extra} \tag{3.23}$$

Figure 3.10b shows the general architecture of DA with Offset Binary Coding; figure 3.11 shows the detailed architecture of DA with OBC, notice that additional hardware complexity must be considered for the exclusive or gate array needed to operate data decodification, one multiplexer is used to invert the ROM output when $p = B - 1$, the other multiplexer is used to provide the initial value ($D_{extra}$) to the accumulator ; additionally two control signals are required to drive the multiplexers and the XOR array, therefore a finite state machine is needed to control de RAC architecture; even this hardware cost is a low price to pay considering ROM size reduction of 50%; this allows circuit size reduction, therefore power

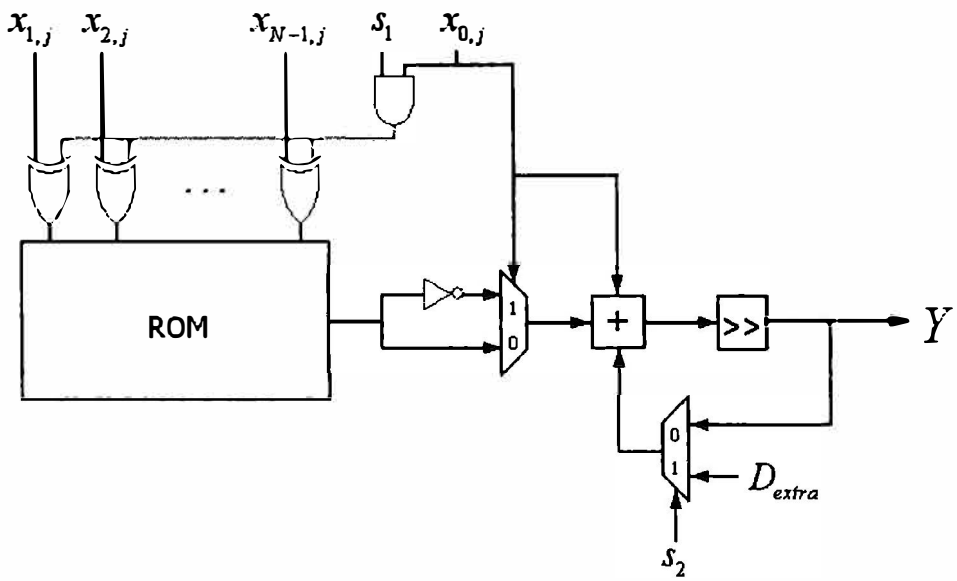



Figure 3.11: Distributed Arithmetic with Offset Binary Coding

| $x_{1,j}$ | $x_{2,j}$ | $x_{3,j}$ | Precomputed Result |
|---|---|---|---|
| 0 | 0 | 0 | $-(c_0+c_1+c_2+c_3)/2$ |
| 0 | 0 | 1 | $-(c_0+c_1+c_2-c_3)/2$ |
| 0 | 1 | 0 | $-(c_0+c_1-c_2+c_3)/2$ |
| 0 | 1 | 1 | $-(c_0+c_1-c_2-c_3)/2$ |
| 1 | 0 | 0 | $-(c_0-c_1+c_2+c_3)/2$ |
| 1 | 0 | 1 | $-(c_0-c_1+c_2-c_3)/2$ |
| 1 | 1 | 0 | $-(c_0-c_1-c_2+c_3)/2$ |
| 1 | 1 | 1 | $-(c_0-c_1-c_2-c_3)/2$ |

Table 3.2: Contents of the reduced size ROM with OBC coding for $N = 4$

consumption is reduced as the maximum operating frequency is enhanced, table 3.2 shows the content of the ROM for $N = 4$.

There are many other architectures for RAC that use some kind of ROM decomposition, the generalized architecture is shown in figure 3.10c; notice that the partial product technique is achieved by halving the ROM size and merging the partial results to form the number that will be accumulated. Area saving can be obtained either by implementing ROM decomposition or by Offset Binary Coding; both techniques offer a similar area saving since they eliminate multiplication operations at the expense of a small number of extra additions [38], every time that a memory reduction takes place we must consider pipelining as an strategy to reduce the critical path for the operation to be performed; ROM reduction can be studied as a recursive process that eventually will lead us to a LUT-less architecture where no memory is required, the hardware cost of a memoryless architecture is a high number of multiplexers and full adders and a high number of pipeline barriers.

If pixels are coded in 8 bits, pure distributed arithmetic DCT implementation requires $8 \times 2^8 \times$ ROM wordsize Kbits of memory, this architecture is shown in figure 3.12; as we stated previously, OBC technique is used to reduce memory requirements. Figure 3.13 shows the architecture of the DCT using OBC, notice that before entering RAC modules, there is a butterfly stage that computes the even-odd decomposition of coefficients; memory requirement for this architecture is calculated as $8 \times 2^4 \times$ ROM wordsize; Assuming a 16-bit word then the memory required for conventional DA and DA with OBC are 32 Kbit and 2 Kbit respectively.

### 3.1.2 Memory Transposition

It is clear that an 8-point 1D-DCT transforms 64 coefficients, after the first transformation we must transpose these transformed coefficients and then feed them to the second 8-point 1D-DCT as shown in figure 3.14; we require 6 bits to address 64 words. Let the vectors $\{000000, 000001, \cdots, 000111\}$ and $\{111000, 111001, \cdots, 111111\}$ be the first and last rows of the $8 \times 8$ coefficient matrix, notice that the most significant bits (MSB) identify the row and the least significant bits (LSB) identify the column, hence, to transpose the coefficient matrix we simply need to exchange the most significant bits with the least significant ones.

Hardware implementation of the memory transposition process can be easily achieved using a single double-port RAM (DPRAM); address generation can be implemented with a six bit counter and a multiplexer array to swap the MSB and LSB blocks as shown in figure 3.15 [50].

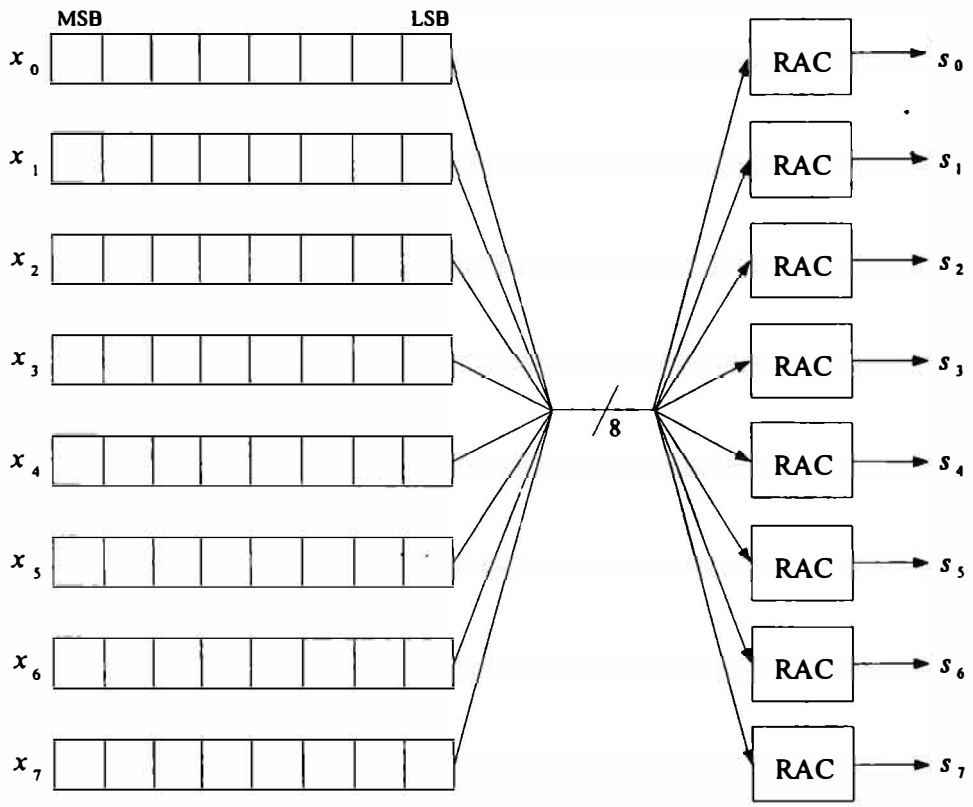



Figure 3.12: Pure Distributed Arithmetic Implementation of DCT

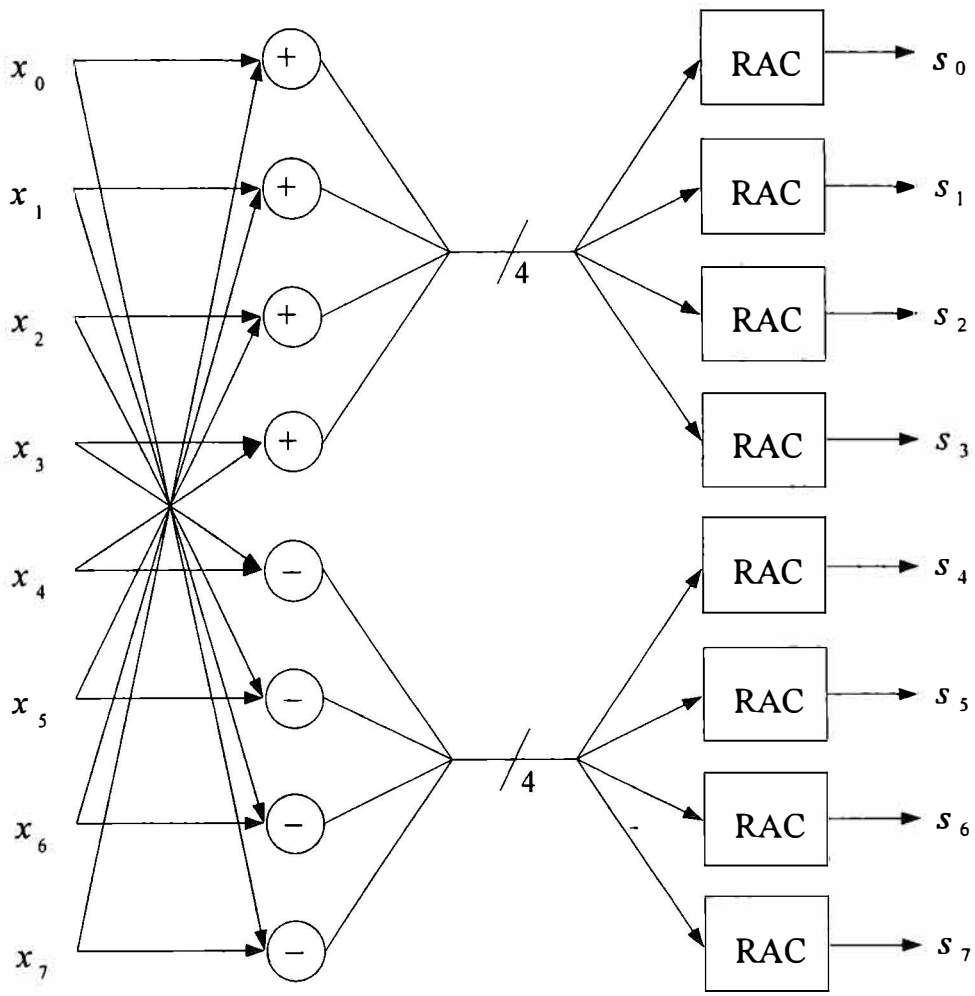



Figure 3.13: Distributed Arithmetic with OBC Implementation of DCT

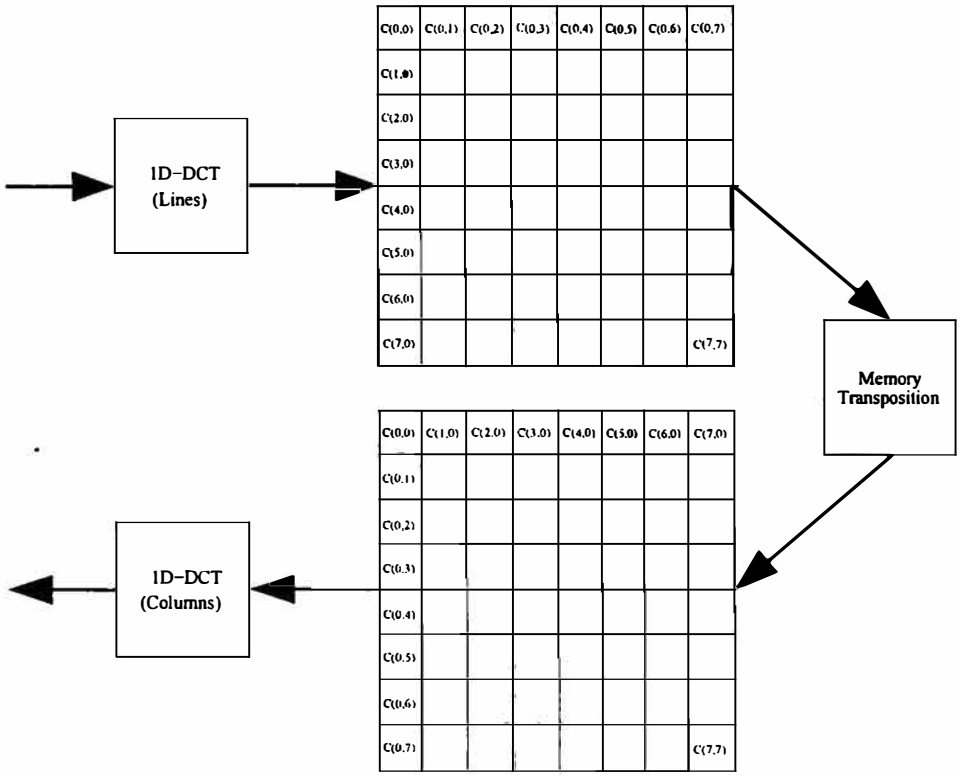



Figure 3.14: Memory Transposition for DCT

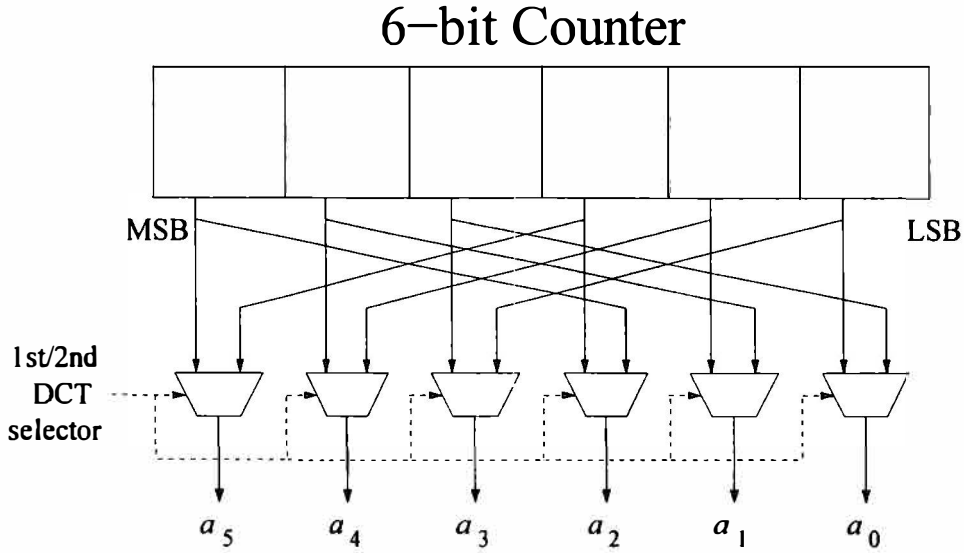



Figure 3.15: Memory Transposing Architecture

## 3.2 Entropy Coding

Entropy is a topic widely studied in [2], [4] and [3], let us consider a source $S$ capable of generating random symbos $s_1, s_2, \cdots, s_N$; digital images and video can be considered as sources, in this case $s_i$ represents one of $N$ possible quantized values for a determined pixel, every symbol has a probability of occurrence denoted as $p_i$, and the information within a symbol or self information is measured as shown in equation 3.24.

$$I(s_i) = \log \frac{1}{p_i} \tag{3.24}$$

The entropy of a source can be understood as the average information of the source $S$ an is defined as

$$H(S) = \sum_i p_i \log_2 \frac{1}{p_i} \tag{3.25}$$

From equation 3.25 we can observe that if the symbols are distinct the average number of bits to encode is bounded by the entropy, it is also clear that we require a probability model to calculate the entropy of the source, so in practice the entropy is unknown until we choose a distribution to fit data.

As system designers we must estimate the probability density function of the data to encode and then map all the symbols into a dictionary or table that contains the codewords for each symbol, in fact, this process is widely known as entropy coding, the main idea of this process is to assign short codewords to high occurring symbols and long codewords to symbols that have low probability of occurrence; figure 3.16 shows the generic model for entropy coding, to perform the entropy decoding we simply need to reverse the flow of operations.

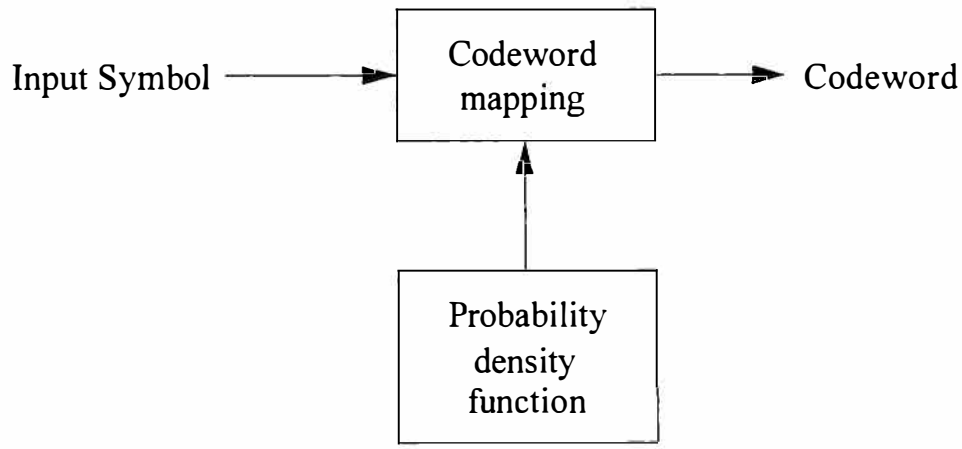



Figure 3.16: Generic entropy coder

Entropy coding is the last process in the majority of video standards like H.261, H.263 and MPEG; the main difference between this compression process and the DCT is that entropy coding is lossless while the DCT is lossy. The entropy coder has two data compression-decompression units:

1. Run-Length Coder: Compresses an input stream representing strings of consecutive zeros by their run-length, when a zero is detected the RLC simply counts the number of consecutive zeros until the last one is reached or when the counter exceeds the maximum zero run-length. The inverse process is done by generating the appropriate number of zeros between nonzero data; this coder can be implemented with an ascending counter, some registers and logic gates.

2. Variable-Length Coder: Maps the input data into codewords of variable length according to their probability; this process is performed with the expectation that the average size of a codeword will be close to the entropy of the data source.

Entropy coding can be done using lookup tables based on the statistics of the input source [2], but decoding is a complex process because codewords have variable lengths and the receiver does not know where a codeword begins and where it ends; for this case, Huffman coding has two interesting properties:

1. The is only one codeword per symbol, this means that it is impossible to have the same arrangement of coding digits for two different messages.

2. No codeword is a prefix of another code, then, each source symbol is a unique leaf of a decoding tree.

### 3.2.1 Huffman Coding

Huffman Code was developed in 1952 [51] to reduce the amount of bits required to transmit a message, since then it has become the most widely used variable length coder and has been included in almost every image and video compression standard; Huffman Code offers high compression ratios and has the advantage of being lossless [52, 53]. Huffman coding requires a code word table (also known as dictionary) that contains the mapping information between read data and coded words; frequent data is encoded using short length codes, as the data frequency decreases, the symbol length required to represent data increases. To build the table a complete scanning of the image to be compressed is required in order to obtain the frequency components that must be ordered from highest to lowest to propperly assign the symbol.

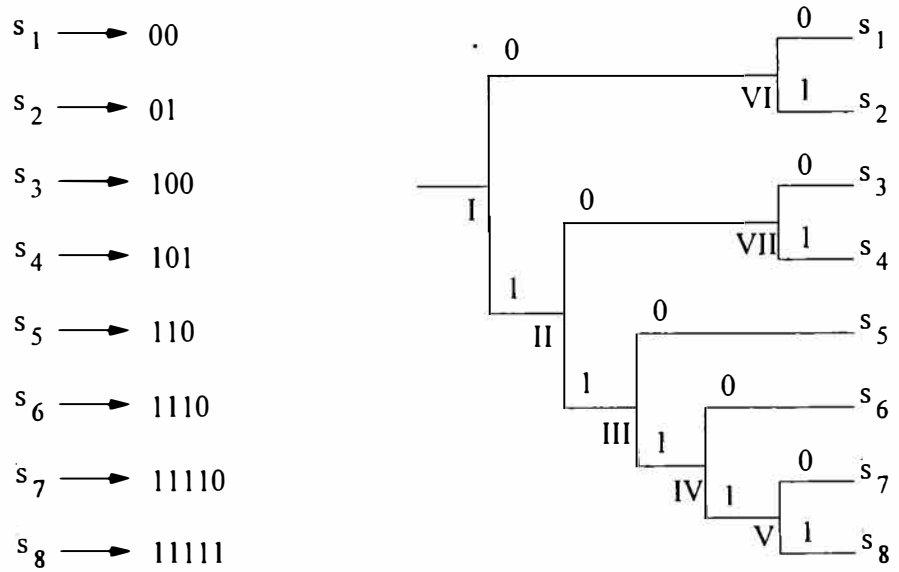
A simple iterative process must be followed to build the huffman tree that is required to construct the Huffman code:

1. Sort the symbols according to their probabilities

2. Merge the two symbols with smaller probabilities

3. Repeat step two until a set of one element is achieved

For $n$ symbols, a Huffman tree has $n-1$ nodes, so the decoding process can be performed by tracing the decoding tree until a leaf is reached; figure 3.17a is an example of a lookup table that maps the incoming data into a unique codeword, figure 3.17b shows the binary tree required to decode the information. The output of the VLC when the input sequence is $s_3 s_6 s_5$ will be 1001110110 as we simply substitute the input symbol by the corresponding codeword; in the decoder, if we have the following stream 11111011001011110 the decoded output will be $s_8 s_2 s_3 s_4 s_6$.




(a) Encoding LUT

(b) Binary Tree

Figure 3.17: Huffman Coding

In general the average codeword length is defined as shown in equation 3.26, where $l_i$ is the length expressed in bits of the corresponding codeword of symbol $s_i$; the average codeword length is then a measurement of the compression ratio.

$$l_{Avg} = \sum l_i pi \tag{3.26}$$

There are two known methods for eliminating pre-scan process in real time applications: the first method is the static Huffman Coding that consists in a default code word table, the main disadvantages of this method are

1. The compression ratio is not efficient as the dictionary is built using frequency values that might not correspond to the image that must be compressed and

2. The image must be scanned twice to compress real data, making real time applications difficult to implement.

The second strategy is using Adaptive Huffman Codes, this method requires an adaptively constructed table that must be maintained both at the sender and the receiver side, as the frequency distribution of symbols changes, frequent symbols could be mapped onto long codewords causing a degradation in compression efficiency [52]; encoding speed is slow because many search and replace operations must be performed along table swapping. Nevertheless,the greatest disadvantage of this method is that material complexity is too high, it requires vast FPGA resources to address all the possible hardware configurations that could be used while compressing an image.

The VLC encoder proposed in [54] is based on programmable logic arrays to store Huffman tables, instead of PLAs we can use ROM, RAM or even CAM devices, figure 3.18 shows the VLC encoder, notice that there are two registers used to buffer the output data, the length of each codeword is calculated in a four bit adder that controls a barrel shifter that places the output from the memory device next to the current bit stream, coded symbols are then packed in a 16-bit string, so, when the sum of the codelengths is greater than 15 a carry out signal is generated in the adder, this signal triggers the output of the upper register, then the process continues in the same way until no further data has to be coded. Table 3.3 shows the process described before, for this example we use the encoding table 3.17a, the codeword assigned to the input symbol is underlined to facilitate comprehension.

| Input | Upper Register | Lower Register | Accum | Control Signal |
|---|---|---|---|---|
| $s_7$ | 1111000000000000 | 0000000000000000 | 5 | 0 |
| $s_5$ | 1111011000000000 | 0000000000000000 | 8 | 0 |
| $s_2$ | 1111011001000000 | 0000000000000000 | 10 | 0 |
| $s_3$ | 1111011001100000 | 0000000000000000 | 13 | 0 |
| $s_8$ | 1111011001100111 | 1100000000000000 | 2 | 1 |
| $s_4$ | 1110100000000000 | 0000000000000000 | 5 | 0 |

Table 3.3: VLC encoder operation

The VLC decoding process is more complex than encoding because there is no way to identify whether a symbol ends and the next starts in the incoming data stream; in [2] a classification of VLC decoders is presented, in general there are constant input-rate (CIR) decoders and constant output-rate decoders (COR), the difference between both families is that CIR decoders have a variable output rate and the COR decoders have a variable input rate.

In the simplest configuration, CIR decoders process input data serially; begining from the root it follows the branches of the decoding tree until a terminal node is reached, when

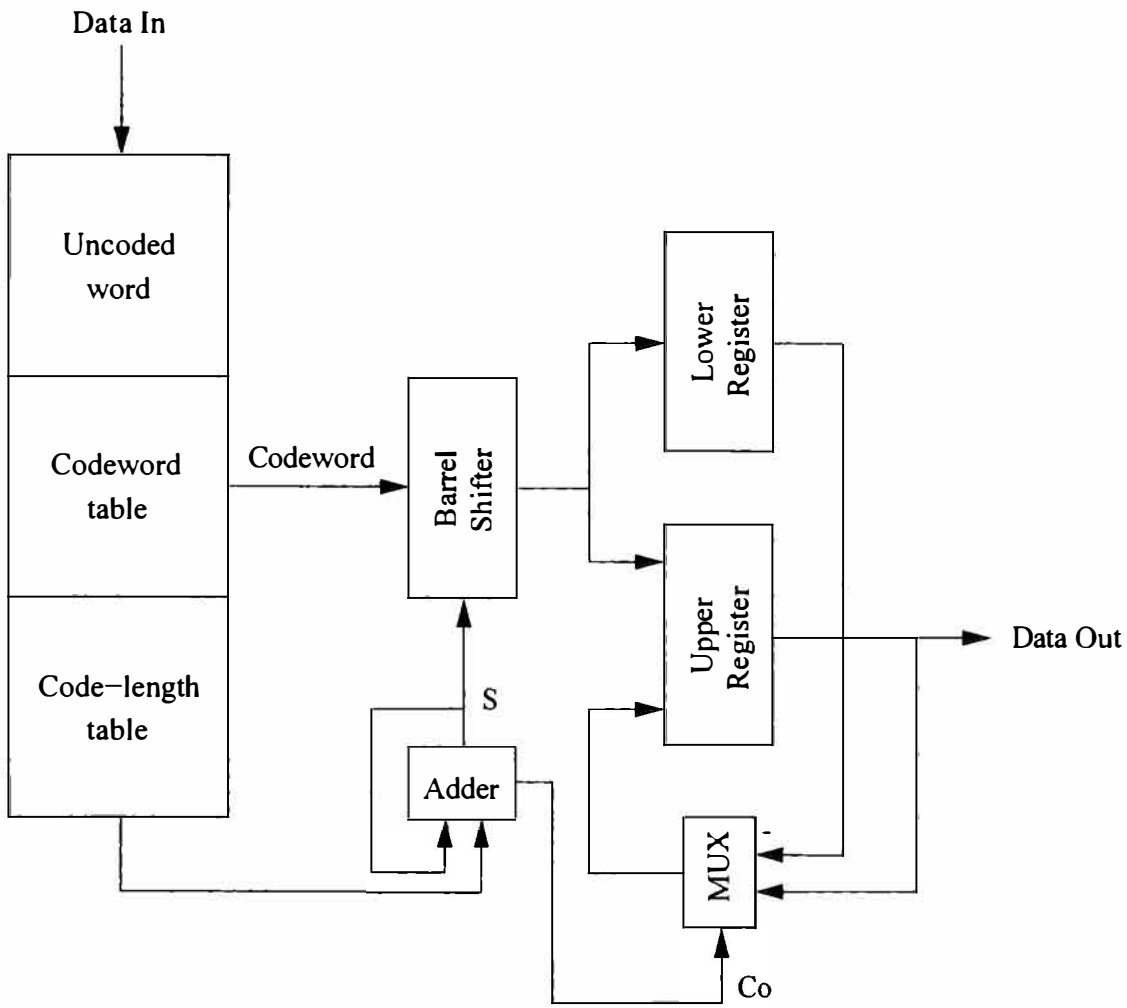



Figure 3.18: VLC Encoder

the codeword is completely decoded, the process restarts to continue with next codeword decodification; in fact, this process can be modeled as a finite-state machine (FSM) where each node is a state of the FSM, a simple control signal can indicate if the state is the end of a sequence or not, so, if we are to implement this kind of decoder we must consider $N-1$ states, where $N$ is the total number of nodes of the decoding tree. Figure 3.19 shows the proposed architecture in [2] for a Huffman decoder using a ROM based FSM, notice that a control signal is required to clear the address register once a codeword has been decoded.

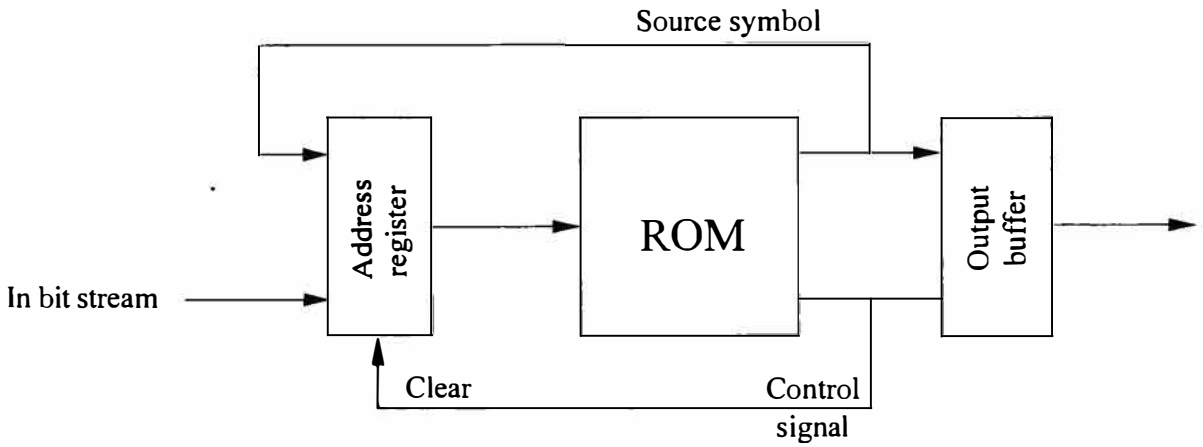



Figure 3.19: Memory based Huffman Decoder

Under this decoding scheme the average time to decode any given codeword represented on a tree with $n$ symbols is $log_2 n$ cycles, and the throughput is determined by the reading cycles of the memory device; performance can be upgraded if multiple bits are traced at the same time over the decoding tree, and the average decoding time is affected by the number of simultaneous bits that are processed in the following way $\frac{\log_2 n}{\text{concurrent bits}}$.

COR decoders are simpler that CIR, but their main disadvantage is that they do not provide a fixed decoding rate; moreover, if we want to process multiple bits concurrently, the Huffman tree has to be reconfigured, figure 3.20 shows the architecture for VLC decoder proposed in [54].

Just as the VLC encoder, the decoder utilizes a set of VLC tables, a barrel shifter, two registers and one 4-bit adder, the upper register is 16-bit long as the maximum codelength is assumed to be also 16-bit so that one codeword can be decoded per cycle; we must remember that a codeword can be splitted into two adjacent input streams, so, this decoder operates in two segments at the same time, in this architecture, the barrel shifter is a window that slides across the contents of both lower and upper registers, the shift operation is controlled by the adder that simply accumulates the length of the decoded codeword.

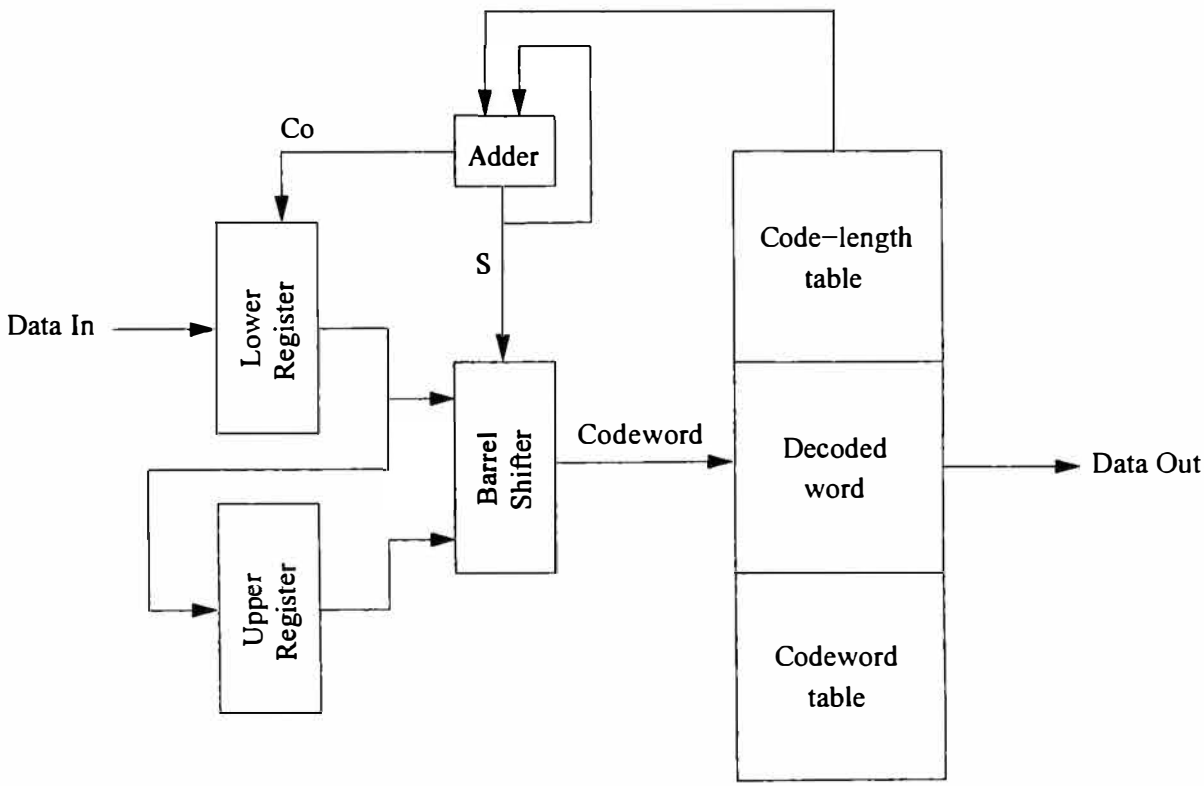



Figure 3.20: VLC decoder

Every cycle, the output of the barrel shifter is compared with al the entries in the memory device, when a match is found the memory outputs the corresponding source symbol and the length of the decoded codeword is shifted both to the beginning of the next codeword and in the barrel shifter; if the adder result is greater than 15, then it resets, the carry out serves as a control signal that indicates whether the upper register has been fully decoded or not. When the logic value of the carry out is '1' the lower register is transferred to the upper one and a new 16-bit segment is stored in the lower register, this process continues until there is no further codeword to process. Table 3.4 shows this process, the output of the barrel shifter is underlined; notice that it is always 16 bit long, when the carry out of the adder changes from '0' to '1' the contents of the lower register are moved to the upper register and a new stream to decode is stored in the upper register.

| Upper Register | Lower Register | Accum | Control Signal | Output |
|---|---|---|---|---|
| 1111011001100111 | 1010101001011110 | 5 | 0 | $s_7$ |
| 1111011001100111 | 1010101001011110 | 8 | 0 | $s_5$ |
| 1111011001100111 | 1010101001011110 | 10 | 0 | $s_2$ |
| 1111011001100111 | 1010101001011110 | 13 | 0 | $s_3$ |
| 1111011001100111 | 1010101001011110 | 2 | 1 | $s_8$ |
| 1010101001011110 | 0010111100001101 | 5 | 0 | $s_4$ |

Table 3.4: VLC decoder operation

#### 3.2.1.1 Known Implementations

In [55] the necessity of a high-speed implementation of Huffman decoders is presented; two architecture families are studied to solve this problem, tree-based architectures and Programmable Logic Array architectures. The first approach consists of a hardwired tree-traversing where the branching function of each node is performed by a 1-to-2 demultiplexer and a register to store the source symbols associated with every terminal node, the architecture is regular and dedicated but the material complexity is too high for large codebooks, figure 3.21 shows the architecture of a tree-based huffman decoder; pipelining can be introduced to reduce the critical path by partitioning the entire decoder into $n$ pipeline stages, each one containing one level of the binary tree.

The architecture can be reduced by combining the demultiplexer-register pairs in a single control/storage unit, in this case, the decoder can be implemented cascading some read only memories; as expected, multiple bit streams can be processed in parallel when the pipelined architecture is employed; If the stream is of infinite length, the throughput is always one code per clock cycle, so the critical path is very short, hence it is suitable for high clock rates, this characteristic is independent of the alphabet size and the maximum codeword length; as stated before, the main disadvantage of the architecture is that material complexity can

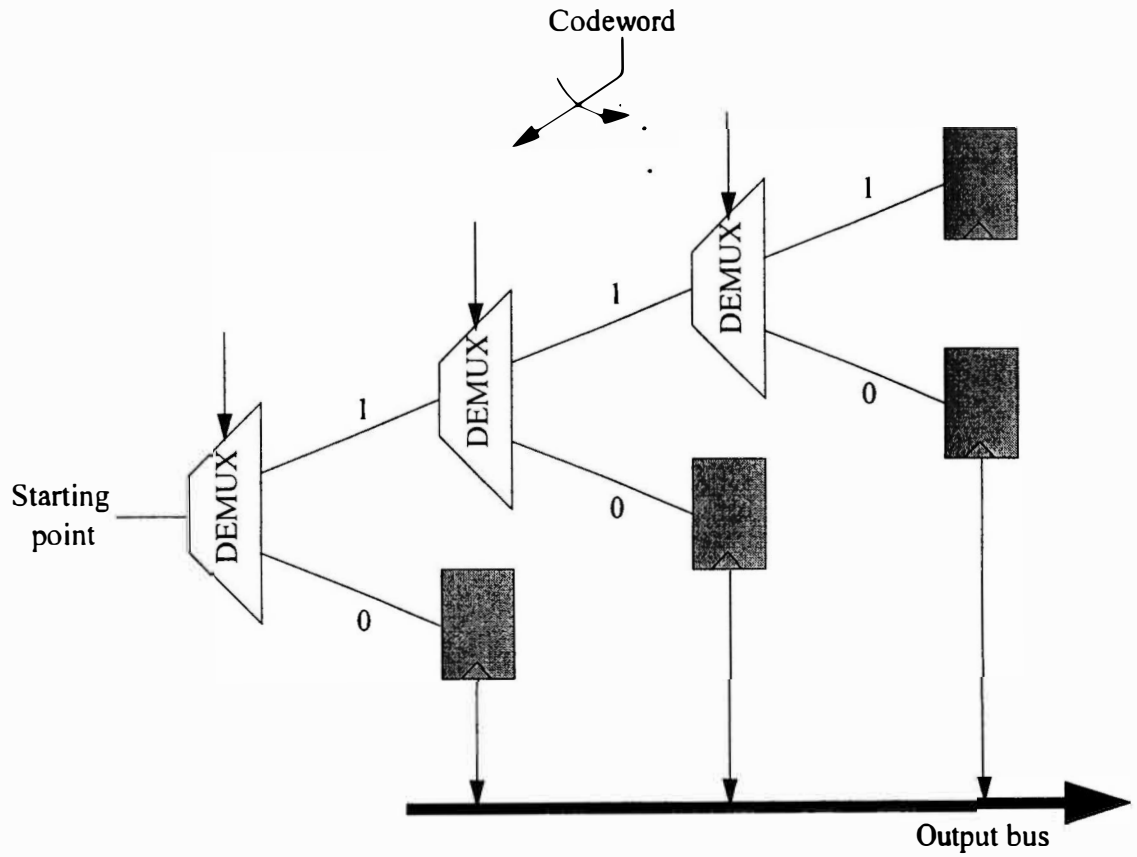



Figure 3.21: Tree-Based architecture

be very high. If the length of the stream is finite and constant then the pipeline speedup decreases as some blocks may run out of data while others still have information to decode.

The PLA-based architectures proposed in Chang's work are constant-input rate and constant-output rate configurations, the contribution in this field is a variable-I/O-rate architecture; this configuration is an extension of a constant-output rate architecture to decode $N$ codewords per clock cycle whenever is possible; they use a PLA as a substitute of ROM devices employed in [54], so the FSM required to operate the architecture is easily coded with hardware description languages (HDLs), figure 3.22 shows the variable input-output rate architecture for a Huffman decoder.

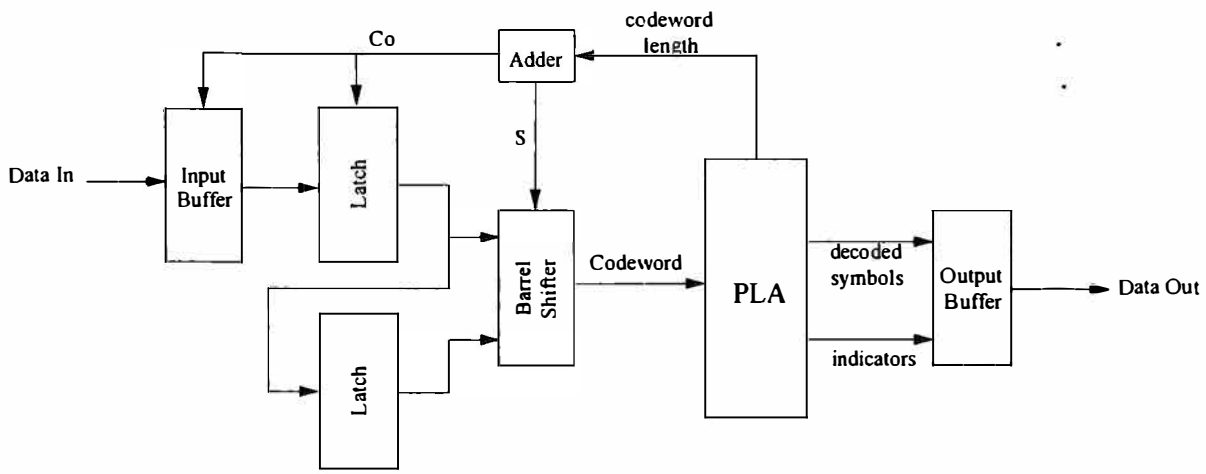



Figure 3.22: Variable I/O-rate architecture

An asynchronous VLSI architecture for Huffman codecs is presented in [56], this architecture is a hardwired Huffman coding and decoding tree using multiplexers and registers, this work claims that is a compact and fast implementation when the symbol dictionary is already known, when the symbol distribution changes the tree is updated using asynchronous logic. The main problem of this approach is that it cannot take advantage of the fast generation of short Huffman codewords, so additional circuitry is required to implement dummy nodes and counters making this architecture unfeasible for large codeword tables.

In [53] a simple scheme for mapping Huffman trees into memory devices is presented; a discussion on optimal Huffman code trees is taken into account, this kind of binary trees can always be represented as *full binary trees*, it does not matter if the tree grows to the left or to the right as long as every node has exactly two children nodes; any binary tree can be transformed into a canonical tree without increasing the average code length [57].

Figure 3.23 shows the proposed architecture, memory device $M_1$ stores a pointer of the symbols and their boundaries, every location of $M_2$ stores a group of four pointers and bound-

aries; both memory devices shall be initialized before the coding or decoding process begins. The Arithmetic-Logic Unit consists of two 8-bit adders and one comparator.

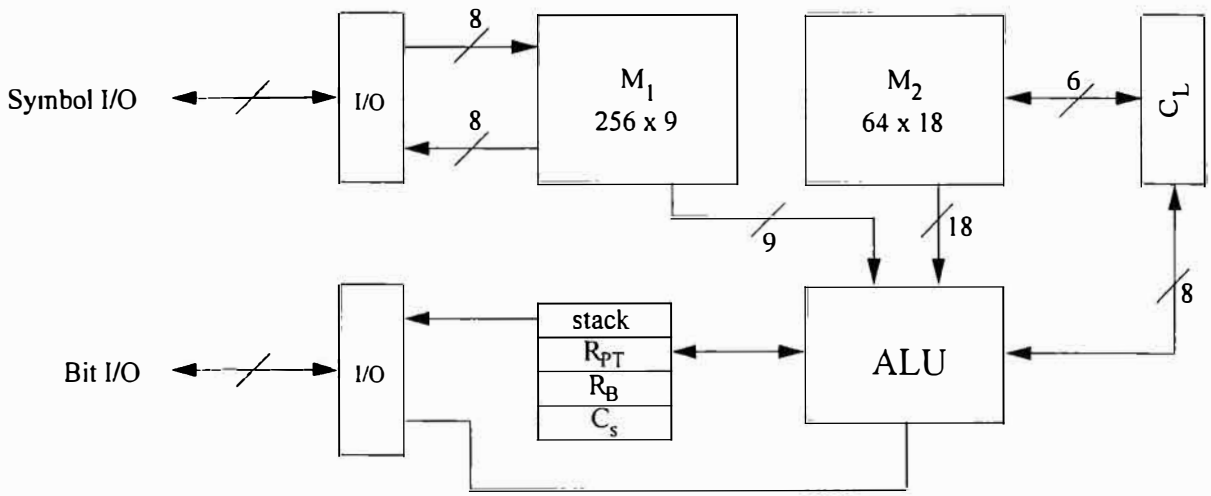



Figure 3.23: Park's Codec Architecture

The stack is used to store the intermediate encoded outputs and to reverse data at the end of each encoding cycle; registers $R_{PT}$ and $R_B$ are used for tree traversing in the decoding process, two flip-flops can be used to implement this registers, finally, there are two counters, $C_s$ is used to accumulate the length of the coded symbol and to store the number of symbols up to the previous traversing level, it is also responsible to send the control signal required to output the symbol while decoding, $C_L$ is used to access memory device $M_2$; notice that bidirectional I/O ports are used in this implementation.

In [58] a Content-Addressable Memory (CAM) based architecture for dynamic Huffman Coding is presented, we already know the difference between-static and adaptive Huffman coding schemes and we must remember that adaptive scheme reaches a better compression result if the symbol distribution of incoming data is uncertain, if the distribution is known, the best option is the static coding scheme, according to [59] dynamic Huffman encoding consists of two procedures: tree tuning and code generation. In [52, 60, 61] other CAM based architectures for adaptive coding are presented; three problems with adaptive coding are identified and discused:

1. Low compression during start-up phase.
2. Frequent symbols will be mapped onto long codewords when the frequency distribution changes.
3. Encoding process is slow, many search and replace operations must be performed when the table is swapped.

The proposed architecture for Huffman coding is presented in figure 3.24; notice that it has three main blocks. The encoder block is composed of a CAM for the symbol table and a RAM for the codeword table, everytime the CAM receives a symbol it is compared and an address is sent to the RAM that outputs the corresponding codeword; notice that two codeword tables are utilized, the active table generates the codewords while the shadow table prepares an optimized version of the active table, when the compression ratio becomes lower than a certain threshold the tables are swapped.

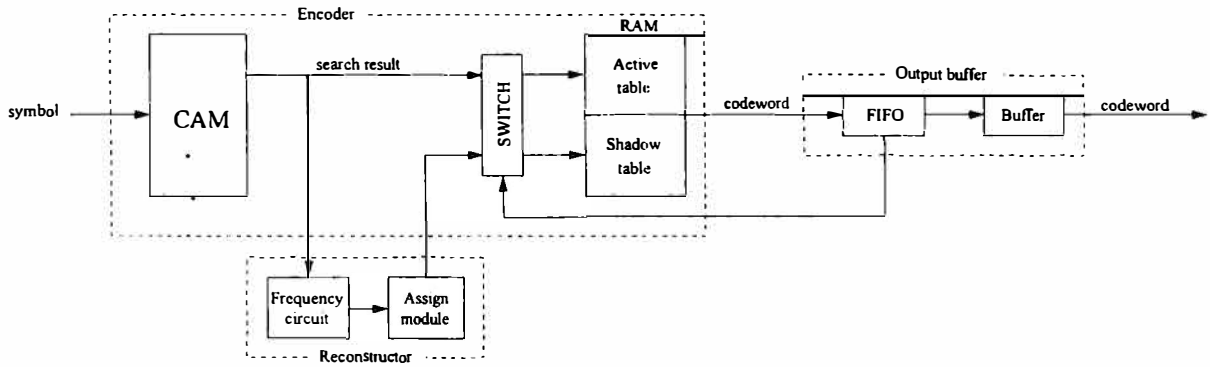



Figure 3.24: CAM based Architecture

The reconstructor block builds an optimized version of the shadow table, everytime the CAM finds a match, the corresponding address is also sent to this block where a match counter for that specific symbol is increased. If the symbol count value is beyond a certain threshold level then the assign module generates an optimized Huffman code according to the new frequency distribution of input symbols; as the output of the encoder block has variable length a FIFO must be placed after the codeword table to manage the variations in data amount, when the capacity of the FIFO is almost exhausted we must assume that the compression ratio has become to small, therefore it will be necessary to switch to the shadow table. This architecture requires a counter for every symbol and a set of control signals that must be propagated to the entire array of counters, hence the material complexity of this solution is too high, even though high operating frequencies are reported there is not a single report table that describes how many Logic Elements or how many LUTs are required to implement this architecture.

In [60] a small change is made over the architecture shown in 3.24, instead of a single port CAM and an output buffer, a multi-port CAM is used and the output codeword is controlled by a series of switches connected to both the active and shadow table; in [61] the self optimizing feature for Huffman encoding is presented; this architecture consists of two blocks, the encoding block and the reconstructing block. in the former one CAM and two RAMs are used, when a symbol arrives and is compared in parallel, the CAM outputs at least one match from the symbol LUT and exactly one match is output from an input pattern table; then all matched symbols are encoded with the same codeword in parallel and a match-flag enables

all the corresponding register. If a match flag has already a valid condition, the encoding block ignores the stored symbol and continues to the next address to select the next input symbol. The reconstructing block of keeps a high compression ratio for Huffman coding, the frequency distribution is calculated by using multiple-CAM matches, if it is required the new codeword is assigned in the shadowtable according to the recent distribution of input symbols, when the compression ratio decreases the active table is swapped with the shadow table, so that the coding efficiency is always high.

A pipelined parallel Huffman decoder is presented in [62], this is a constant output-rate decoder, the main challenge with the proposed architecture is that the symbol decoder and the length decoder operate in parallel on the same code, therefore the code length is not available when the decoding process begins, making the process too complicated; to solve this drawback a buffer can be used in front of the symbol decoder so that the code length is known as the decoding process starts. Figure 3.25 shows the proposed architecture, the codelength is evaluated in the pipelined length decoder; every clock cycle one codelength is compared until a match is found, when this happens, the code has been shifted out from the shift register and stored in a register that feeds the symbol decoder. When symbol decodification process begins the length decoder starts to examine the next code and the process iterates until there is no symbol to decode. The major disadvantage of this architecture is that symbol decoder is designed for a worst case clock rate of $f_{s,max} = f_{clk}$ in order to be able to handle succeding one-bit codes; this causes an skew effect as the clock rate in the symbol decoder is lower when longer codewords are decoded.

In [63] an implementation of a Huffman Coder is presented, figure 3.26 shows the proposed architecture, notice that input data must be available in a memory device consisting on twelve 32-bit length locations, then data is transferred to an occurrence calculator consisting on simple adders and then to the encoding block where a codeword is assigned; the encoding block has two inner blocks, an adder that calculates the sum of values that pass through it and a sorter that arranges incoming data in descending order to assign propper codewords. Finally the decoder block receives incoming codewords and searches for the associated symbol.

## 3.3 Sorting Algorithms

In computer science terminology sorting is defined as the process of rearranging a sequence of values either in ascending or descending order [64], sorting algorithms have both practical importance and theoretical interest, in fact algorithms for sorting data stored in memory devices have been the focus of extensive research, in general, sorting processes have a huge number of practical applications; most sorting algorithms are suitable for software applications rather than hardware ones, even with parallel processing sorting is a complex task that requires large silicon areas and interconection paths [65].

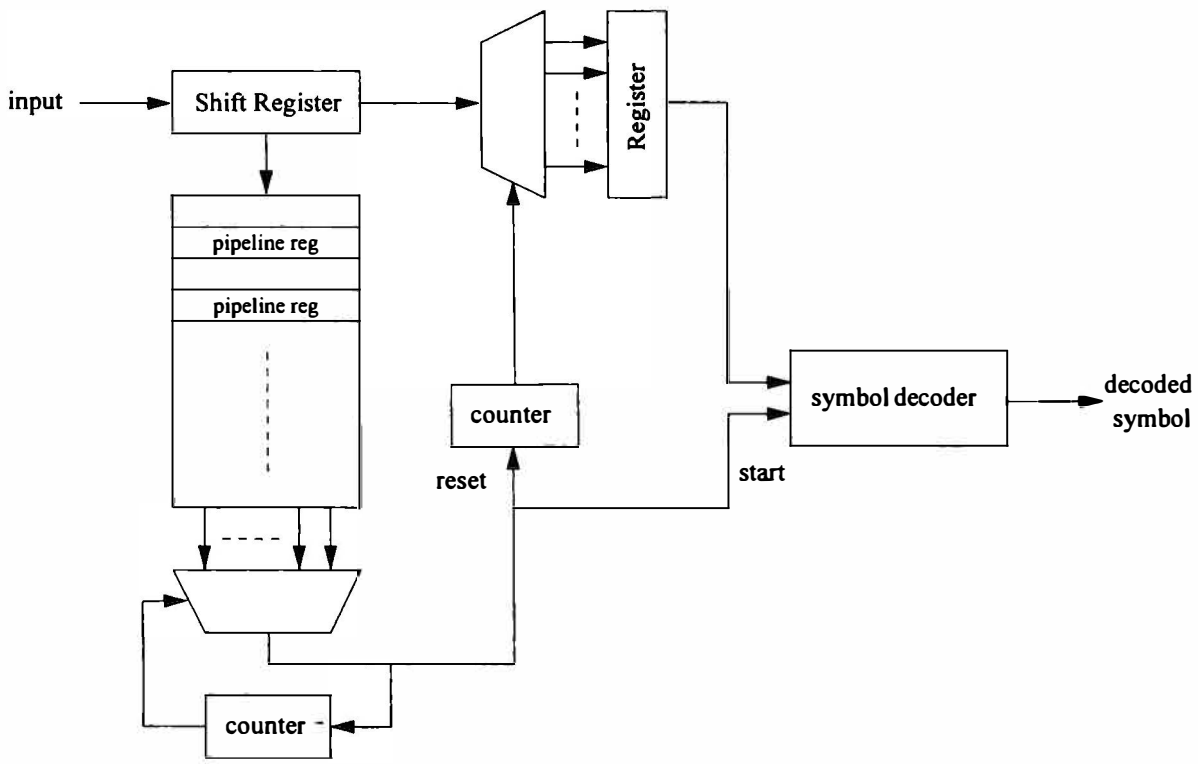



Figure 3.25: Rudberg's Pipelined Parallel Decoder

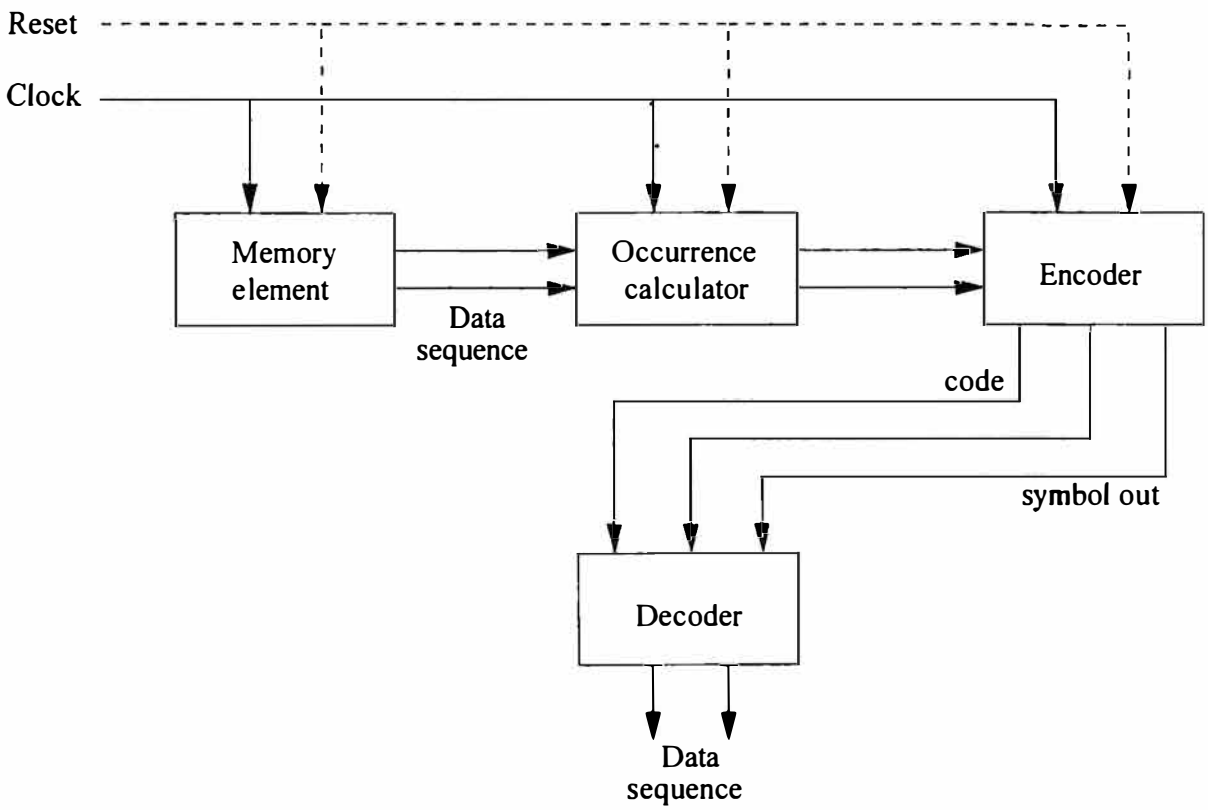



Figure 3.26: Kumar's Huffman codec

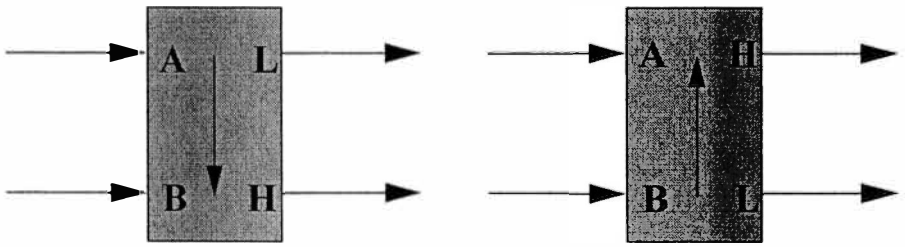
Sorting algorithms are designed to process large amounts of information at once, so it is not wise to use them if the number of values to sort is small as the material complexity will be to high. These algorithms are divided into two classes: serial and parallel, in the former it is assumed that values arrive in a serial fashion to the sorting circuit in such a way that the new value needs to be inserted in an already ordered list [66, 64]; in the latter all the values to be sorted are processed at the same time within an interconnected network [67, 68, 69, 70, 71].

As we know, to achieve high throughput rates, several operations must be performed simultaneously, even in multiprocessor architectures input-output operations must be concurrent with processing operations; the main problem in the design of these architectures is the interconnection between the various parts of the system so that all data transfers between modules can be accommodated within the selected technology, either DSP, FPGA or ASIC; High speed buses and wired matrixes can be used to provide interconnection between system modules but in the former the speed of available hardware limits the performance while in the latter large interconnection paths and crosspoints are required according to a material complexity study presented in [72].

## 3.3.1 Sorting Networks

### 3.3.1.1 Processing Element

The basic element of Sorting Networks is the compare-exchange element (CE) (figure 3.27), it receives two 9-bit numbers over inputs $A$ and $B$ and outputs the minimum number in the $L$ port and the maximum number in the $H$ port, the arrow tail indicates whether the processing element is ascending or descending.




(a) Ascending Sorting Element (b) Descending Sorting Element

Figure 3.27: Compare-Exchange (CE) elements for Sorting Networks

### 3.3.1.2 Bubble Sort

This is a well known algorithm based on the succesive paired comparison of adjacent elements of the list, in this way the small values emerge from the array while the greater values sink into the bottom of the array, the sorting process is depicted in figure 3.28; notice that we

need to compare every element with the next one, but when we finish to run this process there will be unsorted values as the comparison is only between two numbers at a time and we need to run this process over and over again until the array is completely sorted; therefore to achieve hardware implementation of this algorithm we require compare-exchange elements arranged as shown in figure 3.29.

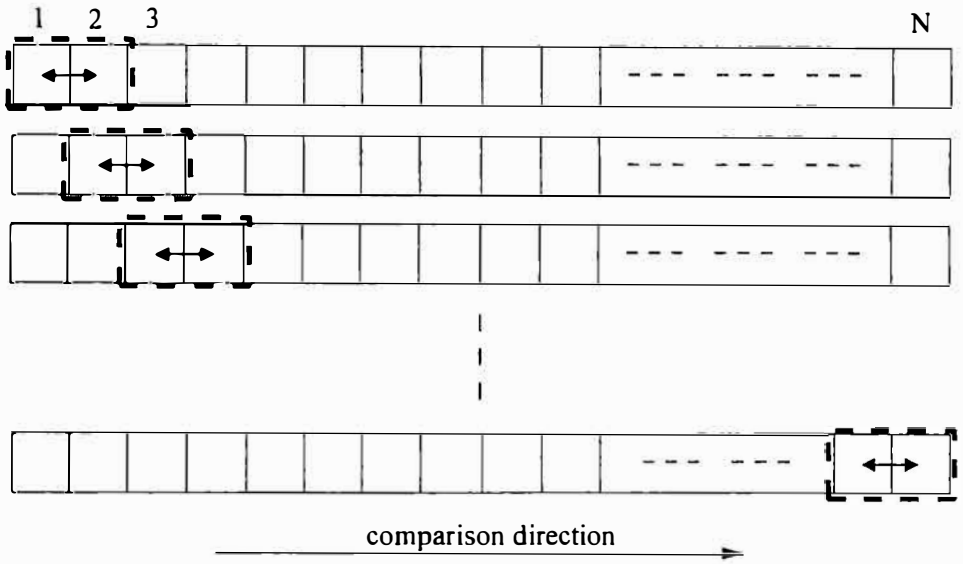



Figure 3.28: Bubble Sorting Process

This basic network has the following features:

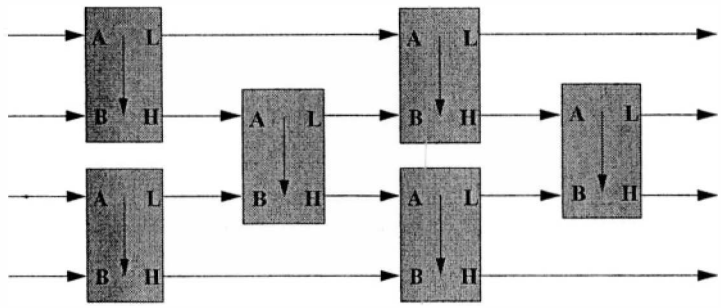
$$\text{Number of CEs} = \frac{N \cdot (N-1)}{2} \tag{3.27}$$

$$\text{Vertical CE layers} = N \tag{3.28}$$

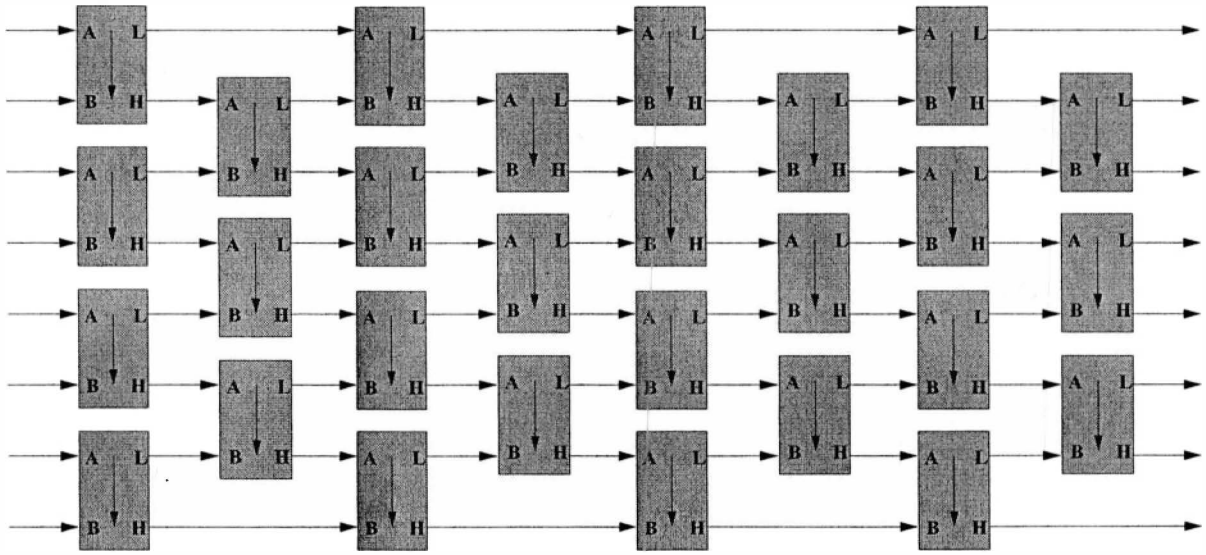
Figure 3.30 shows the sorting network for $N = 16$ items, it is clear that bubble sorting networks are easy to implement in hardware but they are very big arrays that are not efficiently interconnected, so this approach is based on brute force processing as it requires $N$ clock cycles to sort an $N$ item array; for small sets of values it could be useful, but once again, it is not the best option.

#### 3.3.1.3 Even-Odd Sorting Network

This sorting method was proposed by Batcher in [65], this network is based in the fact that two ordered lists of numbers can be combined or *merged* into a single ordered list; the even-odd sorting network is based on iterative merging stages that can be applied to an $N$-item list. The main goal of the network is to create small sorted lists of size $2, 4, \ldots, N$ during succesive




(a) 4 item sorting network




(b) 8 item sorting network

Figure 3.29: Bubble Sort Network

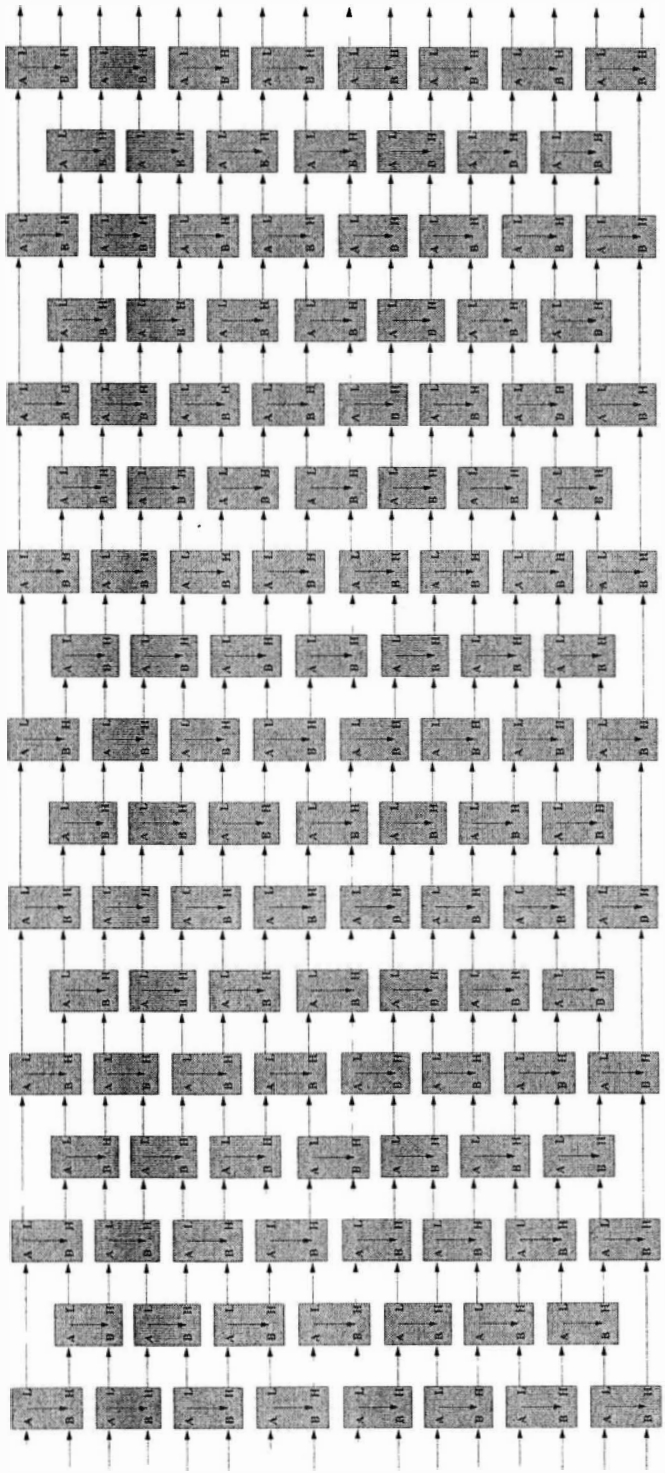


Figure 3.30: 16 item Bubble sorting network

processing stages, figure 3.31 shows the iterative rule of the even-odd merging network, we can observe that there are two sorted lists $A = \{a_1, a_2, \ldots, a_s\}$ and $B = \{b_1, b_2, \ldots, b_t\}$, these lists are fed into the even merging network and into the odd merging network; lists $C$ and $D$ are generated as outputs of the merging blocks; in this case, $C$ is formed by odd-numbered items of lists $A$ and $B$, list $D$ is formed by even-numbered ones, then lists $C$ and $D$ are merged to obtain the descendingly sorted list $E$ [50], equation set 3.29 shows the iterative rule of the even-odd sorting network.

$$\begin{aligned} e_1 &= c_1 \\ e_{2i} &= min(c_{i+1,d_i}) \qquad i = 1, 2, \cdots \\ e_{2i+1} &= max(c_{i+1,d_i}) \\ e_{s+t} &= d_t \end{aligned} \tag{3.29}$$

As we can see, two number merging is done by a single compare-exchange element, then, the merging network of four items is constructed using two 2-item merging networks and the iterative rule as shown in figure 3.32a, as stated previously the condition to use a merging network is that both lists are sorted, so, to implement a 4-item sorting network we need to place two compare-exchange elements before the 4-item merging network as shown in figure 3.32b. A sorting network of 8 items is shown in figure 3.33, notice that we require four 2-item merging netwoks (comparison-exchange elements) and two 4-item merging networks before entering the 8-item merging network; this means that the algorithm is recursive and we must distinguish between a **sorting network** and a **merging network**. Merging networks require two ordered lists as inputs to produce an ordered output list, sorting networks do not require input lists to be ordered as they will be propperly accomodated throughout the merging stages.

Figure 3.34 shows a sorting network for 16 numbers and figure 3.35 shows a sorting network for 32 numbers, notice that as we increase the size of the input lists the number of compare-exchange elements in merging netwoks also increases as shown in equation 3.30

$$\text{Number of CEs} = \log_2\left(\frac{N}{2}\right) \cdot \frac{N}{2} + 1 \tag{3.30}$$

We must remember that merging networks are useful when inputs are previously ordered lists of numbers, so we require additional compare-exchange elements to sort $2^N$ numbers that will serve as inputs of the merging networks; the number of compare-exchange elements required in a $N$-item sorting network are:

$$\text{Number of CEs} = \frac{N}{4} \cdot \left[(\log_2 N)^2 - \log_2 N + 4\right] - 1 \tag{3.31}$$

As the even-odd sorting network is based on an iterative fashion we require some layers of comparison-exchange elements to sort a random list of numbers.

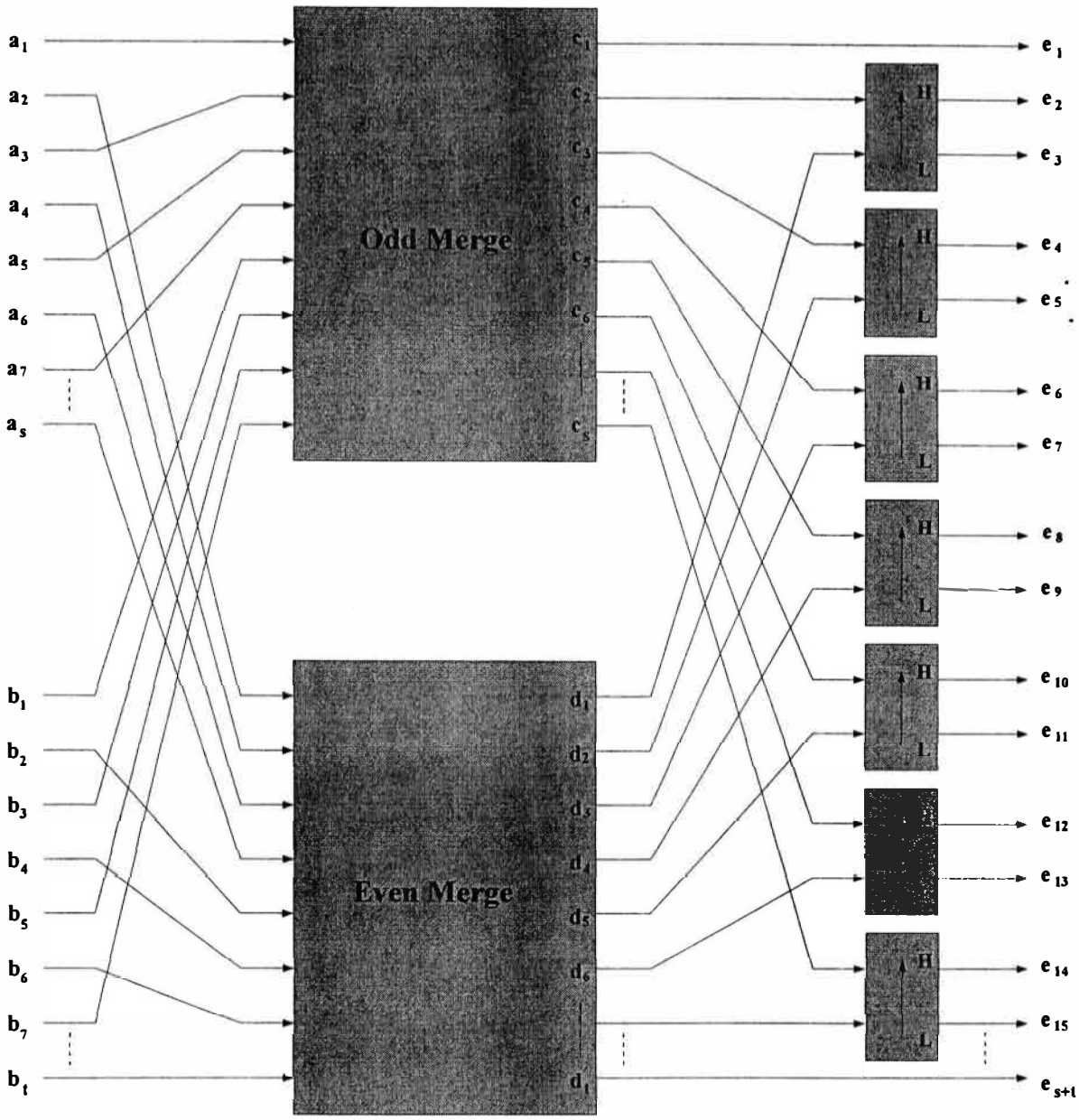
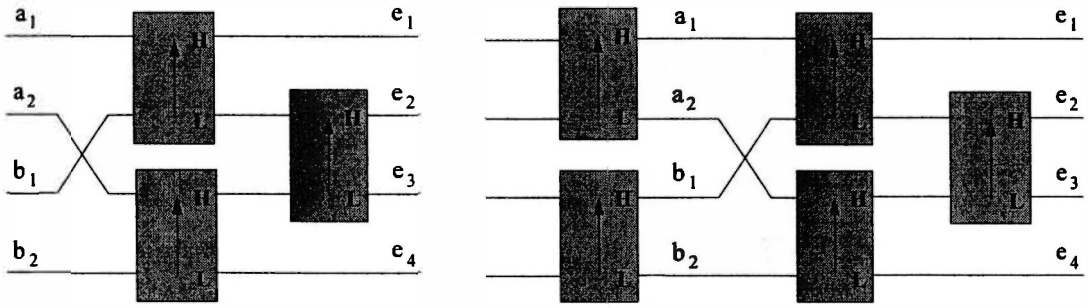



Figure 3.31: Iterative rule for Even-Odd merging networks




(a) even-odd merging network (b) even-odd sorting network

Figure 3.32: 4-item merging and sorting networks

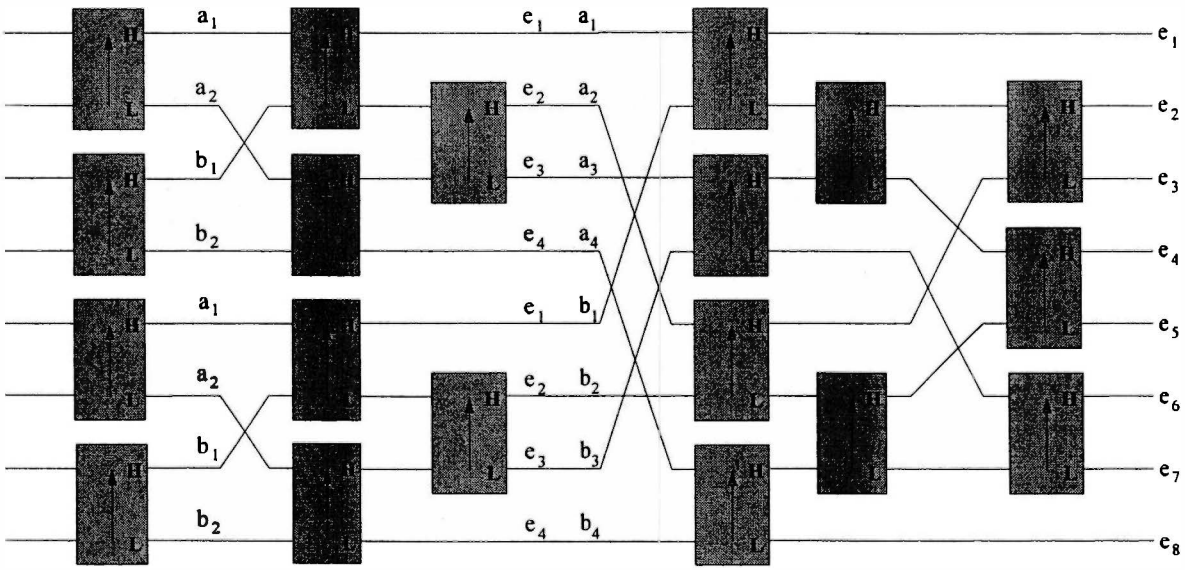



Figure 3.33: 8 item sorting network

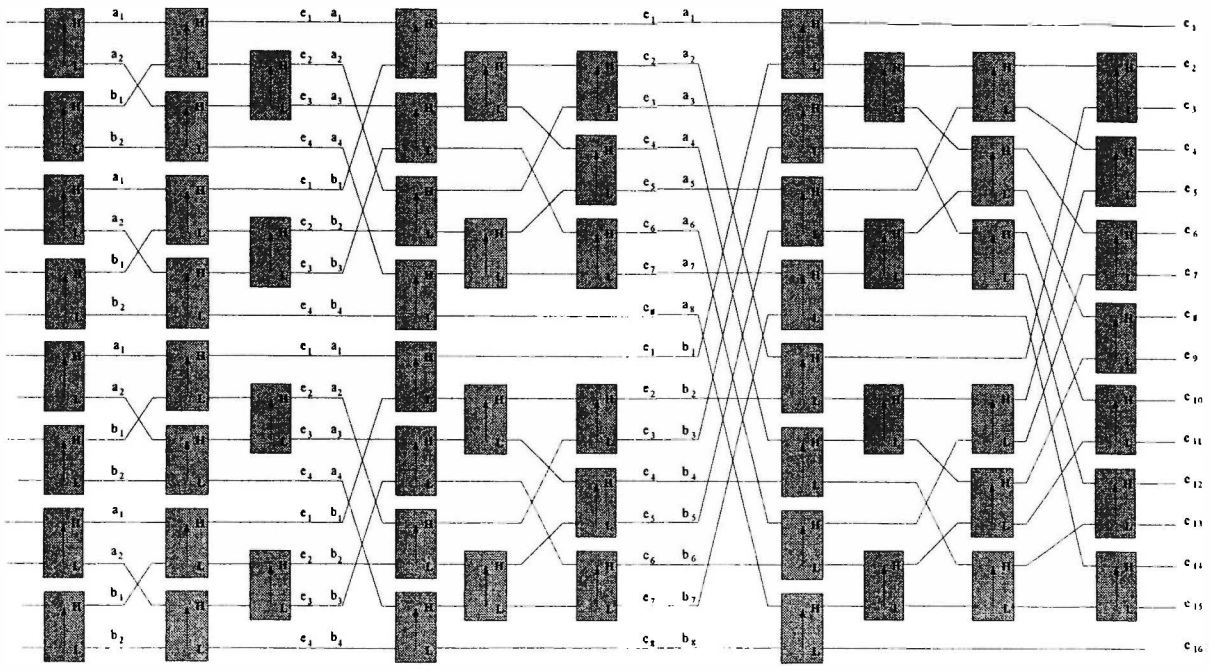


Figure 3.34: 16 item sorting network

$$\text{Vertical CE layers} = \frac{(1 + \log_2 N) \cdot \log_2 N}{2} \tag{3.32}$$

Notice that the smallest or greatest item (depending on the sorting mode) takes $\log_2 N$ to be known because of the merging netwoks structure; also, pipelining can be introduced to reduce the critical path, but we must be careful when implementing such a strategy; it is known that we cannot use pipeline barriers between all the stages of the algorithm because there is a point where pipelining not only does not help to reduce critical path but it decreases the maximum operating frequency.

#### 3.3.1.4 Bitonic Merging Network

This sorting network is also based on Batcher's work as stated in [64]; a bitonic list is obtained by means of the concatenation of two lists, one in ascending order and the other in descending order. Consider $A = \{a_1, a_2, \cdots, a_N\}$ a bitonic list that can be splitted into two lists as shown in 3.33 and 3.34, these two lists are also bitonic and 3.34 contains the smallest value of the $N$-item list; the iterative rule of this network is shown in figure 3.37.

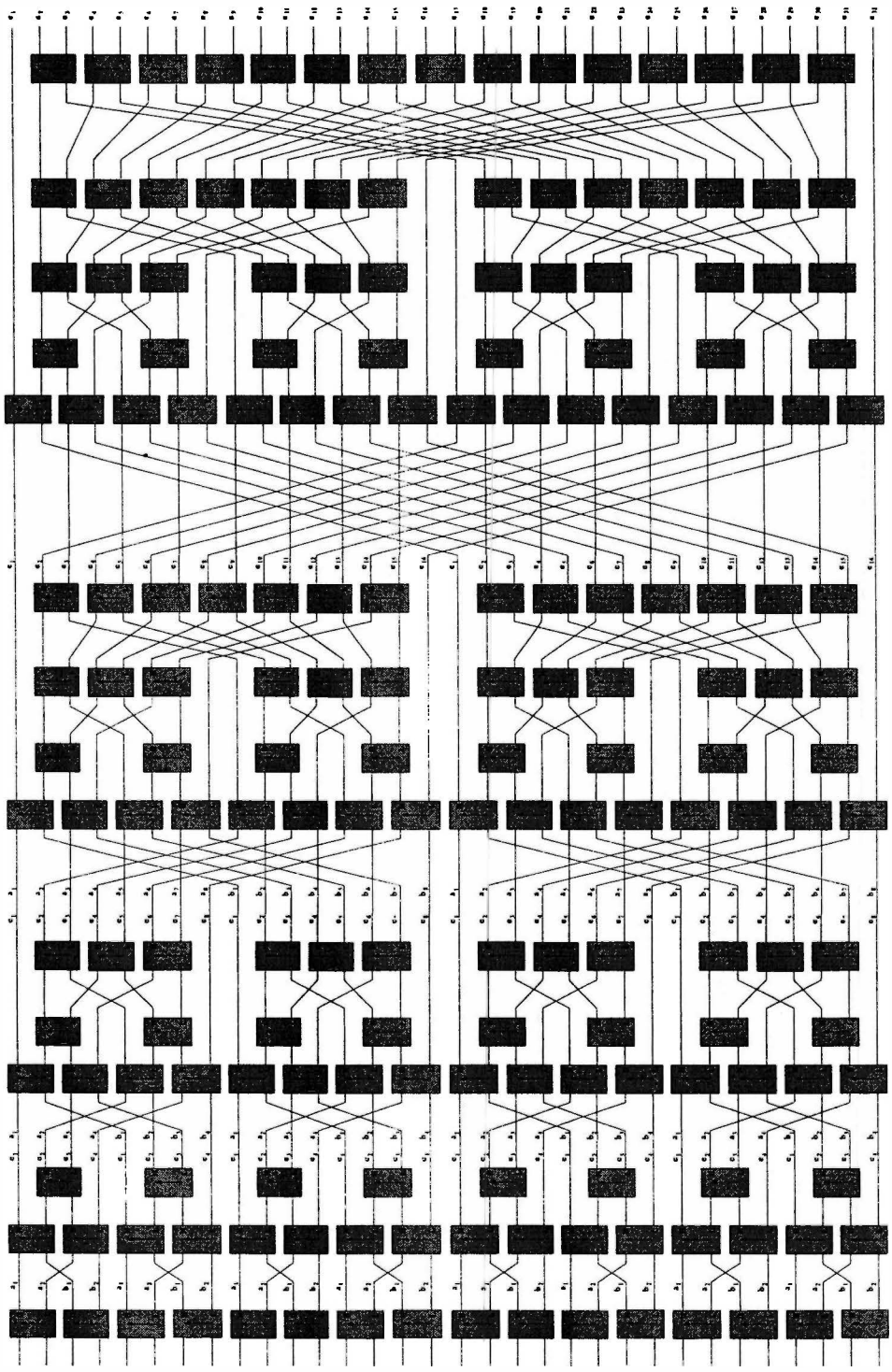


Figure 3.35: 32 item sorting network

$$\text{Ascending list} = \left\{\min\left(a_1, a_{\frac{N}{2}+1}\right), \min\left(a_2, a_{\frac{N}{2}+2}\right), \cdots, \min\left(a_{\frac{N}{2}}, a_N\right)\right\} \tag{3.33}$$

$$\text{Descending list} = \left\{\max\left(a_1, a_{\frac{N}{2}+1}\right), \max\left(a_2, a_{\frac{N}{2}+2}\right), \cdots, \max\left(a_{\frac{N}{2}}, a_N\right)\right\} \tag{3.34}$$

Just as the even-odd merging network, the smallest bitonic sorter is a compare-exchange element, figure shows a four item sorting network, notice that before entering the merging stage, the initial list is divided into two new lists; figure 3.38 shows a sorting network of 8 items, as in the even-odd sorting networks, this algorithm is recursive and after a two element list has been sorted, then 4 item merging networks are required to output 8-item lists that will be fed into 16-item merging networks an so on until they are merged into a single list.

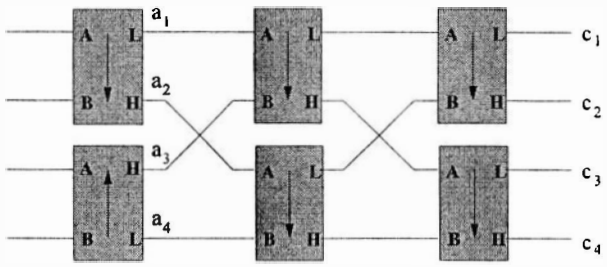



Figure 3.36: 4 item bitonic sorting network

The required number of compare-exchange elements required in an $N$-item merging network is

$$\text{Number of CEs} = \frac{N}{2} \cdot \log_2 N \tag{3.35}$$

The number of CEs increases if we configure a sorting network, equation 3.36 shows the required number of CEs for $N$-item sorting networks.

$$\text{Number of CEs} = \frac{N \cdot \log_2 N \cdot (\log_2 N + 1)}{2} \tag{3.36}$$

Finally, the number of vertical layers of CE elements is expressed in equation 3.37, this algorithm is suitable for pipelining but we must be careful and attend the same recomendations of even-odd sorting networks.

$$\text{Vertical CE layers} = \frac{(1 + \log_2 N) \cdot \log_2 N}{2} \tag{3.37}$$

### 3.3.2 Serial Sorting

Serial sorting or insertion algorithms are based in serial data processing fashion, that is, every number or data to be sorted arrives one by one to a $N$ element array where it is inserted

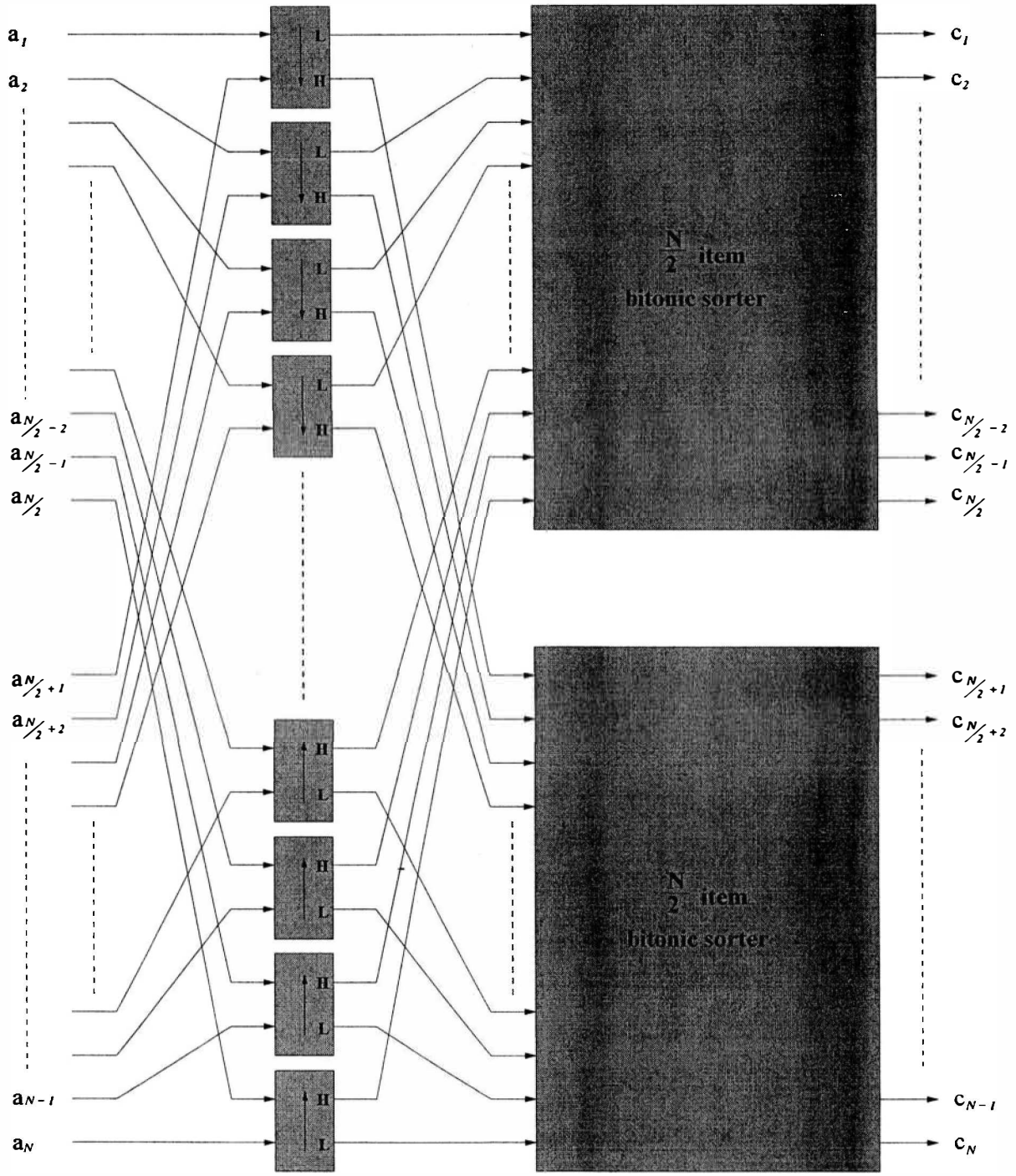



Figure 3.37: Iterative rule for Bitonic Merging Networks

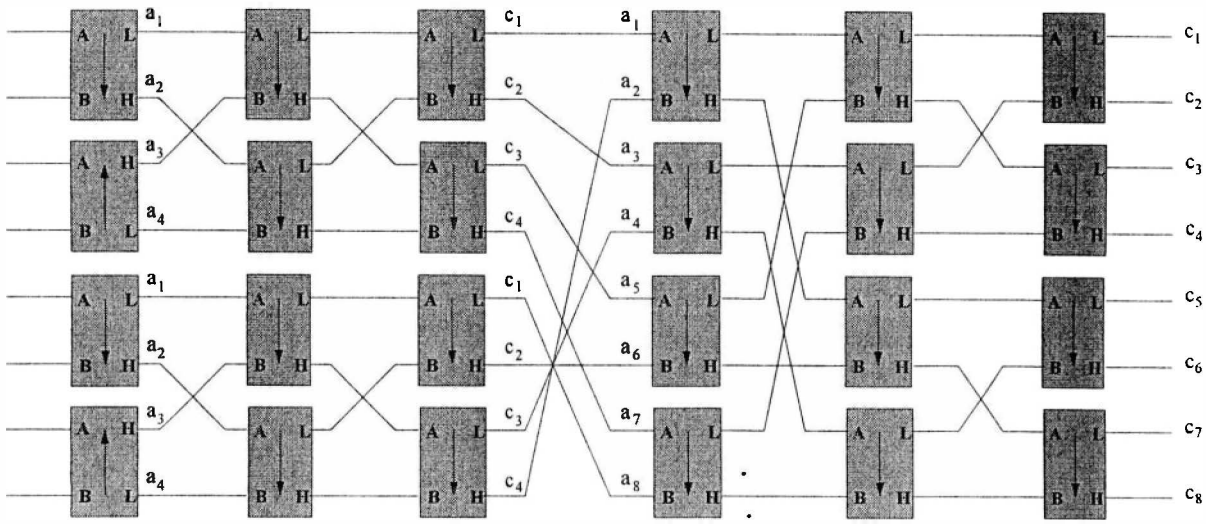


Figure 3.38: 8-item bitonic sorting network

in the corresponding position, there are some reported architectures in literature for serial sorting. For the purpouse of this work we are going to describe in general the most common ones.

### 3.3.2.1 Insertion Algorithm

This algorithm is based on the sequential comparison of the item we want to include in an array with every item already inserted in the sorting architecture, for an insertion to be succesful we require to perform the operations on an already ordered array; when a new element arrives to the array it is compared with the first element of the ordered list if it is smaller it is inserted into that position, if it is greater then it is passed to the second element and so on; if the arriving element is inserted, then the previous value of that position is then compared to the following item producing a shift operation, figure 3.39 shows the single insertion algorithm, notice that we do not require compare-exchange elements, we can manage the operations with a simple comparator that outputs the largest number of two input and generates a control signal that controls the register assigned to that position.

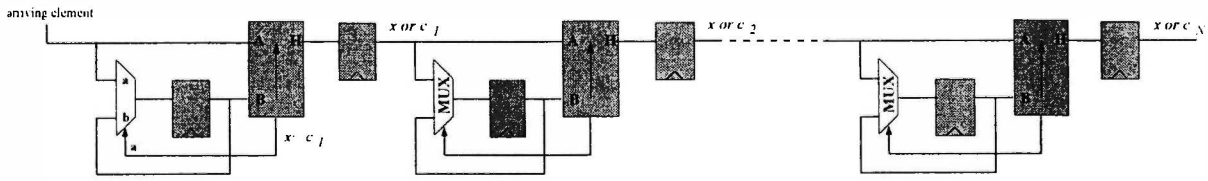



Figure 3.39: Single insertion architecture for $N$ elements

This sorting algorithm requires $N$ comparators and has a processing time of

$$\text{Processing Time} = \frac{N \cdot (N-1)}{2} \tag{3.38}$$

### 3.3.2.2 Parallel Insertion

This algorithm compares the arriving element with all of the previously inserted elements at the same time, there are three posibilities when performing such a comparison shown in figure 3.40, In the first case, when the inserting element is smaller than the elements to its right we insert the value into that position and shift all the elements to the right; the second case is presented when the inserting element is larger than the element of the $i^{th}$ position but smaller than the element in the $i+1$ position, when this occurs, the new element must be inserted in the $(i+1)^{th}$ position. Finally, when the inserting element is larger than the last element we simply insert the value in the last position and no further operations are required.

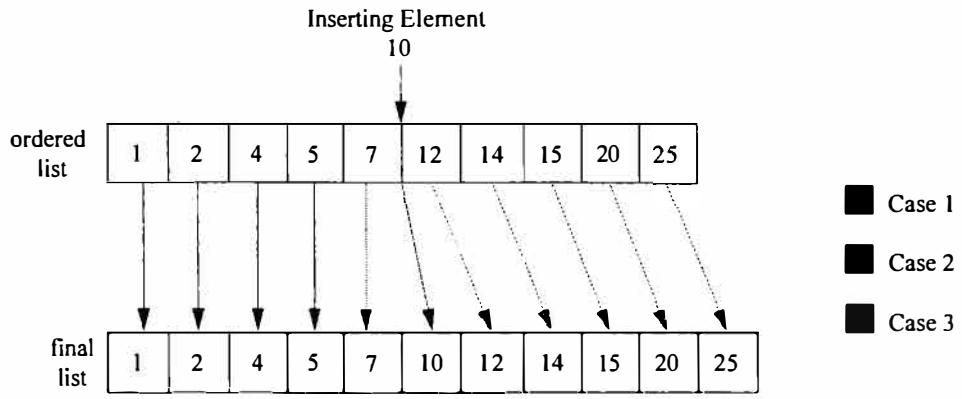



Figure 3.40: Cases of parallel insertion

The parallel insertion architecture is shown in figure , notice that the comparison element consists of a simple comparator that outputs the smalles number, a multiplexer and a register to hold the value for that particular position. The multiplexer that controls register's input is driven by the output of the current register $i$ and the output of the previous register $i-1$. If the inserting element is smaller than the element of register $i$ the output of the comparator will send a control signal to the multiplexer to select the inserting element to be stored in the $i^{th}$ position; the output of comparators $i$ and $i+1$ will select the element previously stored in the $i^{th}$ position to be stored in the register assigned to $i+1$ position, that is, the stored elements will simply be shifted to the right as shown in figure 3.41.

### 3.3.2.3 Dichotomic Insertion

This algorithm compares the inserting element to the middle element of an already sorted list, then, depending on the result of this compariso, the inserting element is moved either

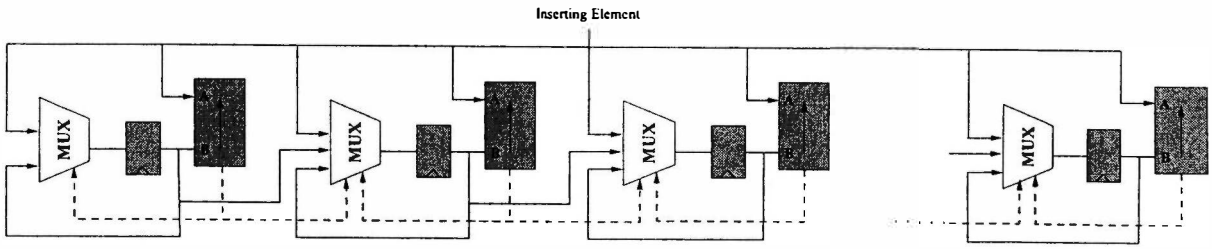



Figure 3.41: Parallel insertion architecture

to the upper half or to the lower half of the array, then, the inserting element is compared to the middle element of the corresponding array half, this process continues until there is only one value to compare with.

The comparison elements used in this structure have a special feature, only the inserting element is routed to the high or low output depending on the comparison result, this sorting algorithm requires a certain number of comparisons before the array is completely ordered, equation 3.39 shows the required number of comparators and equation 3.40 expresses the number of comparator layers.

$$\text{Number of Comparisons} = N \cdot \log_2 N \tag{3.39}$$

$$\text{Comparator Layers} = \log_2 N \tag{3.40}$$

The greatest disadvantage of this algorithm is that pipelining cannot be used to accelerate the sorting process as we must wait for the current inserting element before starting the next element insertion, the reason is that middle elements of the array must be updated.

If we desire to obtain the smallest $N$ values from a list of $M$ items (where $M > N$) we must add an extra compare element and a multiplexer, so that when we compare the largest elements of the list with the inserting element, the latter is either inserted in the penultimate position, the last position or not inserted at all, figure 3.42 shows the architecture for an 8 item list.

## 3.4 Quantizer

Intra coded frames have their DC coefficients coded differentially with respect to the previous block of the same type unless it belongs to another slice, in which case the DC coefficient is differentially coded with respect to 1024, the range of unquantized coefficiets is

$$Q_{levels} = \{0, 1, 2, \cdots 8 \times (2^N - 1)\} \tag{3.41}$$

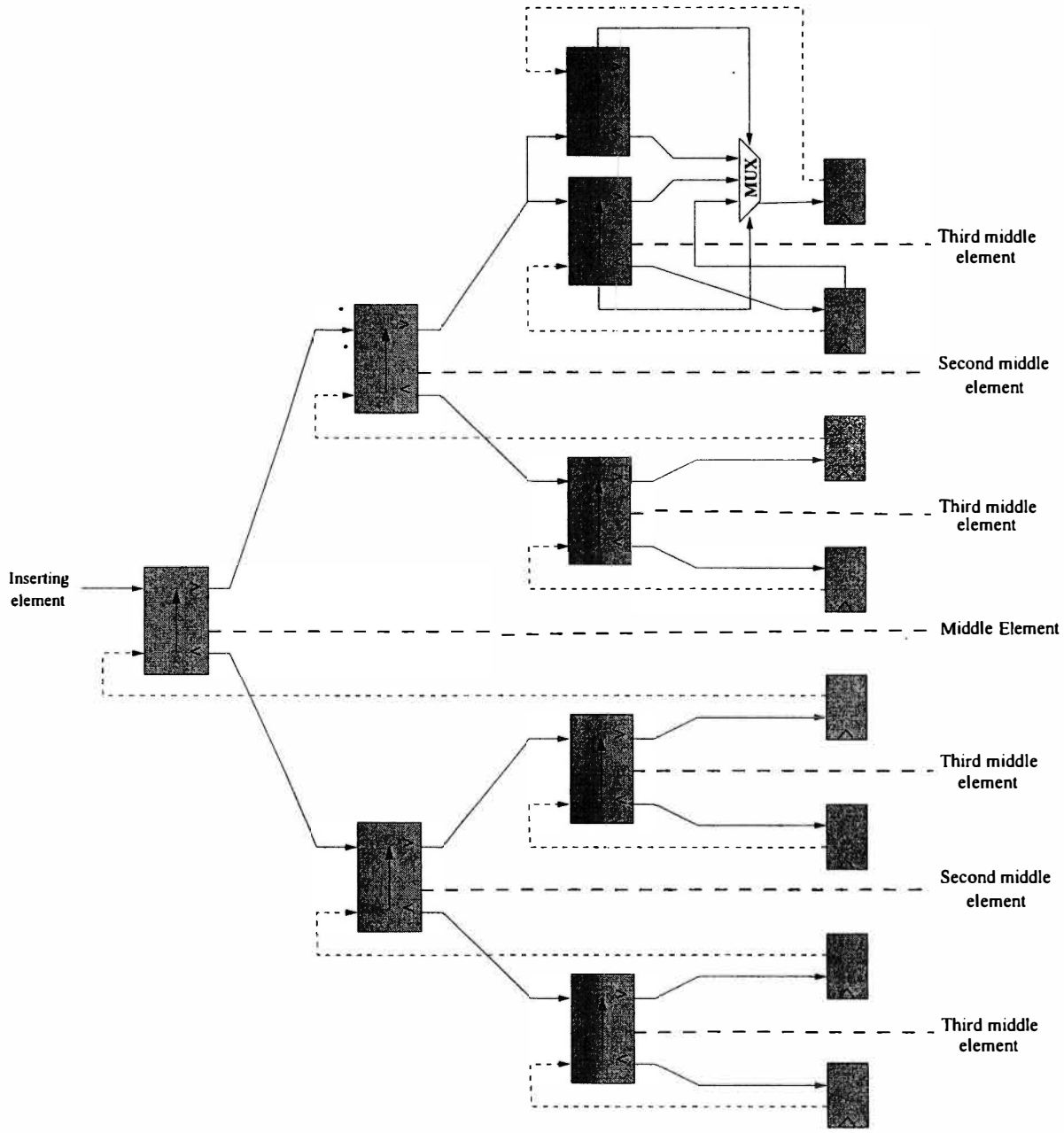



Figure 3.42: Dichotomic Insertion Architecture for $N = 8$

Which means the range of differential values for $N = 9$ is [−4088 to 4088], intra DC differential coefficients are quantized with a fixed step size of 8, thus, quantization is implemented by dividing by eight and rounding to the nearest integer; to accomplish this task in hardware we must consider that DCT coefficients are 12-bit long, so dividing by 8 implies a simple three-position right shift, no rounding stage is needed as the shift operation is performed over binary integers, this simple process is shown from left to right in table 3.5

| Decimal | Hexadecimal | Binary | Right shift | Quantized Coefficient |
|---|---|---|---|---|
| −1511 | A19 | 101000011001 | 101000011 | −188 |
| −176 | F50 | 111101010000 | 111101010 | −22 |
| 510 | 1FE | 000111111110 | 000111111 | 63 |
| 933 | 3A5 | 001110100101 | 001110100 | 116 |
| 1977 | 7B9 | 011110111001 | 011110111 | 247 |

Table 3.5: Quantization

The VLSI implementation of quantization can be achieved using a hardwired shift to crop the three least significan bits of the 12-bit word as shown in figure 3.43; as the quantization is a lossy process, to reverse the process we could simply fill with zeros to complete the 12-bit word length, but in order to add some randomness in the process we propose the use of 2-bit logic gates to generate the required least significan bits as shown in figure 3.44

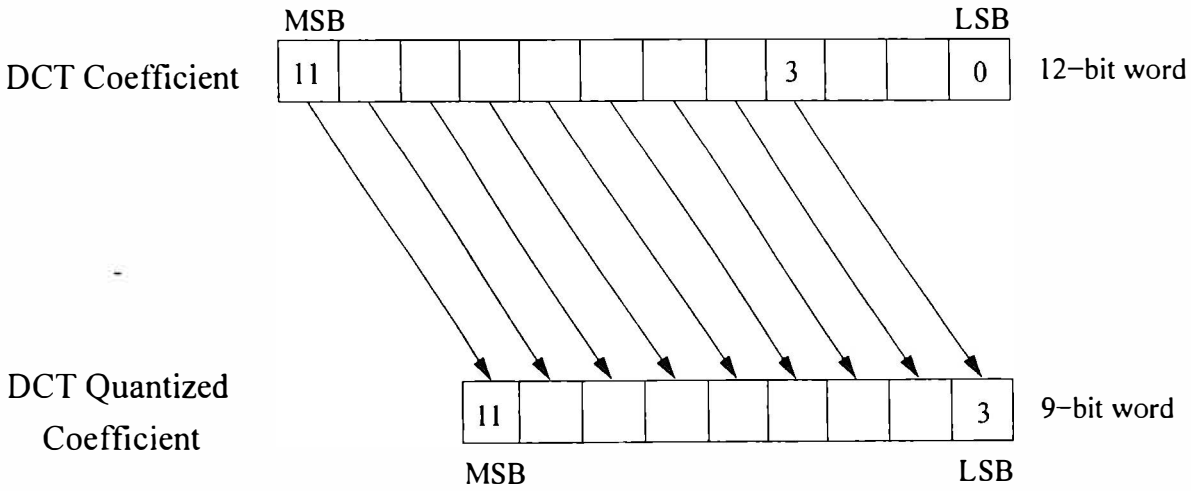



Figure 3.43: Quantization Process

## 3.5 Predictor

Commonly two consecutive frames of a sequence are very similar as they are temporal variations either in the foreground or in the background of an image [4, 3, 2], this fact is the basis

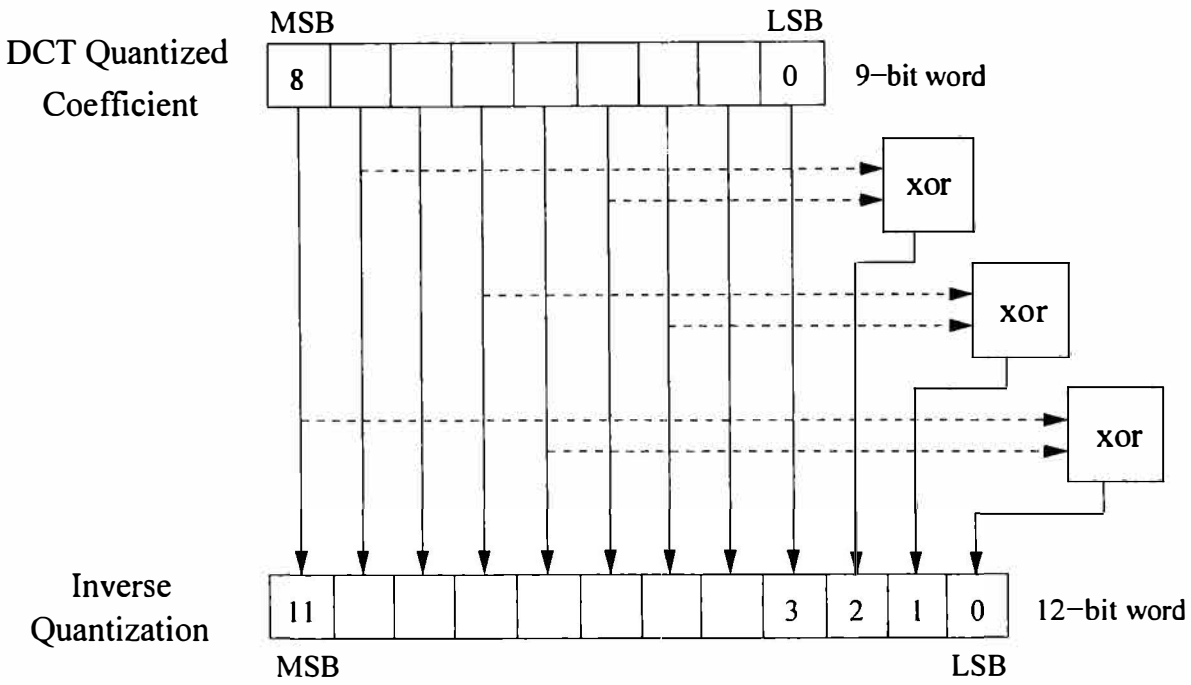



Figure 3.44: Inverse Quantization Process

of Motion Compensation, where a $M \times N$ frame is coded differentially with respect to another frame; in most video coding standards frames are divided into blocks of $n \times n$ pixels as it is easier to process small groups of adjacent pixels rather than processing the entire frame at once [73]; typical values of $n$ are 4, 8 and 16. The resulting blocks of the segmentation of the current frame are called *reference blocks* or *source blocks*; the process of finding a block in a different frame that best matches the reference block is known as *Motion Estimation*, the motion vector points to the position of the best matching block within a rectangular area around the position of the reference block called *search window*. Motion Estimation is the most computationally intensive component of coding algorithms [74], it could consume as much as 75% of the total processing power of a video codec.

Figure 3.45 shows the process of block matching, notice that the search window extends on both sides over $\frac{n}{2}$ pixels, the shaded square represents the source block and the motion vector is represented in red.

Motion estimation is a highly demanding task in terms of computational operations, there are two great families of Motion Estimators, one is based in the Block Matching Algorithm (BMA) [75, 76, 77] and the other is based on a Hierarchical Search Algorithm (HSA) [78, 79]; full search algorithms have high material complexity, but the control logic is simple and suitable for hardware implementation as their structure is highly regular [73], robust and has fixed operation steps, but we must take into account that large silicon areas might be

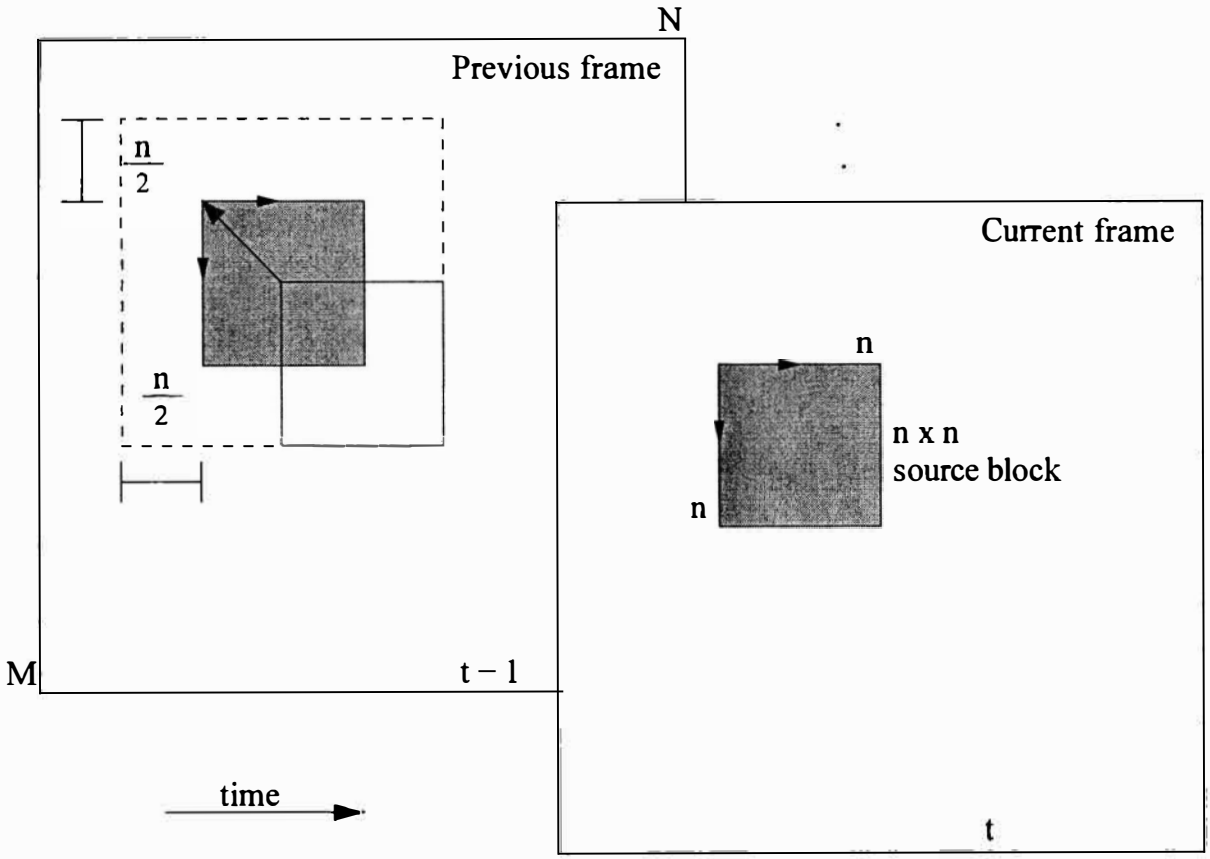



Figure 3.45: Block searching fundamentals

required to achieve significant speed; hierarchical search algorithms emerged as a solution to reduce the vast number of operations required for full search BMAs, this drastical reduction came along with a complex control logic, HSA data flow is not regular, as it is not a sequential process, hardware implementation of HSA must have low latency components to achieve high throughput rates. Both families use many algorithms to determine the motion vector, most common algorithms are Mean Absolute Error (MAE), Sum of Absolute Differences (SAD) [80], Global Elimination Algorithm (GEA) [81] and Mean Absolute Difference (MAD).

### 3.5.1 Full Search Block Matching Algorithm

To estimate a motion vector using block matching we must remember that the frame is divided into $n \times n$ pixel blocks, for each source block a corresponding matching block is sought within a search window surrounding the source block as shown in figure 3.45, as mentioned before, the search area regularly extends on both sides over the half of the distance from 0 to $n$, for notation purpouses we will denote the absolute distance from $-\frac{n}{2}$ to $\frac{n}{2}$ as $m$, so that the search area contains $(n+m)^2$ pixels [73]; the distance from the source to the corresponding block is written as

$$D(\Delta i, \Delta j) = \sum_{k=1}^{n} \sum_{l=1}^{n} |x_t(k,l) - x_{t-1}(k+\Delta i, l+\Delta j)| \tag{3.42}$$

Term $x_t$ represents the source block pixel and $x_{t-1}$ represents the previous frame pixel, $(k,l)$ are simply the pixel coordinates within a search block and $(\Delta i, \Delta j)$ represent the components of the vector that corresponds to the compared block; these components are defined over the search window as shown in 3.43.

$$\begin{aligned} \Delta i &= \left[-\frac{m}{2}, \cdots, -1, 0, 1, 2, \cdots, \frac{m}{2}\right] \\ \Delta j &= \left[-\frac{m}{2}, \cdots, -1, 0, 1, 2, \cdots, \frac{m}{2}\right] \end{aligned} \tag{3.43}$$

The absolute value of $x$ represents the largest number that is less or equal to the **real** number $x$, so $(m+1)^2$ distance measures must be calculated per block; Motion vector $(\Delta i, \Delta j)^*$ is in fact a displacement vector where $D(\Delta i, \Delta j)$ is minimum

$$(\Delta i, \Delta j)^* = min^{-1} D(\Delta i, \Delta j) \tag{3.44}$$

Just as we did with the DCT, we can decompose the full search block matching algorithm to parallelize operations and accelerate calculations, De Vos describes 3.42 as a set of nested loops that can be processed in any order; if we calculate in parallel two of these loops the architecture can be arranged as a two dimensional array of processing elements that accumulate the absolute difference values; if we decide to calculate one loop at a time we will obtain a linear array architecture. In [73] two types of arrays are described, in type 1 array the $k$ and $l$ loops are parallelized and mapped into hardware, the absolute difference values that

correspond to one distance measure are calculated concurrently in $n^2$ processing elements, after $(m+1)^2$ clock cycles all distance measures for $\Delta i$ and $\Delta j$ are calculated; figure 3.46 shows type 1 array computation, notice that all possible positions of the previous block are considered within the search area.

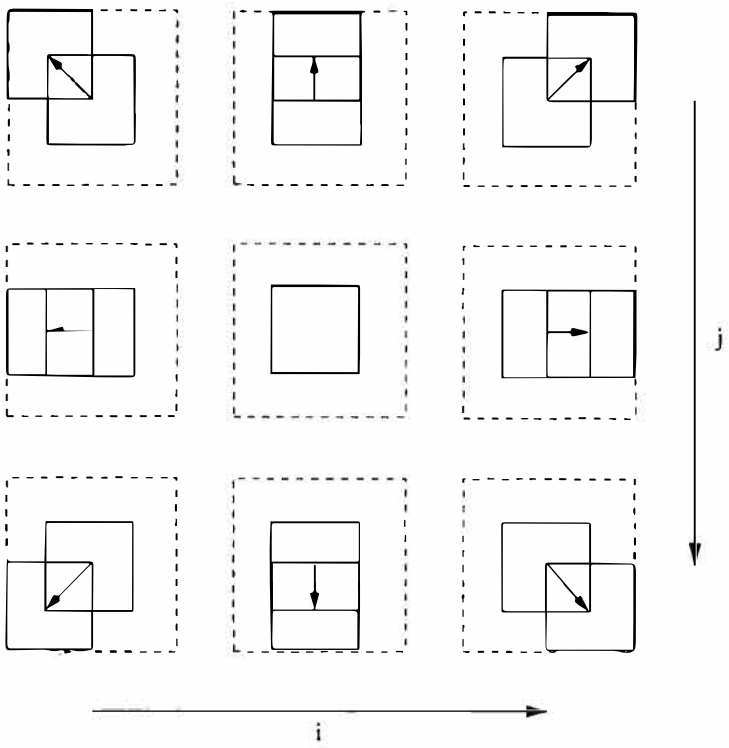



Figure 3.46: Type 1 Array for Block Matching Algorithm

For type 2 array $\Delta i$ and $\Delta j$ loops are mapped into hardware, this implies that one pixel $x_t(k,l)$ is treated in each processing cycle; this array has $(m+1)^2$ processing elements, each of them corresponds to a specific displacement vector; after $n^2$ steps all the pixels of the current block are processed and $(m+1)^2$ distance measures are available simultaneously, figure 3.47 shows the computation sequence required for type 2 arrays. Both array types are Systolic Array Processors [82] this architectures process algorithms with few different instructions but high computational and data rates; this means that they are restricted to the algorithms they were derived from; to achieve an optimum performance the conditional portions of the algorithm are implemented in reconfigurable hardware and the fixed parts are implemented in dedicated hardware, a reported implementation of FBMA can be found in [83, 84, 85].

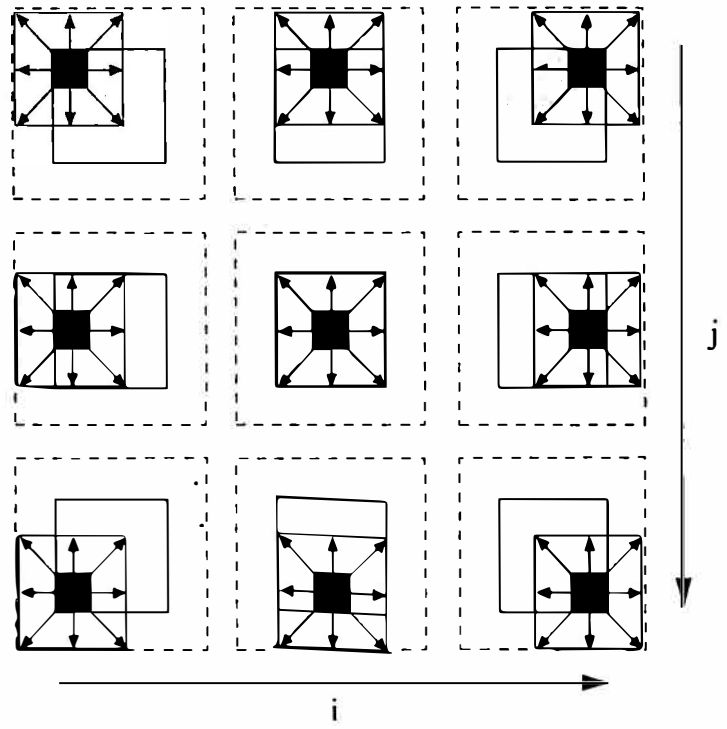



Figure 3.47: Type 2 Array for Block Matching Algorithm

### 3.5.2 Hierarchical Search Algorithm

As mentioned before, full search block matching algorithm architectures involve massive computational efforts and high material complexity [86]; computationally speaking, Hierarchical Search Algorithm architectures are simpler and reduce the amount of operations required to find a motion vector than BMAs, their greatest disadvantage is that they require complex control logic, in exchange they are used to reduce the amount of redundant block matching operations.

HSA technique works with a multi-step search algorithm through successive approximations of the best matching block; HSA is based on the assumption that movement within a sequence must be smooth; let us consider a $N \times N$ block centered in $(0,0)$ as shown in figure 3.48, also take into account a maximum pixel displacement $p$, the mean absolute difference (MAD) of the blocks within a search area delimited by $d = (p+1)/2$ is calculated, pixels labeled as 1 identify the leftmost top pixel of each block, the minimum distortion position is then labeled as $I$; The next step of HSA begins in position $I$, the search area is now delimited by $d/2$, top left pixels of these blocks are labeled as 2, just as in the previous step, the position with the minimum distortion is labeled as $II$, this process continues in the same fashion until the displacement converges to 1. For the maximum displacement $p$ the number of required steps to compute the motion vector are $\log_2(p+1)$, for this reason, the method is also known as Two-Dimensional Logarithmic Search.

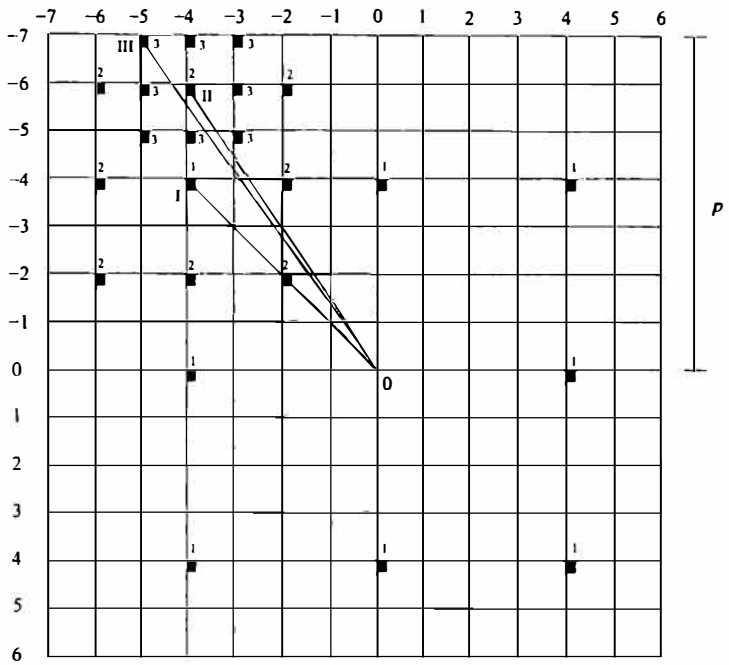



Figure 3.48: Hierarchical search algorithm methodology

### 3.5.3 Known Implementations

In [87] a tree architecture for motion estimation is presented, authors claim that their design has high throughput, low data skewing for the purpouse of parallel computation, low latency and an independent data flow so that the tree can be used for BMA and HSA, the main contribution is the implementation of interleaving in memory devices and pipeline stages to avoid idle cycles in the process, the main drawback is that interleaving techniques cause latency in the architecture. In [88] a full search block matching architecture is presented, the main contribution is that the motion vector is generate in 1252 clock cycles considering a Sum of Absolute Differences (SAD) algorithm, the architecture is based on two independent modules, a SAD unit that computes the difference between blocks and a motion vector generator used to find the minimum value in the SAD array, this system has a systolic behavior but it can operate in a single instruction multiple data (SIMD) fashion, authors report that the motion vector for a $16 \times 16$ block can be computed in $54.4\mu s$.Nam et al. present a flexible VLSI architecture for full search motion estimation in [89], this architecture has serial input ports but the processing is performed in parallel and is parametrizable as it allows to set the search range for different video aplications, finally the structure is highly regular and it is mapped to one dimensional arrays.

Gupta and Chakrabarti propose various architectures for hierarchical block matching algorithms in [78, 79]; in [79] are presented some architectures for hierarchical BMA for generalized parameters, an extensive study of the number of comparisons and number of operations required to match a block is presented, table 3.6 shows the parameters for a simplified hierarchical BMA with a maximum displacement of $\pm 10$, a two step search is assumed.

| Parameter | Level 1 | Level 2 |
|---|---|---|
| Max. update displacement ($d_l$) | $\pm 7$ | $\pm 3$ |
| Search window size ($W_l$) | 64 | 16 |
| Step size ($S_l$) | 32 | 16 |
| Subsampling | 8 | 2 |

Table 3.6: Parameters for simplified hierarchical block matching algorithm

Considering an $M \times N$ image, at any level of estimation the number of motion vector estimates is $M \times N/S_l^2$, and $W_l^2/U_l^2$ operations per block match are required. In the first level 27 blocks are matched per estimate, in the second level 18 blocks are matched; considering a frame frequency of $f$ Hz the number of operations per second is calculated in the following way:

$$M \times N \times f \left( 27 \left( \frac{W_1}{S_1 U_1} \right)^2 + 18 \left( \frac{W_2}{S_2 U_2} \right)^2 \right) \tag{3.45}$$

Considering a processing cycle time $c$, authors estimated the minimum number of processors required for real-time data computation as

$$p = \left\lceil 9 \cdot M \cdot N \cdot f \cdot c \left( 3 \left( \frac{W_1}{S_1 U_1} \right)^2 + 2 \left( \frac{W_2}{S_2 U_2} \right)^2 \right) \right\rceil \tag{3.46}$$

For further details on the processor architecture and the memory organization please refer to [79]; In [90] is presented a low complexity block based motion estimation using one-bit transforms, the main idea of this paper is to transform video sequences into a one-bit/pixel representation an then apply the motion estimation methodology to reduce hardware complexity and power consumption while maintaining a good compression ratio. One-bit strategy is based on the fact that the edges in an image are fundamental for accurate motion estimation, therefore to extract the edges we must compare the frame pixel by pixel to a high-pass filtered version of the frame and force the differences to 0 or to 1; this process causes the thresholded frame to track high frequency noise, so, to eliminate it band-pass thresholding scheme is applied to maintain only the original content of the frame.

Baek et al propose a reduced bits mean absolute difference (RBMAD) in [91] this methodology is used to reduce silicon area for material implementation at the same time that VLSI operations are boosted, the principle of operation is simple we only require the MSB of each

pixel to compare the source block with all the candidate blocks, this criterion is described in equation 3.47, $R$ represents the leftmost top pixel of the source block, $S$ is the position of the pixel on the candidate block of the previous frame, just as in MAD method, we look forward to finding the smallest RBMAD to determine the motion vector.

$$RBMAD(u,v) = \sum_{i=1}^{N}\sum_{j=1}^{N} |R(i,j)_{n-1:n-k} - S(i+u,j+v)_{n-1:n-k}| \tag{3.47}$$

Do and Yun [92] present an architectural enhancement to reduce power consumption of Full-search BMA motion estimation by means of reducing unnecessary computation steps applying a method known as conservative estimate of exact distortion, power consumption is reduced as the differences are only measured if the estimate is smaller than the current minimum distortion; every processing element computes the absolute difference between luminance levels of the source block and the search window, in this architecture the processing elements are connected in a systolic array fashion, each layer of the array has a shift register at the end to store the pixel data of macroblocks not actively under examination.

# Conclusions

There are many families of architectures used to compute the DCT-II, the great majority of reported implementations are either recursive application of even-odd decomposition (fast algorithms) or distributed arithmetic considering direct implementations and memory reduction techniques. All fast algorithms are very efficient in the required number of multiplications and additions and are a well suited for ASIC implementation althoug large silicon areas are required, they are not a good option for FPGA because of its long propagation delays due large routing and interconection butterflies; fast algorithms also have serious problems when computing finite precision numbers because several rounding and truncation adequations are required throughout the different computing stages. In the other hand distributed arithmetic [93] implementations have a regular structure suitable for FPGA; this architectures are memory oriented and they offer high speed, high accuracy and their principal advantage over fast algorithms: *design time is dramatically reduced.*

There are many Huffman coder/decoder implementations, all of them share one interesting feature: *they are designed for small codeword tables* and the material complexity is usually high and require a dedicated integrated circuit; even though the great amount of reported architectures, there are few articles that report the number of logic elements, registers and propagation delays, making difficult to forecast the exact amount of resources that will be used on the implementation.

Sorting algorithms can be used to build symbol frequency tables to accurately assign Huffman codes to input data, we must remember that after DCT the energy is concentrated in the

lowest frequencies of cosine function and even after zig-zag scan we cannot assume that all coefficients are in descending order, therefore it might be some frequent data with long codewords assigned; two general schemes of sorting were described in this section, parallel sorting networks are very efficient in terms of the number of operations and are useful when data arrives in a parallel fashion but their material complexity is just to high; as we should expect serial sorting algorithms are suitable for serial input data schemes, even though the material complexity is moderate, the control logic required to operate this kind of architectures is often complicated as the entire array must be dynamically rearranged when a new value is inserted.

Finally, Motion Estimation architectures are surveyed, there are many families of algorithms to estimate the motion vector but the suitable architectures use either SAD, MAD or MAE algorithms configured as systolic arrays or linear arrays to compute the distortion of an image between adjacent frames; this process is the most computationally demanding in the entire compression chain, not even the DCT or the Entropy Coder require as much operations as the motion estimator, so in order to achieve the required throughput for real-time applications we must find a way to accelerate the prediction process without compromising data integrity.

# Chapter 4

# Architectural Implementations

Algorithm-architecture adequation is defined as the process of translating computational algorithms into hardware; we must take into account that algorithm-architecture adequations always lead to a sub-optimal architecture as there might be some portions of the algorithm that cannot be translated into hardware. As we mentioned before software solutions are efficient in terms of throughput, but they are not efficient in terms of the material complexity required to perform a given task; VLSI architectures represent a better choice than software implementations because they are customly made to work with an specific wordlength and perform specialized functions, also, they do not require an operating system to process data, also memory requirements are drastically reduced and the maximum operating frequency is by far, higher than any computational system.

## 4.1 Discrete Cosine Transform

### 4.1.1 Fast Algorithm

The first DCT implementation was based on Loeffler's Fast Algorithm [35], to build the architecture we need to identify the arithmetic operations required to code them as a separate module; when the modules are coded then we must identify data dependencies to instantiate the blocks in the position where they are required. Figure 3.6 clearly shows that four processing stages are required to perform a 1D-DCT, in every stage eight adder-substractor modules are required to perform the butterfly array operations, in stages two and three Cordic cores are required to perform the required rotations. In stage four of the fast algorithm we need a multiplicator block to calculate the product between incoming data and the constant term $\sqrt{2}$, figure 4.1 shows the implemented architecture.

Each processing stage was coded separately as a single module, once each module was coded we coded the top level entity where all the modules were instantiated, this coding style makes easier the process of building a system and allows a high degree of flexibility, any block can be reconfigured or re-synthesized the necessary times without affecting the entire system.

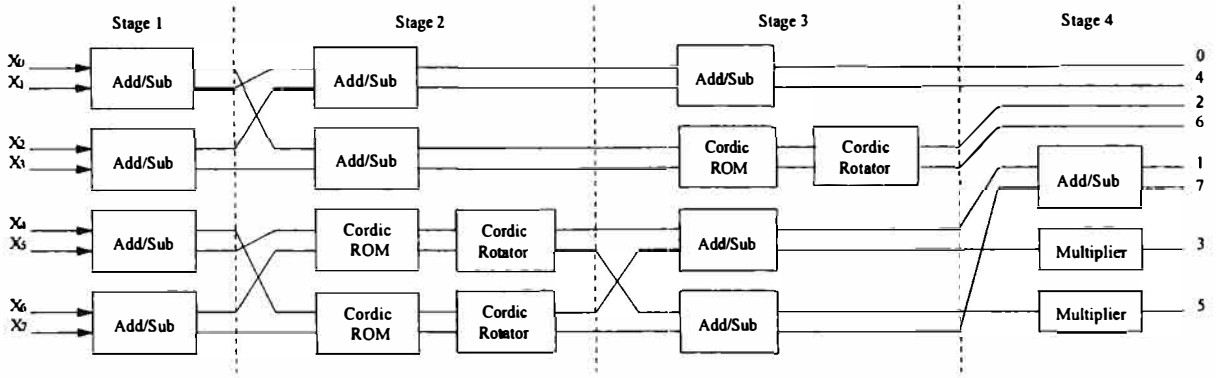



Figure 4.1: 1-D DCT architecture

Stages two and three require cordic rotators to compute the equation set 4.1 using only adding and shift operations, in this implementation we precomputed $I_0/K$ and $I_1/K$ and stored the results in a 512-word ROM, we use the incoming value $I$ as the address input of the ROM that simply outputs the corresponding $I/K$ to avoid the use of an arithmetic divider.

$$O_0 = I_0 \cdot k \cdot \cos\frac{n\pi}{2N} + I_1 \cdot k \cdot \sin\frac{n\pi}{2N} \tag{4.1}$$
$$O_1 = -I_0 \cdot k \cdot \sin\frac{n\pi}{2N} + I_1 \cdot k \cdot \cos\frac{n\pi}{2N}$$

In stage four we need to multiply two coefficients, so a hardware multiplier is required to perform the operation, the most efficient multiplication scheme is the Booth algorithm, the main issue is that the multiplier requires successive stages of fast adders, so, the material complexity increases;, another choice arises if we are not concerned with coefficient accuracy, as we know, DCT is a lossy compression process, so when the inverse process is performed we will not recover the original values under any circumstance, so, as the multiplying factor is $\sqrt{2} \approx 1.4142$ we could round the factor to 1.5, so, to implement the multiplication we could simply add the incoming value with a 1-bit right-shifted version of itself. Multiplicator architecture is explained in detail in section 4.1.1.1 and Cordic architectures are discussed in section 4.1.1.2.

#### 4.1.1.1 Multiplicator

The multiplication process involves two basic operations: partial product generation and accumulation, hence there are two methods to optimize the multiplicator, the first one consists in accelerate the additions, the second aims to reduce the number of partial products. To combine both schemes in an efficient architecture we can use Carry Save Adders (CSA) to accelerate the additions and Booth's algorithm to halve the number of partial products to

be added.

Binary multiplication is performed in the same way that we are used to multiplying two decimal numbers, so, if we consider operands $A$ and $B$, the number of partial products required to multiply $A \times B$ is equal to the number of bits required to represent $B$; Booth's algorithm proposes the recodification of the $B$ operand to a reduced number of coefficients [94], if $B$ has an even number of bits and $P$ is the weight of the MSB, the algorithm eliminates the odd powers of 2 considering that

$$2^i = 2 \cdot 2^i - 2^i = 2^{i+1} - 2 \cdot 2^{i-1}$$

To calculate the coefficients we must consider the ajdacent bits of the even powers to form a 3-bit word, then we shall find the correspondig coefficient in a fixed look up table.

| $b_{i+1}$ | $b_i$ | $b_{i-1}$ | C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 2 |
| 1 | 0 | 0 | -2 |
| 1 | 0 | 1 | -1 |
| 1 | 1 | 0 | -1 |
| 1 | 1 | 1 | 0 |

Table 4.1: Booth Coefficients

From table 4.1 we can observe that there are four possible coefficients $-2, -1, 0, 1, 2$ this means that the operand $A$ must be multiplied by the corresponding coefficient and then be added to the other partial products, the great advantage of this algorithm is that multiplications becomes a simple set of shift operations; to implement a Booth Coder in hardware we require a 2's complement module, a coefficient calculator, a partial product generator, four Carry Save Adders and a Vector Merging Adder (VMA) as shown in figure 4.2.

As the coefficient calculator and partial product generator are combinational circuits, they are both implemented along a 2's complement calculator in a single module named Booth Coder; CSAs are coded using Full Adders, the main difference between a conventional adder and a CSA is that in the latter Sum and Carry-out vectors are kept separated; for this implementation, the VMA was coded as a simple Ripple Carry Adder.

Code 4.1 shows the easiest way to code Booth LUT using a with-select statement, $g$ is a three bit vector that contains $\{b_{i+1}, b_i, b_{i-1}\}$; in any Booth multiplicator there are $\frac{N}{2}$ partial products, this implies that for a fixed wordlength of input operands, $g$ vectors are directly wired in an specific order, Code 4.2 shows how the partial products are generated depending on the power of 2 associated with the coefficient

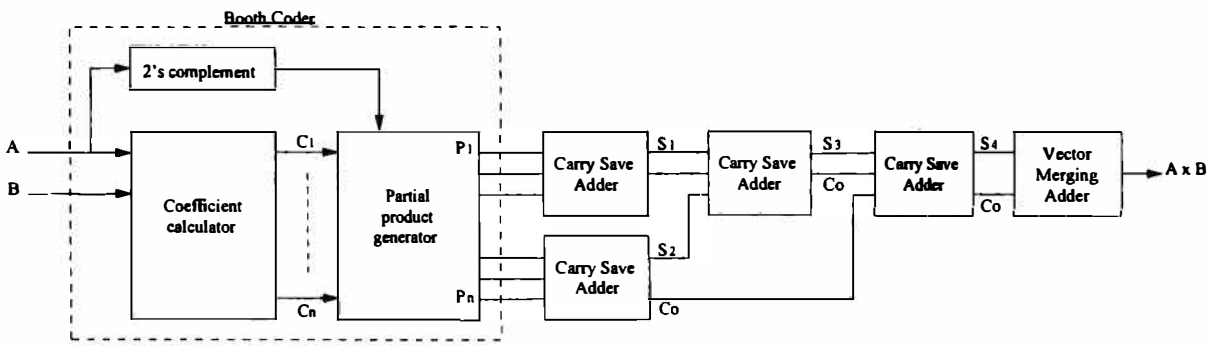



Figure 4.2: Booth Multiplier Architecture

```
with g select
 coef <= "001" when "001",
         "001" when "010",
         "010" when "011",
         "110" when "100",
         "111" when "101",
         "111" when "110",
         "000" when others;
```

Code 4.1: Booth Coding

```
with coef select
 pp1 <= A when "001", -- 1A*2^1
        A(N-2 downto 0)&'0' when "010", -- 2A*2^1
        Acomp(N-2 downto 0)&'0' when "110", -- -2A*2^1
        Acomp when "111", -- -1A*2^1
        X"000000" when others;
```

Code 4.2: Partial product generation

The next step in the design is to accelerate the architecture by reducing the critical path; pipeline technique is suitable for this architecture as the critical path will be the longest delay between pipeline barriers; Pipelines are easily introduced using a process statement that depends on Clock and Reset control signals as shown in code 4.3, in Chapter 5 synthesis results of Booth Multiplicator are presented.

```
process(CLK,RST)
 begin
  if RST='0' then
   Af <= x"00";
   Bf <= x"00";
   MultAB <= x"00000";
 elsif CLK'event and CLK='1' then
   --Barrier located at input port
   Af <= A;
   Bf <= B;
   --Barrier located at output port
   MultAB <= MultABp;
 end if;
end process;
```

Code 4.3: Pipelining in VHDL

#### 4.1.1.2 Cordic Algorithm

The Coordinate Rotation Digit Computer (CORDIC) was first introduced by Volder in [95], this is an iterative arithmetic algorithm for evaluating elementary functions and is especially suited for trigonometric functions [96]; this algorithm is commonly used when no hardware multipliers are available as the only arithmetic operations required are additions, bit shifts and table lookup.

Three basic ideas are behind the Cordic algorithm, first of all we want to embed elementary functions as a generalized rotation operation, the next idea is to decompose rotation operations into a series of successive micro-rotations, finally, each rotation must be realized with hard-wired shift operations and additions.

Two operation modes are allowed in the Cordic algorithm: Rotation and Vectoring, in the former, an input vector is rotated by an specific angle, in the latter the input vector is rotated to the $x$ axis. Ercegovac in [97] proposes two different architectures for a Cordic Core, a word serial architecture and a serial one; for the purpouse of this thesis we implemented a serial architecture for the Cordic Core because pipelining can be introduced to increase the performance, the algorithm that must be translated into hardware is shown in equation set 4.2, this algorithm requires three adder-substractors, one multiplexer to select the sign of $\sigma_{i+1}$ and a four-bit counter to increase the iteration number by one, this is required to determine the shift amount in the next iteration; processor outline is shown in figure 4.3.

$$\begin{aligned} x[i+1] &= x[i] - \sigma_i 2^{-i} y[i] \\ y[i+1] &= y[i] + \sigma_i 2^{-i} x[i] \\ z[i+1] &= z[i] - \underbrace{\sigma_i \tan^{-1} 2^{-i}}_{\alpha_i} \end{aligned} \tag{4.2}$$

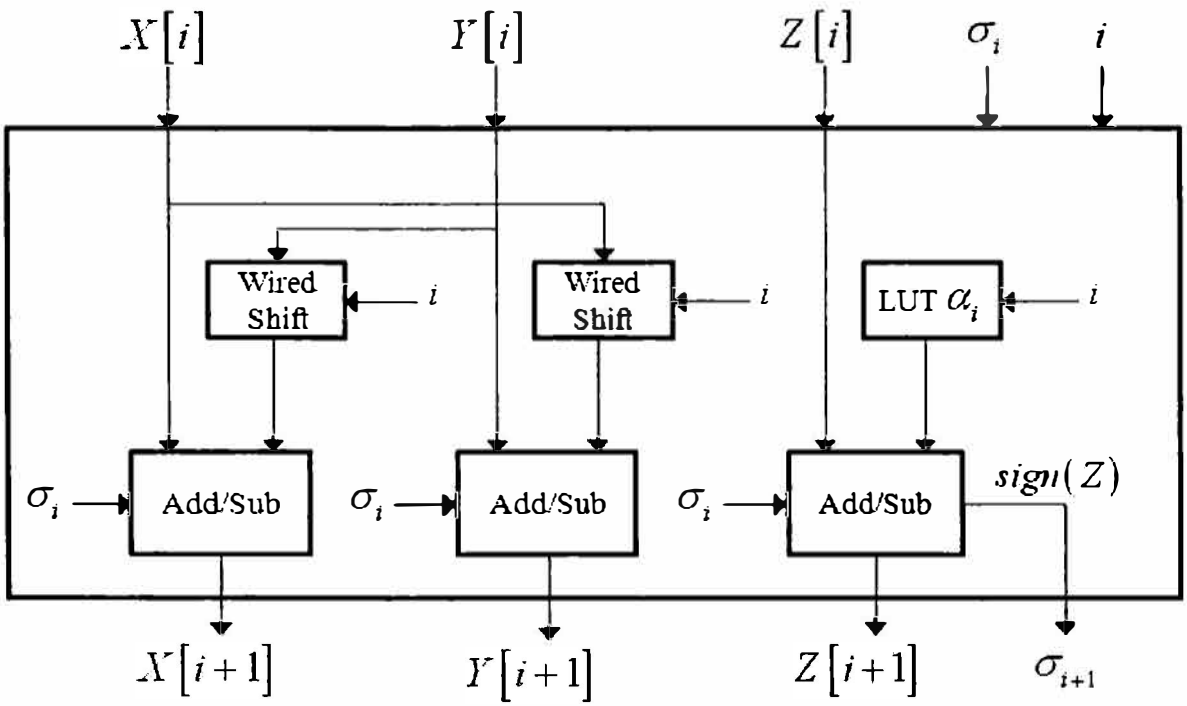



Figure 4.3: Cordic Serial Architecture

The LUT identified as $\alpha_i$ contains the result of $\sigma_i \tan^{-1} 2^{-i}$ where $i$ represents the number of iterations, $\sigma_{i+1}$ is determined by the sign of $Y$ or $Z$ depending on the operation mode, for Rotation mode, the most significan bit of $Z$ determines the sign of $\sigma_i$, for Vectoring mode the sign is determined by the most significant bit of $Y$.

A scaling factor $K$ must be applied in order to get the correct results of the rotation, this factor is constant, independent of the angle being rotated and the number of iterations; in the general, to compute $a\cos\theta - b\sin\theta$ and $a\sin\theta + b\cos\theta$ the scaling factor is introduced directly by setting $x[0] = \frac{a}{K}$ and $y[0] = \frac{b}{K}$.

The proposed Cordic Core can be easily pipelined to increase throughput an reduce the time propagation delay; we must select the number of iterations to approximate the results of the rotation, we decided to implement six iterations as the preliminary tests showed that

the error is less than $2^{-7}$; figure 4.4 shows the general architecture of the Cordic processor based on micro-rotations, notice that pipelining can be easily introduced to accelerate the calculations as shown previously in code 4.3.

For the purpouse of analyzing the effect of the pipeline barriers in the critical path reduction we implemented the architecture using three pipelining configurations, the first one uses two pipeline barriers, one located between the processor's input data and the first cordic core and the other located between the output of the last Cordic Core and the processor's output; the second configuration utilizes three pipeline barriers, the limiting barriers described above and a third barrier located between the third and fourth Cordic Core; finally a pipeline barrier is placed between every core, synthesis results are presented in Chapter 5.

$$K \doteq \prod_{i=0}^{\infty}(1+2^{-2i})^{1/2} \approx 1.6468 \tag{4.3}$$

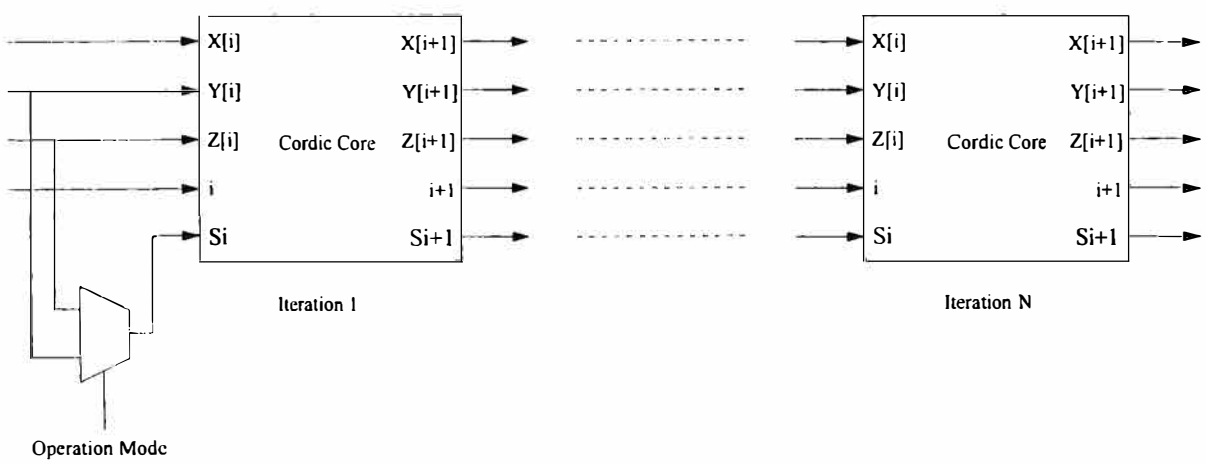



Figure 4.4: Pipelined Cordic Architecture

### 4.1.2 Distributed Arithmetic

This implementation shares the first stage of the Fast Algorithm, in order to calculate the DCT we must rearrange incoming data, into even and odd coefficients, after the butterfly stage is performed, RAC architectures are used to add and accumulate the partial results pre-stored in ROM devices, the architecture is shown in figure 3.12, RAC module is divided into a ROM device and a dedicated Shift-Accumulate unit that consists of four blocks as shown in figure 4.5.

We require a Finite State Machine to control the shift-accumulate operations and the read/write cycles of the associated ROM, it is coded as a simple Mealy State Machine to

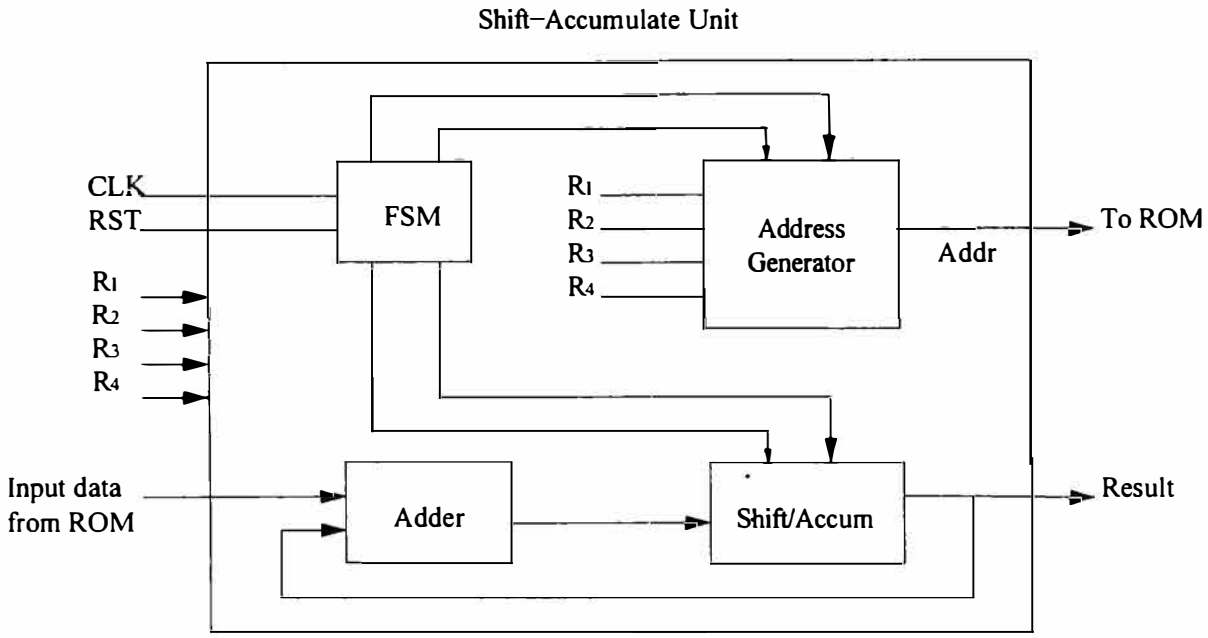



Figure 4.5: Shift-Accumulator Unit

control the described outputs, two of them are the read and write enables or the ROM, the other two outputs correspond to the shift enable and the load instruction of the address generator; figure 4.6 shows the state diagram of the FSM.

Eight ROMs have to be defined as the Distributed Arithmetic technique requires to have precomputed results of products, the DCT-II angles for even and odd coefficients are shown in matrices 4.4 and 4.5 respectively.

$$\alpha_{even} = \begin{bmatrix} \cos 4\theta & \cos 4\theta & \cos 4\theta & \cos 4\theta \\ \cos 2\theta & \cos 6\theta & -\cos 6\theta & -\cos 2\theta \\ \cos 4\theta & -\cos 4\theta & -\cos 4\theta & \cos 4\theta \\ \cos 6\theta & -\cos 2\theta & \cos 2\theta & -\cos 6\theta \end{bmatrix} \tag{4.4}$$

$$\alpha_{odd} = \begin{bmatrix} \cos\theta & \cos 3\theta & \cos 5\theta & \cos 7\theta \\ \cos 3\theta & -\cos 7\theta & -\cos\theta & -\cos 5\theta \\ \cos 5\theta & -\cos\theta & \cos 7\theta & \cos 3\theta \\ \cos 7\theta & -\cos 5\theta & \cos 3\theta & -\cos\theta \end{bmatrix} \tag{4.5}$$

Each row represents of the matrices represent the coefficients $c_0, c_1, c_2$ and $c_3$ of the Distributed Arithmetic lookup table, bits $x_{0,j}, x_{1,j}, x_{2,j}$ and $x_{3,j}$ correspond to the least significant bit of each input vector $R_1, R_2, R_3, R_4$ at $j$ cycle; the Address Generator module takes the LSB of each vector and merges them into a single 4-bit word used to address the lookup table, every clock cycle the registers are right shifted to lookup for the corresponding value

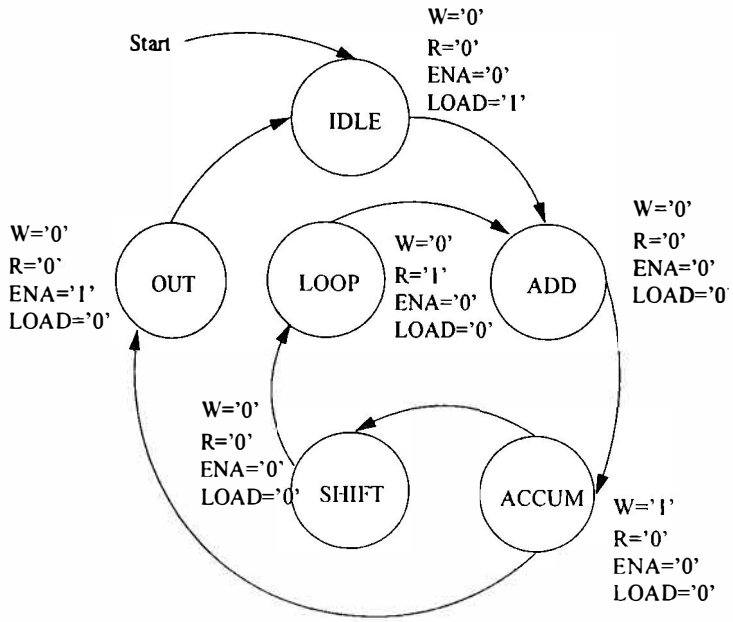



Figure 4.6: DCT- DA Finite State Machine

in the $2^{j+1}$ weight; if we do not want to use memory devices as the data fetch cycle represents a bottleneck we could code the lookup table as an array, the only difference is that we will require dedicated registers, figure 4.7 shows the RAC architecture for conventional Distributed Arithmetic.

Distributed Arithmetic with Offset Binary Coding is used to reduce memory requirements, this implementation is very similar to conventional DA, in fact, the butterfly stage and the finite state machine are the same, the only difference relies in the memory reduction by half, two additional multiplexers and three exclusive-or gates, figure 4.8, XOR gates can be transferred to the ROM module as they are only required to address the input data as shown in code 4.4.

Once the 1D-DCT is described in hardware we have two choices for performing the 2D-DCT, first option is to transpose 1D-DCT result coefficients and re-enter data to the same 1D-DCT core as shown in figure 4.9, even though this solution is efficient in terms of material complexity, throughput is affected as the operations are performed in a serial fashion; the second option is to use two 1D-DCT cores, after the first DCT is calculated and the coefficients are transposed, a second DCT unit performs the remaining DCT, this operating structure allows the first DCT core to process the next block while the second is still processing the first one as shown in figure 4.10; both figures show the required wordlengths for the architectures, notice that after performing the first 1D-DCT two wordlengths are indicated, 12-bit wordlength corresponds to fast algorithm architecture; 16-bit wordlength corresponds to distributed arithmetic algorithms, this implies that we must truncate the 4 least signifi-

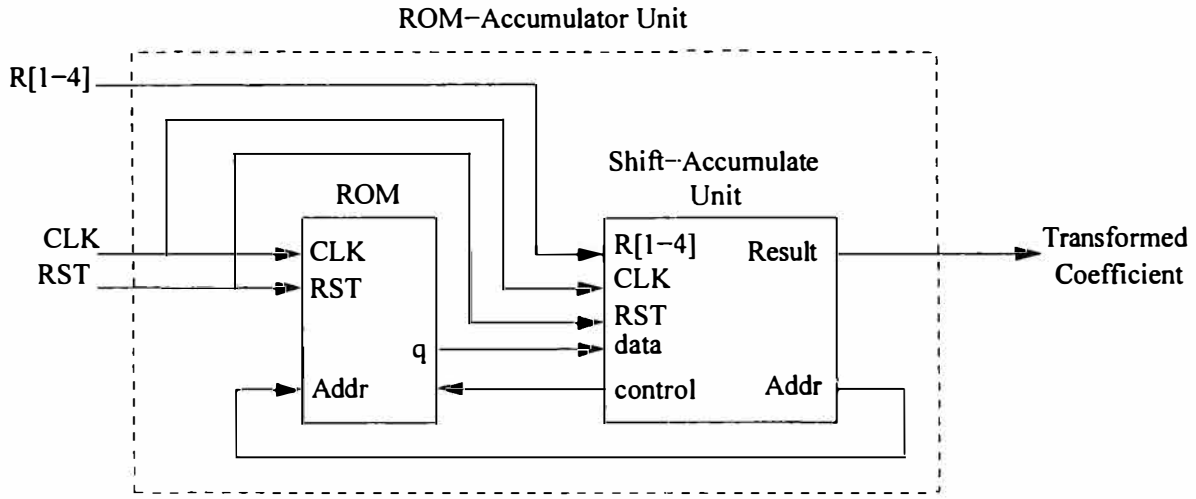



Figure 4.7: ROM-Accumulate Architecture

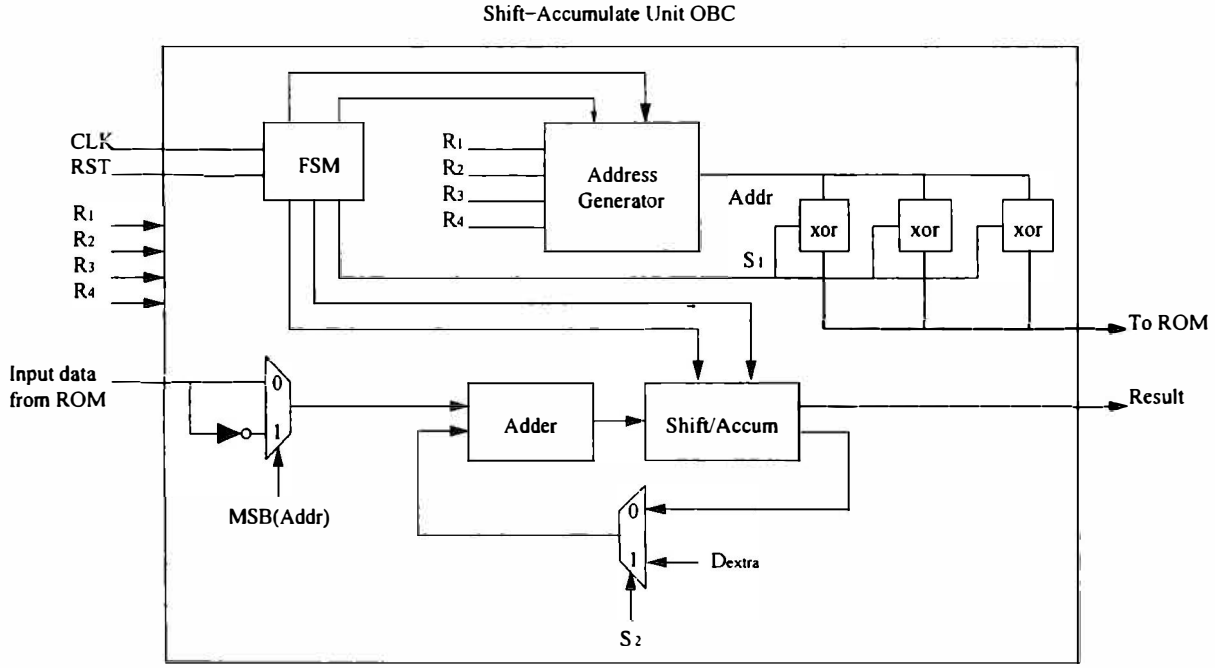



Figure 4.8: Shift-Accumulator unit for OBC

```
type arr is array(0 to 7) of std_logic_vector(15 downto 0);
constant MEM : arr:=(X"D2BF",X"E95F",X"E95F",X"0000",
                     X"E95F",X"0000",X"0000",X"16A1");
signal addrf : std_logic_vector(3 downto 0);
signal qd    : std_logic_vector(15 downto 0);
signal sel   : std_logic;
signal dirm  : std_logic_vector(2 downto 0);

begin
 -- logic gates required to address the memory
 sel <= addrf(3) and stop;
 dirm(2) <= addrf(2) xor sel;
 dirm(1) <= addrf(1) xor sel;
 dirm(0) <= addrf(0) xor sel;

 with dirm select
  qd <= MEM(0) when "000",
        MEM(1) when "001",
        MEM(2) when "010",
        MEM(3) when "011",
        MEM(4) when "100",
        MEM(5) when "101",
        MEM(6) when "110",
        MEM(7) when "111";
```

Code 4.4: ROM module for Distributed Arithmetic with OBC

can bits as they are the decimal part of the coefficient and the accuracy loss is less than a unit.

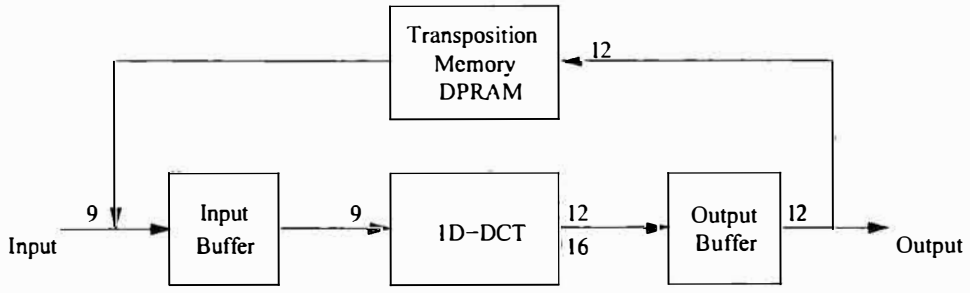



Figure 4.9: Row-Column Transformation architecture for 2D-DCT

### 4.1.3 Memory Transposition

As explained in chapter 3, memory transposition can be achieved using a Double Port Random Access Memory device, code 4.5 shows the easiest way to define a DPRAM. The operation of this module is simple, when the first coefficients of the DCT are calculated we must store them in memory, we use a 6-bit counter to address the memory locations; when the counter resets, we can start reading the memory to feed the next 1D-DCT stage; as shown before in figure 3.15 we must rearrange the counter output vector to generate the read address. The counter is easily coded using cascaded Half-Adders and an XOR gate.

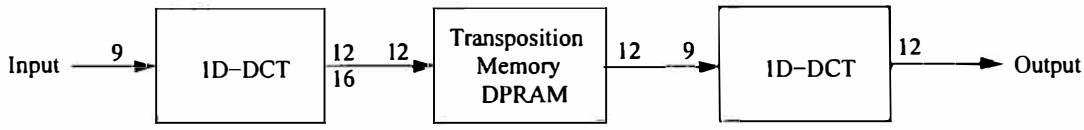



Figure 4.10: Two-core architecture of 2D-DCT

```
architecture rtl of DPRAM is
 -- 2-D array for the RAM
 type MEM is array(2**LONGITUD_DIR-1 downto 0) of
             std_logic_vector((LONGITUD_WORD-1) downto 0);
 -- RAM signal
 signal RAM : MEM;

 begin
  process(clk)
   begin
    if CLK'event and CLK='1' then
     RAM(Waddr) <= dato;
     -- if the same address is read and written at the same time
     -- the reading will be the last stored data of that location
     q <= RAM(Raddr);
    end if;
  end process;
end architecture;
```

Code 4.5: Dual-Port RAM

## 4.2 Entropy Coding

### 4.2.1 Sorting Algorithms

We decided to use sorting networks to generate the probability density function on the fly at the same time that Macroblock to Block conversion, DCT and Quantization processes are performed in order to determine the propper Huffman dictionary as we will know the exact frequency distribution of the symbols within a Macroblock.

The basic component of any sorting algorithm is the compare-exchange module, this entity consists of a comparator that indicates if inputs $a$ and $b$ are equal or if $a < b$, and two multiplexers that determine the outputs $S0$ and $S1$ depending on the values inserted to the module, figure 4.11 shows the configuration of this module, notice that output selection depends on $aeb$, $a < b$, and an input signal identified as $A/D$, this signal indicates whether the compare-exchange module will work in ascending or descending way, we decided to code the CE with the capability of sorting either ascendingly or descendingly because there is practically no difference (two logic elements) between the implemented CE without a selector and the CE with a selector, this gives us the flexibility to chose the sorting mode for any given list without having to modify the architecture. Code 4.6 shows how the output selection is done, notice that the output signals of the comparator are concatenated with the selection bit, then we use this 3-bit signal as the output selector of the multiplexers.

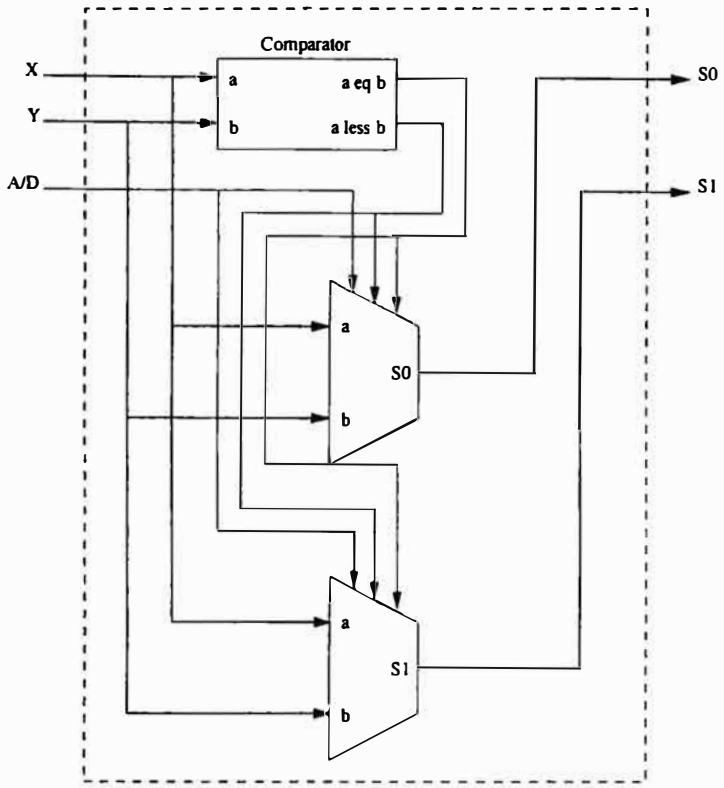



Figure 4.11: Compare-Exchange module

```
outsel <= AD & XlY & XeY;
 I0 : Comparador port map(X,Y,XeY,XlY);

  with outsel select
   S0 <= X when "001" | "010" | "100" | "101", Y when others;

  with outsel select
   S1 <= X when "000" | "001" | "101" | "110", Y when others;
```

Code 4.6: Compare-Exchange output selection

As the compare-exchange element is the smallest sorting network we must interconnect several CEs to sort data arrays or lists, so it is necessary to instantiate the component, code 4.7 must be inserted within the architecture **of** any sorting network. Previously in Chapter 3 the difference between a merging network (MN) and a sorting network (SN) was explained, in order to build a $N$ size sorting network we will require one $N$-item merging network, two $\frac{N}{2}$-item networks, four $\frac{N}{4}$-item networks and so on until we reach the 4-item networks, then, to complete the design of the **sorting network** we have to include $\frac{N}{2}$ CE elements before the 4-item merging networks.

```
component CompExch is
port(
 X,Y : in  std_logic_vector(8 downto 0);
 AD  : in  std_logic;
 S0,S1 : out std_logic_vector(8 downto 0));
end component;
```

Code 4.7: Compare-Exchange module instantiation

We must remember that the only restriction to use a merging network is that the input lists must be sorted, for an 8-item merging network we require two 4-item merging networks and three additional compare exchange elements at the end to sort the output list as shown in figure 4.12, in general a N-item merging network is constructed using two $\frac{N}{2}$-item merging networks and $\frac{N}{2} - 1$ CEs, making this an iterative process.

As a 64-item sorting network is required in our architecture we need to build the merging networks for 4, 8, 16, 32 and 64 items, the great advantage is that they can be coded individually and instantiated when required; this architectures are big in terms of material complexity as they require many CE modules to construct a merging network, huge amounts of signals are required to interconnect each block, meaning that the design is also large in terms of interconnection paths, in exchange it offers a highly regular structure and practically no control logic is required to sort the array.

Code 4.8 shows the instantiation required to build an 8-item MN, as expected, two 4-item merging networks are used as a basis of the new network, instances $I0$ and $I1$ define input and output connections of both 4-item MNs, signals labeled as $C_n$ define the required wires to connect the outputs of the previous MN to the $\frac{N}{2} - 1$ CEs used to rearrange the network's output; the same process must be followed to build larger networks.

A 64-item sorting network was constructed using sixteen 4-item MNs, eight 8-item MNs, four 16-item MNs, two 32-item MNs and one 64-item MN; this architecture is very large (+19K LE), code 4.9 shows the instantiation of all the components required to build a 64-item sorting network, $L_n$ represent the input list of the sorting network, $E_n$ represent the sorted network, notice the regularity of the implementation and the large amount of signals required to wire the necessary blocks.

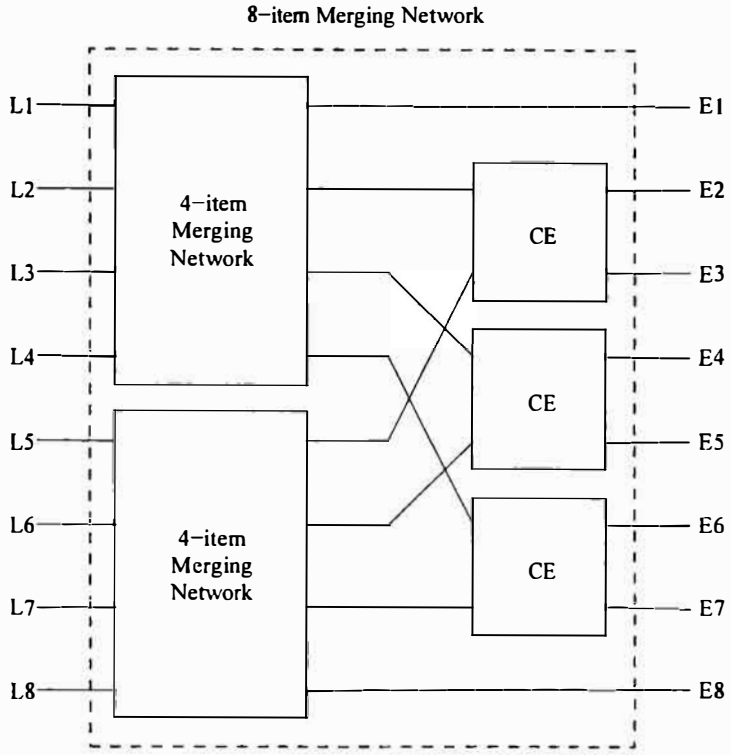



Figure 4.12: 8-item Merging Network Architecture

```
architecture func of Merge8 is
 component CompExch is
  port( X,Y    : in  std_logic_vector(8 downto 0);
        AD     : in  std_logic; -- 0 = Ascending, 1 = Descending
        S0,S1  : out std_logic_vector(8 downto 0)); -- Outputs S0 and S1 are exclusive
 end component;

 component Merge4 is
  port( L1,L2,L3,L4 : in  std_logic_vector(8 downto 0); -- L = input line
        sel         : in  std_logic;
        E1,E2,E3,E4 : out std_logic_vector(8 downto 0));-- E = sorted element
 end component;

 begin
  I0: Merge4 port map(L1,L2,L3,L4,sel,E1,C1,C2,C3);
  I1: Merge4 port map(L5,L6,L7,L8,sel,C4,C5,C6,E8);
  I2: CompExch port map(C1,C4,sel,E2,E3);
  I3: CompExch port map(C2,C5,sel,E4,E5);
  I4: CompExch port map(C3,C6,sel,E6,E7);
end architecture;
```

Code 4.8: 8-item Merging Network coding

After synthesizing all sorting networks for area optimization the propagation delays were measured and analized; we determined that a hardware acceleration technique must be employed to reduce the critical path, pipelining is the obvious choice because parallelism requires large silicon areas; synthesis results for area and speed optimization are presented in Chapter 5.

### 4.2.2 Huffman Coding

As mentioned in chapter 3, there are many architectures for Huffman coding, and the main drawback is that all of them are too large and complex control logic is required to assign the symbols to their corresponding codewords. Static and Adaptive Huffman methods have been discussed and both of them are based on particular facts; static Huffman Coding requires a prior knowledge of the data probability density function, on the other hand, adaptive Huffman Coding requires a dynamic construction of the frequency table, this requirement tends to reduce coding efficiency.

A memory based architecture can be used to dynamically calculate the probability density function of serial input data on the fly; we require two random access memories to construct the frequency table, one hardwired adder block named "+1" that simply adds one unit to any number using only half-adders; a sorting network is required to order in a descendent fashion all the symbols, after the sorting process is performed, a comparison layer and an 8-bit adder are used to determine the exact amount of symbols; depending on the adder result we select the predefined Huffman dictionary that fits the number of symbols.

Both RAM devices are initialized with zeros in all locations, input data is used to address the memories to output the number in that specific location and sends it to the +1 block, once the number has been incremented by one it is stored in the same address of the second RAM, when a new reading cycle begins we update the first RAM, to continue the process.

If we do not want to use memory devices to construct the frequency table we can use sorting networks; when the sorting process is completed we are certain that data is ordered, but we do not know how many symbols are included in the array, perhaps the array has only one symbol, or it might have 256 different symbols; we implemented a compare element that indicates if the inputs A and B are equal; figure 4.14 shows the method to calculate the number of symbols, two signals are required: $aeb$ already present in the Compare-Exchange elements, and $AeB$, generated in the comparator, truth table 4.2 indicates the number that must be accumulated in the adder.

Two special cases are identified: as the first element must be counted anyways, we can initialize the accumulator in 1; for the last element of the network we have to instantiate both inputs of the comparator with the last number of the list to ensure the element counting. If

```
architecture func of Sort64 is
 begin
 I0 : CompExch port map(L1,L2,sel,M1,M2);

 I31 : CompExch port map(L63,L64,sel,M63,M64);

 I32 : Merge4 port map(M1,M3,M2,M4,sel,N1,N2,N3,N4);

 I47 : Merge4 port map(M61,M62,M63,M64,sel,N61,N62,N63,N64);

 I48 : Merge8 port map(N1,N5,N3,N7,N2,N6,N4,N8,sel,P1,P2,P3,P4,P5,P6,P7,P8);

 I55 : Merge8 port map(N57,N58,N59,N60,N61,N62,N63,N64,sel,P57,P58,P59,P60,P61,P62,P63,P64);

 I56 : Merge16 port map(P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13,P14,P15,P16,sel,
       Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10,Q11,Q12,Q13,Q14,Q15,Q16);

 I59 : Merge16 port map(P49,P50,P51,P52,P53,P54,P55,P56,P57,P58,P59,P60,P61,P62,P63,P64,sel,
       Q49,Q50,Q51,Q52,Q53,Q54,Q55,Q56,Q57,Q58,Q59,Q60,Q61,Q62,Q63,Q64);

 I60 : Merge32 port map(Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10,Q11,Q12,Q13,Q14,Q15,Q16,
       Q17,Q18,Q19,Q20,Q21,Q22,Q23,Q24,Q25,Q26,Q27,Q28,Q29,Q30,Q31,Q32,sel,
       A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,
       A17,A18,A19,A20,A21,A22,A23,A24,A25,A26,A27,A28,A29,A30,A31,A32);
 I61 : Merge32 port map(Q33,Q34,Q35,Q36,Q37,Q38,Q39,Q40,Q41,Q42,Q43,Q44,Q45,Q46,Q47,Q48,
       Q49,Q50,Q51,Q52,Q53,Q54,Q55,Q56,Q57,Q58,Q59,Q60,Q61,Q62,Q63,Q64,sel,
       B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15,B16,
       B17,B18,B19,B20,B21,B22,B23,B24,B25,B26,B27,B28,B29,B30,B31,B32);

 I62 : Merge64 port map(A1,B1,A3,B3,A5,B5,A7,B7,A9,B9,A11,B11,A13,B13,A15,B15,
       A17,B17,A19,B19,A21,B21,A23,B23,A25,B25,A27,B27,A29,B29,A31,B31,
       A2,B2,A4,B4,A6,B6,A8,B8,A10,B10,A12,B12,A14,B14,A16,B16,
       A18,B18,A20,B20,A22,B22,A24,B24,A26,B26,A28,B28,A30,B30,A32,B32,sel,
       E1,E2,E3,E4,E5,E6,E7,E8,E9,E10,E11,E12,E13,E14,E15,E16,
       E17,E18,E19,E20,E21,E22,E23,E24,E25,E26,E27,E28,E29,E30,E31,E32,
       E33,E34,E35,E36,E37,E38,E39,E40,E41,E42,E43,E44,E45,E46,E47,E48,
       E49,E50,E51,E52,E53,E54,E55,E56,E57,E58,E59,E60,E61,E62,E63,E64);
end architecture;
```

Code 4.9: 64-item Sorting Network instantiation

| aeb | AeB | Add |
|---|---|---|
| 0 | 0 | 2 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Table 4.2: Frequency Counting Adder

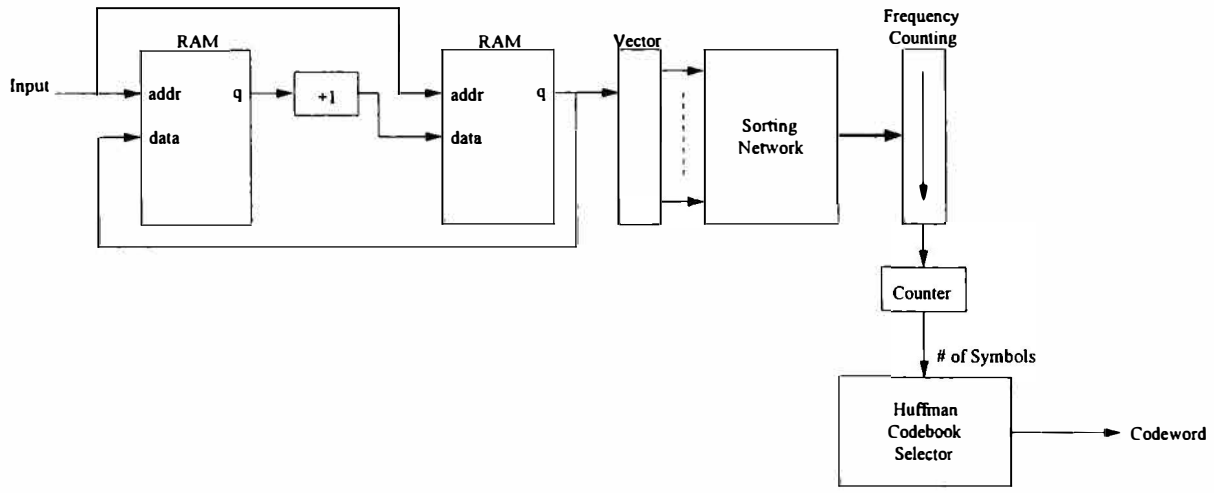



Figure 4.13: Huffman Coder proposal

we do not want to waste resources in a full 8-bit adder we can replace that module with a +1 block and a simple shift-left operation to perform the operations indicated in the table.

## 4.3 Coder Proposed Architecture

A generic encoder consists at least of five basic processes as shown in figure 4.15: Macroblock to Block (MB-B) conversion, 2D-DCT processor, Quantization, Zig-Zag Scanning and Entropy Coding. MB-B conversion, DCT and Quantizarion are performed in parallel fashion, Zig-zag scan transforms the $8 \times 8$ matrix into a $1 \times 64$ vector using a finite state machine; once the vector is obtained we must feed the Huffman coding process to perform the variable length coding.

The worst case scenario after performing the DCT is to have 64 different coefficients, this could happen if the block pixels are not correlated, if this happens we will require a large code book with long codewords; when the image has a pattern (pixels have a tight correlation with its neighbours) we can assume a direct relation between the number of symbols of the source image and the number of symbols of the DCT transformed image [98], this implies that when pixels within a block or macroblock are correlated we will require *approximately the same number of symbols before and after the DCT*. If we know the probability density function of the macroblock before compression, there is a high probability that the selected code book will be the smallest and will increase the coding efficiency for that specific macroblock.

The main idea of this architecture is to initiate the obtention of the macroblock's pdf at the same time than the macroblock to block conversion is performed; after the MB-B process, each block enters the DCT and then enters the quantizer; we have all this time to sort in

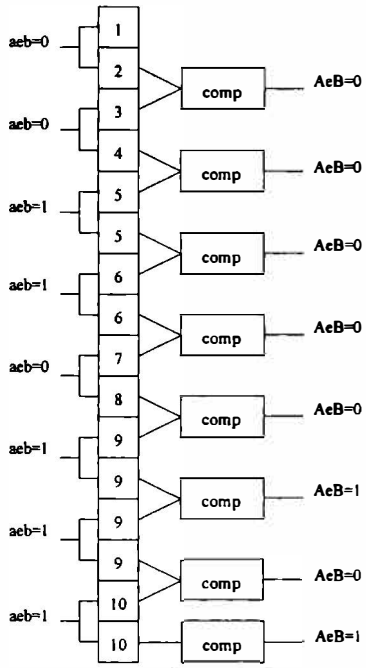



Figure 4.14: Frequency Counting

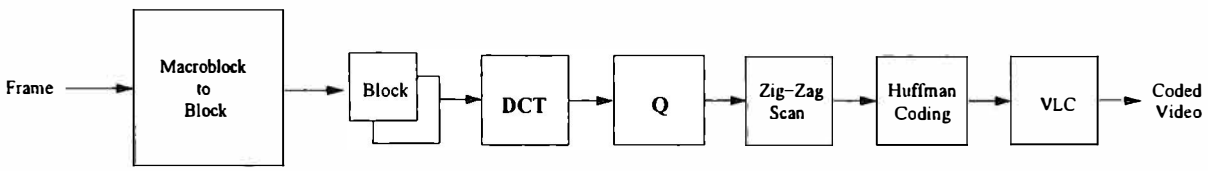



Figure 4.15: Video Compression System

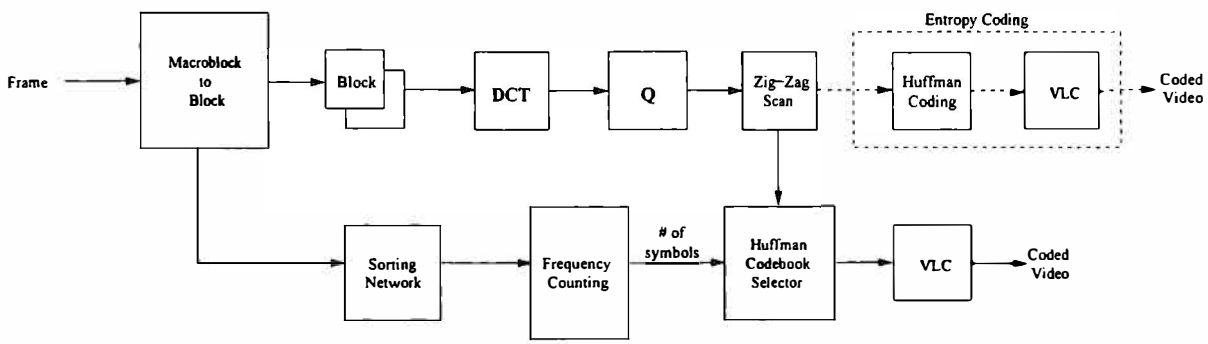



Figure 4.16: Proposed Video Coder

parallel all pixels and determine the number of symbols to select the Huffman Codebook. Figure 4.16 shows the proposed coding method vs the conventional method (dashed lines), notice that entropy coding process can be splitted into two processes: Huffman Coding and Variable Length Coding; conventional method starts the frequency table construction and assignation of codewords after the Quantization and Zig-Zag Scan; by parallelizing frequency table construction after the Q and Zig-Zag scan we simply assign the codewords. The block identified as *Huffman codebook selector* is used to route the corresponding LUT that contains the codewords depending on the number of symbols as depicted in figure 4.17.

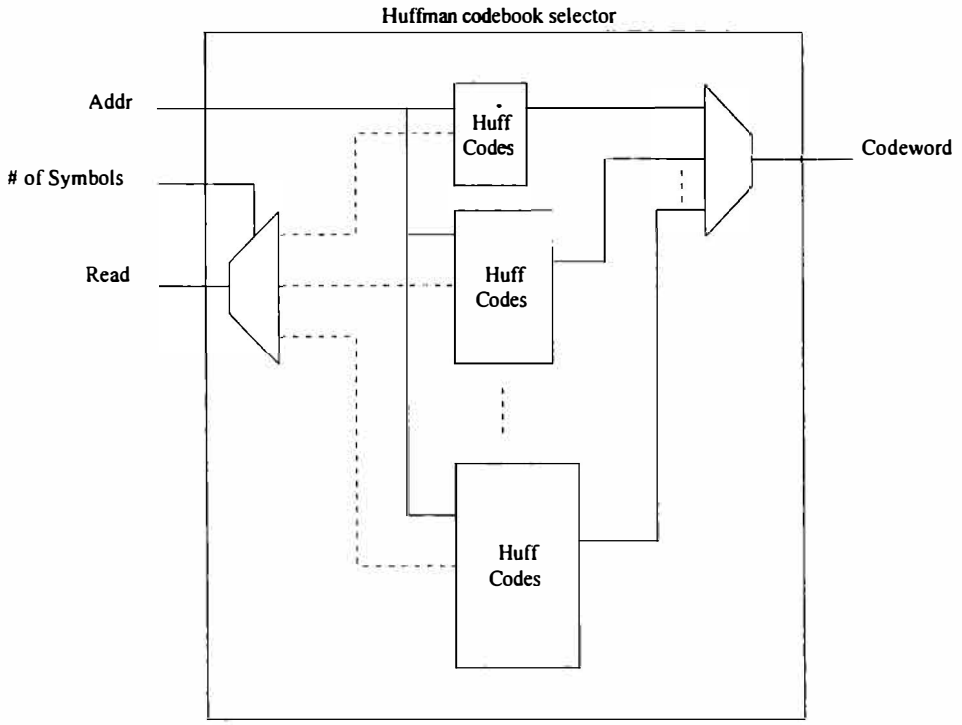



Figure 4.17: Huffman Codebook Selector

Another option derived from this proposal is to perform the same process for every block, the difference will be the size reduction of the sorting network from 256 to 64 elements; figure 4.18 shows that the same process must be followed to obtain the frequency table of the block. If we choose to implement this particular architecture we must be aware that the available clock cycles to perform the frequency count are now limited to the required cycles to process the DCT and the quantization.

Parallelizing highly demanding tasks as DCT and frequency table generation for entropy coding implies large silicon areas, in exchange, we expect lower processing times and high throughput rates.

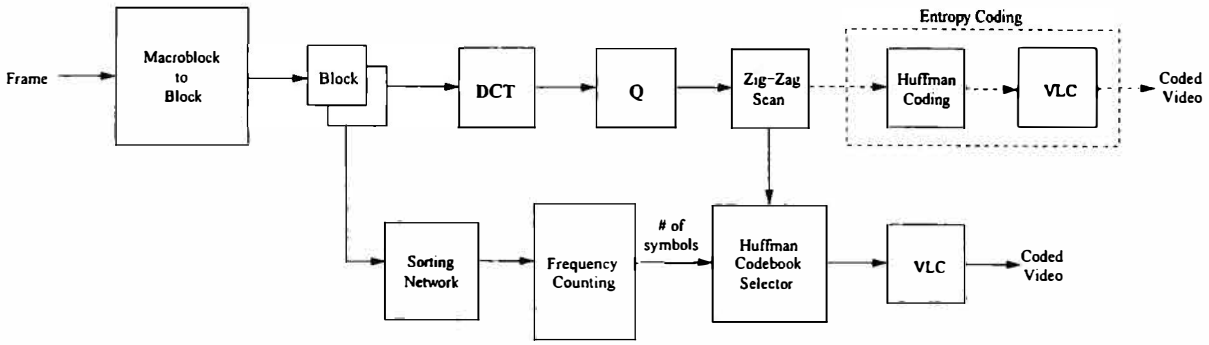



Figure 4.18: Sorting Network reduction for the proposed architecture

## 4.4 Conclusions

In this chapter we reported three different implementations of the Discrete Cosine Transform, first architecture was based on the fast algorithm proposed by Loeffler in 1989, the rotators required in stages two and three were replaced with six micro-rotation Cordic cores; second and third implementations are based on Distributed Arithmetic; conventional DA requires more memory resources but the material complexity is lower than DA with Offset Binary Coding because no additional multiplexers or gates are required to calculate the DCT.

Sorting and Merging networks for different list sizes were presented and coded as an alternative to dynamic Huffman coding; any real-life picture or frame has the characteristic of presenting smooth transitions between adjacent pixels, so after the DCT process we can assume that many coefficients will be eliminated and that the number of symbols within the DCT-transformed image will be similar to the amount of symbols employed in the original image. Calculating the probability density function on the fly allows us to select the smallest codewords for most frequent symbols as they come out of the zig-zag scan.

# Chapter 5

# Results

In this section, the synthesis results for some architectures reviewed in Chapter 4 are presented; some simulations and synthesized RTL modules are shown; all implementations were coded in VHDL and Quartus II software was used to synthezise all the modules; two synthesys techniques were studied: Area and Speed; as we might expect in many cases area optimized codes are slower than speed optimized ones, but in some others area optimized codes are faster than speed optimized architectures; this interesting fact is possible because area and speed constraints are complementary, so, if we synthesize for area, the maximum operating frequency might be diminished, but if the synthesized LEs are adjacent and the interconnection paths are short, this area optimized architecture could also have the highest operating frequency. Area and Speed synthesis is shown for every tested architecture.

## 5.1 Discrete Cosine Transform

As mentioned early in section 4.1, three different architectures to perform the DCT were implemented: Fast Algorithm, Distributed Arithmetic and Distributed Arithmetic with Offset Binary Coding; Fast algorithm implementation requires a Cordic Rotator module to replace the arithmetic operations of the original rotation with simple adders and shifters; at stage four of the algorithm a booth multiplier is required to perform $\sqrt{2} \cdot I_3$ and $\sqrt{2} \cdot I_5$, Distributed Arithmetic implementations require RAM modules to store precomputed results of the DCT coefficients matrix and a dedicated module used to shift and accumulate the coefficients until the final result is obtained.

### 5.1.1 Fast Algorithm

Prior to implementing the fast algorithm we implemented the cordic algorithm and a booth multiplier as they represent important design modules for the architecture; Synthesis results for Cordic Algorithm are shown in table 5.1, in the second column the reported time propagation delay is 76.885 ns, by means of pipelining we implemented seven barriers and we

were able to reduce the critical path so that the maximum operating frequency is 119.98 MHZ.

| Acceleration Technique | None | Pipeline |
|---|---|---|
| Logic Elements | 731 | 1201 |
| Combinational Functions | 731 | 1157 |
| Flip-Flops | 0 | 292 |
| Worst Case TPD | 76.885 $ns$ | - |
| Clock | - | 119.98 MHz |

Table 5.1: Cordic Synthesis Results

Table 5.2 shows the synthesis results of the implemented 1D-DCT based on Loeffler's algorithm replacing rotators for Cordic cores; notice that there is not much difference between the required logic elements and combinational function, the difference between synthesis for speed and for area is the maximum clock frequency; memory requirements are high because we need three modules of 512words x 13bits to store all the possible results of $I_0$and$I_1$ for the Cordic Rotators.

| Synthesized for | Speed | Area |
|---|---|---|
| Logic Elements | 3482 | 3105 |
| Combinational Functions | 3413 | 3037 |
| Flip-Flops | 1015 | 1015 |
| Memory bits | 19968 | 19968 |
| Clock | 107.92 MHz | 96.05 MHz |
| Period | 9.266 $ns$ | 10.411 $ns$ |

Table 5.2: 1D-DCT Fast Algorithm Synthesis Results

Five pipeline barriers were implemented to accelerate the architecture, two barriers are located at input and output ports, the three remaining barriers are placed between each processing stage, figure 5.1 shows the synthesized architecture, flip-flop barriers are clearly identified between the stages.

### 5.1.2 Distributed Arithmetic

Shift-Accumulator Unit for Distributed Arithmetic was synthesized to determine the maximum operating frequency of the module before instantiating it in the DCT-DA design, table 5.3 shows that the module is small and fast; flip-flops are required to operate the State Machine. To complete the design of a Distributed Arithmetic Architecture we must include

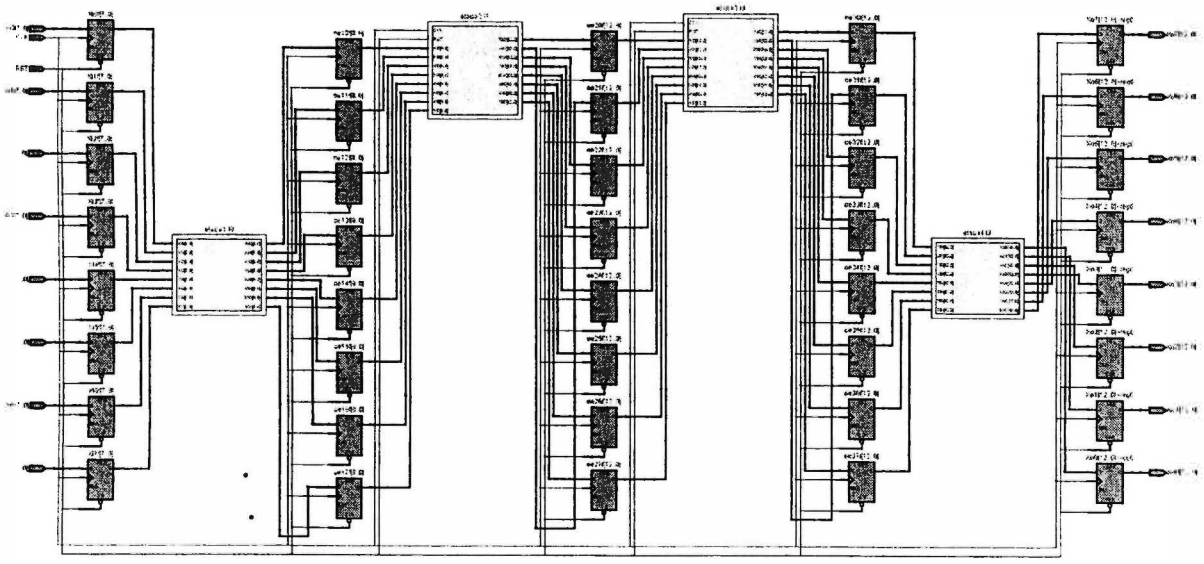


Figure 5.1: RTL view of Fast Algorithm

RAMs and the butterfly stage; table 5.4 shows the synthesys results of the DCT-DA implementation, figure shows the RTL synthesis.

| Device | Cyclone II EP2C35F672C6 |
|---|---|
| Logic Elements | 108/33216 (< 1%) |
| Combinational Functions | 86/33216 (< 1%) |
| Flip-Flops | 100 |
| Pins | 37 |
| Memory bits | 0/483840 (0%) |
| Clock | 137.97 MHz |

Table 5.3: Shift-Accumulator Unit for DCT-DA

Distributed Arithmetic with Offset Binary Coding architecture is very similar to conventional DA, table 5.5 shows synthesis results for DCT with OBC and serves as a comparison between speed and area optimization. For this architecture, area optimization offers a slightly higher operating frequency than speed optimization, notice that LE difference is minimum. RTL view of Distributed Arithmetic DCT with OBC is very similar to figure 5.2, in fact, the only difference relies in two extra multiplexers in every Shift-Accumulator Block.

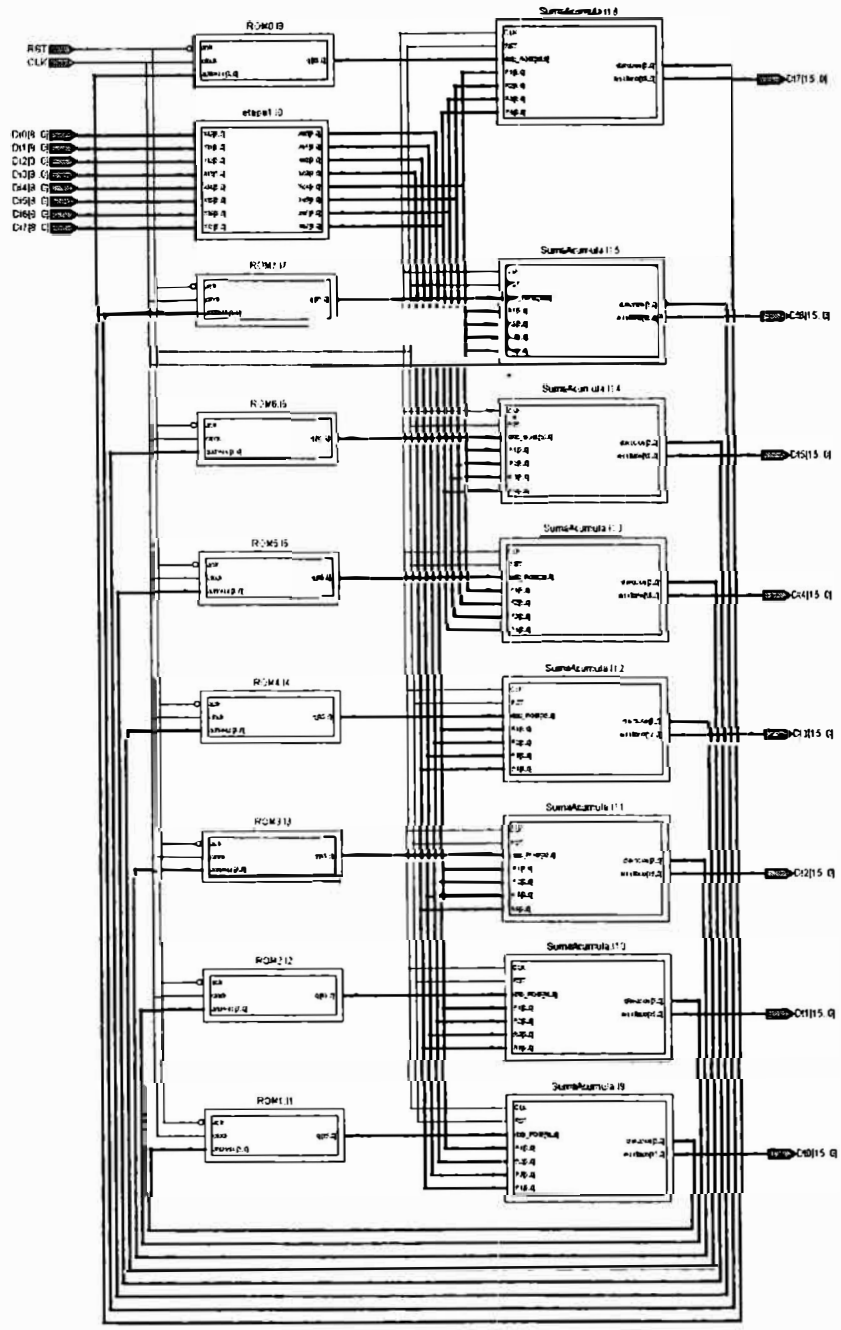


Figure 5.2: RTL view of Distributed Arithmetic 1D-DCT

| Synthesized for | Speed | Area |
|---|---|---|
| Logic Elements | 792 | 632 |
| Combinational Functions | 503 | 427 |
| Flip-Flops | 476 | 476 |
| Memory bits | 2048 | 2048 |
| Clock | 151.33 MHz | 115.54 MHz |
| Period | 8.655 $ns$ | 6.608 $ns$ |

Table 5.4: 1D-DCT with conventional Distributed Arithmetic

| Synthesized for | Speed | Area |
|---|---|---|
| Logic Elements | 977 | 935 |
| Combinational Functions | 659 | 638 |
| Flip-Flops | 564 | 564 |
| Memory bits | 0 | 0 |
| Clock | 111.43 MHz | 112.41 MHz |
| Period | 8.970 $ns$ | 8.896 $ns$ |

Table 5.5: 1D-DCT with Offset Binary Coding

### 5.1.3 DCT Architectures Compared

Table 5.6 summarizes the most relevant features of the implemented 1D-DCT architectures, notice that Distributed Arithmetic implementations operate at higher clock frequencies than the fast algorithm, this takes particular relevance because we did not use any pipeline barrier in DA architectures, so we can assume a higher operating frequency if pipelining technique is introduced in the design.

| Architecture | DCT-1D Fast Algorithm | DCT-1D Distributed Arithmetic | DCT-1D Distributed Arithmetic with OBC |
|---|---|---|---|
| Logic Elements | 3482/33216 (10%) | 632/33216 (2%) | 977/33216 (3%) |
| Combinational Functions | 3413/33216 (10%) | 427/33216 (1%) | 659/33216 (2%) |
| Flip-Flops | 1015/33216 (3%) | 476/33216 (1%) | 564/33216 (2%) |
| Memory | 19968/483840 (4%) | 2048/483840 (<1%) | 0/483840 (0%) |
| Max. Frequency | 107.92 MHz | 115.54 MHz | 111.48 MHz |
| Latency | 120 ns | 470 ns | 470 ns |

Table 5.6: DCT-1D architectures synthesis results for Altera Cyclone II device

## 5.2 Entropy Coding

### 5.2.1 Sorting Algorithms

In this section we show the synthesis results for the implemented sorting networks, the first element to report is the Compare-Exchange module, table 5.7 shows the synthesis report for the CE, in fact it is a very small module consisting in only 34 Logic Elements (LE); synthesis results are the same even if we change the optimization technique to area or speed. Table 5.8 shows the synthesis results for a 4-item merging network and table 5.9 shows the results for 8, 16, 32 and 64 item merging networks; 64-item MN is not fitted to an specific device because we are working on the Web Edition of Quartus II and larger FPGAs are not available for synthesis.

| Device | Cyclone II EP2C35F672C6 |
|---|---|
| Logic Elements | 34/33216 (< 1%) |
| Combinational Functions | 34/33216 (< 1%) |
| Flip-Flops | 0 |
| Pins | 37 |
| Memory bits | 0/483840 (0%) |
| Worst Case TPD | 13.113 $ns$ |

Table 5.7: Compare-Exchange Synthesis Results

| Device | Cyclone II EP2C35F672C6 |
|---|---|
| Logic Elements | 102/33216 (< 1%) |
| Combinational Functions | 102/33216 (< 1%) |
| Flip-Flops | 0 |
| Pins | 73 |
| Memory bits | 0/483840 (0%) |
| Worst Case TPD | 19.134 $ns$ |

Table 5.8: 4-item merging network synthesis

Table 5.9 has an interesting feature; in Chapter 3 the required number of compare-exchange elements to merge a $N$ element list is defined as shown in equation 5.1, Considering that the CE module was synthesized in 34 LEs, then the number of CEs can be determined if we divide the number of LEs by 34. Worst case Time Propagation Delay (TPD) is reported as it will serve as the comparison basis for future pipelined implementations.

| MN Size | 8-item | 16-item | 32-item | 64-item |
|---|---|---|---|---|
| Device | EP2C35F672C6 | EP2C35F672C6 | EP2C70F896C6 | - |
| Logic Elements | 306 (< 1%) | 850 (3%) | 2210 (3%) | 5474 |
| CE Modules | 9 | 25 | 65 | 161 |
| Pins | 145 | 289 | 577 | 1153 |
| Worst Case TPD | 24.210 $ns$ | 30.147 $ns$ | 40.726 $ns$ | - |

Table 5.9: Merging Networks synthesis comparison

$$CE_{merge} = \log_2\left(\frac{N}{2}\right) \cdot \frac{N}{2} + 1 \tag{5.1}$$

Then, the required amount of CE elements to sort a list is expressed in equation 5.2, table 5.10 shows the synthesis results for some sorting networks. Notice that that the logic element number increases drastically as some merging networks have to be recursively integrated in the design.

$$CE_{sort} = \frac{N}{4} \cdot \left[(\log_2 N)^2 - \log_2 N + 4\right] - 1 \tag{5.2}$$

| SN Size | 8-item | 16-item | 32-item | 64-item |
|---|---|---|---|---|
| Device | EP2C35F672C6 | EP2C35F672C6 | EP2C70F896C6 | - |
| Logic Elements | 646 (2%) | 2142 (6%) | 6494 (9%) | 18586 |
| CE Modules | 19 | 63 | 191 | 543 |
| Pins | 145 | 289 | 577 | 1153 |
| Worst Case TPD | 37.907 $ns$ | 59.382 $ns$ | 88.532 $ns$ | - |

Table 5.10: Sorting Networks synthesis comparison

Now that the SNs are synthesized and the worst case time propagation delay is calculated, we can look forward accelerating the circuit using the pipelining technique, table 5.11 shows the results of SN synthesis considering two pipeline barriers, one located at input ports and the other at output ports, notice the slight LE increment and the new time period.

It comes to our attention that the maximum operating frequencies of the SNs are not suitable for MPEG-2 coding as they are below 27 MHz; for the first architectural proposal (Macroblock frequency table generation) we require a 256-item sorter, based on equations 3.31 and 3.32 we need 3839 Compare-Exchange elements, arranged in 36 layers and at least 122848 logic elements *without pipelining*; as we look toward high operating frequency and area optimization, this solution is beyond the scope, so we need to find another way to sort the macroblock (or the block, depending on the chosen architecture); parallelizing smaller sorting networks seems to be the the only option, but we can also include a pipelined strategy

| SN Size | 8-item | 16-item | 32-item | 64-item |
|---|---|---|---|---|
| Device | EP2C35F672C6 | EP2C35F672C6 | EP2C70F896C6 | - |
| Logic Elements | 659 | 2292 | 6800 | 19205 |
| Flip-Flops | 144 | 288 | 576 | 1152 |
| Clock | 41.81 MHz | 25.50 MHz | 16.31 MHz | - |
| Period | 23.915 $ns$ | 39.210 $ns$ | 61.330 $ns$ | - |

Table 5.11: Sorting Networks synthesis comparison

to increase clock rate, table 5.12 shows the synthesis result for a 32-item sorting network using area optimization, notice that the number of LE decreases and the clock frequency complies with MPEG requirements, figure5.3 shows the synthesized architecture of a 32-item SN without pipelining.

| Device | EP2C70F896C6 |
|---|---|
| Logic Elements | 5324 |
| Flip-Flops | 1536 |
| Pipeline Barriers | 5 |
| Clock | 41.70 MHz |
| Period | 23.981 $ns$ |

Table 5.12: 32-item Pipelined Sorting Network

Sorting networks arise as a feasible option to calculate the probability density function of any given image; as these arrays tend to occupy large silicon areas we must look forward a higher level solution that includes the selection of small sorting networks and both acceleration techniques, parallelism and pipelining. We are convinced that the video coding process can be enhanced taking advantage of the "idle" clock cycles between the input of the processor and the output of the quantization block by the parallelization of the pdf obtention to determine the best codeword mapping of the symbols.

## 5.3 Conclusions

Synthesis results show that the elements of the proposed architecture are large, but feasible for implementation in reconfigurable platforms like FPGAs, the largest modules are sorting networks; in exchange they can be pipelined to increase their maximum operating frequency to comply with video coding requirements; the bottleneck of the proposed architectures is located in the transposition memory, where serial to parallel and parallel to serial operations must be performed to address data between DCT cores. If we parallelize the DCT calculation with probability density function we reduce the latency of the architecture as we do not have to wait until the orthogonal transform is performed to begin the codeword assignation.

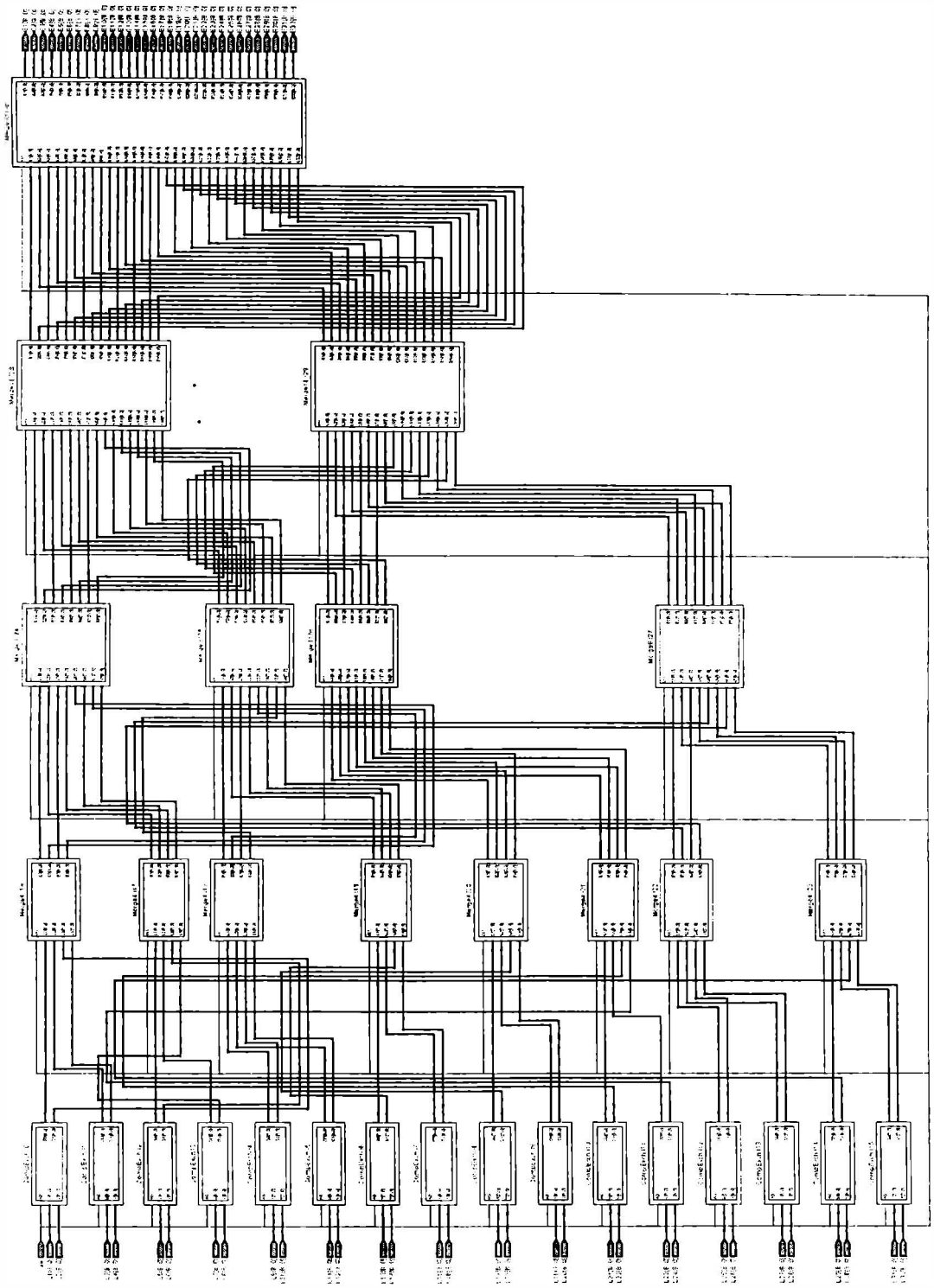


Figure 5.3: RTL of a 32-item Sorting Network

# Chapter 6

# Conclusions

In this thesis, analysis of video coding processes toward the design of a hardware video co-processor was presented; Video coding is a highly demanding process that requires a large amount of arithmetic and logic operations. Many processing blocks are currently implemented in hardware as application specific integrated circuits; our main interest is to determine the feasibility of integrating the main layer of coding systems (DCT, Q and Huffman Coding) in a single device.

Historically, designers have centered their efforts in video decoding, but the emerging necessities in video transmission capabilities for consumer electronics have made evident that a video coder for mobile applications is required; of course we could join existent modules to perform the task, but the main issue is that electronic devices are small and a single integrated circuit is required.

The first coding task implies data compression, three architectures to compute the Discrete Cosine Transform were implemented and compared in terms of material complexity and operating frequency; synthesis results showed that Distributed Arithmetic architectures offer the best trade off between both parameters; the problem with these finite precision implementations is the accuracy loss as many rounding and truncating operations are required to maintain a low material complexity.

Quantization was an easy process as it only required a hardwired operation to reduce the 12-bit outputs of the DCT to a 9-bit quantization level. Huffman coding was analyzed in terms of material complexity and control logic; many reported implementations require vast amounts of resources and complex control logic; we decided to explore the feasibility of sorting networks to generate the probability density function of a given block on the fly instead of waiting until the quantization process is finished.

Sorting Networks were implemented using combinational logic blocks configured in a vertical array fashion; this architectures have the advantage of being simple as no control logic is required to operate the network, the main drawback is that these architectures tend to be

large as we must parallelize comparators over and over again until we configure the required amount of elements.

Synthesis results showed that they are efficient in terms of processing speed as they are capable of sorting large lists of numbers in a few clock cycles, especially when pipelining is introduced. The generation of frequency tables on the fly is a promising area that can be exploited in video coding, normally the dynamic generation of this tables is processed in memory oriented architectures that tend to be slow as there is an strong dependency on memory access cycles for reading and writting.

Two architectures that parallelize DCT and Quantization with sorting networks were proposed, we are certain that the implementation of sorting algorithms to generate frequency tables at the same time that the lossy compression processes are executed will help to reduce the critical path of integrated video processing devices; Hardware implementation results of DCT cores and sorting networks helped to determine the approximate material complexity of the Co-Processor and helped to conclude that these architectures are feasible for implementation in reconfigurable platforms.

Finally, reconfigurable platforms have proven useful for designing VLSI architectures toward full custom designs as they can be used to test many different configurations of the same system without the necessity to wait until one version of the architecture is produced and tested; hence, reconfigurable architectures help to reduce design time at the same that makes easier the translation to full custom designs.

## 6.1 Future Work

There are a number of areas considered in this thesis where further work could be developed:

1. Hardware implementation of the proposed architectures in a single FPGA.
2. Finish the algorithm-architecture adequation of the main layer.
3. Analysis of material complexity and exploration of parallel and pipelining techniques to accelerate the coding process.
4. Establish the parameters to translate the designs to an ASIC.

# Bibliography


[1] M. Al-Mualla, *Video Coding for Mobile Communications.* Academic Press, 2002.

[2] V. Bhaskaran and K. Konstantinides, *Digital Pictures Representation: Compression and Standards.* Massachusetts, United States: Kluwer Academic Publisher, 2003.

[3] R. Gonzalez and R. E. Woods, *Digital Image Processing.* Prentice Hall, 2008.

[4] P. Symes, *Digital Video Compression.* McGraw-Hill, 2004.

[5] A. Netravali and B. Haskell, *Digital Pictures Representation: Compression and Standards.* United States: Newness, Elsevier, 1995.

[6] H. Sun, X. Chen, and T. Chiang, *Digital video transcoding for transmission and storage.* CRC Press, 2005.

[7] T. Sikora, *Digital Video Coding Standards: Signal processing for multimedia.* United States: IOS Press, 1999.

[8] K. Jack, *Video Demystified, a handbook for the digital Engineer.* Newness, Elsevier, 2007.

[9] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed fpga," in *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pp. 22 –28, apr 1997.

[10] B. Cope, P. Cheung, W. Luk, and S. Witt, "Have gpus made fpgas redundant in the field of video processing?," in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pp. 111 –118, dec. 2005.

[11] B. Cope, P. Cheung, and W. Luk, "Bridging the gap between fpgas and multi-processor architectures: A video processing perspective," in *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pp. 308 –313, july 2007.

[12] T. Hamamoto, R. Ooi, Y. Ohtsuka, and K. Aizawa, "Real-time image processing by using image compression sensor," in *Image Processing, 1999. ICIP 99. Proceedings. 1999 International Conference on*, vol. 3, pp. 935 –939 vol.3, 1999.

[13] C. Dick, "Fpgas: The high-end alternative for dsp applications," *The Journal of Embedded Signal Processing, DSP, FPGA*, pp. 935–939, 2000.

[14] C. Claus, W. Stechele, M. Kovatsch, J. Angermeier, and J. Teich, "A comparison of embedded reconfigurable video-processing architectures," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 587 –590, sept. 2008.

[15] J. Dubois, M. Mattavelli, L. Pierrefeu, and J. Miteran, "Configurable motion-estimation hardware accelerator module for the mpeg-4 reference hardware description platform," in *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, vol. 3, pp. III – 1040–3, sept. 2005.

[16] V. Prasad Arava, M. Jo, H. Lee, and K. Choi, "A generic design for encoding and decoding variable length codes in multi-codec video processing engines," in *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual*, pp. 197 –202, april 2008.

[17] S. Vitabile, A. Gentile, S. Siniscalchi, and F. Sorbello, "Efficient rapid prototyping of image and video processing algorithms," in *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, pp. 452 – 458, aug.-3 sept. 2004.

[18] N. Lawal, B. Thornberg, and M. O'Nils, "Power-aware automatic constraint generation for fpga based real-time video processing systems," in *Norchip, 2007*, pp. 1 –5, nov. 2007.

[19] N. Lawal, M. O'Nils, and B. Thornberg, "C++ based system synthesis of real-time video processing systems targeting fpga implementation," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1 –7, march 2007.

[20] N. Lawal and M. O'Nils, "Embedded fpga memory requirements for real-time video processing applications," in *NORCHIP Conference, 2005. 23rd*, pp. 206 – 209, nov. 2005.

[21] N. Lawal, B. Thornberg, and M. O'Nils, "Address generation for fpga rams for efficient implementation of real-time video processing systems," in *Field Programmable Logic and Applications, 2005. International Conference on*, pp. 136 – 141, aug. 2005.

[22] G. De Michell and R. Gupta, "Hardware/software co-design," *Proceedings of the IEEE*, vol. 85, pp. 349 –365, mar 1997.

[23] Xilinx, "Embedded processing, [online]," 2010.

[24] Altera, "Embedded processor, [online]," 2010.

[25] T. Hashimoto, M. Ohashi, M. Matsuo, S. Kuromaru, T. Mori-iwa, M. Hamada, Y. Sugisawa, H. Tomita, M. Hoshino, T. Nakamura, K. Ishida, K. Watada, T. Fukunaga, and J. Michiyama, "A 27-mhz/54-mhz 11-mw mpeg-4 video decoder lsi for mobile applications," *Solid-State Circuits, IEEE Journal of*, vol. 37, pp. 1574 – 1581, nov 2002.

[26] D. Alfonso, A. Artieri, A. Capra, M. Mancuso, F. Pappalardo, F. Rovati, and R. Zafalon, "Ultra low-power multimedia processor for mobile multimedia applications," in *Solid-State Circuits Conference, 2002. ESSCIRC 2002. Proceedings of the 28th European*, pp. 63 –69, sept. 2002.

[27] P. Sedcole, P. Cheung, G. Constantinides, and W. Luk, "Run-time integration of reconfigurable video processing systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 15, pp. 1003 –1016, sept. 2007.

[28] R. Jayaraman, "(when) will fpgas kill asics," *Proceedings of the 38th annual Design Automation Conference*, pp. 321–322, 2001.

[29] N. Ahmed, T. Natarajan, and K. Rao, "Discrete cosine transform," *Computers, IEEE Transactions on*, vol. C-23, pp. 90 –93, jan. 1974.

[30] S. Wolter, D. Birrèck, and R. Laur, "Classification for 2d-dcts and a new architecture with distributed arithmetic," in *Circuits and Systems, 1991., IEEE International Symposium on*, pp. 2204 –2207 vol.4, jun 1991.

[31] E. Feig and S. Winograd, "Fast algorithms for the discrete cosine transform," *Signal Processing, IEEE Transactions on*, vol. 40, pp. 2174 –2193, sep 1992.

[32] W.-H. Chen, C. Smith, and S. Fralick, "A fast computational algorithm for the discrete cosine transform," *Communications, IEEE Transactions on*, vol. 25, pp. 1004 – 1009, sep 1977.

[33] A. Artieri, S. Kritter, F. Jutand, and N. Demassieux, "A one chip vlsi for real time two-dimensional discrete cosine transform," in *Circuits and Systems, 1988., IEEE International Symposium on*, pp. 701 –704 vol.1, jun 1988.

[34] B. Lee, "A new algorithm to compute the discrete cosine transform," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 32, pp. 1243 – 1245, dec 1984.

[35] C. Loeffler, A. Ligtenberg, and G. Moschytz, "Practical fast 1-d dct algorithms with 11 multiplications," in *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*, pp. 988 –991 vol.2, may 1989.

[36] N. I. Cho and S. U. Lee, "Dct algorithms for vlsi parallel implementations," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 38, pp. 121 –127, jan 1990.

[37] C. Cheng and K. Parhi, "A novel systolic array structure for dct," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 52, pp. 366 – 369, july 2005.

[38] S. Yu and J. Swartzlander, E.E., "Dct implementation with distributed arithmetic," *Computers, IEEE Transactions on*, vol. 50, pp. 985 –991, sep 2001.

[39] S. Yu and J. Swartzlander, E.E., "A scaled dct architecture with the cordic algorithm," *Signal Processing, IEEE Transactions on*, vol. 50, pp. 160 –167, jan 2002.

[40] K. Geetha and M. Uttarakumari, "New polynomial transform algorithm for 2-d dct using ramanujan ordered numbers," in *Signal Processing and Communications (SPCOM), 2010 International Conference on*, pp. 1 –5, july 2010.

[41] J. Prado and P. Duhamel, "A polynomial-transform based computation of the 2-d dct with minimum multiplicative complexity," in *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, vol. 3, pp. 1347 –1350 vol. 3, may 1996.

[42] J.-L. Wang, C.-B. Wu, B.-D. Liu, and J.-F. Yang, "Implementation of the discrete cosine transform and its inverse by recursive structures," in *Signal Processing Systems, 1999. SiPS 99. 1999 IEEE Workshop on*, pp. 120 –130, 1999.

[43] Y. Zeng, G. Bi, and A. Leyman, "New polynomial transform algorithm for multidimensional dct," *Signal Processing, IEEE Transactions on*, vol. 48, pp. 2814 –2821, oct 2000.

[44] A. Patino, M. Peiro, F. Ballester, and G. Paya, "2d-dct on fpga by polynomial transformation in two-dimensions," in *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, vol. 3, pp. III – 365–8 Vol.3, may 2004.

[45] F. Kamangar and K. Rao, "Fast algorithms for the 2-d discrete cosine transform," *Computers, IEEE Transactions on*, vol. C-31, pp. 899 –906, sept. 1982.

[46] M. Schulte and J. Swartzlander, E.E., "Hardware designs for exactly rounded elementary functions," *Computers, IEEE Transactions on*, vol. 43, pp. 964 –973, aug 1994.

[47] A. Grigoryan, "An algorithm for calculation of the discrete cosine transform by paired transform," *Signal Processing, IEEE Transactions on*, vol. 53, pp. 265 – 273, jan. 2005.

[48] A. Peled and B. Liu, "A new hardware realization of digital filters," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 22, pp. 456 – 462, dec 1974.

[49] K. Nourji and N. Demassieux, "Optimal vlsi architecture for distributed arithmetic-based algorithms," in *Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, vol. ii, pp. II/509 –II/512 vol.2, apr 1994.

[50] L. F. Gonzalez-Perez, *Architectures VLSI pour le Codage Conjoint Source-Canal en Treillis*. PhD thesis, Ecole Nationale Superieure des Telecommunications, 2000.

[51] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, pp. 1098 –1101, sept. 1952.

[52] T. Kumaki, Y. Kuroda, T. Koide, H. Mattausch, H. Noda, K. Dosaka, K. Arimoto, and K. Saito, "Cam-based vlsi architecture for huffman coding with real-time optimization of the code word table [image coding example]," in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pp. 5202 – 5205 Vol. 5, may 2005.

[53] H. Park and V. Prasanna, "Area efficient vlsi architectures for huffman coding," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 40, pp. 568 –575, sep 1993.

[54] S.-M. Lei and M.-T. Sun, "An entropy coding system for digital hdtv applications," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 1, pp. 147 –155, march 1991.

[55] S.-F. Chang and D. Messerschmitt, "Vlsi designs for high-speed huffman decoder," in *Computer Design: VLSI in Computers and Processors, 1991. ICCD '91. Proceedings, 1991 IEEE International Conference on*, pp. 500 –503, oct 1991.

[56] O. Hauck, H. Sauerwein, and S. Huss, "Asynchronous vlsi architectures for huffman codecs," in *Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium on*, vol. 5, pp. 542 –545 vol.5, may-3 jun 1998.

[57] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms.* The MIT Press, 1990.

[58] L.-Y. Liu, J.-F. Wang, R.-J. Wang, and J.-Y. Lee, "Cam-based vlsi architectures for dynamic huffman coding," in *Consumer Electronics, 1994. Digest of Technical Papers., IEEE 1994 International Conference on*, pp. 204 –205, jun 1994.

[59] D. E. Knuth, *The art of computer programming: Sorting and Searching.* Addison Wesley, 1973.

[60] T. Kumaki, Y. Kuroda, T. Koide, H. Jurgen Mattausch, H. Noda, K. Dosaka, K. Arimoto, and K. Saito, "Multi-port cam based vlsi architecture for huffman coding with real-time optimized code word table," in *Circuits and Systems, 2005. 48th Midwest Symposium on*, pp. 55 – 58 Vol. 1, aug. 2005.

[61] M. Ishizaki, T. Kumaki, Y. Kouno, T. Koide, H. Mattausch, Y. Kuroda, T. Gyoten, H. Noda, K. Dosaka, K. Arimoto, and K. Saito, "Huffman encoding architecture with self-optimizing performance and multiple cam-match utilization," in *TENCON 2006. 2006 IEEE Region 10 Conference*, pp. 1 –4, nov. 2006.

[62] M. Rudberg and L. Wanhammar, "High speed pipelined parallel huffman decoding," in *Circuits and Systems, 1997. ISCAS '97., Proceedings of 1997 IEEE International Symposium on*, vol. 3, pp. 2080 –2083 vol.3, jun 1997.

[63] K. Babu and V. Kumar, "Implementation of data compression using huffman coding," in *Methods and Models in Computer Science (ICM2CS), 2010 International Conference on*, pp. 70 –75, dec. 2010.

[64] D. Bitton, D. J. DeWitt, D. K. Hsaio, and J. Menon, "A taxonomy of parallel sorting," *ACM Comput. Surv.*, vol. 16, pp. 287–318, September 1984.

[65] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), (New York, NY, USA), pp. 307–314, ACM, 1968.

[66] R. Perez-Andrade, R. Cumplido, F. Del Campo, and C. Feregrino-Uribe, "A versatile linear insertion sorter based on a fifo scheme," in *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual*, pp. 357 –362, april 2008.

[67] G. Baudet and D. Stevenson, "Optimal sorting algorithms for parallel computers," *Computers, IEEE Transactions on*, vol. C-27, pp. 84 –87, jan. 1978.

[68] C. Kruskal, "Searching, merging, and sorting in parallel computation," *Computers, IEEE Transactions on*, vol. C-32, pp. 942 –946, oct. 1983.

[69] F. Preparata, "New parallel-sorting schemes," *Computers, IEEE Transactions on*, vol. C-27, pp. 669 –673, july 1978.

[70] H. Yasuura, N. Takagi, and S. Yajima, "The parallel enumeration sorting scheme for vlsi," *Computers, IEEE Transactions on*, vol. C-31, pp. 1192 –1201, dec. 1982.

[71] J.-D. Lee and K. Batcher, "A bitonic sorting network with simpler flip interconnections," in *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings. Second International Symposium on*, pp. 104 –109, jun 1996.

[72] C. Thompson, "The vlsi complexity of sorting," *Computers, IEEE Transactions on*, vol. C-32, pp. 1171 –1184, dec. 1983.

[73] L. de Vos and M. Stegherr, "Parameterizable vlsi architectures for the full-search block-matching algorithm," *Circuits and Systems, IEEE Transactions on*, vol. 36, pp. 1309 –1316, oct 1989.

[74] M. Liou, "Algorithms and vlsi implementation for block-matching motion estimation," in *Circuits and Systems, 1994. APCCAS '94., 1994 IEEE Asia-Pacific Conference on*, pp. 530 –531, dec 1994.

[75] K. Tavassoli and W. Badawy, "A prototype for parallel motion estimation architecture using full-search block matching algorithm," in *Digital and Computational Video, 2002. DCV 2002. Proceedings. Third International Workshop on*, pp. 129 – 134, nov. 2002.

[76] H. Loukil, F. Ghozzi, A. Samet, M. Ben Ayed, and N. Masmoudi, "Hardware implementation of block matching algorithm with fpga technology," in *Microelectronics, 2004. ICM 2004 Proceedings. The 16th International Conference on*, pp. 542 – 546, dec. 2004.

[77] S. Ertuk, "A new perspective to block motion estimation for video compression: High-frequency component matching," *Signal Processing Letters, IEEE*, vol. 14, pp. 113 –116, feb. 2007.

[78] G. Gupta and C. Chakrabarti, "Vlsi architectures for hierarchical block matching," in *Circuits and Systems, 1994. ISCAS '94., 1994 IEEE International Symposium on*, vol. 4, pp. 215 –218 vol.4, may-2 jun 1994.

[79] G. Gupta and C. Chakrabarti, "Architectures for hierarchical and other block matching algorithms," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 5, pp. 477 –489, dec 1995.

[80] S. Wong, S. Vassiliadis, and S. Cotofana, "A sum of absolute differences implementation in fpga hardware," in *Euromicro Conference, 2002. Proceedings. 28th*, pp. 183 – 188, 2002.

[81] C.-P. Fan and S.-W. Lin, "Fast global elimination algorithm and low-cost vlsi design for motion estimation," in *TENCON 2007 - 2007 IEEE Region 10 Conference*, pp. 1 –4, 30 2007-nov. 2 2007.

[82] T. Komarek and P. Pirsch, "Array architectures for block matching algorithms," *Circuits and Systems, IEEE Transactions on*, vol. 36, pp. 1301 -1308, oct 1989.

[83] A. Ryszko and K. Wiatr, "An assessment of fpga suitability for implementation of real-time motion estimation," in *Digital Systems, Design, 2001. Proceedings. Euromicro Symposium on*, pp. 364 –367, 2001.

[84] M. Mohammadzadeh, M. Eshghi, and M. Azadfar, "Parameterizable implementation of full search block matching algorithm using fpga for real-time applications," in *Devices, Circuits and Systems, 2004. Proceedings of the Fifth IEEE International Caracas Conference on*, vol. 1, pp. 200 – 203, nov. 2004.

[85] J. Olivares, I. Benavides, J. Hormigo, J. Villalba, and E. Zapata, "Fast full-search block matching algorithm motion estimation alternatives in fpga," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pp. 1 –4, aug. 2006.

[86] A. Costa, A. De Gloria, P. Faraboschi, and F. Passaggio, "A vlsi architecture for hierarchical motion estimation," *Consumer Electronics, IEEE Transactions on*, vol. 41, pp. 248 –257, may 1995.

[87] Y.-S. Jehng, L.-G. Chen, and T.-D. Chiueh, "An efficient and simple vlsi tree architecture for motion estimation algorithms," *Signal Processing, IEEE Transactions on*, vol. 41, pp. 889 –900, feb 1993.

[88] C.-M. Wu and D.-K. Yeh, "A vlsi motion estimator for video image compression," *Consumer Electronics, IEEE Transactions on*, vol. 39, pp. 837 –846, nov 1993.

[89] S. H. Nam, J. S. Baek, and M. K. Lee, "Flexible vlsi architecture of full search motion estimation for video applications," *Consumer Electronics, IEEE Transactions on*, vol. 40, pp. 176 –184, may 1994.

[90] B. Natarajan, V. Bhaskaran, and K. Konstantinides, "Low-complexity block-based motion estimation via one-bit transforms," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 7, pp. 702 –706, aug 1997.

[91] Y. Baek, H.-S. Oh, and H.-K. Lee, "An efficient block-matching criterion for motion estimation and its vlsi implementation," *Consumer Electronics, IEEE Transactions on*, vol. 42, pp. 885 –892, nov 1996.

[92] V. Do and K. Yun, "A low-power vlsi architecture for full-search block-matching motion estimation," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 8, pp. 393 –398, aug 1998.

[93] K. K. Parhi, *VLSI Digital Signal Processing Systems Design and Implementation.* Wiley-Interscience, 1999.

[94] A. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, pp. 234–240, 1951.

[95] J. E. Volder, "The cordic trigonometric computing technique," *Electronic Computers, IRE Transactions on*, vol. EC-8, pp. 330 –334, sept. 1959.

[96] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, FPGA '98, (New York, NY, USA), pp. 191–200, ACM, 1998.

[97] M. Ercegovac and T. Lang, *Digital Arithmetic.* Morgan and Kauffman, 2003.

[98] D. Salomon and G. Motta, *Handbook of Data Compression.* Springer-Verlag, 2010.