**ABSTRACT**

This paper studies design and implementation of the Turbo encoder to be an embedded module in the in-vehicle system (IVS) chip. Field programmable gate array (FPGA) is employed to develop the Turbo encoder module. Both serial and parallel computations for the encoding technique are studied. The two design methods are presented and analyzed. Developing the parallel computation method using carry increment adder, it is shown that both chip size and processing time are reduced. The logic utilization is enhanced by reduced area. The Turbo encoder module is designed, simulated, and synthesized using Xilinx tools. Xilinx vertex low power is employed. The Turbo encoder module is designed to be a part of the IVS chip on a single programmable device.

Chapter 1

**INTRODUCTION**

In a basic communication system the message is sent from the transmitter and received at the receiver by passing through the channel. The channel is corrupted by external noise sources so the original message sent by transmitter is not received and a transformed message is received. This transformed message may contain bit errors caused by a noisy channel. Redundant bits are introduced into the original message, which enable the receiver to detect and correct errors caused by the channel.

***Why turbo codes are used?***

Error correction codes can detect errors caused by noise and interference, and recreate the error free original data in digital communication systems. Error correcting codes are also to referred as forward error correcting (FEC) codes or channel codes. It detects the errors caused during transmission of message from transmitter to receiver by external noise. In error correcting codes, the introduction of redundancy bits to a message allows us to detect and correct the errors that occurred during transmission.

In block codes, a message is broken into fixed length blocks and message bits are encoded and decoded separately for each block. Convolutional codes can produce a continuous stream of bits with added redundant bits interleaved among data bits. Cyclic and convolutional codes are considered similar as they are implemented with the help of linear shift registers. These two differ as the cyclic and block codes are of fixed block size, whereas the convolutional codes can accept continuous input streams. A cyclic code is a linear system with one input and one output system whereas a convolutional code is multi input and multi output system [9]. Encoders and decoders of block and convolutional codes are implemented with reasonable complexity as they are highly structured. The practical implementation result of these codes is inferior to random coding bounds predicted by Shannon due to the structure of block and convolutional codes. A turbo code is the parallel concatenation of two or more component codes which are subclass of convolution codes. Turbo codes are also structured but they permit practical encoding and decoding algorithms and their structure on the channel is randomized by a pseudo-random interleave. Hence the bit and code error rate performance of turbo codes is closer to the Shannon bound than the performance of block and convolutional codes. The key to turbo code performance is the use of an iterative decoding algorithm, which increases computational complexity and latency, but improves bit error performance. Turbo codes increase performance with increased constraint length relative to convolutional codes. Turbo codes have low power and interference limited applications. BER performance of turbo codes depends on the number of bit errors in the received sequence.

Data transmission efficiency is enriched in digital communication systems by the use of turbo codes. Turbo codes are a combination of recursive systematic encoders, an interleaver, puncturing and the iterative decoder components. A common turbo encoder has two half rate recursive convolutional encoders. The RSC (Recursive Systematic Convolutional) codes is obtained from NSC (Non-systematic Convolutional) by adding a feedback loop and making one of the output bits equal to the input. The interleaver helps in randomizing the error patterns to decode correctly. It increases the minimum distance of the turbo code. Puncturing is a technique to increase the code rate. It is a process in which specific bits in the output stream are removed in a fixed pattern given by the puncturing matrix. It is used to increase the transmission efficiency. The turbo decoder consists of two decoders in which one decoder‟s output is used by the other decoder to produce the transmitted message. The output of the second decoder is fed back to the first decoder and this process is repeated until the decoder estimates converge. Iterative decoding helps greatly in increasing performance of BER in turbo code.

The Maximum a posteriori (MAP) algorithm is not usually implemented because of its computational complexity. Log-MAP and Max-log-MAP decoding algorithms are the logarithmic version of the MAP algorithm. Non-linear functions are required for calculating the probabilities of both addition and multiplication to compute the variables of this algorithm. In this thesis we use the MAX-log-MAP algorithm.

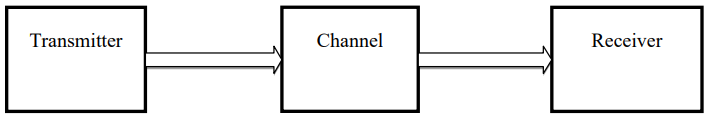
In this thesis, we investigate the use of an external optimization loop, which repeatedly invokes the MAX-log-MAP decoding algorithm on codewords which fail to converge. Locally generated noise increments (i.e., “mutation”) are added to the original received codeword in order to accomplish genetic optimization. Optimization algorithms explored in this thesis include Hill Climbing and Genetic algorithms. Hill Climbing is a simple search optimization algorithm. Hill Climbing executes a loop in which the currently known best solution produces an offspring and if this is better than the parent, then it is replaced. The application areas of hill climbing include networking and communications, robotics, data mining, analysis etc. Genetic algorithms find a global optimum solution for an optimization problem. The applications of genetic algorithms are wide in fields of biology, physics, economics, engineering, etc.

The thesis is structured in a way that theories are explained by bringing together various authors work and views and by understanding the concepts that are relevant for answering the research question. In this thesis, we use a 1/3 rate duo-binary turbo code, with a block size of 1024 data bits, and a standard MAX-log-MAP decoding algorithm. Received codewords which fail to converge within 20 iterations are processed further bythe optimization algorithm. Different feedback and scoring measures are evaluated to empirically obtain the best performance.

**Digital Communication Systems**

Communication systems can be divided into two types, analog systems and digital systems, and this classification depends on the channel through which the signal is transmitted. Analog signals take an infinite number of possible amplitudes in a given range, whereas discrete amplitude signals take a finite number of amplitudes in a fixed range [3]. The important elements of a digital communication system are the transmitter, the channel and the receiver. A digital communication system design is initiated with channel description, available resources such as received power, bandwidth, and signal- to-noise ratio (SNR). A digital communication system is more advantageous than an analog communication system, as it is more resistant to errors caused due to transmission, than information symbolized in an analog medium. A digital communication system transfers information from the digital source to the sink.

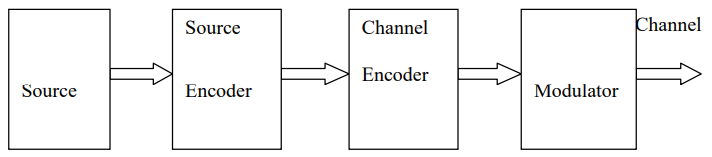
The basic block diagram of the digital communication systems is shown in Figure 1, where a source consists of the messages to be transmitted. The output of this source block is given as input to the transmitter which operates on the messages and prepares them for propagation over the channel. The channel here is considered as the medium for transmitting the messages to the receiver. When the message is passed through the channel many external noises, sources affect the message during transmission of the message. The channel output taken by the receiver is used to recover the source bits at output of the receiver. If the output from the receiver is an electrical signal, then the receiver which contains the transducer takes the signal and converts into its original form.



*Figure 1:* Basic block diagram of Digital Communication System.

***Transmitter***

As shown in Figure 2 the output from the source is given to the source encoder which encodes the digital signal efficiently. The channel encoder present next to the source encoder strives to decrease the probability of errors in the decoding by adding redundancy to the source encoded message. The modulator serves in changing the signal into an appropriate form for transmission over the channel.



*Figure 2* Block Diagram of Transmitter.

***Channel***

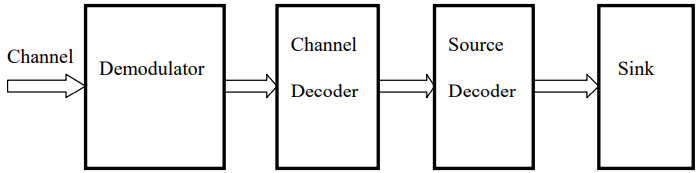
The channel is defined as a medium through which the message is transmitted. The channel is the connector between transmitter and receiver through which the message is passed. Examples of a channel are co-axial cable, free space, linked wire etc. The external noise distorts the message when sent through channel.

***Noise***

In digital communications noise can result in bit errors at the receiver. The types of noises can be classified as natural and man-made noises. The significant types of noises are thermal noise, shot noise and white noise, as well as non-white noise from man-made sources (sometimes referred to as interference).

***Receiver***

Figure 3 represents the block diagram of the receiver. The channel‟s output is given to the demodulator, which reverses the process of the modulator. The channel decoder receives the output from the demodulator and makes an effort to reproduce the input at the channel decoder. The source decoder receives the signal and attempts to acquire an “original signal” as appropriately as possible. Hence the original source information is received.



*Figure: 3* Block Diagram of Receiver

In 1948, Shannon proved that every noisy channel has a maximum rate at which information may be transferred through it and that it is possible to design error-correcting codes that approach this capacity, or Shannon limit, provided that the codes may be unbounded in length. For the last six decades, coding theorists have been looking for practical codes capable of closely approaching the Shannon limit. Turbo codes, a new technique of error correction coding developed in the 1990s.In 1993, a concatenated forward error correction (FEC) scheme, turbo coding was introduced by Berrou. Turbo coding is a very powerful error correction technique that has made a tremendous impact on channel coding in the last few years. It outperforms all previously known coding schemes by achieving near Shannon limit error correction using simple component codes and large interleavers. Turbo codes have become a 3G standard The iterative decoding mechanism, recursive systematic encoders and use of interleavers are the characteristic features of turbo codes.

The use of turbo codes enhances the data transmission efficiency in digital communications systems. Turbo code development proceeded from theoretical analyses of polynomial selection, weight distributions imposed by interleaver designs, decoder error floors, and iterative decoding thresholds. A family of turbo codes was standardized and implemented and is currently inuse by several spacecraft. JPL’s LDPC codes are built from protographs and circulants Turbo codes enable reliable communication over power constrained communication channels at close to Shannon’s limit. However, a significant number of iterations are required to produce this result leading to higher latency. The task of channel coding is to encode the information sent over a communication channel in such a way that in the presence of channel noise, errors can be detected and/or corrected. Thus efficient implementation of turbo codes in order to meet real-time constraints is anactive area of research.

Designing a channel code is always a tradeoff between energy efficiency and bandwidth efficiency*.* Codes with lower rate (i.e. bigger redundancy) can usually correct more errors. If more errors can be corrected, the communication system can operate with a lower transmit power, transmit over longer distances, tolerate more interference, use smaller antennas and transmit at a higher data rate. These properties make the code energy efficient. On the other hand, low-rate codes have a large overhead and are hence heavier on bandwidth consumption. Also, decoding complexity grows exponentially with code length, and long (low-rate) codes set high computational requirements to conventional decoders. According to Viterbi, this is the central problem of channel coding: encoding is easy but decoding is hard.The The European emergency call (eCall) system is a telematics system designed to save more lives in vehicle accidents. It is a governmental mandatory system that is to be implemented by March 2018. The EU eCall system provides an immediate voice and data channel between the vehicles and an emergency center after car accidents. The data channel provides the emergency center with the necessary data for emergency aids. The EU eCall system main parts includes the in-vehicle system (IVS), the public safety answering point (PSAP), a cellular communication channel. The IVS activates the data channel automatically when a car accident occurs. The IVS collects the minimum set of data (MSD) that includes GPS coordinates, the VIN number, and all required data for an emergency aid. It sends the MSD to the closest PSAP through a cellular channel in up to 4 seconds. The PSAP sends the emergency team to the location of the accidents.

The IVS modem employs multiple modules for the MSD signal processing. The modules of the IVS are shown in Figure 1. The IVS employs a Turbo encoder as a forward error correcting (FEC). The Turbo encoder implements the digital data encoding technique in data transmissions. Turbo coding is one of the most popular and efficient coding technique to improve bit error rate (BER) in digital communications. The cyclic redundancy check (CRC), the modulator, the demodulator-decoder modules are projected and implemented on an FPGA device. They are developed to be embedded modules of the IVS chip.



Fig. 1.1: The IVS block diagram.

This work studies the hardware development of the Turbo encoder. It employs FPGA technologies to develop the Turbo encoder to be an embedded module in the IVS modem. It discusses serial and parallel computation techniques for the Turbo encoder. It does not only design and implement the Turbo encoder module, but also proposes a better solution for the turbo encoder implementation. The improvement of the chip size and processing time are exhibited by developing the parallel computation technique for the Turbo encoder.

Chapter 2

**LITERATURE SURVEY**

Accompanying the developments of the decoding system was the exploration into why these codes perform well. This section analyses the structure of turbo codes to show how it was developed, how the understanding of the system matured and how this led to improved turbo code design The first development of the turbo encoder was by [JOE94] and later, [ROB94]. These authors highlighted the necessity for trellis termination, a subject omitted from the original publication. It was understood that, unlike non-systematic convolutional codes which can be forced to the all-zero state with only zeros, the RSC codes required different termination bits to be appended dependent on the final state of the encoder. Realising the optimal solution would be that both component trellises were terminated, but also that the presence of an interleaver in the turbo encoder meant that any tail bits appended to the data stream with regards to terminating the first encoder trellis at the all-zero state were unlikely to terminate the second trellis, [ROB94] showed a practical, though sub-optimal, solution by terminating the first component encoder to ensure that the trellis ended at the same state that it began.

The “eCall data transfer; in-band modem solution; general description,” 3GPP, This paper presents the hardware design and implementation of the in-vehicle system (IVS) for the European Union (EU) emergency call (eCall) system. Modules of the IVS are developed and implemented on a field programmable gate array (FPGA) device. The modules are simulated, synthesized, and optimized to be loaded on a reconfigurable device as a system-on-chip (SoC) for the IVS electronic device. Benchtop test is completed for testing and verification of the developed modules. The hardware architecture and interfaces are discussed. The IVS signal processing time is analyzed for multiple frequencies. A range of appropriate frequency and two hardware interfaces are proposed. A state-of-the-art FPGA design is employed as a first implementation approach for the IVS prototyping platform. This work can be used as an initial step to implement all the modules of the IVS on a single SoC chip.

A. Saleem et at. “Four-Dimensional Trellis Coded Modulation for FlexibleOptical Communications,” This paper studies design and implementation of the Turbo encoder to be an embedded module in the in-vehicle system (IVS) chip Field programmable gate array (FPGA) is employed to develop the Turbo encoder module Both serial and parallel computations for the encoding technique are studied The two design methods are presented and analyzed Developing the parallel computation method, it is shown that both chip size and processing time are improved The logic utilization is enhanced by 73% and the processing time is reduced by 58% The Turbo encoder module is designed, simulated, and synthesized using Xilinx tools Xilinx Zynq-7000 is employed as an FPGA device to implement the developed module The Turbo encoder module is designed to be a part of the IVS chip on a single programmable device Index Term: Turbo encoder module; field programmable gate array; emergency call; in-vehicle system chip

C. Studer, C. Benkeser, S. Belfanti, and Q. Huang “Design and Implementation of a Parallel Turbo-Decoder ASIC for 3GPP-LTE,” Turbo-decoding for the 3GPP-LTE (Long Term Evolution) wireless communication standard is among the most challenging tasks in terms of computational complexity and power consumption of corresponding cellular devices. This paper addresses design and implementation aspects of parallel turbo-decoders that reach the 326.4 Mb/s LTE peak data-rate using multiple soft-input soft-output decoders that operate in parallel. To highlight the effectiveness of our design-approach, we realized a 3.57 mm 2 radix-4based 8× parallel turbo-decoder ASIC in 0.13 μm CMOS technology achieving 390 Mb/s. At the more realistic 100 Mb/s LTE milestone targeted by industry today, the turbo-decoder consumes only 69 mW.

M. Nader and J. Liu, “Design and implementation of CRC module of eCall in-vehicle system on FPGA,” The EU emergency call (eCall) system is used as a vehicle emergency telematic system to reduce the fatalities and save more lives in vehicular incidents. We have designed and implemented the CRC module for the in-vehicle system (IVS) of the EU eCall on an FPGA device. As the CRC is a crucial part of the system to detect bit errors during the transmission, this paper presents the hardware design procedures of the CRC module. The system reads the 1120 serial input bits of the Minimum Set of Data (MSD), calculates the 28-bits of the CRC parity bits, and generates the MSD appended with CRC as the output signal that is consisting of 1148 serial bits. The system is designed in Verilog HDL, compiled, synthesized, and simulated for different MSDs. The results are shown and analyzed for varied applied MSDs. The flowchart of the implemented algorithm is illustrated and discussed. The system is tested and verified for different frequencies to see the range of the applicable frequencies of the design. We noted that the higher frequency we use for the clock source, the more distortion we get in the generated signal. The generated signals for the clock frequencies 50 kHz, 5 MHz, and 10 MHz are discussed. The simulated MSDs and generated signals on the FPGA are compared for multi cases to analyze the performance of the module. We also used a developed CRC module for IVS in C code to verify the performance of the module.

The “Technical Specification Group Radio Access Network; Multiplexing and channel coding (FDD),” 3GPP, Tech. Rep. TS22.212. design and implementation of the Turbo encoder to be an embedded module in the in-vehicle system (IVS) chip. Field programmable gate array (FPGA) is employed to develop the Turbo encoder module. Both serial and parallel computations for the encoding technique are studied. The two design methods are presented and analyzed. Developing the parallel computation method, it is shown that both chip size and processing time are improved. The logic utilization is enhanced by 73% and the processing time is reduced by 58%. The Turbo encoder module is designed, simulated, and synthesized using Xilinx tools. Xilinx Zynq-7000 is employed as an FPGA device to implement the developed module. The Turbo encoder module is designed to be a part of the IVS chip on a single programmable device.

The authors showed experimentally that terminating only the first encoder output and leaving the second trellis ‘open’ had only a small effect on turbo codes with sufficiently large frame sizes. [ROB94] was also amongst the first to show that the performance of turbo codes was affected by the construction of the interleaver, explaining how weight-2 input frames (the minimum weight of data that can cause an RSC encoder to diverge from the all-zero state and converge at some later point) that produced a low weight output could be permuted by a poorly chosen interleaver into another input that again produced a low weight codeword, thus counteracting one of the main objectives of the turbo encoder. To avoid this, the authors described a rather intensive approach to improving the interleaver, the system being based on the observation of all information sequences that cause low codewords, one after another and rearranging the interleaver to suit. The system was longwinded but did improve the flattening effect, or error floor, caused by less well-designed interleavers. With regard to hardware implementation of the turbo encoder, [DIV95] gives a simple solution to the termination problem (for a single component encoder). The authors add a switch between the input and the feedback loop. When receiving data, the encoder resembles the normal RSC encoder and for termination the feedback loop of the component encoder becomes the input to the encoder. Simple and effective, this system requires *m* bits, where *m* is memory length, to return to the all-zero state.

[DIV96], [BEN96] and [BEN96a] defined the effective free distance of a turbo code. Not to be confused with the minimum distance of a convolutional code, but having a similar effect for turbo codes, they showed it to be a function of the minimum parity weight caused by a weight-2 input to a turbo encoder and that to get the most out of a component code, especially at high signal-to-noise ratios, this value must be maximised. [BEN96a] also defined the maximum effective free distance for a RSC encoder with a single input stream and a particular encoder memory and produced a table of component convolutional codes that exhibited the best effective free distance and free distance for memory sizes ranging from 2 to 5. [DIV96] extended these tables to include multiple inputs. [PER96] explained the reasons behind the performance of turbo codes by showing the ability of the codes to cause “spectral thinning” [PER96], an expansion of the ideas in [ROB94]. This thinning of the distance spectrum, brought about by the presence of the interleaver, means that datawords that produce low weight outputs are likely to be permuted such that the new dataword produces a high weight output. Therefore the turbo codewords would consist mostly of average-weight outputs with a small number of low-weight outputs. The authors showed that “spectral thinning is enhanced by increasing interleaver lengths” [PER96] and that this would also lower the error floor for a fixed free distance. Conversely, increasing the free distance of the component codes could also lower the error floor.

The [HO98], [BEN98] and [HO98a] also investigated the effects of the distance properties of convolutional codes, producing extended results on previous papers for varying rates and memory sizes. Rather than simply examining the free distance and effective free distance, the authors of these publications widened the search for good component codes by investigating the qualities of codes for higher weight inputs. They also showed that the number of codewords with these distances was also an important factor when looking for the optimum component code and that the lower the number of possible codewords with these weights, the better the code performed.

Yuan *et al* showed, [YUA99], that the distance spectrum also plays an important role in designing the turbo encoder, especially at low signal-to-noise ratios. They show that choosing a

component code with the smallest error coefficients (Where an error coefficient is the average number of bit errors caused by code words of a particular weight and determines the contribution of those code words to the bit error probability) for a given interleaver size and for low to medium Hamming distances can outperform other codes in low to medium signal-to-noise ratios, even when the effective free distance is not optimal. In the research to understand the qualities of the turbo code structure it has also been necessary to investigate the error bounds of these codes. In order to determine the upper performance bounds for turbo codes [BEN96] treated the code as a systematic block code. The problem, however, was the inclusion of the interleaver. It is possible to determine the weight of the first parity sequence if the conditional weight enumerating function (WEF) is known, but the second parity sequence is not only dependent on the input weight but also on the sequence of the data after interleaving. [BEN96] proposed the use of a uniform interleaver, a probabilistic device based on an analysis of all possible interleavers, and showed that the method could be used to assess the bit error probabilities of turbo codes independent of the interleaver. As a result of this, the effect of the interleaver on the turbo encoder could be assessed, and therefore its gain. The authors showed in their analysis and proved by experimentation that the bit error probability reduced as interleaver size increased.

**2.1 Interleaver Design**

**The** design of, the ‘optimal’ interleaver, some more involved [HOS00], [DAN99], than others. Initial publications were basically trial and error based interleavers [BER93b], [JUN94], helping to show the improvements that were possible. Since then, many researchers have published papers on this aspect of turbo codes, some defining particular interleaver designs, others explaining systems that can be used to obtain good interleavers. [DIV95a] defined “S-random” interleavers, where S = *N* / 2 and N is the size of the interleaver. The method selects a random integer value within the interleaver size and compares it with S previously selected integers. If the chosen integer is equal to, or within + S positions, of one of the previous integers selected, then it is discarded. The process is repeated until all integers have been chosen.

[YUA99] defined the “Code Matched Interleaver” or CMI, where the authors compute the weight spectrum of the lower weight codewords then use performance analysis to determine the inputs that make large contributions to the error probability at high signal-to-noise ratios. The interleaver is then designed such that these patterns are not present after interleaving. Modifying the S-random interleaver of [DIV95a] after the comparison with previous integer choices, the system checks whether the interleaver produces an output that is undesirable. That being the case, the integer is again rejected and a new one selected. If no integer can satisfy both the comparison with previous choices and the output word control, the value of S is reduced by one and a new interleaver is searched for. Used in conjunction with good component codes, this system significantly lowered the error floor of previous designs.

The [BYU99] also further developed the S-random interleaver. The authors pointed out that, for frame sizes larger than around 1000 bits, it is almost impossible to use an S value as described by [DIV95a]. Their design, called the swap interleaver, begins with a block interleaver of equal depth and span. Two random positions within the interleaver are chosen and swapped. These positions are then checked against the given S value and if this is not satisfied they are returned to their previous positions. After a sufficient number of iterations (the authors propose 100 times the frame length) the design is complete. The authors found that the search time was vastly reduced, especially where large values of S were desired and that the performance was in excess of that of the standard S-random interleaver.

The [HO98b] looked at interleavers for punctured turbo codes, noting that puncturing often degrades the performance of turbo codes. The authors show that this is due to uneven parity bit protection. For example, using the common turbo code puncturing method (delete all even bits from the first parity bit stream and all odd bits from the second parity bit stream) may mean that one particular data bit has two parity bits, whereas another has none. This is easier to comprehend if one assumes that the position of the data bit prior to interleaving was in an odd position and after interleaving was in an even position, in which case, that particular information bit has a corresponding parity bit in both received codewords. To rectify this, the authors introduce the mod-*k* interleaver an extension of the odd-even interleaver (mod-2) described in [BAR94]. The odd-even interleaver permutes odd data bits to odd positions and even data bits to even positions, thus ensuring that all data bits will be transmitted with one parity bit after puncturing. The authors of [HO98b] also considered the fact that most interleavers require a corresponding de-interleaver, which, in implementation, doubles the storage. To combat this, they propose the use of symmetric interleavers, one piece of hardware or look-up table that both interleaves and de-interleaves. Through experimentation it was proved that a mod-*k* interleaver could bring about an improvement in performance and that combining the two theories (symmetric and mod-*k*) increased performance even more. Comparing interleaver designs, the gain of the S-symmetricmod- 2 interleaver was shown to improve turbo code performance when compared with its S-random counterpart.

Chapter 3

**INTRODUCTION OF VLSI**

Very-large-scale integration (VLSI) is the process of creating integrated circuits by combining thousands of transistor-based circuits into a single chip. VLSI began in the 1970s when complex semiconductor and communication technologies were being developed. The microprocessor is a VLSI device. The term is no longer as common as it once was, as chips have increased in complexity into the hundreds of millions of transistors.

## 3.1Overview

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 first integrated circuits held only a few devices, perhaps as many as ten diodes, transistors, resistors and capacitors, making it possible to fabricate one or more logic gates on a single device. Now known retrospectively as "small-scale integration" (SSI), improvements in technique led to devices with hundreds of logic gates, known as large-scale integration (LSI), i.e. systems with at least a thousand logic gates. Current technology has moved far past this mark and today's microprocessors have many millions of gates and hundreds of millions of individual transistors.

At one time, there was an effort to name and calibrate various levels of large-scale integration above VLSI. Terms like Ultra-large-scale Integration (ULSI) were used. But the huge number of gates and transistors available on common devices has rendered such fine distinctions moot. Terms suggesting greater than VLSI levels of integration are no longer in widespread use. Even VLSI is now somewhat quaint, given the common assumption that all microprocessors are VLSI or better.

As of early 2008, billion-transistor processors are commercially available, an example of which is Intel's Montecito Itanium chip. This is expected to become more commonplace as semiconductor fabrication moves from the current generation of 65 nm processes to the next 45 nm generations (while experiencing new challenges such as increased variation across process corners). Another notable example is NVIDIA’s 280 series GPU.

This microprocessor is unique in the fact that its 1.4 Billion transistor count, capable of a teraflop of performance, is almost entirely dedicated to logic (Itanium's transistor count is largely due to the 24MB L3 cache). Current designs, as opposed to the earliest devices, use extensive design automation and automated logic synthesis to lay out the transistors, enabling higher levels of complexity in the resulting logic functionality. Certain high-performance logic blocks like the SRAM cell, however, are still designed by hand to ensure the highest efficiency (sometimes by bending or breaking established design rules to obtain the last bit of performance by trading stability).

**3.2What is VLSI?**

## VLSI stands for "Very Large Scale Integration". This is the field which involves packing more and more logic devices into smaller and smaller areas.

## VLSI

## Simply we say Integrated circuit is many transistors on one chip.

## Design/manufacturing of extremely small, complex circuitry using modified semiconductor material

## Integrated circuit (IC) may contain millions of transistors, each a few mm in size

## Applications wide ranging: most electronic logic devices

# 3.3History of Scale Integration

* late 40s Transistor invented at Bell Labs
* late 50s First IC (JK-FF by Jack Kilby at TI)
* early 60s Small Scale Integration (SSI)
* 10s of transistors on a chip
* late 60s Medium Scale Integration (MSI)
* 100s of transistors on a chip
* early 70s Large Scale Integration (LSI)
* 1000s of transistor on a chip
* early 80s VLSI 10,000s of transistors on a
* chip (later 100,000s & now 1,000,000s)
* Ultra LSI is sometimes used for 1,000,000s
* SSI - Small-Scale Integration (0-102)
* MSI - Medium-Scale Integration (102-103)
* LSI - Large-Scale Integration (103-105)
* VLSI - Very Large-Scale Integration (105-107)
* ULSI - Ultra Large-Scale Integration (>=107)

### 3.4Advantages of ICs over discrete components

While we will concentrate on integrated circuits, the properties of integrated circuits-what we can and cannot efficiently put in an integrated circuit-largely determine the architecture of the entire system. Integrated circuits improve system characteristics in several critical ways. ICs have three key advantages over digital circuits built from discrete components:

* + Size. Integrated circuits are much smaller-both transistors and wires are shrunk to micrometer sizes, compared to the millimetre or centimetre scales of discrete components. Small size leads to advantages in speed and power consumption, since smaller components have smaller parasitic resistances, capacitances, and inductances.
  + Speed. Signals can be switched between logic 0 and logic 1 much quicker within a chip than they can between chips. Communication within a chip can occur hundreds of times faster than communication between chips on a printed circuit board. The high speed of circuits on-chip is due to their small size-smaller components and wires have smaller parasitic capacitances to slow down the signal.
  + Power consumption. Logic operations within a chip also take much less power. Once again, lower power consumption is largely due to the small size of circuits on the chip-smaller parasitic capacitances and resistances require less power to drive them.

#### 3.5VLSI and systems

These advantages of integrated circuits translate into advantages at the system level:

* + Smaller physical size. Smallness is often an advantage in itself-consider portable televisions or handheld cellular telephones.
  + Lower power consumption. Replacing a handful of standard parts with a single chip reduces total power consumption. Reducing power consumption has a ripple effect on the rest of the system: a smaller, cheaper power supply can be used; since less power consumption means less heat, a fan may no longer be necessary; a simpler cabinet with less shielding for electromagnetic shielding may be feasible, too.
  + Reduced cost. Reducing the number of components, the power supply requirements, cabinet costs, and so on, will inevitably reduce system cost. The ripple effect of integration is such that the cost of a system built from custom ICs can be less, even though the individual ICs cost more than the standard parts they replace.

Understanding why integrated circuit technology has such profound influence on the design of digital systems requires understanding both the technology of IC manufacturing and the economics of ICs and digital systems.

##### **3.6Applications**

* Electronic system in cars.
* Digital electronics control VCRs
* Transaction processing system, ATM
* Personal computers and Workstations
* Medical electronic systems.

### 3.7Applications of VLSI

Electronic systems now perform a wide variety of tasks in daily life. Electronic systems in some cases have replaced mechanisms that operated mechanically, hydraulically, or by other means; electronics are usually smaller, more flexible, and easier to service. In other cases electronic systems have created totally new applications. Electronic systems perform a variety of tasks, some of them visible, some more hidden:

* + Personal entertainment systems such as portable MP3 players and DVD players perform sophisticated algorithms with remarkably little energy.
  + Electronic systems in cars operate stereo systems and displays; they also control fuel injection systems, adjust suspensions to varying terrain, and perform the control functions required for anti-lock braking (ABS) systems.
  + Digital electronics compress and decompress video, even at high-definition data rates, on-the-fly in consumer electronics.
  + Low-cost terminals for Web browsing still require sophisticated electronics, despite their dedicated function.
  + Personal computers and workstations provide word-processing, financial analysis, and games. Computers include both central processing units (CPUs) and special-purpose hardware for disk access, faster screen display, *etc*.
  + Medical electronic systems measure bodily functions and perform complex processing algorithms to warn about unusual conditions. The availability of these complex systems, far from overwhelming consumers, only creates demand for even more complex systems.

The growing sophistication of applications continually pushes the design and manufacturing of integrated circuits and electronic systems to new levels of complexity.

And perhaps the most amazing characteristic of this collection of systems is its variety-as systems become more complex, we build not a few general-purpose computers but an ever wider range of special-purpose systems. Our ability to do so is a testament to our growing mastery of both integrated circuit manufacturing and design, but the increasing demands of customers continue to test the limits of design and manufacturing

Chapter 4

**TURBO ENCODER MODULE**

The turbo encoder technique is one of the most powerful FEC techniques in digital communication. The IVS employs a Turbo encoder module with 1=3 code rate. The Turbo encoder functionalities are detailed in the third generation partnership project (3GPP) standards. The 3GPP Turbo encoder is illustrated in Figure 1. The input signal of the turbo encodes is the MSD data appended with the CRC parity bits in binary. The block length of the MSD data is 1148 bits. The output of the module is the MSD encoded data in binary. Implementing the turbo coding technique with 1=3 coding rate and thrills bits, the length of the output is 3456 bits. The thrills structure has an impact of the Turbo encoder.

The Turbo encoder employs a parallel concatenated convolution code (PCCC). The PCCC uses two constituent encoders with eight states as it is shown in Figure5.1. The initial status of the register are zeros. The first constituent takes the MSD bits and implements the employed convolutional technique. It takes one bit at a time and generates one bit of parity1 bits. The second constituent implements an identical technique of the first constituent, but it calls for the MSD bit after they are interleaved with a 3GPP designed interleaver technique.

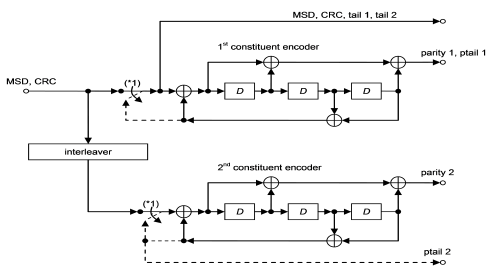


Fig. 4.1: The structure of the Turbo encoder.

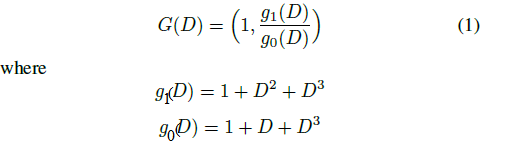
The length of the input data, parity1, and parity2 are 1148 bits. There are 12 bits of the tail bits. They are driven from the shift register feedback. The tail bits are applied for end points between the encoded data blocks. The output structure of the Turbo encoder is illustrated in Figure6.1.



Fig. 4.2: The output buffer of the Turbo encoder.

**4.1 Interleaver**

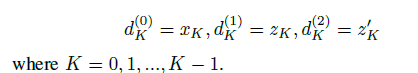
Denote the transfer function of the employed PCCC as:



And denote the input bits to the encoder as the output of the interleaver as  and the output bits of the first and second constituents as  and

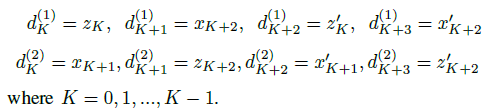
respectively; where K is the number of input bits to the Turbo encoder.

The encoder output is expressed as:



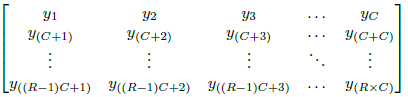
The three code blocks of the output, , K are separated by trellis bits. The trellis bits are generated from the tail bits of the shift registers after encoding of all the input bits. In figure 2, when the upper switch is lowered and the second constituent is disabled, the three tail bits are used to terminate the first constituent. The output bits of the Turbo encoder, including the trellis bits can be expressed as:





The internal interleaver of the 3GPP Turbo encoder is designed to generate a systematic relationship between  and  for any . There is a specific approach to design an internal interleaver for the employed Turbo encoder that is detailed in. This work employs the 3GPP standard approach to design the internal interleaver for the employed Turbo encoder.

First, the input bits of the Turbo encoder is re-arranged in a matrix form that has column, C, and row, R. The rows are labeled as 0,1,…,R - 1 and the columns are organized as 0,1,…,C-1. The numbers of rows and columns are determined according to the 3GPP standard for Turbo encoder interleavers. Then the input bits are re-organized in a matrix where  for k = 1,2,…,K and  for the elements that 



Then an intra-row and inter-row permutation is performedon the R X C matrix. This work employs the 3GPP standard approaches for the intra-row and inter-row.

Denote,



Where  for and p(0) = 0 is the intarow permutation sequence and v is associated primitive root for the specified p from table ??, and use table I to choose the appropriate pattern to compute the inter-row permutation, T(i) for . i is the row number index of R X C matrix, and j is the column number index of the matrix. Also the minimum prime integer (qi) is determined in the sequence q(i) for such that qi > q(i-1), qi > 6 and g.c.d(qi, p - 1) = 1, where g.c.d is the greater common divisor. Then one can build a sequence of the permuted prime integers D r(i) for  such that,

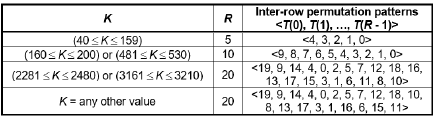
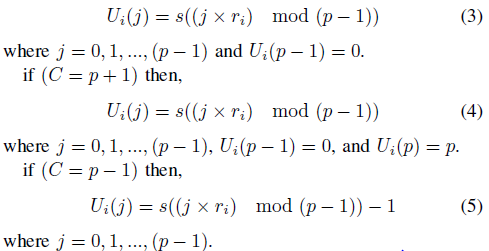
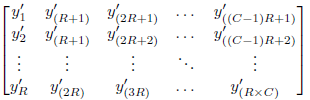


Table 5.1: 3GPP inter-row permutation pattern

Denote the pattern of i \_ th row Intra-row permutation as, 

And then the inter-row permutation is implemented on the R X C matrix by using the sequence pattern T(i) for i = 0,1,2…,R-1.After the permutations, the elements of the R X C matrix is denoted by  such that:



The output of the interleaver, K, is the bit sequence that are read out from the RXC matrix column by column starting with and ending with. The appended zero bits for R - C > K are removed from the output. There are 1148 elements in the interleaver matrix. The 1148 bits of the MSD data are the elements of the interleaver matrix. The interleaver reorganizes the MSD bits in a systematic order. The intra-row and inter-row techniques of the interleaver elements organizations. Denote the MSD bits as B1,B2,…,BK, where K = 1148. According to the algorithm that is explained in the above mathematical modeling, the interleaver matrix is a rectangular matrix, where R is the number of rows and C is the number of columns of the matrix. The interleaver matrix is designed for an input data block that consists of 1148 bits. As a result, the size of the matrix is 20 X 58. The following steps are implemented to drive the interleaver matrix:

1. The input bits of the matrix are denoted as b1, b2,…, bK, where bK = BK and K = 1148. The remaining elements are padded with zeros.

2. The intra-row and inter-row permutation are performed according to 3GPP.

3. The calculated elements of the interleaver matrix, except for the padded bits, are stored in a file in hexadecimal format.

The Turbo encoder is developed in Verilog HDL language. Verilog has ability to read the hexadecimal file to get the data and use it as the interleaver data. The length of the input data, parity1, and parity2 are 1148 bits. There are 12 bits of the tail bits. They are driven from the shift register feedback. The tail bits are applied for end points between the encoded data blocks. The output structure of the Turbo encoder is illustrated in Figure 3. The employed interleaver for the Turbo encoder is designed according to 3GPP standards [8]. The interleaver elements are organized in a rectangular matrix. There are 1148 elements in the interleaver matrix. The 1148 bits of the MSD data are the elements of the interleaver matrix. The interleaver reorganizes the MSD bits in a systematic order. There are intra-row and inter-row techniques of the interleaver elements organizations. Denote the MSD bits as B1,B2,…,BK, where K = 1148.

According to the 3GPP standards , the interleaver matrix is a RxC rectangular matrix, where R is the number of rows and C is the number of columns. The size of the matrix is 20 X 58. The following steps are implemented to drive the

interleaver matrix:

1. The input bits of the matrix are denoted as b1, b2,…, bK,where bK = BK and K = 1148. The remaining elements are padded with zeros.

2. The intra-row and inter-row permutation is performed according to 3GPP .

3. The calculated elements of the interleaver matrix, except for the padded bits, are stored in in a file in hexadecimal format.

The Turbo encoder is developed in Verilog HDL language. Verilog has ability to read the hexadecimal file to get the data and use it as the interleaver data.

The technologies are employed to develop and implement the designed Turbo encoder module. The register transfer level (RTL) of the module is developed in Verilog HDL. There are multiple registers defined for the input, output, and necessary parameters to implement the Turbo encoding technique. This work studies two methods to execute the encoding, which are serial computation and parallel computation.

**4.1. The serial computation method**

The serial computation method processes one bit in one clock cycle. It reads the input data of the MSD, builds the input and output registers, and calculates the parity1, parity2, and the tail bits in a serial process. After performing the encoding, it generates the output bits. Although the method is designed and implemented, it is noted that there is a long processing time that can be overlapped with the other processes in the module. Figure 7.1 shows the pseudocode of the serial computation of the Turbo encoder module.

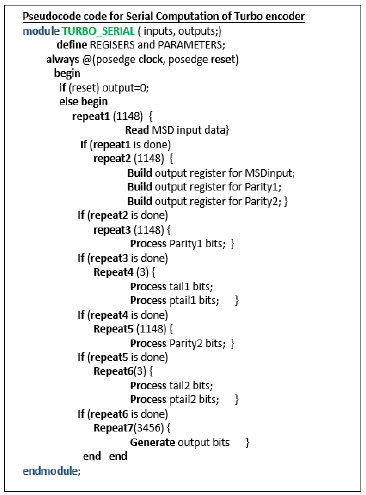


Fig.: The pseudo code for serial computation of the Turbo encoder.

The processing time of the Turbo encoder module (in clock cycles) is denoted by Ts for the serial computation,





where Tr is the time for reading the 1148 bits of the MSD, Tb is the processing time to build the output register, Tparit1 is the time of processing parity bits, Ttail is the time of processing tail bits, and Tw is the time of generating output bits.

**4.2 The parallel computation method**

The parallel computing technique is employed to develop the Turbo encoder in Verilog. There are many processes in the serial computation technique that are overlapped by using parallel computing technique. There are two functions developed in the parallel Turbo encoder. The two functions implements almost all the processing time of the encoding technique. The Turbo encoding technique needs the MSD data as a whole package to implement the encoding. Figure 7.2 shows the pseudocode of the implemented parallel technique for the Turbo encoder. Both Pseudocodes of the serial and parallel computing techniques are designed in Verilog.

The processing time of the Turbo encoder module (in clock cycles) is denoted by Ts for the serial computation and by Tp for the parallel technique, one has,





where Tr is the time for reading the 1148 bits of the MSD, Tb is the processing time to build the output register, Tparit1 is the time of processing parity bits, Ttail is the time of processing tail bits, and Tw is the time of generating output bits. Note that the Tr +Tb +Tparity1 +Ttail1 +Tparity2 +Ttail2 are processed in one clock cycle in the parallel computing technique.



Then one has,



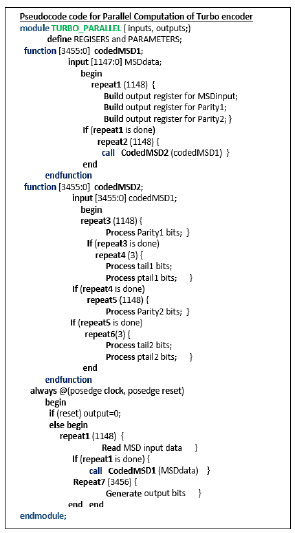


Fig: The pseudo code for parallel computation of the Turbo encoder.

**ADDER DESIGNS**

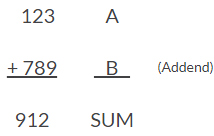
**Adders:**

An **adder** is a digital circuit that performs addition of numbers. In many computers and other kinds of processors adders are used in the arithmetic logic units or **ALU**. They are also used in other parts of the processor, where they are used to calculate addresses, table indices, increment and decrement operators and similar operations.

Although adders can be constructed for many number representations, such as binary-coded decimal or excess-3, the most common adders operate on binary numbers. In cases where two's complement or ones' complement is being used to represent negative numbers, it is trivial to modify an adder into an adder–subtractor. Other signed number representations require more logic around the basic adder.

Another common and very useful combinational logic circuit which can be constructed using just a few basic logic gates allowing it to add together two or more binary numbers is the **Binary** A basic Binary Adder circuit can be made from standard AND and Ex-OR gates allowing us to “add” together two single bit binary numbers, A and B.

The addition of these two digits produces an output called the SUM of the addition and a second output called the CARRY or Carry-out, ( COUT ) bit according to the rules for binary addition. One of the main uses for the Binary Adder is in arithmetic and counting circuits. Consider the simple addition of the two denary (base 10) numbers below.



From our maths lessons at school, we learnt that each number column is added together starting from the right hand side and that each digit has a weighted value depending upon its position within the columns.

When each column is added together a carry is generated if the result is greater or equal to 10, the base number. This carry is then added to the result of the addition of the next column to the left and so on, simple school math’s addition, add the numbers and carry.

The adding of binary numbers is exactly the same idea as that for adding together decimal numbers but this time a carry is only generated when the result in any column is greater or equal to “2”, the base number of binary. In other words 1 + 1 creates a carry.

## 2Binary Addition

**Binary Addition** follows these same basic rules as for the denary addition above except in binary there are only two digits with the largest digit being “1”. So when adding binary numbers, a carry out is generated when the “SUM” equals or is greater than two (1+1) and this becomes a “CARRY” bit for any subsequent addition being passed over to the next column for addition and so on. Consider the single bit addition below.

### Binary Addition of Two Bits

### 

### When the two single bits, A and B are added together, the addition of “0 + 0”, “0 + 1” and “1 + 0” results in either a “0” or a “1” until you get to the final column of “1 + 1” then the sum is equal to “2”. But the number two does not exists in binary however, 2 in binary is equal to 10, in other words a zero for the sum plus an extra carry bit.

Then the operation of a simple adder requires two data inputs producing two outputs, the Sum (S) of the equation and a Carry (C) bit as shown.

**Half-Adder:**

It adds two binary digits called augend and addend produces two outputs as “ sum” and “carry”

1. Half adder is a digital circuit used for arithmetic operation.
2. It performs the addition of two “single bits”.
3. It contains only two inputs and two outputs (sum and carry bits).

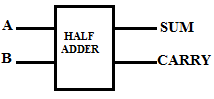


Fig: logic symbol of half adder

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | CARRY | SUM |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Fig: truth table

A, B are inputs and carry, sum are outputs.

Sum = A xor B

Carry=AandB



**Full-Adder:**

1. Full adder is a logic circuit that performs the sum of “three single bits”.
2. It can also design using two half-adders.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | SUM | Co |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Fig: truth table

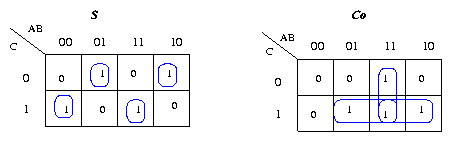


Fig: k-maps for sum and carry

Sum value is calculated using S= A xor B xor C

carry value is calculated using Co = (A and B) or (B and C) or (A and c).

**n-bit Binary Adder**

We have seen above that single 1-bit binary adders can be constructed from basic logic gates. But what if we wanted to add together two n-bit numbers, then n number of 1-bit full adders need to be connected or “cascaded” together to produce what is known as a Ripple Carry Adder.

A “ripple carry adder” is simply “n“, 1-bit full adders cascaded together with each full adder representing a single weighted column in a long binary addition. It is called a ripple carry adder because the carry signals produce a “ripple” effect through the binary adder from right to left, (LSB to MSB).

For example, suppose we want to “add” together two 4-bit numbers, the two outputs of the first full adder will provide the first place digit sum (S) of the addition plus a carry-out bit that acts as the carry-in digit of the next binary adder.

The second binary adder in the chain also produces a summed output (the 2nd bit) plus another carry-out bit and we can keep adding more full adders to the combination to add larger numbers, linking the carry bit output from the first full binary adder to the next full adder, and so forth. An example of a 4-bit adder is given below.

A 4-bit Ripple Carry Adder

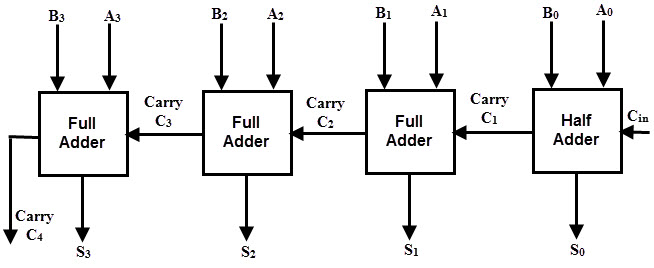
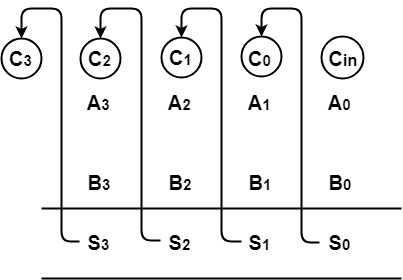


Fig: 4-bit binary parallel adder

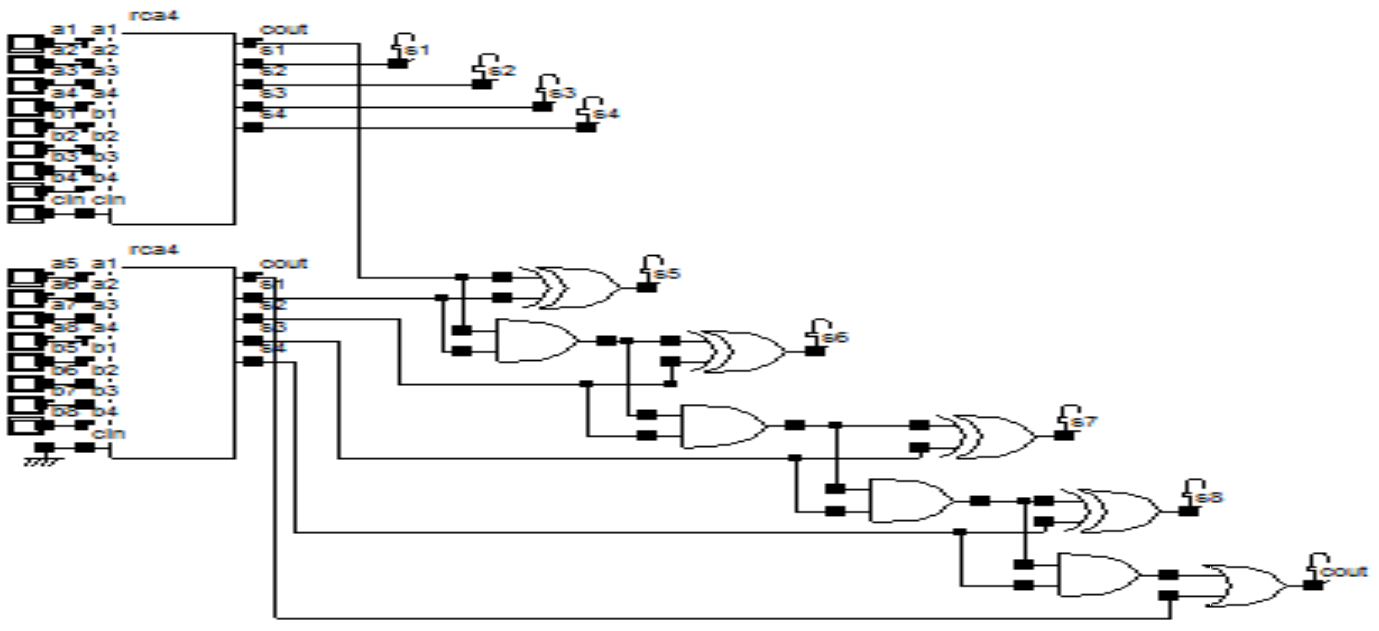
* 4-bit ripple carry adder is used for the purpose of adding two 4-bit binary numbers.
* In mathematics, any two 4-bit binary numbers A3A2A1A0 and B3B2B1B0 will be added as-



As shown, Ripple Carry Adder works in different stages where the carry out produced by each full adder as output serves as the carry in input for its adjacent most significant full adder. When the carry in becomes available to the full adder, it activates that full adder and it comes into operation.

**CARRY INCREMENT ADDER**

An 8-bit increment adder includes two RCA (Ripple carry adder) of four bit each. The first ripple carry adder adds a desired number of first 4-bit inputs generating a plurality of partitioned sum and partitioned carry. Now the carry out of the first block RCA is given to CIN of the conditional increment block. Thus the first four bit sum is directly taken from the ripple carry output. The second RCA block regardless of the first RCA output will carry out the addition operation and will give out results which are fed to the conditional increment block. The input CIN to the first RCA block is given always low value. The conditional increment block consists of half adders. Based on the value of cout of the 1st RCA block, the increment operation will take place. Here the half adder in carry increment block performs the increment operation. Hence the output sum of the second RCA is taken through the carry increment block. The design schematic of Carry Increment Adder is shown in Figure.



**Figure : carry increment adder**

**Chapter5**

**VERILOG**

**5.1 Introduction**

* Verilog synthesis tools can create logic-circuit structures directly from verilog behavioral description and target them to a selected technology for realization (I.e,translate verilog to actual hardware).
* Using verilog , we can design ,simulate and synthesis anything from a simple combinational circuit to a complete microprocessor on chip.
* Verilog HDL has evolved as a standard hardware description language. Verilog HDL offers many useful features for hardware design.
* Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.
* 1Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer needs to learn only one language for stimulus and hierarchical design.
* Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
* All fabrication vendors provide Verilog HDL libraries for post logic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors.
* The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their needs with the PLI.

### 5.2 History Of Verilog HDL

* Verilog was started initially as a proprietary hardware modeling language by Gateway Design Automation Inc. around 1984. It is rumored that the original language was designed by taking features from the most popular HDL language of the time, called HiLo, as well as from traditional computer languages such as C. At that time, Verilog was not standardized and the language modified itself in almost all the revisions that came out within 1984 to 1990.
* Verilog simulator was first used beginning in 1985 and was extended substantially through 1987. The implementation was the Verilog simulator sold by Gateway. The first major extension was Verilog-XL, which added a few features and implemented the infamous "XL algorithm" which was a very efficient method for doing gate-level simulation.
* The time was late 1990. Cadence Design System, whose primary product at that time included Thin film process simulator, decided to acquire Gateway Automation System. Along with other Gateway products, Cadence now became the owner of the Verilog language, 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.
* In 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 the 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.
* 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.
* Soon it was realized that if there were too many companies in the market for Verilog, potentially everybody would like to do what Gateway had done so far - changing the language for their own benefit. This would defeat the main purpose of releasing the language to public domain. As a result 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.
* 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.
* In the meantime, the popularity of Verilog and PLI was rising exponentially. Verilog as a HDL found more admirers than well-formed and federally funded VHDL. It was only a matter of time before people in OVI realized the need of a more universally accepted standard. Accordingly, the board of directors of OVI requested IEEE to form a working committee for establishing Verilog as an IEEE standard. The working committee 1364 was formed in mid 1993 and on October 14, 1993, it had its first meeting.
* The standard, which combined both the Verilog language syntax and the PLI in a single volume, was passed in May 1995 and now known as IEEE Std. 1364-1995.
* After many years, new features have been added to Verilog, and the new version is called Verilog 2001. This version seems to have fixed a lot of problems that Verilog 1995 had. This version is called 1364-2001.

**5.3. Program Structure**

* The basic unit and programming in verilog is ”MODULE”(a text file containing statements and declarations ).
* A verilog module has declarations that describes the names and types of the module inputs and outputs as well as local signals, variables, constants and functions that are used internally to the module ,are not visible outside.
* The rest of the module contains statements that specify the operation of the module output and internal signals.
* Verilog is a case-sensitive language like C. Thus sense, Sense, SENSE, sENse,…etc., are all treated as different entities / quantities in Verilog.

**SYNTAX:**

Module Module\_Name( port list);

Port declaration

Function declaration

Endmodule

**module** 🡨 signifies the beginning of a module definition.

**endmodule** 🡨signifies the end of a module definition.

**IDENTIFIERS:**

Any program requires blocks of statements, signals, etc., to be identified with an attached nametag. Such nametags are identifiers.

There are some restrictions in assigning identifier names. All characters of the alphabet or an underscore can be used as the first character. Subsequent characters can be of alphanumeric type, or the underscore (\_), or the dollar ($) sign .

For example

* name, \_name. Name, name1, name\_$, . . . all these are allowed asidentifiers
* name aa 🡨not allowed as an identifier because of the blank ( “name” and “aa”are interpreted as two different identifiers)
* $name 🡨 not allowed as an identifier because of the presence of “$” as the firstcharacter.1\_name not allowed as an identifier, since the numeral “1” is the first character @name not allowed as an identifier because of the presence of the character “@”.
* A+b not allowed as an identifier because of the presence of the character “+”.

An alternative format makes it is possible to use any of the printable ASCII characters in an identifier. Such identifiers are called “escaped identifiers”; they have to start with the backslash (\) character. The character set between the first backslash character and the first white space encountered is treated as an identifier. The backslash itself is not treated as a character of the identifier concerned.

Examples

\b=c

\control-signal

\&logic

\abc // Here the combination “abc” forms the identifier.

**WHITE SPACE CHARACTERS**

Blanks (\b), tabs (\t), newlines (\n), and form feed form the white space characters in Verilog. In any design description the white space characters are included to improve readability.

**COMMENTS**

It is a healthy practice to comment a design description liberally –A single line comment begins with “//” and ends with a new line, and for multiple comments starts with “\\*” and ends with”\*\”.

**PORT DECLERATION:**

Verilog module declaration begins with a keyword ”module” and ends with”endmodule”. The input and output ports are signals by which the module communicates with each others.

Syntax:

Input identifier………………..identifier;

Output identifier………………..identifier;

Inout identifier………………..identifier;

Input [msb:lsb] identifier………………..identifier;

Output[msb:lsb] identifier………………..identifier;

Inout [msb:lsb] identifier………………..identifier;

**LOGIC SYSTEM:**

Verilog uses 4 –logic system .a 1 –bit signal can take one of only four possible values.

0 LOGIC 0,OR FALSE

1 LOGICAL 1,OR FALSE

X A UNKNOWN LOGICAL VALUE

Z HIGH IMPEDENCE

**5.4 Operators**

**Arithmetic Operators:**

These perform arithmetic operations. The **+** and **-** can be used as either unary (-z) or binary (x-y) operators.

+ (addition)

- (subtraction)

\* (multiplication)

/ (division)

% (modulus)

**Relational Operators:**

Relational operators compare two operands and return a single bit 1or 0. These operators synthesize into comparators.

< (less than)

<= (less than or equal to)

> (greater than)

>= (greater than or equal to)

== (equal to)

!= (not equal to)

**Bit-wise Operators:**

Bit-wise operators do a bit-by-bit comparison between two operands. However see “Reduction Operators” .

~ (bitwise NOT)

& (bitwise AND)

| (bitwiseOR)

^ (bitwise XOR)

~^ or ^~ (bitwise XNOR)

**Logical Operators:**

Logical operators return a single bit 1 or 0. They are the same as bit-wise operators only for single bit operands. They can work on expressions, integers or groups of bits, and treat all values that are nonzero as “1”. Logical operators are typically used in conditional (**if** ... **else**) statements since they work with expressions.

! (logical NOT)

&& (logical AND)

|| (logical OR)

**Reduction Operators**

Reduction operators operate on all the bits of an operand vector and return a single-bit value. These are the unary (one argument) form of the bit-wise operators above.

& (reduction AND)

| (reduction OR)

~& (reduction NAND)

~| (reduction NOR)

^ (reduction XOR)

~^ or ^~ (reduction XNOR)

**Shift Operators**

Shift operators shift the first operand by the number of bits specified by the second operand. Vacated positions are filled with zeros for both left and right shifts (There is no sign extension).

<< (shift left)

>> (shift right)

**Concatenation Operator:**

The concatenation operator combines two or more operands to form a larger vector

{ } (concatenation)

**Replication Operator:**

The replication operator makes multiple copies of an item.

{n{item}} (n fold replication of an item)

**Literals:**

Literals are constant-valued operands that can be used in Verilog expressions. The two common Verilog literals are:

(a) String: A string literal is a one-dimensional array of characters enclosed in double quotes(““).

(b) Numeric: constant numbers specified in binary, octal, decimal or hexadecimal.

Number Syntax

n’Fddd..., where

n - integer representing number of bits

F - one of four possible base formats:

b (binary), o (octal), d (decimal),h (hexadecimal). Default is d.

Literals written without a size indicator default to 32-bits or the word width used by the simulator program, this may cause errors, so we should careful with unsized literals.

**NET**:

Verilog actually has two classes of signals

1. nets.

2. variables.

* Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to.
* The default net type is wire, any signal name that appears in a module input /output list, but not in a net declaration is assumed to be type wire.
* Nets are one-bit values by default unless they are declared explicitly as vectors. The terms wire and net are often used interchangeably.
* Note that net is not a keyword but represents a class of data types such as wire, wand, wor, tri, triand, trior, trireg, etc. The wire declaration is used most frequently.
* The syntax of verilog net declaration is similar to an input/output declaration.

Syntax:

Wire identifier ,………….. identifier;

Wire [msb:lsb] identifier ,………….. identifier;

tri identifier ,………….. identifier;

tri [msb:lsb] identifier ,………….. identifier;

* The keyword tri has a function identical to that of wire. When a net is driven by more than one tri-state gate, it is declared as tri rather than as wire. The distinction is for better clarity.

**VARIABLE:**

* Verilog variables stores the values during the program execution, and they need not have

Physical significance in the circuit.

* They are used in only procedural code (i.e,behavioral design).A variable value can be used ina expression and can be combined and assign to other variables, as in conventional software programming language.
* The most commonly used variables are REG and INTEGERS.

Syntax:

Reg identifier ,………….identifier;

Reg [msb:lsb] identifier,………….identifier;

Integer identifier ,………….identifier;

* A register variable is a single bit or vector of bits , the value of 1-bit reg variable is always 0,1,X,Z. the main use of reg variables is to store values in Verilog procedural code.
* An integer variable value is a32-bit or larger integer ,depending on the word length on the word length used by simulator .An integer variable is typically used to control a repetitive statements ,such as loop, in Verilog procedural code.

**PARAMETER:**

Verilog provides a facility for defining named constants within a module ,to improve readability and maintainability of code. The parameter declaration is

Syntax:

Parameter identifier =value;

Parameter identifier =value,

**: :**

identifier =value;

* An identifier is assigned to a constant value that will be used in place of the identifier throughout the current module.
* Multiple constants can be defined in a single parameter declaration using a comma –separated list of arguments.
* The value in the parameter declaration can be simple constant ,or it can be a constant expression.
* An expression involving multiple operators and constants including other parameters ,that yields a constant result at compile time. The parameter scope is limited to that module in which it is defined.

**ARRAYS:**

Arrays are allowed in Verilog for reg, integer, time, and vector register data types. Arrays are not allowed for real variables. Arrays are accessed by <array\_name> [<subscript>]. Multidimensional arrays are not permitted in Verilog.

Syntax:

Reg identifier [start:end];

Reg [msb:lsb] identifier [start:end];

Integer identifier [start:end] ;

Example: integer count [0: 7] ; I I An array of 8 count variables

**5.5 Dataflow Design Elements**

Continuous assignment statement allows to describe a combinational circuit in terms of the flow of data and operations on the circuit. This style is called “dataflow design or description”. The basic syntax of a continuous –assignment statement in Verilog is

Syntax:

Assign net-name=expression;

Assign net-name[bit-index]=expression;

Assign net-name[msb:lsb]=expression;

Assign net-concatenation =expression;

* “**Assign**” is the keyword carrying out the assignment operation. This type of assignment is called a continuous assignment.
* The keyword “assign ”is followed by the name of a net, then an”=”sign and finally an expression giving the value to be assigned
* If a module contains two statements “assign X=Y” and “assign Y=~X”, then the simulation will loop “forever”(until the simulation times out).

For example:

**assign** c = a && b;

* a and b are operands – typically single-bit logic variables.
* “&&” is a logic operator. It does the bit-wise AND operation on the two
* operands a and b.
* “=” is an assignment activity carried out.
* c is a net representing the signal which is the result of the assignment.

**5.6 Structural Design (Or) Gate Level Modeling**

* Structural Design Is the Series of Concurrent Statement .The Most Important Concurrent Statement In the module covered like instance statements, continuous –assignment statement and always block. These gives rise to three distinct styles of circuit design and description.
* Statement of these types, and corresponding design styles, can be freely intermixed within a Verilog module declaration.
* Each concurrent statement in a Verilog module “executes” simultaneously with other statements in the same module declaration.
* In Verilog module, if the last statement updates a signal that is used by the first statement, then the simulator goes back to that first statement and updates its result.
* In fact, the simulator will propagate changes and updating results until the simulated circuit stabilizes.
* In structural design style, the circuit description or design individual gates and other components are instantiated and connected to each other using nets.
* Verilog has several built in gate types, the names of these gates are reserved words, some of these are

**Syntax of Verilog instance ststements:**

Component\_name

instance-identifier(expression……………expresssion);

Component\_name instance-identifier (**.**port-name(expression),

**: :**

**.**port-name(expression));

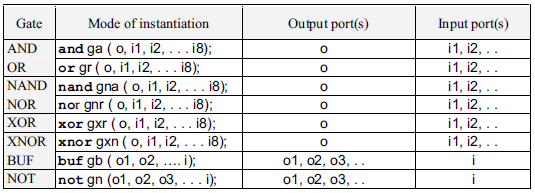


Table 5.1 Basic gate primitives in Verilog with details

**5.7 Behavioral Modeling**

Behavioral level modeling constitutes design description at an abstract level. One can visualize the circuit in terms of its key modular functions and their behavior. The constructs available in behavioral modeling aim at the system level description. Here direct description of the design is not a primary consideration in the Verilog standard. Rather, flexibility and versatility in describing the design are in focus [IEEE].

Verilog provides designers the ability to describe design functionality in an algorithmic manner. In other words, the designer describes the behavior of the circuit. Thus, behavioral modeling represents the circuit at a very high level of abstraction. Design at this level resembles C programming more than it resembles digital circuit design. Behavioral Verilog constructs are similar to C language constructs in many ways.

**Structured Procedures:**

There are two structured procedure statements in Verilog: always and initial .These statements are the two most basic statements in behavioral modeling. All other behavioral statements can appear only inside these structured procedure statements.

Verilog is a concurrent programming language unlike the C programming language, which is sequential in nature. Activity flows in Verilog run in parallel rather than in sequence. Each always and initial statement represents a, separate activity flow in Verilog. Each activity flow starts at simulation time 0.The statements always and initial cannot be nested. The fundamental difference between the two statements is explained in the following sections.

**Always:**

* The key element of Verilog behavioral design is the **always** block the always block contains one or more “procedural statements”.
* Another type of procedural statement is a “begin-end” block. But the ALWAYS block is used in all because of its simplicity, that is why we call it an always block.
* Procedural statement in an always block executes sequentially .The always block executes concurrently with other concurrent statement in the same module.

Syntax:

1).Always @(signal-name …………signal-name)

Procedural statement

2). Always procedural statements

* In the first form of always block, the @ sign and parenthesized list of signal names called “sensitivity list “.
* A verilog concurrent statement such as always block is either executing or suspend
* A concurrent statement initially is in suspending state, when any signal value changes its value, it resumes execution starting its first procedural statement and continuing until the end.
* A properly written concurrent statement will suspend after one or more executions. However it is possible to write a statement that never suspends (e.g.: assign X=~X), since X changes for every pass, the statement will execute forever in zero simulation time(which is not useful).
* As shown in the second part of syntax, the sensitivity list in always block is optional .an always block without a sensitivity list starts running at zero simulation time and keeps looping forever.
* There are different types of procedural statement that can appear with in an always block. They are blocking-assignment statement, non blocking-assignment statement, begin-end blocks, if, case, while and repeat.

**IF AND IF-ELSE BLOCK:**

The IF construct checks a specific condition and decides execution based on the result. Figure shows the structure of a segment of a module with an IF statement. After execution of assignment1, the condition specified is checked. If it is satisfied, assignment2 is executed; if not, it is incrementped. In either case the execution continues through assignment3, assignment4, etc. Execution of assignment2 alone is dependent on the condition. The rest of the sequence remains. The flowchart equivalent of the execution is shown in Figure.

**Syntax:**

If (condition)

. . .

assignment1;

if (condition) assignment2;

assignment3;

assignment4;

. . .

After the execution of assignment1, if the condition is satisfied, alternative1 is followed and assignment2 and assignment3 are executed. Assignment4 and assignment 5 are incrementped and execution proceeds with assignment6.

* If the condition is not satisfied, assignment2 and assignment3 are incrementped and assignment4 and assignment5 are executed. Then execution continues with assignment6.

**For Loops**

Similar to for loops in C/C++, they are used to repeatedly execute a statement or block of statements. If the loop contains only one statement, the *begin ... end* statements may be omitted.

**Syntax:**

for (count = value1;

count </<=/>/>= value2;

count = count +/- step)

begin

... statements ...

End

**While Loops:**

The while loop repeatedly executes a statement or block of statements until the expression in the while statement evaluates to false. To avoid combinational feedback during synthesis, a while loop must be broken with an @(posedge/negedge clock) statement . For simulation a delay inside the loop will suffice. If the loop contains only one statement, the begin ... end statements may be omitted.

**Syntax:**

while (expression)

begin

... statements ...

End

**CASE:**

The case statement allows a multipath branch based on comparing the expression with a list of case choices. Statements in the default block executes when none of the case choice comparisons are true. With no default, if no comparisons are true, synthesizers will generate unwanted latches. Good practice says to make a habit of puting in a default whether you need it or not.

If the defaults are dont cares, define them as ‘x’ and the logic minimizer will treat them as don’t cares and dsave area. Case choices may be a simple constant, expression, or a comma-separated list of same.

**Syntax**

case (expression)

case\_choice1:

begin

... statements ...

end

case\_choice2:

begin

... statements ...

end

... more case choices blocks ...

default:

begin

... statements ...

end

endcase

**casex:**

In casex(a) the case choices constant “a” may contain z, x or ? which are used as don’t cares for comparison. With case the corresponding simulation variable would have to match a tri-state, unknown, or either signal. In short, case uses x to compare with an unknown signal. Casex uses x as a don’t care which can be used to minimize logic.

**Casez:**

Casez is the same as casex except only ? and z (not x) are used in the case choice constants as don’t cares. Casez is favored over casex since in simulation, an inadvertent x signal, will not be matched by a 0 or 1 in the case choice.

**FOREVER LOOPS**

The forever statement executes an infinite loop of a statement or block of statements. To avoid combinational feedback during synthesis, a forever loop must be broken with an@(posedge/negedge clock) statement. For simulation a delay inside the loop will suffice. If the loop contains only one statement, the *begin ... end* statements may be omitted.

**Syntax**

forever

begin

... statements ...

End

sExample

forever begin

@(posedge clk); // or use a= #9 a+1;

a = a + 1;

end

**REPEAT:**

The repeat statement executes a statement or blocks of statements a fixed number of times. repeat CONSTRUCT The repeat construct is used to repeat a specified block a specified number of times. The quantity a can be a number or an expression evaluated to a number. As soon as the repeat statement is encountered, a is evaluated. The following block is executed “a” times. If “a” evaluates to 0 or x or z, the block is not executed.

**Syntax:**

repeat (number\_of\_times)

begin

... statements ...

End

# Chapter 6

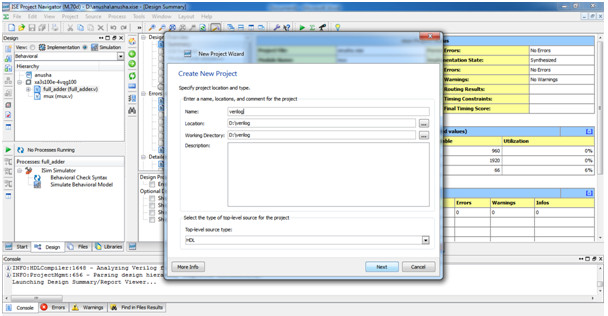
SOFTWARE USED

Xilinx

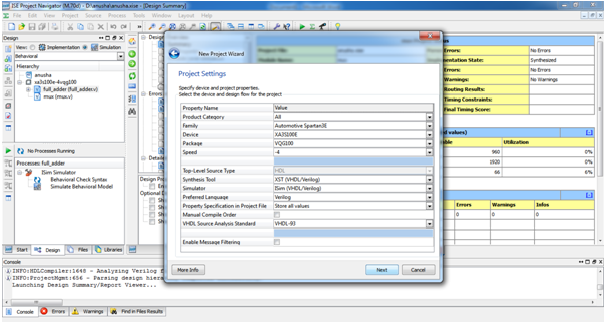
## Xilinx software is used by the VHDL/VERILOG designers for performing Synthesis operation. Any simulated code can be synthesized and configured on FPGA. Synthesis is the transformation of HDL code into gate level net list. It is an integral part of current design flows.

### 6.1Algorithm

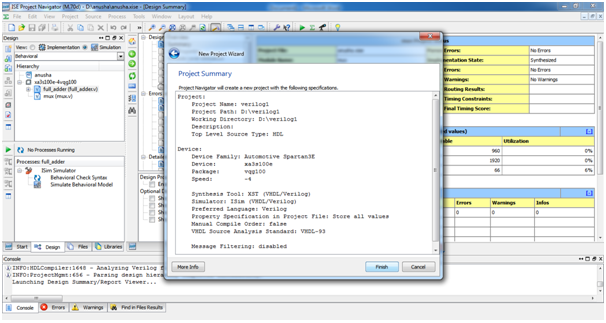
### Start the ISE Software by clicking the XILINX ISE icon.Create a New Project and find the following properties displayed.If the design needs large number of LUTs there is a possibility to change the family ,device and package changes.



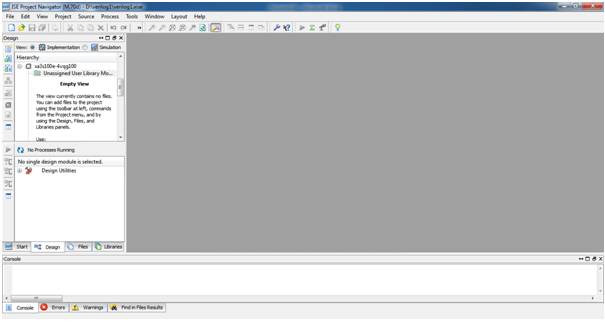
### Fig .1.Create new folder for design



### Fig .2. Set family and device before design a project



### Fig .3. finishing new folder ,Set family and device before design a project



### Fig .4. Ready to design a project

Create a HDL Source formatting all inputs, outputs and buffers if required. which provides a window to write the HDL code, to be synthesized.

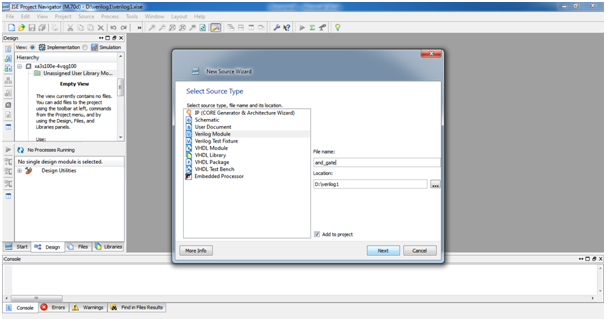


Fig 5. create module name(.v files names) for design

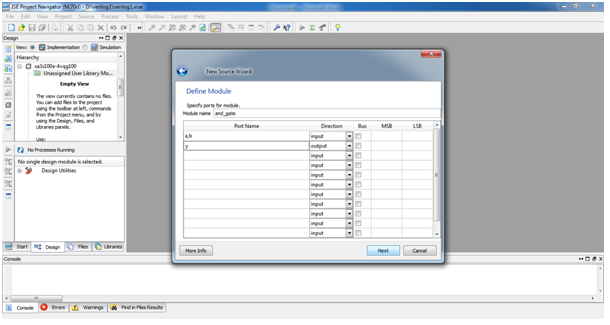


Fig.6.Declaration of input and output ports with their bit lengths

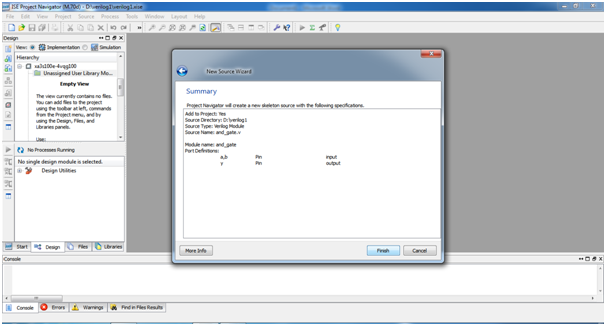


Fig .7. The schematic was created by its ports

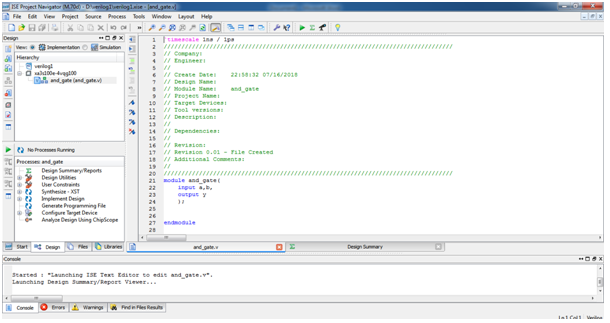


Fig .8: Ready to write the code for design

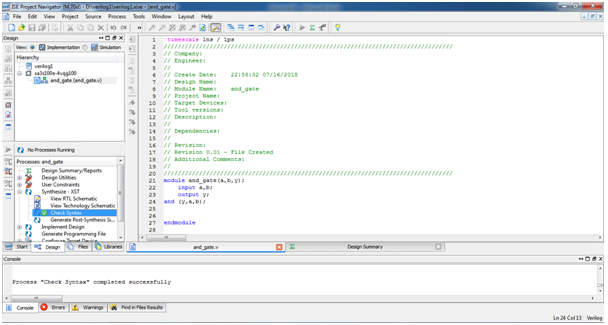


Fig .9: Check syntax for the Design

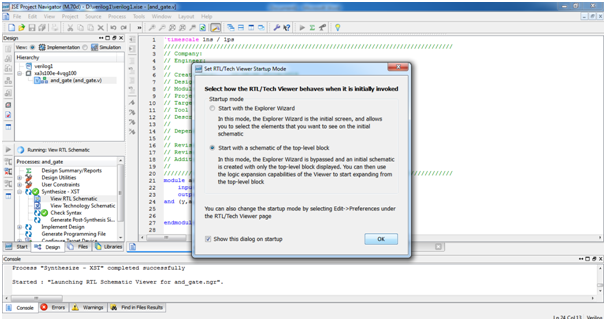


Fig 10: check for RTL schematic view

For RTL (register transfer logic )view ,which is also known as designer view because it is look like what the designer thinks in his mind.

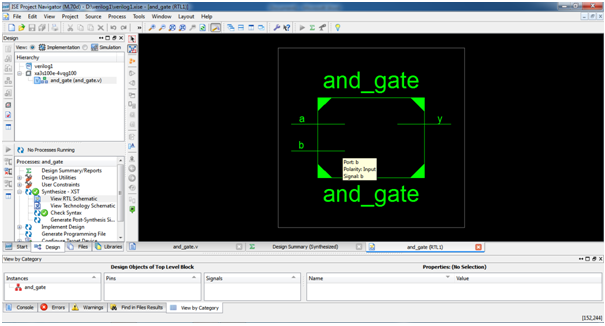


Fig .11.RTL schematic view of design (and gate)

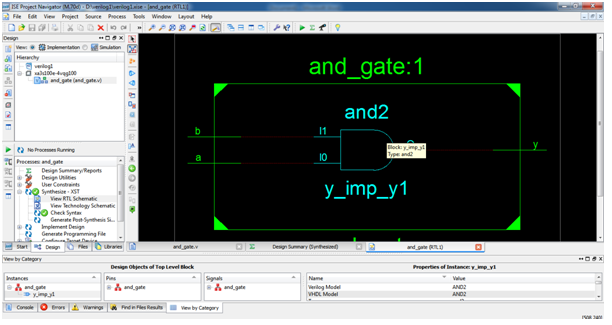


Fig 12:Internal structure of RTL schematic view of design(andgate)

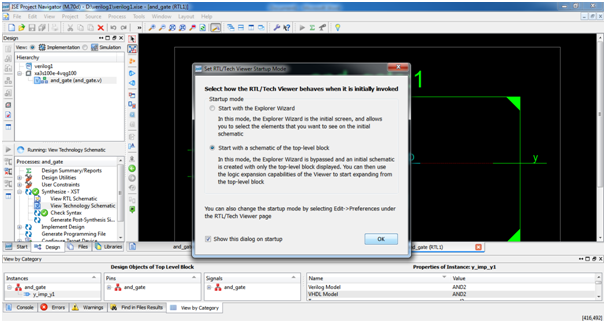


Fig13. Check for view technology schematic view of the project

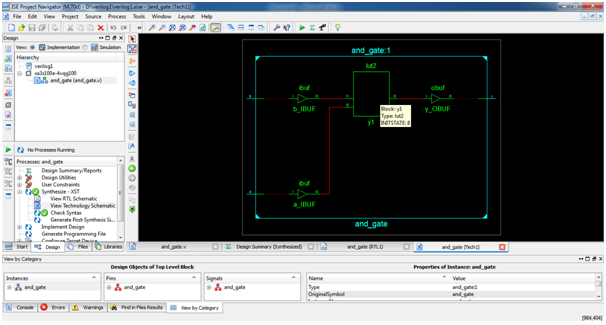


Fig14: Internal structure of view technology schematic view

View technology schematic of design (and gate) ,here LUTs are displayed ,luts are considered as area of the design.

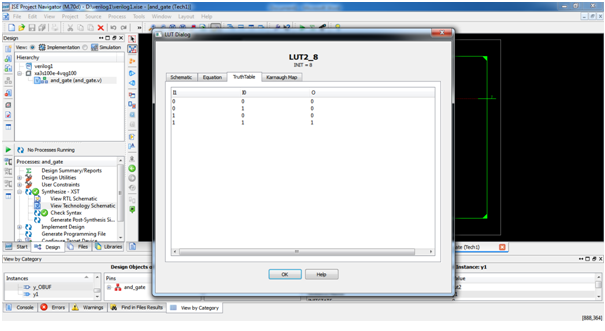


Fig 15 : The truth table ,schematic of design ,Boolean equation and k-map of design.

In Xilinx tool there is a availability to get truth table ,schematic of design ,Boolean equation and k-map

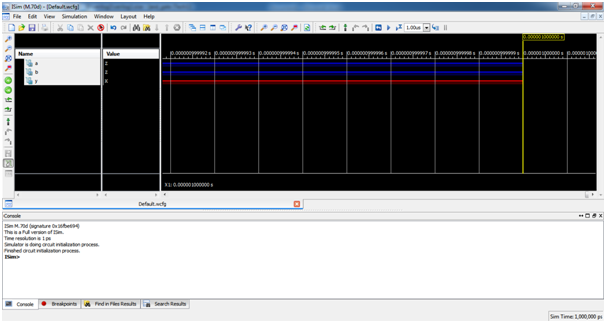


Fig16: Simulation of design to verifying the logics of design.

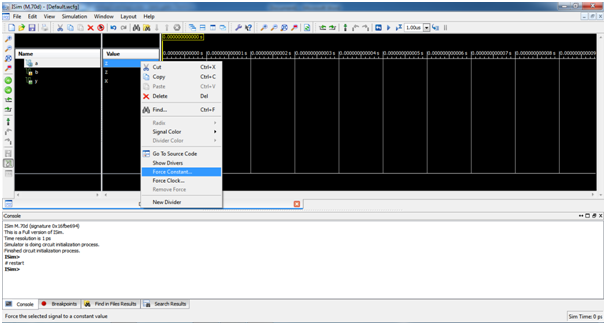


Fig 17: Apply inputs through force constant or force clock for input signals

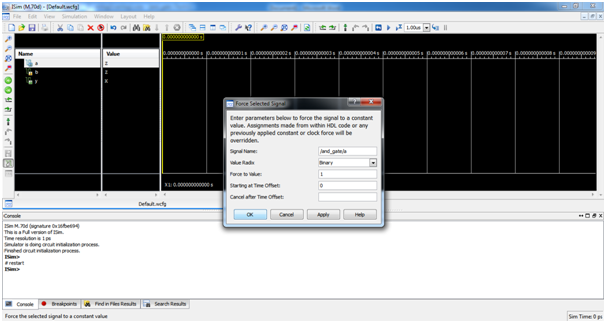


Fig.18 : Apply force to value

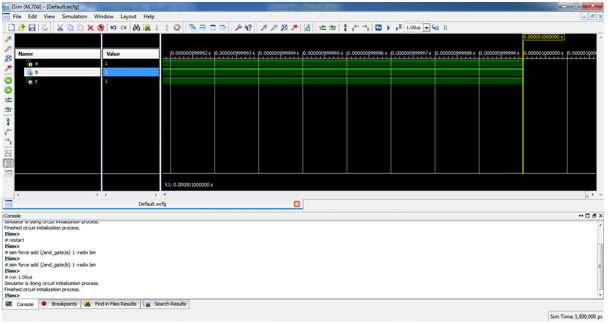


Fig .19: Run the design after applying inputs

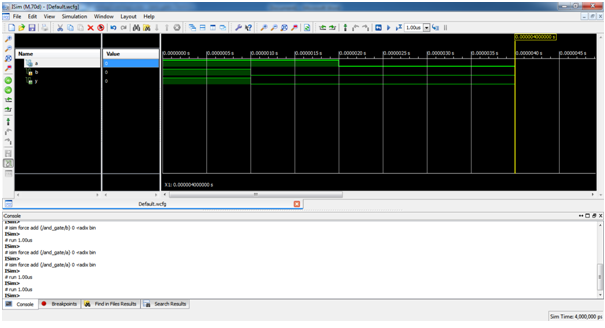


Fig 20: Show all values(zoom to full view) for the design

**Advantages, Disadvantages And Applications**

**Advantages**

The proposed design turbo encoder with parallel computation using less number of clock pulses to getting out put .Infact50% of the power should be clock pulse only,so the parallel computation uses less power compared to serial computation because serial computation method uses more clock pulses compared to parallel computation and area also reduced by using carry increment adder for counter design in turbo encoder module .

**Disadvantages**

The Circuit complexity of proposed design turbo encoder is little bit more.

**Applications**

Turbo encoders are used in minimum and less error rate communication mediums. Turbo encoder has also opened up a new way of thinking in the construction of communication algorithms. The Turbo encoders are used in low latency designs