**Chapter 1**

**INTRODUCTION**

**1.1 Introduction**

Fault tolerance plays a very important role in modern systems where immediate human intervention is not possible and system failure can have disastrous consequences. A fault tolerant system has the ability to detect and then correct the occurrence of a hardware failure. In order to detect the fault, the system must be able to sense any deviations from its normal operation. A fully fault tolerant system also has the ability to correct the fault in order to return the system to its normal functionality.

An optimal design will minimize the amount of extra logic required to detect and then correct the occurrence of the fault. An extreme temperature change is one of the reasons in which fault tolerance is necessary for devices operating in harsh operating environments, as found, for example, in space and military applications. Fault tolerance will also be necessary in nanoelectronic systems, as small device dimensions make the system more susceptible to outside interference, such as cosmic radiation.

This Report will study methods to implement fault tolerant arithmetic circuits on Field Programmable Gate Arrays (FPGAs). FPGAs are integrated circuits (ICs) that can be configured to implement a specific function after the chip has been manufactured. The first FPGAs were put on the market in 1985 and by the late 1990’s FPGAs were becoming more popular than Application Specific Integrated Circuits (ASICs).

The advantages of using FPGAs are their reprogrammable nature, ease of prototyping, rapid time to market, and minimal non-recurring engineering (NRE) cost compared to custom IC designs. Most modern electronic systems contain some high performance digital chips known as Digital Signal Processors (DSPs). DSP designs are commonly used in electronic systems such as avionics, communication systems and also in portable electronics. DSP chips transform and manipulate digitally encoded signals according to some specified system design goal. Various algorithms such as the Fast Fourier Transforms are used to analyze the signals which are in digital form. The main components of a DSP chip are the adder and multiplier along with memory elements.

The performance of the system is based on the speed at which arithmetic operations are performed. Hence the adder plays a vital role in DSPs for carrying out all the required arithmetic operations.

**1.2 Review of the Relevant Literature**

Previous studies have investigated fault tolerant methods implemented on FPGA. The basic fault tolerant approach is Triple Modular Redundancy (TMR) which is used as a point of reference to compare with advanced fault approach considered in this project. TMR is a common solution for hardening digital logic against SEUs and is widely adopted in ASIC designs. Hardware is essentially replicated in triplicate with a voter circuit used to pass the majority rule signals to the output.

Roving fault detection and graceful degradation are some of the fault tolerant approaches used for ensuring reliable FPGA designs. Roving fault detection performs a progressive scan of an FPGA structure by swapping blocks with the same functionality with a block carrying out the test function. Graceful degradation is an approach in which the faulty block is replaced with a spare block. A fault correction approach for a parallel prefix, N bit Kogge-Stone adder, which consists of two independent N/2 bit Han-Carlson (HC) adders, is implemented. In addition, fault tolerance can be implemented by using self-testing areas (STARs) on an FPGA, which allows fault checking to occur without disturbing the normal system operation.

**1.3 Project Objectives**

As the adder plays a vital role in digital signal processing systems and microprocessors, the main objective of this research is to design and implement fault tolerant adders on FPGAs. Existing fault tolerant approaches will be applied to adders of varying bit widths for implementation on FPGAs.

**1.4 Project Method**

To meet the project objectives three adder topologies, namely the ripple carry adder, Kogge-Stone adder, and the sparse Kogge-Stone adder are studied. The ripple carry adder in a TMR configuration is used as the reference design. The Kogge-Stone adder, which is classified as a parallel prefix adder, has a critical path on the order of (where is the width of the adder in bits). The regularity of its structure makes it suitable for VLSI designs as well as FPGA implementations.

This project is performed in two parts. First is to evaluate the previous work of K. Roy’s group on fault tolerant adders meant for VLSI design and by implementing this approach on FPGAs. Second is to investigate and design fully fault tolerant Kogge-Stone adders on FPGAs. The fault tolerant adders which are coded in verilog are synthesized and implemented on the FPGAs.

The designs are synthesized by coding with Verilog using Altera’s Quartus II software. The performance metrics of timing delay and operational cost are observed from the synthesis reports. The functionality of the designed adders are verified and simulating with ModelSim-Altera.

**1.5 Report Outline**

An outline of the report is as follows. Chapter 2 describes the background work on the fault tolerance in FPGAs. It discusses about the adder working. Chapter 3 descries the flow of entities, and how one entity is linked to other. It also gives a brief description of different entities. Chapter 4 gives the various simulation results along with flowchart of various entities. Chapter 5 describes the software and hardware used in the project. Chapter 6 describe the FPGA implementation along with its architecture and design flow. Chapter 7 gives the experimental set up for the conduction of the project. Following chapters deal with the advantages, conclusion and future work.

**Chapter 2**

**KOGGE-STONE ADDER**

**2.1 Introduction**

An adder plays a vital role in many digital circuit designs including Digital Signal Processors (DSPs) and microprocessors. The fault tolerant techniques for high speed adder designs are considered in this chapter. Triple Modular Redundancy (TMR), which is a common fault tolerance approach, is used as the base reference design. This chapter describes the design of the ripple carry adder, parallel prefix adders and then the fault tolerant concepts that can be applied to these designs.

**2.2 Basic Adder Designs**

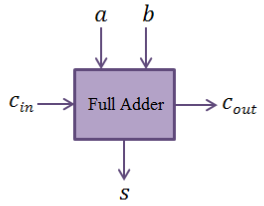
This section describes the basic adders used in this thesis. The adders implemented on FPGAs are the ripple carry adder, Kogge-Stone adder, and sparse Kogge-Stone adders. The ripple carry adder is one of the simplest adder designs. The Kogge-Stone adder is an example of a parallel prefix adder. The internal blocks used in the adder designs are described in detail in this section.

**2.2.1 Full Adder**

A full adder is a circuit which adds three one bit binary numbers and outputs two one bit binary numbers. The block diagram is shown in Figure 2.1. Here, a and b are the two adder inputs and cin is the carry input. The two outputs produced are the sum s and carry cout. Table 2.1 depicts the truth table of the full adder. The following are the Boolean expressions for the full adder.

*s = a ⊕b ⊕ cin*(2.1)

*cout = a ∙ b + b ∙ cin+ a ∙ cin = a ∙ b + ( a ⊕b ) ∙ cin*

**

**Figure 2.1:** Block diagram of a full adder

For prefix adders, it is convenient to define the intermediate signals generate, propagate, and delete given by *g, p,* and *d,* respectively,

*g = a ∙ b* (2.2)

*p = a ⊕b*

*d =*

The sum and carry out are then given by,

*s = p⊕cin* (2.3)

*cout = g + p ∙ cin*

**Table 2.1.**Truth table of a full adder

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| a | B | cin | s | cout | Carry Status |
| 0 | 0 | 0 | 0 | 0 | delete |
| 0 | 0 | 1 | 1 | 0 | delete |
| 0 | 1 | 0 | 1 | 0 | propagate |
| 0 | 1 | 1 | 0 | 1 | propagate |
| 1 | 0 | 0 | 1 | 0 | propagate |
| 1 | 0 | 1 | 0 | 1 | propagate |
| 1 | 1 | 0 | 0 | 1 | generate |
| 1 | 1 | 1 | 1 | 1 | generate |

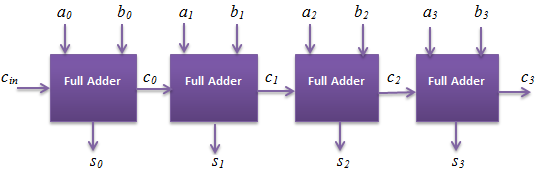
**2.3 Ripple Carry Adder**

The ripple carry adder is one of the simplest adders. It consists of a cascaded series of full adders. For example, a 4-bit adder can be constructed by cascading four full adders together as shown in Figure 2.2. The ripple carry adder is relatively slow as each full adder must wait for the carry bit to be calculated from the previous full adder.

The worst case delay of a ripple carry adder occurs when cin propagates from the first stage to the most significant bit position. The delay for an N-bit adder is given by,

*tadder = (N-1)tcarry + tsum* (2.4)

where, is the carry propagation delay for one stage and is the time required to compute the sum bit for one stage. Hence, the delay of the ripple carry adder is of order N.

**

**Figure 2.2:** 4-bit ripple carry adder

**2.4 Kogge-Stone Adder**

The Kogge-Stone adder is classified as a parallel prefix adder since the generate and the propagate signals are precomputed. In a tree-based adder, carries are generated in tree and fast computation is obtained at the expense of increased area and power. The main advantage of this design is that the carry tree reduces the logic depth of the adder by essentially generating the carries in parallel. The parallel-prefix adder becomes more favorable in terms of speed due to the O(log2*n*) delay through the carry path compared to O(*n*) for the RCA. The Kogge-Stone adder is widely used in high-performance 32-bit, 64-bit, and 128-bit adders as it reduces the critical path to a great extent compared to the ripple carry adder.

The operation of the tree-based adder can be understood using the concept of the fundamental carry operation (*fco*). This operator works on the generate and propagate pairs as defined by,

*(gL,pL­) o (gR,pR) = (gL + pL ∙ gR,pL ∙ pR)* (2.5)

wheregL, pL are the left input generate and propagate pairs and gR, pR are the right input generate and propagate pairs to the cell. For example, in a 4-bit carry lookahead adder, the carry combination equation can be expressed as,

*c4 =(g4,p4­) o [(g3,p3) o [(g2,p2­) o (g1,p1)]]* (2.6)

*= (g4,p4­) o [(g3,p3) o [(g2 + p2­ ∙ g1,p2 ∙ p1)]]*

:

:

*= g4 + p4 ∙ g3  + p4 ∙ p3 ∙ g2 + p4 ∙ p3 ∙ p2 ∙ g1*

Since the *fco* obeys the associativity property, the expression can be reordered to yield parallel computations in a tree based structure,

*c4 = [(g4,p4­) o(g3,p3)] o [(g2,p2­) o (g1,p1)]* (2.7)

**2.4.1 8-bit Kogge-Stone Adder**

The 8-bit Kogge stone adder will be explained in detail in this subsection. An 8-bit Kogge-Stone adder is built from eight generate and propagate (GP) blocks, eight black cells (BC) blocks, eight gray cell (GC) blocks, and nine sum blocks as shown in the Figure 2.3. The details of the various blocks used in the structure of Kogge-Stone adder are discussed below.

**1) GP block**

The generate and propagate block takes a pair of operand bits *(a, b)* as inputs and computes a pair of generate and propagate signals *(g, p)* as output, as depicted in Figure 2.3. The output from the GP block is given by the equation (2.2).

**

**Figure 2.3:** Generate-Propagate block

**2) BC block**

The black cell takes two pairs of generate and propagate signals *(gi, pi)* and*(gj, pj)*as input and computes a pair of generate and propagate signals *(g, p)*as output. It is shown in the Figure 2.4(a).

The expressions for the output signals *g, p* generated by the black cell are given by

*g = gi + pi ∙ gj* (2.8)

*p = pi ∙ pj*

**3) GC block**

The gray cell takes two pairs of generate and propagate signals *(gi, pi)* and *(gj, pj)* as inputs and computes a generate signal *g* as output which is shown in Figure 2.4(b).

The expressions for the output signal *g* obtained by the gray cell is given a

*g = gi + pi ∙ gj*  (2.9)

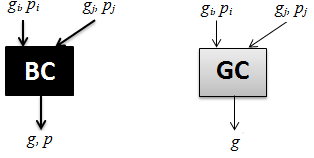
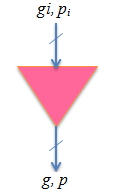
**

Figure 2.4 (a): Black cell Figure 2.4 (b): Gray cell

**4) Buffer**

The buffer takes a pair of the generate and propagate signals *(pi, gi)* as input and passes the same signals to the output. It is shown in Figure 2.5.



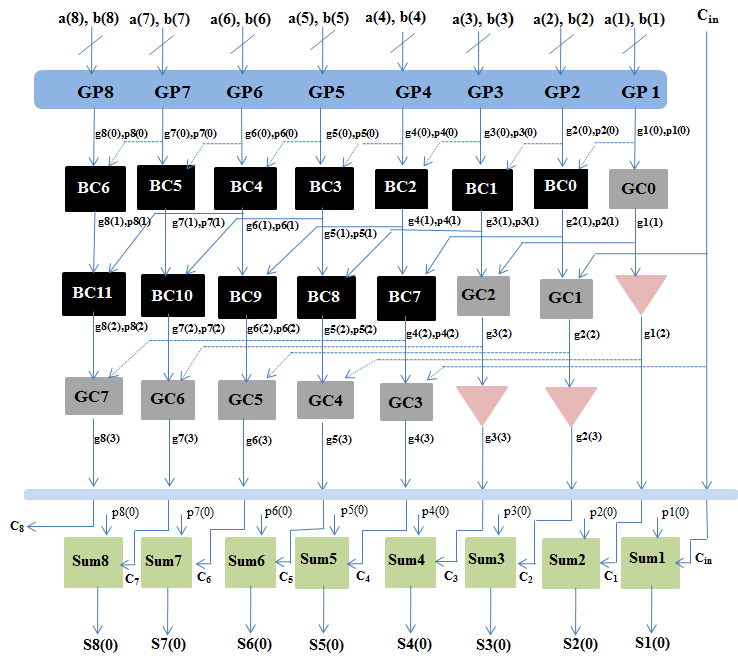
**Figure 2.5:** Buffer

The expressions for the output signals *g, p* obtained by the buffer block are given as

*g = gi* (2.10)

*p =pi*

The complete schematic for the 8-bit Kogge-Stone adder is shown in Figure 2.6. An 8-bit Kogge-Stone adder is built from eight generate and propagate (GP) blocks, twelve black cell (BC) blocks, eight gray cell (GC) blocks, and eight sum blocks. To be aesthetic, an extra column has been added in our design to show the computation of c8. In practice, c8 is generated by just adding an extra gray cell in the last column.



**Figure 2.6:** 8-bit Kogge-Stone adder

**2.4.2 Higher Order Kogge-Stone Adders**

Table 2.6.summarizes the various types of cells required for the Kogge-Stone adders with larger bit widths.

****

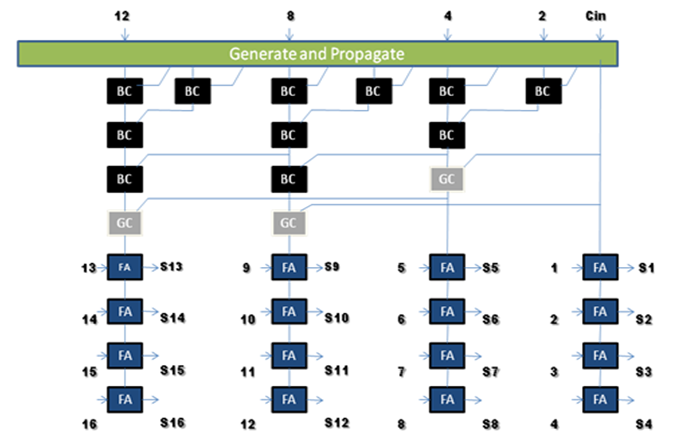
**Figure 2.7:** 16-bit Kogge-Stone Adder**.**

**Table 2.2:** Kogge-Stone adders of different bit widths

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bit Width of Kogge-Stone Adder | No. of GP Blocks | No. of Black Cells | No. of Gray Cells | No. of Sum blocks |
| 16-bit | 16 | 37 | 16 | 16 |
| 64-bit | 64 | 257 | 64 | 64 |
| 128-bit | 128 | 641 | 128 | 128 |
| 256-bit | 256 | 1537 | 256 | 256 |

**2.5 Sparse Kogge-Stone Adder**

The sparse Kogge-Stone adder consists of several smaller ripple carry adders (RCAs) on its lower half and a carry tree on its upper half. Thus, the sparse Kogge-Stone adder terminates with RCAs. The number of carries generated is less in a sparse Kogge-Stone adder compared to the regular Kogge-Stone adder. The functionality of the GP block, black cell and the gray cell remains exactly the same as in the regular Kogge-Stone adder. The schematic for a 16-bit sparse Kogge-Stone adder is shown in Figure 2.8. Sparse and regular Kogge-Stone adders have essentially the same delay when implemented on an FPGA although the former utilizes much less resources.

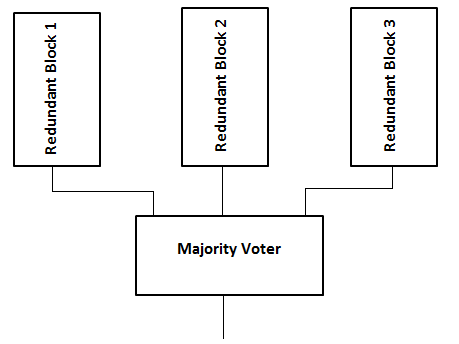


**Figure 2.8:** Sparse Kogge-Stone adder.

**2.6. Basic Fault Tolerance - Hardware Redundancy**

Basic fault tolerance can be achieved by N-module redundancy (NMR) where N refers to the degree of redundancy used in the design. This approach is easy to apply but results in high area overhead. For example, Triple Modular Redundancy (TMR) is a fault tolerant method where the hardware is essentially replicated in triplicate with a voter circuit used to pass the majority rule signals to the output. TMR is one of the most common methods used to create fault tolerant designs in both ASIC and FPGA implementations.

The general TMR is shown in Figure 2.9. Three copies of the same circuit are connected to a majority voter which is used to obtain the fault free output. This method works as long as all the faults are confined to one of the redundant blocks. The latency will be increased because of the voter in the circuit’s critical path. The triple modular redundant ripple carry adder (TMR-RCA) is used as the reference design for this thesis. This adder is the simplest approach for both detecting and correcting faults. The block diagram of the TMR adder circuit using the ripple carry adders is shown in Figure 2.10.



**Figure 2.9:** General Triple Modular Redundancy



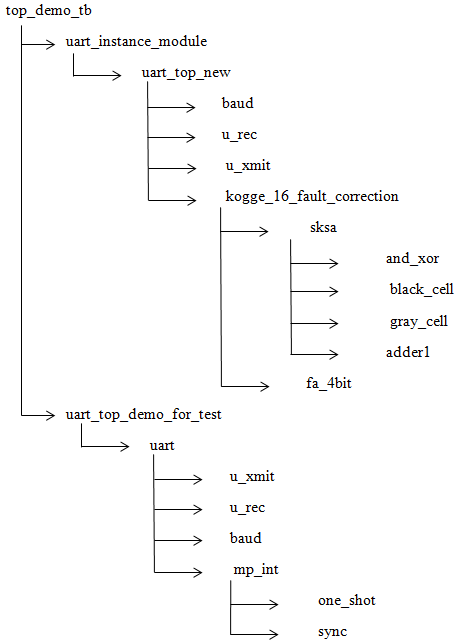
**Figure 2.10:** TMR adder circuit using ripple carry

**Chapter 3**

**DESIGN AND IMPLEMENTATION**

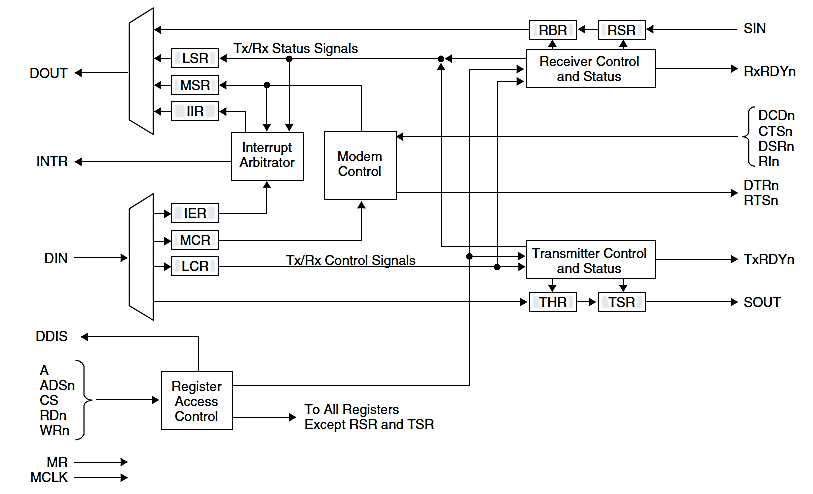
The fault tolerant Kogge-Stone adder consists of different blocks which are put together for its implementation. Different sub-programs have been put together to form a complete program. Herein we will see how the programs have been arranged and linked together.

**3.1 Structure of Programs**

****

**Figure 3.1**: Flow of Programs

For the stimulations output, i.e. on Modelsim-Altera top\_demo\_tb is the top level module, under which several other modules are attached. For the FPGA implementations uart\_instance\_module is the top level module. top\_demo\_tb is the test bench code for the uart\_instanc\_module. These top level module includes the files for UART (Universal Asynchronous Receiver/Transmitter) and fault tolerant Kogge-Stone adder.



**Figure 3.2**: UART Block Diagram

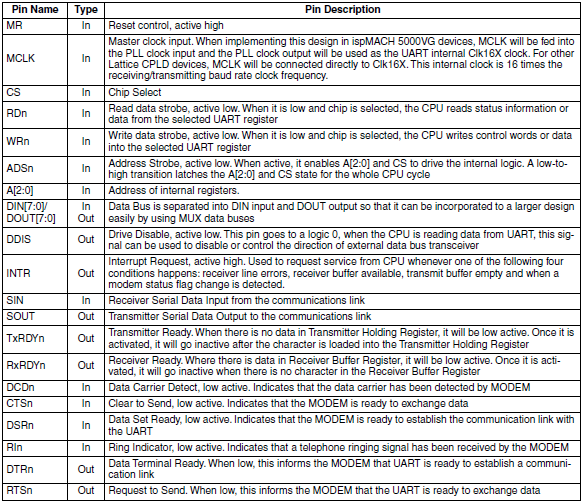
UART consists of the receiver and transmitter. There are also the files of baud and mp\_int.

**2.1.1 baud**

This is the "baud-rate-genrator". The "baud\_clk" is the output clock feeding the receiver and transmitter modules of the UART. By design, the purpose of the "baud\_clk" is to take in the "sys\_clk" and generate a clock which is 16 x BaudRate, where BaudRate is the desired UART baud rate.

**2.1.2 mp\_int**

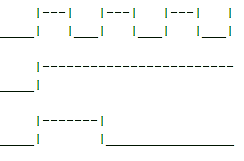
It is Microprocessor Interface module. This module provides the interface between the uart receiver and transmitter and the microprocessor bus.mp\_int calls for two more programs, they are one\_shot and sync.



**Table 3.1**: Pin Description of UART

**2.1.3 one\_shot**

This module generates a one shot pulse whenever an input goes high from low.



**Figure 3.3**: One Shot Pulse.

**2.1.4 sync**

This is a double-rank synchronizer.

**2.1.5 Top level module**

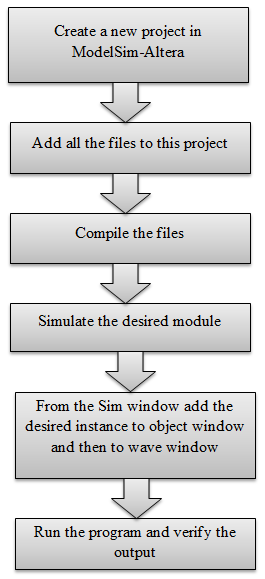
uart\_instance\_module is the top level module. This module calls upon the different files of UART and fault tolerant Kogge-Stone adder. uart\_instance\_modulecalls upon the files of uart\_top\_new, which consists of baud, u\_rec, u\_xmit, and kogge\_16\_fault\_correction; and uart\_top\_demo\_test, which consists of UART files.

**Chapter 4**

**VERIFICATION**

For the simulation in ModelSim-Altera the top level module is top\_demo\_tb, which is the test bench code for the uart\_instance\_module. For stimulations in FPGA uart\_instance\_module is the top level module. in this section we shall look at the verifications of several modules.

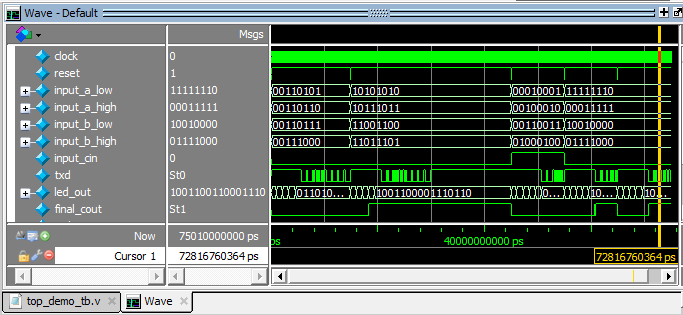
**4.1 ModelSim-Altera**

ModelSim-Altera is used for the verifications using the waveforms. The below figure illustrates how to verify using this software.

**Figure 4.1**: General design flow for simulation.

**4.2 top\_demo\_tb**

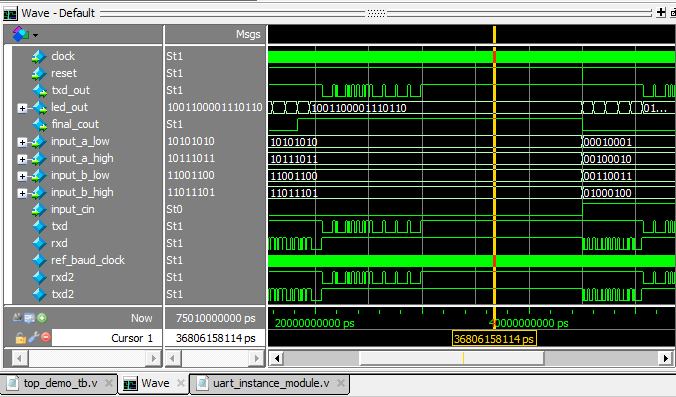
This is the test bench code for the uart\_instance\_module. uart\_instance\_module takes in the inputs a,b, cin, clock and reset. It gives the output led\_out, final\_cout. It also displays the data transmitted and received.



**Figure 4.2**: Simulated output for top\_demo\_tb.v

**4.3 uart\_instance\_module**

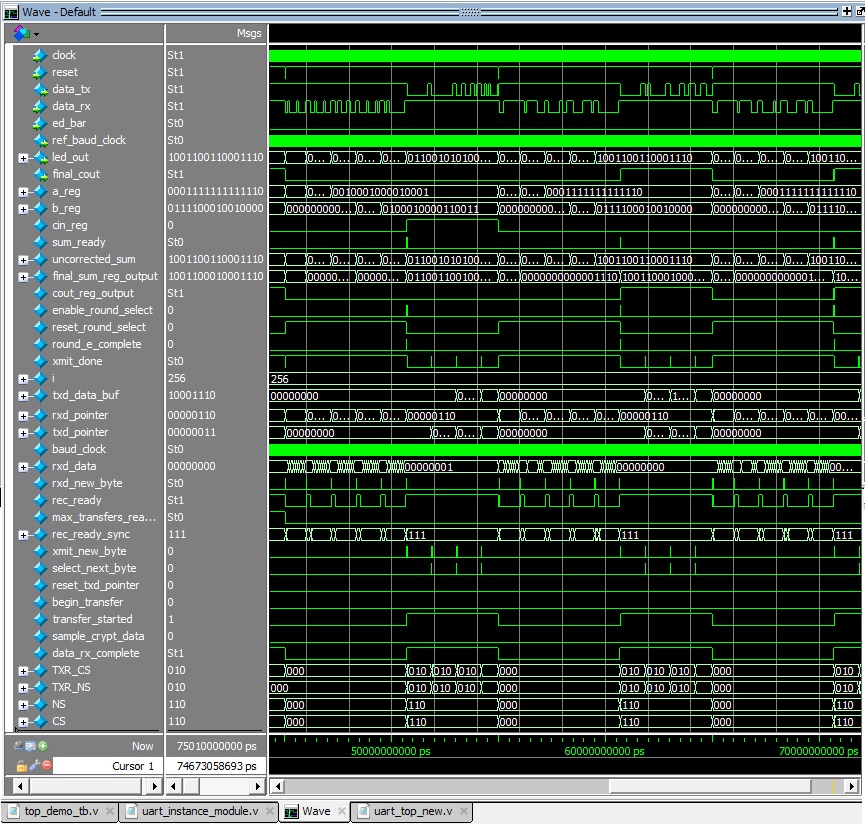
Through this module the port mapping of uart\_top\_new and uart\_top\_demo\_for\_test is made with the input and output ports of uart\_instance\_module, so that data can be effectivelylinked between the between the different modules and reception of data is done.



**Figure 4.3**: Simulated output for uart\_instance\_module.v

**4.4 uart\_top\_new**

This is an important module that links various other modules. It links modules like baud, u\_rec, u\_xmit, and kogge\_16\_fault\_correction. Through this module the speed of the processor of the computer is mapped with the FPGA by means of baud. The data transmitted and received is monitored from here.



**Figure 4.4**: Simulated output for uart\_top\_new.v

Baud is the “baud-rate-generator”. The "baud\_clk" is the output clock feeding the receiver and transmitter modules of the UART. By design, the purpose of the "baud\_clk" is to take in the "sys\_clk" and generate a clock which is 16 x BaudRate, where BaudRate is the desired UART baud rate.

U\_rec is the receiver portion of the UART and u\_xmit is the asynchronous transmitter portion of the UART.

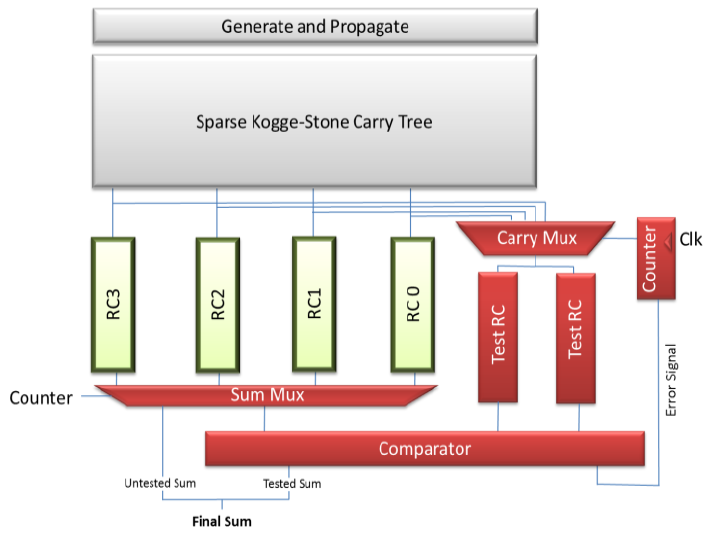
**4.5 uart\_top\_demo\_for\_test**

This module mainly concentrates on the UART. There are several modules attached to it for the proper transfer of the data from the hyper-terminal of the computer to FPGA and from FPGA to hyper-terminal through the UART. It contains modules similar to those in uart\_top\_new, instead of kogge\_ 16\_fault\_correction it contains mp\_int.

Mp\_int is the Microprocessor Interface module. This module provides the interface between the UART receiver and transmitter and the microprocessor bus.

**4.6 kogge\_ 16\_fault\_correction**

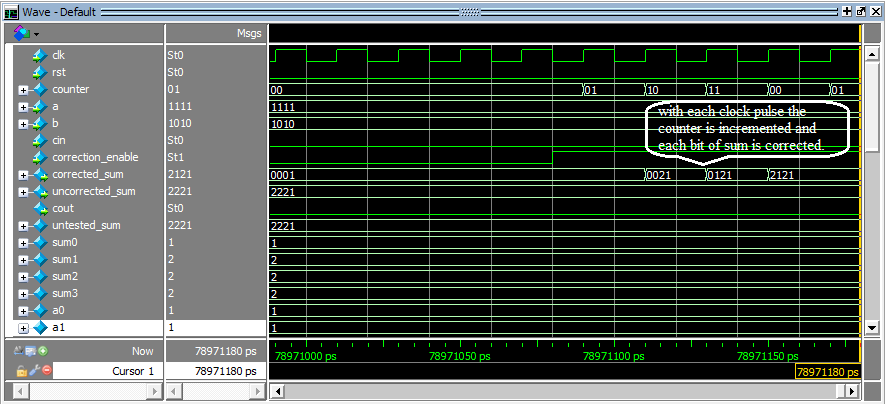
This is the main module which is the core of the project. Through this module we have designed to detect and correct fault in the Sparse Kogge-Stone Adder. The design of Sparse Kogge-Stone is described in further topics. The figure below explains the flow of Sparse Kogge-Stone. Here a and b are taken as 16 bits input and cin is the carry in.



**Figuren4.5**: Block diagram of fault tolerant sparse Kogge-Stone adder

In the above figure a and b are put on the generate and propagate block, which is further put in the Sparse Kogge-Stone Carry tree, which is a part of 16-bit Kogge-Stone Adder to produce the carry at different bits. Cout is generated from this block itself. Below it are 4 4-bit ripple carry adders which produces 4 bit of sum. To detect and correct the error the concept of Triple Mode Redundancy- Ripple Carry Adder is used. Clock is used, which drives the counter. When counter is 0 RC0 is tested with the help of 2 Test RC. RC stands for Ripple Carry Adder. With each clock pulse the counter increments and different RC are tested. By the means of comparator the correct output is selected, with the help of majority voter, and is displayed.

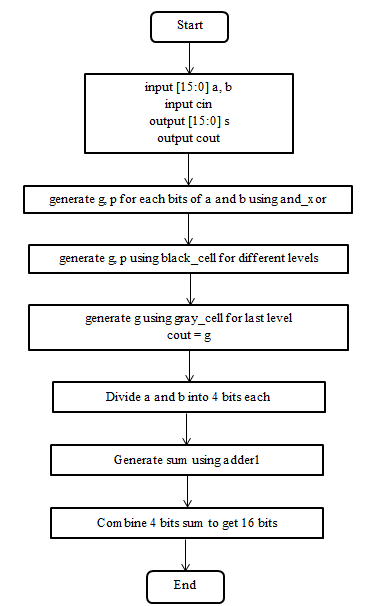
The waveform of fault tolerant sparse Kogge-Stone adder below explains the working.



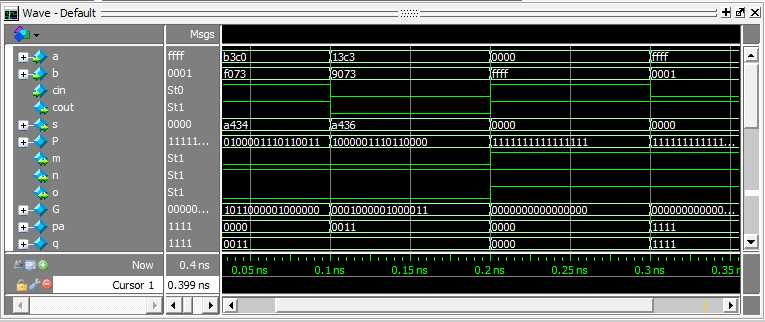
**Figure 4.6**: Waveform of fault tolerant sparse Kogge-Stone adder

**4.7 sksa**

This is the sub module for the kogge\_16\_fault\_correction. Here is the description how sparse Kogge-Stone works. The flowchart below explains the working of the sparse Kogge-Stone. The 16-bit of a and b are used to give the generate and propagate through the generate and propagate block, expressed as and\_xor in the flowchart. Then 8 black\_cells are used to again produce g and p. These black cells work as, for the 1st black cell the previous bits are a0,b0 and the present bits are a1, b1. This level gives g and p for the next level. In the next level there are 4 black\_cells. The further level consists of 1 gray\_cell and 3 black\_cells. Following are 3 gray\_cells, and the last level has a gray cell, which gives cout. Then below is the arrangement of 4 4-bit ripple carry adders, which produces sum.



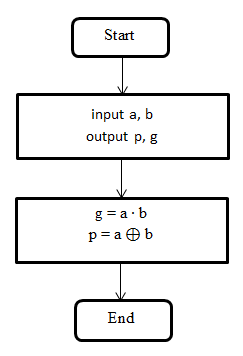
**Figure 4.7**: Flowchart of sksa.v



**Figure 4.8**: Simulated waveform of sksa.v

**4.8 and\_xor**

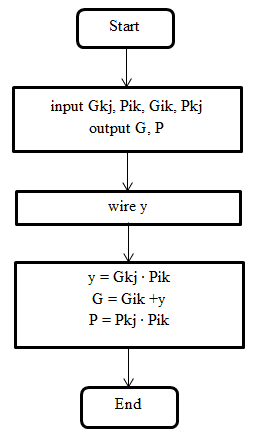
This module is used to produce generate and propagate from the inputs.



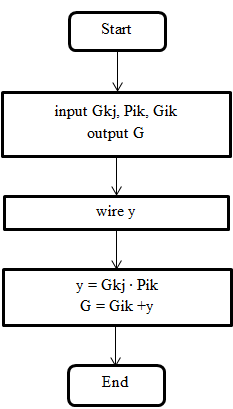
**Figure 4.9**: Flowchart of and\_xor.v

**4.9 black\_cell**

This module is used to produce generate and propagate from the previous and present generate and propagate bits. The below figure shows the flowchart of the black\_cell.Pik, and Gik are the present bits and Gkj, and Pkj are the previous bits. G and P are the final generate and propagate bits.



**Figure 4.10**: Flowchart of black\_cell.v

****

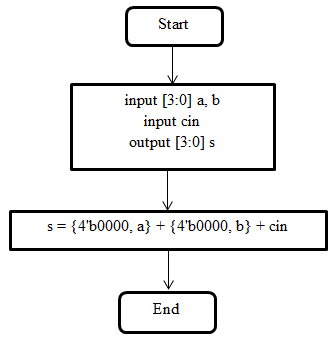
**Figure 4.11**: Flowchart of gray\_cell.v

**4.10 gray\_cell**

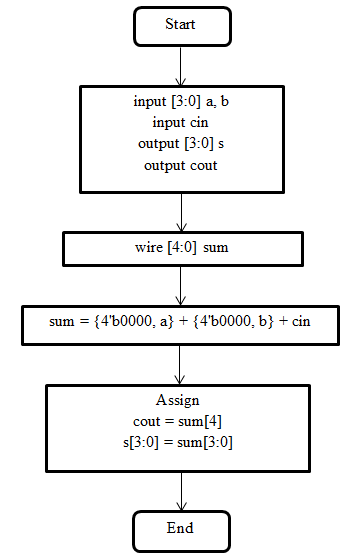
This module is used to produce generate from the previous and present generate and propagate bits. The below aboveshows the flowchart of the gray\_cell.Pik, and Gik are the present bits and Gkj is the previous bit. G is the final generate bits.

**4.11 adder1**

This module is used to generates sum from the inputs, i.e., a and b. It is 4-bt ripple carry adder to generate only the sum. The figure below shows the flowchart of the adder1.



**Figure 4.12**: Flowchart of adder.v



**Figure 4.13**: Flowchart of fa\_4bit.

**4.12 fa\_4bit**

This module is used to generates sum and carry from the inputs, i.e., a and b. It is 4-bit ripple carry adder. The figure above shows the flowchart of the fa\_4bit.

**Chapter 5**

**SOFTWARE AND HARDWARE**

**5.1 Software**

**5.1.1Quartus II**

QUARTUS II is a software tool produced by Altera for analysis and synthesis of HDL designs, which enables developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a designs reaction to different stimuli and configure the target device with the programmer.

The AlteraQuartus II design software provide complete,multiplatform design environment that easily adopts to our design specification needs. It is acomprehensive environment for system–on-a-programmable-chip(SOPC) design. The Quartus II software includes solution for all phases of FPGA and CPLD design.

**5.1.2 ModelSim-Altera software**

Altera provides the Miodelsim –Altera software to simplify design simulation with all readily recompiled Altera simulation libraries. The Altera simulation libraries support the simulation of design that useAltera devices.

ModelSim refers to ModelSim SE,PE and DE

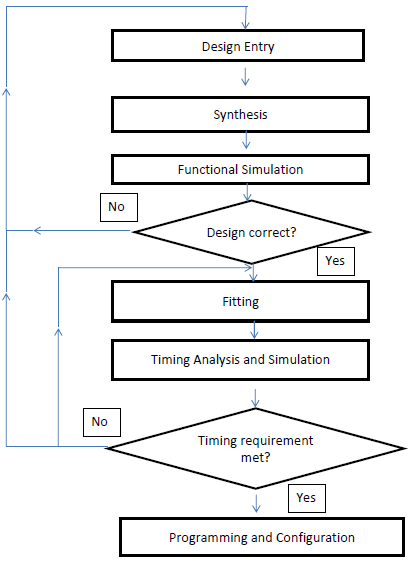
* Support for simulating small FPGA designs
* 10,000 executable limitations

The ModelSim-Altera Editionsoftware is licenced to support written 100% Verilog language and does not support designs that are written in a combination of VHDL and Verilog language, also known as mixed HDL. Mixed HDL support is available as option in the PE and DE versions of ModelSim from Mentor Graphics. ModelSim-Altera Edition software only supports Altera gate-level libraries.

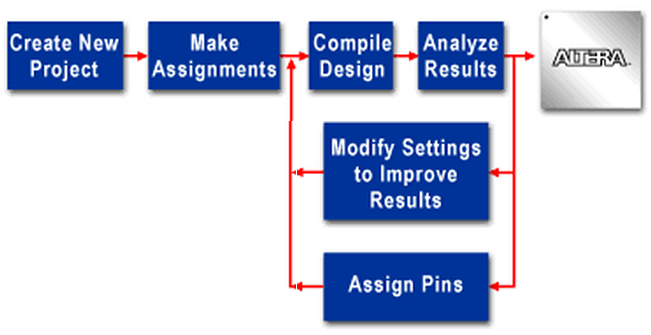
Types of simulations with the ModelSim-Altera

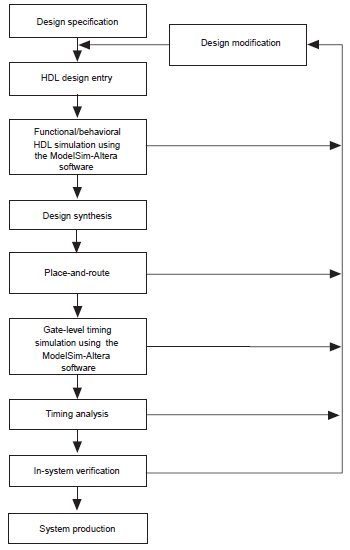
* Functional simulation
* Post-synthesis simulation
* Gate-level timing simulation

**5.1.3 Design flow of Quartus II**



**Figure 5.1**: Design Flow (Hardware Only)



**Figure 5.2**: Quartus II software basic design flow

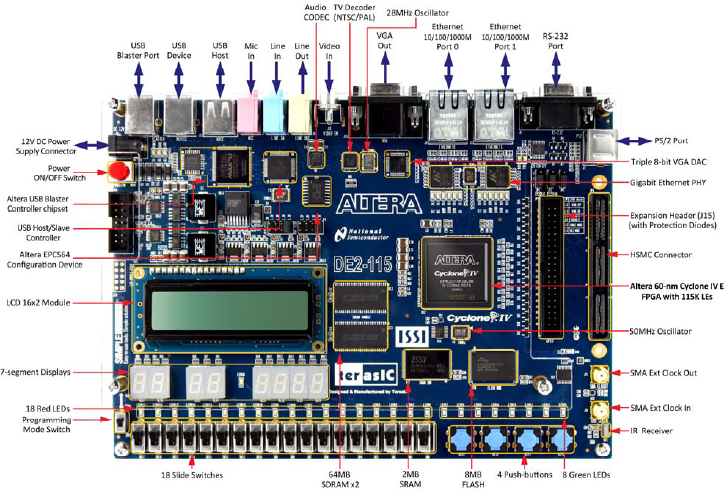
**Figure 5.3**: Altera Design Flow with ModelSim-Altera and Quartus II Software

**5.2 Hardware**

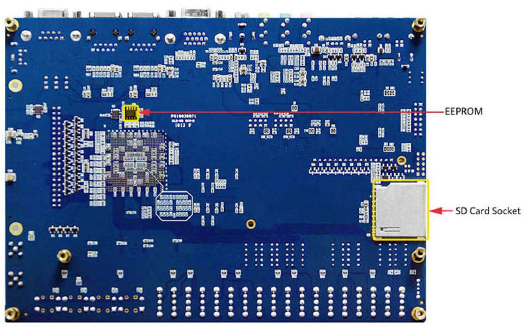
**5.2.1 Introduction to DE2-115 FPGA Board**

The DE2 box includes:

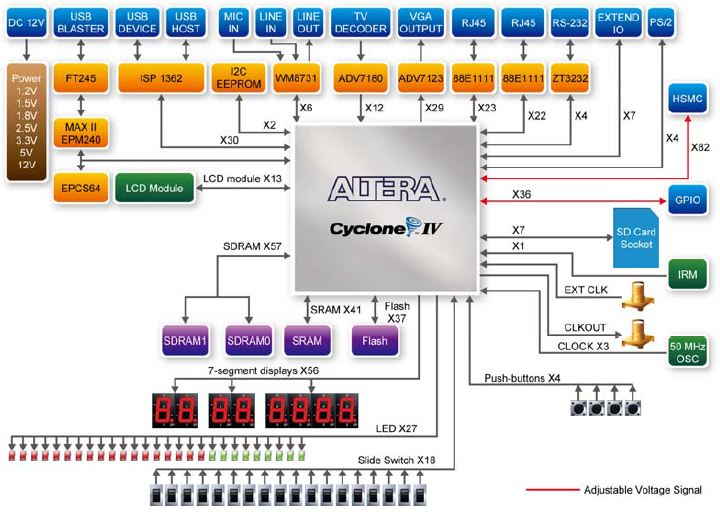
The 8x6 inch DE2 board with a cyclone II EP2C35 (672-pin package)FPGA 9V AC/DC adaptor,USB cable, Plexiglas cover for the DE2 board.



**Figure 5.4**: The DE2-115 board (top view)



**Figure 5.5**: The DE2-115 board (bottom view)



**Figure 5.6**: Block Diagram of DE2-115

The following hardware is provided on the DE2 board:

• Altera Cyclone® II 2C35 FPGA device

• Altera Serial Configuration device - EPCS16

• USB Blaster (on board) for programming and user API control; both JTAG and Active Serial(AS) programming modes are supported

• 512-Kbyte SRAM

• 8-Mbyte SDRAM

• 4-Mbyte Flash memory (1 Mbyte on some boards)

• SD Card socket

• 4 pushbutton switches

• 18 toggle switches

• 18 red user LEDs

• 9 green user LEDs

• 50-MHz oscillator and 27-MHz oscillator for clock sources

• 24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jacks

• VGA DAC (10-bit high-speed triple DACs) with VGA-out connector

• TV Decoder (NTSC/PAL) and TV-in connector

• 10/100 Ethernet Controller with a connector

• USB Host/Slave Controller with USB type A and type B connectors

• RS-232 transceiver and 9-pin connector

• PS/2 mouse/keyboard connector

• IrDA transceiver

• Two 40-pin Expansion Headers with diode protection

**5.3 Cyclone IV FPGA Family:**

Lowest Cost, Lowest Power, Integrated Transceivers

Cyclone IV FPGAs enable us to meet increasing bandwidth requirement while lowering costs. The family includes:

* Cyclone IV GX FPGAs with up to eight integrated 3.125 Gbps transceivers
* Cyclone IV E FPGAs for a wide spectrum of general logic applications

Offering up to 150,000 logic elements (LEs), Cyclone IV FPGAs consume up to 30% less total power. They’re ideal for low-cost, small form factor applications in : Broadcast, Consumer, Industrial, Wireless, Wireline.

All Cyclone IV FPGAs require only two power supplies for operation, simplifying your power distribution network and saving you board casts, board space, and design time. For Cyclone IV GX FPGAs, the cost saving and further increased. With the introduction of integrated transceivers on the leading low power Cyclone IV FPGA architecture, you get cost saving through simplified board design and integration. Furthermore, the flexibility of the transceiver clocking architecture allow you to implement multipleprotocols while fully utilizing all available transceivers resources. The integration and flexibility of Cyclone IV GX FPGA enables you to design in a smaller, lower cost device, lowering your total system cost.

**5.3.1 Reduced Power Consumption**

Built on an optimized 60-mm low-power process, Cyclone IV E FPGAs extend the low-power leadership of previous generation Cyclone III FPGAs. The latest generation devices reduce core voltage, which lowers total power by 25 percent compared to the predecessor. With Cyclone IV GX transceiver FPGAs, we can build a PCI Express to Gigabit bridge for less than 1.5 watts.

**5.3.2 Integrated Transceivers**

Cyclone IV GX FPGAs are built with Altera’s proven GX transceiver technology,known for excellent jitter performance and superior signal integrity.The PCI \_SIG-compliant transceiver variant supports a wide variety of serial protocols.Cyclone IV GX FPGAs also feature the only hard intellectual property (IP) block for PCI Express x1,x2 and x4 in rootport and end point configurations.

**5.3.3 Complete Design Resources**

To assure a smooth,successful design flow, and to make it possible for you to turn ideas into revenue quicker than ever before,Altera provides a complete Cyclone IV FPGA design environment including:

Quartus® II development software,

Library of proven IP

Nios®II,the world’s most versatile embedded processor

**Chapter 6**

**FPGA IMPLEMENTATION**

**6.1 Cyclone IV FPGA: Architecture**

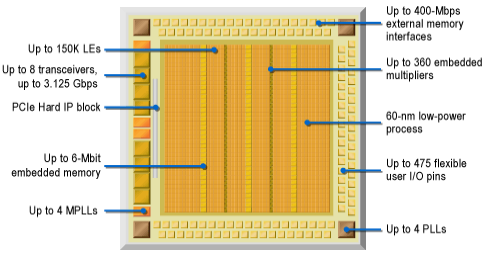
Cyclone® IV FPGAs continue the Cyclone series tradition of offering an unprecedented combination of low power, high functionality, and low cost.

The Cyclone IV GX FPGA architecture consists of up to 150K vertically arranged logic elements (LEs), 6.5 Mbits of embedded memory arranged as 9-Kbit (M9K) blocks, and 360 18 x 18 embedded multipliers. New to the Cyclone series, Cyclone IV GX FPGAs feature integrated transceivers at up to 3.125 Gbps.

The Cyclone IV E FPGA architecture consists of up to 115K vertically arranged LEs, 4 Mbits of embedded memory arranged as 9-Kbit (M9K) blocks, and 266 18 x 18 embedded multipliers.

The logic and routing core fabric is surrounded by I/O elements (IOEs) and phase-locked loops (PLLs), as shown in Figure 1. Both GX and E devices have four general-purpose PLLs located at each corner of the die. The Cyclone IV GX FPGA has I/O elements at the top, bottom, and right sides of the die, while the Cyclone IV E FPGA has I/Os on all four sides of the die. The left side of the Cyclone IV GX die has up to eight transceivers in two quads consisting of four transceivers per quad. The top and bottom of each transceiver quad features a multi-purpose PLL (MPLL) that can be used by the transceiver or, when available, by the FPGA fabric.

Both architectures include highly efficient interconnect and low-skew clock networks, providing connectivity between logic structures for clock and data signals.



**Figure 6.1**: Cyclone IV FPGA Key Architectural Features

Key Features

* Power:Altera's Cyclone® IV FPGAs are optimized for the lowest power consumption, helping us better manage thermal requirements. As a result, you can reduce or eliminate system cooling costs and also extend battery life for handheld applications.
* Embedded Multipliers: Cyclone® IV devices include a combination of on-chip resources and external interfaces that help increase performance, reduce system cost, and lower the powerconsumption of digital signal processing (DSP) systems. Cyclone IV devices, either alone or as DSP device co-processors, are used to improve price-to-performance ratios of DSP systems. Particular focus is placed on optimizing Cyclone IV devices for applications that benefit from an abundance of parallel processing sources, which include video and image processing, intermediate frequency (IF) modems used in wireless communications systems, and multi-channel communications and video systems.
* Memory Blocks: Cyclone® IV devices feature embedded memory structures to address the on-chip memory needs of Altera® Cyclone IV device designs. The embedded memory structure consists of columns of M9K memory blocks that we can configure to provide various memory functions, such as RAM, shift registers, ROM, and FIFO buffers.
* Clock Networks and PLLs:It has the ability to reconfigure the PLL counter clock frequency and phase shift in real time, allowing us to sweep PLL output frequencies and dynamically adjust the output clock phase shift. The Quartus® II software enables the PLLs and their features without externaldevices.
* Configuration, Design Security, and Remote System Upgrades: Cyclone IV devices are configured using one of the following configuration schemes:

■ Active serial (AS)

■ Active parallel (AP) (supported in Cyclone IV E devices only)

■ Passive serial (PS)

■ Fast passive parallel (FPP) (not supported in EP4CGX15, EP4CGX22, and EP4CGX30 [except for the F484 package] devices)

■ JTAG

Cyclone IV devices offer the following configuration features:

■ Configuration data decompression

■ Remote system upgrade

System designers face difficult challenges, such as shortened design cycles, evolving standards, and system deployments in remote locations. Cyclone IV devices help overcome these challenges with inherent re-programmability and dedicated circuitry to perform remote system upgrades. Remote system upgrades help deliver featureenhancements and bug fixes without costly recalls, reduced time-to-market, and extended product life.

* SEU Mitigation: Configuration error detection is supported in all Cyclone® IV devices including Cyclone IV GX devices, Cyclone IV E devices with 1.0-V core voltage, and Cyclone IV E devices with 1.2-V core voltage. However, user mode error detection is only supported in Cyclone IV GX devices and Cyclone IV E devices with 1.2-V core voltage.

Dedicated circuitry built into Cyclone IV devices consists of a CRC error detection feature that can optionally check for a single-event upset (SEU) continuously andautomatically.

In critical applications used in the fields of avionics, telecommunications, system control, medical, and military applications, it is important to be able to:

■ Confirm the accuracy of the configuration data stored in an FPGA device

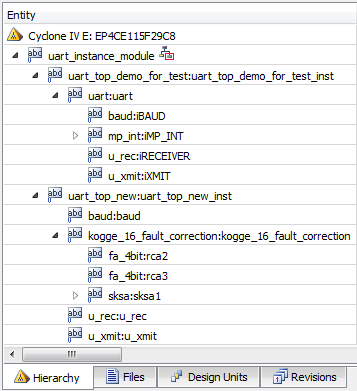
■ Alert the system to an occurrence of a configuration error

Using the CRC error detection feature for Cyclone IV devices does not impact fitting or performance.

**6.2 FPGA Design Flow**

FPGA Design Flow with Quartus II

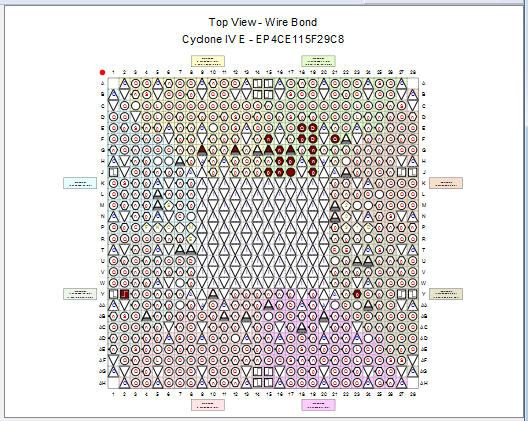
For implementation of the project the family selected is Cyclone IV E, device is EP4CE115F29C8. uart\_instance\_module was the top level module, though which several files are linked. The figure below shows the hierarchy of the different programs.



**Figure 6.2:** Hierarchical structure of programs.

**Table 6.1:** Pin description

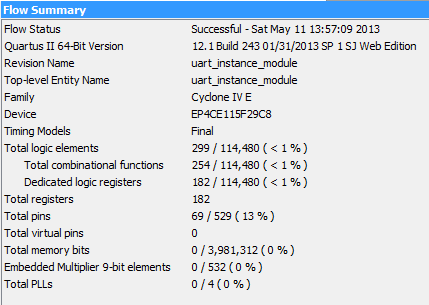
|  |  |  |
| --- | --- | --- |
| **Node Name** | **Direction** | **Location** |
| clock | Input | PIN\_Y2 |
| final\_cout | Output | PIN\_G16 |
| led\_out [15] | Output | PIN\_G15 |
| led\_out [14] | Output | PIN\_F15 |
| led\_out [13] | Output | PIN\_H17 |
| led\_out [12] | Output | PIN\_J16 |
| led\_out [11] | Output | PIN\_H16 |
| led\_out [10] | Output | PIN\_J15 |
| led\_out [9] | Output | PIN\_G17 |
| led\_out [8] | Output | PIN\_J17 |
| led\_out [7] | Output | PIN\_H19 |
| led\_out [6] | Output | PIN\_J19 |
| led\_out [5] | Output | PIN\_E18 |
| led\_out [4] | Output | PIN\_F18 |
| led\_out [3] | Output | PIN\_F21 |
| led\_out [2] | Output | PIN\_E19 |
| led\_out [1] | Output | PIN\_F19 |
| led\_out [0] | Output | PIN\_G19 |
| reset | Input | PIN\_Y23 |
| rxd\_in | Input | PIN\_G12 |
| txd\_out | Output | PIN\_G9 |



**Figure 6.3:** Pin planner

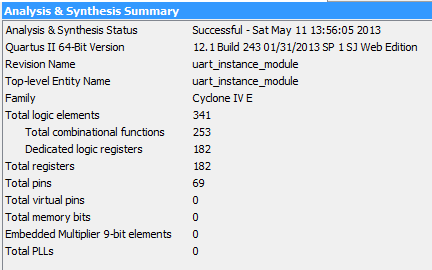
The above table describes the pin assignments onto the FPGA kit. It gives us an overview of the inputs and output ports. The above figure shows the top view of the FPGA when the pins are assigned.

**Table 6.2:** Flow Summary

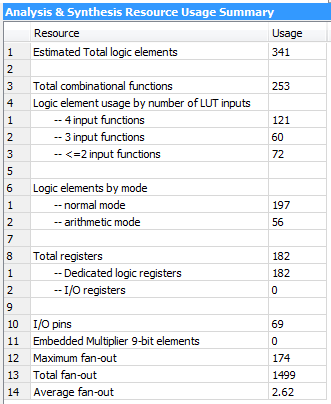


The above table gives the flow summary when full compilation of the project has been done. The table shows the top level entity along with the device and several other constraints. The table below gives the analysis and synthesis summary of the project.

**Table 6.3:** Analysis & Synthesis Summary

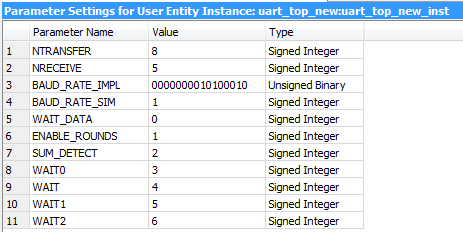


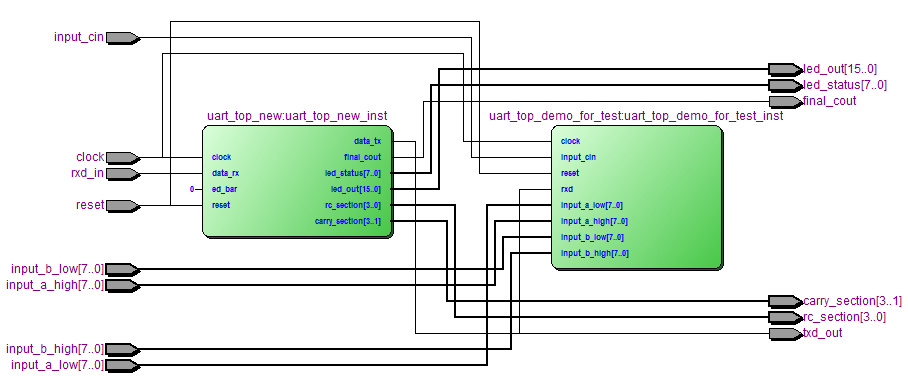
**Table 6.4**: Analysis & Synthesis Resource Usage Summary



The table above gives us the brief information about the resources used in the project. The table below shows the number of bits transferred to the FPGA from the hyper-terminal of the computer at a time and number of bits received from the FPGA by the hyper-terminal of the computer. It also gives the baud rate, and the value of state machines.

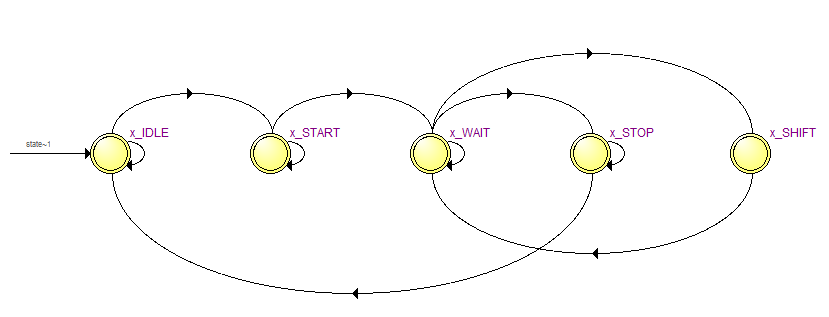
**Table 6.5:** Parameter settings for the user entity instance





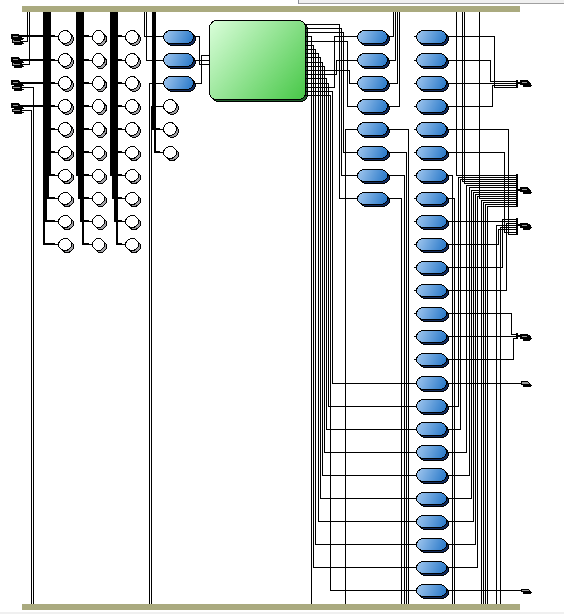
**Figure 6.4:** RTL view of uart\_instance\_module

The above figure gives the RTL view of the top level entity. It shows the connections with the different entities along with the input and outputs. The below figure shows the finite state machines used in the programming.



**Figure 6.5:** State machines

Figure 6.6 shows the Technology Map Viewer-Post Mapping of the project.



**Figure 6.6:** Technology Map- Post Mapping

**Chapter 7**

**EXPERIMENTAL SETUP**

**7.1 Aim**

To conduct an Experiment for the verification of **“**Fault-tolerant Kogge-Stone Configuration based Parallel-Prefix adder design for high speed GPUs”**.**

**7.2 Software and hardware used**

* **Software Requirements:** Quartus II 12.1sp1 web edition, ModelSim-Altera 10.1b (Quartus II 12.1sp1) Starter Edition.
* **Hardware Requirements:** USB Blaster driver, USB Blaster, ALTERA DE2-115 Board, UART, PC, Power supply.

**7.3 Theory**

Fault tolerance plays a very important role in modern systems where immediate human intervention is not possible and system failure can have disastrous consequences. A fault tolerant system has the ability to detect and then correct the occurrence of a hardware failure. In order to detect the fault, the system must be able to sense any deviations from its normal operation. A fully fault tolerant system also has the ability to correct the fault in order to return the system to its normal functionality.

A large portion of an FPGA chip consists of its configuration memory. The logic stored in the memory of the system can be altered by Single-Event Upsets (SEUs). SEUs can occur due to cosmic radiation or a high energy neutron striking the substrate of the device silicon. As SEUs can cause single-bit errors within the configuration memory, a fault tolerant system that uses FPGAs must guard against these occurrences. Fault tolerant systems are also important for circuits implemented on nanoscale technologies as external influences such as electromagnetic interference and cosmic rays can cause transient errors which in turn affect the operation of the devices and can degrade the system reliability. State-of-the-art FPGAs are designed with very fine geometrics making them susceptible to faults due to such electrical interference.

**7.4 Procedure**

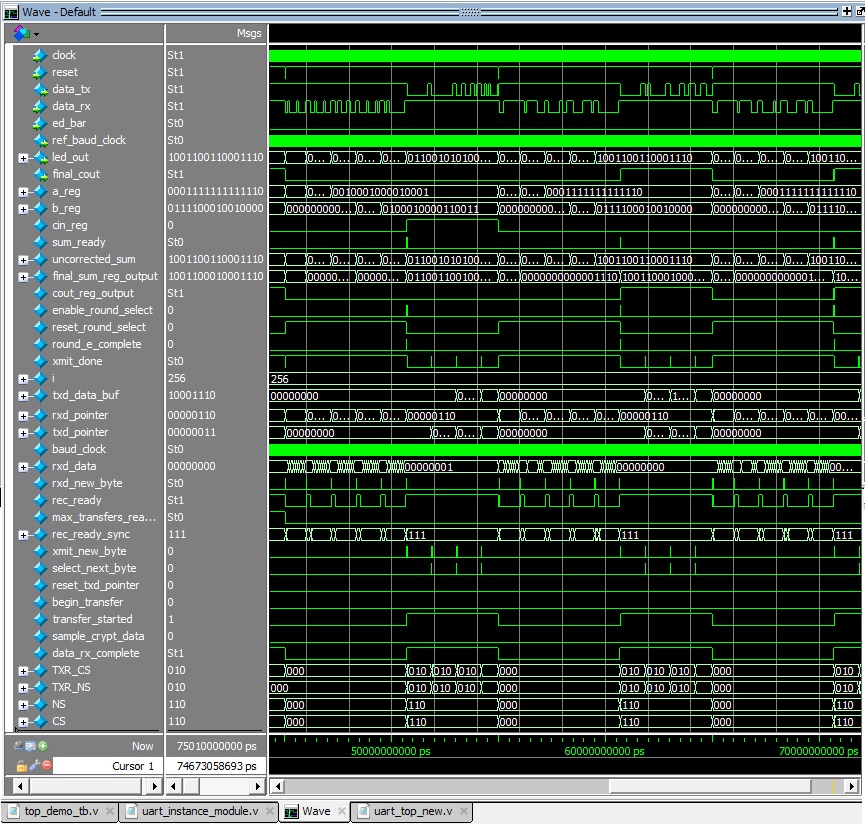
* Double click on the Quartus II 12.1sp1 Web Edition (64-Bit) icon on situated on the desktop.
* Go to file
* New project wizard.
* Specify the directory of the project.
* Give the name of the project.
* Add the files.
* Select the family of the device.
* Give the package of the device.
* Select speed and pin count of the device.
* Finally select the appropriate device.
* Do the EDA settings
* Simulation –Modelsim-Altera.
* Format –verilog Hdl.
* See the summary and click on the finish.
* From the Processingmenu, select **Start - Start Analysis & Elaboration**. Click **OK** when analysis and elaboration is completed.
* Select **Start Compilation** from the **Processing** menu or **click** on  located on the toolbar to perform a full compilation of the design. A dialog box will appear when the compilation is complete. Click **OK**.
* From the **Assignments** menu, open the **Pin Planner**.
* In the Assignment Editor select **Pin** from the **category menu**
* In the Edit menu, Double Click below the **TO** column and Select reset\_n and select PIN\_R22 in the **location** column.

(repeat the same for other pins also)

* From the **Processing** menu, go to **Start** and select **Start Analysis & Synthesis.** Click OK when complete.
* From the Processing menu, go to Start and select Start I/O Assignment Analysis. Click OK once the analysis is complete.
* Click on to do a full recompilation of the design. A full recompilation is necessary to re-run the place and route and to generate timing files.
* From the **Tools** menu, select **TimeQuest Timing Analyzer**.
* Create a timing netlist. In the Tasks pane of the TimeQuest GUI, double-click on Create Timing Netlist OR in the Console pane, type create\_timing\_netlist.
* Read an SDC file. Simply type **read\_sdc** at the **tcl>** prompt in the **Console** pane.
* Double-click on **Read SDC File** in the **Tasks** pane.
* Use the create\_clock command to add a \*\*-ns clock constraint to the clk input.
* From the Constraints menu in TimeQuest, select Create Clock…
* In the Create Clock dialog box, type clk as the clock name.
* In the **period** field, type **\*\***.
* In the **targets** field, click on the browse button .
* In the Name Finder, choose get\_ports from the Collection drop-down menu. Click List.
* Highlight clk in the list of matches found and click on the right arrow button to move it to the selected names section.
* Click **OK**.
* Click **OK**.
* Update the timing netlist. In the Tasks pane of the TimeQuest GUI, double-click on Update Timing Netlist OR in the Console pane, type update\_timing\_netlist.
* In the Tasks pane, double-click on Report SDC.
* In the Tasks pane, double-click on Report All Summaries.
* Highlight the report Summary (Setup).
* Write an output SDC file. In the Tasks pane of the TimeQuest GUI, double-click on Write SDC File .
* click on assignments settings Timing Analysis setting and selet time quest timing analyzer for the flow.
* click on Time quest Timing analyzer and select the file for adding the timing constraints and click ok
* Connect the USB cable to DE1 board and connect the other end to your PC USB port.
* Open the programmer by clicking on switch.
* Click on hardware setup tab and select USB Blaster as programming hardware. If USB Blaster is not available it will prompt you to install the driver .please point to Altera installation directory and select the driver from drivers folder.
* Click on Auto detect it will detect the devices in the JTAG chain. Double click on file and select stopwatch\_top.sof.
* Click on program configure and click on button start. Tool will download the program into FPGA.

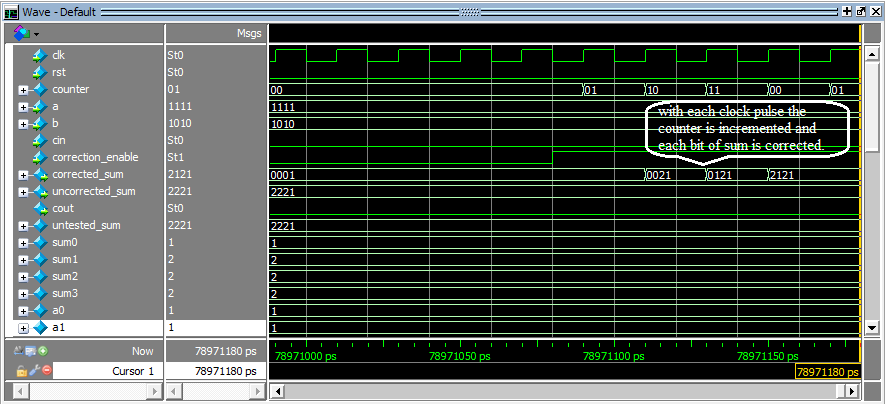
**7.5 Results**

**7.5.1 Software results**



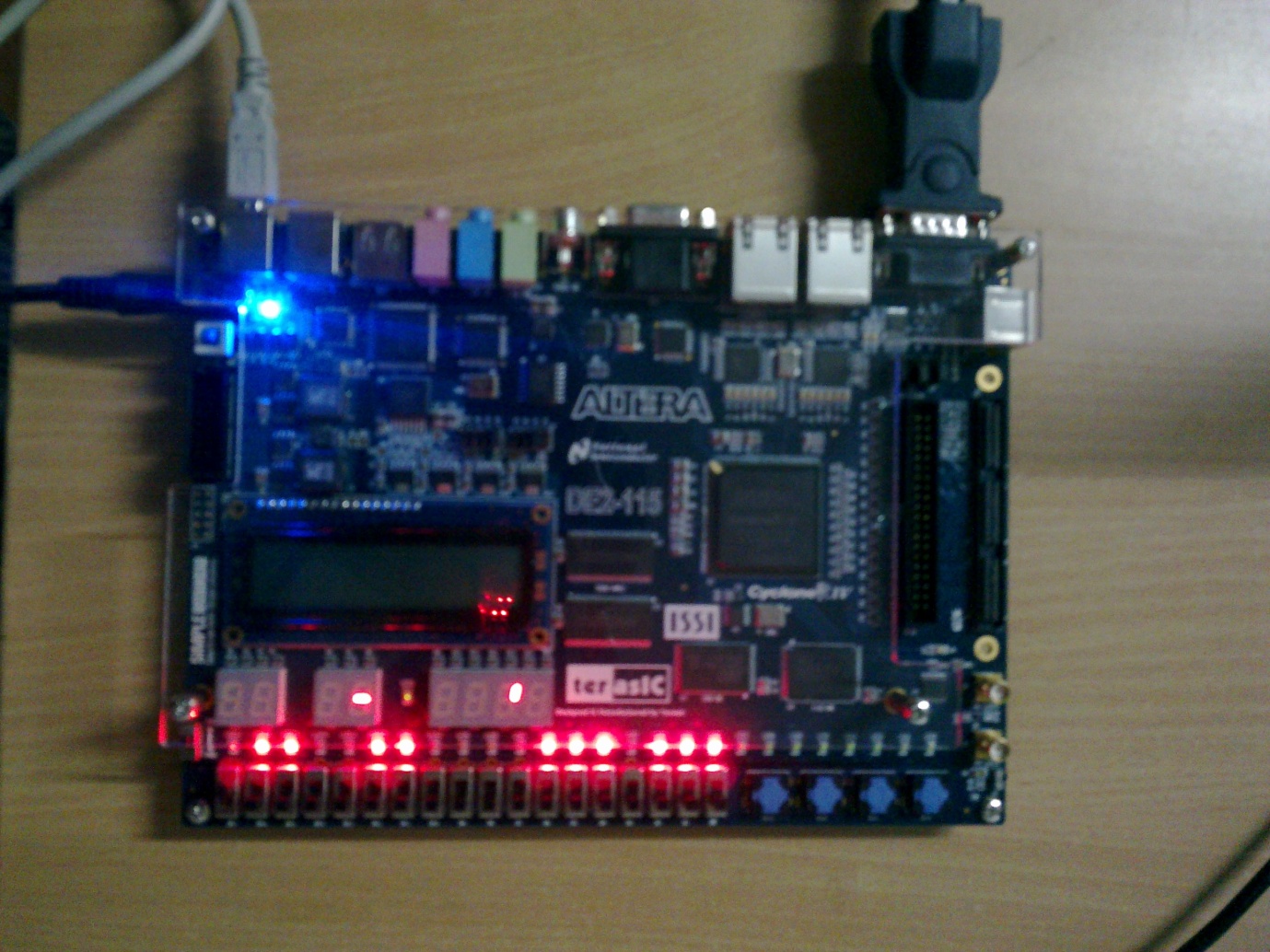
**Figure 7.1:** Simulated result

The simulated results showed the data transmitted and received along with the cock and counter needed to correct the fault.



**Figure 7.2:** Correction of sum.

**7.5.2 Hardware Results**

****

**Figure 7.3:** Hardware output

With the inputs as A = BBAAH and B = DDCCH and Cin = 0, the hardware result was 19876H , and on laptop it showed the data received from FPGA as 19976H, which proved our programming.

****

**Figure 7.4:** Connection of the DE2-115 with the laptop

**7.6 Verification**

As per our programming we saw 4 clock cycles to correct any fault, and both corrected and uncorrected outputs were seen and verified.

**Chapter 8**

**ADVANTAGES AND APPLICATIONS**

It is the fastest adder ever implemented. Ithas 0 (log2n) delays through the carry path compared to 0(n) for the RCA*.* Works satisfactorily in order to satisfy the VLSI requirements

area, power, cost and speed. Complexity is reduced compared to other relevant designs available.

These designs are important for many mission-critical applications, such as avionic and medical electronics, and for improved reliability for systems such as satellites that must operate in harsh and remote environments. Furthermore, a fault tolerant system is also important for circuits implemented in nanoscale technologies where the small device dimensions make the circuit susceptible to external interference, such as cosmic radiation.

**Chapter 9**

**CONCLUSION AND FUTURE WORK**

**9.1 Conclusion**

The fault tolerant adders implemented on FPGAs have been characterized with respect to their delay performance and logic complexity as a function of bit width. Basic fault tolerant adder designs like the Triple Modular Redundancy-Ripple Carry Adder (TMR-RCA) and error correcting regular Kogge-Stone adder were analyzed. A sparse Kogge-Stone adder which is fully fault tolerant in its lower half (i.e., in the ripple carry adders) was proposed. Simulation results demonstrate that this design is able to detect and correct errors in its RCA chains.

Architectures for implementing advanced fault tolerance techniques on the sparse Kogge-Stone adders were proposed. This includes the upper half fault tolerant sparse Kogge-Stone adder and a graceful degradation concept. Simulation and synthesis using FPGA design tools have validated the performance of the lower-half and upper-half sparse Kogge-Stone adders for both fault detection and correction. In this analysis, the TMR-RCA seems to be the most efficient approach for fault tolerant design on an FPGA in terms of its resources due to its simplicity and the ability to take the advantage of the fast-carry chain. However, for very large bit widths, there are indications that the Kogge-Stone adder offers superior performance over a ripple carry adder when implemented on an FPGA.

A fault tolerant sparse Kogge-Stone adder is designed by taking advantage of the existing ripple carry adders in the architecture and adopting a similar approach to the TMR-RCA by inserting two additional ripple carry adders into the design. A graceful degradation approach is implemented with the sparse Kogge-Stone adder. In this approach, a faulty block is permanently replaced with a spare block. As the spare block is initially used for fault checking, the fault tolerant capability of the circuit is degraded in order to continue fault-free operation. The adder delay is faster for graceful degradation with an overhead of 1 ns from measured results and an overhead of 2 ns from the synthesis results independent of the bit widths when compared with the fault tolerant Kogge-Stone adder even though the resource utilization is similar.

**9.2 Future work**

Two main areas for extending the present work are briefly considered. First, the development of methods and tools to make the proposed fault tolerant methods easier to implement can be undertaken. A method for easily scaling to larger bit widths for the upper half fault tolerant sparse Kogge-Stone adder should be investigated. Automated techniques for implementing the fully fault tolerant sparse Kogge-Stone adder should be developed. Second, a largely unexplored area of research is the application of error correcting codes to fault tolerant adder designs. In digital communications, an additional number of bits is added to a message to allow the detection and correction of corrupted bits during transmission. A similar method might be feasible with arithmetic circuits. An optimal error correcting code would take into account the logic structure of the adder and would enable fully fault tolerant implementations while adding a minimum amount of overhead.