VLSI Design Concepts for Iterative Algorithms

Von der Fakultät für Elektrotechnik und Informationstechnik
der Technischen Universität Dortmund
genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
eingereicht von

Chi-Chia Sun

Tag der mündlichen Prüfung:	11.04.2011
Hauptreferent:	Univ.-Prof. Dr.-Ing. Jürgen Götze
Korreferent:	Univ.-Prof. Dr.-Ing. Rüdiger Kays

Arbeitsgebiet Datentechnik, Technische Universität Dortmund

Abstract

Circuit design becomes more and more complicated, especially when the Very Large Scale Integration (VLSI) manufacturing technology node keeps shrinking down to nanoscale level. New challenges come up such as an increasing gap between the design productivity and the Moore's Law. Leakage power becomes a major factor of the power consumption and traditional shared bus transmission is the critical bottleneck in the billion transistors Multi-Processor System–on–Chip (MPSoC) designs. These issues lead us to discuss the impact on the design of iterative algorithms.

This thesis presents several strategies that satisfy various design constraints, which can be used to explore superior solutions for the circuit design of iterative algorithms. Four selected examples of iterative algorithms are elaborated in this respect: hardware implementation of COordinate Rotation DIgital Computer (CORDIC) processor for signal processing, configurable DCT and integer transformations based CORDIC algorithm for image/video compression, parallel Jacobi Eigenvalue Decomposition (EVD) method with arbitrary iterations for communication, and acceleration of parallel Sparse Matrix–Vector Multiplication (SMVM) operations based Network–on–Chip (NoC) for solving systems of linear equations. These four applications of iterative methods have been chosen since they cover a wide area of current signal processing tasks.

Each method has its own unique design criteria when it comes to the direct implementation on the circuit level. Therefore, a balanced solution between various design tradeoffs is elaborated for each method. These tradeoffs are between throughput and power consumption, computational complexity and transformation accuracy, the number of inner/outer iterations and energy consumption, data structure and network topology. It is shown that all of these algorithms can be implemented on FPGA devices or as ASICs efficiently.

Acknowledgements

This thesis was written while I was working as a research assistant at the Information Processing Laboratory of the Dortmund University of Technology. I would like to thank Professor Dr.-Ing. Jürgen Götze, the head of the laboratory, for all the interesting discussions that contributed essentially to this thesis, for creating an open and relaxed atmosphere, and for providing excellent working conditions.

Furthermore, I am very pleased to thank Professor Dr.-Ing. Rüdiger Kays (TU Dortmund) for his interest in my works, his comments on my thesis and his reviews for my DAAD scholarship, and his time.

I would also like to thank Professor Shanq-Jang Ruan (Low-Power System Lab, National Taiwan University of Science and Technology) for his guidance during my Master study in Taipei. I am especially grateful to my present and former colleagues for providing such a stimulating atmosphere at the laboratory. It was a pleasure to share so much time with you. Special thanks goes to many students for their contributions to this work too.

To my parents and my sister for their support and encouragements during the long years of my education.

Dortmund Germany, April 2011

Contents

List of Figures

List of Tables

1 Introduction

Modern Very Large Scale Integration (VLSI) manufacturing technology has kept shrinking down to Very Deep Sub-Micron (VDSM) with a very fast trend and Moore's Law is expected to hold for the next decade [1, 41] or extend to the More than Moore concept (a prediction for the integration of more than thousand cores before 2020) [21, 64, 95]. 10 years ago, for 0.35μm technology, design engineers focused on reducing the area size to lower down the cost. Later, when it came to 0.13μm technology, they paid huge efforts to improve the signal integrity and reduce the power consumption for low power devices. More and more functionalities can be integrated on an integrated circuit due to the continuing improvements.

As the manufacturing technology node decreases to the 65nm, the circuit design methodology poses new challenges: timing delay of the global wire interconnection is increasing severely in relation to the local processor element, leakage power becomes a major factor of the power consumption, and shared bus transmission is the new bottleneck in the billion transistors System–on–Chip (SoC) designs [24, 99, 125]. On the other hand, as product life cycle continues to shrink simultaneously, time–to–market also becomes a key design constraint. In consequence, these problems result in the famous "designer productivity gap" as illustrated in Figure 1.1 [85]. In this chart, the x–axis denotes the progression in time and the y–axis denotes the growing rate as measured by the number of logic transistors per chip. The solid line shows the growth rate based on the Moore's Law, while the dotted line sketches the average number of transistors that design engineers could handle monthly. It can be noticed in Figure 1.1 that there is an increasing gap. Consequently, silicon technology is far outstripping our ability to utilize these transistors efficiently for working designs in a short time.

Several strategies have been proposed to solve this widening produc-

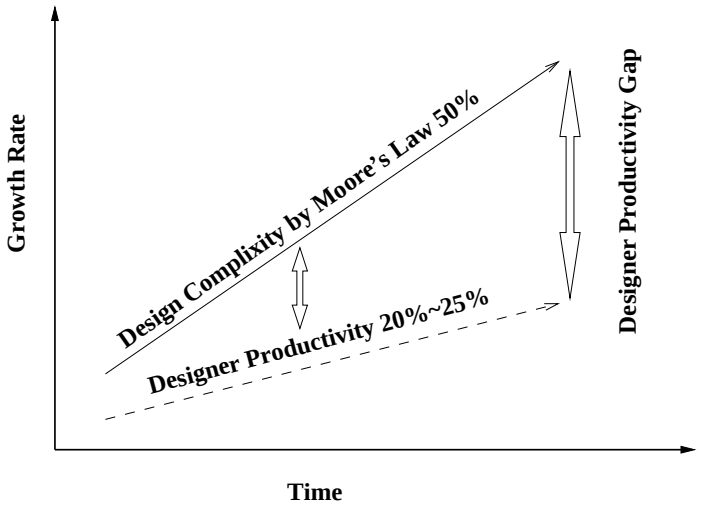


Figure 1.1: Designer productivity gap (modified from SEMATECH)

tivity gap. One of the most efficient solutions is using parallel computing, which has received great attention. It has been introduced in many state-of-the-art applications in the past few years (e.g. Six-Core CPU, MPSoC and parallel processor arrays) [104, 120, 121, 132]. In addition, modulized circuit design has developed to very large SoC by utilizing reusable/configurable Intellectual Property (IP) cores as much as possible [17]. Soon traditional bus transmission architecture will be unable to satisfy the need for more than thousand cores on a single silicon die. Hence, a better network switching method is desired.

These challenges motivate us to analyze their impact on parallel iterative algorithms. We try to present a generalized VLSI design concept that considers the significant impact on power, performance, cost, reliability, and time–to–market. To implement an iterative algorithm on a multiprocessor array, there is a tradeoff between the complexity of an iteration step (assuming that the convergence of the algorithm is retained) and the number of required iteration steps. For example, suppose we have a hardware platform with multiple processors which requires an iteration step of the iterative algorithm to be executed K times in order to obtain the convergence. The iteration step is executed in parallel on the platform. We want to simplify the processors in order to improve the logical utilization of the platform as shown in Figure 1.2. This simplification will usually cause an increased number of iterations for convergence. The number of required iterations will increase from K to $K + L$. That means the number of data transfer in

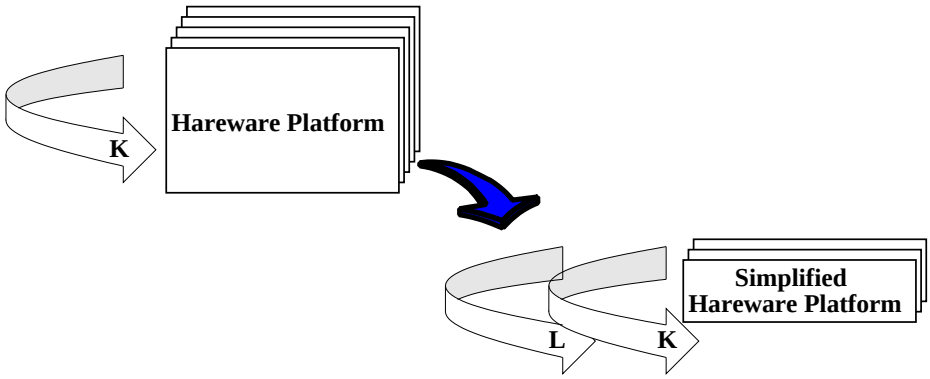


Figure 1.2: Iterative algorithm design concept

the interconnections also increases due to the behavior of the iterative algorithm. Therefore, as long as the convergence properties are guaranteed, it is possible to adjust the architecture which is normally resulting in an increased number of iteration steps. This reduces the complexity with regard to the implementation significantly. However, it is not easy to find a superior solution to balance the design criteria, especially for the performance/complexity of the hardware, the load/throughput of interconnects and the overall energy/power consumption. An example of this in the SoC design is the area and the timing optimization. Area is a metric that is aggressively optimized to achieve low chip cost. In contrast, timing closure is achieved when a particular target clock frequency is met (further timing optimization is not necessary). Optimization of one metric can be traded off for the optimization of other one. Obviously, it is extremely difficult to optimize all metrics at the same time.

As we will emphasize throughout this thesis, a design engineer must think carefully which strategy should be selected for the hardware implementation of iterative algorithms. However, a proper decision becomes more and more difficult as VLSI technology is evolving. This problem motivates us to study the design issues. Four different iterative algorithms have been selected, which cover a wide area of current signal processing tasks. All of them are implemented and realized in order to discuss the relationship between circuit design issues and the algorithmic complexity. The chosen algorithms are:

CORDIC processor A COordinate Rotation DIgital Computer (CORDIC)

can perform a lot of mathematical computations to accelerate many digital functions and signal processing applications in hardware, including linear and orthogonal transformations by requiring only shift and add operations. Since the CORDIC is also an important iterative algorithm, a brief introduction to the generic definition and the hardware implementation issues will be given first. It is a simple, but very flexible arithmetical unit. Very simple modifications to the controller lead to linear and orthogonal operating modes, capable of calculating vector rotations, angle estimations or even multiplications/divisions. Various conditions concerning the circuit design issues will be described and compared, particularly on the architecture level. We elaborate the way of implementing a CORDIC rotation with reasonable computational complexity by trading off the throughput [6, 83].

Discrete Cosine Integer Transform (DCIT) VLSI implementation of both forward and inverse CORDIC based Quantized DCIT (QDCIT) is presented. This configurable architecture not only performs multiplierless 8×8 Quantized DCT (QDCT) and 4×4/8×8 integer transforms but also contains configurable modules such that it can adjust the number of CORDIC rotations for arbitrary accuracy. Therefore, the presented architecture reduces the number of iterations when the target resolution is small (QCIF/-CIF). On the contrary, it will apply more iterations when the target resolution is large (Full-HD/Ultra-HD). Moreover, it still retains an acceptable transformation quality compared to the default methods in terms of PSNR. This leads to a high-accuracy high throughput implementation [106, 107, 112, 113].

Parallel Jacobi EVD method Parallel Jacobi method for Eigenvalue Decomposition (EVD) is chosen as an example to explain the design concepts concerning tradeoff between the complexity and the iteration (see Figure 1.2). Here, it is chosen since its convergence property is very robust. Simplifying the hardware architecture is paid by an increased number of rotations due to the behavior of Jacobi's algorithm. Nevertheless, the computational complexity is actually decreased, which also results in lower energy consumption per EVD operation [46, 109]. The implementation results demonstrate that using the simplified architecture is beneficial concerning the design criteria since it yields smaller area overhead, faster

overall computation time and less energy consumption.

Sparse Matrix-Vector Multiplication based on NoC Future integration of more than thousands IP cores for the very large SoC design will soon challenge the current shared bus transmission system [133]. In this regard, a Sparse Matrix-Vector Multiplication (SMVM) calculator with the chip-internal network is presented as a novel solution for parallel matrix computation to further accelerate many iterative solvers in hardware, such as solving systems of linear equations, Finite Element Method (FEM) and so on [110]. This methodology is called Network–on–Chip (NoC). Using NoC architecture allows the parallel processors to deal with irregular structure of the sparse matrices and achieve a high performance in FPGA by trading off the area overhead, especially when the data transfers are unstable.

In this thesis, our major concern is to explore several VLSI design concepts for iterative algorithms. Contrary to conventional circuit designs, usually reducing the logical utilization and increasing the performance, we will further look into parallel computing, configurable architecture and packet–switched network. The goal is not to optimize one or several criteria as much as possible, but trying to expose the complete tradeoff curves and to have a global view on how large the range is for real-life applications. The major contributions of this thesis are:

1. The description of different circuit design challenges is introduced briefly, especially when the technology node is very small. This leads us further to discuss the design impact on iterative algorithms from the algorithmic and the architectural point of views.

2. The investigation of VLSI design concepts is presented for future circuit design in nanoscale for four selected iterative applications: CORDIC processor for signal processing, configurable transformations for video compression, parallel EVD for communication and parallel SMVM for solving systems of linear equations. Each application has its own unique design criteria requiring design engineers to think carefully. They require investigating the tradeoffs between throughput and power consumption, computational complexity and transformation accuracy, the number of inner/outer

iterations and energy consumption, data structure and network topology. These tradeoffs are further elaborated to obtain a balanced solution for each application.

3. The circuit implementations of both forward and inverse QDCIT transformations based on the CORDIC algorithm are presented. The CORDIC based FQDCIT requires only 120 adders and 40 barrel shifters to perform the multiplierless 8×8 FQDCT and 4×4/8×8 forward quantized integer transform by sharing the hardware resources. Hence it can support different video Codecs, such as JPEG, MPEG–4, H.264 or SVC. Furthermore, for a TSMC 0.18μm circuit implementation, it can achieve small chip area and high throughput for future UHD resolution. On the other hand, the inverse architecture requires only 124 adders and 40 barrel shifters to perform the multiplierless 8×8 IQDCT and quantized 4×4/8×8 inverse integer transform. Meanwhile, the ability to adjust CORDIC iteration steps can be used to support different video resolutions.

4. A configurable Jacobi EVD array has been elaborated with both Full CORDIC (exact rotation, executing W CORDIC iterations, where W is the word length) and μ–CORDIC (approximate rotation, executing only one CORDIC iteration) in order to further study the tradeoff between the performance/complexity of processors and the load/throughput of interconnects. Moreover, utilizing a preconditioned E index method not only reduces more than 35% computational overhead for the Full CORDIC and 10% for the μ–CORDIC in average, but also omits the floating point number comparators. For a TSMC 45nm circuit implementation, a detailed comparison between area, timing delay and power/energy consumption is done.

5. A solution for sparse matrix computation based on the NoC concept is presented. Parallel SMVM computations have been tested in the Xilinx Virtex–6 FPGA. The advantages of introducing the NoC structure into SMVM computation are given by high resource utilization, flexibility and the ability to communicate among heterogeneous systems. This configurable solution can be configured as a larger p×p array as long as there are enough hardware resources ($p = 2, 4, 8, \ldots, 2^k, k \in \mathbb{N}$). Moreover, the NoC structure

can guarantee that arbitrary sparsity structures of the matrix can be handled without interfering the performance by the sparsity of the matrix.

This thesis is organized as follows: Chapter 2 gives a brief introduction to recent VLSI design trends, on the parallel implementation issues, and the low power design methodology. Then, in Chapter 3, the CORDIC algorithm is introduced and it is shown how it is derived and implemented. Two typical iterative algorithms based on the CORIDC architecture, Discrete Cosine and Integer Transform and parallel Jacobi EVD, will be presented and tested in Chapter 4 and Chapter 5 respectively. After that, in Chapter 6, SMVM based NoC is presented and preliminary implementation results are given. Finally, conclusions are given in Chapter 7.

2 Introduction to VLSI Design

In this chapter, a brief introduction to the future trend of VLSI design and its circuit design problems will be addressed. Moore's Law is expected to hold for at least 10 more years. Meanwhile, digital multimedia devices will keep driving the demand for very complex SoC systems. A single chip can contain more than billion transistors to support various functionalities. Therefore, before looking into the design issues of iterative algorithms, it is very important to review today's nanoscale VLSI technology in Section 2.1. The reason how Moore's Law will continue to influence the circuit design trend will be explained in Section 2.2. Several strategies for obtaining the timing convergence and reducing the power optimization from a higher design level to the lower level will be described from Section 2.3 to Section 2.5. The power consumption of CMOS circuit and the corresponding solutions will be introduced in Section 2.6. At the end, our motivation on VLSI design concepts for iterative algorithms will be clarified in Section 2.7.

2.1 Modern Digital Circuit Design

Silicon technology is now at the stage where it is feasible to incorporate numerous transistors on a single square centimeter of silicon such that Intel predicts the availability of 100 billion transistors on a 300mm^2 die in 2015 [17]. At this moment, multi-core SoC design emerged because it allowed design engineers to integrate few cores together for simple parallel computing. This permits to build as a complete SoC for supporting extremely complex functions, which would previously be implemented as a collection of individual chips on a PCB board. Now, since the nano-technology allows the integration of an ever-increasing number of macro-cells on a single silicon die, parallel multiprocessor platforms have received great attention and have been realized into several state-

of-the-art applications (e.g. Six–Core, MPSoC and parallel processor array) [8, 17, 132].

Besides the issue of parallelism, the ability to integrate all parts of applications on the same piece of silicon is also beneficial for lower power, greater reliability and reduced cost of manufacturing for consumer electronic devices. Consequently, increased pressure has been put on design engineers to meet a much shorter time–to–market, now measured in months rather than years. Particularly, with the new industry standards such as H.264, WCDMA, and WiMAX, which keep driving the growth of High–Definition TV (HDTV) and smart phone. However, the decreasing development period for a new Application-Specific Integrated Circuit (ASIC) not only results in a very high Non-recurring Engineering (NRE) cost, but also makes it hard to succeed the goal of time–to–market. Therefore, another popular solution Field Programmable Gate Array (FPGA) plays an important role for fitting the gap between cost and flexibility.

FPGA is an integrated circuit, which is designed to be configured after manufacturing. The FPGA configuration is generally specified using a Hardware Description Language (HDL), similar to that used for an ASIC design. Hence, FPGAs can be used to implement any logical function that an ASIC can do. In this way, the capability to update the functionality, partial reconfiguration of the design and the low NRE costs offer advantages to many applications [134]. However, the manufacturing cost per FPGA still makes it unsuitable for many consumer standard devices. For example, the price of an FPGA die which can be configured as a video decoder with MPEG–4 decoding functionality will be much higher than a dedicated ASIC. Therefore, the choice between the FPGA and the ASIC design is simple the question if either the size of market is large enough to afford ASIC development costs or the devices need the reconfiguration for supporting new functionalities after shipping.

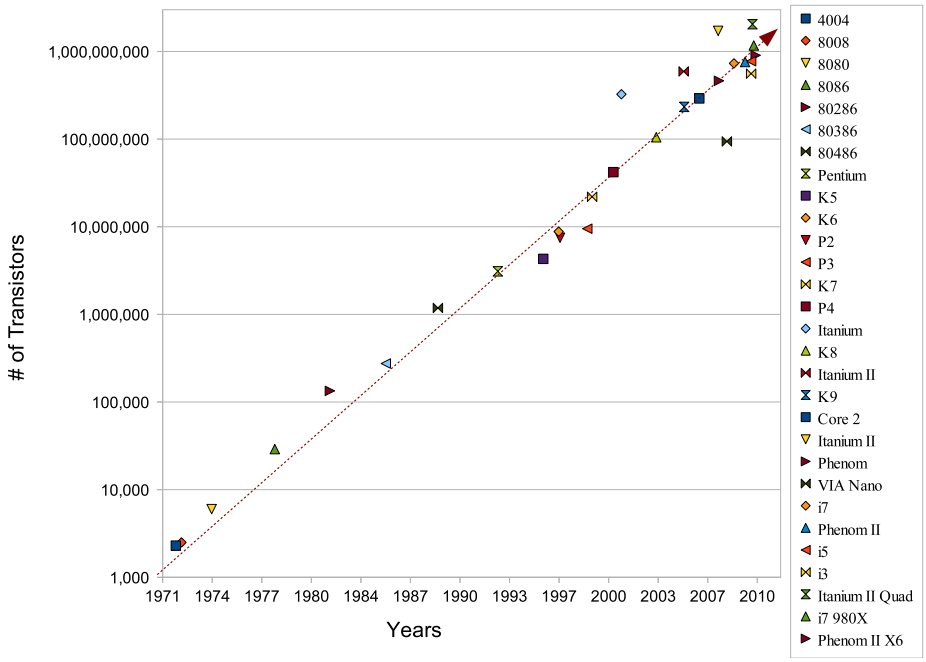


Figure 2.1: Moore's Law: Plot of x86 CPU transistor counts from 1970 until 2010

2.2 Moore's Law

In 1965 Gordon E. Moore has predicted a long-term trend of computing hardware, in which the number of transistors that can be placed on an integrated circuit will double approximately every two years [80]. Note that it is often incorrectly cited as a doubling of transistors every 18 months. The actual average period is about 20 months. Historically, Moore's Law has precisely described a driving force of technological and social change with respect to cost, functionality and performance in the late 20th and early 21st centuries. For example, if we consider the CPU transistor counts from 1970 until 2010 in Figure 2.1, it results in a constant line corresponding to exponential growth due to the logarithmic scale.

At the beginning, the first x86 Intel 4004 contained only thousand transistors; 10 years ago, the transistor counts of an Intel Pentium increased very fast to a million transistors. Until now a single Quad-Core/Six-Core CPU can integrate more than a billion transistors. In Appendix A, Table A.1 shows more detailed information for each x86

CPU model. In the future, Moore's Law will continue until 2020 [41,90] or maybe even further. After that, soon CMOS technology will meet its physical limitation when the node size is smaller than 10nm. Now many scientists are trying to replace the current silicon based MOSFET by a novel carbon based Carbon Nanotube Field Effect Transistor (CNFET) or spintronics in order to shrink the node size into the atom level [7, 61, 86]. If it comes true, the computer will usher in a new era "Beyond CMOS" (also known as "More Moore"). Unfortunately, it seems that this is probably not going to happen so easily in the next 10 years. On the other hand, other groups came out with another potential way to keep Moore's Law alive by using a Three-Dimensional IC (3D-IC) concept to increase the density of transistors [66,91]. As far as we can see the Through-Silicon Via (TSV) technology for 3D-IC will be feasible before 2012.

More and more evidences point out that the trend of Moore's Law becomes slow, especially the 2009 executive summary of International Technology Roadmap for Semiconductors (ITRS) provides a taxonomy of scaling in the traditional, "More than Moore" sense. Figure 2.2 shows three possible trends. They envision future integrated circuit system will perform diverse functions such as high-accuracy sensing of real-time signals, energy harvesting, and on-chip chemical/biological sensors in a System-in-Package (SiP) or a System-of-Package (SoP) design [64, 95]. In this way, the incorporation of functionalities into devices will not necessarily scale according to Moore's Law but provide additional value to the end customer in different ways. The More than Moore approach will allow for the non-digital functionalities (e.q. RF communication, power control, passive components, sensors, actuators) to migrate from the system board-level into particular SiP/SoP potential solutions [119]. So far, we still could not tell which solution will dominate the future design trend, especially there are many design issues require engineers further discuss.

2.3 Circuit Design Issues: Modular Design

Recently, the growing complexity of multi–core architectures will soon require highly scalable communication infrastructure. Today most of

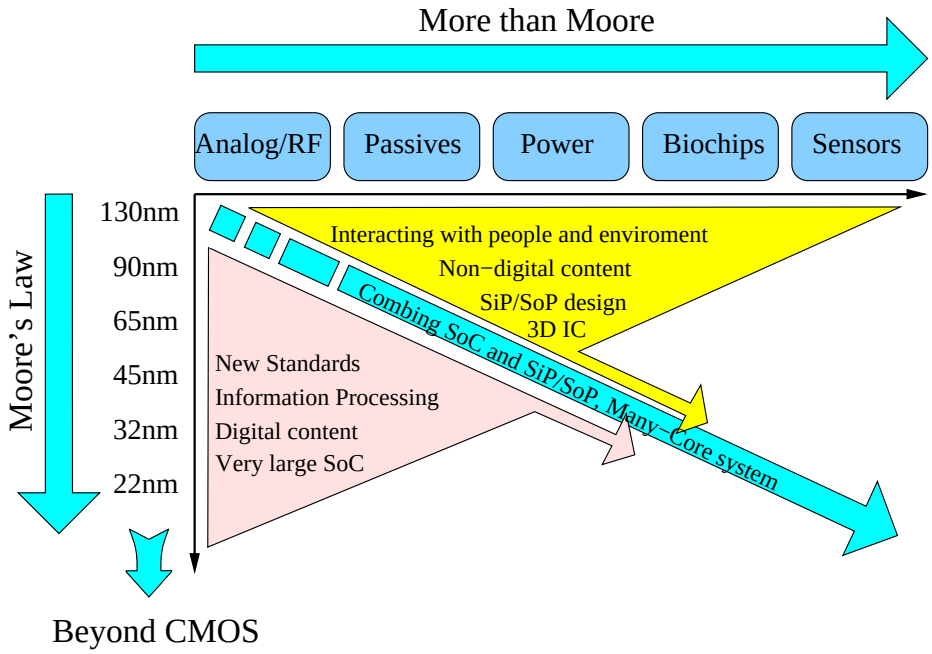


Figure 2.2: IC scaling roadmap for More than Moore (modified figure from 2009 International Technology Roadmap for Semiconductors Executive Summary) [58]

the current communication architectures in multi-core SoC are still based on dedicated wiring. With shirking process technology, logic components such as gates have also decreased in size. However, the traditional wiring lengths do not shrink accordingly, resulting in relatively longer communication path lengths between logic components. For instance, when the technology node is 65nm, the metal–layer wire delay is 10 times larger than the gate node delay as shown in Figure 2.3. Moreover, the data synchronization issue with a single clock source has also become a critical problem for circuit synthesis [14,51]. That means the timing closure issue on the large SoC design is difficult to be solved.

In the meantime, design resource reuse concerns all additional activities that have to be performed to generate an easy–to–use and flexible IP module. This is based on a hierarchical approach, which proceeds by partitioning a system into many small modules and requires compatibility and consistency. Proper system partitioning allows independence between the design of different modules. The decomposition is generally guided by structuring rules aimed at hiding local design decisions in such a way that only the interface of each module is visible. This kind of methodology is also called "a modular design". The overall modular approach can optimize the insertion of reusable IP component within

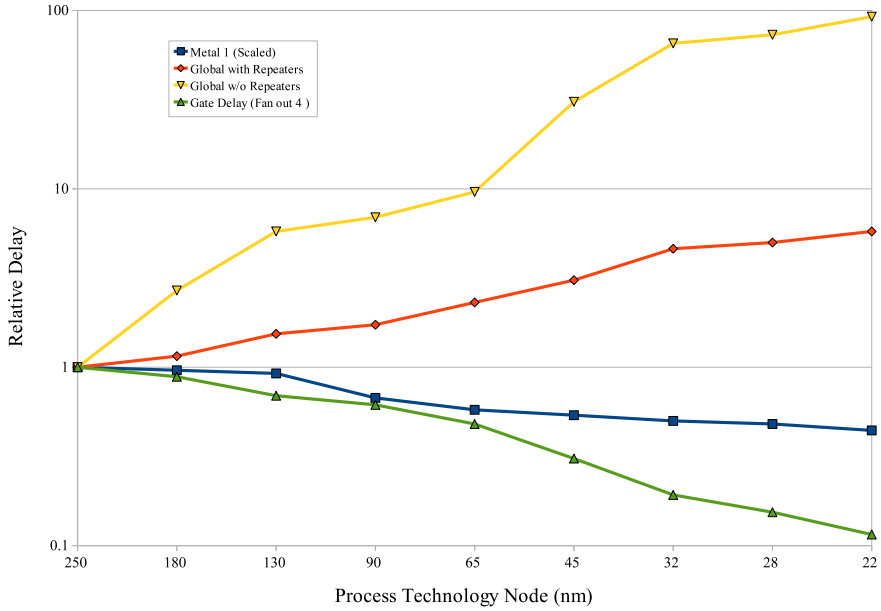


Figure 2.3: Relative delays of interconnection wire and gate in nanoscale level (regenerated figure from International Technology Roadmap for Semiconductors 2003) [57]

the circuit design.

As a result, ITRS has predicted for the next 20 years that a single SoC design will integrate more than one thousand IP components. They assumed the future die area will keep in a constant size and the number of cores will increase by a factor of 1.4 per year. Each processor core's operational frequency and its computational architecture will be both improved by a factor of 1.05 per year simultaneously. This means that the IP core performance will increase by a factor of 1.1025 per year. Figure 2.4 predicts a roughly 1000 times increase in a multi-core SoC system, which is the product number of IP cores and the frequency/performance. Therefore, the system performance with about 80-cores will increase about 20 times compared to an 8-cores implementation in 45nm technology in 2009. Note that these two anticipations are based on current 8-cores for general PC workstations and 2-core for mobile handheld devices.

In order to satisfy the needs for the very huge modular design (i.e. more than thousand IP cores), the flexible reusable interface and the nanoscale global wire delay problem, Network–on–Chip (NoC) was pre-

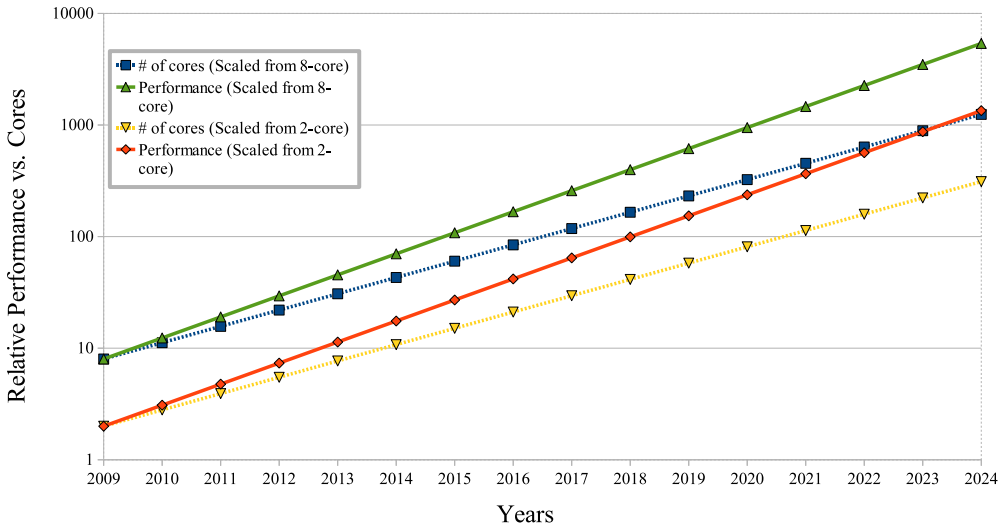


Figure 2.4: The prediction of future multi-core SoC performance (regenerated figure from 2009 International Technology Roadmap for Semiconductors System Drivers) [59]

sented as a new SoC paradigm to replace the traditional bus based on-chip interconnections by packet–switched network architecture [14, 121, 133]. It can yield reduced chip size and cost with higher interconnection efficiency. The components of an on-chip network (e.g. switching fabric, link circuitry, buffer and control logic) and the module interfaces, which are designed to be compatible with both heterogeneous IP cores and homogeneous Processing Elements (PEs), are interoperable and reusable.

The NoC can be used to structure the top-level wires on a chip and facilitate the implementation into a modular design. As shown in Figure 2.5, a typical multi-core system based on a mesh style network consists of a regular n×n array of tiles. Each tile could be a general-purpose processor, a DSP, a customized IP core or a subsystem. The network topology can be mesh, tours, ring, tree, irregular or hybrid style. A Network Interface (NI) is embedded within each tile for connecting itself with its neighboring tiles. The communication can be achieved by routing packets in a packet–switched network. This net-

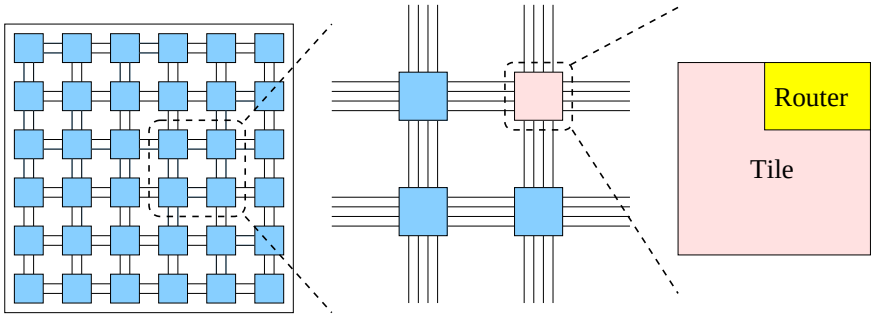


Figure 2.5: A typical NoC architecture with a mesh style packet-switched network

work is an abstraction of the communication among components and must satisfy Quality-of-Service (QoS) requirements, such as reliability and performance [51, 54].

2.4 Circuit Design Issues: Low Power

Besides the modular circuit design issue, power dissipation has also been considered as a critical constraint in the design of digital systems. One reason is the development of massively parallel computers, where hundreds of microprocessors are used. In such systems, power dissipation and required heat removal have become a major concern if each chip dissipates a large amount of power, which will cause heat and reliability problems. Therefore, a short review of the power aware methodology for each design level will be given.

The increasing prominence of multimedia portable systems and the need to limit power consumption in very-high density VLSI chips have led to rapid and innovative developments in low power design during the recent years [69,130]. The driving forces behind these developments are portable applications requiring low power dissipation, such as tablet computer, smart phone and portable embedded device. In most of these cases, the requirements of low power consumption must be met along with equally demanding goals of high performance and high throughput.

Meanwhile, the limited battery lifetime typically imposes very strict demands on the overall power consumption of these portable devices. Even new rechargeable battery types such as Nickel-Metal Hydride

(NiMH) have been developed with high energy capacity. So far, the energy density offered by the NiMH battery technology is about 2300-2700 mAh per AA size battery. It is still low in view of the expanding applications of portable devices. Unfortunately, revolutionary increase of the energy capacity is not expected in the near future. Therefore, low power and energy efficient computing has emerged as a very active and rapidly developing field of integrated circuit design.

2.5 Circuit Design Issues: Synthesis for Power Efficiency

In order to meet not only functionality, performance, cost-efficiency but also power-efficiency, automatic synthesis tools for IC design have become indispensable. The recent trend has considered power dissipation at all phases of the design levels. As we can see in Figure 2.6, large improvements in power dissipation are possible at the higher levels of design abstraction. The opportunities for reducing power consumption are higher if we start the design space from the system design level or the behavioral level. However, with the increasing power dissipation of VLSI, all possible power optimization techniques are used to minimize power dissipation at all levels.

- *System level* - For the first stage, the system architectural and topological choices are made, together with the boundary between hardware and software. This design phase is referred as hardware & software co-design. Obviously, at this stage, the design engineer has a very abstract view of the system. The most abstract representation of a system is the function it performs. A proper choice between the efficient algorithm and energy budget for performing the function (whether implemented in hardware or software) strongly affects system performance and power dissipation [13,53].

- *Behavioral level* - After determining the implementation of the function by hardware or software, this stage targets on the optimization of hardware resources and the optimization of the average number of clock cycles per task required to perform a given

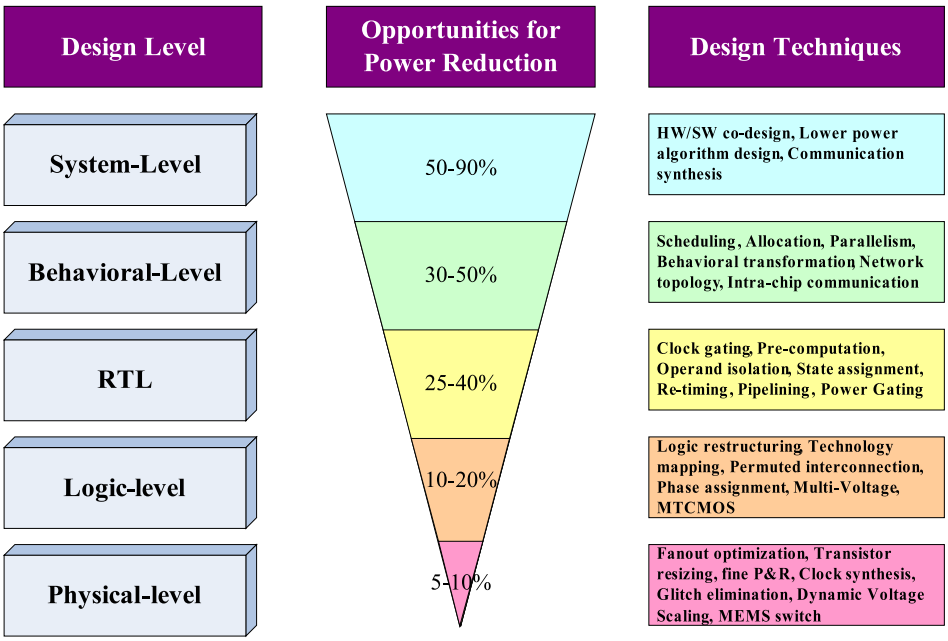


Figure 2.6: Power reduction at each design level [88]

set of modularized tasks [96]. Moreover, refined task arrangement for parallelism choosing an appropriate topology of the interconnection network also play important roles at this level.

- *RTL level* - RTL level design is the most common abstraction level for the manual design concept. The description is then transformed into logic gate implementation. At this level, all synchronous registers, latches and combinational logics between the sequential elements are described in a HDL program such as Verilog or VHDL. Moreover, the right choice of clock optimization strategy and pipelining will strongly affect the power consumption [67, 138].

- *Logic level* - The goal of this level is to generate a structural view of a logic-level model. Logic synthesis is the manipulation of logic specifications to create logic models as interconnection of logic primitives. Thus logic synthesis determines the micro structure of a circuit at gate-level. The task of transforming a logic model into an interconnection instance (netlist) of library cells (i.e. the back–

end logic synthesis tools), is often referred to as a library binding or a technology mapping [78]. At logic level, low power synthesis for a large SoC chip can be further reduced in average 10%–20% by applying these methodologies: Multi–Voltage, Multi–Threshold CMOS (MTCMOS) or Power Gating [20, 67].

- *Physical level* - In the last stage, the circuit representation is converted into a layout of the chip. Layout is created by converting each logic component (cells, macros, gates or transistors) into a geometric representation with specific shapes in multiple layers, which performs the intended logic function of the corresponding instance. Connections between different instances are also expressed as geometric patterns, typically lines, in multiple layers. Various power optimization techniques such as partitioning, fine placement, MEMS based power switch, transistor resizing, dynamic voltage scaling are employed [89, 92]. However, only 5%–10% power reductions could be obtained at this level.

2.6 Circuit Design Issues: Source of Power Dissipation

Power consumption in a CMOS technology can be described by a simple equation that summarizes the three most important contributors to its final value [15, 87].

$$P_{Total} = P_{Dynamic} + P_{Short} + P_{Leakage}. \tag{2.1}$$

These three components are dynamic power dissipation ($P_{Dynamic}$), short circuit power dissipation (P_{Short}) and leakage power dissipation ($P_{Leakage}$). $P_{Leakage}$ considers the static power consumption when the circuit is in static mode. This static power consumption is important for battery life in standby mode because the power is consumed whenever the device is powered up. P_{Short} and $P_{Dynamic}$ are both considered as dynamic power which is important for battery life when operating as it represents the power consumed when processing data.

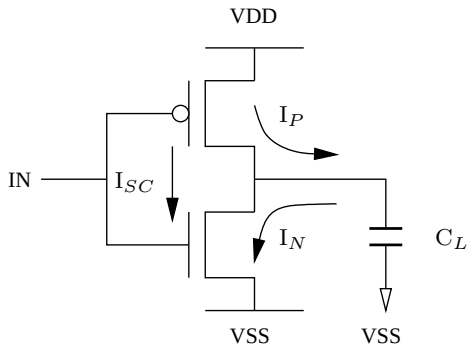


Figure 2.7: A simple CMOS inverter

2.6.1 Dynamic Power Dissipation

For the dynamic power consumption, Figure 2.7 illustrates the currents of a simple CMOS inverter. Assume that a pulse of data is fed into the transistor charging up and charging down the device. Power is consumed when the gate drives its output to a new value. It is dependent on the resistance values of the pmos transistor and the nmos transistor in the inverter. Hence, the charging and discharging of the capacitors result in the dynamic power consumption [131, 134]:

$$P_{Dynamic} = C_L(V_{DD} - V_{SS})^2 f\alpha. \tag{2.2}$$

When the ground voltage V_{SS} is assumed to be 0. It reduces to a better-known expression:

$$P_{Dynamic} = C_L V_{DD}^2 f\alpha, \tag{2.3}$$

where C_L is the loading capacitance at the output of the inverter, V_{DD} denotes the supply voltage and f is the clock frequency. These three parameters are primarily determined by the fabrication technology and circuit layout. α is the switching activity level and is dependent on the target applications (referred as the transition density), which can be determined by evaluating the logic function and the statistical properties of the input vectors. Table 2.1 lists the probability for the different kind of input singles. Obviously, Equation 2.2 shows that the dynamic power dissipation is proportional to the average switching activity, which means that it is influenced by the target application. In a

Signal	Activity (α)
Clock	0.5
Random data signal	0.5
Simple logic circuits driven by random data	0.4-0.5
Finite state machines	0.08-0.18
Video signals	0.1(MSB)-0.5(LSB)
Conclusion	0.05-0.5

Table 2.1: Typical switching activity levels [5]

typical case, dynamic power dissipation is usually the dominant fraction of total power dissipation (50%–80%).

2.6.2 Short Circuit Power Dissipation

Short-circuit currents occur when the rise/fall time at the input of a gate is larger than the output rise/fall time, causing imbalance and meaning that the supply voltage V_{DD} is short-circuited for a very short space of time. This will particularly happen when the transistor is driving a heavy capacity load. Fortunately, the short circuit current is manageable and can easily be avoided in a good design or synthesized by a well condition back–end logic synthesis tool. Therefore, the P_{Short} power dissipation is usually a small fraction (less than 1%) of the total power dissipation in CMOS technology.

2.6.3 Static Leakage Power Dissipation

The scaling of VLSI technology has provided the inspiration for many product evolutions as it gives a scaling of the transistor dimensions, as illustrated in Figure 2.8, where the length and the width are scaled by a factor of k. That means the new dimensions are given by $L = \frac{L}{k}$, $L = \frac{L}{k}$ and $T_{ox} = \frac{T_{ox}}{k}$. This will result in a transistor area reduction up to $\frac{1}{k^2}$ and also increase the transistor speed. Moreover, an expected decrease in transistor power dissipation as known as currents should be equally reduced.

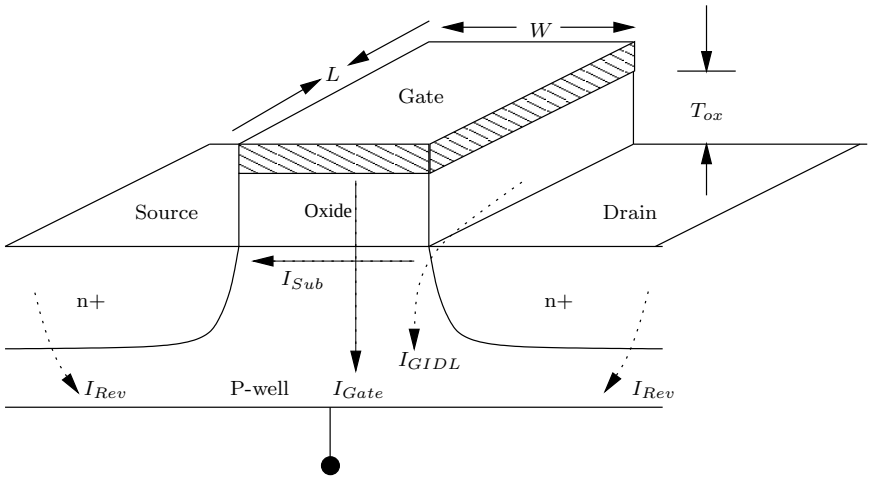


Figure 2.8: There are four components of leakage sources in NMOS: Subthreshold leakage (I_{Sub}), Gate-oxide leakage (I_{Gate}), Reverse biased junction leakage (I_{Rev}) and Gate Induced Drain Leakage (I_{GIDL})

However, when the node size is smaller than 65nm, a formerly ignorable gate leakage current (I_{Gate}) keeps raising explosively when the T_{ox} is reduced to $\frac{T_{ox}}{k}$ because the depth of gate oxide between first metal layer and P-well is too short. Fortunately, this problem had already been solved by using high-k dielectric materials to replace the conventional silicon based dioxide to improve the gate dielectric. This allows similar device performance, but with a thicker gate insulator, thus avoiding this leakage current. In a similar situation, the Reverse biased Junction (I_{Rev}) leakage and the Gate Induced Drain Leakage (I_{GIDL}) can both be suppressed efficiently in the manufacturing process with new materials.

On the other hand, in order to avoid excessively high electric fields in the scaled structure, the input voltage V_{DD} is required to be scaled. This forces a scaling in the threshold voltage V_T, too, otherwise the transistor will not turn off properly. In the past, subthreshold conduction was generally viewed as a parasitic leakage in a state that would ideally have no current. However, now the reduction in V_T will result in an increase of subthreshold drain current (I_{Sub}) with a direction from the drain to the source in a MOSFET. When the transistor is in the subthreshold region or weak-inversion region, it can be defined as [67]:

$$I_{Sub} = (\mu C_{ox} V_{th}^2 \frac{W}{L}) e^{\frac{V_{GS}-V_T}{nV_{th}}}, \tag{2.4}$$

where W and L are the dimension of the transistor, μ is a carrier mobil-

ity, C_{ox} is the gate capacitance, V_{th} is the thermal voltage kT/q (25mV at a room temperature), V_{GS} is the gate-source voltage, n is a number of the device manufacturing process with a range from 1.0 to 2.5. These parameters are considered as constant coefficients for subthreshold leakage problem. Therefore, the major coefficient is the exponent of $V_{GS} - V_T$. Once we decrease the V_{DD} and V_T during shrinking the size of the transistor nodes simultaneous, this results in an exponential increase of the subthreshold leakage power dissipation. In early VLSI circuits, I_{Sub} leakage was a small fraction (far less than 5%) of the total power dissipation. However, this expected decrease in power consumption now becomes a nightmare when the node size is smaller than 65nm (could be more than 50%). So far, the most efficient way to reduce the I_{Sub} leakage power is power gating, when the entire macro is turned off.

2.7 Design Consideration for Iterative Algorithms

With aforementioned design issues, we have to consider the relationship between iterative algorithms and design criteria for circuit implementation. First of all, as mentioned in Section 2.4, we already know that the system level stage provides the most opportunities to reduce the power dissipation. On the other hand, according to the source of power dissipation in Section 2.6, the major sources of power dissipation are dynamic and leakage power dissipations in CMOS circuit. Therefore, to design a low power iterative architecture that can reduce both dynamic and static power dissipations significantly at the system level is one of the major topics in this thesis. In the following chapters, a VLSI design concept will be clarified by four different iterative algorithms/methodologies. These hardware solutions not only balance with the circuit design criteria (area, timing and power) but also retain the good quality of results.

Iterative algorithms usually have a common character, more precise computation or approximation per iteration step will result in more area overhead for iterative hardware core. On the one hand, less precise iteration steps will cause slower convergence property. On the other hand, more precise iteration steps will consume more energy. Here we

will elaborate two iterative examples, CORDIC based QDCIT transformation for video compression and parallel Jacobi method for EVD. They will be handled carefully with a balance between the cost/area, convergency/timing and energy/power.

On–chip network emerges for the next generation SoC and becomes more and more important. With the need for supporting many multimedia standards in a single chip, it has been predicted more than one thousand processor units will be integrated together in the future. However, current bus methodology is facing a critical challenge of data switching for supporting large scale SoC, especially when these processor units are heterogeneous. Moreover, the timing closure will become extremely difficult to be obtained. Therefore, the choice between the ordinary bus-based system and switching based network is a critical design issue especially when the data transfers are irregular. Later, we will show how to utilize this switching feature for Sparse Matrix-Vector Multiplication (SMVM) when solving systems of linear equations iteratively.

2.8 Summary

In this chapter, a brief introduction to the concerns of today's nanoscale VLSI design concepts was given. It was shown that dealing with the power dissipation problem became one of the important tasks concerning an ASIC development. It will cause very huge efforts for both verification and implementation when the power sources are not guaranteed. Therefore, design engineers must carefully think about the relationship between the design criteria (area, timing and power/energy) and the proper way how to realize iterative algorithms in hardware. In next chapter, we will start to discuss the relationship between these important design issues by a well–known iterative algorithm, "CORDIC" algorithm, which can be used to accelerate many digital functions and applications in signal processing.

3 CORDIC Algorithm

COordinate Rotation DIgital Computer (CORDIC) is a typical iterative algorithm which is used to accelerate many digital functions and applications in signal processing. In this chapter, a brief introduction to the generic definition of the CORDIC algorithm with orthogonal rotation mode will be given first in Section 3.1. Then the extension of CORDIC to linear and hyperbolic modes will be further explained in Section 3.2. In Section 3.3, three different dependence flow graphs for the CORDIC hardware implementation will be described and compared in Section 3.4.

3.1 Generic CORDIC Algorithm

Digital Signal Processing (DSP) algorithms exhibit an increasing need for the efficient implementation of complex arithmetic operations. The computation of trigonometric functions, coordinate transformations or rotations of complex valued phases are almost naturally involved with modern DSP algorithms. Popular application examples are algorithms used in digital communication technology and in adaptive signal processing. There are many applications using the CORDIC algorithm, such as solving systems of linear equations [2, 4, 55, 60], computation of eigenvalues and singular values [30,38,102], Discrete Wavelet Transform (DWT) [28,103], Discrete Cosine Transform (DCT) [75,112] and digital filters [31, 32, 115]. The CORDIC algorithm offers the opportunity to calculate all the desired functions and applications in a rather simple and elegant way in circuit design [6].

The CORDIC algorithm was first presented by Jack Volder in 1959 [122] for the computation of trigonometric function, multiplication, division, data type conversion, and later generalized to hyperbolic function

Table 3.1: Three different rotation types of CORDIC with both rotation and vector modes used for implementing the digital processing algorithms (said n iterations and rotated with a target angle ϕ_t)

Mode	Operation
Orthogonal Rotation	$x_n = x_0 \cos\phi_t - y_0 \sin\phi_t$ $y_n = y_0 \cos\phi_t + x_0 \sin\phi_t$ $z_n = 0$
Orthogonal Vector	$x_n = \sqrt{x_0^2 + y_0^2}$ $y_n = 0$ $z_n = \arctan(y_0/x_0) = \tan^{-1}(y_0/x_0)$
Linear Rotation	$x_n = x_0$ $y_n = y_0 + x_0 \cdot z_0$ $z_n = 0$
Linear Vector	$x_n = x_0$ $y_n = 0$ $z_n = z_0 - y_0/x_0$
Hyperbolic Rotation	$x_n = x_0 \cosh\phi_t - y_0 \sinh\phi_t$ $y_n = y_0 \cosh\phi_t + x_0 \sinh\phi_t$ $z_n = 0$
Hyperbolic Vector	$x_n = \sqrt{x_0^2 - y_0^2}$ $y_n = 0$ $z_n = \operatorname{arctanh}(y_0/x_0) = \tanh^{-1}(y_0/x_0)$

by Walther [124]. A large variety of operations can be easily realized by the structure of CORDIC algorithm [50,56,83]. There are three primary types of CORDIC algorithm, the orthogonal, the linear and the hyperbolic. Each type has two basic operation modes, rotation and vector, which are summarized in Table 3.1. The CORDIC algorithm can be realized as an iterative sequence of add, sub and shift operations. Due to the simplicity of the involved operations, the CORDIC algorithm is very well suited for VLSI implementation.

A CORDIC algorithm provides an iterative method of performing vector rotations. The general form of the orthogonal rotation CORDIC

mode is defined as:

$$\begin{aligned} x' &= x\cos\phi_t - y\sin\phi_t \\ y' &= y\cos\phi_t + x\sin\phi_t, \end{aligned} \tag{3.1}$$

which rotates the input vector $< x, y >$ in the Cartesian coordinate system by a rotation angle ϕ_t. Then this equation can be rearranged as:

$$\begin{aligned} x' &= \cos\phi_t(x - y\tan\phi_t) \\ y' &= \cos\phi_t(y + x\tan\phi_t). \end{aligned} \tag{3.2}$$

To further simplify it, we can restrict the rotation angle $\tan\phi_t$ to $\tan\phi_t = \pm 2^{-i}$, such that the multiply operation by the tangent part is reduced to shift operations. This alters Equation 3.2 to become a sequence of arbitrary elementary rotations. Since the decision at each iteration i is fixed, the $cos2^{-i} = cos2^{i}$ becomes a constant number. The new iterative rotation can now be defined as:

$$\begin{aligned} x_{i+1} &= K_i(x_i - y_i \cdot d_i \cdot 2^{-i}) \\ y_{i+1} &= K_i(y_i + x_i \cdot d_i \cdot 2^{-i}) \\ K_i &= \cos(\tan^{-1} 2^{-i}) = \tfrac{1}{\sqrt{1+2^{-2i}}} \\ d_i &= \pm 1 \\ i &= 0, 1, 2, 3, \ldots n. \end{aligned} \tag{3.3}$$

Removing the scaling factor K_i from the iterative part yields a simple shift-add algorithm for orthogonal rotation CORDIC. After several iterations, the product of K_i factors will converge to a constant coefficient 0.607252. The exact gain depends on the number of iterations: $A_n = \prod_n \sqrt{1 + 2^{-2i}} \approx 1.646762 = \frac{1}{K_n}$. In the mode of orthogonal rotation, the angle accumulator is added to the Equation 3.4.

$$\begin{aligned} x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\ y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot \tan^{-1} 2^{-i} \\ d_i &= -1 \text{ when } z_i < 0 \text{ else } +1. \end{aligned} \tag{3.4}$$

The angle accumulator is initialized with the input rotation angle. The rotation direction at each iteration i is decided by the magnitude of the residual angle in the angle accumulator. If the residual angle is

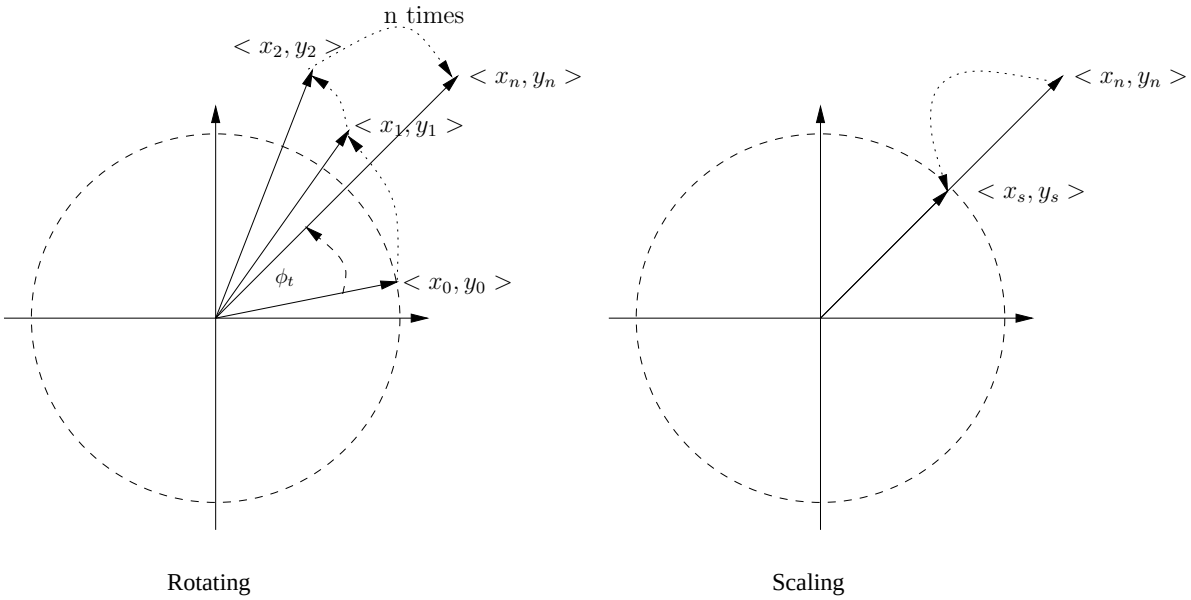


Figure 3.1: CORDIC rotating and scaling a input vector $< x_0, y_0 >$ in the orthogonal rotation mode

positive, then the d_i is set to +1 otherwise -1. After serveral iterations, it will produce the following results:

$$\begin{aligned} x_n &= A_n(x - y \tan z_0) \\ y_n &= A_n(y + x \tan z_0) \\ z_n &= 0. \end{aligned} \tag{3.5}$$

In Figure 3.1, an example for the generic CORDIC processor rotating an input vector $< x_0, y_0 >$ with a target rotation angle ϕ_t is shown. The CORDIC processor rotates it by the desired rotation angle iteratively, said n times (usually $n = 32$ with the single floating precision). After that, a constant scaling value $K = 0.607252$ will be applied to the rotated vector $< x_n, y_n >$ in order to scale the rotation results.

Moreover, for the orthogonal vector mode, the CORDIC rotates the input vector through whatever angle is necessary to align the resulting vector with the x axis. That means the direction d_i is dependent on the current y_i instead of z_i. Equation 3.4 can be modified as:

$$\begin{aligned} x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\ y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot \tan^{-1} 2^{-i} \\ d_i &= -1 \text{ when } y_i \geq 0 \text{ else } +1. \end{aligned} \tag{3.6}$$

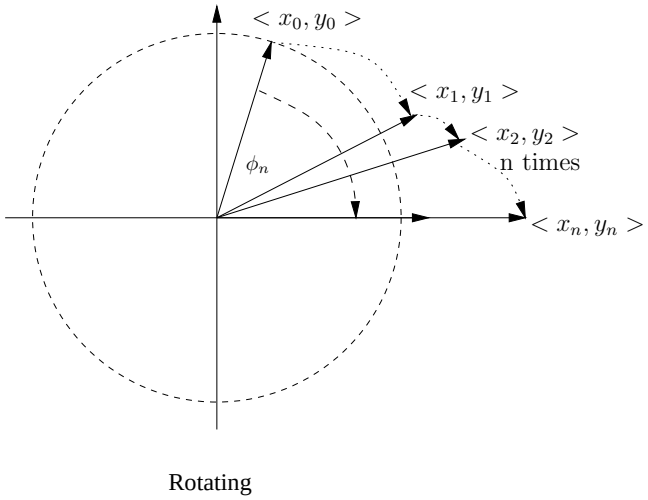


Figure 3.2: CORDIC rotating a input vector $< x_0, y_0 >$ in the orthogonal vector mode

Then:

$$\begin{aligned} x_n &= \sqrt{x_0^2 + y_0^2} \\ y_n &= 0 \\ z_n &= z_0 - d_i \cdot \tan^{-1} 2^{-i}. \end{aligned} \tag{3.7}$$

The CORDIC with orthogonal vector mode can obtain the arctangent result, $\phi = \arctan(\frac{y}{x})$, if the angle accumulator is initialized with zero ($z_0 = 0$). Since the arctangent result is taken from the angle accumulator, the scaling factor A_n can be ignored and does not affect the result.

$$z_n = \arctan(\tfrac{y_0}{x_0}). \tag{3.8}$$

Figure 3.2 shows an example when a CORDIC processor rotates an input vector $< x_0, y_0 >$ to the x axis iteratively. After n iterations, the ϕ_n=arctan($\frac{y_0}{x_0}$), will converge into the z_n accumulator. Note that the constant scaling value $K = 0.607252$ for correcting the $< x_n, y_n >$ is not necessary if we only need the arctangent result.

3.2 Extension to Linear and Hyperbolic functions

A simple modification to Equation 3.4 allows the computation of function in the linear coordinate system using the same architecture:

$$\begin{aligned} x_{i+1} &= x_i - 0 \cdot y_i \cdot d_i \cdot 2^{-i} = x_i \\ y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot 2^{-i} \\ i &= 1, 2, 3, 4, \ldots n. \end{aligned} \tag{3.9}$$

For the linear rotation mode, d_i = -1 if $z_i < 0$ else +1, the linear rotation CORDIC produces:

$$\begin{aligned} x_n &= x_0 \\ y_n &= y_0 + x_0 \cdot z_0 \\ z_n &= 0. \end{aligned} \tag{3.10}$$

This operation is more and less like a multiplier but requires only the shift and add operations. It is a very efficient approximated solution for iterative hardware design [82]. Of course there are more efficient implementations of MAC like structures and especially multipliers, but if a CORDIC is already available it may be used as such [53]. On the other hand, if we redefine d_i = -1 if $y_i \geq 0$ else +1, the linear vector mode results will be converted as:

$$\begin{aligned} x_n &= x_0 \\ y_n &= 0 \\ z_n &= z_0 - \tfrac{y_0}{x_0}. \end{aligned} \tag{3.11}$$

On the contrary, the linear vector mode now can approximate a divide operation iteratively, which is very important for DSP system. Note that the rotations in the linear coordinate system have only a unity gain, so there is no scaling factor for compensation. In Appendix B, two linear CORDIC examples with the rotation mode and the vector mode are described in Figure B.1 and Figure B.2.

The difference between trigonometric function and hyperbolic function is very small, they can share the same CORDIC architecture with a little modification to Equation 3.4. In this case the formulas for the hyperbolic rotation mode are:

$$\begin{aligned} x_{i+1} &= x_i + y_i \cdot d_i \cdot 2^{-i} \\ y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot \tanh^{-1} 2^{-i} \\ i &= 1, 2, 3, 4, 4, 5, \ldots 12, 13, 13, 14, \ldots n, \end{aligned} \tag{3.12}$$

where d_i = -1 if $z_i < 0$ else +1, then the hyperbolic rotation CORDIC will produce:

$$\begin{aligned} x_n &= A_n(x + y \tanh z_0) \\ y_n &= A_n(y + x \tanh z_0) \\ z_n &= 0 \\ A_n &= \textstyle\prod_n \sqrt{1 - 2^{-2i}} \approx 0.80. \end{aligned} \tag{3.13}$$

Moreover, for the hyperbolic vector mode, $d_i = +1$ if $y_i < 0$ else -1, then the hyperbolic vector CORDIC produces:

$$\begin{aligned} x_n &= \sqrt{x_0^2 - y_0^2} \\ y_n &= 0 \\ z_n &= z_0 + \tanh^{-1} \tfrac{y_0}{x_0}. \end{aligned} \tag{3.14}$$

It should be noticed that the rotations in the hyperbolic coordinate system do not converge. It will only converge if we select a particular iteration sequence, i.e. $i = 4, 13, 40, \ldots, k, 3k + 1, \ldots$ are repeated once [2, 124]. The hyperbolic equivalents of all the functions as the circular coordinate system can be computed in a similar fashion. In Appendix B, two hyperbolic CORDIC examples for the rotation mode and the vector mode are described in Figure B.3 and Figure B.4. Essentially, these six different rotation modes can be summarized into an unified formula as (i.e. orthogonal rotation, orthogonal vector, linear rotation, linear

vector, hyperbolic rotation and hyperbolic vector):

$$\begin{aligned} x_{i+1} &= x_i - m \cdot y_i \cdot d_i \cdot 2^{-i} \\ y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot e_i \\ e_i &= \begin{cases} \tan^{-1} 2^{-i} & \text{when } m = 1 \\ 2^{-i} & \text{when } m = 0 \\ \tanh^{-1} 2^{-i} & \text{when } m = -1 \end{cases}, \end{aligned} \tag{3.15}$$

where e_i is the elementary angle of rotation for iteration i in the selected coordinate system. For orthogonal mode in the Cartesian coordinate system, the $e_i = \tan^{-1} 2^{-i}$ and the m=1. For linear mode in the linear coordinate system, the $e_i = 2^{-i}$ and the m=0. For hyperbolic mode in the hyperbolic coordinate system, the $e_i = \tanh^{-1} 2^{-i}$ and the m=-1. This universal representation, credited to Walther, permits the design of a general purpose CORDIC rotation processor. However, the focus of this thesis will be only on the orthogonal rotation and the orthogonal vector modes.

3.3 CORDIC in Hardware

In this section, three different dependence flow graphs are presented for the hardware implementation of the CORDIC. Note that we restrict only to the conventional CORDIC iteration scheme as shown in Equation 3.15. In Figure 3.3, the structure of one stage of the CORDIC iteration is presented, which requires a pair of adders for rotation and another adder for steering the next angle direction (d_i). All internal variables are buffered in the registers separately until the iteration number is large enough to obtain convergence. The signs of all three intermediate variables are fed into a control unit which generates the rotation direction flags d_i to steer the add or sub operations and keeps tracking of the rotation angle z_i. This folded dependence graph is typical for the orthogonal rotation mode and has its benefit in small area for VLSI design. However, the throughput is very low and requires an extra storage (lookup table) to store the rotation angles for following iterations. Moreover, it requires a number of different shifting lengths according

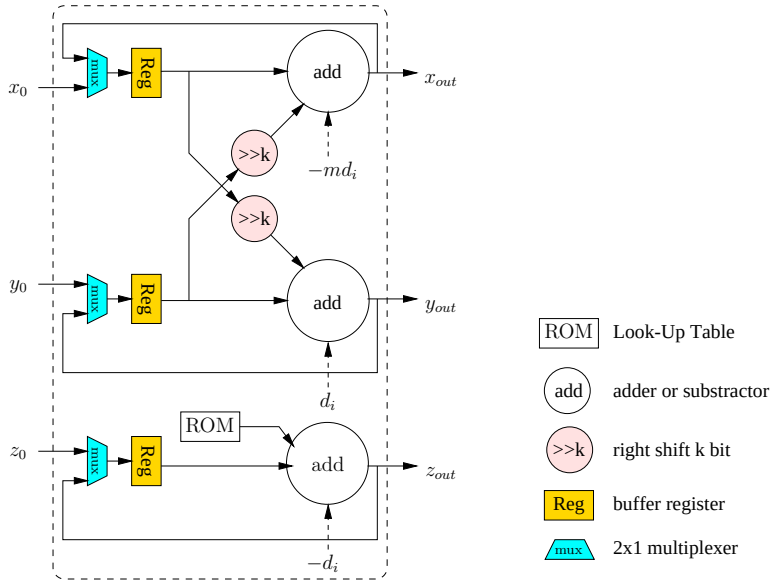


Figure 3.3: Flow graph of a folded CORDIC (recursive) processor

to the chosen shift sequence, additional barrel shifters are required in folded CORDIC architecture.

We can unfold it to improve the throughput and obtain a new flow graph where $n = 32$ stages are cascaded together as shown in Figure 3.4. Note that the fixed shift operations are assumed to be hardwired, hence they do not represent any propagation delay or further hardware resources. In this way, the lookup table is not required and each stage has a specified constant number for steering the next stage's angle direction.

Besides having a purely combinational implementation, buffers can also be inserted between successive stages as indicated in Figure 3.5. In this way, the throughput of the unfolded CORDIC can be improved by using a pipelined architecture. Table 3.2 lists the overall comparison for each flow graph in terms of area, speed and power consumption. It can be noticed that the folded CORDIC is smaller than the other two flow graphs but the processing speed is much slower. On the contrary, the pipelined CORDIC achieves higher throughout but requires larger area overhead. Therefore, the choice between folded, unfolded and pipelined simply becomes a tradeoff problem. Design engineers must further consider which solution is best for their system design.

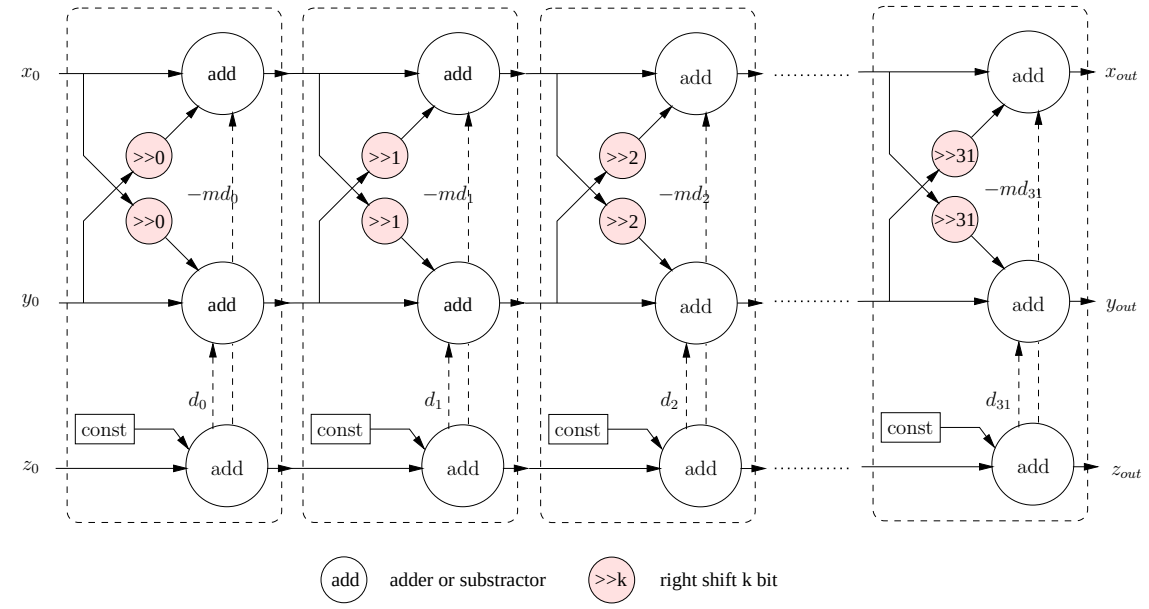


Figure 3.4: Flow graph of an unfolded (parallel) CORDIC processor

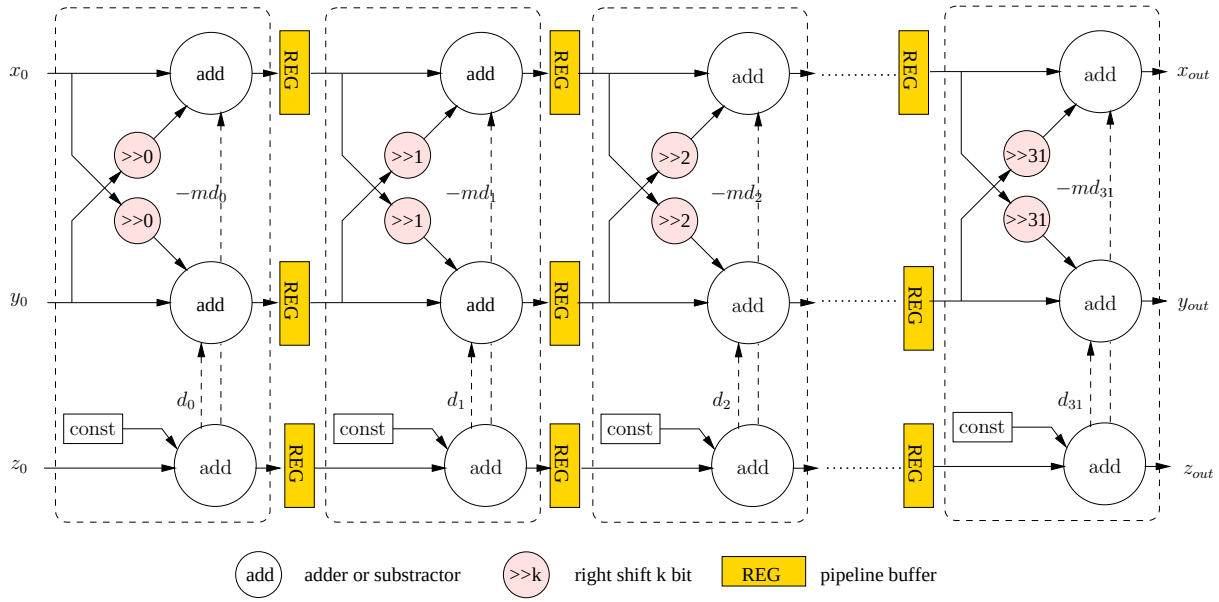


Figure 3.5: Flow graph of an unfolded (parallel) CORDIC processor with pipelining

Table 3.2: Comparison of three different CORDIC dependence flow graphs

Mode	Area	Speed	Power
folded	small	very slow	low
unfold	large	slow	high
pipeline	very large	very fast	very high

Table 3.3: Implementation results of three different CORDIC dependence flow graphs with the orthogonal rotation mode in Xilinx Virtex–5 FPGA (xc5vlx110t-1ff1136)

	folded	unfold	pipeline
Slice Regs	502	162	3,204
Slice LUTs	477	3,528	3,106
Timing	7.340 ns	94.732 ns	5.347 ns
Frequency	136.2 MHz	10.6 MHz	187 MHz
Throughput (MByte)	34.04 MB	84.8 MB	1496 MB
Power Consumption	27.44 mW	14.92 mW	53.38 mW
Energy (per byte)	0.8061 nJ	0.1759 nJ	0.0357 nJ

3.4 Hardware Performance Analysis

In order to further analyze these three different dependence flow graphs, they are directly generated from the Xilinx IP core library and synthesized by the Xilinx ISE 11.5 in Virtex–5 FPGA, where Table 3.3 lists the synthesis results. Note that each generated CORDIC hardcore requires 32 iterations and the input/output word length is 32 bits.

First of all, the area size of the folded CORDIC is very small compared to the others, but the throughput is very low. Although the pipelined CORDIC can achieve higher throughput, the area overhead is very high (due to cascaded units and pipeline buffers) and the power consumption is also high. Second, the timing delay of the unfold CORDIC flow graph is much larger than the others due to its very long data path for clock tree synthesis. Third, if we further consider the energy consumption per byte, the pipelined CORDIC is much more efficient than the folded CORDIC, because the computation period is shorter. Finally, the FPGA synthesis results show that the pipelined CORDIC architecture can achieve higher throughput and low power but require more logical utilization resources in FPGA. On the other hand, the folded CORDIC is small and simple. However, it is very slow and not efficient concerning the energy consumption.

3.5 Summary

In this chapter, a brief introduction to the fundamental CORDIC algorithm with different rotation modes and the corresponding VLSI circuit design of three dependence flow graphs was given. The folded CORDIC flow graph is smaller than the pipelined CORDIC but the throughput is slow. Furthermore, the energy consumption is also higher. Therefore, the design tradeoff is between the throughput and the energy consumption. The choice is dependent on the target application. Although only the orthogonal rotation and orthogonal vector modes are used in this thesis, the presented VLSI design concepts can also be applied to other CORDIC rotation modes.

4 Discrete Cosine Integer Transform (DCIT)

In this chapter, the integration of DCT and integer transform on a configurable IP Core for supporting multi–standard image/video Codecs is presented. In Section 4.1, a short review on today's video transformation and quantization will be given first. Section 4.2 briefly introduces the algorithms of the DCT, Loeffler DCT, CLDCT and the integer transforms. In Section 4.3, a configurable forward and inverse DCIT for supporting the multiple transformations is presented. Later, the framework of the proposed CORDIC based Quantized DCIT architecture on a schematic design level will be described in Section 4.4. The implementation results are shown in Section 4.5. Section 4.6 summarizes this chapter.

4.1 Introduction of DCIT

As the demand for multi-function devices has been growing severely in recent years, more and more requirements have been posed for SoC design, such as low power, high performance, quality awareness and multi-standard integration. The DCT transformation is a very important video coding component of many modern Image/Video compression standards (e.g. JPEG, MPEG–2, MPEG–4, H.263 and so on [26, 94]). In H.264 Codec, which is a joint standard of the ITU-T video compression and the ISO/IEC MPEG-4 Part 10 Advanced Video Coding (AVC), the DCT part is replaced by an integer transform using a block size of 4×4 pixels. For adding profiles for High Definition (HD) videos, an integer transform using a block size of 8×8 pixels has been added [74]. Recently, the H.264/AVC has been extended to support Scalable Video Coding (SVC) by Joint Video Team of ITU-T VCEG

and ISO MPEG (JVT) [100, 101]. Furthermore, Ultra–HD (UHD) TV, producing a 7,680×4,320 pixel resolution (a.k.a. Super Hi-Vision) and the next generation High Efficiency Video Coding (HEVC), will soon need very high throughput performance [25, 79]. Therefore, the multi-function integration of different Codecs with very high throughput is a challenge to design engineers implementing the Codecs for supporting 3D/UHD displays.

A low–power and high-quality CORDIC based Loeffler DCT (CLDCT) is presented. It can reduce the computational complexity from 11 mul and 29 add operations to 38 add and 16 shift operations [112]. It obtains a transformation quality as good as the original Loeffler DCT. Furthermore, 4–point integer transform and 8–point integer transform are integrated in this CLDCT [106], such that a configurable architecture for *Discrete Cosine and Integer Transform (DCIT)* is obtained. It can be configured to perform the multiplierless One–Dimensional (1–D) 8–point DCT and the 1–D 4–point/8–point integer transforms in multi-standard Video Codecs.

In general, the Two–Dimensional (2–D) DCT coefficients are usually followed by a quantization procedure in image/video compressions. Quantization can be uniform or nonuniform. The uniform quantization method is typically implemented as a division by a quantization step size followed by a rounding operation. The nonuniform quantization method can be implemented in a large lookup table or by utilizing a search algorithm in a code book. Incorporating the quantization into the DCT transformation, resulting in Quantized DCT (QDCT), has been considered as an efficient way to improve the computational complexity for video compression. Essentially, only the uniform quantization can be incoperated into the DCT transformation due to its regular structure, which is more efficient for VLSI implementation. In the literature, Alen Docef et al. [34] proposed a joint implementation of Chen's DCT [23] and Quantization (i.e. a conventional QDCT design). Later, Hanli Wang et al. [126] merged the quantization process represented by a quantization matrix that has variable quantization step sizes into Chen's DCT. They called it Novel QDCT (NQDCT). However, both QDCT designs require multipliers to perform DCT and Quantization. Even if the multiplier could be replaced by Canonical Signed Digit (CSD) representation, the QDCT and NQDCT will need more than 300 add operations, in the worst case.

In this chapter, instead of realizing the QDCT architecture with multiplications, CORDIC compensation steps are used to approximate the quantization procedure iteratively. As for CLDCT in [112], the number of CORDIC compensation iterations is reduced, resulting in a CORDIC based Scaler (CORDIC-Scaler). This CORDIC-Scaler has five stages and requires 8 adders and 10 shifters to approximate two scaling operations. The combination of two 1–D DCITs, a row-column transposition memory and four CORDIC-Scalers forms a 2–D Forward QDCIT (FQDCIT) which requires 120 adders and 40 shifters. On the other hand, an extended version of Inverse QDCIT (IQDCIT) has also been modified with seven configurable modules which requires 124 adders and 40 shifters. The proposed core can save more than the half number of adders by using a shared configurable architecture. This CORDIC based IQDCIT can support arbitrary scaling values for quantization by updating the lookup tables for different Codecs which are based on scaler dequantizer, such as JPEG, MPEG–4, H.264 or DivX. After that, different numbers of CORDIC iterations are applied to the Inverse DCIT (IDCIT) and different numbers of compensation steps to the CORDIC-Scaler, respectively, because in most image/video decoders, the required transformation precision is dependent on the target resolution. For example, the concept of dynamic Bit-width adaptation by decreasing the bit length of the high frequency coefficients in DCT has also been adopted into our design [84]. Therefore, it will perform fewer iterations when the target resolution is small (QCIF/CIF). On the contrary, when the target resolution is large (Full-HD/Ultra-HD), more iterations will be applied. Providing arbitrary accuracy for various resolutions is important for embedding video decoders in any consumer electronic devices.

4.2 DCT algorithms

4.2.1 The DCT Background

A 2–D DCT can transform an 8×8 block sample from spatial domain $X(i, j)$ into frequency domain $Y(k, l)$ as follows:

$$Y(k,l) = \tfrac{1}{4}C(k)C(l)\sum_{i=0}^{7}\sum_{j=0}^{7}X(i,j)\cdot\cos[\tfrac{(2i+1)k\pi}{16}]\cos[\tfrac{(2j+1)l\pi}{16}]$$

$$C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } k = 0 \\ 1 & \text{otherwise} \end{cases} \quad C(l) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } l = 0 \\ 1 & \text{otherwise} \end{cases} \tag{4.1}$$

k and $l = 0, 1, 2, \ldots 7$.

Since computing the above 2–D DCT by using matrix multiplication requires 8^4 multiplications, a commonly used approach to reduce the computational complexity is row-column decomposition. This decomposition performs row-wise 1–D transformation followed by column-wise 1-D transformation with intermediate transposition. This approach has two advantages. First, the computational complexity is significantly reduced. Second, the original 1–D DCT can be easily replaced by different DCT algorithms. A 1–D 8–point DCT can transform 8 samples $X(i)$ into the frequency domain $Y(k)$ as follows:

$$Y(k) = \tfrac{1}{2}C(k)\sum_{i=0}^{7}X(i)\cos[\tfrac{(2i+1)k\pi}{16}]$$

$$C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if k} = 0 \\ 1 & \text{otherwise} \end{cases} \tag{4.2}$$

$$k = 0, 1, 2, \ldots 7.$$

4.2.2 The CORDIC based Loeffler DCT

Lately, a simplified 1-D 8–point CLDCT has been presented requiring only 38 add and 16 shift operations [112]. The original Loeffler DCT is selected as the starting point for optimization as shown in Figure 4.1. The trigonometric plane rotation with the angle $3\pi/8$, $3\pi/16$ and $\pi/16$, can be calculated using only 3 mul and 3 add operations instead of 4 mul and 2 add operations using the equivalence function. Hence, including

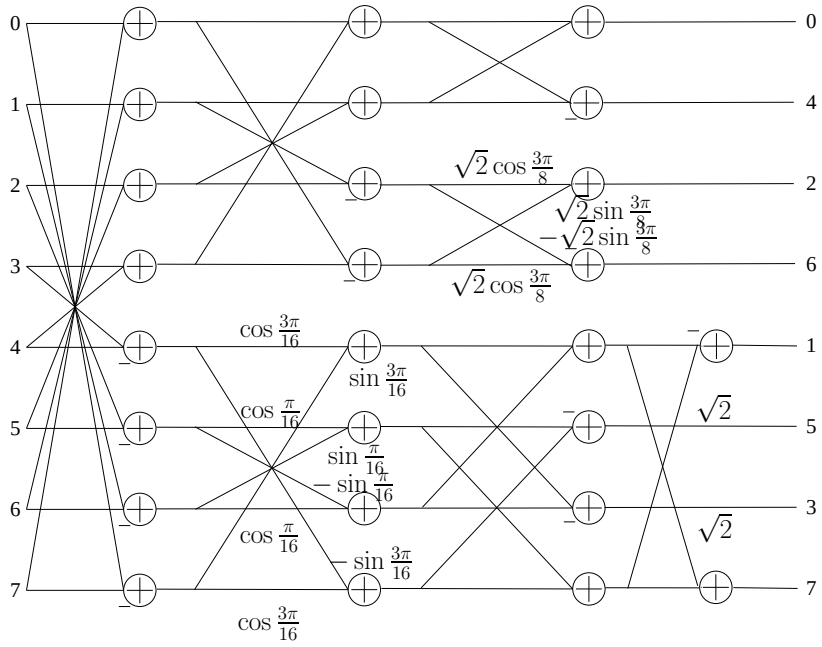


Figure 4.1: Flow graph of an 8–point Loeffler DCT architecture

two $\sqrt{2}$ multiplications, a 1-D Loeffler DCT requires in total 11 mul and 29 add operations. In the meantime, the theoretical lower bound of the number of multiplications required for the 1-D DCT had also been proven to be 11 [36, 73].

Implementing the Loeffler DCT with the CORDIC method by ignoring some unnoticeable iterations and shifting the compensation steps of each angle to the quantizer yields the simplified CLDCT. Figure 4.2 shows the RTL flow graph of the CLDCT, where the white circles are adders, and the dark circles are shifters, including eight scaling factors incorporated into the quantization table. Here, Module 1 represents an orthogonal CORDIC rotation with the angle $\phi = 3\pi/8$. The number of CORDIC iterations is reduced to three with 6 adders and 4 shifters. All compensation steps are shifted to the quantizer. For Module 2, it is a simplification of an orthogonal CORDIC rotation with the angle $\phi = \pi/16$, the compensation iterations of the $\pi/16$ rotation are first ignored and then the iterations are reduced to 4 adders and 4 shifters. For Module 3 with the rotation angle $3\pi/16$, the number of iterations are reduced to 4 adders and 4 shifters. Since there is a data correlation between the subsequent stages of the $\pi/16$ and the $3\pi/16$, two CORDIC compensation steps with 4 adders and 4 shifters are still required to obtain the correct correlation. Although simplifying these rotation angles

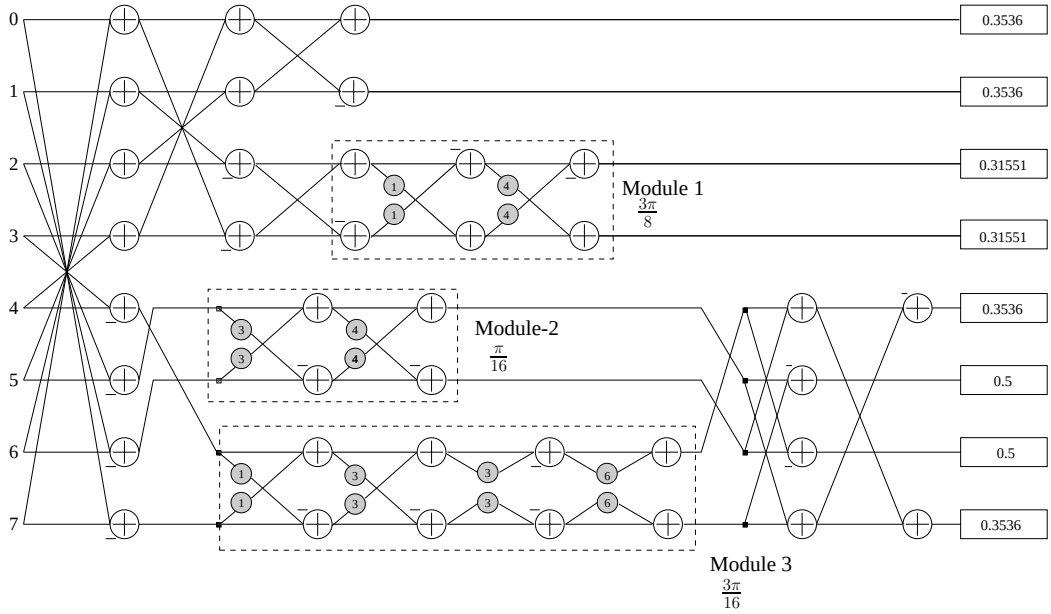


Figure 4.2: Flow graph of an 8–point CORDIC based Loeffler DCT architecture

will decrease the quality of the transformation results, the influences are not noticeable in video sequence streams or image compression. This DCT architecture not only has a similar computational complexity as the binDCT [118] but also keeps the transformation quality as good as Loeffler DCT. More detailed information about this low-power and high-quality DCT can be found in [42–44, 112].

4.2.3 4×4 Integer Transform

The DCT transformation was used in all MPEG-Standards up to the MPEG-4 Visual standard. Since H.264, it has been replaced by a 4×4-integer transform. This transform is implemented similar to the DCT with the difference that all the operations can be directly implemented as integer arithmetics without any loss of accuracy. In H.264, a 4×4

integer transform is defined as:

$$
\begin{aligned}
Y &= (C_f X C_f^T) \otimes E_f \\
&= \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} [X] \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \end{bmatrix},
\end{aligned}
\tag{4.3}
$$

where the size of input matrix X is 4×4 and $C_f X C_f^T$ is the 2–D integer transform. E is a matrix of scaling factors where a=$\frac{1}{2}$, b=$\sqrt{\frac{2}{5}}$ and d=$\frac{1}{2}$. Note that the operator $\otimes$ represents the multiplication of two matrices element by element (i.e. C = A $\otimes$ B means $c_{ij} = a_{ij} \cdot b_{ij}$, $\forall i, j$). The corresponding formula of the inverse transform is given by [74, 94]:

$$
\begin{aligned}
X' &= C_i^T (Y \otimes E_i) C_i \\
&= \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} \left([Y] \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix}.
\end{aligned}
\tag{4.4}
$$

Although the inverse integer transform contains shift operations to the right side, it has no loss of accuracy since the coefficients are prescaled by the corresponding matrix E_i. The scaling with the matrices E_f and E_i are usually approximated by the quantizer and the dequantizer. Figure 4.3 shows the flow graph of the 4–point integer transform that is based on Equation 4.3. Here the multiplication with the factor 2 from matrix C_f is implemented by a left shift operation. In the corresponding inverse transform the multiplication by $\frac{1}{2}$ can be implemented by a right shift operation in a similar way [74].

4.2.4 8×8 Integer Transform

In 2004 the H.264 was expanded by an additional profile so called high profile for HD resolutions. The profiles "Baseline", "Main" and "Extended" use the 4×4 integer transform. The new profile uses an 8×8

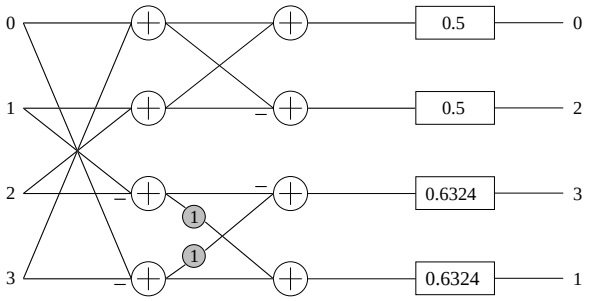


Figure 4.3: Flow graph of the 4–point integer transform in H.264

integer transform, which was incorporated by the standard [94]. The 8×8 integer transform can also be separated into one for horizontal transform and another one for vertical transform. Each transform can be explained as the sum of two 4–point integer transforms. One of the 4-point integer transforms is identical to the transform in Figure 4.3, while the other one is a modification of it. The first one is executed on all the lines and columns with even index, while the modified one is executed on all the lines and columns with an odd index. The following equation explains this interrelation:

$$T_8 = \left(Q_1 \begin{bmatrix} 0 & 1 & -1 & 1.5 \\ -1 & -1.5 & 0 & 1 \\ 1 & 0 & 1.5 & 1 \\ -1.5 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{1}{4} \\ 0 & 1 & \frac{1}{4} & 0 \\ 0 & \frac{1}{4} & -1 & 0 \\ \frac{1}{4} & 1 & 0 & 1 \end{bmatrix} Q_1^T \right. \\ \left. + W_1 \begin{bmatrix} 1 & 1 & 1 & 0.5 \\ 1 & 0.5 & -1 & -1 \\ 1 & -0.5 & -1 & 1 \\ 1 & -1 & 1 & -0.5 \end{bmatrix} W_1^T \right) M, \tag{4.5}$$

where the matrices Q_1, W_1 and M are defined as follows:

$$Q_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad W_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{and}$$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}. \tag{4.6}$$

Therefore, the 8×8 integer transform is defined as:

$$Y = T_8 X T_8^T. \tag{4.7}$$

The left summand from Equation 4.5 shows the modified part of the transform, which is executed on the lines and columns with an even index according to the matrix Q_1. The right summand shows the unmodified part of the transform, which is executed on all the lines and columns with an odd index. Figure 4.4 shows the flow graph of the 8–point integer transform based on Equation 4.5. The end of the flow graph shows the scaling quantization factors.

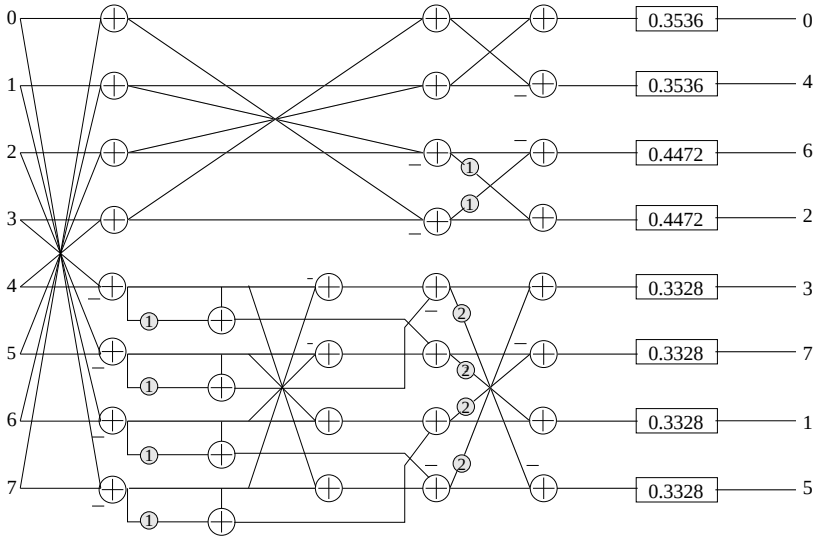


Figure 4.4: Flow graph of the 8–point integer transform in H.264

4.3 Discrete Cosine and Integer Transform

4.3.1 Forward DCIT

In order to combine the DCT transformation and the integer transforms, the CLDCT's flow graph has been modularized, where Figure 4.5 shows the RTL flow graph of the proposed 1–D Forward DCIT (FDCIT) transformation with all required resources. Five configurable modules, which can execute different operations, are used for performing both the multiplierless DCT and integer transforms.

First of all, the proposed DCIT architecture can execute an 8–point DCT with a dedicated sub flow graph as shown in Figure 4.6(a), where Module 1–4 are the original operations for the CLDCT and Module 5 is a bypassing circuit. Note that two data paths have to be switched between Module 2 and Module 3. Furthermore, after Module 2–3 the four adders inside the shaded box and the four data paths indicated by dashed lines have to be bypassed. The required scaling values are identical to Figure 4.2 with a specified order of the outputs.

Next, the five configurable modules have been modified in order to perform the 8–point integer transform on the resulting hardware as

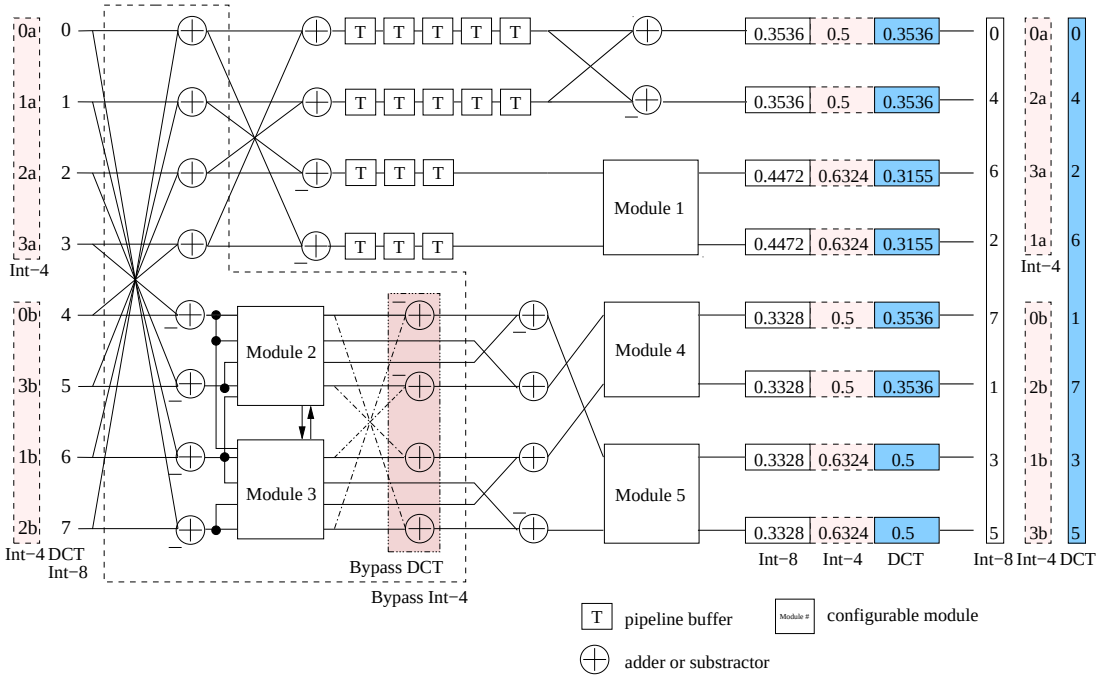
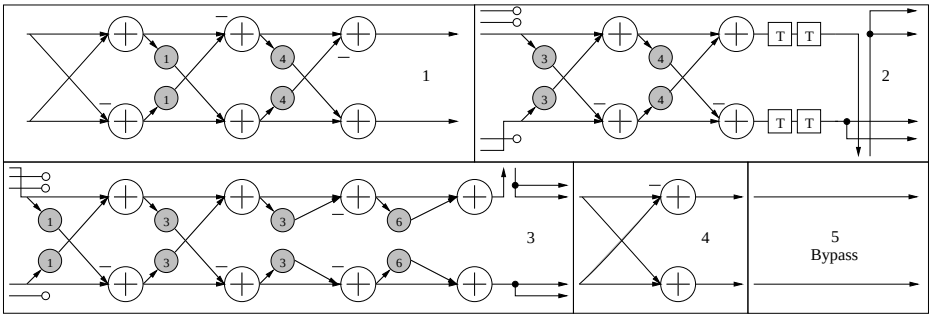


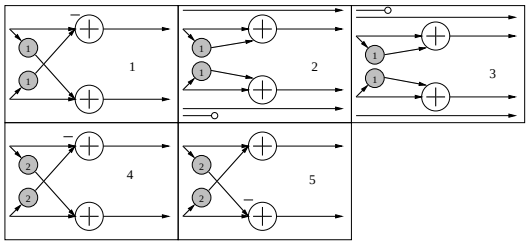
Figure 4.5: Flow graph of an 8–point FDCIT Transform with five configurable modules for multiplierless DCT and integer transforms [106]

shown in Figure 4.6(b). In detail, in the upper part of the flow graph, a pair of shifter and adder is required in Module 1. In the lower part of the flow graph, Module 2 and Module 3 have been simplified into one single CORDIC compensation step with two shifters and two adders for each. Two further shifters are inserted in Module 4 and one pair of shifter and adder is inserted in Module 5. The required scaling values and the output order for the 8–point integer transform are also given at the end of the flow graph.

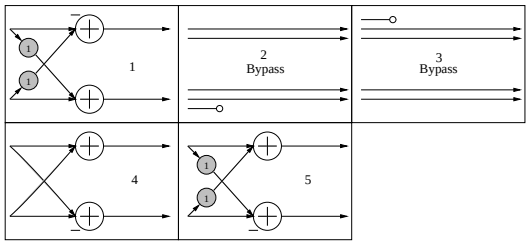
In the same way, two 4–point integer transforms are configured by reusing the flow graph. There are three configurable modules and two bypassing modules. Figure 4.6(c) shows the related sub flow graphs of the 4–point integer transform. First of all, the first stage consisting of butterfly operations, 8 adders, Modules 2-3 and the four adders in the shaded box after Modules 2–3 have to be bypassed in Figure 4.5. Module 4 is simplified into a butterfly operation. Module 1 and Module 5 have one pair of shifter and adder for each. Then, one 4–point integer transform is executed by the upper part of the flow graph, and another one is executed by the lower part. Additionally, the order of the input values has to be changed. The different order of the input/output and



(a) Function of the modules for an 8–point CLDCT



(b) Function of the modules for an 8–point integer transform



(c) Function of the modules for two 4–point integer transforms

Figure 4.6: Three sub flow graphs of the modules of Figure 4.5

scaling values are represented by the dashed boxes.

Overall, Figure 4.5 shows an integration of one 8–point DCT, two 4–point integer transforms and one 8–point integer transform with five configurable modules. The proposed architecture can perform these integer transforms and the DCT with a very small area overhead. Moreover, if we insert two pipeline buffers into Module 2, three buffers in front of Module 1 and five buffers at the upper part of the flow graph, then it will simply result in an 8–stage pipelined 1–D DCIT.

4.3.2 Inverse DCIT

Additionally, an inverse design of the DCIT (IDCIT) is implemented in a similar way as shown in Figure 4.7. This flow diagram can execute an 8-point IDCT, two 4-point inverse integer transforms and an 8-point inverse integer transform. The order of inputs and the corresponding coefficients for each input signal are marked at the beginning. There are seven configurable modules that perform various operations for different transform modes. For the IDCT mode, Modules 1–3, as shown

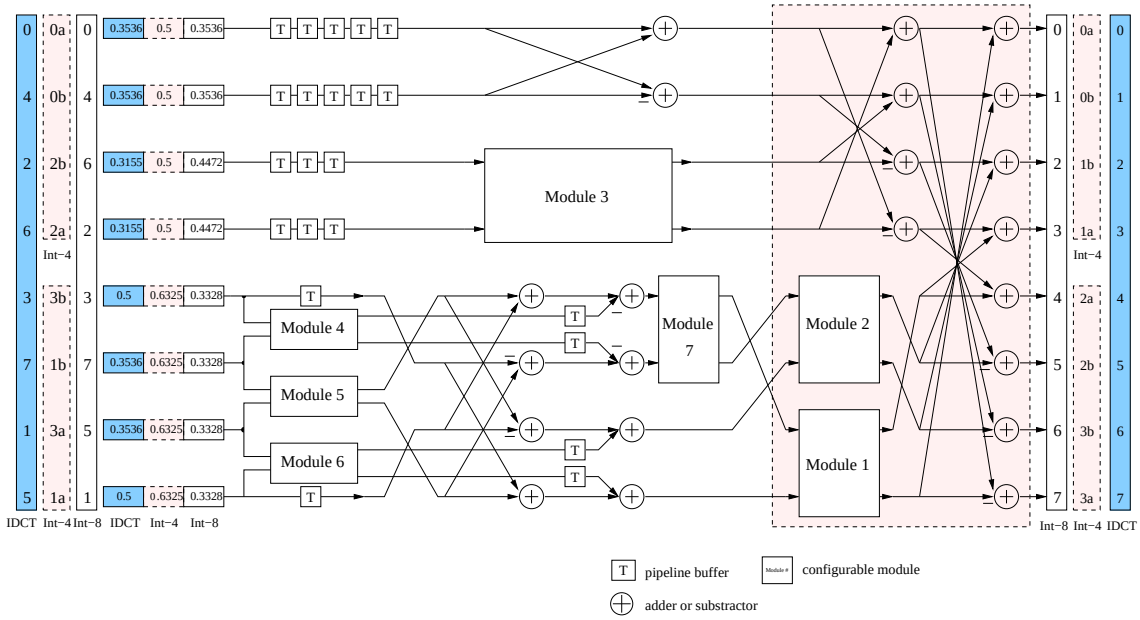
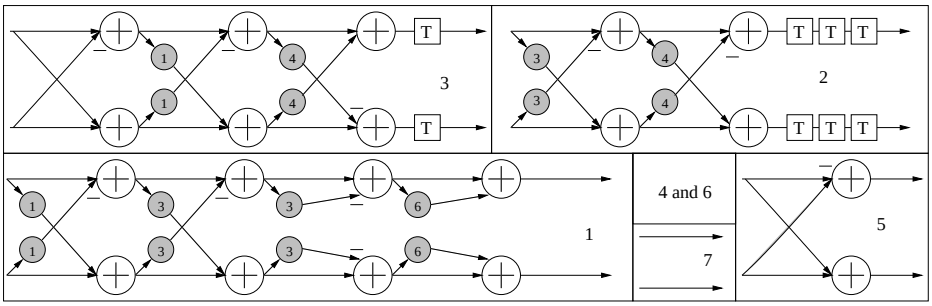


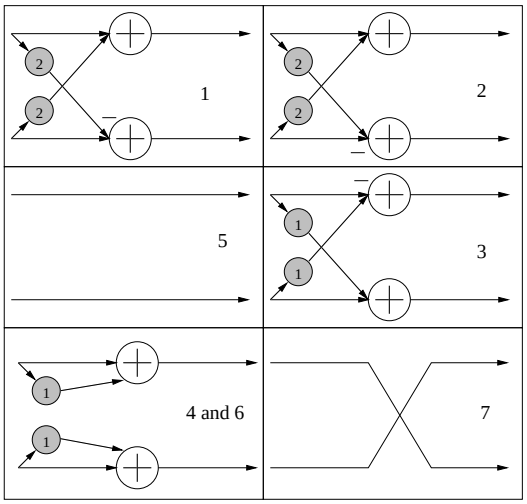
Figure 4.7: Flow graph of an 8–point IDCIT Transform with seven configurable modules for multiplierless IDCT and inverse integer transforms

in Figure 4.8(a), are represented based on the CORDIC algorithm. The length of signal flows in these three modules depends on the number of iterations and the amount of corresponding compensation steps. Module 4 and Module 6 are neglected. Module 5 is a butterfly operation for connecting with Module 1–2. Module 7 is a bypassing circuit.

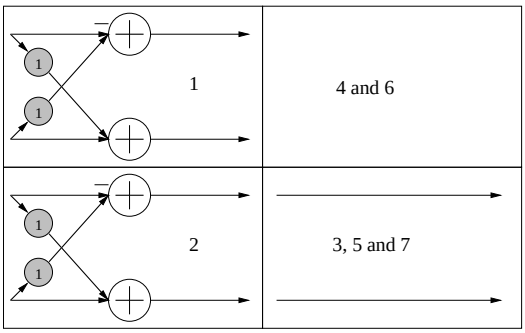
Figure 4.8(b) shows the mode of 8-point inverse integer transform. Module 4 and Module 6 will perform one step CORDIC compensation with two shifters and two adders for the following add and sub operations. Module 5 will bypass the input signals. Then the signals are switched in Module 7. At the same time, Module 1–3 has a pair of shifter and adder for each. Moreover, Figure 4.8(c) shows the configuration of two 4-point inverse integer transforms. Note that only the area bordered by dashed lines in Figure 4.7 will be activated, other parts are bypassed directly. Therefore, the input signals will be shifted to the highlighted area and further processed by Module 1–2. On the other hand, if we insert one pipeline buffer into Module 3 and three buffers into Module 2, three buffers in front of Module 1 and five buffers at the upper part of the flow graph, and six buffers at the lower part of the flow graph, then it will result in an 8–stage pipelined 1–D IDCIT.



(a) Function of the modules for an 8–point inverse CLDCT



(b) Function of the modules for an 8–point inverse integer transform



(c) Function of the modules for two 4–point inverse integer transforms

Figure 4.8: Three sub flow graphs of the modules of Figure 4.7

4.4 The proposed 2–D QDCIT framework

In this section we will describe how a CORDIC based QDICT is wired up in a block schematic view. Figure 4.9 shows the framework of the CORDIC based 2-D FQDCIT. This block diagram shows how the components of the 2–D DCIT transformation and the quantization modules are connected to each other. The inner part of Figure 4.9 which is indicated by dashed lines gives an overview of the 2–D DCIT core. The first 1–D DCIT, which can execute a 1–D DCT, two 1–D 4–point integer transforms or a 1–D 8–point integer transform over the rows by configuring the shared hardware resources, has already been described in Figure 4.5. The results of the 1–D DCIT over the rows of one block with a size of 8×8 pixels (or two 4×4 pixels in the case of the 4–point integer transform) are buffered in the transpose memory. Behind the memory buffer, there is a second DCIT transform core, which is extended from the first transformation core with longer bit lengths. After that, there are four CORDIC-Scalers with 17 dedicated LookUp Tables (LUT) executing the quantization. Note that not all the control signals are shown in this figure and the width of input/output wires are marked by numbers.

4.4.1 The 2–D QDCIT

The input signals of the quantizer are the output signals from the 2–D DCIT transformation. In each case, two of the output signals are connected to one CORDIC-Scaler, which can scale up or quantize the coefficients. The configuration for each CORDIC-Scaler is dependent on the type of transformation to be executed, on the current scaling factor, and on the quantization level. These configurations can be obtained from a LookUp Table read module. They are first analyzed and then stored in 17 dedicated lookup tables (8 are for DCT, and another 9 are for integer transforms). In order to perform the pipelined quantization, a CORDIC-Scaler Configurator is used to configure each pipeline stage of the four CORDIC-Scalers dynamically. At the end, the output signals from the CORDIC-Scalers will be sent to the final Post–Quantizer block.

Table 4.1 lists all the necessary control signals. The *mode* signal tells

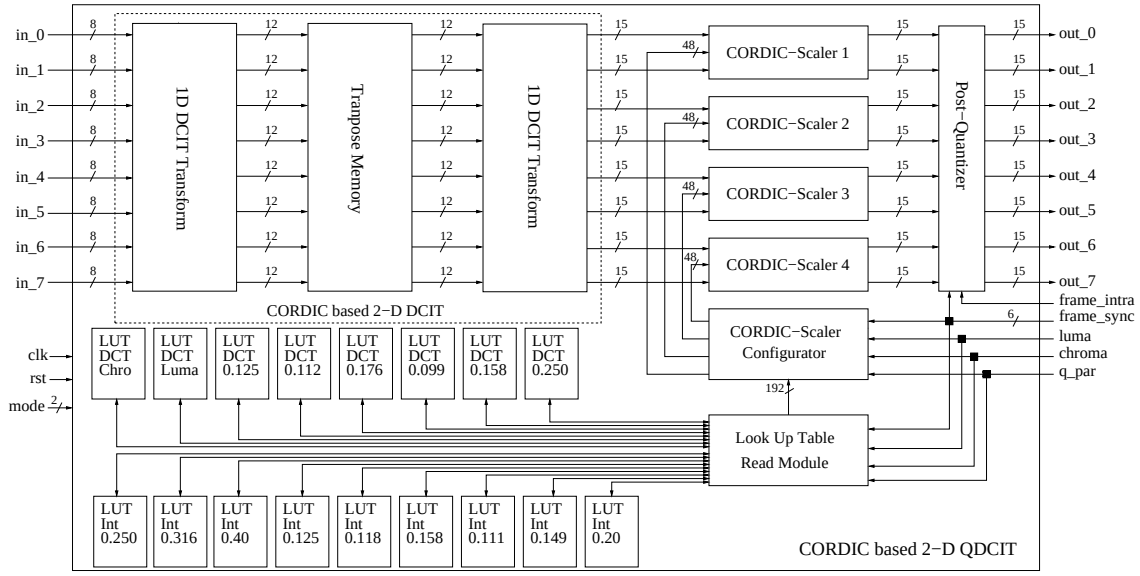


Figure 4.9: The framework of the proposed CORDIC based 2-D FQDCIT with four CORDIC-Scalers, a Post-Quantizer, a CORDIC-Scaler Configurator, a LookUp Table Read Module and 17 dedicated LUTs (8 are for DCT and the other 9 are for integer transforms)

the IP core which transformation should be configured. The *q_par* gives the value of the quantization scaling level. The *frame_intra* indicates that the incoming frame is an intra frame or an inter frame in MPEG-4. *Chroma* and *Luma* need one bit each to decide which kind of intra frame is coming in. If the incoming intra frame belongs to Chroma, the *Chroma* bit is set to "1". On the other hand, when a Luma intra frame is coming in, the *Luma* bit is set to "1". At the end, the *frame_sync* signal is very important to inform the IP core that a new frame is coming in (i.e. the first pixel is arrived).

In a similar way, we can reverse the 2–D FQDCIT as a configurable inverse flow graph by rearranging the components and rewiring them. Figure 4.10 shows this inverse framework. First of all, the inverse Post–Dequantizer is not required for MPEG-4 and H.264. The input pixels will first be dequantized by four CORDIC–Scalers. There is also a CORDIC–Scaler Configurator module, which configures the four parallel modules, to perform the scaling operations dynamically. The configuration of each CORDIC-Scaler is dependent on the type of transformation to be executed, on the current scaling factor, and on the dequantization level. After that, there is a sub–block which is indicated by dashed lines. There is an 1–D IDCIT followed by a row-column

Table 4.1: Control Signals for the proposed framework of 2–D QDCIT

Signal	Bits	Functionality
clock	1 Bit	Synchronous clock input
rst	1 Bit	Low reset input
mode	2 Bits	Transformation modes "00":8×8 DCT, "01":4×4 Int and "10":8×8 Int
frame_intra	1 Bit	Indication of the intra frame or the inter frame
frame_sync	1 Bit	Assert signal for the first input pixel
q_par	6 Bits	Index for the QP scaling level
chroma	1 Bit	Indication of the Chroma DC intra frame
luma	1 Bit	Indication of the Luma DC intra frame

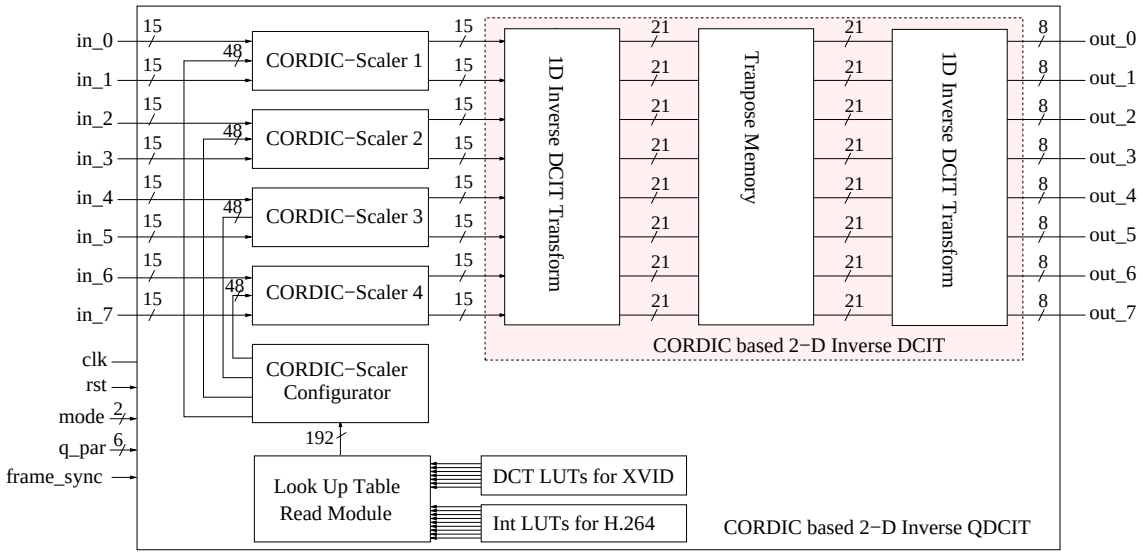


Figure 4.10: Framework of a CORDIC based 2-D IQDCIT

transpose memory, then followed by another 1–D IDCIT to perform the 2–D multiplierless IDCT and inverse integer transforms. The first and second 1–D IDCIT transform is the transform core that has already been described in Figure 4.7.

4.4.2 The CORDIC based Scaler

QDCT is one of the most efficient methods to reduce the transformation complexity compared to the conventional one (i.e. Quantization follows

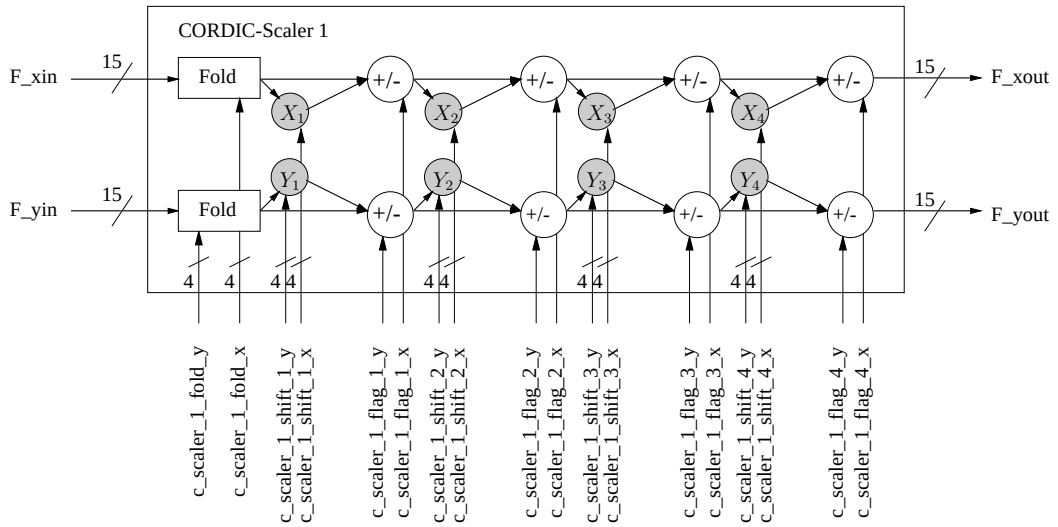


Figure 4.11: Schematic view of the first CORDIC-Scaler with one Fold and four CORDIC compensation steps

Table 4.2: The corresponding QPs of Luma DC and Chroma DC values in MPEG-4

Block type	**QP** ≤ 4	**5** $\leq$ **QP** ≤ 8	**9** $\leq$ **QP** ≤ 24	**25** $\leq$ **QP**
QPs for Luma	8	$2 \times QP$	$QP + 8$	$(2 \times QP) - 16$
QPs for Chroma	8	$(QP + 13)/2$	$(QP + 13)/2$	$QP - 6$

the 2-D DCT). However, direct implementation of the corresponding quantizer by multipliers with the CSD representation is not ideal for VLSI implementation. Besides CSD representation, there is another rather simple and elegant way to approximate multiplierless QDCT by utilizing the compensation phase of the CORDIC algorithm with sequential shifters and adders [83].

In order to integrate both DCT for MPEG-4 and integer transform for H.264 in a single configurable IP core, utilizing the CORDIC compensation iteration method is applied for integration of two different quantization methods. First of all, we look at the scaling factors at the end of the flow graph in Figure 4.2. These scaling factors are deduced from the CORDIC compensation steps and the integer transforms. On the other hand, since the quantization phase behind the DCT/Integer transforms for image/video compression are usually fixed steps, we could combine the transformation scaling factors and the quantization scaling levels together by utilizing the CORDIC compensation circuit, and then store the pre-computed compensation parameters in the lookup tables.

Table 4.3: Values of QStep dependent on QP in H.264

QP	0	1	2	3	4	5	6	7	8	9
QStep	0.625	0.6875	0.8125	0.875	1	1.125	1.25	1.375	1.625	1.75
QP	10	11	12	...	18	...	24	...	...	51
QStep	2	2.25	2.5	...	5	...	10	...	...	224

According to the dependence flow graph of the unfolded CORDIC architecture presented in [83], a CORDIC based Scaler (CORDIC-Scaler) with one fold operation and four compensation steps has been implemented as shown in Figure 4.11. This CORDIC-Scaler can approximate a predefined scaling value and one of the three different quantization values shown at the end of the flow graph in Figure 4.5. In MPEG-4 quantization "method 2" has a range of Quantization Parameter (QP) from 1 to 31. In H.264 the range of Quantization Step (QStep) is from 0 to 51 but results into a nonlinear QStep value as shown in Table 4.3. The combination of transformation scaling factors and quantization steps is described as:

$$\begin{aligned} Y_i^q &= \tfrac{Y_i \times S_i}{[QP \text{ or } QStep]}, \\ &\text{with } i \in \{0, 1, 2, \ldots, 7\}, \end{aligned} \tag{4.8}$$

where Y_i are the transformed results, Y_i^q are the quantized results, S_i are the scaling values, QP as given in Table 4.2, and $QStep$ as given in Table 4.3. Equation 4.8 shows the typical case of quantization phase when the inputs F_i are results from DC values for MPEG-4 (Luma or Chroma) in the DCT mode [93] or result from H.264 in the integer mode [94]. Note that the range of QPs for DC values is not linear as indicated in Table 4.2. On the other hand, when the inputs are AC values for MPEG-4 intra frame in the DCT mode, the quantization formula is described as:

$$\begin{aligned} Y_i^q &= \tfrac{Y_i \times S_i}{2QP} \\ &= (\tfrac{Y_i \times S_i}{QP}) >> 1, \\ &\quad \text{with } QP \in \{1, 2, 3, \ldots, 31\}, \end{aligned} \tag{4.9}$$

where QP is doubled compared to the Equation 4.8 and $>> k$ denotes a right shift by k bits. In order to perform the quantization by sharing the hardware resources, we move the division by 2 out as a right shift by 1 bit. Next, when the inputs are AC values for MPEG-4 inter frame

in the DCT mode, the quantization formula is described as:

$$\begin{aligned} Y_i^q &= \frac{Y_i \times S_i - \frac{QP}{2}}{2QP} \\ &= \frac{Y_i \times S_i}{2QP} - \frac{1}{4} \\ &= ((\frac{Y_i \times S_i}{QP} << 1) - 1) >> 2, \\ &\quad \text{with } QP \in \{1, 2, 3, \ldots, 31\}. \end{aligned} \tag{4.10}$$

Equation 4.10 can also be modified by first performing a left shift by 1 bit, then subtracting one, and right shift by 2 bits. All these three quantization equations have a common part $\frac{Y_i \times S_i}{QP}$, which can be further approximated by the proposed CORDIC–Scaler. At first, there is a stage called "Fold". This Fold is a barrel shift operation which is used to pre-scale down the input values. The implementation of these Folds before the CORDIC compensation steps is required, because the scaling factors are too small after the 2–D transformation and become extremely tiny after combination with the quantization scaler. By using this pre-scaling "Fold" technique the inner iteration number of CORDIC compensation steps can be reduced from 9 or 10 steps to about 4 or 5 steps in average. This Fold operation is followed by 4 CORDIC compensation steps. Note that the shadowed circles represent the barrel shifters, and the white circles represent an adder or a substractor. It has 4 adders and 5 shifters (1 fold and 4 compensation steps) for each input, so each CORDIC-Scaler component has a computational complexity of 8 adders and 10 shifters. Since the bit shifting operations are not fixed, we have to use the barrel shifter to perform variable shifting operation instead of wire shifting as in the transform part. Furthermore, this CORDIC–Scaler can also support the scaling up procedure for the inverse transformation by modifying the direction of barrel shifters and the fold, this could be easily achieved by modifying the configurator and regenerating the lookup tables.

4.4.3 The CORDIC-Scaler Configurator and the LUT Read Module

In our frameworks, four CORDIC-Scalers are simultaneously configured by the component "CORDIC-Scaler Configurator" as shown in both Figure 4.9 and Figure 4.10. For each bit shifting operation in a CORDIC-Scaler it is required to shift by maximal 16 bits. Hence 4 bits for the configuration of each bit shifting operation are enough. To configure the CORDIC compensation step concerning addition or subtraction, there is one bit necessary for each adder/substractor. This results in 48 bits for the configuration of each CORDIC-Scaler. After the 2–D DCIT, the 8×8 blocks of the transformed coefficients need to be further scaled and quantized. In the following equations, the scaling factors will be summarized into a vector, which is called e_{DCT}.

$$\begin{aligned} e_{DCT} &= \left[\tfrac{\sqrt{2}}{4} \quad \tfrac{\sqrt{2}}{4} \quad \tfrac{1}{3.1694} \quad \tfrac{1}{2} \quad \tfrac{\sqrt{2}}{4} \quad \tfrac{1}{2} \quad \tfrac{1}{3.1694} \quad \tfrac{\sqrt{2}}{4}\right] \\ &= \begin{bmatrix} 0.3536 & 0.3536 & 0.31551 & 0.5 & 0.3536 & 0.5 & 0.31551 & 0.3536 \end{bmatrix}. \end{aligned} \tag{4.11}$$

In order to scale the coefficients of a 2-D CLDCT, which has been separated into two 1–D CLDCTs, the scaling must be further processed. If we describe the 2-D CLDCT as a multiplication of matrices, the operation can be described as:

$$Y = (C_{DCT} X C_{DCT}^T) \otimes E_{DCT}, \tag{4.12}$$

where $C_{DCT} X C_{DCT}^T$ represents the 2–D 8×8 CLDCT transformation without the scaling. The matrix E_{DCT} is a matrix of scaling factors and is defined by the outer product of e_{DCT}:

Table 4.4: LUT organization of an entry for Quantization of DCT coefficients

Fold	**Flag 1**	**Flag 2**	**Flag 3**	**Flag 4**	$\mathbf{X_1/Y_1}$	$\mathbf{X_2/Y_2}$	$\mathbf{X_3/Y_3}$	$\mathbf{X_4/Y_4}$
4 Bits	1 Bit	1 Bit	1 Bit	1 Bit	4 Bits	4 Bits	4 Bits	4 Bits

$$E_{DCT} = e_{DCT}^{T} \times e_{DCT}$$
$$= \begin{bmatrix} 0.125 & 0.125 & 0.112 & 0.176 & 0.125 & 0.176 & 0.112 & 0.125 \\ 0.125 & 0.125 & 0.112 & 0.176 & 0.125 & 0.176 & 0.112 & 0.125 \\ 0.112 & 0.112 & 0.099 & 0.158 & 0.112 & 0.158 & 0.099 & 0.112 \\ 0.176 & 0.176 & 0.158 & 0.25 & 0.176 & 0.25 & 0.158 & 0.176 \\ 0.125 & 0.125 & 0.112 & 0.176 & 0.125 & 0.176 & 0.112 & 0.125 \\ 0.176 & 0.176 & 0.158 & 0.25 & 0.176 & 0.25 & 0.158 & 0.176 \\ 0.112 & 0.112 & 0.099 & 0.158 & 0.112 & 0.158 & 0.099 & 0.112 \\ 0.125 & 0.125 & 0.112 & 0.176 & 0.125 & 0.176 & 0.112 & 0.125 \end{bmatrix}. \quad (4.13)$$

If we take a closer look at the scaling matrix E_{DCT} we can notice that there are only 6 different scaling factors: 0.125, 0.112, 0.176, 0.099, 0.158 and 0.25, which means that only these 6 different scaling factors must be stored in lookup tables. In the current architecture, each scaling factor has one lookup table, which contains the configuration of CORDIC-Scalers to scale with the respective value dependent on the QP. The QP has a range from 1 to 31 for the DCT mode. Each entry of the table contains 24 bits configuration data. Table 4.4 shows the structure of one single entry in the lookup table. Therefore, 6 tables for the DCT transformation contain in total 31×24×6=4464 bits. On the other hand, the QPs of DC values in MPEG-4 are not linear as shown in Table 4.2. This results in only 28 possible combinations because the first four QPs are constant numbers. Hence we have created two more lookup tables for Luma DC values and Chroma DC values, respectively. The size of the Luma and Chroma tables are 28×24×2=1344 bits.

In the same way, there are only three scaling factors which are different after the 2–D 4×4 integer transform: 0.25, 0.316 and 0.4. So in the mode of the 4×4-integer transform, CORDIC-Scalers perform a multiplication with the current scaling factors divided by the QPs, which have a range from 0 to 51. Similar to the 4×4 integer transform, there are 6 different scaling factors for the 8×8 integer transform which must be stored in the lookup tables: 0.125, 0.118, 0.158, 0.111, 0.149, 0.2. If we take a closer look at Table 4.3 we can notice that the

Table 4.5: LUT organization of an entry for Quantization of 4 × 4/8 × 8-integer transform coefficients

Flags	$\mathbf{X_1/Y_1}$	$\mathbf{X_2/Y_2}$	$\mathbf{X_3/Y_3}$	$\mathbf{X_4/Y_4}$	**fold values 1...9**
4 Bits	4 Bits	4 Bits	4 Bits	4 Bits	36 Bits

value of QStep doubles for each increment of 6. This means that for each increment of 6 the configuration is the same except for the "Fold" value, which increases by 1. It is not necessary to store all combinations in the lookup tables for each value of QP. It is enough to store only 6 entries with the "Fold" values in order to reduce the size of the table. As there are 52 different values of QP there can be up to 9 increments of 6 for each value from $0 \cdots 5$. Hence for each entry there are 36 bits necessary for the fold. Table 4.5 shows how one entry in a table for the integer transforms is organized. The size of the 4×4 and 8×8 integer transform tables are 6×56×9=3024 bits.

Therefore, 6 tables for DCT scaling factors and two further tables for Luma DC values and Chroma DC values are needed in MPEG-4, and 9 tables are required for 4×4 and 8×8 integer transforms in H.264. In total 17 tables are required for the proposed architecture. In a similar way, for supporting the inverse transformation, the CORDIC–scaler can perform the scaling up and dequantization by regenerating the lookup tables.

The CORDIC-Scaler configurator will first get all the data from the component "LookUp Table Read Module" and forward the relevant configuration to each CORDIC-Scaler. Figure 4.12 shows the inputs and outputs of the CORDIC-Scaler configurator module and the LUT read module. First of all, there are two methods to configure the CORDIC-Scalers, dynamic or static. In the static case the configuration is varied when ever the value of QP changes. In the dynamic case the configuration of the CORDIC-Scalers changes with the clock cycle, because the matrix of scaling factors contains different scaling factors for each column.

Since the architecture is pipelined, the CORDIC-Scalers have to be configured dynamically during each clock cycle. Of course, the overhead of hardware resources is higher compared to the static method due to the extra registers for the buffering and the more complex scheduling

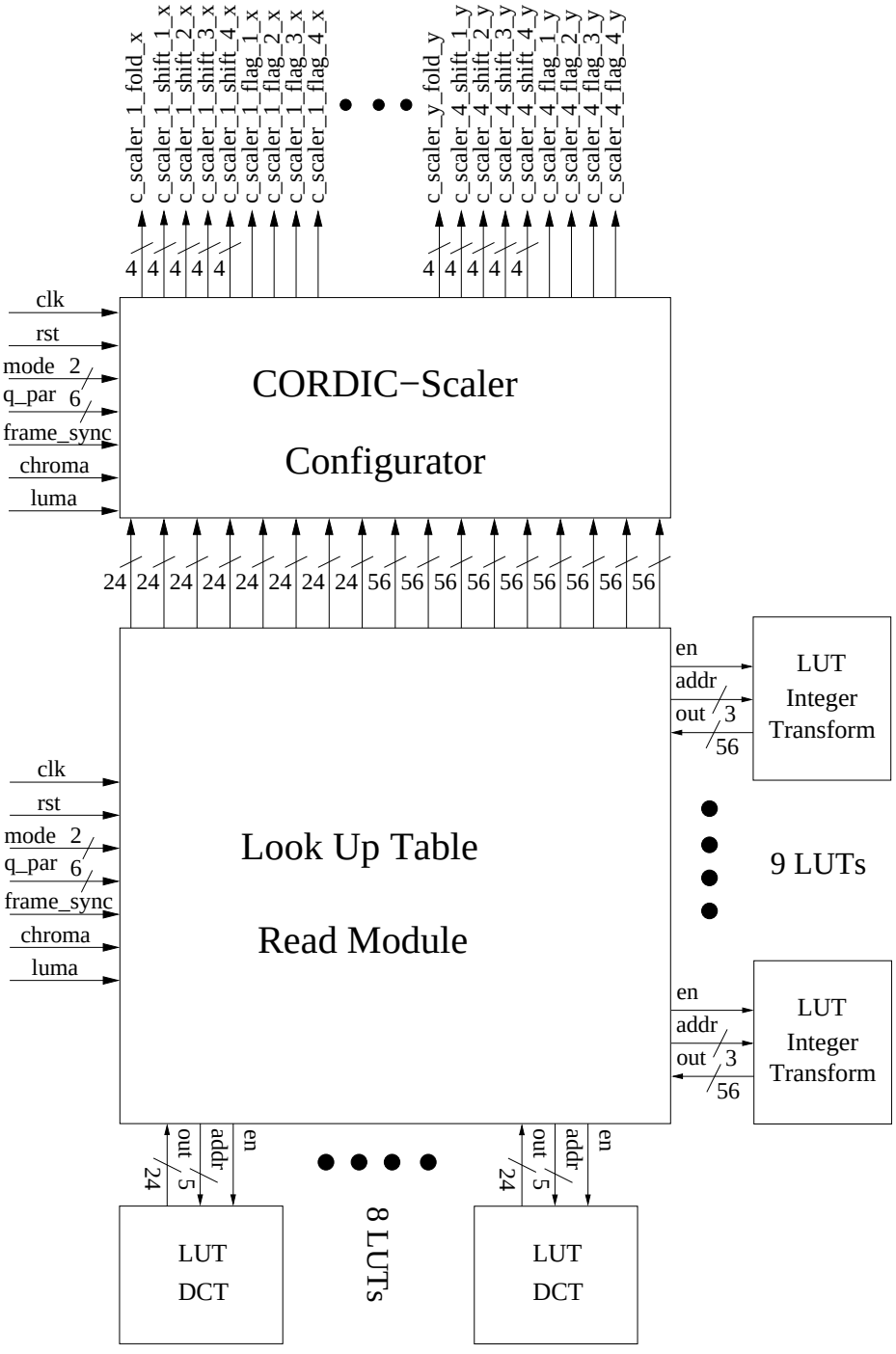


Figure 4.12: Schematic view and IOs of the LUT reader module and CORDIC-Scaler configurator module

state machine for the configurations. The configuration signals contain 48 bits per CORDIC-Scaler. Dependent on the mode and the chroma and the luma signals, the CORDIC-Scalers are configured.

These four CORDIC-Scalers are configured dynamically to process 8 input values per clock cycle through a controller. This state machine will first be triggered by the input signal "frame_sync". And then the state machine will know that there is a new frame coming in and prepare to configure the CORDIC-Scalers. Before it starts to configure the CORDIC-Scalers, it has to wait for 21 clock cycles until the first transformed input pixels arrive in the DCT mode. Similarly, for the 4×4 integer transform 8 clock cycles are required and for the 8×8 integer transform 16 clock cycles are required. After these delays, the state machine will start to configure the CORDIC-Scalers, whose configurations change during each clock cycle. Note that the scaling inside a CORDIC-Scaler is separated into 5 stages, the fold and the 4 CORDIC compensation steps. Each of these stages is configured independently from the others.

4.4.4 The Post-Quantizer

For the forward transformation, the Post-Quantizer is required to perform the remaining compensations after the CORDIC-Scalers. There are three cases for the Post-Quantizer stage. First, if the inputs are zeros, belong to DC values or in the integer transformation modes, it will bypass them as indicated in Equation 4.8. Second, if the inputs belong to the intra frame in the DCT mode, it will perform right shift by 1 bit. Third, if the inputs belong to the inter frame in the DCT mode, it will perform left shift by 1 bit, minus one then right shift by 2 bits for the reason explained in Equation 4.10. Figure 4.13 shows the schematic view of the Post-Quantizer. Note that the Post-Quantizer is not required for the inverse transformation.

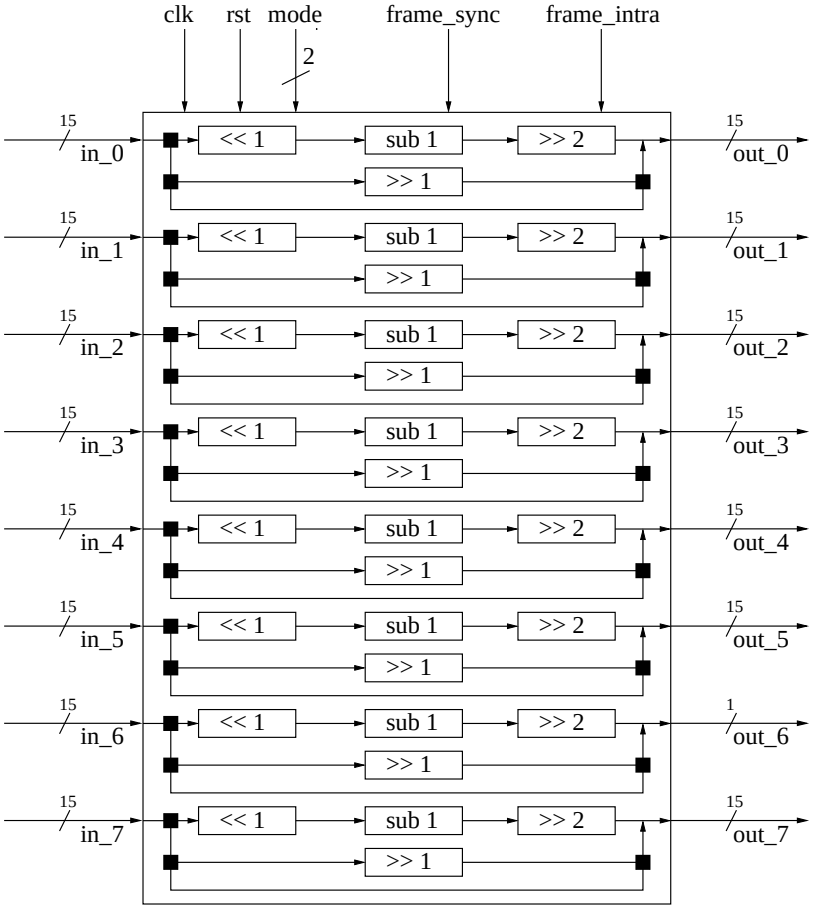


Figure 4.13: Schematic view of the Post-Quantizer

Table 4.6: Complexity for each 2-D quantized transformation architecture

Type \ Operation	Mul	Add	W–Shift	B–Shift	Mux
Conventional Q+DCT [34]	40	52	8	0	0
Novel QDCT [126]	32	52	8	0	0
Novel QDCT (CSD)	0	308	224	72	0
CORDIC based FQDCT	0	108	32	40	0
CORDIC based FQDCIT (with integer transforms)	0	120	40	40	56
CORDIC based FQDCIT (unshared)	0	268	60	120	8

4.5 Experimental Results

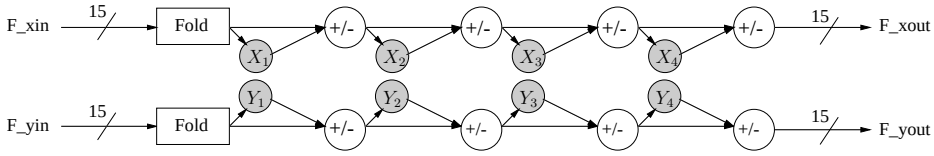
Table 4.6 summarizes the number of arithmetic units for each quantized architecture. At first, the conventional Q+DCT requires 40 mul, 52 add and 8 shift operations to perform 2-D QDCT. For the 2-D Novel QDCT 32 mul, 52 add and 8 shift operations are required [126]. The required multiplications for both methods are not ideal for VLSI design. Even if we use the CSD representation to carry out the multiplications, the Novel QDCT still needs 302 add and 72 barrel shift operations in the worst case. The presented 2-D Quantized CLDCT based on CORDIC architecture with four CORDIC-Scalers reduces the computational complexity dramatically to 108 add operations. The number of barrel shifters is reduced, too. Furthermore, for integration of the 8×8 and 4×4 integer transforms only 12 add, 8 barrel shift and 56 multiplex operations are required additionally. Of course, the FQDCIT requires extra multiplexers for performing the multi-functional transformations. However, the row "CORDIC based FQDCIT (unshared)" shows the complexity, if we implement the three quantized transformations without sharing hardware by using five configurable modules. It requires much more arithmetic units compared to the shared one.

Table 4.7: 2–D Transformation Complexity for arbitrary CORDIC iterations

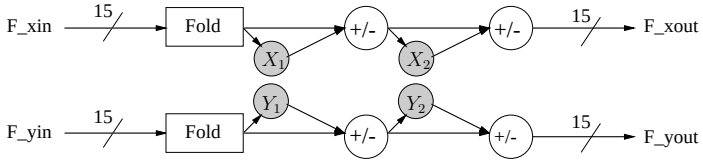
Type/Operation	Mul	Add	W–Shift	B–Shift	Mux
Novel IQDCT [126]	32	52	8	0	0
scaled CORDIC DCT [139]	4	64 + 80×6	80×6	0	0
CORDIC DCT/IDCT [116]	0	36 + 80×10	80×10	0	0
CORDIC based DCT [62]	0	208	176	0	0
CORDIC Loeffler DCT [112]	0	76	32	0	0
IQDCIT–S1	0	92	24	24	44
IQDCIT–S2	0	112	36	32	52
IQDCIT (default)	0	124	40	40	60
IQDCIT–C1	0	184	80	56	76

4.5.1 Variable Iteration Steps of CORDIC

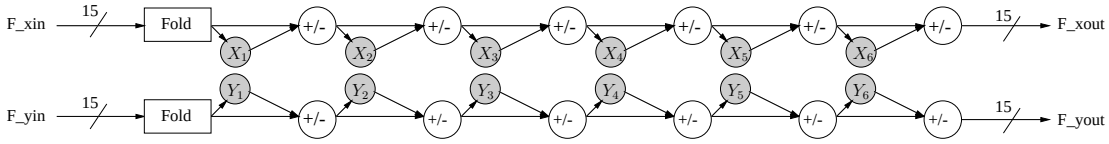
Since the proposed IDCIT is executed based on the CORDIC algorithm iteratively, we can adjust the iterations of trigonometric operations in the IDCIT (module 1–3) and the compensation steps of CORDIC-Scaler as different configurations. Table 4.7 lists the computational complexity if different number of CORDIC iterations are applied to the inverse transforms, where IQDCIT–S1 is a simplified version from the default IQDCIT with only 2 iterations/steps and IQDCIT–S2 has 2–3 iterations/steps. The number of iterations/steps is treated as a variable influencing on the precision. On the other hand, IQDCIT–C1 extends the number of iterations to 6–8 iterations/steps in order to obtain a better quality. For example, Figure 4.14 shows the configuration with different CORDIC compensation steps for the default IQDCIT with 4 CORDIC compensation steps, IQDCIT–S1 with 2 steps and IQDCIT–C1 with 6 steps. Note that the number of iteration is selected by the brute force search with a target resolution. Clearly, the proposed IQDCIT not only reduces more than half arithmetic units than other regular CORDIC DCTs but also executes the dequantization. In [116,139], besides adders for butterfly operations and multipliers, additional CORDIC processors are required. They assumed the DSP inside FPGA will satisfy the need for CORDIC rotations. Therefore, additional number of shift and add operations is required in here (80 shift–add operations for each pipelined CORDIC with compensation steps as mentioned in Section 3.3).



(a) Flow graph of the default CORDIC-Scaler with 4 iterations



(b) Flow graph of the CORDIC-Scaler S1 with 2 iterations



(c) Flow graph of the CORDIC-Scaler C1 with 6 iterations

Figure 4.14: Three flow graphs of CORDIC-Scaler with different number of CORDIC compensation steps

4.5.2 ASIC Implementation

A full-pipelined CORDIC based 2-D FQDCIT has been modeled in VHDL, including two 1-D DCITs, one transposition memory, four CORDIC-Scalers, one Post-Quantizer, one CORDIC-Scaler Configurator, one LUT reader and 17 dedicated lookup tables. Later, these RTL codes are synthesized by Synopsys Design Compiler 2009 with TSMC 0.18μm standard cell libraries. At the end, the final Place & Route stage is performed with the Cadence SoC Encounter 8.1 [19,117]. The final version of the chip ready for fabrication is shown in Figure 4.15. Note that the total number of the IOs required for this design is 198 IO pads, which would result in a low chip density as in [71]. Hence we have to remove the IO pads and treat it as a macro IP design. The input and output IOs are located in the left and lower side of the chip layout. Table 4.8 lists the comparison between our final layout results and other available cores in terms of several design criteria. Note that no further advanced methodologies are introduced to improve the performance. Nevertheless, a simple and straightforward pipelined design is presented (8–stage pipeline for the first 1–D DCIT, 8–stage pipeline for the transpose memory, 8–stage pipeline for the second 1–D DCIT and 5–stage pipeline for the CORDIC-Scaler). Figure 4.16 shows the timing waveform of the

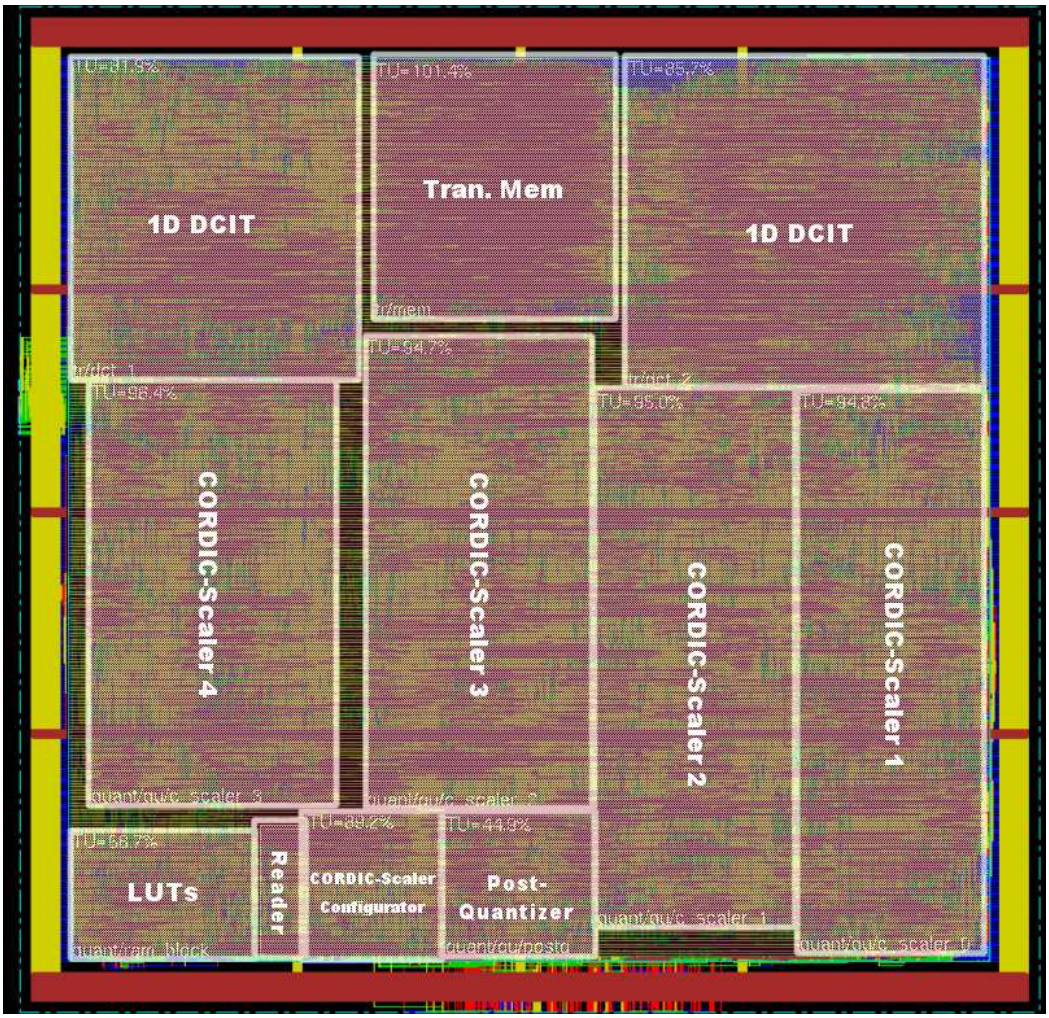


Figure 4.15: Final layout view of the 2–D CORDIC based FQDCIT implementation in TSMC 0.18μm technology library

post simulation in the mode of DCT transformation (QP=13) requiring 29 clock cycles until the first pipeline outputs are ready.

There are some important points can be observed. First, we can see that the proposed pipelined FQDCIT occupies small core size in chip area, because the CORDIC method enables a very simple architecture for implementation. For example, the core area is a little smaller than [97], but the proposed core can achieve much higher throughput and supporting multi-functions for different Codecs. Note that inverse version of the FQDCIT can be easily modified by reconfiguring the modules and flushing the lookup tables. Second, in [97] and [9], the cores achieve very high frequency by optimizing the pipeline stage (assuming better arithmetic units), but provide very low throughput due

to the serial output, i.e. one pixel per cycle. Contrary to the previous approaches, the proposed one processes 8 samples in parallel per clock cycle, leading to a higher throughput, which is very suitable for Full–HD or even future 4K/8K UHD resolution [25, 33]. This, however, does not lead to a very high increase in the chip area.

The latency is shorter than other designs due to the fine–gained pipeline architecture. The critical timing delay is 7.121ns. Of course, we can further reduce the critical timing by replacing the default "Ripple Carry Adder" in current implementation by "Carry Save Adder" or "Carry Look-ahead Adder" for each pipelined stage [49]. On the other hand, the size of the core can be further shrieked if we remove the pipeline and folding the row-column decomposition as one single DCIT and one CORDIC-Scaler with a buffer memory [9].

The presented implementation consumes more power than others due to the higher operational frequency and the parallel output characteristic. However, synthesizing with the multiple voltage threshold library and the clock/power-gating methodology can help reduce the power consumption dramatically [67]. It can be reduced by slowing down the operational frequency, too. Moreover, Dynamic Bit-width adaptation in DCT could also be applied into our design by decreasing the bit length of the high frequency coefficients [84]. The zero prediction algorithm can also reduce the power consumption significantly by omitting the unnecessary computational efforts [63, 127, 140]. These two methods can reduce the computational efforts and power consumption by trading off the transformation quality in PSNR. Finally, the VLSI implementation results show that the presented CORDIC based QDCIT architecture can provide a good solution for integration of different transformations and their quantization methods by utilizing the CORDIC algorithm to share the hardware resources.

4.5.3 Performance in MPEG–4 XVID and H.264

The proposed 2-D forward and inverse QDCIT transformations have been tested with the video coding standard MPEG–4 and H.264 by using publicly available software, XVID Codec 1.2.2 [137] and JM-16.1 Codec [105]. The default DCT algorithm in the Codec of the

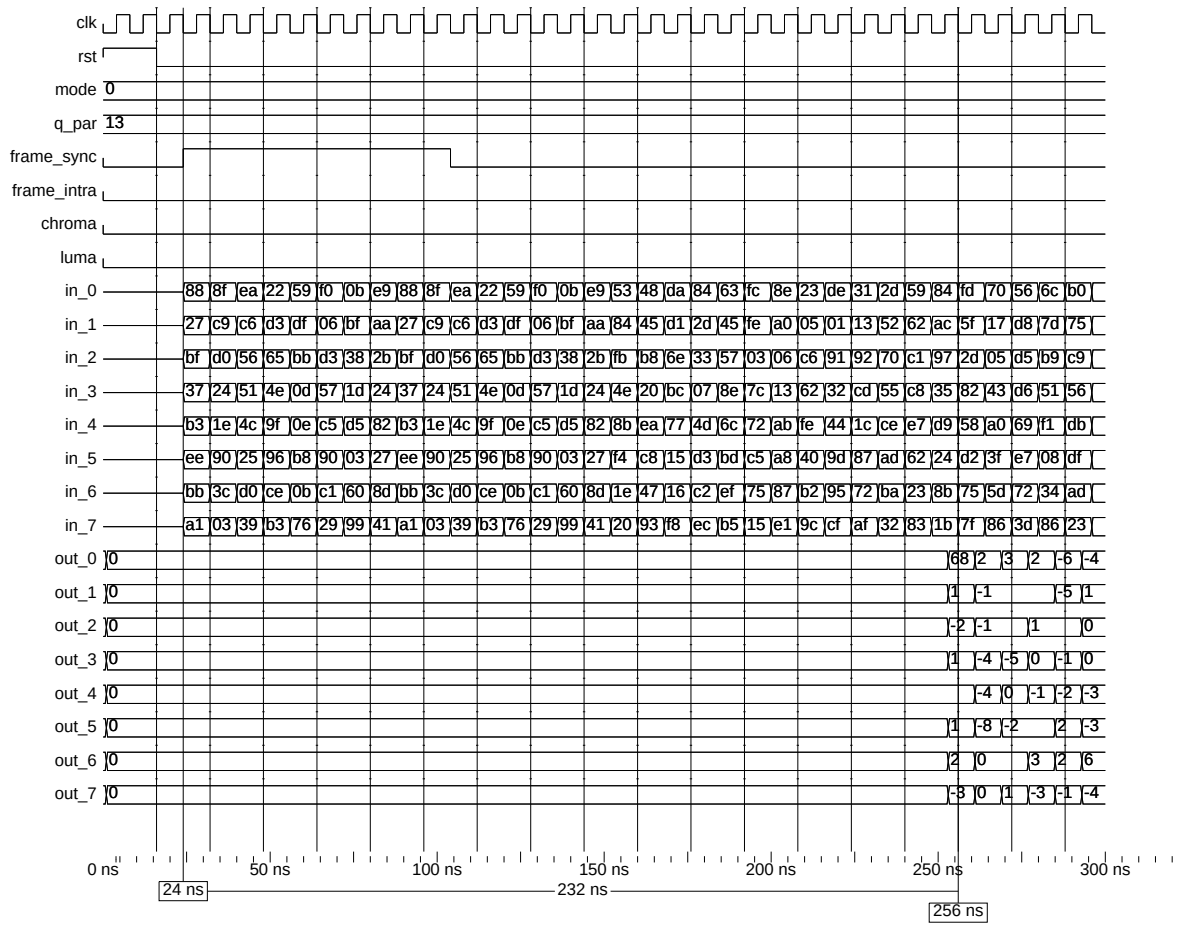


Figure 4.16: Timing waveform of the 2–D CORDIC based FQDCIT in the DCT mode (requiring 29 clock cycles for latency)

Table 4.8: Comparison of various DCT implementations and the proposed 2-D FQDCIT in different design criteria (area, timing, power, latency, throughput and architecture)

Year	Function	Tech	Area(mm)	Gates	Power	Frequency	Gbits/sec	Latency	Architecture
Ours	8×8/4×4 FQDCIT	0.18um 1.8V	2.05	47.1(K)	269(mW)	125 MHz	15.0	29 cycles	FGA TB Pipe PL
2007 [123]	8×8 DCT	0.18um 1.6V	1.32	N/A	4.08(mW)	80.6 MHz	0.967	N/A	AIR RC TB Pipe
2006 [97]	8×8 DCT/IDCT	0.35um 3.3V	3.00	11.7(K)	N/A	300 MHz	2.4	178 cycles	FGA RC TB Pipe
2005 [71]	4×4 QInt	0.18um 1.8V	N/A	51.6(K)	N/A	68 MHz	10.3	N/A	FGA Pipe PL
2005 [27]	8×8 DCT	0.18um 1.55V	0.12(Chip)	N/A	N/A	5 MHz	0.24	46 cycles	FGA RC Pipe TB
2004 [40]	8×8 DCT	0.18um	2.16	N/A	7.5(mW)	N/A	0.9	80 cycles	AIR RC TBRAM
2004 [9]	8×8 DCT	0.18um 1.8V	N/A	14.7(K)	N/A	180 MHz	0.352	392 cycles	MUXRC TB
2000 [22]	8×8 DCT	0.6um 2.0V	50.5	38(K)	138(mW)	100 MHz	N/A	198 cycles	DA PL
1998 [135]	8×8 IDCT	0.7um 1.3V	20.7(Chip)	40(K)	4.65(mW)	14 MHz	0.896	N/A	RC TB Pipe

DA (Distributed Arithmetic), FGA (Flow-Graph Arithmetic), AIR (Algebraic Integer Representations), RC (Row–Column Decomposition), MUXRC (Multiplexed Row–Column Decomposition), Pipe (Full Pipeline), PL (Parallel Architecture), TB (Transpose), TRAM (Transpose RAM).

Table 4.9: The list of test sequences

video sequence	frames	resolution
foreman	300	CIF (352×288)
paris	1065	CIF (352×288)
news	300	CIF (352×288)
crew	300	DVD (720×576)
ice	240	DVD (720×576)
rush hour	250	Full–HD (1920×1080)
blue sky	217	Full–HD (1920×1080)

selected XVID implementation is based on Loeffler's factorization using floating-point multiplications (see Figure 4.1) and MEPG–4 Quantization "method 2". Both the NQDCT and the QDCIT are implemented in a System–C like style with fixed point shift–add operations into the XVID software and the H.264 JM-16.1 software. To test the performance of our configurations, video test sequences with CIF, DVD and Full–HD formats are used, where Table 4.9 lists all the test sequences, where the framerate is 60 FPS (original figures are attached in Appendix B.5).

Figure 4.17 illustrates the average Peak Signal to Noise Ratio (PSNR) of the "foreman" and "paris" CIF video test sequences from low to high bitrates in XVID. The proposed architecture performs very close to the original Forward Q+DCT design and almost the same as the Forward NQDCT (FNQDCT). Figure 4.18 shows that the comparison results between the FQDCIT and the default method in H.264, where Int–4 mode and Int–8 mode are compared separately. Obviously, the FQDCIT architecture can obtain a good transformation quality. Note that the video sequences are first encoded with the FQDCIT architecture of these two referenced softwares, then decoded by using the default decoder.

On the other hand, the inverse version has also been implemented with different iteration numbers and compensation steps from Table 4.7, where Figure 4.19 illustrates the average PSNR of the "foreman", "paris" and "news" CIF video test sequences from low to high bitrates in XVID. Obviously, these results agree with our expectation that the configurations with more shift and add operations or CORDIC iteration steps can obtain better PSNR results. Moreover, the default "IQDCIT" is

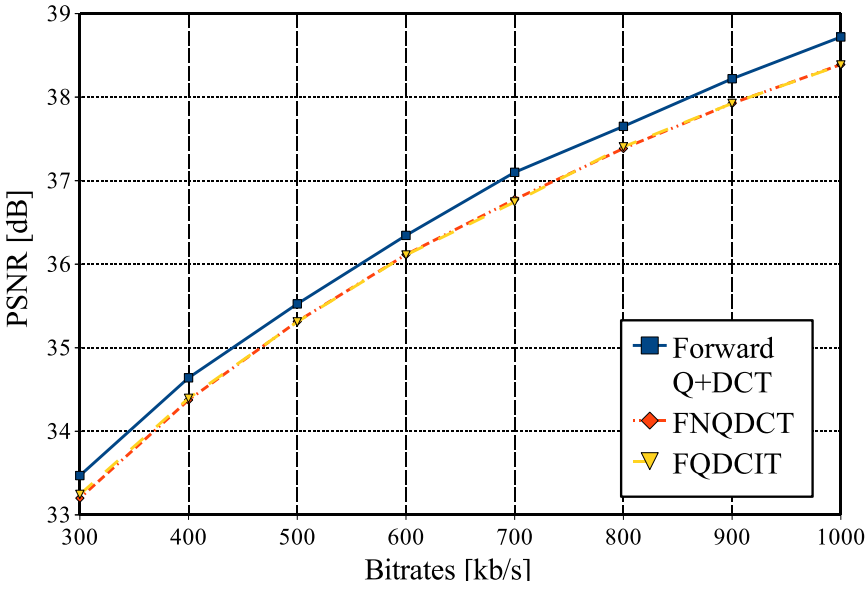


Figure 4.17: The average Forward Q+DCT, FNQDCT and FQDCIT PSNR of the "foreman" and "paris" cif video test from low to high bitrates in XVID

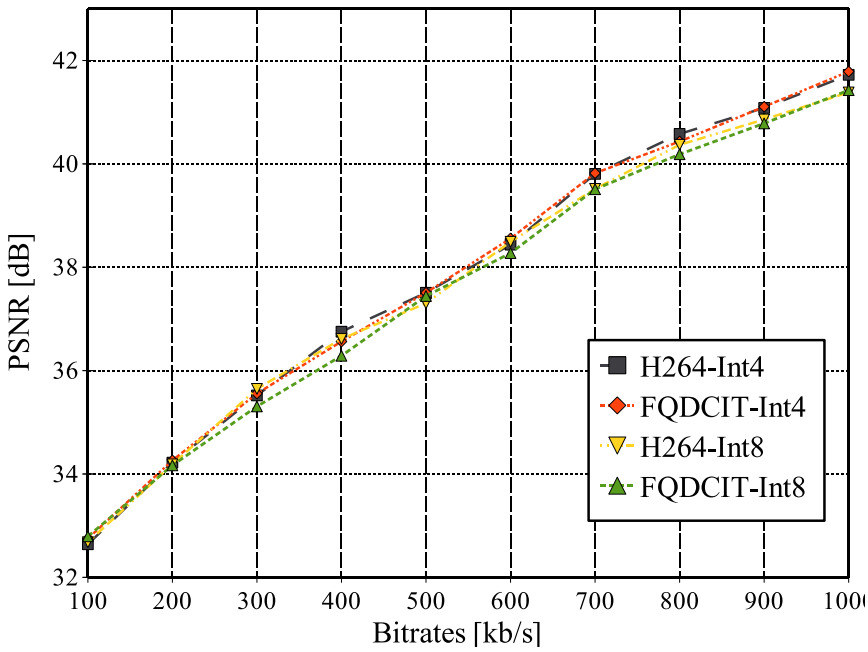


Figure 4.18: The average FQDCIT PSNR of the "foreman" and "paris" cif video test from low to high bitrates in H.264

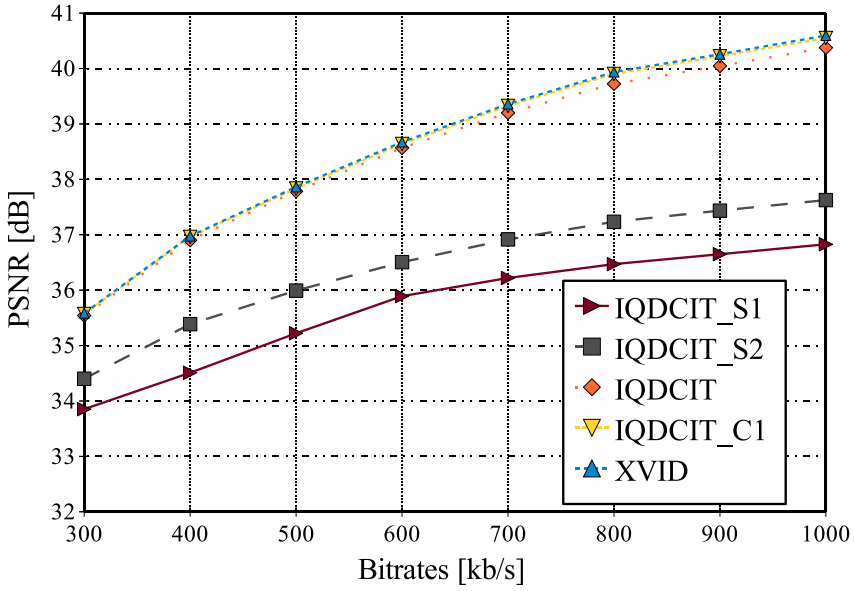


Figure 4.19: The average IQDCIT PSNR of the "foreman", "paris" and "news" cif video test from low to high bitrates in XVID

a good solution that can keep the balance between the hardware expense and the accuracy. It is very close to the original XVID decoder. However, since the resolution of the CIF is very small, the quality of "IQDCIT–S1" is enough for mobile devices.

Next, for testing the DVD format, the video sequences "crew" and "ice" are used to verify the four different configurations. Figure 4.20 shows the average PSNR. The results demonstrate that the four configurations can provide various precisions in terms of their hardware complexity. In this figure, a significant fall at the bitrate with 700kb/s is noticeable due to the insufficient shift and add operations stages of the "IQDCIT–S1". This means that the number of iteration is not enough. The "IQDCIT–S2" can achieve a reasonable tradeoff between the video quality and the computational complexity.

Finally, the arbitrary decoding test for Full–HD format is illustrated in Figure 4.21. The video sequences "rush hour" and "blue sky" are used. These results also prove that various configurations can cover the needs for video quality in consideration of their hardware expense. Clearly "IQDCIT–C1" should be selected for the Full–HD resolution.

In contrast to the arbitrary decoding tests on XVID Codec, the four

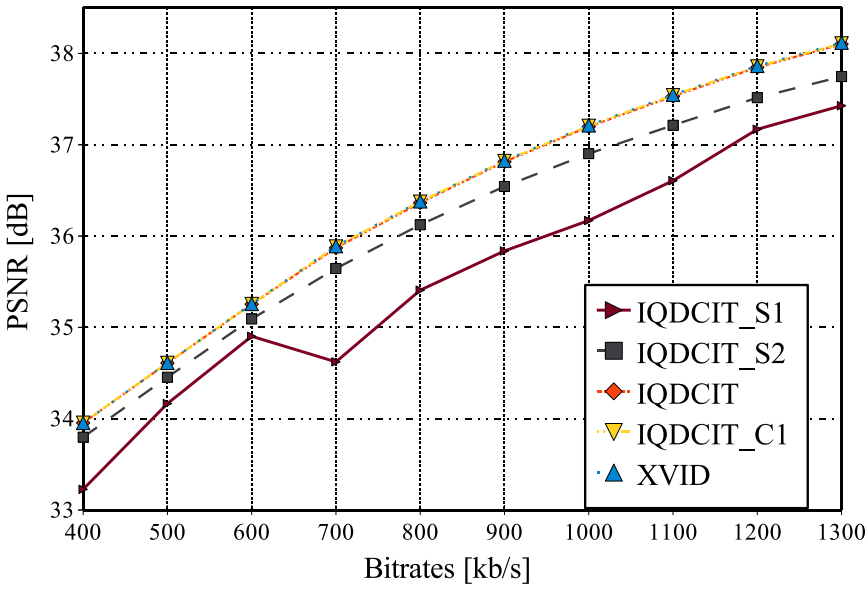


Figure 4.20: The average IQDCIT PSNR of the "crew" and "ice" DVD video test from low to high bitrates in XVID

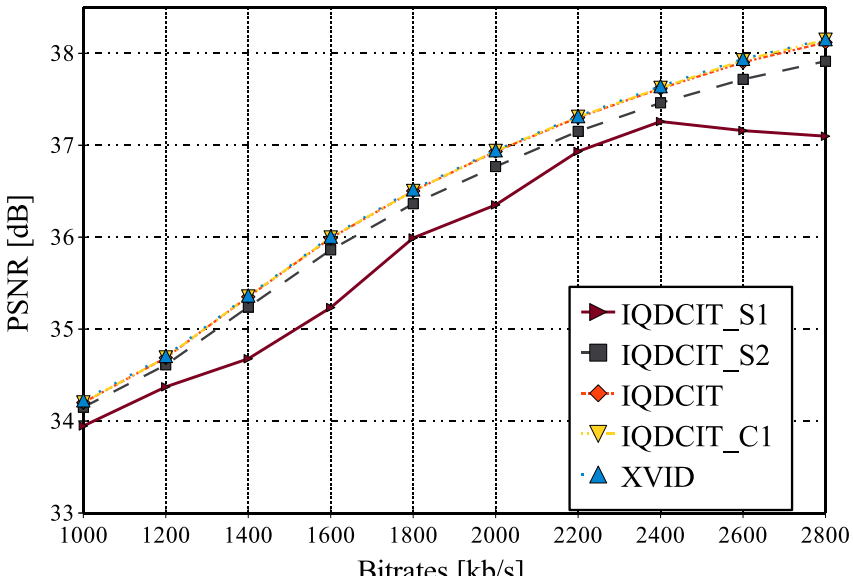


Figure 4.21: The average IQDCIT PSNR of the "rush hour" and "blue sky" Full–HD video test from low to high bitrates in XVID

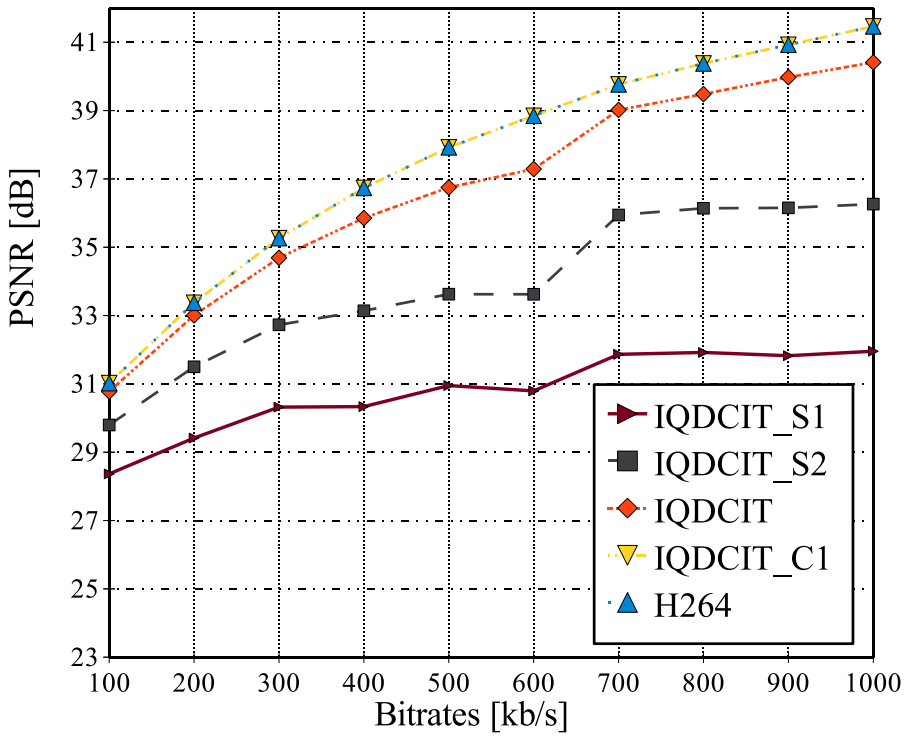


Figure 4.22: The average IQDCIT PSNR of the "foreman", "paris" and "news" cif video test from low to high bitrates in H.264

inverse configurations have also been evaluated for baseline profile and high profile in H.264. In consideration of that it is very practical to decode the high resolutions with the high profile. Hence, we have only tested the CIF format files with the baseline profile (4×4 integer), the DVD format files and the Full–HD format files with the high profile (8×8 integer). Figure 4.22 shows the average PNSR value of three test files for the baseline profile. The PSNR results increase monotonically when the CORDIC compensation steps increase. It can be noticed that the accuracy improves with higher complexity. Moreover, the configuration with "IQDCIT_S2" can provide a good compromise performance as expected, while it has relative low hardware expense.

For high resolution video tests, the video sequences "ice" and "crew" are selected to test the configurations for DVD sequences, and then used the video sequences "river bed" and " rush hour " to measure

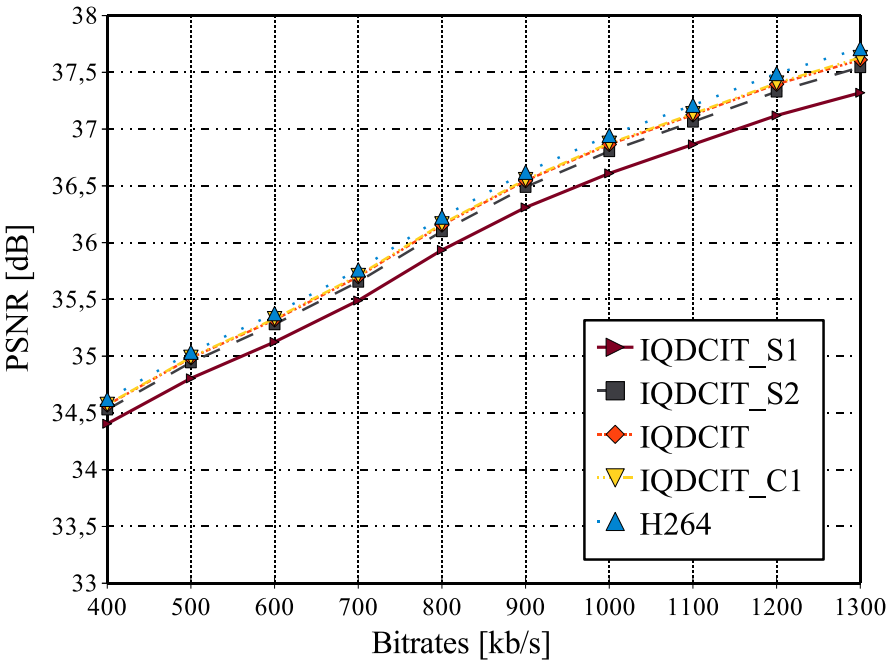


Figure 4.23: The average IQDCIT PSNR of the "crew" and "ice" DVD video test from low to high bitrates in H.264

the performance for Full–HD resolution. Figure 4.23 and Figure 4.24 illustrate the performance for each, respectively. Obviously, all four configurations show relative good performances. First, the results meet the conclusion for CIF sequences that the more expensive configuration is, the better it performs. In total, these four configurations can provide a balanced performance in terms of their hardware expense. The IQDCIT-S1 and the IQDCIT-S2 fit for small resolutions or low bitrates. However, when the bitrates are high, IQDCIT and IQDCIT-C1 are preferred. Therefore, these results agree with our expectation that the configurations with more shift–add operation stages/iteration steps can provide better accuracy. Moreover, the configuration "IQDCIT" is a good solution that can keep the balance between the hardware expense and the accuracy for both XVID and H.264 Codecs.

4.6 Summary

In this chapter, a low-complexity and highly-integrated QDCIT based on the CORDIC architecture was presented. The proposed 2-D FQD-

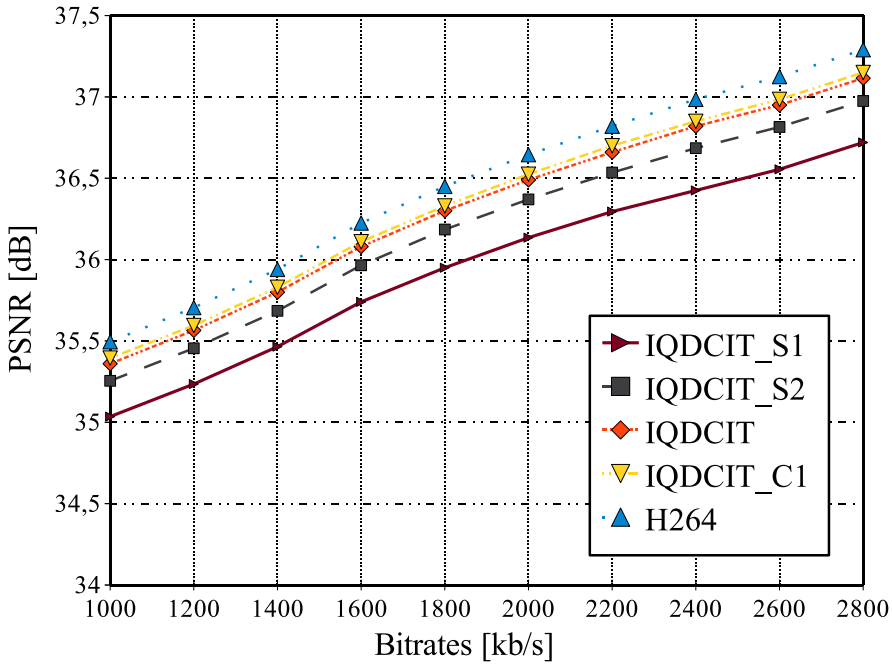


Figure 4.24: The average IQDCIT PSNR of the "rush hour" and "blue sky" Full–HD video test from low to high bitrates in H.264

CIT architecture requires only 120 add and 40 barrel shift operations to perform the multiplierless 8×8 FQDCT and 4×4/8×8 forward quantized integer transform by sharing the hardware resources as much as possible. The proposed design had been implemented with TSMC 0.18μm cell library. On the other hand, an inverse version based on the CORDIC algorithm with arbitrary iteration numbers had also been implemented. The proposed 2–D IQDCIT architecture requires only 124 add and 40 barrel shift operations to perform the multiplierless 8×8 IQDCT and quantized 4×4/8×8 inverse integer transform. The proposed architecture can reduce more than half of the add operations compared to the conventional Q+DCT and NQDCT architecture. Furthermore, it achieved a smaller chip area and higher throughput compared to the other architectures for future UHD resolution. Moreover, both XVID and H.264 simulations showed that the proposed QDCIT architecture can provide a good compromise between PSNR and computational effort, and is also highly satisfactory compared to the referenced implementations in terms of video quality. Also note that using arbitrary iteration steps can adjust the complexity of the architecture according to the target device's resolution.

5 Parallel Jacobi Algorithm

In this chapter, a configurable Jacobi Eigenvalue Decomposition (EVD) array using a scaling–free μ–rotation CORDIC (μ–CORDIC) processor is presented in order to further study the tradeoff between the performance/complexity of processors and the load/throughput of interconnects. Computing the EVD in parallel with Jacobi's iterative method is selected as an important example in this thesis, because the convergence of this method is very robust to modifications of the homogeneous processor elements [18,46,47,70]. It is simple, concise and inherent parallel for both implementation and computation. In [108,109], a Jacobi EVD array was realized by implementing the μ–CORDIC processor, which only performs a predefined number of CORDIC iterations (e.g. only one μ–rotation). In this way, the size of the processor array can be further reduced, such that a larger size of EVD array can be implemented. In order to further study the design impact in VDSM level, a 10×10 EVD array (i.e. computing the EVD of a 20×20 symmetric matrix) was implemented in TSMC 45nm technology. After that, several modifications of the algorithm/processor were studied and their impact on the design criteria were investigated for different sizes of EVD array through a configurable interface (10×10 to 80×80). At the end, a strategy to comply with the design criteria is presented, especially in terms of balancing the number of iterations and the computational complexity.

This chapter is organized as follows: serial and parallel Jacobi methods will be described in Section 5.1. In Section 5.2 the design issues of the parallel Jacobi EVD array are discussed, which lead to the simplification from a regular Full CORDIC to the μ–CORDIC processor with adaptive number of iterations. Section 5.3 shows the implementation results and Section 5.4 summarizes this chapter.

5.1 Parallel Eigenvalue Decomposition

5.1.1 Jacobi Method

An eigenvalue decomposition of a real symmetric n×n matrix A is obtained by factorizing A into three matrices $A = Q \wedge Q^T$, where Q is an orthogonal matrix ($QQ^T = I$) and $\wedge$ is a diagonal matrix which contains the eigenvalues of A. The Jacobi method approximates the eigenvalues iteratively as follows:

$$A_{k+1} = Q_k A_k Q_k^T, \quad \text{with} \quad k = 0, 1, 2, \ldots, \tag{5.1}$$

where Q_k is an orthonormal rotation by the angle θ in the (i, j) plane. It is an identity matrix containing four nonzero elements, $q_{ii} = q_{jj} = \cos\theta_k$, and $q_{ij} = -q_{ji} = \sin\theta_k$:

$$Q_k = \begin{array}{c} \begin{array}{ccccccc} & & \textit{col } i & & \textit{col } j & & \\ & & \downarrow & & \downarrow & & \end{array} \\ \begin{bmatrix} 1 & 0 & & \cdots & & 0 & \\ 0 & \ddots & & & & & \\ & & \cos\theta_k & & \sin\theta_k & & \leftarrow \textit{row } i \\ \vdots & & & \ddots & & \vdots & \\ & & -\sin\theta_k & & \cos\theta_k & & \leftarrow \textit{row } j \\ & & & & & \ddots\ 0 & \\ 0 & & & \cdots & & 1 & \end{bmatrix} \end{array} \tag{5.2}$$

The plane rotations Q_k, where $k = 0, 1, 2, \cdots$, can be executed in various orders to obtain the $\wedge$. The most common order of sequential plane rotations $\{Q_k\}$ is called cyclic–by–row, i.e. (i, j) is chosen as follows:

$$(i, j) = (1, 2)(1, 3) \ldots (1, n)(2, 3) \ldots (2, n) \ldots (n-1, n)\,. \tag{5.3}$$

The execution of all $N = n(n-1)/2$ index pairs (i, j) is called a sweep. If several sweeps are applied, the matrix A will converge into a

diagonal matrix $\wedge$, which contains the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$:

$$\lim_{k\to\infty} A_k \;=\; diag[\lambda_1, \lambda_2, \ldots, \lambda_n] \;=\; \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \lambda_n \end{bmatrix}. \qquad (5.4)$$

Besides the cyclic–by–row order, many different sequential orders have already been presented, such as butterfly–like permutation ordering, odd-even ordering or dynamic ordering by searching the best rotation set [10–12]. These specialized orderings are chosen according to the topology of network in order to guarantee good load balancing during the whole computational process. For example, for a parallel computer with ring style network topology, the odd-even ordering should be selected for the goal of shortest path data transmission. However, the influence from the network topology on different orders becomes less important due to the shorter timing delay between SoC processors, especially when the packet–switched network is gradually replacing the bus transmission in large SoC systems. Therefore, the cyclic–by–row order has been selected for the parallel EVD design.

5.1.2 Jacobi EVD Array

Instead of performing the plane rotations Q_k one by one in a cyclic–by–row order, we can separate them into multi–subproblems and execute them in parallel on a $(\log n)$–dimensional multi-core platform. A parallel array for Jacobi's method was first presented by Brent and Luk [18]. It consists of $\frac{n}{2} \times \frac{n}{2}$ homogeneous PEs and each PE contains a 2×2 sub-block of the matrix A. Figure 5.1 shows a typical 4×4 Jacobi EVD array with 16 PEs. This Jacobi array can perform $\frac{n}{2}$ subproblems in parallel. Initially, each PE holds a 2×2 sub-matrix of A:

$$PE_{pq} = \begin{pmatrix} a_{2p-1,2q-1} & a_{2p-1,2q} \\ a_{2p,2q-1} & a_{2p,2q} \end{pmatrix}, \text{ where } p \text{ and } q \;= 1, 2, \cdots, \tfrac{n}{2}. \quad (5.5)$$

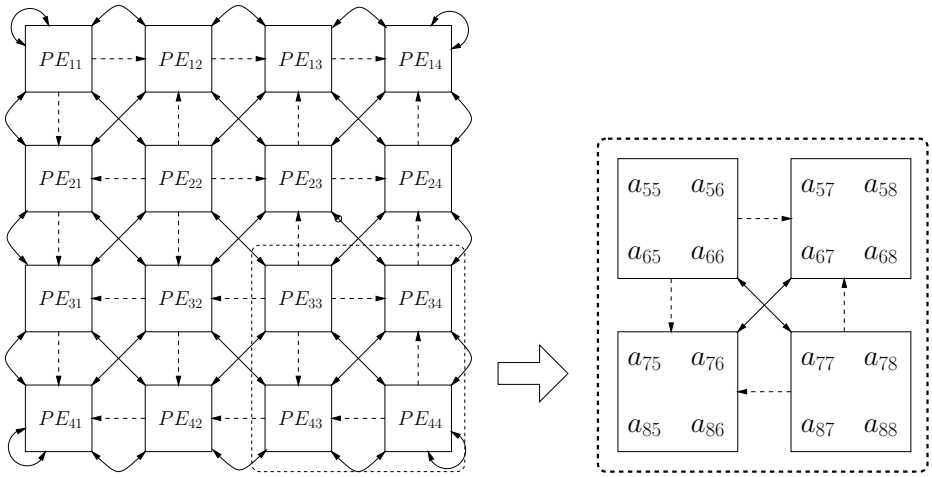


Figure 5.1: A 4×4 EVD array, where n=8 for 8×8 symmetric matrix

A rotation angle has to be choosen in order to zero the off–diagonal elements of the submatrix by solving a 2×2 symmetric EVD subproblem as shown in the following:

$$\begin{aligned} \begin{bmatrix} a'_{ii} & a'_{ij} \\ a'_{ji} & a'_{jj} \end{bmatrix} &= \mathrm{R} \cdot \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix} \cdot \mathrm{R}^T, \\ \text{where R} \;&=\; \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \end{aligned} \tag{5.6}$$

We obtain the maximal reduction $\{a'_{ij}, a'_{ji}\} = 0$ by applying the optimal angle of rotation θ_{opt}:

$$\theta_{\text{opt}} = \tfrac{1}{2}\arctan(\tfrac{2a_{ij}}{a_{jj}-a_{ii}}), \tag{5.7}$$

where the range of θ_{opt} is limited to $|\theta_{\text{opt}}| \leq \frac{\pi}{4}$.

This optimal angle θ_{opt}, which can annihilate the off–diagonal elements ($a_{2p-1,2q}$ and $a_{2p,2q-1}$), is computed by diagonal PEs using Equation 5.7. After these rotation angles are computed, they will be sent to the off–diagonal PEs. This transmission is indicated by the dashed lines in the vertical and horizontal direction in Figure 5.1. All off–diagonal PEs will perform a two-sided rotation with the corresponding rotation angles who are obtained from the row (θ_r) and column (θ_c) respectively.

After these rotations are applied, the matrix elements are interchanged between processors as indicated by the solid lines in diagonal direction in Figure 5.1 for execution of the next $\frac{n}{2}$ rotations. One sweep needs to perform n - 1 of these parallel rotation steps. After several sweeps (iterations) are executed, the eigenvalues will concentrate in the diagonal PEs. In practice, we can observe the Frobenius norm of the off–diagonal elements until it is close to zero or perform a predefined number of sweeps, which depends on the size/structure of matrix A.

5.2 Architecture Consideration

5.2.1 Conventional CORDIC Solution

Inside each PE, the most hardware efficient solution to solve the subproblem of Equation 5.6 for zeroing the off–diagonal elements is using the CORDIC algorithm (see Chapter 3). The CORDIC orthogonal rotation mode can be used to compute Equation 5.6 by separating the two side rotation into two parts, $G = \left[G_1^T; G_2^T\right] = A \cdot R^T$ and $R \cdot G$, and then using two CORDIC rotators to perform $A \cdot R^T$:

$$\begin{aligned} G_1 &= \left[a_{ii}^r, a_{ij}^r\right]^T = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \left[a_{ii}, a_{ij}\right]^T \\ G_2 &= \left[a_{ji}^r, a_{jj}^r\right]^T = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \left[a_{ji}, a_{jj}\right]^T, \end{aligned} \tag{5.8}$$

where the desired rotation angle θ is approximated by two CORDIC rotators iteratively. The CORDIC processors will usually apply $n = 32$ μ–rotations for single floating precision. After that, a constant scaling value $K = \frac{1}{A_n} = 0.6073$ is required to fix the rotated vectors $G_1 = \left[a_{ii}^r, a_{ji}^r\right]^T$ and $G_2 = \left[a_{ij}^r, a_{jj}^r\right]^T$ in order to retain the orthogonal property. In a similar way, we can use these two CORDIC rotators

to compute $R \cdot G$:

$$\begin{aligned} \left[a_{ii}^{'}, a_{ji}^{'}\right]^T &= \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \left[a_{ii}^{r}, a_{ji}^{r}\right]^T \\ \left[a_{ij}^{'}, a_{jj}^{'}\right]^T &= \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \left[a_{ij}^{r}, a_{jj}^{r}\right]^T . \end{aligned} \tag{5.9}$$

Meanwhile, the optimal rotation angle can be also approached by using the CORDIC orthogonal vector mode (see Table 3.1). The θ_{opt} angle can be approximated iteratively as: $2 \times \theta_{\text{opt}} = \arctan(\frac{2a_{ij}}{a_{jj}-a_{ii}})$.

In VLSI design, there are two common ways to realize the CORDIC dependence flow graph in hardware, the folded (serial) or the parallel (pipelining) as earlier described in Section 3.3. Note that we restrict only to the conventional CORDIC iteration scheme as mentioned in Equation 3.15. Although the folded CORDIC flow graph consumes more energy than the pipelined CORDIC, it is still selected here in order to satisfy the need for large EVD array computation. Figure 5.2(a) shows a folded CORDIC PE. It requires a pair of adders for the plane rotation and another adder for steering the next angle direction, which is identical to Figure 3.3. However, after the plane rotation, the remaining scaling procedure has to be further proceeded by the Figure 5.2(b) for fixing the A_n, where two multiplexers are required for selecting the inputs to the barrel shifters for CORDIC compensation steps. This folded dependence graph is typical for the orthogonal rotation mode and benefits in a small area in VLSI design. Moreover, it requires a number of different shifts according to the chosen shift sequence, additional barrel shifters are required in folded CORDIC architecture for successive recursive processing.

In practice, the angle accumulator is not required for the off–diagonal PEs. We can directly use the d_i from Equation 3.15 to steer the rotators. This means that the transmission on the dashed lines in the vertical and horizontal direction in Figure 5.1 will be replaced by a sequence of d_i flags. In this way, the computation effort for the θ_{opt} can be omitted.

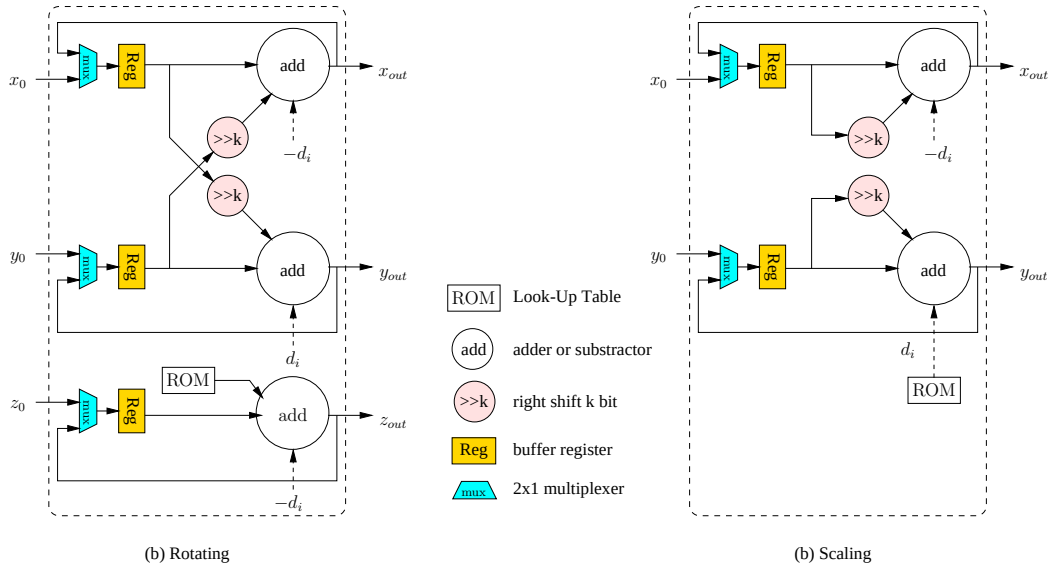


Figure 5.2: Flow graph of a folded CORDIC (recursive) processor with the scaling

5.2.2 Simplified μ–rotation CORDIC

In this subsection, the methodology how to simplify the regular CORDIC architecture will be clarified. As the process technologies continue to shrink to the VDSM level, it becomes possible to directly implement a full parallel Jacobi EVD array [3, 72]. However, the size of EVD array with the regular CORDIC that could be implemented on current device is still small. Therefore, it is necessary to simplify the architecture in order to integrate more processors for solving larger matrices. A scaling–free μ–CORDIC for performing the plane rotation in Equation 5.6 is used, where the number of inner iterations is reduced (from 32 iterations to only one iteration [46, 70]). Here, the μ–CORDIC will be implemented as a circuit and verified under different design criteria (area, timing, power/energy).

The definition of μ–CORDIC can be developed from Equation 3.15 as:

$$\begin{aligned} x_{i+1} &= \hat{m}\left[x_i - y_i \cdot d_i \cdot 2^{-i}\right] \\ y_{i+1} &= \hat{m}\left[y_i + x_i \cdot d_i \cdot 2^{-i}\right] \\ \hat{m} &= \sqrt{\cos^2\theta + \sin^2\theta} = 1 + \epsilon, \end{aligned} \tag{5.10}$$

where $\hat{m}$ is the required scaling factor per iteration and ϵ is the offset error. The idea of μ–CORDIC rotation is to reduce the number of

iterations of the full CORDIC to only few iterations. Meanwhile, the offset error ϵ will be small enough to be neglected as long as the orthonormal property is retained. Figure 5.3 shows four different subtype methods for different size of μ–rotation angles and Table 5.1 shows a lookup table for the μ–CORDIC. This lookup table lists 32 approximated rotation angles for each μ–rotation type, the required number of shift–add operations and its computation cycles. Note that the approximated angles are stored as two times of $\tan\theta$. When the rotation angle is very tiny (i.e. ϵ is tiny too), the Type I with only one iteration will satisfy the limited working range $1 - 2^{-(n_m+1)} < \hat{m} < 1 + 2^{-(n_m+1)}$, if the n_m ($n_m \in 1 \cdots 32$) is selected lager than 16. In Figure 5.3(a), a pair of shift–add operation realizing one iteration is enough. Furthermore, it is scaling free when the angle $2\tan\theta <= 3.05176 \times 10^{-5}$. These orthonormal μ–rotations are chosen such that they satisfy a predefined accuracy condition in order to approximate the original rotation angles and constructed by the cheapest possible method.

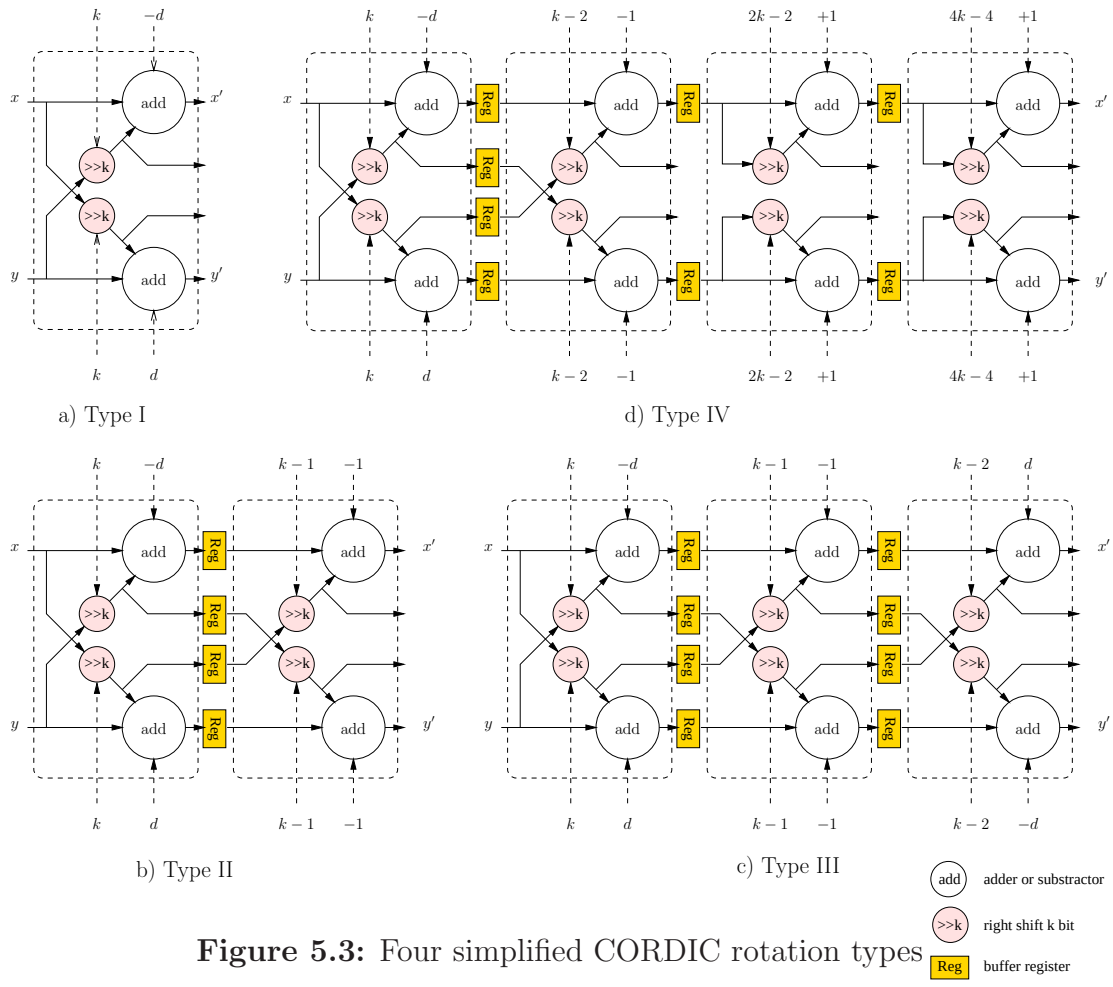


Figure 5.3: Four simplified CORDIC rotation types

Next, for the μ–rotation Type II as shown in Figure 5.3(b), when the n_m is selected from 8 to 15, two pairs of shift–add operations are enough to retain the orthonormal property. Moreover, when the n_m is selected from 5 to 7, Type III repeats three times μ–rotation without any scaling. Finally, for large rotation angles, the offset errors cannot be omitted without scaling. Figure 5.3(d) shows the corresponding dependence flow graph for Type IV. Besides two pairs of shift–add operations for rotation itself at the beginning of the sub flow graph, 2∼4 pairs of shift–add operations are required to fix the scaling factor $\hat{m}$:

$$\hat{m} = (1 + 2^{-2(k+1)})(1 + 2^{-4(k+1)}) \cdots (1 + 2^{-2^M(k+1)}). \tag{5.11}$$

Note that the scaling costs M=2∼4 pairs of shift–add operations. In general, we can say that the cost of Type IV is given by 2+M pairs of shift–add operations. For example, when the index k is 2, the scaling is:

$$\hat{m}_2 = (1 + 2^{-6})(1 + 2^{-12})(1 + 2^{-24}). \tag{5.12}$$

These four subtypes have three identical parts: Type I with one iteration, the scaling part of Type IV and the second iteration of Type II. These three parts can be integrated together by using multiplexers to select the data paths as shown in Figure 5.4, which has 2 adders, 2 shifters and 4 multiplexers. More detailed information about the μ–CORDIC can be found in [46, 108].

5.2.3 Adaptive μ–CORDIC iterations

To improve the computational efficiency, we perform 6 iterations per cycle and name it CORDIC–6. Since the global clock in a synchronous circuit is determined by the longest path, which also means that the maximum timing delay per iteration is 6 cycles (when the index k is 1, Type IV). Therefore, the inner iteration steps are repeated until they are close to the critical one. The required numbers of repetition are quoted in Table 5.1. For example, when the rotation angle index is k=8, it will repeat three times from the index k=8 to the index k=10. When the rotation angle index is k=20, it will repeat six times from the index k=20 to the index k=25. On the other hand, the number of iterations can be adjusted by selecting the average angle during the last

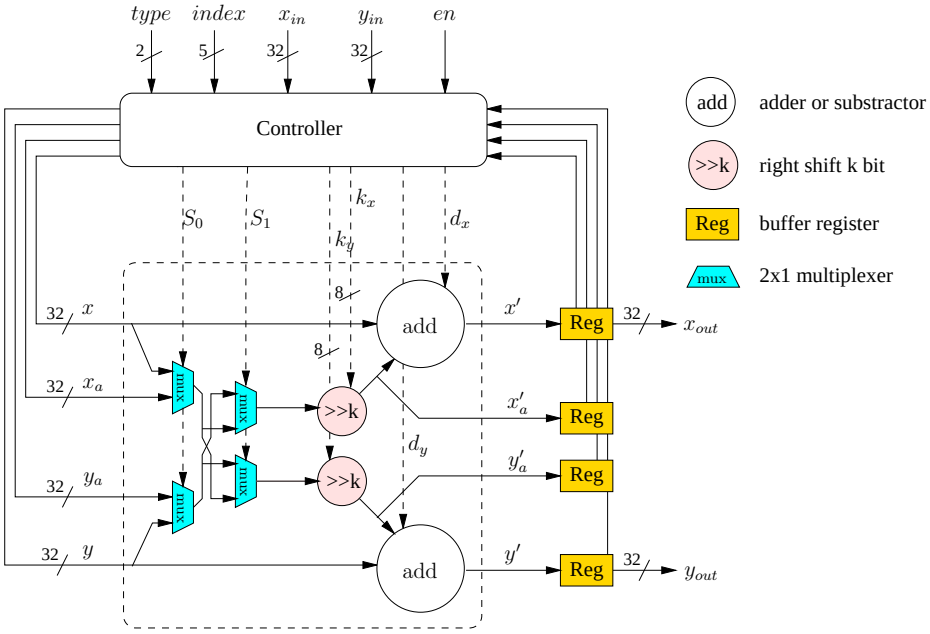


Figure 5.4: The block diagram of a scaling–free μ–CORDIC PE, including 2 adders, 2 shifters and 4 multiplexers

sweep and name it as CORDIC–mean. In this way, the iteration steps of adaptive μ–CORDIC are defined according to the structure of the matrix.

5.2.4 Exchanging inner and outer iterations

Modifying the inner iteration number of CORDIC processor will cause a tradeoff issue. Decreasing the iterations will result in an increased number of outer sweeps due to the imprecise inner iterations. Therefore, the Full CORDIC architecture requires fewer sweeps but needs more area. On the other hand, the simplified μ–CORDIC architecture can reduce the area but results in an increased number of outer sweeps. This problem becomes a tradeoff between the iteration numbers and the computational complexity. In the next section, we will further compare these rotation modes in Matlab.

5.3 Experimental Results

The regular Full CORDIC and the adaptive μ–CORDIC methods have been simulated in Matlab and implemented as circuit designs with the TSMC 45nm technology library for further detailed comparison.

5.3.1 Matlab Simulation

The Full CORDIC with 32 iteration steps, the μ–CORDIC with one iteration step and two different adaptive modes have been tested using numerous random symmetric matrices A of size 8×8 to 160×160 (i.e. EVD array size from 4×4 to 80×80). Figure 5.5 shows the average number of sweeps needed to compute the eigenvalues/eigenvectors for each size of EVD array, where the sweep number increases monotonically. Note that the stop criterion is $||A_{off}||_F \times 10^{-8}$. The $||A_{off}||_F$ is the Frobenius norm of the off–diagonal elements of A, i.e. $A_{off} = A - diag(diag(A))$. That means the sweep k will stop when the current $||A^k_{off}||_F$ is smaller than the $||A_{off}||_F \times 10^{-8}$.

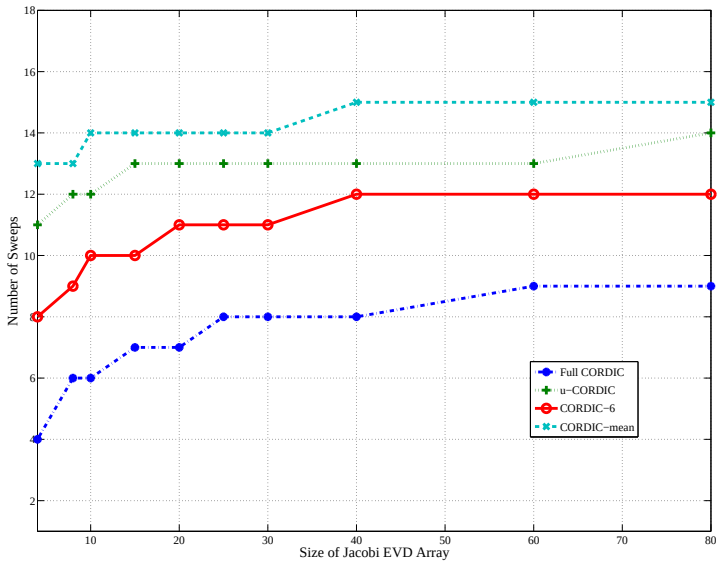


Figure 5.5: The average number of sweeps vs. array sizes for four rotation methods (μ–CORDIC, Full CORDIC and two adaptive methods).

When the Jacobi EVD array size is 10×10, the μ–CORDIC requires 12 sweeps while the Full CORDIC only requires 6 sweeps. If we adjust the inner rotations to six times, the sweep number will be 10, smaller than the μ–CORDIC but more than the Full CORDIC. On the other hand, using the average rotation angle to decide the rotation number as CORDIC–mean will require more sweeps than the others. Although the μ–CORDIC requires twice more sweeps per EVD computation than the Full CORDIC, it actually reduces the number of the inner CORDIC rotations, which results in an improved computational complexity. For example, a 10×10 array with the Full CORDIC PE needs 6 sweeps × 32 inner CORDIC rotations, the CORDIC–6 needs 10 sweeps × 6 inner CORDIC rotations, while the μ–CORDIC PE only requires 12 sweeps × 1 inner CORDIC rotation. In Figure 5.6, the average shift–add operations required for each rotation method with different size of testing EVD arrays is demonstrated, where Full CORDIC needs much more shift–add operations than others and the adaptive CORDIC–6 method can offer a compromise between the hardware complexity and the computational effort. Consequently, from an algorithmic point of view, doubtless we would prefer to realize the Jacobi method by utilizing the μ–CORDIC method. However, when it comes to nanoscale VLSI implementation, other design issues become also important.

Figure 5.7 shows the off–diagonal Frobenius norm versus the sweep numbers. When the array size is 10×10 with double floating precision, each rotation method will easily converge to the predefined stop criterion. Here the stop criterion is also defined as $||A_{off}||_F \times 10^{-8}$. In a similar situation, when the array size is 80×80 as illustrated in Figure 5.8, the convergence is also obtained. However, when using the single floating precision (default IEEE 754 single), it will no longer converge due to the insufficient operand precision.

Figure 5.9 and Figure 5.10 show the reduction of the off–diagonal Frobenius norm versus the sweep numbers for single floating precision. It can be noticed that the off–norms do not reach the convergence. Each size of EVD array has its own stop criterion (maximum offNorm reduction vs. sweep number). Therefore, we must first analyze the Frobenius norm of the off–diagonal elements in Matlab and then observe it until it reaches its maximal reduction. Afterwards, a lookup table has to be generated and directly assigned these stop criteria to the target hardware circuit or IP component.

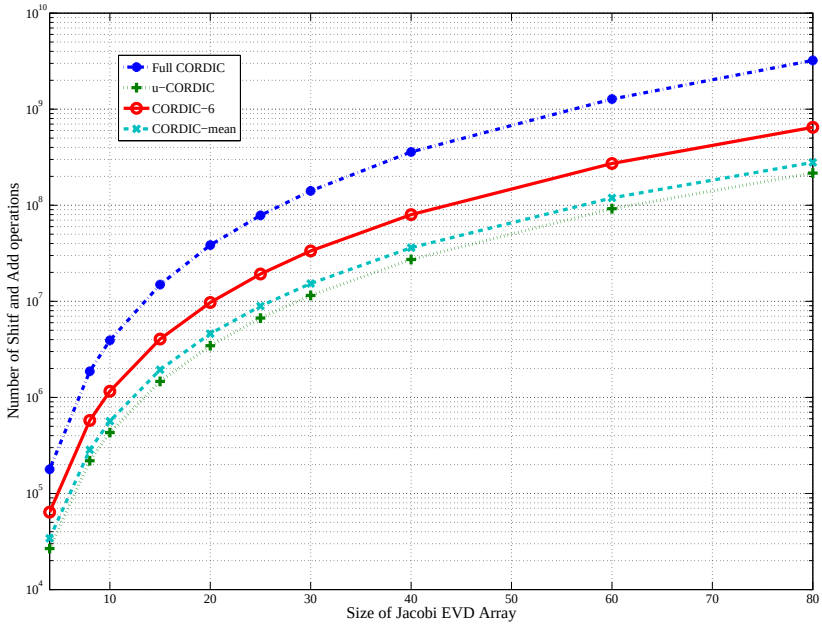


Figure 5.6: The number of shift–add operations for four rotation methods on different size of array

5.3.2 Using threshold methods

In Figure 5.9 and Figure 5.10, the decrease of the off–diagonal norm becomes slower with single floating precision, it is obvious that the rotations along these plateaus do not contribute to the overall result for small rotation angles. A simple way to reduce such inefficient operations is to apply a threshold strategy. That means in each sweep a threshold value is used to decide if a rotation is computed or not. The threshold method was already mentioned by Wilkinson in [128]. He worked with a fixed sequence of threshold values, lowering from sweep to sweep to reduce computation time. Every transformation regarding an off–diagonal element that is below the threshold value is bypassed. However, there is a problem for comparing floating point numbers in circuit design. Since the design complexity of the floating comparator is same as the floating adder, the benefit we could obtain from the simplified μ–CORDIC will become oblivious. Therefore, another simple way to predict the index E is presented in here, which came from $m \times 2^{E-127}$, where m is the mantissa and E is the exponent of an IEEE 754 single floating point number. Figure 5.12 shows the frequency distribution of

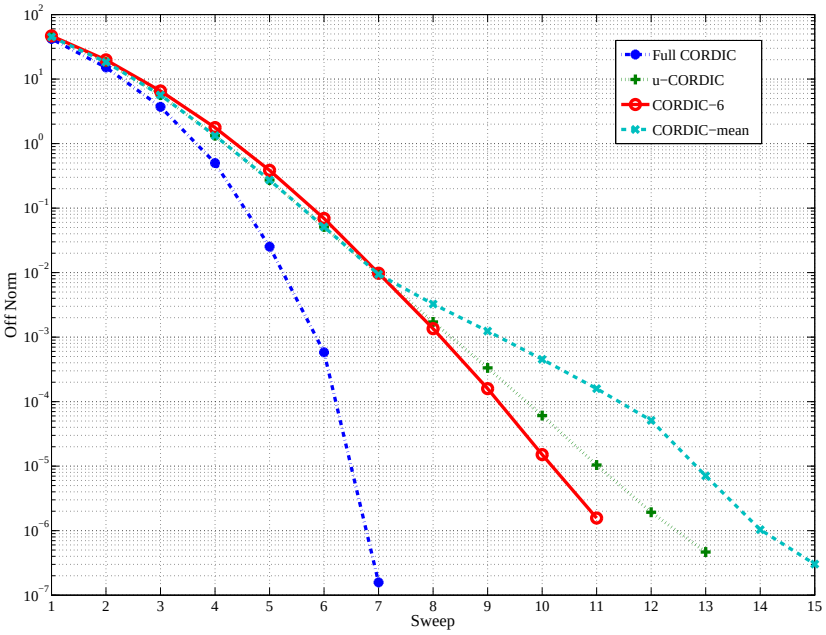


Figure 5.7: The required number of sweeps vs. off–diagonal norm for 10×10 Jacobi EVD array with double floating precision

the optimal rotation angle θ_{opt}'s exponent E over the sweep number for a Jacobi EVD array of size 10×10 in single floating precision. Additionally, Figure 5.13 shows the frequency distribution for the array size of 80×80.

It can be noticed that both average peaks appear in a range from 127 to 96 (equal to 0 to -32). It is very similar to the rotation index $n = 32$ for both Full CORDIC and μ–CORDIC. In order to simplify the comparator operations, this prediction can be utilized to bypass those small rotation angles which have very little contribution to the convergence. Figure 5.11 further shows the reduction of shift–add operations with the threshold strategy and the preconditioned E index method on different size of the array in IEEE 754 single floating precision. If the regular Full CORDIC is selected, a maximal reduction up to 40% can be achieved. For the μ–CORDIC and CORDIC–6, a reduction up to 10% of shift–add operations is obtained. Therefore, using the preconditioned E index method not only achieves a similar reduction of shift–add operations compared to the threshold strategy (sometimes even better), but also replaces the floating comparators by a lookup table.

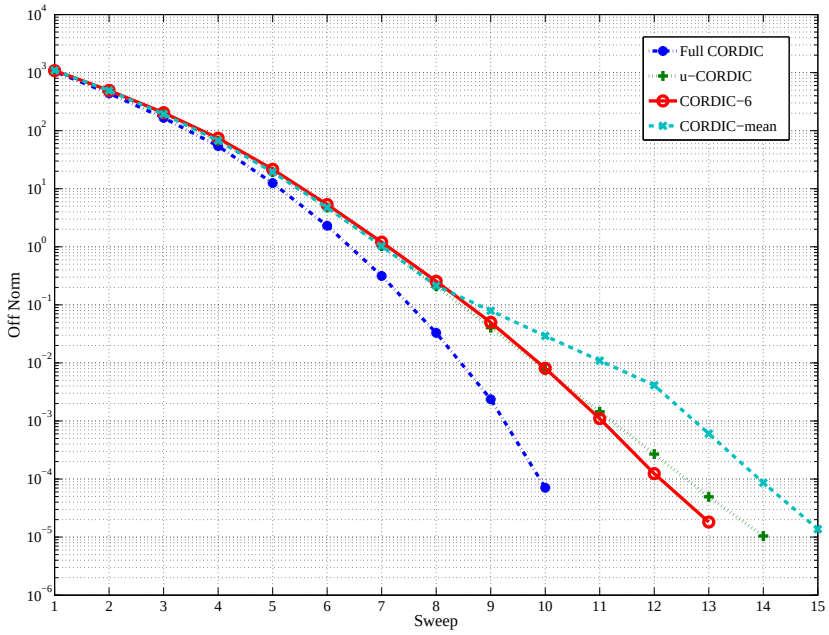


Figure 5.8: The required number of sweeps vs. off–diagonal norm for 80×80 Jacobi EVD array with double floating precision

5.3.3 Configurable Jacobi EVD Array

One of the major contributions in this thesis is to design a configurable Jacobi EVD interface, where a $n \times n$ ($n = 2, 4, 8, \ldots, 2^k, k \in N$) parallel systolic processor array can be generated through the C++ program directly as long as there is enough logic for implementation. In Figure 5.1, it can be easily noticed that there are similar interconnection properties between the neighbor PEs in this 4×4 systolic array. Essentially, these PEs can be categorized into six different types which are highlighted by dashed boxes in Figure 5.14. These six different types can help discover the similar property coding style for parameters design, especially in the data switching behavior between the neighbor PEs. According to their locations, they are "top", "bottom", "inner", "left", "right" and four "corner". The pseudo code for each type is listed in Appendix Table A.2.

In theory, once the off–diagonal elements of matrix A are small enough or the computation reaches the predefined maximal sweep number according to the Matlab analysis, then the systolic processor array will

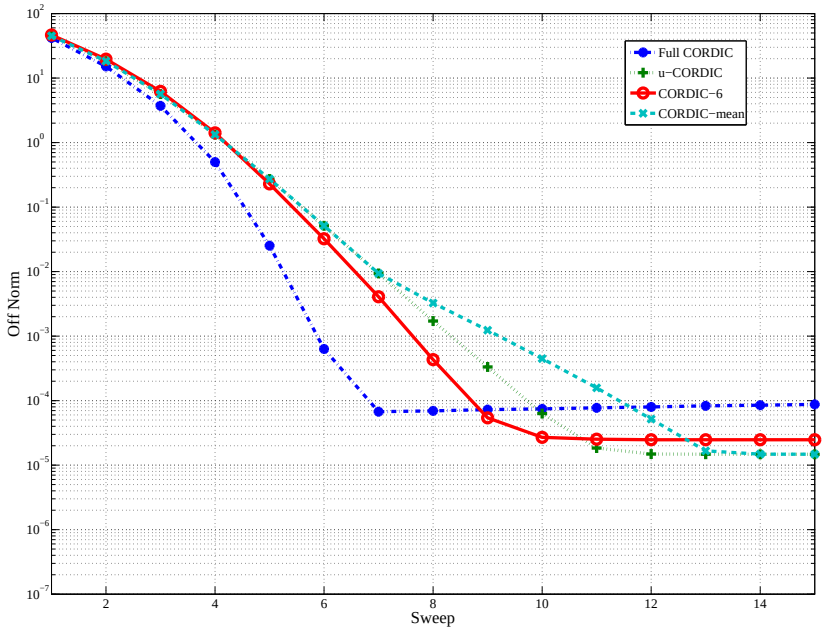


Figure 5.9: The required number of sweeps vs. off–diagonal norm for 10×10 Jacobi EVD array with single floating precision

stop the computation immediately. Therefore, a stop criterion can be defined as:

$$\mathrm{PC} = sweepmax \times [3(n-1) + \Delta + 3], \tag{5.13}$$

where n is the size of matrix A, Δ is the offset length of the $N-1{=}\frac{n}{2}-1$, and $sweepmax$ denotes the sweep number when the off–diagonal Frobenius norm is smaller than the predefined torrence value. Each sweep requires $3\times(n-1)+\Delta+3$ computational steps. Usually, the predefined sweep number can be predicted through testing numerous random matrices. In order to configure the EVD array for the matrix A with the size of $n \times n$, a parameterized design which depends on the size n for computing the eigenvalues in parallel is presented. This configurable parallel EVD array includes a PC counter, a controller and a configurable parallel systolic processor array as illustrated in Figure 5.15.

In Figure 5.15, "CT" is a counter for PC counting. The input "sweepmax" is a predetermined number according to the Equation 5.13. The

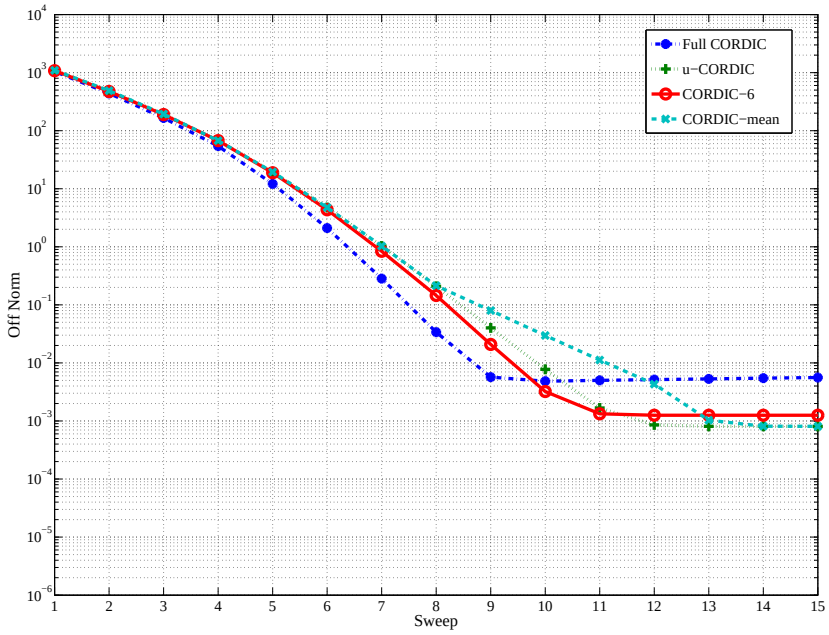


Figure 5.10: The required number of sweeps vs. off–diagonal norm for 80×80 Jacobi EVD array with single floating precision

signal "off" is the stop flag. The signal "stop" tells the parallel systolic array when to stop the computation. The signal "store" notifies the eigenvalues are ready for transmission to the output ports. The instructions for controlling the PEs are identical, if their Δ are equal too. There are N lines for "store" and "stop" singles to notify the N parallel subproblems. Note that the signal "loads" has 6 different instructions. One "*load*" instruction is for diagonal PEs. There are $N-1$ "$loadt$" instructions, N^2-N "$loada_{11}$" instructions, N^2-N "$loada_{12}$" instructions, N^2-N "$loada_{21}$" instructions and N^2-N "$loada_{22}$" instructions for off-diagonal PEs. These six instructions will notify the PEs when to load the input matrix, transmit the output eigenvalues/eigenvector or exchange the matrix elements with its neighbors.

For instance, when the single"LW" is asserted, the matrix A will first be assigned to the array through the "input" ports. Once "enable" is asserted, the EVD array will start to compute the eigenvalues. After several sweep rotations, if the "pc_in" is equal to the "sweepmax", it will stop the computation. The diagonal PEs will forward the results to the output ports. So far we assumed the n should be even for simplicity.

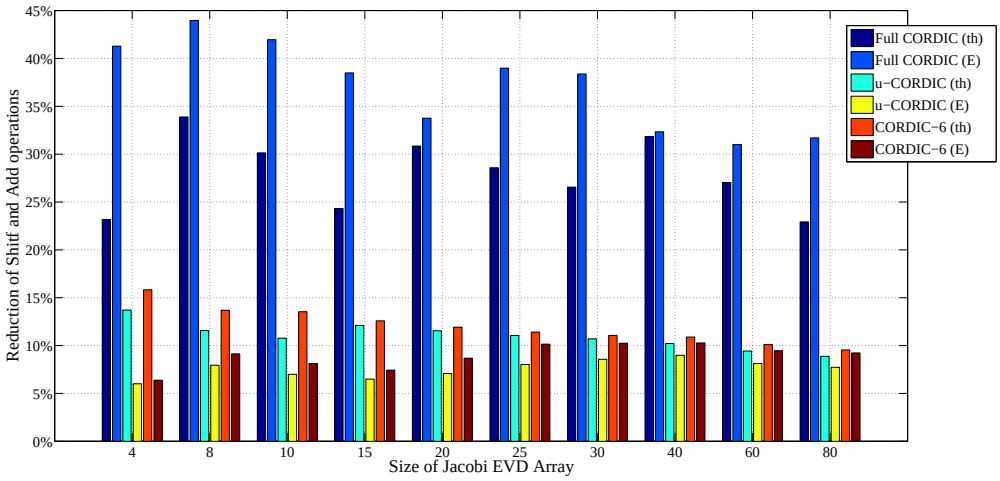


Figure 5.11: The reduction of shift–add operations (in percent) for three rotation methods with the threshold strategy and preconditioned E index on different size of array in IEEE 754 single floating precision

For odd n we can simply expand the matrix A by zeros.

5.3.4 Circuit Implementation

The presented μ–CORDIC has been modeled and compared with the folded Full CORDIC in VHDL. Later, these two methods are implemented into parallel EVD arrays with size 4×4 and 10×10 through a configurable interface separately. After that, they are synthesized by the Synopsys Design Compiler with TSMC 45nm standard cell library. It should be noticed that the word-length is 32 bits for IEEE 754 single floating precision. At the end, the SoC Encounter is used to perform the Place & Route. Figure 5.16 shows the chip layout with 100 μ–CORDIC PEs. We did not add any IO pads to the implementation and treated it as a standard macro design. Table 5.2 lists the synthesis results for area, timing delay and power consumption.

First of all, the combinational logic area and the power consumption of the μ–CORDIC PE is much smaller than for the Full CORDIC. Furthermore, in order to determine the time required to compute the EVD of a $n \times n$ symmetric matrix, we obtain:

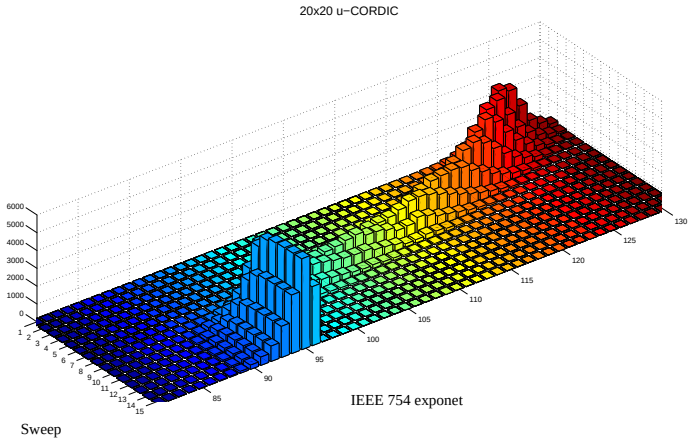


Figure 5.12: 3–D bar statistic view of the $l = E - 127$ index for adaptive index selection for 10×10 Jacobi EVD array with single floating precision

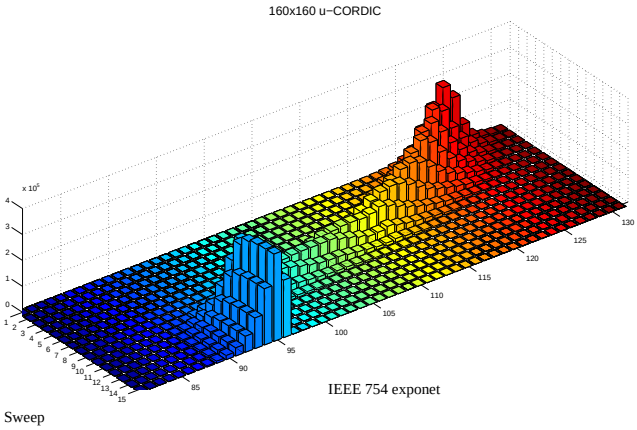


Figure 5.13: 3–D bar statistic view of the $l = E - 127$ index for adaptive index selection for 80×80 Jacobi EVD array with single floating precision

$$\begin{aligned} \mathrm{T}_{total} &= \mathrm{T}_{delay} \times \mathrm{K}_{iteration} \times \mathrm{K}_{sweep} \times [3(n-1) + \Delta + 3], \\ n &= 8, 16, 20, 30, \ldots 160, \\ \Delta &= \frac{n}{2} - 1. \end{aligned} \tag{5.14}$$

The total timing delay per EVD operation is defined as " the critical timing delay × the number of inner CORDIC rotations × average number of outer sweeps × size of the matrix A". We can observe that the total operation time is dependent upon the relation between the inner CORDIC rotations and the outer sweeps. For instance, comput-

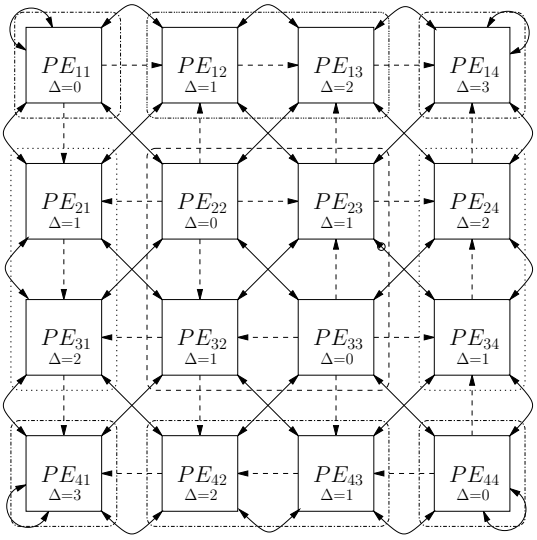


Figure 5.14: The locations of six different PE types in a 4×4 Jacobi EVD array

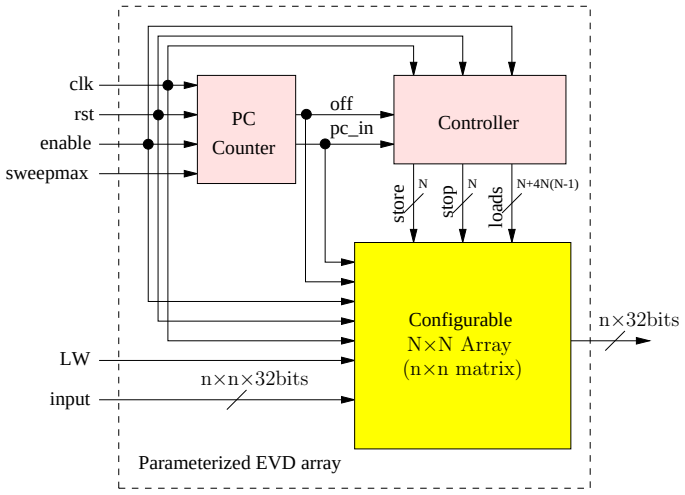


Figure 5.15: A configurable parallel Jacobi EVD design

ing the EVD on the 10×10 EVD array with the Full CORDIC, requires 4.286×32×7×(3×19+9+3) ≅ 66.24 us. Using the μ–CORDIC PE only requires 2.247×2×10×(3×19+9+3) ≅ 3.1 us. Therefore, a speed up by a factor of 21.4 can be obtained from reducing the number of inner CORDIC rotations. Although the reduction of power consumption is not so much due to extra multiplexers, μ–CORDIC's controller and a lookup table, it actually consumes much less energy per EVD computation due to the shorter computation time. Note that the μ–CORDIC PE requires two inner iterations in average due to the different rotation cycles (from six to one inner iteration) as shown in Table 5.1. Therefore, we can incorporate the synthesis results from Table 5.2 and the preconditioned E index to normalize the energy consumption with different

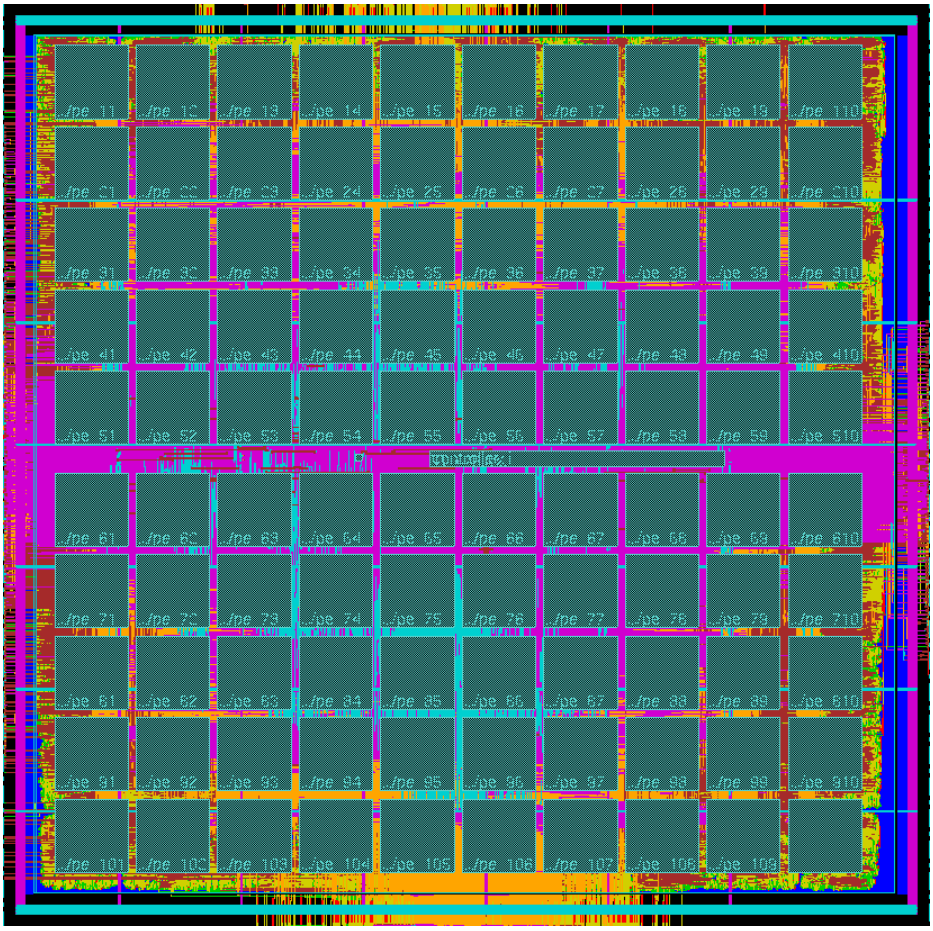


Figure 5.16: Final layout view of a 10×10 Jacobi EVD array with the μ–CORDIC PE with TSMC 45nm technology library.

size of EVD array. Figure 5.17 shows the predicted energy consumption results for size of the array from 4×4 to 80×80. Note that the energy consumption is shows on the logarithmic scale on the Y–axis. Both rotation methods consume much less energy than the Full CORDIC, where the CORDIC–6 can obtain an energy reduction factor of 40.9 and the μ–CORDIC can obtain an energy reduction factor of 104.3 in average compared to the Full CORDIC. In short, the μ–CORDIC not only achieves less combinational logic cell area and faster frequency, but also less energy consumption compared to the Full CORDIC. On the other hand, the CORDIC–6 consumes two times more energy than the μ–CORDIC, but the convergence speed is faster. Therefore, this becomes a tradeoff issue between the μ–CORDIC and the CORDIC–6 (computation time vs. energy consumption).

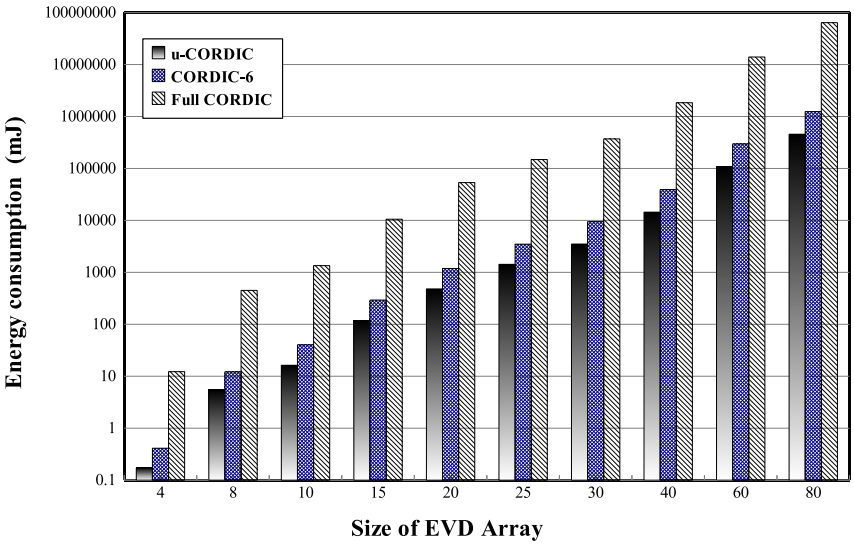


Figure 5.17: The energy consumption per EVD operation with each size of EVD array (operating at 100 MHz)

5.4 Summary

Computing the EVD by the parallel Jacobi method was selected as a typical example due to its robust convergence property. A configurable Jacobi EVD array has been implemented with both Full CORDIC and μ–CORDIC to explain the design concepts for parallel iterative algorithms. Using the preconditioned E index method not only reduced more than 35% shift–add operations for the Full CORDIC and 10% for the μ–CORDIC in average, but also omits the floating point number comparators. Afterwards, for a TSMC 45nm implementation, a detailed comparison between area, timing delay and power/energy consumption was made. It was shown that the modification of inner iterations not only reduced the size of the combinational logic, improved the overall computation time but also saved the energy consumption. Furthermore, the CORDIC–6 provided a compromised solution between the computation time and the energy consumption compared to the μ–CORDIC.

Table 5.1: The lookup table for μ–rotations CORDIC with 32–bit accuracy, showing the rotation type, the $2\times\tan\theta$ angle, the required shift–add operations for rotation and scaling, the required cycle delay and repeat numner for CORDIC–6 [109].

index k	type	angle $2\times\tan\theta$	shift–add rot.	shift–add sca.	cycle cnt.	cycle re.	index k	type	angle $2\times\tan\theta$	shift—add rot.	shift—add scl.	cycle cnt.	cycle re.
1	IV	1.49070	4	8	6	1	17	I	$1.52588\,10^{-5}$	2	0	1	6
2	IV	0.54296	4	6	5	1	18	I	$7.62939\,10^{-6}$	2	0	1	6
3	IV	0.25501	4	6	5	1	19	I	$3.81470\,10^{-6}$	2	0	1	6
4	IV	0.12561	4	4	4	1	20	I	$1.90735\,10^{-6}$	2	0	1	6
5	III	$6.25841\,10^{-2}$	6	0	3	2	21	I	$9.53674\,10^{-7}$	2	0	1	6
6	III	$3.12606\,10^{-2}$	6	0	3	2	22	I	$4.76837\,10^{-7}$	2	0	1	6
7	III	$1.56263\,10^{-2}$	6	0	3	2	23	I	$2.38419\,10^{-7}$	2	0	1	6
8	II	$7.81266\,10^{-3}$	4	0	2	3	24	I	$1.19209\,10^{-7}$	2	0	1	6
9	II	$3.90627\,10^{-3}$	4	0	2	3	25	I	$5.96046\,10^{-8}$	2	0	1	6
10	II	$1.95313\,10^{-3}$	4	0	2	3	26	I	$2.98023\,10^{-8}$	2	0	1	6
11	II	$9.76563\,10^{-4}$	4	0	2	3	27	I	$1.49012\,10^{-8}$	2	0	1	6
12	II	$4.88281\,10^{-4}$	4	0	2	3	28	I	$7.45058\,10^{-9}$	2	0	1	5
13	II	$2.44141\,10^{-4}$	4	0	2	3	29	I	$3.72529\,10^{-9}$	2	0	1	4
14	II	$1.22070\,10^{-4}$	4	0	2	4	30	I	$1.86265\,10^{-9}$	2	0	1	3
15	II	$6.10352\,10^{-5}$	4	0	2	5	31	I	$9.31323\,10^{-10}$	2	0	1	2
16	I	$3.05176\,10^{-5}$	2	0	1	6	32	I	$4.65661\,10^{-10}$	2	0	1	1

Table 5.2: Area, Delay and Power Consumption results of 4×4 and 10×10 Jacobi EVD arrays with the TSMC 45nm technology.

		4×4 Full CORDIC	4×4 μ–CORDIC	10×10 Full CORDIC	10×10 μ–CORDIC
Area	Combinational	0.847 mm^2	0.296 mm^2	5.143 mm^2	1.829 mm^2
	Noncombinational	0.390 mm^2	0.123 mm^2	2.306 mm^2	0.833 mm^2
	Total	1.237 mm^2	0.419 mm^2	7.449 mm^2	2.662 mm^2
Power	Cell	62.283 mW	18.239 mW	388.379 mW	123.215 mW
	Net	0.465 mW	0.433 mW	2.993 mW	2.678 mW
	Leakage	11.909 mW	3.765 mW	86.136 mW	23.966 mW
	Total	74.657 mW	22.437 mW	477.508 mW	149.859 mW
Timing	Critical	4.454 ns	1.213 ns	4.286 ns	2.247 ns
	Frequency	224.5 MHz	824.4 MHz	233.3 MHz	445 MHz

6 Sparse Matrix–Vector Multiplication on Network–on–Chip

A new concept for accelerating Sparse Matrix–Vector Multiplication (SMVM) operations by using Network–on–Chip (NoC) is introduced in this chapter. In traditional circuit design on-chip communications have been designed with dedicated point-to-point interconnections or shared buses. Therefore, regular data transfer is the major concern of many parallel implementations. However, when dealing with the parallel implementation of SMVM, which is the main step of most iterative algorithms for solving systems of linear equations, the required data transfers are usually dependent on the sparsity structure of the matrix and can be extremely irregular. Using a NoC architecture makes it possible to deal with an arbitrary structure of the data transfers. In this chapter, a configurable SMVM calculator based on the NoC architectures (**SMVM–NoC**) is implemented with arbitrary size of $p \times p$ ($p = 2, 4, 8, \ldots, 2^k, k \in \mathbb{N}$) with the IEEE-754 single floating point precision in an FPGA.

This chapter is organized as follows: In Section 6.1, a brief introduction to recent works on SMVM will be given first. After that, Section 6.2 shows the main concept of the parallel NoC based SMVM architecture. Then, the hardware implementation will be described in Section 6.3 in detail. In Section 6.4, the experimental results are given. Section 6.5 summarizes this chapter.

6.1 Introduction of Sparse Matrix–Vector Multiplication

Over the past 30 years, researchers and scientists have tried various approaches to mitigate the poor performance of sparse matrix computations, such as reordering the data to diminish wasted memory bandwidth, modifying the algorithms to reuse the data, and even building specialized memory controllers. Despite these efforts, the sparsity of the matrices still dominates the performance of the SMVM computations [81].

SMVM is used in many applications. For example, Finite Element Method (FEM) is a widely applied engineering analysis tool which is based on obtaining a numerically approximate solution for a given mathematical model of a structure. The resulting linear system is characterized by the system matrix A which is usually large and sparse [29, 37, 83, 98]. In general, iterative solvers, such as the Conjugate Gradient (CG) method, are almost dominated by SMVM operations (usually more than 95%). The CG method is the most popular iterative method for numerically solving a system of linear equations, $A \cdot x = b$, where A is a Symmetric Positive-Definite (SPD) sparse matrix [52, 77]. Moreover, Google's PageRank (PR) Eigenvalue problem is considered to be the world's largest sparse matrix calculation. This problem is also dominated by SMVM operations where the target matrix is extremely sparse, and unstructured.

In the last decade, many researchers were dealing with the integration of pipelining and parallelism inherent in the SMVM computation in hardware designs. Sun et al. [114] proposed a SMVM design containing many Processing Elements (PEs) with pipelined floating-point units in FPGA. Gregg et al. [39] built a specialized memory controller to accelerate the SMVM. Götze and Schwiegelshohn [48] presented a systolic algorithm which allows the parallel execution of SMVM in a dedicated VLSI circuit. Williams et al. [129] used multi-core environment, using the heterogeneous x86 based quad-core CPU to accelerate the SMVM computation in parallel. Google's PR problem has also been investigated for acceleration with FPGA in [76, 141].

Conventional SMVM architectures are usually focused on a dedicated internal chip interconnection to forward vector components and nonzero matrix elements among several processors. For instance, the fat-tree style design, which requires pre-sorting and pre-ordering before the data input [65], will become extremely difficult when the matrix is very large and sparse. This motivates to use a packet–switched network for solving these design issues. This packet switching architecture is called NoC as earlier described in Section 2.3.

In this chapter, an FPGA accelerator for SMVM–NoC architecture is presented in order to solve the design problems which arise from large sparse matrices with their extremely irregular structures. The basic idea of this concept is to implement an on-chip internal network as the main transmission bone for the data transfers required for the SMVM computation. The presented vision provides a superior method to handle large sparse matrices, especially it can deal with arbitrary sparsity structures. The 2×2, 4×4 and 8×8 SMVM-NoC calculators have been implemented on Xilinx Virtex-6 platform. According to the implementation results, utilizing the packet forwarding functionality is beneficial concerning heterogeneous IP integration and flexibility.

6.2 SMVM on Network-on-Chip

6.2.1 Sparse Matrix-Vector Multiplication

A typical SMVM operation can be expressed as follow:

$$A \cdot x = b, \tag{6.1}$$

where x and b are vectors of length n and A is a $n \times n$ sparse matrix. Since the matrix A can be very large and sparse, iterative solvers are typically used to solve the system of linear equations due to their low storage requirements [45]. Many researchers have already utilized the pipelining and the parallelism to improve the performance. However, the computational complexity is still determined by the sparsity of the matrix A [29].

In practice, the sparse matrix is usually stored in a Compressed Sparse Row (CSR) format, in which only the nonzero matrix elements will be stored in contiguous memory locations. In CSR format, there are three vectors: *val* for nonzero matrix elements; *col* for the column index of the nonzero matrix elements; and *ptr* stores the locations in the *val* vector that start a new row [141]. As an example, consider a simple SMVM operation with 5×5 sparse matrix A as follows:

$$\begin{bmatrix} 4 & 0 & 8 & 0 & 0 \\ 6 & 0 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 7 & 0 & 0 & 0 & 4 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}. \tag{6.2}$$

The CSR format of this matrix can be described by three vectors given below:

val:	4	8	6	3	1	5	7	4
col:	1	3	1	3	2	4	1	5
ptr:	1	3	5	6	7	8		

When a CSR format matrix is read in successively, the column index of the nonzero element can be obtained from *col*, while the row index is indirectly indicated by *ptr*. This format can be easily facilitated into the data packets for a switching network.

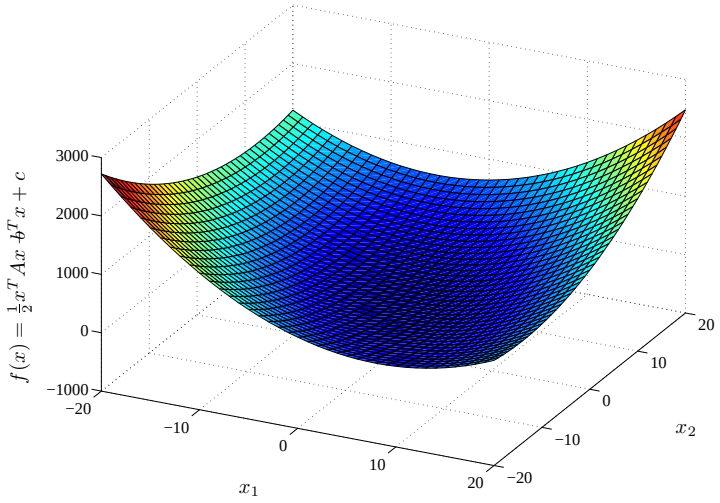


Figure 6.1: The quadratic surface of the $f(x)$

6.2.2 Conjugate Gradient Solver

CG iterative solver, which was presented in 1952 by Magnus Hestenes and Eduard Stiefel [52], is perhaps the best known iterative method used for solving very large and extremely sparse systems of linear equations. The starting point in the derivation is to consider how we might be able to minimize the quadratic function: $f(x) = \frac{1}{2}x^T Ax - b^T x + c$, where $c \in \mathbb{R}$, $b \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$ is assumed to be a SPD matrix. The minimum value of $f(x)$ is $-b^T A^{-1} b/2$, achieved by setting $x = A^{-1}b$. Therefore, minimizing $f(x)$ and solving $A \cdot x = b$ are equivalent problems. For example, when we define them as:

$$A = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}, \quad b = \begin{bmatrix} 2 \\ -8 \end{bmatrix} \text{ and } c = 0, \tag{6.3}$$

we obtain a $(n+1)$–dimensional convex parabolic surface as illustrated in Figure 6.1. Minimizing $f(x)$ corresponds to the solution for $A \cdot x = b$. That means the solution is the lowest point on the surface, which can be obtained by using the steepest descent method [45].

However, when the condition of A increases, the steepest descent method will barely converge to the solution. A better approach is to use A-conjugate direction for the descent. Algorithm 1 briefly describes

a simple scenario of the CG algorithm. This while loop approximates the next value of x (estimated solution), r (residual) and p (search direction) iteratively. Each iteration step will yield a better x by "walking downhill" in the A-conjugate direction of vector p. A convergence test, as idealized by the while clause set, causes the CG algorithm to terminate if the residual r vector is smaller than the predefined tolerance value. More than 95% of the computational efforts are SMVM operations: $q \Leftarrow Ap^k$ [29].

Algorithm 1 Conjugate Gradient Algorithm [45]

Require: A is $n \times n$ SPD matrix, x and b are vectors of length n.
Ensure: Congate(A, x, b, tol)
  $x^{(0)} \Leftarrow x_0$ initial guess
  $p^{(0)} \Leftarrow r^{(0)} \Leftarrow b \Leftarrow Ax^{(0)}$
  $k \Leftarrow 0$
  while $(norm(r^k) \geq tol\ )$ **do**
    $q \Leftarrow Ap^k$ (SMVM operation)
    $\alpha \Leftarrow (r^k, r^k)/(p^k, q)$
    $x^{(k+1)} \Leftarrow x^k + \alpha p^k$
    $r^{(k+1)} \Leftarrow r^k - \alpha q$
    $\beta \Leftarrow (r^{(k+1)}, r^{(k+1)})/(r^k, r^k)$
    $p^{(k+1)} \Leftarrow r^{(k+1)} + \beta p^k$
    $k \Leftarrow k + 1$
  end while

6.2.3 Basic Idea

In order to accelerate the performance of the iterative solvers, a SMVM platform was built based on the NoC architecture according to the earlier reports [110, 111], which contains $p \times p$ PEs ($p = 2, 4, 8, \ldots, 2^k$ $k \in \mathbb{N}$). It is connected by a 2–D mesh network. The packet forwarding function is used to transmit the data.

For instance, Figure 6.2 shows a simple scenario (see Equation 6.2) for mapping a parallel SMVM–NoC computation. The circle symbols denote the nonzero elements from the sparse matrix, the square symbols denote the vector elements, and the rhombus symbols denote the results, respectively. First of all, the vector components x_j have already been

assigned to PE according to the predefined routing addresses. After that nonzero matrix elements A_{ij} will be distributed to the corresponding PE according to the coordinate index i and j through the mesh network; i.e. vector x_1 arrives at the PE with A_{11}, A_{21}, and A_{51} elements, vector x_2 arrives at the PE with A_{32} element, vector x_3 arrives at the PE with A_{13} and A_{23} elements, vector x_4 arrives at the PE with A_{44} element, and vector x_5 arrives at the PE with A_{55} element in the network, respectively. Now the partial products $b_i^{(j)} = A_{ij} \cdot x_j$ must be computed and then the $b_i^{(j)}$ are accumulated to form $b_i = \sum b_i^{(j)}$. Note that the number of nonzero elements n_z is usually larger than the number of PEs. The detailed mapping method will be explained in Section 6.3.6.

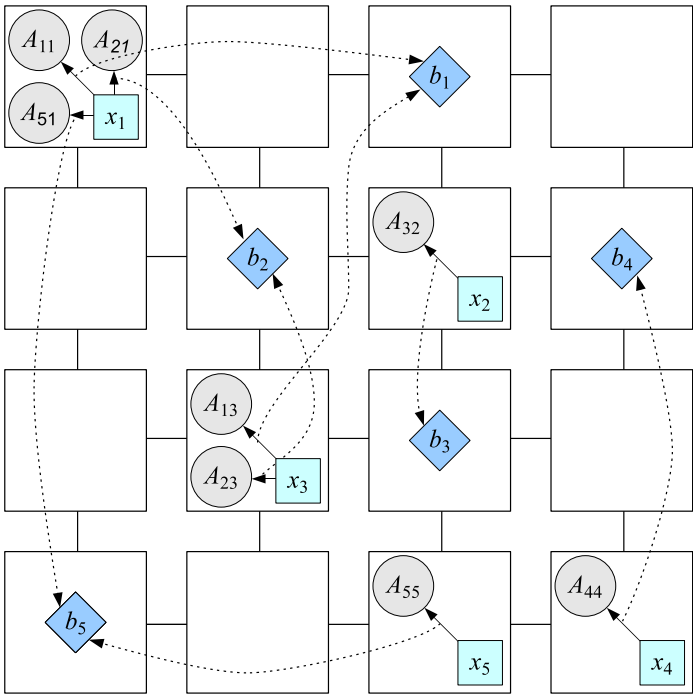


Figure 6.2: A direct mapping of parallel SMVM operations based on the NoC architecture

On the other hand, we can also reverse the first and second step so that nonzero matrix elements A_{ij} will be distributed to the PE array before broadcasting the vector components x_j. This inverse mapping is shown in Appendix Figure B.6. In this way, dense matrix problem can be handled when the number of nonzero matrix elements is far more than the vector components. This, however, does not lead to any modification of the proposed architecture, because this could be done by using the MicroBlaze to modify the packet's headers. For reason of simplicity at beginning, the first scenario is selected.

6.3 Implementation

In this section, the method for mapping the proposed parallel SMVM operations into NoC architecture will be described in detail. Figure 6.3 shows the system level view of a 4×4 SMVM-NoC in Xilinx Virtex–6, including two OPB interfaces for communication between the SMVM–NoC and the MicroBlaze embedded processor (using as packet controller). Therefore, we can first store those CSR based sparse matrices in DDR2 memory and then decompose the CSR format through the MicroBlaze. After that, packets will be injected into the SMVM–NoC.

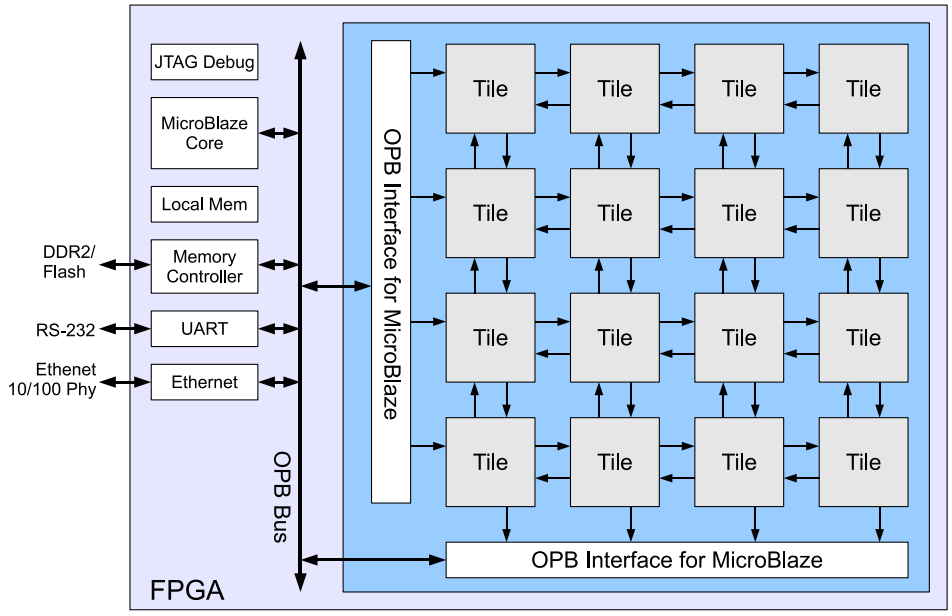


Figure 6.3: The system level view of a 4×4 SMVM-NoC in Xilinx Virtex–6

6.3.1 Packet Format

In packet switching, a message can be transmitted with a fixed-length packet. The head part of a packet contains routing and control information, which is referred to as a packet header, the rest is referred to as a payload. Each packet is individually routed from source to destination by using the given routing algorithm. Packets are buffered in the FIFO before they are forwarded to next hop in the network.

Table 6.1: Packet Format

Bit	Functionality
50	Finish flag
49	OPCode:0⇒Multiplication and 1⇒Accumulation
48	DataType:0⇒Matrix and 1⇒Vector
40-47	Y–Coordinate
32-39	X–Coordinate
0-31	Payload

The packet format that is used to communicate over the proposed NoC architecture is summarized in Table 6.1. The packet is mainly constituted by the header and the payload. The header contains three flags and two addresses. The three flags are named Finish, OPCode, and DataType. The Finish flag tells the routers that this packet contains the result of the SMVM operation and needs to be forwarded to the MicroBlaze controller through the exit port. The OPCode informs the PE which floating-point operation (i.e. multiplication or accumulation) is going to be performed, and the DataType flag represents the type of input data (i.e. matrix element or vector element). The objective of two addresses (i.e. X–Coordinate and Y–Coordinate) is to locate the position of the PE for the multiplier or the accumulator.

6.3.2 Switch Architecture

The switch is the most important component concerning the NoC performance. However, the area and the power consumption are also important design criteria for embedded systems. For this reason, a conventional 5×5 crossbar switch is now separated into two small 3×3 crossbar switches in order to reduce the area overhead [68]. Figure 6.4 illustrates a typical switch component for the local PE to communicate with its neighbor PEs consists of a set of input/output ports, dual-crossbar switches, four input FIFOs and controllers. The switch has five data input ports and five data output ports, named North, East, West, South, and Local, respectively. The Local port is connected to its PE, and the North, East, West, South ports are connected to the mesh network.

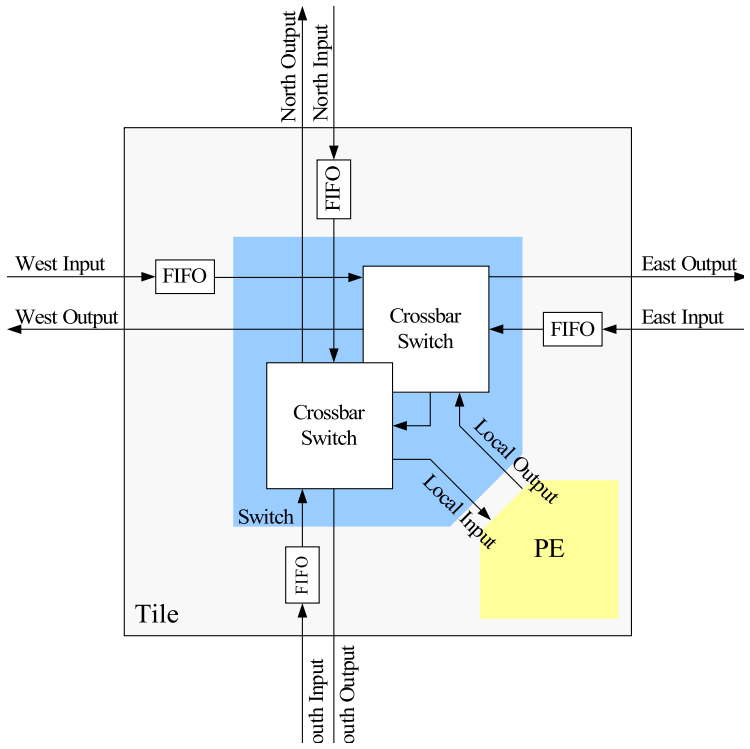


Figure 6.4: Detailed switch interconnection including two 3×3 crossbars, five I/O ports and four FIFOs

Each packet needs at least four cycles for transmitting a packet to next hop. In the first cycle, the PE will read a packet from the FIFO and then decomposes the packet in the second cycle.

The packet will be routed in the third cycle. If the desired port is available the packet will be output, otherwise it will wait until the output port is free. Here the Round-Robin strategy is selected to prevent the deadlock hazard when multiple packets arrive in a switch at the same time [35]. This switching approach is usually known as the store–and–forward method. It is a packet switching protocol in which the node stores the complete packet and forwards it based on the information within its header. Thus the packet may stall if the router in the forwarding path does not have enough buffer space [16]. However, for the simple and easy implementation of the presented SMVM–NoC in the FPGA, store–and–forward routing is preferred at beginning.

On the other hand, wormhole switching combines packet switching with the data streaming quality of circuit switching to achieve a min-

imal packet latency. In here the packets are divided into many small flits. The router will first check the header of the packet for determining the next hop and immediately forward it. The subsequent flits are also forwarded as they arrive. This causes the packet to worm its way through the network. Flits are acting like ants, hence gave the name. However, when a stalling happens, this method has an unpleasantly expensive side effect of occupying all the links that the worm spans. Furthermore, virtual cut–through has a forwarding mechanism similar to the wormhole switching. However, before forwarding the first flit, the node waits for a guarantee that the next hop in the path will accept the entire packet. Thus if the packet stalls, it collects and waits at the current node without blocking any links. Of course, the performance of wormhole switching method and virtual cut–through method are better than the simplest store–and–forward routing, but they also require larger area overhead for implementation.

6.3.3 Pipelined Switch Architecture

In order to improve the performance of the switch, the selected 3×3 crossbar switch is divided into five pipelined stages as shown in Figure 6.5. These five stages are named as Fetch, Routing, Channel Request, Channel Acknowledgment and Output. The structure of pipelined switch is similar to non–pipelined switch, but pipeline buffers are inserted between these five stages. Fetch stage reads a packet when there is a packet available in the FIFO and forwards it to the next stage. Routing stage receives a packet from the Fetch stage and decomposes the header information for routing. Channel Request stage receives information from the Routing stage and sends a channel request to the crossbar. The packet will be forwarded if the desired channel is not occupied, otherwise it will be stalled until the requested channel is free. Ideally, a three-port switch can transmit three packets at each clock cycle. There are two conditions that stall the pipeline, either output FIFO full or the requested channel has been occupied by another request. When a stall event happens, the switch will first move the packet in the last stage to a stall register since the Fetch stage cannot predict the stall event while reading the input FIFO. This might result in one more packet that has been read after stall happened. After the stall event is dissolved, the switch will retrieve the packet from the stall

register.

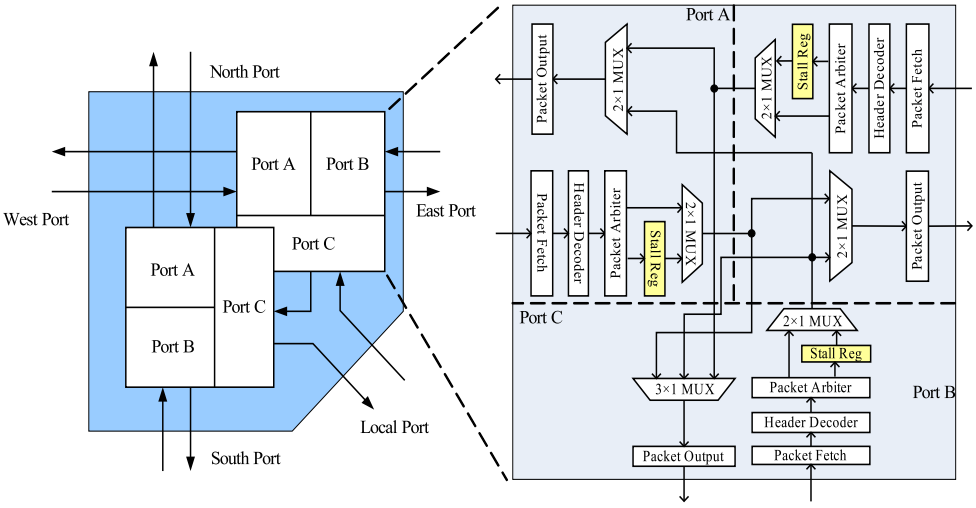


Figure 6.5: A 5–stage pipelined switch with two 3×3 crossbars, five I/O ports and four FIFOs

6.3.4 Routing Algorithm

Although the packet–switched network can forward the data automatically, the design of the network communications is still complicated. The routing decisions are only made locally and some routes can be more efficient than others. Therefore, different kind of network topologies (ring, star, spider, mesh or irregular), routing schemes (direct-route, adaptive-route, broadcast or multi-cast) and switch architectures (store–and–forward, wormhole switching or virtual cut-through) influence the performance intensively [54]. For simplicity and flexibility of the proposed SMVM–NoC hardware implementation, the direct routing algorithm is selected (a.k.a. XY deterministic routing). In XY routing, the switches in the network are indexed by their XY coordinates. When a switch receives a packet, it will first extract the header information of this packet and arbitrate the direction, then transmit to the next hop. Each packet is first routed along X coordinate and then along Y coordinate until the packet reaches its destination.

6.3.5 Processing Element

The PE is designed to deal with the floating-point operations for the SMVM computation as illustrated in Figure 6.6. This dedicated PE contains a controller, a single floating point multiplier, a single floating point adder, a data arbiter, a 2×1 multiplexer, and four FIFOs. Multiplier and adder are used to perform multiplication and accumulation, respectively. The data arbiter receives the packet from the switch and extracts the header. Then it forwards the data to the FIFOs or data lookup tables. The multiplexer is used to select one of the two result FIFOs as data output. The Vector Table and the Sum Table can contain at most 16 matrix vector elements and 16 accumulation results for each. Note that the default data length is 32 bits for the single floating precision operands, which is directly generated from the Xilinx ISE core library. It is also possible to configure it as 64 bits for the double precision or 80 bits for the extended double precision.

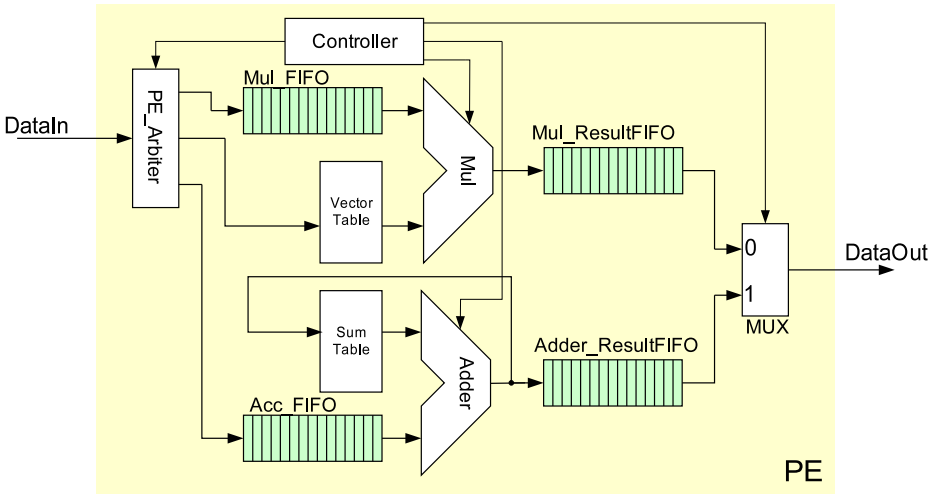


Figure 6.6: Schematic view of the PE for the SMVM–NoC platform

6.3.6 Data Mapping

When mapping a SMVM operation into a 4×4 SMVM-NoC, we first need to pack the nonzero elements from the CSR based sparse matrix and the corresponding vector elements with a proper header. Packing examples are listed in Table 6.2, the two addresses (X–Coordinate and Y–Coordinate) are separated into three sub parts for each. The lower 2

bits of the two addresses are used to indicate the horizontal and vertical positions of PE for multiplication. The third and fourth bits are used to address the PE for performing the accumulation operation in the same way. Note that these two address lengths should be long enough when the size p of SMVM-NoC increases. The higher 4 bits of X-Coordinate are used as an index for Vector Table to cache 16 entries in PE for multiplication, whereas the higher 4 bits of Y-Coordinate are used for Sum Table to buffer 16 entries for accumulation.

If we look at Figure 6.2 and Table 6.2 as an example, the nonzero elements A_{11}, A_{21} and A_{51} need to perform multiplications with the vector element x_1. The addresses of these three nonzero elements and one vector element should be mapped to the same PE. This means that the lower 2 bits in those packets have to be identical (emphasized in italic style). On the other hand, the multiplication results ($A_{21} \times x_1$) and ($A_{23} \times x_3$) need to be accumulated in the same PE so that the third and fourth bits of A_{21} and A_{23} and sum index (emphasized in bold style) should be identical, too.

Table 6.2: An example of packing vector elements and nonzero matrix elements into packet format (× denotes empty and Bold/Italic fonts denote that these packets have to be mapped on the same PE.)

Finish Flag	Operation Code	Data Type	Y–Coordinate			X–Coordinate			Pay load
			Sum	Acc	Mul	Vector	Acc	Mul	
0	×	0	× × ××	××	*10*	*0110*	××	*11*	x_1
0	×	0	× × ××	××	10	0001	××	10	x_3
0	0	1	0111	11	*10*	*0110*	11	*11*	A_{11}
0	0	1	**0100**	**10**	*10*	*0110*	**10**	*11*	A_{21}
0	0	1	1010	10	*10*	*0110*	11	*11*	A_{51}
0	0	1	**0100**	**10**	10	0001	**10**	10	A_{23}

However, sometimes the sparse matrix will be larger than the maximal capacity. In this case, we need to collect as many nonzero elements as the maximum capacity of vector elements can hold. Then we send those packets to the SMVM-NoC for computation. Once the computation is finished, we receive the result from SMVM-NoC as a temporary result, and send another data set for further computation. Finally, all temporary results will be combined in the MicroBlaze processor to get the multiplication result.

6.4 Experimental Result

6.4.1 FPGA Implementation

The proposed SMVM-NoC platform as illustrated in Figure 6.3 has been modeled in Verilog HDL, synthesized by the Xilinx ISE 11.5 and verified on the Xilinx ML–605 development kit with the Xilinx Platform Studio 11.5 [136]. The synthesized resource utilization reports with pipelining are listed in Table 6.3. After that, a set of random matrices from the size of 16×16 to 256×256 with sparsity from 10% to 50% is tested. In Figure 6.7, an overall comparison was made between Pentium-4 PC, non–pipelined SMVM-NoC and pipelined SMVM-NoC with different size of sparse matrice. Obviously, the pipelining can improve the performance but results in a huge area overhead. For example, when the matrix size is 128×128, the performance of the proposed pipelined SMVM–NoC can reach 580 MFlops with the size of 4×4 and 910 MFlops with the size of 8×8 in average. Therefore, it can obtain a speed up by a factor of 2.8∼4.4 compared to the Pentium-4. The 8×8 pipelined SMVM–NoC is too large for device mapping on the Xilinx Virtex–6. The peak performance of the proposed SMVM-NoC platform with the mesh network is 25.47 GFlops in theory. Each PE can perform at most two floating-point operations for each clock cycle. The peak performance can be obtained as:

$$Performance_{peak} = FPOPC * PEC * f, \tag{6.4}$$

where $FPOPC$, PEC and f are the Floating-Point Operations Per Cycle in a single PE, the number of processor and the maximal clock frequency.

Table 6.4 shows the comparison between the non–pipelined and the pipelined switches in terms of performance, resource utilization and power consumption. An average performance enhancement by 74% is achieved by integrating the pipelined switch into the SMVM-NoC. The insertion of pipeline registers results in an increase of resource utilization in terms of Slice Registers and Slice LUTs by 35% and also an increase of dynamic power consumption by 13%.

Table 6.3: Synthesis Results of pipelined SMVM-NoC Architecture in Xilinx Virtex-6 (XC6VLX240T-1FF1156)

Size	Logic utilization	Used	Available	Util.	Freq.	Peak Flops
2×2	Slice Registers	16,171	301,440	5%	286 MHz	2.29 GFlops
	Slice LUTs	18,002	150,720	11%		
	DSP48E1	20	768	3%		
4×4	Slice Registers	69,224	301,440	22%	275 MHz	8.8 GFlops
	Slice LUTs	77,540	150,720	51%		
	DSP48E1	80	768	10%		
8×8	Slice Registers	254,960	301,440	84%	199 MHz	25.47 GFlops
	Slice LUTs	318,076	150,720	211%		
	DSP48E	320	768	41%		

Table 6.4: The performance comparison between non–pipelined 4×4 SMVM-NoC and pipelined 4×4 SMVM-NoC (NZs: Nonzero Elements)

Item		Non–pipelined	Pipelined	Comment
MFlops	256 NZs	171.24	318.5	+86%
	1024 NZs	235.27	454.8	+93%
	4096 NZs	227.58	382.75	+68%
	16384 NZs	244.7	367.05	+50%
Util.	Slice Reg	52,609	69,224	+32%
	Slice LUTs	56,322	77,540	+38%
	DSP48E1	48	48	0%
Power	Dynamic Power	0.80626W	0.90937W	+13%
	Static Power	2.20761W	2.21330W	+0.3%
	Total Power	3.01387W	3.12267W	+4%

6.4.2 Influence of the Sparsity

The performance of SMVM-NoC is not influenced by the sparsity of the matrix. It is more stable than the general processor (i.e. GPP's performance is highly depending on the sparsity). 500 random matrices with the matrix size of 128×128 and 256×256 are generated. They are with random sparsity from 10% to 50%. Figure 6.8 shows that the performance of the proposed SMVM–NoC is not depending on the sparsity, whereas the sparsity affects the performance of SMVM in a general purpose processor. However, the performance of pipelined SMVM-NoC has a huge variance compared to the non–pipelined one. On the other hand, according to Figure 6.7 currently only an average performance in the

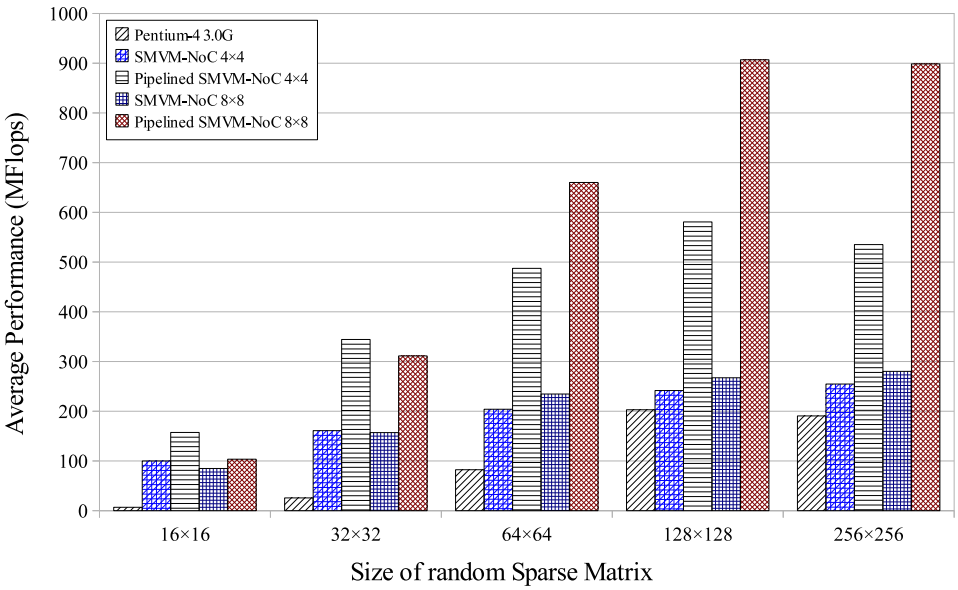
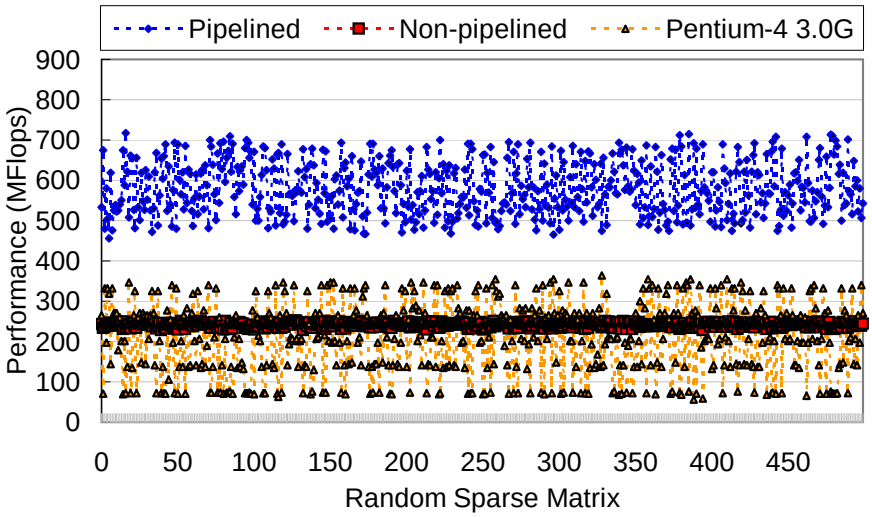
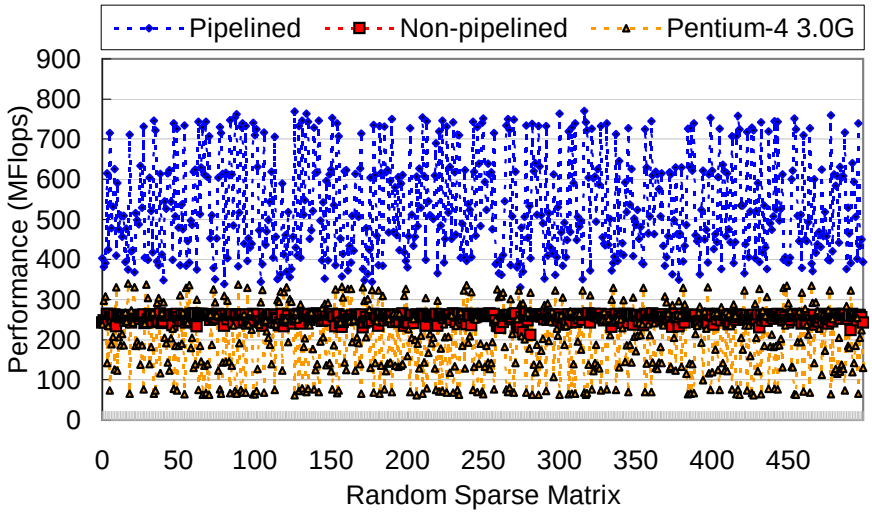


Figure 6.7: Performance analysis of different matrix size with random sparsity on the Pentium-4 PC, non–pipelined 4×4/8×8 SMVM-NoC, 4×4/8×8 pipelined SMVM-NoC (operating at 200MHz)

range of hundreds of MFlops could be obtained from the experimental results. If we further look into the packet distribution analysis between the PEs and the switches as shown in Figure 6.9, we can see that most of the time the packets are stuck in the switches due to the traffic congestion and poor throughput of the switches. This is because the switches transmitting packets over the network have much shorter latency compared to the PEs and then results in an unbalanced performance between the pipelined PEs and the pipelined switches. Therefore, the switch is compromised with the PE concerning clock rate, which results in performance decrements. To solve this problem, we can separate the clock source into two parts, one for PE running at a lower frequency, another for the switch running at a higher frequency. Figure 6.10 shows a simple scenario that the clock domain is divided into two parts by utilizing the asynchronous FIFO buffer. This asynchronous FIFO is a special customized design, which can receive and transmit the data between different clock domains. This is already highly adapted in todays analog and mixed–signal VLSI design. Therefore, using different clocks for NI and PE should enable to improve the performance.



(a) 500–sets random 128×128 sparse matrice

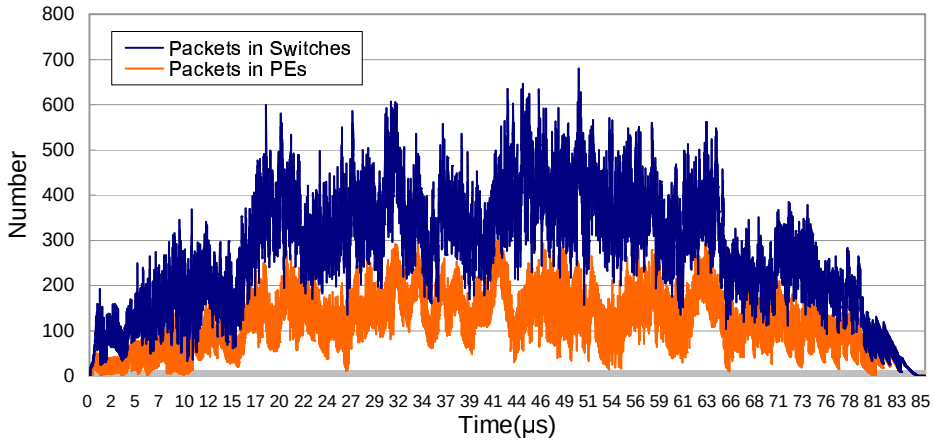


(b) 500–sets random 256×256 sparse matrice

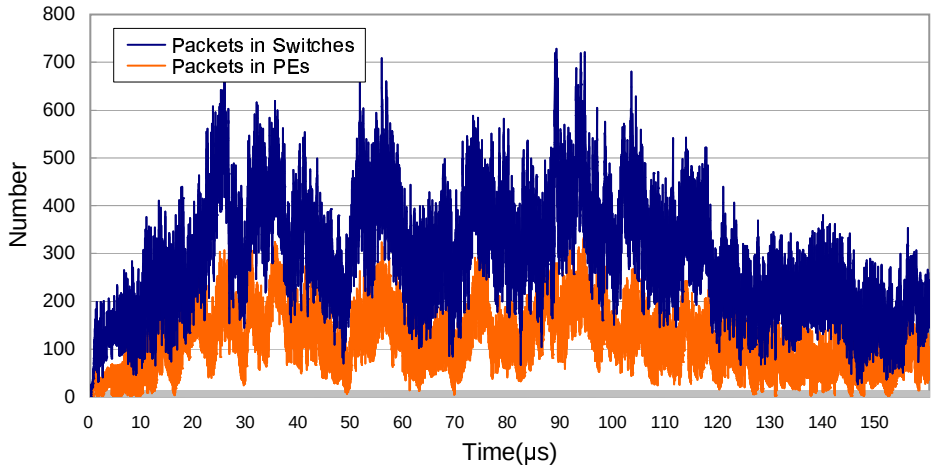
Figure 6.8: Influence of sparsity on different architectures with random sparsity from 10% to 50%

6.4.3 Mapping to Iterative Solver

When we have to map the proposed SMVM-NoC architecture into current iterative solvers, more sophisticated methods (steepest descent, CG) are required. So far, the current architecture can be used for certain methods for solving sparse systems of linear equations, such as the Jacobi and Gauss Seidel methods. The convergence of these methods, however, depends on certain properties of the matrix (e.g. spectral radius, diagonal dominance and so on). Furthermore, there is an important fact specific to the iterative methods. The routing paths are not



(a) 128×128 sparse matrix



(b) 256×256 sparse matrix

Figure 6.9: Analysis of the packet traffics for the 4×4 pipelined SMVM–NoC

known during the first iteration, especially when the sparsity structure is still unknown. However, the sparsity structure and the routing paths on the NoC will be determined once the first iteration is finished. These paths will be identical for the remaining iterations.

Although these methods are dominated by the SMVM concerning their complexity, there are other operations which must be executed between the successive SMVM computations. Incorporating these operations into the overall data flow must be investigated for the different methods.

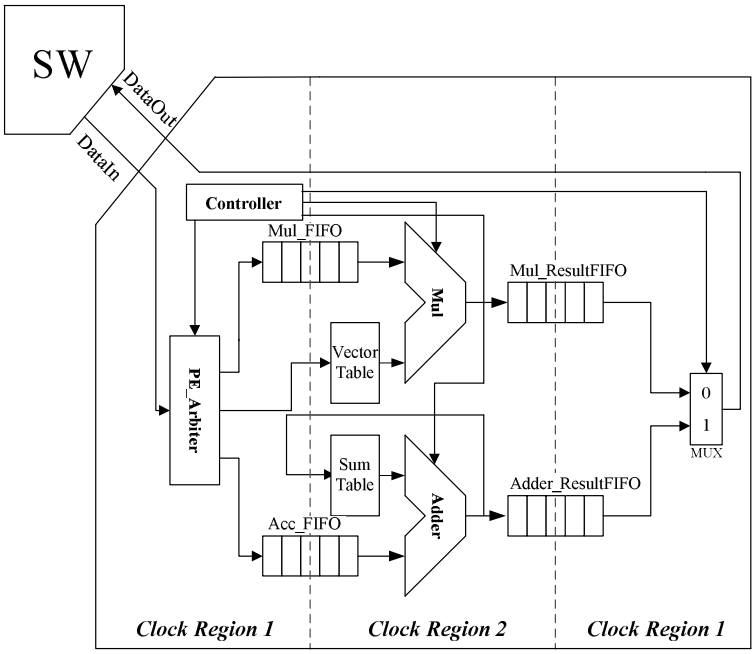


Figure 6.10: Two clock regions for the PE and the switch, one for PE running at higher frequency, another lower frequency

6.5 Summary

In this chapter, a new design concept for accelerating the SMVM based on NoC in an FPGA was presented. Matrix-vector multiplications with various random sparse matrices in IEEE–754 single floating point precision have been tested on the Xilinx Virtex–6 FPGA. The advantages of introducing the NoC structure into SMVM computation were given by high resource utilization, flexibility and the ability to communicate among heterogeneous systems, such that more accelerators can be configured into a larger $p \times p$ array ($p =, 2, 4, 8, \ldots, 2^k, k \in \mathbb{N}$). The synthesis results showed that the advanced FPGA with the chip-internal NoC network can provide a solution for sparse matrix computation to further accelerate many iterative solvers in hardware. Moreover, the NoC structure can receive data from and forward results to different entries simultaneously. This makes it possible to deal with very large sparse matrices with arbitrary sparsity structure of the matrix without interfering the performance by the sparsity of the matrix.

7 Conclusions

Lately, the ever increasing speed of production and standardization cycles for electronic devices drives the VLSI manufacturing technology. It is evolving into VDSM level with a very fast trend. This trend is expected to be continued for at least 10 more years. However, it also causes many nasty side effects like the famous design productivity gap, short development period for time–to–market, increasing of leakage power and the bus transmission bottleneck. In this regard, these challenges motivate to analyze their impact on iterative algorithms and to present generalized VLSI design concepts, which consider the significant impact on area, timing, power, performance, cost, reliability and so on.

Implementing an iterative algorithm on a multiprocessor array, there is a tradeoff between the complexity of an iteration step and the number of required iterations. As long as the convergence property is retained, it is possible to modify the algorithm (i.e. mainly the iteration step). However, it is not easy to find a balanced solution to satisfy all design criteria, especially for the performance/complexity of the hardware, the load/throughput of interconnects and the overall energy/power consumption. Therefore, in order to understand the relationship between the circuit design issues and the iteration steps, we have elaborated the design issues through four different selected iterative algorithms: CORDIC processor for signal processing, configurable QDCIT transformations for video compression, parallel Jacobi method for EVD and parallel SMVM–NoC for solving systems of linear equations. In all cases, it was shown how the iterative algorithm is derived from the system model, and how it can be implemented in a circuit efficiently.

A brief introduction to the fundamental CORDIC algorithm with different rotation modes and their corresponding VLSI circuit implementations was given first. Different flow graphs of the CORDIC archi-

tectures influence the throughput and the energy consumption deeply. It simply becomes a tradeoff problem between the hardware cost (logical utilization) and the throughput (byte per second). For example, the area of the folded CORDIC flow graph is smaller than the pipelined CORDIC but the throughput is much lower. On the contrary, the use of pipelining not only obtains a higher throughput, but also achieves low energy consumption. However, the logical utilization is higher.

Second, a low-complexity and fully configurable QDCIT with arbitrary iteration steps was presented. Since most of the video and image standards require the DCT and integer transform, these two transformations were integrated together by using various modules which are based on the CORDIC iterations and the compensation steps. The proposed 2-D FQDCIT architecture requires only 120 adders and 40 barrel shifters to perform the multiplierless 8×8 FQDCT and 4×4/8×8 forward quantized integer transform by sharing the hardware resources as much as possible. An inverse version with arbitrary iteration numbers has also been implemented. The proposed 2–D IQDCIT architecture requires only 124 adders and 40 barrel shifters to perform the inverse transformations. According to the ASIC implementation results, the QDCIT achieved a smaller chip area and higher throughput compared to the other architectures. Most important is that it provided a good compromise between PSNR and computational effort, and also satisfied the referenced implementations in terms of video quality (both XVID and H.264). On the other hand, the arbitrary iteration steps can further adjust the hardware complexity according to the device's resolution. That means the tradeoff between the video quality and the computational complexity is highly adaptable. Therefore, the presented iterative architecture for video compression is very suitable for low-complexity and multi-standard SoC designs.

Third, a Jacobi EVD method was selected as a typical example for modification of the iteration step, because of its very robust convergence property as long as the orthogonality of the similarity transformation is retained. A configurable 10×10 Jacobi EVD array has been implemented as an ASIC with the Full CORDIC and the μ–CORDIC to explain the circuit design concept for parallel iterative algorithms. An adaptive method was also applied to improve the convergence speed. A detailed comparison was made concerning the tradeoffs between the inner iteration numbers of CORDIC processor and the outer sweep num-

bers. It was shown that the modified Jacobi method can further improve the area overhead of combinational logic, reduce the computation time and save the energy consumption.

Fourth, a novel design to accelerate the SMVM based NoC concept was presented. In this architecture, the advantages of introducing the NoC structure into SMVM computation were high resource utilization, flexibility and the ability to communicate among heterogeneous systems. The performance of SMVM-NoC is not depending on the sparsity structure of the sparse matrix. Since the NoC structure can receive data from and forward results to different entries simultaneously, it is able to deal with very large sparse matrix without caring about the structure of the sparse matrix. The implementation results show that the chip-internal NoC network can achieve a good balanced solution for sparse matrix computation to further accelerate many numerically problems in hardware, such as solving a system of linear equation (CG Method), FEM problem and so on.

The comparison to the established iterative architectures showed significant differences of the system in many respects, such as the achieved PSNR, the need of logical utilization, the power/energy consumption, the computational complexity and others. As we have always emphasized throughout this thesis, a design engineer must think carefully which strategy should be selected for the hardware implementation of iterative algorithms. These four examples indicate that a proper decision can be made efficiently for different design issues.

For future work, FFT algorithm, wavelet transformation or 16×16 integer transform can be also integrated into our configurable DCIT core by using the CORDIC architecture. This will make the transform core be able to support next generation of scalable video Codecs, such as H.264 scalable profiles, High Efficiency Video Coding (HEVC/H.265) and JPEG 2000.

Furthermore, instead of selecting a fixed size parallel EVD array, we will try to design a block Jacobi method, where a smaller array, say 10×10 to 40×40, can solve the EVD/SVD problems with size of more than 1000×1000.

The pipelined SMVM-NoC design is facing a performance bottleneck

due to the unbalanced packet distribution between PEs and switches. This caused a huge performance gap between them. The floating-point units in the PEs can have a critical latency compared to the switch which only performs data transmission. For this reason, we will extend this architecture to speed up the clock rate of the switch by separating the clock signal into NI clock and PE clock. Furthermore, the pipeline stage of PE can also be adjusted in order to reduce the packet injection rate.

In addition, the general concept of exchanging inner and outer iterations in parallel iterative algorithms can be investigated for other iterative algorithms and their applications.

A Appendix Tables

Table A.1: The detailed information for each x86 based CPU from 1970 until 2010

Year	CPU chip	short name	vender	transistor counts	die size (mm)
1971-11-15	4004	4004	Intel	2,300	12
1972-4-1	8008	8008	Intel	2,500	15
1974-4-1	8080	8080	Intel	6,000	20
1978-6-8	8086	8086	Intel	29,000	33
1982-2-1	80268	80286	Intel	134,000	47
1985-10-17	80386	80386	Intel	275,000	104
1989-4-1	80486	80486	Intel	1,185,000	81
1993-3-22	Pentium P5	Pentium	Intel	3,100,000	293
1996-3-27	AMD K5	K5	AMD	4,300,000	251
1997-4-2	AMD K6	K6	AMD	8,800,000	157
1997-5-7	Pentium II	P2	Intel	7,500,000	113
1999-2-26	Pentium III	P3	Intel	9,500,000	128
1999-6-23	Athlon K7	K7	AMD	22,000,000	184
2000-11-20	Pentium 4	P4	Intel	42,000,000	217
2001-6-1	Itanium	Itanium	Intel	325,000,000	300
2003-9-23	Athlon 64	K8	AMD	105,000,000	193
2005-7-18	Itanium Processor 9M	Itanium II	Intel	592,000,000	432
2005-8-1	Athlon 64 X2 Dual-Core	K9	AMD	233,200,000	199
2006-8-1	Core 2 Duo Dual-Core	Core 2	Intel	291,000,000	143
2007-10-31	Itanium 9152M Dual-Core	Itanium II Dual	Intel	1,720,000,000	596
2007-11-19	Phenom X4 9600 Quad-Core	Phenom	AMD	463,000,000	283
2008-5-29	Nano U2350	VIA Nano	VIA	94,000,000	63
2008-11-17	Core i7 Quad-Core	i7	Intel	731,000,000	263
2009-8-13	Phenom II X4 965 Quad-Core	Phenom II	AMD	758,000,000	258
2010-1-7	Core i5 Dual-Core	i5	Intel	774,000,000	296
2010-2-8	Itanium 2 9350 Quad-Core	Itanium II Quad	Intel	2,046,000,000	698.75
2010-3-11	Core i7-980X Extreme Six-Core	i7 980X	Intel	1,170,000,000	248
2010-4-26	Phenom II X6 Six-Core	Phenom II X6	AMD	904,000,000	346

Table A.2: The pseudo code for each type of Jacobi parallel EVD code generation

top, when (p = 1) and (1 < q < N)

$$\begin{bmatrix} a_{11} \text{ in } \mathrm{PE}_{p,q} \leftrightarrow a_{12} \text{ in } \mathrm{PE}_{p,q-1}; & a_{12} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{11} \text{ in } \mathrm{PE}_{p,q+1} \\ a_{21} \text{ in } \mathrm{PE}_{p,q} \leftrightarrow a_{12} \text{ in } \mathrm{PE}_{p+1,q-1}; & a_{22} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{11} \text{ in } \mathrm{PE}_{p+1,q+1} \end{bmatrix}$$

bottom, when (p = N) and (1 < q <N)

$$\begin{bmatrix} a_{11} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{22} \text{ in } \mathrm{PE}_{p-1q-1}; & a_{12} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{21} \text{ in } \mathrm{PE}_{p-1q+1} \\ a_{21} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{22} \text{ in } \mathrm{PE}_{pq-1}; & a_{22} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{21} \text{ in } \mathrm{PE}_{pq+1} \end{bmatrix}$$

inner, when (1 < p < N) and (1 < q < N)

$$\begin{bmatrix} a_{11} \text{ in } \mathrm{PE}_{p,q} \leftrightarrow a_{22} \text{ in } \mathrm{PE}_{p-1,q-1}; & a_{12} \text{ in } \mathrm{PE}_{p,q} \leftrightarrow a_{21} \text{ in } \mathrm{PE}_{p-1,q+1} \\ a_{21} \text{ in } \mathrm{PE}_{p,q} \leftrightarrow a_{12} \text{ in } \mathrm{PE}_{p+1,q-1}; & a_{22} \text{ in } \mathrm{PE}_{p,q} \leftrightarrow a_{11} \text{ in } \mathrm{PE}_{p+1,q+1} \end{bmatrix}$$

left, when (1 < p < N) and (q = 1)

$$\begin{bmatrix} a_{11} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{21} \text{ in } \mathrm{PE}_{p-1q}; & a_{12} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{21} \text{ in } \mathrm{PE}_{p-1q+1} \\ a_{21} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{11} \text{ in } \mathrm{PE}_{p+1q}; & a_{22} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{11} \text{ in } \mathrm{PE}_{p+1q+1} \end{bmatrix}$$

right, when (1 < p < N) and (q = N)

$$\begin{bmatrix} a_{11} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{22} \text{ in } \mathrm{PE}_{p-1q-1}; & a_{12} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{22} \text{ in } \mathrm{PE}_{p-1q} \\ a_{21} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{12} \text{ in } \mathrm{PE}_{p+1q-1}; & a_{22} \text{ in } \mathrm{PE}_{pq} \leftrightarrow a_{12} \text{ in } \mathrm{PE}_{p+1q} \end{bmatrix}$$

four corner

when (p = 1 and q = 1)

$$\begin{bmatrix} a_{11} \text{ in } \mathrm{PE}_{11} \leftrightarrow a_{11} \text{ in } \mathrm{PE}_{11}; & a_{12} \text{ in } \mathrm{PE}_{11} \leftrightarrow a_{11} \text{ in } \mathrm{PE}_{12} \\ a_{21} \text{ in } \mathrm{PE}_{11} \leftrightarrow a_{11} \text{ in } \mathrm{PE}_{21}; & a_{22} \text{ in } \mathrm{PE}_{11} \leftrightarrow a_{11} \text{ in } \mathrm{PE}_{22} \end{bmatrix}$$

when (p = 1) and (q = N)

$$\begin{bmatrix} a_{11} \text{ in } \mathrm{PE}_{1N} \leftrightarrow a_{12} \text{ in } \mathrm{PE}_{1N-1}; & a_{12} \text{ in } \mathrm{PE}_{1N} \leftrightarrow a_{12} \text{ in } \mathrm{PE}_{1N} \\ a_{21} \text{ in } \mathrm{PE}_{1N} \leftrightarrow a_{12} \text{ in } \mathrm{PE}_{2N-1}; & a_{22} \text{ in } \mathrm{PE}_{1N} \leftrightarrow a_{12} \text{ in } \mathrm{PE}_{2N} \end{bmatrix}$$

when (p = N) and (q = 1)

$$\begin{bmatrix} a_{11} \text{ in } \mathrm{PE}_{N1} \leftrightarrow a_{21} \text{ in } \mathrm{PE}_{N-11}; & a_{12} \text{ in } \mathrm{PE}_{N1} \leftrightarrow a_{21} \text{ in } \mathrm{PE}_{N-12} \\ a_{21} \text{ in } \mathrm{PE}_{N1} \leftrightarrow a_{21} \text{ in } \mathrm{PE}_{N1}; & a_{22} \text{ in } \mathrm{PE}_{N1} \leftrightarrow a_{21} \text{ in } \mathrm{PE}_{N2} \end{bmatrix}$$

when (p = N) and (q = N)

$$\begin{bmatrix} a_{11} \text{ in } \mathrm{PE}_{NN} \leftrightarrow a_{22} \text{ in } \mathrm{PE}_{N-1N-1}; & a_{12} \text{ in } \mathrm{PE}_{NN} \leftrightarrow a_{22} \text{ in } \mathrm{PE}_{N-1N} \\ a_{21} \text{ in } \mathrm{PE}_{NN} \leftrightarrow a_{22} \text{ in } \mathrm{PE}_{NN-1}; & a_{22} \text{ in } \mathrm{PE}_{NN} \leftrightarrow a_{22} \text{ in } \mathrm{PE}_{NN} \end{bmatrix}$$

B Appendix Figures

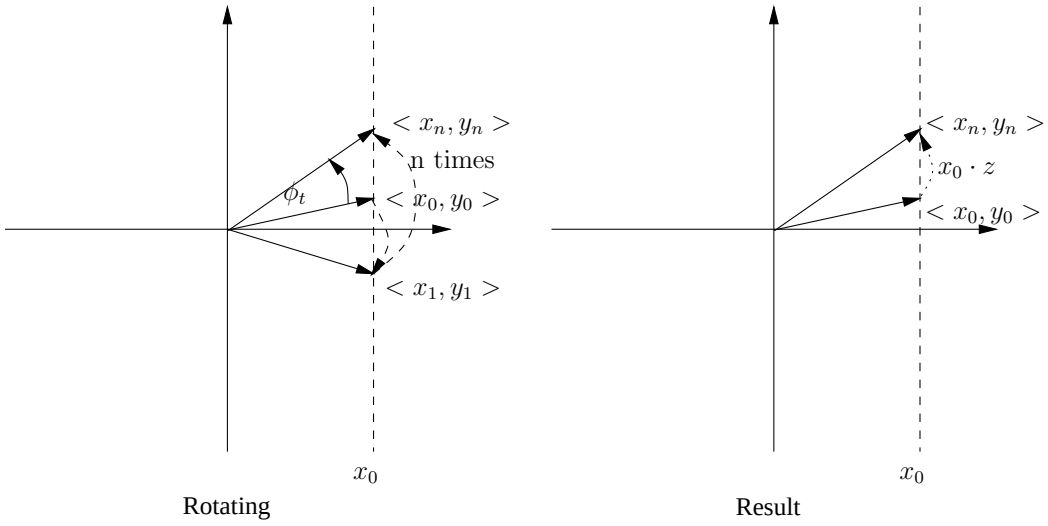


Figure B.1: CORDIC linear rotation mode

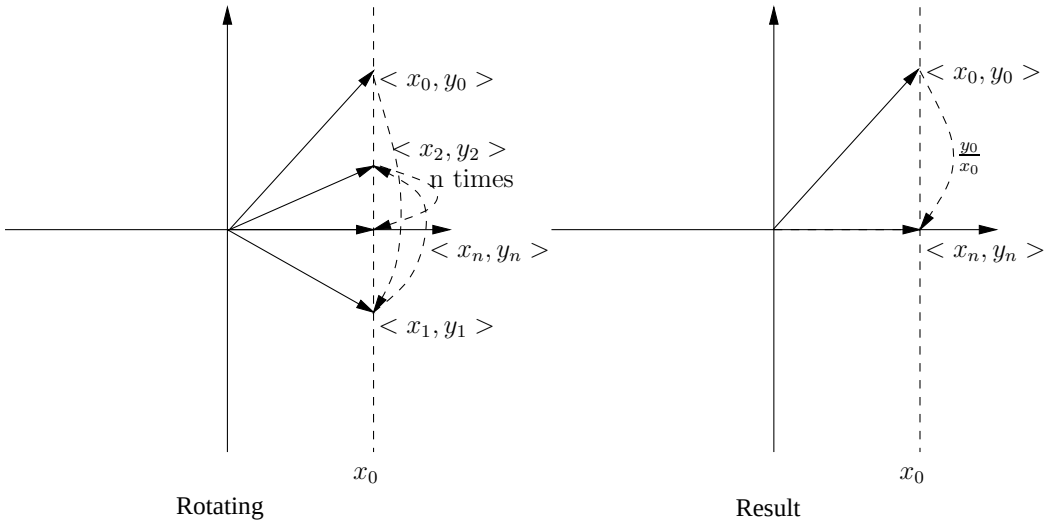


Figure B.2: CORDIC linear vector mode

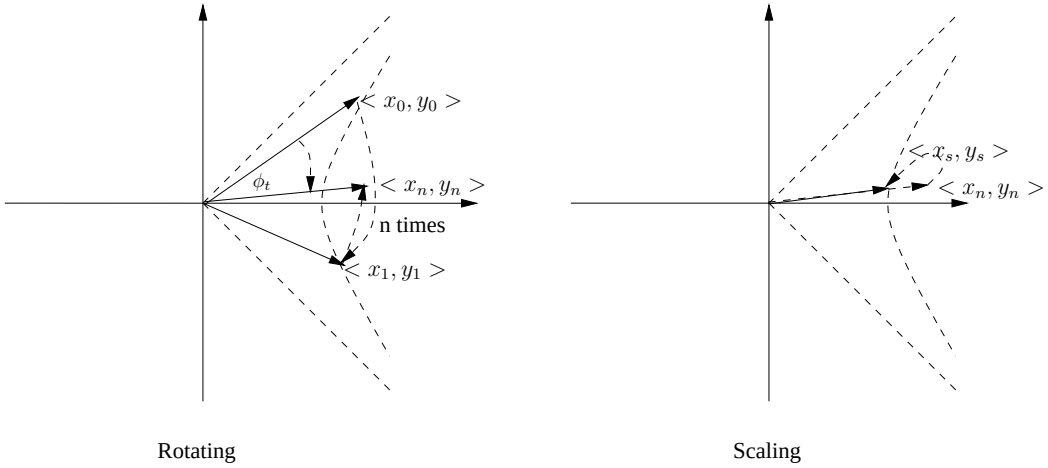


Figure B.3: CORDIC hyperbolic rotation mode

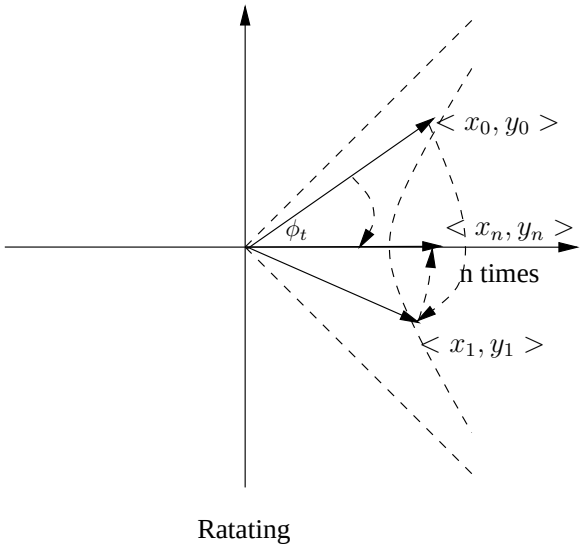


Figure B.4: CORDIC hyperbolic vector mode

(a) foreman CIF sequence

(b) paris CIF sequence

(c) news CIF sequence

(d) crew DVD sequence

(e) ice DVD sequence

(f) rush–hour Full–HD sequence

(g) blue–sky Full–HD sequence

Figure B.5: Seven video sequences for test the QDCIT transformation

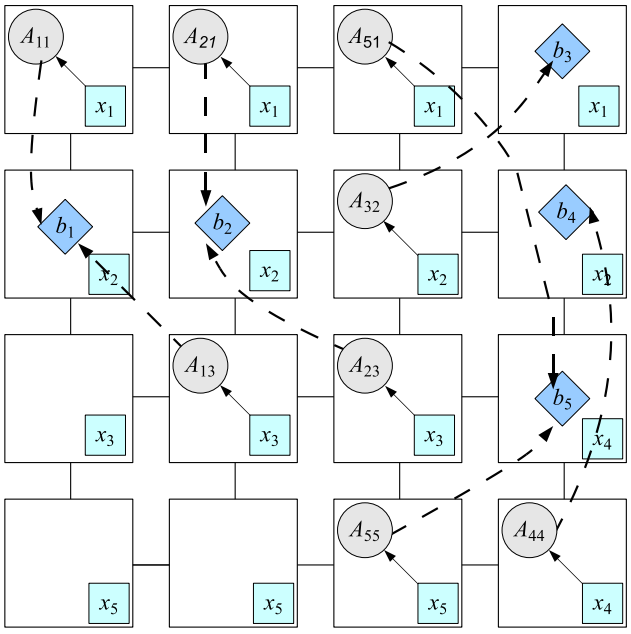


Figure B.6: An inverse mapping of parallel SMVM operations based on the NoC architecture

Notation and Abbreviations

Notation

x	Vectors
$\hat{m}$	offset errors
b	Matrix vector mutlication results
c	Constant, scalar system parameters
A	Matrices
$x(i)$	The ith element of vector x
$A(i,j)$	The element in row i, column j of matrix A
a_{ij}	The element in row i, column j of matrix A
A^T	Transpose of A
A^{-1}	Inverse of A
A^H	Conjugate transpose of A
$\text{diag}(A)$	Diagonal matrix with elements of A on diagonal
$x(t)$	Time-continous scalar function
$x(n)$	Time-discrete scalar function
$C = A \otimes B$	Element–by–element matrix multiplication

Abbreviations

1-D	One-Dimensional
2-D	Two-Dimensional
3D-IC	Three-Dimensional Integrated Circuit
μ-CORDIC	μ-rotation CORDIC
AC	Alternating Current
AIR	Algebraic Integer Representations
ALU	Algorithmic Logic Unit

ASIC	Application Specific Integrated Circuit
CG	Conjugate Gradient
CIF	Common Intermediate Format (352×288)
CLDCT	CORDIC based Loeffler DCT
CMOS	Complementary Metal-Oxide-Semiconductor
CNFET	Carbon Nanotube Field Effect Transistor
Codec	enCoding/decoding
CORDIC	COordinate Rotation for DIgital Computer
CPU	Central Processing Unit
CSD	Canonical Signed Digit
CSR	Compressed Sparse Row
DA	Distributed Arithmetic
DC	Directing Current
DCT	Discrete Cosine Transform
DCIT	Discrete Cosine Integer Transform
DFT	Discrete Fourier Transform
DSM	Deep Sub-Micron
DSP	Digital Signal Processor
DVD	Digital Video Disc
EVD	Eigenvalue Decomposition
FEM	Finite Element Method
FFT	Fast Fourier Transform
FGA	Flow Graph Arithmetic
FP	Floating Point
FPGA	Field Programmable Gate Array
FQDCIT	Forward Quantized Discrete Cosine Integer Transform
GPS	Global Positioning System
GSM	Global System for Mobile communication
H.263	ITU-T H.263 standard
H.264	ITU-T H.264 standard
H.265	ITU-T H.265 standard
HD	High-Definition
HDL	Hardware Description Language
HDTV	High-Definition Television
HEVC	High Efficiency Video Coding
IDCT	Inverse Discrete Cosine Transform
IDCIT	Inverse Discrete Cosine Integer Transform
IEEE 754	IEEE Standard for Floating-Point Arithmetic
IFFT	Inverse Fast Fourier Transform
IO	Input/Output

IP	Intellectual Property
IQDCIT	Inverse Quantized Discrete Cosine Integer Transform
ITU-T	Telecommunication Standardization Sector
ITRS	International Technology Roadmap for Semiconductors
JPEG	Joint Photographic Experts Group
LSB	Least Significant Bit
LUT	Lookup Table
MAC	Multiply Accumulate
MEMS	Micro Electro Mechanical Systems
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MPEG	Moving Picture Experts Group
MPEG-4 AVC	MPEG-4 Advanced Video Coding
MPSoC	MultiProcessor System-on-Chip
MSB	Most Significant Bit
MTCMOS	Multi-Threshold CMOS
MUXRC	Multiplexed Row-Column Decomposition
NI	Network Interface
NiMH	Nickel-Metal Hydride
PMOS	P-type Metal-Oxide-Semiconductor
NoC	Network-on-Chip
NQDCT	Novel QDCT
NRE	Non-Recurring Engineering
OFDM	Orthogonal Frequency Domain Multiplexing
OPB	Xilinx On–chip Peripheral Bus
PR	PageRank
P & R	Place and Route
PE	Processor/Processing Element
PMOS	P-type Metal-Oxide-Semiconductor
PSNR	Peak Signal-to-Noise Ratio
QCIF	Quarter Common Intermediate Format (176×144)
QDCT	Quantized Discrete Cosine Transform
QDCIT	Quantized Discrete Cosine Integer Transform
QP	Quantization Parameter
QRD	QR Decomposition
QStep	Quantization Step
RC	Row-Column Decomposition
RTL	Register Transfer Level
SiP	System-in-Package
SMVM	Sparse Matrix-Vector Multiplication
SMVM-NoC	Sparse Matrix-Vector Multiplication based on Network-on-Chip

SoC	System-on-Chip
SoP	System-of-Package
SPD	Symmetric Positive-Definite
SVD	Singular Value Decomposition
TRAM	Transpose RAM
TSMC	Taiwan Semiconductor Manufacturing Company limited
UHD	Ultra High-Definition
VDSM	Very Deep Sub-Micron
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration
x86	Intel 80x86 CPU family
WCDMA	Wideband Code Division Multiple Access
WiMAX	Worldwide Interoperability for Microwave Access
WLAN	Wireless Local Area Network

Bibliography

[1] 15 Moore's Years: 3D chip stacking will take Moore's Law past 2020. Technical report, IBM, Mar 2010.

[2] H.M. Ahmed, J.M. Delosme, and M. Morf. Highly Concurrent Computing Structure for Matrix Arithmetic and Signal Processing. *IEEE Computer Magazine*, 15:65–82, 1982.

[3] A. Ahmedsaid, A. Amira, and A. Bouridane. Improved SVD systolic array and implementation on FPGA. In *IEEE International Conference on Field-Programmable Technology*, pages 3–42, December 2003.

[4] M. Ali and J. Götze. A VLSI-Suited Algorithm for Solving Linearly Constrained Least Squares Problems. In *Algorithms and Parallel VLSI Architectures*, pages 87–96, 1991.

[5] Chandrakasan P. Anantha and Robert W. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic, 1995.

[6] Ray Andraka. A survey of CORDIC algorithms for FPGA based computers. In *International Symposium on Field Programmable Gate Arrays*, pages 191–200, 1998.

[7] I. Appelbaum. Silicon spintronics. In *International Conference on Ultimate Integration of Silicon*, pages 5–8, 2009.

[8] S. Asaad and T. Bapty. Performance modeling for adaptive parallel embedded systems. In *IEEE International Performance, Computing, and Communications Conference*, pages 57–64, April 2002.

[9] N.J. August and Dong Sam Ha. Low power design of DCT and IDCT for low bit rate video codecs. *IEEE Transactions on Multimedia*, 6(3):414–422, June 2004.

[10] Martin Becka, Gabriel Oksa, and Marian Vajtersic. Dynamic ordering for a parallel block-Jacobi SVD algorithm. *Parallel Computing*, 28(2):243–262, 2002.

[11] Martin Becka and Marian Vajtersic. Block-Jacobi SVD algorithms for Distributed Memory System I: Hypercubes and Rings. *Parallel Algorithms and Applications*, 13(3):265–287, 1999.

[12] Martin Becka and Marian Vajtersic. Block-Jacobi SVD algorithms for Distributed Memory System II:Meshes. *Parallel Algorithms and Applications*, 14(1):37–56, 1999.

[13] Luca Benini and Giovanni De Micheli. System-Level Power Optimization: Techniques and Tools. *ACM Transactions on Design Automation of Electronc Systems*, 5(2):115–192, April 2000.

[14] Luca Benini and Giovanni De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, January 2002.

[15] Luca Benini and Givanni De Micheli. *Dynamic Power Management: Design Techniques and CAD Tools.* Kluwer Academic, 1998.

[16] Tobias Bjerregaard and Shankar Mahadevan. A Survey of Research and Practices of Network-on-Chip. *ACM Computing Surveys*, 38(1):1–51, 2006.

[17] S. Borkar. Thousand Core Chips A Technology Perspective. In *Design Automation Conference*, pages 746–749, 2007.

[18] Richard P. Brent and Franklin T. Luk. The Solution of Singular-Value and Symmetric Eigenvalue Problems on Multiprocessor Arrays. *SIAM Journal on Scientific and Statistical Computing*, 6(1):69–84, January 1985.

[19] Cadence. The Cadence Website. www.cadence.com, Dec 2010.

[20] B.H. Calhoun, F.A. Honore, and A.P. Chandrakasan. A leakage reduction methodology for distributed MTCMOS. *IEEE Journal of Solid-State Circuits*, 39(5):818–826, May 2004.

[21] K. Chakrabarty, R.B. Fair, and Jun Zeng. Design Tools for Digital Microfluidic Biochips: Toward Functional Diversification and More Than Moore. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(7):1001–1017, July 2010.

[22] Hao-Chieh Chang, Jiun-Ying Jiu, Li-Lin Chen, and Liang-Gee Chen. A Low Power 8×8 Direct 2-D DCT Chip Design. *VLSI Signal Processing*, 26(3):319–332, November 2000.

[23] Wen Hsiung Chen, C. Smith, and S. Fralick. A Fast Computational Algorithm for the Discrete Cosine Transform. *IEEE Transactions on Communication*, 25(9):1004–1009, September 1977.

[24] Yuhua Chen. Cell Switched Network-on-Chip Candidate for Billion-Transistor System-on-Chips. In *IEEE International SOC Conference*, pages 57–60, September 2006.

[25] Chang Cho, Jun Heo, and Joon Kim. An extension of J.83 annex B transmission systems for ultra-high definition (UD) TV broadcasting. *IEEE Transactions on Consumer Electronics*, 55(1):63–68, February 2009.

[26] R. C. Conzalez and R. E. Woods. *Digial Image Processing*. Prentice-Hall, 2001.

[27] Philip P. Dang, Paul M. Chau, Truong Q. Nguyen, and Trac D. Tran. BinDCT and Its Efficient VLSI Architecture for Real-Time Embedded Applications. *Journal of Imaging Science and Technol*, 49(2):124–137, April 2005.

[28] B. Das and S. Banerjee. A low complexity architecture for complex discrete wavelet transform. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, 6-10 2003.

[29] Michael deLorimier and André DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, pages 75–85, February 2005.

[30] J.-M. Delosme. A Processor for Two-Dimensional Symmetric Eigenvalue and Singular Value Arrays. In *Asilomar Conference on Circuits, Systems and Computers*, pages 217–221, 1987.

[31] E. Deprettere, P. Dewilde, and R. Udo. Pipelined CORDIC Architecture for Fast VLSI Filtering and Array Processing. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 250–253, 1984.

[32] E. Deprettere and F. Ed. Synthesis and Fixed Point Implementation of Pipelined True Orthogonal Filters. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 217–220, 1983.

[33] Li-Fu Ding, Wei-Yin Chen, Pei-Kuei Tsung, Tzu-Der Chuang, Pai-Heng Hsiao, Yu-Han Chen, Hsu-Kuang Chiu, Shao-Yi Chien, and Liang-Gee Chen. A 212 MPixels/s 4096 2160p Multiview Video Encoder Chip for 3D/Quad Full HDTV Applications. *IEEE Journal of Solid-State Circuits*, 45(1):46–58, January 2010.

[34] A. Docef, F. Kossentini, Khanh Nguuyen-Phi, and I.R. Ismaeil. The quantized DCT and its application to DCT-based video coding. *IEEE Transactions on Image Processing*, 11(3):177–187, March 2002.

[35] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach.* Morgan Kaufmann, 2003.

[36] P. Duhamel and H. H'Mida. New 2^n DCT algorithms suitable for VLSI implementation. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 12, pages 1805–1808, April 1987.

[37] Y. Elkurdi, D. Fernández, E. Souleimanov, D. Giannacopoulos, and W. J. Gross. FPGA architecture and implementation of

sparse matrix-vector multiplication for the finite element method. *Computer Physics Communications*, 178:558–570, April 2008.

[38] M.D. Ercegovac and T. Lang. Redundant and On–Line CORDIC: Application to Matrix Triangularization and SVD. *IEEE Transactions on Computers*, 39:725–740, 1990.

[39] David Gregg et al. FPGA Based Sparse Matrix Vector Multiplication using Commodity DRAM Memory. In *International Conference on Field Programmable Logic and Applications*, pages 786–791, August 2007.

[40] M. Fu, G.A. Jullien, V.S. Dimitrov, and M. Ahmadi. A low-power DCT IP core based on 2D algebraic integer encoding. In *IEEE International Symposium on Circuits and Systems*, volume 2, pages 765–768, May 2004.

[41] Pat Gelsinger. Moore's Law: "We See No End in Sight,". Technical report, Intel Chief Technology Officer, May 2008.

[42] Juergen Goetze, Benjamin Heyne, Shanq-Jang Ruan, and Chi-Chia Sun. European Union Patent Application: EP1850597,Method and circuit for performing cordic based loeffler discrete cosine transformation (dct) for signal processing, 2007.

[43] Juergen Goetze, Benjamin Heyne, Shanq-Jang Ruan, and Chi-Chia Sun. USA Pending Patent: US20070250557, Method and circuit for performing cordic based loeffler discrete cosine transformation (dct) for signal processing, 2007.

[44] Juergen Goetze, Benjamin Heyne, Shanq-Jang Ruan, and Chi-Chia Sun. Taiwan Patent: TW200600143651, Method and circuit for performing cordic based loeffler discrete cosine transformation (dct) for signal processing, 2010.

[45] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, 1996.

[46] J. Götze and G.J. Hekstra. An Algorithm and Architecture Based

on Orthonormal Micro-Rotations for Computing the Symmetric EVD. *Integration, the VLSI Journal*, 20:21–39, 1995.

[47] J. Götze, S. Paul, and M. Sauer. An Efficient Jacobi-Like Algorithm for Parallel Eigenvalue Computation. *IEEE Transactions on Computers*, 42(9):1058–1065, September 1993.

[48] J. Gotze and U. Schwiegelshohn. Sparse matrix-vector multiplication on a systolic array. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 2061–2064 vol.4, April 1988.

[49] Jiun-In Guo, Rei-Chin Ju, and Jia-Wei Chen. An efficient 2-D DCT/IDCT core design using cyclic convolution and adder-based realization. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(4):416–428, April 2004.

[50] Gerben J. Hekstra and Ed F. Deprettere. Floating Point Cordic. In *IEEE Symposium on Computer Arithmetic*, pages 130–137, July 1993.

[51] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oeberg, M. Millberg, and D. Lindquist. Network on a Chip: An Architecture for Billion Transistor Era. In *Proceeding of the IEEE NorChip Conference*, pages 24–31, November 2000.

[52] Magnus Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952.

[53] Benjamin Heyne. *Efficient CORDIC Based Implementation of Selected Signal Processing Algorithms*. PhD thesis, Technology University of Dortmund, October 2008.

[54] Jingcao Hu and R. Marculescu. DyAD - smart routing for networks-on-chip. In *Design Automation Conference*, pages 260–263, June 2004.

[55] Y.H. Hu and H.M. Chern. VLSI CORDIC Array Structure Implementation of Toeplitz Eigensystem Solvers. In *IEEE Inter-*

national Conference on Acoustics, Speech, and Signal Processing, pages 1575–1578, April 1990.

[56] Yu Hen Hu. CORDIC-based VLSI architectures for digital signal processing. *IEEE Signal Processing Magazine*, 9(3):16 –35, July 1992.

[57] ITRS. 2003 Edition, Interconnect. Technical report, International Technology Roadmap for Semiconductors, 2003.

[58] ITRS. 2009 Edition, Executive Summary. Technical report, International Technology Roadmap for Semiconductors, 2009.

[59] ITRS. 2009 Edition, System Drivers. Technical report, International Technology Roadmap for Semiconductors, 2009.

[60] K. Jainandunsing and Deprettere E.F. A New Class of Parallel Algorithm for Solving Systems of Linear Equation. *SIAM Journal on Scientific Computing*, 10:880–912, 1989.

[61] Ali Javey, Jing Guo, Qian Wang, Mark Lundstrom, and Hongjie Dai. Ballistic Carbon Nanotube Field-Effect Transistors. *Letters to Nature*, 424:654–657, August 2003.

[62] Hyeonuk Jeong, Jinsang Kim, and Won Kyung Cho. Low-power multiplierless DCT architecture using image correlation. *IEEE Transactions on Consumer Electronics*, 50(1):262–267, February 2004.

[63] Xiangyang Ji, S. Kwong, D. Zhao, H. Wang, C.-C.J. Kuo, and Qionghai Dai. Early Determination of Zero-Quantized 8 8 DCT Coefficients. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(12):1755–1765, December 2009.

[64] Andrew B. Kahng. Scaling: More than Moore's law. *IEEE Design Test of Computers*, 27(3):86–87, May 2010.

[65] Nachiket Kapre and André DeHon. Optimistic Parallelization of Floating-Point Accumulation. In *IEEE Symposium on Computer Arithmetic*, pages 205–216, June 2007.

[66] G. Katti, M. Stucchi, K. De Mayer, and W. Dehaene. Electrical modeling & characterization of through silicon via (TSV) for 3D ICs. *IEEE Transactions Electron Devices*, 57(1):256–262, January 2010.

[67] Michael Keating, David Flynn, Rob Aitken, Alan Gibbons, and Kaijian Shi. *Low Pwoer Methodology Manual For System-on-Chip Design*. Springer, 2008.

[68] Jongman Kim, C. Nicopoulos, Dongkook Park, V. Narayanan, M.S. Yousif, and C.R. Das. A Gracefully Degrading and Energy-Efficient Modular Router Architecture for On-Chip Networks. In *International Symposium on Computer Architecture*, pages 4–15, 2006.

[69] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, December 2003.

[70] S. Klauke and J. Götze. Low Power Enhancements for Parallel Algorithms. In *IEEE International Symopsium on Circuits and Systems*, pages 234–237, May 2001.

[71] R.C. Kordasiewicz and S. Shirani. ASIC and FPGA implementations of H.264 DCT and quantization blocks. In *IEEE International Conference on Image Processing*, volume 3, pages III–1020–3, September 2005.

[72] Y. Liu, C.-S. Bouganis, and P.Y.K. Cheung. Hardware architectures for eigenvalue computation of real symmetric matrices. *IET Computers and Digital Techniques*, 3(1):72–84, January 2009.

[73] C. Loeffler, A. Lightenberg, and G. S. Moschytz. Practical fast 1-D DCT algorithms with 11-multiplications. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 988–991, May 1989.

[74] H.S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky. Low-Complexity Transform and Quantization in H.264/AVC.

IEEE Transactions of Circuits and Systems for Video Technology, 13(7):598–603, July 2003.

[75] E.P. Mariatos, D.E. Metafas, J.A. Hallas, and C.E. Goutis. A fast DCT processor, based on special purpose CORDIC rotators. In *IEEE International Symposium on Circuits and Systems*, volume 4, pages 271–274, May 1994.

[76] S. McGettrick, D. Geraghty, and C. McElroy. An FPGA architecture for the Pagerank eigenvector problem. In *International Conference on Field Programmable Logic and Applications*, pages 523–526, September 2008.

[77] Gerard Meurant. *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations (Software, Environments and Tools)*. Society for Industrial Mathematics, 2006.

[78] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. Electrical Engineering. McGraw-Hill, 1994.

[79] MAEDA MIKIO. Steps Toward the Practical Use of Super Hi-Vision. In *Proceedings of 2006 NAB BEC*, pages 450–455, January 2006.

[80] Gordon Moore. Cramming More Components Onto Integrated Circuits. *Electronics Magazine*, 38(8), 1965.

[81] Gerald R. Morris and Viktor K. Prasanna. Sparse Matrix Computations on Reconfigurable Hardware. *Computer*, 40(3):58–64, March 2007.

[82] Keshab K. Parhi. *VLSI Digital Signal Processing Systems - Design and Implementation*. John Wiley & Sons, 1999.

[83] Keshab K. Parhi and Takao Nishitani. *Digial Signal Processing for Multimedia Systems*. Marcel Dekker, 1999.

[84] Jongsun Park, Jung Hwan Choi, and Kaushik Roy. Dynamic bit-width adaptation in DCT: image quality versus computation en-

ergy trade-off. In *Design, Automation and Test in Europe*, pages 520–521, March 2006.

[85] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures: System on Chip Interconnect.* Morgan Kaufmann, 2008.

[86] Nishant Patil, Albert Lin, Jie Zhang, H.-S. Philip Wong, and Subhasish Mitra. Digital VLSI logic technology using Carbon Nanotube FETs: frequently asked questions. In *Design Automation Conference*, pages 304–309, 2009.

[87] Massoud Pedram. Power Minimization in IC Design: Principles and Applications. *ACM Transactions on Design Automation of Electronc Systems*, 1(1):3–56, January 1996.

[88] Massoud Pedram and Jan M Rabaey. *Power Aware Design Methodologies.* Springer, 2002.

[89] Massoud Pedram and Hirendu Vaishnav. Power Optimization in VLSI Layout: A Survey. *Journal of VLSI Signal Processing Systems*, 15(3), 1997.

[90] James R. Powell. The Quantum Limit to Moore's Law. *Proceedings of the IEEE*, 96(8), August 2008.

[91] Arifur Rahman and Rafael Reif. System-level performance evaluation of threedimensional integrated circuits. *IEEE Transactions on Very Large Scale Integration Systems*, 8(6):671–678, December 2000.

[92] A. Raychowdhury, Jeong II Kim, D. Peroulis, and K. Roy. Integrated MEMS Switches for Leakage Control of Battery Operated Systems. In *IEEE Custom Integrated Circuits Conference*, pages 457–460, 2006.

[93] Iain E. G. Richardson. *Video Codec Design.* John Wiley & Sons, 2002.

[94] Iain E. G. Richardson. *H.264 and MPEG-4 Video Compression.*

John Wiley & Sons, 2003.

[95] K. Roy, Byunghoo Jung, and A.R. Than. Integrated Systems in the More-than-Moore Era: Designing Low-Cost Energy-Efficient Systems Using Heterogeneous Components. In *International Conference on VLSI Design*, pages 464–469, 2010.

[96] Kaushik Roy and Sharat C. Prasad. *Low-Power CMOS VLSI Circuit Design*. John Wiley, 2000.

[97] G. A. Ruiz, J. A. Michell, and A. Burón. High Throughput Parallel-Pipeline 2-D DCT/IDCT Processor Chip. *Journal of VLSI Signal Processing*, 45(3):161–175, 2006.

[98] Youssef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial Mathematics, 2003.

[99] K. S. Sainarayanan, C. Raghunandan, and M.B. Srinivas. Delay and Power Minimization in VLSI Interconnects with Spatio-Temporal Bus-Encoding Scheme. In *IEEE Computer Society Annual Symposium on VLSI*, pages 401–408, March 2007.

[100] H. Schwarz and M. Wien. The Scalable Video Coding Extension of the H.264/AVC Standard [Standards in a Nutshell]. *IEEE Signal Processing Magazine*, 25(2):135–141, March 2008.

[101] IlHong Shin and Hyun Wook Park. Adaptive Up-Sampling Method Using DCT for Spatial Scalability of Scalable Video Coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(2):206–214, February 2009.

[102] L.H. Sibul and A.L. Fogelsanger. Application of Coordinate Rotation Algorithm to Singular Value Decomposition. In *IEEE International Symposium on Circuits and Systems*, pages 821–824, 1984.

[103] S. Simon, P. Rieder, C. Schimpfle, and J.A. Nossek. CORDIC-based architectures for the efficient implementation of discrete wavelet transforms. In *IEEE International Symposium on Circuits and Systems*, volume 4, pages 77–80, 12-15 1996.

[104] B. Stackhouse, S. Bhimji, C. Bostak, D. Bradley, B. Cherkauer, J. Desai, E. Francom, M. Gowan, P. Gronowski, D. Krueger, C. Morganti, and S. Troyer. A 65 nm 2-Billion Transistor Quad-Core Itanium Processor. *IEEE Journal of Solid-State Circuits*, 44(1):18–31, January 2009.

[105] Karsten Suehring. The JM 16.1 H.264/AVC. iphome.hhi.de/suehring/tml/, December 2010.

[106] Chi-Chia Sun, Philipp Donner, and Jürgen Götze. Low-complexity multi-purpose IP Core for quantized Discrete Cosine and integer transform. In *IEEE International Symposium on Circuits and Systems*, pages 3014–3017, May 2009.

[107] Chi-Chia Sun, Philipp Donner, and Jürgen Götze. VLSI Implementation of a Configurable IP Core for Quantized Discrete Cosine and Integer Transforms. *International Journal of Circuit Theory and Applications*, 2011.

[108] Chi-Chia Sun and Jürgen Götze. A VLSI Design Concept for Parallel Iterative Algorithms. *Advances in Radio Science*, 7:95–100, 2009.

[109] Chi-Chia Sun and Jürgen Götze. VLSI Circuit Design Concepts for Parallel Iterative Algorithms in Nanoscale. In *International Symposium on Communications and Information Technologies*, pages 688–692, September 2009.

[110] Chi-Chia Sun, Jürgen Götze, Hong-Yuan Jheng, and Shanq-Jang Ruan. Sparse Matrix-Vector Multiplication Based on Network-On-Chip. *Advances in Radio Science*, 8:289–294, 2010.

[111] Chi-Chia Sun, Hong-Yuan Jheng, Jurgen Gotze, and Shanq-Jang Ruan. sparse Matrix-Vector Multiplication Based on Network-on-Chip in FPGA. In *International Symposium on Advanced Topics on Embedded Systems and Applications*, pages 2306–2310, jul 2010.

[112] Chi-Chia Sun, Sanq-Jang Ruan, Benjamin Heyne, and Juergen Goetze. Low-power and high-quality Cordic-based Loeffler DCT

for signal processing. *IET Circuit Devices and System*, 1(6):453–461, December 2007.

[113] Chi-Chia Sun, Ce Zhang, and Jürgen Götze. A Configurable IP Core for Inverse Quantized Discrete Cosine and Integer Transform with Arbitrary Accuracy. In *IEEE International Conference on Asia Pacific Circuits and Systems*, pages 915–918, December 2010.

[114] Junqing Sun, Gregory Peterson, and Olaf Storaasli. Sparse Matrix-Vector Multiplication Design on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 349–352, April 2007.

[115] T.Y. Sung and Y.H. Hu. Parallel VLSI Implementation of the Kalman Filter. *IEEE Transactions on Aerospace and Electronic Systems*, 23(2):215–224, 1987.

[116] Tze-Yun Sung, Yaw-Shih Shieh, Chun-Wang Yu, and Hsi-Chin Hsin. High-Efficiency and Low-Power Architectures for 2-D DCT and IDCT Based on CORDIC Rotation. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 191–196, December 2006.

[117] Synopsys. The Synopsys Website. www.synopsys.com, December 2010.

[118] T. D. Tran. The binDCT: fast multiplierless approximation of the DCT. *IEEE Signal Processing Letters*, 7:141–144, June 2000.

[119] R.R. Tummala. Moore's law meets its match (system-on-package). *IEEE Spectrum*, 43(6):44–49, June 2006.

[120] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *IEEE International Conference Solid-State Circuits*, pages 98–589, February 2007.

[121] Francesco Vitullo, Nicola E. L'Insalata, Esa Petri, Sergio

Saponara, Luca Fanucci, Michele Casula, Riccardo Locatelli, and Marcello Coppola. Low-Complexity Link Microarchitecture for Mesochronous Communication in Networks-on-Chip. *IEEE Transactions on Computer*, 57(9):1196–1201, September 2008.

[122] Jack E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8:330–334, 1959.

[123] K.A. Wahid, V.S. Dimitrov, and G.A. Jullien. On the Error-Free Realization of a Scaled DCT Algorithm and Its VLSI Implementation. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 54(8):700–704, August 2007.

[124] J.S. Walther. A unified algorithm for elementary functions. In *Spring Joint Computer Conference*, volume 38, pages 379–385, May 1971.

[125] Alice Wang, Benton H. Calhoun, and Anantha P. Chandrakasan. *Sub-threshold design for ultra low-power systems*. Springer, 2006.

[126] Hanli Wang, Ming-Yan Chan, S. Kwong, and Chi-Wah Kok. Novel quantized DCT for video encoder optimization. *IEEE Signal Processing Letters*, 13(4):205–208, April 2006.

[127] Hanli Wang and Sam Kwong. Hybrid Model to Detect Zero Quantized DCT Coefficients in H.264. *IEEE Transactions on Multimedia*, 9(4):728–735, June 2007.

[128] James Hardy Wilkinson. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, 1995.

[129] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *ACM/IEEE conference on Supercomputing*, pages 1–12, November 2007.

[130] T.W. Williams. The Future Is Low Power and Test. In *European Test Symposium*, page 4, May 2008.

[131] Wayne Wolf. *FPGA-Based System Design*. PRENTICE HALL, 2004.

[132] Wayne Wolf. The future of multiprocessor systems-on-chips. In *Design Automation Conference*, pages 681–685, June 2004.

[133] Wayne Wolf. The future of multiprocessor systems-on-chips. In *Design Automation Conference*, pages 681–685, July 2004.

[134] Roger Woods, John Mcallister, Richard Turner, Ying Yi, and Gaye Lightbody. *FPGA Based Implmentation of Signal Processing Systems*. Wiley, 2008.

[135] T. Xanthopoulos and A.P. Chandrakasan. A low-power IDCT macrocell for MPEG-2 MPML exploiting data distribution properties for minimal activity. *IEEE Journal of Solid-State Circuits*, 34(5):693–703, May 1999.

[136] Xilinx. The Xilinx Website. www.xilinx.com, December 2010.

[137] XVID. The XVID Website. www.xvid.org, December 2010.

[138] Gary Yeap. *Practical Low Power Digital VLSI Design*. Kluwer Academic, 1998.

[139] Sungwook Yu and Jr. Swartzlander, E.E. A scaled DCT architecture with the CORDIC algorithm. *IEEE Transactions on Signal Processing*, 50(1):160–167, January 2002.

[140] Maojun Zhang, Tao Zhou, and Wei Wang. Adaptive Method for Early Detecting Zero Quantized DCT Coefficients in H.264/AVC Video Encoding. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(1):103–107, January 2009.

[141] Ling Zhuo and Viktor K. Prasanna. Sparse Matrix-Vector multiplication on FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, pages 63–74, 2005.

Personal Information

Name:	Chi-Chia Sun
Born:	7_{th} December 1981
Born in:	Taipei / Taiwan

Curriculum Vitae

2000.09–2004.06	Bachelor of Science in Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan.
2003.04–2004.09	IC Design Engineer, Trumpion Microelectronics Inc., Taipei, Taiwan.
2004.09–2006.06	Master of Science in Electronic Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan.
2006.05–2006.07	Visiting Graduate Student, Hong Kong University of Science and Technology, Hong Kong.
2006.07–2007.08	Communication Sergeant, Department of Defense, Kaohsiung, Taiwan.
2008.04–2011.03	Research Assistant, Information Processing Lab, Dortmund University of Technology, Germany.