CHIP DESIGN FOR TURBO ENCODER MODULE FOR IN-VEHICLE SYSTEM

A Mini Project Report submitted to JNTU Hyderabad in partial fulfillment of the requirements for the award of the degree

BACHELOR OF TECHNOLOGY

In

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

SRIVANI GEDDADA	**21S11A0496**
SUDHEER KUMAR TOKALA	**21S11A0497**
SAI KUMAR REDDY MANDAPATI	**21S11A0485**
MAHESH NOMULA	**21S11A0470**

Under the Guidance of

Mrs.D.SAROJA BAI

B. Tech, M. Tech.

Assistant Professor of ECE

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

MALLA REDDY INSTITUTE OF TECHNOLOGY & SCIENCE

(Approved by AICTE New Delhi and Affiliated to JNTUH)

(Accredited by NBA& NAAC with "A" Grade)

An ISO 9001: 2015 Certified Institution

Maisammaguda, Medchal (M), Hyderabad-500100, T. S.

DECEMBER2024

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
MALLA REDDY INSTITUTE OF TECHNOLOGY & SCIENCE
(Approved by AICTE New Delhi and Affiliated to JNTUH)
(Accredited by NBA & NAAC with "A" Grade)
An ISO 9001: 2015 Certified Institution
Maisammaguda, Medchal (M), Hyderabad-500100, T.S.

CERTIFICATE

This is to certify that the Mini project entitled **CHIP DESIGN FOR TURBO ENCODER MODULE FOR IN-VEHICLE SYSTEM** has been submitted by **SRIVANI GEDDADA (21S11A0496), TOKALA SUDHEER KUMAR (21S11A0497), SAI KUMAR REDDY MANDAPATI (21S11A0485) and MAHESH NOMULA (21S11A0470)** in partial fulfillment of the requirements for the award of **BACHELOR OF TECHNOLOGY** in **ELECTRONICS AND COMMUNICATION ENGINEERING**. This record of bonafide work carried out by them under my guidance and supervision. **The result embodied in this Mini project report has not been submitted to any other University or Institute for the award of any degree.**

***Mrs.D.Saroja* Bai**
Assistant Professor of ECE
Project Guide

Mrs. G. Subhashini
Head of the Department

External Examiner

ACKNOWLEDGEMENT

The Mini Project work carried out by our team in the Department of Electronics and Communication Engineering, Malla Reddy Institute of Technology and Science, Hyderabad. ***This work is original and has not been submitted in part or full for any degree or diploma of any other university.***

We wish to acknowledge our sincere thanks to our project guide **Mrs.D.Saroja Bai**, Assistant Professor of Electronics and Communication Engineering for formulation of the problem, analysis, guidance and her continuous supervision during the course of work.

We acknowledge our sincere thanks to **Dr. Vaka Murali Mohan,** Principal and **Mrs. G.Subhashini**, Head of the Department and Coordinator, faculty members of ECE Department for their kind cooperation in making this Case study work a success.

We extend our gratitude to **Sri. Ch. Malla Reddy**, Founder Chairman MRGI and **Sri. Ch. Mahender Reddy,** Secretary MRGI, **Dr.Ch. Bhadra Reddy**, President MRGI, **Sri. Ch. Shalini Reddy**, Director MRGI, **Sri. P. Praveen Reddy**, Director MRGI, for their kind cooperation in providing the infrastructure for completion of our Mini Project.

We acknowledge our special thanks to the entire teaching faculty and non-teaching staff members of the Electronics and Communication Engineering Department for their support in making this project work a success.

SRIVANI GEDDADA **21S11A0496**_____

SUDHEER KUMAR TOKALA **21S11A0497** _____

SAI KUMAR REDDY MANDAPATI **21S11A0485**_____

MAHESH NOMULA **21S11A0470** _____

INDEX

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, it is shown that both chip size and processing time are improved. 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.

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

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 in use by several spacecraft. JPL's LDPC codes are built from proto graphs 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 an active 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 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.

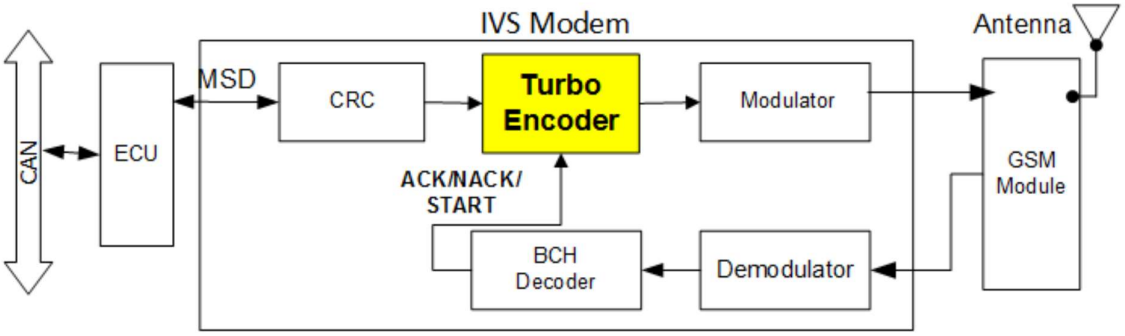


Fig. 1.1: The IVS block diagram.

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.Saleemet 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 0.13µm 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 maximized.

[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 code words with these distances was also an important factor when looking for the optimum component code and that the lower the number of possible code words 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.

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-symmetric mod- 2 interleaver was shown to improve turbo code performance when compared with its S-random counterpart.

CHAPTER 3: PROPOSED SYSTEM

A turbo encoder is a powerful error-correcting code used in digital communication systems. It's particularly popular for cellular communication and wireless networks due to its impressive performance.

o The basic idea behind turbo codes is to use two parallel convolutional encoders, which work together to create a more robust encoded signal.

o These encoders introduce redundancy into the data stream, allowing the receiver to correct errors caused by noise, interference, or other transmission issues.

2. How Does a Turbo Encoder Work?

o Imagine you have a binary input message (a sequence of 0s and 1s) that you want to transmit.

o The turbo encoder takes this input and applies a parallel concatenated encoding scheme: It feeds the input through two separate convolutional encoders.

The outputs of these encoders are interleaved and combined to form the final encoded data bit stream. Additionally, termination bits are appended at the end of the encoded stream.

o The interleaving and parallel structure of turbo codes contribute to their remarkable performance.

o Turbo codes come close to the theoretical Shannon capacity limit, which means they achieve nearly optimal error correction.

o They've been adopted in various third-generation (3G) cellular standards, including WCDMA (UMTS) and cdma2000.

o The decoding algorithm for turbo codes is based on the log-MAP (maximum a posteriori) algorithm, which iteratively refines the estimates of transmitted bits.

3. Efficient Decoding for Software-Defined Radios (SDRs)

o Researchers have proposed efficient decoding algorithms suitable for software-defined radio architectures.

o These software-based decoders allow flexibility and adaptability, making them ideal for SDR implementations.

TURBO ENCODER MODULE :

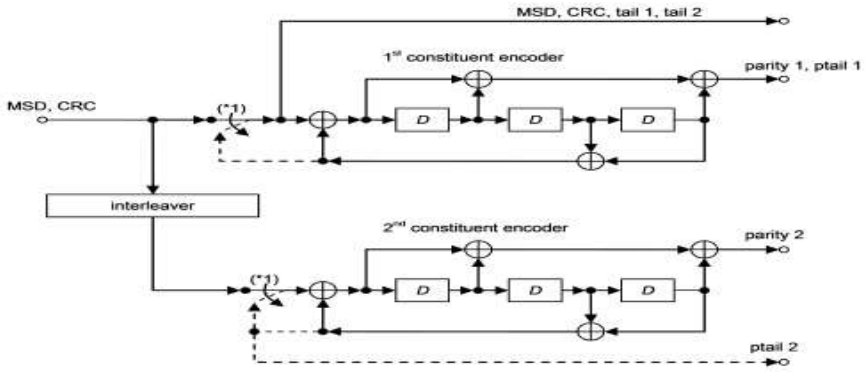


Fig. 3.1: The structure of the Turbo encoder.

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.

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 Figure3.1.

Fig. 3.2w: The output buffer of the Turbo encoder.

3.1 Interleaver

Denote the transfer function of the employed PCCC as:

$$G(D) = \left(1, \frac{g_1(D)}{g_0(D)}\right) \quad (1)$$

where

$$g_1(D) = 1 + D^2 + D^3$$

$$g_0(D) = 1 + D + D^3$$

And denote the input bits to the encoder as $x_1, x_2, ..., x_K$, the output of the interleaver as $x'_1, x'_2, ..., x'_K$, and the output bits of the first and second constituents as $z_1, z_2, ..., z_K$ and $z'_1, z'_2, ..., z'_K$, respectively; where K is the number of input bits to the Turbo encoder. The encoder output is expressed as:

$$d_K^{(0)} = x_K, d_K^{(1)} = z_K, d_K^{(2)} = z'_K$$

where $K = 0, 1, ..., K-1.$

The three code blocks of the output, $d_K^{(0)}$, $d_K^{(1)}$, and $d_K^{(2)}$, 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:

$$d_K^{(0)} = x_K, \quad d_{K+1}^{(0)} = z_{K+1}, \quad d_{K+2}^{(0)} = x'_K, \quad d_{K+3}^{(0)} = z'_{K+1}$$

$$d_K^{(1)} = z_K, \quad d_{K+1}^{(1)} = x_{K+2}, \quad d_{K+2}^{(1)} = z'_K, \quad d_{K+3}^{(1)} = x'_{K+2}$$

$$d_K^{(2)} = x_{K+1}, d_{K+1}^{(2)} = z_{K+2}, d_{K+2}^{(2)} = x'_{K+1}, d_{K+3}^{(2)} = z'_{K+2}$$

where $K = 0, 1, ..., K-1.$

The internal interleaver of the 3GPP Turbo encoder is designed to generate a systematic relationship between x_K and x'_K for any $40 \leq K \leq 5114$. 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 $x_1, x_2, ..., x_K$ are re-organized in a matrix where $y_k = x_k$ for k = 1,2,…,K and $y_k = 0$ for the elements that $R \times C > K$:

$$\begin{bmatrix} y_1 & y_2 & y_3 & \cdots & y_C \\ y_{(C+1)} & y_{(C+2)} & y_{(C+3)} & \cdots & y_{(C+C)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_{((R-1)C+1)} & y_{((R-1)C+2)} & y_{((R-1)C+3)} & \cdots & y_{(R\times C)} \end{bmatrix}$$

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,

$$s(j) = (v \times s(j-1)) \mod p \qquad (2)$$

Where $\langle s(j) \rangle$ for $j \in 1, 2, ..., p-2$ 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 $i \in 0, 1, ..., R-1$. 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 $i \in 0, 1, ..., R-1$ 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 $i \in 0, 1, ..., R-1$ such that,

K	*R*	Inter-row permutation patterns <*T*(0), *T*(1), …, *T*(*R* - 1)>
(40 ≤ *K* ≤ 159)	5	<4, 3, 2, 1, 0>
(160 ≤ *K* ≤ 200) or (481 ≤ *K* ≤ 530)	10	<9, 8, 7, 6, 5, 4, 3, 2, 1, 0>
(2281 ≤ *K* ≤ 2480) or (3161 ≤ *K* ≤ 3210)	20	<19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 16, 13, 17, 15, 3, 1, 6, 11, 8, 10>
K = any other value	20	<19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 10, 8, 13, 17, 3, 1, 16, 6, 15, 11>

Table 3.1: 3GPP inter-row permutation pattern

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

$$U_i(j) = s((j \times r_i) \mod (p-1)) \quad (3)$$

where $j = 0, 1, ..., (p-1)$ and $U_i(p-1) = 0$.
if $(C = p+1)$ then,

$$U_i(j) = s((j \times r_i) \mod (p-1)) \quad (4)$$

where $j = 0, 1, ..., (p-1)$, $U_i(p-1) = 0$, and $U_i(p) = p$.
if $(C = p-1)$ then,

$$U_i(j) = s((j \times r_i) \mod (p-1)) - 1 \quad (5)$$

where $j = 0, 1, ..., (p-1)$.

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 $y'_k = y_k$ such that:

$$\begin{bmatrix} y'_1 & y'_{(R+1)} & y'_{(2R+1)} & \cdots & y'_{((C-1)R+1)} \\ y'_2 & y'_{(R+1)} & y'_{(2R+2)} & \cdots & y'_{((C-1)R+2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y'_R & y'_{(2R)} & y'_{(3R)} & \cdots & y'_{(R\times C)} \end{bmatrix}$$

The output of the interleaver, $x'_2, x'_2, ..., x'_K$ K, is the bit sequence that are read out from the RXC matrix column by column starting with y'_1 and ending with $y'_{(R\times C)}$. 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.

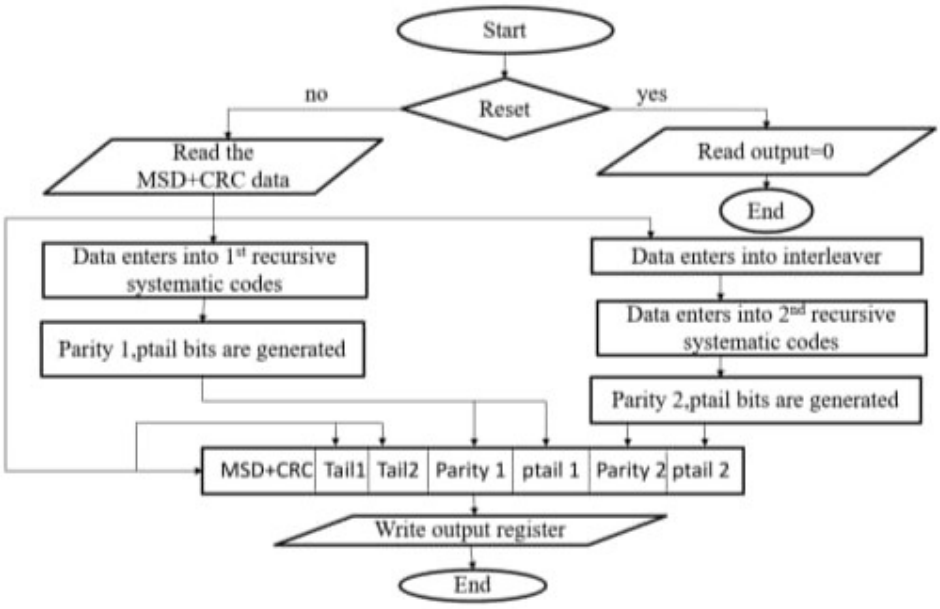


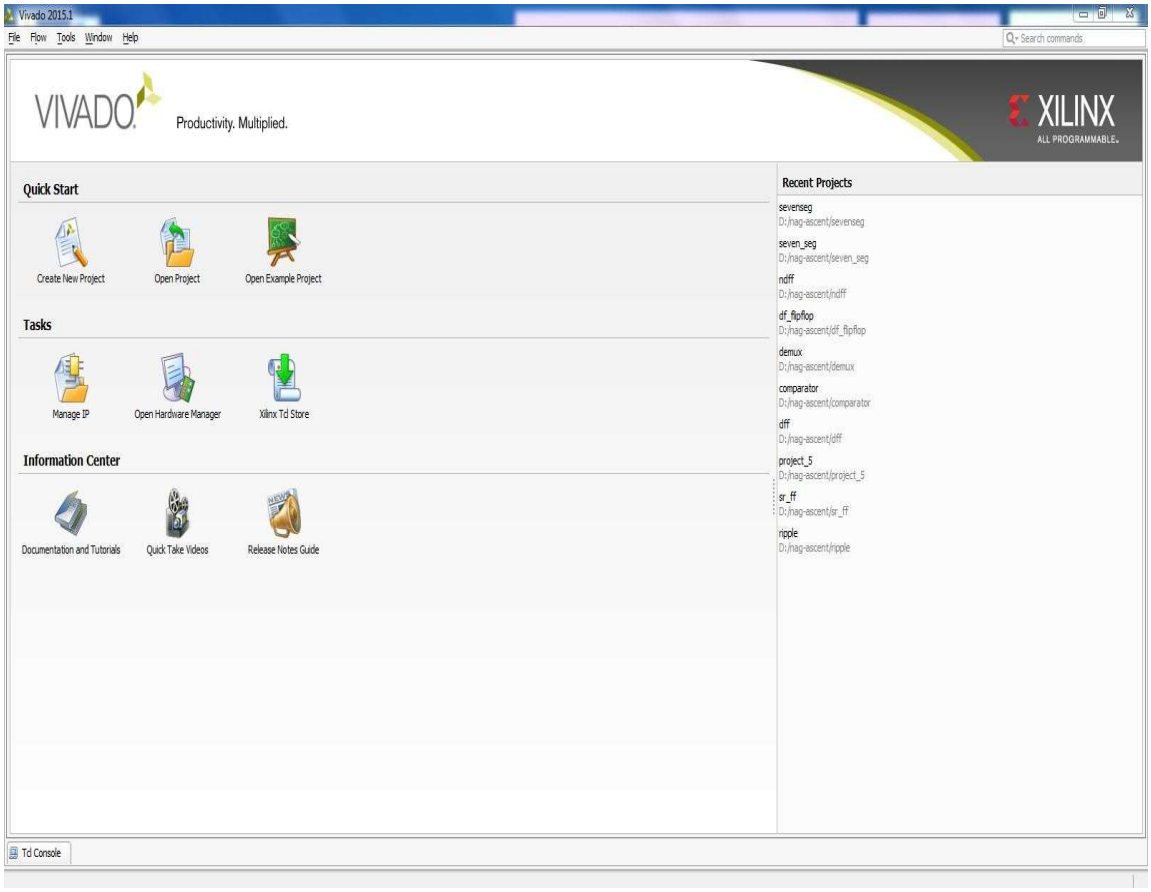
Fig. 3.3 :Turbo encoder module flowchart

CHAPTER 4: SYSTEM REQUIREMENTS

SOFTWARE USED

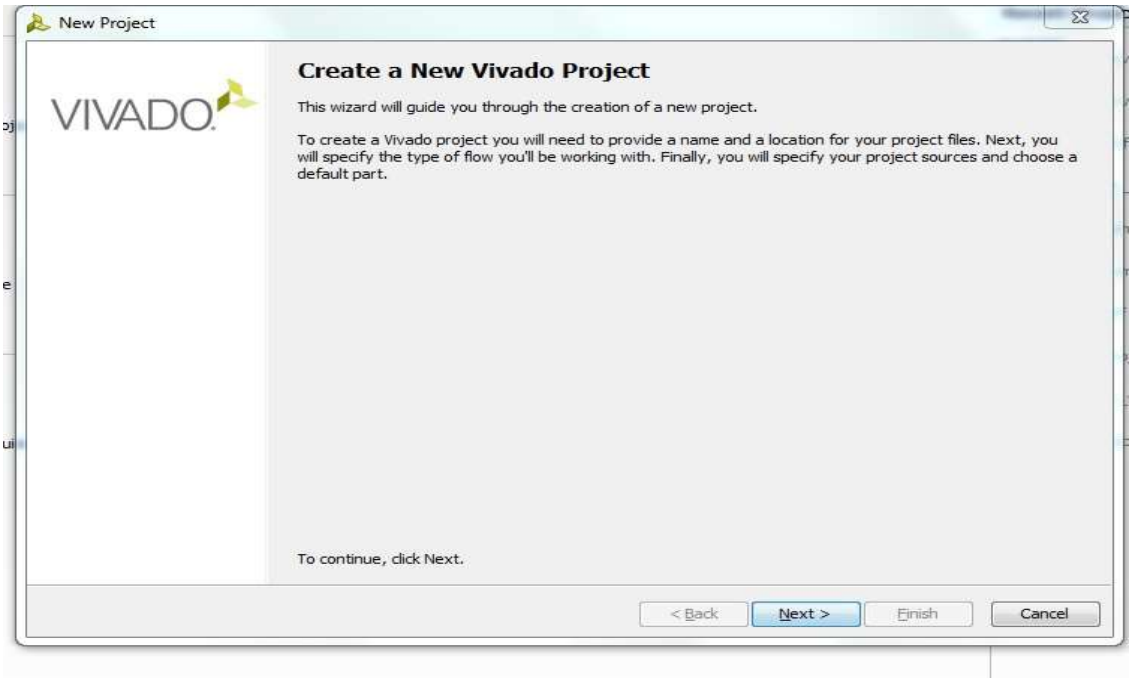
VIVADO

Xilinx Vivado 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.ClickonXilinxVivadoDesignSuiteicon.Itopensthedesignsuitewindowasshown below.



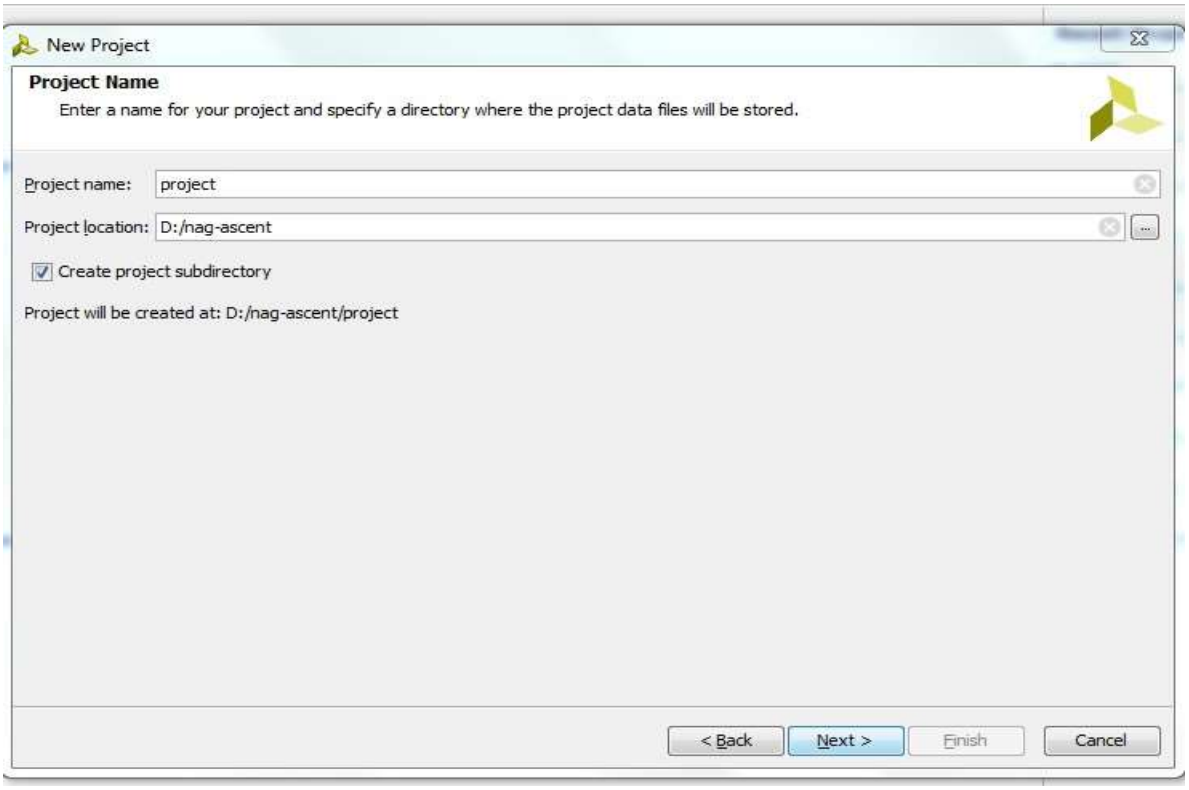
ClickOpenNewProject.Anewprojectcreationwizardwillpopup.ClickNextandthennamethe project "project_1_multiple_inputs". Select the folder path and create project subfolder.

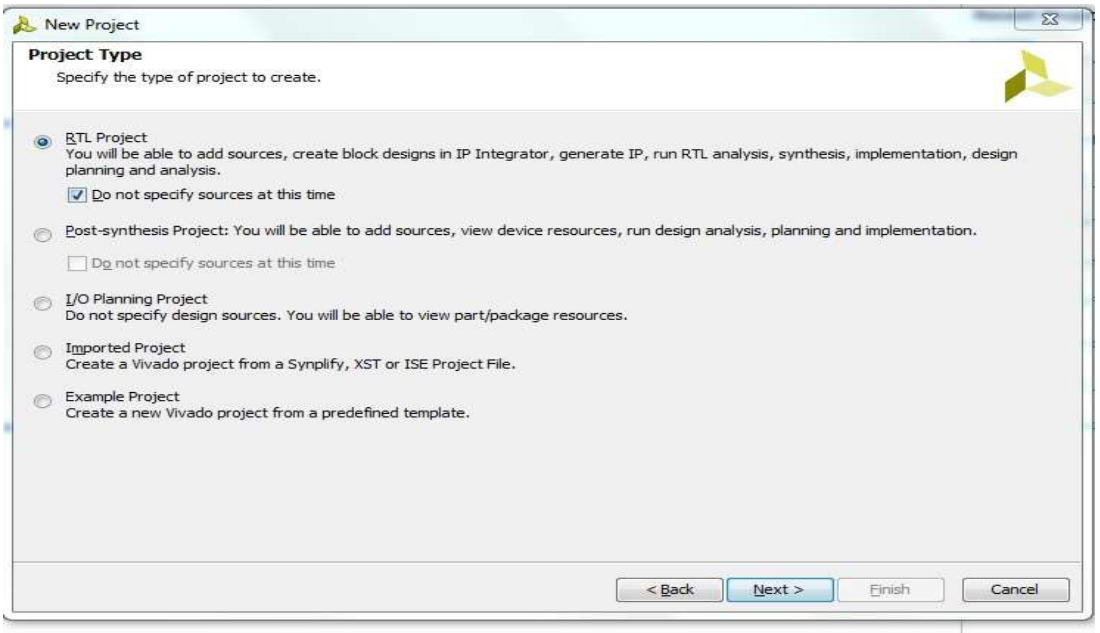
Choose RTL project

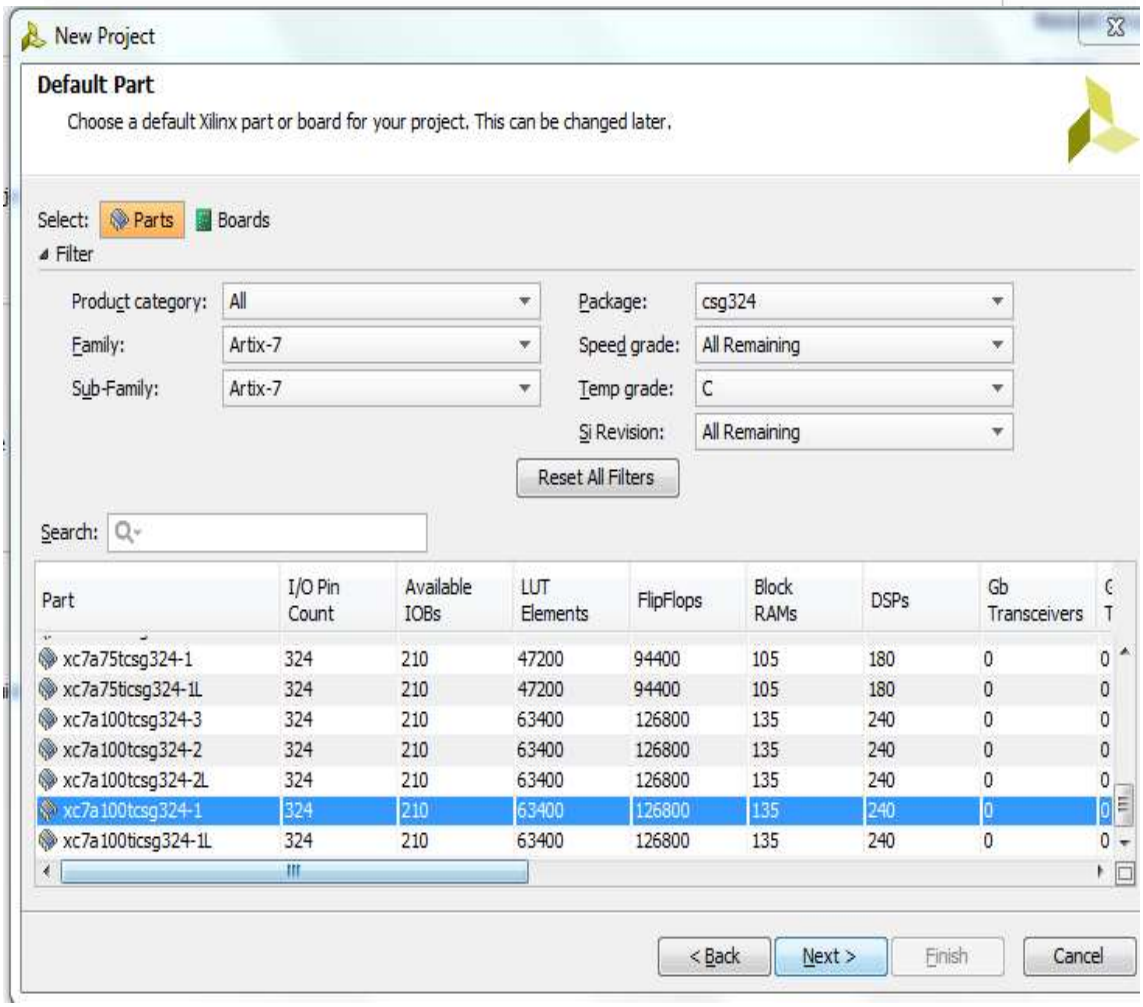


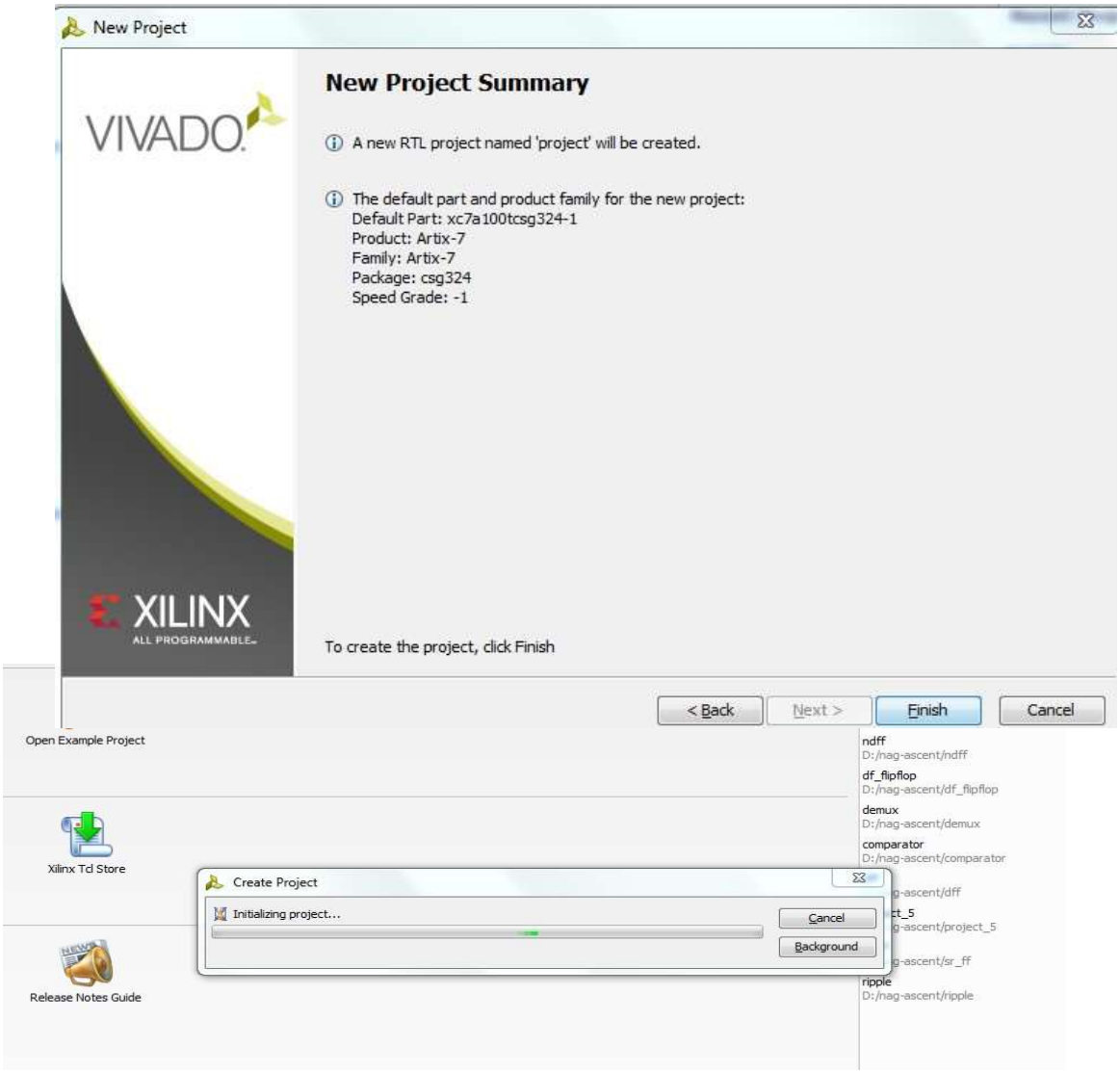
Click on New Project

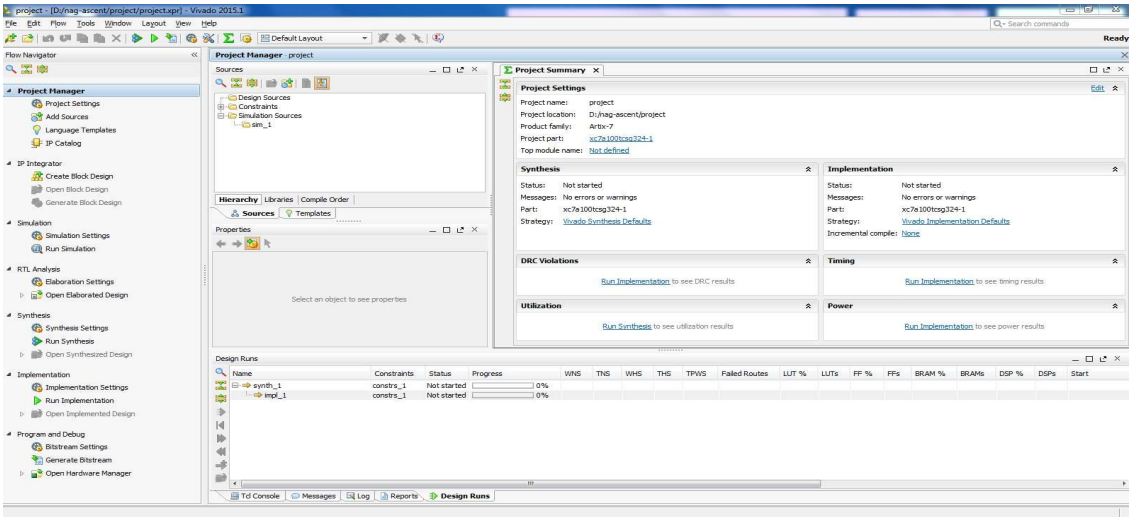
ClickonNext,thenenternameofprojectandspecifythelocationofprojectthenclick Next.

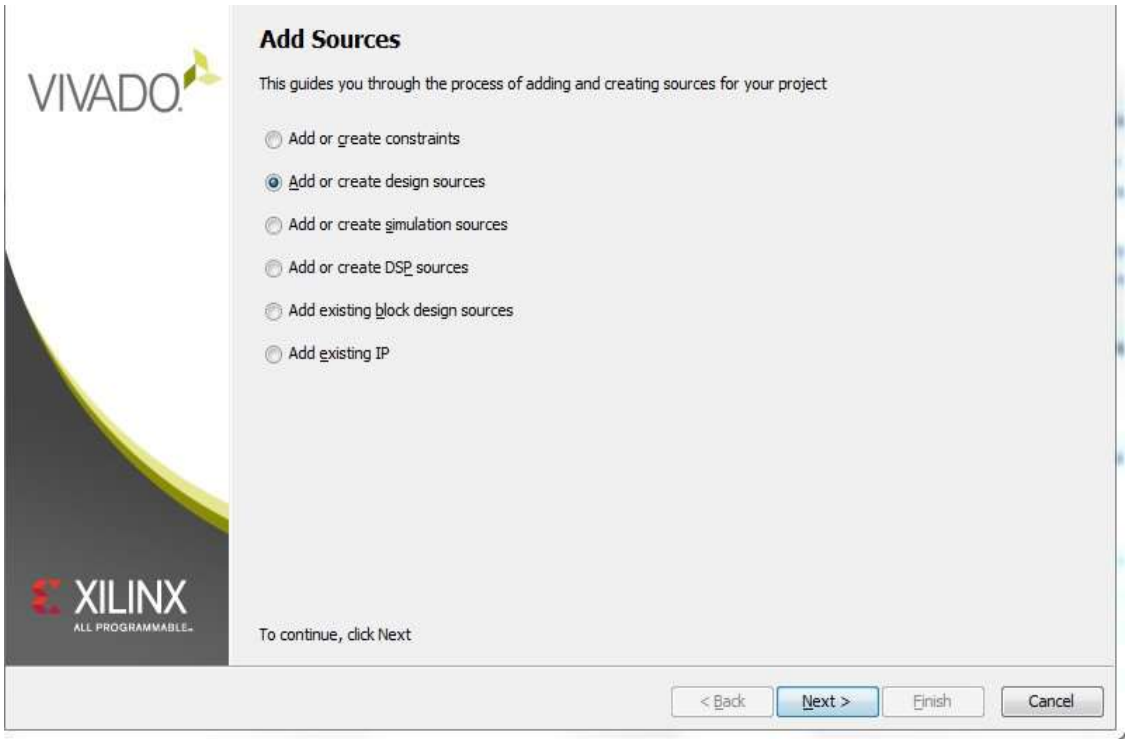


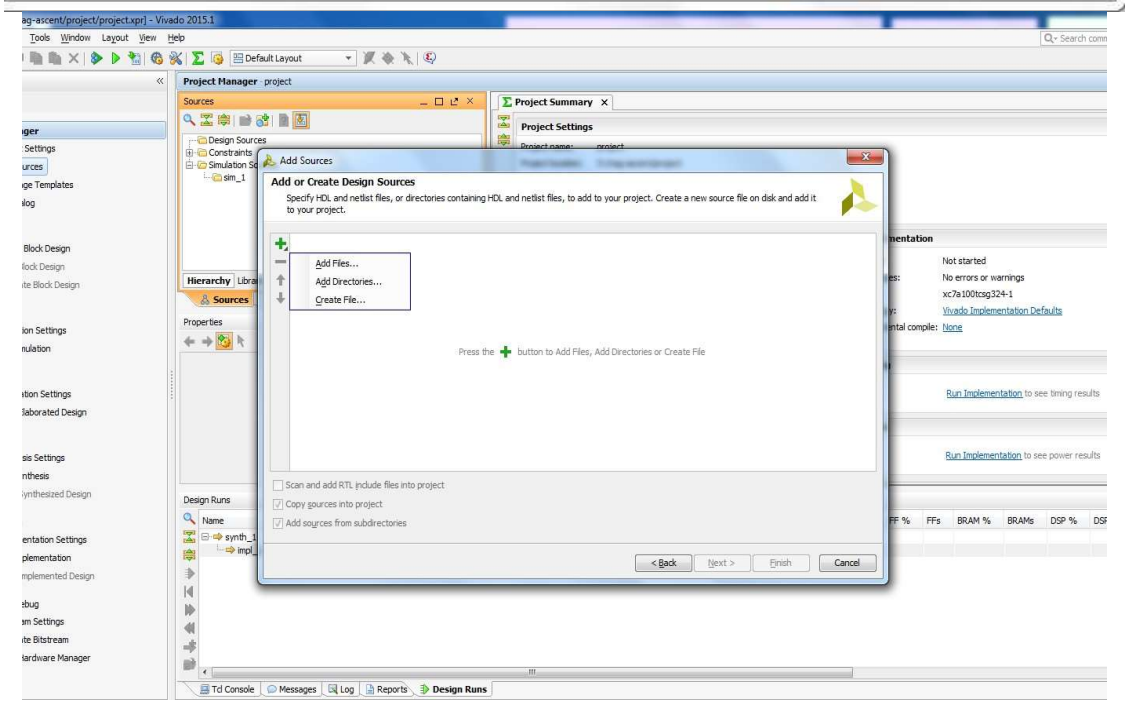

New Project
Project Type
Specify the type of project to create.
RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
Do not specify sources at this time
Post-synthesis Project: You will be able to add sources, view device resources, run design analysis, planning and implementation.
Do not specify sources at this time
I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.
Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.
Example Project
Create a new Vivado project from a predefined template.
< Back
Next >
Finish
Cancel

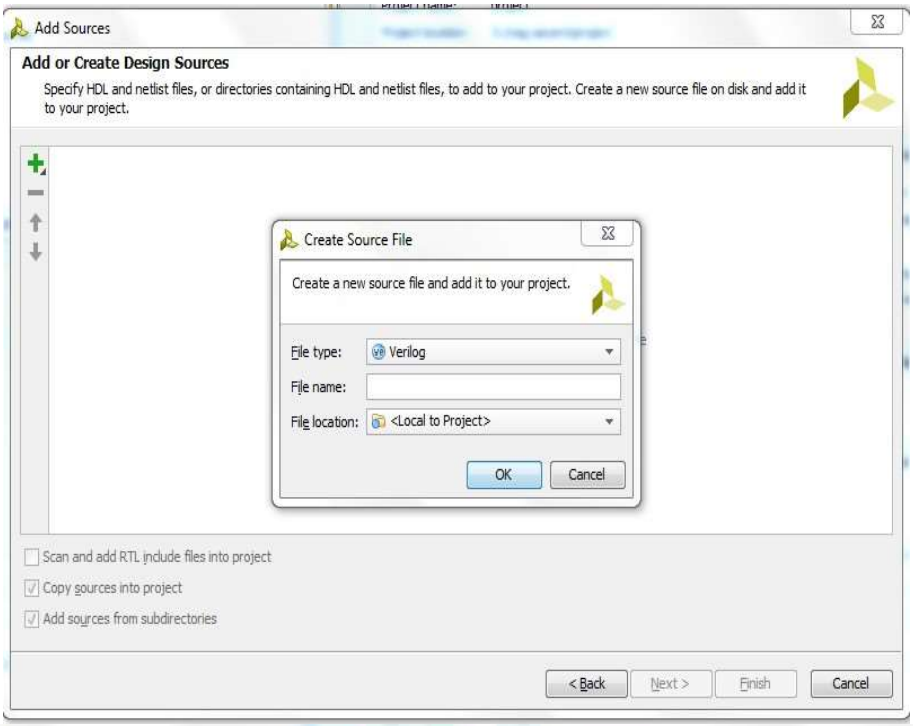

New Project
Default Part
Choose a default Xilinx part or board for your project. This can be changed later.
Select:
Parts
Boards
Filter
Product category:
All
Family:
Artix-7
Sub-Family:
Artix-7
Package:
csg324
Speed grade:
All Remaining
Temp grade:
C
Si Revision:
All Remaining
Reset All Filters
Search:
Part
I/O Pin Count
Available IOBs
LUT Elements
FlipFlops
Block RAMs
DSPs
Gb Transceivers
xc7a75tcsg324-1
xc7a75ticsg324-1L
xc7a100tcsg324-3
xc7a100tcsg324-2
xc7a100tcsg324-2L
xc7a100tcsg324-1
xc7a100ticsg324-1L
< Back
Next >
Finish
Cancel

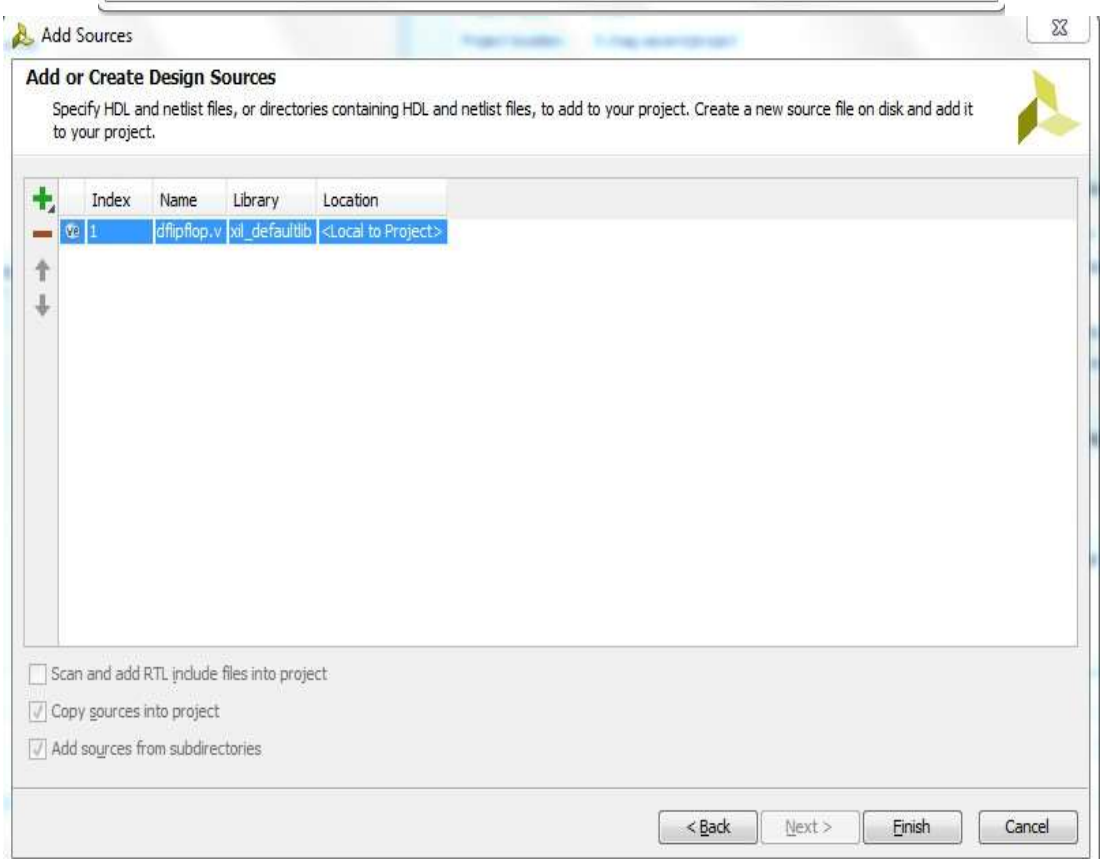

New Project
New Project Summary
A new RTL project named 'project' will be created.
The default part and product family for the new project:
Default Part: xc7a100tcsg324-1
Product: Artix-7
Family: Artix-7
Package: csg324
Speed Grade: -1
To create the project, click Finish
< Back
Next >
Finish
Cancel
Open Example Project
Xilinx Tcl Store
Release Notes Guide
Create Project
Initializing project...
Cancel
Background
ndff
D:/nag-ascent/ndff
df_flipflop
D:/nag-ascent/df_flipflop
demux
D:/nag-ascent/demux
comparator
D:/nag-ascent/comparator
ripple
D:/nag-ascent/ripple

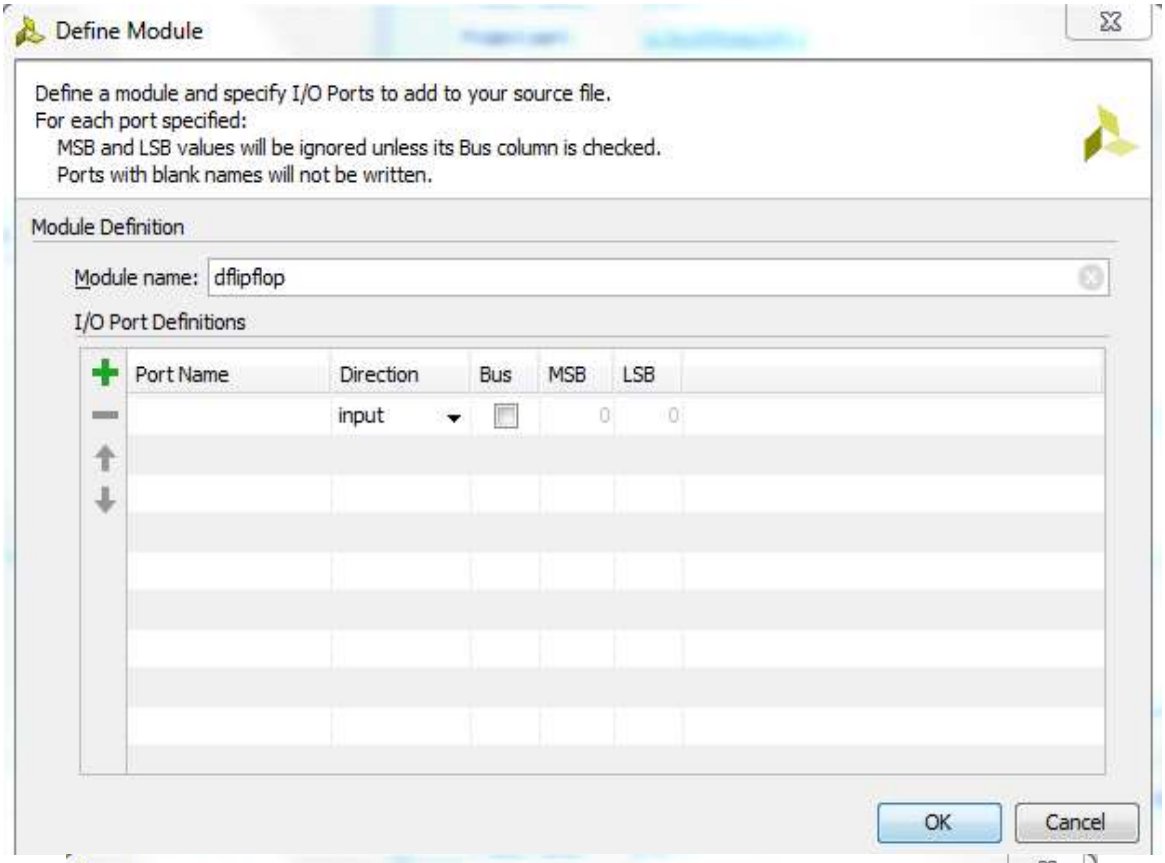


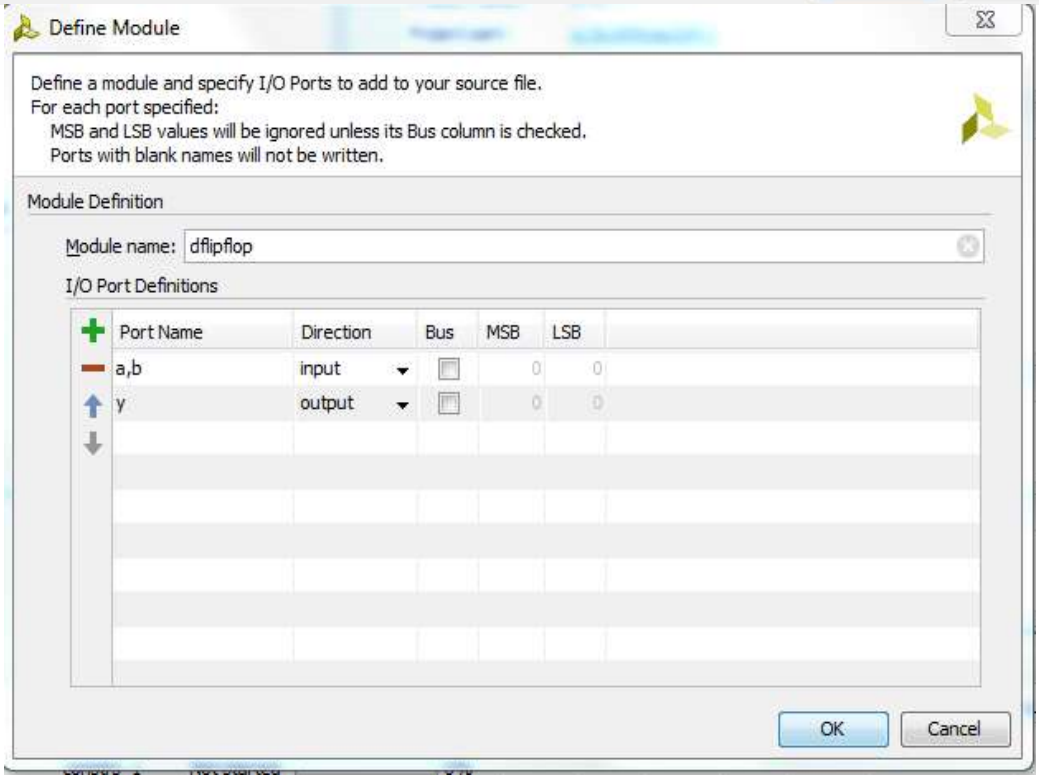

VIVADO
Add Sources
This guides you through the process of adding and creating sources for your project
Add or create constraints
Add or create design sources
Add or create simulation sources
Add or create DSP sources
Add existing block design sources
Add existing IP
XILINX
ALL PROGRAMMABLE.
To continue, click Next
< Back
Next >
Finish
Cancel

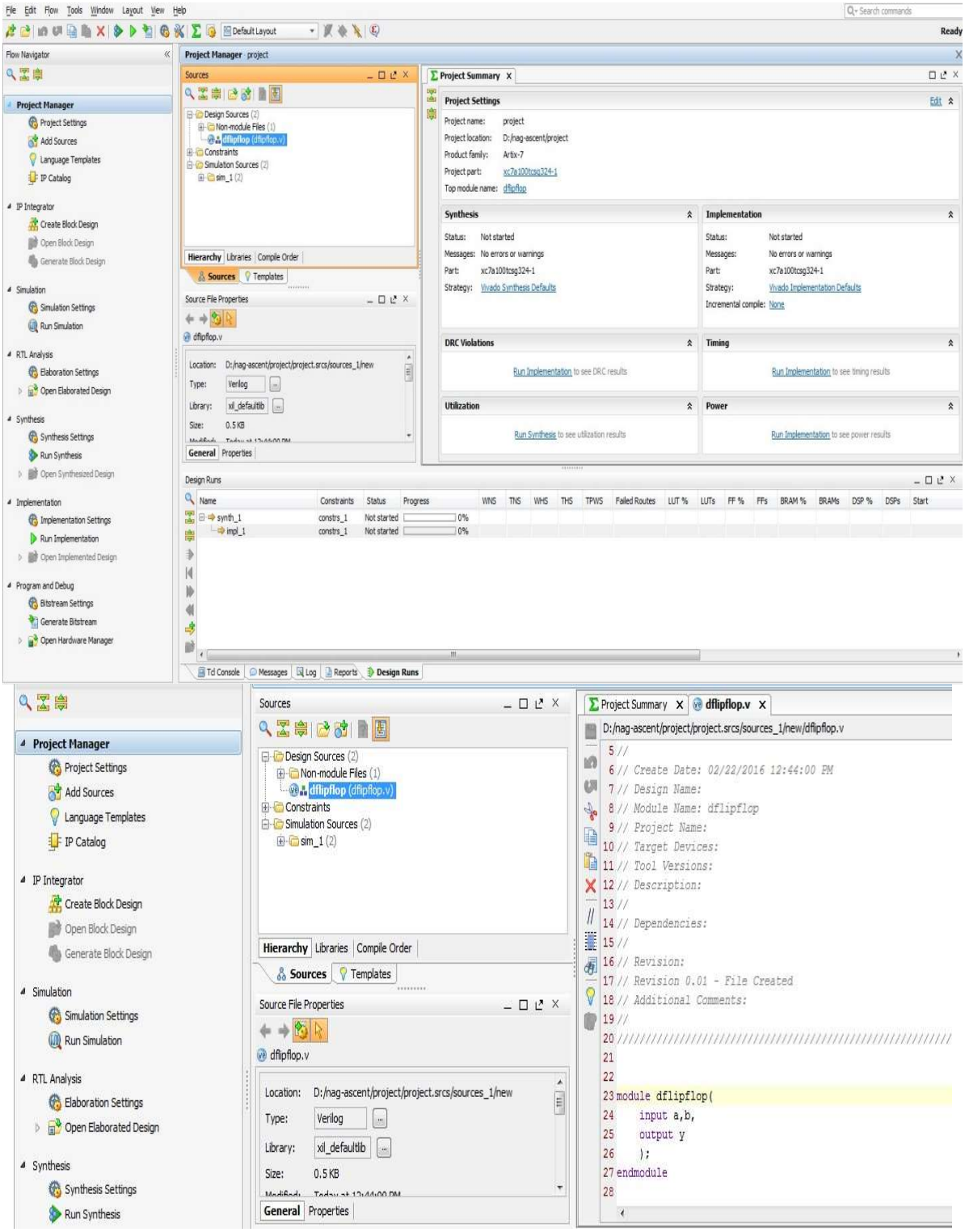

Add Sources
Add or Create Design Sources
Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.
Add Files...
Add Directories...
Create File...
Press the button to Add Files, Add Directories or Create File
Scan and add RTL include files into project
Copy sources into project
Add sources from subdirectories
< Back
Next >
Finish
Cancel

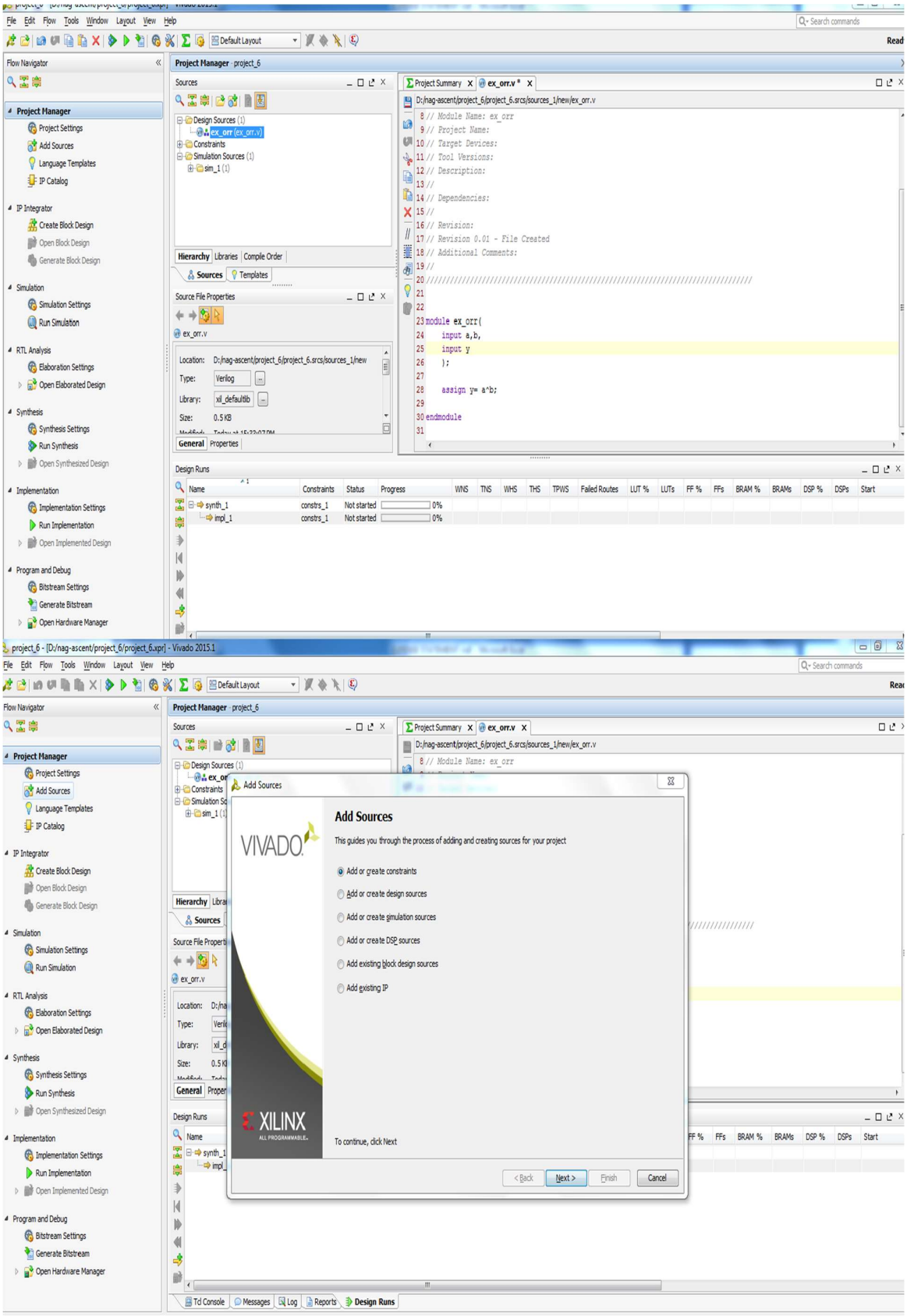

Add Sources
Add or Create Design Sources
Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.
Create Source File
Create a new source file and add it to your project.
File type:
Verilog
File name:
File location:
<Local to Project>
OK
Cancel
Scan and add RTL include files into project
Copy sources into project
Add sources from subdirectories
< Back
Next >
Finish
Cancel

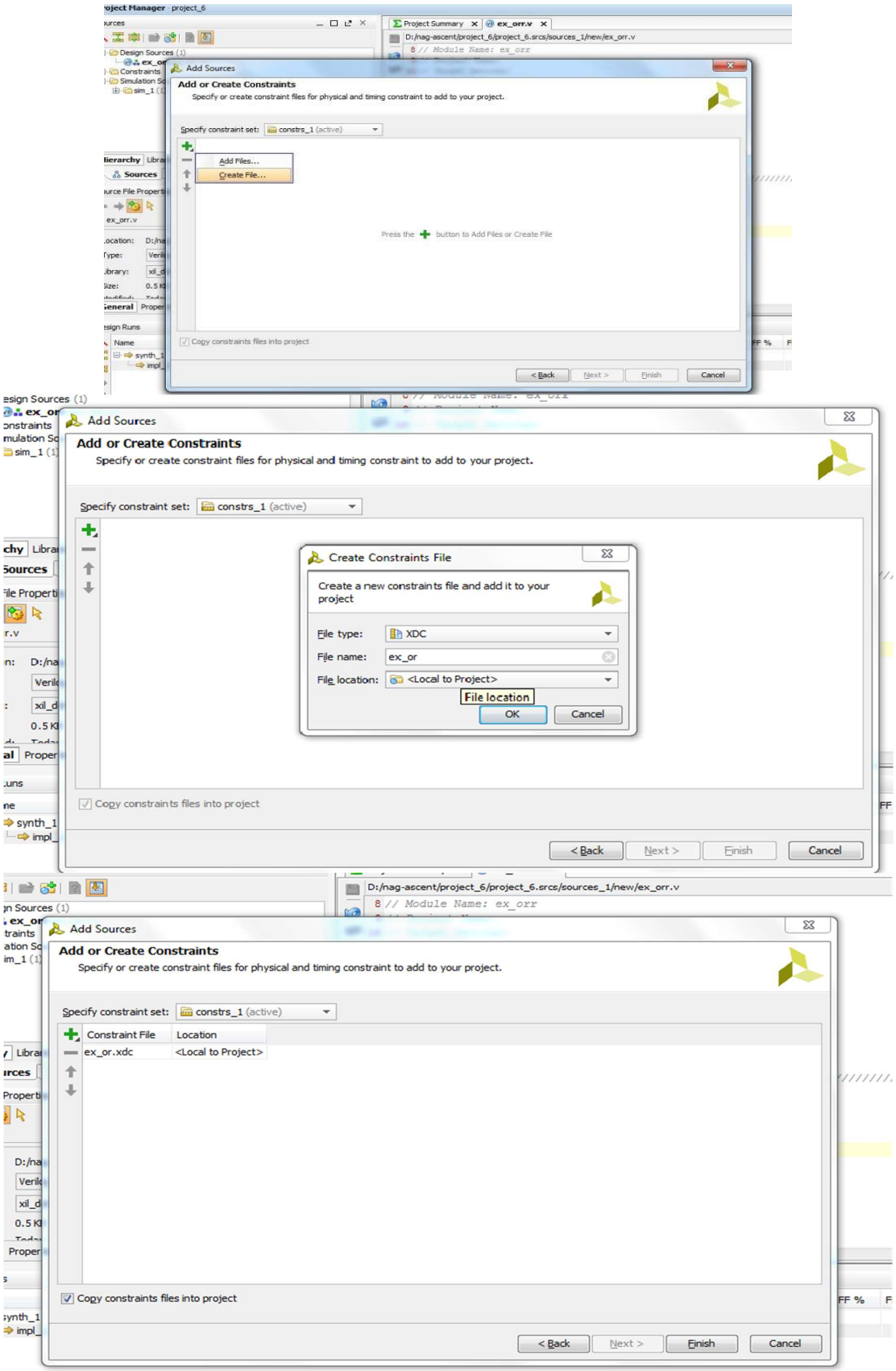

Add Sources
Add or Create Design Sources
Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.
Index
Name
Library
Location
1
dflipflop.v
xil_defaultlib
<Local to Project>
Scan and add RTL include files into project
Copy sources into project
Add sources from subdirectories
< Back
Next >
Finish
Cancel

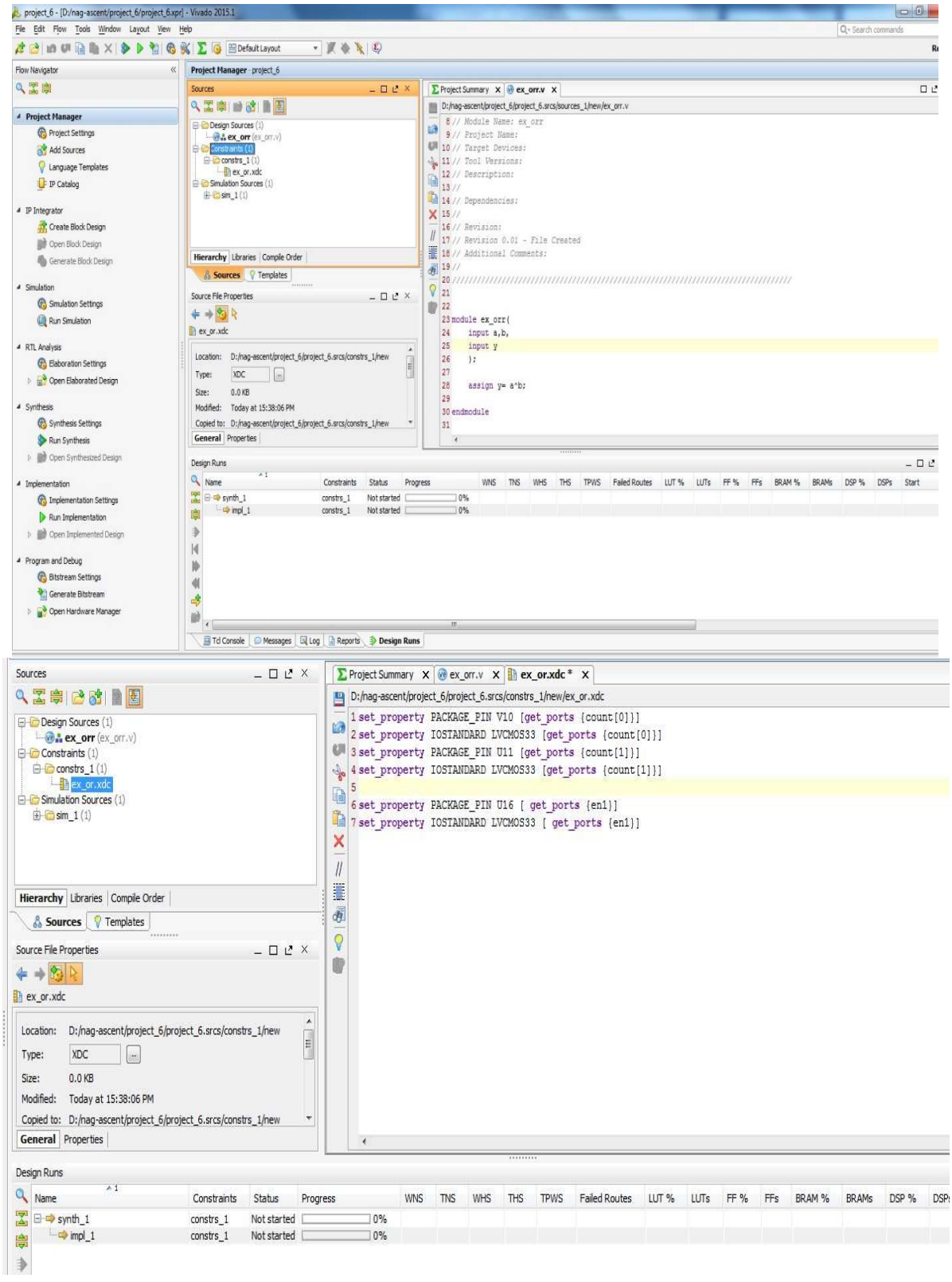

Define Module
Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.
Module Definition
Module name: dflipflop
I/O Port Definitions
Port Name
Direction
Bus
MSB
LSB
input
0
0
OK
Cancel

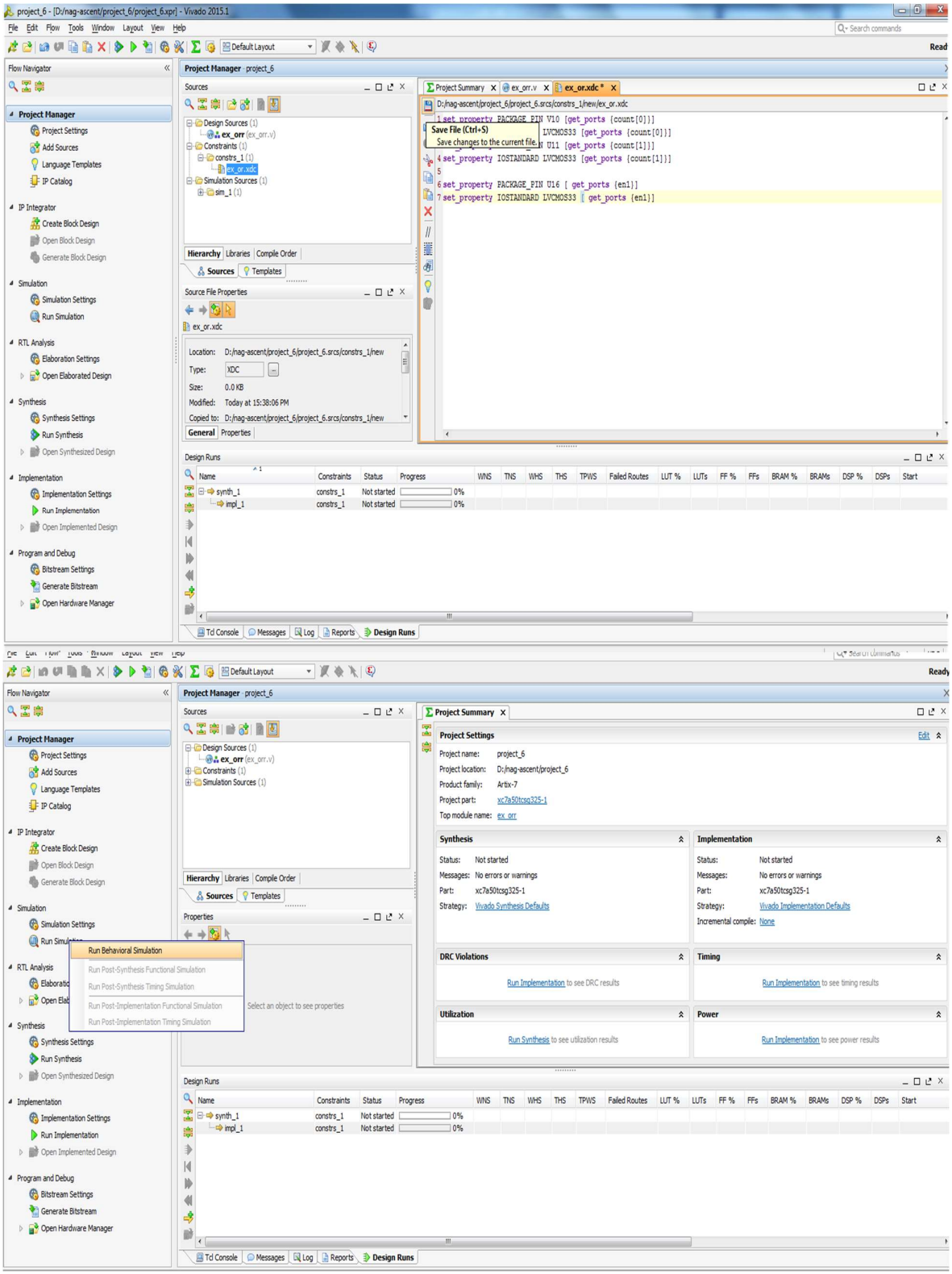

Define Module
Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.
Module Definition
Module name: dflipflop
I/O Port Definitions
Port Name
Direction
Bus
MSB
LSB
a,b
input
y
output
OK
Cancel

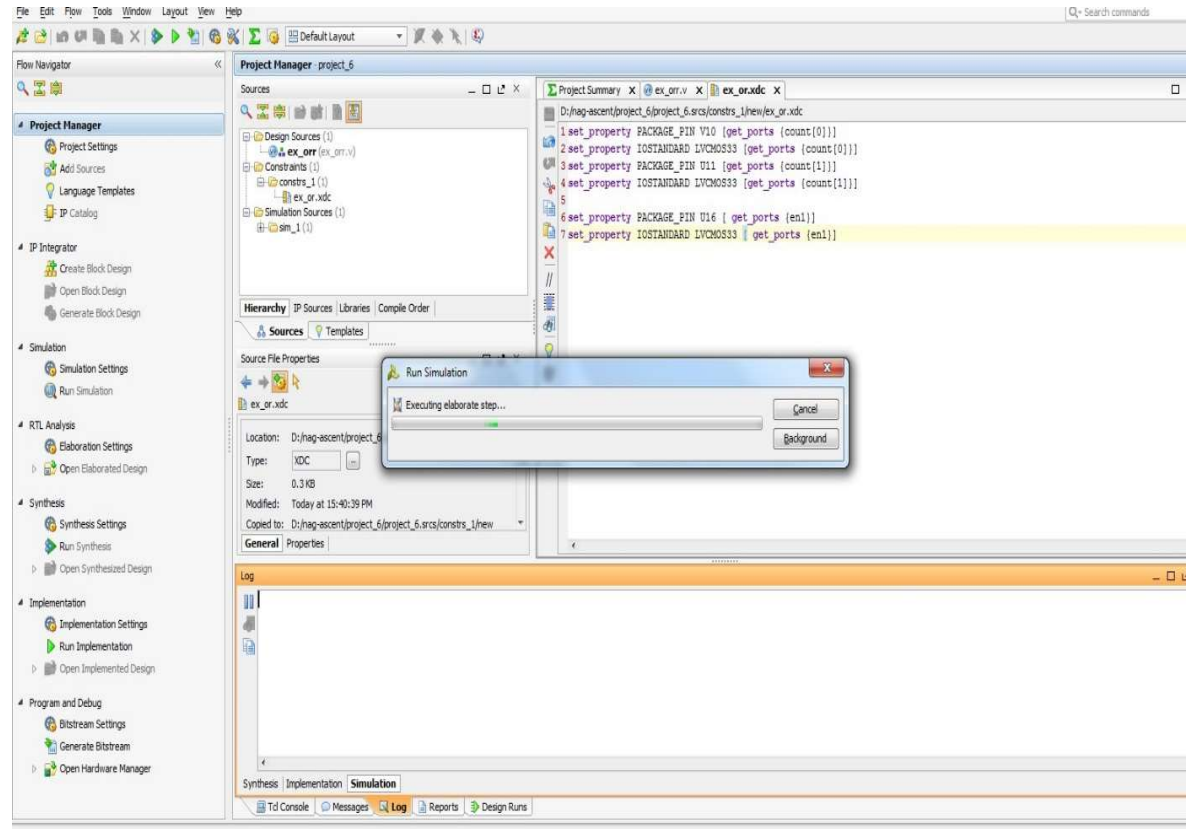

File Edit Flow Tools Window Layout View Help
Default Layout
Ready
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
Project Manager - project
Sources
Design Sources (2)
Non-module Files (1)
dflipflop (dflipflop.v)
Constraints
Simulation Sources (2)
sim_1 (2)
Hierarchy Libraries Compile Order
Sources Templates
Source File Properties
dflipflop.v
Location: D:/nag-ascent/project/project.srcs/sources_1/new
Type: Verilog
Library: xil_defaultlib
Size: 0.5 KB
General Properties
Project Summary
Project Settings
Edit
Project name: project
Project location: D:/nag-ascent/project
Product family: Artix-7
Project part: xc7a100tcsg324-1
Top module name: dflipflop
Synthesis
Status: Not started
Messages: No errors or warnings
Part: xc7a100tcsg324-1
Strategy: Vivado Synthesis Defaults
Implementation
Status: Not started
Messages: No errors or warnings
Part: xc7a100tcsg324-1
Strategy: Vivado Implementation Defaults
Incremental compile: None
DRC Violations
Run Implementation to see DRC results
Timing
Run Implementation to see timing results
Utilization
Run Synthesis to see utilization results
Power
Run Implementation to see power results
Design Runs
Name Constraints Status Progress WNS TNS WHS THS TPWS Failed Routes LUT % LUTs FF % FFs BRAM % BRAMs DSP % DSPs Start
synth_1 constrs_1 Not started 0%
impl_1 constrs_1 Not started 0%
Tcl Console Messages Log Reports Design Runs
dflipflop.v
D:/nag-ascent/project/project.srcs/sources_1/new/dflipflop.v
// Create Date: 02/22/2016 12:44:00 PM
// Design Name:
// Module Name: dflipflop
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
module dflipflop(
input a,b,
output y
);
endmodule

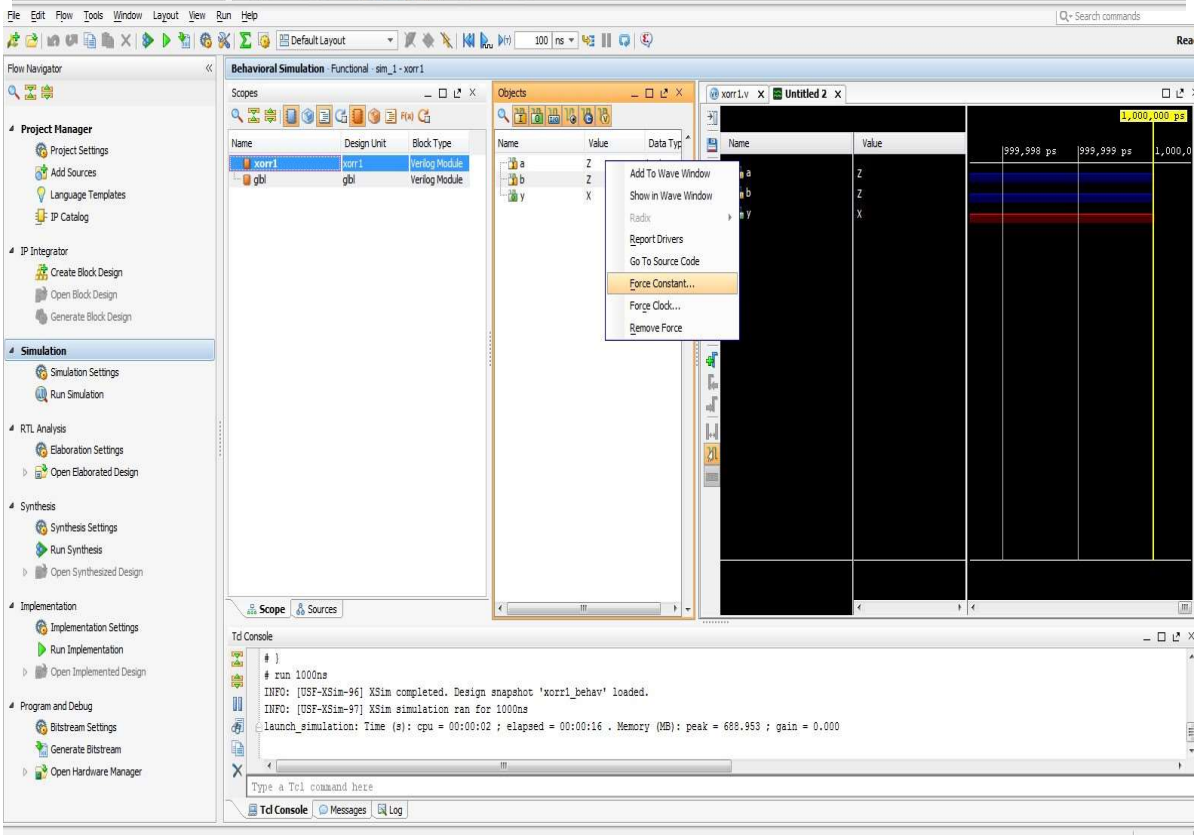

File Edit Flow Tools Window Layout View Help
Default Layout
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
Project Manager - project_6
Sources
Design Sources (1)
ex_orr (ex_orr.v)
Constraints
Simulation Sources (1)
sim_1 (1)
Hierarchy Libraries Compile Order
Sources Templates
Source File Properties
ex_orr.v
Location: D:/nag-ascent/project_6/project_6.srcs/sources_1/new
Type: Verilog
Library: xil_defaultlib
Size: 0.5 KB
General Properties
Project Summary
ex_orr.v *
D:/nag-ascent/project_6/project_6.srcs/sources_1/new/ex_orr.v
// Module Name: ex_orr
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
module ex_orr(
input a,b,
input y
);
assign y= a^b;
endmodule
Design Runs
Name Constraints Status Progress WNS TNS WHS THS TPWS Failed Routes LUT % LUTs FF % FFs BRAM % BRAMs DSP % DSPs Start
synth_1 constrs_1 Not started 0%
impl_1 constrs_1 Not started 0%
project_6 - [D:/nag-ascent/project_6/project_6.xpr] - Vivado 2015.1
Add Sources
VIVADO
Add Sources
This guides you through the process of adding and creating sources for your project
Add or create constraints
Add or create design sources
Add or create simulation sources
Add or create DSP sources
Add existing block design sources
Add existing IP
XILINX
ALL PROGRAMMABLE.
To continue, click Next
< Back
Next >
Finish
Cancel
Tcl Console
Messages
Log
Reports
Design Runs

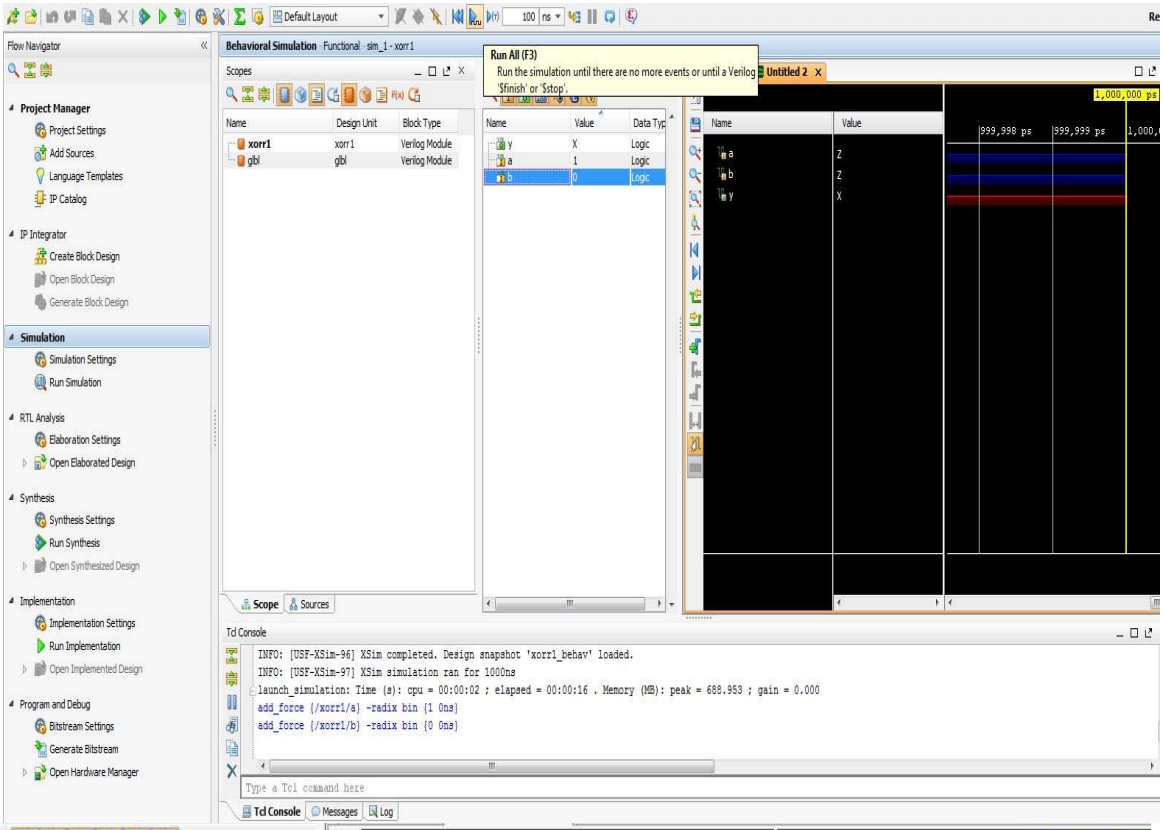

Add Sources
Add or Create Constraints
Specify or create constraint files for physical and timing constraint to add to your project.
Specify constraint set:
constrs_1 (active)
Add Files...
Create File...
Press the + button to Add Files or Create File
Copy constraints files into project
< Back
Next >
Finish
Cancel
Create Constraints File
Create a new constraints file and add it to your project
File type:
XDC
File name:
ex_or
File location:
<Local to Project>
File location
OK
Cancel
Constraint File
Location
ex_or.xdc
<Local to Project>
D:/nag-ascent/project_6/project_6.srcs/sources_1/new/ex_orr.v
8 // Module Name: ex_orr

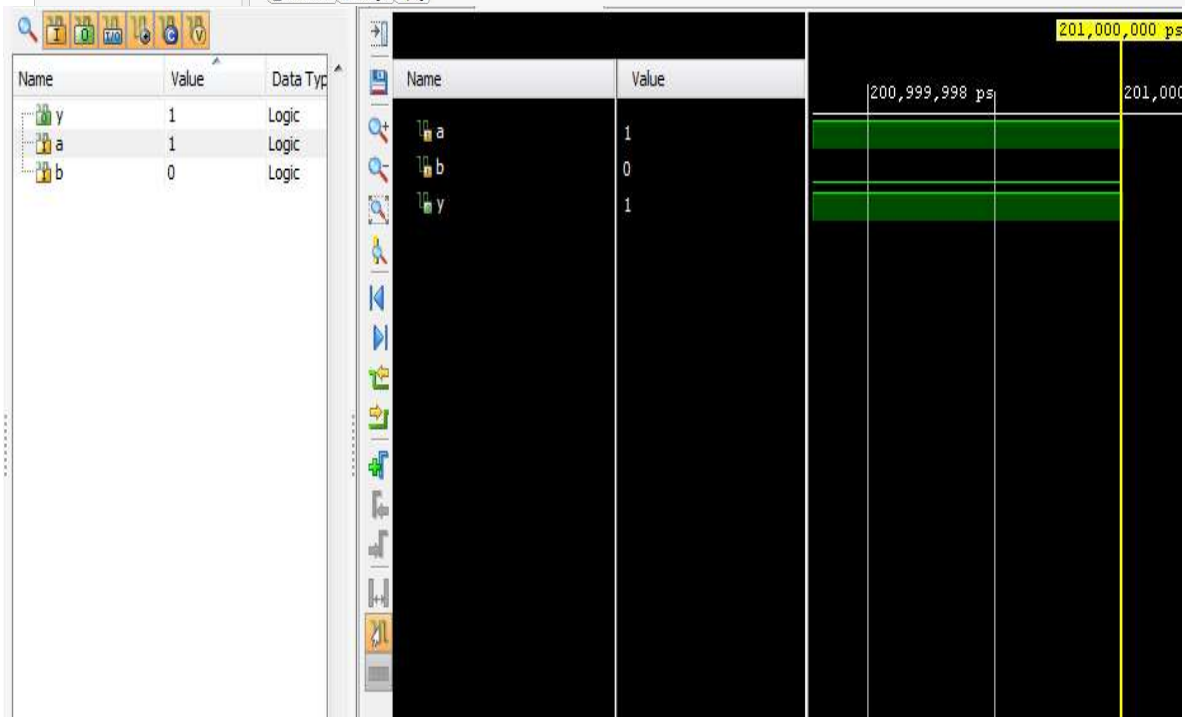

project_6 - [D:/nag-ascent/project_6/project_6.xpr] - Vivado 2015.1
File Edit Flow Tools Window Layout View Help
Default Layout
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
Project Manager - project_6
Sources
Design Sources (1)
ex_orr (ex_orr.v)
Constraints (1)
constrs_1 (1)
ex_or.xdc
Simulation Sources (1)
sim_1 (1)
Hierarchy Libraries Compile Order
Sources Templates
Source File Properties
ex_or.xdc
Location: D:/nag-ascent/project_6/project_6.srcs/constrs_1/new
Type: XDC
Size: 0.0 KB
Modified: Today at 15:38:06 PM
Copied to: D:/nag-ascent/project_6/project_6.srcs/constrs_1/new
General Properties
Project Summary
ex_orr.v
D:/nag-ascent/project_6/project_6.srcs/sources_1/new/ex_orr.v
// Module Name: ex_orr
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
module ex_orr(
input a,b,
input y
);
assign y= a^b;
endmodule
Design Runs
Name Constraints Status Progress WNS TNS WHS THS TPWS Failed Routes LUT % LUTs FF % FFs BRAM % BRAMs DSP % DSPs Start
synth_1 constrs_1 Not started 0%
impl_1 constrs_1 Not started 0%
Tcl Console Messages Log Reports Design Runs
ex_or.xdc *
D:/nag-ascent/project_6/project_6.srcs/constrs_1/new/ex_or.xdc
set_property PACKAGE_PIN V10 [get_ports {count[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {count[0]}]
set_property PACKAGE_PIN U11 [get_ports {count[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {count[1]}]
set_property PACKAGE_PIN U16 [get_ports {en1}]
set_property IOSTANDARD LVCMOS33 [get_ports {en1}]

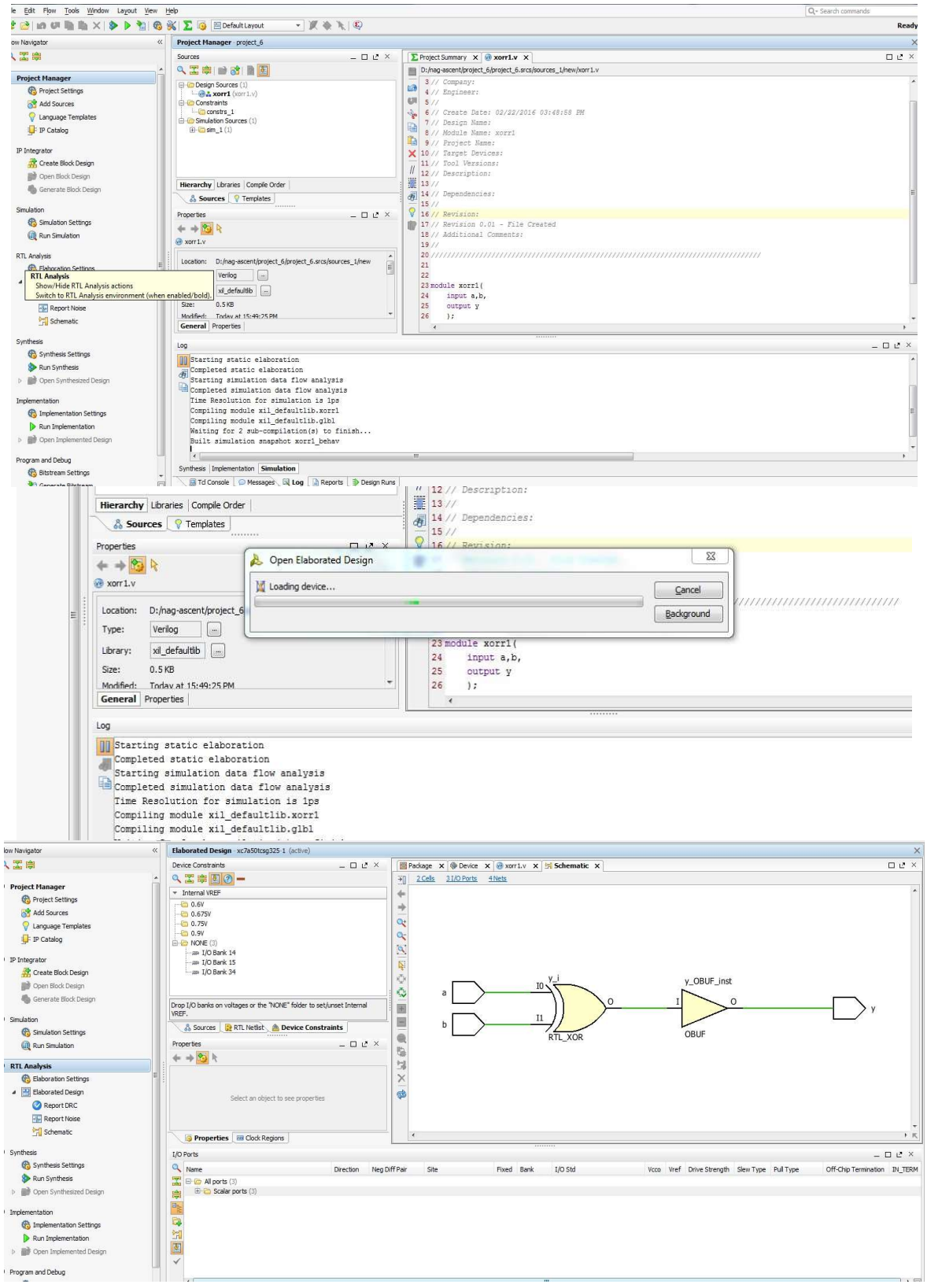

project_6 - [D:/nag-ascent/project_6/project_6.xpr] - Vivado 2015.1
File Edit Flow Tools Window Layout View Help
Default Layout
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
Project Manager - project_6
Sources
Design Sources (1)
ex_orr (ex_orr.v)
Constraints (1)
constrs_1 (1)
ex_or.xdc
Simulation Sources (1)
sim_1 (1)
Hierarchy Libraries Compile Order
Sources Templates
Source File Properties
ex_or.xdc
Location: D:/nag-ascent/project_6/project_6.srcs/constrs_1/new
Type: XDC
Size: 0.0 KB
Modified: Today at 15:38:06 PM
Copied to: D:/nag-ascent/project_6/project_6.srcs/constrs_1/new
General Properties
Project Summary
ex_orr.v
ex_or.xdc *
D:/nag-ascent/project_6/project_6.srcs/constrs_1/new/ex_or.xdc
1 set_property PACKAGE_PIN V10 [get_ports {count[0]}]
2 set_property IOSTANDARD LVCMOS33 [get_ports {count[0]}]
3 set_property PACKAGE_PIN U11 [get_ports {count[1]}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {count[1]}]
5
6 set_property PACKAGE_PIN U16 [get_ports {en1}]
7 set_property IOSTANDARD LVCMOS33 [get_ports {en1}]
Save File (Ctrl+S)
Save changes to the current file.
Design Runs
Name Constraints Status Progress WNS TNS WHS THS TPWS Failed Routes LUT % LUTs FF % FFs BRAM % BRAMs DSP % DSPs Start
synth_1 constrs_1 Not started 0%
impl_1 constrs_1 Not started 0%
Tcl Console Messages Log Reports Design Runs
Run Behavioral Simulation
Run Post-Synthesis Functional Simulation
Run Post-Synthesis Timing Simulation
Run Post-Implementation Functional Simulation
Run Post-Implementation Timing Simulation
Properties
Select an object to see properties
Project Summary
Project Settings
Edit
Project name: project_6
Project location: D:/nag-ascent/project_6
Product family: Artix-7
Project part: xc7a50tcsg325-1
Top module name: ex_orr
Synthesis
Status: Not started
Messages: No errors or warnings
Part: xc7a50tcsg325-1
Strategy: Vivado Synthesis Defaults
Implementation
Status: Not started
Messages: No errors or warnings
Part: xc7a50tcsg325-1
Strategy: Vivado Implementation Defaults
Incremental compile: None
DRC Violations
Run Implementation to see DRC results
Timing
Run Implementation to see timing results
Utilization
Run Synthesis to see utilization results
Power
Run Implementation to see power results

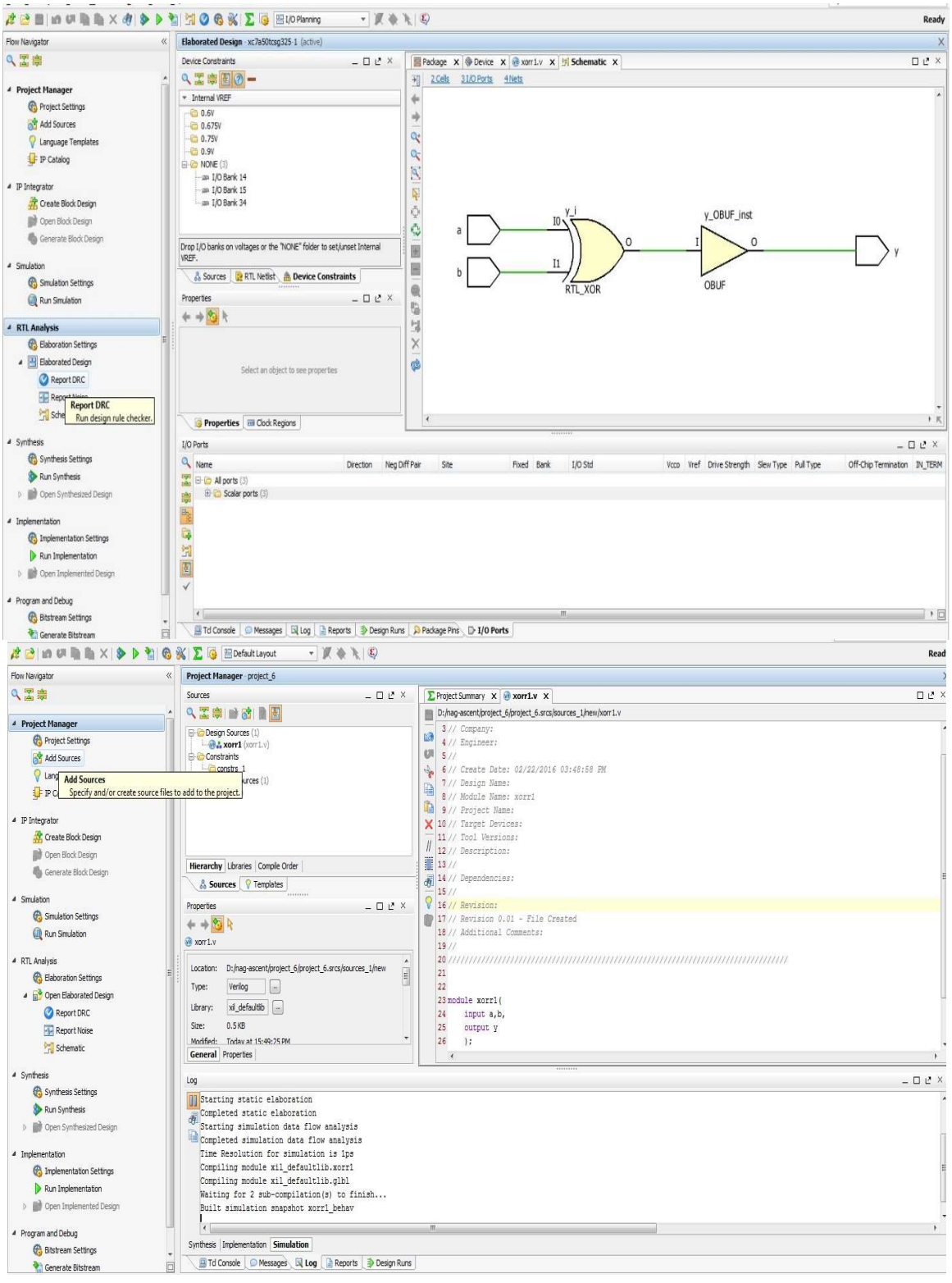

File Edit Flow Tools Window Layout View Help
Default Layout
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
Project Manager - project_6
Sources
Design Sources (1)
ex_orr (ex_orr.v)
Constraints (1)
constrs_1 (1)
ex_or.xdc
Simulation Sources (1)
sim_1 (1)
Hierarchy IP Sources Libraries Compile Order
Sources Templates
Source File Properties
ex_or.xdc
Location: D:/nag-ascent/project_6
Type: XDC
Size: 0.3 KB
Modified: Today at 15:40:39 PM
Copied to: D:/nag-ascent/project_6/project_6.srcs/constrs_1/new
General Properties
Project Summary
ex_orr.v
ex_or.xdc
D:/nag-ascent/project_6/project_6.srcs/constrs_1/new/ex_or.xdc
1 set_property PACKAGE_PIN V10 [get_ports {count[0]}]
2 set_property IOSTANDARD LVCMOS33 [get_ports {count[0]}]
3 set_property PACKAGE_PIN U11 [get_ports {count[1]}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {count[1]}]
5
6 set_property PACKAGE_PIN U16 [get_ports {en1}]
7 set_property IOSTANDARD LVCMOS33 [get_ports {en1}]
Run Simulation
Executing elaborate step...
Cancel
Background
Log
Synthesis Implementation Simulation
Tcl Console Messages Log Reports Design Runs

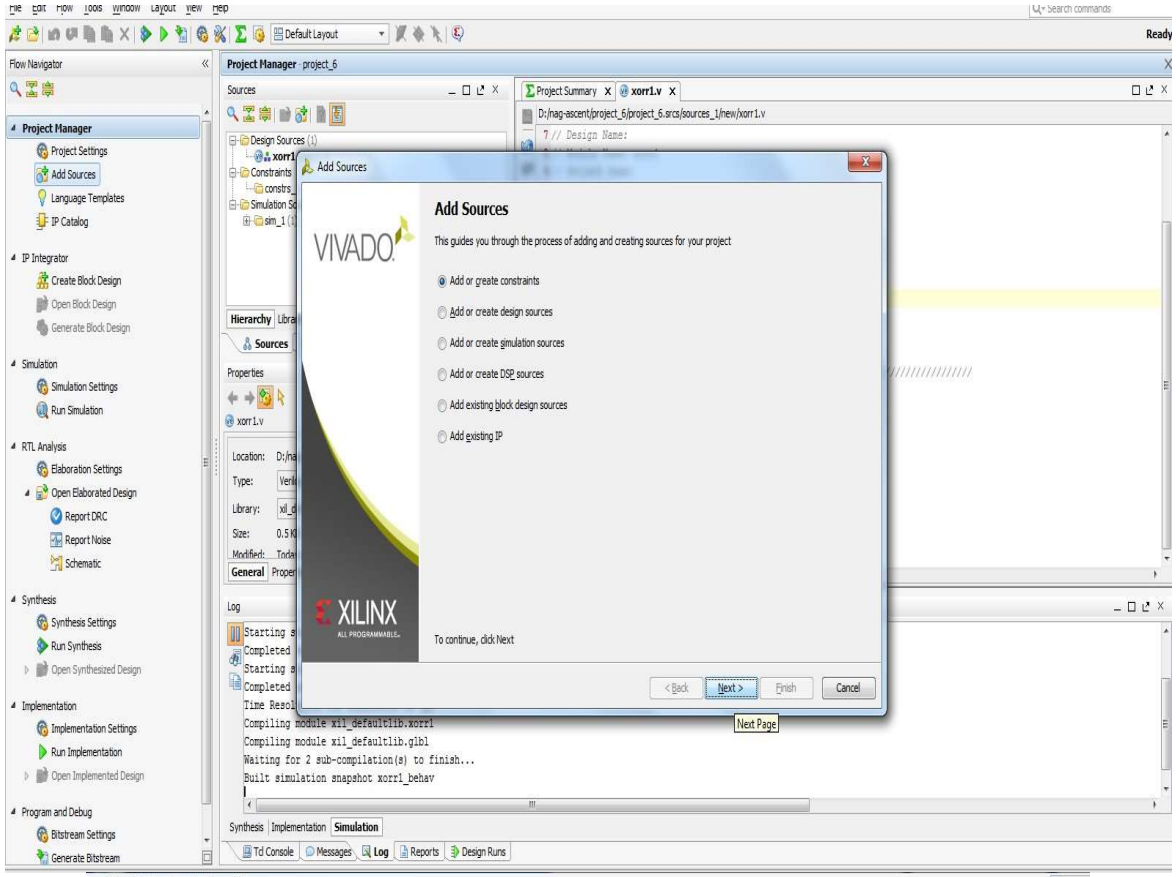

File Edit Flow Tools Window Layout View Run Help
Default Layout
100 ns
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
Behavioral Simulation - Functional - sim_1 - xorr1
Scopes
Name Design Unit Block Type
xorr1 xorr1 Verilog Module
glbl glbl Verilog Module
Objects
Name Value
a Z
b Z
y X
Add To Wave Window
Show in Wave Window
Radix
Report Drivers
Go To Source Code
Force Constant...
Force Clock...
Remove Force
xorr1.v
Untitled 2
1,000,000 ps
999,998 ps
999,999 ps
Scope Sources
Tcl Console
}
run 1000ns
INFO: [USF-XSim-96] XSim completed. Design snapshot 'xorr1_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:16 . Memory (MB): peak = 688.953 ; gain = 0.000
Type a Tcl command here
Tcl Console Messages Log

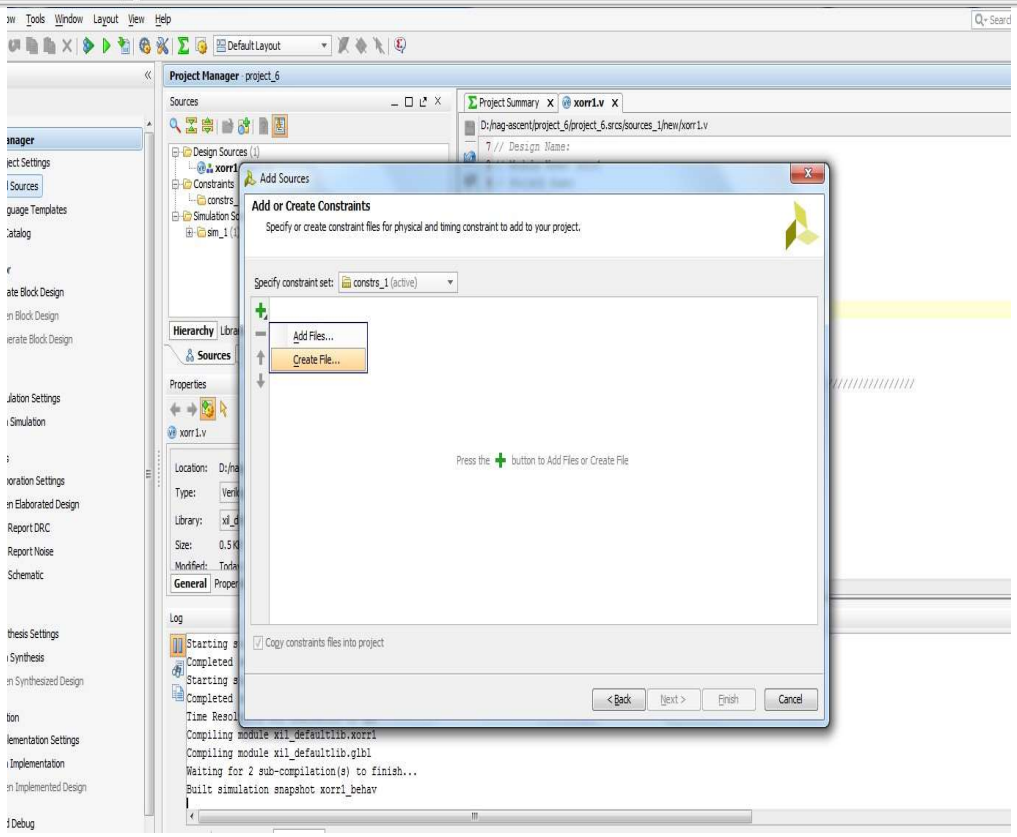

Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
Behavioral Simulation - Functional - sim_1 - xorr1
Run All (F3)
Run the simulation until there are no more events or until a Verilog '$finish' or '$stop'.
Untitled 2
Scopes
Name
Design Unit
Block Type
xorr1
xorr1
Verilog Module
glbl
glbl
Verilog Module
Value
Data Typ
y
X
Logic
a
1
Logic
b
0
Logic
Scope
Sources
Tcl Console
INFO: [USF-XSim-96] XSim completed. Design snapshot 'xorr1_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:16 . Memory (MB): peak = 688.953 ; gain = 0.000
add_force {/xorr1/a} -radix bin {1 0ns}
add_force {/xorr1/b} -radix bin {0 0ns}
Type a Tcl command here
Tcl Console
Messages
Log
1,000,000 ps
999,998 ps
999,999 ps

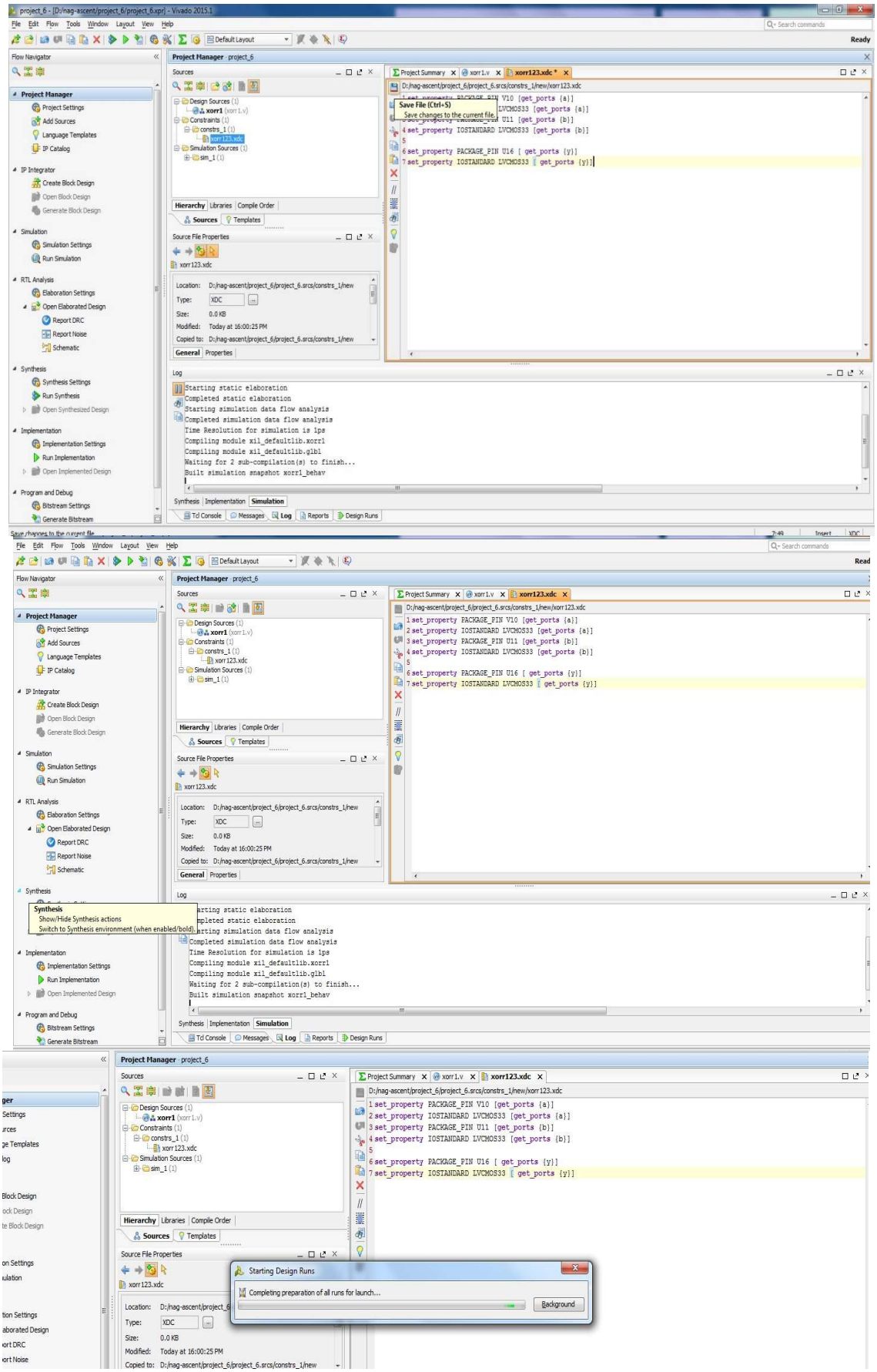

Name
Value
Data Typ
y
1
Logic
a
1
Logic
b
0
Logic
201,000,000 ps
200,999,998 ps
201,000

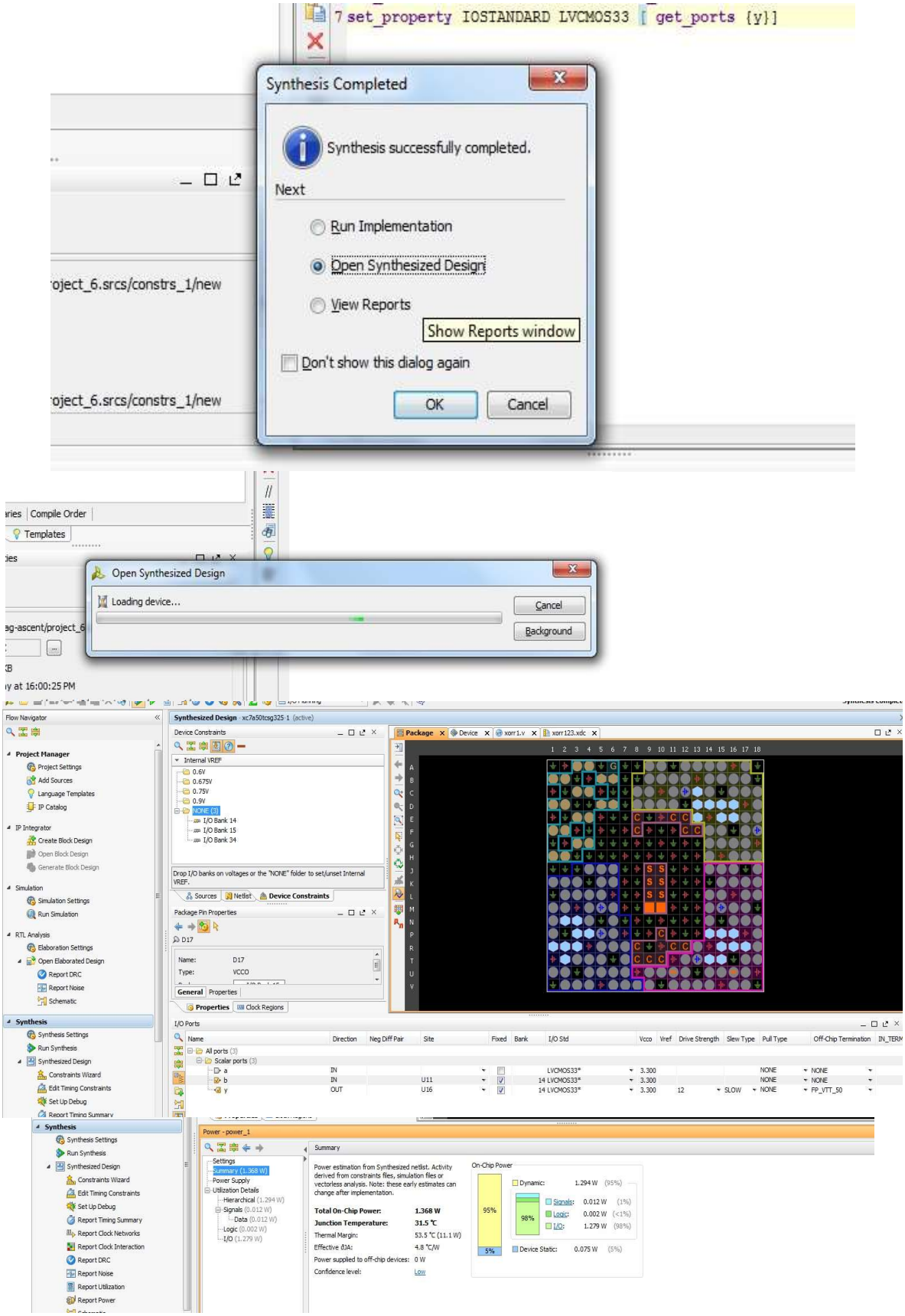

Open Elaborated Design
Loading device...
Cancel
Background
Starting static elaboration
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling module xil_defaultlib.xorr1
Compiling module xil_defaultlib.glbl
y_i
y_OBUF_inst
RTL_XOR
OBUF

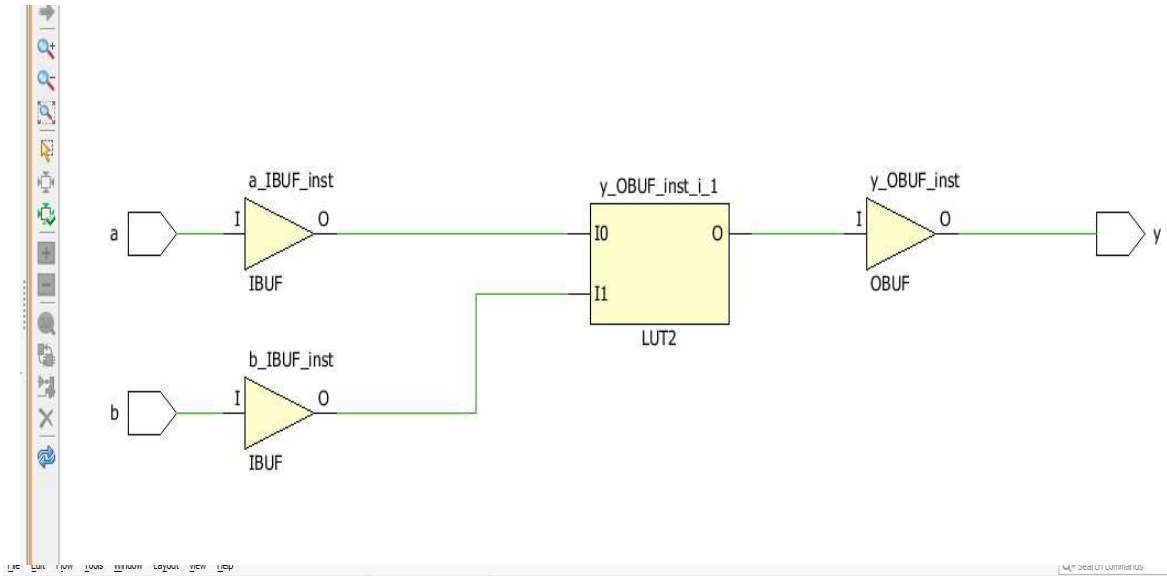

Ready
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Elaborated Design
Report DRC
Report DRC
Run design rule checker.
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Elaborated Design - xc7a50tcsg325-1 (active)
Device Constraints
Internal VREF
0.6V
0.675V
0.75V
0.9V
NONE (3)
I/O Bank 14
I/O Bank 15
I/O Bank 34
Drop I/O banks on voltages or the "NONE" folder to set/unset Internal VREF.
Sources
RTL Netlist
Device Constraints
Properties
Select an object to see properties
Properties
Clock Regions
Package
Device
xorr1.v
Schematic
2 Cells
3 I/O Ports
4 Nets
a
b
I0
I1
y_i
O
RTL_XOR
I
y_OBUF_inst
O
OBUF
y
I/O Ports
Name
Direction
Neg Diff Pair
Site
Fixed
Bank
I/O Std
Vcco
Vref
Drive Strength
Slew Type
Pull Type
Off-Chip Termination
IN_TERM
All ports (3)
Scalar ports (3)
Tcl Console
Messages
Log
Reports
Design Runs
Package Pins
I/O Ports
Default Layout
Read
Flow Navigator
Project Manager
Project Settings
Add Sources
Add Sources
Specify and/or create source files to add to the project.
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Report DRC
Report Noise
Schematic
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Project Manager - project_6
Sources
Design Sources (1)
xorr1 (xorr1.v)
Constraints
constrs_1
Hierarchy
Libraries
Compile Order
Sources
Templates
Properties
xorr1.v
Location: D:/nag-ascent/project_6/project_6.srcs/sources_1/new
Type: Verilog
Library: xil_defaultlib
Size: 0.5 KB
Modified: Today at 15:49:25 PM
General
Properties
Project Summary
xorr1.v
D:/nag-ascent/project_6/project_6.srcs/sources_1/new/xorr1.v
// Company:
// Engineer:
//
// Create Date: 02/22/2016 03:48:58 PM
// Design Name:
// Module Name: xorr1
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
module xorr1(
input a,b,
output y
);
Log
Starting static elaboration
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling module xil_defaultlib.xorr1
Compiling module xil_defaultlib.glbl
Waiting for 2 sub-compilation(s) to finish...
Built simulation snapshot xorr1_behav
Synthesis
Implementation
Simulation
Tcl Console
Messages
Log
Reports
Design Runs

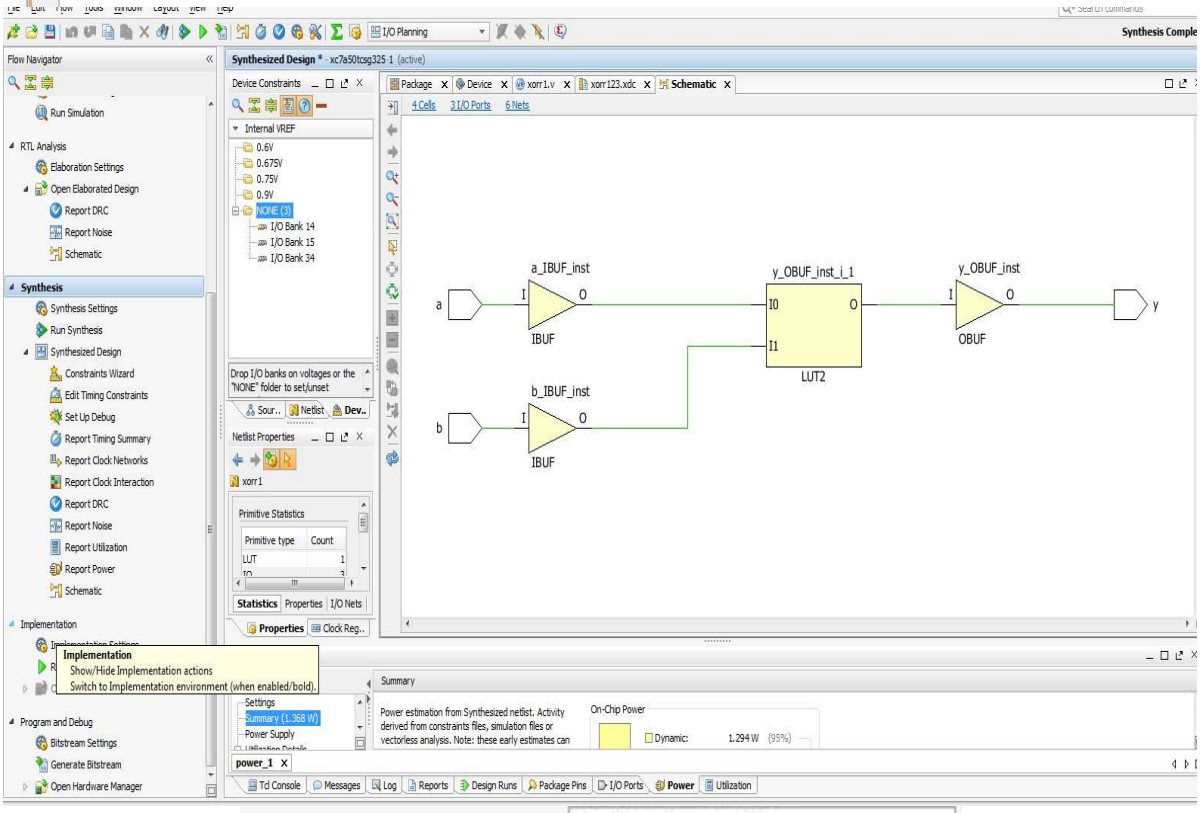

Add Sources
Add Sources
This guides you through the process of adding and creating sources for your project
Add or create constraints
Add or create design sources
Add or create simulation sources
Add or create DSP sources
Add existing block design sources
Add existing IP
To continue, click Next
< Back
Next >
Finish
Cancel
Next Page
VIVADO
XILINX
ALL PROGRAMMABLE.
Compiling module xil_defaultlib.xorr1
Compiling module xil_defaultlib.glbl
Waiting for 2 sub-compilation(s) to finish...
Built simulation snapshot xorr1_behav

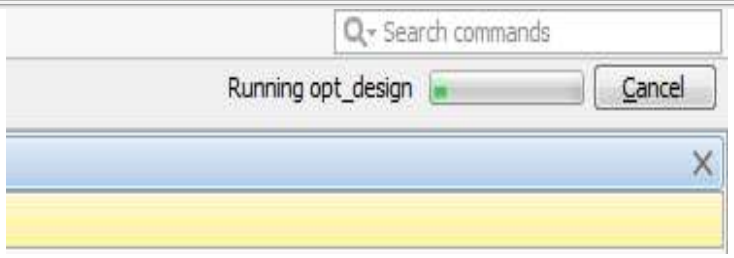

Add Sources
Add or Create Constraints
Specify or create constraint files for physical and timing constraint to add to your project.
Specify constraint set:
constrs_1 (active)
Add Files...
Create File...
Press the + button to Add Files or Create File
Copy constraints files into project
< Back
Next >
Finish
Cancel
Compiling module xil_defaultlib.xorr1
Compiling module xil_defaultlib.glbl
Waiting for 2 sub-compilation(s) to finish...
Built simulation snapshot xorr1_behav

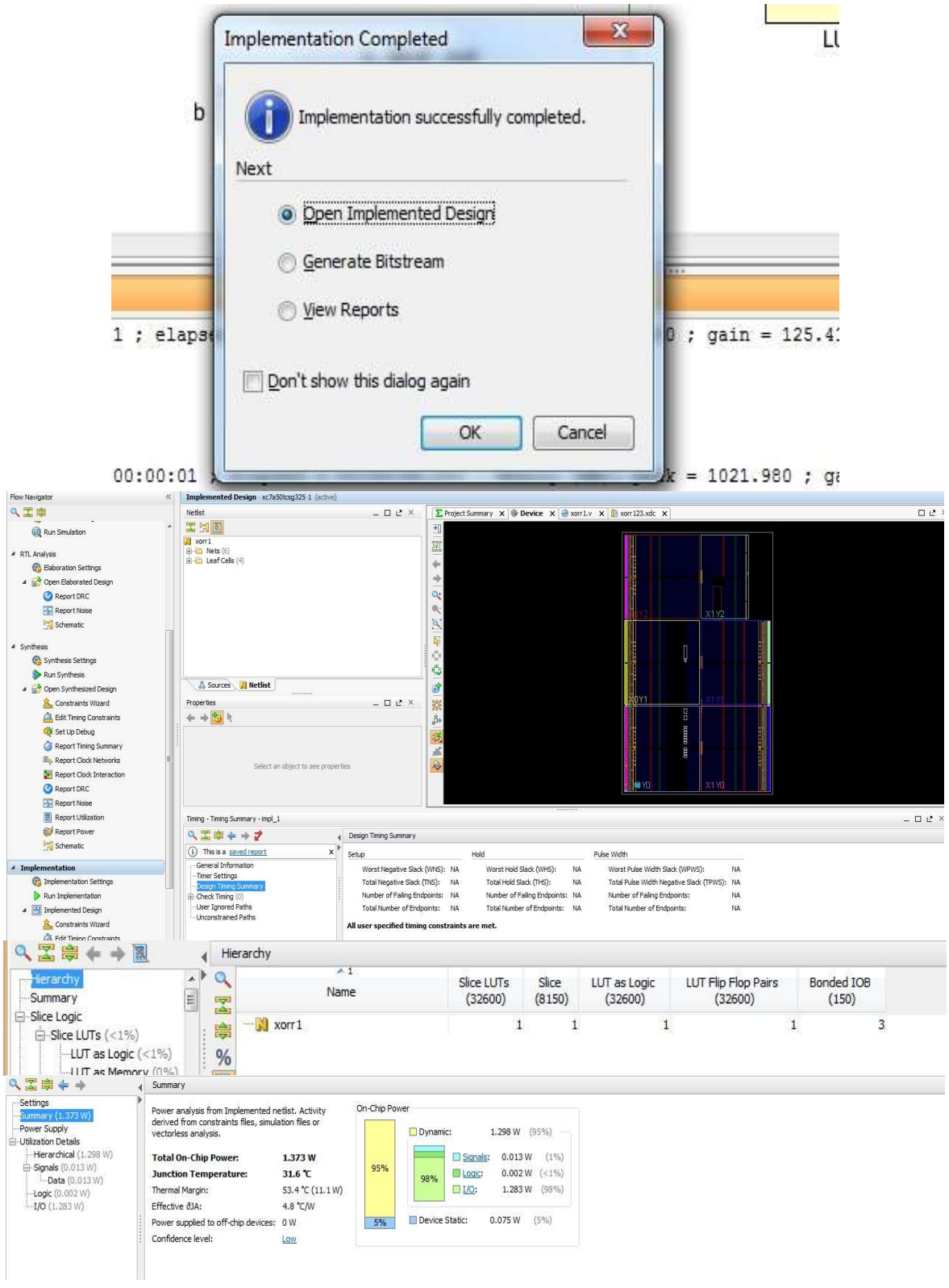

project_6 - [D:/nag-ascent/project_6/project_6.xpr] - Vivado 2015.1
File Edit Flow Tools Window Layout View Help
Default Layout
Ready
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Report DRC
Report Noise
Schematic
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Project Manager - project_6
Sources
Design Sources (1)
xorr1 (xorr1.v)
Constraints (1)
constrs_1 (1)
xorr123.xdc
Simulation Sources (1)
sim_1 (1)
Hierarchy Libraries Compile Order
Sources Templates
Source File Properties
xorr123.xdc
Location: D:/nag-ascent/project_6/project_6.srcs/constrs_1/new
Type: XDC
Size: 0.0 KB
Modified: Today at 16:00:25 PM
Copied to: D:/nag-ascent/project_6/project_6.srcs/constrs_1/new
General Properties
Project Summary
xorr1.v
xorr123.xdc *
D:/nag-ascent/project_6/project_6.srcs/constrs_1/new/xorr123.xdc
Save File (Ctrl+S)
Save changes to the current file.
LVCMOS33 [get_ports {a}]
U11 [get_ports {b}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {b}]
5
6 set_property PACKAGE_PIN U16 [get_ports {y}]
7 set_property IOSTANDARD LVCMOS33 [get_ports {y}]
Log
Starting static elaboration
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling module xil_defaultlib.xorr1
Compiling module xil_defaultlib.glbl
Waiting for 2 sub-compilation(s) to finish...
Built simulation snapshot xorr1_behav
Synthesis Implementation Simulation
Tcl Console Messages Log Reports Design Runs
Synthesis
Show/Hide Synthesis actions
Switch to Synthesis environment (when enabled/bold).
1 set_property PACKAGE_PIN V10 [get_ports {a}]
2 set_property IOSTANDARD LVCMOS33 [get_ports {a}]
3 set_property PACKAGE_PIN U11 [get_ports {b}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {b}]
5
6 set_property PACKAGE_PIN U16 [get_ports {y}]
7 set_property IOSTANDARD LVCMOS33 [get_ports {y}]
Starting Design Runs
Completing preparation of all runs for launch...
Background


7 set_property IOSTANDARD LVCMOS33 [get_ports {y}]
Synthesis Completed
Synthesis successfully completed.
Next
Run Implementation
Open Synthesized Design
View Reports
Show Reports window
Don't show this dialog again
OK
Cancel
Open Synthesized Design
Loading device...
Cancel
Background
Flow Navigator
Project Manager
IP Integrator
Simulation
RTL Analysis
Synthesis
Synthesized Design
Device Constraints
Package Pin Properties
I/O Ports
Power - power_1
Summary
Total On-Chip Power: 1.368 W
Junction Temperature: 31.5 °C


a_IBUF_inst
I
O
IBUF
a
y_OBUF_inst_i_1
I0
I1
O
LUT2
y_OBUF_inst
I
O
OBUF
y
b_IBUF_inst
I
O
IBUF
b


Flow Navigator
Synthesized Design * xc7a50tcsg325-1 (active)
Device Constraints
Schematic
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Report DRC
Report Noise
Schematic
Synthesis
Synthesis Settings
Run Synthesis
Synthesized Design
Constraints Wizard
Edit Timing Constraints
Set Up Debug
Report Timing Summary
Report Clock Networks
Report Clock Interaction
Report DRC
Report Noise
Report Utilization
Report Power
Schematic
Implementation
Implementation
Show/Hide Implementation actions
Switch to Implementation environment (when enabled/bold).
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager
a_IBUF_inst
y_OBUF_inst_i_1
y_OBUF_inst
b_IBUF_inst
IBUF
LUT2
OBUF
Summary
Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can
On-Chip Power
Dynamic: 1.294 W (95%)


Search commands
Running opt_design
Cancel


Implementation Completed
Implementation successfully completed.
Next
Open Implemented Design
Generate Bitstream
View Reports
Don't show this dialog again
OK
Cancel
Hierarchy
Summary
Slice Logic
Slice LUTs (<1%)
LUT as Logic (<1%)
Name
Slice LUTs (32600)
Slice (8150)
LUT as Logic (32600)
LUT Flip Flop Pairs (32600)
Bonded IOB (150)
xorr1
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.
Total On-Chip Power: 1.373 W
Junction Temperature: 31.6 °C
Thermal Margin: 53.4 °C (11.1 W)
Effective θJA: 4.8 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Low
On-Chip Power
Dynamic: 1.298 W (95%)
Signals: 0.013 W (1%)
Logic: 0.002 W (<1%)
I/O: 1.283 W (98%)
Device Static: 0.075 W (5%)

CHAPTER 5: PERFORMANCE ANALYSIS

Evaluating the performance of turbo codes involves two main approaches: theoretical analysis and computer simulation. Theoretical analysis is challenging due to the complex structure of turbo codes. Some research papers have attempted this approach, but their results often don't align closely with simulation results. Instead, computer simulation provides a more practical way to assess turbo code performance.

The turbo code encoder consists of two recursive systematic convolutional (RSC) encoders connected in parallel.These two component encoders are separated by a random interleaver, which scrambles the bit order in the encoded stream.To ensure proper termination of the RSC encoders, m tail bits are added to the systematic code stream.

Turbo decoding is an iterative process. The decoder uses the Viterbi algorithm with soft input and soft output (often referred to as the Soft Output Viterbi Algorithm or SOVA).The decoder iteratively refines its estimates of the transmitted bits by exchanging information between the two component decoders. Researchers analyze different code parameters to understand the trade-offs between code gain, bit error rate (BER), and computational complexity. BER curves are commonly used to visualize how turbo codes perform under varying signal-to-noise ratios (SNRs).

Based on simulation results, engineers optimize turbo code performance by adjusting parameters like interleaver design, code rates, and decoder algorithms. Turbo codes are widely used in wireless communication systems, satellite links, and digital broadcasting.

CHAPTER 6: VERILOG

6.1 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.

Verilog 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.

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 asaHDL 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.

6.2 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,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 as identifiers

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.1name 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

6.3 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 An array of 8 count variables

6.4 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.

6.5 STURCTURAL 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 builtin gate types, the names of these gates are reserved words, some of these are

Syntax of Verilog instance statements:

Component_name

instance-identifier(expression……expresssion);

Component_name instance-identifier (.port-name(expression),

: :

.port-name(expression));

Basic gate primitives in Verilog with details:

Gate	Mode of instantiation	Output port(s)	Input port(s)
AND	**and** ga (o, i1, i2, . . . i8);	o	i1, i2, . .
OR	**or** gr (o, i1, i2, . . . i8);	o	i1, i2, . .
NAND	**nand** gna (o, i1, i2, . . . i8);	o	i1, i2, . .
NOR	**nor** gnr (o, i1, i2, . . . i8);	o	i1, i2, . .
XOR	**xor** gxr (o, i1, i2, . . . i8);	o	i1, i2, . .
XNOR	**xnor** gxn (o, i1, i2, . . . i8);	o	i1, i2, . .
BUF	**buf** gb (o1, o2, ... i);	o1, o2, o3, . .	i
NOT	**not** gn (o1, o2, o3, . . . i);	o1, o2, o3, . .	i

6.6 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 skipped. 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 skipped and execution proceeds with assignment6.

If the condition is not satisfied, assignment2 and assignment3 are skipped 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 putting in a default whether you need it or not.

If the defaults are don't cares, define them as 'x' and the logic minimizer will treat them as don't cares and save 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 case x(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

Examples

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 n

umber 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 7: RESULTS AND DISCUSSIONS

7.1 RESULTS

Simulation

The simulation is the process which is termed as the final verification in respect to its working where as the schematic is the verification of the connections and blocks. The simulation window is launched as shifting from implementation to the simulation on the home screen of the tool,andthe simulation window confines the output in the form of wave forms output. Here it has the flexibility of providing the different radix number systems. In this project serial architecture and parallel methodologies are used .

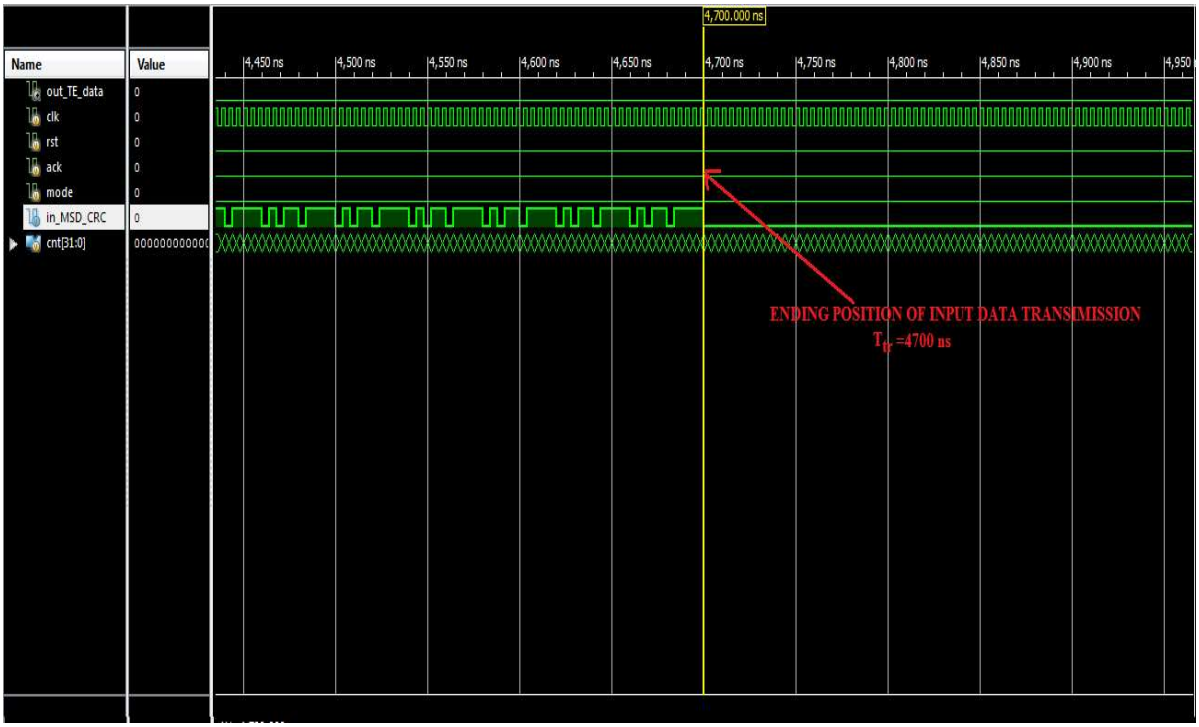


Fig:7.1 Simulated wave form of serial computation when mode is 0 at time 4700 ns

Fig:7.2 Simulated wave form of serial computation when mode is 0 at time 13918 ns

Required time for serial computation = 13618 – 4700 = 9218 ns

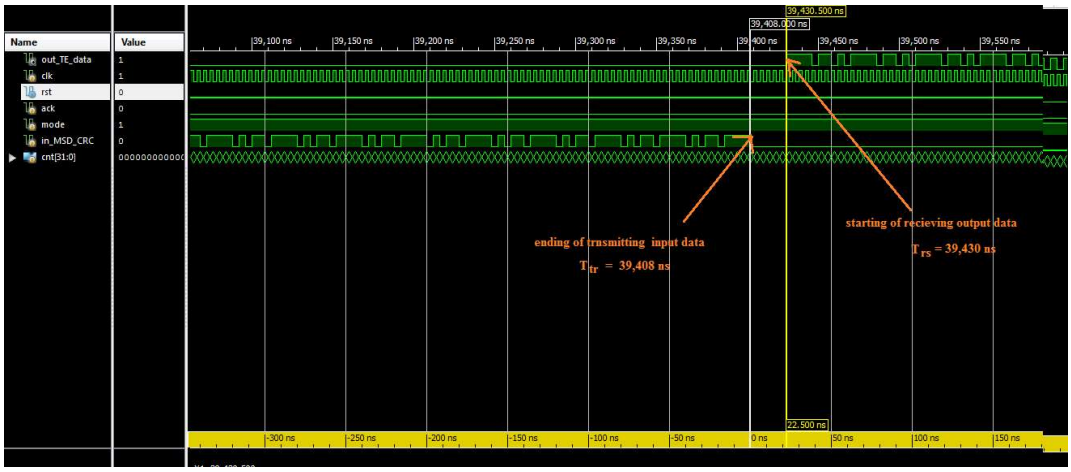


Fig: 7.3 Simulated wave form of parallel computation when mode is 1

Input data transmitting time Ttr = 39408 ns

Output data receiving time Trs = 39430 ns

Required time for parallel computation 39430 – 30408 = 22 ns

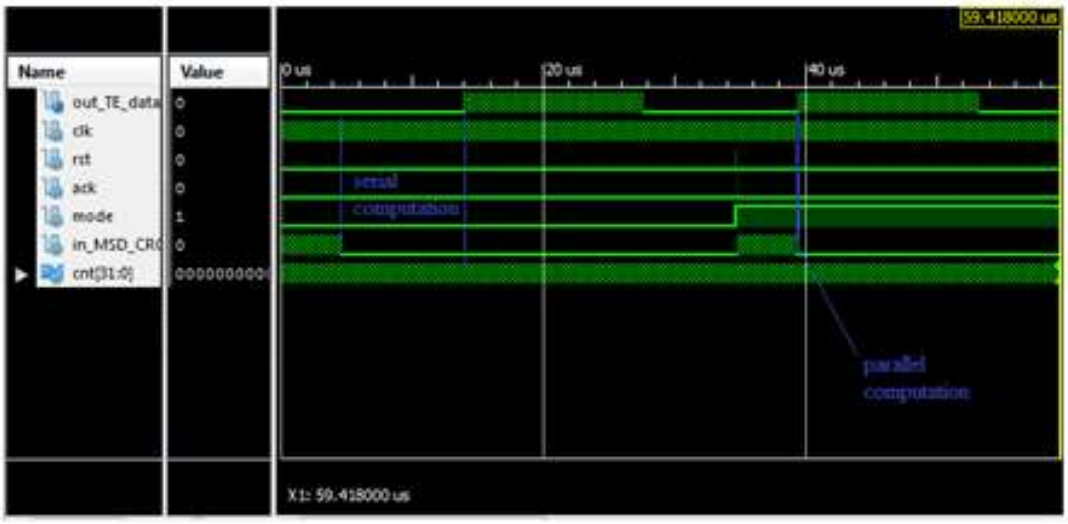


Fig:7.4 simulated wave form of turbo encoder

Parameter	Serial computation	Parallel computation
Required time to receiving output (ns)	**9218**	**22**

Table7.1: Clock comparison table

From table 7.1 the parallel computation method uses less number of clock cycles it will be causes to reduce the power consumption because of 50% of power of the design was utilized by clock pulse only.

Advantages, Disadvantages And Applications:

Advantages

The proposed design turbo encoder with parallel computation using less number of clock pulses to getting output. Infact 50% 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.

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.

CHAPTER 8: CONCLUSION & FUTURE SCOPE:

8.1 CONCLUSION

The Turbo encoder module is designed and implemented to be an embedded module in the IVS modem. FPGA technologies are employed to develop the Turbo encoder module. Xilinx tools and Verilog HDL are employed to design and simulate the module. Both serial and parallel computation techniques are studied for the encoding process. It is shown that the parallel computation can improve the chip size and processing time of the module. Comparing with the serial computation technique, the parallel computation encoding, using only 22 ns time to generate output and it improves the processing time by and serial computation logic utilization by 9218ns pulses. Additionally the proposed structure reduced area and power also. The processing time enhancement can be seen in both simulation and analyzing the chip processing.

8.2 FUTRESCOPE

Turbo encoders are error-correcting codes with performance close to the Shannon theoretical limit [SHA]. The encoder is formed by the parallel concatenation of two convolution codes separated by an inter leaver or permuter . In this project, I modeled and implemented a turbo encoder using magnitude comparator scheme using serial and parallel computing algorithm. The advantage of turbo codes over existing coding schemes is that it attains a very low BER at low signal-to-noise ratios, so it will be used in used in communications. This makes it suitable for wireless applications where low transmission power is desired. However, the performance of turbo encodes on Rayleigh and Ricean fading channels remain an active subject of research. The portability of this code to Advanced Design System (ADS) would be very beneficial for code re-usability and also the synthesis capability of ADS would result in faster product development.

CHAPTER:9 APPENDIX

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.

9.1 Overview

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).

9.2What is VLSI?

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

- 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 device

9.3 History 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.
- 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).

9.4 Advantages 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.

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.

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.

9.5 VLSI 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.

9.6 Applications 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 10 . SOURCE CODE:

```
module turbo_encoder
(       input   clk         ,
        input   rst         ,
        input   ack         ,
        input   mode        ,
        input   in_MSD_CRC ,

        output  out_TE_data
);


wire  [1147:0]  MSD_CRC ;
wire  [2:0]     PTAIL_1     ;
wire  [2:0]     TAIL_1      ;
wire  [1147:0] PARITY_1;
wire  [2:0]     PTAIL_2     ;
wire  [2:0]     TAIL_2      ;
wire  [1147:0] PARITY_2;


read_MSD_input read_MSD_input_i(
            .clk            (clk            ),  // I
            .rst            (rst            ),  // I
            .ack            (ack            ),  // I
            .in_MSD_CRC     (in_MSD_CRC     ),  // I

            .done           (done_rd_MSD    ),  // O
            .MSD_CRC        (MSD_CRC    )   // O
);

process_parity1 process_parity1_i(
            .clk            (clk            ),  // I
            .rst            (rst            ),  // I
            .mode           (mode           ),  // I
            .done_rd_MSD    (done_rd_MSD    ),  // I
            .MSD_CRC        (MSD_CRC    ),  // I
            .ack            (ack            ),  // I
            .in_MSD_CRC     (in_MSD_CRC     ),  // I

            .done_p1        (done_p1    ),  // O
            .PTAIL_1        (PTAIL_1    ),  // O
            .TAIL_1         (TAIL_1         ),  // O
```

```
        .PARITY_1 (PARITY_1    )  // O
);

process_parity2 process_parity2_i(
        .clk          (clk          ), // I
        .rst          (rst          ), // I
        .mode         (mode         ), // I
        .done_p1      (done_p1      ), // I
        .MSD_CRC      (MSD_CRC      ), // I
        .ack          (ack          ), // I
        .in_MSD_CRC   (in_MSD_CRC   ), // I

        .done_p2      (done_p2      ), // O
        .PTAIL_2      (PTAIL_2      ), // O
        .TAIL_2       (TAIL_2       ), // O
        .PARITY_2 (PARITY_2    )  // O
);

generate_output generate_output_i(
        .clk          (clk          ), // I
        .rst          (rst          ), // I

        .done_rd_MSD  (done_rd_MSD  ), // I
        .MSD_CRC      (MSD_CRC      ), // I

        .done_p1      (done_p1      ), // I
        .TAIL_1       (TAIL_1       ), // I
        .PTAIL_1      (PTAIL_1      ), // I
        .PARITY_1 (PARITY_1    ), // I

        .done_p2      (done_p2      ), // I
        .TAIL_2       (TAIL_2       ), // I
        .PTAIL_2      (PTAIL_2      ), // I
        .PARITY_2 (PARITY_2    ), // I

        .out_TE_data  (out_TE_data  )  // O
);
endmodule

// reading MSD and CRC data from serial input
module read_MSD_input(
        input         clk     ,
        input         rst     ,
        input         ack     ,
```

```
        input           in_MSD_CRC ,

        output          done       ,
        output [1147:0]         MSD_CRC
);
 reg [1147:0]   MSD_CRC_shift;
 reg [10:0]     shift_cnt;

 // generating done pulse and bulding MSD_CRC data
 assign MSD_CRC     =  MSD_CRC_shift ;
 assign done    =  (shift_cnt == 1148) ;

 // collecting inputs bit by bit through shift regisers
 always @(posedge clk or posedge rst)
  begin
    if (rst)
        MSD_CRC_shift <= 1148'd0 ;
    else if( ack || (shift_cnt < 1148 ) )
        MSD_CRC_shift <= {MSD_CRC_shift[1146:0],in_MSD_CRC} ;
  end

 // making count of every shift
 always @(posedge clk or posedge rst)
  begin
    if (rst)
        shift_cnt  <= 11'd0;
    else if ( ack )
        shift_cnt  <= 11'd1;
    else if (shift_cnt != 0 && shift_cnt <= 1148 )
        shift_cnt  <= shift_cnt + 1 ;
  end

endmodule


module process_parity1(
        input           clk       ,
        input           rst       ,
        input           mode   ,
        input           done_rd_MSD,
        input [1147:0] MSD_CRC     ,
        input           ack       ,
        input           in_MSD_CRC ,
```

```
        output          done_p1         ,
        output [2:0]    PTAIL_1         ,
        output [2:0]    TAIL_1          ,
        output [1147:0]         PARITY_1
);

reg [10:0]      shift_cnt;
reg [2:0]       parity1_shift_reg;
reg [1147:0]    parity1_reg;
reg [2:0]       ptail;
reg [2:0]       tail ;
wire            start_shift;
wire            in;

// mode 1:parallel 0: serial
assign start_shift      = mode ? ack : done_rd_MSD ;
assign done_p1                  = (shift_cnt == 1151);
assign PTAIL_1                  = ptail;
assign TAIL_1                   = tail ;
assign PARITY_1     = parity1_reg;
assign in                       = mode                  ? in_MSD_CRC    :
                          done_rd_MSD           ? MSD_CRC[0]    :
                          (shift_cnt < 1148)? MSD_CRC[shift_cnt]: parity1_shift_reg[2] ;

always @(posedge clk or posedge rst)
 begin
   if (rst)
        parity1_shift_reg <= 3'd0;
   else
        parity1_shift_reg <=
{parity1_shift_reg[1],parity1_shift_reg[0],(parity1_shift_reg[2]^parity1_shift_reg[1]^ in) };
 end


always @(posedge clk or posedge rst)
 begin
   if (rst)
        parity1_reg <= 1148'd0;
   else if( shift_cnt < 1148 )
        parity1_reg <=
{parity1_reg[1146:0],(parity1_shift_reg[2]^parity1_shift_reg[1]^parity1_shift_reg[0]^in)};
 end

always @(posedge clk or posedge rst)
```

```
 begin
   if (rst) begin
        ptail <= 3'd0;
        tail  <= 3'd0;
    end
   else if( shift_cnt >= 1148 && shift_cnt < 1151 ) begin
        ptail <=
{ptail[1:0],(parity1_shift_reg[2]^parity1_shift_reg[1]^parity1_shift_reg[0]^in)};
        tail  <= {tail[1:0],in};
    end
 end


// making count of every shift
always @(posedge clk or posedge rst)
 begin
   if (rst)
        shift_cnt  <= 11'd0;
   else if (start_shift)
        shift_cnt  <= 11'd1;
   else if (shift_cnt != 0 && shift_cnt <= (1148 + 3) )
        shift_cnt  <= shift_cnt + 1 ;
 end

endmodule



module process_parity2(
        input           clk     ,
        input           rst     ,
        input           mode    ,
        input           done_p1         ,
        input [1147:0] MSD_CRC     ,
        input           ack     ,
        input           in_MSD_CRC ,

        output          done_p2         ,
        output [2:0]    TAIL_2          ,
        output [2:0]    PTAIL_2         ,
        output [1147:0]         PARITY_2
);
```

```
reg [10:0]      shift_cnt;
reg [2:0]       parity2_shift_reg;
reg [1147:0]    parity2_reg;
reg [2:0]       ptail;
reg [2:0]       tail ;
wire            start_shift;
wire            in;
reg [3:0]       in_shift;
wire            interleaver_out;
wire            g1_D;
wire            g0_D;
wire            G_D;

// mode 1:parallel 0: serial
assign start_shift      = mode ? ack : done_p1 ;
assign done_p2                  = (shift_cnt == 1151);
assign PTAIL_2                  = ptail;
assign TAIL_2                   = tail ;
assign PARITY_2     = parity2_reg;
assign in                       = mode                      ? !in_MSD_CRC    :
                          done_p1           ? !MSD_CRC[0]     :
                          (shift_cnt < 1148)? !MSD_CRC[shift_cnt]: parity2_shift_reg[2] ;

// parity 2 is generated from interleaver output
// interleaver implementation
// G(D) = (1 ^ g1(D)) && ( 0 ^ g0(D))
// g0(D) = 1+D+(D)^3
// g1(D) = 1+(D)^2+(D)^3

always @(posedge clk or posedge rst)
 begin
   if (rst)
        in_shift  <= 4'b0;
   else
        in_shift  <= {in_shift[2:0],in};
 end

assign g0_D = (in_shift[0] ^ in_shift[1] ^ in_shift[3]);
assign g1_D = (in_shift[0] ^ in_shift[2] ^ in_shift[3]);
assign G_D  = ( (1 ^ g1_D) && ( 0 ^ g0_D) );
assign interleaver_out = G_D;

always @(posedge clk or posedge rst)
```

```
 begin
    if (rst)
         parity2_shift_reg <= 3'd0;
    else if (shift_cnt <= 1151)
         parity2_shift_reg <=
{parity2_shift_reg[1],parity2_shift_reg[0],(parity2_shift_reg[2]^parity2_shift_reg[1]^
interleaver_out) };
 end


 always @(posedge clk or posedge rst)
 begin
    if (rst)
         parity2_reg <= 1148'd0;
    else if( shift_cnt < 1148 )
         parity2_reg <=
{parity2_reg[1146:0],(parity2_shift_reg[2]^parity2_shift_reg[1]^parity2_shift_reg[0]^interle
aver_out)};
 end

 always @(posedge clk or posedge rst)
 begin
    if (rst) begin
         ptail <= 3'd0;
         tail  <= 3'd0;
     end
    else if( shift_cnt >= 1148 && shift_cnt < 1151 ) begin
         ptail <=
{ptail[1:0],(parity2_shift_reg[2]^parity2_shift_reg[1]^parity2_shift_reg[0]^interleaver_out)};
         tail  <= {tail[1:0],interleaver_out};
     end
 end


 // making count of every shift
 always @(posedge clk or posedge rst)
 begin
    if (rst)
         shift_cnt  <= 11'd0;
    else if (start_shift)
         shift_cnt  <= 11'd1;
    else if (shift_cnt != 0 && shift_cnt <= (1148 + 3) )
         shift_cnt  <= shift_cnt + 1 ;
 end
```

```
endmodule

module generate_output(
        input           clk             ,
        input           rst             ,

        input           done_rd_MSD,
        input [1147:0] MSD_CRC     ,

        input           done_p1         ,
        input [2:0]     TAIL_1          ,
        input [2:0]     PTAIL_1         ,
        input [1147:0] PARITY_1,

        input           done_p2         ,
        input [2:0]     TAIL_2          ,
        input [2:0]     PTAIL_2         ,
        input [1147:0] PARITY_2,

        output  reg     out_TE_data
);


reg [3455:0] out_data;
reg [12:0]   shift_cnt;


// aligning the output to send on serial lane
// output data is aligned in the following format
// MSB---------------------------------------------------------------->LSB
// 3455     3452      2304     2301     1153     1150    1147      0
//   | ptail2 | parity2 | ptail1 | parity1 | tail2 | tail1 | MSD_CRC |
//
//  LSB data will be sent to output.
//  So, the data is stored in reverse compared to paper

 always@ (posedge clk or posedge rst)
  begin
    if (rst)
        out_data <= 3456'd0 ;
    else
     begin
      if(done_rd_MSD)
```

```
            out_data[1147:0]    <= MSD_CRC ;
        if(done_p1) begin
            out_data[1150:1148] <= TAIL_1 ;
            out_data[2304:1154] <= {PTAIL_1,PARITY_1} ;
           end
        if(done_p2) begin
            out_data[1153:1151] <= TAIL_2 ;
            out_data[3455:2305] <= {PTAIL_2,PARITY_2} ;
           end
      end
  end

 // making count of every shift
 always @(posedge clk or posedge rst)
  begin
     if (rst)
          shift_cnt  <= 13'd0;
     else if (done_p2)
          shift_cnt  <= 13'd1;
     else if (shift_cnt != 0 && shift_cnt <= 3456 )
          shift_cnt  <= shift_cnt + 1 ;
  end

  // generating output data serially from the alogned data
 always@ (posedge clk or posedge rst)
  begin
    if (rst)
          out_TE_data  <= 1'b0;
    else if(done_p2)
          out_TE_data  <= out_data[0];
    else if(shift_cnt != 0 && shift_cnt < 3456 )
          out_TE_data  <= out_data[shift_cnt];
    else
          out_TE_data  <= 1'b0;
  end

endmodule
```

CHAPTER:11 BIBLIOGRAPHY:

1] “eCall data transfer; in-band modem solution; general description,” 3GPP,Tech. Rep. TS26.267.
[2] Eroupean Commission, “eCall: Time saved / lives saved,” Press Release,Brussels, August 21, 2015. website: http : //ec:europa:eu/digital –agenda/en/ecall - time - saved - lives - saved.
[3] A. Saleem et at. “Four-Dimensional Trellis Coded Modulation for Flexible Optical Communications,” IEEE Journal of Lightwave Technology., vol. 35, no. 2, pp. 151-158, Nov. 2017.
[4] C. Studer, C. Benkeser, S. Belfanti, and Q. Huang “Design and Implementationof a Parallel Turbo-Decoder ASIC for 3GPP-LTE,” IEEE Journal of Solid-state Circuits., vol. 46, no. 1, pp. 8-17, Jan. 2011.
[5] M. Nader and J. Liu, “Design and implementation of CRC module of eCall in-vehicle system on FPGA,” SAE Technical 2015-01-2844, 2015,doi:10.4271/2015-01-2844.
[6] M. Nader and J. Liu. “Developing modulator and demodulator for the EU eCall in-vehicle system in FPGAs” in IEEE, 2016 International Conference on Computing, Networking and Communications (ICNC),Hawaii, USA, Feb. 15-18, 2016, pp. 1-5.
[7] M. Nader and J. Liu. “FPGA Design and Implementation of Demodulator/Decoder Module for the EU eCall In-Vehicle System” in International Conference on Embedded Systems and Applications (ESA’15), Las Vegas,USA, July 27-30, 2015, pp. 3-9.
[8] “Technical Specification Group Radio Access Network; Multiplexing and channel coding (FDD),” 3GPP, Tech. Rep. TS22.212.
[9] D. Viktor, K. Michal, and D. Milan“Impact of trellis termination on performance of turbo codes,” in ELEKTRO, 2016, pp.48-51.
[10] B. Muralikrishna, G.L. Madhumati, H. Khan, K.G. Deepika, “Reconfigurable system-on-chip design using FPGA,” in 2nd International Conference on Devices, Circuits and Systems (ICDCS), 2014, pp.1-6.
[11] J. J. Rodriguez-Andina, M. J. Moure, and M. D. Valdes, “Features, design tools, and application domains of FPGAs,” IEEE Trans. Ind. Electron., vol. 54, no. 4, pp. 18101823, Aug. 2007.

12.YUKTHI CERTIFICATE

INSTITUTION'S INNOVATION COUNCIL
MOE'S INNOVATION CELL

Institute Name:
Malla Reddy Institute of Technology & Science

Title of the Innovation/Prototype:
CHIP DESIGN FOR TURBO ENCODER MODULE FOR IN-VEHICLE SYSTEM

Team Lead Name:	Team Lead Email:	Team Lead Phone:	Team Lead Gender:
Sudheer Kumar Tokala	sudheerkumartokala@gmail.com	6309194291	Male

FY of Development:	Developed as part of:	Innovation Type:	TRL LEVEL:
2023-24	Academic Requirement/Study Project	Product	4

MRL Level:
MRL 1: Basic manufacturing implications identified

IRL Level:
IRL 1: Basic Research (Need Identification & Peer Review Publications) & Completed First-Pass Business Model Canvas (BMC)

Theme:
IoT based technologies (e.g. Security & Surveillance systems etc),

Define the problem and its relevance to today's market / sociaty / industry need:
The design of a turbo encoder module for in-vehicle systems faces several challenges. These include high computational complexity, which can lead to increased power consumption and latency, affecting real-time performance. Additionally, ensuring robustness against environmental factors like temperature fluctuations and electromagnetic interference is crucial for reliability. The need for integration with existing vehicle communication protocols complicates the design process. Furthermore, optimizing the trade-off between error correction capability and resource constraints, such as area and power, remains a critical issue. Addressing these challenges is essential for enhancing data transmission reliability in advanced automotive applications.

Describe the Solution / Proposed / Developed:
The proposed solution involves designing a highly efficient turbo encoder module using parallel processing and hardware acceleration techniques. By optimizing the encoder's architecture for low power consumption and reduced latency, we enhance real-time performance. Implementing adaptive error correction algorithms ensures robustness against varying environmental conditions. The design integrates seamlessly with existing vehicle communication protocols, facilitating easy deployment. Additionally, utilizing advanced simulation tools allows for thorough testing under diverse scenarios, ensuring reliability and performance. This approach balances error correction capability with resource efficiency, paving the way for enhanced data transmission in in-vehicle systems.

Explain the uniqueness and distinctive features of the (product / process / service) solution:
The proposed turbo encoder module stands out due to its innovative combination of parallel processing and adaptive algorithms, significantly enhancing throughput while minimizing power consumption. Its unique architecture is tailored for harsh automotive environments, ensuring reliable performance against temperature fluctuations and electromagnetic interference. Additionally, the module features a scalable design, allowing easy integration with various in-vehicle communication protocols. Advanced simulation techniques during development provide robust validation across diverse operational scenarios. This blend of efficiency, adaptability, and reliability positions the solution as a pioneering advancement in automotive data transmission technologies, setting it apart from conventional designs.

How your proposed / developed (product / process / service) solution is different from similiar kind of product by the competitors if any:
Our proposed turbo encoder module differentiates itself from competitors through its unique focus on low power consumption and high throughput achieved via advanced parallel processing techniques. Unlike traditional solutions, which often compromise performance for energy efficiency, our design maintains optimal error correction capabilities while minimizing latency. Additionally, our adaptive algorithms enhance robustness in varying environmental conditions, a feature not commonly prioritized by competitors. Furthermore, seamless integration with existing vehicle communication systems and extensive pre-deployment simulation set our solution apart, ensuring reliability and adaptability that are crucial for modern automotive applications.

Is there any IP or Patentable Component associated with the Solution?:
No

Has the Solution Received any Innovation Grant/Seefund Support?:
No

Are there any Recognitions (National/International) Obtained by the Solution?:
No

***Is the Solution Commercialized either through Technology Transfer or Enterprise Development/Startup?:**
No

Had the Solution Received any Pre-Incubation/Incubation Support?:
No

Video URL:
https://drive.google.com/file/d/1WJTxYgRW3zV0Cgik4tkXFh6-0Jh7N4rj/view?usp=sharing

Innovation Photograph:
View File