Comparison of performance between synchronous and asynchronous processors

A thesis submitted in partial fulfilment of the requirement for the award of the degree of

MASTER OF TECHNOLOGY

in VLSI Design

Submitted by

Rachit Saini

601662014

Under supervision of

Dr. Mohit Agarwal

Assistant Professor

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY

(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

JULY, 2018

Date: 5 July 2018

Internship letter for Mr. **Rachit Saini**

This isto certify that**Mr. Rachit Saini,** a studentof **M.Tech. (VLSIDesign)** from **Thapar Institute of Engineering and Technology, Patiala** is doing internship at Intel Mobile Communication Private Limited from 2nd Aug 2017 – 3rd Aug 2018.

Thanking You.

For Intel Mobile Communications India (P) Ltd

Anshul Agrawal

Engineering Manager

Intel mobile communication India Pvt. LTD

DECLARATION

I, Rachit Saini, hereby declare that the work presented in this thesis entitled **"Comparison of performance between synchronous and asynchronous processors"** in partial fulfilment of the requirement for the award of the degree of Master of Technology (VLSI Design) submitted at **Electronics and Communication Engineering Department, Thapar Institute of Engineering and Technology (Deemed to be University), Patiala** is an authentic record of work carried out under supervision of **Dr. Mohit Agarwal (Assistant Professor, ECED)** from July, 2017 to July, 2018. The matter presented here has not been submitted in part or full to any other University or Institute for the award of any other degree.

Date : ..13/..7/..2018....

R Saini

Rachit Saini

601662014

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

Date : ..13/..7/..2018....

Dr. Mohit Agarwal

Assistant Professor

Electronics and Communication Engineering Department

Thapar Institute of Engineering and Technology

(A deemed to be University), Patiala, Punjab

ACKNOWLEDGEMENT

I extend my gratitude to my project guide, **Dr. Mohit Agarwal**, Assistant professor (ECED) who is a constant source of motivation and firm support in carrying out this project. The support and supervision he gave has helped me to progress in the project. His co-operation is highly appreciated and I highly oblige to him for his valuable comments and moral support during this research period. Moreover, I would like to thank **Dr. Alpana Agarwal, HOD, ECED** for her consistent motivation.

I take the opportunity to express my thanks to my project lead, **Abraham Thomas**, my mentors, **Ramya** and **Surendhar** along with my manager, **Anshul Agarwal** at Intel India Private Limited, for their timely support and guidance.

Moreover, I am grateful to my friends at **Thapar Institute of Engineering and Technology, Patiala** and family for constant encouragement and support.

DATE : 13/7/2018

PLACE : PATIALA

Rsaini

Rachit Saini

601662014

ABSTRACT

Up till past few years, the synchronous processors have been at the forefront in performance due to well established methodology and existence of tools to work with. However, the question remains whether it is possible to increase performance by applying some new methodology. One of the potential solutions to this issue is the use of asynchronous methodology. Asynchronous processors have been pondered over for producing result with lesser time and power consumption as compared to synchronous processors. In this thesis, the focus is on comparing the performance of synchronous processor and asynchronous processor. A synchronous processor operates as per a global clock whereas an asynchronous processor operates using the handshaking signals. The time required for computation for both methodologies is compared to prove effectiveness of asynchronous methodology over synchronous methodology. Both types of processors have same set of blocks, however, the way of execution of instructions in both the processors varies. Moreover, a hybrid of synchronous and asynchronous methodology is being utilized these days in order to capture the advantages of both the methodologies.

Keywords – Asynchronous, synchronous, methodology, processor

CONTENTS

FIGURES

LIST OF ABBREVIATIONS

ITRS	International Technology Roadmap for Semiconductors
SoC	System on Chip
NoC	Network on Chip
GALS	Globally Asynchronous Locally Synchronous
Req	Request
Ack	Acknowledge
QoS	Quality of Service
EMI	Electromagnetic Interference

CHAPTER 1

INTRODUCTION

As there have been some challenges in late-Moore era, there is a slight shift in the interests of engineers from synchronous design implementations to the asynchronous design implementations. The International Technology Roadmap for Semiconductors (ITRS) has highlighted the trends such as facing the effect of enhanced variability, bottlenecks due to power and heat, rates of high fault which occur due to ageing and scalability issues with increase in transistor number to multibillion range and use of multi-core architecture [1].

As technology develops at submicron level, some problems like closure of time and too much usage of power occur for synchronous systems which depend on global clock. The asynchronous circuit design methodology is a strong alternative for solving the before mentioned problems [2].

1.1 MOTIVATION

Though the synchronous paradigm, based on single clock, has been in the industry for years, the use of either standalone asynchronous design or hybrid of asynchronous and synchronous elements result in object oriented distributive hardware systems that are inclined to support modular and expandable composition which provides operation on demand without instrumented management of power and tolerant design variable in nature. The report of ITRS highlights the use of asynchronous design methodology as a vital component to resolve the issues posed by the synchronous methodologies [1].

1.2 OBJECTIVE

- To present a case for asynchronous methodology as an alternative to synchronous methodology in terms of computation time parameters.
- To show advantages of asynchronous circuits whether standalone or in complementation with synchronous circuits.

1.3 PROBLEM STATEMENT

The aim is to compare the timewise performance of basic architecture using synchronous and asynchronous methodologies. This basic architecture consists of instruction set, instruction decoder, arithmetic and logic unit and a register set. The comparison rests on performance parameters, clock in case of synchronous circuit and handshake mechanism in case of asynchronous circuit between different modules.

1.4 ORGANIZATION OF DISSERTATION

The whole thesis is divided into eleven chapters. **Chapter 2** describes the literature survey. **Chapter 3** presents block diagram for synchronous processor and asynchronous processor and their explanation. This chapter further highlights points common to asynchronous and synchronous methodologies. **Chapter 4** gives details of instructions in instruction set. **Chapter 5** and **Chapter 6** mention the SystemC code for synchronous and asynchronous methodologies respectively. **Chapter 7** compares the performance between two methodologies. **Chapter 8** presents conclusion and scope of future works. References are enlisted at last.

CHAPTER 2

LITERATURE SURVEY

2.1 SYNCHRONOUS APPROACH

The oscillating crystal is used in the synchronous processor which vibrates at a regular frequency, either in gigahertz or in megahertz, based on the voltage applied. The chip functioning is integrated using the clock which gives out the signals on the circuit path and therefore, the registers, the flow of data and the sequence of performance of important tasks are controlled using the clock [3].

One main benefit of the synchronous chips is that the arrival of signal is irrelevant. The signals are captured by the registers at a clock edge, even if they arrive at different times. The signal processing takes place in the proper order if they arrive before the clock edge. The second advantage of synchronous systems is that the performance calculation utilizes only counting the number of clock cycles required to end the operations. However, performance of asynchronous design is less calculative [3].

With synchronous designs becoming larger and bigger, several types of flaws and inefficiencies creep in. The chip's longest wire delay should be less than duration of clock's single cycle. The clock ticks determine the operation and as a result, the tasks completion performed by parts of the chip before ticks have to wait for proceeding of tick. With escalation in size and intricacy of the chip, it is becoming difficult for clock ticks to get to all components within the chip [3].

In order to cope with the increasing difficulties for designing with synchronous approach, more sophisticated and expensive methods such as bus hierarchies and circuit hierarchies are being utilized. This is done to keep track of clock at different elements of the circuit [3].

As per Ryan Jorgenson, the components on chip have separate clocks these days that interact using buses. So, the clock ticks only cross individual components. Also, the heat is produced due to power consumption by clock. Moreover, in synchronous designs the registers consume energy to switch in order to collect the data on the tick of clock depending on whether they process inputs or not. However, only inputs make the gates switch in asynchronous designs [3].

The synchronous methodology these days in most of digital circuits gives a benefit of ease of implementation and well-established infrastructures. However, several problems are being faced by this methodology. The synchronization of data among domains using single global clock with inclusion of multiple timing domains has become hard. Moreover, the latency overhead is quite considerable in glue logics. Major portion of power being consumed by the clock circuitry is a critical limitation in design of low power circuits. Also, the electromagnetic interference problems is caused by the substantial radiation spectra from clock spikes [2].

2.2 ASYNCHRONOUS APPROACH

Asynchronous or self-timed processors also known as clock less processors does not include crystal oscillation to serve regular ticks of a clock. The result of the work is passed by elements of the asynchronous processors when they are done. The new developments have resulted in boosting of clock less chips performance. The low electromagnetic interference generated along with efficient power use are other advantages of clock less chips. These factors have made chips more robust [3].

Many digital systems are being used these days as System-on-chips (SoCs), within which separate functional blocks require different clock and frequency of operation. It is challenging to distribute the global clock of high frequency with low skew in order to make the selections in different areas such as design effort, area of die and requirement of power. These factors have become more difficult to decide due to integration of several systems on a single chip. An asynchronous methodology appears to be a better alternative than synchronous methodology to handle these requirements [4].

Stretchable clock – In design using stretchable clock, clock is locally generated to stop or extend to minimize meta-stability during transfer of data among separate domains of clock. This concept is same as clock gating in synchronous circuits. It results in discarding of needless clock cycles [4].

Asynchronous technique makes use of common case without the involvement of timing relationship between synchronous clocks whereas loosely synchronous technique makes use of common case between clocks, in clock frequency and phase [4].

2.3 DISADVANTAGES OF SYNCHRONOUS CORE

In synchronous core, the same clock input is used for each part of the core. The synchronous methodology has some flaws. The slowest element restricts the clock frequency. The operation processing time is not constant. And is information dependent. Faster switching happens due to higher voltage or electric energy. With increase in heat or temperature, the operation of gates gets slow [4].

Energy is required by all the elements of the core, even in case of no activity. The switching of clock by clock tree at least consumes energy. The higher clock rates results in higher frequency radiation. Moreover, clock with high frequency requiring quartz piezo-electric oscillator consumes a lot of space of the chip area.

Despite the mentioned disadvantages with synchronous methodology, designers continue to struggle with this methodology because it is well understood and documented along with presence of variety of tools [4].

2.4 POINTS IN FAVOUR OF ASYNCHRONOUS CORE

The elements of asynchronous circuits work with lack of global clock and are responsible for system timing. Using this approach has several benefits. The synchronous methodology encloses toggle lines and signals are pre-charged and discharged. However, the transitions in the asynchronous methodology are involved only in current computation. There is no clock skew in asynchronous circuits since they do not have distributed global clock. The synchronous circuits need to wait until every computation is finished before result is latched, giving inferior performance. The use of asynchronous circuits result in average-case performance. Inside the asynchronous cores, maximum number of transistors don't switch at the same moment for major part of the operations. The result is much lower electromagnetic radiation as compared to the radiation in synchronous cores [4].

2.5 POINTS AND RECURRING THEMES REGARDING ASYNCHRONOUS APPROACH

The asynchronous circuit design is highly convoluted in ad hoc fashion as compared to synchronous circuit design. For asynchronous circuits, existing CAD tools cannot be leveraged off and for synchronous circuits, the implementation circuits cannot be leveraged off. Moreover, it is not accurate to test event driven asynchronous circuit by time driven test equipment. The logically redundant gates are required by asynchronous circuits to eliminate hazards [4].

The clock less elements which do not operate based on clock ticks but on whether there is work to do uses perfect clock gating. Passing of the data between logic modules is controlled by local handshaking [3].

More power is needed by certain handshakes than the operation of a clock. However, consumption of power by an asynchronous circuit happen only when it activates [3]. The commercial products in recent years have migrated to asynchronous design through leading companies. There have been many instances of commercialization of asynchronous designs in the past two decades with substantial cost benefits. Though wide range of applications for asynchronous circuits exists, a few commonly recurring themes are exhibited [1]:

Extreme fine-grain pipelining – The capacity to deliver and capitalize on excessively fine-grain, gate-level and bit-level pipelines, unconstrained by the requirement of disseminating a high speed fixed rate clock [1].

Data-dependent completion times - To back micro-architect systems which can maneuver the minute variations in data-dependent completion time i.e. at a miniscule level [1].

Averting difficulties happening because of rigid global timing - Assisting new patterns of computation, extreme micro-parallelism, dynamic computational adaptive, and ease of large-scale system integration [1].

Sturdy variations in voltage, temperature and process – Allowing smooth compliance of dynamic variations of time [1].

On-demand, i.e. event-driven, operation – Computing with proportions of high-energy which does not require elaborate utilization of clock gating at multiple design levels [1].

2.6 ADVANTAGES OF ASYNCHRONOUS DESIGN

The voltage spikes are caused in synchronous design with data movement on all edges of clock. However, in chips without clock, current flow spans out at different times with respect to data, hence the power and pulsation of spikes reduces, resulting in lower emission of EMI. This further reduces errors within the circuit related to noise and also the interference with nearby devices [3].

No clock exists in asynchronous chips. Only the circuit used in these chips is powered up. Less energy is consumed by asynchronous processors as compared to synchronous counterparts. This is because they send only necessary required voltage for a particular operation [3].

The processors without clock trigger the circuit required to manage data and leave unused circuits which react instantaneously to other demands. So the clock less chips run cooler with sparse and low voltage spikes. Moreover, there is a bleak chance of temperature related problems [3]. The handshaking in clock less chips gives time to data to appear and settle before sending it off. This adds to reliability of the chip [3]. The standard components can be used in asynchronous chips because no modules are needed to work at same clock frequency leading to easier and quicker design and assembly [3].

2.7 CLOCKLESS CHALLENGES

Problems faced by asynchronous chips :

Assimilating clock less and clocked circuits – Both asynchronous and synchronous circuitry needs to interface in clock less chips. However, asynchronous chips do not finish instructions based on time dictated by a clock. This variation can give problems while blending with synchronous circuitry, chiefly memory and bus systems [3]. Data bits are required to be authentic and appear by each clock tick in clocked components, while asynchronous components provide approval and arrival at their own speed. Therefore, to co-ordinate asynchronous information with synchronous system clock, there

is a need of special circuits [3].

Dearth of tools and expertise – The supply of coding and design tools for asynchronous processors is scarce. Moreover, there is a dearth of expertise [3].

2.8 A HYBRID FUTURE

The chip system consisting of different clock circuitries in design is tied together by clock less circuitry that passes data among the clock circuits. So the synchronous chip is benefitted by asynchronous design. By bringing in use the handshaking to communicate through asynchronous fabric, the synchronous circuitries become capable of working at different clock speeds [3].

2.9 MERITS AND DEMERITS OF ASYNCHRONOUS METHODOLOGY

An asynchronous circuit uses handshaking protocol to synchronize and transfer data instead of using a global clock. The adjacent modules in asynchronous circuit localize data transfer within the related modules. This leads to an average-case delay. However, in case of synchronous circuitry, the delay is the worst due to dependence of clock frequency on slowest component [5], [6]. The absence of global clock in asynchronous circuitry allows designers to implement operations requiring low-power implementation and making circuits without consideration of the timing closure through clock tree balancing [7], [8], [9]. `The desirable portability and reusability is assured by asynchronous circuit that communicates with one another using a handshake protocol [2].

The limitations of asynchronous circuits include generation of hazard control signals, testability and a lack of commercialized asynchronous CAD tools. Due to these limitations, it is generally not easy for designers accustomed to synchronous design to make fully asynchronous system [2].

2.10 COMMUNICATION PROTOCOL

The asynchronous systems are classified as a unit of components which communicate using handshaking channels. These communication protocol and data encoding define these channels. Depending on these parameters, intricate asynchronous circuits can be customized. Synchronization and arbitration are two main agendas that exist in agglomerating asynchronous components in bigger systems [1].

The main frame of an asynchronous communication channel, between the two parties, is shown in figure 2.1. Overlooking transaction of data for now, the channel is usually implemented by two wires: req and ack[1]

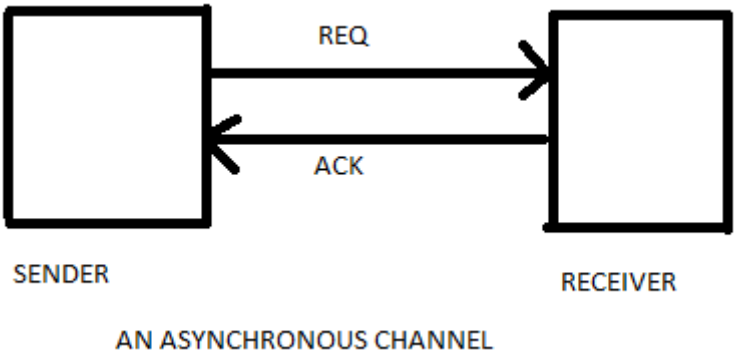

Fig 2.1: An asynchronous channel

To illustrate a single communication transaction, two handshaking protocols are used as shown in Figure 2.2 and Figure 2.3: 1) a four-phase protocol (return-to-zero [RZ]), and 2) a two-phase protocol (non-return-to-zero [NRZ], also known as transition- signaling). The req and ack signals are initially at zero in a four-phase protocol. The sender asserts req signal and the receiver in turn asserts ack signal, in the active or evaluate phase. In return to zero or reset phase, these signals are de-asserted whereas there is no return to zero phase in two phase protocol.

There are striking trade-offs between both the phases. A four phase protocol has the benefit of returning interface to a unique state. It simplifies the hardware design and is also good for dynamic logic. However, it results in lower throughput [10] [11] [12] [13]. A two phase protocol results in complex hardware design but has higher throughput due to one round trip communication per transaction [14] [15] [16] [1].

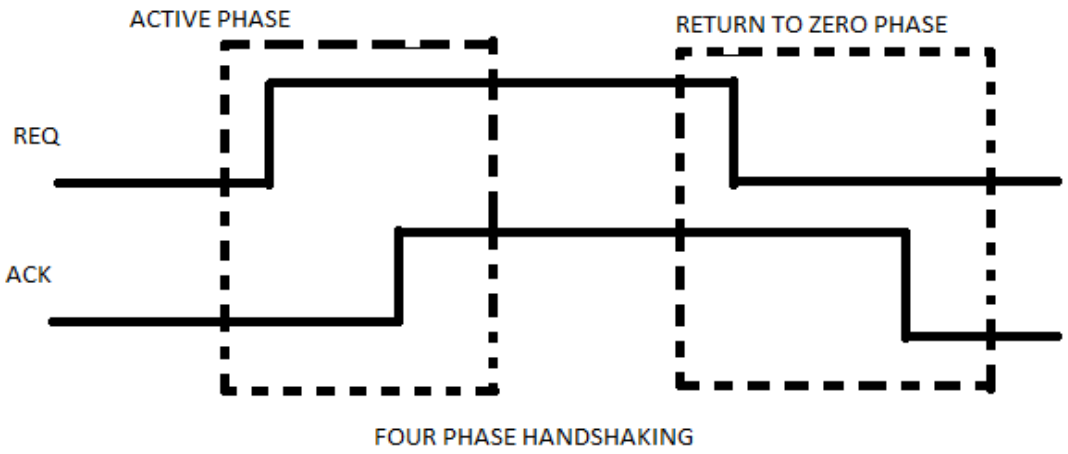

Fig 2.2: Four phase handshaking

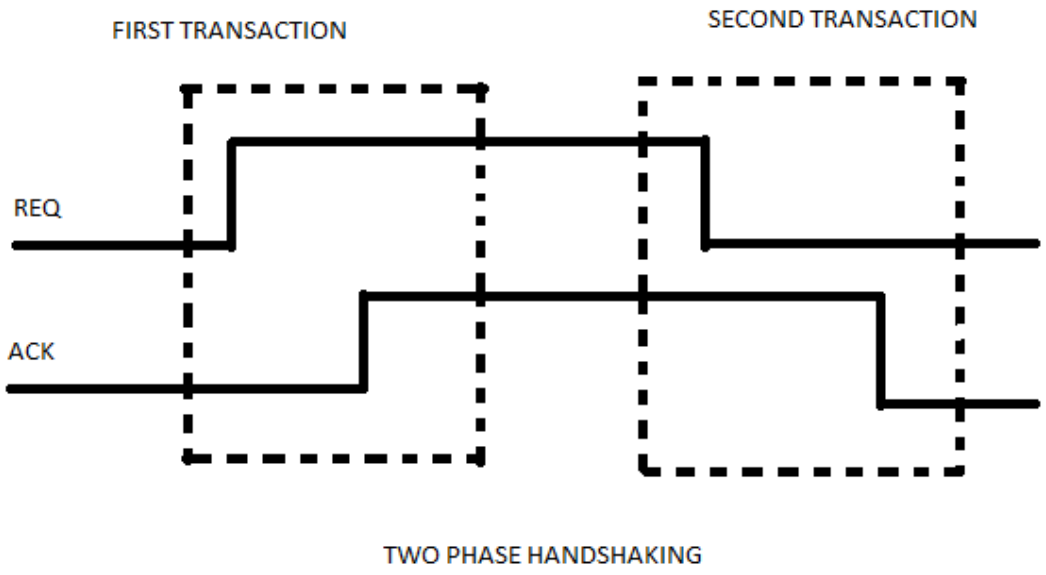

Fig 2.3: Two phase handshaking

Data communication is required after communication protocol for a channel has been fixed. The above figures exemplifies how the data itself replaces the single req wire. There are two common data encoding schemes: 1) delay-insensitive (DI) codes, and 2) single-rail bundled data [1].

2.11 SYNCHRONIZATION AND ARBITRATION

Synchronization and arbitration are the two capabilities required when dealing with continuous time operation for an asynchronous system. The consolidation of asynchronous and synchronous circuits or unrelated two synchronous systems is needed for synchronization. In second case, a synchronous signal must realign to a clock domain at the boundary crossing. The setup time violation leading to metastable operation and possible failure can occur as a result of direct connection of asynchronous inputs to synchronous registers [17] [1].

Arbitration involves requesting of a shared resource by resoling of two or more competing signals. At the beginning of every clock cycle, current requests are examined and in case of synchronous design, one of them is chosen. However, in case of asynchronous design, inputs keep on coming continuously and hence, resolution ought to be ensured to be clean and safe, irrespective of the arrival time of signal [1].

2.12 NOC

Network-on-chips (NoCs) have become the axiomatic standard approach for structured on-chip communication over the last decade for low power embedded systems and high performance chip multi-processors. On-chip networks can supplant traditional, specific bus-based communication with packet switching and can aim at various cost functions (fault tolerance, power, latency, saturation throughput, quality of service) and parameters (network topology, channel width, routinh strategies) [1].

The infrastructure of communication and related timings are separated by NOC approach. Hence, it is a natural choice for an asynchronous methodology. An asynchronous interconnect gives enhanced footing for robust timing, scalability and low power. Hence, the need for global clock management is eliminated [1].

The development of neuromorphic chips is an upcoming area in which asynchronous and GALS NoCs have played a key role.

2.13 GALS

A hybrid approach integrating synchronous components with an asynchronous communication network which form a globally asynchronous locally-synchronous (GALS) system is another option to replace full asynchronous systems [1].

This application brings the best of both worlds which makes it possible to design the reuse of synchronous intellectual property blocks while combining them with flexible asynchronous

interconnect as a global integrative medium. If rigid rate global clock on the communication network is removed, it will be possible to easily assemble the complicated systems with scalable, low-power and robust mechanism [1].

2.14 HAZARDS AND LOGIC SYNTHESIS AND OPTIMIZATION

Logically, an event-driven paradigm is implied due to absence of a fixed-rate clock, where components get activated and compute as soon as they get new inputs. A clear signal should reflect from each component once valid data has been produced. Hazard is potential for a glitch [18]. The hazard free optimal implementations is the goal of asynchronous design [19]. Hazards are temporal phenomena defined with respect to input transition, with change in value for one or more input signals. A static logic hazard occurs due to glitch in place of stable output. A dynamic logic hazard occurs due to glitch in place of clean transition [19].

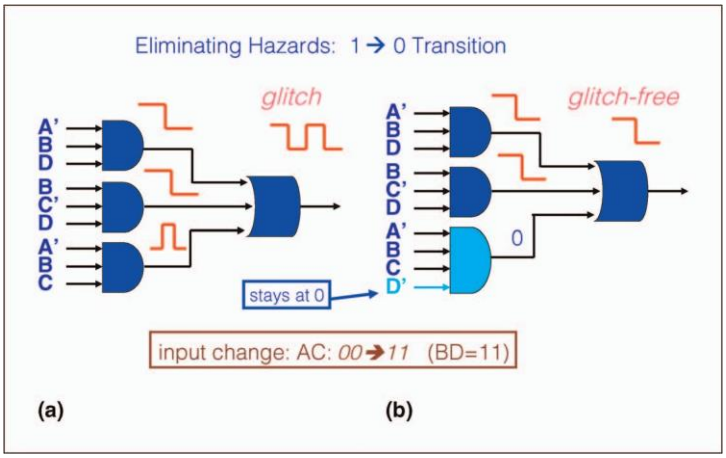

Fig 2.4: Eliminating glitches

Figure 2.4a presents a dynamic logic hazard for a given set of input transition, where circuit output has transition from 1 to 0. The output glitch appears based on relative timing of input signals. The circuit in Figure 2.4b represents identical functionality but is hazard-free for given input transition due to slight change in signals provided to bottom AND gate. The output transition is a clean one from 1 to 0 [19].

2.15 SPECIFICATION LANGUAGES AND TOOL FLOWS

The provision of entry hardware description language and automated computer aided design tool flows gives practical use of an asynchronous design. There have been advances on both the fronts. The two competing needs are to provide congeniality with current synchronous specification language and CAD tool flows and to design language that secures the fine-grain concurrency, distributed synchronization and clock less paradigm of asynchronous circuits. The research agenda is to resolve the balance the two [19].

2.16 SYSTEMC AND ITS NEED

With time-to-market pressure and ever increasing complexity of SoC, the abstraction level in the design flow has risen to the system level. So far, pin level hardware description was needed for modelling architectures. A great of work is needed for designing as well as verifying the models. Simulating them at pin level is sluggish. So the ideal selection to solve the issue is system level designing. The system level functionality is advantageous in many ways such as early software development, early functional verification and higher system quality. At system level, the level of detail required by the designer is only modelled to develop system and sub-system components [20].

The SystemC is put together as a library extending C++ which supports simultaneous behavior, the concept of operation which are timed and sequential, data types which describe hardware and structure hierarchy [21]. The macros, for instance, modules and channels, are provided by the core language for modeling the basic components of the system. The reuse is facilitated and IP transfer is made possible by introducing modularity in design due to encapsulation of C++ and its inheritance capabilities [22]. Further functionality is provided by various libraries. One such library called TLM (Transaction Level Modelling) describes protocols and channelizes and systemize the construction of high-level modules where single transaction replaces intricate communication and protocols. The discussed factors have made SystemC a de facto industry wide standard modeling language [3].

SystemC can deal with both hardware and software components and permits the designer to unite intricate electronic systems. The behavior is simulated and observed to check whether performance objectives are met. SystemC can be compiled using C++ compiler as it is based on standard C++.
The capability to simultaneously simulate a large number of tasks is required by simulation of a system in software. This issue is addressed by provision mechanism for simulating parallel execution in SystemC.

An event-driven simulation kernel is provided by the base layer of SystemC which controls processes in model in an abstract manner. In modules of SystemC,, functionality and data can be included into individual entities. They remain inaccessible to other components of system until they are exposed [23].

CHAPTER 3

DESIGN OF EXPERIMENT

3.1 SYNCHRONOUS PROCESSOR

3.1.1 BLOCK DIAGRAM OF SYNCHRONOUS PROCESSOR

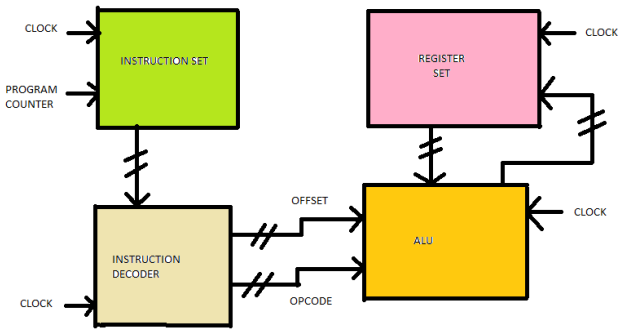

Fig 3.1: Block diagram of synchronous processor

3.1.2 DESCRIPTION OF SYNCHRONOUS PROCESSOR

The synchronous processor consists of four modules – Register set, Instruction set, Instruction decoder and ALU (Arithmetic and logical unit). The register set consists of eight registers each eight bits wide. The address range for the registers is from 0b000 to 0b111. The first five registers store data for the computation by ALU. The rest of the three registers are for storing the result of computation from ALU. The register set has write enable and ALU has read enable bit to ensure correct functionality. The instruction set consists of sixteen instructions each sixteen bits wide. The first four bits indicate the opcode to determine what operation has to be performed by ALU. The instruction either computes on two registers from register set or computes on a register from register set and an offset allocated by user. In some instructions, there are redundant bits which are set to zero for sake of computation. The execution of these instructions is depends on the instruction which is pointed to by the programming counter. The instruction decoder takes on the instruction from instruction set one by one based on the position of program counter and the clock signal. The first four bits are used to determine the computation to be performed in the ALU. The ALU receives opcode

from the instruction decoder based on the clock and determines which functional unit has to be used for computation. The result of computation is stored in the ALU registers which are passed onto the registers in the register set based on the clock.

3.2 ASYNCHRONOUS PROCESSOR

3.2.1 BLOCK DIAGRAM OF ASYNCHRONOUS PROCESSOR

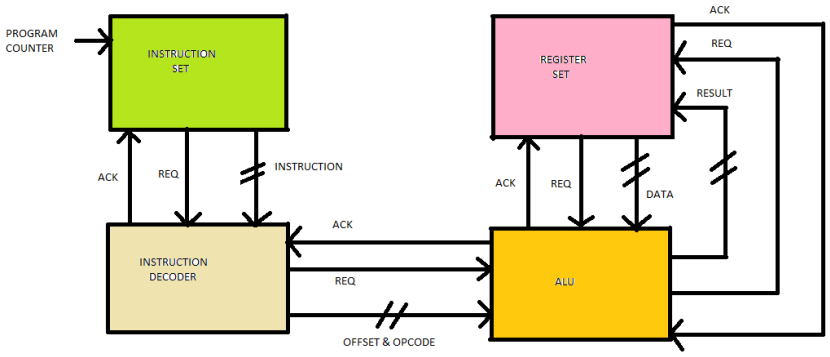

Fig 3.2: Block diagram of asynchronous processor

3.2.2 DESCRIPTION OF ASYNCHRONOUS PROCESSOR

The asynchronous processor also consists of four modules – Register set, Instruction set, Instruction decoder and ALU (Arithmetic and Logical Unit). The register set consists of eight registers each eight bits wide with address ranging from 0b000 to 0b111. The register set interacts with ALU through handshaking mechanism. The first five registers store data for computation by ALU and the rest three registers store the result of computation from the ALU registers. In both the cases, a handshake mechanism is applied with the initiator sending ready signal to the target and then transmitting the data to target on receiving the acknowledge signal from the target. The instructions in the instruction set are sixteen bits wide and sixteen in number. The instructions are passed onto the instruction decoder based on the handshaking mechanism and whether the program counter is pointing to that instruction. The instruction decoder isolates opcode to determine which computation has to be performed in ALU. The same information is sent to ALU using handshake mechanism. The computation is performed either on two register values or on a register value and an offset determined

by the user. The result of computation is stored in the register set using hand shaking mechanism.

3.3 POINTS COMMON TO ASYNCHRONOUS AND SYNCHRONOUS METHODOLOGIES

Program counter keeps track of instruction to be executed currently.

The clock in synchronous processor is required for moving from one instruction execution to next instruction execution.

In case of asynchronous processor, a two way handshake is implemented. Consider one initiator block and one target block. When the initiator block is ready to communicate, it sends the ready signal to the target block. In response, when the target block is ready, it sends a acknowledge signal back to initiator block. Then, the data is transferred from the initiator block to target block.

The opcode used in this explanation is four bits wide and hence, can provide a maximum of 16 different instructions. Based on the decoding of the opcode from instruction in instruction decoder provided by instruction set, three types of functioning can be performed by Arithmetic Logic Unit (ALU). These types include arithmetic operation, logical operation and move operation.

The immediate offset can be set manually in test benches written for both synchronous processor and asynchronous processor. Here, the offset used is 5 bits wide.

The ALU destination registers are all 8 bits wide which store the result of arithmetic or logical or move functionality computation.

The register set consists of eight registers. Each register is eight bits wide. The first five are for storing input data for ALU computations. The rest three registers are used for storing output data from ALU computations.

*The idea for four bit opcode is inspired from references [24] and [25].

CHAPTER 4

DETAILS OF INSTRUCTIONS IN THE INSTRUCTION SET

FIRST INSTRUCTION

ADD C, A, B {4'b0000, 2'b00, 4'b1001, 3'b001, 3'b010}

Opcode = 0000 [15:12]

Destination register – C [1001]

Source register – A [001] & B [010]

Here, all registers have 3 bit address. Each register can hold 8 bits wide data. Add operation is performed on two register values in ALU and stored back in destination register.

SECOND INSTRUCTION

ADD C, A, Offset {4'b0001, 4'b1001, 3'b001, 5'b10101}

Opcode = 0001 [15:12]

Destination register – C [1001]

Source register – A [001]

Offset – Offset [10101]

Here, both registers have 3 bit address. Each register can hold 8 bits wide data. Offset is user defined and is five bits wide. Add operation is performed on a register and offset in ALU and the result is stored back in destination register.

THIRD INSTRUCTION

MOV A, B {4'b0010, 5'b00000, 4'b1000, 3'b011}

Opcode = 0010 [15:12]

Destination register – A [1000]

Source register – B [011]

Here, both registers have 3 bit address. Each register can hold 8 bits wide data. The data is moved from a source register to a destination register.

FOURTH INSTRUCTION

NOT A, B {4'b0011, 5'b00000, 4'b1010, 3'b011}

Opcode = 0011 [15:12]

Destination register – A [1010]

Source register – B [011]

Here, both registers have 3 bit address. Each register can hold 8 bits wide data. The inverted bits of a source register are transferred to a destination register.

FIFTH INSTRUCTION

AND C, A, B {4'b0100, 2'b00, 4'b1010, 3'b011, 3'b100}

Opcode = 0100 [15:12]

Destination register – C [1010]

Source registers – A [011] & B [100]

Here, all registers have 3 bit address. Each register can hold 8 bits wide data. The logical AND operation is performed on two source registers and stored in the destination register.

SIXTH INSTRUCTION

AND C, A, Offset {4'b0101, 4'b1010, 3'b011, 5'b10101}

Opcode = 0101 [15:12]

Destination register – C [1010]

Source registers – A [011]

Offset – Offset [10101]

Here, both registers have 3 bit address. Each register can hold 8 bits wide data. Offset is user defined and is five bits wide. The logical AND operation is performed on a source register and an offset inside ALU and stored in a destination register.

SEVENTH INSTRUCTION

OR C, A, B {4'b0110, 2'b00, 4'b1010, 3'b011, 3'b100}

Opcode = 0110 [15:12]

Destination register – C [1010]

Source registers – A [011] & B [100]

Here, all registers have 3 bit address. Each register can hold 8 bits wide data. The logical OR operation is performed on two source registers and stored in the destination register.

EIGHTH INSTRUCTION

OR C, A, Offset {4'b0111, 4'b1010, 3'b011, 5'b10101}

Opcode = 0111 [15:12]

Destination register – C [1010]

Source registers – A [011]

Offset – Offset [10101]

Here, both registers have 3 bit address. Each register can hold 8 bits wide data. Offset is user defined and is five bits wide. The logical OR operation is performed on a source register and an offset inside ALU and stored in a destination register.

NINTH INSTRUCTION

XOR C, A, B {4'b1000, 2'b00, 4'b1010, 3'b011, 3'b100}

Opcode = 1000 [15:12]

Destination register – C [1010]

Source registers – A [011] & B [100]

Here, all registers have 3 bit address. Each register can hold 8 bits wide data. The logical XOR operation is performed on two source registers and stored in the destination register.

TENTH INSTRUCTION

XOR C, A, Offset {4'b1001, 4'b1010, 3'b011, 5'b10101}

Opcode = 1001 [15:12]

Destination register – C [1010]

Source registers – A [011]

Offset – Offset [10101]

Here, both registers have 3 bit address. Each register can hold 8 bits wide data. Offset is user defined and is five bits wide. The logical XOR operation is performed on a source register and an offset inside ALU and stored in a destination register.

ELEVENTH INSTRUCTION

NAND C, A, B {4'b1010, 2'b00, 4'b1010, 3'b011, 3'b100}

Opcode = 1010 [15:12]

Destination register – C [1010]

Source registers – A [011] & B [100]

Here, all registers have 3 bit address. Each register can hold 8 bits wide data. The logical NAND operation is performed on two source registers and stored in the destination register.

TWELFTH INSTRUCTION

NAND C, A, Offset {4'b1011, 4'b1010, 3'b011, 5'b10101}

Opcode = 1011 [15:12]

Destination register – C [1010]

Source registers – A [011]

Offset – Offset [10101]

Here, both registers have 3 bit address. Each register can hold 8 bits wide data. Offset is user defined and is five bits wide. The logical NAND operation is performed on a source register and an offset inside ALU and stored in a destination register.

THIRTEENTH INSTRUCTION

NOR C, A, B {4'b1100, 2'b00, 4'b1010, 3'b011, 3'b100}

Opcode = 1100 [15:12]

Destination register – C [1010]

Source registers – A [011] & B [100]

Here, all registers have 3 bit address. Each register can hold 8 bits wide data. The logical NOR operation is performed on two source registers and stored in the destination register.

FOURTEENTH INSTRUCTION

NOR C, A, Offset {4'b1101, 4'b1010, 3'b011, 5'b10101}

Opcode = 1101 [15:12]

Destination register – C [1010]

Source registers – A [011]

Offset – Offset [10101]

Here, both registers have 3 bit address. Each register can hold 8 bits wide data. Offset is user defined and is five bits wide. The logical NOR operation is performed on a source register and an offset inside ALU and stored in a destination register.

FIFTEENTH INSTRUCTION

XNOR C, A, B {4'b1110, 2'b00, 4'b1010, 3'b011, 3'b100}

Opcode =1110 [15:12]

Destination register – C [1010]

Source registers – A [011] & B [100]

Here, all registers have 3 bit address. Each register can hold 8 bits wide data. The logical XNOR operation is performed on two source registers and stored in the destination register.

SIXTEENTH INSTRUCTION

XNOR C, A, Offset {4'b1111, 4'b1010, 3'b011, 5'b10101}

Opcode = 1111 [15:12]

Destination register – C [1010]

Source registers – A [011]

Offset – Offset [10101]

Here, both registers have 3 bit address. Each register can hold 8 bits wide data. Offset is user defined and is five bits wide. The logical XNOR operation is performed on a source register and an offset inside ALU and stored in a destination register.

CHAPTER 5

SYSTEMC CODE FOR SYNCHRONOUS PROCESSOR

i_set,h

```
#include "systemc.h"

SC_MODULE(i_set){
    sc_inout <bool> enable;
    sc_in <bool> clk;
    sc_inout <int> prog_cnt;
    sc_out <sc_bv<16>> inst;

    void choose_inst();

    SC_CTOR(i_set){
        SC_METHOD(choose_inst)
        sensitive << clk.pos();
    }

};
```

i_set.cpp

```
#include"i_set.h"
#include<conio.h>

void i_set::choose_inst(){
    if (enable == 1 && clk.read() == 1){
        if(prog_cnt == 0){
            inst = "0000001001001010";
        }
        else if (prog_cnt == 1){
            inst = "0001100100110101";
        }
        else if (prog_cnt == 2){
            inst = "0010000001000000";
        }
        else if (prog_cnt == 3){
            inst = "0011000001010011";
        }
        else if (prog_cnt == 4){
            inst = "0100001010011100";
        }
        else if (prog_cnt == 5){
            inst = "0101101001110101";
        }
        else if (prog_cnt == 6){
            inst = "0110001010011100";
```

```
}
        else if (prog_cnt == 7){
            inst = "0111101001110101";
        }
        else if (prog_cnt == 8){
            inst = "1000001010011100";
        }
        else if (prog_cnt == 9){
            inst = "1001101001110101";
        }
        else if (prog_cnt == 10){
            inst = "1010001010011100";
        }
        else if (prog_cnt == 11){
            inst = "1011101001110101";
        }
        else if (prog_cnt == 12){
            inst = "1100001010011100";
        }
        else if (prog_cnt == 13){
            inst = "1101101001110101";
        }
        else if (prog_cnt == 14){
            inst = "1110001010011100";
        }
        else if (prog_cnt == 15){
            inst = "1111101001110101";
        }
        prog_cnt = prog_cnt + 1;
        if (prog_cnt == 16){
            cout << "The time taken is " << sc_time_stamp();
            getch();
            exit(0);
        }
        enable = 0;
    }
}
```

i_dec.h

```
#include"systemc.h"

SC_MODULE(i_dec){
    sc_inout <sc_bv<4>> opcode_dec;
    sc_out <sc_bv<5>> offset_dec;
    sc_inout <bool> enable;
    sc_inout <bool> enable_dec;
    sc_in <sc_bv<16>> inst_id;
    sc_in <bool> clk;
void choose_opcode();
```

```
    SC_CTOR(i_dec){
        SC_METHOD(choose_opcode)
            sensitive << clk.pos();

    }
};
```

i_dec.cpp

```
#include"i_dec.h"

void i_dec::choose_opcode(){
    if (enable == 0 && enable_dec == 1 && clk.read() == 1){
        sc_bv<16> id = inst_id.read();
        opcode_dec = id.range(15, 12);
        if (opcode_dec.read() == "0001" || opcode_dec.read() == "0101" ||
opcode_dec.read() == "0111" || opcode_dec.read() == "1001"
            || opcode_dec.read() == "1011" || opcode_dec.read() == "1101" ||
opcode_dec.read() == "1111"){
            offset_dec.write(id.range(4, 0));
        }
        enable_dec = 0;
    }
}
```

alu.h

```
#include"systemc.h"

SC_MODULE(alu){
    sc_out <sc_int<8>> mov_alu;
    sc_out <sc_int<8>> add_alu;
    sc_out <sc_int<8>> log_alu;
    sc_inout <bool> enable;
    sc_inout <bool> enable_dec;
    sc_inout <bool>enable_alu;
    sc_in <sc_bv<4>> opcode;
    sc_in <sc_bv<5>> offset;
    sc_in <sc_int<8>> reg_0;
    sc_in <sc_int<8>> reg_1;
    sc_in <sc_int<8>> reg_2;
    sc_in <sc_int<8>> reg_3;
    sc_in <sc_int<8>> reg_4;
    sc_in <bool> clk;
    sc_in <bool> re_alu;

    void destbasedonopcode();
    SC_CTOR(alu){
```

```
        SC_METHOD(destbasedonopcode);
        sensitive << clk.pos();
    }
};
```

alu.cpp

```
#include"alu.h"

void alu::destbasedonopcode(){
    if (clk.read() == 1 && re_alu == 1 && enable == 0 && enable_dec == 0 &&
enable_alu == 1){
        sc_bv<4> op = opcode.read();
        if (op == "0000"){
            add_alu.write(reg_1.read() + reg_2.read());
        }
        else if (op == "0001"){
            add_alu.write(reg_1.read() + reg_2.read());
        }
        else if (op == "0010"){
            mov_alu.write(reg_0.read());
        }
        else if (op == "0011"){
            log_alu.write(~reg_3.read());
        }
        else if (op == "0100"){
            log_alu.write(reg_3.read() & reg_4.read());
        }
        else if (op == "0101"){
            log_alu.write(reg_3.read() & offset.read());
        }
        else if (op == "0110"){
            log_alu.write(reg_3.read() | reg_4.read());
        }
        else if (op == "0111"){
            log_alu.write(reg_3.read() | offset.read());
        }
        else if (op == "1000"){
            log_alu.write(reg_3.read() ^ reg_4.read());
        }
        else if (op == "1001"){
            log_alu.write(reg_3.read() ^ offset.read());
        }
        else if (op == "1010"){
            log_alu.write(~(reg_3.read() & reg_4.read()));
        }
        else if (op == "1011"){
            log_alu.write(~(reg_3.read() & offset.read()));
        }
        else if (op == "1100"){
```

```
        log_alu.write(~(reg_3.read() | reg_4.read()));
        }
        else if (op == "1101"){
            log_alu.write(~(reg_3.read() | offset.read()));
        }
        else if (op == "1110"){
            log_alu.write(~(reg_3.read() ^ reg_4.read()));
        }
        else if (op == "1111"){
            log_alu.write(~(reg_3.read() ^ offset.read()));
        }
        enable_alu = 0;
    }
}
```

r_set.h

```
#include"systemc.h"

SC_MODULE(r_set){
    sc_out<sc_bv<8>> reg_0;
    sc_out<sc_bv<8>> reg_1;
    sc_out<sc_bv<8>> reg_2;
    sc_inout <bool> enable;
    sc_inout <bool> enable_dec;
    sc_inout <bool> enable_alu;
    sc_inout <bool> enable_rs;
    sc_in <bool> clk;
    sc_in <sc_bv<8>> m_reg;
    sc_in <sc_bv<8>> a_reg;
    sc_in <sc_bv<8>> l_reg;
    sc_in <bool> wr_enable;

    void store_result();

    SC_CTOR(r_set){
        SC_METHOD(store_result);
        sensitive << clk.pos();
    }
};
```

r_set.cpp

```
#include"r_set.h"

void r_set::store_result(){
    if (clk.read() == 1 && enable_rs == 1 && enable == 0 && enable_dec == 0 &&
enable_alu == 0 && wr_enable == 1){
```

```
    reg_0 = m_reg;
        reg_1 = a_reg;
        reg_2 = l_reg;
        enable = 1;
        enable_dec = 1;
        enable_alu = 1;
        enable_rs = 1;
    }
}
```

wclk_tb.h

```
#include"systemc.h"

SC_MODULE(wclk_tb){
    sc_in <bool> clk_tb;
    sc_inout <bool> enable_tb;
    sc_inout <bool> enable_dec_tb;
    sc_inout <bool> enable_alu_tb;
    sc_inout <bool> enable_rs_tb;
    sc_inout <int> prog_cnt_tb;
    sc_in <sc_bv<16>> inst_tb;
    sc_inout <sc_bv<4>> opcode_dec_tb;
    sc_in <sc_bv<5>> offset_dec_tb;
    sc_out <sc_bv<16>> inst_id_tb;
    sc_in <sc_int<8>> mov_alu_tb;
    sc_in <sc_int<8>> add_alu_tb;
    sc_in <sc_int<8>> log_alu_tb;
    sc_out <sc_bv<4>> opcode_tb;
    sc_out <sc_bv<5>> offset_tb;
    sc_out <sc_int<8>> reg_0_tb;
    sc_out <sc_int<8>> reg_1_tb;
    sc_out <sc_int<8>> reg_2_tb;
    sc_out <sc_int<8>> reg_3_tb;
    sc_out <sc_int<8>> reg_4_tb;
    sc_out <bool> re_alu_tb;
    sc_in<sc_bv<8>> reg_00_tb;
    sc_in<sc_bv<8>> reg_01_tb;
    sc_in<sc_bv<8>> reg_02_tb;
    sc_out <sc_bv<8>> m_reg_tb;
    sc_out <sc_bv<8>> a_reg_tb;
    sc_out <sc_bv<8>> l_reg_tb;
    sc_out <bool> wr_enable_tb;

    void initialize();

    SC_CTOR(wclk_tb){
        SC_THREAD(initialize);
    }
```

```
};
```

wclk_tb.cpp

```
#include"wclk_tb.h"

void wclk_tb::initialize(){
    enable_tb.write(1);
    enable_dec_tb.write(1);
    enable_alu_tb.write(1);
    enable_rs_tb.write(1);
    wr_enable_tb.write(1);
    re_alu_tb.write(1);
    prog_cnt_tb.write(0);
}
```

main.cpp

```
#include<systemc.h>
#include"wclk_tb.h"
#include"i_set.h"
#include"i_dec.h"
#include"alu.h"
#include"r_set.h"

int sc_main(int argc, char* argv[]){

    sc_clock clk_sig("clk_sig", 1, 0.5);
    sc_signal <bool> enable_sig;
    sc_signal <bool> enable_dec_sig;
    sc_signal <bool> enable_alu_sig;
    sc_signal <bool> enable_rs_sig;
    sc_signal <int> prog_cnt_sig;
    sc_signal <sc_bv<16>> inst_sig;
    sc_signal <sc_bv<4>> opcode_dec_sig;
    sc_signal <sc_bv<5>> offset_dec_sig;
    sc_signal <sc_bv<16>> inst_id_sig;
    sc_signal <sc_int<8>> mov_alu_sig;
    sc_signal <sc_int<8>> add_alu_sig;
    sc_signal <sc_int<8>> log_alu_sig;
    sc_signal <sc_bv<4>> opcode_sig;
    sc_signal <sc_bv<5>> offset_sig;
    sc_signal <sc_int<8>> reg_0_sig;
    sc_signal <sc_int<8>> reg_1_sig;
    sc_signal <sc_int<8>> reg_2_sig;
    sc_signal <sc_int<8>> reg_3_sig;
    sc_signal <sc_int<8>> reg_4_sig;
    sc_signal <bool> re_alu_sig;
```

```
sc_signal <sc_bv<8>> reg_00_sig;
sc_signal <sc_bv<8>> reg_01_sig;
sc_signal <sc_bv<8>> reg_02_sig;
sc_signal <sc_bv<8>> m_reg_sig;
sc_signal <sc_bv<8>> a_reg_sig;
sc_signal <sc_bv<8>> l_reg_sig;
sc_signal <bool> wr_enable_sig;

i_set is1("i_set_1");
i_dec id1("i_dec_1");
alu alu1("alu_1");
r_set rs1("r_set_1");
wclk_tb wclktb1("wclk_tb_1");

wclktb1.clk_tb(clk_sig);
wclktb1.enable_tb(enable_sig);
wclktb1.enable_dec_tb(enable_dec_sig);
wclktb1.enable_alu_tb(enable_alu_sig);
wclktb1.enable_rs_tb(enable_rs_sig);
wclktb1.prog_cnt_tb(prog_cnt_sig);
wclktb1.inst_tb(inst_sig);
wclktb1.opcode_dec_tb(opcode_dec_sig);
wclktb1.offset_dec_tb(offset_dec_sig);
wclktb1.inst_id_tb(inst_id_sig);
wclktb1.mov_alu_tb(mov_alu_sig);
wclktb1.add_alu_tb(add_alu_sig);
wclktb1.log_alu_tb(log_alu_sig);
wclktb1.opcode_tb(opcode_sig);
wclktb1.offset_tb(offset_sig);
wclktb1.reg_0_tb(reg_0_sig);
wclktb1.reg_1_tb(reg_1_sig);
wclktb1.reg_2_tb(reg_2_sig);
wclktb1.reg_3_tb(reg_3_sig);
wclktb1.reg_4_tb(reg_4_sig);
wclktb1.re_alu_tb(re_alu_sig);
wclktb1.reg_00_tb(reg_00_sig);
wclktb1.reg_01_tb(reg_01_sig);
wclktb1.reg_02_tb(reg_02_sig);
wclktb1.m_reg_tb(m_reg_sig);
wclktb1.a_reg_tb(a_reg_sig);
wclktb1.l_reg_tb(l_reg_sig);
wclktb1.wr_enable_tb(wr_enable_sig);

is1.enable(enable_sig);
is1.clk(clk_sig);
is1.prog_cnt(prog_cnt_sig);
is1.inst(inst_sig);

id1.opcode_dec(opcode_sig);
id1.offset_dec(offset_sig);
id1.enable(enable_sig);
id1.enable_dec(enable_dec_sig);
```

```
    id1.inst_id(inst_sig);
    id1.clk(clk_sig);

    alu1.mov_alu(mov_alu_sig);
    alu1.add_alu(add_alu_sig);
    alu1.log_alu(log_alu_sig);
    alu1.enable(enable_sig);
    alu1.enable_dec(enable_dec_sig);
    alu1.enable_alu(enable_alu_sig);
    alu1.opcode(opcode_sig);
    alu1.offset(offset_sig);
    alu1.reg_0(reg_0_sig);
    alu1.reg_1(reg_1_sig);
    alu1.reg_2(reg_2_sig);
    alu1.reg_3(reg_3_sig);
    alu1.reg_4(reg_4_sig);
    alu1.re_alu(re_alu_sig);
    alu1.clk(clk_sig);

    rs1.reg_0(reg_00_sig);
    rs1.reg_1(reg_01_sig);
    rs1.reg_2(reg_02_sig);
    rs1.enable(enable_sig);
    rs1.enable_dec(enable_dec_sig);
    rs1.enable_alu(enable_alu_sig);
    rs1.enable_rs(enable_rs_sig);
    rs1.clk(clk_sig);
    rs1.m_reg(m_reg_sig);
    rs1.a_reg(a_reg_sig);
    rs1.l_reg(l_reg_sig);
    rs1.wr_enable(wr_enable_sig);

    sc_start();
    return 0;
}
```

CHAPTER 6

SYSTEMC CODE FOR ASYNCHRONOUS PROCESSOR

i_set.h

```
#include"systemc.h"

SC_MODULE(i_set){
    sc_inout <bool> rdy_istoid;
    sc_inout <bool> ack_istoid;
    sc_inout <int> prog_cnt;
    sc_out <sc_bv<16>> inst;
    sc_in <bool> clk;
    int count = 0;

    void choose_inst();
    void wait_func();

    SC_CTOR(i_set){
        SC_METHOD(choose_inst);
        sensitive << rdy_istoid << ack_istoid << prog_cnt;
    }
};
```

i_set.cpp

```
#include"i_set.h"
#include<conio.h>


void i_set::choose_inst(){
    if (rdy_istoid.read() == 1 && ack_istoid.read() == 1){
        count = count + 1;
        if (prog_cnt == 0){
            inst = "0000001001001010";
            cout << "The t begin time is " << count;

        }
        else if (prog_cnt == 1){
            inst = "0001100100110101";
            cout << "The t2 time is " << count;
        }
        else if (prog_cnt == 2){
            inst = "0010000001000000";
            cout << "The t3 time is " << count;
        }
        else if (prog_cnt == 3){
            inst = "0011000001010011";
        }
```

```
    else if (prog_cnt == 4){
            inst = "0100001010011100";
        }
        else if (prog_cnt == 5){
            inst = "0101101001110101";
        }
        else if (prog_cnt == 6){
            inst = "0110001010011100";
        }
        else if (prog_cnt == 7){
            inst = "0111101001110101";
        }
        else if (prog_cnt == 8){
            inst = "1000001010011100";
        }
        else if (prog_cnt == 9){
            inst = "1001101001110101";
        }
        else if (prog_cnt == 10){
            inst = "1010001010011100";
        }
        else if (prog_cnt == 11){
            inst = "1011101001110101";
        }
        else if (prog_cnt == 12){
            inst = "1100001010011100";
        }
        else if (prog_cnt == 13){
            inst = "1101101001110101";
        }
        else if (prog_cnt == 14){
            inst = "1110001010011100";
        }
        else if (prog_cnt == 15){
            inst = "1111101001110101";
            cout << "the 15 time is " << count;
        }
        prog_cnt = prog_cnt + 1;
        rdy_istoid = 0;
        ack_istoid = 0;
        if (prog_cnt == 16){
            cout << "The time taken is " << count;
            getch();
            exit(0);
        }
    }
}
```

i_dec.h

```
#include"systemc.h"

SC_MODULE(i_dec){
    sc_inout <bool> rdy_istoid;
    sc_inout <bool> ack_istoid;
    sc_in <sc_bv<16>> inst_dec;
    sc_out <sc_bv<4>> opcode_dec;
    sc_out <sc_bv<5>> offset_dec;
    sc_inout <bool> rdy_idtoalu;
    sc_inout <bool> ack_idtoalu;
    sc_in<bool> clk;

    void choose_opcode();

    SC_CTOR(i_dec){
        SC_METHOD(choose_opcode);
        sensitive << rdy_idtoalu << ack_idtoalu << rdy_istoid << ack_istoid;
    }
};
```

i_dec.cpp

```
#include"i_dec.h"

void i_dec::choose_opcode(){
    if (rdy_idtoalu.read() == 1 && rdy_istoid.read() == 0){
        if (ack_idtoalu.read() == 1 && ack_istoid.read() == 0){
            sc_bv<16> id = inst_dec.read();
            opcode_dec = id.range(15, 12);
            if (opcode_dec.read() == "0001" || opcode_dec.read() == "0101" ||
opcode_dec.read() == "0111" || opcode_dec.read() == "1001"
                || opcode_dec.read() == "1011" || opcode_dec.read() == "1101"
|| opcode_dec.read() == "1111"){
                offset_dec.write(id.range(4, 0));
            }
            ack_idtoalu = 0;
            rdy_idtoalu = 0;
        }
    }
}
```

alu.h

```
#include"systemc.h"

SC_MODULE(alu){
    sc_out <sc_int<8>> mov_alu;
    sc_out <sc_int<8>> add_alu;
    sc_out <sc_int<8>> log_alu;
    sc_inout <bool> rdy_rstoalu;
    sc_inout <bool> ack_rstoalu;
    sc_inout <bool> rdy_idtoalu;
    sc_inout <bool> ack_idtoalu;
    sc_inout <bool> rdy_istoid;
    sc_inout <bool> ack_istoid;
    sc_in <sc_bv<4>> opcode;
    sc_in <sc_bv<5>> offset;
    sc_in <sc_int<8>> reg_0;
    sc_in <sc_int<8>> reg_1;
    sc_in <sc_int<8>> reg_2;
    sc_in <sc_int<8>> reg_3;
    sc_in <sc_int<8>> reg_4;
    sc_in <bool> clk;

    void destbasedonopcode();

    SC_CTOR(alu){
        SC_METHOD(destbasedonopcode);
        sensitive << rdy_idtoalu << ack_idtoalu << rdy_rstoalu << ack_rstoalu;
    }

};
```

alu.cpp

```
#include"alu.h"

void alu::destbasedonopcode(){

    if (rdy_idtoalu.read() == 0 && rdy_rstoalu.read() == 1){
        if (ack_idtoalu.read() == 0 && ack_rstoalu.read() == 1){
            sc_bv<4> op = opcode.read();
            if (op == "0000"){
                add_alu.write(reg_1.read() + reg_2.read());
            }
            else if (op == "0001"){
                add_alu.write(reg_1.read() + reg_2.read());
            }
            else if (op == "0010"){
                mov_alu.write(reg_0.read());
            }
            else if (op == "0011"){
```

```
                log_alu.write(~reg_3.read());
            }
            else if (op == "0100"){
                log_alu.write(reg_3.read() & reg_4.read());
            }
            else if (op == "0101"){
                log_alu.write(reg_3.read() & offset.read());
            }
            else if (op == "0110"){
                log_alu.write(reg_3.read() | reg_4.read());
            }
            else if (op == "0111"){
                log_alu.write(reg_3.read() | offset.read());
            }
            else if (op == "1000"){
                log_alu.write(reg_3.read() ^ reg_4.read());
            }
            else if (op == "1001"){
                log_alu.write(reg_3.read() ^ offset.read());
            }
            else if (op == "1010"){
                log_alu.write(~(reg_3.read() & reg_4.read()));
            }
            else if (op == "1011"){
                log_alu.write(~(reg_3.read() & offset.read()));
            }
            else if (op == "1100"){
                log_alu.write(~(reg_3.read() | reg_4.read()));
            }
            else if (op == "1101"){
                log_alu.write(~(reg_3.read() | offset.read()));
            }
            else if (op == "1110"){
                log_alu.write(~(reg_3.read() ^ reg_4.read()));
            }
            else if (op == "1111"){
                log_alu.write(~(reg_3.read() ^ offset.read()));
            }
            rdy_rstoalu = 0;
            ack_rstoalu = 0;
        }
    }
}
```

r_set.h

```
#include"systemc.h"

SC_MODULE(r_set){
    sc_out<sc_bv<8>> reg_00;
    sc_out<sc_bv<8>> reg_01;
    sc_out<sc_bv<8>> reg_02;
```

```
sc_inout <bool> rdy_istoid;
    sc_inout <bool> ack_istoid;
    sc_inout <bool> rdy_idtoalu;
    sc_inout <bool> ack_idtoalu;
    sc_inout <bool> rdy_alutors;
    sc_inout <bool> ack_alutors;
    sc_inout <bool> rdy_rstoalu;
    sc_inout <bool> ack_rstoalu;
    sc_in <sc_bv<8>> m_reg;
    sc_in <sc_bv<8>> a_reg;
    sc_in <sc_bv<8>> l_reg;
    sc_in <bool> clk;

    void store_result();

    SC_CTOR(r_set){
        SC_METHOD(store_result);
        sensitive << rdy_alutors << rdy_rstoalu << ack_alutors << ack_rstoalu;
    }
};
```

r_set.cpp

```
#include"r_set.h"

void r_set::store_result(){
    if (rdy_alutors.read() == 1 && rdy_istoid.read() == 0 && rdy_idtoalu.read()
== 0 && rdy_rstoalu.read() == 0){
        if (ack_alutors.read() == 1 && ack_istoid.read() == 0 &&
ack_idtoalu.read() == 0 && ack_rstoalu.read() == 0){
            //next_trigger(1, SC_NS);
            reg_00 = m_reg;
            reg_01 = a_reg;
            reg_02 = l_reg;
            rdy_istoid = 1;
            rdy_idtoalu = 1;
            rdy_rstoalu = 1;
            ack_istoid = 1;
            ack_idtoalu = 1;
            ack_rstoalu = 1;
        }
    }
}
```

woclk.h

```
#include"systemc.h"

SC_MODULE(woclk){
    sc_inout <bool> rdy_istoid_tb;
    sc_inout <bool> ack_istoid_tb;
    sc_inout <int> prog_cnt_tb;
    sc_in <sc_bv<16>> inst_tb;

    sc_out <sc_bv<16>> inst_dec_tb;
    sc_in <sc_bv<4>> opcode_dec_tb;
    sc_in <sc_bv<5>> offset_dec_tb;
    sc_inout <bool> rdy_idtoalu_tb;
    sc_inout <bool> ack_idtoalu_tb;

    sc_in <sc_int<8>> mov_alu_tb;
    sc_in <sc_int<8>> add_alu_tb;
    sc_in <sc_int<8>> log_alu_tb;
    sc_inout <bool> rdy_rstoalu_tb;
    sc_inout <bool> ack_rstoalu_tb;
    sc_out <sc_bv<4>> opcode_tb;
    sc_out <sc_bv<5>> offset_tb;
    sc_out <sc_int<8>> reg_0_tb;
    sc_out <sc_int<8>> reg_1_tb;
    sc_out <sc_int<8>> reg_2_tb;
    sc_out <sc_int<8>> reg_3_tb;
    sc_out <sc_int<8>> reg_4_tb;

    sc_in<sc_bv<8>> reg_00_tb;
    sc_in<sc_bv<8>> reg_01_tb;
    sc_in<sc_bv<8>> reg_02_tb;
    sc_inout <bool> rdy_alutors_tb;
    sc_inout <bool> ack_alutors_tb;
    sc_out <sc_bv<8>> m_reg_tb;
    sc_out <sc_bv<8>> a_reg_tb;
    sc_out <sc_bv<8>> l_reg_tb;
    sc_in <bool> clk_tb;

    void initialize();

    SC_CTOR(woclk){
        SC_THREAD(initialize);
    }

};
```

woclk.cpp

```
#include"woclk.h"
#include"i_set.h"
#include"conio.h"


void woclk::initialize(){
    rdy_istoid_tb.write(1);
    ack_istoid_tb.write(1);
    rdy_idtoalu_tb.write(1);
    ack_idtoalu_tb.write(1);
    rdy_alutors_tb.write(1);
    ack_alutors_tb.write(1);
    rdy_rstoalu_tb.write(1);
    ack_rstoalu_tb.write(1);
    prog_cnt_tb.write(0);

}
```

main.cpp

```
#include"systemc.h"
#include"i_set.h"
#include"i_dec.h"
#include"alu.h"
#include"r_set.h"
#include"woclk.h"

int sc_main(int argc, char* argv[]){
    sc_signal <bool> rdy_istoid_sig;
    sc_signal <bool> ack_istoid_sig;
    sc_signal <int> prog_cnt_sig;
    sc_signal <sc_bv<16>> inst_sig;

    sc_signal <sc_bv<16>> inst_dec_sig;
    sc_signal <sc_bv<4>> opcode_dec_sig;
    sc_signal <sc_bv<5>> offset_dec_sig;
    sc_signal <bool> rdy_idtoalu_sig;
    sc_signal <bool> ack_idtoalu_sig;

    sc_signal <sc_int<8>> mov_alu_sig;
    sc_signal <sc_int<8>> add_alu_sig;
    sc_signal <sc_int<8>> log_alu_sig;
    sc_signal <bool> rdy_rstoalu_sig;
    sc_signal <bool> ack_rstoalu_sig;
    sc_signal <sc_bv<4>> opcode_sig;
    sc_signal <sc_bv<5>> offset_sig;
    sc_signal <sc_int<8>> reg_0_sig;
    sc_signal <sc_int<8>> reg_1_sig;
```

```
sc_signal <sc_int<8>> reg_2_sig;
    sc_signal <sc_int<8>> reg_3_sig;
    sc_signal <sc_int<8>> reg_4_sig;

    sc_signal <sc_bv<8>> reg_00_sig;
    sc_signal <sc_bv<8>> reg_01_sig;
    sc_signal <sc_bv<8>> reg_02_sig;
    sc_signal <bool> rdy_alutors_sig;
    sc_signal <bool> ack_alutors_sig;
    sc_signal <sc_bv<8>> m_reg_sig;
    sc_signal <sc_bv<8>> a_reg_sig;
    sc_signal <sc_bv<8>> l_reg_sig;
    sc_clock clk_sig("clk_sig", 1, 0.5);

    i_set is1("i_set_1");
    i_dec id1("i_dec_1");
    alu alu1("alu_1");
    r_set rs1("r_set_1");
    woclk woclk1("woclk_1");

    woclk1.rdy_istoid_tb(rdy_istoid_sig);
    woclk1.ack_istoid_tb(ack_istoid_sig);
    woclk1.prog_cnt_tb(prog_cnt_sig);
    woclk1.inst_tb(inst_sig);
    woclk1.clk_tb(clk_sig);

    woclk1.inst_dec_tb(inst_dec_sig);
    woclk1.opcode_dec_tb(opcode_dec_sig);
    woclk1.offset_dec_tb(offset_dec_sig);
    woclk1.rdy_idtoalu_tb(rdy_idtoalu_sig);
    woclk1.ack_idtoalu_tb(ack_idtoalu_sig);

    woclk1.mov_alu_tb(mov_alu_sig);
    woclk1.add_alu_tb(add_alu_sig);
    woclk1.log_alu_tb(log_alu_sig);
    woclk1.rdy_rstoalu_tb(rdy_rstoalu_sig);
    woclk1.ack_rstoalu_tb(ack_rstoalu_sig);
    woclk1.opcode_tb(opcode_sig);
    woclk1.offset_tb(offset_sig);
    woclk1.reg_0_tb(reg_0_sig);
    woclk1.reg_1_tb(reg_1_sig);
    woclk1.reg_2_tb(reg_2_sig);
    woclk1.reg_3_tb(reg_3_sig);
    woclk1.reg_4_tb(reg_4_sig);

    woclk1.reg_00_tb(reg_00_sig);
    woclk1.reg_01_tb(reg_01_sig);
    woclk1.reg_02_tb(reg_02_sig);
    woclk1.rdy_alutors_tb(rdy_alutors_sig);
    woclk1.ack_alutors_tb(ack_alutors_sig);
    woclk1.m_reg_tb(m_reg_sig);
    woclk1.a_reg_tb(a_reg_sig);
    woclk1.l_reg_tb(l_reg_sig);
```

```
    is1.rdy_istoid(rdy_istoid_sig);
    is1.ack_istoid(ack_istoid_sig);
    is1.prog_cnt(prog_cnt_sig);
    is1.inst(inst_sig);
    is1.clk(clk_sig);

    id1.rdy_istoid(rdy_istoid_sig);
    id1.ack_istoid(ack_istoid_sig);
    id1.inst_dec(inst_dec_sig);
    id1.opcode_dec(opcode_dec_sig);
    id1.offset_dec(offset_dec_sig);
    id1.rdy_idtoalu(rdy_idtoalu_sig);
    id1.ack_idtoalu(ack_idtoalu_sig);
    id1.clk(clk_sig);

    alu1.mov_alu(mov_alu_sig);
    alu1.add_alu(add_alu_sig);
    alu1.log_alu(log_alu_sig);
    alu1.rdy_rstoalu(rdy_rstoalu_sig);
    alu1.ack_rstoalu(ack_rstoalu_sig);
    alu1.rdy_idtoalu(rdy_idtoalu_sig);
    alu1.ack_idtoalu(ack_idtoalu_sig);
    alu1.rdy_istoid(rdy_istoid_sig);
    alu1.ack_istoid(ack_istoid_sig);
//  alu1.rdy_alutors(rdy_alutors_sig);
//  alu1.ack_alutors(ack_alutors_sig);
    alu1.opcode(opcode_sig);
    alu1.offset(offset_sig);
    alu1.reg_0(reg_0_sig);
    alu1.reg_1(reg_1_sig);
    alu1.reg_2(reg_2_sig);
    alu1.reg_3(reg_3_sig);
    alu1.reg_4(reg_4_sig);
    alu1.clk(clk_sig);

    rs1.reg_00(reg_00_sig);
    rs1.reg_01(reg_01_sig);
    rs1.reg_02(reg_02_sig);
    rs1.rdy_istoid(rdy_istoid_sig);
    rs1.ack_istoid(ack_istoid_sig);
    rs1.rdy_idtoalu(rdy_idtoalu_sig);
    rs1.ack_idtoalu(ack_idtoalu_sig);
    rs1.rdy_alutors(rdy_alutors_sig);
    rs1.ack_alutors(ack_alutors_sig);
    rs1.rdy_rstoalu(rdy_rstoalu_sig);
    rs1.ack_rstoalu(ack_rstoalu_sig);
    rs1.m_reg(m_reg_sig);
    rs1.a_reg(a_reg_sig);
    rs1.l_reg(l_reg_sig);
    rs1.clk(clk_sig);

    sc_start();
```

```
return 0;
}
```

CHAPTER 7

COMPARISON OF PERFORMANCE BETWEEN BOTH THE PROCESSORS

The time taken by synchronous processor to complete computation is 64 ns.

The time taken by asynchronous processor to complete computation is 16 ns.

*Here the time taken for handshake per module is considered to be lower than one clock cycle and hence, the total time taken is considered to be equal to one time unit (one clock period) for all four modules for asynchronous processor.

CHAPTER 8

CONCLUSION AND SCOPE OF FUTURE WORKS

It is clear from the result presented that asynchronous processor has better performance than synchronous processor. The future work includes comparison of power consumed using synchronous and asynchronous methodologies to further substantiate the advantage of asynchronous processors over synchronous counterparts.

REFERENCES AND BIBLIOGRAPHY

1 S. M. Nowick (Columbia University), M. Singh (University of North Carolina), "Asynchronous Design - Part 1: Overview and Recent Advances", co-published by *IEEE CEDA, IEEE CASS, IEEE SSCS and TTTC,* May – June, 2015.

2 M. H. Oh, Y. W. Kim, S. Kwak, C. H. Shin, S. N. Kim, "Architectural design issues in a clockless 32 bit processor using an asynchronous HDL", *ETRI Journal,* Volume 35, Number 3, June 2013.

3 D. Geer, "Is it time for clockless chips?", IEEE computer society.

4 R. A. Jain, D. V. Padole, "Design of GALS system based scalable heterogeneous multi-core system", *IJETT,* September 2015, volume 2, issue 2.

5 C.H. Van Bekel et al., "Applications of Asynchronous Circuits," *Proc. IEEE,* vol. 87, no. 2, Feb. 1999, pp. 223-233.

6 C.J. Myers, *Asynchronous Circuit Design,* NY: John Wiley & Sons, Inc., July 2001.

7 J. Kessels and R. Marston, "Designing Asynchronous Standby Circuits for a Low-Power Pager," *Proc. IEEE,* vol. 87, no. 2, Feb. 1999, pp. 257-267.

8 B.Z. Tang et al., "A Low Power Asynchronous GPS Baseband Processor," *Proc. IEEE 18th Int. Symp. Asynchronous Circuits Syst.,* 2012, pp. 33-40.

9 S. Bo et al., "Reducing Power Consumption of Floating-Point Multiplier via Asynchronous Technique," *Proc. 4th Int. Conf. Comput. Inf. Sci.,* Aug. 2012, pp. 1360-1363.

10 M. Davies et al., ''A 72-Port 10G Ethernet switch/router using quasi-delay-insensitive asynchronous design'', *Proc. Int. Symp. Asynch. Circuits Syst.* (ASYNC 14), 2014, pp. 103–104.

11 J. Teifel and R. Manohar, ''Highly pipelined asynchronous FPGAs,'' in *Proc. ACM/SIGDA Int. Symp.* Field Programmable Gate Arrays (FPGA 04), 2004, pp. 133–142

12 T. E. Williams and M. A. Horowitz, ''A zero-overhead self-timed 160 ns 54 b CMOS divider,'' *IEEE J. Solid-State Circuits,* vol. 26, no. 11, pp. 1651–1661, 1991

13 M. Singh et al., ''An adaptively pipelined mixed synchronous-asynchronous digital FIR filter chip operating at 1.3 gigahertz,'' *IEEE Trans. Very Large Scale Integration. (VLSI) Syst.,* vol. 18, no. 7, pp. 1043–1056, 2010

14 I. E. Sutherland, ''Micropipelines,'' *Communication. ACM,* vol. 32, no. 6, pp. 720–738, 1989.

15 M. Singh and S. M. Nowick, ''MOUSETRAP: High-speed transition-signaling asynchronous pipelines,'' *IEEE Trans. Very Large Scale Integration. (VLSI) Syst.,* vol. 15, no. 6, pp. 684–698, 2007.

16 S. M. Nowick and M. Singh, ''High-performance asynchronous pipelines: An overview,'' *IEEE Design & Test,* vol. 28, no. 5, pp. 8–22, 2011.

17 R. Ginosar, ''Metastability and synchronization: A tutorial,'' *IEEE Design & Test,* vol. 28, no. 5,

pp. 23–35, 2011.

18 S. M. Nowick, D. L. Dill, ‘‘Exact two-level minimization of hazard-free logic with multiple-input changes,’’ *IEEE Trans. Computer.-Aided Design Integration Circuits Syst.,* vol. 14, no. 8, pp. 986–997, 1995.

19 S. M. Nowick(Columbia University), M. Singh(University of North Carolina), “Asynchronous Design Part 2: Systems and Methodologies”, *co-published by IEEE CEDA, IEEE CASS, IEEE SSCS and TTTC,* May – June, 2015.

20 T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2004.

21 H. Moinudeen, A. Habibi, S. Tahar, Department of Electrical and Computer Engineering Concordia University 1455 de Maisonneuve, West, Montreal, Quebec H3G 1M8. *Model Based Verification of SystemC Designs.*

22 A. Bunker, G. Gopalakrishnan, and S. A. McKee. (2004, Jan). “ Formal Hardware Specification Languages for Protocol Compliance Verification”. *ACM Transactions on Design Autom. of Elec. Sys.,* 9(1):1–32.

23 D. Tabakov, Schlumberger Information Solutions 5599 San Felipe Str. Houston, TX, USA. Moshe Y. Vardi. Rice University 6100 Main Str. MS-132 Houston, TX, USA. *Automatic Aspectization of SystemC.*

24 S. Bhavsar, A. Rao, A. Sen, R. Joshi. A 16-bit MIPS Based Instruction Set Architecture for RISC processor. *International Journal of Scientific and Research Publications,* Volume 3, Issue 4, April 2013.

25 The *RISC-16 Instruction Set Architecture* by Bruce Jacob, Digital Computer Design, Fall 2000.