CHAPTER 1

INTRODUCTION

1.1 AIM

The Main Objective of this project is to design and test 16-bit barrel shifter.

1.2 SCOPE

Barrel shifters are used for shifting and rotating data which is required in several applications like floating point adders, variable-length coding, and bit-indexing. Barrel shifters are often utilized by embedded digital signal processors and general-purpose processors to manipulate data.

A barrel shifter is a digital circuit that can shift a data word by a specified number of bits in one clock cycle. It can be implemented as a sequence of multiplexers (mux.), and in such an implementation the output of one mux is connected to the input of the next mux in a way that depends on the shift distance.

For example, take a four-bit barrel shifter, with inputs A, B, C and D. The shifter can cycle the order of the bits *ABCD* as *DABC*, *CDAB*, or *BCDA*; in this case, no bits are lost. That is, it can shift all of the outputs up to three positions to the right (and thus make any cyclic combination of A, B, C and D). The barrel shifter has a variety of applications, including being a useful component in microprocessors (alongside the ALU).

In many cases most designs only need simple shift registers that shift the input one bit every clock cycle. But what if one wants to shift or rotate data an arbitrary number of bits in a combinatorial design. To shift data an arbitrary number of bits a barrel shifter is used. This document gives a brief overview of an efficient generic barrel shift implementation that rotates input data to the left.

1.2.1 Sixteen-bit Barrel Shifter

The 16-bit barrel shifter has only two levels of CLB, and is, therefore, twice as fast as one using the 2-input multiplexer approach. However, the shift control must be pipelined, since it uses the 4-input multiplexer shown in Figure 1. The first level of multiplexers rotates by 0, 1, 2 or 3

positions, and the second by 0, 4, 8 or 12 positions. Each level requires 16 CLBs, and the total of 32 is the same as for the 2-input approach. The shift control remains binary. Again, this scheme can be expanded to any number of bits using log4N rotators that successively rotate by four times as many bit positions. For sizes that are odd powers of two, the final level should consist of less costly 2-input multiplexers.

1.3 ORGANIZATION

Chapter 2 it reviews about principle of operation. It provides an overview about block diagram and their advantages and disadvantages.

Chapter 3 this chapter explains about the hardware components used in the design of the system. It gives complete detailed matter about each and every component with description, working, features, applications and neat diagram.

Chapter 4 it gives the information about the software used in the project. We use Xilinx and modelsim software for simulating the project, their working is briefly explained.

Chapter 5 gives the conclusion and future scope of the project.

CHAPTER 2

PRINCIPLE OF OPERATION

2.1 INTRODUCTION

In this chapter we are going to explain about the operation which takes place in the "Design and Testing of 16-bit Barrel Shifter". The below figure is the block diagram of the project.

A barrel shifter is simply a bit-rotating shift register. The bits shifted out the MSB end of the register are shifted back into the LSB end of the register. In a barrel shifter, the bits are shifted the desired number of bit positions in a single clock cycle. For example, an eight-bit barrel shifter could shift the data by three positions in a single clock cycle. If the original data was11110000, one clock cycle later the result will be 10000111. Thus, a barrel shifter is implemented by feeding an N-bit data word into N, N-bit-wide multiplexers. An eight-bit barrel shifter is built out of eight flip-flops and eight8-to-1 multiplexers; a 32-bit barrel shifter requires 32 registers and thirty-two, 32-to-1multiplexers, and so on.

In barrel shifters we have many types like four bit barrel shifter, eight bit barrel shifter, etc.

Barrel shifters are often utilized by embedded digital signal processors and general-purpose processors to manipulate data. This paper examines design alternatives for barrel shifters that perform the following functions.

Shift right logical, shift right arithmetic, rotate right, shift left logical, shift left arithmetic, and rotate left. Four different barrel shifter designs are presented and compared in terms of area and delay for a variety of operand sizes. This paper also examines techniques for detecting results that overflow and results of zero in parallel with the shift or rotate operation. Several Java programs are developed to generate structural VHDL models for each of the barrel shifters.

2.2 BLOCK DIAGRAM AND DESCRIPTION

The block diagram shows "Design and Testing of 16-bit barrel shifter**"**. The 16-bit barrel shifter has only two levels of CLB, and is, therefore, twice as fast as one using the 2-input multiplexer

approach.

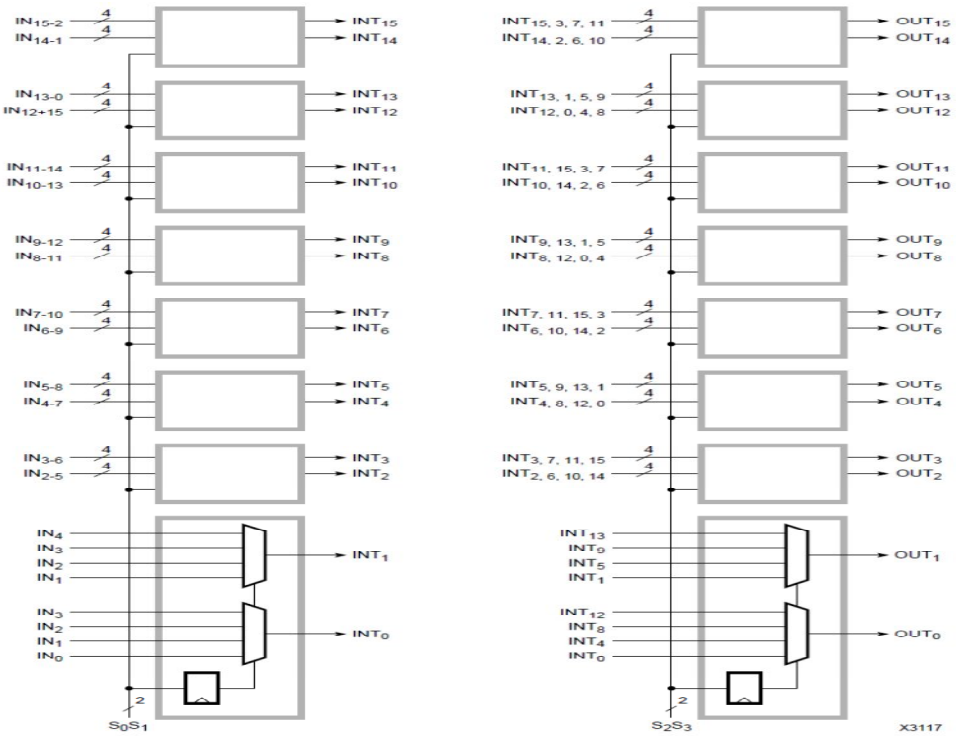


Fig 2.1 Block diagram

However, the shift control must be pipelined, since it uses the 4-input multiplexer. The first level of multiplexers rotates by 0, 1, 2 or 3 positions, and the second by 0, 4, 8 or 12 positions. Each level requires 16 CLBs, and the total of 32 is the same as for the 2-input approach. The shift control remains binary. Again, this scheme can be expanded to any number of bits using log4N rotators that successively rotate by four times as many bit positions. For sizes that are odd powers of two, the final level should consist of less costly 2-input multiplexers.

2.3 ADVANTAGE

- The bits are shifted the desired number of bit positions in a single clock cycle.

2.4 DISADVANTAGE

- To shift more than 16 bits we cannot use this barrel shifter.

CHAPTER 3

HARDWARE DESCRIPTION

3.1 OVERVIEW

In electronics, an adder or summer is a digital circuit that performs addition of numbers. In many computers and other kinds of processors, adders are used not only in the arithmetic logic unit(s), but also in other parts of the processor, where they are used to calculate addresses, table indices, and similar.

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

3.2 HALF ADDER

The half adder adds two one-bit binary numbers A and B. It has two outputs, S and C (the value theoretically carried on to the next addition); the final sum is 2C + S. The simplest half-adder design, pictured on the right, incorporates an XOR gate for S and an AND gate for C. With the addition of an OR gate to combine their carry outputs, two half adders can be combined to make a full adder

A full adder adds binary numbers and accounts for values carried in as well as out. A one-bit full adder adds three one-bit numbers, often written as A, B, and C_{in}; A and B are the operands, and C_{in} is a bit carried in from the next less significant stage. A full adder can be implemented in many different ways such as with a custom transistor-level circuit or composed of other gates. One example implementation is with and $c_{out}=(A.B)+(C_{in}.(A+B))$.

3.3 RIPPLE CARRY ADDER

It is possible to create a logical circuit using multiple full adders to add N-bit numbers. Each full adder inputs a C_{in}, which is the C_{out} of the previous adder. This kind of adder is a ripple carry

adder, since each carry bit "ripples" to the next full adder. Note that the first (and only the first) full adder may be replaced by a half adder.

The layout of a ripple carry adder is simple, which allows for fast design time; however, the ripple carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic. In a 32-bit [ripple carry] adder, there are 32 full adders, so the critical path (worst case) delay is 3 (from input to carry in first adder) + 31 * 2 (for carry propagation in later adders) = 65 gate delays. A design with alternating carry polarities and optimized AND-OR-Invert gates can be about twice as Fas.

3.4 CARRY LOOK AHEAD ADDERS

To reduce the computation time, engineers devised faster ways to add two binary numbers by using carry look ahead adders. They work by creating two signals (P and G) for each bit position, based on if a carry is propagated through from a less significant bit position (at least one input is a '1'), a carry is generated in that bit position (both inputs are '1'), or if a carry is killed in that bit position (both inputs are '0'). In most cases, P is simply the sum output of a half-adder and G is the carry output of the same adder. After P and G are generated the carries for every bit position are created. Some advanced carry look ahead architectures are the Manchester carry chain, Brent–Kung adder, and the Kogge Stone adder.

A carry-look ahead adder (CLA) is a type of adder used in digital logic. A carry look ahead adder improves speed by reducing the amount of time required to determine carry bits. It can be contrasted with the simpler, but usually slower, ripple carry adder for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits (see adder for detail on ripple carry adders). The carry look ahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits. The Kogge-Stone adder and Brent Kung adder are examples of this type of adder.

Charles Babbage recognized the performance penalty imposed by ripple carry and developed mechanisms for anticipating carriage in his computing engines Gerald Rosenberger of IBM filed for a patent on a modern binary carry look ahead adder in 1957.

A ripple-carry adder works in the same way as pencil-and-paper methods of addition. Starting at the rightmost (least significant) digit position, the two corresponding digits are added and a result obtained. It is also possible that there may be a carry out of this digit position (for example, in pencil-and-paper methods, "9+5=4, carry 1"). Accordingly all digit positions other than the rightmost need to take into account the possibility of having to add an extra 1, from a carry that has come in from the next position to the right.

Carry look ahead depends on two things

- Calculating, for each digit position, whether that position is going to propagate a carry if one comes in from the right.
- Combining these calculated values to be able to deduce quickly whether, for each group of digits, that group is going to propagate a carry that comes in from the right.

Supposing that groups of 4 digits are chosen. Then the sequence of events goes something like this

- All 1-bit adders calculate their results. Simultaneously, the look ahead units perform their calculations.
- Suppose that a carry arises in a particular group. Within at most 3 gate delays, that carry will emerge at the left-hand end of the group and start propagating through the group to its left.
- If that carry is going to propagate all the way through the next group, the look ahead unit will already have deduced this. Accordingly, *before the carry emerges from the next group* the look ahead unit is immediately (within 1 gate delay) able to tell the *next* group to the left that it is going to receive a carry - and, at the same time, to tell the next look ahead unit to the left that a carry is on its way.

The net effect is that the carries start by propagating slowly through each 4-bit group, just as in a ripple-carry system, but then move 4 times as fast, leaping from one look ahead carry unit to the next. Finally, within each group that receives a carry, the carry propagates slowly within the digits in that group.

The more bits in a group, the more complex the look ahead carry logic becomes, and the more time is spent on the "slow roads" in each group rather than on the "fast road" between the groups (provided by the look ahead carry logic). On the other hand, the fewer bits there are in a group, the more groups have to be traversed to get from one end of a number to the other, and the less acceleration is obtained as a result.

For very large numbers (hundreds or even thousands of bits) look ahead carry logic does not become any more complex, because more layers of super groups and super groups can be added as necessary. The increase in the number of gates is also moderate if all the group sizes are 4, one would end up with one third as many look ahead carry units as there are adders. However, the "slow roads" on the way to the faster levels begin to impose a drag on the whole system (for instance, a 256-bit adder could have up to 24 gate delays in its carry processing), and the mere physical transmission of signals from one end of a long number to the other begins to be a problem. At these sizes carry-save adders are preferable, since they spend no time on carry propagation at all.

3.4.1 Operation

Carry look ahead logic uses the concepts of generating and propagating carries. Although in the context of a carry look ahead adder, it is most natural to think of generating and propagating in the context of binary addition, the concepts can be used more generally than this. In the descriptions below, the word digit can be replaced by bit when referring to binary addition.

The addition of two 1-digit inputs *A* and *B* is said to generate if the addition will always carry, regardless of whether there is an input carry (equivalently, regardless of whether any less significant digits in the sum carry). For example, in the decimal addition 52 + 67, the addition of the tens digits 5 and 6 generates because the result carries to the hundreds digit regardless of whether the ones digit carries (in the example, the ones digit does not carry (2+7=9)).

In the case of binary addition, A+B generates if and only if both A and B are 1. If we write G(A,B) to represent the binary predicate that is true if and only if A + B generates, we have:

G(A,B) =A.B

The addition of two 1-digit inputs *A* and *B* is said to propagate if the addition will carry whenever there is an input carry (equivalently, when the next less significant digit in the sum carries). For example, in the decimal addition 37 + 62, the addition of the tens digits 3 and 6 propagate because the result would carry to the hundreds digit if the ones were to carry (which in this example, it does not). Note that propagate and generate are defined with respect to a single digit of addition and do not depend on any other digits in the sum.

In the case of binary addition, A+B propagates if and only if at least one of *A* or *B* is 1. If we write P(A,B) to represent the binary predicate that is true if and only if A+B propagates, we have:

P(A,B)=A+B

Sometimes a slightly different definition of propagate is used. By this definition *A* + *B* is said to propagate if the addition will carry whenever there is an input carry, but will not carry if there is no input carry. It turns out that the way in which generate and propagate bits are used by the carry look ahead logic, it doesn't matter which definition is used. In the case of binary addition, this definition is expressed by:

$P'(A,B)=A+B$

For binary arithmetic, or is faster than Xor and takes fewer transistors to implement. However, for a multiple-level carry look ahead adder, it is simpler to use $P'(A,B)$.

It will carry precisely when either the addition generates or the next less significant bit carries and the addition propagates. Written in Boolean algebra, with C_i the carry bit of digit *i*, and P_i and G_i the propagate and generate bits of digit *i* respectively,

$C_{i+1}=G_i+(P_i.C_i)$

3.4.2 Implementation details

For each bit in a binary sequence to be added, the Carry Look Ahead Logic will determine whether that bit pair will generate a carry or propagate a carry. This allows the circuit to "pre-process" the two numbers being added to determine the carry ahead of time. Then, when the actual addition is performed, there is no delay from waiting for the ripple carry effect (or time it takes for the carry from the first Full Adder to be passed down to the last Full Adder). Below is

a simple 4-bit generalized Carry Look Ahead circuit that combines with the 4-bit Ripple Carry Adder we used above with some slight adjustments:

For the example provided, the logic for the generate (g) and propagate (p) values are given below. Note that the numeric value determines the signal from the circuit above, starting from 0 on the far left to 3 on the far right

$C_1=G_0+P_0.C_0$

$C_2=G_1+P_1.C_1$

$C_3=G_2+P_2.C_2$

$C_4=G_3+P_3.C_3$

Substituting C_1 into C_2,then C_2 into C_3, then C_3 into C_4 yields the expanded equations

$C_1=G_0+P_0.C_0$

$C_2=G_1+G_0.P_1+C_0.P_0.P_1$

$C_3= G_2+G_1.P_2+G_0.P_1.P_2.P_3+ C_0.P_0.P_1.P_2$

$C_4= G_3+G_2.P_3+G_1.P_2.P_3+G_0.P_1.P_2.P_3+ C_0.P_0.P_1.P_2.P_3$

To determine whether a bit pair will generate a carry, the following logic works

$G_i=A_i.B_i$

To determine whether a bit pair will propagate a carry, either of the following logic statements work

$P_i=A_i+B_i$

$P_i=A_i+B_i$

The reason why this works is based on evaluation of $C_1=G_0+P_0.C_0$. The only difference in the truth tables between(A+B) and (A+B) is when both A and B are 1. However, if both A and B both are 1, then the G_0 term is 1 (since its equation is A.B), and the $P_0.C_0$ term becomes irrelevant. The XOR is used normally within a basic full adder circuit; the OR is an alternate option (for a carry look ahead only) which is far simpler in transistor-count terms.

The Carry Look Ahead 4-bit adder can also be used in a higher-level circuit by having each CLA Logic circuit produce a propagate and generate signal to a higher-level CLA Logic circuit. The group propagate(PG) and group generate (GG) for a 4-bit CLA are:

$PG=P_0.P_1.P_2.P_3$

$GG= G_3+G_2.P_3+G_1.P_2.P_3+G_0.P_1.P_2.P_3$

Putting 4 4-bit CLAs together yields four group propagates and four group generates. A Lookahead Carry Unit (LCU) takes these 8 values and uses identical logic to calculate C_i in the CLAs. The LCU then generates the carry input for each of the 4 CLAs and a fifth equal to $C_{16.}$

The calculation of the gate delay of a 16-bit adder (using 4 CLAs and 1 LCU) is not as straight forward as the ripple carry adder. Starting at time of zero

- Calculation of P_i and G_i is done at time 1
- Calculation of C_i is done at time 3
- Calculation of the PG is done at time 2
- Calculation of the GG is done at time 3
- Calculation of the inputs for the clas from the LCU are done at
    - Time 0 for the first CLA
    - Time 5 for the second CLA
    - Time 5 for the third & fourth CLA
- Calculation of the S_i are done at
    - Time 4 for the first CLA
    - Time 8 for the second CLA
    - Time 8 for the third & fourth CLA
- Calculation of the final carry bit (C_{16}) is done at time 5

The maximum time is 8 gate delays (for $S_{[8-15]}$). A standard 16-bit ripple carry adder would take 31 gate delays.

3.5 MANCHESTER CARRY CHAIN

The Manchester carry chain is a variation of the carry-look ahead adder that uses shared logic to lower the transistor count. As can be seen above in the implementation section, the logic for generating each carry contains all of the logic used to generate the previous carries. A Manchester carry chain generates the intermediate carries by tapping off nodes in the gate that calculates the most significant carry value. A Manchester-carry-chain section generally won't exceed 4 bits.

3.6 CARRY-SAVE ADDER

A carry-save adder is a type of digital adder, used in computer micro architecture to compute the sum of three or more n-bit numbers in binary. It differs from other digital adders in that it outputs two numbers of the same dimensions as the inputs, one which is a sequence of partial sum bits and another which is a sequence of carry bits.

In electronic terms, using binary bits, this means that even if we have *n* one-bit adders at our disposal, we still have to allow a time proportional to *n* to allow a possible carry to propagate from one end of the number to the other. Until we have done this,

- We do not know the result of the addition.
- We do not know whether the result of the addition is larger or smaller than a given number (for instance, we do not know whether it is positive or negative).

A carry look-ahead adder can reduce the delay. In principle the delay can be reduced so that it is proportional to logn, but for large numbers this is no longer the case, because even when carry look-ahead is implemented, the distances that signals have to travel on the chip increase in proportion to n, and propagation delays increase at the same rate. Once we get to the 512-bit to 2048-bit number sizes that are required in public-key cryptography, carry look-ahead is not of much help.

The basic concept

Here is an example of a binary sum

10111010101011011111000000001101

+11011110101011011011111011101111

Carry save arithmetic works by abandoning the binary notation while still working to base 2.It computes the sum digit by digit as

10111010101011011111000000001101

+11011110101011011011111011101111

=21122120202022022122111011102212.

The notation is unconventional but the result is still unambiguous. Moreover, given *n* adders (here, *n*=32 full adders), the result can be calculated in a single tick of the clock, since each digit result does not depend on any of the others.

3.7 CARRY-SAVE ACCUMULATORS

Supposing that we have two bits of storage per digit, we can use a redundant binary representation, storing the values 0, 1, 2, or 3 in each digit position. It is therefore obvious that one more binary number can be added to our carry-save result without overflowing our storage capacity.

The key to success is that at the moment of each partial addition we add three bits:

- 0 or 1, from the number we are adding.
- 0 if the digit in our store is 0 or 2, or 1 if it is 1 or 3.
- 0 if the digit to its right is 0 or 1, or 1 if it is 2 or 3.

To put it another way, we are taking a carry digit from the position on our right, and passing a carry digit to the left, just as in conventional addition; but the carry digit we pass to the left is the result of the previous calculation and not the current one. In each clock cycle, carries only have to move one step along and not *n* steps as in conventional addition. Because signals don't have to move as far, the clock can tick much faster.

There is still a need to convert the result to binary at the end of a calculation, which effectively just means letting the carries travel all the way through the number just as in a conventional adder. But if we have done 512 additions in the process of performing a 512-bit multiplication, the cost of that final conversion is effectively split across those 512 additions, so each addition bears 1/512 of the cost of that final "conventional" addition.

3.7.1 Drawbacks

At each stage of a carry-save addition

1. We know the result of the addition at once.
2. We still do not know whether the result of the addition is larger or smaller than a given number (for instance, we do not know whether it is positive or negative).

This latter point is a drawback when using carry-save adders to implement modular multiplication (multiplication followed by division, keeping the remainder only). If we cannot know whether the intermediate result is greater or less than the modulus, how can we know whether to subtract the modulus or not?

Montgomery multiplication, which depends on the rightmost digit of the result, is one solution; though rather like carry-save addition itself, it carries a fixed overhead so that a sequence of Montgomery multiplications saves time but a single one does not. Fortunately exponentiation, which is effectively a sequence of multiplications, is the most common operation in public-key cryptography.

3.7.2 Technical details

The carry-save unit consists of *n* full adders, each of which computes a single sum and carry bit based solely on the corresponding bits of the three input numbers. Given the three *n* - bit numbers **a**, **b**, and **c**, it produces a partial sum **ps** and a shift-carry **sc**:

$ps_i = a_i + b_i + c_i$

$sc_i = (a_i\ b_i)\ (a_i\ c_i)\ (b_i\ c_i)$

The entire sum can then be computed by:

1. Shifting the carry sequence **sc** left by one place.

2. Appending a zero to the front (most significant bit) of the partial sum sequence **ps**.
3. Using a ripple carry adder to add these two together and produce the resulting $n + 1$-bit value.

When adding together three or more numbers, using a carry-save adder followed by a ripple carry adder is faster than using two ripple carry adders. A carry-save adder, however, produces all of its output values in parallel, and thus has the same delay as a single full-adder. Thus the total computation time (in units of full-adder delay time) for a carry-save adder plus a ripple carry adder is $n + 1$, whereas for two ripple carry adders it would be $2n$.

3.8 CARRY SELECT ADDER

In electronics, a **carry-select adder** is a particular way to implement an adder, which is a logic element that computes the (n+1)-bit sum of two n-bit numbers. The carry-select adder is simple but rather fast, having a gate level depth of O(√n).

The carry-select adder generally consists of two ripple carry adders and a multiplexer. Adding two n-bit numbers with a carry-select adder is done with two adders (therefore two ripple carry adders) in order to perform the calculation twice, one time with the assumption of the carry being zero and the other assuming one.

In the uniform case, the optimal delay occurs for a block size of (√n). When variable, the block size should have a delay, from addition inputs A and B to the carry out, equal to that of the multiplexer chain leading into it, so that the carry out is calculated just in time. The O(√n) delay is derived from uniform sizing, where the ideal number of full-adder elements per block is equal to the square root of the number of bits being added.

The basic building block of a carry-select adder, where the block size is 4. Two 4-bit ripple carry adders are multiplexed together, where the resulting carry and sum bits are selected by the carry-in. Since one ripple carry adder assumes a carry-in of 0, and the other assumes a carry-in of 1, selecting which adder had the correct assumption via the actual carry-in yields the desired result.

3.8.1 Basic building block

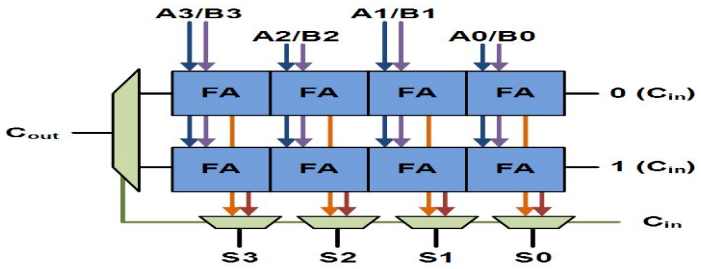


Fig 3.8 carry select adder

3.8.2 Uniform sized adder

A 16-bit carry-select adder with a uniform block size of 4 can be created with three of these blocks and a 4-bit ripple carry adder. Since carry-in is known at the beginning of computation, a carry select block is not needed for the first four bits. The delay of this adder will be four full adder delays, plus three MUX delays.

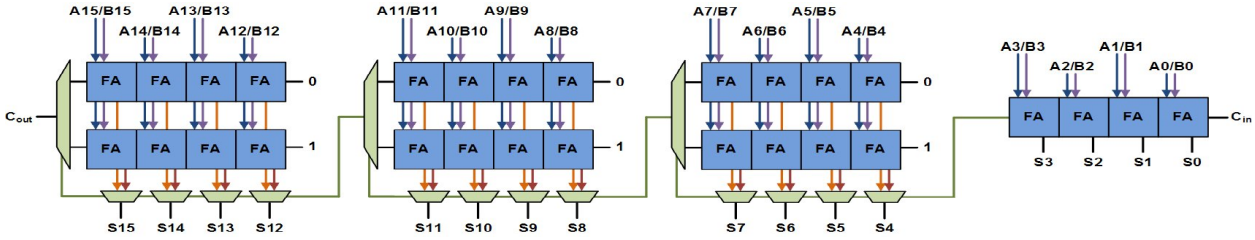


Fig 3.9 uniform sized adder

3.8.3 Variable sized adder

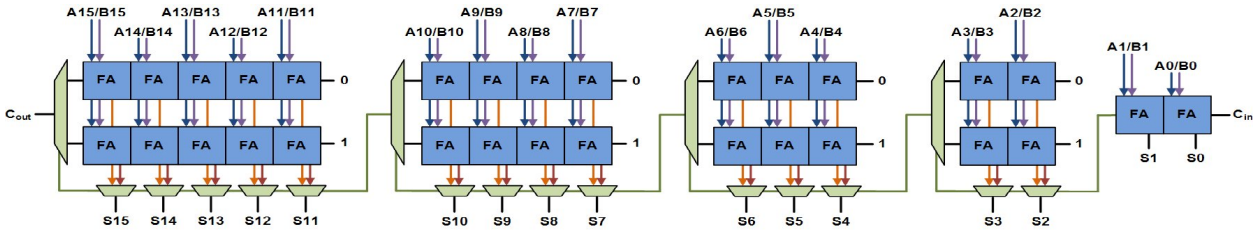


Fig 3.10 variable sized adder

A 16-bit carry-select adder with variable size can be similarly created. Here we show an adder with block sizes of 2-2-3-4-5. This break-up is ideal when the full-adder delay is equal to the MUX delay, which is unlikely. The total delay is two full adder delays, and four mux delays.

3.9 SHIFTERS AND ROTATORS

An n-bit logarithmic barrel shifter uses log2(n) stages [1, 2]. Each bit of the shift amount, B, controls a different stage of the shifter. The data into the stage controlled by bk is shifted by 2k bits if bk = 1; otherwise it is not shifted. Figure 1 shows the block diagram of an 8-bit logical right shifter, which uses three stages with 4-bit, 2-bit, and 1-bit shifts. To optimize the design, each multiplexor that has '0' for one of its inputs can be replaced by a 2-input and gate with the data bit and bk as inputs. A similar unit that performs right rotations, instead of right shifts, can be designed by modifying the connections to the more significant multiplexors. Figure 2 shows the block diagram of an 8-bit right rotator, which uses three stages with 4-bit, 2-bit, and 1-bit rotates. The right rotator and the logical right shifter supply different inputs to the more significant multiplexors. With the rotator, since all of the input bits are routed to the output, there is no longer a need for interconnect lines carrying zeros. Instead, interconnect lines are inserted to enable routing of the 2k low order data bits to the 2k high order multiplexors in the stage controlled by bk. Changing from a non-optimized shifter to a rotator has no impact on the theoretical area or delay. The longer interconnect lines of the rotator, however, can increase both area and delay. The logical right shifter can be extended to also perform shift right arithmetic and rotate right operations by adding additional multiplexors. This approach is illustrated in Figure 3, for an 8-bit right shifter/rotator with three stages of 4-bit, 2-bit, and 1-bit shifts/rotates.

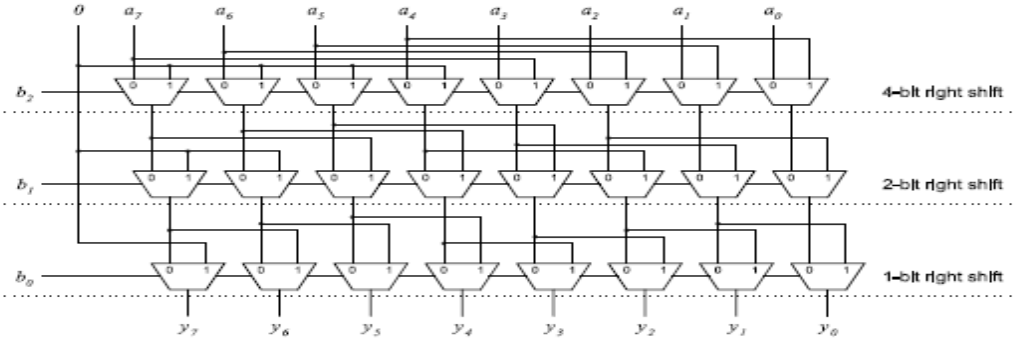


Fig 3.11 8 bit logical right shifter.

Initially, a single multiplexor selects between '0' for logical right shifting and an-1 for arithmetic right shifting to produce s. In the stage controlled by b_k, 2k multiplexors select between s for shifting and the 2k lower bits of the data for rotating.

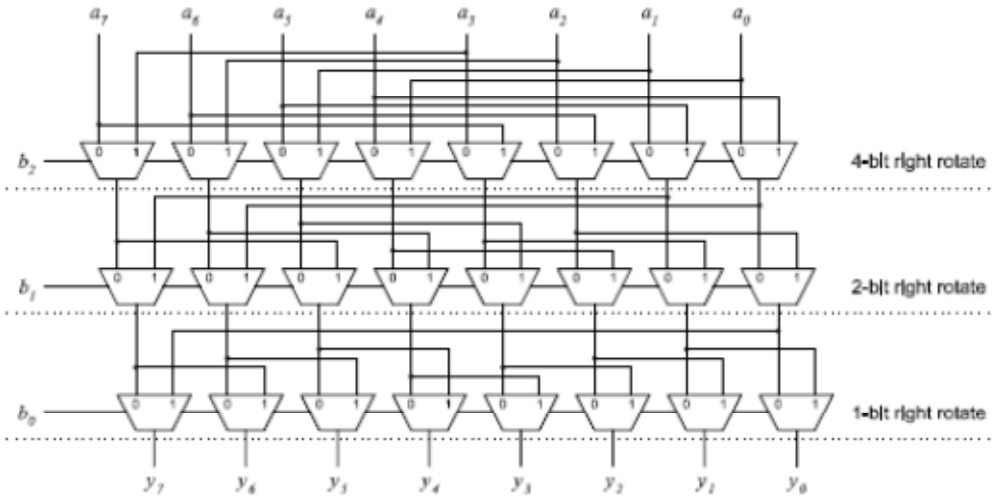


Fig 3.12. 8 bit right rotator.

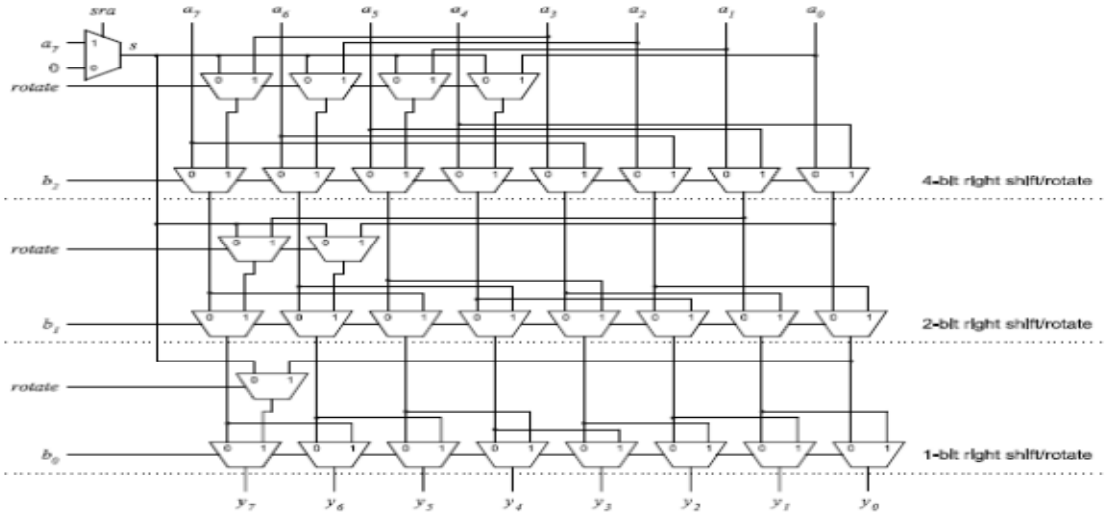


Figure 3.13 8 bit mux-based right shifter/rotator.

A right shifter can be extended to also perform left shift operations by adding a row of n multiplexers both before and after the right shifter [4]. When a left shift operation is performed, these multiplexors reverse the data into and out of the right shifter. When a right shift operation is performed, the data into and out of the shifter is not changed.

3.9.1 Mux-based Data-Reversal Barrel Shifters

The techniques described previously can be combined to form a barrel shifter that performs shift right logical, shift right arithmetic, rotate right, shift left logical, shift left arithmetic, and rotate left. Initially, a row of n multiplexors reverses the order of the data when left = 1 to produce A ^. Then, an n-bit right shifter/rotator performs the right shift or rotate operation on A ^ to produce Y ^. Finally, a row of n multiplexors reverses the data when left = 1 to produce the final result Y. Overflow only occurs when performing a shift left arithmetic operation and one or more of the shifted-out bits differ from the sign bit. A method for detecting overflow in parallel with the shift operation. In each stage, the bits that are shifted out are XOred with the sign bit; when no bits are shifted out, the sign-bit is XOred with itself. The outputs of the XOR gates are then ORed together to produce the overflow Ag, which is '1' when overflow occurs. An additional multiplexor sets ^ y0 to ^ a0 when 111 = 1. The zero flag, which is '1' when Y is zero, is obtained from the logical nor of all of the bits in ^ Y. One disadvantage of this mux-based data-reversal barrel shifter is that the zero flag is not computed until ^ Y is produced.

3.9.2. Mask-based Data-Reversal Barrel Shifters

With this approach, the primary unit that performs the operations is a right rotator and the data-reversal technique is used to support left shift and rotate operations. In parallel with the data reversal and rotation, masks are computed that allow logical and arithmetic shifting to also be performed. With the mask-based data-reversal approach, the overflow and zero flags are computed.

Implementation

A barrel shifter is often implemented as a cascade of parallel 2×1 multiplexers. For a 4-bit barrel shifter, an intermediate signal is used which shifts by two bits, or passes the same data, based on the value of S[1]. This signal is then shifted by another multiplexer, which is controlled by S[0]

Im=IN,if S[1]==0

=IN<<2,if S[1]==1

OUT=im,if S[0]==0

=im<<1,if S[0]==1

3.10 Barrel Shifter

A barrel shifter is a digital circuit that can shift a data word by a specified number of bits in one clock cycle. It can be implemented as a sequence of multiplexers (mux.), and in such an implementation the output of one mux is connected to the input of the next mux in a way that depends on the shift distance.

For example, take a 4-bit barrel shifter, with inputs A, B, C and D. The shifter can cycle the order of the bits ABCD as DABC, CDAB, or BCDA; in this case, no bits are lost. That is, it can shift all of the outputs up to three positions to the right (and thus make any cyclic combination of A, B, C and D). The barrel shifter has a variety of applications, including being a useful component in microprocessors (alongside the ALU).

Barrel shifters are often utilized by embedded digital signal processors and general-purpose processors to manipulate data. This paper examines design alternatives for barrel shifters that perform the following functions.

Shift right logical, shift right arithmetic, rotate right, shift left logical, shift left arithmetic, and rotate left. Four different barrel shifter designs are presented and compared in terms of area and delay for a variety of operand sizes. This paper also examines techniques for detecting results that overflow and results of zero in parallel with the shift or rotate operation. Several Java programs are developed to generate structural VHDL models for each of the barrel shifters. Keywords: barrel shifters, rotators, masks, data-reversal, overflow detection, zero flag, computer arithmetic.

This section discusses barrel shifter designs. Basic shifter and rotator designs are described first. Mux-based data-reversal barrel shifters, mask-based data-reversal barrel shifters, mask-based two's complement barrel shifters, and mask-based one's complement barrel shifters are then discussed in Sections 3.2 through 3.5. In the following discussion the term multiplexor refers to a 1-bit 2-to-1 multiplexor, unless otherwise stated. The operation performed by the barrel shifters is controlled by a 3-bit opcode, which consists of the bits left, rotate, and arithmetic, as summarized in Table 2. Additional control signals, sra and sla, are set to one when performing shift right arithmetic and shift left arithmetic operations, respectively.

Shifting and rotating data is required in several applications including arithmetic operations, variable-length coding, and bit-indexing. Consequently, barrel shifters, which are capable of shifting or rotating data in a single cycle, are commonly found in both digital signal processors and general-purpose processors. This paper examines design alternatives for barrel shifters that perform the following operations shift right logical, shift right arithmetic, rotate right, shift left logical, shift left arithmetic, and rotate left. These designs are optimized to share hardware for different operations. Techniques are also presented for detecting results that overflow and results of zero in parallel with the shift or rotate operation.

3.10.1 INTRODUCTION

3.10.1.1 Basic Barrel Shifter

A barrel shifter is simply a bit-rotating shift register. The bits shifted out the MSB end of the register are shifted back into the LSB end of the register. In a barrel shifter, the bits are shifted the desired number of bit positions in a single clock cycle. For example, an eight-bit barrel shifter could shift the data by three positions in a single clock cycle. If the original data was11110000, one clock cycle later the result will be 10000111. Thus, a barrel shifter is implemented by feeding an N-bit data word into N, N-bit-wide multiplexers. An eight-bit barrel shifter is built out of eight flip-flops and eight8-to-1 multiplexers; a 32-bit barrel shifter requires 32 registers and thirty-two, 32-to-1multiplexers, and so on.

3.10.1.2 Barrel shifter designs

This section discusses barrel shifter designs. Basic shifter and rotator designs are described first Mux-based data-reversal barrel shifters, mask-based data-reversal barrel shifters, mask-based two's complement barrel shifters, and mask-based one's complement barrel shifters are then discussed. In the following discussion the term multiplexor refers to a 1-bit 2-to-1 multiplexer, unless otherwise stated. The operation performed by the barrel shifters is controlled by a 3-bit opcode, which consists of the bits left, rotate, and arithmetic. Additional control signals, sra and sla, are set to one when performing shift right arithmetic and shift left arithmetic operations, respectively

3.10.1.3 Four-Bit Barrel Shifters

A four-input barrel shifter has four data inputs, four data outputs and two control inputs that specify rotation by 0,1, 2 or 3 positions. A simple approach would use four 4-input multiplexers, since each output can receive data from any input. This approach yields the best solution only if the select lines can be pipelined, and the 4-inputmultiplexer design described above is used. The complete barrel shifter can be implemented in one level of four CLBs. If the barrel shifter must be fully combinatorial, it is better to decompose the barrel shifter into 2-stages.

The first stage rotates the data by 0 or 1 positions, and the second rotates the result by 0 or 2 positions. Together, these two shifters provide the desired rotations of 0, 1, 2 or 3 positions. As in the previous design, four CLBs are required, but the number of levels increases to two. A combinatorial 4-input multiplexer approach would have used six CLBs in two levels. This binary decomposition scheme can be used for any number of bits. The number of levels required for an N-bit shifter is log2N, rounded to the next higher number if N is not a power of two. Each level requires N/2 CLBs. The first level rotates 0 or 1 positions, and subsequent levels each rotate by twice as many positions as the preceding level. The select bits to each level form a binary-encoded shift control. For example, an 8-bit barrel shifter can be implemented in three levels of 2-input multiplexers that rotate by 1, 2 and 4 positions.

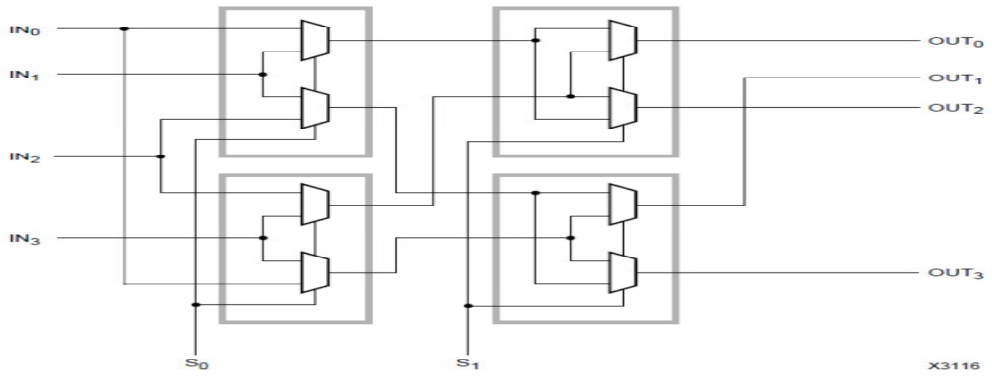


Fig 3.14. 4 bit barrel shifter

Each level requires four CLBs, for a total of 12. For a 12-input barrel shifter, four levels of multiplexer are required. These multiplexers rotate by 1, 2, 4 and 8 positions, and require a total of 24 CLBs.

3.10.1.4 Eight-bit Barrel Shifter

To implement the eight 8-to-1 multiplexors in an eight-bit barrel shifter, it will require two slices per multiplexer, for a total of 16 slices. In the Virtex-II architecture, this uses four CLBs. It will also require an additional CLB for the registering of the outputs. These can be absorbed into the multiplexer CLBs. Virtex-II devices have embedded multipliers, and the functionality of an eight-bit barrel shifter can be implemented in a single MULT18X18. Note, the control bus "SHIFT[7:0]", is a one-hot encoding of the shift desired.

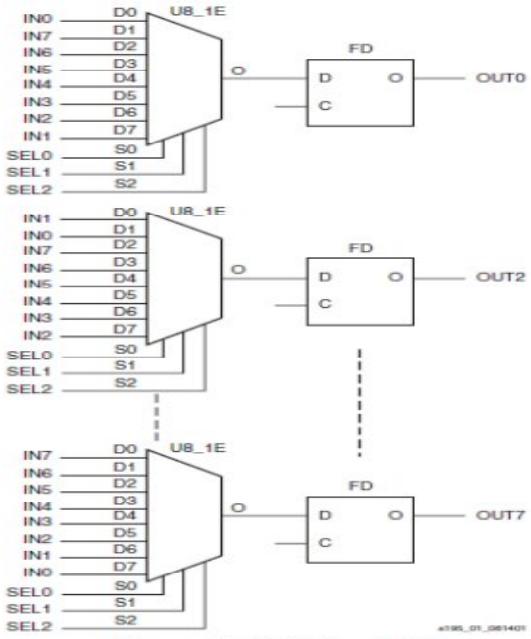


Figure 1: **Eight-Bit Barrel Shifter**

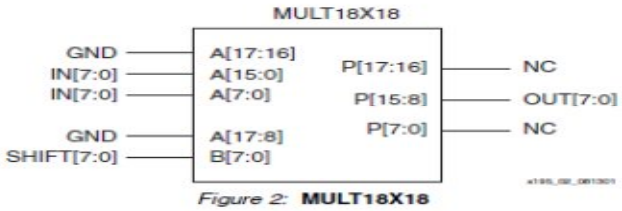


Figure 2: **MULT18X18**

Fig 3.15 8 bit barrel shifter

For example, 0000 0001 causes a multiplication by one, or a shift of zero; 0000 0010 causes a multiplication by two, or a shift of "1", 0000 0100 causes a multiplication by four, or a shift of "2", and so on.

3.10.1.5 Sixteen-bit Barrel Shifter

The 16-bit barrel shifter shown in Figure 6 has only two levels of CLB, and is, therefore, twice as fast as one using the 2-input multiplexer approach.

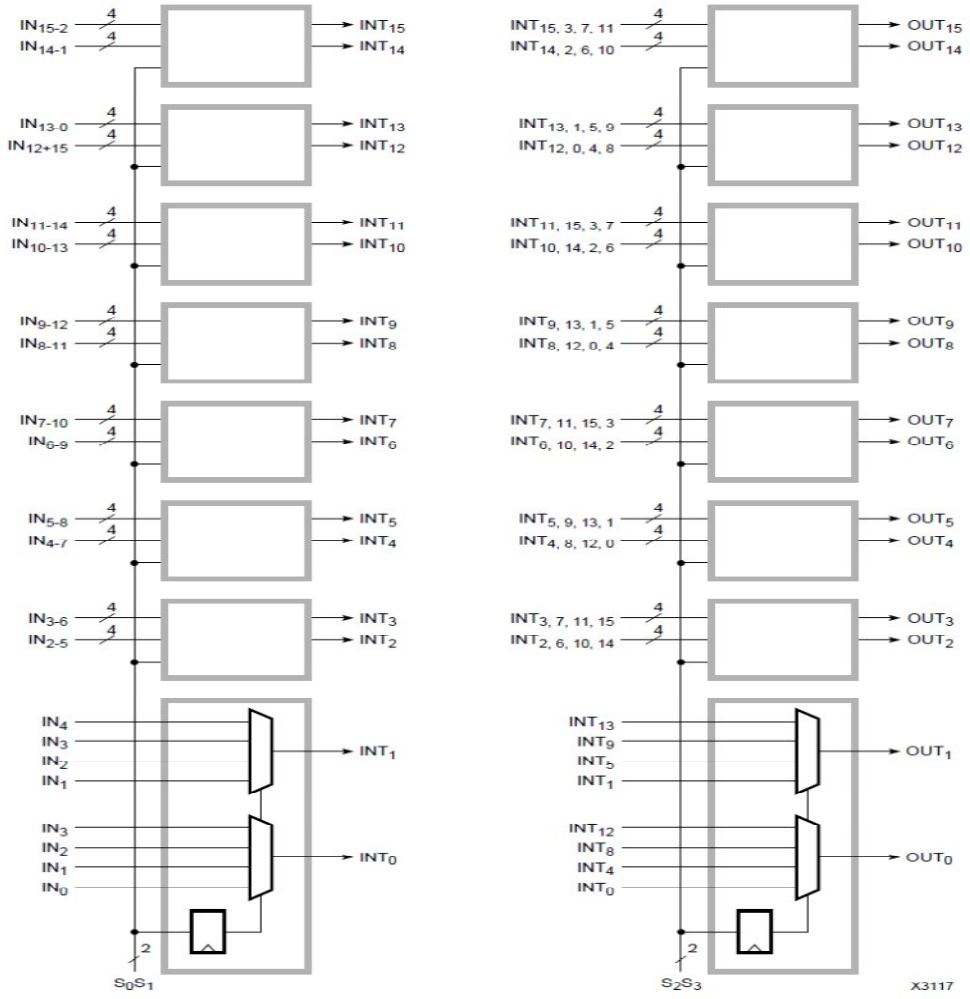


Fig 3.16 16 bit Barrel Shifter

However, the shift control must be pipelined, since it uses the 4-input multiplexer. The first level of multiplexers rotates by 0, 1, 2 or 3 positions, and the second by 0, 4, 8 or 12 positions. Each level requires 16 CLBs, and the total of 32 is the same as for the 2-input approach. The shift control remains binary. Again, this scheme can be expanded to any number of bits using log4N rotators that successively rotate by four times as many bit positions. For sizes that are odd powers of two, the final level should consist of less costly 2-input multiplexers.

CHAPTER 4

SOFTWARE DESCRIPTION

4.1 INTRODUCTION

Very-large-scale integration (VLSI) is the process of creating integrated circuits by combining thousands of transistors into a single chip.

VLSI began in the 1970s when complex semiconductor and communication technologies were being developed. The microprocessor is a VLSI device.

(a) Development

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

At one time, there was an effort to name and calibrate various levels of large-scale integration above VLSI. Terms like ultra-large-scale integration (ULSI) were used. But the huge number of gates and transistors available on common devices has rendered such fine distinctions moot. Terms suggesting greater than VLSI levels of integration are no longer in widespread use.

As of early 2008, billion-transistor processors are commercially available. This became more commonplace as semiconductor fabrication advanced from the then-current generation of 65 nm processes. Current designs, unlike the earliest devices, use extensive design automation and automated logic synthesis to lay out the transistors, enabling higher levels of complexity in the resulting logic functionality. Certain high-performance logic blocks like the SRAM (static random-access memory) cell, are still designed by hand to ensure the highest efficiency. VLSI

technology may be moving toward further radical miniaturization with introduction of NEMS technology.

(b) Structured Design

Structured VLSI design is a modular methodology originated by Carver Mead and Lynn Conway for saving microchip area by minimizing the interconnect fabrics area. This is obtained by repetitive arrangement of rectangular macro blocks which can be interconnected using wiring by abutment. An example is partitioning the layout of an adder into a row of equal bit slices cells. In complex designs this structuring may be achieved by hierarchical nesting.

Structured VLSI design had been popular in the early 1980s, but lost its popularity later because of the advent of placement and routing tools wasting a lot of area by routing, which is tolerated because of the progress of Moore's Law. When introducing the hardware description language KARL in the mid' 1970s, Reiner Hartenstein coined the term "structured VLSI design" (originally as "structured LSI design"), echoing Edsger Dijkstra's structured programming approach by procedure nesting to avoid chaotic spaghetti-structured programs.

(c) Challenges

As microprocessors become more complex due to technology scaling, microprocessor designers have encountered several challenges which force them to think beyond the design plane, and look ahead to post-silicon:

- Process variation, As photolithography techniques tend closer to the fundamental laws of optics, achieving high accuracy in doping concentrations and etched wires is becoming more difficult and prone to errors due to variation. Designers now must simulate across multiple fabrication process corners before a chip is certified ready for production.
- Stricter design rules, Due to lithography and etches issues with scaling, design rules for layout have become increasingly stringent. Designers must keep ever more of these rules in mind while laying out custom circuits. The overhead for custom design is now reaching a tipping point, with many design houses opting to switch to electronic design automation (EDA) tools to automate their design process.

- Timing/design closure, As clock frequencies tend to scale up, designers are finding it more difficult to distribute and maintain low clock skew between these high frequency clock across the entire chip. This has led to a rising interest in multi core and multiprocessor architectures, since an overall speedup can be obtained by lowering the clock frequency and distributing processing.
- First-pass success , As die sizes shrink (due to scaling), and wafer sizes go up (due to lower manufacturing costs), the number of dies per wafer increases, and the complexity of making suitable photo masks goes up rapidly. A mask set for a modern technology can cost several million dollars. This non-recurring expense deters the old iterative philosophy involving several "spin-cycles" to find errors in silicon, and encourages first-pass silicon success. Several design philosophies have been developed to aid this new design flow, including design for manufacturing (DFM), design for test (DFT), and Design for X.

2.5 INTRODUCTION TO VERILOG

In the electronic design industry, Verilog is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design, verification and implementation of digital logic chips at the register-transfer level of abstraction. It is also used in the verification of analog and mixed digital signals.

2.5.1 Overview

Hardware description languages such as Verilog differ from software programming languages because they include ways of describing the propagation of time and signal dependencies (sensitivity). There are two assignment operators, a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables (in any general programming language we need to define some temporary storage spaces for the operands to be operated on subsequently; those are temporary storage variables). Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous productivity improvement for circuit designers who

were already using graphical schematic capture software and specially-written software programs to document and simulate electronic circuits.

The designers of Verilog wanted a language with syntax similar to the C programming language, which was already widely used in engineering software development. Verilog is case-sensitive, has a basic preprocessor (though less sophisticated than that of ANSI C/C++), and equivalent control flow keywords (if/else, for, while, case, etc.), and compatible operator precedence. Syntactic differences include variable declaration (Verilog requires bit-widths on net/reg types), demarcation of procedural blocks (begin/end instead of curly braces {}), and many other minor differences.

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. But the blocks themselves are executed concurrently, qualifying Verilog as a dataflow language.

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

2.5.2 HISTORY

2.5.2.1 Beginning

Verilog was the first modern hardware description language to be invented. It was created by Phil Moorby and Prabhu Goel during the winter of 1983/1984. The wording for this process was "Automated Integrated Design Systems" (later renamed to Gateway Design Automation in 1985) as a hardware modeling language. Gateway Design Automation was purchased by Cadence Design Systems in 1990. Cadence now has full proprietary rights to Gateway's Verilog and the Verilog-XL, the HDL-simulator that would become the de-facto standard (of

Verilog logic simulators) for the next decade. Originally, Verilog was intended to describe and allow simulation; only afterwards was support for synthesis added.

2.5.2.2 Verilog 95

With the increasing success of VHDL at the time, Cadence decided to make the language available for open standardization. Cadence transferred Verilog into the public domain under the Open Verilog International (OVI) (now known as Accellera) organization. Verilog was later submitted to IEEE and became IEEE Standard 1364-1995, commonly referred to as Verilog-95.

In the same time frame Cadence initiated the creation of Verilog-A to put standards support behind its analog simulator Spectre. Verilog-A was never intended to be a standalone language and is a subset of Verilog-AMS which encompassed Verilog-95.

2.5.2.3 Verilog 2001

Extensions to Verilog-95 were submitted back to IEEE to cover the deficiencies that users had found in the original Verilog standard. These extensions became IEEE Standard 1364-2001 known as Verilog-2001.

Verilog-2001 is a significant upgrade from Verilog-95. First, it adds explicit support for (2's complement) signed nets and variables. Previously, code authors had to perform signed operations using awkward bit-level manipulations (for example, the carry-out bit of a simple 8-bit addition required an explicit description of the Boolean algebra to determine its correct value). The same function under Verilog-2001 can be more succinctly described by one of the built-in operators: +, -, /, *, >>>. A generate/endgenerate construct (similar to VHDL's generate/endgenerate) allows Verilog-2001 to control instance and statement instantiation through normal decision operators (case/if/else). Using generate / end generate, Verilog-2001 can instantiate an array of instances, with control over the connectivity of the individual instances. File I/O has been improved by several new system tasks. And finally, a few syntax additions were introduced to improve code readability (e.g. always @*, named parameter override, C-style function/task/module header declaration).

Verilog-2001 is the dominant flavor of Verilog supported by the majority of commercial EDA software packages.

2.5.2.4 Verilog-2005

Not to be confused with System Verilog, Verilog 2005 (IEEE Standard 1364-2005) consists of minor corrections, spec clarifications, and a few new language features (such as the wire keyword).

A separate part of the Verilog standard, Verilog-AMS, attempts to integrate analog and mixed signal modeling with traditional Verilog.

2.5.2.5 System Verilog

SystemVerilog is a superset of Verilog-2005, with many new features and capabilities to aid design verification and design modeling. As of 2009, the SystemVerilog and Verilog language standards were merged into SystemVerilog 2009 (IEEE Standard 1800-2009).

In the late 1990s, the Verilog Hardware Description Language (HDL) became the most widely used language for describing hardware for simulation and synthesis. However, the first two versions standardized by the IEEE (1364-1995 and 1364-2001) had only simple constructs for creating tests. As design sizes outgrew the verification capabilities of the language, commercial Hardware Verification Languages (HVL) such as Open Vera and e were created. Companies that did not want to pay for these tools instead spent hundreds of man-years creating their own custom tools. This productivity crisis (along with a similar one on the design side) led to the creation of Accellera, a consortium of EDA companies and users who wanted to create the next generation of Verilog. The donation of the Open-Vera language formed the basis for the HVL features of SystemVerilog.Accellera's goal was met in November 2005 with the adoption of the IEEE standard P1800-2005 for SystemVerilog, IEEE (2005).

Some of the typical features of an HVL that distinguish it from a Hardware Description Language such as Verilog or VHDL are

- Constrained-random stimulus generation
- Functional coverage
- Higher-level structures, especially Object Oriented Programming
- Multi-threading and interprocess communication
- Support for HDL types such as Verilog's 4-state values

Tight integration with event-simulator for control of the design

There are many other useful features, but these allow you to create test benches at a higher level of abstraction than you are able to achieve with an HDL or a programming language such as C.

System Verilog provides the best framework to achieve coverage-driven verification (CDV). CDV combines automatic test generation, self-checking test benches, and coverage metrics to significantly reduce the time spent verifying a design. The purpose of CDV is to:

- Eliminate the effort and time spent creating hundreds of tests.
- Ensure thorough verification using up-front goal setting.
- Receive early error notifications and deploy run-time checking and error analysis to simplify debugging.

2.6 CONSTANTS

The definition of constants in Verilog supports the addition of a width parameter. The basic syntax is:

<*Width in bits*>'<*base letter*><*number*>

Examples:

- 12'h123 - Hexadecimal 123 (using 12 bits)
- 20'd44 - Decimal 44 (using 20 bits - 0 extension is automatic)
- 4'b1010 - Binary 1010 (using 4 bits)
- 6'o77 - Octal 77 (using 6 bits)

2.6.1 Synthesizable Constructs

There are several statements in Verilog that have no analog in real hardware, e.g. $display. Consequently, much of the language can not be used to describe hardware. The examples presented here are the classic subset of the language that has a direct mapping to real gates.

// Mux examples - Three ways to do the same thing.

// The first example uses continuous assignment

```
wire out;
assign out = sel ? a : b;
 // the second example uses a procedure
// to accomplish the same thing.
 reg out;
always @(a or b or sel)
  begin
    case(sel)
      1'b0: out = b;
      1'b1: out = a;
    endcase
  end
 // Finally - you can use if/else in a
// procedural structure.
reg out;
always @(a or b or sel)
  if (sel)
    out = a;
  else
    out = b;
```

The next interesting structure is a transparent latch; it will pass the input to the output when the gate signal is set for "pass-through", and captures the input and stores it upon transition of the gate signal to "hold". The output will remain stable regardless of the input signal while the gate is set to "hold". In the example below the "pass-through" level of the gate would be when the value of the if clause is true, i.e. gate = 1. This is read "if gate is true, the din is fed to latch_out

continuously." Once the if clause is false, the last value at latch_out will remain and is independent of the value of din.

EX6: // Transparent latch example

```
 reg out;
always @(gate or din)
 if(gate)
   out = din; // Pass through state
   // Note that the else isn't required here. The variable
   // out will follow the value of din while gate is high.
   // When gate goes low, out will remain constant.
```

The flip-flop is the next significant template; in Verilog, the D-flop is the simplest, and it can be modeled as:

```
reg q;
always @(posedge clk)
  q <= d;
```

The significant thing to notice in the example is the use of the non-blocking assignment. A basic rule of thumb is to use **<=** when there is a **posedge** or **negedge** statement within the always clause.

A variant of the D-flop is one with an asynchronous reset; there is a convention that the reset state will be the first if clause within the statement.

```
reg q;
always @(posedge clk or posedge reset)
  if(reset)
    q <= 0;
  else
```

```
  q <= d;
```

The next variant is including both an asynchronous reset and asynchronous set condition; again the convention comes into play, i.e. the reset term is followed by the set term.

```
reg q;
always @(posedge clk or posedge reset or posedge set)
 if(reset)
   q <= 0;
 else
 if(set)
   q <= 1;
 else
   q <= d;
```

Note: If this model is used to model a Set/Reset flip flop then simulation errors can result. Consider the following test sequence of events.

- Reset goes high
- Clk goes high
- Set goes high
- Clk goes high again
- Reset goes low followed by
- Set going low.

Assume no setup and hold violations.

In this example the always @ statement would first execute when the rising edge of reset occurs which would place q to a value of 0. The next time the always block executes would be the rising edge of clk which again would keep q at a value of 0. The always block then executes when set goes high which because reset is high forces q to remain at 0. This condition may or

may not be correct depending on the actual flip flop. However, this is not the main problem with this model. Notice that when reset goes low, that set is still high. In a real flip flop this will cause the output to go to a 1. However, in this model it will not occur because the always block is triggered by rising edges of set and reset - not levels. A different approach may be necessary for set/reset flip flops.

Note that there are no "initial" blocks mentioned in this description. There is a split between FPGA and ASIC synthesis tools on this structure. FPGA tools allow initial blocks where reg values are established instead of using a "reset" signal. ASIC synthesis tools don't support such a statement. The reason is that an FPGA's initial state is something that is downloaded into the memory tables of the FPGA. An ASIC is an actual hardware implementation.

2.7 INITIAL VS ALWAYS

There are two separate ways of declaring a Verilog process. These are the **always** and the **initial** keywords. The **always** keyword indicates a free-running process. The **initial** keyword indicates a process executes exactly once. Both constructs begin execution at simulator time 0, and both execute until the end of the block. Once an **always** block has reached its end, it is rescheduled (again). It is a common misconception to believe that an initial block will execute before an always block. In fact, it is better to think of the **initial**-block as a special-case of the **always**-block, one which terminates after it completes for the first time.

//Examples:

```
initial
  begin
    a = 1; // Assign a value to reg a at time 0
    #1; // Wait 1 time unit
    b = a; // Assign the value of reg a to reg b
  end


always @(a or b) // Any time a or b CHANGE, run the process
```

```
begin
  if (a)
    c = b;
  else
    d = ~b;
end // Done with this block, now return to the top (i.e. the @ event-control)

always @(posedge a)// Run whenever reg a has a low to high change
  a <= b;
```

These are the classic uses for these two keywords, but there are two significant additional uses. The most common of these is an **always**keyword without the **@(...)** sensitivity list. It is possible to use always as shown below:

```
always
 begin // Always begins executing at time 0 and NEVER stops
   clk = 0; // Set clk to 0
   #1; // Wait for 1 time unit
   clk = 1; // Set clk to 1
   #1; // Wait 1 time unit
 end // Keeps executing - so continue back at the top of the begin
```

The **always** keyword acts similar to the "C" construct **while(1) {..}** in the sense that it will execute forever.

The other interesting exception is the use of the **initial** keyword with the addition of the **forever** keyword.

2.8 RACE CONDITION

The order of execution isn't always guaranteed within Verilog. This can best be illustrated by a classic example. Consider the code snippet below:

```
initial
  a = 0;
initial
  b = a;
initial
  begin
    #1;
    $display("Value a=%b Value of b=%b",a,b);
  end
```

Depending on the order of execution of the initial blocks, it could be zero and zero, or alternately zero and some other arbitrary uninitialized value. The $display statement will always execute after both assignment blocks have completed, due to the #1 delay.

2.9 OPERATORS

Note: These operators are *not* shown in order of precedence.

Operator	Operator symbols	Operation performed
Bitwise	~	Bitwise NOT (1's complement)
	&	Bitwise AND
	|	Bitwise OR
	^	Bitwise XOR

	~^ or ^~	Bitwise XNOR
Logical	!	NOT
	&&	AND
	||	OR
Reduction	&	Reduction AND
	~&	Reduction NAND
	|	Reduction OR
	~|	Reduction NOR
	^	Reduction XOR
	~^ or ^~	Reduction XNOR
Arithmetic	+	Addition
	-	Subtraction
	-	2's complement
	*	Multiplication
	/	Division
	**	Exponentiation (*Verilog-2001)
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to

	==	Logical equality (bit-value 1'bX is removed from comparison)
	!=	Logical inequality (bit-value 1'bX is removed from comparison)
	===	4-state logical equality (bit-value 1'bX is taken as literal)
	!==	4-state logical inequality (bit-value 1'bX is taken as literal)
Shift	>>	Logical right shift
	<<	Logical left shift
	>>>	Arithmetic right shift (*Verilog-2001)
	<<<	Arithmetic left shift (*Verilog-2001)
Concatenat.	{ , }	Concatenation
Replication	{n{m}}	Replicate value m for n times
Conditiona,	? :	Conditional

Table: 2.10 operators

2.10 SYSTEM TASKS

System tasks are available to handle simple I/O, and various design measurement functions. All system tasks are prefixed with **$** to distinguish them from user tasks and functions. This section presents a short list of the most often used tasks. It is by no means a comprehensive list.

- $display - Print to screen a line followed by an automatic newline.
- $write - Write to screen a line without the newline.
- $swrite - Print to variable a line without the newline.
- $sscanf - Read from variable a format-specified string. (*Verilog-2001)

- $fopen - Open a handle to a file (read or write)
- $fdisplay - Write to file a line followed by an automatic newline.
- $fwrite - Write to file a line without the newline.
- $fscanf - Read from file a format-specified string. (*Verilog-2001)
- $fclose - Close and release an open file handle.
- $readmemh - Read hex file content into a memory array.
- $readmemb - Read binary file content into a memory array.
- $monitor - Print out all the listed variables when any change value.
- $time - Value of current simulation time.
- $dumpfile - Declare the VCD (Value Change Dump) format output file name.
- $dumpvars - Turn on and dump the variables.
- $dumpports - Turn on and dump the variables in Extended-VCD format.
- $random - Return a random value.

4.2 TOOLS

Tools used

- Xilinx
- Modelsim

(a) Xilinx

Xilinx ISE (Integrated Software Environment) is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize ("compile") their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer.

(bModelsim

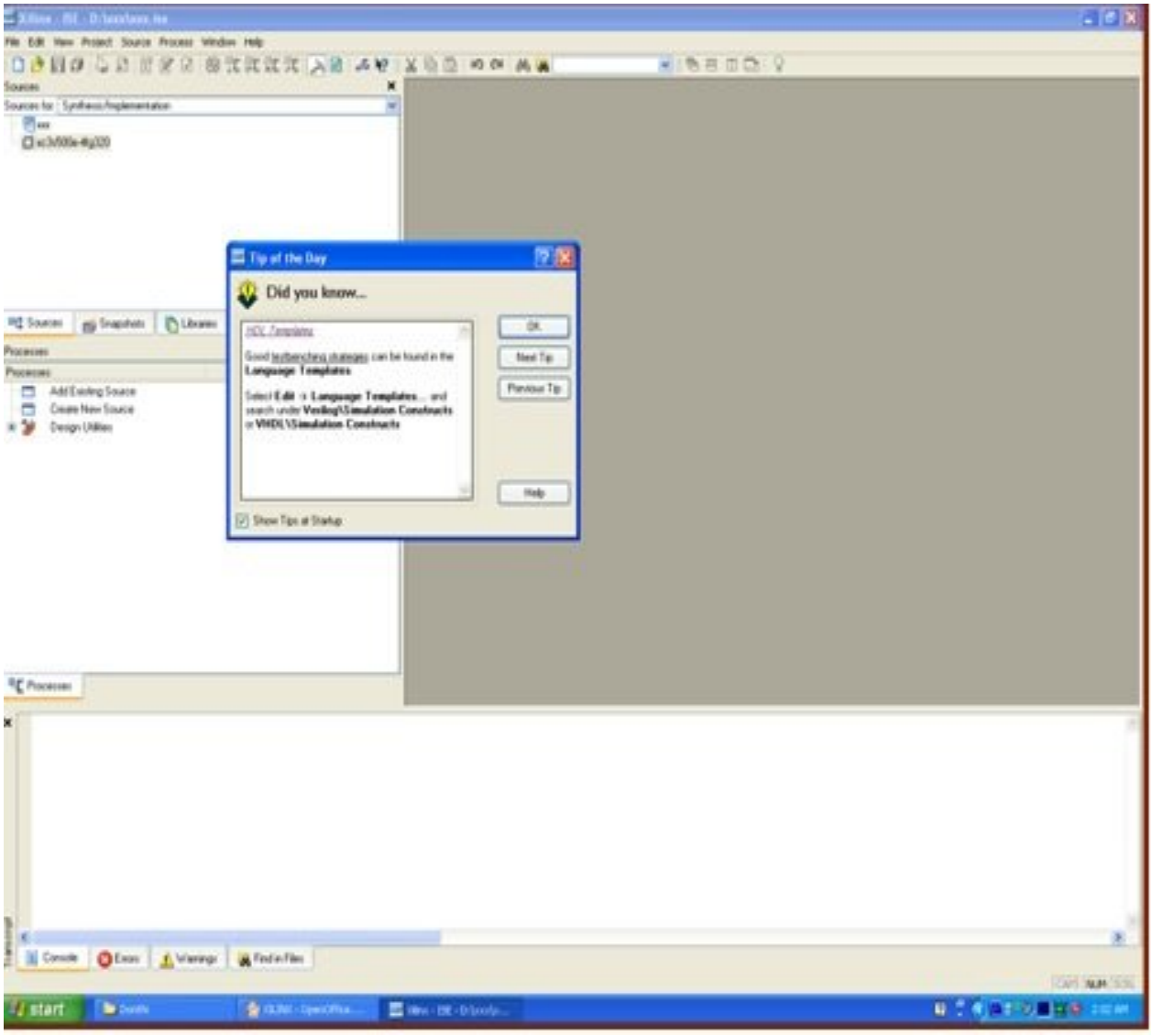
Modelsim is a widely-used logic simulation tool for verification and debugging of digital

circuits Modelsim is an easy-to-use yet versatile VHDL/(System) Verilog/SystemC simulator by Mentor Graphics. It supports behavioral, register transfer level, and gate-level modeling

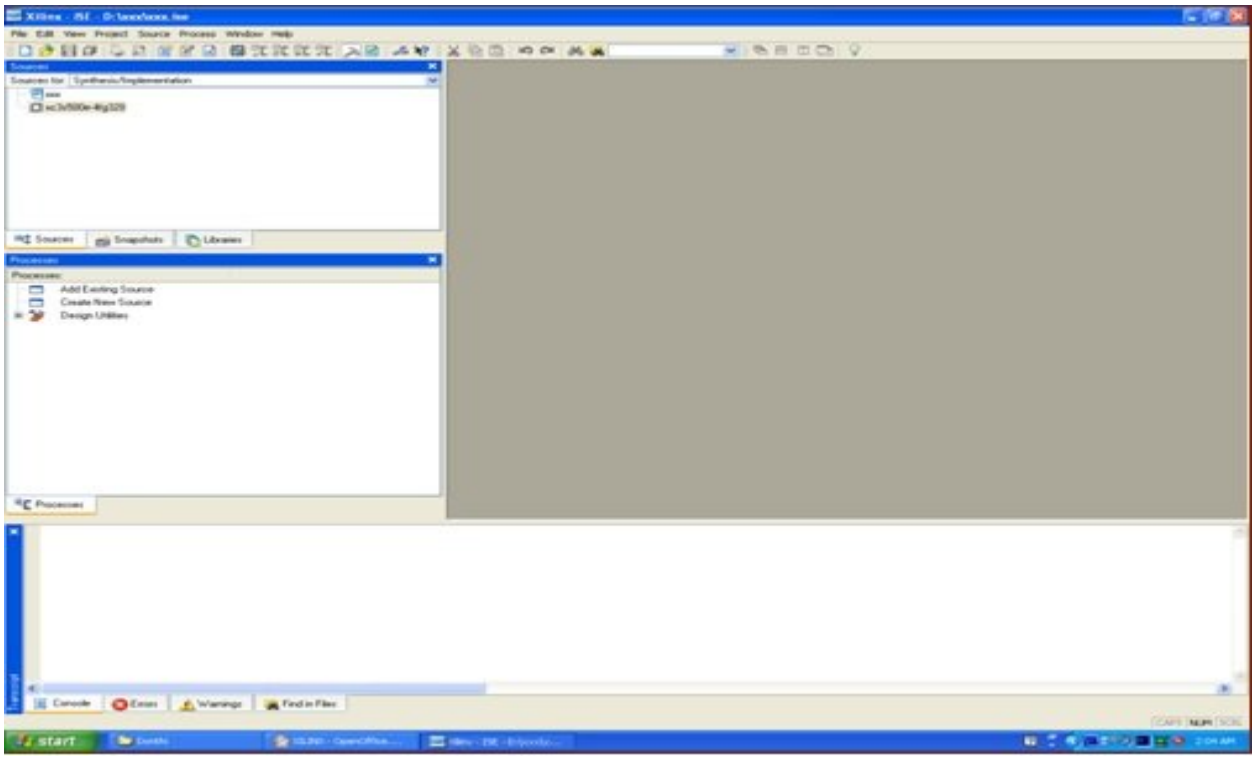
4.3 PROCEDURE

(a) Xilinx

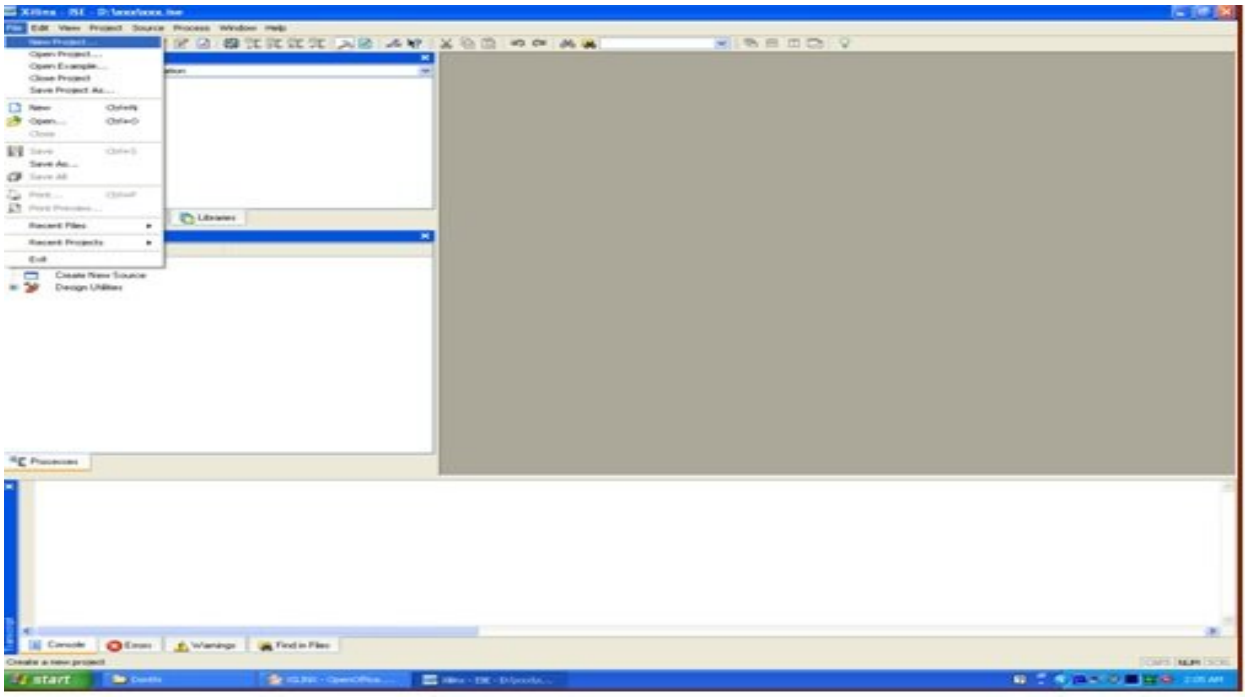
1. open Xilinx
2. a new window which shows Xilinx is opened



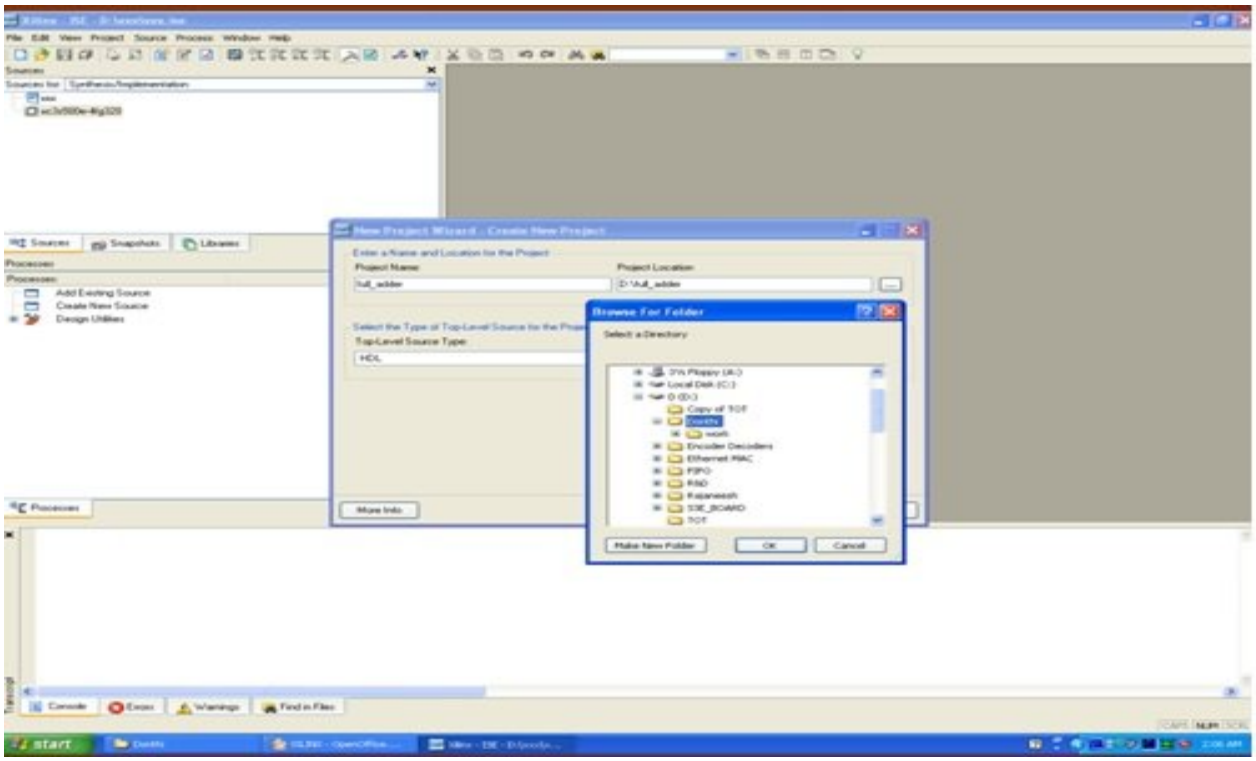
3. Press OK
4. New window will appear after selecting OK



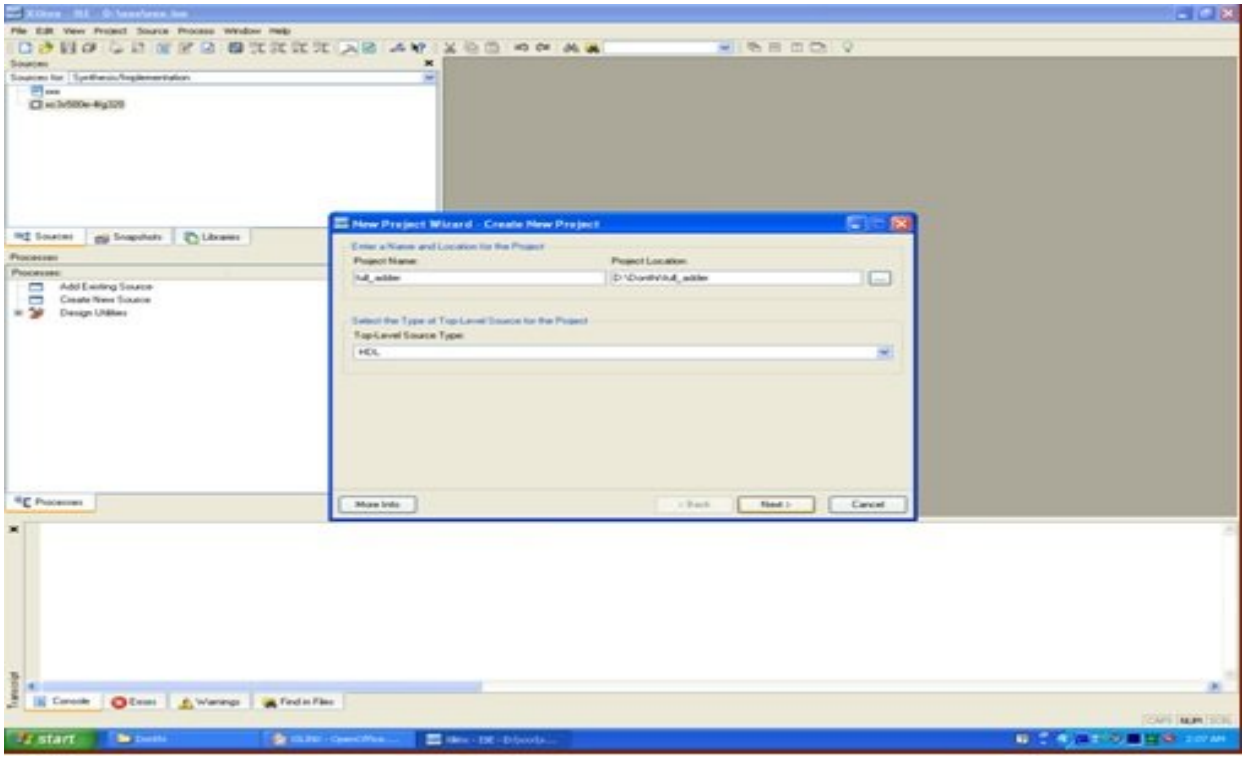
5. Go to File menu, select New Project



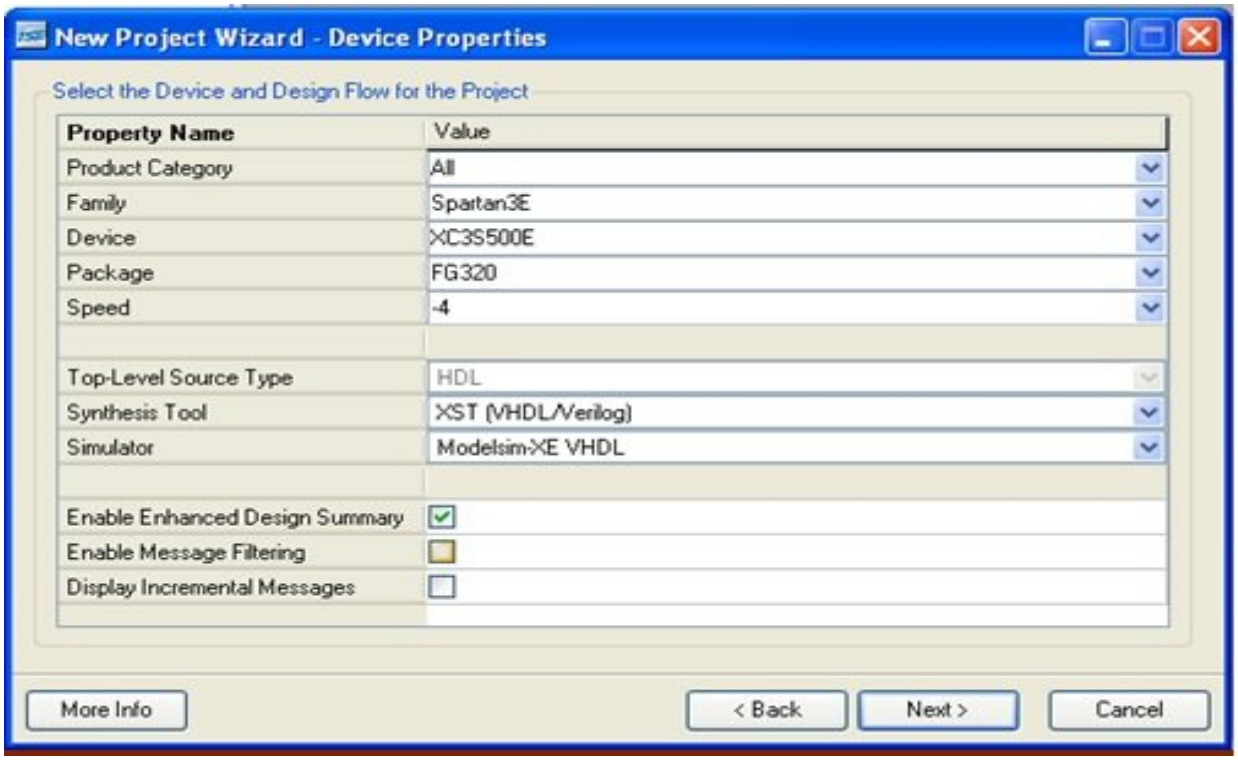
6. Select the destination



7. After Selecting destination select Next

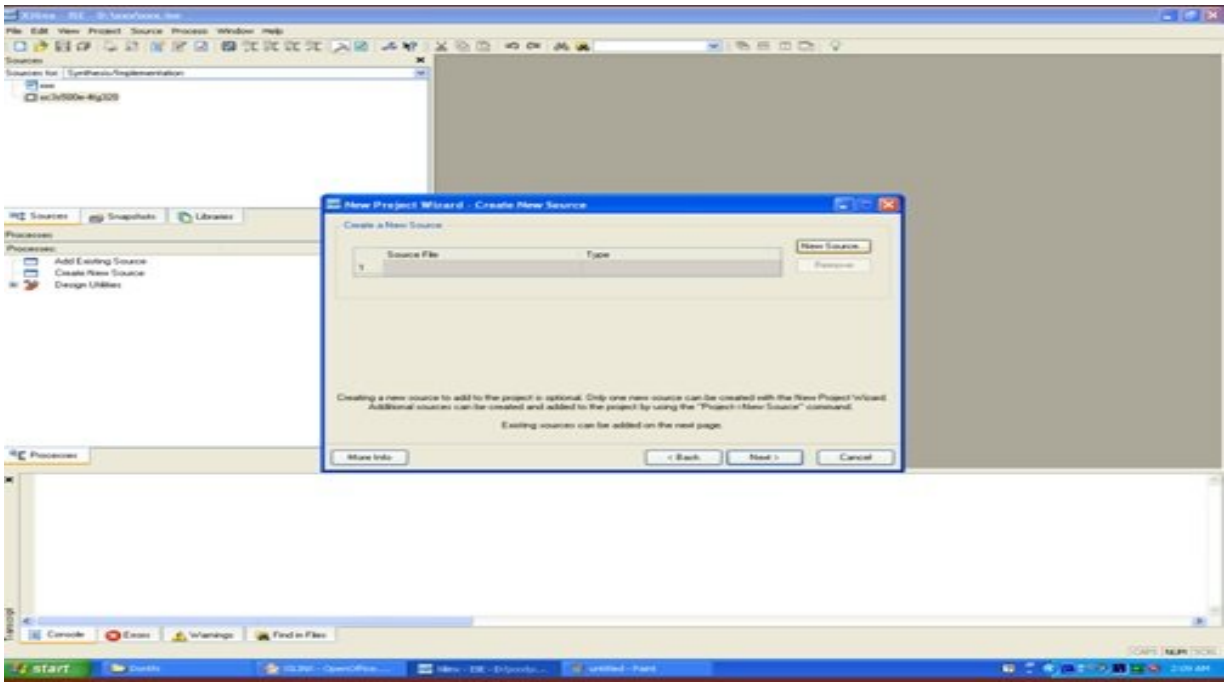


8. A new window which shows Device properties will appear

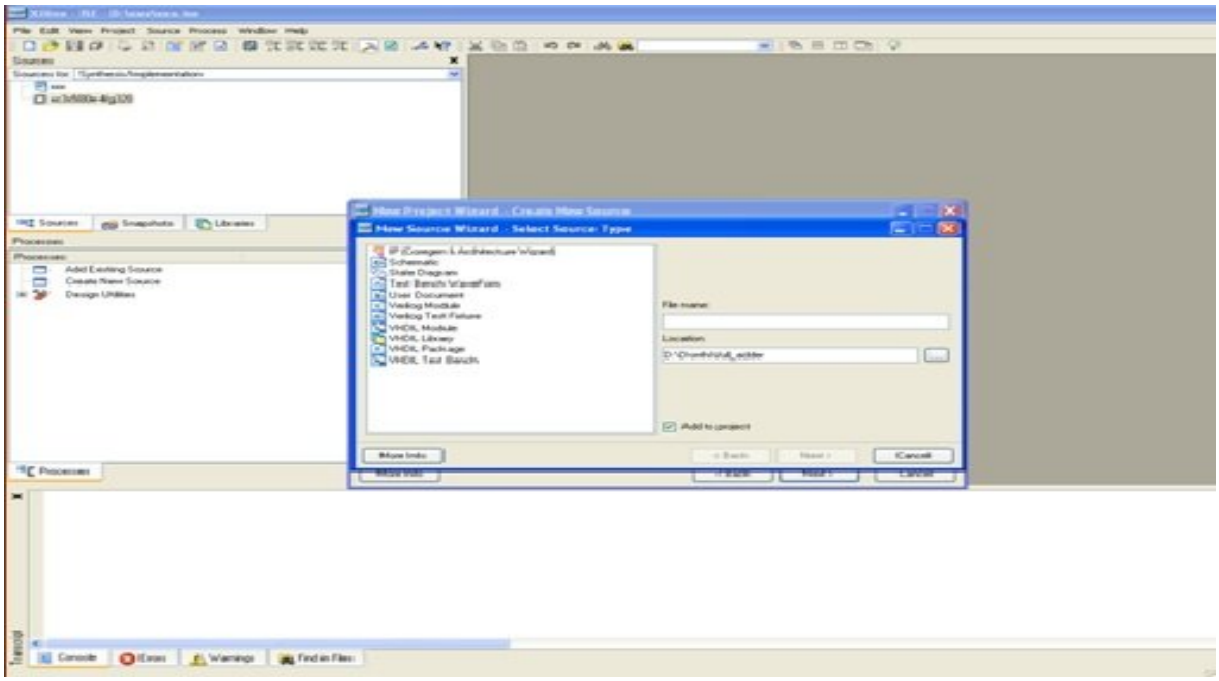


9. Select the Device properties in device properties window and select Next.

10. After device properties window, New project window will appear.

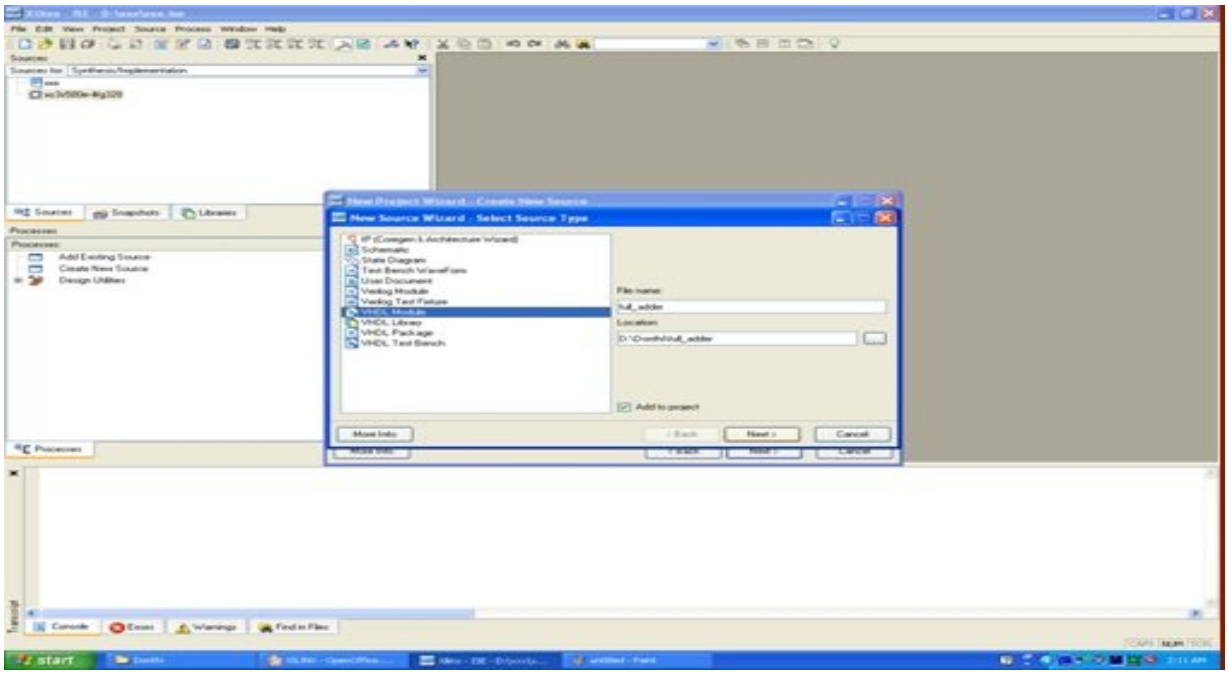


11. Now add source to the project by selecting "New Source" option



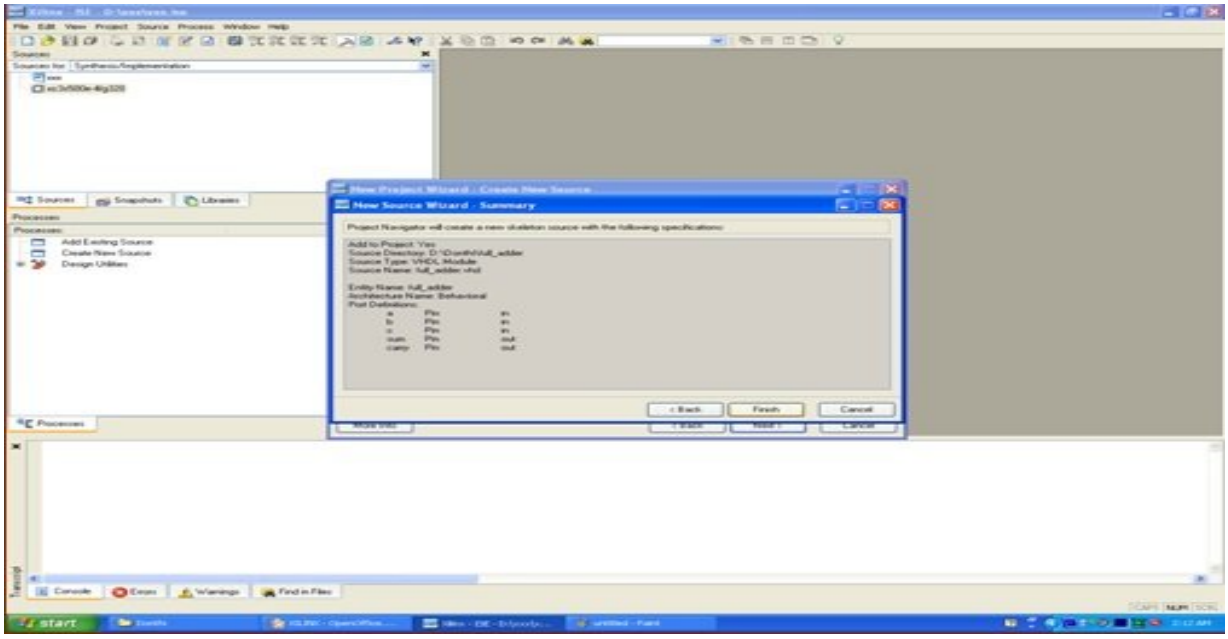
12. Select the verilog module option and enter the file name

13. Save the file name with .v extension.



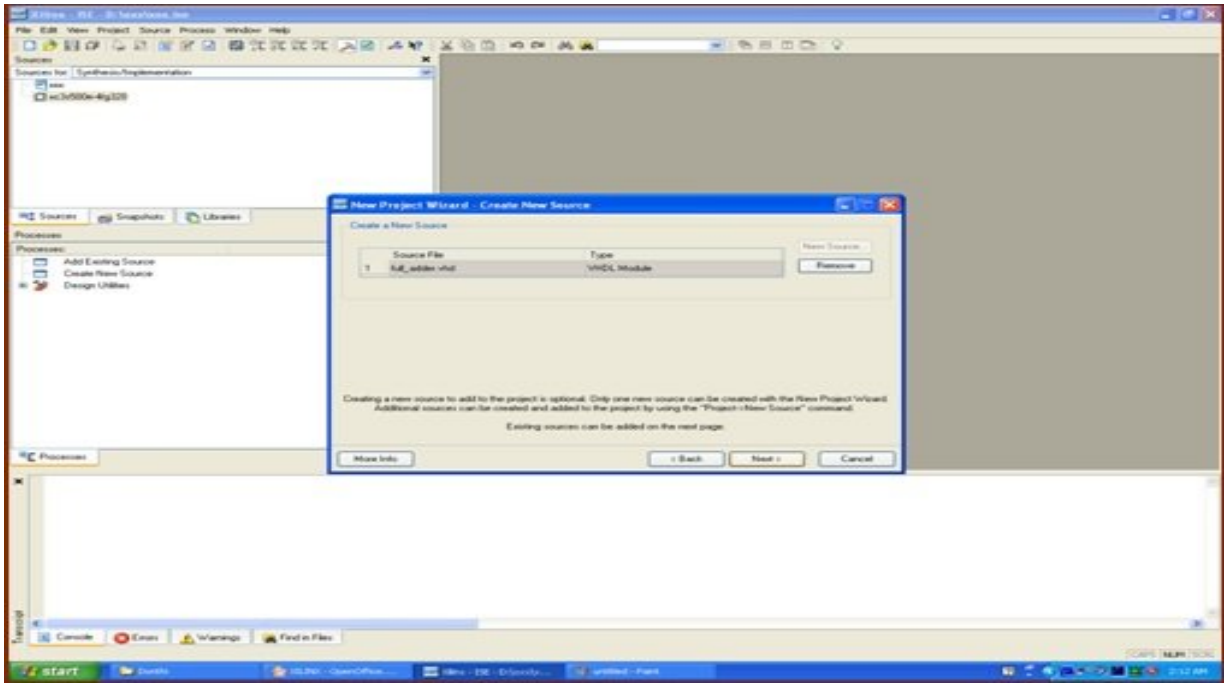
14. After entering file name select Next

15. A new window which shows New source wizard will appear.



16. Select Finish option

17. In the New Project Wizard source is added.



18. Select Next

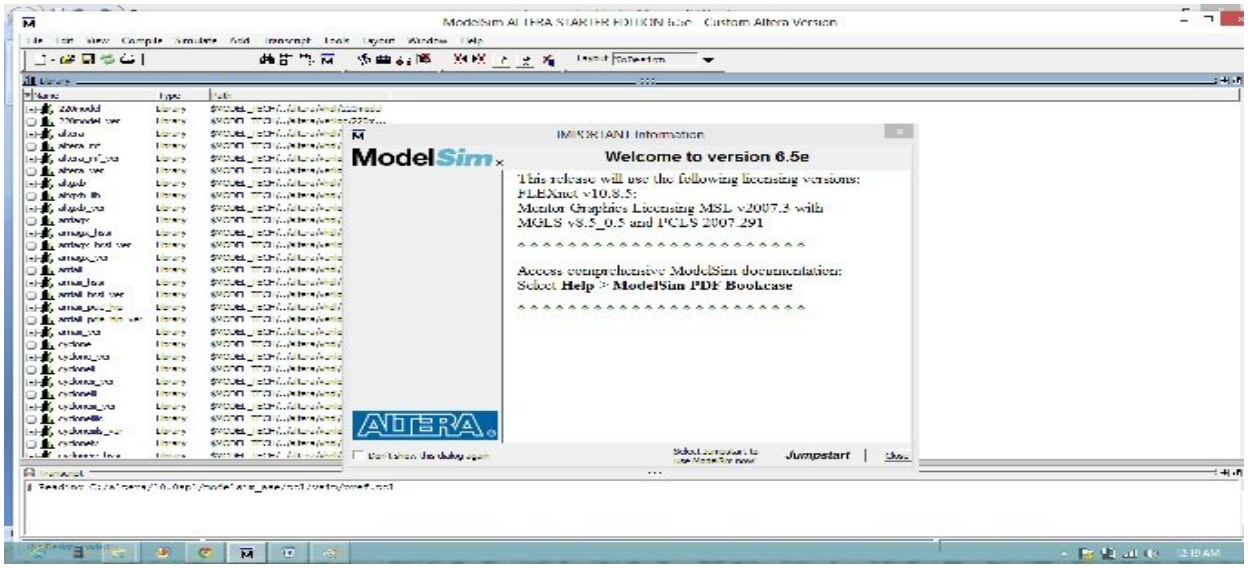
19. A new wizard will appear write the code in this wizard

20. Save the file, Compile, and run the file

(b) Modelsim

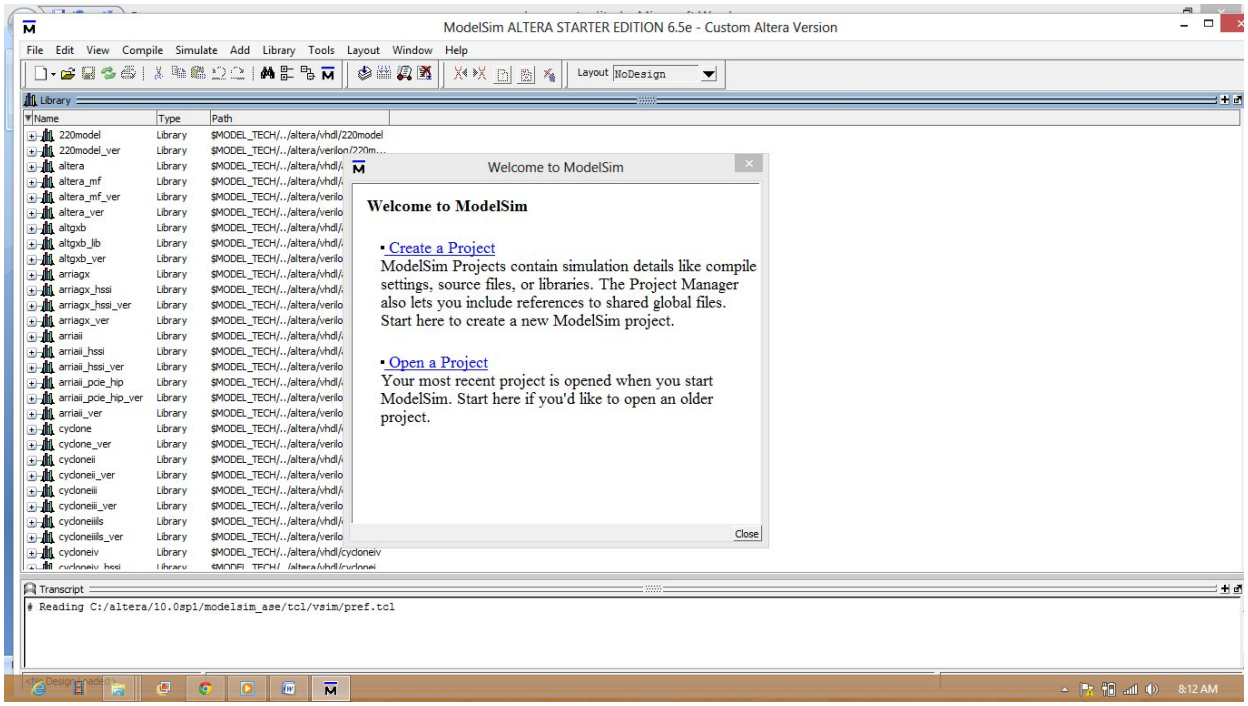
1. Start Modelsim

2. Modelsim window will appear



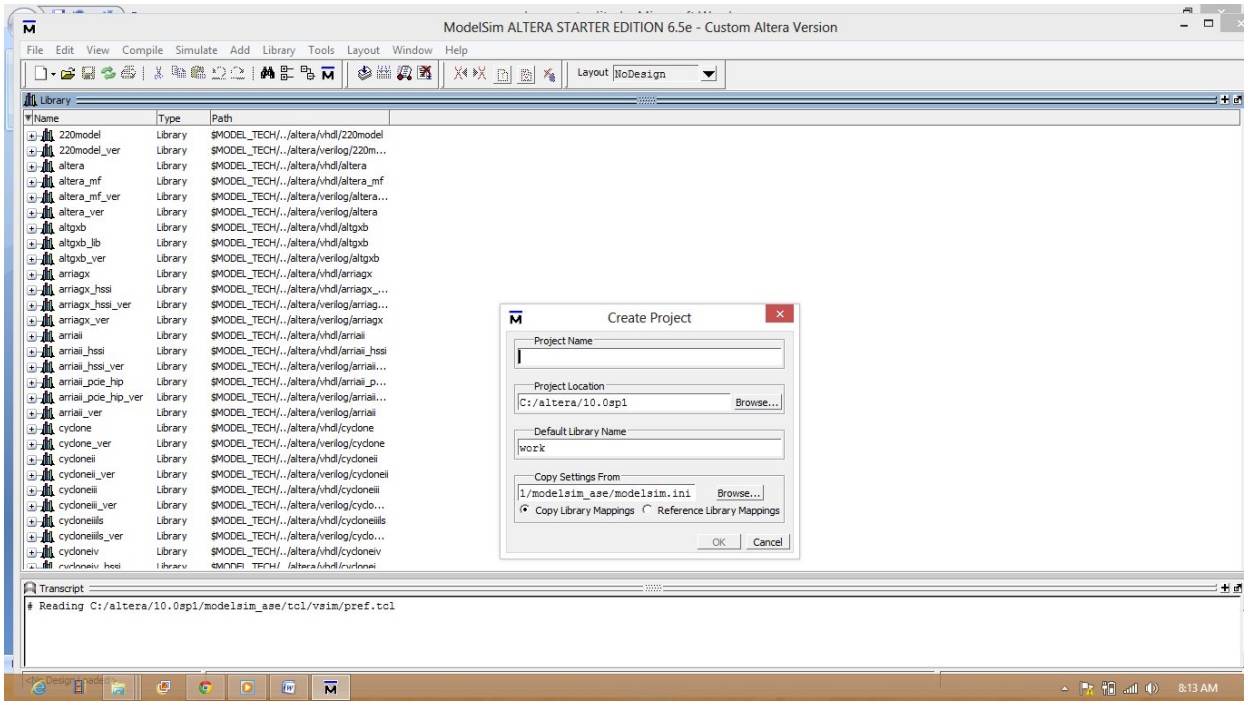
3. Click on jumpstart.

4. A new window which shows welcome to Modelsim will appear.



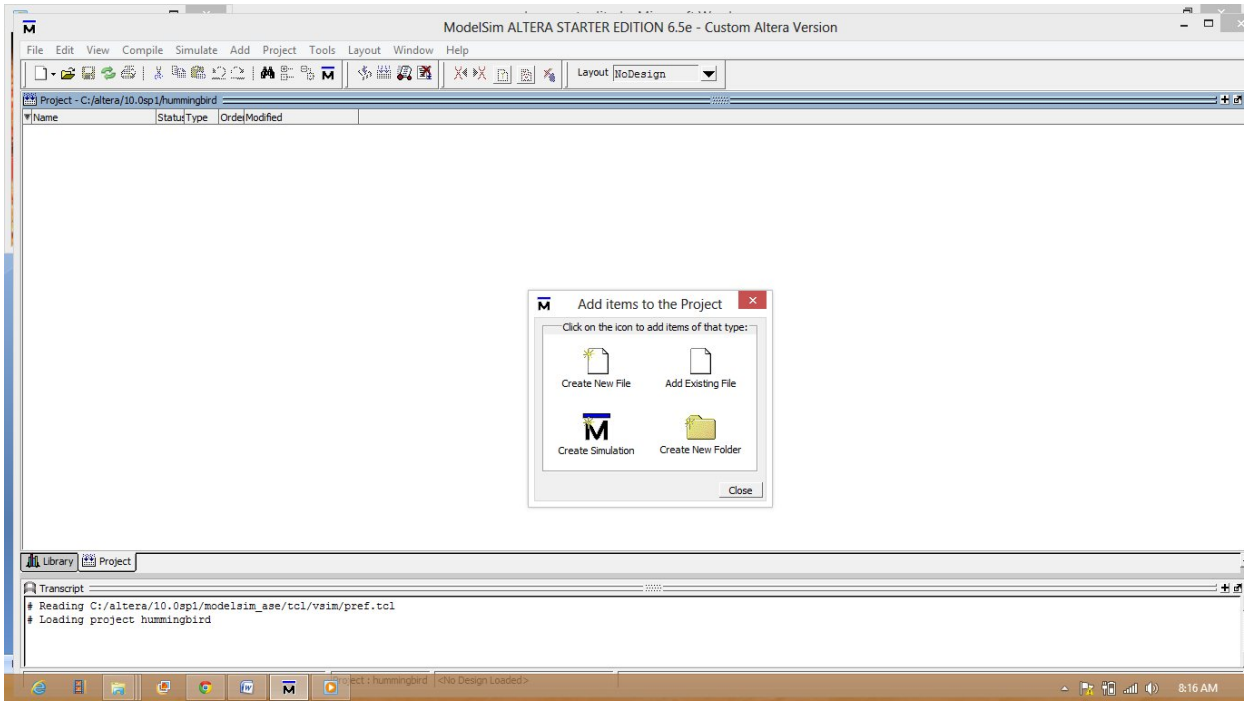
5. Click on create new project.

6. A new window which shows project creation will appear

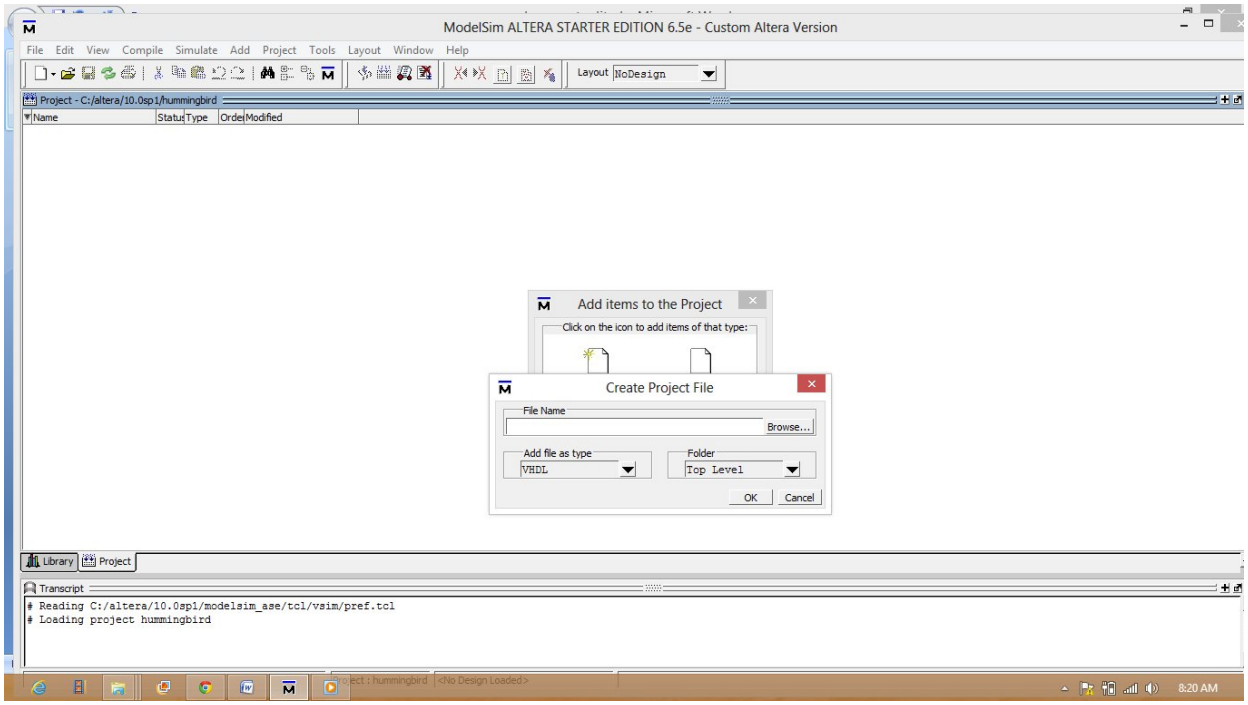


7. Type the project name. example, hummingbird. click OK.

8. A new window which shows "Add items to project", will appear.

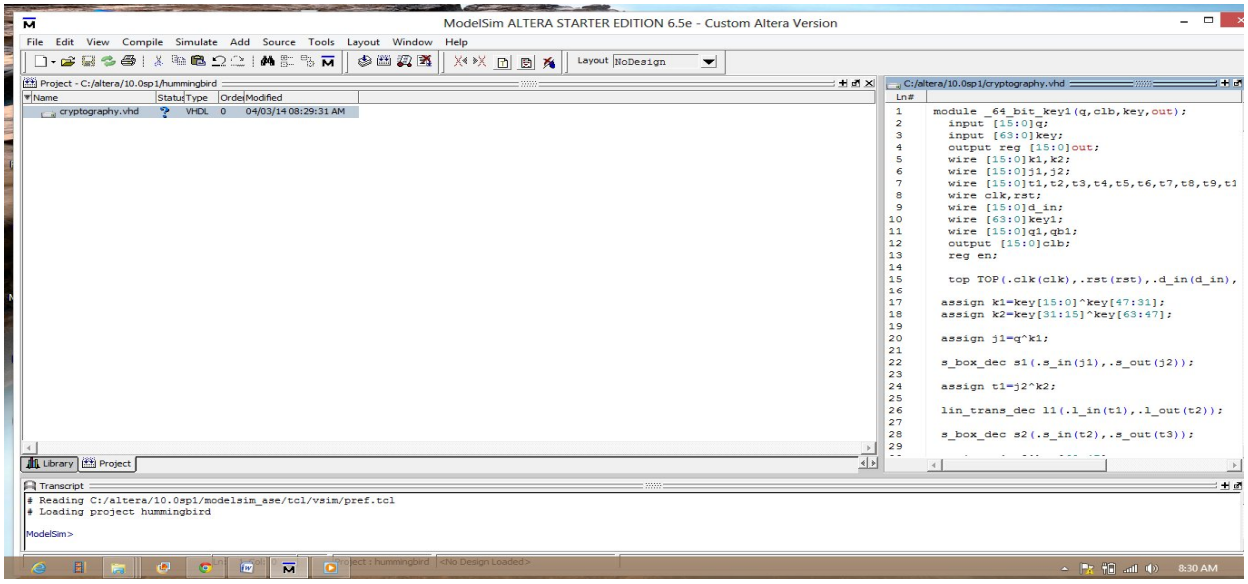


9. Click on "Create New File", a new window naamed"Create Project File" will appear.

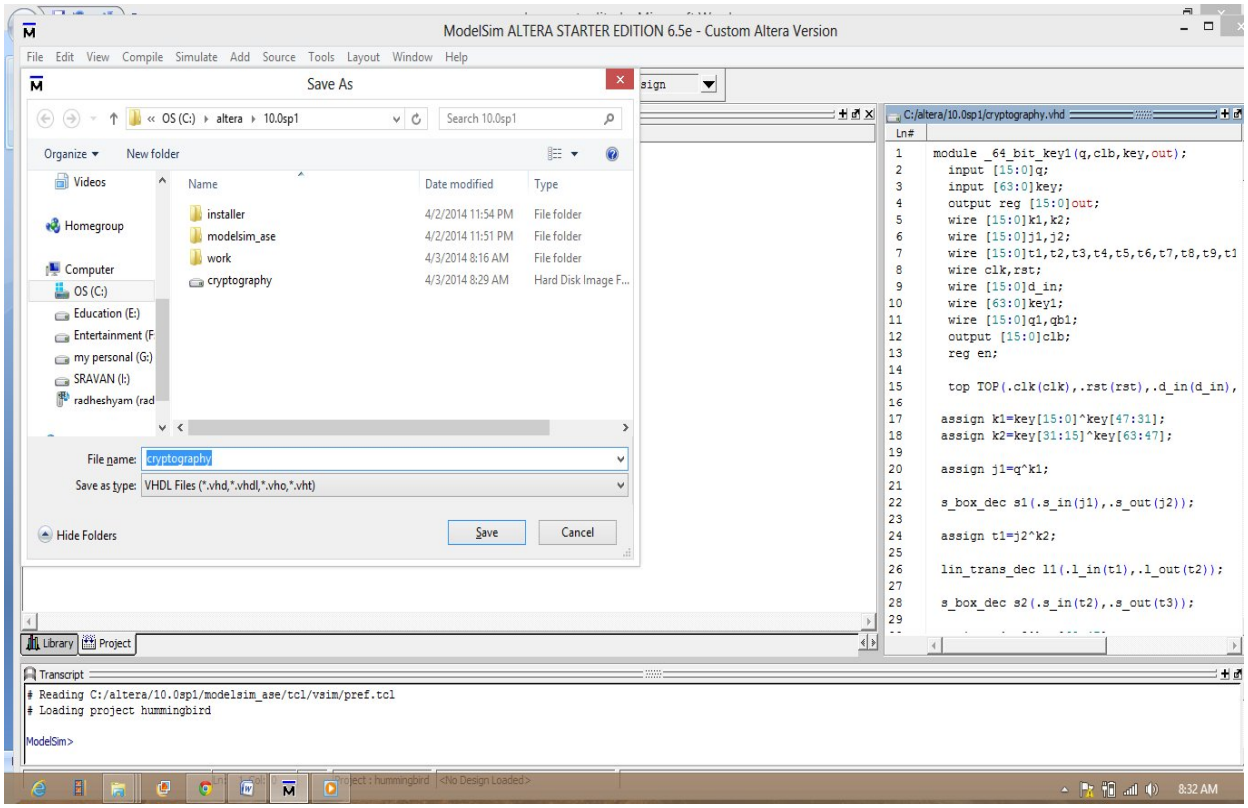


10. Enter the file name and press OK, this file is extended with ".vhd".

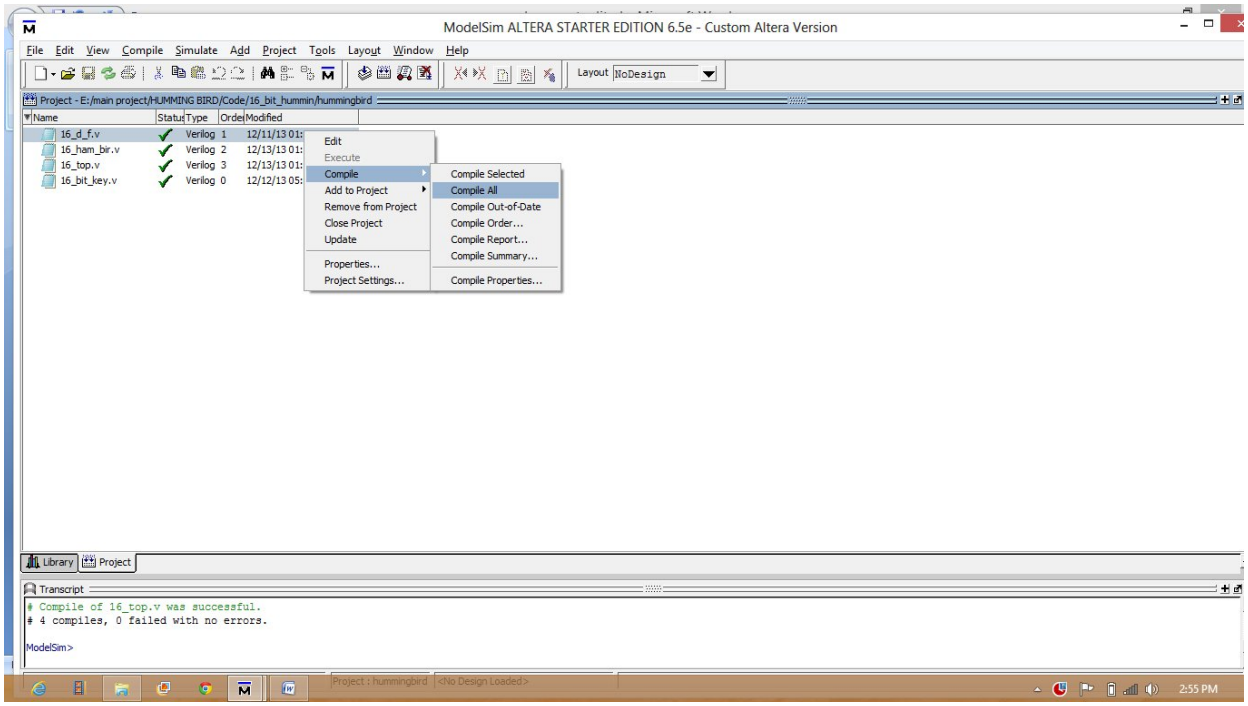
11. Double click on the created file a window will appear, code should be written in that window.



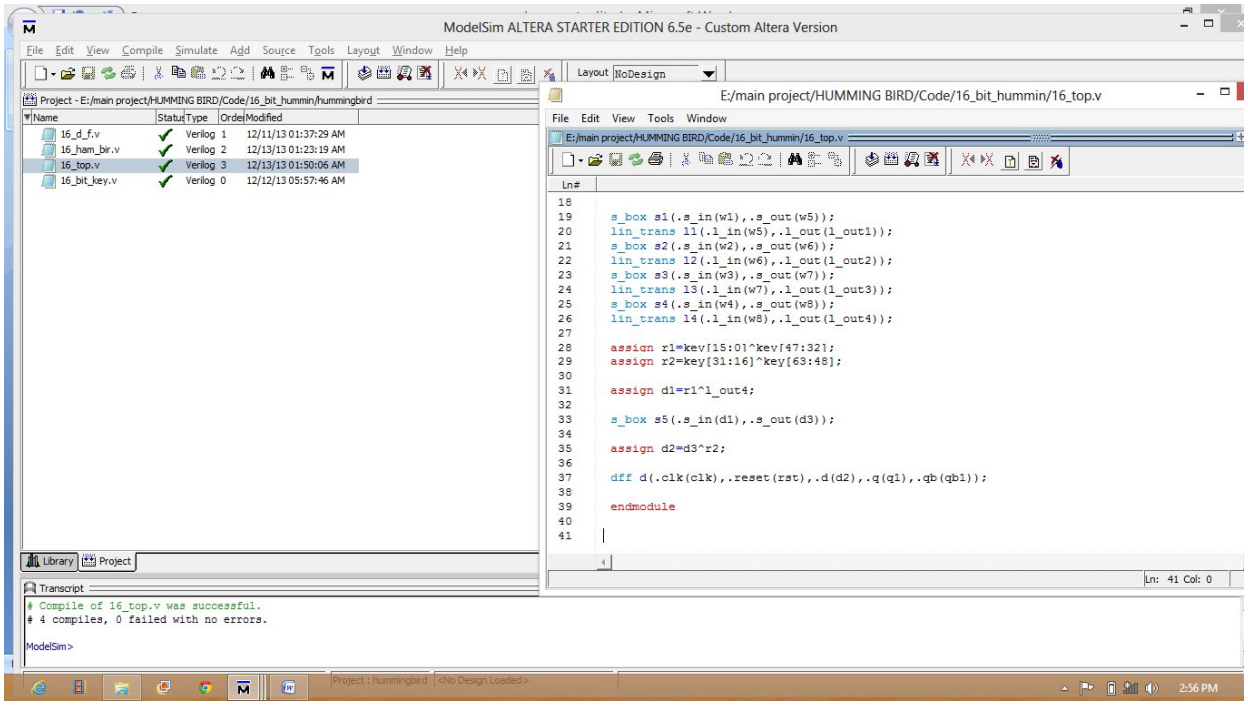
12. After writing code save the file go to file menu>save as>cryptography.vhd, click on save.



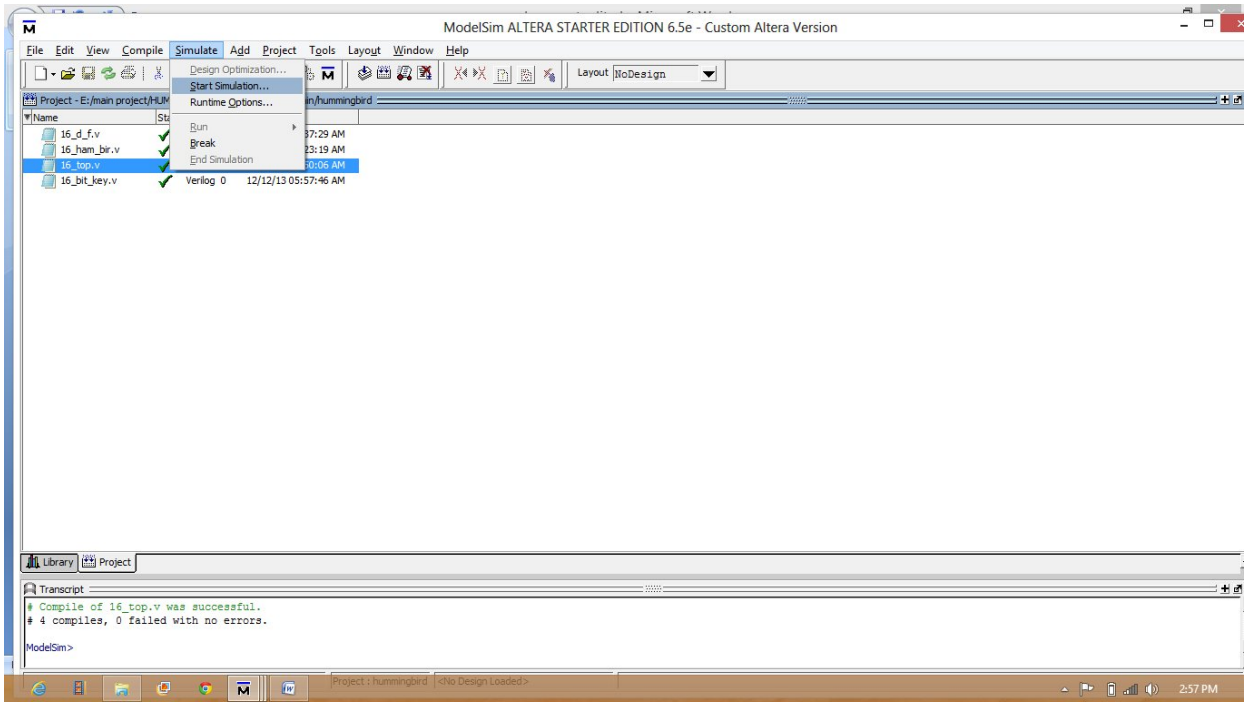
13. Select the file, Right click on file go to compile > compile all.



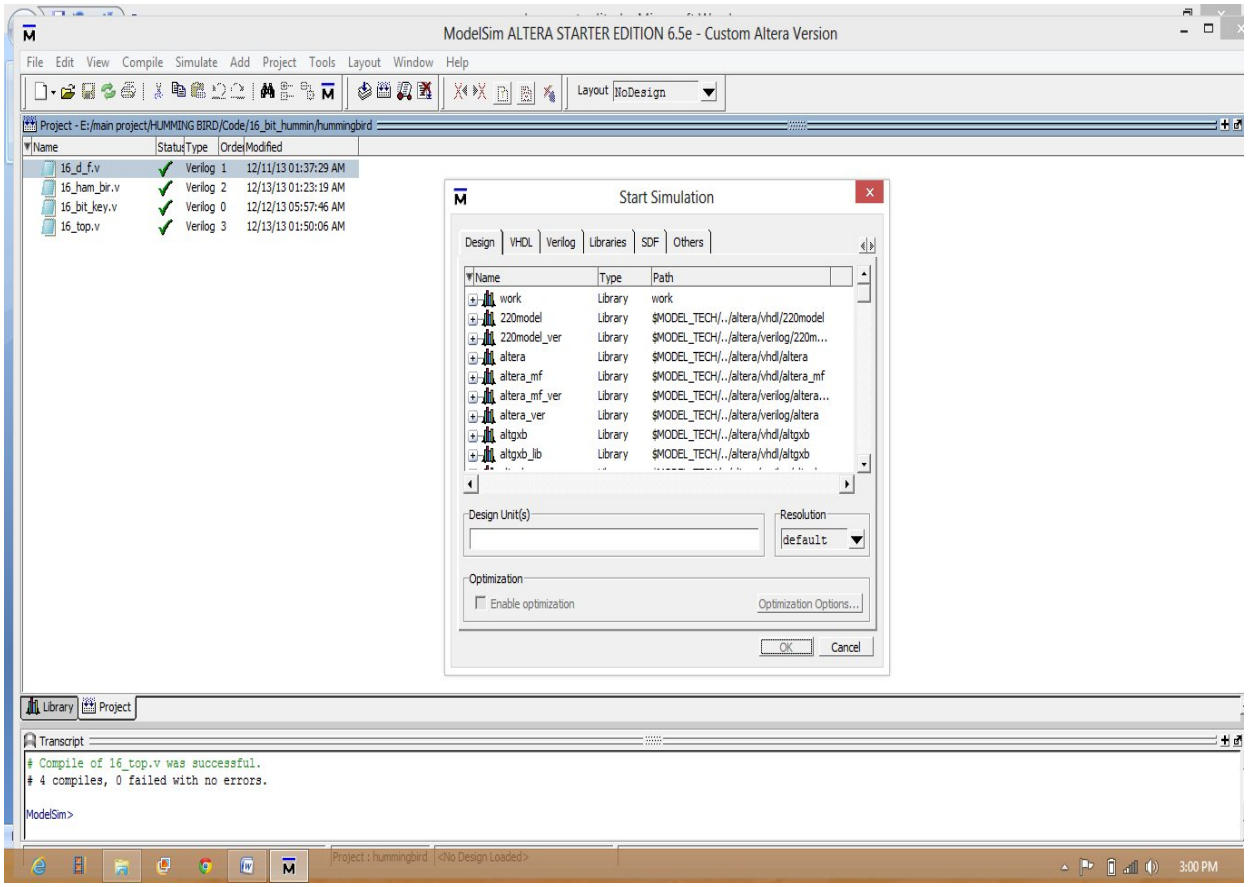
14. Check for errors



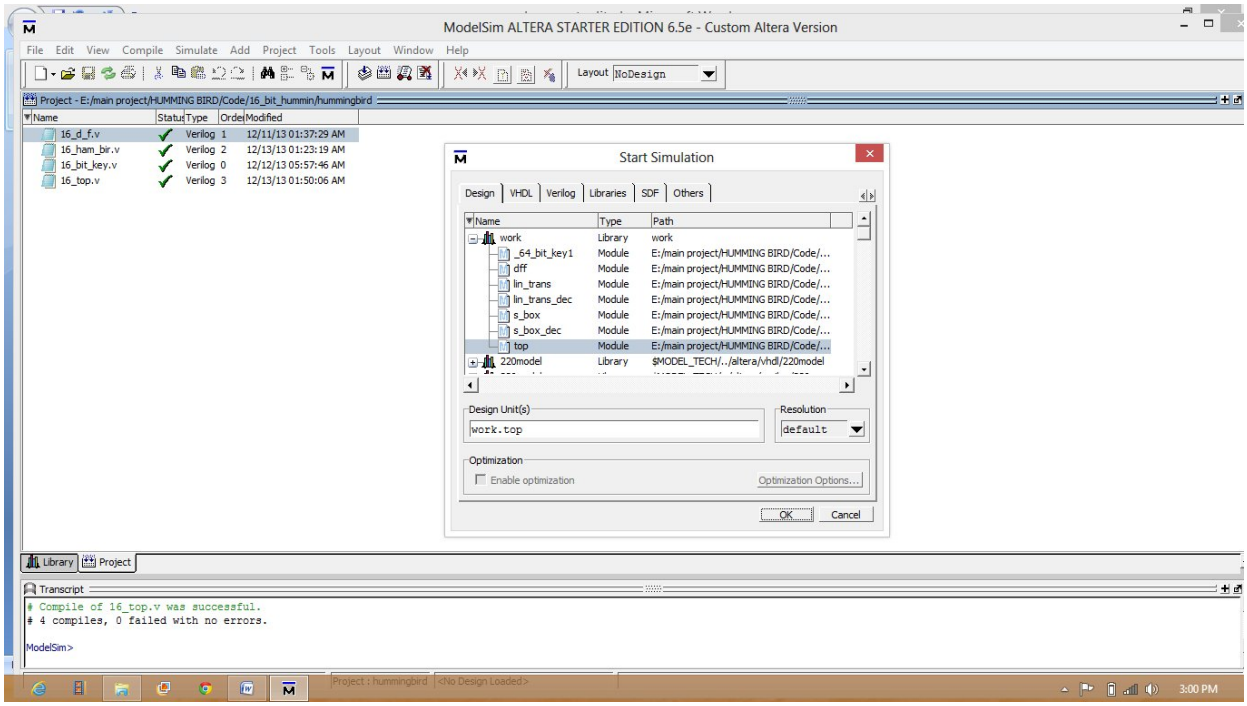
15. Go to Simulate > start simulation



16. start simulation window

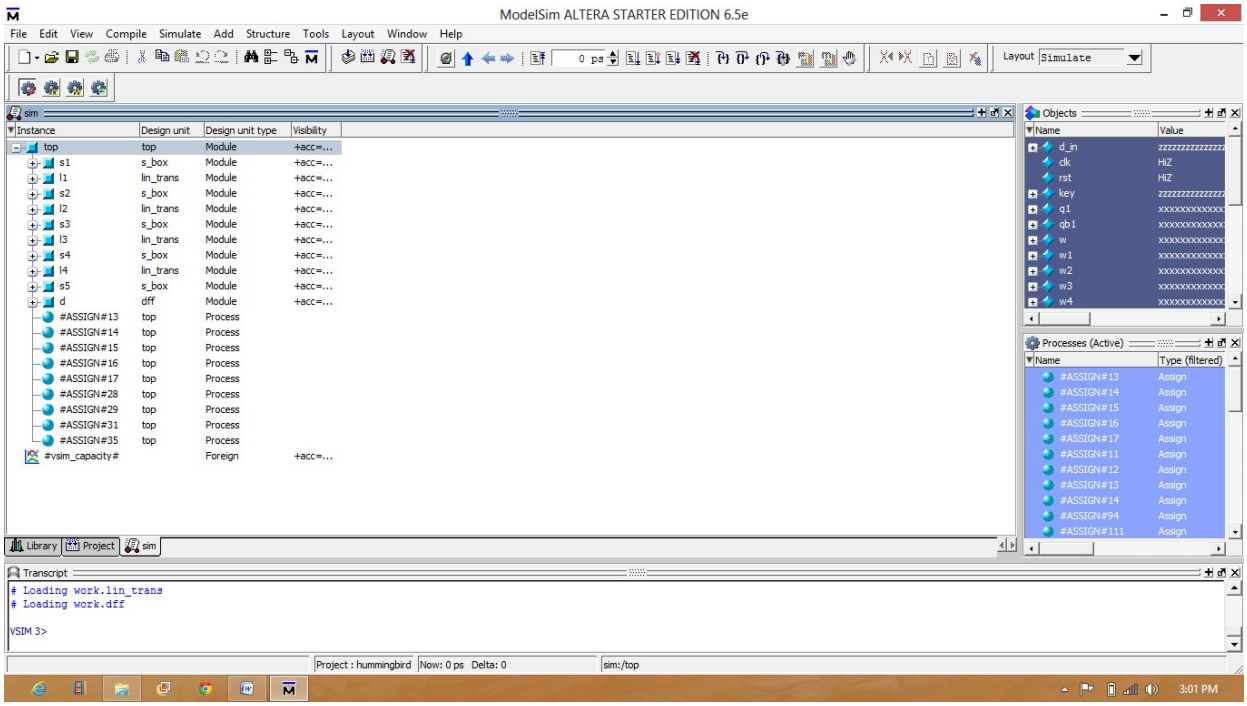


17. go to work select top



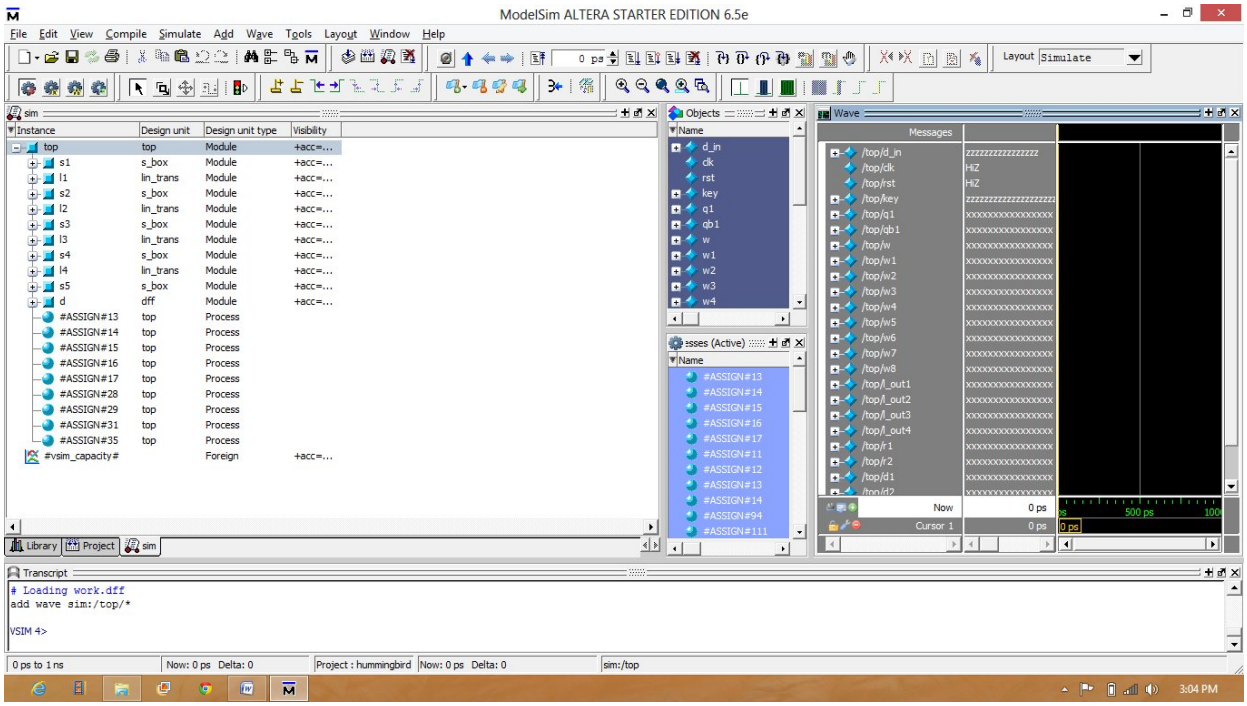
18. double clik on top and select ok

19. a new window instance window will appear

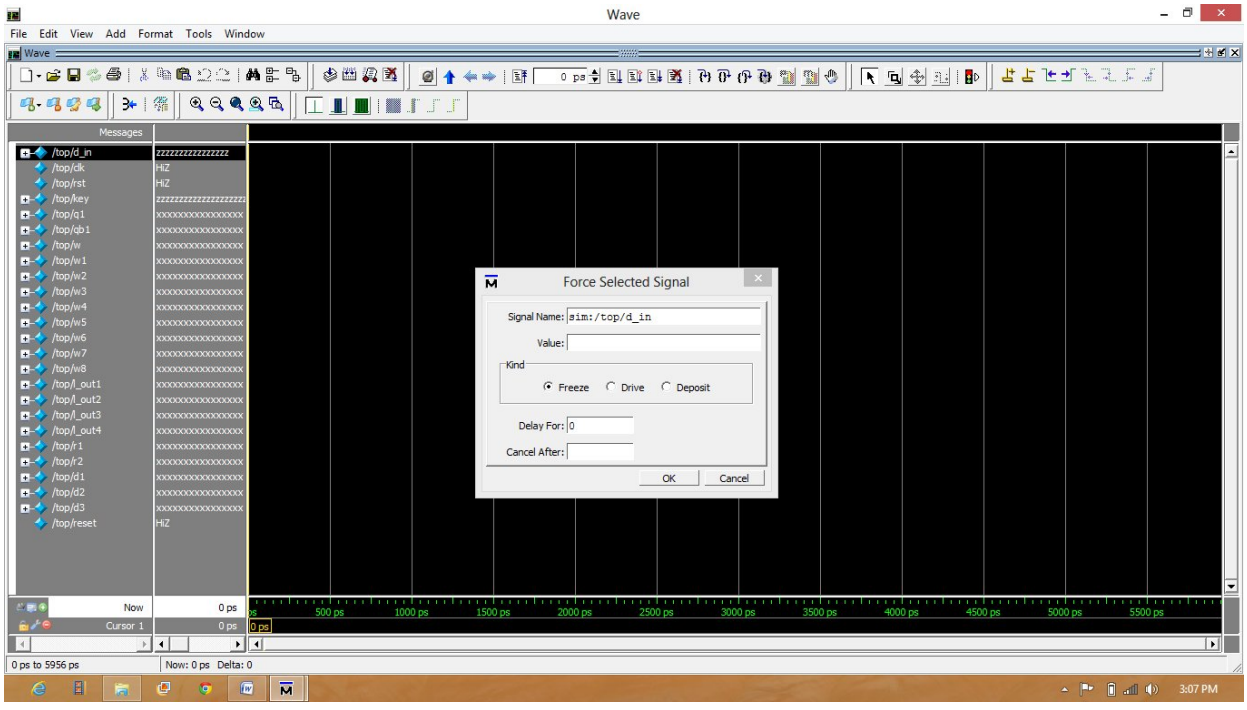


20. write click on top instance Add> to wave > all items in region.

21. after that wave window will appear.



22. Force the values for signals and check for desired output.



23 To continue the simulation (after 200ns) for another 100ns type: run 100ns (enter). The waveform output will now have 300ns of results. To start again from time 0, enter “restart”.

4.4 CODE

```
module barrel_org (S, A_P, B_P);
  input [3:0] S;
  input [15:0] A_P;
  output [15:0] B_P;
  reg [15:0] B_P;
always @ (A_P or S)
  begin
    case (S)
      4'b0000 : // Shift by 0
        begin
          B_P <= A_P;
        end
      4'b0001 : // Shift by 1
        begin
          B_P[15] <= A_P[0];
          B_P[14:0] <= A_P[15:1];
        end
      4'b0010 : // Shift by 2
          begin
            B_P[15:14] <= A_P[1:0];
            B_P[13:0] <= A_P[15:2];
          end
      4'b0011 : // Shift by 3
```

```
    begin
      B_P[15:13] <= A_P[2:0];
      B_P[12:0] <= A_P[15:3];
    end
4'b0100 : // Shift by 4
    begin
      B_P[15:12] <= A_P[3:0];
      B_P[11:0] <= A_P[15:4];
    end
4'b0101 : // Shift by 5
    begin
      B_P[15:11] <= A_P[4:0];
      B_P[10:0] <= A_P[15:5];
    end
4'b0110 : // Shift by 6
    begin
      B_P[15:10] <= A_P[5:0];
      B_P[9:0] <= A_P[15:6];
    end
4'b0111 : // Shift by 7
    begin
      B_P[15:9] <= A_P[6:0];
      B_P[8:0] <= A_P[15:7];
    end
```

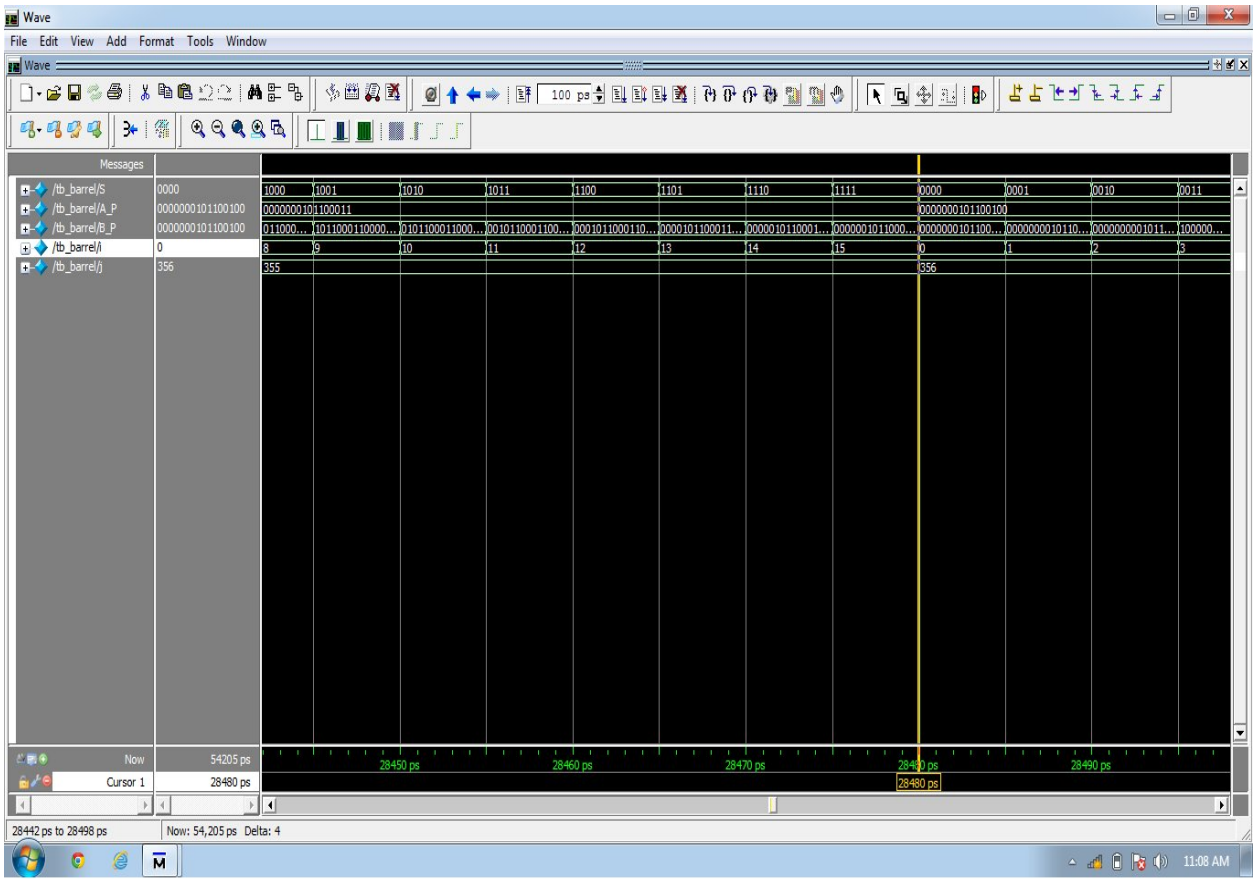
```
4'b1000 : // Shift by 8
  begin
    B_P[15:8] <= A_P[7:0];
    B_P[7:0] <= A_P[15:8];
  end
4'b1001 : // Shift by 9
  begin
    B_P[15:7] <= A_P[8:0];
    B_P[6:0] <= A_P[15:9];
  end
4'b1010 : // Shift by 10
  begin
    B_P[15:6] <= A_P[9:0];
    B_P[5:0] <= A_P[15:10];
end
4'b1011 : // Shift by 11
  begin
    B_P[15:5] <= A_P[10:0];
    B_P[4:0] <= A_P[15:11];
  end
4'b1100 : // Shift by 12
 begin
    B_P[15:4] <= A_P[11:0];
    B_P[3:0] <= A_P[15:12];
```

```
      end
      4'b1101 : // Shift by 13
        begin
          B_P[15:3] <= A_P[12:0];
          B_P[2:0] <= A_P[15:13];
        end
      4'b1110 : // Shift by 14
        begin
          B_P[15:2] <= A_P[13:0];
          B_P[1:0] <= A_P[15:14];
        end
      4'b1111 : // Shift by 15
        begin
          B_P[15:1] <= A_P[14:0];
          B_P[0] <= A_P[15];
        end
      default :B_P <= A_P;
    endcase
  end
endmodule
module tb_barrel ();
 reg [3:0] S;
  reg [15:0] A_P;
  wire [15:0] B_P;
```

```
barrel_org u1(S, A_P, B_P);
 integer i,j;
 /*initial
   begin
     A_P = 16'b1010011110001110;
   end*/
 initial
   begin
     for(j=0;j<65536;j=j+1)
       begin
         for(i=0;i<16;i=i+1)
           begin
             A_P = j;
             S=i;
              #5;
           end
       end
     end
 initial
 $monitor ("S=%b,A_P=%b,B_P=%b",S, A_P, B_P);
 initial
 #50000000 $finish;
Endmodule
```

4.5 RESULT

We have designed barrel shifter that shifts data. This was modeled in modelsim and the results were just as expected.



CHAPTER 5

CONCLUSION & FUTURE SCOPE

5.1 CONCLUSION

This paper has examined four barrel shifter designs mux-based data-reversal, mask-based data-reversal, eight barrel shifter designs, sixteen barrel shifter designs. Area and delay estimates, based on synthesis of structural level VHDL, indicate that data-reversal barrel shifters have less area than two's complement or one's complement barrel shifters and that mask-based data-reversal barrel shifters have less delay than the other designs. As the operand size increases, the delay of the shifters increases as O(log(n)) and their area increases as O(n log(n)).

5.2 FUTURE SCOPE

The design has been done for the 16 bit barrel shifter. The core can be used to design for further designs of 32 bit, 64 bit and so on to be utilized in DSP processors and any communication systems like USB transmitters etc.

REFERENCES

[1]. M. Seckora, “Barrel Shifter”, U.S. Patent 5, November 1995, pp (465,222).

[2]. A. Yamaguchi, “Bidirectional Shifter”, U.S. Patent 5, November 1993, pp (262,971).

[3]. T. Thomson and H. Tam, “Barrel Shifter”, U.S. Patent 5, July 1997, pp (652,718).

[4]. H. S. Lau and L. T. Ly, “Left Shift Over Detection”, U.S. Patent 5, July 1998, pp (777,906).

[5]. K. Dang and D. Anderson, “High Speed Barrel Shifter”, U.S. Patent 5, May 1995, pp(416,731).

Websites

- http://fisher.osu.edu/~muhanna.1/pdf/crypto.pdf
- http://wineyard.in/wp-content/uploads/2012/01/vlsi2.pdf
- http://journal.rtmonline.in/vol20iss8/05269.pdf
- http://comsec.uwaterloo.ca/download/HB_FPGA_Conference.pdf
- http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=%2Frzajc%2Frzajcconcepts.html
- http://all.net/edu/curr/ip/Chap2-4.html
- http://en.wikipedia.org/wiki/Very-large-scale_integration

APPENDIX-A

VLSI INTRODUCTION

HISTORICAL PERSPECTIVE

VLSI stands for Very Large Scale Integrated Circuits. The electronics industry has achieved a phenomenal growth over the last two decades, mainly due to the rapid advances in integration technologies, large-scale systems design - in short, due to the advent of VLSI. The number of applications of integrated circuits in high-performance computing, telecommunications, and consumer electronics has been rising steadily, and at a very fast pace. The current leading-edge technologies (such as low bit-rate video and cellular communications) already provide the end-users a certain amount of processing power and portability.

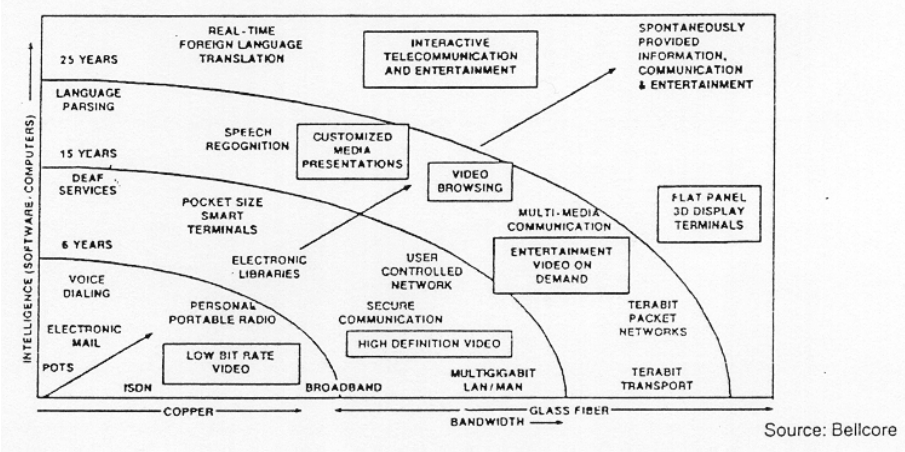


Fig 1.1 Prominent trends in information technologies over the next few decades.

This trend is expected to continue, with very important implications on VLSI and systems design. One of the most important characteristics of information services is their increasing need for very high processing power and bandwidth. The other important characteristic is that the information services tend to become more and more personalized (as opposed to collective services such as broadcasting), which means that the devices must be more intelligent to answer individual demands, and at the same time they must be portable to allow more flexibility/mobility.

As more and more complex functions are required in various data processing and telecommunications devices, the need to integrate these functions in a small system/package is also increasing. The level of integration as measured by the number of logic gates in a monolithic chip has been steadily rising for almost three decades, mainly due to the rapid progress in processing technology and interconnects technology. Table 1.1 shows the evolution of logic complexity in integrated circuits over the last three decades, and marks the milestones of each era. State-of-the-art examples of ULSI chips, such as the DEC Alpha or the INTEL Pentium contain 3 to 6 million transistors.

ERA	DATE	COMPLEXITY
(number of logic blocks per chip)		
Single transistor	1959	less than 1
Unit logic (one gate)	1960	1
Multi-function	1962	2 – 4
Complex function	1964	5 – 20
Medium Scale Integration	1967	20 - 200 (MSI)
Large Scale Integration	1972	200 - 2000 (LSI)
Very Large Scale Integration	1978	2000 - 20000(VLSI)
Ultra Large Scale Integration	1989	20000 - ?(ULSI)

Table-1.1: Evolution of logic complexity in integrated circuits

The most important message here is that the logic complexity per chip has been (and still is) increasing exponentially. The monolithic integration of a large number of functions on a single chip usually provides:

- Less area/volume and therefore, compactness
- Less power consumption
- Less testing requirements at system level

- Higher reliability, mainly due to improved on-chip interconnects
- Higher speed, due to significantly reduced interconnection length
- Significant cost savings

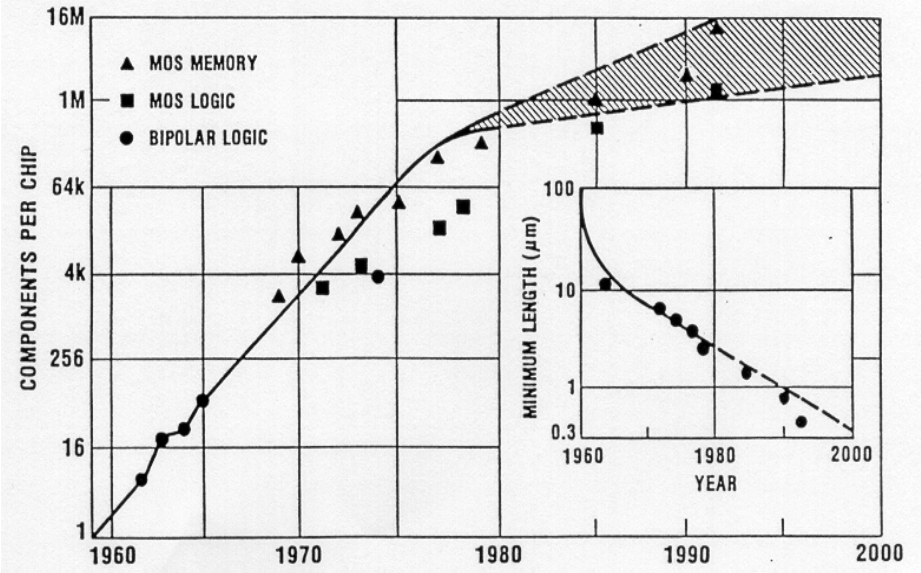


Fig 1.2**:** Evolution of integration density and minimum feature size.

Therefore, the current trend of integration will also continue in the foreseeable future. Advances in device manufacturing technology, and especially the steady reduction of minimum feature size (minimum length of a transistor or an interconnect realizable on chip) support this trend. Figure 1.2 shows the history and forecast of chip complexity - and minimum feature size - over time, as seen in the early 1980s. At that time, a minimum feature size of 0.3 microns was expected around the year 2000. A minimum size of 0.25 microns was readily achievable by the year 1995. As a direct result of this, the integration density has also exceeded previous expectations - the first 64 Mbit DRAM, and the INTEL Pentium microprocessor chip containing more than 3 million transistors were already available by 1994, pushing the envelope of integration density.

When comparing the integration density of integrated circuits, a clear distinction must be made between the memory chips and logic chips. It can be observed that in terms of transistor count, logic chips contain significantly fewer transistors in any given year mainly due to large

consumption of chip area for complex interconnects. Memory circuits are highly regular and thus more cells can be integrated with much less area for interconnects.

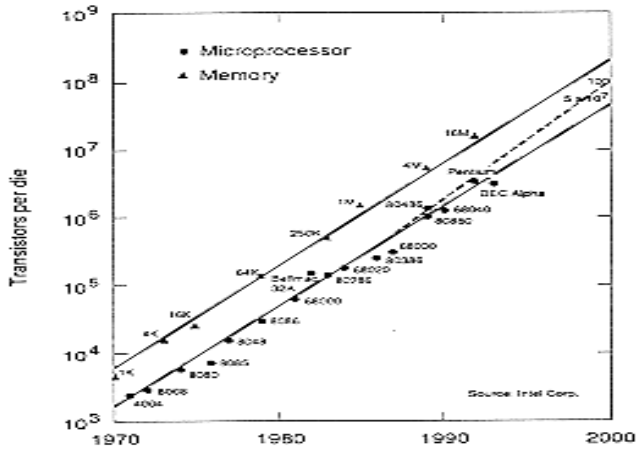


Fig 1.3: Level of integration over time, for memory chips and logic chips.

Generally speaking, logic chips such as microprocessor chips and digital signal processing (DSP) chips contain not only large arrays of memory (SRAM) cells, but also many different functional units. As a result, their design complexity is considered much higher than that of memory chips, although advanced memory chips contain some sophisticated logic functions. The design complexity of logic chips increases almost exponentially with the number of transistors to be integrated. As a result, the level of actual logic integration tends to fall short of the integration level achievable with the current processing technology. Sophisticated computer-aided design (CAD) tools and methodologies are developed and applied in order to manage the rapidly increasing design complexity.

VLSI DESIGN FLOW

The design process, at various levels, is usually evolutionary in nature. It starts with a given set of requirements. Initial design is developed and tested against the requirements. When requirements are not met, the design has to be improved. If such improvement is either not possible or too costly, then the revision of requirements and its impact analysis must be considered. The Y-chart (first introduced by D. Gajski) shown in Fig. 2.4 illustrates a design flow for most logic chips, using design activities on three different axes (domains) which resemble the letter Y.

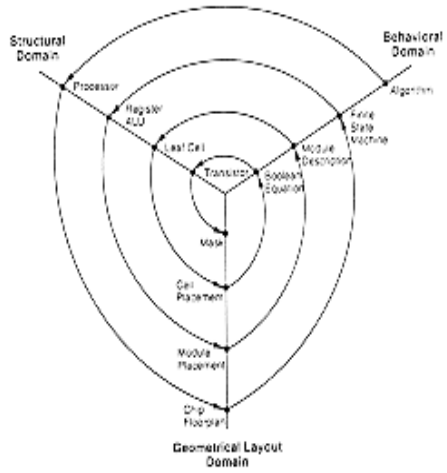


Fig 1.4: Y-chart representation

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

- Behavioral domain
- Structural domain
- Geometrical layout domain

The design flow starts from the algorithm that describes the behavior of the target chip. The corresponding architecture of the processor is first defined. It is mapped onto the chip surface by floor planning. The next design evolution in the behavioral domain defines finite state machines (FSMs) which are structurally implemented with functional modules such as registers and arithmetic logic units (ALUs).

These modules are then geometrically placed onto the chip surface using CAD tools for automatic module placement followed by routing, with a goal of minimizing the interconnects area and signal delays. The third evolution starts with a behavioral module description. Individual modules are then implemented with leaf cells. At this stage the chip is described in terms of logic gates (leaf cells), which can be placed and interconnected by using a cell placement & routing program. The last evolution involves a detailed Boolean description of leaf cells followed by a transistor level implementation of leaf cells and mask generation. In standard-cell based design, leaf cells are already pre-designed and stored in a library for logic design use.

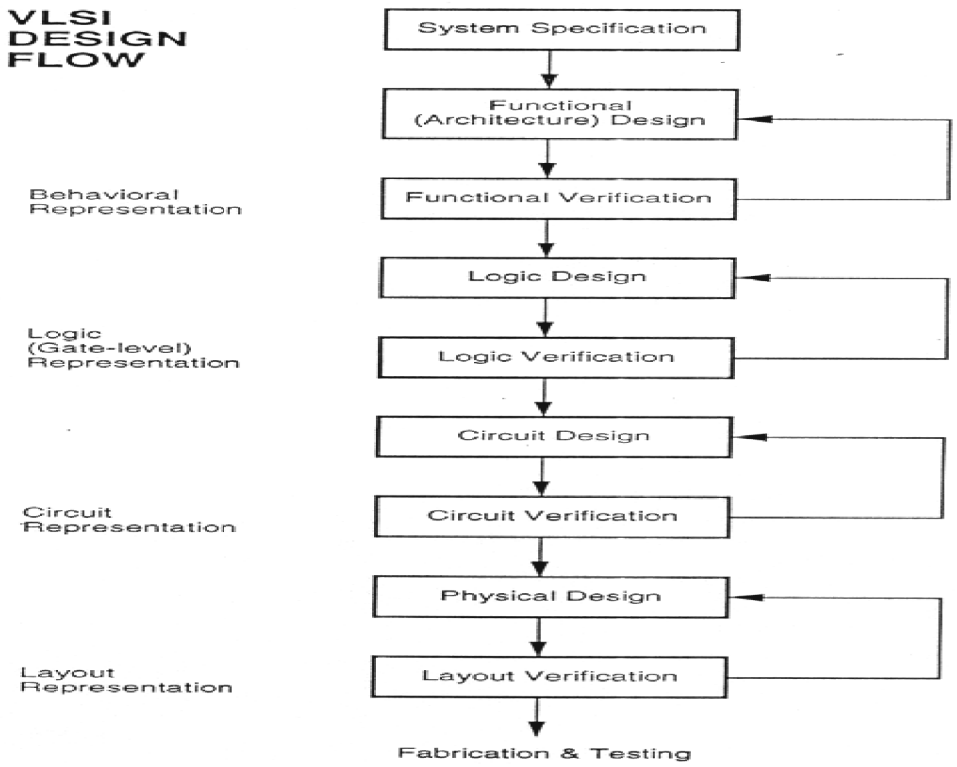


Fig 1.5: VLSI design flow

Note that the verification of design plays a very important role in every step during this process. The failure to properly verify a design in its early phases typically causes significant and expensive re-design at a later stage, which ultimately increases the time-to-market.

Although the design process has been described in linear fashion for simplicity, in reality there are many iterations back and forth, especially between any two neighboring steps, and occasionally even remotely separated pairs. Although top-down design flow provides an excellent design process control, in reality, there is no truly unidirectional top-down design flow. Both top-down and bottom-up approaches have to be combined. For instance, if a chip designer defined architecture without close estimation of the corresponding chip area, then it is very likely that the resulting chip layout exceeds the area limit of the available technology. In such a case, in order to fit the architecture into the allowable chip area, some functions may have to be removed and the design process must be repeated. Such changes may require significant modification of the original requirements. Thus, it is very important to feed forward low-level information to higher levels (bottom up) as early as possible. Some of the classical

techniques for reducing the complexity of IC design are: Hierarchy, regularity, modularity and locality.

DESIGN HIERARCHY

The use of hierarchy, or divide and conquer technique involves dividing a module into sub-modules and then repeating this operation on the sub-modules until the complexity of the smaller parts becomes manageable. This approach is very similar to the software case where large programs are split into smaller and smaller sections until simple subroutines, with well-defined functions and interfaces can be written. Correspondingly, a hierarchy structure can be described in each domain separately.

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

In the physical domain, partitioning a complex system into its various functional blocks will provide a valuable guidance for the actual realization of these blocks on chip. Obviously, the approximate shape and size (area) of each sub-module should be estimated in order to provide a useful floor plan. This physical view describes the external geometry of the adder, the locations of input and output pins, and how pin locations allow some signals (in this case the carry signals) to be transferred from one sub-block to the other without external routing. At lower levels of the physical hierarchy, the internal mask

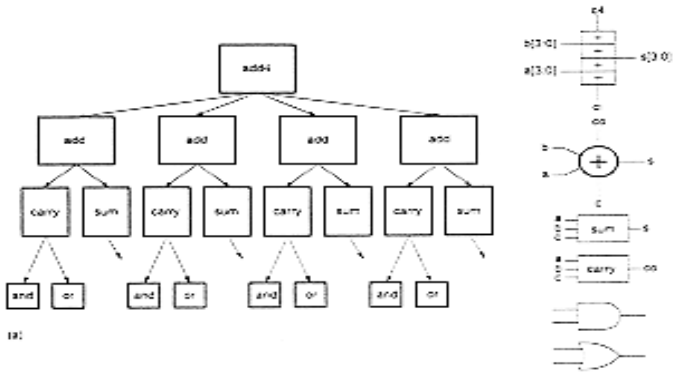


Fig 1.6: Structural decomposition of a four-bit adder circuit, showing the hierarchy down to gate level.

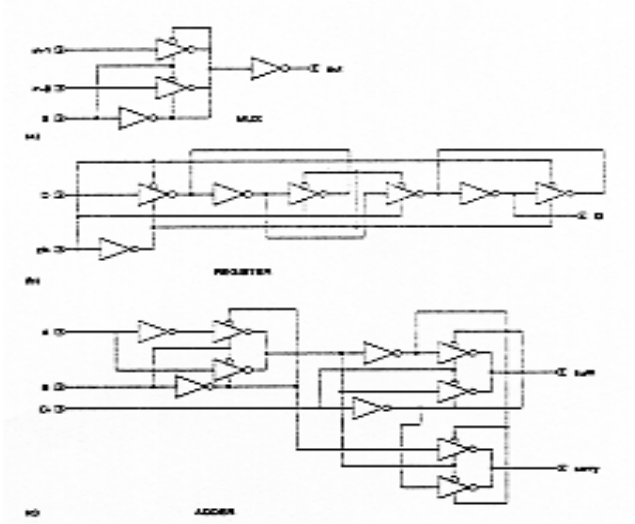


Fig 1.7: Regular design of a 2-1 MUX, a DFF and an adder, using inverters and tri-state buffers.

VLSI DESIGN STYLES

Several design styles can be considered for chip implementation of specified algorithms or logic functions. Each design style has its own merits and shortcomings, and thus a proper choice has to be made by designers in order to provide the functionality at low cost.

Field Programmable Gate Array (FPGA)

Fully fabricated FPGA chips containing thousands of logic gates or even more, with programmable interconnects, are available to users for their custom hardware programming to realize desired functionality. This design style provides a means for fast prototyping and also for cost-effective chip design, especially for low-volume applications. A typical field programmable gate array (FPGA) chip consists of I/O buffers, an array of configurable logic blocks (CLBs), and programmable interconnect structures.

It consists of four signal input terminals (A, B, C, D), a clock signal terminal, user-programmable multiplexers, an SR-latch, and a look-up table (LUT). The LUT is a digital memory that stores the truth table of the Boolean function. Thus, it can generate any function of up to four variables or any two functions of three variables.

The CLB is configured such that many different logic functions can be realized by programming its array. More sophisticated CLBs have also been introduced to map complex functions. The typical design flow of an FPGA chip starts with the behavioral description of its functionality,

using a hardware description language such as VHDL. The synthesized architecture is then technology-mapped (or partitioned) into circuits or logic cells. At this stage, the chip design is completely described in terms of available logic cells.

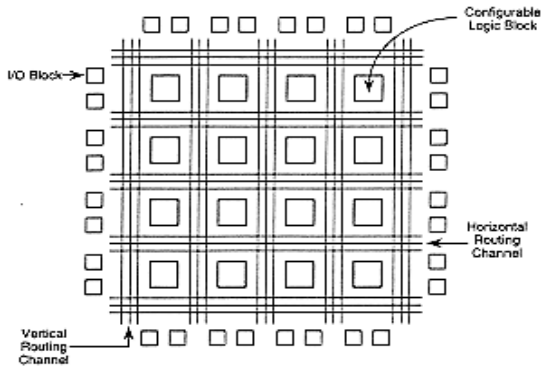


Fig 1.8: General architecture of Xilinx FPGAs

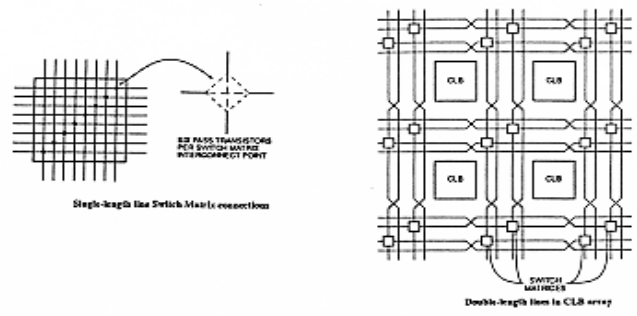


Fig 1.9: switch matrices and interconnection routing between CLBs.

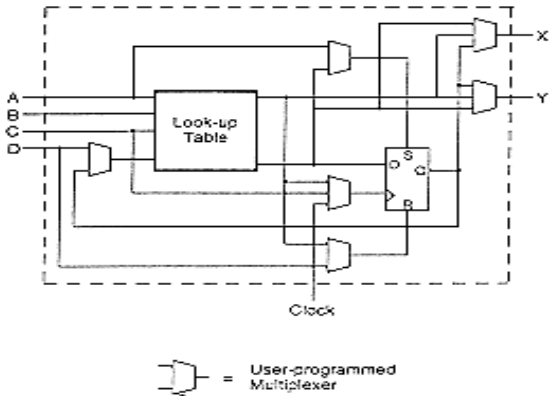


Fig 1.10: XC2000 CLB of the Xilinx FPGA.

Next, the placement and routing step assigns individual logic cells to FPGA sites (CLBs) and determines the routing patterns among the cells in accordance with the netlist. After routing is completed, the on-chip Performance of the design can be simulated and verified before

downloading the design for programming of the FPGA chip. The programming of the chip remains valid as long as the chip is powered-on or until new programming is done. In most cases, full utilization of the FPGA chip area is not possible - many cell sites may remain unused.

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

GATE ARRAY DESIGN

In view of the fast prototyping capability, the gate array (GA) comes after the FPGA. While the design implementation of the FPGA chip is done with user programming, that of the gate array is done with metal mask design and processing. Gate array implementation requires a two-step manufacturing process: The first phase, which is based on generic (standard) masks, results in an array of uncommitted transistors on each GA chip.

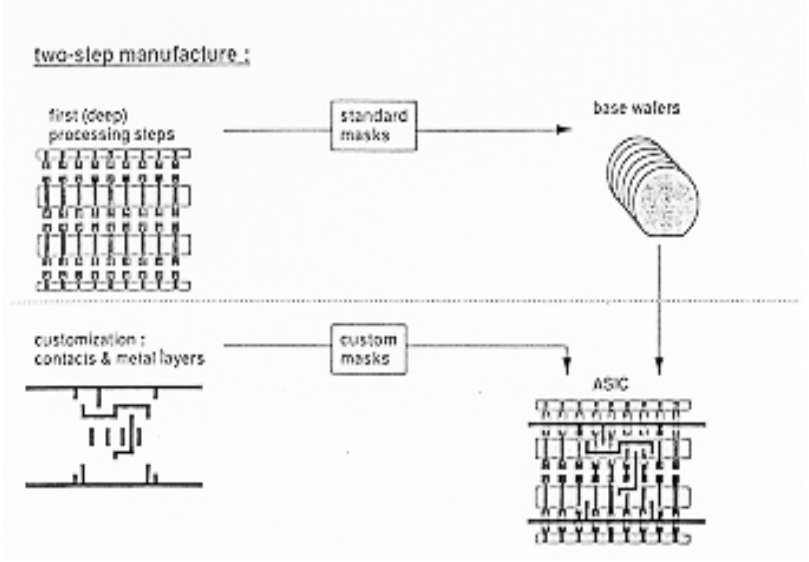


Fig 1.11: gate array implementation

Since the patterning of metallic interconnects is done at the end of the chip fabrication, the turn-around time can be still short, a few days to a few weeks.

The availability of these routing channels simplifies the interconnections, even using one metal layer only. The interconnection patterns to realize basic logic gates can be stored in a library, which can then be used to customize rows of uncommitted transistors according to the netlist.

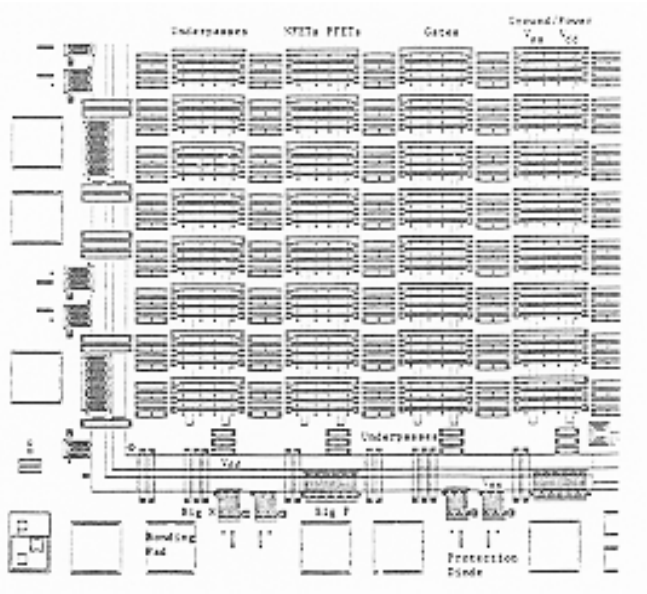


Fig 1.12: A corner of a typical gate array chip

While most gate array platforms only contain rows of uncommitted transistors separated by routing channels, some other platforms also offer dedicated memory (RAM) arrays to allow a higher density where memory functions are required. With the use of multiple interconnect layers, the routing can be achieved over the active cellareas; thus, the routing channels can be removed as in Sea-of-Gates (SOG) chips. Here, the entire chip surface is covered with uncommitted nMOS and pMOS transistors. As in the gate array case, neighboring transistors can be customized using a metal mask to form basic logic gates. For intercell routing, however, some of the uncommitted transistors must be sacrificed. This approach results in more flexibility for interconnections, and usually in a higher density.

In general, the GA chip utilization factor, as measured by the used chip area divided by the total chip area, is higher than that of the FPGA and so is the chip speed, since more customized design can be achieved with metal mask designs. The current gate array chips can implement as many as hundreds of thousands of logic gates.

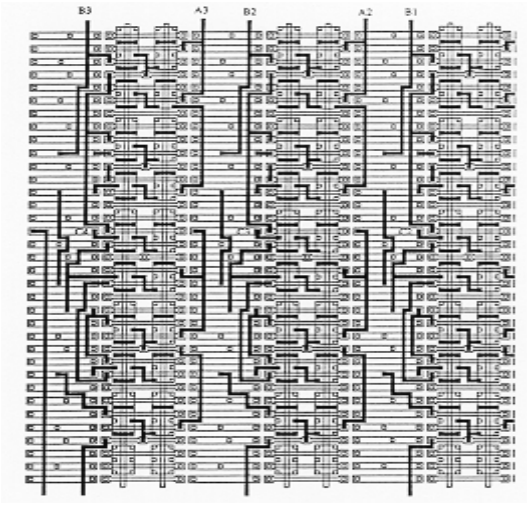


Fig 1.13: Metal mask design to realize a complex logic function on a channeled GA platform.

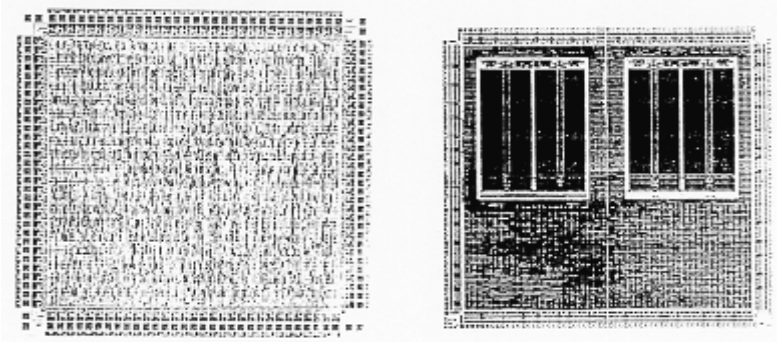


Fig 1.14: conventional GA chip and a gate array with two memory banks

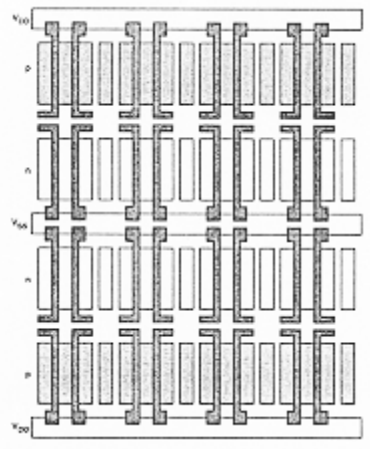


Fig 1.15: The platform of a Sea-of-Gates (SOG) chip

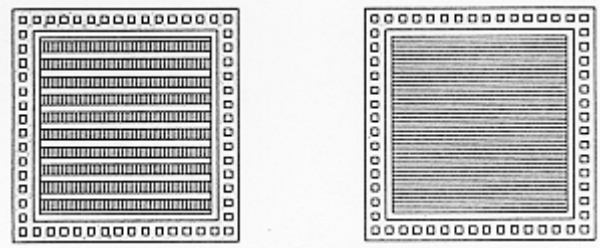


Fig 1.16: channeled (GA) vs. the channel less (SOG) approaches.

STANDARD-CELLS BASED DESIGN

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

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

To enable automated placement of the cells and routing of inter-cell connections, each cell layout is designed with a fixed height, so that a number of cells can be abutted side-by-side to form rows. The power and ground rails typically run parallel to the upper and lower boundaries of the cell, thus, neighboring cells share a common power and ground bus. The input and output pins are located on the upper and lower boundaries of the cell.

Inside the I/O frame which is reserved for I/O cells, the chip area contains rows or columns of standard cells. Between cell rows are channels for dedicated inter-cell routing. As in the case of Sea-of-Gates, with over-the- cell routing, the channel areas can be reduced or even removed provided that the cell rows offer sufficient routing space.

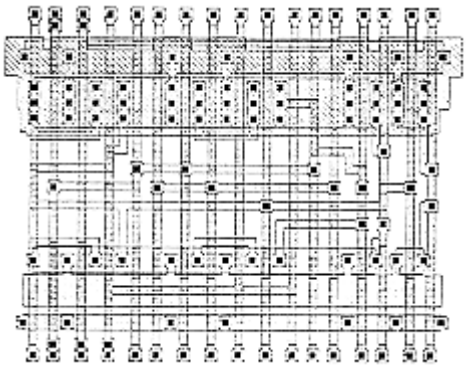


Fig 1.17: A standard cell layout.

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

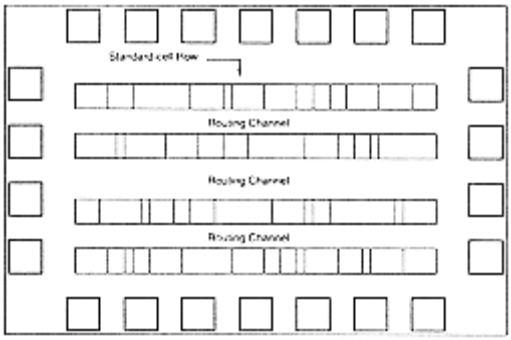


Fig 1.18: standard-cells-based design.

If a number of cells must share the same input and/or output signals, a common signal bus structure can also be incorporated into the standard-cell-based chip layout. Figure 1.23 shows the simplified symbolic view of a case where a signal bus has been inserted between the rows of standard cells. Note that in this case the chip consists of two blocks, and power/ground routing

must be provided from both sides of the layout area. Standard-cell based designs may consist of several such macro-blocks, each corresponding to a specific unit of the system architecture such as ALU, control logic, etc.

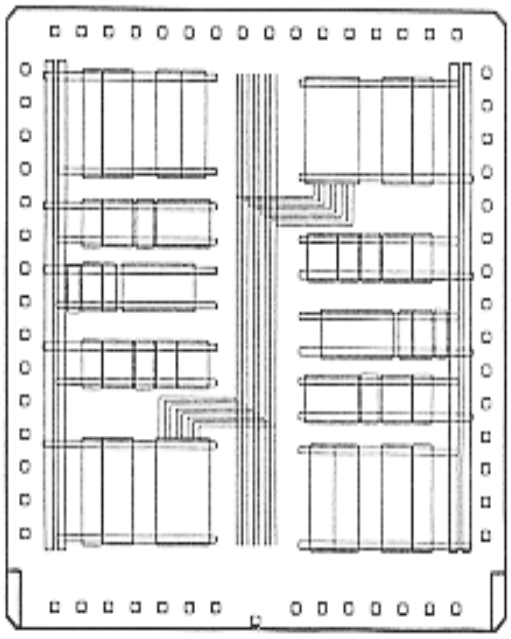


Fig 1.19: Simplified floor plan consisting of two separate blocks and a common signal bus.

After chip logic design is done using standard cells in the library, the most challenging task is to place individual cells into rows and interconnect them in a way that meets stringent design goals in circuit speed, chip area, and power consumption. Many advanced CAD tools for place-and-route have been developed and used to achieve such goals. Also from the chip layout, circuit models which include interconnect parasitics can be extracted and used for timing simulation and analysis to identify timing critical paths. For timing critical paths, proper gate sizing is often practiced to meet the timing requirements. In many VLSI chips, such as microprocessors and digital signal processing chips, standard-cells based design is used to implement complex control logic modules. Some full custom chips can be also implemented exclusively with standard cells.

Notice that within the cell block, the separations between neighboring rows depend on the number of wires in the routing channel between the cell rows. If a high interconnect density can be achieved in the routing channel, the standard cell rows can be placed closer to each other,

resulting in a smaller chip area. The availability of dedicated memory blocks also reduces the area, since the realization of memory elements using standard cells would occupy a larger area.

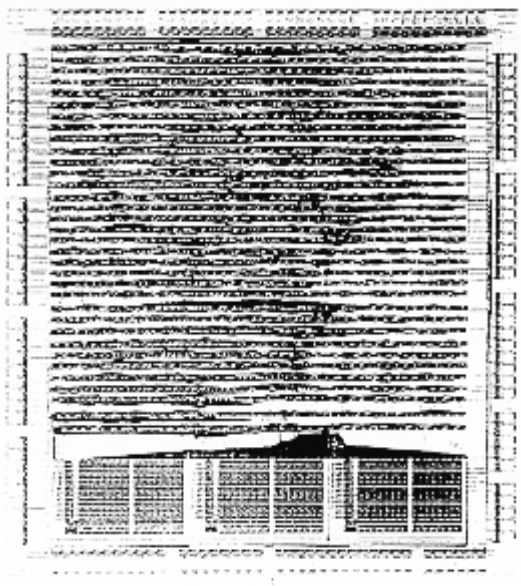


Fig 1.20: Mask layout of a standard-cell-based chip.

FULL CUSTOM DESIGN

Although the standard-cells based design is often called full custom design, in a strict sense, it is somewhat less than fully custom since the cells are pre-designed for general use and the same cells are utilized in many different chip designs. In a fuller custom design, the entire mask design is done anew without use of any library. However, the development cost of such a design style is becoming prohibitively high. Thus, the concept of design reuse is becoming popular in order to reduce design cycle time and development cost.

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

In digital CMOS VLSI, full-custom design is rarely used due to the high labor cost. Exceptions to this include the design of high-volume products such as memory chips, high- performance microprocessors and FPGA masters. Figure 3.21 shows the full layout of the Intel 486 microprocessor chip, which is a good example of a hybrid full-custom design. Here, one can identify four different design styles on one chip: Memory banks (RAM cache), data-path units consisting of bit-slice cells, control circuitry mainly consisting of standard cells and PLA blocks.

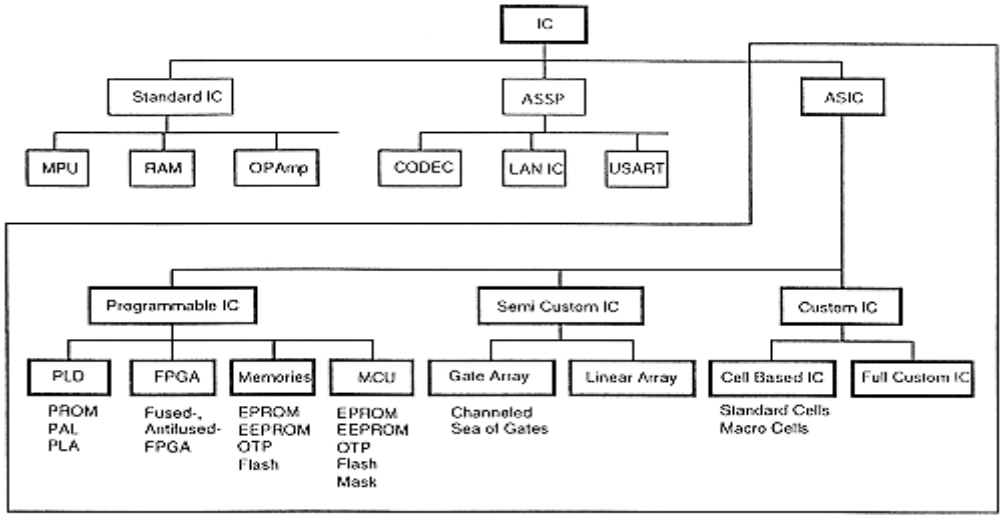


Fig 1.21: VLSI design style.