CHAPTER 1

INTRODUCTION

1.General

Now a days verification very important for VLSI designs. Manufacturing defects in the chip happen due to the interconnection of wires. This leads to unexpected outputs. The process of testing confirms that the chip is fault free. Before manufacturing we need to test the design so we need to apply test patterns for circuit under test. Basically, LCG and LFSR (linear feedback shift register) are used for test pattern generation, in this project we are selected LCG (Linear congruential generator).

1.1 EXISTING SYSTEM

In the existing system for detecting stuck-at faults in a full adder, conventional testing methods are employed, often relying on external tools such as Automated Test Equipment (ATE). These methods involve generating test vectors manually or using external pattern generators, applying them to the circuit, and comparing the actual output with the expected output. The stuck-at fault model is used to determine whether a signal is permanently stuck at logical 0 (stuck-at-0) or logical 1 (stuck-at-1). While this approach is effective in identifying faults, it has several limitations. These include high costs due to the reliance on external testing tools, scalability issues for larger systems, and time-consuming manual test pattern generation. Additionally, exhaustive testing increases power consumption, and achieving complete fault coverage is computationally expensive. These challenges make conventional methods less efficient and less practical for modern circuits, particularly as circuits become more complex and compact.

1.1 PROPSED SYSTEM

The proposed system introduces the use of Built-In Self-Test (BIST) for detecting stuck-at faults in a full adder, overcoming the limitations of the existing system. BIST integrates testing mechanisms directly into the circuit, eliminating the need for external testing tools. It consists of a Test Pattern Generator (TPG) that automatically generates optimized test vectors, a Response Analyzer (RA) to compare actual outputs with expected results, and a control unit to manage the testing process. This system ensures high fault coverage, as it can detect stuck-at-0 and stuck-at-1 faults in inputs, outputs, and internal nodes of the full adder. By automating the testing process, BIST reduces power consumption, speeds

up fault detection, and minimizes costs. It also supports real-time testing, making it scalable and adaptable for larger circuits and systems. This efficient, cost-effective, and reliable solution is particularly beneficial for VLSI circuits within the resulting logic capabilities History of Scale Integration The first transistors are invented in late 40s at Bell Labs, and in late 50s First IC (JK-FF by Jack Kilby at TI), followed in 60's SMALL SCALE INTEGRATION is done with two transistors. By following the Moore's integration law the integration of transistors has advanced and the MEDIUM. SCALE INTEGRATION has evolved with 100 transistors. In late 70's LARGE SCALE INTEGRATION has evolved with integration of 10,000 transistors. later VERY LARGESCALE INTEGRATION has evolved with more than 1 million transistors integration on a single chip.

1.3 ADVANTAGES OF IC'S OVER DISCRETE COMPONENETS

While we will deal with incorporated circuits, the homes of integrated circuits-what we will and cannot efficiently installed an incorporated circuit-in large part determine the structure of the whole system. Integrated circuits improve device traits in numerous crucial approaches. ICs have 3 key advantages over virtual circuits built from discrete components. Size Integrated circuits are a lot smaller- each transistor and wires are gotten smaller to micrometer sizes, compared to the millimeter or centimeter scales of discrete additives. Small length results in blessings in velocity and power consumption, considering that smaller components have smaller parasitic resistances, capacitances, and inductances.

Speed Signals can be switched between logic zero and logic 1 a great deal quicker inside a chip than they can between chips. Communication within a chip can occur loads of times faster than communication between chips on a printed circuit board. The excessive velocity of circuits on-chip is because of their small size-smaller additives and wires have smaller parasitic capacitances to slow down the signal.

Power Consumption operations inside a chip additionally take a whole lot much less power. Once more, decrease power intake is largely due to the small size of circuits on the chip-smaller parasitic capacitances and resistances require much less electricity to force them

VLSI AND SYSTEMS

These advantages of integrated circuits translate into advantages at the system level Smaller bodily length. Smallness is regularly a bonus in itself-do not forget transportable televisions or handheld mobile phones.

Lower power intake. Replacing a handful of popular parts with a single chip reduces total power consumption. Reducing strength intake has a ripple effect at the rest of the gadget: a smaller, cheaper power deliver can be used; for the reason that less power consumption approach less heat, a fan may additionally no longer be essential; a less difficult cupboard with much less protective for electromagnetic protective may be possible, too.

Reduced value. Reducing the wide variety of components, the strength supply requirements, cupboard charges, and so forth, will necessarily lessen system cost. The ripple effect of integration is such that the fee of a gadget built from custom ICs may be less, even though the individual ICs fee greater than the usual parts they replace.

CHAPTER 2

LITERATURE SURVEY

J. Zhou, Z. Cao, X. Dong, and A. V. Vasilakos, "Security and privacy for cloud-based IoT Challenges, The Internet of Things (IoT) is becoming the next Internet-related revolution. It allows billions of devices to be connected and communicate with each other to share information that improves the quality of our daily lives. On the other hand, Cloud Computing provides on-demand, convenient and scalable network access which makes it possible to share computing resources, indeed, this, in turn, enables dynamic data integration from various data sources. There are many issues standing in the way of the successful implementation of both Cloud and IoT. The integration of Cloud Computing with the IoT is the most effective way on which to overcome these issues. The vast number of resources available on the Cloud can be extremely beneficial for the IoT, while the Cloud can gain more publicity to improve its limitations with real world objects in a more dynamic and distributed manner. This paper provides an overview of the integration of the Cloud into the IoT by highlighting the integration benefits and implementation challenges. Discussion will also focus on the architecture of the resultant Cloud-based IoT paradigm and its new applications scenarios. Finally, open issues and future research directions are also suggested.

E. Zenner, "Cryptanalysis of LFSR-based pseudorandom generators—A survey," Pseudorandom bit generator (PRBG) is an essential component for securing data during transmission and storage in various cryptography applications. Among popular existing PRBG methods such as linear feedback shift register (LFSR), linear congruential generator (LCG), coupled LCG (CLCG), and dual-coupled LCG (dual-CLCG), the latter proves to be more secure. This method relies on the inequality comparisons that lead to generating pseudorandom bit at a non-uniform time interval. Hence, a new architecture of the existing dual CLCG method is developed that generates pseudo-random bit at uniform clock rate. However, this architecture experiences several drawbacks such as excessive memory usage and high-initial clock latency, and fails to achieve the maximum length sequence. Therefore, a new PRBG method called as "modified dual-CLCG" and its very large-scale integration (VLSI) architecture are proposed in this paper to mitigate the aforesaid problems. The novel contribution of the proposed PRBG method is to generate pseudorandom bit at uniform clock rate with one initial clock delay and minimum hardware complexity. Moreover, the proposed PRBG method passes all the 15 benchmark tests of NIST standard and achieves the maximal period

of 2^n . The proposed architecture is implemented using Verilog-HDL and prototyped on the commercially available FPGA device

J. Stern, "Secret linear congruential generators are not cryptographically secure," The dual-coupled-linear congruential generator (LCG) (dual-CLCG) is a secure pseudorandom bit generator (PRBG) method

among various linear feedback shift register (LFSR), LCG, and chaotic-based PRBG methods for generating a pseudorandom bit sequence. The hardware implementation of this method has a bottleneck due to the involvement of inequality equations. Initially, a direct architectural mapping of the dual-CLCG method is performed. Since two inequality equations are involved for coupling, it generates pseudorandom bit at unequal interval of time that leads to large variation in output latency. In addition, it consumes a large area and fails to achieve the maximal period. Hence, to overcome the aforesaid drawbacks, a new efficient PRBG method, i.e., "coupled-variable input LCG (CVLCG)," and its architecture are proposed. The novelty of the proposed method is the coupling of two newly formed variable input LCGs that generates pseudorandom bit at every uniform clock rate, attains maximum length sequence, and reduces one comparator area as compared to the dual-CLCG architecture. The proposed architecture is implemented using Verilog-HDL and prototyped on the commercially available field-programmable gate array (FPGA) device. Furthermore, the sequences are captured through the logic analyzer and evaluated for randomness using the National Institute of Standard and Technology (NIST) standard test tool. The experimental result reports that the proposed PRBG method passes all the randomness tests with a high degree of consistency.

W. B. Jone and D. C. Huang and S. C. Wu and K. J. Lee, An efficient BIST method for small buffers, *Astonishing progress in Silicon devices and circuits and highly reliable mass manufacturing techniques have prompted unprecedented revolutions in electronics for the last 4 decades to the extent that electronics is growingly permeating numerous aspects of our life. The application world, consumer and infrastructure, is now used to exponential performance improvements and high yield. Complexity and competitiveness of modern electronic systems demand for optimization across multiple disciplines. Joint optimization of device, circuit, packaging, and test are increasingly important for high performance systems. Design for Manufacturing and Testing is more critical but increasingly challenging in complex systems. So what does optimization mean for the future of ever-growing and complex solid state electronics? A daunting challenge worth spending a talk on.*

IEEE Standard 1687-2014 - IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device, A methodology for accessing instrumentation embedded within a semiconductor device, without defining the instruments or their features themselves, via the IEEE 1149.1(TM) test access port (TAP) and/or other signals, is described in this standard. The elements of the methodology include a hardware architecture for the on-chip network connecting the instruments to the chip pins, a hardware description language to describe this network, and a software language and protocol for communicating with the instruments via this network. This standard develops a methodology for access to embedded instrumentation, without defining the instruments or their features themselves, via the IEEE 1149.1(TM) test access port (TAP) and additional signals that may be required. The elements of the methodology include

a description language for the characteristics of the features and for communication with the features, and requirements for interfacing to the features.

D. Bronzi, Y. Zou, F. Villa, S. Tisa, A. Tosi, and F. Zappa, "Automotive three-dimensional vision through a single-photon counting SPAD camera," Linear-feedback shift register (LFSR) counters have been shown to be well suited to applications requiring large arrays of counters and can improve the area and performance compared with conventional binary counters. However, significant logic is required to decode the count order into binary, causing system-on-chip designs to be unfeasible. This paper presents a counter design based on multiple LFSR stages that retains the advantages of a single-stage LFSR but only requires decoding logic that scales logarithmically with the number of stages rather than exponentially with the number of bits as required by other methods. A four-stage four-bit LFSR proof of concept was fabricated in 130-nm CMOS and was characterized in a time-to-digital converter application at 800MHz.

Vornicu, R. Carmona-Galán, and A. Rodríguez-Vázquez, "A CMOS 0.18μm 64×64 single photon image sensor with in-pixel 11 bimetal- digital converter, "Linear-feedback shift register (LFSR) counters have been shown to be well suited to applications requiring large arrays of counters and can improve the area and performance compared with conventional binary counters. However, significant logic is required to decode the count order into binary, causing system-on-chip designs to be unfeasible. This paper presents a counter design based on multiple LFSR stages that retains the advantages of a single-stage LFSR but only requires decoding logic that scales logarithmically with the number of stages rather than exponentially with the number of bits as required by other methods.

H. Mo and M. P. Kennedy, "Masked dithering of MASH digital delta sigma modulators with constant inputs using multiple linear feedback shift registers," This paper shows that applying a

linear feedback shift register (LFSR) dither to a digital delta-sigma modulator (DDSM) cannot always increase its fundamental period. For some DDSMs, the LFSR dither may reduce its period in some cases, instead of increasing it, which worsens the output spectrum. Hence, the paper calculates the dithered DDSM's period and analyzes the influence of LFSR dither on the period. Furthermore, for such kind of DDSM, the paper explains how to add the LFSR dither to increase the period for a full input range. Finally, experiment is performed to confirm the analysis.

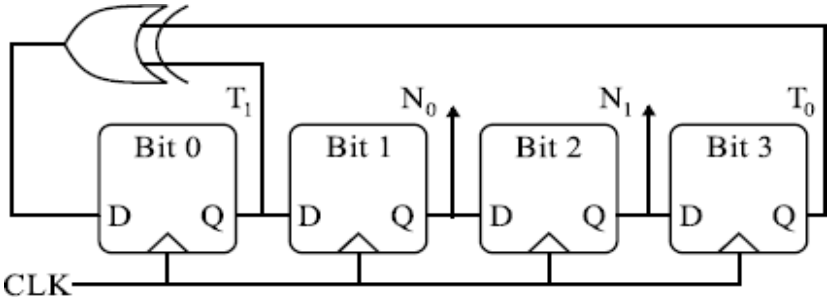
CHAPTER 3

PROJECT DESCRPTION

3.1 LINEAR-FEEDBACK SHIFT REGISTER (LFSR)

In computing, a linear-feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state. The most commonly used linear function of single bits is exclusive-or (XOR). Thus, an LFSR is most often a shift register whose input bit is driven by the XOR of some bits of the overall shift register value. The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and has a very long cycle.

Applications of LFSRs include generating pseudo-random numbers, pseudo-noise sequences, fast digital counters, and whitening sequences. Both hardware and software implementations of LFSRs are common. The mathematics of a cyclic redundancy check, used to provide a quick check against transmission errors, are closely related to those of an LFSR LFSRs can be implemented in hardware, and this makes them useful in applications that require very fast generation of a pseudo-



random sequence, such as direct-sequence

Fig3: Structure of conventional many-to-one 4-LFSRs

spread spectrum radio. LFSRs have also been used for generating an approximation of white noise in various programmable sound generators.

Uses as counter:

The repeating sequence of states of an LFSR allows it to be used as a clock divider or as a counter when a non-binary sequence is acceptable, as is often the case where computer index or

framing locations need to be machine-readable.[7] LFSR counters have simpler feedback logic than natural binary counters or Gray-code counters, and therefore can operate at higher clock rates. However, it is maximal periods. One can obtain any other period by adding to an LFSR that has a longer period.

3.2 CRYPTOGRAPHY

LFSRs have long been used as pseudo-random number generators for use in stream ciphers (especially

in military cryptography), due to the ease of construction from simple electromechanical or electronic circuits, long periods, and very uniformly distributed output streams. However, an LFSR is a linear system, leading to fairly easy cryptanalysis. For example, given a stretch of known plaintext and corresponding cipher text, an attacker can intercept and recover a stretch of LFSR output stream used in the system described, and from that stretch of the output stream can construct an LFSR of minimal size that simulates the intended receiver by using the Berlekamp-Massey algorithm. This LFSR can then be fed the intercepted stretch of output stream to recover the remaining plaintext.

3.2 CIRCUIT TESTING

LFSRs are used in circuit testing for test-pattern generation (for exhaustive testing, pseudo-random testing or pseudo-exhaustive testing) and for signature analysis.

Test-pattern generation:

Complete LFSR are commonly used as pattern generators for exhaustive testing, since they cover all possible

inputs for an n-input circuit. Maximal-length LFSRs and weighted LFSRs are widely used as pseudo- random test-pattern generators for pseudo-random test applications.

Signature analysis:

In built-in self-test (BIST) techniques, storing all the circuit outputs on chip is not possible, but the circuit

output can be compressed to form a signature that will later be compared to the golden signature

(of the good circuit) to detect faults. Since this compression is lossy, there is always a possibility

BIST is accomplished with a multiple-input signature register (MISR or MSR), which is a type of LFSR. A standard LFSR has a single XOR or XNOR gate, where the input of the gate is connected to several "taps" and the output is connected to the input of the first flip-flop. A MISR has the same structure, but the input to every flip-flop is fed through an XOR/XNOR gate. For example, a 4-bit MISR has a 4-bit parallel output and a 4-bit parallel input. The input of the first flip-flop is XOR/XNORd with parallel input bit zero and the "taps". Every other flip-flop input is XOR/XNOR d with the preceding flip-flop output and the corresponding parallel input bit. Consequently, the next state of the MISR depends on the last several states opposed to just the current state. Therefore, a MISR will always generate the same golden signature given that the input sequence is the same every time.

Uses in digital broadcasting and communications:

To prevent short repeating sequences (e.g., runs of 0s or 1s) from forming spectral lines that may complicate symbol tracking at the receiver or interfere with other transmissions, the data bit sequence is combined with the output of a linear-feedback register before modulation and transmission. This scrambling is removed at the receiver after demodulation. When the LFSR runs at the same bit rate as the transmitted symbol stream, this technique is referred to as scrambling. When the LFSR runs considerably faster than the symbol stream, the LFSR-generated bit sequence is called chipping code. The chipping code is combined with the data using exclusive or before transmitting using binary phase-shift keying or a similar modulation method. The resulting signal has a higher bandwidth than the data, and therefore this is a method of spread- spectrum communication. When used only for the spread-spectrum property, this technique is called direct- sequence spread spectrum; when used to distinguish several signals transmitted in the same channel at the same time and frequency, it is called code division multiple access. Neither scheme should be confused with encryption or encipherment; scrambling and spreading with LFSRs do not protect the information from eavesdropping. They are instead used to produce equivalent streams that possess convenient engineering properties to allow robust and efficient modulation and demodulation.

Digital broadcasting systems that use linear-feedback registers

ATSC Standards (digital TV transmission system – North America), DAB (Digital Audio Broadcasting system – for radio) DVB-T (digital TV transmission system – Europe, Australia, parts of Asia) NICAM (digital audio system for television)

Other digital communications systems using LFSRs:

INTELSAT business service (IBS) Intermediate data rate (IDR) SDI (Serial Digital Interface transmission)

Data transfer over PSTN (according to the ITU-T V-series recommend

CDMA (Code Division Multiple Access) cellular telephony 100BASE-T2 "fast" Ethernet scrambles bits using an LFSR

1000BASE-T Ethernet, the most common form of Gigabit Ethernet, scrambles bits using an LFSR PCI

Express 3.0 SATA

Serial attached SCSI (SAS/SPL) USB 3.0

FIG 3.1 Architecture of linear congruential generator

3.3 LINEAR CONGRUENTIAL GENERATOR

In fact, any higher-level programming language offers at least one form of random number generator. The generation of random numbers, however, is not an easy task for a computer, since the computer is a deterministic machine with no built-in randomness. Thus, it is impossible to create true random numbers without any additional hardware. What can be done, is to create **pseudo random numbers** which behave almost like random numbers but which are repeated after a fixed (mostly quite long) period.

These pseudo random numbers are generated by **linear congruential generators** (**LCG**). The principle of an LCG is quite simple: a new pseudo random number is generated on the basis of the previous random number by adding a certain offset and wrapping the result if it exceeds a certain limit.

The process can be denoted by the following equation, The linear congruential method produces a sequence of integers X0,X1,X2… between zero and *m-1* according to the following recursive relationship: $x_{i+1} = a_1 \times x_i + b_1 \bmod 2^n$ (1)

Here, a1, b1 are the constant parameters; $x_i$is the initial seed. Following are the necessary conditions to get the maximum period. b1 is relatively prime with 2^n(m).

a1-1) must be divisible by 4.

LCG method is developed to generate pseudorandom bit at an equal interval of time for encrypting continuous data stream in the stream cipher the architecture is designed with two comparators, four LCG blocks, one controller unit and memory (flip-flops) as shown in Fig. 1. The LCG is the basic functional block in the dual-CLCG architecture that involves multiplication and addition processes to compute n-bit binary random number on every clock cycle.

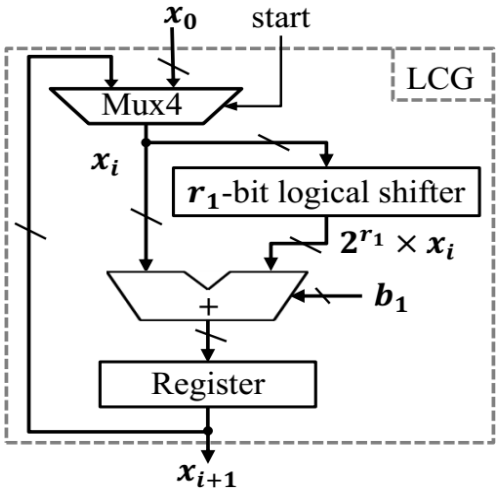


FIG 3.1 Architecture of linear congruential generator

The multiplication in the LCG equation can be implemented with shift operation, when a is considered as (2^r+1). Here, r is a positive integer, $1 < r < 2^n$. Therefore, for the efficient computation of xi+1, the equation (1) can be rewritten as,

$xi+1 = (a1 \times xi + b1) \bmod 2^n = [(2^{r1} + 1) xi + b1] \bmod 2^n$

$=[(2^{r1} \times xi) + xi + b1] \bmod 2^n$

The architecture of LCG shown in Fig. 3 is implemented with a 3-operand modulo 2n adder, 2 × 1 n-bit multiplexer and n-bit register. LCG generates a random n-bit binary equivalent to integer number in each clock cycle. Other three LCG equations can also be mapped to the corresponding architecture similar to the LCG equation

3.4 FAULT TOLERANT TESTING ALGORITHM FOR FULL ADDER

The evolution of integrated circuits started with the introduction of the microprocessor. Proper functioning of a chip is tested before it is manufactured. Various types of the verification involved in the process of chip testing are IP verification, RTL verification, timing verification, etc., Testing becomes a headache in the process of IC fabrication as it consumes 80% of the manufacturing time. A faulty product is generated due to incomplete specification problem, incorrect design, faulty fabrication process, and wrong test method. Therefore, it is necessary to identify the locations at which the faults are present. Adder simulate a fault to any one or more lines of the circuit for testing a chip. There are various types of faults occur in the circuit, they are Stuck-at-Fault, Delay Fault, Transient Fault, Bridging Fault, and so on. These types of faults may occur either on the input or output side of the circuit.

Testing is normally done using a computer with the device under test (DUT). An algorithm developed using a high-level language generates test vectors. These test vectors are in such a way that it provides all the possible combination of inputs that are applied to the DUT. A suitable data transfer medium transfers the test vectors to the DUT and receives the outputs generated by it. Now, the algorithm analyzes the outputs and identifies the presence of the fault(s). The testing time depends on the number of test vectors [4-5]. Whereas, the number of test vectors is based on the possible combinations of inputs that are applied to the DUT. Hence, an optimistic algorithm always tries to reduce the total number of test vectors and hence the testing time. This work has attempted to establish testing using a field programmable gate array (FPGA). The two types of stuck-at-faults are Stuck-at-1 (SA1) and Stuck-at-0 (SA0). Stuck-at fault
identifies the faults that are available in an integrated circuit. It can occur either at the input to the gate or at the output of the gate. When the value of the line or cell is always '0', it is identified as astuck-at-0 fault

(SA0). When the value of the line or cell is always '1', it is identified as a stuck-at-1 fault (SA1). This stuck- at fault model also helps to identify the manufacturing defect of a logic gate itself. Faults in the chip are identified through fault simulation. A typical testing process involves the generation of test vectors or test patterns for the circuit which is to be tested. These test vectors are applied as inputs to the circuit. Outputs of the circuit are verified with the expected outputs. Faults identification are decided based on the result of the comparison. The basic principle of fault simulation is shown in Fig. 1.

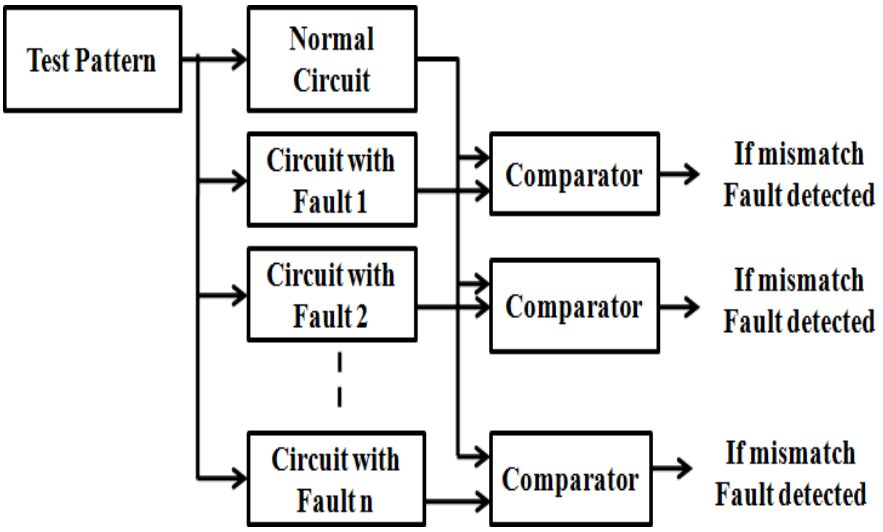


Fig 3.2 Basic Principle of Fault Simulation

Different kinds of algorithms used for fault simulation are serial fault simulation, parallel fault simulation, concurrent fault simulation, and deductive fault simulation.

This work concentrates on parallel fault simulation. Parallel fault simulation is the most effective technique when a circuit is designed using logic gates and modeled with stuck-at- faults. The testing flow diagram shown in Fig. 2. The flow will continue till find a required test vectors.

This work uses the application and testing algorithm in the same chip. It can be re- configured at any point of time with the help of hardware description languages (HDL). The bit file generated after synthesis is implemented into the Xilinx. Xilinx realizes the HDL coding in the form of logic gates.

Path sensitizing method of testing has been a powerful approach for test generation of any combinational circuit. Path Sensitization is based on the assumption that the failure mechanism in a gate result in its inputs or outputs being stuck-at-1 or stuck-at-0. This fault is detected along a path to the circuit output and its effects are noted

This impact determines the occurrence of a fault. It also helps to identify how the fault is propagated to the output by means of a sensitized path or path from the fault through the associated gate. Realization of path sensitizing in a full adder circuit is shown in Fig 3. Path sensitization can be done through the Boolean difference method [9-10].

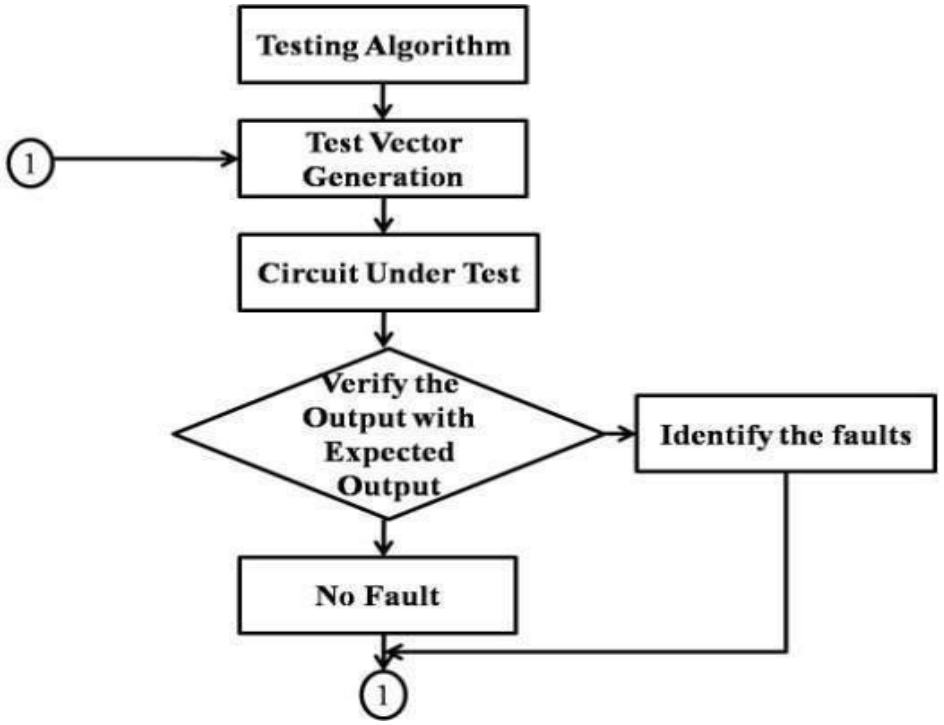


Fig. 3.3 Testing Flow Diagram

The Boolean difference method is a well-known mathematical concept which has found significant application in the single fault analysis of combinational logic circuits. Full adder circuit has two outputs as 'sum' and 'carry'. Consider the output 'sum' which has an input of a, b, and c.

If any one of the inputs of the full adder circuit has stuck-at-faults, then the output 'sum' may lead to erroneous output. Now, let line 1 has a stuck-at-0 fault. Boolean difference method helps to identify this fault.

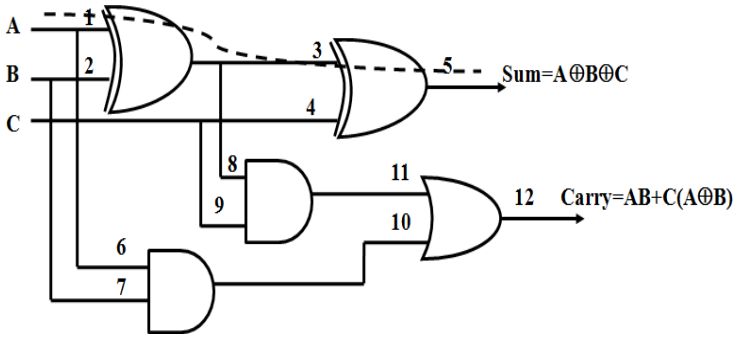


Fig. 3.4 Path Sensitizing in a Full adder circuit

A fault present in the circuit is represented as stuck-at-0 (SA0) and stuck-at-1(SA1). Most of the faults found in VLSI design/ circuits are stuck at fault because the electrical failures occur due to the physical mechanism of the circuit. So, concern stuck at fault which occurs due to broken wires, short to ground or to Vcc, shorted transistors and diodes. The use of all possible inputs for testing is known as exhaustive testing and it is straight forward, but impractical. This can be solved by using analytical modeling of a circuit. Determine the output of the circuit for all possible inputs by forming a fault table. Fault table displays the output of the circuit for a set of faults and fault-free test inputs. The process of determining a test for a given fault in a circuit is known as Test Generation.

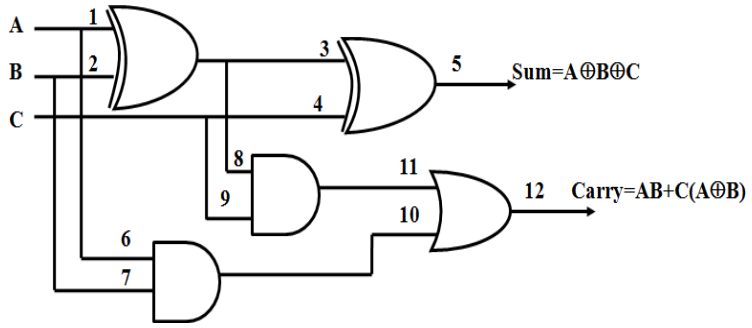


Fig. 3. Single bit Full adder

The Boolean difference method is applied and tested for a full adder circuit. A set of test vectors to detect single stuck-at faults are generated. All the possible faults of full adder were determined and executed using Xilinx 14.7 ISE design Suite. Each line of full adder was analyzed for either stuck-at- 1 or stuck-at-0 and the required test vectors to find multiple faults were identified as test vectors.

CHAPTER 4

SOFTWARE USED VERILOG

4.1INTRODUCTION

- Verilog synthesis gear can create logic-circuit structures without delay from Verilog behavioral description and goal them to a particular generation for attention (IE,translate Verilog to actual hardware).
- Using Verilog, we will layout, simulate and synthesis something from a simple combinational circuit to a entire microprocessor on chip.
- Verilog HDL has evolved as a preferred hardware description language. Verilog HDL offers many beneficial capabilities for hardware layout.
- Verilog HDL is a trendy-reason hardware description language that is straightforward to analyze and easy to use. It is similar in syntax to the C programming language. Designers with C programming enjoy will locate it smooth to research Verilog HDL.
- Verilog HDL permits distinctive levels of abstraction to be combined within the equal version. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer wishes to examine handiest one language for stimulus and hierarchical design.
- Most popular common sense synthesis tools support Verilog HDL. This makes it the language of desire for designers.
- All fabrication vendors offer Verilog HDL libraries for put-up common-sense synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of carriers.
- The Programming Language Interface (PLI) is an effective function that allows the user to jot down custom C code to engage with the inner facts systems of Verilog. Designers can customize a Verilog HDL simulator to their desires with the PLI.

History Of Verilog HDL

Verilog turned into initial of all as a proprietary hardware modeling language by using Gateway Design Automation Inc. Round 1984. It is rumored that the original language turned into designed by way

of taking features from the maximum popular HDL language of the time, known as Hilo, as well as from traditional pc languages which includes C. At that point, Verilog become not standardized and the language changed itself in nearly all of the revisions that got here out inside 1984 to 19900 Verilog synthesis gear can create logic-circuit structures without delay from Verilog behavioral description and goal them to a particular generation for attention (IE, translate Verilog to actual hardware).

- Using Verilog, we will layout, simulate and synthesis something from a simple combinational circuit to an entire microprocessor on chip.
- Verilog HDL has evolved as a preferred hardware description language. Verilog HDL offers many beneficial capabilities for hardware layout.
- Verilog HDL is a trendy-reason hardware description language that is straightforward to analyze and easy to use. It is similar in syntax to the C programming language. Designers with C programming enjoy will locate it smooth to research Verilog HDL.
- Verilog HDL permits distinctive levels of abstraction to be combined within the equal version. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer wishes to examine handiest one language for stimulus and hierarchical design.
- Most popular common sense synthesis tools support Verilog HDL. This makes it the language of desire for designers.
- All fabrication vendors offer Verilog HDL libraries for put-up common-sense synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of carriers.
- The Programming Language Interface (PLI) is an effective function that allows the user to jot down custom C code to engage with the inner facts systems of Verilog. Designers can customize a Verilog HDL simulator to their desires with the PLI.
- The time was late 1990. Cadence Design System, whose primary product at that point blanketed thin film manner simulator, determined to accumulate Gateway Automation System. Along with other Gateway merchandise, Cadence now have become the proprietor of the Verilog language, and continued to marketplace Verilog as each a language and a simulator. At the same time, Synopsys was advertising the pinnacle-down layout method, the usage of Verilog.

- In 1990, Cadence recognized that if Verilog remained a closed language, the pressures of standardization would subsequently purpose the industry to shift to VHDL. Consequently, Cadence organized the Open Verilog International (OVI), and in 1991 gave it the documentation for the Verilog Hardware Description Language. This was the event which "opened" the language.
- OVI did a large number of paintings to enhance the Language Reference Manual (LRM), clarifying matters and making the language specification as dealer-unbiased as viable. Soon it was found out that if there have been too many organizations in the marketplace for

  Verilog, probably each person would like to do what Gateway had performed to date - changing the language for his or her very own benefit. This might defeat the main reason of freeing the language to public area. As a result in 1994, the IEEE 1364 operating institution become shaped to show the OVI LRM into an IEEE preferred. This attempt turned into concluded with a successful ballot in 1995, and Verilog have become an IEEE trendy in December 1995.
- When Cadence gave OVI the LRM, numerous agencies began running on Verilog simulators. In 1992, the primary of those have been announced, and by 1993 there have been numerous Verilog simulators available from organizations other than Cadence. The maximum a hit of these became VCS, the Verilog Compiled Simulator, from Chronologic Simulation. This changed into a true compiler in place of an interpreter, that's what Verilog-XL was. As an end result, collect time changed into full-size, however simulation execution velocity changed into tons quicker.

  In the interim, the recognition of Verilog and PLI became growing exponentially. Verilog as a HDL found greater admirers than well-shaped and federally funded VHDL. It was most effective a be counted of time earlier than humans in OVI found out the want of a more universally general trendy. Accordingly, the board of directors of OVI asked IEEE to shape a operating committee for setting up Verilog as an IEEE trendy. The running committee 1364 was shaped in mid-1993 and on October 14,1993, it had its first meeting.

  The general, which blended each the Verilog language syntax and the PLI in a single volume, turned into passed in May 1995 and now referred to as IEEE Std. 1364-1995.
- After a few years, new capabilities had been introduced to Verilog, and the new version is referred to as Verilog 2001. This version seems to have fixed a variety of issues that Verilog 1995 had.

4.2 PROGRAM STRUCTURE

A Verilog module has declarations that describes the names and types of the module inputs and outputs as well as local signals, variables, constants and functions that are used internally to the module are not visible outside.

The rest of the module contains statements that specify the operation of the module output and internal signals.

Verilog is a case-sensitive language like C. Thus sense, Sense, SENSE,etc are all treated as different entities / quantities in Verilog.

SYNTAX:

Module Module_Name(port list); Port declaration Function declaration Endmodule

module signifies the beginning of a module definition.
endmodule signifies the end of a module definition.

IDENTIFIERS:

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

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

For example,

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

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

Examples

\b=c

\Control-signal

\&logic

\ABC // Here the combination "ABC" forms the identifier.

WHITE SPACE CHARACTERS

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

COMMENTS

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

PORT DECLERATION:

Verilog module declaration begins with a keyword "module" and ends with "end module". The input and output ports are signals by which the module communicates with each other's.

Syntax:

Input identifier……..identifier; Output identifier identifier Input identifier identifier;

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

LOGIC SYSTEM:

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

0-LOGIC 0, OR FALSE

1-LOGICAL 1 OR FALSE

X-A UNKNOWN LOGICAL VALUE

Z- HIGH IMPEDENCE

4.2 OPERATORS

Arithmetic Operators:

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

+ (addition)

- (subtraction)

* (multiplication)

/ (division)

% (modulus)

Relational Operators:

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

< (less than)

<= (less than or equal to)

> (greater than)

>= (greater than or equal to)

== (equal to)

!= (not equal to)

Bit-wise Operators:

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

~ (bitwise NOT)

& (bitwise AND)

| (bitwise OR)

^ (bitwise XOR)

~^ or ^~ (bitwise XNOR)

Logical Operators:

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

! (logical NOT)

& (logical AND)

|| (logical OR

Reduction Operators

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

& (reduction AND)

| (reduction OR)

~& (reduction NAND)

~| (reduction NOR)

^ (reduction XOR)

~^ or ^~ (reduction XNOR)

SHIFT OPERATORS

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

<< (shift left)

>> (shift right)

Concatenation Operator:

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

{ } vector (concatenation)

Replication Operator:

The replication operator makes multiple copies of an item.

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

Literals:

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

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

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

Number Syntax

n’Fddd..., where n - integer representing number of bits F - one of four possible base formats: b (binary), o (octal), d (decimal),h (hexadecimal). Default is d.

Literals written without a size indicator default to 32-bits or the word width used by the simulator Program.**ET**:

Verilog actually has two classes of signals Nets and variables.

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

triand, trior, tri reg, etc. The wire declaration is used most frequently.

- The syntax of Verilog net declaration is similar to an input/output declaration.

Syntax:

Wire identifier, identifier;

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

VARIABLE:

- Verilog variables store the values during the program execution, and they need not have Physical significance in the circuit.
- They are used in only procedural code (i.e,behavioral design).A variable value can be used ina expression and can be combined and assign to other variables, as in conventional software programming language.
- The most commonly used variables are REG and INTEGERS. Syntax:

  Reg identifier, identifier;

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

  Integer identifier, identifier;

- A register variable is a single bit or vector of bits ,the value of 1-bit reg variable is always 0,1,X,Z.

the main use of reg variables is to store values in Verilog procedural code.

An integer variable value is a32-bit or larger integer depending on the word length on the word length used by simulator an integer variable is typically used to control a repetitive statements, such as loop, in Verilog procedural code.

PARAMETER:

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

Syntax:

Parameter identifier =value; Parameter identifier =value,

: :

identifier =value;

- An identifier is assigned to a constant value that will be used in place of the identifier throughout the current module.
- Multiple constants can be defined in a single parameter declaration using a comma – separated list of arguments.

- The value in the parameter declaration can be simple constant or it can be a constant expression.
- An expression involving multiple operators and constants including other parameters that yields a constant result at compile time. The parameter scope is limited to that module in which it is defined.

ARRAYS:

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

Syntax:

Reg identifier [start:end];

Reg [msb:lsb] identifier [start:end]; Integer identifier [start:end] ; Example: integer count [0: 7] ; I I An array of 8 count variables

4.3 DATAFLOW DESIGN ELEMENTS:

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

Syntax:

Assign net-name=expression;

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

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

Assign net-concatenation =expression;

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

For example:

assign c = a && b;

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

4.4 STURCTURAL DESIGN (OR) GATE LEVEL MODELING

- Structural Design Is the Series of Concurrent Statement. The Most Important Concurrent Statement In the module covered like instance statements, continuous –assignment statement and always block. These gives rise to three distinct styles of circuit design and description.
- Statement of these types, and corresponding design styles, can be freely intermixed within a Verilog module declaration

- Structural Design Is the Series of Concurrent Statement. The Most Important Concurrent Statement In the module covered like instance statements, continuous –assignment statement and always block. These gives rise to three distinct styles of circuit design and description.
- Statement of these types, and corresponding design styles, can be freely intermixed within a Verilog module declaration.
- Each concurrent statement in a Verilog module "executes" simultaneously with other statements in the same module declaration.
- In Verilog module, if the last statement updates a signal that is used by the first statement, then the simulator goes back to that first statement and updates its result.
- In fact, the simulator will propagate changes and updating results until the simulated circuit stabilizes

Syntax of Verilog instance ststements:

Component_name

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

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

: :

.port-name(expression));

Basic gate primitives in Verilog with details:

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

4.5 BEHAVIORAL MODELING

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

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

Structured Procedures:

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

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

Always:

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

Syntax:

1). Always @(signal-name signal-name) Procedural statement
2). Always procedural statements

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

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

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

For Loops

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

Syntax:

for (count = value1; count </<=/>/>= value2; count = count +/- step) begin
... statements ... End

While Loops:

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

Syntax:

while (expression) begin

... statements ... End

CASE:

The case statement allows a multipath branch based on comparing the expression with a list of case choices. Statements in the default block executes when none of the case choice comparisons are true. With no default, if no comparisons are true, synthesizers will generate unwanted latches. Good practice says to make a habit of putting in a default whether you need it or not.
If the defaults are don't cares, define them as 'x' and the logic minimizer will treat them as don't cares and d save area. Case choices may be a simple constant, expression, or a comma-separated list of same.

Syntax

case (expression) case_choice1: begin
... statements ... end case_choice2: begin
... statements ... end
... more case choices blocks ... default:
Begin
... statements ... end endcase

casex:

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

Casez:

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

FOREVER LOOPS

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

Syntax forever begin

... statements ...

Ends Example forever begin

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

REPEAT:

The repeat statement executes a statement or blocks of statements a fixed number of times. repeat CONSTRUCT The repeat construct is used to repeat a specified block a specified number of times. The quantity a can be a number or an expression evaluated to a number. As soon as the repeat statement is

encountered, a is evaluated. The following block is executed "a" times. If "a" evaluates to 0 or x or z, the block is not executed.

Syntax:

repeat (number_of_times) begin

... statements ... End

CHAPTER 5 IMPLEMENTATION

DESIGN CODE

```
module fault_tolernt_bist(clk,rst,lfsr_out ,tsum,tcarry,Esum,Ecarry , Eqout ,fault );
input clk,rst;
output fault ;
output [2:0] lfsr_out ;
output tsum,tcarry ;
output  Esum,Ecarry ;
output  Eqout ;
LFSR prg(.clk(clk),.rst(rst),.lfsr_out(lfsr_out));
full_adder inst1(.A(lfsr_out[2]),.B(lfsr_out[1]),.C(lfsr_out[0]),.Sum(tsum),.Carry(tcarry));
expected_results inst2(.a(lfsr_out[2]),.b(lfsr_out[1]),.cin(lfsr_out[0]),.sum(Esum),.carry(Ecarry));
comparator  inst3(.a({tsum,tcarry}),.b({Esum,Ecarry}),.Eqout(Eqout));
assign fault = Eqout ? 1'b0 : 1'b1 ;
endmodule
//expected output
module expected_results(a,b,cin,sum,carry);
input a,b,cin;
output reg sum,carry;
always @*
begin
        case({a,b,cin})//001
        3'd1 : begin
                                sum = 1 ;
                                carry = 0 ;
                                end
        3'd2 : begin
                                sum = 1 ;
                                carry = 0 ;
                                end
        3'd3 : begin
                                sum = 0 ;
                                carry = 1 ;
                                end
```

```
        3'd4 : begin
                                sum = 1 ;
                                carry = 0 ;
                                end
        3'd5 : begin
                                sum = 0 ;
                                carry = 1 ;
                                end
        3'd6 : begin
                                sum = 0 ;
                                carry = 1 ;
                                end
        3'd7 : begin
                                sum = 1 ;
                                carry = 1 ;
                                end
        default  begin
                                sum  0 ;
                                carry= 0 ;
                                end
        endcase
end
endmodule
//cut (full adder)
module full_adder(A,B,C,Sum,Carry);
input A,B,C;
output Sum,Carry ;
xor g1(S1,A,B);////xor
and g2(w1,A,B);
and g3(w2,S1,C);
xor g4(Sum, S1,C);
or g5(Carry,w1,w2);
endmodule
```

```
//comparator
module comparator (a,b,Eqout);
input [1:0] a,b;
output Eqout ;
assign Eqout = a == b ;
endmodule

//lfsr
module LFSR(clk,rst,lfsr_out);
input clk ,rst ;
output reg [2:0] lfsr_out ;
assign fb = lfsr_out[0] ^ lfsr_out[2] ;
always @(posedge clk or posedge rst )
begin
        if(rst)lfsr_out <= 3'b001 ;

        else lfsr_out <= { fb , lfsr_out[2:1] } ;
end
endmodule
```

TESTBENCH CODE

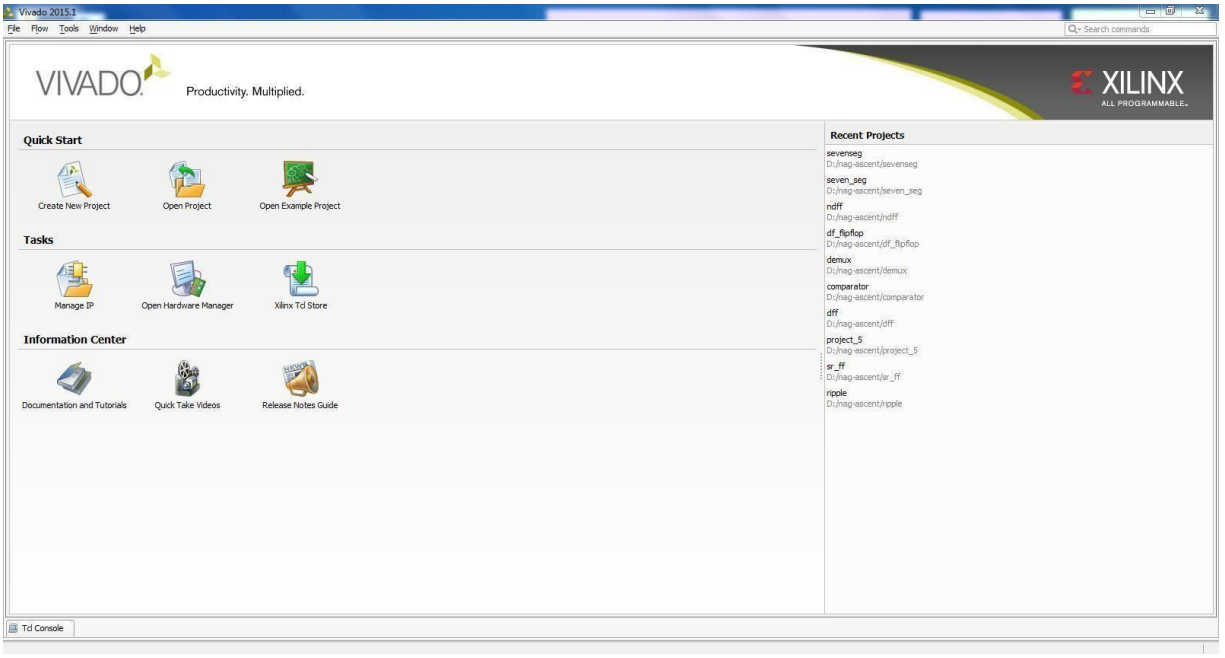
```
`timescale 1ns / 1ps module bist_tb;
// Inputs reg clk=0; reg rst=1;


// Outputs

wire [2:0] lfsr_out; wire fault;
wire tsum,tcarry,Esum,Ecarry , Eqout ;

// Instantiate the Unit Under Test (UUT)

fault_tolernt_bist uut(clk,rst,lfsr_out ,tsum,tcarry,Esum,Ecarry , Eqout ,fault );
always #5 clk=!clk ; initial #10 rst =0; endmodule
```

CHAPTER 6

SIMULATION AND CIRCUIT DESIGN

VIVADO

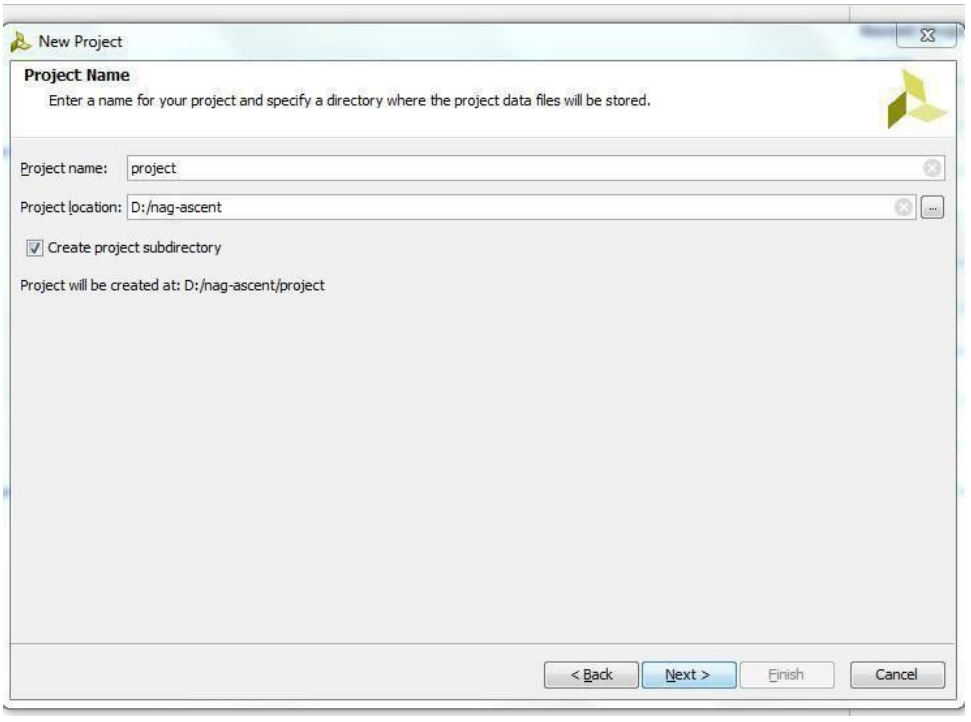
Xilinx Vivado software is used by the VHDL/VERILOG designers for performing Synthesis operation. Any simulated code can be synthesized and configured on FPGA. Synthesis is the transformation of HDL code into gate level net list. It is an integral part of current design flows. Click on Xilinx Vivado Design Suite icon. It opens the design suite window as shown below.

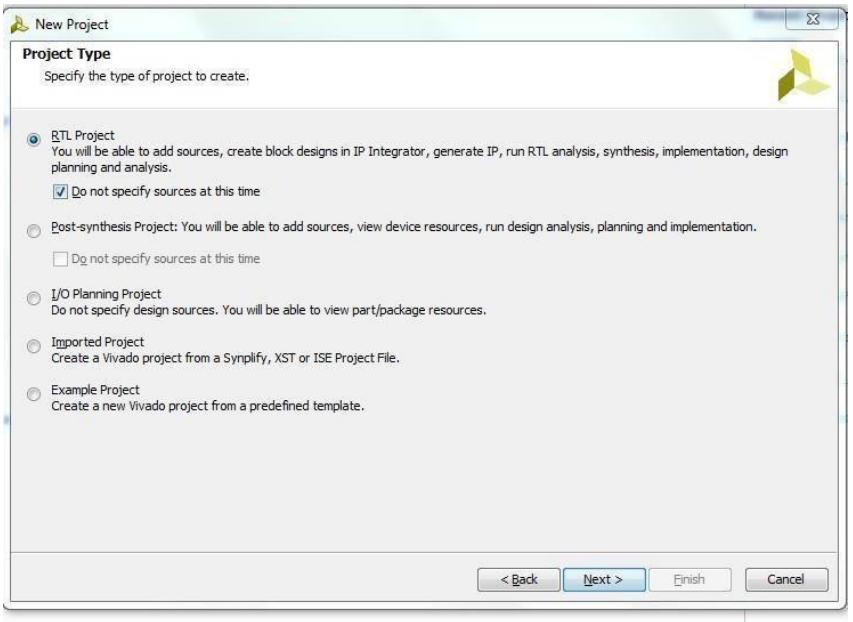


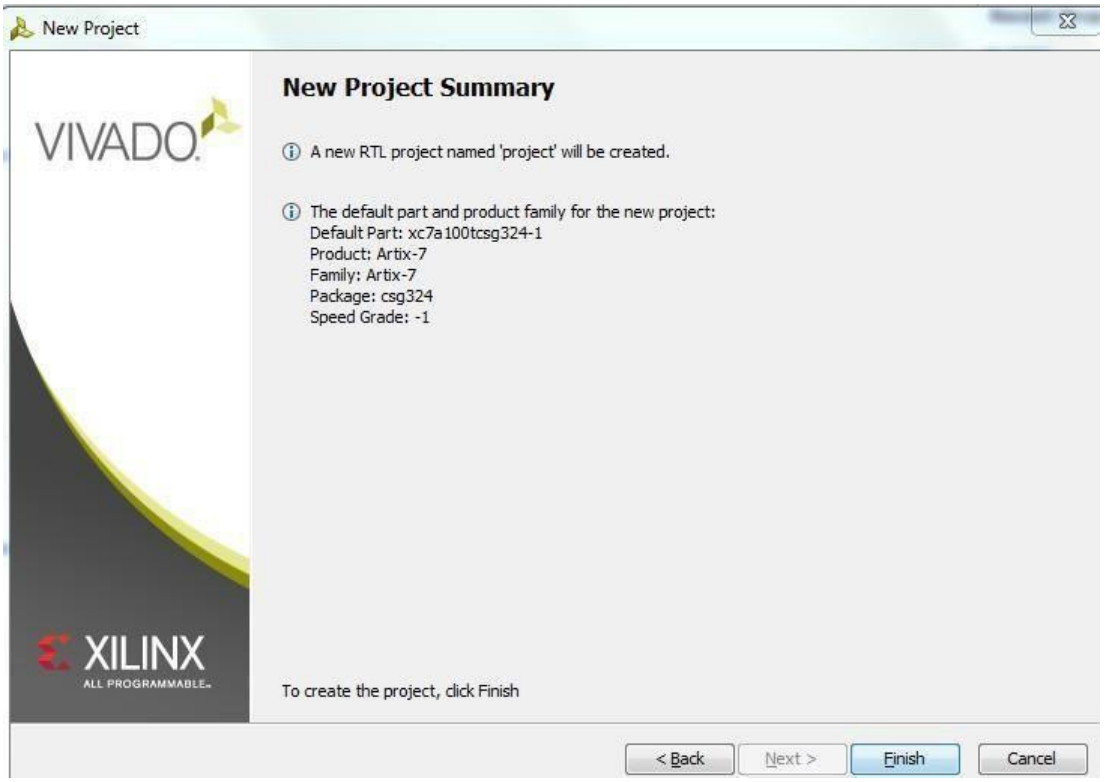
Click Open New Project. A new project creation wizard will pop up. Click Next and then name the project "project_1_multiple_inputs". Select the folder path and create project subfolder. Choose RTL project

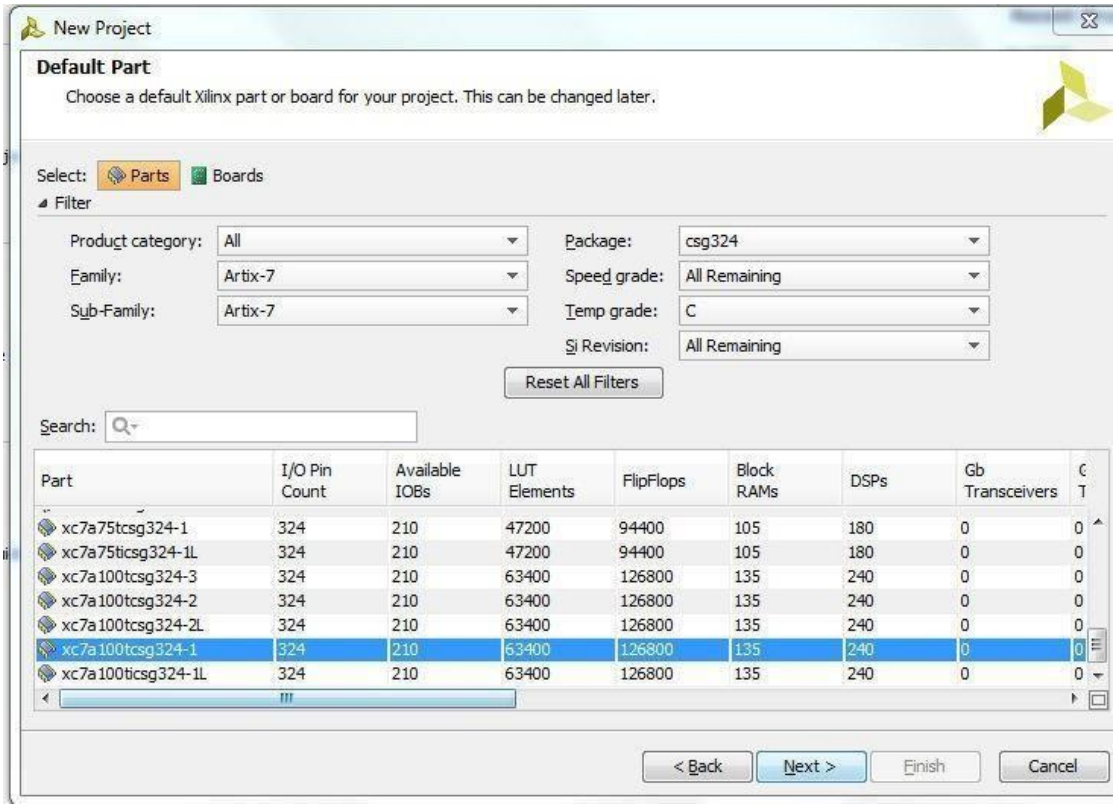
Click on NEW project

Click on Next, then enter name of project and specify the location of project then click Next.

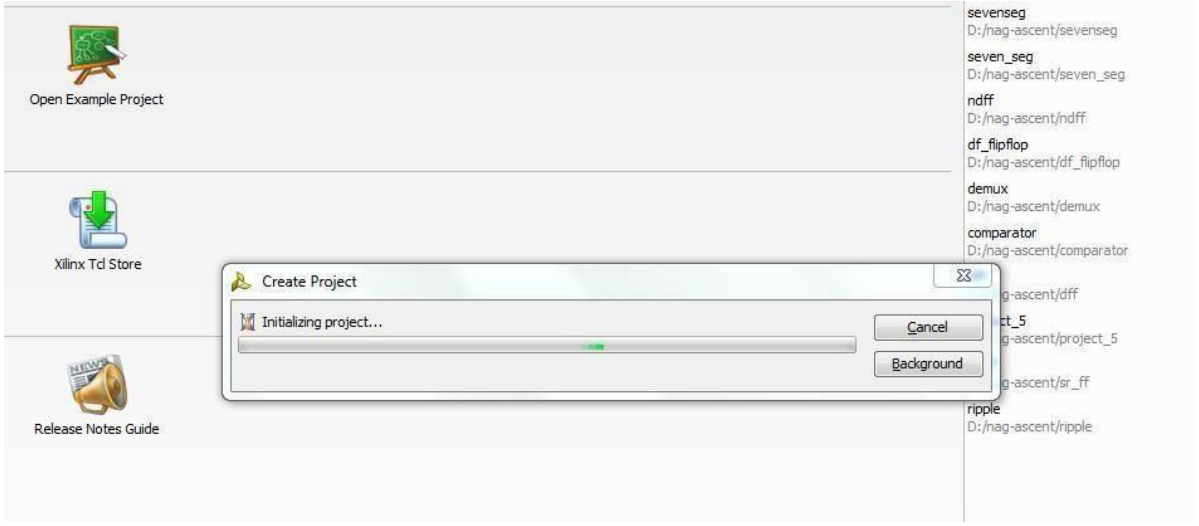


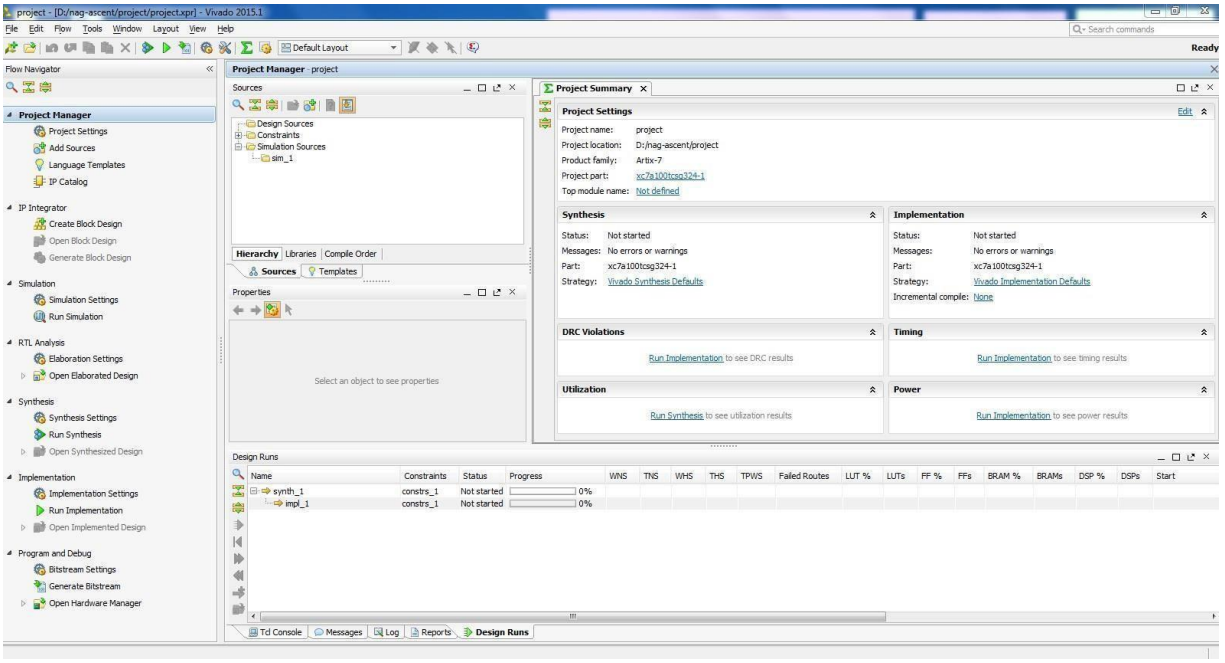

New Project
Project Type
Specify the type of project to create.
RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
Do not specify sources at this time
Post-synthesis Project: You will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


New Project
VIVADO
XILINX
ALL PROGRAMMABLE
New Project Summary
A new RTL project named 'project' will be created.
The default part and product family for the new project:
Default Part: xc7a100tcsg324-1
Product: Artix-7
Family: Artix-7
Package: csg324
Speed Grade: -1
To create the project, click Finish
< Back
Next >
Finish
Cancel

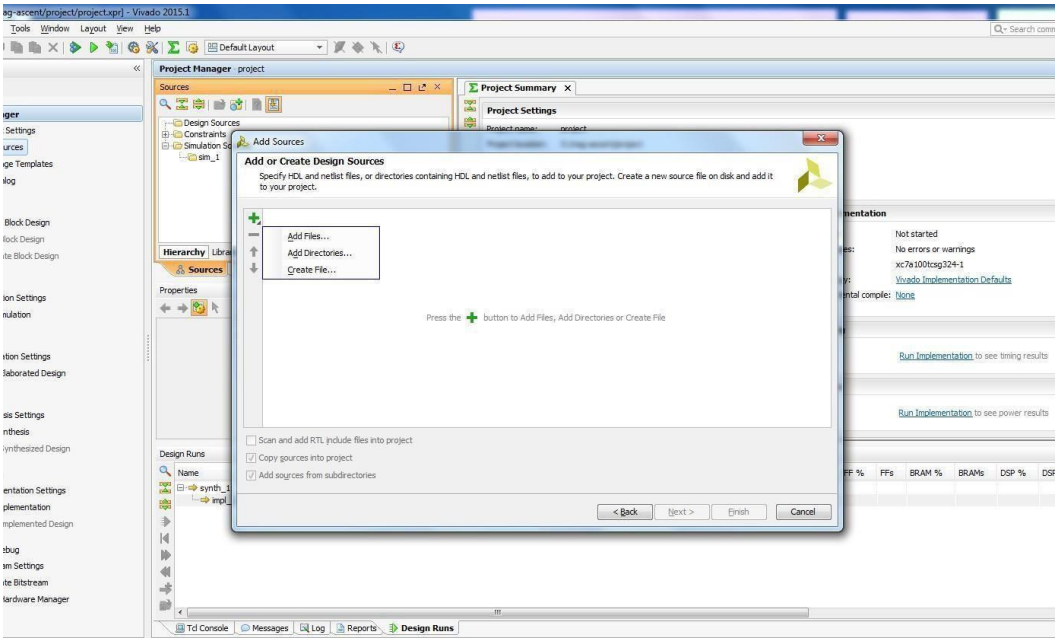

New Project
Default Part
Choose a default Xilinx part or board for your project. This can be changed later.
Select: Parts Boards
Filter
Product category: All
Family: Artix-7
Sub-Family: Artix-7
Package: csg324
Speed grade: All Remaining
Temp grade: C
Si Revision: All Remaining
Reset All Filters
Search:
Part | I/O Pin Count | Available IOBs | LUT Elements | FlipFlops | Block RAMs | DSPs | Gb Transceivers
xc7a75tcsg324-1 | 324 | 210 | 47200 | 94400 | 105 | 180 | 0
xc7a75ticsg324-1L | 324 | 210 | 47200 | 94400 | 105 | 180 | 0
xc7a100tcsg324-3 | 324 | 210 | 63400 | 126800 | 135 | 240 | 0
xc7a100tcsg324-2 | 324 | 210 | 63400 | 126800 | 135 | 240 | 0
xc7a100tcsg324-2L | 324 | 210 | 63400 | 126800 | 135 | 240 | 0
xc7a100tcsg324-1 | 324 | 210 | 63400 | 126800 | 135 | 240 | 0
xc7a100ticsg324-1L | 324 | 210 | 63400 | 126800 | 135 | 240 | 0
< Back
Next >
Finish
Cancel

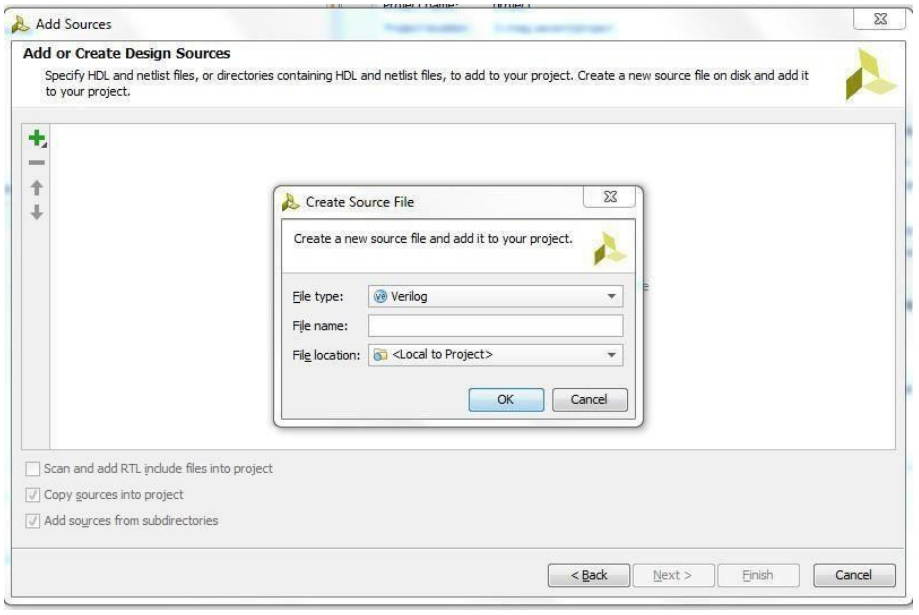

New Project
VIVADO
XILINX
ALL PROGRAMMABLE
New Project Summary
A new RTL project named 'project' will be created.
The default part and product family for the new project:
Default Part: xc7a100tcsg324-1
Product: Artix-7
Family: Artix-7
Package: csg324
Speed Grade: -1
To create the project, click Finish
< Back
Next >
Finish
Cancel

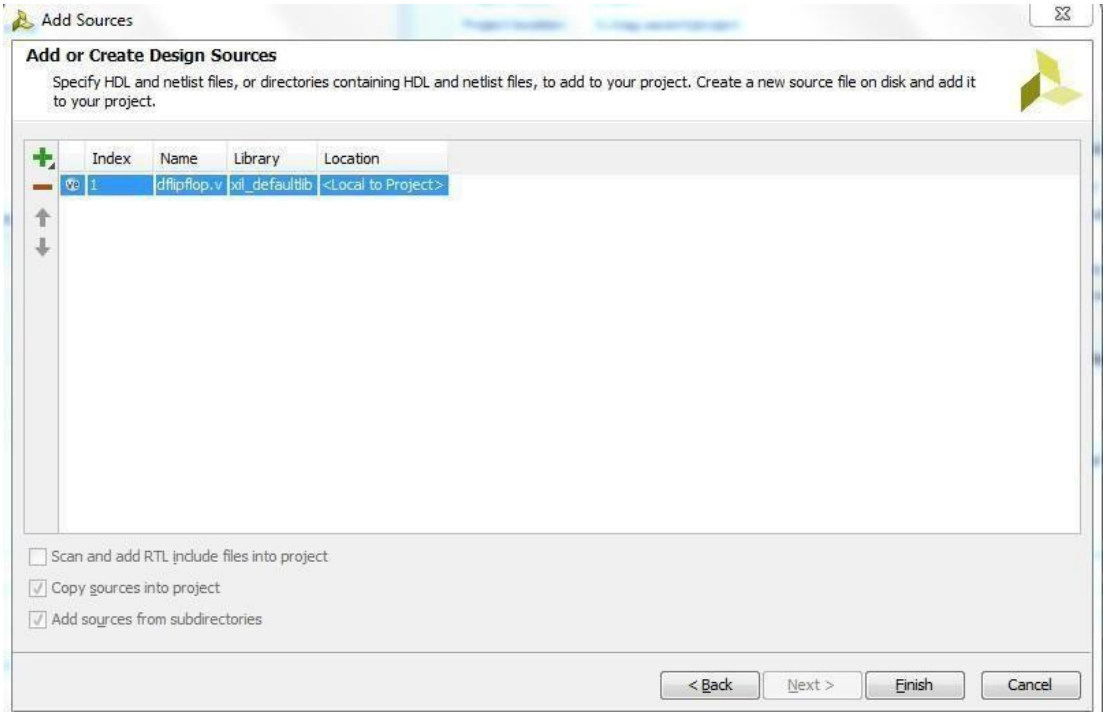

Open Example Project
Xilinx Tcl Store
Release Notes Guide
Create Project
Initializing project...
Cancel
Background
sevenseg
D:/nag-ascent/sevenseg
seven_seg
D:/nag-ascent/seven_seg
ndff
D:/nag-ascent/ndff
df_flipflop
D:/nag-ascent/df_flipflop
demux
D:/nag-ascent/demux
comparator
D:/nag-ascent/comparator
ripple
D:/nag-ascent/ripple

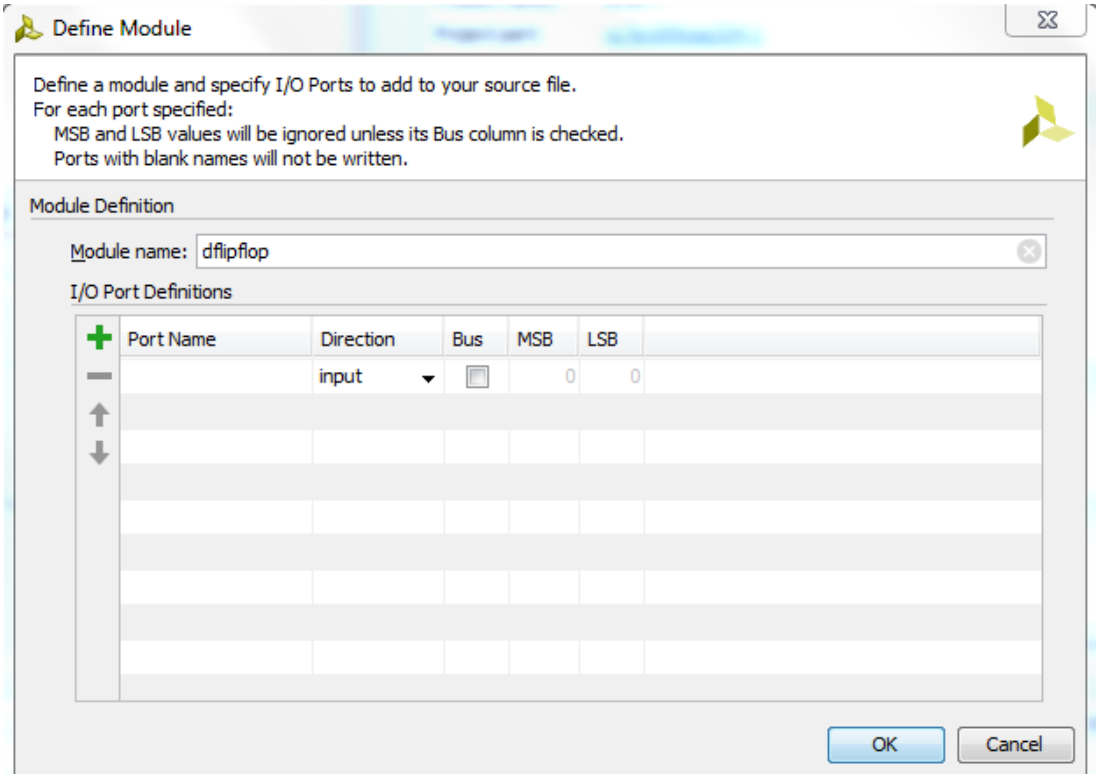

project - [D:/nag-ascent/project/project.xpr] - Vivado 2015.1
Project Manager
Project Summary
Design Runs

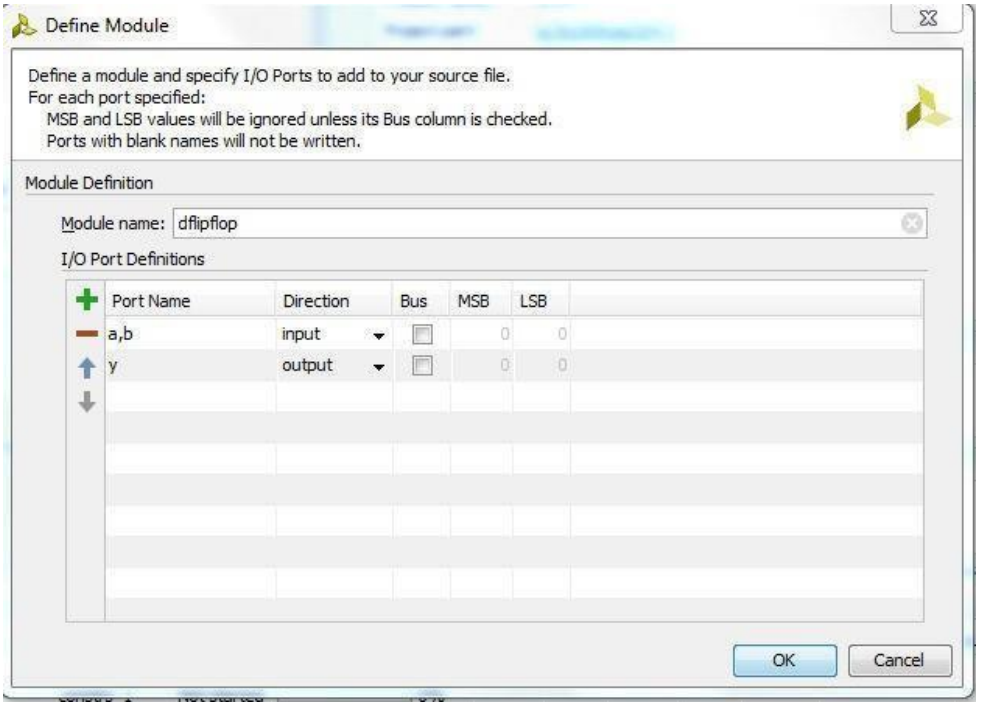

VIVADO
XILINX
ALL PROGRAMMABLE
Add Sources
This guides you through the process of adding and creating sources for your project
Add or create constraints
Add or create design sources
Add or create simulation sources
Add or create DSP sources
Add existing block design sources
Add existing IP
To continue,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

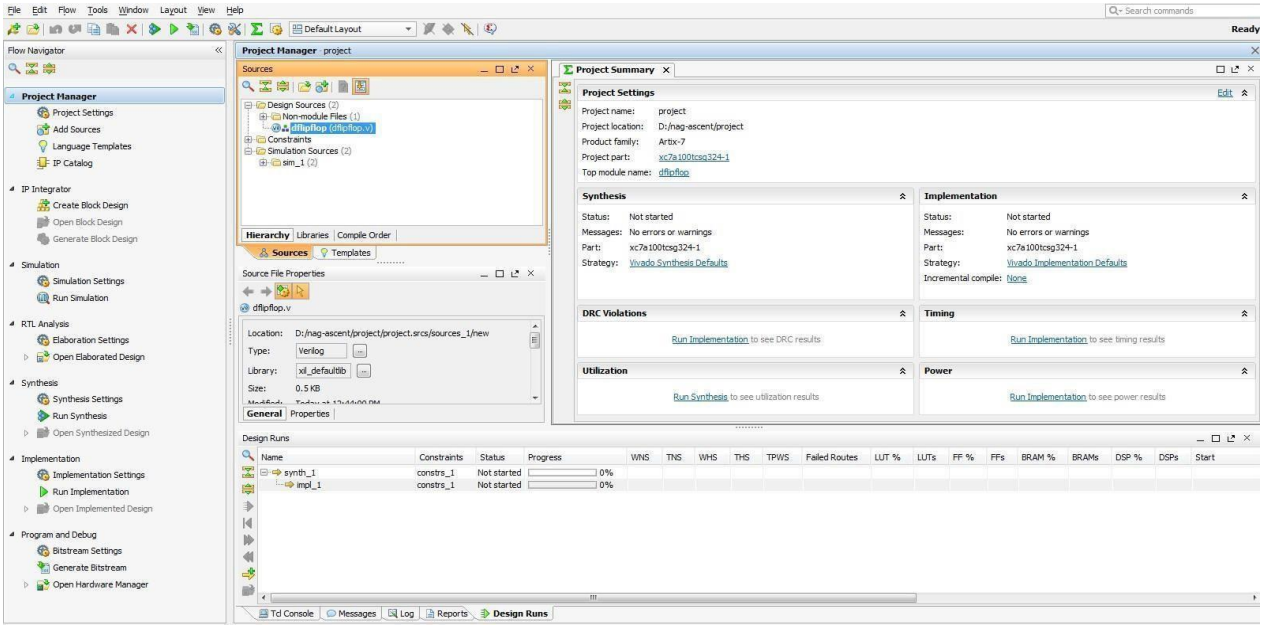

Project Manager
Project Summary
Add Sources
Add or Create Design Sources
Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.
Add Files...
Add Directories...
Create File...
Press the + button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

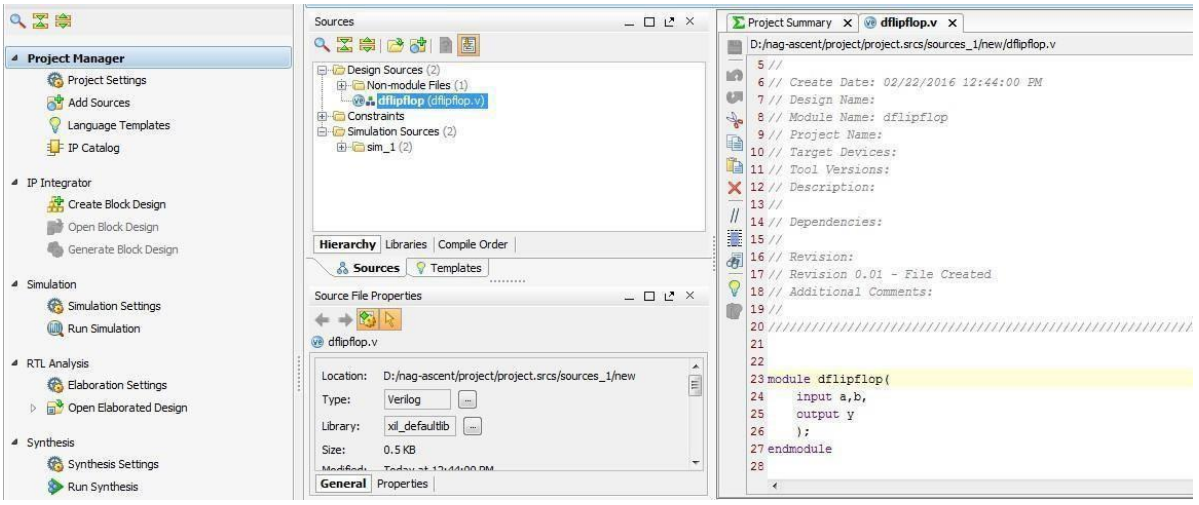

Add Sources
Add or Create Design Sources
Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.
Create Source File
Create a new source file and add it to your project.
File type:
Verilog
File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

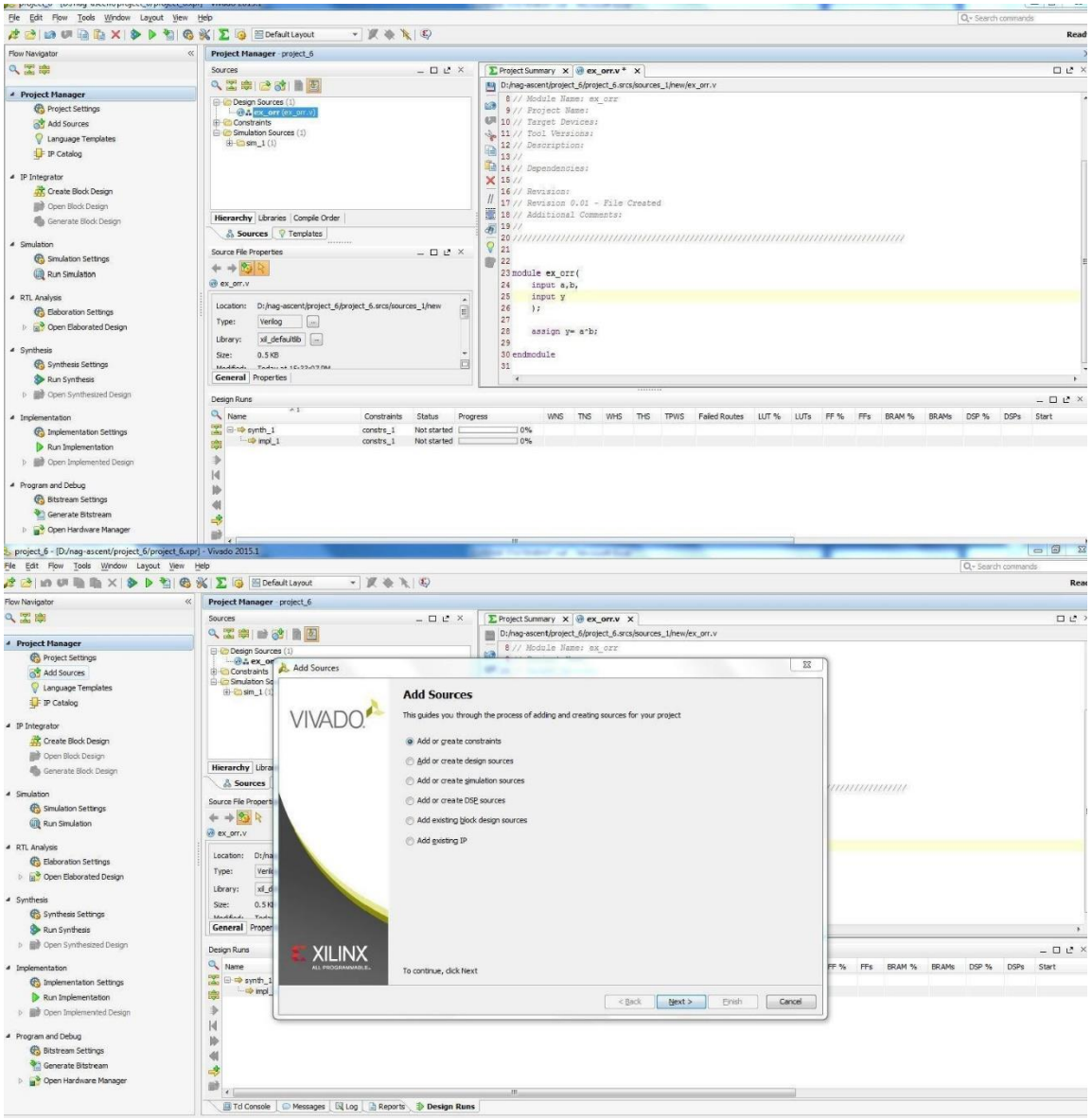

Add Sources
Add or Create Design Sources
Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.
Index
Name
Library
Location
1
dflipflop.v
xil_defaultlib
<Local to Project>
Scan and add RTL include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

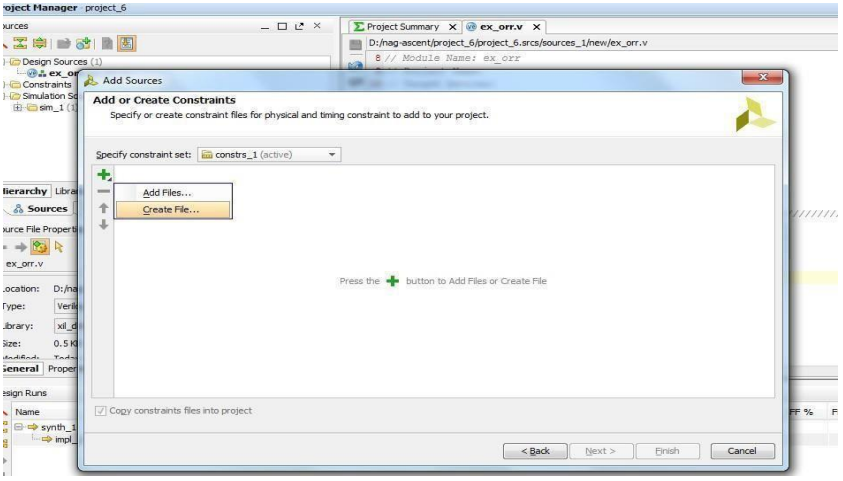

Define Module
Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.
Module Definition
Module name: dflipflop
I/O Port Definitions
Port Name
Direction
Bus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

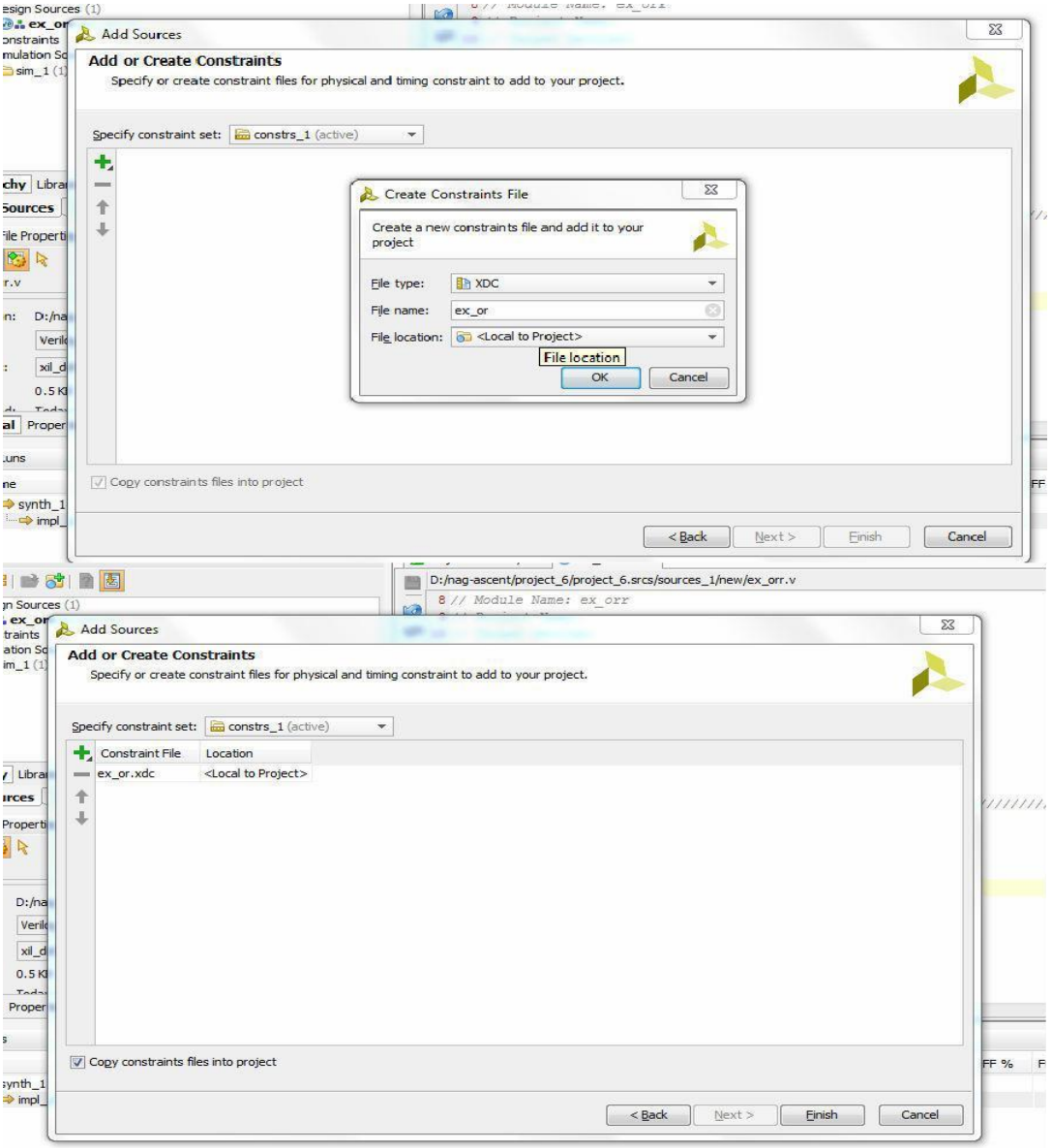

Define Module
Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.
Module Definition
Module name: dflipflop
I/O Port Definitions
Port Name
Direction
Bus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

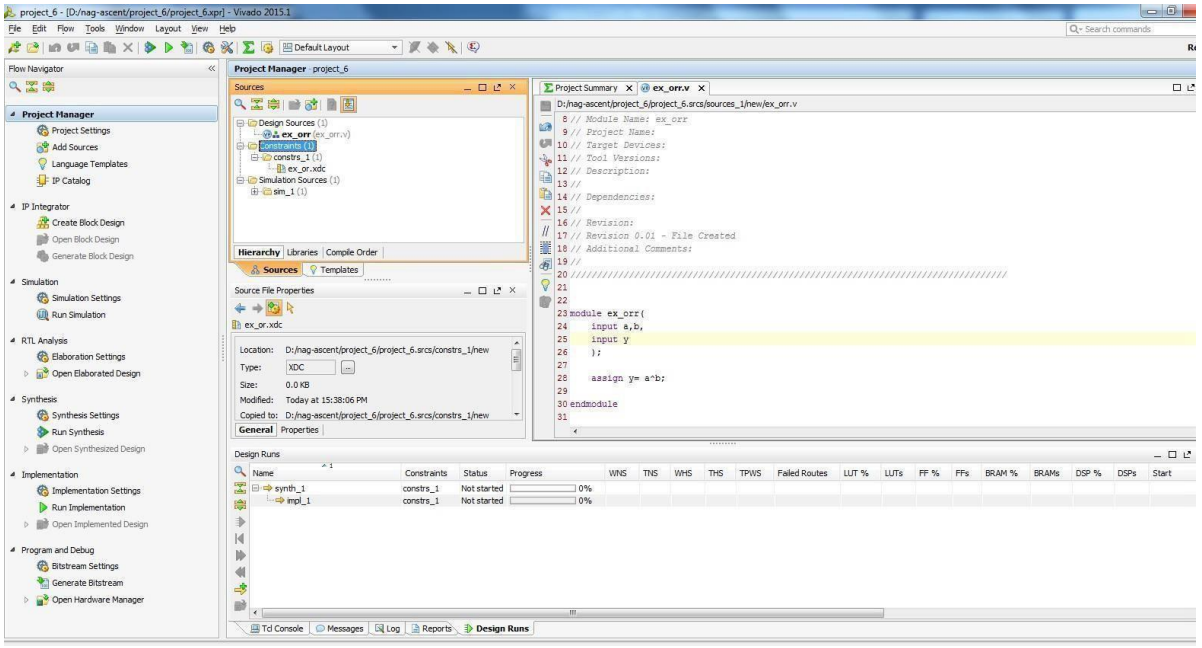

File Edit Flow Tools Window Layout View Help
Default Layout
Ready
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration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

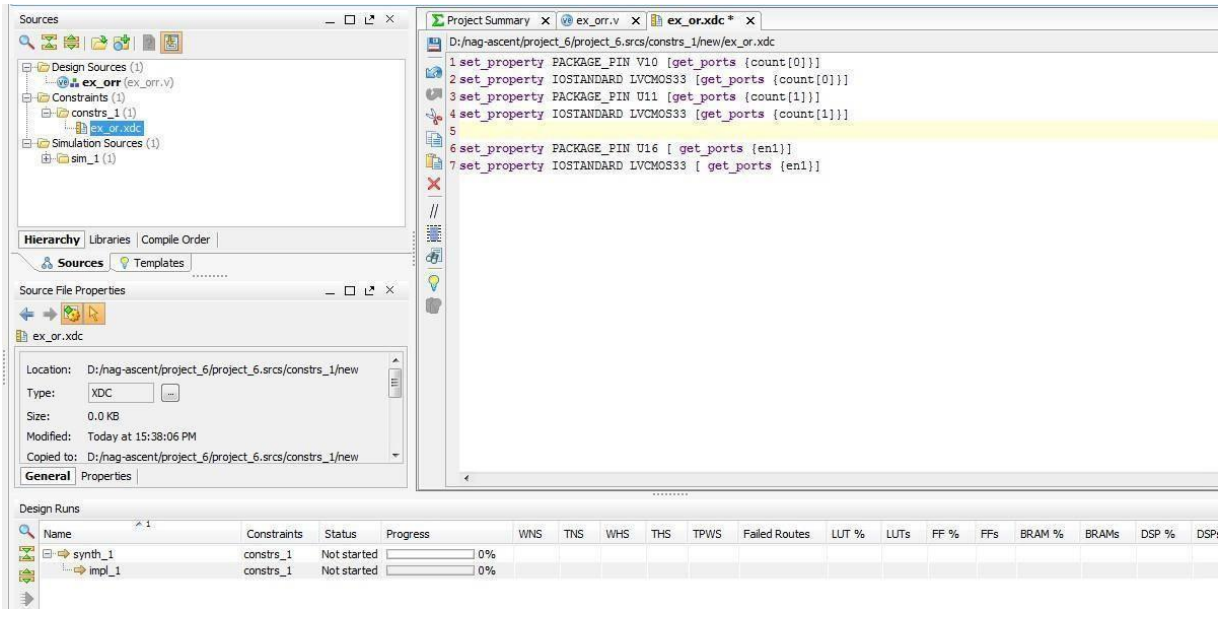

Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Sources
Design Sources (2)
Non-module Files (1)
dflipflop (dflipflop.v)
Constraints
Simulation Sources (2)
sim_1 (2)
Hierarchy Libraries Compile Order
Sources Templates
Source File Properties
dflipflop.v
Location: D:/nag-ascent/project/project.srcs/sources_1/new
Type: Verilog
Library: xil_defaultlib
Size: 0.5 KB
General Properties
Project Summary
dflipflop.v
D:/nag-ascent/project/project.srcs/sources_1/new/dflipflop.v
5 //
6 // Create Date: 02/22/2016 12:44:00 PM
7 // Design Name:
8 // Module Name: dflipflop
9 // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //
21
22
23 module dflipflop(
24 input a,b,
25 output y
26);
27 endmodule
28

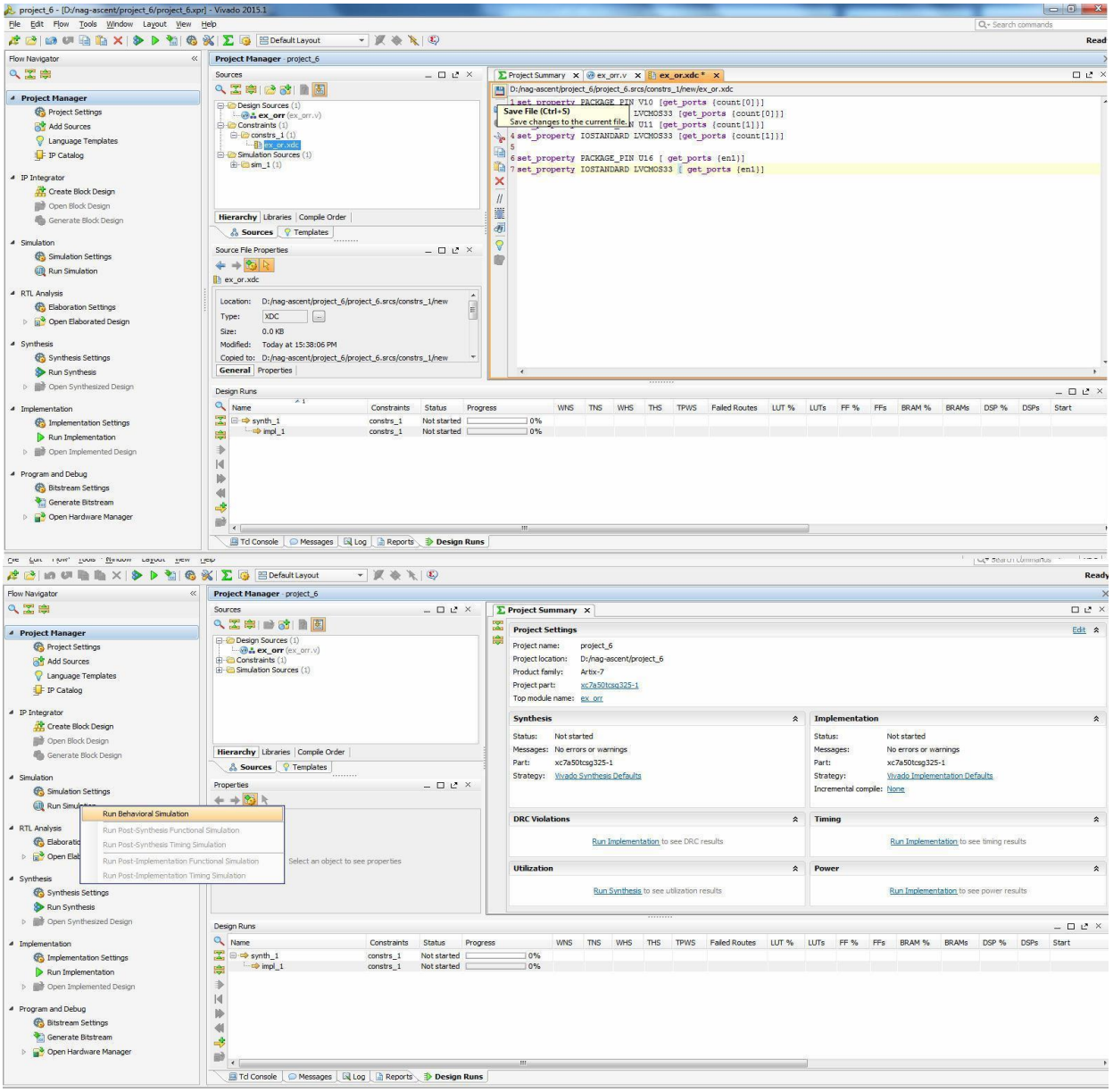

Add Sources
Add Sources
This guides you through the process of adding and creating sources for your project
Add or create constraints
Add or create design sources
Add or create simulation sources
Add or create DSP sources
Add existing block design sources
Add existing IP
To continue, click Next
< Back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

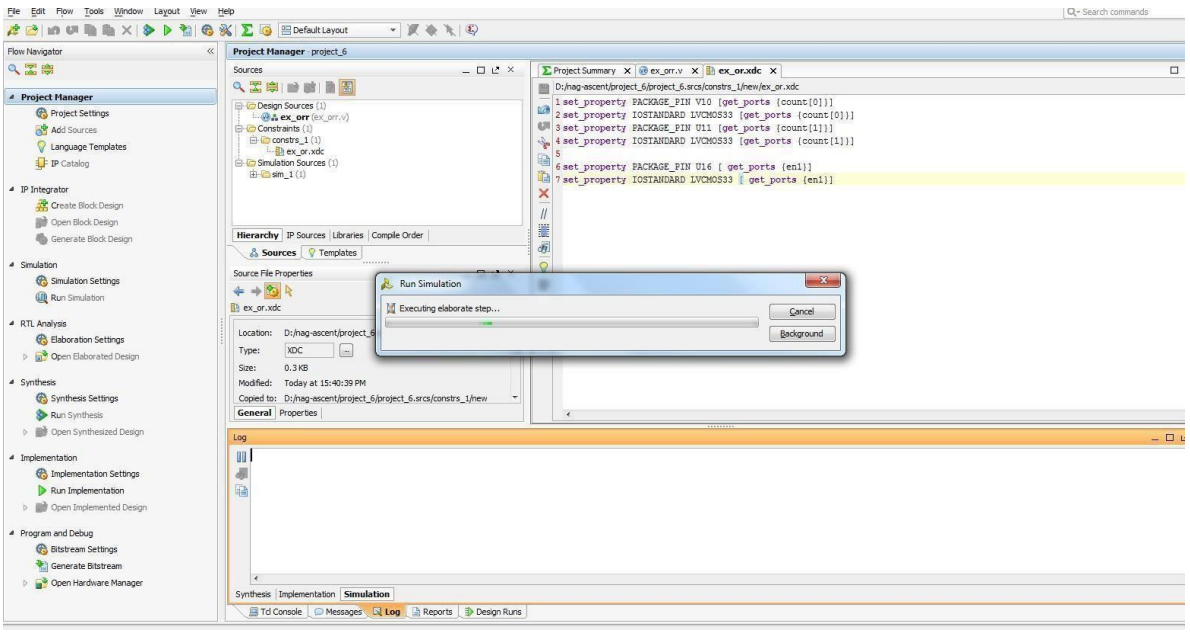

Add Sources
Add or Create Constraints
Specify or create constraint files for physical and timing constraint to add to your project.
Specify constraint set:
constrs_1 (active)
Add Files...
Create File...
Press the button to Add Files or Create File
Copy constraints files into project
< Back
Next >
Finish
Cancel

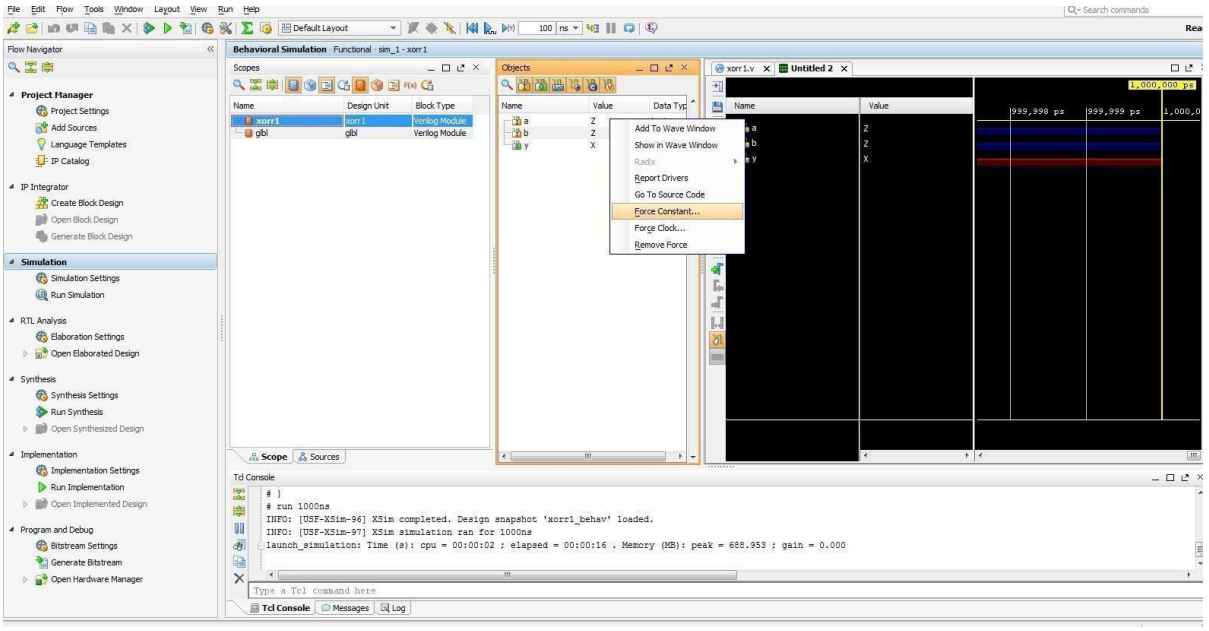

Add Sources
Add or Create Constraints
Specify or create constraint files for physical and timing constraint to add to your project.
Specify constraint set:
constrs_1 (active)
Create Constraints File
Create a new constraints file and add it to your project
File type:
XDC
File name:
ex_or
File location: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

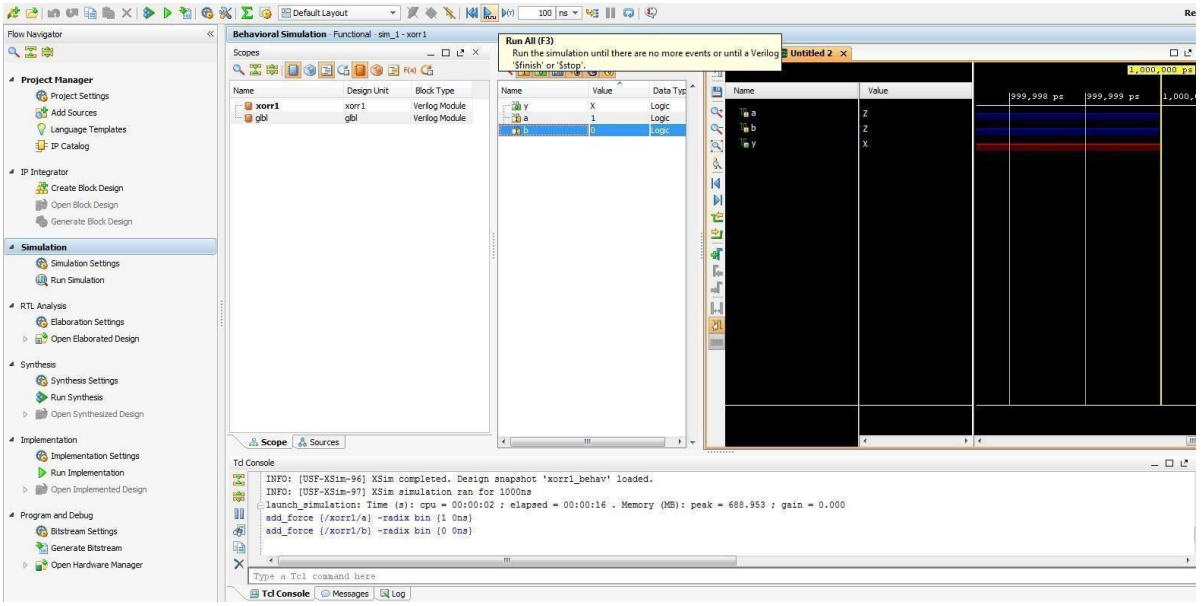

project_6 - [D:/nag-ascent/project_6/project_6.xpr] - Vivado 2015.1
File Edit Flow Tools Window Layout View Help
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings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

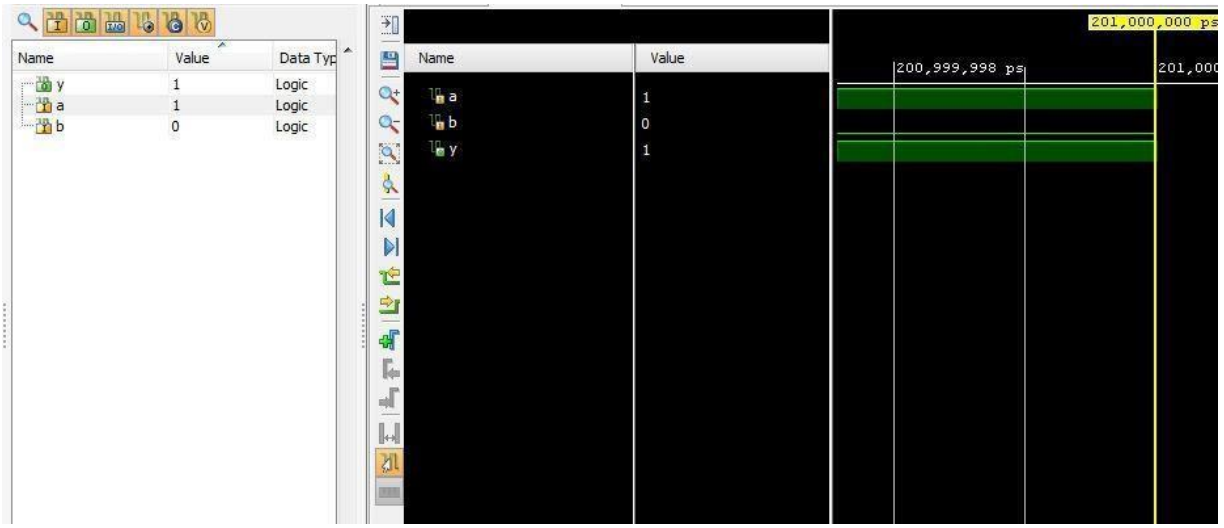

Sources
Design Sources (1)
ex_orr (ex_orr.v)
Constraints (1)
constrs_1 (1)
ex_or.xdc
Simulation Sources (1)
sim_1 (1)
Hierarchy Libraries Compile Order
Sources Templates
Source File Properties
ex_or.xdc
Location: D:/nag-ascent/project_6/project_6.srcs/constrs_1/new
Type: XDC
Size: 0.0 KB
Modified: Today at 15:38:06 PM
Copied to: D:/nag-ascent/project_6/project_6.srcs/constrs_1/new
General Properties
Project Summary
ex_orr.v
ex_or.xdc *
D:/nag-ascent/project_6/project_6.srcs/constrs_1/new/ex_or.xdc
1 set_property PACKAGE_PIN V10 [get_ports {count[0]}]
2 set_property IOSTANDARD LVCMOS33 [get_ports {count[0]}]
3 set_property PACKAGE_PIN U11 [get_ports {count[1]}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {count[1]}]
5
6 set_property PACKAGE_PIN U16 [get_ports {en1}]
7 set_property IOSTANDARD LVCMOS33 [get_ports {en1}]
Design Runs
Name Constraints Status Progress WNS TNS WHS THS TPWS Failed Routes LUT % LUTs FF % FFs BRAM % BRAMs DSP % DSPs
synth_1 constrs_1 Not started 0%
impl_1 constrs_1 Not started 0%

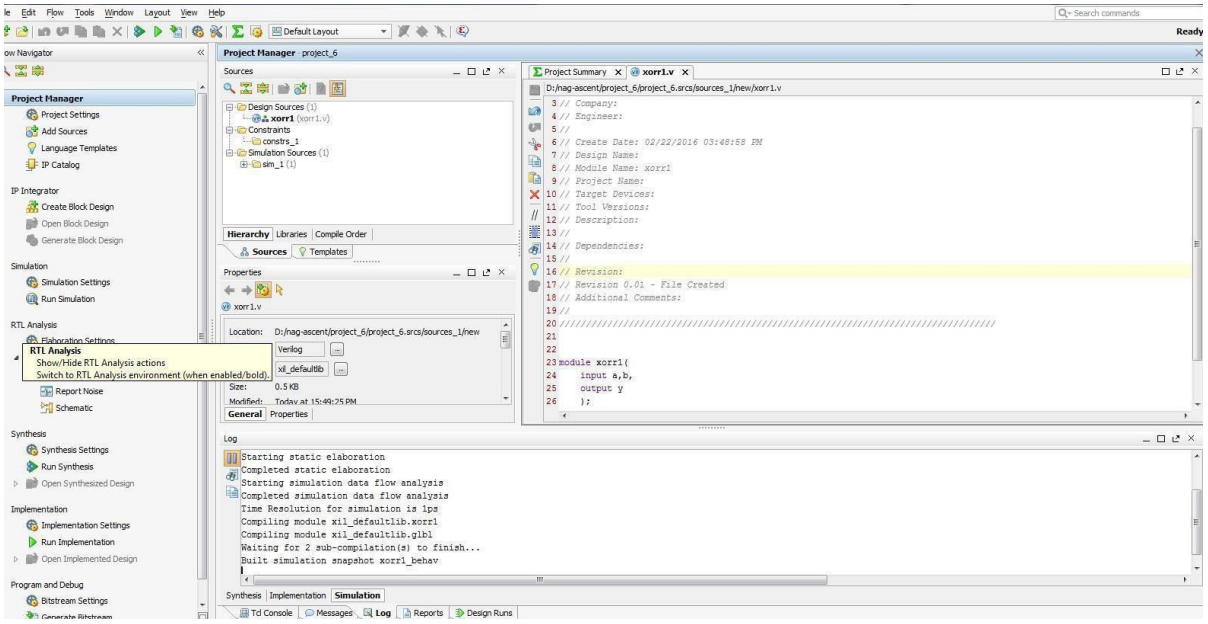

project_6 - [D:/nag-ascent/project_6/project_6.xpr] - Vivado 2015.1
File Edit Flow Tools Window Layout View Help
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings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

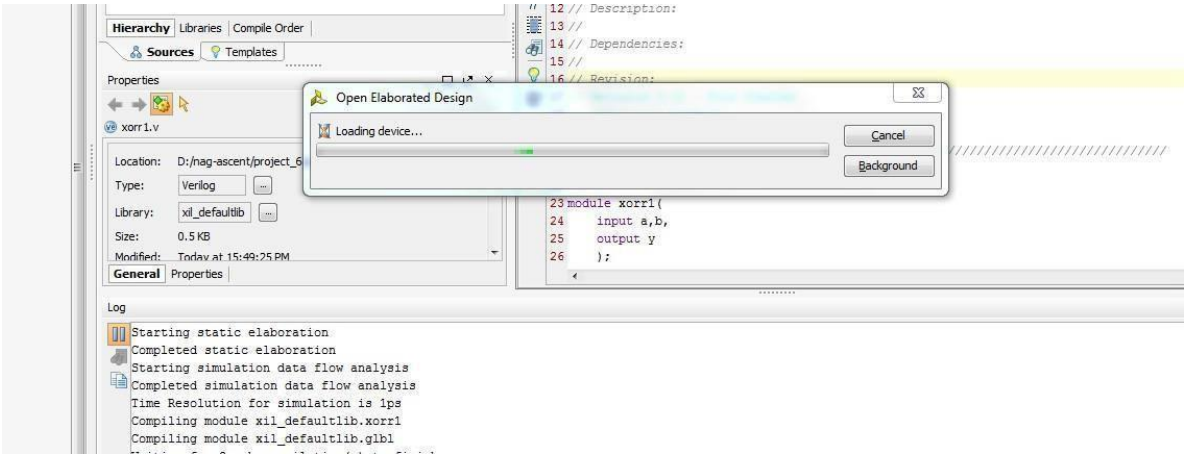

File Edit Flow Tools Window Layout View Help
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated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

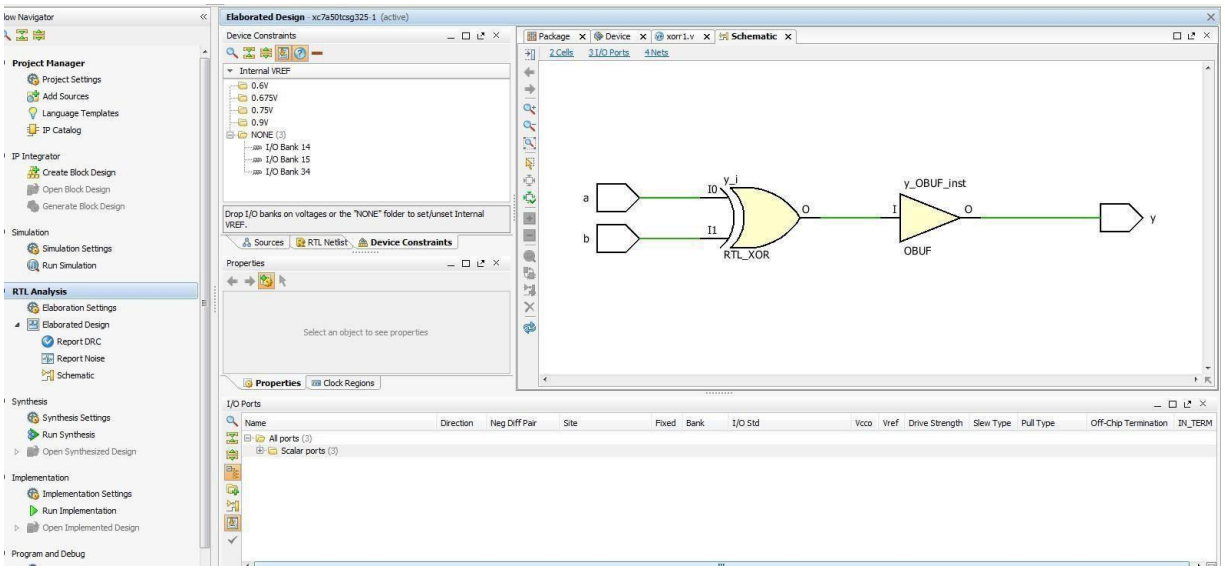

File Edit Flow Tools Window Layout View Run Help
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated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

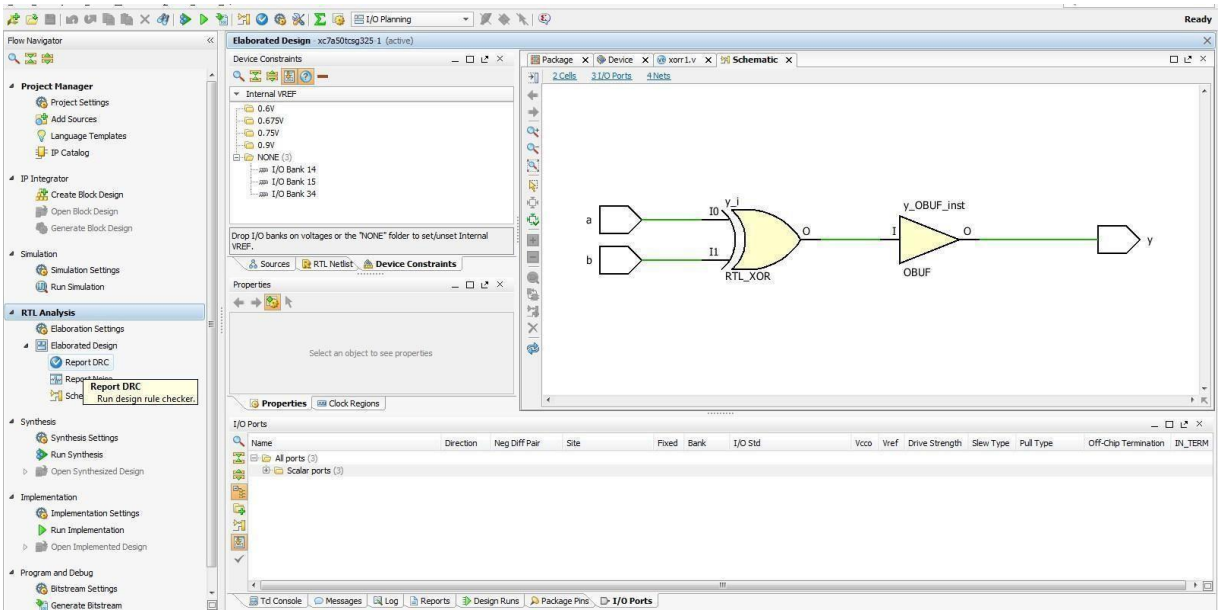

Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis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

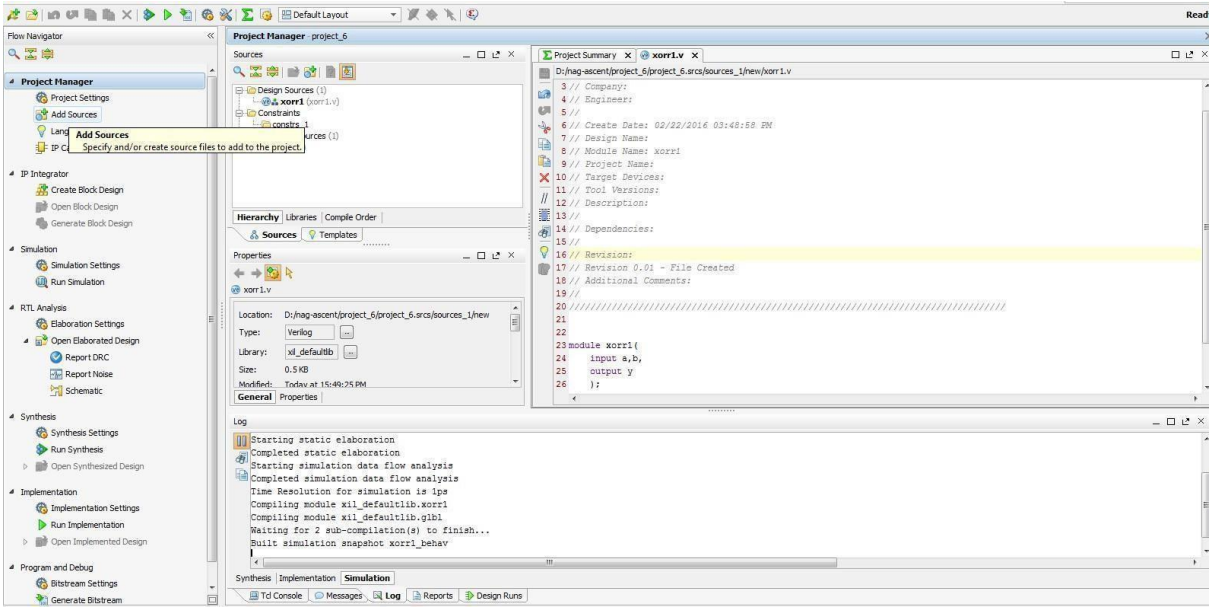

Name
Value
Data Typ
y
1
Logic
a
1
Logic
b
0
Logic
201,000,000 ps
200,999,998 ps

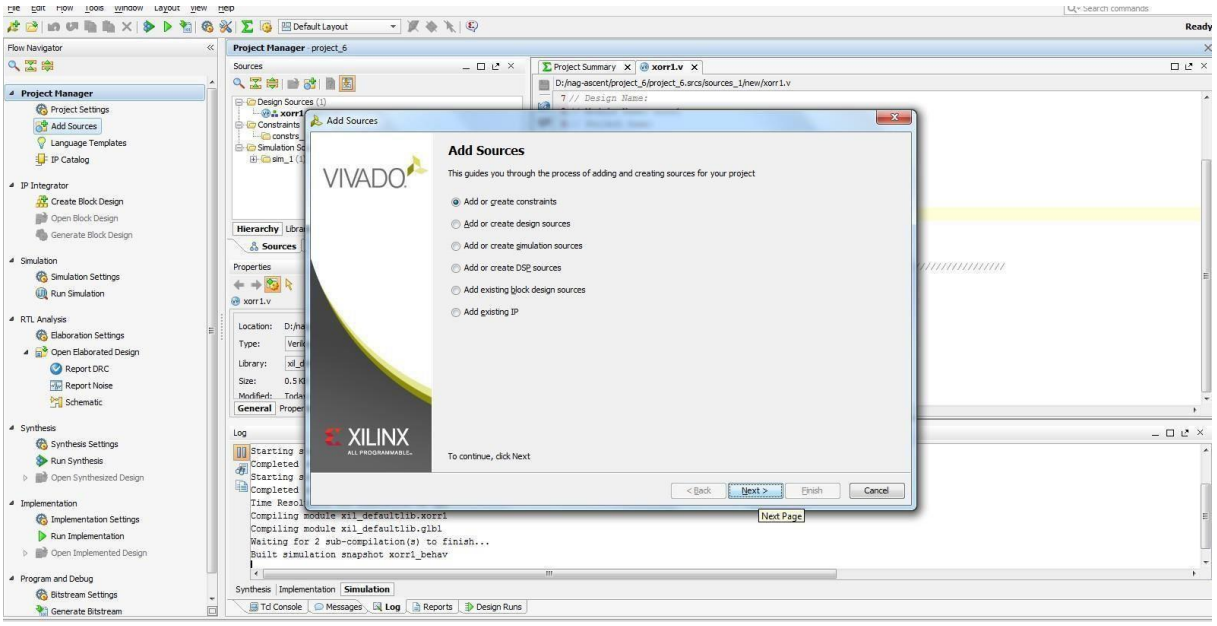

Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
RTL Analysis
Show/Hide RTL Analysis actions
Switch to RTL Analysis environment (when enabled/bold).
Report Noise
Schematic
Synthesis
Synthesis Settings
Run Synthesis
Implementation
Implementation Settings
Run Implementation
Program and Debug
Bitstream Settings
Sources
Design Sources (1)
xorr1 (xorr1.v)
Constraints
constrs_1
Simulation Sources (1)
sim_1 (1)
Hierarchy
Libraries
Compile Order
Templates
Properties
Location: D:/nag-ascent/project_6/project_6.srcs/sources_1/new
Verilog
xil_defaultlib
Size: 0.5 KB
General
Properties
Project Summary
xorr1.v
D:/nag-ascent/project_6/project_6.srcs/sources_1/new/xorr1.v
// Company:
// Engineer:
// Create Date: 02/22/2016 03:48:58 PM
// Design Name:
// Module Name: xorr1
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
// Dependencies:
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
module xorr1(
input a,b,
output y
);
Log
Starting static elaboration
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling module xil_defaultlib.xorr1
Compiling module xil_defaultlib.glbl
Waiting for 2 sub-compilation(s) to finish...
Built simulation snapshot xorr1_behav
Synthesis
Implementation
Simulation
Tcl Console
Messages
Log
Reports
Design Runs

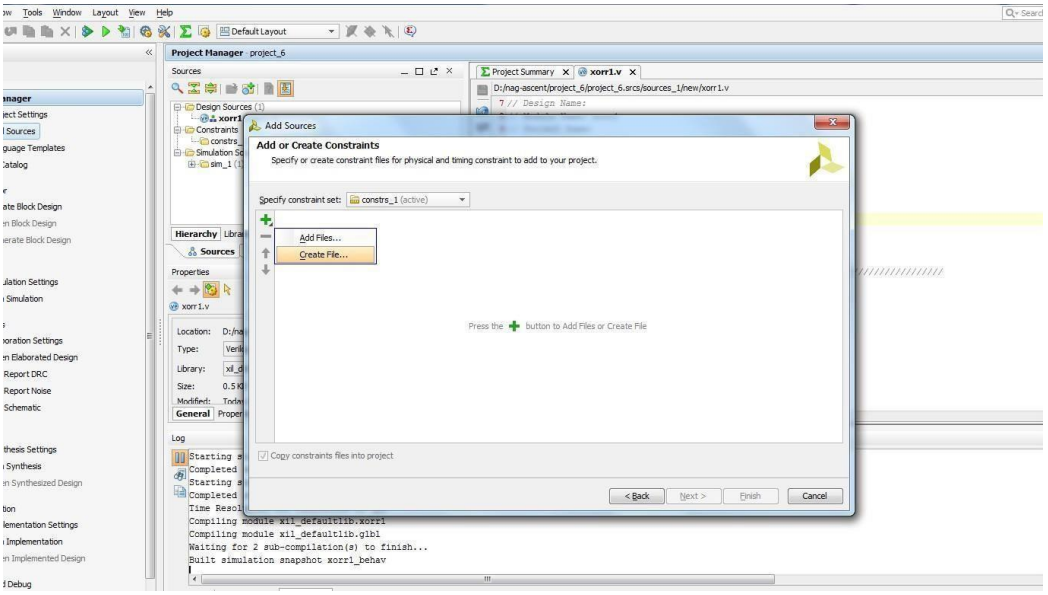

Open Elaborated Design
Loading device...
Cancel
Background
Hierarchy
Libraries
Compile Order
Sources
Templates
Properties
xorr1.v
Location: D:/nag-ascent/project_6
Type: Verilog
Library: xil_defaultlib
Size: 0.5 KB
Modified: Today at 15:49:25 PM
General
Properties
// Description:
// Dependencies:
// Revision:
module xorr1(
input a,b,
output y
);
Log
Starting static elaboration
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling module xil_defaultlib.xorr1
Compiling module xil_defaultlib.glbl

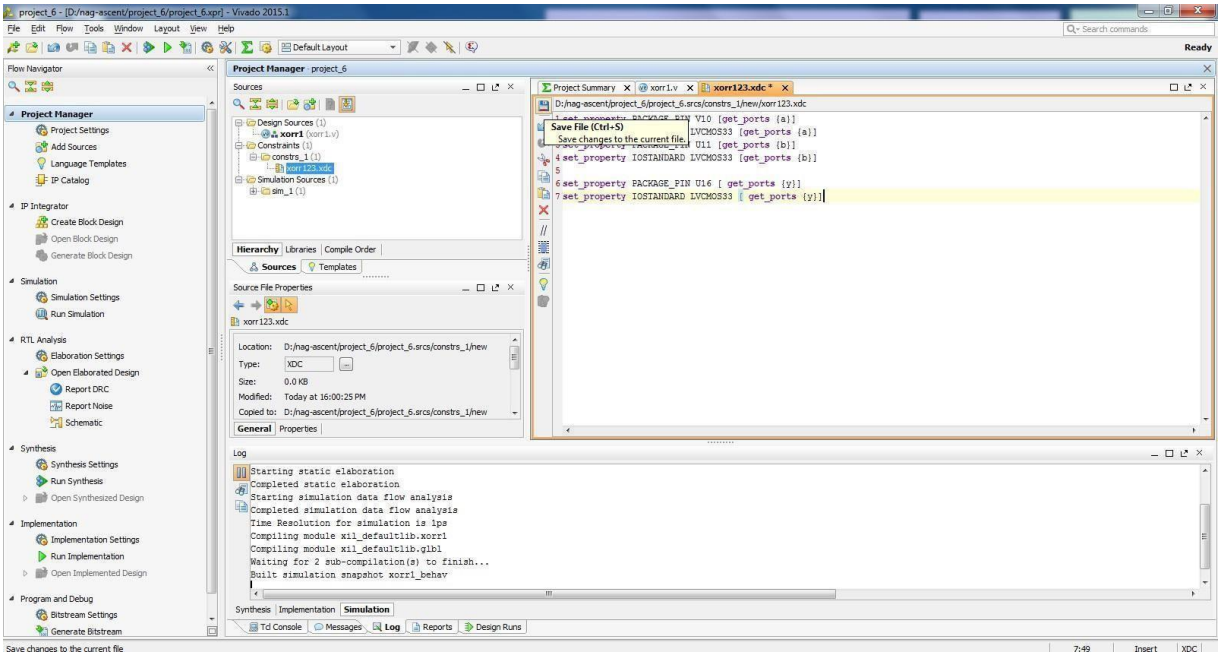

Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Elaborated Design
Report DRC
Report Noise
Schematic
Synthesis
Synthesis Settings
Run Synthesis
Implementation
Implementation Settings
Run Implementation
Program and Debug
Elaborated Design - xc7a50tcsg325-1 (active)
Device Constraints
Internal VREF
0.6V
0.675V
0.75V
0.9V
NONE (3)
I/O Bank 14
I/O Bank 15
I/O Bank 34
Drop I/O banks on voltages or the "NONE" folder to set/unset Internal VREF.
Sources
RTL Netlist
Device Constraints
Properties
Select an object to see properties
Clock Regions
Package
Device
xorr1.v
Schematic
2 Cells
3 I/O Ports
4 Nets
a
b
I0
I1
y_i
O
RTL_XOR
y_OBUF_inst
I
O
OBUF
y
I/O Ports
Name
Direction
Neg Diff Pair
Site
Fixed
Bank
I/O Std
Vcco
Vref
Drive Strength
Slew Type
Pull Type
Off-Chip Termination
IN_TERM
All ports (3)
Scalar ports (3)

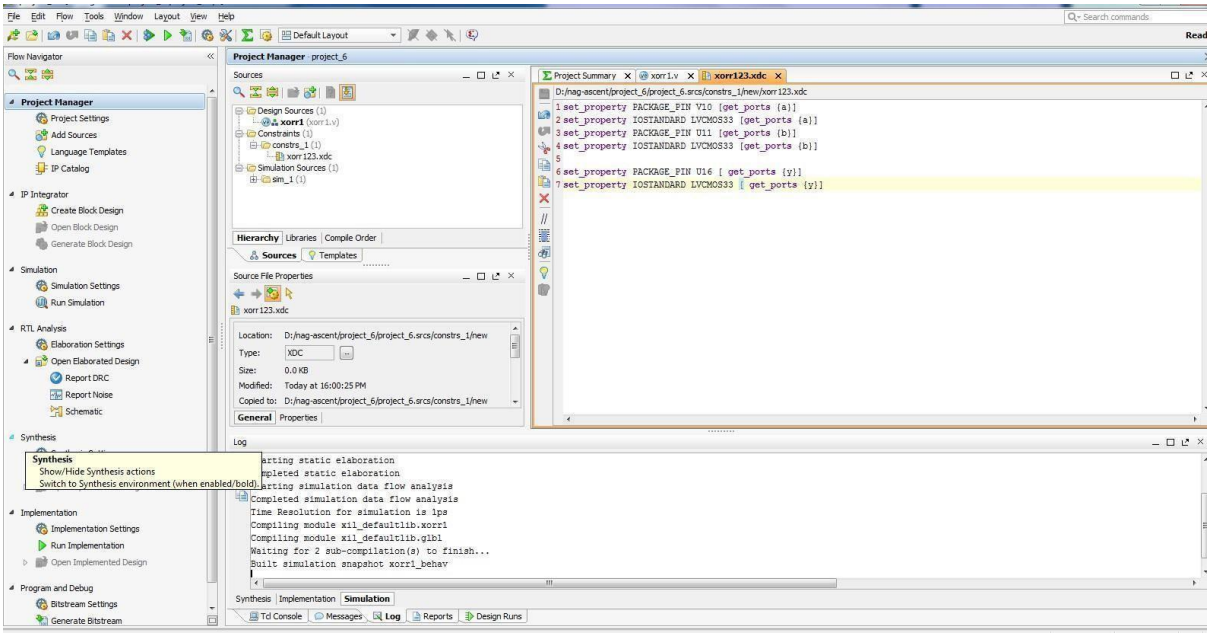

Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Elaborated Design
Report DRC
Report DRC
Run design rule checker.
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Elaborated Design xc7a50tcsg325-1 (active)
Device Constraints
Internal VREF
0.6V
0.675V
0.75V
0.9V
NONE (3)
I/O Bank 14
I/O Bank 15
I/O Bank 34
Drop I/O banks on voltages or the "NONE" folder to set/unset Internal VREF.
Sources
RTL Netlist
Device Constraints
Properties
Select an object to see properties
Properties
Clock Regions
Package
Device
xorr1.v
Schematic
2 Cells
3 I/O Ports
4 Nets
a
b
I0
I1
y_i
O
RTL_XOR
I
O
y_OBUF_inst
OBUF
y
I/O Ports
Name
Direction
Neg Diff Pair
Site
Fixed
Bank
I/O Std
Vcco
Vref
Drive Strength
Slew Type
Pull Type
Off-Chip Termination
IN_TERM
All ports (3)
Scalar ports (3)
Tcl Console
Messages
Log
Reports
Design Runs
Package Pins
I/O Ports

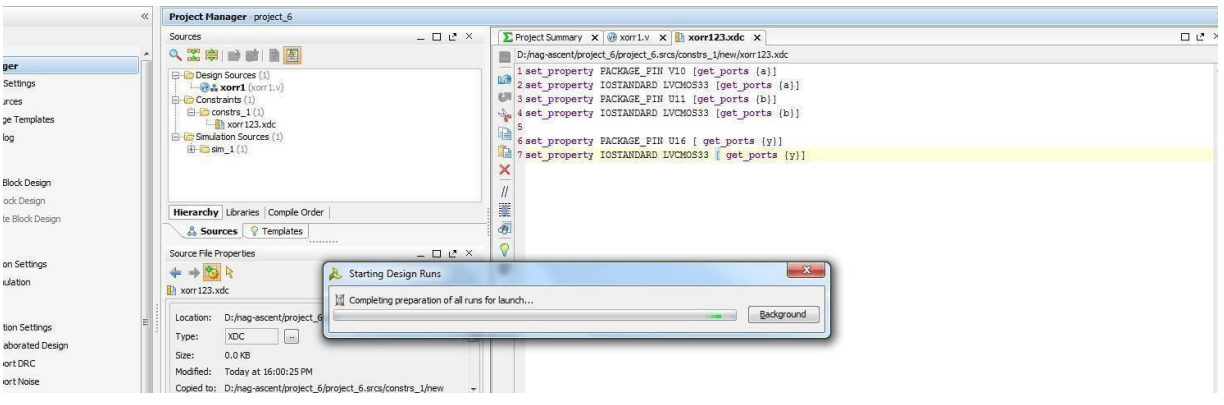

Flow Navigator
Project Manager
Project Settings
Add Sources
Add Sources
Specify and/or create source files to add to the project.
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Report DRC
Report Noise
Schematic
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Project Manager - project_6
Sources
Design Sources (1)
xorr1 (xorr1.v)
Constraints
constrs_1
Hierarchy
Libraries
Compile Order
Sources
Templates
Properties
xorr1.v
Location: D:/nag-ascent/project_6/project_6.srcs/sources_1/new
Type: Verilog
Library: xil_defaultlib
Size: 0.5 KB
General
Properties
Project Summary
xorr1.v
D:/nag-ascent/project_6/project_6.srcs/sources_1/new/xorr1.v
// Company:
// Engineer:
//
// Create Date: 02/22/2016 03:48:58 PM
// Design Name:
// Module Name: xorr1
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
module xorr1(
input a,b,
output y
);
Log
Starting static elaboration
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling module xil_defaultlib.xorr1
Compiling module xil_defaultlib.glbl
Waiting for 2 sub-compilation(s) to finish...
Built simulation snapshot xorr1_behav
Synthesis
Implementation
Simulation
Tcl Console
Messages
Log
Reports
Design Runs


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

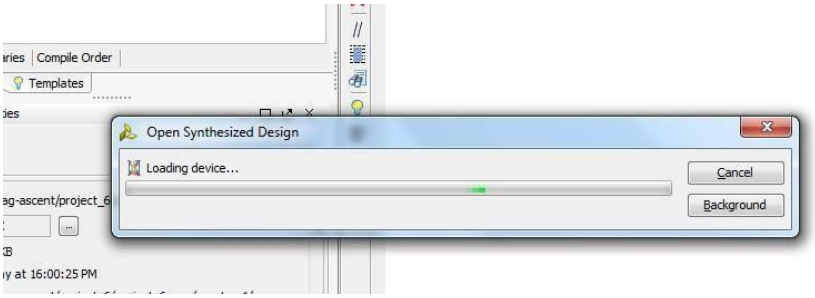

Add Sources
Add or Create Constraints
Specify or create constraint files for physical and timing constraint to add to your project.
Specify constraint set:
constrs_1 (active)
Add Files...
Create File...
Press the + button to Add Files or Create File
Copy constraints files into project
< Back
Next >
Finish
Cancel

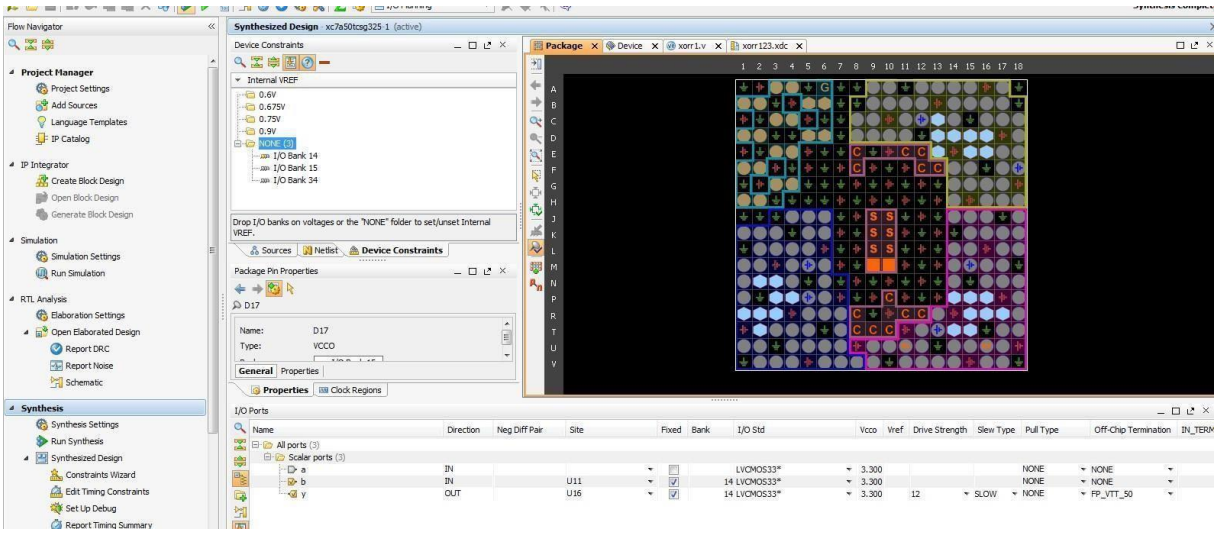

project_6 - [D:/nag-ascent/project_6/project_6.xpr] - Vivado 2015.1
Save File (Ctrl+S)
Save changes to the current file.
set_property PACKAGE_PIN V10 [get_ports {a}]
set_property IOSTANDARD LVCMOS33 [get_ports {a}]
set_property PACKAGE_PIN U11 [get_ports {b}]
set_property IOSTANDARD LVCMOS33 [get_ports {b}]
set_property PACKAGE_PIN U16 [get_ports {y}]
set_property IOSTANDARD LVCMOS33 [get_ports {y}]
Starting static elaboration
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling module xil_defaultlib.xorr1
Compiling module xil_defaultlib.glbl
Waiting for 2 sub-compilation(s) to finish...
Built simulation snapshot xorr1_behav

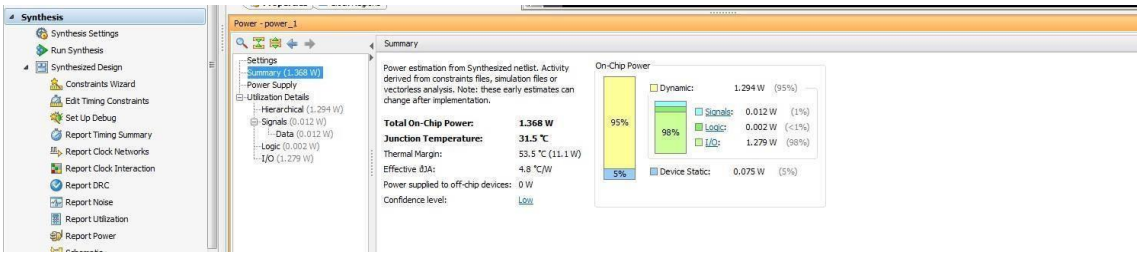

Synthesis
Show/Hide Synthesis actions
Switch to Synthesis environment (when enabled/bold).

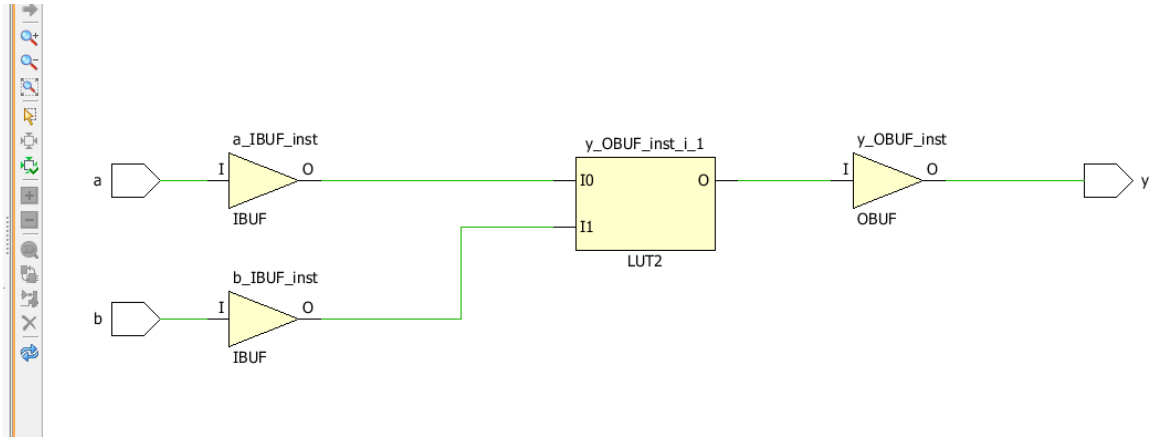

Starting Design Runs
Completing preparation of all runs for launch...
Background

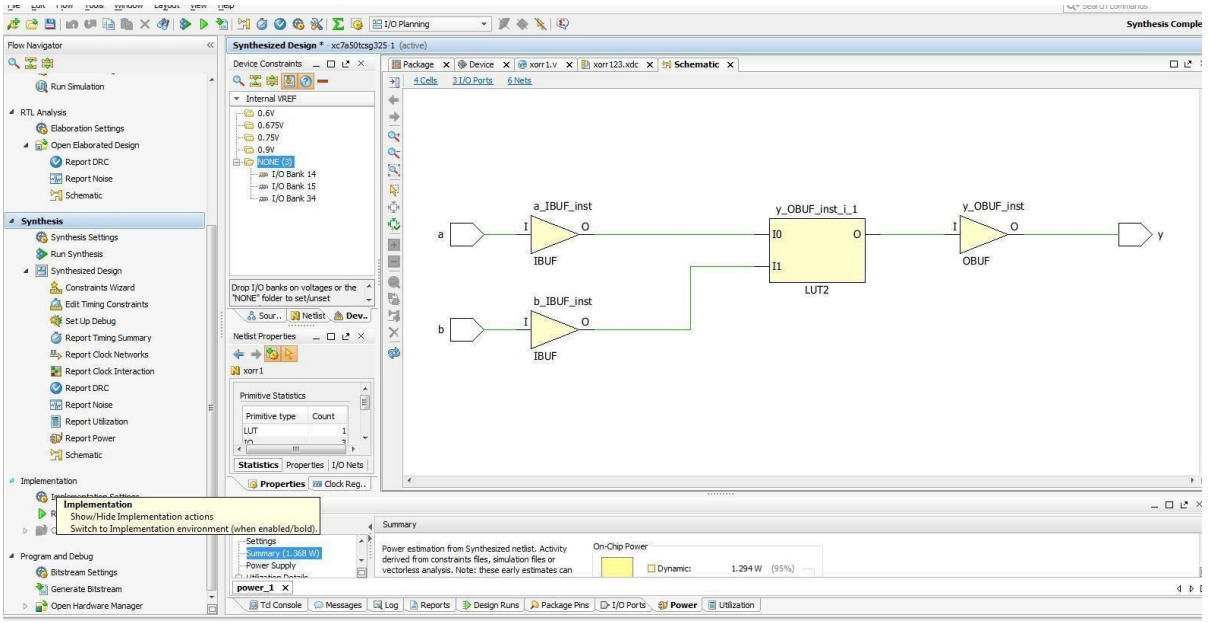

7 set_property IOSTANDARD LVCMOS33 [get_ports {y}]
Synthesis Completed
Synthesis successfully completed.
Next
Run Implementation
Open Synthesized Design
View Reports
Show Reports window
Don't show this dialog again
OK
Cancel

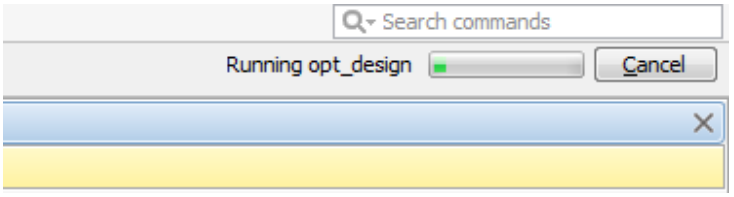

Compile Order
Templates
Open Synthesized Design
Loading device...
Cancel
Background

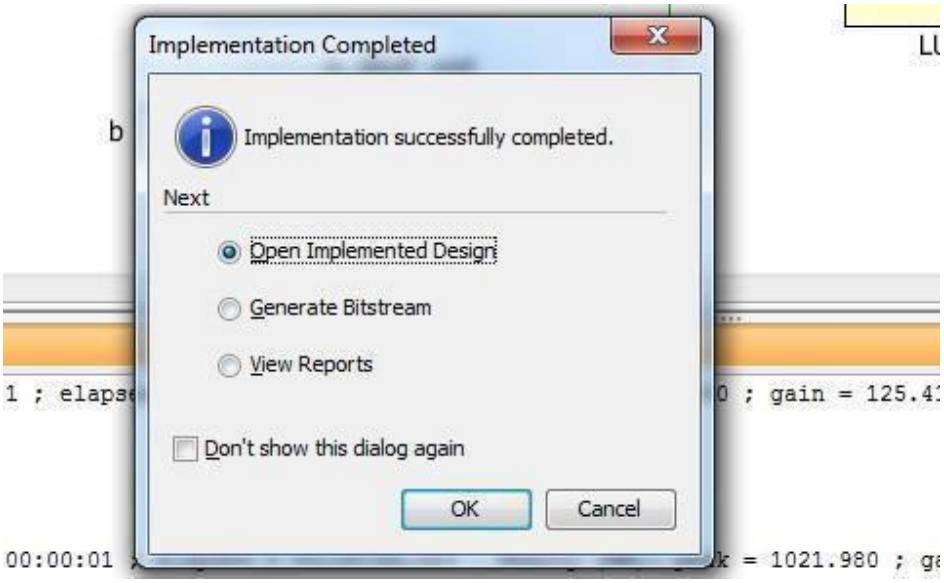

Flow Navigator
Synthesized Design
Device Constraints
Package
I/O Ports

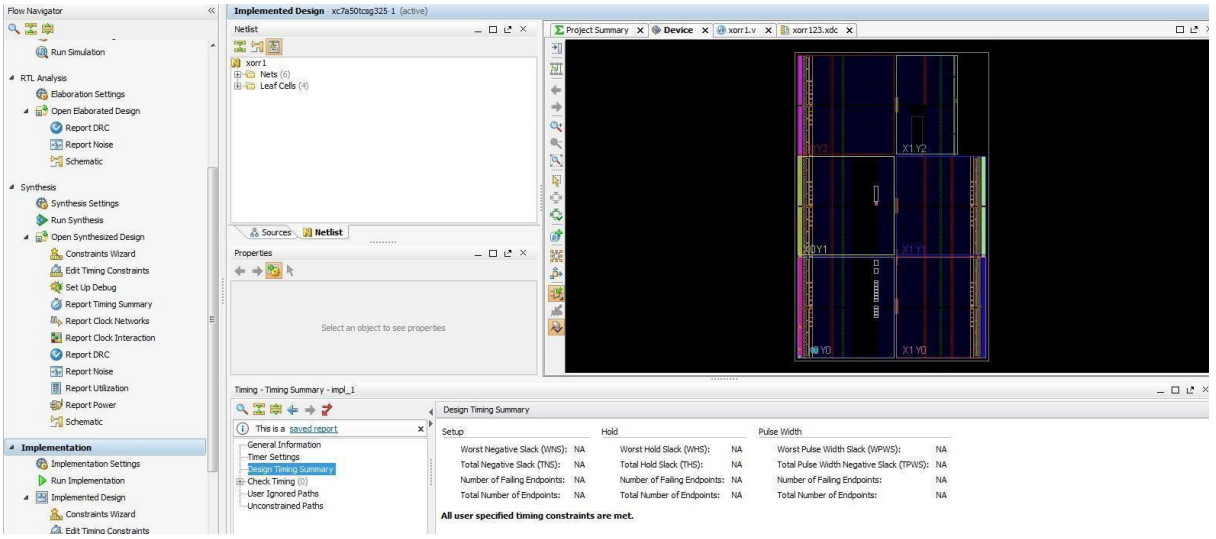

Synthesis
Power - power_1
Summary
On-Chip Power
Total On-Chip Power: 1.368 W
Junction Temperature: 31.5 °C

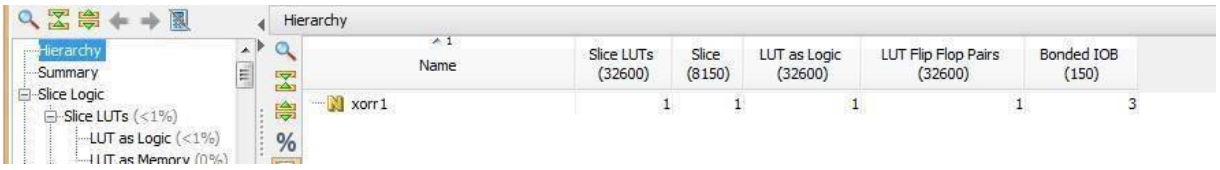

a_IBUF_inst
I
O
IBUF
a
y_OBUF_inst_i_1
I0
I1
O
LUT2
y_OBUF_inst
I
O
OBUF
y
b_IBUF_inst
I
O
IBUF
b

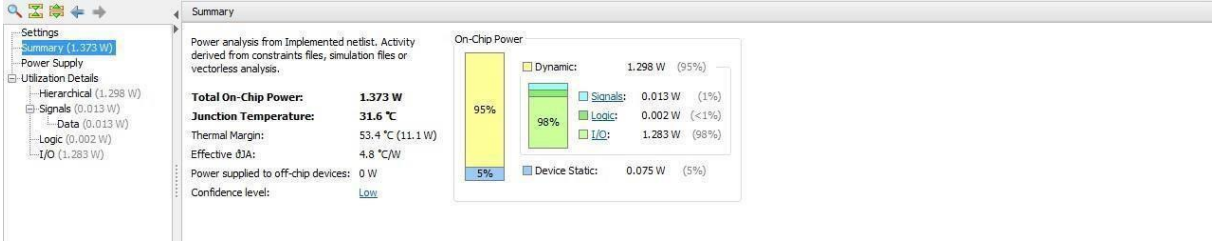

Flow Navigator
Synthesized Design
Device Constraints
Schematic
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Report DRC
Report Noise
Schematic
Synthesis
Synthesis Settings
Run Synthesis
Synthesized Design
Constraints Wizard
Edit Timing Constraints
Set Up Debug
Report Timing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


Search commands
Running opt_design
Cancel


Implementation Completed
Implementation successfully completed.
Next
Open Implemented Design
Generate Bitstream
View Reports
Don't show this dialog again
OK
Cancel


Flow Navigator
Implemented Design
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Report DRC
Report Noise
Schematic
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Constraints Wizard
Edit Timing Constraints
Set Up Debug
Report Timing Summary
Report Clock Networks
Report Clock Interaction
Report DRC
Report Noise
Report Utilization
Report Power
Schematic
Implementation
Implementation Settings
Run Implementation
Implemented Design
Constraints Wizard
Edit Timing Constraints
Netlist
xorr1
Nets (6)
Leaf Cells (4)
Sources
Netlist
Properties
Select an object to see properties
Project Summary
Device
xorr1.v
xorr123.xdc
Timing - Timing Summary - impl_1
Design Timing Summary
Setup
Hold
Pulse Width
Worst Negative Slack (WNS): NA
Total Negative Slack (TNS): NA
Number of Failing Endpoints: NA
Total Number of Endpoints: NA
Worst Hold Slack (WHS): NA
Total Hold Slack (THS): NA
Number of Failing Endpoints: NA
Total Number of Endpoints: NA
Worst Pulse Width Slack (WPWS): NA
Total Pulse Width Negative Slack (TPWS): NA
Number of Failing Endpoints: NA
Total Number of Endpoints: NA
All user specified timing constraints are met.


Hierarchy
Summary
Slice Logic
Slice LUTs (<1%)
LUT as Logic (<1%)
Name
Slice LUTs (32600)
Slice (8150)
LUT as Logic (32600)
LUT Flip Flop Pairs (32600)
Bonded IOB (150)
xorr1
1
1
1
1
3


Summary
Settings
Summary (1.373 W)
Power Supply
Utilization Details
Hierarchical (1.298 W)
Signals (0.013 W)
Data (0.013 W)
Logic (0.002 W)
I/O (1.283 W)
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.
Total On-Chip Power: 1.373 W
Junction Temperature: 31.6 °C
Thermal Margin: 53.4 °C (11.1 W)
Effective θJA: 4.8 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Low
On-Chip Power
95%
5%
98%
Dynamic: 1.298 W (95%)
Signals: 0.013 W (1%)
Logic: 0.002 W (<1%)
I/O: 1.283 W (98%)
Device Static: 0.075 W (5%)

RESULTS

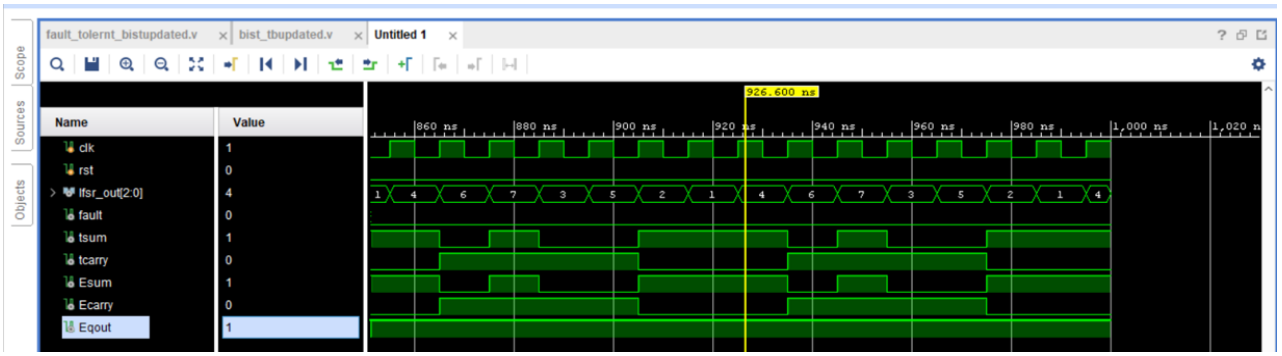


Fig: simulated waveform without fault

Fig: simulated waveform with fault

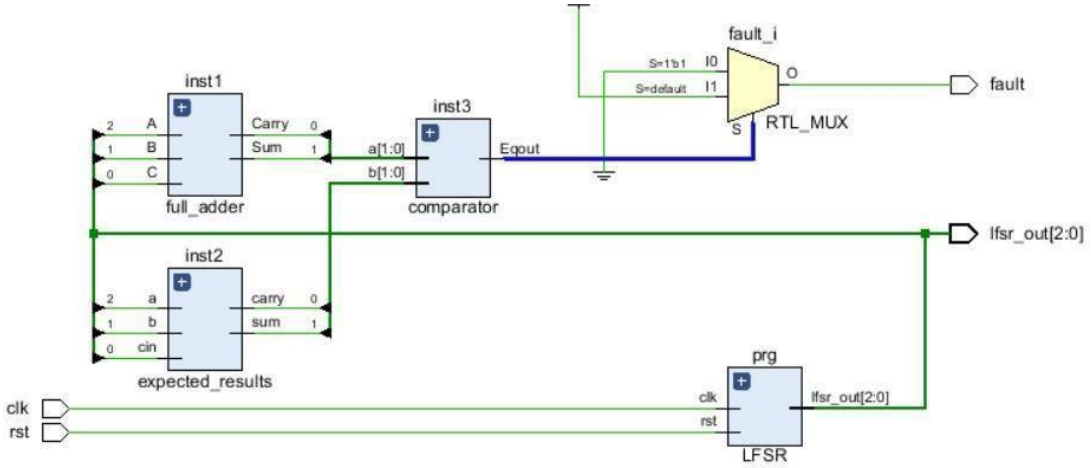


FIG:RTL Schema

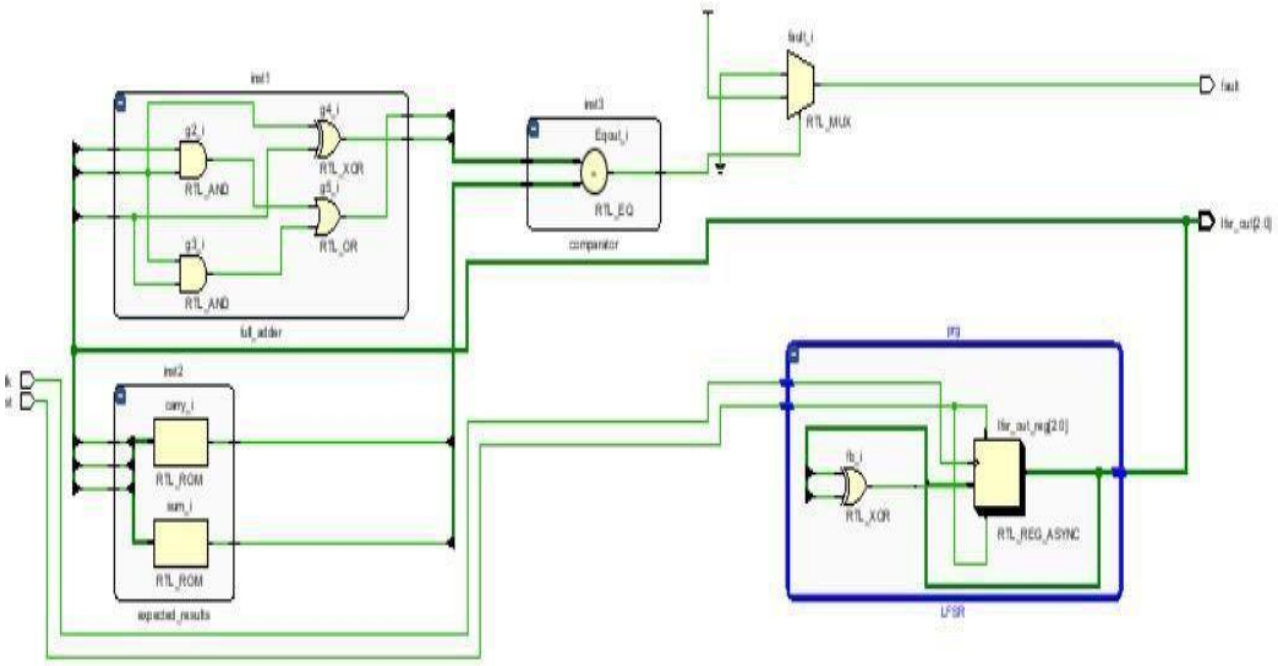


FIG:INTERNAL RTL SCHEMATIC

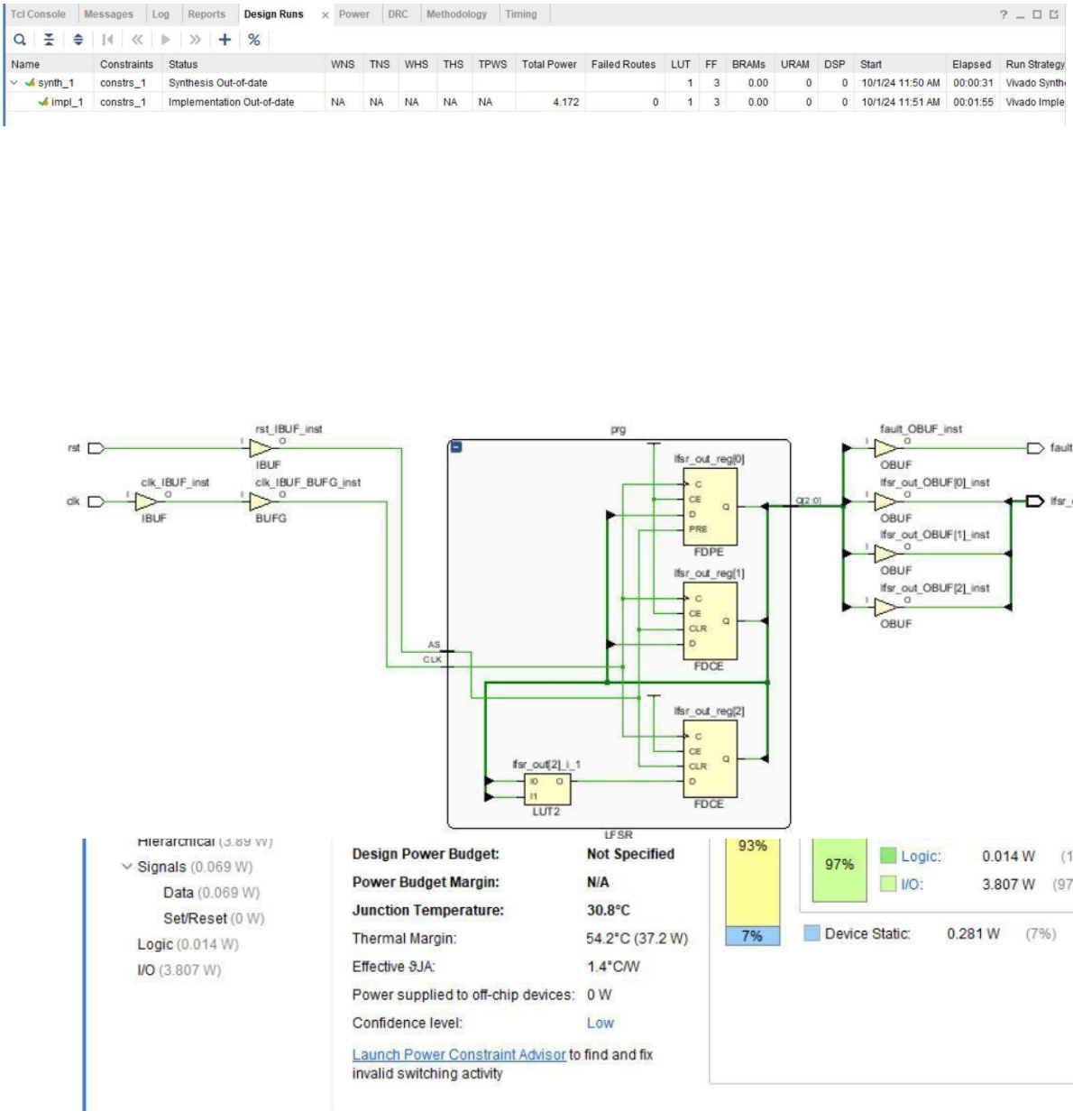


Fig : technology schematic

CHAPTER 7

CONCLUSION AND FUCTURE SCOPE

CONCLUSION

The method for reducing test vectors and performing testing at speed testing is discussed in this paper. A circuit can be simulated in the presence of faults. The fault may be stuck-at-0 or stuck at 1 simulates at a functional level. The simulator is used for design verification and verifying its timing analysis. The process to find test vectors for single stuck at fault using path sensitization method and for multiple stuck- at faults using the Boolean difference method through forming a fault table. The number of test vectors can be minimized to find each line of fault in the circuit. Test vector becomes a major issue for the power consumption of the circuit. The methods for reducing test vector is designed and implemented by Hardware Description Language (HDL) simulation tool

FUCTURE SCOPE

detecting faults in a full adder using Built-In Self-Test (BIST) lies in its potential to enhance fault detection techniques in digital circuits, making testing more efficient and reliable. This can be extended to more complex arithmetic circuits like multipliers and ALUs, enabling hierarchical and scalable testing solutions. Incorporating low-power BIST designs would cater to the needs of energy-efficient devices, while integrating machine learning could improve fault classification and prediction. Additionally, its application in real-time fault detection for safety-critical systems and in self-healing technologies can significantly boost circuit reliability, particularly in fields like aerospace, automotive, and IoT-based devices.

REFERENCES

[1] L.Bushnell, and Vishwani D.Agrawal, "Essentials of Electronic Testing for Digital, Memory & Mixed–Signal VLSI Circuits",:Springer, 2002

[2] MironAbramovici, Melvin A.Breuer, and Arthur D.Friedman. " Digital Testing and Testable Design", Piscataway-New Jersey: IEEE Press,1994.

[3] N. Weste and K. Eshragian, "Principles of CMOS VLSI Design", A Systems Perspective, second ed. Addison-Wesley, 1994.

[4] Vinod Kumar Khera, R.K. Sharma, and A.K. Gupta " A heuristic fault based optimization approach to reduce test vectors count in VLSI testing", Journal of King Saud University – Computer and Information Sciences (2019)

[5] K.L.V.Ramana Kumari, M.Asha Rani and N. Balaji "Design Verification And Test Vector Minimization Using Heuristic Method Of A Ripple Carry Adder", International Journal on Cybernetics & Informatics (IJCI) vol. 5, No. 4, pp. 307-313, August 2016.

[6] Dhanabalan G and Tamil Selvi S, "Design of parallel conversion multichannel analog to digital converter for scan time reduction of programmable logic controller using FPGA", Computer standards and interfaces, vol. 39, pp. 12 – 21, 2015.

[7] Mary D. Pulukuri and Charles E. Stroud " On Built-In Self-Test for Adders", J Electron Test (2009) 25:343–346 DOI 10.1007/s10836- 009-5114-6.

[8] Ashok Kumar, Rahul Raj Choudhary, and Pooja Bhardwaj, "Universal Pattern Set for Arithmetic

Circuits", International Journal of Computer Applications (0975 – 8887) vol. 40– No.15, February 2012.

[9] S Jayanthy, and MC Bhuvaneswari, "Delay Fault Testing of VLSI Circuits", – Springer 2019, Test Generation of Crosstalk Delay Faults, chapter-2, pp 15-35.

[10] Matthias Sauer, Jie Jiang, Sven Reimer, Kohei Miyase, Xiaoqing Wen, Bernd Becker and Ilia Polian, "On Optimal Power-aware Path Sensitization", IEEE 25th Asian Test Symposium, 2016, pp. 179-184.

[11] Nelson, Nagle, Carroll, and Irwin",Digital Logic Circuit Analysis & Design" Prentice-Hall,1995, Chapter 12, pp. 739 to 757.

[12] Mojtaba Valinataj, Abbas Mohammadnezhad, and Jari Nurmi, "A Low-Cost High-Speed Self- Checking Carry Select Adder with Multiple Fault Detection", ELSEVIER Microelectronics Journal,

Volume 81,November 2018, pp. 16-27

[13] Henley Court Pullman,"Genesys™ FPGA Board Reference Manual", Revised April 11, 2016.This manual applies to the Genesys rev.