Introduction

Multipliers play a crucial role in microprocessors, digital signal processing (DSP) circuits, and various communication technologies. Traditional multiplication techniques require a significant number of adders to compute partial products, leading to increased hardware complexity, power consumption, and latency. As the demand for high-speed and energy-efficient processors grows, alternative multiplication methods have gained attention.

Vedic mathematics, an ancient Indian mathematical system, offers efficient computational techniques based on 16 fundamental sutras (formulas). One of these sutras, Urdhva Tiryagbhyam, provides a unique approach to multiplication that enables faster calculations with reduced hardware requirements. Leveraging this technique, approximate multipliers can be developed to optimize performance for specific applications, particularly in image processing, where minor errors are often tolerable.

This project introduces an 8×8 Hybrid Approximate Vedic Multiplier (HAVM), which integrates Vedic multiplication principles with approximation techniques. The primary objective is to achieve a balance between computational accuracy, power efficiency, and hardware optimization. The proposed HAVM employs NAND-based Approximate Half Adders (NHA) to minimize power consumption while maintaining acceptable computational accuracy. Unlike traditional multipliers, this hybrid approach reduces complexity by selectively processing only significant portions of input data, thereby lowering circuit area and energy usage.

The implementation of the HAVM is carried out using Verilog HDL and simulated using the Xilinx Vivado Design Suite. The design is further synthesized on a ZedBoard Zynq-7000 SoC to evaluate its real-time performance. Comparative analysis with conventional multipliers demonstrates that the HAVM achieves notable reductions in power and latency, making it highly suitable for energy-efficient applications such as image and audio processing.

By integrating Vedic mathematical principles with modern approximation techniques, this research paves the way for efficient hardware architectures tailored for low-power computing environments.

LITERATURE SURVEY

The pursuit of high-performance, low-power approximate multipliers for FPGA implementations is critical for various applications, particularly in energy-efficient computing. In this literature survey, we explore previous research efforts focusing on the development of efficient approximate multipliers tailored specifically for FPGA platforms.

1. Title: "Design of 4x4 Bit Vedic Multiplier Using EDA Tool”

Description: This study explores the efficiency of Vedic multipliers, particularly using the Urdhva Tiryagbhyam sutra. It demonstrates improved computational speed and area reduction compared to conventional multiplication techniques. However, it does not focus on power efficiency or approximation strategies. Year of Publication: 2012.

1. Title: "Design and Implementation of 8-Bit Vedic Multiplier"

Description: This research extends the Vedic multiplier to an 8-bit design, showcasing the benefits of carry-skip adders for partial product addition. While it enhances speed and area utilization, it does not employ approximation techniques to further optimize power consumption. Year of Publication: 2013.

1. Title: "Energy Efficient Approximate 8-Bit Vedic Multiplier"

Description: This study introduces approximate computing principles in Vedic multiplication to reduce power consumption. It strategically omits less significant bits, leading to minor computational errors while achieving significant energy savings. However, it does not explore hybrid approaches combining multiple multiplier architectures. Year of Publication: 2022.

1. Title: "Implementation of an Efficient N×N Multiplier Based on Vedic Mathematics and Booth-Wallace Tree Multiplier"

Description: This research integrates Vedic, Booth, and Wallace-tree multiplication algorithms, leveraging their strengths to optimize speed and area efficiency. While it reduces latency, it does not incorporate approximate adders to minimize power consumption further. Year of Publication: 2019.

1. Title: "High-Performance Approximate Half and Full Adder Cells Using NAND Logic Gates"

Description: This paper investigates NAND-based Approximate Half Adders (NHA), which enhance speed and reduce power usage in arithmetic circuits. The methodology is applied to approximate multipliers, but the study lacks an in-depth analysis of its impact on real-world FPGA implementations. Year of Publication: 2019.

The literature surrounding energy-efficient approximate multipliers for FPGA platforms has primarily focused on optimizing speed, area, and power consumption using various mathematical and hybrid approaches. However, there is a gap in designing hybrid approximate multipliers that effectively balance accuracy, power efficiency, and computational speed. This paper aims to fill this gap by proposing an 8×8 Hybrid Approximate Vedic Multiplier (HAVM) that integrates NHA- based approximation techniques while leveraging the efficiency of Vedic multiplication for real- time FPGA applications.

# EXISTING SYSTEM

Traditional multiplication methods play a crucial role in digital systems, particularly in microprocessors, Digital Signal Processing (DSP) circuits, and image processing applications. Conventional multipliers, such as Array Multipliers, Booth Multipliers, and Wallace Tree Multipliers, offer high accuracy but come with significant drawbacks, including high power consumption, increased latency, and large hardware area. These limitations pose challenges for energy-efficient computing, where low power consumption and faster computation speeds are essential.

One widely studied approach to improving multiplication efficiency is Vedic Mathematics, which offers fast and compact computational techniques. The Urdhva Tiryagbhyam sutra of Vedic multiplication is known for its ability to perform parallel computations, reducing the time required for multiplication. However, conventional Vedic multipliers still require a substantial number of full adders and half adders to accumulate partial products, leading to increased hardware complexity and higher power consumption.

To address these inefficiencies, Approximate Computing has emerged as a viable solution for applications where minor errors are tolerable, such as image and audio processing. Approximate multipliers introduce small errors in computations to reduce power consumption, latency, and circuit complexity. Existing approximate Vedic multipliers employ techniques like truncation, segmented computation, and carry-skip adders to optimize performance. However, these designs often lack a hybrid approach that balances speed, power, and accuracy efficiently.

Another existing technique involves Hybrid Multiplier Architectures, which combine multiple multiplication algorithms, such as Booth, Wallace, and Vedic multipliers, to leverage their individual strengths. While hybrid designs improve performance in certain areas, they may compromise accuracy or fail to integrate low-power design strategies effectively.

Thus, the existing system lacks an optimized hybrid multiplier that incorporates approximate computing principles, Vedic multiplication, and low-power adders to achieve an optimal trade- off between accuracy, speed, and energy efficiency.

## Understanding Vedic Sutras

1. The Origins of Vedic Mathematics:

Vedic Mathematics emerged as a mathematical methodology unveiled by the Indian mathematician Jagadguru Shri Bharathi Krishna Tirthaji in the early 20th century. Born in March 1884 in the village of Puri, Orissa, Shri Bharathi Krishna Tirthaji displayed exceptional proficiency in various subjects including mathematics, science, humanities, and Sanskrit language. His interests extended to spiritualism and meditation.

During a period of deep meditation near Sringeri, he purportedly rediscovered ancient mathematical principles known as Vedic sutras. Tirthaji claimed that these sutras, derived directly or indirectly from the Vedas, particularly the Rig-Veda, were spontaneously revealed to him after eight years of meditation. He later documented these sutras, but the original manuscripts were lost. In 1957, Tirthaji published an introductory volume containing 16 sutras titled "Vedic Mathematics" with plans to document additional sutras in the future. Unfortunately, he succumbed to illness, developing cataracts in both eyes, and passed away in 1960.

1. Understanding Vedic Sutras:

Vedic sutras are ancient mathematical principles that have been utilized effectively for a wide range of mathematical operations. These sutras have been applied to resolve various mathematical tasks such as factorization, finding highest common factors (HCF), divisions, calculating squares, reciprocals, square roots, cubes, and cube roots. Additionally, Vedic sutras have been employed to solve algebraic equations, quadratic equations, multiple simultaneous equations, biquadratic equations, cubic equations, and equations of higher degrees. They have also been utilized for tasks such as partial fractions, integration, differential calculus, and for proving geometric theorems including Pythagoras theorem, Apollonius Theorem, and analytical conics, among others.

1. Various Vedic Sutras include:
   * Urdhva-Tiryagbyham: Vertically and crosswise
   * Nikhilam Navatashcaramam Dashatah: All from 9 and the last from 10
   * Ekadhikena Purvena: By one more than the previous one
   * Paravartya Yojayet: Transpose and apply
   * Shunyam Saamyasamuccaye: When the total is the same, then that sum is zero
   * Anurupye Shunyamanyat: If one is in ratio, the other is zero
   * Sankalana-vyavakalanabhyam: By addition and by subtraction
   * Puranapuranabyham: Completion and non-completion
   * Chalana-Kalanabyham: Alterations and similarities
   * Yavadunam: Whatever the deficit
   * Vyashtisamasthi: Individual and total
   * Shesanyankena Charamena: The remainder by the last digit
   * Sopaantyadvayamantyam: The last and double the penultimate
   * Ekanyunena Purvena: By one less than the previous one
   * Gunitasamuchyah: The product of the sum is equal to the sum of the products
   * Gunakasamuchyah: The factors of the sum are equal to the sum of the factors.
2. Urdhva-Tiryagbyham: Vertically and crosswise:

Urdhva-Tiryagbyham, a fundamental principle in Vedic Mathematics, holds significant importance in the realm of mathematical computations. This ancient concept, originating from the Vedas, offers a unique approach to mathematical operations, particularly in multiplication.

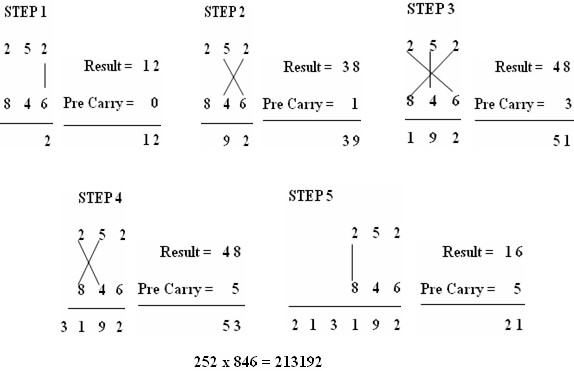
The term "Urdhva-Tiryagbyham" translates to "Vertically and Crosswise" in English, encapsulating the essence of its application. This principle enables individuals to multiply numbers efficiently by considering both vertical and diagonal components simultaneously.

At its core, Urdhva-Tiryagbyham relies on the concept of breaking down multiplication into simpler steps, thereby streamlining the process and reducing complexity. Rather than approaching multiplication in a traditional linear manner, this method encourages individuals to leverage vertical and diagonal relationships between digits to expedite calculations.

One of the key advantages of Urdhva-Tiryagbyham lies in its versatility and applicability across various multiplication scenarios. Whether multiplying two-digit, three-digit, or multi-digit numbers, this principle remains effective and efficient. By strategically organizing digits and identifying vertical and diagonal patterns, individuals can swiftly arrive at accurate multiplication results.

Explaining Urdhva-Tiryagbyham with an Example: Multiplication of 252 x 846

To demonstrate the Urdhva-Tiryagbyham method, let's consider the multiplication of two decimal numbers, 252 and 846. This method involves a systematic approach where digits on both sides of a vertical line are multiplied and then added with the carry from the previous step.



Here's how it works:

1. Start by aligning the two numbers vertically, with one number above the other, as shown in Fig. above.
2. Multiply the digits on each side of the vertical line and add them to any carry from the previous step. This yields one bit of the result and a carry for the next step.
3. Repeat this process for each digit pair, moving from right to left.
4. If multiple lines are present in one step, add all the results to the previous carry.
5. In each step, the least significant bit becomes the result bit, while all other bits act as carry for the next step.
6. Initially, the carry is assumed to be zero.

By following this method, we can efficiently compute the product of 252 and 846 while considering both vertical and diagonal relationships between digits. This approach simplifies the multiplication process and ensures accurate results, making it a valuable technique in mathematical computations.

## Basics of Approximate Multipliers

**Introduction to Approximate Computing**

Approximate computing is an emerging paradigm that seeks to improve computational efficiency by trading off accuracy for reduced power consumption, area, and delay. This approach is particularly useful in applications where perfect accuracy is not necessary, such as image processing, artificial intelligence, and multimedia applications. One of the most widely explored areas of approximate computing is approximate multiplication, which aims to optimize multiplication operations for energy-efficient processing.

**Need for Approximate Multipliers**

Multiplication is one of the most complex arithmetic operations in digital circuits, requiring multiple **adders, partial product generators, and large bit-wise computations**. In **Very Large-Scale Integration (VLSI) design**, multipliers significantly contribute to **power consumption, circuit delay, and area utilization**. Conventional multipliers, such as **Array Multipliers, Booth Multipliers, and Wallace Tree Multipliers**, provide high accuracy but suffer from high power and area overhead. To address these challenges, **approximate multipliers** introduce small, controlled errors in multiplication to achieve **lower power consumption, reduced delay, and compact hardware design**.

**Principles of Approximate Multipliers**

Approximate multipliers work on the principle of **simplifying the multiplication process** by either **reducing the number of operations, modifying the partial product addition, or selectively ignoring less significant bits**. The main strategies used in approximate multiplication include:

1. **Truncation-Based Approximate Multiplication:**
   * In conventional multipliers, **all bits of operands** contribute to the final result.
   * In truncation-based approaches, **least significant bits (LSBs) are ignored or approximated**, reducing the complexity of the addition process.
   * This method is highly effective for image and video processing applications where minor numerical errors do not impact overall performance.
2. **Segmented Multiplication:**
   * In this method, the multiplication operation is **divided into smaller segments**, where each segment is computed separately, and only the **most significant segments** contribute to the final result.
   * It reduces hardware complexity by eliminating unnecessary computations.
3. **Approximate Partial Product Generation:**
   * Instead of generating **exact partial products**, approximate multipliers use

**simplified logic circuits** to compute partial products with minor errors.

* + Techniques such as **AND gate reduction, modified Booth encoding, or inexact adders** are used to optimize energy efficiency.

1. **Use of Approximate Adders:**
   * Multipliers require multiple **half adders and full adders** to accumulate partial products.
   * Approximate adders, such as **NAND-based Approximate Half Adders (NHA)**, reduce computation complexity and power usage by **introducing small errors in sum and carry operations**.

**Types of Approximate Multipliers**

1. **Truncated Multipliers** – These multipliers eliminate lower-order bits to reduce power and area.
2. **Probabilistic Multipliers** – Operate based on probability-based logic to minimize computations.
3. **Hybrid Approximate Multipliers** – Combine different approximation techniques (e.g., Vedic, Wallace, and Booth) to optimize speed, power, and accuracy trade-offs.

***PROPOSED Hybrid Approximate Multiplier***

### 2×2 Bit Vedic Multiplier (VM)

**VM Architecture Based on Urdhva Tiryakbyham Sutra**

The **2×2 Vedic Multiplier (VM)** is a fundamental multiplication unit based on the **Urdhva Tiryakbyham sutra**, a key concept from ancient Vedic mathematics. This multiplier consists of **four AND gates**, making it a simple yet efficient approach to multiplication.

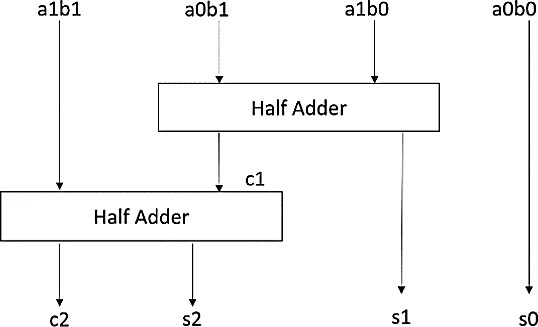
Exploring the **2×2 Vedic multiplier**, we observe how ancient mathematical principles can be effectively implemented in modern hardware design. Unlike conventional multiplication techniques, the Vedic approach enables **efficient computations with minimal logic gates** and streamlined calculations.

This **2×2 multiplier unit** serves as the foundation for constructing higher-bit multipliers, such as **4×4 and 8×8 multipliers**. Its modular design ensures scalability and efficiency in digital arithmetic operations.

**Operational Steps of the 2×2 Vedic Multiplier**

Consider two **2-bit binary numbers**, **A and B**, represented as:

* **A = a1a0**
* **B = b1b0**

****

Block Diagram of 2×2 bit VM

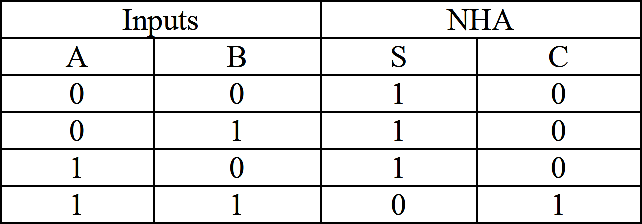
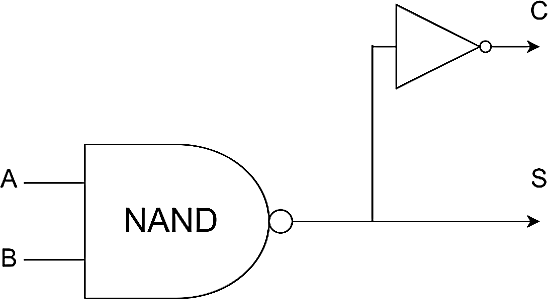
As illustrated in **Figure 1**, the multiplication process follows these steps:

* 1. Computing the Least Significant Bit (LSB):
     + Multiply the LSBs of A and B (a0 × b0) to obtain the LSB of the result.
  2. Intermediate Partial Products (PPs):
     + Multiply the LSB of B with the next higher bit of A (a1 × b0).
     + Multiply the LSB of A with the next higher bit of B (a0 × b1).
     + Add these intermediate products using a Half Adder (HA), generating a sum and a carry (carry 1).
  3. Computing the Most Significant Bit (MSB):
     + Multiply the MSBs of A and B (a1 × b1) to generate the final partial product.
     + Add this value to the previously computed carry, forming the final result.
  4. Final Result Formation:
     + The third bit of the result is obtained from the sum of the second HA.
     + The fourth bit (MSB) is the carry generated from the same HA operation.

The below equations represent the **operation of the 2×2 Vedic multiplier**, demonstrating its efficiency in performing multiplication with minimal logic components.

### NHA Cell

The NAND-based Approximate Half Adder (NHA) is designed to compute the Sum (S) and Carry (C), similar to a traditional Half Adder (HA). However, when utilizing a NAND gate for computation, the Sum output contains one incorrect result, whereas the Carry output remains accurate.

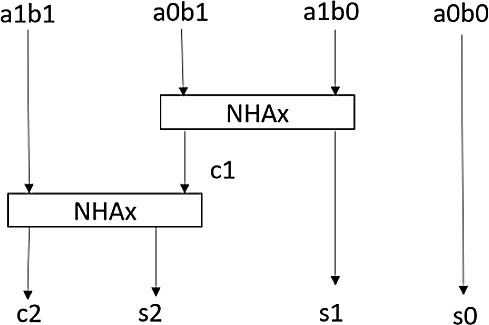


*Logic diagram of NHA Logic table of NAND based Approximate half adder*

One of the key advantages of the NHA cell is the reduction in Sum propagation delay, achieving a 28% improvement in latency compared to the previous best approximation. As observed in Table 1, when comparing the Exact Half Adder (HA) and NHAx, the Carry output remains unchanged, but the Sum output exhibits a single incorrect result.

1. **2×2 Bit Approximate Vedic Multiplier (AVM)**

For complete accuracy, traditional Vedic multipliers may require additional hardware resources (gates) due to their complexity. However, approximate versions aim to simplify the design by intentionally overlooking a small percentage of the multiplication process. This simplification reduces the complexity of the design, but the resulting output may have minor errors.



*Block Diagram of 2-bit AVM*

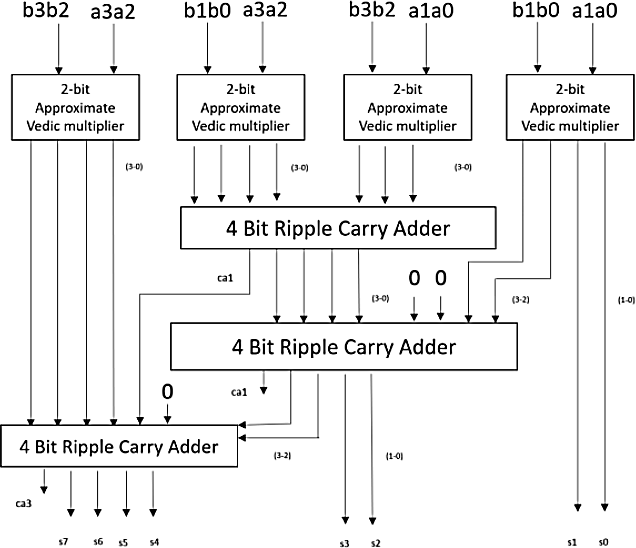
In certain cases, the least significant bits of one operand may have a negligible contribution to the final result. To implement this, the block diagram of the 2×2 AVM is shown in Figure. In this design, the traditional Half Adder (HA) is replaced with Non-Recursive Half Adder (NHA) cells. While the output of this module may contain some errors, these errors are typically tolerable in many practical applications.

1. **4×4 Bit Approximate Multiplier (AVM)**

The 4-bit operands are divided into upper and lower halves by the algorithm. These segments are then multiplied, including the cross products. To obtain the final output, the results are combined in the subsequent addition step, and any necessary carry propagation is carried out.

In the implementation of the 4-bit Approximate Multiplier (AVM), the 2-bit AVM serves as the foundation. Figure 4 illustrates the understanding of this module. As shown in the figure, **four 2- bit AVM modules** are used in the design of the 4-bit AVM. The addition operation of two 4-bit operands is handled by a **4-bit ripple carry adder**. This is achieved through the coordinated operation of four full adders.

Each full adder can add a carry from the previous addition as well as two binary digits. The carry bit produced by one adder ripples through to the next, forming a chain of strategically connected full adders. This design offers a **simple yet efficient method** for performing the addition operation.



*Block Diagram of 4-bit AVM*

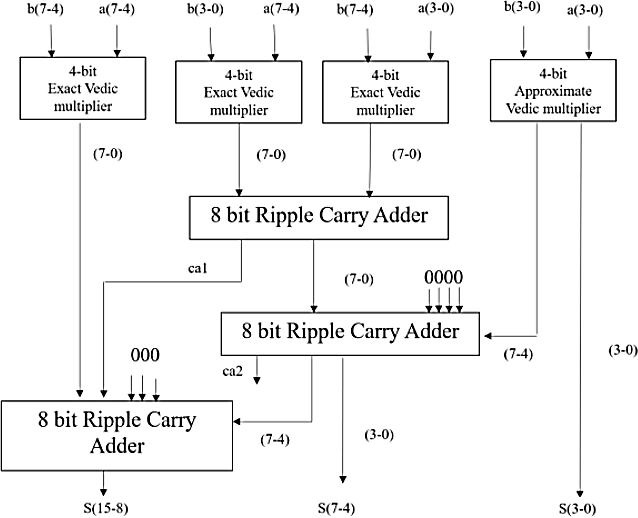
The outputs from this module, labeled **s0 to s7**, are generated accordingly. While the module produces the desired output, it also introduces some **tolerable errors**, with the least significant portions of the operand contributing less to the final result.

### 8×8 bit Hybrid Approximate Vedic Multiplier

To balance speed, power consumption, and accuracy, an **8×8 bit Hybrid Approximate Vedic Multiplier (HAVM)** combines the efficiency of Vedic multiplication with approximation techniques. Below is an overview of its key features:

**Vedic Multiplication Approach**

The multiplier leverages **Vedic algorithms**, which simplify multiplication by breaking it down into smaller, more manageable sub-processes. This method can lead to faster computation compared to traditional multiplication approaches. The design of the 8-bit Vedic multiplier (VM) involves utilizing four 4-bit VMs along with adder circuits. The output of this 8-bit VM is accurate, consumes a reasonable amount of energy, and experiences reduced delay.



***Block Diagram of 8×8 bit HAVM***

**Design of 8×8 bit Hybrid Approximate Vedic Multiplier**

The proposed **8×8 HAVM design** includes three 4-bit VM modules and one 4-bit Approximate Vedic Multiplier (AVM) module. The use of the 4-bit AVM on the least significant bits introduces some permissible errors in the output. However, this approximation method, along with the hybrid design, results in a **simpler circuit**, requiring fewer gates and consuming less power. The reduced complexity leads to more compact designs, potentially saving significant chip area.

**Error Tolerance and Application Areas**

Despite the approximation, the final output will inevitably have some inaccuracies. The degree of inaccuracy depends on the specific approximation techniques used. The **HAVM** is suitable for applications where a small margin of error is acceptable. For instance, in **digital signal processing (DSP)** tasks like **image and audio processing**, the error tolerance of the proposed multiplier is advantageous, making it an ideal choice for these applications.

## "Error Matrix Calculation for Approximate Circuits"

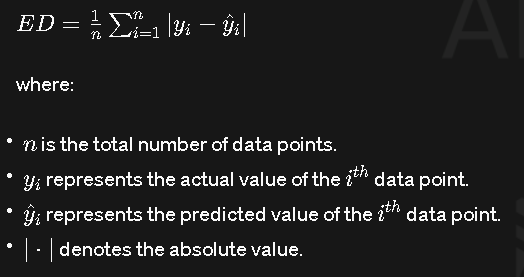
**Error Distance(ED) :**

Error Distance (ED), also known as Mean Absolute Error (MAE), is a metric used to quantify the average magnitude of errors between predicted values and actual values in a dataset. It provides insight into the overall accuracy of a predictive model or estimator.

The calculation of Error Distance involves the following steps:

* 1. **Compute the Error:** Begin by calculating the error for each data point in the dataset. The error is typically determined by taking the absolute difference between the predicted value and the corresponding actual value.
  2. **Sum the Absolute Errors:** Once you have computed the absolute error for each data point, sum up all these individual absolute errors.
  3. **Calculate the Mean:** After summing the absolute errors, compute the mean (average) of these absolute errors. This is achieved by dividing the sum of absolute errors by the total number of data points in the dataset.
  4. **Interpretation:** The resulting value of Error Distance represents the average magnitude of errors between predicted and actual values in the dataset. A lower Error Distance indicates that the model's predictions are closer to the actual values, suggesting better performance.

Mathematically, Error Distance (ED) or Mean Absolute Error (MAE) can be expressed as:

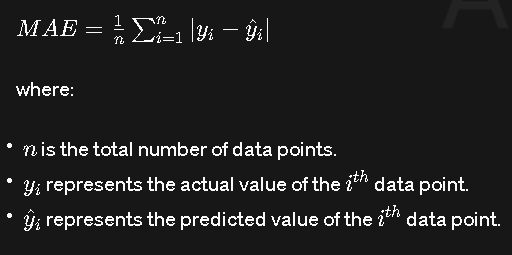


## MAE(Mean Absolute Error):

Mean Absolute Error (MAE) is a metric used to assess the average magnitude of errors between predicted and actual values in a dataset. It is widely utilized in various fields such as statistics, machine learning, and finance to evaluate the accuracy of predictive models or estimators.

The calculation of MAE involves the following steps:

1. Compute the Error: Begin by calculating the error for each data point in your dataset. The error is typically determined by subtracting the predicted value from the corresponding actual value.
2. Take the Absolute Value: After computing the error for each data point, take the absolute value of each individual error. This step ensures that all errors are treated as positive values, regardless of whether the prediction is higher or lower than the actual value.
3. Calculate the Mean: Once you have obtained the absolute errors for all data points, compute the mean (average) of these absolute errors. This is achieved by summing up all the absolute errors and then dividing by the total number of data points in the dataset.
4. Interpretation: The resulting value of MAE represents the average magnitude of errors between predicted and actual values in the dataset. Unlike MSE, which penalizes large errors more heavily due to squaring, MAE provides a more straightforward measure of the average discrepancy between predictions and observations.

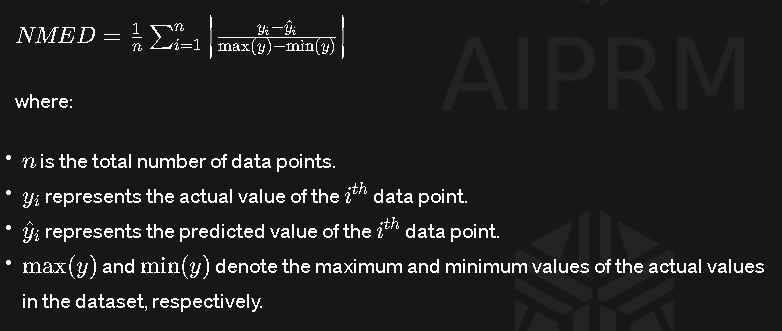


NMED (Normalized Error Distance): Normalized Error Distance (NMED) is a metric used to evaluate the accuracy of a predictive model by measuring the normalized average distance between the predicted values and the actual values in a dataset. NMED is particularly useful when comparing models across different datasets or when the scale of the data varies significantly.

The calculation of NMED involves the following steps:

1. Compute the Error: Begin by calculating the error for each data point in your dataset. The error is typically determined by subtracting the predicted value from the corresponding actual value.
2. Normalize the Errors: Next, normalize the errors by dividing each error by the range of the actual values in the dataset. This normalization step ensures that the NMED is independent of the scale of the data.
3. Calculate the Mean: After normalizing the errors, compute the mean (average) of these normalized errors. This is achieved by summing up all the normalized errors and then dividing by the total number of data points in the dataset.
4. Interpretation: The resulting value of NMED represents the average normalized distance between the predicted and actual values in the dataset. It provides a measure of the average discrepancy between the predictions made by a model and the observed data points, relative to the range of the actual values. A lower NMED indicates that the model's predictions are closer to the actual values, suggesting better performance.

Mathematically, NMED can be expressed as:



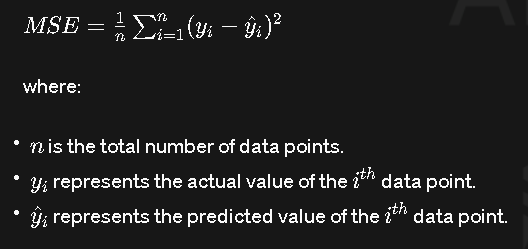
## MSE(Mean Squared Error) :

Mean Squared Error (MSE) is a measure used to evaluate the average squared difference between the predicted values and the actual values in a dataset. It is commonly employed in various fields such as statistics, machine learning, and signal processing to assess the performance of predictive models or estimators.

The calculation of MSE involves several steps:

1. Compute the Error: Begin by calculating the error for each data point in your dataset. The error is typically determined by subtracting the predicted value from the corresponding actual value.
2. Square the Errors: Once you have computed the error for each data point, square each of these individual errors. Squaring the errors ensures that all errors contribute positively to the final MSE value, preventing cancellation of positive and negative errors.
3. Calculate the Mean: After squaring the errors, compute the mean (average) of these squared errors. This is achieved by summing up all the squared errors and then dividing by the total number of data points in the dataset.
4. Interpretation: The resulting value of MSE represents the average of the squared differences between the predicted and actual values in the dataset. It provides a measure of the average discrepancy between the predictions made by a model and the observed data points. A lower MSE indicates that the model's predictions are closer to the actual values, suggesting better performance.

Mathematically, MSE can be expressed as:



VLSI

INTRODUCTION

What is VLSI?

VLSI stands for "Very Large Scale Integration". This is the field which involves packing more and more logic devices into smaller and smaller areas. Thanks to VLSI, circuits that would have taken boardfuls of space can now be put into a small space few millimetres across! This has opened up a big opportunity to do things that were not possible before. VLSI circuits are everywhere ... your computer, your car, your brand new state-of-the-art digital camera, the cell-phones, and what have you.

VLSI has been around for a long time, there is nothing new about it ... but as a side effect of advances in the world of computers, there has been a dramatic proliferation of tools that can be used to design VLSI circuits. Alongside, obeying Moore's law, the capability of an IC has increased exponentially over the years, in terms of computation power, utilisation of available area, yield. The combined effect of these two advances is that people can now put diverse functionality into the IC's, opening up new frontiers. Examples are embedded systems, where intelligent devices are put inside everyday objects, and ubiquitous computing where small computing devices proliferate to such an extent that even the shoes you wear may actually do something useful like monitoring your heartbeats.

DEALING WITH VLSI CIRCUITS

Digital VLSI circuits are predominantly CMOS based. The way normal blocks like latches and gates are implemented is different from what students have seen so far, but the behaviour remains the same. All the miniaturisation involves new things to consider. A lot of thought has to go into actual implementations as well as design. Let us look at some of the factors involved ...

1. Circuit Delays. Large complicated circuits running at very high frequencies have one big problem to tackle - the problem of delays in propagation of signals through gates and wires

... even for areas a few micrometers across! The operation speed is so large that as the delays add up, they can actually become comparable to the clockspeeds.

1. Power. Another effect of high operation frequencies is increased consumption of power. This has two-fold effect - devices consume batteries faster, and heat dissipation increases. Coupled with the fact that surface areas have decreased, heat poses a major threat to the stability of the circuit itself.
2. Layout. Laying out the circuit components is task common to all branches of electronics. Whats so special in our case is that there are many possible ways to do this; there can be multiple layers of different materials on the same silicon, there can be different arrangements of the smaller parts for the same component and so on.

The power dissipation and speed in a circuit present a trade-off; if we try to optimise on one, the other is affected. The choice between the two is determined by the way we chose the layout the circuit components. Layout can also affect the fabrication of VLSI chips, making it either easy or difficult to implement the components on the silicon.

THE VLSI DESIGN PROCESS

A typical digital design flow is as follows:

1.Specification 2.Architecture 3.RTLCoding 4.RTLVerification 5.Synthesis 6.Backend

7. Tape Out to Foundry to get end product….a wafer with repeated number of identical Ics.

All modern digital designs start with a designer writing a hardware description of the IC (using HDL or Hardware Description Language) in Verilog/VHDL. A Verilog or VHDL program essentially describes the hardware (logic gates, Flip-Flops, counters etc) and the interconnect of the circuit blocks and the functionality. Various CAD tools are available to synthesize a circuit based on the HDL. The most widely used synthesis tools come from two CADcompanies. Synposys and Cadence.

Without going into details, we can say that the VHDL, can be called as the "C" of the VLSI industry. VHDL stands for "VHSIC Hardware Definition Language", where VHSIC stands for "Very High Speed Integrated Circuit". This languages is used to design the circuits at a high- level, in two ways. It can either be a behavioural description, which describes what the circuit is supposed to do, or a structural description, which describes what the circuit is made of. There are other languages for describing circuits, such as Verilog, which work in a similar fashion.

Both forms of description are then used to generate a very low-level description that actually spells out how all this is to be fabricated on the silicon chips. This will result in the manufacture of the intended IC.

A typical analog design flow is as follows:

In case of analog design, the flow changes somewhat.

1. Specifications
2. Architecture
3. Circuit Design
4. SPICE Simulation
5. Layout
6. Parametric Extraction/Back Annotation
7. Final Design
8. Tape Out to foundry.

While digital design is highly automated now, very small portion of analog design can be automated. There is a hardware description language called AHDL but is not widely used as it does not accurately give us the behavioral model of the circuit because of the complexity of the effects of parasitic on the analog behavior of the circuit. Many analog chips are what are termed as “flat” or non-hierarchical designs. This is true for small transistor count chips such as an operational amplifier, or a filter or a power management chip. For more complex analog chips such as data converters, the design is done at a transistor level, building up to a cell level, then a block level and then integrated at a chip level. Not many CAD tools are available for analog design even today and thus analog design remains a difficult art. SPICE remains the most useful simulation tool for analog as well as digital design.

MOST OF TODAY’S VLSI DESIGNS ARE CLASSIFIED INTO THREE CATEGORIES:

1. Analog:

Small transistor count precision circuits such as Amplifiers, Data converters, filters, Phase Locked Loops, Sensors etc.

1. ASICS or Application Specific Integrated Circuits:

Progress in the fabrication of IC's has enabled us to create fast and powerful circuits in smaller and smaller devices. This also means that we can pack a lot more of functionality into the same area. The biggest application of this ability is found in the design of ASIC's. These are IC's that are created for specific purposes - each device is created to do a particular job, and do it well. The most common application area for this is DSP - signal filters, image compression, etc. To go to extremes, consider the fact that the digital wristwatch normally consists of a single IC doing all the time-keeping jobs as well as extra features like games, calendar, etc.

1. SoC or Systems on a chip:

These are highly complex mixed signal circuits (digital and analog all on the same chip). A network processor chip or a wireless radio chip is an example of an SoC.

# CHAPTER 1 INTRODUCTION TO VLSI

#### VLSI TECHNOLOGY

Gone are the days when huge computers made of vacuum tubes sat humming in entire dedicated rooms and could do about 360 multiplications of 10 digit numbers in a second. Though they were heralded as the fastest computing machines of that time, they surely don’t stand a chance when compared to the modern day machines. Modern day computers are getting smaller, faster, and cheaper and more power efficient every progressing second. But what drove this change? The whole domain of computing ushered into a new dawn of electronic miniaturization with the advent of semiconductor transistor by Bardeen (1947-48) and then the Bipolar Transistor by Shockley (1949) in the Bell Laboratory.

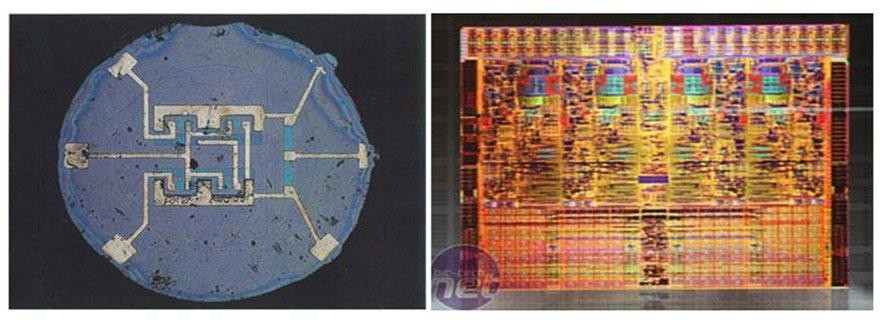


Fig: A comparison: first planar IC(1961) and Intel nehalem quad core die

Since the invention of the first IC (Integrated Circuit) in the form of a Flip Flop by Jack Kilby in 1958, our ability to pack more and more transistors onto a single chip has doubled roughly every 18 months, in accordance with the Moore’s Law. Such exponential development had never been seen in any other field and it still continues to be a major area of research work.

#### VLSI

**Very-large-scale integration** (**VLSI**) is the process of creating an [integrated](http://en.wikipedia.org/wiki/Integrated_circuit)

[circuit](http://en.wikipedia.org/wiki/Integrated_circuit) (IC) by combining thousands of [transistors](http://en.wikipedia.org/wiki/Transistors) into a single chip. VLSI began in the 1970s when complex [semiconductor](http://en.wikipedia.org/wiki/Semiconductor) and [communication](http://en.wikipedia.org/wiki/Communication) technologies were being developed. The [microprocessor](http://en.wikipedia.org/wiki/Microprocessor) is a VLSI device. Before the introduction of VLSI technology most ICs had a limited set of functions they could perform. An [electronic circuit](http://en.wikipedia.org/wiki/Electronic_circuit) might consist of a [CPU](http://en.wikipedia.org/wiki/Central_processing_unit), [ROM](http://en.wikipedia.org/wiki/Read-only_memory), [RAM](http://en.wikipedia.org/wiki/Random_Access_Memory) and other [glue logic](http://en.wikipedia.org/wiki/Glue_logic). VLSI lets IC designers add all of these into one chip.

#### HISTORY

During the mid-1920s, several inventors attempted devices that were intended to control current in solid-state diodes and convert them into triodes. Success did not come until after WWII, during which the attempt to improve silicon and germanium crystals for use as radar detectors led to improvements in fabrication and in the understanding of quantum mechanical states of carriers in semiconductors. Then scientists who had been diverted to radar development returned to solid-state device development. With the invention of transistors at Bell Labs in 1947, the field of electronics shifted from vacuum tubes to solid- state devices.

With the small transistor at their hands, electrical engineers of the 1950s saw the possibilities of constructing far more advanced circuits. As the complexity of circuits grew, problems arose.

One problem was the size of the circuit. A complex circuit, like a computer, was dependent on speed. If the components of the computer were too large or the wires interconnecting them too long, the electric signals couldn't travel fast enough through the circuit, thus making the computer too slow to be effective.[[1]](http://en.wikipedia.org/wiki/Very-large-scale_integration#cite_note-The_History_of_the_Integrated_Circuit-1)

[Jack Kilby](http://en.wikipedia.org/wiki/Jack_Kilby) at [Texas Instruments](http://en.wikipedia.org/wiki/Texas_Instruments) found a solution to this problem in 1958. Kilby's idea was to make all the components and the chip out of the same block (monolith) of semiconductor material. Kilby presented his idea to his superiors, and was allowed to build a test version of his circuit. In September 1958, he had his first integrated circuit ready. Although the first integrated circuit was crude and had some problems, the idea was groundbreaking. By making all the parts out of the same block of material and adding the

metal needed to connect them as a layer on top of it, there was no need for discrete components. No more wires and components had to be assembled manually. The circuits could be made smaller, and the manufacturing process could be automated. From here, the idea of integrating all components on a single silicon wafer came into existence, which led to development in small-scale integration (SSI) in the early 1960s, medium-scale integration (MSI) in the late 1960s, and then large-scale integration (LSI) as well as VLSI in the 1970s and 1980s, with tens of thousands of transistors on a single chip (later hundreds of thousands, then millions, and now billions (109)).

#### INTEGRATED CIRCUITS

An **integrated circuit** or **monolithic integrated circuit** (also referred to as an **IC**, a **chip**, or a **microchip**) is a set of [electronic circuits](http://en.wikipedia.org/wiki/Electronic_circuit) on one small plate ("chip") of [semiconductor material](http://en.wikipedia.org/wiki/Semiconductor_material), normally [silicon](http://en.wikipedia.org/wiki/Silicon). This can be made much smaller than a [discrete](http://en.wikipedia.org/wiki/Discrete_circuit) [circuit](http://en.wikipedia.org/wiki/Discrete_circuit) made from independent [electronic components](http://en.wikipedia.org/wiki/Electronic_component). ICs can be made very compact, having up to several billion [transistors](http://en.wikipedia.org/wiki/Transistor) and other [electronic components](http://en.wikipedia.org/wiki/Electronic_component) in an area the size of a fingernail. The width of each conducting line in a circuit can be made smaller and smaller as the technology advances; in 2008 it dropped below 100 [nanometers](http://en.wikipedia.org/wiki/Nanometer), and now is tens of nanometers.

ICs were made possible by experimental discoveries showing that [semiconductor](http://en.wikipedia.org/wiki/Semiconductor_device) [devices](http://en.wikipedia.org/wiki/Semiconductor_device) could perform the functions of [vacuum tubes](http://en.wikipedia.org/wiki/Vacuum_tube)and by mid-20th-century technology advancements in [semiconductor device fabrication](http://en.wikipedia.org/wiki/Semiconductor_device_fabrication). The integration of large numbers of tiny[transistors](http://en.wikipedia.org/wiki/Transistor) into a small chip was an enormous improvement over the manual assembly of circuits using discrete [electronic components](http://en.wikipedia.org/wiki/Electronic_component). The integrated circuit's [mass](http://en.wikipedia.org/wiki/Mass_production) [production](http://en.wikipedia.org/wiki/Mass_production) capability, reliability and building-block approach to [circuit design](http://en.wikipedia.org/wiki/Integrated_circuit_design) ensured the rapid adoption of standardized integrated circuits in place of designs using discrete transistors.

ICs have two main advantages over [discrete circuits](http://en.wikipedia.org/wiki/Discrete_circuit): cost and performance. Cost is low because the chips, with all their components, are printed as a unit by [photolithography](http://en.wikipedia.org/wiki/Photolithography) rather than being constructed one transistor at a time. Furthermore, packaged ICs use much less material than discrete circuits. Performance is high because the

IC's components switch quickly and consume little power (compared to their discrete counterparts) as a result of the small size and close proximity of the components. As of 2012, typical chip areas range from a few square millimeters to around 450 mm2, with up to 9 million [transistors](http://en.wikipedia.org/wiki/Transistor) per mm2.

Integrated circuits are used in virtually all electronic equipment today and have revolutionized the world of [electronics](http://en.wikipedia.org/wiki/Electronics). [Computers](http://en.wikipedia.org/wiki/Computer), [mobile phones](http://en.wikipedia.org/wiki/Mobile_phone), and other digital [home](http://en.wikipedia.org/wiki/Home_appliance) [appliances](http://en.wikipedia.org/wiki/Home_appliance) are now inextricable parts of the structure of modern societies, made possible by the low cost of integrated circuits.

#### DEVELOPMENTS

The first semiconductor chips held two transistors each. Subsequent advances added 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](http://en.wikipedia.org/wiki/Diode), [transistors](http://en.wikipedia.org/wiki/Transistor), [resistors](http://en.wikipedia.org/wiki/Resistor) and [capacitors](http://en.wikipedia.org/wiki/Capacitor), making it possible to fabricate one or more [logic gates](http://en.wikipedia.org/wiki/Logic_gate) on a single device. Now known retrospectively as [*small-scale*](http://en.wikipedia.org/wiki/Integrated_circuit#SSI.2C_MSI_and_LSI)[*integration*](http://en.wikipedia.org/wiki/Integrated_circuit#SSI.2C_MSI_and_LSI)(SSI), improvements in technique led to devices with hundreds of logic gates, known as [*medium-scale integration*](http://en.wikipedia.org/wiki/Integrated_circuit#SSI.2C_MSI_and_LSI)(MSI). Further improvements led to [*large-scale*](http://en.wikipedia.org/wiki/Integrated_circuit#SSI.2C_MSI_and_LSI)[*integration*](http://en.wikipedia.org/wiki/Integrated_circuit#SSI.2C_MSI_and_LSI)(LSI), i.e. systems with at least a thousand logic gates. Current technology has moved far past this mark and today's [microprocessors](http://en.wikipedia.org/wiki/Microprocessor) have many millions of gates and billions 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*](http://en.wikipedia.org/wiki/Integrated_circuit#ULSI.2C_WSI.2C_SOC_and_3D-IC)(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.

As of early 2008, billion-transistor processors are commercially available. This became more commonplace as semiconductor fabrication advanced from the then-current generation of [65 nm](http://en.wikipedia.org/wiki/65_nanometer) processes. Current designs, unlike the earliest devices, use extensive [design automation](http://en.wikipedia.org/wiki/Electronic_Design_Automation) and automated [logic synthesis](http://en.wikipedia.org/wiki/Logic_synthesis) to [lay out](http://en.wikipedia.org/wiki/Integrated_circuit_layout) the transistors, enabling higher levels of complexity in the resulting logic functionality. Certain high-performance logic blocks like the SRAM ([static random-access memory](http://en.wikipedia.org/wiki/Static_random-access_memory)) cell, are still designed by hand to

ensure the highest efficiency. VLSI technology may be moving toward further radical miniaturization with introduction of [NEMS](http://en.wikipedia.org/wiki/Nanoelectromechanical_systems) technology.

#### GENERATIONS

In the early days of simple integrated circuits, the technology's large scale limited each chip to only a few transistors, and the low degree of integration meant the design process was relatively simple. Manufacturing yields were also quite low by today's standards. As the technology progressed, millions, then billions of transistors could be placed on one chip, and good designs required thorough planning, giving rise to new [design methods](http://en.wikipedia.org/wiki/Y_diagram).

**SSI, MSI, LSI & VLSI**

The first integrated circuits contained only a few transistors. Called "small-scale integration" (SSI), digital circuits containing transistors numbering in the tens provided a few logic gates for example, while early linear ICs such as the [Plessey](http://en.wikipedia.org/wiki/Plessey) SL201 or the [Philips](http://en.wikipedia.org/wiki/Philips) TAA320 had as few as two transistors. The term Large Scale Integration was first used by [IBM](http://en.wikipedia.org/wiki/IBM) scientist [Rolf Landauer](http://en.wikipedia.org/wiki/Rolf_Landauer) when describing the theoretical concept from there came the terms for SSI, MSI, VLSI, and ULSI.

SSI circuits were crucial to early aerospace projects, and aerospace projects helped inspire development of the technology. Both the [Minuteman missile](http://en.wikipedia.org/wiki/Minuteman_missile) and [Apollo program](http://en.wikipedia.org/wiki/Apollo_program) needed lightweight digital computers for their inertial guidance systems; the [Apollo guidance](http://en.wikipedia.org/wiki/Apollo_guidance_computer) [computer](http://en.wikipedia.org/wiki/Apollo_guidance_computer) led and motivated the integrated-circuit technology, while the Minuteman missile forced it into mass-production. The Minuteman missile program and various other Navy programs accounted for the total $4 million integrated circuit market in 1962, and by 1968,

U.S. Government space and defense spending still accounted for 37% of the $312 million total production. The demand by the U.S. Government supported the nascent integrated circuit market until costs fell enough to allow firms to penetrate the industrial and eventually the consumer markets. The average price per integrated circuit dropped from $50.00 in 1962 to $2.33 in 1968. Integrated circuits began to appear in consumer products by the turn of the decade, a typical application being [FM](http://en.wikipedia.org/wiki/Frequency_modulation) inter-carrier sound processing in television receivers.

The next step in the development of integrated circuits, taken in the late 1960s, introduced devices which contained hundreds of transistors on each chip, called "medium- scale integration" (MSI).

They were attractive economically because while they cost little more to produce than SSI devices, they allowed more complex systems to be produced using smaller circuit boards, less assembly work (because of fewer separate components), and a number of other advantages.

Further development, driven by the same economic factors, led to "large-scale integration" (LSI) in the mid-1970s, with tens of thousands of transistors per chip.

Integrated circuits such as 1K-bit RAMs, calculator chips, and the first microprocessors, that began to be manufactured in moderate quantities in the early 1970s, had under 4000 transistors. True LSI circuits, approaching 10,000 transistors, began to be produced around 1974, for computer main memories and second-generation microprocessors.

The final step in the development process, starting in the 1980s and continuing through the present, was "very large-scale integration" ([VLSI](http://en.wikipedia.org/wiki/VLSI)). The development started with hundreds of thousands of transistors in the early 1980s, and continues beyond several billion transistors as of 2009.

Multiple developments were required to achieve this increased density. Manufacturers moved to smaller design rules and cleaner fabrication facilities, so that they could make chips with more transistors and maintain adequate yield. The path of process improvements was summarized by the [International Technology Roadmap for Semiconductors](http://en.wikipedia.org/wiki/International_Technology_Roadmap_for_Semiconductors) (ITRS). [Design](http://en.wikipedia.org/wiki/Electronic_design_automation) [tools](http://en.wikipedia.org/wiki/Electronic_design_automation) improved enough to make it practical to finish these designs in a reasonable time. The more energy efficient [CMOS](http://en.wikipedia.org/wiki/CMOS) replaced [NMOS](http://en.wikipedia.org/wiki/NMOS_logic) and [PMOS](http://en.wikipedia.org/wiki/PMOS_logic), avoiding a prohibitive increase in power consumption.

In 1986 the first one megabit [RAM](http://en.wikipedia.org/wiki/Random_Access_Memory) chips were introduced, containing more than one million transistors. Microprocessor chips passed the million transistor mark in 1989 and the billion transistor mark in 2005. The trend continues largely unabated, with chips introduced in 2007 containing tens of billions of memory transistors.

#### ULSI, WSI, SOC, 3D-IC

To reflect further growth of the complexity, the term *ULSI* that stands for "ultra-large- scale integration" was proposed for chips of more than 1 million transistors.

[Wafer-scale integration](http://en.wikipedia.org/wiki/Wafer-scale_integration) (WSI) is a means of building very large integrated circuits that uses an entire silicon wafer to produce a single "super-chip". Through a combination of

large size and reduced packaging, WSI could lead to dramatically reduced costs for some systems, notably massively parallel supercomputers. The name is taken from the term Very- Large-Scale Integration, the current state of the art when WSI was being developed.

A [system-on-a-chip](http://en.wikipedia.org/wiki/System-on-a-chip) (SoC or SOC) is an integrated circuit in which all the components needed for a computer or other system are included on a single chip. The design of such a device can be complex and costly, and building disparate components on a single piece of silicon may compromise the efficiency of some elements. However, these drawbacks are offset by lower manufacturing and assembly costs and by a greatly reduced power budget: because signals among the components are kept on-die, much less power is required (see [Packaging](http://en.wikipedia.org/wiki/Integrated_circuit#Packaging)).

A [three-dimensional integrated circuit](http://en.wikipedia.org/wiki/Three-dimensional_integrated_circuit) (3D-IC) has two or more layers of active electronic components that are integrated both vertically and horizontally into a single circuit. Communication between layers uses on-die signaling, so power consumption is much lower than in equivalent separate circuits. Judicious use of short vertical wires can substantially reduce overall wire length for faster operation.

#### STRUCTURED DESIGN

Structured VLSI design is a modular methodology originated by [Carver](http://en.wikipedia.org/wiki/Carver_Mead) [Mead](http://en.wikipedia.org/wiki/Carver_Mead) and [Lynn Conway](http://en.wikipedia.org/wiki/Lynn_Conway) for saving microchip area by minimizing the interconnect fabrics area. This is obtained by repetitive arrangement of rectangular macro blocks which can be interconnected using [wiring by abutment](http://en.wikipedia.org/w/index.php?title=Wiring_by_abutment&action=edit&redlink=1). An example is partitioning the layout of an adder into a row of equal bit slices cells. In complex designs this structuring may be achieved by hierarchical nesting.

Structured VLSI design had been popular in the early 1980s, but lost its popularity later because of the advent of [placement and routing](http://en.wikipedia.org/wiki/Place_and_route) tools wasting a lot of area by [routing](http://en.wikipedia.org/wiki/Routing_(EDA)), which is tolerated because of the progress of [Moore's Law](http://en.wikipedia.org/wiki/Moore%27s_Law). When introducing the [hardware](http://en.wikipedia.org/wiki/Hardware_description_language) [description language](http://en.wikipedia.org/wiki/Hardware_description_language) KARL in the mid' 1970s, [Reiner Hartenstein](http://en.wikipedia.org/w/index.php?title=Reiner_Hartenstein&action=edit&redlink=1) coined the term "structured VLSI design" (originally as "structured LSI design"), echoing [Edsger Dijkstra](http://en.wikipedia.org/wiki/Edsger_Dijkstra)'s [structured](http://en.wikipedia.org/wiki/Structured_programming) [programming](http://en.wikipedia.org/wiki/Structured_programming) approach by procedure nesting to avoid chaotic *spaghetti-structured* programs.

#### CHALLENGES

As microprocessors become more complex due to [technology scaling](http://en.wikipedia.org/wiki/Moore%27s_law), microprocessor designers have encountered several challenges which force them to think beyond the design plane, and look ahead to post-silicon:

* **Process variation** – As [photolithography](http://en.wikipedia.org/wiki/Photolithography) techniques tend closer to the fundamental laws of optics, achieving high accuracy in [doping](http://en.wikipedia.org/wiki/Dopant) concentrations and etched wires is becoming more difficult and prone to errors due to variation. Designers now must simulate across multiple fabrication [process corners](http://en.wikipedia.org/wiki/Process_corners) before a chip is certified ready for production.
* **Stricter design rules** – Due to lithography and etch issues with scaling, [design](http://en.wikipedia.org/wiki/Design_rule_checking) [rules](http://en.wikipedia.org/wiki/Design_rule_checking) for [layout](http://en.wikipedia.org/wiki/Integrated_circuit_layout) have become increasingly stringent. Designers must keep ever more of these rules in mind while laying out custom circuits. The overhead for custom design is now reaching a tipping point, with many design houses opting to switch to [electronic](http://en.wikipedia.org/wiki/Electronic_design_automation) [design automation](http://en.wikipedia.org/wiki/Electronic_design_automation) (EDA) tools to automate their design process.
* [**Timing/design closure**](http://en.wikipedia.org/wiki/Design_closure)– As [clock frequencies](http://en.wikipedia.org/wiki/Clock_frequency) tend to scale up, designers are finding it more difficult to distribute and maintain low [clock skew](http://en.wikipedia.org/wiki/Clock_skew) between these high frequency clocks across the entire chip. This has led to a rising interest in [multicore](http://en.wikipedia.org/wiki/Multi-core_processor) and [multiprocessor](http://en.wikipedia.org/wiki/Multiprocessor) architectures, since an [overall speedup](http://en.wikipedia.org/wiki/Amdahl%27s_law) can be obtained by lowering the clock frequency and distributing processing.
* **First-pass success** – As [die](http://en.wikipedia.org/wiki/Die_(integrated_circuit)) sizes shrink (due to scaling), and [wafer](http://en.wikipedia.org/wiki/Wafer_(electronics)) sizes go up (due to lower manufacturing costs), the number of dies per wafer increases, and the complexity of making suitable [photo masks](http://en.wikipedia.org/wiki/Photomask) goes up rapidly. A [mask set](http://en.wikipedia.org/wiki/Mask_set) for a modern technology can cost several million dollars. This non-recurring expense deters the old iterative philosophy involving several "spin-cycles" to find errors in silicon, and encourages first- pass silicon success. Several design philosophies have been developed to aid this new design flow, including design for manufacturing ([DFM](http://en.wikipedia.org/wiki/Design_for_manufacturability_(IC))), design for test ([DFT](http://en.wikipedia.org/wiki/Design_for_Test)).

**FPGA design flow using with Vivado® IDE**

**Step 1: Creating a Project** **Launch Vivado**

1. To launch Vivado, do the following: On Linux:

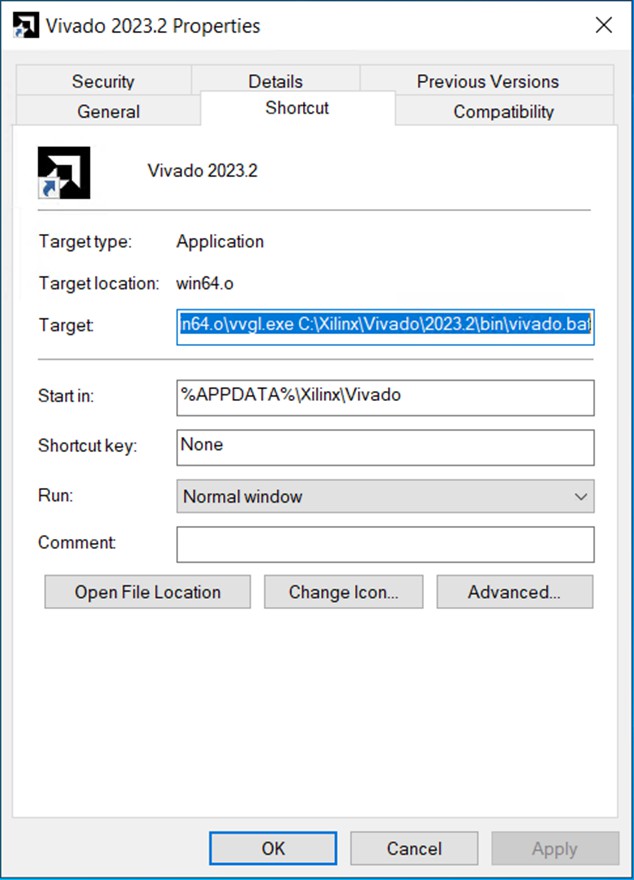
o

* 1. Change to the directory where the lab materials are stored. cd <Extract\_Dir>/Vivado\_Tutorial
  2. Launch the Vivado IDE.

vivado

* + On Windows:
    1. Before clicking the desktop icon to launch the Vivado tool, configure the icon to indicate where to write the vivado.log and vivado.jou files.
    2. Right-click the Vivado <version> Desktop icon and select Properties from the popup menu.
    3. Under the Shortcut tab, set the Start in value to the extracted Vivado Tutorial directory, as shown in the following figure:

<Extract\_Dir>/Vivado\_Tutorial/

* + 1. Click OK to close the Properties dialog box.
    2. Double-click the Vivado <version> Desktop icon to start the Vivado IDE.

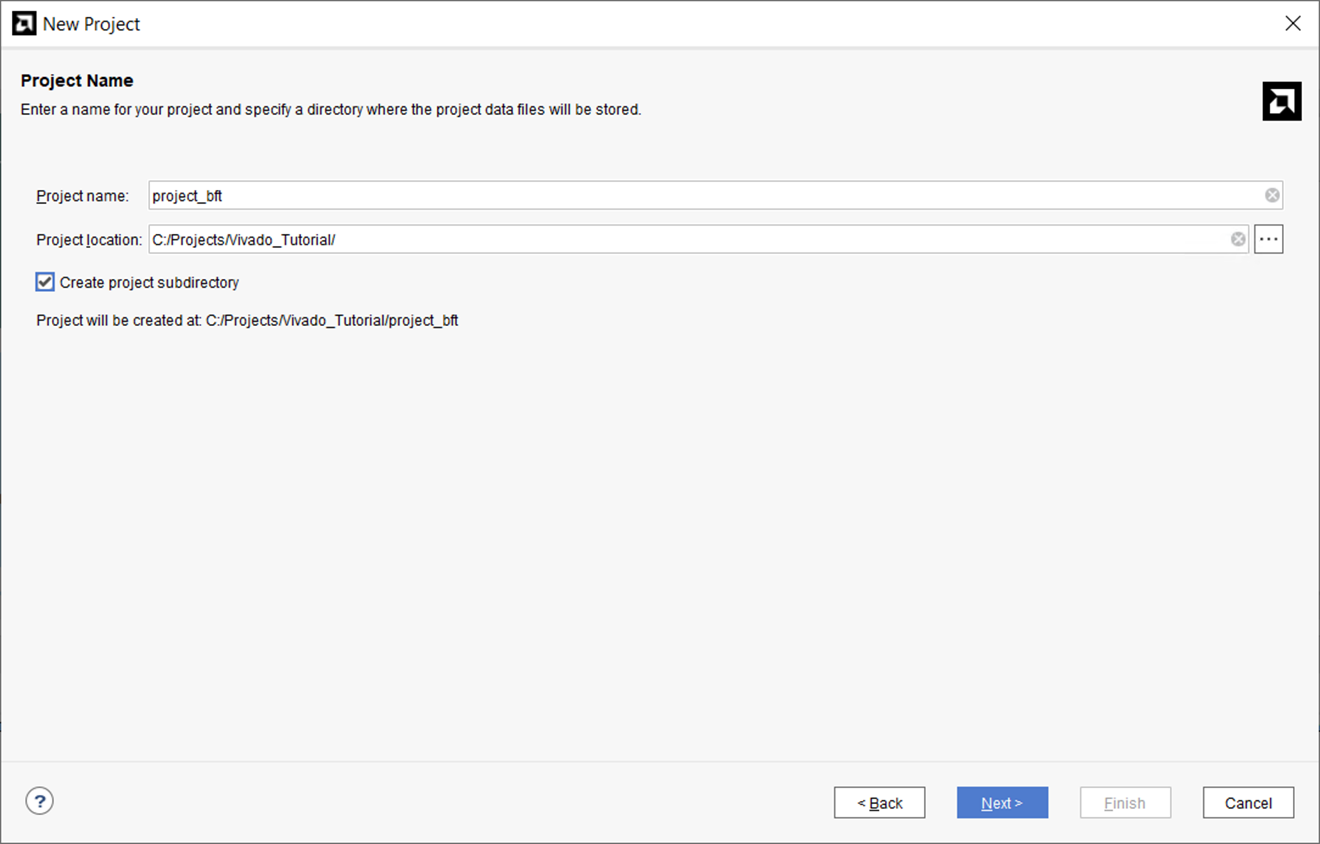
**Create a New Project**

1. After Vivado opens, select Create Project on the Getting Started page.
2. Click Next in the New Project wizard.
3. Specify the Project Name and Location:

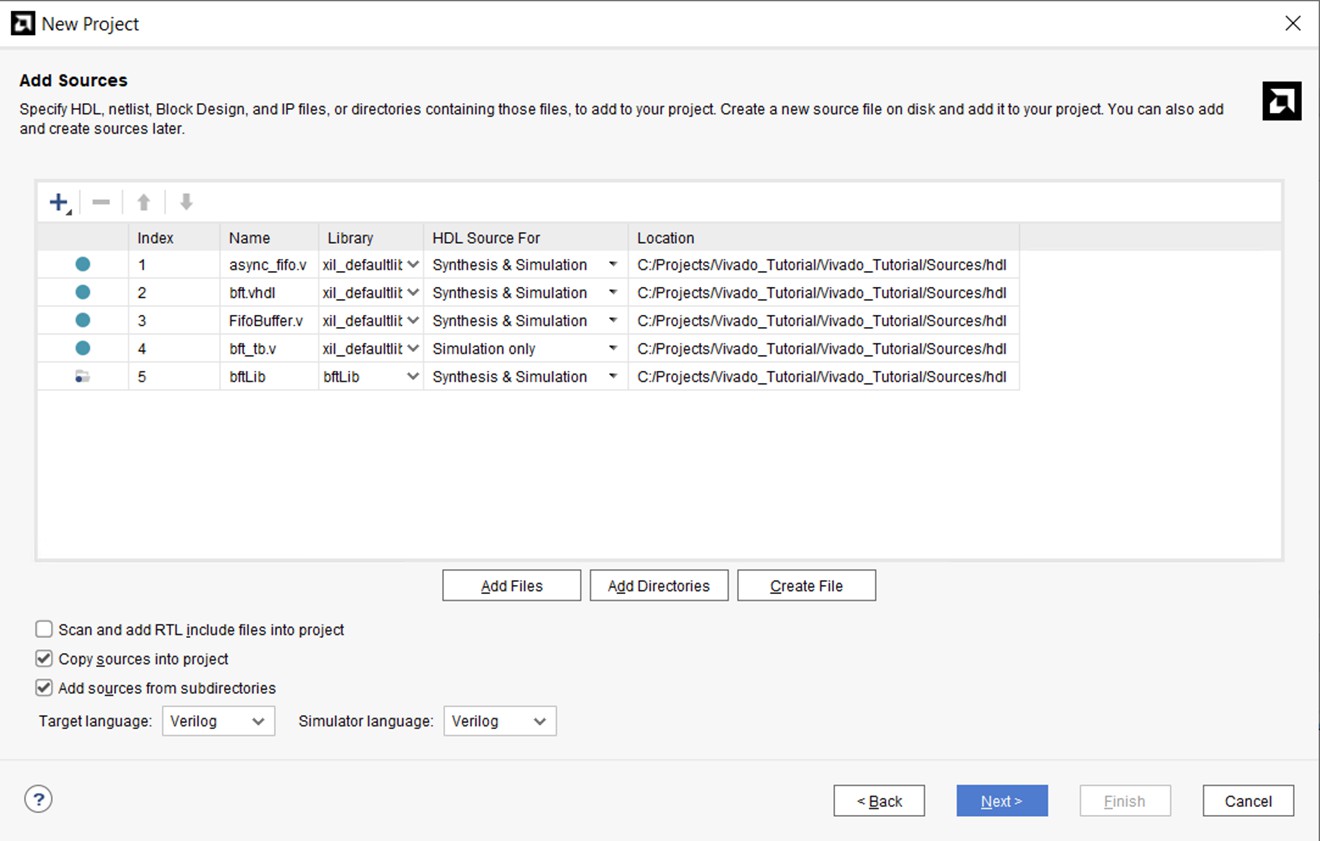
Project name project\_bft Project Location

<Extract\_Dir>/Vivado\_Tutorial/Tutorial\_Created\_Data

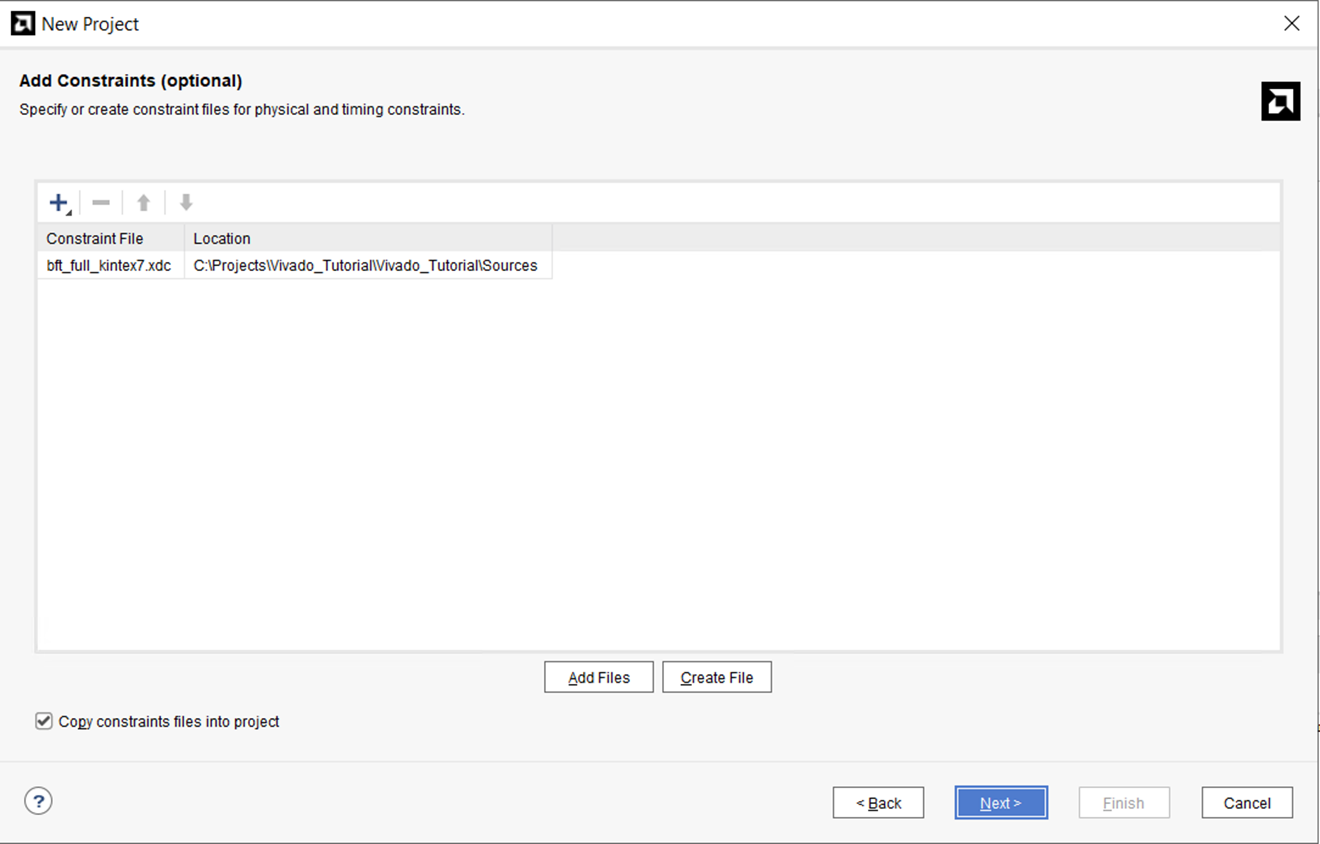
1. Click Next.



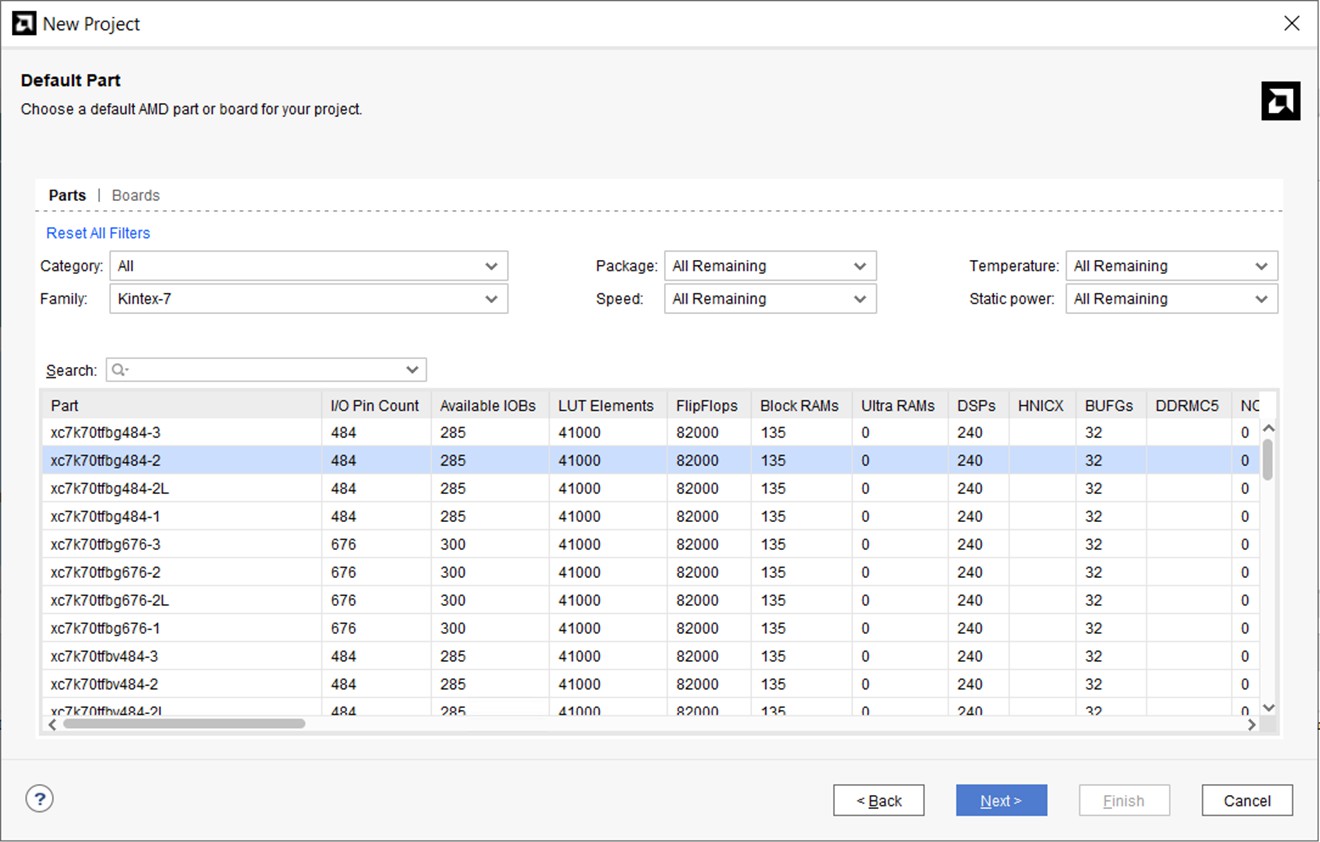
1. Select RTL Project as the Project Type and click Next.
2. Click the button and select Add Files.
   1. Browse to <Extract\_Dir>/Vivado\_Tutorial/Sources/hdl/
   2. Press and hold the Ctrl key, and click to select the following files, async\_fifo.v, bft.vhdl, FifoBuffer.v.
   3. Click OK to close the File Browser.
3. Click the button and select Add Directories.
   1. Select the <Extract\_Dir>/Vivado\_Tutorial/Sources/hdl/bftLib directory.
   2. Click Select.
4. Click the HDL Sources For column for the bft\_tb.v file and change Synthesis and Simulation to Simulation only, as shown in the following figure.



1. Click in the Library column for the bftLib, and manually edit the value to change it from xil\_defaultlib (or work) to bftLib, as shown in the following figure.
2. Enable the check boxes for Copy sources into project, and Add sources from subdirectories.
3. Set the Target Language to Verilog to define the language of the netlist generated by Vivado synthesis.
4. Set the Simulator Language to Verilog to define the language required by the logic simulator.
5. Click Next.
6. On the Add Constraints Page, click Add Files.
7. Browse to and select <Extract\_Dir>/Vivado\_Tutorial/Sources/bft\_full\_kintex7.xdc.
8. Click OK to close the File Browser.
9. Enable the check box for Copy constraints files into project.

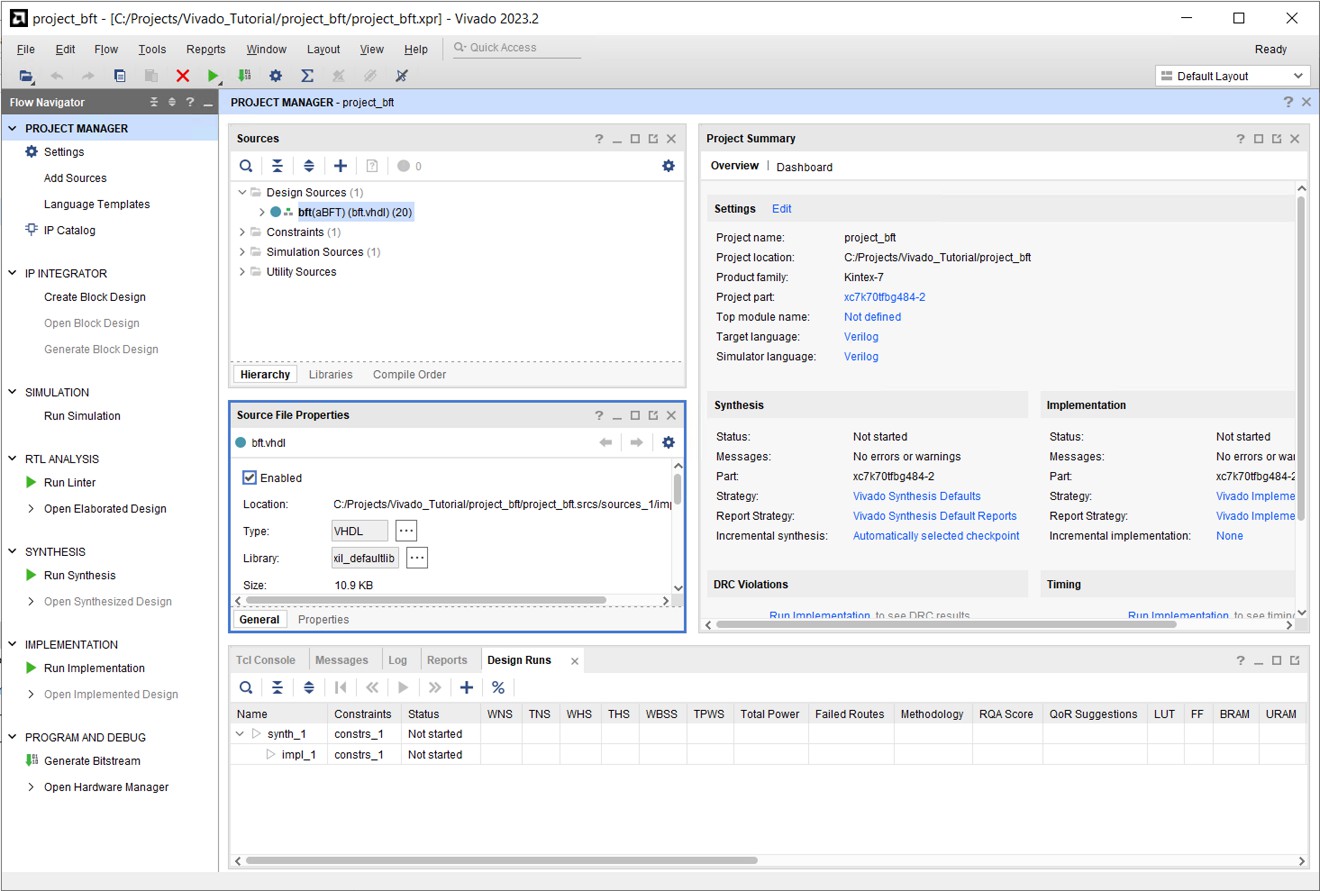


1. Click Next to move to the Default Part page.
2. On the Default Part page, click the Family filter and select the Kintex-7 family.
3. Scroll to the top of the list and select the xc7k70tfbg484-2 part, and click Next.



1. Click Finish to close the New Project Summary page, and create the project.

The Vivado IDE opens project\_bft in the default layout.



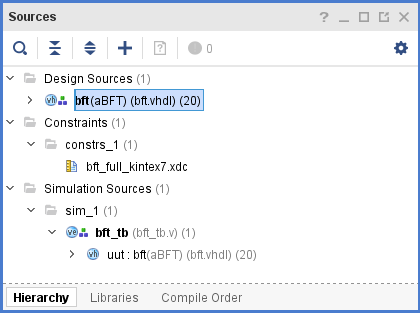
**Step 2: Using the Sources Window and Text Editor**

The Vivado tool lets you add different design sources including Verilog, VHDL, EDIF, NGC format cores, SDC, XDC, DCP design checkpoints, Tcl constraints files, and simulation test benches. These files can be sorted in a variety of ways using the tabs at the bottom of the Sources window (Hierarchy, Libraries, or Compile Order).

Important: NGC format files are not supported in the Vivado Design Suite for UltraScale™ devices. It is recommended that you regenerate the IP using the Vivado Design Suite IP customization tools with native output products. Alternatively, you can use the NGC2EDIF command to migrate the NGC file to EDIF format for importing. However, Xilinx recommends using native Vivado IP rather than XST-generated NGC format files going forward.

The Vivado IDE includes a context sensitive text editor to create and develop RTL sources, constraints files, and Tcl scripts. You can also configure the Vivado IDE to use third party text editors. Refer to the *Vivado Design Suite User Guide: Using the Vivado IDE* ([UG893](https://www.xilinx.com/cgi-bin/docs/rdoc?v=2020.2%3Bd%3Dug893-vivado-ide.pdf)) for information on configuring the Vivado tool.

**Explore the Sources Window and Project Summary**

1. Examine the information in the Project Summary. More detailed information is presented as the design progresses through the design flow.
2. Examine the Sources window and expand the Design Sources, Constraints and Simulation Sources folders.

The Design Sources folder helps keep track of VHDL and Verilog source files and libraries. Notice the Hierarchy tab displays by default.

1. Select the Libraries tab and the Compile Order tabs in the Sources window and notice the different ways that sources are listed.

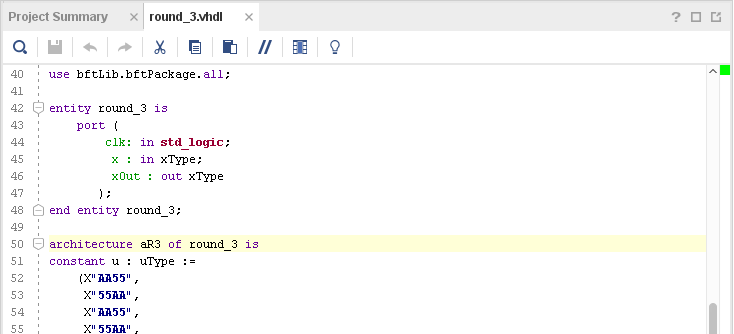
The Libraries tab groups source files by file type. The Compile Order tab shows the file order used for synthesis.

1. Expand the various folders to view the design source information.
2. Select the Hierarchy tab.

**Explore the Text Editor**

1. Select one of the VHDL sources in the Sources window.
2. Right-click to review the commands available in the popup menu.
3. Select Open File, and use the scroll bar to browse the file contents in the Text Editor.

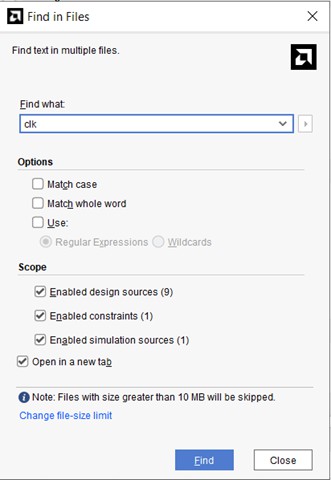
You can also double-click source files in the Sources window to open them in the Text Editor.



Notice that the Text Editor displays the RTL code with context sensitive coloring of keywords and comments. The Fonts and Colors used to display reserved words can be configured using the Tools > Settings command. Refer to *Vivado Design Suite User Guide: Using the Vivado IDE* ([UG893](https://www.xilinx.com/cgi-bin/docs/rdoc?v=2020.2%3Bd%3Dug893-vivado-ide.pdf)) for more information.

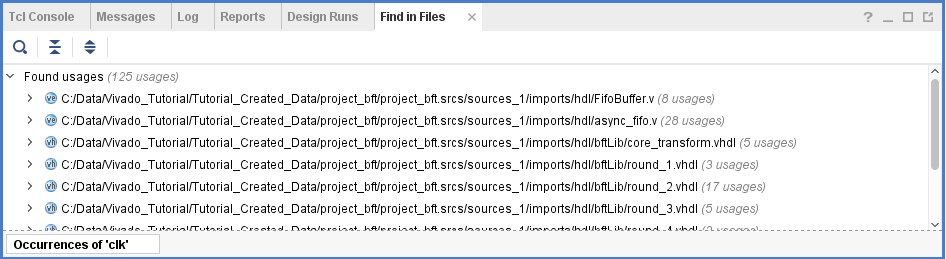
1. With the cursor in the Text Editor, right-click and select Find in Files. Note the Replace in Files command as well.

The Find in Files dialog box opens with various search options.



1. Enter clk in the Find what: field, and click Find.

The Find in Files window displays in the messaging area at the bottom of the Vivado IDE.



1. In the Find in Files window, expand one of the displayed files, and select an occurrence of clk in the file.

Notice that the Text Editor opens the selected file and displays the selected occurrence of clk in the file.

1. Close the Find in Files – Occurrences window.
2. Close the open Text Editor windows.

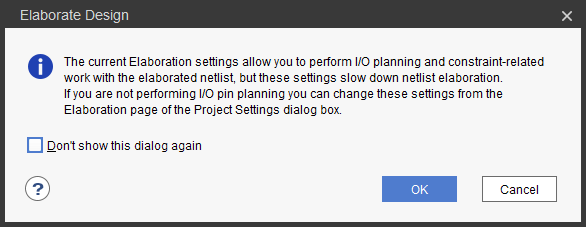
The next few steps highlight some of the design configuration and analysis features available prior to running synthesis.

**Step 3: Elaborating the RTL Design**

The Vivado IDE includes an RTL analysis and IP customizing environment. There are also several RTL Design Rule Checks (DRCs) to examine ways to improve performance or power on the RTL design.

1. Select Open Elaborated Design in the Flow Navigator to elaborate the design.

Tip: A dialog box appears informing you that your current settings will slow down netlist elaboration. You can click OK to continue or Cancel to return to your project and edit your Elaboration Settings, available in the Flow Navigator.



1. Ensure that the Layout Selector pull down menu in the main Toolbar has Default Layout selected.

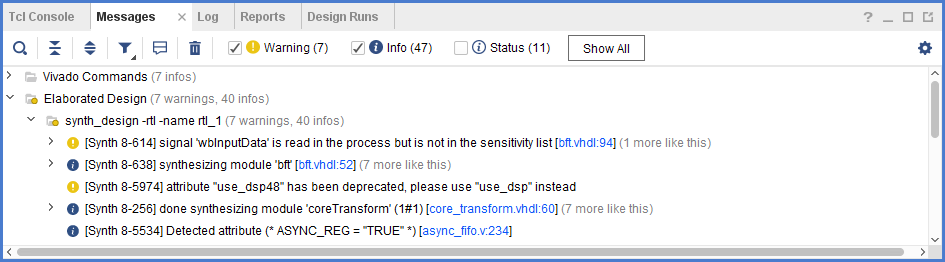
The Elaborated Design enables various analysis views including an RTL Netlist, Schematic, and Graphical Hierarchy. The views have a cross-select feature, which helps you to debug and optimize the RTL.

1. Explore the logic hierarchy in the RTL Netlist window and examine the Schematic.

You can traverse the schematic by double-clicking on cells to push into the hierarchy, or by using commands like the Expand Cone or Expand/Collapse from the Schematic popup menu. Refer to the *Vivado Design Suite User Guide: Using the Vivado IDE* ([UG893](https://www.xilinx.com/cgi-bin/docs/rdoc?v=2020.2%3Bd%3Dug893-vivado-ide.pdf)) for more information on using the Schematic window.

1. Select any logic instance in the Schematic and right-click to select the Go to Source or Go to Definition commands.

The Text Editor opens the RTL source file for the selected cell with the logic instance highlighted. In the case of the Go to Definition command, the RTL source file containing the module definition is opened. With Go to Source, the RTL source containing the instance of the selected cell is opened.

1. Click the Messages window at the bottom of the Vivado IDE, and examine the messages.
2. Click the Collapse All button in the Messages toolbar.
3. Expand the Elaborated Design and the synth\_design -rtl -name rtl\_1 messages.

Notice there are links in the messages to open the RTL source files associated with a message.

1. Click one of the links and the Text Editor opens the RTL source file with the relevant line highlighted.
2. Close the Text Editor windows.
3. Close the Elaborated Design by clicking on the X on the right side of the Elaborated Design window banner, and click OK to confirm.

**Step 4: Using the IP Catalog**

The Xilinx IP catalog provides access to the Vivado IP configuration and generation features. You can sort and search the Catalog in a variety of ways. IP can be customized, generated, and instantiated.

1. Click the IP Catalog button in the Flow Navigator, under Project Manager.
2. Browse the IP catalog to examine the various categories and IP filtering capabilities.
3. Click the Group by taxonomy and repository icon and notice the selection to Group by taxonomy and Group by repository.
4. Expand the Basic Elements folder.
5. Double-click DSP48 Macro.

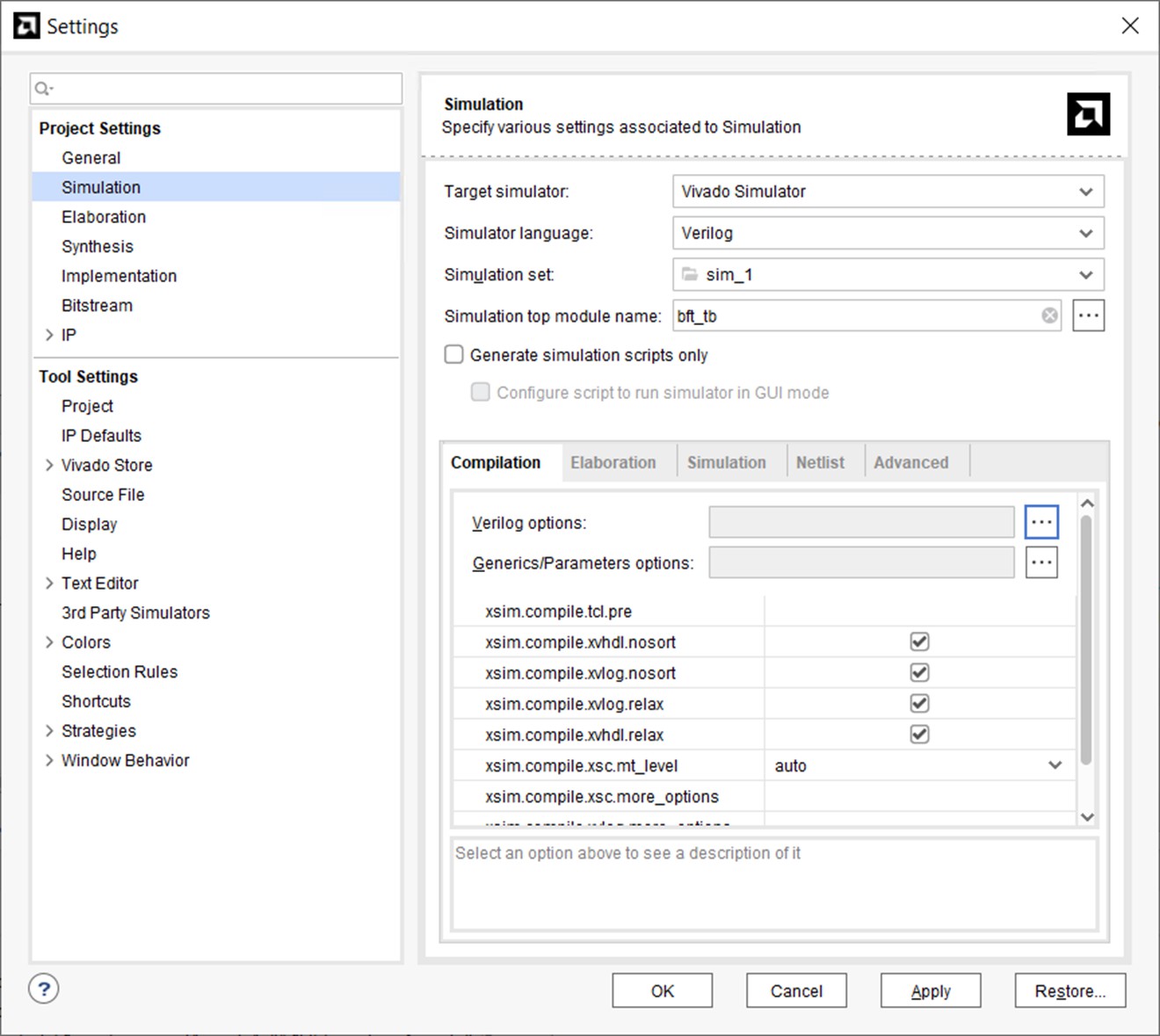
The Customize IP dialog is opened directly within Vivado Design Suite, which allows you to perform native customization and configuration of IP within the tool. To learn more about IP configuration and implementation, see the *Vivado Design Suite User Guide: Designing with IP* ([UG896](https://www.xilinx.com/cgi-bin/docs/rdoc?v=2020.2%3Bd%3Dug896-vivado-ip.pdf)) and the *Vivado Design Suite Tutorial: Designing with IP* ([UG939](https://www.xilinx.com/cgi-bin/docs/rdoc?v=2020.2%3Bd%3Dug939-vivado-designing-with-ip-tutorial.pdf)).

1. Click Cancel to close the Customize IP dialog without adding the IP to the current design.
2. Close the IP Catalog tab by clicking on the X on the window tab.

**Step 5: Running Behavioral Simulation**

The Vivado IDE integrates the Vivado Simulator, which enables you to add and manage simulation sources in the project. You can configure simulation options, and create and manage simulation source sets. You can run behavioral simulation on RTL sources, prior to synthesis.

1. In the Flow Navigator, under Project Manager, click the Settings command. The Settings dialog box opens with Project Settings at the top, and Tool Settings below that.



1. Examine the settings available on the Simulation page, then click Cancel to close the dialog box.
2. Click the Run Simulation command in the Flow Navigator, then click the Run Behavioral Simulation in the sub-menu.
3. Examine and explore the Simulation environment. Simulation is covered in detail in the *Vivado Design Suite User Guide: Logic Simulation* ([UG900](https://www.xilinx.com/cgi-bin/docs/rdoc?v=2020.2%3Bd%3Dug900-vivado-logic-simulation.pdf)) and the *Vivado Design Suite Tutorial: Logic Simulation* ([UG937](https://www.xilinx.com/cgi-bin/docs/rdoc?v=2020.2%3Bd%3Dug937-vivado-design-suite-simulation-tutorial.pdf)).
4. Close the simulation by clicking the X icon on the Behavioral Simulation view banner.
5. Click OK to close the Simulation window and click No if prompted to save changes.

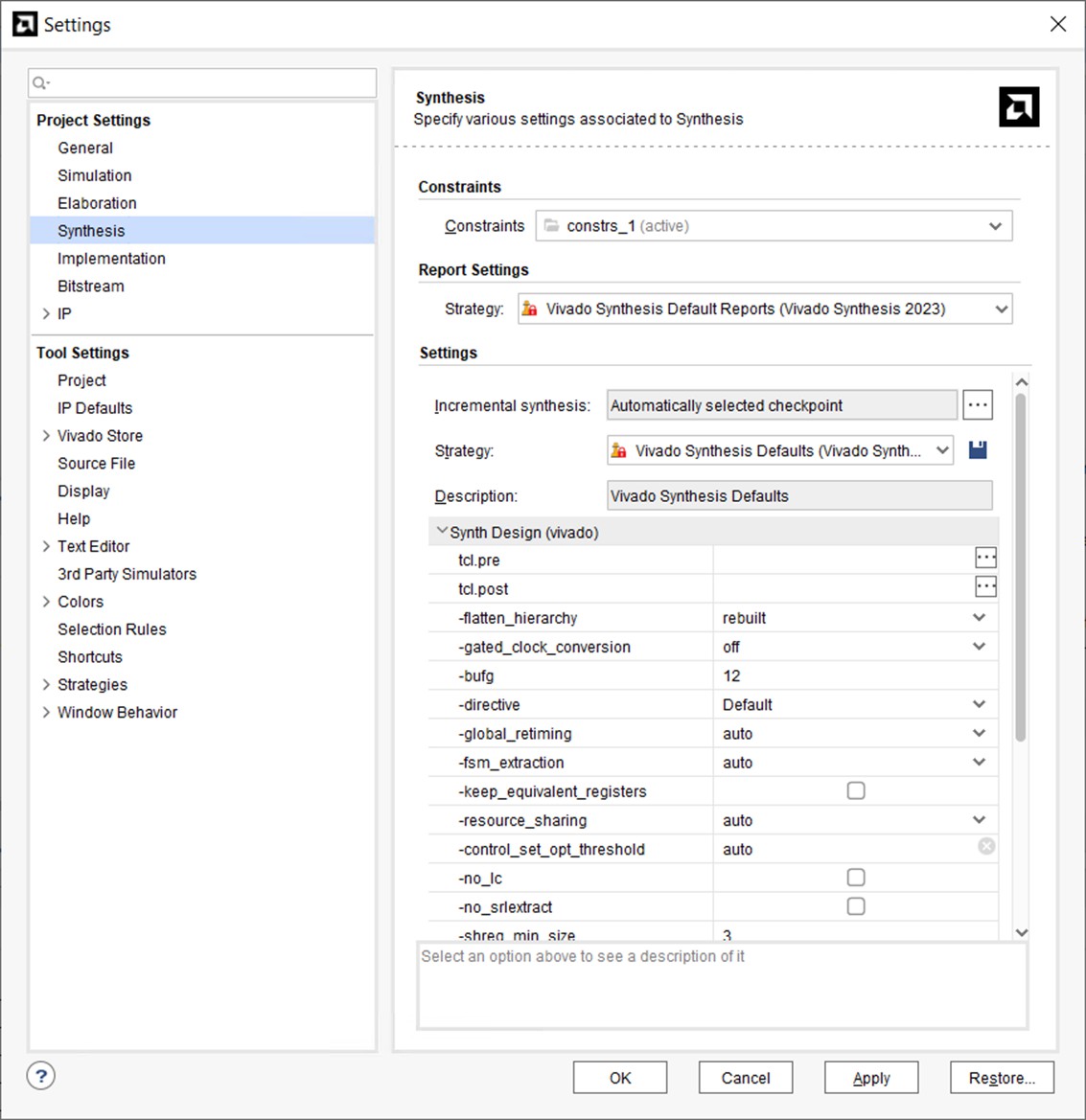
**Step 6: Reviewing Design Run Settings**

One of the main differences between the Non-Project mode you used in Lab #1 and the Project mode, which you are now using, is the support of design runs for synthesis and implementation. Non-Project mode does not support design runs.

Design runs are a way of configuring and storing the many options available in the different steps of the synthesis and implementation process. You can configure these options and save the configurations as strategies to be used in future runs. You can also define Tcl.pre and Tcl.post scripts to run before and after each step of the process, to generate reports before and after the design progresses.

Before launching the synthesis and implementation runs you will review the settings and strategies for these runs.

In the Flow Navigator, under Project Manager, click the Settings command. The Settings dialog box opens.

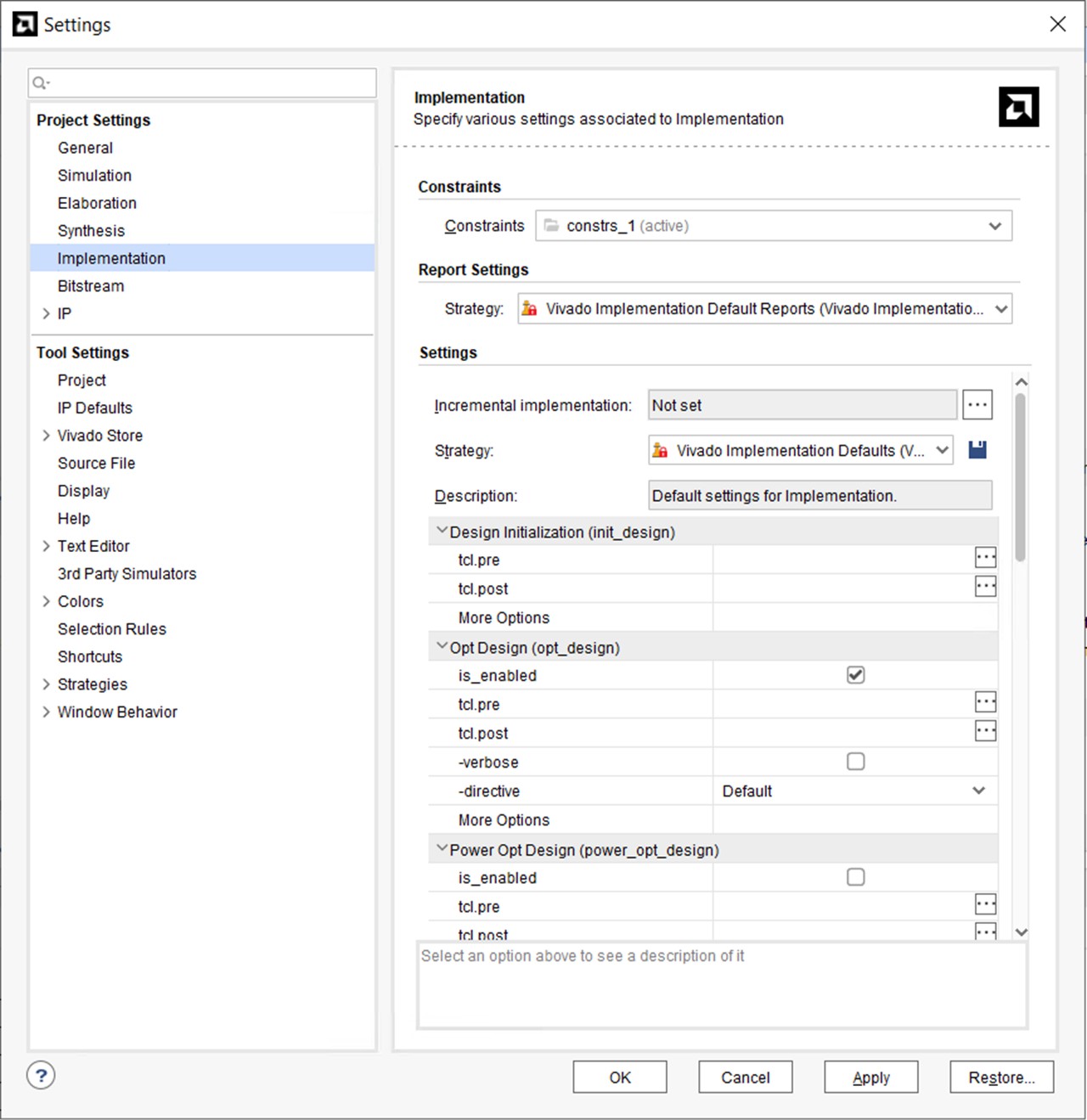


1. Select the Synthesis page under Project Settings.

The Synthesis Settings provide you access to the many options available for configuring Vivado synthesis. For a complete description of these options, see the *Vivado Design Suite User Guide: Synthesis* ([UG901](https://www.xilinx.com/cgi-bin/docs/rdoc?v=2020.2%3Bd%3Dug901-vivado-synthesis.pdf)).

1. After reviewing the various synthesis options, select the Implementation page on the left side of the Settings dialog box, as shown in the following figure.

The Settings change to reflect the Implementation settings. You can view the available options for implementation runs. For a complete description of these options, see the *Vivado Design Suite User Guide: Implementation* ([UG904](https://www.xilinx.com/cgi-bin/docs/rdoc?v=2020.2%3Bd%3Dug904-vivado-implementation.pdf)).



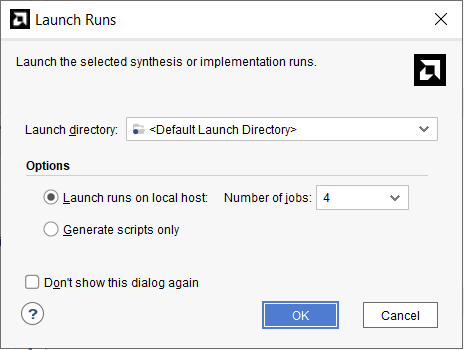
1. Click Cancel to close the Settings dialog box.

You are now ready to launch Vivado synthesis and implementation.

**Step 7: Synthesizing and Implementing the Design**

After configuring the synthesis and implementation run options, you can:

* + Use the Run Synthesis command to run only synthesis.
  + Use the Run Implementation command, which will first run synthesis if it has not been run and then run implementation.
  + Use the Generate Bitstream command, which will first run synthesis, then run implementation if they have not been run, and then write the bitstream for programming the Xilinx device.

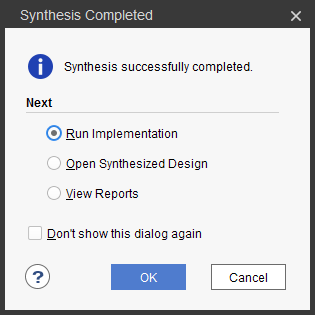


For this tutorial, run these steps one at a time.

1. In the Flow Navigator, click the Run Synthesis button.
2. Click OK to launch Synthesis with the default options and wait for the task to complete.

Notice the progress bar in the upper-right corner of the Vivado IDE, indicating the run is in progress. Vivado launches the synthesis engine in a background process to free up the tool for other actions. While the synthesis process is running in the background, you can continue browsing Vivado IDE windows, run reports, and further evaluate the design. You will notice that the Log window displays the synthesis log at the bottom of the IDE. This is also available through the Reports window.

After synthesis has completed, the Synthesis Completed dialog box prompts you to choose the next step.



1. Select Run Implementation, and click OK.
2. Click OK to launch Implementation with the default options and wait for the task to complete.

The implementation process is launched, and placed into a background process after some initialization.

The next step in this tutorial shows you how to perform design analysis of the synthesized design while waiting for implementation to complete.

**Chapter 4**

# HARDWARE DESCRIPTION LANGUAGES & TOOLS USED

#### HARDWARE DESCRIPTION LANGUAGE

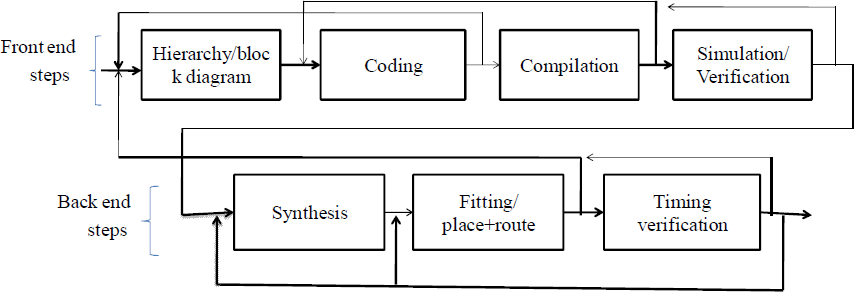
In [electronics](http://en.wikipedia.org/wiki/Electronics), a **hardware description language** (**HDL**) is a specialized [computer language](http://en.wikipedia.org/wiki/Computer_language) used to program the structure, design and operation of [electronic circuits](http://en.wikipedia.org/wiki/Electronic_circuit), and most commonly, [digital](http://en.wikipedia.org/wiki/Digital_logic) [logic](http://en.wikipedia.org/wiki/Digital_logic) circuits.

A hardware description language enables a precise, [formal](http://en.wikipedia.org/wiki/Formal_language) description of an electronic circuit that allows for the automated analysis, [simulation](http://en.wikipedia.org/wiki/Simulation_software), and simulated [testing](http://en.wikipedia.org/wiki/Software_testing) of an electronic circuit. It also allows for the [compilation](http://en.wikipedia.org/wiki/Compiler) of an HDL program into a lower level specification of physical electronic components, such as the set of masks used to create an [integrated circuit](http://en.wikipedia.org/wiki/Integrated_circuit).

A hardware description language looks much like a programming language such as [C](http://en.wikipedia.org/wiki/C_(programming_language)); it is a textual description consisting of expressions, statements and control structures. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time.

HDLs form an integral part of [electronic design automation (EDA)](http://en.wikipedia.org/wiki/Electronic_design_automation) systems, especially for complex circuits, such as [microprocessors](http://en.wikipedia.org/wiki/Microprocessor).

Hardware description languages (HDLs), mainly to describe logic equations to be realized in programmable logic devices (PLDs).in the 1990s, HDL usage by digital systems designers accelerated as PLDs, CPLDs, and FPGAs became inexpensive and common place. Designers turned to HDLs as a means to design individual modules within a system-on-chip.



**Figure: Steps in an HDL Based Design Flow**

#### MOTIVATION

Due to the exploding complexity of digital electronic circuits since the 1970s (see [Moore's law](http://en.wikipedia.org/wiki/Moore%27s_law)), circuit designers needed digital logic descriptions to be performed at a high level without being tied to a specific electronic technology, such as [CMOS](http://en.wikipedia.org/wiki/CMOS) or [BJT](http://en.wikipedia.org/wiki/BJT). HDLs were created to implement [register-transfer level](http://en.wikipedia.org/wiki/Register-transfer_level) abstraction, a model of the data flow and timing of a circuit.[[1]](http://en.wikipedia.org/wiki/Hardware_description_language#cite_note-1)

There are two major hardware description languages: VHDL and Verilog. There are different types of description in them "dataflow, behavioral and structural.

#### STRUCTURE OF HDL

HDLs are standard text-based expressions of the structure of electronic systems and their behaviour over time. Like [concurrent programming](http://en.wikipedia.org/wiki/Concurrent_programming) languages, HDL syntax and semantics include explicit notations for expressing [concurrency](http://en.wikipedia.org/wiki/Concurrency_(computer_science)). However, in contrast to most software [programming languages](http://en.wikipedia.org/wiki/Programming_language), HDLs also include an explicit notion of time, which is a primary attribute of hardware. Languages whose only characteristic is to express circuit connectivity between a hierarchy of blocks are properly classified as [net list](http://en.wikipedia.org/wiki/Netlist) languages used in electric computer-aided design ([CAD](http://en.wikipedia.org/wiki/CAD)). HDL can be used to express designs in structural, behavioral or register-transfer-level architectures for the same circuit functionality; in the latter two cases the [synthesizer](http://en.wikipedia.org/wiki/Logic_synthesis) decides the architecture and logic gate layout.

HDLs are used to write executable specifications for hardware. A program designed to implement the underlying semantics of the language statements and simulate the progress of time provides the hardware designer with the ability to model a piece of hardware before it is created physically. It is this executability that gives HDLs the illusion of being [programming](http://en.wikipedia.org/wiki/Programming_languages) [languages](http://en.wikipedia.org/wiki/Programming_languages), when they are more precisely classified as [specification languages](http://en.wikipedia.org/wiki/Specification_language) or [modeling](http://en.wikipedia.org/wiki/Modeling_language) [languages](http://en.wikipedia.org/wiki/Modeling_language). Simulators capable of supporting discrete-event (digital) and continuous-time (analog) modeling exist, and HDLs targeted for each are available.

In any design, specifications are written first. Specifications describe the functionality, interface and overall architecture of the digital circuit to be designed. The next step is the actual writing of HDL code for modules, their interfaces and their internal details. After the code has written we have to compile the code, this step is known as compilation. Here the HDL compiler

analyzes the code for syntax errors and also checks it for compatibility with other modules which it relies.

The most satisfying step is simulation or verification. The HDL simulator allows to define and apply the inputs to the design and to observe its outputs without ever having to build the physical circuit. There are at least two dimensions to verification. In timing verification, the circuit operation including estimated delays, the setup, hold and other timing requirements for sequential devices like flip flops are met. In the functional verification the circuits logical operation independent of timing considerations; gate delays and other timing parameters are considered to be zero.

After verification step, the synthesis process is done in the back end stage. There are three basic steps, the first synthesis, converting the HDL description into a set of primitive or components that can be assembled in the target technology and it may generate a list of gates and a net list that specifies how they are interconnected.

In the fitter step, a fitter maps the synthesized components on to available device resources. It may mean selecting microcells or laying down individual gates in a pattern an finding ways to connect them within the physical constraints of the FPGA or ASIC die, is calle as place and route process. The final step is post fitting verification of the fitted circuit. It is only at the stage that the actual circuit delays due to wire lengths, electrical loading, and other factors can be calculated with reasonable precision.

#### HDL TOOL SUITES

HDL tool suite really has several different tools with their own names and purposes:

* A text editor allows to write, edit and save an HDL program. It often contain HDLspecific features, such as recognizing specific file name extensions and recognizing HDL reserved and comments and displaying them in different colors.
* The compiler is responsible for parsing the HDL program, finding syntax errors and figuring out what the program really says.
* A synthesizer or synthesis tools targets the design to a specific hardware technology, such as FPGA, ASIC etc...
* The simulator runs the specified input sequence on the described hardware and determines the values of the hardware¡¦s internal signals and its outputs over a specified period of time.
* The output of the simulator can be include waveforms to be viewed using the waveform editor.
* A schematic viewer may create a schematic diagram corresponding to an HDL program, based on the intermediate-language output of the compiler.
* A translator targets the compilers intermediate language output to a real device such as PLD, FPGA OR ASIC.
* A timing analyser calculates the delays through some or all of the signal paths in the final chip and produces a report showing the worst case paths and their delays.

#### VHDL HARDWARE DESCRIPTION LANGUAGE

VHDL¨ stands for VHSIC hardware description language¨. VHSIC¨ in turns stands for Very High Speed Integrated Circuit.

#### VHDL ADVANTAGES

The key advantage of VHDL, when used for systems design, is that it allows the behavior of the required system to be described (modelled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires). Another benefit is that VHDL allows the description of a concurrent system. VHDL is a dataflow language, unlike procedural computing languages such as BASIC, C, and assembly code, which all run sequentially, one instruction at a time.

VHDL project is multipurpose. Being created once, a calculation block can be used in many other projects. However, many formational and functional block parameters can be tuned (capacity parameters, memory size, element base, block composition and interconnection structure).VHDL project is portable. Being created for one element base, a computing device project can be ported on another element base, for example VLSI with various technologies. Concurrency, timing and clocking can be modelled. VHDL handles asynchronous as well as synchronous sequential-circuit structures. The logical operation and timing behaviour of a design can be simulated. VHDL allows for various design methodologies, both the top-down, bottom-up and is very flexible in its approach to describing hardware.

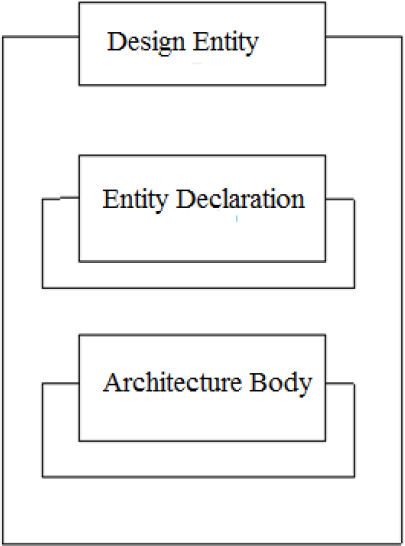
#### VHDL HISTORY AND FEATURES

In the mid-1980s, the U.S. Department of Defence (DOD) and the IEEE sponsored the development of a highly capable hardware description language called VHDL and this was got extended in 1993 and again 2002. And some of the features of the VHDL are:

* Packages are used to provide a collection of common declaration, constants, and/or subprograms to entities and architectures.
* Generics provide a method for communicating information to architecture from the external environment. They are passed through the entity construct.
* Ports provide the mechanism for a device to communicate with its environment. A port declaration defines the names, type¡¦s directions and possible default values for the signals in a components interface.
* Configuration is an instruction used to bind the component instances to design entities. In it, we specify which real entity interface and corresponding architecture body should be used for any component instances.
* Bus is a signals group or a particular method of communication.
* Driver is a source for a signal in that it provides values to be applied to the signal.
* Attribute is aVHDL object is additional information.

#### VHDL STRUCTURE

The VHDL structure or model is shown in figure. A single component model is composed of one entity and one of more architecture. The entity represents the interface specification (I/O) of the component. It defines the components external view, sometimes referred to as its pins¨ .The architecture(s) describe(s) the internal implementation of an entity.



**Figure 4.2: VHDL Structure**

#### TYPES OF ARCHITECTURES

There are three general types of architectures. A VHDL Model can be created at different abstraction levels (behavioral, dataflow, structural), according to a refinement of starting specification.

#### DATAFLOW MODELLING

Several additional concurrent statements allow VHDL to describe a circuit in terms of the flow of data and operations on it within the circuit. This style is called a dataflow description or dataflow design. Concurrent statements execute when data is available on their inputs. These statements occur in any order within the architecture. This method is to use logic equations to develop a data flow description.

#### STRUCTURAL MODELLING

Structural description can be created from pre described components. These gates can be pulled from a library of parts. A VHDL architecture that uses components is often called a structural description or structural design, because it defines the precise interconnection structure of signals and entities that realize the entity.

#### BEHAVIORAL MODELLING

Behavioural description in which the functional and possibly timing characteristic are described using VHDL concurrent statements and processes. Process is a collection of sequential statements that executes in parallel with other concurrent statement and processes. Using a process, can specify a complex interaction of signals and events in a way that executes in essentially zero simulated time during simulation and that give rise to a synthesized combinational or sequential circuit that performs the modelled operation directly.

A VHDL process statement can be used anywhere that a concurrent statement can be used. Process statement is introduced by the keyword process. AVHDL process is always either running or suspended. The list of signals in the process definition, called the sensitivity list. All statements within a process execute sequential order until it gets suspended by a wait statement.

#### VERILOG HDL

**Verilog**, standardized as **IEEE 1364**, is a [hardware description language](http://en.wikipedia.org/wiki/Hardware_description_language) (HDL) used to model electronic systems. It is most commonly used in the design and verification of [digital](http://en.wikipedia.org/wiki/Digital_electronics) [circuits](http://en.wikipedia.org/wiki/Digital_electronics) at the [register-transfer level](http://en.wikipedia.org/wiki/Register-transfer_level) of [abstraction](http://en.wikipedia.org/wiki/Abstraction_(computer_science)). It is also used in the verification of [analog](http://en.wikipedia.org/wiki/Analogue_electronics) [circuits](http://en.wikipedia.org/wiki/Analogue_electronics) and [mixed-signal circuits](http://en.wikipedia.org/wiki/Mixed-signal_integrated_circuit).

Hardware description languages such as Verilog differ from software [programming](http://en.wikipedia.org/wiki/Programming_language) [languages](http://en.wikipedia.org/wiki/Programming_language) because they include ways of describing the propagation time and signal strengths (sensitivity). There are two types of assignment operators; a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables. Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous productivity improvement for circuit designers who were already using graphical [schematic capture](http://en.wikipedia.org/wiki/Schematic_capture) software and specially written software programs to document and [simulate electronic circuits](http://en.wikipedia.org/wiki/Electronic_circuit_simulation).

A Verilog design consists of a hierarchy of modules. Modules encapsulate *design hierarchy*, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a

begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a [dataflow language](http://en.wikipedia.org/wiki/Dataflow_language).

Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating, undefined") and signal strengths (strong, weak, etc.). This system allows abstract modeling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

**Verilog - Modules**

The module is the basic unit of hierarchy in Verilog

* Modules describe:
  + boundaries [module, endmodule]
  + inputs and outputs [ports]
  + how it works [behavioral or RTL code]
* Can be a single element or collection of lower level modules
* Module can describe a hierarchical design (a module of modules)
* A module should be contained within one file
* Module name should match the file name
* Module fadder resides in file named fadder.sv
* Multiple modules can reside within one file (not recommended)
* Correct partitioning a design into modules is critical.

#### SOME LEXICAL CONVENTIONS - COMMENTS

* Comments are signified the same as C
* One line comments begin with ”//”
* Multi-line comments start: /\*, end: \*/

#### SOME LEXICAL CONVENTIONS - IDENTIFIERS

* Identifiers are names given to objects so that they may be referenced
* They start with alphabetic chars or underscore
* They cannot start with a number or dollar sign
* All identifiers are case sensitive

**VERILOG vs VHDL**

VERILOG AND VHDL are Hardware Description languages that are used to write programs for electronic chips. These languages are used in electronic devices that do not share a computer’s basic architecture. VHDL is the older of the two, and is based on Ada and Pascal, thus inheriting characteristics from both languages. Verilog is relatively recent, and follows the coding methods of the C programming language.

VHDL is a strongly typed language, and scripts that are not strongly typed, are unable to compile. A strongly typed language like VHDL does not allow the intermixing, or operation of variables, with different classes. Verilog uses weak typing, which is the opposite of a strongly typed language. Another difference is the case sensitivity. Verilog is case sensitive, and would not recognize a variable if the case used is not consistent with what it was previously. On the other hand, VHDL is not case sensitive, and users can freely change the case, as long as the characters in the name, and the order, stay the same.

In general, Verilog is easier to learn than VHDL. This is due, in part, to the popularity of the C programming language, making most programmers familiar with the conventions that are used in Verilog. VHDL is a little bit more difficult to learn and program.

VHDL has the advantage of having a lot more constructs that aid in high-level modeling, and it reflects the actual operation of the device being programmed. Complex [data](http://www.differencebetween.net/language/difference-between-data-and-information/) types and packages are very desirable when programming big and complex systems, that might have a lot of functional parts. Verilog has no concept of packages, and all programming must be done with the simple data types that are provided by the programmer.

Lastly, Verilog lacks the library [management](http://www.differencebetween.net/business/differences-between-management-and-marketing/) of software programming languages. This means that Verilog will not allow programmers to put needed modules in separate files that are called during compilation. Large projects on Verilog might end up in a large, and difficult to trace, file.

#### SUMMARY:

1. Verilog is based on C, while VHDL is based on Pascal and ADA.
2. Unlike Verilog, VHDL is strongly typed.
3. Unlike VHDL, Verilog is case sensitive.
4. Verilog is easier to learn compared to VHDL.
5. Verilog has very simple data types, while VHDL allows users to create more complex data types.
6. Verilog lacks the library management, like that of VHDL

#### TOOLS USED XILINX ISE 14.4

The Xilinx ISE tools allow you to use schematics, hardware description language (HDLs), and specially designed modules in number of ways. Schematics are drawn by using symbols for components and lines for wires. Xilinx tools is a suite of software tools used for the design of digital circuits implemented using Xilinx Field Programmable Gate Array(FPGA) or Complex programmable logic Device (CPLD).

The design procedure consists of (a) design entry, (b) synthesis and implementation of the design,(c) functional simulation and (d) testing and verification. Digital designs can be entered in various ways using the above CAD tools: using a schematic hardware description language (HDL) – Verilog or VHDL or a combination of both. In this lab we will only use the design flow that involves the use of Verilog HDL.

The steps of the design procedure are listed below:

* + Create Verilog design input file(s) using template driven editor.
  + Compile and implement the Verilog design file(s).
  + Create the test-vectors and simulate the design (functional simulation) without using a PLD (FPGA or CPLD).
  + Assign input/output pins to implement the design on a target device. Download bit stream to an FPGA or CPLD device

Test design on FPGA/CPLD device

A Verilog input file in the Xilinx software environment consists of the following segments:

Header : module name, list of input and output ports. Declarations : input and output ports, registers and wires. Logic Descriptions : equations, state machines and logic functions. End : end module

The Integrated Software Environment (ISE) is the Xilinx design software suite that allows you to take your design from design entry through Xilinx device programming.

The ISE project Navigator manages and Processes your design through the following steps in the ISE design flow.

#### DESIGN ENTRY

Design entry is the first step in the ISE design flow. During design entry, you create your source files based on your design objectives. You can create your top-level design file using a Hardware Description Language (HDL), such as VHDL, Verilog, or ABEL, or using schematic. You can use multiple formats for the lower-level source files in your design.

#### SYNTHESIS

After design entry and optimal simulation, you run synthesis. During this step, VHDL, Verilog, or mixed language designs become net list files that are accepted as input to the implementation step.

#### IMPLEMENTATION

After synthesis, you run design implementation, which converts the logical design into a physical file format that can be downloaded to selected target device. From project navigator, you can run the implementation process in one step, or you can run each of the implementation separately. Implementation processes vary depending on whether you are targeting a Field Programmable Gate Array (FPGA) or a Complex Programmable Logic Device (CPLD).

#### VERIFICATION

You can verify the functionality of your design at several points in the design flow. You can use simulator software to verify the functionality and timing of your design or a portion of your design. The simulator interprets VHDL or Verilog code into circuit functionality and displays logical results of described HDL to determine correct circuit operation. Simulation allows you to create and verify complex functions in a relatively small amount of time. You can also run in-circuit verification after programming your device.

#### DEVICE INSTALLATION

After generating a programming file, you conFig. your device. During configuration, you generate configuration files and download the programming files from a host computer to a Xilinx devi

**ISE**

Xilinx ISE is a Hardware Description Language (HDL) simulator that enables you to perform functional and timing simulations for VHDL, Verilog and mixed VHDL/Verilog designs.

#### LANGUAGE SUPPORT

**Table I Languages supported by ISE**

|  |  |
| --- | --- |
| Language | Support |
| VHDL | IEEE-STD-1076-2000 |
| Verilog | IEEE-STD-1364-2001 |
| SDF | Xilinx [NetGen] generated SDF files |
| VITAL | VITAL-2000 |
| Mixed VHDL/Verilog | Yes |
| VHDL FLI/VHPI  Verilog PLI | No  No |
| System Verilog | No |
| Other Assertion-Based Languages | No |
|  |  |

**Feature Support**

**Table Features supported by ISE**

|  |  |
| --- | --- |
| Feature | Support |
| Incremental Compilation | Yes |
| Source Code Debugging | Yes |
| SDF Annotation | Yes |
| VCD Generation | Yes |
| SAIF Support | Yes |
| Hard IP-MGT, PPC etc | Yes |
| Multi-threading | Yes |

**Simulation Using ISE**

Now that you have a test bench in your project, you can perform behavioral simulation on the design using ISE. The ISE software has full integration with ISE. The ISE software enables ISE to create the work directory, compile the source files, load the design, and perform simulation based on simulation properties.

To select ISE as your project simulator, do the following:

* In the Hierarchy pane of the Project Navigator Design Panel, right-click the device line (xc3s100E-5tq114), and select “Design Properties”.
* In the Design Properties dialog box, set the simulator field to “ISE (VHDL/Verilog)”.

**Locating the Simulation Processes**

The simulation processes in the ISE software enable you to run simulation on the design using ISE.

To locate the ISE processes, do the following:

* In the View Pane of the Project Navigator Design Panel, select “Simulation”, and select “Behavioral” from the drop-down list.
* In the Hierarchy Pane, select the test bench files (ex: stopwatch\_tb).
* In the Processes Pane, expand “ISE Simulator” to view the process hierarchy.

The following simulation processes are available:

Check “Syntax” .This process checks for syntax errors in the test bench. Simulate “Behavioral Model” .This process starts the design simulation.

**Specifying Simulation Properties**

You will perform a behavioral simulation on the stopwatch design after you set process properties for simulation.

The ISE software allows you to set several ISE properties in addition to the simulation net list properties. To see the behavioral simulation properties and to modify the properties for this tutorial, do the following:

In the Hierarchy pane of the Project Navigator

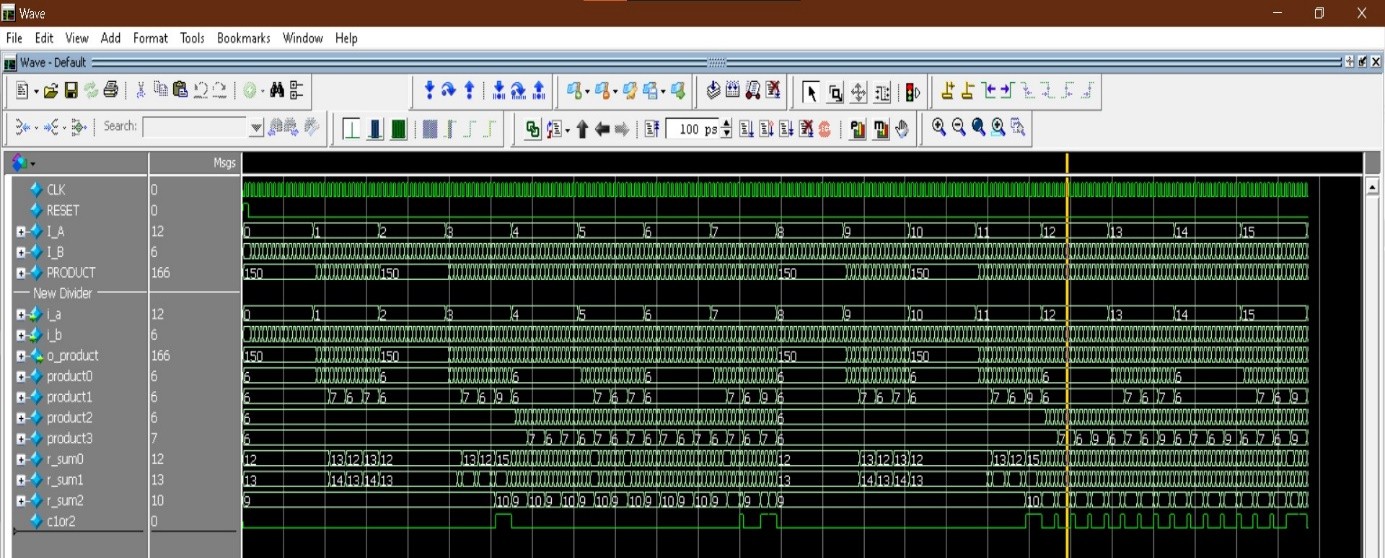
* + In the Process Pane, expand “ISE simulator”, right-click “Simulate Behavioral Model”, and select “Process Properties”.
  + In the Process Properties dialog box, set the property display level to “Advanced”. This global setting enables you to see all available properties.
  + Change the Simulation Run Time to “2000 ns”. Click “OK”.

**Performing Simulation**

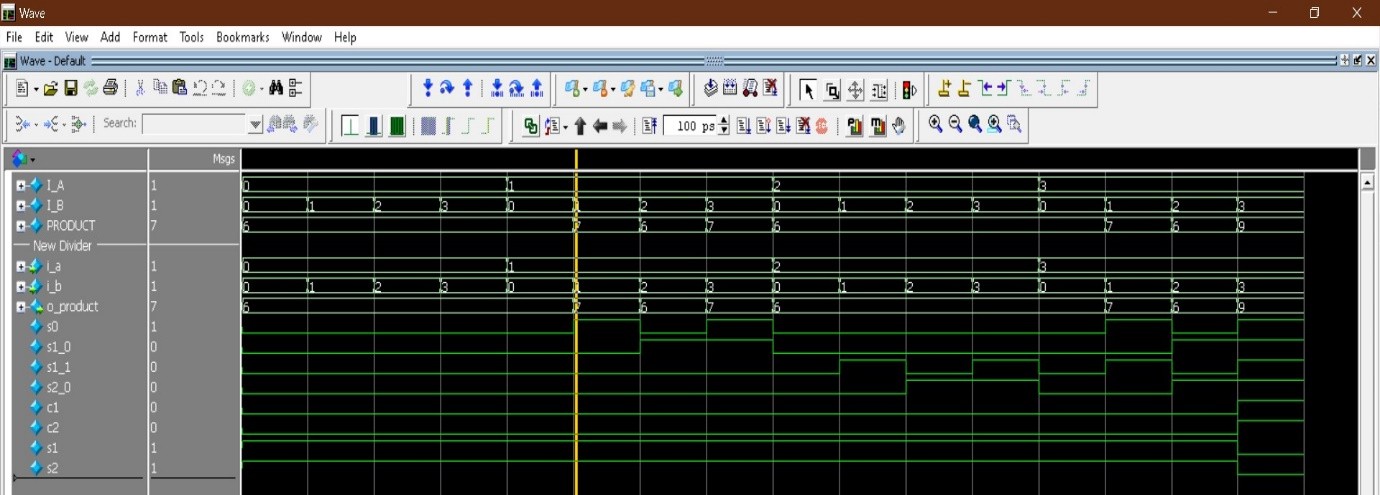
After the process Properties have been set, you are ready to run ISE to simulate the design. To start the behavioral simulation, double-click “Simulate Behavioral Model”. ISE creates the work directory, compiles the source files, loads the design, and performs simulation for the time specified.

The majority of the design runs at 100 Hz and would take a significant amount of time to simulate. The first outputs to transition after RESET is released are SF\_D and LCD\_E at around 33ms. This is why the counter may seem like it is not working in a short simulation. For the purpose of this tutorial, only the DCM signals are monitored to verify that they work correctly.

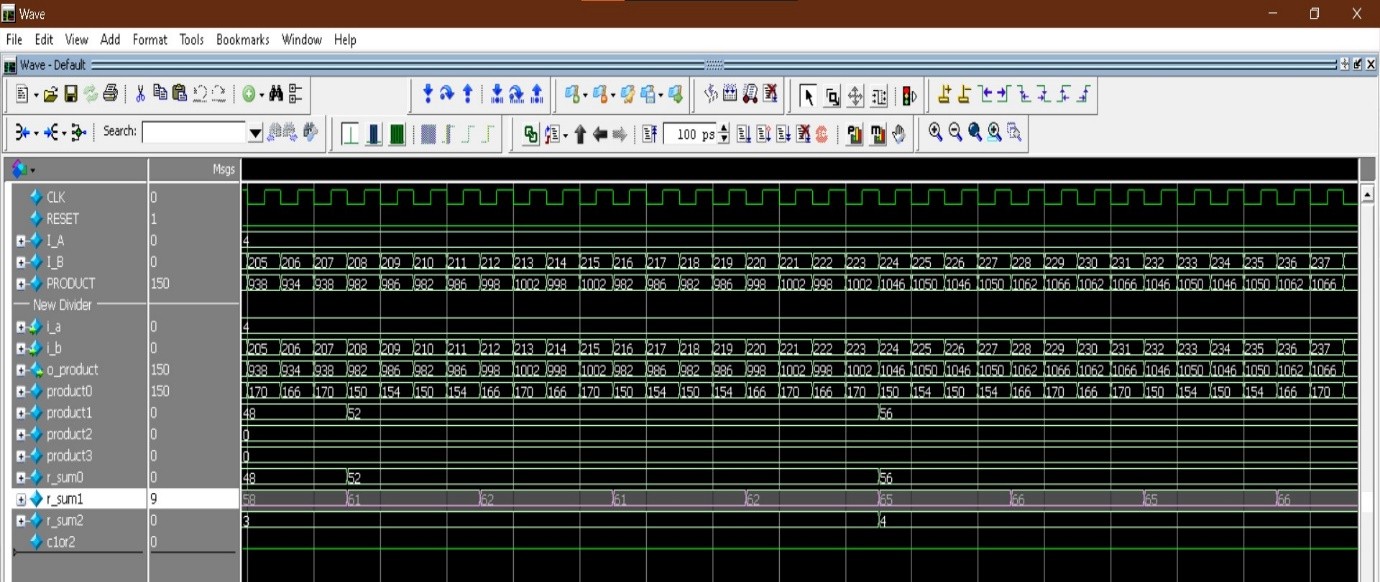
# SIMULATION RESULTS

****

**Simulation Results 4x4 AVM**

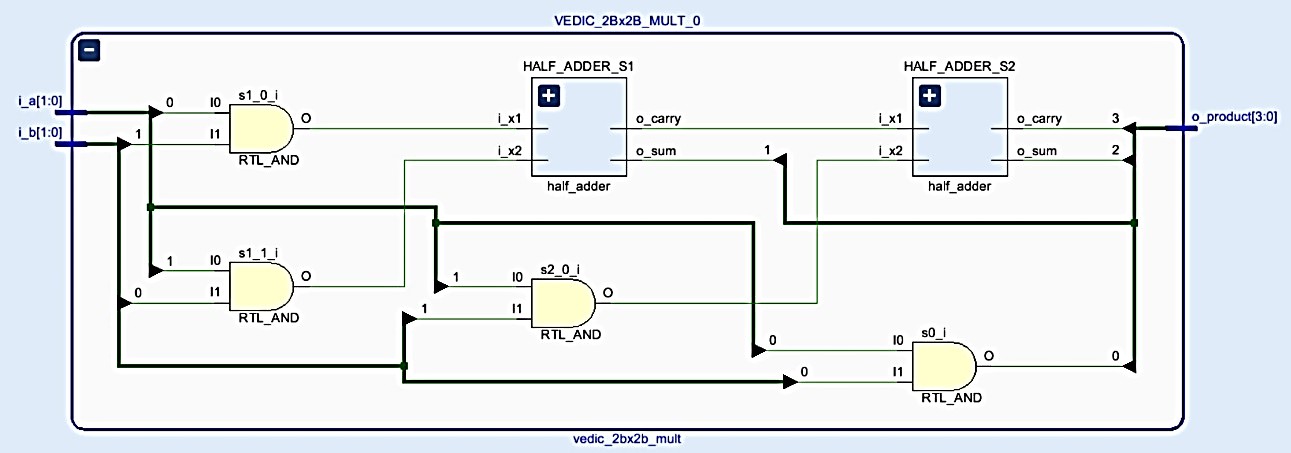
****

**Simulation Results 2x2 AVM**

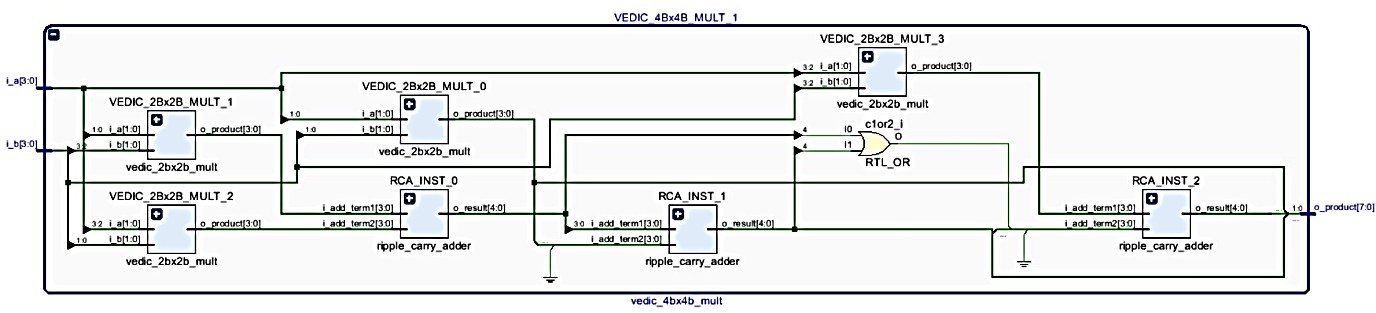
****

**Simulation Results 8x8 AVM**

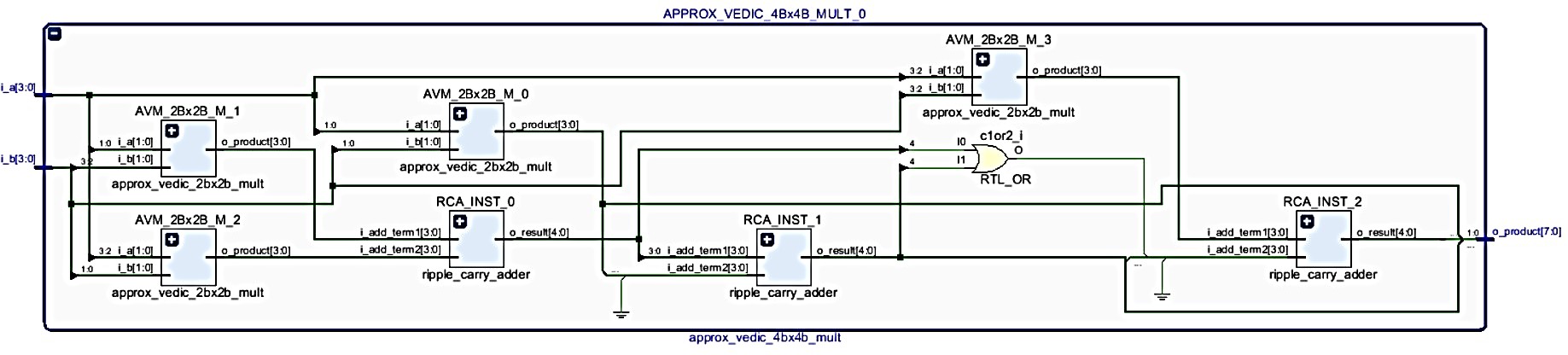
**RTL SCHEMATIC**

****

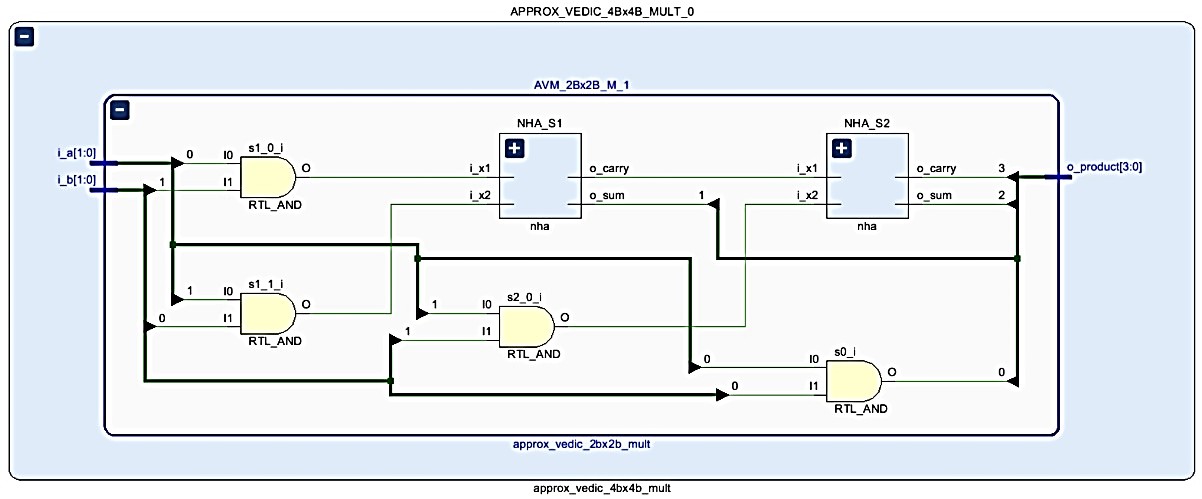
**RTL Schematic Exact 2x2 Vedic Multiplier**

****

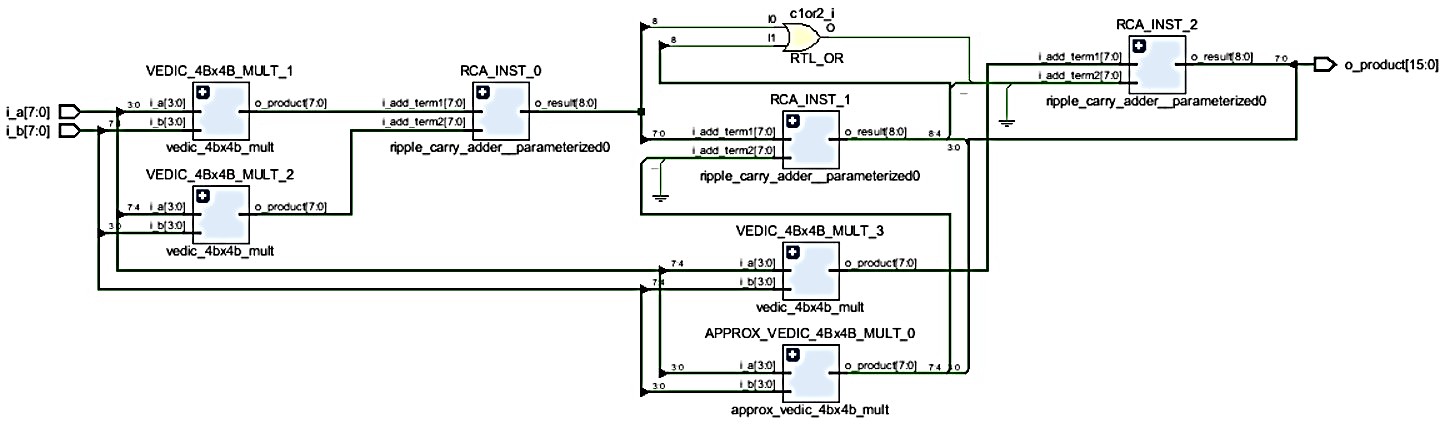
**RTL Schematic Exact 4x4 Vedic Multiplier**

****

**RTL Schematic Exact 4x4 Approx Vedic Multiplier**

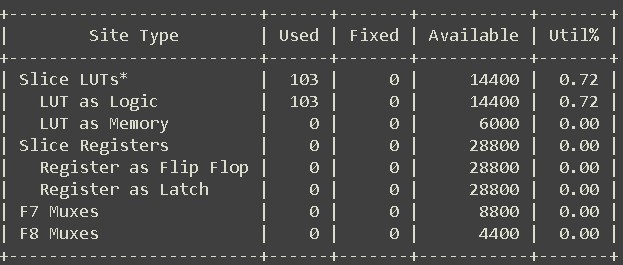


**RTL Schematic Exact 2x2 Approx Vedic Multiplier**

****

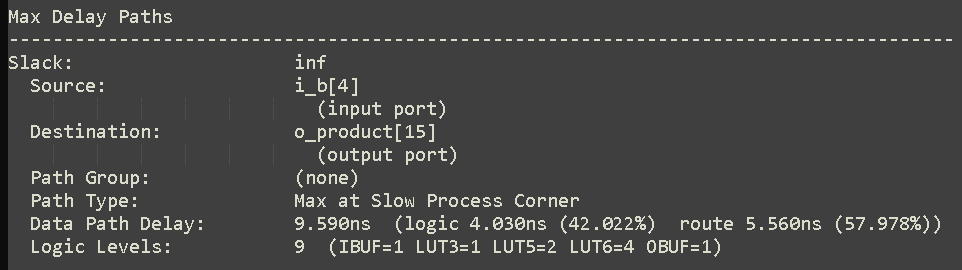
**RTL Schematic Exact 8x8 Approx Vedic Multiplier**

# AREA REPORT

****

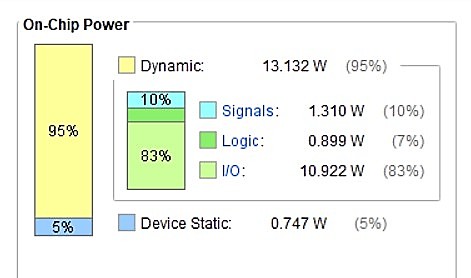
**Area Report 8x8 AVM**

## Timing Report

****

**Max Delay Path 8x8 AVM**

## Power Report

****

**APPLICATIONS AND ADVANTAGES**

**ADVANTAGES:**

**The project "A Hybrid Approximate Multiplier for Energy Efficient Image Processing" offers several advantages, particularly in applications where energy efficiency and performance trade-offs are critical. Here are the key benefits:**

1. **Energy Efficiency**
   * **The use of approximate computing reduces power consumption, making it ideal for battery-powered devices such as smartphones, IoT devices, and embedded systems.**
   * **Lower energy requirements contribute to longer battery life in portable applications.**
2. **High-Speed Computation**
   * **Approximate multipliers typically require fewer logic gates and shorter critical paths, leading to reduced computation time and faster processing speeds.**
   * **Enables real-time image processing in applications like video streaming, medical imaging, and autonomous systems.**
3. **Reduced Hardware Complexity**
   * **By using approximate multiplication techniques, the circuit complexity is significantly reduced compared to traditional exact multipliers.**
   * **Requires fewer transistors, leading to smaller chip area and cost savings in hardware manufacturing.**
4. **Acceptable Image Quality Trade-offs**
   * **Many image processing applications (e.g., object detection, facial recognition, and edge detection) do not require fully precise calculations, making approximate multipliers an effective solution.**
   * **The small amount of error introduced does not significantly degrade the overall image quality.**
5. **Improved Performance in Machine Learning & AI**
   * **Neural networks and deep learning models often operate on approximate computations. Using hybrid approximate multipliers in AI accelerators can reduce power consumption without significantly affecting accuracy.**
   * **Beneficial for AI-based image classification, object detection, and facial recognition tasks.**
6. **Cost-Effective Hardware Implementation**
   * **The reduction in power consumption, chip area, and transistor count makes this approach economical for mass production, particularly in FPGA and ASIC- based applications.**
7. **Scalability to Higher-Resolution Processing**
   * **The hybrid approach can be optimized for higher resolution images (e.g., 4K, 8K) while maintaining energy efficiency, making it suitable for advanced imaging systems.**
8. **Customizable Approximation Levels**
   * **The degree of approximation can be adjusted based on the application requirements, balancing speed, power, and accuracy.**
   * **Suitable for different domains, such as medical imaging (where accuracy is crucial) and real-time video processing (where speed is more important).**

# APPLICATIONS

The project **"A Hybrid Approximate Multiplier for Energy Efficient Image Processing"** has a wide range of applications across various domains where power efficiency and high- speed computation are crucial. Here are some of the key applications:

1. **Image & Video Processing**
   * Used in **real-time video streaming**, where fast image computations are needed with minimal energy consumption.
   * Suitable for **video compression algorithms** like JPEG, MPEG, and HEVC to reduce computational complexity.
   * Can be applied in **image filtering, edge detection, and feature extraction** in computer vision applications.
2. **Machine Learning & AI Accelerators**
   * Helps in **deep learning inference** for image classification and object detection while reducing power consumption.
   * Used in AI-powered **facial recognition** and **gesture recognition** systems in smart devices.
   * Optimizes **neural network processing** by speeding up matrix multiplications.
3. **Medical Image Processing**
   * Used in **X-ray, MRI, and CT scan processing** to enhance medical images while maintaining energy efficiency.
   * Helps in **pattern recognition for disease detection**, such as identifying tumors in medical scans.
   * Assists in **biomedical signal processing**, like ECG and EEG analysis.
4. **Augmented Reality (AR) & Virtual Reality (VR)**
   * Used in **real-time AR/VR rendering**, where energy-efficient processing is needed for smooth user experiences.
   * Enhances **gesture tracking and motion detection** in immersive environments.
5. **Mobile and Embedded Systems**
   * Useful in **smartphones, tablets, and wearables** for camera image processing, face unlocking, and video encoding.
   * Helps optimize **augmented reality applications** in mobile devices without draining the battery.
6. **FPGA & ASIC-Based Hardware Implementations**
   * Can be integrated into **custom hardware accelerators** for low-power, high-speed digital signal processing (DSP).
   * Useful in **energy-efficient hardware designs for AI and multimedia applications**.

# CONCLUSION

The efficiency of VLSI circuit design is primarily determined by three key factors: **area, power consumption, and delay**. Traditional multiplication architectures often face challenges in optimizing these parameters simultaneously. In contrast, the **proposed Hybrid Approximate Vedic Multiplier (HAVM)** presents a novel approach that significantly enhances computational efficiency while minimizing power consumption and hardware complexity.

By leveraging the **Urdhva Tiryagbhyam sutra** of Vedic Mathematics, the proposed HAVM generates **partial products in parallel**, thereby **eliminating redundant multiplication steps** and **accelerating the multiplication process**. This results in a **faster and more power- efficient computation** compared to conventional multipliers. Additionally, the HAVM's **two- stage operation**, which includes **partial product generation** followed by **approximate addition using NAND-based Half Adders (NHA) and Ripple Carry Adders (RCA)**, further reduces the overall computational burden.

One of the most significant advantages of this architecture is its **ability to lower power consumption and area requirements** while maintaining acceptable accuracy for specific applications. By reducing the number of **half adders and full adders**, the HAVM achieves an optimal balance between **hardware efficiency and computational speed**. This makes it **highly suitable for energy-constrained environments**, such as **real-time image and signal processing applications**, where slight inaccuracies are tolerable.

Compared to traditional multipliers, the proposed **8-bit HAVM demonstrates superior energy efficiency**, making it an ideal choice for **low-power, high-speed applications**. The integration of **NHA cells** in the Vedic architecture further enhances **power optimization and reduces latency**. Given its design advantages, the HAVM is particularly well-suited for **DSP applications, embedded systems, and AI-driven computing**, where speed and power efficiency are prioritized over absolute precision.

**FUTURE SCOPE**

The project "A Hybrid Approximate Multiplier for Energy Efficient Image Processing" has significant potential for future development and application, especially in fields requiring high-performance yet energy-efficient computation. Here are some key areas where this project could evolve:

1. **Enhanced Energy Efficiency**:
   * The demand for low-power computing, particularly in mobile devices, IoT, and edge computing, is growing. Improving the energy efficiency of image processing algorithms using hybrid approximate multipliers could result in substantial power savings, especially for real-time applications.
2. **AI and Machine Learning Integration**:
   * Combining approximate multipliers with machine learning models could lead to faster processing with lower power consumption. As AI increasingly relies on image data (e.g., in computer vision and object recognition), leveraging hybrid approximate multipliers could optimize both training and inference phases in AI systems.
3. **Applications in Embedded Systems**:
   * Hybrid approximate multipliers could be implemented in embedded systems like drones, medical imaging devices, and surveillance cameras, where low power and real-time image processing are crucial. This could help in achieving longer battery life and improving processing capabilities in resource- constrained devices.
4. **Optimization Algorithms for Approximation**:
   * The future of the project could involve the development of more sophisticated optimization algorithms for determining the best trade-off between approximation error and energy savings. These algorithms could be adaptive to specific application needs, offering more tailored energy savings without compromising image quality.
5. **Higher-Resolution Image Processing**:
   * While the focus might currently be on standard image resolutions, future iterations of the project could aim to scale up to high-definition or even 4K image processing. Research could be done on how hybrid approximate multipliers could handle the increased complexity of large images without significantly impacting energy consumption.
6. **FPGA and ASIC Implementations**:
   * To make the project practical for commercial and industrial use, developing FPGA or ASIC implementations of hybrid approximate multipliers could accelerate real-time image processing tasks in hardware, providing much lower power consumption compared to traditional software-based solutions.
7. **Edge Computing in IoT**:
   * In the realm of Internet of Things (IoT), where devices often need to process images locally (e.g., smart cameras, autonomous vehicles), hybrid approximate multipliers could be used to improve performance while reducing the energy burden on edge devices.
8. **Robustness and Error Tolerance**:
   * Further research could focus on improving the robustness of the system against errors, ensuring that approximation techniques do not lead to perceptible quality loss in images. Adaptive error correction techniques could also be developed to adjust the multiplier approximation based on the application.
9. **Cross-Disciplinary Applications**:
   * Hybrid approximate multipliers could be explored in other domains beyond image processing, like audio processing or signal processing, where similar trade-offs between precision and power consumption can be made.

In summary, the project has immense potential for improving energy efficiency in a wide range of image processing applications, and further research can expand its reach and performance in emerging technologies like AI, IoT, and embedded systems.

**REFERENCES**

1. G. Ganesh Kumar and V. Charishma, "Design of High Speed Vedic Multiplier using Vedic Mathematics Techniques," International Journal of Scientific and Research Publications, vol. 2, no. 3, pp. 1-5, 2012.
2. P. Verma and K. K. Mehta, "Implementation of an Efficient Multiplier based on Vedic Mathematics Using EDA Tool," International Journal of Engineering and Advanced Technology, vol. 1, no. 5, pp. 75-79, 2012.
3. M. Ramalatha, K. D. Dayalan, P. Dharani, and S. Deoborah, "High Speed Energy Efficient ALU Design using Vedic Multiplication Techniques," International Conference on Advances in Computational Tools for Engineering Applications, pp. 600-603, 2009.
4. A. A. M. Shamsuddin, M. F. Ain, and M. H. M. Yusof, "Design and Implementation of 8- bit Vedic Multiplier using VHDL," International Journal of Computer Applications, vol. 63, no. 16, pp. 975-8887, 2013.
5. S. S. Kerur, P. R. Arjun, R. S. Balaji, and S. G. Vijaya, "Implementation of Vedic Multiplier for Digital Signal Processing," International Journal of Computer Applications, vol. 19, no. 6, pp. 975-8887, 2011.
6. B. P. Soni and P. K. Jain, "Design and Implementation of Low Power Multiplier using Vedic Multiplication Technique," International Journal of Computer Science and Mobile Computing, vol. 3, no. 5, pp. 1281-1286, 2014.
7. R. Sridevi, A. Palakurthi, A. Sadhula, and H. Mahreen, "Design of a High Speed Multiplier (Ancient Vedic Mathematics Approach)," International Journal of Engineering Research and Applications, vol. 2, no. 3, pp. 383-386, 2012.
8. P. Mehta and D. Gawali, "Conventional versus Vedic Mathematical Method for Hardware Implementation of a Multiplier," International Conference on Advances in Computing, Control, and Telecommunication Technologies, pp. 640-642, 2009.
9. A. K. Singh and R. S. Gohil, "Design and Implementation of 32-bit Vedic Multiplier," International Journal of Engineering Research and Applications, vol. 4, no. 1, pp. 117- 120, 2014.