

1-15-2003

Logical simulation of communication subsystem for Universal Serial Bus (USB)

Terikere Badarinarayana
Florida International University

Follow this and additional works at: <http://digitalcommons.fiu.edu/etd>



Part of the [Computer Engineering Commons](#)

Recommended Citation

Badarinarayana, Terikere, "Logical simulation of communication subsystem for Universal Serial Bus (USB)" (2003). *FIU Electronic Theses and Dissertations*. Paper 1363.
<http://digitalcommons.fiu.edu/etd/1363>

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

LOGICAL SIMULATION OF COMMUNICATION SUBSYSTEM FOR UNIVERSAL
SERIAL BUS (USB)

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by


Terikere Badarinarayana

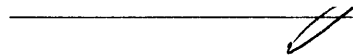
2003

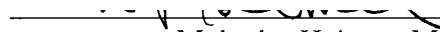
To: Dean Vish Prasad
College of Engineering

This thesis, written by Terikere Badarinarayana, and entitled Logical Simulation of Communication Subsystem for Universal Serial Bus (USB), having been approved in respect to style and intellectual content is referred to you for judgment.

We have read this thesis and recommend that it be approved.



Subbarao Wunnava

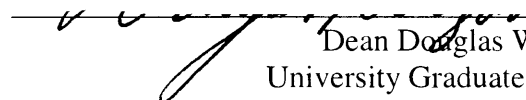

Tadeusz M. Babij


Malcolm Heimer, Major Professor

Date of Defense: January 15, 2003

The thesis of Terikere Badarinarayana is approved.


Dean Vish Prasad
College of Engineering


Dean Douglas Wartzok
University Graduate School

Florida International University, 2003

DEDICATION

I dedicate this thesis to my parents: T. P Srinivasa and Sathya Prema, and my brother Raghavendra. Without their support, care and patience, this thesis would not have been possible.

ACKNOWLEDGMENTS

I would like to thank my committee members: Dr. Tadeusz Babij, Dr. Subbarao Wunnava, and my Major Professor Dr. Malcolm Heimer for guiding me through this thesis. Without their invaluable guidance and patience, completion of this thesis would not have been possible. I would also like to thank all my lab members for their support. I would also like to thank my friends Suneel Konduru and Ravi Kishan Arikere for all the help they have provided me.

I would like to thank everyone in FIU Electrical Engineering Department, especially to Pat Brammer for her constant words of encouragement throughout my stay here.

ABSTRACT OF THE THESIS

LOGICAL SIMULATION OF COMMUNICATION SUBSYSTEM FOR UNIVERSAL SERIAL BUS (USB)

by

Terikere Badarinarayana

Florida International University, 2003

Miami, Florida

Professor Malcolm Heimer, Major Professor

The primary purpose of this thesis was to design a logical simulation of a communication sub block to be used in the effective communication of digital data between the host and the peripheral devices. The module designed is a Serial interface engine in the Universal Serial Bus that effectively controls the flow of data for communication between the host and the peripheral devices with the emphasis on the study of timing and control signals, considering the practical aspects of them.

In this study an attempt was made to realize data communication in the hardware using the Verilog Hardware Description language, which is supported by most popular logic synthesis tools. Various techniques like Cyclic Redundancy Checks, bit-stuffing and Non Return to Zero are implemented in the design to provide enhanced performance of the module.

TABLE OF CONTENTS

CHAPTER	PAGE
1. Very Large Scale Integration.....	1
1.1 Introduction	1
1.2 History of VLSI.....	3
1.3 Design Methodologies.....	4
1.4 Design Hierarchy.....	8
1.5 Design Styles.....	9
1.5.1 Field Programmable Gate Array	9
1.5.2 Gate Array Design.....	12
1.5.4 Full Custom Design.....	17
2. Verilog Hardware Description Language.....	19
2.1 Hardware Description Language.....	19
2.2 History and growth of Verilog	20
2.3 Hierarchical Modeling concepts.....	22
2.3.1 Design Methodologies.....	23
2.3.2 Modules.....	24
2.3.3 Instance.....	25
2.4 Ports.....	25
2.5 Components of a simulation	26
3. Veriwell Simulations for Verilog.....	27
3.1 The levels of module abstraction inVerilog HDL.....	27
3.1.1 Behavioral or Algorithmic Level.....	27

3.1.2 Dataflow Level	28
3.1.3 Gate Level	28
3.1.4 Switch Level.....	29
3.2 Simulation Tool	30
3.2.1 Simulation Steps for the VeriWell Verilog Simulator	31
3.2.2 Steps to Create Simulation Waves	32
3.2.3 Examples using Veriwell Verilog simulator	33
3.2.3.1 Half Adder.....	33
3.2.3.2 Full Adder	37
3.2.3.3 Decoder	41
4. Universal Serial Bus.....	46
4.1 Introduction to Universal Serial Bus	46
4.2 Universal Serial Bus System	49
4.2.1 Role of Host PC Hardware and Software.....	49
4.2.2 Universal Serial Bus Hardware	51
4.2.3 Role of the Peripherals	54
4.2.4 Universal Serial Bus Software	56
4.2.4.1 Universal Serial Bus Device Drivers.....	56
4.2.4.2 Universal Serial Bus Driver	56
4.2.4.3 Universal Serial Bus Host Controller Driver	57
4.3 Serial Interface Engine	58

5. Module Design	61
5.1 Design Features of the Communication System	61
5.2 Techniques used for Reliable and Efficient Transmission	62
5.2.1 Bit-stuffing	62
5.2.2 Error checking using Cyclic Redundancy Check	63
5.2.3 NRZ (Non Return to Zero)	64
5.3 Design description of Serial Interface Engine	65
6. Transmitter and Receiver	68
6.1 Transmitter	68
6.1.1 Decoder	68
6.1.2 Buffer	70
6.1.2.1 Logical functioning of the Buffer	71
6.1.3 Clock	72
6.1.4 First In First Out	73
6.1.5 Control Logic	75
6.1.6 Multiplexer	76
6.1.7 CRC Generator	77
6.1.8 Parallel to Serial Converter	80
6.1.9 Transmitter Module	82
6.2 Receiver	84
6.2.1 Serial to Parallel Converter	84
6.2.2 Demultiplexer	87
6.2.3 Control Logic	88

6.2.4 Address Decoder	91
6.2.5 Timer	92
6.2.6 CRC Generator	93
6.2.7 CRC Comparator	96
6.2.8 First In First Out	97
6.2.9 Receiver Module	99
6.3 Serial Interface Engine	101
7. Results and Conclusion	104
7.1 Simulated Waves of the Serial Interface Engine (SIE)	104
7.2 Conclusions	109
7.3 Future Work	110
References	111

Chapter 1 Very Large Scale Integration (VLSI)

1.1 Introduction

Very Large Scale Integration (VLSI) of systems of transistor-based circuits on a single chip first occurred in the 1980s as part of the semiconductor and communication technologies that were being developed.

The first semiconductor chips held one transistor each. Subsequent advances added more and more transistors, and as a consequence more individual functions or systems were integrated over time. The microprocessor is a VLSI device. In its short life span, microelectronics has become the most complex of our everyday technologies, embracing as it does physics, chemistry, materials, thermodynamics, and micro mechanical engineering, as well as electrical and electronic engineering and computer science. (No one person can hope to be expert in all these diverse aspects.) With the advent of VLSI. The number of applications of integrated circuits in high-performance computing, telecommunications, and consumer electronics has been rising steadily, and at a very fast pace. Typically, the required computational power (or, in other words, the intelligence) of these applications is the driving force for the fast development of this field. The current leading-edge technologies (such as low bit-rate video and cellular communications) already provide the end-users a certain amount of processing power and portability. This trend is expected to continue, with very important implications on VLSI and systems design. Figure 1.1 gives an overview of the prominent trends in information technologies over the next few decades. One of the most important characteristics of

information services is their increasing need for very high processing power and bandwidth (in order to handle real-time video, for example). The other important characteristic is that the information services tend to become more and more personalized (as opposed to collective services such as broadcasting), which means that the devices must be more intelligent to answer individual demands, and at the same time they must be portable to allow more flexibility/mobility. [1][2] [3]

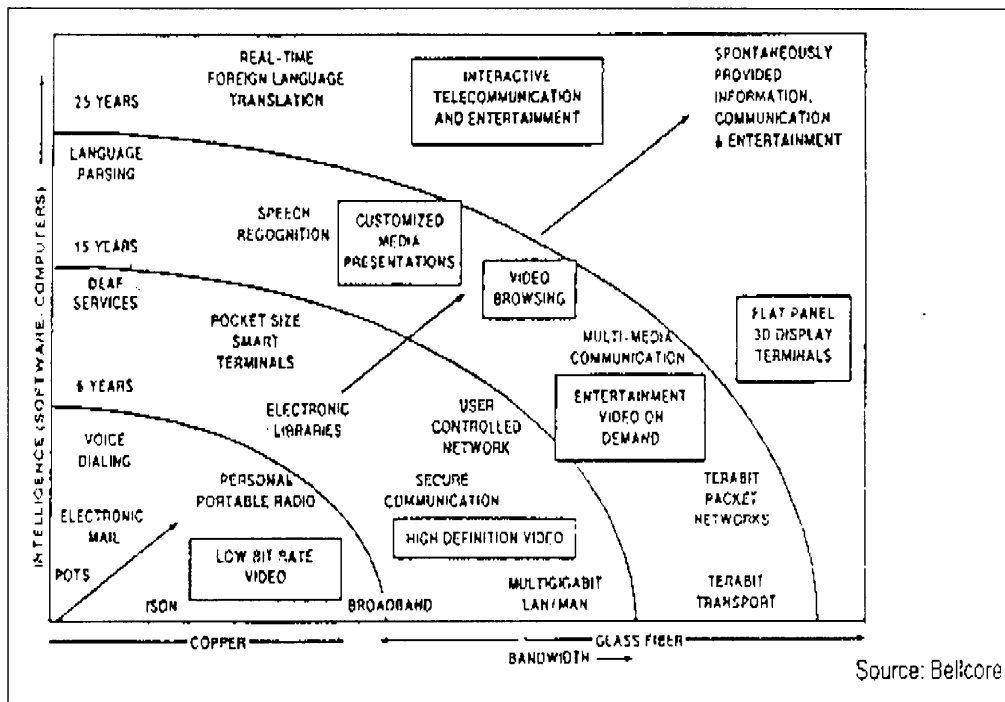


Figure 1.1 Prominent trends in information service technologies [11]

The communication systems have many complex functions thus integrating these is not simple in to a single package the need to integrate them is also on a rise. The levels of integration as measured by the number of logic gates in a monolithic chip has been

steadily rising for almost three decades, mainly due to the rapid progress in processing technology and interconnect technology

1.2 History of VLSI [10]

1948	TRANSISTOR INVENTED (SHOCKLEY AT&T) GERMANIUM-GOLD CONTACT
1954	SILICON TRANSISTOR (TEAL TI) HIGHT TEMP.
1956	TRANSISTOR COMPUTER (CRAY)
1958	FIRST MONOLITHIC CIRCUIT (IC) BJTs (KIRBY - TI & NOYCE - FAIRCHILD)
1960	SSI (< 100 TRANSISTORS) MOSFET - PMOS, METAL GATE (BELL LABS)
1961	TTL (PACIFIC MICROTEL) - 25UM FEATURE SIZE
1962	ECL (MOTOROLA)
1964	OPAMP (WILDAR - FAIRCHILD U709)
1965	PDP-8 < \$20,000
1966	MSI (100 - 1000 TRANSISTORS)
1967	FIRST PRODUCTION MOS CHIPS
1969	LSI (1000 - 10000 TRANSISTORS) PMOS, NMOS, CMOS
1969	E-BEAM PRODUCTION, DIGITAL WATCHES, CALCULATORS
1970	CCD (BELL LABS), MICROPROCESSOR (HOFT - INTEL)
1971	ION IMPLANTATION
1972	I2L (IBM), 16 BIT MICROS

1975	VLSI (10,000 - 100,000 TRANSISTORS) SELF-ALIGNED PROCESSES
1975	SPICE DEVELOPED (U CAL. BERKLEY)
1980's	ULSI (> 100,000 TRANSISTORS) ASICS, PLD, TRENCH CAPS, DUAL WELL, BIMOS, HVICS FEATURE SIZE 2UM
1990's	> 1,000,000 TRANSISTORS 64-bit MICROS, MICROMACHINING, FPGA SYNTHESIS, VHDL, FEATURE SIZE 0.5UM

1.3 Design Methodologies

The design process, at various levels, is usually evolutionary in nature. It starts with a given set of requirements. Initial design is developed and tested against the requirements. When requirements are not met, the design has to be improved. If such improvement is either not possible or too costly, then the revision of requirements and its impact analysis must be considered. The Y-chart (first introduced by D. Gajski) shown in Fig. 1.2 illustrates a design flow for most logic chips, using design activities on three different axes (domains) that resemble the letter Y.

The Y-chart consists of three major domains, namely:

- Behavioral domain,
- Structural domain,
- Geometrical layout domain.

The design flow starts from the algorithm that describes the behavior of the target chip. The corresponding architecture of the processor is first defined. It is mapped onto the chip surface by floor planning. The next design evolution in the behavioral domain defines finite state machines (FSMs) that are structurally implemented with functional modules such as registers and arithmetic logic units (ALUs). These modules are then geometrically placed onto the chip surface using CAD tools for automatic module placement followed by routing, with a goal of minimizing the interconnects area and signal delays. The third evolution starts with a behavioral module description. Individual modules are then implemented with leaf cells. At this stage the chip is described in terms of logic gates (leaf cells), which can be placed and interconnected by using a cell placement & routing program. The last evolution involves a detailed Boolean description of leaf cells followed by a transistor level implementation of leaf cells and mask generation. In standard-cell based design, leaf cells are already pre-designed and stored in a library for logic design use.

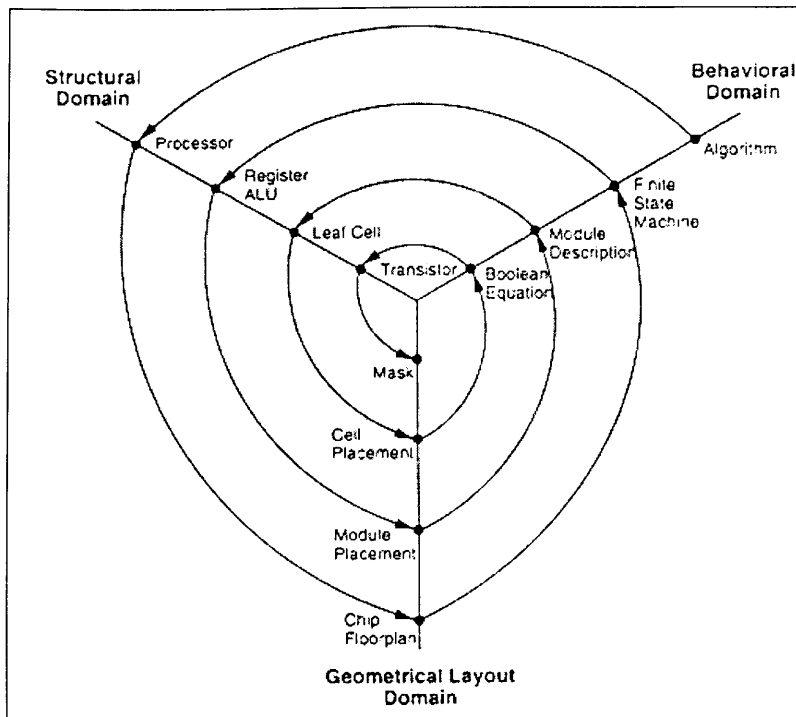


Figure 1.2 Typical VLSI design flow in three domains (Y-chart representation) [11]

Figure 1.3 depicts a more simplified view of the VLSI design flow that takes into account the various representations, or abstractions of design - behavioral, logic, circuit and mask layout. Note that at every step the verification of design is performed, which is a very important role during this process. The failure to do so in its early phases typically causes significant and expensive re-design at a later stage, which ultimately increases the time-to-market.

The diagram showing the design process is doing it in linear fashion for simplicity, in reality there is much iteration back and forth, especially between any two neighboring steps, and occasionally even remotely separated pairs. Although top-down design flow provides an excellent design process control, in reality, there is no truly

unidirectional top-down design flow. Both top-down and bottom-up approaches have to be combined. For instance, if a chip designer defined architecture without close estimation of the corresponding chip area, then it is very likely that the resulting chip layout exceeds the area limit of the available technology. In such a case, in order to fit the architecture into the allowable chip area, some functions may have to be removed and the design process must be repeated. Such changes may require significant modification of the original requirements. Thus, it is very important to feed forward low-level information to higher levels (bottom up) as early as possible.

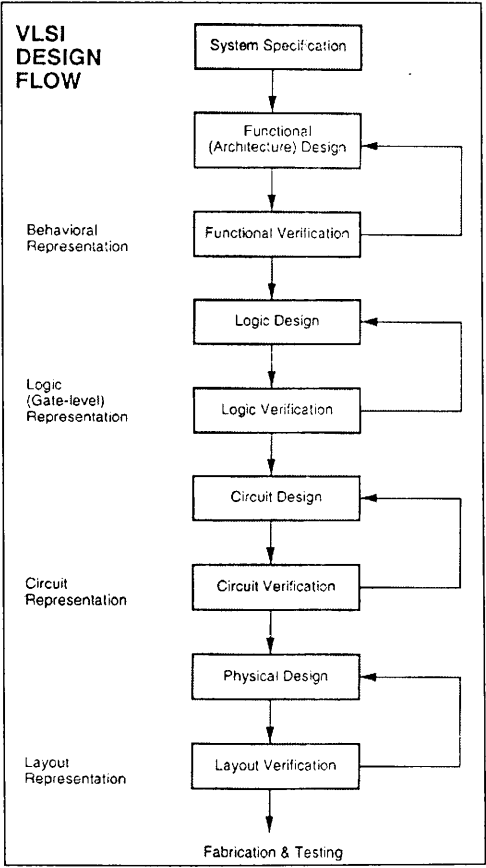


Figure 1.3 VLSI design flow [11]

1.4 Design Hierarchy

This technique involves dividing the module into simpler ones reducing the complexity levels to the level that it can be managed.

As an example of structural hierarchy, Fig. 1.4 shows the structural decomposition of a CMOS four-bit adder into its components. The adder can be decomposed progressively into one-bit adders, separate carry and sum circuits, and finally, into individual logic gates. At this lower level of the hierarchy, the design of a simple circuit realizing a well-defined Boolean function is much more easier to handle than at the higher levels of the hierarchy.

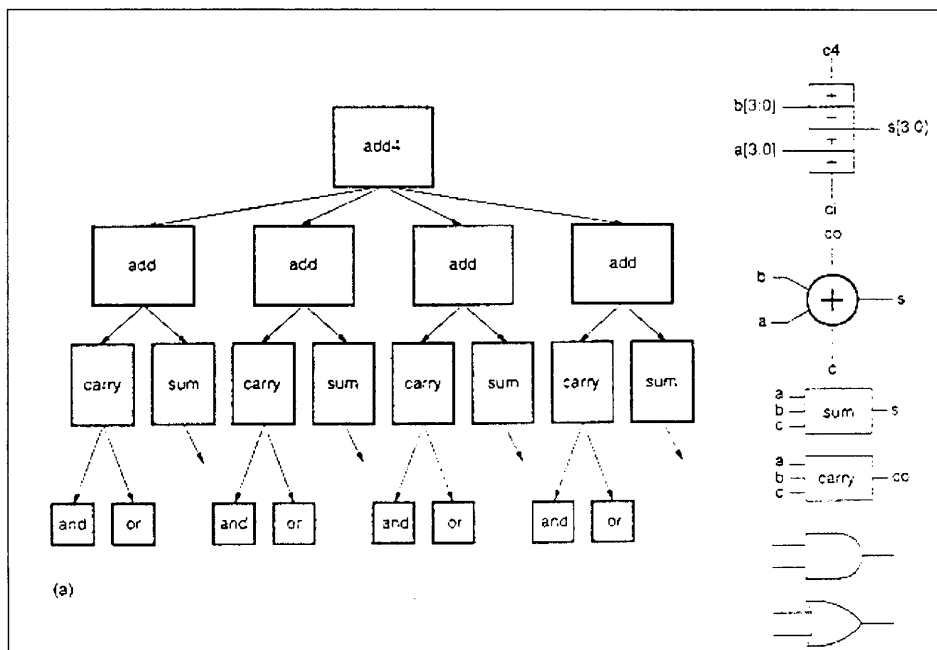


Figure 1.4 Structural decomposition of a four-bit adder circuit, showing the hierarchy down to gate level [11]

1.5 Design Styles

There are several designs available for chip implementations of specified algorithms or logic functions. These come with their own merits hence a right choice of it is very important to provide functionality at low cost.

1.5.1 Field Programmable Gate Array (FPGA)

Fully fabricated FPGA chips containing thousands of logic gates or even more, with programmable interconnects, are available to users for their custom hardware programming to realize desired functionality. This design style provides a means for fast prototyping and also for cost-effective chip design, especially for low-volume applications. A typical field programmable gate array (FPGA) chip consists of I/O buffers, an array of configurable logic blocks (CLBs), and programmable interconnect structures. The programming of the interconnects is implemented by programming of RAM cells whose output terminals are connected to the gates of MOS pass transistors. A general architecture of FPGA from XILINX is shown in Fig.1.5. A more detailed view showing the locations of switch matrices used for interconnect routing is given in Fig.1.6

A simple CLB (model XC2000 from XILINX) is shown in Fig. 1.7 It consists of four signal input terminals (A, B, C, D), a clock signal terminal, user-programmable multiplexers, an SR-latch, and a look-up table (LUT). The LUT is a digital memory that stores the truth table of the Boolean function. Thus, it can generate any function of up to

four variables or any two functions of three variables. The control terminals of multiplexers are not shown explicitly in Fig.1.7.

The CLB is configured such that many different logic functions can be realized by programming its array. More sophisticated CLBs have also been introduced to map complex functions. The typical design flow of an FPGA chip starts with the behavioral description of its functionality, using a hardware description language such as VHDL or Verilog HDL. The synthesized architecture is then technology-mapped (or partitioned) into circuits or logic cells. At this stage, the chip design is completely described in terms of available logic cells. Next, the placement and routing step assigns individual logic cells to FPGA sites (CLBs) and determines the routing patterns among the cells in accordance with the netlist. After routing is completed, the on-chip performance of the design can be simulated and verified before downloading the design for programming of the FPGA chip. The programming of the chip remains valid as long as the chip is powered-on, or until new programming is done. In most cases, full utilization of the FPGA chip area is not possible - many cell sites may remain unused.

The largest advantage of FPGA-based design is the very short turn-around time, i.e., the time required from the start of the design process until a functional chip is available. Since no physical manufacturing step is necessary for customizing the FPGA chip, a functional sample can be obtained almost as soon as the design is mapped into a specific technology.

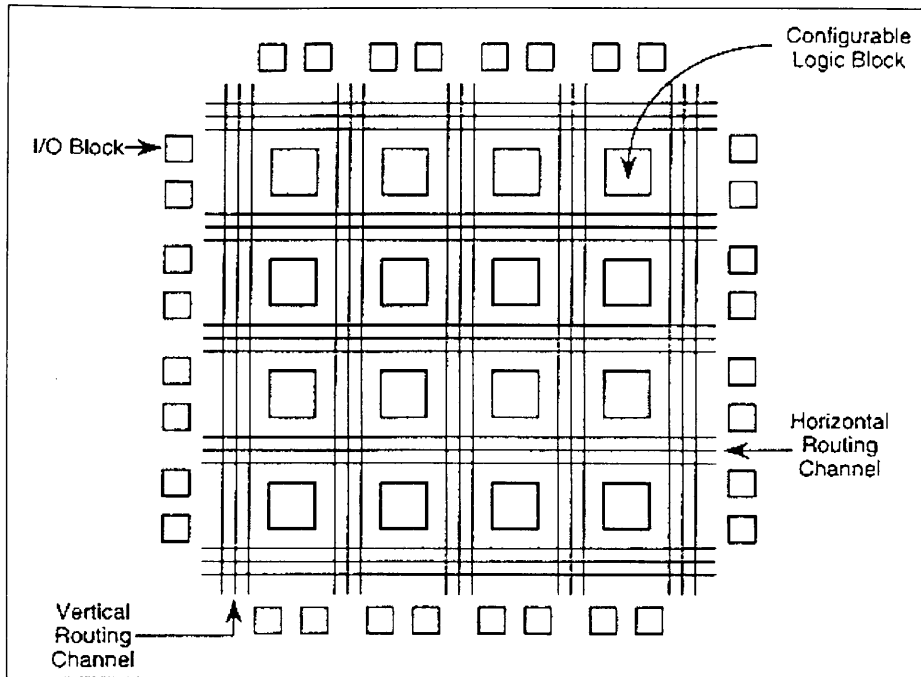


Figure 1.5 General architecture of Xilinx FPGAs [11]

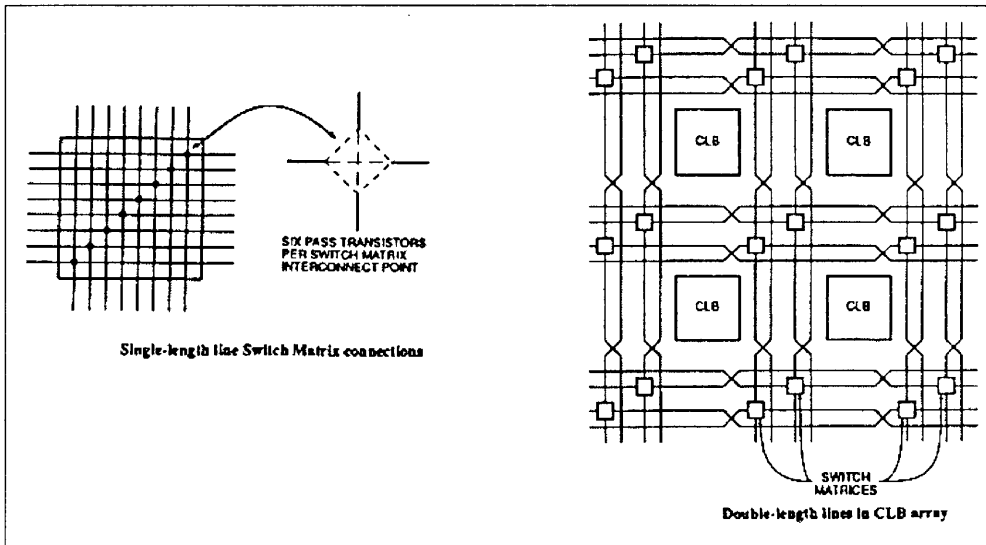


Figure 1.6 Detailed view of switch matrices and interconnection routing between CLBs [11]

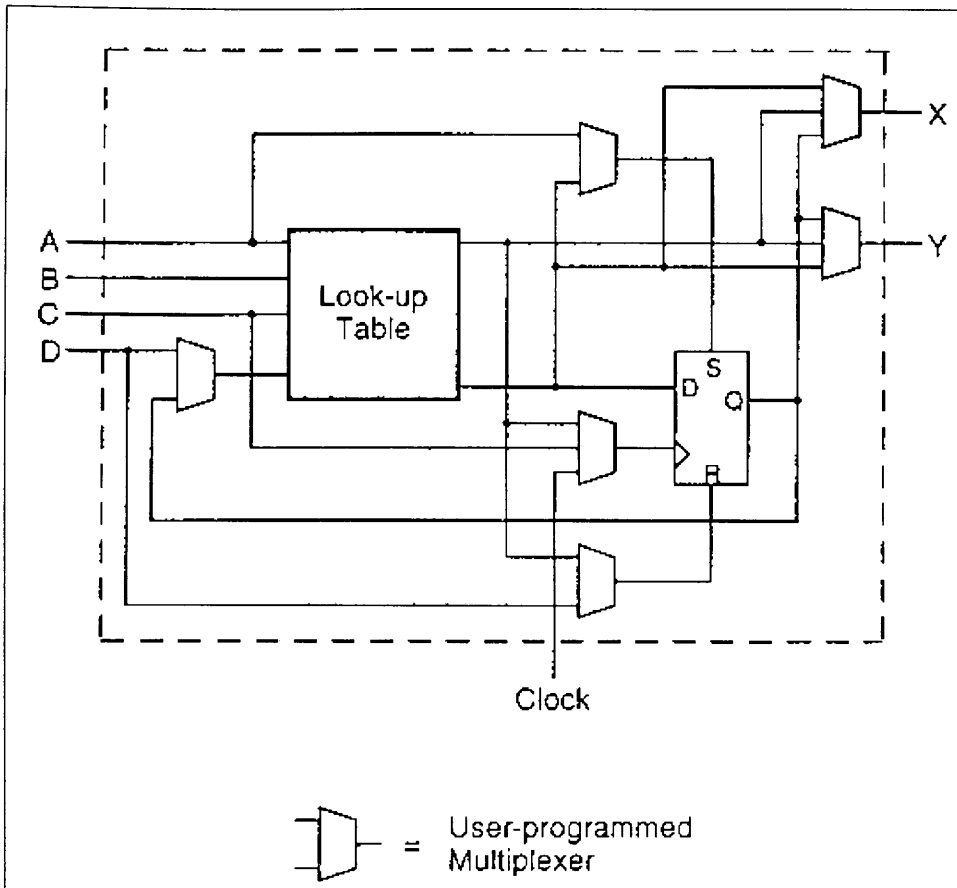


Figure 1.7 XC2000 CLB of the Xilinx FPGA [11]

1.5.2 Gate Array Design

The design implementation of the gate array is done with metal mask design and processing while that of the FPGA chip is done with user programming.

Gate array implementation requires a two-step manufacturing process: The first phase, which is based on generic (standard) masks, results in an array of uncommitted transistors on each GA chip. These uncommitted chips can be stored for later

customization, which is completed by defining the metal interconnects between the transistors of the array as in fig 1.8 Since the patterning of metallic interconnects is done at the end of the chip fabrication, the turn-around time can be still short, a few days to a few weeks. Figure 1.9 shows a corner of a gate array chip which contains bonding pads on its left and bottom edges, diodes for I/O protection, nMOS transistors and pMOS transistors for chip output driver circuits in the neighboring areas of bonding pads, arrays of nMOS transistors and pMOS transistors, underpass wire segments, and power and ground buses along with contact windows.

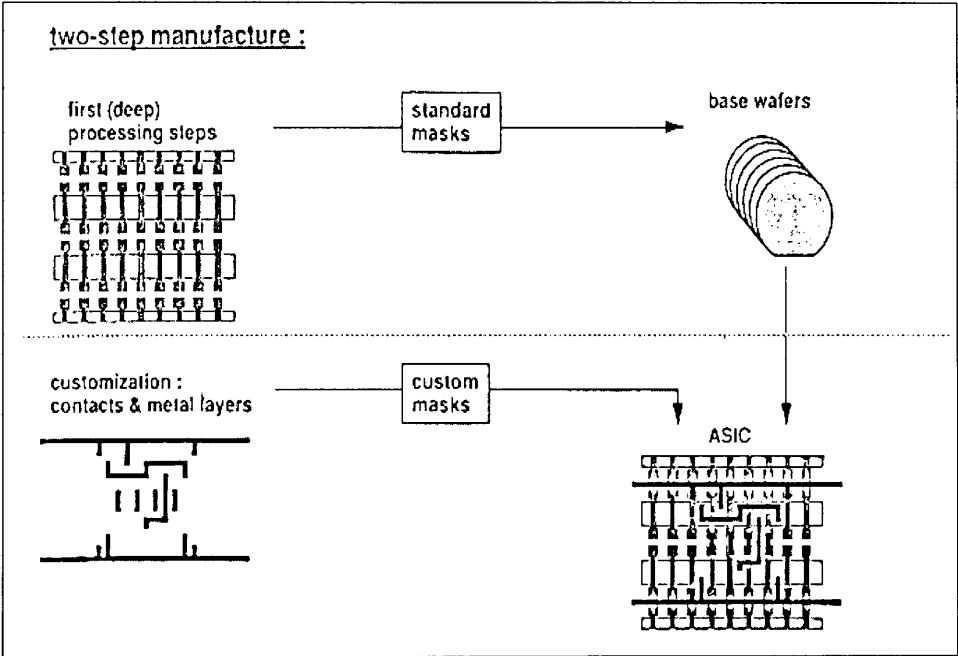


Figure 1.8 Basic processing steps required for gate array implementation [11]

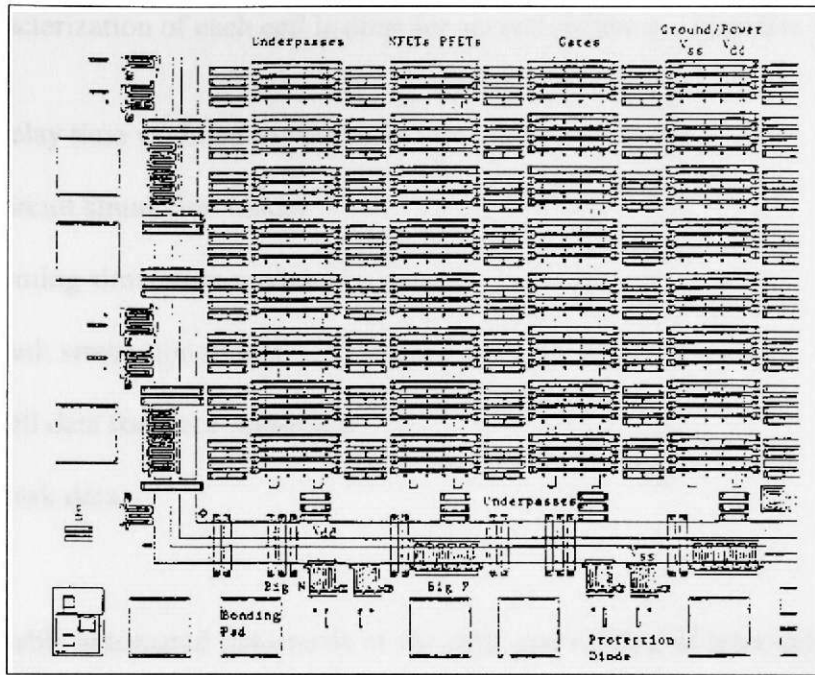


Figure 1.9 A corner of a typical gate array chip [11]

1.5.3 Standard-Cells Based Design

The standard-cells based design is one of the most prevalent full custom design styles that require development of a full custom mask set. The standard cell is also called the polycell. In this design style, all of the commonly used logic cells are developed, characterized, and stored in a standard cell library. A typical library may contain a few hundred cells including inverters, NAND gates, NOR gates, complex AOI, OAI gates, D-latches, and flip-flops. Each gate type can have multiple implementations to provide adequate driving capability for different fan-outs. For instance, the inverter gate can have standard size transistors, double size transistors, and quadruple size transistors so that the chip designer can choose the proper size to achieve high circuit speed and layout density.

The characterization of each cell is done for several different categories. It consists of

- Delay time vs. Load capacitance
- Circuit simulation model
- Timing simulation model
- Fault simulation model
- Cell data for place-and-route
- Mask data

To enable automated placement of the cells and routing of inter-cell connections, each cell layout is designed with a fixed height, so that a number of cells can be abutted side-by-side to form rows. The power and ground rails typically run parallel to the upper and lower boundaries of the cell, thus, neighboring cells share a common power and ground bus. The input and output pins are located on the upper and lower boundaries of the cell. Figure 1.10 shows the layout of a typical standard cell. Notice that the nMOS transistors are located closer to the ground rail while the pMOS transistors are placed closer to the power rail. Figure 1.11 shows a floor plan for standard-cell based design. Inside the I/O frame that is reserved for I/O cells, the chip area contains rows or columns of standard cells. Between cell rows are channels for dedicated inter-cell routing. As in the case of Sea-of-Gates, with over-the- cell routing, the channel areas can be reduced or even removed provided that the cell rows offer sufficient routing space.

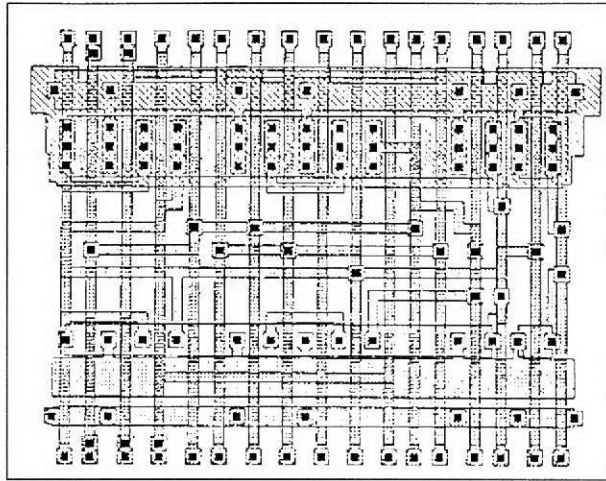


Figure 1.10 A standard cell layout example [11]

The physical design and layout of logic cells ensure that when cells are placed into rows, their heights are matched and neighboring cells can be abutted side-by-side, which provides natural connections for power and ground lines in each row. The signal delay, noise margins, and power consumption of each cell should be also optimized with proper sizing of transistors using circuit simulation.

1.5.4 Full Custom Design

Although the standard-cells based design is often called full custom design, in a strict sense, it is somewhat less than fully custom since the cells are pre-designed for general use and the same cells are utilized in many different chip designs. In a fuller custom design, the entire mask design is done anew without use of any library.

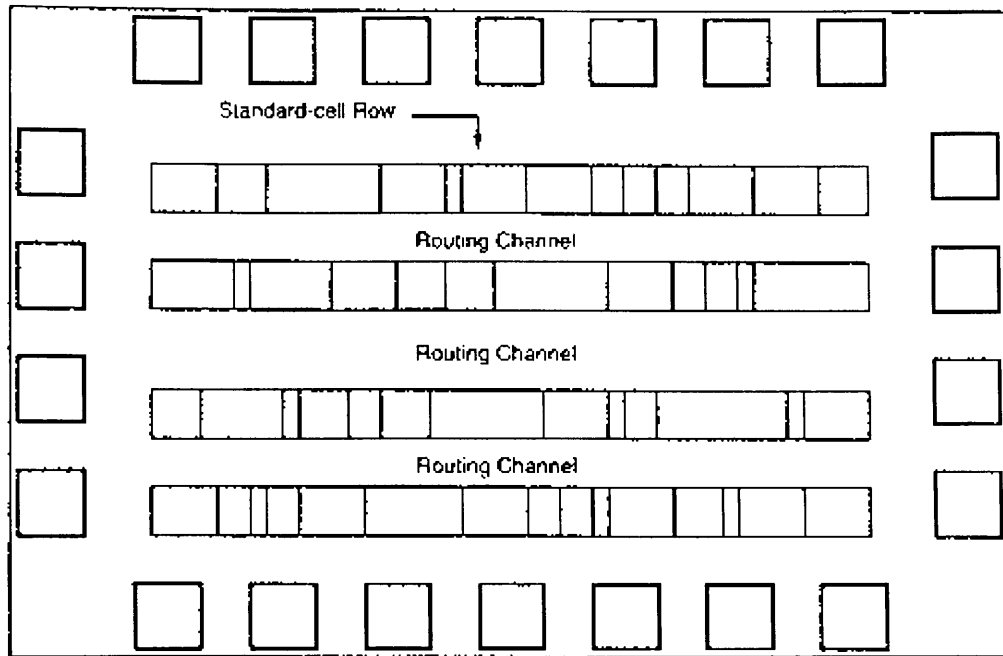


Figure 1.11 A simplified floor plan of standard-cells-based design [11]

However, the development cost of such a design style is becoming prohibitively high. Thus, the concept of design reuse is becoming popular in order to reduce design cycle time and development cost. The most rigorous full custom design can be the design of a memory cell, be it static or dynamic. Since the same layout design is replicated, there would not be any alternative to high-density memory chip design. For logic chip design, a good compromise can be achieved by using a combination of different design styles on the same chip, such as standard cells, data-path cells and PLAs. In real full-custom layout in which the geometry, orientation and placement of every transistor is done individually by the designer, design productivity is usually very low - typically 10 to 20 transistors per day, per designer.

In digital CMOS VLSI, full-custom design is rarely used due to the high labor cost. Exceptions to this include the design of high-volume products such as memory chips, high-performance microprocessors and FPGA masters.

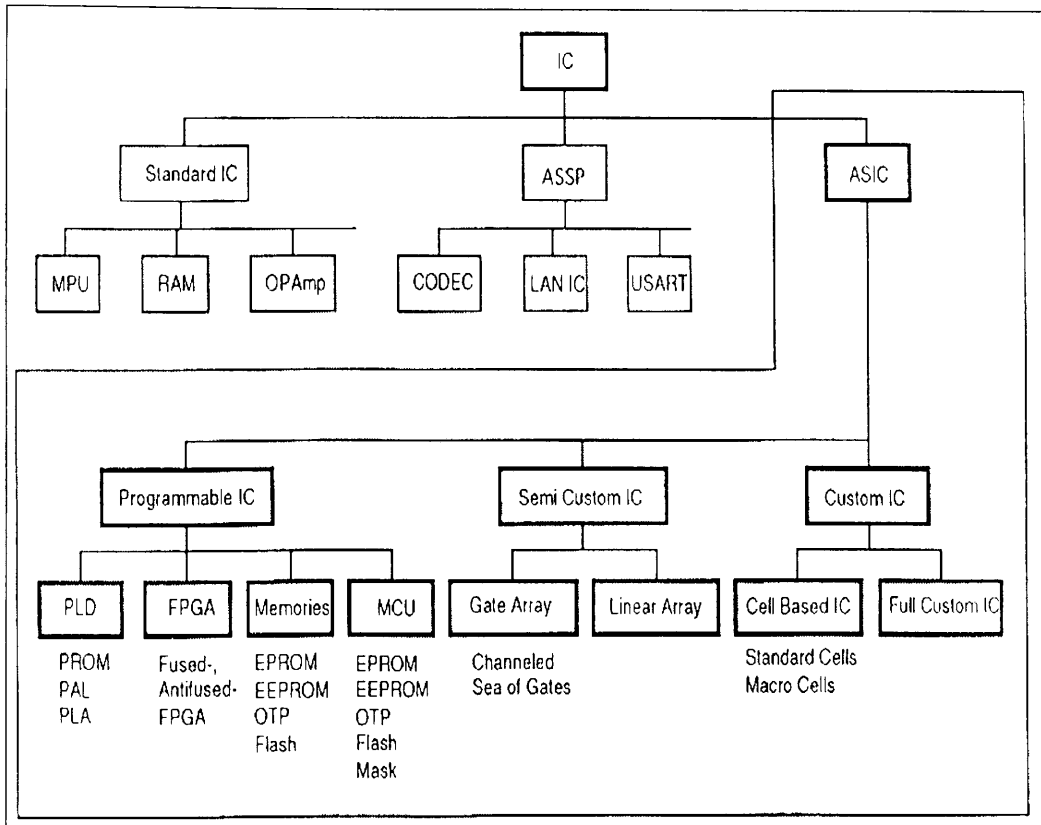


Figure 1.12 Overview of VLSI design styles [11]

Chapter 2 Verilog Hardware Description Language

2.1 Hardware Description Language

Hardware Description Languages, or HDLs, are languages used to design hardware with. As the name implies, an HDL can also be used to describe the functionality of hardware as well as its implementation.

With the advent of VLSI (Very Large Scale Integration) technology, designers could design single chips with more than 1,00,000 transistors. Because of the complexity of these circuits, it was not possible to verify these circuits on a breadboard. Computer-aided techniques became critical for verification and design of VLSI digital circuits. Computer programs to do automatic placement and routing of circuit layouts also became popular. The designers were now building gate-level digital circuits manually on graphic terminals. They would build small building blocks and then derive higher-level blocks from them. This process would continue until they had built the top-level block. Logic simulators came into existence to verify the functionality of these circuits before they were fabricated on chip. As designs got larger and more complex, logic simulation assumed an important role in the design process. Designers could iron out functional bugs in the architecture before the chip was designed further.

The principal feature of a hardware description language is that it contains the capability to describe the function of a piece of hardware independently of the implementation. The great advance with modern HDLs was the recognition that a single language could be used to describe the function of the design and also to describe the

implementation. This allows the entire design process to take place in a single language, and thus a single representation of the design.

2.2 History and growth of Verilog [2]

The Verilog Hardware Description Language, usually just called Verilog, was designed and first implemented by Phil Moorby at Gateway Design Automation in 1984 and 1985. It was first used beginning in 1985 and was extended substantially through 1987. The implementation was the Verilog-XL simulator sold by Gateway.

1986 - Verilog-XL

The first major extension to the language was Verilog-XL, which added a few features and implemented the infamous "XL algorithm," a very efficient method for doing gate-level simulation. This occurred in 1986, and marked the beginning of Verilog's growth period. Many leading-edge electronic designers began using Verilog at this time because it was fast at gate level simulation, and had the capabilities to model at higher levels of abstraction. These users began to do full system simulation of their designs, where the actual logic being designed was represented by a netlist and other parts of the system were modeled behaviorally.

1988 - Synopsis Design Compiler

In 1988, Synopsis delivered the first logic synthesizer, which used Verilog as an input language. This was a major event, as now the top-down design methodology could actually be used effectively. The design could be done at the "register transfer level", and

then Synopsys' Design Compiler could translate that into gates. With this event, the use of Verilog increased dramatically.

1989 - ASIC Signoff Certification

Beginning in 1989, another major trend began to emerge, the use of Verilog-XL for sign-off certification by ASIC vendors. As Verilog became popular with the semiconductor vendor's customers, they began to move away from their own, proprietary simulators, and started allowing customers to simulate using Verilog-XL for timing certification. As more ASIC vendors certified Verilog-XL, they requested more features, especially related to timing checks, back annotation, and delay specification. In response, Gateway implemented many new features in the language and the simulator to accommodate this need.

Cadence Design Systems acquired Gateway in December 1989, and continued to market Verilog as both a language and a simulator. At the same time, Synopsys was marketing the top-down design methodology, using Verilog. This was a powerful combination.

1990-5 - Opening of Verilog

From its inception through the end of the 1980s, Verilog was a proprietary language. No other vendors were allowed to make a Verilog simulator. By 1990, Cadence recognized that if Verilog remained a closed language, the pressures of standardization would eventually cause the industry to shift to VHDL. Consequently, Cadence organized Open Verilog International (OVI), and in 1991 gave it the documentation for the Verilog

Hardware Description Language. This was the event, which "opened" the language. Subsequently, OVI did a considerable amount of work to improve the Language Reference Manual (LRM), clarifying things and making the language specification as vendor-independent as possible.

In 1994, the IEEE 1364 working group was formed to turn the OVI LRM into an IEEE standard. This effort was concluded with a successful ballot in 1995, and Verilog became an IEEE standard in December 1995.

1992-Present - Multiple Vendors

When Cadence gave OVI the LRM, several companies began working on Verilog simulators. In 1992, the first of these were announced, and by 1993 there were several Verilog simulators available from companies other than Cadence. The most successful of these was VCS, the Verilog Compiled Simulator, from Chronologic Simulation. This was a true compiler as opposed to an interpreter, which is what Verilog-XL was. As a result, compile time was substantial, but simulation execution speed was much faster. Now, Verilog simulators are available for most computers at a variety of prices, and which have a variety of performance characteristics and features. Verilog is more heavily used than ever, and it is growing faster than any other hardware description language. It has truly become the standard hardware description language.

2.3 Hierarchical Modeling concepts

In digital design it is very important to understand basic hierarchical concepts. The designer must use a good design methodology to do efficient Verilog HDL based design.

2.3.1 Design Methodologies

There are two basic types of digital design methodologies

Top-down design methodology and

Bottom-up design methodology.

In top down we define the top-level block and identify the sub blocks necessary to build the top-level block Further, sub blocks are divided until we come to leaf cells, which are the cells that cannot be further divided.

In bottom up design methodology, first identify the building blocks that are available for designing of the module. After identifying the basic cells, bigger cells are built using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design.

Typically a Combination of the top down and bottom up flows is used Design architects define the specifications of the top-level block. Logical designers decide how the design should be structured by breaking up the functionality into blocks and sub blocks. At the same time the circuit designers are designing optimized circuits for leaf level cells they build higher level cells by using these leaf cells The flow meets at an intermediate point where the switch level circuit designers have created a library of cells by using switches, and logic designers have designed from top down until all modules are defined in terms of leaf cells.

2.3.2 Modules

HDL has an concept of module .it forms the basic building block in Verilog .A module can be an element or an collection of lower level design blocks. Typically, elements are grouped in to modules to provide common functionality that is used at many places in the design .a module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to module internals without affecting the rest of the design

Module is a logical component of a model.

Model is the logic design that a set of Verilog source files describes. This is a generic term, which comes from "simulation model". System and design are often used as synonyms.

Modules have definitions and instances. The definition contains declarative and procedural code sections, net and registers declarations, task and function definitions, module instantiations, and port definitions for connecting to other parts of the hierarchy.

A module is defined like this:

```
module <module_name> (<portlist>);  
    // module components  
endmodule
```

In Verilog it is illegal to nest modules. One module definition cannot contain another module definition within the module and end module statements. Instead a module can incorporate the copies of the other modules by instantiating them.it is important not to confuse module definitions and instances of a module. Module

definitions simply specify how the module works its internals and its interface. Modules must be instantiated for use in the design.

2.3.3 Instance

Is an embodiment of a module in the overall Verilog model.

A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters and I/O interface. The process of creating objects from a module template is called instantiation, and objects are called instances.

2.4 Ports

Ports are Verilog structures that pass data between parent and child modules. Thus, ports can be thought of as wires connecting modules. The connections provided by ports can be input (input port), output (output port), or bi-directional (inout port).

Ports are listed in the port list in the module definition, and their direction is declared following the module statement

Signal names in the instance port list are matched up left-to-right with signal names in the module definition port list.

Signal names in the instance port list can also be matched up with the signal names in the module definition by name.

The <module_name> is the type of this module. The <portlist> is the list of connections, or ports, which allows data to flow into and out of modules of this type.

2.5 Components of a simulation

Once a design block is completed it has to be verified, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results .it is called the stimulus block. It is good practice to keep the stimulus and the design blocks separate. The stimulus block can be written in Verilog separate language is not required to describe stimulus. The stimulus block is also commonly called a test bench. Different test benches can be used to thoroughly test the design block.

Two styles of stimulus application are possible.

In the first style the stimulus block instantiates the design block and directly drives the signals in the design block.

The second style of applying stimulus is to instantiate both the stimulus and the design blocks in a top-level dummy module. Stimulus block interacts with the design block only through the interface.

Chapter 3 Veriwell Simulations for Verilog

3.1 The levels of module abstraction in Verilog HDL

The hierarchical modeling concepts of the VerilogHDL provide a concept of module, which is the basic building block of the Verilog designs. Verilog is both a behavioral and a structural language, internals of each module can be defined at four levels of abstraction, depending on the needs of the design .the module behaves identically with the external environment irrespective of the level of abstraction at which the module is described. The internals of the modules are hidden from the environment. Thus the level of abstraction, to describe a module can be changed without any change in the environment. Given are the following four levels

3.1.1 Behavioral or Algorithmic Level

This is the highest level of abstraction provided by Verilog HDL.A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.

The behavior of a design is described using procedural constructs. These are:

- Initial statement: This statement executes only once.
- Always statement: This statement always executes in a loop, that is, the statement is executed repeatedly.

Only a register data type can be assigned a value in either of these statements. Such a data type retains its value until a new value is assigned. All initial statements and always statements begin execution at time 0 concurrently.

3.1.2 Dataflow Level

At this level the module is designed by specifying the data flow .the designer is aware of how data flows between hardware registers and how the data is processed in the design

The basic mechanism used to model a design in the dataflow style is the continuous assignment. In a continuous assignment, a value is assigned to a net. The syntax of a continuous assignment is:

```
assign [delay] LHS_net = RHS_expression
```

Anytime the value of an operand used in the right-hand side expression changes, the right-hand side expression is evaluated, and the value is assigned to the left-hand side net after the specified delay. The delay specifies the time duration between a change of operand on the right-hand side and the assignment to the left-hand side. If no delay value is specified, the default is zero delay.

3.1.3 Gate Level

The module is implemented in terms of the logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of gate level logic diagram. Verilog supports basic logic gates as predefined primitives. These

are instantiated like modules except that they are predefined in Verilog and do not need a module definition. All logic circuits can be designed by using basic gates. There are two types of basic gates they are AND/OR gates and BUF/NOT gates. AND/OR gates have one scalar output and multiple scalar inputs.

The AND/OR gates available in Verilog are shown below

AND, OR, XOR, NAND, NOR, XNOR, BUF/NOT gates have one scalar input and one or more scalar outputs.

Two basic gate primitives are

BUF

NOT

3.1.4 Switch Level

This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches storage nodes and the interconnections between them. Design at this level requires the knowledge of switch level implementation details.

Verilog allows the designer to mix and match all four levels of abstractions in a design. In the digital design community, the term register transfer level is frequently used for a Verilog description that uses a combination of behavioral and dataflow constructs and is acceptable to logic synthesis tools. If a design contains four modules. Verilog allows each of the modules to be written at a different level of abstraction. As the design matures, most modules are replaced with gate-level implementations.

Normally, the higher the level of abstraction, the more flexible and technology independent the design. As one goes lower toward the switch level design, the design becomes technology dependent and inflexible. A small modification can cause a significant number of changes in the design. Consider the analogy with C programming and assembly language programming. It is easier to program in a higher-level language such as C. The program can be easily ported to any machine. However, if you design at the assembly level the program is specific for that machine and cannot be easily ported to another machine.

3.2 Simulation Tool

The simulation tool used in this project to run the Verilog HDL programs is VeriWell 2.0. VeriWell was developed by Wellspring Solutions, Inc.

VeriWell is a comprehensive implementation of Verilog HDL. VeriWell supports a number of platforms and operating environments. These currently include 386/486/Pentium systems under DOS, Sparc or Sparc compatible systems under SunOS 4.1.x or greater and Solaris. VeriWell is designed to be as portable as possible. Nearly 100% of the sources are shared between the different platform versions. The DOS version uses a DOS extender to compensate for the shortcomings of DOS and to fully utilize the 32-bit architecture of the 386/486/Pentium processors.

VeriWell supports the Verilog language as specified by the OVI (Open Verilog International) Language Reference Manual. VeriWell was first introduced in December 1992, and is the first independently developed simulator to be written, from the first line of code, to be compatible with the OVI standard and with Verilog-XL. Because it was developed on the PC, it was specifically designed to be memory-efficient with relatively high performance.

VeriWell is used by IC designers and consultants for all pre-synthesis model development. As new features are added, VeriWell can be used in all phases of model development, including structural verification and back-annotated timing verification. As a component of a large-scale top-down design methodology, VeriWell is used in conjunction with other high-end OVI-compliant simulators, such as Verilog-XL or Chronologic's.

3.2.1 Simulation Steps for the VeriWell Verilog Simulator

1. Open the Veriwell simulation window by clicking on the saved Veriwell executable file.
2. Create a new project.

A VeriWell project is a collection of all the Verilog source files you need to run the simulation.

Steps to create a new project

- a. From the menu at the top of the window, select project -> new project.

- b. Enter a project name in the Save as window.
 - c. VeriWell creates a blank window that represents the new project.
 3. Create a new Source File

Steps to create a new source file
 - a. Select File New from main menu, to create a new verilog source file.
 - b. Select File -> Save As to store the file with the desired name.
 - c. Assume the project name is testP. When saved projects have an extension .prj. Let the source file be named testS. Source files have an extension .v. Thus we have a project testP.pri and a source file testS.v.
 4. Enter the Verilog code in the source file (testS.v).
 5. Add the source file to the project, by selecting Project->Add from the main menu.
 6. Run the simulation. Select Project->Run.
 7. VeriWell compiles the source file in the project.
 8. If the compilation is successful, the console window reports a success and a command prompt appears.

3.2.2 Steps to Create Simulation Waves

To create waves the Veriwell waves output capabilities from the Veriwell is used. Veriwave is integrated in to Veriwell and does not require a separate executable.

1. During the creation of the source file, the statement “dumpvars” should be added to the source code.

2. After successful compilation of the source code, to view the simulation waves, select, Project->dumpvars, in the main window.
3. This creates a wave file. Assign a name to the wave file. It will be stored with a .Vwf extension. A simulation window now pops up, containing the waveforms.

3.2.3 Examples using Veriwell Verilog Simulator

Software Details

Name: VeriWell Verilog Simulator

Version: 2.1.1

Operating System: Microsoft Windows

Source: <http://www.ece.ogi.edu/~strom/ece573/downloads.htm>

Note: The evaluation version of the simulator will not execute programs that exceed 1000 lines of Verilog Code.

3.2.3.1 Half Adder

The 'Half Adder' module consists of the following

Input ports: ' a ' and ' b '

Out ports: ' s' and ' c '

All ports are 1 bit.

The Half Adder shown below is an example of a group of logic gates connected to produce a logic circuit. The Half Adder has two inputs (the bits to be summed) and two

outputs (the sum bit and the carry bit). A Half Adder is the simplest form of an adder circuit. It has two operand bits a and b that are added to form a sum bit s and a carry bit c.

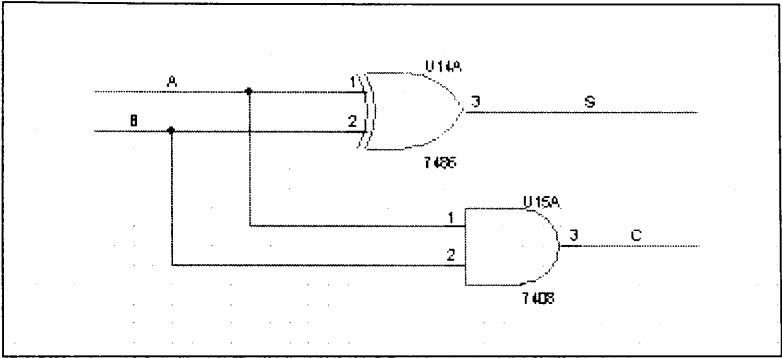


Figure 3. 1 Half Adder [2]

Table 3.1 Half Adder Truth Table.

A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Gate Level Modeling

```
/******
```

Gate level modeling - half adder

```
*****/
```

```
module ha (s,c,a,b); // name of module
    input a,b; // input declarations
    output s,c; // output declarations
    wire a,b,s,c;
    xor x1 (s,a,b); // instances for xor
    and a1 (c,a,b); // instances for and
endmodule // end of module
module test; // module test
    reg a,b; // input registers
    wire s,c; // output wires
    ha h1 (s,c,a,b); // instance for half adder
initial
begin
    $dumpvars; // dumping the variables for the wave file
    $dumpfile("x1.dmp");
    a=0;b=0;
    $monitor("a=%b,b=%b,c=%b,s=%b",a,b,c,s);
#5 a=0;b=1;
#5 a=1;b=0;
#5 a=1;b=1;
#5 $finish;
end
endmodule
```

Register Level Modeling

```
/******
```

Register level modeling of half adder

```
*****/
```

```
module ha(d,c,a,b); // module name
    input a,b;
    output d,c;
```

```

        wire a,b;
        reg d,c;
always @ (a or b)           // start for signals at 'a' or 'b'
begin
    if(a==1 && b==0)
        begin
            d=1;
            c=0;
        end
    else if(a==0 & b==1)
        begin
            d=1;
            c=0;
        end
    else if(a==1 & b==1)
        begin
            d=0;
            c=1;
        end
    else
        begin
            d=0;
            c=0;
        end
end
endmodule
module test;
    reg a,b;
    wire s,c;
    ha h1 (s,c,a,b);
initial
begin
    $dumpvars;
    $dumpfile("ha.dmp");
    a=1'b0;b=1'b0;
    $monitor("a=%b,b=%b,c=%b,s=%b",a,b,c,s);
    #5 a=1'b0;b=1'b1;
    #5 a=1'b1;b=1'b0;
    #5 a=1'b1;b=1'b1;
    #5 $finish;
end
endmodule

```

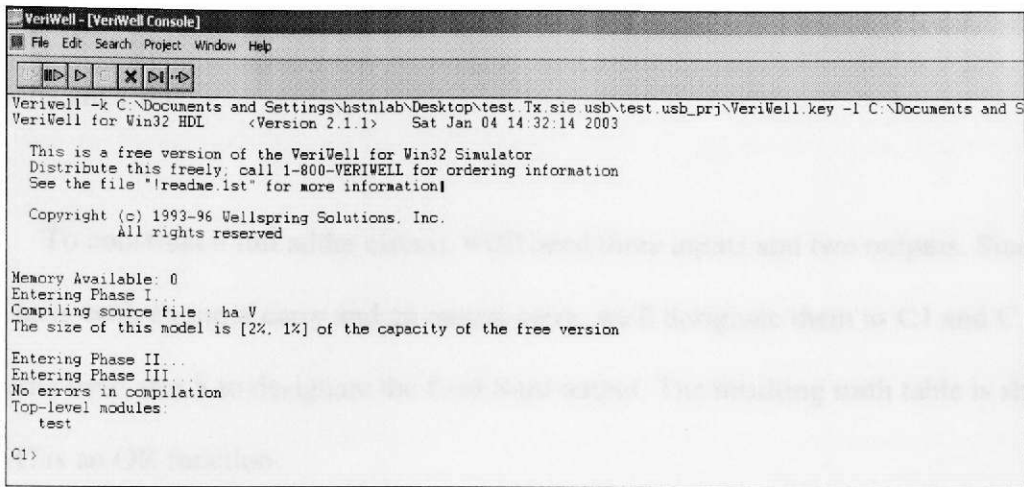


Figure 3.2. Veriwell Console for Half Adder

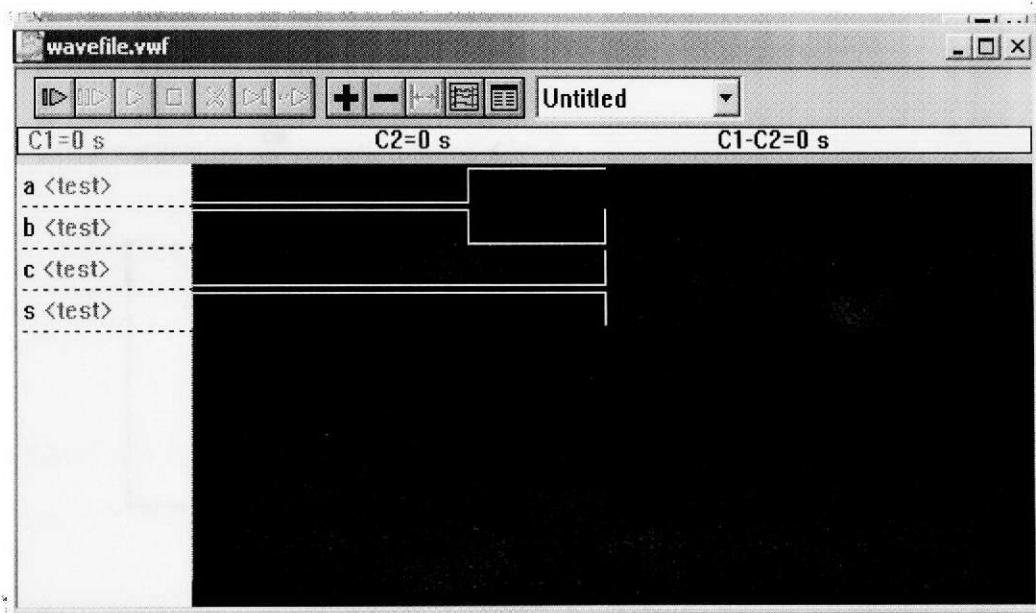


Figure 3.3 Waveform Analysis of Half Adder

3.2.3.2 Full Adder

The module ' Full Adder ' consists of the following

Input ports: ' a ', ' b 'and ' cin '

Out ports: ' c ' and ' d '

All ports are 1 bit

To construct a full adder circuit, we'll need three inputs and two outputs. Since we'll have both an input carry and an output carry, we'll designate them as C1 and C. At the same time, use S to designate the final Sum output. The resulting truth table is shown.

Here C is an OR function.

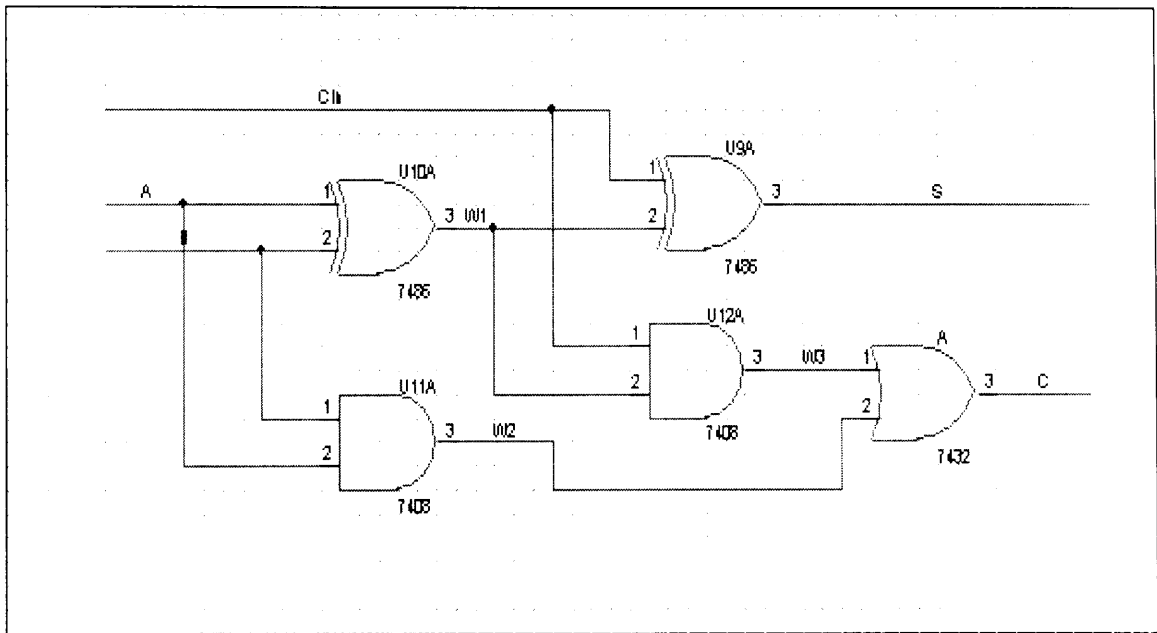


Figure 3.4 Full Adder [2]

Table 3.2 Full Adder Truth Table

A	B	CIN	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	0	1	1	0
1	1	1	1	1

Gate Level Modeling

```
/******
```

Gate level full adder

```
*****/
```

```
module FA (s,c,a,b,cin);           // module name
  input a,b,cin;                  // input ports
  output s,c;                     // output ports
  wire a,b,c,cin,s;              // internal wires
  wire w1,w2,w3;                 // creation of instances
  xor x1(w1,a,b);                 // creation of instances
  and a1 (w2,a,b);                // creation of instances
  xor x2(s,w1,cin);              // creation of instances
```

```

    and a2(w3,cin,w1);        // creation of instances
    or o1(c,w3,w2);         // creation of instances
endmodule

module test;                // module test
reg a,b,cin;
wire s,c;
FA f1(s,c,a,b,cin);
initial
begin
//$dumpvars;
//$dumpfile("x2.dmp");
a=0;b=0;cin=0;
$monitor("a=%b,b=%b,cin=%b,s=%b,c=%b",a,b,cin,s,c,$time);
#5 a=0;b=0;cin=1;
#5 a=0;b=1;cin=0;
#5 a=0;b=1;cin=1;
#5 a=1;b=0;cin=0;
#5 a=1;b=0;cin=1;
#5 a=1;b=1;cin=0;
#5 a=1;b=1;cin=1;
#5 $finish;
end
endmodule

```

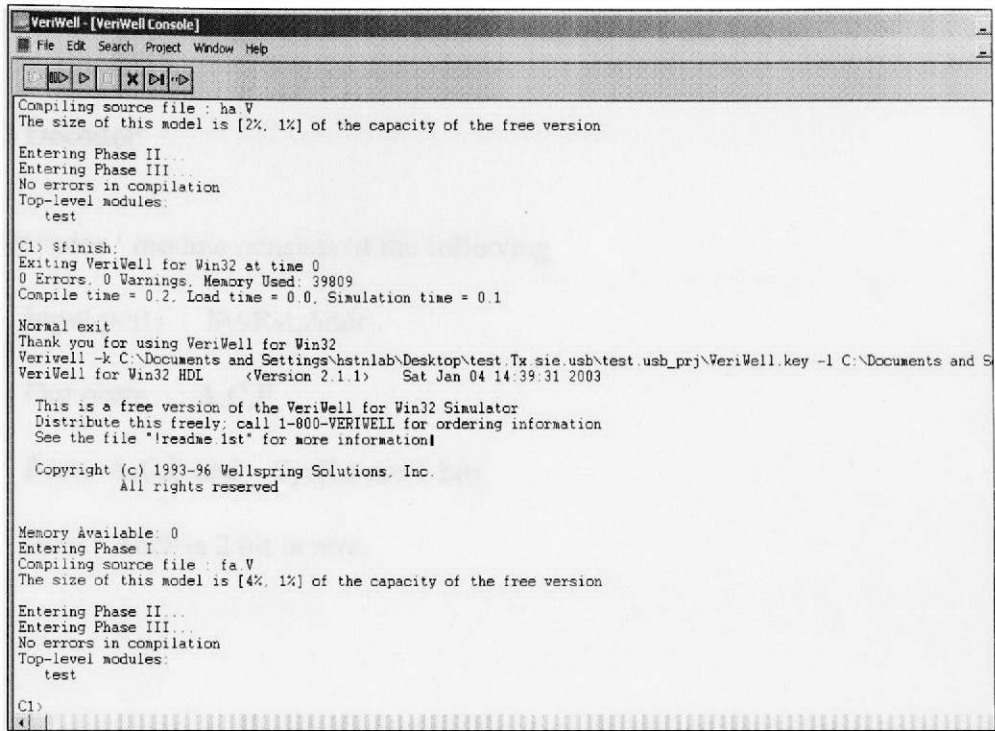


Figure 3.5 Veriwell Console for Full Adder

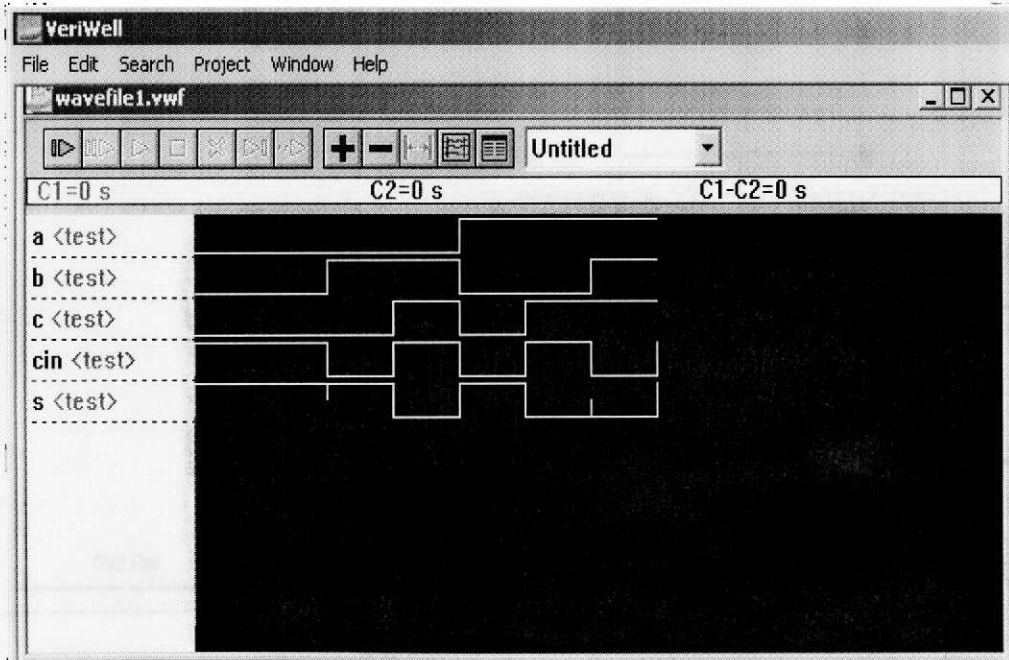


Figure 3.6 Waveform Analysis of Full Adder

3.2.3.3 Decoder

The 'Decoder ' module consists of the following

- Input ports : SysRst,Addr .
- Out ports : A,C,F
- Ports A,C,F and SysRst are 1 bits
- Port Addr is 2 bit in size.

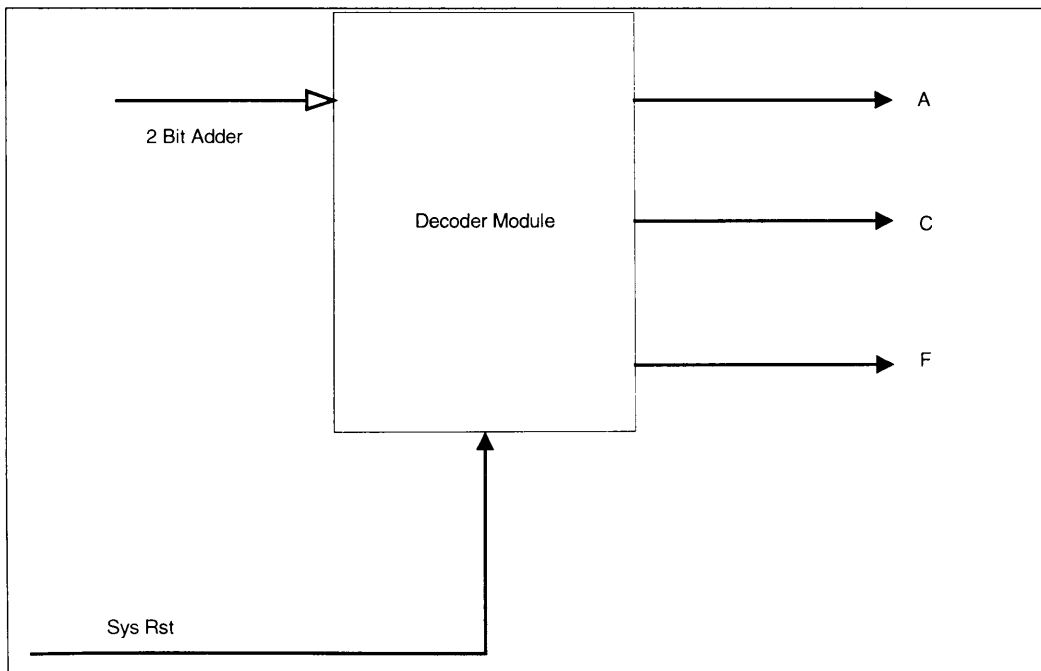


Figure 3.7 Block Diagram of Decoder

2-Bit Address Decoder

```
/**
 *
 */

2-bit address decoder

/**

module Deco (SysRst,Addr,A,C,F); //module name
input      SysRst;
input  [1:0] Addr;
output     A,C,F;
wire      SysRst;
wire  [1:0] Addr;
reg       A,C,F;
always    @ (SysRst or Addr)
begin
    if (!SysRst)
        begin
            A=1'b0;
            C =1'b0;
            F=1'b0;
        end
    else
        begin
            case (Addr)
2'b00: begin
            A=1'b1;
            C =1'b0;
            F=1'b0;
        end
2'b01: begin
            A=1'b0;
            C =1'b1;
            F=1'b0;
        end
2'b10: begin
            A=1'b0;
            C =1'b0;
            F=1'b1;
        end
default:begin
            A=1'b0;
            C =1'b0;
```

```

        F=1'b0;
    end
endcase
end
end
endmodule

module test;
reg      SysRst;
reg  [1:0] Addr;
wire     A,C,F;
Deco d1 (SysRst,Addr,A,C,F);
initial
begin

//$dumpvars(1,d1.Addr,d1.AddrEn,d1.CrcEn,d1.FifoEn);
$dumpvars;
$dumpfile("Deco.dmp");
Addr=2'b00;SysRst=0;
$monitor("SysRst=%b,Addr=%b,A=%b,C=%b,F=%b",SysRst,Addr,A,C,F);
#5 Addr=2'b00;SysRst=1;
#5 Addr=2'b01;
#5 Addr=2'b10;
#5 Addr=2'b11;
#5 $finish;
end
endmodule

```

```

VeriWell - k C:\Documents and Settings\hstnlab\Desktop\test Tx.sie usb\test usb_prj\VeriWell.key
VeriWell for Win32 HDL (Version 2.1.1) Sat Jan 04 14:47:54 2003

This is a free version of the VeriWell for Win32 Simulator
Distribute this freely; call 1-800-VERIWELL for ordering information
See the file "readme.lst" for more information

Copyright (c) 1993-96 Wellspring Solutions, Inc
All rights reserved

Memory Available 0
Entering Phase I...
Compiling source file : deco[1].usb.V
The size of this model is [4%, 4%] of the capacity of the free version
Entering Phase II...
Entering Phase III...
No errors in compilation
Top-level modules:
test

C1>
SysRst=0,Addr=00,A=0,C=0,F=0
SysRst=1,Addr=00,A=1,C=0,F=0
SysRst=1,Addr=01,A=0,C=1,F=0
SysRst=1,Addr=10,A=0,C=0,F=1
SysRst=1,Addr=11,A=0,C=0,F=0
Exiting VeriWell for Win32 at time 25
0 Errors, 0 Warnings, Memory Used: 44326
Compile time = 0.2, Load time = 0.0, Simulation time = 0.0

Normal exit
Thank you for using VeriWell for Win32

```

Figure 3.8 Veriwell Console for Decoder

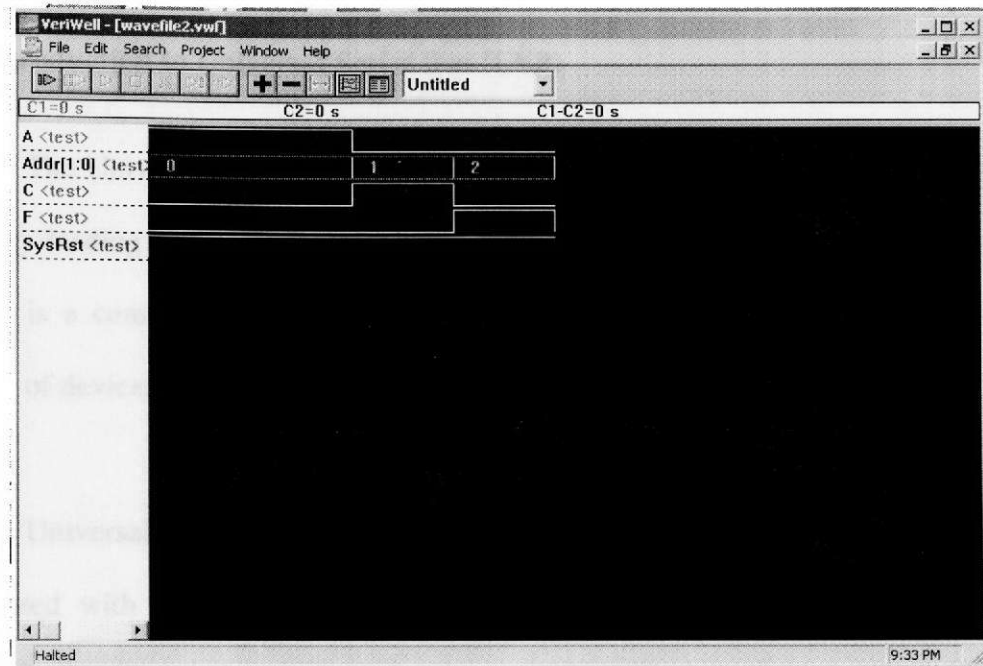


Figure 3.9 Waveform Analysis of a 2-bit Decoder

Chapter 4 Universal Serial Bus (USB)

4.1 Introduction to Universal Serial Bus (USB)

The Universal Serial Bus was originally developed in 1995 by many of the same industry leading companies currently working on USB 2.0. The Universal Serial Bus (USB) is a communications architecture that gives a PC the ability to interconnect a variety of devices.

Universal serial Bus in short called USB emerged as a result of the difficulties associated with the cost, configuration and attachment of peripheral devices in the personal computer environment. In short, USB creates a method of attaching and accessing peripheral devices that reduces overall cost, simplifies the attachment and configuration from the end-user perspective, and solves several technical issues associated with old style peripherals.

The major goal of USB was to define an external expansion bus, which makes adding peripherals to a PC as easy as hooking up a telephone to a wall-jack.

The program's driving Goals were ease-of-use and low cost.

- PC host controller hardware and software
- Robust connectors and cable assemblies
- Peripheral friendly master-slave protocols
- Expandable through multi-port hubs.

Universal Serial Bus (USB) has been around for a few years now, and USB ports are fitted to just about every computer now, the Operating Systems didn't have the required level on support until recently

The USB supports the following characteristics.

- Upto 127 devices on one port.
- USB supplies power to the peripherals, reducing the need for wall warts, power bricks and power stealing from the keyboard connector
- Full speed devices communicate with the PC at 12Mbps. Mice and keyboards etc. can communicate at a lower 1.5Mbps rate to reduce cost.
- Hot Pluggable.
- PlugNPlay - The PC recognises each device that is plugged in and loads the appropriate driver. If it's a new device for which it has no driver, and doesn't run with a generic driver, it prompts for a driver to be loaded.
- No confusing cabling - no null modem cables, handshaking lines to mess with etc.
- Supports 4 different data transfer types: - Isochronous, Control, Interrupt, Bulk

USB breaks away from the resource problems associated with legacy PC IO implementations. The resource constraints related to IO address space, IRQ (Interrupt Request) lines, and DMA (Direct Memory Access) channels no longer exist with the

USB implementation. Devices residing on the USB are assigned an address known only to the USB subsystem and this does not consume any system resources. The number of USB devices supported in a single implementation is limited in number to 127. USB devices typically contain a number of individual registers or ports that can be indirectly accessed by USB device drivers. These registers are known as USB device endpoints.

When a transaction is sent over the USB, all devices (except low speed devices) will see the transaction. Each transaction begins with a packet transmission that defines the type of transaction being performed along with the USB device and endpoint addresses. This addressing is managed by USB software, and other non-USB devices and related software within the system are not impacted by these addresses.

Every USB device must have an endpoint address zero that is reserved for configuration. Via endpoint zero, USB system software accesses USB devices descriptors from the device. These descriptors provide information necessary for identifying the device, specifying the number of endpoints, and the purpose of each. In this manner, system software can detect the device type or class and determine how the device is to be accessed

4.2 Universal Serial Bus (USB) System

4.2.1 Role of Host PC Hardware and Software

The role of the system software is to provide a uniform view of IO system for all applications software. It hides hardware implementation details so that application software is more portable. For the USB IO subsystem in particular, it manages the dynamic attach and detach of peripherals. This phase, called enumeration, involves communicating with the peripheral to discover the identity of a device driver that it should load, if not already loaded. A unique address is assigned to each peripheral during enumeration to be used for run-time data transfers. During run-time the host PC initiates transactions to specific peripherals, and each peripheral accepts its transactions and responds accordingly. Additionally the host PC software incorporates the peripheral into the system power management scheme and can manage overall system power without user interaction.

Figure1 shows the USB system in terms of its hardware and software configuration. USB software initiates all the transactions on the USB system. A USB driver acts as an interface between the device driver and the host controller when the device driver is communicating with its device. This software is responsible for translating client requests into one or more transactions that are directed to or from a target USB device.

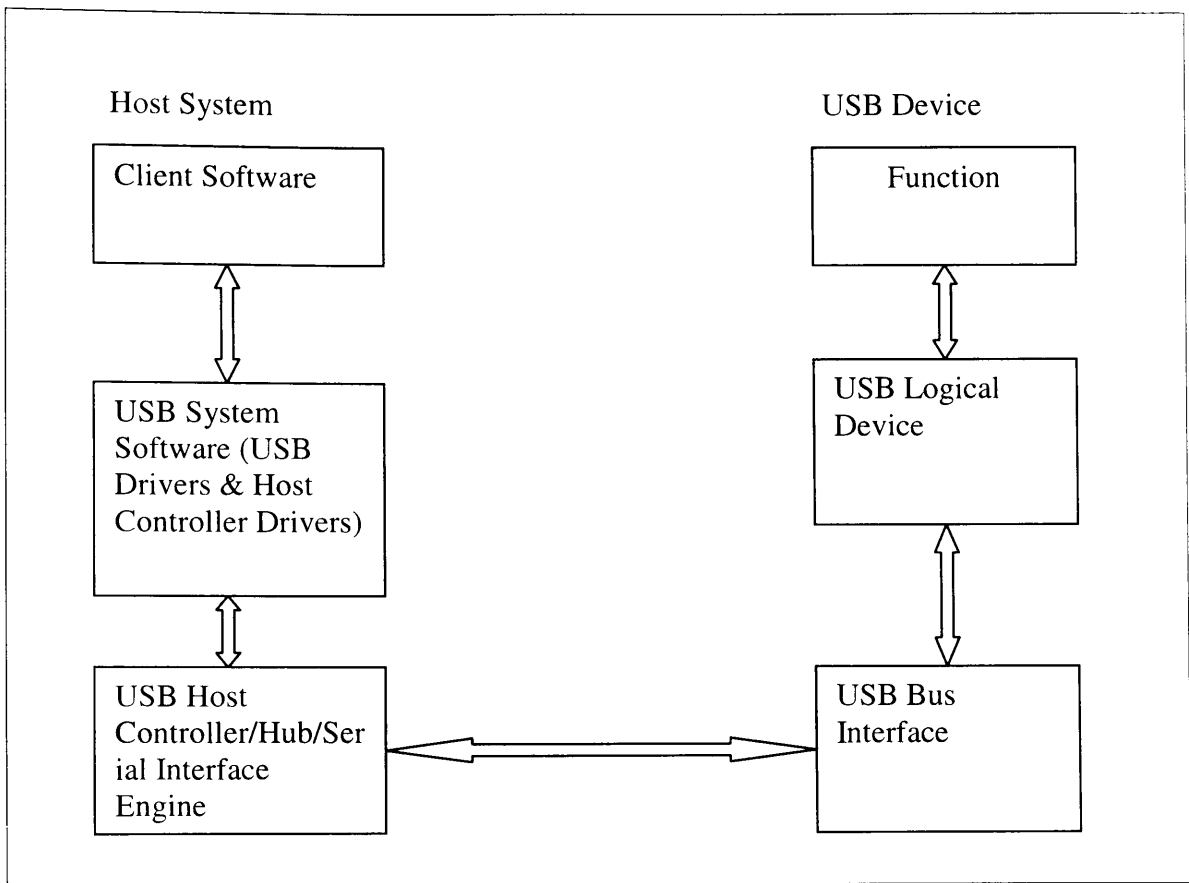


Figure 4.1 Communication Flow in a USB System

The SIE (Serial Interface Engine) is the link between the physical and logical components of the USB. The SIE is to the USB what the UART (Universal Asynchronous Receiver Transmitter) is to the RS-232 interface. The primary USB Hardware consists of the following USB Host Controller/Root Hub/Serial Interface Engine, USB Hubs and USB Devices. The primary USB Software includes USB Device Drivers, USB Driver and Host Controller Driver.

4.2.2 Universal Serial Bus Hardware

All communication on USB originates at the host under software control. The host hardware consists of the USB host controller, which initiates transactions over the USB system, and the root hub, which provides attachment points for USB devices. The host controller is responsible for generating the transactions that have been scheduled by the host software. The host controller driver, or HCD, software builds a linked list of data structures in memory that defines the transactions that are scheduled to be performed during a given frame. These data structures, called transfer descriptors, contain all of the information the host controller needs to generate the transactions. This information includes USB Device Address, Type of Transfer, Direction of Transfer, and Address of Device Driver's Memory Buffer.

The host controller performs writes to a target device by reading data from a memory buffer that is to be delivered to the target device. The host controller performs a parallel to serial conversion on the data, creates the USB transaction, and forwards it to the root hub to send over the bus.

If a read transfer is required, the host controller builds the read transaction and sends it to the root hub. The hub transmits the read transaction over the USB. The target device recognizes that it is being addressed and that data is being requested. The device then transmits data back to the root hub, which forwards the data on to the host controller. The host controller performs the serial to parallel conversion on the data and transfers the

data to the device driver's memory buffer. Transactions generated by the host controller are forwarded to the root hub to be transmitted to the USB. Consequently, every USB transaction originates at the root hub. The root hub provides the connection points for USB devices and performs the following key operations:

- Controls power to its USB ports
- Enables and disables ports
- Recognizes devices attached to each port
- Sets and reports status events associated with each port

In addition to the root hub, USB systems support additional hubs that permit extension of the USB system by providing one or more USB ports for attaching other USB devices. USB hubs may be integrated into devices such as keyboards or monitors, or implemented as stand-alone devices as shown in Fig. 3.2. Furthermore, hubs are bus powered (i.e., derive power for itself and all attached devices from the USB bus) or may be self-powered. Bus powered hubs are limited by the amount of power available from the bus and can therefore support a maximum of four USB ports.

Hubs contain two major functional elements:

- Hub Controller
- Repeater

The Hub Controller contains a USB interface, or serial interface engine (SIE). It also contains the descriptors that software reads to identify the device as a hub. The hub

controller gathers hub and port status information also read by the USB host software to detect the connection and removal of devices and to determine other status information. The controller also receives commands from host software to control various aspects of the hub's operation (e.g., powering and enabling the ports).

Bus traffic arriving at the hub must be forwarded on in either the upstream (toward the host) or downstream (away from the host) direction. Transmissions originating at the host will arrive on the hub's root port and must be forwarded to all enabled ports. When a target device responds to a host-initiated transaction it must transmit a response upstream, which the hub must forward from the downstream port to the root port

The hub controller contains a USB interface, or serial interface engine (SIE). It also contains the descriptors that software reads to identify the device as a hub. The hub controller gathers hub and port status information also read by the USB host software to detect the connection and removal of devices and to determine other status information. The controller also receives commands from host software to control various aspects of the hub's operation (e.g., powering and enabling the ports).

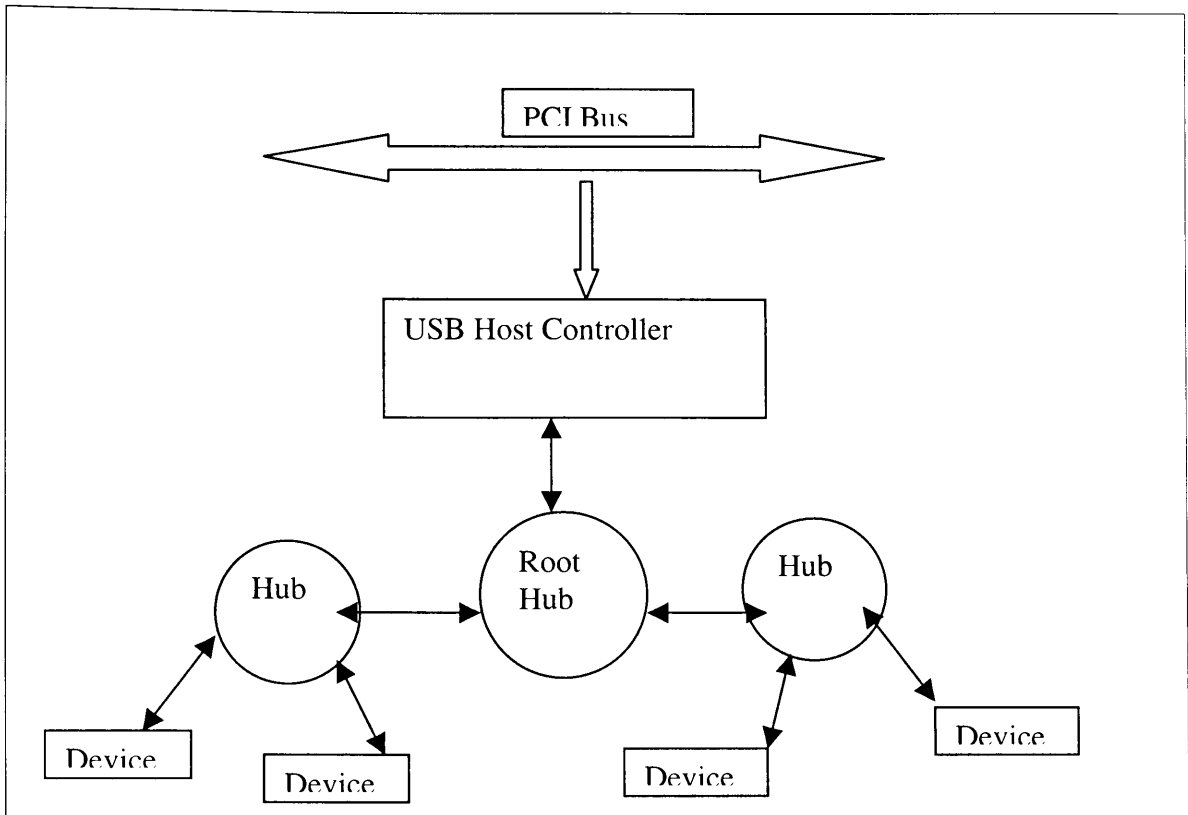


Figure 4.2 USB Hub Types

Bus traffic arriving at the hub must be forwarded on in either the upstream (toward the host) or downstream (away from the host) direction. Transmissions originating at the host will arrive on the hub's root port and must be forwarded to all enabled ports. When a target device responds to a host-initiated transaction it must transmit a response upstream, which the hub must forward from the downstream port to the root port.

4.2.3 Role of the Peripherals

All USB peripherals which are the USB devices are slaves that obey a defined protocol. They must react to request transactions sent from the host PC. The peripheral responds to control transactions that, for example, request detailed information about the device and its configuration. The peripheral sends and receives data to/from the host using a standard USB data format. This standardized data movement to/from the PC host and interpretation by the peripheral gives USB its enormous flexibility with little PC host software changes

USB devices contain descriptors that specify a given devices attributes and characteristics. This information specifies to host software a variety of features and capabilities that are needed to configure the device and to locate the USB client software driver. The USB device driver may also use device descriptors to determine additional information needed to access the device in the proper fashion. This mechanism is referred to as the Device Framework and must be understood by software in order to configure and access the device correctly. USB devices can be implemented either as high-speed or low-speed devices.

High-Speed devices see all transactions broadcast over the USB and can be implemented as full-feature devices. These devices accept and send serial data at the maximum 12Mb/s rate.

Low-speed devices are limited in not only throughput (1.5Mb/s) but also feature support. Furthermore, low-speed devices only see USB transactions that follow a preamble packet. Low-speed hub ports remain disabled during full-speed transactions, preventing full-speed bus traffic from being sent over low-speed cables. Preamble packets specify that the following transaction will be broadcast at low speed. Hubs enable their low-speed ports after detecting a preamble packet, permitting low-speed devices to see the low-speed bus activity.

4.2.4 Universal Serial Bus Software

4.2.4.1 Universal Serial Bus Device Drivers

USB device drivers (or client drivers) issue requests to the USB driver via IO Request Packets (IRPs). These IRPs initiate a given transfer to or from a target USB device. For example, a USB keyboard driver must initiate an interrupt transfer by establishing an IRP and supplying a memory buffer into which data will be returned from the USB keyboard. Note that the client driver has no knowledge of the USB serial transfer mechanisms.

4.2.4.2 Universal Serial Bus Driver

The USB driver knows the characteristics of the USB target device and how to communicate with the device via the USB. The USB driver detects the USB characteristics when it parses the device descriptors during device configuration. For

example, some devices require a specific amount of throughput during each frame, while others may only require periodic access every nth frame.

When an IRP is received from a USB client driver, the USB driver organizes the request into individual transactions that will be executed during a series of 1ms frames. The USB driver sets up the transactions based on its knowledge of the USB device requirements, the needs of the client driver, and the limitations/capabilities of the USB.

Depending on the operating environment, the USB driver may be shipped along with the operating system or added as an extension via a loadable device driver.

4.2.4.3 Universal Serial Bus Host Controller Driver

The USB host controller driver (HCD) schedules transactions to be broadcast over the USB. The host controller driver schedules transactions by building a series of transaction lists. Each list consists of pending transactions targeted for one or more of the USB devices attached to the bus. A transaction list, or frame list, defines the sequence of transactions to be performed during each 1ms frame. The USB host controller executes these transaction lists at 1ms intervals. Note that a single block transfer requested by a USB client may be performed as a series of transactions that are scheduled and executed during consecutive 1ms frames. The actual scheduling depends on a variety of factors including; the type of transaction, transfer requirements specified by the device and the transaction traffic of other USB devices.

The USB host controller initiates transactions via its root hub or hubs. Each 1ms frame begins with a start of frame (SOF) transaction and is followed by the serial broadcast of all transactions contained within the current list. For example, if one of the requested transactions is a request to transfer data to a USB printer, the host controller would obtain the data to be sent from a memory buffer supplied by the client software and transmit the data over the USB. The hub portion of the controller converts the requested transactions into the low level protocols required by the USB.

4.3 Serial Interface Engine

USB implementation is via a layered model of software and hardware functionality that is reflected in both transmitting and receiving devices in a manner similar to the Open Systems Interconnection (OSI) or Transmission Control Protocol/Internet Protocol (TCP/IP) models. Actual communications flow occurs through the layers in a transmitting system, across a physical link to a receiving system, and then through a similar stack of hardware and software layers in the receiver. However, there is also a logical interconnection between corresponding layers in the transmitter and receiver. To some degree, these protocols provide a level of abstraction where successful communication can be achieved based on knowledge of adjacent and corresponding layers without having to deal with the complexity of the entire model.

A key difference between the USB model and the more familiar communications protocols is the level on which the subscribers to the protocol are communicating. TCP/IP

and OSI can be considered macro protocols that allow connection of multiple processors across an external network, while the USB can be considered a micro protocol that allows peripherals to interconnect with a processing platform on its self contained network. A second difference is the network implementation. OSI and TCP/IP compliant communications can be implemented under multiple networking protocols, the USB format provides for a token-based network that provides full requested bandwidth for its users, but denies entry to the net for new users if their bandwidth requirement cannot be met.

While the USB specification does not limit bus implementation to any single processor type or electrical interconnection format, USB controllers most commonly reside on the Peripheral Component Interconnect (PCI) bus of Pentium Class or Power PC Macintosh computers. The required circuitry is normally included as a built-in feature of the motherboard, but PCI add-in card USB ports are available for older X86 based PC's.

The SIE is the border between the physical and logical components of the USB. The SIE is to the USB what the UART is to the RS-232 interface. The SIE is built into most USB micro-controllers along with a USB transceiver.

The SIE is commonly called upon to perform the following list of tasks:

- Recognition of bits and proper transaction sequence.

- Generation and detection of start of data bits, end of data bits, reset, and resume signals.
- Separation of clock and data.
- Generating and verifying Cyclic Redundancy Checks (CRC) for data.
- Performing parallel to serial and serial to parallel conversion.

Chapter 5 Module Design

5.1 Design Features of the Communication System

The communication device is a logical block, which is implemented in the USB as a Serial Interface Engine in this thesis. It is at the interface between Host and the peripherals, but is also an independent block by itself, which can be interfaced with devices where transmission of 16 bits of data from host to the peripheral devices is needed.

- The communication block does the following.
- Recognition of bits and proper transaction sequence.
- Generation and detection of start of data bits, end of data bits, reset and resume signals
- Separation of clock and data.
- Generating and verifying Cyclic Redundancy Checks (CRC) for error handling of data.
- Performing parallel to serial and serial to parallel conversions.
- Storing the data in buffers and FIFO's.
- Synchronizing the host speed with peripheral devices speed.
- To separate address, data and CRC bits from the stream of bits coming from the host i.e., a processor.

- Controlling all the operations and conversions with respect to clock.
- Using techniques like bitstuffing to ensure reliability of the data bits.
- NRZ- I coding is employed on the bits to improve the performance.

5.2 Techniques used for Reliable and Efficient Transmission

The transmission of a stream of bits from one device to another across a transmission link requires a great deal of cooperation and the agreement between the two sides. One of the most fundamental requirements is synchronization. The receiver must know the rate at which the bits are received so that it can sample the line at regular intervals to determine the value at each received bit.

5.2.1 Bit-Stuffing

The bit stuffing is the insertion of noninformation bits into data. It is the practice of adding bits to a stream of data. Bit-stuffing is required by many network and communications protocols for the following reasons:

- To prevent data being interpreted as control information. For example, many frame-based protocols, such as HDLC (high level data link control), signal the beginning and end of a frame with six consecutive 1 bits. Therefore, if the actual data being transmitted has six 1 bits in a row; a zero is inserted after the first 5 so that the data is not interpreted as a frame delimiter. Of course, on the receiving end, the stuffed bits must be discarded.
- For protocols that require a fixed-size frame, bits are sometimes inserted to make the frame size equal to this set size.

- For protocols that required a continuous stream of data, zero bits are sometimes inserted to ensure that the stream is not broken.
- Stuffed bits should not be confused with overhead bits.
- In data transmission, bit stuffing is used for various purposes, such as for synchronizing bit streams that do not necessarily have the same or rationally related bit rates, or to fill buffers or frames. The location of the stuffing bits is communicated to the receiving end of the data link, where these extra bits are removed to return the bit streams to their original bit rates or form. Bit stuffing may be used to synchronize several channels before multiplexing or to rate-match two single channels to each other.
- The receiver needs to be able to determine what the relationship of the bits in the received stream have to one another, that is, what the logical units of transfer are, and where each received bit fits into the logical units.

5.2.2 Error checking using CRC (Cyclic Redundancy Check)

Regardless of the design of the transmission system, there will be errors, resulting in the change of one or more bits in a transmitted frame. There are transmission impairments. This can be defined as probabilities with respect to errors in transmitted frames. The error detection techniques operate on the principle that the probability of occurrence of errors in the communication system is very low compared to the magnitude of information signals that are transmitted in the communication system. For a given frame of bits additional code of bits that constitute the error detecting code are added by

the transmitter. This code is calculated as the function of the other transmitted bits the receiver performs the same calculation and compares the two results. A detected error occurs if and only if there is a mismatch.

One of the most common and one of the powerful error detecting codes is the Cyclic Redundancy Check also known as CRC, which can be described as follows. Given a k-bit block of bits, or message, the transmitter generates an n bit sequence, know as frame check sequence (FCS), so that the resulting frame, consists of k+n bits it is this that is transmitted to the receiver and the receiver calculates CRC on the data received. if there is no error CRC calculated in the receiver matches with that of the transmitter.

5.2.3 NRZ (Non Return to Zero)

A binary encoding scheme in which a signal parameter, such as electric current or voltage, undergoes a change in a significant condition or level every time that a "one" occurs, but when a "zero" occurs, it remains the same, that is no transition occurs. The transitions could also occur only when "zeros" occur and not when "ones" occur. If the significant condition transition occurs on each "zero," the encoding scheme is called "non-return-to-zero space" (NRZ-S). NRZ-M and NRZ-S signals are technically interchangeable, one is the logical "NOT" (inverse) of the other. It is necessary for the receiver to have prior knowledge of which scheme is being used. Without such knowledge, it is impossible for the receiver to interpret the data stream correctly; its output may be the correct data stream or the logical inverse of the correct data stream.

5.3 Design Description of Serial Interface Engine

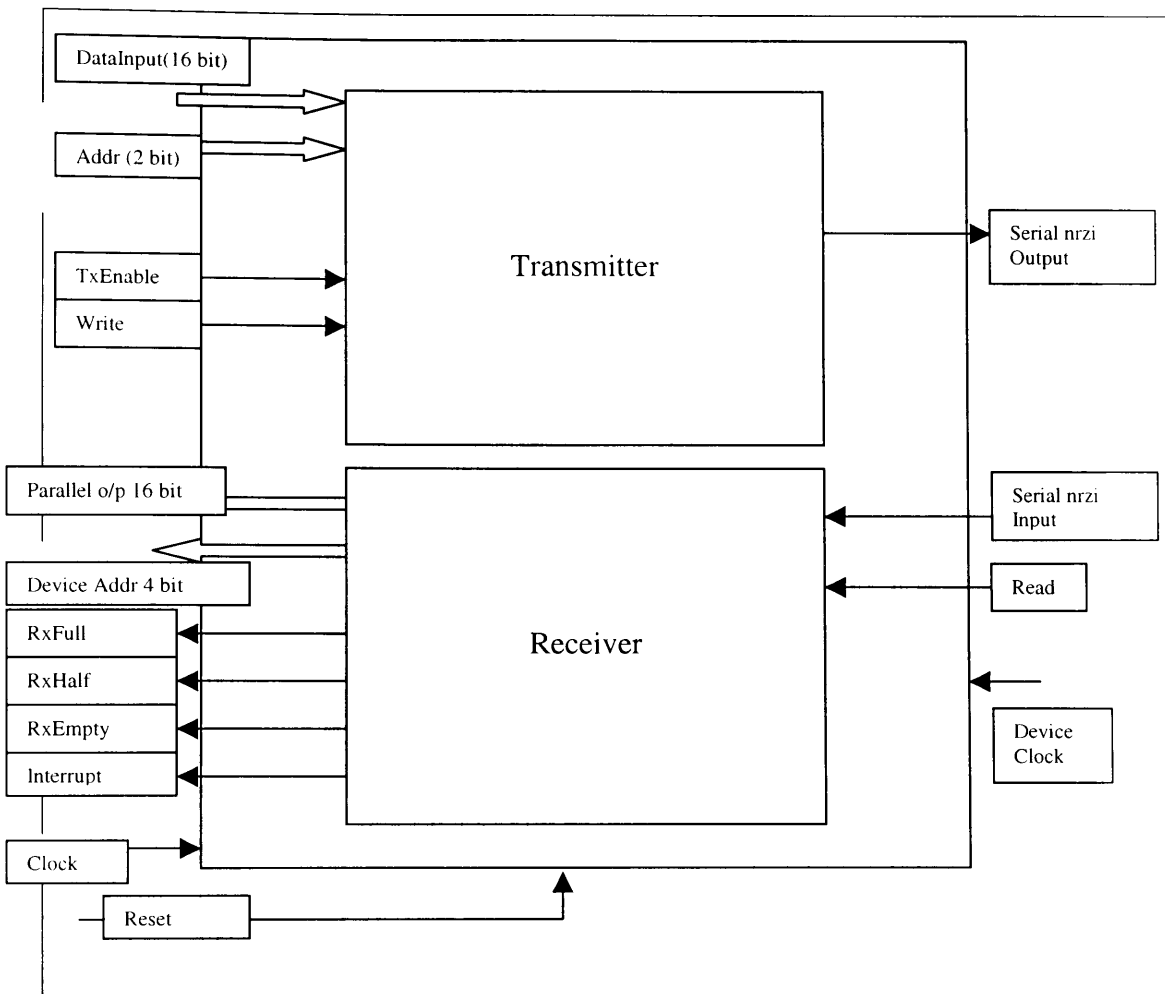


Figure 5.1 Block Diagram of Serial Interface Engine

The device which is a Serial Interface Engine is the main module, has two modules which comprises it these functional modules are the Transmitter and the Receiver.

- Transmitter: the data from the host that has to be sent across to the connected USB devices is sent to this block. This block has many functions, which process the data and coordinate the movement within the block synchronized with the host processor before sending it over to the next block. This module consists of the following sub modules,
 - Decoder
 - Clock
 - Buffer
 - TxFIFO (Transmitter side First In First Out)
 - Control Logic
 - Multiplexer
 - CRC Generator
 - Parallel to Serial Converter
 - NRZ-I coding
 - Bitstuffing

- Receiver: This module consists of the following sub modules,
 - D-NRZI
 - DeBitstuffing decoder
 - Serial to Parallel Converter
 - Demultiplexer
 - Control Logic

- Address Decoder
- Timer
- CRC Generator
- CRC Comparator
- Receiver side First In First Out

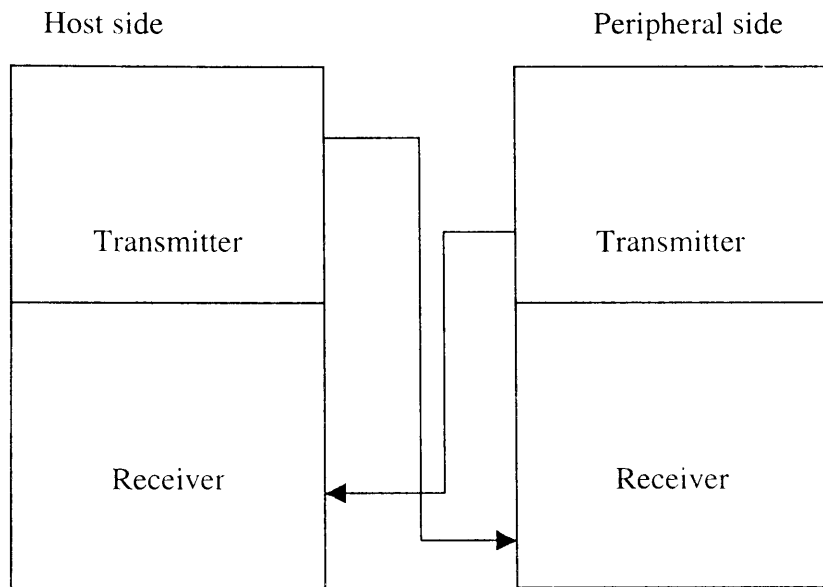


Figure 5.2 Design of the Block Diagram of Full Duplex

Chapter 6 Transmitter and Receiver

6.1 Transmitter

6.1.1 Decoder

Module name is Decoder. The Binary-Output Decoder block produces a binary message vector (signal) from a binary input vector (signal) that it receives from the host processor.

The module consists of:

The input ports are SysRst (System Reset)

Addr (2 bit, Address)

The output ports are AddrEn (Address Enable Signal)

FifoEn (FIFO Enable Signal)

CrcEn (CRC Enable Signal)

The logic used is shown in the form of table, which shows the various signals generated for the different addr signals.

Table 1. Relation between address and the signals generated

Addr	Signal generated
00	Address Enable
01	CRC Enable
10	FIFO Enable

Source code for Decoder Tx_Decoder

```
/******  
Module Name is Decoder  
*****/  
module Deco (SysRst,Addr,AddrEn,CrcEn,FifoEn);  
  
//***** Input/Output Declarations *****  
input SysRst;  
input [1:0] Addr;  
  
output AddrEn,CrcEn,FifoEn;  
  
//***** Wire/Reg Declarations *****  
wire SysRst;  
wire [1:0] Addr;  
  
reg AddrEn,CrcEn,FifoEn;  
  
//***** Functional Description *****  
  
always @ (SysRst or Addr)  
begin  
    if (!SysRst)  
        begin  
            AddrEn=1'b0;  
            CrcEn =1'b0;  
            FifoEn=1'b0;  
        end  
    else  
        begin  
            case (Addr)  
2'b00: begin  
            AddrEn=1'b1;  
            CrcEn =1'b0;  
            FifoEn=1'b0;  
            end  
2'b01: begin  
            AddrEn=1'b0;  
            CrcEn =1'b1;  
            FifoEn=1'b0;  
            end  
2'b10: begin  
            AddrEn=1'b0;  
            CrcEn =1'b0;  
            FifoEn=1'b1;  
            end  
            default:begin  
            AddrEn=1'b0;  
            CrcEn =1'b0;  
            FifoEn=1'b0;  
            end  
        endcase  
end
```

```
end
end
endmodule
```

6.1.2 Buffer

In electronic modules buffer is used to temporary store data, when speed of the processor is slower than the data incoming in order to constantly feed the data with out overwhelming the processor.

The module Buffer consists of:

Input ports are DataIn (16 bit Data Input), Wr (Write Signal),
 AddrEn (Address Enable Signal), CrcEn (CRC Enable Signal),
 FifoEn (FIFO Enable Signal), SysClk (System Clock),
 SysRst (System Reset)

Output ports are AddrReg (16 bit Address Register),
 CrcReg (16 bit CRC Register), FifoSel (FIFO Select Signal)

6.1.2.1 Logical functioning of the Buffer

When there is a system reset all outputs are set to zero. The input data is stored in Address Register when both Write Signal and Address Enable signal are high. The input data is stored into CRC register. When both Write Signal and CRC Enable Signal are high. The FIFO Select Signal is generated when both Write Signal and FIFO Enable signal are high. For the rest of the Signals same data is maintained until a respective change occurs.

Source code for Tx_Buffer.V

```
/******  
Tx_Buffer  
*****/  
module Buffer (SysClk,Wr,DataIn,AddrEn,CrcEn,FifoEn,SysRst,CrcReg,AddrReg,FifoSel);  
  
//***** Input/Output Declarations *****  
input      SysClk,Wr,AddrEn,CrcEn,FifoEn,SysRst;  
input  [15:0]  DataIn;  
  
output  [15:0]  CrcReg,AddrReg;  
output      FifoSel;  
  
//***** Wire/Reg Declarations *****  
  
wire      SysClk,Wr,AddrEn,CrcEn,FifoEn,SysRst;  
reg       FifoSel;  
wire  [15:0]  DataIn;  
  
reg  [15:0]  CrcReg,AddrReg;  
  
//***** Functional Description *****  
  
always  @ (SysClk or SysRst)  
begin  
    if (!SysRst)  
        begin  
            AddrReg<=16'h0000;  
            CrcReg <=16'h0000;  
            FifoSel<=1'b0;  
        end  
    else begin  
        if (Wr)  
            begin  
                if (AddrEn)  
                    begin  
                        AddrReg<=DataIn;  
                        //$strobe ("*****AddrReg=%h",AddrReg);  
                    end  
  
                else if(CrcEn)  
                    CrcReg <=DataIn;  
                else if(FifoEn)  
                    FifoSel<=DataIn[0];  
                else  
                    begin  
                        AddrReg<=AddrReg;  
                        CrcReg <=CrcReg;  
                        FifoSel<=FifoSel;  
                    end  
            end  
    end  
end
```

```

        else
        begin
            AddrReg<=AddrReg;
            CrcReg <=CrcReg;
            FifoSel<=FifoSel;
        end
    end
end
endmodule

```

6.1.3 Clock

Module Name is Tx_Clock. File Name is Tx_Clock.V.

Clock Produces continuous digital clock pulses this does not need any input. This is the system clock module.

Input Ports are none. Output Ports are SysClk

Source Code for Clock Module Tx_Clock.V

```

/*****
    Module Name is Tx_Clock
    *****/
module Tx_Clock (SysClk);
/*****INPUT/OUTPUT DECLARATION*****/
output    SysClk;
/*****WIRE/REG DECLARATION*****/
reg       SysClk;
/*****FUNCTIONAL
DESCRIPTION*****/
initial
begin
    SysClk=1'b1;
    //$dumpvars;
    //$dumpfile("xx.dmp");
    //$monitor("clk1=%b",clk1);
#2000 $finish;
end

always #5 SysClk=~SysClk;

endmodule

```

6.1.4 First In First Out

Module Name is First In First Out. File Name is Tx_FIFO.V

FIFO is made up of 16 registers also called as 16x16 bit FIFO, which stores the data before it is sent out to the multiplexer for transmission. There are three signals, which determine the state of the data flowing in to FIFO. They are empty signal, half signal and full signal.

Input Ports are FifoIn (16 bit FIFO Input), FifoSel (FIFO Select Signal),
 Eoc (End of Conversion Signal), Wr (Write Signal),
 SysClk (System Clock), SysRst (System Reset)

Output Ports are FifoOut (16 bit FIFO Output), Full (FIFO Full Signal),
 Empty (FIFO Empty Signal), Half (FIFO Half Signal)

Source code for Tx_FIFO

```
/******  
Module Name   :FirstInFirstOut (FIFO)  
*****/  
module        Tx_FIFO  
              (FifoIn,FifoOut,SysClk,Eoc,Wr,SysRst,FifoSel,Full,Empty,Half);  
//*****INPUT/OUTPUT DECLARATION*****  
  
input         [15:0]       FifoIn;  
input                     SysClk,SysRst,Wr,FifoSel,Eoc;  
  
output        [15:0]       FifoOut;  
output                     Full,Empty,Half;  
  
//*****WIRE/REG DECLARATION*****  
  
wire         [15:0]       FifoIn;  
wire                     SysClk,SysRst,Wr,FifoSel,Eoc;  
  
reg         [15:0]       FifoOut;  
reg                     Full,Empty,Half;  
  
reg                     [15:0] FIFO [0:15];  
reg         [3:0]        Rdptr,Wrptr;  
integer       [4:0]       Count;
```

```
/**FUNCTIONAL
```

```
DESCRIPTION**
```

```
always @ (posedge SysClk or negedge SysRst)
```

```
begin
```

```
    if (!SysRst)
        Wrptr=4'd0;
    else if (Wr && FifoSel)
        Wrptr=Wrptr+1;
    else
        Wrptr=Wrptr;
```

```
end
```

```
always @ (posedge SysClk)
```

```
begin
```

```
    if (Wr && FifoSel)
        begin
            FIFO[Wrptr]=FifoIn;
            // $display("Fifo[%d]=%h",
FifoSel=%b",Wrptr,FIFO[Wrptr],FifoSel);
            Count =Count+1;
        end
```

```
    else
        FIFO[Wrptr]=FIFO[Wrptr];
```

```
end
```

```
always @ (posedge Eoc or negedge SysRst)
```

```
begin
```

```
    if (!SysRst)
        begin
            Rdptr=4'd0;
            Count=0;
        end
```

```
    else
        begin
            FifoOut=FIFO[Rdptr];
            Rdptr =Rdptr+1;
            Count =Count-1;
        end
```

```
end
```

```
always @ (Count or SysRst)
```

```
begin
```

```
    if (!SysRst)
        begin
            Full =1'b0;
            Empty=1'b0;
            Half =1'b0;
        end
```

```
    else if (Count==0)
        begin
            Full =1'b0;
            Empty=1'b1;
            Half =1'b0;
        end
```

```
    else if (Count>=15)
        begin
            Full =1'b1;
```

```

                                Empty=1'b0;
                                Half =1'b0;
                                end
else if      (Count==8)
                                begin
                                Full =1'b0;
                                Empty=1'b0;
                                Half =1'b1;
                                end
                                else
                                begin
                                Full =1'b0;
                                Empty=1'b0;
                                Half =1'b0;
                                end
end
endmodule

```

6.1.5 Control Logic

Module is Control Logic. File Name is Tx_Control.V .

This module generates three-control signals Select Address Signal, Select FIFO Signal and Select CRC signal.

Input Ports are TxEn (Transmitter Enable), Eoc (End of Conversion Signal), Empty (Empty Signal)

Output Ports are SelAddr (Select Address Signal), SelFifo (Select FIFO Signal), SelCrc (Select CRC Signal)

Source Code for Control Logic Tx_Control.V

```

/*****
Module Name is Control Logic
*****/
module Tx_Control (TxEn,Empty,Eoc,SelAddr,SelFifo,SelCrc);

//***** Input/Output Declarations *****/
input TxEn,Empty,Eoc;
output SelAddr,SelFifo,SelCrc;

//***** Wire/Reg Declarations *****/

```

```

wire          TxEn,Empty,Eoc;
wire          SelAddr,SelFifo,SelCrc;
wire          w1;

//***** Functional Description *****/
assign       w1  =~Empty;
assign       SelAddr=(w1 & TxEn);
assign       SelFifo=(w1 & Eoc);
assign       SelCrc = (Eoc & Empty);
endmodule

```

6.1.6 Multiplexer

Module Name is Multiplexer. File Name is Tx_Mux.V.

This module multiplexer multiplexes data from different modules that it selects as the input data. Such as address from the address register, data from the FIFO and CRC bits from the CRC module. At all other times the data remains the same.

Input Ports are DataAddr (16bit Data Address), FifoOut (16 bit FIFO Output),
 CRC (4 bit CRC), SelAddr (Select Address Signal),
 SelFifo (Select FIFO Signal), SelCrc (Select CRC Signal),
 SysRst (System Reset)

Output Ports are PDataIn (16 bit Parallel Data Input)

Source Code for Multiplexer Module Tx_Mux.V

```

/*****
Module Name :Multiplexer
*****/
module Tx_Mux (DataAddr,FifoOut,CRC,PDataIn,SelAddr,SelFifo,SelCrc,SysRst);

//***** Input/Output Declarations *****/
input [15:0] DataAddr,FifoOut;
input [3:0] CRC;
input SelAddr,SelFifo,SelCrc,SysRst;

output [15:0] PDataIn;

//***** Wire/Reg Declarations *****/
wire [15:0] DataAddr,FifoOut;

```

```
wire [3:0] CRC;
wire SelAddr,SelFifo,SelCrc,SysRst;
reg [15:0] PDataIn;
```

```
/** Functional Description **/
```

```
always @ ( SysRst or DataAddr or FifoOut or CRC or SelAddr or SelFifo or SelCrc)
begin
    if (!SysRst)
        PDataIn=16'hzzzz;
    else if(SelAddr)
        PDataIn=DataAddr;
    else if(SelFifo)
        PDataIn=FifoOut;
    else if(SelCrc)
        begin
            PDataIn[3:0]=CRC;
            PDataIn[15:4]=12'hzzzz;
        end
    else PDataIn=PDataIn;
end
endmodule
```

6.1.7 CRC Generator

Module Name is Cyclic Redundancy Check (CRC) Generator. File Name is Tx_Crc.V.

This is the module that does the error handling by a technique called as CRC (Cyclic Redundancy Checks) creating Cyclic Redundancy Check bits. The CRC Generator first calculates the CRC bits for the first 16 bits of FIFO output and the CRC bits for the next 16 bit FIFO Output is calculated with respect to the first generated CRC bits. In this way the process continues and finally 4 CRC bits are generated for whole data and sent to the Parallel to Serial Converter for transmission.

Input Ports are FifoOut (16 bit FIFO Output), CRC_Reg (16 bit CRC Register).
Eoc (End of Conversion Signal), SysRst (System Reset)

Output Ports are CRC (4 bit CRC)

Source Code for CRC Generator Module Tx_Crc.V

```

/*****
Module Name is Tx_CRC Calculator
*****/
module CrcGen (CRC,FifoOut,SysRst,CRC_Reg,Eoc);
/*****INPUT/OUTPUT DECLARATION*****/
input [15:0] FifoOut; //Message coming from FIFO
input [15:0] CRC_Reg; // Generator polynomial value
input SysRst,Eoc;

output [3:0] CRC;
/*****WIRE/REG DECLARATION*****/
wire [15:0] FifoOut;
wire SysRst,Eoc;
wire [15:0] CRC_Reg;

reg [3:0] CRC;

/*****Internal Wire/Reg Declaratio *****/
reg [4:0] GenPoly; //Generator Polynomial value (5 bit)
reg [3:0] Temp;

/*****FUNCTIONAL DESCRIPTION*****/
always @ (posedge Eoc or negedge SysRst)
begin
    GenPoly=CRC_Reg[4:0];
    if (!SysRst)
        Temp =4'd0;
    else
        begin
            CRC =CRC_Cal(FifoOut,Temp,GenPoly);
            Temp =CRC;
        end
end

function [3:0] CRC_Cal;
parameter Zero=5'b00000;

input [15:0] FifoOut;
input [3:0] Temp;
input [4:0] GenPoly;
/***** Internal Reg Declaratio *****/
reg [20:1] msgtemp;
```



```

reg                [4:0]                Temp1;
reg                [4:0]                Rem;
integer            i;

begin

                                msgtemp={FifoOut,Temp};

                                //display("@@@@@msgtemp=%b",msgtemp);
                                Temp1=msgtemp[20:16];
                                //display("*****Temp=%b",Temp1);
                                for(i=15;i>0;i=i-1)
                                begin
                                if      (GenPoly<=Temp1)
                                begin
                                Rem=GenPoly ^ Temp1;
                                //display("*****Rem(Result)=%b",Rem);
                                Rem=Rem << 1;
                                //display("*****Rem(after removing
MSB)=%b",Rem);

                                Rem[0] = msgtemp[i];
                                //display("*****Rem(after appending M(x)
bit)=%b",Rem);

                                end
                                else
                                begin
                                Rem=Temp1^Zero;
                                //display("*****Rem(Result)=%b",Rem);
                                Rem=Rem<< 1;
                                //display("*****Rem(after removing
MSB)=%b",Rem);

                                Rem[0]=msgtemp[i];
                                //display("*****Rem(after appending M(x)
bit)=%b",Rem);

                                end
                                //display("(Remainder=%b",Rem);
                                Temp1=Rem;
                                end
                                CRC_Cal=Rem[3:0];

                                //display("^^^^^^^^^^^^^^^^CRC_Cal=%b",CRC_Cal);

end
endfunction
endmodule

```

6.1.8 Parallel to Serial Converter

Module Name is Parallel to Serial Bit Converter. File Name is Tx_PtoS.V.

This Module is converting the Parallel data in to serial data for transmission .It employs a buffer to store the 16 bit parallel data temporarily before it is converted to serial data bits for transmission.

Input Ports are PDataIn (16 bit Parallel Data Input),
 SelAddr (Select Address Signal), SelCrc (Select CRC Signal),
 Start (Start Signal), Clk (Clock synchronizing with device),
 SysRst (System Reset)

Output Ports are Sout (Serial Output), Eoc (End of Conversion Signal)

Source Code for Parallel to Serial Converter Module Tx_PtoS.V

```
/******  
Module Name is Parallel to Serial Bit Converter  
*****/  
module Tx_PtoS (PDataIn,nrziout,Start,SysRst,Clk,Eoc,SelCrc,SelAddr);  
  
//*****INPUT/OUTPUT DECLARATION*****  
input [15:0] PDataIn;  
input Clk,SysRst,Start,SelCrc,SelAddr;  
  
output nrziout,Eoc;  
  
//*****WIRE/REG DECLARATION*****  
wire [15:0] PDataIn;  
wire Clk,SysRst,Start,SelCrc,SelAddr;  
reg Sout,ref,nrziout,Eoc;  
reg [15:0] Buffer;  
integer i,d;  
//*****FUNCTIONAL  
DESCRIPTION*****  
always @ (negedge Clk)  
begin  
    if (Start | SelAddr)  
    begin  
        Buffer<=PDataIn;  
        Eoc <=1'b0;  
        i <=0;
```

```

        end
    else if (SelCrc)
    begin
        Buffer <=PDataIn;
        Eoc <=1'b0;
        i <=0;
    end

    else
    Buffer <=Buffer;
end
always @ (posedge Clk or negedge SysRst)
begin
    if (!SysRst)
    begin
        Buffer=16'hzzzz;
        Eoc =1'b0;
        i =0;
        Sout =1'bz;
        ref =0;
        nrziout=1'bz;
        d =0;
    end
    else
    begin
        if(d<6)
        begin
            Sout=Buffer[i];
            i =i+1;
            d =d+1;
            if(i==16)
            begin
                Eoc=1'b1;
                i =0;
            end
        end
        else
            Eoc=1'b0;
    end

    else
    begin
        Sout=1'b0;
        d =0;
    end
    end
    if (Sout==0||Sout==1)
    begin
        if (Sout==0)
        begin
            nrziout=~ref;
            ref =nrziout;
        end
        else
        begin
            nrziout=ref;
            ref =nrziout;
        end
    end
end

```

```

        end
    end
    else
        nrziout=nrziout;
    end
endmodule

```

6.1.9 Transmitter Module

Module Name is Transmitter. File Name is Tx.V.

Transmitter is the collection of modules that integrates all the modules explained above in order to process the data from the host to make it error free for transmission. it Synchronizes the working of various modules .

Input Ports are DataIn (16 bit Input Data), Addr (2 bit Address),
 TxEn (Transmitter Enable), Wr (Write Signal),
 SysClk (System Clock), SysRst (System Reset)

Output Ports are Sout (Serial Output)

Source Code for Transmitter Module Tx.V

```

/*****
Module Name is Transmitter
*****/
module Tr (SysClk,SysRst,Addr,DataIn,Wr,nrziout,TxEn);

//***** Input/Output Declarations *****/
input      SysClk,SysRst,Wr,TxEn;
input  [1:0]  Addr;
input  [15:0] DataIn;

output      nrziout;

//***** Wire/Reg Declarations *****/

wire      SysClk,SysRst,Wr,TxEn;
wire  [1:0]  Addr;
wire  [15:0] DataIn;
wire      nrziout;

//***** Internal Wire Declarations *****/

```



```

#10 DataIn=16'haaaa;
#10 DataIn=16'haaaa;
#10 DataIn=16'haaaa;
#10 DataIn=16'haaaa;
#10 DataIn=16'haaaa;
#1000 $finish;
end
endmodule

```

6.2 Receiver

6.2.1 Serial to Parallel Converter

Module Name is Serial to Parallel bit converter. File Name is Rec_StoP.V.

This module is the first one to receive data from the transmitter that converts the serial data to parallel which is the form the data has to be for further processing. The data is stored in the buffer temporarily for conversion. It generates address flag and end of conversion signals which are the control signals for other modules.

Input Ports are Sin (Serial Input), Ld (Load Signal), SysClk (System Clock),
 SysRst (System Reset)

Output Ports are POut (16 bit Parallel Output), AddrFlag (Address Flag),
 RxEoc (Receiver side End of Conversion)

Source Code for Serial to Parallel Converter Module Rec_StoP.V

```

/*****
Module Name is Serial To Parallel Converter
*****/

module Rec_StoP (SysClk,SysRst,nrziout,Ld,POut,RxEoc,AddrFlag);
//***** Input/Output Declarations *****/
input        SysClk,SysRst,nrziout,Ld;
output    [15:0] POut;
output        RxEoc,AddrFlag;

//***** Wire/Reg Declarations *****/

wire        SysClk,SysRst,nrziout,Ld;
reg        [15:0] POut;

```

```
reg          RxEoc,AddrFlag,ref,Sin;
```

```
reg  [15:0] Buffer;  
integer  i;  
integer [2:0] d;  
integer [3:0] x;
```

```
/******* Functional Description *****/
```

```
always @ (negedge SysClk)  
begin
```

```
  if (Ld)  
    begin  
      POut =Buffer;  
      RxEoc =RxEoc;  
      i =0;  
      #5 AddrFlag =1'b0;  
      RxEoc =1'b0;  
    end  
  else  
    begin  
      POut =POut;  
      RxEoc =RxEoc;  
      AddrFlag=AddrFlag;  
    end
```

```
end
```

```
always @ (posedge SysClk or negedge SysRst)  
begin
```

```
  if(!SysRst)  
    begin  
      Buffer =16'h0000;  
      AddrFlag =1'b1;  
      i =1'b0;  
      d =3'd0;  
      x =4'd8;  
      ref =1'b0;  
      RxEoc =1'b0;  
    end
```

```
      //added str 1
```

```
  else  
    begin  
      d=d+1;  
      if((nrziout==1'b0)|(nrziout==1'b1))  
        begin
```

```
          if (nrziout==ref)  
            begin  
              Sin =1'b1;  
              ref =nrziout;
```

```
//          d =d+1;  
            end
```

```
          else
```

```

begin
  Sin =1'b0;
  ref =nrziout;
//    d =d+1;
end
end
else
  d =d;

end

//added stop 1
//always @ (posedge SysClk) //or negedge SysRst)
//if(d<3'd6)
if(d<x)
begin
  if ((Sin==1'b0)|(Sin==1'b1))

//    begin

//
begin
  Buffer[i]=Sin;
  i =i+1;
  RxEoc =1'b0;
  d =d;

  if (i==16)
begin
  i =0;
//    d =d; //
  RxEoc=1'b1;
end
else
  RxEoc=1'b0;
end
else
begin
  d =d;
  i =i;
end
end

else
begin
//    Sin =Sin;
//    Buffer[i]=Buffer[i];
  i =i;
  d =0;
  x =4'd7;
end
/* end
else
begin

```



```

        i =0;
        d =d;
    end

*/
end
endmodule

```

6.2.2 Demultiplexer

Module Name is Demultiplexer. File Name is Rec_Dmux.V.

This module does opposite in function to what multiplexer does in transmitter it sends parallel data to address register, FIFO and to CRC depending on the control signals. The purpose is to send the received information to respective places where they are processed later.

Input Ports are POut (16 bit Parallel Data), RxAddrEn (Address Enable Signal),
 RxFifoEn (FIFO Enable Signal), RxCrcEn (CRC Enable Signal),
 SysRst (System Reset)

Output Ports are RxAddrReg (16 bit Address Register),
 RxFifoIn (16 bit FIFO Input), RxCrcReg (4 bit CRC Register)

Source Code for Demultiplexer Module Rec_Dmux.V

```

/*****
    Module Name is Receiver Side Demultiplexer
*****/
module        Rec_Dmux
    (RxAddrReg,RxFifoIn,RxCrcReg,POut,RxAddrEn,RxFifoEn,RxCrcEn,SysRst);

//***** Input/Output Declarations *****/
input        [15:0]        POut;
input                      RxAddrEn,RxFifoEn,RxCrcEn,SysRst;

output       [15:0]        RxAddrReg,RxFifoIn;
output   [3:0]    RxCrcReg;

//***** Wire/Reg Declarations *****/

```

```

wire          [15:0]          POut;
wire          RxAddrEn,RxFifoEn,RxCrcEn,SysRst;

reg           [15:0]          RxAddrReg,RxFifoIn;
reg           [3:0]          RxCrcReg;

//***** Functional Description *****
always       @          (POut or RxAddrEn or RxFifoEn or RxCrcEn or SysRst)
    begin
        if(!SysRst)
            begin
                RxAddrReg = 16'hzzzz;
                RxFifoIn = 16'hzzzz;
                RxCrcReg = 4'hz;
            end
        else if(RxAddrEn)
            begin
                RxAddrReg = POut;
            end
        else if(RxFifoEn)
            begin
                RxFifoIn = POut;
            end
        else if(RxCrcEn)
            begin
                RxCrcReg = POut[3:0];
            end
        else
            begin
                RxAddrReg = RxAddrReg;
                RxFifoIn = RxFifoIn;
                RxCrcReg = RxCrcReg;
            end
    end
endmodule

```

6.2.3 Control Logic

Module Name is Control Logic Block. File Name is Rec_Control.V.

The module receiver control generates control signals, which are address enable signal, FIFO enable signal and CRC Enable signal. These are important for control and to maintain timing for other modules.

Input Ports are AddrFlag (Address Flag), CrcFlag (CRC Flag),
 RxEoc (End of Conversion)

Output Ports are RxAddrEn (Address Enable Signal),
RxFifoEn (FIFO Enable Signal), Rx_crcEn (CRC Enable Signal)

Source Code for Control Logic Module Rec_Control.V

```

/*****
Module Name is CRC Calculator
*****/
module Rec_crcGen (RxCRC,RxFifoIn,SysRst,RxEoc);
/*****INPUT/OUTPUT DECLARATION*****/
input [15:0] RxFifoIn; //Message coming from FIFO
// Generator polynomial value
input SysRst,RxEoc;

output [3:0] RxCRC;
/*****WIRE/REG DECLARATION*****/
wire [15:0] RxFifoIn;
wire SysRst,RxEoc;

reg [3:0] RxCRC;

/*****Internal Wire/Reg Declaratio *****/

reg [4:0] GenPoly; //Generator Polynomial value (5 bit)

/*****FUNCTIONAL DESCRIPTION*****/

always @ (negedge RxEoc or negedge SysRst)
begin
    if (!SysRst)
        begin
            Temp =4'd0;
            GenPoly=5'b10011;
        end
    else
        begin
            RxCRC =CRC_Cal(RxFifoIn,Temp,GenPoly);
            Temp =RxCRC;
        end
end

function [3:0] CRC_Cal;
parameter Zero=5'b00000;

input [15:0] FifoOut;
input [3:0] Temp;
input [4:0] GenPoly;

```

```

//***** Internal Reg Declaratio *****
reg          [20:1]          msgtemp;
reg          [4:0]          Temp1;
reg          [4:0]          Rem;
integer      i;

begin

                                msgtemp=({FifoOut,Temp});

                                //display(" @@@@ @ msgtemp=%b",msgtemp);
                                Temp1=msgtemp[20:16];
                                //display("*****Temp=%b",Temp1);
                                for(i=15;i>0;i=i-1)
                                begin
                                if      (GenPoly<=Temp1)
                                    begin
                                        Rem=GenPoly ^ Temp1;
                                        //display("*****Rem(Result)=%b",Rem);
                                        Rem=Rem << 1;
                                        //display("*****Rem(after removing
MSB)=%b",Rem);

                                        Rem[0] = msgtemp[i];
                                        //display("*****Rem(after appending M(x)
bit)=%b",Rem);

                                        end
                                    else
                                        begin
                                            Rem=Temp1^Zero;
                                            //display("*****Rem(Result)=%b",Rem);
                                            Rem=Rem<< 1;
                                            //display("*****Rem(after removing
MSB)=%b",Rem);

                                            Rem[0]=msgtemp[i];
                                            //display("*****Rem(after appending M(x)
bit)=%b",Rem);

                                        end
                                        //display("(Remainder=%b",Rem);
                                        Temp1=Rem;
                                    end
                                CRC_Cal=Rem[3:0];

                                //display("^^^^^^^^^^^^^^^^^CRC_Cal=%b",CRC_Cal);

end
endfunction
endmodule

```

6.2.4 Address Decoder

Module Name is Address Decoder. File Name is Rec_AddrDec.V

This module extracts the 4-bit address that corresponds to 16 devices connected and the size of the data sent which is 12-bit.

Input Ports are RxAddrReg (16 bit Address Register),

RxAddrEn (Address Enable Signal), SysRst (System Reset)

Output Ports are DevAddr (4 bit Device Address), DataSize (12 bit Data Size)

Source Code for Address Decoder Module Rec_AddrDec.V

```

/*****
Module Name is Receiver Address Decoder
*****/

module Rec_AddrReg (DevAddr,DataSize,RxAddrEn,RxAddrReg,SysRst);
//***** Input/Output Declarations *****/
input      [15:0]  RxAddrReg;
input      RxAddrEn,SysRst;

output     [3:0]   DevAddr;
output     [11:0]  DataSize;
//***** Wire/Reg Declarations *****/
wire      [15:0]  RxAddrReg;
wire      RxAddrEn,SysRst;

reg       [3:0]   DevAddr;
reg       [11:0]  DataSize;

//***** Functional Description *****/

always @ (RxAddrEn or RxAddrReg or SysRst)
begin
    if(!SysRst)
    begin
        DevAddr =4'd0;
        DataSize=12'd0;
    end
    else if(RxAddrEn)
    begin
        $display("*****DevAddr=%b DataSize=%h",DevAddr,DataSize);
        DevAddr = RxAddrReg[3:0];
        DataSize= RxAddrReg[15:4];
    end
    else

```



```

always @ (negedge SysRst or negedge RxEoc )
begin
    if(!SysRst)
        begin
            CrcFlag = 1'b0;
            Count = 12'h001;
            Flag = 1'b1;
        end
    else if (Flag==1'b1)
        begin
            Count = DataSize;
            Flag = 1'b0;
            //$strobe("$$$$$$Count=%h",Count);
            CrcFlag = 1'b0;
        end
    else begin
        Count =Count-1;
        if(Count==0)
            begin
                CrcFlag =1'b1;
                Count =Count;
            end
        else
            begin
                CrcFlag =CrcFlag;
                Count =Count;
            end
        end
    end
end
endmodule

```

6.2.6 CRC Generator

Module Name is Cyclic Redundancy Check (CRC) Generator. File Name is Rec_Crc.V

This module calculates the CRC bits for error handling. The CRC Generator first calculates the CRC bits for the first 16 bits of FIFO Input and the CRC bits for the next 16 bit FIFO Input is calculated with respect to the first generated CRC bits. In this way the process continues and finally 4 CRC bits are generated for whole data and sent to the CRC Comparator to check whether the generated CRC bits and received CRC bits are

same. If they are same then the data is received without any error otherwise there is an error in received data.

Input Ports are RxFifoIn (16 bit FIFO Input), RxEoc (End of Conversion Signal),
 SysRst (System Reset)

Output Ports are RxCRC (4 bit CRC)

Source Code for CRC Generator Module Rec_Crc.V

```

/*****
Module Name is CRC Calculator
*****/
module Rec_CrcGen (RxCRC,RxFifoIn,SysRst,RxEoc);
//*****INPUT/OUTPUT DECLARATION*****
input [15:0] RxFifoIn; //Message coming from FIFO
// Generator polynomial value
input SysRst,RxEoc;

output [3:0] RxCRC;
//*****WIRE/REG DECLARATION*****
wire [15:0] RxFifoIn;
wire SysRst,RxEoc;

reg [3:0] RxCRC;

//*****Internal Wire/Reg Declaratios *****

reg [4:0] GenPoly; //Generator Polynomial value (5 bit)
reg [3:0] Temp;

//*****FUNCTIONAL DESCRIPTION*****

always @ (negedge RxEoc or negedge SysRst)
begin
    if (!SysRst)
        begin
            Temp =4'd0;
            GenPoly=5'b10011;
        end
    else
        begin
            RxCRC =CRC_Cal(RxFifoIn,Temp,GenPoly);
            Temp =RxCRC;
        end
end
end

```



```

function [3:0]          CRC_Cal;
parameter              Zero=5'b00000;

input      [15:0]      FifoOut;
input      [3:0]       Temp;
input      [4:0]       GenPoly;
//***** Internal Reg Declaratios *****
reg        [20:1]      msgtemp;
reg        [4:0]       Temp1;
reg        [4:0]       Rem;
integer    i;

begin

                                msgtemp=({FifoOut,Temp});

                                //display("@ @ @ @ @ @ msgtemp=%b",msgtemp);
                                Temp1=msgtemp[20:16];
                                //display("*****Temp=%b",Temp1);
                                for(i=15;i>0;i=i-1)
                                begin
                                    if      (GenPoly<=Temp1)
                                        begin
                                            Rem=GenPoly ^ Temp1;
                                //display("*****Rem(Result)=%b",Rem);
                                            Rem=Rem << 1;
                                //display("*****Rem(after removing
MSB)=%b",Rem);

                                            Rem[0] = msgtemp[i];
                                //display("*****Rem(after appending M(x)
bit)=%b",Rem);

                                        end
                                    else
                                        begin
                                            Rem=Temp1^Zero;
                                //display("*****Rem(Result)=%b",Rem);
                                            Rem=Rem<< 1;
                                //display("*****Rem(after removing
MSB)=%b",Rem);

                                            Rem[0]=msgtemp[i];
                                //display("*****Rem(after appending M(x)
bit)=%b",Rem);

                                        end
                                //display("(Remainder=%b",Rem);
                                        Temp1=Rem;
                                    end
                                CRC_Cal=Rem[3:0];

                                //display("^^^^^^^^^^^^^^^^CRC_Cal=%b",CRC_Cal);

```

```
end
endfunction
endmodule
```

6.2.7 CRC Comparator

Module Name is CRC Comparator. File Name is Rec_CrcComp.V.

This module compares the CRC bits received from the transmitter with the CRC bits generated by the receiver. If the two CRC's are equal then the data is received without any error and if the two CRC's are not equal then an Interrupt is generated to resend the data again.

Input Ports are CRC (4 bit CRC from Transmitter),
 RxCRC (4 bit CRC generated in the receiver),
 RxCrcEn (CRC Enable Signal), SysRst (System Reset)

Output Ports are Intr (Interrupt)

Source code for Rec_CrcComp.V

```
/******
Module Name is CRC Comparator
*****/
module Rec_CrcComp (Intr,CRC,RxCRC,RxCrcEn,SysRst);
//***** Input/Output Declarations *****
input           [3:0]     CRC,RxCRC;
input                     RxCrcEn,SysRst;

output                    Intr;

//***** Wire/Reg Declarations *****

wire        [3:0]     CRC,RxCRC;
wire                    RxCrcEn,SysRst;
reg                     Intr;
```

```
/******* Functional Description *****
```

```
always          @          (negedge RxCrcEn or negedge SysRst)
begin
  if (!SysRst)
    Intr = 1'b0;
  else if(RxCrcEn==1'b0)
    begin
      if(RxCRC!=CRC)
        Intr=1'b1;
      else
        Intr = 1'b0;
      end
    else
      Intr = Intr;
end
end
endmodule
```

6.2.8 Rec_FIFO (First In First Out)

Module Name is First In First Out. File Name is Rec_FIFO.V

In this module if FIFO has nothing in it FIFO Empty Signal is generated allowing the data to be written into the FIFO, if the FIFO is half full then FIFO Half Signal is generated allowing the data to be still written in FIFO and if the FIFO is full then FIFO Full Signal is generated not allowing any data to be written in the FIFO until there is free size to be written. An internal Flag is created in such a way that whenever it is high the data is copied into the FIFO and whenever the Read Signal is high data which is written first is sent out to the device. This is a 16x16 bit FIFO

Input Ports are RxFifoIn (16 bit FIFO Input), Rd (Read Signal),
 RxEoc (End of Conversion Signal),
 DevClk (Device Clock), SysRst (System Reset)

Output Ports are RxFifoOut (16 bit FIFO Output), RxFull (FIFO Full Signal),
 RxEmpty (FIFO Empty Signal), RxHalf (FIFO Half Signal)

The source code for Rec_FIFO.V

```

/*****
Module Name is FirstInFirstOut (FIFO)
*****/
module Rec_FIFO(RxFifoIn,RxFifoOut,RxEoc,DevClk,Rd,SysRst,RxFull, RxEmpty,RxHalf);
/*****INPUT/OUTPUT DECLARATION*****/

input      [15:0]      RxFifoIn;
input      DevClk,SysRst,Rd,RxEoc;

output     [15:0]      RxFifoOut;
output     RxFull,RxEmpty,RxHalf;

/*****WIRE/REG DECLARATION*****/

wire       [15:0]      RxFifoIn;
wire       DevClk,SysRst,Rd,RxEoc;

reg        [15:0]      RxFifoOut;
reg        RxFull,RxEmpty,RxHalf;

reg        [15:0] FIFO [0:15];
reg        [3:0]      Rdptr,Wrptr;
integer    [4:0]      Count;
reg        Flag;
/*****FUNCTIONAL
DESCRIPTION*****/
always     @          (negedge RxEoc or negedge SysRst)
begin
    if (!SysRst)
begin
    Wrptr=4'b0000;
    Flag =0;
end
else if (Flag)
begin
    FIFO[Wrptr]=RxFifoIn;
    //$display("^^^^^^^^^^^^^^^^FIFO[%d]=%h",Wrptr,FIFO[Wrptr],$time);
    Wrptr  =Wrptr+1;
    Count  =Count+1;
end
else
    Flag=1;
end

always     @          (posedge DevClk or negedge SysRst)
begin
    if      (!SysRst)
begin
    Rdptr=4'd0;
    Count=0;
end
end

```

```

else if (Rd)
begin
    RxFifoOut=FIFO[Rdptr];
    Rdptr  =Rdptr+1;
    Count  =Count-1;
end
end

always @          (Count or SysRst)
begin
    if (!SysRst)
begin
    RxFull =1'b0;
    RxEmpty=1'b0;
    RxHalf =1'b0;
end
    else if (Count==0)
begin
    RxFull =1'b0;
    RxEmpty=1'b1;
    RxHalf =1'b0;
end
    else if (Count>=15)
begin
    RxFull =1'b1;
    RxEmpty=1'b0;
    RxHalf =1'b0;
end
    else if (Count==8)
begin
    RxFull =1'b0;
    RxEmpty=1'b0;
    RxHalf =1'b1;
end
    else
begin
    RxFull =1'b0;
    RxEmpty=1'b0;
    RxHalf =1'b0;
end
end
endmodule

```

6.2.9 Receiver Module

Module Name is Receiver. File Name is Receiver.V.

This module is basically the integration of all the 8 modules explained above which are responsible for receiving the data from the peripheral device to the host without any error


```

//StoP s (SysClk, SysRst, Sin, RxEoc, POut, RxEoc, AddrFlag);
StoP s (SysClk, SysRst, nrziout, RxEoc, POut, RxEoc, AddrFlag);
RxControl c (RxAddrEn, RxFifoEn, RxCrcEn, AddrFlag, CrcFlag, RxEoc);
RxDmux d (RxAddrReg, RxFifoIn, CRC, POut, RxAddrEn, RxFifoEn, RxCrcEn, SysRst);
//*****0000000000000000000000000000000000000000*****

RxAddrReg a (DevAddr, DataSize, RxAddrEn, RxAddrReg, SysRst);
RxFIFO f (RxFifoIn, RxFifoOut, RxEoc, DevClk, Rd, SysRst, RxFull, RxEmpty, RxHalf);

RxCrcGen g (RxCRC, RxFifoIn, SysRst, RxEoc);
Timer t (CrcFlag, RxEoc, DataSize, SysRst, RxAddrEn);
CrcComp m (Intr, CRC, RxCRC, RxCrcEn, SysRst);
//***** Functional Description *****/
endmodule

```

6.3 Serial Interface Engine

Module Name is Serial Interface Engine. File Name is Comm.V.

This module is an integration of both receiver and transmitter. This module is tested by connecting transmitter serial output to receiver serial input and all the results are verified.

Input Ports are DataIn (16 bit Input Data), Addr (2 bit Address),
 TxEn (Transmitter Enable), Wr (Write Signal),
 Rd (Read Signal), SysClk (System Clock),
 SysRst (System Reset)

Output Ports are RxFifoOut (16 bit FIFO Output), DevAddr (4 bit Device Address)
 RxFull (FIFO Full Signal), RxHalf (FIFO Half Signal),
 RxEmpty (FIFO Empty Signal), Intr (Interrupt)

Source Code for Serial Interface Engine Module Comm.V

```

/*****
Module Name is Serial Interface Engine
Module Description
*****/

```

```

module Sie
(SysClk,SysRst,Addr,DataIn,Wr,TxEn,Rd,RxFifoOut,DevAddr,RxFull,RxHalf,RxEmpty,Intr);
//***** Input/Output Declarations *****
input SysClk,SysRst,Wr,TxEn,Rd;
input [1:0] Addr;
input [15:0] DataIn;

output [15:0] RxFifoOut;
output [3:0] DevAddr;
output RxFull,RxHalf,RxEmpty,Intr;
//***** Wire/Reg Declarations *****

wire SysClk,SysRst,Wr,TxEn,Rd;
wire [1:0] Addr;
wire [15:0] DataIn;

wire [15:0] RxFifoOut;
wire [3:0] DevAddr;
wire RxFull,RxHalf,RxEmpty,Intr;

wire nrziout;

//***** Functional Description *****

Tx t (SysClk,SysRst,Addr,DataIn,Wr,nrziout,TxEn);
Rx r (SysClk,DevClk,Rd,RxFifoOut,SysRst,nrziout,DevAddr,RxFull,RxHalf,RxEmpty,Intr);

endmodule
//***** Simulation Module *****
module SieTest;
//***** Wire/Reg Declarations *****
reg SysRst,Wr,TxEn,Ld; //addedf Ld
reg [1:0] Addr;
reg [15:0] DataIn;
wire SysClk;

wire [15:0] RxFifoOut;
wire [3:0] DevAddr;
wire RxFull,RxHalf,RxEmpty,Intr;

Sie s (SysClk,SysRst,Addr,DataIn,Wr,TxEn,Rd,RxFifoOut,DevAddr,RxFull,RxHalf,RxEmpty,Intr);
Clock c (SysClk);

initial
begin
SysRst=1; DataIn=16'h00aa; Addr=2'b00;TxEn=1'b0;Wr=1'b1; // added Ld

$dumpvars(1,s.t.SysClk,s.t.SysRst,s.t.DataIn,s.t.nrziout,s.t.TxEn,s.r.f.RxFifoIn,s.r.RxFull,s.r.RxHalf,s.r.Rx
Empty,s.r.Intr,s.r.DevAddr);
$dumpfile("sie.dmp");

```


Chapter 7 Results and Conclusion

7.1 Simulated Waves of the Serial Interface Engine (SIE)



Figure7.1 Simulated Waves of the final module

Contd.

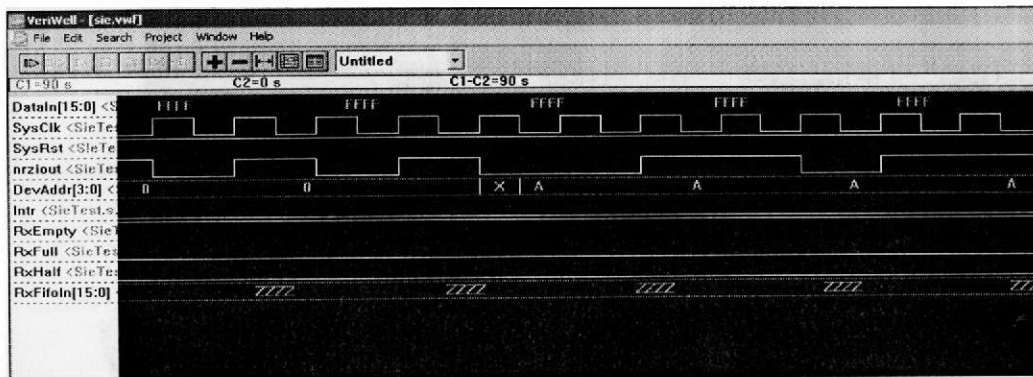


Figure 7.2 Simulated Wave files of the final module

Contd.

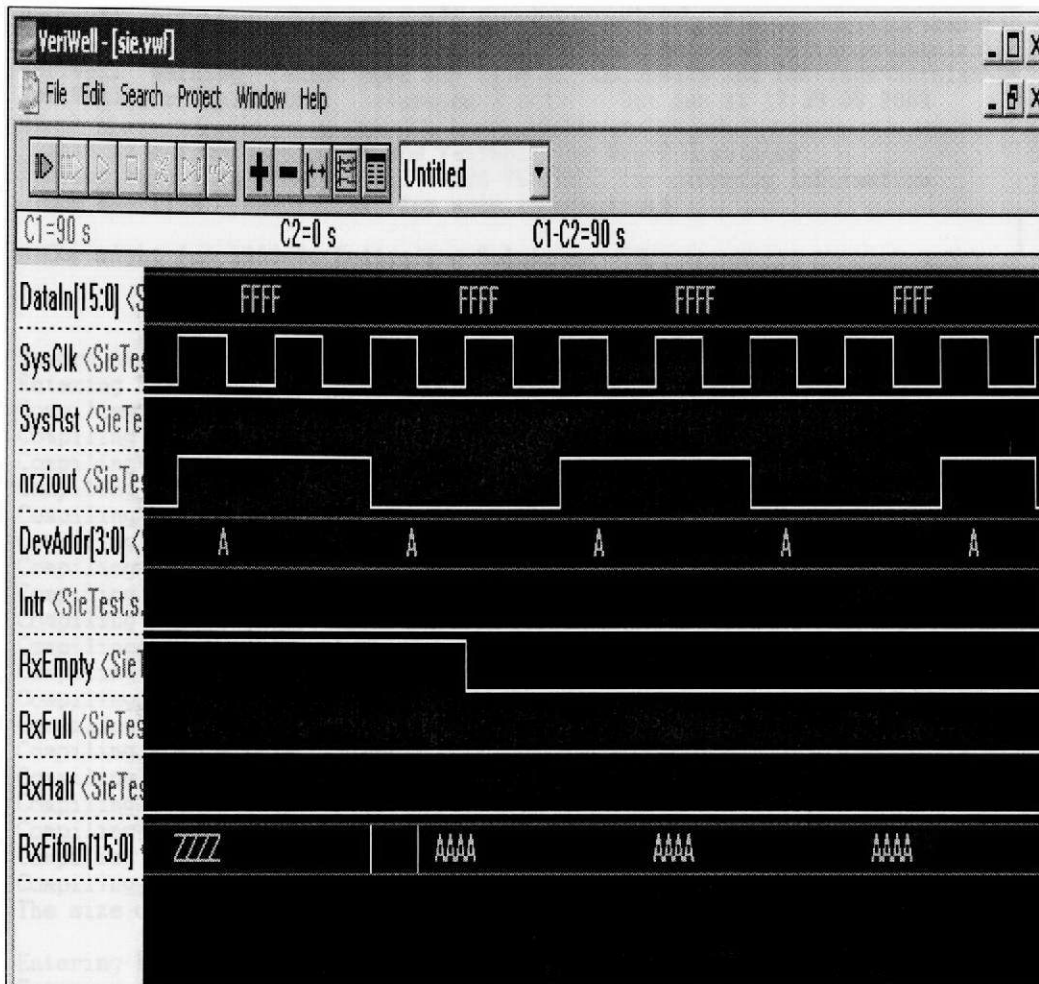


Figure 7.3 Simulated Waves of the final module

```

VeriWell - [VeriWell Console]
File Edit Search Project Window Help

VeriWell -k C:\Documents and Settings\hstnlab\Desktop\WINDOWS\DESKTOP\baser\p
VeriWell: warning: Cannot open log file 'C:\Documents and Settings\hstnlab\De
VeriWell: warning: Cannot open key file 'C:\Documents and Settings\hstnlab\De
VeriWell for Win32 HDL <Version 2.1.1> Sat Jan 11 17:39:05 2003

This is a free version of the VeriWell for Win32 Simulator
Distribute this freely; call 1-800-VERIWELL for ordering information
See the file "readme.lst" for more information!

Copyright (c) 1993-96 Wellspring Solutions, Inc.
All rights reserved

Memory Available: 0
Entering Phase I...
Compiling source file : Buffer.V
Compiling source file : Clock.V
Compiling source file : Control.V
Compiling source file : Rx.V
Compiling source file : Crc.V
Compiling source file : Deco.V
Compiling source file : Mux.V
Compiling source file : RxAddrDec.V
Compiling source file : RxControl.V
Compiling source file : Rx_crc.V
Compiling source file : Rx_crcComp.V
Compiling source file : RxDmux.V
Compiling source file : Rx_FIFO.V
Compiling source file : Timer.V
Compiling source file : Tx_FIFO.V
Compiling source file : PtoS2.V
Compiling source file : Sei_test.V
Compiling source file : Tx1.V
Compiling source file : stop_1.V
The size of this model is [54%, 86%] of the capacity of the free version

Entering Phase II...
Entering Phase III...
2 warnings in compilation
No errors in compilation
Top-level modules:

```

Figure 7.4 VeriWell Console for the SIE

VeriWell - [VeriWell Console]						
File Edit Search Project Window Help						
SieTest						
DataIn=00aa	nrziout=x	Eoc=x	RxEoc=x	PDataIn=xxxx	RxFifoIn=xxxx	Sin=x Addr=x 0
DataIn=00aa	nrziout=z	Eoc=0	RxEoc=0	PDataIn=zzzz	RxFifoIn=zzzz	Sin=x Addr=0 2
DataIn=0013	nrziout=z	Eoc=0	RxEoc=0	PDataIn=zzzz	RxFifoIn=zzzz	Sin=x Addr=0 14
DataIn=0001	nrziout=z	Eoc=0	RxEoc=0	PDataIn=zzzz	RxFifoIn=zzzz	Sin=x Addr=0 24
DataIn=aaaa	nrziout=z	Eoc=0	RxEoc=0	PDataIn=zzzz	RxFifoIn=zzzz	Sin=x Addr=0 26
DataIn=5555	nrziout=z	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=x Addr=0 36
DataIn=0000	nrziout=z	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=x Addr=0 46
DataIn=0000	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=x Addr=0 50
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=x Addr=0 56
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 60
DataIn=0000	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 66
DataIn=0000	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=1 Addr=0 70
DataIn=ffff	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=1 Addr=0 76
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 80
DataIn=0000	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 86
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 96
DataIn=ffff	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=1 Addr=0 100
DataIn=0000	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=1 Addr=0 106
DataIn=0000	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 110
DataIn=ffff	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 116
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=1 Addr=0 120
DataIn=0000	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=1 Addr=0 126
DataIn=0000	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 130
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 136
DataIn=ffff	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=1 Addr=0 140
DataIn=0000	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=1 Addr=0 146
DataIn=0000	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 150
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 156
DataIn=ffff	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 160
DataIn=0000	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 166
DataIn=0000	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 170
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 176
DataIn=ffff	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 180
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 190
DataIn=ffff	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 200
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 210
DataIn=ffff	nrziout=0	Eoc=0	RxEoc=0	PDataIn=00aa	RxFifoIn=zzzz	Sin=0 Addr=0 220
DataIn=ffff	nrziout=1	Eoc=1	RxEoc=0	PDataIn=aaaa	RxFifoIn=zzzz	Sin=0 Addr=0 230
DataIn=ffff	nrziout=1	Eoc=0	RxEoc=0	PDataIn=aaaa	RxFifoIn=zzzz	Sin=0 Addr=0 235

Figure 7.5 VeriWell Console for the SIE

7.2 Conclusion

In this thesis, hardware is designed using the VLSI techniques, which is a logical design of communication block to show the data communication of binary signals. The design considers the various aspects for providing reliable and effective digital data transmission in the hardware. It employs control signals in a very effective manner to integrate the functioning of various modules. Though the model designed is a part of an USB it can also be used as an interfacing block in any device-connecting host with peripheral devices. Unlike, USART and UART, this device can interface host with more than one device. The design shows that it can connect 16 devices. The device is a full duplex where both the host and the peripheral devices can communicate signals at the same time.

In addition some effective Data Communication techniques are employed in the design. They are

- Generation of bit patterns in an optimal manner
- Use of CRC, Bitstuffing and NRZ techniques
- Synchronization of Clock and Data
- Effective use of control signals like Reset and Resume signals.

7.3 Future work

- Different encoding schemes can be used like that of differential encoding.
- The use of different Simulation environment like using Verilog-XL by Cadence Design Systems, inc. Active HDL by Aldec for better Designing.
- Usage of Improved algorithms to realize the complexity of Functions and to reduce the number of modules by integrating various Functions in the Algorithms.
- Implementations of communication protocols like Higher Data Level Control.
- Performance of the Hardware is best utilized in association with Suitable software like Device Drivers.

References

- [1] William Stallings, "Data and Computer Communications", Prentice Hall, 6th Edition (September 2000).
- [2] Samir Palnitkar, "Verilog HDL A Guide to Digital Design and Synthesis". Prentice Hall, 2nd Edition (February 1996).
- [3] Tom Sheldon "Encyclopedia of Networking & Telecommunications", Tata McGraw-Hill, 2nd Edition (May 2001)
- [4] Universal Serial Bus Specifications, Revision 2.0 April 27, 2000.
<http://www.usb.org/>
- [5] Don Anderson, Dave Dzatko, Inc Mindshare. "Universal Serial Bus System Architecture", Addison-Wesley, 2nd Edition (April 2001).
- [6] Wayne Wolf. "Modern VLSI Design", Prentice Hall, 3rd Edition (January 2002).
- [7] Alexander, M. J., and Robins, G. "New Performance-Driven FPGA Routing Algorithms, "IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 12, December 1996, pp. 1505-1517.
- [8] V. Betz and J. Rose, "Circuit Design, Transistor Sizing and Wire Layout of FPGA Interconnect," *IEEE Custom Integrated Circuits Conference*, San Diego, CA, May 1999, pp. 171 - 174.
- [9] History of VLSI
<http://www.doe.carleton.ca/~rmason/Teaching/489-a.pdf>
- [10] Design of VLSI Systems
<http://vlsi.wpi.edu/webcourse/ch01/ch01.html>
- [11] Zainalabedin Navabi. "VHDL, Analysis and Modeling of Digital Systems". McGraw-Hill, 2nd Edition (December 1997).
- [12] Alexander, M. J., Cohoon, J. P., Ganley, J. L., Robins, G., Placement and Routing for Performance-Oriented FPGA Layout, *VLSI Design: an International Journal of Custom-Chip Design, Simulation, and Testing*, Vol. 7, No. 1, 1998.
- [13] Steven M Rubin, "Computer Aids for VLSI Design", Addison-Wesley, 2nd Edition (June 1987).