HARDWARE REALIZATION OF DISCRETE WAVELET TRANSFORM CAUCHY REED SOLOMON MINIMAL INSTRUCTION SET COMPUTER ARCHITECTURE FOR WIRELESS VISUAL SENSOR NETWORKS

ONG JIA JAN, MEng (Hons.)

THESIS SUBMITTED TO THE UNIVERSITY OF NOTTINGHAM
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
JULY 2016

ABSTRACT

Large amount of image data transmitting across the Wireless Visual Sensor Networks (WVSNs) increases the data transmission rate thus increases the power transmission. This would inevitably decreases the operating lifespan of the sensor nodes and affecting the overall operation of WVSNs. Limiting power consumption to prolong battery lifespan is one of the most important goals in WVSNs. To achieve this goal, this thesis presents a novel low complexity Discrete Wavelet Transform (DWT) Cauchy Reed Solomon (CRS) Minimal Instruction Set Computer (MISC) architecture that performs data compression and data encoding (encryption) in a single architecture. There are four different programme instructions were developed to programme the MISC processor, which are Subtract and Branch if Negative (SBN), Galois Field Multiplier (GF MULT), XOR and 11TO8 instructions. With the use of these programme instructions, the developed DWT CRS MISC were programmed to perform DWT image compression to reduce the image size and then encode the DWT coefficients with CRS code to ensure data security and reliability. Both compression and CRS encoding were performed by a single architecture rather than in two separate modules which require a lot of hardware resources (logic slices). By reducing the number of logic slices, the power consumption can be subsequently reduced. Results show that the proposed new DWT CRS MISC architecture implementation requires 142 Slices (Xilinx Virtex-II), 129 slices (Xilinx Spartan-3E), 144 Slices (Xilinx Spartan-3L) and 66 Slices (Xilinx Spartan-6). The developed DWT CRS MISC architecture has lower hardware complexity as compared to other existing systems, such as Crypto-Processor in Xilinx Spartan-6 (4828 Slices), Low-Density Parity-Check in Xilinx Virtex-II (870 slices) and ECBC in Xilinx Spartan-3E (1691 Slices). With the use of RC10 development board, the developed DWT CRS MISC architecture can be implemented onto the Xilinx Spartan-3L FPGA to simulate an actual visual sensor node. This is to verify the feasibility of developing a joint compression, encryption and error correction processing framework in WVSNs.

LIST OF ASSOCIATED PUBLICATIONS

Published Papers

Journal:

Jia Jan Ong, L.-M. Ang, and K.P. Seng, “Selective Secure Error Correction on SPIHT Coefficients for Pervasive Wireless Visual Network,” International Journal of Ad Hoc and Ubiquitous Computing, vol. 13, pp. 73-82, 2013.

Book Chapter:

Jia Jan Ong, L.-M. Ang and K. P. Seng, “Lifting Scheme DWT Implementation in a Wireless Vision Sensor Network”, Visual Informatics: Bridging Research and Practice, Lecture Notes in Computer Science, Vol. 5857/2009, pp. 627-635, 2009.

Jia Hao Kong, Jia Jan Ong, L.-M. Ang, and K. P. Seng, “Low Complexity Processor Designs for Energy-Efficient Security and Error Correction in Wireless Sensor Networks”, Wireless Sensor Networks and Energy Efficiency: Protocols, Routing and Management, pp. 350-368, January 2012.

Conference Paper:

Jia Jan Ong, Jia Hao Kong, L.-M. Ang, and K. P. Seng, “Implementation of the One Instruction Set Computer (OISC) on FPGA using Handel-C”, The International Conference on Embedded Systems and Intelligent Technology ICESIT 2010 Proceedings, February 2010.

Jia Jan Ong, L.-M. Ang, and K. P. Seng, “Implementation of (255,223) Reed Solomon Minimal Instruction Set Computing using Handel-C,” 3rd IEEE International Conference on Computer Science and Information Technology (ICSIT) 2010, Vol. 5, pp. 49-54, July 2010.

Jia Jan Ong, L.-M. Ang, and K.P. Seng, “Implementation of (15, 9) Reed Solomon Minimal Instruction Set Computing on FPGA using Handel-C,” 2010 International Conference on Computer Applications and Industrial Electronics (ICCAIE), pp. 356-361, December 2010.

Jia Jan Ong, L.-M. Ang, K.P. Seng, and Fong Tien Ong , “Implementation of (255, 251) Reed Solomon Minimal Instruction Set Computing using Handel-C,” 2011 International Conference on Information Networking (ICOIN), pp. 429-434, January 2011

Jia Jan Ong, L. -M. Ang, and K.P Seng, “FPGA Implementation Reed Solomon Encoder for Visual Sensor Networks,” the 2011 International Conference on Telecom Technology and Applications (ICTTA 2011), Proc. of CSIT, Vol 5, pp. 88-92, 2011

Jia Jan Ong, L.-M Ang, K.P.Seng, and Ong Fong Tien, "Implementation of Selective Error Protection on SPHIT coefficients for wireless visual network", Proceedings of the 3rd International Conference on Software Technology and Engineering, pp. 383-388, 2011.

ACKNOWLEDGEMENT

First and foremost, I would express my greatest gratitude to my supervisor Dr. Kenneth Ang Li-Minn for his guidance in my PhD studies. His creative thoughts and wide knowledge in the relevant field have greatly motivated me to put my best effort in completing this study. Nevertheless, his never ceasing patience in giving guidance for this study has gained my utmost respects.

At the same time, I would like to thank my interim supervisor Dr. Wong Yee Wan who has given me guidance, encouragement and support during difficulties time. I would like to thank my co-supervisor Prof. Dr. Jasmine Seng Kah Phooi, my family members and peers for their strong moral support, giving me the courage and strength during the duration of research. They have encouraged me to continue working on the problems when I encountered obstacles. With these efforts and words of encouragement, I strived to continue on the research work even when I meet with difficulties. Once again I would like to express my sincere appreciation for those who have directly and indirectly contributed to this piece of writing and research.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ADC	Analogue-to-Digital Converter
AES	Advanced Encryption Standard
ARQ	Automatic Request for Retransmission
AWGN	Additive White Gaussian Noise
BCH	Bose, Chaudhuri, and Hocquenghem
BER	Bit Error Rate
BIBO	Binary-Input and Binary-Output
CCD	Charge Coupled Device
CISC	Complex Instruction Set Computer
CL-DCT	Cordic Loeffler Discrete Cosine Transform
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CPU	Computer Processing Unit
CRC	Cyclic Redundancy Check
CRS	Cauchy Reed Solomon
CS	Compressed Sensing
CWT	Continuous Wavelet Transform
DES	Data Encryption Standard
DWT	Discrete Wavelet Transform
ECBC	Error Correction-Based Cipher
ECC	Elliptic Curve Cryptography
ECC	Error-Correction Code
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
EEPROM	Electrically Erasable Programmable Read-Only Memory
FEC	Forward Error Correction
FPGA	Field Programmable Gated Array
GF	Galois Field
GF MULT	Galois Field Multiplier

HARQ	Hybrid ARQ/FEC
HD	High Diffusion
HEX	Hexadecimal
IBBSC	Identity-Based Broadcast Signcryption Scheme
IBE	Identity-Based Encryption
ICES	Image Compression Encryption Scheme (ICES)
IPJ	Information Per Joule
JPEG	Joint Photographic Experts Group
LBT	Lapped Biorthognal Transform
LDPC	Low-Density Parity-Check
LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit
LUT	Look-Up-Table
LZW	Lempel-Ziv-Welch
MAR	Memory Address Register
MDR	Memory Data Register
MIMO	Multiple Input Multiple Output
MIPS	Microprocessor without Interlocked Pipeline Stages
MISC	Minimal Instruction Set Computer
MIT	Multiple-Input Turbo
MSB	Most Significant Bit
MUX	Multiplexer
MVMP	Multi-Version Multi-Path
NIST	National Institute of Standards and Technology
OISC	One Instruction Set Computer
PC	Programme Counter
PCCC	Parallel Concatenated Convolutional Code
PKC	Public-Key Cryptography
PKE	Public-Key Encryption
PSNR	Peak Signal-to-Noise Ratio
RAM	Random Access Memory
RF	Radio Frequency
RFID	Radio-Frequency Identification

RISC	Reduced Instruction Set Computer
RS	Reed Solomon
SBN	Subtract and Branch if Negative
SEC	Secure Erasure Coding
SECDED	Single-bit Error and Detect Double-bit Errors
SKC	Secret-Key Cryptography
SLCCA	Significance-Linked Connected Component Analysis
SNEP	Secure Network Encryption Protocol
SNR	Signal-to-Noise Ratio
SOT	Spatial Orientation Tree
SPIHT	Set Partitioning In Hierarchical Trees
SPINS	Security Protocols for Sensor Networks
SRAM	Static Random-Access Memory
UMC	United Microelectronics Corporation
URISC	Ultimate Reduced Instruction Set Computer
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WISN	Wireless Image Sensor Network
WMSN	Wireless Multimedia Sensor Network
WSN	Wireless Sensor Network
WVSN	Wireless Visual Sensor Network
XOR	Exclusive-OR

CHAPTER 1

INTRODUCTION

Wireless Sensor Networks (WSNs) consists of many tiny sensor nodes that are capable of sensing the environment conditions, process the sensed data and send the data to the sink. At the sink, the received data will be displayed and analyzed either by computer(s) or human being(s). With these capabilities, the WSNs has a wide range of applications such as health, military and security [1]. The rapid deployment, self-organisation and fault tolerance characteristic of sensor networks, make them very important in military command, control, communications, reconnaissance and targeting systems [2].

With advances of image sensor technology, many products are now embedded with low-powered image sensors, such as cellular phones, computers, toys and robots. At the same time, recent development in WSNs and distributed processing have promoted the use of image sensors for the network that resulted the development of Wireless Visual Sensor Networks (WVSNs) [3], where sometimes it is referred to as Visual Sensor Networks (VSNs) [4]. The WVSNs provides a broad range of applications such as remote and distributed video-based surveillance systems that collect visual data from a network with smart distributed image sensor nodes. These systems can be connected to the Internet which allow authorized Internet users to do remote visiting interesting locations (i.e. virtual reality), monitoring the environment, surveillance of sensitive headquarters and industrial process control [5] [6].

However, most of the WSNs applications have low bandwidth demands with delay tolerant and they measure the physical environment conditions such as pressure, humidity, and temperature. Once the sensor nodes are deployed, they are usually battery driven and operated with sacred energy source [7]. Furthermore, these sensor nodes are required to operate for months or years since battery replacement is not recommended for networks with thousands of distributed physically embedded nodes. The introduction of image sensors to WSNs generates large amount of image data into the widely distributed visual sensor nodes. Consequently. this requires larger network bandwidth usage thus higher energy consumption is required for transmitting the large amount of image data [3].

The thesis presents a novel low complexity Discrete Wavelet Transform (DWT) Cauchy Reed Solomon (CRS) Minimal Instruction Set Computer (MISC) architecture that performs data compression, data encryption and data correction in a single architecture. The developed DWT CRS MISC processor was programmed to perform DWT image compression to decrease the image data thus subsequently reduces the network bandwidth. Later on, the programmed MISC processor performs the CRS coding scheme onto the reduced image data to provide secure and reliable data transmission. With the CRS encoded data, the base-station can correct a small number of errors occurred onto the received image data thus it requires less number of retransmission. The developed new DWT CRS MISC architecture was implemented into a Field Programmable Gate Array (FPGA) to demonstrate its feasibility for use in the WVSNs.

1.1 PROBLEM STATEMENT

Transmitting large amount of image data requires longer transmission time that increases the power consumption of transceiver thus reduces the life span of sensor nodes [8]. Therefore, introducing compression scheme into the WVSNs to reduce the amount image data to be transmitted has a great potential in reducing communication energy costs and thus increasing the sensor nodes operating lifespan [9]. Meanwhile, image data transmitted across the wireless communication channel are prone to security threat such as eavesdropping [10]. Adversary can intercept the unencrypted data transmission in the WVSNs and learn any important information from the intercepted data. In the meantime, it requires high retransmission rate when the WVSNs operates in noisy environment, whereby the base-station often requests the sensor nodes to retransmit the received error image data. Consequently, this increases the power consumption of the transceiver which may reduce the operating lifetime of the sensor nodes [5]. As a result, there is a need to develop a low complexity image processing system to address the aforementioned issues in the resource constrained WVSNs.

The first issue to be addressed is to have an image processing system that reduces large amount of image data produced by visual sensor, before transmitting the data across the WVSNs. The cost of transmitting 1kB data (in terms of energy) is comparable to the same amount of energy used by a general-purpose processor executes 3 million instructions [11]. As a result, this provides improvement by reducing the power consumption for transmitting lower amount of image data [12].

Second issue to be resolved is the security protection on the image data that are to be transmitted across wireless channel. Sensitive data such as surveillance image that are used for military purposes, especially in providing information on vital battlefield telemetry or monitoring [13], are exposed to security threats across the wireless communication channel. These image data transmitted across the WVSNs are prone to adversary attacks and eavesdrops on the wireless communication network [14].

Lastly, the compressed and encrypted image data are usually prone to error while transmitting the data across the noisy wireless communication channel. Although the WVSNs do have error correction capabilities at the link layer [15], it is still not sufficient because unreliable data transmission will results in incorrect encrypted data received by the base-station (sink) or by remote user through the Internet. Thus decrypting correct plaintext is not possible with the use of these corrupted ciphertext (encrypted data). With these three aforementioned issues, there is a need to develop a novel solution (architecture) that could resolve the issues. The novel architecture need to be of low hardware utilisation, such that low power consumption is achieved.

1.2 SYSTEM OVERVIEW

An image processing system for WVSNs was developed to resolve the aforementioned issues in Section 1.1. As shown in Figure 1, the final developed image processing system consists of a novel DWT CRS MISC architecture. The proposed MISC architecture has the capability of reducing the large amount of image data, and at the same time provides data security and reliable data transmission in a single architecture.

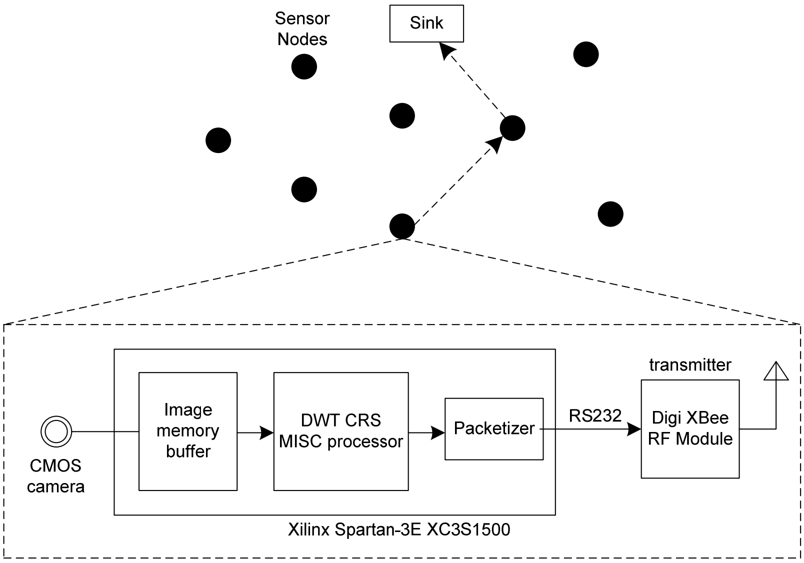


Figure 1 Developed image processing system with DWT CRS MISC.

Modifications were made onto the Subtract and Branch if Negative (SBN) architecture [16] such that it can be further developed into DWT CRS MISC architecture. The additional functional blocks were added into the modified SBN architecture, which are Galois Field (GF) [17] Multiplier (MULT) block, XOR block and 11TO8 block. By adding these functional blocks, the SBN (i.e. One Instruction Set Computer [18]) was developed into a MISC architecture that consists of a minimal number of instructions.

The developed DWT CRS MISC architecture consists 4 programme instructions, which are SBN, GF MULT, XOR and 11TO8 instructions. Based on these programme instructions, the DWT CRS MISC was programmed to perform Lifting Scheme DWT image compression [19] to reduce the image data and CRS encoding (encryption) [20] onto the selected DWT coefficients to provide secure and reliable data transmission. Therefore, the developed system reduces the amount of image data that are required to be transferred across the WVSNs and thus decreases the network bandwidth. At the same time, the compressed image data were encoded using the CRS coding scheme [20] to provide Forward Error Correction (FEC) capability such that the base-station can correct the errors that occurred onto the received image data. This reduces the number of request for retransmission of incorrect received data by the base-station. Thus less amount of energy is consumed by the sensor nodes since lower data retransmission rate is required. Besides, the CRS encoded image data are also encrypted such that any adversary cannot eavesdrop and extract information from the intercepted the data.

1.3 RESEARCH AIM AND OBJECTIVES

The aim of research works presented in this thesis is to develop a low complexity joint compression, forward error correction and encryption processing framework for resource constrained WVSNs. The main objectives of the research works are listed as follows:

1. Develop an alternative processing approach for the Reed Solomon (RS) encoding scheme to be used in the WVSNs such that the base station can correct the errors that occurred onto the RS encoded image data. Therefore, less retransmission of image data are required from the sensor nodes when incorrect data are received.

2. Further improvement on the RS processing approach such that the CRS encoding scheme can be integrated into the WVSNs. The CRS processing approach allows encryption and FEC to be performed onto the image data transmitted from the sensor nodes.

3. Develop a new processing approach for combined DWT filtering and CRS encoding scheme to be used in the WVSNs such that image compression, encryption and FEC can be performed onto the image data transmitted from the sensor nodes.

4. Design of a custom visual sensor platform with the DWT CRS MISC architecture onto a reconfigurable hardware to verify the feasibility of implementing the developed low complexity joint compression, encryption and FEC image processing system for image data transmitted from the sensor nodes.

1.4 SIGNIFICANCE OF RESEARCH

A new low complexity RS MISC architecture was developed for use in WVSNs to provide reliable data transmission. The developed RS MISC architecture has lower hardware utilisations (61.2% less Slices) as compared to the traditional RS Linear Feedback Shift Register (LFSR) encoder circuit [17]. Both the RS MISC and RS LFSR encoder were implemented in the Xilinx Spartan-3L FPGA. Meanwhile, the power consumption of developed RS MISC architecture is also reduced by 17.4%. Because the developed RS MISC architecture has only one GF Multiplier block. The RS MISC does not encode the data as fast as the RS LFSR, which has many parallel GF Multiplier blocks. However, for WVSNs application, the interval between image transmissions to sink may be in hours, days, weeks or months. Therefore, the use of RS MISC in the WVSNs is justifiable as achieving low power consumption is of the utmost importance. With RS MISC, it provides error protections onto the image data such that less request for retransmission is required. With lower retransmission rate, less amount of energy is consumed by the sensor nodes.

Further improvement was made to the RS MISC architecture, whereby a CRS MISC architecture was developed to provide both data security and data reliability. With the CRS MISC, the encoded image data are protected from any eavesdropping and errors while the data are transmitted across the wireless communication network.

In comparison to the existing ECBC method [21], which were used to provide security and reliable data transmission, the CRS MISC architecture requires less than 90.8% of the hardware utilisations that were required by the ECBC. The existing ECBC method was developed in Xilinx Spartan-3E FPGA, whereby the Spartan-3E FPGA builds on the Spartan-3 family of FPGA [22]. The contents of this part of research forms part of the work published in [23].

Further development was made onto the CRS MISC architecture such that image compression capability was incorporated in the architecture. Therefore, a DWT CRS MISC architecture was developed and proposed for use in WVSNs to reduce size of the image data, provide data security and data reliability. With the DWT CRS MISC, the DWT image compression is performed first and followed by CRS coding scheme that encodes (encrypt) the compressed image data. To the knowledge of the author, the DWT CRS MISC architecture was the only jointed compression, encryption and error correction scheme processing framework for WVSNs that was ever developed. In comparison to the literature in [24], an existing simulation study was performed to show the feasibility of using the existing SAC (jointed compression and encryption techniques) combined with Multiple-Input Turbo (MIT) error correcting coding technique for WSNs. However, the SAC combined with MIT code technique only process scalar data (eg. temperature) that allowed occasional losses of sensor measurements and this technique is impractical for WVSNs that has larger data traffic [5].

By considering existing methods, with the lowest hardware utilisations, both CL-DCT compression [25] and ECBC [21] requires 1060 Slices (Xilinx Spartan-3L) and 1,691 Slices (Xilinx Spartan-3E) respectively. The developed DWT CRS MISC architecture only requires 144 Slices (Xilinx Spartan-3L) and 129 Slices (Xilinx Spartan-3E) respectively, which is comparatively a very low amount of hardware utilisations. Note that the CL-DCT, ECBC and DWT CRS MISC were implemented in the Xilinx Spartan-3 family of FPGA. Therefore, the developed DWT CRS MISC performs both DWT image compression and CRS encoding in a single architecture such that low hardware complexity of the proposed image processing system was achieved.

1.5 THESIS ORGANISATION

This thesis presents the concepts, approaches and methods that are involved to develop the new low complexity joint compression, FEC and encryption processing approach for WVSNs with adhere to the constrained hardware resources. The thesis organisations are as follow:

Chapter 1 – Introduction

For this chapter, it gives the introduction on the related area of research and provides the motivations of this research. From the problem statements, the listed problems were addressed by the proposed image processing system stated in this chapter.

Chapter 2 – Literature Review

This chapter reviews on the existing image processing system in WVSNs. The reviewed image processing systems are those that provide data compression, data encryption and data reliability. The chapter also provides the necessary details on the Galois Field, Lifting Scheme DWT and CRS coding scheme that are required by the MISC architectures developed.

Chapter 3 – DWT CRS Minimal Instruction Set Computer Architecture

The chapter explains the process of developing the new DWT CRS MISC architecture. The detail explanations on the DWT CRS MISC architecture are presented in this chapter. Besides, the algorithm to perform the joint processing schemes framework is discussed. As such, the written programme instructions are listed out and the number of clock cycles required to run these instructions are estimated.

Chapter 4 – Results and Discussions

The results and simulation waveforms for the developed DWT CRS MISC architecture are discussed in this chapter. Analysis shows the improvement of data transmission time, reconstructed image quality for errors occurred on compress image data and security level of the encoded image data are presented in this chapter.

Chapter 5 – Hardware Implementations

The focus in this chapter is to develop a custom Wireless Visual Sensor Platform. This is done by integrating the proposed DWT CRS MISC architecture with a wireless transceiver such that to verify its feasibility for use in the WVSNs.

Chapter 6 – Conclusions and Future Works

The last chapter of the thesis provides conclusion on the research works and offers suggestions on future works that can be performed through the use of the newly developed DWT CRS MISC architecture discussed from previous chapters.

CHAPTER 2

LITERATURE REVIEW

Chapter 2 reviews related existing research works that were performed by the research community. First, a brief introduction on the DWT was given in Section 2.1. This includes the reason of using DWT over CWT are also discussed. Secondly, Section 2.2 provides a brief introduction on the Reduced Instruction Set Computer (RISC) architecture. The DWT CRS MISC architecture was developed based on the Subtract and Branch if Negative (SBN) architecture. Section 2.3 discusses on the Wireless Visual Sensor Networks (WVSN) and the existing WVSN platforms that are available in the literature. Section 2.4 reviews the existing compression techniques that were used in Wireless Sensor Networks (WSNs). Section 2.5 discusses on the available proposed FEC encoding scheme for WSNs. Section 2.6 reviews on the existing security schemes for WSNs. Section 2.7 reviews on the combined schemes that provides data compression, secure and reliable data transmissions in WSNs.

2.1 DISCRETE WAVELET TRANSFORM

Discrete Wavelet Transform (DWT) is known to be a wavelet transform that processes and represents a continuous-time signal [26]. The DWT is also used as a means to extract relevant features from signals [27], images [28] and video [29]. In contrast to the DWT, the Continuous Wavelet Transform (CWT) compares the analysed signals (images) with many shifted and stretched (scaled) wavelet. The CWT uses only 1 filter and produces a large amount of wavelet coefficients. As a result, the CWT is considered to produce more redundant wavelet coefficients [30] and this does not reduce the amount of data representing the signals (mainly used in signal processing). Whereas for the DWT, it uses both high pass filter and low pass filter to analyse the signals (images), with each filter's outputs are down-sampled by 2 [31]. As a result of down-sampling, the DWT high pass filter produces Detail DWT coefficients and the DWT low pass filter produces Approximate DWT coefficients, with each of these DWT coefficients are half the length of the original signals. The Detail DWT coefficients represent the high frequency of the signals (or represent the edges in images) and the Approximate DWT coefficients represent the Low frequency of the signals (or represent an approximation of the images). With half the amount of data to represent the original signals (images), reconstruction of signals (images) is made possible with the use of Approximate DWT coefficients only. To reduce the amount of data representing the signals (images), the DWT was used for image compression because it has the advantages which overcome the wavelet redundancy issues encountered in CWT [32]. There are two different approaches of DWT, one is through the use of traditional filter bank [26] and another will be lifting scheme [19] [33].

2.1.1 Recent DWT and CWT Research Works

In [34], image compression integrated with selective encryption was introduced. The authors proposed the use of combined Embedded Zerotrees of Wavelet (EZW) transforms [35] and random shuffle or permute the wavelet Spatial Orientation Trees (SOTs). The DWT is used to decompose the image data and then a randomized key is used to shuffle the wavelet trees (SOTs). Later, it was found that this proposed method, when used as the only security mechanism, is insure against a chosen plaintext attack [36]. With the knowledge on wavelet trees configurations, an adversary would be able to predict the correct size of image and number level of decompositions. The intercepted cipher-text will leak information on the randomized key used to shuffle the SOTs. Therefore, this allows the adversary to decipher the encrypted compressed image data.

Later on, both CWT and DWT were used for analyzing non-stationary and quick changing Partial Discharge (PDs) signals [37]. The PDs measurement is a method to diagnose the insulating system condition of High Voltage (HV) electrical equipment. With the use of CWT, it calculates the wavelet coefficients at every possible scale and along every time instant. This result in having a wavelet details coefficients distribution throughout the entire time-scale view. For the de-noising process, a model-based DWT simulator is used to filter the corrupted PDs signals. With noise disturbance has been thinned out, the unchanged waveform of PDs can then be extracted [37].

A study and analysis on the usage of these techniques - Principle Component Analysis (PCA), Discrete Cosine Transform (DCT) and DWT, to perform the image fusion (processing) [38]. Image fusion is used to improve the quality of information from a set of images. From this study, the authors found that the use of DWT technique itself in fusing the images, does provide a significant fused image quality of 78.9555 dB. Compared to both PCA and DCT techniques, the fused image quality obtained were only 30.7729 dB and 30.9663 dB respectively.

The used of CWT for power limited wearable Electroencephalography (EEG) was introduced by the literature [39]. The EEG is a device that measures the voltage of electrodes placed between the scalp. Usually the voltages recorded are in between peak-to-peak voltage of 1 - 150μV over a bandwidth frequency of 1-70 Hz. In this research work, the CWT will be used to perform physiological (analogue) signal processing. As a result, a low power g_mC Low Power CWT (LPCWT) filter was developed using 0.35μm CMOS technology process. The developed LPCWT filter is capable in reducing analogue domain signals, such that it lowers the total system power consumption of EEG [39].

Meanwhile, the surface Electrocardiogram (ECG) is a non-invasive tool for diagnosis of many heart diseases. In order to detect abnormal cardiac events, continuous electrical recording of heart behaviour is performed. In order to extract the relevant information from ECG signal, the literature [40] proposed a method in segmenting and analyzing the waveforms of ECG signal. For segmentation, multi-scale CWT is used to detect different ECG waveforms. Later on, the PCA is used in detecting and locating defects. It is done by modelling the behaviour of the biological process in a normal state. Then the PCA compares the observed behaviour with the normal state behaviour. This is used to detect any defects that may occur [40]. Next, the authors in [41] had proposed and designed a CWT-based multi-functional processor. The CWT-based processor is suitable for long-term real-time ECG signal analysis and abnormal cardiac event detection. Based on reported results, it was concluded that the proposed architecture is feasible to use in continuous ECG monitoring system [41].

Table 1 summarizes the recent DWT and CWT research works that were mentioned in this section. It can be seen that the CWT transform was mainly used in signal processing. As mentioned in [42], the main usage of CWT is for analysis of signals (images). For images, the CWT is usually used to detect specific features - hierarchical structure, edges, contours etc. The CWT are very efficient in detecting specific features in signals or images [42]. As oppose to CWT, the DWT is mainly used when data compression is considered to be essential. The DWT produces impressive data compression rates that are very useful in image processing [42].

Table 1 List of recent research works related to DWT and CWT for image/signal processing.

Authors	**Year**	**Research Works**	**Type of Wavelet**	**Summary**	**Applications**
Salama, Paul; King, Brian [34]	2005	Efficient Secure Image Transmission	EZW (DWT)	DWT image compression	Image Processing
Candela, R.; Romano, E.; Romano, P. [37]	2009	Combined CWT-DWT Method using Model-based Design Simulator	CWT / DWT	Filtering onto online partial discharges signals measurement systems, after analysis CWT coefficient, with model-based DWT simulator.	Signal Processing
Assegie, Samuel; Salama, Paul; King, Brian [36]	2010	An Attack on Wavelet Tree Shuffling Encryption Schemes	DWT	Proved that the research work [34] is not strong against plaintext attack.	Image Processing
Casson, A. J.; Rodriguez-Villegas, E. [39]	2011	60pW g_mC CWT Circuit	CWT	Portable EEG systems with 7th order CWT band pass filter for analogue physiological signals	Signal Processing
Hanen Chaouch; Khaled Ouni; Lotfi Nabli [40]	2012	Segmenting and supervising an ECG signal	CWT	Segmentation and analysis of ECG signal with CWT and PCA	Signal Processing
Li-Fang Cheng; Tung-Chien Chen [41]	2012	Wavelet-based ECG Microprocessor	CWT	Real-time ECG analysis and abnormal cardiac event detection	Signal Processing
Desale, R. P.; Verma, S. V. [38]	2013	Study and Analysis of PCA, DCT & DWT based Image Fusion Techniques	DWT	Analysis shows that DWT based fusion techniques provide good quality fused images than PCA & DCT based techniques	Image Processing

2.1.2 DWT and CWT Comparison

In Section 2.2.1, there are many literatures that focus on the usage of DWT for image processing. In order to show the difference in image compression performance for both CWT and DWT, MATLAB simulations were performed. The same “lena1.tif” image (256 x 256 pixels) was used to perform the simulations that determine the quality of reproduced compressed image. For the CWT image compression, the CWT Fast Fourier Transform based algorithm was considered. This is the available MATLAB function, capable in performing both forward and inverse CWT transform onto the image data.

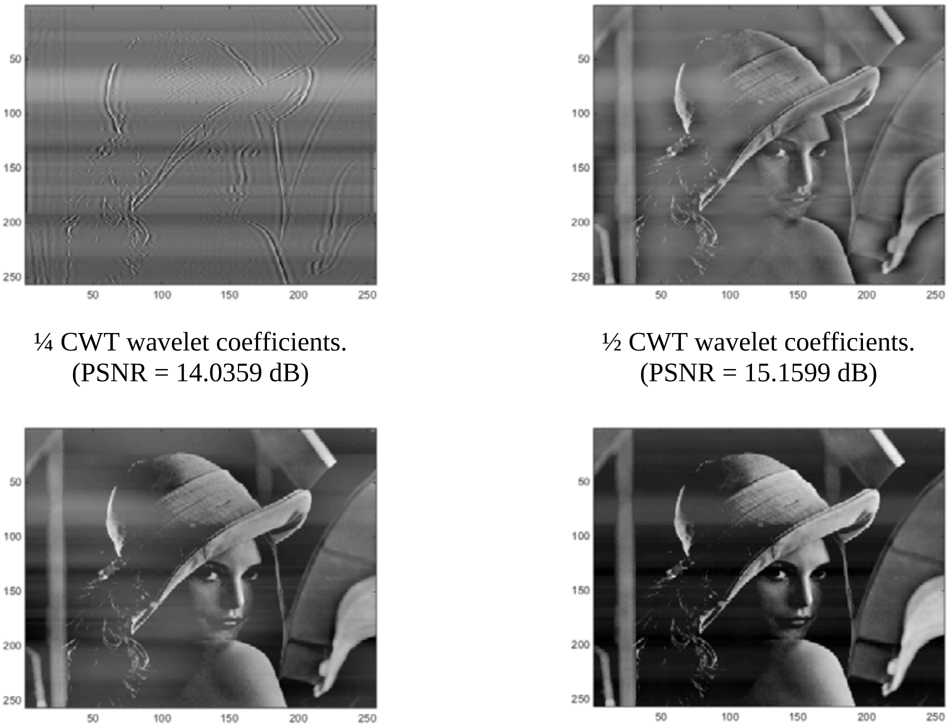
For each row of the image data, the CWT transform produces a total of 17 x 256 = 4,352 real and imaginary CWT wavelet coefficients. Since there are a total of 256 rows of image data, a total of 4,352 x 256 = 1,114,112 Real and Imaginary (number) CWT wavelet coefficients are produced. By summing up both Real and Imaginary CWT wavelet coefficients, for a single image, the total amount of CWT wavelet coefficients produced are 2,228,224. Since each wavelet coefficients are represented by 7-bit signed integer (1 Byte). The memory spaces needed to store the wavelet coefficients are approximately 2.228 MBytes. Further on, taking all the CWT coefficients, an inverse CWT transform was performed to reproduce the original image. In Figure 2(a), using all the CWT wavelet coefficients, the quality of the reproduced compressed image is 19.4513 dB. For image compression, only ¾, ½ and ¼ of all the CWT coefficients are used to reconstruct the images. The quality of the reproduced images are 17.3952 dB, 15.1599 dB and 14.0359 dB respectively. These reproduced images are shown in Figure 2(b), Figure 2(c) and Figure 2(d) respectively.

Subsequently, the DWT transform performed onto the ‘lena1.tif’ image and produces a total of 65,536 DWT wavelet coefficients. With each wavelet coefficients are represented by 7-bit integer (1 Byte), total amount of memory needed is 65,536 Bytes. This the similar amount of memory spaces needed to store ‘lena1.tif’ image data (256 x 256 pixels/Bytes), where each pixels is represented by 1 Byte. By taking all the wavelet coefficients, a lossless compressed image can be reproduced with the image quality measured as infinite, as shown in Figure 3(a). To consider for image compression, only ¾, ½ and ¼ of wavelet coefficients will be used to reproduce the image. The quality of the reproduced images are 37.1123 dB, 28.9856 dB and 27.6737 dB respectively, which are shown in Figure 3(b), Figure 3(c) and Figure 3(d).

The simulation results for both CWT and DWT transform image compressions are listed in Table 2.

Table 2 MATLAB simulation results for image compression performed using CWT and DWT.

	CWT Coefficients				DWT Coefficients			
	Real Number (Bytes)	***Imaginary Number (Bytes)***	***Total Data (Bytes)***	***PSNR (dB)***	***Real Number (Bytes)***	***Imaginary Number (Bytes)***	***Total Data (Bytes)***	***PSNR (dB)***
All Wavelet Coefficients	1,114,112	1,114,112	2,228,224	19.4513	65,536	0	65,536	∞
3/4 Wavelet Coefficients (Approx.)	851,968	851,968	1,703,936	17.3952	49,152	0	49,152	37.1123
2/4 Wavelet Coefficients (Approx.)	589,824	589,824	1,179,648	15.1599	32,768	0	32,768	28.9856
1/4 Wavelet Coefficients (Approx.)	262,144	262,144	524,288	14.0359	16,384	0	16,384	27.6737



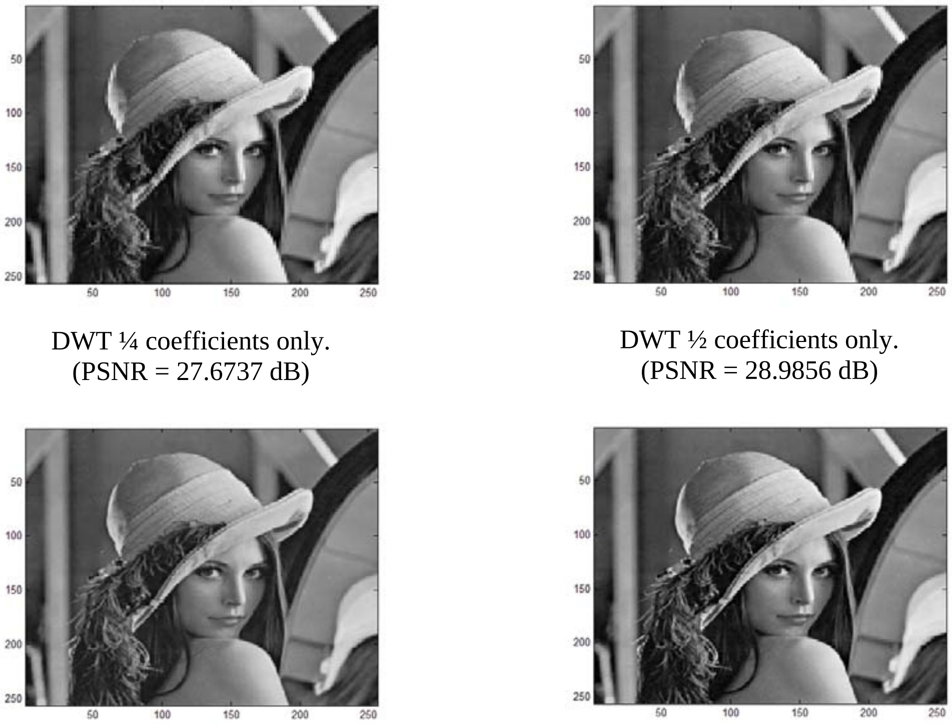
¼ CWT wavelet coefficients.
(PSNR = 14.0359 dB)

½ CWT wavelet coefficients.
(PSNR = 15.1599 dB)

¾ CWT wavelet coefficients.
(PSNR = 17.3952 dB)

All CWT wavelet coefficients.
(PSNR = 19.4513 dB)

Figure 2 Reproduced compressed image using CWT in MATLAB simulation.



DWT ¼ coefficients only.
(PSNR = 27.6737 dB)

DWT ½ coefficients only.
(PSNR = 28.9856 dB)

DWT ¾ coefficients only.
(PSNR = 37.1123 dB)

All DWT subband coefficients.
(PSNR = ∞ dB, lossless)

Figure 3 Reproduced compressed image using DWT in MATLAB simulation.

Based on the results in Table 2, the DWT produces less amount of DWT wavelet coefficients as compared to the CWT wavelet coefficients. Even with the used of statistical analysis onto these CWT wavelet coefficients, there is a memory constraint issue for the low powered sensor nodes to store these coefficients, which needs to be resolved. For example, each of the Telos sensor nodes only has 10kB of RAM buffer. As a result, these sensor node will not be able to store large amount of data (2,228,244 coefficients/Bytes) while processing these coefficients. Even though performing inverse CWT transform with a complete set of CWT wavelet coefficients, the quality of reproduced compressed image is very poor. For the DWT technique, with only ¼ of coefficients, the reproduced compressed image achieves a better image quality compared to the CWT technique. Based on the experimental result, it can be concluded that the DWT is much more suitable for use in performing image compression at resource constrained WISNs.

2.2 REDUCED INSTRUCTION SET COMPUTER

Primary goal in computer design is to develop a much more cost-effective computer than their predecessor. However, this results architectural change with its trend going towards more and more complex machines were developed [43]. Therefore, the Complex Instruction Set Computer (CISC) was developed and less programme memory was required because implementing complex instructions in high-order language requires many words of main memory [44]. Next, a Reduced Instruction Set Computer (RISC) was also presented and it could be employed with little or no microcode. The RISC requires only simple instruction decode procedure that could be implemented by a fast combinational circuit [44]. Therefore, RISC is comparable in terms of performance and cost-effective to the CISC [43]. Besides, the complexity of CISC architecture will determine the size of the processor and the critical path in the machine that affects the speed of processor [45]. As such, the RISC tends to have simpler implementations and can be easily optimised in hardware and it operates at fast speed [45].

2.2.1 Ultimate Reduced Instruction Set Computer

In 1988, Farhad and Behrooz had presented a very simple processor, which is known as the Ultimate Reduced Instruction Set Computer (URISC) [16] [18]. The URISC processor is also known as the One Instruction Set Computer (OISC) and it is also considered to be as the penultimate Reduced Instruction Set Computer (RISC) [46] [44]. As oppose to OISC, the current available Complex Instruction Set Computer (CISC) has many complex instructions as micro programmes within the processor [18]. The advantages of the OISC is that its architecture is very simple since it is implemented using only single instructions [44]. The next advantage for using the OISC will be the instruction decoder circuitry and its complexity can be eliminated since all the instructions are the same. There are three different paradigms of OISC [47], these includes the Subtract and Branch if Negative (SBN) [48] [49], MOVE and Half Adder [50].

The MOVE processor is known to as a “practical” OISC that offloads much of its processing to a memory-mapped functional unit [46] [44]. The MOVE processor architecture is shown in Figure 4 [18]. The function of the MOVE processor is to have the Operand *A* move to Operand *B*. However, the actual computational is performed by its available underlying hardware.

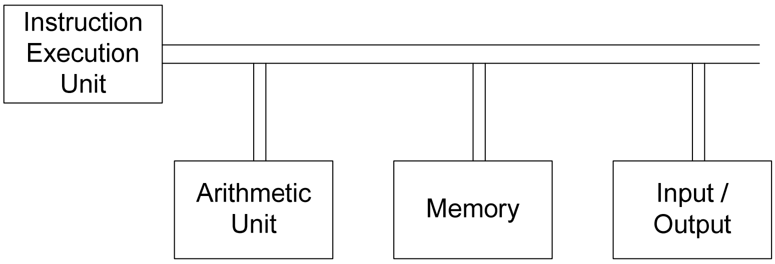


Figure 4 MOVE processor architecture [18].

Another simple OISC that was built previously with the use of the Half Adder (HA) as the basic instruction set [46] [44] [50]. For a Half Adder, it has all the program counter, general registers, special registers and memory location and they are all treated to be as one part of the address space. The Half Adder can also be used to implement the SBN instruction [46] [44]. The basic element of the Half Adder is shown in Figure 5. With the use of basic elements, it can then be connected together to form a mesh of connected Half Adder elements, as shown in Figure 6. From the architecture shown, it is suggested as a possible concept that can be used in the artificial intelligence and neural computing [44].

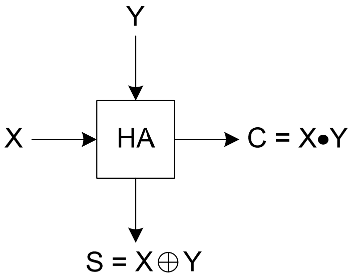


Figure 5 Basic Half Adder element [44].

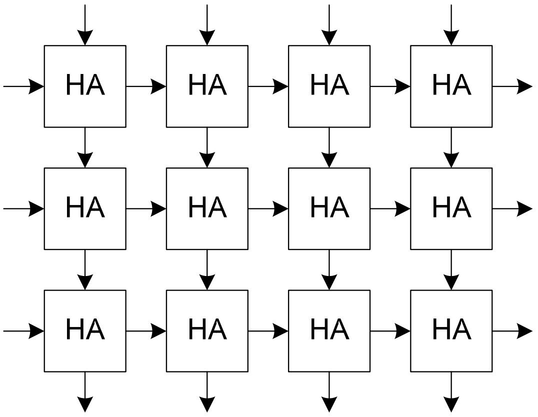


Figure 6 Mesh connected Half Adder elements [44].

A simple Subtract and Branch if Negative (SBN) processor, was initially presented by van der Poel in 1952 [48]. This type of processor will only execute single standard 3-address instruction that is shown in Figure 7. Although this SBN processor has only one standard instruction, it has the capability of executing a particular function with the use of a few simple SBN instructions. For this processor with only one standard instruction, the SBN instruction, the processor will execute the arithmetic operation of subtracting the 2nd operand (operand B) with the 1st operand (operand A). Then the result from the subtraction will be stored back to the 2nd operand ($B = B – A$) thus replacing its original values [16] [44].

SBN	***operand A***	***operand B***	***'jump target' address***

Figure 7 SBN instruction format.

If the result of the arithmetic operation is a negative value, the processor will then jump to another targeted address instruction that is stated in the program instruction, instead of executing the next following address instruction. In order to have the target address instruction to be executed by the processor for the next program instructions, the "jump target" address will be added to the current program counter (PC) register value. Consequently, the "jump target" address needs to be set such that the program counter will be set to execute the targeted address instruction [44].

2.2.2 Summary

The SBN architecture was modified such that the architecture can be expanded and further developed into RS MISC, CRS MISC and DWT CRS MISC. The SBN architecture was modified such that it allows additional functional blocks to be added into the architecture. By adding these functional blocks, the SBN was further developed into a MISC architecture which consists of a minimal number of instructions. The new DWT CRS MISC architecture specifically performs DWT image compression and CRS encoding in a single architecture.

2.3 WIRELESS VISUAL SENSOR NETWORK

In recent years, inexpensive hardware such as CMOS cameras are widely available in the market, which allows images to be captured from the environment in the WSN [15]. With this development, it gives rise to a new form of network known as Wireless Visual Sensor Networks (WVSNs). Each of these sensor nodes, it can process captured image data locally and extract the relevant information to be sent to the base-station. Besides, the sensor nodes are able to collaborate with other nodes on the application-specific task that is able to provide the user with information rich in description for a particular captured event [3]. The WVSN offers a wide range of applications, from remote and distributed video-based surveillance system to ambient assisted living and personal care applications. Users can remotely visit interesting locations through virtual reality using WVSN [5].

The WVSN offers many new applications compared to the WSN that uses only scalar sensors. Nevertheless, WVSN does face new problems such as a huge amount of data produced from the camera sensors [5]. Processing such large amount of data under constrained conditions, where there are limited amount of energy source, low bandwidth resources and limited processing power [3], is a challenge for generally low-powered sensor nodes. In the literature [7], it is stated that the amount of energy that can be used to process the data is much lower compared to the energy for use in transmitting the data across the wireless network. The transmitting cost (in terms of energy) of 1kb data is comparable to the same amount of energy use by a general-purpose processor that executes 3 million instructions [11]. As a result, large amount of data that is produced by the visual sensor nodes can be locally compressed, such that it reduces the large amount of data transmitting through the wireless network [9].

Since the sensor nodes have limited energy constraints and also limited processing power, the image compression techniques/encoders developed for use in WVSNs need to be low in power consumption [5]. Traditional image compression algorithms are not suitable for use in WVSNs [9] as they are mainly designed for multicasting/broadcasting applications [5], which is shown in Figure 8. The emphasis is to design a low-complexity decoder with the tradeoffs that the encoder bears the computational burden in the process of transmitting the information. However, for the

WVSNs, the complexity requirements are reversed due to their mostly many-to-one information flow.

For this research, the focus is on sensor node that captures still images and transfers the image data to the sink (base-station) across the resource-constrained WVSNs. The purpose of developing the visual sensor node is to provide surveillance for the military, especially to determine the number of enemy soldiers beyond the enemy line. Various existing WVSN platforms that were previously developed are reviewed in the following Section 2.2.1 to show the differences between these platforms and the developed joint image compression, encryption and error correction processing framework for WVSNs.

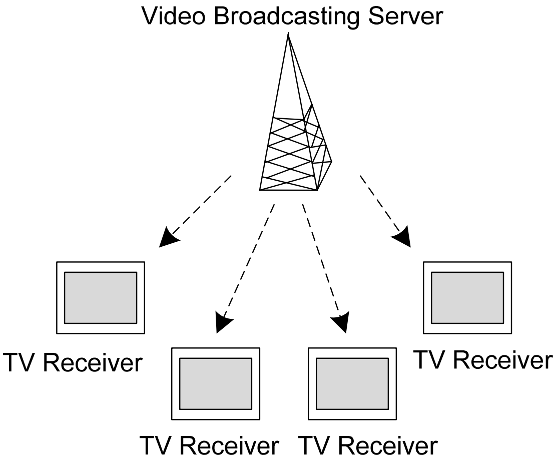


Figure 8 Information flows in traditional broadcasting application [5].

2.3.1 Existing WVSN Platforms

In [51], a wireless sensor device or “mote” known as Telos was introduced by the researchers from the University of California, Berkeley. The Telos was designed to sleep for major of time, wake up quickly on an event, process the information and return to sleep. The Telos platform was controlled using the Texas Instrument (TI) MSP430 microcontroller with 48kB of Programme Memory (FLASH memory) and 10kB of RAM buffer (SRAM). For the communication radio, the Telos uses the Chipcon CC2420 radio that operates in IEEE 802.15.4 standard [52] with transmission frequency of 2.4GHz. The Telos platform was developed such that it provides the capability to incorporate sensor into the platform. For the microcontroller unit to write data into flash memory, the power consumed by the Telos platform is 27.18mW at operating voltage of 1.8V [51]. However, the developed Telos platform was designed to be a basic WVSN platform that transmitted scalar data to the base-station without performing any data processing.

Later on in the literature [53], Cyclops platform was developed to perform hand posture recognition onto the captured images. Cyclops is an electronic interface that connects a camera module and a lightweight wireless host together. The Cyclops module is made up of a Xilinx XC2C256 CoolRunner Complex Programmable Logic Device (CPLD), 64kB of SRAM, 512kB of FLASH memory storage, an Atmel ATmega128L Micro-Controller Unit (MCU) running at 4MHz and an Agilent CMOS camera module (ADCM-1700). For the maximum power consumption, the worst case scenarios is considered when the Cyclops platform performs a write data operation with the permanent memory access. As reported, the maximum power consumed by the Cyclops platform operating at 3.0V is 64.8mW [53].

Next, an Intel Mote platform for industrial monitoring that measures vibration was developed [54]. The developed Intel Mote platform incorporates an Zeevo integrated wireless microcontroller module, industrial vibration sensor and a surface-mount 2.4GHz antenna together as a complete platform. The Zeevo module used for the Intel Mote consists of an ARM7TDMI architecture core, 64kB of SRAM, 512kB FLASH memory and a CMOS Bluetooth radio. TinyOS operating system was ported into this ARM architecture and this leaves about 11kB of free SRAM available to be used by any written applications in the platform [54].

In [55], the Panoptes video sensor platform was developed by integrating Intel StrongARM 206MHz embedded processor, a Logitech 3000 USB video camera, 64MB of memory, Linux 2.4.19 operating system kernel and an 802.11-based networking card together in a Bitsy board. For the proposed Panoptes video sensor platform, the power consumed by Computer Processing Unit (CPU) alone is 2.287W. The total power consumption required for Panoptes platform to capture, process and transmit the video is 5.268W. Since the proposed Panoptes platform is intended for use with a wind-powered generator that has unlimited energy source. Therefore, the high power consumption by the Panoptes platform is suitable in this applications [55].

The author in [56] had proposed an image sensor mote for use in Wireless Image Sensor Networks (WISNs). The microcontroller unit used in this sensor mote is the Atmel AT91SAM7S128 microcontroller based on the ARM7TDMI architecture core. The operating frequency of the microcontroller was set at 48MHz with available memory storages: 32kB of RAM and 128kB of Flash memory. Meanwhile, the proposed sensor mote used the Chipcon CC2420 based on IEEE 802.15.4 communication radio. The sensor mote operating voltage is at 3.3V and the power consumption for the microcontroller itself is 99mW. The developed image sensor mote was used to detect and determine the direction of pedestrian movement in a narrow pathway [56].

A CRITIC wireless camera mote was developed for the Heterogeneous Sensor Networks (HSNs) [57]. The CRITIC platform consists a 1.3 Megapixel OmniVision OV9655 CMOS sensor, Intel XScale PXA270 fixed-point processor (256kB internal SRAM), 64MB of external SRAM, 16MB Flash memory and Chipcon CC2420 radio. In the Idle mode (with no active processes), the power consumption of the Intel XScale processor was between 428mW - 478mW. The proposed CRITIC platform was developed to perform multiple target tracking and sending low amount of image data across the low bandwidth WSNs. This was done by processing the captured images locally on the camera board by using background subtraction for single target tracking and camera localization for multiple target tracking. Then transmitting only compressed low-dimensional image features to the sink (base-station), which routes the information to various clients for further processing and visualisation [57].

2.3.2 Summary

Most of the existing sensor nodes mentioned earlier were operating in the IEEE 802.15.4 standard that only consists of an error detection (CRC-16) [52]. In the IEEE 802.15.4 standard, there is no security protection is applied onto the data. The available data security protections is only applicable for the Waspmote (eg. Digi XBee, which is a proprietary Radio Frequency transceiver) that operates in IEEE 802.15.4 standard. As such, the CRS coding scheme was used in this research to provide data security. The CRS coding scheme has similar security level as the Advanced Encryption System (AES), which is the standard use for encrypting data by the US National Institute of Standards and Technology (NIST) [58]. At the same time, the CRS coding scheme also offers error protection capability as offered by traditional Reed Solomon coding scheme [59].

2.4 COMPRESSION IN WSN

Large amount of image data are produced by the visual sensor nodes and the captured information are then transmitted to the base station. However, transmitting these image data would increase the power usage in data transmission. Hence this would reduce the operating lifespan of the visual sensor nodes and interruption of image surveillances from that particular area. Therefore, many research were initiated to compress the image data before transmitting in the resource constrained WSNs such that to reduce the amount of energy used in data transmission. In this Section 2.3, the existing research works on the compression scheme for use in WSNs were reviewed.

2.4.1 S-LZW Compression for Energy-Constrained WSNs

A modified Lempel-Ziv-Welch (LZW) [60] lossless compression algorithm for Sensor Nodes (S-LZW) was presented to reduce energy consumption of sensor nodes [12]. The proposed S-LZW compresses data with the use of a dictionary of 512 entries which requires 2,618Bytes of RAM and 1,264Bytes of ROM. Besides, the author also proposed a 32-entry mini-cache version of S-LZW (S-LZW-MC32) that requires 2,687Bytes of RAM and 1,566Bytes of ROM. The propose S-LZW compression algorithm is used to compress strings of data in 1-dimensional rather than consider it as image data in 2-dimensional [12].

2.4.2 Lapped Biorthognal Transform for WSNs

In [61], a distributed Lapped Biorthogonal Transform (LBT) based image compression is introduced for WSN. First, the image is captured by the camera node which later send a message to cluster head through its neighbour *S*. The Cluster head selects the idle nodes (above a threshold energy level) within the cluster and requests the camera node to send the image data. Then neighbour *S* perform LBT pre-processing on 8 rows of image data received from camera node. After LBT pre-processing, neighbour *S* sends these processed image data to idle node *P*. Finally, node *P* sends the processed compressed image data to the Cluster head.

The performance of proposed scheme is evaluated by using MATLAB [61]. The proposed approach does overcome the computation and energy limitation of individual nodes, by sharing the processing tasks among the nodes in the cluster [61]. However, the high frequency of transmitting and receiving data among the nodes may reduce the lifespan of the nodes since RF transceiver module is the device with largest energy consumption [62] [63].

2.4.3 SPHIT MIPS Processor for WVSNs

The Set Partitioning In Hierarchical Trees (SPIHT) encoder can be developed using a Microprocessor without Interlocked Pipeline Stages (MIPS) processor that has a very low memory wavelet compression architecture using strip-based processing introduced in [64] [65]. The process of SPIHT image compression consists of a few separate modules. Figure 9 illustrates the first few lines of the image which are loaded into the Discrete Wavelet Transform (DWT) module to perform wavelet transform onto the image data. Then the computed wavelet coefficients are stored into a strip-buffer that will be used for SPIHT encoding in the later part. Once the image is fully encoded by the SPIHT encoder, the bit-stream generated will be output and then can be transmitted across the communication channel.

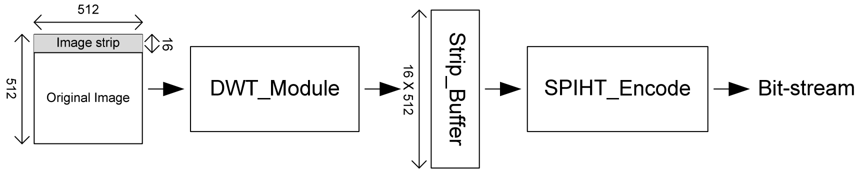


Figure 9 Block diagram of strip-based compression [64].

Before the image data is passed through the SPIHT encoder module, a four-scale DWT decomposition is applied onto an image of size pixels [64] [65]. Figure 10 illustrates the DWT_Module used in applying the DWT decomposition onto the input images. Initially, the image data are read into the DWT_Module in a row-by-row order from the external memory. After which the row filtering is performed on the image row and the coefficients are stored into a temporary buffer (Temp_Buffer). With these four lines of row-filtered coefficients available, the column filtering is then carried out onto the row filtered coefficients. Finally, these DWT coefficients HH, HL, LH and LL are stored into the STRIP_BUFFER [64] [65]. For a particular N-scale of DWT decomposition, the LL coefficients generated from $N-1$ stage will then be loaded back to the Temp_Buffer from STRIP_BUFFER. Then a further N-scale of

DWT decomposition is performed onto these LL coefficients. As the produced wavelet coefficients are arranged in a pyramidal structure in the STRIP_BUFFER, the SPIHT_ZTR coding is implemented using a one-pass upward scanning and a one/multipass downward scanning methodology [64] [65]. Figure 11 illustrates the architecture of the SPIHT encoder implemented in the SPIHT MIPS processor. The strip-based SPIHT-ZTR architecture is implemented using a soft-core microprocessor based approach, where a customized MIPS processor architecture is adopted. The actual implementation is onto a Xilinx Spartan-3L 3S1500L FPGA and it requires a total of 2,366 slices (1,272 flip-flops and 3,416 LUTs) [65].

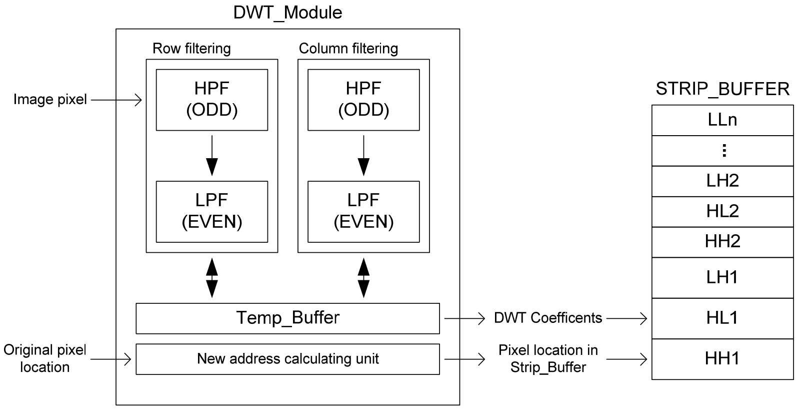


Figure 10 DWT_Module architecture [65].

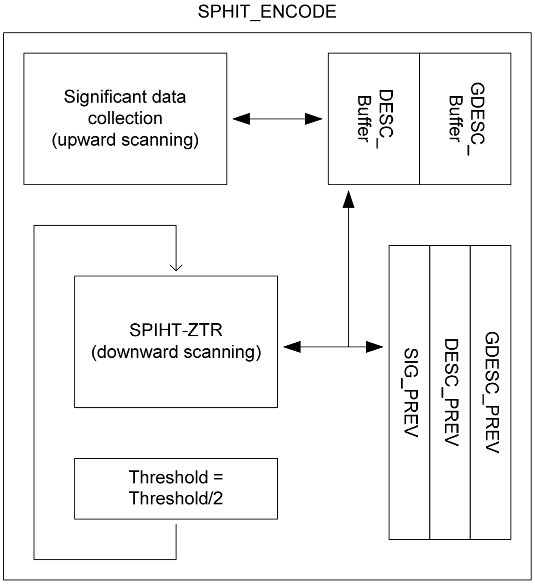


Figure 11 SPIHT_Encoder architecture [65].

2.4.4 JPEG FPGA-Based Wireless Vision Sensor Node

A prototype of the vision sensor node is developed, which consists of CMOS image sensor, Field-Programmable Gate Array (FPGA), Nios II soft-core microprocessors and nRF24L01 transceiver [66]. The vision sensor node adopts Joint Photographic Experts Group (JPEG) Baseline system for image compression using the Altera EP2C35 FPGA development platform. Then the Nios II microprocessor controls the nRF24L01 transceiver to transmit the compressed image data to the base-station (sink). For the developed JPEG system, the system requires 7,173 Logic Elements (LEs) as mentioned in [66].

2.4.5 Low Power Wavelet Transform for WSNs

In [67], the authors proposed to use the fractional wavelet for WSNs. As mentioned in the tutorial, a sensor node was built using a 16-bit Microchip dsPIC30F45013 digital signal microcontroller with 2 kB of RAM. Besides, a C328-7640 camera module with Universal Asynchronous Receiver/Transmitter (UART) was also integrated. Study made on the proposed system by having the sensor node to capture 8 frames of 256 x 256 pixels of images. Then performs a six-level fractional wavelet transform that involves the 16-bit integer arithmetic. The study concluded that fractional wavelet transform for integer arithmetic requires half of the time needed for the floating point case [67].

2.4.6 DWT Selective Retransmission for Wireless Image Sensor Networks

At the sensor node, Discrete Wavelet Transform (DWT) decomposes the captured still image data into subbands of DWT coefficients. These produced DWT coefficients have different relevancies in reconstruction of original image [68]. By knowing the different priority level of DWT coefficients, a DWT-based selective retransmission mechanism is proposed for Wireless Image Sensor Networks (WISNs). Depending on the application requirement, reliable transmission is only assured for the most relevant data, with providing retransmission of corrupted data; while the low relevant data are not retransmitted, if they are corrupted during transmission. Hop-by-hop retransmission of corrupted compressed image data is performed rather than end-to-end approaches to increase energy saving on nodes. For the purposed mechanism, comprehensive energy consumption models were designed and extensive mathematical verifications were performed [68]. However, this approach of retransmission may reduces the lifespan of these intermediate nodes (between end nodes and sink) since they consume more energy for retransmission of image data in noisy communication channel [62].

2.4.7 CL-DCT for Wireless Camera Sensor Networks

In [25], the Cordic Loeffler Discrete Cosine Transform (CL-DCT) compression processor was introduced and implemented onto FPGA. The CL-DCT processor was designed to perform compression on captured image data for the WVSNs. The CL-DCT processor was implemented onto the Xilinx Spartan-3 XC3S200 FPGA, which requires a total of 2,385 Logic Cells (1,060 Slices). However, the authors did not consider the external SRAM memory used by the proposed CL-DCT processor as part of the hardware implementation [25].

2.4.8 Summary

From the existing compression techniques, a huge amount of hardware utilisations were required for implementing the image compression schemes in the WSNs. The reason is that these existing image compression techniques required complex algorithm/architecture in order to perform image compression. The Lifting Scheme DWT image compression was used in this research because this method is much simpler compared to other compression. With simple compression techniques, the reuse of hardware (MISC architecture) to encode the compressed image data was made possible. This would lead to improved efficiency and less hardware usage [21]. Besides, there is a reduction in the amount of image data required to be transmitted across the WVSNs. Longer operating lifespan of the sensor node could be achieved when lower data transmission energy is required for sending these reduced amount of image data. Therefore, this research was performed in order to develop an image compression processing framework for use in the resource constrained WVSNs.

2.5 FORWARD ERROR CORRECTION IN WSN

The error control system is an important area in communication to maintain the reliability and integrity of the data transmitted across a communication channel [69]. While the data is transmitted across the communication channel, it is prone to error due to interferences that often occur in the channel. For some applications, it requires high data integrity to be transmitted across the communication channel and to be received at the receiver, for example satellite communications [70]. With the use of error correction code, the receiver can correct the errors that occurred on the received data. Therefore, the receiver requests less number of data retransmission that resulted in less data transmission performed by the sensor node. Under normal situation, error controls coding are applied to the data before it is transmitted [71]. After the data is received at the receiver, the receiver performs error correction on incorrect received data to retrieve the correct data.

Error correction methods are divided into two different areas. The first correction method is the Automatic Request for Retransmission (ARQ) [72]. Whereas the second correction method is the Forward Error Correction (FEC) [73]. For ARQ technique, the receiver detects any error on the received message. If there is any error on the received message, then the receiver requests for retransmission of correct message from the transmitter again. Whereas the FEC coding scheme, the transmitter side encodes the message to produce additional data (redundant bits) and add these data onto the message. A complete codeword is formed by adding the redundant bits to the original message. Then the codeword are transmitted across through the communication channel. With codeword received, the receiver can detect any error and correct the error using the available redundant bits to reproduce the correct message.

There are two common codes that are still in use, which include the block codes and the convolution codes [17] [69] [74]. The block codes encode the input message in a fixed block size of k information bits (symbols) for each codeword. The block codes will then produce n symbols of output codeword with respect to the corresponding k input message symbols. Each of the input messages is encoded independently and it is not related to the previously produced codewords. Therefore, the encoder is memoryless and this allows the implementation of the encoder in the form of combination logic circuit as mentioned in [75]. Examples of the block coding are the Reed Solomon [76] and Hamming code [17] [74]. For the convolutional codes, the message are encoded depending on the corresponding *k*-bit of message block and *m* previous message blocks. Hence an *m* order of memory space is required for the encoder. As a result, the convolutional codes encoder will be implemented in the form of sequential logic circuit. Examples of a convolutional codes are the Odenwalter code [69] and Self-Orthogonal Codes [17].

Initially, the input message for block coding is divided into blocks, with each block in *k*-bit that is known as datawords (message) [69]. Then it is followed by $2t$ redundant bits that will be added to each corresponding message block to form *n*-bit of codeword. Equation (1) shows the relationship between *n* codeword, $2t$ redundant bits and *k* datawords.

$$\text{Codeword length,}\quad n = k + 2t \tag{1}$$

In error corrections, the Hamming distance between the two words is the measure of differences between the corresponding bits in these words [17] [74]. The Hamming distance can be determined through applying the XOR operation between these two words. Next, the minimum Hamming distance is determined through finding the smallest Hamming distance between the whole words. The notation for the minimum Hamming distance is d_{min}. The Hamming distance is related to the number of error bits that occurs in a particular message word. For example, a codeword in the form of 10101 is sent and the receiver received the codeword as 00111 [77]. As a result, there will be 2 error bits occurring on the received codeword and the Hamming distance will then be 3.

In order to have a block code that is capable of detecting up to *e* number of errors, the minimum Hamming distance for the block code must be, $d_{min} = e + 1$. For the block code to be able to correct up to *t* corrupted bits codeword, the minimum Hamming distance for the block code would be, $d_{min} = 2t + 1$ [69]. This ensures that the block code can correct up to *t*-bit of errors that occurred on the codeword. As shown in Equation (4), the coderate, *r* for a block code is a measure of how efficient of a block code in protecting the codewords [17].

$$\text{Hamming distance,} \quad d = e + 1 \tag{2}$$

$$\text{Minimum Hamming distance,} \quad d_{min} = 2t + 1 \tag{3}$$

$$\text{Coderate,} \quad r = \frac{k}{n} \tag{4}$$

The following sections review on the existing research works that are related to Forward Error Correction (FEC) coding schemes in WSNs.

2.5.1 Old-Weight-Column Code in Wireless Sensor Network

In [78], the Old-Weight-Column code that can correct Single-bit Error and Detect Double-bit Errors (SECDED) was implemented onto the Mica2dot sensor node with ChipCon CC1000 radio. Two different coding schemes were implemented, where the first one was Old-Weight-Column code with 8-bit data and 13-bit codeword (SECDEC (13,8)). The second coding scheme implemented was Old-Weight-Column code with 24-bit data and 30-bit codeword (SECDEC(24,30)). Besides the Old-Weight-Column code, a (16,8) systematic quasi-cyclic coding scheme was also implemented. The implemented (16,8) systematic quasi-cyclic coding scheme (DECTED (16,8)) can correct up to 2-bit errors and detect 3-bit errors [78].

With the use of Error-Correction Code (ECC), it helps to reduce the packet drop rate for both outdoors and indoors transmission tests performed [78]. From these ECC implementations, the SECDEC(13,8) produced the smallest packet drop rate as compared to the SECDEC(30,28). The reason is the SECDEC(30,28) had large amount of data (bits) in one packet with 1-bit error correction capability only thus having weaker error-correction capability as compared to SECDEC(13,8). As for the double-bit error correction coding scheme, the DECTED (16,8) is not that efficient than SECDEC(13,8) since most errors encountered are single-bit or multiple-bit errors [78].

With this ECC implementations, they are effective on bit error rate that is not high and most errors occurred were single bit for each packet of data transferred [78]. Due to constraints of low power consumption and small form factor, the error-correction codes have been designed to be simple. Therefore, these codes could only correct single-bit or double-bit errors. When most errors are burst errors, then these codes would not be able to reduce packet losses effectively. As a result, error correction codes that could correct more than double-bit errors are needed for such situations. However, these error correction schemes were likely to be computationally complex and required intensive processing with large amount of memory storage [78].

2.5.2 Reed Solomon Code in WSNs

In 1960, the Reed Solomon (RS) coding scheme was presented by Irving Reed and Gus Solomon [76] [79]. The discovery of Reed Solomon leads to broad range of applications, such as digital television, wireless communications, broadband, compact disc (CD) players, satellite communications etc [80]. Reed Solomon code is considered as a sub class of the Bose, Chaudhuri, and Hocquenghem (BCH) codes [17]. As compared to Hamming code, Reed Solomon has the capability in encoding more data since it can encode *k* number of symbols, with each discrete symbol is an *m*-bit of message [81].

For an (n, k) Reed Solomon (RS) code, the encoder will take in *k* information symbols and generates $2t = n - k$ of redundant symbols [82]. The redundant symbols produced by the RS encoder, are also known as the parity symbols that will be used in RS decoding. Having 2*t* redundant symbols, the encoder can combine both parity symbols and information symbols to produce one block of codeword, with *n* number of symbols for each codeword. With this configuration, the minimum distance of the Reed Solomon code will be $d_{\min} = 2t + 1$. The breakdown of one block of codeword produced by the Reed Solomon encoder is shown in Figure 12.

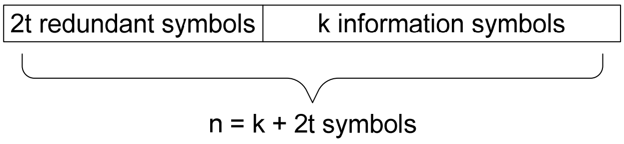


Figure 12 One codeword of Reed Solomon.

For RS coding scheme, the encoding of the input message involves the Galois Field arithmetic operations in generating the parity symbols. The Galois Field arithmetic operations are also involved in decoding the received RS codeword by the RS decoder. Consider that α to be a primitive element in the finite field $GF(2^m)$ with symbols of α, α^2, ... , α^{2t-1}, α^{2t} are from the field $GF(2^m)$, the generator polynomial $g(x)$ for RS(n,k) coding scheme for *t* number of error correcting capability is shown in

Equation (5) [17]. Using the generator polynomial *g*(*x*), the message symbols *u*(*x*) is multiplied with x^{n-k} that shifts the message symbols into the rightmost *k* stages of a codeword. Then the message symbols are divided by the generator polynomial *g*(*x*) through the use of Galois Field arithmetic operations. In Equation (6), the remainder of the division operations produces 2*t* parity symbols *p*(*x*). After which the final codeword will be produced by having the message symbols added with the parity symbols, as shown in Equation (7).

$$\textit{generator polynomial,} \quad g(x) = (x-\alpha)(x-\alpha^2)......(x-\alpha^{2t-1})(x-\alpha^{2t}) \\ = g_0 + g_1x + g_2x^2 + ...+ g_{2t-1}x^{2t-1} + x^{2t} \qquad (5)$$

$$\textit{parity symbols,} \quad p(x) = x^{n-k}u(x) \bmod g(x) \qquad (6)$$

$$\textit{codeword generated,} \quad v(x) = p(x) + x^{n-k}u(x) \\ = x^{n-k}u(x) \bmod g(x) + X^{n-k}u(x) \qquad (7)$$

In hardware implementation, the parity symbols are computed through the use of a division circuit, known as the Linear Feedback Shift Register (LFSR) circuit [17] [83] [84]. Figure 13 represents the arrangement of the generator polynomial coefficients in the LFSR circuit. For the LFSR circuit, the message symbols will be clocked into the circuit sequentially to all the 2*t* registers at the input message symbols. While the message symbols are clocked in, Switch 1 is closed with Switch 2 is set to the input message symbols position. This will allow the *k* message symbols to be input into the registers and at the same time with the entire message symbols are clocked out as part of the codeword. After which the entire message symbols are input, the Switch 1 will be set to open and Switch 2 will be set to the output of LFSR circuit. This will allow the 2*t* parity symbols to be output to form one block of complete codeword. As the Switch 1 is opened, the parity symbols that are shifted from one register to another would be affected. Consequently, the LFSR circuit requires a total of *n* clock cycles to generate one block of complete codeword for *k* message symbols and 2*t* parity symbols.

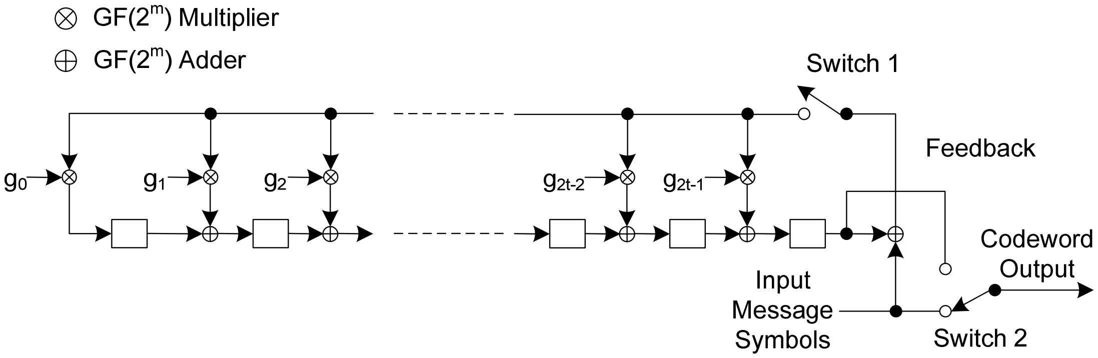


Figure 13 RS(n,k) encoder in LFSR circuit configuration [17].

Beside the Old-Weight-Column Code was used in the WSNs, the RS coding scheme was also used in WSNs to provide data reliability. For example in [85], RS coding scheme [76] was implemented on the WSNs using the Mica2Dot motes. The RS(29,8) coding scheme was then used in such implementation. It required 512Bytes for operation table, 64Bytes for matrix, 232Bytes for 8 packet buffers, 68Bytes for erasure code component and 4Bytes for other variables. Such implementation requires a larger amount of memory storage to store these data in the processor available at the motes [85].

In another literature [86], the effect of using different Forward Error Correction (FEC) codes for the developed μAMPS sensor node was studied. The μAMPS sensor node was made up of Intel StrongARM 1100 processor and RFM-TR1000 radio module that operates at 916.5MHz. The FEC codes used in this study were the RS, Viterbi and Bose, Chaudhuri and Hoquenghem (BCH) coding schemes. For this study, the energy consumption by different FEC codes decoding algorithms were determined by using the JouleTrack. By using JouleTrack, the RS coding scheme was found to be the most energy efficient for used in the WSNs as compared to the other FECs [86].

In simulation study [81], the Bit Error Rate (BER) performance of a few different FEC codes for WSNs were considered, which included Hamming, Golay, Convolution and RS coding schemes. Among these FEC codes, the RS coding scheme was considered to be the best choice for WSNs since it outperformed the others. With RS code considered, the power consumption for different of the RS coding schemes were studied. The RS(31,21) coding scheme was found to give the lowest power consumption in comparison with the other coderates of RS coding scheme. Therefore, the RS(31,21) was considered to be the optimal choice of ECC for Wireless Sensor Network applications [81].

In [87], the Reed Solomon channel code was performed on WSNs by using software implementations. In this study, the CC2530 Texas Instrument sensor nodes (motes) were used. The use of soft implementations of Reed Solomon coding scheme on the transmitted data required MATLAB program that was run on a PC workstation. Such implementation requires a PC to perform RS encoding before the image data transmit across the WSNs. It would not be feasible for actual WSNs deployment since a PC could not be put together to be used with a sensor node that has limited energy source [87].

2.5.3 Turbo Codes in WSNs

In the simulation study [88], powerful soft-decision decoding algorithm such as Turbo Code coding scheme [89] [90] was considered for hard-detected signal in the WSNs. For a simplified wireless multi-hop sensor network channel model (Figure 14), the channel could be assumed to be a chain of symmetric Binary-Input and Binary-Output (BIBO) channel. Usually, the intermediate nodes perform the regenerative repeating process such as error correction, detection and re-encoding. If no error is detected, the data are sent to the next intermediate node until to the Central Station (sink). If error is detected, the node requests retransmission of the data again. However, these processes extend latency but reliable relay data transmission are guaranteed [88].

The author presented a channel model without error correction and detection processes in the intermediate nodes [88]. Therefore, these intermediate notes only have regenerative repeating process that performed routing and relayed the data to the Central Station, which is shown in Figure 15. For the Central Station (destination), a powerful soft-decision decoding (Turbo Code decoding) was performed onto the received hard-detected signal, as shown in Figure 16. The simplified system model presented by the authors did show that the error-detected reliability had slight improvement than the conventional turbo decoding scheme with hard-decision at low range of SNR. As for the moderate to high range of SNR, the simplified system model had shown a 2.0dB improvement in BER performance [88].

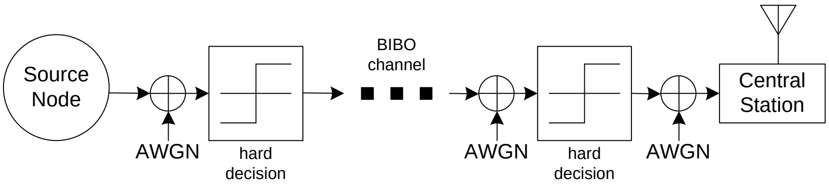


Figure 14 Simplified multi-hop channel model of WSN [88].

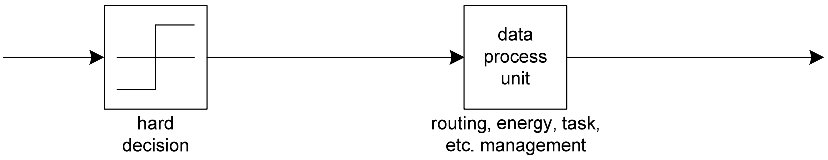


Figure 15 Regenerative repeating process at intermediate node [88].

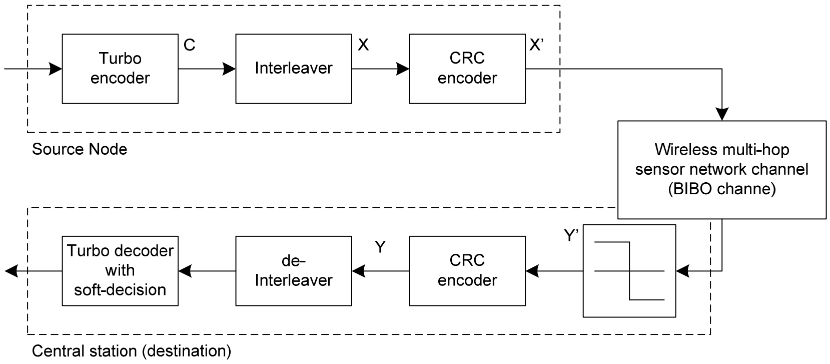


Figure 16 Simplified system model without regenerative repeating process [88].

In another literature [91], a MATLAB simulation on the Parallel Concatenated Convolutional Code (PCCC) Turbo Code configuration was studied for WSNs. The design of the PCCC Turbo Code technique was built with coding rate of 1/3 for performance and systematic feature. The approach used the PCCC encoder circuit at the source node to encode data packets. Then the PCCC decoder circuits on the routing nodes were shifted to the base station such that the decoding process was performed at unlimited energy resources platform. Such method is much more energy efficient and less processing time required on the routing nodes. MATLAB simulation was performed for a one dimensional sensor network with a source node transmitting Turbo Coded data over a multi-hops network to the base-station. The results from the simulation were very promising with increased in Bit Error Rate performance [91].

In another simulation study [92], a new algorithm for distributed encoding and decoding of Turbo Code for a heterogeneous WSNs was presented. The proposed algorithm distributes parallel concatenation of multiple convolutional codes encoder and iterative decoder structures into various sensor nodes. With this algorithm, larger coding gains were able to be achieved by using the parallel concatenation of multiple convolutional codes. In the mean time, it also provided reliable communication over noisy channel. The simulation results had shown the practicality of using the high performance of distributed encoding and decoding of Turbo Codes in sensor nodes with limited computational resources [91].

In [93], a Xilinx CoolRunner-II Complex Programmable Logic Device (CPLD) Turbo Code encoder was developed for WSNs. The developed CPLD Turbo Code encoder was coupled with Crossbow’s MICAz to form a complete sensor node. When processing and transmission of data, the use of CPLD Turbo Code encoder (hardware-based) had led to a 40.7% decrease in energy consumption compared to the software-based Turbo Code encoder. At coderate of 1/3, the processing/transmission time for each packet (480bits) of hardware-based Turbo Code encoded data required 12.682ms. Meanwhile, there was a significant amount of reduction in message retransmissions when using the Turbo Coding compared to uncoded data transmission in noisy communication channel. As such, the reduced in message retransmission also lead to a reduction of 44% overall energy consumption of the sensor node [93].

2.5.4 Cauchy Reed Solomon in WSNs

A reliable data transmission for metal fill monitoring with the use of WSNs was introduced [94]. The literature uses the Cauchy Reed Solomon (CRS) erasure coding scheme to recover correct data. This reduces the amount of retransmission thus increase the network bandwidth with more packets transmitted by each mote. Cauchy matrix was used to perform the systematic Reed Solomon encoding onto the sensed information (data). Systematic Reed Solomon encoding still shows the message (original data) in the encoded codeword that does not give any security protection against eavesdropping [94].

2.5.5 Hybrid ARQ/FEC Error Control in WSNs

The simulation study in [62] showed that the use of both convolutional codes for error correction and Automatic Request Retransmission (ARQ) in WSNs can significantly increase the node and network lifetime. There are situations where the link error rates among different pairs of network nodes are different. Since there are some nodes that have over dimensioned error control schemes, this generates a reduced network energy efficiency which is not desirable. Therefore, a Hybrid ARQ/FEC (HARQ) protocol was proposed to recover errors before requesting a retransmission. This is done by optimising the code complexity to be employed along each transmission hop. The memory order is also optimised depending on the average received Signal to Noise Ration (SNR) value. With the use of the optimised complexity, the overall energy consumption by the local node processing and the HARQ retransmission protocol is reduced [62].

In [95], a simulation study was carried out on the use of hybrid ARQ/RS scheme to enhance the quality of service for multimedia content (eg. video) over multi-hop WSNs. In this study, a few different RS coding schemes were used and combined with ARQ to give a hybrid ARQ/RS scheme. Based on the MICAz video sensor nodes (motes) platform, the simulation study was performed by using ns-2 network simulator and along with video quality evaluation tool, Evalvid. Based on a perceived video quality and frame loss rate, the results showed that hybrid ARQ/RS scheme outperformed the individual RS and ARQ schemes [95].

2.5.6 Hamming Code in WSNs

Hamming code [83] was considered to be the first class of linear block codes for use in error correction [17]. For Hamming code, with a minimum Hamming distance of d_{min} = 3, it is capable to correct up to single error bit that occurs over a length of *n* codeword [77]. In the meantime, the Hamming code can detect up to a total of 2 error bits for a particular codeword. Details on the method of encoding and decoding the codeword can be referred to [17].

Hamming coding [83] is one of the most commonly used techniques in sensor networks, to guarantee data integrity at the sink node [96]. Two different Hamming coding schemes were tested and compared with the CRC coding scheme in [96]. The Hamming coding schemes implemented were (63, 57) Hamming code and (7, 4) Hamming code. As for comparison, the 12-bit Cyclic Redundancy Check (CRC) coding scheme was used to compare with the Hamming codes.

A MATLAB simulation model was built to study on the effect of using error correction coding schemes [96]. With the Additive White Gaussian Noise (AWGN) channel, the effectiveness of these coding schemes was compared by measuring its

Information Per Joule (IPJ). This measured the compromise between the data throughput and the lifetime of the sensor node. From the MATLAB simulation model, the Hamming code has a very low effect on sensor network lifetime. Whereas, the CRC code has a significant effect on the lifetime of the sensor network, which was a reduction of 37.3% as compared to an uncoded system. This is due to CRC's high processing energy required at high Signal to Noise Ratio (SNR), where no retransmission was needed [96].

As for the long (63,57) Hamming code, it was found that the code had given the lowest SNR and the highest Energy per Binary Operations, E_{oper} for higher IPJ [96]. In comparison to the IPJ for both CRC and (7,4) Hamming code at highest SNR and lowest E_{oper}, the IPJ for the long Hamming code at the worst conditions outperforms both codes at its best condition. Therefore, long (63,57) Hamming code is much more suitable in sensor networks than that of the (7,4) Hamming code for implementation [96].

2.5.7 Error Concealment for Robust Image Transmission over WSNs

The watermarking based Error Concealment (EC) approach [97] was used to provide a robust image transmission over WSNs [63]. For this simulation study, DWT is performed onto the image data to produce DWT wavelet coefficients and also a minimised replica of the image (LL_2). These minimised replica is then embedded into the macro-blocks of subbands (LH_1, HL_1, HH_1). After which, inverse DWT is performed onto the watermarked image in wavelet domain. The EC encoding phase produces watermarked image without changing the size of original image. Without reducing the image data size, the whole reproduced image is then transmitted to the base-station [63].

2.5.8 LDPC Coding in WSNs

For LDPC error correction, a custom designed processor that performed data encoding in bits was implemented onto the FPGA [98]. The LDPC encoder/processor was developed onto a Xilinx Vertex-II XC2V6000-6 FPGA, having four different configurations. The hardware utilisation for 1 encoder instances was 870 slices and 19 Block RAMs. For the case with 16 encoder instances, the LDPC encoding was performed onto the data with block length of 2,000 bits at a code rate of ½ and operating frequency of 82MHz. In this 16 encoder instances, this implementation required a large amount of hardware utilisation required, which required a total of 16906 slices and 107 Block RAMs. Furthermore, half of the encoded block length of the codeword consisted of only the data to be transmitted. Therefore, through the use of the LDPC processor, the sensor nodes have to transmit a large amount of redundant data and it is considered to be not efficient for coding in WVSNs [98].

In [99], simulation studies on the proposed combined Multiple Input Multiple Output (MIMO) and Low-Density Parity-Check (LDPC) codes [100] system for WSNs was presented. Multiple sensor nodes were used to form the MIMO system that allows higher data transmission rate between the master unit (base-station) and the sensor nodes. To further enhance the data transmission rate, a (128,256) Non-Systematic LDPC code was used in the proposed system. From the MATLAB simulation models, the results showed that good BER performance was maintained with the used the LDPC code. As such, the use of LDPC code did improve on the data transmission and reduce the number of retransmission required which would relatively reduce the transceiver transmission power [99].

2.5.9 Summary

Eight different Forward Error Correction (FEC) coding schemes were discussed, which include the Old-Weight-Column Code, Hamming Code, RS Code, Turbo Code, CRS Code, LDPC Code and BCH Code. From these reviewed literatures, most of the FEC codes proposed for use in the WSNs were simulation studies. A few research works were implemented onto actual sensor nodes that were equipped with a sensor, a microcontroller and a Radio Frequency (RF) transceiver.

For the Old-Weight-Column coding scheme, the presented SECDEC(13,8) had the capabilities of correcting only a single bit error. The SECDEC(13,8) is suitable for use in low data transmission, for example WSNs. This is not suitable for WVSNs because it is not efficient in correcting single error bit for large amount of image data that may have more than 1 error bit. As for Hamming code, it also could correct a single bit error which is not efficient in providing error protection for large amount of data. Turbo Code, LDPC Code and BCH Code do provide an acceptable level of error protections but they need complex coding scheme in order to be implemented onto the WSNs. Complex architecture of the coding scheme would require a large amount of hardware area and thus increases the power consumption.

The block coding method, particularly Reed Solomon coding is considered for this research. The reason is that the RS coding schemes can correct more number of errors (in Bytes) in one codeword. Whereas, Hamming code only can correct 1-bit of error in one codeword. As such, the RS code is very effective in correcting random errors and random burst errors [17]. Hence the RS code is considered to be efficient coding schemes as compared with other block coding schemes [86] [81]. As a result, Reed Solomon coding scheme is preferred to be implemented on the WSN to provide data integrity. Meanwhile, the RS coding scheme does not provide any protection against security threats when the data are transmitted across the wireless communication channel.

2.6 ENCRYPTION IN WSN

Transmitting captured information securely in a battlefield is the upmost importance such that these information would not be known when they are eavesdropped by the enemy. Therefore, the research on security in data transmission for WSNs were one of the mainly focused area. Many research works were available in the literature on this research area. In this Section 2.5, some of the existing research works that were related to the security scheme in the WSNs were reviewed.

2.6.1 SPINS: Security Protocols for Sensor Networks

A set of Security Protocols for Sensor Networks (SPINS) was presented in the literature [101]. The SPINS security protocol run on two secure building blocks, which are the Secure Network Encryption Protocol (SNEP) and the μTesla. The SNEP offered data confidentiality, two-party data authentication and data freshness. Whereas, the Tesla provided authenticated streaming broadcasting. In this literature, the SPINS was implemented onto SmartDust prototype nodes, which consists of a 916MHz communication radio, 8kB instruction FLASH memory, 512Bytes of RAM, 512Bytes of EEPROM and an 4MHz 8-bit processor. The nodes were run on TinyOs operating system that consumed almost half of the instruction FLASH memory. Therefore, it only had 4500Bytes of available memory for security application. With the resources constraints, the implementation of the μTesla was achieved with 2kB of programme memory and 120Bytes of RAM [101].

2.6.2 TinySec Security Architecture for WSNs

In year 2004, a TinySec security architecture, which is the first fully implemented link layer security suite, was developed for WSN [102]. The TinySec was implemented onto a few different platforms, which are the Mica, Mica2 and Mica2Dot nodes. Each of these nodes consists of Atmel processors. While Mica sensor node uses the RFM TR1000 radio, both the Mica2 and Mica2Dot sensor nodes use the Chipcon CC1000 radio. Besides, the TinySec was ported to a Texas Instruments microprocessor. The TinySec was written in 3000 lines of NesC code, which is the programming language for TinyOS. Such implementation required a total of 728Bytes of RAM and 7,146Bytes of program space. It can be seen that the TinySec required large amount of memory (hardware) and heavy processing while the data were encrypted thus a 10% increase in power consumption as compared to without having TinySec [102].

2.6.3 Advanced Encryption Standard for WSNs

In [103], an Advanced Encryption Standard AES-128 Block Cipher was developed that performs the AES encryption. The proposed design for the AES-128 Block Cipher was optimized for low die size and low power consumption. The AES-128 Block Cipher was developed onto a 0.35μm CMOS technology chip and occupied an area of 3,400 GEs (Gate Equivalents). The power consumed by the developed AES chip was 5mW when the chip was operating at 100kHz and the operating voltage at 1.5V [103].

In [104], a simulation study to develop secure Advanced Encryption Standard (AES) coprocessors with four different S-Box configurations for WSNs were performed. All these configurations were implemented with the use of United Microelectronics Corporation (UMC) 0.25μm 1.8V technology chip library from Synopsys Design Compiler. First method of applying the AES was by constructing the S-Box circuits directly by using combinatorial Look-Up-Table (LUT), which required 573 gates count. Second method of implementing the AES was by using the Decode-Switch-Encoder (DSE) S-Box which needed 780 gates count. The third method was by multiplicative inverse in composite field GF(2^4) that needed 373 gates count. The last method of implementing the AES was by using the improved power-efficient Galois Field with Positive Polarity Reed-Muller required a total of 577 gates count. From all the configurations of S-Box, the GF AES is considered to have the least hardware utilisations in performing AES encryption [104].

In [105], AES encryption scheme was implemented onto the WSNs with the use of Texas Instrument MSP430 processor and Chipcon CC2420 ZigBee radio. The written C code for AES encryption-decryption was optimized such that to match the communication speed of the ZigBee radio. The software implementations of the AES encryption scheme for the MSP430 processor required a total of 260Bytes of RAM and 5,160Bytes of ROM [105].

2.6.4 HIGHT Block Cipher for Low-Resource Device

In the year 2006, a new block cipher known as HIGHT was implemented in [106]. The authors had presented HIGHT block cipher with 64-bit block length and 128-bit key length. The block cipher was designed for low-resource environment such as RFID tag and other tiny ubiquitous devices. This block cipher requires a total of 3048 gates (435 flip-flops) with a throughput of 150.6 Mbps, operating under the frequency of 80MHz. Each HIGHT encryption performed by the designed block cipher would require 34 clock cycles [106].

Later, the HIGHT encryption algorithm was considered for implementation in the WSNs, which was presented by the literature [107]. Such implementation was performed onto the Mica2 mote, a low-powered sensor node developed by UC Berkeley. The Mica2 mote consists of an 8-bit Atmel AVR processor, 128kB of code memory, 512kB EEPROM, 4kB of data memory and a ChipCon CC1000 radio. Since the operating system of the Mica2 mote is in TinyOS, the HIGHT block cipher was written in NesC. The HIGHT block cipher implementation requires a total of 3906Bytes of ROM and 584Bytes of RAM [107].

2.6.5 MiniSec Architecture for Secure WSNs

In later years, a secure sensor network communication architecture known as MiniSec was then introduced in the literature [108]. The MiniSec was developed as to achieve lower energy consumption and higher security level than the previously proposed TinySec [102] architecture. The MiniSec was implemented onto Moteiv Telos mote that features 8 MHz TI MSP430 microcontroller that has a 16-bit Reduce Instruction Set Computer (RISC) processor. The Moteiv Telos mote transmits data through the use of available CC2420 radio.

Two different modes of MiniSec were presented, which are the MiniSec-U and MiniSec-B. For the MiniSec-U, it involves two security primitives, which are the Offset CodeBook (OCB) encryption and Skipjack encryption. The MiniSec-U requires about 4000 lines of NesC code with 874 bytes of RAM and 16KB of code memory utilisations. Nevertheless, the 80-bit symmetric keys used in the Skipjack block cipher will not be secured in future that was mentioned in the literature [109]. Whereas for MiniSec-B, it utilizes both loose time synchronization and Bloom filters for implementation. However, details of hardware implementations for MiniSec-B were not mentioned in the literature [108].

2.6.6 TinyECC: Elliptic Curve Cryptography in WSNs

In [110], a TinyECC was developed based on the Elliptic Curve Cryptography (ECC) and implemented onto the WSNs. The developed TinyECC included three ECC schemes, which are the Elliptic Curve Diffie-Hellman (ECDH) key agreement scheme, the Elliptic Curve Digital Signature Algorithm (ECDSA) and the Elliptic Curve Integrated Encryption Scheme (ECIES). These ECC configurations were implemented onto a few different sensor platforms, which are MICAz, TelosB, Tmote Sky and Imote2. These are the popular sensor platforms which were embedded with 8-bit, 16-bit and 32-bit processors respectively [110].

Besides, a set of optimization switches was added to provide flexible configuration of TinyECC such that different resource consumptions and performance demands were met [110]. As reported in the literature, lower energy consumption was achieved for ECDSA, ECIES and ECDH configurations when all the optimization switches enabled. However, such implementations had a great increased in ROM and

RAM utilisation. For example, ECDSA required a total of 19,308Bytes of ROM and 1,510Bytes RAM on MICAz sensor platform when the optimization switches were enabled. While the optimization switches were disabled, the ECDSA required 10,180Bytes of ROM and 152Bytes of RAM. With all the optimization switches being disabled, the code size has been greatly reduced but the execution time of TinyECC is increased which increased the energy cost by 6 to 25.4 times [110].

2.6.7 CURUPIRA Block Cipher for WSNs

In [111], the CURUPIRA encryption was implemented by using VLSI with 0.13μm technology. The proposed CURUPIRA block cipher uses a 96-bit key and it operates for 10 rounds. For this CURUPIRA block cipher, it involved both the Key Scheduling unit and the CURUPIRA Core unit. The implementation achieves a throughput 960 Kbps at 100 KHz frequency. This implementation requires gate counts of 9,450 gates (1,350 flip-flops) and 40 bytes of block RAM. The CURUPIRA Block Cipher requires to undergo a complex processing in order to have the data encrypted that would increase the image data transmission latency. Besides, high throughput with high hardware utilisation of the CURUPIRA Block Cipher is not recommended for use in the WSNs, as it introduces higher power consumption [111].

Previously developed TinySec [102] and MiniSec [108] do have security concerns that lead to the development of CURUPIRA-2 block cipher for WSNs [112]. In this literature [112], there are 2 different configurations of CURUPIRA-2 block ciphers were developed and evaluated. One of the CURUPIRA$_C$-2 configurations was implemented with two 256-byte tables (one for S-Box and another for the *xtimes* operations) and uses many pointers and matrices. For another case, CURUPIRA$_{k96}$-2 configurations were restricted to 96-bit keys and relied on basic-type variables instead of indirect addressing instructions used in CURUPIRA$_C$-2. Both of these configurations were implemented onto the 8-bit PIC18F8490 microcontroller. The CURUPIRA$_C$-2 requires a total of 512Bytes of ROM and 1,238Bytes of programme memory. Then the CURUPIRA$_{k96}$-2 requires a total of 512Bytes of ROM and 1532Bytes of program memory [112].

2.6.8 Broadcast Encryption Scheme in WSNs

In 1993, the broadcast encryption scheme was initially introduced by Fiat and Naor [113]. Based on the Identity-Based Encryption (IBE) and Attribute-Based Encrpytion (ABE), an efficient broadcast encryption scheme was proposed by the literature [114]. The simulation study showed that the proposed scheme was also collusion resistant and stateless. The proposed broadcast encryption scheme in this literature is assumed to be performed in simulation since the authors did not mention the hardware involved [114].

In [115], another efficient Identity-Based Broadcast Signcryption (IBBSC) scheme was proposed for WSNs. The proposed IBBSC scheme was an extension of Delerablée's Identity-Based Broadcast Encryption scheme [116]. The proposed IBBSC scheme was then implemented on the Tmote Sky sensor platform which was equipped with MSP 430F1611 microprocessor [115].

2.6.9 Authenticated-Encryption Schemes in WSNs

In [10], a different Authenticated-Encryption with Associated Data (AEAD) schemes were tested in WSNs. The authors found that the CCFB+H, EAX, OCB and LETTERSOUP algorithms have its corresponding performance for different security level and data size. With the use of MSP430-size and MSP430-ram-usage tools, the authors managed to determine that CCFB+H algorithm requires 3,856Bytes of ROM and 204Bytes of RAM. As a result, the CCFB+H algorithm was selected for used in the proposed AEAD scheme because the small memory usage of the block cipher used in this algorithm. The proposed AEAD schemes were implemented onto the Crossbow TelosB nodes (motes) [10].

2.6.10 Crypto-Processor Encryption Algorithms for WSNs

The Crypto-Processor architecture that performs the encryption algorithms onto the sensed data before transmitting it across the WSNs [117]. As mentioned in this literature, software implementation of the encryption algorithms are considered to be less energy efficient, less time efficient and also less secure. Software based encryption algorithms are vulnerable to security threat such as ease in modification and compromising the keys used for encryption. As a result, a Crypto-Processor architecture was designed for a custom hardware platform that provides hardware-based key generation, storage and encryption. The Crypto-Processor consists of a few encryption techniques which includes AES-128 encryption, Elliptic-Curve GF(163) and SHA-256 encryption. The developed Crypto-Processor was implemented onto Xilinx Spartan-6 FPGA and required a total of 4828 slices and operates at 92.67MHz with power consumption of 17mW. However, the Crypto-Processor only provides security protection on the data and does provide any error correction capabilities on the encrypted data. Usually encrypted data (ciphertexts) do not have any tolerance against error since decrypting the ciphertexts requires correctly received data [117].

2.6.11 Summary

For most of the proposed encryption block cipher, the implementations were mainly programmed onto the available microcontroller of the sensor nodes. As for FPGA implementation of encryption schemes, the AES technique would require the least amount of hardware compared to both Crypto-Processor and CURUPIRA encryption method. The encrypted data produced from the developed Block Cipher do not have any error correction capabilities. Decrypting erroneous ciphertext (encrypted data) would lead to obtaining incorrect data (plaintext). Any received error encrypted data at base-station, it still need to rely on Automatic Request for Retransmission (ARQ) to retransmit the encrypted data until correct data is received. Thus it would require the sensor node to spend additional transmission energy for retransmitting the data. Therefore, this research was carried out as to develop a joint encryption and forward error correction scheme processing framework that would provide both data security and data reliability in the resource constrained WVSNs. Hence, the Cauchy Reed Solomon (CRS) coding scheme was considered in this research, as this coding scheme has both encryption and forward error correction capabilities [20].

2.7 JOINT SCHEMES

Compression schemes, forward error correction schemes and encryption schemes for use in WVSNs were review in the previous Section 2.3, Section 2.4 and Section 2.5, respectively. All these schemes were performed in separate module and combining them would result in complex hardware implementation for the WSNs. In order to reduce the hardware complexity, many research works were performed and to prove the feasibility in combining the aforementioned schemes together in WSNs. Therefore, the following Section 2.6 describes the existing research works on combined schemes (eg. compression with encryption, encryption with error correction, compression with error correction) for used in the WSNs.

2.7.1 Joint Source Channel Coding and Power Control for WSNs

In [118], the literature combines JPEG2000 compression and error resilient coding scheme on image transmission over WSNs to provide both image compression and error protection. The authors proposed an energy efficient system to minimize the overall processing-and-transmission energy consumption, known as low-complexity Joint Source Channel Coding and Power Control (JSCCPC) algorithm. The system does not provide any security protections and this allows the adversary to learn any coded data transmitted across the WSNs. Simulation study shows good performance is achieved by the JSCCPC algorithm [118].

2.7.2 Video Compression BCH Code in Wireless Video-Surveillance Networks

In another literature [119], simulation studies on the effect of combined video compression scheme and FEC coding scheme for Wireless Video-Surveillance Networks were carried out. For the simulated system platform, an Intel StrongARM 1100 microprocessor operating at 59MHz was considered to be used for this simulation studies. The JPEG integer kernel with 8:1 image compression ratio was selected to compress streams of image data. As reported by the authors, such operations (compression) would required to have energy consumption of 2.87mJ and execution time of 89.8ms. Next, the BCH(255,177,11) coding scheme was integrated with the JPEG integer kernel to provide reliable data transmission in a noisy communication channel and protect streams of compressed image data. As such, the effects of using the FEC coding scheme on the compressed image data were investigated. From the simulation studies, it showed that FEC always provides successful image delivery to the base-station. This reduced the energy consumption of sensor nodes when less number of compressed image data retransmission was needed. However, the software implementation (StrongARM processor) of FEC required an enormous energy cost as compared to hardware implementation. Therefore, the authors recommended to use FEC dedicated integrated circuit (hardware implementation) for the Wireless Video-Surveillance Networks, especially when the communication channel is very bad [119].

2.7.3 SAC and Multiple-Input Turbo Code for WSNs

In the effort of combining three schemes together, combined secure data aggregation and source-channel coding algorithm (called as SAC) was proposed to provide data compression, data security and reliability for the WSNs [24]. Data aggregation was used to reduce the redundant data and combine several unreliable data measurement to produce more accurate data. When direct communication between two neighbour sensor nodes is available, sensor node with more residual energy performed the data encryption and source-channel coding using the Multiple-Input Turbo (MIT) code on the encrypted data. However, the MIT code was only performed by the sensor nodes to provide reliable data transmission in WSNs when the Bit Error Rate (BER) is not acceptable. For unacceptable BER, the performance of the MIT code was evaluated by simulating it with an Additive White Gaussian Noise (AWGN) channel at a coderate of ½. From the evaluation results, it showed the feasibility of using combined secure data aggregation and MIT code for WSNs [24].

2.7.4 Robust Encryption for Secure Image Transmission in Wireless Channels

An “Opportunistic Encryption” was introduced and used to encrypt the JPEG compressed image [120]. The “Opportunistic Encryption” performed encryption on image data with different security level based on the channel conditions. The study claims that the “Opportunistic Encryption” does provide a better throughput compared to fixed block length encryption. At the same time, better performance was achieved by “Opportunistic Encryption” under all channel conditions. However, the authors did not mention the error correction code used to protect the encrypted compressed image data. If there are any errors occurred onto the encrypted image data, it may cause incorrect compressed image data to be decrypted. Furthermore, incorrect compressed image may cause degradation in the quality of reconstructed image. Therefore, this may increase the rate of retransmission thus reducing throughput of the wireless network [120].

2.7.5 FPGA Image Compression Encryption Scheme

Image Compression Encryption Scheme (ICES) implemented on Altera FPGA was proposed in [121]. The ICES compresses the image through the use Significance-Linked Connected Component Analysis (SLCCA) algorithm, which can achieve a high compression ratio and little distortion. After compressing the image, AES is used to encrypt the compressed image data such that it provides security on the image data. The study also shows that the hardware approach (FPGA) of encryption are faster compared to software-based solution (eg. Visual C++, JAVA) [121].

2.7.6 Error-Correcting Cipher for Wireless Networks

In [122], an error-correcting block cipher was proposed that performs both encryption and error correction for use in wireless networks. The High Diffusion (HD) code was used in the proposed cipher which is capable of providing security level similar to AES (in terms of number of active S-Boxes). Besides, the HD code also ensures good error-resilient on the encoded data. The proposed block cipher requires a long complicated processes and many keys (eg. Cipher key, Round key) to perform the encryption process [122]. As such, it is much more suitable to be performed in software based rather than in hardware based.

2.7.7 MVMP Secure and Reliable Data Transmission in WSNs

In [123], a new Multi-Version Multi-Path (MVMP) mechanism was proposed for use in the WSNs. The proposed MVMP mechanism integrates data segmentation, FEC coding, multiple paths and multiple versions of cryptographic algorithms together in order to achieve secure and reliable data transmission in WSNs. For encrypting the data, the MVMP mechanism uses a group of different Secret-Key Cryptography (SKC) (such as Skipjack, AES/Rijndael etc) and Public-Key Cryptography (PKC) (including Elliptic Curve Cryptography (ECC), NtrEncrypt etc). Once the data are encrypted, encrypted data are reorganized into k-packet size blocks. Then the selected Reed Solomon (RS) code encodes the encrypted data to ensure reliability and less request retransmission of data is needed [123].

2.7.8 ContikiSec in WSN

The use of Contiki operating system for Wireless Sensor Network with encryption schemes was presented in [124]. The author presented it as ContikiSec that is implemented onto the Modular Sensor Board (MSB-430). It has three different modes of operations, which are ContikiSec-Enc, ContikiSec-Auth and ContikiSec-AE. ContikiSec-Enc provides confidentiality and integrity, with Initialization Vector (IV) added, encryption using Cipher Block Chaining-Ciphertext Stealing (CBC-CS) with AES as underlying block cipher. ContikiSec-Auth provides authentication and integrity, through removing checksum field and includes a Cipher-based Message Authentication Code (CMAC) cryptographic algorithm [124].

ContikiSec-AE gives the highest security level that provides confidentiality, authentication and integrity [124]. This mode uses the OCB mode with AES as underlying block cipher, with a single shared-key for encryption and authentication. However, the ContikiSec-AE has the highest power consumptions among all these three operations. Besides, these implementations use the available microcontroller MSP430F1612 and Chipcon CC1020 that is embedded onto the MSB-430 board. The Contiki configuration was written in C language and utilized 2kB of RAM and 40kB of ROM [124]. The amount of actual hardware utilisation is not fully optimized since there will be unused peripherals available in the microcontroller.

2.7.9 Joint AES-LDPCC-CPFSK Schemes in WSN

In simulation study [125], a new joint scheme ‘Multilevel/Advanced Encryption Standard-Low Density Parity Check Coded-Continuous Phase Frequency Shift Keying’ (ML/AES-LDPCC-CPFSK) [126] was introduced for use in the WSNs. In this literature, the authors had incorporated multilevel inputs to AES encryption block in multiple of 128 bits (5 x 128 bits). Besides, multilevel inputs to LDPC encoder was also incorporated to protect the individual bits at each level of signal points. Based on the simulation results, the proposed ML/AES-LDPCC-CPFSK coding scheme did have improvement in coding gain and reduction in the number of CPFSK levels [125].

2.7.10 Secure and Reliable Distributed Data Storage in Unattended WSNs

Secure and reliable data distributed storage scheme was introduced that is based on RS coding scheme [127]. The proposed scheme provides Forward Secrecy (FSe), probabilistic Backward Secrecy (BSe) and reliability of data without relying on reliable nodes and communication channels. The simulation study also guarantees data confidentiality and data reliability against attacks launched by the Mobile Adversary [127].

2.7.11 Reliable and Secure Distributed In-Network Data Storage in WSNs

Reliable and secure distributed in-network data storage scheme for resource-constrained WSNs were introduced based on the combination of an elliptic curve cryptography scheme and erasure coding scheme [128]. For the proposed scheme, integration of both elliptic curve based stateful Public-Key Encryption (PKE) scheme and authenticated encryption mode-offset cookbook (OCB). This integration provides confidentiality, integrity and authentication, but also forward and backward secrecy of data confidentiality at lower energy consumption and memory overhead. The erasure coding scheme (RS coding scheme) was used to encode the data into sets of redundant fragments that guarantee original data recovery against pollution attack. To maintain reliable storage node with valid coded data fragments, simple and efficient data fragment integrity and consistency verification scheme was also integrated [128]. Performance evaluation was performed in MATLAB with results showing that less storage cost and energy consumption, as compared to similar approaches [127].

2.7.12 Error Correction-Based Cipher in WSN

From the previous literatures [124] [125], two different separate encryption block and error correction block were arranged together to provide a secure communication in Wireless Sensor Network (WSN). Figure 17 shows the traditional arrangement of the two separate encryption and error correction blocks for a secure communication in WSN. In [21], the authors had proposed a secure communication system using Error Correction-Based Cipher (ECBC) model for use in WSN. The proposed ECBC carries out both encryption and error correction in a single step rather than in two separate steps. Figure 18 shows the proposed ECBC model that will provide a secure communication system that prevents eavesdropper from retrieving the transmitted message [21].

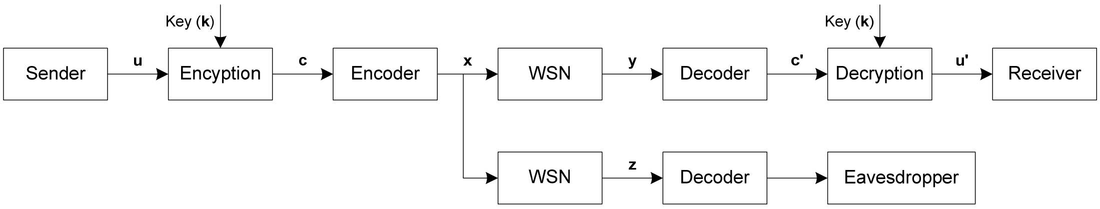


Figure 17 Conventional secure communication system model [21].

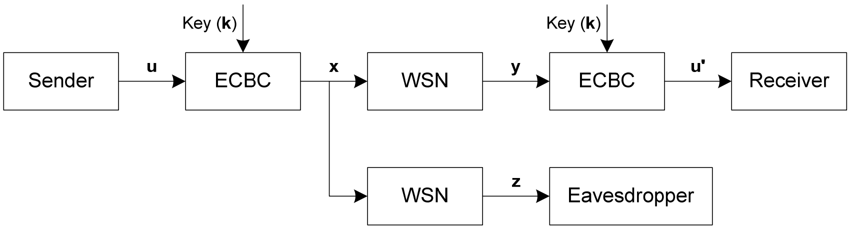


Figure 18 Secure communication system using ECBC model [21].

Based on the block chaining technique, the ECBC does not only provide security protection against eavesdropping but also provides data integrity [21]. For hardware implementations, the authors proposed two different ECBC architectures, which are the non-pipelined ECBC architecture and the pipelined ECBC architecture. For the non-pipelined ECBC architecture, it operates at a maximum frequency of 130.924 MHz and requires a total of 1,691 slices. Whereas the pipelined ECBC architecture, it operates at a maximum frequency of 105 MHz and requires a total of 2,058 slices [21]. Although hardware usage is lower, compared with the two separate AES and LDPC units, the ECBC architecture still acquires a significant amount of hardware area. This would also lead to high power consumption of nodes while performing encryption and encoding. Besides, the ECBC architecture has high operating frequency that would also relatively contribute to high power consumption. Thus the operating lifespan of sensor nodes will be significantly reduced.

2.7.13 Multipath Routing Approach for Secure and Reliable Data in WSNs

For the proposed approach [129], RS code is used to provide reliability and selective encryption is perform to provide security on multipath WSNs data transmission. First, the sink estimates the available node-disjoint paths from source node to sink. Then the sink decides on which *np* node path is the most secure paths (satisfy the required security level) to transmit data. Subsequently, using the RS coding scheme, the source node encodes *M* fragments of data that produces *K* parity fragments, which forms a complete codeword. Depending on the required security level, encryption is performed on the *K* parity fragments and different amount of data fragments [129]. Different fragments these encoded and encrypted data are then transmitted to the sink through different node path. This prevents the adversary from decoding the encrypted codeword. Extensive simulation studies were conducted on the performance of the proposed protocol using C++. However, the encryption method was not stated and the hardware complexity of the proposed protocol was not presented [129].

2.7.14 Compressed Sensing System for WSNs with Reliability

A system was designed with a Compressed Sensing (CS) based source encoding system for the data compression in WSNs [130]. With the use of Cyclic-Redundancy Check (CRC) coding, the proposed system has error detection capabilities and it discards any received data that consists of any errors. While discarding these error data, there is a small reduction in the measurement value. However, it is less destructive in obtaining reduced measurement value as compared to reconstructing the signal with corrupted measurements. The proposed system is suitable for used in WSNs which transfers compressed measurement value and not compressed image data. Discarding error compressed image data may cause distortion in the reconstructed image [130].

2.7.15 Summary

Nonetheless, joint compression, encryption and error correction capabilities are needed to provide a reliable secure low bandwidth data communications for the WVSNs. Compression schemes, security protection schemes and error correction schemes were previously developed to work independently without processing them in a single architecture. This is because processing them together requires complex hardware implementation. Consequently, there is a lack of research focus in this particular area. Nevertheless, there are some research works done in combining two schemes together.

There a few different joint encryption and error correction schemes that were previously proposed for WSNs. These include the Error-Correcting Cipher, MVMP, ContikiSec and Joint AES-LDPCC-CPFSK scheme. However, these approaches combined encryption and error correction schemes together without data compression. Besides, combined of compression and error correction schemes together for WSNs were also developed, including JSCCPC algorithm, JPEG combined BCH and CS combined CRC. Nevertheless these methods do not provide any security protection against adversary attack, such as eavesdropping of sensitive data (surveillance images) across the wireless communication channel. In the effort of combining these three schemes, simulation study on secure data aggregation combined with Slepian-Wolf coding principles on Multiple-Input Turbo (MIT) code was introduced. This simulation study showed that combining compression, encryption and channel coding (error correction) schemes together are feasible for WSNs.

As a result, this research was performed to develop a joint image compression, encryption and forward error correction processing framework for use in the resource constrained WVSNs. The joint image processing framework combined all three different schemes such that it provided the capabilities in reducing the size of image data, data security protection against eavesdropping and error data recovery at the base station.

CHAPTER 3

DEVELOPED DWT CRS MISC

In this Chapter 3, the methodology of the developed DWT CRS MISC architecture is discussed. At first, the Galois Field (GF) arithmetic operations were briefly introduced and explained in Section 3.1. The GF multiplier was one of the important functional blocks in the developed DWT CRS MISC architecture. Next, Section 3.2 describes the developed DWT CRS MISC architecture. Then Section 3.3 describes the control signals that are generated by a combinational circuit based on the Boolean Logic equation. The memory used in this DWT CRS MISC architecture, including programme memory and data memory, are discussed in Section 3.4. The programme instructions formats of the DWT CRS MISC architecture are shown in Section 3.5. With the developed DWT CRS MISC architecture, an algorithm that performs both DWT image compression (filtering) and CRS encoding was described in Section 3.6. In Section 3.7, the programme instructions were written and programmed into the DWT CRS MISC architecture.

3.1 GALOIS FIELD

GF operations can be found in many areas such as in error correction coding [17] [73] and cryptography systems [131] [132]. Sometimes, the Galois Field operations are also known as the finite field arithmetic operations in some literature. Usually, the simplest prime field GF(2) is extended to perform the subsequent finite field GF(2^2), GF(2^4) and GF(2^8) arithmetic operations. For example, the finite field GF(2^8)

arithmetic operation is used to perform the RS coding scheme and CRS coding scheme. For this research, the developed DWT CRS MISC architecture used the Galois Field GF(2^8) arithmetic operations to perform the CRS encoding onto the compressed image data. Therefore, a brief introduction on the Galois Field arithmetic operations was included in the following Section 3.1.1 and Section 3.1.2.

3.1.1 Galois Field GF(2)

The Galois Field, GF(2), is the simplest form of prime field arithmetic operations. In some context, the Galois Field is also known as the finite field [133]. For this arithmetic operations, it contains the elements of 0 or 1 only. Its arithmetic operations are only in modulo-2. The arithmetic operations are shown in Equation (8) to Equation (11) [73] [17].

$$1+0=1 \tag{8}$$

$$1+1=0 \tag{9}$$

$$1\times 0=0 \tag{10}$$

$$1\times 1=1 \tag{11}$$

3.1.2 Extension Galois Field GF(2^8)

In the Galois Field GF(2^8), the addition between these two field elements, $q(x)$ and $w(x)$, also follows the same principle as the addition of Galois Field GF(2). For the addition, the arithmetic operation is carried out by using the bitwise XOR onto the corresponding binary representations [133]. Nonetheless, the product of multiplication between these two field elements is the product of multiplying both $q(x)$ and $w(x)$, in modulo of the irreducible polynomial, for example $P_2(x)=x^8+x^3+x^2+1$. This GF(2^8) arithmetic operations are used in the DWT CRS MISC architecture that performs the CRS(20,16) coding scheme (in Section 3.2 and Section 3.6)

By considering that finite field element $q(x), w(x), z_a(x), z_b(x) \in GF(2^8)$, the Galois Field GF($2^8$) arithmetic operations are shown in Equation (12) to Equation (14).

$$Let \quad q(x)=q_7x^7+q_6x^6+q_5x^5+q_4x^4+q_3x^3+q_2x^2+q_1x+q_0$$

$$w(x) = w_7x^7 + w_6x^6 + w_5x^5 + w_4x^4 + w_3x^3 + w_2x^2 + w_1x + w_0$$

$$\begin{aligned} \textit{addition,}\quad z_a(x) &= q(x) + w(x) \\ &= (q_7 \oplus w_7)x^7 + (q_6 \oplus w_6)x^6 + (q_5 \oplus w_5)x^5 + (q_4 \oplus w_4)x^4 \\ &\quad + (q_3 \oplus w_3)x^3 + (q_2 \oplus w_2)x^2 + (q_1 \oplus w_1)x + (q_0 \oplus w_0) \end{aligned} \tag{12}$$

$$\begin{aligned} \textit{multiplication,}\quad z_b(x) &= q(x) \times w(x) \bmod P_2(x) \\ &= z_7x^7 + z_6x^6 + z_5x^5 + z_4x^4 + z_3x^3 + z_2x^2 + z_1x + z_0 \end{aligned} \tag{13}$$

$$\begin{aligned} \textit{where}\quad z_7 &= q_7w_0 \oplus q_6w_1 \oplus q_5w_2 \oplus q_4w_3 \oplus q_3w_4 \oplus q_2w_5 \oplus q_1w_6 \oplus q_0w_7 \oplus q_7w_6 \\ &\quad \oplus q_6w_7 \oplus q_7w_5 \oplus q_6w_6 \oplus q_5w_7 \oplus q_7w_4 \oplus q_6w_5 \oplus q_5w_6 \oplus q_4w_7 \\ z_6 &= q_6w_0 \oplus q_5w_1 \oplus q_4w_2 \oplus q_3w_3 \oplus q_2w_4 \oplus q_1w_5 \oplus q_0w_6 \oplus q_7w_5 \oplus q_6w_6 \\ &\quad \oplus q_5w_7 \oplus q_7w_4 \oplus q_6w_5 \oplus q_5w_6 \oplus q_4w_7 \oplus q_7w_3 \oplus q_6w_4 \oplus q_5w_5 \oplus q_4w_6 \\ &\quad \oplus q_3w_7 \\ z_5 &= q_5w_0 \oplus q_4w_1 \oplus q_3w_2 \oplus q_2w_3 \oplus q_1w_4 \oplus q_0w_5 \oplus q_7w_4 \oplus q_6w_5 \oplus q_5w_6 \\ &\quad \oplus q_4w_7 \oplus q_7w_3 \oplus q_6w_4 \oplus q_5w_5 \oplus q_4w_6 \oplus q_3w_7 \oplus q_7w_2 \oplus q_6w_3 \oplus q_5w_4 \\ &\quad \oplus q_4w_5 \oplus q_3w_6 \oplus q_2w_7 \\ z_4 &= q_4w_0 \oplus q_3w_1 \oplus q_2w_2 \oplus q_1w_3 \oplus q_0w_4 \oplus q_7w_7 \oplus q_7w_3 \oplus q_6w_4 \oplus q_5w_5 \\ &\quad \oplus q_4w_6 \oplus q_3w_7 \oplus q_7w_2 \oplus q_6w_3 \oplus q_5w_4 \oplus q_4w_5 \oplus q_3w_6 \oplus q_2w_7 \oplus q_7w_1 \\ &\quad \oplus q_6w_2 \oplus q_5w_3 \oplus q_4w_4 \oplus q_3w_5 \oplus q_2w_6 \oplus q_1w_7 \\ z_3 &= q_3w_0 \oplus q_2w_1 \oplus q_1w_2 \oplus q_0w_3 \oplus q_7w_5 \oplus q_6w_6 \oplus q_5w_7 \oplus q_7w_4 \oplus q_6w_5 \\ &\quad \oplus q_5w_6 \oplus q_4w_7 \oplus q_7w_2 \oplus q_6w_3 \oplus q_5w_4 \oplus q_4w_5 \oplus q_3w_6 \oplus q_2w_7 \oplus q_7w_1 \\ &\quad \oplus q_6w_2 \oplus q_5w_3 \oplus q_4w_4 \oplus q_3w_5 \oplus q_2w_6 \oplus q_1w_7 \\ z_2 &= q_2w_0 \oplus q_1w_1 \oplus q_0w_2 \oplus q_7w_6 \oplus q_6w_7 \oplus q_7w_5 \oplus q_6w_6 \oplus q_5w_7 \oplus q_7w_3 \\ &\quad \oplus q_6w_4 \oplus q_5w_5 \oplus q_4w_6 \oplus q_3w_7 \oplus q_7w_1 \oplus q_6w_2 \oplus q_5w_3 \oplus q_4w_4 \oplus q_3w_5 \\ &\quad \oplus q_2w_6 \oplus q_1w_7 \end{aligned}$$

$$\begin{aligned} \mathrm{z}_1 &= \mathrm{q}_1\mathrm{w}_0 \oplus \mathrm{q}_0\mathrm{w}_1 \oplus \mathrm{q}_7\mathrm{w}_7 \oplus \mathrm{q}_7\mathrm{w}_6 \oplus \mathrm{q}_6\mathrm{w}_7 \oplus \mathrm{q}_7\mathrm{w}_2 \oplus \mathrm{q}_6\mathrm{w}_3 \oplus \mathrm{q}_5\mathrm{w}_4 \oplus \mathrm{q}_4\mathrm{w}_5 \\ &\quad \oplus \mathrm{q}_3\mathrm{w}_6 \oplus \mathrm{q}_2\mathrm{w}_7 \\ \mathrm{z}_0 &= \mathrm{q}_0\mathrm{w}_0 \oplus \mathrm{q}_7\mathrm{w}_7 \oplus \mathrm{q}_7\mathrm{w}_6 \oplus \mathrm{q}_6\mathrm{w}_7 \oplus \mathrm{q}_7\mathrm{w}_5 \oplus \mathrm{q}_6\mathrm{w}_6 \oplus \mathrm{q}_5\mathrm{w}_7 \oplus \mathrm{q}_7\mathrm{w}_1 \oplus \mathrm{q}_6\mathrm{w}_2 \\ &\quad \oplus \mathrm{q}_5\mathrm{w}_3 \oplus \mathrm{q}_4\mathrm{w}_4 \oplus \mathrm{q}_3\mathrm{w}_5 \oplus \mathrm{q}_2\mathrm{w}_6 \oplus \mathrm{q}_1\mathrm{w}_7 \end{aligned}$$

$$\textit{inverse/division,}\quad q^{-1} = \left(q^2 \times q^4 \times q^8 \times q^{16} \times q^{32} \times q^{64} \times q^{128}\right) \bmod P_2(X) \tag{14}$$

The Galois Field GF(2^8) multiplications shown above was performed using the Massey-Omura method that is stated in the literature [134]. The GF(2^8) multiplication operation for hardware implemented is accomplished by performing bitwise XOR operations on all the listed bitwise AND operations, which is shown in Equation (13). The Equation (13) can be determined through multiplying both field element polynomials together. The product of multiplications between both field elements will be substituted with the corresponding primitive polynomial as stated in literature [134].

Less hardware resources were needed for the RS MISC architecture when only one Galois Field, GF(2^8) multiplier block was used to perform the Reed Solomon (RS) coding scheme. Example for RS(255,223), the RS Linear Feedback Shift Register (LFSR) encoding circuit would require a total of 32 GF(2^8) multiplier block. The RS LFSR encoding circuit performed the GF(2^8) multiplication in parallel that requires large hardware resources. However, the RS MISC performed the GF(2^8) multiplication sequentially, and it only requires to use one GF(2^8) multiplier block.

3.2 PROPOSED DWT CRS MISC ARCHITECTURE

The proposed new DWT CRS MISC architecture which consists of four main functional blocks is shown in Figure 19. The four main functional blocks are 11-bit XOR, Galois Field GF(2^8) Multiplier (GF MULT), 11-bit To 8-bit Conversion (11TO8) and an 11-bit ADDER. With these functional blocks, the new custom designed DWT CRS MISC architecture has four different programme instructions to be used to programme the architecture. The corresponding four programme instructions are XOR, GFMULT, 11TO8 and SBN instructions (refer to Section 3.3 for details on these programme instructions). These four functional blocks are designed to take in two input data and then process the input data depending to its block's functionality. The details of these functional blocks are discussed in Section 3.2.2 to Section 3.2.5.

In Figure 19, it shows that the DWT CRS MISC architecture consists of many registers. They are the three 11-bit registers (PC, R, MAR), one 12-bit register (MDR), and four 1-bit registers (OPCODE0, OCPODE1, N, Z). Besides, there are also two 11-bit 1-to-4 Multiplexers (MUXs), one 11-bit 4-to-1 MUXs, one 11-bit 2-to-1 MUXs and one 2-bit 2-to-1 MUXs. The most important part of the DWT CRS MISC architecture is the MEMORY. This is because the MEMORY stores both the programme memory and data memory. At the 12-bit output of MEMORY, there is a branch off to right from the output, show in Figure 19. The branch off is a 11-bit wide word length signal that is connected to the MAR. This is to output the 1st Operand and

2nd Operand Memory Addresses to the MAR. Whereas for 12-bit output to the left, it will output the 1st Operand and 2nd Operand (data) to the R register and the functional blocks respectively. Notice that the Most Significant Bit (MSB) of right output is connect to the MSB of MDR. This is have the OPCODE of the programme memory to remain the same, when there is a modification on the programme memory. The details on the Memory is presented in Section 3.4.

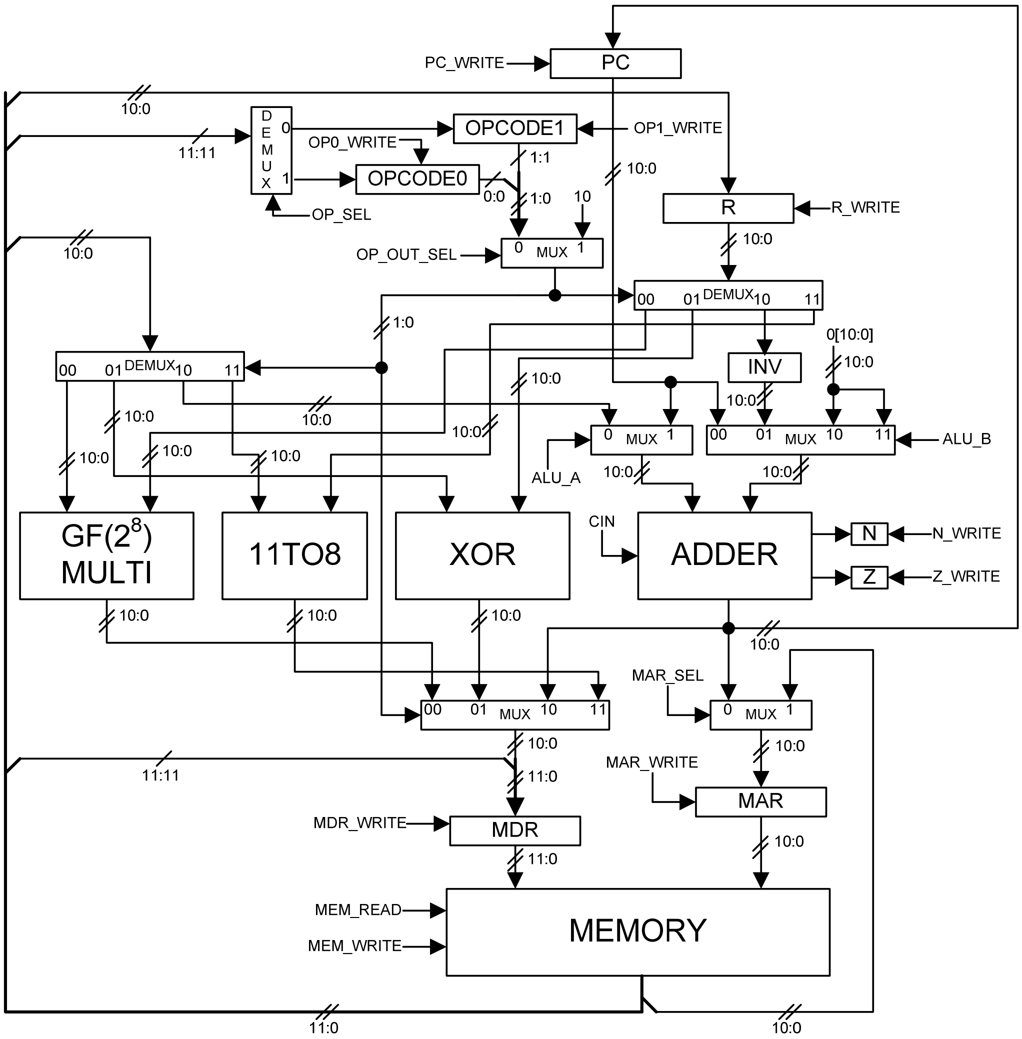


Figure 19 Proposed New DWT CRS MISC architecture.

For Programme Counter (PC) register, it stores the programme memory address location that needs to be read and executed. This is followed by the Read (R) register that stores the 1st Operand (data) value read from the Memory. The Memory Data Register (MDR) stores the output data from the functional blocks before the data is written to the Memory. This is done by having the output of 4:1 MUX drives the 10:0 bit line input of MDR. Then the MSB of MDR (bit 11) is driven by the MSB of the 2nd Operand, output from Memory. These two connections make up the complete 11:0 bits input signals into the MDR register. Next, the Memory Address Register (MAR) stores the memory address location before data/programme instruction are read and written back into the Memory. Both OPCODE0 and OPCODE1 registers store OPCODEs that are read from the programme instructions. This will determine the two Operands to input to required functional blocks. Lastly, the Negative (N) register indicates whether the SBN instruction produces a negative result and the Zero (Z) register indicates whether the PC value is zero. Further details on register, refer Section 3.3.3 on the detail arrangement of a 11-bit register.

Before the MISC architecture operates, the Program Counter (PC) register stores the location of the initial programme memory address, where this is the first programme instruction that will be executed (read) from the Memory. Once the MISC reads its first line of programme code, the PC register will store the next programme memory address such that the MISC will read the next line of programme code from the Memory again. Usually, the PC register stores next programme code address determined by increasing the current programme memory address by 1. However, if there is any programme branch occurred, then the current programme memory address at the PC register will be added with the ‘Target Address’. As such, the MISC will jump to and execute the corresponding targeted programme instructions when negative result is obtained from the SBN instruction. Details on how the MISC jump to and execute the new ‘targeted’ programme instructions, are further explained in the Section 3.5.

3.2.1 NAND Gate Representations

Logic gates are used in designing digital circuits, such as Inverter (INV), AND gate, OR gate, NOR gate and XOR gate. In realization the logic gates, these gates can be represented by NAND gates. The reason to represent them in NAND gates is because this research is implementing the proposed new DWT CRS MISC architecture into FPGA. The FPGA logic blocks can be fine grain modules such as two-input NAND gates. Besides, the logic blocks can also be coarse gain modules consisting complex structures, such as multiplexers, Look-Up Tables (LUTs) and Programmable Array Logic (PAL) [135]. Therefore, conversion of the basic logic gates into NAND gates to determine the longest gate delay that may encounter. This is required to set the operating frequency of the proposed DWT CRS MISC. The commonly used gates for this DWT CRS MISC architecture are INV, AND, OR and XOR. In Figure 20 to Figure 23, the NAND gates representations for AND, OR and XOR gates are shown.

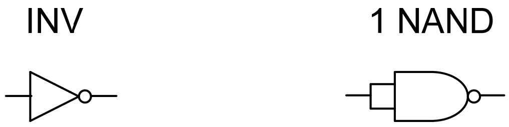


Figure 20 NAND gate representation of INVERTER.

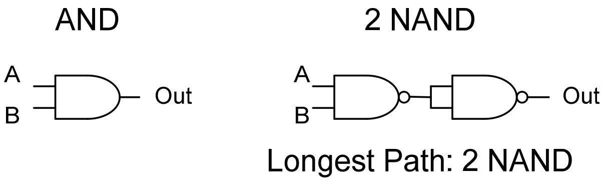


Figure 21 NAND gates representation of AND gate.

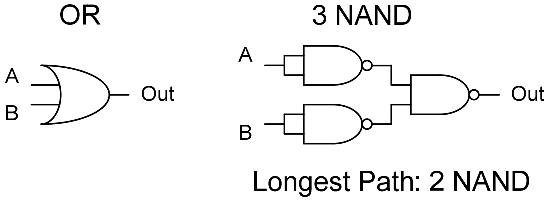


Figure 22 NAND gates representation of OR gate.

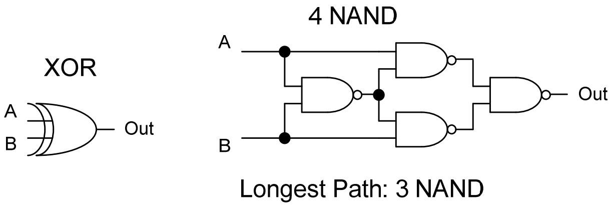
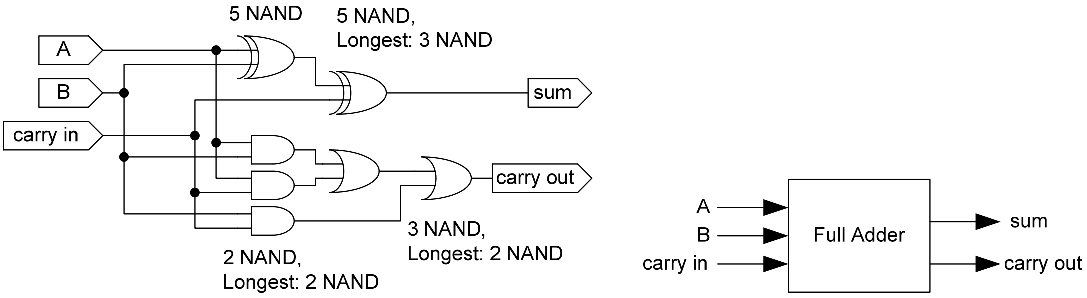


Figure 23 NAND gates representation of XOR gate.

3.2.2 ADDER Block

The ADDER block in this DWT CRS MISC will be made up of 11 full adders that are interconnected with each other. In Figure 24, the basic full adders are made up of 3 AND gates, 2 OR gates and 2 XOR gates. Based on the NAND gates representations in Section 3.2.1, a total of 28 NAND gates are required to represent a basic full adder.

Meanwhile, for the output to change, the longest delay path encounter would be 6 NAND gates, when there are changes at the inputs.



(a) Basic logic circuit build up. (b) Block diagram representation.

Figure 24 The representations of a full adder.

In Figure 25, the logic circuit of the ADDER block was designed for used in the proposed DWT CRS MISC architecture. Based on the logic circuit of ADDER block, the minimum longest delay path of the ADDER block is determined by considering the Bit 0 Full Adder (with 6 NAND gates), 9 subsequent Full Adder (each with 4 NAND gates) and 2 XOR gates. In NAND logic gate representations, it will encounter a total of 45 NAND gates delays for the outputs of ADDER block to stabilise. This situation only occurs when the summation for two inputs does not give zero output result. As for zero output result situation, the ZERO (Z) signal output will require an additional 9 NAND gates delays. The Z signal output has gates delays of an INV gate and 4 AND gates. This result in having the maximum longest gate delay path for the ADDER block to be 54 NANDs.

The main purpose of the ADDER functional block in the DWT CRS MISC is to perform the Subtract and Branch If Negative (SBN) instructions, as mentioned in Section 2.1.1. The functional block performs the subtraction of Operand *B* with Operand *A* (*B* = *B* - *A*). The result of arithmetic subtraction between these two Operands is stored back to Operand *B*. To have Operand *A* to become a negative value, the 1’s complement is performed by bitwise inverse onto Operand *A*. Then it will be followed by 2’s complement onto the bitwise inversed Operand *A*. The 2’s complement is done by adding a Logic 1 to the 1’s Complement of Operand *A*. In order to do this, the C_{IN} signal is used to input a Logic 1 by the control signal circuit at Clock Cycle 5.

Besides performing the SBN instructions, the ADDER block also increases the PC value during Clock Cycles 1, 4 and 8. If negative result is obtained from the SBN instruction, the ADDER block also adds the ‘Target Address’ onto the PC value at Clock Cycles 7. This is to execute the next targeted (jump to) programme instruction after the current instruction has completely executed.

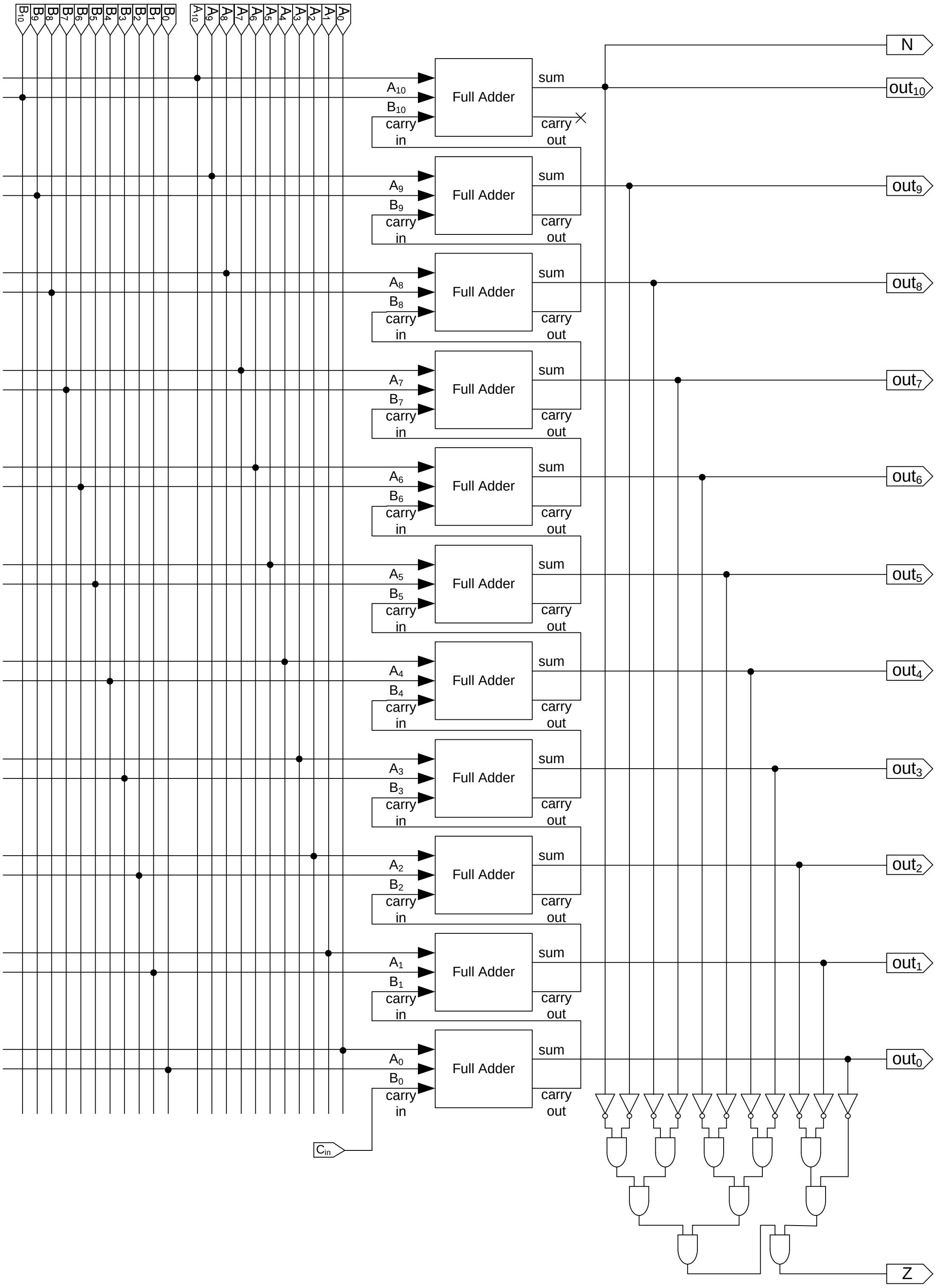


Figure 25 The ADDER Block consisting of 11 full adder

The ADDER functional block is designed to take in two 11-bit data. The reason for using 11-bit wide word length of data is the large memory spaces that are provided to store the large amount of programme instructions and data into the memory. With the 11-bit wide word length, this gives a total of 2^{11} = 2048 address memory location available in the memory.

3.2.3 GF(2^8) MULT Block

The MISC architecture was designed with the capabilities of performing the Galois Field arithmetic multiplications that are required for the CRS coding scheme. The advantage of the MISC architecture is that the Galois Field arithmetic multiplications are performed on the fly. In comparison to the method that uses lookup table for the Galois Field (GF) arithmetic multiplications [71], less amount of memory is needed for the CRS MISC processor to perform the GF multiplications. The GF(2^8) Multiplier (GF MULT) block is added into MISC architecture. This will allow the MISC to perform the GF(2^8) arithmetic multiplication onto two input Operands (B = B x A). The GF MULT block is needed to perform the CRS(20,16) coding scheme since each of the message symbols (input data) is in 8-bit wide word length.

The GF MULT block is made up of combinational logic circuits that performs the GF(2^8) Multiplier. The block consists of bitwise ANDs and XORs logic circuit processes two input Operands (data). Ideally, the output of combinational logic circuit will change immediately once the input changes. However, in reality, the changes in output will encounter a slight delay due to the delay of logic gates that are used in the circuits. In Figure 26, there are two data inputs (q, w) are fed into the GF Multiplier block. Only the 8 Least Significant Bits (LSBs, bit 7 to bit 0) from both data inputs are used and the remaining 3 Most Significant Bits (MSBs, bit 10 to bit 8) of both data inputs are not used. The reason for not using the 3 MSBs, from both data inputs, is the GF(2^8) multiplications only performs on 8-bit (LSBs) of data. Therefore, the 8 LSBs from the two input data will be fed into the Block Z_7 to Z_0. Each of these blocks is made up of logic circuits that represents the logic Equation (13) mentioned in Section 3.1.2. Figure 27 to Figure 34 shows the details of logic circuits that made up the Block Z_7 to Z_0. The output from these block represents the result for GF multiplications of two input data (in 8-bit). So the outputs of Block Z_7 to Z_0 are connected to the outputs of GF Multiplier block, from bit 7 to bit 0 respectively. Meanwhile, the output of 3 MSBs of the GF Multiplier block are set to zero since the outputs of GF block are only in 8-bit (8 LSBs of 11-bit).

Table 3 Gates delays of the GF MULT Block.

No.	Block	Gates Delays		
		AND	XOR	NAND
1	Z_7	1	5	17
2	Z_6	1	5	17
3	Z_5	1	5	17
4	Z_4	1	5	17
5	Z_3	1	5	17
6	Z_2	1	5	17
7	Z_1	1	4	13
8	Z_0	1	4	13

The GF MULT block does affect the operating frequency of MISC architecture. Before the result of GF multiplication is written in the Memory, a stable output from the GF MULT block is needed. For the output to be stabled, it will encounter a certain amount of gates delays. The gates delays for each of the individual Block Z_7 to Z_0 are listed in Table 3. From this table, the longest logic gates delays path encounter will be 1 AND gate delay and 5 XOR gates delays. This is considered to have 2 NAND gates delays and 15 NAND gates delays respectively. These gates are interconnected with each other. The final total count of gates delays will encounter in GF MULT block are 17 NAND gates delays.

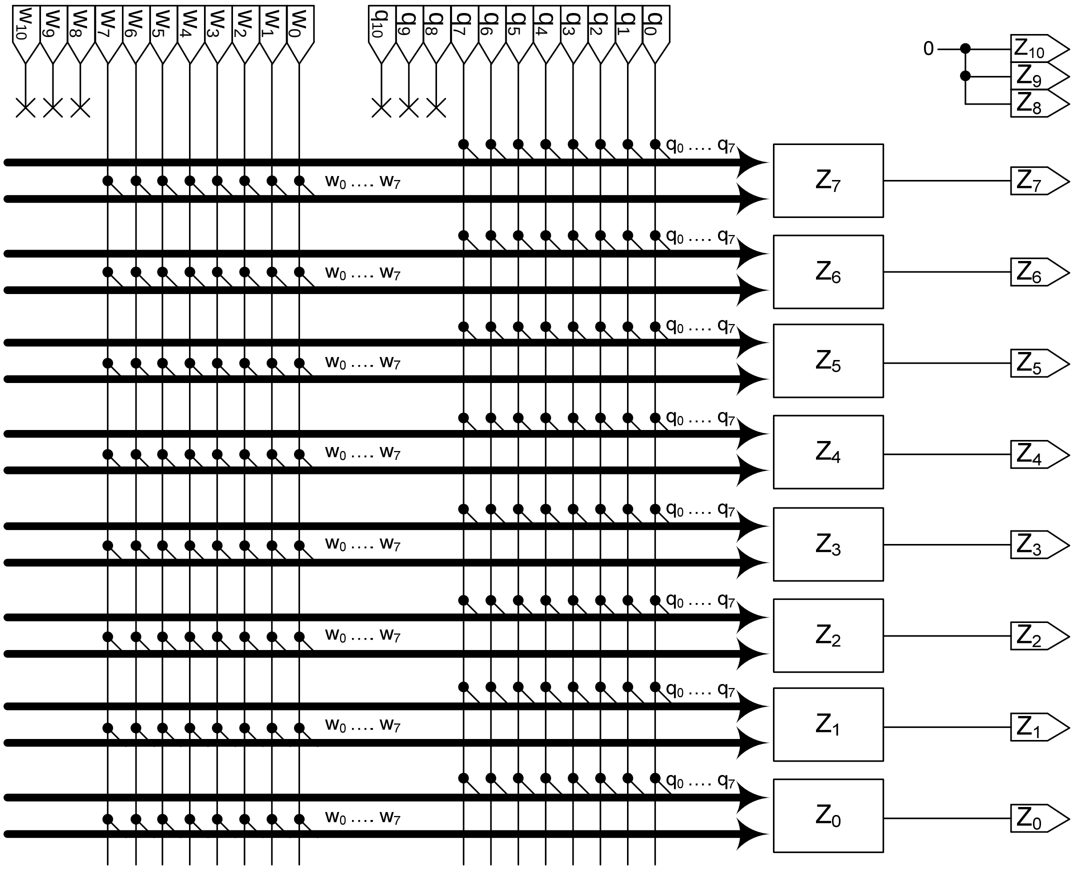


Figure 26 The complete GF(2^8) Multiplier block for DWT CRS MISC architecture.

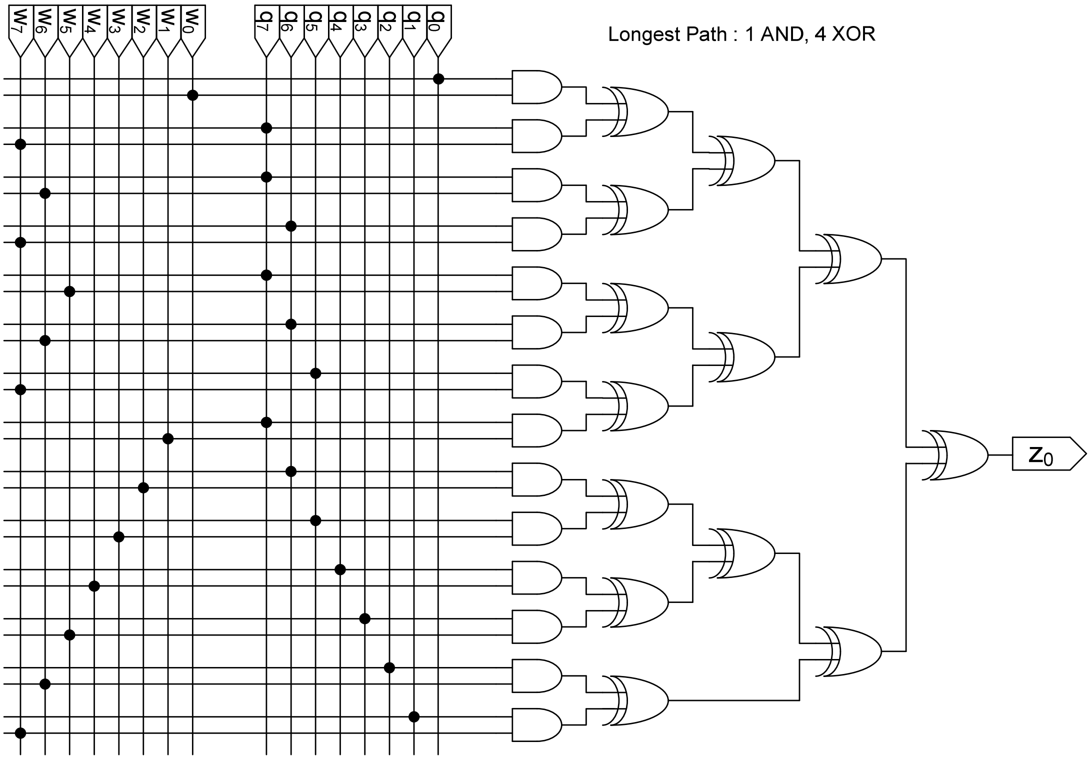


Figure 27 Block Z_0 (Bit 0) internal logic circuit block for the GF(2^8) Multiplier.

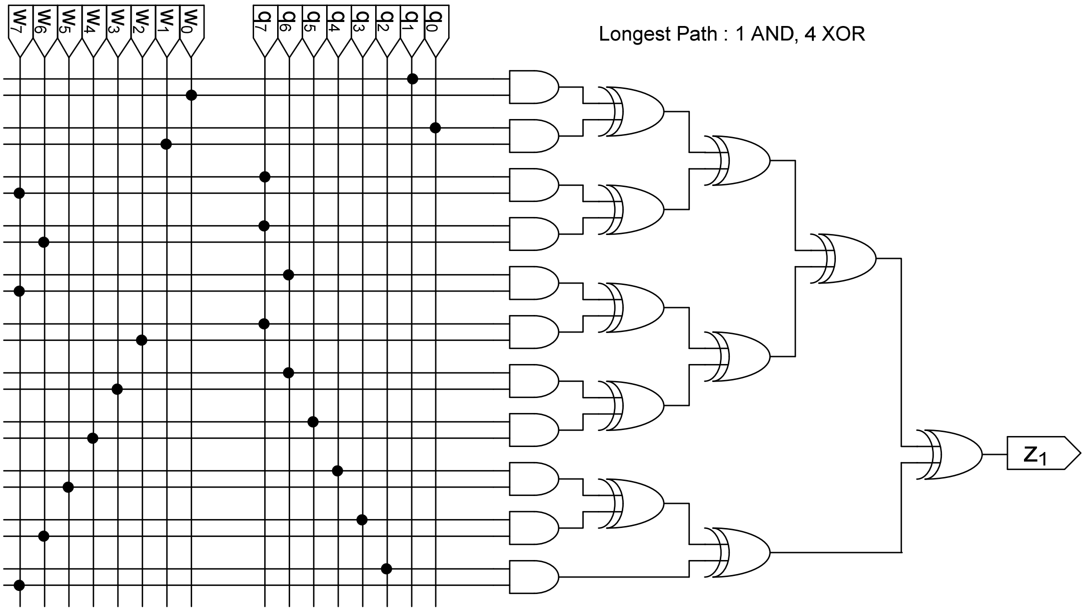


Figure 28 Block Z_1 (Bit 1) internal logic circuit block for the GF(2^8) Multiplier.

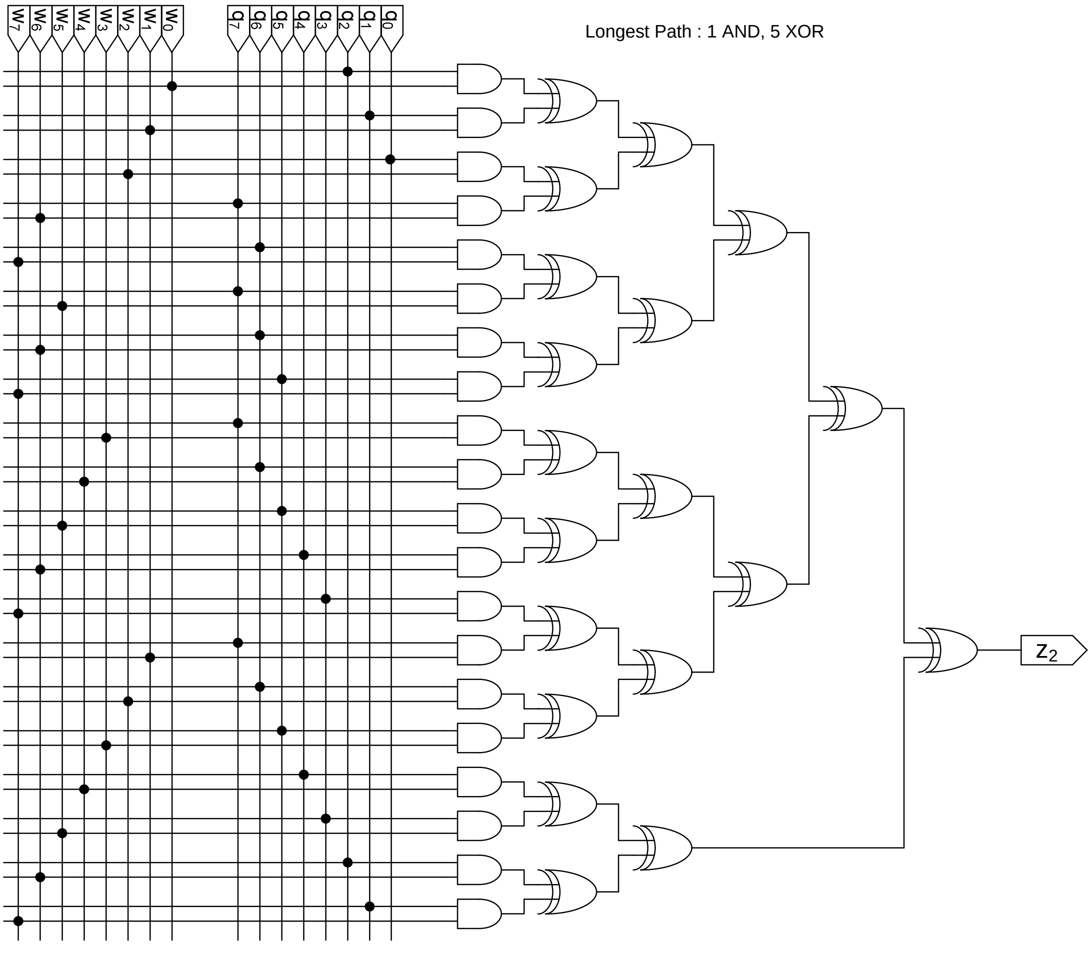


Figure 29 Block Z_2 (Bit 2) internal logic circuit block for the GF(2^8) Multiplier.

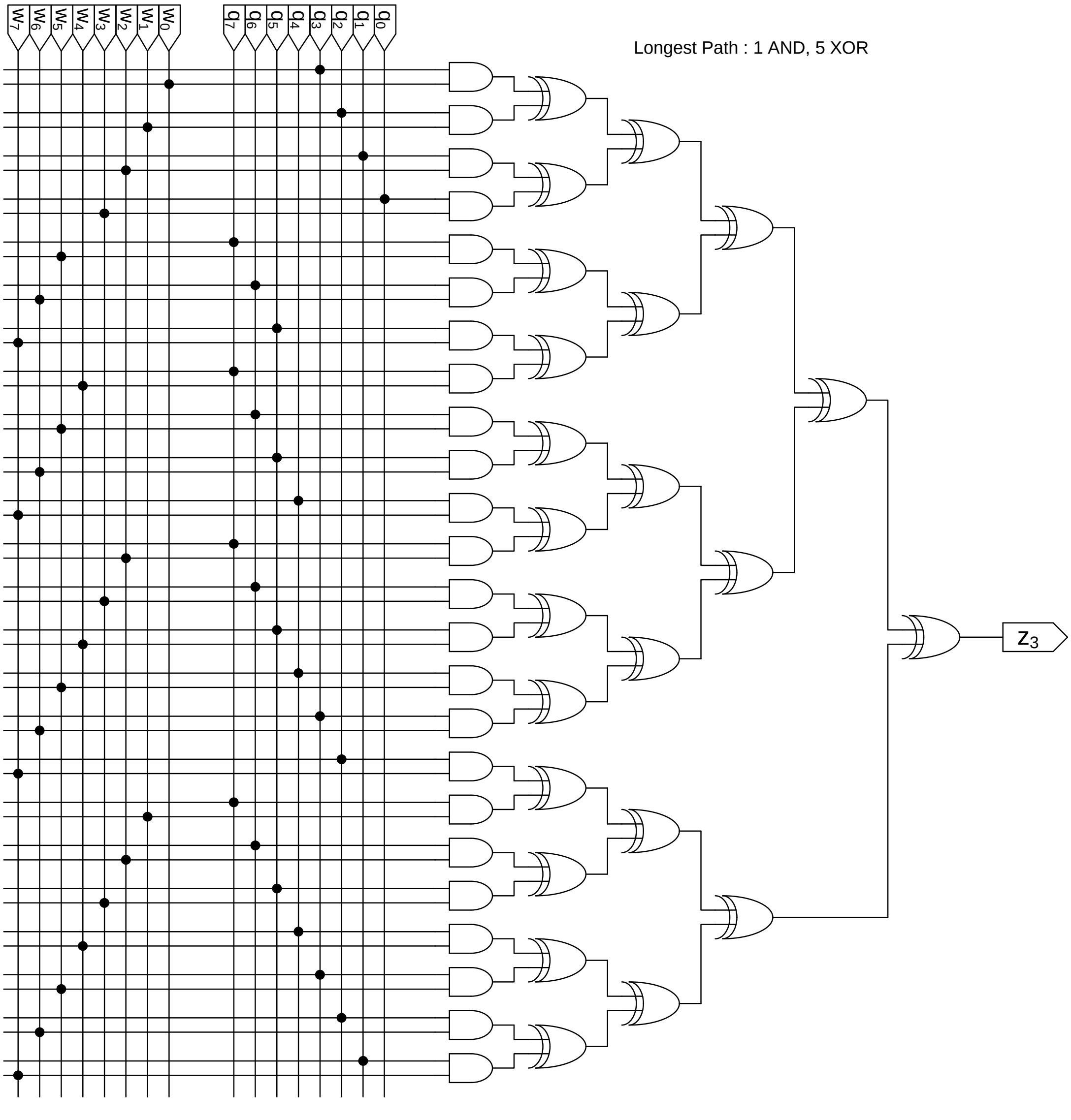


Figure 30 Block Z_3 (Bit 3) internal logic circuit block for the GF(2^8) Multiplier.

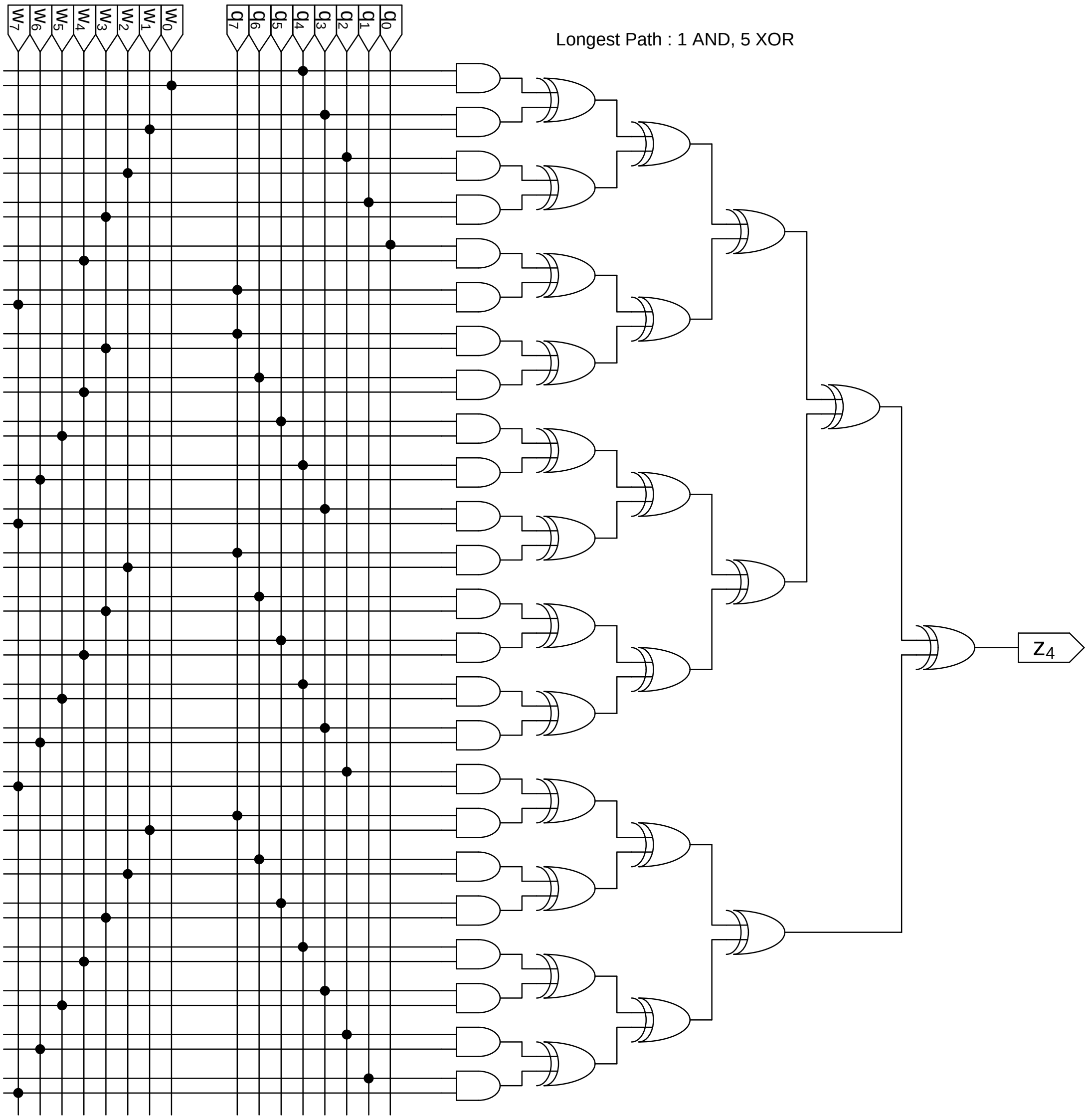


Figure 31 Block Z_4 (Bit 4) internal logic circuit block for the GF(2^8) Multiplier.

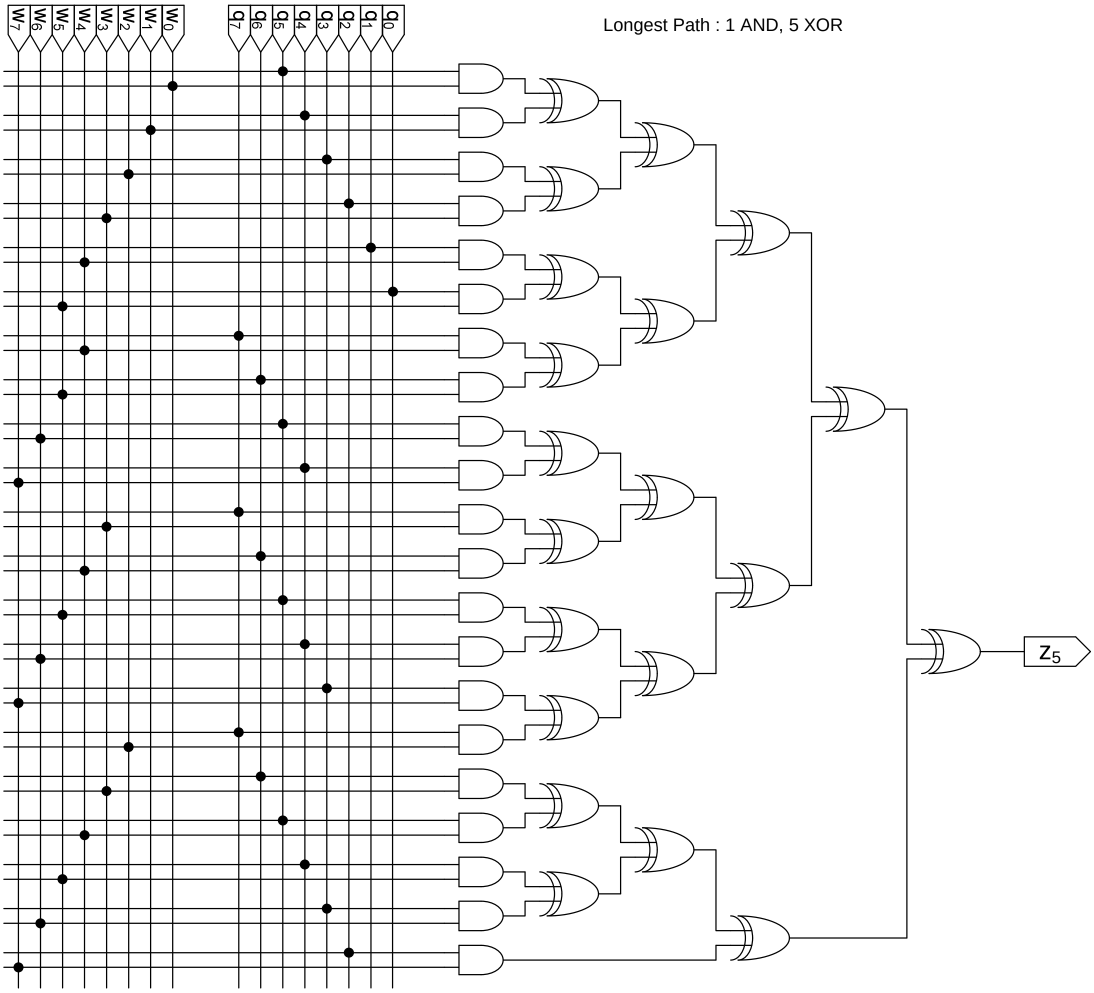


Figure 32 Block Z_5 (Bit 5) internal logic circuit block for the GF(2^8) Multiplier.

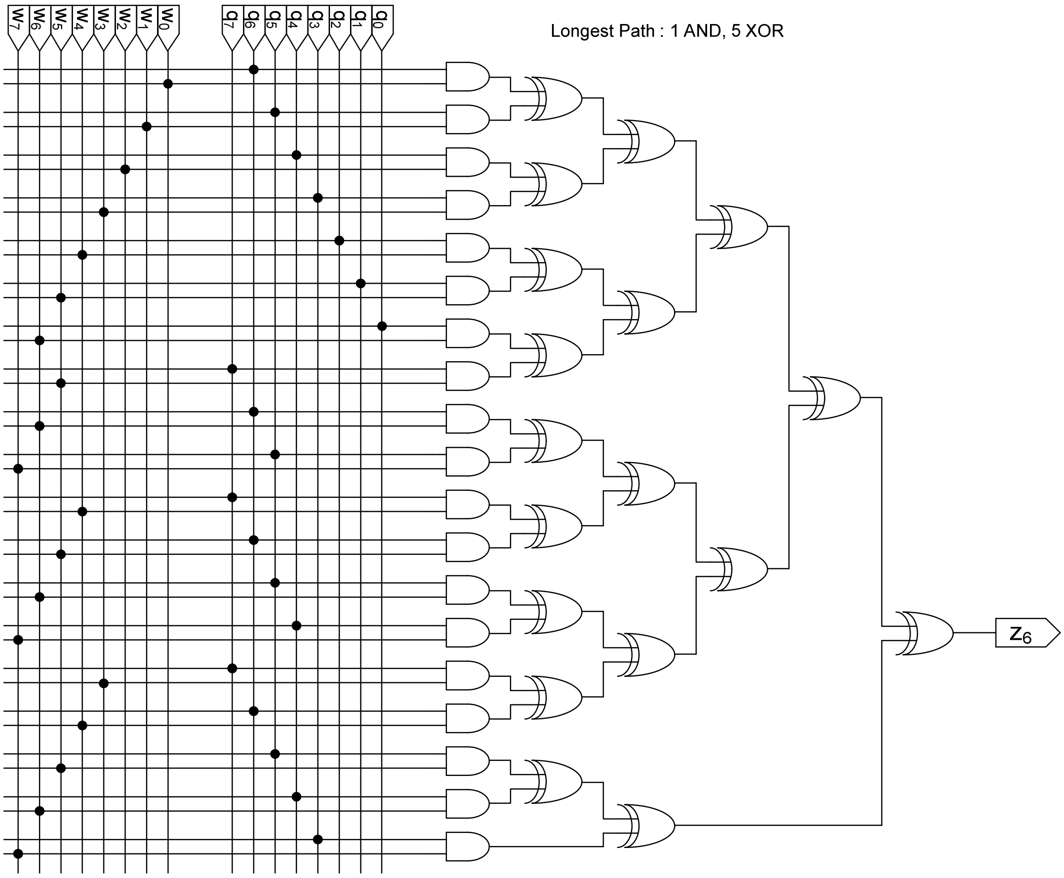


Figure 33 Block Z_6 (Bit 6) internal logic circuit block for the GF(2^8) Multiplier.

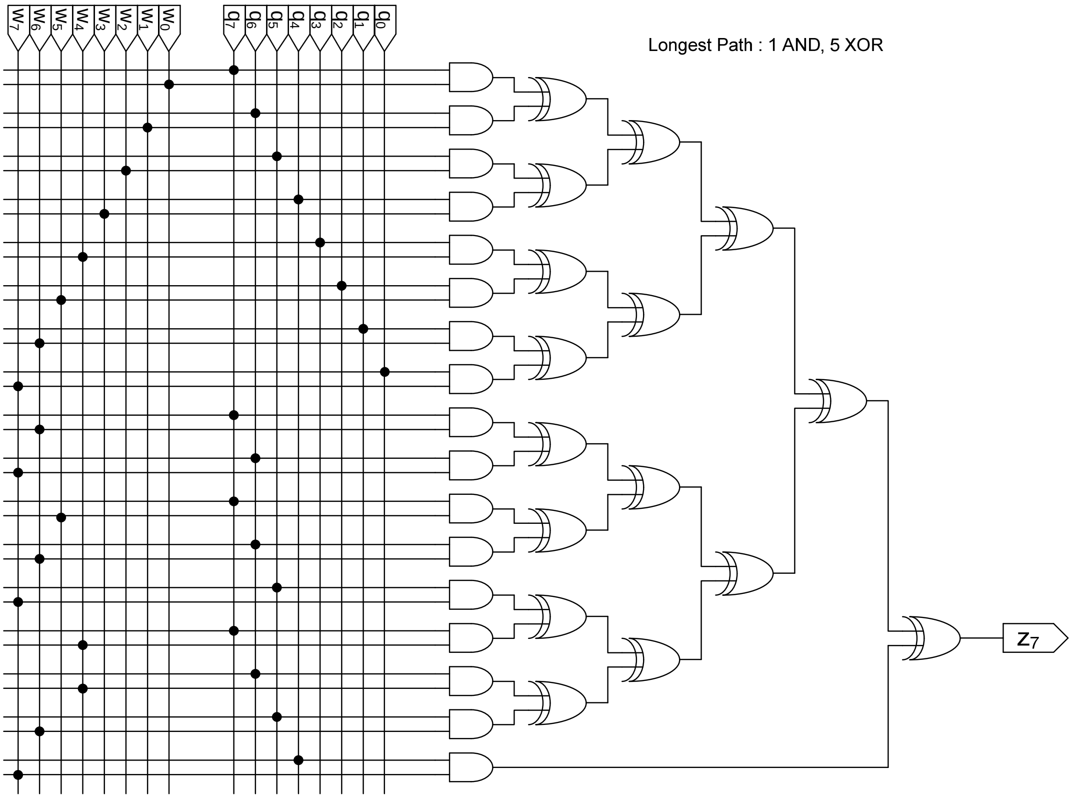


Figure 34 Block Z_7 (Bit 7) internal logic circuit block for the GF(2^8) Multiplier.

3.2.4 XOR Block

The XOR block performs the 11-bit bitwise XOR onto the two Operands (data) that are input into this functional block (*B* = *B* XOR *A*). Figure 35 shows the internal logic circuits of the XOR block. The longest logic gate delays for the XOR block would be 3 NANDs (1 XOR gate). This would not be the determining factors on the operating frequency of the MISC architecture.

With the XOR block, it allows the data located at a particular memory location to be cleared. This is done by XORing the two same input data together and the zero value output data is stored back to the particular memory location. Besides, the XOR block can be used to copy the data from the initial location to another empty data memory location. In comparison to SBN instruction, the XOR instruction saves an additional programme instruction that is required to copy the data from one memory location to another new location. Although the XOR block performs 11-bit XOR, the XOR can still perform the GF arithmetic addition that involves 8-bit data. Therefore, the XOR block plays an important role in performing the GF(2^8) arithmetic addition which is required in encoding the data using the CRS coding scheme.

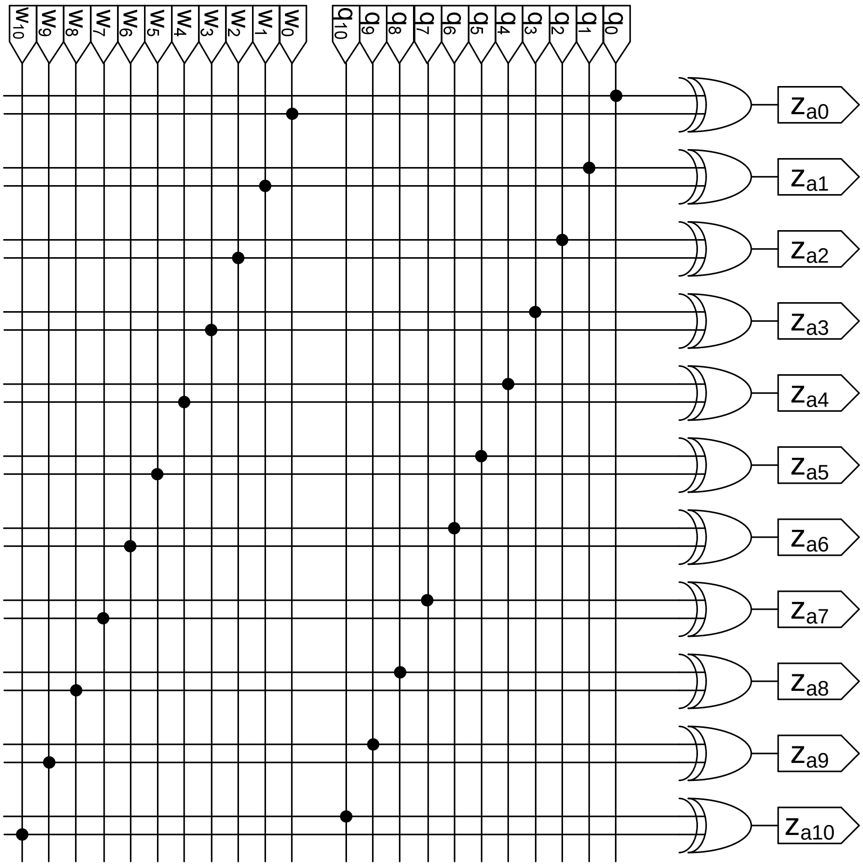


Figure 35 11-bit XOR Block that performs GF(2^8) Additions, Data Copying and Clearing Data.

3.2.5 11TO8 Block

For the 11TO8 functional block, its main purpose is to perform the conversion of 11-bit input data to become 8-bit output data. This is performed by taking the 7 LSBs of the input data combined with the MSBs (sign bit or bit 10) of the input data to produce a final 8-bit output data. It can seen that the bit 7 output is directly connected to bit 10 of the input. This is also meant the concatenations between MSB of the 11-bit input data and the 7 LSBs of the same 11-bit input data. The two MSBs (10:9) output from 11TO8 are fixed to Logic 0. They are fixed to Logic 0 since these bit line are not used for the GF arithmetic operations. Once the 4:1 MUX selected the 11TO8 block, the 10:0 bits of MDR will be driven by the output of the 4:1 MUX. The MSBs of the MDR will be driven by the MSBs of the 2nd Operand output from the Memory. The internal logic circuit of the 11TO8 block is shown in Figure 36. The logic circuit is made up of wire connection from input to output. There is no logic gate involved and the logic gate delay is considered to be insignificant as compared to other functional blocks, mentioned in Section 3.2.2 to Section 3.2.4. The reason of having this conversion function for the MISC architecture is to convert any negative values of the DWT coefficients from 11-bit signed data to 8-bit signed data.

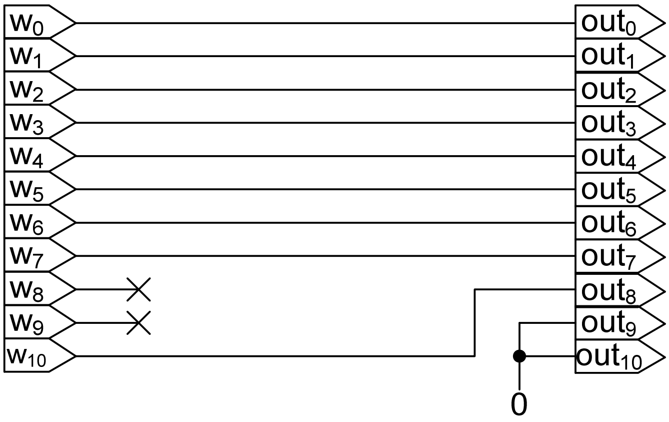


Figure 36 11TO8 block internal logic circuit.

3.3 DWT CRS MISC CONTROL SIGNALS

The proposed DWT CRS MISC architecture requires input control signals to perform compression and encoding onto the image data. Each of the programme instructions executed by the MISC requires a total of 9 Clock Cycles. During each Clock Cycles, different sequences of control signals are generated and input into the MISC architecture. The Logic state of these control signals required to input into the MISC architecture are shown in the Truth Table (Table 4). The Boolean Logic equation for each of the corresponding control signals can be determined from the Truth Table. With the Boolean Logic equations determined, as stated in Equation (15) to Equation (31), a combinational logic circuit is developed and it generates these control signals based on a 4-bit counter. The 4-bit counter counts from 0 to 8 and then restarts to 0 again once it has reached the maximum value 8. Each counter value changes the control signals according to the combinational circuit. Figure 37 illustrates the combinational logic circuit that generates the required control signals input into the MISC architecture. Note that during Clock Cycle 7, the PC_WRITE control signal is depended on the N signal output from the N register in the MISC architecture. The longest gate delay encountered by the control signals circuit is 6 NAND gates (i.e. 1 AND gate and 2 OR gates). The complete simulation waveforms on the control signals produced by the combinational logic circuit can be found in Section 4.1.

$$\text{ALU\_A} = \overline{C_3}\,\overline{C_2} + \overline{C_3}\,\overline{C_0} + \overline{C_2}\,\overline{C_1}\,\overline{C_0} \tag{15}$$

$$\text{ALU\_B}_1 = \overline{C_3}\,\overline{C_2} + \overline{C_3}\,\overline{C_0} + \overline{C_2}\,\overline{C_1}\,\overline{C_0} \tag{16}$$

$$\text{ALU\_B}_0 = \overline{C_3}\,C_2\,\overline{C_1}\,C_0 \tag{17}$$

$$\text{CIN} = \overline{C_3}\,C_2\,\overline{C_1} + \overline{C_3}\,\overline{C_1}\,C_0 + C_3\,\overline{C_2}\,\overline{C_1}\,\overline{C_0} \tag{18}$$

$$\text{MAR\_SEL} = \overline{C_3}\,C_2\,\overline{C_1} + \overline{C_3}\,\overline{C_1}\,C_0 \tag{19}$$

$$\text{PC\_WRITE} = \overline{C_3}\,C_2 C_1 C_0 N + \overline{C_3}\,\overline{C_2}\,\overline{C_1}\,C_0 + \overline{C_3}\,C_2\,\overline{C_1}\,\overline{C_0} + C_3\,\overline{C_2}\,\overline{C_1}\,\overline{C_0} \tag{20}$$

$$\text{R\_WRITE} = \overline{C_3}\,\overline{C_2}\,C_1\,\overline{C_0} \tag{21}$$

$$\text{Z\_WRITE} = \overline{C_3}\,\overline{C_2}\,\overline{C_1}\,\overline{C_0} \tag{22}$$

$$\text{N\_WRITE} = \overline{C_3}\,C_2\,\overline{C_1}\,C_0 \tag{23}$$

$$\text{MAR\_WRITE} = \overline{C_3}\,\overline{C_2}\,C_0 + \overline{C_3}\,C_2\,\overline{C_0} + \overline{C_3}\,\overline{C_1}\,\overline{C_0} \tag{24}$$

$$\text{MDR\_WRITE} = \overline{C_3}C_2\overline{C_1}C_0 \tag{25}$$

$$\text{MEM\_READ} = \overline{C_3}C_2\overline{C_1} + \overline{C_3}C_1C_0 + \overline{C_3}\,\overline{C_1}C_0 + \overline{C_3}\,\overline{C_2}C_1\overline{C_0} \tag{26}$$

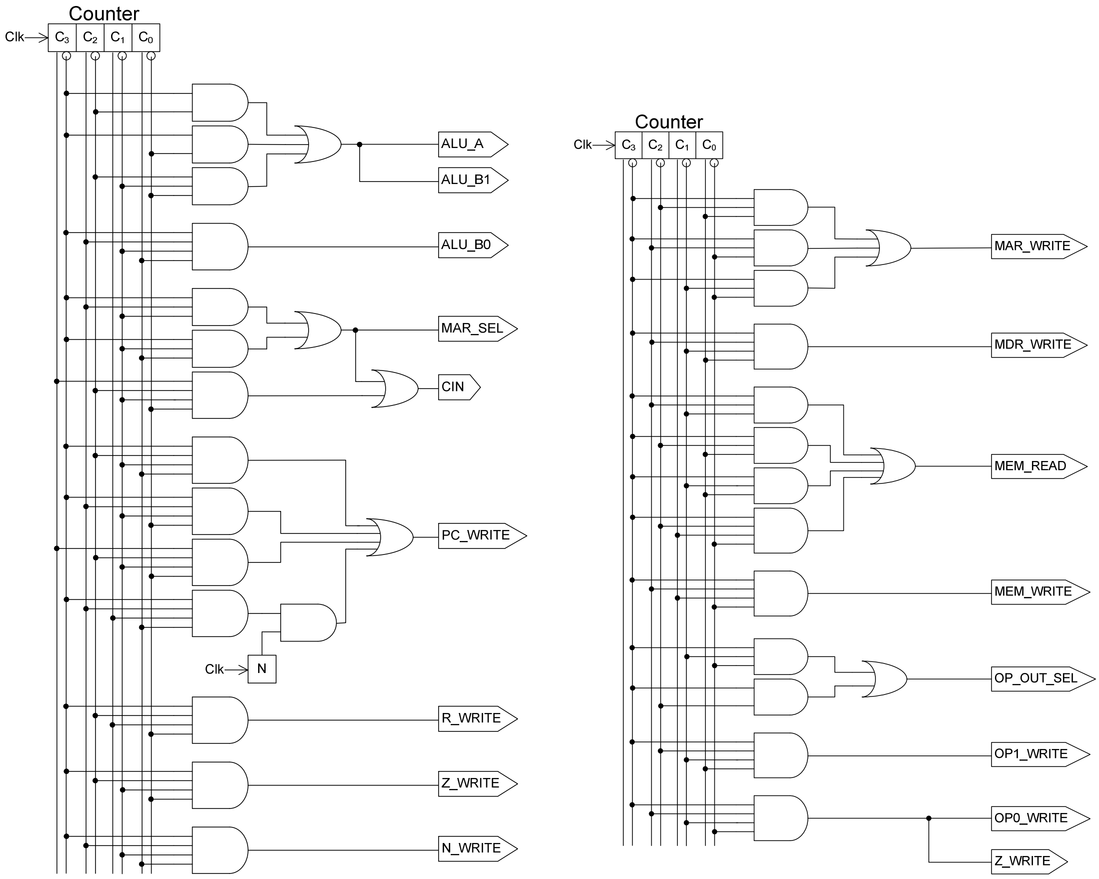
$$\text{MEM\_WRITE} = \overline{C_3}C_2C_1\overline{C_0} \tag{27}$$

$$\text{OP\_OUT\_SEL} = \overline{C_3}\,\overline{C_1}\,\overline{C_0} + \overline{C_3}\,\overline{C_2} \tag{28}$$

$$\text{OP1\_WRITE} = \overline{C_3}\,\overline{C_2}\,\overline{C_1}C_0 \tag{29}$$

$$\text{OP0\_WRITE} = \overline{C_3}C_2\overline{C_1}\,\overline{C_0} \tag{30}$$

$$\text{OP\_SEL} = \overline{C_3}C_2\overline{C_1}\,\overline{C_0} \tag{31}$$



(a) Part 1 (b) Part 2

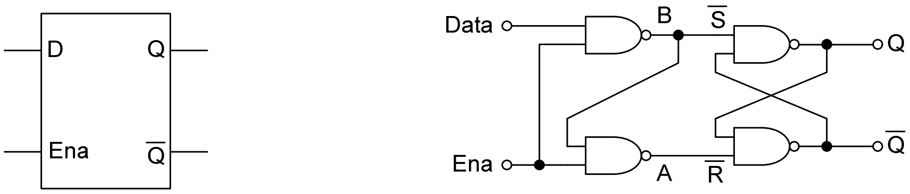
Figure 37 Combinational logic circuit for MISC architecture control signals.

Table 4 Truth table of DWT CRS MISC control signals.

CLK	ALU_A	ALU_B1	ALU_B0	CIN	MAR_SEL	PC_WRITE	R_WRITE	Z_WRITE	N_WRITE	MAR_WRITE	MDR_WRITE	MEM_READ	MEM_WRITE	OP_OUT_SEL	OP1_WRITE	OP0_WRITE	OP_SEL
0	1	1	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0
1	1	1	0	1	1	1	0	0	0	1	0	1	0	1	1	0	0
2	X	X	X	X	X	0	1	0	0	0	0	1	0	1	0	0	0
3	1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
4	1	1	0	1	1	1	0	0	0	1	0	1	0	1	0	1	1
5	0	0	1	1	X	0	0	0	1	0	1	1	0	0	0	0	0
6	1	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0
7	0	0	0	0	X	1/0 (N)	0	0	0	0	0	1	0	0	0	0	0
8	1	1	0	1	X	1	0	0	0	0	0	0	0	0	0	0	0

3.3.1 D-Latch

In designing digital circuits, the D-Latch is considered as one of the basic building block in constructing most sequential circuits [136]. Latches are usually used by digital designer to have a sequential device that watches its inputs consistently. When enable input is at High (Logic 0), the D-Latch will change its outputs at any time. Figure 38(a) shows the block diagram of a simple D-latch. The logic circuit that made up the D-latch is shown Figure 38(b). Based on the different inputs of D-Latch, the corresponding output waveforms are shown in Figure 39.



(a) Block Diagram. (b) Logic Circuit.

Figure 38 The Block Diagram Representations and Logic Circuit of a Typical D-Latch.

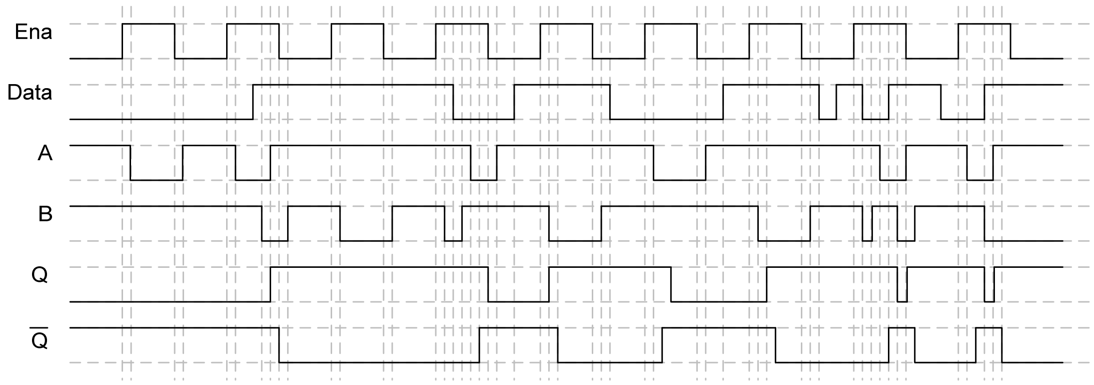


Figure 39 Output Waveforms of a Typical D-Latch with delays.

3.3.2 Edge-Triggered D Flip-Flop

Next, the edge-triggered D Flip-Flop is the basic building blocks that are used to construct a sequential device that normally samples its inputs [136]. Based on the inputs to D Flip-Flop, it will change its output only when there is a raise (or fall) in clock signal. The positive edged-triggered D Flip-Flop is constructed by joining two D-Latches. This is done by having the output of first D-latch connected to the input of next D-latch, shown in Figure 40 and Figure 41. From the D Flip-Flop logic circuit, it can be seen that the longest logic delay path is 2 INV and 4 NAND gates (total of 6 NAND gates). The functional behaviour waveform for a typical D Flip-Flop is also shown in Figure 42.

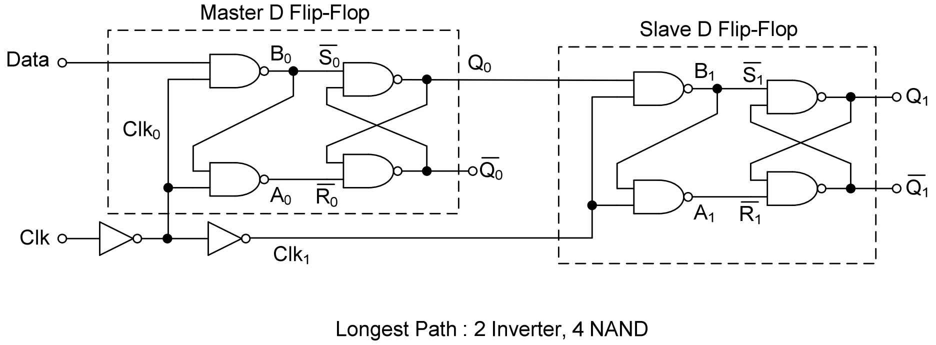


Figure 40 Positive Edge-Triggered D Flip-Flop Logic Circuit.

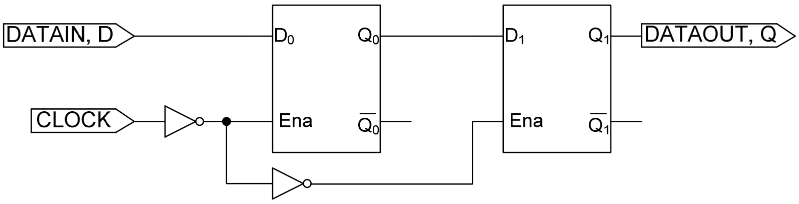


Figure 41 Block diagram of D Flip-Flop Constructed with 2 D-Latches.

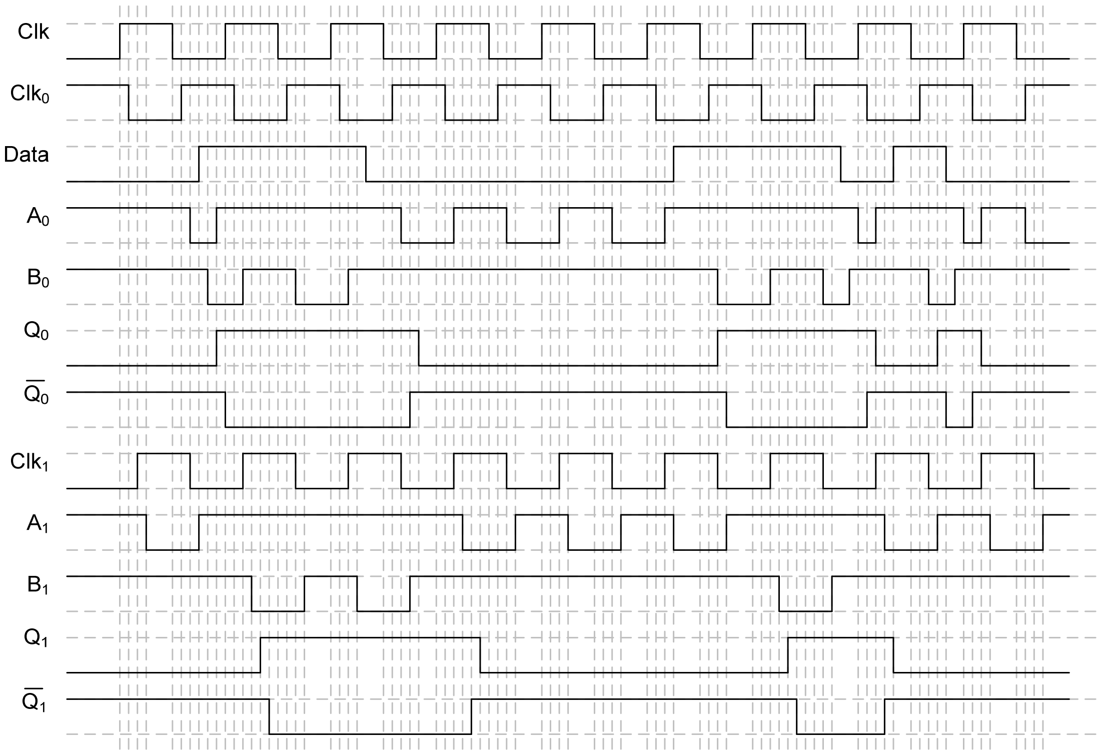


Figure 42 Functional Behaviour Waveform of a Positive Edge-Triggered D Flip-Flop.

3.3.3 Registers

The registers are memory storage devices that can be used to store data. For example, the MDR is a very important register that are used to store the result data. The result data are produced from each functional blocks (XOR, GF MULT, ADDER, 11TO8). As for the R register, it is used to store the Operand A read from the Memory. Another MAR is used to store the Operands' memory addresses (locations). The Operands' memory addresses are read from the Memory first before reading the data register. This is because the DWT CRS MISC architecture is built on a von Neumann architecture. The von Neumann architecture has both data memory and programme memory combined together as a single memory. In this DWT CRS MISC, the programme memory contains the Operands' memory addresses with an OPCODE.

For this research, these registers used are considered to be edge-triggered register. The data will be stored into the register during the rising edge of clock only. In Figure 43, there are 11-bit lines inputs/outputs from the registers. For each bit line (inputs/outputs), they are all connected to a edge-triggered D Flip-Flop, mentioned in Section 3.3.2. As for the 12-bit register, there will be an additional D Flip-Flop for one more extra bit line (inputs/outputs) compared to the 11-bit register.

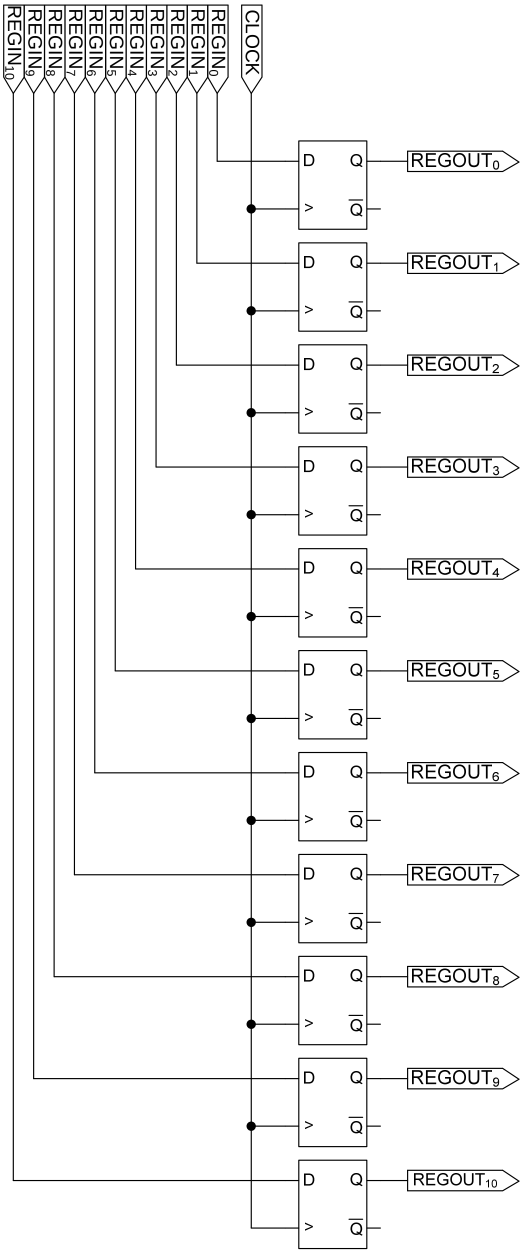


Figure 43 D Flip-Flops Arrangements to Form a 11-bit Registers.

3.3.4 4-Bit Counter for Control Signals

The 4-bit counter is made up of 4 D Flip-Flops and a sequential circuit. In Figure 44, the sequential circuit forms the connection between the outputs and inputs of the D Flip-Flops. This will allow these 4 D Flip-Flops to operate as a 4-bit counter. The 4-bit counter will keep on increasing its value during each rising edge of clock. For each counter value, specific control signals will be output to the DWT CRS MISC architecture. These control signals control the writing data to registers, read/write data to memory, multiplexers (MUXs) and DE-MUXs.

From Figure 44, the longest logic path encountered by the sequential circuit is 6 NAND gates. Since the 4-bit counter are depended on the sequential circuit mentioned. The longest logic delay path encountered for the 4-bit counter, such that its value to become stable, is 12 NAND gates.

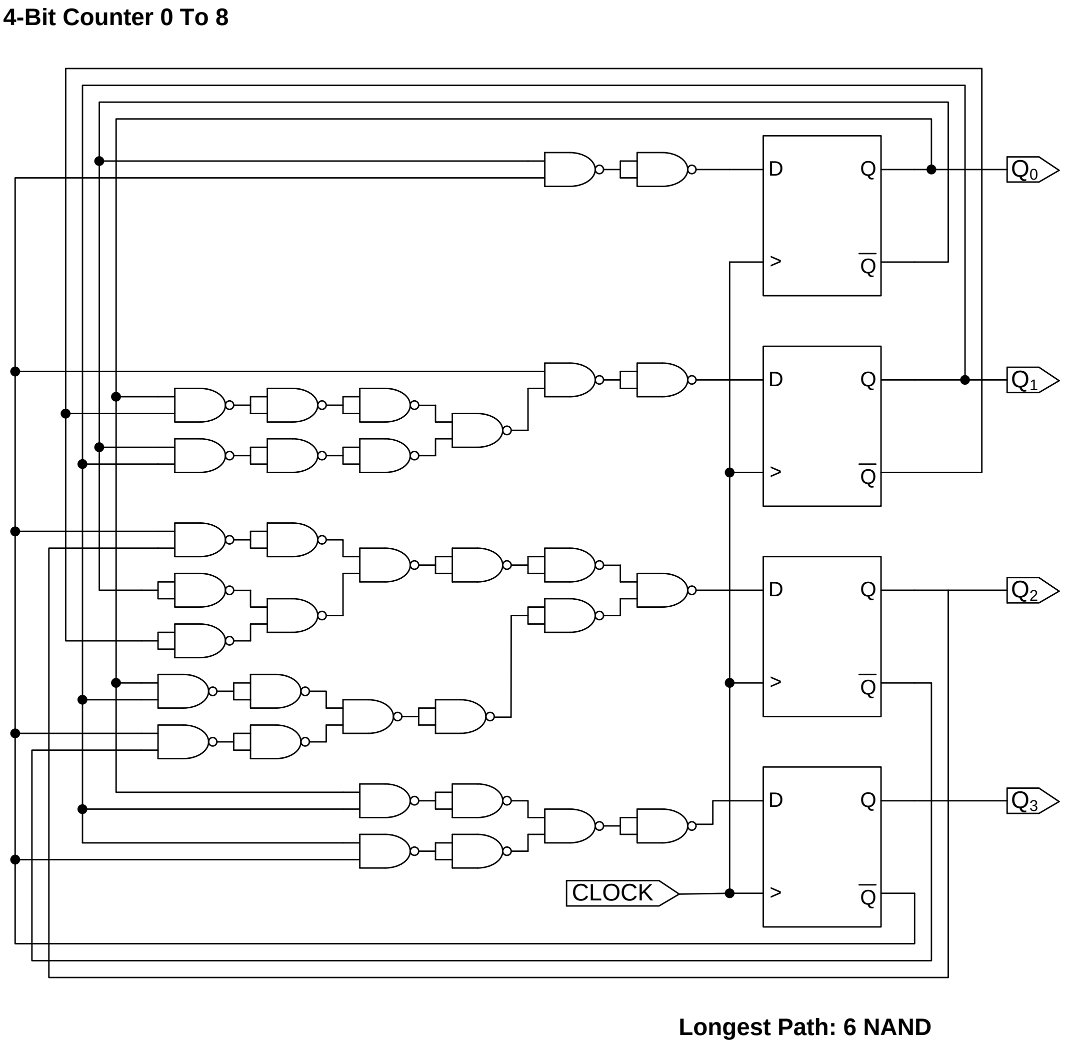
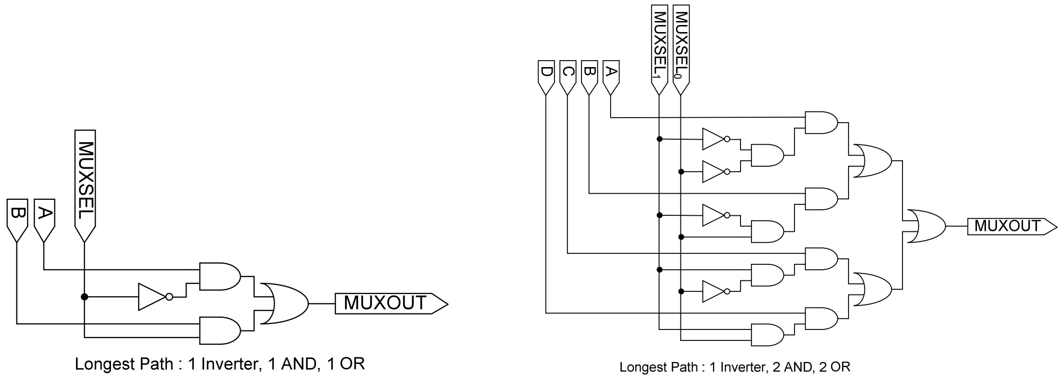


Figure 44 A 4-Bit Counter that Counts from 0 to 8.

3.3.5 Multiplexer and De-Multiplexer

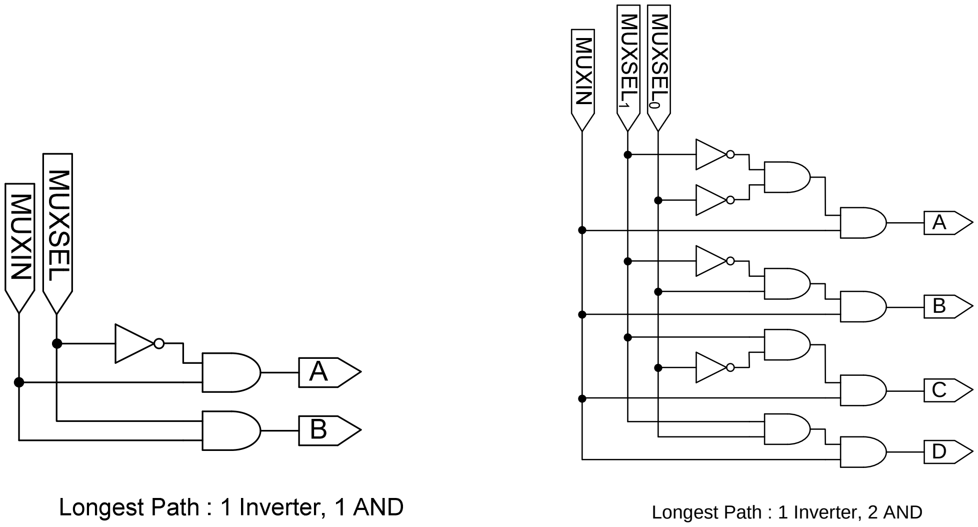
Multiplexer (MUX) is used to select different digital inputs and routed onto a single line that transmits the digital data into a common destination [137]. The MUX usually has a few digital input signals and only one digital output signal. By using the data-select inputs, the MUX allows particular digital input signal to be selected. Then the selected digital input signal will become the digital output signal. Figure 45 shows single bit 2-To-1 and 4-To-1 MUX. From these MUXs, the longest logic gates delay path encounter will be 5 NAND gates (1 INV. 1 AND, 1 OR) and 9 NAND gates (1 INV, 2 AND, 2 OR) respectively.


B
A
MUXSEL
MUXOUT
Longest Path : 1 Inverter, 1 AND, 1 OR
D
C
B
A
MUXSEL$_1$
MUXSEL$_0$
MUXOUT
Longest Path : 1 Inverter, 2 AND, 2 OR

(a) 2-To-1 MUX.

(b) 4-To-1 MUX.

Figure 45 Multiplexer logic circuit diagram.


MUXIN
MUXSEL
A
B
Longest Path : 1 Inverter, 1 AND
MUXIN
MUXSEL$_1$
MUXSEL$_0$
A
B
C
D
Longest Path : 1 Inverter, 2 AND

(a) 1-To-2 DEMUX.

(b) 1-To-4 DEMUX.

Figure 46 De-Multiplexer logic circuit diagram.

As for the Demultiplexer (DEMUX), it has the opposite function of MUX [137]. Instead, it takes one digital input signal and distributes to a particular number of digital output signals. Figure 46 show the logic circuit diagram of single bit 1-To-2 and 1-To-4 DEMUX. The longest logic gates delay encounter are 3 NAND gates (1 INV, 1 AND) and 5 NAND gates (1 INV, 2 AND) respectively. The single bit MUX and DEMUX are actually the basic building blocks for the 11-bit 2-To-1 MUX, 4-To-

1 MUX and 1-To-4 DEMUX. The complete logic block diagram for these MUXs are shown in Figure 47, Figure 48 and Figure 49 respectively.

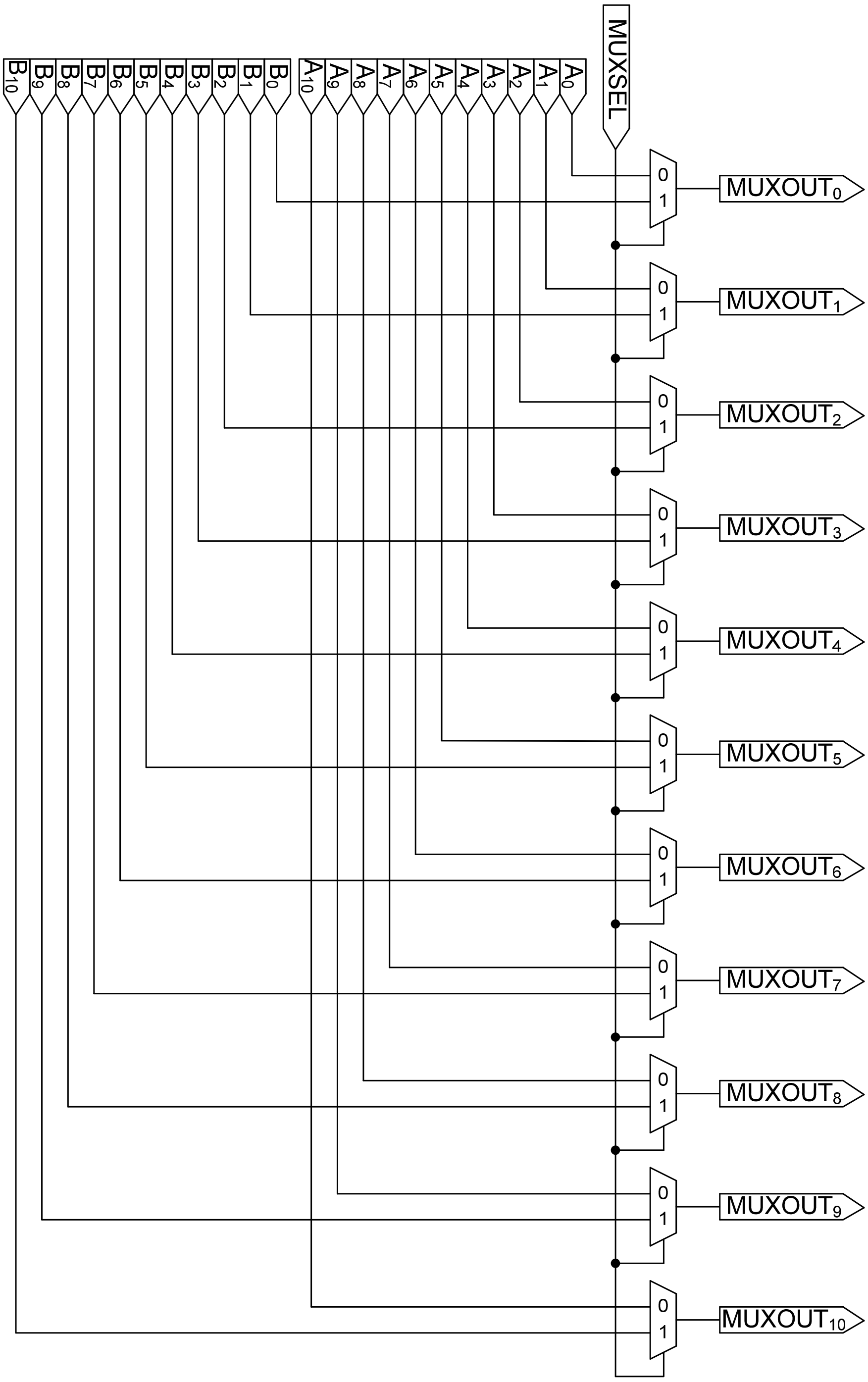


Figure 47 2-To-1 11-bit Multiplexer logic block diagram.

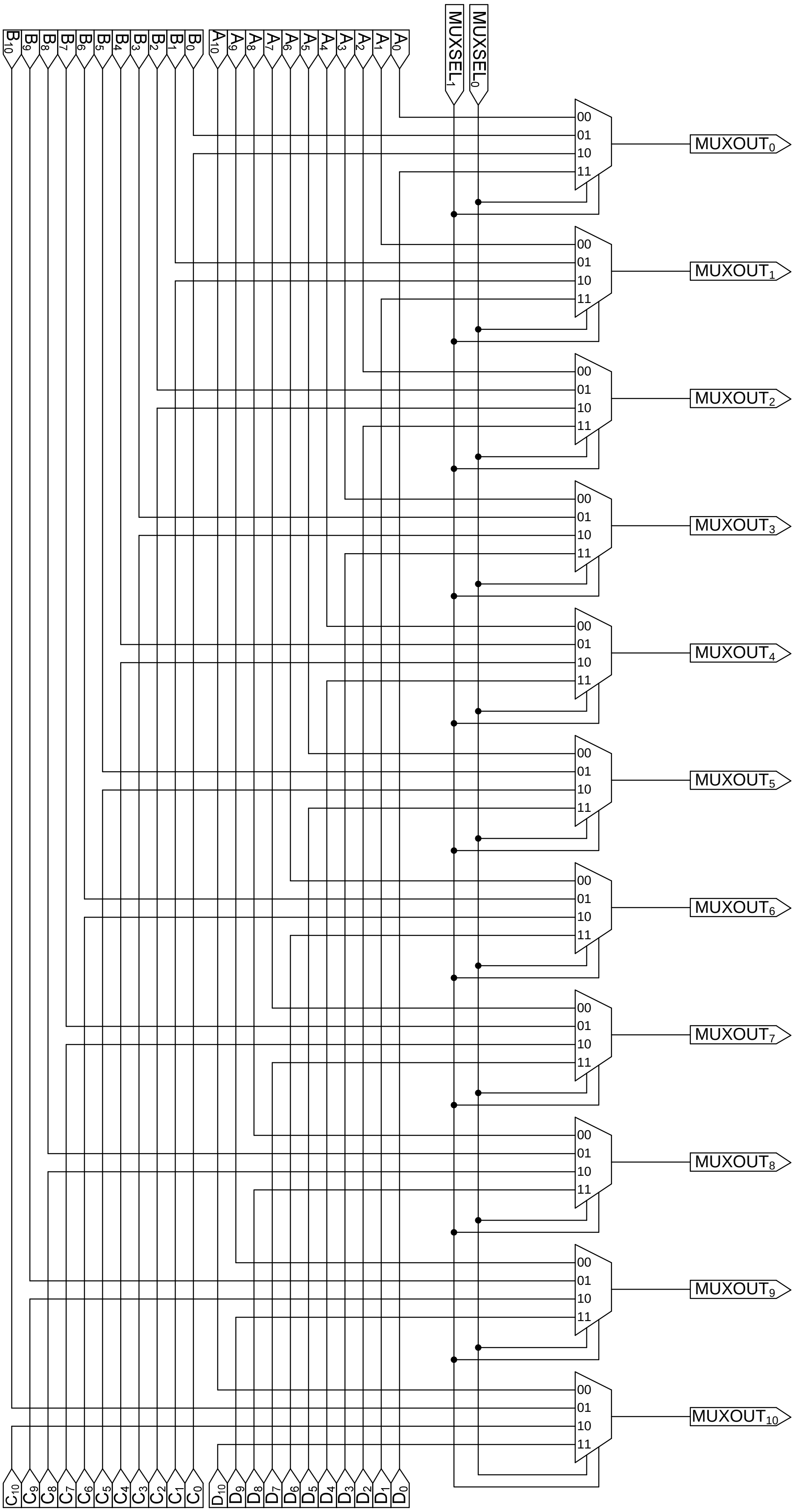


Figure 48 4-To-1 inputs 11-bit Multiplexer logic block diagram.

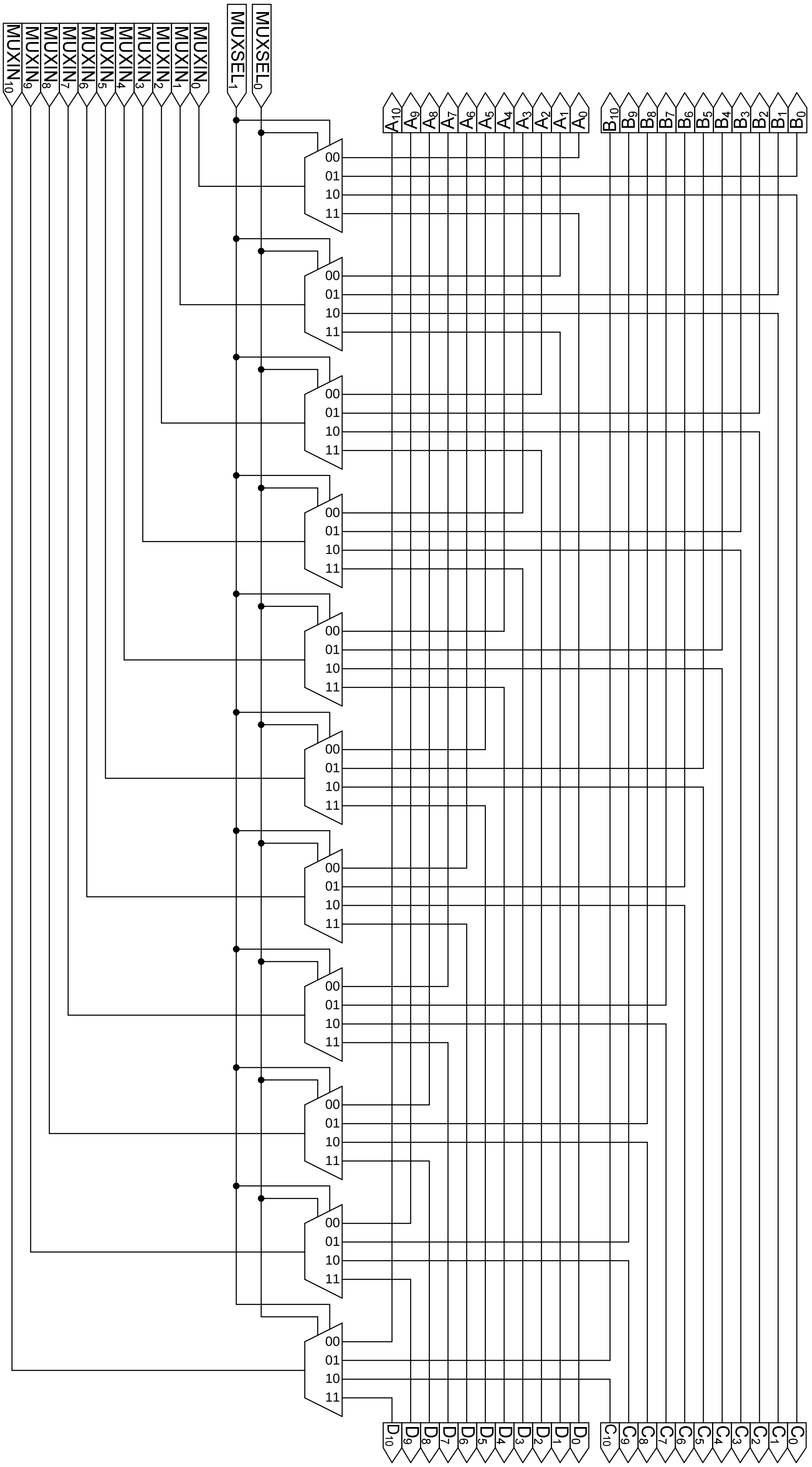

MUXIN$_{10}$
MUXIN$_{9}$
MUXIN$_{8}$
MUXIN$_{7}$
MUXIN$_{6}$
MUXIN$_{5}$
MUXIN$_{4}$
MUXIN$_{3}$
MUXIN$_{2}$
MUXIN$_{1}$
MUXIN$_{0}$
MUXSEL$_{1}$
MUXSEL$_{0}$
A$_{10}$ A$_{9}$ A$_{8}$ A$_{7}$ A$_{6}$ A$_{5}$ A$_{4}$ A$_{3}$ A$_{2}$ A$_{1}$ A$_{0}$
B$_{10}$ B$_{9}$ B$_{8}$ B$_{7}$ B$_{6}$ B$_{5}$ B$_{4}$ B$_{3}$ B$_{2}$ B$_{1}$ B$_{0}$
00
01
10
11
D$_{10}$ D$_{9}$ D$_{8}$ D$_{7}$ D$_{6}$ D$_{5}$ D$_{4}$ D$_{3}$ D$_{2}$ D$_{1}$ D$_{0}$
C$_{10}$ C$_{9}$ C$_{8}$ C$_{7}$ C$_{6}$ C$_{5}$ C$_{4}$ C$_{3}$ C$_{2}$ C$_{1}$ C$_{0}$

Figure 49   1-To-4 11-bit Demultiplexer logic block diagram.

3.3.6 Estimated Longest Logic Gates Delays

Based on the proposed new DWT CRS MISC architecture, as described in Section 3.2, the longest path delay encountered is listed in Table 5. This includes counter, control signals logic circuit, R OUT DEMUX, INV block (Inverters), ALU_B MUX, Adder, MDR IN MUX and MDR. By referring to Fujitsu 300mm FAB process [138], a 90nm CMOS NAND gate, with considering interconnect load, has gate delay of 30.8 ps. With the known NAND gate delay, an estimated of 102 NAND gates delay is expected that constitutes to time delay of 3.1416 ns, as calculated in Equation (15). Next, Equation (16) shows the estimated maximum operating frequency of the propose architecture will be 318.3091 MHz. The estimated maximum operating frequency only considers the combinational logic circuit delay. However, this does not include the Memory (Block RAM) used in this designed architecture. As a result, the maximum operating frequency will be much lower to the estimated value.

Table 5 Sequence of Logic components in longest delay path.

Sequence No.	Logic Components	Logic Gates Delay	NAND Gates Delay
1	Counter	12 NAND	12
2	Control Signals Circuit	1 AND, 2 OR	6
3	R OUT DEMUX	1 INV, 2 AND	5
4	Inverter (INV)	1 INV	1
5	ALU_B MUX	1 INV, 2 AND, 2 OR	9
6	Adder Block	54 NAND	54
7	MDR IN MUX	1 INV, 2 AND, 2 OR	9
8	MDR	2 INV, 4 NAND	6
		Total	**102**

$$\begin{aligned}\text{Total combinational delay} &= \text{Number of NAND gates x 1 NAND gate delay}\\ &= 102 \text{ x } 30.8 \text{ x } 10^{-12}\\ &= 3.1416 \text{ x } 10^{-9}\text{ s}\\ &= 3.1416\text{ ns}\end{aligned} \tag{15}$$

$$\begin{aligned}\text{Maximum Operating Frequency} &= \frac{1}{\text{Combinational Delay}}\\ &= \frac{1}{3.1416 \text{ x } 10^{-9}}\\ &= 318.3091 \text{ x } 10^{6}\text{ Hz}\\ &= 318.3091\text{ MHz}\end{aligned} \tag{16}$$

3.3.7 Control Signals Timing Waveforms

The expected control signals timing waveforms are described based on the Boolean Logic Equation and Truth Table mentioned in Section 3.3. There are two different situations on the control signals outputs. In Figure 50, the control signals will follow this timing waveforms when the DWT CRS MISC is executing SBN Instruction that does not have negative result. This condition also applies to the execution of GF, XOR and 11TO8 Instructions.

When negative result is obtain from SBN Instruction, the timing waveforms shown in Figure 51, will only be applicable. Both control signals timing waveforms are quite similar but with a difference that occurs during Clock Cycle 7 for PC_WRITE signal. When negative results are produced while executing the SBN Instruction, the PC_WRITE signal will be set to High (Logic 1) during Clock Cycle 7. This will allow the newly changed Program Counter (PC) values to be written into the PC register. Thus not executing the subsequent programme instruction. Subsequently, it will execute the new 'targeted' programme instruction. However, the PC_WRITE signal will be Low (Logic 0) during Clock Cycle 7 when no negative results are obtained or non-SBN Instructions are executed.

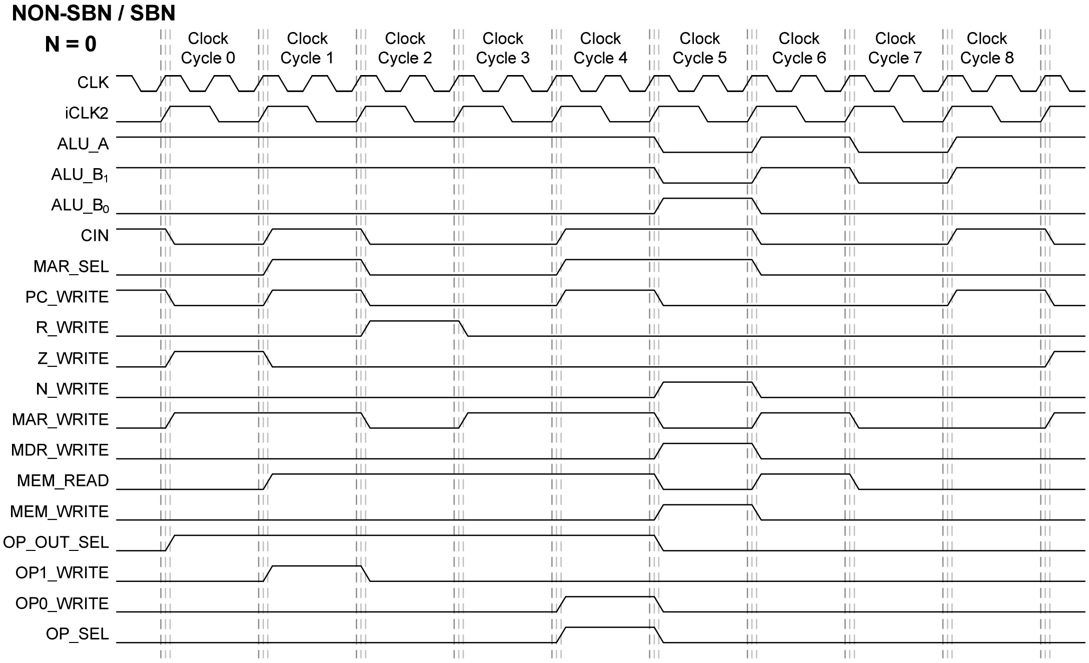


Figure 50 Control Signals generated at particular Clock Cycle, non-SBN / N=0 (SBN).

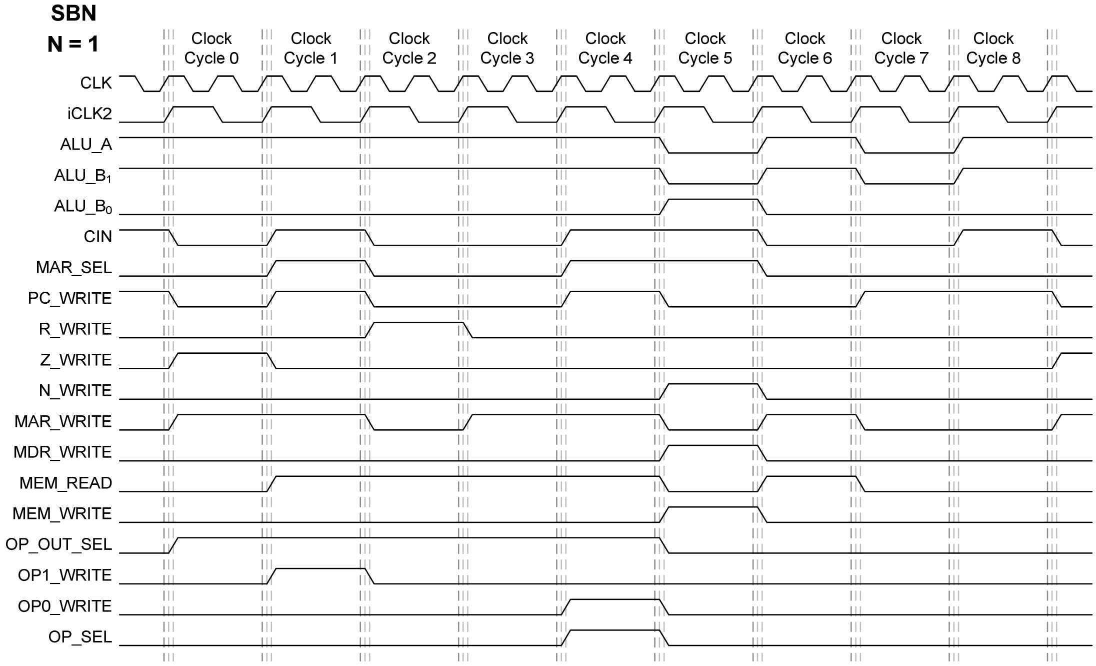


Figure 51 Control Signals generated at particular Clock Cycle, N=1 (SBN).

3.3.8 Data Flow in DWT CRS MISC Architecture

Based on these control signals, the path that data flows in the proposed DWT CRS MISC architecture are not the same for each Clock Cycles. Figure 52 to Figure 64 illustrate the data paths that are active from Clock Cycle 0 to Clock Cycle 8. These active data paths are shown in dark black colour in the MISC architecture while the inactive data paths are those in light gray colour. There are data paths in blue colour that represents active paths but these are not the main focus in these figures. The reason is that the blue colour active paths will not have any significant impact on the data written into the registers and Memory. These registers and Memory are not in the write state and will not overwrite its current value with any new input data.

Besides having the main Clock, there is an input clock which is double the frequency of the main Clock. This clock is also required to drive the Memory such that it is able output the data before the next rising edge of main Clock. The data will be output from the Memory during the failing edge of the main Clock. The frequency of input clock is divided by two to give the main Clock, that drives all the operations in DWT CRS MISC architecture.

During Clock Cycle 0, the Program Counter (PC) value of the PC register is output to the Memory Address Register (MAR) through two Multiplexers (MUXs) and an ADDER block. As shown in Figure 52, the PC value is stored into MAR and is available to be read after the next rising edge Clock Cycle 1. The PC value is the memory address locations for the first line of the programme instruction to be read from the Memory.

At Clock Cycle 1, the PC value is increased by 1 through the ADDER block with CIN input as Logic 1. Then the increased PC value is stored back into the PC register at the rising edge Clock Cycle 2. This is to have the PC value to become memory address location for second line of the programme instruction. Meanwhile, at failing edge Clock Cycle 1, the MAR provides the memory address location for the Memory to read and output the first line of programme instruction. The programme instruction contains the information of 1st Operand (*A*) address location. Figure 53 shows that the first line of programme instruction (read from the Memory) consists of 1st OPCODE at the Most Significant Bit (MSB) and the 1st Operand (*A*) memory address location.

Once the 1st OPCODE is read from the MSB of the output memory, it is stored into OPCODE1 register at rising edge of Clock Cycle 2. Similarly at rising edge of Clock Cycle 2, the 1st Operand memory address location is then stored into the MAR and this overwrites the currently stored PC value. After rising edge of Clock Cycle 2, the MAR provides the data address location to read the 1st Operand (*A*) value from the Memory. The 1st Operand (*A*) value is read during the falling edge Clock Cycle 2. As shown in Figure 54, the read 1st Operand (*A*) value from the Memory is then stored into the Read (R) register. The 1st Operand value is only stored into the R register during the rising edge of Clock Cycle 3.

As for Clock Cycle 3, the previously increased PC value is output from the PC register to the MAR again through two MUXs and the ADDER block. As shown in Figure 55, the PC value is stored into MAR and is available to be read after the next rising edge of Clock Cycle 4. The PC value is the memory address locations of the second line of the programme instruction to be read from the Memory.

During failing edge of Clock Cycle 4, the second line of programme instruction is read from the memory that consists of the 2nd OPCODE at the MSB and the 2nd Operand (*B*) data address location. The 2nd OPCODE is read from the MSB of the output data from the Memory and stored it into OPCODE0 register.

At the rising edge of Clock Cycle 5, the 2nd Operand memory address location is then stored into the MAR which overwrites the currently store PC value, shown in

Figure 56. Meanwhile, the PC value is again increased by 1 through the ADDER block and stored the PC value back to the PC register at the rising edge of Clock Cycle 6. At Clock Cycle 5, the MAR provides the data address location to read the 2nd Operand (*B*) value from the Memory. During the falling edge Clock Cycle 5, the 2nd Operand (*B*) value is only output from the Memory. Note that there are four different data paths for the MISC architecture during Clock Cycle 5, which are shown in Figure 57 to Figure 60.

For these four differences, the 1st Operand (*A*) value from the R register and the 2nd Operand (*B*) value are input into the corresponding functional block (GF MULT, XOR, 11TO8, ADDER). The two Operands (*A*, *B*) input into the corresponding functional block are based on the OPCODE1 and OPCODE0 values, which are read from the first two lines of programme instruction. Both OPCODE1 and OPCODE0 are combined together to form a complete output of 2-bit OPCODE, where OPCODE1 is the MSB and OPCODE0 is Least Significant Bit (LSB). For OPCODE is 00, the Operands (*A*, *B*) are input into the GF MULT functional block, which is shown in Figure 57. The output from the GF MULT functional block is the result of GF arithmetic multiplication. As shown in Figure 58, the Operands (*A*, *B*) are input into the XOR functional block when the OPCODE is 01. The output for the XOR functional block is the result of bitwise XOR operation performed onto both input Operands (*A*, *B*). If the OPCODE is 10, then the Operands (*A*, *B*) are input into the ADDER block that performs the SBN instruction. The data paths for executing the SBN instruction is shown in Figure 59. If the OPCODE is 11, then the Operands (*A*, *B*) are input into the 11TO8 functional block that performs the conversion of Operand *B* from 11-bit data to 8-bit data. Figure 60 shows the data path for the Operands (*A*, *B*) input to 11TO8 functional block. The output data from all these functional blocks (GF MULT, XOR, 11TO8, ADDER) is stored into the MDR. The data stored into the MDR will be available after the next rising edge of Clock Cycle 6.

At falling edge Clock Cycle 6, the data stored in the MDR is then input into the Memory. The data input to the Memory is stored at data memory address of Operand *B*, which is output from MAR. Meanwhile, the PC value is also input into the MAR through two MUXs and ADDER, which is shown in Figure 61. The PC value input into the MAR will only be available at the next rising edge Clock Cycle 7. The PC value is actually the memory address location for the third line of the programme instruction that contains the ‘Target Address’.

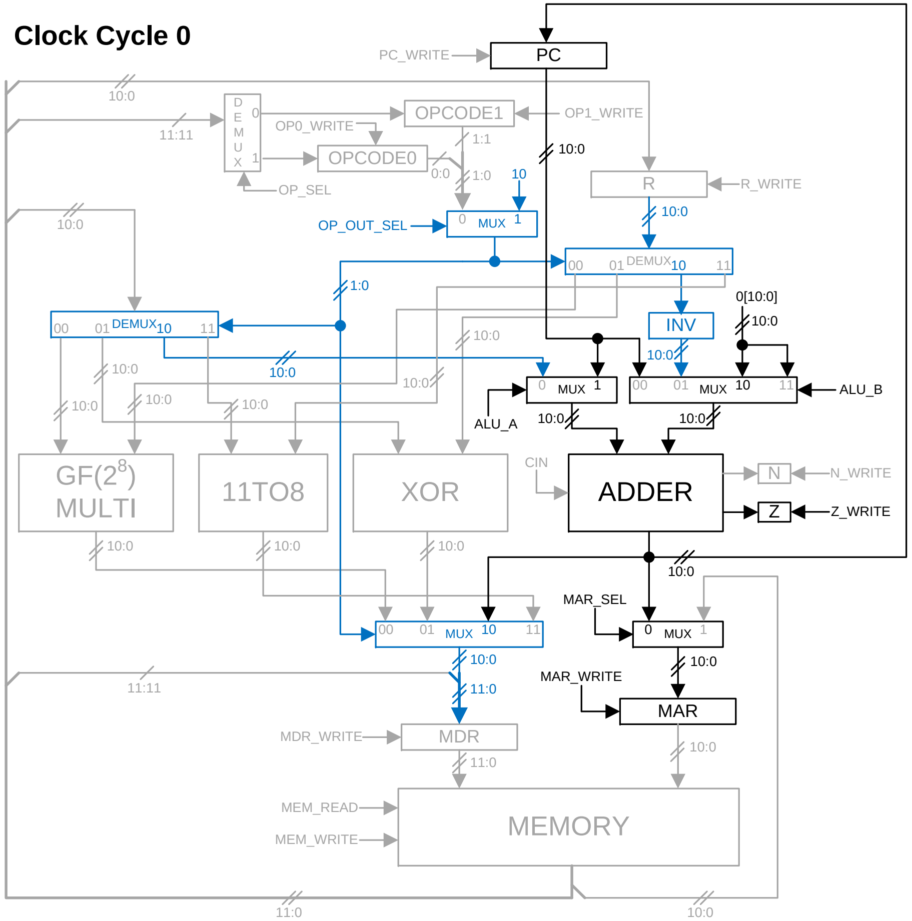


Figure 52 Data flow in DWT CRS MISC at Clock Cycle 0.

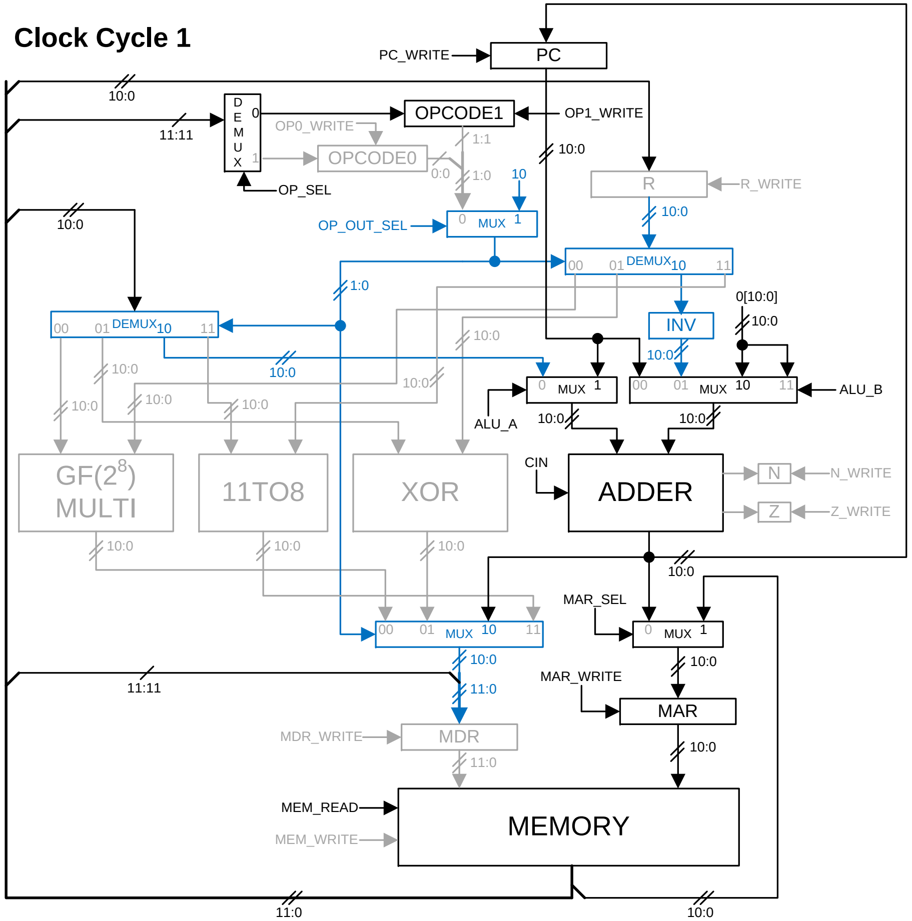


Figure 53 Data flow in DWT CRS MISC at Clock Cycle 1.

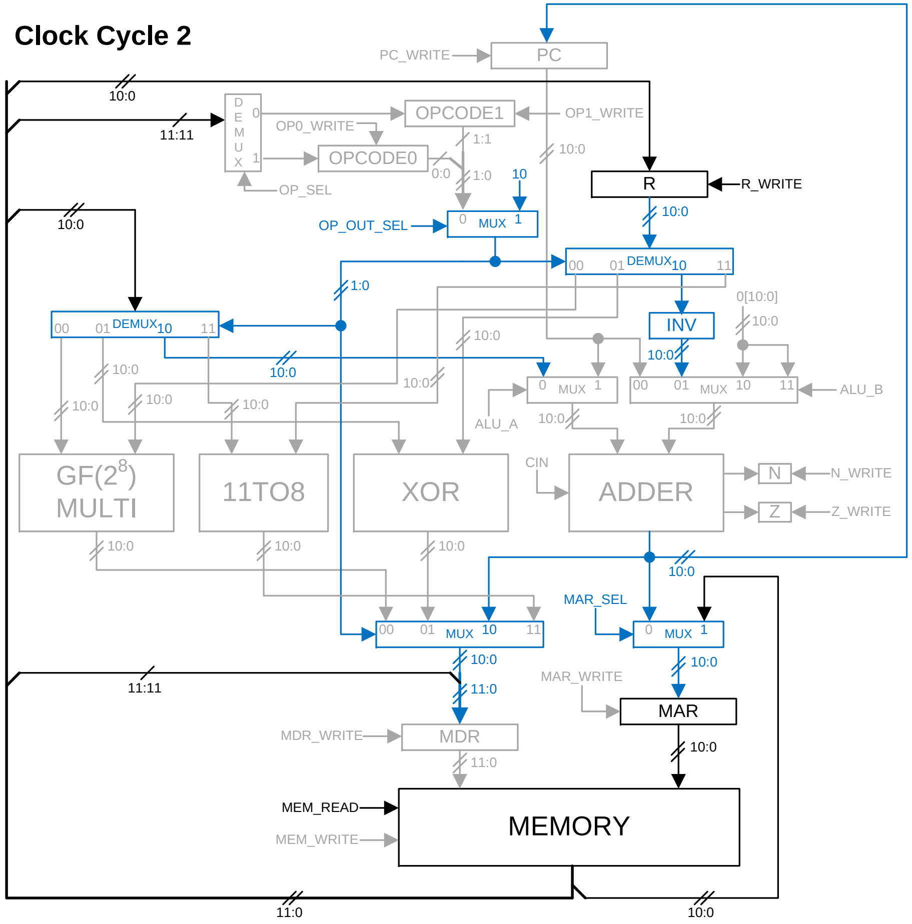


Figure 54 Data flow in DWT CRS MISC at Clock Cycle 2.

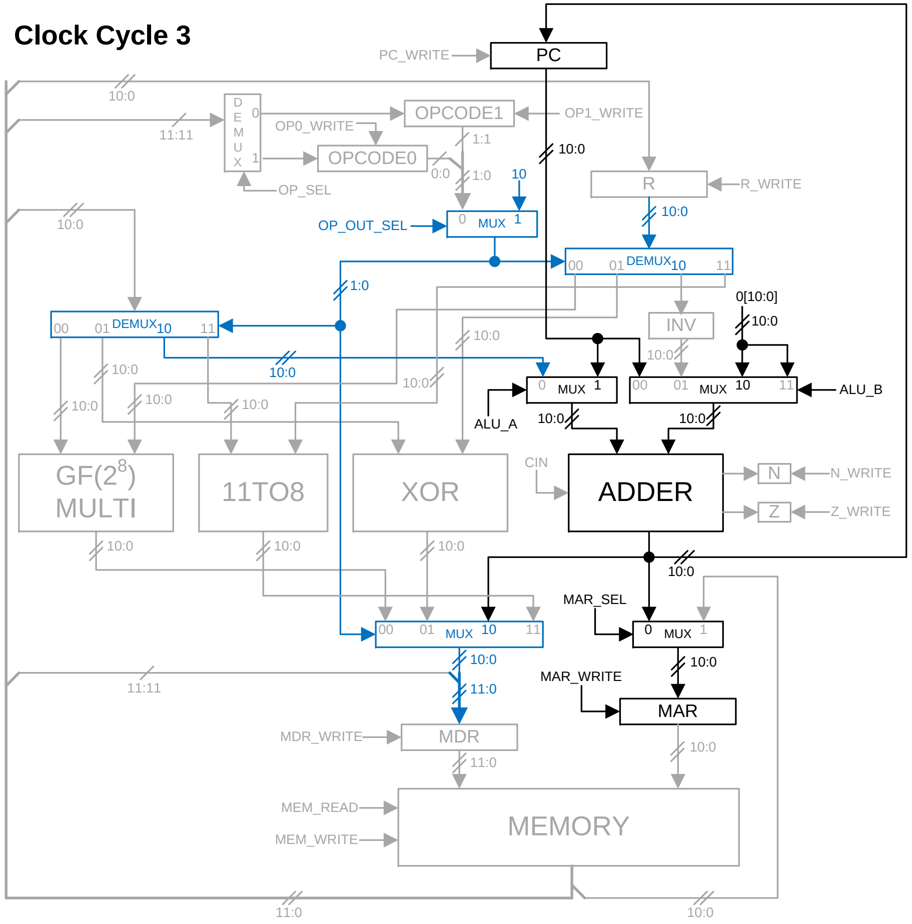


Figure 55 Data flow in DWT CRS MISC at Clock Cycle 3.

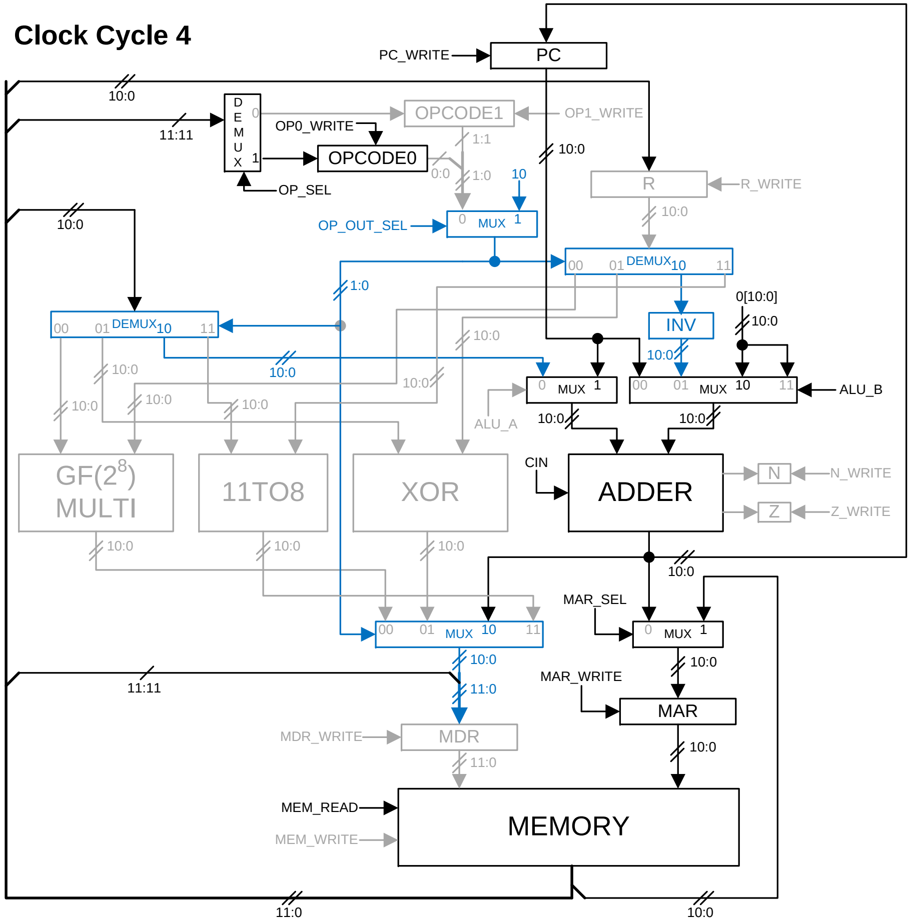


Figure 56 Data flow in DWT CRS MISC at Clock Cycle 4.

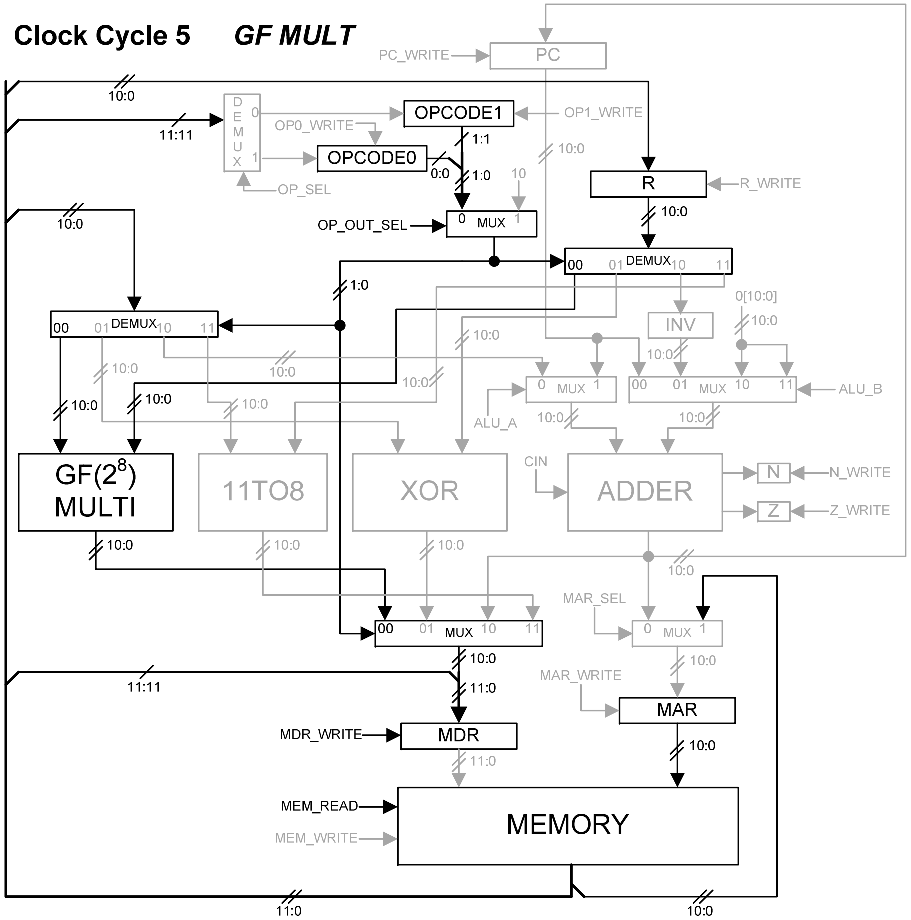


Figure 57 Data flow in DWT CRS MISC at Clock Cycle 5 (GF MULT).

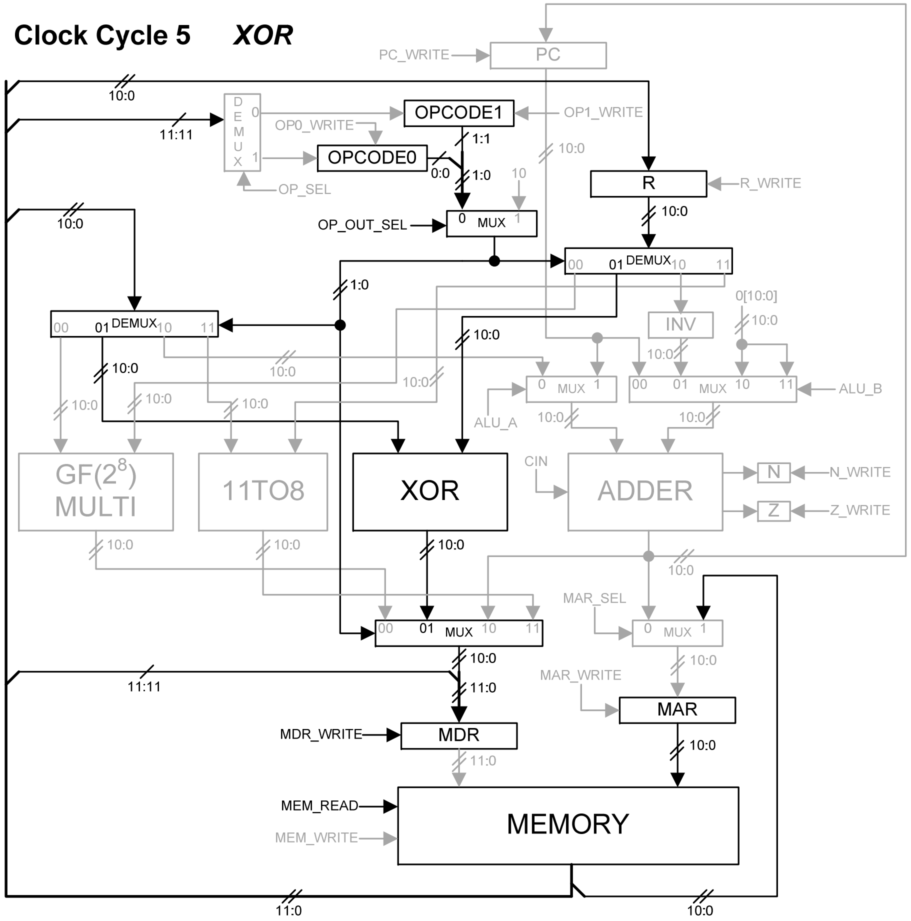


Figure 58 Data flow in DWT CRS MISC at Clock Cycle 5 (XOR).

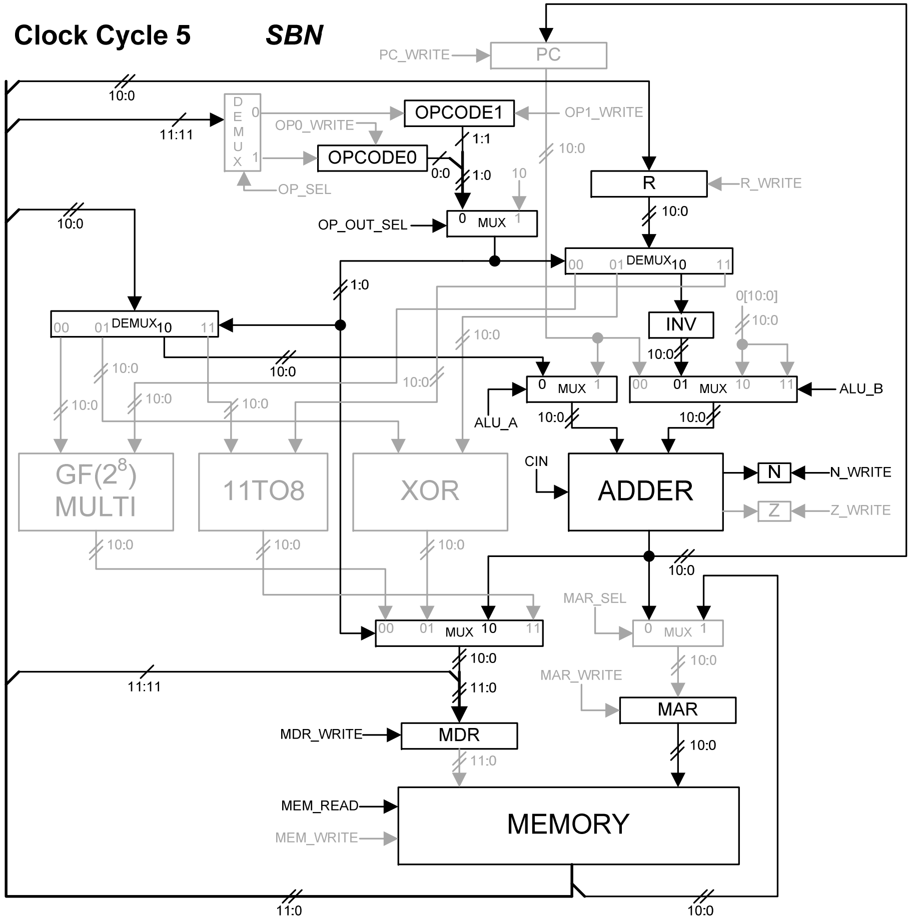


Figure 59 Data flow in DWT CRS MISC at Clock Cycle 5 (SBN).

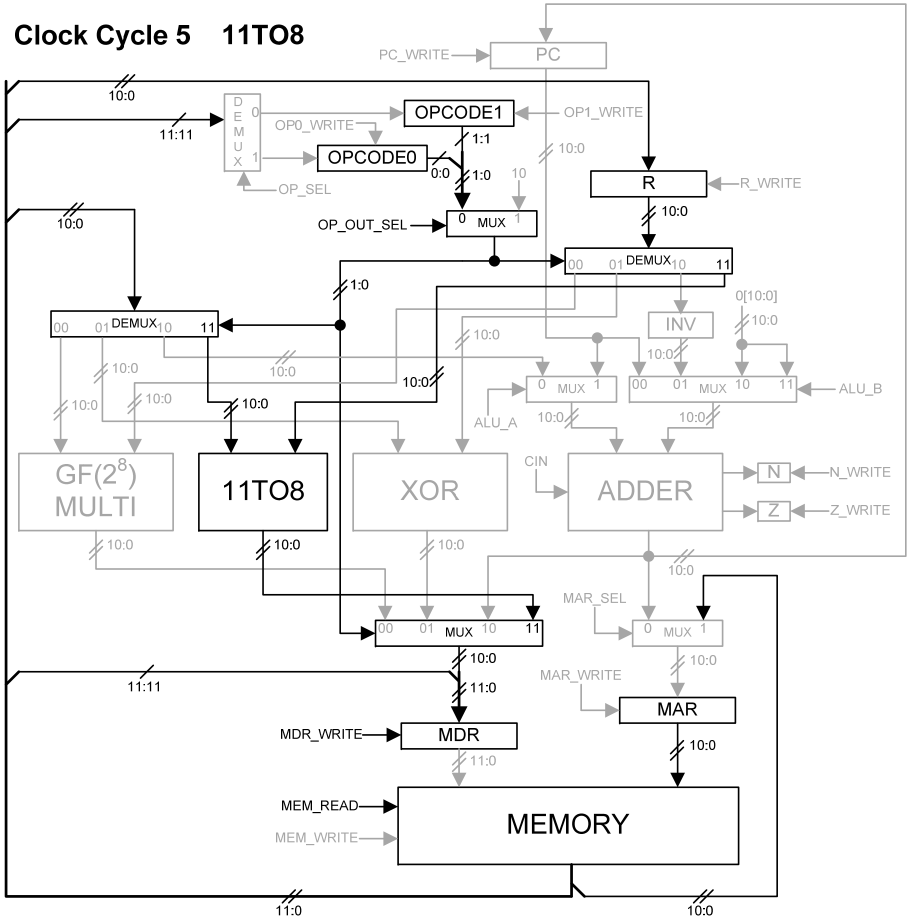


Figure 60 Data flow in DWT CRS MISC at Clock Cycle 5 (11TO8).

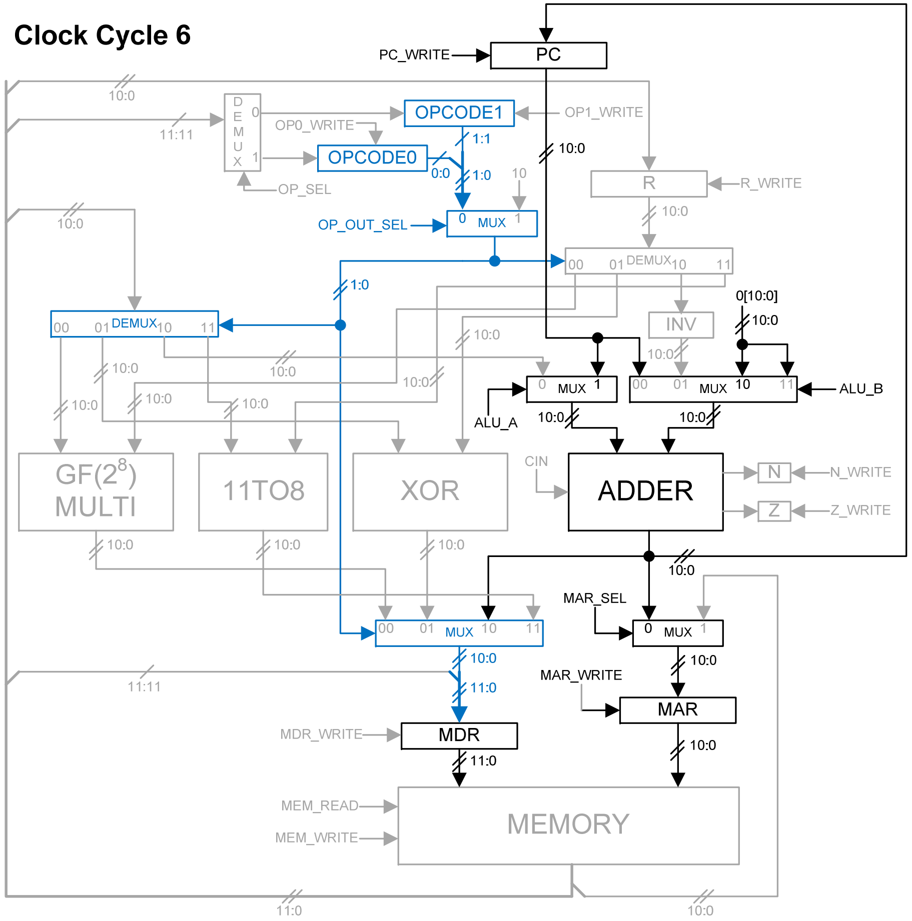


Figure 61 Data flow in DWT CRS MISC at Clock Cycle 6.

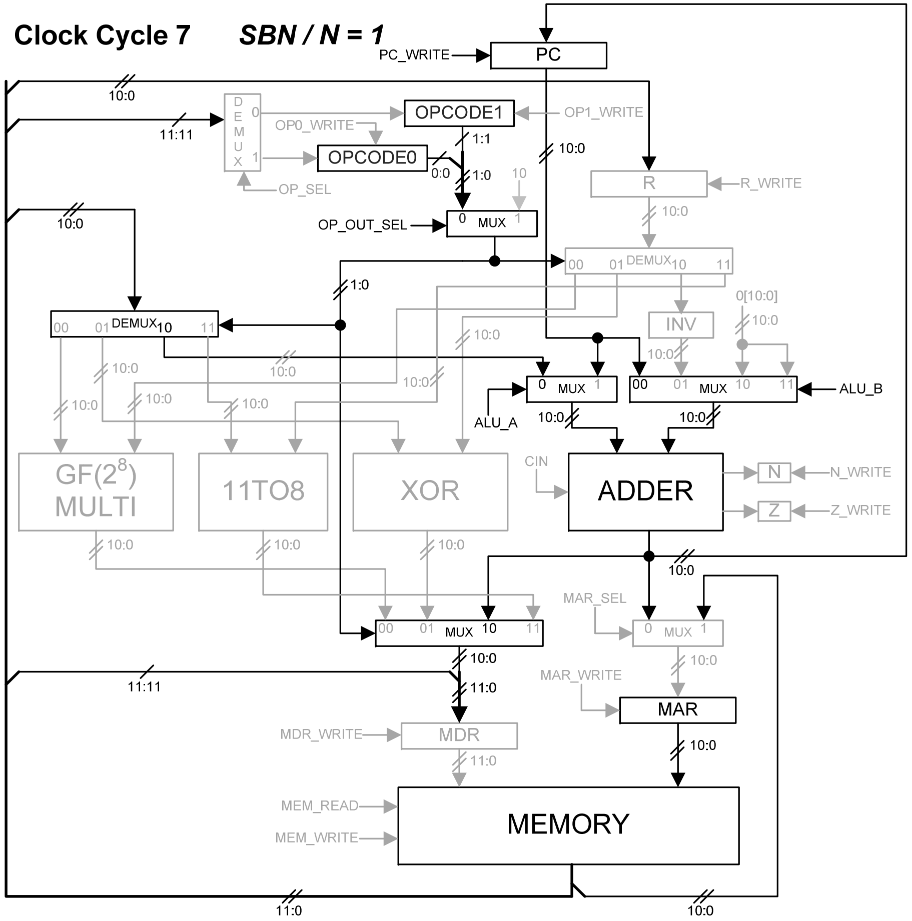


Figure 62 Data flow in DWT CRS MISC at Clock Cycle 7 (N = 1).

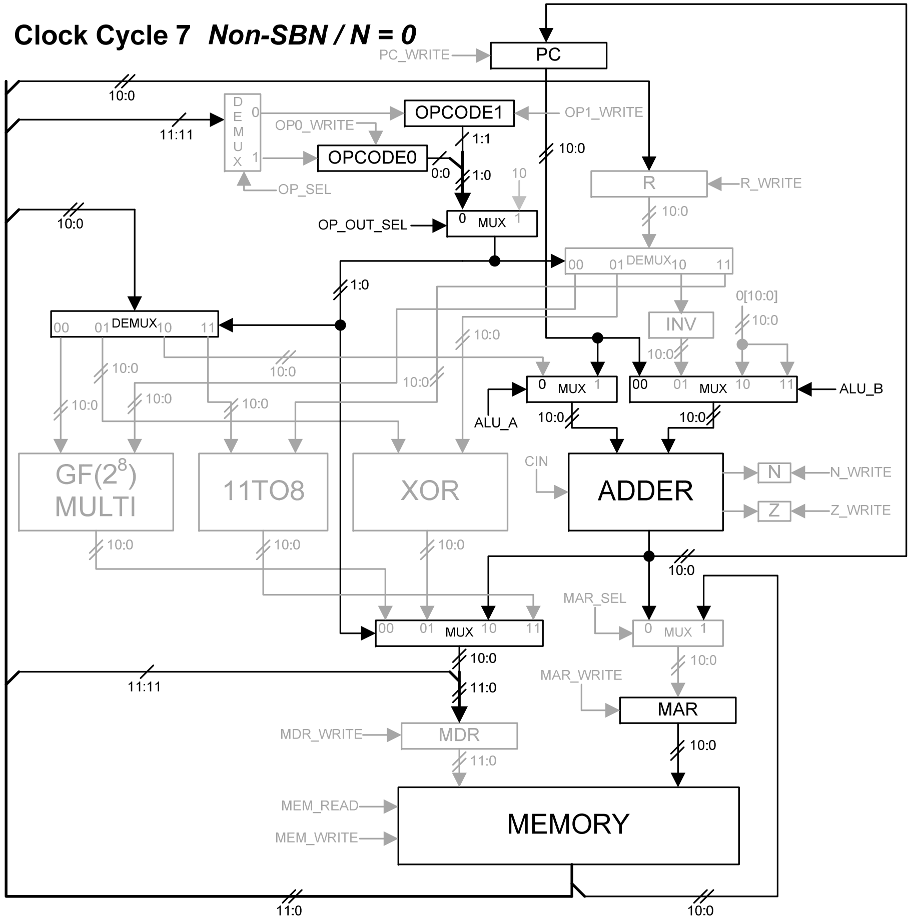


Figure 63 Data flow in DWT CRS MISC at Clock Cycle 7 (Non-SBN/N = 0).

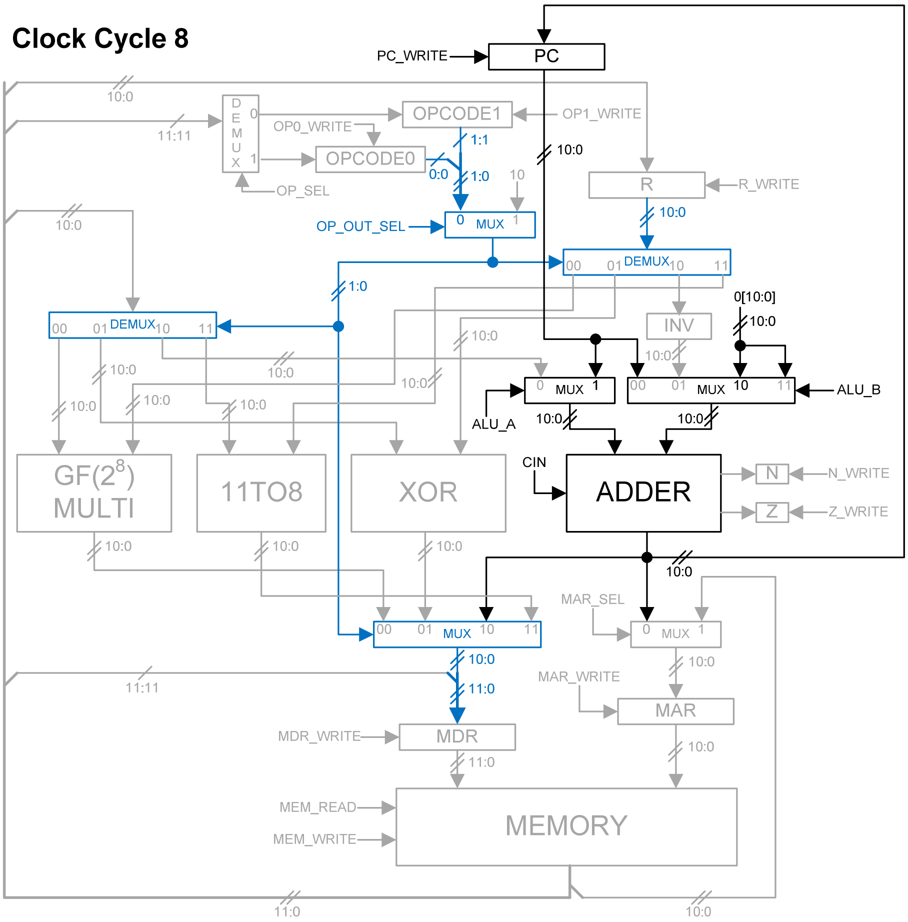


Figure 64 Data flow in DWT CRS MISC at Clock Cycle 8.

At falling edge Clock Cycle 7, the ‘Target Address’ is read from the Memory at the memory address location input from the MAR. When the SBN instruction is executed, for N = 1 case, the ‘Target Address’ is added with the PC value read from the PC register at the ADDER block. The addition of the ‘Target Address’ and the PC value produces the next programme memory address location (that jumps to ‘Target’ programme instruction) that the MISC will execute next. Figure 62 shows the next ‘Target’ programme memory address produced. At the rising edge of Clock Cycle 8, this address is stored into the PC register when negative result is obtained from the SBN instruction (*N* = 1). Next, Figure 63 shows that the next ‘Target’ programme memory address is not stored into the PC register. In this situation, the value inside the PC register will not be modified. This allows the MISC to continue executing the subsequent programme instruction. This particular data paths occurred only for the case when non-SBN instruction is executed or a positive result is obtained from SBN instruction (N = 0).

For the last Clock Cycle 8, the PC value in the PC register is output to the ADDER block and the PC value is increased by 1. Figure 64 shows the increased PC value is input back into the PC register. The PC register will only store the increased PC value at the next rising edge Clock Cycle 0. The increased PC value is the next programme instruction (programme memory location) that will be executed by the MISC architecture. With 9 Clock Cycles, these complete the execution of one programme instruction consists of 3 lines of programme codes.

3.3.9 Timing Diagram

The timing diagrams for each registers and Memory in the DWT CRS MISC architecture are presented in Figure 65 to Figure 79. Based on the Clock Cycle 0 to Clock Cycle 8, each of these timing diagram illustrates how the data are output from the Memory and written into the registers. From the timing diagrams, it can be seen that data are stored into the registers after the rising edge of clock. Note that the timing diagrams shown is just a representation of what logic signals (data) will be expected. Although these timing diagrams show the logic signals delays, that usually occurred in hardware, they do not represent the actual precise logic signals delays.

Besides the registers, the timing diagrams for the MUXs are also shown in Figure 80 to Figure 96. These timing diagrams illustrates how logic signals (data) are selected, from different sources, and then output to registers or to control another MUX and DEMUXs. Next, the timing diagrams for DEMUXs are also indicated in Figure 97 to Figure 108. Each of these timing diagrams shows how the single logic signal, either from Memory or registers, is to be output to the selected outputs. These outputs can either be functional blocks or the registers.

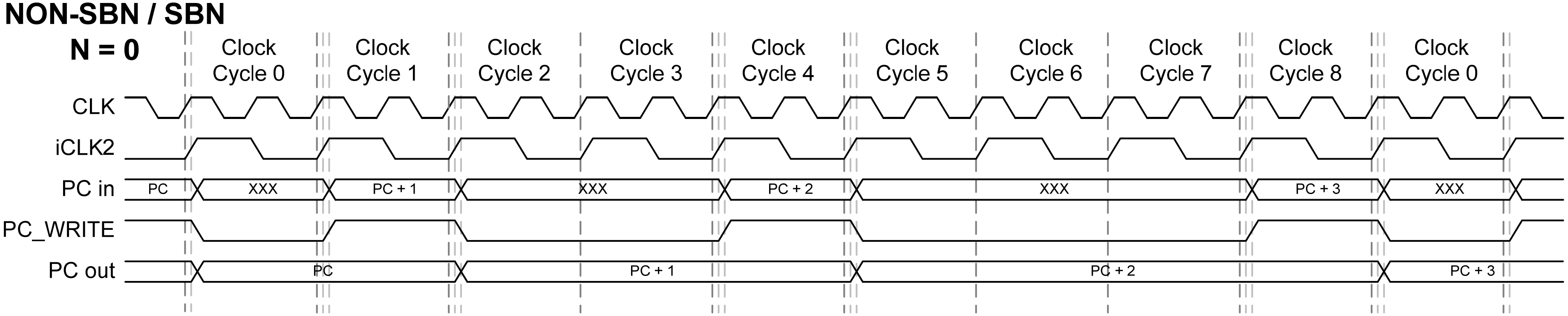


Figure 65 PC register timing diagram for N = 0 (SBN / NON-SBN).

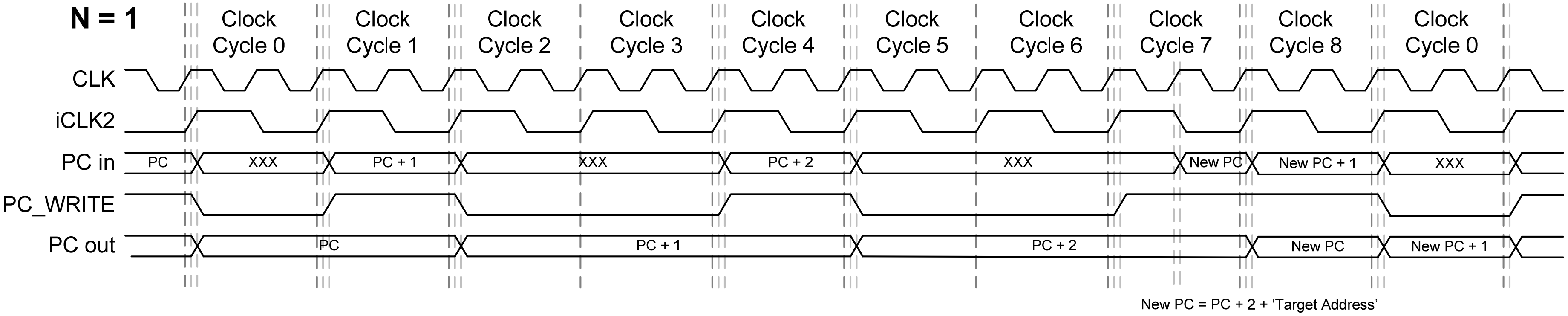


Figure 66 PC register timing diagram for N = 1 (SBN).

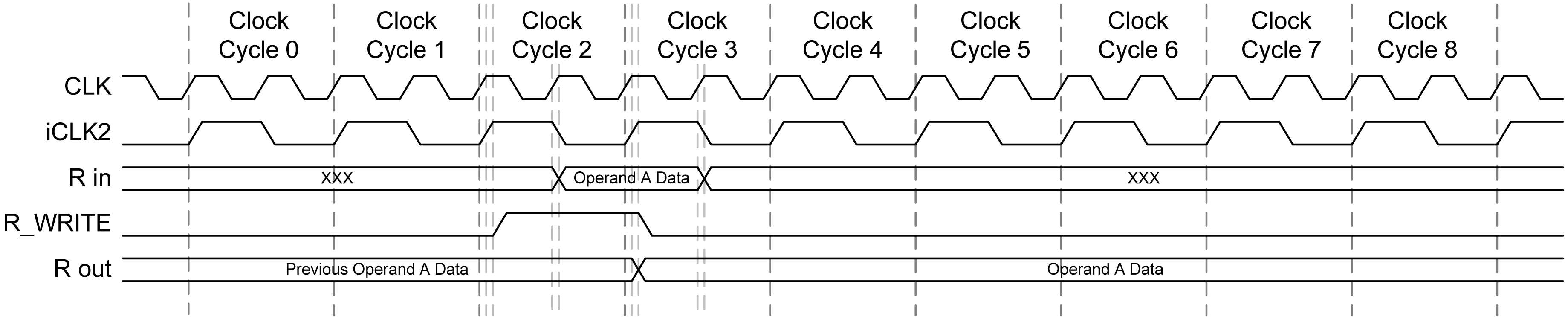


Figure 67   R register timing diagram.

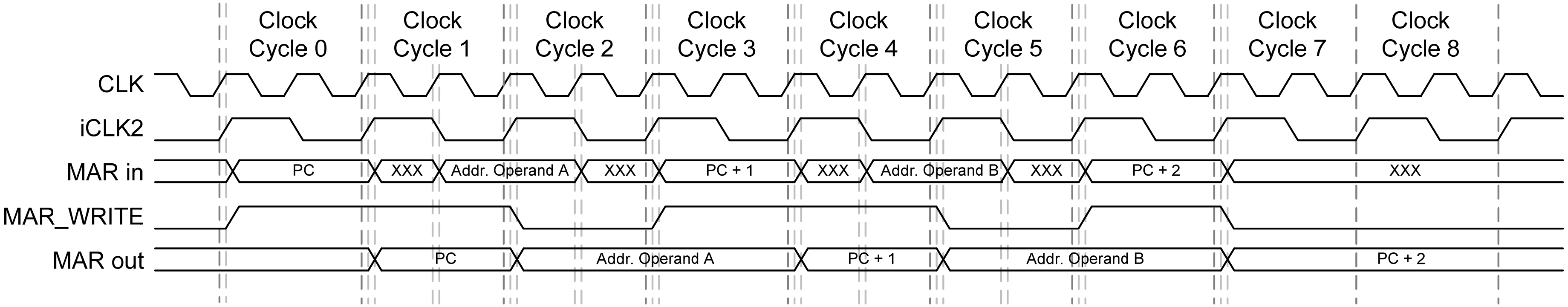


Figure 68   MAR register timing diagram.

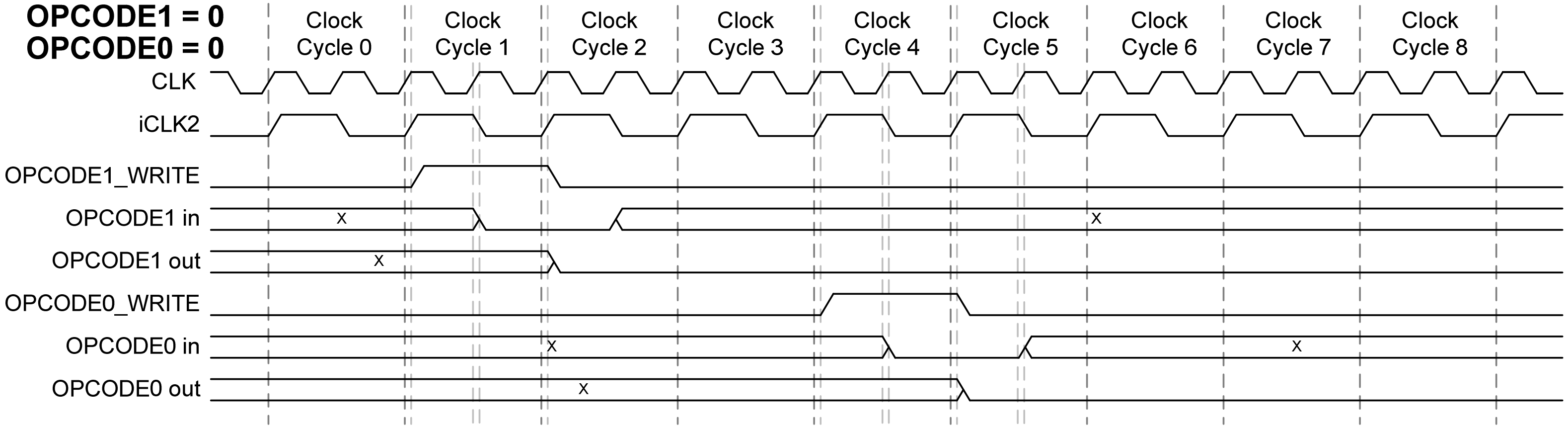


Figure 69 OPCODE1 and OPCODE0 registers timing diagram for GF MULT.

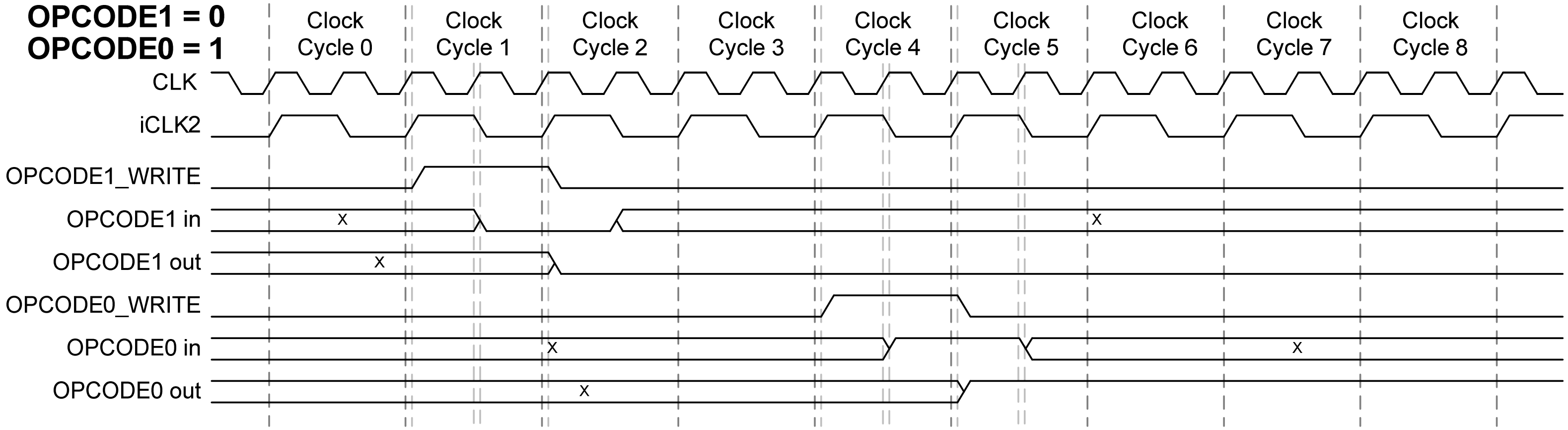


Figure 70 OPCODE1 and OPCODE0 registers timing diagram for XOR.

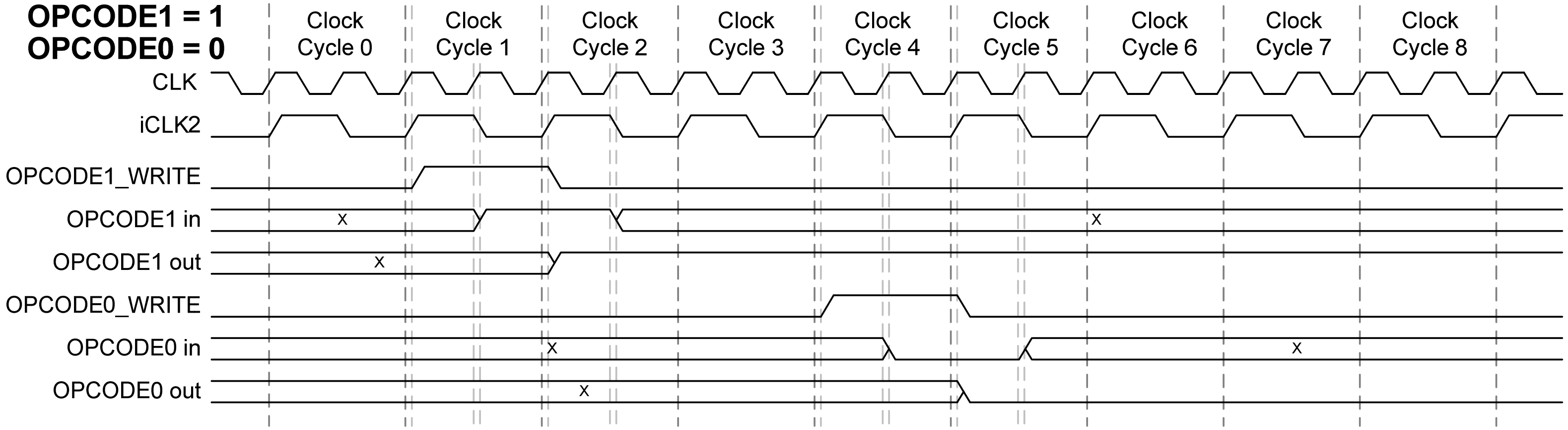


Figure 71 OPCODE1 and OPCODE0 registers timing diagram for SBN.

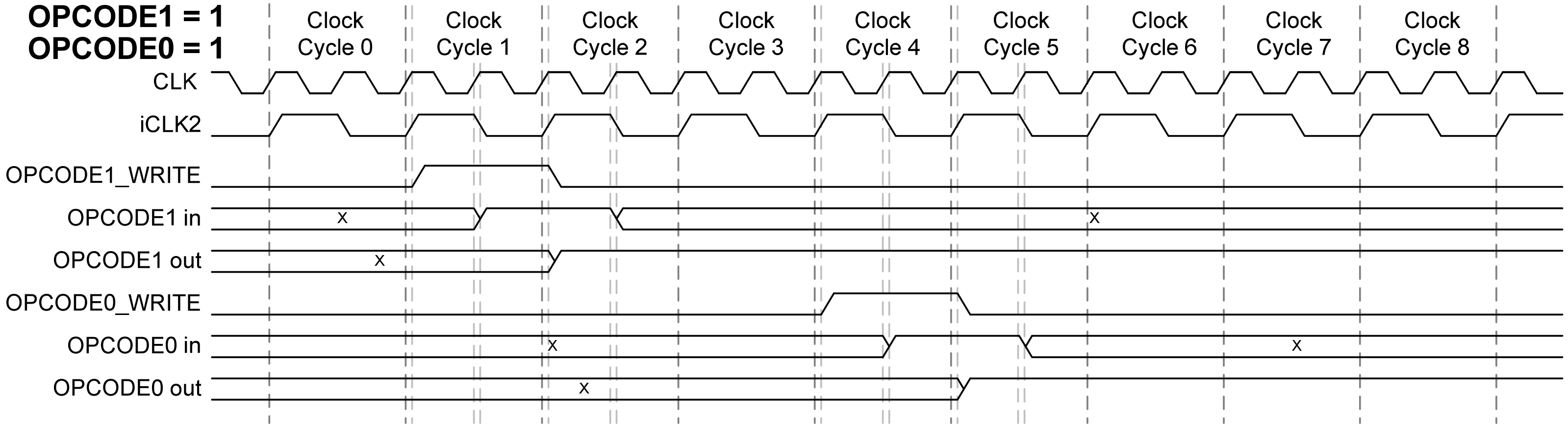


Figure 72 OPCODE1 and OPCODE0 registers timing diagram for 11TO8.

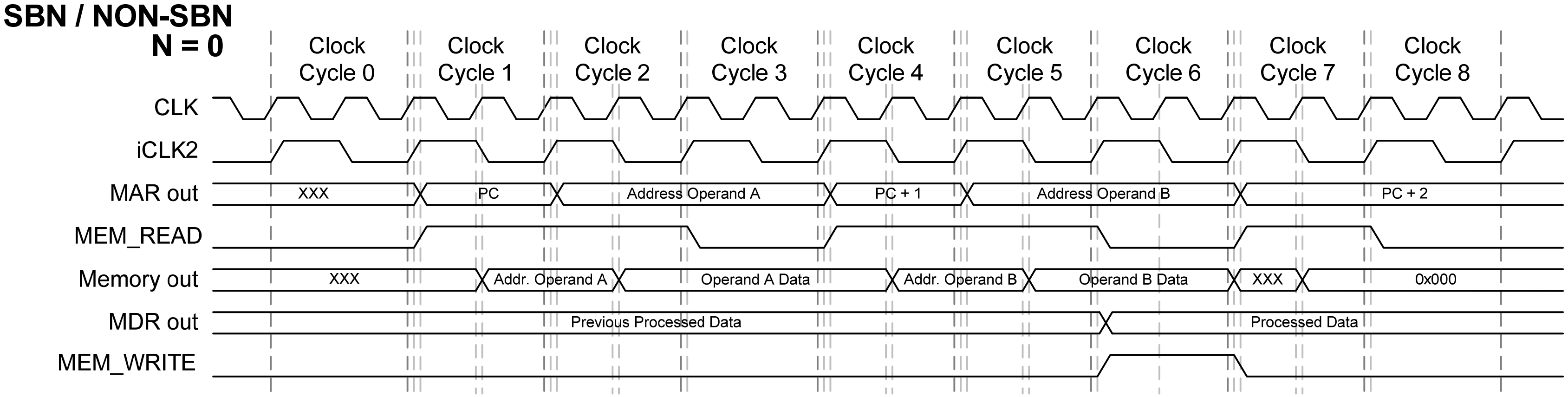


Figure 73 Memory output and input timing diagram for Non-SBN / N=0.

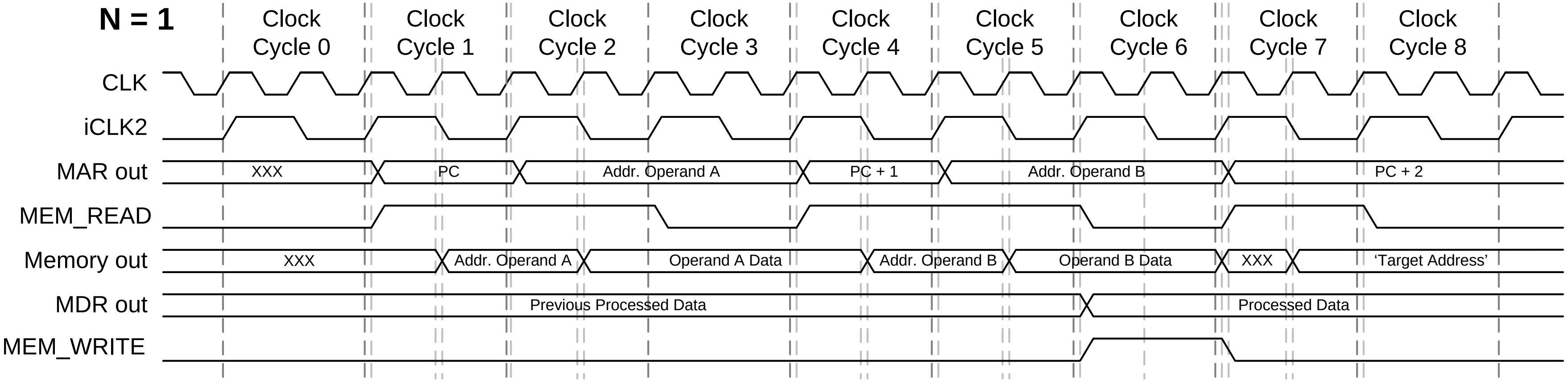


Figure 74 Memory output and input timing diagram for N=1.

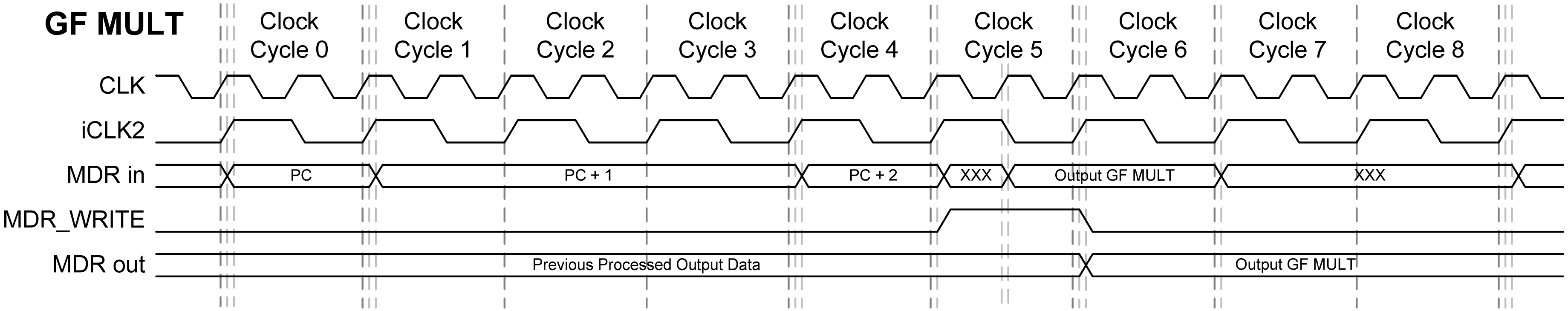


Figure 75 MDR register timing diagram for GF MULT instruction.

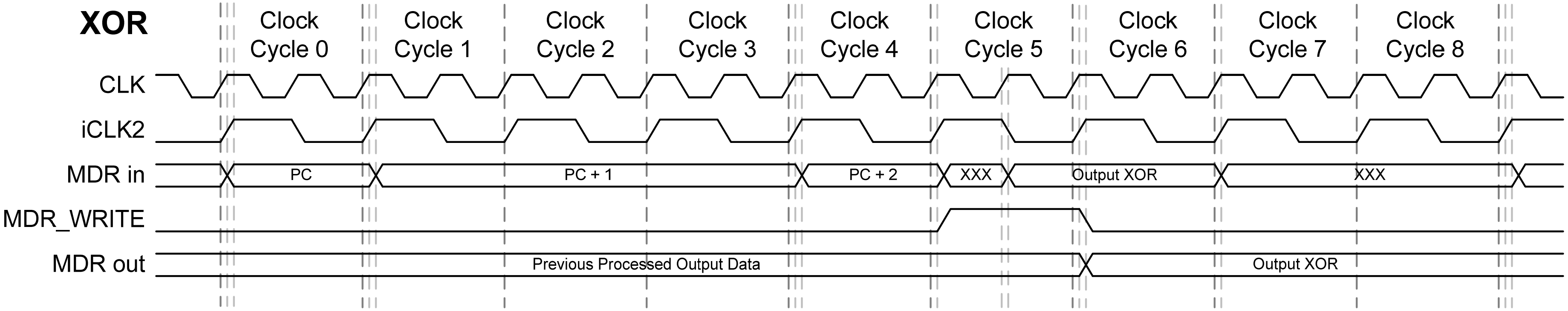


Figure 76 MDR register timing diagram for XOR instruction.

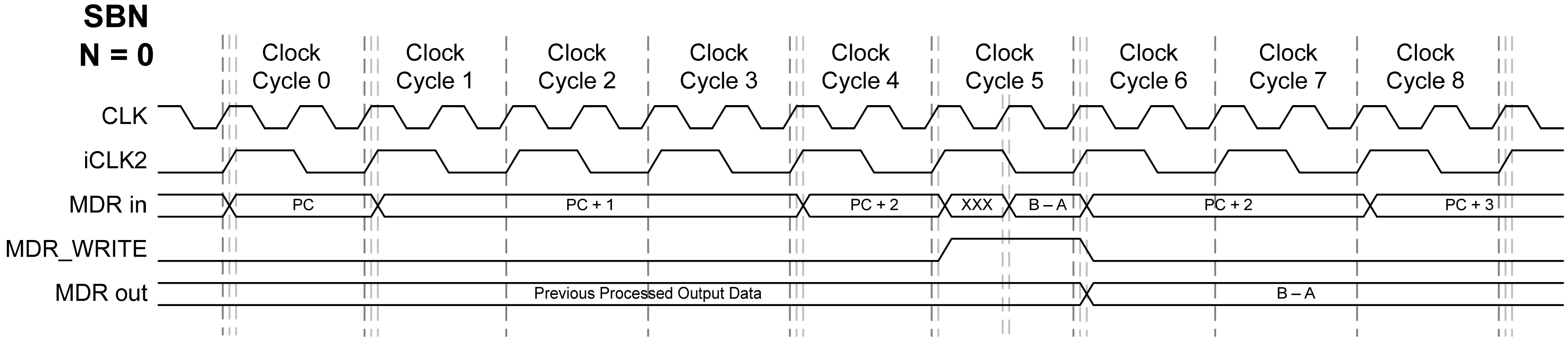


Figure 77 MDR register timing diagram for SBN instruction (N = 0).

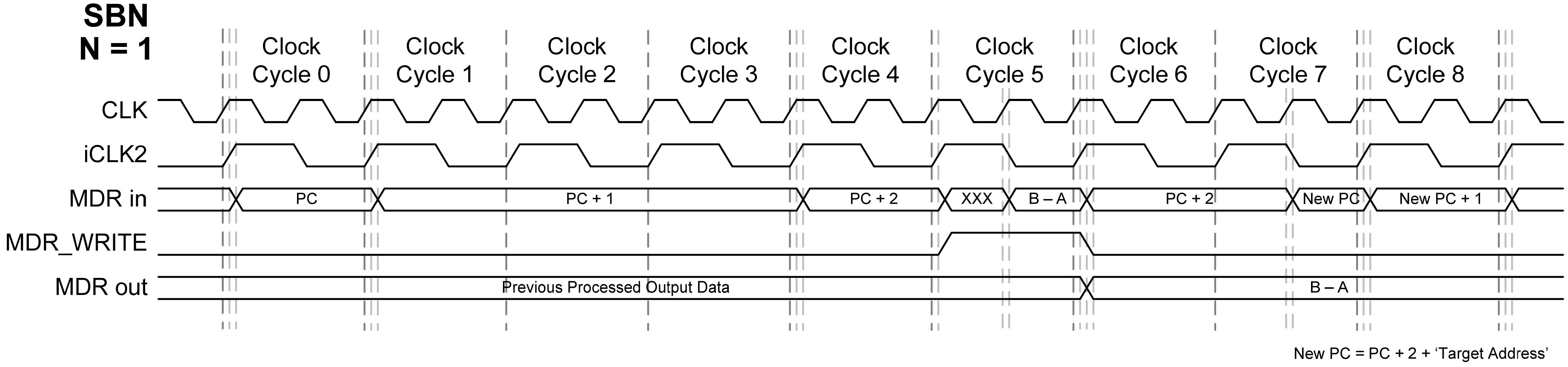


Figure 78 MDR register timing diagram for SBN instruction (N = 1).

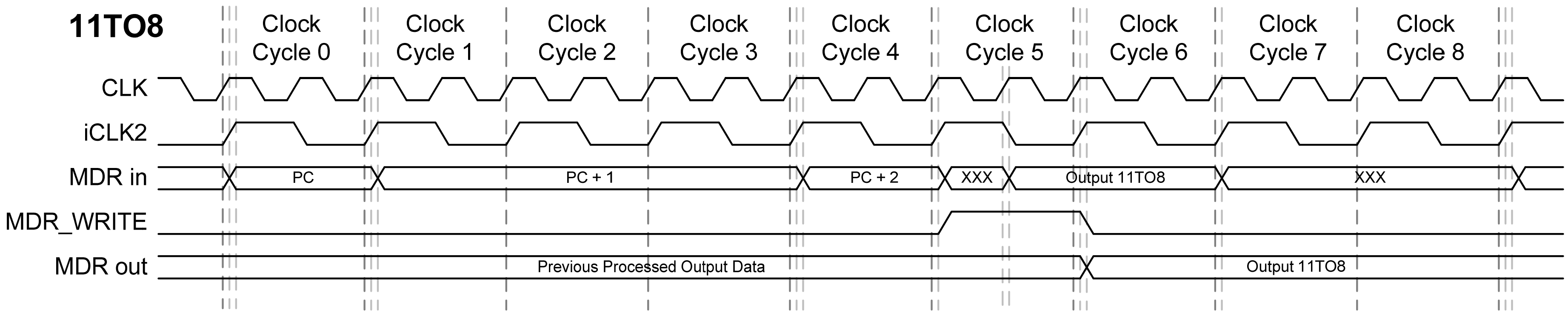


Figure 79 MDR register timing diagram for 11TO8 instruction.

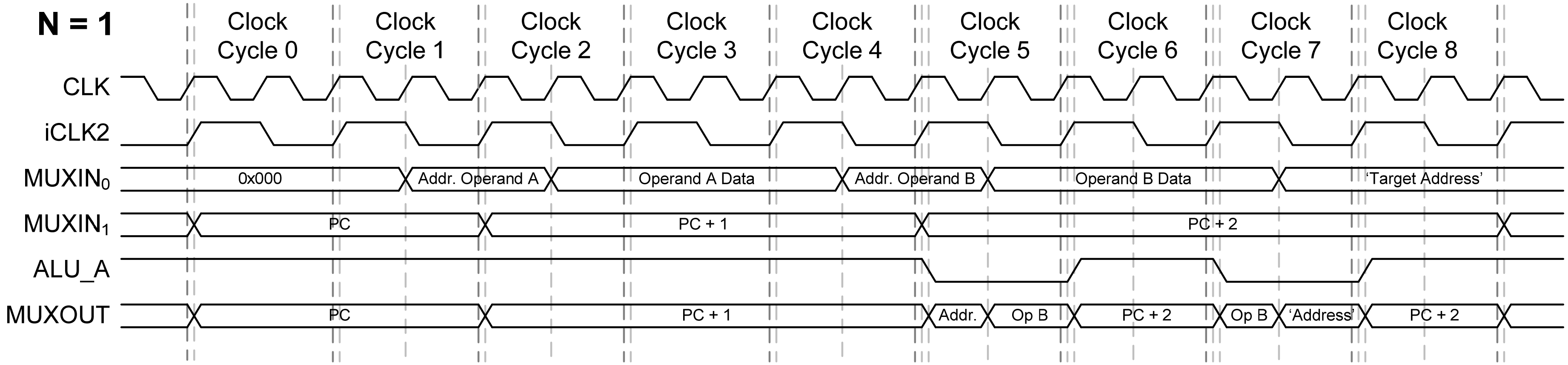


Figure 80 ALU_A MUX timing diagram for SBN instruction (N = 1).

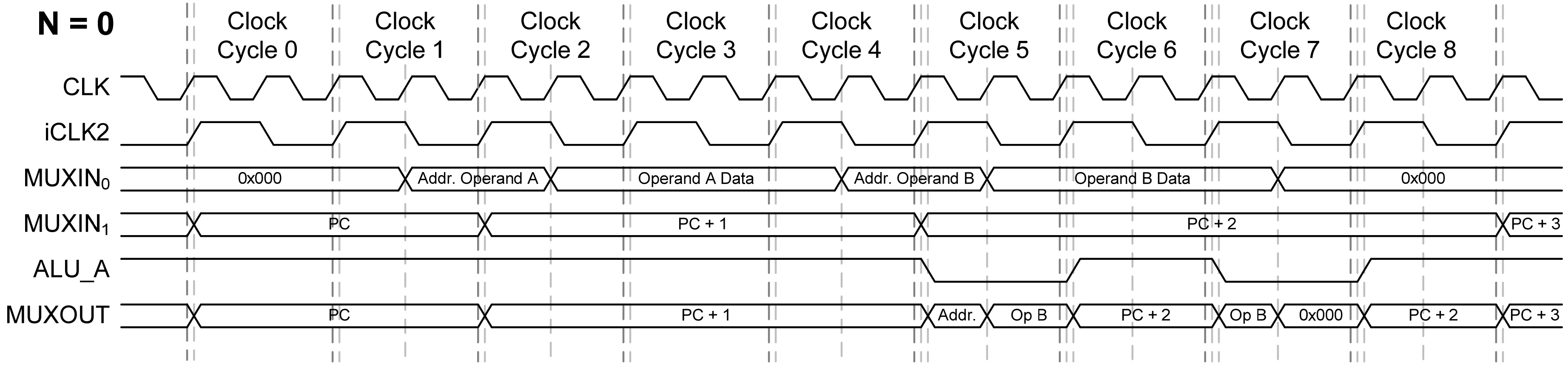


Figure 81 ALU_A MUX timing diagram for SBN instruction (N = 0)

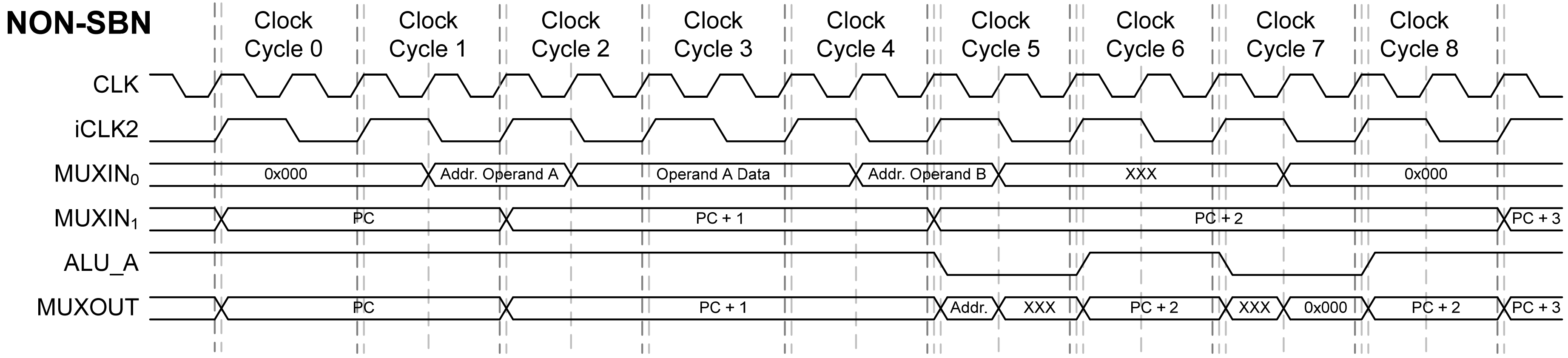
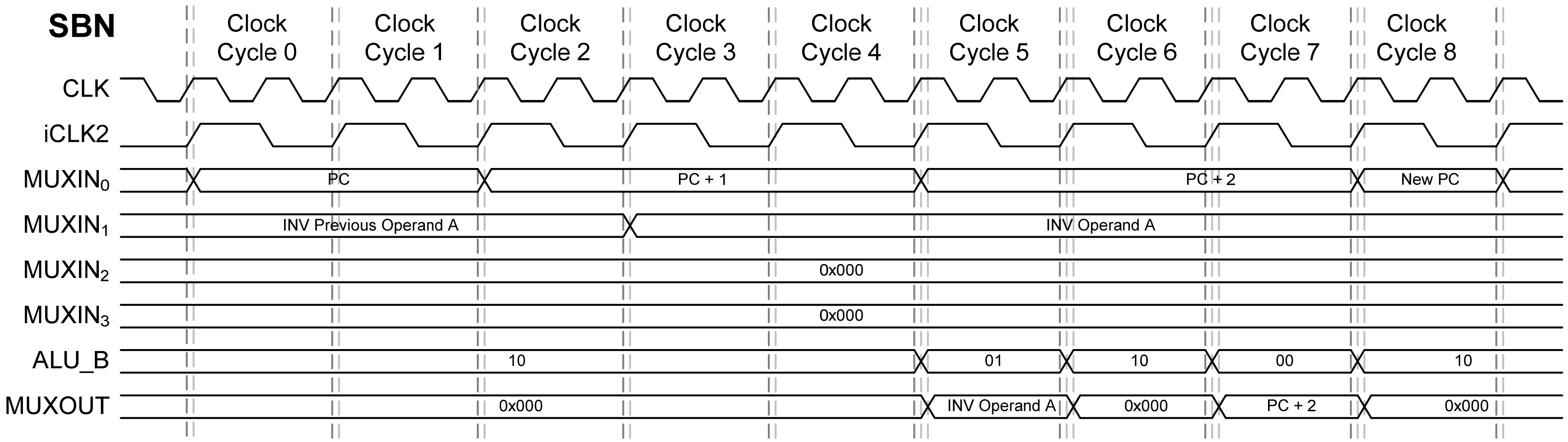


Figure 82 ALU_A MUX timing diagram for Non-SBN instruction.



New PC = PC + 2 + 'Target Address'

Figure 83 ALU_B MUX timing diagram for SBN instruction.

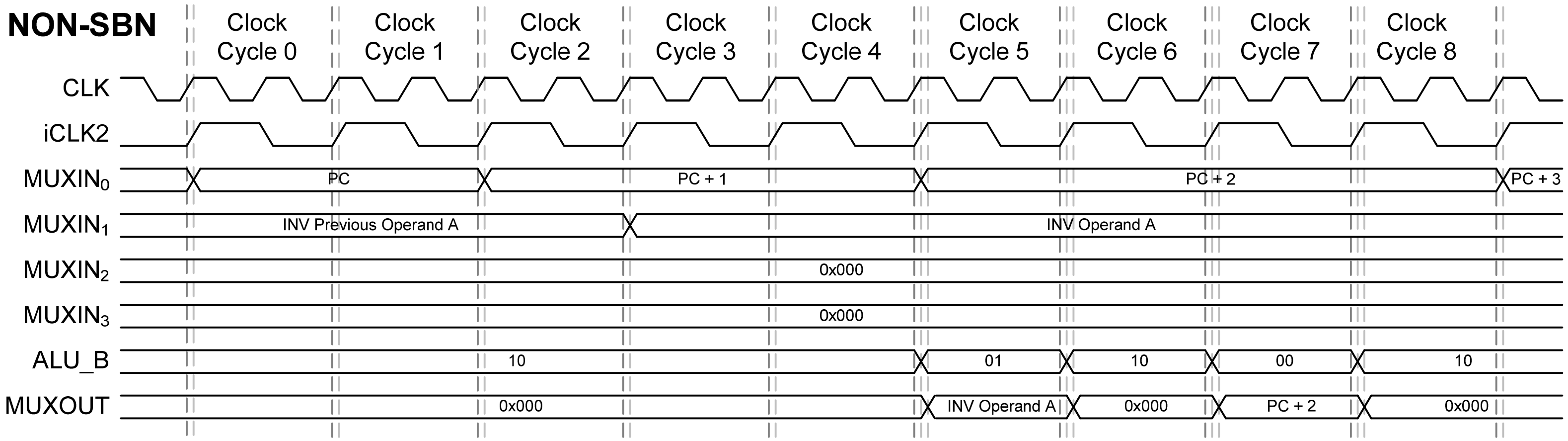


Figure 84 ALU_B MUX timing diagram for Non-SBN instruction.

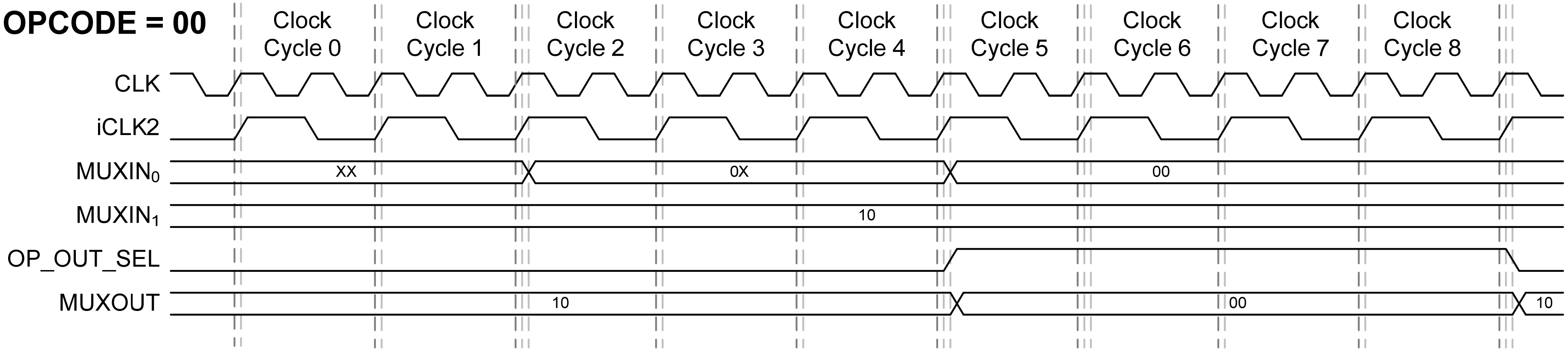


Figure 85 OP_OUT MUX timing diagram for GF MULT instruction (OPCODE=00).

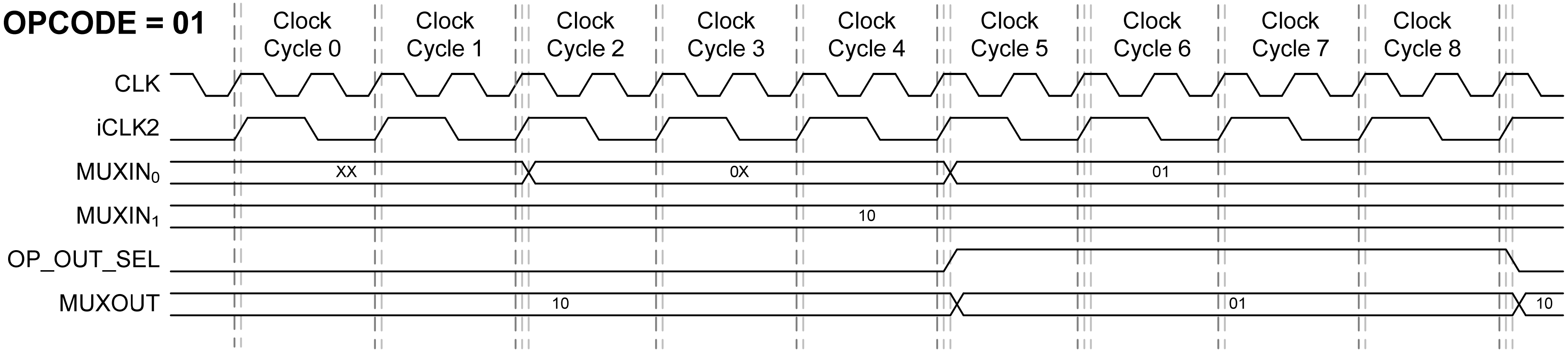


Figure 86 OP_OUT MUX timing diagram for XOR instruction (OPCODE=01).

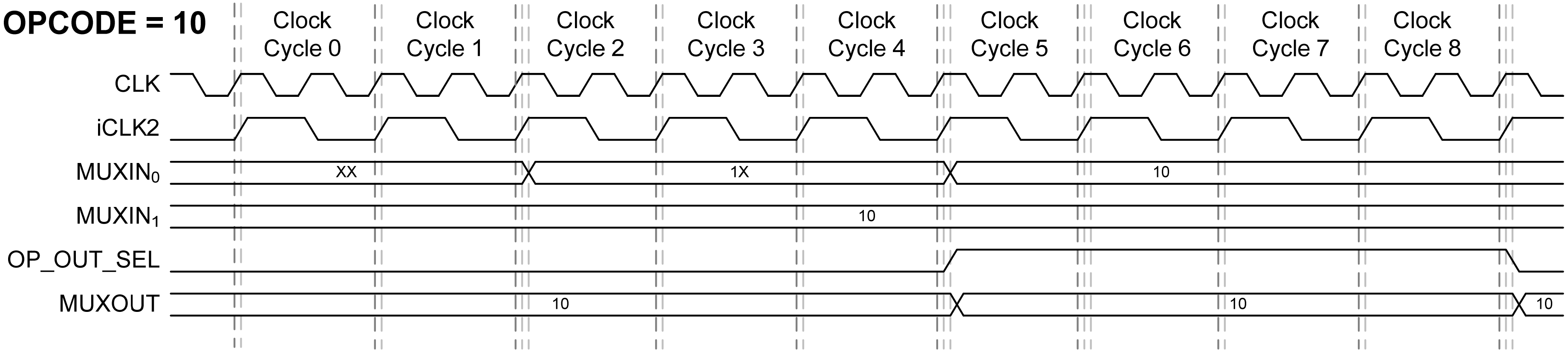


Figure 87 OP_OUT MUX timing diagram for SBN instruction (OPCODE=10).

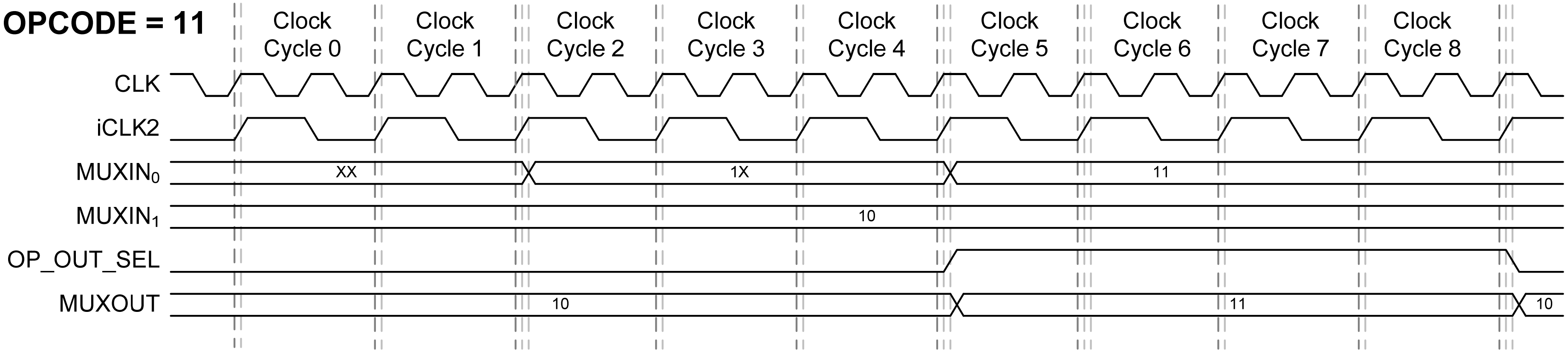


Figure 88 OP_OUT MUX timing diagram for 11TO8 instruction (OPCODE=11).

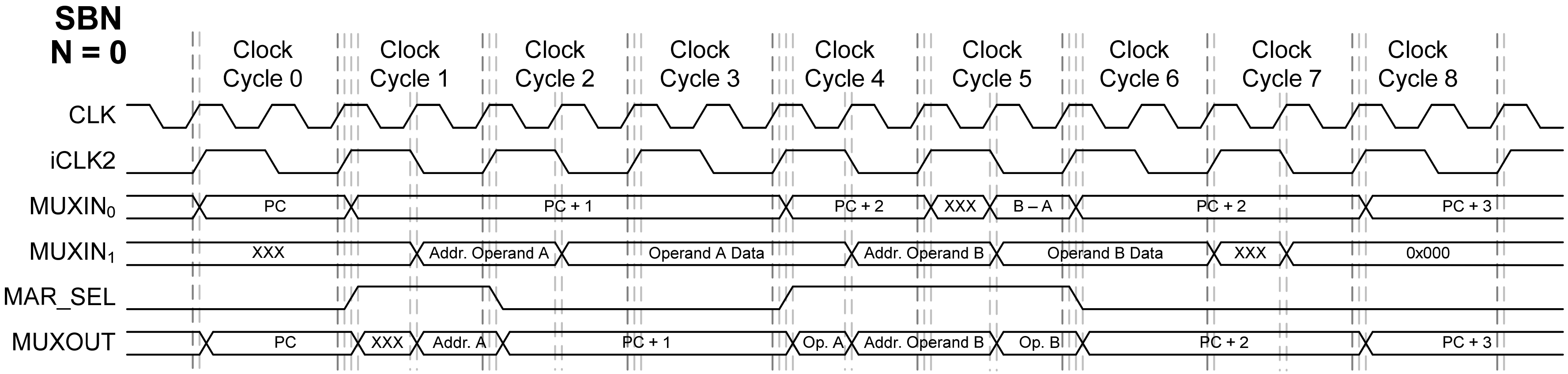


Figure 89   MAR_IN MUX timing diagram for SBN instruction (N = 0).

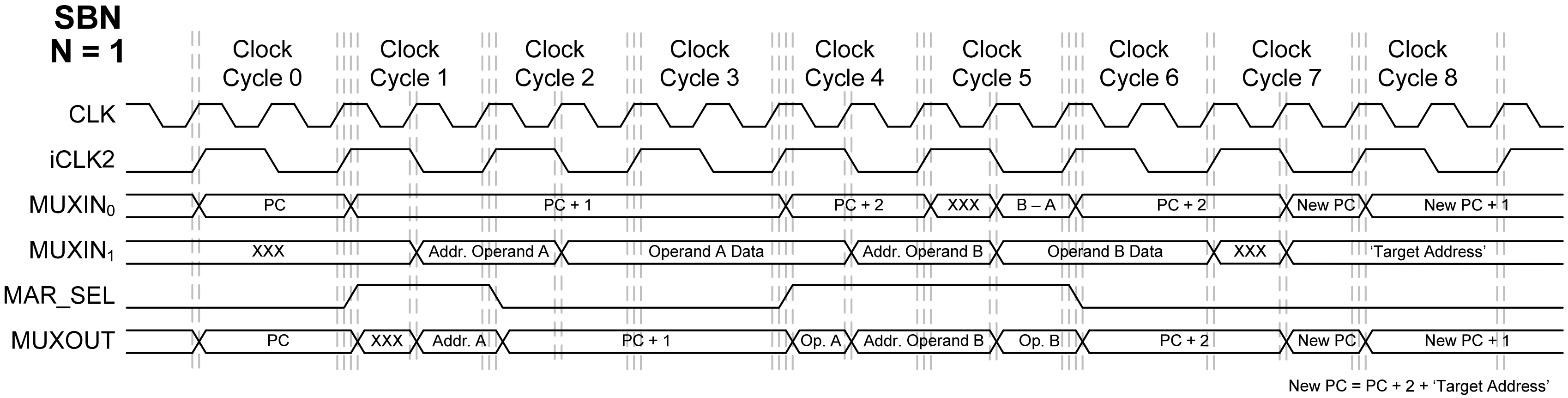


Figure 90   MAR_IN MUX timing diagram for SBN instruction (N = 1).

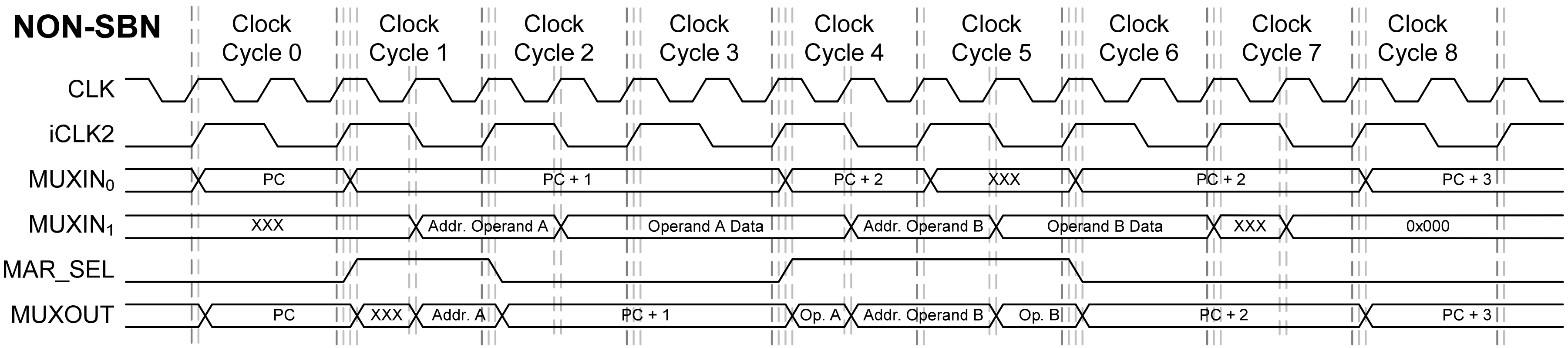


Figure 91   MAR_IN MUX timing diagram for Non-SBN instruction.

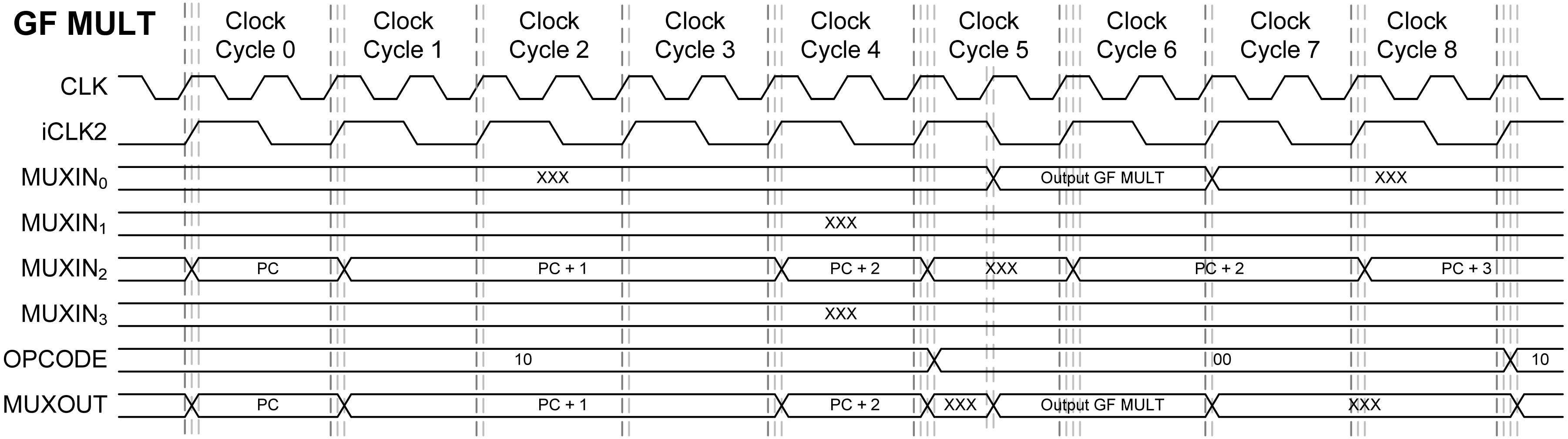


Figure 92   MDR_IN MUX timing diagram for GF MULT instruction.

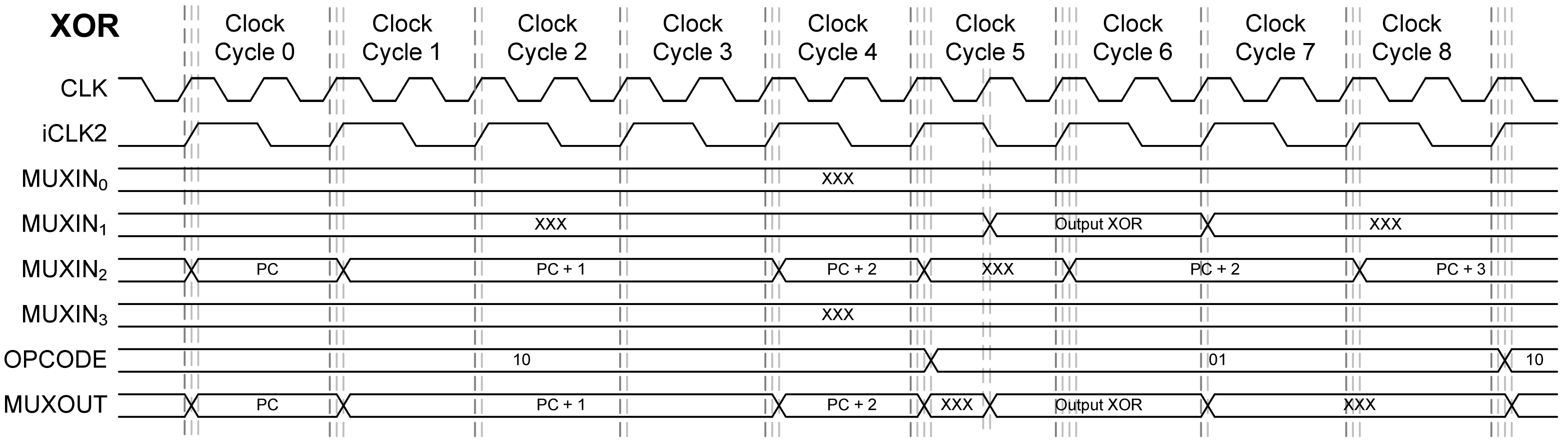


Figure 93 MDR_IN MUX timing diagram for XOR instruction.

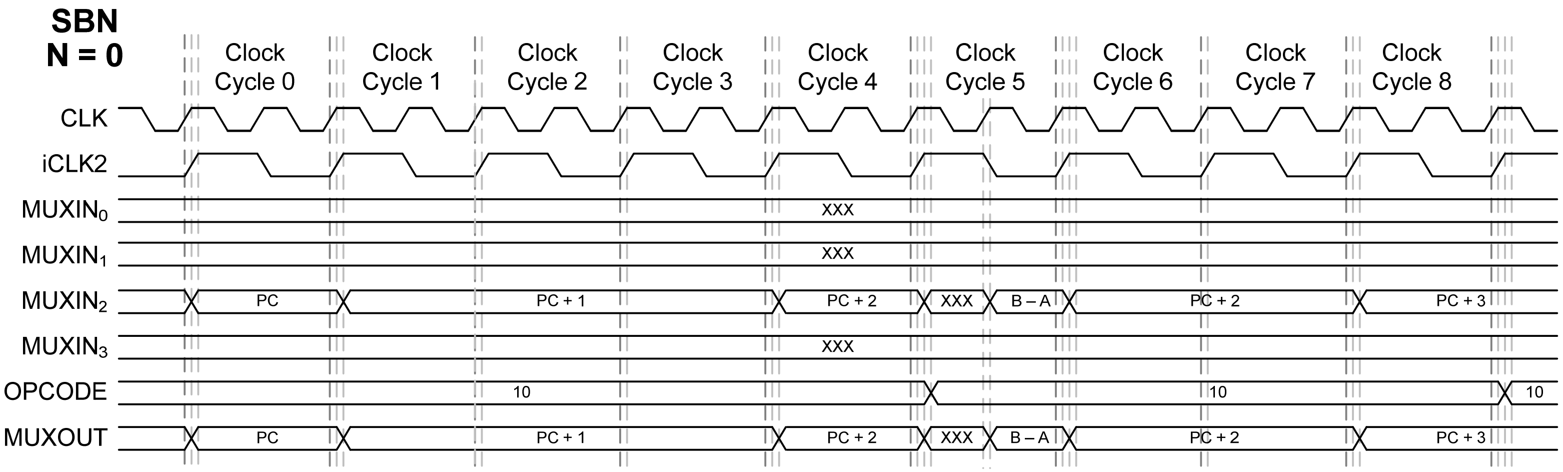


Figure 94 MDR_IN MUX timing diagram for SBN instruction (N = 0).

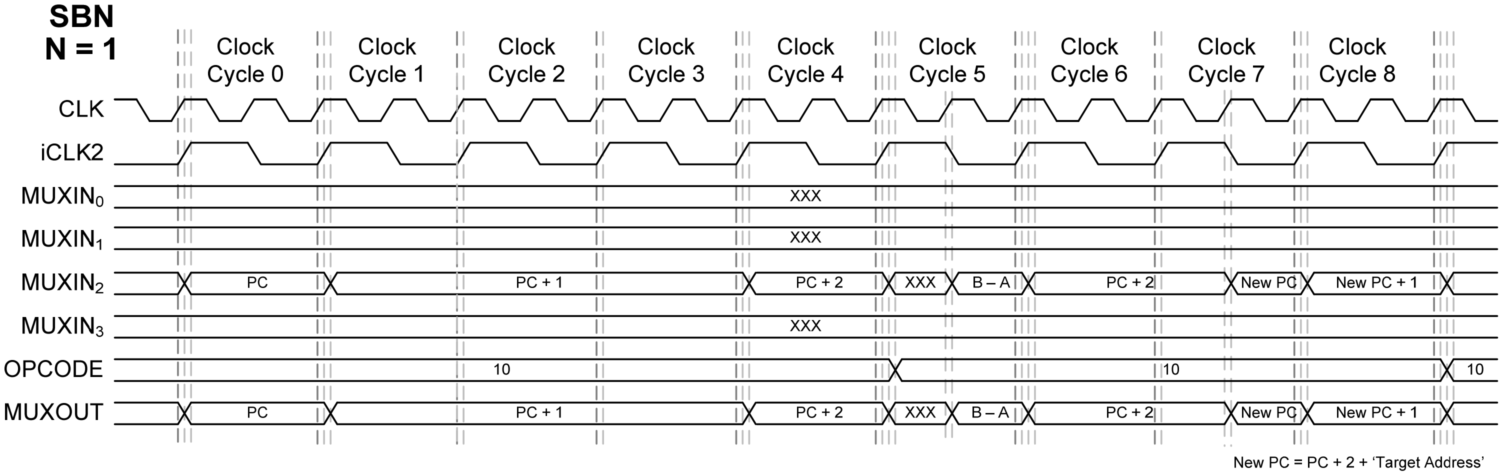


Figure 95 MDR_IN MUX timing diagram for SBN instruction (N = 1).

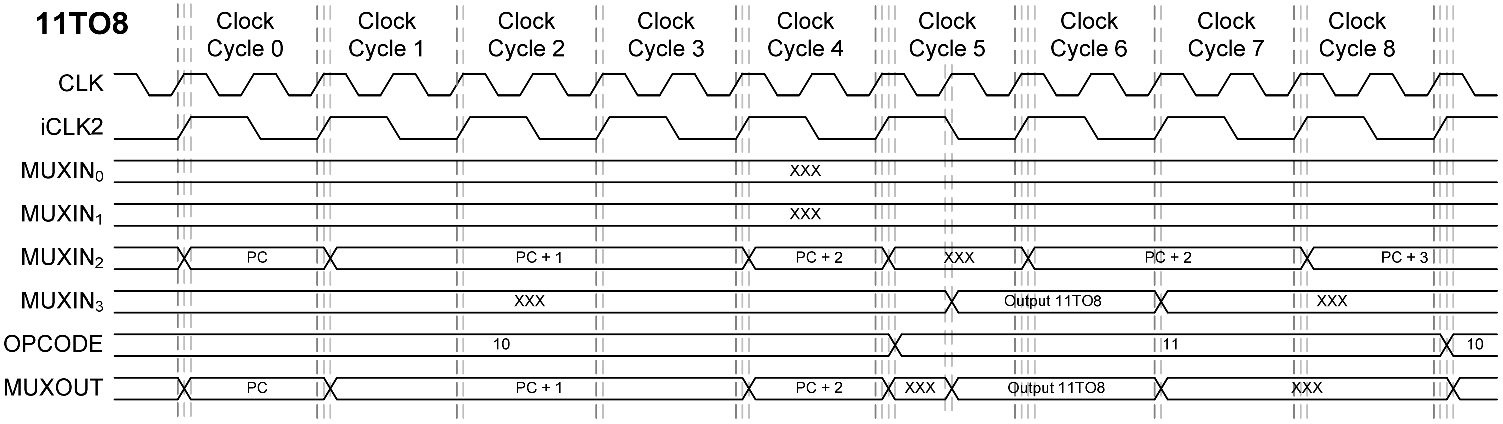


Figure 96 MDR_IN MUX timing diagram for 11TO8 instruction.

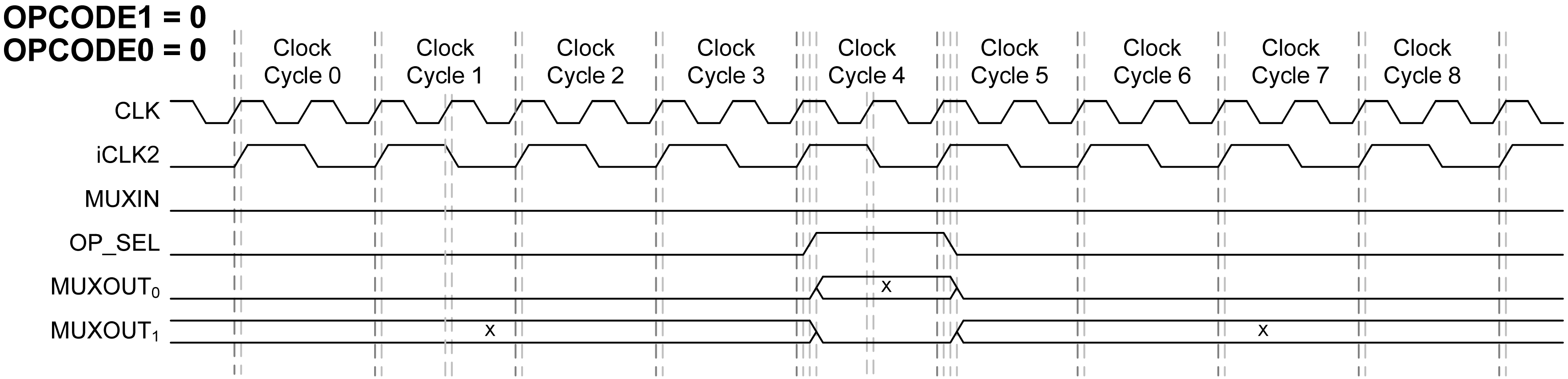


Figure 97 OP_SEL DEMUX timing diagram for GF MULT instruction (OPCODE=00).

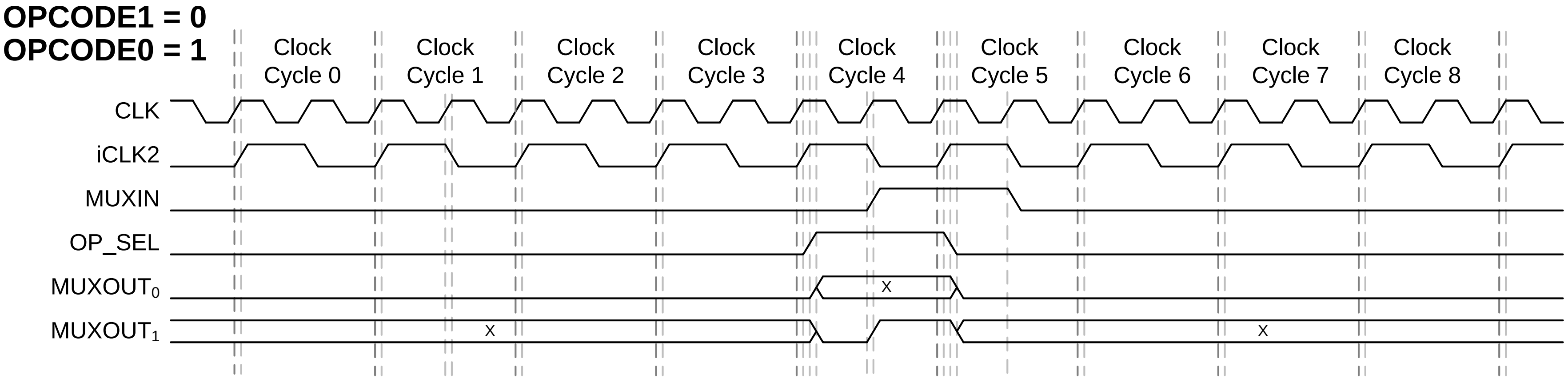


Figure 98 OP_SEL DEMUX timing diagram for XOR instruction (OPCODE=01).

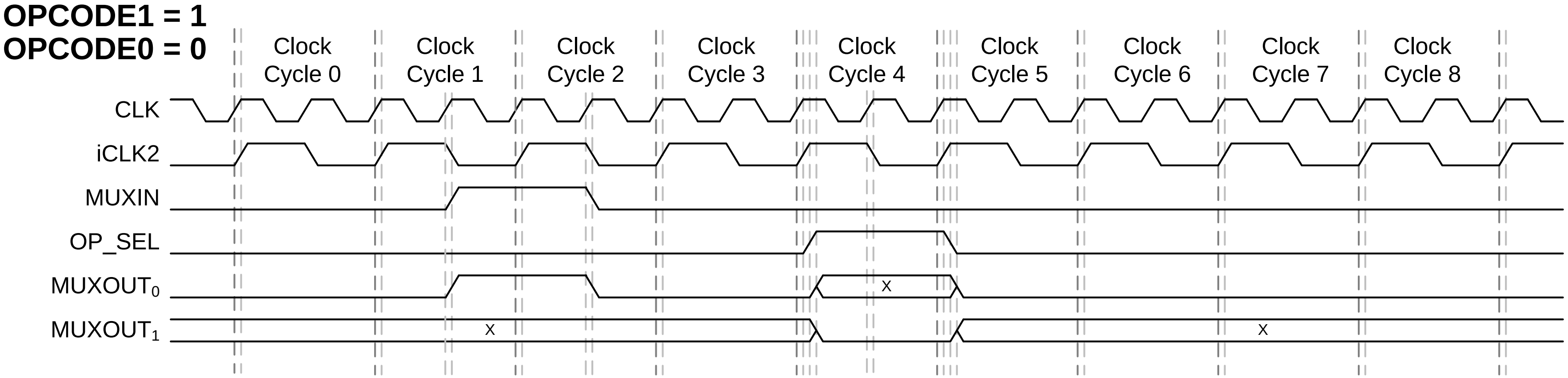


Figure 99 OP_SEL DEMUX timing diagram for SBN instruction (OPCODE=10).

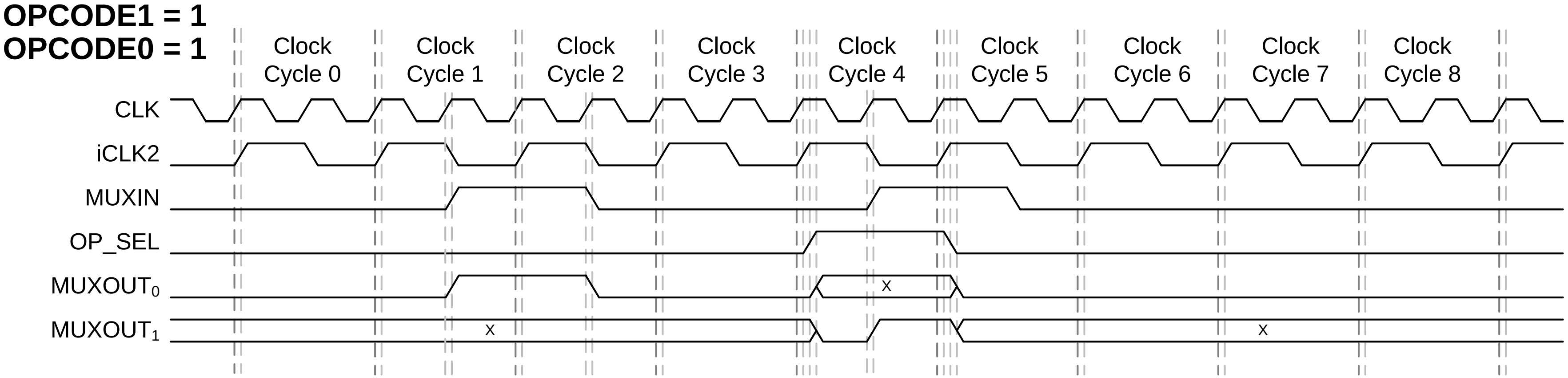


Figure 100 OP_SEL DEMUX timing diagram for 11TO8 instruction (OPCODE=11).

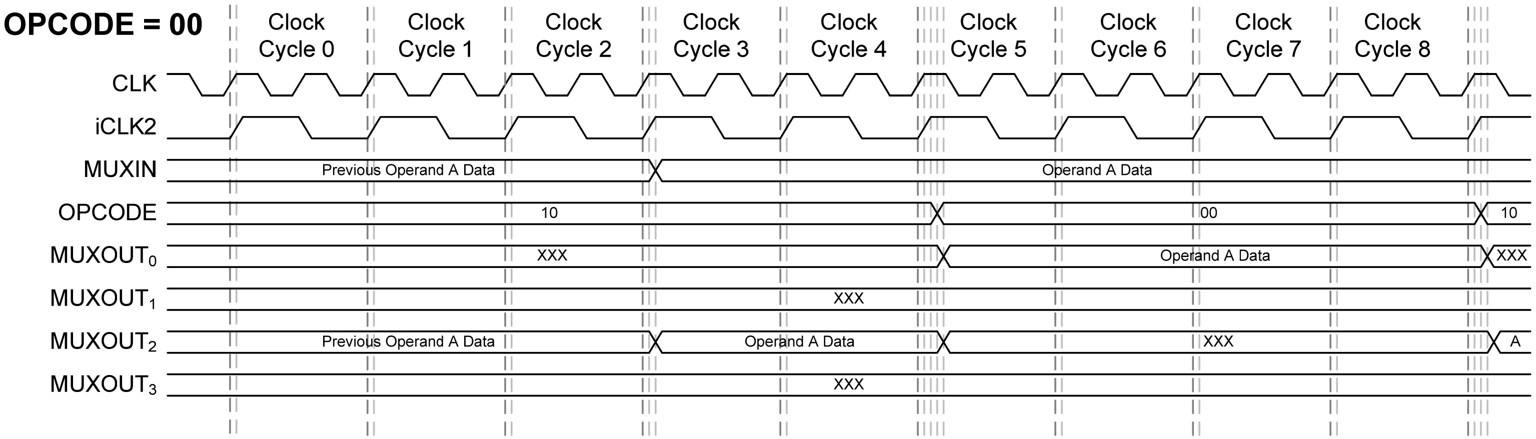


Figure 101 R_OUT DEMUX timing diagram for GF MULT instruction (OPCODE=00).

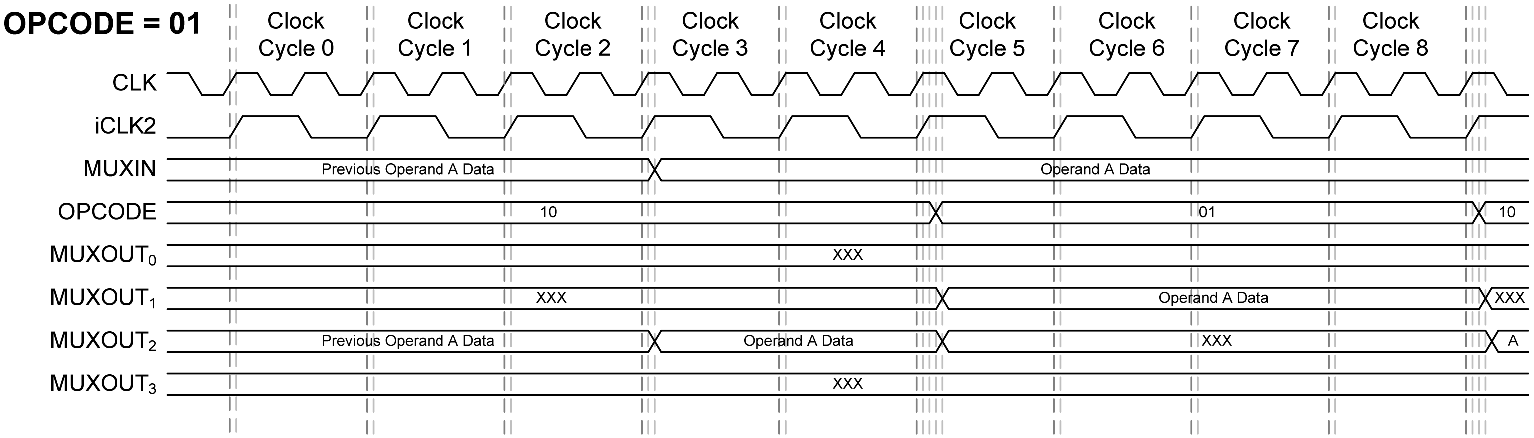


Figure 102 R_OUT DEMUX timing diagram for XOR instruction (OPCODE=01).

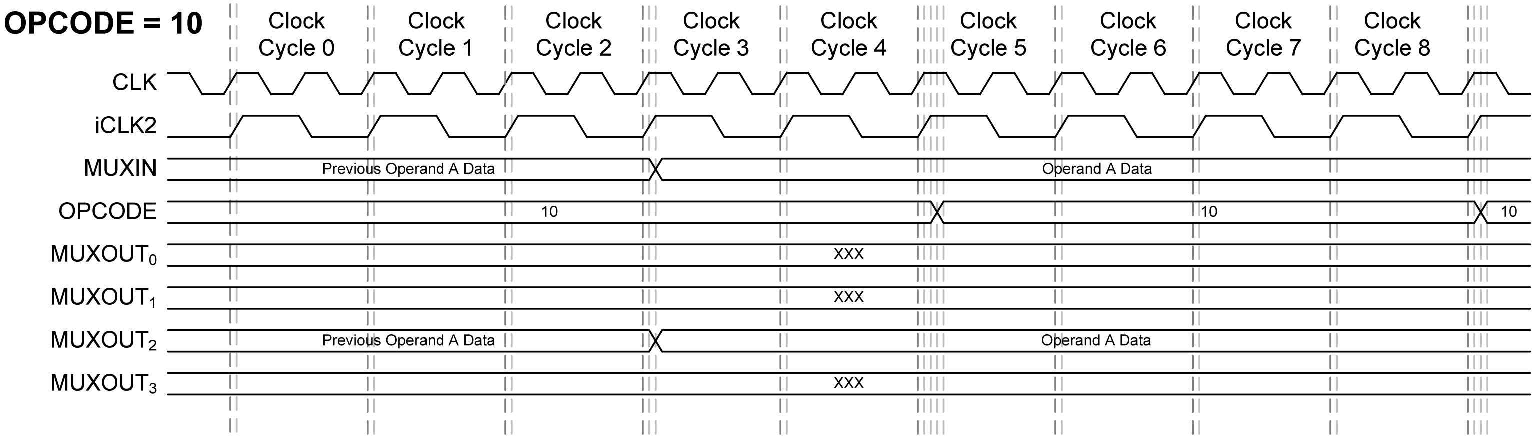


Figure 103 R_OUT DEMUX timing diagram for SBN instruction (OPCODE=10).

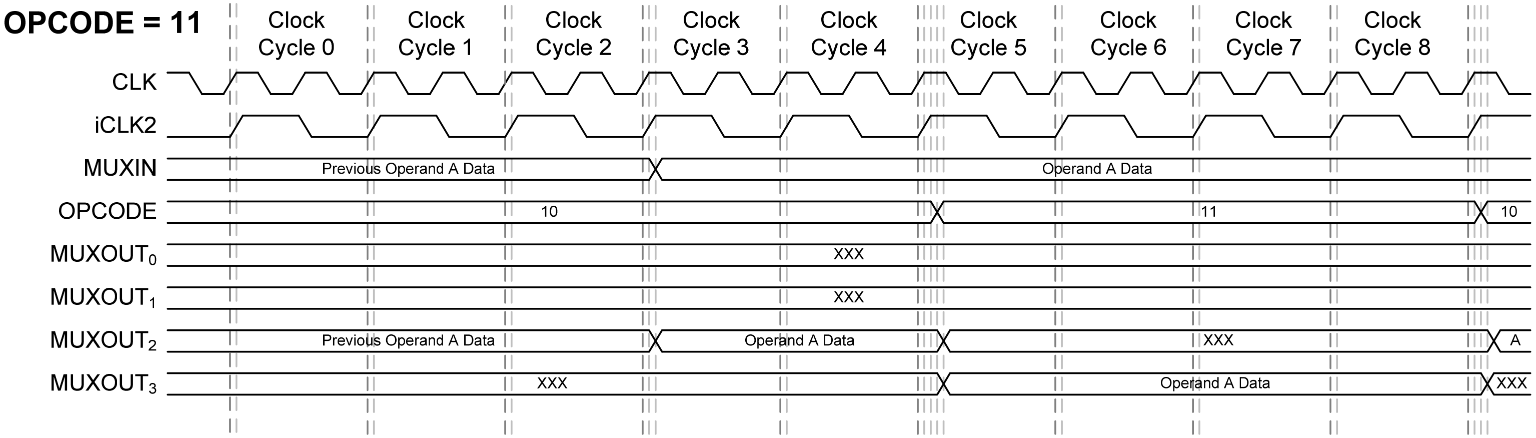


Figure 104 R_OUT DEMUX timing diagram for 11TO8 instruction (OPCODE=11).

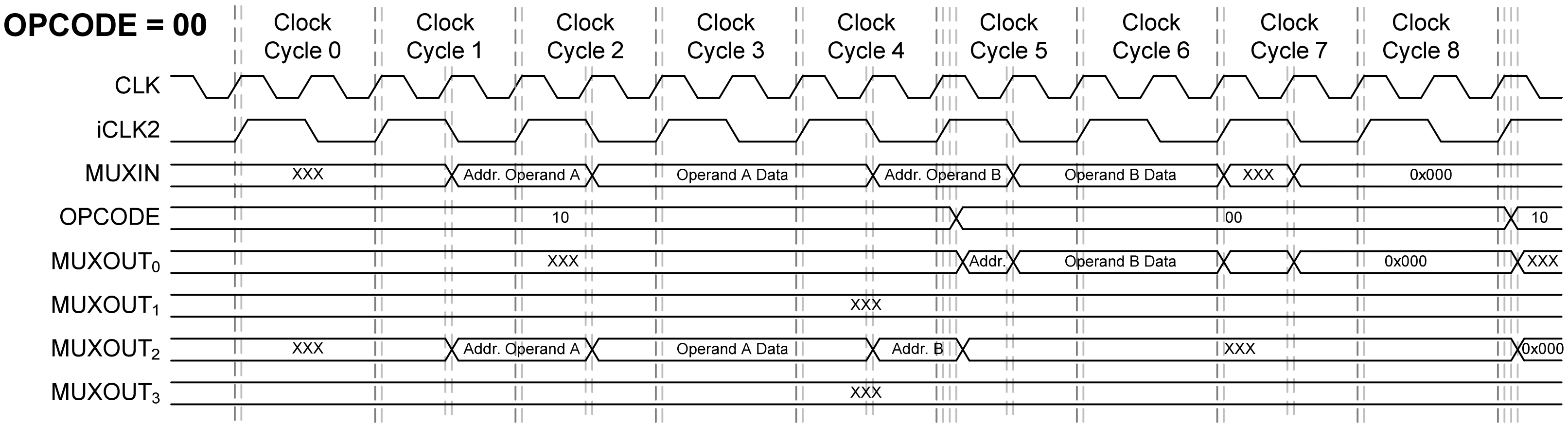


Figure 105 MEM_OUT DEMUX timing diagram for GF MULT instruction (OPCODE=00).

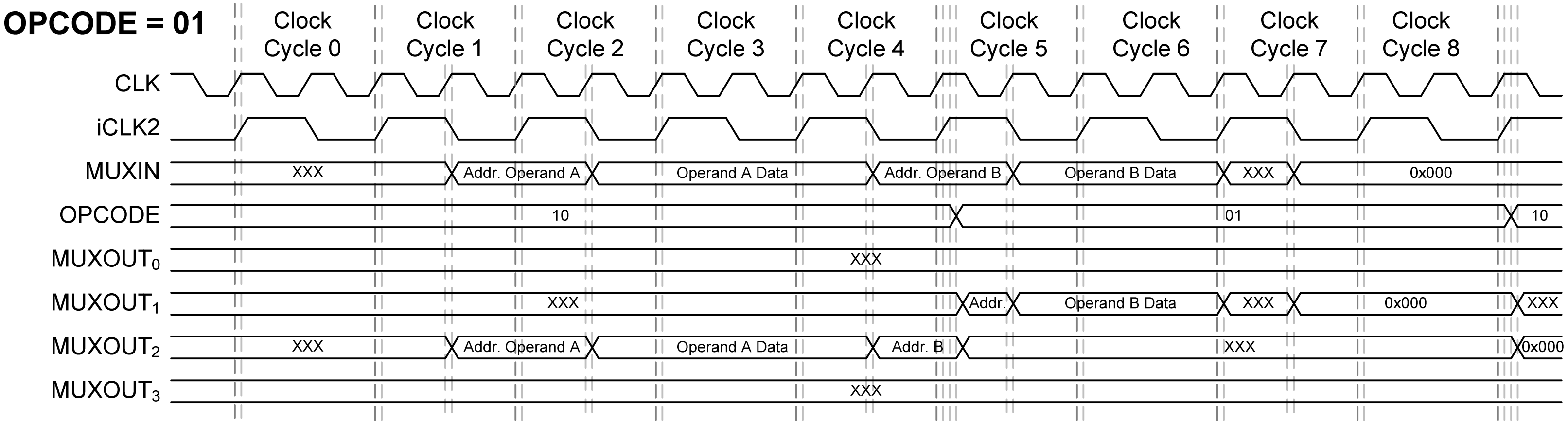


Figure 106 MEM_OUT DEMUX timing diagram for XOR instruction (OPCODE=01).

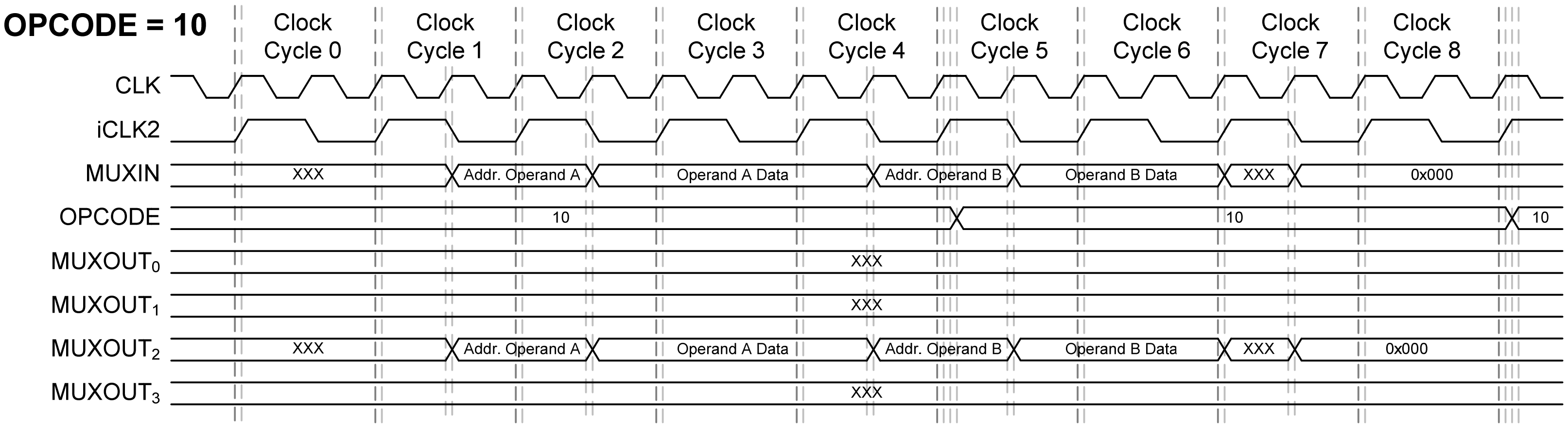


Figure 107 MEM_OUT DEMUX timing diagram for SBN instruction (OPCODE=10).

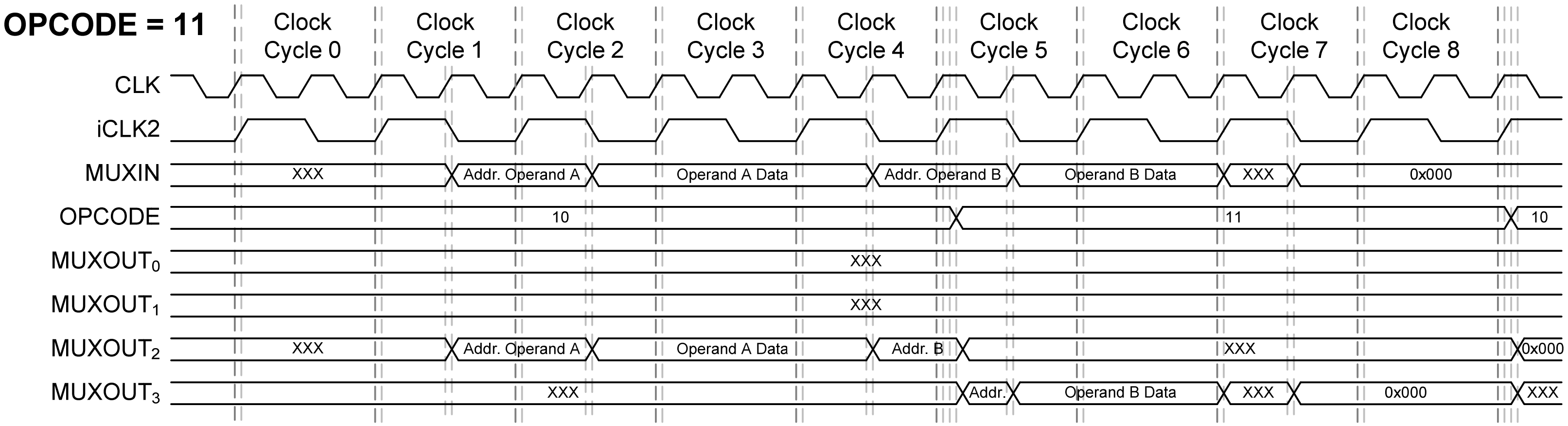


Figure 108 MEM_OUT DEMUX timing diagram for 11TO8 instruction (OPCODE=11).

3.4 DWT CRS MISC MEMORY

The DWT CRS MISC architecture contains only single memory storage that follows the von Neumann architecture. The reason to use the single memory space for DWT CRS MISC is to design a MISC architecture with low hardware complexity. Similarly, most of the general-purpose computers are modelled with this concept because of the simplicity of von Neumann architecture [139].

As a result, the DWT CRS MISC architecture were designed with both data memory and program memory stored in the single memory storage space. The total amount of available memory is 3,072 Bytes (2,048 x 12-bit), which is shown in Figure 109. The memory is separated into two parts, where 33.59% for storing of data and another 66.41% is used to store the program instructions. In actual implementations, the data only takes up 100% of the total available data memory and the program instructions utilized 46.32% of the total available programme memory.

The input data (image data) are located at memory location from 2 to 257 (64 pixels x 4 pixels). These image data will be overwritten to become different level subband of DWT coefficients after the MISC performs the DWT filtering (Algorithm 1.0). Next, the data in the memory location from 258 (0x102) to 596 (0x254) are made up of fixed data. Examples of the fixed data are the coefficients of the SEC Generator Matrix G_{SEC} values and the counter value used in repeating the program instructions (Algorithm 2.0). As for data situated at memory location from 597 (0x255) to 687 (0x2AF), these data are the temporary data that will be used during the program execution. At the end of the program, the CRS encoded symbols will also be written back into these memory locations. The encoded CRS symbols are stored at memory location from 608 (0x260) to 687 (0x2AF). For the program instructions, they are stored at the memory location from 688 (0x2B0) to 1,517 (0x5ED).

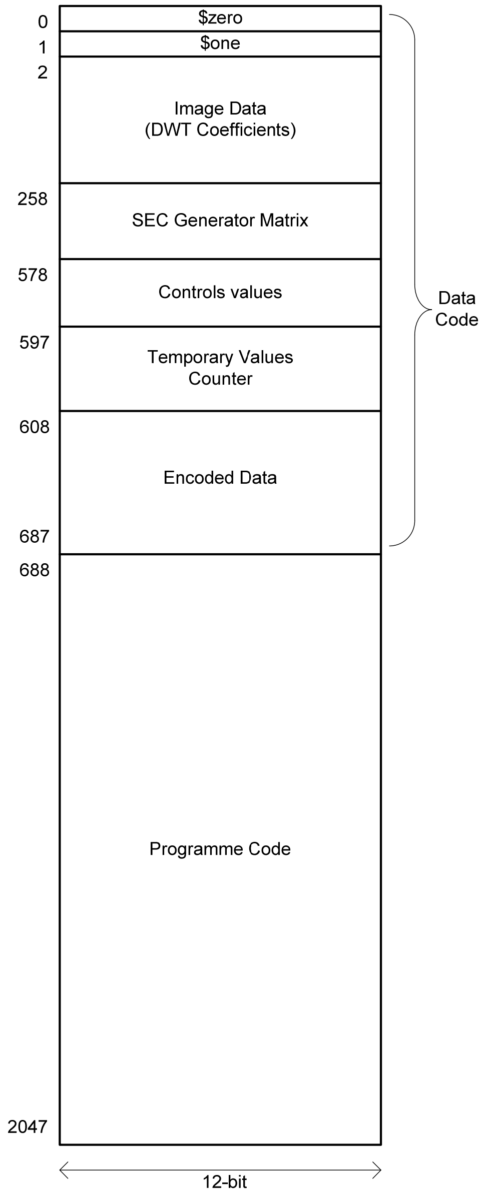
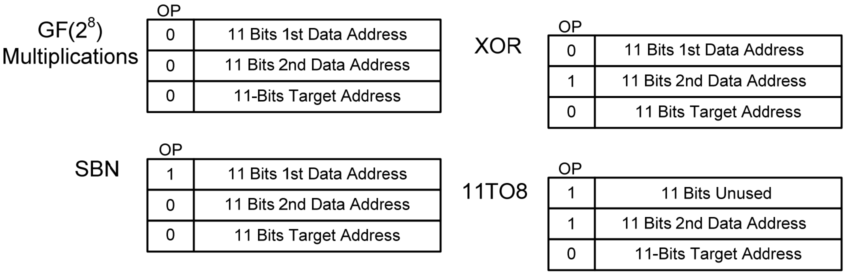


Figure 109 DWT CRS MISC memory location.

3.5 PROGRAMME INSTRUCTIONS FORMAT

With four functional blocks mentioned in Section 3.2, four different programme instructions were developed such that these instructions control the operation of the MISC architecture. As shown in Figure 110, the corresponding four programme instructions are GF, SBN, XOR and 11TO8 instructions. The standard format for all the programme instructions is written in 3 lines of Hexadecimal (HEX) codes in the programme memory. The first line of HEX code states the memory address location for the 1st input Operand (data). Next, the second line of the HEX code consists of the 2nd input Operand (data) memory address location.



GF(2^8) Multiplications

OP	
0	11 Bits 1st Data Address
0	11 Bits 2nd Data Address
0	11-Bits Target Address

XOR

OP	
0	11 Bits 1st Data Address
1	11 Bits 2nd Data Address
0	11 Bits Target Address

SBN

OP	
1	11 Bits 1st Data Address
0	11 Bits 2nd Data Address
0	11 Bits Target Address

11TO8

OP	
1	11 Bits Unused
1	11 Bits 2nd Data Address
0	11-Bits Target Address

Figure 110 Program Instructions for DWT CRS MISC architecture.

Lastly, the third line of HEX code represents the ‘Target Address’ that is used by SBN instruction to jump to the designated programme memory address location. If negative result is obtained from the SBN instructions, then the ‘Target Address’ is added to the PC value read from PC register. This allows the MISC to jump and execute the targeted next programme memory address location. As for the other three instructions, the MISC only executes the subsequent programme instruction. For failsafe, the ‘Target Address’ is set to zero in these three programme instructions to prevent the MISC to make any changes on the current PC value.

Memory Address	Instructions	Operand A	Operand B	‘Target Address’
0x2F8	**SBN**	**$one**	**$t7**	**0x7F7**

↓ Machine Codes

Memory Address	Operand A Address	Operand B Address	‘Target Address’
0x2F8	**0x801**	**0x259**	**0x7F7**

Figure 111 Written Programme Instruction and its corresponding machine codes.

In Figure 111, an example of DWT CRS MISC programme instructions that performs the SBN instruction is written at memory address location 0x2F8. At Clock Cycle 6, the PC value is 0x2FA and it is output to the MAR register. After the rising-edge Clock Cycle 7, the MAR register is set to the 0x2FA PC value. During falling-edge Clock Cycle 7, the ‘Target Address’ (0x7F7) will be read from the Memory. While the ‘Target Address’ is read, current PC value 0x2FA is added with 0x7F7 to get the new PC value 0x2F1. If SBN instruction produces negative results (N = 1), this new PC value (0x2F1) is stored into PC register after the rising-edge Clock Cycle 8. However, if SBN instruction produces positive results (N = 0), this new PC value will not be stored into the PC register. After that, the new PC value is increased by 1 to become 0x2F2. This increased new PC value (0x2F2) is actually the next programme instruction memory address location. Therefore, it allows the DWT CRS MISC to jump to a different location of programme instruction. Figure 112 gives a better illustrations how the PC value is manipulated when negative results are obtained for SBN instructions (N = 1).

To jump to ‘Target’ Instruction Address	**0x2F2**
Current PC Value after rising-edge Clock Cycle 7	**0x2FA**
‘Target Address’ Read from Memory after falling-edge Clock Cycle 7	**0x7F7**
New PC Value after falling-edge Clock Cycle 7	**0x2FA + 0x7F7 = 0x2F1**
Increased New PC Value to be stored in PC register after rising-edge Clock Cycle 8 (N = 1)	**0x2F2**
PC Value at Next rising-edge Clock Cycle 0	**0x2F2** (‘Target’ Instruction’s Address)

Figure 112 Setting the 'Target Address' for SBN (N = 1).

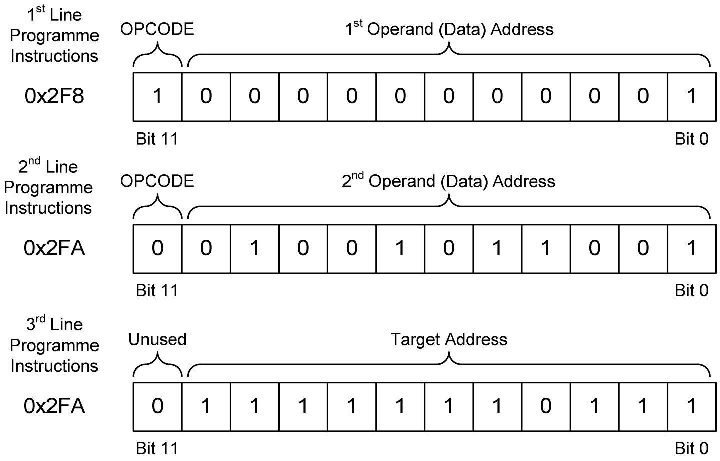


Figure 113 Machine Code of SBN instruction in programme memory.

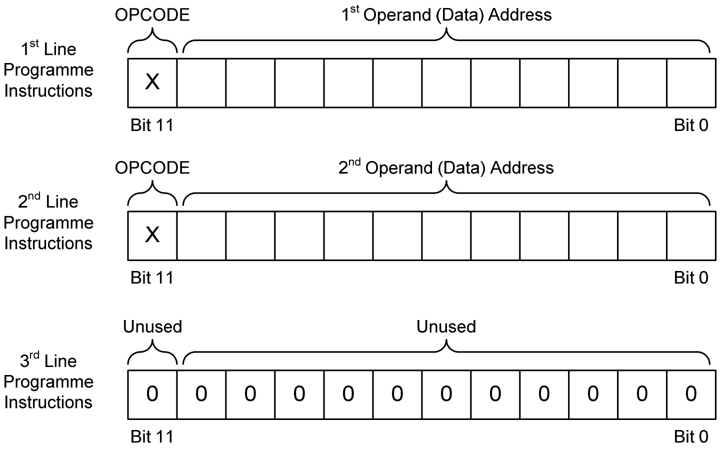

OPCODE
1st Operand (Data) Address
1st Line Programme Instructions
X
Bit 11
Bit 0
OPCODE
2nd Operand (Data) Address
2nd Line Programme Instructions
X
Bit 11
Bit 0
Unused
Unused
3rd Line Programme Instructions
0 0 0 0 0 0 0 0 0 0 0 0
Bit 11
Bit 0

Figure 114 Machine Code of Non-SBN instruction in programme memory

For each programme instructions, the 2-bit OPCODE is read from the MSB of the 1st and 2nd line of HEX code. With the OPCODE read, it determines which functional blocks that will process the two input data. Figure 113 and Figure 114 illustrate how the OPCODE are arranged in the each programme instructions (1st and 2nd line of HEX code). It can be seen that the OPCODE are located at the MSB of the 1st and 2nd line of the 3 lines HEX code programme instruction.

However, an exception for the 11TO8 programme instruction, whereby this particular functional block only converts the 2nd input Operand (data) from 11-bit to become 8-bit output data. Section 3.2.5 presents a better illustration on how the 11TO8 block converts the input data from 11-bit to 8-bit. For better illustrations on these programme instructions, refer to Section 4.2 for the simulation waveforms on these four programme instructions.

3.6 DWT CRS ALGORITHM

For the DWT CRS MISC to perform DWT compression and CRS coding, programme instructions need to be written and programmed it into memory of the MISC architecture. First, the Lifting Scheme DWT filter was briefly explained in Section 3.6.1. Next, Section 3.6.2 describes how the MISC operate as an DWT filter that processes and decomposes the image data into DWT coefficients. With the use of DWT, image compression could be performed onto the image data captured by the visual sensor node. Afterwards, Section 3.6.3 briefly describes on the CRS coding scheme used in the DWT CRS MISC processor. Lastly, Section 3.6.4 describes the selective CRS encoding (encrypting) process on the DWT coefficients.

3.6.1 Lifting Scheme Discrete Wavelet Transform

In the year of 1998, Wim Sweldens introduced the Lifting Scheme DWT as another alternative of DWT computation [19] [33]. The Lifting Scheme DWT was developed into integer wavelet transforms that can be used in lossless image compression [140]. The advantage of the Lifting Scheme DWT is that less operation is required to perform the DWT computation as compared with the traditional filter bank scheme [141] [142]. Another advantages of the Lifting Scheme DWT is that it computes an integer wavelet transform thus enabling both lossless and lossy image encoders to be designed in an embedded system [27]. Therefore, the Lifting Scheme DWT is selected to be used to reduce the amount of image captured at the sensor nodes.

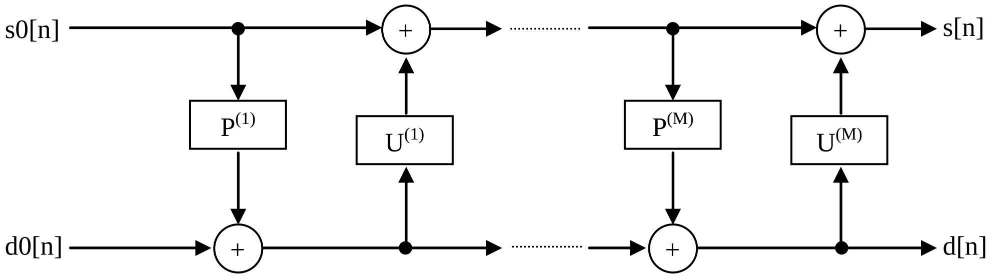


Figure 115 Lifting Scheme Discrete Wavelet Transform Filter Bank [143].

For Lifting Scheme DWT, the sampled image pixels are split into odd, s[n] and even, d[n] samples. These samples will then be input into filter bank using lifting step [144]. The odd, s[n] and even, d[n] samples properties are improved using the alternative lifting and dual lifting steps [145], as shown in Figure 115. The Predict step, P is a dual lifting step that consists of applying a Low Pass filter on the even samples and subtracting the result from the odd samples. The Update step, U is the lifting step that does the opposite of Predict step, where the odd samples are being passed through a High Pass filter and subtracting it from the even samples. Lastly, with going through several of the dual and primal lifting steps, the Low Pass coefficients are the even samples output from the filter bank. For the High Pass coefficients, it would be the odd samples output from the filter bank [145].

3.6.2 DWT Image Compression Algorithm

The DWT CRS MISC processor perform 2 Levels of 2-dimensional Lifting Scheme DWT filtering onto the image data (size of 64 pixels x 4 pixels) that are input to it. For 1st Level 2-dimensional DWT, the Row (Horizontal) Filtering onto the original image data was performed first and followed by Column (Vertical) Filtering onto the L and H subband coefficients. For better visualisation, the DWT coefficients were arranged in the manner illustrated in Figure 116. However, the actual arrangement is a continuous data memory as shown in the Figure 117.

$L_{0,0}$	$H_{0,0}$	$L_{0,1}$	$H_{0,1}$	$L_{0,2}$	$H_{0,2}$	$L_{0,3}$	$H_{0,3}$
$L_{1,0}$	$H_{1,0}$	$L_{1,1}$	$H_{1,1}$	$L_{1,2}$	$H_{1,2}$	$L_{1,3}$	$H_{1,3}$
$L_{2,0}$	$H_{2,0}$	$L_{2,1}$	$H_{2,1}$	$L_{2,2}$	$H_{2,2}$	$L_{2,3}$	$H_{2,3}$
$L_{3,0}$	$H_{3,0}$	$L_{3,1}$	$H_{3,1}$	$L_{3,2}$	$H_{3,2}$	$L_{3,3}$	$H_{3,3}$

(a) Row (horizontal) filtering.

$LL_{0,0}$	$LH_{0,0}$	$LL_{0,1}$	$LH_{0,1}$	$LL_{0,2}$	$LH_{0,2}$	$LL_{0,3}$	$LH_{0,3}$
$HL_{0,0}$	$HH_{0,0}$	$HL_{0,1}$	$HH_{0,1}$	$HL_{0,2}$	$HH_{0,2}$	$HL_{0,3}$	$HH_{0,3}$
$LL_{1,0}$	$LH_{1,0}$	$LL_{1,1}$	$LH_{1,1}$	$LL_{1,2}$	$LH_{1,2}$	$LL_{1,3}$	$LH_{1,3}$
$HL_{1,0}$	$HH_{1,0}$	$HL_{1,1}$	$HH_{1,1}$	$HL_{1,2}$	$HH_{1,2}$	$HL_{1,3}$	$HH_{1,3}$

(b) Column (vertical) filtering.

Figure 116 Level 1 DWT coefficients arrangement in 2D.

$L_{0,0}$	$H_{0,0}$	$L_{0,1}$	$H_{0,1}$	......	$L_{1,0}$	$H_{1,0}$	......	$L_{2,0}$	$H_{2,0}$	......	$L_{3,2}$	$H_{3,2}$	$L_{3,3}$	$H_{3,3}$

(a) Row (horizontal) filtering.

$LL_{0,0}$	$LH_{0,0}$	$LL_{0,1}$	$LH_{0,1}$	......	$HL_{0,0}$	$HH_{0,0}$	......	$LL_{1,0}$	$LH_{1,0}$	......	$HL_{1,2}$	$HH_{1,2}$	$HL_{1,3}$	$HH_{1,3}$

(b) Column (vertical) filtering.

Figure 117 Actual Level 1 DWT coefficients arrangement in memory.

The DWT Filter used in the DWT CRS MSC processor is the Lifting Scheme DWT Haar filter. The process of filtering is performed by subtracting the Even image pixel with the Odd image pixel (H = \$R1 - \$R0) to give High Pass DWT coefficient (L subband). Then the division of two is performed onto the High Pass DWT coefficients and the result is summed with the Odd image pixel (L = \$R0 + ½ x H) to obtain the low pass coefficients (H subband). Algorithm 1.0 describes how the DWT CRS MISC actually performed the Level 1 DWT Filtering onto the image data that were stored in the memory. Inside the Algorithm 1.0, it also includes both Algorithm 1.1 and Algorithm 1.2 that describe the Row and Column DWT Filtering process respectively.

Algorithm 1.0 Level 1: DWT Filtering

```
Define:
$R0 - odd image pixel (reads from: row filtering - $s0, column filtering - $s0)
$R1 - even image pixel (reads from: row filtering - $s1, column filtering - $s64)
$s576 - division value, 2; $s577 - half image column size, 32
$s578 - image row size to be processed, 4; $s579 - full image column size, 64
$t6 - column counter; $t5 - row counter
$t4 - temporary variable for use in jumping programme instruction
$t3 - temporary variable for storing divided value
$t2 - temporary variable for storing high pass coefficients
Initialisation data address for row filtering: $R0 = 2; $R1 = 3
Initialisation for row filtering: $t5 = -4
STEP 1: Perform level 1 row filtering onto the image
do level 1 DWT row filtering (Algo. 1.1)
STEP 2: Reset the data address back to initial settings by decreasing the data address
Initialisation: $t5 = -4
while $t5 < 0 do
    Decrease $R0 data address by $s579
    Decrease $R1 data address by $s579
    $t5 = $t5 + 1
end while
Initialisation data address for column filtering: $R0 = 2; $R1 = 66
Initialisation for column filtering: $t5 = -2
STEP 3: Perform level 1 column filtering onto the image
do level 1 DWT column filtering (Algo. 1.2)
STEP 4: Reset the data address back to initial settings by decreasing the data address
Initialisation: $t5 = -4
while $t5 < 0 do
    Decrease $R0 data address by $s579 (64)
    Decrease $R1 data address by $s579 (64)
    $t5 = $t5 + 1
end while
```

Algorithm 1.1 Level 1: DWT Filtering (Row Filtering)

```
while $t5 < 0 do
    Initialisation: $t6 = -32
    while $t6 < 0 do
        STEP 1: Determine high pass coefficient, $R1
        $R1 = $R1 - $R0
        Initialisation: $t2 = 0; $t3 = 0; $t4 = 0
        STEP 2: Determine low pass coefficient, $R0
        if $R1 is positive then
            $t2 = -$R1
            while $t2 < 0 do
                $t2 = $t2 - 2
                if $t2 < 0 then
                    $t3 = $t3 + 1
                end if
            end while
            $R0 = $R0 + $t3 = $R0 + 0.5*$R1
        else
            $t2 = $R1
            while $t2 < 0 do
                $t2 = $t2 - 2
                if $t2 < 0 then
                    $t3 = $t3 + 1
                end if
            end while
            $R0 = $R0 - $t3 = $R0 + 0.5*(-$R1)
        end if
        $t6 = $t6 + 1
        STEP 3: Update the data address in the programme memory
        Increase $R0 data address by 2
        Increase $R1 data address by 2
    end while
    $t5 = $t5 + 1
end while
```

Algorithm 1.2 Level 1: DWT Filtering (Column Filtering)

```
while $t5 < 0 do
    Initialisation: $t6 = -64
    while $t6 < 0 do
        STEP 1: Determine high pass coefficient, $R1
        $R1 = $R1 - $R0
        Initialisation: $t2 = 0; $t3 = 0; $t4 = 0
        STEP 2: Determine low pass coefficient, $R0
        if $R1 is positive then
            $t2 = -$R1
            while $t2 < 0 do
                $t2 = $t2 - 2
                if $t2 < 0 then
                    $t3 = $t3 + 1
                end if
            end while
            $R0 = $R0 + $t3 = $R0 + 0.5*$R1
        else
            $t2 = $R1
            while $t2 < 0 do
                $t2 = $t2 - 2
                if $t2 < 0 then
                    $t3 = $t3 + 1
                end if
            end while
            $R0 = $R0 - $t3 = $R0 + 0.5*(-$R1)
        end if
        $t6 = $t6 + 1
        STEP 3: Update the data address in the programme memory
        Increase $R0 data address by 1
        Increase $R1 data address by 1
    end while
    $t5 = $t5 + 1
    STEP 4: Increase the data address to the 3rd and 4th row of coefficients
    Increase $R0 data address by $s579 (64)
    Increase $R1 data address by $s579 (64)
end while
```

Once the 1st Level DWT Filtering process was completed, the MISC processor continued to perform the 2nd Level DWT Filtering onto the LL_1 coefficients. 2nd Level DWT Filtering process was performed as to have an additional reduction of image data size such that lower amount of image data could be transferred across the WVSNs, especially when the sensor nodes were low in energy resources (low battery power). For better visualisation, the Level 2 DWT coefficients were then arranged in the manner illustrated in Figure 118. However, the actual arrangement of Level 2 DWT coefficients is a continuous data memory as shown in the Figure 119.

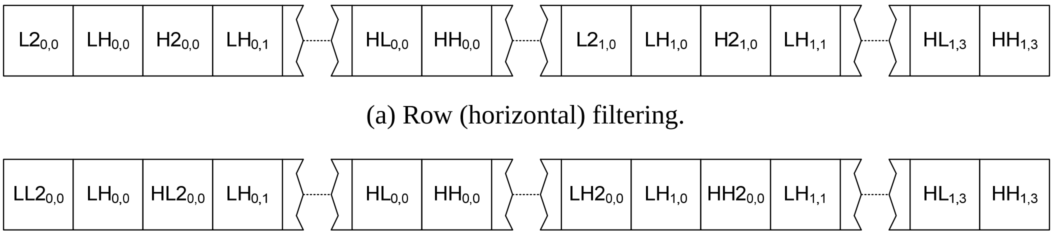
$L2_{0,0}$	$LH_{0,0}$	$H2_{0,0}$	$LH_{0,1}$	$L2_{0,1}$	$LH_{0,2}$	$H2_{0,1}$	$LH_{0,3}$
$HL_{0,0}$	$HH_{0,0}$	$HL_{0,1}$	$HH_{0,1}$	$HL_{0,2}$	$HH_{0,2}$	$HL_{0,3}$	$HH_{0,3}$
$L2_{1,0}$	$LH_{1,0}$	$H2_{1,0}$	$LH_{1,1}$	$L2_{1,1}$	$LH_{1,2}$	$H2_{1,1}$	$LH_{1,3}$
$HL_{1,0}$	$HH_{1,0}$	$HL_{1,1}$	$HH_{1,1}$	$HL_{1,2}$	$HH_{1,2}$	$HL_{1,3}$	$HH_{1,3}$

(a) Row (horizontal) filtering.

$LL2_{0,0}$	$LH_{0,0}$	$HL2_{0,0}$	$LH_{0,1}$	$LL2_{0,1}$	$LH_{0,2}$	$HL2_{0,1}$	$LH_{0,3}$
$HL_{0,0}$	$HH_{0,0}$	$HL_{0,1}$	$HH_{0,1}$	$HL_{0,2}$	$HH_{0,2}$	$HL_{0,3}$	$HH_{0,3}$
$LH2_{0,0}$	$LH_{1,0}$	$HH2_{0,0}$	$LH_{1,1}$	$LH2_{0,1}$	$LH_{1,2}$	$HH2_{0,1}$	$LH_{1,3}$
$HL_{1,0}$	$HH_{1,0}$	$HL_{1,1}$	$HH_{1,1}$	$HL_{1,2}$	$HH_{1,2}$	$HL_{1,3}$	$HH_{1,3}$

(b) Column (vertical) filtering.

Figure 118 Level 2 DWT coefficients arrangement in 2D.



(a) Row (horizontal) filtering.

(b) Column (vertical) filtering.

Figure 119 Actual Level 2 DWT coefficients arrangement in memory.

Again, the LL_1 DWT coefficients undergone Row (Horizontal) Filtering and then followed by Column (Vertical) Filtering, which is described in Algorithm 2.0. The Algorithm 2.0 shows that the algorithm is broken up into Algorithm 2.1 for DWT Row Filtering and Algorithm 2.2 for DWT Column Filtering. Once both Row and Column DWT Filtering was performed, the DWT CRS MISC processor had completed performing the 2 Levels 2-dimensional DWT Filtering onto the image data.

Algorithm 2.0 Level 2: DWT Filtering

```
Define:
$R0 - odd image pixel (reads from: row filtering - $s0, column filtering - $s0)
$R1 - even image pixel (reads from: row filtering - $s2, column filtering - $s128)
$s576 - division value, 2; $s577 - half image column size, 32
$s578 - image row size to be processed, 4; $s579 - full image column size, 64
$s580 - level 2 subband column size. 16
$t8 - temporary variable to store -64; $t7 - temporary variable to store -4
$t6 - column counter; $t5 - row counter
$t4 - temporary variable for use in jumping programme instruction
$t3 - temporary variable for storing divided value
$t2 - temporary variable for storing high pass coefficients
Initialisation data address for row filtering: $R0 = 2; $R1 = 4
Initialisation for row filtering: $t5 = -2
STEP 1: Perform level 2 row filtering onto the image
do level 1 DWT row filtering (Algo. 2.1)
STEP 2: Reset the data address back to initial settings by decreasing the data address
Initialisation: $t5 = -2
while $t5 < 0 do
    Decrease $R0 data address by $s579
    Decrease $R1 data address by $s579
    $t5 = $t5 + 1
end while
Initialisation data for column filtering: $R0 = 2; $R1 = 130
STEP 3: Perform level 2 column filtering onto the image
do level 1 DWT column filtering (Algo. 2.2)
STEP 4: Reset the data address back to initial settings by decreasing the data address
Decrease $R0 data address by $s579 (64)
Decrease $R1 data address by $s579 (64)
```

Algorithm 2.1 Level 2: DWT Filtering (Row Filtering)

```
while $t5 < 0 do
    Initialisation: $t6 = -16
    while $t6 < 0 do
        STEP 1: Determine high pass coefficient, $R1
        $R1 = $R1 - $R0
        Initialisation: $t2 = 0; $t3 = 0; $t4 = 0
        STEP 2: Determine low pass coefficient, $R0
        if $R1 is positive then
            $t2 = -$R1
            while $t2 < 0 do
                $t2 = $t2 - 2
                if $t2 < 0 then
                    $t3 = $t3 + 1
                end if
            end while
            $R0 = $R0 + $t3 = $R0 + 0.5*$R1
        else
            $t2 = $R1
            while $t2 < 0 do
                $t2 = $t2 -2
                if $t2 < 0 then
                    $t3 = $t3 + 1
                end if
            end while
            $R0 = $R0 - $t3 = $R0 + 0.5*(-$R1)
        end if
        $t6 = $t6 + 1
        STEP 3: Update the data address in the programme memory
        Increase $R0 data address by 4
        Increase $R1 data address by 4
    end while
    $t5 = $t5 + 1
    STEP 4: Update the data address in the programme memory to next row of LL1 coefficients
    Increase $R0 data address by $s579 (64)
    Increase $R1 data address by $s579 (64)
end while
```

Algorithm 2.2 Level 2: DWT Filtering (Column Filtering)

```
Initialisation: $t6 = -32
while $t6 < 0 do
    STEP 1: Determine high pass coefficient, $R1
    $R1 = $R1 - $R0
    Initialisation: $t2 = 0; $t3 = 0; $t4 = 0
    STEP 2: Determine low pass coefficient, $R0
    if $R1 is positive then
        $t2 = -$R1
        while $t2 < 0 do
            $t2 = $t2 - 2
            if $t2 < 0 then
                $t3 = $t3+ 1
            end if
        end while
        $R0 = $R0 + $t3 = $R0 + 0.5*$R1
    else
        $t2 = $R1
        while $t2 < 0 do
            $t2 = $t2 -2
            if $t2 < 0 then
                $t3 = $t3+ 1
            end if
        end while
        $R0 = $R0 - $t3 = $R0 + 0.5*(-$R1)
    end if
    $t6 = $t6 + 1
    STEP 3: Update the data address in the programme memory
    Increase $R0 data address by 2
    Increase $R1 data address by 2
end while
```

3.6.3 Cauchy Reed Solomon Coding Scheme

The Reed Solomon (RS) coding scheme works in the infinite *Z* field [20]. However, the computers that were available to perform computation in binary words of a fixed word length *L*. As a result, the RS codes are implemented over a Galois Field with 2^L elements, represented as GF(2^L). The elements for GF(2^L) are made up of integer numbers from 0 to $2^L - 1$, with each element comprising of *L* bits word length. In order to ensure that the computational of GF(2^L) is correct, the GF(2^L) should contain at least more than $n + m$ elements, where $2^L > m+n$ [20] [59].

From the mentioned RS coding scheme in Section 2.4.2, part of the encoded codeword comprises the message word itself, which is called as the Systematic Code [59] [94]. Instead of using the Vandermonde matrix, to encode the data with RS coding scheme, the Cauchy Reed Solomon (CRS) coding scheme can also be used to encode the data. The CRS coding scheme uses the Cauchy matrix to encode (encrypt) the data and it also provides the similar error protection capabilities. The CRS coding scheme is also known as the Secure Erasure Code (SEC) that was introduced in [20]. In order to decode the encoded (encrypted) data, the generator matrix *G* of the particular CRS coding scheme is required. If the generator matrix *G* is kept as a secret key, even the adversary that manages to retrieve sufficient number of fragments, he/she will not be able to reproduce the original data. Based on the CRS coding scheme, the data is encrypted by using the generator matrix G that is created from a user specified key [20].

For an $m \times n$ matrix *G* over GF(2^L), this matrix is known as a Secure Generator matrix of the SEC scheme if and only if the matrix *G* is non-singular. The generator matrix *G* must also not consist of any $(m-1)\times(n-1)$ singular matrix. With the Cauchy matrices which is considered to be an important subclass of non-singular matrices over GF(2^L) [20], the generator matrix G_{SEC} can be constructed by customizing the Cauchy matrix, which is defined as follows.

$$x_i \neq x_j \quad \forall\, i, j \in \{1, \ldots, m\}, i \neq j \tag{34}$$

$$y_i \neq y_j \quad \forall\, i, j \in \{1, \ldots, n\}, i \neq j \tag{35}$$

$$x_i \neq y_j \quad \forall\, i \in \{1, \ldots, m\}, \forall\, j \in \{1, \ldots, n\} \tag{36}$$

$$\begin{bmatrix} \frac{1}{x_1 + y_1} & \frac{1}{x_1 + y_2} & \ldots\ldots & \frac{1}{x_1 + y_n} \\ \frac{1}{x_2 + y_1} & \frac{1}{x_2 + y_2} & \ldots\ldots & \frac{1}{x_2 + y_n} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \frac{1}{x_m + y_1} & \frac{1}{x_m + y_2} & \ldots\ldots & \frac{1}{x_m + y_n} \end{bmatrix} \tag{37}$$

First, let $\{x_1, \cdots\cdots, x_m\}$ and $\{y_1, \cdots\cdots, y_n\}$ to be the two sets of elements in Galois Field GF(2^L) with both sets *x* and *y* must satisfy the following constraints stated in Equation (34) to Equation (37). Therefore, the matrix stated in Equation (37) will be known as the Cauchy matrix over GF(2^L). Every square submatrix of this

Cauchy matrix is non-singular where the square Cauchy matrix is invertible, provided that the determinant matrix is not zero, $\det(G) \neq 0$.

The generator matrix G_{SEC} is made up of a large matrix which is impractical to keep it as a secret key. Consequently, the G_{SEC} can be constructed by using a user defined l_k bit secret key k_G. Since the G_{SEC} is an $m \times n$ matrix over GF(2^L), the algorithm parameters needed besides k_G will be m and n. Hence the value of l_k needs to be in a multiple of $m+n$, which constitute $L = l_k /(m+n)$ [20] [59]. Then the value of k_G can be divided into a vector consisting of $m+n$ elements, where each of the elements is a bit string of size L. Considering that the redundancy rate as $r = n/m$, the following Equation (38) and Equation (39) are the constraints of the parameters in generating the key [20].

$$L > \log_2(m+n) = \log_2[m \times (1+r)] \tag{38}$$

$$l_k = L \times (m+n) = L \times m \times (1+r)] \tag{39}$$

To have the original data to be encoded, the data will be segmented into vectors, with each of them having m bits strings of size L. Hence, the vector-matrix multiplication yields an encoded vector of size n for each vector. Nevertheless, all the i^{th} elements in these encoded vectors will produce the i^{th} encoded fragment and finally n encoded fragments are obtained [20]. Considered that the original data fragments can be viewed as a vector $D = (D_1, D_2, \ldots\ldots, D_m)$ and the encoding function *Enc* that maps the original data fragments onto the encoded fragments, $Enc(D) = E$ with the vector $E = (E_1, E_2, \ldots\ldots, E_n)$. The Reed Solomon encoding is described as the vector-matrix multiplication $Enc(D) = DG$, where G is the generator matrix or G_{SEC} is the SEC generator matrix [20] [59]. Equation (40) shows how the data in the vector E is being encoded through the use of G_{SEC}.

$$\begin{aligned} E &= Enc(D) \\ &= D \times G_{SEC} \\ &= \begin{bmatrix} D_1 & \ldots\ldots & D_m \end{bmatrix} \begin{bmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \ldots\ldots & \frac{1}{x_1+y_n} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \ldots\ldots & \frac{1}{x_2+y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_m+y_1} & \frac{1}{x_m+y_2} & \ldots\ldots & \frac{1}{x_m+y_n} \end{bmatrix} \\ &= \begin{bmatrix} E_1 & \ldots\ldots & E_n \end{bmatrix} \end{aligned} \tag{40}$$

$$\begin{bmatrix} D_1 & \ldots\ldots & D_m \end{bmatrix} \begin{bmatrix} \frac{1}{x_1+y_{i_1}} & \frac{1}{x_1+y_{i_2}} & \ldots\ldots & \frac{1}{x_1+y_{i_m}} \\ \frac{1}{x_2+y_{i_1}} & \frac{1}{x_2+y_{i_2}} & \ldots\ldots & \frac{1}{x_2+y_{i_m}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_m+y_{i_1}} & \frac{1}{x_m+y_{i_2}} & \ldots\ldots & \frac{1}{x_m+y_{i_m}} \end{bmatrix} = \begin{bmatrix} E_{i_1} & \ldots\ldots & E_{i_m} \end{bmatrix} \tag{41}$$

$$P\left(2^L, 2m\right) = P\left(2^{\frac{l_k}{(l+r)\times m}}, 2m\right) \tag{42}$$

Supposedly if an *m* encoded fragments of data $E = \left(E_{i_1}, E_{i_2}, \ldots\ldots, E_{i_m}\right)$ has been retrieved by the adversary from the initial *n* number of encoded fragments of data. Then this implies that only one sequence of $\{x_1, \ldots\ldots, x_m, y_{i_1}, \ldots\ldots, y_{i_m}\}$ that can generate the $m \times m$ matrix. This matrix will be used in Equation (41) to recover the original data from *m* encoded fragments. Consequently, the adversary can only guess the particular sequence over GF(2^L) for decrypting the encoded data [20]. Since there is only *2m* of different elements required in that sequence, the adversary needs to figure out a total of $P\left(2^L, 2m\right)$ times for the worst case scenario. From the stated Equation (42), it can be seen that at a particular fix l_k and *r* values, the strength of the key is determined by *m*.

3.6.4 CRS Encoding Algorithm

After completing the 2 Levels 2-dimensional DWT Filtering process, the DWT CRS MISC processor continued to perform the CRS(20,16) encoding onto all the produced Level 2 DWT coefficients. Based on the CRS coding scheme method in Section 2.4.3, the Secure Generator Matrix, G_{SEC} is stored in the Memory of the MISC processor. For each 16 DWT coefficients, the processor performed Galois Field (GF) arithmetic multiplication and addition onto the DWT coefficients with the Secure Generator Matrix. Before GF arithmetic operation was performed, the DWT coefficients were undergone a 11-bit to 8-bit conversion, such that the DWT coefficients are encoded by the CRS coding scheme that required 8-bit wide word length data. A complete 20 symbols (Bytes) of codeword was obtained from the GF arithmetic operations performed between the DWT coefficients and the Secure Generator Matrix. After completing the first codeword, another 16 DWT coefficients were processed to produce the second, third and fourth codeword. Algorithm 3.0 shows the complete process of CRS encoding performed by the DWT CRS MISC processor onto the DWT coefficients. As for Algorithm 3.2, it shows how each codeword was produced by performing GF arithmetic operation onto the DWT coefficients. Algorithm 3.3 shows how a particular symbol of codeword was generated from the GF arithmetic operation performed.

Algorithm 3.0 Cauchy Reed Solomon

```
Define:
$R0 - to be clear variables (from $t8 - $t90)
$R1 - target write CRS value (from $t11 - $t90)
$R2 - read data value (reads from $s0 - $s190)
$R3 - read generator matrix value (reads from $s256 - $s575)
$s576 - division value, 2; $s577 - half image column size, 32
$s582 - value of number of data to be clear for CRS, 83
$s583 - number of generator matrix coefficients, 320
$t5 - counter for row of level 2 DWT coefficients
$t7 - counter for next codeword in the same row
$t8 - counter for complete codeword; $t9 - counter for 16 input symbols
Initialisation: $t7 = -83
STEP 1: Clear previously processed data
do clear previous data (Algo. 3.1)
Initialisation data address: $R1 = 608; $R2 = 2; $R3 = 258
Initialisation for CRS encoding: $t6 = -32; $t5 = -2; $t4 = -64
STEP 2: Perform CRS encoding on all the level 2 DWT coefficients
while $t5 < 0 do
    Initialisation: $t7 = -2
    STEP 3: Encode the next codeword from next 16 input symbols
    while $t7 < 0 do
            Initialisation: $t8 = -20
            STEP 4: Encode a complete codeword with 20 symbols
            do encode complete codeword (Algo. 3.2)
            $t7 = $t7 + 1
    end while
    Increase $R2 data address by 64
    $t5 = $t5 + 1
end while
Initialisation: $t7 = -4
STEP 5: Reset data address back to initial settings
do reset data address (Algo. 3.4)
```

Algorithm 3.1 Cauchy Reed Solomon (Clear Previous Data)

```
STEP 1: Clear the previous data in the memory
Initialisation data address: $R0 = 605
while $t7 < 0 do
    $R0 = 0
    Increase $R0 data address by 1
    $t7 = $t7 + 1
end while
STEP 2: Reset the data address back to initial settings
Decrease $R0 data address by $s582 (83)
```

Algorithm 3.2 Cauchy Reed Solomon (Encode Complete Codeword)

```
while $t8 < 0 do
    Initialisation: $t9 = -16
    STEP 1: Encode each output symbol from 16 input symbols
    do encoding for each output symbol (Algo. 3.3)
    $t8 = $t8 + 1
end while
Decrease $R3 data address by 320
Increase $R2 data address by 32
```

Algorithm 3.3 Cauchy Reed Solomon (Encode Codeword Symbol)

```
while $t9 < 0 do
    $t10 = $R2
    Convert $t10 from 11-bit to 8-bit
    $t10 = $t10 x $R3, GF Multiplication
    $R1 = $R1 + $t10
    Initialisation: $t10 = 0
    Increase $R2 data address by 2
    Increase $R3 data address by 1
    $t9 = $t9 + 1
end while
Increase $R1 data address by 1
Decrease $R2 data address by 32
```

Algorithm 3.4 Cauchy Reed Solomon (Reset Data Address)

```
while $t7 < 0 do
    Decrease $R1 data address by 20
    Decrease $R2 data address by 64
    $t7 = $t7 + 1
end while
```

3.7 PROGRAMME INSTRUCTIONS/CLOCK CYCLES

In this section, the programme instructions were written for the developed MISC processor to perform image data processing. The programme instructions can be broken into 3 different parts, which are Level 1 Lifting Scheme DWT, Level 2 Lifting Scheme DWT and CRS(20,16) encoding scheme. Each part of the programme instructions are discussed in the following Sections. The programme instructions are executed in sequence, where the programme instructions on Level 1 Lifting Scheme DWT were executed first. Then it is followed by executing programme instructions on Level 2 Lifting Scheme DWT and finally executing the CRS(20,16) encoding scheme programme instructions.

3.7.1 Level 1 Lifting Scheme DWT Programme

There are 90 lines of programme instructions written and programmed onto the DWT CRS MISC, such that it performs the 1st Level Lifting Scheme DWT Filtering onto the image data size (64 x 4 pixels). Listing 1 shows the 1st Level 2-dimensional Lifting Scheme DWT Filtering programme instructions that covered both DWT Row (Horizontal) Filtering and DWT Column (Vertical) Filtering processes.

In the DWT Row Filtering, there are groups of programme instructions are executed repeatedly, as shown in Listing 1(a). Programme instructions at address 0x2C8 to 0x313 are repeated for 32 times to process the image data by performing the 1st Level DWT Row Filtering. This produces both 1st Level Low Pass (L_1) and High Pass (H_1) subband of DWT coefficients. Once the MISC completed processing the first row of image data, these programme instructions were executed again to process the next 3 rows of the image data. Therefore, these programme instructions at address 0x2C8 to 0x313 were repeated for 4 times, including two additional instructions at address 0x2C5 and 0x316. After processing the image data, the programme instructions at address 0x31C to 0x331 are repeated for 4 times to reset the modified data address of the programme instructions at address 0x2C8, 0x2C9, 0x2D5, 0x2D7, 0x2EA, 0x2EF, 0x2FC back to initial data address. This is done by subtracting the address value by 64 each time.

```
Addr.   Instructions
0x2B0   XOR   $t0     $t0     0
0x2B3   XOR   $t1     $t1     0
0x2B6   XOR   $t5     $t5     0
0x2B9   XOR   $t6     $t6     0
0x2BC   SBN   $one    $t0     0
0x2BF   SBN   $s576   $t1     0
0x2C2   SBN   $s578   $t5     0
0x2C5   SBN   $s577   $t6     0
0x2C8   SBN   $s0     $s1     0      Repeat
0x2CB   XOR   $t2     $t2     0        x32
0x2CE   XOR   $t3     $t3     0
0x2D1   XOR   $t4     $t4     0
0x2D4   SBN   $zero   $s1     0x018
0x2D7   XOR   $s1     $t2     0
0x2DA   SBN   $s576   $t2     0x006
0x2DD   SBN   $t0     $t3     0
0x2E0   SBN   $one    $t4     0x7F7
0x2E3   XOR   $t2     $t2     0
0x2E6   SBN   $t3     $t2     0
0x2E9   SBN   $t2     $s0     0
0x2EC   SBN   $one    $t4     0x00F          Repeat
0x2EF   SBN   $s1     $t2     0                x4
0x2F2   SBN   $s576   $t2     0x006
0x2F5   SBN   $t0     $t3     0
0x2F8   SBN   $one    $t4     0x7F7
0x2FB   SBN   $t3     $s0     0
0x2FE   SBN   $t1     0x2C8   0
0x301   SBN   $t1     0x2C9   0
0x304   SBN   $t1     0x2D5   0
0x307   SBN   $t1     0x2D7   0
0x30A   SBN   $t1     0x2EA   0
0x30D   SBN   $t1     0x2EF   0
0x310   SBN   $t1     0x2FC   0
0x313   SBN   $t0     $t6     0x7B2
0x316   SBN   $t0     $t5     0x7AC
0x319   SBN   $s578   $t5     0
0x31C   SBN   $s579   0x2C8   0      Repeat
0x31F   SBN   $s579   0x2C9   0        x4
0x322   SBN   $s579   0x2D5   0
0x325   SBN   $s579   0x2D7   0
0x328   SBN   $s579   0x2EA   0
0x32B   SBN   $s579   0x2EF   0
0x32E   SBN   $s579   0x2FC   0
0x331   SBN   $t0     $t5     0x7E8
```

(a) Row (Horizontal) Filtering.

```
Addr.   Instructions
0x334   SBN   $s576   $t5     0
0x337   SBN   $s579   $t6     0
0x33A   SBN   $s0     $s64    0      Repeat
0x33D   XOR   $t2     $t2     0        x64
0x340   XOR   $t3     $t3     0
0x343   XOR   $t4     $t4     0
0x346   SBN   $zero   $s64    0x018
0x349   XOR   $s64    $t2     0
0x34C   SBN   $s576   $t2     0x006
0x34F   SBN   $t0     $t3     0
0x352   SBN   $one    $t4     0x7F7
0x355   XOR   $t2     $t2     0
0x358   SBN   $t3     $t2     0
0x35B   SBN   $t2     $s0     0
0x35E   SBN   $one    $t4     0x00F
0x361   SBN   $s64    $t2     0
0x364   SBN   $s576   $t2     0x006
0x367   SBN   $t0     $t3     0
0x36A   SBN   $one    $t4     0x7F7          Repeat
0x36D   SBN   $t3     $s0     0                x2
0x370   SBN   $t0     0x33A   0
0x373   SBN   $t0     0x33B   0
0x376   SBN   $t0     0x347   0
0x379   SBN   $t0     0x349   0
0x37C   SBN   $t0     0x35C   0
0x37F   SBN   $t0     0x361   0
0x382   SBN   $t0     0x36E   0
0x385   SBN   $t0     $t6     0x7B2
0x388   SBN   $s579   $t6     0
0x38B   SBN   $t6     0x33A   0
0x38E   SBN   $t6     0x33B   0
0x391   SBN   $t6     0x347   0
0x394   SBN   $t6     0x349   0
0x397   SBN   $t6     0x35C   0
0x39A   SBN   $t6     0x361   0
0x39D   SBN   $t6     0x36E   0
0x3A0   SBN   $t0     $t5     0x797
0x3A3   SBN   $s578   $t5     0
0x3A6   SBN   $s579   0x33A   0      Repeat
0x3A9   SBN   $s579   0x33B   0        x4
0x3AC   SBN   $s579   0x347   0
0x3AF   SBN   $s579   0x349   0
0x3B2   SBN   $s579   0x35C   0
0x3B5   SBN   $s579   0x361   0
0x3B8   SBN   $s579   0x36E   0
0x3BB   SBN   $t0     $t5     0x7E8
```

(b) Column (Vertical) Filtering.

Listing 1 Level 1 Lifting Scheme DWT Filtering programme instructions.

For the DWT column filtering, there are also programme instructions to be executed repeatedly, as shown in Listing 1(b). Programme instructions at address 0x33A to 0x385 are repeated for 64 times to process the coefficients by performing the 1st Level DWT Column Filtering. This produces 1st Level of Low Pass Low Pass (LL$_1$), Low Pass High Pass (LH$_1$), High Pass Low Pass (HL$_1$) and High Pass High Pass (HH$_1$) subband of DWT coefficients. Once the MISC completed processing the first 2 rows of DWT coefficients, these programme instructions continued to process the subsequent 2 row of DWT coefficients. Therefore, these programme instructions at address 0x33A to 0x385 were repeated for 2 times, which includes the 10 programme instructions at address 0x337, 0x388 to 0x3A0. After completing the 1st

Level DWT Filtering process, the programme instructions at address 0x3A6 to 0x3BB were repeated for 4 times to have the modified data address of the programme instructions at address 0x33A, 0x33B, 0x347, 0x349, 0x35C, 0x361, 0x36E to be reset back to the initial data address value.

In the Row Filtering, High Pass DWT coefficients were determined first from the image data available in the Memory. Then the programme instructions at address 0x2D7 to 0x2FB were executed to determine the Low Pass DWT coefficients. The Low Pass coefficients were determined by dividing the High Pass coefficients and adding it with odd image pixels. For these programme instructions at address 0x2D7 to 0x2FB, they are selectively executed based on the value of High Pass coefficients determined previously. If positive High Pass coefficients were obtained, then 8 programme instructions at address 0x2D7 to 0x2EC were executed to determine the Low Pass coefficients. If negative High Pass coefficients were obtained, 5 written programme instructions at address 0x2EF to 0x2FB were executed instead. In each of the (positive/negative) division programme instructions, the number of repetition in executing these three lines of instructions at address 0x2DA to 0x2E0 or 0x2F2 to 0x2F8 were depended on the High Pass coefficients value. As for the Column Filtering, the same programme instructions at address 0x349 to 0x36D were written and executed to determine the Low Pass coefficients for the corresponding Row Filtered DWT coefficients.

$$\begin{aligned}\text{Row Filtering Instructions} &= 4\times\left\{32\times\left[5+8+16\times3+\frac{1}{2}(5+2)\right]+2\right\}+8+4\times8\\ &= 4\times\{32\times[5+8+48+4]+2\}+8+32\\ &= 8{,}368\ \text{instructions executed}\end{aligned}\tag{43}$$

$$\begin{aligned}\text{Column Filtering Instructions} &= 2\times\left\{64\times\left[5+8+16\times3+\frac{1}{2}(5+2)\right]+9\right\}+3+4\times8\\ &= 2\times\{64\times[5+8+48+4]+9\}+2+32\\ &= 8{,}372\ \text{instructions executed}\end{aligned}\tag{44}$$

$$\begin{aligned}\text{Total Instructions} &= \text{Column Filtering Instructions} + \text{Row Filtering Instructions}\\ &= 8{,}368+8{,}372\\ &= 16{,}739\ \text{instructions executed}\end{aligned}\tag{45}$$

$$\begin{aligned}\text{Total Clock Cycles} &= 16{,}739\ \text{instructions executed}\times 9\ \text{clock cycles}\\ &= 150{,}651\ \text{clock cycles}\end{aligned}\tag{46}$$

In order to determine the Low Pass coefficients, estimations on the number of programme instructions executed were made in the followings. Therefore, the average number of instructions required to perform the division of High Pass coefficients are estimated to be, 16 x 3 + ½ x (5 + 2) = 52 programme instructions. The estimated values is calculated based on the assumption that the average High Pass DWT coefficients values were 32. Therefore, 16 times of division by 2 was performed onto the assume High Pass DWT coefficients values. The number of instructions were executed to perform the 1st Level DWT Row Filtering were estimated to be 8,368 instructions, which is determined in Equation (43). Next, a complete 1st Level DWT Column Filtering process required the MISC to execute 8,372 instructions, which is shown in Equation (44). In Equation (45), the combined of both 1st Level of Row Filtering and Column Filtering processes required a total of 16,739 instructions to be executed. Based on each instruction requires 9 clock cycles, the total clock cycles required to perform the Level 1 DWT Filtering are 150,651 cycles, which is calculated in Equation (46).

3.7.2 Level 2 Lifting Scheme DWT Programme

For 2nd Level 2-dimensional DWT Filtering, a total of 84 lines of programme instructions were written and programmed onto the MISC architecture. The written Level 2 DWT Filtering programme instructions are shown in Listing 2. The Level 2 DWT Filtering programme instructions covered both DWT Row (Horizontal) Filtering and DWT Column (Vertical) Filtering.

Similar to Level 1 Lifting Scheme DWT, the Level 2 DWT Row Filtering consists many parts of the programme instructions that were repeatedly executed, shown in Listing 2(a). However, the Level 2 DWT Row Filtering is only performed onto the LL_1 DWT coefficients produced from the 1st Level DWT Filtering process. Therefore, the programme instructions at address 0x3D3 to 0x41E were repeated for 16 times to process the LL_1 DWT coefficients by performing 2nd Level DWT row Filtering on these coefficients. This produces the Level 2 Low Pass (L_2) and High Pass (H_2) subband of DWT coefficients. Once the MISC completed processing the first row of LL_1 coefficients, these programme instructions are executed again to process the next rows of LL_1 coefficients. Therefore, these programme instructions at address 0x3D0 to 0x436 were repeated for 2 times. After completed processing the image, the programme instructions at address 0x43C to 0x451 were repeated for 4 times to reset the modified data address of the programme instructions at address 0x3D3, 0x3D4, 0x3E0, 0x3E2, 0x3F5, 0x3FA, 0x407 back to initial data address. This is done by subtracting the address value by 64 each time.

For 2nd Level DWT column filtering, there are many parts of the programme instructions were repeatedly executed, as shown in Listing 2(b). Programme instructions at address 0x457 to 0x4A2 were repeated for 32 times to process the 2nd Level DWT coefficients by performing the 2nd Level DWT Column Filtering. This produces 2nd Level of Low Pass Low Pass (LL_2), Low Pass High Pass (LH_2), High Pass Low Pass (HL_2) and High Pass High Pass (HH_2) subband of DWT coefficients from the DWT Row Filtered coefficients. After completed the 2nd Level DWT Column Filtering process, the programme instructions at address 0x4A5 to 0x4B7 were executed such that the modified data address of the programme instructions at address 0x457, 0x458, 0x464, 0x466, 0x479, 0x47E, 0x48B were reset back to the initial data address value.

The number of instructions were executed to perform the 2nd Level DWT Row Filtering are 2,137 instructions, which is determined from the Equation (47). Next, for a complete 2nd Level DWT Column Filtering process, the MISC required to execute a total of 2,098 instructions, which is shown in Equation (48). In Equation (49), the combine 2nd Level of Row Filtering and Column Filtering processes required a total 4,235 instructions to be executed. Since each programme instruction requires 9 Clock Cycles, the total Clock Cycles required by the MISC to perform the Level 2 DWT Filtering were 38,115 Clock Cycles, as calculated in Equation (50).

```
Addr.   Instructions
0x3BE   XOR   $t6     $t6     0
0x3C1   XOR   $t7     $t7     0
0x3C4   XOR   $t8     $t8     0
0x3C7   SBN   $s579   $t8     0
0x3CA   SBN   $s578   $t7     0
0x3CD   SBN   $s576   $t5     0
0x3D0   SBN   $s580   $t6     0
0x3D3   SBN   $s0     $s2     0      Repeat
0x3D6   XOR   $t2     $t2     0       x16
0x3D9   XOR   $t3     $t3     0
0x3DC   XOR   $t4     $t4     0
0x3DF   SBN   $zero   $s2     0x018
0x3E2   XOR   $s2     $t2     0
0x3E5   SBN   $s576   $t2     0x006
0x3E8   SBN   $t0     $t3     0
0x3EB   SBN   $one    $t4     0x7F7
0x3EE   XOR   $t2     $t2     0
0x3F1   SBN   $t3     $t2     0
0x3F4   SBN   $t2     $s0     0
0x3F7   SBN   $one    $t4     0x00F
0x3FA   SBN   $s2     $t2     0
0x3FD   SBN   $s576   $t2     0x006
0x400   SBN   $t0     $t3     0
0x403   SBN   $one    $t4     0x7F7         Repeat
0x406   SBN   $t3     $s0     0               x2
0x409   SBN   $t7     0x3D3   0
0x40C   SBN   $t7     0x3D4   0
0x40F   SBN   $t7     0x3E0   0
0x412   SBN   $t7     0x3E2   0
0x415   SBN   $t7     0x3F5   0
0x418   SBN   $t7     0x3FA   0
0x41B   SBN   $t7     0x407   0
0x41E   SBN   $t0     $t6     0x7B2
0x421   SBN   $t8     0x3D3   0
0x424   SBN   $t8     0x3D4   0
0x427   SBN   $t8     0x3E0   0
0x42A   SBN   $t8     0x3E2   0
0x42D   SBN   $t8     0x3F5   0
0x430   SBN   $t8     0x3FA   0
0x433   SBN   $t8     0x407   0
0x436   SBN   $t0     $t5     0x797
0x439   SBN   $s578   $t5     0
0x43C   SBN   $s579   0x3D3   0      Repeat
0x43F   SBN   $s579   0x3D4   0       x4
0x442   SBN   $s579   0x3E0   0
0x445   SBN   $s579   0x3E2   0
0x448   SBN   $s579   0x3F5   0
0x44B   SBN   $s579   0x3FA   0
0x44E   SBN   $s579   0x407   0
0x451   SBN   $t0     $t5     0x7E8
```

(a) Row (Horizontal) Filtering.

```
Addr.   Instructions
0x454   SBN   $s577   $t6     0
0x457   SBN   $s0     $s128   0      Repeat
0x45A   XOR   $t2     $t2     0       x32
0x45D   XOR   $t3     $t3     0
0x460   XOR   $t4     $t4     0
0x463   SBN   $zero   $s128   0x018
0x466   XOR   $s128   $t2     0
0x469   SBN   $s576   $t2     0x006
0x46C   SBN   $t0     $t3     0
0x46F   SBN   $one    $t4     0x7F7
0x472   XOR   $t2     $t2     0
0x475   SBN   $t3     $t2     0
0x478   SBN   $t2     $s0     0
0x47B   SBN   $one    $t4     0x00F
0x47E   SBN   $s128   $t2     0
0x481   SBN   $s576   $t2     0x006
0x484   SBN   $t0     $t3     0
0x487   SBN   $one    $t4     0x7F7
0x48A   SBN   $t3     $s0     0
0x48D   SBN   $t1     0x457   0
0x490   SBN   $t1     0x458   0
0x493   SBN   $t1     0x464   0
0x496   SBN   $t1     0x466   0
0x499   SBN   $t1     0x479   0
0x49C   SBN   $t1     0x47E   0
0x49F   SBN   $t1     0x48B   0
0x4A2   SBN   $t0     $t6     0x7B2
0x4A5   SBN   $s579   0x457   0
0x4A8   SBN   $s579   0x458   0
0x4AB   SBN   $s579   0x464   0
0x4AE   SBN   $s579   0x466   0
0x4B1   SBN   $s579   0x479   0
0x4B4   SBN   $s579   0x47E   0
0x4B7   SBN   $s579   0x48B   0
```

(b) Column (Vertical) Filtering.

Listing 2 DWT Level 2 Filtering Programme Instructions.

$$\begin{aligned}\text{Row Filtering Instructions} &= 2\times\left\{16\times\left[5+8+16\times3+\frac{1}{2}(5+2)\right]+1+8\right\}+7+4\times8\\ &= 2\times\{16\times[5+8+48+4]+9\}+7+32\\ &= 2{,}137 \text{ instructions executed}\end{aligned} \qquad (47)$$

$$\begin{aligned}\text{Column Filtering Instructions} &= \left\{32\times\left[5+8+16\times3+\frac{1}{2}(5+2)\right]\right\}+1+7\\ &= \{32\times[5+8+48+4]+10\}+8\\ &= 2{,}098 \text{ instructions executed}\end{aligned} \qquad (48)$$

$$\begin{aligned}\text{Total Instructions} &= \text{Column Filtering Instructions} + \text{Row Filtering Instructions}\\ &= 2{,}137 + 2{,}098\\ &= 4{,}235 \text{ instructions executed}\end{aligned} \tag{49}$$

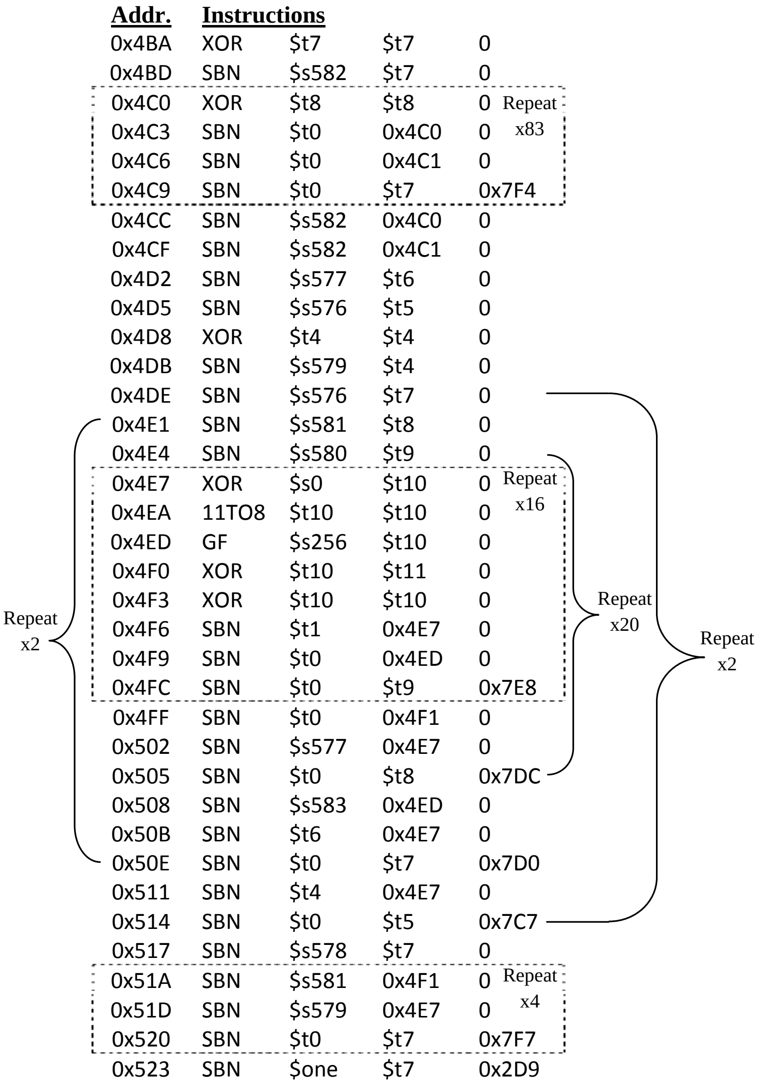
$$\begin{aligned}\text{Total Clock Cycles} &= 4{,}235 \text{ instructions executed} \times 9 \text{ clock cycles}\\ &= 38{,}115 \text{ clock cycles}\end{aligned} \tag{50}$$

3.7.3 Cauchy Reed Solomon Encoding Programme

In Listing 3, there are 36 lines of programme instructions were written to perform the CRS(20,16) encoding onto the Level 2 DWT coefficients. First, the programme instructions at address 0x4C0 to 0x4C9 were repeated for 83 times to clear any data in the temporary data memory from \$t8 to \$t90. Next, the CRS Encoding process was started by performing a multiplications between the data vector and Secure Generator Matrix G_{SEC}, to produce a complete codeword with 20 symbols.

For each time, the CRS Encoder programme encodes 16 DWT coefficients (input symbols) to produce a codeword with 20 encoded symbols. The programme instructions at address 0x4E7 to 0x4FC were repeatedly executed for 16 times such that the MISC performed GF multiplication between the 16 input symbols with corresponding 16 coefficients in the G_{SEC}. Then followed by GF addition on these 16 GF multiplication operation results to produce 1 codeword symbols. In order to produce the subsequence 19 symbols, the GF multiplication and addition operations were executed repeatedly for 19 times with the use of same input symbols and different G_{SEC} coefficients. Therefore, for the same 16 input symbols (DWT coefficients), the programme instructions at address 0x4E4 to 0x502 were repeatedly executed for 20 times. Details on CRS Coding Scheme are mentioned in Section 2.4.3. Once the first 16 Level 2 DWT Coefficients were encoded, the following 16 Level 2 DWT coefficients were encoded with executing the programme instructions at address 0x4E1 to 0x50E again.

There are 2 rows of Level 2 DWT coefficients, with each row having 32 DWT coefficients, that needed to be encoded. The programme instructions at address 0x4DE to 0x514 were executed again to process and encode the next row of DWT coefficients. With the DWT coefficients encoded, programme instructions at address 0x51A to 0x520 were repeatedly executed for 4 times to reset the modified data address (0x4F1, 0x4E7) back to its initial data address. These programme instructions needed to be executed such that the MISC could perform the subsequent part of the image data.



```
Addr.   Instructions
0x4BA   XOR     $t7     $t7     0
0x4BD   SBN     $s582   $t7     0
0x4C0   XOR     $t8     $t8     0
0x4C3   SBN     $t0     0x4C0   0
0x4C6   SBN     $t0     0x4C1   0
0x4C9   SBN     $t0     $t7     0x7F4
0x4CC   SBN     $s582   0x4C0   0
0x4CF   SBN     $s582   0x4C1   0
0x4D2   SBN     $s577   $t6     0
0x4D5   SBN     $s576   $t5     0
0x4D8   XOR     $t4     $t4     0
0x4DB   SBN     $s579   $t4     0
0x4DE   SBN     $s576   $t7     0
0x4E1   SBN     $s581   $t8     0
0x4E4   SBN     $s580   $t9     0
0x4E7   XOR     $s0     $t10    0
0x4EA   11TO8   $t10    $t10    0
0x4ED   GF      $s256   $t10    0
0x4F0   XOR     $t10    $t11    0
0x4F3   XOR     $t10    $t10    0
0x4F6   SBN     $t1     0x4E7   0
0x4F9   SBN     $t0     0x4ED   0
0x4FC   SBN     $t0     $t9     0x7E8
0x4FF   SBN     $t0     0x4F1   0
0x502   SBN     $s577   0x4E7   0
0x505   SBN     $t0     $t8     0x7DC
0x508   SBN     $s583   0x4ED   0
0x50B   SBN     $t6     0x4E7   0
0x50E   SBN     $t0     $t7     0x7D0
0x511   SBN     $t4     0x4E7   0
0x514   SBN     $t0     $t5     0x7C7
0x517   SBN     $s578   $t7     0
0x51A   SBN     $s581   0x4F1   0
0x51D   SBN     $s579   0x4E7   0
0x520   SBN     $t0     $t7     0x7F7
0x523   SBN     $one    $t7     0x2D9
```

Listing 3   CRS(20,16) Encoder Programme Instructions.

Based on Listing 3, the number of instructions executed for the written CRS Encoder programme were calculated that gives a total of 10,936 instructions. From Equation (51), it can be seen that there are parts of programme instructions that were repeated for a few times. Inside these programme instructions, there are certain parts of the instructions that were repeatedly executed. Since each programme instruction requires a total of 9 Clock Cycles, a complete CRS encoded DWT coefficients needs a total of 98,424 Clock Cycles. Equation (52) shows the calculated number of Clock Cycles needed by MISC to perform the CRS Encoding onto the DWT coefficients.

$$
\begin{aligned}
\text{CRS Encoding Instructions} &= 2\times\{2\times[20\times(16\times8+4)+4]+3\}\\
&\quad +2+(83\times4)+6+2+(4\times3)\\
&= 2\times\{2\times[20\times(132)+4]+3\}\\
&\quad +2+(332)+6+2+(12)\\
&= 2\times\{2\times[2644]+3\}+2+(332)+6+2+(12)\\
&= 2\times\{5291\}+2+(332)+6+2+(12)\\
&= 10{,}936 \text{ instructions executed}
\end{aligned}
\tag{51}
$$

$$
\begin{aligned}
\text{Total Clock Cycles} &= 10936 \text{ instructions executed} \times 9 \text{ clock cycles}\\
&= 98{,}424 \text{ clock cycles}
\end{aligned}
\tag{52}
$$

3.7.4 Clock Cycles of Complete DWT CRS MISC Programme

The calculated number of programme instructions executed for Level 1 DWT Filter and Level 2 DWT Filter were just an estimated values since it depends on number of times that the High Pass DWT coefficients values were divided by 2. By estimating the number of instructions to be executed, a total 31,910 lines of instructions were executed by the DWT CRS MISC in order to perform compression and encoding onto the image (64 pixels x 4 pixels). As shown in Table 6, it takes an estimated 28,7190 Clock Cycles to process and encode a part of the image (64 pixels x 4 pixels). Based on the calculation in Equation (53), a total of 4,595,040 Clock Cycles is required for the MISC to completely process the complete image (64 pixels x 64 pixels) captured by a CMOS camera. Although the FPGA input frequency is at 48MHz, the DWT CRS MISC architecture is set to operate at the frequency of 24MHz. The MISC is set to a lower operating frequency is because of the maximum time delay (32.732ns) required for the data to become stable in between the connected components (eg. registers, multiplexers, functional blocks, Memory etc.) in the MISC architecture.

With estimated number of Clock Cycles, the total amount of time required by the MISC to process a complete image (64 pixels x 64 pixels) can be calculated. As shown in Equation (54), the MISC requires 0.1915s to completely process the image with the size of 64 pixels x 64 pixels. At this speed, the estimated number of images can be processed per second by the MISC architecture are 3 frames of images (3Hz).

Table 6 Number of Programme Instructions executed for DWT CRS MISC.

Programme	No. Instructions	Memory (bits)	Memory (Bytes)	Instructions Cycles	Clock Cycles
Level 1 DWT Filter	90	3,240	405	16,739	150,651
- *Row Filtering*	*44*	*1,584*	*198*	*8,368*	*75,312*
- *Column Filtering*	*46*	*1,656*	*207*	*8,372*	*75,348*
Level 2 DWT Filter	84	3,024	378	4,235	38,115
- *Row Filtering*	*50*	*1,800*	*255*	*2,137*	*19,233*
- *Column Filtering*	*34*	*1,224*	*153*	*2,098*	*18,882*
CRS Coding Scheme	36	1,296	162	10,936	98,424
Total	**210**	**7,560**	**945**	**31,910**	**287,190**

$$\begin{aligned}\text{Total Clock Cycles} &= 287{,}190 \text{ instructions executed} \times 16 \\ &= 4{,}595{,}040 \text{ clock cycles}\end{aligned} \tag{53}$$

$$\begin{aligned}\text{Time} &= \frac{4{,}595{,}040 \text{ clock cycles}}{24{,}000{,}000 \text{ Hz}} \\ &= 0.1915\, s\end{aligned} \tag{54}$$

3.8 SUMMARY

In Chapter 3, the methodology on developing the DWT CRS MISC architecture was presented. By designing the DWT CRS MISC architecture, a complete circuitry that interconnects the registers, functional blocks, MUXs and Memory was described by using the hardware descriptions language, VHDL. The functionality of the functional blocks were also described in separate VHDL files and then combined these files together at the architecture level. As such, the described functional blocks can be reused in other area of hardware implementations, for example the GF block and XOR block can be used to describe a RS LFSR encoder.

Subsequently, the developed DWT CRS MISC architecture requires to be programmed such that it will operate as an image processing framework for use in the WVSNs. Without programming the DWT CRS MISC, it would not operate and perform any image processing onto the image data available in its Memory. As a result, programme instructions were written based on the algorithms that are described in Section 3.5. Then the DWT CRS MISC was programmed using the written programme instructions which are described in Section 3.6. The DWT CRS MISC architecture operating at 24MHz, would require 287,190 Clock Cycles (11.966ms) to completely execute the written programme instructions. However, the DWT CRS MISC only processed 4 rows of the image data (complete image size is 64 pixels x 64 pixels). As a result, the DWT CRS MISC continued to process the subsequent 4 rows of image data until complete 64 rows of image was processed. This would require 4,595,040 Clock Cycles (0.1915s) to have the whole image processed.

Once the DWT CRS MISC architecture was described, simulations were performed onto the combinational circuit that generates the control signals to verify its operations. Besides, simulations were also performed onto the MISC architecture for executing different programme instructions. This is to ensure that the data read from the Memory was flowing correctly in the MISC architecture and the processed data was written back correctly into the Memory. With considering the hardware delay, the Post & Route simulations were performed onto the DWT CRS MISC architecture. From these simulations, the longest delay occurred can then be determined such that correct operating frequency of the DWT CRS MISC can be set. Further information on the generated simulation waveforms for the DWT CRS MISC architecture are shown in Chapter 4.

CHAPTER 4

RESULTS AND DISCUSSIONS

In Chapter 4, the results of the developed DWT CRS MISC architecture are presented and discussed. The Behavioral and Post & Route simulation were performed and the waveforms for the control signals in the DWT CRS MISC architecture are presented in Section 4.1. Next, Section 4.2 presents the Behavioral and Post & Route simulation waveforms on the operation of the four programme instructions. After the simulations were performed, the amount of hardware utilisations of the existing techniques and the FPGA synthesised DWT CRS MISC architecture were compared, which is discussed in Section 4.3. Meanwhile, the effect on the amount of DWT coefficients to be transmitted across the WVSNs were studied and discussed in Section 4.4. By determining the quality of reconstructed image, Section 4.5 shows the effect of errors that occurred onto the DWT coefficients. Lastly, Section 4.6 presents a study on the security level, in terms of number possible trials, for using different CRS coding scheme in encrypting (encoding) the image data.

4.1 CONTROL SIGNALS WAVEFORMS

The control signals waveforms that were produced by simulating the combinational logic circuit are shown in this section. This includes both the Behavioral waveforms and the Post & Route waveforms. The Behvioral simulation only simulated the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) described circuit and generated the circuit’s output waveforms without considering the hardware delays that occurred in the actual hardware implementation. Whereas, the Post & Route simulation simulated the VHDL described circuit with considering the hardware delays. These waveforms were generated by using the Xilinx ISim Simulator 11.5 software.

4.1.1 Control Signals: Behavioral Simulation Waveforms

The combinational logic circuit that produced the control signals for DWT CRS MISC architecture was described in Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). Simulation was performed onto the VHDL described control signals combinational logic circuit in the Xilinx ISE 11.5 environment. In Figure 120, the Behavioral waveforms were generated based on the control signals combinational logic circuits shown in Section 3.2. The Behavioral waveforms were compared with the control signals Truth Table, which is shown in Table 4. For each Clock Cycle, it can be seen that the control signals output for both the Behavioral waveforms and the Truth Table were the same. Therefore, the control signals generated from the combinational logic circuit (based on the Boolean Logic equations) matched the required control signals shown in the Truth Table. Without considering hardware delays, this verified the output of control signals generated at a particular Clock Cycles in an ideal hardware implementation environment.

4.1.2 Control Signals: Post and Route Simulation Waveforms

In actual FPGA implementation, the designed combinational circuit may encounter hardware delays (eg. setup delay, hold delay etc.) that affect the operation frequency of the MISC architecture. By considering the hardware delays, the Post & Route simulation was performed to determine the control signals generated in an actual hardware implementation of the combinational circuits. For the Post & Route simulation, the waveforms of the control signals generated are shown in Figure 121 and Figure 122. In these waveforms, it can seen that a 4-bit Counter (*iCOUNT[3:0]*) increased by 1 during every rising edge of input Clock (*tb_CLK*) with a delay of 2.833ns. The MAR_SEL control signal was considered to be the maximum delay since the MAR_SEL control signal had the longest delay for the expected Logic state to become stable as compared to other control signals. For the MAR_SEL control signal to reach the stable Logic state, an additional delay of 9.973ns was required after the 4-bit Counter value had been stabilized. For these control signals to become stable, the total maximum delay of 12.806ns from the rising edge of input Clock (*tb_CLK*) was required.

In Figure 121 and Figure 122, there are slight differences between these two waveforms. Figure 121 shows that the N (*tb_N*) signal did not come to the Logic state HIGH (Logic 1) between the Clock Cycle 6 to Clock Cycle 8. The N signal was determined by the result obtained from the SBN instruction. When the result is negative, then it would generate a HIGH (Logic 1) state to the N register. Then the N register would output its value as N input signal to the combinational circuit. The N signal affected the MISC architecture whether to add the 'Target Address' to the current Program Counter (PC) value. If negative result was obtained from the SBN instruction, then the 'Target Address' was added to the current PC value and overwriting the PC register. Otherwise, the 'Target Address' would not be added to the current PC value when positive result was obtained from the SBN instruction. Figure 122 shows that the PC_WRITE (*tb_PC_WRITE*) signal became HIGH (Logic 1) state when N signal was at HIGH (Logic 1) state during the Clock Cycle 7. In the meantime, the delay of PC_WRITE signal had became longer (12.730ns) and this was caused by the additional N input signal that determined the PC_WRITE control signal at Clock Cycle 7. Table 7 and Table 8 listed out the corresponding time delays required by the control signals to become stable after each rising edge of the input Clock. The time delays required to increase the *iCount4* counter value and become stable are also shown in Table 7 and Table 8.

Note that Table 7 shows the corresponding time delays of the control signals for the case where the input signal, N = 0. Whereas, Table 8 shows the respective time delays of the control signals for the case where the input signal, N = 1. For the case when N = 1, the MISC architecture was required to branch off to another programme address location. Therefore, the MISC would not execute the next in line programme instruction. Instead, the MISC would execute the programme instruction that was located at the targeted (jump to) programme address location.

From the time delays listed in Table 7 and Table 8, the longest time delay occurred among all the control signals was determined to be 12.730ns. Therefore, the operating frequency of the actual FPGA implementation at 48MHz (with period of 20.833ns) is divided by two such that a lower operating frequency 24MHz (period

41.667ns) can be used to generate these control signals. This is to have the operating frequency of the DWT CRS MISC to meet the time delays that occurred on the combinational logic circuit that generates the control signals into the DWT CRS MISC architecture. Besides considering the time delays of control signals combinational logic circuit, the time delays of the DWT CRS MISC architecture also played an important role in determining the operating frequency of the MISC architecture. Therefore, the following Section 4.2 presents the Behavioral and Post & Route simulations that were performed onto the DWT CRS MISC architecture, which had the hardware time delays taken into account for.

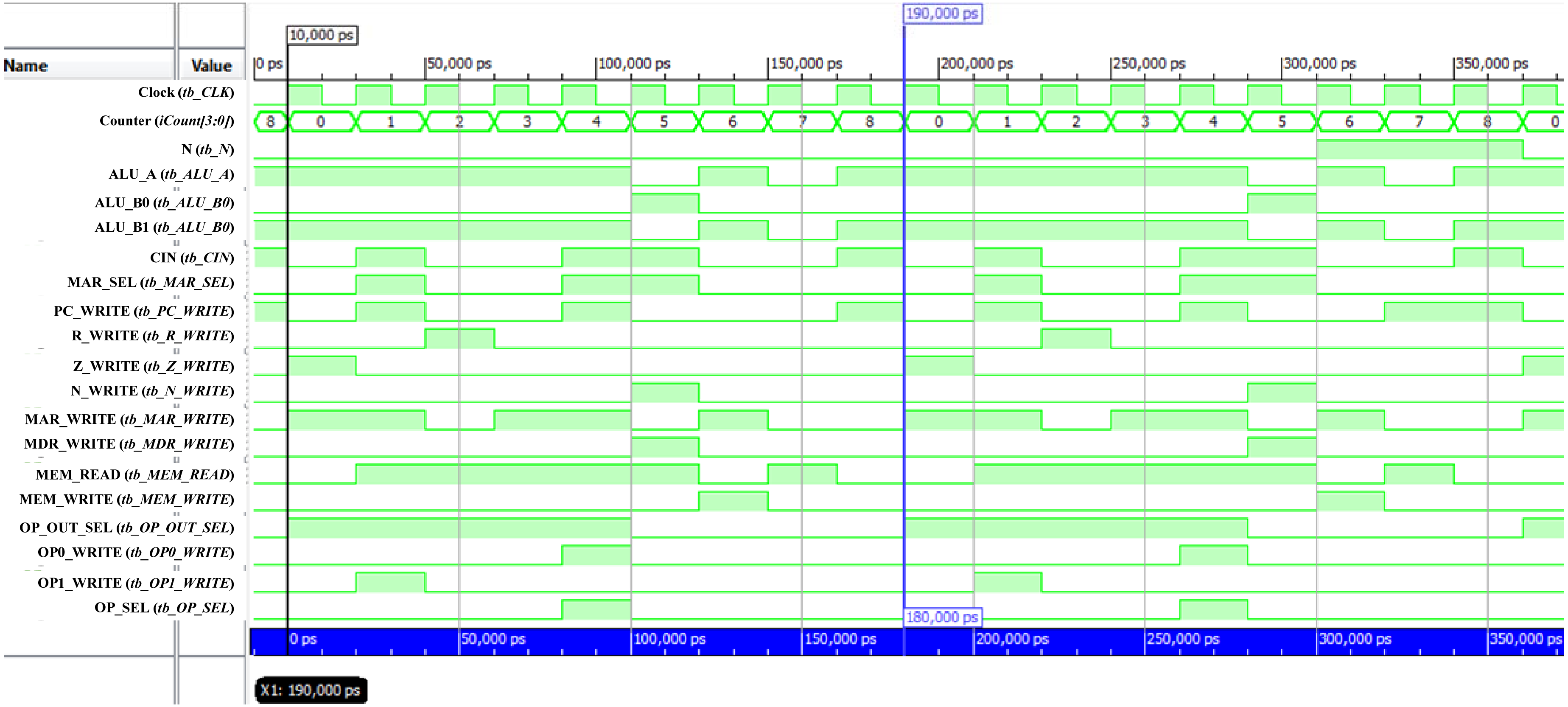


Figure 120   DWT CRS MISC Control Signals Behavioral Waveforms.

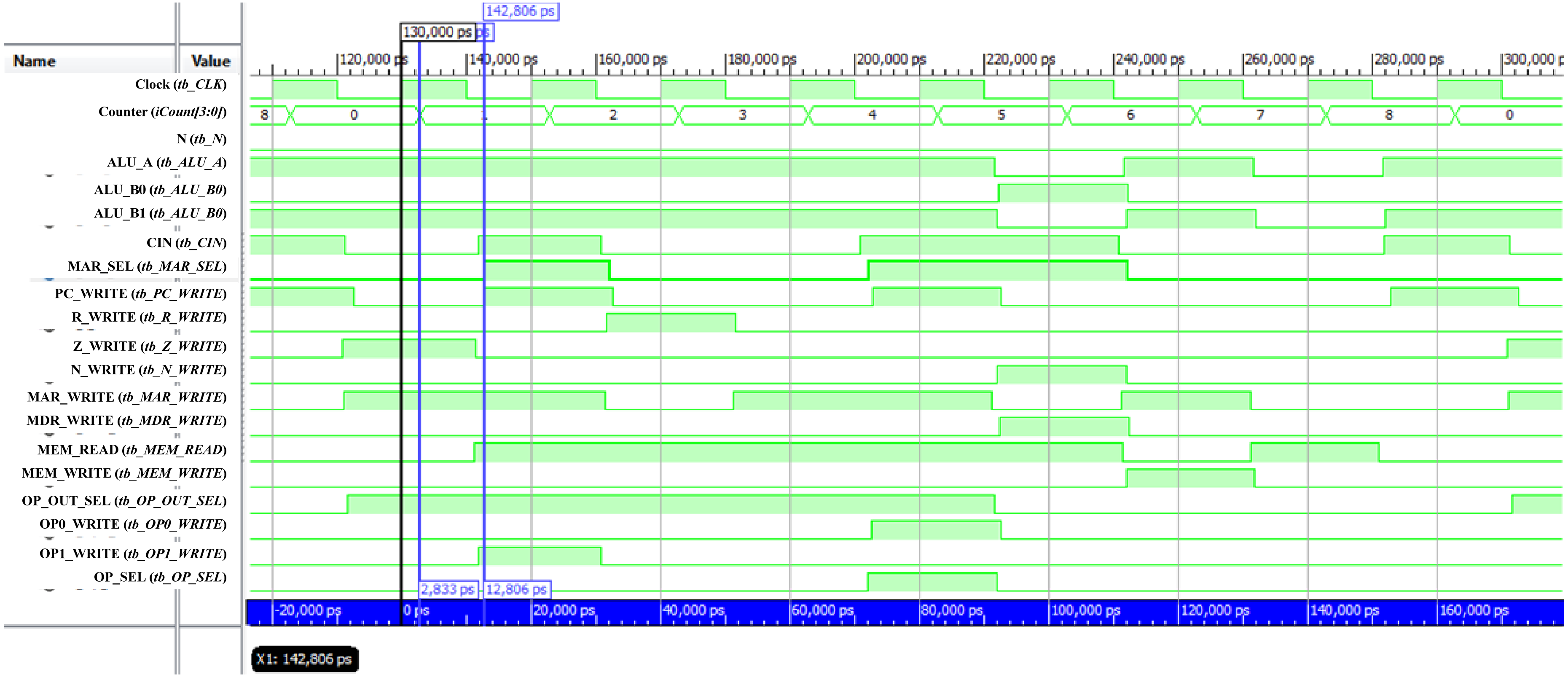


Figure 121   DWT CRS MISC Control Signals Post & Route Waveforms with N = 0.

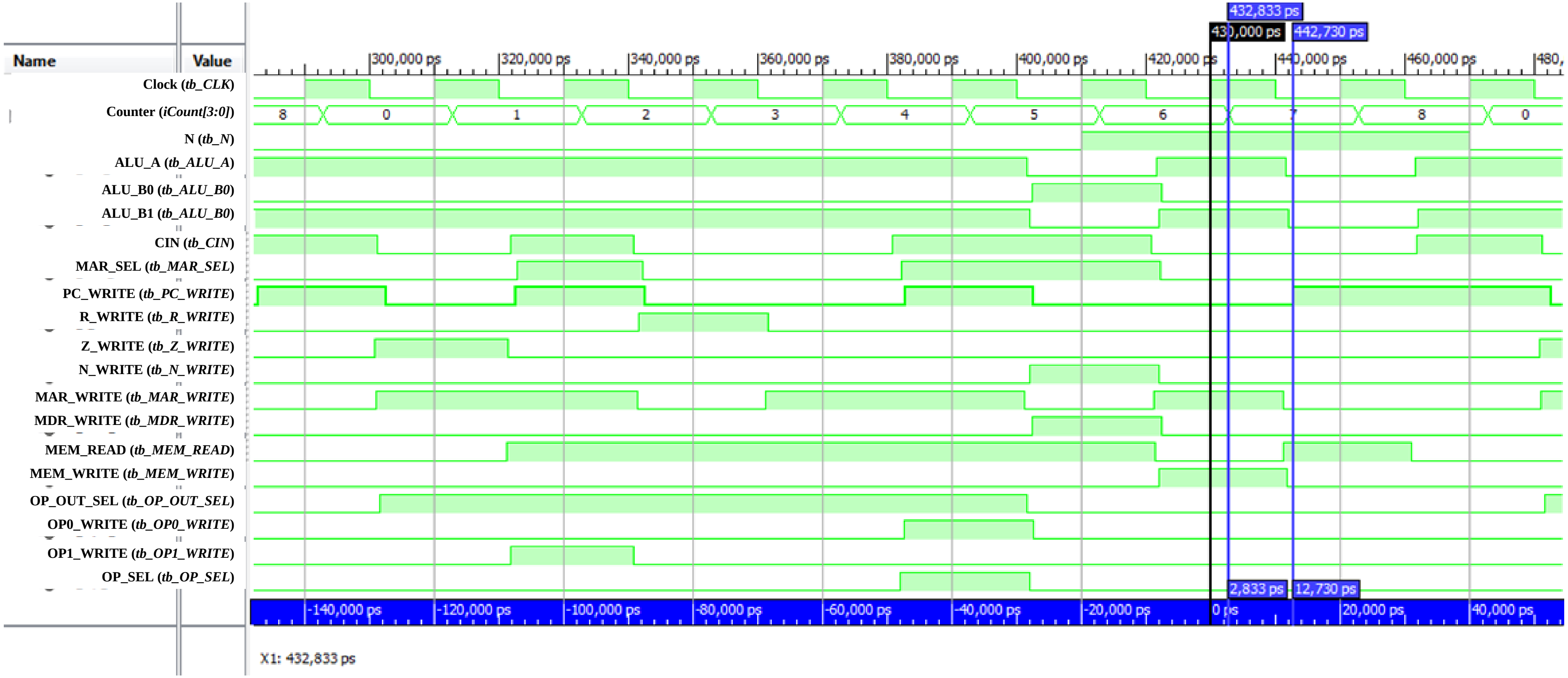


Figure 122 DWT CRS MISC Control Signals Post & Route Waveforms with N = 1.

Table 7 Time delays of control signals generated for case input signal, N = 0.

Clk	Delay (ns)																	
	iCount4	**ALU_A**	**ALU_B0**	**ALU_B1**	**CIN**	**MAR_SEL**	**PC_WRITE**	**R_WRITE**	**Z_WRITE**	**N_WRITE**	**MAR_WRITE**	**MDR_WRITE**	**MEM_READ**	**MEM_WRITE**	**OP_OUT_SEL**	**OP0_WRITE**	**OP1_WRITE**	**OP_SEL**
0	2.833	-	-	-	11.208	-	12.511	-	10.816	-	10.972	-	-	-	11.489	-	-	-
1	2.833	-	-	-	11.716	12.806	12.614	-	11.446	-	-	-	11.241	-	-	-	11.822	-
2	2.833	-	-	-	10.798	12.152	12.614	11.643	-	-	11.389	-	-	-	-	-	10.819	-
3	2.833	-	-	-	-	-	-	11.643	-	-	11.256	-	-	-	-	-	-	-
4	2.833	-	-	-	10.798	12.152	12.725	-	-	-	-	-	-	-	-	12.576	-	11.893
5	2.833	11.652	12.273	11.996	-	-	12.614	-	-	11.932	11.256	12.300	-	-	11.572	12.576	-	11.893
6	2.833	11.652	12.273	11.996	10.798	12.152	-	-	-	11.932	11.256	12.300	11.374	11.933	-	-	-	-
7	2.833	11.652	-	11.996	-	-	-	-	-	-	11.256	-	11.241	11.715	-	-	-	-
8	2.833	11.665	-	12.009	11.716	-	12.725	-	-	-	-	-	10.957	-	-	-	-	-

Table 8 Time delays of control signals generated for case input signal, N = 1.

Clk	Delay (ns)																	
	iCount4	**ALU_A**	**ALU_B0**	**ALU_B1**	**CIN**	**MAR_SEL**	**PC_WRITE**	**R_WRITE**	**Z_WRITE**	**N_WRITE**	**MAR_WRITE**	**MDR_WRITE**	**MEM_READ**	**MEM_WRITE**	**OP_OUT_SEL**	**OP0_WRITE**	**OP1_WRITE**	**OP_SEL**
0	2.833	-	-	-	11.208	-	12.511	-	10.816	-	10.972	-	-	-	11.489	-	-	-
1	2.833	-	-	-	11.716	12.806	12.614	-	11.446	-	-	-	11.241	-	-	-	11.822	-
2	2.833	-	-	-	10.798	12.152	12.614	11.643	-	-	11.389	-	-	-	-	-	10.819	-
3	2.833	-	-	-	-	-	-	11.643	-	-	11.256	-	-	-	-	-	-	-
4	2.833	-	-	-	10.798	12.152	12.725	-	-	-	-	-	-	-	-	12.576	-	11.893
5	2.833	11.652	12.273	11.996	-	-	12.614	-	-	11.932	11.256	12.300	-	-	11.572	12.576	-	11.893
6	2.833	11.652	12.273	11.996	10.798	12.152	-	-	-	11.932	11.256	12.300	11.374	11.933	-	-	-	-
7	2.833	11.652	-	11.996	-	-	12.730	-	-	-	11.256	-	11.241	11.715	-	-	-	-
8	2.833	11.665	-	12.009	11.716	-	-	-	-	-	-	-	10.957	-	-	-	-	-

4.2 PROGRAMME INSTRUCTIONS WAVEFORMS

Simulations were performed onto the four programme instructions designed for the DWT CRS MISC. For Section 4.2.1, the Behavioral simulation waveforms for each programme instructions were shown. Next, the Post & Route simulations were performed onto these programme instructions and the results are shown in Section 4.2.2.

4.2.1 Programme Instructions: Behavioral Simulation Waveforms

Before implementing the proposed DWT CRS MISC architecture in an actual FPGA, Behavioral simulation was performed in order to verify correct functionality of MISC architecture. Therefore, the DWT CRS MISC architecture was described using the VHDL hardware description language and the Behavioral simulation can then be performed on the described MISC architecture. The Behavioral simulation was performed using the simulation software integrated in the Xilinx ISE 11.5.

By referring to the waveforms in Figure 123, the SBN instruction required up to a total 9 Clock Cycles (*iclock2*) to completely execute the instruction. The signal *iclock2* is an important clock signal input to the control signals combinational logic circuit. As mentioned in Section 3.2, the control signals combinational logic circuit generated the control signals that were input into the MISC architecture. These control signals controlled the registers when to write the data and also controlled the data when to read or write into the Memory. Note that the signal *iclock2* was produced by dividing the main input clock signal (*tb_clk*) into half. The main input clock signal (*tb_clk*) was the external clock signal input into the FPGA. The reason of dividing the signal *iclock2* by half was the Block RAM used by the DWT CRS MISC architecture requires a significant amount of time delay in order to correctly read or write the data into the Block RAM. The details on the time delay of the Block RAM will be further discussed in Section 4.2.2. In Figure 123, an example of the simulation waveforms showing the data flow in and out of the register/Memory while the MISC executed the SBN instruction. In the simulation waveforms, there are 3 programme addresses (0x2F8, 0x2F9, 0x2FA) that stored the two data addresses (0x001, 0x259) and the target address (0x7F7) at the Memory respectively. During Clock Cycle 0 and

3, the Program Counter (PC) register consisted the programme addresses (0x2F8, 0x2F9) that would be stored into the Memory Address Register (MAR) to provide memory addresses when reading the data addresses from the Memory. The two data addresses (0x001, 0x259) were read from the Memory at Clock Cycle 1 and 4 respectively. Then these two data addresses (0x001, 0x259) were used to read Operand *A* and *B* from the Memory during Clock Cycle 2 and 5, respectively. In Clock Cycle 5, the Operand *B* (0x000) was subtracted with Operand *A* (0x001) that gave the final result (0x7FF) in negative value. The final result (0x7FF) from the arithmetic subtraction was stored into the Memory Data Register (MDR). Throughout Clock Cycle 6, the final result (0x7FF) stored in the MDR, would be written into the Memory at the Operand *B* memory location (0x259). At Clock Cycle 7, the target address (0x7F7, 3rd line of the programme code) was added to the current PC value 0x2F9 to give a jump to address 0x2F1. Before the next programme instruction was executed, the PC value was increased by 1 to become 0x2F2 during Clock Cycle 8. This completed the execution of SBN instruction inside the MISC architecture.

For the GF MULT programme instruction, Galois Field GF(2^8) arithmetic multiplication was performed on two Operands (data) that are read from the Memory (*B* = *B* x *A*). Then the result from the GF(2^8) multiplication was written back to Operand *B* (second read data location). The operation of the GF MULT instruction was similar to the SBN instruction. The only difference was that GF multiplication was performed onto the two Operands, *A* and *B* instead of arithmetic subtraction. From the example simulation waveforms shown in Figure 124, two data (address 0x103 and 0x25F) were read from the Memory at Clock Cycle 2 and 5, where each of its value was 0x053 and 0x0F8, respectively. Once the second data was read at Clock Cycle 5, the result of the Galois Field multiplication (*igf[10:0]*) was obtained (0x0C2) and stored the result back to the second data location (0x25F) at Clock Cycle 6. The complete operation of GF MULT instruction performed by the MISC architecture can be observed from the waveforms in Figure 124.

Next, the XOR programme instruction commands the MISC to perform 11-bit XOR operation onto two Operands (data) read from the Memory (*B* = *B* XOR *A*). Again the result of the XOR operation was written to Operand *B* (second read data location). By referring to the example in Figure 125, two data (address 0x004 and 0x25F) were read from the Memory during Clock Cycle 2 and 5, with the value 0x7F8 and 0x000 respectively. Once the second data was read, the result of the XOR (*ixor[10:0]*) was obtained (0x7F8) at Clock Cycle 5 and written back to the Operand *B* memory location (0x25F) at Clock Cycle 6. The XOR instruction was required to perform the GF arithmetic addition when encoding the data using the CRS coding scheme. A complete execution of the XOR instruction by the MISC architecture was shown in the simulation waveforms in Figure 125.

The fourth programme instruction, 11TO8 instruction was required for converting the negative DWT coefficients from 10-bit signed data (11-bit) to 7-bit signed data (8-bit). This 11TO8 instruction only read the operand *B* (second read data) from the Memory. Then the result obtained was the combined of MSB (sign bit, bit 10) of operand *B* with the 7 LSB of the operand *B*. For example shown in Figure 126, Operand *B* (address 0x25F) of value 0x7F8 was read at Clock Cycle 5 and the output (value 0x0F8) of this instruction (*i11TO8[10:0]*) was written back to Operand *B* memory location again (0x25F) at Clock Cycle 6.

Based on the waveforms for the programme instructions, it is shown that these programme instructions were able to process the data (read from the Memory) correctly. Afterwards, the processed data obtained from the respective programme instructions were then correctly written back to the Memory. Therefore, the operation of these four programme instructions in the DWT CRS MISC architecture were verified. With these four programme instructions, a complete DWT filter and CRS(20,16) encoder programme were written. The written programme is then programmed into the DWT CRS MISC such that it compresses and encodes the image data in a single MISC architecture.

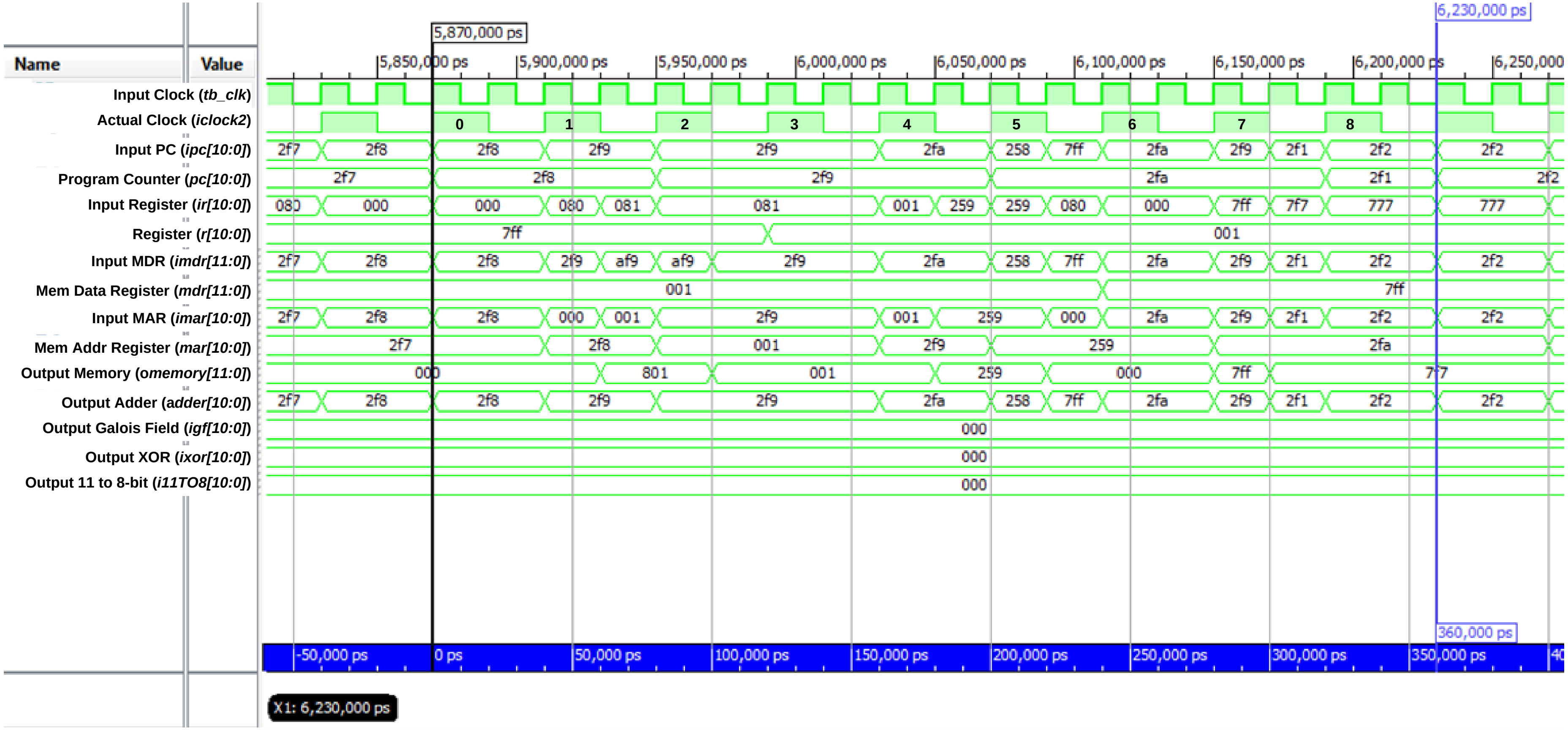

Name
Value
5,870,000 ps
6,230,000 ps
5,850,000 ps
5,900,000 ps
5,950,000 ps
6,000,000 ps
6,050,000 ps
6,100,000 ps
6,150,000 ps
6,200,000 ps
6,250,000
Input Clock (tb_clk)
Actual Clock (iclock2)
0
1
2
3
4
5
6
7
8
Input PC (ipc[10:0])
2f7 2f8 2f8 2f9 2f9 2fa 258 7ff 2fa 2f9 2f1 2f2 2f2
Program Counter (pc[10:0])
2f7 2f8 2f9 2fa 2f1 2f2
Input Register (ir[10:0])
080 000 000 080 081 081 001 259 259 080 000 7ff 7f7 777 777
Register (r[10:0])
7ff 001
Input MDR (imdr[11:0])
2f7 2f8 2f8 2f9 af9 af9 2f9 2fa 258 7ff 2fa 2f9 2f1 2f2 2f2
Mem Data Register (mdr[11:0])
001 7ff
Input MAR (imar[10:0])
2f7 2f8 2f8 000 001 2f9 001 259 000 2fa 2f9 2f1 2f2 2f2
Mem Addr Register (mar[10:0])
2f7 2f8 001 2f9 259 2fa
Output Memory (omemory[11:0])
000 801 001 259 000 7ff 7f7
Output Adder (adder[10:0])
2f7 2f8 2f8 2f9 2f9 2fa 258 7ff 2fa 2f9 2f1 2f2 2f2
Output Galois Field (igf[10:0])
000
Output XOR (ixor[10:0])
000
Output 11 to 8-bit (i11TO8[10:0])
000
360,000 ps
-50,000 ps
0 ps
50,000 ps
100,000 ps
150,000 ps
200,000 ps
250,000 ps
300,000 ps
350,000 ps
X1: 6,230,000 ps

Figure 123 Behavioral Simulation Waveforms SBN Instruction for DWT CRS MISC.

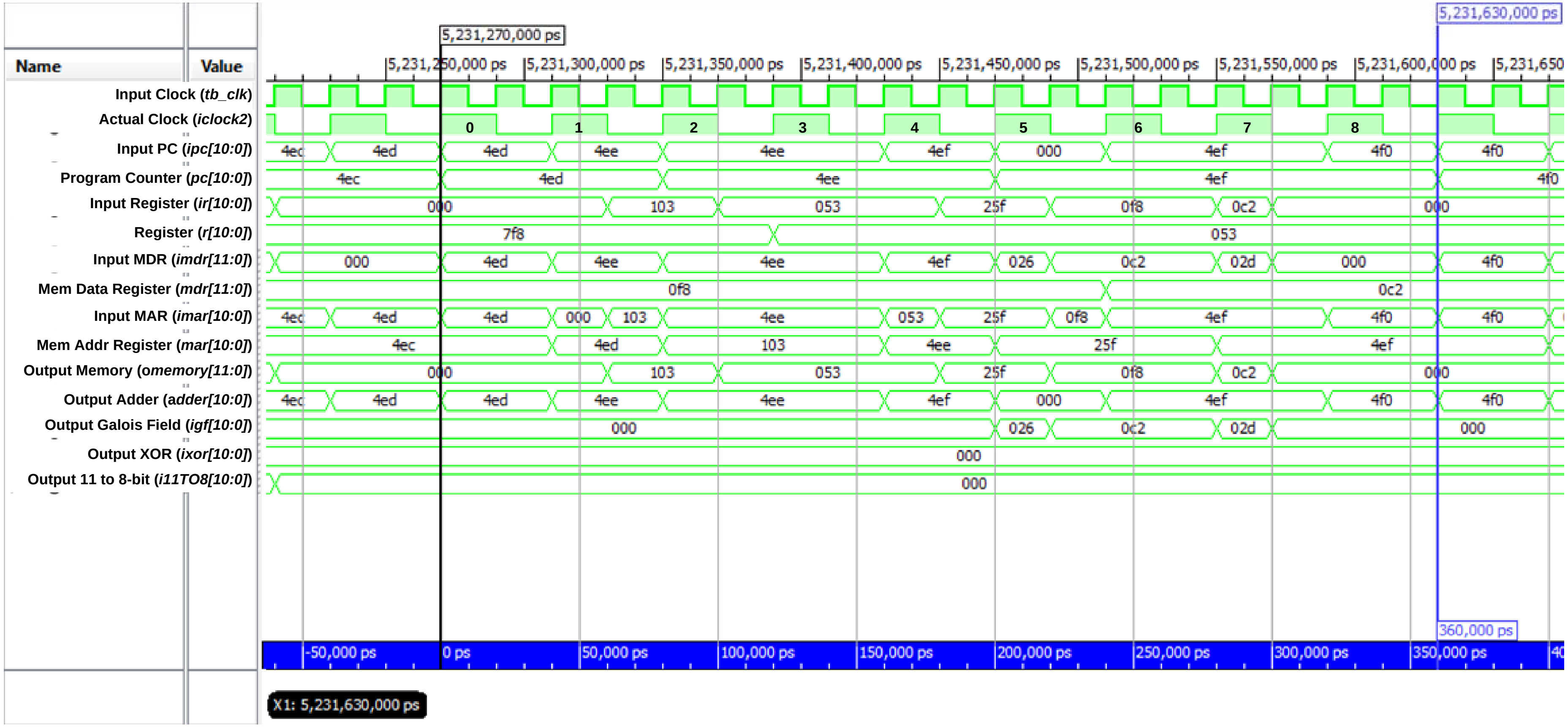


Figure 124 Behavioral Simulation Waveforms GF MULT Instruction for DWT CRS MISC.

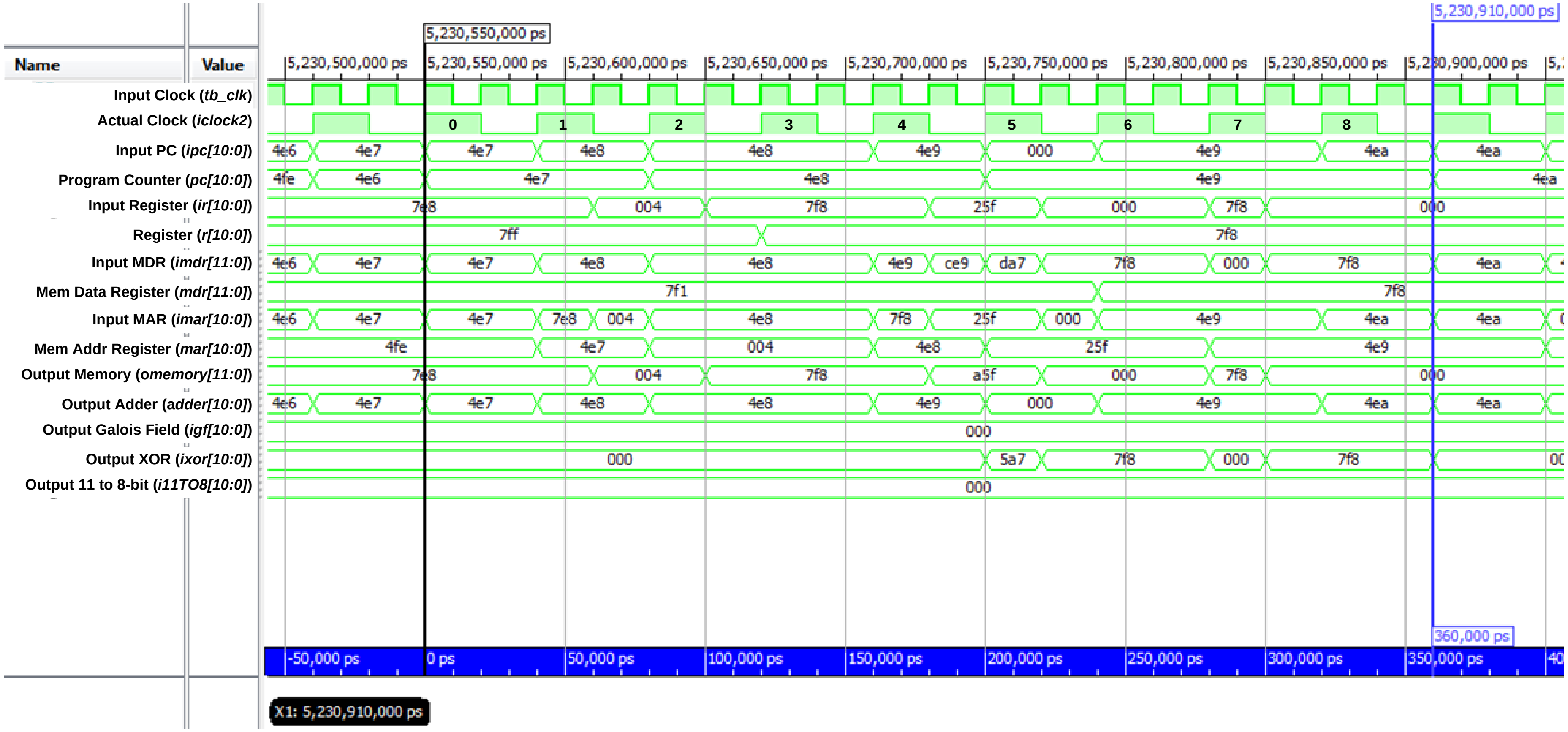


Figure 125 Behavioral Simulation Waveforms XOR Instruction for DWT CRS MISC.

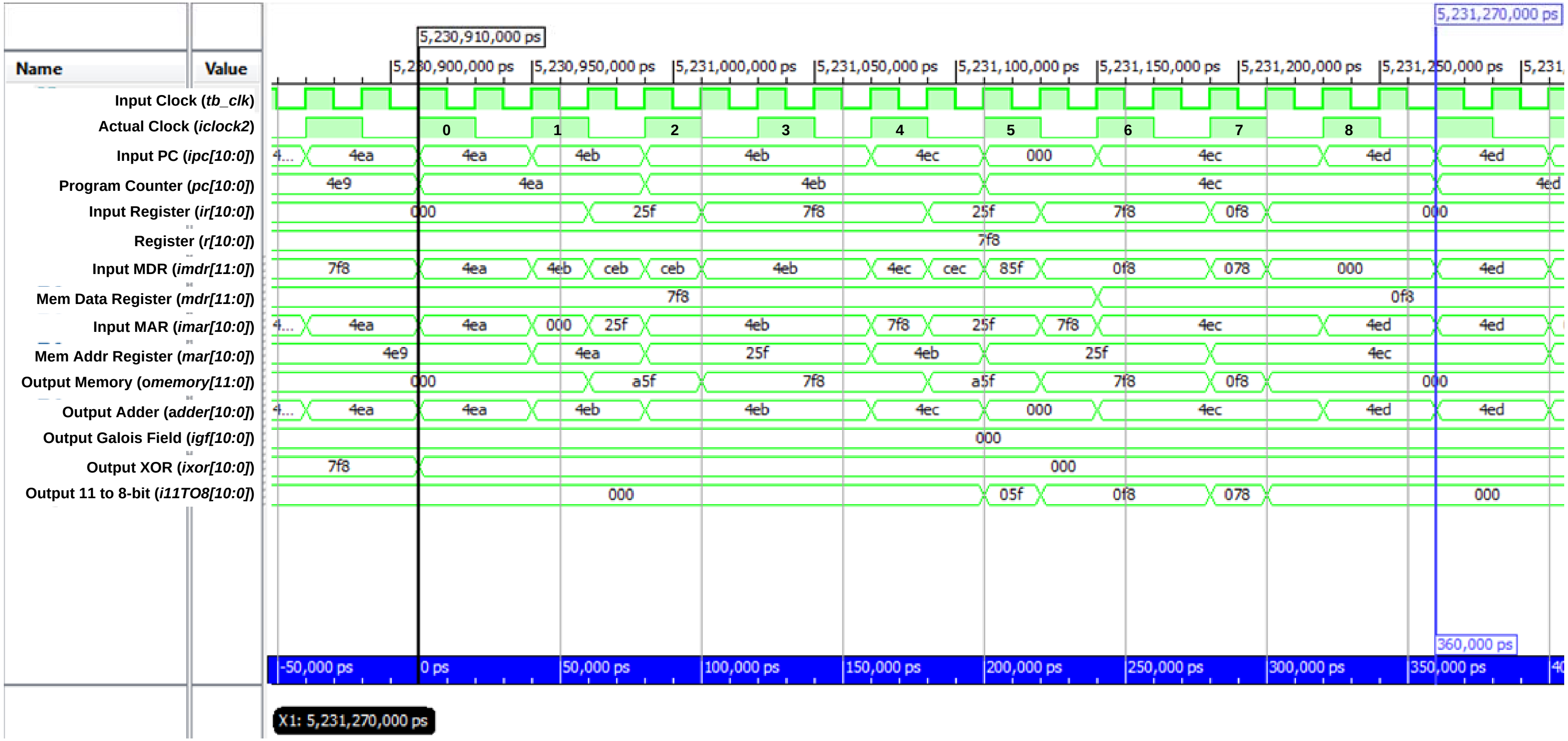


Figure 126 Behavioral Simulation Waveforms 11TO8 Instruction for DWT CRS MISC.

4.2.2 Programme Instructions: Post and Route Simulation Waveforms

The Post & Route simulations were performed onto the DWT CRS MISC architecture to verify the four programme instructions operate according to the requirements (mentioned in Section 3.1) in an actual hardware (FPGA) implementation. Besides, these simulations were performed to determine the maximum time delay required by the outputs to become stable. By determining the longest delay, the correct operating frequency of the DWT CRS MISC architecture can be determined such that correct operation of the MISC architecture is achieved.

In Figure 127, the Post and Route simulation waveforms for SBN instruction show that the longest time delay of 32.095ns occurs on the Adder Output (*adder[10:0]*) at Clock Cycle 5. At this particular Clock Cycle 5, the Adder Output requires 32.095ns of time delay to have the correct output data to become stable. Then the next longest delay (30.429ns) is also the Adder Output to be stabled at Clock Cycle 7. Other time delays for the proposed DWT CRS MISC architecture are recorded in the Table 9. From Table 9, the data output from Memory (*omemory[11:0]*) requires a longest amount of time delay of 24.549ns to become stable. This was the longest time delay encountered in the operation of SBN instruction.

Table 9 DWT CRS MISC Architecture SBN Instruction delays.

Clock	**Delays (ns)**								
	iClock2	***PC_reg***	***MAR_reg***	***OMemory***	***Adder***	***R_reg***	***MDR_reg***	***iGF***	***iXOR***
0	6.283	7.811	-	-	-	-	-	-	-
1	6.283	-	8.036	24.549	11.820	-	-	-	-
2	6.283	7.811	8.062	24.531	11.839	-	-	-	-
3	6.283	-	-	-	-	8.097	-	-	-
4	6.283	-	8.062	24.549	12.464	-	-	-	-
5	6.283	7.811	8.062	24.549	32.095	-	-	-	-
6	6.283	-	-	-	-	-	8.027	-	-
7	6.283	-	8.062	24.549	30.429	-	-	-	-
8	6.283	7.811	-	-	14.108	-	-	-	-

Next, the Post & Route simulation was also performed onto the GF MULT instruction with its corresponding waveforms shown in Figure 128. From these waveforms, the longest time delay encountered was 32.732ns for the output of the GF MULT functional block (*igf[10:0]*) to become stable during Clock Cycle 5. The subsequent longest time delay was the output data from the Memory (*omemory[11:0]*) to be stabled for different Clock Cycles. By referring to Table 10, other time delays were also recorded, such as time delay required for writing correct programme addresses to Programme Counter register (*PC_reg*), writing correct memory addresses to Memory Address Register (*MAR_reg*), writing correct data to Read register (*R_reg*), and writing correct output data from the functional block to the Memory Data Register (*MDR_reg*). Therefore, the longest time delay, encountered in the operation of GF MULT instruction, would be considered to be one of the affecting factors for the operating frequency of DWT CRS MISC architecture.

Table 10 DWT CRS MISC Architecture GF MULT Instruction delays.

Clock	**Delays (ns)**								
	iClock2	***PC_reg***	***MAR_reg***	***OMemory***	***Adder***	***R_reg***	***MDR_reg***	***iGF***	***iXOR***
0	6.283	7.811	-	-	-	-	-	-	-
1	6.283	-	7.967	24.549	11.839	-	-	-	-
2	6.283	7.811	8.062	24.549	-	-	-	-	-
3	6.283	-	-	-	-	8.097	-	-	-
4	6.283	-	8.062	24.549	12.445	-	-	-	-
5	6.283	7.811	8.062	24.549	-	-	-	32.732	-
6	6.283	-	-	-	-	-	8.027	-	-
7	6.283	-	8.062	24.549	-	-	-	-	-
8	6.283	-	-	-	13.509	-	-	-	-

Table 11 DWT CRS MISC Architecture XOR Instruction delays.

Clock	**Delays (ns)**								
	iClock2	***PC_reg***	***MAR_reg***	***OMemory***	***Adder***	***R_reg***	***MDR_reg***	***iGF***	***iXOR***
0	6.283	7.811	-	-	-	-	-	-	-
1	6.283	-	7.967	24.549	12.184	-	-	-	-
2	6.283	7.811	8.062	24.549	-	-	-	-	-
3	6.283	-	-	-	-	8.097	-	-	-
4	6.283		8.062	24.549	12.445	-	-	-	-
5	6.283	7.811	8.062	24.549	-	-	-	-	27.301
6	6.283	-	-	-	-	-	8.024	-	-
7	6.283	-	8.062	24.549	-	-	-	-	-
8	6.283	-	-	-	13.509	-	-	-	-

For the XOR instruction, the longest time delay occurred at Clock Cycle 5 at the output of XOR functional block to become stable for 27.301ns, as shown in Figure 129. The subsequent longest time delay (24.549ns) is the time delay required for the data output from the Memory (*omemory[11:0]*) to be stabled for different Clock

Cycles. Table 11 shows the other time delays that were required by the XOR instruction to operate and process the data correctly. The longest time delay encountered in the operation of XOR instruction was not the main affecting factor on the operating frequency of the DWT CRS MISC architecture.

Table 12 DWT CRS MISC Architecture 11TO8 Instruction delays.

Clock	**Delays (ns)**								
	iClock2	***PC_reg***	***MAR_reg***	***OMemory***	***Adder***	***R_reg***	***MDR_reg***	***iGF***	***iXOR***
0	6.283	7.811	-	-	-	-	-	-	-
1	6.283	-	7.967	24.549	11.82	-	-	-	-
2	6.283	7.811	8.062	24.549	-	-	-	-	-
3	6.283	-	-	-	-	-	-	-	-
4	6.283	-	8.062	24.549	12.79	-	-	-	-
5	6.283	7.811	8.062	24.549	-	-	-	-	-
6	6.283	-	-	-	-	-	7.958	-	-
7	6.283	-	8.062	24.549	-	-	-	-	-
8	6.283	-	-	-	13.509	-	-	-	-

As shown in Figure 130, the 11TO8 instruction had the longest time delay of 24.549ns for the output data from Memory (*omemory[11:0]*) to become stable. By referring to Table 12, the subsequence longest time delay (13.509ns) was the output from ADDER to be stabled at Clock Cycle 8. It can be seen that the output from the ADDER required significant amount of time delays to become stable. The output from the 11TO8 functional block was not able to determine in the simulator. The reason to this is the 11TO8 functional block was made of wire lines that were combined together to form an output. Consequently, it was assumed that the delay of these wire lines were less significant as compared to the time delays required to store data into the registers. The Memory Data Register (*MDR_reg*) required 7.958ns of time delay to store the output from the 11TO8 functional block.

By performing Post & Route simulations, the actual operations of these four programme instructions available in the DWT CRS MISC architecture could be verified. This included the time delays required for components in the MISC architecture to correctly functions as designed. The longest time delay encounter by the MISC architecture is the time delay (32.732ns) required by the output of GF MULT functional block to become stable. With the time delay known, this determined the maximum operating frequency of the MISC architecture described into the Xilinx Spartan-3L FPGA. In order to meet these time delays requirements, the operating frequency of the DWT CRS MISC architecture was set lower than the maximum allowable frequency.

As a result, the proposed MISC architecture was set to operate at 24MHz (clock period of 41.667ns) instead of using the actual input clock frequency at 48MHz (clock period of 20.833ns) into the FPGA. This operating frequency is set to meet the hardware delays that occurred in the proposed architecture. This was done by dividing the actual input clock frequency into half such that the actual operating frequency of the DWT CRS MISC architecture could meet the available time delays from the architecture.

For future improvement, usage of only single frequency clock for the DWT CRS MISC architecture can be considered. This can be implemented by using a buffer delay to delay the clock signal fed into the Memory (Block RAM). With delayed clock, the Memory will be able to response correctly towards the changes in control signals. This will provide the minimum required amount of time for the control signals and registers' output data to become stable. Later on, the Memory will only be able to input/output the correction information, such as the Operands' memory addresses and the Operands' values.

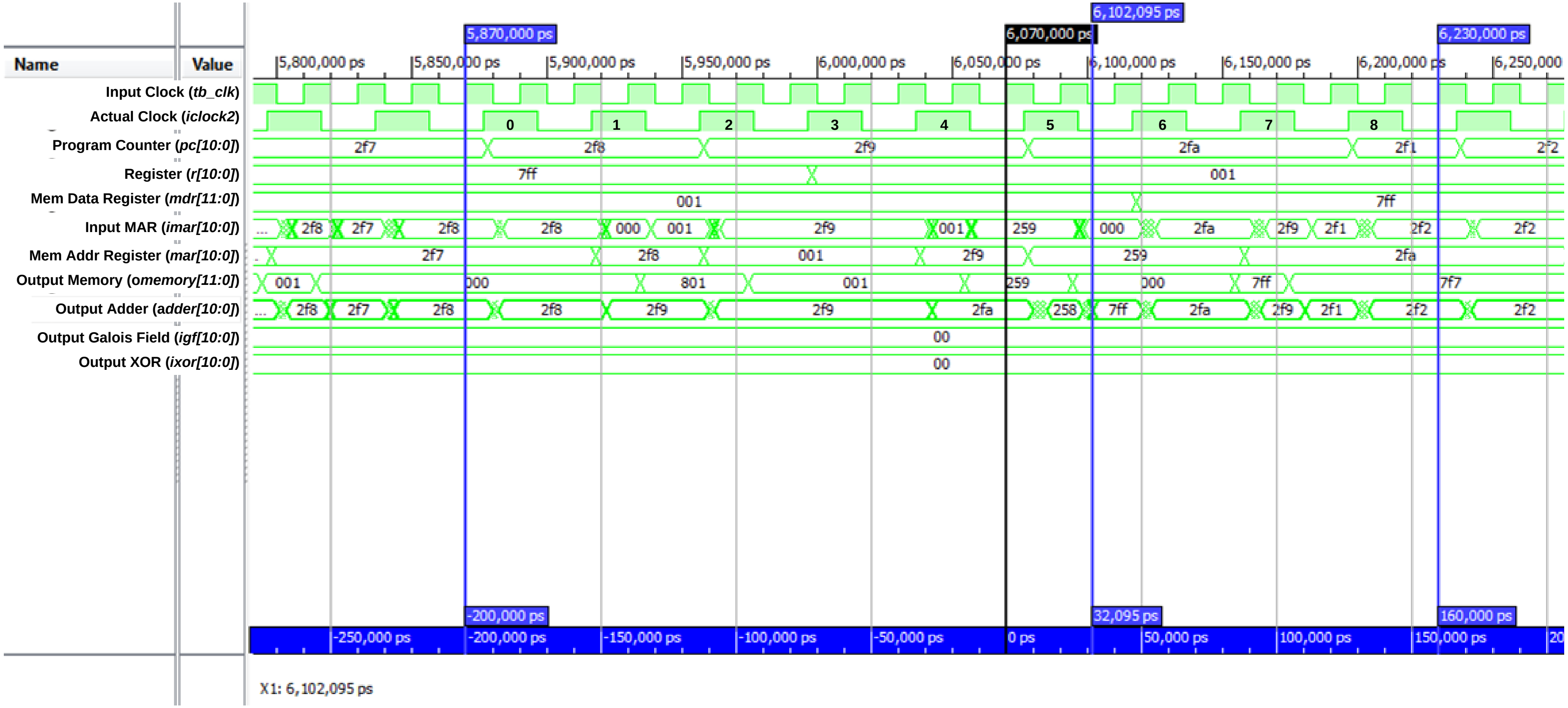


Figure 127 Post & Route Simulation Waveforms SBN Instruction for DWT CRS MISC.

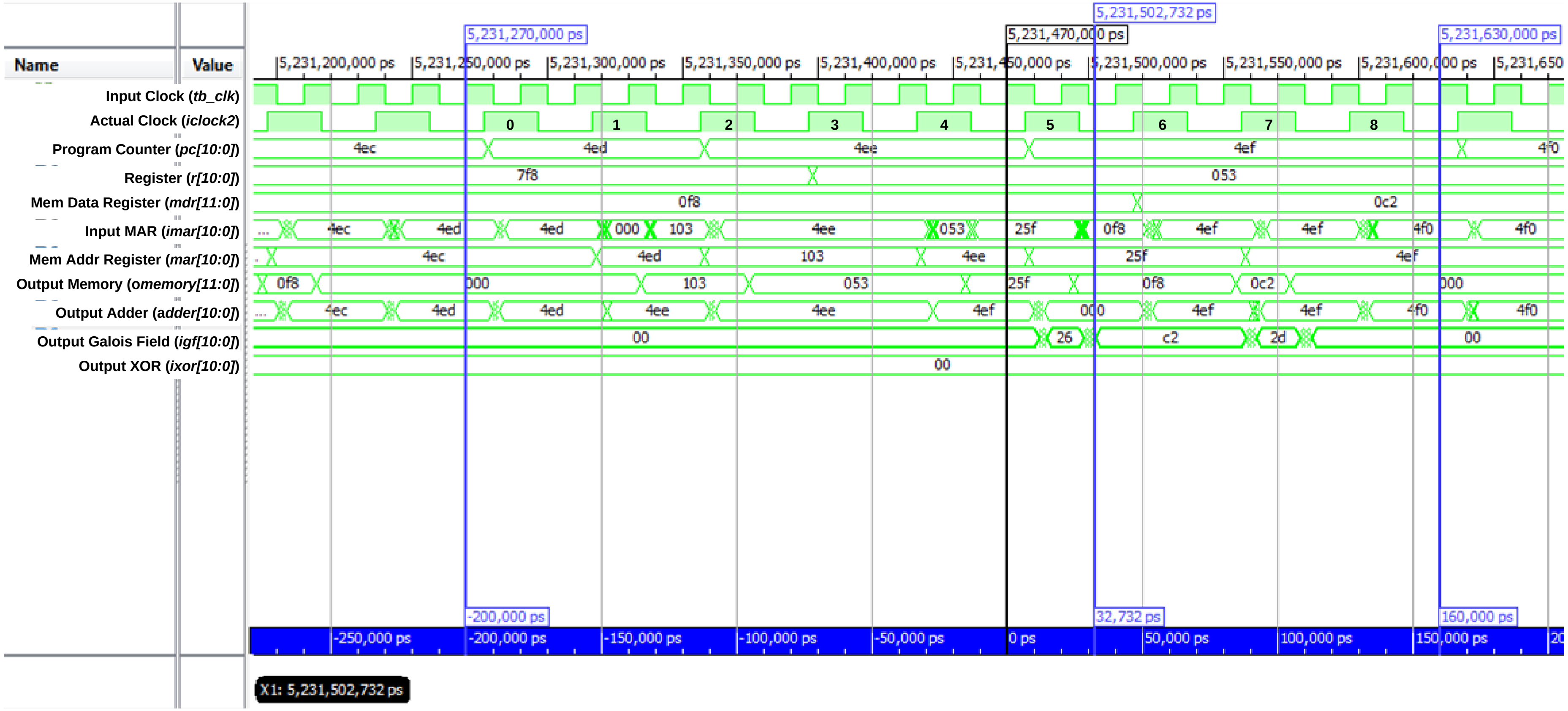


Figure 128 Post & Route Simulation Waveforms GF MULT Instruction for DWT CRS MISC.

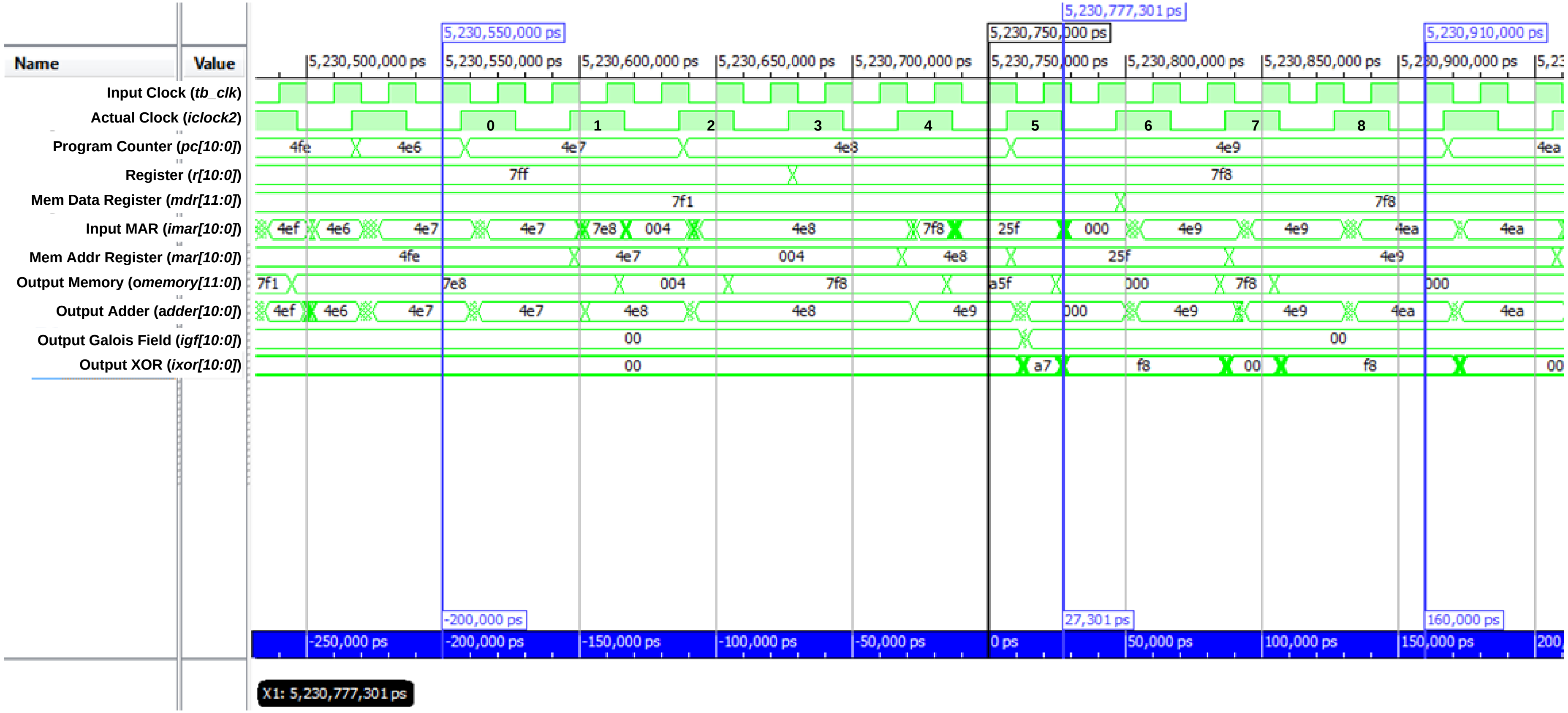


Figure 129 Post & Route Simulation Waveforms XOR Instruction for DWT CRS MISC.

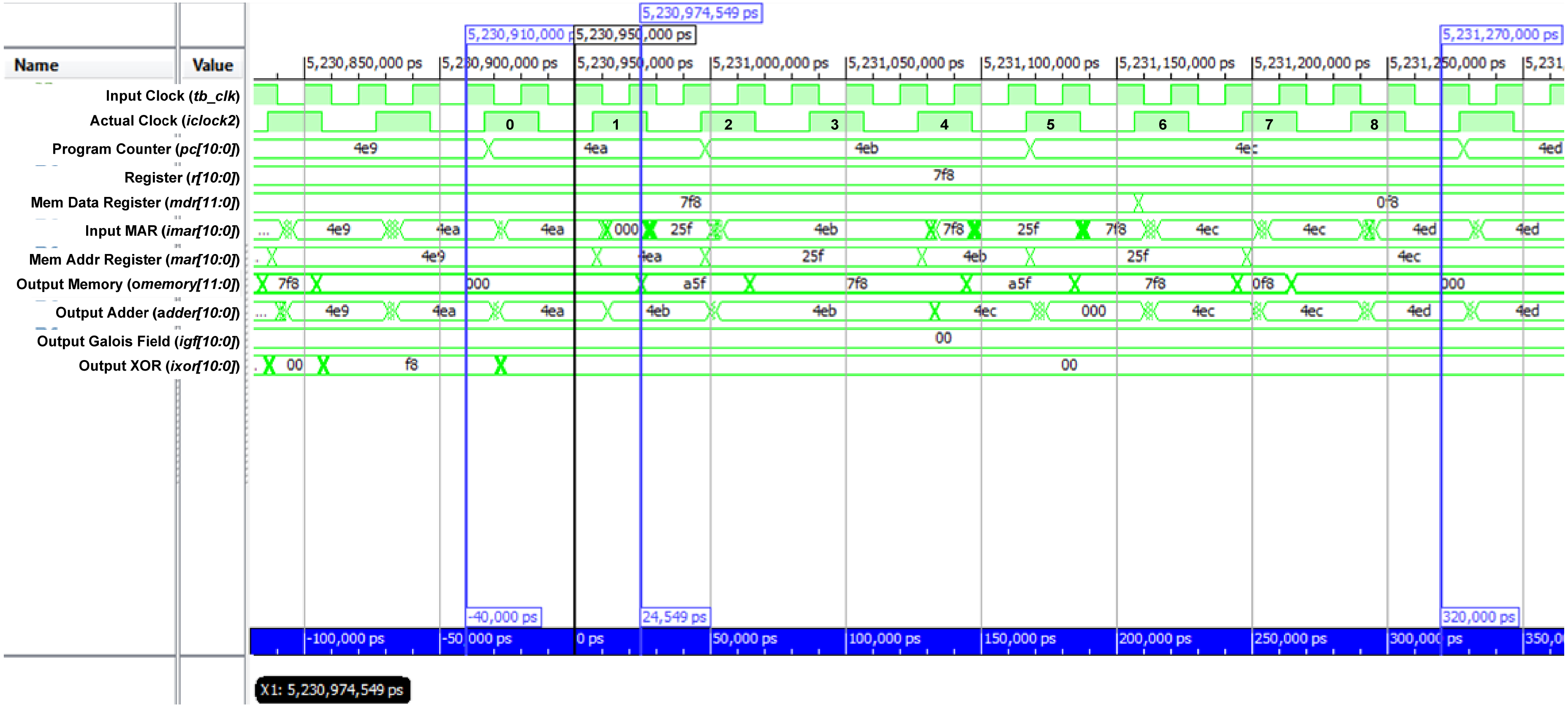


Figure 130 Post & Route Simulation Waveforms 11TO8 Instruction for DWT CRS MISC.

4.3 DWT CRS MISC HARDWARE UTILISATION

To verify that the DWT CRS MISC architecture can actually operate in the FPGA, a hardware implementation was performed onto a FPGA. This is done by writing the VHDL code that described the DWT CRS MISC architecture and performed a hardware implementation onto the Xilinx Spartan-3L FPGA. The hardware utilisation for such implementation is discussed in Section 4.3.1. Initial input image data was set in the memory such that it allowed the programmed DWT CRS MISC architecture to process the data and then produced the corresponding encoded data. After that, the MISC architecture produced encoded data were compared with the MATLAB generated encoded data. By comparing the data, both the encoded data produced from the DWT CRS MISC architecture and the MATLAB generated encoded data were of the same values. Therefore, this verified that the DWT CRS MISC architecture operates and processes the input image data correctly.

4.3.1 DWT CRS MISC in FPGA

The hardware utilisations for the proposed DWT CRS MISC architecture implemented onto Xilinx Spartan-3L FPGA was studied. This was obtained by synthesising the developed DWT CRS MISC architecture in the Xilinx ISE Design Suite 11.5. For this implementation, the MISC architecture required a total of 144 Slices (i.e. 94 Flip-Flops, 248 LUTs, 2 Block RAMs), which is shown in Table 13. Note that in Spartan-3L FPGA, each of the Slices contains of 2 Flip-Flops and 2 Four-Input LUTs [146] [147]. Besides synthesizing into one type of FPGA platform, the DWT CRS MISC architecture is also considered for Xilinx Virtex-II and Xilinx Spartan-3E. The hardware utilizations of the DWT CRS MISC are 142 Slices (i.e. 92 Flip-Flops, 225 LUTs, 2 Block RAMs) and 129 Slices (i.e. 66 Flip-Flops, 223 LUTs, 1 Block RAMs). Different FPGA platform was also considered because it is used for comparison of existing method with the same FPGA family.

Table 14 shows that there are 6 existing techniques that were previously developed for WSNs, whereby they either performed compression, encryption and error corrections. Some of the listed techniques or systems performed the aforementioned techniques separately rather than in a single architecture. For example, the Cordic Loeffler Discrete Cosine Transform (CL-DCT) occupied 1,060 Slices in Xilinx Spartan-3L [25]. The Low-Density Parity-Check (LDPC) error correction encoder utilized a total of 870 Slices in Xilinx Virtex-II [98].

Table 13 Hardware utilisation of DWT CRS MISC architecture in Spartan-3L FPGA.

Components	**Quantity**	**Total**	**Usage**
Slices	144	13,312	1.08%
Flip-Flops	94	26,624	0.35%
4-Input LUTs	248	26,624	0.92%
- Logic	*226*	-	-
- Route-thru	*22*	-	-
- Dual Port RAMs	*0*	-	-
- Shift Registers	*0*	-	-
Bonded IOB	26	221	26.70%
Block RAMs	2	32	6.25%
GCLKs	2	8	25.00%

For the Reed Solomon (RS) error correction, the synthesised RS Linear Feedback Shift Register (LFSR) method [17] shown in Table 14, required hardware utilisation of 415 Slices and the power consumption of 198.9mW. As for the developed RS MISC architecture, it only required 161 Slices and 164.2mW. It can be seen that the RS MISC architecture has 61.2% lower hardware utilisations and 17.4% lower power consumption as compared with the RS LFSR method of hardware implementation.

Combining the encryption and error correction modules together, with the AES MISC [148] and followed by the RS MISC, would require a total of 480 Slices. The combined AES MISC and RS MISC may required power consumption of at least 199.7mW, which is contributed by the AES MISC. The stated power consumption have not included the power consumed by the RS MISC and it may be even higher than the stated value. This method of implementing the image processing system has large amount of hardware utilisation and require relatively high power consumption. Instead of combining two separate modules together, a CRS MISC architecture was developed that uses the single code of encryption and error correction code (CRS coding scheme). The developed CRS MISC architecture only required a total of 155 Slices and power consumption of 167.3mW. It can be seen that using single code of encryption and error correction code is better than using two different modules in terms of the hardware utilisations and power consumption.

Table 14 Hardware utilisations of developed and existing method used in similar FPGA technology (Spartan-3, Virtex-II) for WVSNs/WSNs.

Designs	**Functions**	**Slices**	**Flip-Flops**	**4-Input LUTs**	**Block RAMs**	**Power (mW)**
Xilinx Virtex-II						
DWT CRS MISC	**Compression, Encryption, Error Correction**	**142**	**92**	**225**	**2**	**N/A**
CRS MISC	Encryption, Error Correction	132	83	209	1	N/A
RS MISC	Error Correction	138	88	218	1	N/A
LDPC [98]	Error Correction	870	Not Mentioned	Not Mentioned	19	Not Mentioned
Xilinx Spartan-3E						
DWT CRS MISC	**Compression, Encryption, Error Correction**	**129**	**66**	**233**	**1**	**179.0**
CRS MISC	Encryption, Error Correction	120	56	216	1	178.9
RS MISC	Error Correction	124	61	221	1	178.9
ECBC [21]	Encryption, Error Correction	1,691	Not Mentioned	Not Mentioned	Not Mentioned	Not Mentioned
Xilinx Spartan-3L						
DWT Filter Module [149]	Compression	1,458	806	2714	14	424.6
CL-DCT [25]	Compression	1,060	Not Mentioned	Not Mentioned	Not Mentioned	Not Mentioned
RS LFSR [17]	Error Correction	415	346	720	2	198.9
RS MISC	Error Correction	161	91	279	1	164.2
AES MISC [148]	Encryption	319	157	569	2	199.7
CRS MISC	Encryption, Error Correction	155	87	269	1	167.3
DWT CRS MISC	**Compression, Encryption, Error Correction**	**144**	**94**	**248**	**2**	**167.3**

4.3.2 DWT CRS MISC: Further Improvements

For further improvements, in hardware utilisation and power consumption, the developed DWT CRS MISC architecture was also considered for Xilinx Spartan-6 FPGA implementation. The hardware utilisation of the DWT CRS MISC architecture that was synthesized in Xilinx Spartan-6 FPGA, required a total of 66 Slices (i.e. 87 Flip-Flops, 176 LUTs, 2 Block RAMs). Lower hardware utilisation (in terms number of Slices) was expected for the implementation of the MISC architecture is because each Slices in the Spartan-6 FPGA contains of 8 Flip-Flops and 4 Six-Input LUTs [150], which is more than the number of Flip-Flops and LUTs in each Slices for the Spartan-3L FPGA. The hardware utilisation for the developed DWT CRS MISC architecture to be implemented onto the Spartan-6 FPGA is shown in Table 15. Therefore, there is a huge reduction in hardware complexity if the developed DWT CRS MISC architecture to be implemented into the Spartan-6 FPGA.

In Table 16, the power consumption for the synthesised DWT CRS MISC architecture in two different type of FPGAs were estimated using the Xilinx XPower Analyzer 11.5. The total power consumption for MISC architecture in Spartan-3L FPGA was 167.29mW and Spartan-6 FPGA was 21.63mW. The differences in power consumption between these two technologies of FPGA is because that the Spartan-3L is 90nm FPGA [151] and the Spartan-6 is 45nm FPGA [152]. Since there is improvement on the technology of the FPGA, the Spartan-6 FPGA has 87.1% lower power consumption as compared to Spartan-3L FPGA [152]. Therefore, there is an great improvement in power consumption of the DWT CRS MISC architecture when it will be implemented onto the Spartan-6 FPGA.

Table 15 Hardware utilisation of DWT CRS MISC architecture in Spartan-6 FPGA.

Components	**Quantity**	**Total**	**Usage**
Slice	66	2,278	2.90%
Flip-Flops	87	18,224	0.48%
6-Input LUTs	176	9,112	1.93%
- Logic	*171*	-	-
- Route-thru	*5*	-	-
- Memory	*0*	-	-
Bonded IOB	26	232	11.21%
Block RAMs	2	32	6.25%

In Table 17, it can be seen that there are at least 54.2% reduction in number of hardware utilisations when the developed RS MISC, CRS MISC and DWT CRS MISC architectures were synthesised into Spartan-6 FPGA. The great amount of reduction in hardware utilisations is because the Spartan-6 FPGA uses the 6-Input LUTs required less LUTs for the same implementations in Spartan-3L FPGA which uses the 4-Inpu LUTs [153].

Table 16 Xilinx XPower estimated power consumption of DWT CRS MISC architecture.

	Power (W)	
	Spartan-3L	**Spartan-6**
Clocks	0.00100	0.00076
Logics	0.00018	0.00007
Signals	0.00031	0.00003
IOs	0.02042	0.00007
BRAMs	0.00013	0.00080
Total Quiescent Power	0.14525	0.01989
Total Dynamic Power	0.02204	0.00174
Total Power	0.16729	0.02163

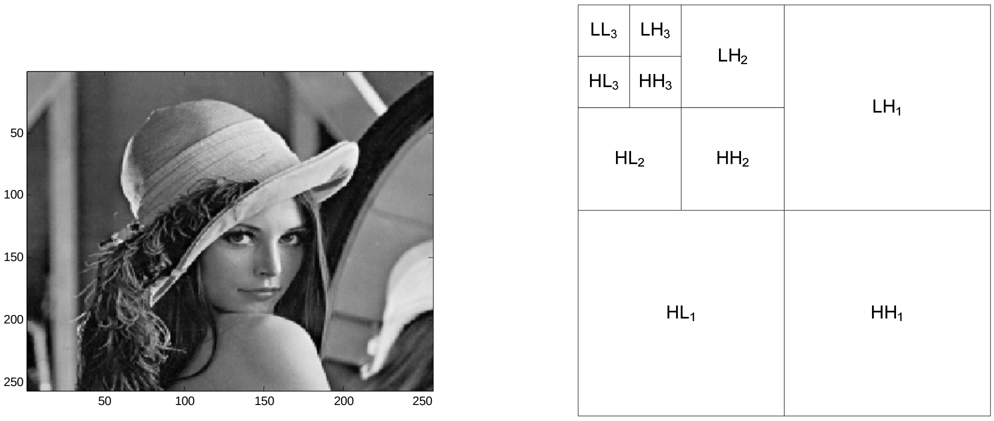
Table 17 Hardware utilisations of developed and existing method in Spartan-6 FPGA for WVSNs/WSNs.

Designs	**Functions**	**Slices**	**Flip-Flops**	**6-Input LUTs**	**Block RAMs**	**Power (mW)**
Xilinx Spartan-6						
Crypto-Processor [117]	Encryption	4,828	Not Mentioned	Not Mentioned	19	17.0
RS MISC	Error Correction	60	82	167	1	17.4
CRS MISC	Encryption, Error Correction	60	77	160	1	21.3
DWT CRS MISC	**Compression, Encryption, Error Correction**	**66**	**87**	**176**	**2**	**21.6**

Table 17 shows the Crypto-Processor encryption required a total of 4,828 Slices [117]. As for the developed DWT CRS MISC, it only takes 1.3670% hardware utilization of the Crypto-Processor. This is a huge reduction in hardware utilizations compared to the Crypto-Processor. As a result, the developed DWT CRS MISC architecture has a low hardware utilisations as compared to the different existing techniques used in compression, encryption and error correction for WVSNs/WSNs. Meanwhile, the DWT CRS MISC has the capabilities in reducing large amount of image data, providing data security and data reliability combined together in a single architecture with low hardware utilisations.

4.4 DWT RECONSTRUCTED IMAGE QUALITY

In actual implementation, three Digi XBee RF transceivers were used to form a wireless communication network, with the ability to transmit data in between these transceivers. This simulated the environment of WVSNs that had a camera sensor node sending image data to the base-station (sink) through an intermediate node (router). The Digi XBee RF transceivers used has a typical transfer rate of 16kbps (2kBps) with power consumption of 50mW [154]. With these information, transmission time and energy required to transmit different amount of DWT compressed image data could be evaluated. In the meantime, simulations were performed in MATLAB environment to evaluate the quality of reconstructed image. This was done by performing a 3-level Lifting Scheme DWT Haar filtering onto the image ‘lena1.tif’ with size of 256 pixels x 256 pixels, shown in Figure 131(a). The image was decomposed into 3 different levels of subband DWT coefficients (LL_3, LH_3, HL_3, HH_3, LH_2, HL_2, HH_2, LH_1, HL_1, HH_1) and they were all arranged in the order as shown in Figure 131(b). The number of DWT coefficients transferred across the WVSNs to the base-station (sink) significantly affected the quality of reconstructed image. With more DWT coefficients received, the quality of reconstructed image was substantially improved (increased Peak Signal-to-Noise Ratio (PSNR) values). Table 18 shows the number of DWT coefficients sent with respect to the quality of reconstruct image, transmission time and energy. It can be seen that there is an increase in transmission time and energy consumption when more DWT coefficients were transmitted.



(a) Original ‘lena1.tif’ image.

(b) DWT coefficients arrangement.

Figure 131 DWT Haar on 'lena1.tif' image.

Transmitting all the DWT coefficients to the base-station allowed a lossless reconstruction of compressed image, which is similar to transmitting an uncompressed image. However, this required the largest amount of data (65,536 bytes) to be transferred across the wireless networks and consuming at least 1.6384 J of energy. This increased the power consumption of the transceiver and it was not recommended for WVSNs that operated on limited energy resources (such as battery powered). By considering the LL_3 coefficients, the amount of data to be transmitted was the lowest (1,024Bytes) among all the scenarios in Table 18. The amount of LL_3 coefficients required to be transferred constituted only 1.563% of all the DWT coefficients. The energy required (0.0256J) to transmit LL_3 coefficients was the lowest compared to other scenarios in Table 18. However, this produced low image quality (PSNR = 20.4009dB) and less information could be extracted from the reconstructed image. To achieve an acceptable image quality and low energy consumption, transmission of level 2 and level 3 DWT coefficients to the base-station were considered. For this case, it only consumed approximately 25% of the energy (0.4096 J) required to transmit all the DWT coefficients. As highlighted in Table 18, transmitting this amount of DWT coefficients still allowed sufficient information to be extracted from the reconstructed image (with image quality at 27.6737dB). In the meantime, there was a significant improvement in data transmission time between the node to base-station since less amount of image data are needed to be transferred.

Table 18 Image quality, transmission time and energy for different amount of DWT (Haar) coefficients transferred.

DWT Coefficients	PSNR (dB)	Bytes Transferred	Time (s)	Energy (J)
LL_3	20.4009	1,024	0.512	0.0256
LL_3 LH_3	20.8603	2,048	1.024	0.0512
LL_3 LH_3 HL_3	22.9123	3,072	1.536	0.0768
LL_3 LH_3 HL_3 HH_3	23.4419	4,096	2.048	0.1024
LL_3 LH_3 HL_3 HH_3 LH_2	24.1629	8,192	4.096	0.2048
LL_3 LH_3 HL_3 HH_3 LH_2 HL_2	26.8969	12,288	6.144	0.3072
LL_3 LH_3 HL_3 HH_3 LH_2 HL_2 HH_2	**27.6737**	**16,384**	**8.192**	**0.4096**
LL_3 LH_3 HL_3 HH_3 LH_2 HL_2 HH_2 LH_1	28.9856	32,768	16.384	0.8192
LL_3 LH_3 HL_3 HH_3 LH_2 HL_2 HH_2 LH_1 HL_1	37.1123	49,152	24.576	1.2288
LL_3 LH_3 HL_3 HH_3 LH_2 HL_2 HH_2 LH_1 HL_1 HH_1	∞	65,536	32.768	1.6384

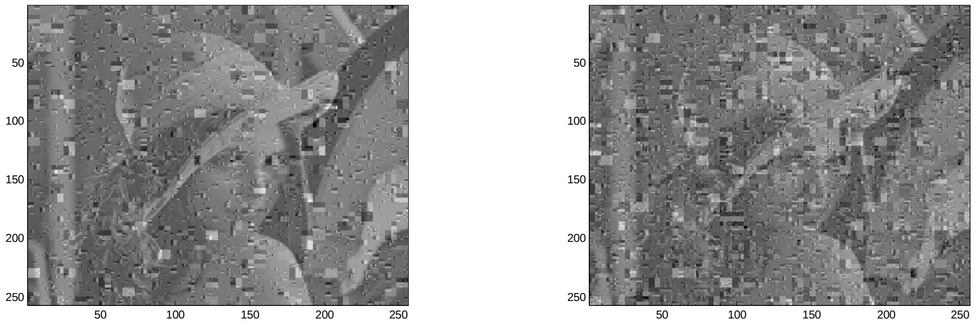
4.5 ERRORS ON DWT COEFFICIENTS

By introducing different number of errors into the DWT coefficients, the quality of the reconstructed image (measured in PSNR, dB) could be determined. This was done by finding the average image quality for 10 reconstructed image. Without CRS coding scheme protecting the DWT coefficients, the quality of the reconstructed image degraded if errors were to occur onto these DWT coefficients. Having no error occurred onto the DWT coefficients, the quality of the reproduced image was infinite since the reproduced image did not have any difference as compared to the original image. The quality (in PSNR) of the reconstructed image from the DWT coefficients with errors occurred into each packet of DWT coefficients, are shown in Figure 132(a) to Figure 132(d). From these figures, it can be seen that the quality of reconstructed image reduced as more errors occurred on each codeword. Table 19 shows the quality of reproduced image degraded as more errors were introduced into each packets (blocks) of data, each consisted of 16 DWT coefficients. For the case with 4 errors, the reconstructed image with “Lena” inside was almost beyond recognition (PSNR = 7.9531 dB), as compared to Figure 132(a).

Table 19 Quality of 10 reconstructed image with errors on DWT coefficients.

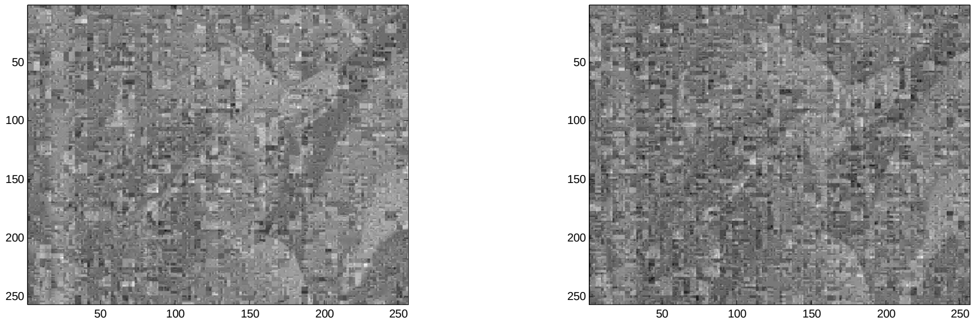
No Errors	PSNR (dB)										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Avg.**
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	13.8073	13.6610	13.7012	13.4123	13.5312	13.5479	13.3009	13.2482	13.7173	13.7697	13.5697
2	11.0146	10.7739	10.5853	10.8349	10.8505	10.6495	10.5056	10.8603	10.7916	10.7794	10.7646
3	8.9746	8.8661	8.9741	8.9879	9.0561	9.1719	9.0317	9.0551	9.0776	9.0570	9.0252
4	7.8621	7.9643	7.8135	7.9449	8.1282	7.7630	8.0290	7.9857	8.0838	7.9562	7.9531

By introducing the CRS(20,16) coding scheme, the DWT coefficients were encoded to produce a codeword with 20 symbols, where each symbol was 8-bit. For up to 4 errors, the CRS(20,16) coding scheme was capable of recovering correct DWT coefficients for each codeword. As a result, high quality of reconstructed image was achievable, even there were 20% of errors on the encoded DWT coefficients. Figure 133 shows the quality of reconstructed image (PSNR = 44.3373 dB) from the DWT coefficients with 4 errors on each encoded codeword. The quality of the reconstructed image was not infinite is due to the fact that there was slight precision error while performing the conversion of DWT coefficients from double precision to unsigned integer in 8-bit (for example 8.5 might became 8.0) in the MATLAB environment.



(a) One error occurred.

(b) Two errors occurred.



(c) Three errors occurred.

(d) Four errors occurred.

Figure 132 Errors occurred on each packet of DWT coefficients.

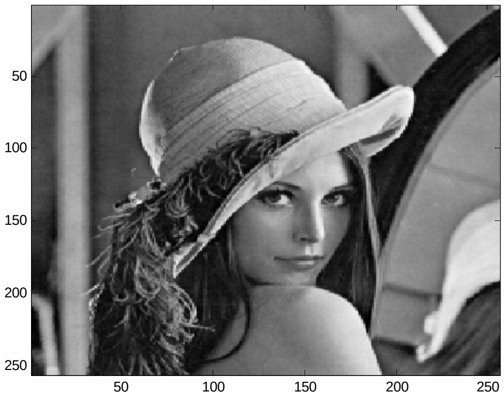


Figure 133 Reconstructed CRS encoded compressed image data with 4 errors on each packet.

4.6 CRS CODING SCHEME CONFIGURATION

An important question to consider is the security level of the selected CRS coding scheme. For different configuration of the CRS coding scheme, the coding scheme provides different specific level of security. The security level are determined by the number of trials that are required to be performed by the adversary to decode the encoded data. In Table 20, it shows the number of possible trials required for different configurations of the CRS coding scheme. From Table 20, I_m is the total number of data used for encoding in bits, and I_k is the number of bits that are needed to generate the key, which is the generator matrix G_{SEC}. I_{2m} is the key, in number of bits, required to recover the encoded data using the CRS coding scheme. It is necessary to select a particular configuration which has a large enough key space security. This is to prevent the adversary, having sufficient computational power, to recover the ciphertext (encrypted image data) that is intercepted during transmission. From Table 20, it can be seen that for n = 20, L = 8 bits and m = 16, there are I_m = 128 bits of data that will be encoded by using, I_k = 288 bits of key. The highlighted configuration in Table 20 is the selected CRS(20,16) coding scheme that are used in encoding the DWT coefficients.

Table 20 Number of possible trials, P for different CRS coding scheme configurations.

n	**m**	**I_m [bits]**	**n + m**	**l_k [bits]**	**r (n/m)**	**L**	**2^L**	**2m**	**I_{2m} [bits]**	**Probability, P**
20	1	8	21	168	20.0000	8	256	2	16	65280
20	2	16	22	176	10.0000	8	256	4	32	4195023360
20	3	24	23	184	6.6667	8	256	6	48	$2.6534\text{x}10^{14}$
20	4	32	24	192	5.0000	8	256	8	64	$1.6518\text{x}10^{19}$
20	5	40	25	200	4.0000	8	256	10	80	$1.0118\text{x}10^{24}$
20	6	48	26	208	3.3333	8	256	12	96	$6.0981\text{x}10^{28}$
20	7	56	27	216	2.8571	8	256	14	112	$3.6157\text{x}10^{33}$
20	8	64	28	224	2.5000	8	256	16	128	$2.1088\text{x}10^{38}$
20	9	72	29	232	2.2222	8	256	18	144	$1.2096\text{x}10^{43}$
20	10	80	30	240	2.0000	8	256	20	160	$6.8228\text{x}10^{47}$
20	11	88	31	248	1.8182	8	256	22	176	$3.7839\text{x}10^{52}$
20	12	96	32	256	1.6667	8	256	24	192	$2.0631\text{x}10^{57}$
20	13	104	33	264	1.5385	8	256	26	208	$1.1056\text{x}10^{62}$
20	14	112	34	272	1.4286	8	256	28	224	$5.8234\text{x}10^{66}$
20	15	120	35	280	1.3333	8	256	30	240	$3.0140\text{x}10^{71}$
20	**16**	**128**	**36**	**288**	**1.2500**	**8**	**256**	**32**	**256**	**$1.5326\text{x}10^{76}$**
20	17	136	37	296	1.1765	8	256	34	272	$7.6556\text{x}10^{80}$
20	18	144	38	304	1.1111	8	256	36	288	$3.7560\text{x}10^{85}$
20	19	152	39	312	1.0526	8	256	38	304	$1.8096\text{x}10^{90}$
20	20	160	40	320	1.0000	8	256	40	320	$8.5607\text{x}10^{94}$

Using the selected coding configuration, the adversary will require I_{2m} = 256 bits of key to recover the encrypted data. The adversary would need to perform a total of P = 1.5326x10^{76} trials in order to decrypt the data. This coding configuration of the CRS has a similar key space security as that provided by AES encryption [155] with 128 bits of input data and 128 bits of key. Figure 134 shows that as the number of data to be encoded increases, the number of possible key combination (trials) also increases. This increases the security level of the encoded (encrypted) data since the adversary needs to perform a large number of trials to decode (decrypt) the data. The selected CRS coding scheme has the similar security key space as the AES encryption scheme which considered as a standard use for encrypting data by the US National Institute of Standards and Technology (NIST) [58]. Therefore, this CRS(20,16) coding scheme is selected for use in the proposed system. In comparison with the original image, visual simulation results were also performed to further verify that the image data is well decorrelated, as shown in Figure 135(a) and Figure 135(b).

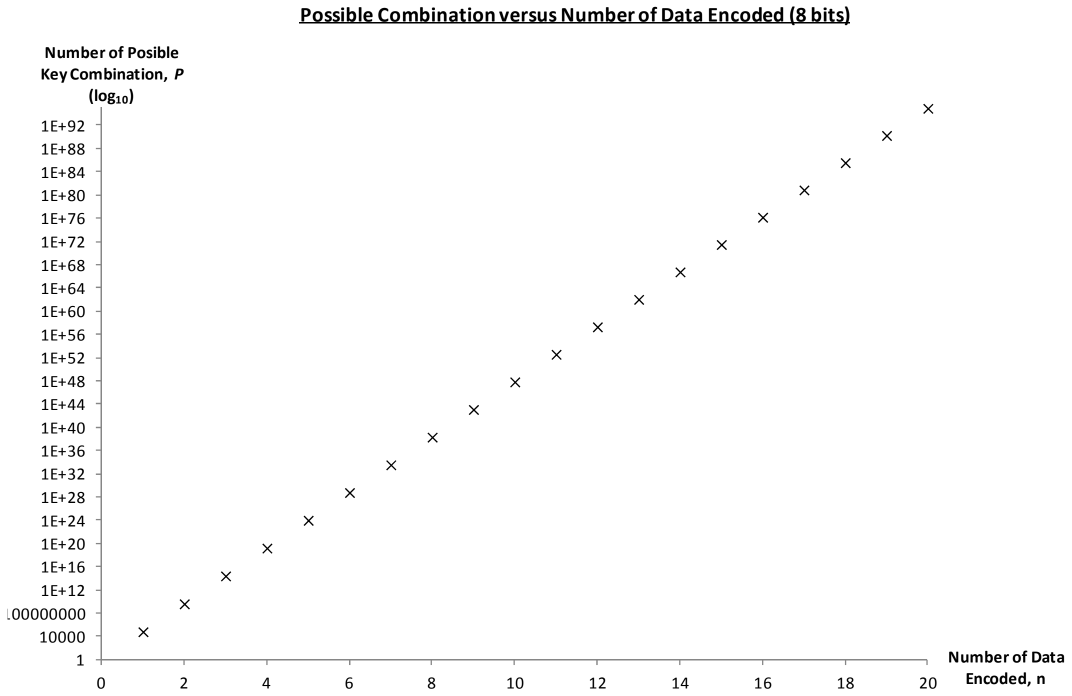
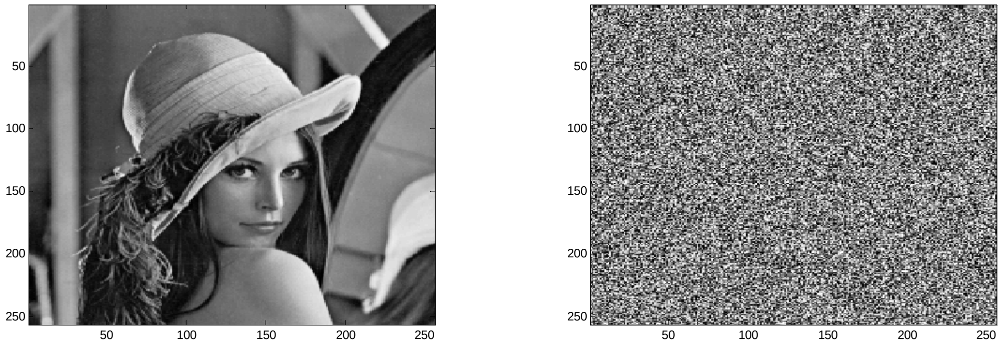


Figure 134   Possible combination versus the number of data encoded.



(a) Original ‘Lena.tif’ image before encoded.

(b) Encoded Lena image.

Figure 135   CRS coding scheme performed onto Lena image.

4.7 SUMMARY

In this Chapter 4, the Behavioral and Post & Route simulations were performed onto the DWT CRS MISC architecture. From these simulations, the hardware time delays that occurred in the control signals combinational logic circuit and the operation of the programme instructions in the DWT CRS MISC architecture were determined. As a result, the correct operating frequency of the DWT CRS MISC architecture could be determined to meet the hardware time delays. Afterwards, the DWT CRS MISC architecture was then synthesised such that it would be implemented onto the FPGA. As compared with existing techniques (eg. ECBC), lower amount of hardware utilisation for the FPGA synthesised DWT CRS MISC architecture was achieved. The Lifting Scheme DWT Haar was used to reduce the large amount of image data such that less transmission of data was required. Instead of transmitting the full size image data, this would relatively reduce the amount of energy required for image data transmission. With the use of CRS coding scheme, the degradation in quality of reconstructed image could be avoid because the base station could rectify a few errors that occurred onto the compressed image data. Meanwhile, based on the study on security level of different CRS coding scheme, the selected CRS(20,16) coding scheme used in the MISC architecture offered a similar security protection level as compared to the AES encryption scheme.

CHAPTER 5

HARDWARE IMPLEMENTATIONS

The designed DWT CRS MISC architecture was integrated into the WVSNs to form a complete system. Before the DWT CRS MISC architecture was developed, a SPIHT MIPS processor was integrated with the CRS MISC architecture to reduce the image and encode (encrypt) the reduced image data. However, the SPIHT MIPS and CRS MISC integrations required large amount of hardware utilisation, which is 5,041 Slices (3,060 Flip-Flops, 8,795 4-Input LUTs). Consequently, a DWT filter module was used to replace the SPIHT MIPS processor such there is a reduction in hardware utilisation and still capable of reducing the image data. However, the combined DWT filter module and CRS MISC architecture still requires a significant amount hardware utilisation, which is 2,552 Slices (1,440 Flip-Flops, 4,403 4-Input LUTs). As such, a single architecture DWT CRS MISC processor was developed that required lower amount of hardware utilisation as compared to the two previously developed system. Section 5.1 presents the developed system that used the CRS MISC architecture to perform a selectively Secure Erasure Code (SEC) onto the SPIHT coefficients, which is generated from the SPIHT MIPS. The developed system in Section 5.2 was to reduce the image data through the use of DWT filter module and encode (encrypt) the reduced data by the CRS MISC processor. Lastly, Section 5.3 presents the use of DWT CRS MISC architecture to perform DWT image compression and CRS encoding in a single architecture.

5.1 SELECTIVE SEC ON SPIHT COEFFICIENTS FOR WVSN

The objectives of reducing the amount of data transmission across the WVSNs and to extend the battery lifespan of the sensor nodes, can be achieved by performing selective secure error correction on compressed coefficients for the WVSNs. By combining two different modules, where one performs data compression and another provides error protection on the data, the image data produced from the sensor nodes can be compressed and the mapping bits will be protected from errors occurring during transmission in the WVSNs. With low amount of image data transmitting across WVSNs, this decreases the transmission bandwidth thus reducing the power consumption of sensor nodes required in transmitting data. The use of FEC coding scheme provides less retransmission of errors occurred on the compress data thus reduce the power consume by sensor nodes in retransmission of correct data.

This Section 5.1 presents a system that use SPIHT MIPS processor [64] to reduced the image data and CRS MISC processor together to have both secure and reliable data transmission across the WVSNs. The CRS MISC processor performed the CRS coding scheme onto the reduced image data, which has the capabilities of recovering 4 lost packets from the remaining correctly received codeword. At the same time, these data were encrypted without exposing its actual data contents during transmission in the WVSNs. Similar key space security in comparison with the AES encryption, which is shown in Section 4.6, was selected for the CRS MISC configuration. Both the SPIHT MIPS processor and CRS MISC processor were implemented onto a Field Programmable Gate Array (FPGA) to demonstrate the feasibility in implementing a complete WVSN sensor node system.

5.1.1 System Overview: Selective SEC on SPIHT Coefficients

The developed system that performed selective Secure Erasure Code (SEC) for the SPIHT coefficients in WVSNs, consists of one module of 330 lines Charge Coupled Device (CCD) camera, memory buffer, SPIHT MIPS processor, CRS MISC processor, two Digi XBee ZB RF modules and a computer. Initially, the CCD camera captured one frame of image at the size of 128×128 pixels. The captured image data were stored into the image memory buffer. Then the SPIHT MIPS processor [64] processed these image pixels which were available in the image memory buffer. Later on, the image pixels undergone DWT decomposition and followed by the SPIHT encoding that produced many bit-streams of data. Later on, these streams of data were sorted into mapping bits and refinement bits that were stored into the mapping bits memory buffer and refinement bits memory buffer respective.

Since each fragments (symbols) in the CRS coding scheme is in one byte, the mapping bits were converted into mapping bytes. Once the conversion had been performed, the CRS(20,16) coding scheme was used to encode these mapping bytes by the CRS MISC processor. These encoded mapping bytes (bits) will be transmitted across the wireless channel in packets, as shown in Figure 136. These mapping bytes (bits) produced by SPIHT encoder do not have any tolerances for errors since the tree structure of the compressed image will be destroyed with incorrect values. In Figure 137, the refinement bits will not be protected by the CRS coding scheme since the mapping bytes (bits) are important in reconstructing the images. Consequently, the refinement bits are arranged in packets with each consists of 100 bytes of refinement bits and 1 byte of packet header.

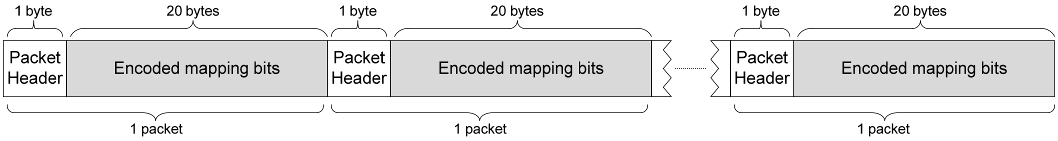


Figure 136 CRS protected mapping bytes (bits) in packets for wireless transmission.

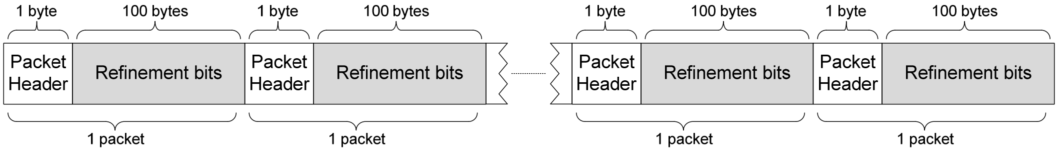


Figure 137 Refinement bytes (bits) in packets for wireless transmission.

The codeword will be transmitted across the wireless network through the Digi XBee ZB RF modules to the sink (base-station) once a block of codeword was completely produced by the CRS MISC processor. The sink received the codeword and performed the CRS decoding to recover the mapping bytes. After which, the SPIHT decoder performed the reconstruction of original image once all the mapping bytes and refinement bytes were received. Both CRS and SPIHT decoding were performed in MATLAB environment at the sink (base-station). The flow of the image data from the CCD camera through the Digi XBee RF modules to the sink is shown in Figure 138.

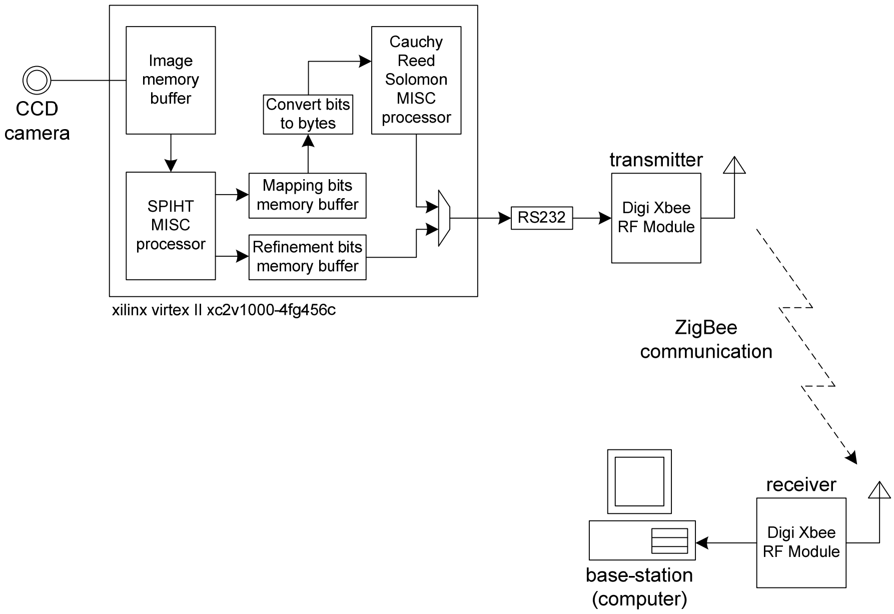


Figure 138 Selective SEC coding on SPIHT coefficients in WVSNs.

The Digi XBee RF modules used were ZigBee network based which is considered as a low data rate communication and low cost wireless networking protocol. From the datasheet [154], the Digi XBee RF module have a data throughput of 21kbps operating at Industrial, Scientific and Medical (ISM) radio band frequency of 2.4GHz. The RS232 is not the limiting factor as the Digi XBee RF modules has a lower data throughput. As a result, the RS232 baudrate was set to 19,200bps such that to match the data throughput of the Digi XBee RF modules.

5.1.2 SPHIT Reconstructed Image Quality

The quality of the reconstructed images were compared by introducing errors into the SPIHT coefficients (mapping bytes and refinement bytes). This was to study the effect of errors in reconstructing the SPIHT compressed image. An image (Lena.tif) is used for this study and the SPIHT encoding was performed in MATLAB environment. In this study, the quality of reconstructed compressed image was measured, in terms of PSNR (dB), at certain of Bits Per Pixel (bpp). The SPIHT coefficients without error protection are arranged in packets with size of 20Bytes each, such that it is similar to the codeword size produced by the CRS coding scheme. Then errors were introduced into each of these packets, either onto the mapping bytes or the refinement bytes.

For the case without errors on the SPIHT coefficients, the quality of the reconstructed image obtained after compression was 58.7053 dB. From Table 21, the quality of reconstructed compressed image dropped significantly once there were errors that occurred onto the mapping bytes. The reason is the reconstruction of original image is not possible when there is incorrect tree structure of the compressed image which is represented by these error mapping bytes (bits). The reconstruction of the original image is not possible at 0.1bpp, even for higher number of Bits Per Pixel (bpp). As a result, there is a minor variation on the reconstructed image quality (PSNR) for higher number of bpp. This can be seen that with only one error, the quality of reconstructed image dropped to 7.5827 dB, which is a significant reduction in the quality of reconstructed image.

Table 21 Reconstructed image quality with errors on mapping bytes.

bpp	**PSNR (dB)**				
	No Error	**1 Error**	**2 Errors**	**3 Errors**	**4 Errors**
0.10	23.9789	7.5464	5.6559	2.8416	2.0719
0.25	27.4265	7.5467	5.6837	2.8734	2.1238
0.50	30.9458	7.5696	5.7104	2.9039	2.1505
1.00	35.8657	7.5799	5.7170	2.9132	2.1619
uncompress	58.7053	7.5827	5.7190	2.9151	2.1633

Table 22 Reconstructed image quality with errors on refinement bytes.

bpp	**PSNR (dB)**				
	No Error	**1 Error**	**2 Errors**	**3 Errors**	**4 Errors**
0.10	23.9789	23.2945	22.7415	22.6651	21.3074
0.25	27.4265	26.0182	25.0506	24.8359	22.8149
0.50	30.9458	28.3284	26.8039	26.4262	23.7052
1.00	35.8657	30.5488	28.2493	27.6683	24.2945
uncompress	58.7053	32.4296	29.1939	28.3910	24.5908

Whereas for the refinement bytes, errors that occurred onto these SPIHT coefficients did not significantly affect the quality of compressed image. Table 22 shows the quality of the reconstructed compressed image did not reduced significantly, as compared to the mapping bytes errors. The reason is the tree structure that is represented by the mapping bytes (bits) did not have error in it and reconstruction of original image was possible. With a total of 4 errors occurred onto each packet of refinement bytes, the quality of reconstruct image was still 24.5908 dB. The reconstructed original image was still visible to the naked eye without noticing on the slightly reduced quality of the reconstructed image.

In comparison between both Figure 139 and Figure 140, the compressed image data that has 4 errors occurred onto each packet of the mapping bytes and refinement bytes respectively. It can be seen that the original image could not be reconstructed as these mapping bytes carries important information on the tree structure of the original image for SPIHT coding were incorrect. However, with 4 errors only occurred onto the refinement bytes (bits), it did not affect the quality of reconstructed image since the tree structure was not affected. The refinement bytes (bits) were used to provide refinement on the reconstructed image such that to achieve higher image quality. Even without the refinement bytes (bits), the reconstruction of low quality original image is still possible to achieve. Both of these scenarios simulated the sensor nodes with errors occurred onto the compressed image data while they were transmitting the data across the wireless communication channel. Through the use of CRS coding scheme to encode the mapping bytes (bits), the original image was able to reconstruct back using the SPIHT coefficients with error protection.

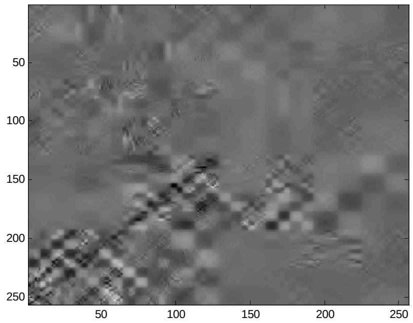


Figure 139 Reconstructed image with 4 errors on mapping bytes.

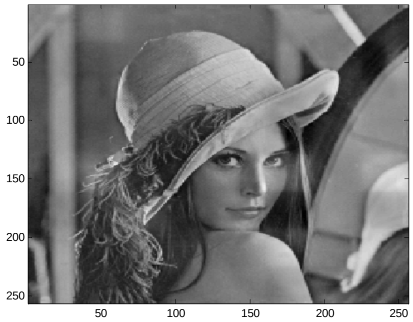


Figure 140 Reconstructed image with 4 errors on refinement bytes.

Through the use of CRS coding scheme to encode the mapping bytes (bits), the original image was able to reconstruct back using the protected SPIHT coefficients. For the CRS(20,16) coding scheme, a maximum of 4 errors occurred in each packets of mapping bytes (bits), same quality of the reconstructed compressed image was achievable, as shown in Figure 141. Nevertheless, the CRS coding scheme used could only protect a maximum of 4 errors occurred on each packets. This coding scheme will not be able to correct the incorrect SPIHT coefficients if there are more than 4 errors occurred on each packets of mapping bytes. Consequently, data transmit across a noisy wireless communication channel would then need a lower code rate CRS coding scheme.

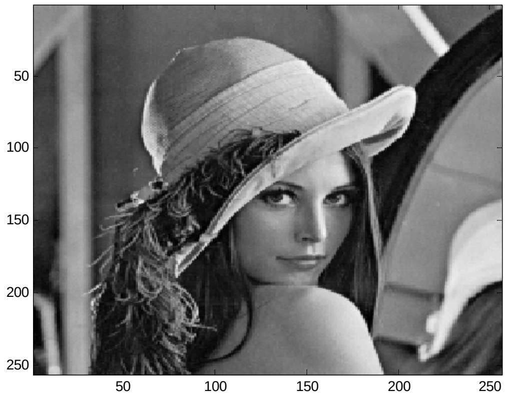


Figure 141 Reconstructed compress image without errors.

5.1.3 Hardware Utilisations: Selective SEC on SPIHT Coefficients

The proposed system was implemented onto a Celoxica RC203 board. The Celoxica RC203 board consists of a CCD camera, ZBT SRAM, Xilinx Virtex 2 FPGA, RS232 communication port and etc. The implementation imitates the visual sensor node with both SPIHT MIPS processor and CRS MISC processor. Once the image captured by the visual sensor, it will be compressed by the SPIHT MIPS processor. Once the SPIHT coefficients are produced, the mapping bytes will be encoded using the CRS(20,16) coding scheme. The compressed image data were encoded by using the CRS(20,16) coding scheme which is performed by the CRS MISC processor.

For this error correction scheme, it could tolerate to a maximum of 4 errors on each codeword of data. At the sink, the CRS decoder could recover the correct data from the remaining correctly received encoded data. Thus this will reduce the rate of retransmission of data across the wireless network and decrease the power consumption required for data transmission. The hardware implementation required a total of 5017 flip flops with 82 block RAMs. Table 23 shows the hardware usage for the implementation of both SPIHT MIPS processor and CRS MISC processor on WVSNs. For clarification, implemented visual sensor node is represented by a CCD camera, ZBT SRAM, Xilinx Virtex 2 FPGA and a Digi XBee ZB RF module. The sink (base-station) is represented by a Digi XBee ZB RF module and a computer. The usage of computer at base-station is to perform the CRS decoding once these encoded mapping bytes are received. In the later stage, reconstruction of the original image was performed from the decoded mapping bytes and refinement bytes.

Table 23 Hardware utilisation of combined SPIHT MIPS and CRS MISC architecture.

Components	**Quantity**	**Total**	**Usage**
Slices	5,041	14,336	1.08%
Flip-Flops	3,060	28,672	0.35%
4-Input LUTs	8,795	28,672	0.92%
- Logic	*7,586*	-	-
- Route-thru	*1,099*	-	-
- Dual Port RAMs	*64*	-	-
- 16x1 ROMs	*16*	-	-
- Shift Registers	*30*	-	-
Bonded IOB	188	484	26.70%
Block RAMs	82	96	6.25%
GCLKs	2	16	25.00%

5.2 LIFTING SCHEME DWT FILTER CRS MISC FOR WVSN

This Section 5.2 presents a complete processing system that performs image data compression, encryption and correction for the WVSNs. In this system, the DWT filter module first decomposed the original image into DWT coefficients to reduce the size of image data. This decreased the network bandwidth thus reducing the power consumption of sensor nodes in transmitting the image data. Then the coefficients were encrypted using the Cauchy Reed Solomon, CRS (20, 16) coding scheme to ensure data security, which is mentioned in Section 2.4.3. The CRS was used because it is a Forward Error Correction coding scheme which allowed the received encoded data to be corrected at the base-station (sink). By doing so, less retransmission was required to obtain the correct compressed data thus reducing the power consumption of sensor nodes. To perform data compression, encryption and error correction encoding in low complexity system, a CRS MISC architecture with a DWT filtering module was developed. The developed system is then implemented into a FPGA to demonstrate the feasibility of the proposed system in the WVSNs.

5.2.1 System Overview: Lifting Scheme DWT Filter CRS MISC

As shown in Figure 142, the proposed image processing system for the Wireless Visual Sensor Networks (WVSNs) consists of a CMOS camera, image memory buffer, a Discrete Wavelet Transform (DWT) filter module, a Cauchy Reed Solomon (CRS) Minimal Instruction Set Computer (MISC) processor and a Digi XBee RF module. The CMOS camera captures image with the size of 64 x 64 pixels and stores the image data into the image memory buffer. Then the DWT filter module processes the image data, producing DWT coefficients that stored into another memory buffer. Figure 142 shows how the captured image data are transmitted from the sensor nodes to the base-station.

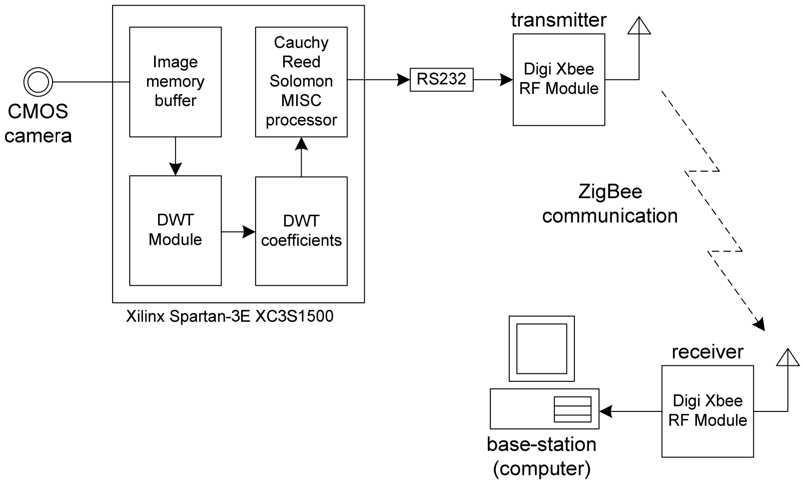


Figure 142 Proposed DWT module combined CRS MISC image processing system for WVSNs.

In order to protect these DWT coefficients from errors, the DWT coefficients are then encoded using the CRS (20,16) coding scheme. The CRS(20,16) coding scheme is capable of correcting up to 4 errors for each codeword. The reason for using the CRS (20,16) coding scheme is that it only adds a total of 4 redundant symbols on each codeword. Nevertheless, this coding scheme can correct a few errors occurred on the codeword without the need of retransmitting the image data again. Consider the case without CRS coding, the produced DWT coefficients are arranged in packet form that is shown in Figure 143. Each of the packets consists of 4 bytes of Packet Header and 16 bytes of DWT coefficients. As for the proposed system, each packet consists of CRS encoded DWT coefficients and it is arranged with a 4 bytes of Packet Header and followed by the CRS encoded codeword (20 bytes), as shown in Figure 144. The method of packetizing the CRS encoded DWT coefficients, shown in Figure 144, is the actual method performed in the hardware implementations.

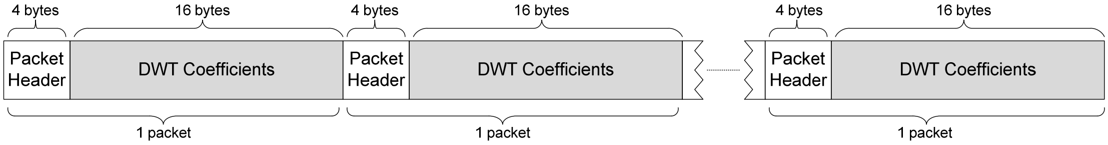


Figure 143 Packet arrangements of DWT coefficients without CRS coding.

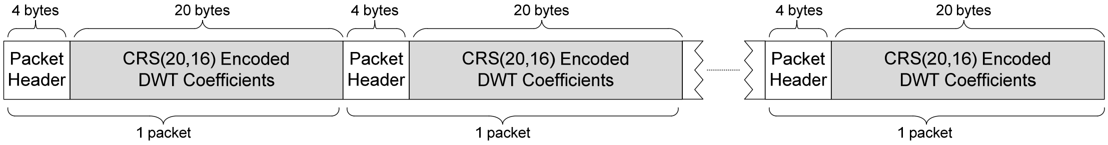


Figure 144 Packet arrangements of CRS encoded DWT coefficients.

5.2.2 Hardware Utilisations: Lifting Scheme DWT Filter CRS MISC

For the proposed system, shown in Figure 142, a hardware implementation is performed to simulate the actual like operating environment of a sensor node for the WVSNs. The implementation is performed with the use of a Celoxica RC10 board that consists of both DWT filtering module and CRS MISC described in the FPGA. The available CMOS camera, RS232 serial port and Xilinx Spartan-3L XC3S1500 FPGA are utilized in this implementation. With having both DWT module and CRS MISC are described, this occupies up to a total of 2,536 slices and 17 Block RAMs, as shown in Table 24. The Block RAMs constitutes the image memory buffer (stores captured images from CMOS camera) and DWT coefficients memory buffer (stored DWT coefficients produced by DWT module).

The hardware utilisation on the FPGA includes describing the control and operations for the CMOS camera and RS232 serial port. The RS232 serial port available on the RC10 board is connected to the Digi XBee RF transceiver. The baudrate of RS232 was set to 19200bps such that it is compatible with the data transmission rate of Digi XBee RF transceiver module at 21kbps (without security) which is stated in the datasheet [154]. This allows the CRS encoded DWT coefficients to be transmitted across wireless communication channel towards the base-station. At the base-station, there is another Digi XBee RF transceiver that receives the transmitted data from sensor nodes. In Figure 145, the Digi XBee RF transceiver placed on the left is connected to the desktop that simulates as the base-station with unlimited energy constraints and resources. On the right side, it simulates the sensor node in WVSNs, with the integration of an onboard CMOS camera, memory buffer, DWT filter module, CRS MISC and Digi XBee RF transceiver. Figure 146 shows the reconstructed image (captured at sensor node) and the corresponding DWT coefficients decoded.

Table 24 Hardware utilisation of the proposed DWT module and CRS MISC system for WVSNs.

Components	**Quantity**	**Total**	**Usage**
Slices	2,552	13,312	1.08%
Flip-Flops	1,440	26,624	0.35%
4-Input LUTs	4,403	26,624	0.92%
- Logic	*3,388*	-	-
- Route-thru	*412*	-	-
- Dual Port RAMs	*60*	-	-
- Shift Registers	*543*	-	-
Bonded IOB	35	221	26.70%
Block RAMs	17	32	6.25%
DCMs	2	8	25.00%

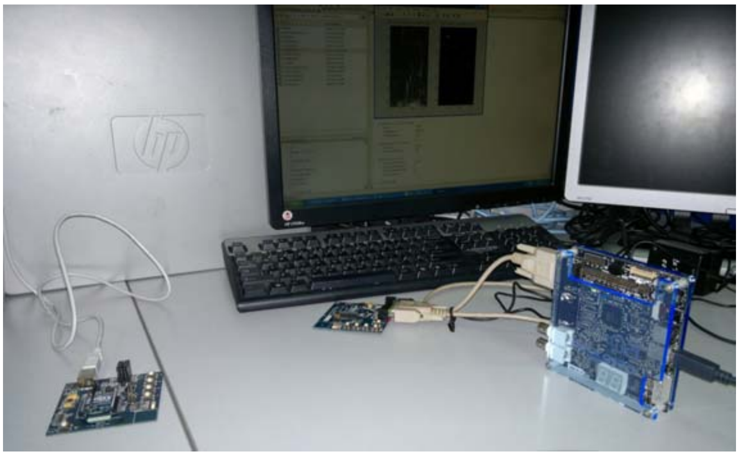


Figure 145 Single hop hardware simulation on the proposed system.

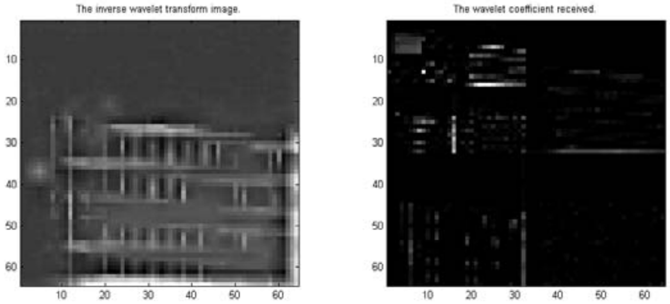


Figure 146 Reconstructed original image captured at sensor node.

5.3 DWT CRS MISC FOR WVSN

In this Section 5.3, the developed new low complexity DWT CRS MISC architecture that performs data compression, data encryption and data correction in a single architecture was integrated into the WVSNs. With the CRS encoded data, the base-station can correct a small number of errors that occurred onto the received image data and requires less retransmission of the data. A complete sensor node system with DWT CRS MISC architecture was developed and then implemented into a Field Programmable Gate Array (FPGA) to demonstrate the feasibility for use in the WVSNs.

5.3.1 System Overview: DWT CRS MISC

The proposed system consists of a CMOS camera which captures image (64 x 64 pixels) and stores the image data into the image memory buffer. Then it is followed by the DWT CRS MISC processor (mentioned in Chapter 3) performs DWT compression and CRS encoding in a single architecture. The reason to consider DWT Haar wavelet filtering method is the compression method used by Haar is less complex compared to other wavelet filtering methods [141]. The CRS coding scheme is chosen because it provides both data security and data reliability for the encoded data [20]. With data encoded, the base-station can correct a small amount of errors without the need to request for retransmission of data from the sensor nodes. At the same time, these CRS encoded data are also encrypted which prevents adversary from eavesdropping on the data. Therefore, the sensor node requires less amount of energy to perform data transmission and thus extending its operating lifespan. In Figure 147, it shows the proposed system integrated with DWT CRS MISC architecture in WVSNs.

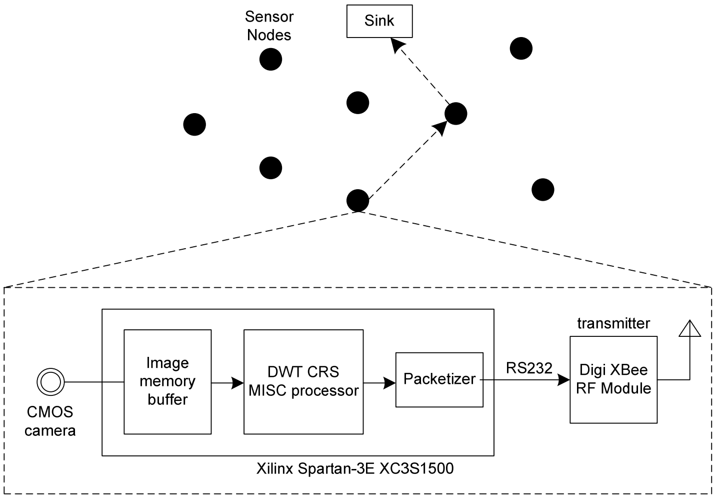


Figure 147 Proposed system with DWT CRS MISC architecture for WVSNs.

In the proposed system, the MISC processes an image part by part. It takes 4 rows of image (64 x 4 pixels) to process at a particular time until the whole image is completely processed. This is to avoid large memory spaces needed to store and process the whole image (64 x 64 pixels) in the MISC architecture. The MISC processes and encodes each part of image data (64 x 4 pixels) to produce 4 CRS encoded codewords. After the data are encoded, a packetizer packetizes the encoded data (codewords) by adding a Packet Header (4 bytes) to the front of each codewords, as shown in Figure 148. Once a packet of data is packetized, the data will be transmitted to the base-station through the Digi XBee RF transceiver modules.

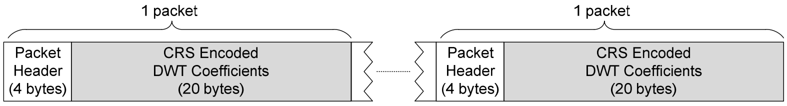

1 packet
1 packet
Packet Header (4 bytes)
CRS Encoded DWT Coefficients (20 bytes)
Packet Header (4 bytes)
CRS Encoded DWT Coefficients (20 bytes)

Figure 148 Transmission of packet for DWT CRS MISC processor encoded image data.

5.3.2 Hardware Utilisations: DWT CRS MISC

A hardware implementation for the proposed new DWT CRS MISC architecture is performed onto the FPGA, to simulate the actual like operating environment of a sensor node in WVSNs. For this reason, a Celoxica RC10 board (consists of CMOS camera, RS232 serial port and Xilinx Spartan-3L XC3S1500 FPGA) was utilized for such implementation. The hardware realization on the FPGA also included the hardware description of controls/operations for the CMOS camera, DWT CRS MISC architecture and RS232 serial port. A Digi XBee RF transceiver module was connected to the RS232 serial port such that it transmits the CRS encoded compressed image data to the base station. The baudrate of the RS232 was set to 19200bps such that the data sent out from the FPGA matched the data transmission rate of the Digi XBee RF transceiver modules. For this hardware simulation, the camera sensor node (end device) was placed at one end of the room which is shown in Figure 149.

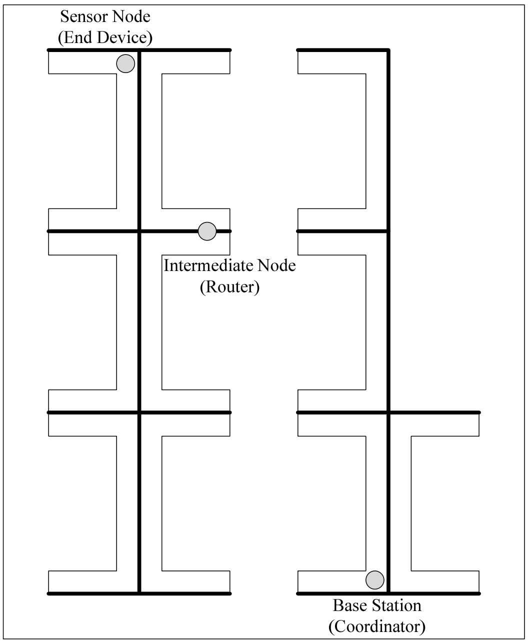
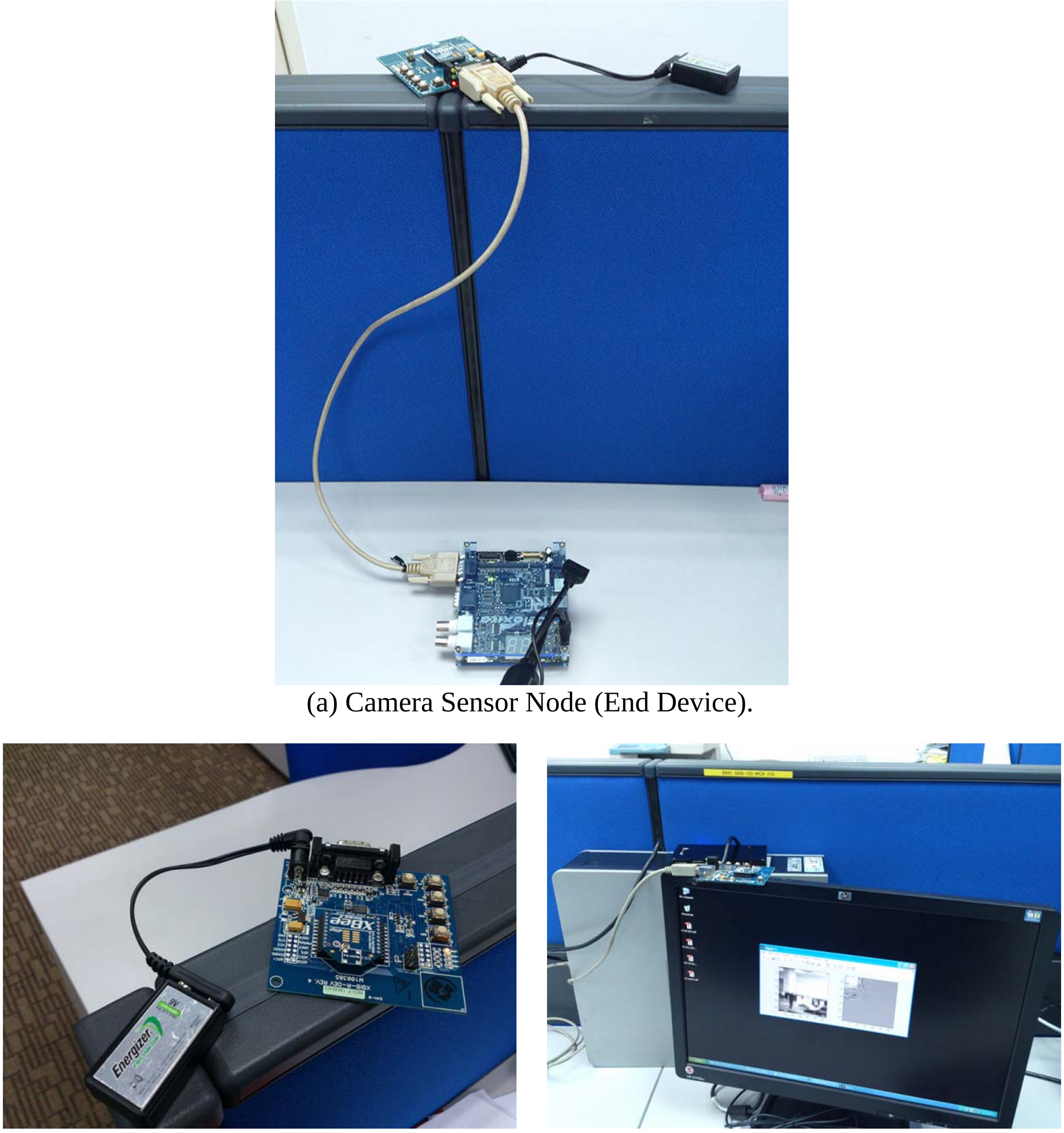


Figure 149 Arrangements of sensor node, intermediate node and base station inside a room.

At the base station, the computer that was connected to another Digi XBee RF transceiver module performed CRS decoding on the data received from the sensor node. Once the CRS decoding process was completed, reconstruction of image was performed by using the obtained Level 2 DWT coefficients. As shown in Figure 149, a base station was placed at another end of room that received the data transmitted from the sensor node. An intermediate node was placed in between the sensor node (end device) and the base station (coordinator) such that it relay the transmitted data from sensor node to the base station. The actual hardware implementation for sensor node (end device), intermediate node (router) and base station (coordinator), is shown in Figure 150.



(a) Camera Sensor Node (End Device).

(b) Intermediate Node (Router).

(c) Base Station (Coordinator).

Figure 150 Hardware simulation for the proposed system with DWT CRS MISC architecture.

In Table 25, it shows the hardware implementation of the proposed system requires a total of 1065 slices and 5 Block RAMs. The Block RAMs constitutes the image memory buffer which stores the image data captured from CMOS camera. Subsequently, a Digi XBee RF transceiver was connected directly to the RS232 serial port. This allows the CRS encoded DWT coefficients to be transferred across the wireless communication network towards the base-station (sink).

Table 25 Hardware utilisation of the proposed system with DWT CRS MISC for WVSNs

Component	**Quantity**	**Total**	**Usage**
Slices	1,065	13,312	8.0%
Flip-Flops	729	26,624	2.7%
4 input LUTs	1,611	26,624	6.1%
- Logic	*1,421*	-	-
- Route-thru	*117*	-	-
- Dual Port Rams	*60*	-	-
- Shift registers	*13*	-	-
Bonded IOB	35	221	15.8%
Block RAMs	5	32	15.6%
BUFGMUXs	5	8	62.5%
DCMs	2	4	50.0%

At the base-station, a desktop computer is connected to Digi XBee RF transceiver to receive the transmitted encoded data. This simulates a base-station with unlimited energy constraints and resources. Once the base-station receives the transmitted data, it proceeds to decode the received data by recovering the DWT coefficients and then reconstruct the image. For this implementation, only the Level 2 DWT coefficients (LL_2, LH_2, HL_2, HH_2) were encoded and transmitted to the base-station. Consequently, there were less amount of image data (compared to full image data size) needed to be transferred across the wireless network thus reducing the power consumption of sensor node used in data transmission. Figure 151 shows the reconstructed image using only the Level 2 DWT coefficients.

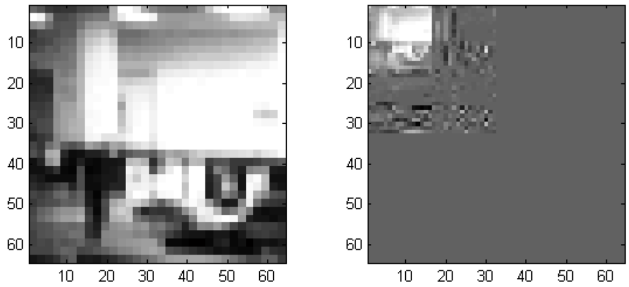


Figure 151 Reconstructed image with Level 2 DWT Coefficients.

5.4 SUMMARY

Initially, the combined SPIHT MIPS processor and CRS MISC processor to form a sensor node image processing system was developed to achieve the following objectives, which are to compress the image data, provide security protection and data reliability for transmitting image data in the WVSNs. By referring to Table 26, the hardware implementation of the SPIHT MIPS and CRS MISC sensor node system required 5,036 Slices (3,058 Flip-Flops, 8,794 LUTs, 82 Block RAMs).

To further improve (less) on the hardware utilisations of developed sensor node system, the SPIHT MIPS was replaced by a DWT filter module to reduce the image data. This resulted in 49.3% reduction in number of Slices required as compared to the SPIHT MIPS for reducing the image data. For Xilinx Virtex-II FPGA, each Slices contains of 2 4-Input LUTs and 2 Flip-Flops [156]. Whereas, each Slices of the Xilinx Spartan-3L is made up of 2 4-Input LUTs and 2 Flip-Flops [151]. Therefore, the comparison using number of Slices can be made since both different technology of FPGAs have the same number of LUTs and Flip-Flops for one Slices.

Subsequently, a further 58.3% reduction in number of hardware utilisations (in terms of Slices) of the sensor node image processing system was achieved by removing the DWT filter module. Instead of using the DWT filter module, the developed DWT CRS MISC architecture was incorporated with the ability of performing a 2-D DWT image compression as well into the initially developed CRS MISC architecture. As a result, the developed DWT CRS MISC can perform both DWT image compression and CRS encoding by reusing the available hardware in the architecture. Table 26 shows the hardware utilisations of the developed sensor node image processing system in FPGAs for use in the WVSNs.

Besides, the estimated FPGA power consumption of the developed sensor node systems were determined using the Xilinx xPower Analyzer 11.5. The estimated power consumption of DWT filter module with CRS MISC sensor nodes system would be 552.0mW. Since there is a 58.3% reduction in hardware utilisations for the DWT CRS MISC, a 57.7% decrease in the estimated power consumption was expected. The power consumption for combined SPIHT MIPS and CRS MISC sensor node system was not able to be determined as software errors occurred while the system was analysed. Even different versions of the Xilinx xPower Analyzer (8.2, 9.2, 10.1, 11.5) were tried but the software still encounters errors. However, it would be expected to be high since there is large amount of Slices required for its implementation.

Table 26 Hardware utilisations of the developed systems for WVSNs.

System Design	Slices	Flip-Flops	4-Input LUTs	Block RAMs	Power (mw)
Xilinx Virtex-II					
SPIHT MIPS + CRS MISC	5,036	3,058	8,794	82	N/A
Xilinx Spartan-3L					
DWT Filter + CRS MISC	2,552	1,440	4,403	17	552.0
DWT CRS MISC	1,065	729	1,611	5	233.4

Therefore, a low complexity new DWT CRS MISC sensor node system was developed which reduces the image data, offers similar security protection level as compared to AES and provides data reliability that is offered by traditional Reed Solomon coding scheme. With the reduce number of required Slices, the power consumption of the developed system can be subsequently reduced. As compared to other existing system (ECBC, DWT module combined CRS MISC and SPIHT CRS MISC), this new low complexity DWT CRS MISC sensor node image processing system is much suitable to be used in the resource constraint WVSNs.

CHAPTER 6

CONCLUSIONS AND FUTURE WORKS

In this research, several joint compression, encryption and forward error correction processing frameworks for use in WVSNs were presented. The presented processing frameworks have the capabilities to reduce large amount of image data, provide data security and reliable data transmission. The presented processing frameworks consist of two separate modules that were combined together to perform image compression and CRS encoding. Rather than using separate modules, a novel DWT CRS MISC architecture was developed which performs all the aforementioned capabilities in a single architecture. This would reduce the amount of hardware utilisations required to realize the processing framework that perform all these tasks. At the same time, the power consumption of the processing framework was also reduced when achieving less amount of hardware utilisations. The evidences on the improvement of hardware utilisations for using the DWT CRS MISC architecture as the processing framework are shown in the Section 5.4.

Behavioral and Post & Route simulations were performed onto the DWT CRS MISC architecture before the MISC architecture was implemented as the processing framework for WVSNs. From these simulations, the generated waveforms showed that the developed DWT CRS MISC architecture does operate and function according to the design specifications. Based on the generated Behavioral simulation waveforms, it was found that these waveforms for the control signals combinational logic circuit and the execution of programme instructions in DWT CRS MISC architecture do comply with the design specifications. Meanwhile, the Post & Route simulation waveforms provided the information on the hardware delay encountered in the DWT CRS MISC architecture. With the longest delay (32.732ns) determined, the operating frequency of the DWT CRS MISC architecture was set to operate at 24MHz

(41.667ns) such that it could cope with the hardware delays. Therefore, the DWT CRS MISC would require 0.1915s to completely process an image of size 64 pixels x 64 pixels.

Simulation result also shows an acceptable quality of the reconstructed image with the selected number of DWT coefficients to be transmitted. Consequently, the new DWT CRS MISC reduces the image data, offers similar security protection level as compared to AES and provides data reliability that is offered by traditional Reed Solomon coding scheme. In comparison to other existing systems (ECBC, DWT module combined CRS MISC and SPIHT MIPS combined CRS MISC), this new low complexity DWT CRS MISC architecture is much suitable to be used as the joint schemes processing framework in the resource constraint WVSNs.

6.1 FUTURE WORKS

In future, actual hardware implementation of the developed novel DWT CRS MISC architecture can be performed onto the Xilinx Spartan-6 FPGA. This is to achieve an even lower hardware complexity and lower power consumption of the image processing framework for the resource constraint WVSNs. With the use of Spartan-6 FPGA, the operating lifespan of the visual sensor nodes could be extended thus providing more surveillance information to the base station.

The developed novel image processing framework for WVSNs is considered to be in development stage and the actual Integrate Circuit (IC) chip for this processing framework was not developed. Consequently, further improvement in power consumption can be achieved when the developed image processing framework is implemented into Application Specific Integrated Circuits (ASIC). Further reduction in power consumption may be achieved as further optimisations will be made onto the integrated circuit of the image processing framework.

Another future work to consider is the development of decoder for use in the WVSNs. The decoder will be integrated with the intermediate nodes such that error correction can be performed if there is any errors occurred onto the transmitted data. This is to increase the data reliability in the intermediate nodes by reducing the number errors occurred onto the transmitted image data before they are received at the base station. Therefore, the base station will be able to correct the number of errors that occurred in between the maximum allowable of errors that the code scheme can correct. This is considered as future work because the decoder requires a complex algorithm in determining the number of errors, error locations and the correct values to replace the errors. As such, these are a few future works that are recommended in the thesis to further improve image data transmission the resource constrained WVSNs.

REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks,* vol. 38, pp. 393-422, 2002.

[2] I. F. Akyildiz, S. Weilian, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *Communications Magazine, IEEE,* vol. 40, pp. 102-114, 2002.

[3] S. Soro and W. Heinzelman, "A Survey of Visual Sensor Networks," *Advances in Multimedia,* vol. 2009, pp. 1-22, 2009.

[4] B. Tavli, K. Bicakci, R. Zilan, and J. Barcelo-Ordinas, "A Survey of Visual Sensor Network Platforms," *Multimedia Tools and Applications,* vol. 60, pp. 689-726, 2012/10/01 2012.

[5] Y. Charfi, N. Wakamiya, and M. Murata, "Challenging Issues in Visual Sensor Networks," *Wireless Communications, IEEE,* vol. 16, pp. 44-49, 2009.

[6] I. F. Akyildiz, T. Melodia, and K. R. Chowdury, "Wireless Multimedia Sensor Networks: A Survey," *Wireless Communications, IEEE,* vol. 14, pp. 32-39, 2007.

[7] V. Raghunathan, C. Schurgers, P. Sung, and M. B. Srivastava, "Energy-aware wireless microsensor networks," *Signal Processing Magazine, IEEE,* vol. 19, pp. 40-50, 2002.

[8] D. Schmidt, M. Berning, and N. Wehn, "Error Correction in Single-Hop Wireless Sensor Networks - A Case Study," in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09.*, 2009, pp. 1296-1301.

[9] M. A. Razzaque, C. Bleakley, and S. Dobson, "Compression in Wireless Sensor Networks: A Survey and Comparative Evaluation," *ACM Trans. Sen. Netw.,* vol. 10, pp. 1-44, 2013.

[10] M. A. Simplicio, B. T. de Oliveira, P. S. L. M. Barreto, C. B. Margi, T. C. M. B. Carvalho, and M. Naslund, "Comparison of Authenticated-Encryption schemes in Wireless Sensor Networks," in *Local Computer Networks (LCN), 2011 IEEE 36th Conference on*, 2011, pp. 450-457.

[11] G. J. Pottie and W. J. Kaiser, "Wireless integrated network sensors," *Commun. ACM,* vol. 43, pp. 51-58, 2000.

[12] C. M. Sadler and M. Martonosi, "Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks," presented at the Proceedings of the 4th international conference on Embedded networked sensor systems, Boulder, Colorado, USA, 2006.

[13] T. Mancill and O. Pilskalns, "Combining Encryption and Compression in Wireless Sensor Networks," *International Journal of Wireless Information Networks,* vol. 18, pp. 39-49, 2011/03/01 2011.

[14] M. Mutschlechner, B. Li, R. Kapitza, and F. Dressler, "Using Erasure Codes to overcome reliability issues in energy-constrained sensor networks," in *Wireless On-demand Network Systems and Services (WONS), 2014 11th Annual Conference on*, 2014, pp. 41-48.

[15] I. F. Akyildiz, T. Melodia, and K. R. Chowdhury, "A Survey on Wireless Multimedia Sensor Networks," *Computer Networks,* vol. 51, pp. 921-960, 2007.

[16] F. Mavaddat and B. Parhami, "URISC: the ultimate reduced instruction set computer," *International Journal of Electrical Engineering Education,* vol. 25, pp. 327-334, 1988.

[17] S. Lin and D. Costello, *Error Control Coding (2nd Edition)*: Prentice Hall, 2004.

[18] D. W. Jones, "The ultimate RISC," *SIGARCH Comput. Archit. News,* vol. 16, pp. 48-55, 1988.

[19] W. Sweldens, "The Lifting Scheme: A Construction of Second Generation Wavelets," *SIAM Journal on Mathematical Analysis,* vol. 29, pp. 511-546, 1998.

[20] J. Tian, Z. Yang, and Y. Dai, "SEC: A Practical Secure Erasure Coding Scheme for Peer-to-Peer Storage System," in *14th Symposium on Storage System and Technology*, 2006.

[21] O. Adamo and M. R. Varanasi, "Joint Scheme for Physical Layer Error Correction and Security," *ISRN Communications and Networking,* vol. 2011, 2011.

[22] Spartan-3E FPGA Family Data Sheet, Last Accessed: 10/12/2014, Available: http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

[23] J. J. Ong, L. M. Ang, and K. P. Seng, "Selective Secure Error Correction on SPIHT Coefficients for Pervasive Wireless Visual Network," *International Journal of Ad Hoc and Ubiquitous Computing,* vol. 13, pp. 73-82, 2013.

[24] H. Cam, "Secure Data Aggregation and Source-Channel Coding with MIT Code for Wireless Sensor Networks," in *Performance, Computing, and Communications Conference, 2005. IPCCC 2005. 24th IEEE International*, 2005, pp. 377-382.

[25] M. L. Kaddachi, A. Soudani, V. Lecuire, K. Torki, L. Makkaoui, and J.-M. Moureaux, "Low Power Hardware-Based Image Compression Solution for Wireless Camera Sensor Networks," *Computer Standards & Interfaces,* vol. 34, pp. 14-23, 2012.

[26] M. Weeks and M. Bayoumi, "Discrete Wavelet Transform: Architectures, Design and Performance Issues," *J. VLSI Signal Process. Syst.,* vol. 35, pp. 155-178, 2003.

[27] S. Gnavi, B. Penna, M. Grangetto, E. Magli, and G. Olmo, "Wavelet kernels on a DSP: a comparison between lifting and filter banks for image coding," *EURASIP J. Appl. Signal Process.,* vol. 2002, pp. 981-989, 2002.

[28] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using wavelet transform," *Image Processing, IEEE Transactions on,* vol. 1, pp. 205-220, 1992.

[29] D. Lazar and A. Averbuch, "Wavelet-based video coder via bit allocation," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 11, pp. 815-832, 2001.

[30] C. Gargour, M. Gabrea, V. Ramachandran, and J. M. Lina, "A Short Introduction to Wavelets and Their Applications," *Circuits and Systems Magazine, IEEE,* vol. 9, pp. 57-68, 2009.

[31] D. L. Fugal, *Conceptual Wavelets in Digital Signal Processing: An In-depth, Practical Approach for the Non-mathematician*: Space & Signals Technical Pub., 2009.

[32] P. Mather and B. Tso, *Classification Methods for Remotely Sensed Data, Second Edition*: Taylor & Francis, 2009, pp. 235-239.

[33] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *Journal of Fourier Analysis and Applications,* vol. 4, pp. 247-269, 1998.

[34] P. Salama and B. King, "Efficient secure image transmission: compression integrated with encryption," 2005, pp. 47-58.

[35] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *Signal Processing, IEEE Transactions on,* vol. 41, pp. 3445-3462, 1993.

[36] S. Assegie, P. Salama, and B. King, *An Attack on Wavelet Tree Shuffling Encryption Schemes* vol. 78: Springer Berlin Heidelberg, 2010, pp. 139-148.

[37] R. Candela, E. Romano, and P. Romano, "A combined CWT-DWT method using model-based design simulator for partial discharges online detection," in *Electrical Insulation and Dielectric Phenomena, 2009. CEIDP '09. IEEE Conference on*, 2009, pp. 429-432.

[38] R. P. Desale and S. V. Verma, "Study and analysis of PCA, DCT & DWT based image fusion techniques," in *Signal Processing Image Processing & Pattern Recognition (ICSIPR), 2013 International Conference on*, 2013, pp. 66-69.

[39] A. J. Casson and E. Rodriguez-Villegas, "A 60 pW gmC Continuous Wavelet Transform Circuit for Portable EEG Systems," *Solid-State Circuits, IEEE Journal of,* vol. 46, pp. 1406-1415, 2011.

[40] H. Chaouch, K. Ouni, and L. Nabli, "Segmenting and supervising an ECG signal by combining the CWT & PCA," *IJCSI International Journal of Computer Science Issues,* vol. 9, pp. 433-440, 2012.

[41] C. Li-Fang, C. Tung-Chien, and C. Liang-Gee, "Architecture design of the multi-functional wavelet-based ECG microprocessor for realtime detection of abnormal cardiac events," in *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, 2012, pp. 4466-4469.

[42] J.-P. Antoine, "The continuous wavelet transform in image processing," *CWI Q,* vol. 11, pp. 323-345, 1998.

[43] D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," *SIGARCH Comput. Archit. News,* vol. 8, pp. 25-33, 1980.

[44] W. F. Gilreath and P. A. Laplante, *Computer Architecture*: Kluwer Academic Publishers, 2003.

[45] D. Patterson and S. Seccombe, "Complex versus reduced instruction set computers," in *Solid-State Circuits Conference. Digest of Technical Papers. 1983 IEEE International*, 1983, pp. 218-219.

[46] P. A. Laplante and W. Gilreath, "Single Instruction Set Architectures for Image Processing," Boston, MA, USA, 2002, pp. 20-29.

[47] P. Frenger, "The Ultimate RISC: A zero-instruction computer," *SIGPLAN Not.,* vol. 35, pp. 17-24, 2000.

[48] W. Van Der Poel, "A simple electronic digital computer," *Applied Scientific Research, Section B,* vol. 2, pp. 367-400, 1952.

[49] P. A. Laplante, "An improved conditional branching scheme for a single instruction computer architecture," *SIGARCH Comput. Archit. News,* vol. 19, pp. 66-68, 1991.

[50] P. A. Laplante, "A novel single instruction computer architecture," *SIGARCH Comput. Archit. News,* vol. 18, pp. 22-26, 1990.

[51] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling Ultra-Low Power Wireless Research," in *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, 2005, pp. 364-369.

[52] IEEE 802.15.4-2011 Standard, Last Accessed: 27/10/2014, Available: http://standards.ieee.org/getieee802/download/802.15.4-2011.pdf

[53] M. Rahimi, R. Baer, O. I. Iroezi, J. C. Garcia, J. Warrior, D. Estrin, and M. Srivastava, "Cyclops: In Situ Image Sensing and Interpretation in Wireless Sensor Networks," presented at the Proceedings of the 3rd international conference on Embedded networked sensor systems, San Diego, California, USA, 2005.

[54] L. Nachman, R. Kling, R. Adler, J. Huang, and V. Hummel, "The Intel Mote Platform: A Bluetooth-Based Sensor Network for Industrial Monitoring," presented at the Proceedings of the 4th international symposium on Information processing in sensor networks, Los Angeles, California, 2005.

[55] W.-C. Feng, E. Kaiser, W. C. Feng, and M. L. Baillif, "Panoptes: Scalable Low-Power Video Sensor Networking Technologies," *ACM Trans. Multimedia Comput. Commun. Appl.,* vol. 1, pp. 151-167, 2005.

[56] I. Downes, L. B. Rad, and H. Aghajan, "Development of a Mote for Wireless Image Sensor Networks," in *Proc. of COGnitive systems with Interactive Sensors (COGIS),* Paris, France, 2006.

[57] P. Chen, P. Ahammad, C. Boyer, I. H. Shih, L. Leon, E. Lobaton, M. Meingast, O. Songhwai, S. Wang, Y. Posu, A. Y. Yang, Y. Chuohao, C. Lung-Chung, J. D. Tygar, and S. S. Sastry, "CITRIC: A Low-Bandwidth Wireless Camera Network Platform," in *Distributed Smart Cameras, 2008. ICDSC 2008. Second ACM/IEEE International Conference on*, 2008, pp. 1-10.

[58] Specification of the Advanced Encryption Standard, Last Accessed, Available:

[59] J. S. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications," in *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, 2006, pp. 173-180.

[60] T. A. Welch, "A Technique for High-Performance Data Compression," *Computer,* vol. 17, pp. 8-19, 1984.

[61] Q. Lu, W. Luo, J. Wang, and B. Chen, "Low-Complexity and Energy Efficient Image Compression Scheme for Wireless Sensor Networks," *Computer Networks,* vol. 52, pp. 2594-2603, 2008.

[62] M. Pellenz, R. Souza, and M. Fonseca, "Error Control Coding in Wireless Sensor Networks," *Telecommunication Systems,* vol. 44, pp. 61-68, 2010/06/01 2010.

[63] P. Boluk, S. Baydere, and A. Harmanci, "Robust Image Transmission Over Wireless Sensor Networks," *Mobile Networks and Applications,* vol. 16, pp. 149-170, 2011.

[64] C. Li Wern, A. Li-Minn, and S. Kah Phooi, "Reduced Memory SPIHT Coding Using Wavelet Transform with Post-Processing," in *Intelligent Human-Machine Systems and Cybernetics, 2009. IHMSC '09. International Conference on*, 2009, pp. 371-374.

[65] L. W. Chew, W. C. Chia, L.-M. Ang, and K. P. Seng, "Very Low-Memory Wavelet Compression Architecture using Strip-Based Processing for Implementation in Wireless Sensor Networks," *EURASIP J. Embedded Syst.,* vol. 2009, pp. 1-1, 2009.

[66] Z. Chao Hu, P. Liu Yingzi, Z. Zhenxing, and M. Q. H. Meng, "A Novel FPGA-Based Wireless Vision Sensor Node," in *Automation and Logistics, 2009. ICAL '09. IEEE International Conference on*, 2009, pp. 841-846.

[67] S. Rein and M. Reisslein, "Low-Memory Wavelet Transforms for Wireless Sensor Networks: A Tutorial," *Communications Surveys & Tutorials, IEEE,* vol. 13, pp. 291-307, 2011.

[68] D. G. Costa and L. A. Guedes, "A Discrete Wavelet Transform (DWT)-Based Energy-Efficient Selective Retransmission Mechanism for Wireless Image Sensor Networks," *Journal of Sensor and Actuator Networks,* vol. 1, pp. 3-35, 2012.

[69] E. Berlekamp, R. Peile, and S. Pope, "The application of error control to communications," *Communications Magazine, IEEE,* vol. 25, pp. 44-57, 1987.

[70] G. M. Almeida, E. A. Bezerra, L. V. Cargnini, R. D. R. Fagundes, and D. G. Mesquita, "A Reed-Solomon Algorithm for FPGA Area Optimization in Space Applications," in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, 2007, pp. 243-249.

[71] R. H. Morelos-Zaragoza, *The Art of Error Correcting Coding*: John Wiley, 2006.

[72] B. A. Forouzan and S. C. Fegan, *Data Communications and Networking*: McGraw-Hill, 2007.

[73] G. Wade, *Coding Techniques: An Introduction to Compression and Error Control*: Palgrave, 2000.

[74] S. B. Wicker, *Error control systems for digital communication and storage*: Prentice Hall, 1995.

[75] C. Paar, P. Fleischmann, and P. Roeise, "Efficient multiplier architectures for Galois fields GF(2^{4n})," *Computers, IEEE Transactions on,* vol. 47, pp. 162-170, 1998.

[76] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics,* vol. 8, pp. 300-304, 1960.

[77] P. Sweeney, *Error Control Coding: From Theory to Practice*: Wiley, 2002.

[78] J. Jeong and C. T. Ee, "Forward Error Correction in Sensor Networks," *University of California at Berkeley,* 2003.

[79] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Their Applications*: John Wiley & Sons, 1999.

[80] M. A. Khan, S. Afzal, and R. Manzoor, "Hardware Implementation of Shortened (48,38) Reed Solomon Forward Error Correcting Code," in *Multi Topic Conference, 2003. INMIC 2003. 7th International*, 2003, pp. 90-95.

[81] M. R. Islam, "Error Correction Codes in Wireless Sensor Network: An Energy Aware Approach," *World Academy of Science, Engineering and Technology,* vol. 37, pp. 627-632, 2010.

[82] S. S. Shah, S. Yaqub, and F. Suleman, "Self-correcting codes conquer noise Part 2: Reed-Solomon codecs," *EDN,* vol. 46, pp. 107-120, 2001.

[83] R. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal,* vol. 26, pp. 147-160, 1950.

[84] J. A. S. Azaleah Amina P. Chio, Delfin Jay M. Sabido, "VLSI Implementation of a (255,223) Reed Solomon Error Correction Codec," in *2nd National ECE Conference*, 2001.

[85] K. Sukun, R. Fonseca, and D. Culler, "Reliable Transfer on Wireless Sensor Networks," in *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, 2004, pp. 449-459.

[86] K. Zeinab Hajjarian and S. Mohsen, "Channel Coding in Multi-hop Wireless Sensor Networks," in *ITS Telecommunications Proceedings, 2006 6th International Conference on*, 2006, pp. 965-968.

[87] D. F. W. Yap, S. K. Tiong, J. Koh, D. P. Andito, K. C. Lim, and W. K. Yeo, "Link Performance Enhancement for Image Transmission with FEC in Wireless Sensor Networks," *Journal of Applied Sciences,* vol. 12, pp. 1465-1473, 2012.

[88] T. Yamazato, H. Okada, M. Katayama, and A. Ogawa, "A Simple Data Relay Process and Turbo Code Application to Wireless Sensor Networks," in *Wireless Communication Systems, 2004, 1st International Symposium on*, 2004, pp. 398-402.

[89] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *Communications, 1993. ICC 93. Geneva. Technical Program, Conference Record, IEEE International Conference on*, 1993, pp. 1064-1070 vol.2.

[90] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: turbo-codes," *Communications, IEEE Transactions on,* vol. 44, pp. 1261-1271, 1996.

[91] N. Abughalieh, K. Steenhaut, and A. Nowe, "Low Power Channel Coding for Wireless Sensor Networks," in *Communications and Vehicular Technology in the Benelux (SCVT), 2010 17th IEEE Symposium on*, 2010, pp. 1-5.

[92] J. Chen and A. Abedi, "Distributed Turbo Coding and Decoding for Wireless Sensor Networks," *Communications Letters, IEEE,* vol. 15, pp. 166-168, 2011.

[93] A. Brokalakis and I. Papaefstathiou, "Using Hardware-Based Forward Error Correction to Reduce the Overall Energy Consumption of WSNs," in *Wireless Communications and Networking Conference (WCNC), 2012 IEEE*, 2012, pp. 2191-2196.

[94] P. K. R. Junga, M. A. Abdelrahman, and C. Thurman, "Algorithms for Reliable Data Transmission for Metal Fill Monitoring using Wireless Sensor Networks," in *Southeastcon, 2008. IEEE*, 2008, pp. 7-14.

[95] M. Y. Naderi, H. R. Rabiee, and M. Khansari, "Performance Analysis of Selected Error Control Protocols in Wireless Multimedia Sensor Networks," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, 2010, pp. 447-450.

[96] N. A. Ali, H. M. ElSayed, M. El-Soudani, and H. H. Amer, "Effect of Hamming Voding on WSN Lifetime and Throughput," in *Mechatronics (ICM), 2011 IEEE International Conference on*, 2011, pp. 749-754.

[97] G. Gur, Y. Altug, E. Anarim, and F. Alagoz, "Image Error Concealment using Watermarking with Subbands for Wireless Channels," *Communications Letters, IEEE,* vol. 11, pp. 179-181, 2007.

[98] D. U. Lee, W. Luk, C. Wang, and C. Jones, "A Flexible Hardware Encoder for Low-Density Parity-Check Codes," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, 2004, pp. 101-111.

[99] J. S. Rahhal, "LDPC Coding for MIMO Wireless Sensor Networks with Clustering," in *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on*, 2012, pp. 58-61.

[100] R. Gallager, "Low-density parity-check codes," *Information Theory, IRE Transactions on,* vol. 8, pp. 21-28, 1962.

[101] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler, "SPINS: security protocols for sensor networks," *Wirel. Netw.,* vol. 8, pp. 521-534, 2002.

[102] C. Karlof, N. Sastry, and D. Wagner, "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks," presented at the Proceedings of the 2nd international conference on Embedded networked sensor systems, Baltimore, MD, USA, 2004.

[103] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen, "AES Implementation on a Grain of Sand," *Information Security, IEE Proceedings,* vol. 152, pp. 13-20, 2005.

[104] Y. Chen, X. Zou, Z. Liu, and Y. Han, "Secure AES Coprocessor Against Power Analysis for Wireless Sensor Networks," in *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*, 2007, pp. 2234-2237.

[105] S. Didla, A. Ault, and S. Bagchi, "Optimizing AES for Embedded Devices and Wireless Sensor Networks," presented at the Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities, Innsbruck, Austria, 2008.

[106] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B. S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee, "HIGHT: A New Block Cipher Suitable for Low-Resource Device," vol. 4249 LNCS, ed. Yokohama, 2006, pp. 46-59.

[107] K. Woo Kwon, L. Hwaseong, K. Yong Ho, and L. Dong Hoon, "Implementation and Analysis of New Lightweight Cryptographic Algorithm Suitable for Wireless Sensor Networks," in *Information Security and Assurance, 2008. ISA 2008. International Conference on*, 2008, pp. 73-76.

[108] M. Luk, G. Mezzour, A. Perrig, and V. Gligor, "MiniSec: A Secure Sensor Network Communication Architecture," presented at the Proceedings of the 6th international conference on Information processing in sensor networks, Cambridge, Massachusetts, USA, 2007.

[109] A. Lenstra and E. Verheul, *Selecting Cryptographic Key Sizes* vol. 1751: Springer Berlin Heidelberg, 2000, pp. 446-465.

[110] L. An and N. Peng, "TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks," in *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, 2008, pp. 245-256.

[111] P. Kitsos, G. Selimis, O. Koufopavlou, and A. N. Skodras, "A Hardware Implementation of CURUPIRA Block Cipher for Wireless Sensors," in *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, 2008, pp. 850-853.

[112] M. Simplicio Jr, P. S. Barreto, T. C. Carvalho, C. B. Margi, and M. Näslund, "The CURUPIRA-2 Block Cipher for Constrained Platforms: Specification and Benchmarking," 2008.

[113] A. Fiat and M. Naor, *Broadcast Encryption* vol. 773: Springer Berlin Heidelberg, 1994, pp. 480-491.

[114] C. Yanli and Y. Geng, "An Efficient Broadcast Encryption Scheme for Wireless Sensor Network," in *Wireless Communications, Networking and Mobile Computing, 2009. WiCom '09. 5th International Conference on*, 2009, pp. 1-4.

[115] K. In Tae and H. Seong Oun, "An Efficient Identity-Based Broadcast Signcryption Scheme for Wireless Sensor Networks," in *Wireless and Pervasive Computing (ISWPC), 2011 6th International Symposium on*, 2011, pp. 1-6.

[116] C. Delerablée, *Identity-Based Broadcast Encryption with Constant Size Ciphertexts and Private Keys* vol. 4833: Springer Berlin Heidelberg, 2007, pp. 200-215.

[117] A. Moh‘d, H. Marzi, N. Aslam, W. Phillips, and W. Robertsona, "A Secure Platform of Wireless Sensor Networks," *Procedia Computer Science,* vol. 5, pp. 115-122, 2011.

[118] Y. Wei, Z. Sahinoglu, and A. Vetro, "Energy Efficient JPEG 2000 Image Transmission over Wireless Sensor Networks," in *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, 2004, pp. 2738-2743 Vol.5.

[119] C. F. Chiasserini and E. Magli, "Energy-Efficient Coding and Error Control for Wireless Video-Surveillance Networks," *Telecommunication Systems,* vol. 26, pp. 369-387, 2004.

[120] C. Nanjunda, M. A. Haleem, and R. Chandramouli, "Robust Encryption for Secure Image Transmission over Wireless Channels," in *Communications, 2005. ICC 2005. 2005 IEEE International Conference on*, 2005, pp. 1287-1291 Vol. 2.

[121] S.-C. Ou, H.-Y. Chung, and W.-T. Sung, "Improving the Compression and Encryption of Images using FPGA-Based Cryptosystems," *Multimedia Tools and Applications,* vol. 28, pp. 5-22, 2006/01/01 2006.

[122] C. N. Mathur, K. Narayan, and K. Subbalakshmi, "On the Design of Error-Correcting Ciphers," *EURASIP Journal on Wireless Communications and Networking,* vol. 2006, p. 042871, 2006.

[123] M. Ruiping, X. Liudong, and H. E. Michel, "A New Mechanism for Achieving Secure and Reliable Data Transmission in Wireless Sensor Networks," in *Technologies for Homeland Security, 2007 IEEE Conference on*, 2007, pp. 274-279.

[124] L. Casado and P. Tsigas, *ContikiSec: A Secure Network Layer for Wireless Sensor Networks under the Contiki Operating System* vol. 5838: Springer Berlin Heidelberg, 2009, pp. 133-147.

[125] H. Cam, O. N. Ucan, N. Odabasioglu, and A. C. Sonmez, "Performance of Joint Multilevel/AES-LDPCC-CPFSK Schemes over Wireless Sensor Networks," *International Journal of Communication Systems,* vol. 23, pp. 77-90, 2010.

[126] Y. Hekim, N. Odabasioglu, and O. N. Ucan, "Performance of Low Density Parity Check Coded Continuous Phase Frequency Shift Keying (LDPCC-CPFSK) over Fading Channels," *International Journal of Communication Systems,* vol. 20, pp. 397-410, 2007.

[127] R. Yi, V. Oleshchuk, and F. Y. Li, "A Scheme for Secure and Reliable Distributed Data Storage in Unattended WSNs," in *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, 2010, pp. 1-6.

[128] M. B. Abdullahi, W. Guojun, and F. Musau, "A Reliable and Secure Distributed In-Network Data Storage Scheme in Wireless Sensor Networks," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, 2011, pp. 548-555.

[129] H. Alwan and A. Agarwal, "A Multipath Routing Approach for Secure and Reliable Data Delivery in Wireless Sensor Networks," *International Journal of Distributed Sensor Networks,* vol. 2013, p. 10, 2013.

[130] F. Chen, F. Lim, O. Abari, A. Chandrakasan, and V. Stojanovic, "Energy-Aware Design of Compressed Sensing Systems for Wireless Sensors Under Performance and Reliability Constraints," *Circuits and Systems I: Regular Papers, IEEE Transactions on,* vol. 60, pp. 650-661, 2013.

[131] A. Rudra, P. Dubey, C. Jutla, V. Kumar, J. Rao, and P. Rohatgi, *Efficient Rijndael Encryption Implementation with Composite Field Arithmetic* vol. 2162: Springer Berlin / Heidelberg, 2001, pp. 171-184.

[132] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, *A Compact Rijndael Hardware Architecture with S-Box Optimization* vol. 2248: Springer Berlin / Heidelberg, 2001, pp. 239-254.

[133] J. P. DesChamps, J. L. Imaña, and G. D. Sutter, *Hardware Implementation of Finite-Field Arithmetic*: McGraw-Hill, 2009.

[134] C. C. Wang, T. K. Troung, H. M. Shao, L. J. Deutsch, J. K. Omura, and I. S. Reed, "VLSI Architectures for Computing Multiplications and Inverses in GF(2^m)," *Computers, IEEE Transactions on,* vol. C-34, pp. 709-717, 1985.

[135] S. M. Sait and H. Youssef, *VLSI Physical Design Automation: Theory and Practice*: World Scientific, 1999.

[136] J. F. Wakerly, *Digital Design - Principles and Practices: 3rd Edition.*

[137] T. L. Floyd, *Digital Fundamentals*: Pearson Prentice Hall, 2006.

[138] 65nm CMOS Process Technology, Last Accessed: 3/01/2016, Available: http://www.fujitsu.com/us/Images/65nmProcessTechnology.pdf

[139] S. Katzen, *The Essential PIC18® Microcontroller*: Springer, 2010, pp. 41-44.

[140] A. R. Calderbank, I. Daubechies, W. Sweldens, and Y. Boon-Lock, "Lossless image compression using integer to integer wavelet transforms," in *Image Processing, 1997. Proceedings., International Conference on*, 1997, pp. 596-599 vol.1.

[141] L. Sze-Wei and L. Soon-Chieh, "VLSI Design of a Wavelet Processing Core," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 16, pp. 1350-1361, 2006.

[142] M. Angelopoulou, K. Masselos, P. K. Cheung, and Y. Andreopoulos, "Implementation and Comparison of the 5/3 Lifting 2D Discrete Wavelet Transform Computation Schedules on FPGAs," *Journal of Signal Processing Systems,* vol. 51, pp. 3-21, 2008/04/01 2008.

[143] L. Chung-Jr, C. Kuan-Fu, C. Hong-Hui, and C. Liang-Gee, "Lifting based discrete wavelet transform architecture for JPEG2000," in *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, 2001, pp. 445-448 vol. 2.

[144] A. R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo, "Wavelet Transforms That Map Integers to Integers," *Applied and Computational Harmonic Analysis,* vol. 5, pp. 332-369, 1998.

[145] G. Strang and T. Nguyen, *Wavelets and Filter Banks*: Wellesley-Cambridge Press, 1996.

[146] Spartan-3L Low Power FPGA Family, Last Accessed: 27/10/2014, Available: http://www.xilinx.com/support/documentation/data_sheets/ds313.pdf

[147] Spartan-3 FPGA Family Data Sheet, Last Accessed: 27/10/2014, Available: http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf

[148] J. H. Kong, L. M. Ang, K. P. Seng, and A. O. Adejo, "Minimal Instruction Set FPGA AES Processor using Handel-C," in *Computer Applications and Industrial Electronics (ICCAIE), 2010 International Conference on*, 2010, pp. 340-344.

[149] J. Ong, L. Ang, and K. Seng, *Lifting Scheme DWT Implementation in a Wireless Vision Sensor Network* vol. 5857: Springer Berlin / Heidelberg, 2009, pp. 627-635.

[150] Spartan-6 Family Overview, Last Accessed: 27/10/2014, Available: http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf

[151] Spartan-3 Generation FPGA User Guide, Last Accessed: 27/10/2014, Available: http://www.xilinx.com/support/documentation/user_guides/ug331.pdf

[152] High-Volume Spartan-6 FPGAs: Performance and Power Leadership by Design, Last Accessed: 27/10/2014, Available: http://www.xilinx.com/support/documentation/white_papers/wp396_S6_HV_Perf_Power.pdf

[153] Power Consumption at 40 and 45 nm, Last Accessed: 18/11/2014, Available: http://www.xilinx.com/support/documentation/white_papers/wp298.pdf

[154] Product Manual: XBee / XBee-PRO ZB RF Modules, Last Accessed: 05/11/2014, Available: http://ftp1.digi.com/support/documentation/90000976_U.pdf

[155] J. Daemen and V. Rijmen, *The Design of Rijndael: AES--the Advanced Encryption Standard*: Springer, 2002.

[156] Virtex-II Platform FPGAs: Complete Data Sheet, Last Accessed: 21/11/2014, Available: http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf

A. APPENDICES

A.1 DWT CRS MISC ARCHITECTURE IN VHDL

A.1.1 Control Signals Combinational Circuit Testbench - tb_Control.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY tb_Control IS
END tb_Control;

ARCHITECTURE behavior OF tb_Control IS

    -- COMPONENT controls
    PORT(
        CLK : IN  std_logic;
        N : IN  std_logic;
        ALU_A : OUT  std_logic;
        ALU_B0 : OUT  std_logic;
        ALU_B1 : OUT  std_logic;
        CIN : OUT  std_logic;
        MAR_SEL : OUT  std_logic;
        PC_WRITE : OUT  std_logic;
        R_WRITE : OUT  std_logic;
        Z_WRITE : OUT  std_logic;
        N_WRITE : OUT  std_logic;
        MAR_WRITE : OUT  std_logic;
        MDR_WRITE : OUT  std_logic;
        MEM_READ : OUT  std_logic;
        MEM_WRITE : OUT  std_logic;
        OP_OUT_SEL : OUT  std_logic;
        OP0_WRITE : OUT  std_logic;
        OP1_WRITE : OUT  std_logic;
        OP_SEL : OUT  std_logic
       );
    END COMPONENT;

   --Inputs
   signal tb_CLK : std_logic := '0';
   signal tb_N : std_logic := '0';

   --Outputs
   signal tb_ALU_A : std_logic;
   signal tb_ALU_B0 : std_logic;
   signal tb_ALU_B1 : std_logic;
   signal tb_CIN : std_logic;
   signal tb_MAR_SEL : std_logic;
   signal tb_PC_WRITE : std_logic;
   signal tb_R_WRITE : std_logic;
   signal tb_Z_WRITE : std_logic;
   signal tb_N_WRITE : std_logic;
   signal tb_MAR_WRITE : std_logic;
   signal tb_MDR_WRITE : std_logic;
   signal tb_MEM_READ : std_logic;
   signal tb_MEM_WRITE : std_logic;
   signal tb_OP_OUT_SEL : std_logic;
   signal tb_OP0_WRITE : std_logic;
   signal tb_OP1_WRITE : std_logic;
   signal tb_OP_SEL : std_logic;

   -- Clock period definitions
   constant CLK : time := 20 ns;

BEGIN

                -- Instantiate the Unit Under Test (UUT)
   uut: controls PORT MAP (
          CLK => tb_CLK,
          N => tb_N,
          ALU_A => tb_ALU_A,
          ALU_B0 => tb_ALU_B0,
          ALU_B1 => tb_ALU_B1,
          CIN => tb_CIN,
          MAR_SEL => tb_MAR_SEL,
          PC_WRITE => tb_PC_WRITE,
          R_WRITE => tb_R_WRITE,
          Z_WRITE => tb_Z_WRITE,
          N_WRITE => tb_N_WRITE,
          MAR_WRITE => tb_MAR_WRITE,
          MDR_WRITE => tb_MDR_WRITE,
          MEM_READ => tb_MEM_READ,
          MEM_WRITE => tb_MEM_WRITE,
          OP_OUT_SEL => tb_OP_OUT_SEL,
          OP0_WRITE => tb_OP0_WRITE,
          OP1_WRITE => tb_OP1_WRITE,
          OP_SEL => tb_OP_SEL
        );

   -- Clock process definitions
   CLK_process :process
   begin
                              tb_CLK <= '0';
                              wait for CLK/2;
                              tb_CLK <= '1';
                              wait for CLK/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ms.
      wait for CLK*5;
                              tb_N <= '0';
      wait for CLK;
                              tb_N <= '1';
      wait for CLK;
                              tb_N <= '0';
      wait;
   end process;

END;
```

A.1.2 Control Signals Combinational Circuit - controls.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity controls is
    port(   CLK : in std_logic;
            N : in std_logic;
            ALU_A : out std_logic;
            ALU_B0 : out std_logic;
            ALU_B1 : out std_logic;
            CIN : out std_logic;
            MAR_SEL : out std_logic;
            PC_WRITE : out std_logic;
            R_WRITE : out std_logic;
            Z_WRITE : out std_logic;
            N_WRITE : out std_logic;
            MAR_WRITE : out std_logic;
            MDR_WRITE : out std_logic;
            MEM_READ : out std_logic;
            MEM_WRITE : out std_logic;
            OP_OUT_SEL : out std_logic;
            OP0_WRITE : out std_logic;
            OP1_WRITE : out std_logic;
            OP_SEL : out std_logic);
end controls;

architecture Behavioral of controls is

    signal iCount4 : std_logic_vector(3 downto 0) := X"8";

begin

    -- 4 Bit Counter
    process(CLK)
    begin

        if CLK'event and CLK = '1' then
            if iCount4 = "1000" then
                iCount4 <= (others=>'0');
            else
                iCount4 <= iCount4 + 1;
            end if;
        end if;

    end process;


    -- Output control signals
    ALU_A <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(0))) )
        OR ( (NOT(iCount4(2))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) );
    ALU_B0 <= (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) AND (iCount4(0));
    ALU_B1 <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(0))) )
        OR ( (NOT(iCount4(2))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) );
    CIN <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (iCount4(0)) )
        OR ( (iCount4(3)) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
                AND (NOT(iCount4(0))) );
        MAR_SEL <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) )
                OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (iCount4(0)) );
        PC_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (iCount4(1)) AND (iCount4(0)) AND N )
                OR ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
                        AND (iCount4(0)) )
                OR ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
                        AND (NOT(iCount4(0))) )
                OR ( (iCount4(3)) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
                        AND (NOT(iCount4(0))) );
        R_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (iCount4(1))
                AND (NOT(iCount4(0))) );
        Z_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
                AND (NOT(iCount4(0))) );
        N_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) AND (iCount4(0)) );
        MAR_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (iCount4(0)) )
                OR ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(0))) )
                OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) );
        MDR_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
                AND (iCount4(0)) );
        MEM_READ <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) )
                OR ( (NOT(iCount4(3))) AND (iCount4(1)) AND (iCount4(0)) )
                OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (iCount4(0)) )
                OR ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (iCount4(1))
                        AND (NOT(iCount4(0))) );
        MEM_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (iCount4(1))
                AND (NOT(iCount4(0))) );
        OP_OUT_SEL <= ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) )
                OR ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) );
        OP0_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
                AND (NOT(iCount4(0))) );
        OP1_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
                AND (iCount4(0)) );
        OP_SEL <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
                AND (NOT(iCount4(0))) );

end Behavioral;
```

A.1.3 DWT CRS MISC Architecture Testbench - tb_DWTCRSMISC.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY tb_DWTCRSMISC_vhd IS
END tb_DWTCRSMISC_vhd;

ARCHITECTURE behavior OF tb_DWTCRSMISC_vhd IS

        -- Component Declaration for the Unit Under Test (UUT)
        COMPONENT DWTCRSMISC
        PORT(
                CLK : IN std_logic;
                UP : IN std_logic;
                DOWN : IN std_logic;
                EnaRead : IN std_logic;
                Seg0 : OUT std_logic_vector(6 downto 0);
                Seg1 : OUT std_logic_vector(6 downto 0);
                RamAdd : OUT std_logic_vector(7 downto 0)
                );
        END COMPONENT;
        --Inputs
        SIGNAL tb_CLK :  std_logic := '0';
        SIGNAL tb_UP :  std_logic := '0';
        SIGNAL tb_DOWN :  std_logic := '0';
        SIGNAL tb_EnaRead :  std_logic := '0';

        --Outputs
        SIGNAL tb_Seg0 :  std_logic_vector(6 downto 0);
        SIGNAL tb_Seg1 :  std_logic_vector(6 downto 0);
        SIGNAL tb_RamAdd :  std_logic_vector(7 downto 0);

        SIGNAL tb_CLK_State : std_logic := '0';

        CONSTANT period : time := 20ns;

BEGIN
        -- Instantiate the Unit Under Test (UUT)
        uut: DWTCRSMISC PORT MAP(
                Seg0 => tb_Seg0,
                Seg1 => tb_Seg1,
                RamAdd => tb_RamAdd,
                CLK => tb_CLK,
                UP => tb_UP,
                DOWN => tb_DOWN,
                EnaRead => tb_EnaRead
        );

        -- Clock
        PROCESS
        BEGIN

                tb_CLK <= '0';
                wait for period*5;
                tb_CLK <= '0';
                wait for period/2;
                -- Start Clock 0
                tb_CLK <= '1';
                wait for period/2;
                tb_CLK <= '0';
                wait for period/2;
                tb_CLK <= '1';
                wait for period/2;
                tb_CLK <= '0';
                -- Start Clock 1
                wait for period/2;
                tb_CLK <= '1';
                wait for period/2;
                tb_CLK <= '0';
                wait for period/2;
                tb_CLK <= '1';
                wait for period/2;
                tb_CLK <= '0';
                -- Start Clock 2
                wait for period/2;
                tb_CLK <= '1';
                wait for period/2;
                tb_CLK <= '0';
                wait for period/2;
                tb_CLK <= '1';
                wait for period/2;
                tb_CLK <= '0';
                -- Start Clock 3
                wait for period/2;
                tb_CLK <= '1';
                wait for period/2;
                tb_CLK <= '0';
```

```
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 4
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 5
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 6
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 7
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 8
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 0
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 1
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
```

```
-- Start Clock 2
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 3
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 4
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 5
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 6
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 7
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 8
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 0
wait for period/2;
tb_CLK <= '1';
wait for period/2;
```

```
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 1
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 2
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 3
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 4
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 5
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 6
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
-- Start Clock 7
wait for period/2;
tb_CLK <= '1';
wait for period/2;
tb_CLK <= '0';
wait for period/2;
tb_CLK <= '1';
wait for period/2;
```

```
			tb_CLK <= '0';
			-- Start Clock 8
			wait for period/2;
			tb_CLK <= '1';
			wait for period/2;
			tb_CLK <= '0';
			wait for period/2;
			tb_CLK <= '1';
			wait for period/2;
			tb_CLK <= '0';
			-- Start Clock 0
			wait for period/2;
			tb_CLK <= '1';
			wait for period/2;
			tb_CLK <= '0';
			wait for period/2;
			tb_CLK <= '1';
			wait for period/2;
			tb_CLK <= '0';
			-- Start Clock 1
			wait for period/2;
			tb_CLK <= '1';
			wait for period/2;
			tb_CLK <= '0';
			wait for period/2;
			tb_CLK <= '1';
			wait for period/2;
			tb_CLK <= '0';
			wait;

		END PROCESS;


		PROCESS
		BEGIN

			tb_UP <= '0';
			tb_DOWN <= '0';
			tb_EnaRead <= '0';
			wait;

		END PROCESS;

END;
```

A.1.4 Top Level DWT CRS MISC Architecture - DWTCRSMISC.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity DWTCRSMISC is
    Port ( Seg0, Seg1 : out  STD_LOGIC_VECTOR (6 downto 0);
           RamAdd : out  STD_LOGIC_VECTOR (7 downto 0);
           CLK : in  STD_LOGIC;
           UP : in  STD_LOGIC;
```

```
        DOWN : in  STD_LOGIC;
                                   EnaRead : in STD_LOGIC);
end DWTCRSMISC;

architecture Behavioral of DWTCRSMISC is

        -- Components
        -- Block RAM
        component memory
  port (
  addra: IN std_logic_VECTOR(10 downto 0);
  addrb: IN std_logic_VECTOR(10 downto 0);
  clka: IN std_logic;
  clkb: IN std_logic;
  dina: IN std_logic_VECTOR(11 downto 0);
  douta: OUT std_logic_VECTOR(11 downto 0);
  doutb: OUT std_logic_VECTOR(11 downto 0);
  ena: IN std_logic;
  enb: IN std_logic;
  wea: IN std_logic);
        end component;

        -- Controls
        component controls
                port(   CLK : in std_logic;
                        N : in std_logic;
                        ALU_A : out std_logic;
                        ALU_B0 : out std_logic;
                        ALU_B1 : out std_logic;
                        CIN : out std_logic;
                        MAR_SEL : out std_logic;
                        PC_WRITE : out std_logic;
                        R_WRITE : out std_logic;
                        Z_WRITE : out std_logic;
                        N_WRITE : out std_logic;
                        MAR_WRITE : out std_logic;
                        MDR_WRITE : out std_logic;
                        MEM_READ : out std_logic;
                        MEM_WRITE : out std_logic;
                        OP_OUT_SEL : out std_logic;
                        OP0_WRITE : out std_logic;
                        OP1_WRITE : out std_logic;
                        OP_SEL : out std_logic);
        end component;

        -- MUX
        -- MUX2-1 1-Bit
        component MUX21 is
                port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
                         B : in  STD_LOGIC_VECTOR (7 downto 0);
                         SEL : in STD_LOGIC;
                         C : out  STD_LOGIC_VECTOR (7 downto 0));
        end component;
        -- MUX2-1 2-Bit
        component MUX22 is
                port ( A : in  STD_LOGIC_VECTOR (1 downto 0);
                         B : in  STD_LOGIC_VECTOR (1 downto 0);
                         SEL : in STD_LOGIC;
                         C : out  STD_LOGIC_VECTOR (1 downto 0));
        end component;
        -- MUX4-1 11-Bit
        component MUX411 is
                port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
                         B : in  STD_LOGIC_VECTOR (10 downto 0);
                         C : in  STD_LOGIC_VECTOR (10 downto 0);
                         D : in  STD_LOGIC_VECTOR (10 downto 0);
                         E : out  STD_LOGIC_VECTOR (10 downto 0);
```

```
                              SEL : in  STD_LOGIC_VECTOR (1 downto 0));
end component;
-- MUX1-4 11-Bit
component MUX114 is
          port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
                              SEL : in  STD_LOGIC_VECTOR (1 downto 0);
                              B : out  STD_LOGIC_VECTOR (10 downto 0);
                              C : out  STD_LOGIC_VECTOR (10 downto 0);
                              D : out  STD_LOGIC_VECTOR (10 downto 0);
                              E : out  STD_LOGIC_VECTOR (10 downto 0));
end component;
-- MUX1-2 1-Bit
component MUX11 is
          port ( A : in  STD_LOGIC;
                              SEL : in  STD_LOGIC;
                              B : out  STD_LOGIC;
                              C : out  STD_LOGIC);
end component;
-- MUX2-1 11-Bit
component MUX211 is
          port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
                              B : in  STD_LOGIC_VECTOR (10 downto 0);
                              SEL : in  STD_LOGIC;
                              C : out  STD_LOGIC_VECTOR (10 downto 0));
end component;

-- Registers
-- 11-Bit Register
component REG is
          port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
                              B : out  STD_LOGIC_VECTOR (10 downto 0);
                              CLK : in  STD_LOGIC;
                              ENA : in  STD_LOGIC);
end component;
-- 1-Bit Register
component REG1BIT is
          Port ( A : in  STD_LOGIC;
                              B : out  STD_LOGIC;
                              CLK : in  STD_LOGIC;
                              ENA : in  STD_LOGIC);
end component;
-- 12-Bit Register
component REG12BIT is
          Port ( A : in  STD_LOGIC_VECTOR (11 downto 0);
                              B : out  STD_LOGIC_VECTOR (11 downto 0);
                              CLK : in  STD_LOGIC;
                              ENA : in  STD_LOGIC);
end component;
-- PC Register
component REGPC is
          port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
                              B : out  STD_LOGIC_VECTOR (10 downto 0);
                              CLK : in  STD_LOGIC;
                              ENA : in  STD_LOGIC);
end component;

-- SBN
component SBN is
          Port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
                              B : in  STD_LOGIC_VECTOR (10 downto 0);
                              CIN : in  STD_LOGIC;
                              N : out  STD_LOGIC;
                              Z : out  STD_LOGIC;
                              O : out STD_LOGIC_VECTOR (10 downto 0));
end component;

-- 11-Bit XOR
```

```
component GF211Add is
        Port ( a : in  STD_LOGIC_VECTOR (10 downto 0);
                        b : in  STD_LOGIC_VECTOR (10 downto 0);
                        c : out  STD_LOGIC_VECTOR (10 downto 0));
end component;

-- 11-Bit To 8-Bit Conversion
component C11TO8 is
        Port ( a : in  STD_LOGIC_VECTOR (10 downto 0);
                        b : in  STD_LOGIC_VECTOR (10 downto 0);
                        c : out  STD_LOGIC_VECTOR (10 downto 0));
end component;

-- GF28
component GF28 is
        Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
                        b : in  STD_LOGIC_VECTOR (7 downto 0);
                        c : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

-- D4to7 Conversion
component D4to7 is
        Port ( Q : in  STD_LOGIC_VECTOR (3 downto 0);
                        Seg : out STD_LOGIC_VECTOR (6 downto 0));
end component;

-- Signals
signal iCount1 : std_logic := '0';
signal iClock2 : std_logic;
signal iCount23 : std_logic_vector(22 downto 0);
signal iClock23 : std_logic;
signal iRamAdd : std_logic_vector(10 downto 0);
signal iRamRead : std_logic_vector(11 downto 0);

-- Program Counter
signal iPC : std_logic_vector(10 downto 0) := (others=>'0');
signal PC : std_logic_vector(10 downto 0) := (others=>'0');
signal PC_Write : std_logic;
-- R Register
signal iR : std_logic_vector(10 downto 0);
signal R : std_logic_vector(10 downto 0);
signal R_Write : std_logic;

-- OPCODE
-- MUX
signal OP_SEL : std_logic;
-- OP0 Register
signal iOP0 : std_logic;
signal OP0_Write : std_logic;
-- OP1 Register
signal iOP1 : std_logic;
signal OP1_Write : std_logic;
-- OPCODE output
signal OPCODE : std_logic_vector(1 downto 0);
signal OP_OUT_SEL : std_logic;
signal oOPCODE : std_logic_vector(1 downto 0);

-- MEMORY
-- MDR register
signal iMDR : std_logic_vector(11 downto 0);
signal MDR : std_logic_vector(11 downto 0);
signal MDR_Write : std_logic;
-- MAR
signal iMAR : std_logic_vector(10 downto 0);
signal MAR : std_logic_vector(10 downto 0);
signal MAR_Write : std_logic;
signal MAR_SEL : std_logic;
```

```
-- MEM Output
signal oMemory : std_logic_vector(11 downto 0);
-- MEM Controls
signal MEM_READ : std_logic;
signal MEM_WRITE : std_logic;
signal MEM_ENA : std_logic;
signal MEM_WEA : std_logic;

-- SBN/ALU Block
-- N register
signal i_N : std_logic;
signal N : std_logic;
signal N_Write : std_logic;
-- Z register
signal iZ : std_logic;
signal Z : std_logic;
signal Z_Write : std_logic;
-- ALU MUX
-- ALU MUX A
signal iMUXALUA0 : std_logic_vector(10 downto 0) := (others=>'0');
signal iMUXALUA1 : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXALUA : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXALUASEL : std_logic;
signal ALU_A : std_logic;
-- ALU MUX B
signal iMUXALUB0 : std_logic_vector(10 downto 0) := (others=>'0');
signal iMUXALUB1 : std_logic_vector(10 downto 0) := (others=>'0');
signal iMUXALUB2 : std_logic_vector(10 downto 0) := (others=>'0');
signal iMUXALUB3 : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXALUB : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXALUBSEL : std_logic_vector(1 downto 0);
signal ALU_B : std_logic_vector(1 downto 0);
-- INV
signal iINV : std_logic_vector(10 downto 0) := (others=>'0');
signal oINV : std_logic_vector(10 downto 0) := (others=>'0');
-- CIN
signal CIN : std_logic;
-- Output
signal ADDER : std_logic_vector(10 downto 0) := (others=>'0');

-- GF Block
signal iGFA : std_logic_vector(10 downto 0) := (others=>'0');
signal iGFB : std_logic_vector(10 downto 0) := (others=>'0');
signal iGF : std_logic_vector(10 downto 0) := (others=>'0');

-- XOR Block
signal iXORA : std_logic_vector(10 downto 0) := (others=>'0');
signal iXORB : std_logic_vector(10 downto 0) := (others=>'0');
signal iXOR : std_logic_vector(10 downto 0) := (others=>'0');

-- 11TO8 Block
signal i11TO8A : std_logic_vector(10 downto 0) := (others=>'0');
signal i11TO8B : std_logic_vector(10 downto 0) := (others=>'0');
signal i11TO8 : std_logic_vector(10 downto 0) := (others=>'0');

-- Instruction MUX
-- MUX A
signal iMUXA : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXA0 : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXA1 : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXA2 : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXA3 : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXA4 : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXASEL : std_logic_vector(1 downto 0);
-- MUX B
signal iMUXB : std_logic_vector(10 downto 0) := (others=>'0');
signal MUXB0 : std_logic_vector(10 downto 0) := (others=>'0');
```

```
		signal MUXB1 : std_logic_vector(10 downto 0) := (others=>'0');
		signal MUXB2 : std_logic_vector(10 downto 0) := (others=>'0');
		signal MUXB3 : std_logic_vector(10 downto 0) := (others=>'0');
		signal MUXB4 : std_logic_vector(10 downto 0) := (others=>'0');
		signal MUXBSEL : std_logic_vector(1 downto 0);
		-- MUX Out
		signal MUXO : std_logic_vector(10 downto 0) := (others=>'0');
		signal iMUXO0 : std_logic_vector(10 downto 0) := (others=>'0');
		signal iMUXO1 : std_logic_vector(10 downto 0) := (others=>'0');
		signal iMUXO2 : std_logic_vector(10 downto 0) := (others=>'0');
		signal iMUXO3 : std_logic_vector(10 downto 0) := (others=>'0');
		signal iMUXO4 : std_logic_vector(10 downto 0) := (others=>'0');
		signal MUXOSEL : std_logic_vector(1 downto 0);

begin
		-- Controls Block
		Ctrl : controls
						port map(
										CLK => iClock2,
										N => N,
										ALU_A => ALU_A,
										ALU_B0 => ALU_B(0),
										ALU_B1 => ALU_B(1),
										CIN => CIN,
										MAR_SEL => MAR_SEL,
										PC_WRITE => PC_Write,
										R_WRITE => R_Write,
										Z_WRITE => Z_Write,
										N_WRITE => N_Write,
										MAR_WRITE => MAR_Write,
										MDR_WRITE => MDR_Write,
										MEM_READ => MEM_READ,
										MEM_WRITE => MEM_WRITE,
										OP_OUT_SEL => OP_OUT_SEL,
										OP0_WRITE => OP0_Write,
										OP1_WRITE => OP1_Write,
										OP_SEL => OP_SEL);

		-- Registers
		-- PC
		PC_Reg : REGPC
				port map(
								A => iPC,
								B => PC,
								CLK => iClock2,
								ENA => PC_Write);
		iPC <= ADDER;

		-- R
		R_Reg : REG
				port map(
								A => iR,
								B => R,
								CLK => iClock2,
								ENA => R_Write);
		iR <= oMemory(10 downto 0);

		-- Z
		Z_Reg : REG1BIT
				port map(
								A => iZ,
								B => Z,
								CLK => iClock2,
								ENA => Z_Write);

		-- N
		N_Reg : REG1BIT
```

```
        port map(
                        A => i_N,
                        B => N,
                        CLK => iClock2,
                        ENA => N_Write);

-- OPCODE
-- OPCODE1
OPCODE1_Reg : REG1BIT
        port map(
                        A => iOP1,
                        B => OPCODE(1),
                        CLK => iClock2,
                        ENA => OP1_Write);

-- OPCODE0
OPCODE0_Reg : REG1BIT
        port map(
                        A => iOP0,
                        B => OPCODE(0),
                        CLK => iClock2,
                        ENA => OP0_Write);

-- oMUXOP
oMUXOP : MUX22
        port map (
                        A => OPCODE,
                        B => "10",
                        SEL => OP_OUT_SEL,
                        C => oOPCODE);

-- MUXOP
MUXOP : MUX11
        port map (
                        A => oMemory(11),
                        SEL => OP_SEL,
                        B => iOP1,
                        C => iOP0);


-- MUXA
MUXA : MUX114
        port map (
                A => iMUXA,
                SEL => MUXASEL,
                B => MUXA0,
                C => MUXA1,
                D => MUXA2,
                E => MUXA3);
MUXASEL <= oOPCODE;
iMUXA <= oMemory(10 downto 0);
iGFA <= MUXA0;
iXORA <= MUXA1;
iMUXALUA0 <= MUXA2;
i11TO8A <= MUXA3;

-- MUXB
MUXB : MUX114
        port map (
                A => iMUXB,
                SEL => MUXBSEL,
                B => MUXB0,
                C => MUXB1,
                D => MUXB2,
                E => MUXB3);
MUXBSEL <= oOPCODE;
iMUXB <= R;
```

```
iGFB <= MUXB0;
iXORB <= MUXB1;
iINV <= MUXB2;
i11TO8B <= MUXB3;

-- MUX OUT
MUXOUT : MUX411
        port map (
                        A => iMUXO0,
                        B => iMUXO1,
                        C => iMUXO2,
                        D => iMUXO3,
                        E => MUXO,
                        SEL => MUXOSEL);
MUXOSEL <= oOPCODE;
iMUXO0 <= iGF;
iMUXO1 <= iXOR;
iMUXO2 <= ADDER;
iMUXO3 <= i11TO8;
iMDR <= oMemory(11) & MUXO;

-- 11TO8
C11TO8BLOCK : C11TO8
                port map (
                                a =>i11TO8A,
                                b =>i11TO8B,
                                c =>i11TO8);
-- i11TO8(7 downto 0) <= i11TO8A(10) & i11TO8A(6 downto 0);
-- i11TO8(10 downto 8) <= "000";

-- GF
GF28MULT : GF28
        port map (
                        a => iGFA(7 downto 0),
                        b => iGFB(7 downto 0),
                        c => iGF(7 downto 0));
iGF(10 downto 8) <= (others=>'0');

-- XOR
GF211XOR : GF211Add
                port map (
                                a =>iXORA,
                                b =>iXORB,
                                c =>iXOR);

-- SBN
SBN_BLOCK : SBN
        port map (
                        A => MUXALUA,
                        B => MUXALUB,
                        CIN => CIN,
                        N => i_N,
                        Z => iZ,
                        O => ADDER);

-- ALU_A MUX
MUX_ALU_A : MUX211
               port map (
                        A => iMUXALUA0,
                        B => iMUXALUA1,
                        SEL => MUXALUASEL,
                        C => MUXALUA);
MUXALUASEL <= ALU_A;
iMUXALUA1 <= PC;

-- ALU_B MUX
MUX_ALU_B : MUX411
```

```
            port map (
                            A => iMUXALUB0,
                            B => iMUXALUB1,
                            C => iMUXALUB2,
                            D => iMUXALUB3,
                            E => MUXALUB,
                            SEL => MUXALUBSEL);
    MUXALUBSEL <= ALU_B;
    iMUXALUB0 <= PC;
    iMUXALUB1 <= oINV;
    iMUXALUB2 <= "00000000000";
    iMUXALUB3 <= "00000000000";

    -- INV
    oINV <= NOT iINV;

    -- MDR
    MDR_Reg : REG12BIT
            port map(
                            A => iMDR,
                            B => MDR,
                            CLK => iClock2,
                            ENA => MDR_Write);

    -- MUX MAR
    MUXMAR : MUX211
            port map(
                            A => ADDER,
                            B => oMemory(10 downto 0),
                            SEL => MAR_SEL,
                            C => iMAR);

    -- MAR
    MAR_Reg : REG
            port map(
                            A => iMAR,
                            B => MAR,
                            CLK => iClock2,
                            ENA => MAR_Write);
    --      iMAR <= ADDER;

    -- Block RAM
    Block_RAM : memory
port map (
  addra => MAR,
  addrb => iRamAdd,
  clka => CLK,
  clkb => CLK,
  dina => MDR,
  douta => oMemory,
  doutb => iRamRead,
  ena => MEM_ENA,
  enb => EnaRead,
  wea => MEM_WEA);
    -- Memory Control
    process(MEM_READ,MEM_WRITE,CLK)
    begin
            if MEM_WRITE = '1' then
                    MEM_ENA <= '1';
                    MEM_WEA <= MEM_WRITE;
            elsif MEM_READ = '1' then
                    MEM_ENA <= MEM_READ;
                    MEM_WEA <= '0';
            else
                    MEM_ENA <= MEM_READ;
                    MEM_WEA <= '0';
            end if;
```

```
	end process;

	-- RAM Address
	process(iClock23,UP,DOWN)
	begin

		if iClock23'event and iClock23 = '1' then
			if UP = '1' then
				iRamAdd <= iRamAdd + '1';
			elsif DOWN = '1' then
				iRamAdd <= iRamAdd - '1';
			else
				iRamAdd <= iRamAdd;
			end if;
		end if;

	end process;

	RamAdd <= iRamAdd(7 downto 0);
	-- RamRead <= iRamRead(7 downto 0);

	-- Seg0
	Seg70 : D4to7
		 port map(
					Q => iRamRead(3 downto 0),
					Seg => Seg0);

	-- Seg1
	Seg71 : D4to7
		 port map(
					Q => iRamRead(7 downto 4),
					Seg => Seg1);

	-- Clock24
	process(CLK)
	begin

		if CLK'event and CLK = '1' then
			iCount23 <= iCount23 + '1';
		end if;

	end process;
	-- Actual Implementation
		iClock23 <= iCount23(22);
	-- Simulation
	--	PiClock24 <= CLK;

	-- Clock2
	process(CLK)
	begin

		if CLK'event and CLK = '1' then
			if PC /= "11111111111" then
				iCount1 <= NOT iCount1;
			end if;
		end if;

	end process;
	-- Actual Implementation
		iClock2 <= iCount1;
	-- Simulation
	-- iClock2 <= CLK;

end Behavioral;
```

A.1.5 11-Bit Programme Counter Register - REGPC.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity REGPC is
   Port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
          B : out  STD_LOGIC_VECTOR (10 downto 0);
          CLK : in  STD_LOGIC;
          ENA : in  STD_LOGIC);
end REGPC;

architecture Behavioral of REGPC is

begin

        -- Register function
        process(CLK)

                -- Start of programme
                -- 0x2AF, start at 0x2B0
                variable sig_data : std_logic_vector (10 downto 0) := "01010101111";
                -- 0x2E8 For SBN waveform extraction
                -- variable sig_data : std_logic_vector (10 downto 0) := "01011101000";
                -- 0x4E3 For XOR/11TO8/GF MULT waveform extraction
                -- variable sig_data : std_logic_vector (10 downto 0) := "10011100011";
                -- Test Last Line
                -- variable sig_data : std_logic_vector (10 downto 0) := "111011001";

        begin

                if CLK'event and CLK = '1' then
                        if ENA = '1' then
                                sig_data := A;
                        end if;
                else
                        sig_data := sig_data;
                end if;

                B <= sig_data;

        end process;

end Behavioral;
```

A.1.6 11-Bit Register - REG.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
```

```
--use UNISIM.VComponents.all;

entity REG is
   Port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
          B : out  STD_LOGIC_VECTOR (10 downto 0);
          CLK : in  STD_LOGIC;
          ENA : in  STD_LOGIC);
end REG;

architecture Behavioral of REG is

begin

            -- Register function
            process(CLK)

                        variable sig_data : std_logic_vector (10 downto 0) := (others=>'0');

            begin

                        if CLK'event and CLK = '1' then
                                    if ENA = '1' then
                                                sig_data := A;
                                    end if;
                        else
                                    sig_data := sig_data;
                        end if;

                        B <= sig_data;

            end process;

end Behavioral;
```

A.1.7 1-Bit Register - REG1BIT.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity REG1BIT is
   Port ( A : in  STD_LOGIC;
          B : out  STD_LOGIC;
          CLK : in  STD_LOGIC;
          ENA : in  STD_LOGIC);
end REG1BIT;

architecture Behavioral of REG1BIT is

begin

            -- Register function
            process(CLK)

                        variable sig_data : std_logic := '0';
```

```
        begin

                if CLK'event and CLK = '1' then
                        if ENA = '1' then
                                sig_data := A;
                        end if;
                else
                        sig_data := sig_data;
                end if;

                B <= sig_data;

        end process;

end Behavioral;
```

A.1.8 2-Bit 2-To-1 Multiplexer - MUX22.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX22 is
    Port ( A : in  STD_LOGIC_VECTOR (1 downto 0);
           B : in  STD_LOGIC_VECTOR (1 downto 0);
                                        SEL : in STD_LOGIC;
           C : out  STD_LOGIC_VECTOR (1 downto 0));
end MUX22;

architecture Behavioral of MUX22 is

begin

        -- MUX2-1 2 bit
        process(SEL,A,B)
        begin

                if SEL = '0' then
                        C <= A;
                else
                        C <= B;
                end if;

        end process;

end Behavioral;
```

A.1.9 1-Bit 1-To-2 Multiplexer - MUX11.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX11 is
   Port ( A : in  STD_LOGIC;
          SEL : in  STD_LOGIC;
          B : out  STD_LOGIC;
          C : out  STD_LOGIC);
end MUX11;

architecture Behavioral of MUX11 is

begin

        -- MUX1-2 1-Bit
        process(SEL,A)
        begin

                if SEL = '0' then
                        B <= A;
                        C <= '0';
                else
                        B <= '0';
                        C <= A;
                end if;

        end process;

end Behavioral;
```

A.1.10 11-Bit 1-To-4 Multiplexer - MUX114.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX114 is
   Port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
          SEL : in  STD_LOGIC_VECTOR (1 downto 0);
          B : out  STD_LOGIC_VECTOR (10 downto 0);
          C : out  STD_LOGIC_VECTOR (10 downto 0);
                              D : out  STD_LOGIC_VECTOR (10 downto 0);
          E : out  STD_LOGIC_VECTOR (10 downto 0));
end MUX114;

architecture Behavioral of MUX114 is
```

```
begin

        -- MUX1-4 11 bits
        process(SEL,A)
        begin

                if SEL = "00" then
                        B <= A;
                        C <= "00000000000";
                        D <= "00000000000";
                        E <= "00000000000";
                elsif SEL = "01" then
                        B <= "00000000000";
                        C <= A;
                        D <= "00000000000";
                        E <= "00000000000";
                elsif SEL = "10" then
                        B <= "00000000000";
                        C <= "00000000000";
                        D <= A;
                        E <= "00000000000";
                elsif SEL = "11" then
                        B <= "00000000000";
                        C <= "00000000000";
                        D <= "00000000000";
                        E <= A;
                else
                        B <= "00000000000";
                        C <= "00000000000";
                        D <= "00000000000";
                        E <= "00000000000";
                end if;

        end process;

end Behavioral;
```

A.1.11 11-Bit 4-To-1 Multiplexer - MUX411.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX411 is
    Port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
           B : in  STD_LOGIC_VECTOR (10 downto 0);
                                    C : in  STD_LOGIC_VECTOR (10 downto 0);
                                    D : in  STD_LOGIC_VECTOR (10 downto 0);
           E : out  STD_LOGIC_VECTOR (10 downto 0);
           SEL : in  STD_LOGIC_VECTOR (1 downto 0));
end MUX411;

architecture Behavioral of MUX411 is

begin
```

```
        -- MUX4-1 11-bit
        process(SEL,A,B,C,D)
        begin

                if SEL = "00" then
                        E <= A;
                elsif SEL = "01" then
                        E <= B;
                elsif SEL = "10" then
                        E <= C;
                elsif SEL = "11" then
                        E <= D;
                else
                        E <= "00000000000";
                end if;

        end process;

end Behavioral;
```

A.1.12 Functional Block 11TO8 - C11TO8.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity C11TO8 is
    Port ( a : in  STD_LOGIC_VECTOR (10 downto 0);
           b : in  STD_LOGIC_VECTOR (10 downto 0);
           c : out  STD_LOGIC_VECTOR (10 downto 0));
end C11TO8;

architecture Behavioral of C11TO8 is

        signal sig_output : std_logic_vector (10 downto 0) := (others=>'0');

begin

        -- Conversion of 11-bit to 8-bit
        sig_output(7 downto 0) <= a(10) & a(6 downto 0);
        sig_output(10 downto 8) <= "000";

        -- output
        c <= sig_output;

end Behavioral;
```

A.1.13 Functional Block GF(28) Multiplier - GF28.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity GF28 is
   Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
          b : in  STD_LOGIC_VECTOR (7 downto 0);
          c : out  STD_LOGIC_VECTOR (7 downto 0));
end GF28;

architecture Behavioral of GF28 is

        signal output_i : std_logic_vector (7 downto 0) := (others=>'0');

begin

        -- GF(2^8) Multiplier
        -- bit 7
        output_i(7) <= (a(7) AND b(0)) XOR (a(6) AND b(1)) XOR (a(5) AND b(2)) XOR (a(4) AND b(3))
                XOR (a(3) AND b(4)) XOR (a(2) AND b(5)) XOR (a(1) AND b(6)) XOR (a(0) AND b(7))
                XOR (a(7) AND b(6)) XOR (a(6) AND b(7)) XOR (a(7) AND b(5)) XOR (a(6) AND b(6))
                XOR (a(5) AND b(7)) XOR (a(7) AND b(4)) XOR (a(6) AND b(5)) XOR (a(5) AND b(6))
                XOR (a(4) AND b(7));
        -- bit 6
        output_i(6) <= (a(6) AND b(0)) XOR (a(5) AND b(1)) XOR (a(4) AND b(2)) XOR (a(3) AND b(3))
                XOR (a(2) AND b(4)) XOR (a(1) AND b(5)) XOR (a(0) AND b(6)) XOR (a(7) AND b(5))
                XOR (a(6) AND b(6)) XOR (a(5) AND b(7)) XOR (a(7) AND b(4)) XOR (a(6) AND b(5))
                XOR (a(5) AND b(6)) XOR (a(4) AND b(7)) XOR (a(7) AND b(3)) XOR (a(6) AND b(4))
                XOR (a(5) AND b(5)) XOR (a(4) AND b(6)) XOR (a(3) AND b(7));
        -- bit 5
        output_i(5) <= (a(5) AND b(0)) XOR (a(4) AND b(1)) XOR (a(3) AND b(2)) XOR (a(2) AND b(3))
                XOR (a(1) AND b(4)) XOR (a(0) AND b(5)) XOR (a(7) AND b(4)) XOR (a(6) AND b(5))
                XOR (a(5) AND b(6)) XOR (a(4) AND b(7)) XOR (a(7) AND b(3)) XOR (a(6) AND b(4))
                XOR (a(5) AND b(5)) XOR (a(4) AND b(6)) XOR (a(3) AND b(7)) XOR (a(7) AND b(2))
                XOR (a(6) AND b(3)) XOR (a(5) AND b(4)) XOR (a(4) AND b(5)) XOR (a(3) AND b(6))
                XOR (a(2) AND b(7));
        -- bit 4
        output_i(4) <= (a(4) AND b(0)) XOR (a(3) AND b(1)) XOR (a(2) AND b(2)) XOR (a(1) AND b(3))
                XOR (a(0) AND b(4)) XOR (a(7) AND b(7)) XOR (a(7) AND b(3)) XOR (a(6) AND b(4))
                XOR (a(5) AND b(5)) XOR (a(4) AND b(6)) XOR (a(3) AND b(7)) XOR (a(7) AND b(2))
                XOR (a(6) AND b(3)) XOR (a(5) AND b(4)) XOR (a(4) AND b(5)) XOR (a(3) AND b(6))
                XOR (a(2) AND b(7)) XOR (a(7) AND b(1)) XOR (a(6) AND b(2)) XOR (a(5) AND b(3))
                XOR (a(4) AND b(4)) XOR (a(3) AND b(5)) XOR (a(2) AND b(6)) XOR (a(1) AND b(7));
        -- bit 3
        output_i(3) <= (a(3) AND b(0)) XOR (a(2) AND b(1)) XOR (a(1) AND b(2)) XOR (a(0) AND b(3))
                XOR (a(7) AND b(5)) XOR (a(6) AND b(6)) XOR (a(5) AND b(7)) XOR (a(7) AND b(4))
                XOR (a(6) AND b(5)) XOR (a(5) AND b(6)) XOR (a(4) AND b(7)) XOR (a(7) AND b(2))
                XOR (a(6) AND b(3)) XOR (a(5) AND b(4)) XOR (a(4) AND b(5)) XOR (a(3) AND b(6))
                XOR (a(2) AND b(7)) XOR (a(7) AND b(1)) XOR (a(6) AND b(2)) XOR (a(5) AND b(3))
                XOR (a(4) AND b(4)) XOR (a(3) AND b(5)) XOR (a(2) AND b(6)) XOR (a(1) AND b(7));
        -- bit 2
        output_i(2) <= (a(2) AND b(0)) XOR (a(1) AND b(1)) XOR (a(0) AND b(2)) XOR (a(7) AND b(6))
                XOR (a(6) AND b(7)) XOR (a(7) AND b(5)) XOR (a(6) AND b(6)) XOR (a(5) AND b(7))
                XOR (a(7) AND b(3)) XOR (a(6) AND b(4)) XOR (a(5) AND b(5)) XOR (a(4) AND b(6))
                XOR (a(3) AND b(7)) XOR (a(7) AND b(1)) XOR (a(6) AND b(2)) XOR (a(5) AND b(3))
                XOR (a(4) AND b(4)) XOR (a(3) AND b(5)) XOR (a(2) AND b(6)) XOR (a(1) AND b(7));
        -- bit 1
        output_i(1) <= (a(1) AND b(0)) XOR (a(0) AND b(1)) XOR (a(7) AND b(7)) XOR (a(7) AND b(6))
                XOR (a(6) AND b(7)) XOR (a(7) AND b(2)) XOR (a(6) AND b(3)) XOR (a(5) AND b(4))
                XOR (a(4) AND b(5)) XOR (a(3) AND b(6)) XOR (a(2) AND b(7));
        -- bit 0
        output_i(0) <= (a(0) AND b(0)) XOR (a(7) AND b(7)) XOR (a(7) AND b(6)) XOR (a(6) AND b(7))
                XOR (a(7) AND b(5)) XOR (a(6) AND b(6)) XOR (a(5) AND b(7)) XOR (a(7) AND b(1))
                XOR (a(6) AND b(2)) XOR (a(5) AND b(3)) XOR (a(4) AND b(4)) XOR (a(3) AND b(5))
```

```
                        XOR (a(2) AND b(6)) XOR (a(1) AND b(7));

        -- connect output to signal output_i
        c <= output_i;

end Behavioral;
```

A.1.14 Functional Block 11-Bit XOR - GF211Add.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity GF211Add is
   Port ( a : in  STD_LOGIC_VECTOR (10 downto 0);
          b : in  STD_LOGIC_VECTOR (10 downto 0);
          c : out  STD_LOGIC_VECTOR (10 downto 0));
end GF211Add;

architecture Behavioral of GF211Add is

        signal sig_output : std_logic_vector (10 downto 0) := (others=>'0');

begin

        -- GF Addition / XOR
        sig_output <= a XOR b;

        -- output
        c <= sig_output;

end Behavioral;
```

A.1.15 Functional Block SBN - SBN.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity SBN is
   Port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
          B : in  STD_LOGIC_VECTOR (10 downto 0);
          CIN : in  STD_LOGIC;
          N : out  STD_LOGIC;
          Z : out  STD_LOGIC;
                              O : out STD_LOGIC_VECTOR (10 downto 0));
```

```
end SBN;

architecture Behavioral of SBN is

	signal sig_output : std_logic_vector (10 downto 0) := (others=>'0');

begin

	sig_output <= A + B + CIN;

	-- output zero, Z
	process(sig_output)
	begin
		if sig_output = X"000" then
			Z <= '1';
		else
			Z <= '0';
		end if;
	end process;

	-- output negative, N
	process(sig_output)
	begin
		if sig_output(8) = '1' then
			N <= '1';
		else
			N <= '0';
		end if;
	end process;

	-- output O
	O <= sig_output;

end Behavioral;
```

A.1.16 11-Bit 2-To-1 Multiplexer - MUX211.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX211 is
    Port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
           B : in  STD_LOGIC_VECTOR (10 downto 0);
           SEL : in  STD_LOGIC;
           C : out  STD_LOGIC_VECTOR (10 downto 0));
end MUX211;

architecture Behavioral of MUX211 is

begin

	-- MUX2-1 11 bit
	process(SEL,A,B)
	begin
```

```
		if SEL = '0' then
			C <= A;
		else
			C <= B;
		end if;

	end process;

end Behavioral;
```

A.1.17 12-Bit Register - REG12BIT.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity REG12BIT is
   Port ( A : in  STD_LOGIC_VECTOR (11 downto 0);
          B : out  STD_LOGIC_VECTOR (11 downto 0);
          CLK : in  STD_LOGIC;
          ENA : in  STD_LOGIC);
end REG12BIT;

architecture Behavioral of REG12BIT is

begin

	-- Register function
	process(CLK)

		variable sig_data : std_logic_vector(11 downto 0) := (others=>'0');

	begin

		if CLK'event and CLK = '1' then
			if ENA = '1' then
				sig_data := A;
			end if;
		else
			sig_data := sig_data;
		end if;

		B <= sig_data;

	end process;
end Behavioral;
```

A.1.18 LED 7-Segment Display - D4to7.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity D4to7 is
   Port ( Q : in  STD_LOGIC_VECTOR (3 downto 0);
          Seg : out STD_LOGIC_VECTOR (6 downto 0));
end D4to7;

architecture Behavioral of D4to7 is
            -- Segment encoding
            --                      a
            --             ---
            --           f|             |b
            --             --- <- g
            -- e|          |c
            --             ---
            --                      d
begin
            -- Conditional signal assignmens
            -- LED seg order = a,b,c,d,e,f,g = seg6, seg5, seg4, seg3, seg2, seg1, seg0
            Seg<=     "1111110" when q = "0000" else
                                "0110000" when q = "0001" else
                                "1101101" when q = "0010" else
                                "1111001" when q = "0011" else
                                "0110011" when q = "0100" else
                                "1011011" when q = "0101" else
                                "1011111" when q = "0110" else
                                "1110000" when q = "0111" else
                                "1111111" when q = "1000" else
                                "1111011" when q = "1001" else
                                "1110111" when q = "1010" else
                                "0011111" when q = "1011" else
                                "1001110" when q = "1100" else
                                "0111101" when q = "1101" else
                                "1001111" when q = "1110" else
                                "1000111" when q = "1111" else
                                "0000000";

end Behavioral;
```

A.2 DWT CRS MISC PROCESSING SYSTEM IN HANDEL-C

```
#define RC10_TARGET_CLOCK_RATE          25175000
#include "rc10.hch"
#include "stdlib.hch"

#define RegWidth   11
#define MemWidth 12
#define DataWidth  8

macro expr ClockRate = RC10_ACTUAL_CLOCK_RATE;

// Image Data
```

```
static ram unsigned 6 data[4200] with {block = 1};

static ram unsigned MemWidth Memory[2048] = {
//data
//$zero, $one
// 0x000
0x000, 0x001,
//$s0 - $s583
//input data / image data, $s0 - $s255
// 0x002
0x034, 0x01A, 0x011, 0x02D, 0x039, 0x016, 0x024, 0x00E, 0x010, 0x019, 0x031, 0x029, 0x01F, 0x027, 0x039,
0x005, 0x039, 0x03A, 0x002, 0x030,
0x03D, 0x035, 0x01E, 0x03A, 0x033, 0x004, 0x018, 0x02E, 0x01B, 0x025, 0x03E, 0x010, 0x008, 0x032, 0x006,
0x011, 0x023, 0x025, 0x000, 0x009,
0x01B, 0x00F, 0x00F, 0x029, 0x01C, 0x00D, 0x01C, 0x033, 0x03A, 0x03D, 0x034, 0x02B, 0x008, 0x023, 0x015,
0x034, 0x03A, 0x007, 0x019, 0x01C,
0x013, 0x013, 0x007, 0x001, 0x028, 0x029, 0x02C, 0x029, 0x009, 0x03A, 0x00A, 0x022, 0x00B, 0x00B, 0x006,
0x023, 0x020, 0x01E, 0x010, 0x03B,
0x006, 0x002, 0x014, 0x00A, 0x010, 0x012, 0x032, 0x03F, 0x010, 0x00F, 0x008, 0x012, 0x020, 0x00E, 0x01A,
0x02E, 0x011, 0x036, 0x03C, 0x007,
0x035, 0x030, 0x013, 0x005, 0x009, 0x01A, 0x03C, 0x02F, 0x034, 0x036, 0x026, 0x01F, 0x023, 0x03B, 0x002,
0x01F, 0x010, 0x030, 0x021, 0x01C,
0x008, 0x003, 0x03D, 0x00C, 0x032, 0x00C, 0x010, 0x025, 0x03D, 0x02B, 0x01C, 0x03D, 0x034, 0x018, 0x00A,
0x006, 0x037, 0x039, 0x024, 0x02B,
0x029, 0x00E, 0x026, 0x00F, 0x03D, 0x030, 0x018, 0x015, 0x00F, 0x024, 0x026, 0x03D, 0x025, 0x03C, 0x003,
0x00B, 0x018, 0x00A, 0x02D, 0x01D,
0x00A, 0x02F, 0x030, 0x025, 0x03B, 0x004, 0x010, 0x000, 0x023, 0x01F, 0x00F, 0x017, 0x033, 0x00E, 0x00E,
0x03D, 0x03E, 0x019, 0x032, 0x00E,
0x016, 0x003, 0x029, 0x031, 0x009, 0x01F, 0x016, 0x028, 0x022, 0x01B, 0x007, 0x022, 0x03D, 0x029, 0x00B,
0x030, 0x00C, 0x021, 0x02C, 0x034,
0x036, 0x015, 0x034, 0x031, 0x016, 0x013, 0x012, 0x021, 0x01F, 0x00A, 0x01F, 0x010, 0x010, 0x031, 0x02F,
0x037, 0x027, 0x039, 0x000, 0x005,
0x03C, 0x03B, 0x014, 0x00E, 0x033, 0x02D, 0x01C, 0x020, 0x027, 0x03B, 0x01C, 0x005, 0x016, 0x017, 0x002,
0x03B, 0x038, 0x01B, 0x01B, 0x01F,
0x009, 0x002, 0x029, 0x02C, 0x01E, 0x008, 0x005, 0x019, 0x020, 0x007, 0x00A, 0x031, 0x023, 0x00B, 0x020,
0x027,
//generator matrix coefficients, $s256 - $s575
0x07B, 0x053, 0x01E, 0x0A1, 0x062, 0x0D9, 0x026, 0x0F1, 0x0C7, 0x0AC, 0x0D5, 0x04F, 0x0CF, 0x066,
0x005, 0x093, 0x0CF, 0x0AA, 0x062, 0x0F4,
0x01E, 0x0B5, 0x008, 0x017, 0x02A, 0x083, 0x0A5, 0x0C0, 0x07B, 0x00F, 0x02C, 0x0A8, 0x055, 0x08B,
0x0F3, 0x051, 0x0B3, 0x0C8, 0x069, 0x0A6,
0x0E1, 0x00C, 0x0E6, 0x06F, 0x094, 0x0BF, 0x07A, 0x0B2, 0x057, 0x025, 0x0DF, 0x013, 0x0D8, 0x03B,
0x0D3, 0x00C, 0x092, 0x0A6, 0x0DC, 0x088,
0x014, 0x07D, 0x0CD, 0x0BC, 0x079, 0x036, 0x0B0, 0x045, 0x07A, 0x0B6, 0x014, 0x04A, 0x068, 0x038,
0x0F5, 0x041, 0x0D3, 0x078, 0x0B3, 0x01D,
0x073, 0x0B2, 0x0D0, 0x014, 0x0CE, 0x072, 0x045, 0x0CC, 0x07F, 0x07E, 0x0ED, 0x0E7, 0x013, 0x0FB,
0x0B8, 0x08B, 0x09D, 0x015, 0x0D4, 0x048,
0x0DB, 0x059, 0x0C7, 0x01A, 0x026, 0x033, 0x0E7, 0x0ED, 0x049, 0x0C2, 0x03E, 0x0FF, 0x0E1, 0x0AD,
0x03A, 0x07A, 0x045, 0x0C9, 0x0CE, 0x05B,
0x055, 0x090, 0x09E, 0x066, 0x0AE, 0x04F, 0x051, 0x05A, 0x0CD, 0x02B, 0x069, 0x092, 0x03D, 0x0F2, 0x0F3,
0x0ED, 0x013, 0x0CB, 0x0CC, 0x01A,
0x07F, 0x071, 0x057, 0x031, 0x072, 0x02C, 0x03F, 0x070, 0x024, 0x073, 0x0E4, 0x067, 0x0A4, 0x040, 0x074,
0x022, 0x018, 0x09A, 0x0AA, 0x0B1,
0x02F, 0x0C1, 0x0C7, 0x061, 0x02A, 0x046, 0x0D4, 0x06F, 0x01E, 0x088, 0x00F, 0x0A6, 0x06A, 0x0A5,
0x011, 0x0BD, 0x0EB, 0x01D, 0x092, 0x069,
0x0B8, 0x084, 0x051, 0x0FA, 0x0DF, 0x04F, 0x016, 0x0AC, 0x0C3, 0x090, 0x0CE, 0x036, 0x01C, 0x05C,
0x038, 0x0DC, 0x071, 0x052, 0x05E, 0x07F,
0x0CC, 0x0B0, 0x013, 0x049, 0x0ED, 0x03D, 0x095, 0x081, 0x075, 0x085, 0x098, 0x0A5, 0x027, 0x02E, 0x020,
0x005, 0x019, 0x099, 0x0F4, 0x0F6,
0x006, 0x061, 0x0F1, 0x096, 0x0B9, 0x08C, 0x09B, 0x00E, 0x03E, 0x09C, 0x0EA, 0x050, 0x05F, 0x0D5,
0x0AC, 0x016, 0x0B1, 0x0D2, 0x0DB, 0x018,
0x010, 0x06A, 0x070, 0x03F, 0x03B, 0x0D8, 0x093, 0x07C, 0x0FF, 0x0FD, 0x06C, 0x0E3, 0x0C1, 0x0E9,
0x065, 0x026, 0x00F, 0x082, 0x017, 0x0B7,
0x0F1, 0x037, 0x001, 0x062, 0x06B, 0x0E5, 0x02F, 0x0AE, 0x066, 0x0CF, 0x027, 0x0CA, 0x078, 0x06D,
0x04A, 0x0E6, 0x00D, 0x058, 0x016, 0x0B0,
0x07E, 0x07F, 0x051, 0x0EA, 0x0CB, 0x079, 0x09E, 0x06E, 0x051, 0x0E0, 0x0F6, 0x055, 0x068, 0x0B4,
0x0EB, 0x0FC, 0x07A, 0x047, 0x078, 0x098,
0x09F, 0x0F5, 0x0E1, 0x0D6, 0x08E, 0x0C3, 0x0B5, 0x0A9, 0x0D9, 0x062, 0x0BC, 0x037, 0x08A, 0x056,
0x0E8, 0x01F, 0x01D, 0x0BE, 0x04B, 0x0D3,
//control values, $s576 - $s583
0x002, 0x020, 0x004, 0x040, 0x010, 0x014, 0x053, 0x140,
// empty data
0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000,
//temporary register, $t0 - $t24
0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000,
0x000, 0x000, 0x000, 0x000, 0x000,
0x000, 0x000, 0x000, 0x000, 0x000,
//empty data
0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000,
0x000, 0x000, 0x000, 0x000, 0x000,
0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000,
0x000, 0x000, 0x000, 0x000, 0x000,
0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000,
0x000, 0x000, 0x000, 0x000, 0x000,
0x000, 0x000, 0x000, 0x000, 0x000, 0x000,

//program
// Level 1 DWT
//0x2B0
0x255, 0xA55, 0x000, 0x256, 0xA56, 0x000, 0x25A, 0xA5A, 0x000, 0x25B, 0xA5B, 0x000, 0x801, 0x255,
0x000, 0xA42, 0x256, 0x000,
//0x2C2
0xA44, 0x25A, 0x000, 0xA43, 0x25B, 0x000, 0x802, 0x003, 0x000, 0x257, 0xA57, 0x000, 0x258, 0xA58, 0x000,
0x259, 0xA59, 0x000,
//0x2D4
0x800, 0x003, 0x018, 0x003, 0xA57, 0x000, 0xA42, 0x257, 0x006, 0xA55, 0x258, 0x000, 0x801, 0x259, 0x7F7,
0x257, 0xA57, 0x000,
//0x2E6
0xA58, 0x257, 0x000, 0xA57, 0x002, 0x000, 0x801, 0x259, 0x00F, 0x803, 0x257, 0x000, 0xA42, 0x257, 0x006,
0xA55, 0x258, 0x000,
//0x2F8
0x801, 0x259, 0x7F7, 0xA58, 0x002, 0x000, 0xA56, 0x2C8, 0x000, 0xA56, 0x2C9, 0x000, 0xA56, 0x2D5, 0x000,
0xA56, 0x2D7, 0x000,
//0x30A
0xA56, 0x2EA, 0x000, 0xA56, 0x2EF, 0x000, 0xA56, 0x2FC, 0x000, 0xA55, 0x25B, 0x7B2, 0xA55, 0x25A,
0x7AC, 0xA44, 0x25A, 0x000,
//0x31C
0xA45, 0x2C8, 0x000, 0xA45, 0x2C9, 0x000, 0xA45, 0x2D5, 0x000, 0xA45, 0x2D7, 0x000, 0xA45, 0x2EA,
0x000, 0xA45, 0x2EF, 0x000,
//0x32E
0xA45, 0x2FC, 0x000, 0xA55, 0x25A, 0x7E8, 0xA42, 0x25A, 0x000, 0xA45, 0x25B, 0x000, 0x802, 0x042,
0x000, 0x257, 0xA57, 0x000,
//0x340
0x258, 0xA58, 0x000, 0x259, 0xA59, 0x000, 0x800, 0x042, 0x018, 0x042, 0xA57, 0x000, 0xA42, 0x257, 0x006,
0xA55, 0x258, 0x000,
//0x352
0x801, 0x259, 0x7F7, 0x257, 0xA57, 0x000, 0xA58, 0x257, 0x000, 0xA57, 0x002, 0x000, 0x801, 0x259, 0x00F,
0x842, 0x257, 0x000,
//0x364
0xA42, 0x257, 0x006, 0xA55, 0x258, 0x000, 0x801, 0x259, 0x7F7, 0xA58, 0x002, 0x000, 0xA55, 0x33A, 0x000,
0xA55, 0x33B, 0x000,
//0x376
0xA55, 0x347, 0x000, 0xA55, 0x349, 0x000, 0xA55, 0x35C, 0x000, 0xA55, 0x361, 0x000, 0xA55, 0x36E, 0x000,
0xA55, 0x25B, 0x7B2,
//0x388
0xA45, 0x25B, 0x000, 0xA5B, 0x33A, 0x000, 0xA5B, 0x33B, 0x000, 0xA5B, 0x347, 0x000, 0xA5B, 0x349,
0x000, 0xA5B, 0x35C, 0x000,
//0x39A
0xA5B, 0x361, 0x000, 0xA5B, 0x36E, 0x000, 0xA55, 0x25A, 0x797, 0xA44, 0x25A, 0x000, 0xA45, 0x33A,
0x000, 0xA45, 0x33B, 0x000,
//0x3AC
0xA45, 0x347, 0x000, 0xA45, 0x349, 0x000, 0xA45, 0x35C, 0x000, 0xA45, 0x361, 0x000, 0xA45, 0x36E, 0x000,
0xA55, 0x25A, 0x7E8,
// Level 2 DWT
//0x3BE
0x25B, 0xA5B, 0x000, 0x25C, 0xA5C, 0x000, 0x25D, 0xA5D, 0x000, 0xA45, 0x25D, 0x000, 0xA44, 0x25C,
0x000, 0xA42, 0x25A, 0x000,
//0x3D0
0xA46, 0x25B, 0x000, 0x802, 0x004, 0x000, 0x257, 0xA57, 0x000, 0x258, 0xA58, 0x000, 0x259, 0xA59, 0x000,
0x800, 0x004, 0x018,
//0x3E2
0x004, 0xA57, 0x000, 0xA42, 0x257, 0x006, 0xA55, 0x258, 0x000, 0x801, 0x259, 0x7F7, 0x257, 0xA57, 0x000,
0xA58, 0x257, 0x000,
//0x3F4
0xA57, 0x002, 0x000, 0x801, 0x259, 0x00F, 0x804, 0x257, 0x000, 0xA42, 0x257, 0x006, 0xA55, 0x258, 0x000,
0x801, 0x259, 0x7F7,
//0x406
0xA58, 0x002, 0x000, 0xA5C, 0x3D3, 0x000, 0xA5C, 0x3D4, 0x000, 0xA5C, 0x3E0, 0x000, 0xA5C, 0x3E2,
0x000, 0xA5C, 0x3F5, 0x000,
//0x418
0xA5C, 0x3FA, 0x000, 0xA5C, 0x407, 0x000, 0xA55, 0x25B, 0x7B2, 0xA5D, 0x3D3, 0x000, 0xA5D, 0x3D4,
0x000, 0xA5D, 0x3E0, 0x000,
//0x42A
0xA5D, 0x3E2, 0x000, 0xA5D, 0x3F5, 0x000, 0xA5D, 0x3FA, 0x000, 0xA5D, 0x407, 0x000, 0xA55, 0x25A,
0x797, 0xA44, 0x25A, 0x000,
//0x43C
0xA45, 0x3D3, 0x000, 0xA45, 0x3D4, 0x000, 0xA45, 0x3E0, 0x000, 0xA45, 0x3E2, 0x000, 0xA45, 0x3F5,
0x000, 0xA45, 0x3FA, 0x000,
//0x44E
0xA45, 0x407, 0x000, 0xA55, 0x25A, 0x7E8, 0xA43, 0x25B, 0x000, 0x802, 0x082, 0x000, 0x257, 0xA57, 0x000,
0x258, 0xA58, 0x000,
//0x460
0x259, 0xA59, 0x000, 0x800, 0x082, 0x018, 0x082, 0xA57, 0x000, 0xA42, 0x257, 0x006, 0xA55, 0x258, 0x000,
0x801, 0x259, 0x7F7,
//0x472
0x257, 0xA57, 0x000, 0xA58, 0x257, 0x000, 0xA57, 0x002, 0x000, 0x801, 0x259, 0x00F, 0x882, 0x257, 0x000,
0xA42, 0x257, 0x006,
//0x484
0xA55, 0x258, 0x000, 0x801, 0x259, 0x7F7, 0xA58, 0x002, 0x000, 0xA56, 0x457, 0x000, 0xA56, 0x458, 0x000,
0xA56, 0x464, 0x000,
//0x496
0xA56, 0x466, 0x000, 0xA56, 0x479, 0x000, 0xA56, 0x47E, 0x000, 0xA56, 0x48B, 0x000, 0xA55, 0x25B,
0x7B2, 0xA45, 0x457, 0x000,
//0x4A8
0xA45, 0x458, 0x000, 0xA45, 0x464, 0x000, 0xA45, 0x466, 0x000, 0xA45, 0x479, 0x000, 0xA45, 0x47E, 0x000,
0xA45, 0x48B, 0x000,

// CRS Encoding
//0x4BA
0x25C, 0xA5C, 0x000, 0xA48, 0x25C, 0x000, 0x25D, 0xA5D, 0x000, 0xA55, 0x4C0, 0x000, 0xA55, 0x4C1,
0x000, 0xA55, 0x25C, 0x7F4,
//0x4CC
0xA48, 0x4C0, 0x000, 0xA48, 0x4C1, 0x000, 0xA43, 0x25B, 0x000, 0xA42, 0x25A, 0x000, 0x259, 0xA59,
0x000, 0xA45, 0x259, 0x000,
//0x4DE
0xA42, 0x25C, 0x000, 0xA47, 0x25D, 0x000, 0xA46, 0x25E, 0x000, 0x002, 0xA5F, 0x000, 0xA5F, 0xA5F,
0x000, 0x102, 0x25F, 0x000,
//0x4F0
0x25F, 0xA60, 0x000, 0x25F, 0xA5F, 0x000, 0xA56, 0x4E7, 0x000, 0xA55, 0x4ED, 0x000, 0xA55, 0x25E,
0x7E8, 0xA55, 0x4F1, 0x000,
//0x502
0xA43, 0x4E7, 0x000, 0xA55, 0x25D, 0x7DC, 0xA49, 0x4ED, 0x000, 0xA5B, 0x4E7, 0x000, 0xA55, 0x25C,
0x7D0, 0xA59, 0x4E7, 0x000,
//0x514
0xA55, 0x25A, 0x7C7, 0xA44, 0x25C, 0x000, 0xA47, 0x4F1, 0x000, 0xA45, 0x4E7, 0x000, 0xA55, 0x25C,
0x7F7, 0x801, 0x25C, 0x2D9,

//0x526
0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000,
0x000, 0x000, 0x000
```

```
} with {block=1};

static macro proc RunDWTCRSURISC();
static macro proc Sleep (Milliseconds);

void main(void)
{
    // Signals
    signal unsigned int 12 Sig_MemOutput_RS232;

    // Camera outputs to FPGA
    unsigned int 16 Pixel;
    unsigned int 11 X;
    unsigned int 11 Y;

    // RS232 Data read and send
    unsigned int 8 RS232SEND;
    unsigned int 8 RS232READ;

    unsigned int 13 t;
    unsigned int 1 RUN_NEXT;
    unsigned int 1 RUN;
    unsigned int 9 sendpointer;
    unsigned int 11 CRS_t;
    unsigned int 8 WriteMem;

    par
    {
        RC10CameraRun(OV9650_RGB565_QQVGA_LowLight, ClockRate);
        RC10RS232Run(RC10RS232_19200Baud, RC10RS232ParityNone,
                RC10RS232FlowControlNone, ClockRate);

        seq
        {
            RC10CameraSetMode(OV9650_RGB565_QQVGA_LowLight);

            while(1)
            {
                par{
                    t = 1;
                    RUN_NEXT = 0;
                    RUN = 1;
                }

                do
                {
                    RC10CameraReadRGB565(&X,&Y,&Pixel);
                } while( !(X==159 && Y==119) );

                do
                {
                    par
                    {
                        RC10CameraReadRGB565(&X,&Y,&Pixel);
                        if(Y<64)
                        {
                            if(X<64)
                            {
                                par
                                {
                                    data[t] = Pixel[10:5];
                                    t = t + 1;
                                }
                            }
                            else
                                delay;
                        }
                        else
                            delay;
                    }
                } while( !(X==64 && Y==64) );

                t = 1;
                //RC10RS232Write(0xFF);
                while(RUN == 1)
                {
                    par
                    {
                        CRS_t = 2;
                        sendpointer = 0;
                    }

                    while(sendpointer<256)
                    {
                        par
                        {
                            //Read Image Data
                            // increase to 8 bits
                            WriteMem = ( (unsigned 8) (0[1:0] @
                                data[t]) );

                            // Stop the DWT CRS
                            if( t==4096)
                            {
                                RUN = 0;
                            }
                            else
                                delay;
                        }

                        par
                        {
                            // Transfer data to DWT CRS MISC memory
                            // increase to 12 bits
                            Memory[CRS_t] = 0[3:0] @ WriteMem;
                            //RC10RS232Write(CRS_t[7:0]);

                            sendpointer++;
                            t++;
                            CRS_t++;

                        }

                    }


                    RunDWTCRSURISC();

                    par{
                        sendpointer = 0;
                        //CRS_t = 688;       // program memory
                        //CRS_t = 0x002;     // input data
                        CRS_t = 0x260;       // output data
                    }

                    //while(CRS_t<1318)      // program memory
                    //while(CRS_t<=257)      // input data
                    while(CRS_t<=687)        // output data
                    {
                        sendpointer = 0;

                        RC10RS232Write(0x52); //R
                        RC10RS232Write(0x41); //A
                        RC10RS232Write(0x57); //W
```

```
                        RC10RS232Write(0x3A); //:

                        while(sendpointer<20)
                        {
                            par
                            {
                                Sig_MemOutput_RS232 =
                                    Memory[CRS_t];

    RC10RS232Write(Sig_MemOutput_RS232[7:0]);
                                //RC10RS232Write(CRS_t[7:0]);

                                // Increase counter
                                CRS_t++;
                                sendpointer++;
                            }
                        }

                    }
                }

                while( RUN_NEXT!=1 )
                {
                    par
                    {
                        RC10RS232Read(&RS232READ);
                        RUN_NEXT = RS232READ[7] & RS232READ[6] &
                            RS232READ[5] & RS232READ[4] &
                            RS232READ[3] & RS232READ[2] &
                            RS232READ[1] & RS232READ[0];
                    }
                }
            }
        }
    }
}


static macro proc RunDWTCRSURISC()
{
    signal unsigned int RegWidth Sig_PC_Out;
    signal unsigned int RegWidth Sig_R_Out;
    signal unsigned int RegWidth Sig_R_Inv_In;
    signal unsigned int RegWidth Sig_R_Inv;
    signal unsigned int RegWidth Sig_Add_MEM;
    signal unsigned int RegWidth Sig_Add_In1;
    signal unsigned int RegWidth Sig_Add_In2;
    signal unsigned int RegWidth Sig_Add_Out;
    signal unsigned int RegWidth Sig_GF_In1;
    signal unsigned int RegWidth Sig_GF_In2;
    signal unsigned int RegWidth Sig_GF_Out;
    signal unsigned int RegWidth Sig_XOR_In1;
    signal unsigned int RegWidth Sig_XOR_In2;
    signal unsigned int RegWidth Sig_XOR_Out;
    signal unsigned int RegWidth Sig_MDR_In;
    signal unsigned int RegWidth Sig_MAR_In;
    signal unsigned int RegWidth Sig_MAR_Out;
    signal unsigned int RegWidth Sig_11to8bit_In1;
    signal unsigned int RegWidth Sig_11to8bit_In2;
    signal unsigned int RegWidth Sig_11to8bit_Out;

    signal unsigned int MemWidth Sig_MDR_Out;
    signal unsigned int MemWidth Sig_MEM_Out;

    signal unsigned int 2 Sig_OPCODE;

    signal unsigned int 1 Sig_Add_N;
signal unsigned int 1 Sig_Add_Z;
signal unsigned int 1 Sig_R_Write;
signal unsigned int 1 Sig_PC_Write;
signal unsigned int 1 Sig_OP0_Write;
signal unsigned int 1 Sig_OP1_Write;
signal unsigned int 1 Sig_OP_Sel;
signal unsigned int 1 Sig_OP0;
signal unsigned int 1 Sig_OP1;
signal unsigned int 1 Sig_OP_Out_Sel;
signal unsigned int 1 Sig_OP_VAR;
signal unsigned int 1 Sig_ALU_A;
signal unsigned int 1 Sig_ALU_B0;
signal unsigned int 1 Sig_ALU_B1;
signal unsigned int 1 Sig_CIN;
signal unsigned int 1 Sig_MAR_Sel;
signal unsigned int 1 Sig_MAR_Write;
signal unsigned int 1 Sig_MDR_Write;
signal unsigned int 1 Sig_MEM_Read;
signal unsigned int 1 Sig_MEM_Write;
signal unsigned int 1 Sig_N_Write;
signal unsigned int 1 Sig_Z_Write;
signal unsigned int 1 Sig_GF_0;
signal unsigned int 1 Sig_GF_1;
signal unsigned int 1 Sig_GF_2;
signal unsigned int 1 Sig_GF_3;
signal unsigned int 1 Sig_GF_4;
signal unsigned int 1 Sig_GF_5;
signal unsigned int 1 Sig_GF_6;
signal unsigned int 1 Sig_GF_7;

// Registers
unsigned int RegWidth R;
unsigned int RegWidth PC;
unsigned int RegWidth MAR;

unsigned int MemWidth MDR;
unsigned int MemWidth MEM_Out;

// Counter for each instructions
unsigned int 4 counter;

unsigned int 1 OPCODE0;
unsigned int 1 OPCODE1;
unsigned int 1 N;
unsigned int 1 Z;
unsigned int 1 RUN;

// Set initial value of registers
par
{
        PC = 688;
        counter = 0;
        R = 0;
        MDR = 0;
        MAR = 0;
        OPCODE0 = 0;
        OPCODE1 = 0;
        N = 0;
        Z = 0;
        MEM_Out = 0;
        RUN = 1;
}

// architecture
while( RUN != 0 )
{
        par
```

```
{
        par
        {
                Sig_PC_Out = PC;
                Sig_R_Out = R;
                Sig_R_Inv = ~Sig_R_Inv_In;
                Sig_MAR_Out = MAR;
                Sig_MDR_Out = MDR;
                Sig_OP_VAR = Sig_MEM_Out[11:11];
        }

        // Program Counter
        if(Sig_PC_Write == 1)
                PC = Sig_Add_Out;
        else
                delay;

        // R register
        if(Sig_R_Write == 1)
                R = Sig_MEM_Out[10:0];
        else
                delay;

        // OPCODE
        par
        {
                // OPCODE0 register
                if(Sig_OP0_Write == 1)
                        OPCODE0 = Sig_OP0;
                else
                        delay;

                // OPCODE1 register
                if(Sig_OP1_Write == 1)
                        OPCODE1 = Sig_OP1;
                else
                        delay;

                // MUX OPCODE
                if(Sig_OP_Sel == 0)
                        Sig_OP1 = Sig_MEM_Out[11:11];
                else
                        Sig_OP0 = Sig_MEM_Out[11:11];

                // MUX OPCODE Out
                if(Sig_OP_Out_Sel == 1)
                        Sig_OPCODE = 2;
                else
                        Sig_OPCODE = OPCODE1 @ OPCODE0;

                // R register output MUX
                if(Sig_OPCODE == 0)
                        Sig_GF_In2 = Sig_R_Out;
                else if(Sig_OPCODE == 1)
                        Sig_XOR_In2 = Sig_R_Out;
                else if(Sig_OPCODE == 2)
                        Sig_R_Inv_In = Sig_R_Out;
                else
                        Sig_11to8bit_In2 = Sig_R_Out;


                // MEM output MUX
                if(Sig_OPCODE == 0)
                        Sig_GF_In1 = Sig_MEM_Out[10:0];
                else if(Sig_OPCODE == 1)
                        Sig_XOR_In1 = Sig_MEM_Out[10:0];
                else if(Sig_OPCODE == 2)
        Sig_Add_MEM = Sig_MEM_Out[10:0];
    else
        Sig_11to8bit_In1 = Sig_MEM_Out[10:0];

    // MDR input MUX
    if(Sig_OPCODE == 0)
        Sig_MDR_In = Sig_GF_Out;
    else if(Sig_OPCODE == 1)
        Sig_MDR_In = Sig_XOR_Out;
    else if(Sig_OPCODE == 2)
        Sig_MDR_In = Sig_Add_Out;
    else
        Sig_MDR_In = Sig_11to8bit_Out;
}

// Adder
par
{
    Sig_Add_Out = Sig_Add_In1 + Sig_Add_In2 + (0[9:0] @ Sig_CIN);

    // Negative Flag
    if(Sig_Add_Out[10:10] == 1)
        Sig_Add_N = 1;
    else
        Sig_Add_N = 0;

    // Zero Flag
    if(Sig_Add_Out == 0)
        Sig_Add_Z = 1;
    else
        Sig_Add_Z = 0;

    // MUX ALU_B
    if(Sig_ALU_B1 == 0 && Sig_ALU_B0 == 0)
        Sig_Add_In2 = Sig_PC_Out;
    else if(Sig_ALU_B1 == 0 && Sig_ALU_B0 == 1)
        Sig_Add_In2 = Sig_R_Inv;
    else
        Sig_Add_In2 = 0;

    // MUX ALU_A
    if(Sig_ALU_A == 0)
        Sig_Add_In1 = Sig_Add_MEM;
    else
        Sig_Add_In1 = Sig_PC_Out;
}

// N register
if(Sig_N_Write == 1)
    N = Sig_Add_N;
else
    delay;

// Z register
if(Sig_Z_Write == 1)
    Z = Sig_Add_Z;
else
    delay;

// MDR Register
if(Sig_MDR_Write == 1)
    MDR = Sig_OP_VAR @ Sig_MDR_In;
else
    delay;

// MAR Register
if(Sig_MAR_Write == 1)
```

```
        MAR = Sig_MAR_In;
else
        delay;

// MAR MUX
if(Sig_MAR_Sel == 1)
        Sig_MAR_In = Sig_MEM_Out[10:0];
else
        Sig_MAR_In = Sig_Add_Out;

// Memory
par
{
        if(Sig_MEM_Read == 1)
        {
                par
                {
                        Sig_MEM_Out = Memory[Sig_MAR_Out];
                        MEM_Out = Sig_MEM_Out;
                }
        }
        else if(Sig_MEM_Write == 1)
                Memory[Sig_MAR_Out] = Sig_MDR_Out;
        else
                Sig_MEM_Out = MEM_Out;
}

// 11 to 8-bit Conversion
Sig_11to8bit_Out = 0[2:0] @ Sig_11to8bit_In1[10] @ Sig_11to8bit_In1[6:0];

// XOR
Sig_XOR_Out = Sig_XOR_In1 ^ Sig_XOR_In2;

// GF Multiplier (Direct wire connections)
Sig_GF_0 = (Sig_GF_In1[0] & Sig_GF_In2[0])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[7]) ^ (Sig_GF_In1[7] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[7]) ^ (Sig_GF_In1[7] & Sig_GF_In2[5])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[6]) ^ (Sig_GF_In1[5] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[1]) ^ (Sig_GF_In1[6] & Sig_GF_In2[2])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[3]) ^ (Sig_GF_In1[4] & Sig_GF_In2[4])
        ^ (Sig_GF_In1[3] & Sig_GF_In2[5]) ^ (Sig_GF_In1[2] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[1] & Sig_GF_In2[7]);
Sig_GF_1 = (Sig_GF_In1[1] & Sig_GF_In2[0])
        ^ (Sig_GF_In1[0] & Sig_GF_In2[1]) ^ (Sig_GF_In1[7] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[6]) ^ (Sig_GF_In1[6] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[2]) ^ (Sig_GF_In1[6] & Sig_GF_In2[3])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[4]) ^ (Sig_GF_In1[4] & Sig_GF_In2[5])
       ^ (Sig_GF_In1[3] & Sig_GF_In2[6]) ^ (Sig_GF_In1[2] & Sig_GF_In2[7]);
Sig_GF_2 = (Sig_GF_In1[2] & Sig_GF_In2[0])
        ^ (Sig_GF_In1[1] & Sig_GF_In2[1]) ^ (Sig_GF_In1[0] & Sig_GF_In2[2])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[6]) ^ (Sig_GF_In1[6] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[5]) ^ (Sig_GF_In1[6] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[7]) ^ (Sig_GF_In1[7] & Sig_GF_In2[3])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[4]) ^ (Sig_GF_In1[5] & Sig_GF_In2[5])
        ^ (Sig_GF_In1[4] & Sig_GF_In2[6]) ^ (Sig_GF_In1[3] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[1]) ^ (Sig_GF_In1[6] & Sig_GF_In2[2])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[3]) ^ (Sig_GF_In1[4] & Sig_GF_In2[4])
        ^ (Sig_GF_In1[3] & Sig_GF_In2[5]) ^ (Sig_GF_In1[2] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[1] & Sig_GF_In2[7]);
Sig_GF_3 = (Sig_GF_In1[3] & Sig_GF_In2[0])
        ^ (Sig_GF_In1[2] & Sig_GF_In2[1]) ^ (Sig_GF_In1[1] & Sig_GF_In2[2])
        ^ (Sig_GF_In1[0] & Sig_GF_In2[3]) ^ (Sig_GF_In1[7] & Sig_GF_In2[5])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[6]) ^ (Sig_GF_In1[5] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[4]) ^ (Sig_GF_In1[6] & Sig_GF_In2[5])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[6]) ^ (Sig_GF_In1[4] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[2]) ^ (Sig_GF_In1[6] & Sig_GF_In2[3])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[4]) ^ (Sig_GF_In1[4] & Sig_GF_In2[5])
        ^ (Sig_GF_In1[3] & Sig_GF_In2[6]) ^ (Sig_GF_In1[2] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[1]) ^ (Sig_GF_In1[6] & Sig_GF_In2[2])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[3]) ^ (Sig_GF_In1[4] & Sig_GF_In2[4])
        ^ (Sig_GF_In1[3] & Sig_GF_In2[5]) ^ (Sig_GF_In1[2] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[1] & Sig_GF_In2[7]);
Sig_GF_4 = (Sig_GF_In1[4] & Sig_GF_In2[0])
        ^ (Sig_GF_In1[3] & Sig_GF_In2[1]) ^ (Sig_GF_In1[2] & Sig_GF_In2[2])
        ^ (Sig_GF_In1[1] & Sig_GF_In2[3]) ^ (Sig_GF_In1[0] & Sig_GF_In2[4])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[7]) ^ (Sig_GF_In1[7] & Sig_GF_In2[3])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[4]) ^ (Sig_GF_In1[5] & Sig_GF_In2[5])
        ^ (Sig_GF_In1[4] & Sig_GF_In2[6]) ^ (Sig_GF_In1[3] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[2]) ^ (Sig_GF_In1[6] & Sig_GF_In2[3])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[4]) ^ (Sig_GF_In1[4] & Sig_GF_In2[5])
        ^ (Sig_GF_In1[3] & Sig_GF_In2[6]) ^ (Sig_GF_In1[2] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[1]) ^ (Sig_GF_In1[6] & Sig_GF_In2[2])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[3]) ^ (Sig_GF_In1[4] & Sig_GF_In2[4])
        ^ (Sig_GF_In1[3] & Sig_GF_In2[5]) ^ (Sig_GF_In1[2] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[1] & Sig_GF_In2[7]);
Sig_GF_5 = (Sig_GF_In1[5] & Sig_GF_In2[0])
        ^ (Sig_GF_In1[4] & Sig_GF_In2[1]) ^ (Sig_GF_In1[3] & Sig_GF_In2[2])
        ^ (Sig_GF_In1[2] & Sig_GF_In2[3]) ^ (Sig_GF_In1[1] & Sig_GF_In2[4])
        ^ (Sig_GF_In1[0] & Sig_GF_In2[5]) ^ (Sig_GF_In1[7] & Sig_GF_In2[4])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[5]) ^ (Sig_GF_In1[5] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[4] & Sig_GF_In2[7]) ^ (Sig_GF_In1[7] & Sig_GF_In2[3])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[4]) ^ (Sig_GF_In1[5] & Sig_GF_In2[5])
        ^ (Sig_GF_In1[4] & Sig_GF_In2[6]) ^ (Sig_GF_In1[3] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[2]) ^ (Sig_GF_In1[6] & Sig_GF_In2[3])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[4]) ^ (Sig_GF_In1[4] & Sig_GF_In2[5])
       ^ (Sig_GF_In1[3] & Sig_GF_In2[6]) ^ (Sig_GF_In1[2] & Sig_GF_In2[7]);
Sig_GF_6 = (Sig_GF_In1[6] & Sig_GF_In2[0])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[1]) ^ (Sig_GF_In1[4] & Sig_GF_In2[2])
        ^ (Sig_GF_In1[3] & Sig_GF_In2[3]) ^ (Sig_GF_In1[2] & Sig_GF_In2[4])
        ^ (Sig_GF_In1[1] & Sig_GF_In2[5]) ^ (Sig_GF_In1[0] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[5]) ^ (Sig_GF_In1[6] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[5] & Sig_GF_In2[7]) ^ (Sig_GF_In1[7] & Sig_GF_In2[4])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[5]) ^ (Sig_GF_In1[5] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[4] & Sig_GF_In2[7]) ^ (Sig_GF_In1[7] & Sig_GF_In2[3])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[4]) ^ (Sig_GF_In1[5] & Sig_GF_In2[5])
       ^ (Sig_GF_In1[4] & Sig_GF_In2[6]) ^ (Sig_GF_In1[3] & Sig_GF_In2[7]);
Sig_GF_7 = (Sig_GF_In1[7] & Sig_GF_In2[0])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[1]) ^ (Sig_GF_In1[5] & Sig_GF_In2[2])
        ^ (Sig_GF_In1[4] & Sig_GF_In2[3]) ^ (Sig_GF_In1[3] & Sig_GF_In2[4])
        ^ (Sig_GF_In1[2] & Sig_GF_In2[5]) ^ (Sig_GF_In1[1] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[0] & Sig_GF_In2[7]) ^ (Sig_GF_In1[7] & Sig_GF_In2[6])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[7]) ^ (Sig_GF_In1[7] & Sig_GF_In2[5])
        ^ (Sig_GF_In1[6] & Sig_GF_In2[6]) ^ (Sig_GF_In1[5] & Sig_GF_In2[7])
        ^ (Sig_GF_In1[7] & Sig_GF_In2[4]) ^ (Sig_GF_In1[6] & Sig_GF_In2[5])
       ^ (Sig_GF_In1[5] & Sig_GF_In2[6]) ^ (Sig_GF_In1[4] & Sig_GF_In2[7]);
Sig_GF_Out = 0[2:0] @ Sig_GF_7 @ Sig_GF_6 @ Sig_GF_5 @ Sig_GF_4
                @ Sig_GF_3 @ Sig_GF_2 @ Sig_GF_1 @ Sig_GF_0;

// Control Signal
par
{
        // Clock counter
        if(counter == 8)
                counter = 0;
        else
                counter++;

        //if (Sig_MAR_Out == 0x273)
        //      delay;
        //else
        //      delay;


        //if(PC >= 0x4E4)
```

```
                //  delay;
                //else
                //  delay;

                // Run or stop
                if(PC == 2047)
                    RUN = 0;
                else
                    delay;

                Sig_ALU_A = (~counter[3] & ~counter[2]) | (~counter[2] & ~counter[1]
                        & ~counter[0]) | (~counter[3] & ~counter[0]);
                Sig_ALU_B1 = Sig_ALU_A;
                Sig_ALU_B0 = (~counter[3] & counter[2] & ~counter[1] & counter[0]);
                Sig_CIN = (~counter[3] & counter[2] & ~counter[1]) | (~counter[3]
                        & ~counter[1] & counter[0])
                    | (counter[3] & ~counter[2] & ~counter[1] & ~counter[0]);
                Sig_MAR_Sel = (~counter[3] & counter[2] & ~counter[1])
                    | (~counter[3] & ~counter[1] & counter[0]);
                Sig_PC_Write = (~counter[3] & counter[2] & counter[1] & counter[0]
                        & N)
                    | (~counter[3] & ~counter[2] & ~counter[1] & counter[0])
                    | (~counter[3] & counter[2] & ~counter[1] & ~counter[0])
                    | (counter[3] & ~counter[2] & ~counter[1] & ~counter[0]);
                Sig_R_Write = (~counter[3] & ~counter[2] & counter[1] & ~counter[0]);
                Sig_Z_Write = (~counter[3] & ~counter[2] & ~counter[1] & ~counter[0]);
                Sig_N_Write = (~counter[3] & counter[2] & ~counter[1] & counter[0]);
                Sig_MAR_Write = (~counter[3] & ~counter[2] & counter[0])
                    | (~counter[3] & counter[2] & ~counter[0])
                    | (~counter[3] & ~counter[1] & ~counter[0]);
                Sig_MDR_Write = (~counter[3] & counter[2] & ~counter[1]
                        & counter[0]);
                Sig_MEM_Read = (~counter[3] & counter[2] & ~counter[1])
                    | (~counter[3] & counter[2] & counter[0])
                    | (~counter[3] & ~counter[1] & counter[0])
                    | (~counter[3] & ~counter[2] & counter[1] & ~counter[0]);
                Sig_MEM_Write = (~counter[3] & counter[2] & counter[1]
                        & ~counter[0]);
                Sig_OP1_Write = (~counter[3] & ~counter[2] & ~counter[1]
                        & counter[0]);
                Sig_OP0_Write = (~counter[3] & counter[2] & ~counter[1]
                        & ~counter[0]);
                Sig_OP_Sel = Sig_OP0_Write;
                Sig_OP_Out_Sel = (~counter[3] & ~counter[2])
                    | (~counter[3] & ~counter[1] & ~counter[0]);
            }

        }
    }

}


static macro proc Sleep (Milliseconds)
{
   macro expr Cycles = (ClockRate * Milliseconds) / 1000;
   unsigned (log2ceil (Cycles)) Count;

   Count = 0;
   do
   {
      Count++;
   }
   while (Count != Cycles - 1);
}
```

A.3 CRS MISC ARCHITECTURE IN VHDL

A.3.1 Top Level CRS MISC Architecture - CRSMISC.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity CRSMISC is
  Port ( Seg0, Seg1 : out  STD_LOGIC_VECTOR (6 downto 0);
      RamAdd : out  STD_LOGIC_VECTOR (7 downto 0);
      CLK : in  STD_LOGIC;
      UP : in  STD_LOGIC;
      DOWN : in  STD_LOGIC;
                          EnaRead : in STD_LOGIC);
end CRSMISC;

architecture Behavioral of CRSMISC is

        -- Components
        -- Block RAM
        component memory
  port (
  addra: IN std_logic_VECTOR(8 downto 0);
  addrb: IN std_logic_VECTOR(8 downto 0);
  clka: IN std_logic;
  clkb: IN std_logic;
  dina: IN std_logic_VECTOR(9 downto 0);
  douta: OUT std_logic_VECTOR(9 downto 0);
  doutb: OUT std_logic_VECTOR(9 downto 0);
  ena: IN std_logic;
  enb: IN std_logic;
  wea: IN std_logic);
        end component;

        -- Controls
        component controls
                port(   CLK : in std_logic;
                                N : in std_logic;
                                ALU_A : out std_logic;
                                ALU_B0 : out std_logic;
                                ALU_B1 : out std_logic;
                                CIN : out std_logic;
                                MAR_SEL : out std_logic;
                                PC_WRITE : out std_logic;
                                R_WRITE : out std_logic;
                                Z_WRITE : out std_logic;
                                N_WRITE : out std_logic;
                                MAR_WRITE : out std_logic;
                                MDR_WRITE : out std_logic;
                                MEM_READ : out std_logic;
                                MEM_WRITE : out std_logic;
                                OP_OUT_SEL : out std_logic;
                                OP0_WRITE : out std_logic;
                                OP1_WRITE : out std_logic;
                                OP_SEL : out std_logic);
        end component;
```

```
-- MUX
-- MUX2-1 1-Bit
component MUX21 is
        port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
                        B : in  STD_LOGIC_VECTOR (7 downto 0);
                        SEL : in STD_LOGIC;
                        C : out  STD_LOGIC_VECTOR (7 downto 0));
end component;
-- MUX2-1 2-Bit
component MUX22 is
        port ( A : in  STD_LOGIC_VECTOR (1 downto 0);
                        B : in  STD_LOGIC_VECTOR (1 downto 0);
                        SEL : in STD_LOGIC;
                        C : out  STD_LOGIC_VECTOR (1 downto 0));
end component;
-- MUX4-1 9-Bit
component MUX49 is
        port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
                        B : in  STD_LOGIC_VECTOR (8 downto 0);
                        C : in  STD_LOGIC_VECTOR (8 downto 0);
                        D : out  STD_LOGIC_VECTOR (8 downto 0);
                        SEL : in  STD_LOGIC_VECTOR (1 downto 0));
end component;
-- MUX1-4 9-Bit
component MUX94 is
        port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
                        SEL : in  STD_LOGIC_VECTOR (1 downto 0);
                        B : out  STD_LOGIC_VECTOR (8 downto 0);
                        C : out  STD_LOGIC_VECTOR (8 downto 0);
                        D : out  STD_LOGIC_VECTOR (8 downto 0));
end component;
-- MUX1-2 1-Bit
component MUX11 is
        port ( A : in  STD_LOGIC;
                        SEL : in  STD_LOGIC;
                        B : out  STD_LOGIC;
                        C : out  STD_LOGIC);
end component;
-- MUX2-1 9-Bit
component MUX29 is
        port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
                        B : in  STD_LOGIC_VECTOR (8 downto 0);
                        SEL : in  STD_LOGIC;
                        C : out  STD_LOGIC_VECTOR (8 downto 0));
end component;

-- Registers
-- 8-Bit Register
component REG is
        port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
                        B : out  STD_LOGIC_VECTOR (8 downto 0);
                        CLK : in  STD_LOGIC;
                        ENA : in  STD_LOGIC);
end component;
-- 1-Bit Register
component REG1BIT is
        Port ( A : in  STD_LOGIC;
                        B : out  STD_LOGIC;
                        CLK : in  STD_LOGIC;
                        ENA : in  STD_LOGIC);
end component;
-- 10-Bit Register
component REG10BIT is
        Port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
                        B : out  STD_LOGIC_VECTOR (9 downto 0);
                        CLK : in  STD_LOGIC;
                        ENA : in  STD_LOGIC);
```

```
end component;
-- PC Register
component REGPC is
        port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
                        B : out  STD_LOGIC_VECTOR (8 downto 0);
                        CLK : in  STD_LOGIC;
                        ENA : in  STD_LOGIC);
end component;

-- SBN
component SBN is
        Port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
                        B : in  STD_LOGIC_VECTOR (8 downto 0);
                        CIN : in  STD_LOGIC;
                        N : out  STD_LOGIC;
                        Z : out  STD_LOGIC;
                        O : out STD_LOGIC_VECTOR (8 downto 0));
end component;

-- 8-Bit XOR
component GF28Add is
        Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
                        b : in  STD_LOGIC_VECTOR (7 downto 0);
                        c : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

-- GF28
component GF28 is
        Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
                        b : in  STD_LOGIC_VECTOR (7 downto 0);
                        c : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

-- D4to7 Conversion
component D4to7 is
        Port ( Q : in  STD_LOGIC_VECTOR (3 downto 0);
                        Seg : out STD_LOGIC_VECTOR (6 downto 0));
end component;


-- Signals
signal iCount1 : std_logic := '0';
signal iClock2 : std_logic;
signal iCount23 : std_logic_vector(22 downto 0);
signal iClock23 : std_logic;
signal iRamAdd : std_logic_vector(8 downto 0);
signal iRamRead : std_logic_vector(9 downto 0);

-- Program Counter
signal iPC : std_logic_vector(8 downto 0) := (others=>'0');
signal PC : std_logic_vector(8 downto 0) := (others=>'0');
signal PC_Write : std_logic;
-- R Register
signal iR : std_logic_vector(8 downto 0);
signal R : std_logic_vector(8 downto 0);
signal R_Write : std_logic;

-- OPCODE
-- MUX
signal OP_SEL : std_logic;
-- OP0 Register
signal iOP0 : std_logic;
signal OP0_Write : std_logic;
-- OP1 Register
signal iOP1 : std_logic;
signal OP1_Write : std_logic;
-- OPCODE output
```

```
signal OPCODE : std_logic_vector(1 downto 0);
signal OP_OUT_SEL : std_logic;
signal oOPCODE : std_logic_vector(1 downto 0);

-- MEMORY
-- MDR register
signal iMDR : std_logic_vector(9 downto 0);
signal MDR : std_logic_vector(9 downto 0);
signal MDR_Write : std_logic;
-- MAR
signal iMAR : std_logic_vector(8 downto 0);
signal MAR : std_logic_vector(8 downto 0);
signal MAR_Write : std_logic;
signal MAR_SEL : std_logic;
-- MEM Output
signal oMemory : std_logic_vector(9 downto 0);
-- MEM Controls
signal MEM_READ : std_logic;
signal MEM_WRITE : std_logic;
signal MEM_ENA : std_logic;
signal MEM_WEA : std_logic;

-- SBN/ALU Block
-- N register
signal i_N : std_logic;
signal N : std_logic;
signal N_Write : std_logic;
-- Z register
signal iZ : std_logic;
signal Z : std_logic;
signal Z_Write : std_logic;
-- ALU MUX
-- ALU MUX A
signal iMUXALUA0 : std_logic_vector(8 downto 0) := (others=>'0');
signal iMUXALUA1 : std_logic_vector(8 downto 0) := (others=>'0');
signal MUXALUA : std_logic_vector(8 downto 0) := (others=>'0');
signal MUXALUASEL : std_logic;
signal ALU_A : std_logic;
-- ALU MUX B
signal iMUXALUB0 : std_logic_vector(8 downto 0) := (others=>'0');
signal iMUXALUB1 : std_logic_vector(8 downto 0) := (others=>'0');
signal iMUXALUB2 : std_logic_vector(8 downto 0) := (others=>'0');
signal MUXALUB : std_logic_vector(8 downto 0) := (others=>'0');
signal MUXALUBSEL : std_logic_vector(1 downto 0);
signal ALU_B : std_logic_vector(1 downto 0);
-- INV
signal iINV : std_logic_vector(8 downto 0) := (others=>'0');
signal oINV : std_logic_vector(8 downto 0) := (others=>'0');
-- CIN
signal CIN : std_logic;
-- Output
signal ADDER : std_logic_vector(8 downto 0) := (others=>'0');

-- GF Block
signal iGFA : std_logic_vector(8 downto 0) := (others=>'0');
signal iGFB : std_logic_vector(8 downto 0) := (others=>'0');
signal iGF : std_logic_vector(8 downto 0) := (others=>'0');

-- XOR Block
signal iXORA : std_logic_vector(8 downto 0) := (others=>'0');
signal iXORB : std_logic_vector(8 downto 0) := (others=>'0');
signal iXOR : std_logic_vector(8 downto 0) := (others=>'0');

-- Instruction MUX
-- MUX A
signal iMUXA : std_logic_vector(8 downto 0) := (others=>'0');
signal MUXA0 : std_logic_vector(8 downto 0) := (others=>'0');
```

```
        signal MUXA1 : std_logic_vector(8 downto 0) := (others=>'0');
        signal MUXA2 : std_logic_vector(8 downto 0) := (others=>'0');
        signal MUXASEL : std_logic_vector(1 downto 0);
        -- MUX B
        signal iMUXB : std_logic_vector(8 downto 0) := (others=>'0');
        signal MUXB0 : std_logic_vector(8 downto 0) := (others=>'0');
        signal MUXB1 : std_logic_vector(8 downto 0) := (others=>'0');
        signal MUXB2 : std_logic_vector(8 downto 0) := (others=>'0');
        signal MUXBSEL : std_logic_vector(1 downto 0);
        -- MUX Out
        signal MUXO : std_logic_vector(8 downto 0) := (others=>'0');
        signal iMUXO0 : std_logic_vector(8 downto 0) := (others=>'0');
        signal iMUXO1 : std_logic_vector(8 downto 0) := (others=>'0');
        signal iMUXO2 : std_logic_vector(8 downto 0) := (others=>'0');
        signal MUXOSEL : std_logic_vector(1 downto 0);

begin

        -- Controls Block
        Ctrl : controls
                        port map(
                                        CLK => iClock2,
                                        N => N,
                                        ALU_A => ALU_A,
                                        ALU_B0 => ALU_B(0),
                                        ALU_B1 => ALU_B(1),
                                        CIN => CIN,
                                        MAR_SEL => MAR_SEL,
                                        PC_WRITE => PC_Write,
                                        R_WRITE => R_Write,
                                        Z_WRITE => Z_Write,
                                        N_WRITE => N_Write,
                                        MAR_WRITE => MAR_Write,
                                        MDR_WRITE => MDR_Write,
                                        MEM_READ => MEM_READ,
                                        MEM_WRITE => MEM_WRITE,
                                        OP_OUT_SEL => OP_OUT_SEL,
                                        OP0_WRITE => OP0_Write,
                                        OP1_WRITE => OP1_Write,
                                        OP_SEL => OP_SEL);

        -- Registers
        -- PC
        PC_Reg : REGPC
                port map(
                                A => iPC,
                                B => PC,
                                CLK => iClock2,
                                ENA => PC_Write);
        iPC <= ADDER;

        -- R
        R_Reg : REG
                port map(
                                A => iR,
                                B => R,
                                CLK => iClock2,
                                ENA => R_Write);
        iR <= oMemory(8 downto 0);

        -- Z
        Z_Reg : REG1BIT
                port map(
                                A => iZ,
                                B => Z,
                                CLK => iClock2,
                                ENA => Z_Write);
```

```
-- N
N_Reg : REG1BIT
		port map(
					A => i_N,
					B => N,
					CLK => iClock2,
					ENA => N_Write);

-- OPCODE
-- OPCODE1
OPCODE1_Reg : REG1BIT
		port map(
					A => iOP1,
					B => OPCODE(1),
					CLK => iClock2,
					ENA => OP1_Write);

-- OPCODE0
OPCODE0_Reg : REG1BIT
		port map(
					A => iOP0,
					B => OPCODE(0),
					CLK => iClock2,
					ENA => OP0_Write);

-- oMUXOP
oMUXOP : MUX22
		port map (
					A => OPCODE,
					B => "10",
					SEL => OP_OUT_SEL,
					C => oOPCODE);

-- MUXOP
MUXOP : MUX11
		port map (
					A => oMemory(9),
					SEL => OP_SEL,
					B => iOP1,
					C => iOP0);


-- MUXA
MUXA : MUX94
		port map (
				A => iMUXA,
				SEL => MUXASEL,
				B => MUXA0,
				C => MUXA1,
				D => MUXA2);
MUXASEL <= oOPCODE;
iMUXA <= oMemory(8 downto 0);
iGFA <= MUXA0;
iXORA <= MUXA1;
IMUXALUA0 <= MUXA2;

-- MUXB
MUXB : MUX94
		port map (
				A => iMUXB,
				SEL => MUXBSEL,
				B => MUXB0,
				C => MUXB1,
				D => MUXB2);
MUXBSEL <= oOPCODE;
iMUXB <= R;
```

```
iGFB <= MUXB0;
iXORB <= MUXB1;
iINV <= MUXB2;

-- MUX OUT
MUXOUT : MUX49
            port map (
                                    A => iMUXO0,
                                    B => iMUXO1,
                                    C => iMUXO2,
                                    D => MUXO,
                                    SEL => MUXOSEL);
MUXOSEL <= oOPCODE;
iMUXO0 <= iGF;
iMUXO1 <= iXOR;
iMUXO2 <= ADDER;
iMDR <= oMemory(9) & MUXO;


-- GF
GF28MULT : GF28
            port map (
                                    a => iGFA(7 downto 0),
                                    b => iGFB(7 downto 0),
                                    c => iGF(7 downto 0));
iGF(8) <= '0';

-- XOR
GF28XOR : GF28Add
                        port map (
                                              a =>iXORA(7 downto 0),
                                              b =>iXORB(7 downto 0),
                                              c =>iXOR(7 downto 0));
iXOR(8) <= '0';


-- SBN
SBN_BLOCK : SBN
            port map (
                                    A => MUXALUA,
                                    B => MUXALUB,
                                    CIN => CIN,
                                    N => i_N,
                                    Z => iZ,
                                    O => ADDER);

-- ALU_A MUX
MUX_ALU_A : MUX29
                        port map (
                                    A => iMUXALUA0,
                                    B => iMUXALUA1,
                                    SEL => MUXALUASEL,
                                    C => MUXALUA);
MUXALUASEL <= ALU_A;
iMUXALUA1 <= PC;

-- ALU_B MUX
MUX_ALU_B : MUX49
            port map (
                                    A => iMUXALUB0,
                                    B => iMUXALUB1,
                                    C => iMUXALUB2,
                                    D => MUXALUB,
                                    SEL => MUXALUBSEL);
MUXALUBSEL <= ALU_B;
iMUXALUB0 <= PC;
iMUXALUB1 <= oINV;
```

```
	iMUXALUB2 <= "000000000";

	-- INV
	oINV <= NOT iINV;

	-- MDR
	MDR_Reg : REG10BIT
		 port map(
					A => iMDR,
					B => MDR,
					CLK => iClock2,
					ENA => MDR_Write);

	-- MUX MAR
	MUXMAR : MUX29
		 port map(
					A => ADDER,
					B => oMemory(8 downto 0),
					SEL => MAR_SEL,
					C => iMAR);

	-- MAR
	MAR_Reg : REG
		 port map(
					A => iMAR,
					B => MAR,
					CLK => iClock2,
					ENA => MAR_Write);
	--		iMAR <= ADDER;

	-- Block RAM
	Block_RAM : memory
port map (
  addra => MAR,
  addrb => iRamAdd,
  clka => CLK,
  clkb => CLK,
  dina => MDR,
  douta => oMemory,
  doutb => iRamRead,
  ena => MEM_ENA,
  enb => EnaRead,
  wea => MEM_WEA);
	-- Memory Control
	process(MEM_READ,MEM_WRITE,CLK)
	begin
		if MEM_WRITE = '1' then
			MEM_ENA <= '1';
			MEM_WEA <= MEM_WRITE;
		elsif MEM_READ = '1' then
			MEM_ENA <= MEM_READ;
			MEM_WEA <= '0';
		else
			MEM_ENA <= MEM_READ;
			MEM_WEA <= '0';
		end if;
	end process;

	-- RAM Address
	process(iClock23,UP,DOWN)
	begin

		if iClock23'event and iClock23 = '1' then
			if UP = '1' then
				iRamAdd <= iRamAdd + '1';
			elsif DOWN = '1' then
				iRamAdd <= iRamAdd - '1';
			else
				iRamAdd <= iRamAdd;
			end if;
		end if;

	end process;

	RamAdd <= iRamAdd(7 downto 0);
	-- RamRead <= iRamRead(7 downto 0);

	-- Seg0
	Seg70 : D4to7
		 port map(
					Q => iRamRead(3 downto 0),
					Seg => Seg0);

	-- Seg1
	Seg71 : D4to7
		 port map(
					Q => iRamRead(7 downto 4),
					Seg => Seg1);

	-- Clock24
	process(CLK)
	begin

		if CLK'event and CLK = '1' then
			iCount23 <= iCount23 + '1';
		end if;

	end process;
	-- Actual Implementation
		iClock23 <= iCount23(22);
	-- Simulation
	--	PiClock24 <= CLK;

	-- Clock2
	process(CLK)
	begin

		if CLK'event and CLK = '1' then
			if PC /= "111111111" then
				iCount1 <= NOT iCount1;
			end if;
		end if;

	end process;
	-- Actual Implementation
		iClock2 <= iCount1;
	-- Simulation
	-- iClock2 <= CLK;


end Behavioral;
```

A.3.2 Control Signals Combinational Circuit - controls.vhd

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

```
---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity controls is
    port(   CLK : in std_logic;
            N : in std_logic;
            ALU_A : out std_logic;
            ALU_B0 : out std_logic;
            ALU_B1 : out std_logic;
            CIN : out std_logic;
            MAR_SEL : out std_logic;
            PC_WRITE : out std_logic;
            R_WRITE : out std_logic;
            Z_WRITE : out std_logic;
            N_WRITE : out std_logic;
            MAR_WRITE : out std_logic;
            MDR_WRITE : out std_logic;
            MEM_READ : out std_logic;
            MEM_WRITE : out std_logic;
            OP_OUT_SEL : out std_logic;
            OP0_WRITE : out std_logic;
            OP1_WRITE : out std_logic;
            OP_SEL : out std_logic);
end controls;

architecture Behavioral of controls is

    signal iCount4 : std_logic_vector(3 downto 0) := X"8";

begin

    -- 4 Bit Counter
    process(CLK)
    begin

        if CLK'event and CLK = '1' then
            if iCount4 = "1000" then
                iCount4 <= (others=>'0');
            else
                iCount4 <= iCount4 + 1;
            end if;
        end if;

    end process;

    -- Output control signals
    ALU_A <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(0))) )
        OR ( (NOT(iCount4(2))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) );
    ALU_B0 <= (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) AND (iCount4(0));
    ALU_B1 <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(0))) )
        OR ( (NOT(iCount4(2))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) );
    CIN <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (iCount4(0)) )
        OR ( (iCount4(3)) AND (NOT(iCount4(2))) AND (NOT(iCount4(1))) AND
            (NOT(iCount4(0))) );
    MAR_SEL <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (iCount4(0)) );
    PC_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (iCount4(1)) AND (iCount4(0)) AND N )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
            AND (iCount4(0)) )
        OR ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
            AND (NOT(iCount4(0))) )
        OR ( (iCount4(3)) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
                                AND (NOT(iCount4(0))) );
        R_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (iCount4(1))
                                AND (NOT(iCount4(0))) );
        Z_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
                                AND (NOT(iCount4(0))) );
        N_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) AND (iCount4(0)) );
        MAR_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (iCount4(0)) )
                        OR ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(0))) )
                        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) );
        MDR_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
                                AND (iCount4(0)) );
        MEM_READ <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) )
                        OR ( (NOT(iCount4(3))) AND (iCount4(1)) AND (iCount4(0)) )
                        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (iCount4(0)) )
                        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (iCount4(1))
                                AND (NOT(iCount4(0))) );
        MEM_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (iCount4(1))
                                AND (NOT(iCount4(0))) );
        OP_OUT_SEL <= ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) )
                        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) );
        OP0_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
                                AND (NOT(iCount4(0))) );
        OP1_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
                                AND (iCount4(0)) );
        OP_SEL <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
                                AND (NOT(iCount4(0))) );

end Behavioral;
```

A.3.3 9-Bit Programme Counter Register - REGPC.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity REGPC is
   Port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
          B : out  STD_LOGIC_VECTOR (8 downto 0);
          CLK : in  STD_LOGIC;
          ENA : in  STD_LOGIC);
end REGPC;

architecture Behavioral of REGPC is

begin

        -- Register function
        process(CLK)

                -- Start of programme
                variable sig_data : std_logic_vector (8 downto 0) := "101101111"; -- 0x16F
                -- Test Last Line
                -- variable sig_data : std_logic_vector (8 downto 0) := "111011001";

        begin
```

```
		if CLK'event and CLK = '1' then
			if ENA = '1' then
				sig_data := A;
			end if;
		else
			sig_data := sig_data;
		end if;

		B <= sig_data;

	end process;

end Behavioral;
```

A.3.4 9-Bit Register - REG.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity REG is
    Port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
           B : out  STD_LOGIC_VECTOR (8 downto 0);
           CLK : in  STD_LOGIC;
           ENA : in  STD_LOGIC);
end REG;

architecture Behavioral of REG is

begin

	-- Register function
	process(CLK)

		variable sig_data : std_logic_vector (8 downto 0) := (others=>'0');

	begin

		if CLK'event and CLK = '1' then
			if ENA = '1' then
				sig_data := A;
			end if;
		else
			sig_data := sig_data;
		end if;

		B <= sig_data;

	end process;

end Behavioral;
```

A.3.5 1-Bit Register - REG1BIT.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity REG1BIT is
   Port ( A : in  STD_LOGIC;
          B : out  STD_LOGIC;
          CLK : in  STD_LOGIC;
          ENA : in  STD_LOGIC);
end REG1BIT;

architecture Behavioral of REG1BIT is

begin

            -- Register function
            process(CLK)

                        variable sig_data : std_logic := '0';

            begin

                        if CLK'event and CLK = '1' then
                                    if ENA = '1' then
                                                sig_data := A;
                                    end if;
                        else
                                    sig_data := sig_data;
                        end if;

                        B <= sig_data;

            end process;

end Behavioral;
```

A.3.6 2-Bit 2-To-1 Multiplexer - MUX22.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX22 is
   Port ( A : in  STD_LOGIC_VECTOR (1 downto 0);
          B : in  STD_LOGIC_VECTOR (1 downto 0);
                                    SEL : in STD_LOGIC;
          C : out  STD_LOGIC_VECTOR (1 downto 0));
end MUX22;
```

```
architecture Behavioral of MUX22 is

begin

        -- MUX2-1 2 bit
        process(SEL,A,B)
        begin

                if SEL = '0' then
                        C <= A;
                else
                        C <= B;
                end if;

        end process;

end Behavioral;
```

A.3.7 1-Bit 1-To-2 Multiplexer - MUX11.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX11 is
    Port ( A : in  STD_LOGIC;
           SEL : in  STD_LOGIC;
           B : out  STD_LOGIC;
           C : out  STD_LOGIC);
end MUX11;

architecture Behavioral of MUX11 is

begin

        -- MUX1-2 1-Bit
        process(SEL,A)
        begin

                if SEL = '0' then
                        B <= A;
                        C <= '0';
                else
                        B <= '0';
                        C <= A;
                end if;

        end process;

end Behavioral;
```

A.3.8 9-Bit 1-To-3 Multiplexer - MUX94.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX94 is
   Port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
          SEL : in  STD_LOGIC_VECTOR (1 downto 0);
          B : out  STD_LOGIC_VECTOR (8 downto 0);
          C : out  STD_LOGIC_VECTOR (8 downto 0);
          D : out  STD_LOGIC_VECTOR (8 downto 0));
end MUX94;

architecture Behavioral of MUX94 is

begin

            -- MUX1-4 9 bits
            process(SEL,A)
            begin

                        if SEL = "00" then
                                    B <= A;
                                    C <= "000000000";
                                    D <= "000000000";
                        elsif SEL = "01" then
                                    B <= "000000000";
                                    C <= A;
                                    D <= "000000000";
                        elsif SEL = "10" then
                                    B <= "000000000";
                                    C <= "000000000";
                                    D <= A;
                        else
                                    B <= "000000000";
                                    C <= "000000000";
                                    D <= "000000000";
                        end if;

            end process;

end Behavioral;
```

A.3.9 9-Bit 3-To-1 Multiplexer - MUX49.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```
entity MUX49 is
   Port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
          B : in  STD_LOGIC_VECTOR (8 downto 0);
                                    C : in  STD_LOGIC_VECTOR (8 downto 0);
          D : out  STD_LOGIC_VECTOR (8 downto 0);
          SEL : in  STD_LOGIC_VECTOR (1 downto 0));
end MUX49;

architecture Behavioral of MUX49 is

begin

            -- MUX4-1 9-bit
            process(SEL,A,B,C)
            begin

                        if SEL = "00" then
                                    D <= A;
                        elsif SEL = "01" then
                                    D <= B;
                        elsif SEL = "10" then
                                    D <= C;
                        else
                                    D <= "000000000";
                        end if;

            end process;

end Behavioral;
```

A.3.10 Functional Block GF(28) Multiplier - GF28.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity GF28 is
   Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
          b : in  STD_LOGIC_VECTOR (7 downto 0);
          c : out  STD_LOGIC_VECTOR (7 downto 0));
end GF28;

architecture Behavioral of GF28 is

            signal output_i : std_logic_vector (7 downto 0) := (others=>'0');

begin

            -- GF(2^8) Multiplier
            -- bit 7
            output_i(7) <= (a(7) AND b(0)) XOR (a(6) AND b(1)) XOR (a(5) AND b(2)) XOR (a(4) AND b(3))
                        XOR (a(3) AND b(4)) XOR (a(2) AND b(5)) XOR (a(1) AND b(6)) XOR (a(0) AND b(7))
                        XOR (a(7) AND b(6)) XOR (a(6) AND b(7)) XOR (a(7) AND b(5)) XOR (a(6) AND b(6))
                        XOR (a(5) AND b(7)) XOR (a(7) AND b(4)) XOR (a(6) AND b(5)) XOR (a(5) AND b(6))
                        XOR (a(4) AND b(7));
            -- bit 6
```

```
        output_i(6) <= (a(6) AND b(0)) XOR (a(5) AND b(1)) XOR (a(4) AND b(2)) XOR (a(3) AND b(3))
                XOR (a(2) AND b(4)) XOR (a(1) AND b(5)) XOR (a(0) AND b(6)) XOR (a(7) AND b(5))
                XOR (a(6) AND b(6)) XOR (a(5) AND b(7)) XOR (a(7) AND b(4)) XOR (a(6) AND b(5))
                XOR (a(5) AND b(6)) XOR (a(4) AND b(7)) XOR (a(7) AND b(3)) XOR (a(6) AND b(4))
                XOR (a(5) AND b(5)) XOR (a(4) AND b(6)) XOR (a(3) AND b(7));
        -- bit 5
        output_i(5) <= (a(5) AND b(0)) XOR (a(4) AND b(1)) XOR (a(3) AND b(2)) XOR (a(2) AND b(3))
                XOR (a(1) AND b(4)) XOR (a(0) AND b(5)) XOR (a(7) AND b(4)) XOR (a(6) AND b(5))
                XOR (a(5) AND b(6)) XOR (a(4) AND b(7)) XOR (a(7) AND b(3)) XOR (a(6) AND b(4))
                XOR (a(5) AND b(5)) XOR (a(4) AND b(6)) XOR (a(3) AND b(7)) XOR (a(7) AND b(2))
                XOR (a(6) AND b(3)) XOR (a(5) AND b(4)) XOR (a(4) AND b(5)) XOR (a(3) AND b(6))
                XOR (a(2) AND b(7));
        -- bit 4
        output_i(4) <= (a(4) AND b(0)) XOR (a(3) AND b(1)) XOR (a(2) AND b(2)) XOR (a(1) AND b(3))
                XOR (a(0) AND b(4)) XOR (a(7) AND b(7)) XOR (a(7) AND b(3)) XOR (a(6) AND b(4))
                XOR (a(5) AND b(5)) XOR (a(4) AND b(6)) XOR (a(3) AND b(7)) XOR (a(7) AND b(2))
                XOR (a(6) AND b(3)) XOR (a(5) AND b(4)) XOR (a(4) AND b(5)) XOR (a(3) AND b(6))
                XOR (a(2) AND b(7)) XOR (a(7) AND b(1)) XOR (a(6) AND b(2)) XOR (a(5) AND b(3))
                XOR (a(4) AND b(4)) XOR (a(3) AND b(5)) XOR (a(2) AND b(6)) XOR (a(1) AND b(7));
        -- bit 3
        output_i(3) <= (a(3) AND b(0)) XOR (a(2) AND b(1)) XOR (a(1) AND b(2)) XOR (a(0) AND b(3))
                XOR (a(7) AND b(5)) XOR (a(6) AND b(6)) XOR (a(5) AND b(7)) XOR (a(7) AND b(4))
                XOR (a(6) AND b(5)) XOR (a(5) AND b(6)) XOR (a(4) AND b(7)) XOR (a(7) AND b(2))
                XOR (a(6) AND b(3)) XOR (a(5) AND b(4)) XOR (a(4) AND b(5)) XOR (a(3) AND b(6))
                XOR (a(2) AND b(7)) XOR (a(7) AND b(1)) XOR (a(6) AND b(2)) XOR (a(5) AND b(3))
                XOR (a(4) AND b(4)) XOR (a(3) AND b(5)) XOR (a(2) AND b(6)) XOR (a(1) AND b(7));
        -- bit 2
        output_i(2) <= (a(2) AND b(0)) XOR (a(1) AND b(1)) XOR (a(0) AND b(2)) XOR (a(7) AND b(6))
                XOR (a(6) AND b(7)) XOR (a(7) AND b(5)) XOR (a(6) AND b(6)) XOR (a(5) AND b(7))
                XOR (a(7) AND b(3)) XOR (a(6) AND b(4)) XOR (a(5) AND b(5)) XOR (a(4) AND b(6))
                XOR (a(3) AND b(7)) XOR (a(7) AND b(1)) XOR (a(6) AND b(2)) XOR (a(5) AND b(3))
                XOR (a(4) AND b(4)) XOR (a(3) AND b(5)) XOR (a(2) AND b(6)) XOR (a(1) AND b(7));
        -- bit 1
        output_i(1) <= (a(1) AND b(0)) XOR (a(0) AND b(1)) XOR (a(7) AND b(7)) XOR (a(7) AND b(6))
                XOR (a(6) AND b(7)) XOR (a(7) AND b(2)) XOR (a(6) AND b(3)) XOR (a(5) AND b(4))
                XOR (a(4) AND b(5)) XOR (a(3) AND b(6)) XOR (a(2) AND b(7));
        -- bit 0
        output_i(0) <= (a(0) AND b(0)) XOR (a(7) AND b(7)) XOR (a(7) AND b(6)) XOR (a(6) AND b(7))
                XOR (a(7) AND b(5)) XOR (a(6) AND b(6)) XOR (a(5) AND b(7)) XOR (a(7) AND b(1))
                XOR (a(6) AND b(2)) XOR (a(5) AND b(3)) XOR (a(4) AND b(4)) XOR (a(3) AND b(5))
                XOR (a(2) AND b(6)) XOR (a(1) AND b(7));

        -- connect output to signal output_i
        c <= output_i;

end Behavioral;
```

A.3.11 Functional Block 8-Bit XOR - GF28Add.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity GF28Add is
   Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
          b : in  STD_LOGIC_VECTOR (7 downto 0);
```

```
        c : out  STD_LOGIC_VECTOR (7 downto 0));
end GF28Add;

architecture Behavioral of GF28Add is

            signal sig_output : std_logic_vector (7 downto 0) := (others=>'0');

begin

            -- GF Addition / XOR
            sig_output <= a XOR b;

            -- output
            c <= sig_output;

end Behavioral;
```

A.3.12 Functional Block SBN - SBN.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity SBN is
   Port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
        B : in  STD_LOGIC_VECTOR (8 downto 0);
        CIN : in  STD_LOGIC;
        N : out  STD_LOGIC;
        Z : out  STD_LOGIC;
                                    O : out STD_LOGIC_VECTOR (8 downto 0));
end SBN;

architecture Behavioral of SBN is

            signal sig_output : std_logic_vector (8 downto 0) := (others=>'0');

begin

            sig_output <= A + B + CIN;

            -- output zero, Z
            process(sig_output)
            begin
                        if sig_output = X"000" then
                                    Z <= '1';
                        else
                                    Z <= '0';
                        end if;
            end process;

            -- output negative, N
            process(sig_output)
            begin
                        if sig_output(8) = '1' then
                                    N <= '1';
                        else
            N <= '0';
        end if;
    end process;

    -- output O
    O <= sig_output;

end Behavioral;
```

A.3.13 9-Bit 2-To-1 Multiplexer - MUX29.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX29 is
   Port ( A : in  STD_LOGIC_VECTOR (8 downto 0);
          B : in  STD_LOGIC_VECTOR (8 downto 0);
          SEL : in  STD_LOGIC;
          C : out  STD_LOGIC_VECTOR (8 downto 0));
end MUX29;

architecture Behavioral of MUX29 is

begin

    -- MUX2-1 8 bit
    process(SEL,A,B)
    begin

        if SEL = '0' then
            C <= A;
        else
            C <= B;
        end if;

    end process;

end Behavioral;
```

A.3.14 10-Bit Register - REG10BIT.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```
entity REG10BIT is
   Port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
       B : out  STD_LOGIC_VECTOR (9 downto 0);
       CLK : in  STD_LOGIC;
       ENA : in  STD_LOGIC);
end REG10BIT;

architecture Behavioral of REG10BIT is

begin

          -- Register function
          process(CLK)

                    variable sig_data : std_logic_vector(9 downto 0) := (others=>'0');

          begin

                    if CLK'event and CLK = '1' then
                              if ENA = '1' then
                                        sig_data := A;
                              end if;
                    else
                              sig_data := sig_data;
                    end if;

                    B <= sig_data;

          end process;

end Behavioral;
```

A.3.15 LED 7-Segment Display - D4to7.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity D4to7 is
   Port ( Q : in  STD_LOGIC_VECTOR (3 downto 0);
       Seg : out STD_LOGIC_VECTOR (6 downto 0));
end D4to7;

architecture Behavioral of D4to7 is
          -- Segment encoding
          --                   a
          --          ---
          --        f|          |b
          --          --- <- g
          -- e|       |c
          --          ---
          --                   d
begin
          -- Conditional signal assignmens
          -- LED seg order = a,b,c,d,e,f,g = seg6, seg5, seg4, seg3, seg2, seg1, seg0
          Seg<=     "1111110" when q = "0000" else
```

```
                              "0110000" when q = "0001" else
                              "1101101" when q = "0010" else
                              "1111001" when q = "0011" else
                              "0110011" when q = "0100" else
                              "1011011" when q = "0101" else
                              "1011111" when q = "0110" else
                              "1110000" when q = "0111" else
                              "1111111" when q = "1000" else
                              "1111011" when q = "1001" else
                              "1110111" when q = "1010" else
                              "0011111" when q = "1011" else
                              "1001110" when q = "1100" else
                              "0111101" when q = "1101" else
                              "1001111" when q = "1110" else
                              "1000111" when q = "1111" else
                              "0000000";

end Behavioral;
```

A.4 RS MISC ARCHITECTURE IN VHDL

A.4.1 Top Level RS MISC Architecture - RSMISC.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity RSMISC is
   Port ( Seg0, Seg1 : out  STD_LOGIC_VECTOR (6 downto 0);
          RamAdd : out  STD_LOGIC_VECTOR (7 downto 0);
          CLK : in  STD_LOGIC;
          UP : in  STD_LOGIC;
          DOWN : in  STD_LOGIC;
                              EnaRead : in STD_LOGIC);
end RSMISC;

architecture Behavioral of RSMISC is

            -- Components
            -- Block RAM
            component memory
   port (
   addra: IN std_logic_VECTOR(9 downto 0);
   addrb: IN std_logic_VECTOR(9 downto 0);
   clka: IN std_logic;
   clkb: IN std_logic;
   dina: IN std_logic_VECTOR(10 downto 0);
   douta: OUT std_logic_VECTOR(10 downto 0);
   doutb: OUT std_logic_VECTOR(10 downto 0);
   ena: IN std_logic;
   enb: IN std_logic;
   wea: IN std_logic);
```

```
end component;

-- Controls
component controls
        port(   CLK : in std_logic;
                N : in std_logic;
                ALU_A : out std_logic;
                ALU_B0 : out std_logic;
                ALU_B1 : out std_logic;
                CIN : out std_logic;
                MAR_SEL : out std_logic;
                PC_WRITE : out std_logic;
                R_WRITE : out std_logic;
                Z_WRITE : out std_logic;
                N_WRITE : out std_logic;
                MAR_WRITE : out std_logic;
                MDR_WRITE : out std_logic;
                MEM_READ : out std_logic;
                MEM_WRITE : out std_logic;
                OP_OUT_SEL : out std_logic;
                OP0_WRITE : out std_logic;
                OP1_WRITE : out std_logic;
                OP_SEL : out std_logic);
end component;

-- MUX
-- MUX2-1 1-Bit
component MUX21 is
        port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
                B : in  STD_LOGIC_VECTOR (7 downto 0);
                SEL : in STD_LOGIC;
                C : out  STD_LOGIC_VECTOR (7 downto 0));
end component;
-- MUX2-1 2-Bit
component MUX22 is
        port ( A : in  STD_LOGIC_VECTOR (1 downto 0);
                B : in  STD_LOGIC_VECTOR (1 downto 0);
                SEL : in STD_LOGIC;
                C : out  STD_LOGIC_VECTOR (1 downto 0));
end component;
-- MUX4-1 10-Bit
component MUX410 is
        port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
                B : in  STD_LOGIC_VECTOR (9 downto 0);
                C : in  STD_LOGIC_VECTOR (9 downto 0);
                D : out  STD_LOGIC_VECTOR (9 downto 0);
                SEL : in  STD_LOGIC_VECTOR (1 downto 0));
end component;
-- MUX1-4 10-Bit
component MUX104 is
        port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
                SEL : in  STD_LOGIC_VECTOR (1 downto 0);
                B : out  STD_LOGIC_VECTOR (9 downto 0);
                C : out  STD_LOGIC_VECTOR (9 downto 0);
                D : out  STD_LOGIC_VECTOR (9 downto 0));
end component;
-- MUX1-2 1-Bit
component MUX11 is
        port ( A : in  STD_LOGIC;
                SEL : in  STD_LOGIC;
                B : out  STD_LOGIC;
                C : out  STD_LOGIC);
end component;
-- MUX2-1 10-Bit
component MUX210 is
        port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
                B : in  STD_LOGIC_VECTOR (9 downto 0);
```

```
                                SEL : in  STD_LOGIC;
                                C : out  STD_LOGIC_VECTOR (9 downto 0));
end component;

-- Registers
-- 10-Bit Register
component REG is
          port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
                                B : out  STD_LOGIC_VECTOR (9 downto 0);
                                CLK : in  STD_LOGIC;
                                ENA : in  STD_LOGIC);
end component;
-- 1-Bit Register
component REG1BIT is
          Port ( A : in  STD_LOGIC;
                                B : out  STD_LOGIC;
                                CLK : in  STD_LOGIC;
                                ENA : in  STD_LOGIC);
end component;
-- 11-Bit Register
component REG11BIT is
          Port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
                                B : out  STD_LOGIC_VECTOR (10 downto 0);
                                CLK : in  STD_LOGIC;
                                ENA : in  STD_LOGIC);
end component;
-- PC Register
component REGPC is
          port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
                                B : out  STD_LOGIC_VECTOR (9 downto 0);
                                CLK : in  STD_LOGIC;
                                ENA : in  STD_LOGIC);
end component;

-- SBN
component SBN is
          Port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
                                B : in  STD_LOGIC_VECTOR (9 downto 0);
                                CIN : in  STD_LOGIC;
                                N : out  STD_LOGIC;
                                Z : out  STD_LOGIC;
                                O : out STD_LOGIC_VECTOR (9 downto 0));
end component;

-- 8-Bit XOR
component GF28Add is
          Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
                                b : in  STD_LOGIC_VECTOR (7 downto 0);
                                c : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

-- GF28
component GF28 is
          Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
                                b : in  STD_LOGIC_VECTOR (7 downto 0);
                                c : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

-- D4to7 Conversion
component D4to7 is
          Port ( Q : in  STD_LOGIC_VECTOR (3 downto 0);
                                Seg : out STD_LOGIC_VECTOR (6 downto 0));
end component;

-- Signals
signal iCount1 : std_logic := '0';
signal iClock2 : std_logic;
signal iCount23 : std_logic_vector(22 downto 0);
signal iClock23 : std_logic;
signal iRamAdd : std_logic_vector(9 downto 0);
signal iRamRead : std_logic_vector(10 downto 0);

-- Program Counter
signal iPC : std_logic_vector(9 downto 0) := (others=>'0');
signal PC : std_logic_vector(9 downto 0) := (others=>'0');
signal PC_Write : std_logic;
-- R Register
signal iR : std_logic_vector(9 downto 0);
signal R : std_logic_vector(9 downto 0);
signal R_Write : std_logic;

-- OPCODE
-- MUX
signal OP_SEL : std_logic;
-- OP0 Register
signal iOP0 : std_logic;
signal OP0_Write : std_logic;
-- OP1 Register
signal iOP1 : std_logic;
signal OP1_Write : std_logic;
-- OPCODE output
signal OPCODE : std_logic_vector(1 downto 0);
signal OP_OUT_SEL : std_logic;
signal oOPCODE : std_logic_vector(1 downto 0);

-- MEMORY
-- MDR register
signal iMDR : std_logic_vector(10 downto 0);
signal MDR : std_logic_vector(10 downto 0);
signal MDR_Write : std_logic;
-- MAR
signal iMAR : std_logic_vector(9 downto 0);
signal MAR : std_logic_vector(9 downto 0);
signal MAR_Write : std_logic;
signal MAR_SEL : std_logic;
-- MEM Output
signal oMemory : std_logic_vector(10 downto 0);
-- MEM Controls
signal MEM_READ : std_logic;
signal MEM_WRITE : std_logic;
signal MEM_ENA : std_logic;
signal MEM_WEA : std_logic;

-- SBN/ALU Block
-- N register
signal i_N : std_logic;
signal N : std_logic;
signal N_Write : std_logic;
-- Z register
signal iZ : std_logic;
signal Z : std_logic;
signal Z_Write : std_logic;
-- ALU MUX
-- ALU MUX A
signal iMUXALUA0 : std_logic_vector(9 downto 0) := (others=>'0');
signal iMUXALUA1 : std_logic_vector(9 downto 0) := (others=>'0');
signal MUXALUA : std_logic_vector(9 downto 0) := (others=>'0');
signal MUXALUASEL : std_logic;
signal ALU_A : std_logic;
-- ALU MUX B
signal iMUXALUB0 : std_logic_vector(9 downto 0) := (others=>'0');
signal iMUXALUB1 : std_logic_vector(9 downto 0) := (others=>'0');
signal iMUXALUB2 : std_logic_vector(9 downto 0) := (others=>'0');
signal MUXALUB : std_logic_vector(9 downto 0) := (others=>'0');
```

```
    signal MUXALUBSEL : std_logic_vector(1 downto 0);
    signal ALU_B : std_logic_vector(1 downto 0);
    -- INV
    signal iINV : std_logic_vector(9 downto 0) := (others=>'0');
    signal oINV : std_logic_vector(9 downto 0) := (others=>'0');
    -- CIN
    signal CIN : std_logic;
    -- Output
    signal ADDER : std_logic_vector(9 downto 0) := (others=>'0');

    -- GF Block
    signal iGFA : std_logic_vector(9 downto 0) := (others=>'0');
    signal iGFB : std_logic_vector(9 downto 0) := (others=>'0');
    signal iGF : std_logic_vector(9 downto 0) := (others=>'0');

    -- XOR Block
    signal iXORA : std_logic_vector(9 downto 0) := (others=>'0');
    signal iXORB : std_logic_vector(9 downto 0) := (others=>'0');
    signal iXOR : std_logic_vector(9 downto 0) := (others=>'0');

    -- Instruction MUX
    -- MUX A
    signal iMUXA : std_logic_vector(9 downto 0) := (others=>'0');
    signal MUXA0 : std_logic_vector(9 downto 0) := (others=>'0');
    signal MUXA1 : std_logic_vector(9 downto 0) := (others=>'0');
    signal MUXA2 : std_logic_vector(9 downto 0) := (others=>'0');
    signal MUXASEL : std_logic_vector(1 downto 0);
    -- MUX B
    signal iMUXB : std_logic_vector(9 downto 0) := (others=>'0');
    signal MUXB0 : std_logic_vector(9 downto 0) := (others=>'0');
    signal MUXB1 : std_logic_vector(9 downto 0) := (others=>'0');
    signal MUXB2 : std_logic_vector(9 downto 0) := (others=>'0');
    signal MUXBSEL : std_logic_vector(1 downto 0);
    -- MUX Out
    signal MUXO : std_logic_vector(9 downto 0) := (others=>'0');
    signal iMUXO0 : std_logic_vector(9 downto 0) := (others=>'0');
    signal iMUXO1 : std_logic_vector(9 downto 0) := (others=>'0');
    signal iMUXO2 : std_logic_vector(9 downto 0) := (others=>'0');
    signal MUXOSEL : std_logic_vector(1 downto 0);

begin

    -- Controls Block
    Ctrl : controls
            port map(
                    CLK => iClock2,
                    N => N,
                    ALU_A => ALU_A,
                    ALU_B0 => ALU_B(0),
                    ALU_B1 => ALU_B(1),
                    CIN => CIN,
                    MAR_SEL => MAR_SEL,
                    PC_WRITE => PC_Write,
                    R_WRITE => R_Write,
                    Z_WRITE => Z_Write,
                    N_WRITE => N_Write,
                    MAR_WRITE => MAR_Write,
                    MDR_WRITE => MDR_Write,
                    MEM_READ => MEM_READ,
                    MEM_WRITE => MEM_WRITE,
                    OP_OUT_SEL => OP_OUT_SEL,
                    OP0_WRITE => OP0_Write,
                    OP1_WRITE => OP1_Write,
                    OP_SEL => OP_SEL);

    -- Registers
    -- PC
```

```
PC_Reg : REGPC
        port map(
                        A => iPC,
                        B => PC,
                        CLK => iClock2,
                        ENA => PC_Write);
iPC <= ADDER;

-- R
R_Reg : REG
        port map(
                        A => iR,
                        B => R,
                        CLK => iClock2,
                        ENA => R_Write);
iR <= oMemory(9 downto 0);

-- Z
Z_Reg : REG1BIT
        port map(
                        A => iZ,
                        B => Z,
                        CLK => iClock2,
                        ENA => Z_Write);

-- N
N_Reg : REG1BIT
        port map(
                        A => i_N,
                        B => N,
                        CLK => iClock2,
                        ENA => N_Write);

-- OPCODE
-- OPCODE1
OPCODE1_Reg : REG1BIT
        port map(
                        A => iOP1,
                        B => OPCODE(1),
                        CLK => iClock2,
                        ENA => OP1_Write);

-- OPCODE0
OPCODE0_Reg : REG1BIT
        port map(
                        A => iOP0,
                        B => OPCODE(0),
                        CLK => iClock2,
                        ENA => OP0_Write);

-- oMUXOP
oMUXOP : MUX22
        port map (
                        A => OPCODE,
                        B => "10",
                        SEL => OP_OUT_SEL,
                        C => oOPCODE);

-- MUXOP
MUXOP : MUX11
        port map (
                        A => oMemory(10),
                        SEL => OP_SEL,
                        B => iOP1,
                        C => iOP0);
```

```
-- MUXA
MUXA : MUX104
             port map (
                            A => iMUXA,
                            SEL => MUXASEL,
                            B => MUXA0,
                            C => MUXA1,
                            D => MUXA2);
MUXASEL <= oOPCODE;
iMUXA <= oMemory(9 downto 0);
iGFA <= MUXA0;
iXORA <= MUXA1;
IMUXALUA0 <= MUXA2;

-- MUXB
MUXB : MUX104
             port map (
                            A => iMUXB,
                            SEL => MUXBSEL,
                            B => MUXB0,
                            C => MUXB1,
                            D => MUXB2);
MUXBSEL <= oOPCODE;
iMUXB <= R;
iGFB <= MUXB0;
iXORB <= MUXB1;
iINV <= MUXB2;

-- MUX OUT
MUXOUT : MUX410
              port map (
                                        A => iMUXO0,
                                        B => iMUXO1,
                                        C => iMUXO2,
                                        D => MUXO,
                                        SEL => MUXOSEL);
MUXOSEL <= oOPCODE;
iMUXO0 <= iGF;
iMUXO1 <= iXOR;
iMUXO2 <= ADDER;
iMDR <= oMemory(10) & MUXO;


-- GF
GF28MULT : GF28
              port map (
                                        a => iGFA(7 downto 0),
                                        b => iGFB(7 downto 0),
                                        c => iGF(7 downto 0));
iGF(9 downto 8) <= "00";

-- XOR
GF28XOR : GF28Add
                          port map (
                                                    a =>iXORA(7 downto 0),
                                                    b =>iXORB(7 downto 0),
                                                    c =>iXOR(7 downto 0));
iXOR(9 downto 8) <= "00";


-- SBN
SBN_BLOCK : SBN
              port map (
                                        A => MUXALUA,
                                        B => MUXALUB,
                                        CIN => CIN,
                                        N => i_N,
                                        Z => iZ,
                                        O => ADDER);

        -- ALU_A MUX
        MUX_ALU_A : MUX210
                            port map (
                                        A => iMUXALUA0,
                                        B => iMUXALUA1,
                                        SEL => MUXALUASEL,
                                        C => MUXALUA);
        MUXALUASEL <= ALU_A;
        iMUXALUA1 <= PC;

        -- ALU_B MUX
        MUX_ALU_B : MUX410
                port map (
                                        A => iMUXALUB0,
                                        B => iMUXALUB1,
                                        C => iMUXALUB2,
                                        D => MUXALUB,
                                        SEL => MUXALUBSEL);
        MUXALUBSEL <= ALU_B;
        iMUXALUB0 <= PC;
        iMUXALUB1 <= oINV;
        iMUXALUB2 <= "0000000000";

        -- INV
        oINV <= NOT iINV;

        -- MDR
        MDR_Reg : REG11BIT
                port map(
                                        A => iMDR,
                                        B => MDR,
                                        CLK => iClock2,
                                        ENA => MDR_Write);

        -- MUX MAR
        MUXMAR : MUX210
                port map(
                                        A => ADDER,
                                        B => oMemory(9 downto 0),
                                        SEL => MAR_SEL,
                                        C => iMAR);

        -- MAR
        MAR_Reg : REG
                port map(
                                        A => iMAR,
                                        B => MAR,
                                        CLK => iClock2,
                                        ENA => MAR_Write);
        --      iMAR <= ADDER;

        -- Block RAM
        Block_RAM : memory
    port map (
      addra => MAR,
      addrb => iRamAdd,
      clka => CLK,
      clkb => CLK,
      dina => MDR,
      douta => oMemory,
      doutb => iRamRead,
      ena => MEM_ENA,
      enb => EnaRead,
      wea => MEM_WEA);
```

```
-- Memory Control
process(MEM_READ,MEM_WRITE,CLK)
begin
        if MEM_WRITE = '1' then
                MEM_ENA <= '1';
                MEM_WEA <= MEM_WRITE;
        elsif MEM_READ = '1' then
                MEM_ENA <= MEM_READ;
                MEM_WEA <= '0';
        else
                MEM_ENA <= MEM_READ;
                MEM_WEA <= '0';
        end if;
end process;

-- RAM Address
process(iClock23,UP,DOWN)
begin

        if iClock23'event and iClock23 = '1' then
                if UP = '1' then
                        iRamAdd <= iRamAdd + '1';
                elsif DOWN = '1' then
                        iRamAdd <= iRamAdd - '1';
                else
                        iRamAdd <= iRamAdd;
                end if;
        end if;

end process;

RamAdd <= iRamAdd(7 downto 0);
-- RamRead <= iRamRead(7 downto 0);

-- Seg0
Seg70 : D4to7
         port map(
                                 Q => iRamRead(3 downto 0),
                                 Seg => Seg0);

-- Seg1
Seg71 : D4to7
         port map(
                                 Q => iRamRead(7 downto 4),
                                 Seg => Seg1);

-- Clock24
process(CLK)
begin

        if CLK'event and CLK = '1' then
                iCount23 <= iCount23 + '1';
        end if;

end process;
-- Actual Implementation
        iClock23 <= iCount23(22);
-- Simulation
--      PiClock24 <= CLK;

-- Clock2
process(CLK)
begin

        if CLK'event and CLK = '1' then
                if PC /= "1111111111" then
                        iCount1 <= NOT iCount1;
```

```
				end if;
			end if;

		end process;
		-- Actual Implementation
			iClock2 <= iCount1;
		-- Simulation
		-- iClock2 <= CLK;


end Behavioral;
```

A.4.2 Control Signals Combinational Circuit - controls.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity controls is
	port(	CLK : in std_logic;
			N : in std_logic;
			ALU_A : out std_logic;
			ALU_B0 : out std_logic;
			ALU_B1 : out std_logic;
			CIN : out std_logic;
			MAR_SEL : out std_logic;
			PC_WRITE : out std_logic;
			R_WRITE : out std_logic;
			Z_WRITE : out std_logic;
			N_WRITE : out std_logic;
			MAR_WRITE : out std_logic;
			MDR_WRITE : out std_logic;
			MEM_READ : out std_logic;
			MEM_WRITE : out std_logic;
			OP_OUT_SEL : out std_logic;
			OP0_WRITE : out std_logic;
			OP1_WRITE : out std_logic;
			OP_SEL : out std_logic);
end controls;

architecture Behavioral of controls is

	signal iCount4 : std_logic_vector(3 downto 0) := X"8";

begin

	-- 4 Bit Counter
	process(CLK)
	begin

		if CLK'event and CLK = '1' then
			if iCount4 = "1000" then
				iCount4 <= (others=>'0');
			else
				iCount4 <= iCount4 + 1;
			end if;
```

```
        end if;

    end process;


    -- Output control signals
    ALU_A <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(0))) )
        OR ( (NOT(iCount4(2))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) );
    ALU_B0 <= (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) AND (iCount4(0));
    ALU_B1 <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(0))) )
        OR ( (NOT(iCount4(2))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) );
    CIN <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (iCount4(0)) )
        OR ( (iCount4(3)) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
            AND (NOT(iCount4(0))) );
    MAR_SEL <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (iCount4(0)) );
    PC_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (iCount4(1)) AND (iCount4(0)) AND N )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
            AND (iCount4(0)) )
        OR ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
            AND (NOT(iCount4(0))) )
        OR ( (iCount4(3)) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
            AND (NOT(iCount4(0))) );
    R_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (iCount4(1))
            AND (NOT(iCount4(0))) );
    Z_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
            AND (NOT(iCount4(0))) );
    N_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) AND (iCount4(0)) );
    MAR_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (iCount4(0)) )
        OR ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(0))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) );
    MDR_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
            AND (iCount4(0)) );
    MEM_READ <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1))) )
        OR ( (NOT(iCount4(3))) AND (iCount4(1)) AND (iCount4(0)) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (iCount4(0)) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (iCount4(1))
            AND (NOT(iCount4(0))) );
    MEM_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (iCount4(1))
            AND (NOT(iCount4(0))) );
    OP_OUT_SEL <= ( (NOT(iCount4(3))) AND (NOT(iCount4(1))) AND (NOT(iCount4(0))) )
        OR ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) );
    OP0_WRITE <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
            AND (NOT(iCount4(0))) );
    OP1_WRITE <= ( (NOT(iCount4(3))) AND (NOT(iCount4(2))) AND (NOT(iCount4(1)))
            AND (iCount4(0)) );
    OP_SEL <= ( (NOT(iCount4(3))) AND (iCount4(2)) AND (NOT(iCount4(1)))
            AND (NOT(iCount4(0))) );

end Behavioral;
```

A.4.3 9-Bit Programme Counter Register - REGPC.vhd

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

---- Uncomment the following library declaration if instantiating
```

```
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity REGPC is
   Port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
          B : out  STD_LOGIC_VECTOR (9 downto 0);
          CLK : in  STD_LOGIC;
          ENA : in  STD_LOGIC);
end REGPC;

architecture Behavioral of REGPC is

begin

            -- Register function
            process(CLK)

                        -- Start of programme
                        -- program start at 0x200, put 0x1FF
                        variable sig_data : std_logic_vector (9 downto 0) := "0111111111";
                        -- Test Last Line
                        -- variable sig_data : std_logic_vector (9 downto 0) := "111011001";

            begin

                        if CLK'event and CLK = '1' then
                                    if ENA = '1' then
                                                sig_data := A;
                                    end if;
                        else
                                    sig_data := sig_data;
                        end if;

                        B <= sig_data;

            end process;

end Behavioral;
```

A.4.4 9-Bit Register - REG.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity REG is
   Port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
          B : out  STD_LOGIC_VECTOR (9 downto 0);
          CLK : in  STD_LOGIC;
          ENA : in  STD_LOGIC);
end REG;

architecture Behavioral of REG is

begin
```

```
        -- Register function
        process(CLK)

                variable sig_data : std_logic_vector (9 downto 0) := (others=>'0');

        begin

                if CLK'event and CLK = '1' then
                        if ENA = '1' then
                                sig_data := A;
                        end if;
                else
                        sig_data := sig_data;
                end if;

                B <= sig_data;

        end process;

end Behavioral;
```

A.4.5 1-Bit Register - REG1BIT.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity REG1BIT is
    Port ( A : in  STD_LOGIC;
           B : out  STD_LOGIC;
           CLK : in  STD_LOGIC;
           ENA : in  STD_LOGIC);
end REG1BIT;

architecture Behavioral of REG1BIT is

begin

        -- Register function
        process(CLK)

                variable sig_data : std_logic := '0';

        begin

                if CLK'event and CLK = '1' then
                        if ENA = '1' then
                                sig_data := A;
                        end if;
                else
                        sig_data := sig_data;
                end if;

                B <= sig_data;
```

```
		end process;

end Behavioral;
```

A.4.6 2-Bit 2-To-1 Multiplexer - MUX22.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX22 is
    Port ( A : in  STD_LOGIC_VECTOR (1 downto 0);
           B : in  STD_LOGIC_VECTOR (1 downto 0);
                                  SEL : in STD_LOGIC;
           C : out  STD_LOGIC_VECTOR (1 downto 0));
end MUX22;

architecture Behavioral of MUX22 is

begin

		-- MUX2-1 2 bit
		process(SEL,A,B)
		begin

				if SEL = '0' then
						C <= A;
				else
						C <= B;
				end if;

		end process;

end Behavioral;
```

A.4.7 1-Bit 1-To-2 Multiplexer - MUX11.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX11 is
    Port ( A : in  STD_LOGIC;
           SEL : in  STD_LOGIC;
           B : out  STD_LOGIC;
           C : out  STD_LOGIC);
```

```
end MUX11;

architecture Behavioral of MUX11 is

begin

        -- MUX1-2 1-Bit
        process(SEL,A)
        begin

                if SEL = '0' then
                        B <= A;
                        C <= '0';
                else
                        B <= '0';
                        C <= A;
                end if;

        end process;

end Behavioral;
```

A.4.8 10-Bit 1-To-3 Multiplexer - MUX104.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX11 is
  Port ( A : in  STD_LOGIC;
         SEL : in  STD_LOGIC;
         B : out  STD_LOGIC;
         C : out  STD_LOGIC);
end MUX11;

architecture Behavioral of MUX11 is

begin

        -- MUX1-2 1-Bit
        process(SEL,A)
        begin

                if SEL = '0' then
                        B <= A;
                        C <= '0';
                else
                        B <= '0';
                        C <= A;
                end if;

        end process;

end Behavioral;
```

A.4.9 10-Bit 3-To-1 Multiplexer - MUX410.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX410 is
   Port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
          B : in  STD_LOGIC_VECTOR (9 downto 0);
                                    C : in  STD_LOGIC_VECTOR (9 downto 0);
          D : out  STD_LOGIC_VECTOR (9 downto 0);
          SEL : in  STD_LOGIC_VECTOR (1 downto 0));
end MUX410;

architecture Behavioral of MUX410 is

begin

            -- MUX4-1 10-bit
            process(SEL,A,B,C)
            begin

                        if SEL = "00" then
                                    D <= A;
                        elsif SEL = "01" then
                                    D <= B;
                        elsif SEL = "10" then
                                    D <= C;
                        else
                                    D <= "0000000000";
                        end if;

            end process;

end Behavioral;
```

A.4.10 Functional Block GF(28) Multiplier - GF28.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity GF28 is
   Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
          b : in  STD_LOGIC_VECTOR (7 downto 0);
          c : out  STD_LOGIC_VECTOR (7 downto 0));
end GF28;

architecture Behavioral of GF28 is
```

```
        signal output_i : std_logic_vector (7 downto 0) := (others=>'0');

begin

        -- GF(2^8) Multiplier
        -- bit 7
        output_i(7) <= (a(7) AND b(0)) XOR (a(6) AND b(1)) XOR (a(5) AND b(2)) XOR (a(4) AND b(3))
                XOR (a(3) AND b(4)) XOR (a(2) AND b(5)) XOR (a(1) AND b(6)) XOR (a(0) AND b(7))
                XOR (a(7) AND b(6)) XOR (a(6) AND b(7)) XOR (a(7) AND b(5)) XOR (a(6) AND b(6))
                XOR (a(5) AND b(7)) XOR (a(7) AND b(4)) XOR (a(6) AND b(5)) XOR (a(5) AND b(6))
                XOR (a(4) AND b(7));
        -- bit 6
        output_i(6) <= (a(6) AND b(0)) XOR (a(5) AND b(1)) XOR (a(4) AND b(2)) XOR (a(3) AND b(3))
                XOR (a(2) AND b(4)) XOR (a(1) AND b(5)) XOR (a(0) AND b(6)) XOR (a(7) AND b(5))
                XOR (a(6) AND b(6)) XOR (a(5) AND b(7)) XOR (a(7) AND b(4)) XOR (a(6) AND b(5))
                XOR (a(5) AND b(6)) XOR (a(4) AND b(7)) XOR (a(7) AND b(3)) XOR (a(6) AND b(4))
                XOR (a(5) AND b(5)) XOR (a(4) AND b(6)) XOR (a(3) AND b(7));
        -- bit 5
        output_i(5) <= (a(5) AND b(0)) XOR (a(4) AND b(1)) XOR (a(3) AND b(2)) XOR (a(2) AND b(3))
                XOR (a(1) AND b(4)) XOR (a(0) AND b(5)) XOR (a(7) AND b(4)) XOR (a(6) AND b(5))
                XOR (a(5) AND b(6)) XOR (a(4) AND b(7)) XOR (a(7) AND b(3)) XOR (a(6) AND b(4))
                XOR (a(5) AND b(5)) XOR (a(4) AND b(6)) XOR (a(3) AND b(7)) XOR (a(7) AND b(2))
                XOR (a(6) AND b(3)) XOR (a(5) AND b(4)) XOR (a(4) AND b(5)) XOR (a(3) AND b(6))
                XOR (a(2) AND b(7));
        -- bit 4
        output_i(4) <= (a(4) AND b(0)) XOR (a(3) AND b(1)) XOR (a(2) AND b(2)) XOR (a(1) AND b(3))
                XOR (a(0) AND b(4)) XOR (a(7) AND b(7)) XOR (a(7) AND b(3)) XOR (a(6) AND b(4))
                XOR (a(5) AND b(5)) XOR (a(4) AND b(6)) XOR (a(3) AND b(7)) XOR (a(7) AND b(2))
                XOR (a(6) AND b(3)) XOR (a(5) AND b(4)) XOR (a(4) AND b(5)) XOR (a(3) AND b(6))
                XOR (a(2) AND b(7)) XOR (a(7) AND b(1)) XOR (a(6) AND b(2)) XOR (a(5) AND b(3))
                XOR (a(4) AND b(4)) XOR (a(3) AND b(5)) XOR (a(2) AND b(6)) XOR (a(1) AND b(7));
        -- bit 3
        output_i(3) <= (a(3) AND b(0)) XOR (a(2) AND b(1)) XOR (a(1) AND b(2)) XOR (a(0) AND b(3))
                XOR (a(7) AND b(5)) XOR (a(6) AND b(6)) XOR (a(5) AND b(7)) XOR (a(7) AND b(4))
                XOR (a(6) AND b(5)) XOR (a(5) AND b(6)) XOR (a(4) AND b(7)) XOR (a(7) AND b(2))
                XOR (a(6) AND b(3)) XOR (a(5) AND b(4)) XOR (a(4) AND b(5)) XOR (a(3) AND b(6))
                XOR (a(2) AND b(7)) XOR (a(7) AND b(1)) XOR (a(6) AND b(2)) XOR (a(5) AND b(3))
                XOR (a(4) AND b(4)) XOR (a(3) AND b(5)) XOR (a(2) AND b(6)) XOR (a(1) AND b(7));
        -- bit 2
        output_i(2) <= (a(2) AND b(0)) XOR (a(1) AND b(1)) XOR (a(0) AND b(2)) XOR (a(7) AND b(6))
                XOR (a(6) AND b(7)) XOR (a(7) AND b(5)) XOR (a(6) AND b(6)) XOR (a(5) AND b(7))
                XOR (a(7) AND b(3)) XOR (a(6) AND b(4)) XOR (a(5) AND b(5)) XOR (a(4) AND b(6))
                XOR (a(3) AND b(7)) XOR (a(7) AND b(1)) XOR (a(6) AND b(2)) XOR (a(5) AND b(3))
                XOR (a(4) AND b(4)) XOR (a(3) AND b(5)) XOR (a(2) AND b(6)) XOR (a(1) AND b(7));
        -- bit 1
        output_i(1) <= (a(1) AND b(0)) XOR (a(0) AND b(1)) XOR (a(7) AND b(7)) XOR (a(7) AND b(6))
                XOR (a(6) AND b(7)) XOR (a(7) AND b(2)) XOR (a(6) AND b(3)) XOR (a(5) AND b(4))
                XOR (a(4) AND b(5)) XOR (a(3) AND b(6)) XOR (a(2) AND b(7));
        -- bit 0
        output_i(0) <= (a(0) AND b(0)) XOR (a(7) AND b(7)) XOR (a(7) AND b(6)) XOR (a(6) AND b(7))
                XOR (a(7) AND b(5)) XOR (a(6) AND b(6)) XOR (a(5) AND b(7)) XOR (a(7) AND b(1))
                XOR (a(6) AND b(2)) XOR (a(5) AND b(3)) XOR (a(4) AND b(4)) XOR (a(3) AND b(5))
                XOR (a(2) AND b(6)) XOR (a(1) AND b(7));

        -- connect output to signal output_i
        c <= output_i;

end Behavioral;
```

A.4.11 Functional Block 8-Bit XOR - GF28Add.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity GF28Add is
   Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
          b : in  STD_LOGIC_VECTOR (7 downto 0);
          c : out  STD_LOGIC_VECTOR (7 downto 0));
end GF28Add;

architecture Behavioral of GF28Add is

            signal sig_output : std_logic_vector (7 downto 0) := (others=>'0');

begin

            -- GF Addition / XOR
            sig_output <= a XOR b;

            -- output
            c <= sig_output;

end Behavioral;
```

A.4.12 Functional Block SBN - SBN.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity SBN is
   Port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
          B : in  STD_LOGIC_VECTOR (9 downto 0);
          CIN : in  STD_LOGIC;
          N : out  STD_LOGIC;
          Z : out  STD_LOGIC;
                                    O : out STD_LOGIC_VECTOR (9 downto 0));
end SBN;

architecture Behavioral of SBN is

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

begin

            sig_output <= A + B + CIN;

            -- output zero, Z
```

```
        process(sig_output)
        begin
                if sig_output = X"000" then
                        Z <= '1';
                else
                        Z <= '0';
                end if;
        end process;

        -- output negative, N
        process(sig_output)
        begin
                if sig_output(8) = '1' then
                        N <= '1';
                else
                        N <= '0';
                end if;
        end process;

        -- output O
        O <= sig_output;

end Behavioral;
```

A.4.13 10-Bit 2-To-1 Multiplexer - MUX210.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity MUX210 is
    Port ( A : in  STD_LOGIC_VECTOR (9 downto 0);
           B : in  STD_LOGIC_VECTOR (9 downto 0);
           SEL : in  STD_LOGIC;
           C : out  STD_LOGIC_VECTOR (9 downto 0));
end MUX210;

architecture Behavioral of MUX210 is

begin

        -- MUX2-1 10 bit
        process(SEL,A,B)
        begin

                if SEL = '0' then
                        C <= A;
                else
                        C <= B;
                end if;

        end process;

end Behavioral;
```

A.4.14 11-Bit Register - REG11BIT.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity REG11BIT is
   Port ( A : in  STD_LOGIC_VECTOR (10 downto 0);
          B : out  STD_LOGIC_VECTOR (10 downto 0);
          CLK : in  STD_LOGIC;
          ENA : in  STD_LOGIC);
end REG11BIT;

architecture Behavioral of REG11BIT is

begin

        -- Register function
        process(CLK)

                variable sig_data : std_logic_vector(10 downto 0) := (others=>'0');

        begin

                if CLK'event and CLK = '1' then
                        if ENA = '1' then
                                sig_data := A;
                        end if;
                else
                        sig_data := sig_data;
                end if;

                B <= sig_data;

        end process;

end Behavioral;
```

A.4.15 7-Segment LED Display - D4to7.vhd

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity D4to7 is
   Port ( Q : in  STD_LOGIC_VECTOR (3 downto 0);
          Seg : out STD_LOGIC_VECTOR (6 downto 0));
end D4to7;

architecture Behavioral of D4to7 is
```

```
        -- Segment encoding
        --                a
        --         ---
        --       f|        |b
        --         --- <- g
        -- e|      |c
        --         ---
        --                d
begin
        -- Conditional signal assignmens
        -- LED seg order = a,b,c,d,e,f,g = seg6, seg5, seg4, seg3, seg2, seg1, seg0
        Seg<=   "1111110" when q = "0000" else
                "0110000" when q = "0001" else
                "1101101" when q = "0010" else
                "1111001" when q = "0011" else
                "0110011" when q = "0100" else
                "1011011" when q = "0101" else
                "1011111" when q = "0110" else
                "1110000" when q = "0111" else
                "1111111" when q = "1000" else
                "1111011" when q = "1001" else
                "1110111" when q = "1010" else
                "0011111" when q = "1011" else
                "1001110" when q = "1100" else
                "0111101" when q = "1101" else
                "1001111" when q = "1110" else
                "1000111" when q = "1111" else
                "0000000";

end Behavioral;
```