POWER PERFORMANCE ANALYSIS OF EMBEDDED BLOCK CODING FOR OPTIMIZED TRUNCATION OF JPEG2000

By

JIGAR K SHAH

07MCE019

INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY

AHMEDABAD-382481

MAY 2009

POWER PERFORMANCE ANALYSIS OF EMBEDDED BLOCK CODING FOR OPTIMIZED TRUNCATION OF JPEG2000

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

By

JIGAR K SHAH

07MCE019

INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY

AHMEDABAD-382481

May 2009

Certificate

This is to certify that the Major Project entitled **"POWER PERFORMANCE ANALYSIS OF EMBEDDED BLOCK CODING FOR OPTIMIZED TRUNCATION OF JPEG2000"** submitted by **JIGAR K SHAH (07MCE019)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. (Dr.) S.N. Pradhan
Guide and Professor,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr K.Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

DATE: / /

Abstract

The increasing popularity of power constrained mobile computers and embedded computing applications drives the need for analyzing and optimizing power in all the components of a system. Recent years have witnessed a rapid growth in research activity targeted at reducing energy consumption in microprocessor based systems. However, this research has by and large not recognized the potential energy savings achievable through optimization of software running on the microprocessor. Some research work suggested the energy efficient techniques at basic gate and architecture level, but not at the instruction level. Some software optimization techniques and compiler techniques are also suggested for that.

Software constitutes a major component of today's systems, and its role is projected to grow even further. Thus, an ever increasing portion of the functionality of today's systems is in the form of instructions, as opposed to gates. This motivates the need for analyzing power consumption from the point of view of instructions something that traditional circuit and gate level power analysis tools are inadequate for.

This study describes an alternative, measurement based software level power analysis approach that provides an accurate and practical way of quantifying the power cost of software for the ARM processor based architecture. For that,try to analyze power consumption of the JPEG2000 codec.For that Simplescalar Toolset (simplesim/ARM) which gives architecture performance analysis and Sim-Panalyzer which gives power performance analysis are used.The main source of power and memory access of JPEG2000 is EBCOT (Embedded Block Coding for Optimized Truncation) part. In this study analysis of the EBCOT algorithm and its relation in terms of power consumption are presented.

Acknowledgements

The successful completion of a project is generally not an individual effort. It is an outcome of the cumulative efforts of a number of persons, each having own importance to the objective. This session is a vote of thanks and gratitude towards all those persons who have directly or indirectly contributed in their own specials way towards the completion of this project.

It gives me great pleasure in expressing thanks and profound gratitude to **Dr. S. N. Pradhan,** PG Coordinator, Department of Computer Science & Engineering, Institute of Technology, Nirma University of Science & Technology, Ahmedabad for his valuable guidance and continual encouragement throughout the project. I heartily thankful him for his time to time suggestions and the clarity of the concepts of the topic that helped me a lot during the project and without his guidance, this project would have been an uphill task.

I would like to thank **Dr. K. Kotecha,** Director, Institute of Technology, Nirma University of Science & Technology, Ahmedabad for providing me the facilities in the Nirma campus.

I am thankful to all the faculty members of Computer Science and Engg department, Nirma University of Science & Technology, Ahmedabad for providing me needed suggestions with crucial feedback that influenced me to complete this work.

The blessings of God, my Guruji and my parents make the way for completion of major project. I am very much grateful to them.

Last but not the least, I am equally thankful to all my classmates for their support and everything.

- JIGAR K SHAH

07MCE019

Contents

List of Tables

List of Figures

Abbreviation

EBCOT	Embedded Block Coding for Optimized Truncation
DWT	Discrete Wavelet Transform
DCT	Discrete Cosine Transform
SoC	System on Chip
AIO	Address Input Output
DIO	Data Input Output
ITLB	Instruction Table Look-aside Buffer
DTLB	Data Table Look-aside Buffer
IL1 Cache	Address Input Output
DL1 Cache	Address Input Output
AIO	Address Input Output
IRF	Instruction Register File
PSNR	Peak Signal to Noise Ratio
MSE	Mean Swuare Error
ICT	Irreversible Component Transform
RCT	Reversible Component Transform
EZW	Embedded Zero-Tree Wavelet Compression
SPIHT	Spatial Partitioning of Image into Hierarchical Trees
MSB	most significant bit-plane
LSB	least significant bit-plane
RLC	Run Length Coding
SC	Sign Coding
ZC	Zero Coding
MR	Magnitude Refinement
CF	Context Formation
AE	Arithmetic Encoder

Chapter 1

Introduction

1.1 General

Until recently, power dissipation was an issue that primarily concerned designers of embedded portable computer systems. However, power issues are becoming some of the primary design constraints for even very high-end microprocessors. Power consumption analysis is the basis of high-level power reduction techniques because they do not rely on actual physical design. High-level power reduction of microprocessor-based systems saves power consumption by changing energy-sensitive factors such as instruction fetch addresses, opcode encoding, register encoding, data fetch addresses, immediate operands, etc. Some of the energy-sensitive factors have great degrees of freedom while others are more restrictive. Under certain circumstances, even data and instructions can be changed as far as the original semantic is preserved. Consequently, it is important to be informed of power consumption variations with respect to the energy-sensitive factors for setting up proper power reduction strategies.

Power analysis can be performed by "simulation-based" or "measurement- based" approaches. [1] Simulation-based power analysis is convenient as far as a simulation model is available because it does not necessitate a prototype. Simulation is preferable to avoid system dependent bias as power consumption is also variable to bus config-

uration and peripheral devices. Measurement-based power consumption analysis is sometimes more feasible due to the availability of existing models even if a prototype is necessary. Even with a prototype, correct measurements are not easily obtainable because digital systems consume power in a spiky manner with over hundreds MHz in the power spectrum.

Instruction-level power studies of ARM ISA processors have been performed earlier. However, none of these accurately characterize or model external memory accesses or stalls. While this is acceptable for small benchmarks, a real OS and application can spend a significant number of cycles in such states. We accurately model both instructions and events such as stalls and memory accesses, in order to create a power model that is sufficiently accurate for modeling a full-featured OS and complete applications running over billions of clock cycles.

The intricacy involved by these new electronic appliances imposed a new design paradigm to cope with the specific requirements, e.g., low cost with fast time to market, and restrictions they have. Also, energy consumption is a critical factor in system-level design of embedded portable appliances. A hardware-software co-design framework must be employed to proceed with the design from the software applications intended to run on these appliances to the final specifications of the hardware that implements the desired functionality given the above-mentioned constraints. Studies have demonstrated that circuit- and gate-level techniques have less than a 2x impact on power, while architecture- and algorithm-level strategies offer savings of 10x - 100x or more. Hence, the greatest benefits are derived by trying to assess early in the design process the merits of the potential implementation. Architecture optimization corresponds to searching for the best design that optimizes all objectives. Since the optimization problem involves multiple criteria (power consumption, throughput, and cost) to reach the global optimum a set of desicive points in the design space have to be found. Ideally, when designing an embedded system, a designer would like to explore a number of architectural alternatives and test functionality, energy consumption, and performance without the need to build a prototype.

1.2 Motivation

The increasing popularity of power constrained mobile computers and embedded computing applications drives the need for analyzing and optimizing power in all the components of a system. This has forced an examination of the power consumption characteristics of all modules - ranging from disk-drives and displays to the individual chips and interconnects. Focusing solely on the hardware components of a design tends to ignore the impact of the software on the overall power consumption of the system. Software constitutes a major component of systems where power is a constraint. Its presence is very visible in a mobile computer, in the form of the system software and application programs running on the main CPU. But software also plays an even greater role in general digital application, since an ever growing fraction of these applications are now being implemented as embedded systems. Embedded systems are characterized by the fact that their functionality is divided between hardware and a software component. The software component usually consists of application-specific software running on a dedicated processor, while the hardware component usually consists of application-specific circuits. In light of the above, there is a clear need for considering the power consumption in systems from the point of view of software. Software impacts the system power consumption at various levels of the design. At the highest level, this is determined by the way functionality is partitioned between hardware and software. The choice of the algorithm and other higher level decisions about the design of the software component can affect system power consumption in a big way. The design of system software, the actual application source code, and the process of translation into machine instructions - all of these determine the power cost of the software component. In order to systematically analyze and quantify this cost, however, it is important to start at the most fundamental level. This is at the level of the individual instructions executing on the processor. Just as logic gates are the fundamental units of computation in digital hardware circuits, instructions can be thought of as the fundamental unit of software. This motivates the need for analyzing

power consumption from the point of view of instructions. Accurate modeling and analysis at this level is the essential capability needed to quantify the power costs of higher abstractions of software, and to search the design space in software power optimizations.

Instruction level analysis of a processor helps in the development of models for power consumption of software executing on that processor. The ability to evaluate software in terms of power consumption makes it feasible to search for low power implementations of given programs. In addition, it can guide the development of general tools and techniques for low power software. Several ideas in this regard as motivated by the power analysis of the subject processors are also described.[2, 3]

As advance in computer technology continues, use of internet, 3G, 4G techniques also increases and so Battery-powered devices such as mobile cellular phones, personal multimedia player, and personal digital assistants have recently become prevalent platforms to run image and video compression applications. Image and Video compression applications require high computational complexity, which means battery powered devices need high operating clock rate and large battery. And so these battery supported devices need techniques that reduces power requirement of the devices.

And last but not least, as use of environment aware system, use of "GREEN" system and its use in technology increases. This study will help to advance the technology in terms of power study.

1.3 Scope of Work

Software power analysis can help in reducing power consumption of battery supported system and also there are other several additional applications of this analysis and it is instructive to list the important ones here:

- The information provided by the analysis is useful in assigning an accurate power cost to the software component of a system. For power constrained

embedded systems, this can help in verifying if the overall system meets its specified power budget.

- The most common way of specifying power consumption in processors is through a single number - the average power consumption. Instruction level analysis provides additional resolution about power consumption that cannot be captured through just this one number. This additional resolution can guide the careful development of special programs that can be used as power benchmarks for more meaningful comparisons between processors.

- The measurement based instruction level analysis methodology has the novel strength that it does not require knowledge of the lower level details of the processor. However, if micro-architectural details of the CPU are available, they can be related to the results of the analysis. This can lead to more refined models for software power consumption, as well as power models for the micro-architecture that may potentially be more accurate than circuit or logic simulation based models.

- The additional insight provided by an instruction-level power model also provides directions for modifications in processor design that lead to the most effective overall power reduction. Instructions can be evaluated both in terms of their power cost as well as frequency of occurrence in typical compiler or even hand-generated code. This combined information can be used to prioritize instructions that should be re-implemented to be less expensive in terms of power.

This paper describes that with power analysis of Embedded Block Coding for Optimized Truncation (EBCOT) ,we can decide power budget for embedded system of EBCOT. And also Enable us to Design its specific SoC based VLSI chips that can be efficiently work in Digital cameras.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, **Literature Survey**, In it different power aware techniques at all level is described. And software power consumption and estimation research is described.

Chapter 3, **JPEG2000 Architecture & Design**, presents the JPEG2000 architecture and its basics.

chapter 4, **EBCOT Design & Analysis**,presents part of JPEG2000, EBCOT architecture and its algorithm.

Chapter 5, **Simulation Environment**, describes the different simulator and cross compiler for power analysis.

Chapter 6, **Proposed Design**, In it C programming power aware techniques and EBCOT algorithm modification techniques for power aware are described.

Chapter 7, **Results**, In it experienced results are shown.

chapter 8, **Conclusion and Future Scope** concluding remarks and scope for future work is presented.

Chapter 2

Literature Survey

2.1 General

Power is dissipated in various forms:

a. Interconnect Power

- Resistances and Parasitic Capacitances
- Transitions because of change in data results in charging and discharging

b. Memory power

- Depends on type of memory
- Depends on the number of access, the size of the memory (code and data), the number of ports used.

c. CPU power

- Instruction fetch, decode and execution results in power consumption
- Depends on current withdrawn by each instruction

Power is the basic constraint for the embedded environment and computer architecture.Power optimization can be done by mainly two approaches:

• Hardware Power Management

• Software Power Management

A lot research has been done in both Hardware and Software approaches. Power optimization can be done at different levels as shown in Table I.

Levels	**Techniques for power reduction**
System	Design partitioning, Power Down
Algorithm	Complexity, Concurrency, Locality, Regularity, Data representation
Architecture	Voltage scaling, Parallelism, Instruction-set, Signal correlations
Circuit/logic	Transistor Sizing, Logic optimization, Activity Driven Power Down,
Technology	Threshold Reduction, Multi thresholds

Table I: Power Reduction Techniques

2.2 Power Optimization Techniques

a. At System Level

- Design partitioning
- Power Down

  Detect and shut down unused units.

  To reduce the power in synchronous designs, it is important to minimize switching activity by powering down execution units when they are not performing "useful" operations. This is an Important concern since logic modules can be switching and consuming power even when they are not being actively utilized.

- Scheduling:
  Hardware allocation and task assignment.
- HW/SW co-design

b. At Algorithm Level

• The choice of an algorithm is the most highly leveraged decision in meeting the power constraints. The power consumption is strongly correlated to of a number of properties that a given algorithm may have.

• Some algorithmic properties which are critical for selection an algorithm for low power design:

(1) Size measures includes quantities such as the number of nodes, the bit width, the number of I/O operations, the number of operations, and the number of memory accesses.

(2) Concurrency measures the number of operations and interconnect accesses that can be executed concurrently.

(3) Temporality captures information about the lifetimes of variables in the computation. A computation is considered to be temporally local if the expected lifetimes of the variables are short. It is temporally dense if the measured maximum expected number of variables alive at any time is large.

(4) Spatial locality characterizes the degree to which the algorithm has natural clusters of computation, within which significant amounts of computation can be done independently.

(5) Regularity captures the degree to which common patterns appear.

c. At Architecture Level

• The architectural level is the design entry point for the large majority of digital designs and design decisions at this level can have dramatic impact on the power budget design.

• Perhaps the most important strategy for reducing power consumption involves employing concurrent processing at the architecture level. This is a direct trade-off of area and performance for power.

• Power Reduction Techniques.

  - Low system clocks
    High frequencies are generated with on-chip PLLs.
  - High-level of integration (single chip)
    Avoid off-chip components
  - Power management: shutdown unused blocks
  - Memory partitioning
    Selectively enabled blocks
  - Parallelism, pipelining
  - Reduction of global busses
  - Simplification of instruction coding and execution
  - Component minimization
    Arithmetic
    Memories/registers
  - Scheduling and allocation
  - Data/number coding

• Voltage reduction

  - To use this technique a designer should follow the next rules:
    Repeat the data path hardware n times.
    Use clock rate of 1/nT.
    Use a multiplexer to produce the output stream.
    The input and output stream should have the same throughput rate.

- Depending on the data path architecture, a typical estimation of the voltage reduction of 40% can be achieved. Although the use of parallelism leads to large power reduction, the penalty is increased area occupation.

- Parallel processing and pipelining
  - Parallel processing can be an important technique for reducing power consumption in CMOS systems.
  - Pipelining does not share this advantage because it achieves Concurrency by increasing the lock frequency, which limits the ability to scale the voltage.
  - This is an interesting reversal because pipelining is simpler that Parallel processing;
  - Therefore, pipelining is traditionally the first choice to speed up Execution. In practice, the trade-off between pipelining and parallelism is not so distinct:
  - Replicating function units rather than pipelining them has the negative effect of increasing area and wiring, which in turn can increase power consumption.
  - The degree to which designers can parallelize computations varies widely. Although some computations are "embarrassingly parallel," they are usually characterized by identical operations on array data structures.

- Buses
  - Buses are a significant source of power loss, especially interchip buses, which are often very wide.
  - A chip can expend 15 percent to 20
  - One approach to limiting this swing is to encode the address lines into a Gray code because address changes, particularly from cache refills, are

often sequential, and counting in Gray code switch the least number of signals.

- Transmitting the difference between successive address values achieves a result similar to the Gray code.
- Compressing the information in address lines further reduces them.
- These techniques are best suited to interchip signaling because designers can integrate the encoding into the bus controllers. At Circuit/Logic levelCode compression results in significant instruction memory savings if the system stores the program in compressed form and decompresses it on the fly, typically on a cache miss. [4]

d. At Circuit/Logic level

- Clock gating

  Widely used to turn off clock tree branches to latches or flip-flops whenever they are not used. Until recently, developers considered gated clocks to be a poor design practice because the clock tree gates can exacerbate clock skew. More accurate timing analyzers and more flexible design tools have made it possible to produce reliable designs with gated clocks

- Half-frequency and half-switching clocks

  A half-frequency clock uses both edges of the clock to synchronize events half the frequency of a conventional clock. Drawbacks: the latches are more complex and occupy more area, and that the clock's requirements are more stringent. [4]

  The half-swing clock swings only half of V dd. It increased the latch design's requirements and is difficult to use in systems with low V dd.

However, lowering clock swing usually produces greater gains than clocking on both edges.

- Choices between static versus dynamic topologies, conventional CMOS versus pass-transistor logic styles and synchronous versus asynchronous timing styles have to be made during the design of a circuit
- In static CMOS circuits, the component of power due to short circuit current is about the 10% of the total power consumption. However, in dynamic circuits does not appear this problem, since there is not any direct dc path from supply voltage to ground. Only in domino-logic circuits there is such a path, in order to reduce sharing, hence there is a small amount of short-circuit power dissipation. [5]
- Asynchronous logic. Because the systems do not have a clock, they save the considerable power that a clock tree requires.
- Optimizing switching activity (example from Rabaey: DIC).

e. At Transistor level

- Physical Capacitance: Three sources of capacitance: gate capacitance, diffusion capacitance, and interconnects capacitance. If all three components can be scaled down by the same factor, then the net power dissipation will be scaled down as well.
- Transistor Sizing: To minimize the physical capacitance, all transistors should be minimum size. Exceptions:

2.3 Software Power Consumption

The overall power dissipation of an embedded system does not only originate from the application - specific hardware, but also from the CPU, the memory and the address

and data buses when an embedded application is running on the platforms microprocessor, microcontroller or digital signal processor. This above power dissipation is referred to as software power dissipation. Obviously, the power is actually dissipated in the processors hardware, but it is as a consequence of executing an application program. There are a number of sources of power dissipation influenced by software and contributing to the overall power dissipation of the system, which are explained below. [6]

Bus Power

Busses in an embedded system consist of unidirectional address bus lines and instruction bus lines (opcodes to be executed) and bi-directional data bus lines. With these groups of interconnecting lines, the communication of the CPU with the memory, I/O circuits and peripheral modules is established. Each of the above lines can be modeled as an RC transmission line, where R and C are the lines resistance and capacitance, respectively. Activation of a line prompts for the charging or discharging of the capacitive load, depending on the previous value that this line had. For example, a transition in an 8-bit data bus between words 00101011 and 11100111 implies charging of 3 lines and discharging of 1 line. Usually, bus charging and discharging of I/O lines can occur up to 80% of the software execution time.

Memory Power

Power dissipated by memory read and write accesses is usually one of the dominating components (ranging from 10% - 25%) of the total software power dissipation for mobile devices and portable computers. In the case of DSP applications, where a significant amount of data is processed, this contribution can be substantially higher. Memory power dissipation has a number of components, namely power dissipated in the cell array, in the decode logic and sense amplifier as well as power dissipated due to charging and discharging of the address or data lines capacitances. The type of access is also significant in how much it contributes to the overall power dissipation. In the

ARM microprocessor, which is considered in our system, CPU cycles are divided in S-cycles, N-cycles, and I-cycles.

The S-cycles refer to sequential memory accesses, where the next word is returned from the same buffer, dissipating a relatively small amount of power. Power dissipation in the address lines during sequential memory accesses is also small, due to the fact that the address word changes only in one bit. If the last memory location of a page is to be accessed though, more power has to be consumed in order to access the next page.

The N-cycles refer to non-sequential memory accesses. In this kind of access, more power is dissipated relative to sequential accesses because consecutive address words are irrelevant, causing large activity in the address bus. In N-cycles usually different pages have to be accessed, contributing to more power dissipation, as explained above.

Finally, the I-cycles refer to cycles that no memory access is involved, so no power is dissipated in the memory system. One more significant contributor to the power dissipated in the memory is the memory access patterns, affecting mainly the cache hits and misses. The cache, residing closer to the CPU than main memory, dissipates less power because the address and data lines are shorter and have less internal capacitance. Inappropriate memory access patterns lead to cache misses, consequently, power expensive main memory accesses.

CPU Power

When instructions are executed in the CPU, they contribute significant power to the overall power dissipation. The instructions can be divided into four broad categories:

- Load / Store instructions
- Branch instructions
- Type-1 Arithmetic instructions, such as addition, subtraction, shift etc.
- Type-2 Arithmetic instructions, such as multiplication and division.

If we assume that the average power consumption to execute one instruction is W_j , where j represents one of the categories mentioned above, and I_j is the number of times this instruction is executed, we can easily derive the CPU power dissipation as:

$$P_{CPU}\alpha\frac{\sum(W_j * I_j)}{\sum I_j} \tag{2.1}$$

The power dissipated when an arithmetic instruction is executed depends primarily on the ALU or FPU data path that it instructed by software. Many different ways of optimization exist in this context, as for example replacing a division by power of two with corresponding right shifts. Finally, instruction scheduling is very important, because unsuccessful scheduling can lead to pipeline stalls, which in turn consume a considerable amount of power.

Other Power Dissipation Sources

There is a number of additional sources of power dissipation during software execution that must be taken into account, as they contribute as an overhead to the overall power dissipation. These sources are the clock distribution and the control logic and they accompany code execution in each cycle. In it was shown that short code sequences in a number of microcontrollers and DSPs always dissipated a smaller amount of power than longer sequences. The program in longer code sequences, which demanded extra execution cycles, took more time to execute so that the overhead was more than that in shorter code sequences. Despite this problem, power management techniques nowadays tend to eliminate such overhead by cleverly dealing with power dissipation that does not have a direct contribution to the involved computational tasks.

2.4 Software Power Estimation

The methodology proposed by Vivek Tiwari et al[1, 2, 3]. has been adopted as starting point for analysis of software power estimation. In spite of importance of estimation

at this level of abstraction, previous work exists for analyzing power consumption from the point of view of software. In [1, 2] the authors show that the choice of the algorithm and other high level decisions about the design of the software component can affect system power consumption in a big way. They propose an instruction level power analysis technique based on a physical measurement that helps in formulating instruction level power models that provide fundamental information needed to evaluate the power cost of an entire program. The basic components of each power model are the same. The first component is a set of base costs for the instructions set. The other components is the power cost of inter-instruction effects which involve more than one instruction. The base cost of an instruction can be thought of as the cost associated with the basic processing needed to execute the instruction. Experiments to determine this cost requires a program containing a loop consisting of several instances of the given instruction. The average current drawn during the execution of this loop is measured by an ameter inserted in series with the power supply and the CPU. The simple characterization with a base cost does not take into account the power consumption due to the change of circuit state when a sequence of two different instruction is executed. The difference between the measured current and the average base costs of the two instructions is defined as the circuit state overhead for the pair. For a sequence consisting of a mix of instructions, using the base costs of instructions almost always underestimates the actual cost. Adding in the average the circuit state overhead for each pair of consecutive instructions leads to a much closer estimate. The basic idea to determine this power contribution is to write programs where these effects occur repeatedly.

Definition of Power and Energy

Average Power:

$$P_{avg} = I_{avg} * V_{cc} \tag{2.2}$$

Energy :

$$E = P_{avg} * T \tag{2.3}$$

$$T = N * t \tag{2.4}$$

$$E = I_{avg} * V_{cc} * t \tag{2.5}$$

P_{avg} : Average power

I_{avg} : Average current

V_{cc} : Supply voltage

E : Energy consumption

T : Time taken

N : Number of cycles

t : Cycle time

MOV DX, [BX] Power = 1.15 W

MOV AX, CX Energy = $8.6 * 10 - 8$ J

MOV AX, DX

$$Programenergycost = \sum_{i}(Base_i * N_i) + \sum_{i,j}(Ovhd_{i,j} * N_i) + \sum_{k} Energy_k \tag{2.6}$$

N_i : Number of times instruction i is executed

$Base_i$: Base energy cost of i

$Ovhd_{i,j}$: Circuit state overhead when i, j are adjacent

$Energy_k$: Energy overhead of stalls, cache misses

Program power cost = Energy cost / execution time

Chapter 3

JPEG2000 Architecture & Design

3.1 Image Compression Basics

Uncompressed multimedia (graphics, audio and video) data requires considerable storage capacity and transmission bandwidth. Despite rapid progress in mass-storage density, processor speeds, and digital communication system performance, demand for data storage capacity and data-transmission bandwidth continues to outstrip the capabilities of available technologies. The recent growth of data intensive multimedia-based web applications have not only sustained the need for more efficient ways to encode signals and images but have made compression of such signals central to storage and communication technology.

For still image compression, the 'Joint Photographic Experts Group' or JPEG standard has been established by ISO (International Standards Organization) and IEC (International Electro-Technical Commission). The performance of these coders generally degrades at low bit-rates mainly because of the underlying block-based Discrete Cosine Transform (DCT) scheme. More recently, the wavelet transform has emerged as a cutting edge technology, within the field of image compression. Wavelet-based coding provides substantial improvements in picture quality at higher compression ratios.

Over the past few years, a variety of powerful and sophisticated wavelet-based schemes for image compression, as discussed later, have been developed and implemented. Because of the many advantages, the top contenders in the upcoming JPEG-2000 standard are all wavelet-based compression algorithms.

A typical lossy image compression system is shown in Fig. 3.1. It consists of three closely connected components namely (a) Source Encoder (b) Quantizer, and (c) Entropy Encoder. Compression is accomplished by applying a linear transform to decorrelate the image data, quantizing the resulting transform coefficients, and entropy coding the quantized values.

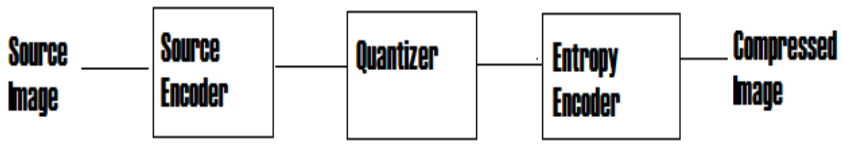

Figure 3.1: A Typical Lossy Signal/Image Encoder

The goal of this article is two-fold. First, for readers new to compression, we briefly review some basic concepts on image compression and present a short overview of the DCT-based JPEG standard and the more popular wavelet-based image coding schemes. Second, for more advanced readers, we mention a few sophisticated, modern, and popular wavelet-based techniques including one we are currently pursuing. The goal of the upcoming JPEG-2000 image compression standard, which is going to be wavelet-based, is briefly presented. For those who are curious, a number of useful references are given. There is also abundance of information about image compression on the Internet. Background

3.2 JPEG2000

JPEG 2000 is an initiative that will provide an image coding system using compression techniques based on the use of wavelet technology.

Wavelet theory [7]is also a form of mathematical transformation, similar to the FT in that it takes a signal in time domain, and represents it in frequency domain. Wavelet functions are distinguished from other transformations in that they not only dissect signals into their component frequencies, they also vary the scale at which the component frequencies are analyzed. Therefore wavelets, as component pieces used to analyze a signal, are limited in space. In other words, they have definite stopping points along the axis of a graph–they do not repeat to infinity like a sine or cosine wave does. As a result, working with wavelets produces functions and operators that are "sparse" (small), which makes wavelets excellently suited for applications such as data compression and noise reduction in signals. The ability to vary the scale of the function as it addresses different frequencies also makes wavelets better suited to signals with spikes or discontinuities than traditional transformations such as the FT. JPEG 2000 is a new standard for image coding published by the JPEG committee (ISO/IEC JTC 1/SC 29/WG 1) at the turn of the millennium. It is still being enhanced. The core, defined in Part 1 of the standard (ISO/IEC 15444 1), is built around the relatively new technologies of wavelet-based compression and bit-plane coding. It enables scalability with respect to five different defined image progression orders. The idea is similar to progressive JPEG, but the packetised structure of a JPEG 2000 codestream makes it possible to convert between different progression orders by the systematic reordering of packets, without any low-level decoding.

JPEG 2000 refers to all parts of the standard: [8]
The parts are:
◦ Part 1, Core coding system (intended as royalty and license-fee free - NB NOT patent-free)
◦ Part 2, Extensions (adds more features and sophistication to the core)

◦ Part 3, Motion JPEG 2000

◦ Part 4, Conformance

◦ Part 5, Reference software (Java and C implementations are available)

◦ Part 6, Compound image file format (document imaging, for pre-press and fax-like applications, etc.)

◦ Part 7 has been abandoned

◦ Part 8, JPSEC (security aspects)

◦ Part 9, JPIP (interactive protocols and API)

◦ Part 10, JP3D (volumetric imaging)

◦ Part 11, JPWL (wireless applications)

◦ Part 12, ISO Base Media File Format (common with MPEG-4)

As part of part5 there are C and Java implementation of JPEG2000 are available. And they are the acceptable standard of JPEG2000 and widely used.

• C implementation of JPEG2000 [9]

• Java implementation of JPEG2000 [10]

3.3 JPEG2000 overview

The JPEG2000 is composed of several main procedures such as DC level shifting, component transform, discrete wavelet transform, quantization and EBCOT (Context modeling, arithmetic coder and rate-distortion), as shown in Figure 3.2

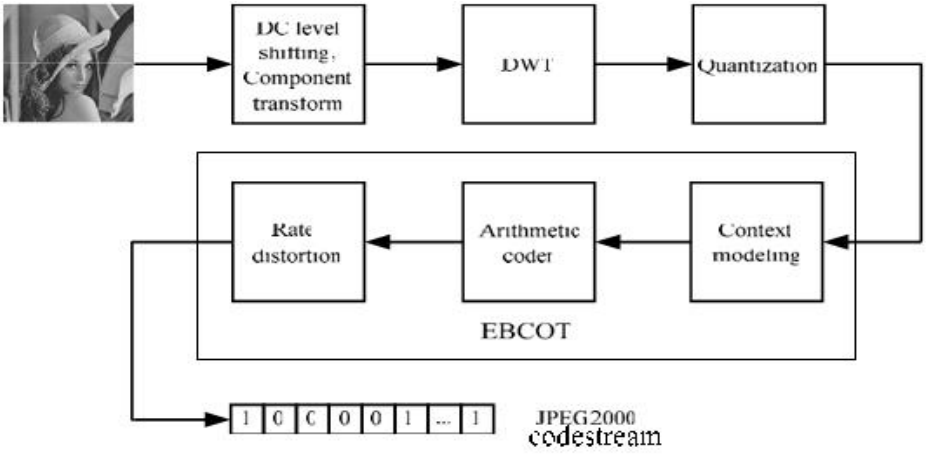

Figure 3.2: The JPEG2000 encoding block diagram [11]

Each block shown in Figure 3.2 is depicted below.

- DC Shift & Component Transform

  An image is usually composed of many components, for example, the R, G, B for color image. Each component is encoded by the DC shift coding so that the samples are changed from unsigned numbers to signed numbers. If the input image consists the color component, the reversible component transform (RCT) and the irreversible component transform (ICT) are enabled to improve the compression efficiency. [12]

- Discrete Wavelet Transform

  DWT transfers the image information from spatial domain to frequency domain thus the spatial correlation can be removed by DWT. There are two modes for DWT in JPEG2000. The integer (5, 3) filter applied in DWT is used to enable the lossless compression. Another one is for the default mode, DWT with the Daubechies (9, 7) filter, for the lossy compression. DWT is to separate high frequency part and low frequency part from vertical and horizontal directions

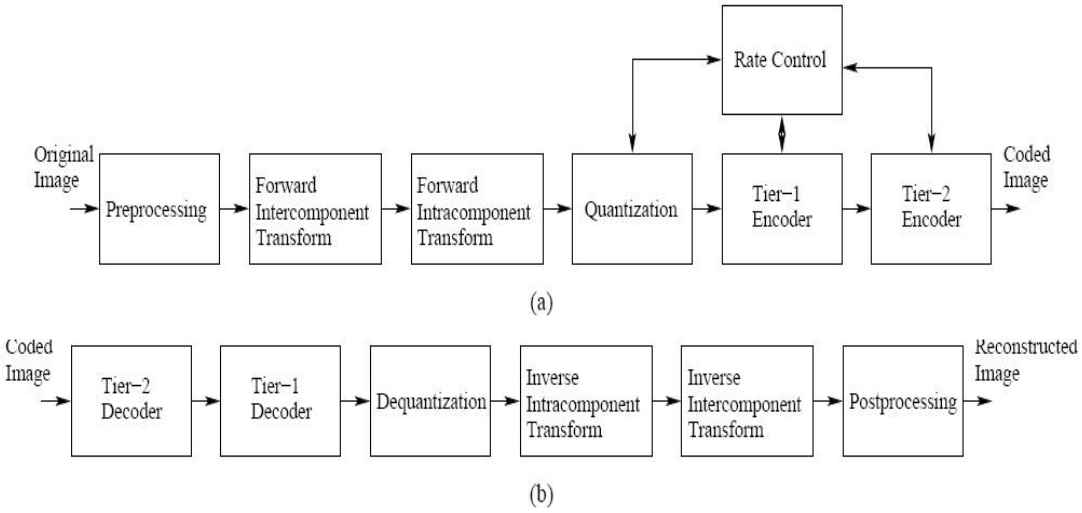

Figure 3.3: Codec structure. The structure of the (a) encoder and (b) decode

of an image respectively. [12]

- Scalar Quantization

  After DWT operated, the image coefficients is quantized to achieve the target bit-rate or distortion. Notice that the quantization operation causes the distortion for reconstruction image hence the quantization operation is adopted in lossy compression only.The quantization is used to adjust compression ratio by reducing the accuracy resolution. By the way, the compression ratio in JPEG2000 also can be controlled by EBCOT so that the quantization operation can be skipped. [12]

- Embedded Block Coding with Optimized Truncation

  EBCOT [13] is the entropy coder for JPEG2000 and detailed discussed in the following chapter. Comparing with the other wavelet-based entropy coding algorithms, such as the well know EZW and SPIHT , EBCOT provides better compression performance. The EBCOT coder divides each subband coefficient into code blocks.[13] Each code block is coded separately into a block based

embedded bit-stream, i.e. it doesn't need to consult other code blocks. The transformed coefficients of a code block are coded bit-plane by bit-plane from most significant bit-plane (MSB) to least significant bit-plane (LSB) instead of coefficient by coefficient with three passes called Significance Propagation Pass (Pass1), Magnitude Refinement Pass (Pass2), and Clean-Up Pass (Pass3). The arithmetic coder outputs are the sub-bit-streams of each compressed code block data. The rate-distortion block is designed to achieve progressive transmission and rate-distortion control.

EBCOT is the main base algorithm and followed in JPEG2000 as a block coder. From Table I it is sure that the main bottleneck of JPEG2000 is EBCOT, mainly tire-1. So here in this thesis performance analysis of EBCOT is described.

operation	**Single Component(sec)**	**RGB Component(sec)**
Intercomponent transform		14.12
Intracomponent transform	26.38	23.97
Quantization	6.42	5.04
EBCOT Tier 1	52.26	43.85
pass 1	14.82	12.39
pass 2	7	5.63
pass 3	16.09	13.77
arithmatic encoder	14.35	12.06
EBCOT Tier 2	14.95	13.01
layer formation	9.52	7.95
marker insertion	5.43	5.06

Table I: Run time profile for JPEG2000(Image 1792 x 1200,5 wavelet decomposition,profile at PIII-733 128M RAM,visual C++ and windows ME)*
[14]

Chapter 4

EBCOT Design & Analysis

4.1 Introduction

JPEG2000 entropy coder is EBCOT (Embedded Block Coding with Optimal Truncation Points) contextual coder. It is a bit-plane block coder i.e. it codes the wavelet coefficients by blocks.It codes the wavelet coefficients by blocks.On each bit-plane, there are three coding passes: a pass of Significance Propagation, a pass of Magnitude Refinement and a Cleanup pass. Four coding primitives are used: the RL(Run-Length) primitive, the ZC (Zero Coding) primitive, the MR (Magnitude Refinement) and the SC (Sign Coding) primitive. Here is an example how those coding passes are jointly used with the coding primitives.

4.2 Embedded Block Coding with Optimized Truncation(EBCOT)

The entropy coding for JPEG2000 encoding system is a bitplane-based coding, embedded block coding with optimized truncation (EBCOT). The EBCOT algorithm consists of two major steps: EBCOT block coding and rate-distortion optimization, i.e. the tier1 and tier2 in Figure 4.1, and the block coding also include two part,

context modeling and arithmetic coding. The EBCOT coder divides each subband coefficient into code blocks. Each code block is coded separately into a block-based embedded bitstream, i.e. it doesnt need to refer other code blocks. The tier2 coder organizes the bitstream to form a full-featured JPEG2000 bitstream.

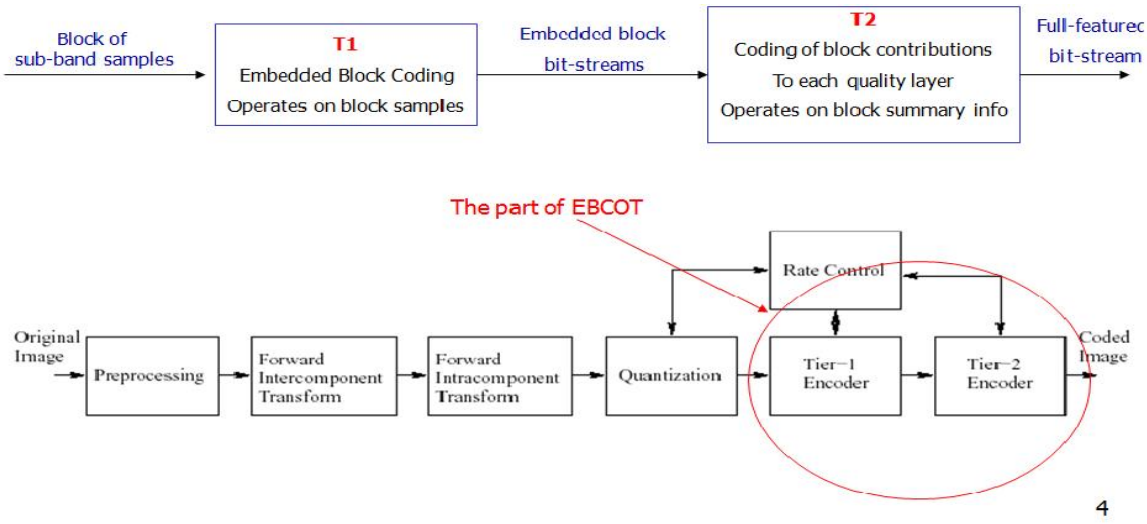

Figure 4.1: Two tiered coding in EBCOT

The context modeling, the first stage of EBCOT tier-1 coding, partitions the quantizer indices for each subband into code-blocks. It scans these quantizer indices in code-block bitplane by bitplane then generates a context-decision pair Figure 4.2to arithmetic coder (MQ coder).

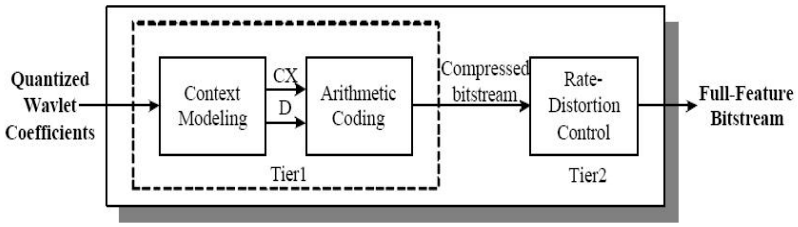

Figure 4.2: CX-D pair in EBCOT

4.3 EBCOT Algorithm and Analysis

Tier- 1 of EBCOT utilizes context-based arithmetic coding method to encode each code block into independent embedded bit-stream. Tier-1 coder can be viewed as two parts: Context Formation (CF) and Arithmetic Encoder (AE).[14] CF scans all pixels in code block in a specific order, and generates corresponding contexts for each bit. AE encodes the code block data according to their. contexts. EBCOT encodes the quantized wavelet coeficients bitplane by bitplane from MSB to LSB. Eveiy 4 rows in a bitplane are called a stripe, and each pass in every bitplane scans in order stripe by stripe. Then in every stripe, data are scanned column by coluinn. Every column is composed of 4 bits. So the scanning hierarchy of a code block is bitplane, pass, stripe, column, bit, as shown in Fig. 4.3.

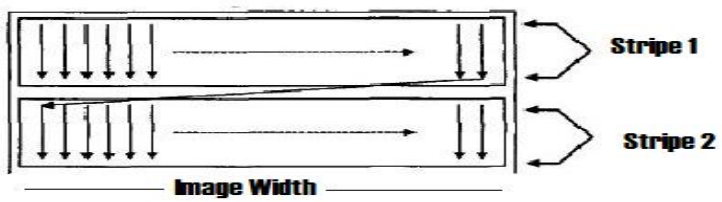

Figure 4.3: Scanning order of context formation in every pass

Contexts for all bits are generated according to their neighbors using four coding methods. Before CF, the quantized wavelet coefficients are separated into sign and magnitude (in 1s compliment). A pixel is called significant after the first 1 bit is met while encoding magnitude part from MSB to LSB, and insignificant before the first 1 bit appears. The context of each bit is determined by significant situations of its neighbors. There are four coding methods to generate context for each bit in a code block: Zero Coding, Run-Length Coding, Sign Coding, and Magnitude Refinement.Every bitplane is encoded using 3 passes in turn. Each pixel in a bitplane is encoded in one of 3 passes. Pass 1 is Significant Propagation Pass. Pixels having at

least one significant neighbor are coded in this pass. Pass 2 is Magnitude-Refinement Pass. All significant pixels are coded in this pass. Pass 3 is Clean-up Pass. Pixels not coded in first two passes are coded in this pass. While coding a bitplane, every pixel is checked once in all 3 passes to detennine if this pixel should be coded. In Taubmans architecture, a straightforward method is used. Every single bit is checked and (or) coded in all 3 passes, which cost total 3 clocks. Coding a 64x64 code block with 8-bit precision will take 64x64 8 cl3oc ks. That makes tier- 1 coder become bottleneck of PEG-2000 system design.

4.3.1 Concepts for Bitplane, Scanning order and Significant

The key idea of EBCOT coding is bitplane scanning, as presented in Figure 4.4, rather than coefficient scanning. Every bitplane takes three passes. The coefficient-bits scanning order in each pass is stripe-based method with stripe height of 4 compared with zigzag scan adopted in JPEG. An example code-block scan pattern of a code-block with size 8 x n is shown in Figure 4.5.

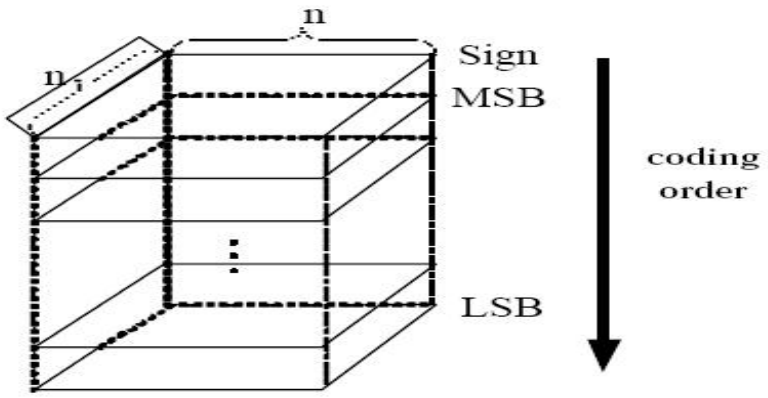

Figure 4.4: The bitplane views for quantized coefficients with code block size n x n [12]

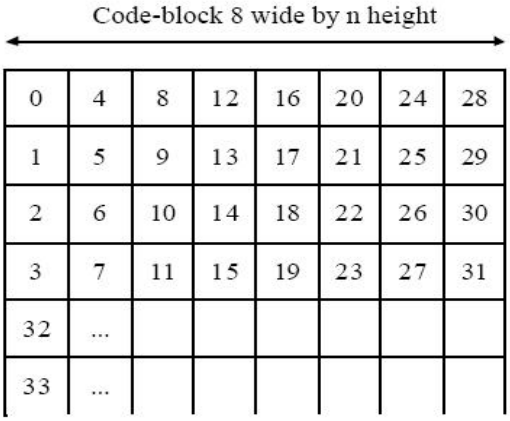

Figure 4.5: Context modeling scanning sequence for one bitplane in each pass [12]

Besides, the EBCOT algorithm starts coding process when the first non-zero bitplane met and skips the all- zero bitplane. The number of skipped bitplanes is record in the packet header.In the EBCOT context modeling, the significant bit is employed in the pass and coding decision. A sample became significant when the first 1 of magnitude bit is found, and this concept is illustrated in Figure 4.6, and the sign bit is coded instantly after the first significant bit coded.

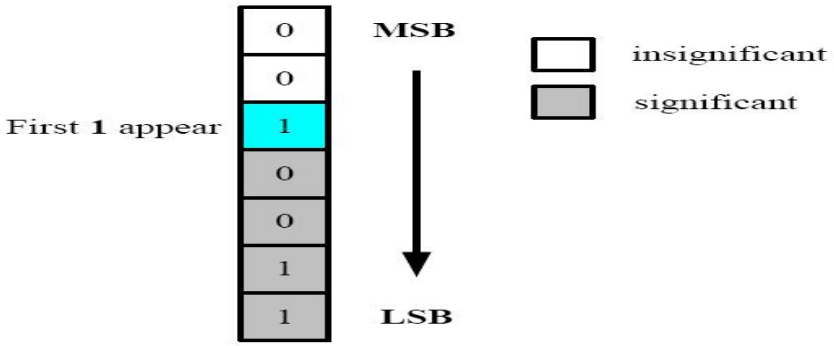

Figure 4.6: An example for the significant state transition [12]

4.3.2 Fractional Bitplanes Coding: Three Coding Passes

To achieve the EBCOT coding algorithm, the coefficients in a code block must be encoded from the most significant bitplane (MSB) to the least significant bitplane (LSB) in the one of three coding passes. They are Significant Propagation Pass (Pass1), Magnitude Refinement Pass (Pass2) and Cleanup Pass (Pass3) in sequence. Figure 4.7 illustrates the concept of fractional bitplanes coding.

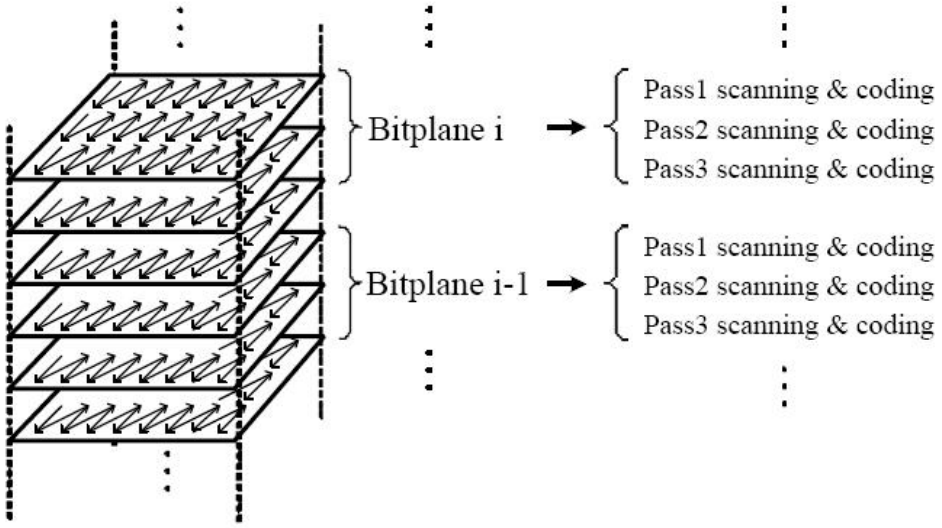

Figure 4.7: Fractional bitplanes coding
[12]

Significance Propagation Pass (Pass1)

This coding pass focuses on the coefficient-bits that are insignificant but have at least one of eight neighbors being significant, as shown in Figure 4.8. In another words, the most proper coefficients becoming significant are searched and encoded in Pass1. In Pass1 stage, the zero coding and sign coding operations are used.

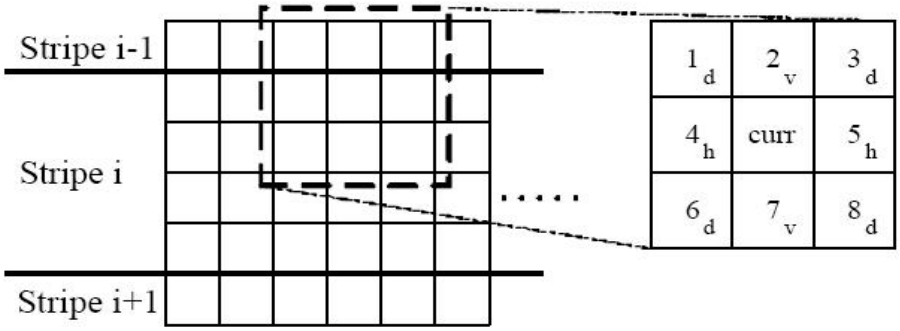

Figure 4.8: The current coding bit and its eight neighbors [12]

Magnitude Refinement Pass (Pass2)

The coefficients-bits that have become significant and are not coded during coding Pass1, i.e. the coefficients-bits have become significant in previous bitplane, are found and coded in Pass2. The coding operation in Pass2 is magnitude refinement coding.

Clean Up Pass (Pass3)

These remained coefficients rejected by pass1 and 2 are coded in Pass3. In this pass, the zero coding, sign coding and run- length coding operations are used. Notice that only Pass3 codes the first non- zero bitplane because the coefficients are all insignificant in the beginning. Compared with other non-zero bitplanes, each coefficient-bit in a bitplane is coded by one of three coding passes. This coding property causes the context modeling to waste the most time on checking and skipping the redundant bits. The EBCOT context modeling being the bottleneck for the JPEG2000 system is expected.

4.4 Four Types Coding Operations

As introduced before, the EBCOT context modeling assigns the coefficients-bits a context with different coding operations depending on the pass type and the significant state of eight neighbor-bits. The eight immediate neighbor-bits as illustrated in Figure 2.13 can be classed with horizontal set (neighbor 4, 5), vertical set (neighbor 2, 7) and diagonal set (neighbor 1, 3, 6 and 8). We now introduce the four different coding operations for context modeling.

• **Zero Coding (ZC)**

Zero coding is used in Pass1 and Pass3 coding. If the coefficient-bit is insignificant, ZC uses on of 9 different contexts in Table I assigning to the coefficient-bit depending on the significant state of immediate neighbors. Note that if the current coding coefficient-bit lies beyond the boundary of code block, its neighbors are taken as insignificant for context assignment. This is due to reduce the dependence between different code blocks.

In Table I ΣH, ΣV, and ΣD are represented as the sum of significant neighbors in horizontal, vertical, and diagonal directions, as shown in Figure 4.8. table

ΣH	**ΣV**	**ΣD**	**ΣH**	**ΣV**	**ΣD**	**ΣH + ΣV**	**ΣD**	**Context Label**
2	x	x	x	2	x	x	≥3	8
1	≥1	x	≥1	1	x	≥1	2	7
1	0	≥1	0	1	≥1	0	2	6
1	0	0	0	1	0	≥2	1	5
0	2	x	2	0	x	1	1	4
0	1	x	1	0	x	0	1	3
0	0	≥2	0	0	≥2	≥2	0	2
0	0	1	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0

Table I: Context assignment table for zero coding

• **Sign Coding (SC)**

Sign coding is used in Pass1 and Pass3, which the same as ZC. When a coefficient switches from insignificant to significant, SC operation turns on and the sign bit for the coefficient is coded to one of 5 different contexts. The decision of contexts in SC is depend upon the sign and the significance of the horizontal and vertical neighbors as presented in Table II and Table III. Notices that SC is operated at most once for each coefficient. Referring to Table II, the XOR bit is employed to generate the

Sign Contribution	**Significant, (+)**	**Significant, (-)**	**Insignificant**
Significant, (+)	1	0	1
Significant, (-)	0	-1	-1
Insignificant	-1	-1	0

Table II: Sign contribution of the horizontal and vertical neighbors for sign coding

Horizontal Contribution	**Vertical Contribution**	**Context Label**	**XOR bit**
1	1	13	0
1	0	12	0
1	-1	11	0
0	1	10	0
0	0	9	0
0	-1	10	1
-1	1	11	1
-1	0	12	1
-1	-1	13	1

Table III: Context assignment table for sign coding

output D of EBCOT context modeling, as defined in Equation (4.1).

$$D = sign\_bit \oplus bit \tag{4.1}$$

• **Magnitude Refinement (MR)**

The magnitude refine operation is utilized in Pass2 coding. Based on the significant state for 8 neighbors and whether or not the coefficients are first refining, MR assigns one of 3 contexts in Table IV to the coefficients-bits that have been significant in previous bitplanes.

ΣH + ΣV + ΣD	**First refinement for this coefficient**	**Context Label**
x	FALSE	16
≥1	TRUE	15
0	TRUE	14

Table IV: Context assignment table for magnitude refinement

• **Run-Length Coding (RLC)**

Run-length coding is operated in Pass3 coding. It is enabled when the following conditions are true.

• Four consecutive coefficients in the same stripe are insignificant.

• The neighbors for the consecutive four coefficients must be insignificant, too.

The coefficients-bits relate to RLC is illustrated in Figure 4.9. Compared RLC with ZC, they are used to code the insignificant coefficients-bits but the RLC assigns one context to four coefficients-bits.

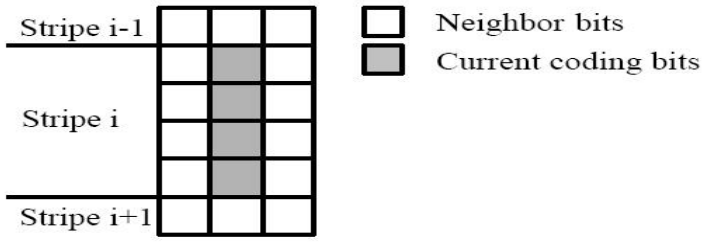

Figure 4.9: Current coding bits and their neighbors for RLC operation

Chapter 5

Simulation Environment

5.1 The SimpleScalar Tool Set

The SimpleScalar tool set is a system software infrastructure used to build modeling applications for program performance analysis, detailed microarchitectural modeling, and hardware-software co-verification. Using the SimpleScalar tools, users can build modeling applications that simulate real programs running on a range of modern processors and systems. The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. The SimpleScalar tools are used widely for research and instruction, for example, in 2000 more than one third of all papers published in top computer architecture conferences used the SimpleScalar tools to evaluate their designs. In addition to simulators, the SimpleScalar tool set includes performance visualization tools, statistical analysis resources, and debug and verification infrastructure. [15] SimpleScalar [16] simulators can emulate the Alpha, PISA, ARM, and x86 instruction sets. The tool set includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations. All of the simulators distributed with the current release of SimpleScalar can run programs from any of the above

listed instruction sets. Complex instruction set emulation (e.g., x86) can be implemented with or without microcode, making the SimpleScalar tools particularly useful for modeling CISC instruction sets. The PISA instruction set (a.k.a. the portable instruction set architecture) is a simple MIPS-like instruction set maintained primarily for instructional use. A GNU GCC-based cross-compiler and pre-built libraries are also available for this target. The PISA target is particularly useful for computer engineering instruction as the tools can be built on a wide range of host platforms, including Linux/x86, Win2000, SPARC Solaris, and others. SimpleScalar builds on most 32-bit and 64-bit flavors of UNIX and Windows NT-based operating systems. The internal software architecture of the tool set includes a host interface module, permitting fast porting to other host platforms. The host interface module permits cross-endian emulation, thus it is possible to use emulate a target on a host platform with a different endian, e.g., running Alpha ISA emulation on a SPARC Solaris host platform. Most SimpleScalar users and developers (including SimpleScalar LLC) [16] use SimpleScalar on Linux/x86.

The type of simulator varies from fast functional simulators to detailed processor model simulators that are able to simulate caches, branch predictors, and many other features of modern processors. Since SimpleScalar can emulate the ARM instruction set, and because its reliability is in very high levels (in 2000 more than one third of all papers published in top computer architecture conferences used the SimpleScalar tools to evaluate their designs), it appears ideal for emulating the ARM processor in our case.

5.2 Sim-Panalyzer Tool

The Sim-Panalyzer [17] tool on the other hand is an infrastructure for micro-architectural power simulation. It is broken out into several components that model distinct parts of a computer: cache power models; datapath and execution unit power models; clock tree power models; and I/O power models. These power models can be configured

into an augmented SimpleScalar simulator that will then produce power consumption figures. It is positioned above the "sim-outorder" component of the SimpleScalar simulator. The Sim-Panalyzer program contains components that model specific parts of the ARM processor. Sim-Panalyzer focuses efficiently on basic micro-architectural blocks and provides power information over a wide range of power dissipation sources, such as caches, clock trees, external I/O, on-chip memories and datapath and execution blocks. For micro-architectural blocks, the basic method to calculate the power dissipation is by multiplying the appropriate switching capacitance by the number of micro-architectural accesses. For external I/O accesses, a transaction model counts the I/O pin switches in a cycle accurate way. Moreover, support for more sophisticated and accurate power models is provided through libraries, containing basic building blocks for the embedded logic simulator and ability to extract switching capacitance for CMOS gates. The logic simulator accumulates the switching capacitance of internal nodes into a switching capacitance estimation of each functional block's node.

5.3 Compilation of Sim-Panalyzer

Untar "sim-panalyzer-2.0.3.tar.gz" into your install directory. The source code can be downloaded from the website.[17] Sim-Panalyzer has currently been compiled using gcc 3.2. Other gcc versions have not been tested thoroughly. 'make sim-panalyzer' generates a binary for the simulator. Go to the root directory for each version './Implementations/targetmachine/sim-panalyzer2.0.3' and execute this command. This should generate the executable file 'sim-panalyzer'. [18]

5.4 How to run the simulator

Sim-Panalyzer has created a separate script file that parses the cmd file. The format for a cmd file is similar to a Microsoft Windows ini file. The configuration variables

are divided into sections and are parse through these sections to generate an appropriate configuration for our simulator. An example of a cmd file is shown in Figure 5.1. Power configurations can be given as follows below. [18]

```
[Component]
AIO
DIO
IL1 Cache
DL1 Cache
IL2 Cache
DL2 Cache
ITLB
DTLB
IRF
FPRF
Random Logic
Clock

[Global]
supply_voltage=1.8
frequency=200

[AIO]
frequency=200
IO_voltage=3.3
numberofbufferstages=5
microstrip length=10
external load=1

[DIO]
frequency=200
IO_voltage=3.3
numberofbufferstages=5
microstrip length=10
external load=1
```

Figure 5.1: Example of cmd file
[18]

The [Component] section that is shown in the beginning of Figure 5.1 represents the components that are intended for power analysis. Currently the components that are supported are Caches, Branch Target Buffers, Branch Predictors, Register files, Clock Trees and Random Logic. Based on the chosen components in the [Component]

section, the configuration variables are defined in the following subsections. For example, the [AIO], which configures the address IO pads, has the parameters frequency for the bus frequency, IO_voltage to describe the supply voltage for the IO pad, Buffer ratio for buffer sizing, microstrip length for modeling the PCB, and finally external load to model the load that is connected to this IO. test_arm.cmd is located in the source code, is the template command files that can be used as a reference.

It is important to note that in the cmd file, we assume capacitance to be in pF, time unit to be in ps, frequency to be in MHz, and voltage to be in V. The power configurations are then integrated with the architectural configurations and create a single configuration file. Power configuration templates are also provided for these microprocessors in the $./cmd\_files/directory$. The typical method for executing Sim-Panalyzer would be executing the $/gen\_cfg\_< target\_machine > .plscript$ and then using the generated output file as the configuration file for Sim-Panalyzer.

$\$gen\_cfg\_< target\_machine > .pl < architectural\_config\_filename >$
$< PA\_cmd\_filename >$

$\$sim - panalyzer - config < configurationfilename >< executingprogram >$
$< programparameters >$

5.5 Estimation Procedure

This Section explains the estimation procedure for a Sample Source Code. After building the cross compiler, the command line argument required in order to compile a C application (for example hello.c) for the ARM is the following: [18]

$ **arm-unknown-linux-gnu-g++ -static -o hello hello.c**

After building the Sim-panalyzer tool, the following command is outputting the power dissipation report of the hello.cpp application:

$ **sim-panalyzer -config hello.cfg hello**

The -config defines the name of the configuration file that contains architecture specific information for the ARM processor, such as the operating frequency, supply voltage etc. This configuration file is generated from a script provided by the Sim-Panalyzer tool, that parses a command file. The command file that we use in order to generate the configuration file is the default command file provided by Sim-Panalyzer. [15]

5.6 ARM-LINUX Cross compiler

Since Sim-panalyzer can work only on arm_elf binaries,we n Since the target architecture for our experiments is the ARM architecture, the inputs to the Sim-Panalyzer tool for program emulation and power dissipation calculation must be ARM binaries. In this case, a cross-compiler kit targeting the ARM should be built on a Linux platform in order to acquire an ARM executable from C code. A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the cross compiler runs. Since Sim-panalyzer can work only on arm_elf binaries,we need to use the cross-compiler capable of elf binary and for that GNUARM cross compiler is used. [19]

The Crosstool is a collection of scripts to build and test several versions of gcc and glibc for most architecture supported by glibc.

5.7 SimItARM Simulator

Simit-ARM 3.0 is an instruction-set simulator that runs both system-level and user-level ARM programs within an operating system running on a non-ARM processor by simulating a single-processor memory space and the logical operation of the processor upon it.Simit-ARM reads ELF32 little-endian ARM binaries. [20]

Simit-ARM has build in interpreter called "ema" and Dynamic-compiled Simulator called "ema_jit". Simit-ARM gives the result in terms of system time and CPU

time. Also it gives "total number of instruction" of the ARM program. So it can be useful for Instruction calculation of the ARM program.

5.7.1 SimIt-ARM features:

Full system simulation SimIt-ARM supports the ARM v5 architecture, including the memory management unit and some fundamental I/O devices. High simulation speed On a Pentium D 2.8GHz desktop, the interpreter runs at above 30MIPS.

Built-in debugger The '-d' flag enables a light-weight debugging interface, allowing one to step through a program and to observe register/memory values.

Modular design SimIt-ARM is developed in C++. All simulation states are packaged in C++ classes. Therefore it is easy to create multiple simulator objects for modeling multiprocessor targets.

Chapter 6

The Proposed Design

6.1 Efficient C Programming Techniques for ARM

There are many factors that determine the performance of a program. The choice of hardware can mean the difference between a few MIPS and a few hundred. Good data structures and algorithms are essential, and bookshelves have been written on this topic. A good compiler is also essential. One should evaluate the features and optimization capabilities of a compiler before spending too much time working with it

This section explains different optimization techniques that would be useful for writing efficient C programming for ARM processors. This techniques will speed up execution time and produce low density C code. [21]

6.1.1 BASIC C VARIABLE TYPES

ARM (Advanced RISC machine) is a 32 bit RISC processor.Because of their power saving features,ARM CPUs are dominant in the embedded systems where low power is a critical design goal. The ARM ANCI C compiler is capable of generating high quality machine code. The ARM architecture is RISC load/store architecture. In other words you must load values from memory into registers before acting on them.

Most ARM data processing operations are 32-bit only. For this reason, you should use a 32-bit data type, int or long, for local variables wherever possible. Avoid using char and short as local variable types, even if you are manipulating an 8-bit or 16-bit value. For the types char and short the compiler needs to reduce the size of the local variable to 8 or 16 bits after each assignment. This is called sign-extending for signed variables and zero-extending for unsigned variables. It is implemented by shifting the register left by 24 or 16 bits, followed by a signed or unsigned shift right by the same amount, taking two instructions (zero-extension of an unsigned char takes one instruction).

We should use unsigned int instead of int if we know the value will never be negative. Some processors can handle unsigned integer arithmetic considerably faster than signed (this is also good practice, and helps make for self-documenting code).

So, the best declaration for an int variable in a tight loop would be:

register unsigned int variable_name;

6.1.2 Global variables

Global variables are never allocated to registers. Global variables can be changed by assigning them indirectly using a pointer, or by a function call. Hence, the compiler cannot cache the value of a global variable in a register, resulting in extra (often unnecessary) loads and stores when globals are used. We should therefore not use global variables inside critical loops. If a function uses global variables heavily, it is beneficial to copy those global variables into local variables so that they can be assigned to registers. This is possible only if those global variables are not used by any of the functions which are called.

6.1.3 Local variables

Where possible, it is best to avoid using char and short as local variables. For the types char and short, the compiler needs to reduce the size of the local variable to

8 or 16 bits after each assignment. This is called sign-extending for signed variables and zero extending for unsigned variables. It is implemented by shifting the register left by 24 or 16 bits, followed by a signed or unsigned shift right by the same amount, taking two instructions (zero-extension of an unsigned char takes one instruction). These shifts can be avoided by using int and unsigned int for local variables. This is particularly important for calculations which first load data into local variables and then process the data inside the local variables. Even if data is input and output as 8- or 16-bit quantities, it is worth considering processing them as 32-bit quantities.

6.1.4 Function Argument Types

The char or short type function arguments and return values introduce extra casts. These increase code size and decrease performance. It is more efficient to use the int type for function arguments and return values, even if you are only passing an 8-bit value. [22]

6.1.5 Using Aliases

Consider the following example -

```
void func1( int *data )
{
    int i;
    for(i=0; i<10; i++)
    {
    anyfunc( *data, i);
    }
}
```

Even though *data may never change, the compiler does not know that anyfunc () did not alter it, and so the program must read it from memory each time it is used - it may be an alias for some other variable that is altered elsewhere. If we know it

won't be altered, we could code it like this instead:

```
void func1( int *data )
{
    int i;
    int localdata;
    localdata = *data;
    for(i=0; i<10; i++)
    {
    anyfunc ( localdata, i);
    }
}
```

This gives the compiler better opportunity for optimization.

6.1.6 REGISTER ALLOCATION

The most important optimization supported by the ARM compilers is called register allocation. This is a process where the compiler allocates variables to ARM registers, rather than to memory. This has the advantage that those variables can be accessed quickly whenever needed, without needing instructions to transfer them from/to memory. As a result of register allocation, most variables are kept in registers, resulting in dramatic improvement in code size and performance. When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called spilled or swapped out variables since they are written out to memory [22]. Spilled variables are slow to access compared to variables allocated to registers. If the compiler does need to swap out variables, then it chooses which variables to swap out based on frequency of use. To implement a function efficiently, you need to minimize the number of spilled variables and ensure that the most important and frequently accessed variables are stored in registers. The C compiler can assign 14 variables to registers without spillage. Some compilers

use a fixed register such as r12 for intermediate scratch working and do not assign variables to this register. Therefore, try to limit the number of local variables in the internal loop of functions to 12. The compiler should be able to allocate these to ARM registers. [22]

6.1.7 Use of access types

For global data, use the static keyword (or C++ anonymous namespaces) whenever possible. In some cases, static allows a compiler to deduce things about the access patterns to a variable. The static keyword also "hides" the data, which is generally a good thing from a programming practices standpoint. Declaring a function as static is also helpful in many cases.

6.1.8 Other tips

• Minimize the use of global variables.

• Declare anything within a file (external to functions) as static, unless it is intended to be global.

• Use word-size variables if you can, as the machine can work with these better (instead of char, short, double, bit fields etc.).

• Don't use recursion. Recursion can be very elegant and neat, but creates many more function calls which can become a large overhead.

• Avoid the sqrt() square root function in loops - calculating square roots is very CPU intensive.

• Use Single dimension arrays as they are faster than multi-dimension arrays.

• Compilers can often optimize a whole file - avoid splitting off closely related functions into separate files, the compiler will do better if it can see both of them together (it might be able to inline the code, for example).

• Single precision math may be faster than double precision - there is often a compiler switch for this.

• Addition is quicker than multiplication - use val + val + val instead of val * 3. puts() is quicker than printf(), although less Flexible.

• Use #defined macros instead of commonly used tiny functions - sometimes the bulk of CPU usage can be tracked down to a small external function being called thousands of times in a tight loop. Replacing it with a macro to perform the same job will remove the overhead of all those function calls, and allow the compiler to be more aggressive in its optimization.

• Binary/unformatted file access is faster than formatted access, as the machine does not have to convert between human-readable ASCII and machine-readable binary. If you don't actually need to read the data in a file yourself, consider making it a binary file.

6.2 EBCOT Modifications

The reason of EBCOT Tier-1 occupying highest computation is that the operations are bit-level processing. This study exploit parallel and pipelined architecture to accelerate the operations and reduce power consumption. There are three factors affecting parallelism of CF: 1) scanning order, 2) checking neighbors, and 3) changing state. Within a stripe, all bits are scanned in a specific order. The context of a bit is generated by checking states of its neighborhood bits. However, the state of the coded bit can change and affect later coding results, as shown in Figure. 6.1.

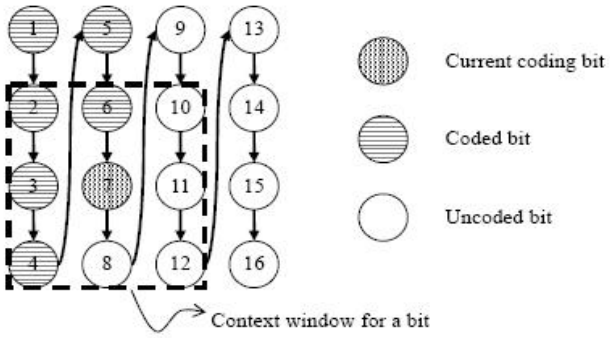

Figure 6.1: Scanning order

Observation shows the data dependency of sixteen bits and depict a DFG, as shown in Figure. 6.2. And from that the 16-bit parallel architecture is proposed. Since its delay is ten, not sixteen, times than sample-based architecture, can use little voltage to achieve the identical throughput. Estimate shows that the 16-bit parallel architecture can save power consumption compared with the sample-based architecture.

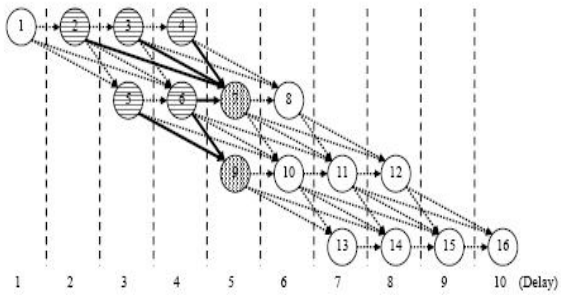

Figure 6.2: Data dependency within sixteen bits

By alleviating the frequency of memory access and utilization of efficient memory bandwidth, power consumption can also be reduced as well. Here the memory-saving algorithm [23] and proposed memory arrangement for 16-bit parallel context generat-

ing are followd. Every eight bits are grouped as word, and words are placed in three memories in an interleaving format, as shown in Figure. 6.3. During memory access, the order of memory data depends on the stripe. When want to code Stripe n, the data order is (C, A, B). After memory arrangement, we can utilize efficient memory bandwidth.

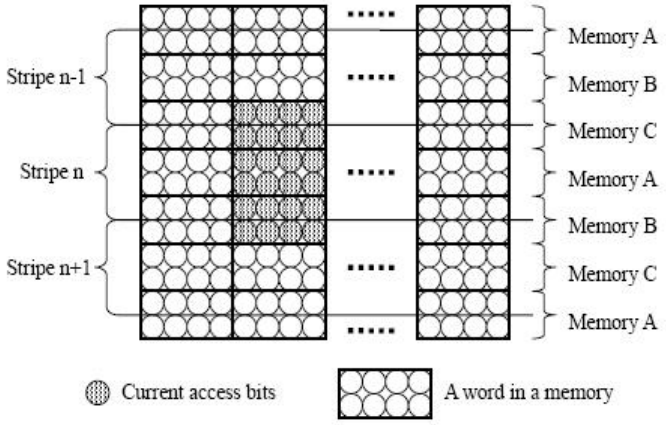

Figure 6.3: Memory arrangement

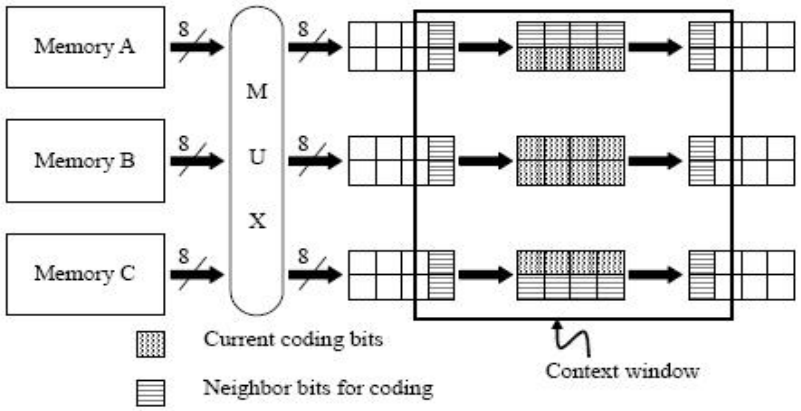

Figure 6.4: Context window and memory bandwidth

Here use of nine 8-bit shift registers for the context window, as shown in Figure. 6.4. The shaded samples represent the sixteen coding bits, and their neighbors

should be included as well. The context window can reuse data locally. The memory bandwidth is twenty-four bits.

Although 16-bit parallel context generating can reduce cycles, there still are many wasted cycles.For that The stripes skipping method can be applied. This strategy is easier to implement for 16-bit parallel processing than other multiple-column-skipping. Moreover, multiple-columns kipping incurs a little memory overhead, but the memory requirement of stripe-skipping is less. This strategy just use three 16-bit registers to record the coding condition of all stripes in three passes enough. This memory organization and access method can lead us to reduce the instruction count and so will help to reduce total number of instruction and so that will lead to save the power by maintaining the same quality of the image without affecting the PSNR and MSE for the image. But if the in the stripe skipping method if certain limit of strike skip is not followed then it will results in degraded performance in terms of PSNR and image quality deteriorates.

Chapter 7

Results

7.1 Experienced Result

In this chapter, the results based on the so far discussed theory and methodology of Chapters 3, 4, 5 and 6 are introduced. The power dissipation values for the software implementation are presented. These values were derived using the estimation procedure described in Chapter 6. The research work carried out in this thesis mainly concentrate on Optimizing EBCOT part of JPEG2000. Table I shows the power analysis of the different part of JPEG2000. From the Table I , among the component

JPEG2000 Modules	**Lossy Compression**	**Lossless Compression**
DWT	20.1%	12.2%
Quantization	5.5%	5.8%
EBCOT	**70.4%**	**77.1%**
Coefficient Bit Modeling	51.8%	55.0%
Arithmetic Coding	6.9%	8.2%
Rate Distortion Control	11.7%	13.9%
Others	4.0%	4.9%
Total	100%	100%

Table I: Complexity estimation (execution time) for JPEG2000 encoder* [14]

of JPEG2000 the most time consuming part is EBCOT which takes around 70% of the total execution time.

7.1.1 By SimIt-ARM

The ebcot arm binary file which have been generated in the gnuarm cross compiler .Using "*./build/bin/ema*" command where ema is the execution command which will execute the ebcot program on the ARM simulator environment and give output and required user time,system time,simulation speed and total instruction required for executing the ARM binary file.
$[root@mtech - 12SimIt - ARM - 3.0]\#$ ./build/bin/ema test/ebcot

ema: Simulation starts ...
ema: Program exited normally.
Total user time : 56 sec.
Total system time: 0.000 sec.
Simulation speed : 6.402e+07 inst/sec.
Total instructions : 1065347
After applying the efficient C programming techniques and applying the modification in EBCOT which applies the stripe skipping method suggested in section 6.2 we get,
$[root@mtech - 12SimIt - ARM - 3.0]\#$ ./build/bin/ema test/ebcot_modified

ema: Simulation starts ...
ema: Program exited normally.
Total user time : 54 sec.
Total system time: 0.000 sec.
Simulation speed : 6.402e+07 inst/sec.
Total instructions : 1032103

7.1.2 By Sim-panalyzer

The average power consumed by a microprocessor while running a certain program is given by: P = I x V_{cc}, where P is the average power, I is the average current and V_{cc} is the supply voltage. Since power is the rate at which energy is consumed, the energy consumed by a program is given by: E = P x T, where T is the execution time of the program. This in turn is given by: T = N x Υ, where N is the number of clock cycles taken by the program and Υ is the clock period. In common usage, the terms power consumption and energy consumption are often interchanged. However, it is important to distinguish between the two when we talk of either of these in the context of programs running on wireless systems. Since wireless systems run on the limited energy available in a battery. Therefore, the energy consumed by the system or by the software running on it, determines the length of the battery life. Energy consumption is thus the focus of attention.

The Sim-Panalyzer tool output gives the average power dissipation for the components listed in the cmd file. The components for which power dissipation is given as follows:

Address Input-Output (AIO): AIO which configures the address IO pads, has the parameters frequency for the bus frequency, IO_voltage to describe the supply voltage for the IO pad, Buffer ratio for buffer sizing, micro strip length for modeling the PCB, and finally external load to model the load that is connected to this IO.

Data Input-Output (DIO): DIO, similar to AIO configures the data IO pads, has the parameters similar to AIO i.e. frequency, IO_voltage, Buffer ratio, micro strip length and external load.

Instruction Register File (IRF): IRF configures the instruction register bank where the parameter Capacitance is used.

Instruction Level 1 (IL1) Cache: IL1 cache configures the instruction level 1 cache that has the parameters number of bitlines for the bit lines, number of wordlines for the word lines in the cache and Capacitance that consumes energy in IL1

cache.

Data Level 1 (DL1) Cache: DL1 cache configures for data level 1 cache has parameters similar to IL1 cache i.e. number of bitlines, number of wordlines and Capacitance.

Instruction Table Look-aside Buffer (ITLB): ITLB configures parameters number of bitlines, number of wordlines and Capacitance.

Data Table Look-aside Buffer (DTLB): DTLB also configures parameters number of bitlines, number of wordlines and Capacitance.

	EBCOT	
Average Power Dissipation in Component	**Standard**	**Modified**
AIO	1.7893	1.7818
DIO	7.2137	7.0208
IRF	0.6618	0.6482
*IL*1 Cache	2.1658	2.1610
*DL*1 Cache	1.2492	1.2401
ITLB	1.0180	1.0980
DTLB	0.7150	0.7090
Clock	3.0210	3.0210
ALU	0.0022	0.0022
Micro-arch	4.3250	4.3015

Table II: Average Power Dissipation for EBCOT on StrongARM-1110

Unit	**Standard**	**Modified**
Average Power (W)	22.16	21.98
Instruction Committed	465740	452414
Instruction Executed	1064352	1031037
Cycles Per Instruction	1.5512	1.5851
Clock Frequency (MHz)	43.47	43.47

Table III: Average Power Consumption for EBCOT on StrongARM-1110

Table II gives the average power dissipation for the different component for the EBCOT on Intels StrongARM-1110 processor. And Table III gives the average power consumption of the same EBCOT. By using the optimization techniques discussed in Chapter 6 for efficient C programming for ARM and EBCOT, gives the modified version of EBCOT. By using the methodology explained in Chapter 6 a small reduction in the power consumption is achieved and a small change is achieved in the number of instruction executed. The use of decrementing loop, reduction in number of local variables, and properly assigning the data types to the variable helped to reduce this number of instructions.

By analyzing the results ,it is sure that the main concern in terms of the power consumption of the EBCOT algorithm is the memory and its access pattern.And so the main component of power consumption is DIO due to high access of stripes in each pass of EBCOT. Also simulation shows that the number of instruction for EBCOT is too high and so total number of cycle count is high. Due to the stripe-skipping method sometimes AIO access exceeds than that of standard algorithm. So in stripe-skipping method skip bound should be selected such that it will not increases the AIO access.

Chapter 8

Conclusion and Future Scope

8.1 Conclusion

The basic concepts and characteristics of JPEG200 algorithms and Embedded Block Coding for Optimized Truncation($EBCOT$) are introduced.The reduction in the power consumption is an important issue in modern embedded systems and is the main concern in the study; therefore, any method that provides a way to reduce this consumption must be studied,evaluated. The work accomplished shows how power aware EBCOT system has to be made in embedded environment. In this work, a framework for analyzing the power consumption of EBCOT algorithm is presented. Among the JPEG2000 compression architecture EBCOT is the main source of computation time and power.So EBCOT becomes the bottleneck.As the power cost of any algorithm is depending on the total number of instruction and the memory operation ,in the study C programming optimization techniques and memory access pattern is applied on EBCOT. From the results shown in Figure 8.1 it can be seen that the total instruction count reduces to about 1% and power consumption to 0.75%. But significant reduction in total instruction count and power consumption will not be achieved due to the high memory access count. So if the memory organization and architecture has been changed or specially efficient VLSI designed SoC can provide

significant power efficient and Also in this proposed design

Embedded System lifetime is severly constrained by the available energy sources so

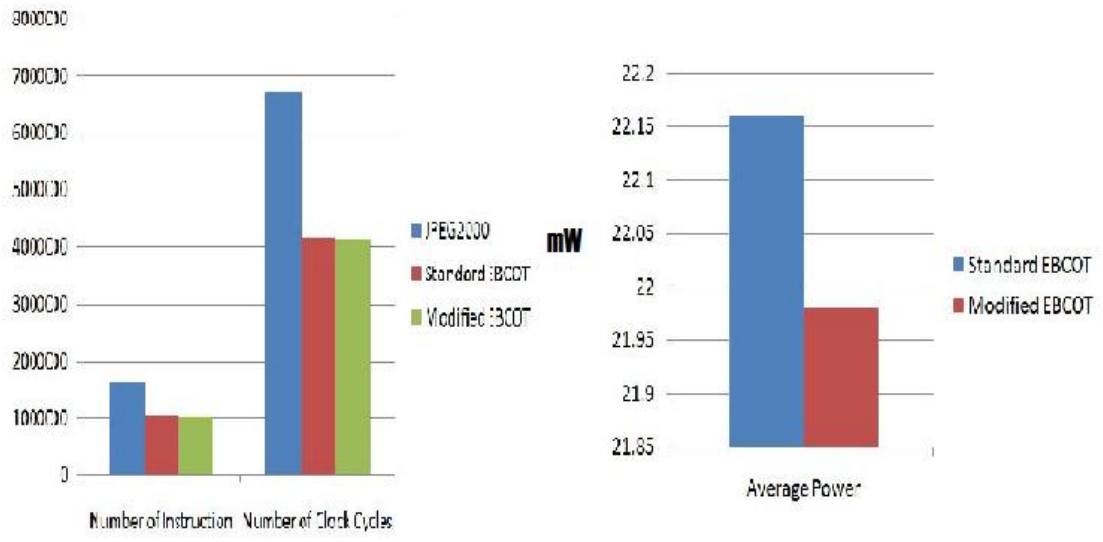

Figure 8.1: (a). Comparison in terms of Total Instruction and Clock Count (b). Comparison in terms of Power

naive and power-oblivious operation is not appropriate. System software techniques and efficient C programming techniques are more general, and hence have broader applicability than application-specific techniques, but application specific techniques can be still more effective. Since the particular mix of storing, compression computation, and restorage is usually quite application-specific, with different applications demonstrating dramatically different mixes, power profiles can vary widely. System software can provide only general solutions to power management but if the system can expose appropriate power management interfaces or implement power optimizations, as many of the techniques presented in this study do, then applications can influence or control system components and reduce power without having to be explicitly power-aware.

8.2 Future Scope

By analysis of the result,the efficient C programming techniques will result in reduction of instruction in a small amount. Also the proposed EBCOT modification will lead to change in memory architecture. The more efficient way of reducing the power is to design EBCOT memory architecture and design specialized efficient VLSI design or implement a specialized EBCOT SoCs. The specialized design hardware architecture of EBCOT may enhance the performance and can reduce the power by a significant amount. Hardware implementation of Specialized designed EBCOT tier-1 architecture will lead to a good and very efficient implementation which can save power in quite a good manner.

Appendix A

SimpleScalar Simulator

A.1 SimpleScalar Tool set

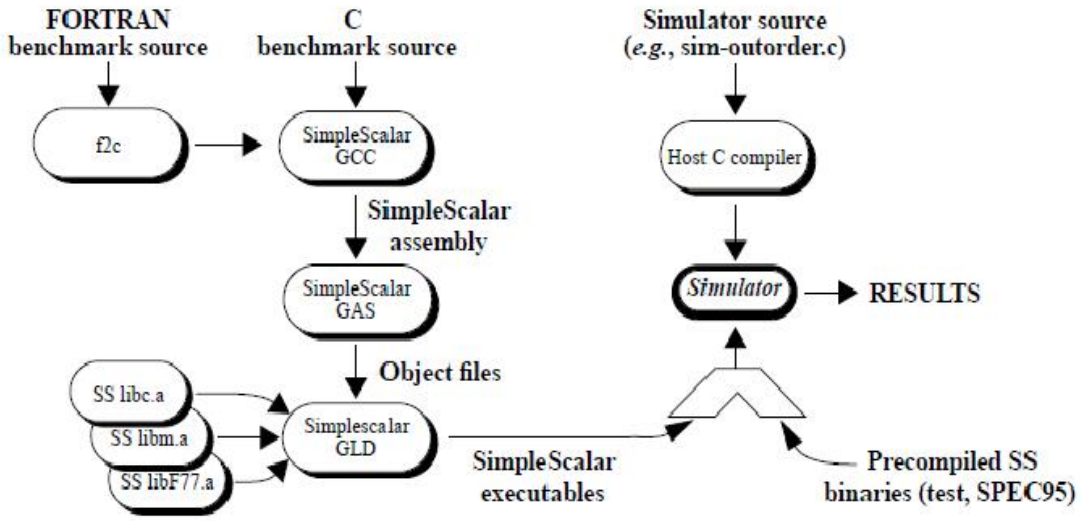

Figure A.1: SimpleScalar tool set overview

The SimpleScalar architecture is derived from the MIPS-IV ISA. The tool suite defines both little-endian and big-endian versions of the architecture to improve portability (the version used on a given host machine is the one that matches the endianness

of the host). The semantics of the SimpleScalar ISA are a superset of MIPS with the following notable differences and additions:

- There are no architected delay slots: loads, stores, and control transfers do not execute the succeeding instruction.
- Loads and stores support two addressing modesfor all data typesin addition to those found in the MIPS architecture. These are: indexed (register+register), and auto-increment/ decrement.
- A square-root instruction, which implements both singleand double-precision floating point square roots.
- An extended 64-bit instruction encoding.

SimpleScalar has different sub modules that can be use to analyze the different parameter of the architectural component

- Sim-Fast
- Sim-Safe
- Sim-Profile
- Sim-Cache/Sim-Cheetah
- Sim-Outorder

Appendix B

List of Useful Web sites

- GNUARM Crosscompiler : A cross compiler for ARM

  `http://www.gnuarm.com`

- SimIt-ARM Simulator

  `http://www.simitarm.com`

- The SimpleScalar-Arm Power Modelling Project (Sim-panalyzer)

  `http://www.eecs.umich.com/~panalyzer`

- SimpleScalar LLC

  `http://www.simplescalar.com`

References

[1] V. Tiwari, S. Malik, and A. Wolfe, "Instruction level power analysis and optimization of software," tech. rep., Dept of Electrical Engg, Princeton University, January 1996.

[2] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power optimization," tech. rep., Dept of Electrical Engg,Princeton University, January 1994.

[3] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: An overview," tech. rep., Dept of Electrical Engg,Princeton University, Octomber 1994.

[4] G. Z. N. Cai and C. H. Lim, "Microarchitectural power analysis for cpu power/performance optimization," tech. rep., INTEL Research Group.

[5] D. Taubman, E. Ordentlich, M. Weinberger, and G. Seroussi, "Power a first class architecture design constarint," tech. rep., COMPUTER SOCIETY STANDARDS WORKING GROUP.

[6] J. Agrawal, "Implementing security protocols in wireless embedded system: A power performance analysis," Master's thesis, Institute Of Technology, Nirma University, May 2007.

[7] S.-H. Yang and W.-J. Liao, "Performance analysis of embedded wavelet coders," tech. rep., Dept of Computer Sci & Tech,National Taipei University of Technology, Octomber 2007.

[8] "Jpeg 2000 part 1 final committee draft version 1.0," in *"JPEG 2000 IMAGE CORE CODING SYSTEM"*, ISO/IEC FCD15444-1 : 2000, March 2000.

[9] M. D. Adams, *JasPer Software Reference Manual (Version 1.900.0)*. Dept. of Electrical and Computer Engineering, University of Victoria, P. O. Box 3055 STN CSC, Victoria, BC, V8W 3P6, CANADA, December 2006. www.ece.uvic.ca.

[10] D. Taubman, *Kakudo Software Reference Manual*. Dept. of Electrical and Computer Engineering, University of New South Wales, Sydney. www.kakudo.org.

[11] C. Christopoulos and A. Skodras, "The jpeg2000 still image coding system: An overview," tech. rep., Media Lab, Ericsson Research Corporate Unit, Ericsson Radio Systems AB, S-16480 Stockholm, Sweden, November 2000.

[12] K.-L. Lin, "Analysis and architecture design for jpeg2000 still image encoding system," Master's thesis, Department of Electrical Engineering, National Central University, Chung-Li, 32054, Taiwan, ROC.

[13] D. Taubman, E. Ordentlich, M. Weinberger, and G. Seroussi, "Embedded block coding in jpeg2000," tech. rep., School of Electrical Engg. & Telecommunication, University of New South Wales And HP Research Laboratories,Palo Alto, Sydney, January 2000.

[14] Kuan-Fu, C. jr Lian, H.-H. Chen, and L.-G. Chen, "Analysis and architecture design of ebcot for jpeg2000," tech. rep., Dept of Electrical Engg,National Taiwan University of Technology, Octomber 2001.

[15] D. Burger and T. Austin, *The SimpleScalar Tool Set.* Simplescalar LLC, 2395 Timbercrest Court,Ann Arbor,MI,48105, 2.0 ed., January 2001.

[16] Web portal SimpleScalar LLC. `http://www.simplescalar.com`.

[17] Web portal The SimpleScalar-Arm Power Modelling Project. `http://www.eecs.umich.com/~panalyzer`.

[18] The SimpleScalar-Arm Power Modelling Project, *Sim-panalyzer Reference Manual.*

[19] Web portal GNUARM Crosscompiler : A cross compiler for ARM. `http://www.gnuarm.com`.

[20] Web portal SimIt-ARM Simulator. `http://www.simitarm.com`.

[21] *Writing Efficient C for ARM.* ARM DAI 0034A, January 1998.

[22] A. N. Sloss, D. Symes, and C. Wright, "Arm system developers guide: Designing and optimizing system software," tech. rep., Elsevier, 2004.

[23] Y.-T. Hsiao, H.-D. Lin, K.-B. Lee, and C.-W. Jen, "High-speed memory-saving architecture for the embedded block coding in jpeg2000," vol. 5, pp. 133–136, IEEE International Symposium on Circuits and Systems, May 2002.