B.V.M Engineering College

(AN AUTONOMOUS INSTITUTION)

(ELECTRONICS ENGINEERING DEPARTMENT)

ACADEMIC YEAR: AY 2021-22

SUBJECT: 3EL42: Digital Logic Design

Institute Vision: "Produce globally employable innovative engineers with core values"

Institute Mission:

1. Re-engineering curricula to meet global employment requirements
2. Promote innovative Practices at all levels
3. Imbibe core values
4. Reform policies, systems and processes at all levels
5. Develop faculty and staff members to meet the challenges

Department: Electronics Engineering

Vision: "Produce globally employable, innovative Electronics engineers with core values"

Mission:

1. Promote Innovative Practices to strengthen teaching and learning process in electronics engineering
2. Develop faculty and staff to meet challenges in Electronics engineering
3. Adapt Engineering curricula to meet global requirements for Electronics engineering program me.
4. Reform policies, systems and processes at all levels
5. Imbibe core Values.

Program Educational Objectives (PEOs):

1. Learn hardware description language (HDL).
2. Utilize HDL to design and analyze digital systems
3. Learn field programmable gate array (FPGA) technologies and implementation of digital circuits using EDA Tools and PLD board.

A Project Report On

16-Bit Barrel Shifter using Verilog HDL

Mini Project

(3EL42)

SUBMITTED BY:

Harsh Mehta

(ID No. 19EL083)

IN PARTIAL FULFILLMENT FOR THE AWARD OF THE DEGREE

OF

BECHELOR OF TECHNOLOGY

IN

Electronics Engineering

BIRLA VISHWAKARMA MAHAVIDALAYA

Engineering Collage, Vallabh Vidyanagar, 388120

[An Autonomous Institution]

Electronics Department

Lab **Teacher**

Course Coordinator & Guide

Prof.Chintan Patel

ELECTRONICS ENGG DEPARTMENT

BIRLA VISHWAKARMA MAHAVIDYALAYA ENGG COLLEGE V V NAGAR

CERTIFICATE

This is to certify that M/r. MEHTA HARSH ANILKUMAR **ID No**. 19EL083, of B.Tech. (Electronics Engineering) SEM-VI has satisfactorily completed the term work of the subject **DIGITAL LOGIC DESIGN (3EL42)** prescribed by BVM an Autonomous Institution during the Academic Year **2021-2022**

Lab Teacher

Course Coordinator & Guide

Prof. Chintan Patel

Acknowledgment

Firstly, we offer thanks to our guide and Course coordinator Prof. Chintan Patel Electronics Department, Birla Vishvakarma Mahavidyalaya, our Lab teacher: Prof. Chintan Patel Lecturer, Birla Vishvakarma Mahavidyalaya, Electronics Department, for their invaluable support, guidance and advice given throughout this semester and for helping to establish our direction.

We express our deep and sincere sense of gratitude to specially, Prof. Chintan Patel Electronics Department, Birla Vishvakarma Mahavidyalaya, who has given us invaluable support and given opportunities to learn and develop researcher skills and has been unending source of inspiration to us. We also thankful to all our other faculty members for their consistence support and guidance.

Harsh Mehta

INDEX

CHAPTER 1

INTRODUCTION

AIM

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

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.

1.1.1 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.

CHAPTER 2

PRINCIPLE OF OPERATION:

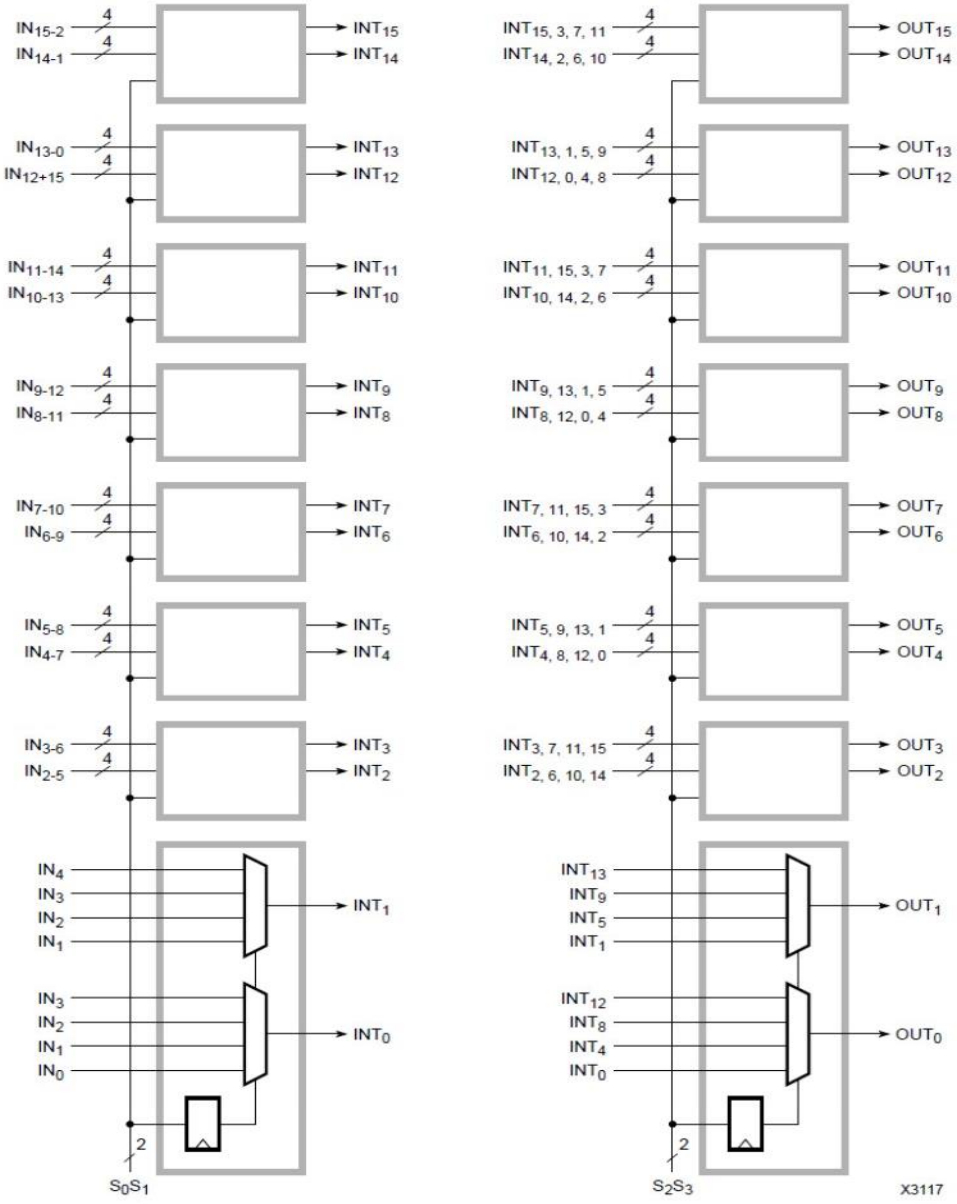
2.1 INTRODUCTION

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.



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 su 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 2*n*.

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 -bit numbers. The carry-select adder is simple but rather fast, having a gate level depth of $O(\sqrt{n})$.

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 ($\sqrt{n}$). 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($\sqrt{n}$) 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 Basic building block

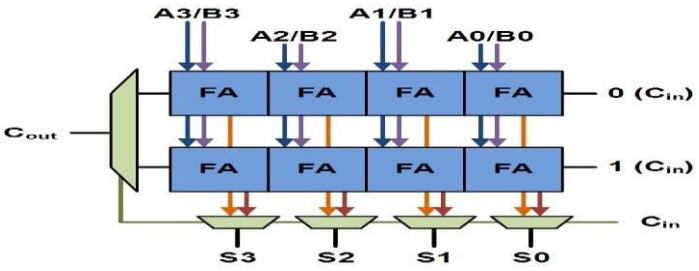


Fig 3.8 carry select adder

3.8.1 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

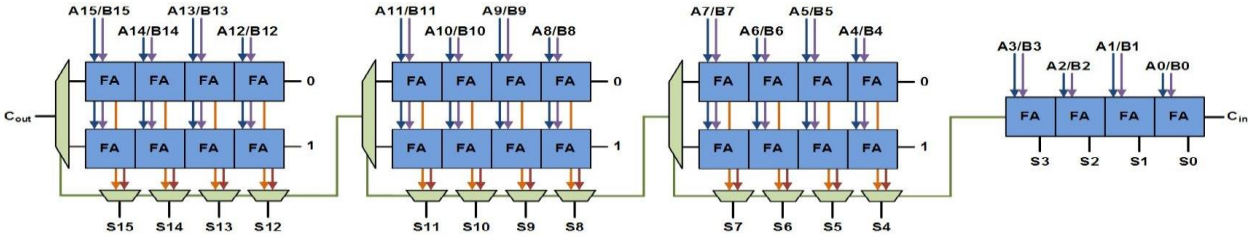


Fig 3.9 uniform sized adder

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.

3.8.2 Variable sized adder

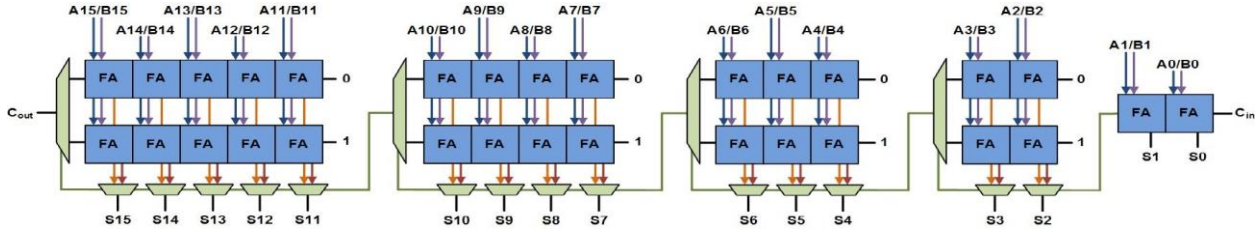


Fig 3.10 variable sized adder

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.

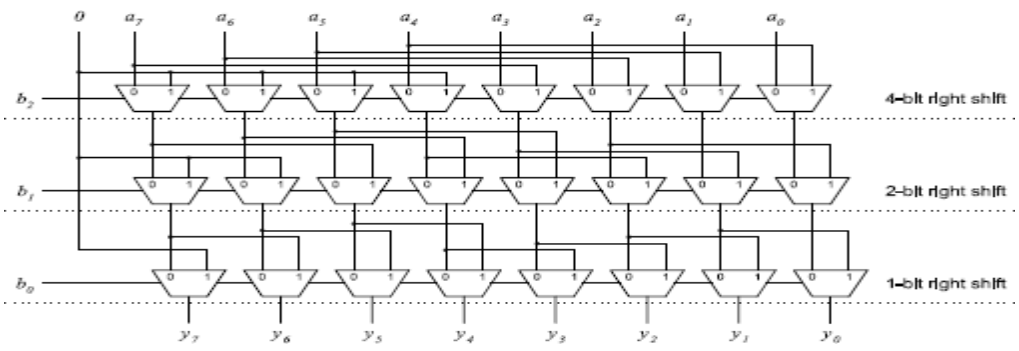


Fig 3.11 8 bit logical right shifter.

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.

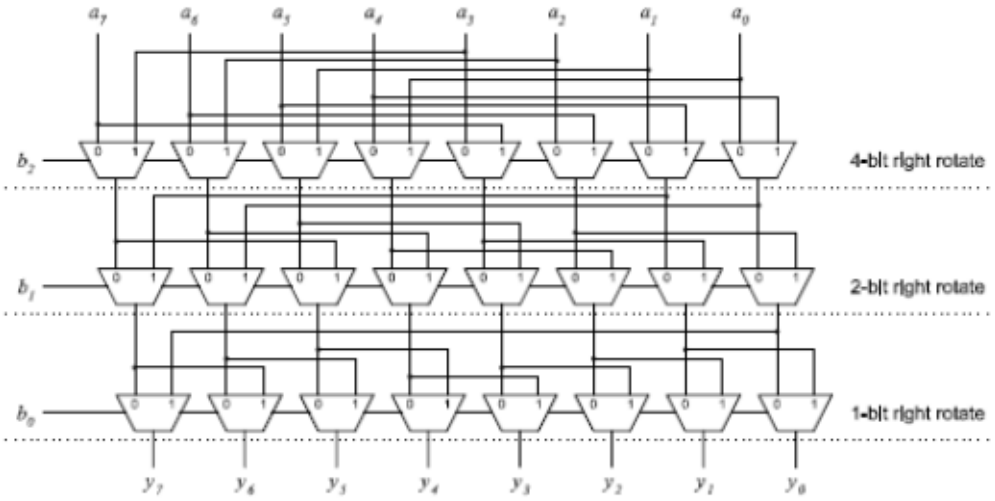


Fig 3.12. 8 bit right rotator.

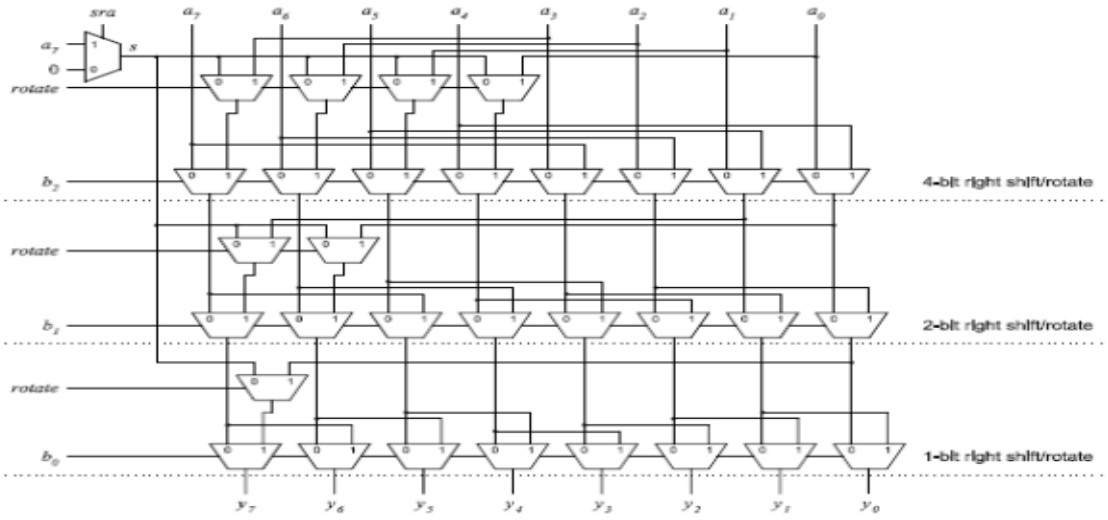


Figure 3.13 8 bit mux-based right shifter/rotator.

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.

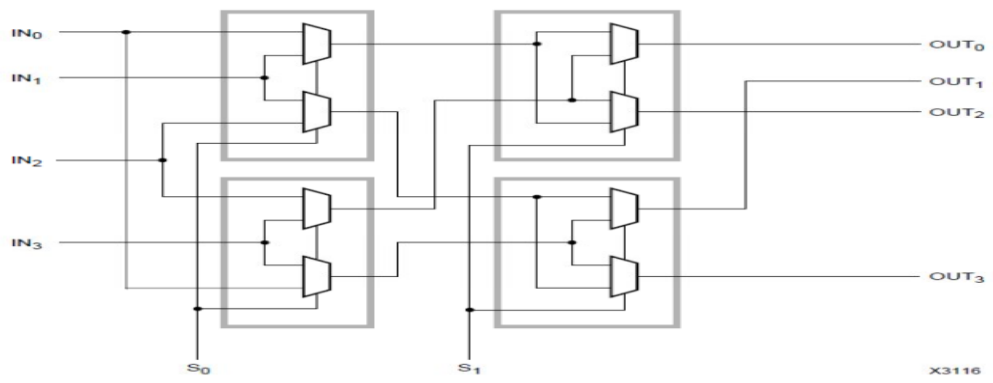


Fig 3.14. 4 bit barrel shifter

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.

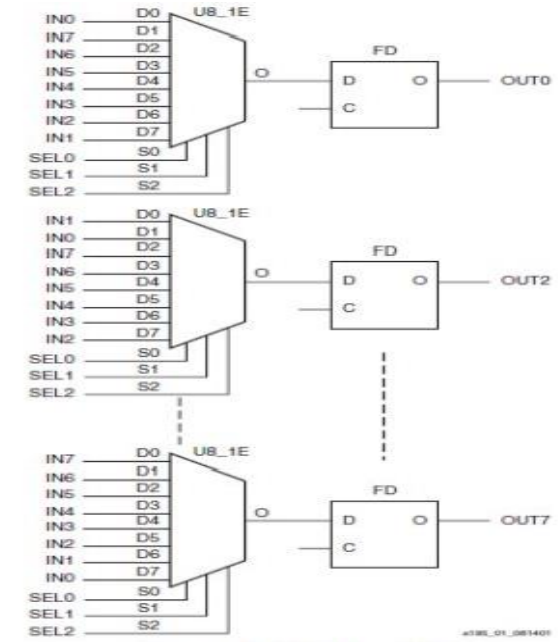


Figure 1: **Eight-Bit Barrel Shifter**

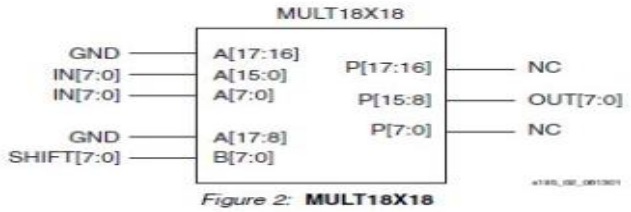


Figure 2: **MULT18X18**

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.

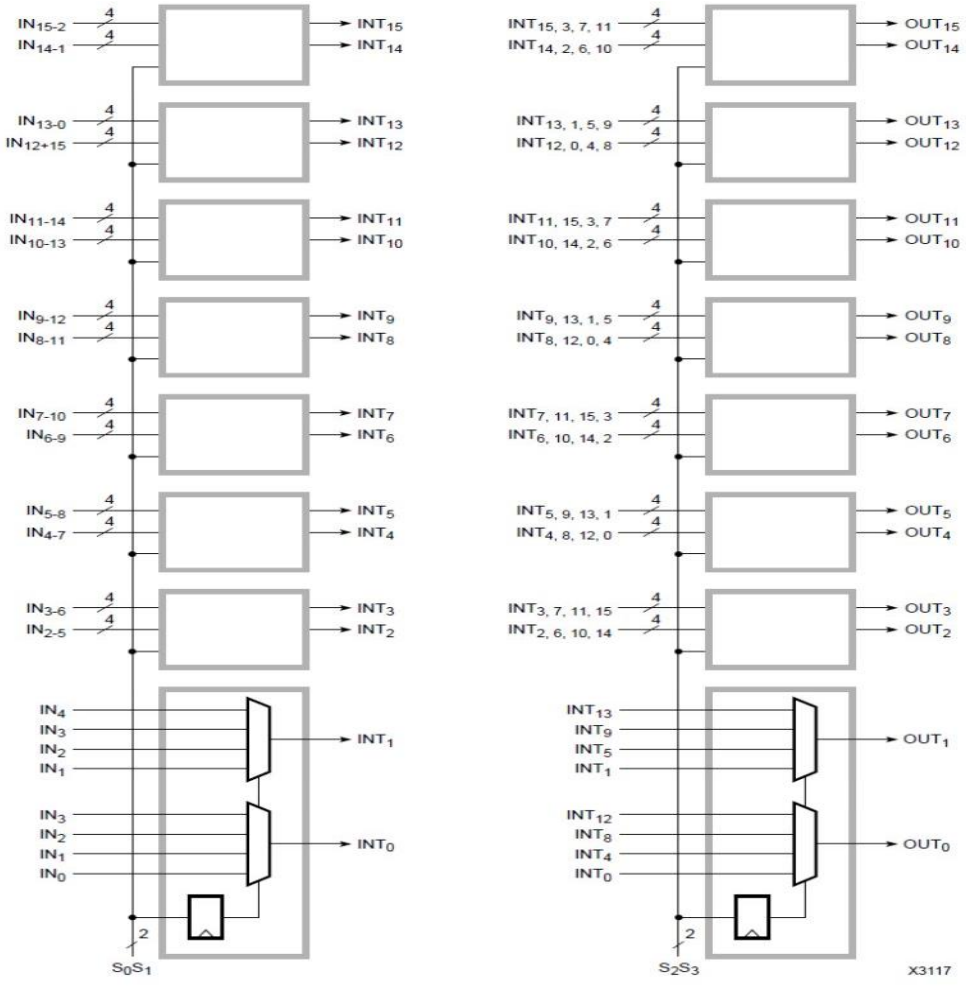


Fig 3.16 16 bit Barrel Shifter

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.

(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.

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.

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.

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
```

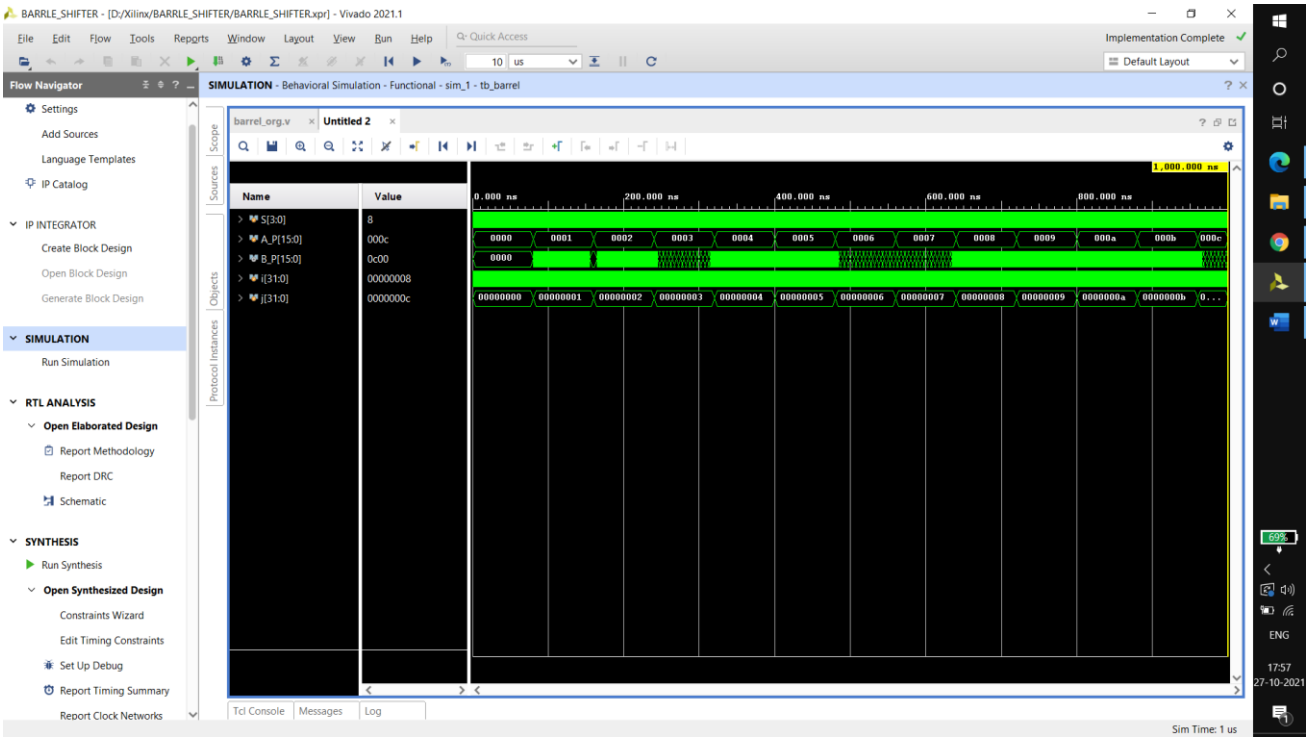
TestBench

```
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
```

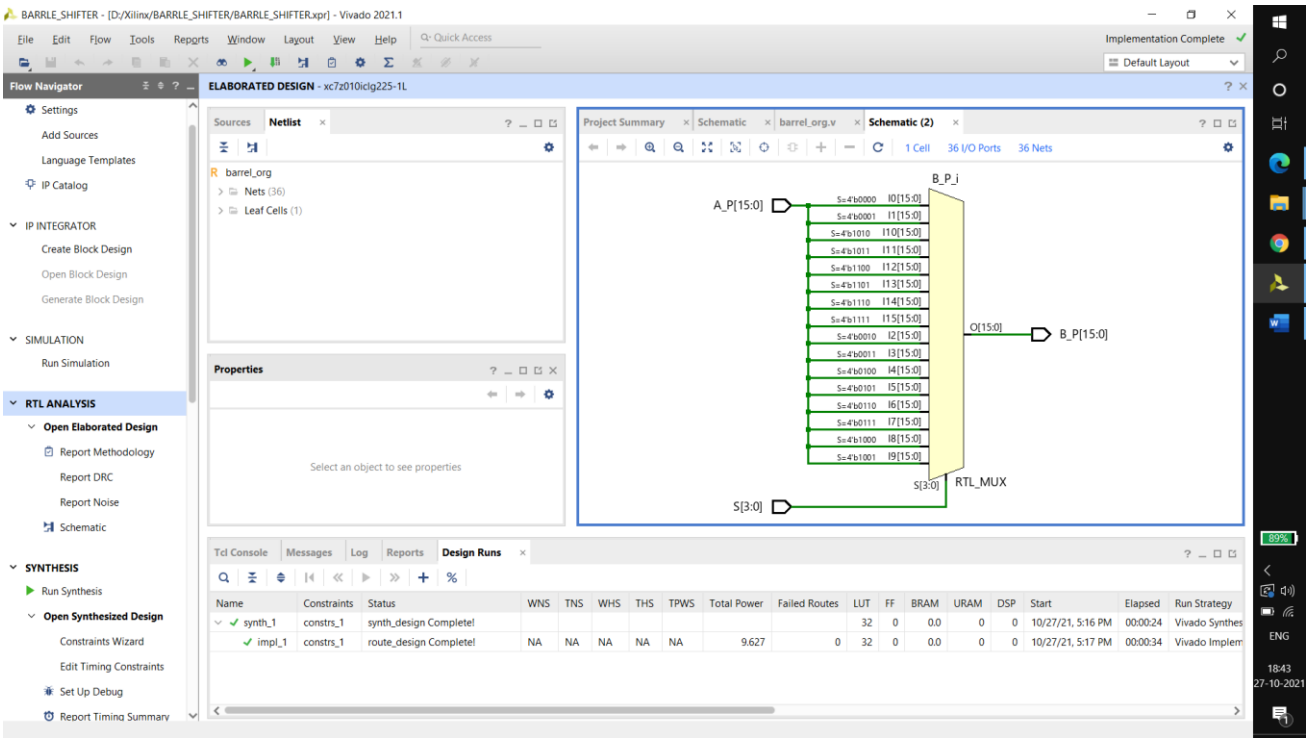
```
initial
$monitor ("S=%b,A_P=%b,B_P=%b",S, A_P, B_P); initial
  #50000000 $finish;
endmodule
```

4.2 RESULT

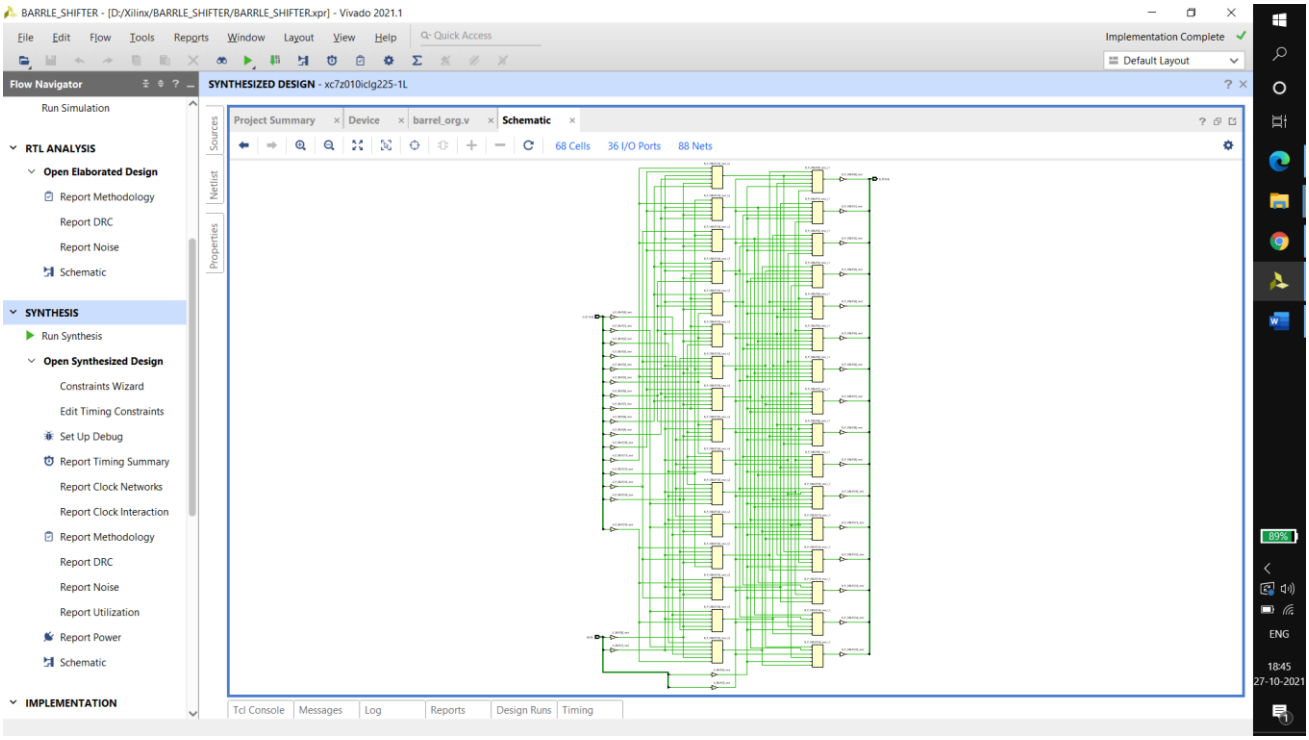
We have designed barrel shifter that shifts data. This was modeled in Xillix Vivado and the results were just as expected.



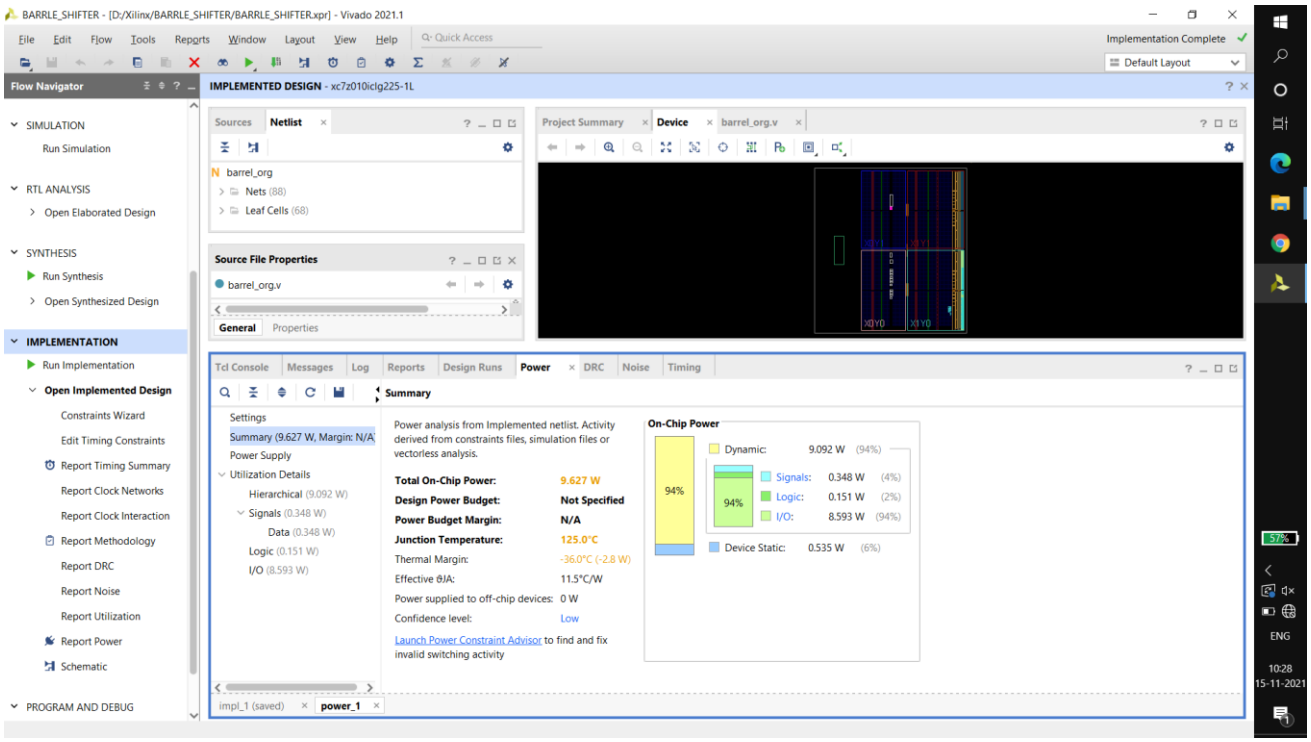
RTL Schematic:



Synthesized Design:



Power Report:



Synthesis Report:

synthesis_report.txt

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%2Frzaj cconcepts.html
- http://all.net/edu/curr/ip/Chap2-4.html