Implementation of an Efficient Library for Asynchronous Circuit Design with Synopsys

Tri Caohuu and John Edwards

1 Introduction

As the needs of the industry demand ever more complex integrated circuits, it becomes more difficult to supply a single uniform clock signal to the entire device. Currently, the transistors in a clock tree often use an amount of power comparable to the amount used by the transistors implementing the logic. The tree also occupies a large fraction of the chip's area. The problem of clock skew requires elaborate and costly solutions, such as placing multiple phase-locked loops on the same chip.

The alternative is asynchronous circuitry: digital logic without clock signals. Instead of communicating at regular and defined intervals, asynchronous interfaces use control signals to indicate when they are ready to process data. Asynchronous storage elements do not load new values when a clock ticks. Rather, data signals are accompanied by control signals, which notify storage elements when the data signals are valid and should be stored.

Industry standard tools, such as those from Cadence and Synopsys, are designed to deal with problems that arise in synchronous workflows. Timing tools for synchronous circuits tend to focus on solving long paths, but asynchronous circuits are subject to a broader range of timing problems. Common test methods, such as adding scanning to the registers, do not work for asynchronous circuits, which do not have clocked registers.

The asynchronous design methods also pose challenges to integration. The models of delay used in asynchronous circuits are rarely needed for circuits that are constrained by clocks. There is a need for special primitives for encoding data such as used in dual-rail protocols[1] and ternary logic[2]. Integrating these unique features with synchronous tools would be difficult. It would be more effective to begin with asynchronous workflows comparable to the synchronous ones.

In order to use common tools to design asynchronous circuitry, the components used by such circuits first must be integrated with the tools. The first step is to create a design library containing these components.

2 Asynchronous Library

This library should contain basic asynchronous components such as the Muller C-element, join, fork, mux and demux[1]. It can be incorporated into complex digital designs within common EECAD software. This makes it possible to develop a workflow for creating asynchronous circuits with these design tools.

The main feature of the library is a Muller C-element, a simple sequential device often used to control asynchronous pipelines. It has two input signals and one output. When both inputs are high, the output rises and remains high until both inputs are low. The output falls and stays low until both inputs rise again. The output does not change when the inputs are different. It can be used to represent a condition that depends on two prerequisite conditions.

C-element implemented as custom cell will obviously be more efficient than those constructed from logic gates. The implementation chosen was the symmetric static CMOS version, which uses 6 transistor pairs and is more efficient [3] than other static versions.

The transistors gated by the inputs (A and B) are a pull-up and pull-down network. The behavior of the four transistors in a network depends on the transistor gated by the output (C). If it is open, the network is equivalent to two transistors in series. If it is closed, the network is equivalent to two transistors in parallel. Depending on the value of C, the pull-up network will be in series and the pull-down network in parallel, or the other way around. They will behave as either a NAND gate or a NOR gate. The final two transistors are an inverter, changing the function to AND or OR.

T. Caohuu (✉) • J. Edwards
Department of Electrical Engineering, San José State University, San José, CA, USA

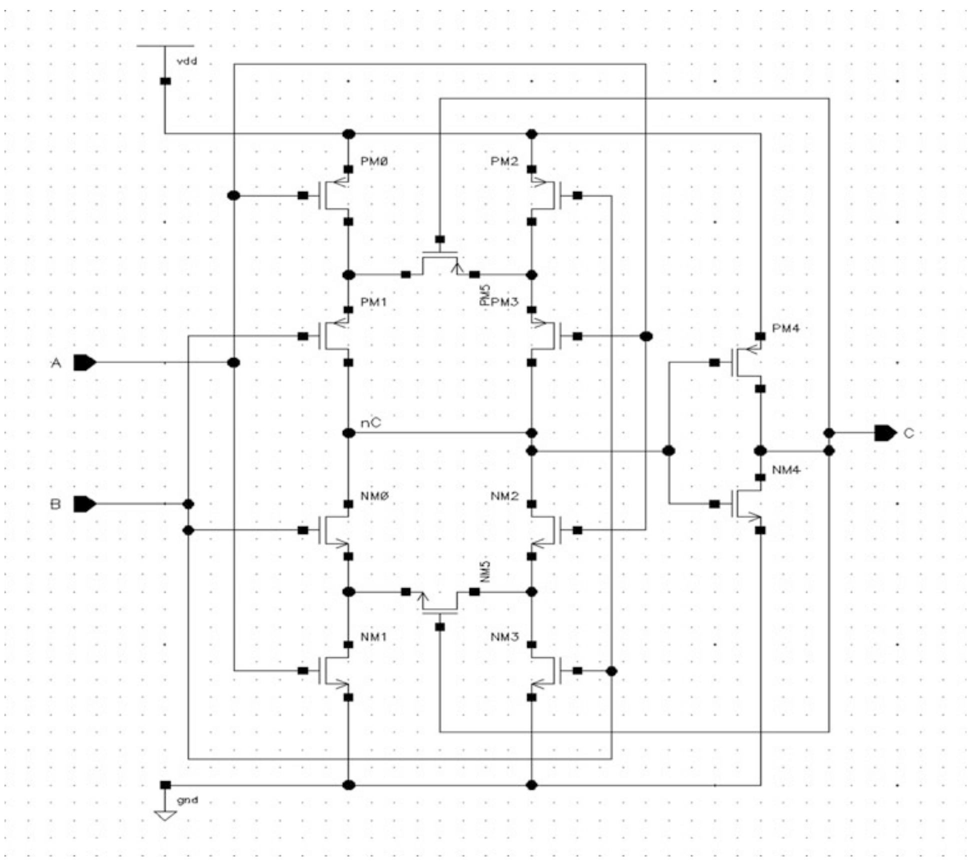

Fig. 1 C-element schematic

This cell was created on a 45 nm process. Transistors with a high threshold voltage were chosen, to minimize leakage current. The widths of the pull-up PMOS transistors were set at 480 nm, and the pull-down NMOS transistors at 240 nm. These were determined by the constraints of the desired cell size. After trying various sizes for the inverter transistors, widths of 480 nm for PMOS and 355 nm for NMOS were found to give approximately equal rise and fall times for a variety of loads.

To minimize cell area, the transistors were ordered using an Euler path technique [5].

The cells were converted to a netlist, which was then characterized by running simulations to measure the cells' electrical characteristics. Their logical and timing behaviors can be determined by this process. The characterizer produces three formats Verilog, VHD, and Liberty. The Liberty format is compatible with synthesis tools, such as Synopsys Design Compiler, and it can be used as a target when synthesizing asynchronous designs. The Verilog and VHDL designs also include timing information, and can be invoked from those languages when running simulations. This allows the accurate testing of asynchronous designs before and after synthesis.

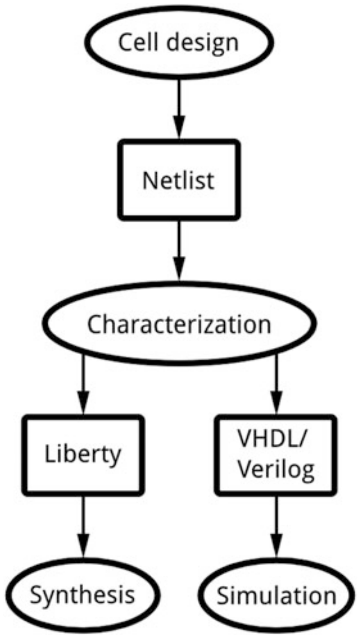

Fig. 2 Library workflow

3 Test Designs

3.1 Asynchronous FIFO

To test the library's performance, the C-element was used as part of an asynchronous FIFO. This FIFO was compared with a similar synchronous FIFO to investigate the relative merits of synchronous and asynchronous circuits for this application. The FIFO is practical for this kind of test, since it is commonly used, particularly in asynchronous circuits. One application is buffering data between different clock domains. An asynchronous FIFO could be used within a mostly synchronous IC.

This FIFO was based on the Muller circuit, a basic asynchronous design pattern. This is a pipeline where propagation is controlled by Muller C-elements.[1] The stages of the pipeline are separated by clock-less latches. Between the latches is only combinational logic. If there is no logic, data simply propagates through, and the pipeline is a FIFO. C-elements drive the control signals, a simple request and acknowledge pair.

It is important that the data signals and the control signals remain synchronized. If there are logic blocks, there must be delays in the control signals that use the same amount of time as the logic.[1]

The pipeline chosen in this design uses a push protocol: the request signal indicates to the next stage that data is available, and the acknowledge signal indicates to the sender that the data is being read. Each of the latches should store a data word whenever the previous latch is sending one and the next latch has already received the one currently stored.

In the push protocol, the request signal indicates to the next stage that data is available. In a pull protocol, it would indicate to the previous stage that there is room for more data. The C-element can be used in a similar manner to control a pull pipeline.

In the push protocol, the request signal is raised to push data to the next stage, and the acknowledge signal is raised once that data is stored in the next latch. In a 2-phase protocol, the request signal falls when the next data word is ready, and the acknowledge signal falls in response.

Since data is sent on both transitions, it must be stored on both transitions of the C-element's output. This is done with a double edge-triggered flip-flop, which is built by combining positive and negative edge-triggered flip-flops and using a multiplexer to select the proper data value. Yun, Beerel, and Arceo point out [4] the complexity of the control circuitry is reduced at the expense of requiring more space for data storage.

Several delays had to be inserted in each pipeline stage to meet timing requirements. The acknowledge signal sent to the previous stage must be delayed until the register has finished loading the data, because once the acknowledgment is sent, the data signals could change, and a hold violation would occur. Similarly, the request signal to the next stage must be delayed until the register's output has stabilized.

Setup violations could occur if the enable signal from the C-element arrived too soon after a transition on the data bus. This solution would be to place a delay block between the C-element and the register. However, this condition was not found to occur during simulations, because the C-element's internal delay was already longer than the setup time. Hence, no delay block was needed.

In case of a pipeline, additional delay proportional to the logic delay for each processing stage is added to the *enable* signal.

Once appropriate delay blocks are chosen, the stage module will adhere to the protocol, even when it is incorporated into a larger design. The asynchronous timing constraints can be met by the library internally, without any other action from the library user or changes to the EDA software.

The FIFO's depth can be changed by adding or removing stages. For this, depths from 4 to 24 were studied. They were implemented as VHDL entities and synthesized. During

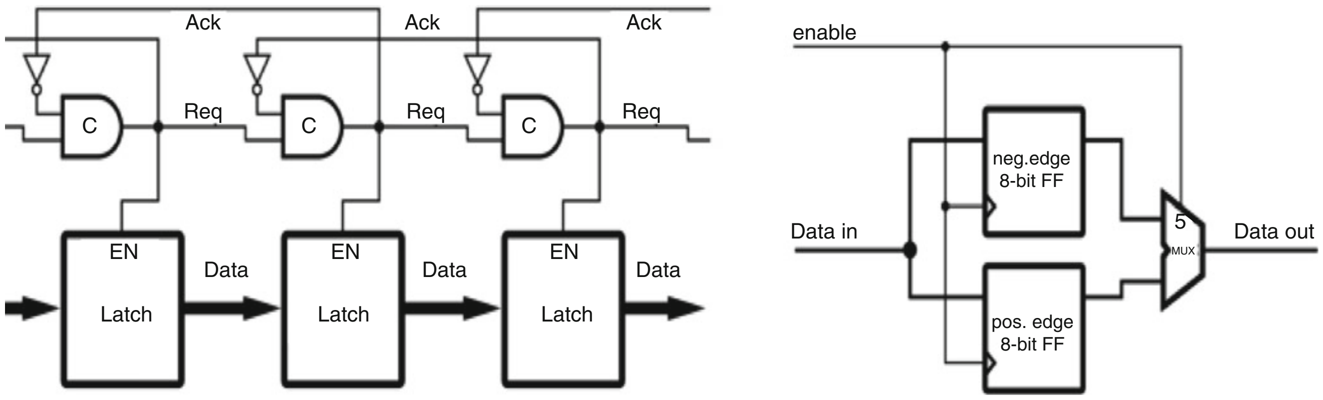

Fig. 3 Async FIFO and Latch Details

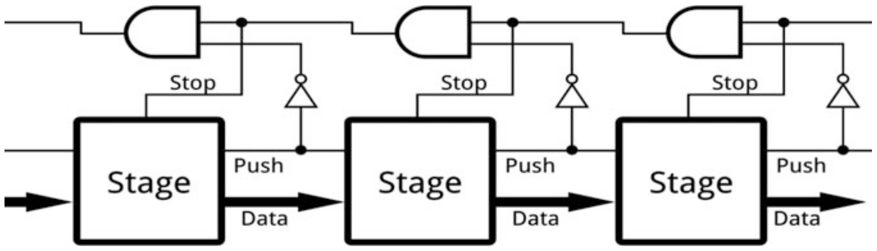

Fig. 4 Block diagram of synchronous FIFO

Table 1 Comparison data for depth 8

	Synchronous	Asynchronous
Maximum throughput	323 MHz	298 MHz
Latency	24.8 ns	12.0 ns
Area	709.0 μm^2	975.2 μm^2
Power (max throughput)	6.300 mW	7.471 mW
Power (200 MHz)	6.141 mW	5.907 mW
Power (100 MHz)	6.038 mW	4.305 mW
Power (40 MHz)	6.096 mW	3.406 mW

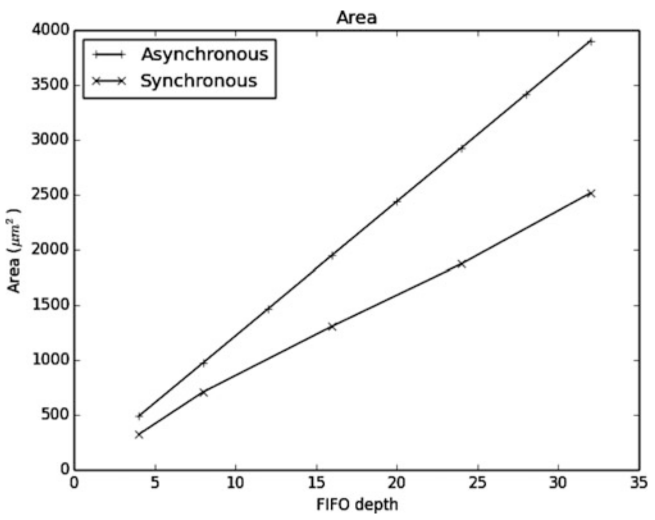

Fig. 5 Area vs. FIFO depth

synthesis, the C-element was provided by the asynchronous library and logic by a set of standard cells.

3.2 Synchronous FIFO

In this case study, a shift register based design was used for the synchronous FIFO. While a memory-based design would have much lower latency, it would differ too much from the asynchronous design architecturally for a more meaningful comparision.

Like the asynchronous design, it contained identical stages, and data words propagated from one stage to the next. The stages communicate with push signals, which indicate that a stage has valid data to send, and stop signals, which indicate that a stage is unable to receive data. As in the asynchronous FIFO, these signals determine whether a stage's register stores or holds. However, these registers are flip-flops, all driven by a single clock signal.

The synchronous FIFO's depth can also be changed by adding more stages. But the stop signals run the entire length of the pipeline, making a longer path if there are more stages. The clock frequency must be reduced to compensate.

This FIFO was also implemented in VHDL and synthesized, using as a synthesis target the same standard cell library used for the asynchronous FIFO.

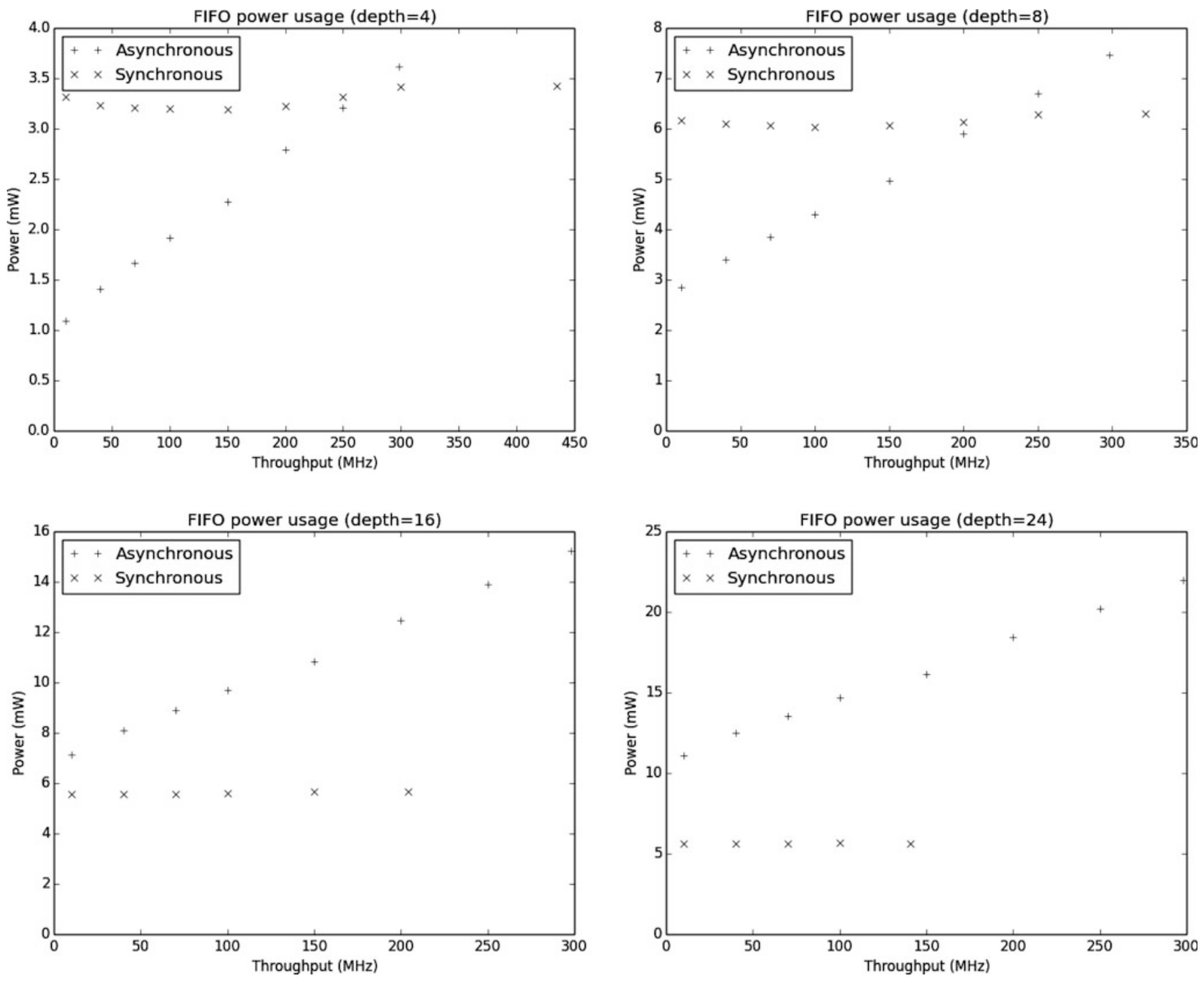

Fig. 6 Power usage vs. throughput at various FIFO depths

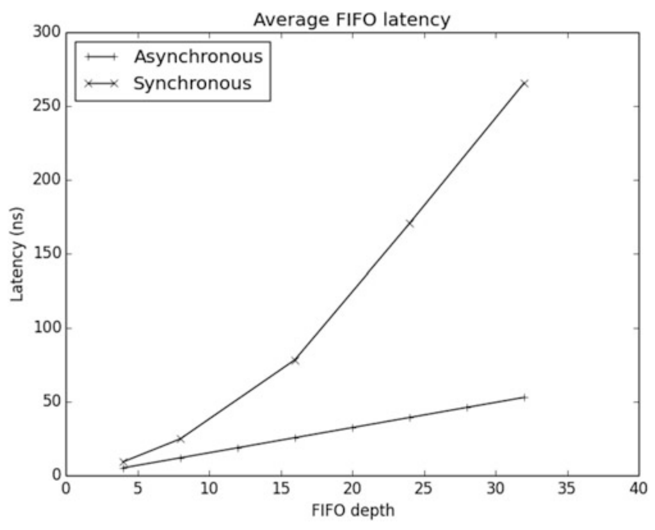

Fig. 7 Latency vs. FIFO depth

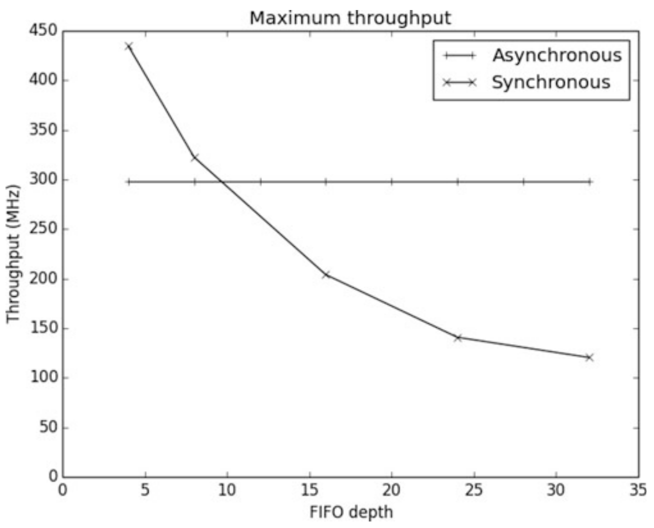

Fig. 8 Throughput vs. FIFO depth

4 Result Comparison and Analysis

The FIFO designs of various depths were compared on the criteria of minimum latency, maximum throughput, and area. The power usage was also measured at various throughput rates. The measurements were produced during the synthesis step. The synthesized designs were then tested to determine maximum performance.

The circuits' performance was quantified by measuring throughput and latency. Throughput was defined as the number of data blocks processed per second. Latency was defined as the time required for one data block to pass from one end of an empty FIFO to the other. For the asynchronous design these were measured by simulations, which used the HDL simulation version of the library.

When taking power measurements of asynchronous designs, the throughput was controlled by setting the switching rate on the write-side request signal. The switching rate of the read-side acknowledge signal was the same, because that is necessary to prevent filling and stalling the pipeline.

The synchronous FIFO clock rates were set to the highest values that did not cause timing violations. The throughput was controlled by the fraction of time the push signal was high. The latency was one clock cycle per stage.

Area varies linearly with depth for both designs. This is to be expected, since both are repetitions of a stage. The linearized curve for area used in the synchronous FIFO rises with a smaller coefficient and uses less area at any depth. This is to be expected, since the asynchronous FIFO has more complex control circuitry. The latency is linear for the asynchronous FIFO, but the synchronous FIFO shows a quadratic dependency.

The asynchronous FIFO's maximum throughput is independent of depth. The synchronous FIFO has an inverse relationship between depth and throughput. Note that they are equal at a depth of 10.

The asynchronous design saves power more effectively when utilization is less than 100 %. At a depth of 8, it required 45 % less power when idle and 25 % less when half idle. The asynchronous design does not show an advantage in power at depths of 16 or more.

5 Conclusions

We have demonstrated that the library performs successfully as indicated by the results obtained. We are able to invoke the Library at the HDL level similar like the case of other synchronous primitive.

While the chosen synchronous design does not reflect the most effective design, the throughput of the asynchronous is more or less constant with respect to the FIFO depth. Moreover, at a smaller depth the asynchronous design show clearly some power advantage. It would be worthwhile to develop more asynchronous primitive components to accommodate the design of asynchronous circuits of higher level of complexity.

References

1. Sparsø, J.; *Asynchronous Circuit Design: A Tutorial*; Technical University of Denmark; p.16-18, March 2006
2. Nagata, Y.; Mukaidono, M.;, "Design of an asynchronous digital system with B-ternary logic," *Multiple-Valued Logic, 1997. Proceedings., 1997 27th International Symposium on*; pp.265-271; May 1997

3. Shams, M.; Ebergen, J.C.; Elmasry, M.I.;, "Modeling and comparing CMOS implementations of the C-element," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol.6, no.4; p.564; Dec. 1998
4. Yun, K., Beerel. P., Arceo, J.; "High-Performance Asynchronous Pipeline Circuits", *ASYNC '96: Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems*; 1996
5. Roy, K. "Optimal Gate Ordering of CMOS Logic Gates Using Euler Path Approach: Some Insights and Explanation", *Journal of Computing and Information Technology*" vol. 15. No. 1 pp. 85-92, 2007