VLSI Implementation of Discrete Cosine Transform Using a New Asynchronous Pipelined Architecture

LEE Chi-wai

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Electronic Engineering

© The Chinese University of Hong Kong
February 2002

Abstract of this thesis entitled:

VLSI Implementation of Discrete Cosine Transform Using a New Asynchronous Pipelined Architecture

Submitted by LEE Chi-wai

for the degree of Master of Philosophy in Electronic Engineering

at The Chinese University of Hong Kong in June 2001

This thesis presents two different asynchronous VLSI implementations of Discrete Cosine Transform (DCT). Although asynchronous design has potential advantages over the synchronous design, the handshaking overhead and the design difficulties limit the speed performance of asynchronous design. In order to break through the barrier, a new asynchronous pipelined architecture is described in this thesis. It relaxes the handshaking protocol and has a simpler architecture, the performance of asynchronous design is improved. Since the new architecture employs dynamic logic, a new technique called Refresh Control Circuit is also introduced to reduce the performance degradation associated with the traditional technique.

The first DCT implementation is realized in a programmable DSP processor. This programmable processor makes use of asynchronous pipeline, dataflow architecture and parallelism, a reasonable but encouraging result of 22Mpixel/sec in DCT operation is obtained with a limited number of arithmetic units.

The second DCT implementation is performed on a dedicated 2D DCT/IDCT processor. It is a fully pipelined design and can operate at 76Mpixel/sec for 2D DCT/IDCT operation. It is capable of processing the high quality MPEG-2 and baseband HDTV signal in real-time, and is competitive to other synchronous designs even less arithmetic units are included in this processor.

The results of the two implementations demonstrate the high performance of the new asynchronous pipelined architecture and the advantages of the asynchronous technique in system design. It also encourages further development in asynchronous design.

摘要

本論文介紹了兩個可應用於離散餘弦變換 (Discrete Cosine Transform) 的異步 (asynchronous) 超大規模集成電路。雖然相對於同步設計 (synchronous design)，異步設計 (asynchronous design) 擁有潛在的優點，但因聯絡額外開銷 (handshaking overhead) 和設計上的困難而限制了異步設計的速度。爲了突破這個界限，本論文描述了一個新的異步管線式架構 (asynchronous pipelined architecture)。它放寬了聯絡協定 (handshaking protocol) 及擁有更簡單的架構，令到異步設計的效能得以提升。因爲這個新的架構應用了動態邏輯 (dynamic logic)，本論文亦提出了一個名叫更新控制電路 (Refresh Control Circuit) 的新技術。這技術能減少因使用傳統技術而導致的效能降格。

第一個離散餘弦變換設計是建於一個可程序數碼訊號處理器 (programmable DSP processor)。這個可程序處理器應用了異步管線、數據流程架構 (dataflow architecture) 及並聯 (parallelism)，在執行離散餘弦變換時能夠在一秒中計算二十二萬素象。在有限的運算部件條件下，這是一個合理而且有鼓勵性的結果。

第二個離散餘弦變換設計是一個專用的二維離散餘弦變換處理器。這處理器是完全管線式設計及能夠在二維離散餘弦變換運作中達到每秒七十六萬素象的運算速度。這結果顯示這處理器能夠實時處理高質素的 MPEG-2 及高清晰度電視

(High Definition Television) 訊號。即使這處理器是使用了比較少的運算部件，但在性能上仍可和其他同步設計競爭。

由這兩個設計所得出的結果可看出新的異步管線式架構的高效能表現及異步技術在系統設計上的好處。另外這結果亦對將來異步設計的發展起鼓舞作用。

Acknowledgements

I would like to express my deepest gratitude to various individuals who provided me with sincere assistance in this research.

First of all, I would like to thank my supervisor, Prof. Oliver, C.S. Choy, for his invaluable guidance, advice, and support during the course of this research work. Notwithstanding his busy schedule, he has worked with me throughout the lengthy and demanding project, providing me continuous comments, patience, supervision, and encouragement. Without his endless help and assistance, this thesis would never have been possible. I would also like to express my gratitude to Prof. C.F. Chan who has given me insightful suggestions during my research work. In addition, a special expression of thanks goes to the research assistant, Mr. Jan Butas, for their kind assistance during my study.

Thanks are also due to my colleagues Mr. Cheng Wan Chi, Mr. Hon Kwok Wai, Mr. Leung Lai Kan, Miss Mak Wing Sum, Mr. Siu Pui Lam and Mr. Tang Tin Yau, the laboratory technician Mr. Yeung Wing Yee, and who have always been my sources of fun and encouragement.

I would also like to thank my close friend, Miss Vivian Tsoi, for her kind assistance and concern throughout the process of this study. Her constant encouragement and everlasting patience and support were my strength, motivation, and inspiration all along. With her cordially support, I could exert my best effort on this study.

At last, I would also like to express my truly gratitude to my parents and my sisters for their understanding and devoted love throughout my whole course of studies. Without their concerns and support, I would not finish my study successfully. I am once again indebted to all of these people.

Lee Chi Wai

June 2001

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Synchronous Design

Synchronous design is the most popular digital circuit design technique today in the VLSI world. In a synchronous circuit, global clock is used to synchronize and trigger all the operations. As the technology of VLSI grows towards higher speed, smaller feature size and larger chip size, the performance of synchronous circuit is limited due to its global clock approach.

The main reason of the limitation is the clock skew problem [1][2]. Clock skew is the difference in the arrival time of clock signal at different parts of the circuit. As the chip size gets larger, it is difficult to manage the global clock signal to arrive at different parts of the design at the same time. Also, as the clock speed becomes higher, the global clock period becomes shorter and thus the transition time needs to be shorter comparing with the clock period. However, the transition time can only be reduced to a limited extent. As a result, the operating speed is forced to slow down so as to accommodate the problem.

In addition, frequency of the global clock is also restricted by the slowest part of the whole design. The period between two consecutive active clock edges must be long enough for all computations to be completed before latching the result. As a result, the clock period is determined by the slowest stage such that every stage is given enough time to fully process a data, thus yielding a worst-case performance.

It is believed that by eliminating the restrictions, a design can reach a higher level of performance, and this is the motivation of the development of asynchronous circuit.

1.2 Asynchronous Design

The main difference between the synchronous and asynchronous design is the use of the global clock and the local handshake signals. In asynchronous design, operations on a functional unit are controlled by the communications between neighbouring units. When there is an event occurred on the communication wire, an operation will be started or stopped by the triggering of the event.

Since the global clock signal is removed, there is no clock skew problem existed in the asynchronous design. Also, without the restriction of the global clock signal, different parts in an asynchronous circuit can operate at their own intrinsic speeds and thus the average-case performance can be achieved rather than worst-case performance in the synchronous circuit. Therefore, the problems of the synchronous design can be eliminated and higher speed can be achieved in asynchronous design. In addition, asynchronous design offers other potential advantages such as low power consumption, automatic adoption to physical properties, high modularity and less electromagnetic emission, these make the asynchronous design attractive.

Despite of all the potential advantages motivating the development of asynchronous circuit, it has yet to achieve widespread use. This is because asynchronous circuit suffers from several problems as well.

The major problem of the asynchronous circuit is hazards [1]. In the synchronous design, hazards can be easily removed by adding more registers or slowering the clock rate. However, designers of the asynchronous circuit must remove all hazards to prevent incorrect operation. At the same time, there are little supports from CAD tools, design automation and optimization of the asynchronous design has still not been fully achieved. As a result, extra attention and extensive simulations are required and thus the development cost is increased.

Furthermore, an additional handshake circuitry is required in asynchronous design in order to handle the communication signals. This circuitry is usually complex and leads to a larger area in asynchronous design. Also asynchronous circuit generally requires extra time for handshaking protocol and thus an operation requires more time to be completed due to the communication overhead. As a result, the expected average-case performance is not fully realized. These two reasons cause an asynchronous circuit running at a speed even slower than the synchronous circuit.

Due to the maturity in synchronous design methodologies and the difficulties of asynchronous design as mentioned above, designers still prefer synchronous design in most of their system development today.

1.3 Discrete Cosine Transform

The Discrete Cosine Transform (DCT), proposed by Ahmed et. al. in 1974 [3], and its inverse (IDCT) have become an important tool for image and video signal processing applications due to their adoption in standards such as CCITT H.261 [4] for video telephony and teleconference, JPEG (Joint Photographic Experts Group) [5]for colored still image transmission and MPEG (Moving Picture Experts Group) [6] for moving pictures on the storage media. The advantages of DCT are that its performance closes to the optimal Karhunen-Loeve transform (KTL) for highly correlated signals and the existence of the fast algorithms [7][8][9] which reduce the number of operations.

The role of DCT is providing a data compression on the picture while a reasonable quality can still be maintained. It helps to reduce memory size and transmission bandwidth in the image and video applications. DCT basically involve additions and multiplications. The operation of 1D N-point DCT and IDCT can be described by following equations,

$$DCT: \quad Y_n = \frac{1}{2}c(n)\sum_{i=0}^{N-1} x_i \cos\frac{(2i+1)n\pi}{2N} \qquad \text{- Equation 1.1}$$

$$IDCT: \quad x_i = \frac{1}{2}\sum_{i=0}^{N-1} c(n)Y_n \cos\frac{(2i+1)n\pi}{2N} \qquad \text{- Equation 1.2}$$

$$where \quad i, n = 0, 1, \ldots\ldots, N-1$$

$$c(0) = 1/\sqrt{2} = 1 \ for \ i \neq 0$$

In recent year, the increasing demand of high image and video quality signal, such as MPEG-2 and High Definition Television (HDTV), requires higher and higher computation in signal processing. To meet with the real-time computation requirement, a processor which rapidly computes DCT has become a key component in image compression VLSI.

1.4 Motivation

Up to now, most of the past asynchronous circuits are not good in performance in terms of speed. Together with the difficulties discussed in the previous section, it discourages the development of asynchronous design. However, there are methods exist so that full performance potential of the asynchronous design can be realized. The worse speed performance of the asynchronous circuit is mainly due to the complicated handshaking circuitry and slow communication protocol. It is believed that by developing a new asynchronous architecture having simpler handshaking circuitry, more aggressive handshaking protocol and together with a careful circuit arrangement, the hazard can be removed and a competitive asynchronous design can be obtained. This is the motivation of this project.

DCT is chosen for the realization of a new asynchronous architecture. This is because digital signal processing (DSP) algorithm is suitable to be implemented by asynchronous technique as the process is data-dependent that fits the style of the asynchronous design. Among various DSP algorithms, DCT is a widely used algorithm in many image and video applications and high throughput is required. It helps to demonstrate the practicality of the new asynchronous architecture and the fulfillment of the requirement of image and video applications today.

1.5 Organization of the Thesis

This thesis is organized into eight chapters. The first chapter describes the background of the asynchronous design, Discrete Cosine Transform, and the motivation of this project. The second chapter introduces the basic operation and past methodologies in the asynchronous circuit design, and the new asynchronous pipelined architecture is presented at the end of this chapter. In chapter 3, various methods and algorithms of DCT implementation and two different approaches of the asynchronous implementation of DCT processor are described. Since dynamic logic is employed in the new asynchronous pipelined architecture, a new technique of operating dynamic logic in low frequency is presented in chapter 4. Chapter 5 describes the detailed architecture of the programmable DSP asynchronous processor, and the DCT implementation is given as well. Chapter 6 presents another implementation of DCT on a dedicated DCT processor. The architecture and flow of operations on the processor, and the design of the transpose memory are all provided. In chapter 7, all the implementation results and performance of the designs proposed in this thesis are given. Based on the results, the performance comparisons, discussions and suggestions are also provided in the same chapter. Finally, conclusion of the thesis is given in the last chapter.

Chapter 2

Asynchronous Design Methodology

2.1 Overview

The operation of an asynchronous circuit is not based on the global clock signal, which is used in the synchronous circuit, but on its local handshake signals. The handshake signals are the controlling signals in the communication between the sender and receiver. For most of asynchronous circuits, they usually make use of similar handshaking protocol involving requests and acknowledgements.

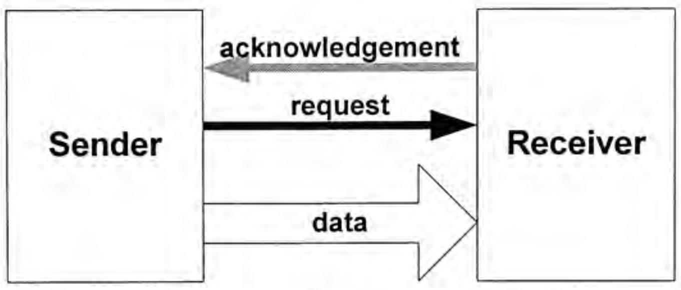


Figure 2.1 – Communications between sender and receiver in an asynchronous circuit

Figure 2.1 shows a basic communication interface in asynchronous circuit. This kind of communication style is called the bundled data approach [1][10]. In this approach, the interface between sender and receiver consists of a bundle of data which carries information (using one wire for each bit) and two control wires. When the data from the sender side is ready, a transition will occur on the request wire to inform the receiver, and acknowledgement wire from the receiver to the sender carries a transition when the data has been processed. Also the data will be maintained constantly during the receiver's active phase preventing wrong operation.

There are many types of handshaking protocol and different kinds of circuit for implementing this asynchronous communication interface. In this chapter, a brief introduction to different handshaking protocols will be given. In addition, some of the past designs and the micropipeline structure will be introduced. At last, the new asynchronous architecture will be presented.

2.2 Background

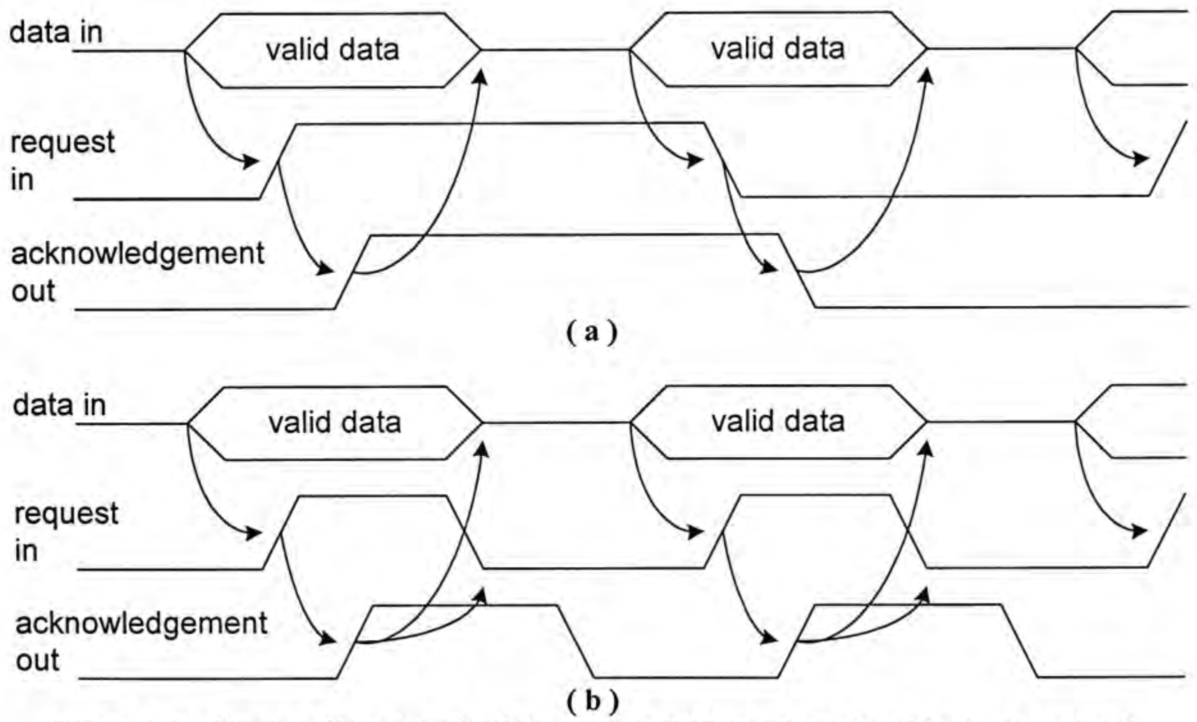


Figure 2.2 – Timing diagram of (a)two-phase, (b)four-phase handshaking protocol

There are 2 classes of handshaking protocol, one is the two-phase and the other is the four-phase [1][10][11] and their timing diagram is shown in Figure 2.2. Two-phase handshaking protocol means that any transition in the handshake signal represents an event occurred. Different from the two-phase, the four-phase handshaking protocol is a level-triggered protocol. The occurrence of an event is represented by an active level, and the return to non-active level is required after the event has been finished. In general, the two-phase handshaking protocol has better performance than the four-phase one as it makes use of all transitions of the signal to represent an event, it leads to a faster communication rate.

Compared to the synchronous circuit, the request and acknowledgement signals are additional signals. As a result an extra control circuit is required in asynchronous design so as to handle these two signals, and usually this circuit is called handshake control circuit or handshake cell.

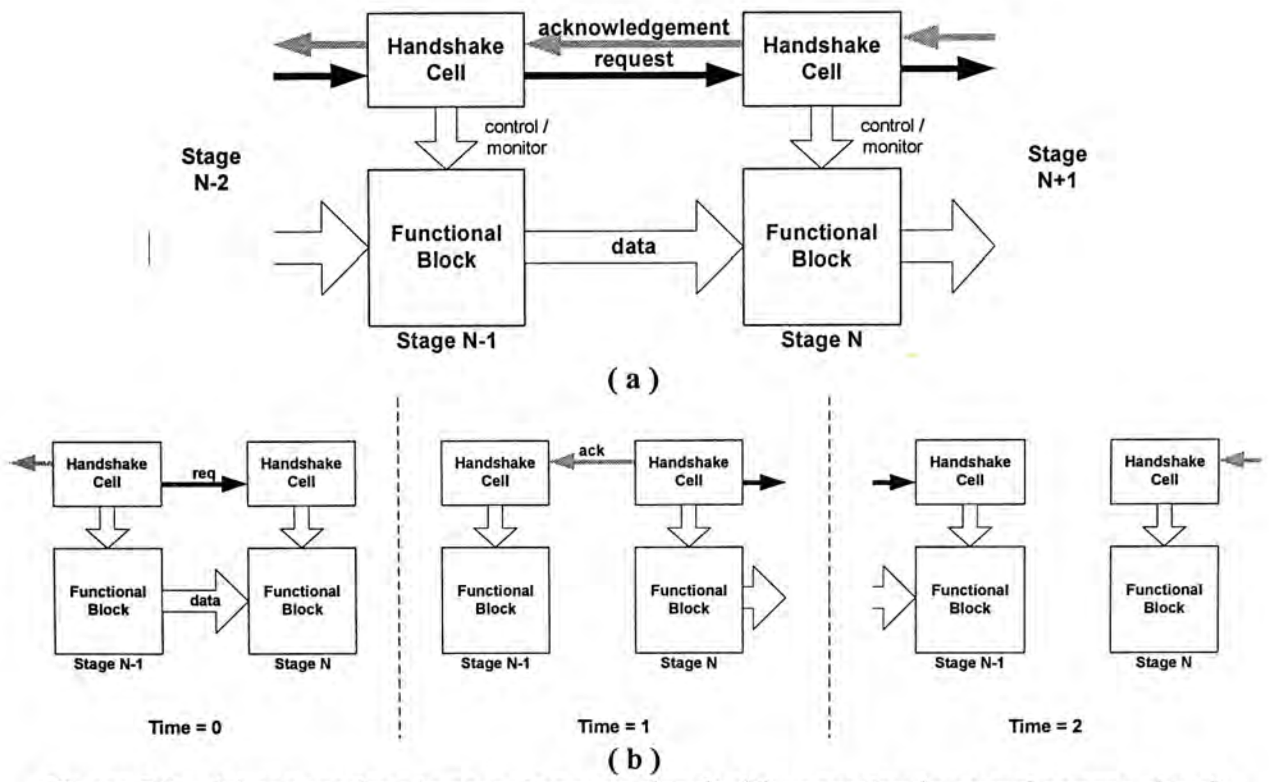


Figure 2.3 - (a) connections in asynchronous circuit, (b) operation in asynchronous circuit

Figure 2.3(a) shows the basic connection in asynchronous design using the handshake cell. In this connection, the operations depend totally on the handshake signals, and that can be explained with the help of Figure 2.3(b). Initially when the operation of functional block in stage N-1 is completed, the output data will be passed to the functional block in stage N. At the same time, the handshake cell in stage N-1 will detect the completion of computation and generate a request signal for stage N. This request signal is used to indicate that the operation of stage N-1 is completed, and the output data of stage N-1 is held and ready for the stage N to process. Starting from this moment, stage N-1 needs to hold the output data until stage N finishes the computation.

The handshake cell in stage N detects the request signal from the previous stage, and then allows the functional block in stage N to process the data. After the computation is completed, the handshake cell in stage N will generate two signals. The first one is the acknowledgement signal which is used to inform stage N-1 that the data has been processed. As a result, stage N-1 becomes idle and wait for the data from stage N-2 for the next operation. The second signal is the request signal to the stage N+1 for further processing of data.

This communication interface and protocol exist between all the stages and its neighbouring stages in the asynchronous circuit. Since all the operations are controlled by the handshake signals, the performance of the handshake cell becomes the main factor of determining the speed of the asynchronous circuit.

2.3 Past Designs

The design of the handshake cell and the use of the handshaking protocol are important as they determine the throughput and latency of the whole asynchronous system. For the handshake cell, an accurate detection of the completion of the operation and a quick generation of the request signal are the most important issues as they are used to guarantee the circuit operating correctly and quickly. If the request signal is generated before the functional block finishes its computation process or before data is valid, hazard will occur as incorrect data will be latched by the next stage and incorrect result will be obtained. If the request signal is generated a long time after the end of computation, it is secure to have a correct output but the whole circuit will be slowed down. However, to generate the request signal just in time while maintaining simple structure is really a difficult task. By using a suitable handshake cell, the complexity of the handshaking protocol can be reduced and thus, the communication time can be reduced too. As a result, the speed and performance of the whole circuit can be enhanced.

In the past decades, there were many kinds of handshake cell developed [12][13][14][15]. And the most famous and commonly used one is the C-element. C-element is firstly introduced by D.E. Muller in 1956 [16]. It is a rendezvous element, or an event-driven element. Figure 2.4 shows the symbol and 2 different CMOS structures of the C-element.

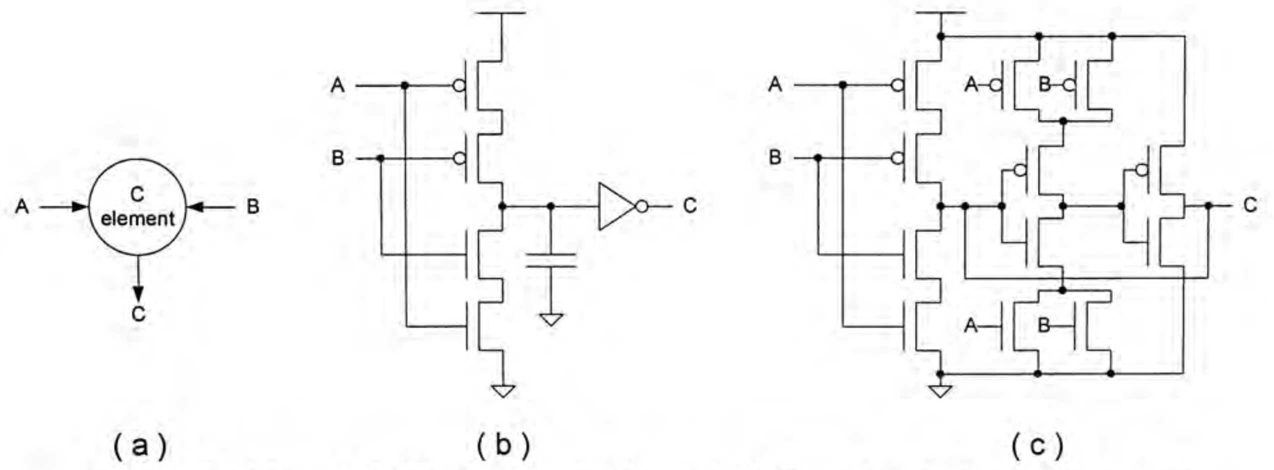


Figure 2.4 – (a) symbol of C-element, (b) dynamic C-element and (c) static C-element

The operation of the C-element is that, when both inputs are the same, then the data will be copied to the output, else the previous output will be maintained. Therefore, the output will only be toggled when there are events occurred at the both inputs of the C-element.

C-element is usually incorporated in the two-phase handshaking protocol with the bundled data approach. In applying the C-element in the asynchronous circuit, the input *A* and *B* are served as the inputs of request or completion signal from previous stage and acknowledgement signal from next stage. The output *C* has 3 functions. The first one is to control the operation of the function block. The second one is acted as the acknowledgement signal which is sent back to the previous stage, and the last one is acted as the request signal sending to the next stage. A more detailed operation of C-element in asynchronous circuit will be discussed in the next part.

2.4 Micropipeline

No matter synchronous or asynchronous design, pipeline is an important methodology to improve the performance of a circuit or system. The principle of the pipeline is to divide a single operation into several sub-operations, and allows them to operate simultaneously [10]. For the asynchronous circuit, pipeline can be done by breaking down the complex functional block into several simpler functional blocks, and each of them is governed by a dedicated handshake cell. The widely known pipeline methodology in asynchronous circuit is micropipeline.

Micropipeline was introduced in Ivan Sutherlands' Turing Award lecture [10] primarily as an asynchronous alternative to synchronous elastic pipelines. From the definition by Ivan, micropipeline means a simple form of event-driven elastic pipeline with or without internal processing.

The basic operation of the micropipeline can be explained by the control first-in-first-out (FIFO) sequence structure as shown in Figure 2.5. The control FIFO sequence is operated in two-phase handshaking protocol. Assuming that all the wires are initially set at zero, when there is a transition in the request input, then output of the first C-element will be changed from zero to one. This transition will be sent out of the control sequence as an acknowledgement signal, and also will be propagated to the input of the second C-element. Since the input is toggled, same situation will occur in the second C-element, as well as the third C-element. As a result, the request signal passes through all the C-elements in series, and emerges on request out.

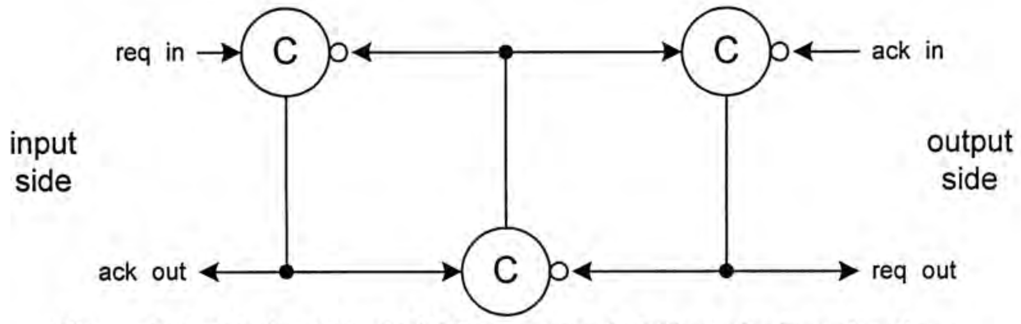


Figure 2.5 – Basic control FIFO sequence in Micropipeline structure

However, when there is another request signal coming from the request input, this new request signal may not be emerged on the request out this time. This is because the control FIFO sequence may still not received the acknowledgement from the output side, as a result no transition has been made in the acknowledgement input terminal and thus the output of the third C-element cannot be toggled. However, this phenomenon is normal as no transition on acknowledgement input means that the output side, or the recipient side, still has not processed the previous request, the new request should not pass to it before the previous event is completed.

Figure 2.6 shows the block diagram of the Sutherland's micropipeline system. The connections are actually similar to the previous FIFO sequence, but a storage element and a logic block are included in each stage. The storage element used is called Capture and Pass latch (CP latch), which is an event-controlled storage element. The inputs *C* and *P* are responsible for controlling the capture and pass function, and the outputs *Cd* and *Pd* are just simply the delayed version of the inputs *C* and *P* respectively. In this micropipeline structure, when there is a transition occurred in the request input, data will be captured and stored in the CP latch. However, the stored data will not be passed out from the output of the CP latch until there a transition occurs at input *P*. If the CP latch in the next stage has captured the previous data, the phase of acknowledgement signal will be changed and passed back to the first CP latch. Then the first CP latch will pass the stored data to the logic block to perform the logic operation. This operation will be repeated when the next request signal arrives. The delay element is used to delay the arrival of the request (capture) signal to the next stage so as to ensure the logic operation have been completed, therefore it needs to be the worst-case delay of the corresponding logic block.

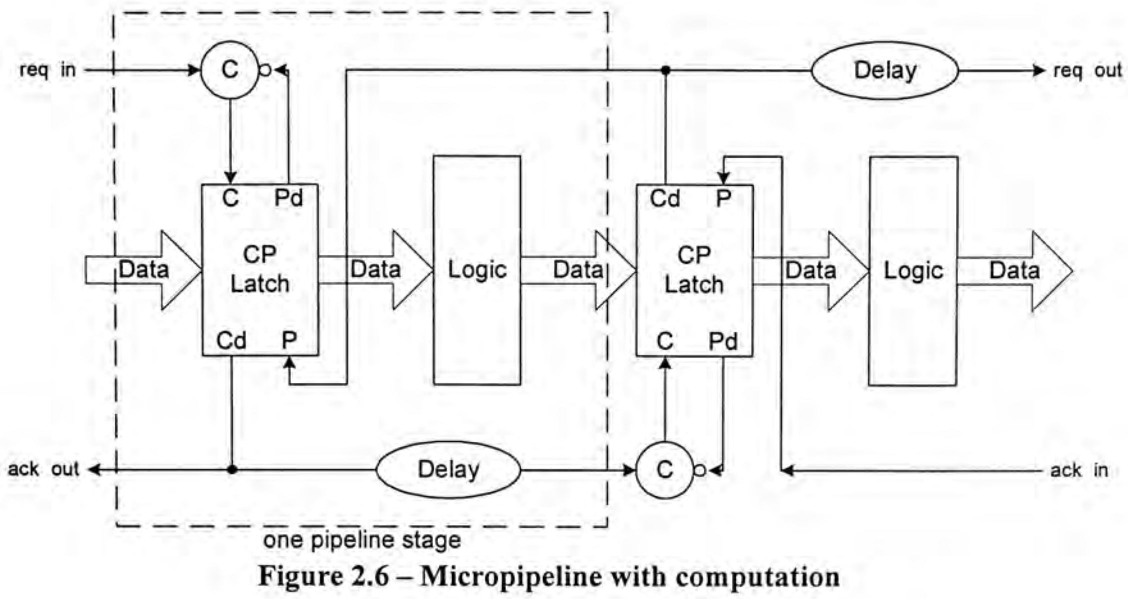


Figure 2.6 – Micropipeline with computation

There are several benefits of using the micropipeline structure. First, the architecture is simple and effective, it is easy to implement and a good throughput can be easily achieved. Also, the latches moderate the flow of data through the pipeline and can be used to filter out hazards. Thus, any logic structure can be used in the logic blocks, including the straightforward structures used in synchronous designs. At last, micropipeline is automatically elastic [10], data can be sent to and received from a micropipeline at arbitrary times.

Although micropipeline is a powerful implementation strategy which elegantly implements elastic pipelines, it delivers worst-case performance in each stage by adding delay elements to the control path to match with the worst-case computation time of the corresponding function block. Besides from this, the circuit of its CP latch is rather complicated, and delays are added on the capture and pass signal to make sure the data has been latched. Therefore the performance is degraded.

2.5 New Asynchronous Architecture

As previously discussed, although Micropipeline is a powerful and widely used methodology in the asynchronous circuit design, it still has some areas for improvement.

The first improvement from the micropipeline is the use of dynamic logic, and in our design, domino logic [18] is used. Domino logic is one of the logic types in the dynamic logic family, and its basic structure is shown in Figure 2.7. The logical function of the domino logic is characterized by the nMOS logic block. There are two phases for the operation of the domino logic, one is the Precharge phase and the other is the Evaluation phase. When the clock signal is low, then the domino logic is in the Precharge phase. At this moment the output must be low as a pull-up path is connected to the floating node. When the clock signal is high, then it is in the Evaluation phase and the output depends on the input data. If the input data creates a pull-down path in the nMOS tree, then the floating node will be discharged and the output will go high. Otherwise the output will be kept in low.

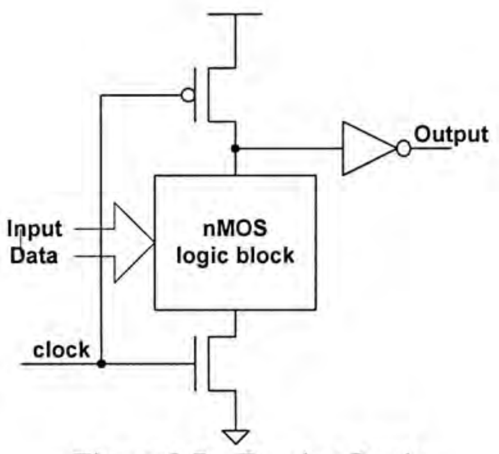


Figure 2.7 – Domino Logic

The advantage of the dynamic logic is that it has lower processing delays and more compact in size in comparison to conventional CMOS data-paths. Due to these, many asynchronous circuits [11][19][20][21][23][25] also adopted the dynamic logic in their micropipeline design. However most of them have not utilized all the functions of dynamic logic. One of the interesting properties of the dynamic logic is its ability of temporary storage [17][19]. Dynamic Logic is actually a combination of the logic and storage elements, the output data can be held even though the input data have been changed under some conditions. As a result, the complex CP latch in the micropipeline can be omitted if the dynamic logic (domino logic in our case) is used. This implementation of dynamic logic in asynchronous circuit has been proven by Renaudin et. al. [17], and its pipeline structure is shown in Figure 2.8. In this architecture, the completion detection is no longer relied on the worst-case delay, it is done by a dedicated circuit. It monitors the output of the logic block and provides a faster and accurate response when the output is ready. Although dynamic logic brings benefits for the asynchronous circuit, it introduces other problems of charge leakage and charge redistribution. These problems limit the dynamic logic to have a minimum operating frequency from preventing the logic error. As a result, extra attention must be paid in using dynamic logic. A further discussion on this problem and some possible solutions will be given in chapter 4.

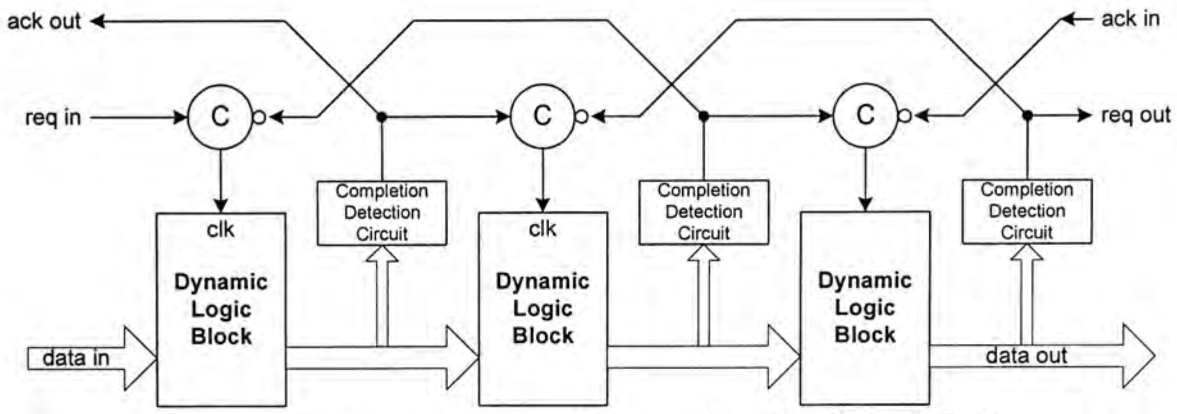
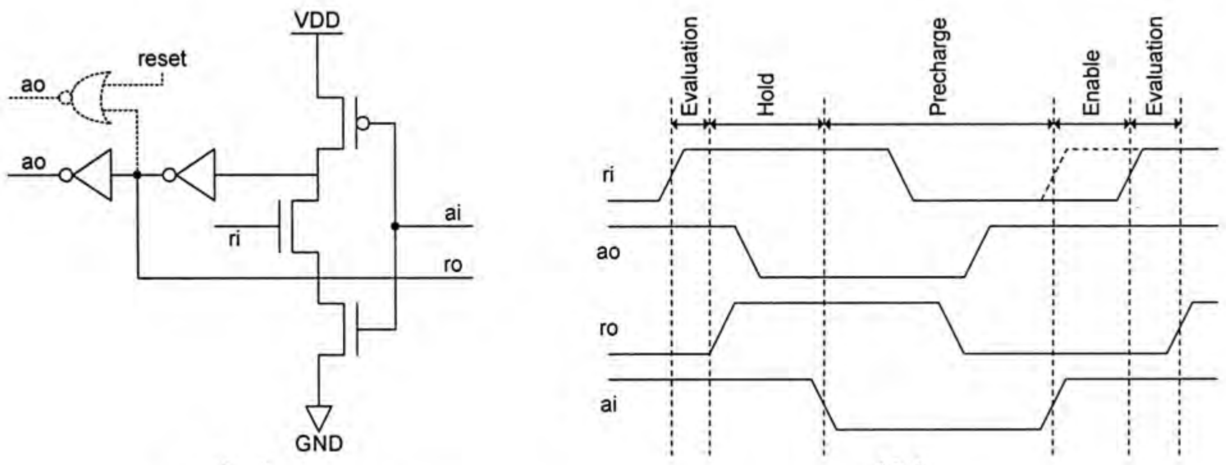


Figure 2.8 – Asynchronous architecture by using dynamic logic

Besides from the dynamic logic, another improvement is on the handshaking protocol and handshake cell. Referring to the previous implementation shown in Figure 2.8, a very restrictive handshaking protocol is used to guarantee secure operation of the asynchronous pipeline. For a certain stage in this pipeline architecture, a new operation, either precharge or evaluate, can only be carried out when both the previous and next stage finished their current operation. This strict protocol limits the performance of the handshake signal.

In the new asynchronous architecture, some improvements on the protocol have been made. First a current stage is allowed to go into the Evaluation phase when the next stage goes into the Precharge phase, i.e. no need to wait for the precharge confirmation from the Precharge phase. Second, a current stage is allowed to finish the Precharge phase even the previous stage is still in Evaluation phase. This introduces a flexible "Enable" period between the Precharge and Evaluation period. In order to carry out this new handshaking protocol, a new handshake cell is used and it is shown in Figure 2.9(a).



(a) (b)
Figure 2.9 – (a) new handshake cell, (b) timing diagram of a pipeline stage

The new handshake cell is also in domino style. Compared with the classical architecture, this handshake cell is faster due to its simplicity, low input capacitance from the request and using simple transistor in pull-up. In this new structure, the handshake cell and the domino logic cell will enter the Precharge phase and Evaluation phase respectively at the same time. As a result, the handshake cell can be seen as a logic element of the pipeline stage and the throughput of the system can be minimized [30]. The handshake cell can be easily modified to receive more than one request signal by connecting more nMOS transistors in series in the nMOS tree, which is similar to the dynamic AND structure. The difference in speed will be more significant in logically joining handshake signals as the classical C-element with many inputs is very slow.

One of the disadvantages of this handshake cell is the requirement of the four-phase handshaking protocol which requires longer communication time. However, this four-phase fits the operation of dynamic logic as the non-active phase can be used for the precharge of the dynamic logic.

Based on the new handshake cell, the operation of this new asynchronous pipeline architecture can be divided into 4 phases: Evaluation, Hold, Precharge and Enable. The timing diagram is shown in Figure 2.9(b). In the Evaluation phase, the current stage processes the data, which is valid at the input. After the current stage has finished its process, it will enter the Hold phase. In this phase, the input data may become invalid but the output should be held for the process in the next stage. After that, the stage will enter the Precharge phase, and will enter the Enable phase afterwards. In this phase, the stage is waiting for the valid data appearing at the input. This phase can be omitted when the valid input data has already appeared during the Precharge phase. Since all the handshake cells and logic cells should be precharged first during the power up, a NOR gate will be used, as shown in Figure 2.9(a), in the handshake cell. In this configuration, one of NOR gate inputs connects to the Reset signal thus that the all the cells in previous stage can be precharged initially.

Figure 2.10 shows the connection and the flow of the pipeline operations of this new asynchronous architecture. When data arrives, the current stage will enter the Evaluation phase to process the data. Afterward, it will enter the Hold phase to hold the data for the next pipeline stage to process. At this moment, it will send a request signal to the following stage and acknowledgement signal to the previous stage.

After the following stage has processed the data, the current stage enters the Precharge phase. And at last it will enter the Enable phase to wait for a new data from the previous stage.

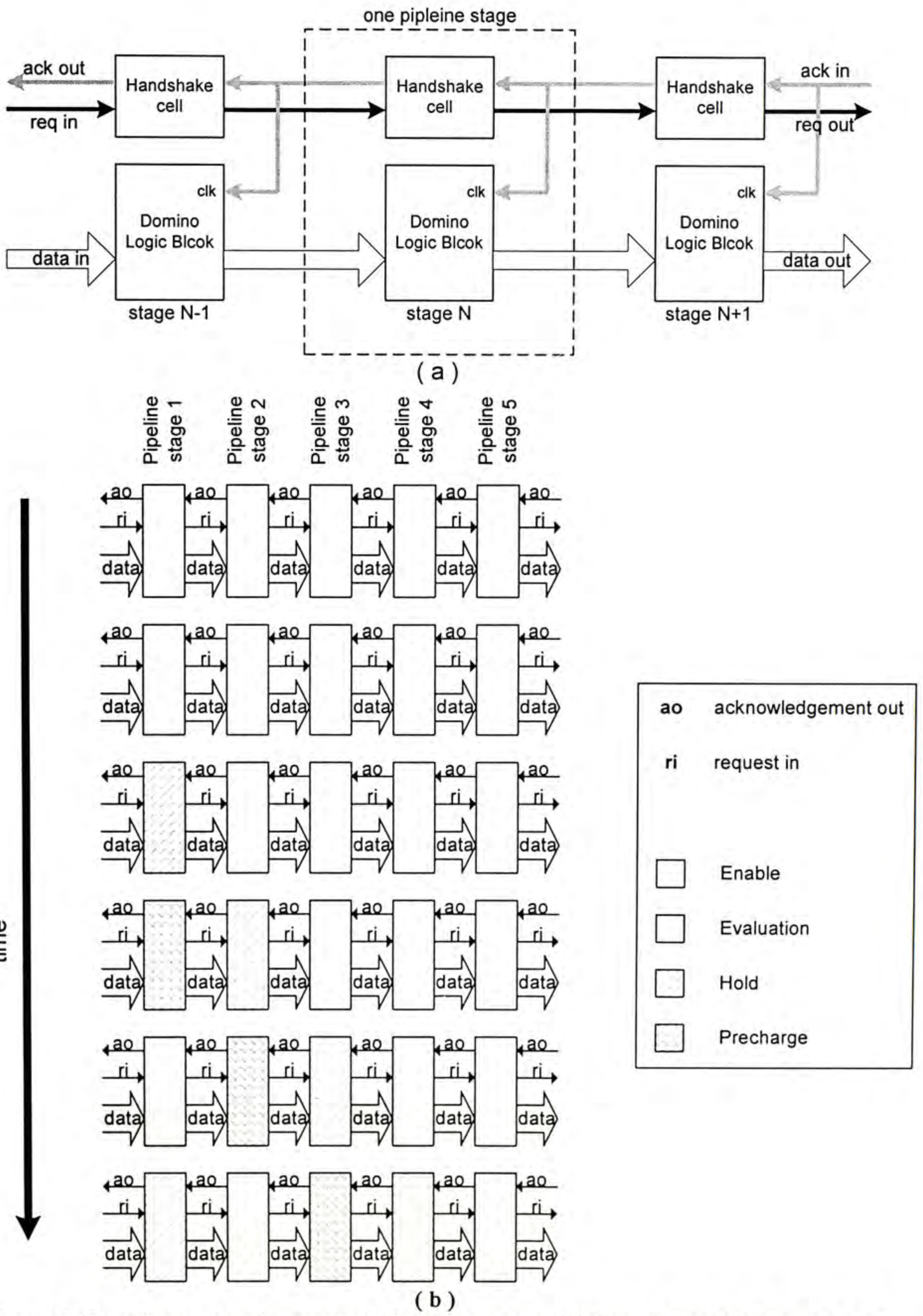


Figure 2.10 – (a) new asynchronous pipeline connection, (b) flow of operations in the new asynchronous pipeline

The use of Differential Cascode Voltage Switch Logic (DCVSL) [24], a type of domino logic, can also improve the speed of the circuit. Figure 2.11 shows the basic structure of a DCVSL cell. Its operation is similar to that of the domino logic. In the Precharge phase, both of the true and complementary outputs will be kept at low. When in the Evaluation phase, the computation is enabled. Due to the complementary structure of the nMOS logic blocks in DCVSL, one and only one of the outputs will go high.

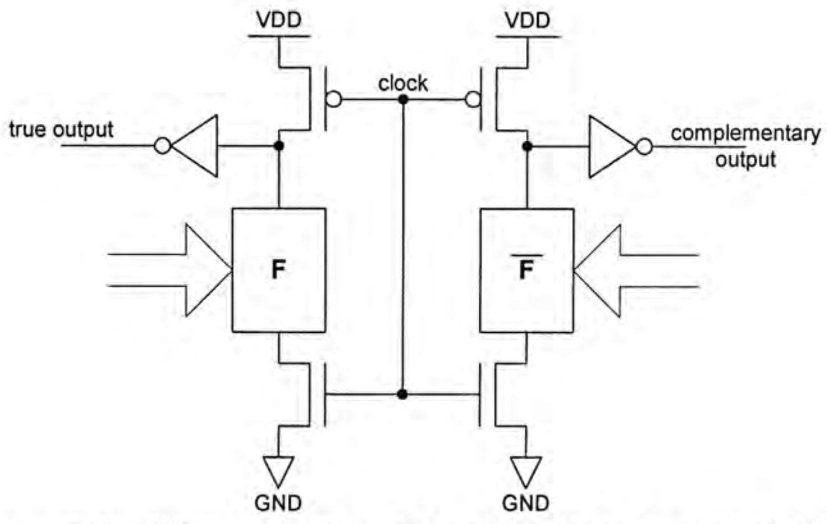


Figure 2.11 – Differential Cascode Voltage Switch Logic (DCVSL)

There are benefits of using DCVSL in asynchronous logic. First, it is based on the structure of the domino logic and thus it has the benefits of domino logic, namely, are fast computation time and storage property. Second, the DCVSL provides dual rail coded data which provides a very reliable completion signal by simply OR-ing both the outputs as shown in Figure 2.12. Due to these, DCVSL is an attractive way to implement asynchronous operation functions [26][28][29] and has been used in many asynchronous designs [17][22][26][27].

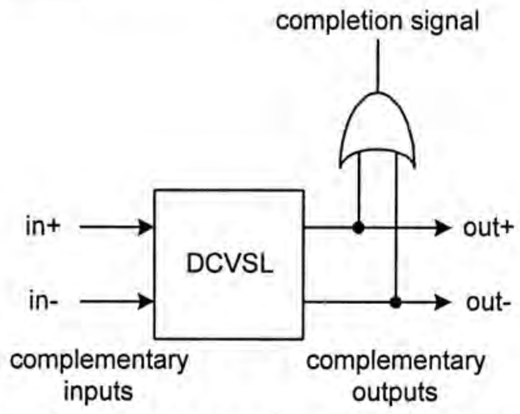


Figure 2.12 – Completion signal generated from the DCVSL

Although this way to generate completion signal is very simple, one gate delay is still added after the completion. In fact, the completion of the computation can be detected directly without the OR gate by modifying the handshake cell. Figure 2.13(a) shows the modified CMOS structure of the new handshake cell. In this new structure, the true and complementary outputs from the DCVSL block can be directly connected to the handshake cell for the completion detection. As a result, the OR gate and the request signal can be eliminated, and the completion detection matches closely the original computation time of the DCVSL block, and thus the average case performance can be achieved. Figure 2.13(b) shows an example of the single bit basic FIFO cell with the modified handshake cell and Figure 2.13(c) shows the new connection of the asynchronous pipeline by using the new handshake cell.

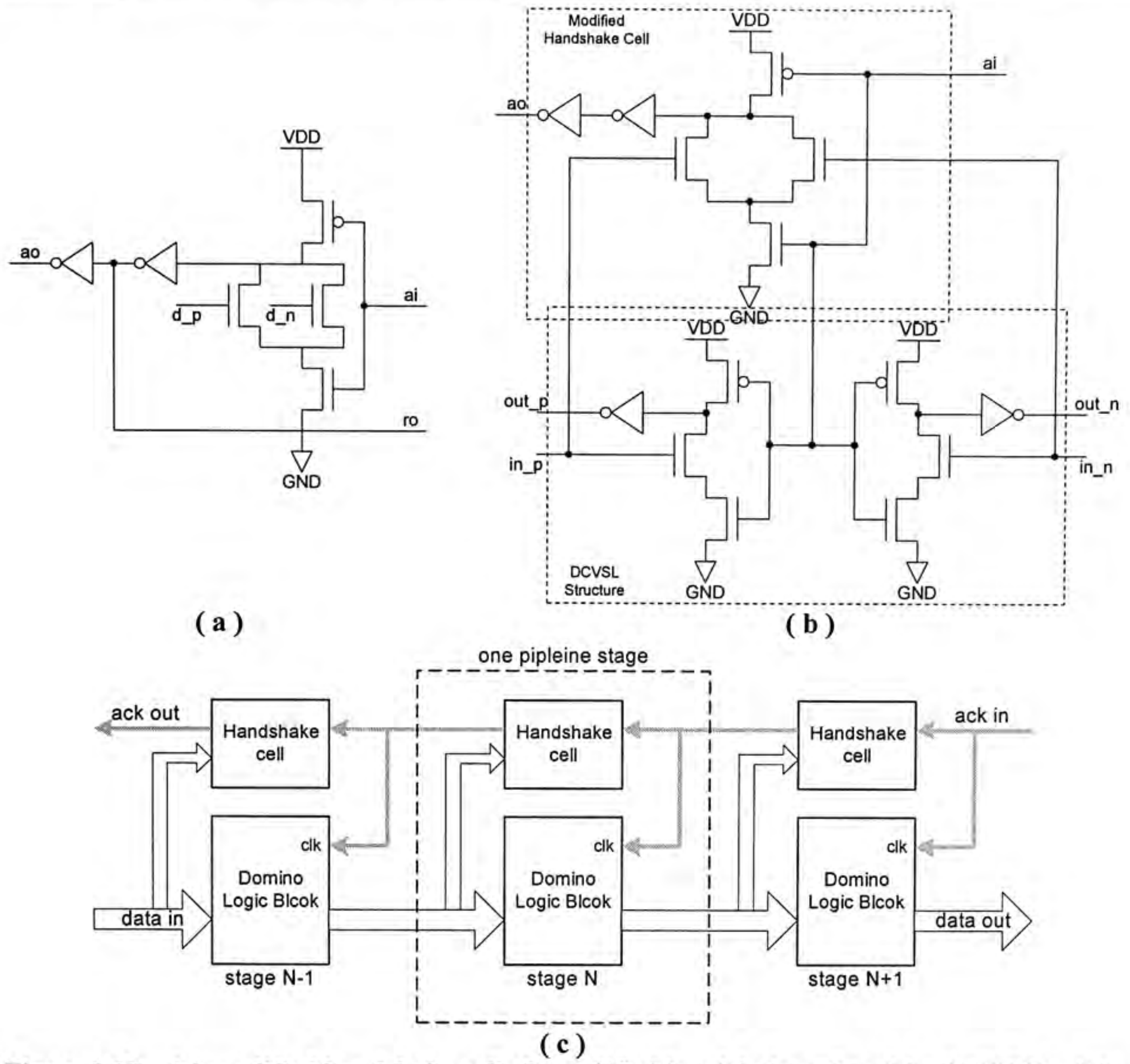


Figure 2.13 – (a) modified handshake cell, (b)modified handshake cell and basic FIFO cell in DCVSL structure, (c) connection of the asynchronous pipeline

The use of DCVSL will improve the speed as the communication protocol is simpler, but the trade-off is the size penalty incurred by DCVSL. Moreover, dual-rail data requires large routing area in the physical layout as the bus width is doubled. Therefore within the processing units, DCVSL is used in order to maximize the performance. On the other hand, in each connection between all the processing units, where they may be separated quite far away in the physical layout, a dual-to-single or single-to-dual rail conversion interface is inserted so as to reduce the routing area by using single-rail data.

Chapter 3

DCT/IDCT Processor Design Methodology

3.1 Overview

Most digital signal processing (DSP) algorithms involve many mathematical operations which require high computational resources. There is no exception for the Discrete Cosine Transform (DCT) [3]. Although there are arithmetic units within the general purpose micro-processor or micro-controller, they are not specifically designed for the pure mathematical operations. As a result, the implementation of DSP algorithm on them may not be efficient and has poor performance. Due to this, it motivates the development of the DSP chip, and the DCT chip in this thesis.

There are many hardware architectures to implement the DCT algorithm, such as using a programmable DSP processor, or dedicated ASIC. At the same time, there are many kinds of DCT algorithms, some of them focus on reducing the number of operations, some of them allow more regular architecture of VLSI implementation. Careful analysis on these is required in order to find out a most suitable combination for the DCT implementation in asynchronous circuit.

The advantage of using asynchronous technique to implement the DCT or other DSP processors is its average case performance. There are many functional blocks in the design, and their computational time may differ from each other a lot. The global clock frequency in synchronous circuits is governed by the worst case delay in the whole system whereas each functional block in asynchronous circuits by its own operation speed. As a result the computation time of an asynchronous DSP chip may be shorter than the synchronous one.

In this chapter, different hardware architectures and DCT algorithms will be considered and compared.

3.2 Hardware Architecture

Different from the general purpose micro-processor or micro-controller, DSP processor has traditionally been optimized to compute different arithmetic operations, such as the convolution, recursive filtering and fast transform operations that typically characterize most signal processing algorithms. They are used in many application areas such as communications, speech and video/image processing. As mentioned in the previous part, DSP processor can be either programmable or of a dedicated nature.

Programmable DSP processor has the advantages in the flexibility and design time for different algorithms as it allows the implementation of a variety of DSP algorithms. Besides from arithmetic units, extra memory and control units are required in order to store the application programme and control the operations of data. The performance of the DSP algorithm is not only depended on the hardware, but also depended on the application programme. Therefore the application programme should be optimized for utilizing the hardware in the processor.

On the other hand, the dedicated ASIC is hardwired to perform a specific algorithm, and usually no extra control or programme is required. Once it is designed, the performance of the dedicated ASIC is fixed. Although the flexibility of the dedicated ASIC can be considered to be zero, this approach is expected to perform better than the programmable approach as the DSP algorithm is optimized in hardware level, and also it is more efficient in terms of area and speed.

3.3 DCT Algorithm

The main application of the DCT is in the video or image compression. For most of the image and video applications, the whole image will not be processed with DCT directly as it will require a lot of computations. In contrast, the image will be divided into several regular blocks for processing. The block size is usually eight pixels or sixteen pixels in both of x and y direction, as shown in Figure 3.1. The reason to have a block size of 8×8 or 16×16 is that they have been found to provide sufficient details and localized activities of the picture to enable reasonable adaptive processing of the image [31]. And for most of the current DCT applications such as H.261 [4], JPEG [4] and MPEG [6], the block size of 8×8 have been recommended. Therefore the effort of hardware development has been concentrated on an 8×8 two-dimensional (2D) DCT.

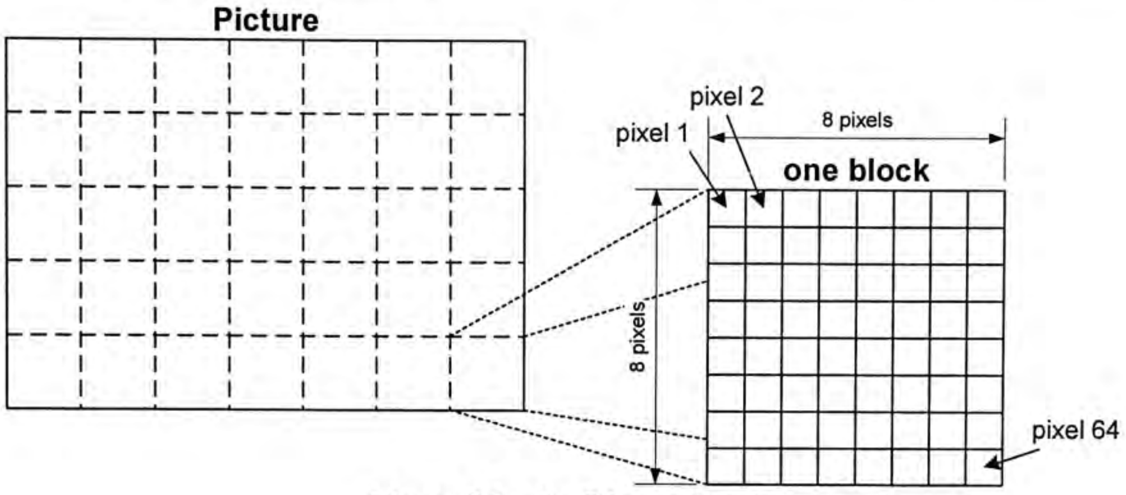


Figure 3.1 – 8 x 8 image block

In general, the *N*×*N* point of 2D DCT is given by Equation 3.1,

$$Y_{k,l} = \frac{2}{N}c(k)c(l)\sum_{n=0}^{N-1}\sum_{m=0}^{N-1}x_{m,n}\cos\left[\frac{2\pi(2m+1)k}{4N}\right]\cos\left[\frac{2\pi(2n+1)l}{4N}\right] \qquad \text{- Equation 3.1}$$
$$\text{where} \quad m,n,k,l = 0,1,\ldots\ldots,N-1$$
$$c(0) = 1/\sqrt{2} = 1 \; for \; i \neq 0$$

Since for the video or image application, the block size is 8×8, i.e. *N*=8. Therefore Equation 3.1 becomes

$$Y_{k,l} = \frac{1}{4}c(k)c(l)\sum_{n=0}^{7}\sum_{m=0}^{7}x_{m,n}\cos\left[\frac{\pi(2m+1)k}{16}\right]\cos\left[\frac{\pi(2n+1)l}{16}\right] \qquad \text{- Equation 3.2}$$
$$\text{where} \quad m,n,k,l = 0,1,\ldots\ldots,7$$
$$c(0) = 1/\sqrt{2} = 1 \; for \; i \neq 0$$

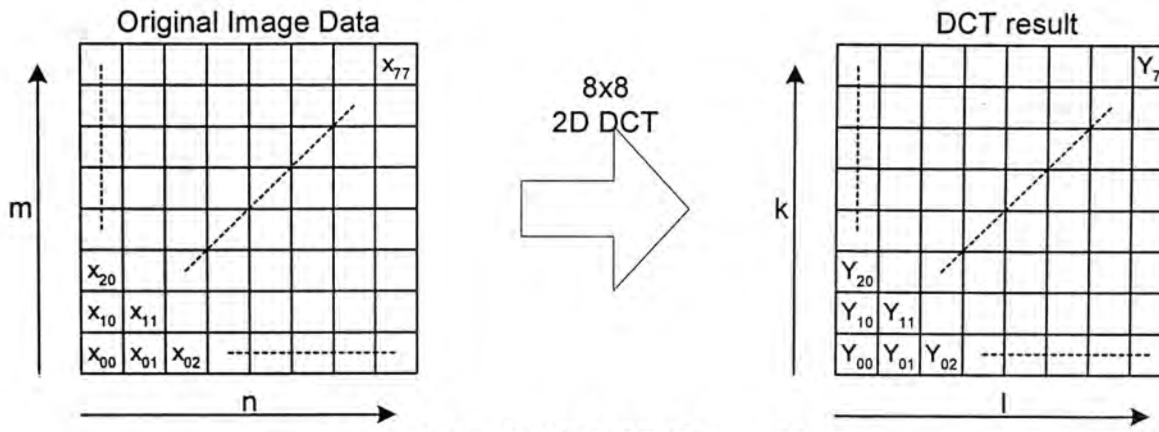


Figure 3.2 – 2D DCT of 8×8 image block

Figure 3.2 shows the 8×8 2D DCT of an image block. If the 8×8 2D DCT is directly implemented from Equation 3.2, totally 4096 (8^4) multiplications and 4032 ($8^2 \times 8 \times 7$) additions are required to calculate all the 64 DCT outputs. This number of arithmetic operations is extremely high, especially for the number of multiplication as it requires higher computational resources. It is not possible to perform the 2D transform in real-time applications even for a dedicated DSP processor. Fortunately, there are many kinds of fast 2D DCT algorithm to reduce the number of operations, and thus makes the real-time 2D DCT implementation possible.

There are two main types of fast algorithm for VLSI implementation of 2D DCT. The first type is the row-and-column decomposition method which is shown in Figure 3.3. This method separates the 2D DCT into two one-dimensional (1D) DCT operations based on the symmetry and regularity of the 2D DCT structure. The first 1D transforms are applied on the data row-wise, which is called the row operation. Afterwards, next 1D transforms are applied on the intermediate results of the row operation column-wise, and this is called the column operation. The reordering of the results of row operation into column order can be done by a transpose memory. In this way, a complex $N \times N$ 2D DCT can be decomposed into $2N$ 1D DCT operation and the number of multiplications is reduced from N^4 to $2N \times N^2$. As a result, the computational requirement is greatly reduced. A better result can be achieved by further applying the fast 1D DCT algorithm [7][8][9] in the row and column operations. Since the row-and-column decomposition method requires two 1D DCT processors and the implementation is straight forward, this method has been chosen by many other developers [32][33][34][35].

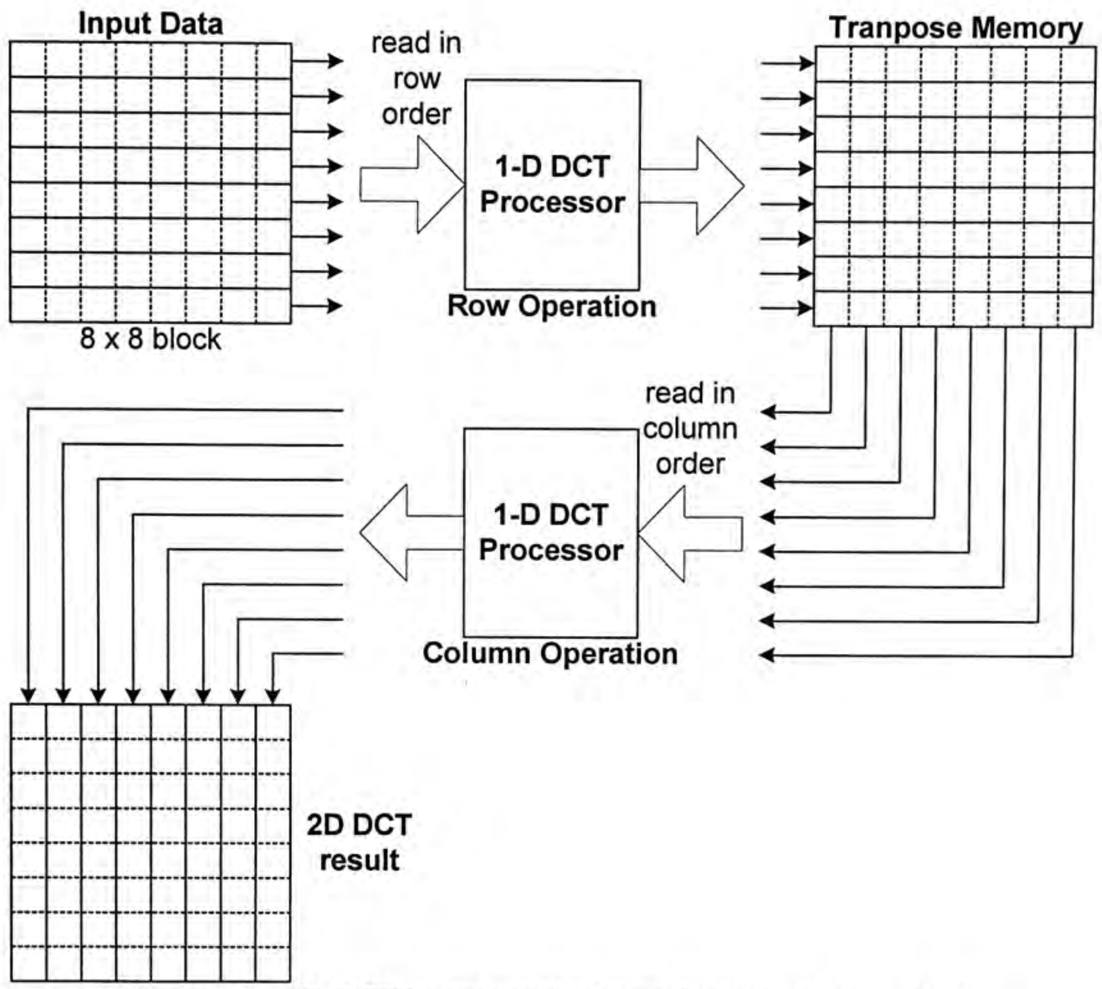


Figure 3.3 – 2D DCT by row-and-column decomposition method

The second type of the fast 2D DCT algorithm is called the direct method. This method directly uses the 2D DCT algorithm to compute 2D DCT. There are many proposed fast 2D algorithms to handle this [36][37][38]. They explore the trigonometry equality of 2D DCT such that the $N \times N$ 2D DCT can be decomposed into N 1D DCT plus some extra additions as shown in Figure 3.4, and thus the number of multiplications can be reduced to $N \times N^2$. Similar to the row-and-column decomposition method, the number of operations can be further reduced by applying fast 1D DCT algorithm on the 1D DCT processor design.

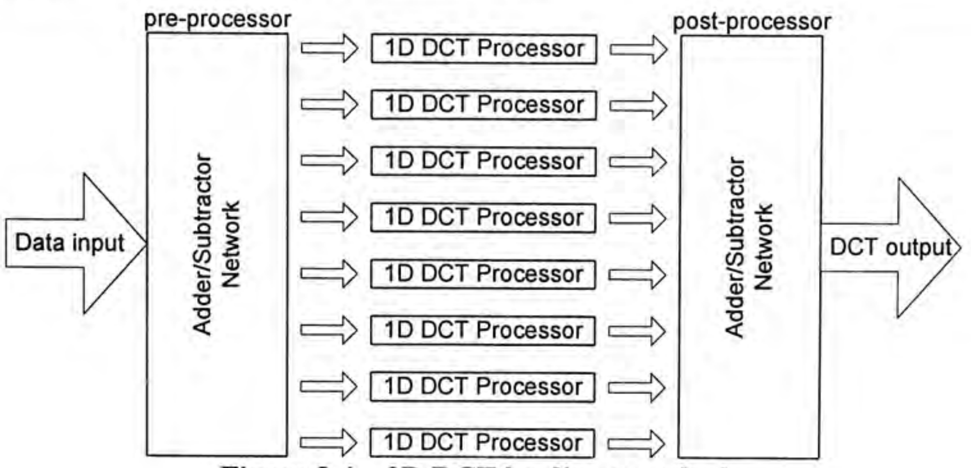


Figure 3.4 – 2D DCT by direct method

By comparing the two approaches, the 2D direct method is more superior than the row-and-column decomposition method. This is because it involves much less multiplication which directly leads to better performance. Furthermore it does not require the transpose memory. However, most of these fast 2D direct algorithms require very complex data path in the adder/subtractor network of the pre- and post-processors which cause difficulty in the VLSI implementation [37][39]. Besides the complex routing overhead, it also introduces a large handshaking overhead in asynchronous implementation. On the other hand, although the row-and-column decomposition requires more arithmetic operations, it requires only two 1D DCT processors saving a lot of hardware. Also the data path in a 1D DCT is simpler and regular which leads to an easier hardware implementation, and this favours the asynchronous implementation. Due to these reasons, the row-and-column decomposition is chosen for the implementation of the 2D DCT in asynchronous circuit.

3.4 Used Architecture and DCT Algorithm

In this thesis, two different implementations of the DCT will be shown. As previously discussed, the row-and-column decomposition is more suitable for the implementation of the 2D DCT using asynchronous technology. Therefore the following parts and chapters will be focused on the design and the implementation of the 1D DCT algorithm. For the two implementations of the 1D DCT, one is constructed based on a programmable DSP processor, and the other one is implemented as a dedicated one. The implementation of the transpose memory will be discussed in chapter 6.

3.4.1 Implementation on Programmable DSP Processor

Recalling from Equation 1.1, the 8-point DCT is given by the following equation

$$Y_n = \frac{1}{2}c(n)\sum_{i=0}^{7} x_i \cos\frac{(2i+1)n\pi}{16} \qquad \text{- Equation 3.3}$$

and its matrix representation is shown as follows

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} A & A & A & A & A & A & A & A \\ D & E & F & G & -D & -E & -F & -G \\ B & C & -C & -B & B & C & -C & -B \\ E & -G & -D & -F & -E & G & D & F \\ A & -A & -A & A & A & -A & -A & A \\ F & -D & G & E & -F & D & -G & -E \\ C & -B & B & -C & C & -B & B & -C \\ G & -F & E & -D & -G & F & -E & D \end{bmatrix} \bullet \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_7 \\ x_6 \\ x_5 \\ x_4 \end{bmatrix} \qquad \text{- Equation 3.4}$$

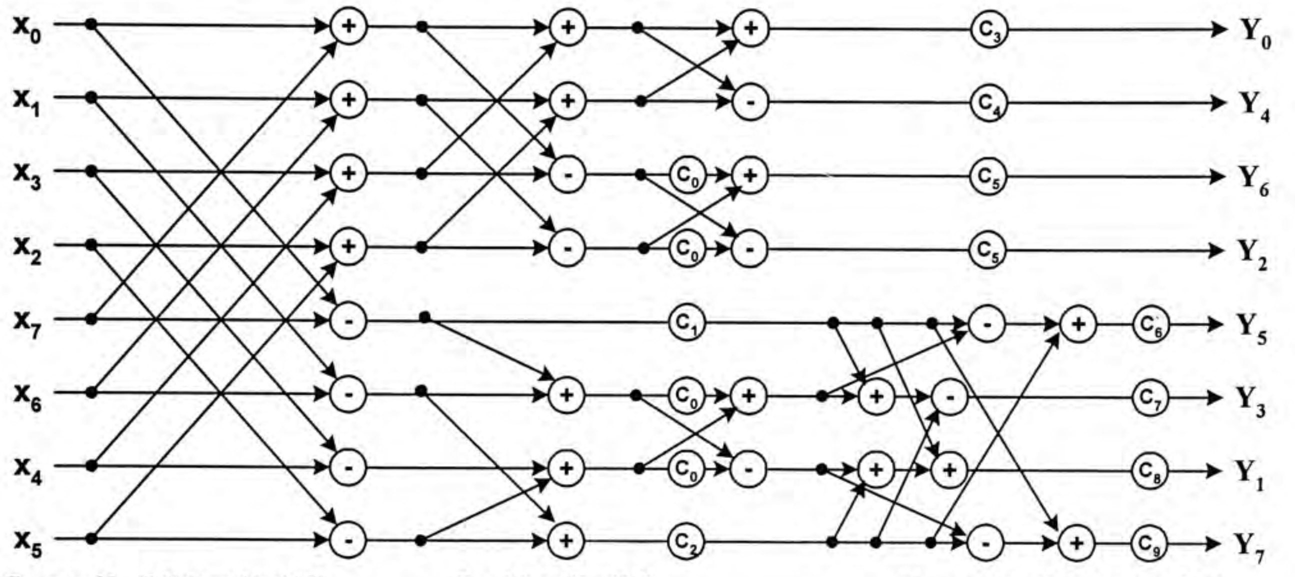
where $A = cos(\pi/4)$, $B = cos(\pi/8)$, $C = sin(\pi/8)$, $D = cos(\pi/16)$, $E = cos(3\pi/16)$, $F = sin(3\pi/16)$, $G = sin(\pi/16)$

Since the programmable DSP processor has fixed number of arithmetic units, the lesser number of operations, the shorter the computational time and thus the higher performance of the DCT implementation. Therefore a fast algorithm with smaller number of operations should be chosen for the implementation on the programmable DSP processor.

There are many kinds of fast algorithm aided to reduce the total number of operations. The most well know ones are the Lee's [7] and Hou's [8] algorithms. They both reduce the DCT operations to have 12 multiplications and 29 additions. The number of arithmetic operations is greatly reduced from the original 64 (8^2) multiplications and 56 (8×7) additions. However, these two algorithms were not chosen for the implementation of DCT in this processor because the accuracy of the DCT algorithm should also be considered.

The main error of the DCT comes from the truncation after the multiplications as the bit length of the data is increased after each multiplication. A truncation must be taken in order to match the width of data bus. As truncation on a data makes it differ from its actual value, if a data in the processor is multiplied several times continuously, it resultant value could be greatly differed from its exact value. Therefore, a fast algorithm with less multiplication stages on a data path should be chosen.

By comparing several fast algorithms, the one proposed by the Jeong et. al. [40] is chosen, and its signal flow diagram is shown in Figure 3.5. This fast algorithm requires 14 multiplications and 29 additions, and requires only a maximum of 2 multiplications in each data path. Therefore it can provide a better accuracy than Lee's or Hou's algorithms in a fixed width system.



C_0=cos(6π/16)/cos(2π/16), C_1=1/cos(2π/16), C_2=cos(4π/16)/cos(2π/16),
C_3=1/√2, C_4=cos(4π/16), C5=cos(2π/16),
C_6=cos(2π/16)/2cos(5π/16), C_7=cos(2π/16)/2cos(3π/16), C_8=cos(2π/16)/2cos(1π/16),
C_9=cos(2π/16)/2cos(7π/16),

Figure 3.5 – Signal flow diagram of the Jeong's 1D DCT fast algorithm

The detailed architecture of this programmable DSP processor and the implementation of the 1D DCT will be discussed in chapter 5.

3.4.2 Implementation on Dedicated Processor

For the dedicated implementation of the 1D DCT, the fast algorithms mentioned in the previous part are not suitable. This is because most of the fast algorithms have similar data flow as shown in Figure 3.5. The data flow of such fast algorithm is usually quite complex in the last stage. This makes the asynchronous implementation a disadvantage as a large handshake overhead will be introduced, and a degradation in the performance of the processor will be resulted. The solution to overcome this problem is to use dedicated multipliers and adders for each multiplication and addition. However this costs a lot of silicon area and thus is not practical.

For the asynchronous circuit, the dataflow should be as simple as possible. This helps to reduce the handshaking overhead and hence the performance can be enhanced. Therefore a semi-direct method is used in this dedicated DCT processor.

This semi-direct method is obtained by decomposing the 8×8 matrix multiplication into two 4×4 matrix multiplications. As a result, Equation 3.3 can be decomposed into 2 equations as shown in Equation 3.5 and Equation 3.6.

$$\begin{bmatrix} Y_0 \\ Y_2 \\ Y_4 \\ Y_6 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} A & A & A & A \\ B & C & -C & -B \\ A & -A & -A & A \\ C & -B & B & -C \end{bmatrix} \bullet \begin{bmatrix} x_0 + x_7 \\ x_1 + x_6 \\ x_2 + x_5 \\ x_3 + x_4 \end{bmatrix}$$ **- Equation 3.5**

$$\begin{bmatrix} Y_1 \\ Y_3 \\ Y_5 \\ Y_7 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} D & E & F & G \\ E & -G & -D & -F \\ F & -D & G & E \\ G & -F & E & -D \end{bmatrix} \bullet \begin{bmatrix} x_0 - x_7 \\ x_1 - x_6 \\ x_2 - x_5 \\ x_3 - x_4 \end{bmatrix}$$ **- Equation 3.6**

and similarly the IDCT can be decomposed into Equation 3.7 and Equation 3.8.

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} A & B & A & C \\ A & C & -A & -B \\ A & -C & -A & B \\ A & -B & A & -C \end{bmatrix} \bullet \begin{bmatrix} Y_0 \\ Y_2 \\ Y_4 \\ Y_6 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} D & E & F & G \\ E & -G & -D & -F \\ F & -D & G & E \\ G & -F & E & -D \end{bmatrix} \bullet \begin{bmatrix} Y_1 \\ Y_3 \\ Y_5 \\ Y_7 \end{bmatrix}$$ **- Equation 3.7**

$$\begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} A & B & A & C \\ A & C & -A & -B \\ A & -C & -A & B \\ A & -B & A & -C \end{bmatrix} \bullet \begin{bmatrix} Y_0 \\ Y_2 \\ Y_4 \\ Y_6 \end{bmatrix} - \frac{1}{2} \begin{bmatrix} D & E & F & G \\ E & -G & -D & -F \\ F & -D & G & E \\ G & -F & E & -D \end{bmatrix} \bullet \begin{bmatrix} Y_1 \\ Y_3 \\ Y_5 \\ Y_7 \end{bmatrix}$$ **- Equation 3.8**

This method has been used in many other DCT implementations [32][33]. There are several advantages for this semi-direct method. First the number of multiplications and additions are reduced to half of the original number. Second, it involves one multiplication only in each data path and thus it requires less numbers of bits to represent the data. Furthermore, the dataflow is simple, which is multiply-and-add, this favours the asynchronous implementation. Finally the structure of the DCT and IDCT are similar, it is easier to implement the DCT and IDCT on the same hardware by this method.

Based on the above reason, a 1D DCT core processor is constructed by using this semi-direct method, and is used in the dedicated 2D DCT processor. This processor is capable of handling DCT and IDCT, and can be cascaded to perform the 2D DCT. The detailed architecture and the implementation of the DCT algorithm will be discussed in chapter 6.

Chapter 4

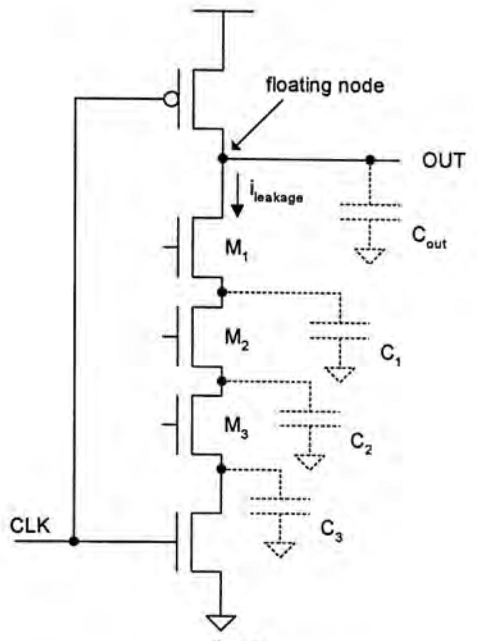
New Techniques for Operating Dynamic Logic in Low Frequency

4.1 Overview

Dynamic logic has some advantages over the static logic, they include higher speed and more compact in size. Moreover, it is suitable for used in the asynchronous circuit design as mentioned in chapter 2. However, dynamic logic is not widely used as it suffers from two main problems which are the racing problem [41], and the charge redistribution and leakage problem [41][42][43]. The racing problem can be avoided by a proper arrangement of the logic cell. However, the charge redistribution and leakage problem cannot be simply overcome as it is caused by its internal structure. This problem causes the dynamic logic to have a bad noise margin and a lower bound of operating frequency.

In this chapter, the problem of the charge redistribution and leakage problem, and its traditional solution will be discussed. Afterwards, a new technique to overcome this problem will be introduced.

4.2 Background



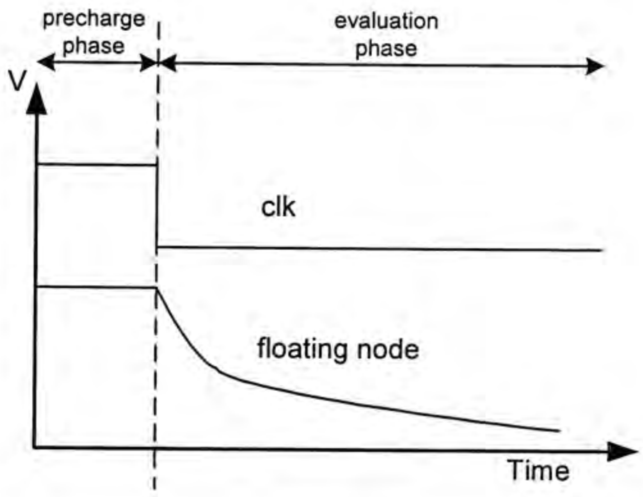


Figure 4.1 – (a) 3-input NAND dynamic logic, (b) voltage in the floating node of the dynamic logic

The output value of a dynamic logic depends on the charges stored in the floating node. By considering a 3-input NAND dynamic logic shown in Figure 4.1(a), during the Precharge phase, the output of the dynamic logic will be kept at high as the pMOS transistor is turned on and current is flowed from VDD to the floating node. During the Evaluation phase, unless all the nMOS transistors are turned on such that a pull-down path is created, the charges kept in the parasitic capacitor C_{out} at the floating node will hold the output at high. Otherwise, the output will become low as all the stored charges in the floating node flow out through the pull-down path.

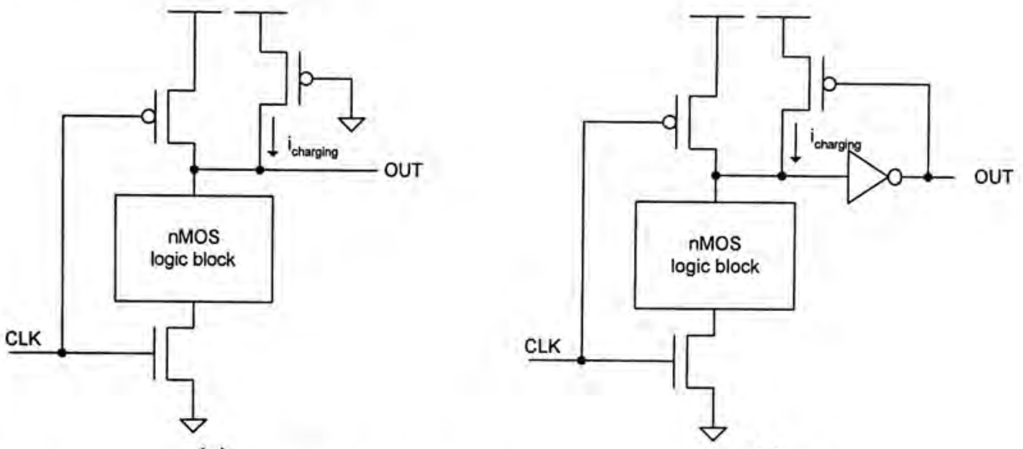
There are several advantages of the dynamic logic over the static logic. First the dynamic logic is more compact as the complementary pMOS transistor tree is replaced by only one pMOS transistor. Also the operation can be run faster as the output only needs to be selectively discharged during the Evaluation phase, and the charging speed is faster as there is only one pMOS transistor. An additional advantage for the asynchronous circuit is its temporary storage of data due to charges stored at the parasitic capacitor.

However, the dynamic logic is suffering from the charge redistribution and leakage problem. As mentioned previously, the output of the dynamic logic depends on the charges stored at the floating node. Theoretically if the pull-down path does not exist, the output should be kept at logic high during the Evaluation phase. In practice, the output voltage will be dropping continuously with time as shown in Figure 4.1(b). This problem is caused by the charge redistribution and charge leakage.

The charge redistribution problem can be explained by Figure 4.1(a). Suppose that during the Evaluation phase, the nMOS transistors M1 and M2 are turned on while M3 is turned off, there is no pull-down path to the ground and the output should keep high. However, since M1 and M2 are turned on two more capacitors C1 and C2 are introduced and they will share the charges stored in the floating node. This is called charge redistribution. As a result, voltage at the output drops and degrades the noise margin in the dynamic logic. If C1 and C2 are large and large amount of charges is flown out from the floating node to C1 and C2, the voltage at the floating node may be dropped below the switching threshold of the next stage and causes a logic error.

Furthermore, the charges will also be leaked out from the parasitic capacitor due to the leakage current [45][47]. If the time of the Evaluation phase is sufficient long, the diminishing charge will even induce a logic error at the output. Therefore the duration of the Evaluation phase should be short in order to prevent the logic error from occurring at the output. This limits the dynamic logic from operating in low frequency.

4.3 Traditional Technique



(a) (b)

Figure 4.2 – Addition of the pull-up path in (a)dynamic logic, (b)domino logic

The traditional method [42][43] used to overcome the charge redistribution and charge leakage problem is adding an additional small pull-up pMOS at the floating node. Figure 4.2 shows the traditional method used in the basic dynamic logic and domino logic. This additional pull-up pMOS directly solves both of the problems as it allows a current flow to the floating node during the Evaluation phase, and thus the charges stored in the parasitic capacitor can be maintained, or refilled. Due to its simplicity, this method is commonly used in most of the dynamic logic.

However, this method has a drawback of speed degradation. During the Evaluation phase, if a pull-down path is created by the nMOS logic block, the discharging current will be needed to fight against the charging current created from the additional pMOS, and thus a overall discharging current is decreased and the evaluation time is increased. Although a smaller charging current can be obtained by smaller pMOS transistor, but the transistor size is limited to the technology used and can only be reduced to a certain extent. Sometimes in poor design, the discharging current may even weaker than the charging current. In this case, the logic block will not be operated correctly and cause an error. These problems are caused by the limited or no control of the charging current from the additional pMOS.

4.4 New Technique – Refresh Control Circuit

Regarding the charge redistribution, there are several techniques [42] to overcome this problem, and some of them are shown in Figure 4.3. Also in practical design, the dynamic logic with a large nMOS logic block is always avoided as it has a poor performance due to the weak discharge current. In this case, the logic cell will usually be broken into two, or more simpler logic cells which have less transistors in the nMOS logic block. By using these techniques, the charge redistribution problem can be minimized. Therefore the charge leakage problem will only be dealt with in the new technique.

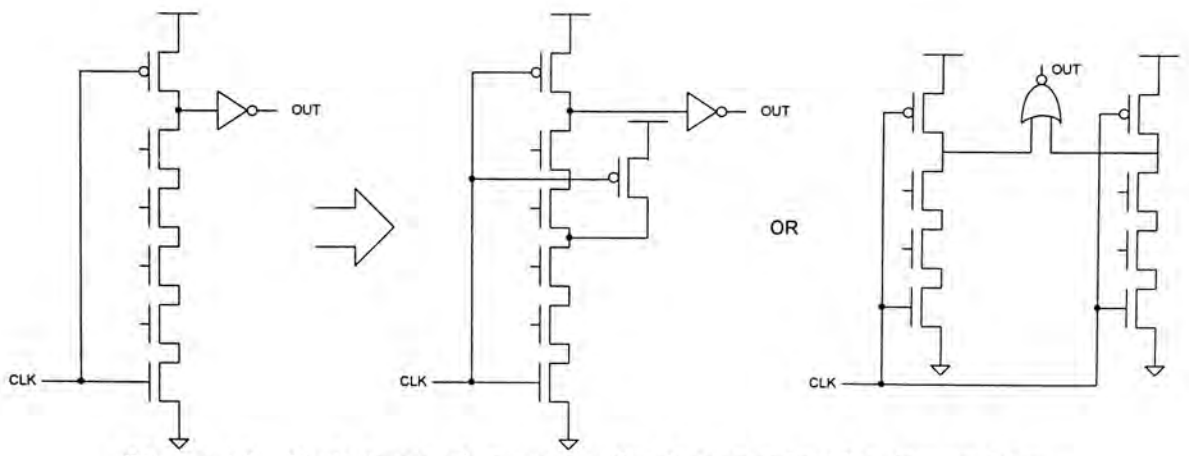


Figure 4.3 – Techniques of overcome the charge redistribution problem

In order to solve the charge leakage problem, the introduction of the pull-up path at the floating node seems to be necessary. However, the continuous flow of the current from VDD to floating node via the additional pull-up path causes the speed degradation. If the amount of current via the pull-up path is controlled, the speed degradation will be minimized. This is the aim of the new technique.

4.4.1 Principle

The idea of the new technique comes from the refresh technique used in Dynamic Random Access Memory (DRAM) [44][45]. The core circuit of the new technique is called the Refresh Control Circuit (RCC), and it is used to monitor the voltage of the floating node in the dynamic logic. When the floating node voltage meets the pre-determined minimum voltage, or V_{ref}, a pull-up path at the floating node is created for each dynamic logic in order to refill the charges in it. This process is called Refresh. Since the pull-up path is not present all the time, this technique causes less speed degradation compared with the traditional methods. Furthermore it is self-timed and self-operating. It does not need extra control from user. Figure 4.4 shows the modified structure of the dynamic and domino logics.

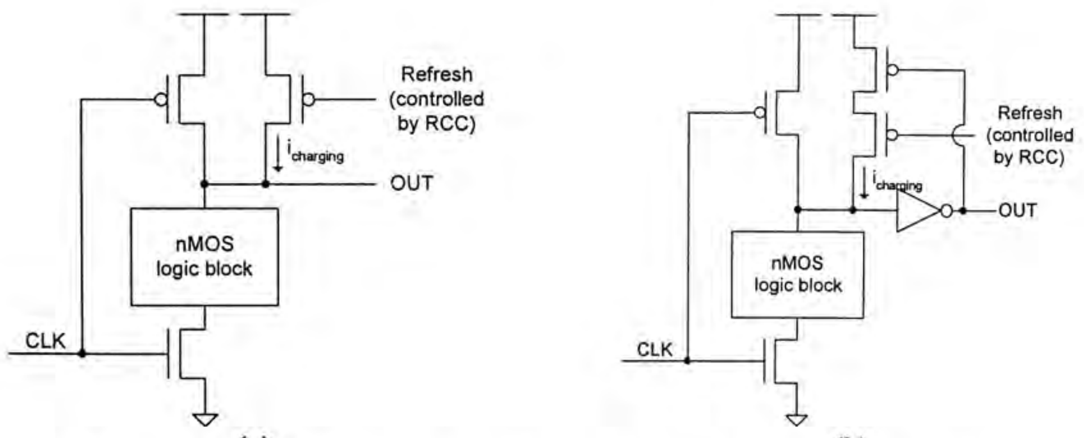


Figure 4.4 – Modified structure of the (a)dynamic logic, (b)domino logic

4.4.2 Voltage Sensor

First, in order to detect the voltage of the floating node and compare it with V_{ref}, a voltage sensor is used. In the system, not all the logic is connected to the voltage sensor. Only a dummy circuit modeling with the worst dynamic logic structure in terms of leakage is used to represent all the logic cells in the circuit and it is connected to the voltage sensor as shown in Figure 4.5.

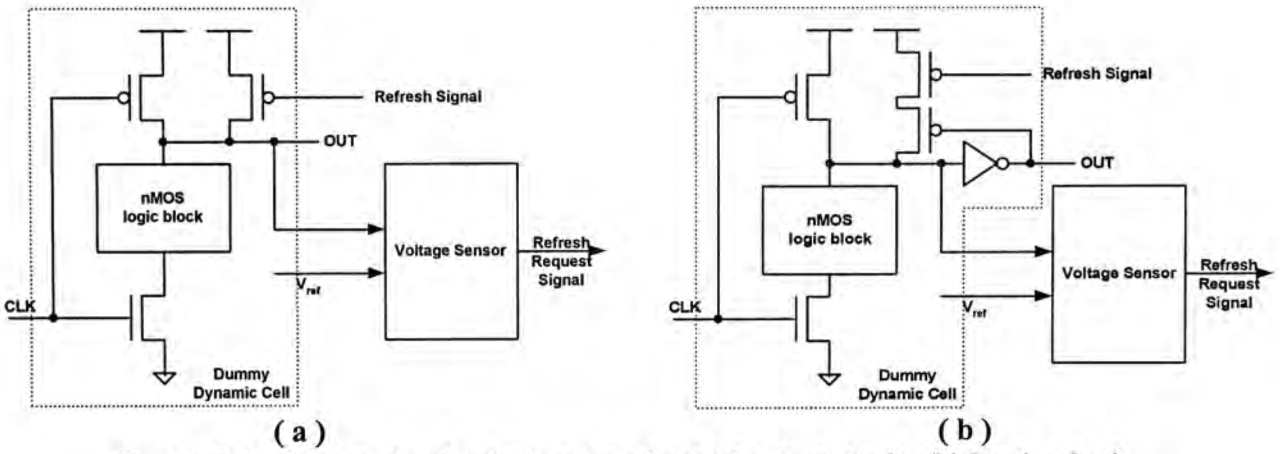
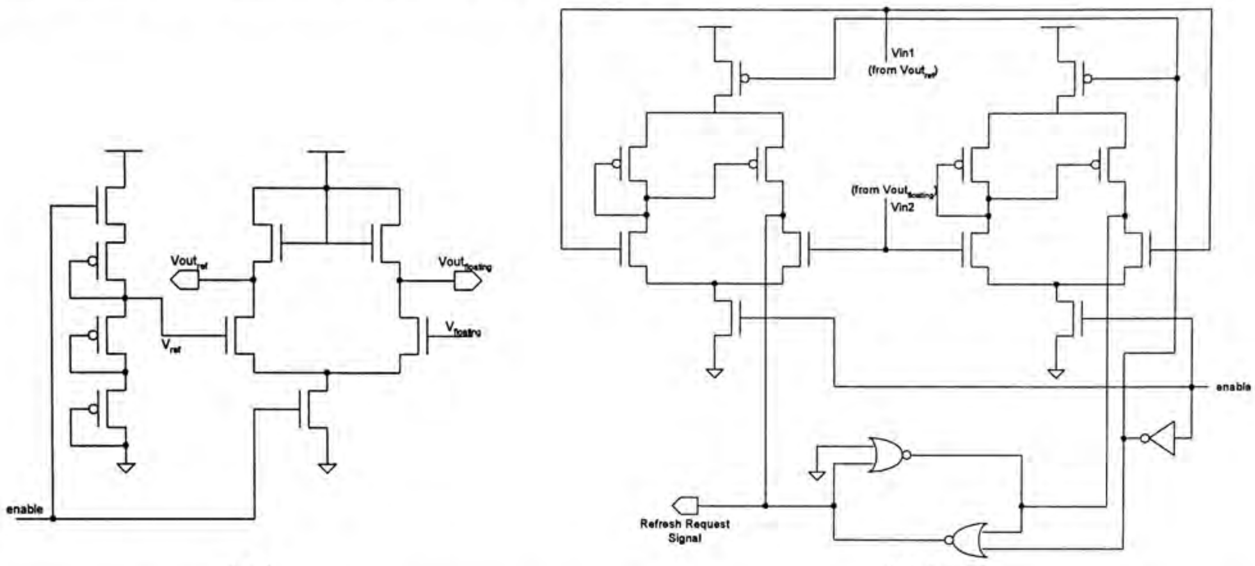


Figure 4.5 – Proposed refresh structure for (a)dynamic logic, (b)domino logic

The voltage sensor consists of 2 stages, the first stage is the differential amplifier and the second stage is the two-stage sense amplifier. Their structures are shown in Figure 4.6. The first stage, the differential amplifier, is used to compare the voltage of the floating node with V_{ref}, and to amplify their difference. The inputs of the second stage, the two-stage sense amplifier, are connected to the outputs of the first stage to provide a more accurate comparison. If the voltage in the floating node becomes smaller than V_{ref}, the second stage will generate the refresh request signal to indicate to the dynamic logic that refresh is needed.



(a) (b)

Figure 4.6 – Voltage sensor, (a)differential amplifier as the first stage with reference voltage generator, (b)two-stage sense amplifiers as the second stage

The two-stage voltage sensor can provide a good detection. However, it consumes a lot of power as there is a current always flowing from VDD to GND, it should be prevented from operating all the time. As the sense amplifiers are only used to determine the time for refresh, a timer can be used to record the time required for refresh in the first refresh process. Afterwards, the sense amplifier can be turned off and the signal from the timer can be used to indicate a time for refresh. In practice, the combination of a ring oscillator, a counter and latch can form a timer.

4.4.3 Ring Oscillator

Ring oscillator is constructed by connecting an odd number of inverters with a feedback, which is shown in Figure 4.7.

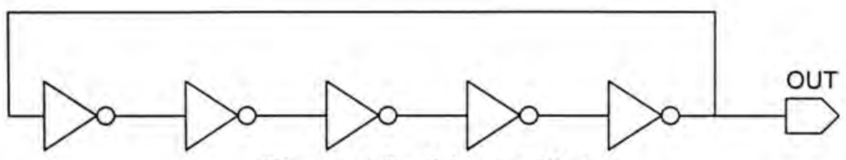


Figure 4.7 – Ring oscillator

For a ring oscillator, its period (or frequency) is controlled by the size and number of inverters. The size of the inverter means the width-to-length (W/L) ratio of the pMOS and nMOS in the inverter. In general, the smaller the W/L ratio, the longer the period can be obtained. This is because the charging and discharging current, as shown in Figure 4.8, are smaller in small pMOS and nMOS, and thus it requires longer time to charge or discharge the input capacitor of the next inverter. Furthermore, the more the inverters used, the longer the period can be made in the oscillator as a longer delay is created in the feedback path.

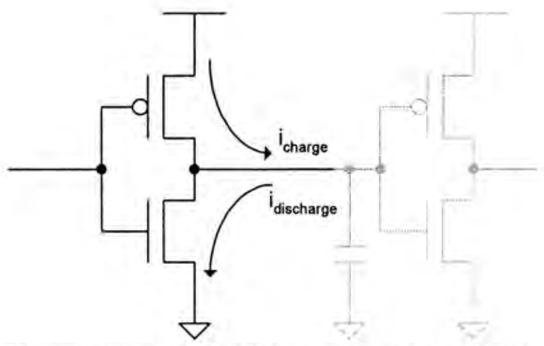


Figure 4.8 – Charging and discharge current in the inverter chain

Normally, the time for a logic error occurring at the floating node due to charge leakage should be in the order of milli-second (10^{-3} second) [47]. If a ten-bit counter is used to count the refresh time, the ring oscillator will need to have a period in the order of micro-second (10^{-6} second). However, the period of an ordinary oscillator is very short (several nano-second, 10^{-9} second) even when the smallest inverters are used. The increase in the number of inverters can increase the period, but it is not practical. It is because the difference between the delay of an inverter and the required period is too large, it may require thousands of inverters so as to achieve the required oscillating period and this makes the oscillator very large. Also, the large amount of inverter causes a large amount of power consumption. Rather than inverter, delay elements are used. It is added between each inverter and creates larger delay in the feedback path. Figure 4.9 shows the ring oscillator with the delay elements.

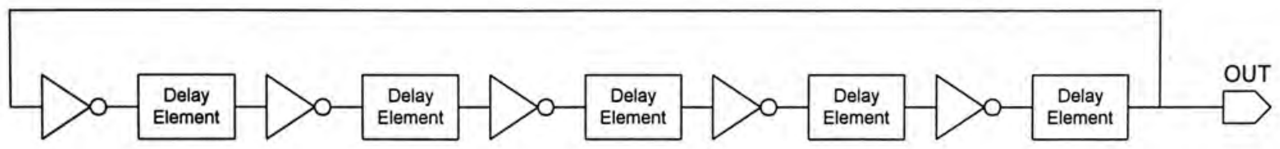


Figure 4.9 – Ring oscillator with delay elements

There are many types of delay elements. The common one is a transmission gate but it cannot achieve a long delay. By referring to the comparison done by Mahapatra et. al. [46], the transmission gate with Schmitt trigger [46] is chosen as the delay element in the ring oscillator as it produces longer delay. The CMOS structure of the transmission gate with Schmitt trigger is shown in Figure 4.10

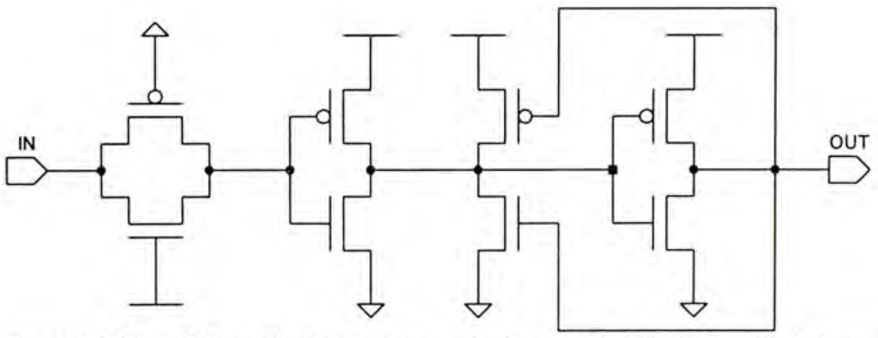


Figure 4.10 – Delay element, transmission gate with Schmitt trigger

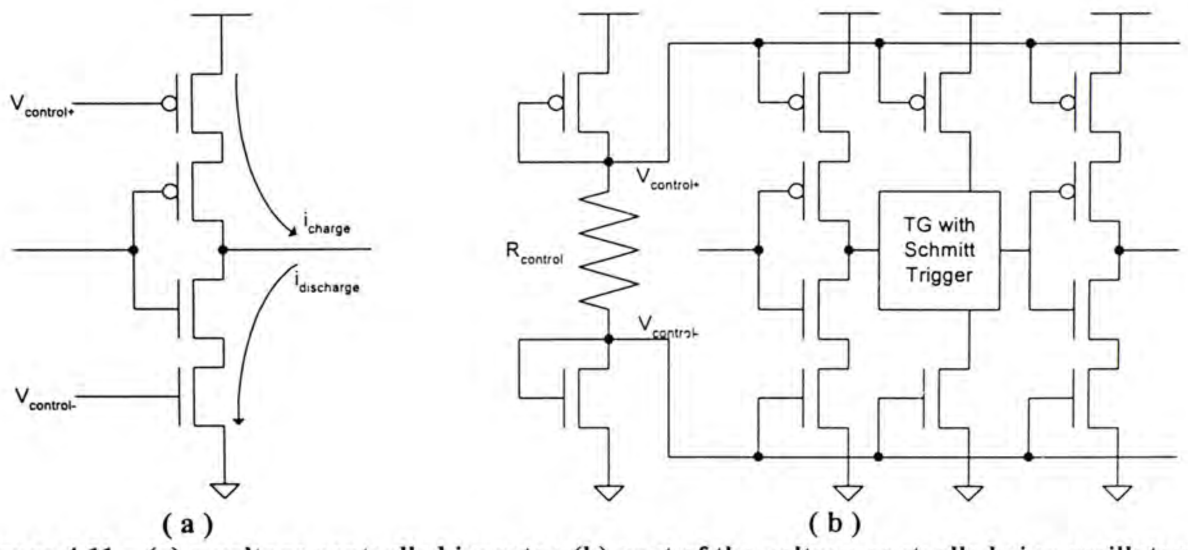


Figure 4.11 – (a) a voltage controlled inverter, (b) part of the voltage controlled ring oscillator

In order to further increase the delay, the minimization of the charging and discharging currents (Figure 4.8) are required. As mentioned previously in this section, the smaller the current, the longer the charging/discharging time and thus a longer period can be achieved. The minimization of current can be done by adding small transistors in the VDD and GND paths, which is shown in Figure 4.11(a). By providing the control voltage near to the threshold to the added transistors, the charging and discharge currents can be adjusted to a very small value as both currents are limited by the added transistors. The method of providing the controlled voltage is shown in Figure 4.11(b). As a result, a frequency of 38.5 KHz (period of 26us) is achieved in this ring oscillator.

4.4.4 Counter, Latch and Comparator

Counter is connected to the ring oscillator in order to record its number of period. As mentioned before, the time for a logic error occurring at the floating node due to charge leakage is in the order of milli-seconds. Therefore, the dynamic logic should be refreshed every several or tens of milli-seconds. This constrain indicates that the timer should be able to record the time in the order of milli-seconds.

As the ring oscillator is constructed at 38.5KHz, a ten-bit counter is enough the for recording the time as

$$\begin{aligned} \textit{Recordable Time} &= \textit{Clock Period} \times 2^{\textit{Number of Bit of Counter}} \qquad \textbf{- Equation 4.1} \\ &= 26us \times 2^{10} \\ &= 26us \times 1024 \\ &= 26.6\ ms \end{aligned}$$

The latch is used to record the number of clock period required to carry out the refresh process for the first time. The input of the latch is connected to the output of the counter. When the first refresh is required, the refresh request signal from the voltage sensor will trigger the latch and causes the latch to record the value of the counter. This value is meaningful as it indicates the number of clock period required to have a refresh. Afterwards, the voltage sensor can be turned off, and the refresh process is controlled by the comparator. The comparator is used to compare the output values of the counter and latch all the time. When their values are the same, this means that the dynamic logic reaches the time to carry out the refresh process, the comparator will send out a signal to request a refresh.

4.4.5 Recalibrate Circuit

The amount of leakage current is highly related to the temperature [44]. The higher the temperature, the larger the leakage current flows out from the floating node. As a result, the time required to carry out a refresh process is varied with the temperature. In the real world, the temperature of the chip may vary with time, therefore the circuit should have a recalibrate function such that the refresh time is recalculated after certain time.

The recalibrate circuit is actually a five-bit counter. It counts the number of refresh processes has been taken and controls the ON and OFF of the voltage sensor. Initially the refresh counter starts counting from zero, and the voltage sensor is enabled. After the first refresh took place, the refresh counter is incremented to one and the voltage sensor is disable afterward. After $2^5 - 1$ refresh processes, the refresh counter counts back to zero and the voltage sensor is enabled again. As a result the latch will record a new counter's value by the trigger of the refresh request signal from the voltage sensor and thus the recalibration can be made.

4.4.6 Operation Monitoring Circuit

When the actual system is operating, i.e. there is a transition of the clock signal synchronous circuit or there is a request signal in asynchronous circuit, the voltage sensor is not required to detect the floating node voltage as the charge in the floating node will be retained during the normal Precharge phase. Under this situation, the voltage sensor is not necessary to be turned on and thus power can be saved. Therefore the operation monitoring circuit helps to detect when the system is operating, and it will disable the voltage sensor and reset the counter if necessary.

4.4.7 Overall Circuit

By combining all the necessary units, the Refresh Control Circuit is formed as shown in Figure 4.12.

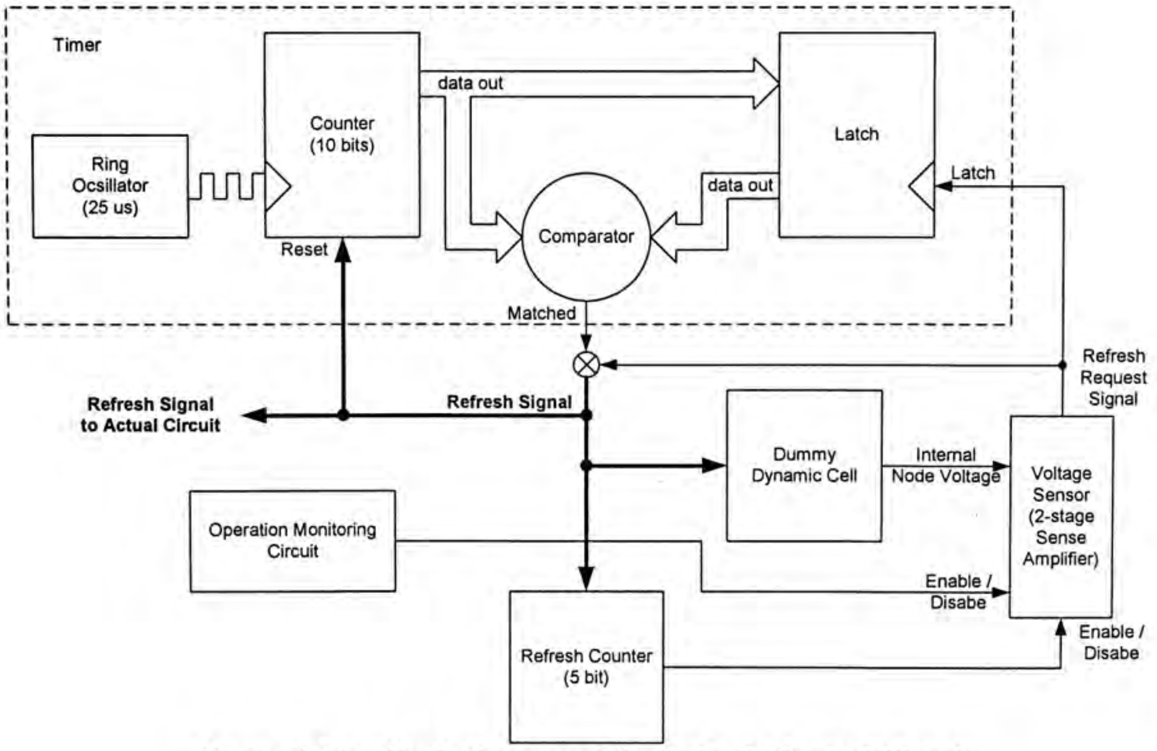


Figure 4.12 – Block diagram of the Refresh Control Circuit

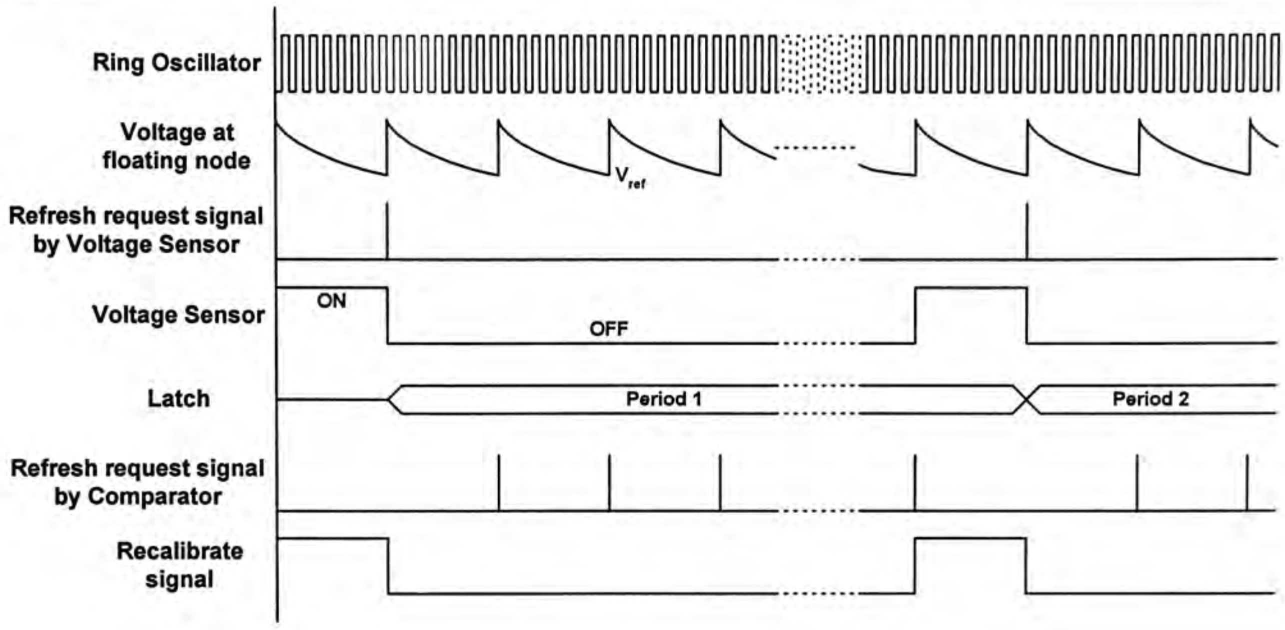


Figure 4.13 – Timing diagram of the Refresh Control Circuit

The timing diagram of the operation of the Refresh Control Circuit is shown in Figure 4.13. Initially the voltage sensor is enabled and the voltage of the floating node of the dummy dynamic logic cell decreases with time. When the voltage meets the pre-defined minimum level, the voltage sensor generates the refresh request immediately. This signal will first trigger the latch to record the value of the counter. Also the refresh request signal will be passed out to reset the counter, increment the refresh counter and refresh the dummy dynamic cell and the actual circuit.

Due to the increment in the refresh counter, the voltage sensor is disabled. However, the timer is now enabled and is able to generate the refresh signal by comparing the value of the counter and latch. It continues until the refresh counter returns to zero, then a recalibration is required and the voltage sensor is enabled again. The whole process will then be repeated afterwards.

The performance of the Refresh Control Circuit will be shown in chapter 7. Also, multipliers are constructed to test and compare the performance of this new technique with the traditional technique. The result will be shown in chapter 7 as well.

As the Refresh Control Circuit is still in the schematic level design, this technique is not applied on the implementation of programmable DSP processor and dedicated DCT/IDCT processor.

Chapter 5

DCT Implementation on Programmable DSP Processor

5.1 Overview

As the number of transistors is increasing, it becomes attractive to build design system in asynchronous style as it has benefits of no clock skew, lower power consumption and low electromagnetic noise. Several asynchronous processors have been built [11][48][49][50], and the AMULET3 [50] has been used commercially. This indicates that asynchronous designs are plausible alternative to synchronous designs.

In this chapter, a pipelined dataflow [10][51] micro-coded DSP asynchronous processor will be discussed. The architecture of this DSP processor was developed by our research group, and I have made some modifications, and I am responsible for the DCT implementation and layout generation of the whole processor. The programming technique and the implementation of DCT will also be given at the end of this chapter.

5.2 Processor Architecture

The design of this DSP processor follows the dataflow architecture. In other word, this is a data-driven system. The dataflow architecture naturally fits the asynchronous design. The combined architecture allows the data to be sent into the system continuously without external control or clock, and the presence of data triggers the operation of the asynchronous system automatically.

In order to realize the pipelined dataflow architecture in an asynchronous system, a pipelined processor is developed. The target of this processor is to implement some simple DSP operations such as Infinite Impulsive Response (IIR) filter, Fast Fourier Transform (FFT) and DCT, where addition, subtraction and multiplication with constant are required.

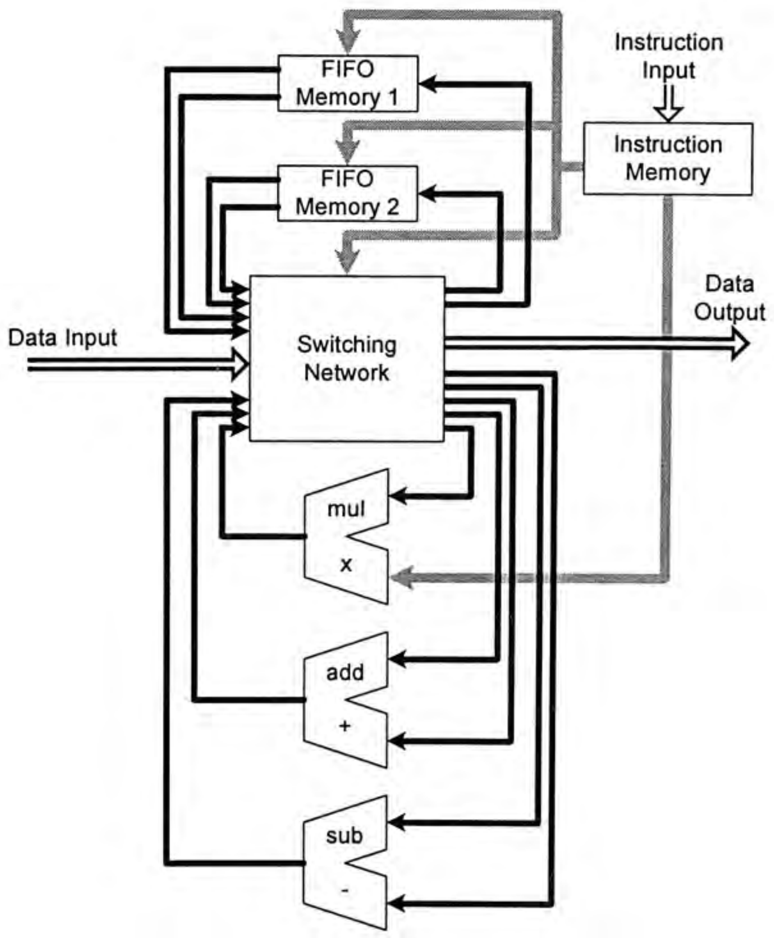


Figure 5.1 – Dataflow architecture of the programmable DSP processor

The architecture of the processor with the necessary functional blocks is shown in Figure 5.1. The processor includes an adder, a subtractor, a multiplier, two FIFO memories, a switching network, and an instruction memory. It the following sections, each part of the processor will be discussed.

5.2.1 Arithmetic Unit

In this processor, the adder, subtractor and multiplier are all pipelined and are designed in DCVSL structure in order to maximize the performance. Multiplier is based on the bit-parallel architecture. In this architecture, the multiplier core can be built by an array of a Product Full Adder (PFA), which is shown in Figure 5.2. Each PFA carries out four functions, which are given by

$$Aout = Ain \qquad \text{- Equation 5.1}$$
$$Bout = Bin \qquad \text{- Equation 5.2}$$
$$Pout = (Ain \bullet Bin) \oplus (Pin \oplus Cin) \qquad \text{- Equation 5.3}$$
$$Cout = Ain \bullet Bin \bullet Cin + Pin \bullet (Ain \bullet Bin + Cin) \qquad \text{- Equation 5.4}$$

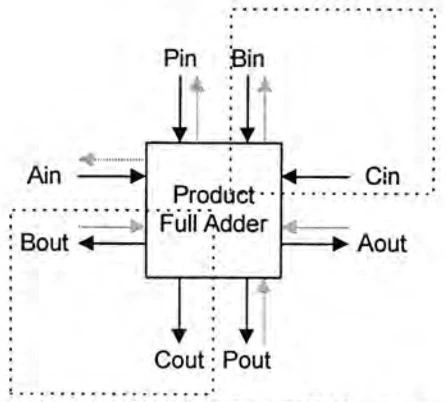


Figure 5.2 – Product Full Adder (PFA) of the multiplier

All the signals in the PFA have their own handshake signals, except B shares the handshake signal of P as they propagate to the same direction. The overall structure of the multiplier core is shown in Figure 5.3.

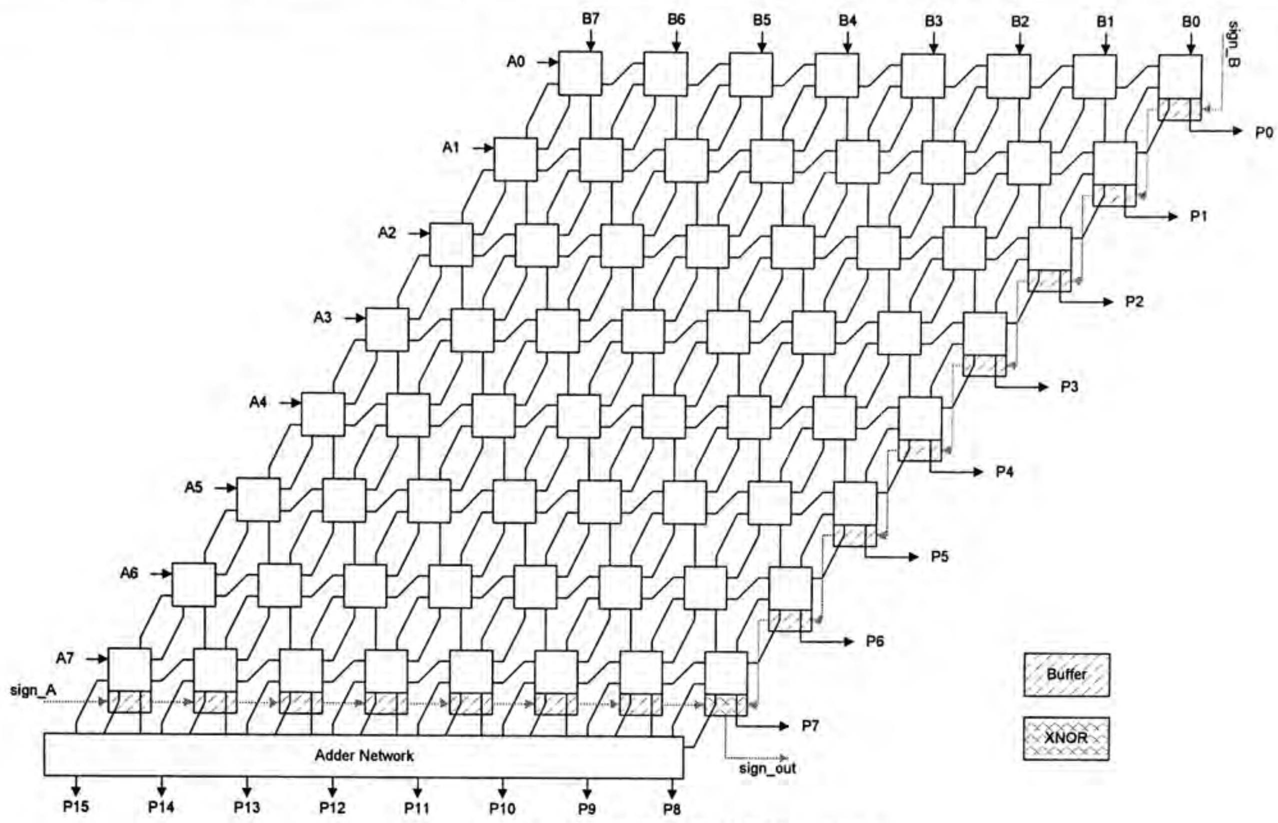


Figure 5.3 – The 8x8 multiplier core

In Figure 5.3, A and B are the inputs while P is the product of A and B. The number behind the inputs and output represent the bit position, where bit0 is the least significant bit (LSB). Since the data format of this processor is a 1-bit sign bit with 8-bit magnitude, the sign bit of the final product is just the XOR result of the two inputs' sign bit. As a result, buffers are added in the multiplier core so that the sign bits of both inputs are shifted to the right-bottom block to perform the XOR operation, and the sign bit of the final product can be obtained.

Unlike the synchronous version, the asynchronous bit-parallel multiplier requires different bits of the inputs arriving at different time. This is because within the multiplier core the next PFA can only start operation when the results, C and P, of the previous PFAs are ready. In the current architecture, (A0, B7) will be calculated first, the next operation will be started at (A0, B6) and (A1, B7), and so on. Due to this requirement, a ladder-shape input buffers are used at two inputs in order to schedule the arrival time of different bits. The structure of the input buffer is shown in Figure 5.4.

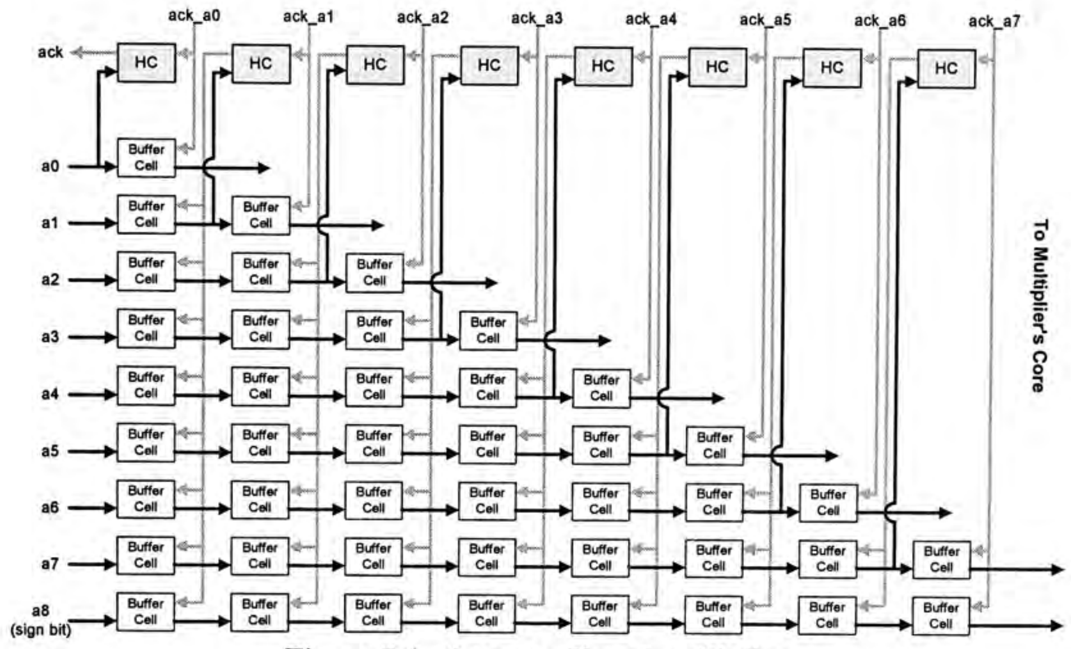


Figure 5.4 – Input buffer of multiplier

Similarly, different bits of the output P come out at different time, and bit0 will come out first in this configuration. As a result, a ladder-shape output buffer is also required at the output side.

The adder is based on the Carry Look-ahead (CLA) architecture [52][53]. This architecture provides a faster computation time by reconstructing the Sum and Carry of the addition by 2 new values, which are Propagate P and Generate G. The new formulae are given as the followings,

$$G_i = A_i \bullet B_i \qquad \text{- Equation 5.5}$$

$$P_i = A_i \oplus B_i \qquad \text{- Equation 5.6}$$

$$C_i = G_i + P_i \bullet C_{i-1} \qquad \text{- Equation 5.7}$$

$$S_i = C_{i-1} \oplus P_i \qquad \text{- Equation 5.8}$$

By computing several Ps and Gs in parallel, the Sum and Carry of several bit locations can be obtained simultaneously. As a result, the addition can be carried out in a faster way.

In this processor, the subtractor is actually another Carry Look-ahead adder with an inversed input as $A-B=A+(-B)$.

5.2.2 Switching Network

In some designs, data transfer is done via a common data bus. However, it is difficult to be implemented in an asynchronous dataflow system as large handshaking overhead and long delay will be introduced. For example, there is a common data bus shared by one receiver and three transmitters. When the data exists in the data bus, the handshake cell in the receiver is required to communicate with all the three transmitters in order to know which the source is. The time required must be longer than a normal handshaking time in the pipeline stage. If the number of the receivers and transmitters is increased, the time required for handshaking will be increased exponentially and a longer delay will happen.

Instead of using common data bus, a multi-stage switching network is used to connect the different units. There are several advantages for using multi-stage switching network in an asynchronous system. Firstly, the network allows parallelism. In other words, data from different inputs can be sent to different outputs simultaneously. Secondly, the network is pipelined resulting in higher data transfer rate via the network. Lastly, the switching network distributes the handshake signals to the corresponding destination only and thus the large handshake overhead is avoided.

The basic component of the multi-stage switching network is a two-to-two programmable switch cell. It can perform six modes of connection according to a 3-bit instruction. The structure and the modes of connection are shown in Figure 5.5(a).

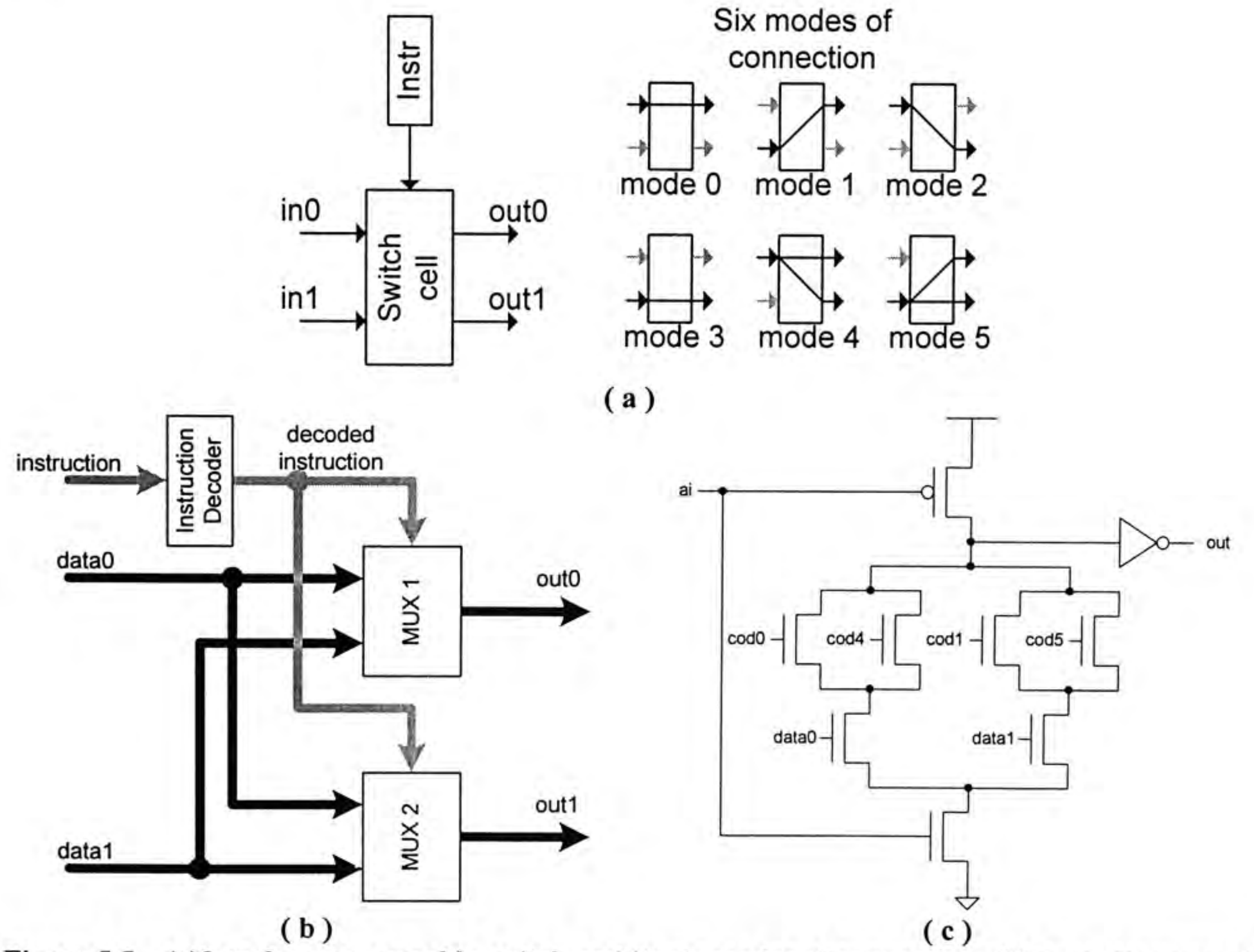


Figure 5.5 – (a)2-to-2 programmable switch and its six modes of connection, (b)block diagram of the internal structure of switch, (c)CMOS structure of basic multiplier cell of the MUX 1

The design of the switch cell follows the dataflow architecture. It uses the same communication protocol and handshake cell as the one presented in the previous chapter. The switch is basically built up by an instruction decoder and two two-to-one multiplexers. During operation, the instruction decoder receives and decodes the instructions from the instruction memory. It translates the 3-bit instruction into a six-bit decoded word, which is shown in Table 5.1, and then passes it to the two multiplexers. Each bit of the decoded word corresponds to one mode of connection. It helps to have a simpler design of multuplexer cell for faster operation. The multiplexer is built in the form of sum-of-product structure and domino style, which as shown in Figure 5.5(c). It receives the decoded instruction and detects the presence of the input data. Once the corresponding input data has been ready, the data is copied to the output of the multiplexer, and thus the transmission of data can be done.

Instruction	*Function / Connection Mode*	*Decoded Word (COD)*
000	in0→out0 / mode 0	000001
001	in1→out0 / mode 1	000010
010	in0→out1 / mode 2	000100
011	in1→out1 / mode 3	001000
110	In0→out0 & out1/ mode 4	010000
111	In1→out0 & out1/ mode 5	100000

Table 5.1 – Instructions of switch

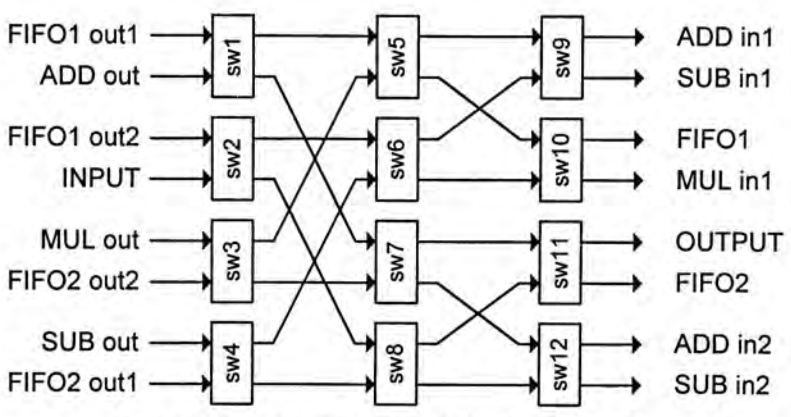


Figure 5.6 – 8-to-8 switching network

In this programmable DSP processor, the switching network is a matrix of 3×4 switch cells allowing eight-to-eight connections, as shown in Figure 5.6. The position of the inputs and outputs is tuned and optimized to allow maximal concurrency and efficient resources assignment.

5.2.3 FIFO Memory

There are two FIFO memories within the processor, which are responsible for temporary data storage during the operation. The structure of the FIFO memory is shown in Figure 5.7. It is organized in four short FIFO sets (FIFO A to D, each stores 4 data) and one long FIFO set (FIFO E, stores 16 data). Demultiplexers, which have similar structure as the switch, are placed at the input and output of the FIFO memory for controlling the data flow to/out from the corresponding FIFO set.

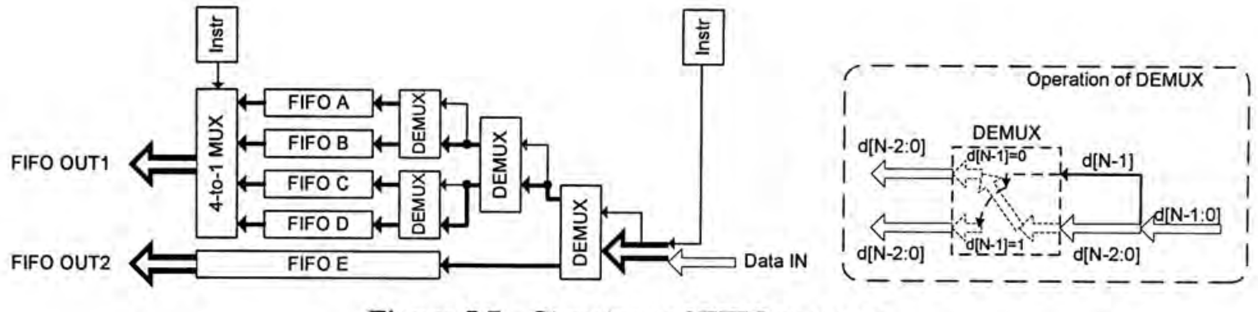


Figure 5.7 – Structure of FIFO memory

The basic building unit of the FIFO set cell is the basic FIFO cell, which is shown in Figure 2.13(b) in chapter 2. The basic FIFO cell captures the input data in the Evaluation phase and retains it in the Hold phase. A parallel connection of n FIFO memory cells to a handshake cell can form a single n-bit FIFO memory stage. If several FIFO pipeline stages are cascaded, a FIFO set will be formed. The input data will queue and be held inside the FIFO set until the switching network is ready for accepting the data from the FIFO block.

The input of the FIFO memory is connected to a three-stage demultiplexing network. Inside the FIFO memory, the instruction and data are first merged to be a single data which is in the form of [instruction] + [data]. When it arrives at the input of the demultiplexer, the most significant bit (MSB) of the instruction will be extracted and acts as the controlling signal for switching, and the rest of the bits will pass through the demultiplexers. This combination of instruction and data reduces the handshaking overhead in the demultiplexer and thus a faster transfer speed can be achieved inside the demultiplexing network. Also, the use of three-stage demultiplexing network prevents the fan-out and the large handshaking overhead problems occurred in a single one-to-five switch. Also the data can be transferred to the long FIFO set in shorter latency such that the data in long FIFO set can be reused in shorter time. The four-to-one mulitplexer is used at the output of the FIFO block. Its structure is similar to that of the switch cell, which is shown in the previous part.

5.2.4 Instruction Memory

The inclusion of the instructions allows the processor to be programmable and to perform different operations. In this processor, the instruction is used to control the connections of the switches and multiplexers in the switching network and FIFO memories, and also used for the multiplicand for the multiplier. The instructions are all stored in the instruction memory.

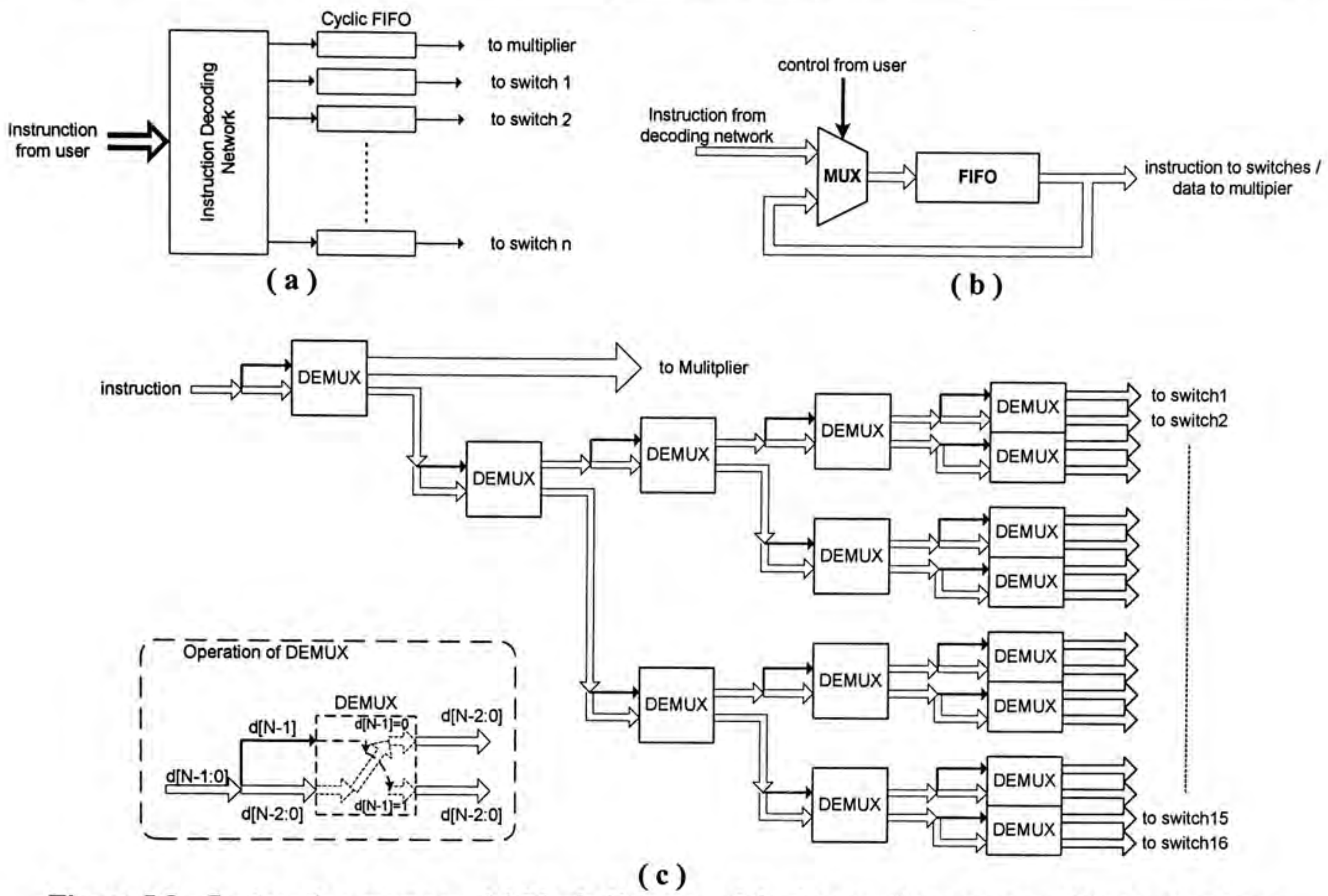


Figure 5.8 – Instruction memory, (a) block diagram of the instruction memory, (b) the structure of cyclic FIFO, (c) structure of the instruction decoding network

There are two mains parts in the instruction memory which are the instruction decoding network and the cyclic FIFOs, as shown in Figure 5.8(a). An instruction is in the format of [address] + [data]. After receiving the instruction, the instruction decoding network, as shown in Figure 5.8(c), decodes the address by demultiplexing, which is similar to the demultiplexing network in the FIFO memory, and sends the data to the corresponding FIFO or the multiplier.

The FIFO in here has a cyclic feature, which is shown in Figure 5.8(b). Besides from sending to the corresponding destination, the outputted instruction is fed back to the FIFO as well. This feature permits the instructions to be recycled and thus the application can be run repeatedly without further programming. During programming, the switch at the input is connected to the instruction decoding network for collecting the instructions, and then it will be switched to another end for recycling the instructions.

5.3 Programming

In this dataflow processor, programming is just the organization of the flow of data. In other words, the switches are programmed to perform a connection from one unit to another unit. For example, there are 2 inputs A and B. In order to perform an addition of A and B in this processor, 2 cycles are required to send the inputs to the adder and a third cycle is needed to send the adder's output to the processor's output.

Step 1 : *A (from input) → Adder Input1,*
Step 2 : *B (from input) → Adder Input 2*
Step 3 : *Adder Output → Output*

In the actual programming, the following switches are required to be programmed as follows,

For step 1 : *sw2 → mode 1, sw6 → mode 0, sw9 → mode 1,*
For step 2 : *sw2 → mode 3, sw8 → mode 2, sw12 → mode 0,*
For step 3 : *sw1 → mode 3, sw7 → mode 0, sw11 → mode 0*

Therefore, an addition requires 3 cycles. However, for example, if the two inputs are sent from the internal FIFO memories 1 and 2, only 2 cycles are required for an addition as no switch is shared between both the input paths (referred to the switching network in Figure 5.6). Therefore the data from FIFO memories 1 and 2 can be sent to adder input1 and input2 respectively within the same cycle. Similarly, data can be sent to different arithmetic units or FIFO memories in the same cycle provided that their paths do not share the same switch. Programming which can fully utilize the parallelism of the switching network maximizes the concurrency of the arithmetic operations and thus the greatest performance of the processor can be achieved.

5.4 DCT Implementation

As mentioned in chapter 3, the implementation of DCT in the processor is based on the algorithm proposed by Jeong et. al. [40]. The DCT programme can be divided into four stages, which is in shown from Figure 5.9 to Figure 5.12.

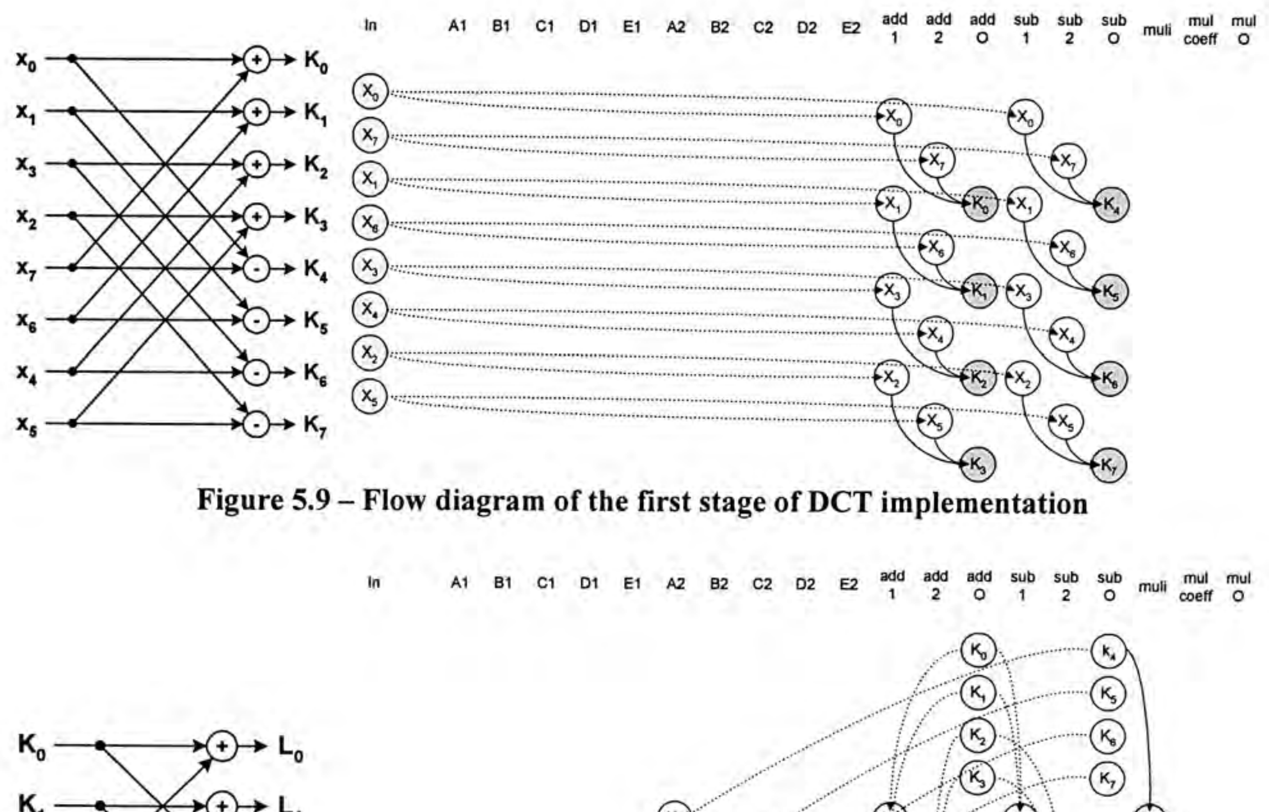


Figure 5.9 – Flow diagram of the first stage of DCT implementation

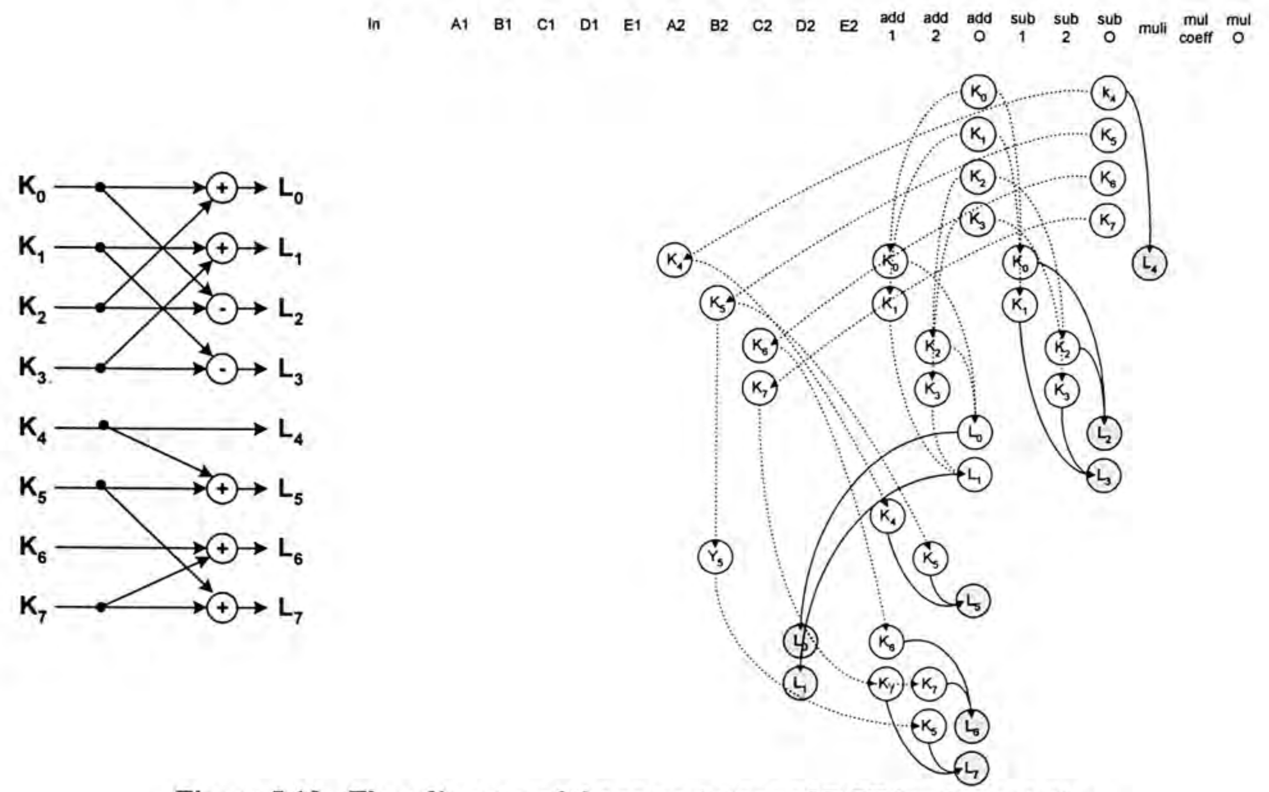


Figure 5.10 - Flow diagram of the second stage of DCT implementation

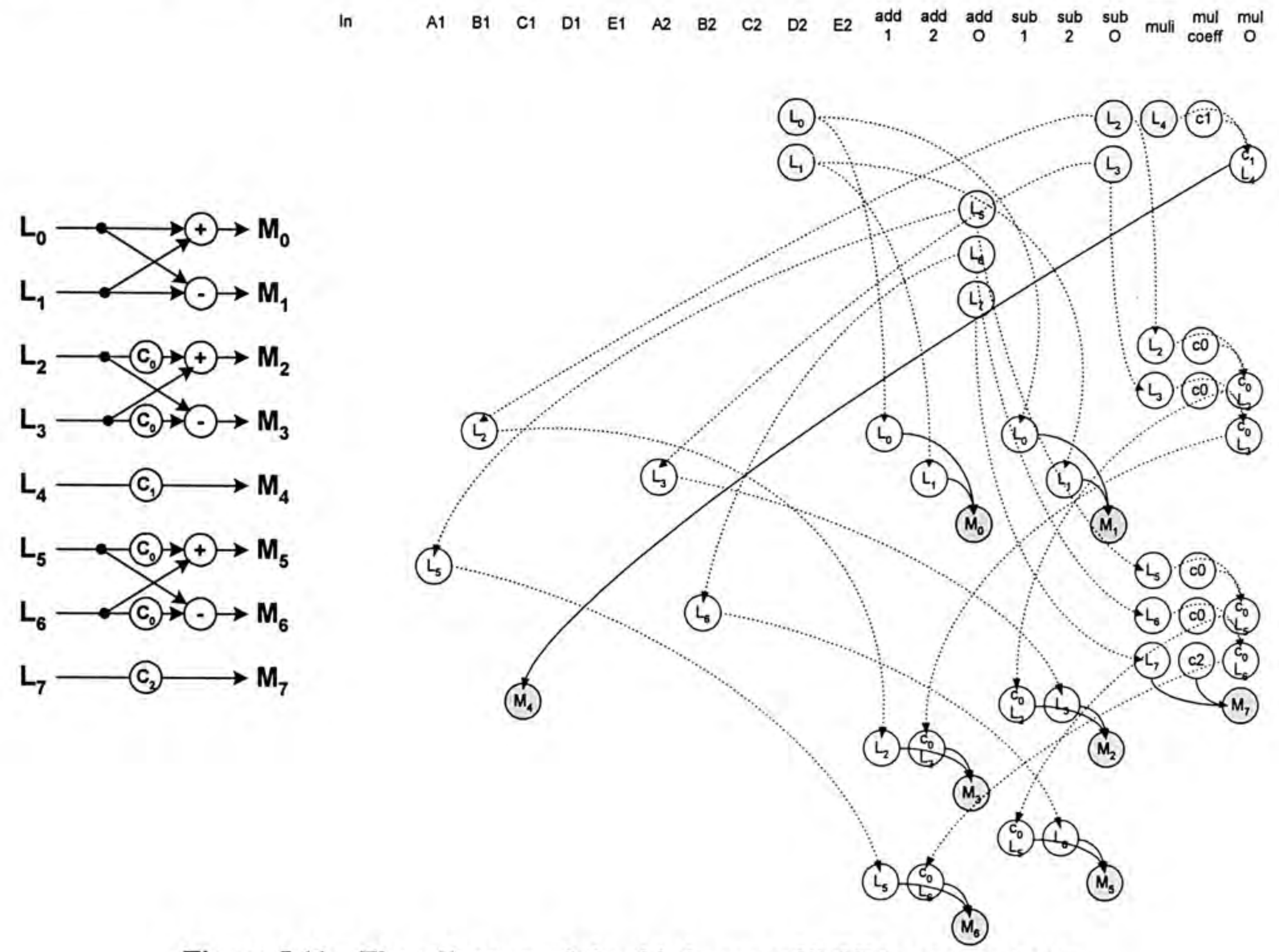


Figure 5.11 – Flow diagram of the third stage of DCT implementation

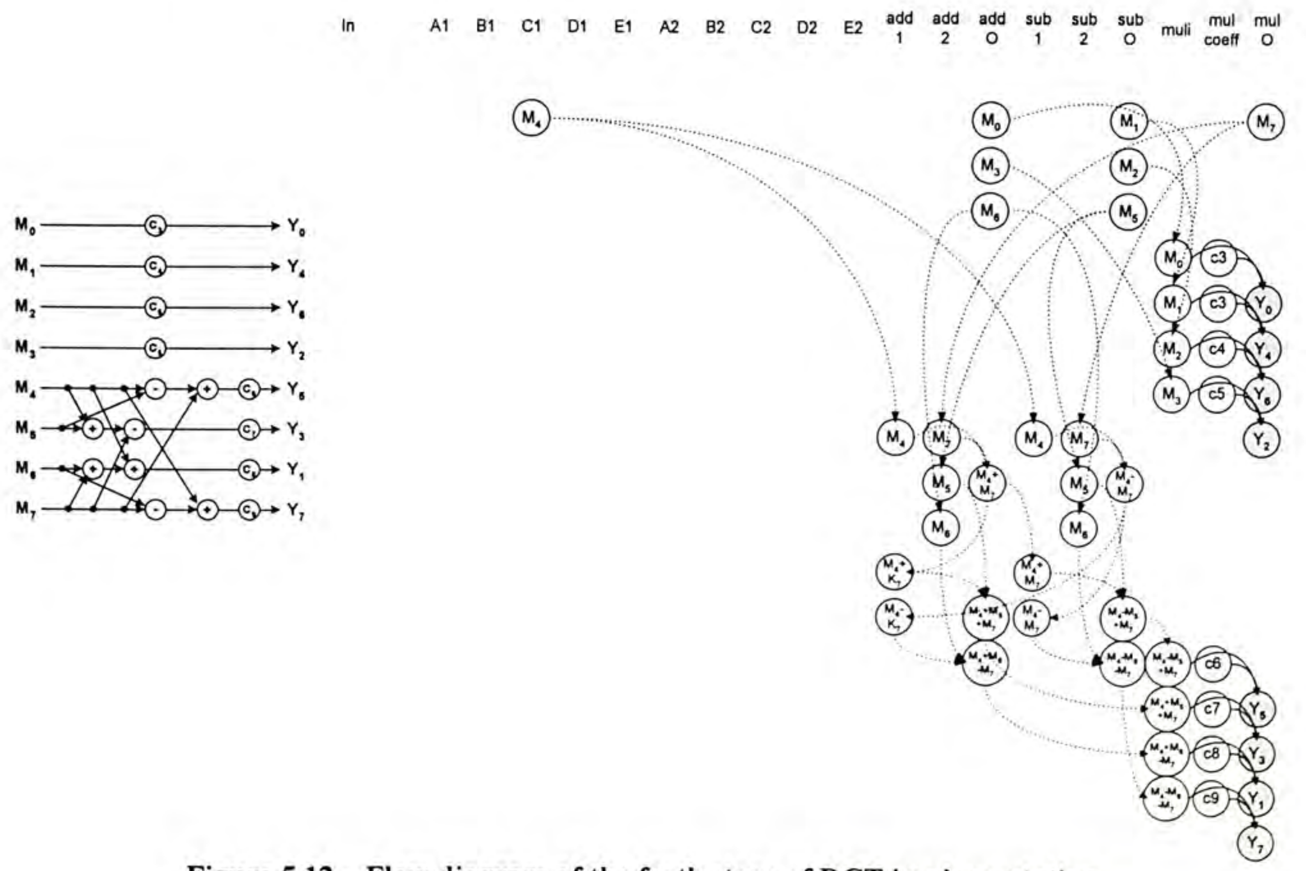


Figure 5.12 – Flow diagram of the forth stage of DCT implementation

The flow diagrams show the data flow in the DCT algorithm. In the flow diagram, *In* means the input, *A*, *B*, *C*, *D* and *E* mean the FIFO sets in the two FIFO memories. *add 1*, *add 2* and *add O* are referring to the input 1, input 2 and the output of the adder respectively. Subtractor and multiplier also have the similar representations. Due to the parallelism and concurrency of the switching network, two or more data are always controlled to transfer simultaneously in order to increase the throughput of the switching network, and thus more operations can be carried out by the arithmetic units and the performance can be increased. Also in order to avoid data queuing, it is necessary to send the data to FIFO memories for temporarily storage in sometime.

The detailed steps of this programme (includes the instructions of each switches) are shown in appendices, and the performance of the DCT implementation is given in the chapter 7.

Chapter 6

DCT Implementation on Dedicated DCT Processor

6.1 Overview

As the demand of the high quality signal, the computation requirement of the video and image applications nowadays becomes higher and higher. For the application of the discrete cosine transform such as the MPEG2 (640×480, 30 fps, 4:2:0, 13.82 Mpixel/sec) or High Definition Television (HDTV) (74.23MHz in luminance signal for baseband HDTV), a very high processing rate of a 2D DCT/IDCT design is required. Although the processing power of a general purpose processor is high, it is still difficult to provide a real-time processing on these signals. On the other hand, dedicated processor for specific application can provide an effective solution. It always provides a cost effective and higher performance solution for these applications. By further applying the asynchronous pipelined architecture on these designs, a higher performance may be achieved.

In this chapter, a dedicated 8×8 2D DCT/IDCT asynchronous processor is introduced. The processor has a fully pipelined in the architecture, and provides a very high transform rate which is capable of real-time processing on high quality signal. The architecture of the 2D DCT/IDCT processor will be introduced at the beginning. Since the architecture is based on the row-and-column decomposition method, the design of the 1D DCT core and the transpose memory will be given afterwards.

6.2 DCT Chip Architecture

As discussed in chapter 3, the 2D DCT design is based on the row-and-column decomposition method which provides a simpler implementation and is more suitable for the asynchronous architecture. Figure 6.1 shows the dataflow in the 2D DCT by using the row-and-column decomposition method. In the row operation, 1D DCTs are applied on each row of data, and then the results are stored in the transpose memory row-wise immediately. For the column operation, the 1D DCTs are applied on the data stored in the transpose memory in column-wise, and the resultant values of the column operation are the 2D DCT result.

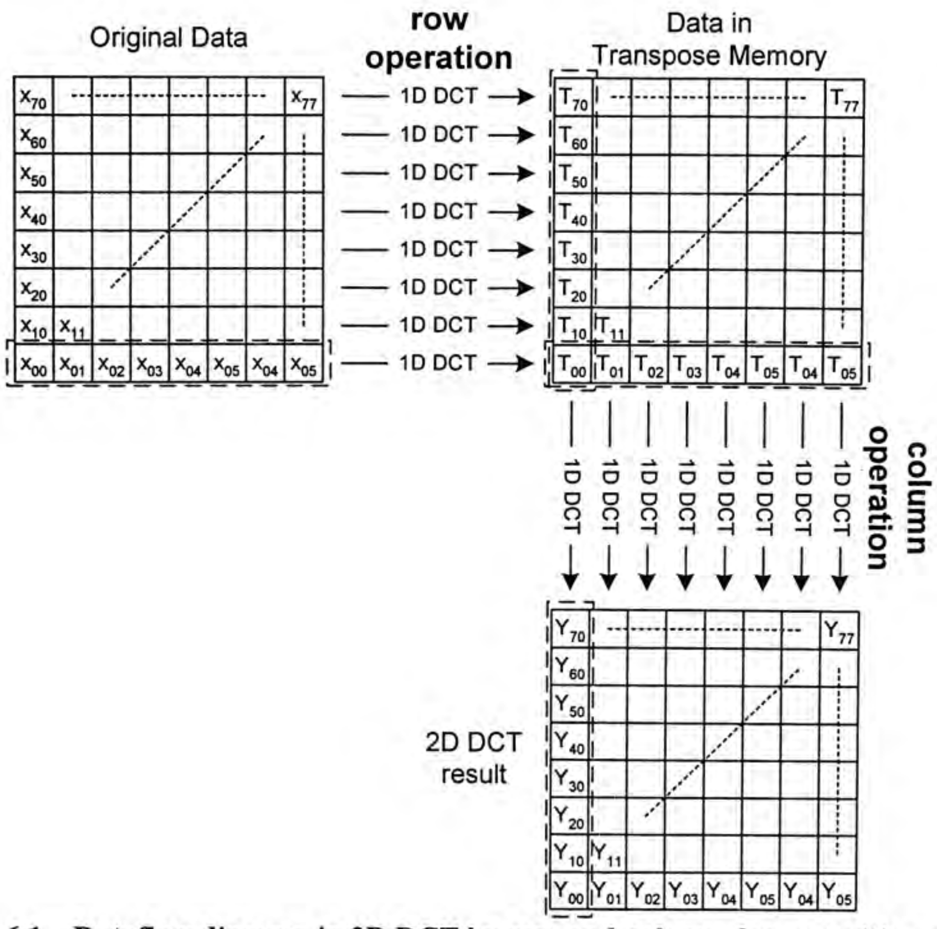

Original Data
row operation
Data in Transpose Memory
1D DCT
column operation
2D DCT result
x_{00} x_{01} x_{02} x_{03} x_{04} x_{05} x_{04} x_{05} x_{10} x_{11} x_{20} x_{30} x_{40} x_{50} x_{60} x_{70} x_{77}
T_{00} T_{01} T_{02} T_{03} T_{04} T_{05} T_{04} T_{05} T_{10} T_{11} T_{20} T_{30} T_{40} T_{50} T_{60} T_{70} T_{77}
Y_{00} Y_{01} Y_{02} Y_{03} Y_{04} Y_{05} Y_{04} Y_{05} Y_{10} Y_{11} Y_{20} Y_{30} Y_{40} Y_{50} Y_{60} Y_{70} Y_{77}

Figure 6.1 – Dataflow diagram in 2D DCT by row-and-column decomposition method

From the dataflow diagram, it is shown that it requires eight DCT operations in both of the row and column operations. It means there are totally 16 DCT operations in the whole 2D DCT operation. However in the physical realization of the row-and-column decomposition method, it is not necessary to use 16 independent 1D DCT cores to perform the row and column operations. This is because the data is entering the processor serially. A single 1D DCT core can be shared for the eight 1D DCTs by each operation. As a result, only one 1D DCT core is required in row and column operation, and the block diagram of the 2D DCT architecture is shown in Figure 6.2. Since the architecture of both the 1D DCT core can be the same, it saves the time of designing.

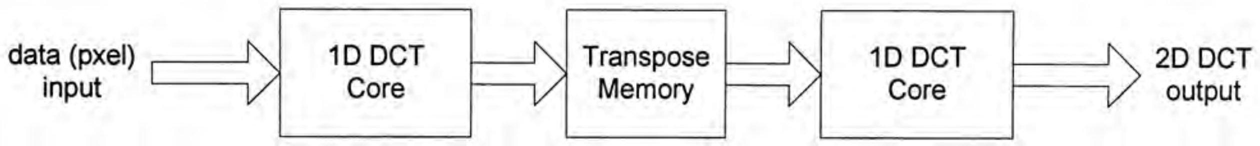


Figure 6.2 – Block diagram of 2D DCT processor

The detailed architecture of the 1D DCT core will be discussed in next section of this chapter. For the transpose memory, it is built by an ordinary Static Random Access Memory (SRAM) with an address generator to control the write and read processes. The detailed architecture of the transpose memory will be discussed in section 6.3.

6.2.1 1D DCT Core

The implementation of the 1D DCT core is based on Equation 3.5 and Equation 3.6 shown in chapter 3. By dividing the equations into two parts, Equation 6.1 to Equation 6.3 can be obtained.

$$\begin{bmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{bmatrix} = \begin{bmatrix} x_0 + x_7 \\ x_1 + x_6 \\ x_2 + x_5 \\ x_3 + x_4 \end{bmatrix} \text{ and } \begin{bmatrix} Z_4 \\ Z_5 \\ Z_6 \\ Z_7 \end{bmatrix} = \begin{bmatrix} x_0 - x_7 \\ x_1 - x_6 \\ x_2 - x_5 \\ x_3 - x_4 \end{bmatrix}$$ **- Equation 6.1**

$$\begin{bmatrix} Y_0 \\ Y_2 \\ Y_4 \\ Y_6 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} A & A & A & A \\ B & C & -C & -B \\ A & -A & -A & A \\ C & -B & B & -C \end{bmatrix} \bullet \begin{bmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{bmatrix}$$ **- Equation 6.2**

$$\begin{bmatrix} Y_1 \\ Y_3 \\ Y_5 \\ Y_7 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} D & E & F & G \\ E & -G & -D & -F \\ F & -D & G & E \\ G & -F & E & -D \end{bmatrix} \bullet \begin{bmatrix} Z_4 \\ Z_5 \\ Z_6 \\ Z_7 \end{bmatrix}$$ **- Equation 6.3**

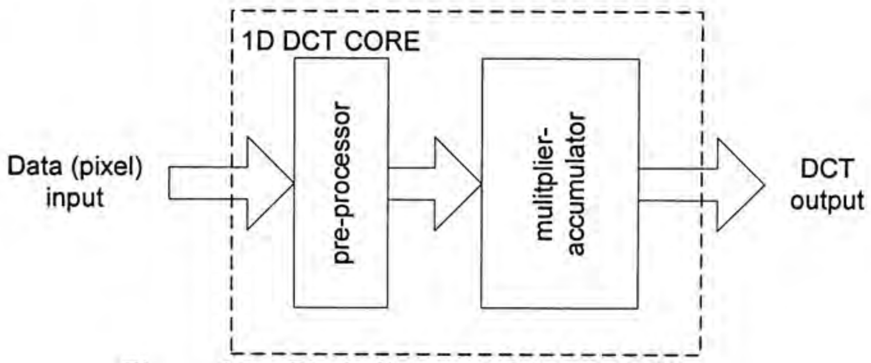


Figure 6.3 – Block diagram of the 1D DCT core

Figure 6.3 shows the basic architecture of the 1D DCT core [32][33][34], which is constructed by a pre-processor and a multiplier-accumulator. The pre-processor is responsible for the operation described by Equation 6.1. It collects the input data and performs the addition and subtraction, according Equation 6.1. Since only simple addition and subtraction are required, the pre-processor includes an adder and subtractor.

In this processor, the Binary Look-ahead Carry (BLC) adder [30][54] is used. Compared to the Carry Look-ahead (CLA) adder, each processing block of the BLC adder handle two sets of Propagate and Generate only. This simplifies the operation within the basic block and thus the speed can be increased. However, the drawback is the silicon area and longer latency. A 8-bit version of the BLC is shown in Figure 6.4

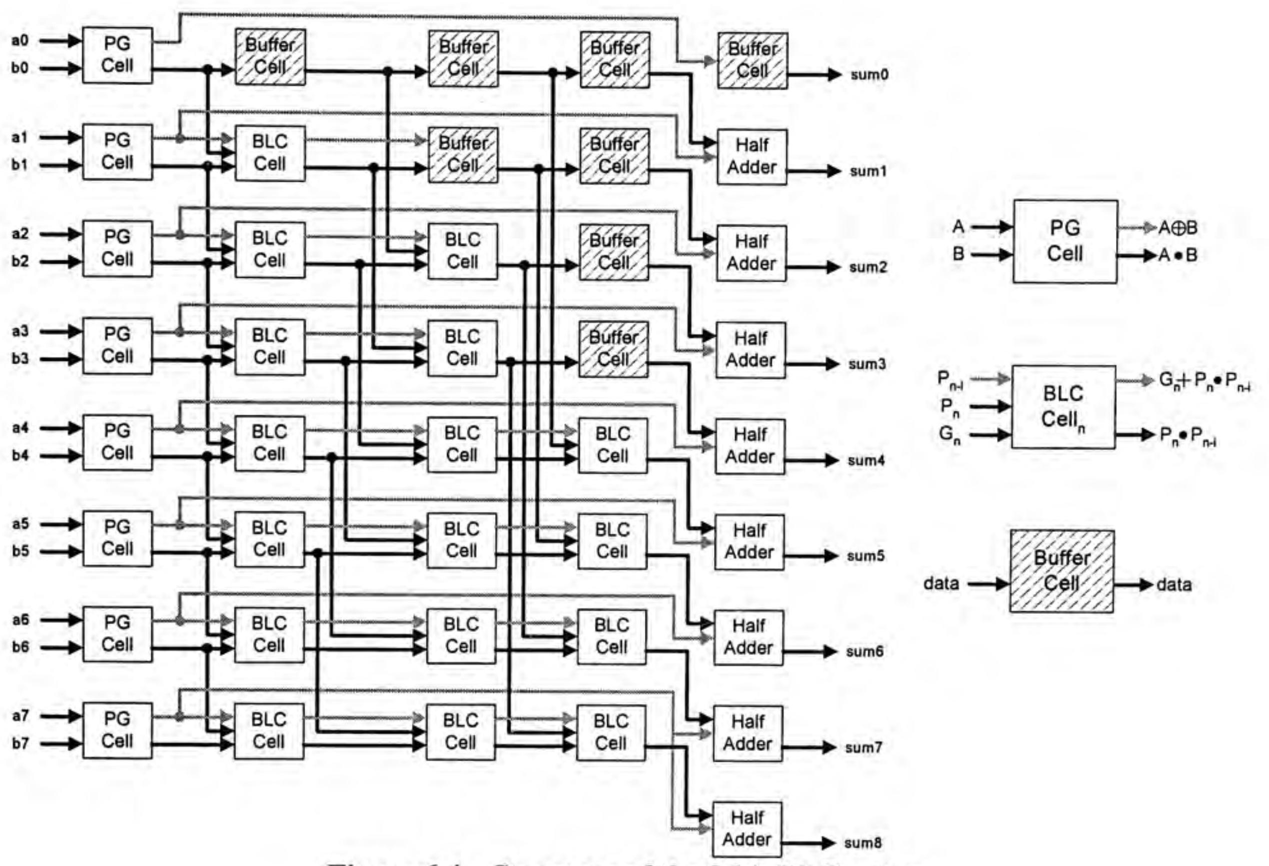


Figure 6.4 – Structure of the 8-bit BLC adder

The second part of the ID DCT core is the multiplier-accumulator. It is responsible for the matrix multiplication described by Equation 6.2 and Equation 6.3. The matrix multiplication can be done by multiply-and-add. It receives the output from the pre-processor and performs 16 multiplications with the DCT coefficients, and then adds the results according to the order. As a result, the multiplier-accumulator is constructed by multipliers and adders.

Besides from directly using the multipliers and adders, distributed arithmetic (DA) [55] method is used to implement the multiplier-accumulator in some designs [32][33]. The principle of the DA is to use a Read Only Memory (ROM) based look-up table (LUT) to replace the multiplier. Since the DCT coefficients are fixed, the result of the multiplication can be pre-calculated and stored in the ROM. In this way, the input data acts as an address to read the data which is stored in the ROM. Since the ROM based LUT can be built very compactly, the advantage of DA is saving silicon area as a general dedicated multiplier is avoided. However, the DA does not fit the style of the asynchronous architecture, and the read operation on ROM cannot be pipelined. As a result, a general pipelined multiplier based on the bit-parallel algorithm is used in this 1D DCT core as it can be pipelined and run very fast, but the trade-off is the size.

Basically the architecture of this bit-parallel multiplier is the same as the one used in the programmable processor, which has been described in Chapter 5. However, it cannot be used directly in this DCT core. This is because the bit-parallel architecture is primary designed for the multiplication of two unsigned value, but it is two complement data format in the DCT core. As a result, a conversion of a two complement data into a unsigned value with a sign bit is required. This conversion is done in the input buffer, and the converted output can be used in the multiplier core. The mechanism of conversion can be illustrated in the following example. For a 9-bit data having a the binary representation of 111110101, the conversion can be done by

$$\begin{array}{lccccccccc}
\text{Original 2-complement binary value} & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
\textit{Step 1 : Inversion} & & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\textit{Step 2 : add 1 to the result} & & & & & & & & + & 1 \\
\hline
 & & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0
\end{array}$$

The resultant binary number shows a decimal value of 12, so the original value represents a value of –12.

In this implementation, the conversion is divided into 2 stages. The first stage is the step 1, which will invert the input bit if the input data is a negative value. This can be done by the XOR gates which XOR all the data bits with the sign bit. The second stage is the step 2 which can be performed by an adder. As the conversion is handled at the input buffer part, a ripple adder is used as it fits the ladder structure of the input buffer. Figure 6.5 shows the modified input buffer used in the 2-complement multiplier.

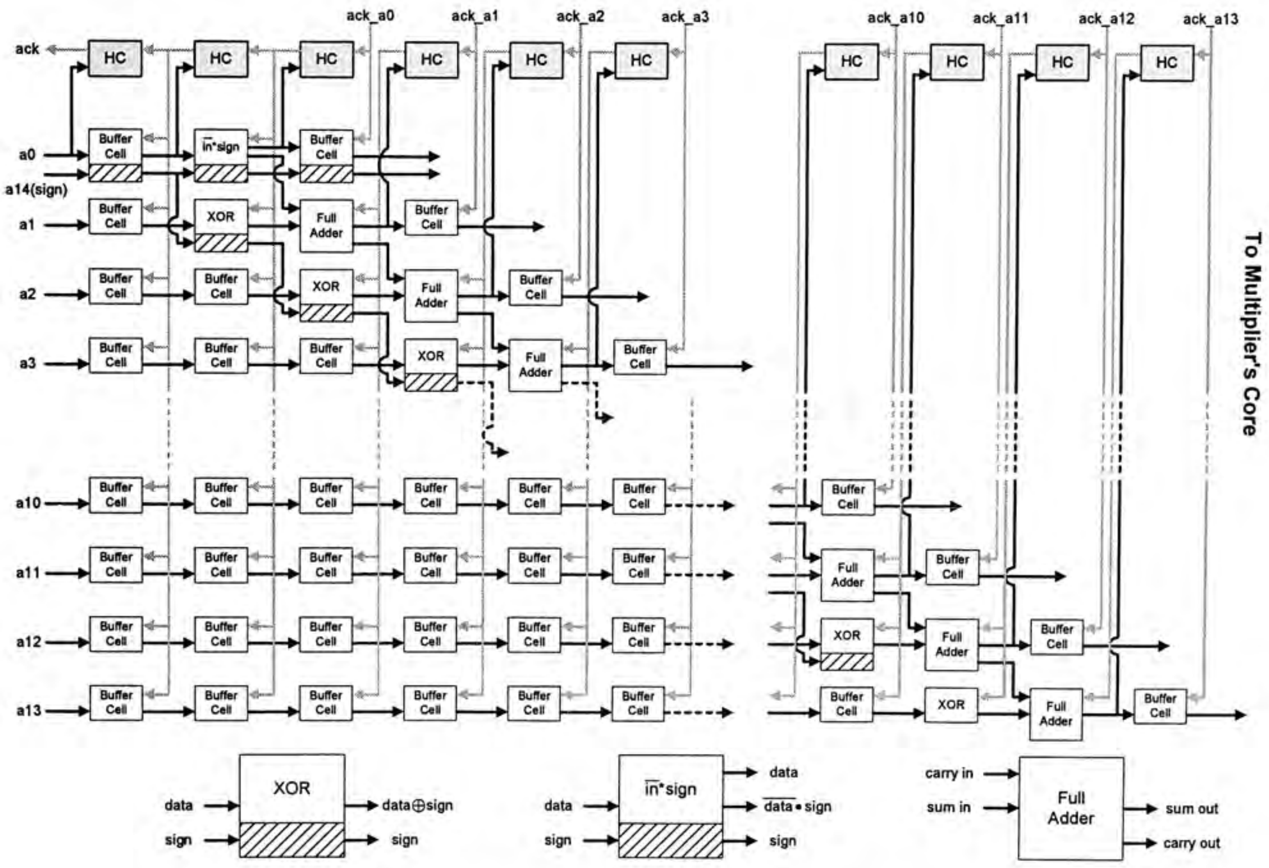


Figure 6.5 – Modified input buffer for 2 complement input

A similar conversion is required at the output as the unsigned product result is required to be converted back to a two complement data format. The conversion is merged into the output buffer and its structure is similar to the input buffer shown in Figure 6.5. For the conversion at output, the sign bit of the result must be ready at the same time as the bit0 in order to perform the conversion at once. Therefore the sign bits path in the multiplier core is modified in order to calculate the output sign bit first. The resultant structure of the multiplier core is shown in Figure 6.6. Together with the modified input and output buffers, a two complement bit-parallel multiplier is formed.

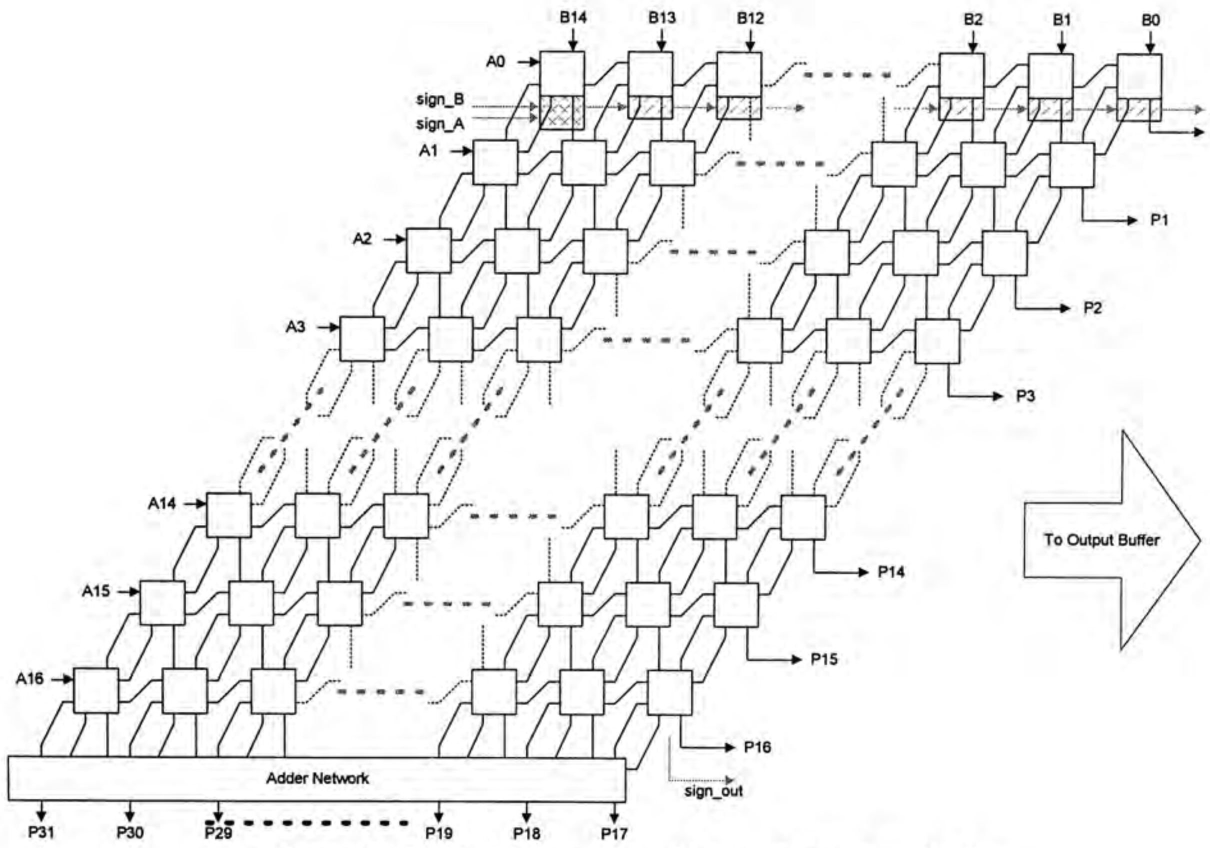


Figure 6.6 – Multiplier core of 2 complement multiplier

For the accumulator part, some of the design uses an adder with an output feedback to perform the accumulation of the multiplier's outputs. However, this structure is very slow as the second addition cannot be performed until the previous addition is finished and fed back to the input. Also, it wastes the pipeline architecture as only one addition can be carried out at anytime. Therefore several BLC adders are used in this design in order to achieve a better performance.

6.2.1.1 Core Architecture

For the asynchronous architecture, simpler and direct dataflow allows easier implementation and better performance as it reduces the handshaking overhead and fits the asynchronous pipeline architecture. In order to develop a simple dataflow, Equation 6.2 is further decomposed to Equation 6.4,

$$\begin{bmatrix} Y_0 \\ Y_2 \\ Y_4 \\ Y_6 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} A \\ B \\ A \\ C \end{bmatrix} \bullet [Z_0] + \frac{1}{2}\begin{bmatrix} A \\ C \\ -A \\ -B \end{bmatrix} \bullet [Z_1] + \frac{1}{2}\begin{bmatrix} A \\ -C \\ -A \\ B \end{bmatrix} \bullet [Z_2] + \frac{1}{2}\begin{bmatrix} A \\ -B \\ A \\ -C \end{bmatrix} \bullet [Z_3]$$ - **Equation 6.4**

Let

$$U_0 = \frac{1}{2}\begin{bmatrix} A \\ B \\ A \\ C \end{bmatrix} \bullet [Z_0],\ U_1 = \frac{1}{2}\begin{bmatrix} A \\ C \\ -A \\ -B \end{bmatrix} \bullet [Z_1],\ U_2 = \frac{1}{2}\begin{bmatrix} A \\ -C \\ -A \\ B \end{bmatrix} \bullet [Z_2],\ and\ U_3 = \frac{1}{2}\begin{bmatrix} A \\ -B \\ A \\ -C \end{bmatrix} \bullet [Z_3]$$ - **Equation 6.5**

then, $$\begin{bmatrix} Y_0 \\ Y_2 \\ Y_4 \\ Y_6 \end{bmatrix} = U_{01} + U_{23}$$ - **Equation 6.6**

where $U_{01} = U_0 + U_1$ and $U_{23} = U_2 + U_3$ - **Equation 6.7**

Similarly for Equation 6.3, let

$$L_0 = \frac{1}{2}\begin{bmatrix} D \\ E \\ F \\ G \end{bmatrix} \bullet [Z_4],\ L_1 = \frac{1}{2}\begin{bmatrix} E \\ -G \\ -D \\ -F \end{bmatrix} \bullet [Z_5],\ L_2 = \frac{1}{2}\begin{bmatrix} F \\ -D \\ G \\ E \end{bmatrix} \bullet [Z_6],\ and\ L_3 = \frac{1}{2}\begin{bmatrix} G \\ -F \\ E \\ -D \end{bmatrix} \bullet [Z_7]$$ - **Equation 6.8**

then, $$\begin{bmatrix} Y_1 \\ Y_3 \\ Y_5 \\ Y_7 \end{bmatrix} = L_{01} + L_{23}$$ - **Equation 6.9**

where $L_{01} = L_0 + L_1$ and $L_{23} = L_2 + L_3$ - **Equation 6.10**

Based on Equation 6.1 and Equation 6.4 to Equation 6.10, the architecture of the 1D DCT core is formed as shown in Figure 6.7. It is a fully pipelined design and the datapath is simple and in single direction without any feedback. In this architecture, the pre-processor is constructed by one adder and subtractor, the multiplier-accumulator consists of two general purpose multipliers and three adders.

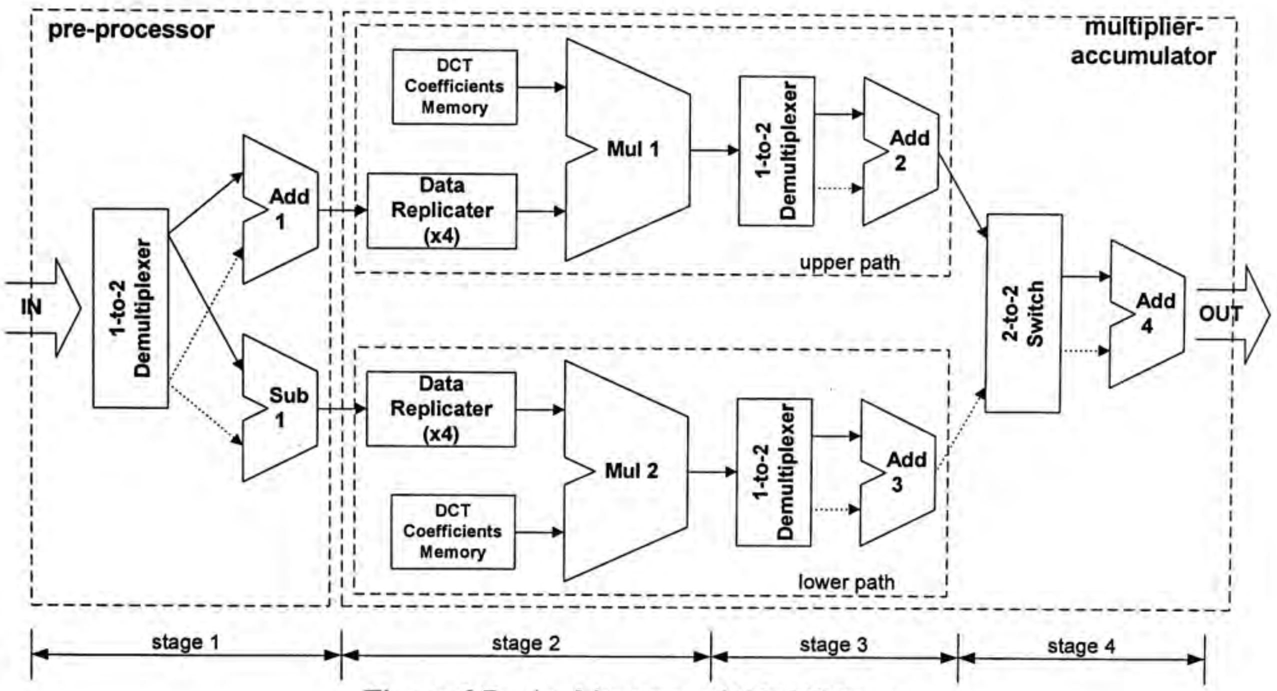


Figure 6.7 – Architecture of 1D DCT core

It should be noticed that only two multipliers are used in this design. In order to achieve a high performance, some other designs require parallel input of data or require four or more multipliers or LUTs [32][34]. In this way, several multiplications can be processed in parallel such that a higher throughput can be achieved. Another reason is that they are synchronous designs, they need to maintain a constant data rate throughout the datapath. Otherwise, some clock cycles may be wasted for waiting the input data. However it is not necessary in this design as it is based on an asynchronous architecture. Different units in the asynchronous design can operate at different rates as their operations based only on the local handshake signals rather than the global clock signal. Also, the asynchronous pipelined architecture is applied on the design of the multipliers such that the multiplier can be run very fast. As a result, the multipliers can be adjusted to run faster than the other units, then a similar or better performance can still be achieved by this design even less multipliers are used. Furthermore, it does not require parallel input of data as the operation will only be started when all the inputs are ready, no operation (no power is consumed) will occur while waiting the input data.

6.2.1.2 Flow of Operation

The dataflow of the 1D DCT core can be explained by Equation 6.1 and Equation 6.4 to Equation 6.10. Figure 6.7 can be divided into four stages.

Stage 1:

Stage 1 is the operation of the pre-processor, represented by Equation 6.1. Firstly it receives the input data in the order [x0, x7, x1, x6, x2, x5, x3, x4], and then the one-to-two demultiplexer will send the data to input1 and input2 of the adder and subtractor alternatively, that means the odd-th input data will be sent to input 1 of the adder and subtractor, and the even-th input data will be sent to the lower path input 2 of the adder and subtractor. As a result, the output sequence of the adder 1 is [x0+x7, x1+x6, x2+x5, x3+x4] or [Z_0, Z_1, Z_2, Z_3] (refer to Equation 6.1) and the output sequence of the subtractor is [x0-x7, x1-x6, x2-x5, x3-x4] or [Z_4, Z_5, Z_6, Z_7] (refer to Equation 6.1)

Since the addition and subtraction can only be carried out when both inputs are ready, the output rate of the adder and subtractor is half of the input data rate.

Stage 2:

The multiplications with DCT coefficients are performed at stage 2. At this stage, data is split into two paths, which are the upper and lower path. Both of the paths are totally identical, and the upper path is responsible for Equation 6.5 while the lower path is responsible for Equation 6.8.

By considering Equation 6.5, there are totally sixteen multiplications, in which each input data needs to multiply with four different DCT coefficients. Therefore, a data replicator is used to duplicate the input data four times and then send to the multiplier. Therefore, the output sequence of the data replicator at the upper path is [Z_0, Z_0, Z_0, Z_0, Z_1, Z_1, Z_1, Z_1, Z_2, Z_2, Z_2, Z_2, Z_3, Z_3, Z_3, Z_3]. Similarly, the output sequence of the data replicator at the lower path is [Z_4, Z_4, Z_4, Z_4, Z_5, Z_5, Z_5, Z_5, Z_6, Z_6, Z_6, Z_6, Z_7, Z_7, Z_7, Z_7].

The DCT coefficients are stored in the DCT coefficients memory, and they are arranged and sent out to the multipliers in the sequence of [A, B, A, C, A, C, -A, -B, A, -C, -A, C, A, -B, A, -C] in the upper path and [D, E, F, G, E, -G, -D, -F, F, -D, G, E, G, -F, E, -D] in the lower path. As a result, Equation 6.5 and Equation 6.8 can be performed and the output sequence of the multiplier 1 is [U_0^t, U_1^t, U_2^t, U_3^t] while the output sequence of multiplier 2 is [U_4^t, U_5^t, U_6^t, U_7^t]. The output data rate of each multiplier is two times of the input data rate as the data replicator increases the output data rate of the first stage by four times.

Stage 3:

The output of the multiplier will go through the one-to-two demultiplexer at stage 3. Its operation is similar to that of stage 1, but the outputs of the demultiplexer are alternating every 4 times in order to perform the addition shown in Equation 6.7 and Equation 6.10. For example in the upper path, U_0^t and U_2^t will connect to the first input of the adder 2, U_1^t and U_3^t will connect to the second input of adder 2. Therefore the output sequence of adder 2 is $[U_{01}^t, U_{23}^t]$ and that of adder 3 is $[L_{01}^t, L_{23}^t]$. For the output data rate, it is reduced to be the same as the input data rate. This is because an addition can only be performed when both inputs are ready, it is reduced to half of the output data rate of the multipliers.

Stage 4:

Stage 4 is responsible for performing Equation 6.6 and Equation 6.9. Originally two adders are required for each equation. However, data rate after stage 3 is halved, a single adder can be shared by both equations. Therefore a two-to-two switch is inserted at the beginning of this stage. It is used to collect data from upper and lower paths and distribute them to adder 3. Finally the output sequence of stage 4 is $[Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7]$.

In this stage, the output data rate can be maintained at the input data rate as it combined the data from the upper and lower paths. As a result, the final output of the 1D DCT core has the same data rate as the input.

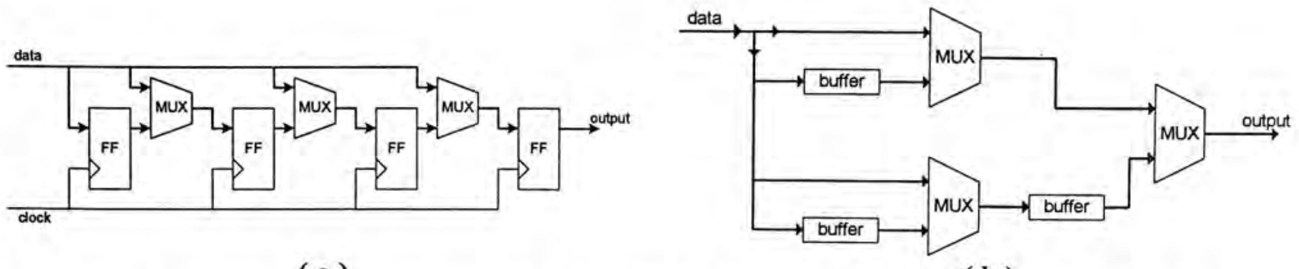
Table 6.1 shows the summary of data rate at different stages of the 1D DCT core. It shows that the critical part of the design is in stage 2, where the multiplications are performed. As a result the throughput of the whole design is limited to the half of the speed of the multiplications in stage 2.

	Stage 1	Stage 2	Stage 3	Stage 4
Input	Input data rate	½ × Input data rate	2 × Input data rate	Input data rate
Output	½ × Input data rate	2 × Input data rate	Input data rate	Input data rate

Table 6.1 – Data rate at different stages of the 1D DCT core

6.2.1.3 Data Replicator

The purpose of the data replicator is used to keep a single operand for the multiplier to perform four multiplications. In synchronous design, a latch can be used to hold a data for four clock cycles. However, it is not possible in asynchronous design as data will be lost after used due to the Precharge phase of the domino logic. A simple way which uses a buffer with a feedback output can perform a cyclic function and the data can be reused. However, the resultant speed is slow and the pipeline architecture is destroyed due to the feedback. As a result, a dedicated circuit is constructed in order to duplicate a single data four times, and thus four multiplications on a single data can be done.



(a) (b)

Figure 6.8 – (a) block diagram parallel-to-serial shift register in synchronous design, (b) block diagram of data replicator

The idea of the data replicator is similar to that of the parallel-to-serial shift register in synchronous design, which is shown in Figure 6.8(a). However, it is not suitable to implement the parallel-to-serial shift register in asynchronous design. The first reason is that flip-flop does not fit the style of the new asynchronous architecture. Also the single stage parallel-to-serial structure requires more difficult control.

In the data replicator, which the shown in Figure 6.8(b), multiplexers are used instead of flip-flops and the structure is divided into two stages. Since the multiplexers can only handle one of the two inputs every time, buffers are also included for temporarily storage purpose. In this structure, data in the four input paths will quickly be transferred to the next stage by the multiplexers or stored in the buffers, and then the next data can be inputted. However, it is necessary to ensure that all data must be sent out in the single stage parallel-to-serial shift register before next data comes in. Therefore, the control of the data replicator is simpler and has less overhead, and thus allows faster duplication on data.

6.2.1.4 DCT Coefficients Memory

As mentioned in the previous section about the distributed arithmetic, ROM is not suitable to be used in asynchronous design. In order to pre-store the DCT coefficients for the multiplication, a new logic cell is used, which is shown in Figure 6.9(b).

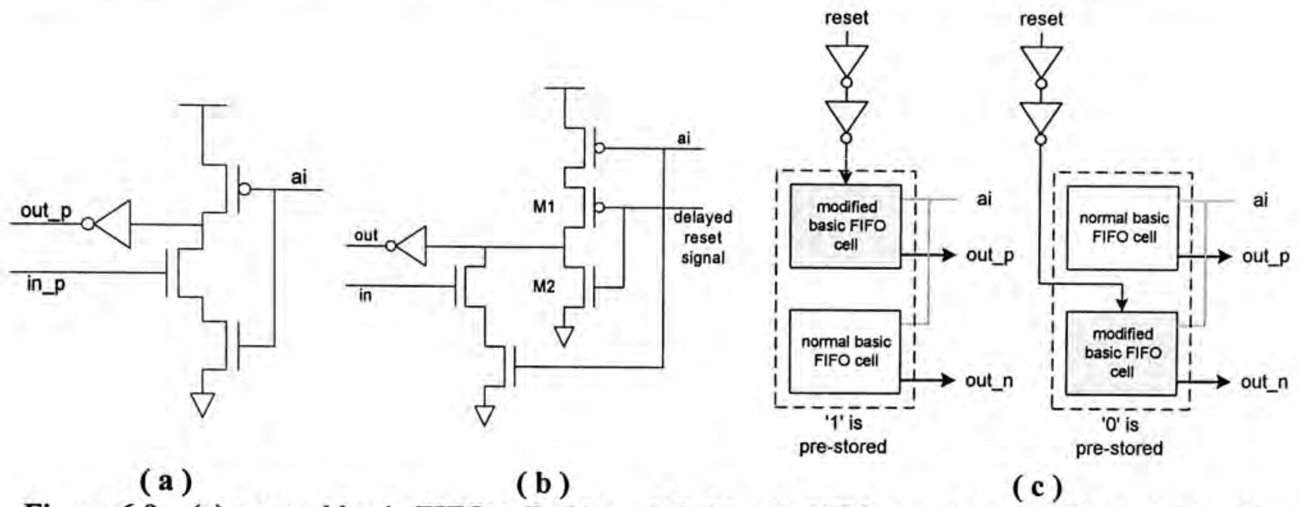


Figure 6.9 – (a) normal basic FIFO cell, (b) modified basic FIFO cell, (c) basic DCVSL structure of pre-storing data

The main difference between the new cell and the normal basic FIFO cell, which is shown in Figure 6.9(a), is the addition of transistors M1 and M2. Initially when the system is being resetted, the reset signal is high and the acknowledgement signal becomes low. For the normal basic FIFO cell, it enters the Precharge phase and the output becomes logic low. After the reset has finished, it will go into the Enable phase and wait for the input data. Since charge is still kept at the floating node, the output of the normal basic FIFO cell is still kept low.

However for the modified basic FIFO cell, although the acknowledgement input is low, it is not in the Precharge phase as transistor M1 is turned off and M2 is turned on by the delayed reset signal. As a consequence, a pull-down path is created and the output is kept in high. When reset is finished, the next stage will go to the Enable phase but the output of the modified basic FIFO cell is still kept high due to the delayed reset signal. As a result, this high output, which presents having a data of logic one, requests the next stage to receive the data. As a result, a data of logic one can be sent out. Owing to this feature, the modified basic FIFO cell can be treated as a memory cell of either pre-storing logic high or low, as shown in Figure 6.9(c). By connecting N memory cells in parallel with a handshake cell, it forms a single FIFO stage in which a N-bit data is pre-stored. The DCT coefficients memory is constructed from these FIFO stages with pre-stored DCT coefficients according to the required sequence listed in section 6.2.1.2. An example of the DCT coefficients memory in the upper path is shown in Figure 6.10.

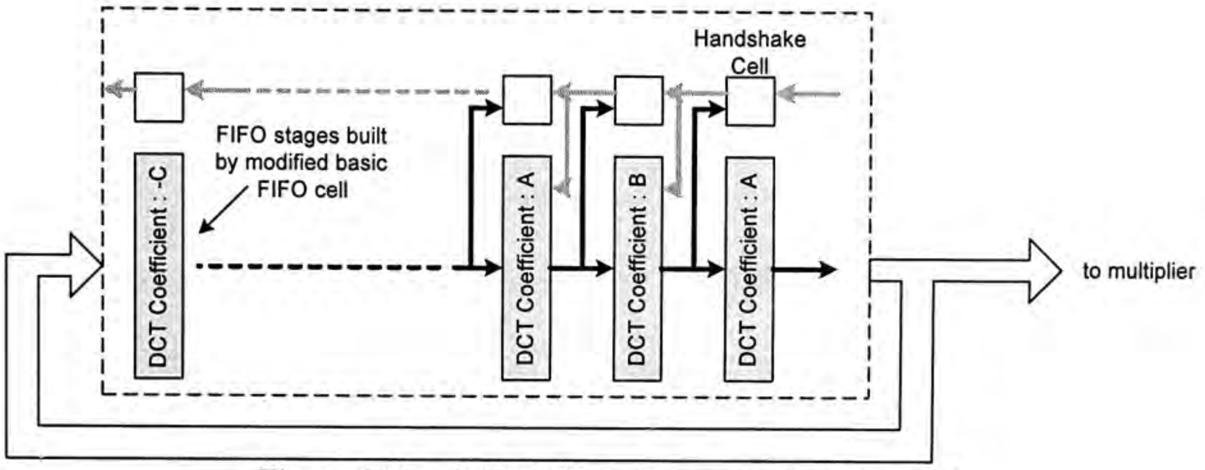


Figure 6.10 – DCT coefficients memory in upper path

When the delayed reset signal becomes zero, then transistor M1 is always turned on while M2 is always turned off. As a result, the modified basic FIFO can be treated as a normal basic FIFO cell. Therefore by applying the cyclic feature in this DCT coefficients memory as shown in Figure 6.10, the DCT coefficients can be recycled and can be used repeatedly.

6.2.2 Combination of IDCT to 1D DCT core

Similar to the 1D DCT, the 1D IDCT can also be implemented by similar architecture. Referring to Equation 3.7 and Equation 3.8, they can be divided into two stages as the following equations,

$$\begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} A & B & A & C \\ A & C & -A & -B \\ A & -C & -A & B \\ A & -B & A & -C \end{bmatrix} \bullet \begin{bmatrix} Y_0 \\ Y_2 \\ Y_4 \\ Y_6 \end{bmatrix} \text{ and } \begin{bmatrix} S_4 \\ S_5 \\ S_6 \\ S_7 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} D & E & F & G \\ E & -G & -D & -F \\ F & -D & G & E \\ G & -F & E & -D \end{bmatrix} \bullet \begin{bmatrix} Y_1 \\ Y_3 \\ Y_5 \\ Y_7 \end{bmatrix} \quad \textbf{- Equation 6.11}$$

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} S_0 + S_4 \\ S_1 + S_5 \\ S_2 + S_6 \\ S_3 + S_7 \end{bmatrix} \text{ and } \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} S_0 - S_4 \\ S_1 - S_5 \\ S_2 - S_6 \\ S_3 - S_7 \end{bmatrix} \quad \text{- Equation 6.12}$$

Referring to Equation 6.11 and Equation 6.12, the operations of the 1D IDCT are similar to that of 1D DCT, but different in order. The 1D IDCT first requires a matrix multiplication, and then followed by addition and subtraction. Therefore for IDCT, the pre-processor is eliminated while a post-processor is added after the multiplier-accumulator, which is shown in Figure 6.11. This post-processor is responsible for the operation according to Equation 6.12, and it consists of an adder and subtractor which is the similar as the pre-processor.

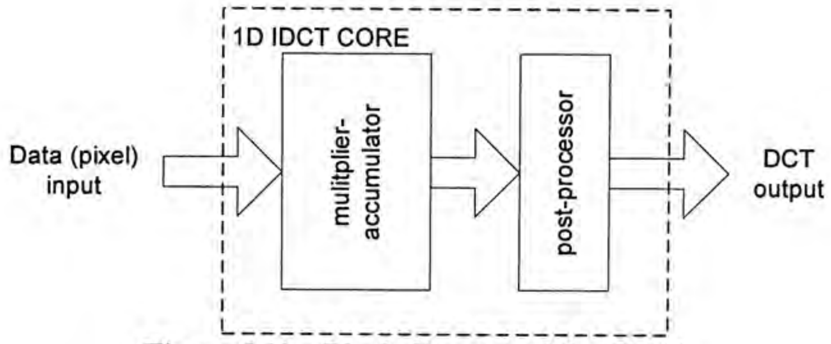


Figure 6.11 – Block diagram of the IDCT

By comparing Equation 6.11 to Equation 6.2 and Equation 6.3, their structures are the same and thus both the multiplier-accumulators can share on the same hardware. Therefore, the 1D IDCT can also be performed on the 1D DCT core by adding a post-processor at the end of the original 1D DCT core. Switches are also inserted inside the core so as to select the path for performing DCT or IDCT. As a result, the overall architecture of the 1D DCT/IDCT processor is shown in Figure 6.12

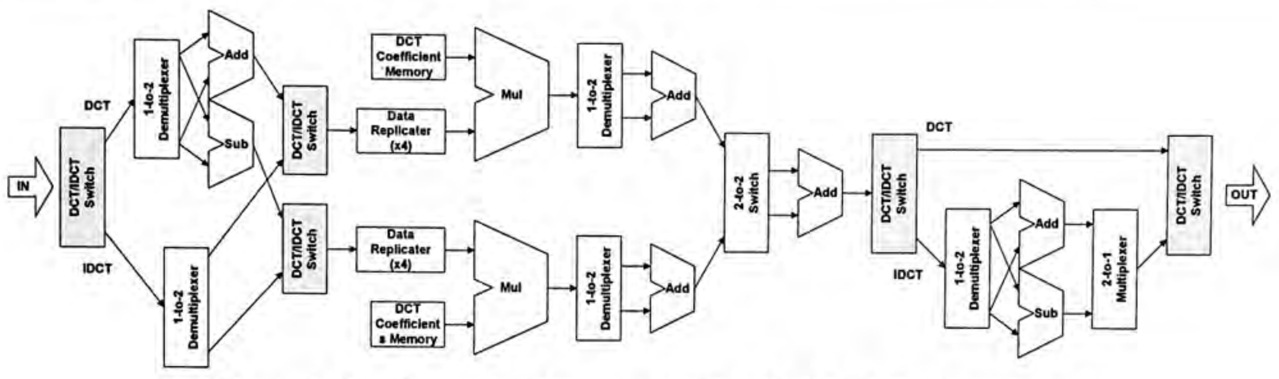


Figure 6.12 – Overall architecture of the 1D DCT/IDCT processor

One more modification is made on the DCT coefficients memory. The DCT coefficients of DCT and IDCT are different in the upper path as the matrix multiplications are different, which is shown in Equation 6.2 and Equation 6.11. Therefore the content of the DCT coefficient memory needs to be changed when performing IDCT. As the pre-storage of the modified basic FIFO cell only depends on the delayed reset signal, it is not necessary to use an additional DCT coefficients memory to store the additional IDCT coefficient. The change of the DCT coefficients can be done by adding some logic gates to control the presence of delayed reset signal in the memory cell, which is shown in Figure 6.13. As a result, the pre-storing data can be changed for DCT and IDCT.

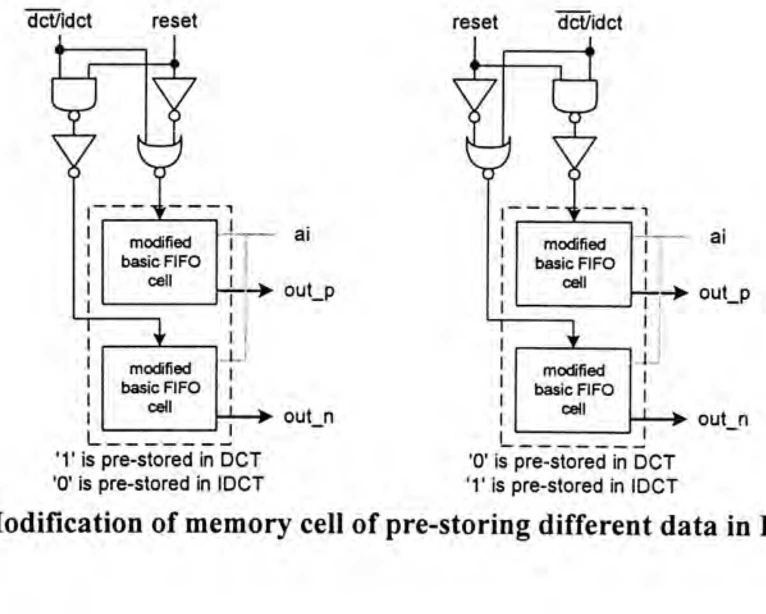


Figure 6.13 – Modification of memory cell of pre-storing different data in DCT and IDCT

In performing the IDCT, the order of the input and output sequence is changed too. The input sequence of the IDCT is [Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7] and the output sequence is [x_0, x_7, x_1, x_6, x_2, x_5, x_3, x_4].

6.2.3 Accuracy

According to the IEEE specification [56], the 2D IDCT should achieve certain accuracy in order to prevent the quality degradation in the reconstructed signal after the inverse transform. Therefore in this design, the bit length of the different parts should be considered in order to achieve the specified accuracy.

By considering different combinations of the bit length of the DCT coefficient, transpose memory and multiplier's output with the verification of the C program, Table 6.2 shows the bit length of the different parts of the DCT/IDCT processor. The architecture is shown in Figure 6.14. According to this result, truncations are needed at the outputs of the multipliers and the 1D DCT/IDCT core. Truncation on the multiplier's output can be merged inside the multiplier as the last stage of the bit-parallel multiplier is an adder, only a little modification is required. However for the output of the DCT/IDCT core, a dedicated circuit for the truncation is added in order to truncate and round up the result.

	Bit length
Input	9/12(DCT/IDCT)
Output	12/9(DCT/IDCT)
Multiplier's output in the row operation	19
Transpose Memory	15
Multiplier's output in the column operation	20

Table 6.2 – Bit length in different parts of the 2D DCT/IDCT processor

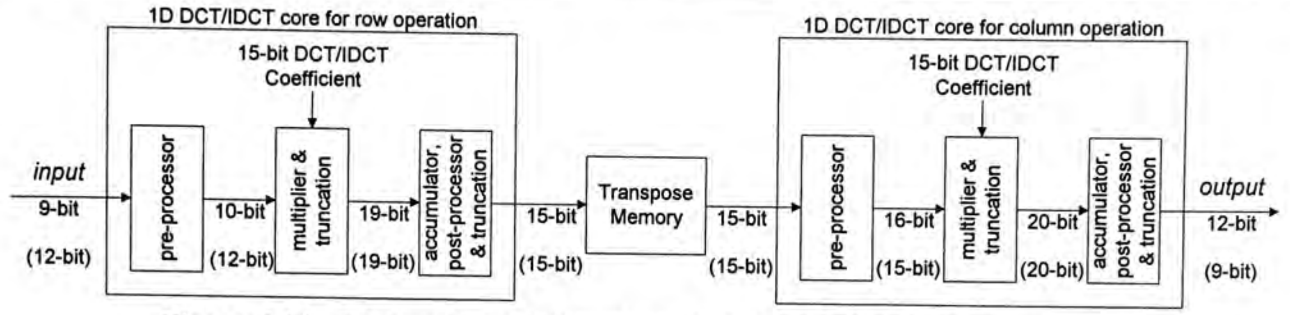


Figure 6.14 – Bit length in different parts of the 2D DCT/IDCT processor

Table 2 shows the IDCT error produced by using the architecture shown in Figure 6.12 with the bit length provided in Table 6.3. It shows that the precision meets the IEEE specification.

	Spec.	Error [-256, 255]	Error [-5, 5]	Error [-300,300]
Maximum Pixel Error	1	1	1	1
Overall Mean Error	0.0015	0.000777	0.000856	0.000675
Overall Mean Square Error	0.02	0.009842	0.009237	0.008331
Maximum Pixel Mean Error	0.015	0.004300	0.004400	0.004700
Maximum Pixel Mean Square Error	0.06	0.012300	0.012600	0.010600

Table 6.3 – Accuracy of the 2D DCT/IDCT processor

In the VLSI implementation of the 2D DCT/IDCT processor, in order to reduce the cost, the 1D DCT/IDCT core and the transpose memory are separated into two chips. The structure of the 1D DCT/IDCT core for the row and column operation is unified such that a single 1D DCT/IDCT core can be used for both operations, and the 2D DCT can be done by cascading the 1D DCT/IDCT core with the transpose memory, and then connect to another 1D DCT/IDCT core again. As a result the bit length is further modified in the 1D DCT/IDCT core as shown in Figure 6.15. In this new configuration, the unified DCT/IDCT core can perform four different mode of operation, which is listed in Table 6.4. The unused bits at the input are needed to fill with zero while the unused output bits can be ignored.

DCT / IDCT	Row/Column Operation	Number of Input Bit	Used Range of Input Data Bus	Number of Output Bit	Used Range of Output Data Bus
DCT	Row	9	Datain[14:6]	15	Dataout[14:0]
DCT	Column	15	Datain[14:0]	12	Dataout[14:3]
IDCT	Row	12	Datain[14:3]	15	Dataout[14:0]
IDCT	Column	15	Datain[14:0]	9	Dataout[14:6]

Table 6.4 – Four different operation modes of the unified ID DCT/IDCT core

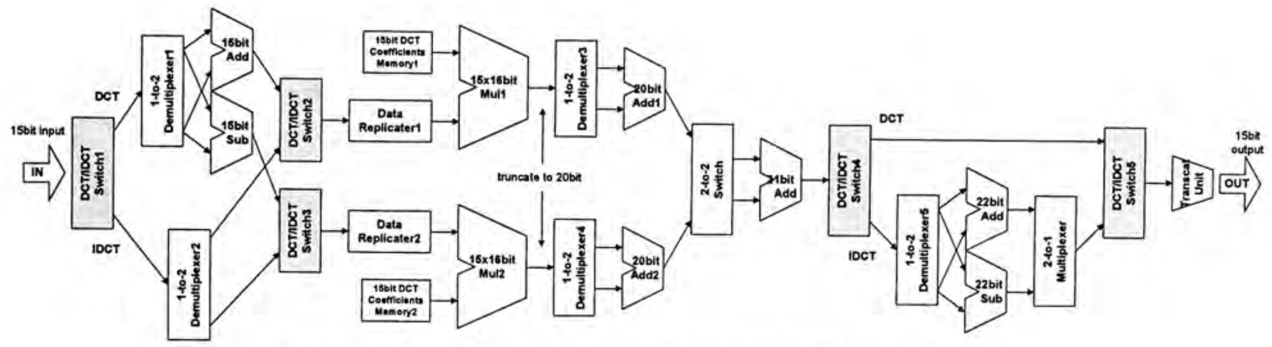


Figure 6.15 – Unified structure of 1D DCT/IDCT core

The result and performance of this 1D DCT/IDCT core will be given in chapter 7.

6.3 Transpose Memory

The purpose of the Transpose Memory is to store the result of the row operation, then re-order the data and send them out for the column operation. The name "transpose" means that the re-ordering is similar to the transpose of matrix, in which the data in the rows and columns are exchanged. In order to be used in the 2D 8×8 DCT/IDCT operation, the transpose memory should be capable of storing 64 15-bit data, and re-ordering the data for the column operation.

In order to fit the architecture of the 1D DCT/IDCT core, the transpose memory is required to have two different modes of operation. This is because the input and output sequences are different in the DCT and IDCT operation in the proposed 1D DCT/IDCT core, which has been mentioned in section 6.2.1.2 and 6.2.2. The transpose memory should be able to rearrange the data in two different orders such that the rearranged data sequence fits the corresponding operation.

To avoid changing both the write and read order at the same time, the write order of the transpose memory in both operations are set to be the same. The output data of the 1st stage 1D DCT/DCT core (row operation) is configured to be written into the transpose memory in row-wise order, which is shown in Figure 6.16.

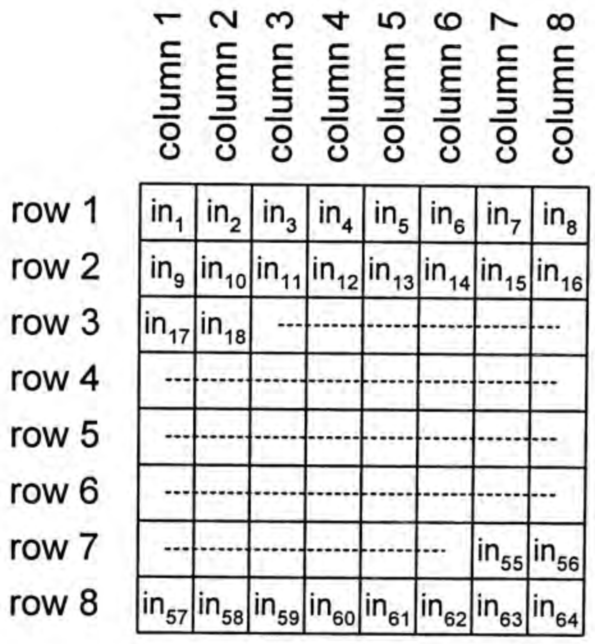

column 1
column 2
column 3
column 4
column 5
column 6
column 7
column 8
row 1
row 2
row 3
row 4
row 5
row 6
row 7
row 8
in_1 in_2 in_3 in_4 in_5 in_6 in_7 in_8
in_9 in_{10} in_{11} in_{12} in_{13} in_{14} in_{15} in_{16}
in_{17} in_{18}
in_{55} in_{56}
in_{57} in_{58} in_{59} in_{60} in_{61} in_{62} in_{63} in_{64}

Figure 6.16 – Write order of the transpose memory

As the write order is fixed, the read order of the data from transpose memory should be altered according DCT or IDCT operation. No matter in which mode of operations, the data are outputted in column-wise order from the transpose memory. However, their orders are not the same as shown in Figure 6.17.

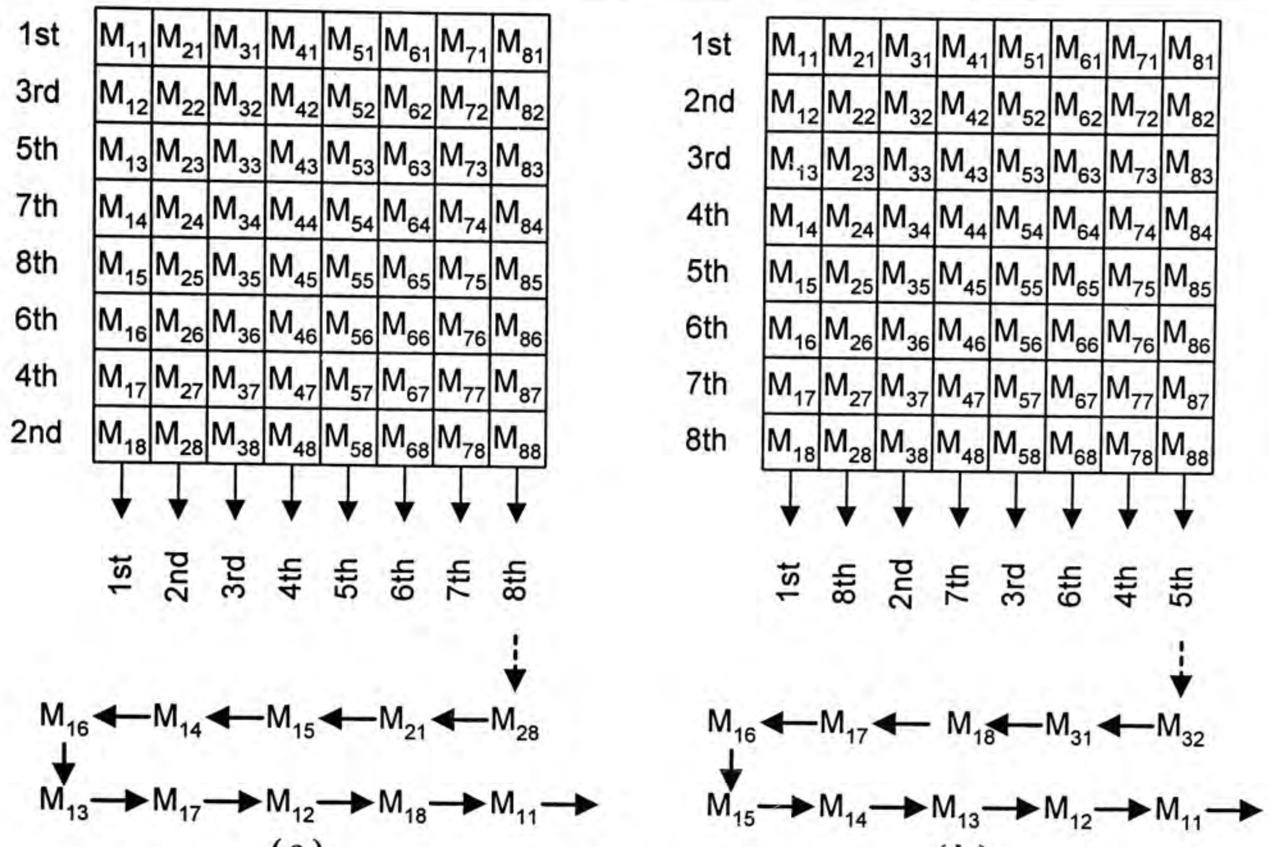

1st
3rd
5th
7th
8th
6th
4th
2nd
1st
2nd
3rd
4th
5th
6th
7th
8th
M16 ← M14 ← M15 ← M21 ← M28
M13 → M17 → M12 → M18 → M11 →
(a)
1st
2nd
3rd
4th
5th
6th
7th
8th
1st
8th
2nd
7th
3rd
6th
4th
5th
M16 ← M17 ← M18 ← M31 ← M32
M15 → M14 → M13 → M12 → M11 →
(b)

Figure 6.17 – Read order of (a) DCT operation, (b) IDCT operation

6.3.1 Architecture

Figure 6.18 shows the block diagram of the transpose memory. It consists of a write/read address generator, two RAM blocks and two multiplexing networks. Although it is only required to store 64 data, two 64×15-bit RAM blocks are used in this design. This is because if a single 64×15-bit RAM is used, the second row operation cannot be started immediately after the first row operation as data are still stored inside the RAM for the column operation, data cannot be written into the RAM until the column operation is completed. As a result row and column operation cannot be carried out simultaneously and thus the performance is poor. If two RAM blocks are used, the result of the second row operation can be written into the RAM block1, and the data stored in RAM block0 is used for the column operation. Since the computation time of the row and column operation are the same, the roles of the RAM blocks can be exchanged after the current row and column operation are completed. As a result, both operations can be run simultaneously and the whole 2D DCT/IDCT operations can be run non-stopping.

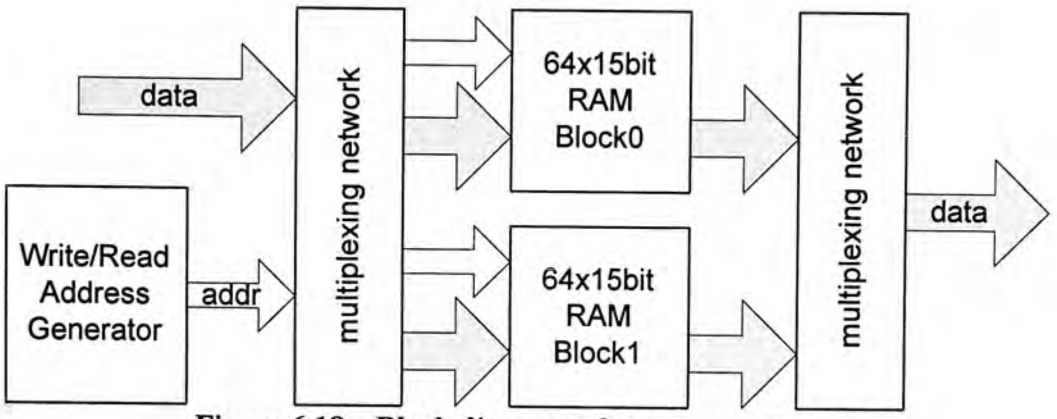


Figure 6.18 – Block diagram of transpose memory

The multiplexing networks are built by multiplexers and demultiplexers. The first multiplexing network is responsible for scheduling the flow of data and address to the two RAM blocks, and thus the write and read operations of the two RAM blocks can be controlled. The second multiplexing network is responsible for detecting and collecting output data from RAM.

In order to further improve the performance of the Transpose Memory, the architecture of the RAM block is further modified as shown in Figure 6.19.

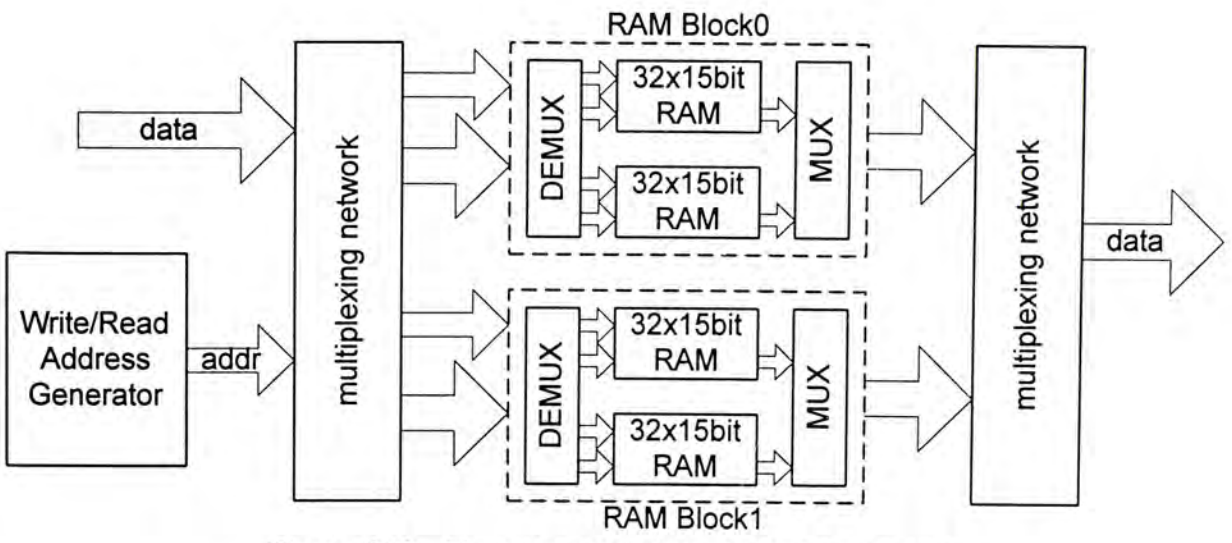


Figure 6.19 – New structure of the transpose memory

In this modification, an interleaving technique is used. The single 64×15bit RAM blocks is replaced by two 32×15bit RAM block with a multiplexer and demultiplexer. In write operation, the demultiplexer delivers the write address and data to two 32×15bit RAM blocks alternatively. As a result, the time allowed for the write operation is doubled due to the interleaving policy, and thus the performance requirement of the 32×15bit RAM block is relaxed. However, area is the trade off of this modification.

6.3.2 Address Generator

The address generator is composed of two units, which are the write address generator and the read address generator. Besides from generating address for the write and read operations in the RAM blocks, they also control the switching of the multiplexing networks.

Since 64 data can be stored in a RAM block, 6-bit RAM address is required as $2^6=64$. The structure of the address generator is similar to the DCT coefficients memory, it uses memory cell to pre-store the RAM address. Instead of directly storing the 64 6-bit addresses, the address is split into 2 parts and pre-stored by two cyclic FIFO memories. An example of the write address generator is shown in Figure 6.20.

address	MSB	LSB
0	000	000
1	000	001
2	000	010
3	000	011
4	000	100
5	000	101
6	000	110
7	000	111
8	001	000
9	001	001
⋮	⋮	⋮
62	111	110
63	111	111

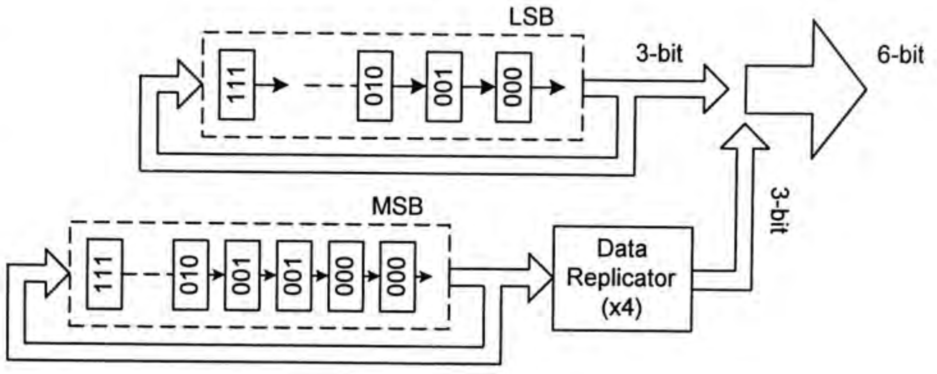


Figure 6.20 – Write address generator

In this configuration, the number of data needed to be stored in the address generator can be reduced and thus the area can be reduced too. Read address generator also has the same architecture as the write address generator. However, the addresses stored in the read address generator are different for the DCT and IDCT operation as their input and output sequences are different, as described in section 6.2.2. The change of the pre-stored addresses uses the same method as the one used in DCT coefficients memory.

Figure 6.21 shows the operation of the transpose memory. Initially the row operation is required to be carried out first, i.e. a write operation on the RAM is required. As a result, the write address generator controls the multiplexer to send the write address to RAM block0. At the same time, it blocks the read address from entering RAM block0 and switches the input data to RAM block0, which is shown in Figure 6.21(a).

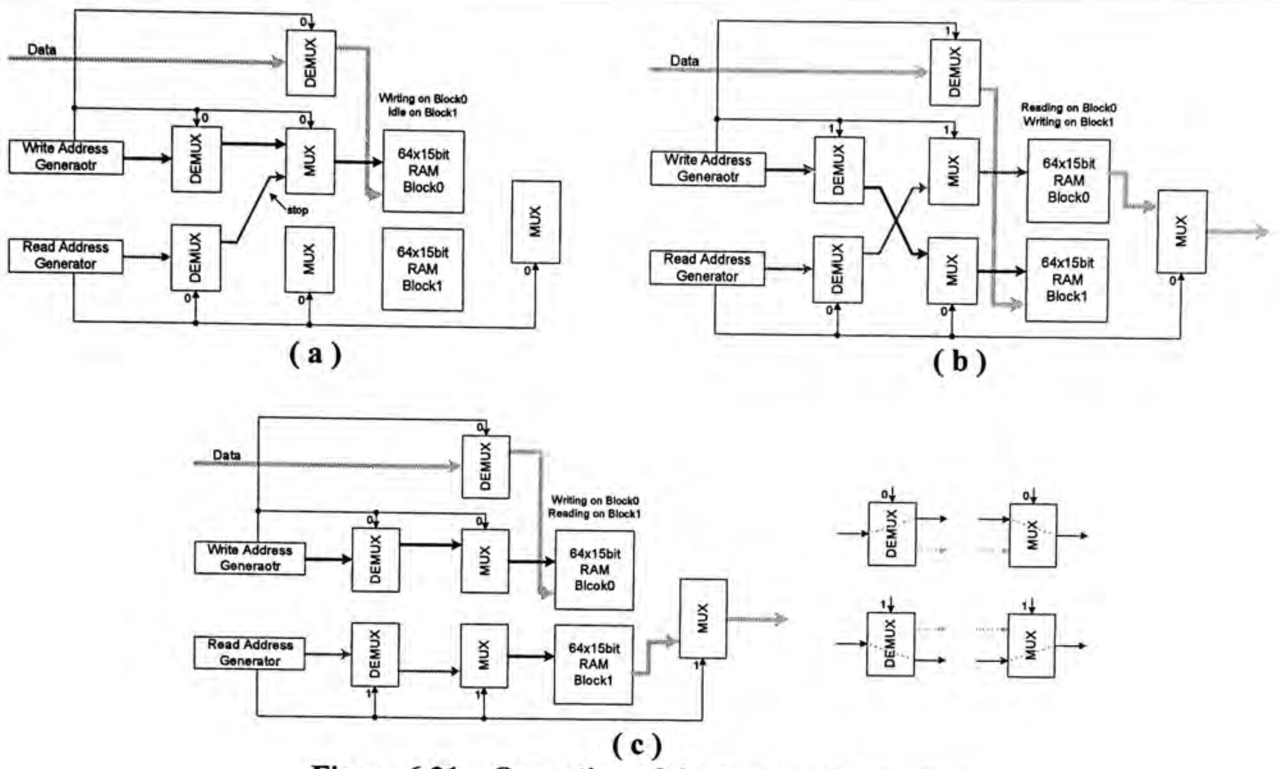


Figure 6.21 – Operation of the transpose memory

After the first row operation is completed, the second row (write) operation and the first column (read) operation are started at the same time. By changing the controlling signal in the multiplexing network, the read and write addresses are transmitted to RAM block0 and block1 respectively, and the output data can be collected at the output side. This is shown in Figure 6.21(b). As a result, both row operation and column operation can be run concurrently. After these two operations are completed, the controlling signals are altered such that the flow of the addresses is changed and the role of the RAM blocks is also changed, as shown in Figure 6.21(c). The controlling signals are altering after every row and column operations, and thus the roles of the RAM blocks are alternating repeatedly. This allows the 2D DCT/IDCT runs continuously and simultaneously without any user's control.

6.3.3 RAM Block

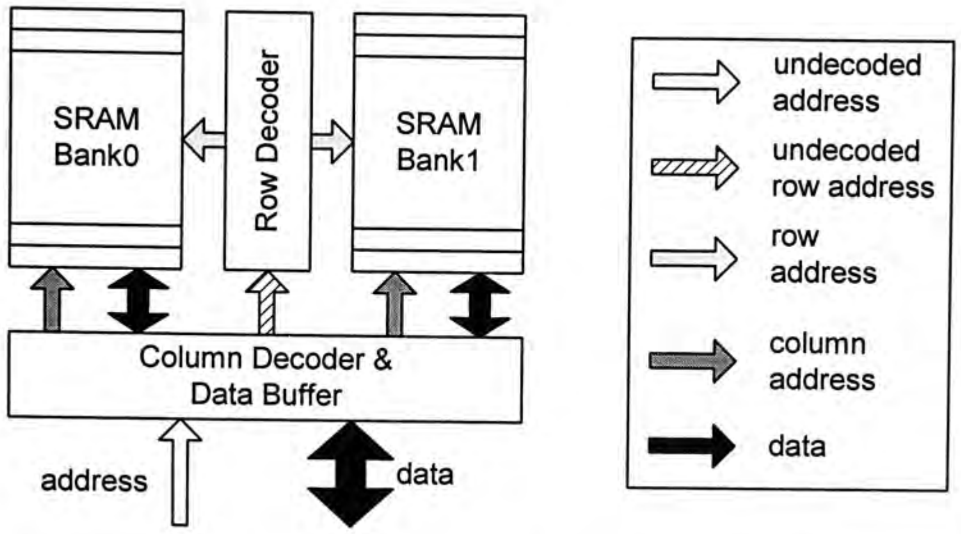


Figure 6.22 - Block diagram of the RAM block

The RAM Block is basically a SRAM. Its structure follows the traditional design which consists of a column address decoder, a row address decoder and 2 SRAM banks. Each RAM bank is capable of storing 16×15 bit data. The structure of the RAM block is shown is shown in Figure 6.22.

There is one difficulty in using SRAM in asynchronous design, which is the completion detection. It may have no problem in the read operation as the presence of the data at the output representing the completion of the read operation. However, there is no related signal representing the completion of the write operation. As a result, additional circuits are required so as to detect the completion.

There are several methods to detect the completion in SRAM. One of the methods is the use of delay element [57]. In this approach, it is assumed that the write operation must be finished within a certain period of time. Therefore a delay element can be used to delay the write request signal, and the delayed write request signal can be acted as the completion signal. Although this method is simple, it provides a worst case performance. Another method is the use of current sensing technique [57][57][59]. Since the current drawn will be decreased after the operation, the completion can be known by sensing the current drawn from the power supply in RAM block. However, this technique is difficult to implement and the result may not be accurate. In our design, a monitor cell is used to detect the completion of the write operation.

The structure of the monitor cell is shown in Figure 6.23(b). It is treated as an additional SRAM basic cell and placed inside the bit column of the SRAM. By comparing with the normal SRAM cell structure, which is shown in Figure 6.23(a), the monitor cell is actually composed of two SRAM basic cells with two additional pMOSs. The purpose of the additional pMOSs is forcing the two SRAM basic cells to store complementary values when the monitor cell is not yet enabled.

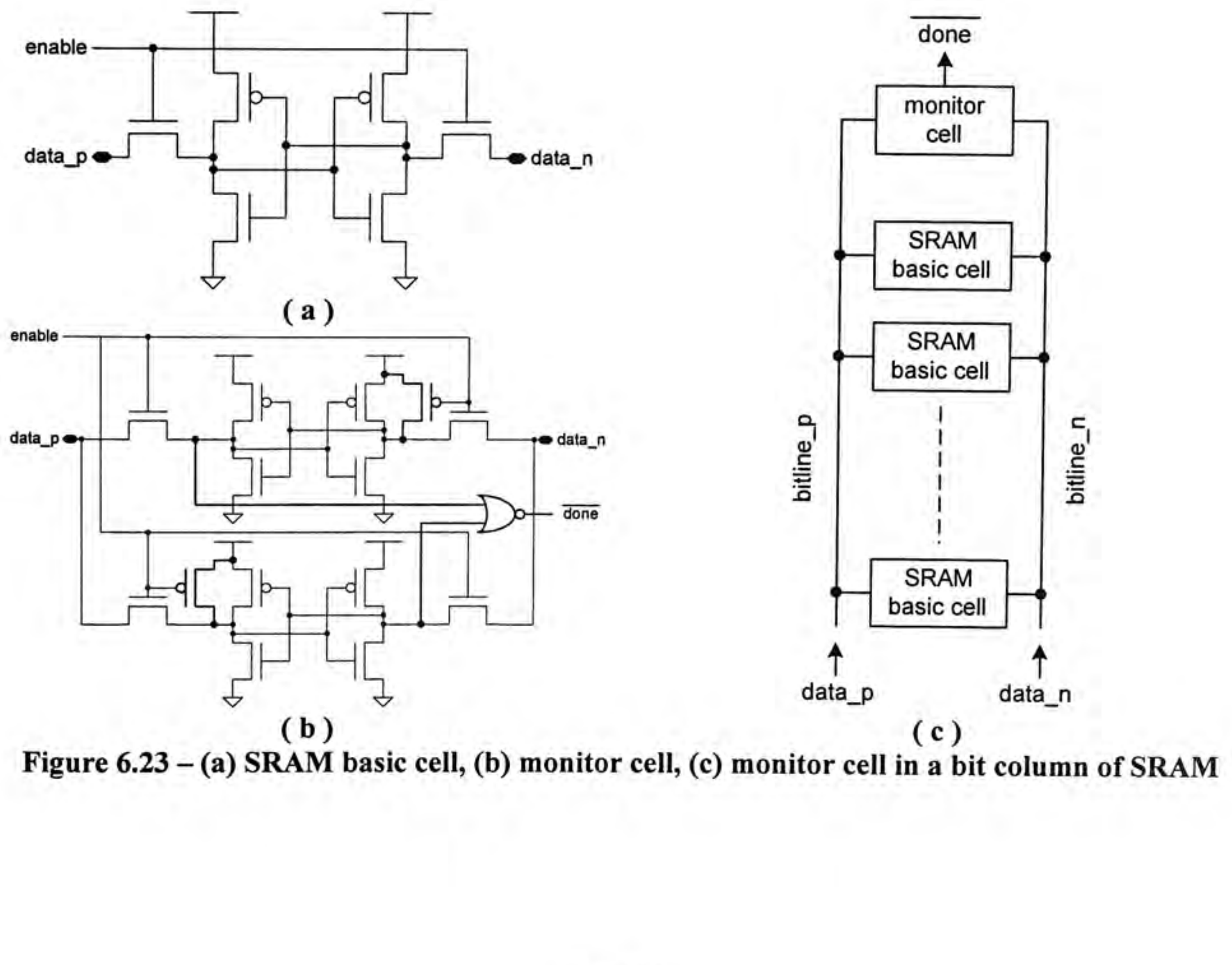


Figure 6.23 – (a) SRAM basic cell, (b) monitor cell, (c) monitor cell in a bit column of SRAM

When there is a write operation, the monitor will be enabled and will perform the write operation. Since the write operation causes both of the SRAM basic cells inside the monitor cell to store the same value, either one of the values in the SRAM basic cell will change and thus the change can be detected by the NOR gate. This signal will be sent out to indicate the completion of the write operation. Due to geometrical reason, the monitor cell is placed at the top of the bit column of the SRAM, as shown in Figure 6.23(c), in order to prevent the monitor cell from being written before the normal SRAM basic cell. Also due to the same reason, the detection of completion is only required on bit0 and bit14 of the SRAM banks as their write operation is the slowest among all the bits.

The advantage of this method is that the monitor cell can be treated a normal SRAM cell which is simply placed in the bit column, it will not cause a large modification in the traditional architecture of RAM block design. Also it directly monitors the write operation, and the completion signal is immediately generated after the write operation is done. As a result, the average case performance can be achieved.

The result and performance of the transpose memory will be given in chapter 7.

Chapter 7

Results and Discussions

7.1 Overview

In this chapter, the implementation results and performance of the designs, which have been described in chapter 4 to 6, will be provided. Based on the results obtained, discussion will be given in each design. The results and discussions will be presented in the order of the Refresh Control Circuit, programmable DSP processor, 1D DCT/IDCT core and transpose memory.

7.2 Refresh Control Circuit

7.2.1 Implementation Results and Performance

The Refresh Control Circuit is designed based on the AMS 3M1P 0.6u CMOS technology. Table 7.1 shows the transistor count on different units of the Refresh Control Circuit.

By the HSPICE simulation under 5V supply voltage, it is shown that the oscillating frequency of the ring oscillator is about 38.1 MHz, which is shown in Figure 7.1. Also the function of the Refresh Control Circuit is verified in the simulation, which is shown in Figure 7.2.

	Transistor Count
Ring Oscillator	128
Timer - Counter	234
Timer - Latch	166
Timer - Comparator	142
Recalibrate Circuit	124
Operation Monitoring Circuit	90
Voltage Sensor	33
Total	917

Table 7.1 – Transistor count on different units of Refresh Control Circuit

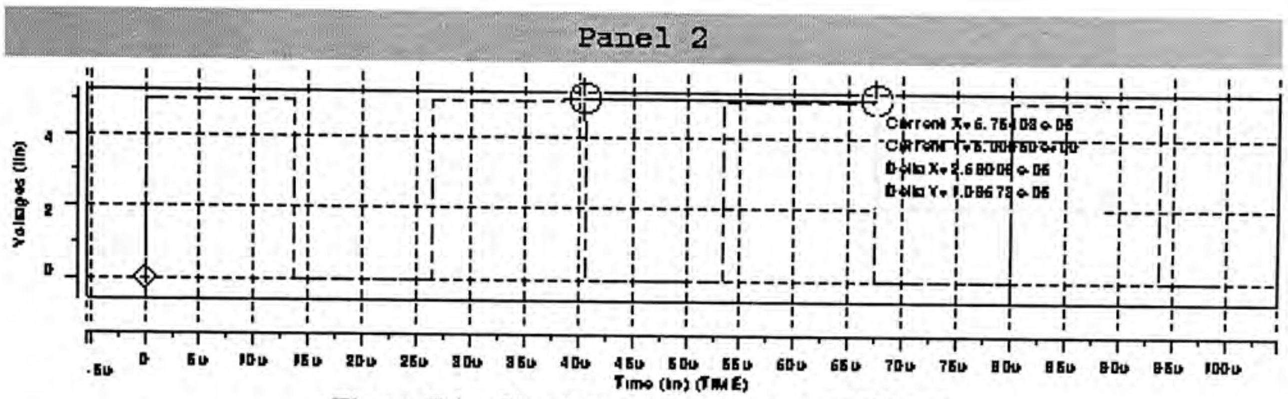


Figure 7.1 – Simulation result of ring oscillator

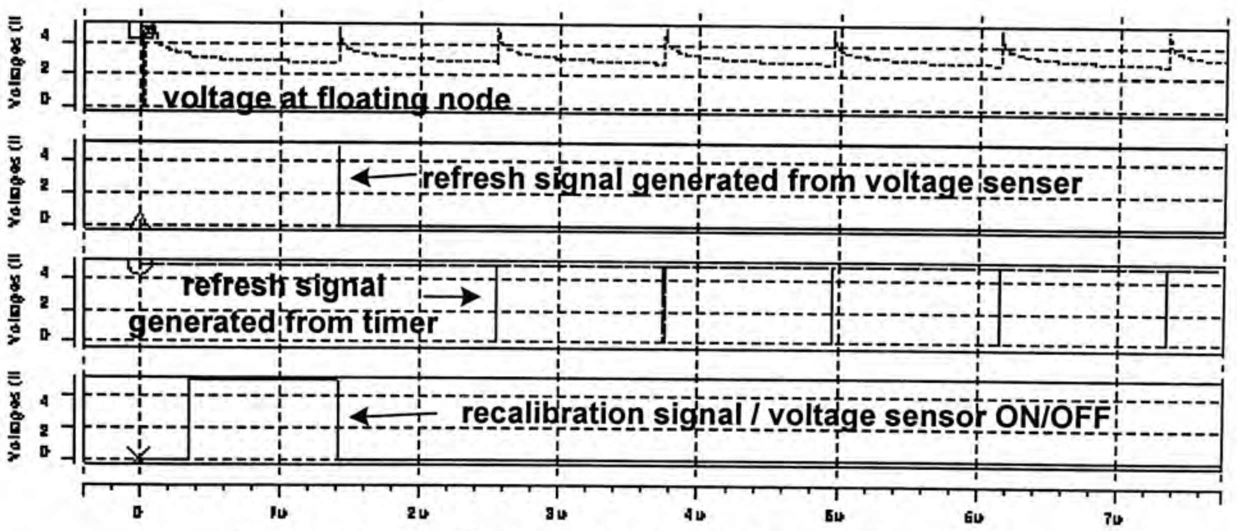


Figure 7.2 – Simulation result of the Refresh Control Circuit

Furthermore, the power consumptions of different parts are estimated from the simulation result, which is shown in Table 7.2. The average current drawn by the whole Refresh Control Circuit is about 15 uA when the voltage sensor is not activated, and is about 3.6 mA when the voltage sensor is enabled. Since the voltage sensor is not operating all the time, the percentage shown in Table 7.2 is not directly proportional to the average current, but is proportional to the current drawn through the whole process.

	Average current	Percentage
Ring Oscillator	12.5053 uA	10.06%
Timer - Counter	0.4901 uA	0.39%
Timer - Latch	0.6007 uA	0.48%
Timer - Comparator	0.2541 uA	0.14%
Recalibrate Circuit	0.1685 uA	0.20%
Operation Monitoring Circuit	0.2662 uA	0.21%
Voltage Sensor*	3.5204 mA	88.5078%
**Average current of voltage sensor when it is enabled*		

Table 7.2 – Current drawn by the each parts of the Refresh Control Circuit

Figure 7.1 shows the simulation result of the ring oscillator. It shows that the ring oscillator can oscillate with a period of around 26 us. Figure 7.2 shows the simulation result of the Refresh Control Circuit and its function verified.

The purpose of developing the Refresh Control Circuit is to reduce the performance degradation due to the pull-up path. In order to investigate the performance improvement from the traditional technique, three multipliers were built so as to provide a comparison. All the multipliers were built in the asynchronous pipeline architecture and based on the bit-parallel algorithm. The first multiplier uses the normal domino logic without any pull-up path. The second one uses the domino logic which a pull-up path as shown in Figure 4.2(b). The last one is using the technique of the Refresh Control Circuit, and its logic structure is shown in Figure 4.4(b).

Three multipliers were simulated by HSPICE under 5V supply voltage. Since they all are asynchronous circuits, the simulations were done by sending inputs to the multipliers continuously, and their intrinsic throughputs and latencies are then measured.

	Throughput	Latency
Without pull-up path	2.8721 ns	18.7925 ns
With pull-up path(traditional)	3.0306 ns (+5.819%)	20.3573 ns (+8.326%)
Refresh Control Circuit	2.9090 ns (+1.105%)	19.0668 ns (+1.459%)

Table 7.3 – Performance of multipliers by different techniques

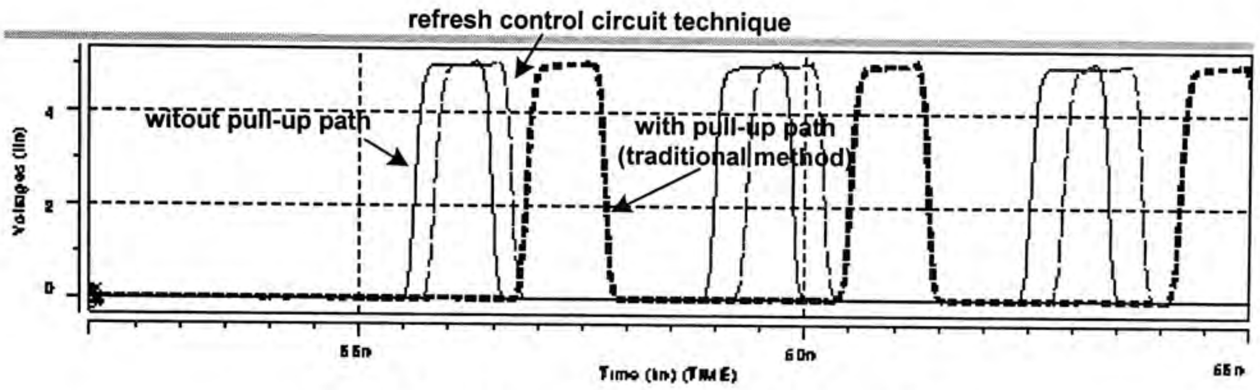


Figure 7.3 – Output signals of different multipliers

Table 7.3 shows the throughput and latency of the three multipliers and Figure 7.3 shows the signal outputs from the different multipliers.

7.2.2 Discussion

From Table 7.1, it shows that the new proposed technique provides a better performance than the traditional technique. It provides less than 2% performance degradation compared with the multiplier using the ordinary domino logic, while the traditional technique degrades the throughput by 5.8% and latency by 8.3%. It indicates that the goal of the Refresh Control Circuit is achieved. It provides a self-timed and reliable method for solving the problem of the charge leakage.

Regarding the power consumption of the circuit, voltage sensor consumes near 90% of the total power. This is because the differential amplifier and sense amplifiers always allow current to flow through. Future work can be focused on minimizing the power consumption of the voltage sensor by using other architectures which have lower power consumption, or using other technique to detect the charge leakage on the floating node of the dynamic logic.

Although all the discussion on the refresh control system is based on the dynamic or domino logic and the comparison is done on the asynchronous circuits, it is not restricted to be used this technique on this area only. Other logic types or circuits which also encounter the charge leakage problem can employ the Refresh Control Circuit technique. However, the method of sensing may need to be modified so as to suit the application.

The disadvantage of this technique is the inclusion of the Refresh Control Circuit in the design, and one or two more transistors are added on each logic cell. As a result the area of the whole system will be increased and the compact property of the dynamic logic is somewhat degraded. It is not suitable to apply the Refresh Control Circuit technique on a compact system with which the area is concerned. However, for a large system and the speed of the operations are concerned, this technique can be employed.

7.3 Programmable DSP Processor

7.3.1 Implementation Results and Performance

In chapter 5, it has shown the dataflow of 1D DCT program on the programmable DSP asynchronous processor. It requires 50 steps for the processor to perform the whole 1D DCT operation. Since the size the processor is very large, it cannot be simulated by HSPICE. On the other hand, all the basic cells were simulated in HSPICE under different loading conditions, and the parameters were extracted to construct a Verilog HDL model for each logic cells. The performance of the DCT implementation is estimated by using the Verilog models to simulate a 9-bit version of the proposed processor. Table 7.4 lists the bit length information of the 9-bit processor.

External	
Primary IO of the programmable DSP processor	9 bits
Instruction Input	10 bits
Internal Functional Units	
Adder	9 bits
Subtractor	9 bits
Multiplier	9 bits (output is truncated to 9 bits)
FIFO Memory	9 bits

Table 7.4 – Bit length information of the 9-bit programmable DSP processor

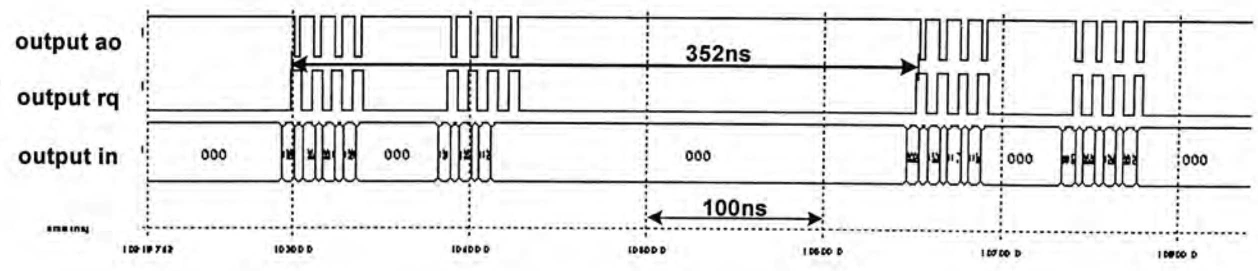


Figure 7.4 – Simulation result of the programmable DSP processor

From the simulation result, the latency between the 8 pixel inputs and the 8 outputs is around 400ns. Since the processor is pipelined, the next DCT operation can be started even the current one is still processing. As a result, the 8-point DCT throughput can reach 352ns, as shown in Figure 7.4. Also, the input frequency is around 130MHz(7ns). Figure 7.5 shows the timing diagram of the DCT operation and Table 7.5 shows the comparison of the 1D DCT core performance with other VLSI implementations.

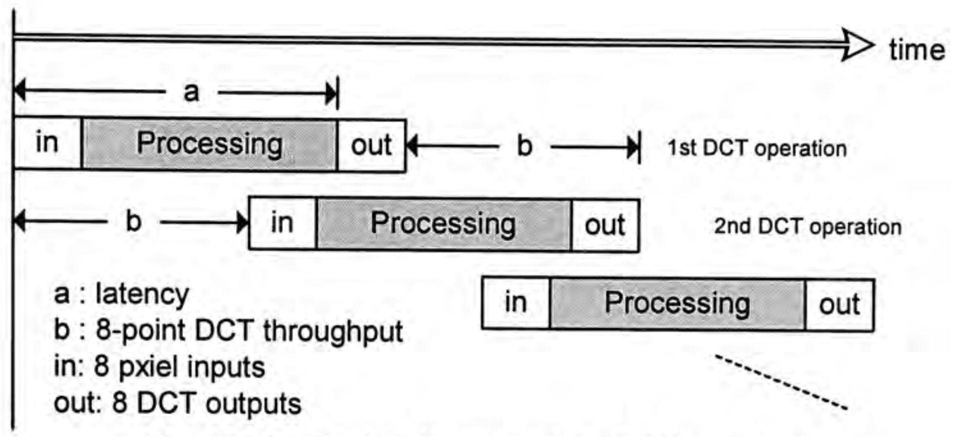


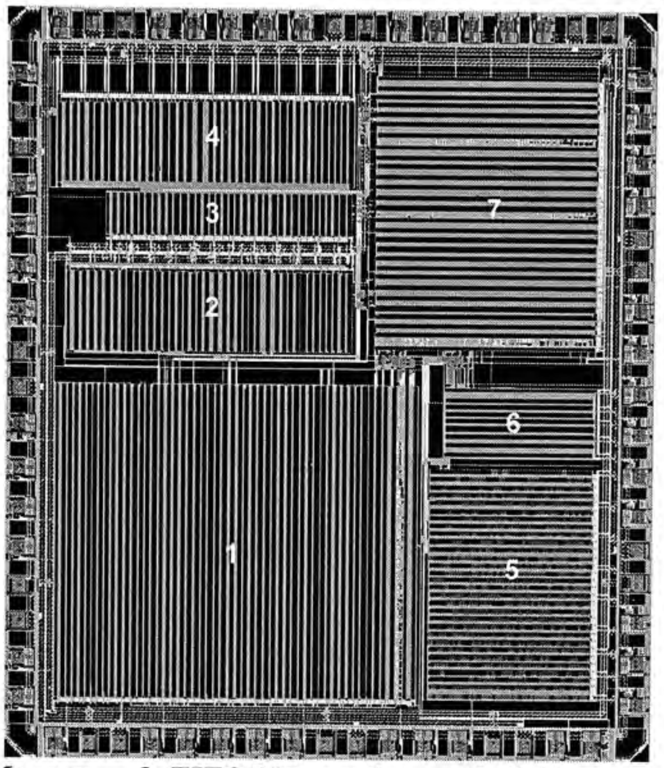
Figure 7.5 – Timing diagram of the DCT operation

Design	Year	Tech.	Processing Unit	Operating frequency (MHz)	Pixel throughput (Mpixel/sec)
Cheng et. al. [35]	2000	0.6u	9 MUL, 21 ADD	100	100
Hsiao et. al. [60]	1999	0.6u	3 MUL, 5 ADD	40	40
Jang et. al. [33]	1994	0.8u	4 MUL, 1 Accumulator, 1 pre- and post processor	100	100
This DSP processor		0.6u	1 MUL, 2 ADD/SUB	/	22.7*

Note : [1] and [3] are 2D DCT chips which use 1D DCT cores and transpose RAM to handle the 2D transform by using the row-and-column decomposition method[8][9]. Normally, the critical path exists in the 1D DCT core as it consists of many arithmetic and control units. Therefore, the speeds of the 1D DCT cores are assumed to be the same as their 2D transform.
**Average result (352ns / 8 = 44 ns/pixel, 1/44 ns = 22.7Mpixel/sec)*

Table 7.5 – Performance comparison of different 1D DCT implementations

The layout of the 9-bit version of the processor is shown in Figure 7.6. It has 153k transistors and is designed by using standard cells based on AMS 3M 1P 0.6u CMOS technology. The core dimension is 4.7mm × 4.2mm.



1. Instruction Memory ***2. FIFO Memory 1*** ***3. Adder*** ***4. FIFO Memory 2***
5. Multiplier ***6. Subtractor*** ***7. Switching Network***

Figure 7.6 – Layout of the 9-bit programmable DSP processor

7.3.2 Discussion

By the comparison with other dedicated designs as shown in Table 7.5, a worse throughput and latency obtained by the general purpose processor with only 3 arithmetic units is understandable as this is a tradeoff of the flexibility. As all the internal arithmetic units are occupied for the current DCT operation, the next 8 pixels can only be sent to the processor nearly at the end of the current operation. This is the reason explaining the slowness of the 8-point DCT throughput. Moreover, there are two main reasons for the large latency. First, the limited number of arithmetic units causes more data queuing. Second, the results from the arithmetic units are required to be fed back to the switching network for the next operation, while in other VLSI implementations, the arithmetic units are directly connected to the next arithmetic units of the following stage. Unfortunately, this cannot be avoided in a general purpose processor. A better latency and throughput of the DCT operation can be achieved if two or more processors are cascaded serially, or more arithmetic units are connected to the switching network. Both changes allow more operations to process concurrently and reduce the data queuing problem. In addition, further improvement can be made in the switch cell in order to reduce its latency.

On the other hand, the high input rate shows that this processor is capable of operating over 100MHz, and it is competitive with other VLSI designs. Also, this frequency indicates the high throughput rate of the switching network. Furthermore, an asynchronous-to-synchronous IO conversion interface is included in this design for the purposes of testing and measuring. If this interface is removed, the performance of the processor can be further improved.

Due to the size of the FIFO memories, the 2D DCT cannot be implemented in this processor. On the other hand, if the size of the FIFO memories is increased or an additional memory unit is added, the 2D DCT can be implemented. Based on this assumption, the performance of the 2D DCT implementation was estimated. By using the row-and-column decomposition method, the 2D DCT can be decomposed into sixteen 1D DCT operations. Therefore, the computation time for 2D DCT operation on this programmable DSP processor can be obtained by Equation 7.1.

$$\begin{aligned} \textit{2D DCT computation time} &= \textit{8-point DCT throughput} \times 16 \\ &= 352ns \times 16 \\ &= 5632\ ns \end{aligned} \qquad \text{- Equation 7.1}$$

Therefore, the average pixel throughput is

$$\begin{aligned} \textit{Average Pixel throughput} &= \textit{2D DCT computation time} \\ &\quad \div \textit{number of pixel} \\ &= 5632ns \div 64 \\ &= 11.3\ Mpixel/sec \end{aligned} \qquad \text{- Equation 7.2}$$

Design	Year	Tech.	Processing unit	Clock (MHz)	Pixel throughput (M pixel/sec)
TI C6201 [38]	/	/	2 MUL, 6 ALU	200	56.63
upd77016 [61]	1993	0.8u	1 MAC, 1 ALU	33	2.6
V830 [62]	1995	/	RISC with 1 MAC	100	9.8
Chang et. al. [38]	2000	0.6	1 ALU	33	1.7
This DSP processor		0.6	1 MUL, 2 ADD/SUB	/	11.3

Table 7.6 – Performance comparison of 2D DCT implementation on different programmable processors

Under the same condition of having limited resources for computing, the results shown in Table 7.6 indicate that this programmable DSP asynchronous processor has a good performance when compared with other general purpose processor designs. This result shows the dataflow architecture and the use of switching network favour the asynchronous processor design, and a competitive performance can be achieved. Future development on this processor can be focused on the 2D DCT operation, or other complex DSP algorithms. Although the estimated 2D DCT performance is good, it still cannot meet the requirement of processing the MPEG-2 or HDTV signal in real-time.

Obviously, this 9-bit processor will introduce a large error and cannot achieve a reasonable accuracy in the DCT operation. A better accuracy can be obtained easily by increasing the word length of the processor. Increase in word length on the switching network and FIFO memory will not cause a performance degradation as they just pass the data without processing. For the arithmetic units which are adder, subtractor and multiplier, increase in the word length causes extra pipeline stages as they are implemented by the carry-look-ahead and bit-parallel algorithm and constructed in the asynchronous pipelined architecture. Thus the throughput will not be affected but the latency will increase. As a result, the increase in word length will not directly affect the throughput of the processor, but the trade-off is the latency and chip size.

7.4 1D DCT/IDCT Core

7.4.1 Simulation Results

Similar to the case of the programmable DSP processor, the whole 1D DCT/IDCT core was failed to be simulated by the HSPICE due to the size problem. As a result, the whole core was simulated by using the Verilog models and the correctness of the operation on the core is verified.

In order to have a more accurate performance analysis on the processing units in the 1D DCT/IDCT core, all the processing units were simulated separately by HSPICE under 5V supply voltage. Due to the limitation of processing power of workstation and HSPICE, only the parasitic within the standard cells is considered while the parasitic information of the routing is not included in the simulation. The simulated performance of different processing units are listed in Table 7.7.

	Tested frequency (MHz)	Required frequency (MHz)
15-bit adder	250	98
15-bit subtractor	250	98
16-bit data replicator	400*	196
DCT coefficient memory	196.07**	196
Multiplier	220***	196
20-bit adder	250	96
21-bit adder	250	96
22-bit adder	250	96
22-bit subtractor	250	96
Truncation unit	250	96
** The output rate of the data replicator* *** Self-generated frequency* ****Transistor-level simulation only*		

Table 7.7 – Performance of different processing units on the 1D DCT/IDCT core

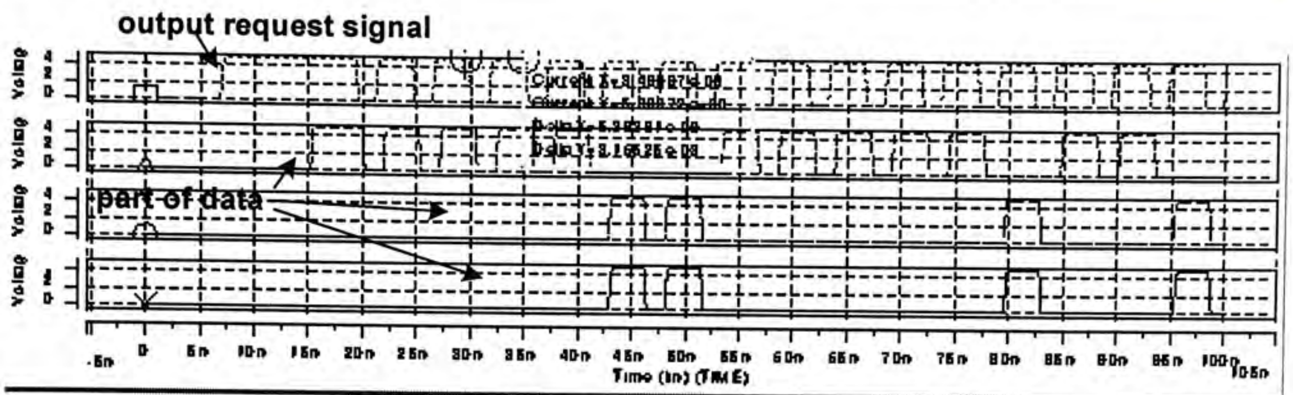
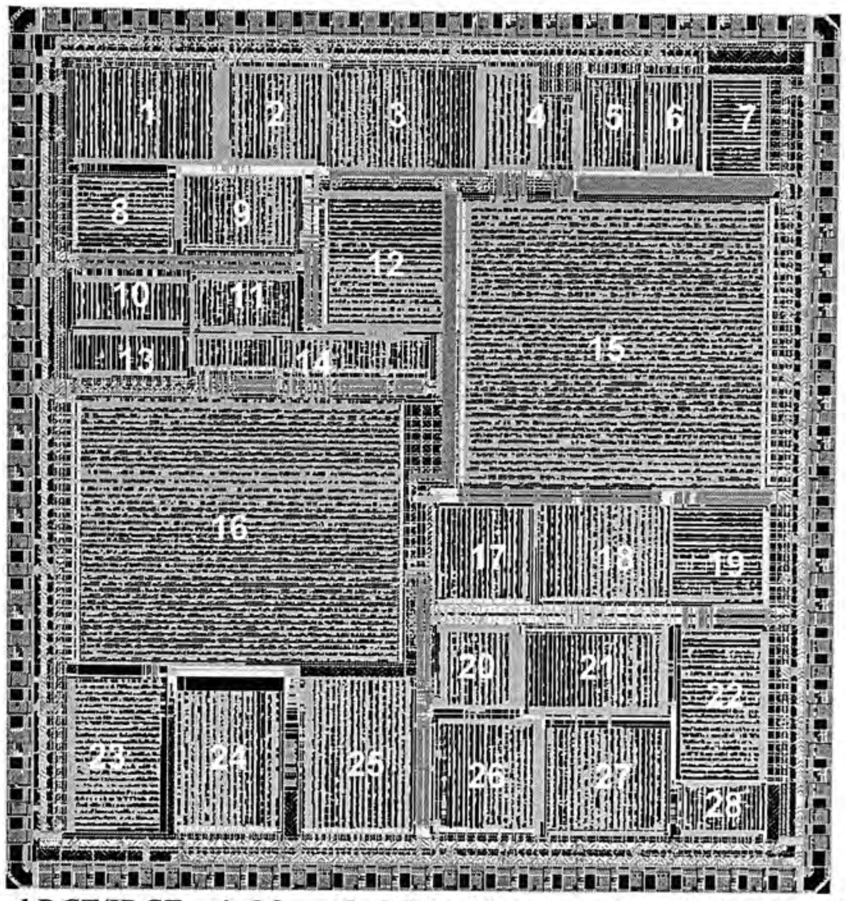


Figure 7.7 – Simulation result of the DCT coefficients memory

All the processing units, except the multiplier, data replicator and the DCT coefficients memory, were simulated at 250MHz of the data input rate. This is because according to the architecture of the 1D DCT/IDCT core shown in Figure 6.9 in chapter 6 and discussed in section 6.2.1.2, the throughput of the 1D DCT/IDCT core is greatly depended on the speed of the multiplication as it is the bottleneck of the whole operation. From the simulation results, the DCT coefficients memory only generates the DCT coefficients at the maximum frequency of 196MHz, as shown in Figure 7.7. As a result, the maximum rate of multiplication can only be 196MHz, and thus all other units are only required to work equal to or less than 196MHz in actual processing. To ensure the processing units can meet the requirement, a higher frequency which is 250MHz is chosen to verify the their operations. For the multiplier, it is found that it can work at 220MHz. However it is only the simulation result in the transistor-level simulation in which the all parasitic information is not taken into account. This is because the circuit is very large. The simulator HSPICE and workstation were unable to handle the simulation if the parasitic information is included. The rest of the simulation results show that other units are working properly at or below 250MHz without error. Based on this simulation result, the throughput of the whole 1D DCT/IDCT core should be able to work at 98 Mpixel/sec, which is half of the multiplication rate.



1. Input buffer and DCT/IDCT switch2 ***2. 15bit adder*** ***3. Data Replicator 1***
4 & 15. 15x16bit Multiplier1 ***5, 6 & 7. DCT Coefficients Memory1***
8. 1-to-2 DEMUX1 ***9.15bit subtractor***
10, 11 & 13. DCT Coefficients Memory2 ***12. Data Replicator 2*** ***14 & 16. 15x16bit Multiplier2***
17. 1-to-2 DEMUX4 ***18. 20bit adder2*** ***19. 1-to-2 DEMUX3***
20. DCT/IDCT switch4 & 5 ***21 & 28. 2-to-to switch*** ***22. 22bit adder1***
23. 22bit adder ***24. 22bit subtractor***
25. 1-to-2 DEMUX5 & 2-to-1 MUX ***26 Truncation unit and output buffer***
27. 21bit adder

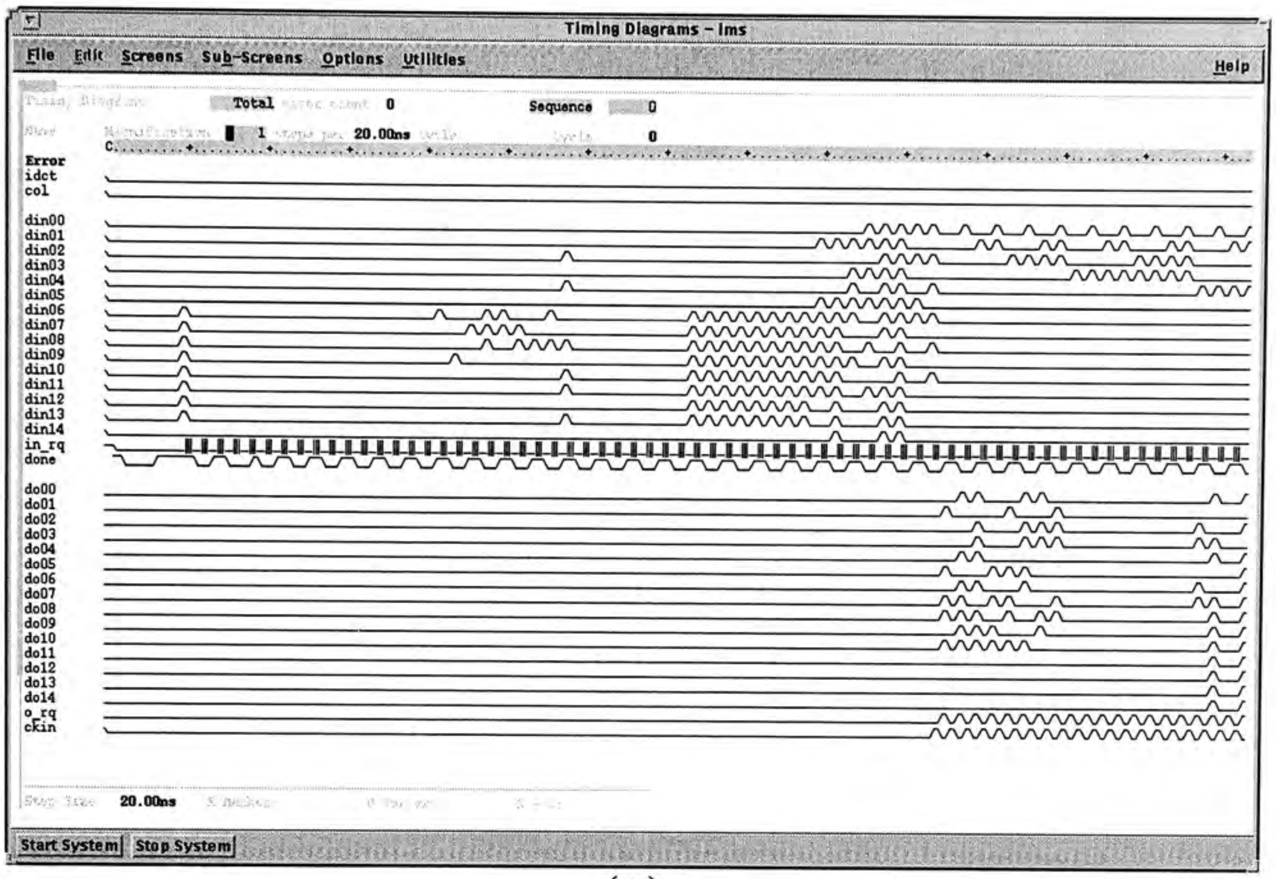
Figure 7.8 – Layout of the 1D DCT/IDCT core processor

The layout of the unified 1D DCT/IDCT processor is shown in Figure 7.8. It has 334k transistors and is designed using standard cells based on AMS 3M 1P 0.6u CMOS technology. The core dimension is 6.8mm × 7.5mm.

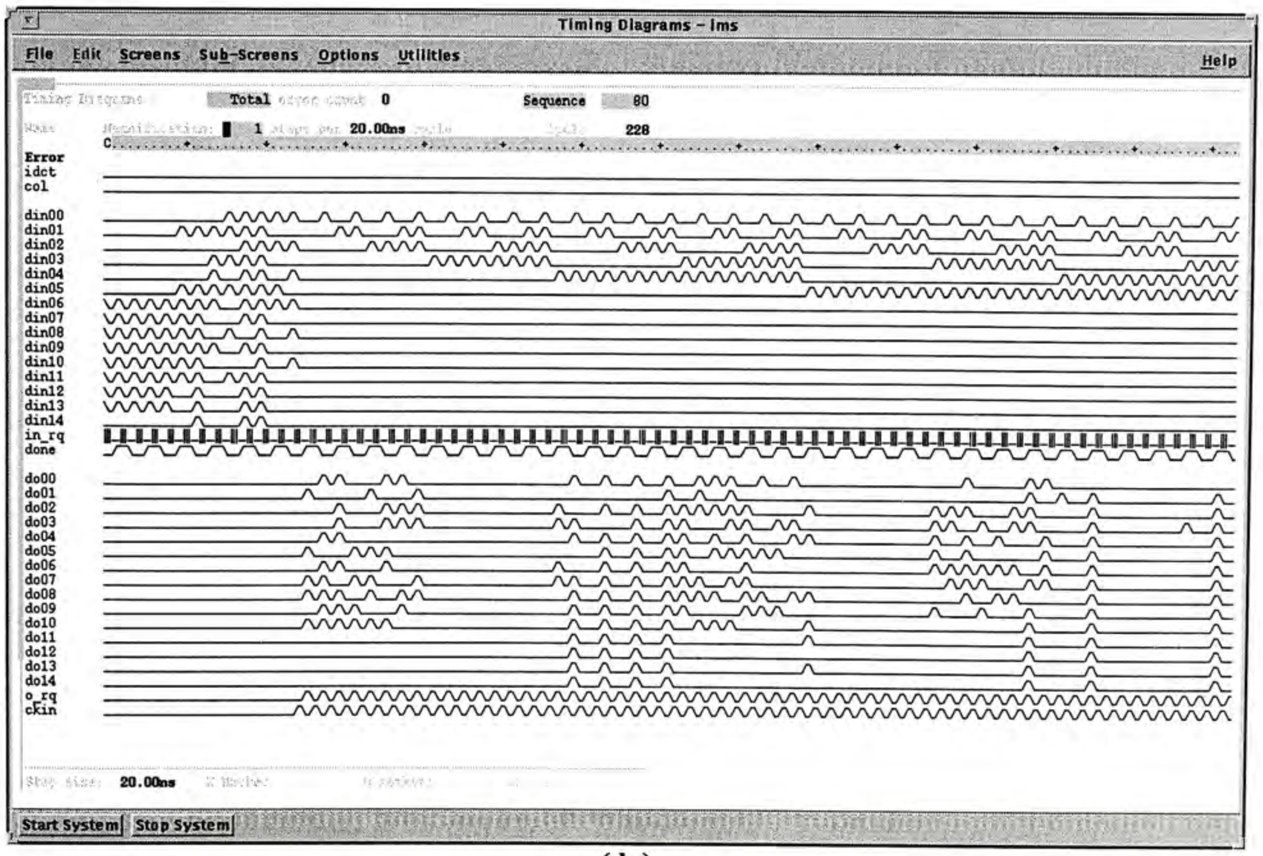
7.4.2 Measurement Results

The testing equipments of the DCT/IDCT core include the IMS XL-60 IC Tester, HP Infinium Oscilloscope and HP E3631A Triple Output DC Power Supply. The functionality of the chip was tested by IMS tester, and all the functions (row and column operation, DCT and IDCT) were verified and the chip is working properly.

Figure 7.9 shows part of the captured Input and Output waveforms of DCT row operation. Table 7.8 shows part of the input, measured and calculated data sets.



(a)



(b)

Figure 7.9 – (a) input waveform of the DCT/IDCT core, (b) measured output waveform of the DCT/IDCT core in DCT row operation

Input Data Set	Input Data((Re-ordered)	Measured Result	Calculated result
Set 1	255,0,0,0,0,0,0,0	90.125, 125.0625, 117.8125, 106, 90.125, 70.8125, 48.8125, 24.875	90.1561, 125.0501, 117.7946, 106.0124, 90.1561, 70.8352, 48.7921, 248740
Set 2	0,0,0,0,0,0,0,0	0,0,0,0,0,0,0,0	0,0,0,0,0,0,0,0
Set 3	1,2,3,4,5,6,7,8	12.75, -6.4375 0, -0.6875, 0, -0.1875, 0, -0.0625	12.7279, -6.4423, 0, -0.6735, 0, -0.2009, 0, -0.0507
Set 4	180,0,0,0,0,0,0,0	63.75, 88.4375, 83.3125, 74.9375, 63.75, 50.0625, 24.5, 17.5625	63.6396, 88.2707, 83.1492, 74.8323, 63.6396, 50.0013, 34.4415, 17.5581
Set 5	255,255,255,255,255,255,255,255	721.25, 0, 0, 0, 0, 0, 0, 0	721.2489, 0, 0,0, 0,0, 0,0,

Table 7.8 – Input data, measured result and calculated result of the DCT row operation

During the measurement, the actual Input Acknowledgement was not measured as it's duration is short and causes difficulty in the measurement. Instead of Input Acknowledgement, a signal Done was measured. The signal Done is created by a toggle flip-flop which input is the Input Acknowledgement. As a result, the signal Done is toggled in every Input Acknowledgement and it makes the measurement easier. Figure 7.10(a) shows the creation of the signal Done from the Input Acknowledgement while Figure 7.10(b) shows the timing relationship of the Input Acknowledgement and the signal Done.

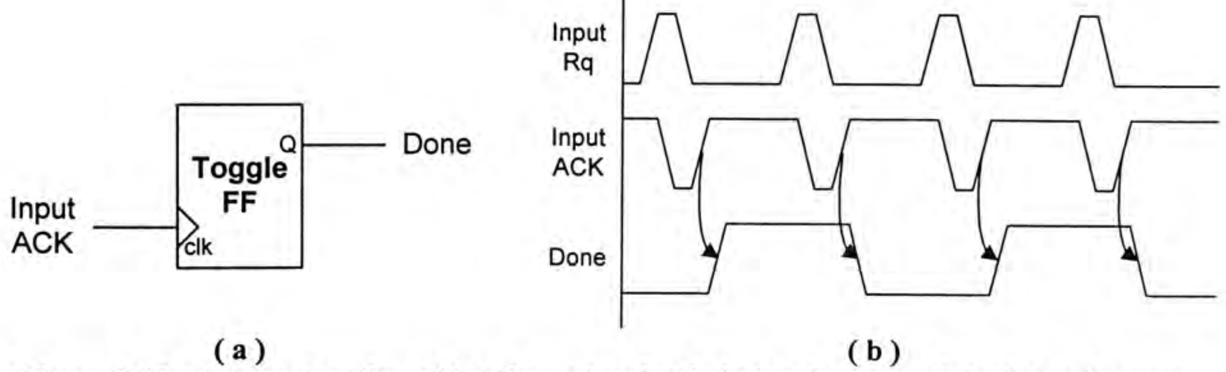
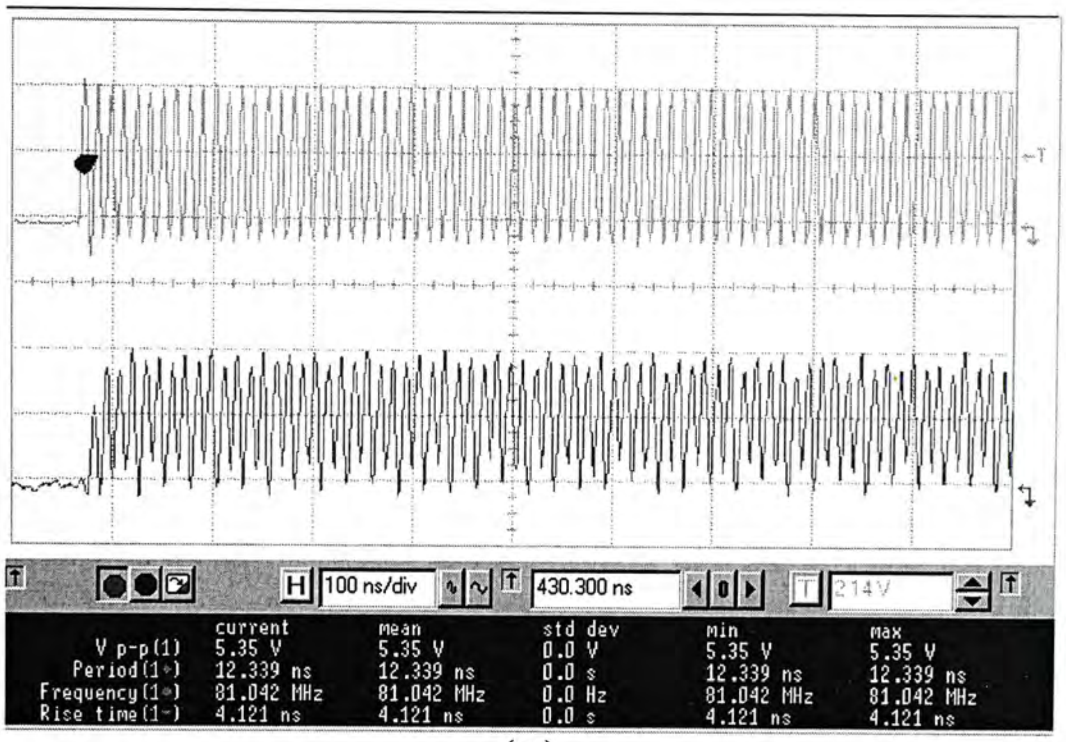


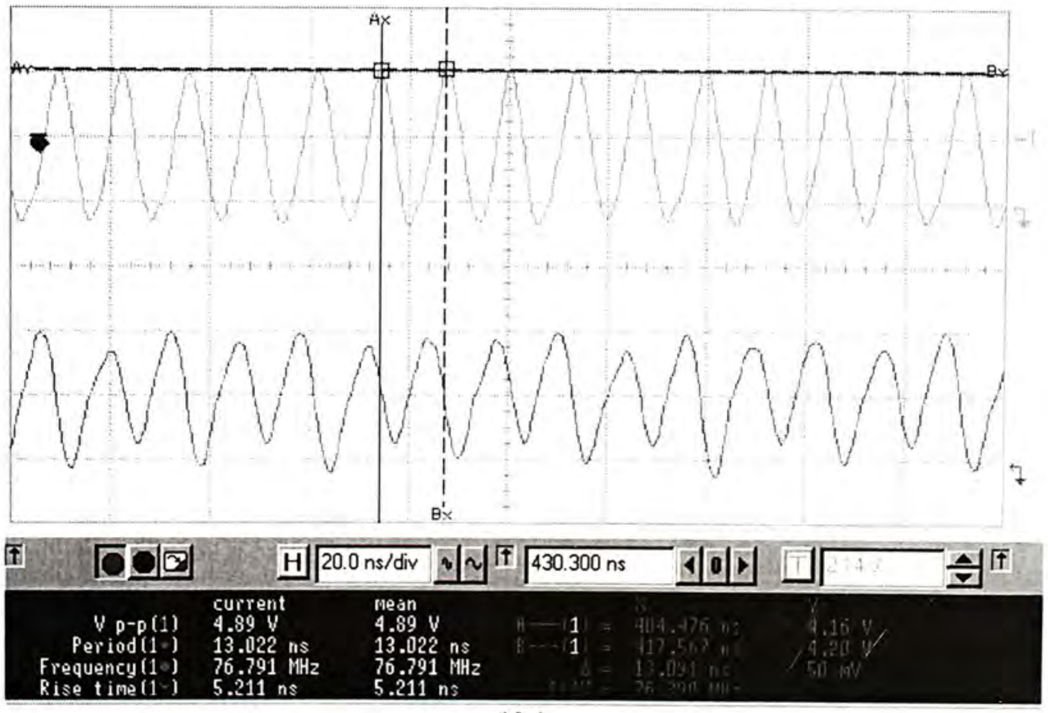
Figure 7.10 – (a) construction of the Done signal, (b) timing diagram of the Input Request, Acknowledgement and Done signal

The measurement shows the maximum frequency (throughput) of the DCT/IDCT is around 76MHz. Figure 7.11 shows the measured Output Request of the DCT/IDCT chip by the HP Infinium Oscilloscope. The average current of the DCT/IDCT core chip is about 1.43A under 5V power supply, so the average power consumption of the chip is about 7.15W. Table 7.9 shows the performance comparison of the 2D DCT/IDCT processor with other VLSI implementations.



```
                current      mean         std dev     min          max
      V p-p(1)  5.35 V       5.35 V       0.0 V       5.35 V       5.35 V
     Period(1)  12.339 ns    12.339 ns    0.0 s       12.339 ns    12.339 ns
  Frequency(1)  81.042 MHz   81.042 MHz   0.0 Hz      81.042 MHz   81.042 MHz
  Rise time(1)  4.121 ns     4.121 ns     0.0 s       4.121 ns     4.121 ns
```

(a)



```
                current      mean
      V p-p(1)  4.89 V       4.89 V        A—(1) = 404.476 ns     4.16 V
     Period(1)  13.022 ns    13.022 ns     B—(1) = 417.567 ns     4.20 V
  Frequency(1)  76.791 MHz   76.791 MHz        Δ = 13.091 ns      50 mV
  Rise time(1)  5.211 ns     5.211 ns      1/Δ = 76.390 MHz
```

(b)

Figure 7.11 – (a) measured waveforms of the Output Request (lower) and Acknowledgement (upper) signal, (b) zoomed waveforms which shows the average throughput is 76MHz

Design	Year	Tech.	Processing unit	Clock (MHz)	Pixel throughput (Mpixel/sec)
Cheng et. al. [35]	2000	0.6u	9 MUL, 21 ADD	100	100*
Kim et. al. [64]	1999	0.5u	/	80	26.6
Johnson et. al. [63]**	1998	1.2u	/	/	55.5
Jang et. al. [33]	1994	0.8u	8 MUL(DA), 2 accumulator, 2 pre- and post processors	100	100
Uramoto et. al. [32]	1992	0.8u	8 MUL and accumulators, 2 pre- and post processors	100	100
This processor**		0.6u	4 MUL, 14 ADD/SUB	/	76
***Transistor-level simulation result** ****Asynchronous design**					

Table 7.9 – Performance comparison of different 2D DCT implementations

7.4.3 Discussion

From the measurement result and Table 7.9, they indicate that the performance of this 2D DCT/IDCT processor is competitive to other designs. Since there were only few dedicated DCT/IDCT processors developed in asynchronous way previously, one of the recent asynchronous design which is developed by Johnson [63] is chosen for the comparison as it has similar architecture and the best performance among the others. By comparing with his asynchronous design, our design can run faster at about 36.9%. However, it should be noted that a more advanced technology has been used in our design, a certain portion of the superior performance may be caused by the benefits gained in the advanced technology. Although a direct comparison cannot be carried out, this comparison can be treated as a reference that this processor, the latest asynchronous DCT/IDCT processor, has improved performance and superior than previous asynchronous designs.

In comparison with the other similar synchronous DCT/IDCT designs, this DCT/IDCT processor has better performance than [64], while worse than [35], [33] and [32]. Although the performance is not as good as that of these synchronous designs, this processor has less operation units while a similar performance can still be achieved. This is because the operation units in an asynchronous system are not required to work on the same frequency. If a certain operation unit has better performance than other units, it can be scheduled to perform more operations by sending more input data to it. However, this cannot be done in synchronous design as all units must work in the same global clock frequency. This result explores the benefit of using the asynchronous architecture in system design as it can utilize every operation units in the system, and thus the number of operation units can be reduced.

However, the measurement result shows a performance deviation from the simulation result, where the difference is about 22%. This deviation is properly caused by two factors which are the temperature and the multiplier. For the environment setting in the HSPICE simulation, the temperature was set to be the room temperature. However in the actual measurement, it was found that the 1D DCT/IDCT core chip was very hot during the operation, and thus the temperature was much higher than room temperature. The increase in the temperature is due to the fact of high power consumption of this DCT/IDCT processor chip. As the design of current chip's package is not good for the heat dissipation, temperature cannot be cooled down effectively even a heat sink was added on the top of the chip. This causes a large amount of heat generated from the chip but cannot be dissipated, and thus the performance degradation is as a result. By setting the temperature at 90 degree and re-simulating the DCT memory coefficient memory again, the new result shows that the maximum operating frequency is lowered to around 168MHz (such the multiplication rate is 84MHz). In this case, the difference between the simulation and measurement result is more reasonable. The rest of the difference may be due to extra delay caused by the parasitic of the routing, which is not included in HSPICE simulation, and the difference between the parameters of the HSPICE models and the actual fabrication process.

Another possible reason of causing large performance deviation is the multiplier. Since the all parasitic information was not taken into account in the HSPICE simulation, the actual performance of the multiplier may be much lower than 220MHz if parasitic was considered as well. However, this cannot be verified by neither the simulation nor measurement.

In order to improve the performance of the current design, the modification of the of the DCT coefficient memory and the multiplier must be considered. The limiting factor on the speed of the DCT coefficients memory is the output feedback path. The DCT coefficients memory is not only required to transmit the output to the multipliers, but also send the output to the input of the DCT coefficients memory simultaneously. This split-path introduces a large handshaking overhead and thus its performance is limited. The reducing of the handshaking overhead can be investigated in future, and thus the performance of the DCT coefficient memory, as well as the whole DCT processor can be improved. For the multiplier, the performance can be improved by building a PFA as a single standard cell. The current implementation uses several basic logic standard cells to build the PFA. This causes a large parasitic in the auto-placement and auto-routing process. Since many identical PFAs are used in the multiplier core, building the PFA as a single standard cell can minimize the parasitic due to the routing, and also the silicon area can be saved as well since the PFA can be built more compactly in this way.

For further analyzing the practicality of the asynchronous design, a comparison can be made on the asynchronous and synchronous implementations of this 1D DCT/IDCT core. In the synchronous implementation, the bottleneck should no longer be the DCT coefficients memory but in the multiplier. This is because DCT coefficients memory can be implemented easily in synchronous design by using ROM or counters, both can be run very fast as not many computations are required. In the multiplier, the critical path is inside the carry generation, which is given by Equation 7.3(same as Equation 5.4)

$$Cout = A \bullet B \bullet Cin + P \bullet \left(Cin + A \bullet B\right) \qquad \text{- Equation 7.3}$$

The implementation of the Equation 7.3 in domino logic is shown in Figure 7.12(a). For the synchronous implementation, Equation 7.3 is modified to Equation 7.4 as the inverting static CMOS logic provides a faster response than the non-inverting logic.

$$\begin{aligned} Cout &= \overline{\overline{A \bullet B \bullet Cin + P \bullet \left(Cin + A \bullet B\right)}} \\ &= \overline{\overline{A \bullet B \bullet Cin} \bullet \overline{P \bullet (Cin + A \bullet B)}} \\ &= \overline{\overline{A \bullet B \bullet Cin} \bullet \overline{P} + \overline{(Cin + A \bullet B)}} \end{aligned} \qquad \text{- Equation 7.4}$$

According to Equation 7.4, the synchronous implementation of the carry generation is shown in Figure 7.12(b). From the information provided in the 0.6u standard cell databook [65], the delay which is under 25°C and 5V supply voltage of each logic cell is extracted for the performance estimation.

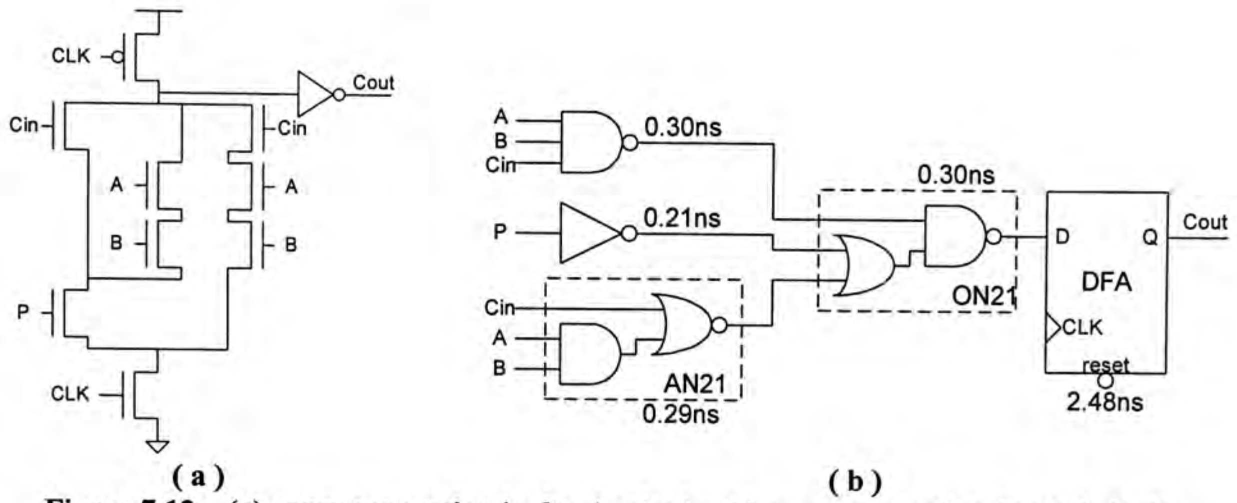


Figure 7.12 – (a) carry generation in domino logic, (b) carry generation in static logic

$$\begin{aligned}\textit{Operating Frequency} &= 1 / (\textit{longest delay}) \\ &= 1 / (0.3 + 0.3 + 2.48)\,ns \\ &= 1 / 3.08ns \\ &= 325\ MHz\end{aligned}$$ **- Equation 7.5**

The result of Equation 7.5 shows that the synchronous multiplier could run at about 325 MHz. However, this estimation doesn't include the worst case temperature, supply voltage and clock skew. In practical, a margin of 50% or more is required in the global clock frequency when compared with its performance in typical condition due to worst case performance assumption in synchronous design. Therefore, the performance of the synchronous multiplier should be around 216MHz (for 50% margin), which should be similar to that of the asynchronous performance in typical condition. And in overall, as the synchronous multiplier is limited to 216MHz, the synchronous implementation of the whole 1D DCT/IDCT core may be able to run faster than the asynchronous implementation described in this thesis, but the difference may not be so great. This result indicates that asynchronous design is practical and the performance can be similar to synchronous design.

Although the performance of this processor is good, the tradeoffs are the area and power. In order to maximize the performance, DCVSL structure is used inside all the processing units. This causes nearly a double of size to perform the same logic function as other designs. Also, all operation units within the 1D DCT/IDCT core are deeply pipelined, especially the bit-parallel architecture of the multiplier. The deeply pipeline structure decomposes all the complex functions into simple logics with several stages. As a tradeoff of speed, this causes more area are required to implement the design. Furthermore, the handshake cell in the asynchronous circuit also causes an additional size overhead to the synchronous circuit.

For the power consumption, the measurement result shows that the average power consumption of the dedicated DCT/IDCT processor is about 7.15W under 5V supply voltage, which is an extremely high value compared with other designs. In order to verify the correctness of the power consumption, each of the functional units was simulated separately by HSPICE under 5V supply voltage. The simulation results are listed in Table 7.10.

	Used in DCT / IDCT / Both	Number	Operating Frequency (MHz)	Current Drawn (mA)	Current Drawn in DCT operation	Current Drawn in IDCT operation
15-bit adder	DCT	1	76/2	9.32	9.32	0
15-bit subtractor	DCT	1	76/2	9.85	9.85	0
16-bit data replicator	Both	2	76×2	28.33	56.66	56.66
DCT coefficient memory	Both	2	76×2	125.83	251.66	251.66
Multiplier	Both	2	76×2	478.00	956.00	956.00
20-bit adder	Both	2	76	32.45	64.90	64.90
21-bit adder	Both	1	76	33.82	33.82	33.82
22-bit adder	IDCT	1	76/2	18.04	0	18.04
22-bit subtractor	IDCT	1	76/2	18.05	0	18.05
Truncation unit	Both	1	76	15.00	15.00	15.00
				Total Power	1397.21	1414.13

Table 7.10 – Simulation results of power consumption of different operation units in the 1D DCT/IDCT core

From the simulation results, the total power consumption of the 1D DCT/IDCT core is about 6.986W (1397.21mA × 5V) and 7.071W (1414.13mA × 5V) in the DCT and IDCT operation respectively. This results show that the power consumption is consistent in both the simulation and measurement result. Therefore the power consumption in the measurement is correct.

The main reason of large power consumption is due to the use of DCVSL structure. Since both the true and complement value are presented in the DCVSL structure, either one of the true or complement logic block must be discharged in each Evaluation phase. Therefore every logic functional block must consume power in each Evaluation cycle, which causes a constant and high discharge current. However in single-rail design, which uses the true logic block only, there is no discharge current if the pull-down path is not conducted during the Evaluation phase (the output kept at logic zero in the domino logic). Since the pull-down path is conducted occasionally, a single-rail design consumes less power than the DCVSL design in average when performing the same logic function. In order to verify this, a single-rail 15-bit adder is constructed for the comparison. Both circuits are simulated by HSPICE under 5V supply voltage. Three different input patterns which are random number, all zeros and all ones patterns are fed into the inputs of the adder at a frequency of 76MHz to investigate the current drawn in different conditions. The simulation results are shown in Table 7.11.

	Average current drawn at random input (mA)	Average current drawn at all zeros input (mA)	Average current drawn at all ones input (mA)
DCVSL adder	20.69	20.34	20.59
Single-rail adder	9.58	4.33	8.90

Table 7.11 – Comparison of power consumption on DCVSL and single-rail adder

Although the power consumption of this dedicated DCT/IDCT power is high, if there is no data input, this processor consumes less power than other designs as no transition will be occurred in asynchronous design when there is no request of operation.

Future work can be focused on reducing the power and size of the processor. As mentioned before, DCVSL structure is the main reason of the high power consumption of this design. In order to reduce the power consumption while not affecting the current performance, single-rail design or conventional asynchronous structure should be used in the non-critical parts, such as the adders and subtractors. This modification not only helps to reduce the power consumption, but also helps to reduce the area required to implement the design. For the area, since around 30% of the area is consumed by the multipliers, as shown in Figure 7.8, minimizing the size of the multiplier can greatly reduce the size of the whole design. This can be done by using Booth coding [66] or common sub-expression elimination [67] to reduce the complexity of the multiplier design, and thus its size can be reduced. Also, grouping the PFA into a single standard cell, which has been mentioned before, can also reduce the overall area too. Moreover, the adders and subtractors are not the limiting factor of the performance of the processor, area-saved or power-saved algorithm, for example ripple adder, can be used in the implementations of adder and subtractor instead of the size-consuming fast BLC algorithm.

From the result shown in Table 7.11, it indicates that the power consumptions of the DCVSL adder are nearly the same in three input patterns, while those of the single-rail adder are depended on the input patterns. Also the single-rail adder consumes not more than half of the power of DCVSL adder in any patterns. This result shows that the DCVSL structure consumes more than a double of power when compared with usual structure, and this causes our design to have an extremely high power consumption.

This result also shows the disadvantage of the domino logic (or dynamic logic), which is the high dynamic power consumption. Although the single-rail adder has relatively low power consumption design than the DCVSL design, it consumes higher power than the static logic. In the design which uses the static logic with latch or flip-flop, the power consumption should be relatively low and nearly the same in constant input patterns (all zeros and all ones patterns). This is because for a constant input pattern, the output of the logic gate will not change and thus there is no switching during the operation. Therefore it consumes very little or even no dynamic power under this situation. However in the domino logic, the requirement of the Precharge phase causes the output to be precharged to logic zero in every Precharge phase. As a result, if a logic block has an output of logic one in Evaluation phase, it will consume power in the Precharge phase. This explains that even having a constant input pattern, the domino logic still has switching during operation, and it causes a relatively high power dynamic power consumption compared with using static logic and latch or flip-flop in conventional architecture.

7.5 Transpose Memory

7.5.1 Simulated Results

Due the size of the whole design, different units of the transpose memory were simulated separately by HSPICE under 5V supply voltage in order to obtain the estimated throughput.

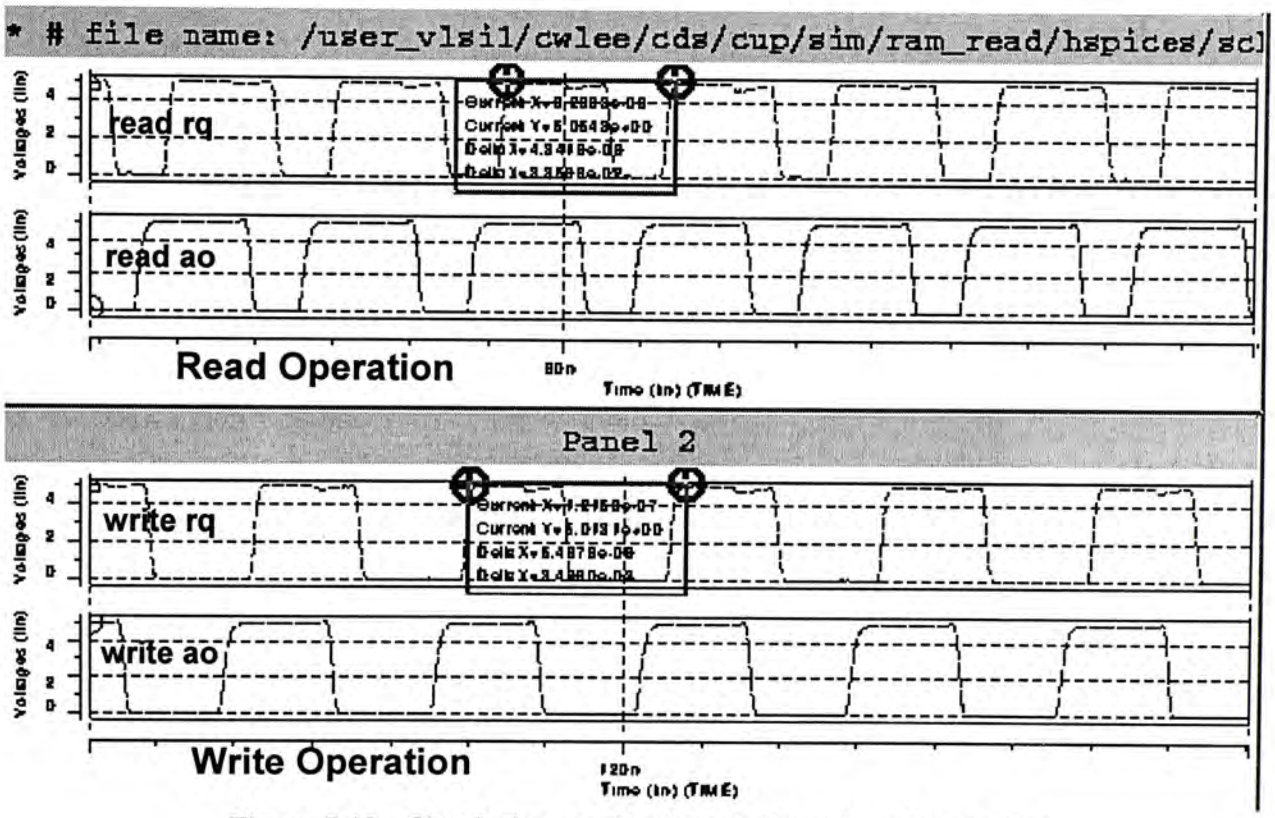


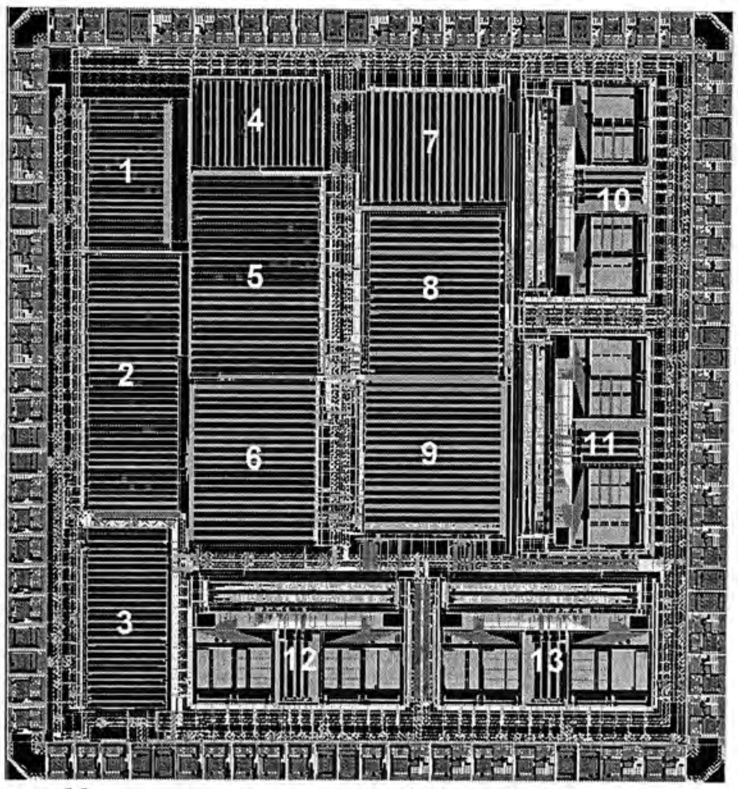
Figure 7.13 – Simulation result of the write and read operation

	Tested Frequency(MHz)
Write address generator	276*
Read address generator	276*
Multiplexing network	276
32x16bit SRAM block	182.22/230.31
**Self-generated frequency*	

Table 7.12 – Performance of different units in the transpose memory

Since the interleaving technique is applied on the write operation, the performance of the transpose memory is now limited by the read operation of the SRAM block. As a result, the minimum operating speed of the whole transpose memory should be 230MHz.

The layout of the transpose memory is shown in Figure 7.14. It has 111k transistors. The SRAM Blocks are full custom designs and other parts are designed by using standard cells based on AMS 3M 1P 0.6u CMOS technology. The core dimension is 3.9mm × 4.2mm.



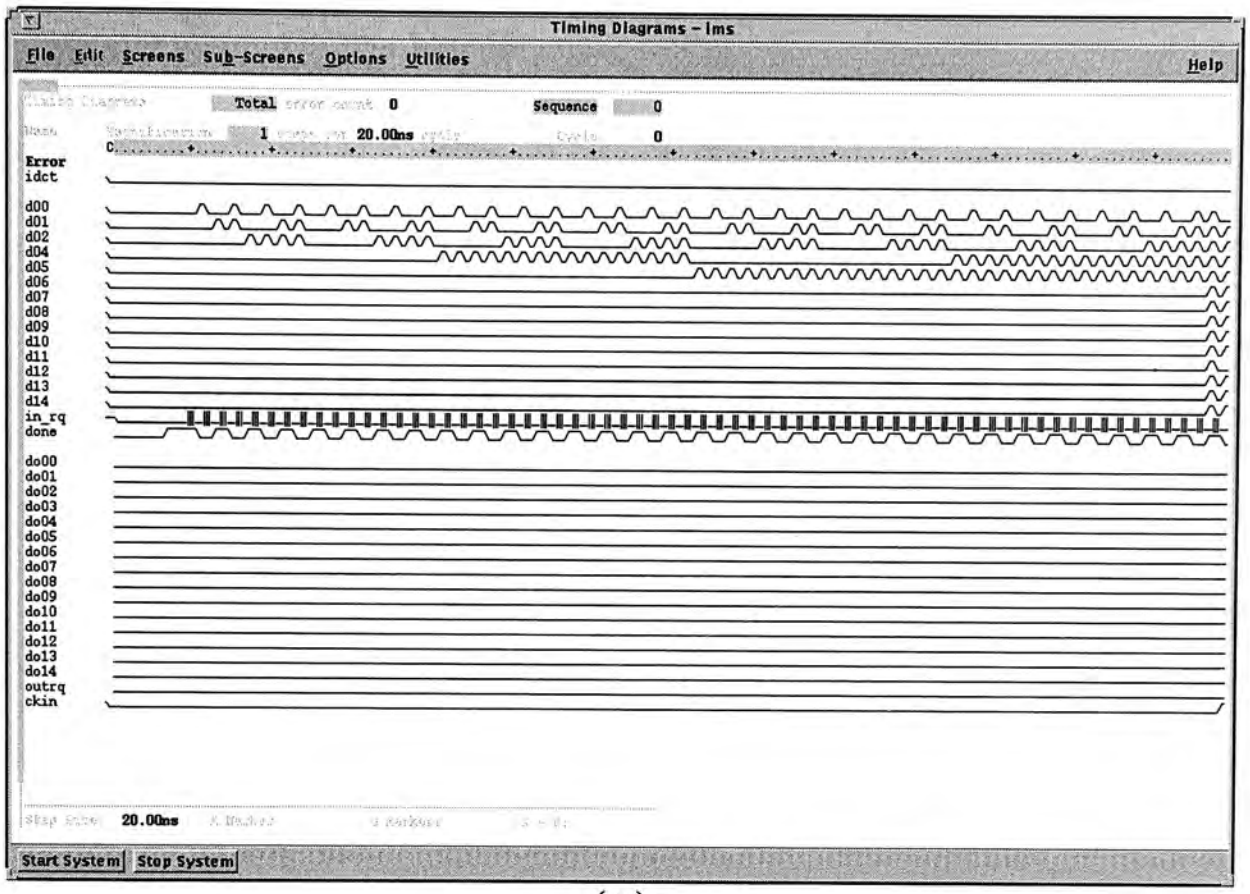
1 & 2. Column address generator ***3. Input buffer***
4 & 5. Read address generator ***6, 8 and 9. Multiplexing network***
7. Output buffer ***10, 11, 12, 13. 32x16bit SRAM block***

Figure 7.14 – Layout of the transpose memory

7.5.2 Measurement Results

The testing equipments of the transpose memory include the IMS XL-60 IC Tester, HP Infinium Oscilloscope and HP E3631A Triple Output DC Power Supply, which are the same of those of the DCT/IDCT chip. The functionality of the transpose memory chip was tested by IMS XL-60 IC tester, and all the functions (DCT and

IDCT) were verified and the chip is working properly. Figure 7.15 shows part of the captured Input and Output waveforms of TRAM.

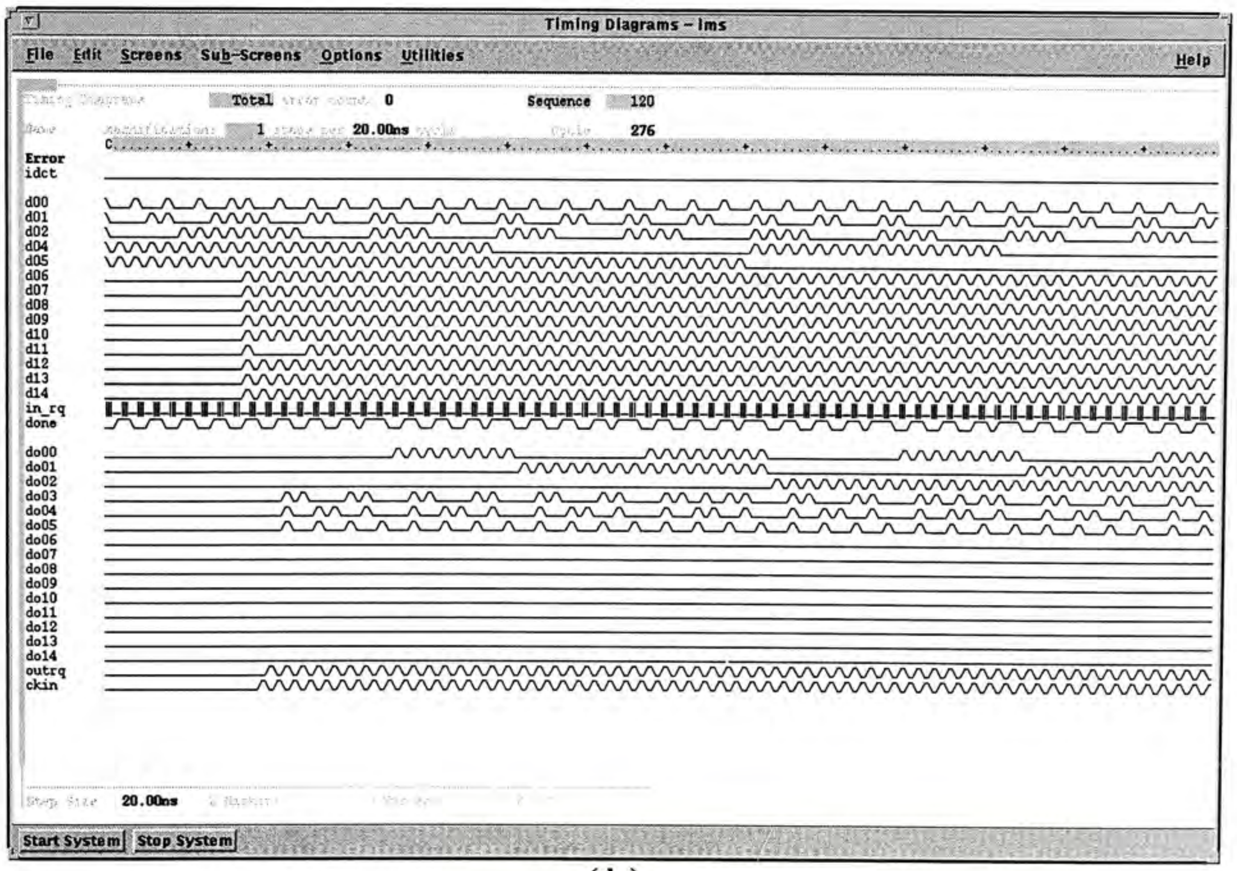


Timing Diagrams Total error count 0 Sequence 0

Magnification 1 steps per 20.00ns cycle Cycle 0

Step Size 20.00ns

(a)



Timing Diagrams Total error count 0 Sequence 120

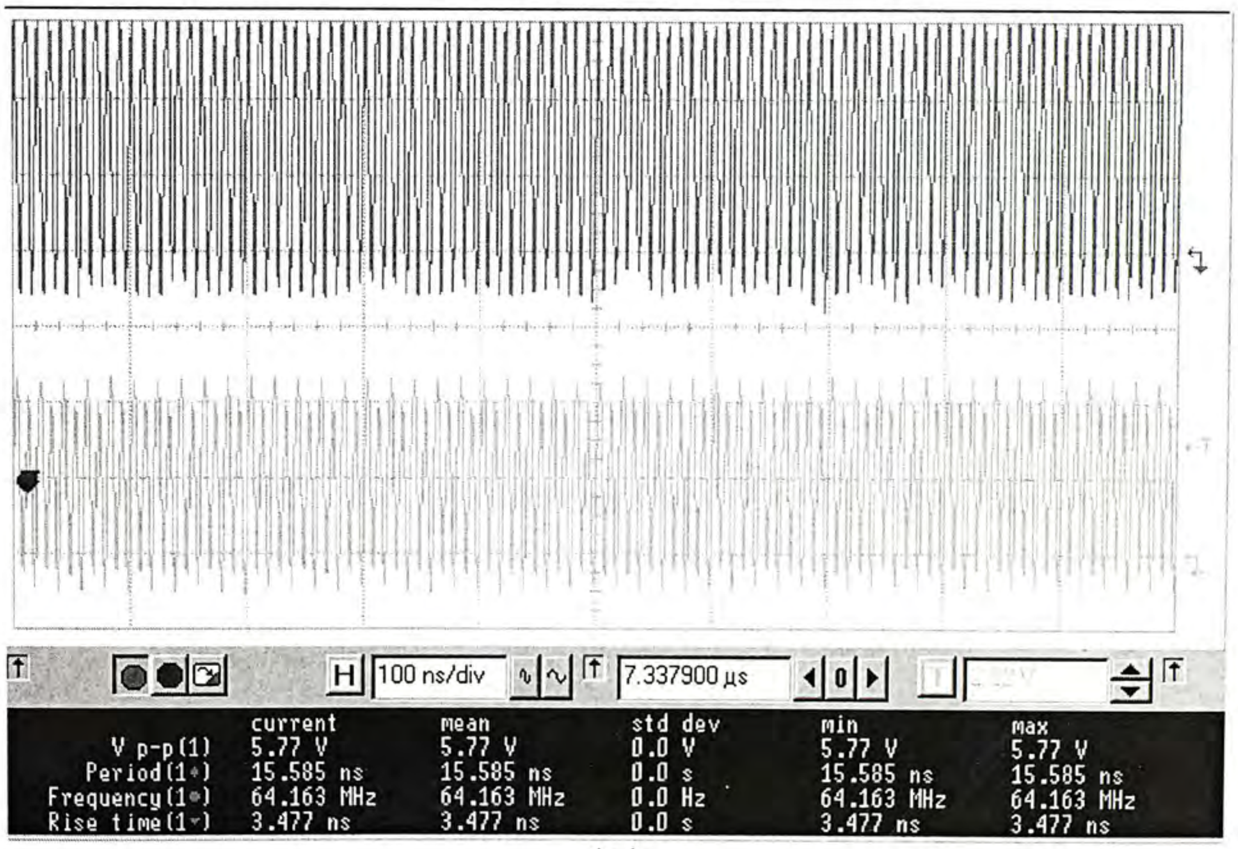
Magnification 1 steps per 20.00ns cycle Cycle 276

Step Size 20.00ns

(b)

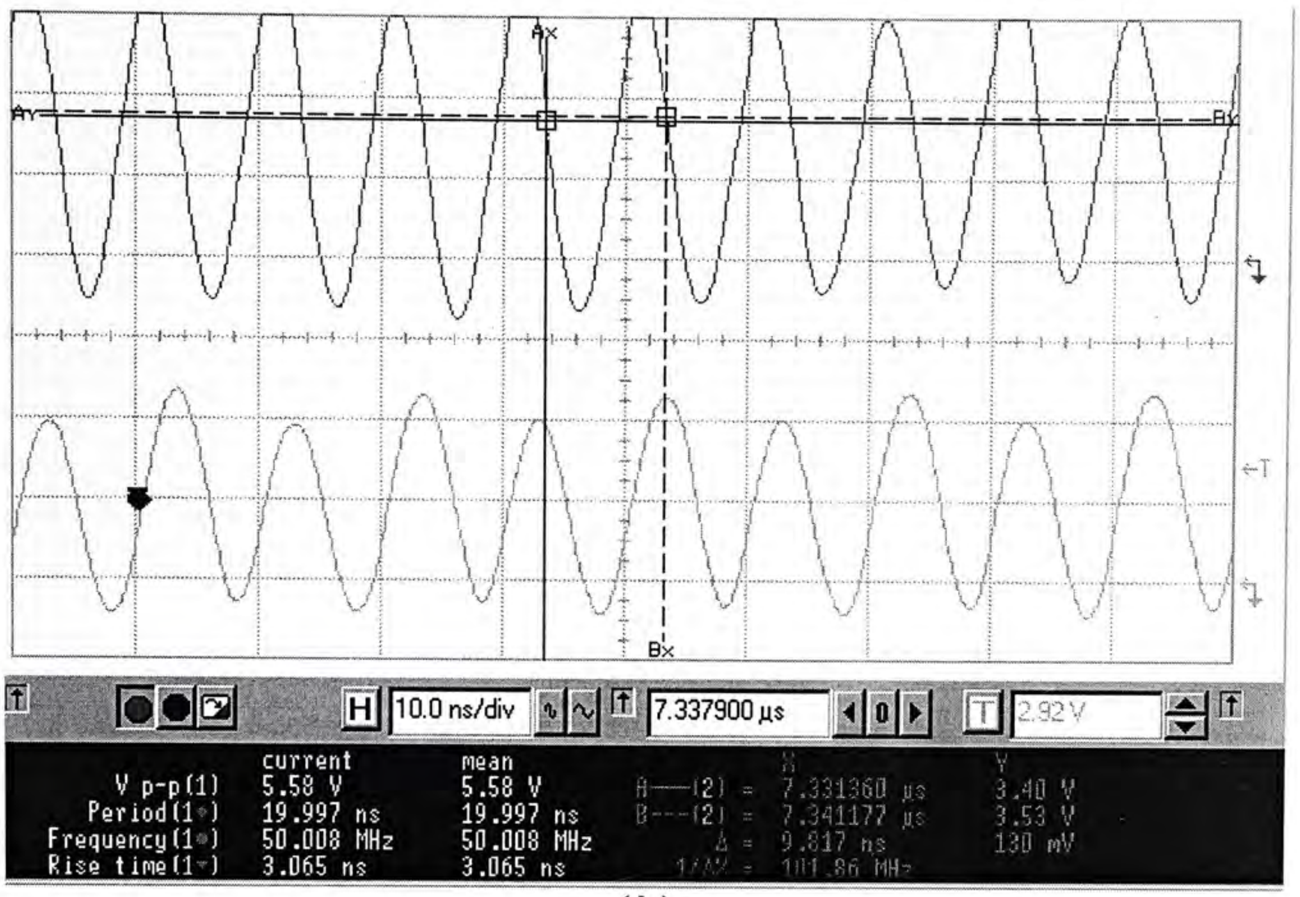
Figure 7.15 – (a) input waveform of the transpose memory, (b) measured output waveform of transpose memory in DCT operation mode

Although the simulation result showed that the transpose memory could be operated at 230MHz, there is no method to verify this. This is because that there is a limitation of the IMS XL-60 IC Tester that test vectors can only be generated at a maximum rate of 100MHz. As a result, the transpose memory can only be verified that it is working properly at 100MHz input rate. However, the transpose memory is supposed to be worked with the DCT/IDCT core chip which is operated at 76MHz only. This result at least can prove that the transpose memory can work well with the 1D DCT/IDCT chips. Figure 7.16 shows the Output Request of the transpose memory at 100MHz. The average current drawn of the transpose memory at 100MHz under 5V power supply is around 350mA, which means that the average power consumption is about 1.75W.



```
                 current       mean          std dev      min           max
     V p-p(1)    5.77 V        5.77 V        0.0 V        5.77 V        5.77 V
   Period(1↑)    15.585 ns     15.585 ns     0.0 s        15.585 ns     15.585 ns
Frequency(1●)    64.163 MHz    64.163 MHz    0.0 Hz       64.163 MHz    64.163 MHz
 Rise time(1↓)   3.477 ns      3.477 ns      0.0 s        3.477 ns      3.477 ns
```

(a)



```
              current      mean          X              Y
   V p-p(1)   5.58 V       5.58 V        A——(2) = 7.331360 µs   3.40 V
 Period(1)    19.997 ns    19.997 ns     B---(2) = 7.341177 µs  3.53 V
Frequency(1)  50.008 MHz   50.008 MHz        Δ = 9.817 ns       130 mV
Rise time(1)  3.065 ns     3.065 ns      1/ΔX = 101.86 MHz
```

(b)

Figure 7.16 – (a) measured waveforms of the Output Request (upper) and Acknowledgement (lower) signal, (b) zoomed waveforms which shows the average throughput is 100MHz

7.5.3 Discussion

From the result shown in Table 7.12 and by the help of the interleaving technique, the transpose memory can be operated continuously at 230MHz. And from the measurement result, it indicates that the chip has no problem in operating at 100MHz. As a result, the read/write operation at 76MHz in the transpose memory was fulfilled and thus the whole 2D DCT/IDCT processor can provide a throughput at 76 Mpixel/sec.

Considering the RAM block alone, its performance is restricted by the write operation. The poor speed of the write operation is due to the slow detection of the completion of write operation. Although the monitor cells are used to provide a fast detection of completion of the write operation, overhead exists on collecting all the done signals from different monitor cells.

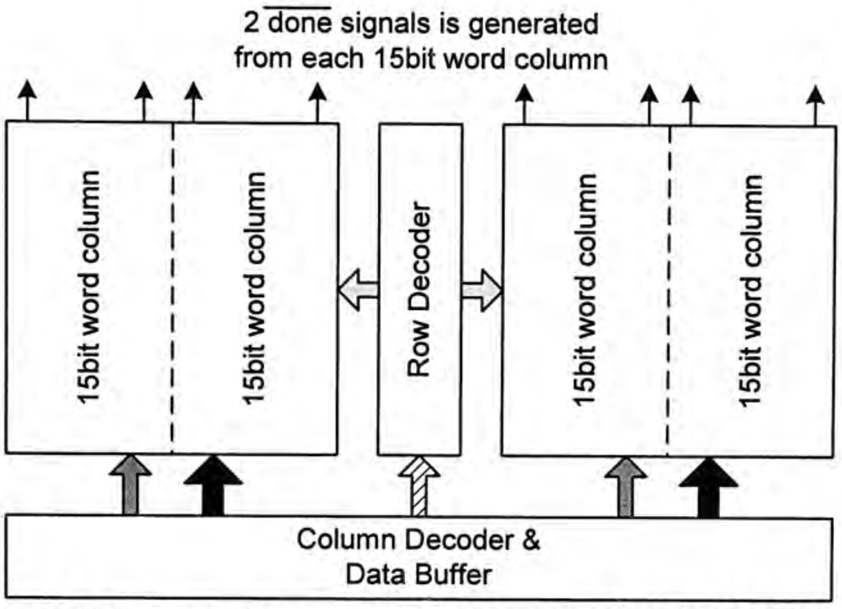

2 $\overline{\text{done}}$ signals is generated from each 15bit word column
15bit word column
15bit word column
Row Decoder
15bit word column
15bit word column
Column Decoder & Data Buffer

Figure 7.17 – Done signals generated from the 32×15bit RAM block

Figure 7.17 shows the done signals generated from the monitor cells in a 32×15bit RAM block. It is required to handle a total of 8 done signals and thus the overhead is introduced. If the detection method can be improved, a better performance in the write operation can be achieved.

Although the goal of the transpose memory is achieved in this implementation, it has limitations and the design still can be further improved. In the current architecture, an interleaving technique is used so as to achieve a higher operating speed in the transpose memory. However, the actual speed requirement of the 2D DCT/IDCT is now limited by the 1D DCT/IDCT core, which is 76 MHz, the transpose memory is not necessary to be run at such a high frequency. Therefore, the interleaving technique is not necessary and can be removed from the transpose memory. In this case, the maximum operating speed of the transpose memory will be lowered to about 182MHz but is still capable of handling the 2D DCT/IDCT operation. In this way, the area of the additional multiplexers and demultiplexers can be saved.

In order to realize the whole 2D DCT/IDCT processor design in asynchronous pipeline architecture, the address generator and the multiplexing networks are all built according to the methodology introduced in chapter 2. The simulation shows that they can be operated in a very high frequency. However, they are not required to work at such high frequency as the operation is limited to 76MHz, and thus the benefit the asynchronous pipeline architecture cannot be gained in this implementation. On the other hand, the address generator and the multiplexing networks consume over 50% of the whole design, it causes the transpose memory to be not cost effective. Therefore other approaches should be applied to the design of transpose memory.

First for the address generator and the multiplexing network, the conventional micropipeline structure can be used. As latch is used in the micropipeline, a counter can be easily implemented and can be used to replace the area-consuming address generator. Also, as the multiplexing network is not the critical unit regarding the performance of the transpose memory, DCVSL structure is not necessary to be used. As a result, the area of the transpose memory can be largely reduced. Also the removal of DCVSL structure can help to reduce the power consumption which have been discussed in section 7.4.3, the discussion part of 1D DCT/DCT core.

Another possible approach is the replacement of the RAM block with an array of shift registers, or storage elements. Since the sequences of the DCT and IDCT are fixed, the flow of the data in the transpose memory can be pre-determined. Therefore hardwired connections with some multiplexers on the array of shift registers may be able to perform the same function of the RAM blocks. Also it can eliminate the address generator and multiplexing, thus the area can be reduced.

Therefore future improvement or development of the transpose memory can be based on the above suggestions, then a more cost effective implementation can be achieved.

Chapter 8

Conclusions

In this thesis, several asynchronous methodologies have been discussed, and a new asynchronous pipelined architecture is then presented. This new architecture uses a novel, simple but fast handshake cell which adopts a more relaxed handshaking protocol than in the traditional architecture. Furthermore, this new architecture employs the DCVSL structure in logic design, and thus the complex latch can be removed and the completion can be directly detected by the handshake cell. With the new asynchronous pipelined architecture, the circuit developed has a simpler architecture and has higher performance than the traditional methodologies. The performance of the new asynchronous pipelined architecture has been proven by the programmable DSP processor and the dedicated DCT processor

Since the dynamic logic is used in the new asynchronous pipelined architecture, a new technique called Refresh Control circuit is introduced in this thesis to solve the charge leakage problem. The new technique is a self-timed, self-calibrated and self-operating circuit, it monitors the charge leakage in the dynamic logic and controls the refresh process effectively in order to reduce the pull-up current. From the result, it is shown that this technique causes less performance degradation than the traditional technique, it is suitable to apply to a large system which requires a high performance. Moreover, the Refresh Control circuit uses a general purpose monitoring scheme and thus it can be applied to other circuits which also encounter the charge leakage problem.

Based on the new asynchronous pipelined architecture, a programmable DSP processor and a dedicated 2D DCT/IDCT processor have been constructed. Although the performance of the DCT implemented in the programmable processor is not as good as other dedicated designs, it still has a reasonable performance of 22.7Mpixel/sec with a small number of operation units. And the impressive result shown in the estimation of 2D DCT operation demonstrates the advantages of the combination of asynchronous pipeline and dataflow architecture in circuit design, and the use of switching network and parallelism in the processor architecture. This result encourages the further development of the processor and the use of the asynchronous pipelined architecture.

Finally, the development of the dedicated 2D DCT/IDCT processor is shown in the thesis. This processor is fully pipelined and the throughput is 76Mpxiel/sec, which is competitive with other high performance synchronous designs when considering the number of operation units used. Also, this dedicated 2D DCT/IDCT processor fully satisfies the IEEE specification and is capable of real-time processing on the MPEG-2, or even the more computational demanding HDTV signals. The result indicates that the asynchronous design with the new pipelined architecture can perform as good as other synchronous designs, and the proposed DCT/IDCT architecture is suitable for the asynchronous implementation. Furthermore the benefit of the asynchronous approach in system design has been demonstrated, in which operation units can be fully utilized and the number of operation units can be reduced in the whole system.

Both of the results of the programmable DSP processor and the dedicated DCT/IDCT processor imply the high performance of the new asynchronous pipelined architecture. In other words, the use of the new asynchronous pipelined architecture favours the asynchronous approach in system implementation, especially for the DSP applications.

However, it is found that there are area and power penalties in these designs. The use of the DCVSL structure not only causes a large increase in the silicon area, but also causes a high dynamic power consumption. Also, the inappropriate architectures used in different operation units cause unnecessary area overhead and cause the whole design to be inefficient in term of area. Designers should consider different approaches in the implementation of different operation units in a system in order to minimize the penalties, and further development of the new asynchronous pipelined architecture can be focused in these two areas.

Appendix

Operations of switches in DCT implementation of programmable DSP processor

Note:
Switch 1 to 12 are located at the switching networking, their instructions can be referred to Table 5.1. Switch 13 and 14 represents the input demultiplexing networks of the FIFO memory 1 and 2 respectively, while Switch 15 and 16 represents the four-to-one multiplexers of FIFO memory 1 and 2 respectively. For switch 13 to 16, the register name means the FIFO set that the demultiplexing networks and four-to-one multiplexers connecting to.

		Switches																		
#	In	15	16	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Coef f	Outp ut	Operation
1	X0				1				0			7								input X0 -> Ai1, Si1
2	X7				3						2				7					input X7 -> Ai2, Si2
3	X1				1				0			7								input X1 -> Ai1, Si1
4	X6				3						2				7					input X6 -> Ai2, Si2
5	X3				1				0			7								input X3 -> Ai1, Si1
6	X4				3						2				7					input X4 -> Ai2, Si2
7	X2				1				0			7								input X2 -> Ai1, Si1
8	X5				3						2				7					input X5 -> Ai2, Si2
9				1			6	0	3		1	6	3	3			A2	C1		Ao(K0) -> Ai1, Si1, So(K4) ->regA2, Muli
10				1			2	0			1	6		3			B2			Ao(K1) -> Ai1, Si1, So(K5) -> regB2
11				3			2			2	1			3	6		C2			Ao(K2) -> Ai2, Si2, So(K6) -> regC2
12				3			2			2	1			3	6		C2			Ao(K3) -> Ai2, Si2, So(K7) -> regC2
13			A2				1		1			1								RegA2(K4) -> Ai1
14			B2				3				7			3	1		B2			RegB2(L5) -> RegB2, Ai2
15							0		3				7			B1		C0		So(L2) -> RegB1, Muli
16							6		3		1		3	3			A2	C0		So(L3) -> RegA2, Muli
17			C2	3			1		1	0		1		0			D2			RegC2(K6) -> Ai1, Ao(L0) -> RegD2
18			C2	3			7		1	0	0	1		0	1		D2			RegA1(K7) -> Ai1, Ai2, Ao(L1) -> RegD2
19			B2				3				3				1					RegB2(K5) -> Ai2
20			D2				1		1			7								RegD2(L0) -> Ai1, Si1
21			D2				3				3				7					RegD2(L1) -> Ai2, Si2
22				1				2					6			A1		C0		Ao(L5) -> RegA1, Muli
23				7				2		0			2	2			B2	C0		Ao(L6) -> RegB2, Muli
24				1				2					2					C2		Ao(L7) -> Muli
25				1				5					2					C3		Ao(M0) -> Muli
26						0		3					0			C1				Mo(c1*Z4) -> RegC1
27			A2			0	3	1			3	2			3					RegA2(L3) -> Si2, Mo(c0*L0) -> Si1
28		B1		0		2		0		3		0			0					RegB1(L2) -> Ai1, Mo(c0*L3) -> Ai2
29			B2			0	3	1			3	2			3					RegB2(L6) -> Si2, Mo(c0*L5) -> Si1
30		A1		0		2		0		3		0			0					RegA1(L5) -> Ai1, Mo(c0*L6) -> Ai2
31							0		3				3					C4		So(M1) -> Muli
32							0		3				3					C5		So(M2) -> Muli
33							0		3				3					C5		Ao(M3) -> Muli
34		C1		0		2		0		3		6			6					RegC1(M4) -> Ai1, Si1, Mo(M7) -> Ai2, Si2
35							2				3				7					So(M5) -> Ai2, Si2
36				3						2					6					Ao(M6) -> Ai2, Si2
37							0			1		7								So(M4-M7) -> Ai1, Si1
38				1				0				6								Ao(M4+M7) -> Ai1, Si1
39							0		3				3					C6		So[(M4-M7)-M5] -> Mi

40				1				2					2					C7		Ao[(M4-M7)+M5] -> Mi
41				1				2					2					C8		Ao[(M4+M7)+K6] -> Mi
42							0		3				3					C9		So[(M4+M7)-K6] -> Mi
43						2				1				0					D0	Mo(c3*M0) - > Out
44						2				1				0					D1	Mo(c4*M1) - > Out
45						2				1				0					D2	Mo(c5*M2) - > Out
46						2				1				0					D3	Mo(c5*M3) - > Out
47						2				1				0					D4	Mo(c6*[(M4-M7)-M5]) - > Out
48						2				1				0					D5	Mo(c7*[(M4-M7)+M5]) - > Out
49						2				1				0					D6	Mo(c8*[(M4+M7)+M6]) - > Out
50						2				1				0					D7	Mo(c9*[(M4+M7)-M6]) - > Out

C Program for evaluating the error in DCT/IDCT core

Generation of data set

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

double          pi;
double          onedctresult[8];
double          twodctresult[64];
union           hexcontent {
                        long    half[2];
                        double  full;
                        };


//Define Function 2-D DCT
void            twodct(long twoinput[])
{
int                     i,j,u,v;
double          input8x8[8][8];
double          temp;

                        for (i=0; i<=7; i++)
                        {       for (j=0; j<=7; j++)
                                {
                                        input8x8[i][j]=twoinput[8*i+j];
                                }
                        }

                        //Direct 2D
                        for (u=0; u<=7; u++)
                        {       for (v=0;v<=7;v++)
                                {
                                        temp = 0;
                                        for (i=0; i<=7; i++)
                                        {       for (j=0; j<=7; j++)
                                                {
                                                        temp +=
input8x8[i][j]*cos((2*i+1)*u*pi/16)*cos((2*j+1)*v*pi/16);
                                                }
                                        }
                                        temp = 0.25*temp*((u==0)/pow(2,
0.5)+(u!=0))*((v==0)/pow(2, 0.5)+(v!=0));
                                        if (temp > 2047)
                                                temp = 2047;
                                        if (temp < -2048)
                                                temp = -2048;

                                        twodctresult[u*8+v] = temp;
                                }
                        }
}



//Define Function Inverse 2-D DCT
void            twoidct(long twoinput[])
{
int                     i,j,u,v;
double          input8x8[8][8];
double          temp;

                        for (i=0; i<=7; i++)
                        {       for (j=0; j<=7; j++)
                                {
                                        input8x8[i][j]=twoinput[8*i+j];
                                }
                        }

                        //Direct 2D
                        for (i=0; i<=7; i++)
                        {       for (j=0;j<=7;j++)
                                {
                                        temp = 0;
                                        for (u=0; u<=7; u++)
                                        {       for (v=0; v<=7; v++)
                                                {
                                                        temp +=
input8x8[u][v]*cos((2*i+1)*u*pi/16)*cos((2*j+1)*v*pi/16)*((u==0)/pow(2,
0.5)+(u!=0))*((v==0)/pow(2, 0.5)+(v!=0));
                                                }
                                        }
                                        temp = 0.25*temp;
                                        if (temp > 255)
                                                temp = 255;
                                        if (temp < -256)
                                                temp = -256;
                                        //twodctresult[i*8+j] = 0.25*temp;
                                        twodctresult[i*8+j] = temp;
                                }
                        }
}


long randnum(L, H)
long L,H;
{
static long             randx = 1; /*long is 32 bits*/
static double   z= (double) 0x7fffffff;

long                    i,j;
double                  x;      /*double is 64 bits*/
                                randx = (randx*1103515245)+12345;
                                i = randx & 0x7ffffffe;
                                x = ((double)i)/z;
                                x = x*(L+H+1);
                                j = x;
                                return(j-L);
}

long roundup(double testnumber)
{
double                  reminder;
long                    result;

                                result = testnumber;
                                reminder = testnumber - result;
                                if (reminder >= 0.5)
                                        result += 1;
                                else if (reminder <= -0.5)
                                        result -= 1;
                                return result;
}

main()
{
long                    L, H;
int long                result[64];
int                             kk, ll, m, n;
long                    idctcoeff[64];
char                     filename[]="data00.dat";
//char                  odct_file[]="odct00.dat";
char                    idct_file[]="idct00.dat";
char                    fdct_file[]="fdct00.dat";
//double                        temp;
union hexcontent        upperbound;
FILE                    *result_id, *dct_id2, *dct_id3;//*dct_id,

                                pi=atan(1)*4;

                                upperbound.half[0]=0x00000001;
                                upperbound.half[1]=0x40aff000;
```

```
                             printf("Please enter the Lower Bound....\n");
                             scanf("%d", &L);
                             printf("Please enter the Upper Bound....\n");
                             scanf("%d", &H);
                             for (m=0; m<=9; m++)
                             {      filename[4] = filename[4] + m;
                                    //odct_file[4] = odct_file[4] + m;
                                    idct_file[4] = idct_file[4] + m;
                                    fdct_file[4] = fdct_file[4] + m;
                                    for (n=0; n<=9; n++)
                                    {
                                           filename[5] = filename[5] + n;
                                           //odct_file[5] = odct_file[5] + n;
                                           idct_file[5] = idct_file[5] + n;
                                           fdct_file[5] = fdct_file[5] + n;
                                           result_id = fopen(filename, "w");
                                           //dct_id  = fopen(odct_file, "w");
                                           dct_id2 = fopen(idct_file, "w");
                                           dct_id3 = fopen(fdct_file, "w");
                                           for (ll=0; ll<=99; ll++)
                                           {
                                                  for (kk=0; kk<=63; kk++)
                                                  {      result[kk] = randnum(L,
H);

                                                         fprintf(result_id,
"%ld\n", result[kk]);

                                                  }
                                                  twodct(result);
                                                  for (kk=0; kk<=63; kk++)
                                                  {
                                                         //fprintf(dct_id,
"%20.15lf\n", twodctresult[kk]);

       idctcoeff[kk]=roundup(twodctresult[kk]);
                                                         fprintf(dct_id2, "%d\n",
idctcoeff[kk]);

                                                  }
                                                  twoidct(idctcoeff);
                                                  for (kk=0; kk<=63; kk++)
                                                  {
                                                         //fprintf(dct_id3,
"%20.15lf\n", twodctresult[kk]);

       idctcoeff[kk]=roundup(twodctresult[kk]);
                                                         fprintf(dct_id3, "%d\n",
idctcoeff[kk]);
                                                  }
                                           }
                                           fclose(result_id);
                                           //fclose(dct_id);
                                           fclose(dct_id2);
                                           fclose(dct_id3);
                                           filename[5]='0';
                                           //odct_file[5]='0';
                                           idct_file[5]='0';
                                           fdct_file[5]='0';
                                    }
                                    filename[4]='0';
                                    //odct_file[4]='0';
                                    idct_file[4]='0';
                                    fdct_file[4]='0';
                             }
                             return 0;
}
```

Testing of DCT/IDCT architecture

```
//
//  This is program is used to generate a Inverse DCT result
//  from Forward DCT coefficients.
//
//  The input files "idctXX.dat" contain 12-bit DCT coeffieceients.
//  The output files 'nfdctXX.dat" contain 9-bit reconstrcuted
//  pixel values.
//

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

long    trunvalue[7];
long    mul_product[4][8];
long    dctresult_i[8];
double  dctresult_f[8];

void mul_coeff(int bit_length)
{
double                  value[7];
int                            i;
long                    roundup;

                               value[0]=cos(atan(1));
                               value[1]=cos(atan(1)/2);
                               value[2]=sin(atan(1)/2);
                               value[3]=cos(atan(1)/4);
                               value[4]=cos(3*atan(1)/4);
                               value[5]=sin(3*atan(1)/4);
                               value[6]=sin(atan(1)/4);


                               for (i=0;i<=6;i++)
                               {
                                      trunvalue[i] = value[i]*pow(2,28);
                                      //Round up
                                      roundup = (trunvalue[i] >> (28 - bit_length)) &
0x00000001;
                                      //create the coeff. at given bit length
                                      trunvalue[i] = (trunvalue[i] >> (28 - bit_length
+ 1)) + roundup;
                               }

}

void mul_matrix(long input1[4], long input2[4], int coeff_length, int trun_length)
{
long                    matrix1[4][4], matrix2[4][4], roundup;
int                            i, j;

                               //Form the coeff. Matrix
                               matrix1[0][0]=trunvalue[0];
        matrix1[0][1]=trunvalue[1];
                               matrix1[0][2]=trunvalue[0];
        matrix1[0][3]=trunvalue[2];
                               matrix1[1][0]=trunvalue[0];
        matrix1[1][1]=trunvalue[2];
                               matrix1[1][2]=-1*trunvalue[0];        matrix1[1][3]=-
1*trunvalue[1];
                               matrix1[2][0]=trunvalue[0];           matrix1[2][1]=-
1*trunvalue[2];
                               matrix1[2][2]=-1*trunvalue[0];
        matrix1[2][3]=trunvalue[1];
                               matrix1[3][0]=trunvalue[0];           matrix1[3][1]=-
1*trunvalue[1];
                               matrix1[3][2]=trunvalue[0];           matrix1[3][3]=-
1*trunvalue[2];

                               matrix2[0][0]=trunvalue[3];
        matrix2[0][1]=trunvalue[4];
                               matrix2[0][2]=trunvalue[5];
        matrix2[0][3]=trunvalue[6];
```

```
                                matrix2[1][0]=trunvalue[4];             matrix2[1][1]=-
1*trunvalue[6];
                                matrix2[1][2]=-1*trunvalue[3];          matrix2[1][3]=-
1*trunvalue[5];
                                matrix2[2][0]=trunvalue[5];             matrix2[2][1]=-
1*trunvalue[3];
                                matrix2[2][2]=trunvalue[6];
        matrix2[2][3]=trunvalue[4];
                                matrix2[3][0]=trunvalue[6];             matrix2[3][1]=-
1*trunvalue[5];
                                matrix2[3][2]=trunvalue[4];             matrix2[3][3]=-
1*trunvalue[3];

                                //Matrix Multiplcation
                                for (i=0;i<=3;i++)
                                {
                                        for (j=0;j<=3;j++)
                                        {
                                                mul_product[j][i]=matrix1[j][i] *
input1[i];
                                                mul_product[j][4+i]=matrix2[j][i] *
input2[i];
                                        }
                                }

                                //Truncation
                                for (i=0;i<=3;i++)
                                {
                                        for (j=0;j<=7;j++)
                                                if (trun_length == 0)
        mul_product[i][j]=(mul_product[i][j] >> trun_length);
                                                else
                                                {
                                                        roundup = (mul_product[i][j] >>
(trun_length-1)) & 0x00000001;
        mul_product[i][j]=(mul_product[i][j] >> trun_length) + roundup;
                                                }
                                }

}

void onedct(long input[8], int input_length, int coeff_length, int mul_trun_length,
int final_length, int second)
{
long                    half1[4], half2[4];
int                             i;
long                    stage31[4], stage32[4], stage33[4], stage34[4], stage41[4],
stage42[4];
long                    result[8], reduce_length, roundup;
long                    temp;

                                //First Stage
                                for (i=0;i<=3;i++)
                                {
                                        half1[i]=input[2*i];
                                        half2[i]=input[(2*i)+1];
                                }

                                //Second Stage
                                mul_matrix(half1, half2, coeff_length,
mul_trun_length);

                                for (i=0;i<=3; i++)
                                {
                                        stage31[i]=mul_product[i][0]+mul_product[i][1];
                                        stage32[i]=mul_product[i][2]+mul_product[i][3];
                                        stage33[i]=mul_product[i][4]+mul_product[i][5];
                                        stage34[i]=mul_product[i][6]+mul_product[i][7];
                                        stage41[i]=stage31[i]+stage32[i];
                                        stage42[i]=stage33[i]+stage34[i];
                                }

                                //Third Stage
                                for (i=0;i<=3;i++)
                                {
```

```
                                                result[i] = stage41[i]+stage42[i];
                                                result[7-i] = stage41[i]-stage42[i];
                                        }

                                        //Round up and Truncation
                                        reduce_length = input_length+coeff_length-
mul_trun_length+2-final_length-second-3;
                                        for (i=0;i<=7;i++)
                                        {
                                                roundup = (result[i] >> (reduce_length-1)) &
0x00000001;
                                                temp = (result[i] >> reduce_length)+ roundup;
                                                if (second == 1)
                                                {       if (((temp & 0x80000000) == 0x00000000)
&& ((temp & 0x00000100) == 0x00000100))
                                                        temp = 255;
                                                        if (((temp & 0x80000000) == 0x80000000)
&& ((temp & 0x00000100) == 0x00000000))
                                                                temp = -256;
                                                }
                                                dctresult_i[i] = temp;
                                                dctresult_f[i] =
dctresult_i[i]/pow(2,final_length-input_length-3+1+second);
                                        }
}

main()
{
int                                     input[8][8];
int                                     inputfile[64];
long                            temp[8];
long                            result[8][8];
double                          result_f[8][8];
int                                     input_length, coeff_length, mul_trun_length,
final_length;
int                                     i, j, kk, ll, m, n, test;
char                            idct_file[]="idct00.dat";
char                            fdct_file[]="nfdct00.dat";
FILE                            *idct_id, *fdct_id;

                                        coeff_length = 15;
                                        mul_coeff(coeff_length);

                                        for (m=0; m<=9; m++)
                                        {       idct_file[4] = idct_file[4] + m;
                                                fdct_file[5] = fdct_file[5] + m;

                                                for (n=0; n<=9; n++)
                                                {       idct_file[5] = idct_file[5] + n;
                                                        fdct_file[6] = fdct_file[6] + n;

                                                        idct_id = fopen(idct_file, "r");
                                                        fdct_id = fopen(fdct_file, "w");
                                                        //Read 100times, 64 element in each time
                                                        for (kk=0;kk<=99;kk++)
                                                        {       for (ll=0;ll<=63;ll++)
                                                                        fscanf(idct_id, "%d\n",
&inputfile[ll]);

                                                                //Reorder the input vector into
8x8 matrix
                                                                for (i=0;i<=7;i++)
                                                                {
                                                                        for (j=0;j<=7;j++)
                                                                                input[i][j]=
inputfile[i*8+j];
                                                                }

                                                                input_length = 12;
                                                                mul_trun_length = 7;
                                                                final_length = 15;

                                                                for (i=0;i<=7;i++)
                                                                {
                                                                        for (j=0;j<=7;j++)
                                                                                temp[j] =
input[j][i];
```

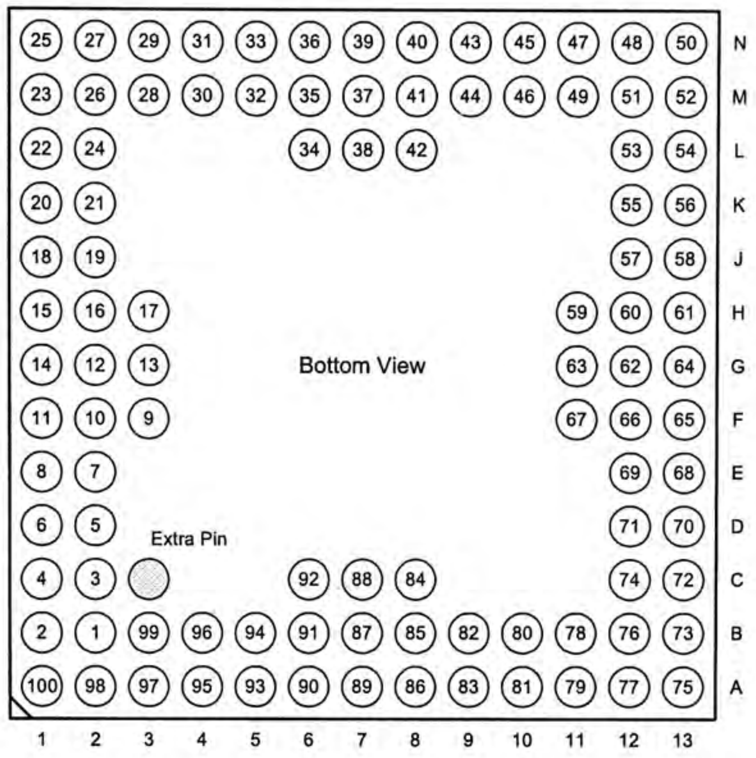
```
                                                             onedct(temp,
input_length, coeff_length, mul_trun_length, final_length, 0);
                                                             for (j=0;j<=7;j++)
                                                             {
                                                                    result[j][i] =
dctresult_i[j];
                                                                    result_f[j][i] =
dctresult_f[j];
                                                                    if
(abs(dctresult_i[j]) >= pow(2, final_length-1)-1)
                                                                    {

        printf("Excess Limit, x=%d, %08x\n", dctresult_i[j], dctresult_i[j]);

        scanf("%d", &test);
                                                                    }
                                                             }
                                                     }

                                                     input_length = 15;
                                                     mul_trun_length = 9;
                                                     final_length = 9;

                                                     for (i=0;i<=7;i++)
                                                     {
                                                            for (j=0;j<=7;j++)
                                                                   temp[j] =
result[i][j];
                                                            onedct(temp,
input_length, coeff_length, mul_trun_length, final_length, 1);
                                                            for (j=0;j<=7;j++)
                                                            {
                                                                   result[i][j] =
dctresult_i[j];
                                                            }
                                                     }

                                                     for (i=0;i<=7;i++)
                                                     {
                                                            for (j=0;j<=7;j++)
                                                                   fprintf(fdct_id,
"%d\n", result[i][j]);
                                                     }

                                             }
                                             fclose(idct_id);
                                             fclose(fdct_id);
                                             idct_file[5] = '0';
                                             fdct_file[6] = '0';
                                      }
                                      idct_file[4] = '0';
                                      fdct_file[5] = '0';
                               }
                               return 0;
}
```

Pin Assignments of the Programmable DSP Processor Chip

Pin Number	Pin Name	IN / OUT	Description
1	request	IN	Input data request signal
2	VDD	IN	
3	GND	IN	
4	start	IN	Input data start signal
5	in_reset	IN	Input data buffer reset signal
6	empty	OUT	Input data buffer empty signal
7	done	OUT	Input data acknowledgement signal
8	VDD	IN	
9	GND	IN	
10	instr_done	IN	Instruction acknowledgement signal
11	instr_rq	OUT	Instruction request signal
12	instr<0>	IN	Instruction bit0
13	instr<1>	IN	Instruction bit1
14	VDD	IN	
15	GND	IN	
16	instr<2>	IN	Instruction bit2
17	instr<3>	IN	Instruction bit3
18	instr<4>	IN	Instruction bit4
19	VDD	IN	
20	GND	IN	
21	instr<5>	IN	Instruction bit5
22	instr<6>	IN	Instruction bit6
23	instr<7>	IN	Instruction bit7
24	instr<8>	IN	Instruction bit8
25	instr<9>	IN	Instruction bit9
26	VDD	IN	
27	GND	IN	
28	cmplt_clr_instr	IN	Clear the instruction input buffer
29	mode	IN	Switch of the cyclic FIFO in instruction memory. 0=cyclic, 1=receive instruction from user
30	get_output	IN	Output mode 1 : Get the output handshaking signal
31	VDD	IN	
32	GND	IN	
33	open_cmplp	IN	Output mode 1 : Get the output handshaking signal
34	out_full	OUT	Output mode 0 : Output buffer full signal
35	out_ready	OUT	Output mode 0 : Output request signal
36	out_buf_in<0>	OUT	Output mode 0 : Output data bit0
37	out_buf_in<1>	OUT	Output mode 0 : Output data bit1
38	VDD	IN	
39	GND	IN	
40	out_buf_in<2>	OUT	Output mode 0 : Output data bit2
41	out_buf_in<3>	OUT	Output mode 0 : Output data bit3

42	out_buf_in<4>	OUT	Output mode 0 : Output data bit4
43	out_buf_in<5>	OUT	Output mode 0 : Output data bit5
44	out_buf_in<6>	OUT	Output mode 0 : Output data bit6
45	out_buf_in<7>	OUT	Output mode 0 : Output data bit7
46	VDD	IN	
47	GND	IN	
48	out_buf_in<8>	OUT	Output mode 0 : Output data bit8
49	cmplt_out	OUT	Output mode 1 : Output request signal
50	cmplt_out_d	OUT	Output mode 1 : cmplt_out ÷ 4
51	out_sel	IN	Output mode selection : mode 0 for data verification, mode 1 for speed measurement
52	VDD	IN	
53	GND	IN	
54	out_aki	IN	Output mode 0 : Output acknowledgement signal
55	in1<0>	IN	Input data bit0
56	in1<1>	IN	Input data bit1
57	in1<2>	IN	Input data bit2
58	in1<3>	IN	Input data bit3
59	VDD	IN	
60	GND	IN	
61	reset	IN	Global reset signal
62	in1<4>	IN	Input data bit4
63	in1<5>	IN	Input data bit5
64	in1<6>	IN	Input data bit6
65	in1<7>	IN	Input data bit7
66	VDD	IN	
67	GND	IN	
68	in<8>	IN	Input data bit8

Pin Assignments of the 1D DCT/IDCT Core Chip

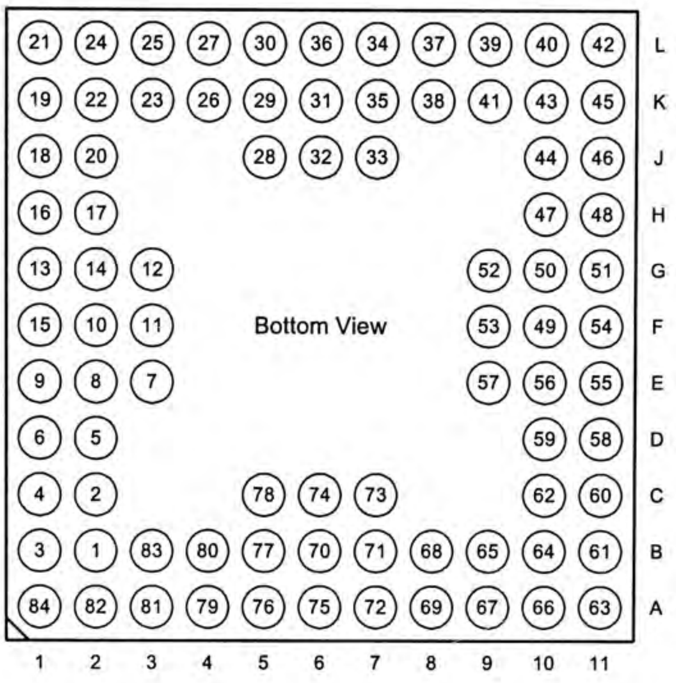


Pin Number	Pin Name	In / Out	Description
1	VDD	IN	
2	GND	IN	
3	In<5>	IN	Input data bit5
4	In<4>	IN	Input data bit4
5	In<3>	IN	Input data bit3
6	In<2>	IN	Input data bit2
7	VDD	IN	
8	GND	IN	
9	In<1>	IN	Input data bit1
10	In<0>	IN	Input data bit0
11	VDD	IN	
12	GND	IN	
13	test6	OUT	Testing signal from DCT coefficients memory 2
14	test8	OUT	Testing signal from data replicator 2
15	test10	OUT	Testing signal from multiplier 2
16	VDD	IN	
17	GND	IN	
18	test14	OUT	Testing signal from 22bit subtractor
19	test15	OUT	Testing signal from 22bit adder
20	VDD	IN	
21	GND	IN	
22	output_rq	OUT	Output mode 0 : Output request signal

23	Ckin	IN	Output mode 0 : Output acknowledgement signal
24	VDD	IN	
25	GND	IN	
26	Out<14>	OUT	Output mode 0 : Output data bit14
27	Out<13>	OUT	Output mode 0 : Output data bit13
28	Out<12>	OUT	Output mode 0 : Output data bit12
29	VDD	IN	
30	GND	IN	
31	Out<11>	OUT	Output mode 0 : Output data bit11
32	Out<10>	OUT	Output mode 0 : Output data bit10
33	Out<9>	OUT	Output mode 0 : Output data bit9
34	VDD	IN	
35	GND	IN	
36	Out<8>	OUT	Output mode 0 : Output data bit8
37	Out<7>	OUT	Output mode 0 : Output data bit7
38	Out<6>	OUT	Output mode 0 : Output data bit6
39	VDD	IN	
40	GND	IN	
41	Out<5>	OUT	Output mode 0 : Output data bit5
42	Out<4>	OUT	Output mode 0 : Output data bit4
43	Out<3>	OUT	Output mode 0 : Output data bit3
44	Out<2>	OUT	Output mode 0 : Output data bit2
45	VDD	IN	
46	GND	IN	
47	Out<1>	OUT	Output mode 0 : Output data bit1
48	Out<0>	OUT	Output mode 0 : Output data bit0
49	open_cmplp	IN	Output mode 1 : Get the output handshaking signal
50	out_sel	IN	Output mode selection : mode 0 for data verification, mode 1 for speed measurement
51	VDD	IN	
52	GND	IN	
53	test17	OUT	Testing signal from truncation unit
54	test16	OUT	Testing signal from DCT/IDCT switch 5
55	test13	OUT	Testing signal 21 bit adder
56	VDD	IN	
57	GND	IN	
58	complt_out	OUT	Output mode 1 : Output request signal
59	cmplt_out_d	OUT	Output mode 1 : cmplt_out ÷ 4
60	VDD	IN	
61	GND	IN	
62	get_out	IN	Output mode 1 : Get the output handshaking signal
63	test12	OUT	Testing signal from 20bit adder 2
64	test11	OUT	Testing signal from 20bit adder 1
65	VDD	IN	
66	GND	IN	
67	test9	OUT	Testing signal from multiplier 1
68	test7	OUT	Testing signal from data replicator 1
69	test5	OUT	Testing signal from DCT coefficients memory 1

70	VDD	IN	
71	GND	IN	
72	block2	IN	Set for the column operation
73	Idct	IN	Set for IDCT operation
74	VDD	IN	
75	GND	IN	
76	Reset	IN	Reset
77	Start	IN	Input data start signal
78	test4	OUT	Testing signal from 15bit subtractor
79	test3	OUT	Testing signal from 15bit adder
80	VDD	IN	
81	GND	IN	
82	test2	OUT	Testing signal from 1-to-2 MUX1
83	test1	OUT	Testing signal from input buffer
84	Done	OUT	Input data acknowledgement signal
85	input_rq	IN	Input data request signal
86	VDD	IN	
87	GND	IN	
88	In<14>	IN	Input data bit14
89	In<13>	IN	Input data bit13
90	In<12>	IN	Input data bit12
91	VDD	IN	
92	GND	IN	
93	In<11>	IN	Input data bit11
94	In<10>	IN	Input data bit10
95	In<9>	IN	Input data bit9
96	VDD	IN	
97	GND	IN	
98	In<8>	IN	Input data bit8
99	In<7>	IN	Input data bit7
100	In<6>	IN	Input data bit6

Pin Assignments of the Transpose Memory Chip

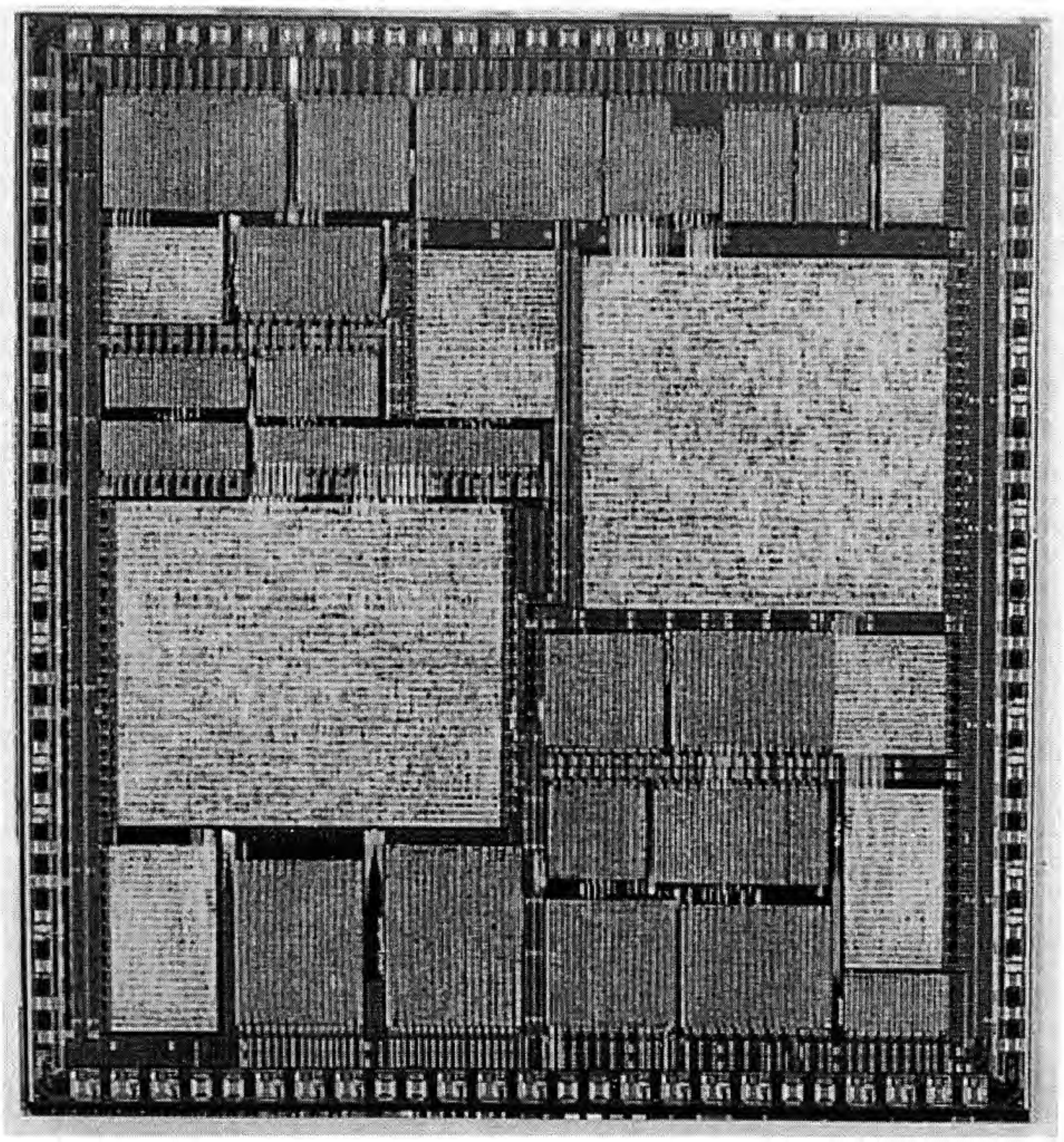


Pin Number	*Pin Name*	*In / Out*	*Description*
1	test1	OUT	Testing signal for LSB generator in write address generator
2	test2	OUT	Testing signal for MSB generator in write address generator
3	test3	OUT	Testing signal for in write address generator
4	VDD	IN	
5	GND	IN	
6	test4	OUT	Testing signal for LSB generator in read address generator
7	test5	OUT	Testing signal for MSB generator in read address generator
8	test6	OUT	Testing signal for in read address generator
9	idct	IN	Set for the IDCT operation
10	data_rq	IN	Input data request signal
11	VDD	IN	
12	GND	IN	
13	I<0>	IN	Input data bit0
14	I<1>	IN	Input data bit1
15	I<2>	IN	Input data bit2
16	I<3>	IN	Input data bit3
17	I<4>	IN	Input data bit4
18	VDD	IN	
19	GND	IN	
20	I<5>	IN	Input data bit5

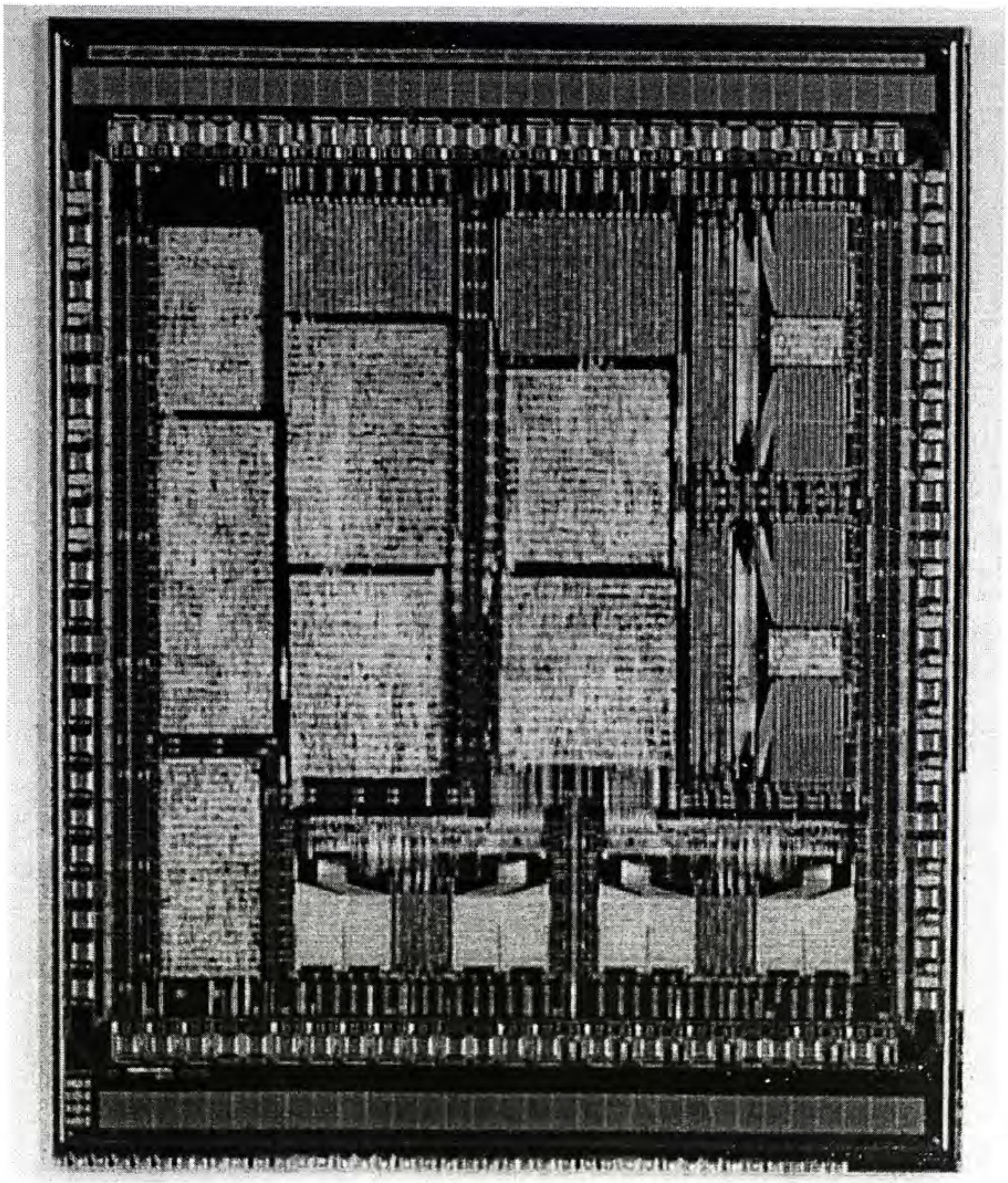
21	I<6>	IN	Input data bit6
22	I<7>	IN	Input data bit7
23	I<8>	IN	Input data bit8
24	I<9>	IN	Input data bit9
25	VDD	IN	
26	GND	IN	
27	I<10>	IN	Input data bit10
28	I<11>	IN	Input data bit11
29	I<12>	IN	Input data bit12
30	I<13>	IN	Input data bit13
31	I<14>	IN	Input data bit14
32	VDD	IN	
33	GND	IN	
34	Start	IN	Input data start signal
35	Done	OUT	Input data acknowledgement signal
36	test8	OUT	Testing signal from input multiplexing network
37	test9	OUT	Testing signal from input multiplexing network
38	test10	OUT	Testing signal from input multiplexing network
39	VDD	IN	
40	GND	IN	
41	test11	OUT	Testing signal from input multiplexing network
42	test7	OUT	Testing signal from input data buffer
43	test15	OUT	Testing signal from input multiplexing network
44	test16	OUT	Testing signal from input multiplexing network
45	test17	OUT	Testing signal from RAM block0
46	VDD	IN	
47	GND	IN	
48	test12	OUT	Testing signal from input multiplexing network
49	test13	OUT	Testing signal from input multiplexing network
50	test14	OUT	Testing signal from RAM block1
51	test18	OUT	
52	Dataout<14>	OUT	Output mode 0 : Output data bit14
53	VDD	IN	
54	GND	IN	
55	Dataout<13>	OUT	Output mode 0 : Output data bit13
56	Dataout<12>	OUT	Output mode 0 : Output data bit12
57	Dataout<11>	OUT	Output mode 0 : Output data bit11
58	Dataout<10>	OUT	Output mode 0 : Output data bit10
59	Dataout<9>	OUT	Output mode 0 : Output data bit9
60	VDD	IN	
61	GND	IN	
62	Dataout<8>	OUT	Output mode 0 : Output data bit8
63	Dataout<7>	OUT	Output mode 0 : Output data bit7
64	Dataout<6>	OUT	Output mode 0 : Output data bit6
65	Dataout<5>	OUT	Output mode 0 : Output data bit5
66	Dataout<4>	OUT	Output mode 0 : Output data bit4
67	VDD	IN	
68	GND	IN	

69	Dataout<3>	OUT	Output mode 0 : Output data bit3
70	Dataout<2>	OUT	Output mode 0 : Output data bit2
71	Dataout<1>	OUT	Output mode 0 : Output data bit1
72	Dataout<0>	OUT	Output mode 0 : Output data bit0
73	dataout_rq	OUT	Output mode 0 : Output data request signal
74	VDD	IN	
75	GND	IN	
76	Ckin	IN	Output mode 0 : Output acknowledgement signal
77	Reset	IN	Reset
78	output_sel	IN	Output mode selection : mode 0 for data verification, mode 1 for speed measurement
79	cmplt_out_d	OUT	Output mode 1 : cmplt_out ÷ 4
80	cmplt_out	OUT	Output mode 1 : Output request signal
81	VDD	IN	
82	GND	IN	
83	open_cmplp	IN	Output mode 1 : Get the output handshaking signal
84	get_output	IN	Output mode 1 : Get the output handshaking signal

Chip microphotograph of the 1D DCT/IDCT core

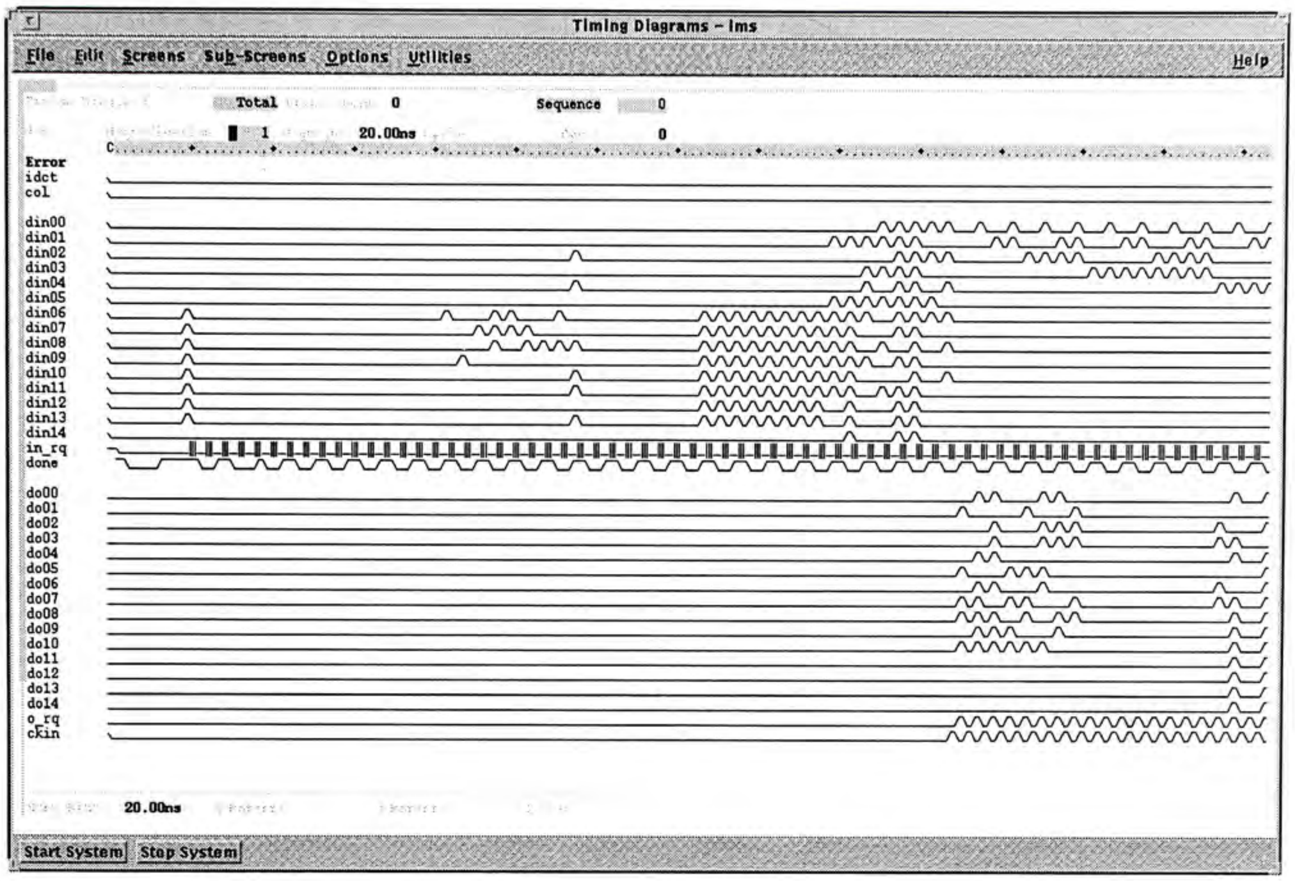


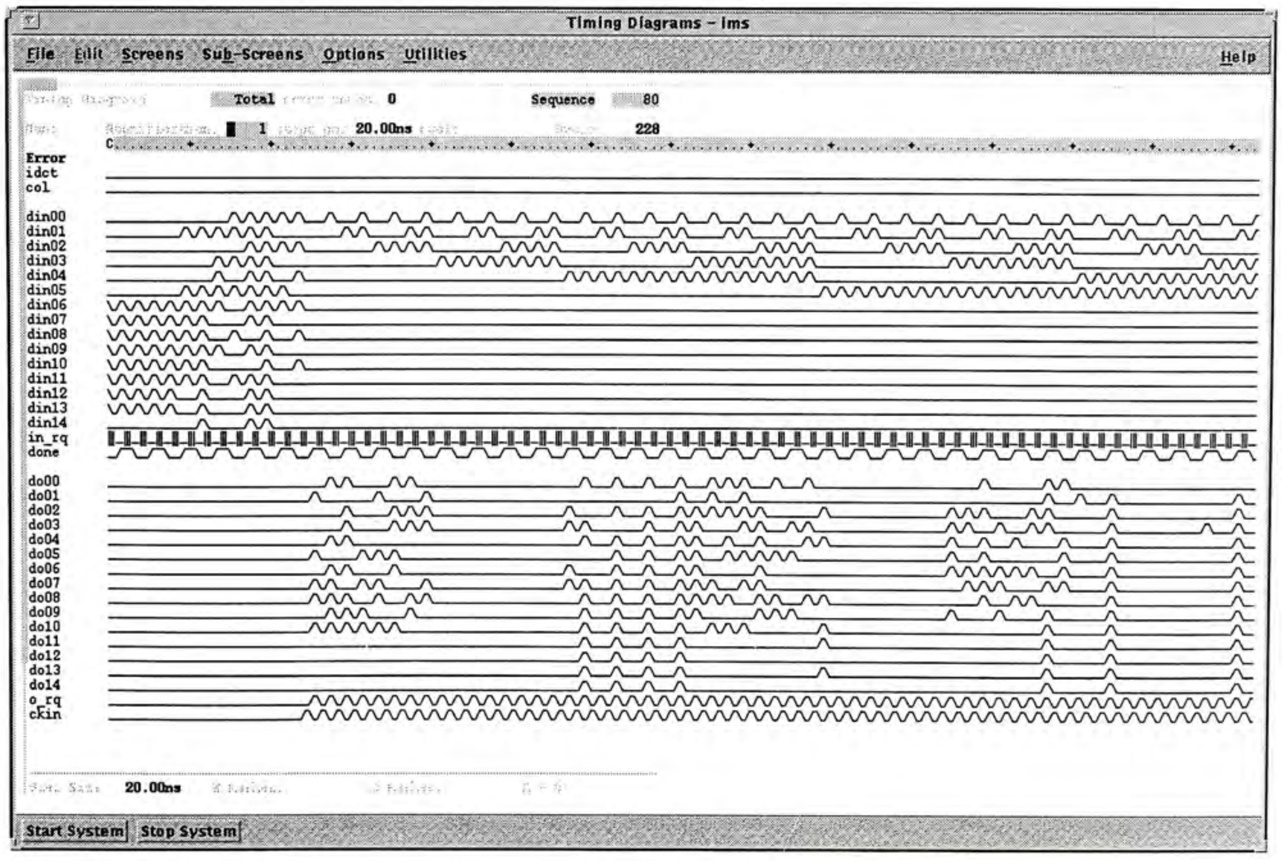
Chip Microphotograph of the Transpose Memory



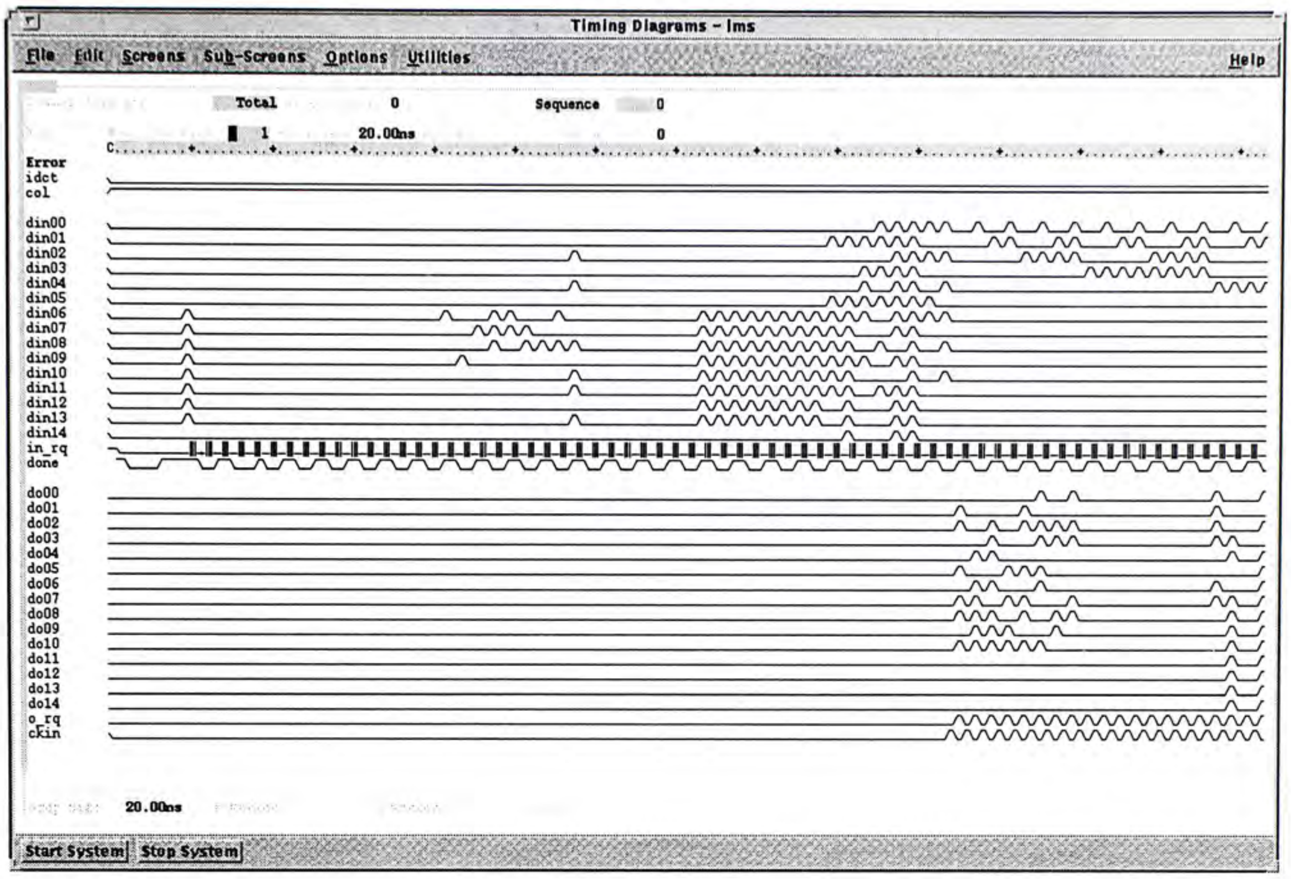
Measured Waveforms of 1D DCT/IDCT Chip

Waveforms of DCT row operation





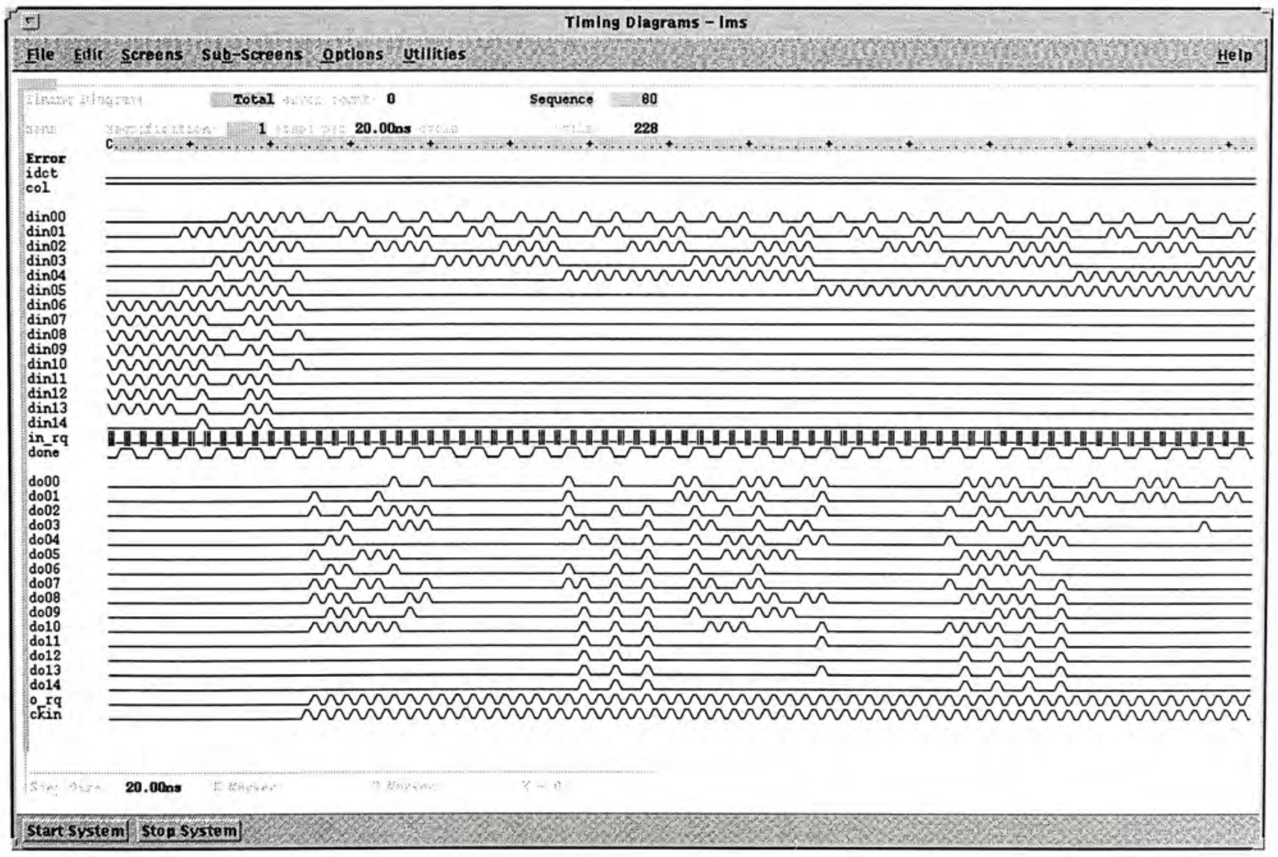
Waveforms of DCT column operation



Total 0 Sequence 0

1 20.00ns 0

20.00ns

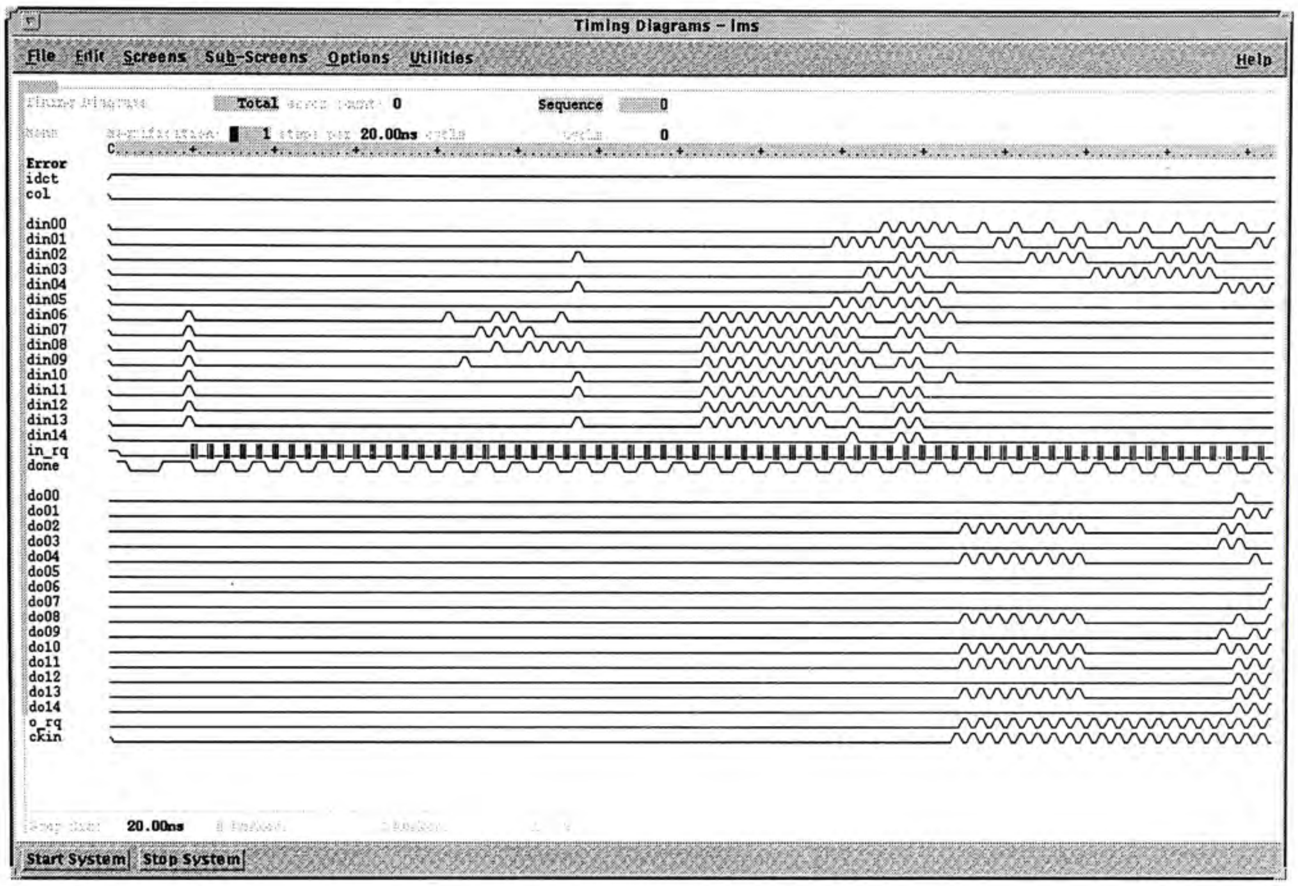


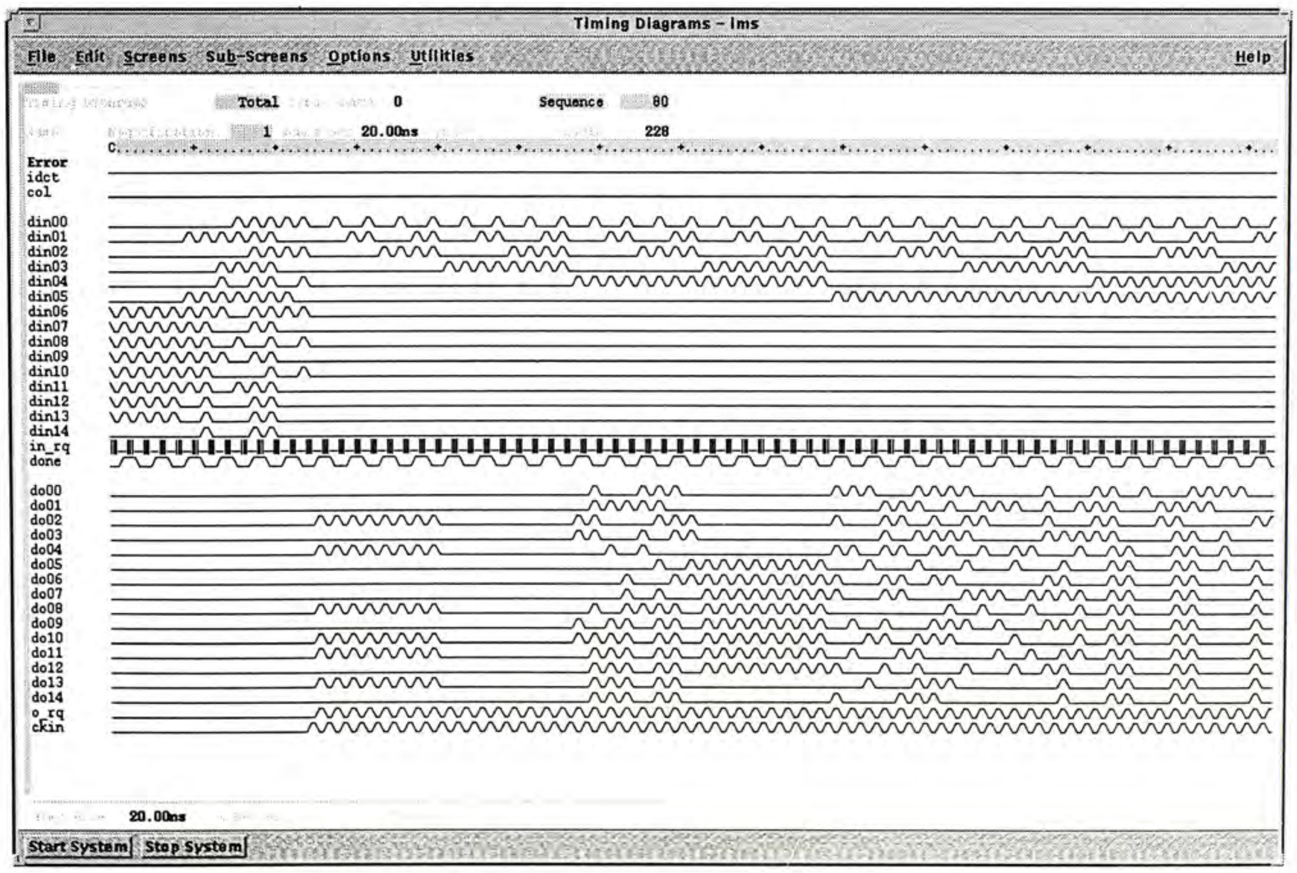
Timing Diagram Total 0 Sequence 80

Magnification 1 20.00ns 228

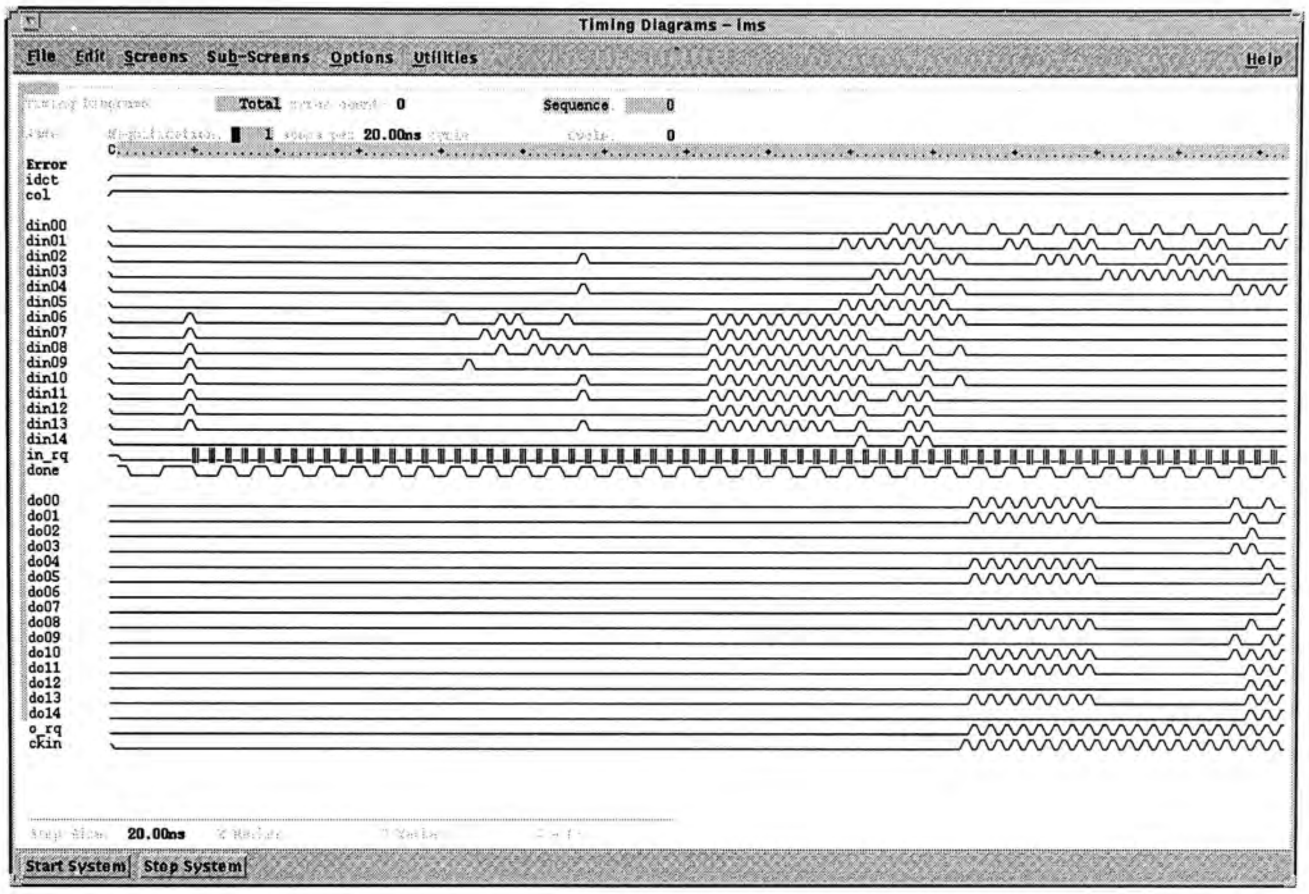
Step Size 20.00ns

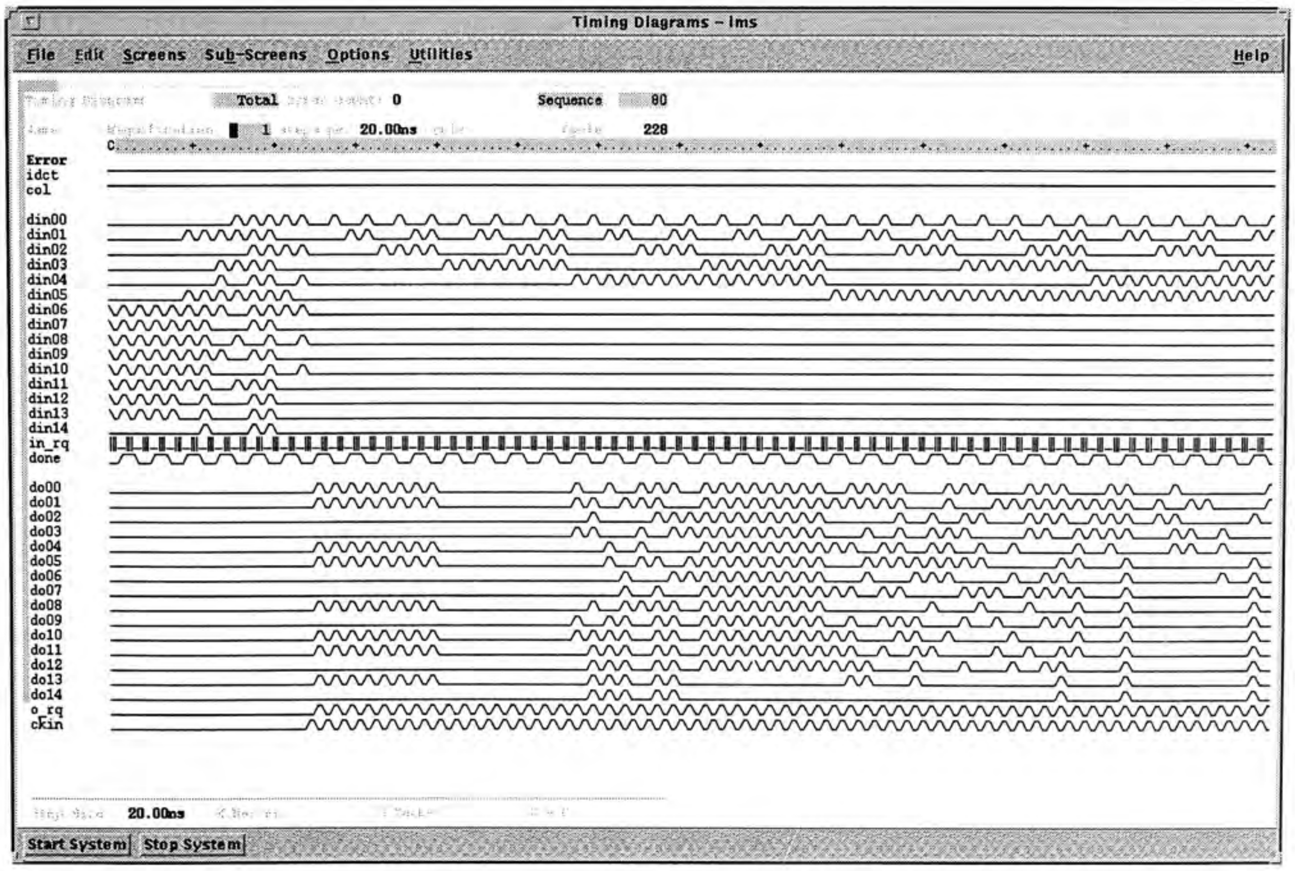
Waveforms of IDCT row operation





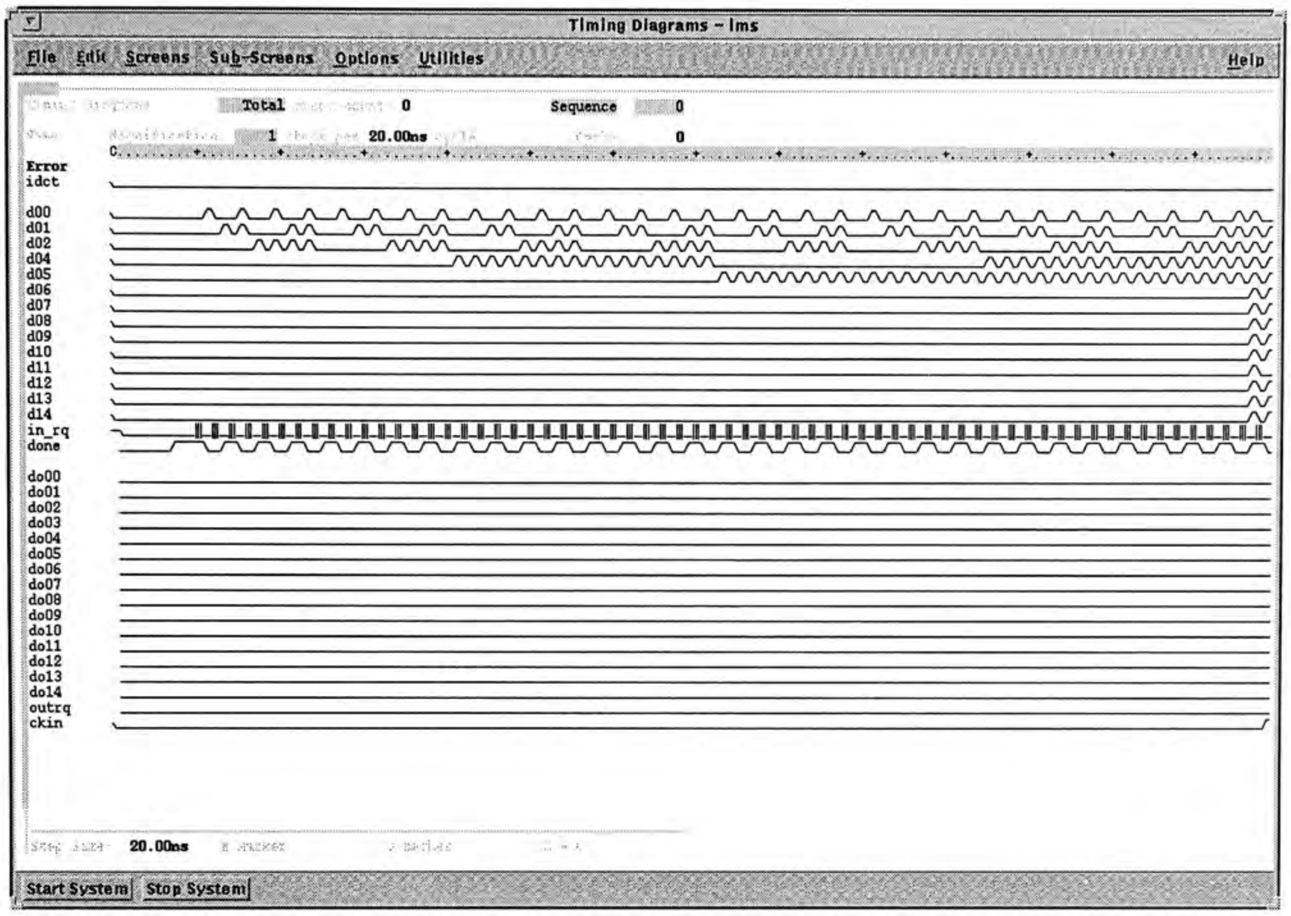
Waveforms of IDCT row operation

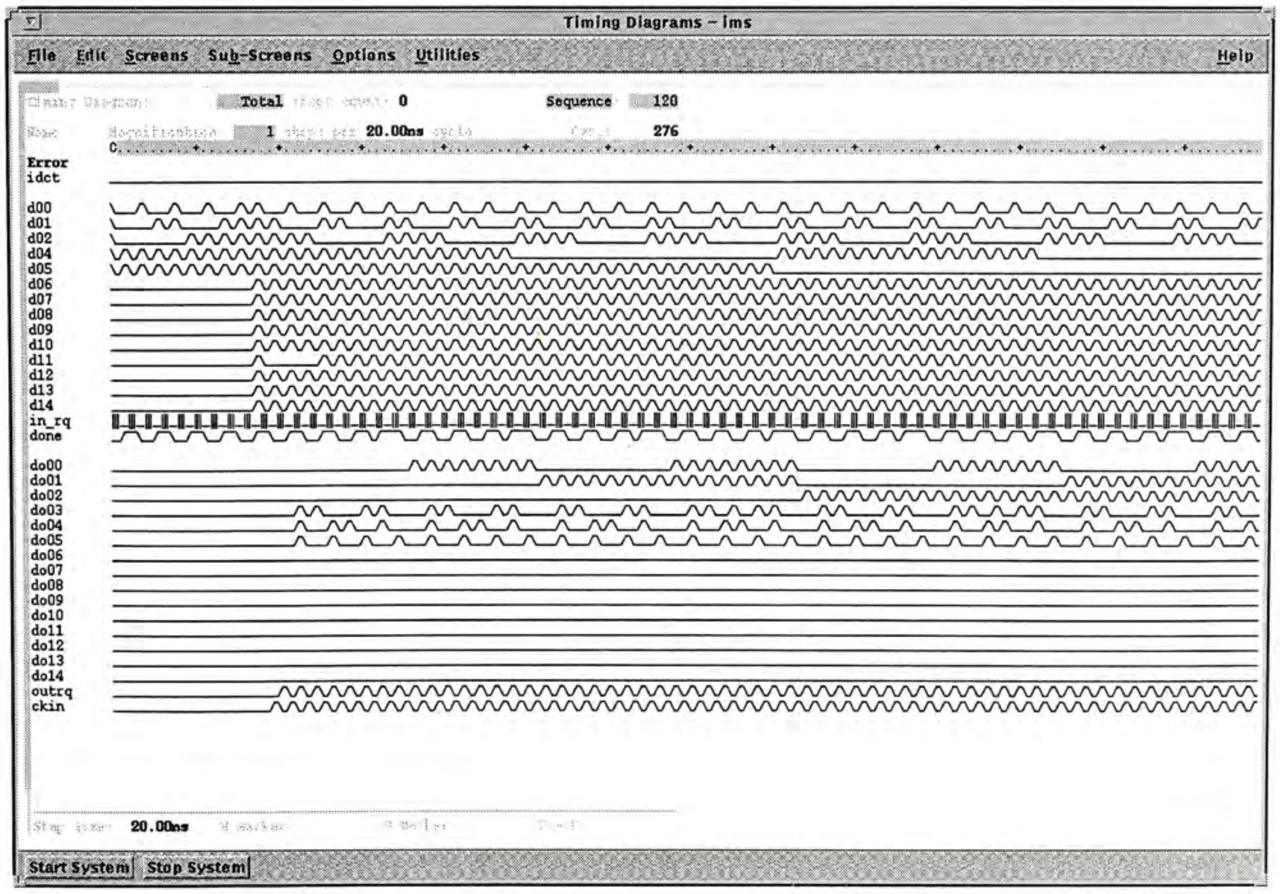




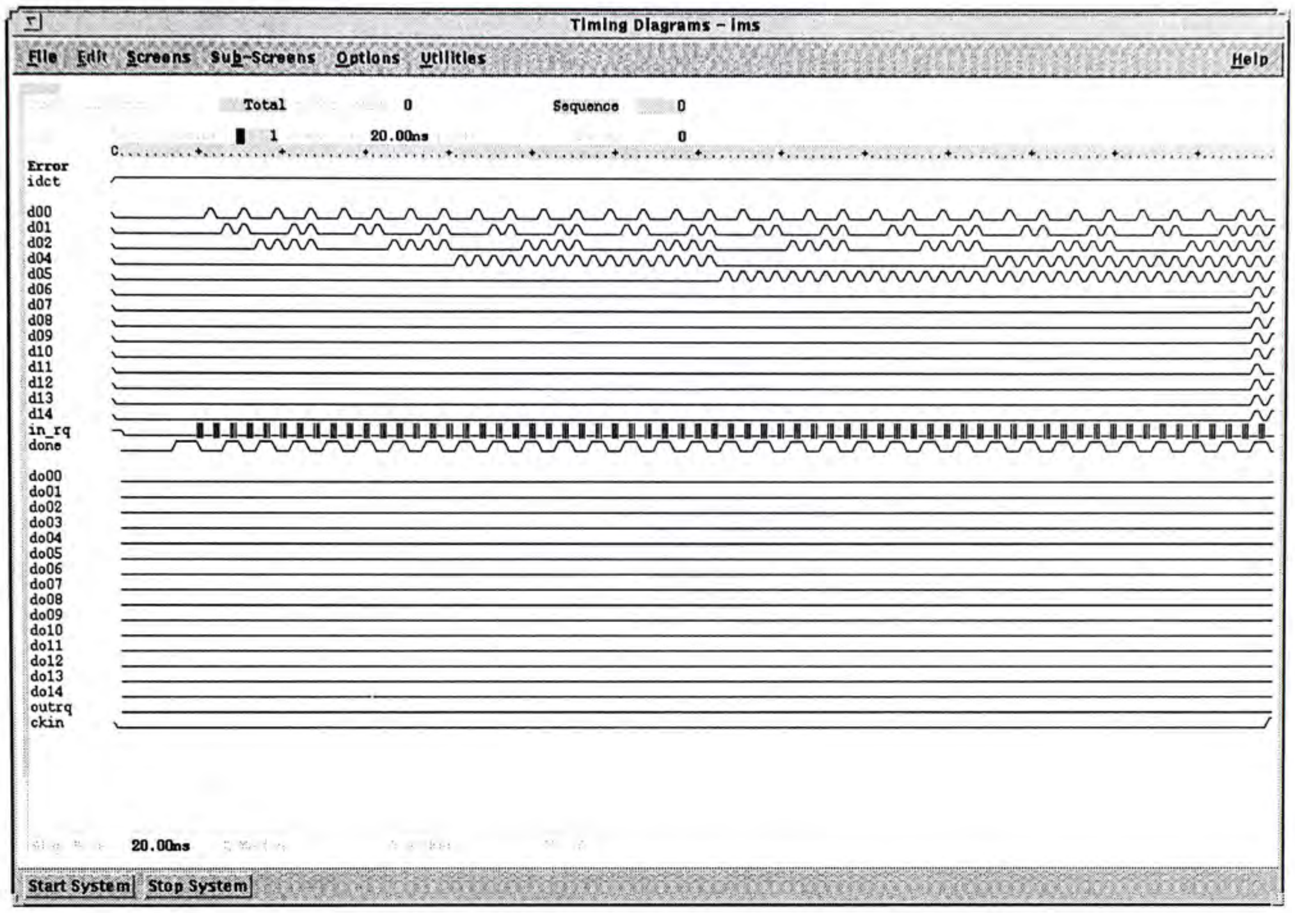
Measured Waveforms of Transpose Memory Chip

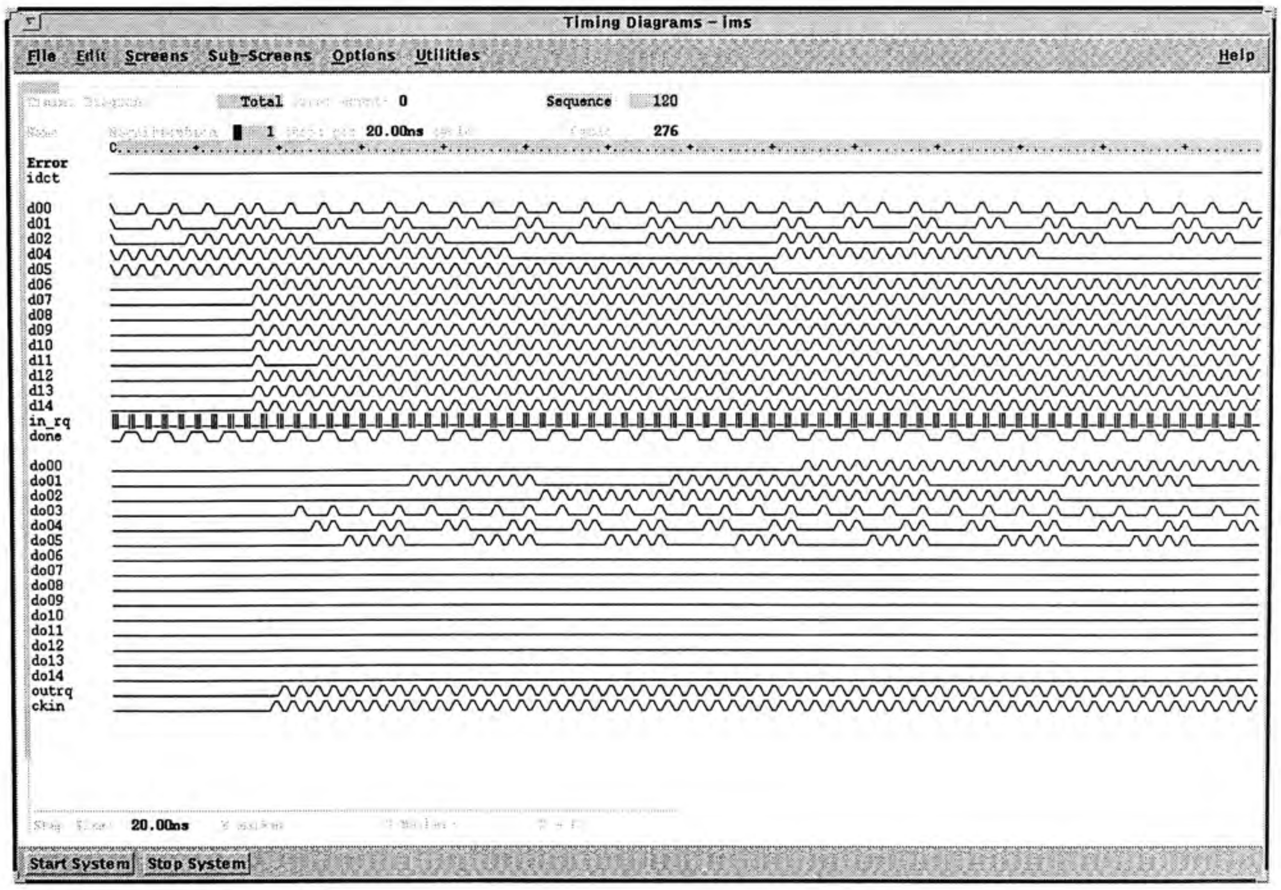
Waveforms of DCT operation mode





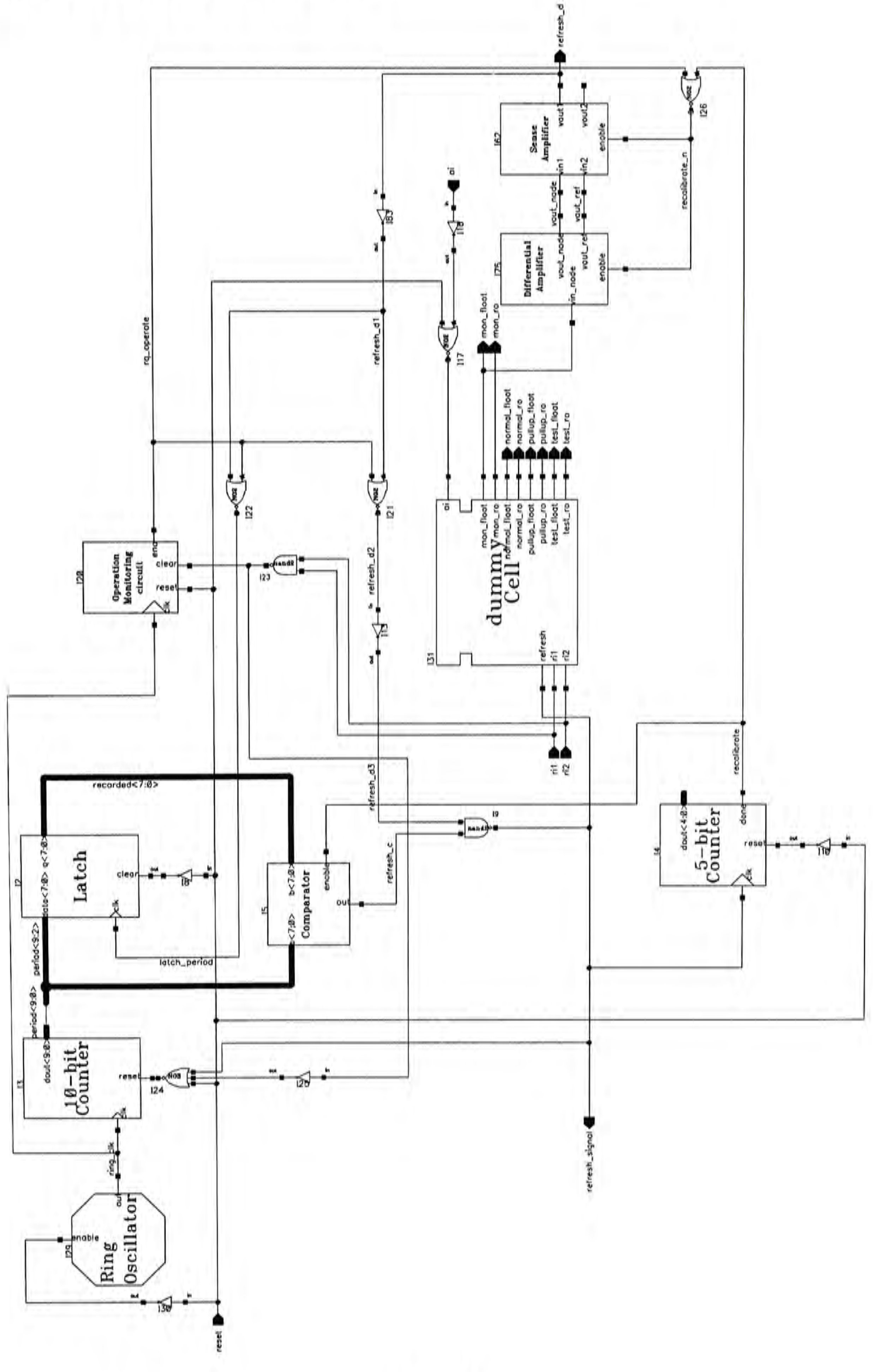
Waveforms of IDCT operation mode



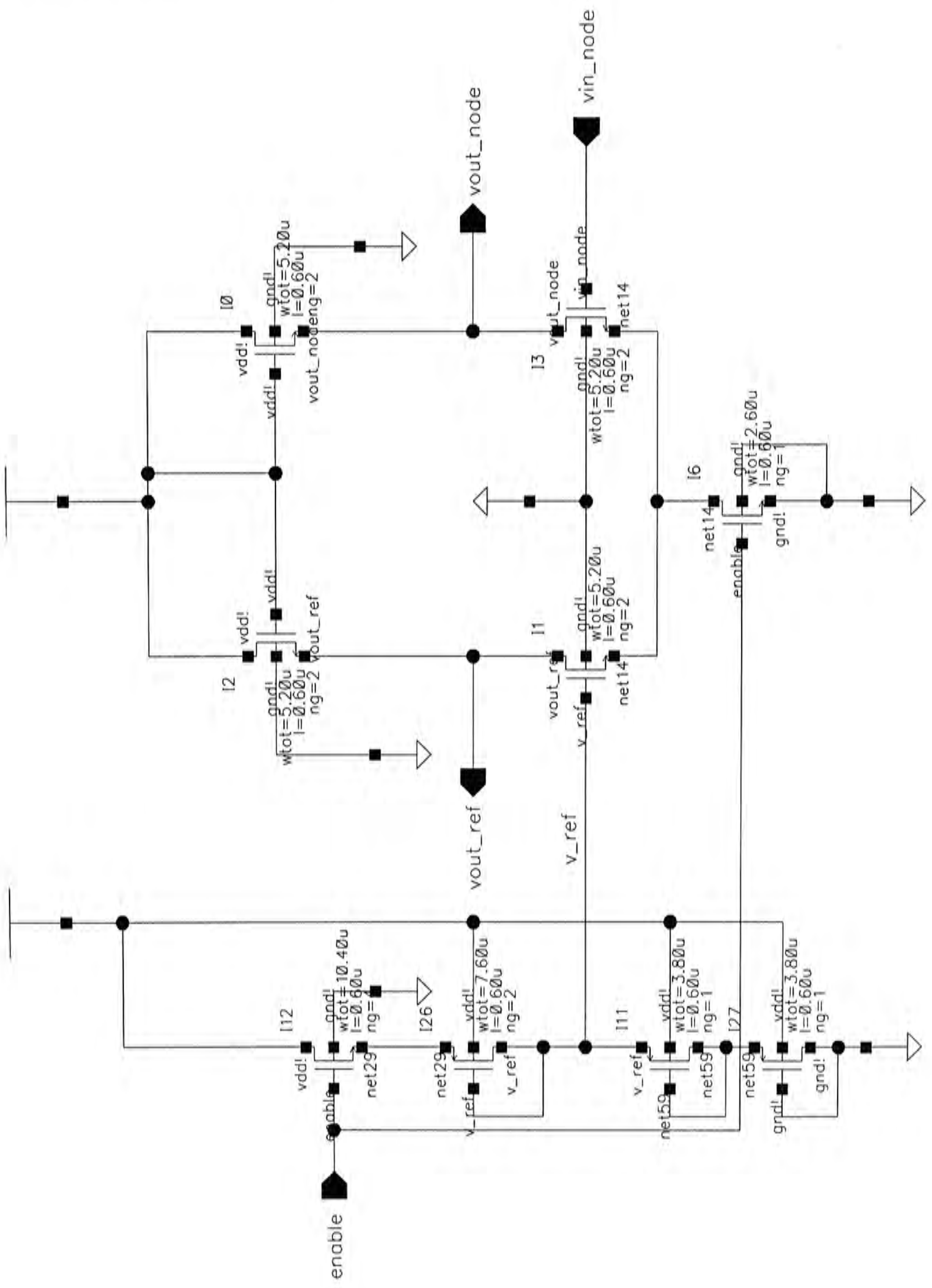


Schematics of Refresh Control Circuit

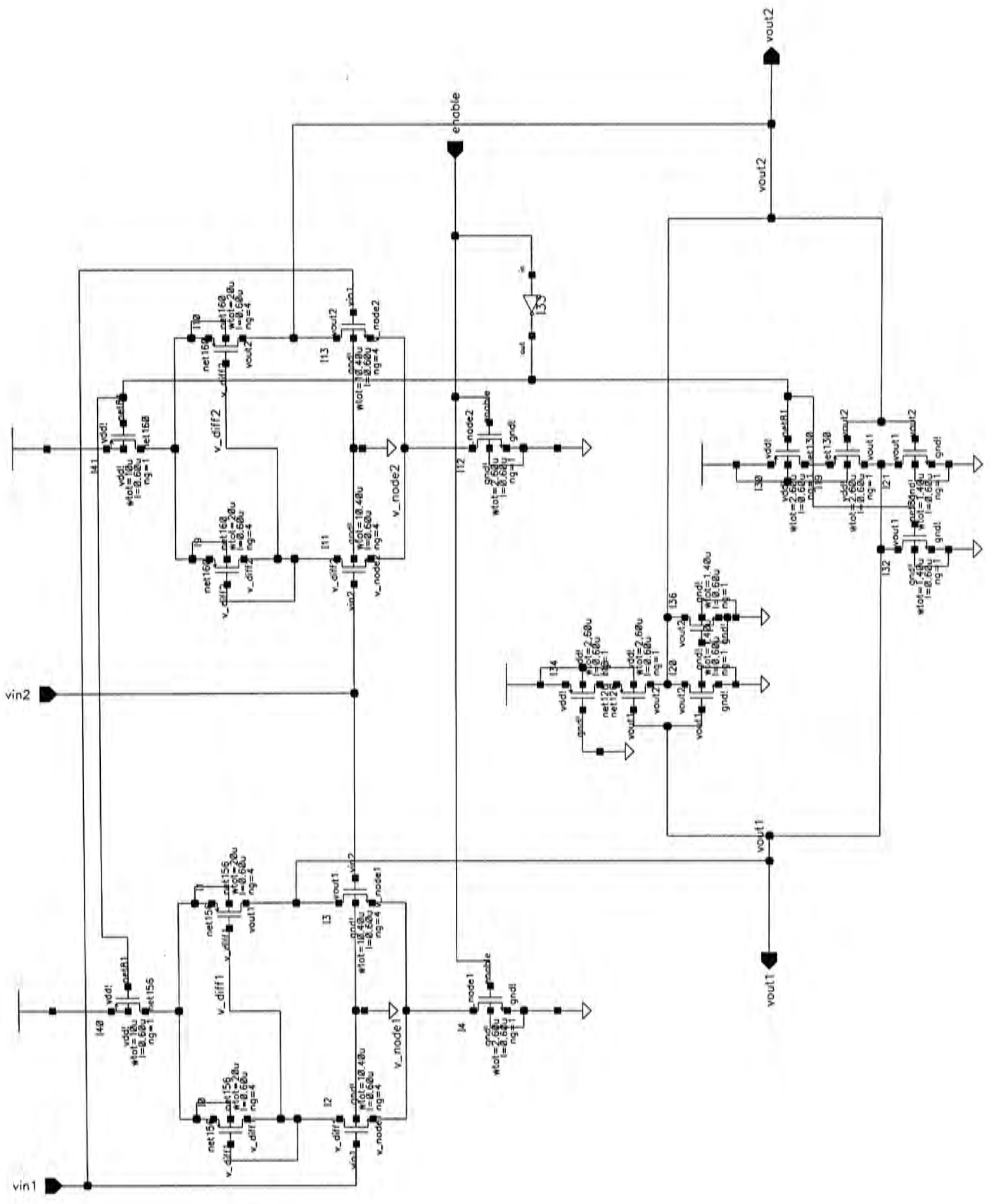
Schematic of Refresh Control Circuit



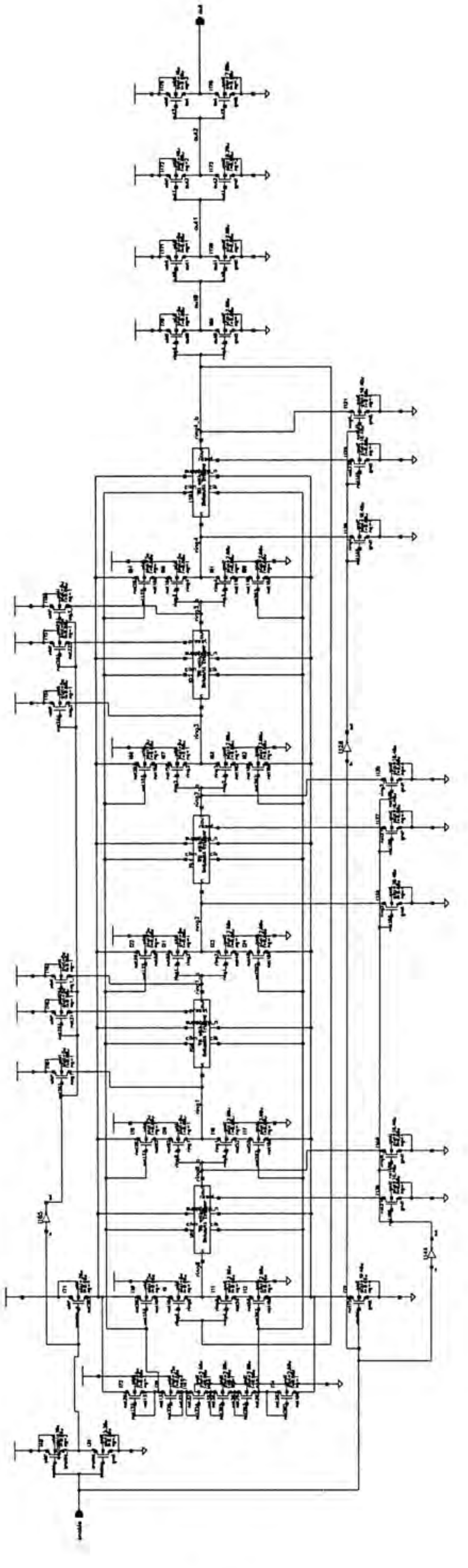
Schematic of differential amplifier



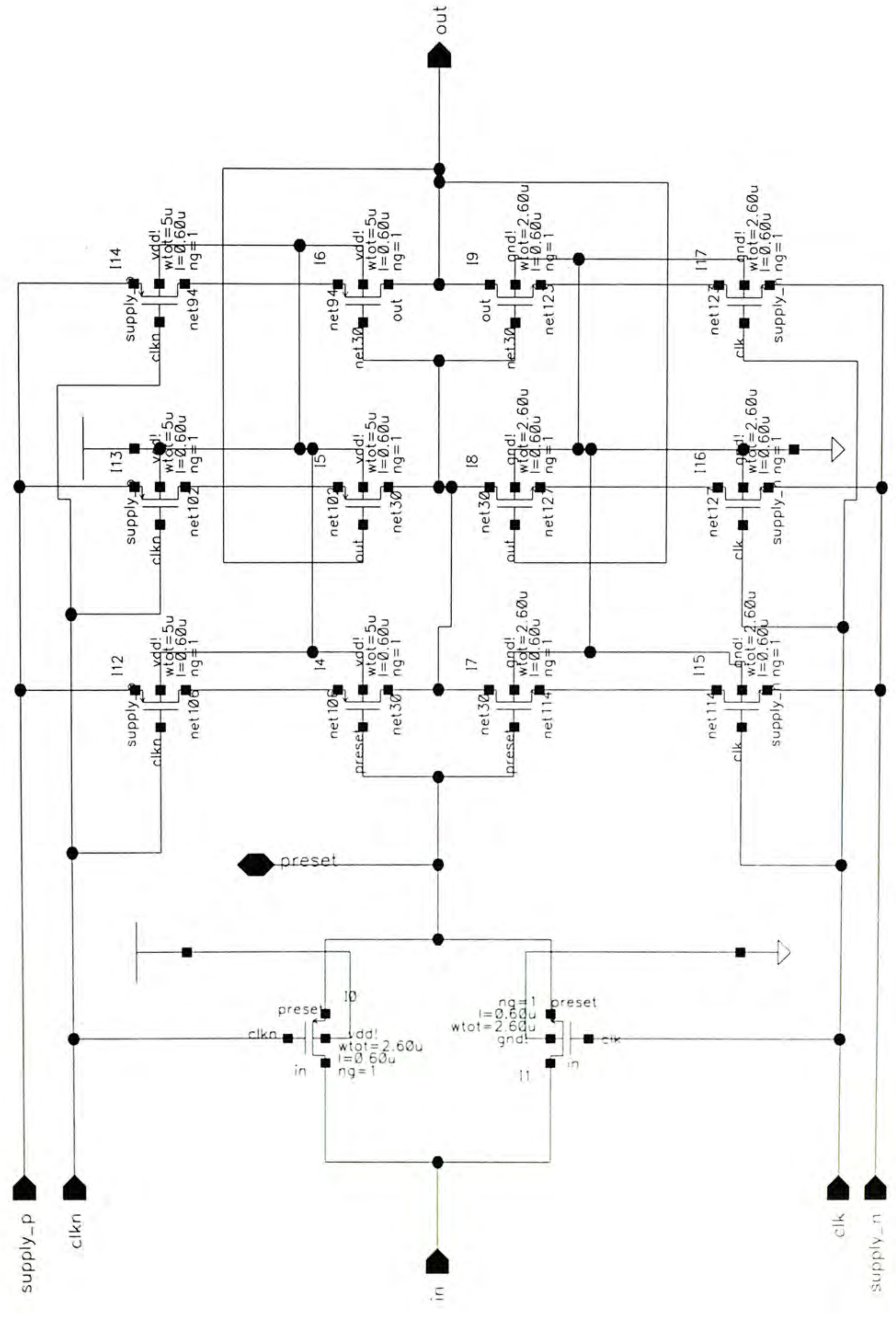
Schematic of sense amplifier



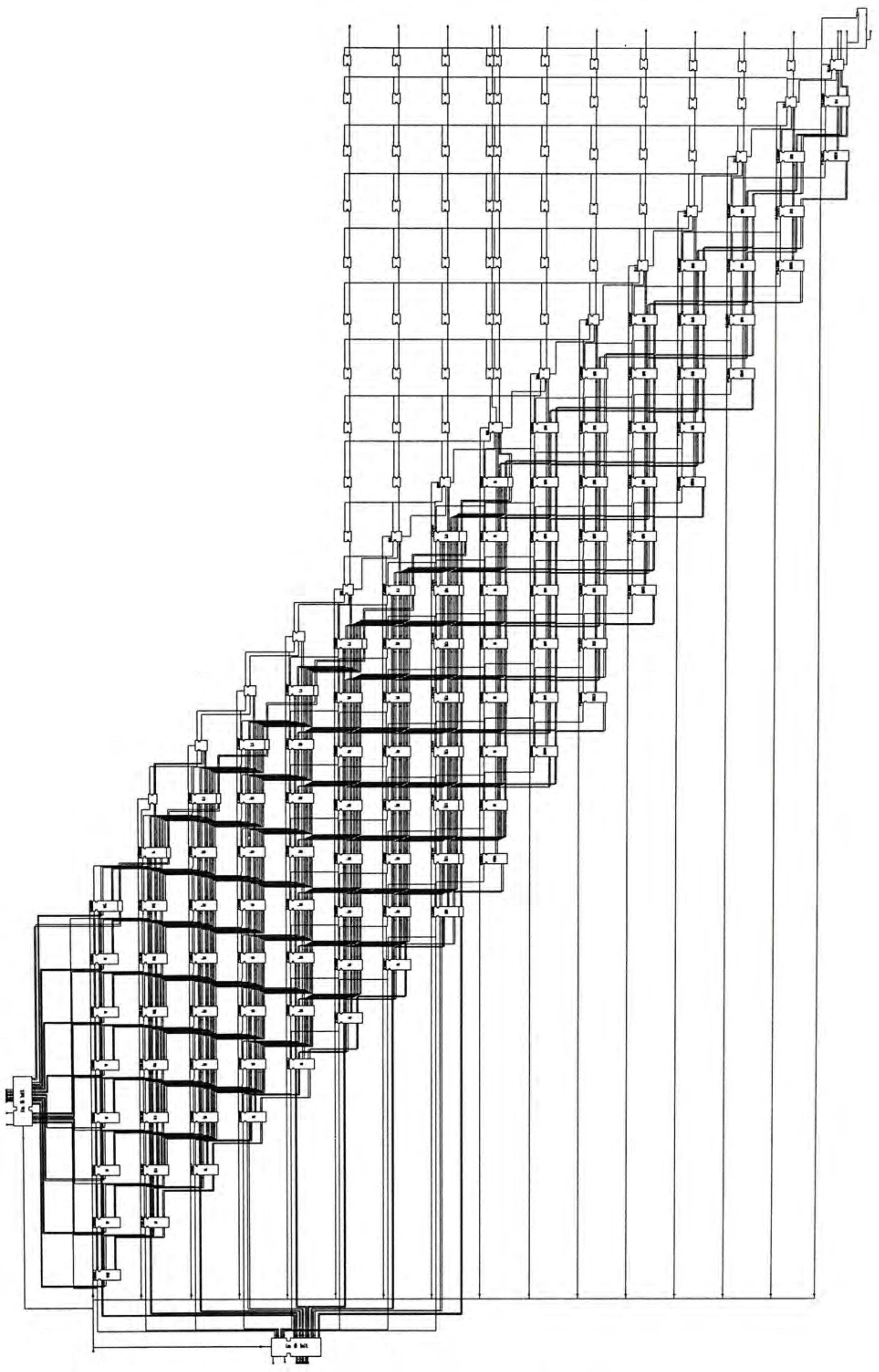
Schematic of ring oscillator



Schematic of voltage-controlled transmission gate with Schmitt trigger

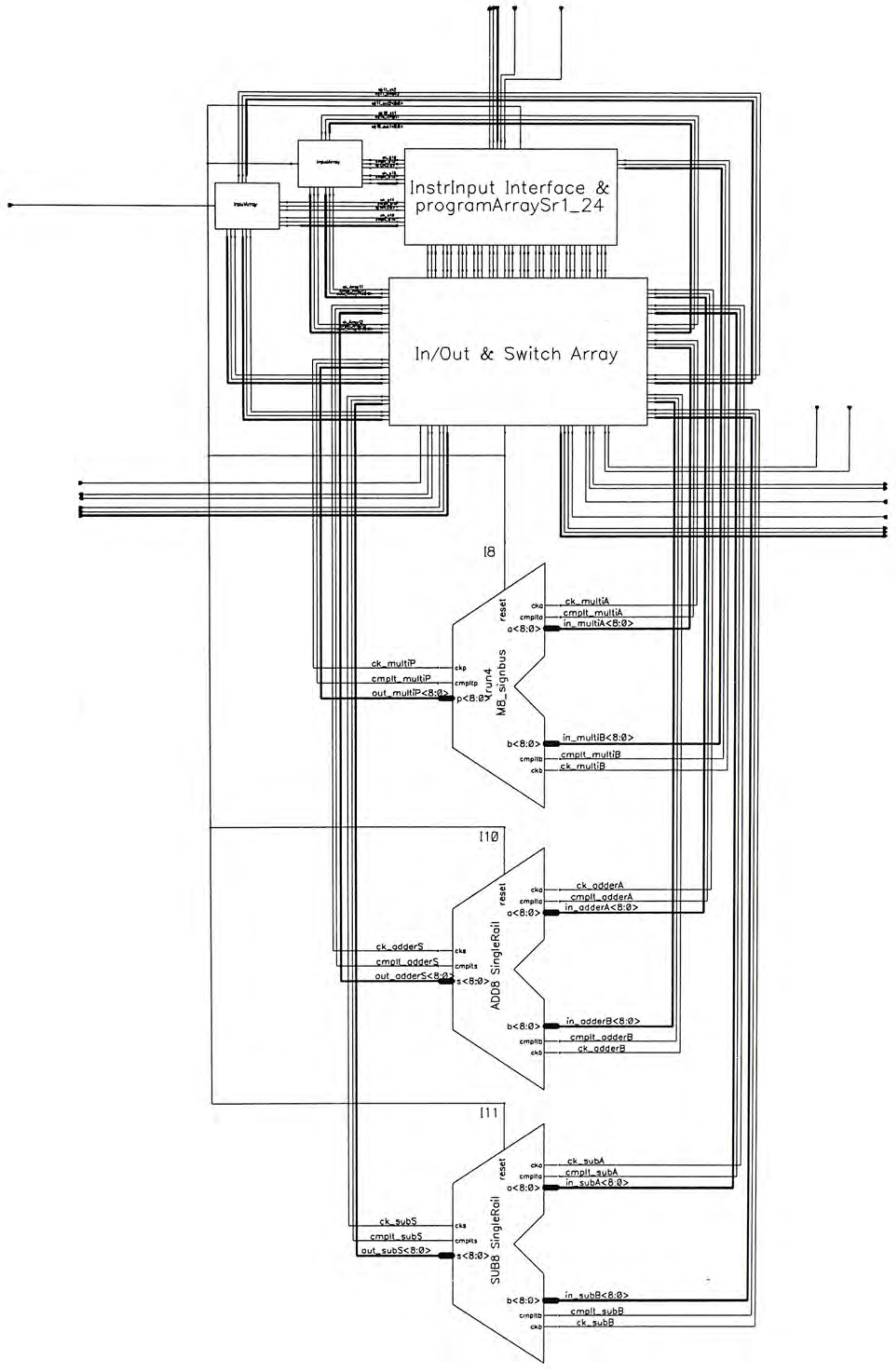


Schematic of asynchronous bit-parallel multiplier

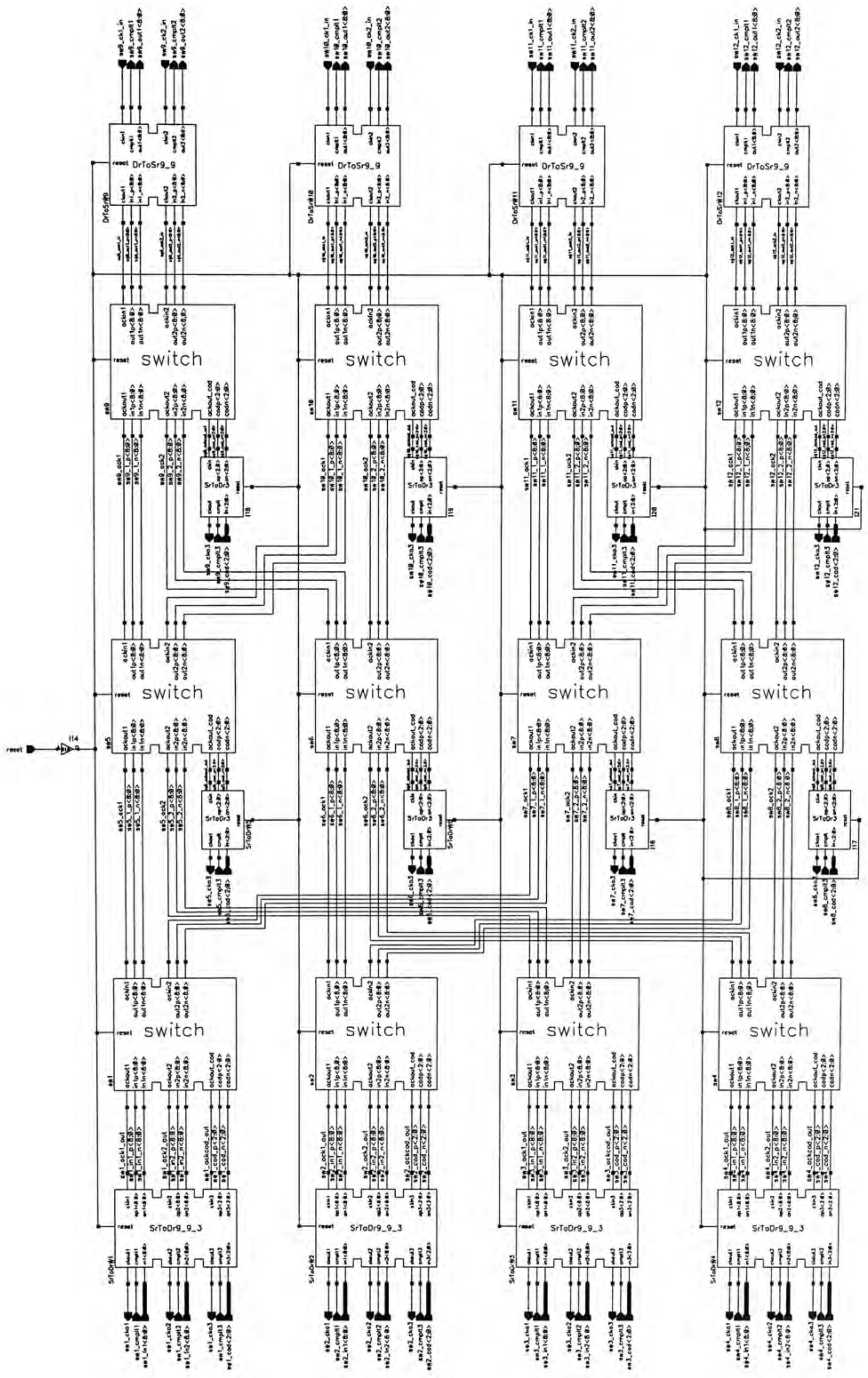


Schematics of Programmable DSP Processor

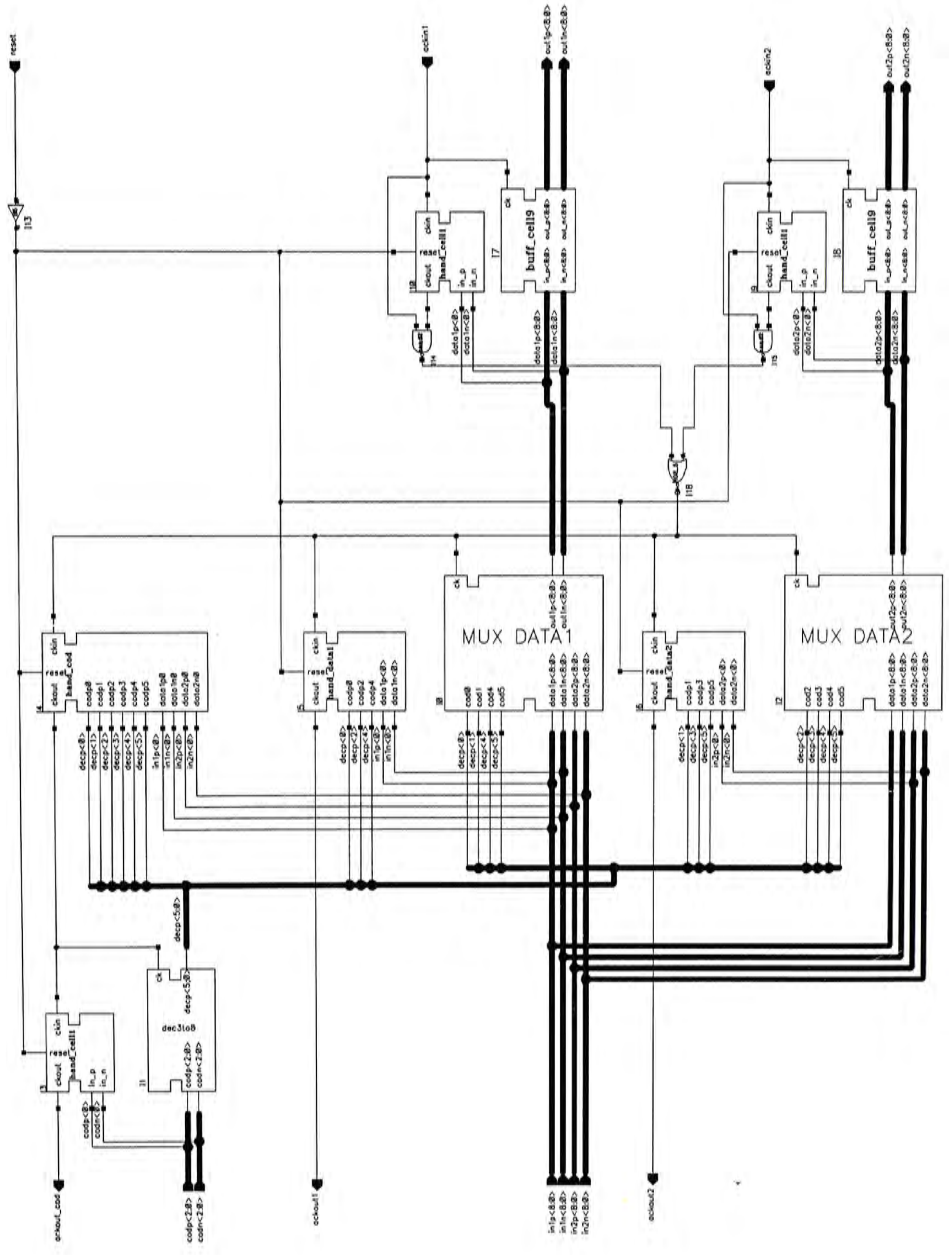
Schematic of programmable DSP processor



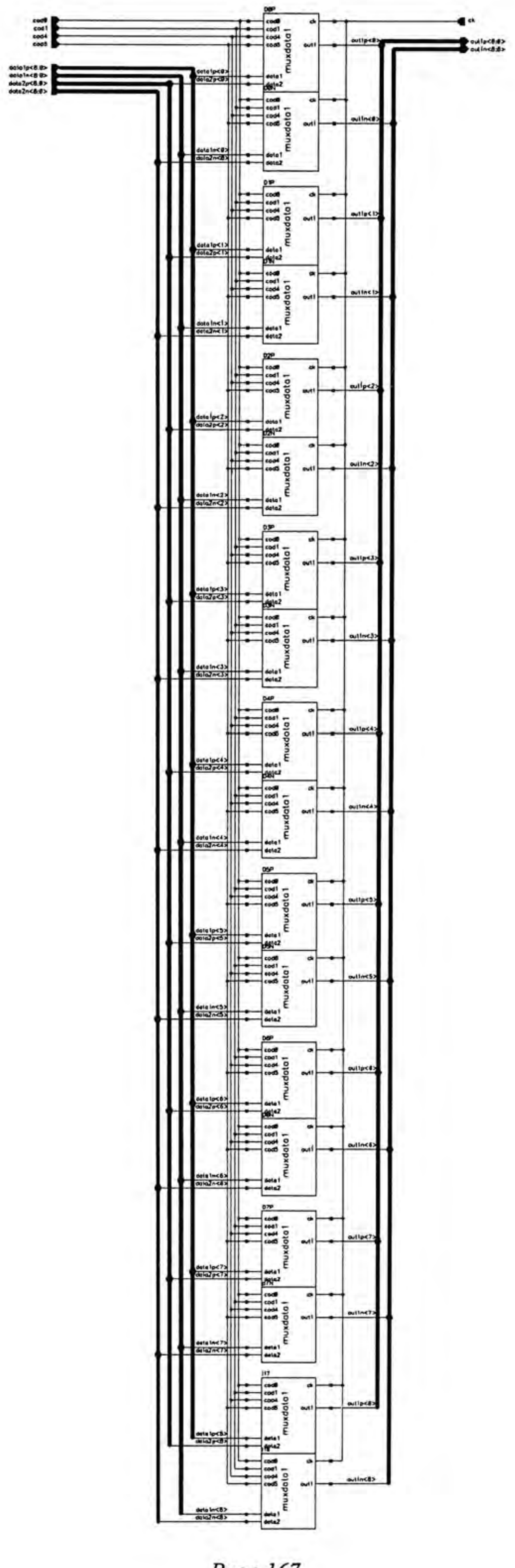
Schematic of switch network



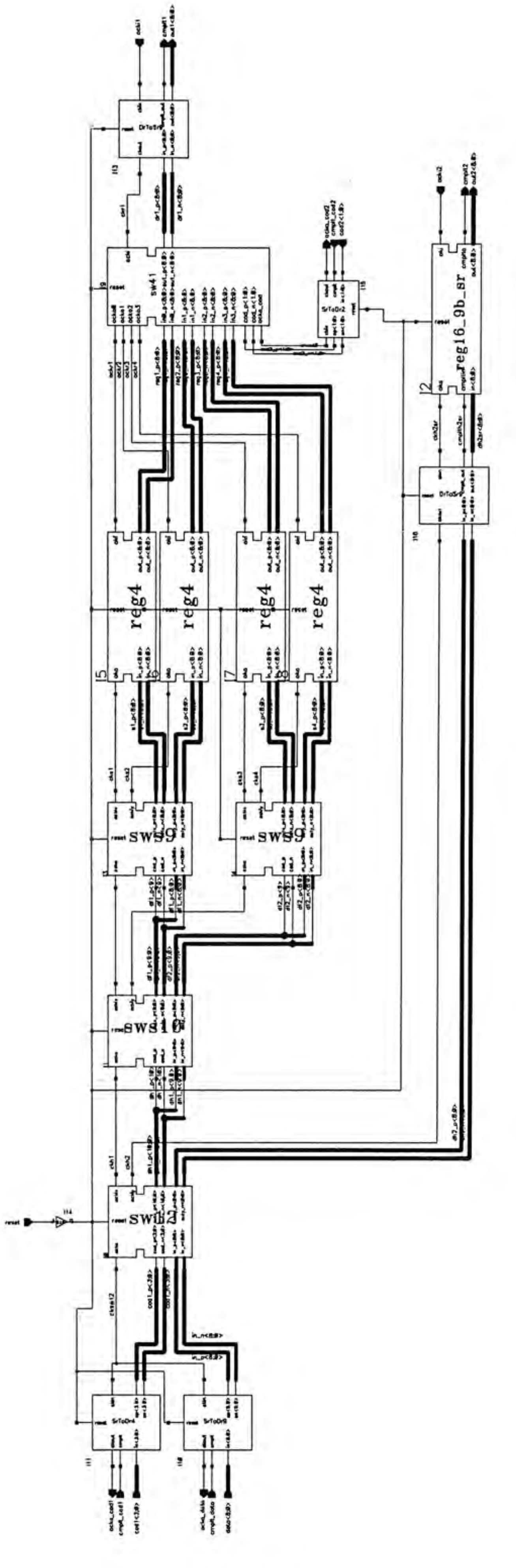
Schematic of switch cell



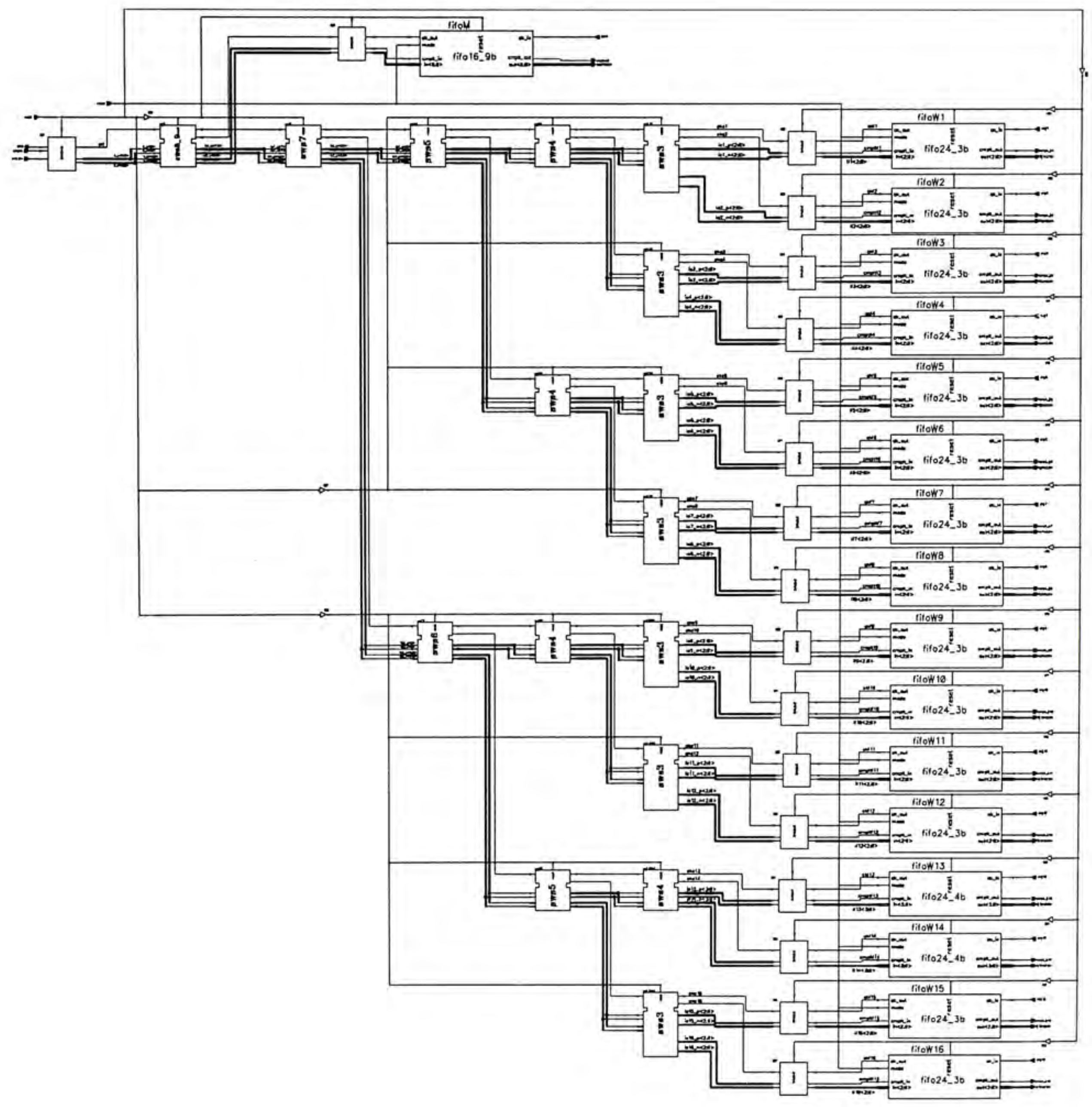
Schematic of multiplexer cell used in switch cell



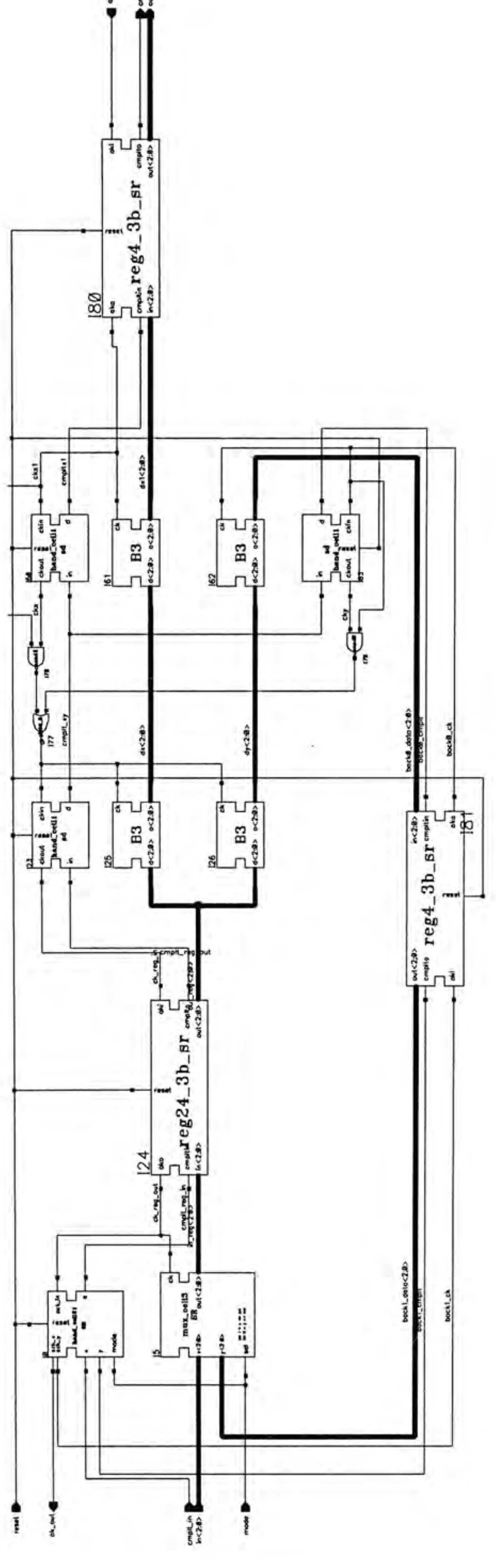
Schematic FIFO memory



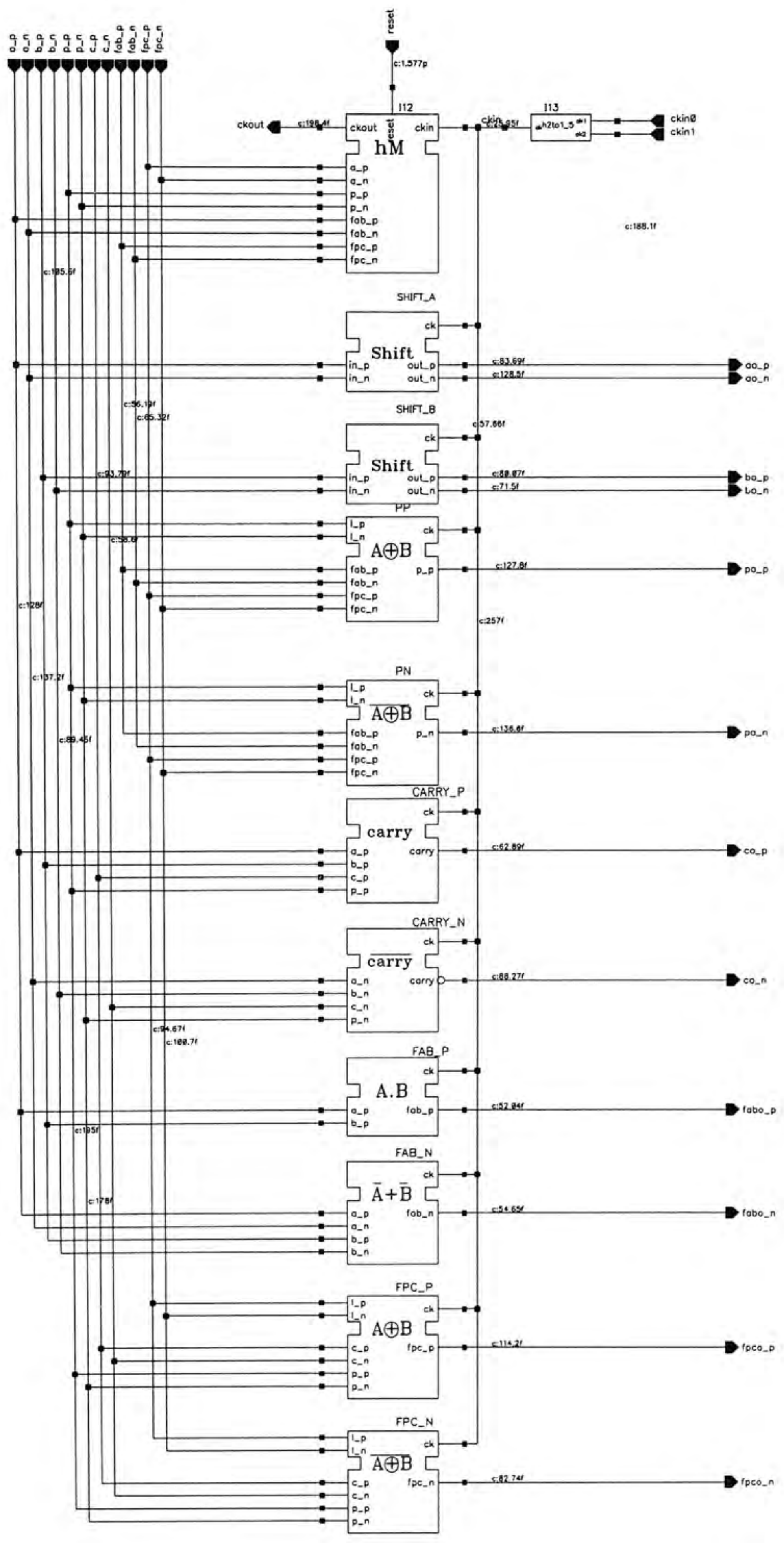
Schematic of Instruction Memory (instruction decoding network and cyclic FIFOs)



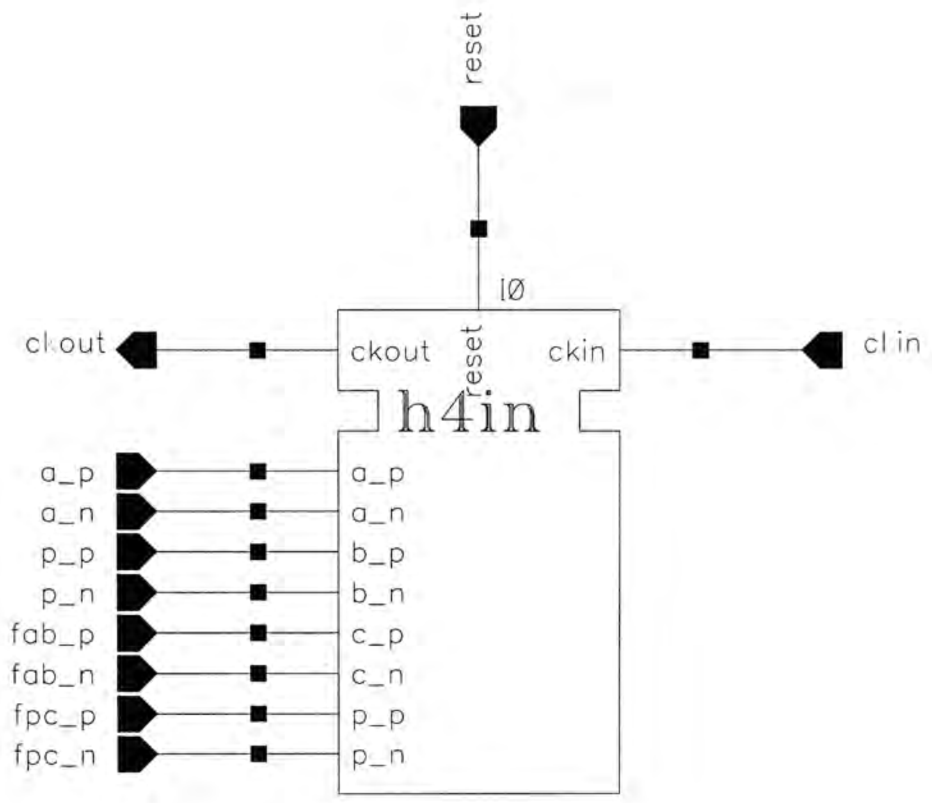
Schematic of cyclic FIFO



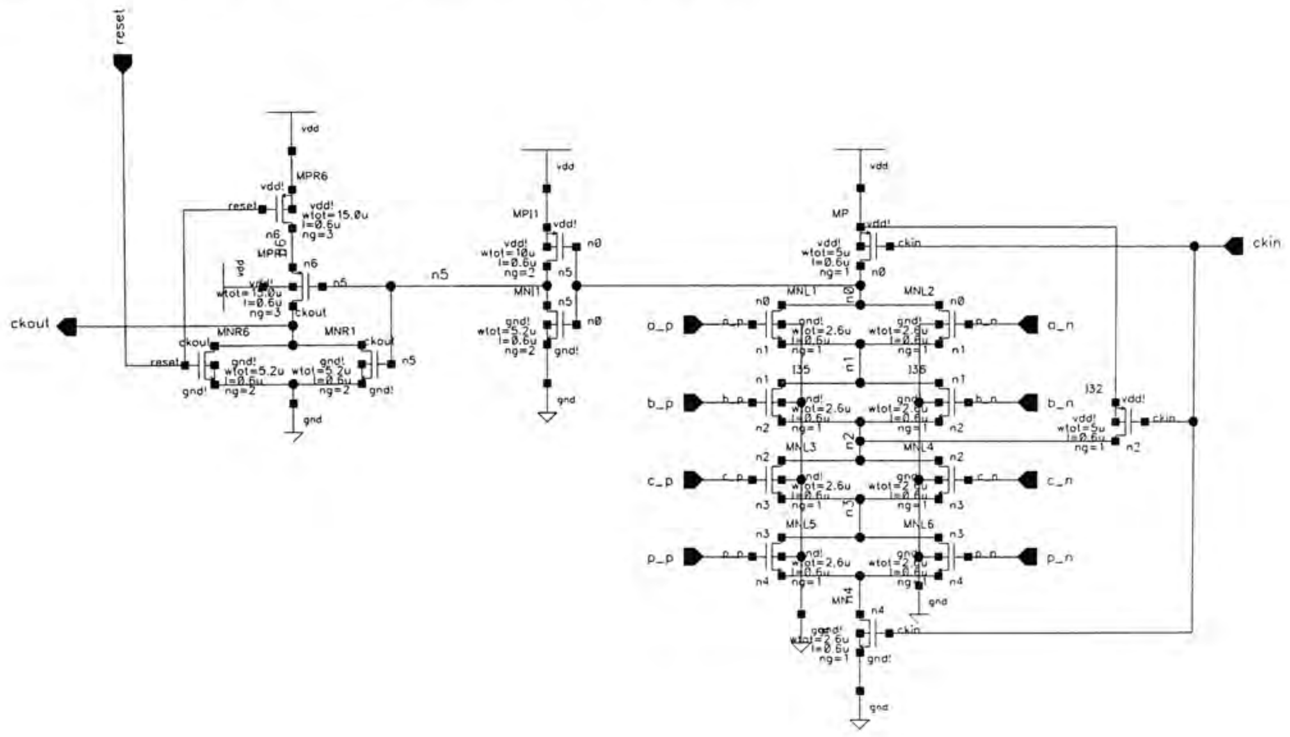
Schematic of Product Full Adder (PFA) in multiplier core



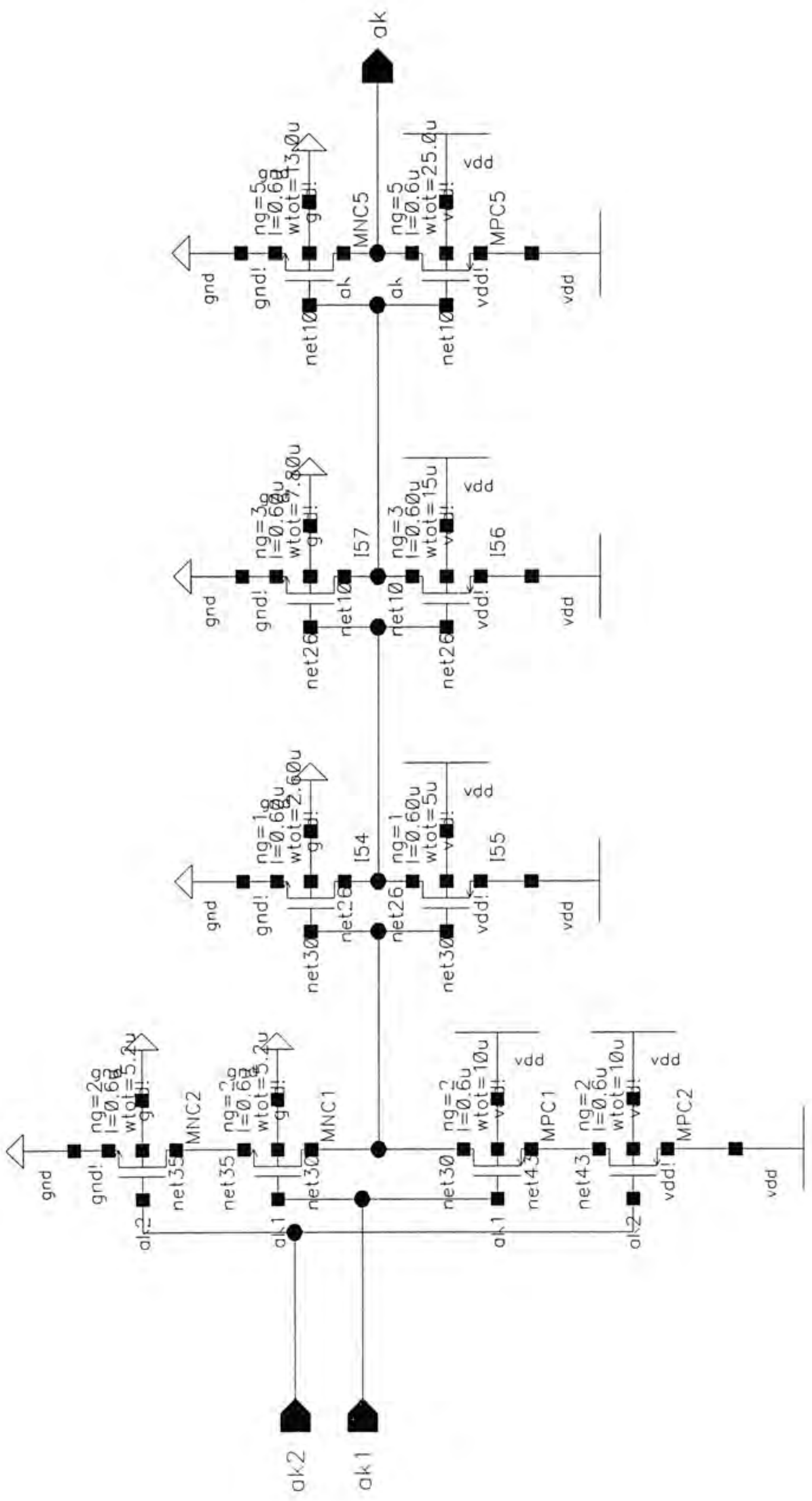
Schematic of handshake cell hM (calling h4in) used in FPA



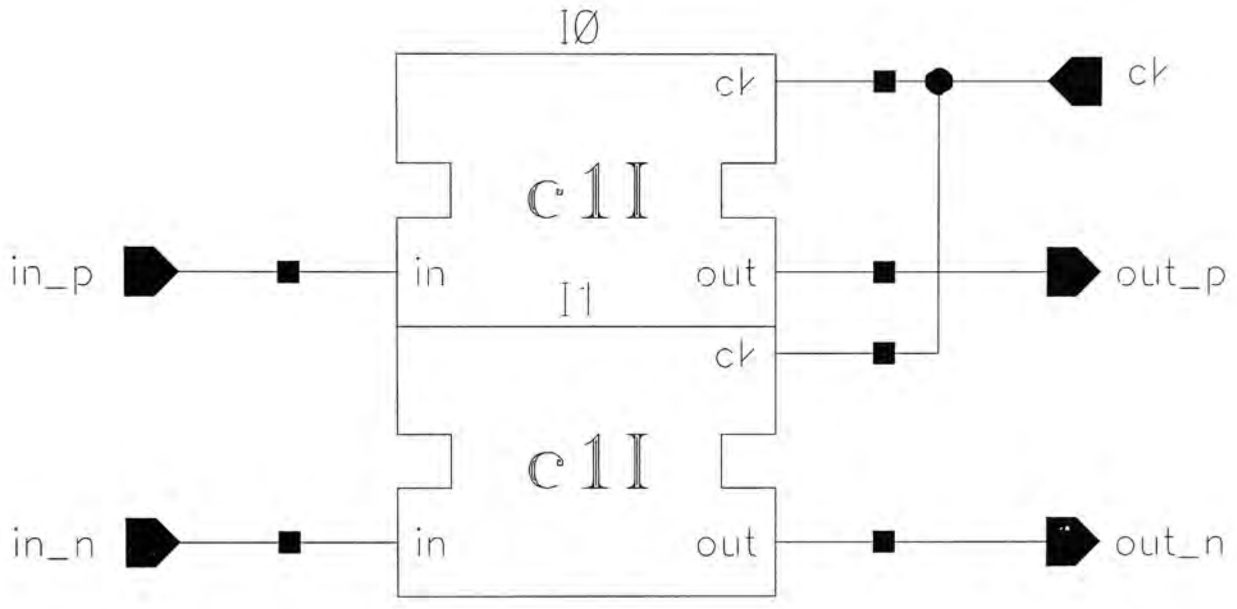
Schematic of the handshake cell h4in(called by hM)



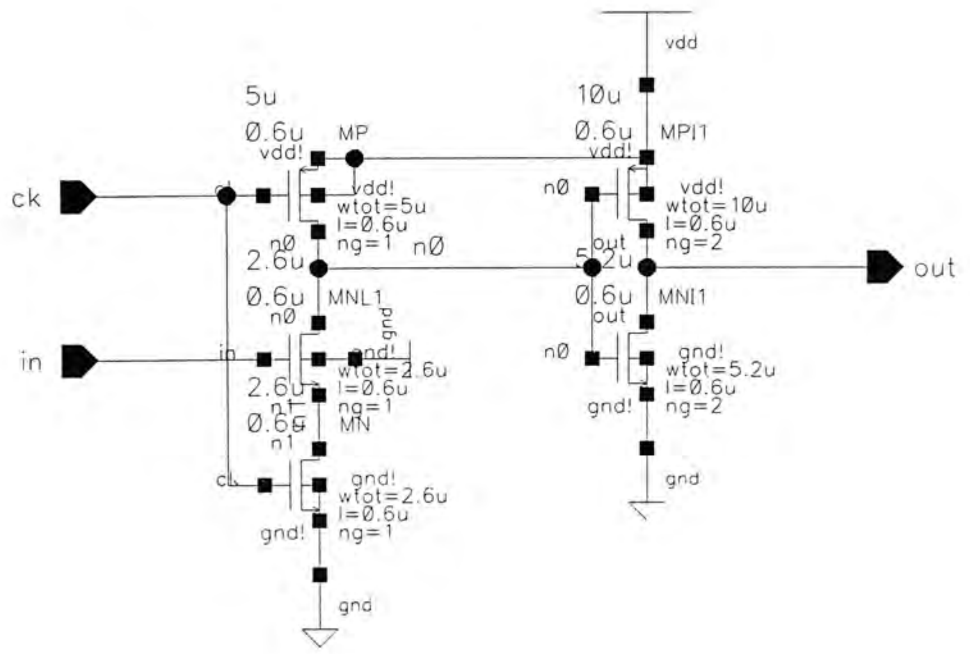
Schematic of the handshake signal merger h2to1_5



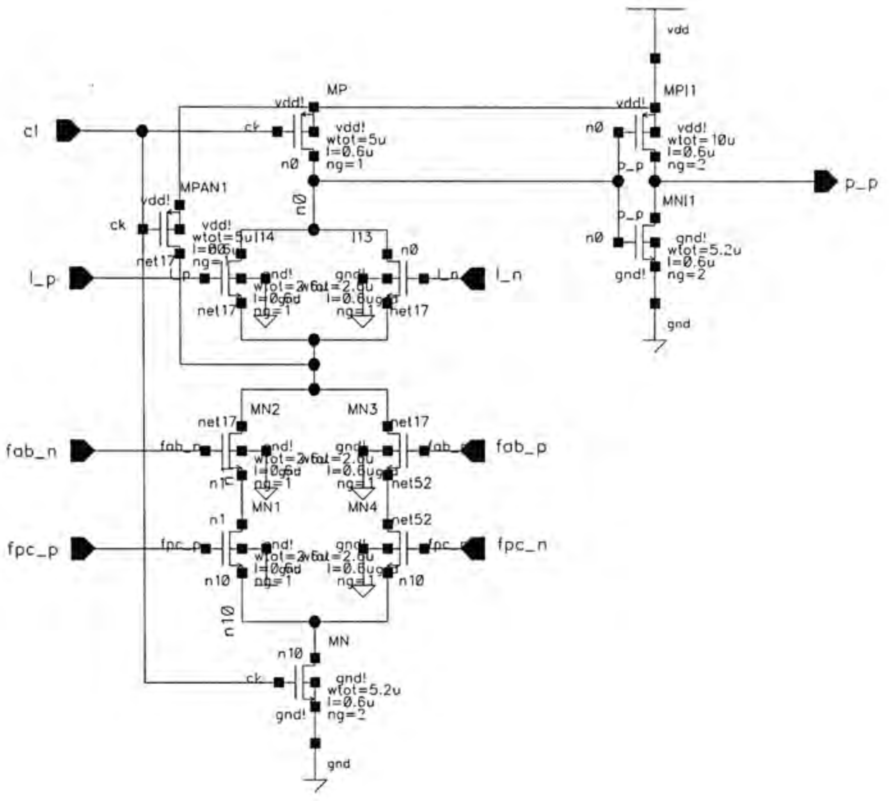
Schematic of the shift cell (calling c1I)



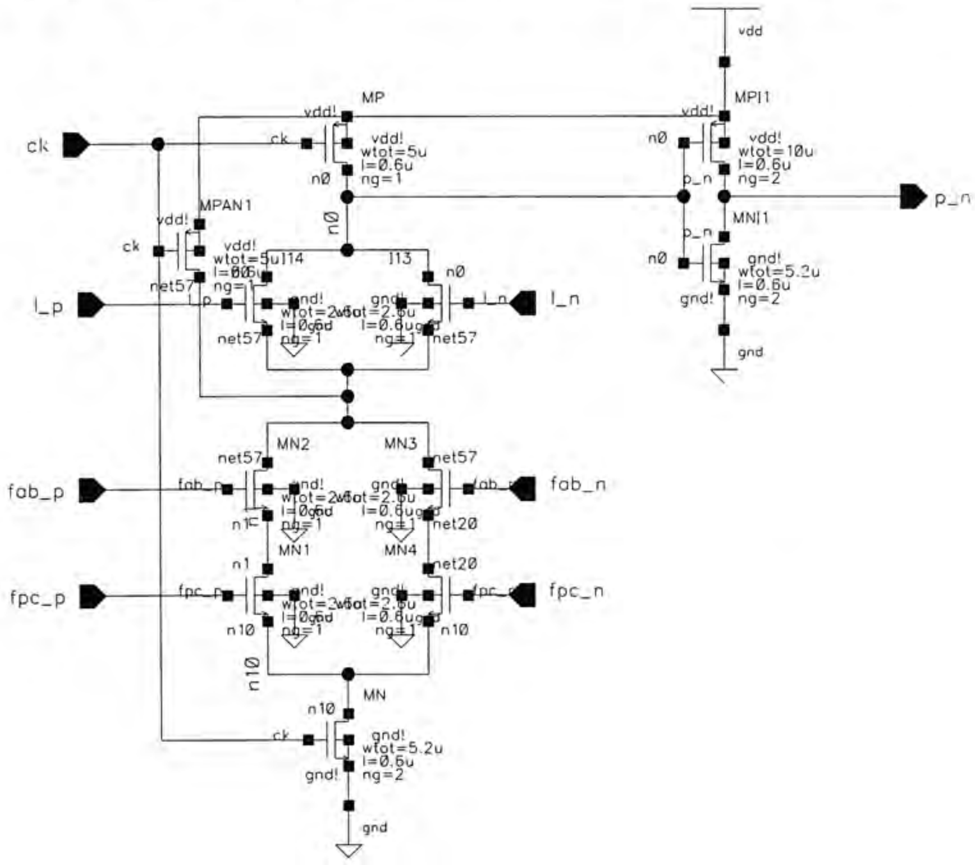
Schematic of the buffer cell C1I (signal rail)



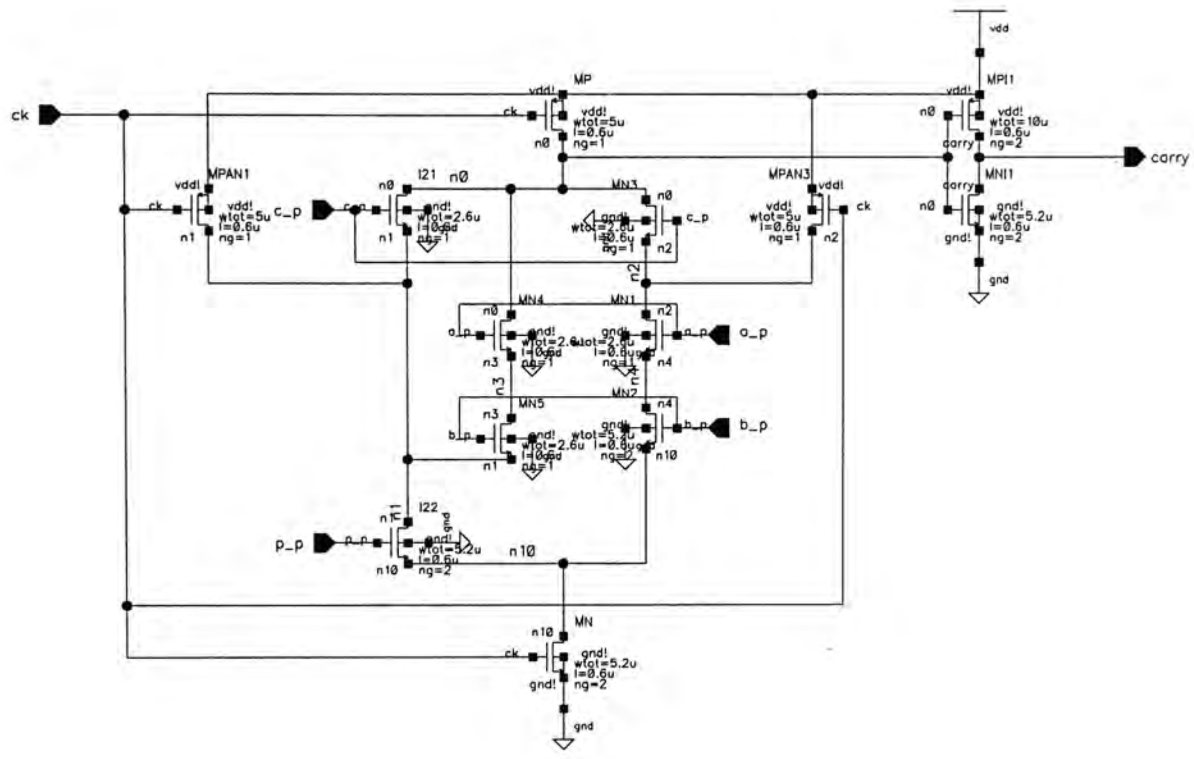
Schematic of 2nd stage Product (true value) generation cell PP



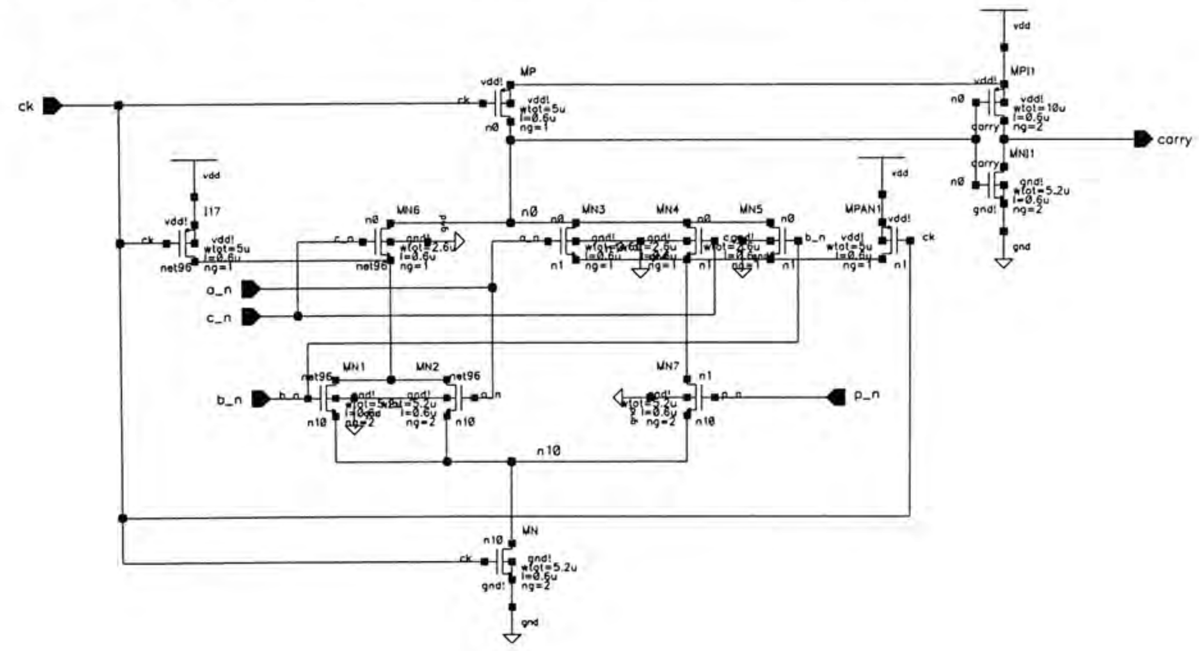
Schematic of 2nd stage Product (complement value) generation cell PN



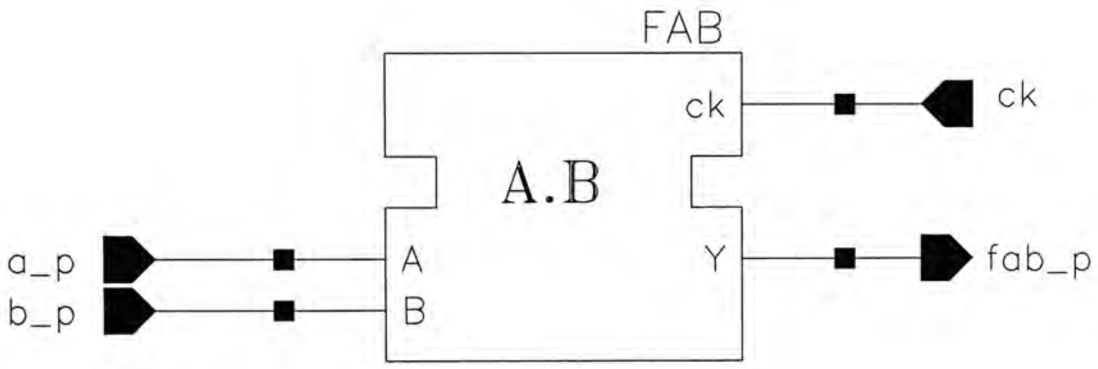
Schematic of Carry (true value) generation cell CARRY_P



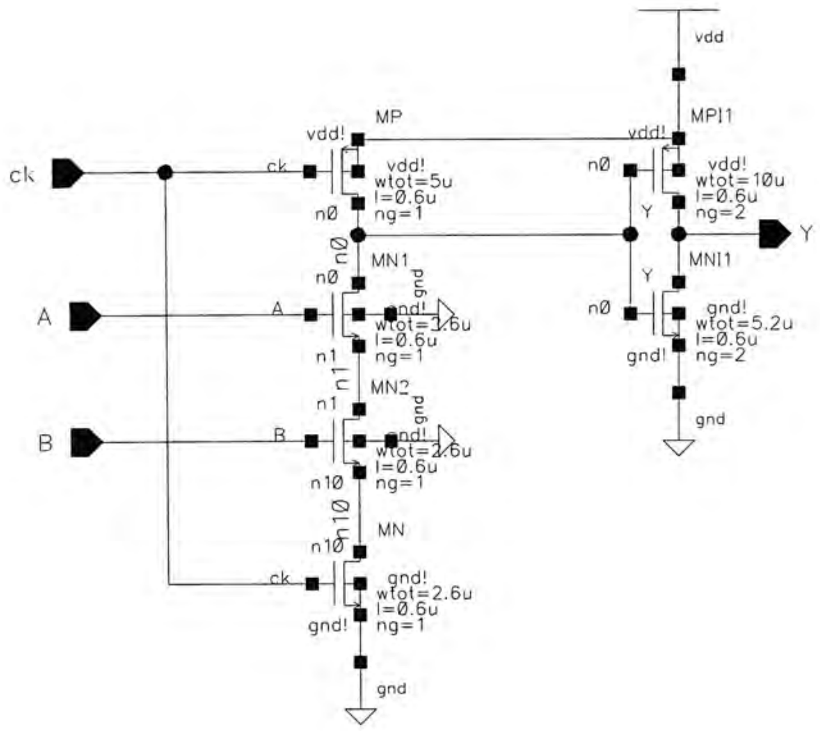
Schematic of Carry (complement value) generation cell CARRY_N



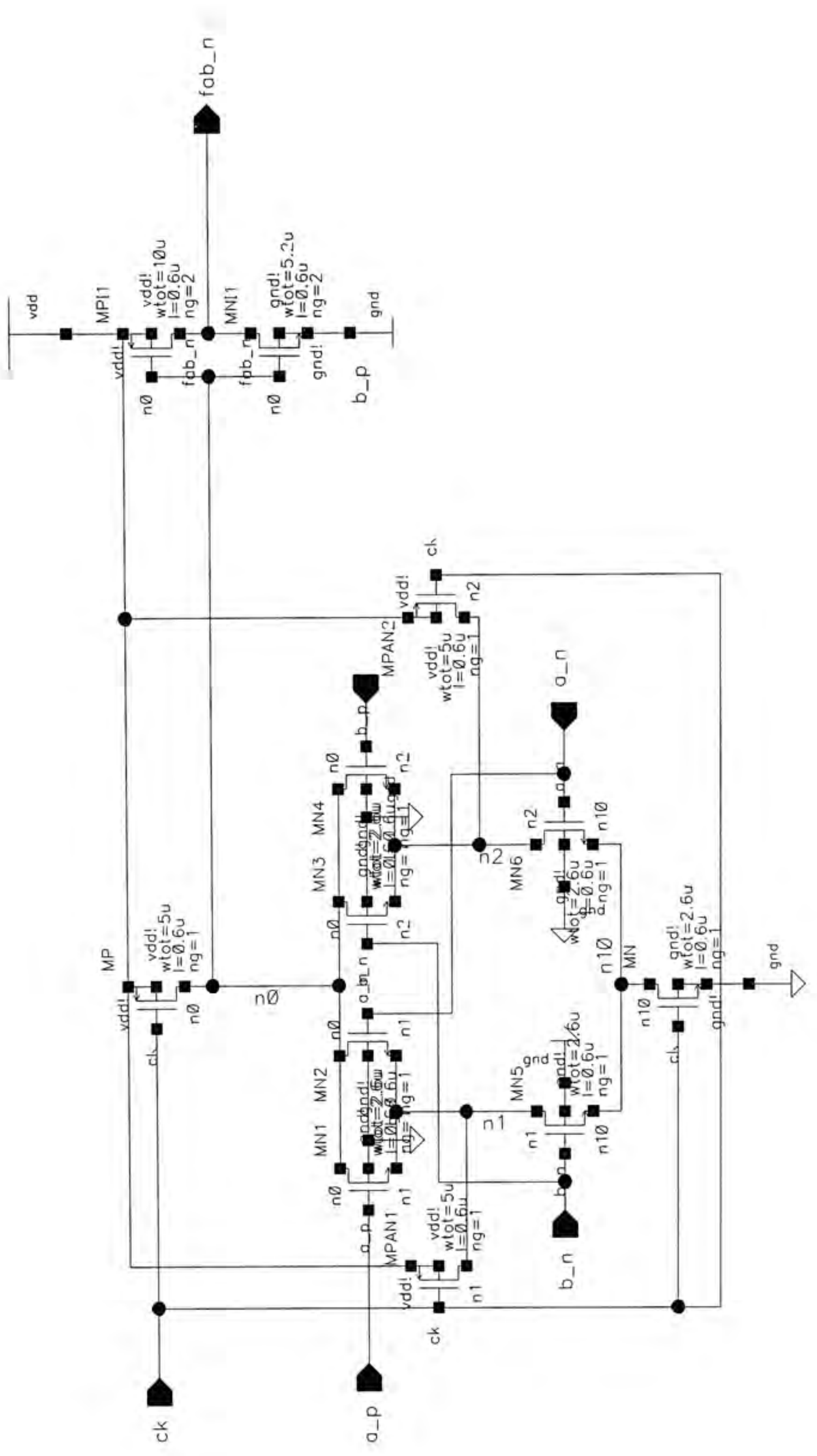
Schematic of 1st stage Product (true value) generation cell FAB_P (A AND B)



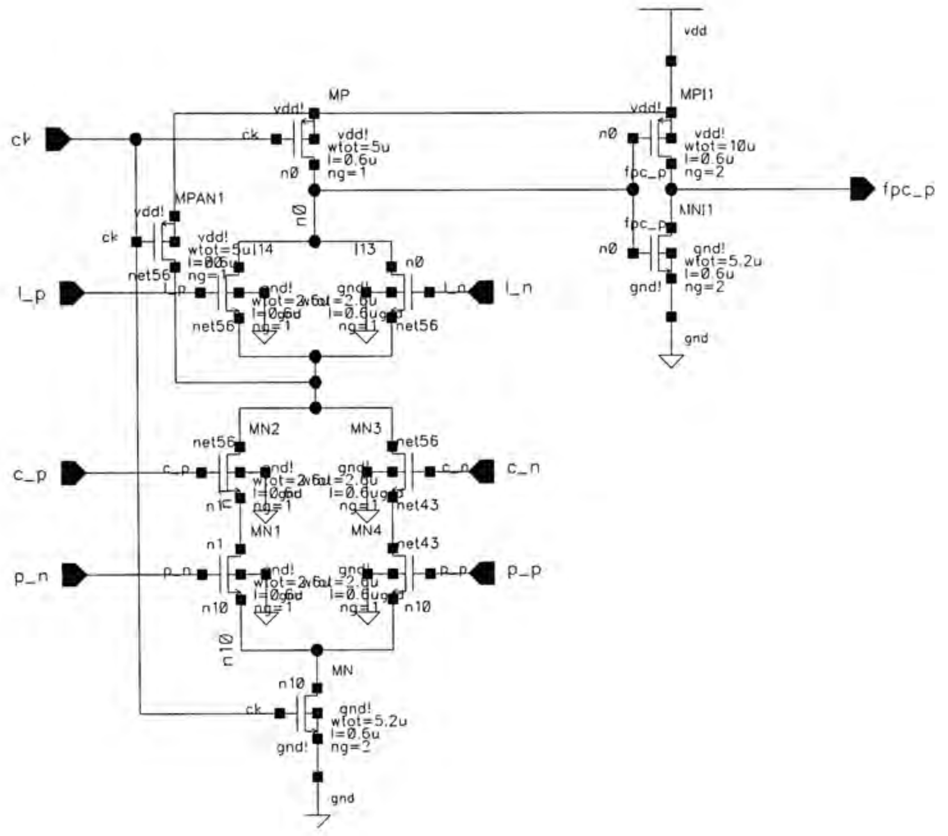
Schematic of 2-input AND gate



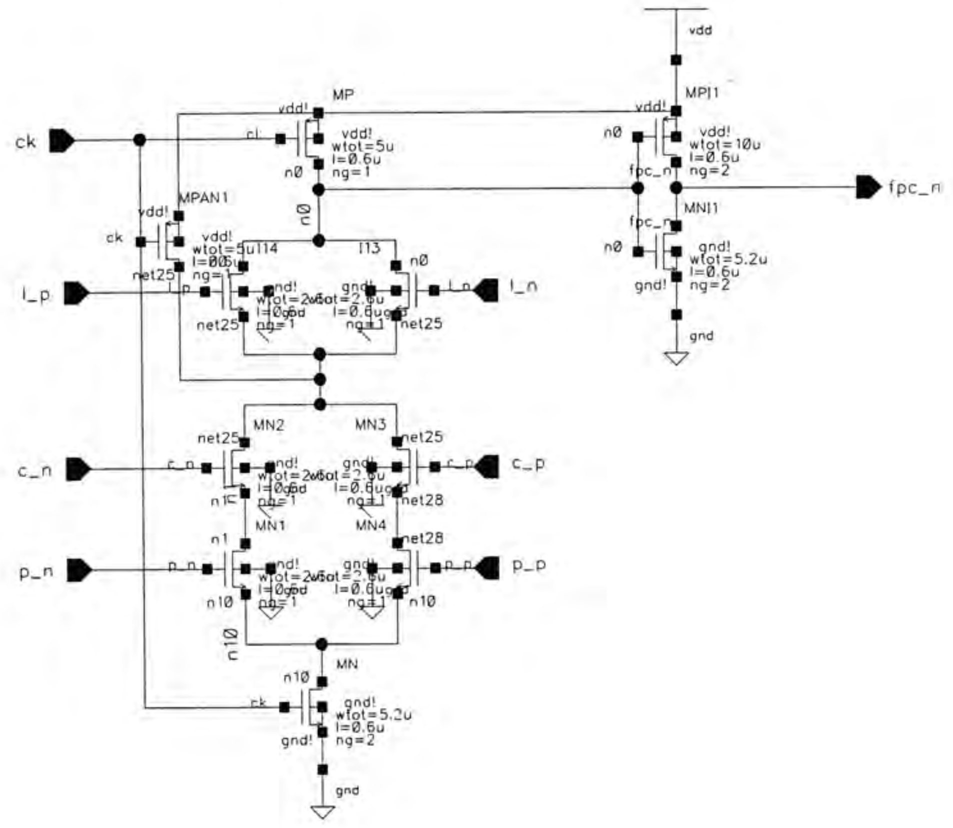
Schematic of 1st stage Product generation (complement value) cell FAB_N (A AND B)



Schematic of 1st stage Product generation (true value) cell FPC_P (P XOR C)

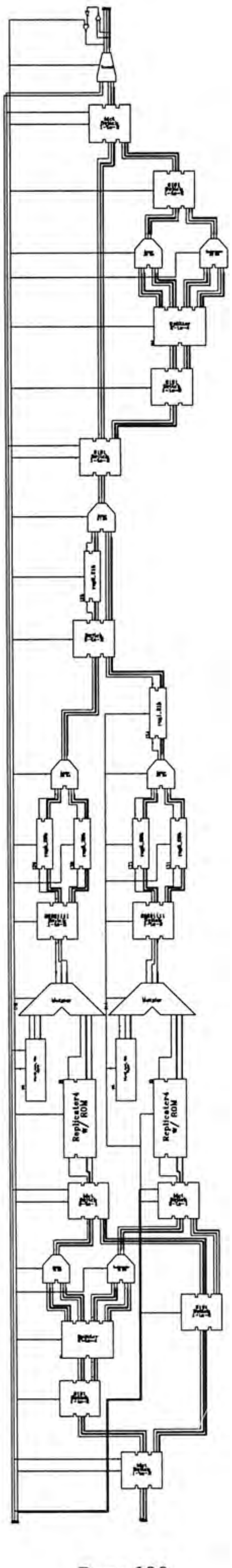


Schematic of 1st stage Product generation (complement value) cell FPC_P (P XNOR C)

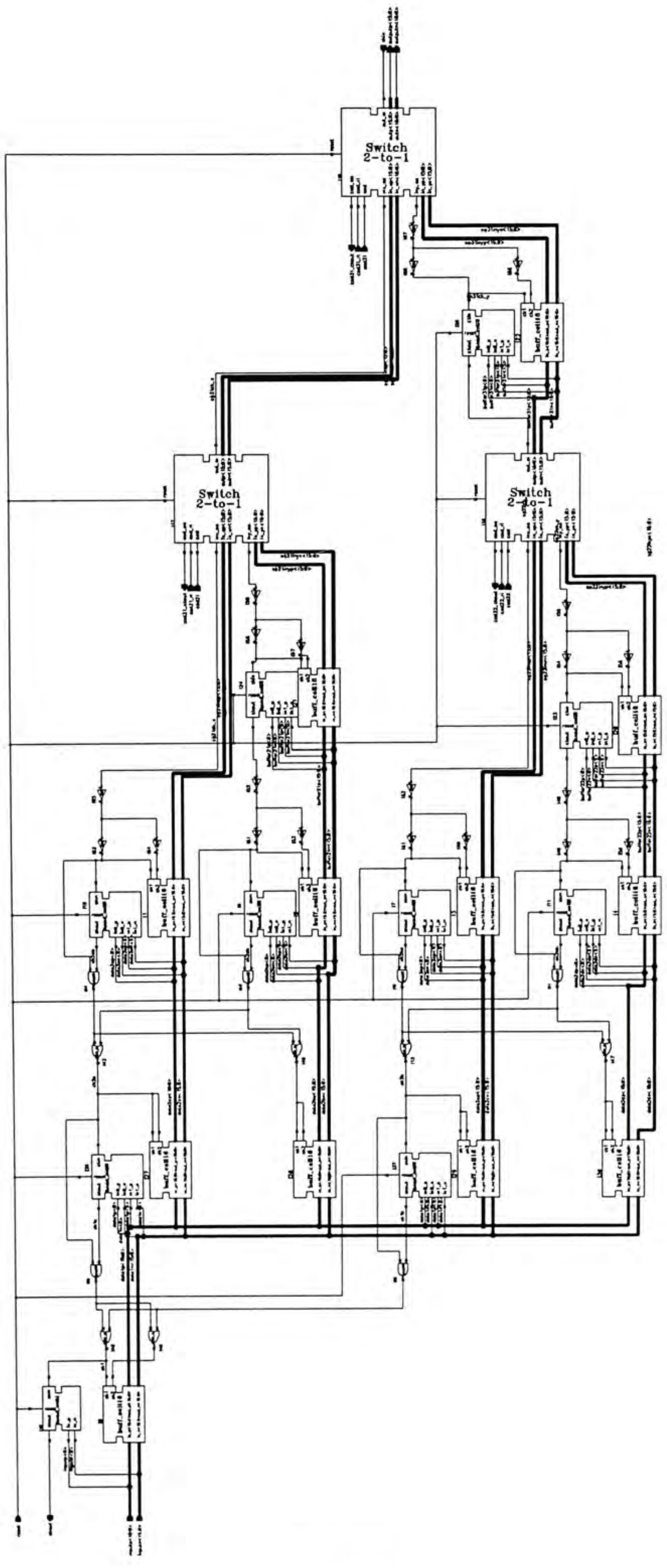


Schematics of 1D DCT/IDCT Core

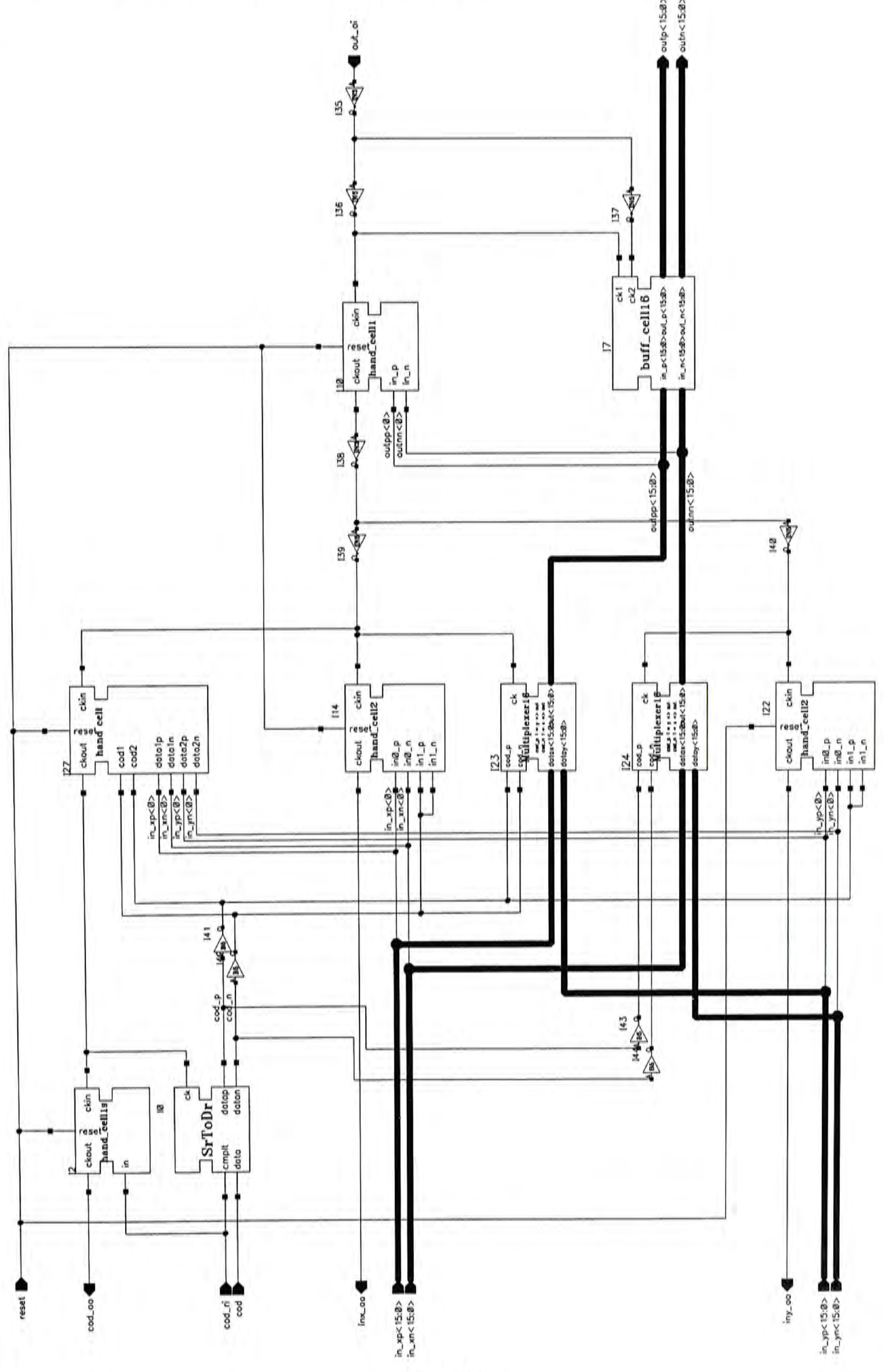
Schematic of 1D DCT/IDCT core



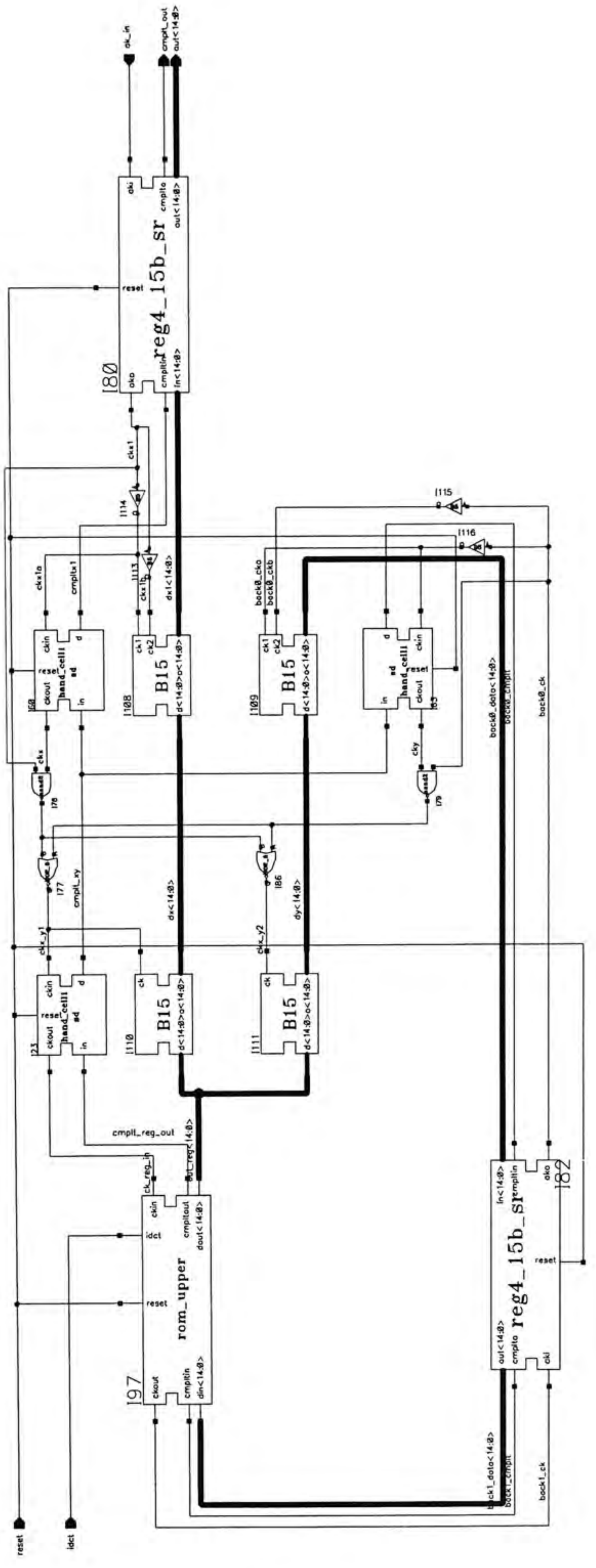
Schematic of data replicator



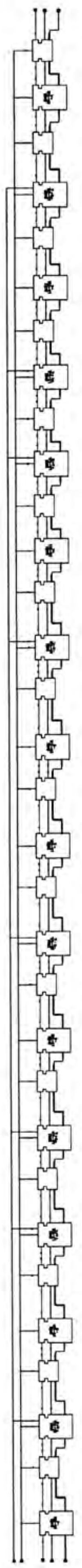
Schematic of multiplexer cell used in data replicator



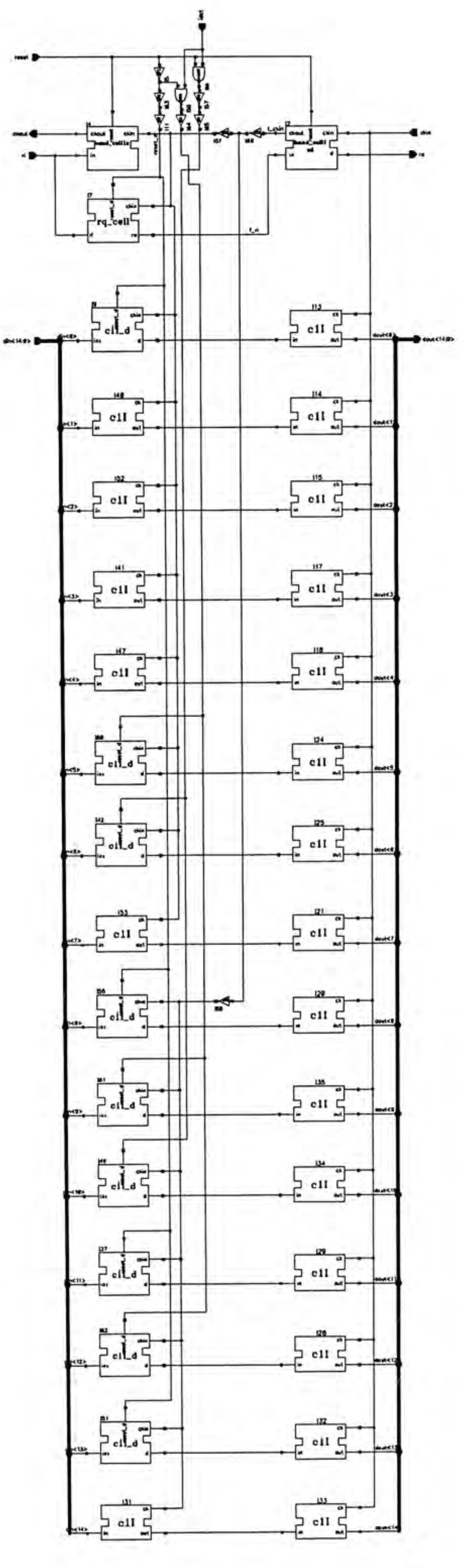
Schematic of DCT coefficients memory



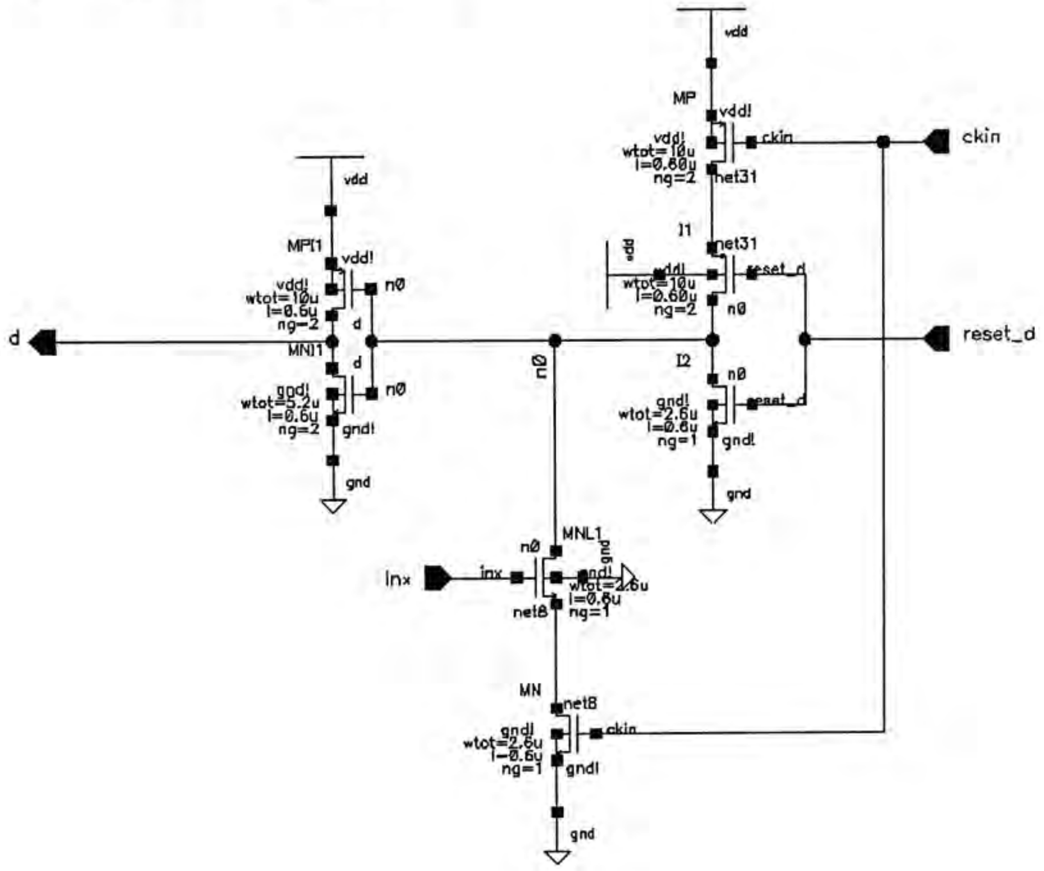
Schematic of memory pipeline stages in DCT coefficient memory



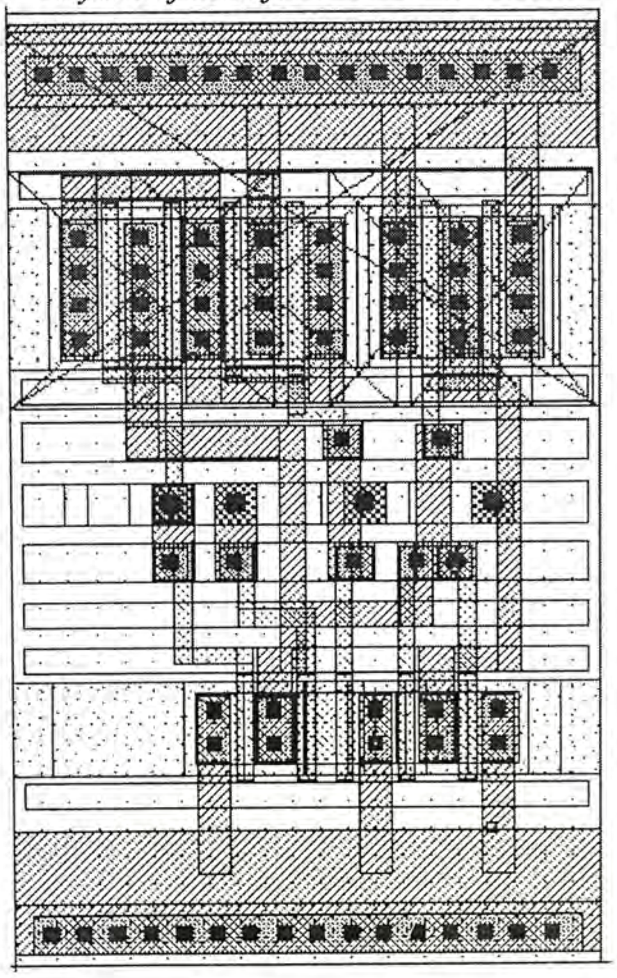
Schematic of single 15bit memory pipeline stage (signal rail)



Schematic of modified basic FIFO cell

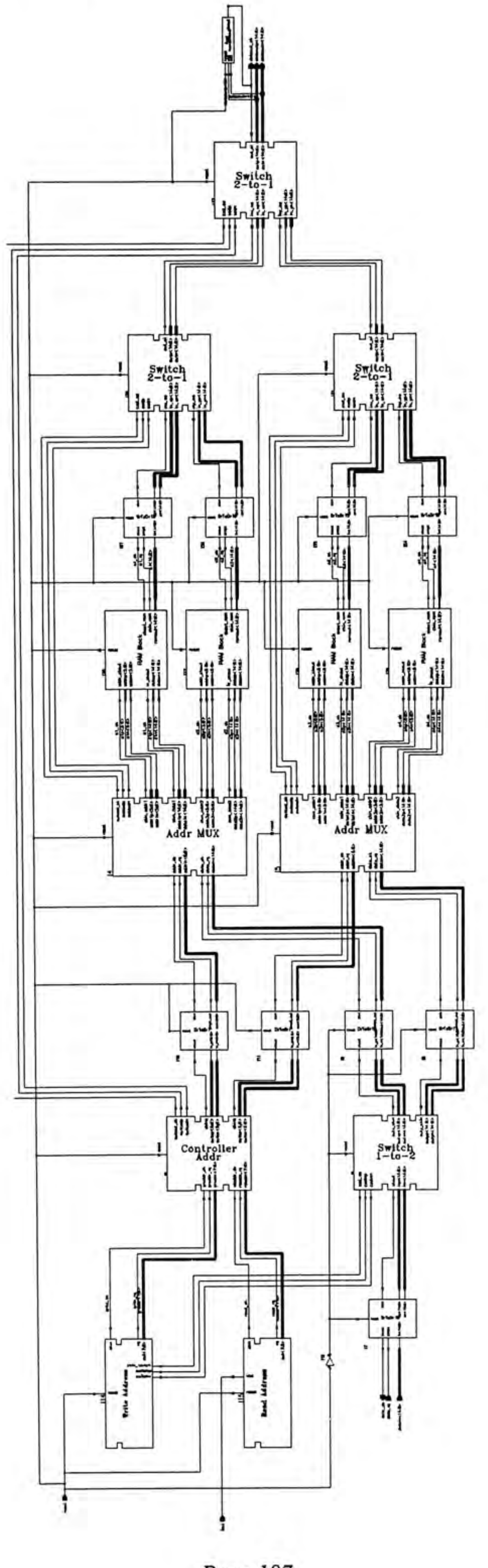


Layout of modified basic FIFO cell

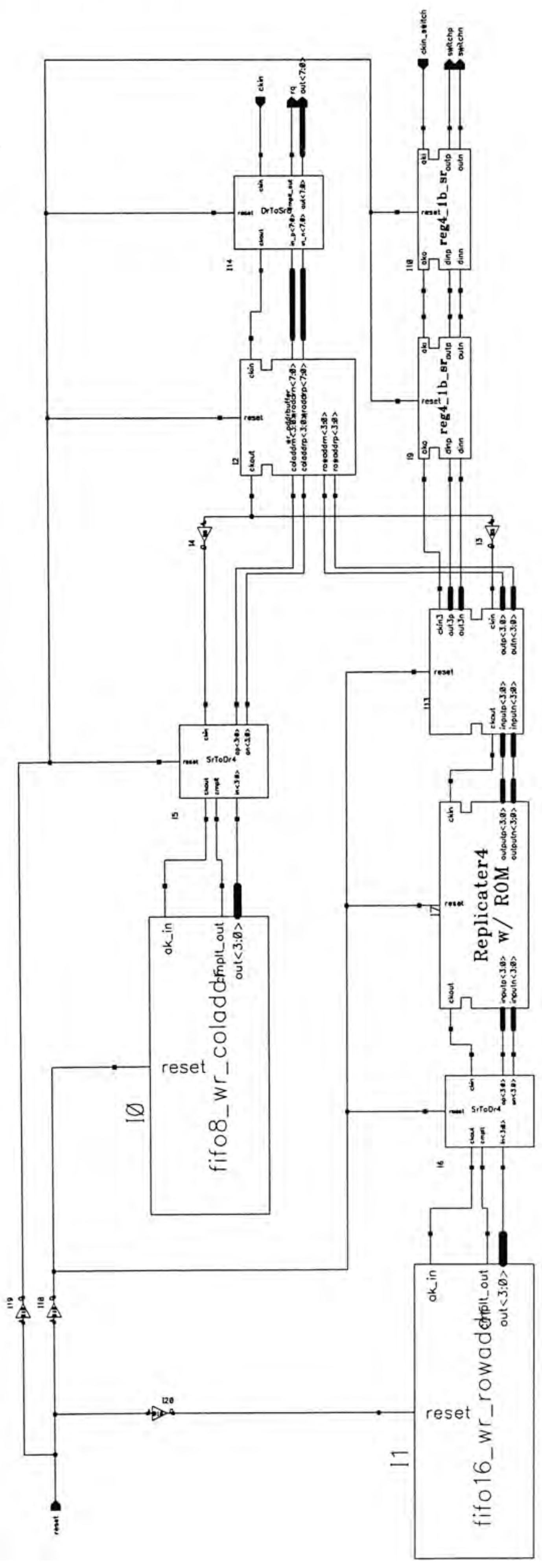


Schematics of Transpose Memory

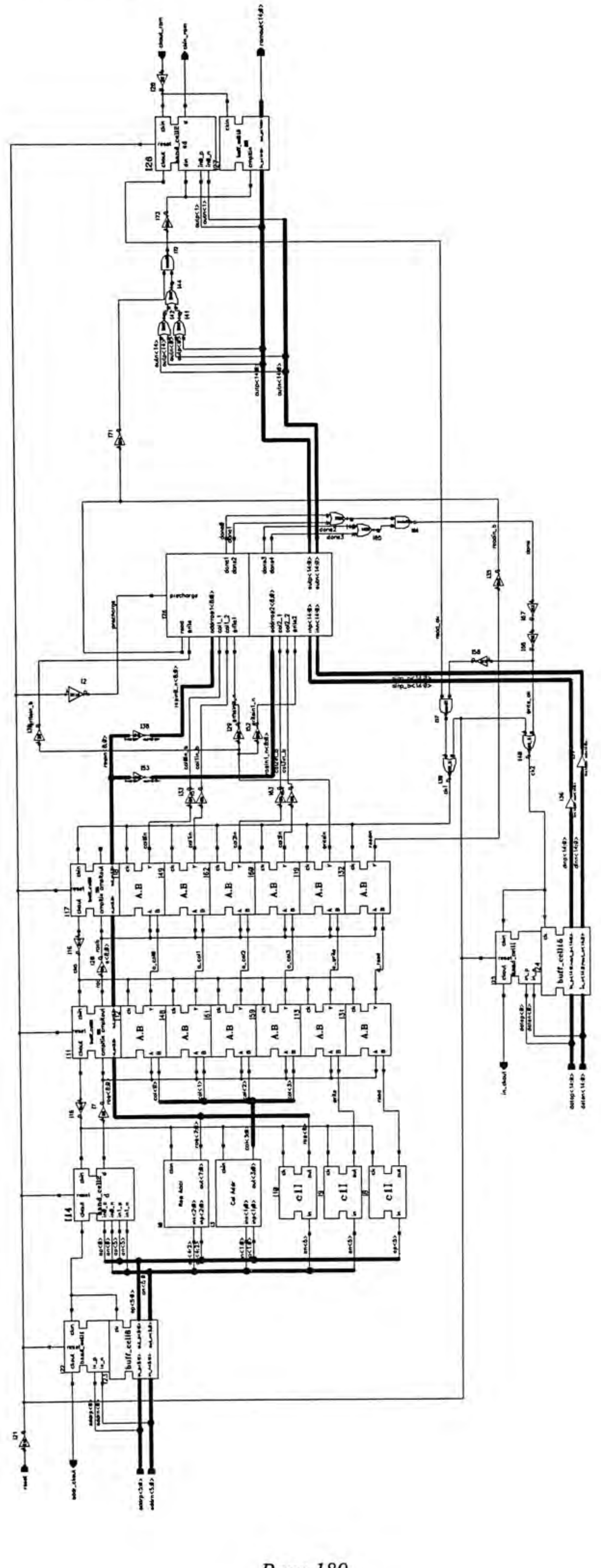
Schematic of transpose memory



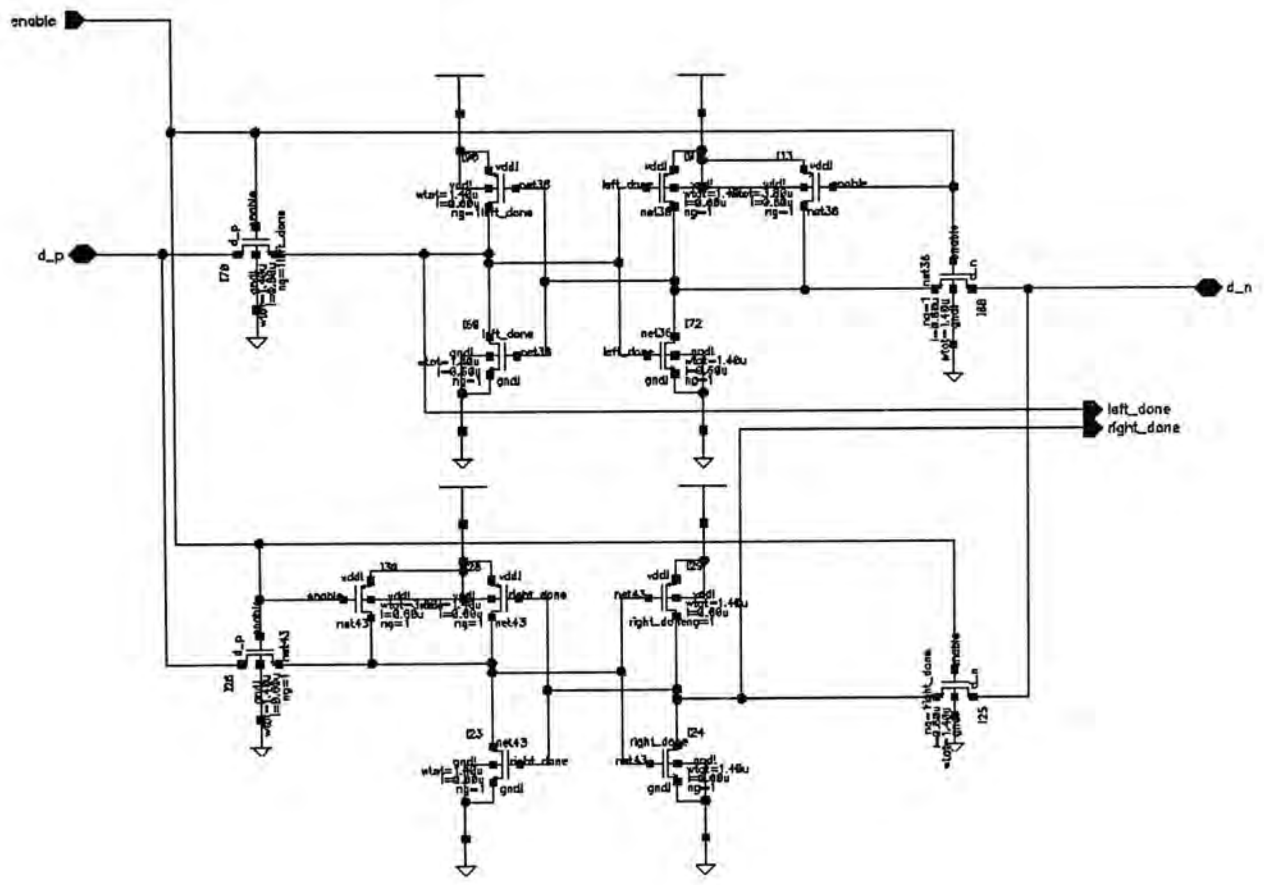
Schematic of write address generator



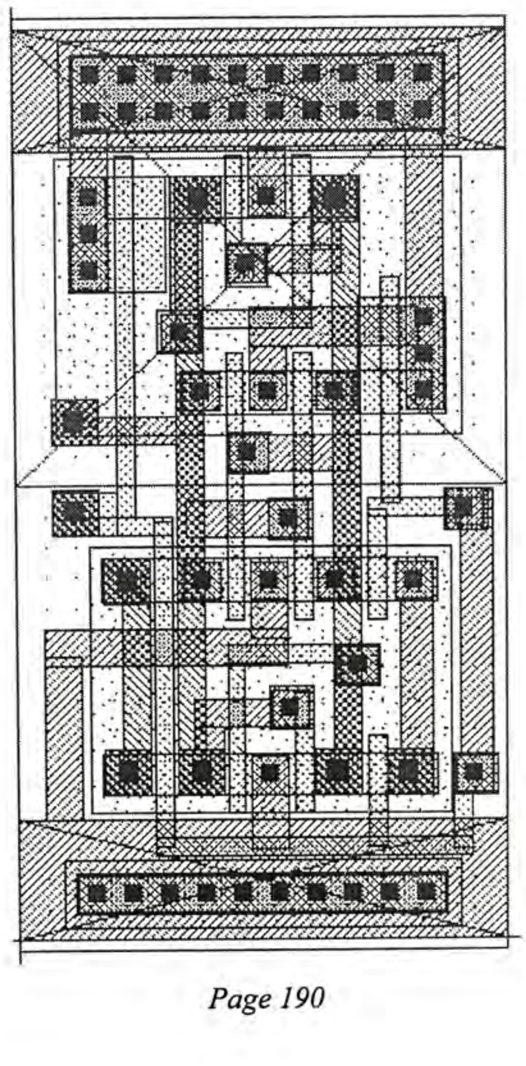
Schematic of 32×15bit RAM block



Schematic of monitor cell



Layout of monitor cell



References

[1] S. Hauck, "Asynchronous Design Methodologies: An Overview", Proceedings of the IEEE, Vol. 83, No. 1, page 69 – 93, January 1995

[2] K.D. Emerson, "Asynchronous Design – an Interesting Alternative", 10th International Conference on VLSI Design, page 315 – 321, January 1997

[3] N. Ahmed, T. Natatajan and K.R. Rao, "Discrete Cosine Transform", IEEE Transaction on Communications, Vol. 23, page 90 – 33, January 1974

[4] CCITT Recommendation H.261, 1990

[5] ISO/IEC JTCI/SC29/WG10. JPEG Committee Draft CD10918, 1991

[6] ISO/IEC JTCI/SC29/WG11. MPEG Committee Draft CD11172, 1991

[7] B.G. Lee, "A New Algorithm to Compute the Discrete Cosine Transform", IEEE Transaction on Acoustics, Speech, and Signal Processing, Vol. 32, No. 6, page 1243 – 1245, December 1984

[8] H.S. Hou, "A Fast Recursive Algorithm for Computing the Discrete Cosine Transform", IEEE Transaction on Acoustics, Speech, and Signal Processing, Vol. 35, No. 10, page 1445 – 1461, October 1987

[9] C. Loeffler, A. Ligtenberg and G.S. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications", International Conference on Acoustics, Speech, and Signal Processing, Vol. 2, page 988 – 991, 1989

[10] I.E. Sutherland, "Micropipelines", Communications of the ACM, Vol. 32, No. 6, page 720 – 738, June 1989

[11] J.V. Woods, P. Day, S.B. Furber, J.D. Garside, N.C. Paver and S. Temple, "AMULET1: An Asynchronous ARM Microprocessor", IEEE Transactions on Computers, Vol. 46, No. 4, page 385 – 398, April 1997

[12] T.E. Williams, "Analyzing and Improving the Latency and Throughput Performance of Self-Timed Pipelines and Rings", Proceedings of IEEE International Symposium on Circuits and Symstems, page 665 – 668, 1992

[13] T.E. Williams and M.A. Horowitz, "A Zero-Overhead Self-Timed 160ns 54b CMOS Divider", IEEE Journal of Solid-State Circuits, Vol. 26, No. 11, page 1651 – 1661, November 1991

[14] M. Singh and S.M. Nowick, "High-throughput Asynchronous Pipelines for Fine-Gain Dynamic Datapaths", Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems, page 198 – 209, 2000

[15] G. Matsubara and N. Ide, "A Low Power Zero-Overhead Self-Timed Division and Square Root Unit Combining a Single-Rail Static Circuit with a Dual-Rail Dynamic circuit", Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems, page 198 – 209, 1997

[16] D.E. Muller and W.C. Bartkey, "A Theory of Asynchronous Circuits", Report 75, Univerity of Illionis, USA, 1956

[17] M. Renaudin, B.E. Hassan and A. Guyot, "A New Asynchronous Pipeline Scheme: Application to the Design of a Self-Timed Ring Divider", IEEE Journal of Solid-State Circuits, Vol. 31, No. 7, page 1001 – 1013, July 1996

[18] R.H. Krambeck, C.M. Lee and H.S. Law, "High-speed Compact Circuits with CMOS", IEEE Journal of Solid-State Circuits, Vol. 17, page 614 – 619, June 1982

[19] A.J. McAuley, "Dynamic Asynchronous Logic for High-Speed CMOS Systems", IEEE Journal of Solid-State Circuits, Vol. 27, No. 3, page 382 – 388, March 1992

[20] C. Farnsworth, D.A. Edwards and S.S. Sikand, "Utilising Dynamic Logic for Low Power Consumption in Asynchronous Circuits", Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems, page 186 – 194, 1994

[21] H. Yoshizawa, K. Taniguchi and K. Nakashi, "An Implementation Technique of Dynamic CMOS Circuit Applicable to Asynchronous/Synchronous Logic", Proceedings of IEEE International Symposium on Circuits and Systems, Vol. 2, page 145 – 148, 1998

[22] J. Ahmed and S.G. Zaky, "Asynchronous Design in Dynamic CMOS", IEEE Canadian Conference on Electrical and Computer Engineering, Vol. 2, page 528 – 531, 1997

[23] G.N. Hoyer and C. Sechen, "A Locally-Clocked Dynamic Logic Serial/Parallel Multiplier", IEEE Custom Integrated Circuits Conference, page 481 – 484, 2000

[24] C.H. Erdekyi, W.R. Griffin and R.D. Kilmoyer, "Cascode Voltage Switch Logic Design", VLSI Design, page 78 – 86, October 1984

[25] S.B. Furber and J. Liu, "Dynamic Logic in Four-Phase Micropipelines", Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems, page 11 – 16, 1996

[26] M. Renaudin and B.E. Hassan, "The Design of Fast Asynchronous Adder Structures and Their Implementation Using DCVS Logic", Proceedings of IEEE International Symposium on Circuits and Systems, Vol. 4, page 291 – 294, 1994

[27] G.A. Ruiz and M.A. Manzano, "Compact 32-bit CMOS Adder in Multiple-Output DCVS Logic for Self-Timed Circuits", IEE Proceedings of Circuits and Devices Systems, Vol. 147, No. 3, page 183 – 188, June 2000

[28] K.M. Chu and D.L. Pulfrey, "A Comparison of CMOS Techniques; Differential Cascode Voltage Switch Logic versus Conventional Logic", IEEE Journal Solid-State circuits, Vol. 22. No. 4, page 528 – 532, August 1987

[29] C.D. Nielsen, "Evaluation of Function Blocks for Asynchronous Design", EURODAC 1994, page 454 – 459, September 1994

[30] T.Y. Tang, C.S. Choy, J. Butas and C.F. Chan, 'A ALU Design using a Novel Asynchronous Pipeline Architecture", Proceedings of IEEE International Symposium on Circuits and Systems, Sec. V, page 361 – 364, 2000

[31] K.R. Rao and P. Yip, " Discrete cosine Transform: Algorithms, Advantages, Applications", Academic Press, Inc, 1990

[32] S.I. Uramoto, Y. Inoue, A. Takabatake, J. Takeda, Y. Yamashita, H. Terance and M. Yoshimoto, "A 100-MHz 2-D Discrete Cosine Transform Core Processor", IEEE Journal of Solid-State Circuits, Vol. 27, No. 4, page 492 – 499, April 1992

[33] Y.F. Jang, J.N. Kao, J.S. Yang and P.C. Huang, "A 0.8u 100-MHz 2-D DCT Core Processor", IEEE Transactions on Consumer Electronics, Vol. 40, No. 3, page 703 – 710, August 1994

[34] T. Kuroda, T. Fujita, S. Mita, T. Nagamatsu, S. Yoshioka, K. Suzuki, F. Sano, M. Norishima, M. Murota, M. Kako, M. Kinugawa, M. Kakumu and T. Sakurai, "A 0.9-V, 150-MHz, 10-mW, 4mm^2, 2-D Discrete Cosine Transform Core Processor with Variable Threshold-Voltage (VT) Scheme", IEEE Journal of Solid-State Circuits, Vol. 31, No. 11, page 1770 – 1779, November 1996

[35] K.H. Cheng, C.S. Huang and C.P. Lin, "The Design and Implementation of DCT/IDCT Chip with Novel Architecture", Proceedings of IEEE International Symposium on Circuits and systems, Sec. IV, page 741 – 744, 2000

[36] N.I. Cho and S.U. Lee, "Fast Algorithm and Implementation of 2-D Discrete Cosine Transform", IEEE Transaction on Circuits and Systems, Vol. 38, No. 3, page 297 – 305, March 1991

[37] Y.P. Lee, T.H. Chen, L.G. Chen, M.J. Chen and C.W. Ku, "A Cost-Effective Architecture for 8×8 Two-Dimensional DCT/IDCT Using Direct Method", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 7, No. 3, page 459 – 467, June 1997

[38] T.S. Chang, C.S. Kung and C.W. Jen, "A Simple Processor Core Design for DCT/IDCT", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 10, No. 3, page 439 – 447, April 2000

[39] C.L. Wang and Y.T. Chang, "Highly Parallel VLSI Architectures for the 2-D DCT and IDCT Computations", Proceedings of TENCON, vol. 1, page 295 – 299, 1994

[40] Y. Jeong, I. Lee, T. Yun, G. Park and K.T. Park, "A Fast Algorithm Suitable for DCT Implementation with Integer Multiplications", IEEE TENCON – Digital Signal Processing Applications, Vol. 2, page 784 – 787, 1996

[41] J.B. Kuo and C.S. Chiang, "Charge Sharing Problems in the Dynamic Logic Circuits: BiCMOS versus CMOS and a 1.5 V BiCMOS Dynamic Logic Circuit Free from Charge Sharing Problems", IEEE Transaction on Circuits and Systems – I: Fundamental Theory and Applications, Vol. 42, No. 11, page 974 – 977, November 1995

[42] J.A. Pretorius, A.S. Shubat and C.A. Salama, "Charge Redistribution and Noise Margins in Domino CMOS Logic", IEEE Transaction on Circuits and Systems, Vol. 33, No. 8, page 786 – 793, August 1986

[43] L.A. Knauth, "Dynamic CMOS", Honors Project, http://www.stanford.edu/~lknauth/academic/DynCMOS.html

[44] H.J. Song, "A Self-Off-Time Detector for Reducing Standby Current of DRAM", IEEE Journal of Solid-State Circuits, Vol. 32, No. 10, page 1535 – 1542, October 1997

[45] J. Nyathi and J.G. Delgado-Frias, "Self-timed Refreshing Approach for Dynamic Memories", Proceedings of ASIC Conference, page 169 – 173, 1998

[46] N.R. Mahapatra, S.V. Garimella and A. Tareen, "An Empirical and Analytical Comparison of Delay Elements and a New Delay Element Design", Proceedings of IEEE Computer Society Workshop on VLSI, page 81 – 86, 2000

[47] R.J. Baker, H.W. Li and D.E. Boyce, "CMOS: Circuit Design, Layout, and Simulation", IEEE Press, 1997

[48] G.M. Jacobs and R.W. Brodersen, "A Fully Asynchronous Digital Signal Processor Using Self-Timed Circuits", IEEE Journal of Solid-State Circuits, Vol. 25, No. 6, page 1526 – 1537, December 1990

[49] T.M.Y. Meng, R.W. Brodersen and D.G. Messerschmitt, "Asynchronous Design for Programmable Digital Signal Processors", IEEE Transactions on Signal Processing, Vol. 39, No. 4, page 939 – 952, April 1991

[50] S.B. Furber, D.A. Edwards and J.D. Garside, "AMULET3: a 100 MIPS Asynchronous Embedded Processor", Proceedings of International Conference on Computer Design, page 329 – 334, 2000

[51] S.L. Lu and C.M. Chang, "Modelling of a Self Timed Dataflow Processor in VHDL", Proceedings of IEEE International ASIC Conference and Exhibit, page 228 – 231, September 1993

[52] I.S. Hwang and A.L. Fisher, "Ultra Fast Compact 32-bit CMOS Adder in Multiple-output domino Logic", IEEE Journal of Solid-State Circuits, Vol. 24, page 358-369, 1989

[53] Z. Wang, G.A. Jullien, W.C. Miller, J. Wang and S.S. Bizzan, "Fast Adders Using Enhanced Multiple-Output Domino Logic", IEEE Journal of Solid-State Circuits, Vol. 32, No. 2, page 206 – 214, February 1997

[54] S.M. Nowick, "Design of a Low-latency Asynchronous Adder Using Speculative Completion", Proceedings of IEE Computers and Digital Techniques, Part E, Vol. 143, No. 3, page 301 – 307, September 1996

[55] A. Peled, and B. Liu, "A New Hardware Realization of Digital Filters", IEEE Transaction Acoustic, Speech, Signal Processing, Vol. 22, page 456 – 462, December 1974

[56] "IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform", IEEE Std 1180-1990, December 1990

[57] M.E. Dean, D.L. Dill and M. Horowiz, "Self-timed Logic Using Current-Sensing Completion Detection", Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors, page 187 – 191, 1991

[58] T.C. Pang, "An ICT Image Processing Chip Based on Fast Computation Algorithm and Self-timed Circuit Technique", MPhil Thesis, Department of Electronic Engineering, The Chinese University of Hong Kong, 1997

[59] W.Y. Sit, "Asynchronous Memory Design", MPhil Thesis, Department of Electronic Engineering, The Chinese University of Hong Kong, 1998

[60] S.F. Hsiao, W.R. Shiue and J.M. Tseng, "A Cost-Efficient and Fully Pipelinable Architecture for DCT/IDCT", IEEE Transactions on Consumer Electronics, Vol. 43, No. 3, page 515 – 525, August 1999

[61] M. Yoshida, H. Ohtomo and I. Kuroda, "A New Generation 16-bit General Purpose Programmable DSP and Its Video Rae Application", IEEE Workshop on VLSI Signal Processing, page. 93-101, 1993

[62] I. Kuroda, "Processor Architecture Driven Algorithm optimization for Fast 2-D DCT', IEEE Workshop on VLSI Signal Processing, Vol. VIII, page 481 – 490, 1995

[63] D. Johnson, V. Akella and B. Stoot, "Micropipelined Asynchronous Discrete Cosine Transform (DCT/IDCT) Processor", IEEE Transactions on Very Large Scale Integration Systems, Vol. 6, No. 4, page 731 - 740, December 1998

[64] K. Kin and J.S. Koh, "An Area Efficient DCT Architecture for MPEG-2 Video Encoder", IEEE Transactions on Consumer Electronics, Vol. 45, No. 1, page 62 – 67, February 1999

[65] "0.6-Micron Standard Cell Databook", Austria Mikro Systeme International, 1997

[66] C.N. Lyu and D.W. Matula, "Reducing Binary Booth Recording", Symposium on computer Arithmetic, page 50 – 57, July 1995

[67] M. Potkonjak, M.B. Srivastava and A.P. Chandrakasan, "Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No 2, February 1996.

Design Libraries – CD-ROM

The CD contains the design libraries of the Refresh Control Circuit, programmable DSP processor, dedicated DCT/IDCT process and other necessary libraries. All the libraries are designed in the AMS CMOS CUP 0.6u 3M1P technology using the Cadence 4.4.1.

Design Libraries – CD-ROM

The CD contains the design libraries of the Refresh Control Circuit, programmable DSP processor, dedicated DCT/IDCT process and other necessary libraries. All the libraries are designed in the AMS CMOS CUP 0.6u 3M1P technology using the Cadence 4.4.1.