Implementation of the CUSUM Algorithm on FPGA for Transient Signal Detection

Li Kang

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING (ME) IN THE DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

Auckland University of Technology

JANUARY 2012

SUPERVISOR: Dr. Hamid GholamHosseini

Acknowledgement

First and foremost, I would like to thank my supervisor, Dr. Hamid GholamHosseini, for his constant guidance, valued advice and continuous support during the time of my master study. He has been always available and highly supportive throughout this study.

I also want to thank my family for the endless love, invaluable support and encouragement through the years.

Many thanks go out to J.Huang for his helpful advice, understanding and encouragement when he was working as my technical support during the first part of this study.

I am grateful for the research environment and the substantial resources provided by the School of Engineering of AUT University that enabled me to gain extensive practical experience through the experimental work performed.

Abstract

Radio transient signals are non-periodic and discrete obtained from high energy physical processes in space. One of the most challenging issues in transient signal detection is the speed and accuracy with which a signal can be detected. The target for this design is selecting appropriate detection algorithm and minimize time consumption. Gene. S [1] and P.A. Fridman [2] proposed the use of Cumulative Sum (CUSUM) algorithm for transient signal detection capable of meeting the necessary requirements. However, as ordinary software-based programs are unable to handle large scale sampling of signals, the current research focuses on implementing the CUSUM algorithm on Field Programmable Gate Array (FPGA) which is a specific integrated circuit within the field of semi-customised circuits that can greatly enhance the speed of detection and analysis.

In this research, standard deviation was used in CUSUM algorithm as the threshold to determine whether the detection signal is out of range (abnormal signal). The author chose a top-down design method to split the CUSUM algorithm into several sub-modules including parallel module, standard deviation module and delay module. These modules were implemented one by one using a hardware description language (Verilog) and link together to achieve to the objective of the project. During the design process the author analyzed the problems found within the design and selected appropriate solutions. The CUSUM core uses pipeline processing architecture, with an incubation period of only 128 ns (64 clock cycles). After being compiled and verified, it was shown the design was successful. A detection speed of 64 ns per sampling group was achieved via implementation on an Altera Cyclone IV device with a clock speed of 50 MHz. Furthermore, the author then analyzed the power consumption and discusses the power consumption in the context of multi-core application. The analyzed result shows the power consumption of a single CUSUM core to be only 136.75mW.

Finally, the findings of the design process are summarised with the author presenting suggestions for further improvement. The improvement proposed is based on two key aspects: coding design and multiprocessor operation.

Table of Contents

List of Figures

List of Tables

List of Equations

Statement of Originality

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning, except where due acknowledgment is made in the acknowledgments.

_____ (SIGNED)

_____ (DATE)

Acronyms and Abbreviations

ASIC: Application Specific Integrated Circuit

ALU: Arithmetic Logic Unit

ADC: Analog Digital Converter

A/D: Analog-to-Digital

CPLD: Complex Programmable Logic Device

CPU: Central Processing Unit

CWT: Continuous Wavelet Transform

CUSUM: Cumulative Sum

DSP: Digital Signal Processor

DWT: Discrete Wavelet Transform

DMA: Direct Memory Access

EPE: Early Power Estimator

FPGA: Field Program Gate Array

FIFO: First In First Out

GSR: Global Set/Reset

HPC: High performance computing

IP: Intellectual Property

IRQ: Interrupt Request

KAREN: Kiwi Advanced Research and Education Network

LE: Logic Element

LPM: Library of Parameterized Module

LFSR: Linear Feedback Shift Register

LUT: Look up Table

LAB: Logic Array Block

LOFAR: Low Frequency Array

MHz: Mega Hertz

mW: Milliwatts

Ns: Nanosecond

OCWT: Over Complete Wavelet Transform

PC: Personal Computer

PCI: Peripheral Component Interconnect

RTL: Register Transfer Level

RRAT: Rotating radio transients

RAM: Random Access Memory

SKA: Square Kilometre Array

SKS: Sigle Kernal Simulation

SRAM: Static Random Access Memory

SDRAM: Synchronous Dynamic Random Access Memory

SOPC: System on a Programmable Chip

TREAD: Transient Radio Emission Array Detector

TB: Terabyte

VDHL: Very High Speed Integrated Circuit Description Language

Chapter 1 Introduction

Radio transients are non-periodic, discrete signal, obtained from high energy physical processes in space. This includes solar flares, supernovae, pulsars, quasars and active galaxies. Other speculations include evaporating black holes, colliding neutron stars and a number of unknown events. The detection of radio transient presents a challenge due to their short and non-periodic nature, as well as the high risk of misdetection [1]. To undertake their detection requires the backend of radio telescopes to be equipped with the appropriate hardware and software. Generally, a de-dispersion procedure is used to improve detectability and test the property of the signal. However, due to the large scale of the signal from outer space, the computational demands of this method appear insufficiently robust [2]. Therefore, a new improved algorithm and method for detecting transient signals is developed.

1.1 Background of Research Radio Transient

Radio transients are energetically charged atomic particles. About 89% of radio transients are simple protons or hydrogen nuclei, 10% are helium nuclei or alpha particles, and 1% consists of the nuclei of heavier elements. Solitary electrons (much like beta particles, although their ultimate source is unknown) constitute much of the remaining 1% [4]. This variety of particle energies reflects the wide variety of sources. *Figure 1-1* shows the energy spectrum for radio transients. The axis stands for the cosmic ray flux and y coordinate stands for particle energy. The flux for the lowest energies (yellow zone) are mainly attributed to solar cosmic rays, intermediate energies (blue) to galactic cosmic rays and highest energies (purple) to extragalactic cosmic rays.

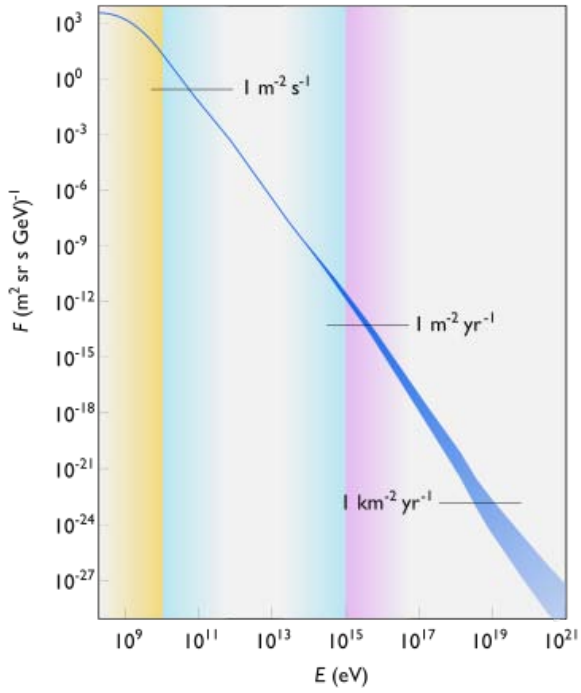


Figure 1-1 Energy spectrum for radio transients [4]

Transient radio emissions in space have been recognized as one of the key factors in the discovery of new objects and phenomena. In radio astronomy, a transient radio emission can be defined as a non-periodic short burst of electromagnetic radiation. Emissions with duration of less than a few seconds are often referred to as fast transients, and those with duration longer than a few seconds are referred to as slow transients. A number of astronomy phenomena are known to produce transient emissions. One well known phenomenon is the sun, which can continuously produce radio transients from 10 ms to week-long storms. The cause of a solar transient is the occurrence of activity on the surface of the sun, such as thermal radiation or plasma emission. Detecting and researching these transients can lead to greater understanding of the relations between stars, with radio telescopes (such as LOFAR (Low Frequency Array) and SKA (Square Kilometre Array)) extending our opportunities for detecting transients [5].

Several types of transients have been recognized. Gamma-ray burst is one of them. It has been observed that Gamma-ray has a short wave length emission, high electromagnetic energy and strong penetrability. In space, Gamma-ray is produced by the fusion core of the star [4]. As it cannot penetrate to the earth's atmosphere, it only can be detected in space. Gamma-ray was first observed in 1967 by the satellite "Vilas".

From the early 1970s, scientists have found hundreds of stars and black holes by analysing various Gamma-ray images provided by different satellites. In the process, some other mysteries of astronomical phenomena have also been solved, such as the origin of supernovae and quasars.

Rotating radio transients (RRAT) are sources of short, moderately bright, radio pulses. First discovered in 2006, RRATs are thought to be associated with rotating magnetized neutron stars [4]. The general character of pulses from RRATs is short in duration, lasting from a few milliseconds, with the radio emission from RRATs typically detectable at less than one second per day. While the analysis of RRATs allows us to obtain some information about pulsars, it is still unclear exactly how RRAT pulsars and other sources of radio bursts relate to each other.

Some other types of transient radio activities previously postulated have yet to be observed or detected. Scientists have predicted that supernova should radiate an electromagnetic pulse at radio frequencies during their collapse. Black hole vaporization is another example of transient radiation proposed by theorists. Failure to observe these transients cannot yet to be used as proof of their non existence. Transient signal detection is not easy and normally requires different methods of approaches [5].

1.2 Project Description

This project is part of the “Transient Radio Emission Array Detector” project. The overall project aims to support the establishment of transient radio emission array detectors in order to facilitate the exploration of electromagnetic phenomena within our environment both on earth and in space.

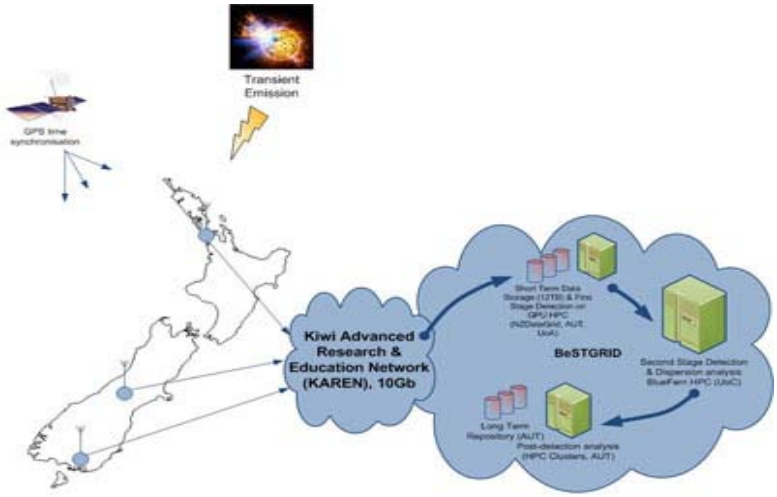


Figure 1-2 Development of TREAD in New Zealand [3]

Transient radio emission array detectors are currently set up in three locations within New Zealand. A high performance computer is used to analyse the data detected. *Figure 1-2* illustrates the development of TREAD (Transient Radio Emission Array Detector) in New Zealand [3]. *Figure 1-3* shows one of the antenna sensors on the field. Each sensor element in the array produces tens of MB of raw data each second, generating several TBytes of data daily. The data is pre-processed to detect pulse-like signals and then streamed over KAREN (KIWI ADVANCED RESEARCH AND EDUCATION NETWORK) for further processing and storing [3].

Figure 1-3 Antenna-Sensor [3]

Dealing with Tbytes of raw data every day is a difficult task. There are generally three steps to analyze the raw data. The first step is to develop a more accurate and effective detectable algorithm. The second step implements the algorithm via a software programme to verify that all the functions work well. The last step is to enhance the computational speed and signal processing ability.

1.3 Related Works

Recently, most solutions developed have been based on software programs targeted for general purpose processors. The defects of these platforms are obvious. For example, these platforms are usually constrained by a fixed number of processors, a limited operating speed and a fixed bandwidth, and are characterised by high power consumption. Most importantly, not all the resources on such platforms are used for transient signal operation, with some parts of the resources consumed by the operating system and software setup. Therefore, the development of a customised hardware platform would be the optimal solution. This platform would be used to optimise only transient signal operation, thereby saving on the cost of other unnecessary components. In addition, the power consumption of the customised platform would be much lower than a general purpose computer.

The most common technologies available to achieve this result are the Digital Signal Processor (DSP), the Application Specific Integrated Circuit (ASIC), and the Field Program Gate Array (FPGA).

DSP is a particular microprocessor used in fast digital signal processing, and is usually used to measure, filter and compress continuous real-world analog signals. It is widely implemented in image and voice processing. Heavy calculations such as floating point calculation and fast Fourier transformation can be done efficiently via DSP. For the designer however, it takes a long cycle to develop an algorithm using C or assembly language, and is not conducive to the rapid algorithm validation and product development required. Moreover, DSP is limited by its processing frequency and necessitates the use of a general purpose processor.

ASIC is an integrated circuit customised to perform a certain task. It can be divided into full-custom design and semi-custom design. Full-custom design requires the designer to complete all aspects of circuit design and takes a lot of time and resources. It has high flexibility and runs faster than the semi-custom design option. Semi-custom design allows the designer to select logical cells from standard libraries such as ALU, memory, data bus, IP Core, etc. As the layout of these logical cells is already complete and fully tested, the designer can easily complete the system design. The advantages of ASIC include low power consumption, high operating frequency and high logical density. The disadvantages are that ASIC systems come with a high design cost and require specialized designer knowledge. Further, because ASICs are non-reprogrammable, the design must be finalised before production.

FPGA is a field programmable gate array developed from PAL, GAL, and CPLD devices. It contains a flexible array of simple interconnected 'logic cells,' and programming these logic cells and their interconnection to creates the digital circuit *(Figure 1-4)* [10].

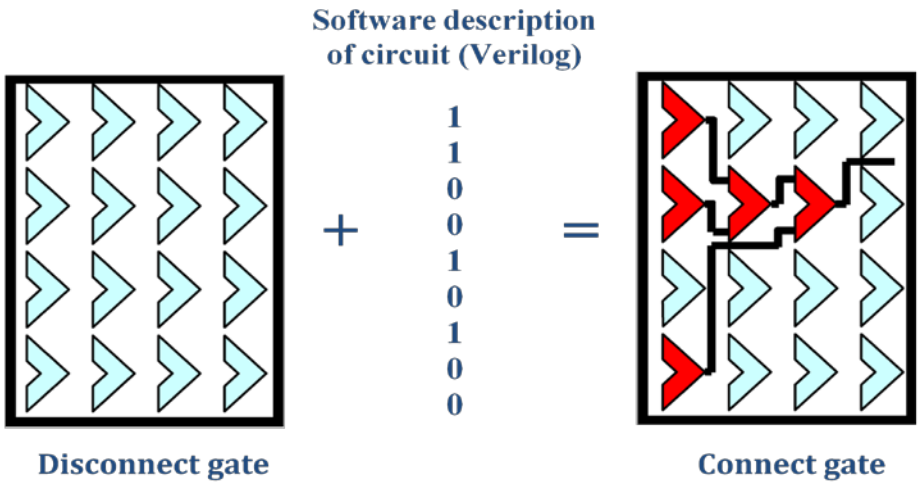


Figure 1-4 Field Programmable Gate Array

FPGA is a specific integrated circuit (ASIC) within the field of semi-customised circuits that solves the lack of customised circuits and overcomes the existing limitation of gate numbers. The circuit is designed in hardware description language (Verilog or VDHL) allowing easy layout and burn to the FPGA chip. FPGAs can be reconfigured at any

time, resulting in a lower non-recurring engineering cost and enabling a faster time to market.

Another important application of FPGA is its function as a co-processor for a High Performance Computer. High performance computing (HPC) entails the use of parallel computing systems to solve difficult computational problems. The HPC platform, as shown in (*Figure 1-5*), consists of a number of distributed computing nodes (Computer Node), each node connected by some interconnections and associated with a reconfigurable computing unit (Configurable ICN) [6]. This architecture provides the user with greater computational performance than traditional parallel computers.

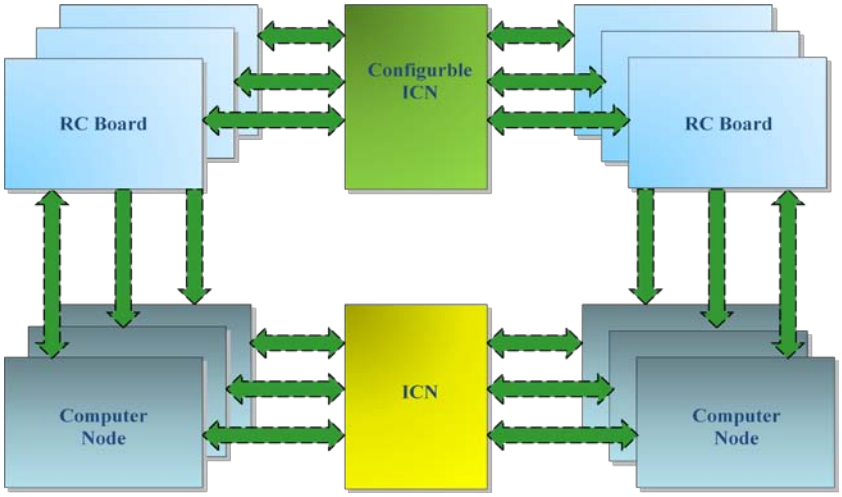


Figure 1-5 Architecture Diagram of HPC

HPCs can be classified as either Loosely-coupled co-processors or Tightly-coupled co-processors. A co-processor described as loosely-coupled means that the FPGA plug-in CPU serial bus and CPUs are connected together via interconnection networks. This model is commonly found in smaller system CPUs and is characterised by poor parallel performance due to the serial bus being easily overloaded and the length of time required transferring data [9]. By contrast a tightly-coupled co-processor means the FPGA module is directly connected to the interconnection network and delivers great parallel performance. One downside of this model is that it requires a reasonably complex network topology [7].

However, it seems easier and more reasonable to implement algorithms via FPGAs, with the design on FPGAs be able to migrate to HPCs or ASICs in most cases. The remainder of this thesis discusses the design of transient signal detection algorithms based on the above technologies.

1.4 Research Aims and Objectives

A novel approach of using FPGAs to implement the transient signal detector is presented in this thesis. The system proposed is based on CUSUM algorithm IP core design. Algorithms and hardware level optimization have been employed to improve the performance of transient signal detection and reduce latency. The algorithm was simulated by ModelSim and implemented via DE2-115 board with Cyclone IV families.

1.5 Thesis Structure

This chapter has introduced an overview of transient signal detection and its possible applications. A survey of the most common embedded technologies that may be used for its implementation was subsequently presented. Finally, the aim and objectives of this research were discussed.

The rest of the thesis is organised as follows. A literature review on transient signal detection is presented in Chapter 2. The comparison of different algorithms is discussed in this chapter with the chosen CUSUM algorithm explained in detail.

Chapter 3 gives a general introduction of FPGA technologies. The Altera FPGA, design tools, Simulation tools and the DE2 115 development board are introduced in detail. The Mega Core components, which are used in design, are also explained.

Chapter 4 explains the technologies which are used to reduce the power consumption of Cyclone IV FPGA. The power consumptions of the CUSUM IP core are tested. The results are further analyzed.

Chapter 5 explores the architecture of CUSUM Core design. Several optimization methods are presented. These include Pipeline, Ping-Pong, Serial to Parallel, and Cross

Clock domain design methods. Finally, the module setup was present as a key point of this chapter.

Chapter 6 explains the test bench design methodology. ModelSim simulation results are plotted and Time Sequential Analysis is discussed.

Chapter 7 concludes this research project by summarising the methodology employed and highlighting the milestones achieved. Finally a discussion of future research that could be undertaken to extend the current work is presented.

Chapter 2 Transient Signal Detection

The first section of this chapter will review and discuss the existing solutions of transient signal detection. The next section will then present CUCUM algorithm application in transient signal detection. Finally, a conclusion is drawn based on the topics discussed.

2.1 Literature Review

Transient signal detection can be considered as a complex stochastic model. Any abnormal signal can affect changes of the model. The aim of detection is to monitor the difference between input signals with the threshold. Currently, there are two ways to monitor those changes; one is from the perspective of signal processing, the other is from a statistical point of view.

2.1.1 Signal Processing Method

Signal processing methods usually transform the sampling signal to a time domain or frequency domain and observe the changes. In [11], Cornel Loana provides an adaptive time-frequency method based on the over-complete wavelet transform concepts, which lead to signal processing on interest frequency bands. This method is based on the fourth order moment, and is applied for each sub-band, in order to establish the optimal weight for each sample. The result obtained proves the capability of the proposed approach to accurately detect a transient signal, when compared with other methods (e.g. Spectrogram or Standard Wavelet Transform) [26].

The author [11], discovered that the commonly used method, discrete wavelet transform (DWT) was not well suited to this kind of signal processing problem. From a mathematical point of view, the discrete wavelet transform (DWT) is generated by the sampling in time-scale plant of a corresponding continuous wavelet transform (CWT). Despite the fact that there is an infinite possible discretization of CWT, the term discrete wavelet transform (DWT) is commonly used to refer to that associated with the dyadic sampling lattice [27]. In certain analyses it will cause wavelet orthogonal basis and the

use of orthogonal representation will lose the signal characteristics [12]. In order to eliminate this drawback, the key factor is the use of a non-dyadic sampling structure, which in this case is the Over Complete Wavelet Transform (OCWT). This method can be separated into two stages: the first is to decompose the signal with the linear filter bank structure. The second stage is to sample the signal issue at the filter bank output.

The author [11] also proposes an irregular sampling procedure for OCWT. Generally, there are some advantages in adopting an irregular sampling method. The theoretical reasoning behind the irregular sampling method is detailed later in this work [13]. The application of irregular sampling in data acquisition is considered a non traditional approach. This approach requires a more complicated data acquisition process and still needs to be improved [25].

Melvin J. Hinich [15] has proposed to use bispectral analysis for detecting transient signals. This method uses a statistic computed from the sample bispectrum of a sampled record. The key result underlying the method is that the bispectrum of the noise is zero in a triangle that is a proper subset of the principal domain triangle. The result implies that the bispectrum based test may detect a weak signal of unknown form which evades detection by other methods. However, the limitation of this method is the signal's frequency band must lie within the interval (0, f) and the duration value T should fall within some error band [28].

2.1.2 Statistical Method

The aim of the statistical method is to discover the characteristics of the sampled signal. There are many statistical methods can be used. For example, we know the simplest technique for testing a signal change is the mean value. While regression analyses can be used to detect changes, they are not very sensitive to small deviations. The change detection problem can also be solved by means of a Bayesian analysis [16] when a mathematical model of the data is available.

The log likelihood ratio method provides another option. Log likelihood ratio is a powerful and sufficiently generic method of testing model assumptions [1]. It is based on using a ratio of two probability distribution functions (*pdf*) to build an indicator upon which a threshold can be applied. If the ratio exceeds a given threshold, it indicates the

transient signal has been detected. However, this method needs to know the distribution to which the sampling belongs.

Generally speaking, statistical methods can be divided into two categories, parametric and non-parametric. The parametric method is observed via the value of equal size sampling in the unit time period, while the non-parametric method is observed via continuous changes of sampling and is determined by whether the sampling follows statistical distribution or not. The parametric method relies on an assessment of the overall information (overall distribution, characteristics and variance) to analyze the sampling feature and can only be used for sampling equivalent intervals. By contrast, the non-parametric method does not require an assessment of overall information. Rather, it uses the sampling information to speculate the overall distribution. From a statistical point of view, the non-parametric method is more suitable for solving transient detection problems as it does not require the sampling of data from any particular distribution.

The Mann-Whitney U and the Wilcoxon signed-rank are both non-parametric statistical methods. The Mann–Whitney U [17] is used for assessing which of two independent observations have larger values than the other and is one of the most well-known non-parametric significance tests. The Wilcoxon [18] signed-rank test is used when comparing two related samples, or repeated measurements on a single sample, to assess whether their population means differ (i.e. a paired difference test). This method is based on the assumption that there is no significant difference between the two samples' overall distribution. The main limitation of these methods is that they were originally designed for detecting single point changes. By contrast, the Mann-Kendall [19] and the CUSUM methods are particularly suitable for sequential analysis.

Specifically, the Mann-Kendall method is used to measure the association between two measured quantities. It is easy to implement and widely used in the analysis of climate change. CUSUM is a sequential analysis technique typically used for monitoring change detection [19]. It has several advantages including its relative simplicity, a graphical interpretation of results, and the ability to detect unusual patterns. It has been successfully used in fault detection, onset detection, and defect detection in mechanical systems. Both the Mann-Kendall and the CUSUM tests have particular parameters that need to be fixed at design-time [24] in order to allow the test to detect changes.

Specifically, the Mann-Kendall test requires setting a level of significance for the test, while the CUSUM test needs to fix the thresholds in order to detect the possible changes in statistical behaviour. One of the significant benefits of CUSUM for signal detection is its stability in the presence of regression behaviour for signal sampling [19]. The CUSUM test was chosen as the method of implementation for this research due to its high detection accuracy and real time computation. This chapter introduces the CUSUM algorithm in detail.

2.2 CUSUM Algorithm

CUSUM is a detection procedure proposed by Page (1954) and Lorden (1971) [20]. It is a sequential analysis technique in statistical quality control, typically used for monitoring change detection. As its name implies, CUSUM involves the calculation of a cumulative sum (making it "sequential").

2.2.1 CUSUM Equation

In this algorithm, a constant reference value is subtracted from the data collected. This difference is added to the previous difference (the cumulated sum). Usually this average value is referred to as “M.” The equation is summed as below:

2-1:

$$S_1 = (X_1 - M)$$

$$S_2 = (X_1 - M) + (X_2 - M) = S_1 + (X_2 - M)$$

$$S_3 = S_2 + (X_3 - M)$$

$$\dots\dots\dots\dots$$

$$S_n = S_{n-1} + (X_n - M)$$

The CUSUM test requires a reference threshold value h. When the value of Sn exceeds a certain threshold, an abrupt change can be detected.

Figure 2-1 shows the implementation of the CUSUM test in signal processing. The first part shows random input signals. Second and third part shows the error has been accumulated. The ascending line conveys that the signal runs higher than the reference

value while the descending line means the signal runs below this value. The horizontal line shows the signal runs at the reference status.

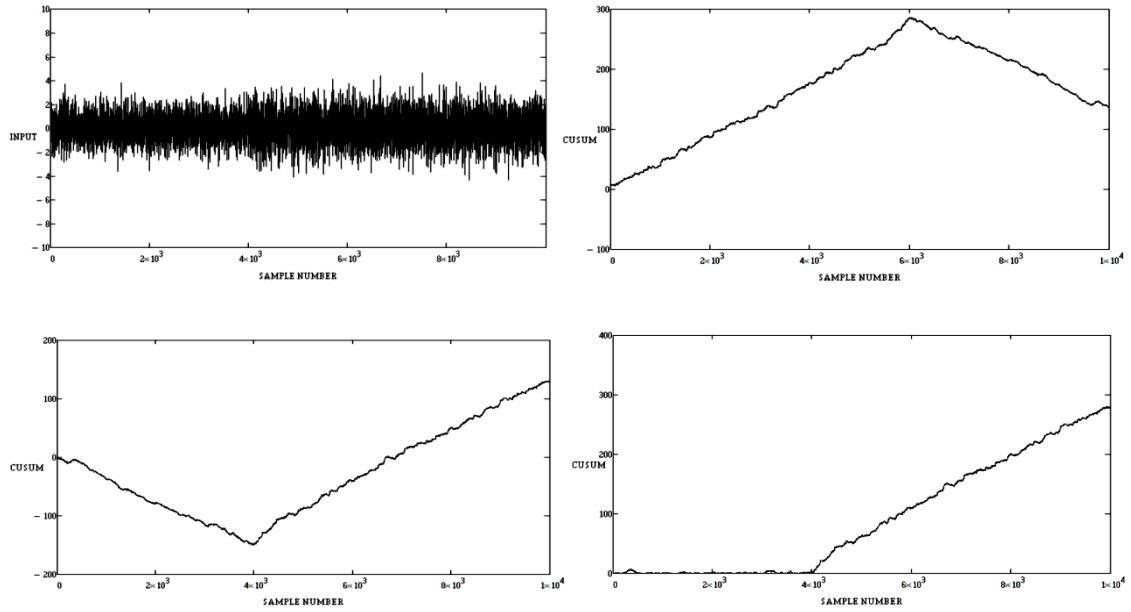


Figure 2-1 CUSUM test implementation in Signal Processing [2]

2.2.2 Threshold Selection

Threshold selection is the key point for CUSUM detection. However, there is no fixed method to detect transient signals. In [1] the author discovered a method in which the threshold can be derived from Wald Sequential tests on the mean of a normal population. Assuming that $\mu_1 > \mu_0$, the value of threshold h is equivalent to a sequence of Wald Sequential tests. The formula can thus be expressed as:

2-2:

$$h = -\frac{\sigma_1^2}{\sigma_0^2}\frac{\ln \alpha}{(\mu_1 - \mu_0)}$$

Where α is interpreted as an approximation of the proportion of samples that trigger false alarm. This value can also be interpreted as a probability of false alarm in a traditional sense. If it assumes that the variance $\sigma_1^2 = \sigma_0^2$, the formula can be rewritten as:

2-3:

$$h = -\frac{\ln \alpha}{(\mu_1 - \mu_0)}$$

As the value of $\mu_1 - \mu_0$ is generally unknown, it can be considered as a parameter that specifies the maximum value of difference between mean values of noise and signal. Meanwhile, to select an appropriate denominator for this formula is a complex task, especially in the context of the unknown character of signal detection. Obviously, the more closely chosen the value, the more sensitive the CUSUM processing will be.

Another method used to determine threshold value is the Standard Deviation (SD) method. SD is a method widely used in statistics to measure variability or diversity [21]. It shows the extent of variation offset from the mean or expected value. As it is an important indicator of precision in statistics it is widely used in quality control. For example, if there are two groups of data with the same average value, group A has a larger SD than group B. This means the spread in group A is much larger than in group B. In other words, a low SD indicates that the data points tend to be very close to the mean, whereas a high SD indicates that the data points are spread out over a large range of values [11]. Here SD was selected as threshold value because it can achieve a relatively fast calculation time with quite accurate results.

The SD of a data set is the square root of its Variance. A useful property of SD that, unlike Variance, it is expressed in the same units as the data. The SD equation is:

2-4:

$$S_n = \sqrt{\frac{\sum_{i=1}^{n}(X_i - \bar{X})^2}{N}}$$

Here the $\bar{X}$ refers to the mean value of the sampling unit. In transient signal detection, to find an overall SD is unrealistic, however, one can choose a certain amount of sampling and use the SD of the sampling as its own threshold. In this design, two times the SD of each sample set is selected the sample set threshold. The reason for this is explained below.

In probability theory, the Normal Distribution is a continuous probability distribution that is often used to describe real-value random variables that tend to cluster around a single mean value [22]. The Normal Distribution is considered the most prominent probability distribution in statistics. There are several reasons for this. First, the Normal Distribution is very tractable analytically; that is, a large number of results involving this distribution can be derived in explicit form. Second, the Normal Distribution arises as the outcome of the central limit theorem, which states that under mild conditions the sum of a large number of random variables is distributed approximately normally. Finally, the "bell" shape of the normal distribution makes it a convenient choice for modelling a large variety of random variables encountered in practice. For this reason, the Normal Distribution is commonly encountered in practice, and used as a simple model for complex phenomena [23]. In Normal Distribution, one SD stands for 68% of overall values and two SD are representative of 95% of overall values. *Figure 2-2* shows a plot of normal distribution with each band the width of one SD. In transient signal detection, if a sampled signal extends over two times the SD this indicates that the signal is abnormal.

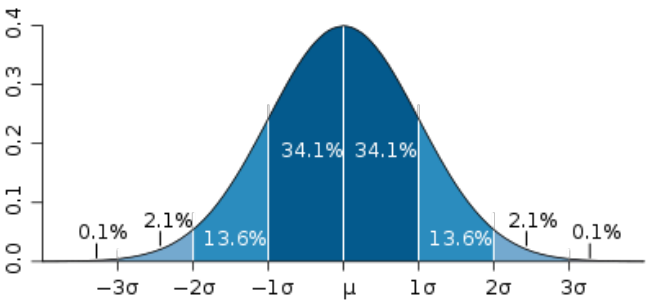


Figure 2-2 Standard Deviation in Normal Distribution

The choice of the size of sampling unit is important, because it’s related to detection accuracy. The standard error equation below expresses the relation between sampling size and standard error [29].

2-5:

$$\sigma = \frac{1}{\sqrt{N}} \times S_n$$

This equation reflects the degree of dispersion of samples. The smaller of standard error means the samples close to overall average value, otherwise the samples appears more discrete. Obviously, more samples are chosen in each sampling unit more closer to overall standard deviation. In practical, sampling size normally choose around 50. Here, the author chooses 32 samples as a sampling unit in design. The main reason is reduced the incubation period for sampling data processing, details will explain in Chapter 6.

2.3 Computer Simulation

The initial simulation framework was developed by Gene Soudlenkov [1] based on MATLAB ®7.9.0. Simulation of transient signals was achieved by mixing a pulse of the desired noise with the result plotted in the time domain. A simulation was undertaken for three cases, each making the detection increasingly more difficult. The difficulty was increased by increasing the Signal to Noise Ratio (SNR). Where σ^2_{signal} and σ^2_{noise} represent variance of the signal and noise, the calculation of SNR is:

2-6:

$$\text{SNR} = 10 \log_{10} \frac{\sigma^2_{\text{signal}}}{\sigma^2_{\text{noise}}}$$

Where σ^2_{signal} and σ^2_{noise} are variance of the signal and noise. This simulation approach provides well controlled signal shaping and can be accommodated for a wide variety of signal mixes. Three cases simulations are shown as below

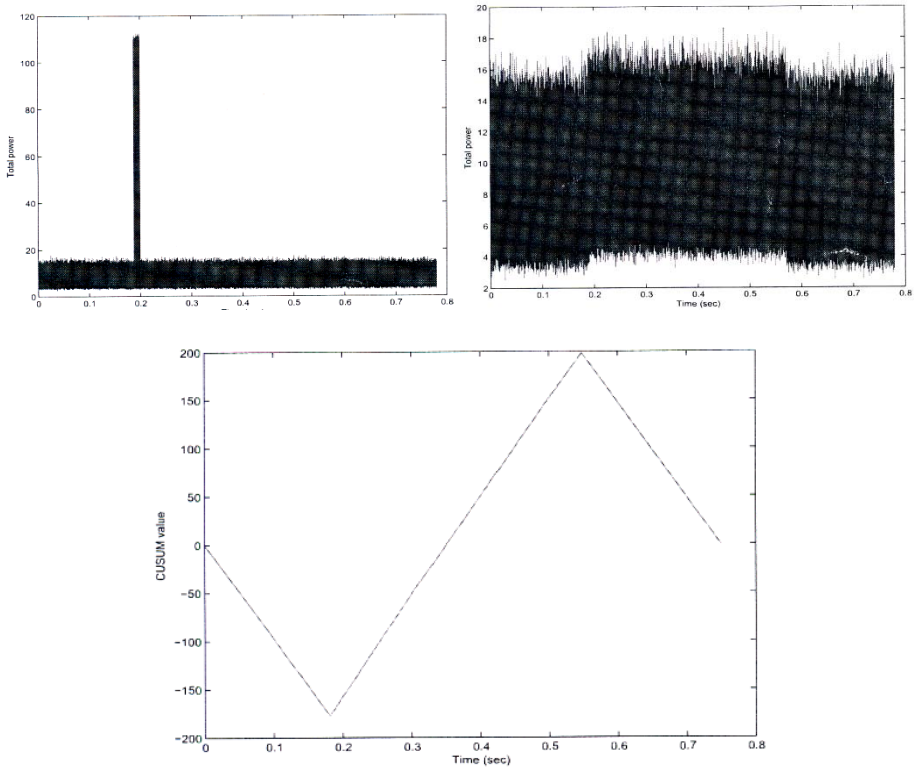


Figure 2-3 Simulated pulse with additive Gaussian noise SNR = 2dB, pulse onset = 0.18 sec, pulse duration = 0.38 sec [2]

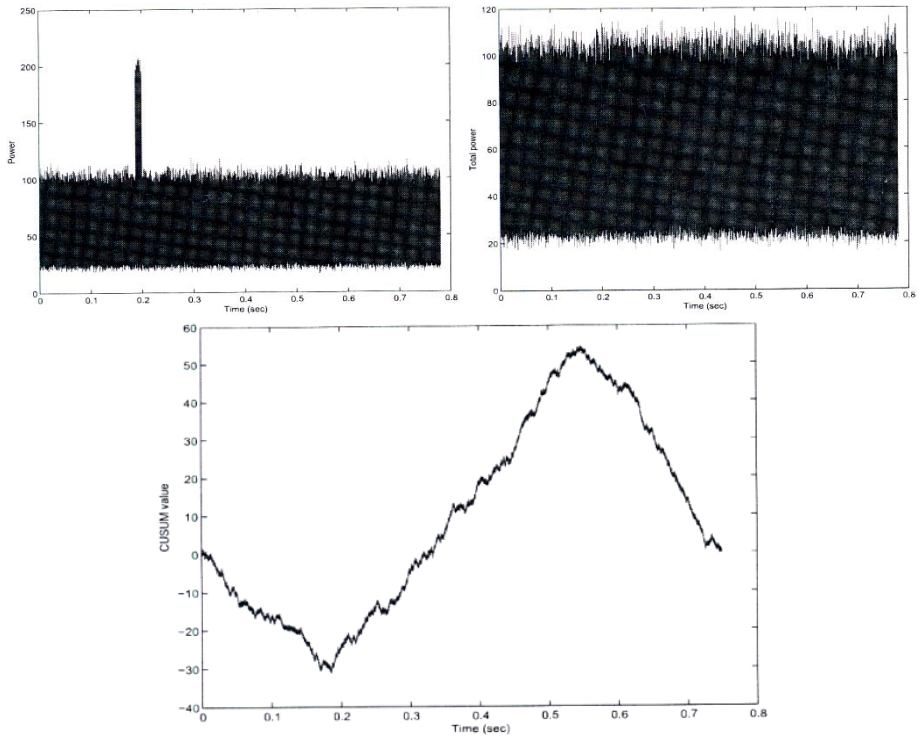


Figure 2-4 Simulated pulse with additive Gaussian noise SNR = -26dB, pulse onset = 0.18 sec, pulse duration = 0.38 sec [2]

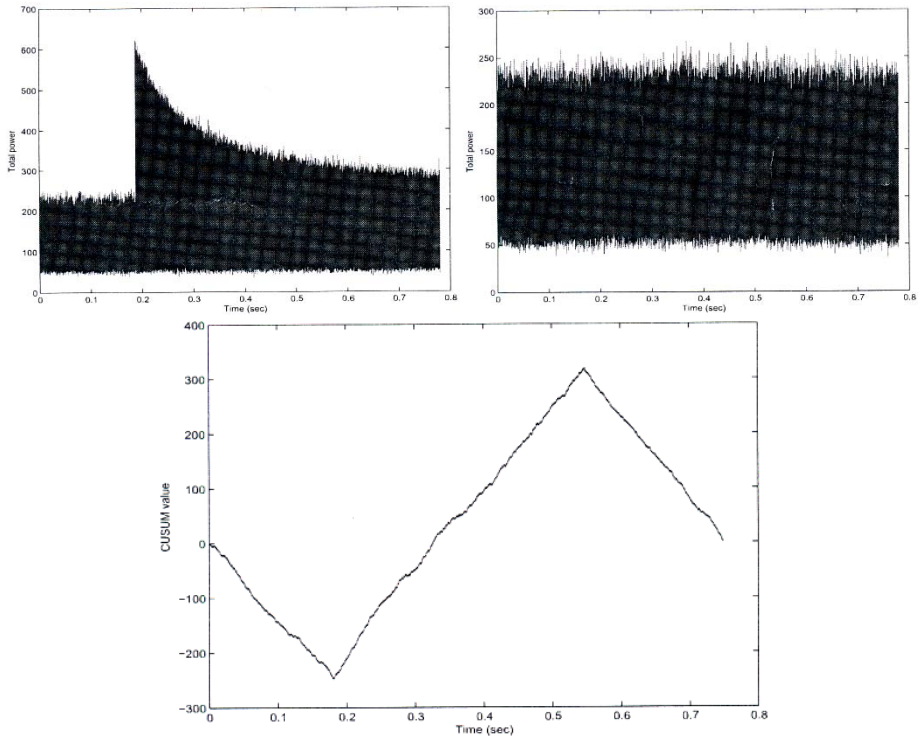


Figure 2-5 Simulated pulse with additive Gaussian noise SNR = -32dB, pulse onset = 0.18 sec, pulse duration = 0.38 sec [2]

2.4 Summary

Different algorithms were discussed in this chapter with the CUSUM algorithm explained in detail. In comparison with other transient detection algorithms, the CUSUM algorithm was shown to achieve a relatively faster speed with reasonably accurate results. For this reason the CUSUM algorithm was chosen for this design.

Chapter 3 FPGA Implement Platform

The CUSUM algorithm was implemented using an Altera Cyclone II FPGA development board. The FPGA based system was developed under Quartus ® 10.1 and simulated by ModelSim ® 6.6 software. An overview of the FPGA and Mega Function core is given in this chapter. The implementation is then presented.

3.1 Introduction to FPGA

The Field-Programmable Gate Array (FPGA) is a semiconductor device that can be programmed after manufacturing. As it overcomes the restrictions of any predetermined hardware function FPGA allows the designer to program product features and functions, adapt to new standards, and reconfigure hardware for specific applications, even after the product has been installed in the field - hence the name "field-programmable." FPGA can implement any logical function that an application-specific integrated circuit (ASIC) can perform but with the additional ability to update the functionality after shipping, which offers advantages for many applications.

Compared to ASICs, FPGAs offer many design advantages, including:

- Rapid prototyping
- Shorter time to market
- The ability to re-program in the field for debugging
- Lower NRE costs
- Long product life cycle to mitigate obsolescence risk

Unlike previous generation FPGAs using I/Os with programmable logic and interconnects, today's FPGAs consist of various mixes of configurable embedded SRAM, high-speed transceivers, high-speed I/Os, logic blocks, and routing. Specifically, an FPGA contains programmable logic components called logic elements (LEs) and a hierarchy of reconfigurable interconnects that allow the LEs to be physically connected.

It can configure LEs to perform complex combinational functions, or merely simple logic gates like AND, XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

Altera [30] and Xilinx [31] are two major manufacturers in the current FPGA market. They take over 80% of the market share, with Xilinx alone representing over 50% [32]. Other manufacturers include Lattice Semiconductor [33], Actel [34], SiliconBlue Technologies [35], Achronix [36] and QuickLogic [37].

3.2 Altera FPGA

The Altera Company is the world's pioneer of FPGA solutions and was founded in 1983. Altera combines reprogrammable logic technology with software tools, intellectual property (IP), and design services to provide high-value programmable solutions worldwide. Its reprogrammable solutions deliver fast time to market and other significant advantages over costly, high-risk ASIC development and digital signal processors [38]. Altera's products have been widely used in many end markets including the automotive industry, the audio and video equipment industry, and the computer industry, as well as the storage, medical equipment, military, and telecommunications industries [39].

3.2.1 Altera Device Family

There are three series of FPGAs manufactured by Altera; Cyclone, Arria and Stratix. Cyclone ® series FPGAs are the industries least expensive, most power efficient FPGAs, ideal for high-volume, cost-sensitive applications. Comparably, the Arria ® series FPGAs provide an optimal balance of performance, power, and price for mid-range transceiver-based applications. Stratix ® FPGAs are the industry's highest bandwidth, highest density FPGAs and ideal for high-end applications and are designed for high performance products.

Considering the advantages of lowest cost, most efficient use of power and high-volume, the Cyclone ® series FPGAs are in general better suited for applications. A Cyclone IV development kit with a Cyclone® IV EP4CE115F29C7 device was therefore selected for the design.

3.2.2 Comprehensive Development Suite

Altera's comprehensive development suite includes Quartus II, a customized version of ModelSim, Nios II Embedded Design Suite and DSP Builder. Numerous design features, including various design entry, scripting support, incremental compilation, SOPC Builder, MegaWizard plug-in manager, I/O pin assignment analysis, Quartus II integrated synthesis, rapid recompile, third-party design entry and synthesis, and basic compilation flow are offered in Quartus II to accelerate the design process. ModelSim is used for the functional simulation of the design. Nios II Embedded Design Suite includes a collection of cutting-edge software tools, utilities, libraries and drivers to help bring the design to market within a short time. DSP Builder allows the designer to implement high-performance DSP functionality on FPGAs by using Matlab ® Simulink as the modelling, simulation and implementation environment.

3.3 Cyclone IV Devices and Development Board

All the experimentation of this research work is implemented on a DE2- 115 board to verify its function. The main elements in all FPGAs are Logical Elements (LEs), Memory Blocks, DSP blocks, PLLs and User I/Os. Table 3-1 shows a summary of resources on Cyclone IV device.

Table 3-1 Resources on DE2 – 115 Development Board

Cyclone IV Device	EP4CE115F29C7
Logical Element (LEs)	144,480
Embedded Memory (Kbit)	3,888
Embedded Multipliers (18 x 18)	266
General Purpose PLLs	4
User I/O Bank	8
User I/O	528

3.4 Altera SOPC Builder

SOPC Builder (System on a Programmable Chip Builder) is Altera software that automates the connection of soft-hardware components to create a complete computer system that runs on any of its various FPGA chips [40]. SOPC Builder incorporates a library of pre-made components (including the flagship Nios II soft processor, memory

controllers, interfaces, and peripherals), as well as an interface for incorporating customised components.

The SOPC Builder defines and adds custom components or selects from a list of provided components. By connecting multiple modules together to create a top-level HDL file called the SOPC Builder system, the SOPC Builder generates a system interconnect fabric that contains the necessary logic to manage the connectivity of all modules in the system. Interconnections are made though the Avalon bus with bus arbitration, bus width matching, and even clock domain crossing, all handled automatically when SOPC Builder generates the system. A GUI is the only additional tool required to configure the soft-hardware components (which often have many options), and to specify the bus topology [40].

3.4.1 Nios II Processor

The Nios II processor is a 32-bit embedded soft core processor [41]. This core allows users to add or remove features on a system-by-system basis to meet price or performance goals. A configurable soft-core processor enables us to add or remove features on a system-by-system basis to meet price or performance goals. Figure 3-1 shows a block diagram of the Nios II processor core.

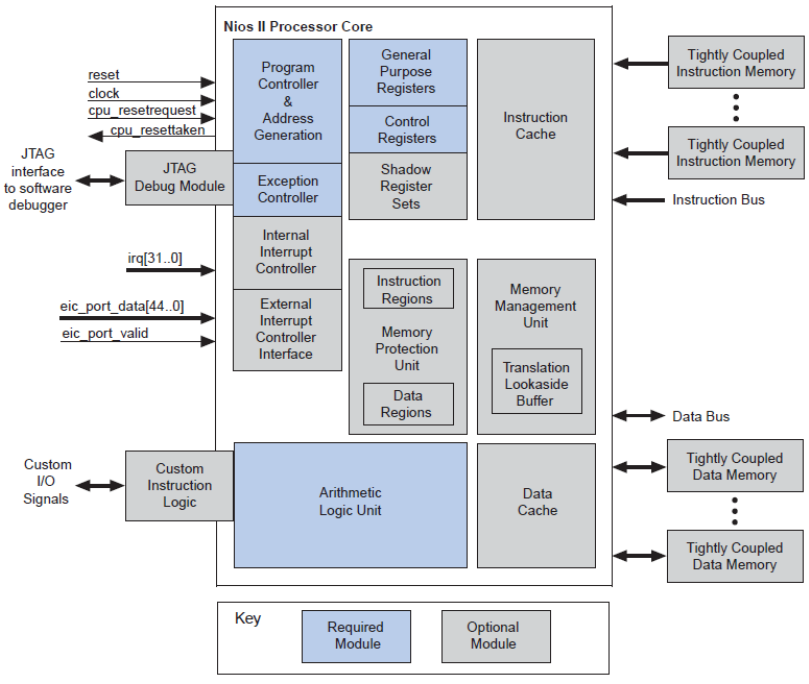


Figure 3-1 Nios II Processor Block Diagram [41]

There are three different levels of Nios II processors. Differentiated as Nios II/e (economy), Nios II/e (standard), and Nios II/f (fast), they are designed for cost-sensitivity, medium-performance and performance-critical processing respectively. The following discussion outlines the major components of the Nios II processor [41].

I. Arithmetic Logic Unit (ALU)

The main component of the Nios II processor is the arithmetic logic unit (ALU). The ALU operates by taking one or two inputs from a register and storing the result back in the register. The ALU supports the data operations described in Table3-2.

Table 3-2 Operations Supported by Nios ALU

Category	Details
Arithmetic	The ALU supports addition, subtraction, multiplication, and division on signed and unsigned operands
Relational	The ALU supports the equal, not-equal, greater-than-or-equal, and less-than relational operations on signed and unsigned operands.
Logical	The ALU supports AND, OR, NOR, and XOR logical operations.
Shift and Rotate	The ALU supports shift and rotate operations, and can shift/rotate data by 0 to 31 bit positions per instruction. The ALU supports arithmetic shift right and logical shift right/left. The ALU supports rotate left/right.

II. Instructions

The instructions in the Nios II processor are 32 bits long. In addition to the machine instructions that are executed directly by the processor, the Nios II instruction set includes a number of pseudo instructions that can be used in assembly language programs. The Assembler replaces each pseudo instruction with one or more machine instructions. There are three types of instruction formats: I-type, R-type and J-type.

- I-type – Five-bit fields A and B are used to specify general purpose registers. A 16-bit field IMMED16 provides immediate data which can be sign extended to provide a 32-bit operand.
- R-type – Five-bit fields A, B and C are used to specify general purpose registers. An 11-bit field OPX is used to extend the OP code.

- J-type – A 26-bit field IMMED26 contains an unsigned immediate value. This format is used only in the Call instruction.

III. Interrupts (IRQ)

The Nios II architecture supports 32 internal hardware interrupts. The processor core has 32 level-sensitive interrupt request (IRQ) inputs, “irq0” through “irq31,” providing a unique input for each interrupt source. IRQ priority is determined by software. The architecture supports nested interrupts. The software can enable and disable any interrupt source individually through the “ienable” control register which contains an interrupt-enable bit for each of the IRQ inputs.

IV. Memory

The Nios II architecture supports cache memories on both the instruction master port (instruction cache) and the data master port (data cache). Cache memory resides on-chip as an integral part of the Nios II processor core. The size of the caches can vary from 512 Bytes to 64 KB or omitted because the chosen sizes of caches will directly affect the execution time of code running on the Nios II processor. The other type of memory provided by the Nios II processor is tightly-coupled memory. Similar to cache memory, in that it does not have real-time caching overhead, such as loading, flushing or invalidating memory, tightly-coupled memory provides guaranteed low-latency memory access for performance-critical applications.

In practice, most FPGA designs require the addition of extra logic elements to the processor system. There are many logic elements provided by SOPC, such as PLL, Tri-Bridge, and Memory. It is also possible to custom design a component and add it to the processor. In this way the Nios II processor provides flexibility to add features to the system. *Figure 3-2* shows an example of the Nios II system.

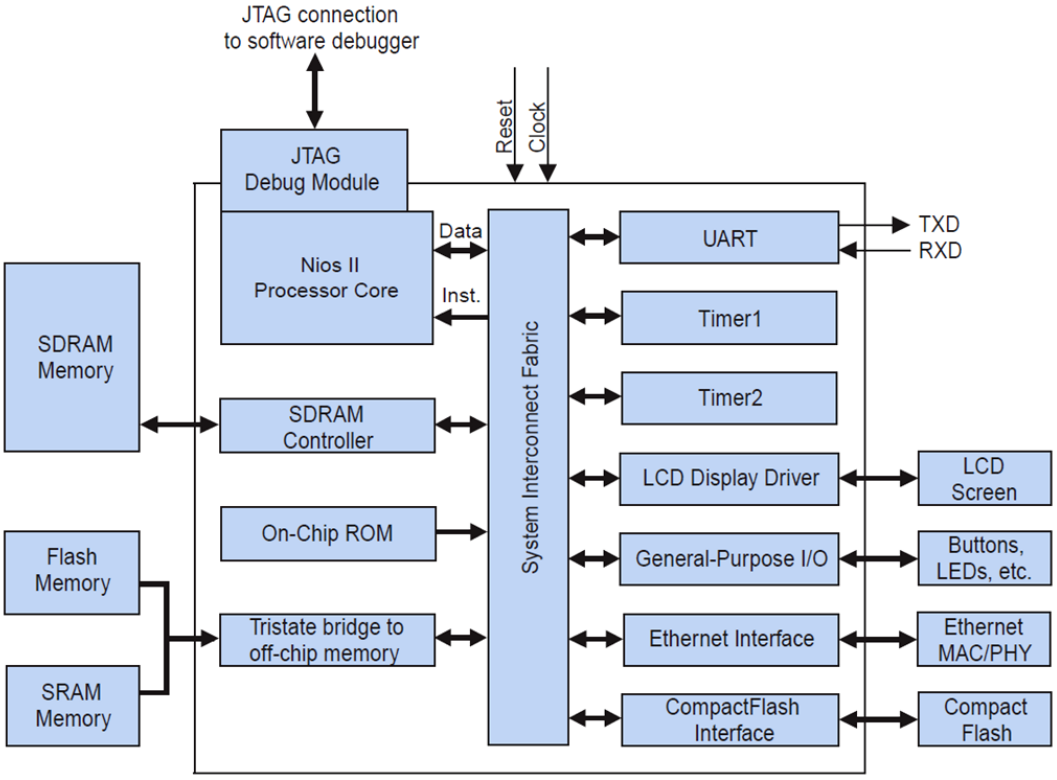


Figure 3-2 Example of a Nios II Processor System [41]

3.4.2 Avalon Switch Fabric

Avalon switch fabric is a chip-level communication bus that connects internal modules. High bandwidths interconnect structure that consumes minimal logic resources and provides greater flexibility, the main functions provided by Avalon switch fabric are: address decoding, data-path multiplexing, arbitration, clock domain crossing and interrupt controller. *Figure 3-3* shows a block diagram of Avalon switch fabric.

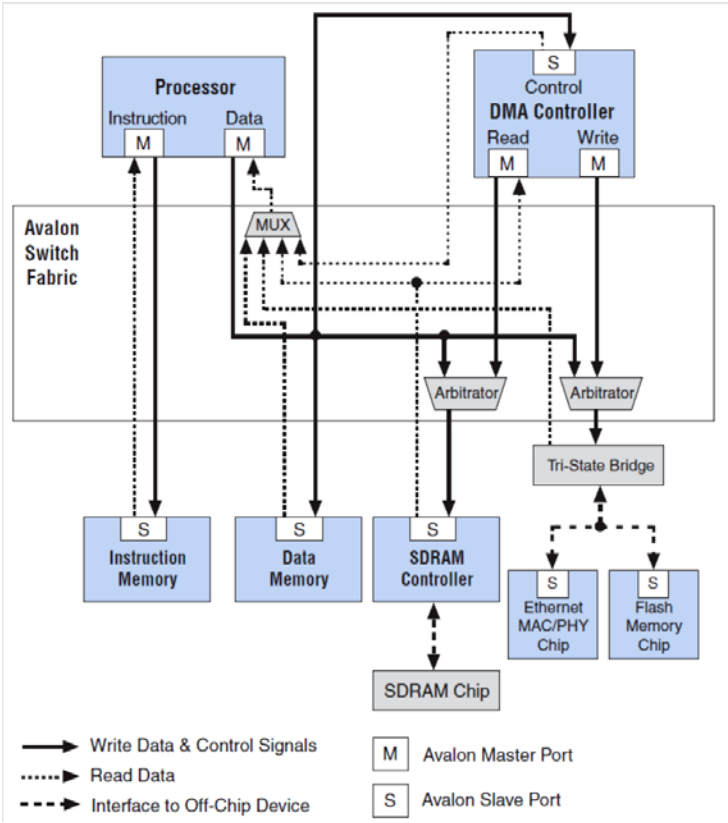


Figure 3-3 Avalon Switch Fabric Block Diagram [41]

Compared with other bus topologies like Wishbone, Avalon Switch Fabric provides a one-to-one, one-to-many, many-to-one or many-to-many communication mechanism for systems with multiple masters and slaves. It can also provide multiple channel communications. While one master is communicating to one slave, other masters can communicate with other slaves at the same time. In SOPC builder Avalon Switch Fabric is generated automatically. Users do not need to know anything about its internal functionality, thereby minimising the development time.

3.5 Mega function

Altera integer arithmetic mega functions provide the convenience of performing mathematical operations on FPGAs through parameterized functions that are optimized for Altera device architectures. These functions are customized by configuring parameters to accommodate the design requirement. Altera integer arithmetic mega functions are divided into two categories: Library of parameterized modules (LPM) and

Altera-specific (ALT) mega functions. Table 3-2 lists the mega functions as described in the Altera user guide:

Table 3-3 List of Megafunctions

Megafunction Name	**Function Overview**
LPM Mega function (LPM)	
LPM_ABS	Absolute value
LPM_ADD_SUB	Adder/Subtractor
LPM_COMPARE	Comparator
LPM_COUNTER	Counter
LPM_DIVIDE	Divider
LPM_MULT	Multiplier
Altera –specific (ALT) Mega functions	
ALTACCUMULATE	Accumulator
ALTECC	ECC Encoder/Decoder
ALTMEMMULT	Memory-based Constant Coefficient Multiplier
ALTMULT_ACCUM	Multiply-Accumulator
ALTMULT_ADD	Multiply-Adder
ALTMULT_COMPLEX	Complex Multiplier
ALTSQRT	Integer Square-Root
PARALLEL_ADD	Parallel Adder

In this research project, two Megafunctions are implemented in this design. They are ALTMULT_ADD and ALTSQRT.

3.5.1 ALTMULT_ADD Function

The ALTMULT_ADD megafunction allows the implementation of a multiplier-adder. It acts to accept pairs of inputs, multiply the values, and add all pairs together [42]. In addition, this function offers many variations within a dedicated DSP block circuit. Data input sizes of up to 18 bits are accepted and as the DSP blocks allow for one or two levels of 2-input add or subtract operations on the product this function can create up to four multipliers. Figure 3-4 shows the ports for the ALTMULT_ADD megafunction.

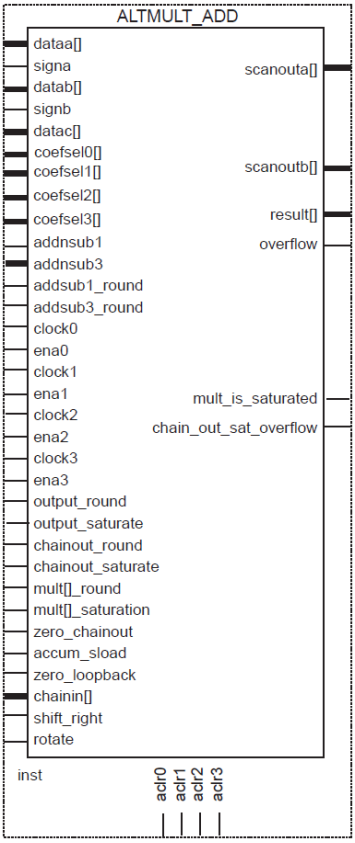


Figure 3-4 ALTMULT_ADD ports shows in MegaWizard Plug-In Manager

This megafunction offers the following features:

- Generates a multiplier to perform multiplication operations of two complex numbers
- Supports data widths of 1– 256 bits
- Supports signed and unsigned data representation format
- Supports pipelining with configurable output latency
- Provides a choice of implementation in dedicated DSP block circuitry or logic elements (LEs)
- Supports optional asynchronous clear and clock enable input ports

In project design, latency is the key point. Longer latency means more stages of pipelines and greater resources are required. It also means higher maximum clock frequency because the circuit in each stage of the pipeline is shorter. For best performance, one should choose the appropriate megafunction for the design. In this project 4 x multi-adder units were implemented in the design. Each unit requires 8 DSP

(9 bit), 100 LUT and 34 reg. The latency for each unit is 8 clock-cycles. *Figure 3-5* shows the single multi-adder unit.

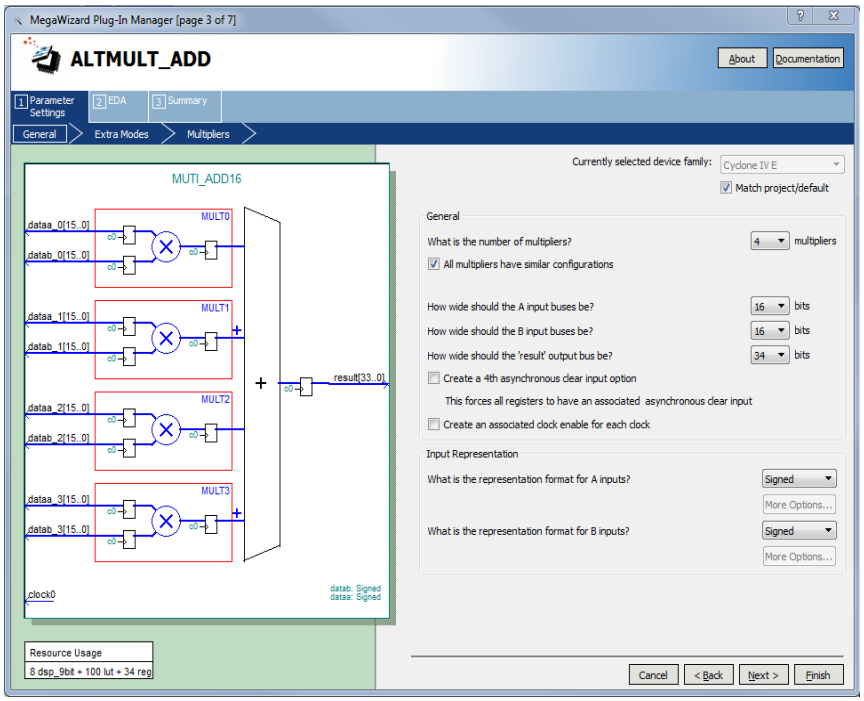


Figure 3-5 ALTMULT_ADD Unit

As a sampling is a 16-bit signed signal, all the widths of the input channels should be setup at 16 bits, and with a signed format. Table 3-4 below shows the resource usage after synthesis.

Table 3-4 Resource Usage for Single ALTMULT_ADD

Total combinational functions	1,321 / 114,480 (1 %)
Dedicated logic registers	272 / 114,480 (< 1 %)
Device	EP4CE115F29C7
Family	Cyclone IV E
Total logic elements	1,321 / 114,480 (1 %)
Total registers	272
Embedded Multiplier	9-bit elements 64 / 532 (12 %)

3.5.2 ALTSQRT Function

The ALTSQRT megafunction implements a square root function that calculates the square root and remainder of an input [42]. Figure 3-6 shows the ports for the ALTSQRT megafunction:

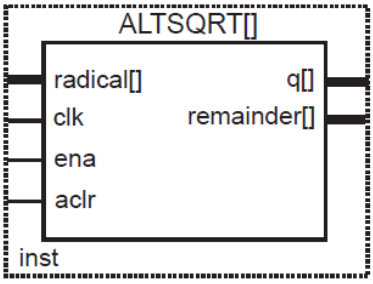


Figure 3-6 ALTSQRT ports in MegaWizard Plug-In Manager

The ALTSQRT megafunction offers the following features:

- Calculates the square root and the remainder of an input
- Supports data width of 1–256 bits
- Supports pipelining with configurable output latency
- Supports optional asynchronous clear and clock enable input ports

This module is used to calculate a square root output from the previous step. One should note that the width of the radical must match the previous output; otherwise this module will not produce a result. Figure 3-7 shows the ALTSQRT unit.

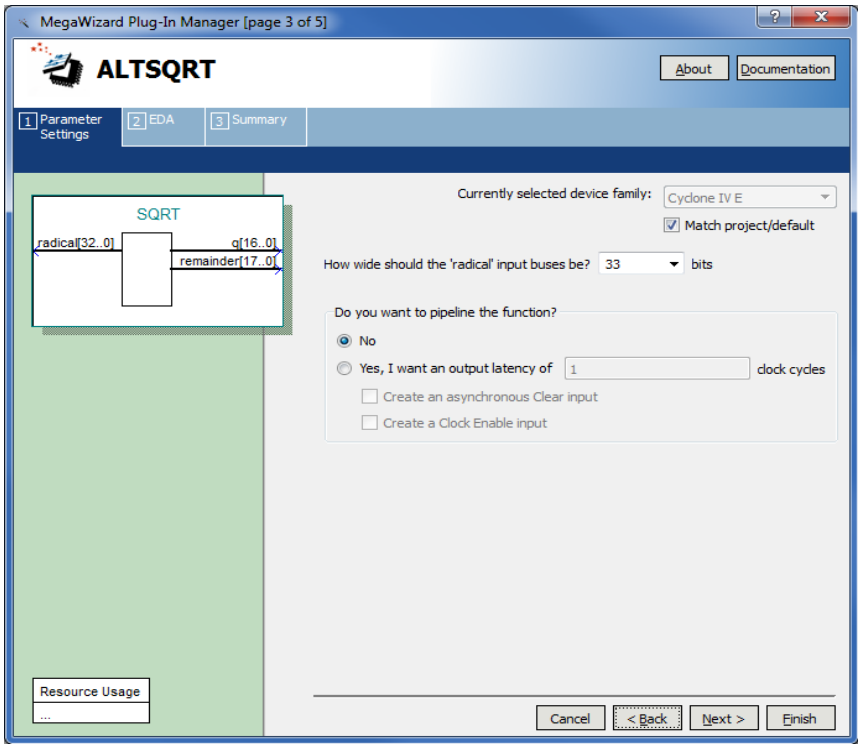


Figure 3-7 ALTSQRT Unit

Table 3-5 below shows the resource usage after synthesis.

Table 3-5 Resource Usage for Single ALTSQRT Unit.

Total combinational functions	571 / 114,480 (< 1 %)
Dedicated logic registers	311 / 114,480 (< 1 %)
Device	EP4CE115F29C7
Family	Cyclone IV E
Total logic elements	587 / 114,480 (< 1 %)
Total registers	311
Embedded Multiplier	9-bit elements 0 / 532 (0 %)

3.6 Summary

An overview of Altera FPGA and its design suite has been presented in this chapter. Then the system components which include Nios II processor, Avalon Switch Fabric and Megafunction have been introduced. The performance and resource usage of these components was presented in detail.

Chapter 4 Power Consumption

Many applications require low-power programmable logic solutions. For this reason many programmable logic vendors have focused on minimizing device power consumption. There are five different power components that must be considered when evaluating different FPGAs. Figure 4-1 shows these components.

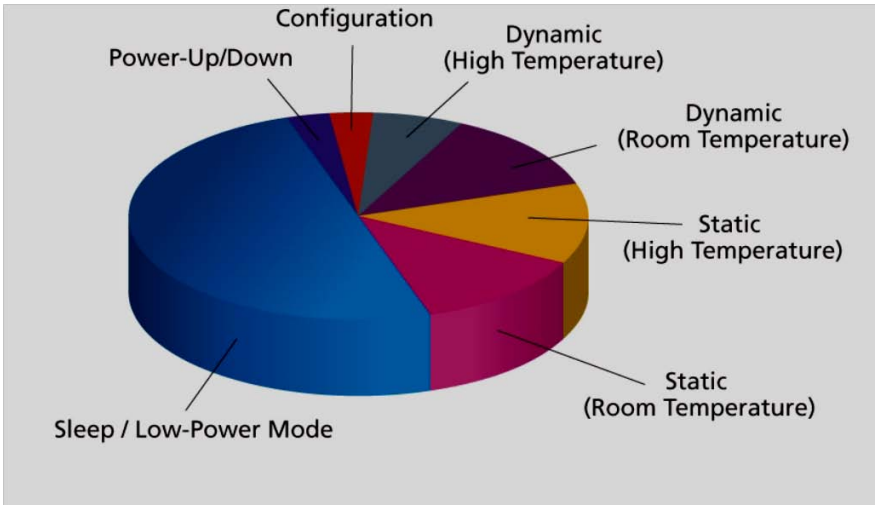


Figure 4-1 Power Consumption Components [54]

The important power components to consider include power up, configuration, dynamic, static and sleep power. Power up is the amount of power drawn by the device during power up. Configuration power refers to the power required during the loading of the FPGA upon power up. Some FPGA devices offer low-power or sleep modes. In some cases, this may be different from static power.

It can be seen that besides the sleep mode, the most power consumption is composed of static and dynamic power. Static power is the power consumed by leakage current. Both digital and analog logic consume static power; static power is primarily composed of the quiescent current of the analog circuit based on its interface configuration. Traditionally, digital logic has not consumed significant static power, but this has changed with the advent of very small process nodes. Leakage current in digital logic is now the primary challenge for FPGAs as process geometries decrease. While the move

to the 60-nm Cyclone IV processor delivers the expected Moore's law benefits of increased density and performance, the performance increases can in turn result in significant increases in power consumption, thereby causing the risk of consuming unacceptable amounts of power [49]. Figure 4-2 shows the source of static leakage current.

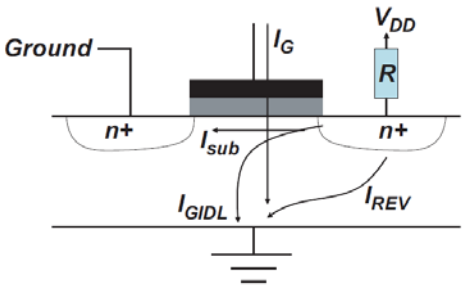


Figure 4-2 Transistor Leakage Diagram [49]

The sources of static leakage current include sub-threshold leakage (I_{SUB}), gate-include drain leakage (I_{GIDL}), gate direct-tunnelling leakage (I_G), and reverse-biased junction leakage current (I_{REV}). The sub-threshold leakage has the dominant impact on static power. It is sensitive to supply voltage, gate threshold voltage, temperature and channel length. The sub-threshold leakage can be reduced by reducing the core voltage, increasing voltage threshold and increasing gate length. Both Gate-induced drain leakage and gate direct-tunneling leakage have a small impact on static power. They are sensitive to gate oxide thickness and supply voltage and can be reduced by using dual gate oxide. The impact by reverse-biased junction leakage current to static power can be negligible [50].

Dynamic power is the amount of power the device consumes when it is actively operating. It is the additional power consumed during the operation caused by signals toggling and capacitive loads charging and discharging. The dynamic power can be calculated via the following equation:

4-1:

$$P_{dynamic} = \left[\frac{1}{2} CV^2 + Q_{short\ Circuit} V\right] f * activity$$

As shown in the equation, the main variables affecting dynamic power are capacitance charging, supply voltage, and clock frequency. The function activity means the percent

of circuit that switched each cycle. Dynamic power decreases with Moore's law by taking advantage of process shrinks to reduce capacitance and voltage. The challenge is that more circuits are implemented with each process shrink, and the maximum clock frequency increases. While the power reduction declines for an equivalent circuit from process node to process node, the FPGA capacity keeps doubling and the maximum clock frequency keeps increasing[50].

4.1 Altera Power Optimize Methods

Altera has taken significant steps to reduce static power in Cyclone IV FPGAs by implementing a low power (LP) process technology traditionally used in semiconductor manufactures for handset components. This has minimized the leakage current for low static power. The smaller geometries made possible by this advanced process, combined with architectural optimizations, enable Cyclone IV FPGAs to keep static and dynamic power consumption to a minimum. The process is enhanced by the use of low-k dielectrics, variable channel lengths and oxide thicknesses, and multiple transistor threshold voltages. By using those technologies, the power consumption has been reduced by up to 25% compared to Cyclone III devices. Figure 4-3 shows the comparison of static power consumption of Cyclone III and Cyclone IV E devices at 85°C junction temperature. The smallest Cyclone IV E FPGA, the EP4CE6 device, consumes as little as 38 mW at 85°C, and the largest Cyclone IV E FPGA, the EP4CE115 device, consumes as little as 163 mW static power at 85°C [51].

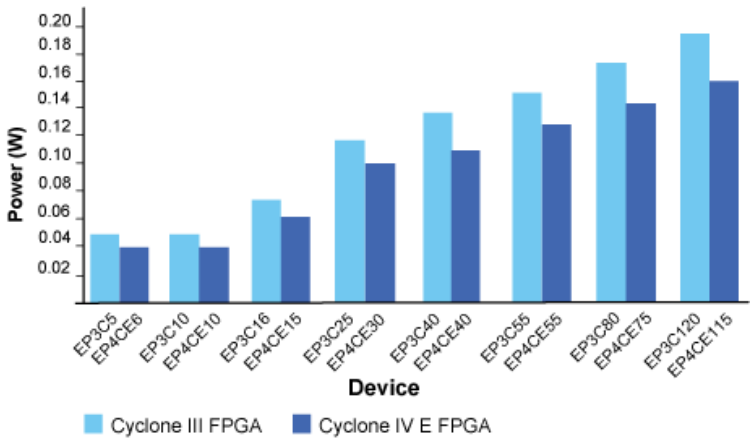


Figure 4-3 Typical Static Power Consumption of Cyclone IV E FPGAs

In the comparative study with similar density devices, Cyclone IV only consumed half of the static power of Xilinx Spartan devices. Figure 4-4 summarizes the result of the static power comparison.

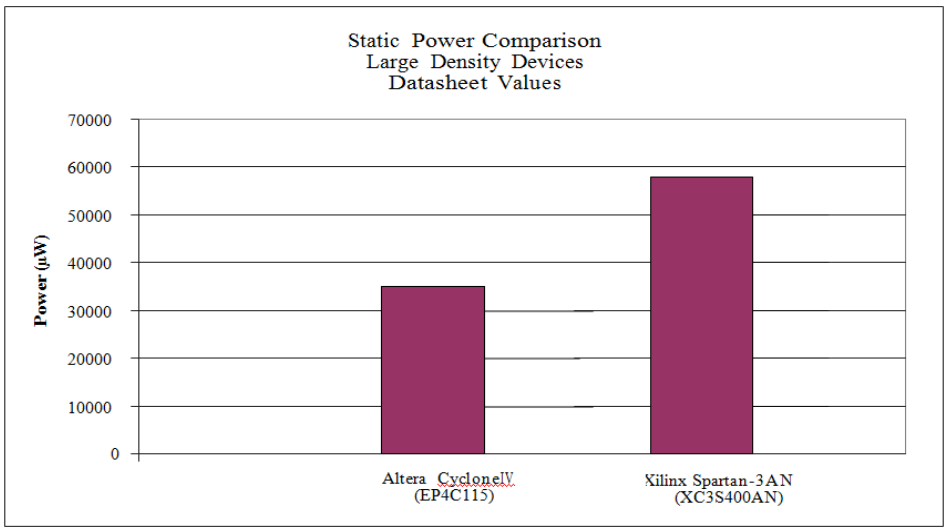


Figure 4-4 Static Power Summary

Altera has also introduced a radical and unprecedented method called "Programmable Power Technology" for reducing power consumption for high-end FPGAs. Traditionally, all high-performance FPGAs are implemented within a high-performance fabric where every logic element (LE) provides the maximum performance with a subsequent high leakage power. Not all the circuits need maximum performance however in real design. Usually only a small amount of circuits are speed critical and the rest has excess slack [52]. Figure 4-5 shows a typical excess slack histogram where the majority of paths have slack and only a few critical paths need the highest performance logic to meet the timing requirement.

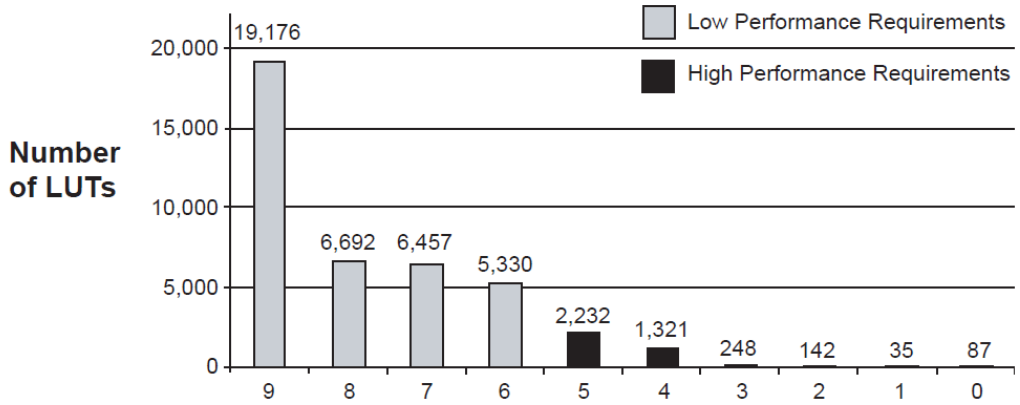


Figure 4-5 Amount of Slack per Unit Delay

Programmable power technology enables the Stratix III logic fabric to be programmed at the logic array block (LAB) level to provide high-speed logic or low-power logic, depending on what is required by the specific logic path [52]. Figure 4-6 illustrates this technology. The longest path is the timing critical path. The LABs in the timing critical path receive a high-speed setting, while the rest of the LABs use the low-power setting. The leakage power of the LABs with a low-power setting is 70% less than that of the LABs with a high-speed setting. In addition, unused logic elements, digital signal processing blocks, and memory blocks are set to low-power mode for further power saving. Programmable Power Technology enables an optimal combination of setting timing critical logic to high-speed mode for achieving the desired system performance, while setting the rest of the logic to low-power mode for minimizing leakage current. In this way the design of Stratix III FPGA consumes as little power as possible.

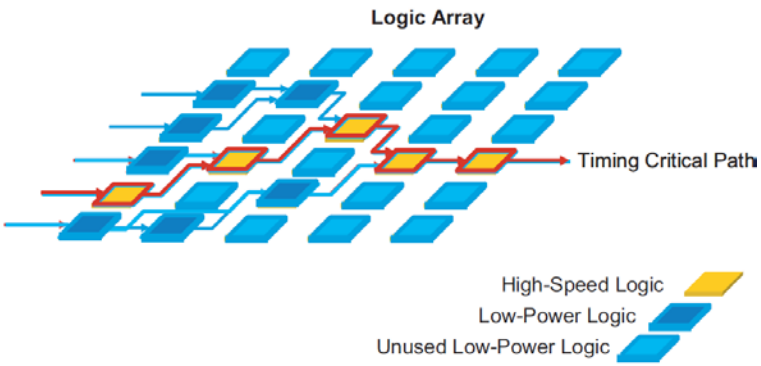


Figure 4-6 Programmable Power Technology

The core voltage of Stratix III FPGA can be set to 0.9V or 1.1V depending on the performance requirement. The 0.9V core voltage provides lower dynamic and leakage power than 1.1V core voltage, while the 1.1V core voltage delivers the highest overall performance. Dynamic power scales with the square of core voltage, while static power scales by the power of 2.5 of core voltage - as shown in Table 4-1.

Table 4-1 Power Compared with Selectable Core Voltage

Power Component	Stratix III		Stratix II
	0.9V	1.1V	1.2V
Relative Static Power	0.36	0.48	1
Relative Dynamic Power	0.45	0.67	1

It can be seen that the 1.1V core consumes 33% less dynamic power and 52% less static power than 1.2V voltage core, while 0.9V voltage consumes 55% less dynamic power

and 64% less static power. This core voltage supplies all the LABs, memories, and DSP functions in the core fabric. It is important to select an appropriate core voltage because the corresponding timing and power model will be used in timing-dependent and power-dependent analysis and optimization. When selecting the core voltage, the designer needs to decide on a suitable voltage based on the timing analysis. If the system performance requirements can be achieved by using 0.9V core voltage, 0.9V core voltage should be selected because less power will be consumed.

Competitively, by implementing these advantage technologies Stratix III FPGAs provide an average of 29 percent (at 1.1V) and 45 percent (at 0.9V) lower total power than the nearest competing FPGA. Figure 4-7 shows the total power advantage seen in Stratix III FPGAs over Virtex-5.

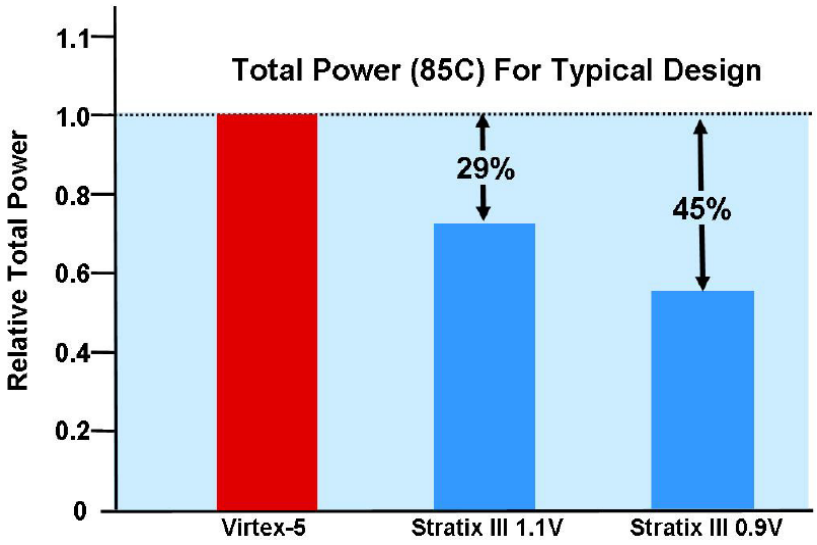


Figure 4-7 Power Comparison between Stratix III FPGAs and Virtex-5

4.2 Power Consumption of Multi-Core System

Altera provides two kinds of power measurement features; PowerPlay early power estimator and PowerPlay power analyzer. The PowerPlay early power estimator (EPE) is a spreadsheet-based analysis tool that enables early power scoping based on device and package selection, operating conditions, and device utilization. The other option, PowerPlay power analyzer, is a far more detailed power analysis tool that uses actual design placement, and routing and logic configuration. This tool can use simulated

waveforms to very accurately estimate dynamic power. The power analyzer, in aggregate, usually provides ± 10% accuracy when used with accurate design information. The PowerPlay power models closely correlate to actual silicon measurements.

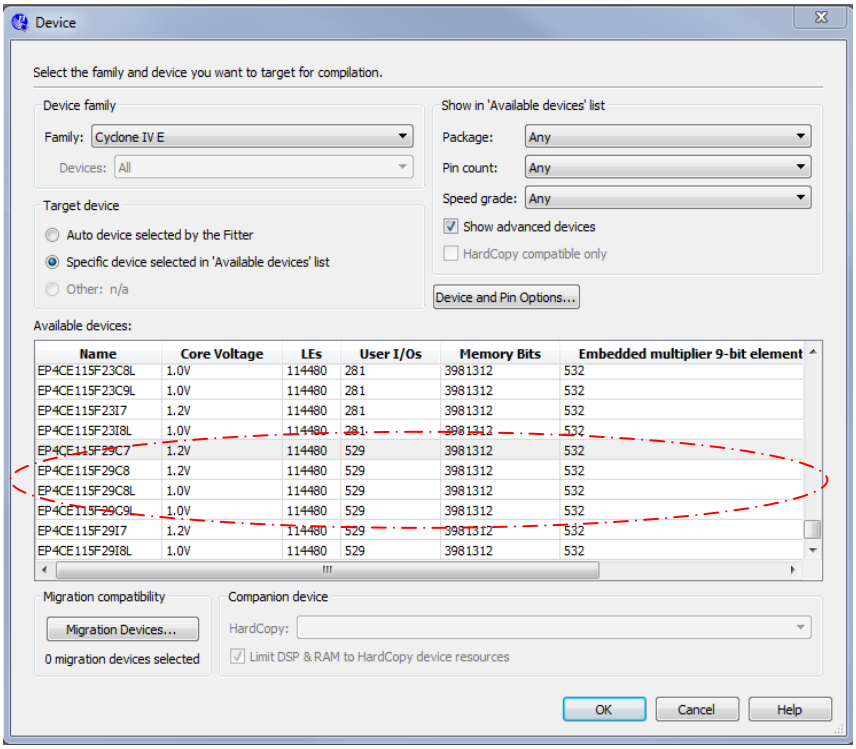


Figure 4-8 Selectable Core Voltage of Cyclone IV Devices

The DE2-115 development board has an EP4CE115F29C7 device on board. The speed grade of the device is "7" which is relatively fast in the EP4CE115 series. The goal was to examine the power consumption of the design when combined with these power saving technologies. The voltage option in the Quartus II software under the operating setting and condition category can be set to 1.2V or 1.0V. The speed grade of the 1.0V device however, is slower than that of 1.2V. In Figure 4-8, it can be seen that there are two kinds of selection for speed grade and voltage. The EP4CE115F29C7 and EP4CE115F29C8 have the same core voltage (1.2V) but a different speed grade, as do the devices EP4CE115F29C8L and EP4CE115F29C9L (1.0V).

The PowerPlay Power Optimization option in the Fitter Setting dialog box controls the configuration of tiles in the high-speed or low-power mode. The Quartus II software automatically determines which tiles operate in high-speed mode and which operate in

low-power mode, based on the timing constraint specified for the design. Realistic timing constraints must be provided for the design to achieve the lowest possible power consumption. There are three PowerPlay Power Optimization options: Off, Normal compilation and Extra effort. Figure 4-9 shows the fitter setting selection.

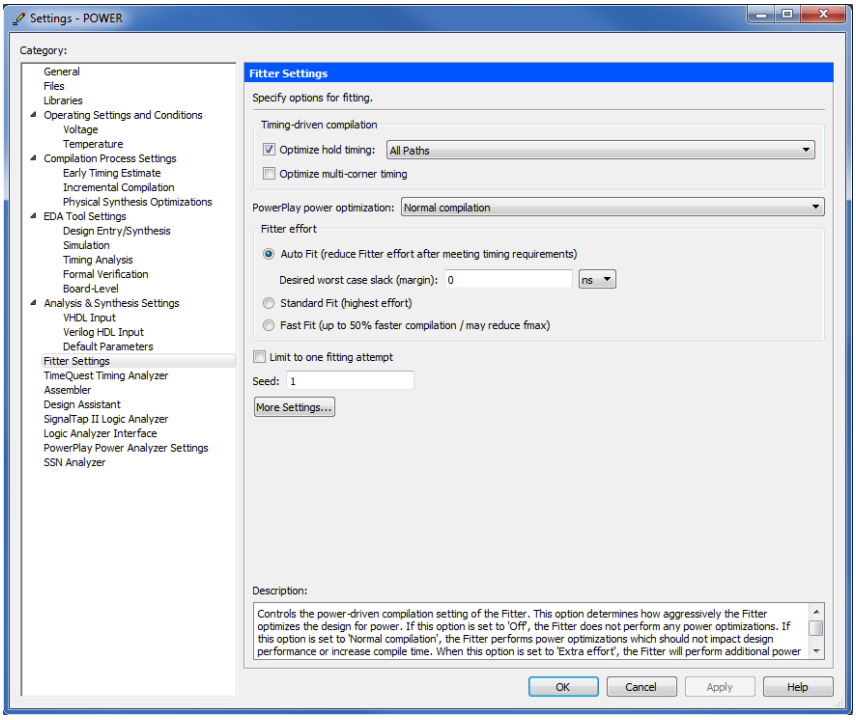


Figure 4-9 Fitter Setting

No netlist, placement or routing optimizations are performed to minimize power when Off is chosen. The Normal compilation is the default setting. In normal compilation, tiles are configured in high-speed or low-speed mode depending on the timing constraints of the design. The rest tiles are configured in low-power mode to reduce the overall power consumption. This level of power optimization does not have any effect on the fitting, timing results, or compilation time.

The Extra effort setting performs the functions of the Normal compilation setting, as well as extra place and route optimizations during fitting to fully optimize the design for power. Extra effort is applied by the fitter to minimize power consumption even after timing requirements have been met. The Extra effort setting uses a Value Change Dump File (.vcd) that guides the fitter to fully optimize the design for power, based on the signal activity of the design. Signal activities from full post-fit netlist (timing)

simulation provide the highest accuracy as all node activities reflect the actual design. The extra effort includes moving the logic closer during placement to localize high-toggling nets, using routes with low capacitance, and trying to configure more high-speed mode tiles to low-power mode tiles when possible. Note however this option will increase the compilation time.

The power consumption of the multi-core systems was measured by the EPE tool as shown in Figure 4-10. This tool requires the import of the .pof file which is generated after compilation. The supply voltage was set to 1.2V with the PowerPlay Power Optimization option set to Normal Compilation. The multi-core system with 2, 4, 6 and 8 cores were complied respectively with the test undertaken at room temperature. The measured power consumption of FPGA is summarized in Table 4-2.

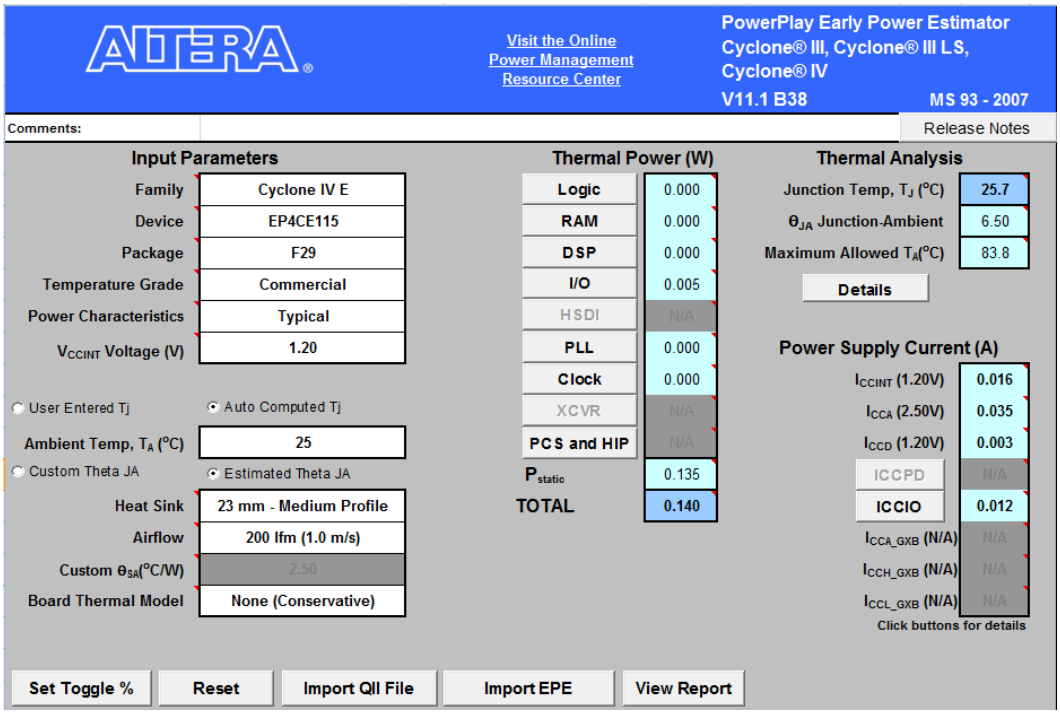


Figure 4-10 PowerPlay Early Power Estimator

The results in Table 4-2 are plotted in Figure 4-11. The power should track linearly with frequency and percentage of resources usage. One can see that the power consumption tracks in an approximately linear fashion with the increase in the number of cores. This is because the resource utilization is directly proportional to the number of cores. The difference between the total power and the static power is the I/O thermal power.

Table 4-2 Power consumption with Multi-Core System

Number of Cores	Static Power(mW)	Total Power(mW)
1	98.9	136.75
2	99.36	144.8
4	100.25	159.7
8	436.94	484.05

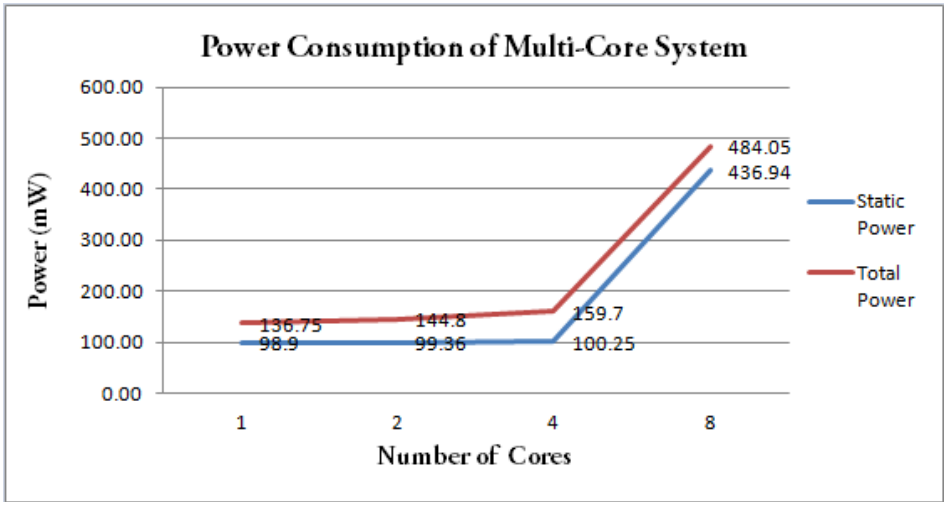


Figure 4-11 Power consumption with Multi-Core System

4.3 Summary

Leading-edge technology is continuously developed in order to maximize the performance and minimize the power consumption in FPGA devices. The Programmable Power Technology selectable core voltage enables the lowest possible power for Altera's FPGA. Compare with computer based transient detection, it difficult to say how much energy was saving by using FPGA. That because the computer processing speed is relatively fast (usually in GHz), while FPGA's processing frequency is still in develop. Obviously, if they are running at same frequency, same hardware resources, FPGA will greatly reduce the power consumption as it doesn't need to waste power on software implementation. In a real application, as the transient signal detection usually needs to run 24 hours a day and 7 days a week, the energy saved by using FPGA is significant.

Chapter 5 Methodology and Design Flow

This chapter explains the overall project design flow. Design principles of HDL are described in detail with the three primary physical characteristics of a digital design and its implementation in each module discussed. Methods for architectural optimization in FPGA are also discussed in this chapter.

In order to properly design the system architecture, the following project roadmap was established.

1. Understanding and feasibility study of the entire system operation and the delineation of each functional module.

2. Design of each module with Verilog. Measure performance of each module and optimize design architecture.

3. Verilog test bench design, simulate each module and observe the results from ModelSim waveform.

5.1 FPGA Design Principles

There are three important characteristics in digital design: “speed,” “space,” and “power.” Speed refers to the highest stable frequency that a design module can achieve. There are three primary definitions of speed: throughput, latency and timing. In the context of processing data in FPGA, throughput refers to the amount of data that is processed per clock cycle, and latency refers to the time between data input and processed data output. The typical metric for latency is time or clock cycles. Timing refers to the logic delays between sequential elements. A design that does not meet timing delays the critical path; that is, the largest delay between flip-flops (composed of combinatorial delay, clock-to-out delay, routing delay, setup timing and so on) is greater than the target clock period.

In FPGA design, the following criteria should be followed:

- High-throughput architectures for maximizing the number of bits per second that can be processed by the design.
- Low-Latency architectures for minimizing the delay from the input of a module to the output.
- Timing optimizations to reduce the combinatorial delay of the critical path.
- Adding register layers to divide combinatorial logic structures.
- Parallel structures for separating sequentially executed operations into parallel operations.

Space refers to the usage of resources in module design, normally counted by flip-flops and LUT. Selecting the correct topology in design can reduce resource usages. Here, topology refers to the high level organization of the design. Circuit-level reduction performed by the synthesis and layout tools refers to the minimization of the number of gates in a subset of the design and may be device specific. A topology that targets area is one that reuses the logic resources to the greatest extent possible, often at the expense of speed. In most cases, it requires a recursive data flow where the output of one stage is fed back to the input for the same processing. This can be a simple loop that flows naturally with the algorithm, or it may be that the logic reuse is complex and requires special controls.

As a design requires high frequency, and a small usage of space is unrealistic, speed and space become two conflicting aspects of FPGA design. The proper design approach thus requires meeting the timing requirement while minimizing space. If the design has a relatively large timing allowance and operates at high frequency, it means the design is more robust. On the other hand, the less space consumed by the design means the more function modules per unit space while reducing the cost of the design [44]. Comparing both aspects of a design, the timing requirement is more important. When they conflict, speed is a much higher priority. There are a variety of methods that can achieve the conversion between speed and space, such as module reuse, Ping-pong operation and Pipe-line operation. The following chapter discusses these methods in greater detail.

The other important physical characteristic of digital design is power. Compared with ASIC (application specific integrated circuit) design, FPGA is costly, and typically not well suited to ultralow-power design. Some FPGA vendors offer low power CPLDs (complex programmable logic device), but these are very limited in size and capability

and thus will not always fit an application that requires a respectable amount of computing power. In CMOS technology, dynamic power consumption is related to charging and discharging capacitances on gate and metal traces [43]. The general equation for current dissipation in a capacitor is:

5-1:

$$I = V \times C \times f$$

Where "I" is total current, "V" stands for voltage, "C" stands for capacitance, and "f" for frequency. In order to reduce the current, we must reduce one of the three parameters. The capacitance C is related to the number of gates, while frequency f is directly related to the clock frequency. As the voltage is usually fixed in FPGA design, the power-reduction should aim at reducing one of these two factors.

5.2 FPGA Design Operations

5.2.1 Ping-Pong Operation

Ping-Pong operation is widely used for data flow control. Figure 5-1 shows the flow chart of Ping-Pong operation. Input data pass through an input multiplexer and then into a data buffer storage module. Here, Dual-port RAM is usually used as a buffer. This system can have multiple buffer modules. In the storage cycle, data will pass into the first buffer module until it has filled up. The output multiplexer will then switch to the second buffer module. Meanwhile, the computational module will access the data in the first buffer through a second multiplexer. The same procedure applies for the second buffer.

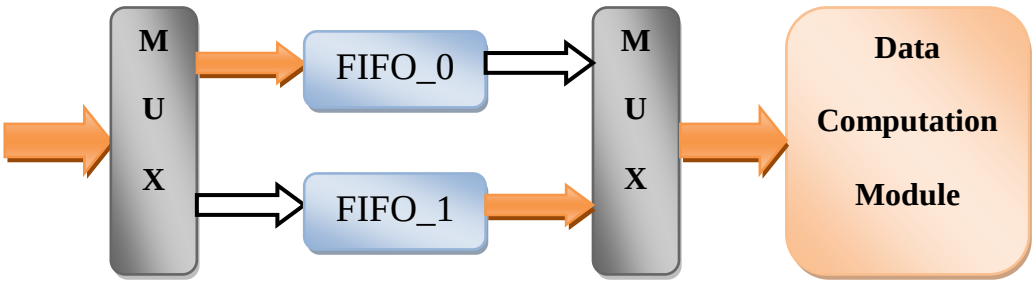


Figure 5-1 Ping-Pong Flow Chart

The most important feature of this method is the seamless connection achieved for data flow through the input and output multiplexers as they switch between each other. Another feature is the saving of buffer space. It can store the first part of an entire data flow into one buffer and process the data from another buffer at same time. It also processes high-speed data flow through a low-speed module.

The difficulty in implementing this method is balancing the data entry speed and processing time. With the ideal design, when the first buffer is filled up, the data process in the second buffer is complete. Another challenge with this method is the buffer design. The buffer setting needs to focus on size and full/empty flags to avoid data overflow.

5.2.2 Serial to Parallel Conversion

Serial to Parallel conversion is one of the important techniques in processing high-speed data. It generally increases the throughput by copying the program logic. There are various methods that can achieve serial/parallel conversion. For some relatively small designs, shift register can be used to complete the conversion. Figure 5-2 illustrates the framework of serial/parallel conversion, showing serial data input to FPGA's internal shift register, and output of an N-bit wide parallel data.

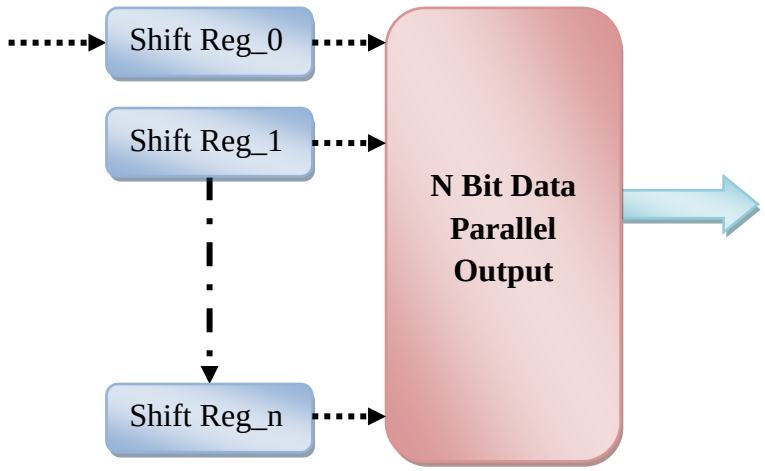


Figure 5-2 Serial to Parallel Conversion

Generally this operation needs to be clock synchronized. This means the data sampled during several clock cycles needs to wait another equivalent time period to obtain

parallel output. As mentioned in 5.1, serial to parallel conversion is an implementation that uses space to exchange speed. In FPGA design most of the statements are executed in parallel - this is the significant difference when compared with C or other languages. Handling the conversion between serial to parallel well is one of the key features of FPGA design.

5.2.3 Pipeline Operation

Pipeline here refers to an operation that processes data flow. A pipeline is a systolic array where all data flow goes in one direction and there is no feedback. Figure 5-3 illustrates the structure of pipeline operation. Pipeline operation can improve system frequency and is commonly used in high speed signal processes.

Figure 5-3 Pipeline Operation

The basic structure of pipeline operation involves the appropriate connection of one-way steps. The output of the previous step becomes the input for the next step. A typical pipeline operation divides the original combinational logic module into several parts. This means it takes more time to complete the new modules which have the same function of the previous one. However, the operating frequency of these new modules is significantly improved, especially in cases where the pipeline operation upgrades the performance of the design. For example, assume that the previous design needs three steps to complete one group of data. That means it takes three clock cycles from the data input to obtain the result. By using pipeline operation, only the first result will take three clock cycles to complete with the remaining results obtained from each clock cycle after that. The outcome is an increase in speed of nearly three times.

The key point for pipeline operation design is to arrange reasonable timing sequences for each step. To avoid data overflow, the size of data flow for each step also needs to be considered. If the operation time for the first step equals that of the second, the output of the first step can pass directly to the secondary step. If the operation time for the first step is less than the second, it will need a buffer to store the output data from the first step before passing it to the next step. The third issue in arranging timing

sequences for pipeline operation is the problem of operation time for the first step being greater than the second. This requires the use of methods such as module reuse or serial to parallel conversion to separate data flow, otherwise it will cause a mismatch in data processing

5.3 Data acquisition

High speed data acquisition systems are generally divided into analog signal conditioning circuits, analog to digital signal conversion circuits, large capacity memory trigger circuits, and system timing and control logic units, etc. Generally data acquisition systems work based on the idea of fast write and slow read, which samples data by high-speed ADC and stores it to large cache (such as SDRAM), before uploading to a data processing unit. The data quantity of this form of sampling is determined by the cache. At present the larger capacity cache are mainly SDRAM, which generally store dozens of MB. SDRAM is a type of storage that needs frequent refreshing and requires the clock to synchronize with the read and write operation. As the sample frequency is faster, the data acquisition system, which is based on SDRAM, can only collect a little data at one time, thereby restricting the quantity of data sampled. On the control side, SDRAM needs to be refreshed every few nanoseconds in order to maintain the stability of the data.

SRAM is another cache option. A storage device that does not require periodic refreshing, SRAM uses bi-stable latching circuitry to store data. Unlike SDRAM, the storage addresses in SRAM are independent, and since SRAM does not need refreshing, the access speed is much faster. Normally, it takes 60~75 ns to access one address for SDRAM, while it only takes 10 ns for SRAM. The power consumption of SRAM is also more stable than SDRAM.

Regardless of SRAM's advantages over SDRAM, neither SDRAM nor SRAM are the best choice, as explained detail in Chapter 5.4. Accordingly, for this project design, a high speed, real time data acquisition system based on on-chip memory was chosen. The on-chip memory is the logic resources that commonly integrated with the processor. It often used for data cache and instruction storage, serving an interface between the processor and the off-chip memory. The on-chip memory is controlled through the

FPGA with the idea of space-for-speed allowing for high speed, real-time data upload. Figure 5-4 shows the structure of SRAM operation.

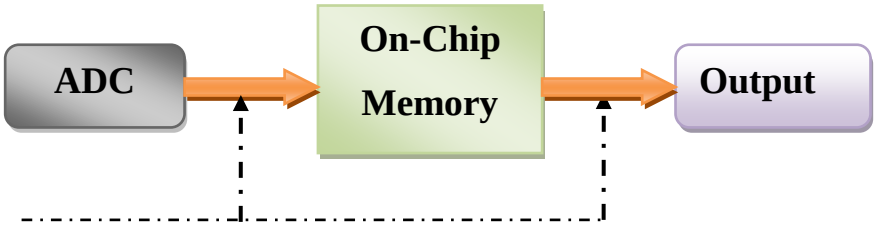


Figure 5-4 Dual SRAM Operation

In this project, the ADC converter connects a large scale of antenna array to the sampling data. Due to the limitation of the testing environment, random generated data was used for the entire program testing.

5.4 FIFO Module Design

The FIFO module is related to the reliability and stability of the entire design. Accessing the FIFO must be managed by this module to ensure that there is no confliction or overwrite operation. For this project, on-chip memory block is used as a FIFO to store the temporary data from the ADC device.

On a DE2-115 development board, a Cyclone IV processor supports some M9K on-chip memory blocks that are used to store the data from the ADC device. These M9K memory blocks have the following features: 8,192 memory bits per block, variable port configurations (8192 x 1, 1024 x 8, 512 x 16, etc), independent read/write enable signals, and two clock enable control signals for each port. Single port and dual port modes are supported for all blocks. The DE2-115 development board also provides other off-chip memories such as SRAM, SDRAM. They are all able to be used as cache.

When compared with on-chip memory, off-chip memory has some shortcomings. Firstly, on-chip memory is single cycle access, with the access for each address fixed. Off-chip memory has an unstable access time, in some cases it will take two or more clock cycles to access data that will affect optimization for the compiler. For example, the SDRAM has an average latency of approximately 20 clock cycles. However, on-chip memory can be configured to have one clock cycle read latency and zero cycle write latency. This feature is particularly suitable for high-speed data access. In some

cases, large latency of access off-chip memory will slow down a CPU's pipeline operation. Secondly, for most applications, the use of on-chip memory reduces power consumption by 40% when compared with off-chip memory [45]. By comparison, the disadvantages of on-chip memory are its size and the number of memory block limitations.

This project provides two kinds of FIFO design modules, in order to adapt to the ADC device when working on different clock frequencies. They are respectively synchronous and asynchronous FIFO. Further details on FIFO are provided in the following chapter.

5.4.1 Synchronous FIFO Design

Synchronous FIFO refers to a read and write operation under the same clock frequency. This means that their addresses are synchronized. In program design, a counter (*cnt*) is usually used to count the number of existing data that cannot be read out. In read only status the counter will subtract one, while in write only status it will add one. The counter will remain the same in the case of reading and writing at the same time, or neither reading or writing.

To ensure the FIFO does not overflow in operation, some flags are used in design. Write enable (*wr_en*) and read enable (*rd_en*) flags indicate the valid operation for writing and reading. Full (*full*) and empty (*empty*) flags are used to protect FIFO from being overwritten and overread.

In program design, internal read and write enable flags are a self-protection mechanism. Read enable flags are a combination of an external read enable interface and a non-empty flag (*Figure 5-5*). Similarly, the write enable flag is composed of an external write enable interface and non-full flag.

```
assign read_enable = read_enable && ! empty;
assign write_enable = write_enable && ! full;
```

Figure 5-5 Read/Write Enable Flag

In initial status, full flag should be setup at "1," which indicates the FIFO is full. If there is no reading operation and the counter (*cnt*) equals the depth of the FIFO (*depth_fifo*), or the counter is equal to "*depth_fifo – 1*," the write enable is still valid and a full flag

should be setup at "1," otherwise it should be kept at "0." For empty flags (*empty*), at initial status it should be setup at "1," to indicate the FIFO is empty. If the counter (*cnt*) equals "0" or if it equals "1" while read enable is valid, the empty flag should be setup at "1," otherwise it should be kept at "0."

In this design, FIFO is operating at high frequency status. To avoid an unpredictable address pointer overflow, almost empty (almost_empty) and almost full (almost_full) flags are used to give early notice for FIFO status. When these flags are triggered, it means the counter is nearly empty or full but still allowed to read/write a small amount of data.

Address control is relatively simple. When at read enable status, the read address will automatically carry one step in each clock cycle; the same as the write address operation. However, from the perspective of logic gate design, the simple "add" operation is complicated as it involves the "carry" and "flip" operations on the counter and these operations easily generate glitches in a high-speed circuit. When the FIFO is working at high frequency status, linear feedback shift register (LFSR) can be adopted. This approach uses "shift" and "XOR" operations to control FIFO's address and is simple and fast.

Linear feedback shift register (LFSR) is a kind of encryption circuit and its widely used in communication coding field. It can be presented by following equation:

5-2:

$$G(x) = g_m x^m + g_{m-1} x^{m-1} + \ldots + g_1 x + g_0$$

This polynomial is a primitive polynomial. For example, $m = 3$ and initial status $G_2G_1G_0 = 001$ as the circuit flow chart shows below.

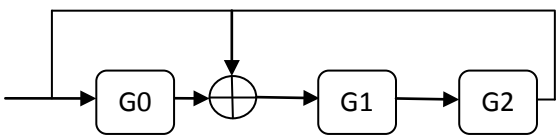


Figure 5-6 Linear Feedback Shift Register

In every clock cycle the output will generate by this circuit. Table 5-1 shows the result. The character of this circuit is cycling back to start point after seven clock cycles.

Table 5-1 Truth Table of LFSR

Status	Code	Results
0	$G_2G_1G_0$	001
1	$G_2G_1G_0$	010
2	$G_2G_1G_0$	100
3	$G_2G_1G_0$	011
4	$G_2G_1G_0$	110
5	$G_2G_1G_0$	111
6	$G_2G_1G_0$	101
7	$G_2G_1G_0$	001

LFSR can be optimized into LUT (Look up Table) in digital design. In the context of real circuit application this is another element employed to improve the speed. Figure 5-7 shows the Verilog implementation of the LFSR program.

```
/* Generation of read address pointer */
wire read_liearfeedback
assign read_liearfeedback = !(read_addr[8] ^ read_addr[4]);
always @ (posedge clk or negedge rst_n)
        if(rst_n) read_addr <= 'b0;
        else if (read_enable)
        read_addr <= { read_addr[7], read_addr[6], read_addr[5],
                       read_addr[4], read_addr[3], read_addr[2],
                       read_addr[1], read_addr[0], read_liearfeedback };
```

Figure 5-7 Verilog Implementation of LFSR

5.4.2 Asynchronous FIFO Design

Asynchronous FIFO means read and write operation under different clock frequency or in same frequency but different clock phase. It needs to compare the read and write address when generate full/empty flags. However, it cannot compare them directly. That because they are working under different clock domains. Another reason is the physical character of address register. Address register usually contains many bits, in case of "carry" situation it cannot guarantee that all these bits are flip in same time. It needs

time to stabilise. If read clock cycle sampling the write address during stable time, it will make mistake.

To avoid this defect, this module implements Gray code in design. Gray code is a form of binary that uses a different method of incrementing from one number to the next. With Gray code, only one bit changes state from one position to another. Gray code is the most popular absolute encoder output type because its use prevents certain data errors which can occur with natural binary during state changes. In program design, Gray code and binary code can be converted to each other by "XOR" operation. Table 5-2 shows the comparison between binary code and Gray code.

Table 5-2 Truth Table of Gray Code

Number	Bin	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Gray code cannot generate full/empty signals by a simple "add" operation. To determine the FIFO status, the read address (read_addr) should first convert to gray code (read_gray). Then the write clock domain is used to synchronize this read address (rag_wt_syn) and convert this address to binary code. Finally, the current write address (write_addr_pl) is allowed to delay for one clock cycle, with the difference between read address (rag_wt_syn) and write address (write_addr_pl) the status of this FIFO. The aim of delaying the write address one clock cycle is to ensure it synchronizes with the read address, because the read address takes one clock cycle to convert to gray code.

An empty flag is set on two conditions (*Figure 5-8*):

1. Read gray code is equal to write gray code (*write_gray = read_gray*).

2. Only one address is left inside FIFO and read enable is valid (write_gray =

read_next_gray)

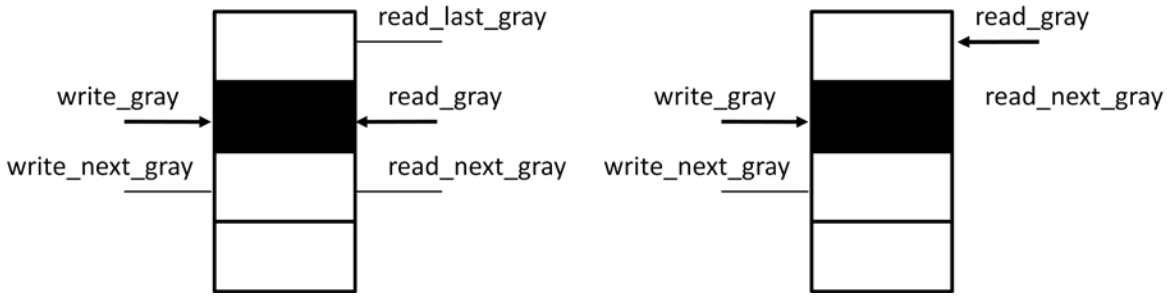


Figure 5-8 Empty Flag Generation

A full flag is set on two conditions (*Figure 5-9*):

1. Write gray code is equal to the last read gray code (*write_gray = read_last_gray*).

2.The next write gray code is equal to the last read gray code and write enable is valid

(write_next_gray = read_last_gray)

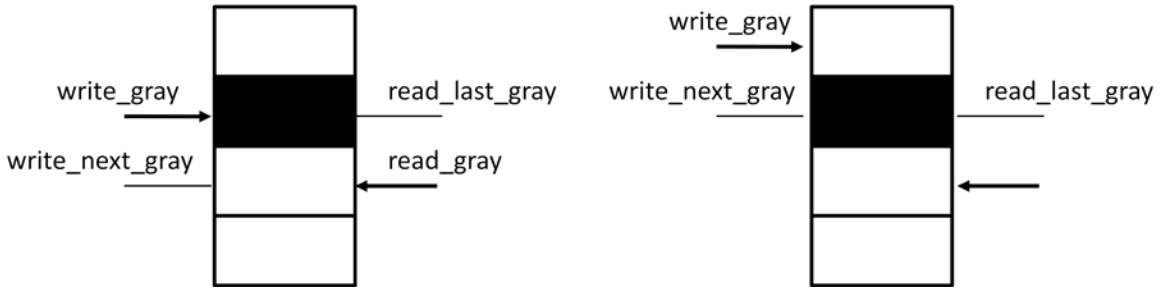


Figure 5-9 Full Flag Generation

5.5 SP_MEAN Module Design

As mentioned in section 2.2, the CUSUM algorithm is designed for sequential data analysis. However, sampling data from an ADC device gives an infinite continuous signal. To implement a CUSUM algorithm, the sampling signal should be divided into equal length intervals. In this program design, each interval was set up to contain 32 samplings. The main function of this module was to parallelize the serial input data from the FIFO module and calculate the average of these samplings.

There are 32 samplings of 16-bit registers used in the module, with input data stored in these registers with each clock cycle. Meanwhile, the cumulative operation is carried out simultaneously. A 5-bit counter is used to control the amount of input data. Once completed, the entire data of this interval, plus its mean value, is output in parallel. Figure 5-10 shows the block diagram of SP_MEAN module.

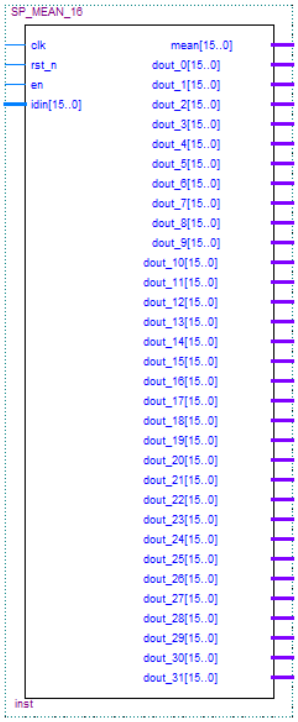


Figure 5-10 Block Diagram of SP_MEAN Module

This module is a typical application of serial to parallel conversion. Here, speed is the primary priority, so it consumes more register resources. There is no latency of each interval output.

5.6 Standard Deviation Module Design

The role of this module is the calculation of the SD as a threshold for a CUSUM algorithm. Standard deviation is calculated for a finite length of sequential data. In this case, the number of data was determined by a previous module (SP_MEAN module).

As the amount of data has been determined in a previous module, formula 2.3 can be expanded as:

5-3:

$$S_n = \sqrt{\frac{(X_0 - \overline{X})^2 + (X_1 - \overline{X})^2 + (X_2 - \overline{X})^2 + \cdots + (X_{31} - \overline{X})^2}{N}}$$

This equation can be split into two parts: cumulative operation and square root operation. Cumulative operation is completed here by parameterized ALTMUTI_ADD megafunction. This function has four groups of "MUTI" port and each port contains two groups of input (*figure 3-5*). Each input can be expressed as the difference between the sampling data and the mean value. A square calculation can be done by implementing the same input for each "MUTI" port. *Figure 5-11* shows one group of parameterized ALTMUTI_ADD functions. The new function is named "MUTI_ADD_BASE" module.

```
MUTI_ADD_BASE  MUTI_ADD_BASE_0
        (
                .clock0(clk),
                .dataa_0(idin_0 - mean), .dataa_1(idin_1 - mean),
                .dataa_2(idin_2 - mean), .dataa_3(idin_3 - mean),
                .datab_0(idin_0 - mean), .datab_1(idin_1 - mean),
                .datab_2(idin_2 - mean), .datab_3(idin_3 - mean),
                .result(MUTI_ADDR_OUT_0)
        );
```

Figure 5-11 MUTI_ADD_BASE module

Each "MUTI_ADD_BASE" module function can complete four groups of input data. This requires module reuse (*Figure 5-12*) or module parallel processing (*Figure 5-13*) in order to finish thirty-two numbers. Module reuse means the MUTI_ADD_BASE" module is repeated eight times to complete the calculation. A counter should be added to control the number of running times.

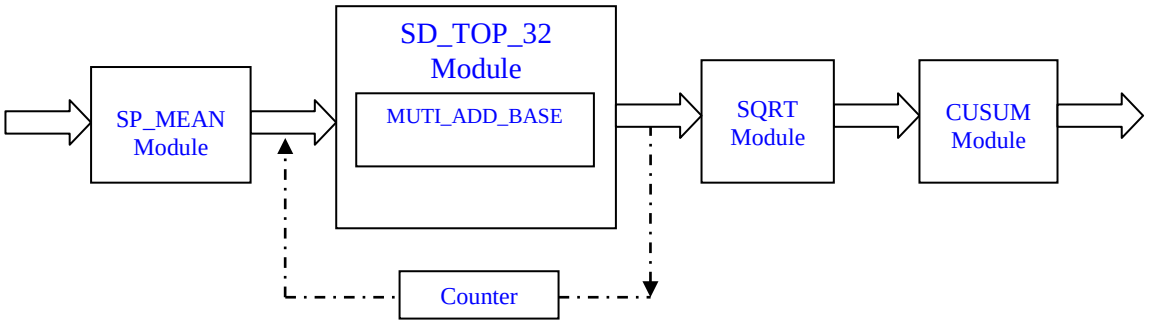


Figure 5-12 SD_TOP_32 Module Reuse

Module parallel processing refers to the parallelized connection of eight MUTI_ADD_BASE modules to one another. This approach reduces the calculation time by consuming more resources and is a kind of implementation of the space for speed method.

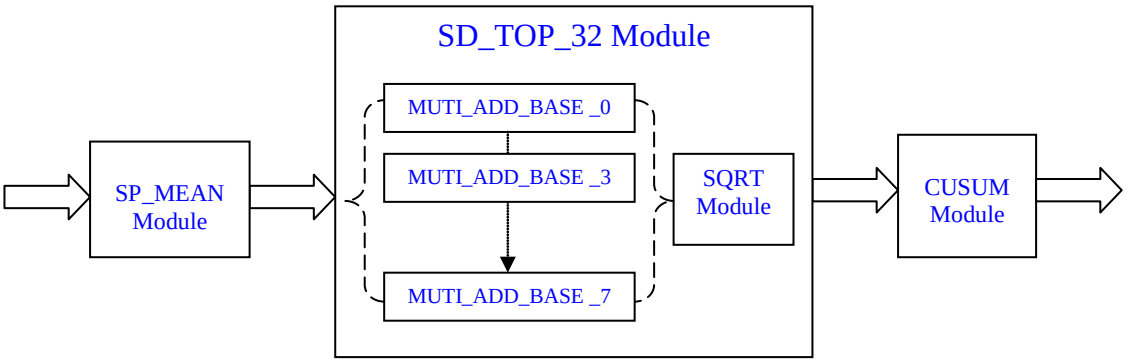


Figure 5-13 SD_TOP_32 Module Parallel Processing

However, one needs to analyse the timing in order to determine which approach is more suitable. It can be seen that it takes 32 clock cycles to get the output from the SP_MEAN module. In order to maintain the pipeline works well, the running time for the SD_TOP_32 module must be equal to or less than 32 clock cycles. According to the timing analysis in the following chapter, the running time for the SD_TOP_32 module is 8 clock cycles. Clearly, the module reuse method is not suitable here because it takes 65 clock cycles. Comparably, while the module parallel processing consumes more resources it meets the timing requirement. In this design therefore, the calculation was done with eight parallelized SD_TOP_32 modules.

The output from each MUTI_ADD_BASE module is added up and divided by the total numbers. It is then sent to the SQRT module to get the SD. Pipeline design is also implemented here to minimize the operation of the clock cycle. Figure 5-14 illustrates the structure of the pipeline adder in the design.

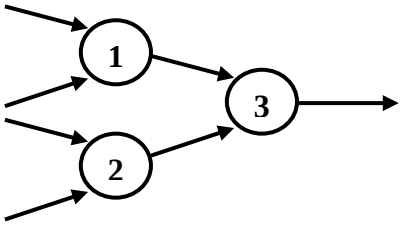


Figure 5-14 Structure of Pipeline Adder

It only takes three clock cycles to add eight outputs together. Division calculation is achieved via a shift operation. In binary systems, shifting one bit to the left means it becomes divided by two. In this case, a division of thirty-two can be achieved through shifting five bits left. In program design, the shifting operation is much faster than algebraic calculation.

After the above operations, the width of the output data is thirty-three bits. One should note that the width of the output must match the input of the SQRT module; otherwise this module will not give a correct result.

5.7 CUSUM Module Design

The aim of the CUSUM module is to detect abnormal signals in each sampling group. In the system it is used to cumulate the difference between sampled signals with their corresponding mean value. If the cumulative is more than twice the SD this means the sampled signal is an abnormal signal. The sampled signal is then output for further processing. Otherwise, the output port will be pull up to high resistance status.

As the sampling signal is a sign-based signal, the cumulative value may appear as a negative number. Therefore this module should have the ability to detect either positive or negative abnormal signals. However, Verilog is a hardware description language that does not support negative numbers directly, as it cannot be converted to digital circuits. In computer systems there is a method called “2’s complement” which can be implemented here [46]. The 2’s complement is a method that encodes negative numbers into ordinary binary. This method generates a negative number by inverting each bit of the positive number, then adding one. For example: 011(+3) → 100 + 1 = 101(-3). This method features the use of the highest bit of binary code as its sign bit, where “1” stands for negative and “0” stands for positive.

In this module, the threshold (2xSD) needs to be converted to negative. By implementing the 2’s complement method, the code can be written as “assign n_sd_out = ~(sd_out - 1'b1);” which means the negative threshold is equal to the complement of the positive threshold minus one. Figure 5-15 illustrates the threshold in coordinate system.

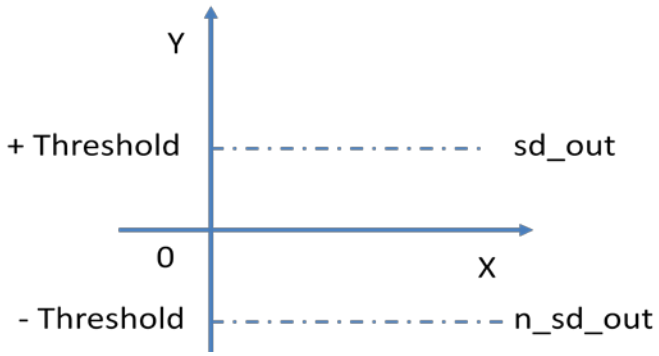


Figure 5-15 Thresholds in coordinate system

5.8 SRAM Module Design

The DE2-115 board has 2MB SRAM memory with 16-bit data width. Featuring a maximum performance frequency of about 125MHz under the conditions of standard 3.3V single power supply makes it suitable for dealing with high-speed processing applications that need ultra data throughput. The related schematic is shown in Figure 5-16.

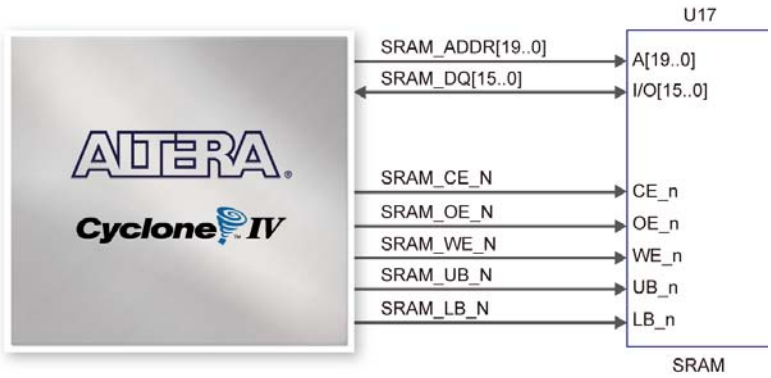


Figure 5-16 Connections between FPGA and SRAM

The clock frequency of 50 MHZ is synchronous with the Cyclone IV device. The SRAM data bus is a bi-directional interface that allows writing in or reading out. For the writing process, data can be directly assigned to the SRAM data bus. For reading, it needs to pull this data bus to high resistance status; otherwise it will latch the last input.

The SRAM in this design is used to store abnormal signals that come from the CUSUM module. Here, the SRAM is only designed for validation of data storage. For further improvement, the abnormal signals could be housed in other storage (such as an SD

card). In this way it could achieve off-line analysis, or even pass through those signals by PCI Express to PC to achieve on-line analysis. Whatever the choice, SRAM can be used as a good buffer.

5.9 Module Assembly

Module assembly is the key point in digital design. Chapter 5.5, 5.6, 5.7 briefly introduced the design method for each single module. How to assemble these modules and have them execute as expected then becomes a critical task. When using C or other processor, all functions are executed sequentially. As Verilog modules all run in parallel this means, if stitching these modules simply, they cannot achieve the expected operating results.

Figure 5-17 illustrates the function setup designed in C language. The first step was to establish four sub-functions, ALT_MUTI_ADD, SQRT, SP_MEAN and CUSUM. The next step was to build the function SD_TOP_32 that we can call ALT_MUTI_ADD and SQRT sequentially. Finally, we set up the main function with which we can call these three functions (SP_MEAN, SD_TOP_32, and CUSUM) directly.

```
int main (...)
{
    SP_MEAN(...);
    SD_TOP_32 (...);
    CUSUM (...);
}

SD_TOP_32 (...)
{
    ALT_MUTI_ADD(...);
    SQRT(...);
}
```

Figure 5-17 C Language Design Structure

In the main function, C language has already executed the “function call,” “return,” and other commands to achieve the above functions. As there is no such convenient terminology in Verilog language it is virtually impossible to achieve the “sequential operation” in digital design. However, by adding some control signals there is a way to

imitate the module sequential operation. For example, one can add a “Start” signal to activate the first module. Once the first module is completed, it can then generate a “Done” signal to inform the second module. In this way, these communication signals can achieve the sequential operation.

In the CUSUM IP core design, the author used two IP cores that parameterized from Quartus II Mega function. However, most of the IP cores supported by Quartus are multi-functional and do not provide communication signals such as “Start,” and “Done,” as these cores do not contain sub module, functional module and control module. They are therefore only employed in many “always @” statements inside modules to achieve desired results, which increases the difficulty for sequential operations.

There are two ways to solve this problem. The first way is rewrite two of the IP cores, adding communication signals. Another way of approaching the issue is to consider a parallel operation instead of a sequential operation. From the perspective of saving development time, the author decided to adopt the second method. This method necessitates the precise knowledge of the execution time for each module in order to build the pipeline structure via sequential timing. The results obtained by simulation (provided in Chapter 6) shows there are 32 clock cycles consumed by SP_MEAN module, 9 clock cycles used by SD_TOP_32 module and another 33 clock cycles used by the CUSUM module. Obviously, the running times of these three modules are not equal, meaning that it cannot build the pipeline structure directly, as it will cause a data hazard when processing old data as new data comes in.

The author found a way to solve this problem that increased the latency for SD_TOP_32 module. The total latency added on the SD_TOP_32 module is 23 clock cycles. This means although the computing time only consumes 9 clock cycles, the results are being latched until the 33rd clock cycle. Figure 5-18 shows the pipeline structure of these modules

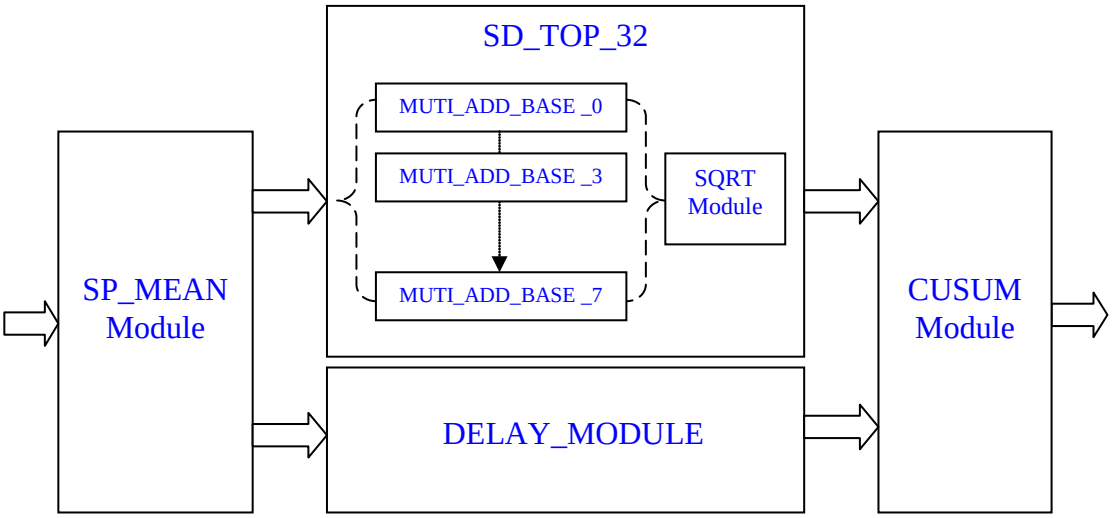


Figure 5-18 Pipeline Structure of Design Module

Another improvement is adding the DELAY module in this design. The role of this module is obtained the output from SP_MEAN module, delay 32 clock cycles to synchronise with SD_TOP_32 module and then passes to CUSUM module. By using this way, this design can achieve pipeline structure and seamless connection for input/output data.

5.10 Discussion Blocking and Non-blocking Operation in Design

Blocking and non-blocking operations are the most easily confused structures in both Verilog and VHDL Languages. It is difficult to understand the difference between them when they are simulating and synthesizing. Failure to understand where and how to use the respective languages can lead not only to unexpected behaviour, but also to mismatches between simulation and synthesis.

In software design, functionality is created by defining operations that are executed in a predefined sequence. In Verilog design this type of execution can be defined as blocking. This means that future operations are blocked until after the current operation has been completed. All future operations are under the assumption that all previous operations have been completed and all variables in memory have been updated. By comparison, a non-blocking operation executes independent of order, with updates

triggered by specified events, and all updates occur simultaneously when the trigger event occurs.

Before further analysis of the execution process of blocking and non-blocking operations, one needs to understand the IEEE Verilog standard stratified event queue. The stratified event queue, shown in Figure 5-19, includes active events, inactive events, non-blocking events and monitor events. Obviously, the blocking operation belongs to active events. Active events refer to events executed in current simulation time. The non-blocking operation belongs to two events; active events and non-blocking events. It is the evaluating RHS (right hand side) of the non-blocking operation in active events that then update the LHS (left hand side) in non-blocking events.

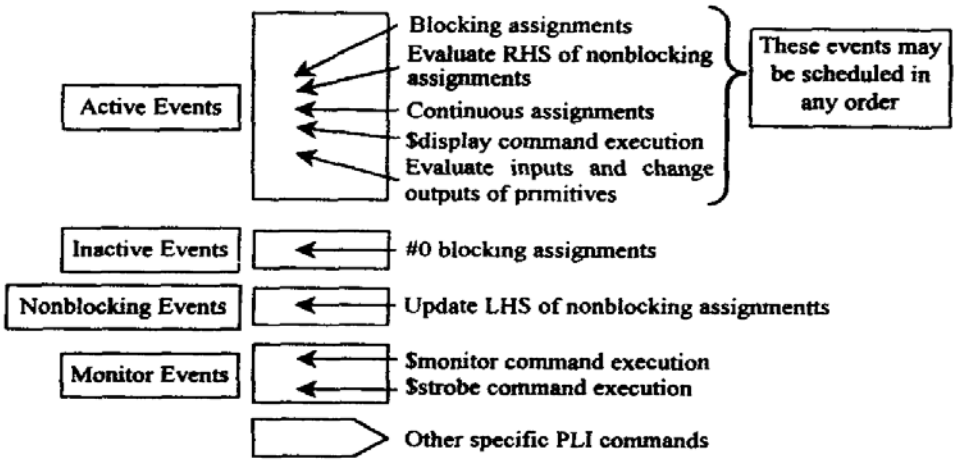


Figure 5-19 Stratified Event Queue

Classical theory establishes that blocking operations should be used in combinational logic and non-blocking operations should be used in sequential logic [47]. According to the author's experience this theory is adhered to in most of cases. However, in some cases the blocking operation can also used in sequential logic. Figure 5-20 shows a case that requires an instant result in "always" block. Using the blocking operation can obtain a result in one clock cycle, while a non-blocking operation needs two clock cycles. If the timing constraint is only one clock cycle as it is here, it would only use the blocking operation.

```
always @ (posedge clk or negedge rst_n)
……
C = A + B;
D = C + B;
……
```

```
always @ (posedge clk or negedge rst_n)
…
case:
0: C <= A + B;
1: D <= C + B;
…..
```

Figure 5-20 Blocking (left) and Non-Blocking (right) assignment

In order to meet the timing requirement of the design, the author adopted a blocking operation in the CUSUM module so as to obtain instantaneous calculation results. By observing the synthesized RTL view, the author found the result registers were synthesized to "wire" structure, thus allowing the calculation to be achieved.

5.11 Discussion Synchronous Reset and Asynchronous Reset

In FPGA design, reset can be achieved in two ways; synchronous reset and asynchronous reset. Synchronous reset means the reset signal is synchronous with the system clock signal and is triggered at the rising edge of the input system clock. Asynchronous reset means the reset signal is not synchronous with the system clock and can reset the system at any time. However, they both have their respective advantages and disadvantages.

Synchronous reset in FPGA design is often achieved by adding an "AND" gate to the input signal. Figure 5-21 shows Verilog code and its RTL view. This design is easy to synthesize to synchronous sequential circuits which greatly benefit the timing analysis, with the maximum frequency "fmax" normally higher than other reset designs. It also can filter out the glitches that are higher than the system clock's frequency because the reset signal is only activated by the rising edge of the system clock.

Synchronous reset also has some disadvantages. The reset signal must be longer than the system clock cycle otherwise it cannot be identified by system. One also needs to consider the side effects caused by clock skew, combinational logic path delay, and reset delay. Another disadvantage is the logic resource utilization as most logic resources only have an asynchronous reset port. If using a synchronous reset, the synthesizer will generate a combinational logic with the input port, which will require more logical resource.

```
always @ (posedge clk)
if (!rst)
   A <= 1'b0;

……
```

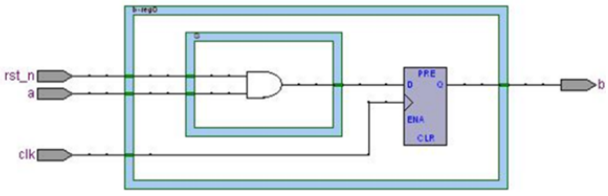


Figure 5-21 Synchronous Reset with RTL View

Asynchronous reset in FPGA design often connects the reset signal to the CLR port. Figure 5-22 shows its Verilog code and RTL view. The advantages of asynchronous reset are obvious; compared with synchronous reset, asynchronous reset can save a lot of logic resources with the global reset port "GSR" easily used in FPGA design. The defects of asynchronous reset are more serious however, and often appear when the reset signal is being released. If the reset signal is released near the rising edge of the clock cycle, this can easily cause the register metastable status.

```
 always @ (posedge clk or negedge rst_n)
if (!rst) b <= 1'b0;
else      b <= a;

always @ (posedge clk or negedge rst_n)
if (!rst) c <= 1'b0;
else      c <= b;
……
```

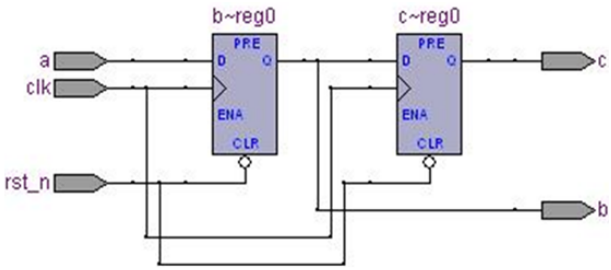


Figure 5-22 Asynchronous Reset with RTL View

In Figure 5-21, at the rising edge of the clock cycle normally "b" will be equal to "a," and "c" will be equal to "b." Once the reset signal is triggered, "b" and "c" are all equal to zero. However, one cannot determine when the reset signal will finish, so if the reset signal recovers faster than the register's latch edge setup time (*Latch edge setup time = latch edge + hold time*), this will cause the register "b" and "c" metastable status.

The author discovered a way to solve this problem in Altera's System Verilog notes [48]. That is, asynchronous reset and synchronous release. Figure 5-23 shows the verilog code and its RTL view.

```
always @ (posedge clk)
rst_nr <= rst_n

always @ (posedge clk or negedge rst_nr)
 if (!rst_nr) b <= 1'b0;
 else         b <= a;

always @ (posedge clk or negedge rst_nr)
if (!rst_nr) c <= 1'b0;
else         c <= b;
……
```

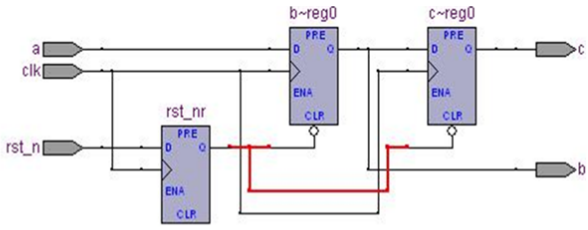


Figure 5-23 Asynchronous Reset and Synchronous Release with RTL View

This method uses the system clock cycle's rising edge to synchronize the asynchronous reset signal's rising edge. The advantage of this approach is the timing analysis tools which automatically check the relation between the asynchronous reset signal and the system clock cycle (recovery/removal). This ensures the reset signal's recover time is equal to, or greater than, the register's latch edge setup time. In this design, the author implemented this reset method to ensure the stability of the core.

5.12 Summary

Firstly, an overview of FPGA design methods was given in this chapter. The author then described the design process and the design requirements for each module. The main focus of the chapter was module assembly as it involves the timing sequence operation. A detailed analysis of module assembly methods was therefore discussed, as well as the solution of timing sequence operations. Finally, the author analyzed the Verilog coding style for this design.

Chapter 6 Verification and Evaluation

This chapter presents the verification process of this work. First, the chapter outlines the testbench design of the project. The following section covers the ModelSim software in application. The remainder of the chapter then describes the verification process of the CUSUM core and outlines the results.

6.1 Testbench Description

The term testbench usually refers to the simulation code used to create a predetermined input sequence for a design, and optionally to the observation of the response. Testbench are implemented using System Verilog, but they may also include external data files or C routines.

Figure 6-1 shows how a testbench interacts with a design under verification (DUV). The testbench provides inputs to the design and watches any outputs. In the context of the design, the testbench is effectively a model of the universe with the verification challenge being to determine what input patterns to supply to the design, and what the expected output of a properly working design is when submitted to those input patterns.

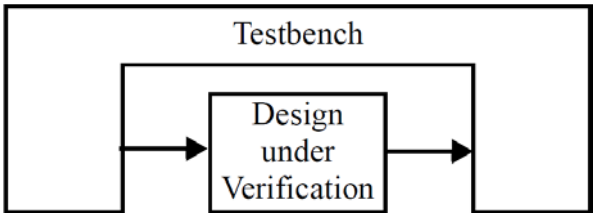


Figure 6-1 Structure of a Testbench and Design under Verification

Today, in the era of multi-million gate ASICs and FPGAs, reusable intellectual property (IP), and system-on-a-chip (SoC) designs, verification consumes about 70% of the design effort. Design teams, properly staffed to address the verification challenge, include engineers dedicated to verification. The proportion of verification engineers can

be up to twice the number of RTL designers. Verification is often considered after the design has been completed, when the schedule has already been ruined, which compounds the problem. Verification is also the target of the most recent tools and methodologies which attempt to reduce the overall verification time by enabling parallelism of effort, higher abstraction levels, and automation.

The basic verification method is functional verification. The main purpose of functional verification is to ensure that a design implements the intended functionality. As shown in Figure 6-2, functional verification reconciles a design with its specification. Without functional verification, one must trust that the transformation of a specification document into RTL code was performed correctly, without misinterpretation of the specification's intent.

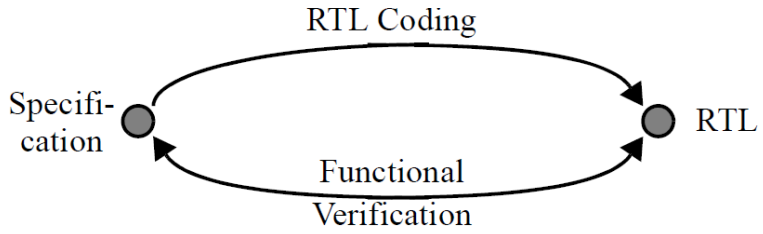


Figure 6-2 Functional Verification Paths

Functional verification can be accomplished using three complementary approaches: black box, white box and grey box. With a black box approach, functional verification is performed without any knowledge of the actual implementation of a design. All verification is accomplished through the available interfaces, without direct access to the internal state of the design or knowledge of its structure and implementation. This method suffers from an obvious lack of visibility and controllability. It is often difficult to setup an interesting state combination or to isolate some aspect of functionality. It is also difficult to observe the response from the input and locate the source of the problem. The advantage of black box verification is that it does not depend on any specific implementation. Whether the design is implemented in ASIC, RTL gates or entirely in software, is irrelevant. The black box functional verification approach forms a true conformance verification that can be used to show that the particular design implements the intent of a specification.

By comparison, the white box approach has full visibility and controllability of the internal structure and implementation of the design being verified. The results can be

easily observed as the verification progresses, with discrepancies from the expected behaviour immediately reported. This approach, however, is tightly integrated with a particular implementation, meaning that changes in the design may require changes in the testbench. Furthermore, these testbenches cannot be used in gate-level simulations, or alternative implementations or further redesigns. It also requires detailed knowledge of the design implementation to know which significant conditions to create and which results to observe.

Grey-box verification then is a compromise between the aloofness of the black box verification and the dependence on the implementation of white-box verification. While the former may not fully exercise all parts of a design, the latter is not portable. As with black-box verification, a grey-box approach controls and observes a design entirely through its top-level interfaces. However, the particular verification being accomplished is intended to exercise significant features specific to the implementation. The same verification of a different implementation would be successful, but the verification may not be particularly more interesting than any other black box verification. A typical grey box test case is one written to increase coverage metrics, with the input stimulus designed to execute a specific line of code, or create a specific set of conditions in the design.

6.2 ModelSim Tools

ModelSim is a powerful HDL simulation tool that allows simulating the inputs of the modules and viewing both output and internal signals. ModelSim offers VHDL, Verilog or mixed language simulation. Coupled with the most popular HDL debugging capabilities in the industry, ModelSim is known for delivering high performance, ease of use, and outstanding product support.

Model Technology's award-winning Single Kernel Simulation (SKS) technology enables transparent mixing of VHDL and Verilog in one design. ModelSim's architecture allows platform independent compilation via the outstanding performance of native compiled code.

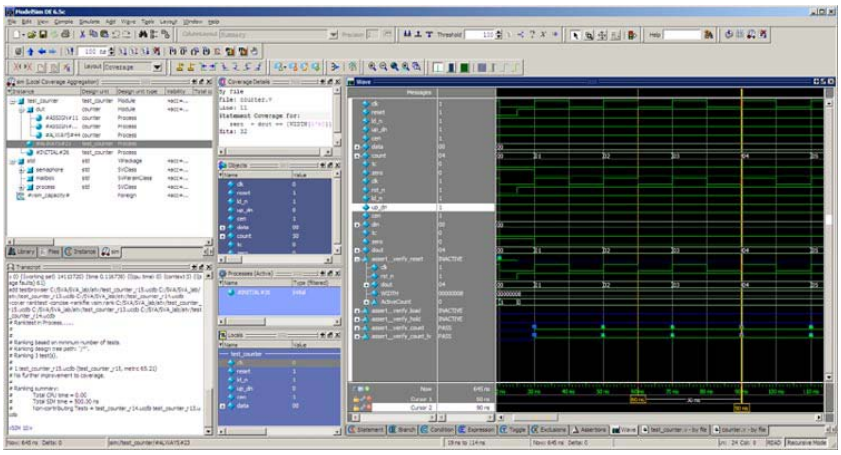


Figure 6-3 ModelSim Interface

An easy to use graphical user interface enables the user to quickly identify and debug problems, aided by dynamically updated windows. Figure 6-3 shows the user's interface. For example, selecting a design region in the structure window automatically updates the source, the signals process and variable windows. These cross-linked ModelSim windows create a powerful debug environment. Once a problem is found, it can edit, recompile and re-simulate without leaving the simulator. So, the author uses Modelsim to verify the design,

6.3 SP_MEAN Module Verification

Before verification, a testbench for the SP_MEAN module was undertaken. The verification process can be divided into two parts; observation of the timing sequence and verification of the results. In order to observe the timing sequence efficiently, the

author used a set of sequential numbers at beginning. Figure 6-4 shows the verification result. The result shows the conversion of the serial input to parallel output while giving the mean value for each output group. One can easily see from the timeline cursor on the figure that each group of output consumed 32 clock cycles. Notably this approach avoids any unnecessary time delay for each output and reduces the chance of data missing.

Result verification was undertaken after the timing sequence analysis. The author used random numbers as input to detect whether the output of the mean value was correct or not. Figure 6-5 illustrates the returned answers of the result verification. The author randomly selected a set of outputs (cycled in Figure 6-5) and calculated the result manually.

Table 6-1 shows the manually calculated result to be the same as the verification result. It can be considered therefore, that this module met the design requirements.

Table 6-1 Result Comparison

Manually Calculated Result:

0	1	2	3	4	5	6	7	*SUM*
-6343	-28897	-2349	12165	-30600	22875	19273	-20929	-34805
8	9	10	11	12	13	14	15	
-20694	25432	14470	3214	-3428	-26118	-17370	6003	-18491
16	17	18	19	20	21	22	23	
-19293	-22481	-27981	22111	3396	-521	14027	6886	-23856
24	25	26	27	28	29	30	31	
-9382	-16087	-7443	27866	-18843	-7499	-27425	6521	-52292
							Sub Total	-129444
							Mean	-4045.13

ModelSim Result:

0	1	2	3	4	5	6	7	
-6343	-28897	-2349	12165	-30600	22875	19273	-20929	
8	9	10	11	12	13	14	15	
-20694	25432	14470	3214	-3428	-26118	-17370	6003	
16	17	18	19	20	21	22	23	
-19293	-22481	-27981	22111	3396	-521	14027	6886	
24	25	26	27	28	29	30	31	
-9382	-16087	-7443	27866	-18843	-7499	-27425	6521	
							Mean	-4046

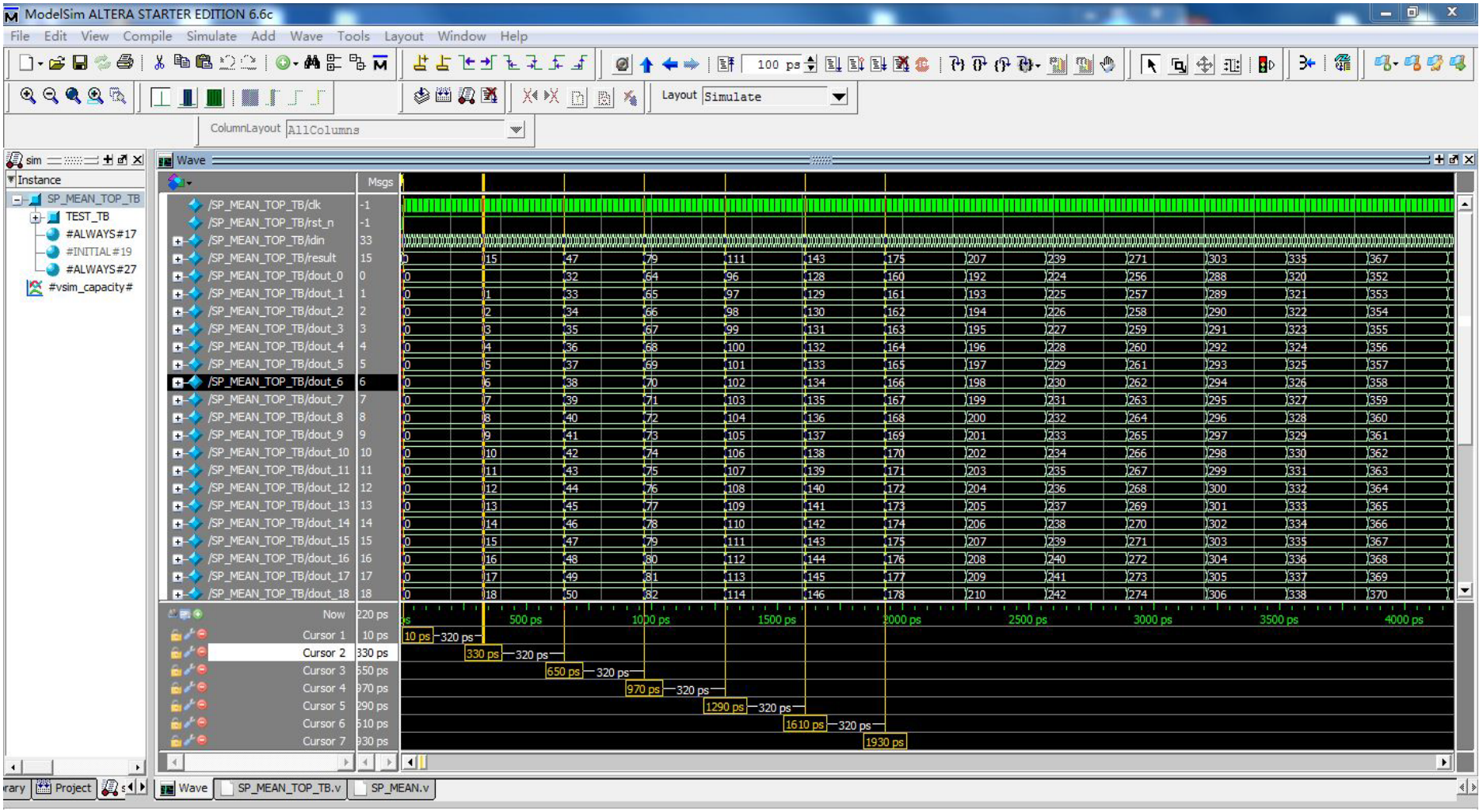


Figure 6-4 Verification of SP_MEAN Module with Sequential Numbers

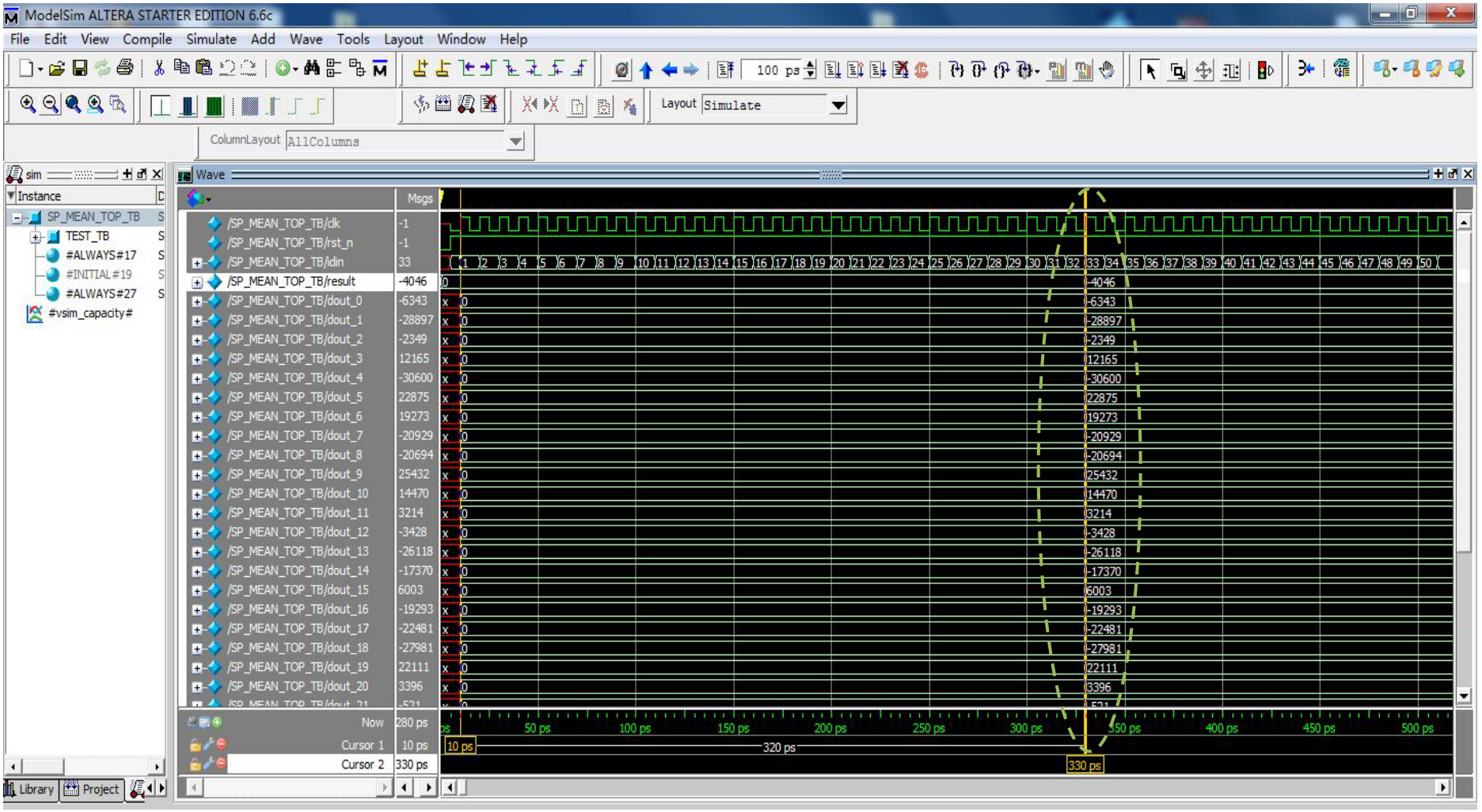


Figure 6-5 Verification of SP_MEAN Module with Random Numbers

6.4 Standard Deviation Module (SD_TOP_32) Verification

The main purpose of the verification of a SD module is to verify its computational time and calculated result. This module is verified individually and the author used same input data as previous module (SP_MEAN Module). The verification can be divided into three steps. First, the author uses a fixed input number to verify its time consumption and result. Then, the author uses random numbers to increase the complexity of calculation and examine whether the module can deliver the same performance or not. Finally, the module adjusted by the timing requirement is verified as to whether the module meets the design requirements.

Figure 6-6 shows the results. Clearly, it can be seen from the cursor that the time consumption is 9 clock cycles. In order to assemble pipeline structure, the time consumption of this module can extend up to 32 clock cycles, as illustrated in Figure 6-7. The manually calculated results shown in the table below are the same as the verified results. After several episodes of randomly achieved verification, the author found the results to be all correct and the time consumption the same. Therefore, it can confirm that this module meets the design requirements.

Table 6-2 Result Comparison

Manually Calculated Result:

0	1	2	3	4	5	6	7
-6343	-28897	-2349	12165	-30600	22875	19273	-20929
8	9	10	11	12	13	14	15
-20694	25432	14470	3214	-3428	-26118	-17370	6003
16	17	18	19	20	21	22	23
-19293	-22481	-27981	22111	3396	-521	14027	6886
24	25	26	27	28	29	30	31
-9382	-16087	-7443	27866	-18843	-7499	-27425	6521

Standard Deviation 17580.84

ModelSim Result:

0	1	2	3	4	5	6	7
-6343	-28897	-2349	12165	-30600	22875	19273	-20929
8	9	10	11	12	13	14	15
-20694	25432	14470	3214	-3428	-26118	-17370	6003
16	17	18	19	20	21	22	23
-19293	-22481	-27981	22111	3396	-521	14027	6886
24	25	26	27	28	29	30	31
-9382	-16087	-7443	27866	-18843	-7499	-27425	6521

Standard Deviation 17580

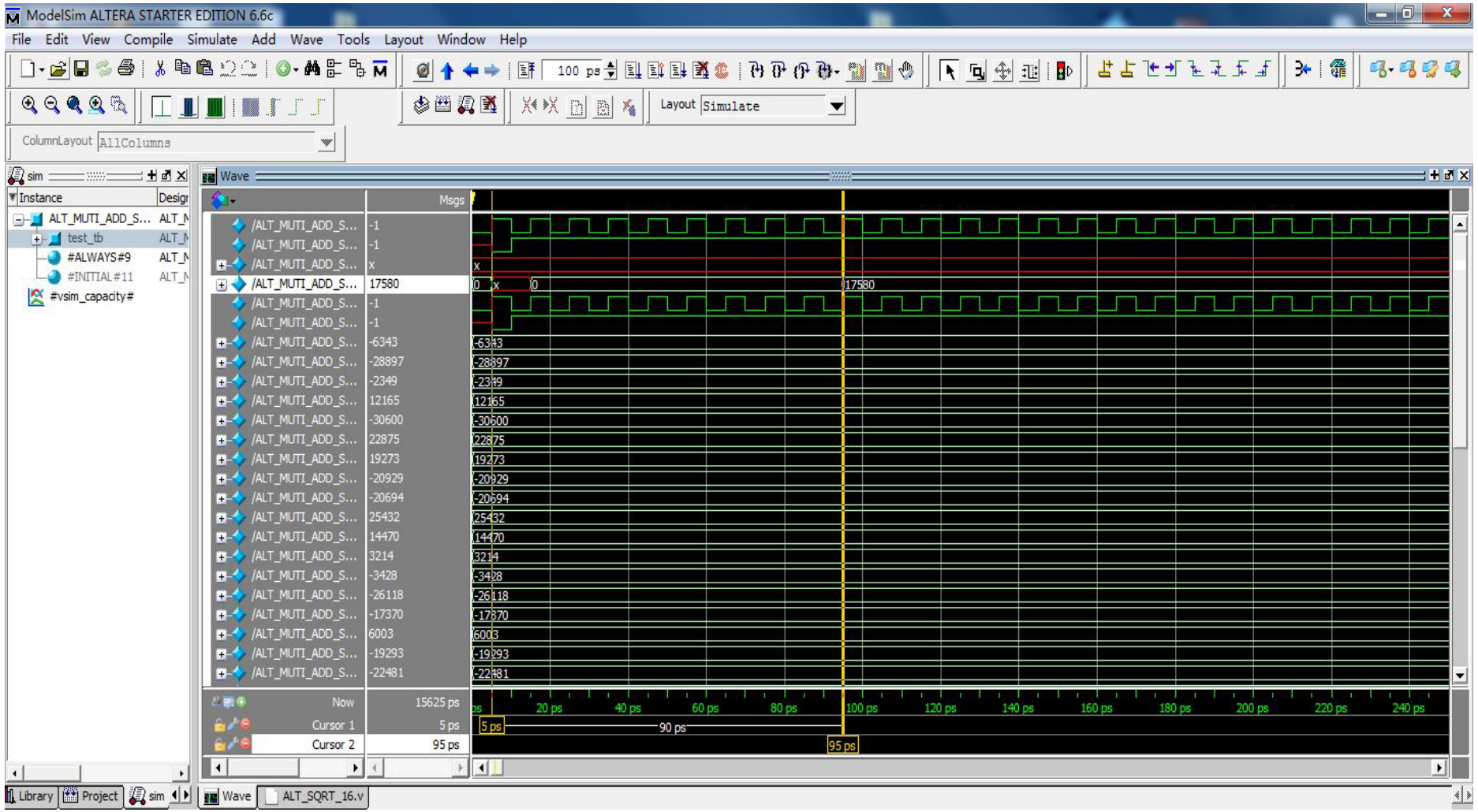


Figure 6-6 Standard Deviation Module Verification

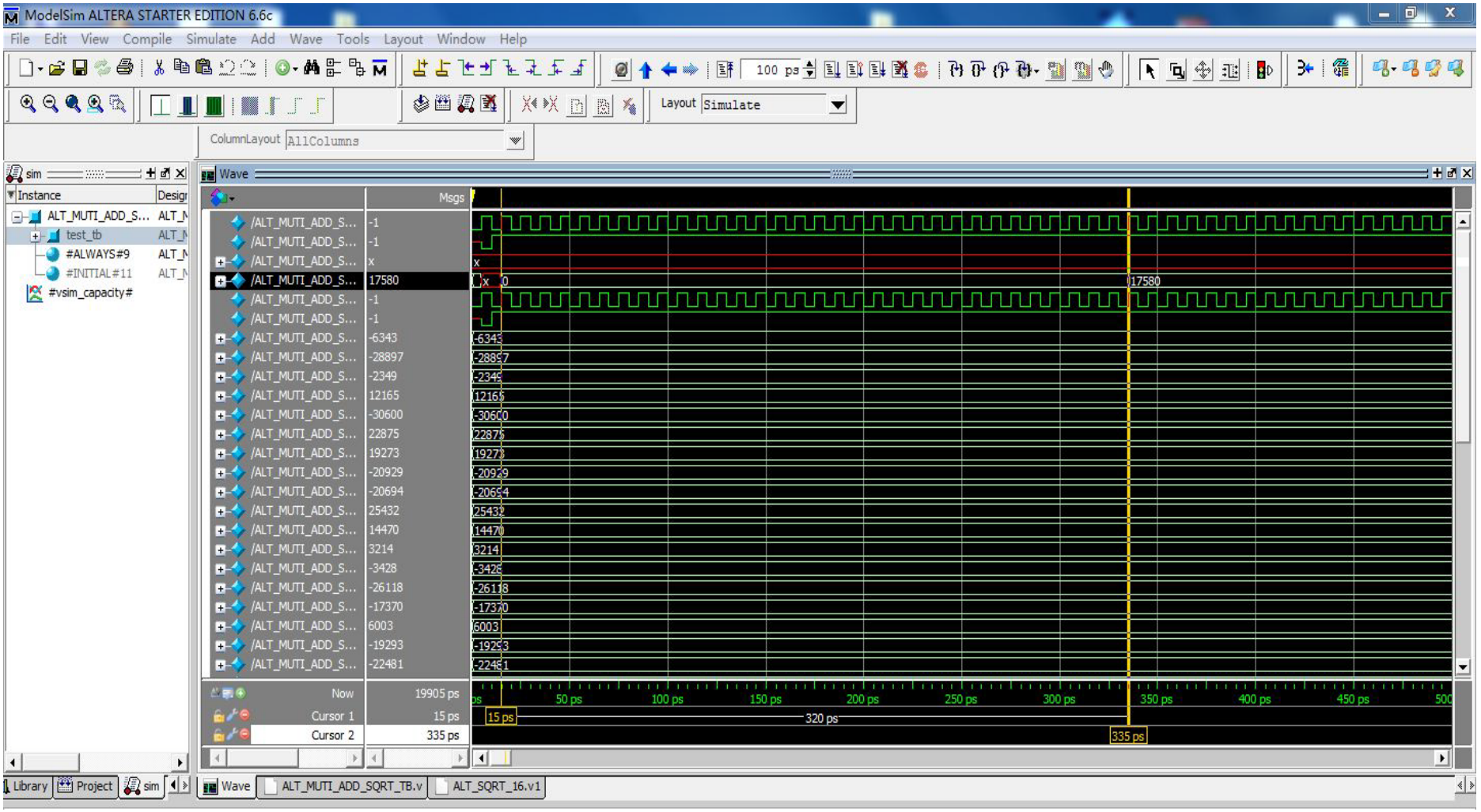


Figure 6-7 Standard Deviation Module Verification with Output Delay

6.5 CUSUM Module Verification

The verification of a CUSUM module aims to verify its detectability and time consumption. Because this is single module verification it also requires manually assigned input, as well as the mean and threshold value. The author chooses pervious data to observe it's time consumption. One also needs to observe the output signal, once the input signal exceeds the threshold.

The verification result is shown in Figure 6-9. The threshold is calculated by SD module which is +/-17580 and the mean is -4046. One can see that the time consumption to finish a set of input is 32 clock cycles. As observed from the input data, 12 sets of data are less than the threshold. In Figure 6-10 these data are shown as having a high impedance status, which means that these pieces of data are normal input signals and do not need to be output. The remaining data which exceed the threshold are output from the "DOUT" port. The manually calculated result is plotted in Figure 6-8. From the graph it can be clearly seen that the accumulated error exceeds the threshold when it has accumulated to the point of the seventh input data. Two sets of testing results were matched. After several episodes of random verification, the author found the results to be all correct with the time consumption remaining at 32 clock cycles with each data group. This module can therefore be considered to meet the design requirements.

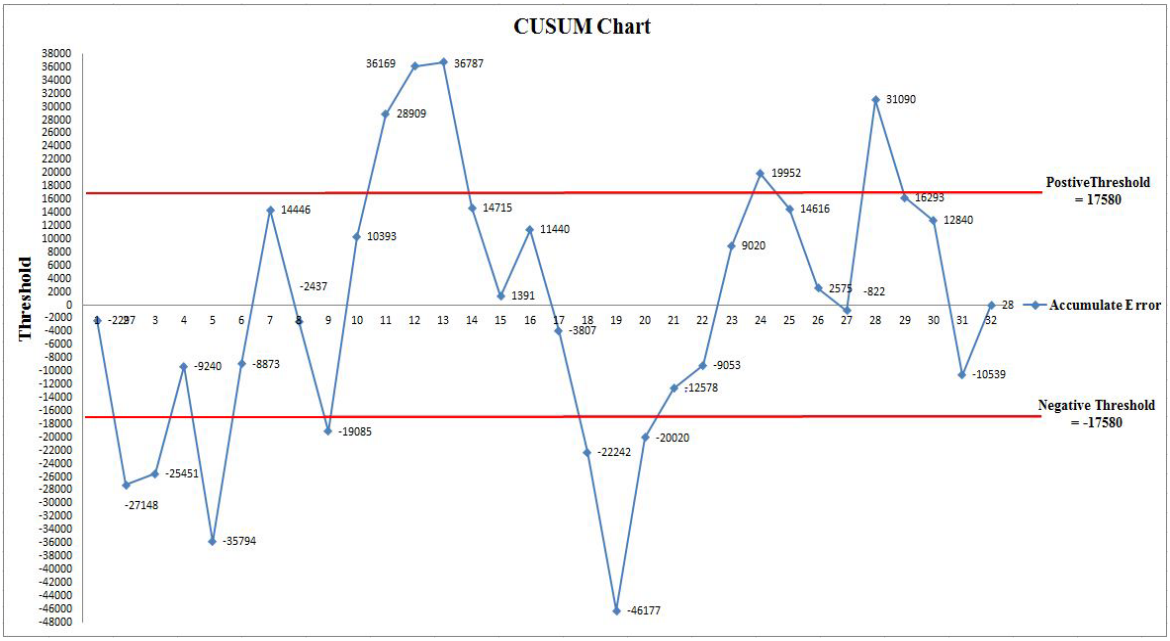


Figure 6-8 CUSUM Module Manually Verification

Table 6-3 CUSUM Module Manually Verification Result

Mean	**-4046**
Threshold	**17580 /-17580**

	Input	Error	Accumulate Error	Status
0	-6343	-2297	-2297	
1	-28897	-24851	-27148	Over Threshold
2	-2349	1697	-25451	Over Threshold
3	12165	16211	-9240	
4	-30600	-26554	-35794	Over Threshold
5	22875	26921	-8873	
6	19273	23319	14446	
7	-20929	-16883	-2437	
8	-20694	-16648	-19085	Over Threshold
9	25432	29478	10393	
10	14470	18516	28909	Over Threshold
11	3214	7260	36169	Over Threshold
12	-3428	618	36787	Over Threshold
13	-26118	-22072	14715	
14	-17370	-13324	1391	
15	6003	10049	11440	
16	-19293	-15247	-3807	
17	-22481	-18435	-22242	Over Threshold
18	-27981	-23935	-46177	Over Threshold
19	22111	26157	-20020	Over Threshold
20	3396	7442	-12578	
21	-521	3525	-9053	
22	14027	18073	9020	
23	6886	10932	19952	Over Threshold
24	-9382	-5336	14616	
25	-16087	-12041	2575	
26	-7443	-3397	-822	
27	27866	31912	31090	Over Threshold
28	-18843	-14797	16293	
29	-7499	-3453	12840	
30	-27425	-23379	-10539	
31	6521	10567	28	

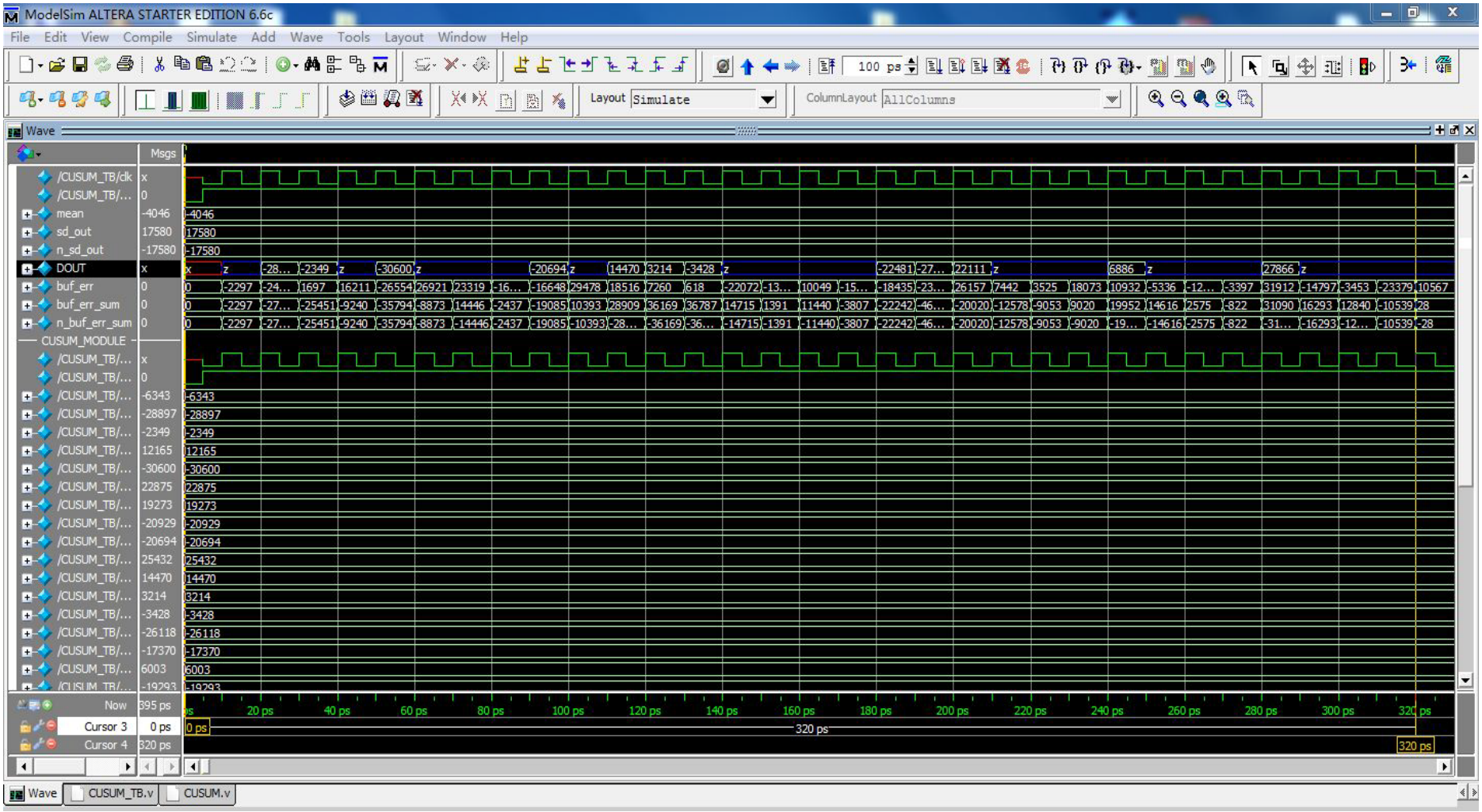


Figure 6-9 CUSUM Module Verification

6.6 Top Module Verification

Top module verification refers to the verifying of the overall function and timing sequence of the module combined with the SP_MEAN_ Module, SD_TOP_32 Module, Delay Module and CUSUM Module. Compared with previous verifications, top module's verification is more complicated as all the sub-modules are running parallel when the clock signal is being activated. This requires good coordination between the sub-modules, as well as the observation of each sub-module's performance in top entities after being parameterized. The main task of top module's verification regards the latency period of the module. Its success is determined by its ability to achieve seamless data transfer between sub-modules and thus enable the pipeline design to achieve the desired results.

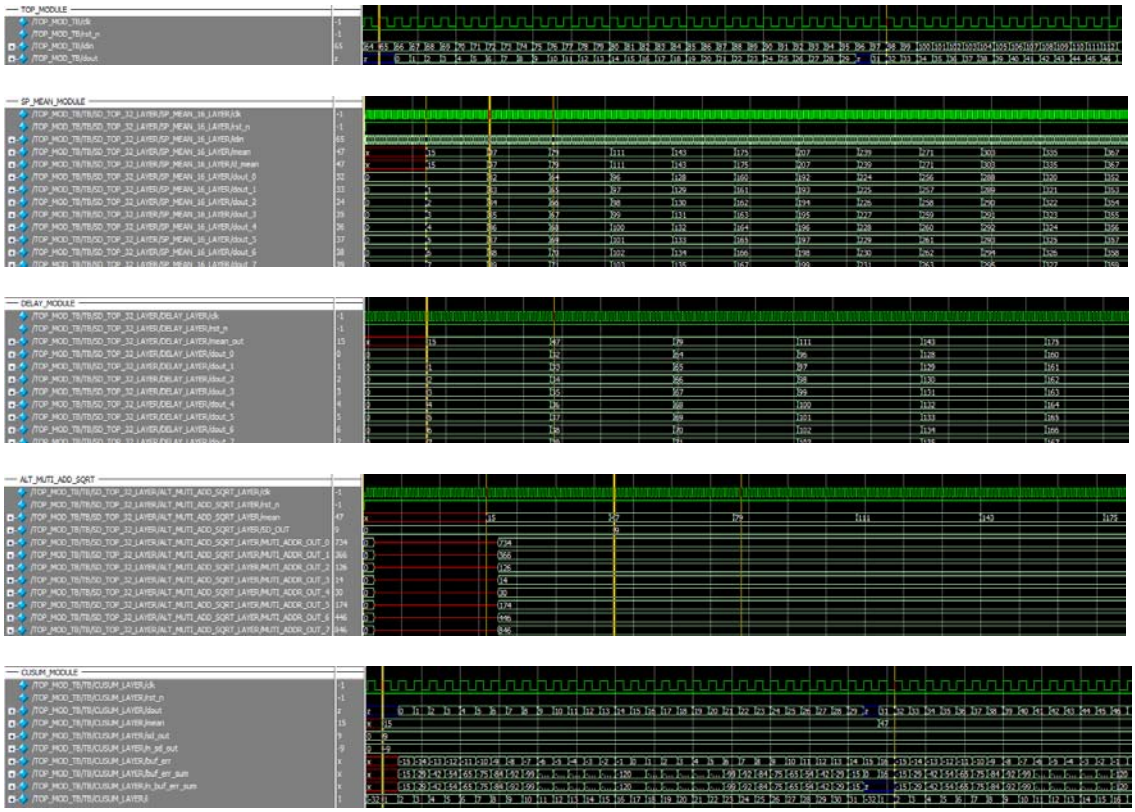


Figure 6-10 TOP Module Verification

Figure 6-10 shows the verified result. From internal registers SP_MEAN module output parallelised data can be seen, with the mean value at the 32rd clock cycle. Here, the outputs are separated into two groups, one group to pass to SD_TOP_32 module to calculate the SD, and the other group to send to the Delay Module to meet the timing requirement. The SD calculation is done at the 64th clock cycle, then sends the result to the CUSUM module at the same clock cycle. The internal result register has been

integrated to "wire" status; it does not need any extra clock cycles to drive the result to the next module. Delay Module passes the data that has been delayed by 33 clock cycles to the CUSUM Module at same time. At the 65^{th} clock cycle, CUSUM Module outputs the detecting result. This continues until the 97^{th} clock cycle to finish the detection of first group of input signals.

It can also be seen that the second group of input signals that enter the SP_MEAN module are waiting for parallelized processing when the first group of input signals are in the SD calculating process. The calculation for the first group and parallelized processing for the second group are completed at same time. This means that at the time that the second group signals enter the CUSUM Module, the first group signals have just passed through. Similarly this operation applies for the following groups.

It can be concluded from the above that the entire latency is 64 clock cycles. This means it will consume 64 clock cycles to process one group of input signals. In this way the Pipeline structure can achieve a continuous processing flow, avoids wasting clock cycles, and maximizes processing speed. After observing several groups of sampling data, it was found that the data could achieve seamless transfer without any missing. This module can therefore be considered to meet the design requirements.

6.7 Summary

First, this chapter gave an overview of FPGA verification methods. The verification was summarised based on two aspects; timing sequence analysis and functional analysis. In the context of these two aspects, the author verified each sub module by using ModelSim software. Timing analysis sequence results and functional analysis results were obtained from the waveform. Finally, the chapter concluded with the analysis of top module verification. All the modules discussed met the design requirements and performed well.

Chapter 7 Conclusions and Future Work

CUSUM algorithm, which was used for transient signal detection, was explained in detail in Chapter 2. Chapter 4 provided a power analysis for the CUSUM core. With leading-edge technology being continuously developed to maximize the performance and minimize the power consumption in FPGA devices, the programmable power technology and selectable core voltage enabled the lowest possible power for Altera's FPGA. The post-programmed power consumption on the Cyclone IV device was only 136.75 mW, which, when compared with PC based detection, points to a significant energy saving.

In Chapter 5, the FPGA design methods were outlined and the design requirements and design process for each sub-module then were explained. At the end of chapter 5, a detailed discussion of the modules assembly process for the entire CUSUM algorithm core was presented, together with the problems and solutions encountered within the design. In chapter 6, the author initially gave a brief explanation of the FPGA verification process, then proceeded to verify each of the sub-modules, as well as the top module. The verification waveforms and results were plotted in this chapter which showed the detection speed to be 64ns (32 clock cycles) for each sampling group and the pipeline incubation period to be 128ns (64 clock cycles). The results were found to match the author's expectations, and the timing sequence met the design requirement.

There are still improvements that need to be made to this system however, in order to achieve processing several TByte data per day. These improvements can be achieved via three aspects: coding design, multiprocessor operation, and multi-core operation. From the perspective of coding design, a memory arbitration module can be designed to coordinate data transfer between modules. The memory arbitration module could be implemented in two places. Firstly, as mentioned in chapter 5.2.1, a multiple FIFO structure could be designed in which the memory arbitration is used to control the storage of the first part of a data flow into one FIFO, while processing the data from another FIFO at the same time. Secondly, it can control a multiplexer, and choose to

either connect the data from the SP_MEAN module to the SD module, or connect the data from the SP_MEAN module to the CUSUM module. Employing a memory arbitration module could further improve the internal logic resource utilization.

Another discussed improvement was the use of multiprocessor architecture to enhance processing speed. The SOPC Builder was observed to allow users to add custom instructions to the Avalon Bus and build their own system via the NIOS II processor. The number of processors in a multiprocessor system was shown to be scalable, with processors able to be easily added or removed from the multiprocessor system. Avalon Bus was discussed in terms of its capacity to control the on-chip memories which store the shared data in the detection process. Further, the workload of detection was shown to be able seamlessly allocated to any number of processors. The resource limitation of extending such a multiprocessor system, which is mainly associated with the on-chip shared memories and logic resources of adding an additional slave processor, is relatively low. Therefore, more processors are easily added to improve the performance if more on chip memory is available. One needs to balance the process efficiency however, with the number of processors. This efficiency ratio is defined as [53]:

7-1:

$$E = \frac{Execution\ time\ using\ one\ processor}{Execution\ time\ using\ multiporcessor\ *\ number\ of\ processors}$$

If the computational complexity is not sufficient to have all processors running at the same time, adding more processors will decrease the performance and also waste resources.

Another issue that needs to be considered is the use of one NIOS II processor to control a multi CUSUM core, or the use of multiprocessors to process one CUSUM core. According to the author's previous experience researching high performance computing architecture, the wrong selecting architecture will introduce side effects to the processing speed. Here, using one NIOS II processor to control multi CUSUM cores can be considered as a kind of intra-node architecture, while using a multiprocessor to process one CUSUM core can be considered as a kind of inter-node architecture. Generally, for small or medium complexity computation, intra-node architecture achieves higher performance than inter-node architecture. This is due to the

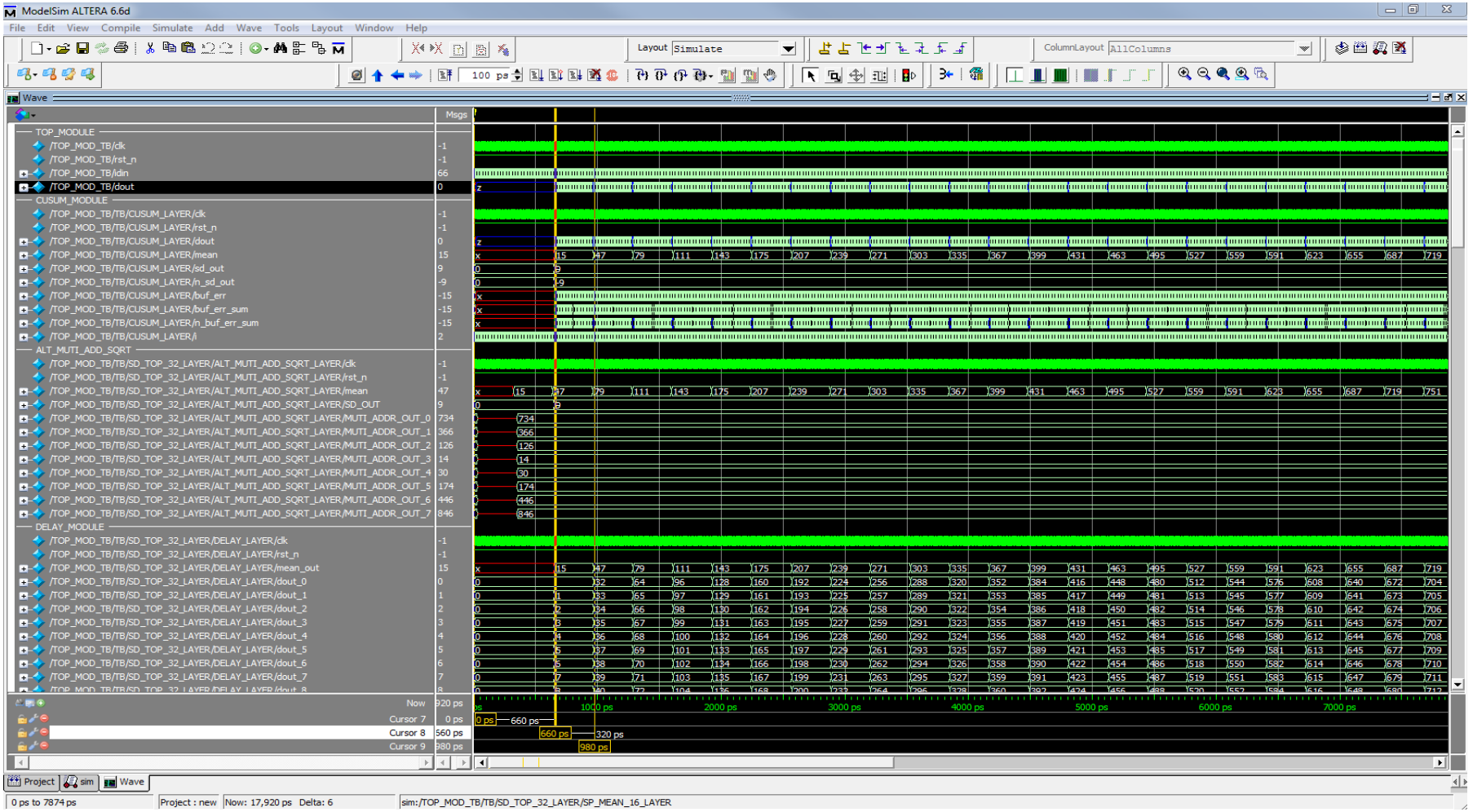
communication time required within inter-node architecture, which slows down the processing speed. If the computational complexity is increased multiprocessor architecture will deliver improved performance, so there needs to balance these two forms of computing architecture in the future work.

Appendix A CUSUM Module Verify Data

Mean	-4046
Threshold	17580 /-17580

	Input	Error	Accumulate Error	Status
0	-6343	-2297	-2297	
1	-28897	-24851	-27148	Over Threshold
2	-2349	1697	-25451	Over Threshold
3	12165	16211	-9240	
4	-30600	-26554	-35794	Over Threshold
5	22875	26921	-8873	
6	19273	23319	14446	
7	-20929	-16883	-2437	
8	-20694	-16648	-19085	Over Threshold
9	25432	29478	10393	
10	14470	18516	28909	Over Threshold
11	3214	7260	36169	Over Threshold
12	-3428	618	36787	Over Threshold
13	-26118	-22072	14715	
14	-17370	-13324	1391	
15	6003	10049	11440	
16	-19293	-15247	-3807	
17	-22481	-18435	-22242	Over Threshold
18	-27981	-23935	-46177	Over Threshold
19	22111	26157	-20020	Over Threshold
20	3396	7442	-12578	
21	-521	3525	-9053	
22	14027	18073	9020	
23	6886	10932	19952	Over Threshold
24	-9382	-5336	14616	
25	-16087	-12041	2575	
26	-7443	-3397	-822	
27	27866	31912	31090	Over Threshold
28	-18843	-14797	16293	
29	-7499	-3453	12840	
30	-27425	-23379	-10539	
31	6521	10567	28	

Appendix B Top Module Verification Plot



Appendix C CUSUM Core

CUMULATIVE SUM Interconnect Matrix IP Core

Version 1.0

```
module TOP_MOD ( clk, rst_n, idin, dout );

input clk;
input rst_n;
input [15:0] idin;
output [15:0] dout;

wire   [15:0]  dout_0, dout_1, dout_2, dout_3,
                dout_4, dout_5, dout_6, dout_7,
                dout_8,  dout_9,  dout_10, dout_11,
                dout_12, dout_13, dout_14, dout_15,
                dout_16, dout_17, dout_18, dout_19,
                dout_20, dout_21, dout_22, dout_23,
                dout_24, dout_25, dout_26, dout_27,
                dout_28, dout_29, dout_30, dout_31;
wire [15:0] mean;
wire [15:0] sd_out;

SD_TOP_32 SD_TOP_32_LAYER
(
  // Input
  .clk(clk), .rst_n(rst_n), .idin(idin),
  //Output
  .SD_OUT(sd_out), .mean_out(mean),
  .dout_0(dout_0), .dout_1(dout_1), .dout_2(dout_2), .dout_3(dout_3),
  .dout_4(dout_4), .dout_5(dout_5), .dout_6(dout_6), .dout_7(dout_7),
  .dout_8(dout_8), .dout_9(dout_9), .dout_10(dout_10), .dout_11(dout_11),
  .dout_12(dout_12), .dout_13(dout_13), .dout_14(dout_14), .dout_15(dout_15),
  .dout_16(dout_16), .dout_17(dout_17), .dout_18(dout_18), .dout_19(dout_19),
  .dout_20(dout_20), .dout_21(dout_21), .dout_22(dout_22), .dout_23(dout_23),
  .dout_24(dout_24), .dout_25(dout_25), .dout_26(dout_26), .dout_27(dout_27),
  .dout_28(dout_28), .dout_29(dout_29), .dout_30(dout_30), .dout_31(dout_31)
);

CUSUM CUSUM_LAYER
(
  //Input
  .clk(clk), .rst_n(rst_n), .sd_out(sd_out),  .mean(mean),
  .idin_0(dout_0), .idin_1(dout_1), .idin_2(dout_2), .idin_3(dout_3),
  .idin_4(dout_4), .idin_5(dout_5), .idin_6(dout_6), .idin_7(dout_7),
  .idin_8(dout_8), .idin_9(dout_9), .idin_10(dout_10), .idin_11(dout_11),
  .idin_12(dout_12), .idin_13(dout_13), .idin_14(dout_14), .idin_15(dout_15),
  .idin_16(dout_16), .idin_17(dout_17), .idin_18(dout_18), .idin_19(dout_19),
  .idin_20(dout_20), .idin_21(dout_21), .idin_22(dout_22), .idin_23(dout_23),
  .idin_24(dout_24), .idin_25(dout_25), .idin_26(dout_26), .idin_27(dout_27),
  .idin_28(dout_28), .idin_29(dout_29), .idin_30(dout_30), .idin_31(dout_31),
  //Output
  .dout(dout)
);
Endmodule
```

```
module SD_TOP_32
(
  // Input
  clk, rst_n, idin,
   //Output
  SD_OUT, mean_out,
  dout_0, dout_1, dout_2, dout_3, dout_4,
  dout_5, dout_6, dout_7, dout_8,  dout_9,
  dout_10, dout_11, dout_12, dout_13, dout_14,
  dout_15, dout_16, dout_17, dout_18, dout_19,
  dout_20, dout_21, dout_22, dout_23, dout_24,
  dout_25, dout_26, dout_27, dout_28, dout_29,
  dout_30, dout_31
);

input clk, rst_n;
input  [15:0] idin;
output [15:0] SD_OUT;
output [15:0] mean_out;
output[15:0]

dout_0, dout_1, dout_2, dout_3, dout_4,
dout_5, dout_6, dout_7, dout_8, dout_9,
dout_10, dout_11, dout_12, dout_13, dout_14,
dout_15, dout_16, dout_17, dout_18, dout_19,
dout_20, dout_21, dout_22, dout_23, dout_24,
dout_25, dout_26, dout_27, dout_28, dout_29,
dout_30, dout_31;

wire [15:0] SD_OUT;
wire [15:0] mean, d_mean;
wire [15:0]

idin_0, idin_1, idin_2, idin_3, idin_4,
idin_5, idin_6, idin_7, idin_8, idin_9,
idin_10, idin_11, idin_12, idin_13, idin_14,
idin_15, idin_16, idin_17, idin_18, idin_19,
idin_20, idin_21, idin_22, idin_23, idin_24,
idin_25, idin_26, idin_27, idin_28, idin_29,
idin_30, idin_31;

wire [15:0]
d_idin_0, d_idin_1, d_idin_2, d_idin_3,
d_idin_4, d_idin_5, d_idin_6, d_idin_7,
d_idin_8, d_idin_9, d_idin_10, d_idin_11,
d_idin_12, d_idin_13, d_idin_14, d_idin_15,
d_idin_16, d_idin_17, d_idin_18, d_idin_19,
d_idin_20, d_idin_21, d_idin_22, d_idin_23,
d_idin_24, d_idin_25, d_idin_26, d_idin_27,
d_idin_28, d_idin_29, d_idin_30, d_idin_31;
```

```
SP_MEAN_16 SP_MEAN_16_LAYER
(
  //Input
  .clk(clk), .rst_n(rst_n), .idin(idin),
  //Output
  .mean(mean),
  .dout_0(idin_0), .dout_1(idin_1), .dout_2(idin_2),
  .dout_3(idin_3), .dout_4(idin_4), .dout_5(idin_5),
  .dout_6(idin_6), .dout_7(idin_7), .dout_8(idin_8),
  .dout_9(idin_9),   .dout_10(idin_10), .dout_11(idin_11),
  .dout_12(idin_12), .dout_13(idin_13), .dout_14(idin_14),
  .dout_15(idin_15), .dout_16(idin_16), .dout_17(idin_17),
  .dout_18(idin_18), .dout_19(idin_19), .dout_20(idin_20),
  .dout_21(idin_21), .dout_22(idin_22), .dout_23(idin_23),
  .dout_24(idin_24), .dout_25(idin_25), .dout_26(idin_26),
  .dout_27(idin_27), .dout_28(idin_28), .dout_29(idin_29),
  .dout_30(idin_30), .dout_31(idin_31),

  .d_mean(d_mean),
  .d_dout_0(d_idin_0), .d_dout_1(d_idin_1), .d_dout_2(d_idin_2),
  .d_dout_3(d_idin_3), .d_dout_4(d_idin_4), .d_dout_5(d_idin_5),
  .d_dout_6(d_idin_6), .d_dout_7(d_idin_7), .d_dout_8(d_idin_8),
  .d_dout_9(d_idin_9),   .d_dout_10(d_idin_10), .d_dout_11(d_idin_11),
  .d_dout_12(d_idin_12), .d_dout_13(d_idin_13), .d_dout_14(d_idin_14),
  .d_dout_15(d_idin_15), .d_dout_16(d_idin_16), .d_dout_17(d_idin_17),
  .d_dout_18(d_idin_18), .d_dout_19(d_idin_19), .d_dout_20(d_idin_20),
  .d_dout_21(d_idin_21), .d_dout_22(d_idin_22), .d_dout_23(d_idin_23),
  .d_dout_24(d_idin_24), .d_dout_25(d_idin_25), .d_dout_26(d_idin_26),
  .d_dout_27(d_idin_27), .d_dout_28(d_idin_28), .d_dout_29(d_idin_29),
  .d_dout_30(d_idin_30), .d_dout_31(d_idin_31)
);

DELAY DELAY_LAYER
(
  //Input  Port
  .clk(clk), .rst_n(rst_n), .d_mean(d_mean),
  .d_idin_0(d_idin_0), .d_idin_1(d_idin_1), .d_idin_2(d_idin_2),
  .d_idin_3(d_idin_3), .d_idin_4(d_idin_4), .d_idin_5(d_idin_5),
  .d_idin_6(d_idin_6), .d_idin_7(d_idin_7), .d_idin_8(d_idin_8),
  .d_idin_9(d_idin_9),   .d_idin_10(d_idin_10), .d_idin_11(d_idin_11),
  .d_idin_12(d_idin_12), .d_idin_13(d_idin_13), .d_idin_14(d_idin_14),
  .d_idin_15(d_idin_15), .d_idin_16(d_idin_16), .d_idin_17(d_idin_17),
  .d_idin_18(d_idin_18), .d_idin_19(d_idin_19), .d_idin_20(d_idin_20),
  .d_idin_21(d_idin_21), .d_idin_22(d_idin_22), .d_idin_23(d_idin_23),
  .d_idin_24(d_idin_24), .d_idin_25(d_idin_25), .d_idin_26(d_idin_26),
  .d_idin_27(d_idin_27), .d_idin_28(d_idin_28), .d_idin_29(d_idin_29),
  .d_idin_30(d_idin_30), .d_idin_31(d_idin_31),

  //Output Port
  .mean_out(mean_out),
```

```
  .dout_0(dout_0), .dout_1(dout_1), .dout_2(dout_2),
  .dout_3(dout_3), .dout_4(dout_4), .dout_5(dout_5),
  .dout_6(dout_6), .dout_7(dout_7), .dout_8(dout_8),
  .dout_9(dout_9),   .dout_10(dout_10), .dout_11(dout_11),
  .dout_12(dout_12), .dout_13(dout_13), .dout_14(dout_14),
  .dout_15(dout_15), .dout_16(dout_16), .dout_17(dout_17),
  .dout_18(dout_18), .dout_19(dout_19), .dout_20(dout_20),
  .dout_21(dout_21), .dout_22(dout_22), .dout_23(dout_23),
  .dout_24(dout_24), .dout_25(dout_25), .dout_26(dout_26),
  .dout_27(dout_27), .dout_28(dout_28), .dout_29(dout_29),
  .dout_30(dout_30), .dout_31(dout_31)
);

ALT_MUTI_ADD_SQRT ALT_MUTI_ADD_SQRT_LAYER
(
  //Input
  .clk(clk), .rst_n(rst_n), .mean(mean),
  .idin_0(idin_0), .idin_1(idin_1), .idin_2(idin_2),
  .idin_3(idin_3), .idin_4(idin_4), .idin_5(idin_5),
  .idin_6(idin_6), .idin_7(idin_7),.idin_8(idin_8),
  .idin_9(idin_9),   .idin_10(idin_10), .idin_11(idin_11),
  .idin_12(idin_12), .idin_13(idin_13), .idin_14(idin_14),
  .idin_15(idin_15), .idin_16(idin_16), .idin_17(idin_17),
  .idin_18(idin_18), .idin_19(idin_19),.idin_20(idin_20),
  .idin_21(idin_21), .idin_22(idin_22), .idin_23(idin_23),
  .idin_24(idin_24), .idin_25(idin_25), .idin_26(idin_26),
  .idin_27(idin_27), .idin_28(idin_28), .idin_29(idin_29),
  .idin_30(idin_30), .idin_31(idin_31),

  //Output
  .SD_OUT(SD_OUT)
);


endmodule
```

Appendix D Thesis on CD-ROM

The CD-ROM attached to the back provides all the FPGA programme, testbench and verifications.

Appendix E Journal Paper

The paper has been published in:

Hamid GholamHosseini and Kang Li, *Implementation of Transient Signal Detection Algorithm on FPGA.* International Journal of Computer Applications (0975 - 8887) Volume 41 –No.12, March 2012

List of References

1. G. Soudlenkov, S. Kitave, *Two-staged Algorithm for Dispersed Transient Radio Emission Detection*, in *Engineering*. 2010, Auckland University of Technology: Auckland. pp. 13.

2. P.A.Fridman (2010). "A method of detecting radio transients." Astronomy & Astrophysics 409(2). pp.808-820.

3. *Transient Radio Emission Array Detector*. 2011 [Retrived 2011 March 10]; Available from: http://wiki.karen.net.nz/index.php/Transient_Radio_Emission_Array_Detector.

4. Wikipedia. *Cosmic Ray*. 2011 [Retrived 2011 April 15]; Available from: http://en.wikipedia.org/wiki/Cosmic_ray.

5. M. D. Potgieter, M. S. Potgieter, *Cosmic Ray Anisotropies in the Outer Heliosphere.* Advances in Space Research, 2007.

6. N. Edwin, A. Paul, *High Performance Evolutionary Computing*, in *US Army Space and Missle Defense Command (SMDC), Redstone Arsenal, AL*. 2006, HPCMP Users Group Conference, 2006. pp. 6.

7. J. Curreri, S. Koehler, B. Holland, *Performance Analysis with High-Level Languages for High Performance Reconfigurable Computing*, in *Field-Programmable Custom Computing Machines*. pp. 7.

8. M. Molla, M. Taylor. (2010). Towards a Unified Source-Propagation Model of Cosmic Rays. International Conference of the Hellenic Astronomical Society. 424: 98.

9. G. Newby, *Hardware Acceleration Prospects and Challenges for High Performance Computing*, in *Computer Systems and Applications, IEEE/ACS International Conference*. 2009. pp. 4.

10. J. P. Song, D. Shires, *Reconfigurable Computing for High Performance Computing Computational Science*, in *DoD High Performance Computing Modernization Program Users Group Conference*. 2008. pp. 9.

11. L. Cornel, Q. Andre, *Transient Signal Detection Using Overcomplete Wavelet Transform and High-Order Statistics.* Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03), 2003. 6: VI - 449-52 vol.6, pp. 16.

12. A. Theolis, *Computational Signal Processing with Wavelets*. 1998: Birkhauser, Boston.

13. M. Grochnig, *Irregular Sampling of Wavelet and Short Time Fourier Transforms.* 1993(Ronald A. DeVore.).

14. E. Sousa, A. Ghasemi, *Collaborative Spectrum Sensing for Opportusitic Access in Fading Environments*, in *New Frontiers in Dynamic Spectrum Access Networks*. 2005. pp. 131 – 136.

15. J. Melvin Hinich, *Bispectral Based Tests for the Detection of Gaussianity and Linearity in Time Series.* Journal of American Statistical Association, 1988. pp. 657-664.

16. L. Perreaulta, J. Berniera, B. Bobéeb, E. Parent, *Bayesian Change-point Analysis In Hydrometeorological Time Series.* Journal of Hydrology, 2000. 235(3-4): pp. 242-263.

17. Wikipedia. *Mann–Whitney U*. 2011 [Retrived 201 May 15]; Available from: http://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U.

18. Wikipedia. *Wilcoxon signed-rank test*. 2011 [Retrived 2011 May 20]; Available from: http://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test.

19. M. Roveri, A. Cesare, *An adaptive CUSUM-based test for signal change detection* in *Circuits and Systems, 2006. ISCAS 2006. Proceedings*. 2006. pp. 4.

20. Wikipedia. *CUSUM*. 2011 [Retrived 2011 May 11]; Available from: http://en.wikipedia.org/wiki/CUSUM.

21. Wikipedia. *Standard deviation*. 2011 [Retrived 2011 Aug 10]; Available from: http://en.wikipedia.org/wiki/Standard_deviation.

22. Wikipedia. *Normal Distribution*. 2011 [Retrived 2011 Aug 10]; Available from: http://en.wikipedia.org/wiki/Normal_distribution.

23. K. Ashenayi, S. Singh, I. Hoballah, *Application of Normal Distribution In Modeling Global Irradiation* 1988, The University of Tulsa & The University of Wisconsin-Milwaukee: Wisconsin. pp. 470 – 474.

24. S. Kun, Z. Yin, H. Xing, *Fundamental of Cumulative Sum Method and its Application in Measurement Data Processing*, in *Dept of Measurement and Control*. 2000, He Fei University of Technology: He Fei.

25. H. Aghajan, Y. Pati, T. Kailath, *Transient Signal Detection Using High Resolution Line Detection on Wavelet Transforms*, in *Signals, Systems and Computers*. 1994 USA. pp. 1114 - 1118.

26. A. Jayaprakasam, V. Sharma, *Cooperative Robust Sequential Detection Algorithms for Spectrum Sensing in Cognitive Radio.* Ultra Modern Telecommunications & Workshops, 2009: pp. 1 - 8.

27. S. Zarrin, T. Joon Lim, *Cooperative Quickest Spectrum Sensing in Cognitive Radio with Unknow Prameters*, in *Dept. of Electr. & Comput. Eng*. 2009, Univ. of Toronto: Canada, pp. 6.

28. J. Hall, M. Barbeau, E. Kranakis, *Detection Of Transient In Radio Frequency Fingerprinting Using Signal Phase* in *IASTED International Conference on Wireless and Optical Communications (WOC)*. 2003: Banff, Alberta, Canada.

29. H. Dong, L. Yong, *Standard Deviation & Standard Error*. China Academic Journal Electronic Publishing House, 2005. 17: pp. 2.

30. *Altera Corporation*. 2011 [Retrived 2011 Aug 25]; Available from: http://www.altera.com.

31. *Xilinx*. 2011 [cited 2011 Aug 25]; Available from: http://www.xilinx.com.

32. *Altera and Xilinx Report*. [Retrived 2011 Aug 25]; Available from: http://seekingalpha.com/article/85478-altera-and-xilinx-report-the-battle-continues.

33. *Lattice Semiconductor Corporation*. 2011 [Retrived 2011 Aug 25]; Available from: http://www.latticesemi.com/.

34. *Actel Corporation*. 2011 [Retrived 2011 Aug 25]; Available from: http://www.actel.com.

35. *SiliconBlue Technologies*. 2011 [Retrived 2011 Aug 25]; Available from: http://www.siliconbluetech.com/.

36. *Achronix Semiconductor Corporation*. 2011 [Retrived 2011 Aug 25]; Available from: http://www.achronix.com/.

37. *QuickLogic Corporation*. 2011 [Retrived 2011 Aug 25]; Available from: http://www.quicklogic.com/.

38. *Alera About Us*. 2011 [Retrived 2011 Aug 25]; Available from: http://www.altera.com/corporate/about_us/abt-index.html.

39. *Altera Corporation*. 2011 [Retrived 2011 Aug 25]; Available from: http://www.altera.com/end-markets/end-index.html.

40. *Altera Corporation*. 2011 [Retrived 2011 Aug 25]; Available from: http://www.altera.com.

41. *Altera Corporation*. [Retrived 2011 Aug 25]; Available from: http://www.altera.com/literature/ds/ds_nios2_perf.pdf.

42. *Altera Arithmetic Megafunctions*. [Retrived 2011 Aug 25]; Available from: http://www.altera.com/literature/ug/ug_lpm_alt_mfug.pdf?GSA_pos=1&WT.oss_r=1&WT.oss=Arithmetic Megafunctions.

43. S. Kilts, *Advanced FPGA Design: Architecture, Implementation*. 2007, Hoboken.

44. W. Xu Hua, W. Chen, *Altera FPGA/CPLD Design (Advance)*. 2006, Bei Jing: Posts Telecom Press.

45. W. Shu Hong, T. Kun, *Efficient Utilization of Scratch-pad Memory Banks*. 2005, Tsinghua Unversity: Bei Jing. pp. 4.

46. Wikipedia. *Signed_number_representations*. 2011 [Retrived 2011 Aug]; 26]. Available from: http://en.wikipedia.org/wiki/Signed_number_representations.

47. X. Yu Wen, Verilog Digital System Design. 2008, Bei Jing: Bei Jing Aviation University. pp.477.

48. Altera. System Verilog. 2011 [Retrived 2011 20 Nov]; Available from: http://www.altera.com/education/training/courses/OHDL1125.

49. Altera. *White Paper: Stratix III Programmable Power*. 2011 [Retrived 2011 11, Nov]; Available from: http://www.altera.com/literature/wp/wp-01006.pdf?GSA_pos=1&WT.oss_r=1&WT.oss=Stratix III Programmable Power.

50. Altera. *Decrease Total System Costs with Industry's Lowest Cost,Lowest Power FPGA*. 2011 [Retrived 2011 11, Nov]; Available from: http://www.altera.com/literature/wp/wp-01113-lowest-system-cost.pdf.

51. Altera. *Cyclone IV Programmable Power Technology*. 2011 [Retrived 2011 11, Nov]; Available from: http://www.altera.com/devices/fpga/cyclone-iv/overview/power/cyivpower.html?GSA_pos=14&WT.oss_r=1&WT.oss=cycloneIV programmable power.

52. Altera. *Cyclone IV FPGAs: Optimized for Lowest Power*. 2011 [Retrived 2011 11, Nov]; Available from: http://www.altera.com/support/software/power/powerplay_faq.pdf.

53. M. Allen, B. Wilkinson, *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*. 2006.

54. *Actel Corporation*. *Power_Comparison_WP*, 2011 [Retrived 2011 Aug 25]; Available from: http://www.actel.com.