NTNU – Trondheim
Norwegian University of
Science and Technology

Low Power Capacitive Touch Digital Detection Filter

A Comparative Study of Synchronous and
Asynchronous Methodologies

Truls Magnus Aamodt Gulbrandsen

Master of Science in Electronics
Submission date: June 2012
Supervisor: Snorre Aunet, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

In this assignment, the student should make a capacitive touch digital detection filter circuit. Two implementations of the circuit should be made, one using traditional synchronous methods and one using asynchronous methods. The student should evaluate and compare the methods used for implementing the circuit. Both implementations must be functionally verified. In addition, the student should compare the two implementations with regards to power consumption, emission and implementation cost.

Assignment given: 16. January 2012
Internal Supervisor: Snorre Aunet, IET, NTNU
External Supervisor: Kristoffer E. Koch, Atmel Norway AS

Abstract

In this thesis, both synchronous and asynchronous methodologies is explored for implementing a capacitive touch digital detection filter circuit. Asynchronous methodologies promise characteristics such as lower power, higher area cost and lower emission than synchronous methodologies. The aim of this thesis is to show if this can be exploited for this application.

The synchronous implementation is written in Verilog, and follows a standard synchronous design flow. The asynchronous implementation is written in Balsa, and follows a Balsa Asynchronous Synthesis System design flow. Both implementations have been synthesised to netlist. A simple clock tree was generated for the synchronous implementation. Both netlists was simulated with wire load models.

Netlist simulation of the synchronous and the asynchronous implementation shows that the power consumption is similar for the two implementations, because the fixed sample rate of the capacitance measurement operation dominates over the filter operations. The overhead from the handshake logic results in double the area for the asynchronous implementation. The asynchronous implementation has lower emission because of the randomness of the power consumption from the handshake circuits when the circuit is not sampling, while the synchronous implementation has large frequency components with harmonics from both clock flanks, resulting in higher emissions. Thus, asynchronous methodologies do not automatically lead to low power consumption, but can lead to larger area cost and lower emission.

In addtion, new approaches for interfacing an asynchronous circuit, described in Balsa, with an analog circuit, and implementing a variable speed sampler clock with a minimum fixed sample period has been found, but not implemented.

Sammendrag

I denne oppgaven har vi utforsket synkrone og asynkrone metoder for å implementere en kapasativberøring-digital-deteksjonsfilter-krets. Asynkrone metoder lover karakteristikker som lavere effekt, større areal og mindre elektromagnetisk emisjon enn synkrone metoder. Målet med denne oppgaven er å vise om dette kan utnyttes for denne kretsen.

Den synkrone implementasjonen er skrevet i Verilog, og følger en standard synkron designflyt. Den asynkrone implementasjonen er skrevet i Balsa, og følger en Balsa Asynchronous Synthesis System designflyt. Begge implementasjoner har blitt syntetisert til nettliste. Et simpelt klokketre har blitt generert for den synkrone implementasjonen. Begge implementasjoner har blitt simulert med banelastmodeller.

Nettlistesimulering av den synkrone og den asynkrone implementasjonen viser at effektforbruket er liknende for de to implementasjonene, fordi den faste samplingsraten til kapasitansmålingsoperasjonen dominerer over filter operasjonene. Overhead fra håndtrykklogikk resulterer i dobbelt så mye areal for den asynkrone implementasjonen. Den asynkrone implementasjonen stråler mindre på grunn av den randomiserte karakteristikken fra håndtrykklogikken når kretsen ikke utfører samplingsoperasjoner, mens den synkrone implementasjonen har store frekvenskomponenter med harmoniske fra begge klokkeflanker, noe som resulterer i mer elektromagnetisk stråling. Asynkrone metoder leder derfor ikke automatisk til lavere effektforbruk, men kan lede til større arealkostnad og mindre elektromagnetisk stråling.

I tillegg har vi funnet nye metoder for å interface en asynkron krets, beskrevet i Balsa, med en analog krets, samt en metode for å implementere en samplingsklokke med variabel fart og minimum periode, men ikke implementert dem.

Preface

This thesis was written in the period January to June 2012. Most of the work was carried out at Atmel Norway AS. This report builds on the work done in the specialisation project fall 2011. This report was written in LaTeX. This document was created using pdfTeX. BibTeX is used for references.

Acknowledgements

I want to thank the following for helping me or in other ways supporting my work with this master thesis:
Are Aarseth, for help with design flow, libraries and scripts.
Atmel Norway AS, for providing workspace and the necessary tools.
Bjarne Drotningshaug, for help with and providing custom Atmel technology library for Balsa synthesis.
Bjarte Løken Nystøyl, for general tips and great discussions about circuit design, and for keeping spirits high.
Kristoffer E. Koch, my supervisor from Atmel Norway AS.
Snorre Aunet, my supervisor from NTNU.
My family, for their support.
Milla, the rabbit, for cheering me up.

Truls Magnus Aamodt Gulbrandsen, June 2012, Skatval

Table of Contents

Problem Description **I**

Abstract **III**

Sammendrag **V**

Preface **VII**

Acknowledgements **IX**

List of Figures **XVII**

List of Tables **XIX**

Acronyms **XXI**

1 Introduction **1**

1.1 Specification . . . 3

1.1.1 Goals . . . 3

1.1.2 Design Constraints . . . 4

1.1.3 Structure and Functionality . . . 4

1.1.4 Technology . . . 4

1.1.5 Design Techniques . . . 5

1.1.6 Fabrication . . . 5

1.2 Outline of the Thesis . . . 5

2 Background **7**

2.1 Capacitive Sensing . . . 7

2.1.1 RC Circuit and Relaxation Oscillator . . . 7

2.1.2 Classification of Signal . . . 9

2.1.3 Analog to Digital Conversion . . . 9

2.2 Digital Filters . . . 9

2.2.1 Median-3 Filter . . . 9

2.2.2 Exponential Moving Average Filter . . . 10

2.3 Asynchronous Circuit Design . . . 10

2.3.1 Handshake Protocols . . . 10

Bundled Data Protocols 10
Bundled Data Channel Types 10
4-Phase Bundled Data Protocol 11
2-Phase Bundled Data Protocol 12
2.3.2 Data Validity Schemes 12
2.3.3 Muller-C Element 12
Gate Level Implementation 13
2.4 Clock Domain Crossing 14
2.4.1 Setup, Hold time and Metastability of Flop 14
Setup time 14
Hold time 15
Metastability 15
2.4.2 Mean Time Between Failures 15
2.5 Balsa Asynchronous Synthesis System 16
2.5.1 Balsa Design Flow 16
2.5.2 Data Typing Issues 16
Non-Delay-Insensitive Components 17

3 Implementation **19**
3.1 Top Module 19
3.1.1 Architecture 20
Channels 20
Behaviour of Top Module 20
3.2 Control Module 23
3.3 Register Bank 23
3.3.1 Registers 23
Synchronous Implementation 24
Asynchronous Implementation 24
3.4 Sampler Top Module 24
3.4.1 Synchronous Implementation of Sampler Top Module 24
Implementation 25
Synchronisation 26
3.4.2 Asynchronous Implementation of Sampler Top Module 26
Wrapper Module 27
3.5 Sampler Module 27
3.5.1 Synchronous Implementation of Sampler Module 27
3.5.2 Synchronous Implementation of Sense/Drive Output Ports . . 28
3.5.3 Asynchronous Implementation of Sampler Module 28
Alternative 1 29
Alternative 2 30
Alternative 3 31
Alternative 4 32
Implementation 33
3.5.4 Implementation of Sense/Drive Output Ports 33
Alternative 1 34
Alternative 2 34
Implementation 35
3.6 Median-3 Filter Top Module 35

3.7 Median-3 Filter Module 35
3.7.1 Registers 36
Register Storage 36
3.7.2 Median-3 Algorithm 36
Comparison 37
3.8 EMA Filter Top Module 37
3.9 EMA Filter Module 37
3.9.1 EMA Algorithm 38
3.9.2 Addition 38
3.9.3 Subtraction 38
3.9.4 Multiplication 39
Alternative 1 39
Alternative 2 40
Implementation 40
3.9.5 Synchronous Implementation of EMA Filter Module 40
Addition in Verilog 40
Subtraction in Verilog 40
3.9.6 Asynchronous Implementation of EMA Filter Module 40
Adder in Balsa 40
Subtraction 41
Multiplication in Balsa 43
3.10 Threshold Comparator Top Module 45
3.11 Threshold Comparator Module 45

4 Functional Verification 47
4.1 Method 47
4.2 Pad/RC Circuit Model 47
4.3 Testbench 48
4.4 Tests 48
4.4.1 Top Module Tests 49
Configuration Test 49
Sample, Filter and Threshold Comparison Test 49
4.4.2 Submodule Tests 50
4.5 Simulation Flow 51
4.6 Submodule Simulation 51
4.6.1 Synchronous Implementation - Submodule RTL Simulation . . 51
4.6.2 Asynchronous Implementation - Submodule Breeze Simulation 52
4.7 Top Module Simulation 52
4.7.1 Synchronous Implementation - Top Module RTL Simulation . . 52
4.7.2 Synchronous Implementation - Top Module NTL Simulation . 53
Sampling and Pad/RC Model Circuit 53
4.7.3 Asynchronous Implementation - Top Module NTL Simulation . 53

5 Synthesis 57
5.1 Synthesis of Synchronous Implementation 57
5.1.1 Flow 57
5.1.2 Synthesis 57
5.1.3 Quick P&R 59

5.2 Synthesis of Asynchronous Implementation 59

6 Time-Based Power Estimation 61

6.1 Script for Generating SDF for all corners 62
6.2 Script Sequence for Converting VPD to VCD 62
6.3 Script Sequence for Time Based Power Estimation 63

7 Results 65

7.1 Cell Area Cost 65
7.2 Power Consumption 65
7.2.1 Power Waveform 65
7.2.2 Power Distribution and Power Density 66
7.2.3 Average Total Power 67
7.2.4 Energy/sequence 67
7.2.5 Average Power In Active Mode 67
7.2.6 Peak Power 68
7.3 Emission 74

8 Discussion 75

8.1 Results 75
8.1.1 Power Consumption 75
8.1.2 Area Cost 75
8.1.3 Emission 76
8.2 Design Optimisation 76
8.2.1 Data Path 76
Reducing the Number Range 76
Dividing the Number Range 76
Recurring Structures 77
8.3 Verification 77
8.3.1 Simulation Time 77
8.3.2 Timing Violations 77

9 Conclusion 79

9.1 Conclusions Drawn from This Thesis 79
9.2 Summary of the Contributions this Thesis Has Made 79
9.3 Prospect of Further Research 80

Bibliography 81

A Balsa Code 83

A.1 Parameters 83
A.2 Modules 84
A.3 Verification Tests 99

B Verilog Code 103

B.1 Parameters 103
B.2 Synchronous Modules 104
B.3 Verification Tests 115
B.4 Pad/RC Circuit Model 123

C Python Code **125**

D Tools **129**

D.1 Common Tools . . . 129

D.2 Synchronous Flow Tools . . . 130

D.3 Asynchronous Flow Tools . . . 130

List of Figures

2.1 RC circuit. 8
2.2 RC oscillator. 8
2.3 The Four Fundamental Channel Types [9],p117 11
2.5 4-Phase Bundled Data Protocol [9] 12
2.6 2-Phase Bundled Data Protocol [9] 12
2.8 Muller-C Element 12
2.9 Truth Table 12
2.7 Data Validity Schemes for 2-phase and 4-phase Bundled Data 13
2.10 Two Flip-Flop Synchronizer 15
2.11 Balsa Design Flow. 17

3.2 Top Module Architecture 21
3.4 Start, Sample, Filter and Compare Sequence. 22
3.12 Reset Signal Synchronizer Module 26
3.14 Sampler Module Architecture 28
3.16 Simple Inverter Chain 30
3.18 Delayed Inverter Chain 30
3.19 Delay Element 30
3.20 Metastability 31
3.24 Handshake Components. 33
3.29 Median-3 Filter Module Architecture 36
3.34 EMA Filter Module Architecture 39
3.35 Breeze Component: Binary Function. 41
3.38 Shift-Add Algorithm 44
3.39 Shift-Add Architecture 44

4.1 Pad/RC Circuit Model. 48
4.2 Testbench. 48
4.3 Configuration Test Sequence 49
4.4 Typical Test Sequence 49
4.6 Sampler and Pad/RC Circuit Waveform. 55
4.7 Latch and BrzVariable Waveform. 56

5.1 Synchronous Synthesis Flow. 58
5.2 Physical Layout After Clock Tree Synthesis. 59

6.1 Time Based Power Estimation Flow. 62

7.4 Power Distribution. 66
7.5 Power Distribution. 66
7.8 Peak Power. 68
7.1 Synchronous Implementation - Power Consumption Waveform. 69
7.2 Asynchronous Implementation - Power Consumption Waveform. . . . 70
7.3 Total Power Comparison Waveform. 71
7.6 Power Density - Synchronous Implementation. 72
7.7 Power Density - Asynchronous Implementation. 73
7.9 DFT of Power - Synchronous Implementation. 74
7.10 DFT of Power - Asynchronous Implementation. 74

List of Tables

1.1 Power Consumption 5

3.1 Register Bank Map 23

7.1 Implementation Cost 65

Acronyms

APT Advanced Processor Technologies. 16

BD bundled data. 5, 16, 75, 77, 78, 80

CTS Clock Tree Synthesis. 58

DC Design Compiler. 58, 61
DFT Discrete Fourier Transform. 74, 76
DR dual-rail. 5, 78, 80

EMA exponential moving average. 4, 9, 10, 50, 77

FE First Encounter. 58
FSDB Fast Signal Database. 74
FSM Finite State Machine. 53

IIR infinite impulse response. 10

NTL netlist. 54, 80

P&R place & route. 57
PT PrimeTime. 61, 65, 74

QDI Quasi-Delay Insensitive. 16

RTL register transfer level. 19

SBPF Synopsys Binary Parasitics Format. 61
SDF Standard Delay Format. 61
SR single-rail. 16

Chapter 1

Introduction

[JP Morgan] expects 657m smartphones to be sold in 2012, up from 459m this year. - The Financial Times [4]

A variety of current smartphones use a combination of touch based and mechanical button interfaces. Mechanical buttons are used as a supplement to the touch based interface for implementing system functionality, but most importantly as a way to power up or wake up a device from standby mode.

The disadvantages of using a mechanical button are wear and tear from use, and the physical space needed for implementation. The advantage of using a mechanical button is that it does not consume any active power.

The advantages of a capacitive touch based interfaces is that it is robust, and that it e.g. can be incorporated into the screen of a smartphone. However, capacitive sensing is an active process, and therefore uses more power than the passive mechanical button.

If the mechanical buttons are going to be replaced by a purely capacitive touch based interface, the wake-up capabilities of the mechanical button must be incorporated into the capacitive touch based interface, without consuming too much active power. This can be achieved by combining the capacitive touch based interface with a dedicated low power capacitive touch detection filter circuit for implementing the wake-up functionality.

No current mobile devices use capacitive touch sensing for wake-up capabilities. However, capacitive touch sensing [14] is nothing new, and this application is analogous with capacitive sensing of a touch button.

For this thesis, two implementations of a captive touch digital detection filter circuit has been made, one using synchronous methodologies and one using asynchronous methodologies. The synchronous implementation is used as reference circuit which the asynchronous implementation can be compared to.

Traditional synchronous methodologies are known/common to designers, and known to give good results. Asynchronous methodologies are unknown/uncommon to most designers, but promises several advantages over synchronous methodologies. This is the motivation for exploring asynchronous methodologies. [9, p. 3-5] claims the advantages and the disadvantages of using asynchronous methodologies over traditional synchronous methodologies. A summary of the claims is shown in figure 1.1.

Advantages
+ Low power consumption.
+ High operating speed.
+ Less emission of electro-magnetic noise.
+ Robustness towards variations in supply voltage, temperature and fabrication process parameters.
+ Better composability and modularity.
+ No clock distribution and clock skew problems.

Disadvantages
- Handshake circuits lead to overhead in terms of area, speed and power consumption.
- Lack of CAD tools.

Figure 1.1

[9, p. 4] also notes that in order to achieve good results, the designer is required to be familiar/have much experience with asynchronous methodologies to make a good asynchronous implementation, and not end up with a circuit that performs worse than its synchronous counter-part. There are differences among application areas and asynchronous methodologies can only be exploited if the application at hand allows for it. Thus, the performance of an asynchronous circuit depends both on design choices and the application of the circuit.

For general purpose processors (e.g. the Amulet3i [11]) the performance (speed and power) has been shown to be similar between synchronous and asynchronous implementations. For some signal processing applications (e.g. a hearing aid [15] and a contactless smart card [9, p. 221-248]) an asynchronous implementation has shown lower power consumption than a synchronous implementation.

Both implementations of the capacitive touch digital detection filter have been simulated post-synthesis to get time based power estimation, emissions and area cell cost. Figure 1.2 shows the results from comparing the performance of the synchronous and the asynchronous.

- Double the area for the asynchronous implementation.
- Close in terms of dynamic power consumption.
- Less emission for the asynchronous implementation.

Figure 1.2

The overhead from the handshake logic results in twice the area for the asynchronous circuit. The sampling operation dominates the power consumption, which results in similar power consumption for the synchronous and asynchronous implementation. The lower emissions of the asynchronous implementation is because of the randomness of the power consumption from the handshake circuits when the circuit is not sampling. The synchronous implementation shows large frequency components with harmonics from both clock flanks, resulting in higher emissions.

Asynchronous circuit design is not something new [7]. However, an asynchronous implementation of a capacitive touch digital detection filter circuit is something completely new. There are no known records of an asynchronous implementation of a capacitive touch digital detection filter circuit in the public domain.

The reasons that anyone have not implemented a capacitive touch digital detection filter circuit using asynchronous methodologies before, may be that most designers in the industry are unfamiliar with or lack the training in use of asynchronous methodologies, and the lack of industry standard tools or design flow.

Therefore, it is important that research is done in the field of asynchronous circuit design to make it easier for designers to explore both synchronous and asynchronous solutions for a given application. The asynchronous implementation of a capacitive touch digital detection filter shows a practical application area, where

1.1 Specification

The following sections describe the requirements for the capacitive touch digital detection filter circuit.

1.1.1 Goals

The main goal for a capacitive touch digital detection filter circuit is low power and low implementation cost. However, due to the challenge of learning asynchronous methodologies and using immature tools for the asynchronous design flow, the main goals for this thesis are to learn different implementation methods using asynchronous methodologies and to complete the design flow for both implementations of the circuit. The circuit is fast enough to perform 16 sample, filter and threshold comparison sequences per second for the worst case run time, thus achieving good responsiveness for the touch application. In addition, both implementations is functionally verified and compared in terms of power consumption, implementation cost and emission.

1.1.2 Design Constraints

The sampler clock runs on the same frequency, $f_S = 10\text{MHz}$, for both implementations. The clock for the synchronous implementation runs on the frequency $f = 5\text{MHz}$. The number of switchings in the datapath for the synchronous implementation is not dependent on the clock frequency, and therefore the clock frequency of the circuit should not have much impact on the power consumption results.

The path of the design constraints file is *src/synch/standalone/synt/constraints.tcl.*

1.1.3 Structure and Functionality

Figure 1.3 shows a simplified flow for the four main modules in the capacitive touch digital detection filter circuit. The sampler module performs capacitance measurements. The median-3 filter module and the exponential moving average (EMA) filter module performs noise filtering and smoothing of the capacitance measurements. The threshold comparator module checks if the threshold for detecting a touch has been crossed.

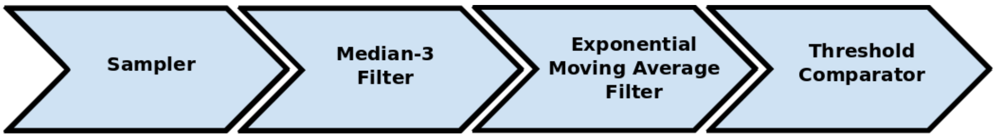


Figure 1.3

The capacitive touch digital detection filter circuit supports three commands from an external circuit, *Start*, *Write* and *Read*. The *Start* command starts a sample, filter and threshold comparison sequence. The *Write* command writes data to a configuration register. The *Read* command reads data from an internal register. The *Write* or the *Read* command can be issued when the circuit is performing a sample, filter and threshold comparison sequence.[1]

1.1.4 Technology

Both implementations uses the same proprietary 350nm technology library (*NDC35900L*), which in today's market is regarded as an old technology. This library is characterised as high voltage, high threshold and approximately 0[2] leakage, and it is area optimised for 350nm production. It should be noted that it has high emission[3].

The low leakage of this technology is ideal for a capacitive touch digital detection filter application, where the circuit is idle most of the time.

[1]This can result in a short halt of operation.

[2]Due to the very low leakage of this technology library, all leakage table entries are 0.

[3][12] shows how a synchronous microcontroller implemented in this technology can be used as an FM transmitter.

[5] shows how a Balsa handshake component library can be made for this technology library. This Balsa handshake component library supports synthesis of asynchronous circuits using 1-wire bundled data (BD) 2-phase and 4-phase, and 2-wire dual-rail (DR) protocol.

Table 1.1 shows the three corner cases from the technology library that is used to cover variations in temperature and voltage.

Corner	Voltage [V]	Temperature [°C]
Max	2.7	105
Typ	3.0	25
Min	3.6	-40

Table 1.1: Power Consumption

1.1.5 Design Techniques

Both the synchronous and the asynchronous implementation will use 2's complement number representation, because it is the default option in Verilog and Balsa.

To achieve low power consumption, the synchronous implementation of the circuit uses automatic clock gate insertion, while the asynchronous implementation relies on asynchronous methodologies.

The asynchronous implementation uses a 4-phase BD protocol, because the DR protocol uses more wires and switchings, and the 2-phase BD [15, p. 273-274] tend to use more area and be slower. Benchmarks in [5, p. 48] show that a 2-phase implementation can consume less power, but also notes that this result can be biased by less optimised handshake component implementations [5, p. 45].

1.1.6 Fabrication

Due to limited time and resources, the two implementations of the capacitive touch digital detection filter circuit will not be fabricated, only simulated.

1.2 Outline of the Thesis

Chapter 2 presents the background knowledge needed for understanding this thesis. Chapter 3 presents the architecture and methods for implementing the capacitive touch digital detection filter circuit. Chapter 4 presents the functional verification of the synchronous and asynchronous implementation of the circuit. Chapter 5 presents the synthesis of the synchronous and asynchronous implementation of the circuit. Chapter 7 presents the results from synthesis and power estimation. Chapter 8 discusses implementation methods, observations, possible optimisations and results. Chapter 9 presents conclusions, contributions and possible further research.

Chapter 2

Background

In order to appreciate this thesis to the full extent, it is necessary with some background knowledge of concepts such as capacitive sensing, digital-to-analog converters, digital filters, multi-clock domains, asynchronous circuit methodologies and power consumption in CMOS circuits. The following sections give a brief introduction to these concepts.

2.1 Capacitive Sensing

Capacitive sensing [14] is a technology based on capacitive coupling that is used in many different types of sensors. Capacitive sensors can detect anything that is conductive. E.g. a capacitive sensor can be used to detect and measure the touch or proximity of a human hand. A capacitive sensor is very robust due to its lack of mechanical components.

There are two types of capacitive sensing systems; mutual capacitance and self capacitance. Mutual capacitance sensing is when the object (finger, conductive stylus etc.) alters the mutual coupling between row and column electrodes, which are scanned sequentially. Self capacitance sensing is when the object loads the sensor or increases the parasitic capacitance to ground.

2.1.1 RC Circuit and Relaxation Oscillator

Capacitance is typically measured indirectly, e.g. by using it to control the frequency of an oscillator. The design of a capacitance meter can be based on a relaxation oscillator. The capacitance to be sensed forms a portion of the oscillator's RC circuit.

A relaxation oscillator works by storing and dissipating the energy in the capacitor in an RC circuit repeatedly to setup the oscillations. The output of the IC is driven to the supply voltage to charge the capacitor, and driven to ground to discharge the capacitor.

Figure 2.1 from [19] shows how an RC circuit can be combined with a microcontroller to create an relaxation oscillator circuit.

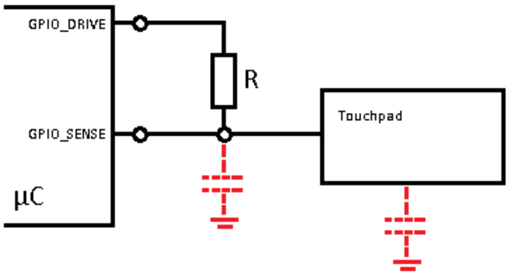


Figure 2.1: RC circuit.

Figure 2.2 from [19] shows the oscillations generated from the microcontroller driven relaxation oscillator circuit. The green line shows the drive voltage, while the yellow line shows the voltage over the capacitance in the RC circuit.

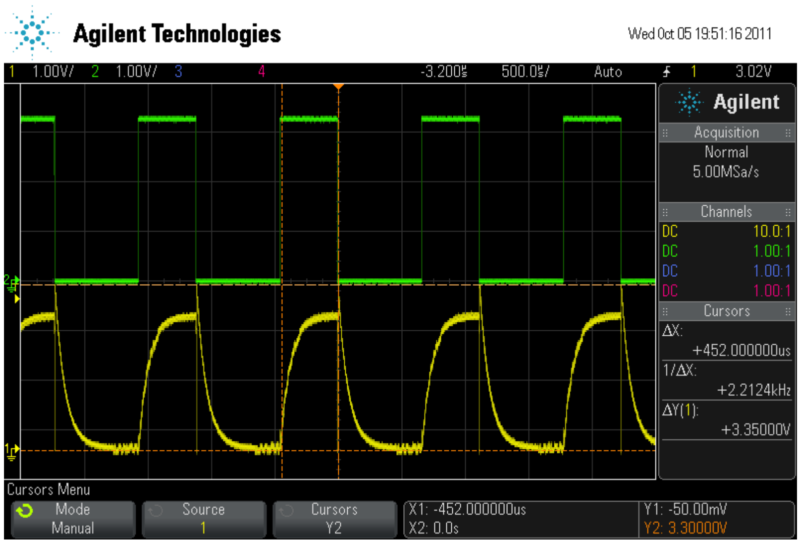


Figure 2.2: RC oscillator.

2.1.2 Classification of Signal

The voltage drop over the capacitance in the RC circuit seen on $sense_{in}$ is a one-dimensional real valued continuous analog signal. When charging the capacitor, the signal can be described by the function:

$$V_C(t) = V_{CC} * (1 - e^{\frac{t}{RC}}) \tag{2.1}$$

When discharging the capacitor, the signal can be described by the function:

$$V_C(t) = V_{CC} * e^{\frac{t}{RC}} \tag{2.2}$$

2.1.3 Analog to Digital Conversion

The voltage drop over the capacitance in the RC circuit can be sampled over one charge and one discharge period. This doubles the precision of the sampling. The noise component of the signal will make it difficult to set a threshold for detecting a *touch*. Therefore a set of digital filters is needed to remove this noise component.

2.2 Digital Filters

A combination of a median-3 filter and an EMA filter has been selected for removing the majority of the noise component, thus easing the task of setting a threshold for detecting a *touch*. The filters are described in sections 2.2.1 and 2.2.2.

2.2.1 Median-3 Filter

A median filter is a non-linear digital filter. It is useful for suppressing impulse noise. A median-3 filter is a median filter with window length $N = 3$. Experiments with filter lengths in [19] shows that this is adequate. This is the minimum length of a median filter, and has the lowest computational cost for a median filter. However, since a median filter is non-linear, the algorithm has a generally high computational cost. The median-3 algorithm takes the current sample and the two previous samples, sorts the values and picks the median value. This can be achieved with a simple bubblesort [18, p. 40] algorithm. If on average one of the three samples is a noise spike, most noise will be filtered. If the filtered signal was very noisy, with an average of more than one out of three noise spikes, the filter length could be increased to compensate.

2.2.2 Exponential Moving Average Filter

An EMA filter is a hybrid infinite impulse response (IIR) [10, p. 196] filter. It is useful for smoothing signals. It uses a weighted moving average function, where the weighting factors of the filter decrease exponentially. The advantage of an EMA filter is that it only needs to store the current and the previous value, and a constant α factor.

Equation 2.3 shows the formula for calculating the EMA.

$$\begin{aligned} \alpha &= \frac{2}{N+1} \\ EMA_i &= EMA_{i-1} + \alpha * (MED_i - EMA_{i-1}) \end{aligned} \tag{2.3}$$

2.3 Asynchronous Circuit Design

Sections 2.3.1, 2.3.2 and 2.3.3 give a short description of handshake protocols, data validity schemes and the Muller-C element. For more information on the fundamentals about asynchronous circuit design, the reader is referred to [9, p. 5-28].

2.3.1 Handshake Protocols

In an asynchronous circuit the clock signal is replaced with handshaking between neighbouring registers. Asynchronous circuits are controlled by locally derived clock pulses that can occur at any time. [9]

Bundled Data Protocols

The term *bundled data* refers to a situation where the data signals use normal Boolean levels to encode information, and where separate request and acknowledge wires are bundled with the data signals.

Bundled Data Channel Types

There are four fundamental channel types - non-put, push, pull and bi-put channel. The non-put channel is a dataless channel used for synchronisation. The push channel is a channel where the sender initiates the transfer of data from the sender to the receiver. The pull channel is a channel where the receiver initiates the transfer. The bi-put channel is a channel where the receiver communicates data with the acknowledge signal. Figure 2.3 shows the four fundamental channel types.

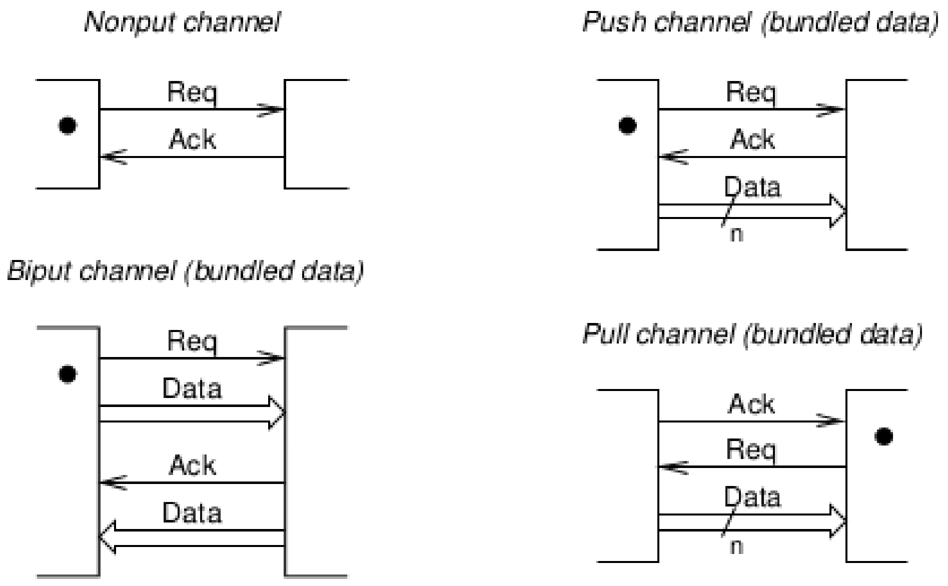

Nonput channel
Req
Ack
Push channel (bundled data)
Req
Ack
Data
n
Biput channel (bundled data)
Req
Data
Ack
Data
Pull channel (bundled data)
Ack
Req
Data
n

Figure 2.3: The Four Fundamental Channel Types [9],p117

4-Phase Bundled Data Protocol

In the 4-phase protocol illustrated in figure 2.5 the request and acknowledge wires also use normal Boolean levels to encode information. The term *4-phase* refers to the number of communication actions, as shown in figure 2.4. The 4-phase protocol has

1. The sender issues data and sets request high.
2. The receiver absorbs the data and sets acknowledge high.
3. The sender responds by taking request low (at which point data is no longer guaranteed to be valid).
4. The receiver acknowledges this by taking acknowledge low.
5/1. The sender may initiate the next communication cycle.

Figure 2.4

a disadvantage over the 2-phase protocol in the return-to-zero transitions that cost unnecessary time and energy.

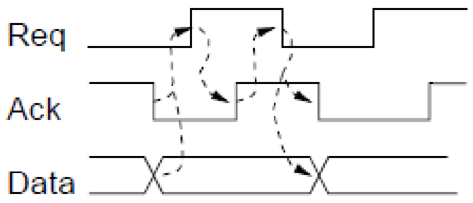


Figure 2.5: 4-Phase Bundled Data Protocol [9]

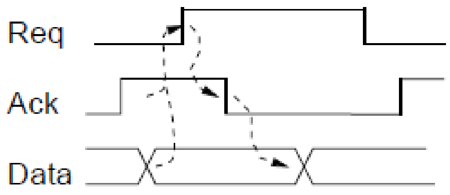


Figure 2.6: 2-Phase Bundled Data Protocol [9]

2-Phase Bundled Data Protocol

In the 2-phase protocol illustrated in figure 2.6 the information on the request and acknowledge wires is encoded as signal transitions on the wires. There is no difference between a *0* $\rightarrow$ *1* and a *1* $\rightarrow$ *0* transition, they both represent a *signal event*. The implementation of the 2-phase protocol is more complex than the 4-phase protocol, so even though it seems faster because of using less transitions, there is no general answer to which is better.

2.3.2 Data Validity Schemes

A data validity scheme [9, p. 116] defines the time interval in which data is valid. Figure 2.7 from [9, p. 117] shows the different possible schemes for the bundled data protocol.

2.3.3 Muller-C Element

The Muller C-element [9, p. 14-16] is a common asynchronous logic component. It applies logical operations on the inputs and has hysteresis. The output of the C-element reflects the inputs when the states of all inputs match. The output then remains in this state until the inputs all transition to the other state. Table 2.9 shows the truth table for a 2-input Muller-C element. Q_{n-1} denotes a *no change* condition.

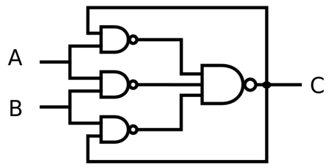


Figure 2.8: Muller-C Element

A	B	Q
0	0	0
0	1	Q_{n-1}
1	0	Q_{n-1}
1	1	1

Figure 2.9: Truth Table

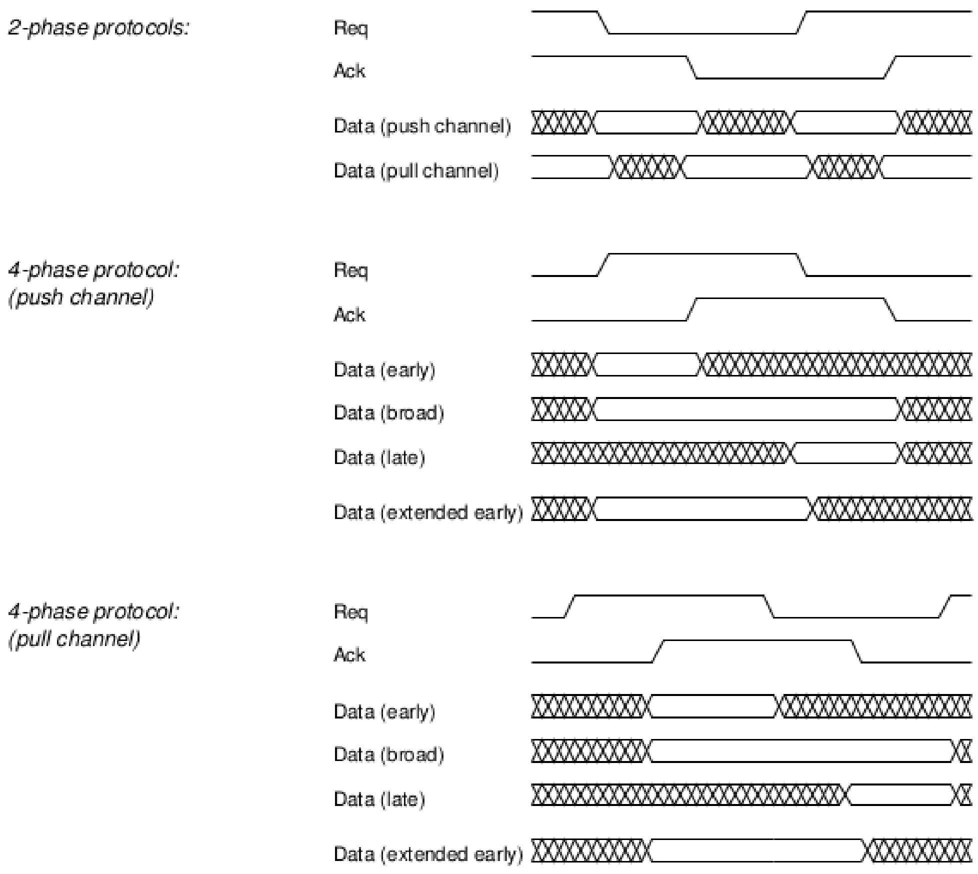


Figure 2.7: Data Validity Schemes for 2-phase and 4-phase Bundled Data

Gate Level Implementation

Different gate-level implementations of the Muller-C element are possible. The technology library used for this project implements the Muller-C element with 4 NAND gates. The gate-level implementation of the Muller-C element using 4 NAND gates is shown in figure 2.8.

2.4 Clock Domain Crossing

When a circuit has more than one clock signal, it is common to partition the circuit into clock domains. Communication between clock domains requires extra design consideration.

2.4.1 Setup, Hold time and Metastability of Flop

Setup time

Setup time is measured at the input of the flip-flop with respect to rising/falling edge of the clock to the flop. The time signifies the minimum duration of data stability before the arrival of rising/falling clock edge. With this requirement the flops will reliably sample the data at the output.

Hold time

Hold time is measured at the output of the flip-flop with respect to rising/falling edge of the clock to the flop. The time signifies the minimum duration of data stability at the output after the rising/falling clock edge. With this requirement the output flip-flop data is stable enough to drive the digital logic.

Metastability

Metastability is a condition on the output signal of a flip-flop due to setup or hold time violations. A metastable signal does not represent a high *1* or a low *0* and results in unstable output or a glitch to the digital circuit. Metastability is a condition on the output signal of a flip-flop due setup or hold time violation on the digital input signal. A metastable signal does not represent a high *1* or a low *0* and results in unstable output or a glitch to the digital circuit.

Figure 2.10 from [17] shows how a two flip-flop synchronizer scheme can be used to implement clock domain crossing for phase offset clocks.

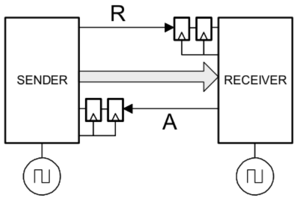


Figure 2.10: Two Flip-Flop Synchronizer

2.4.2 Mean Time Between Failures

The mean time between failures (MTBF) is the time separation between the two clock inputs of the two flops of the synchronizers.

$$MTBF = \frac{e^{\frac{T}{\tau}}}{T_W * f_A * f_D} \tag{2.4}$$

T : The settling window.
τ : Settling time constant of the flip-flop.
T_W : Parameter related to its time window of susceptibility.
f_A : The synchronizer's clock frequency.
f_D : The frequency of pushing data across the clock domain boundary.

Example using the constraints for the capacitive touch digital detection filter circuit:
T_W = 50ps

$f_A = 10\text{MHz}$
$f_D = 5\text{MHz}$
Rate of entering metastability $= T_W * f_A * f_D = 2500$ Hz

2.5 Balsa Asynchronous Synthesis System

Balsa is the name of both the framework for synthesising asynchronous hardware systems and the language for describing such systems. Balsa has been developed over a number of years at the Advanced Processor Technologies (APT) group of the School Of Computer Science, The University of Manchester [1]. Balsa is built around the Handshake Circuits methodology and can generate gate level netlists from high-level descriptions in the Balsa language. Both APT (Quasi-Delay Insensitive (QDI)) and single-rail (SR) (BD) circuits can be generated. The approach adopted by Balsa is that of syntax-directed compilation into communicating handshaking components. The advantage of this approach is that the compilation is transparent: there is a one-to-one mapping between the language constructs in the specification and the intermediate handshake circuits that are produced. It is relatively easy for an experienced user to envisage the architecture of the circuit that results from the original description. Incremental changes made at the language level result in predictable changes at the circuit implementation level. This is important if optimisations and design-trade-offs are to be made easily at the source level and contrasts with a Verilog description in which small changes in the specification may make radical alterations to the resulting circuit.

For more information about Balsa, the reader is referred to [8] and [9, p. 153-204]. The development of Balsa Asynchronous Synthesis System can be followed on its project page [2]. The version used for this thesis is Balsa version 4.0, which was released June 10 2010. This release can be obtained from [3].

2.5.1 Balsa Design Flow

Figure 2.11 from [8, p. 4] shows an overview of the Balsa design flow.

2.5.2 Data Typing Issues

Balsa is strongly typed: both left and right-hand side of assignments are expected to have the same type. The only form of implicit type-casting is the promotion of numeric literals and constants to a wider numeric type. In particular care must be taken to ensure that the result of an arithmetic operation will always be compatible with the declared result type.

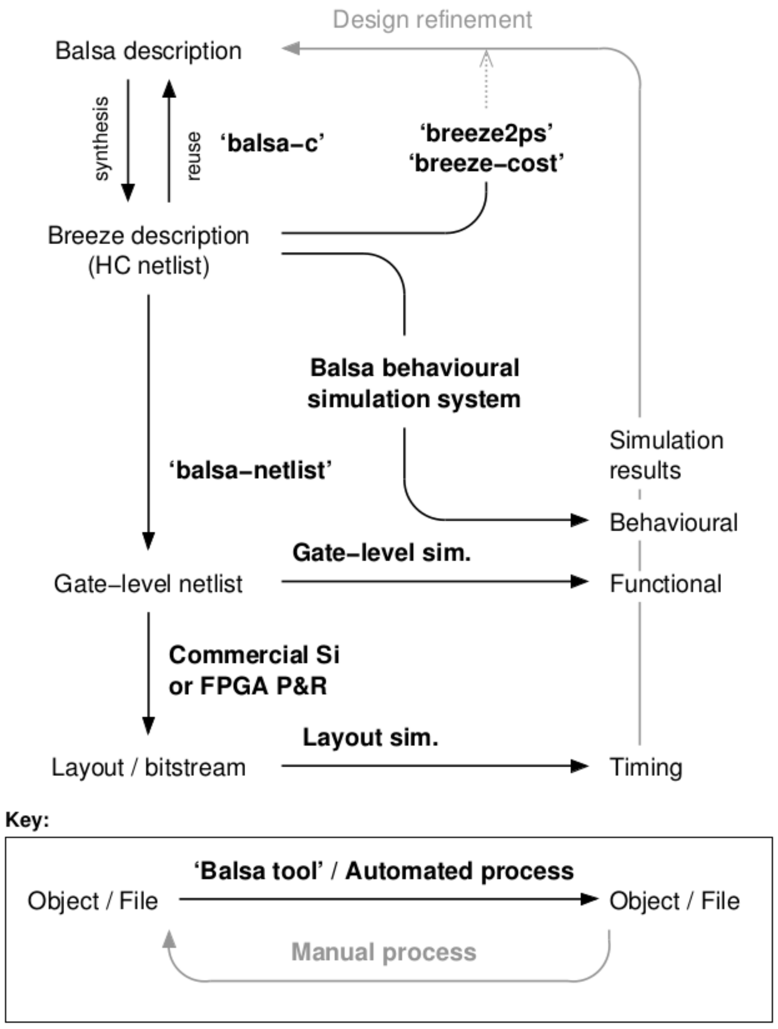


Figure 2.11: Balsa Design Flow.

Non-Delay-Insensitive Components

Non-delay-insensitive components are unsafe components whose behaviour can break due to race conditions. They are generated by the Balsa compiler when sequenced select/arbitrate statements on the same channel are used. The activation of their input leads to the activation of all their outputs, but only one output acknowledgement is expected in return. Other outputs will be *Returned-To-Zero* (if 4-phase protocol) even without a proper acknowledgement. These components are: *CallActive* and *CallDemuxPush* [8, p. 145].

Chapter 3

Implementation

Two implementations of the capacitive touch digital detection filter circuit specified in chapter 1.1 has been made, one using synchronous methodologies and one using asynchronous methodologies. Both implementations use an architecture where the circuit is partitioned into smaller modules. The smaller modules is arranged in a hierarchical manner to form a larger, more complex, module. This is done to reduce the complexity of the circuit, thus making it easier to implement and verify correct behaviour.

The register transfer level (RTL) code for the synchronous implementation is written in Verilog and the asynchronous implementation is written in Balsa. The code listings for the Verilog code is found in appendix B. The code listings for the Balsa code is found in appendix A.

The following sections describe both the synchronous and the asynchronous implementation of each module in the circuit.

3.1 Top Module

The top level module is on the top of the hierarchy and contains all the modules needed by the circuit. Figure 3.1 shows the modules that are instantiated by the top module.

The asynchronous implementation does not include a bridge for interfacing with a synchronous circuit, since the the external circuit that uses the interface of the capacitive touch digital detection filter circuit has not been specified. [1]

Submodules
- Control
- Sampler
- Median-3 filter
- Exponential moving average filter
- Threshold comparator
- Register bank

Figure 3.1

[1]This is to not add unfavourable overhead to the asynchronous implementation.

The organisation of the register bank is different for the synchronous and the asynchronous implementation. [2]

3.1.1 Architecture

The architecture for the top module puts distinct functionality into its own modules. The advantage of this approach is that it is easy to envision and therefore easy to build. The disadvantage is that the modules can not share common hardware structures. Figure 3.2 shows the architecture of the top module.

Channels

Channels are used for communication between modules. A channel is connected between an active and a passive port. A filled circle denotes an active port, while an open circle denotes passive port. An arrow represent a channels and the direction of the arrow represent the direction of the data flow. An arrow from an active port to a passive port denotes a push channel, while an arrow from an passive port to an active port denotes a pull channel.

Behaviour of Top Module

The behaviour of signals in the top module gives a good overview of how the whole circuit works. Figure 3.4 shows an example waveform where the circuit is started, and a sample, filter and threshold compare sequence is performed. Activity in submodules is omitted for brevity, but is instead described in figure 3.3.

Waveform Description
1. Sampling.
2. Median-3 filtering.
3. EMA filtering.
4. Threshold comparison.

Figure 3.3

[2]The registers associated with the register bank in the synchronous implementation is distributed among the modules which are using them. This could be organised into one single register bank module, as done in the asynchronous implementation.

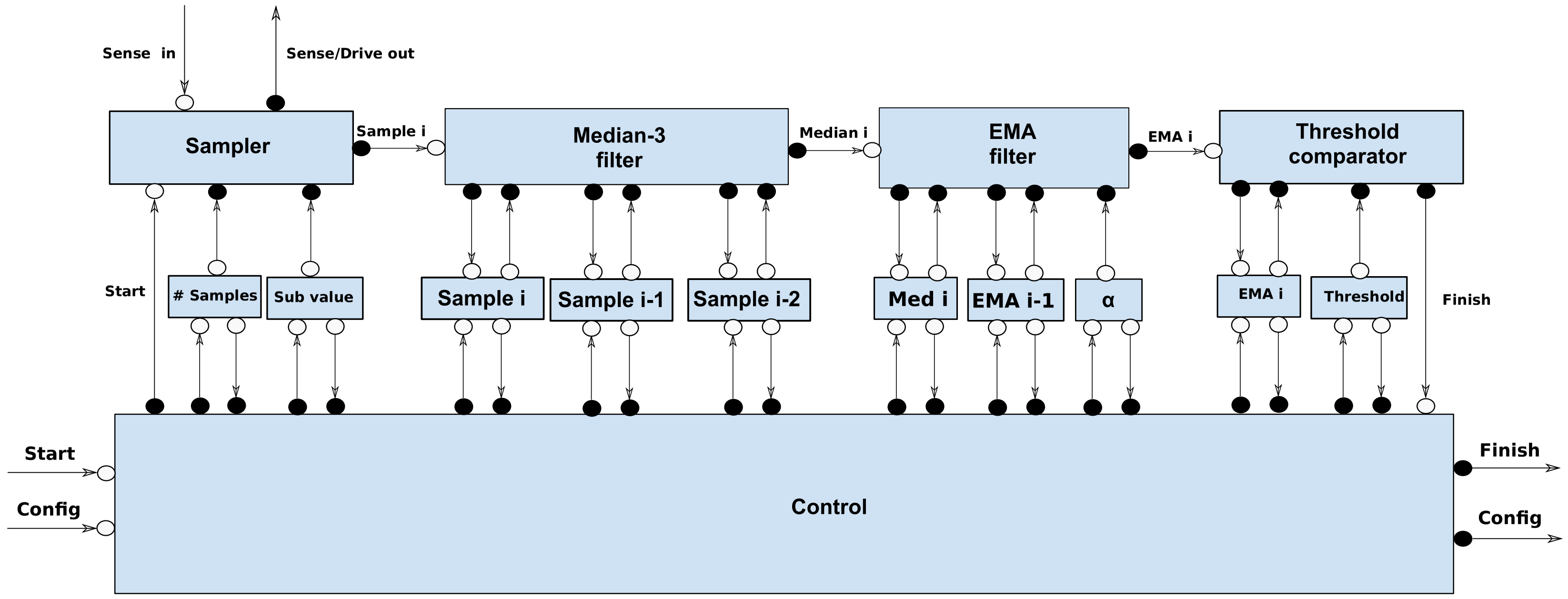


Figure 3.2: Top Module Architecture

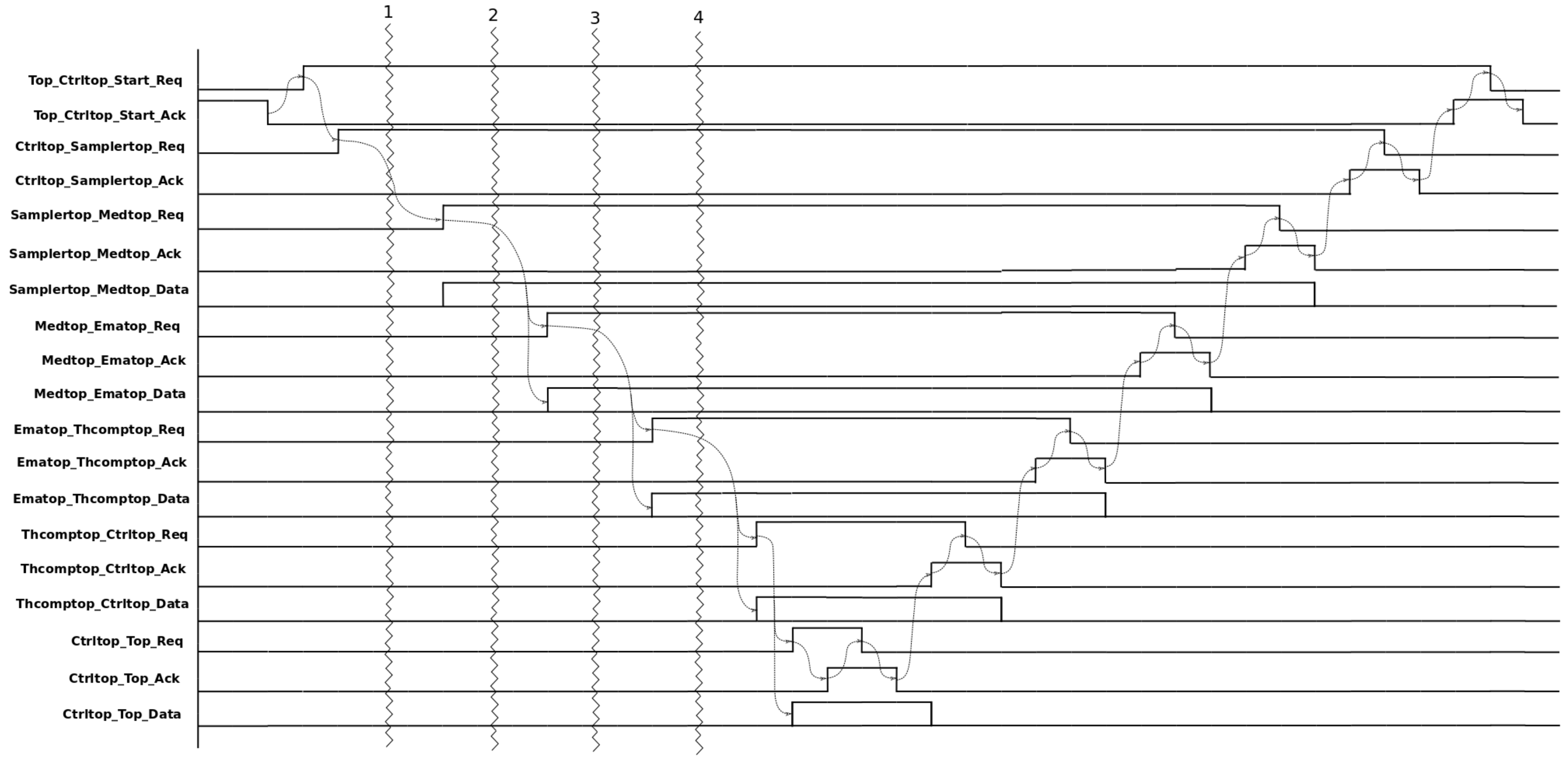


Figure 3.4: Start, Sample, Filter and Compare Sequence.

3.2 Control Module

The control module is responsible for controlling operations initiated via the top module interface. The control module responds to the commands listed in figure 3.5. The *Start* command initiates a sample, filter and threshold comparison sequence. The *Read* command reads data from a register. The *Write* command writes data to a register.

Commands
- *Start*
- *Read*
- *Write*

Figure 3.5

3.3 Register Bank

Important calculation parameters and results are stored in a register bank. This register bank is accessible via the top module interface using the *Read* and *Write* commands. The parameter registers can be read and written to from the top module interface. The result registers can only be read from the top module interface.

Table 3.1 shows the register bank map. The map shows which registers that can be read/written and their address.

3.3.1 Registers

There are two *types* of registers - *reg1regw2r* and *reg1cfgw2r*. The difference between the two types is the means of addressing the register for the write port. The first is addressed directly by a module, while the second is addressed via the configuration write bus. They are functionally equivalent, but the name of the write port is different to make wiring modules in the top module easier.

Register name	Readable	Writable	Address
Number of samples	✓	✓	0x00
Constant value to subtract	✓	✓	0x01
Alpha value	✓	✓	0x02
Threshold value	✓	✓	0x03
Sample$_i$	✓	X	0x04
Sample$_{i-1}$	✓	X	0x05
Sample$_{i-2}$	✓	X	0x06
Median$_i$	✓	X	0x07
EMA$_i$	✓	X	0x08
EMA$_{i-1}$	✓	X	0x09

Table 3.1: Register Bank Map

Synchronous Implementation

A read or a write operation on a register uses one clock cycle to complete. If a read and a write operation on a register happens at the same time, the old register value is read. The result of the write operation is observable after one clock cycle.

Asynchronous Implementation

If the write or read operation on a register is governed by the Balsa statement *select* and there is a possibility for a read and a write operation to happen at the same time, then there is a possibility for metastability. Thus, if the asynchronous implementation is going to have the same functionality as the synchronous implementation, allowing reading from and writing to registers when the circuit is active, the write and read operations on a register need to be arbitrated. Two arbiters are needed per register to achieve this functionality. Arbitration is achieved using the Balsa statement *arbitrate*. The arbiter handshake component contains a mutual exclusion component which is very expensive in terms of area cost and slow speed.

3.4 Sampler Top Module

Both the synchronous and the asynchronous circuit use a *sampler_clock* signal for timing the sampling of the analog *sense_in_d* signal. The synchronous circuit uses a different *clock* signal for the rest of the circuit to avoid running on as high frequency as the *sampler_clk* signal. Therefore all signals going to and from the synchronous sampler module must be synchronised to reduce probability of metastability issues. The asynchronous implementation on the other hand uses a clever trick to make the *sampler_clk* signal work for it. Figure 3.6 shows the modules which are instantiated by the sampler top module. Figure 3.7 shows the additional modules for the synchronous implementation. Figure 3.7 shows the additional modules for the asynchronous implementation. Figure 3.9 shows the tasks performed by the sampler top module.

Submodules
- Sampler
- Sampler registers

Figure 3.6

- Synchronizer for *clk_sampler_en.*
- Synchronizer for *sampler_reset.*
- Synchronizer for *start.*
- Synchronizer for *finish.*
- Synchronizer for *sense_in_d.*

Figure 3.7

- I/O Wrapper.

Figure 3.8

3.4.1 Synchronous Implementation of Sampler Top Module

When communicating across clock domains, such as in the sampler top module, it is crucial that the communication happens in a safe way. Unsafe communication can result in metastable signals and unpredicted behaviour.

Tasks
1. Wait for start signal from control top module.
2. Read *numsamples* register.
3. Read *subvalue* register.
4. Send start signal with data to sampler module.
5. Wait for finish signal with data from sampler module.
6. Send start signal with data to median-3 filter top module.

Figure 3.9

Implementation

Figures 3.10a and 3.11a show two alternatives for implementing a communication protocol between the sampler top module and sampler module has been considered. Alternative 1 uses a *safe by design* approach, while alternative 2 uses a strict 4-phase protocol. Both require a *finish* signal synchronizer.

Alternative 1
1. *Reset* is held low.
2. *Reset*, *start* and *sampler_clock_en* goes high.
3. Sampler makes a measurement.
4. When sampler is finished, *finish* is held high until sampler is reset.
5. Samplertop resets sampler after *finish* goes high.

(a)

Advantages
+ No *start* signal synchronizer.
Disadvantages
- Must know details of the implementation to use the interface.

(b)

Figure 3.10

Alternative 2
1. Samplertop holds *start* high.
2. Sampler makes a measurement.
3. Sampler outputs data.
4. Sampler holds *finish* high.
5. Samplertop captures data.
6. Samplertop holds *start* low.
7. Sampler holds *finish* low.

(a)

Advantages
+ Modularity of interface.
+ Robust 4-phase protocol.
Disadvantages
- Additional *start* signal synchronizer.

(b)

Figure 3.11

Alternative 1 uses the reset signal for the sampler module to keep the sampler module in idle mode until it is used. Alternative 2 uses a four-phase pull protocol. *start* =

request signal. *finish* = acknowledge signal. The sampler is in idle mode until *start* goes high.

Alternative 2 is chosen for implementation due to the modularity of the interface. The 4-phase protocol employed in the second alternative had been optimised because of the following observations. The clock frequency of the sampler module is assumed to be $f_{sampler} >= f_{clk}$. This assumption is required for the design to work correctly. The sampler module uses a maximum of two clock cycles to set *finish* low. The sampler top module uses two/three clock cycles to reach the state *MEASURE* where it tests for *finish* low. Thus it is safe to remove the *wait for finish/ack low*-state in sampler top module. Data is outputted and *samplertop_mediantop_start* is set high when *finish/ack* is detected high.

Synchronisation

All control signals crossing clock domain borders need to be synchronised in order to avoid metastability. The *start*, *finish*, *reset*, *sampler_clk_en* and $sense_{in}$ signals needs to be synchronised in order to avoid metastability. The synchronizers for the *start*, *finish*, *sampler_clk_en* and $sense_{in}$ signals use standard double flip-flops for synchronisation. The synchronizer for the *reset* signal is a variant of the double flip-flop synchronizer. Figure 3.12 shows the architecture for the synchronised-trail negative reset signal synchronizer module.

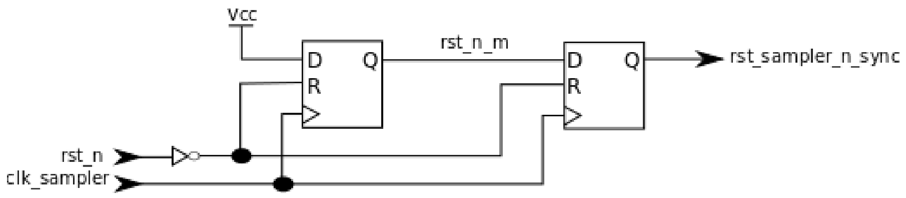


Figure 3.12: Reset Signal Synchronizer Module

3.4.2 Asynchronous Implementation of Sampler Top Module

The Balsa language has no handshake components for driving/reading I/O ports directly. Thus a custom Verilog wrapper module is needed between the Balsa handshake I/O ports and the sampler module interface.

Wrapper Module

The sampler module interface is forwarded through the sampler top module to the external interface of the top module, and connected to the wrapper module in the testbench. The wrapper module responds to handshakes on the *sensedrive_out* port with the configuration for the *sense_oe*, *sense_out*, *drive_oe*, *drive_out* signals and sets them accordingly. The wrapper module contains a synchronizer for both the *sensedrive_out_r* request signal coming from the sampler module and the *sense_in_d* data signal going to the sampler module to avoid metastability. The *reset_sampler* signal comes from the testbench and only at the start of the simulation in synchronisation with the *clk_sampler* signal and is therefore not implemented with a synchronizer.

3.5 Sampler Module

The sampler module is the most important module in the design. Figure 3.13 shows the tasks performed by the sampler module. It performs capacitance measurements by counting the time it takes to charge and discharge the capacitance in the RC circuit connected to the $sense_{in}$ input signal. This type of capacitive sensing is called self capacitance sensing (sec. 2.1).

The sampler needs a time reference in order to sample the $sense_{in}$ input signal with a fixed period. This requires a local synchronous clock and a counter for doing a fixed number of samples. The number of samples is configurable by writing to the *numsamples* register. Figure 3.14 shows the architecture for the sampler module.

Tasks
1. Wait for start signal from sampler top module.
2. Read (*numsamples*) register.
3. (Read (*subvalue*) register).
4. (Load value register with ($-$*subvalue*)).
5. Start *charge* sequence.
6. Sample $sense_{in}$ *numsamples* times.
7. Start *discharge* sequence.
8. Sample $sense_{in}$ *numsamples* times.
9. Send finish signal with data to sampler top module.

Figure 3.13

3.5.1 Synchronous Implementation of Sampler Module

The synchronous implementation of the sampler module has its own clock signal, *clk_sampler*. The *clk_sampler* signal is used as the time reference when sampling the $sense_{in}$ data signal. The $sense_{in}$ data signal is synchronised with a double flip-flop synchronizer, to reduce the possibility of metastability.

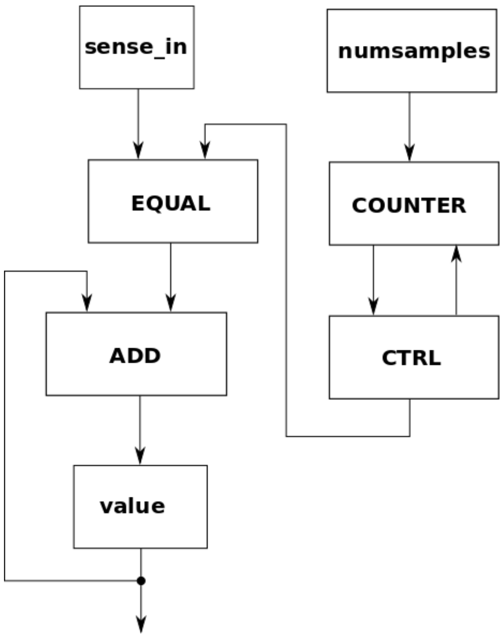


Figure 3.14: Sampler Module Architecture

Listing 3.1: src/asynch/module/balsa/sampler.balsa

```
                select sense_in then -- Synchronize on clock signal
                    [
                      if (sense_in = 0) then
                          add()
                      end
                    ;
                      dec()
                    ]
                end -- select sense_in
```

3.5.2 Synchronous Implementation of Sense/Drive Output Ports

Verilog allows for using registered signals. This synthesises into flip-flops which drives output ports.

3.5.3 Asynchronous Implementation of Sampler Module

Four possible implementations for sampling the $sense_{in}$ signal are investigated in the following sections. All implementations assume a passive *select* statement enclosing the sampling of $sense_{in}$ as shown below. Listing 3.1 shows a code excerpt from the Balsa implementation of the sampler module.

The *select* statement is placed inside the enclosing *select* statement which holds the data from the sampler top module valid until the end of the sequence inside the *loop* statement. Sequential use of select channels result in the following compilation error:

```
sampler.balsa:82:13: making sequential use of arbitrate'd/select'ed channels is usually non-DI
(specify the "-c allow-sequential-selection" compilation option to override) 'sense_in'

*** 1 error, 0 warnings
```

The compilation option *-c allow-sequential-selection* is used to override this error message and replace it with a warning.

Alternative 1

The first alternative is to connect an odd number of inverters between the *acknowledge* output and the *request* input of the $sense_{in}$ port of the sampler module. This implementation of a clock signal is analogous to a ring oscillator. Figure 3.16 shows an implementation using a single inverter.

Advantages
+ Fast.
+ Simple.

Disadvantages
- Cannot be described in Balsa.
- Need to edit Verilog netlist.
- Fixed sample rate dependent on fixed circuit speed.

Figure 3.15

The implementation works in the following manner: The *acknowledge* signal from the sampler module is initialised to *0*. This means that the *request* signal to the sampler module is initialised to *1*. When the sampler module sequence comes to the passive *select* statement, it receives a request signal immediately. When the sampler module has finished the sample and decrement counter sequence, the *acknowledge* signal is set to *1*. This results in the *request* signal going to *0* and then the *acknowledge* signal is set to *0*, effectively completing the handshake. The *request* signal is again set to *1*, waiting for the sampler module to perform a new sample sequence.

The sampler module is a passive component, while the inverter circuit is active.

The sample frequency from this implementation is dependent on the odd number of inverters in the inverter chain, in addition to the speed of the addition and subtraction in the sampler module.

The $sense_{in}$ data signal is synchronised using two positive edge triggered flip-flops. The flip-flops are clocked by the *request* signal. Data becomes valid after *request* goes high. Data is valid until *acknowledge* goes low, resulting in request *1* and a positive edge on the flip-flops. This results in an extended *broad* signal validity scheme. The 4-phase bundled data protocol implementation in the technology used for this project uses a *broad* or *reduced broad* validity scheme, which is a subset and thus it is compatible. However, there is one potential problem with clocking the flip-flops on the positive edge when *request* goes high. The setup time of the flip-flop may be longer than the propagation of the *request* signal into a latch, resulting in metastability. Possible solutions are insertion of delay element or clocking of negative edge triggered flip-flops from the output of the Muller-C element [7].

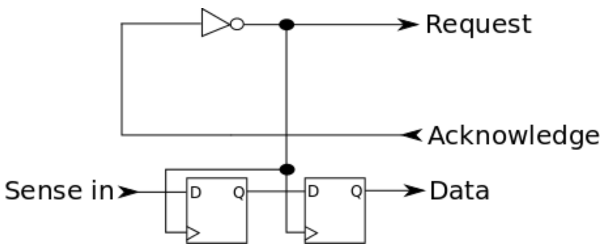


Figure 3.16: Simple Inverter Chain

Alternative 2

The second alternative is an improvement on alternative 1. It uses a *Muller-C* element and a delay element to regulate the speed of the handshake, i.e. the sample period. If the sampler circuit uses shorter time than the delay from the delay element, then the minimum sample period is set by this delay. If the sampler circuit uses longer time than the delay from the delay element, then the sample period matches the time the sampler circuit uses. Figure 3.18 shows the architecture for the delayed inverter chain.

Advantages
+ Fast.
+ Simple.
+ Can set sample frequency by adjusting the delay.
+ Allows for variable circuit speed.

Disadvantages
- Cannot be described in Balsa.
- Need to edit Verilog netlist.

Figure 3.17

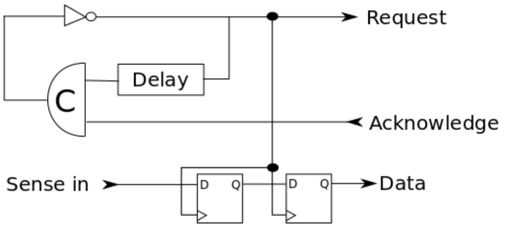


Figure 3.18: Delayed Inverter Chain

Delay Element One challenge with this implementation is how to design the delay element. One option is to use a fixed even number of inverter connected in series. An other option is to use two inverter buffers with an RC circuit between, as shown in figure 3.19. The resistor and capacitor is variable, so the delay can be changed. Figure 3.19 shows a possible implementation of the delay element.

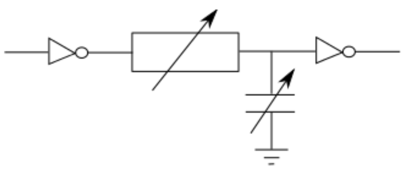


Figure 3.19: Delay Element

C_{in0}	C_{in1}	C_{out}
x	x	x
0	x	x
0	x	0
0	1	0

(a) Output of C-Element Settling to *0*

C_{in0}	C_{in1}	C_{out}
x	x	x
0	x	x
0	x	1
0	0	1
0	0	0
0	1	0

(b) Output of C-Element Settling to *1*

Figure 3.20: Metastability

Initialisation An other challenge with this implementation is how to initialise both inputs of the *Muller-C* element to zero. Only the *acknowledge* signal from the sampler module can be assumed to have the reset value *0*. The output signal from the C-element when the circuit is reset is unknown, i.e. metastable. It is theoretically possible for this signal to be metastable forever, but the probability of staying metastable decreases exponentially over time. Thus, the output signal from the C-element will settle to the value *0* or *1* after some time. Figure 3.20a and 3.20b shows the two most likely sequences. Both sequences end up in the same input and output states for the Muller-C element. The flip-flops can use the global reset signal *initialise* to initialise their outputs to *0*. This together makes this implementation safe.

Alternative 3

Alternative 3 uses a clock signal to time the request signal for the $sense_{in}$ channel going to the sampler module. This handshake operation uses a 4-phase protocol. The acknowledge signal from the sampler circuit is not connected to anything. The correctness of this implementation depends on that the time it takes for the sampler circuit to sample the $sense_{in}$ signal and decrement the sample counter is shorter than the time it takes for the clock signal to reach a negative clock flank (request = *0*).

Advantages
+ Easy to match the sampler clock speed in both implementations.
+ No netlist editing to change clock speed.

Disadvantages
- Possibility of metastability because of trailing edge of *sampler clock* signal.

Figure 3.21

Figure 3.22 shows the architecture for alternative 3.

Challenge: Trailing edge of *clk_sampler* signal.

Challenge: Start clock.

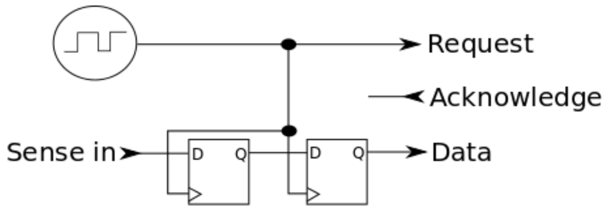


Figure 3.22

Listing 3.2: src/asynch/module/balsa/sense/sense.balsa

```
    loop
        [
          sense_in_d_m := 1 -- Constant
        ;
          sense_in_d := sense_in_d_m -- Double latched buffer
        ;
          push <- sense_in_d -- Push channel
        ]
    end -- loop
```

Alternative 4

Alternative 4 uses a combination of handshake components generated from Balsa code and editing the Verilog netlist. It works by routing the $sense_{in}$ data signal through a modified *BrzConstant* module called *sensemodule* into a double latch/buffer before it is outputted through a push channel. Double latching the data gives similar [3] probability of resolving metastability as the double flip-flop synchronizer equivalent for the synchronous implementation. Figure 3.24 shows the generated handshake components from the Balsa code.

Advantages
+ Safe.
+ Can generate most of the Verilog netlist for the wrapper module using Balsa.
+ Simple.

Disadvantages
- Slow.
- Need to edit Verilog netlist.
- Fixed sample rate depends on fixed circuit speed.

Figure 3.23

The sampling rate of this implementation is dependent on the speed of the speed of *sensemodule*. The sampling frequency is fixed only if the speed of the sampler module is the same for all samples. Listing 3.2 shows the Balsa code for the *sensemodule*.

The *BrzConstant* handshake module in the Verilog netlist is replaced with a custom *sensemodule* module and an extra input and wire for the *sense_in* channel. The sample frequency is set by the speed of the circuit.

[3] This depends on the physical parameters for the latch.

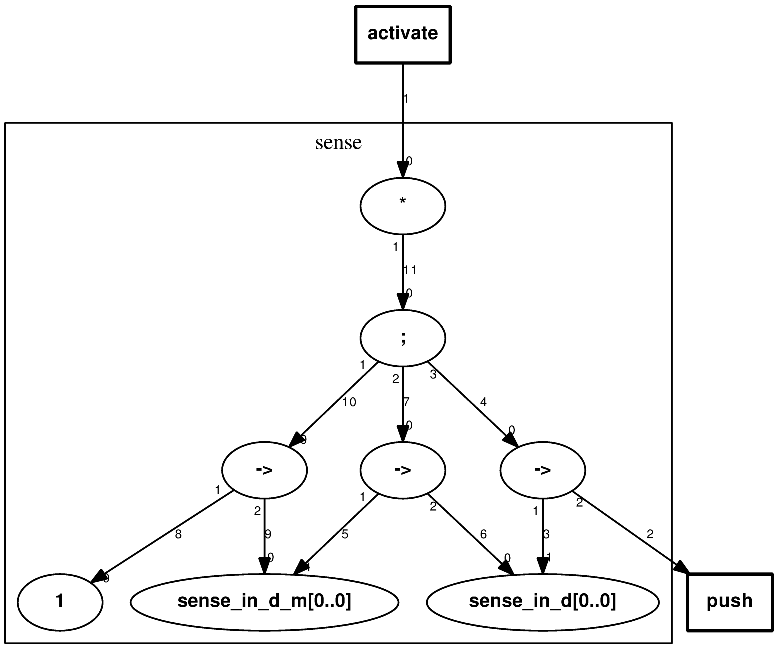


Figure 3.24: Handshake Components.

Implementation

While alternative 1, 2 or 4 would most likely be chosen for a physical implementation, alternative 3 is chosen. This is because changes in the Balsa code lead to synthesis of a completely new netlist, discarding all changes to the netlist. It is easier to test the circuit if the additional Verilog code wrapper module can be instantiated in the testbench and connected to an unedited Balsa netlist. It is also easier to match the sampling frequency of the synchronous circuit, allowing for a more fair comparison.

3.5.4 Implementation of Sense/Drive Output Ports

Before sampling, the output ports $sense_{oe}$, $sense_{out}$, $drive_{oe}$ and $drive_{out}$ need to be configured. Balsa does not support driving I/O buffers.

Alternative 1

Alternative 1 is to write the configuration to a variable in the Balsa code, as shown in listing 3.3. Then the synthesised netlist is edited, so that the output of the latches, storing the configuration for the output ports, is connected to the RC circuit model and driving the correct inputs.

Advantages
+ Elegant.
Disadvantages
- Need to edit Verilog netlist.

Figure 3.25

Listing 3.3: src/asynch/module/balsa/sensedrive/sampler.balsa

```
                sensedrive_out := 0b1010
```

Alternative 2

Alternative 2 is to use a push channel output for sending the configuration for the output ports, as shown in listing 3.4. The configuration must be received by an external module which in turn drives the output ports. The external module called *top_wrapper* is written in Verilog and is instantiated in the testbench. The wrapper module path is *src/asynch/module/verilog/top_wrapper.v.*

Advantages
+ Do not have to edit the Balsa netlist.
Disadvantages
- Need external Verilog module.

Figure 3.26

Listing 3.4: src/asynch/module/balsa/sampler.balsa

```
                sensedrive_out <- {
                  -- sense_oe high
                  1,
                  -- sense_out low
                  0,
                  -- drive_oe high
                  1,
                  -- drive_out low
                  0
                }
```

Implementation

Alternative 2 was chosen, because it separates the netlist generated from the Balsa code and the wrapper module in the testbench. Thus avoiding having to rewrite the changes to the netlist every time the Balsa code is changed and re-synthesised.

Alternative 1 would be chosen for a physical implementation, but since the circuit is only simulated pre-layout this is good enough.

3.6 Median-3 Filter Top Module

The median-3 filter top module contains the first in a series of two filters. In addition it contains a system for storing and retrieving the three last capacitance measurements received from the sampler top module. Figure 3.27 shows the modules which are instantiated by the median-3 filter top module. Figure 3.28 shows the tasks performed by the median-3 filter top module.

Submodules
- Median-3 filter
- Median-3 filter registers

Figure 3.27

Tasks
1. Wait for start signal with data from sampler top module.
2. Write data to register containing the oldest data.
3. Read data from 3 registers.
4. Send start signal with data to median filter module.
5. Wait for finish signal with data from median filter module.
6. Send start signal with data to EMA filter top module.

Figure 3.28

3.7 Median-3 Filter Module

The median-3 filter module performs the first step of noise filtering. Here shot noise is filtered by taking the median of the three last capacitance measurements. Figure 3.29 shows the implementation of the median-3 filter.

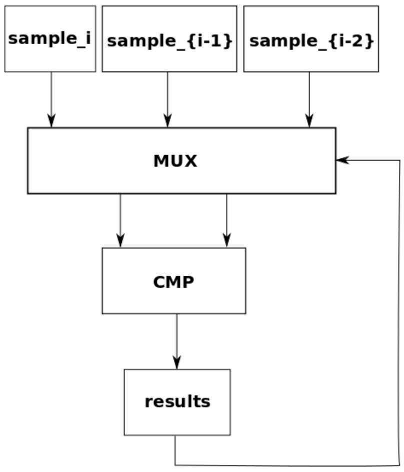


Figure 3.29: Median-3 Filter Module Architecture

3.7.1 Registers

The median-3 filter module needs to store the three previous data values received from the sampler module. I.e. it needs at least three N-bit registers.

Register Storage

For each new computation the median-3 filter module receives one data value on the input. This new data value must replace the oldest sample value. Two possible implementations are a shift register or a cyclic register. If a shift register is used, all register values are shifted one place in parallel. If a cyclic register is used, only the oldest data value is overwritten. In order to keep track of which register contains the oldest data value, a 2-bit counter is used. When storing a new data value, the counter is checked to see which register the new data value should overwrite.

Since a shift register moves all register values, and a cyclic register only moves one register value, a cyclic register should lead to fewer signal transitions when storing a new data value. This does not take into the account the fact that a cyclic register requires more control logic.

3.7.2 Median-3 Algorithm

The median algorithm can be implemented using a bubble-sort algorithm. The bubble-sort algorithm is used to sort values into a list, and then the median value can be picked from the middle of the list. A median filter with window length N=3 needs to do a total of 3 comparisons to sort the 3 data values and find the median. To minimise the number of comparator structures, one comparator is used and the result of each comparison is stored in a 1-bit register for a total of 3 1-bit result registers.

Comparison

Both Verilog and Balsa supports the $>$ operator. The comparison $(A > B)$ is equivalent to $(A - B > 0)$. Both implementations result in a full-adder structure which performs a subtraction and checks the result if it is larger than zero.

3.8 EMA Filter Top Module

The EMA filter top module contains the second filter in a series of two filter, in addition to a a system for storing and retrieving data needed for the EMA filter module. Figure 3.30 shows the modules which are instantiated by the EMA filter top module. Figure 3.31 shows the tasks performed by the EMA filter top module.

Submodules
- EMA filter
- EMA filter registers

Figure 3.30

The EMA filter needs to store the two N-bit previous data values received from the median-3 filter. A precomputed value, α, is stored in a N-bit configuration register. I.e. it needs three N-bit registers.

Tasks
1. Wait for start signal with data from median-3 filter top module.
2. Write data to register.
3. Read data from registers.
4. Send start signal with data to EMA filter module.
5. Wait for finish signal with data from EMA filter module.
6. Write data to register.
7. Send start signal with data to threshold comparator top module.

Figure 3.31

3.9 EMA Filter Module

The EMA filter module performs the EMA algorithm. Figure 3.32 shows the modules which are instantiated by the EMA filter module. Figure 3.33 shows the tasks performed by the EMA filter module.

Submodules
- Multiplier

Figure 3.32

3.9.1 EMA Algorithm

Equation 3.1 shows the EMA algorithm. The algorithm consists of one subtraction, one multiplication and one addition. The intermediate result of the three operations is stored in a (N+1) signed register.

$$EMA_i = EMA_{i-1} + \alpha * (MED_i - EMA_{i-1}) \tag{3.1}$$

Given that MED_i is always a positive value implies that EMA_i is always a positive value. Since EMA_{i-1} can be larger than MED_i and the result of the subtraction negative, the intermediate result of the subtraction, multiplication and addition must be a signed value. However, the result of the last addition is always positive and can safely be cast into an unsigned value. Thus signed numbers are only present in the EMA filter module, and only positive numbers are stored in registers available to reads from the configuration interface.

Figure 3.34 shows the architecture of the EMA filter algorithm.

3.9.2 Addition

An addition can be performed with a carry-propagation adder. A carry-propagation adder is a chain of full adders where the $carry_{out}$ of full adder i is connected to $carry_{in}$ of full adder *i+1*.

3.9.3 Subtraction

The result the subtraction can be negative. Both operands in the subtraction are N-bit vectors. An adder can support subtraction if both operands are represented using 2's complement. Then both operands must be sign extended to (N+1)-bit vectors. The result of the subtraction (addition) is a (N+1)-bit signed vector. The subtraction is performed by first inverting the B operand and adding *1*, and then adding the A operand as shown in equation 3.9.3.

$$A - B = A + (-B) = A + \overline{B} + 1 \tag{3.2}$$

Tasks
- Wait for start signal with data from EMA filter top module.
- Calculate EMA_i.
- Send finish signal with data to EMA filter top module.

Figure 3.33

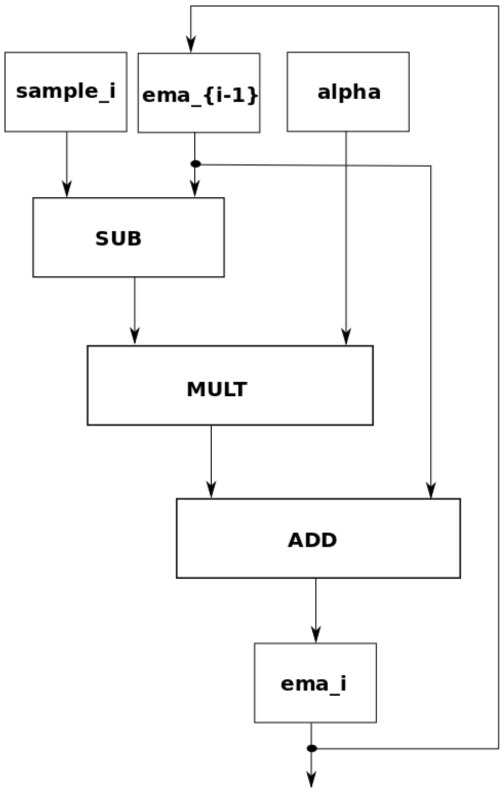


Figure 3.34: EMA Filter Module Architecture

3.9.4 Multiplication

A multiplier is needed to perform the multiplication in the EMA algorithm. The multiplicand operator in the multiplication can be a negative number. Two alternative methods for handling multiplication of two numbers represented using 2's-complement have been considered.

Alternative 1

The first alternative is to first check if the multiplier is negative. If so, take the 2's complement of both operands before multiplying. The multiplier will then be positive so the algorithm will work. Because both operands are negated, the result will still have the correct sign.

Alternative 2

The second alternative is to subtract the partial product resulting from the MSB (pseudo sign bit) in the multiplier instead of adding it like the other partial products. This method requires the multiplicand's sign bit to be extended by one position, being preserved during the shift right actions.

Implementation

Alternative 1 is chosen for implementation, because since the α multiplier operand in the multiplication is always positive and the multiplicand is represented in 2's complement the multiplication will work without more consideration.

In addition, α is a decimal number always smaller than 1, and is represented using fixed point number representation. The result of the multiplication is a (2*N + 1)-bit signed vector. The extra precision in the result is not needed and therefore only the (N+1) most significant bits are sliced from the result, effectively truncating the result.

3.9.5 Synchronous Implementation of EMA Filter Module

Addition in Verilog

The + operator in Verilog supports both signed and unsigned addition. It generates a full-adder chain (carry-propagation adder).

Subtraction in Verilog

The **-** operator in Verilog supports both signed and unsigned subtraction. However, this would generate a permanent structure with inverters for signal B and a full-adder chain with *carry in* set to *1*. Since the EMA filter also needs to support addition, the following code makes it possible to use the full-adder chain for addition as well. 2's complement number representation is used for supporting negative numbers.

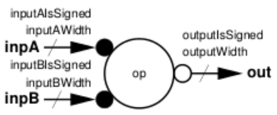
Excerpt from 'ema.v:

```
C = A + (~B + 1'b1);
```

3.9.6 Asynchronous Implementation of EMA Filter Module

Adder in Balsa

The *+* operator in balsa generates a carry-propagation adder handshake component. Figure 3.35 shows the binary function breeze component which is used to implement binary functions.



BinaryFunc
(**parameter** outputWidth : cardinal;
parameter inputAWidth : cardinal;
parameter inputBWidth : cardinal;
parameter op : BinaryOperator;
parameter outputIsSigned : boolean;
parameter inputAIsSigned : boolean;
parameter inputBIsSigned : boolean;
passive output out : outputWidth **bits**;
active input inpA : inputAWidth **bits**;
active input inpB : inputBWidth **bits**)

type BinaryOperator is enumeration (op symbol between brackets)
Add (+), Subtract (-), ReverseSubtract (\\-), Equals (==), NotEquals (!=), LessThan (<), GreaterThan (>), LessOrEquals (<=), GreaterOrEquals (>=), And (&), Or (|)
end

#[out !° inpA ?• inpB ?• *op*(*outputWidth*, *outputIsSigned*, *inputAIsSigned*, *inputBIsSigned*, *op*, *inpA*, *inpB*)]

Figure 3.35: Breeze Component: Binary Function.

Subtraction

It is possible to share hardware by using the *shared* Balsa procedure.

A carry-propagation adder has a *carry in* to the LSB full adder which can be set to 1b'1. This can be exploited when performing subtraction using 2's complement number representation.

Attempt 1 The first attempt at a shared carry propagation adder for addition and subtraction results in two adders. One adder for adding the *carry bit* to one operand, and one adder for adding the immediate result to the second operand. This is because balsa is strongly typed and therefore both + operators results in a handshake component.

Listing 3.5 shows an excerpt from the first version of *ema.balsa*.

Breeze-cost shows that the relative cost is high due to instantiating two binary function adders, one for each + operator.

Excerpt from *breeze-cost* output:

```
(929.5 (component "$BrzBinaryFunc" (9 10 1 "Add" "false" "false" "false")
(48 47 44) (at 23 29 "ema.balsa" 0)))
(1147.5 (component "$BrzBinaryFunc" (10 9 9 "Add" "false" "false" "false")
(47 46 45) (at 23 24 "ema.balsa" 0)))
```

Listing 3.5: Excerpt from the first version of ema.balsa

```
type TN1 is (NUM_BITS+1) bits
........
    variable r0 : TN1
    variable r1 : TN1
    variable carry_in : bit
    variable res_tmp : TN1
........
    shared add is
    begin
        res_tmp := (r0 + r1 + carry_in as TN1)
    end -- shared add
........
              r0 := (ematop_ema[0] as TN1)
            ;
              r1 := (not(ematop_ema[1] as TN1) as TN1)
            ;
              carry_in := 1
            ;
              add()
```

Attempt 2 In the second attempt, signed registers are used for both operands and the result. By loading the *r1* register with a negative value, the subtraction is performed using only one adder structure as seen in the *breeze-cost* excerpt below.

Listing 3.6 shows an excerpt from the second and final version of *ema.balsa*. The complete listing is found in appendix 3.20.

Listing 3.6: Excerpt from the second version of ema.balsa

```
type TN is (NUM_BITS) bits
type TN1 is (NUM_BITS+1) bits
type TNS is (NUM_BITS+1) signed bits
........
    variable r0 : TNS
    variable r1 : TNS
    variable res_tmp : TNS
........
    shared add is
    begin
        res_tmp := (r0 + r1 as TNS)
    end -- shared add
........
              r0 := (ematop_ema[0] as TNS)
            ;
              r1 := (-(ematop_ema[1] as TN1) as TNS)
            ;
              add()
```

The result from *Breeze-cost* shows that the relative cost is almost halved in comparison to the first attempt.

Excerpt from *breeze-cost*:

```
(1147.5 (component "$BrzBinaryFunc" (10 9 9 "Add" "true" "true" "true")
(44 43 42) (at 22 24 "ema.balsa" 0)))
```

Gotcha : Casting and Negative Zero Compare the following two lines:

```
1      r1 := (-(ematop_ema[1]) as TNS)
2      r1 := (-(ematop_ema[1] as TN1) as TNS)
```

Keep in mind that ematop_ema[1] is of type TN (one bit shorter than TN1 and TNS). Will the value loaded into r1 be same for both lines for all values of ematop_ema[1]? No. Figure 3.36 shows an example of code line 1 used for loading the value *0b0000* loaded into register *r1*. This example shows that line 1 results in an error when ematop_ema[1] = 0. The value stored in register r1 is called *negative zero*. This is not an allowed value in 2's complement encoding. The reason for the error is because the length of the intermediate register is not increased before the value -(0) is loaded into it, and then it is cast into a sign extended vector. Figure 3.37 shows an example of code line 2 where the length of the intermediate register is increased before the value -(0) is loaded into it. Line 2 is correct because the vector width is increased before putting the value -(0) into it and casting it into a signed vector.

```
!    0000 Invert
     1111
+    0001 Add '1' and cast to (4+1) signed bits
=   10000
```

Figure 3.36

```
     0000 Cast into (4+1) bits
!   00000 Invert
    11111
+   00001 Add '1' and cast to (4+1) signed bits
=   00000
```

Figure 3.37

Multiplication in Balsa

The Balsa operator takes numeric types and is only applicable to constants. Therefore, multiplication in Balsa requires the design of a multiplication module.

A shift-add algorithm has been chosen for the implementation of the multiplier in Balsa. This algorithm supports multiplication of two N-bit operands represented using 2's complement. Figures 3.38 and 3.39 from [6] shows the shift-add algorithm and circuit.

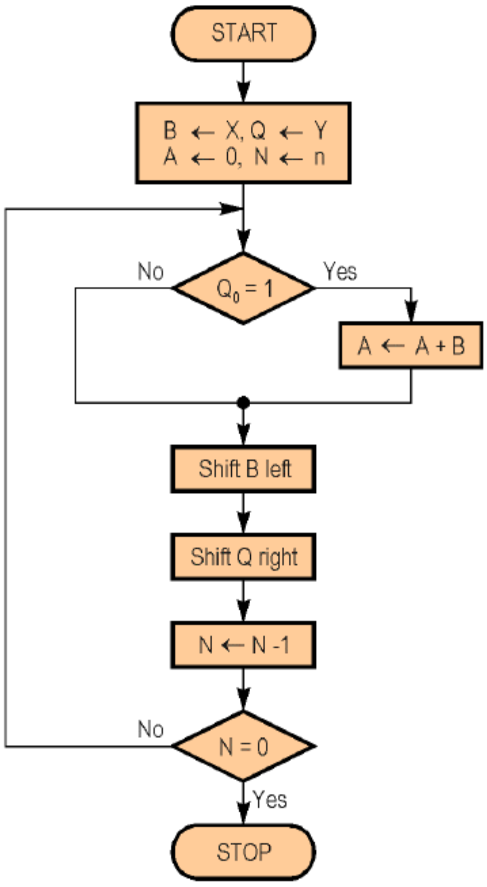


Figure 3.38: Shift-Add Algorithm

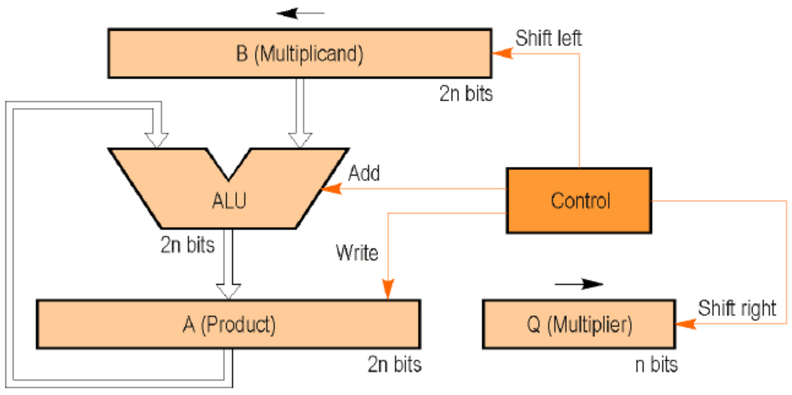


Figure 3.39: Shift-Add Architecture

3.10 Threshold Comparator Top Module

The threshold comparator top module contains a threshold comparator and a system for storing and retrieving the last value received from the EMA filter top module, in addition to retrieving the threshold value stored in a configuration register. Figure 3.40 shows the modules which are instantiated by the threshold comparator top module. Figure 3.41 shows the tasks performed by the threshold comparator top module.

Submodules
- Threshold comparator
- Threshold comparator registers

Figure 3.40

Tasks
1. Start signal with data from ematop.
2. Write data to register (EMA_i).
3. Read data from 2 registers (EMA_i, THRESHOLD).
4. Send start signal with data to threshold comparator module.
5. Wait for finish signal with data from threshold comparator module.
6. Send start signal with data to control module.

Figure 3.41

3.11 Threshold Comparator Module

Figure 3.42 shows the tasks performed by the threshold comparator module.

Tasks
1. Wait for start signal with data from threshold comparator top module.
2. Perform comparison ($EMA_i > THRESHOLD$).
3. Send finish signal with result to threshold comparator top module.

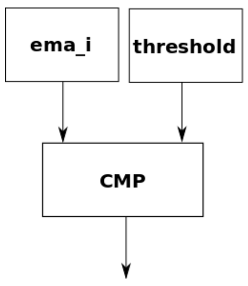
Figure 3.42

The threshold comparator compares the current filtered capacitance measurement value to the threshold register value. If it is larger than the threshold value, the result is *1*, else the result is *0*. Figure 3.43a shows the pseudocode for the threshold comparison. Figure 3.43b shows the architecture of the threshold comparator.

The threshold comparator is equivalent to the comparator used in the median-3 filter, described in section 3.7.2.

```
if (value > threshold) then
    o <- 1
else
    o <- 0
end
```

(a) Threshold Comparator Code



(b) Threshold Comparator Module Architecture

Chapter 4

Functional Verification

The synchronous and the asynchronous implementation of the capacitive digital touch detection filter have been functionally verified.

4.1 Method

A series of tests have been designed to verify the correct functionality of the synchronous and asynchronous implementation of the capacitive touch digital detection filter circuit. Both the synchronous and the asynchronous implementation are built up of a hierarchy of modules.

To test the correct behaviour of a circuit, it can be useful to test the circuit on more than one level of the hierarchy. Some modules may be more complex than others, and therefore it may be easier to test these on their own before they are tested as a part of a larger module.

4.2 Pad/RC Circuit Model

A SystemVerilog model has been developed for modelling the behaviour of a pad connected to an RC network. The pad includes a Schmitt-trigger [16] with a buffer. The path of the model is *src/synch/module/verilog/single_extres_model.sv*

Figure 4.1 shows the Pad/RC circuit model.

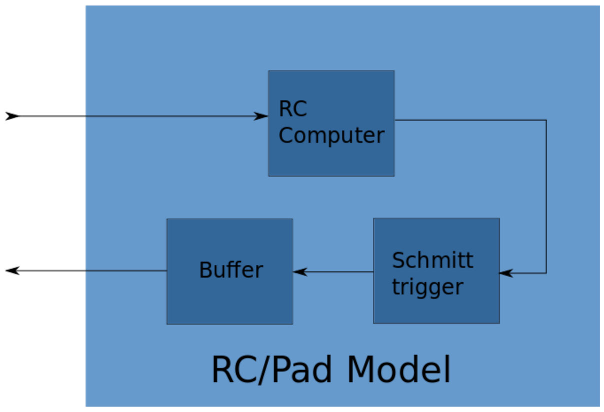


Figure 4.1: Pad/RC Circuit Model.

4.3 Testbench

The testbench is the platform which all tests are run on. It instantiates everything needed to perform a test, such as the circuit under test, the RC circuit model, clock generators and logging. The testbench can be setup to run individual tests, and the circuit under test can be the top level module of the circuit or a submodule. Figure 4.2 shows a block schematic of the testbench.

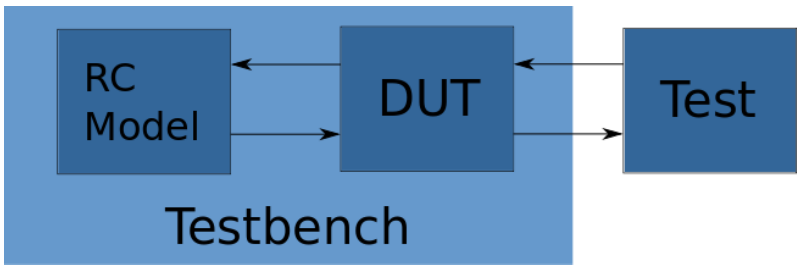


Figure 4.2: Testbench.

4.4 Tests

The tests are divided into two groups; top module tests and submodule tests. The top module tests are designed to simulate typical use of the circuit, while the submodule tests are designed to test for algorithmic errors in modules that do computations. [1]

[1]These tests do not cover all possible input combinations.

4.4.1 Top Module Tests

Two types of top module tests are performed on both implementations; a configuration test and a typical use test.

Configuration Test

This test is designed to test writing and reading from configuration registers. Figure 4.3 shows the configuration test sequence.

1. Typical values are written to all configuration registers.
2. All configuration registers are read and the output from the top module is compared to the values which were written.

Figure 4.3: Configuration Test Sequence

Sample, Filter and Threshold Comparison Test

This test is designed to test a typical scenario for the circuit. A typical scenario is to initiate 16 sample, filter and threshold comparison sequences periodically over the course of 1 second. A timer is used to initiate a new sequence with a frequency of 16 Hz. Figure 4.4 shows the typical test sequence. [2]

1. Typical values are written to all configuration registers.
2. 16 x Sample, filter and threshold comparison sequence.

Figure 4.4: Typical Test Sequence

While this test sequence gives typical behaviour, it does not say anything about the correctness of the circuit. This can be done by monitoring communication between modules in the circuit. The results from the sampler, median-3 filter, EMA filter and threshold comparison are the most important. Since these four modules are connected in a chain, it is possible to verify correct behaviour by monitoring change on control signals and associated data signals between them.

This monitoring can be done using probes. A probe in this case is just an alias of an internal signal in the circuit. Two probes are used for each channel that is monitored; one for the control signal and one for the data signal(s). The value of the data signal(s) are printed on negative edge of the control signal for the synchronous implementation, and on the positive edge of *REQ* (pull channel).

Listing 4.1 shows an example excerpt from where the channel between the sampler top module and the median top module is monitored.

[2]This test is also used for time based power estimation.

Listing 4.1: ../src/synch/standalone/sim/tests/top/top_tb.sv

```
  initial begin
    wait(enabled);
    forever begin
      @(negedge `ME_START);
      $display("medtopematop data %d", `ME_DATA);
    end
  end
  // Probe ematop_thcomptop data signal
`define ET_START tb.U_DUT.THCOMPTOP.ematop_thcomptop_start
`define ET_DATA tb.U_DUT.THCOMPTOP.ematop_thcomptop_data
```

4.4.2 Submodule Tests

The median-3 filter, EMA filter and threshold comparator perform computations. The results of these computations can be compared to values which are known to be correct, thus verifying the correctness of the computation.

Input vector stimuli for a module with matching correct output vectors is generated with a Python script. One Python script has been written for each of the three modules.

The paths for the Python scripts are:

```
src/python/med/med.py
src/python/ema/ema.py
src/python/thcomp/thcomp.py
```

Each Python script generates $N = 100$ input/output vector pairs.

The paths for the input/output vector pairs:

```
src/python/med/_input.dat
src/python/med/_output.dat
src/python/ema/_input.dat
src/python/ema/_output.dat
src/python/thcomp/_input.dat
src/python/thcomp/_output.dat
```

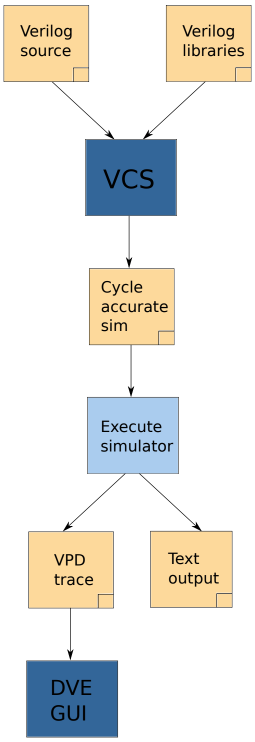
For each test, the corresponding *_input.dat* and *_output.dat* files are read. The test uses the input vector file as input stimuli to the module under test and the output vector file for comparison with the output vector from the module under test. If the output vector from the module under test matches the output vector read from the file, a match counter is incremented. If the value does not match, an error counter is incremented. If an error is encountered, the time, output value and expected value is written to a file, *_monitor.dat*. The number of matched output values and errors is also written to this file.

4.5 Simulation Flow

Figure 4.5a and 4.5b describes the simulation flow.

1. Describe modules in Verilog.
2. VCS takes a set of Verilog files as input and produces a simulator.
3. The simulator is executed.
4. The simulator generates textual trace information (using display statements in the Verilog code) or the simulator can be instructed to write transition information about each signal in the design to a file.
5. Open the generated VPD file in the DVE waveform viewer.

(a)



(b)

Figure 4.5: Simulation Flow

4.6 Submodule Simulation

4.6.1 Synchronous Implementation - Submodule RTL Simulation

The submodule RTL simulation of the synchronous implementation follows the flow described in section 4.4.2.

The testbenches are run with the commands:

```
make TEST=med_tb
make TEST=ema_tb
make TEST=thcomp_tb
```

The results from the tests are logged to:

```
src/synch/standalone/sim/tests/med/_monitor.dat
src/synch/standalone/sim/tests/ema/_monitor.dat
src/synch/standalone/sim/tests/thcomp/_monitor.dat
```

The result from the tests are:

```
median3_tb test started.
Matches :         100
Errors  :           0
median3_tb test finished.

ema_tb test started.
Matches :         100
Errors  :           0
ema_tb test finished.

thcomp_tb test started.
Matches :         100
Errors  :           0
thcomp_tb test finished.
```

4.6.2 Asynchronous Implementation - Submodule Breeze Simulation

[8, p. 77-98] shows how test harnesses can be built using Balsa. The test harnesses for submodule tests can be found in appendix A.3. They are run from *Balsa-manager*. Each test ran without errors.

4.7 Top Module Simulation

4.7.1 Synchronous Implementation - Top Module RTL Simulation

The testbench is run with the command:

```
make TEST=top_tb
```

The result from the test is logged to:

```
src/synch/standalone/sim/tests/top/_monitor.dat
```

The result from the test is:

```
top_tb test started.
top_tb test finished.
```

4.7.2 Synchronous Implementation - Top Module NTL Simulation

The top module NTL simulation of the synchronous implementation is run for each corner case using the following three commands:

```
src/synch/standalone/sim/tb/
make TEST=top_tb ntl CORNER=max
make TEST=top_tb ntl CORNER=typ
make TEST=top_tb ntl CORNER=min
```

The result from the test was logged to:

```
src/synch/standalone/sim/tests/top/_monitor.dat
```

The result from the test is:

```
top_tb test started.
top_tb test finished.
```

Sampling and Pad/RC Model Circuit

Figure 4.6 on page 55 shows signals from the sampler module and the pad/RC model module from the typical sequence test simulation of the top module. The $clk_{sampler}$ signal shows the sampler clock signal. *numsamples* shows the number of samples that is performed during the *charge/discharge* states. $state_r$ shows the sequence of the Finite State Machine (FSM). $counter_r$ shows that the counter counts from 255 to 0 in each of the *charge/discharge* states. *vc* shows the voltage over the capacitance in the pad/RC model module charging and discharging. $value_r$ shows that the *charge/discharge* time is increased until the hysteresis threshold of the transistors in the Schmitt-trigger is reached. $sense_{in}$ shows the threshold passing by toggling its value. $sense_{oe}$, $sense_{out}$,$drive_{oe}$ and $drive_{out}$ shows that the sampler module drives the inputs of the pad/RC model module correctly. This behaviour is similar to the RC oscillator shown in section 2.1.1.

4.7.3 Asynchronous Implementation - Top Module NTL Simulation

The top module NTL simulation of the asynchronous implementation is run for each corner case using the following three commands:

```
src/asynch/standalone/sim/tb/
make TEST=top_tb ntl CORNER=max
make TEST=top_tb ntl CORNER=typ
make TEST=top_tb ntl CORNER=min
```

When simulating the netlist of the asynchronous implementation for the *max* corner case ... reports no warnings, but when simulating the netlist of the asynchronous implementation for the *typ* and *min* corner cases .... reports timing violations. The number of timing violations increases from the *typ* to the *min* corner case.

Excerpt from *src/asynch/standalone/sim/tb/logs/top_tb.ntl.min.log*:

```
8243: Timing violation in tb.U_DUT.I23.I3.I3.I0
    $width( posedge G:53866 ns,  : 53867 ns, limit: 3 ns );

8243: Timing violation in tb.U_DUT.I23.I3.I4.I0
    $width( posedge G:53866 ns,  : 53867 ns, limit: 3 ns );
```

For the *min* corner case, timing violations are reported for all latches in the circuit. Figure 4.7 on page 56 shows an example waveform of signals connected to a latch and the *BrzVariable* component which surrounds the latch. Highlighted in the figure is the width of the pulse on the *G* input of the latch. The pulse width is 0.76ns. Even though that the pulse width is too short, the netlist (NTL) simulation works, since the simulation tool does not put an *X* or undefined value on the output, but only reports a warning.

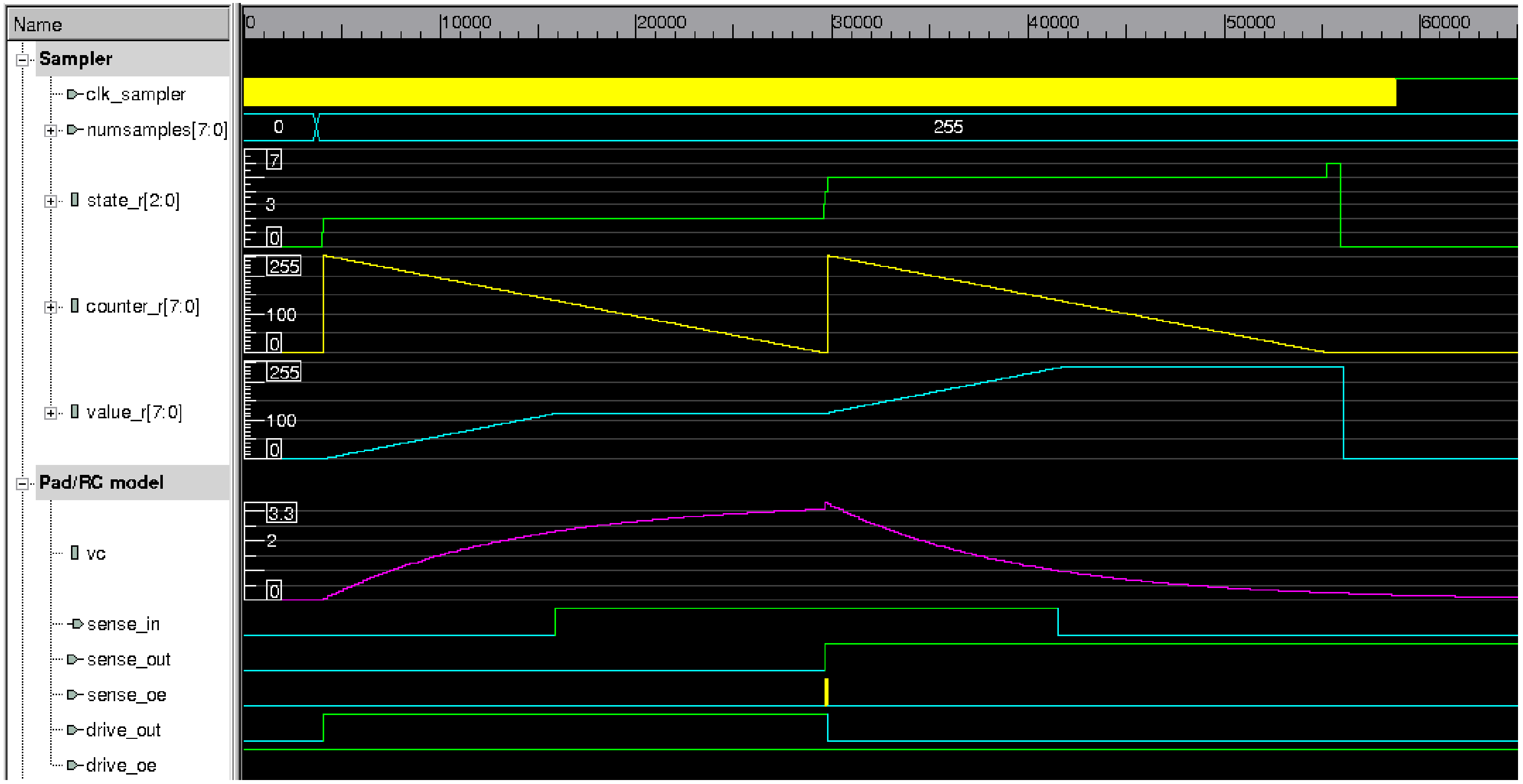


Figure 4.6: Sampler and Pad/RC Circuit Waveform.

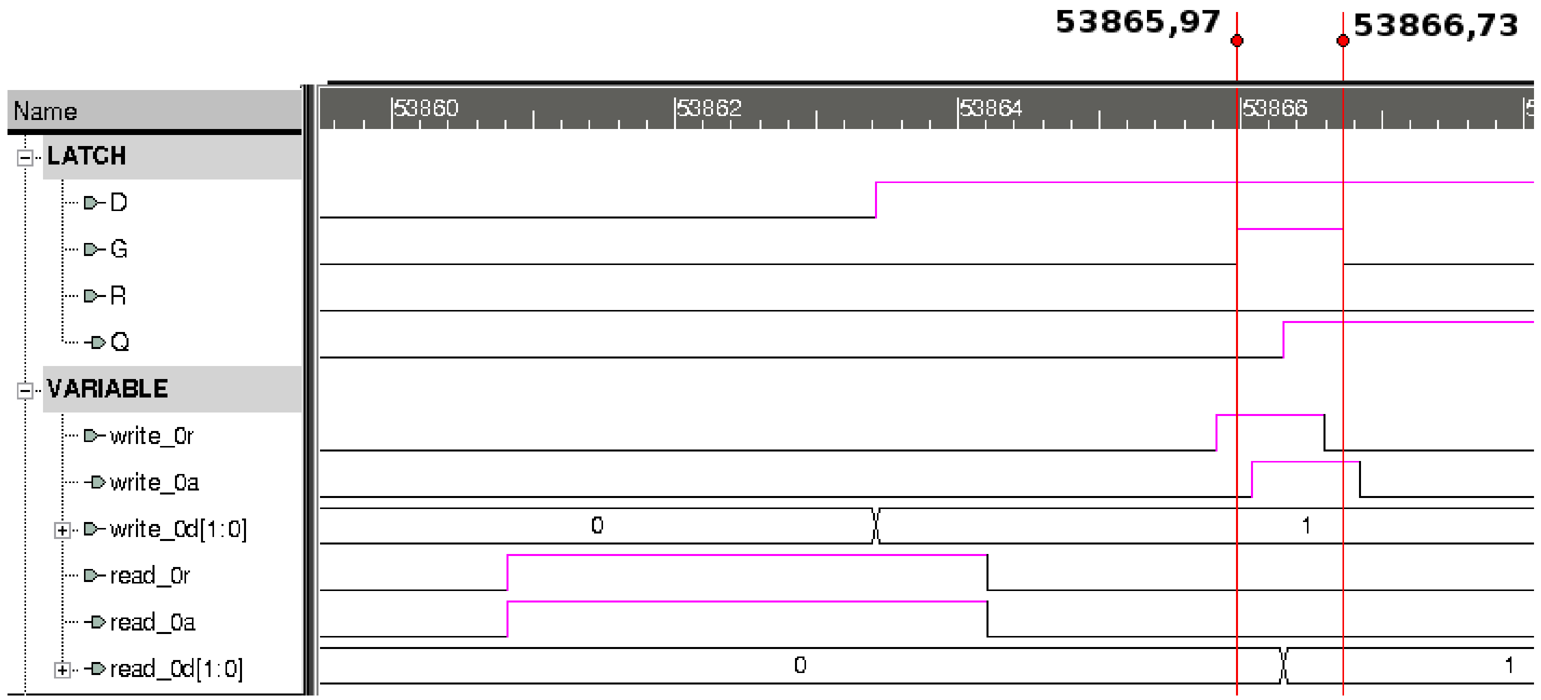


Figure 4.7: Latch and BrzVariable Waveform.

Chapter 5

Synthesis

This chapter describes the synthesis of the synchronous and the asynchronous implementation of the capacitive touch digital detection filter.

Both implementations are parametrised, and this allows for specifying the size of internal registers in the data path. Both implementations are synthesised for $N = 8$ bits, using the technology library specified in section 1.1.4.

Section 5.1 describes the synthesis and place & route (P&R) of the synchronous implementation, and section 5.2 describes the synthesis of the asynchronous implementation.

5.1 Synthesis of Synchronous Implementation

The synthesis of the synchronous implementation follows the flow described in section 5.1.1 using the tools listed in appendix D.2. The parameters for the synchronous implementation is found in appendix B.1.

5.1.1 Flow

Figures 5.1a and 5.1b show the flow for synthesis of the synchronous implementation.

5.1.2 Synthesis

The following script is used to do synthesis of the synchronous implementation:

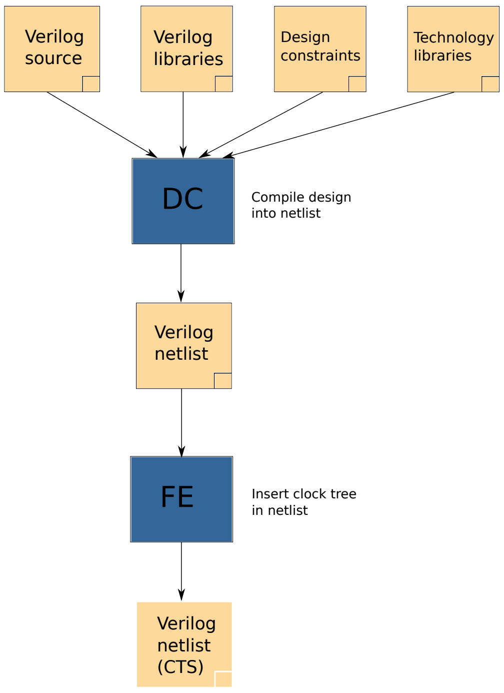
```
src/synch/standalone/synt/Makefile
```

The script is run with the command:

```
-> src/synch/standalone/synt/
make synt TOPO=0
```

1a Describe modules in Verilog.
1b Setup design constraints.
2 Compile Verilog code into Verilog netlist with Design Compiler (DC).
2a Analyse design.
2b Elaborate and write design.
2c Link.
2d Apply logical design constraints.
2e Put clock domains in path groups.
2f Setup clock gating style
2g Compile design
2h Save design
2i Generate reports (timing, clock gating).
3 Clock Tree Synthesis (CTS) with First Encounter (FE).
3a Load chip config data.
3b Load timing constraints.
3c Setup/generate floorplan of chip.
3d Place design on chip.
3e Optimise design before clock tree synthesis.
3f Create Clock tree specification.
3g Generate clock tree.
3h Optimise design after CTS for hold time.
3i Optimise design after CTS for design rule violations
3j Save design
3k Save netlist

(a)



(b)

Figure 5.1: Synchronous Synthesis Flow.

The synthesise report gives the following warnings:

```
Warning:  ../../module/verilog/ema.v:66: unsigned to signed assignment occurs. (VER-318)
Warning:  ../../module/verilog/ema.v:87: signed to unsigned assignment occurs. (VER-318)
```

Both warnings can safely be ignored due to the assumptions described in section 3.9.1.

The synthesis report path is *src/synch/standalone/synt/synt_logs/synt.log*.

5.1.3 Quick P&R

The following script is used to do a quick *place&route* and insert the clock tree into the netlist:

```
src/synch/standalone/quick_fe/quick_cts.tcl
```

The script is run with the command:

```
-> src/synch/standalone/quick_fe/
make quick_cts
```

Figure 5.2 shows the physical layout of the netlist after inserting the clock tree.

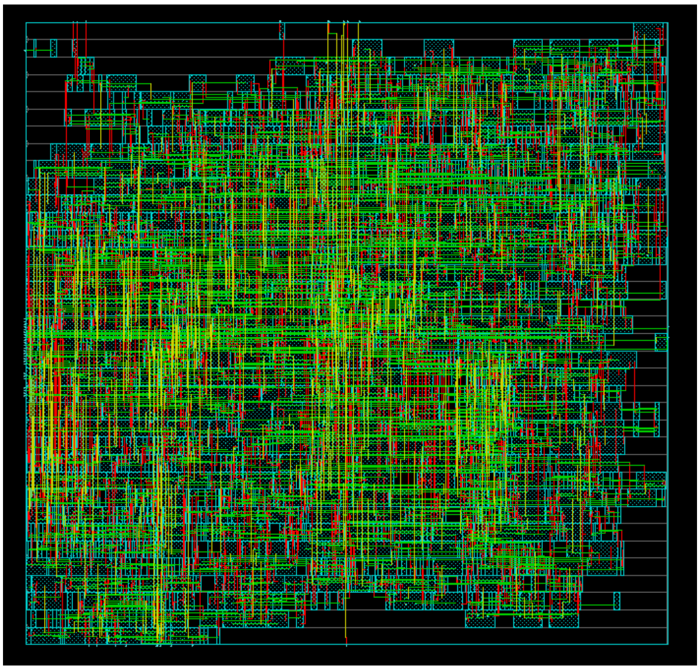


Figure 5.2: Physical Layout After Clock Tree Synthesis.

5.2 Synthesis of Asynchronous Implementation

The synthesis of the asynchronous implementation follows the design flow described in section 2.5.1 using the tools listed in section D.3.

The following Makefile is used to synthesise a netlist from the Balsa description files.

```
-> src/asynch/module/balsa/Makefile
```

The script is run with the command:

```
-> src/asynch/module/balsa
make impl-top-impl4phbd
```

The synthesis gives no warnings or errors.

The synthesis report path is *src/asynch/module/balsa/impl-top-impl4phbd.log*.

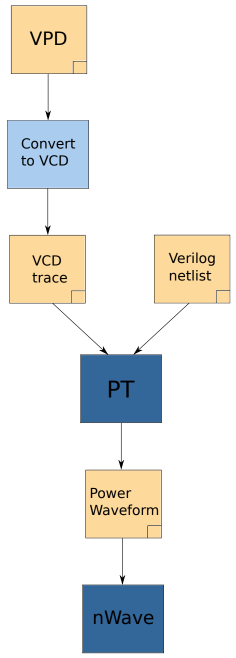
Chapter 6

Time-Based Power Estimation

Figures 6.1a and 6.1b show the flow for doing time based power estimation. PrimeTime (PT) is used to estimate the power consumption of the synchronous and the asynchronous implementation from a time-based simulation. PT can take a Synopsys Binary Parasitics Format (SBPF) file, which has information about parasitics in the synthesised circuit. However, only synthesis of a circuit using DC can generate an SBPF, while synthesis using Balsa Asynchronous Synthesis System can not. Instead wire load models are used. A wire load model uses statistical information to estimate the wire load based on the number of fan-out pins on a net. This information is stored in an Standard Delay Format (SDF) file. Using wire load models is less accurate, but using it for both implementations is more fair when the aim is to compare them on even grounds.

1 Convert VPD to VCD.
2 Write an SDF and PT session for each corner.
3 Open FSDB file in nWave.

(a)



(b)

Figure 6.1: Time Based Power Estimation Flow.

6.1 Script for Generating SDF for all corners

```
standalone/quick_fe/make_sdf.tcl
```

```
standalone/quick_fe
```

6.2 Script Sequence for Converting VPD to VCD

Convert VPD to VCD and compress for all corners:

```
-> standalone/sim/tb/
make logs/top_tb.ntl.max.vcd.gz
make logs/top_tb.ntl.typ.vcd.gz
make logs/top_tb.ntl.min.vcd.gz
```

6.3 Script Sequence for Time Based Power Estimation

Run time based power estimation and report cell area for all corners:

```
-> standalone/pwr/
make power_vcd CORNER=max
make power_vcd CORNER=typ
make power_vcd CORNER=min
```

Chapter 7

Results

Sections 7.1, 7.2 and 7.3 present cell area cost results, power estimation results and emission results.

7.1 Cell Area Cost

PT reports cell area. Cell area is measured in terms of the smallest two-input NAND gate in the technology library that is used.

Table 7.1 shows the cell area cost of the synchronous and the asynchronous implementation.

Implementation	Cell Area
Synchronous	150754
Asynchronous	308985

Table 7.1: Implementation Cost

7.2 Power Consumption

7.2.1 Power Waveform

The power waveform is used to compare the power consumption in the synchronous and asynchronous implementation, and to show when the submodules of the top module in the two implementations contribute to the power consumption.

Figures 7.1 and 7.2 show the power consumption of the top module and its submodules for the synchronous and the asynchronous implementation over one sample, filter and threshold comparison sequence.

Figure 7.3 shows a comparison of the total power waveforms of the synchronous and the asynchronous implementation for each corner case.

7.2.2 Power Distribution and Power Density

Figures 7.4 and 7.5 show what contribute most to the power consumption in the two implementations.

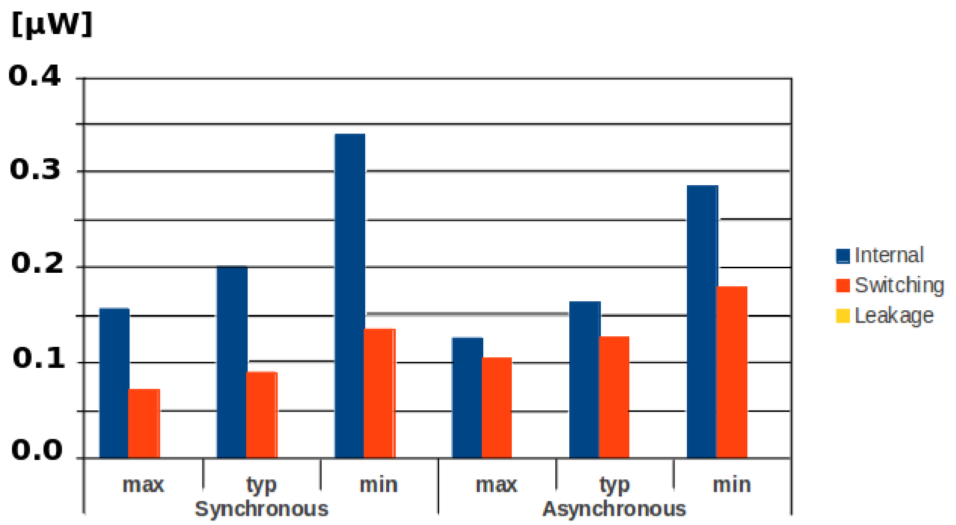


Figure 7.4: Power Distribution.

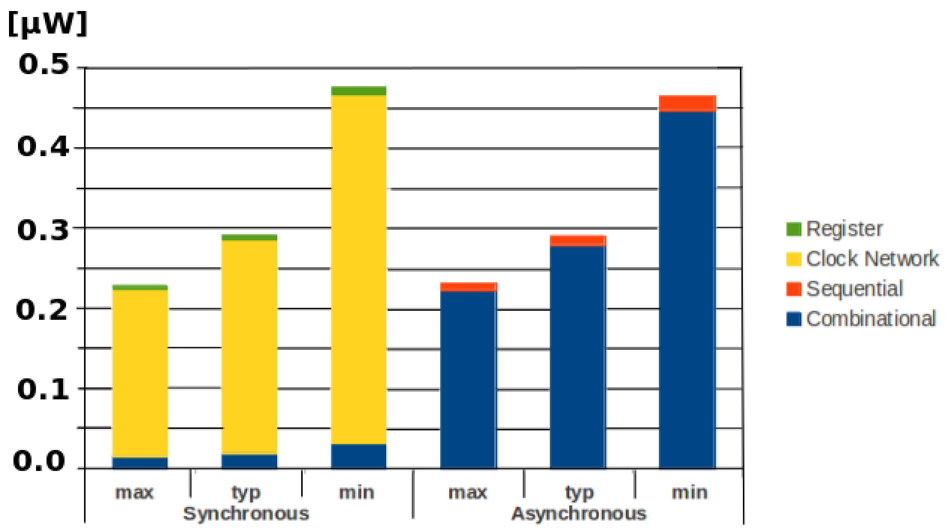


Figure 7.5: Power Distribution.

Figures 7.6 and 7.7 show which modules that have the largest power density in the two implementations.

7.2.3 Average Total Power

When doing 16 sequences per second:

```
Synch:
max: 228.5 [nW]
typ: 291.3 [nW]
min: 475.4 [nW]

Asynch (4phbd):
max: 231.1 [nW]
typ: 290.2 [nW]
min: 464.9 [nW]
```

7.2.4 Energy/sequence

Equation 7.1 shows how to convert total average power into energy per sequence.

$$E/\text{seq} = \frac{P_{avgtot} * t_{tot}}{N_{seq}}[\text{J/seq}] \tag{7.1}$$

When doing 16 sequences per second:

```
Synch:
max: (228.5e-9)*1/16 = 14.3 [nJ/seq]
typ: (291.3e-9)*1/16 = 18.2 [nJ/seq]
min: (475.4e-9)*1/16 = 29.7 [nJ/seq]

Asynch (4phbd):
max: (231.1e-9)*1/16 = 14.4 [nJ/seq]
typ: (290.2e-9)*1/16 = 18.1 [nJ/seq]
min: (464.9e-9)*1/16 = 29.1 [nJ/seq]
```

7.2.5 Average Power In Active Mode

Equation 7.2 shows how to convert average total power into average power in active mode.[1]

$$P_{avgact} = \frac{\frac{P_{avgtot}}{t_{tot}} * t_{seq}}{N_{seq}}[\text{W}] \tag{7.2}$$

```
Synch:
```

[1]The active time is the same for the three corners for the synchronous implementation, while it is different for the three corners for the asynchronous implementation.

```
max: ((2.285e-7)/e9)*587500/16 =  8.390 [pW]
typ: ((2.913e-7)/e9)*587500/16 = 10.697 [pW]
min: ((4.754e-7)/e9)*587500/16 = 17.456 [pW]

Asynch (4 Phase):
max: ((2.311e-7)/e9)*547000/16 =  7.900 [pW]
typ: ((2.902e-7)/e9)*544000/16 =  9.867 [pW]
min: ((4.649e-7)/e9)*541500/16 = 15.734 [pW]
```

7.2.6 Peak Power

The peak power says what the maximum power was during a simulation using typical computing values. Figure 7.8 shows the peak power consumption of the synchronous and the asynchronous implementation for each corner case.

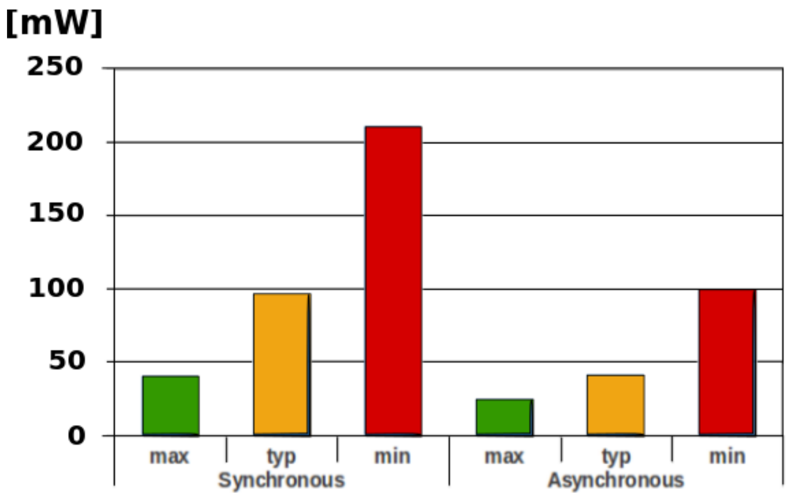


Figure 7.8: Peak Power.

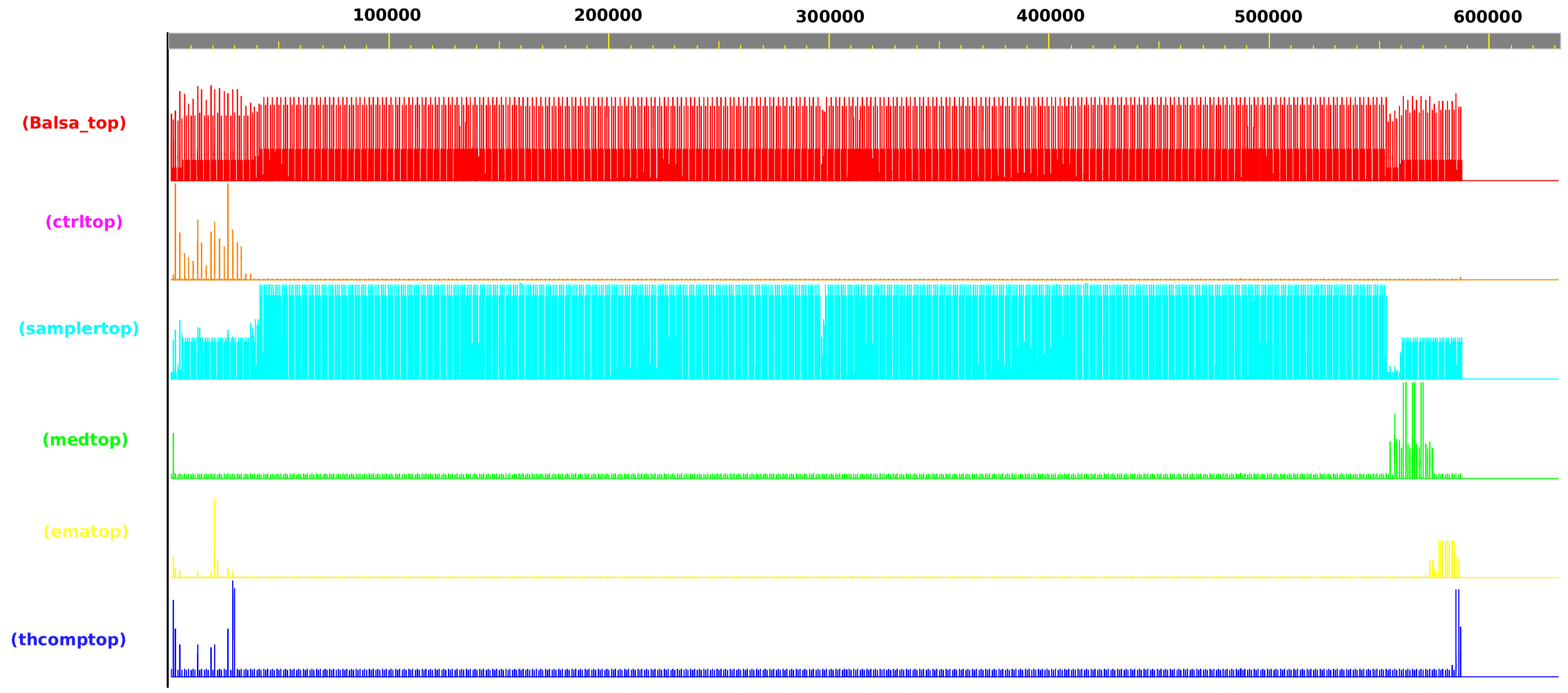


Figure 7.1: Synchronous Implementation - Power Consumption Waveform.

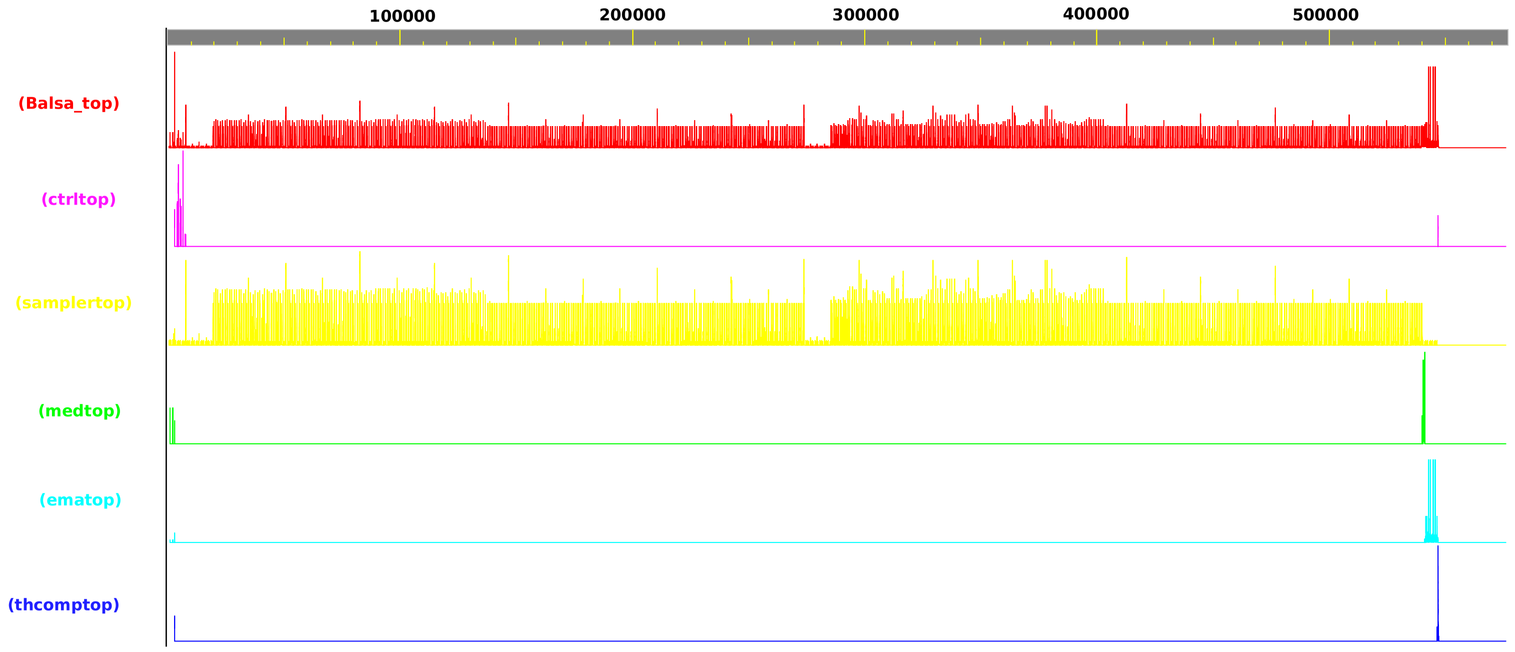


Figure 7.2: Asynchronous Implementation - Power Consumption Waveform.

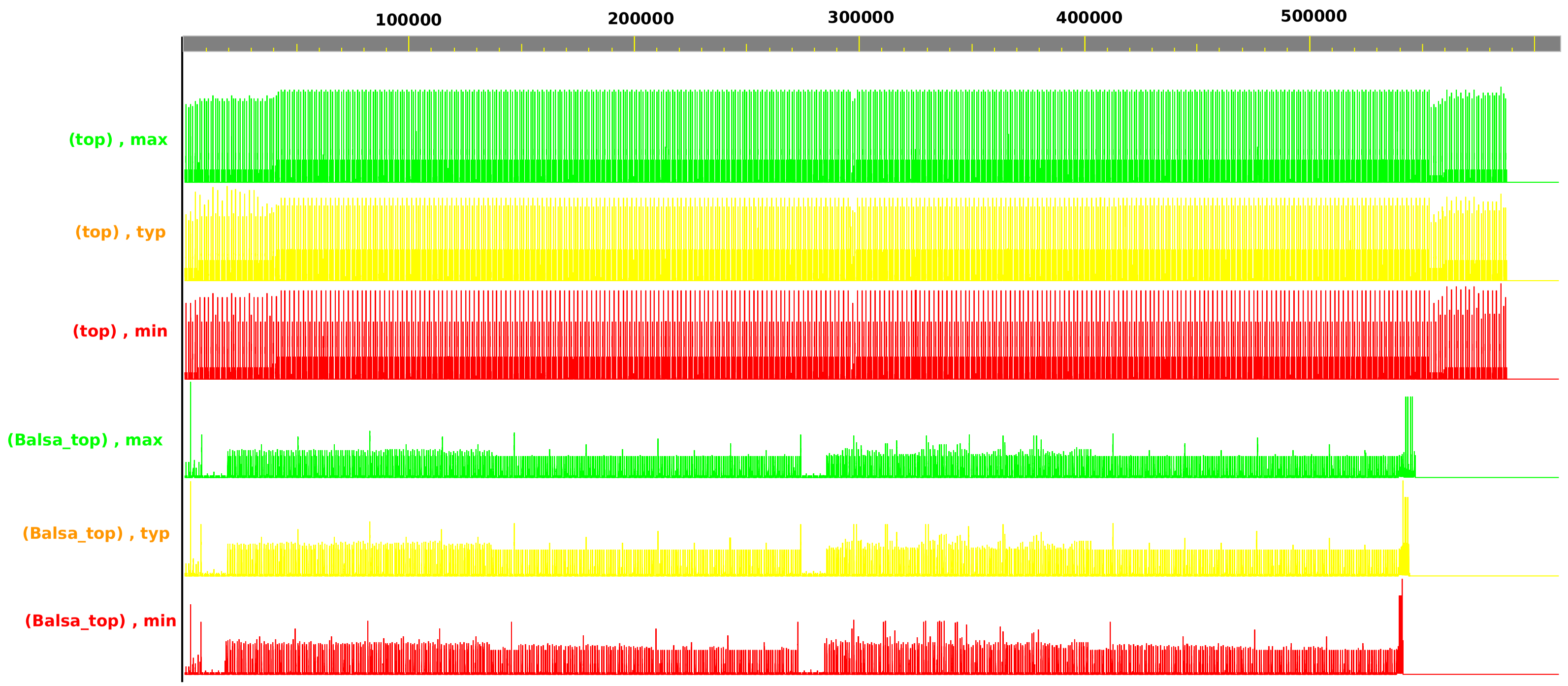


Figure 7.3: Total Power Comparison Waveform.

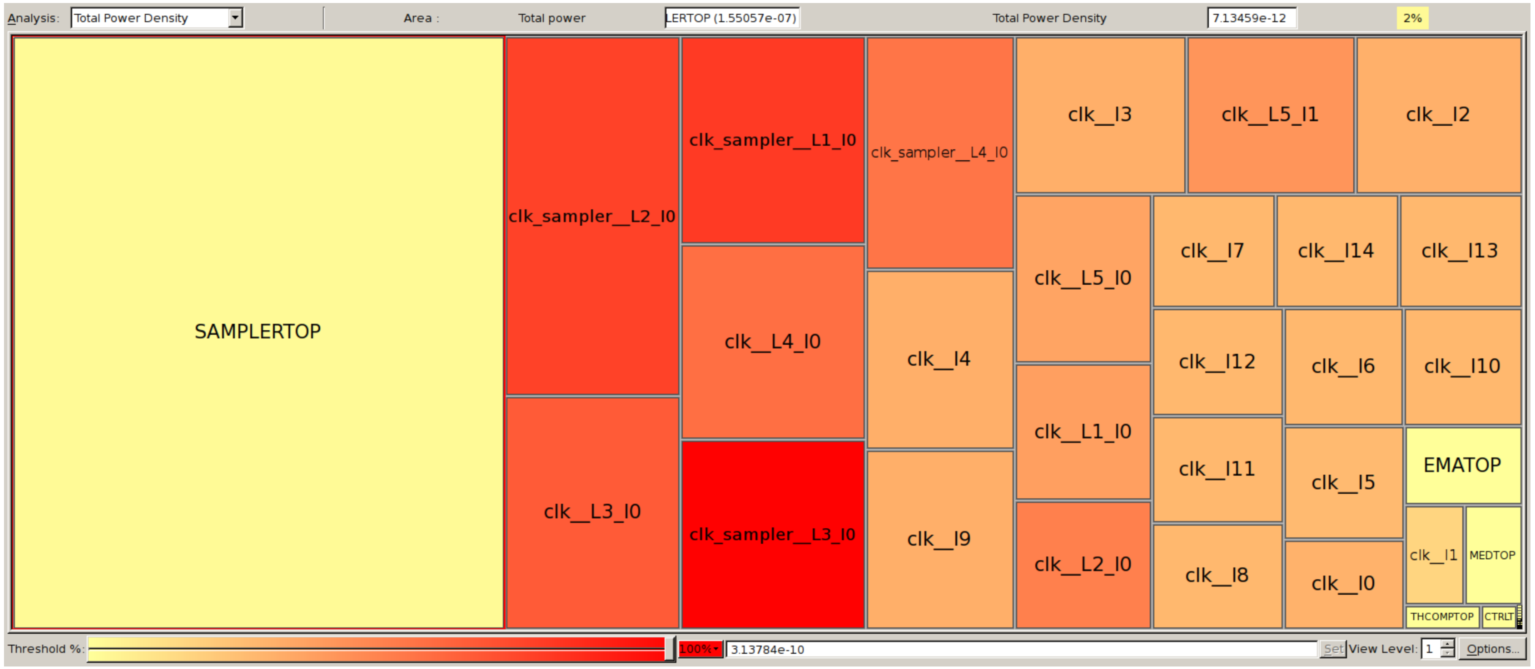
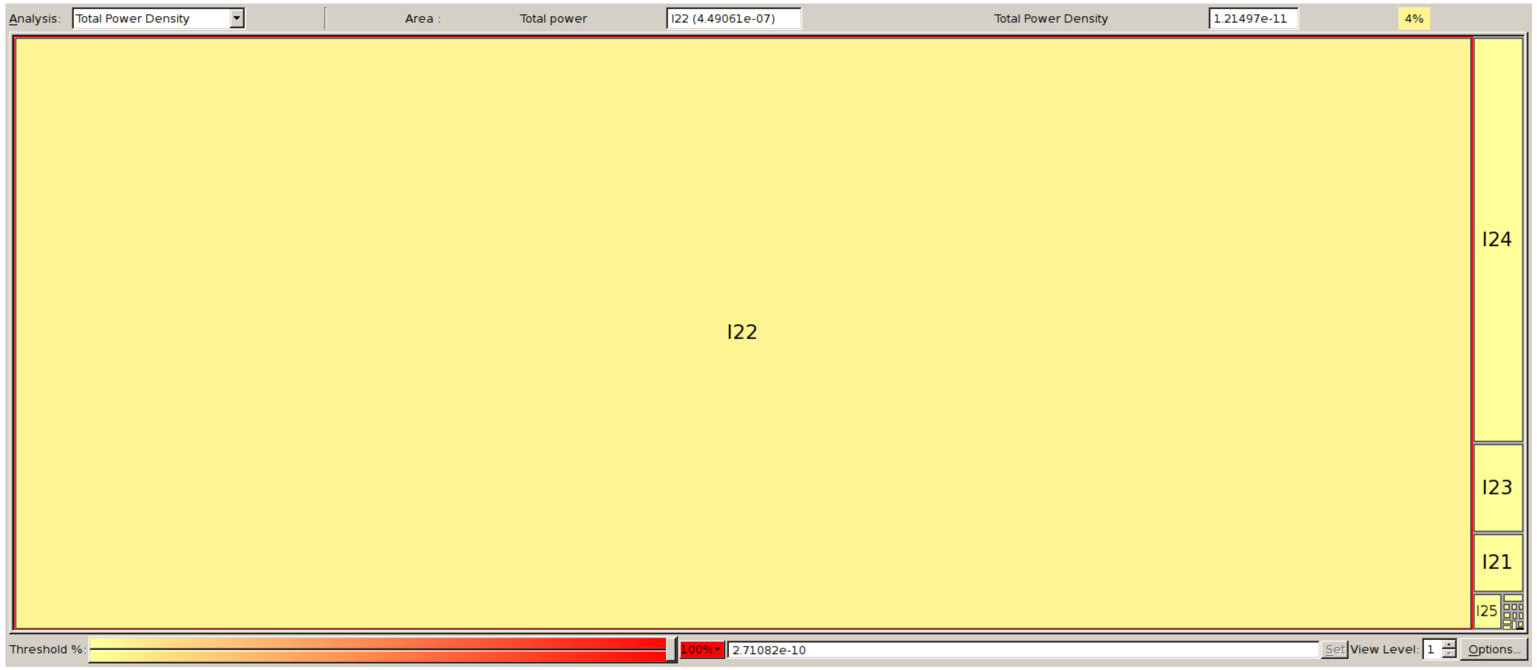

Analysis: Total Power Density
Area :
Total power LERTOP (1.55057e-07)
Total Power Density 7.13459e-12 2%
Threshold %: 100% 3.13784e-10
View Level: 1

Figure 7.6: Power Density - Synchronous Implementation.


Analysis: Total Power Density

Area :

Total power I22 (4.49061e-07)

Total Power Density 1.21497e-11

4%

Threshold %: 100% 2.71082e-10

View Level: 1

Figure 7.7: Power Density - Asynchronous Implementation.

7.3 Emission

The emission from the circuit was found by performing a Discrete Fourier Transform (DFT) on the time-based power simulation.

The Fast Signal Database (FSDB) file from PT is opened with nWaven. nWave performs a DFT on the time-based power waveform. The DFT uses a Hamming window [13, p. 622-628] and the maximum allowed sample rate (>2 the maximum frequency).

Figure 7.9 shows the result of performing a DFT of the time-based power simulation for the synchronous implementation during one sample, filter and threshold comparison sequence. Figure 7.10 shows the result of a DFT of the time-based power simulation for the asynchronous implementation during one sample, filter and threshold comparison sequence.

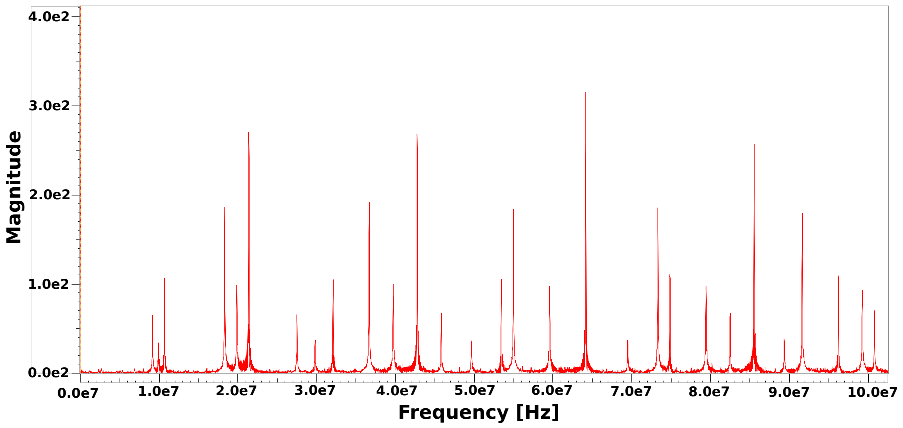


Figure 7.9: DFT of Power - Synchronous Implementation.

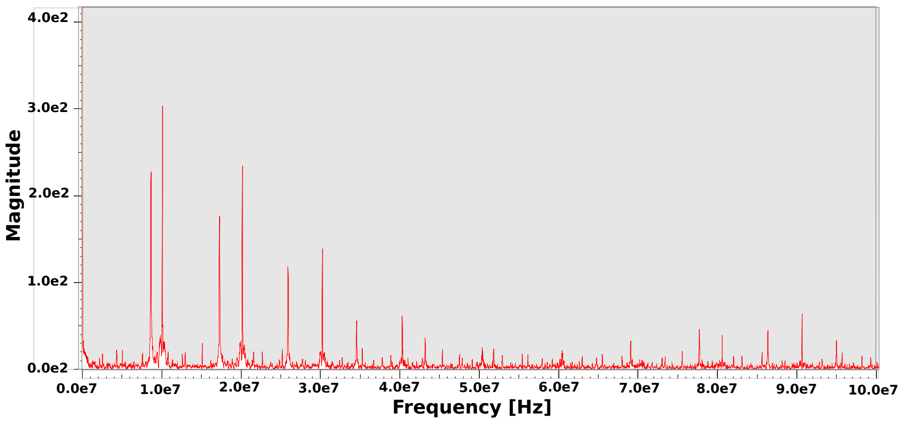


Figure 7.10: DFT of Power - Asynchronous Implementation.

Chapter 8

Discussion

8.1 Results

8.1.1 Power Consumption

The time based power estimation shows that the power consumption of the synchronous and the asynchronous implementation are similar. It also shows that >90% of the power in the synchronous implementation goes into the clock network, while >90% of the power consumption in the asynchronous implementation goes into combinatorial logic. This is because the power consumption is dominated by activity in the sampler module.

This result is not in accordance with the claims of low power consumption in [9, p. 3-5], but it also notes that the designer must have much experience with designing asynchronous circuits and the application must show characteristics that can benefit from using a asynchronous methodologies.

The results from using a 4-phase BD handshake protocol can be biased by low performing handshake components [5, p. 45]. The results could improve if a 2-phase BD handshake protocol was used instead.

8.1.2 Area Cost

The overhead of the control logic in the asynchronous implementation led to twice the area of the synchronous implementation. This result is in accordance with the claims in [9, p. 3-5].

8.1.3 Emission

The emission is analysed by looking at the frequency components in the time-based power simulation. The DFT of the synchronous shows two major frequency components at $\tilde{1}$0MHz and $\tilde{2}$0MHz, and their harmonies. The DFT of the asynchronous shows one major frequency component at $\tilde{1}$0MHz and its harmonies, while the rest of the frequency content has a randomised characteristic. The asynchronous implementation shows a significant improvement over the synchronous implementation in terms of lower frequency components. This result is in accordance with the claims in [9, p. 3-5].

8.2 Design Optimisation

It is possible to improve the performance of the design by implementing a variety of optimisations.

8.2.1 Data Path

Reducing the Number Range

If a constant value is subtracted from each *charge+discharge* time measurement from the sampler module, the number range for the filter calculations can be reduced. This would introduce an additional subtraction operation. Instead of doing the subtraction after receiving the value from the sampler module, the subtraction can be performed by loading the result register for the *charge+discharge* time measurement with a negative value before doing the additions related to the sampling. In this way, the number range can be reduced without adding much extra logic. [1]

Dividing the Number Range

The data path can be optimised by dividing it into a high and low number range, where the high range is only active when interesting stuff happens (like a touch) and the low range is for processing noise when nothing interesting happens. By keeping half of the data path inactive, the power consumption could in theory almost be reduced to a half.

[1]The register for the constant to be subtracted and the load operation has been implemented, but the register is set to zero and the rest of the data path still use the full number range.

Recurring Structures

The median-3 filter, the EMA filter and the threshold comparator all use full-adder chain structures for addition and subtraction. All three use 2's complement number representation and N-bit length unsigned vectors, with the exception of the EMA filter which uses (N+1)-bit length signed vectors.

The multiplicator in the EMA filter has been implemented using a combination of adder and shift structures. Apart from the shift operations and conditional additions, the adder portion of the multiplicator could share adder structure with other operations. In this circuit the multiplicator has been implemented using (2*N)-bit length intermediate registers[2].

There is a potential for saving area by combining these structures into a architecture more resembling a processor architecture. An architecture with sharing of hardware leads to more control signals and a more complex architecture. Whether it is beneficial to choose such an architecture with regards to saving power and area as opposed to the chosen implementation remains unanswered.

8.3 Verification

8.3.1 Simulation Time

Simulation of an asynchronous circuit goes fast if the circuit is idle, since it does not generate new events to the simulator queue. An asynchronous circuit only contributes to dynamic power consumption when it is not idle. Thus it is possible to get a feel for the dynamic power consumption of an asynchronous circuit if the simulation is fast. This is would be more discernible in a larger design, when synthesising for large register sizes/vector lengths.

Simulation of a synchronous circuit can be almost as fast, if the clock gating is good enough. The synthesis log shows that the current clock gating style is set to using a minimum register bank size of 3.

8.3.2 Timing Violations

No delay matching has been done for the 4-phase BD handshake protocol used in the asynchronous implementation. The BD protocol is relies on the timing assumption that the circuit is not faster than the control logic. Section 4.7.3 shows that the *typ* and *min* corner cases result in timing violations, while the *max* corner case does not. This is because the *typ* and *min* corner run faster than the *max* corner case. The timing violations are related to setup and hold-time violations of latches used in the *BrzVariable* Balsa handshake component. This can be solved by either inserting delays (e.g. inverter(s)) in the control signal path, thus widening the pulse width, or a faster

[2]It is possible to use N-bit intermediate registers, but the signed portion of the multiplication becomes a bit more complex due to the subtraction for the sign bit.

latch must be designed. Since, the circuit is almost 4 times faster than the latch, the former is the most feasible option. An other option is to use a delay-insensitive DR protocol instead of the BD protocol.

Chapter 9

Conclusion

9.1 Conclusions Drawn from This Thesis

For a capacitive touch digital detection filter circuit, a straight forward asynchronous implementation using Balsa lead to similar power consumption (sec. 7.2) to the synchronous implementation. It was claimed that using asynchronous methodologies would result in lower power consumption [9, p. 3-5] , but this was not the case for the chosen implementation. This is because the number of switchings in the sampler module dominates the power consumption, as shown in section 7.2.2.

The asynchronous implementation shows less peak power consumption (sec. 7.2.6) and less harmonics in the emission spectrum (sec. 7.3) than the synchronous implementation. This supports the claim that asynchronous methodologies lead to less emission than synchronous methodologies [9, p. 3-5].

The asynchronous implementation uses twice as much area as the synchronous implementation. This supports the claim that asynchronous methodologies lead to higher area cost than synchronous methodologies [9, p. 3-5].

9.2 Summary of the Contributions this Thesis Has Made

In this thesis, both synchronous and asynchronous implementation methods have been explored for implementing a capacitive touch digital detection filter circuit. An asynchronous implementation of a capacitive touch digital detection filter circuit has never before been published.

Comparisons between the synchronous and asynchronous implementation show that asynchronous circuit methodologies do not automatically lead to low power consumption, but can lead to larger area cost and lower emission. These results lead to a better understanding of the possibilities and limitations of asynchronous methodologies. Thus, this thesis can be used as a reference when other designers evaluate whether their application will benefit from using asynchronous methodologies.

In addition, new approaches for interfacing an asynchronous circuit, described in Balsa, with an analog circuit has been found (sec. 3.5.3). This also covers an approach for implementing a variable speed sampler clock with a minimum fixed sample period.

9.3 Prospect of Further Research

The approaches for interfacing an asynchronous circuit with an analog circuit can be implemented and tested.

The design and test of the delay element for the variable speed sampler clock can be done.

Startup time and power consumption of synchronous vs. asynchronous oscillator can be researched.

The data path of the capacative touch digital detection filter can be optimised by employing a reduced range dual bank data path.

The Balsa Asynchronous Synthesis System makes it possible to synthesise for different handshake protocol, e.g. 2-phase BD, and 2-phase and 4-phase DR protocols. Research into combinations of different handshake protocols and optimizations to existing or completely new handshake libraries is possible.

Section 8.3.2 discusses possible solutions for solving the timing violations from the simulation of the asynchronous implementations NTL. Automatic delay insertion based on static timing analysis is missing from Balsa Asynchronous Synthesis System design flow.

Interfacing the capacative touch digital detection filter circuit with a microcontroller. The microcontroller can be responsible for timing the start of new capacitance measurements, by using a low power slow oscillator.

The technology used is old and is characterised by high threshold and high voltage. New technology comes with a new set of challenges, but can benefit from lower total area cost and less dynamic power consumption.

Bibliography

[1] Advanced Processor Technologies Group Balsa Project Site. `http://apt.cs.manchester.ac.uk/projects/tools/balsa/`.

[2] Balsa Freecode Project Site. `http://freecode.com/projects/balsaasync`.

[3] Balsa FTP Download Site. `ftp://ftp.cs.man.ac.uk/pub/apt/balsa/4.0/`.

[4] Financial Times. `http://www.ft.com/intl/cms/s/2/6ecd4a0a-280a-11e1-a4c4-00144feabdc0.htm`.

[5] Bjarne Drotningshaug. Design of Asynchronous Circuits with Focus on Low Power Consumption using the Balsa Synthesis System, 2011.

[6] C. N.Marimuthu, Dr. P. Thangaraj, Aswathy Ramesan. Low Power Shift And Add Multiplier Design. `http://arxiv.org/ftp/arxiv/papers/1006/1006.1179.pdf`, 2010.

[7] D.E. Muller an W.S. Bartky. A Theory of Asynchronous Circuits. *Proc. of an International Symposium on the Theory of Switching*, pages 204–243, 1959.

[8] Doug Edwards, Andrew Bardsley, Lilian Janin, Luis Plana and Will Toms. Balsa: A Tutorial Guide. `ftp://ftp.cs.man.ac.uk/pub/apt/balsa/3.5/BalsaManual3.5.pdf`, 2006.

[9] J. Sparso. Asynchronous Circuit Design: A Tutorial. `http://www.imm.dtu.dk/pubdb/views/publicationdetails.php?id=855`, 2006.

[10] James H. McClellan, Ronald W. Schafer, Mark A. Yoder. Signal Processing First, 2003.

[11] J.D. Garside, W.J. Bainbridge, A. Bardsley, D.A. Edwards, S.B. Furber, J. Liu, D.W. Lloyd, S. Mohammadi, J.S. Pepper, O. Petlin, S. Temple, J.V. Woods. AMULET3i – an Asynchronous System-on-Chip. *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 162–175, April 2000.

[12] Jeroen Domburg. AVR-based FM-transmitter. `http://spritesmods.com/?art=avrfmtx`.

[13] John G. Proakis, Dimitris G. Manolakis. Digital Signal Processing - Principles, Algorithms and Applications (Fourth Edition). , 2007.

[14] Larry K. Baxter. Capacitive Sensors: Design and Applications, 1996.

[15] LS Nielsen, J. Sparso. Designing Asynchronous Circuits for Low Power: An IFIR Filter Bank for a Digital Hearing Aid. `http://www.eng.utah.edu/~cs5830/handouts/00740020.pdf`, 1999.

[16] Otto H. Schmitt. A Thermionic Trigger. Journal of Scientific Instruments, Volume 15, Number 1, 1938.

[17] Ran Ginosar. Fourteen Ways to Fool Your Synchronizer. Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems, 2003.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms, 2009.

[19] Truls Magnus Aamodt Gulbrandsen. Capacative Touch Digital Detection Filter Circuit, 2011. Specialization Report for Master Thesis.

Appendix A

Balsa Code

The following sections show Balsa code listings.

A.1 Parameters

Listing A.1: params.balsa

```
-- params.balsa
constant N = 10
constant M = 4
constant NUM_BITS = 8
constant EMA_FRACBITS = NUM_BITS
constant MSB = NUM_BITS-1

type regdata is NUM_BITS bits
type mode is enumeration READ, WRITE end
type addr is (log N) bits
type cntr is (log N) bits


type in_bundle is record
    data : regdata;
    mode : mode;
    addr : addr
end

type out_bundle is record
    data : regdata
end

type samplertop_sampler_bd is record
    numsamples, subvalue : regdata
end

type sensedrive_bd is record
    sense_oe, sense_out, drive_oe, drive_out : bit
end

```

```
type thcomptop_thcomp_bd is record
    value, threshold : regdata
end
```

A.2 Modules

Listing A.2: top.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]
import [regs]
import [samplertop]
import [medtop]
import [ematop]
import [thcomptop]
import [ctrltop]

procedure top (
  -- Input(s)
  sync top_ctrltop_start;
  input top_ctrltop : in_bundle;
  input sense_in : bit;
  -- Output(s)
  output sensedrive_out : sensedrive_bd;
  output ctrltop_top : regdata;
  output ctrltop_top_start : bit
) is
    -- Channel(s)
    sync ctrltop_samplertop
    channel samplertop_medtop : regdata
    channel medtop_ematop : regdata
    channel ematop_thcomptop : regdata
    channel thcomptop_ctrltop : bit
    array N of sync reg_r
    array N of sync cfg_r
    array N-M of channel reg_data_in : regdata
    array M of channel cfg_data_in : regdata
    array N of channel reg_data_out : regdata
    array N of channel cfg_data_out : regdata
begin
    regs(
      -- Input(s)
      reg_r,
      cfg_r,
      reg_data_in,
      cfg_data_in,
      -- Output(s)
      reg_data_out,
      cfg_data_out
    )
 ||
    samplertop(
      -- Input(s)
      ctrltop_samplertop,
      sense_in,
      -- Output(s)
      {reg_r[0], reg_r[1]},
      {reg_data_out[0], reg_data_out[1]},
      sensedrive_out,
      samplertop_medtop
    )
  ||
    medtop(
      -- Input(s)
      samplertop_medtop,
      {reg_data_out[4], reg_data_out[5], reg_data_out[6]},
      -- Output(s)
      {reg_r[4], reg_r[5], reg_r[6]},
      {reg_data_in[0], reg_data_in[1], reg_data_in[2]},
      medtop_ematop
    )
  ||
    ematop(
      -- Input(s)
      medtop_ematop,
      {reg_data_out[7], reg_data_out[8], reg_data_out[2]},
      -- Output(s)
      {reg_r[7], reg_r[8], reg_r[2]},
      {reg_data_in[3], reg_data_in[4]},
      ematop_thcomptop
    )
  ||
    thcomptop(
      -- Input(s)
      ematop_thcomptop,
      {reg_data_out[9], reg_data_out[3]},
      -- Output(s)
      {reg_r[9], reg_r[3]},
      reg_data_in[5],
      thcomptop_ctrltop
    )
  ||
    ctrltop(
      -- Input(s)
      top_ctrltop_start,
      top_ctrltop,
      thcomptop_ctrltop,
      cfg_data_out,
      -- Output(s)
      ctrltop_samplertop,
      ctrltop_top,
      ctrltop_top_start,
      cfg_r,
      cfg_data_in
    )
end -- procedure top
```

Listing A.3: ctrltop.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]

procedure ctrltop (
  -- Input(s)
  sync top_ctrltop_start;
  input top_ctrltop : in_bundle;
  input thcomptop_ctrltop : bit;
  array N of input cfg_data_out : regdata;
  -- Output(s)
  sync ctrltop_samplertop;
  output ctrltop_top : regdata;
  output ctrltop_top_start : bit;
  array N of sync cfg_r;
  array M of output cfg_data_in : regdata
) is
begin
    loop
        select top_ctrltop_start then
            [
              sync ctrltop_samplertop
           ||
              select thcomptop_ctrltop then
                  [
                    ctrltop_top_start <-  thcomptop_ctrltop
                  ]
              end -- select thcomptop_ctrltop
            ]
        end
    end -- select top_ctrltop_start
 ||
    loop
        select top_ctrltop then
            case (top_ctrltop.mode as mode) of
            WRITE then
                case (top_ctrltop.addr as addr) of
                    0 then cfg_data_in[0] <- top_ctrltop.data |
                    1 then cfg_data_in[1] <- top_ctrltop.data |
                    2 then cfg_data_in[2] <- top_ctrltop.data
                      else cfg_data_in[3] <- top_ctrltop.data
                end -- case ( top_ctrltop.addr as 2 bits )
            |
                READ then
                case (top_ctrltop.addr as addr) of
                    0 then cfg_data_out[0] -> ctrltop_top || sync
                        cfg_r[0] |
                    1 then cfg_data_out[1] -> ctrltop_top || sync
                        cfg_r[1] |
                    2 then cfg_data_out[2] -> ctrltop_top || sync
                        cfg_r[2] |
                    3 then cfg_data_out[3] -> ctrltop_top || sync
                        cfg_r[3] |
                    4 then cfg_data_out[4] -> ctrltop_top || sync
                        cfg_r[4] |
                    5 then cfg_data_out[5] -> ctrltop_top || sync
                        cfg_r[5] |
                    6 then cfg_data_out[6] -> ctrltop_top || sync
                        cfg_r[6] |
                    7 then cfg_data_out[7] -> ctrltop_top || sync
                        cfg_r[7] |
                    8 then cfg_data_out[8] -> ctrltop_top || sync
                        cfg_r[8]
                      else cfg_data_out[9] -> ctrltop_top || sync
                          cfg_r[9]
                end -- case ( top_ctrltop.addr as addr )
            end -- case ( top_ctrltop.mode as mode )
        end -- select top_ctrltop
    end -- loop
end -- procedure ctrltop
```

Listing A.4: samplertop.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]
import [sampler]

procedure samplertop (
  -- Input(s)
  sync ctrltop_samplertop;
  input sense_in : bit;
  array 2 of sync reg_r;
  array 2 of input reg_data_out : regdata;
  -- Output(s)
  output sensedrive_out : sensedrive_bd;
  output samplertop_medtop : regdata
) is
    -- Channel(s)
    channel samplertop_sampler : samplertop_sampler_bd
    channel sampler_samplertop : regdata
    -- Variable(s)
    variable numsamples : regdata
    variable subvalue : regdata
begin
    sampler(
      samplertop_sampler,
      sense_in,
      sensedrive_out,
      sampler_samplertop
    )
 ||
    loop
        [
          -- Wait for ctrltop_samplertop
          select ctrltop_samplertop then continue end
        ;
          -- Read numsamplesreg
          [sync reg_r[0] || reg_data_out[0] -> numsamples]
        ;
          -- Read subvaluereg
          [sync reg_r[1]|| reg_data_out[1] -> subvalue]
        ;
          [
            -- Start sampler and wait for sampler data
            samplertop_sampler <- {
              numsamples,
              subvalue
            }
         ||
            select sampler_samplertop then
                -- Send data to medtop
                samplertop_medtop <- sampler_samplertop
            end -- select sampler_samplertop
          ]
        ]
    end -- loop
end -- procedure samplertop
```

Listing A.5: sampler.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]

procedure sampler (
  -- Input(s)
  input samplertop_sampler : samplertop_sampler_bd;
  input sense_in : bit;
  -- Output(s)
  output sensedrive_out : sensedrive_bd;
  output sampler_samplertop : regdata
) is
    -- Variable(s)
    variable counter : regdata
    variable value : regdata
    -- Shared
    shared add is
    begin
        value := (value + 1 as regdata)
    end
    shared dec is
    begin
        counter := (counter - 0b1 as regdata)
    end
begin
    loop
        -- Wait for data
        select samplertop_sampler then
              -- Load value reg with negative constant value
              value := samplertop_sampler.subvalue
            ;
              -- Reset counter
              counter := samplertop_sampler.numsamples
            ;
              sensedrive_out <- {
                -- sense_oe high
                1,
                -- sense_out low
                0,
                -- drive_oe high
                1,
                -- drive_out low
                0
              }
            ;
               sensedrive_out <- {
                -- sense_oe high-Z
                0,
                -- sense_out low
                0,
                -- drive_oe high
                1,
                -- drive_out high
                1
              }
            ;
```

```
              -- Count number of times sense_in is low
              loop while counter > 0 then
                  select sense_in then -- Synchronize on clock signal
                      [
                        if (sense_in = 0) then
                            add()
                        end
                      ;
                        dec()
                      ]
                  end -- select sense_in
              end -- loop
            ;
              -- Reset counter
              counter := (samplertop_sampler.numsamples as regdata)
            ;
              sensedrive_out <- {
                -- sense_oe high
                1,
                -- sense_out high
                1,
                -- drive_oe high
                1,
                -- drive_out high
                1
              }
            ;
               sensedrive_out <- {
                -- sense_oe high-Z
                0,
                -- sense_out low
                1,
                -- drive_oe high
                1,
                -- drive_out low
                0
              }
            ;
              -- Count number of times sense_in is high
              loop while (counter > 0) then
                  select sense_in then
                      [
                        if (sense_in) then
                            add()
                        end
                      ;
                        dec()
                      ]
                  end -- select sense_in
              end -- loop
            ;
              -- Output sample data
              sampler_samplertop <- (value as regdata)
          end -- select samplertop_sampler
    end -- loop
end -- procedure sampler
```

Listing A.6: medtop.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]
import [med]

procedure medtop (
  -- Input(s)
  input samplertop_medtop : regdata;
  array 3 of input reg_data_out : regdata;
  -- Output(s)
  array 3 of sync reg_r;
  array 3 of output reg_data_in : regdata;
  output medtop_ematop : regdata
) is
    -- Channel(s)
    channel medtop_med : array 3 of regdata
    channel med_medtop : regdata
    -- Variable(s)
    variable data : array 3 of regdata
    variable old : 2 bits
    variable newold : 2 bits
begin
    med(medtop_med, med_medtop)
 ||
    loop
        -- Wait for samplertop_medtop
        select samplertop_medtop then
            -- Write samplertop data to reg
            case old of
                0b00 then
                    [
                      reg_data_in[0] <- samplertop_medtop
                    ;
                      newold := 0b01
                    ]
                |
                0b01 then
                    [
                      reg_data_in[1] <- samplertop_medtop
                    ;
                      newold := 0b10
                    ]
                else
                    [
                      reg_data_in[2] <- samplertop_medtop
                    ;
                      newold := 0b00
                    ]
            end -- case old
        end -- select samplertop_medtop
      ;
        old := newold
      ;
        -- Read medregs
        reg_data_out[0] -> data[0] || sync reg_r[0]
      ;
        reg_data_out[1] -> data[1] || sync reg_r[1]
      ;
        reg_data_out[2] -> data[2] || sync reg_r[2]
      ;
        [
          -- Start med and wait for med data
          medtop_med <- data
       ||
          select med_medtop then
              -- Send data to ematop
              medtop_ematop <- med_medtop
          end -- select med_medtop
        ]
    end -- loop
end -- procedure medtop
```

Listing A.7: med.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]

procedure med (
  -- Input (s)
  input medtop_med : array 3 of regdata;
  -- Output (s)
  output med_medtop : regdata
) is
    -- Variable(s)
    variable res_tmp : bit
    variable res : array 3 of bit
    variable c : array 2 of regdata
    -- Shared procedure(s)
    shared cmp is
    begin
        if c[0] > c[1] then
            res_tmp := 1
        else
            res_tmp := 0
        end
    end
begin
    loop
        select medtop_med then
            [
              -- LOAD1
              c[0] := medtop_med[0]
            ;
              c[1] := medtop_med[1]
            ;
              -- COMP1
              cmp()
            ;
              res[0] := res_tmp
            ;
              -- LOAD2
              if res[0] then
                  c[0] := medtop_med[0]
              else
                  c[0] := medtop_med[1]
              end -- if res [ 0 ]
            ;
              c[1] := medtop_med[2]
            ;
              -- COMP2
              cmp()
            ;
              res[1] := res_tmp
            ;
              -- LOAD3
              if res[0] then
                  [
                    c[0] := medtop_med[1]
                  ;
                    if res[1] then
                        c[1] := medtop_med[2]
                    else
                        c[1] := medtop_med[0]
                    end -- if res [ 1 ]
                  ]
              else
                  [
                    c[0] := medtop_med[0]
                  ;
                    if res[1] then
                        c[1] := medtop_med[2]
                    else
                        c[1] := medtop_med[0]
                    end -- if res [ 1 ]
                  ]
              end -- if res [ 0 ]
            ;
              -- COMP3
              cmp()
            ;
              res[2] := res_tmp
            ;
              -- FIN
              case res of
                  {0,1,1}, {0,0,1}, {1,0,0} then med_medtop <-
                      medtop_med[0]
              |
                  {1,0,1}, {1,1,1}, {0,0,0} then med_medtop <-
                      medtop_med[1]
              |
                  {0,1,0}, {1,1,0} then med_medtop <- medtop_med[2]
              end -- case res
            ]
        end -- select medtop_med
    end -- loop
end -- procedure med
```

Listing A.8: ematop.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]
import [ema]

procedure ematop (
  -- Input(s)
  input medtop_ematop : regdata;
  array 3 of input reg_data_out : regdata;
  -- Output(s)
  array 3 of sync reg_r;
  array 2 of output reg_data_in : regdata;
  output ematop_thcomptop : regdata
) is
    -- Channel(s)
    channel ematop_ema : array 3 of regdata
    channel ema_ematop : regdata
    -- Variable(s)
    variable data : array 3 of regdata
begin
    ema(ematop_ema, ema_ematop)
 ||
    loop
        [
          -- Wait for medtop_ematop
          select medtop_ematop then
              -- Write to reg
              reg_data_in[0] <- medtop_ematop  -- med_i
          end -- select medtop_ematop
        ;
          -- Read emaregs
          [reg_data_out[0] -> data[0] || sync reg_r[0]] -- med_i
        ;
          [reg_data_out[1] -> data[1] || sync reg_r[1]] -- ema_{i
              -1}
        ;
          [reg_data_out[2] -> data[2] || sync reg_r[2]] -- alpha
        ;
          -- Start ema and wait for ema data
          [
            ematop_ema <- data
         ||
            select ema_ematop then
                -- ema_i overwrites ema_{i-1}
                reg_data_in[1] <- ema_ematop
              ;
                -- Send data to thcomptop
                ematop_thcomptop <- ema_ematop -- ema_i
            end -- select ema_ematop
          ]
        ]
    end -- loop
end -- procedure ematop
```

Listing A.9: ema.balsa

```
-- Import libraries
import [balsa.types.basic]
import [balsa.sim.string]
import [mult_shiftadd]

procedure ema (
  -- Input (s)
  input ematop_ema : array 3 of regdata;
  -- Output (s)
  output ema_ematop : regdata
) is
    -- Variable(s)
    variable r0 : TNS
    variable r1 : TNS
    variable res_tmp : TNS
    -- Channel(s)
    channel ch_xy : EMA_MULT_BD
    channel ch_product : TN
    -- Shared procedure(s)
    shared add is
    begin
        res_tmp := (r0 + r1 as TNS)
    end -- shared add
begin
    mult_shiftadd(
      ch_xy,
      ch_product
    )
 ||
    loop
        select ematop_ema then
            [
              r0 := (ematop_ema[0] as TNS) -- (med_i)
            ;
              r1 := (-(ematop_ema[1] as TN1) as TNS) -- (-ema_{i
                  -1})
            ;
              add() -- (med_i + (-ema_{i-1})
            ;
              [
                ch_xy <- {
                  (ematop_ema[2] as TN) , -- (alpha)
                  (res_tmp as TNS)
                }
             ||
                select ch_product then
                    r1 := (ch_product as TNS)
                end -- select ch_product
              ]
            ;
              r0 := (ematop_ema[1] as TNS)
            ;
              add() -- (ema_{i-1} + res_tmp)
            ;
              ema_ematop <- (res_tmp as regdata)
            ]
        end -- select ematop_ema
    end -- loop
end -- procedure ema
```

Listing A.10: mult_shiftadd.balsa

```
-- Import libraries
import [balsa.types.basic]
import [balsa.sim.string]
import [params]
-- Local type(s)
type cntrmult is (log NUM_BITS) bits
type T2N is (NUM_BITS*2)+1 signed bits
type TN is (NUM_BITS) bits
type TN1 is (NUM_BITS+1) bits
type TNS is (NUM_BITS+1) signed bits
type EMA_MULT_BD is record
```

```
    multiplier : TN;
    multiplicand : TNS
end

procedure mult_shiftadd(
  -- Input(s)
  input xy : EMA_MULT_BD;
  -- Output(s)
  output product : TN
) is
    --Variable(s)
    variable a : T2N
    variable b : T2N
    variable q : TN
    variable n : cntrmult
    -- Shared procedure(s)
    shared shiftright is
    begin
        q := (#q[MSB..1] @ #0b0 as TN)
    end -- shared shiftright
    shared shiftleft is
    begin
        b := (((#0b0) @ (#b[2*MSB+1..0])) as T2N)
    end -- shared shiftleft
    shared add is
    begin
        a := (a + b as T2N)
    end -- shared add
begin
    loop
        select xy then
            [
              b := (xy.multiplicand as T2N)
            ;
              q := xy.multiplier
            ;
              a := 0
            ;
              n := (NUM_BITS as cntrmult)
            ;
              loop
                  [
                    if #q[0] then
                        add()
                    end
                  ;
                    shiftleft()
                  ;
                    shiftright()
                  ;
                    n := (n - 1 as cntrmult)
                  ]
              while (n /= 0)
              end -- loop
            ;
              product <- (#a[(2*MSB)+1..MSB+1] as TN) -- Trunk
            ]
        end -- select xy
    end -- loop
end -- procedure mult_shiftadd
```

Listing A.11: thcomptop.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]
import [thcomp]

procedure thcomptop (
  -- Input(s)
  input ematop_thcomptop : regdata;
  array 2 of input reg_data_out : regdata;
  -- Output(s)
  array 2 of sync reg_r;
  output reg_data_in : regdata;
  output thcomptop_ctrltop : bit
) is
    -- Channel(s)
    channel thcomptop_thcomp : thcomptop_thcomp_bd
    channel thcomp_thcomptop : bit
    -- Variable(s)
    variable value : regdata
    variable threshold : regdata
begin
    thcomp(
      thcomptop_thcomp,
      thcomp_thcomptop
    )
 ||
    loop
        [
          -- Wait for ematop_thcomptop
          select ematop_thcomptop then
              -- Write ematop data to reg
              reg_data_in <- ematop_thcomptop
          end -- select ematop_thcomptop
        ;
          -- Read thcompreg
          [reg_data_out[0] -> value || sync reg_r[0]]
        ;
          -- Read thresholdreg
          [reg_data_out[1] -> threshold || sync reg_r[1]]
        ;
          [
            -- Start thcomp and wait for thcomp data
            thcomptop_thcomp <- {
              value,
              threshold
            }
         ||
            select thcomp_thcomptop then
                -- Send data to ctrltop
                thcomptop_ctrltop <- thcomp_thcomptop
            end -- select thcomp_thcomptop
          ]
        ]
    end -- loop
end -- procedure thcomptop
```

Listing A.12: thcomp.balsa

```
-- Import library
import [params]

procedure thcomp (
  -- Input(s)
  input thcomptop_thcomp : thcomptop_thcomp_bd;
  -- Output(s)
  output thcomp_thcomptop : bit
) is
begin
    loop
        -- Wait for data
        select thcomptop_thcomp then
            -- Compare data with threshold
            if (thcomptop_thcomp.value > thcomptop_thcomp.threshold
                ) then
                thcomp_thcomptop <- 1
            else
                thcomp_thcomptop <- 0
            end -- if ( thcomptop_thcomp.value > thcomptop_thcomp.
                threshold )
        end -- select thcomptop_thcomp
    end -- loop
end -- procedure thcomp
```

Listing A.13: regs.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]
import [reg1regw2r]
import [reg1cfgw2r]

procedure regs (
  -- Input(s)
  array N of sync reg_r;
  array N of sync cfg_r;
  array N-M of input reg_data_in : regdata;
  array M of input cfg_data_in : regdata;
  -- Output(s)
  array N of output reg_data_out : regdata;
  array N of output cfg_data_out : regdata
) is
begin
    for || i in M .. N-1 then
        reg1regw2r(
          reg_r[i],
          cfg_r[i],
          reg_data_in[i-M],
          reg_data_out[i],
          cfg_data_out[i]
        )
    end -- for || i in M .. N 1
 ||
    for || i in 0 .. M-1 then
        reg1cfgw2r(
          reg_r[i],
          cfg_r[i],
          cfg_data_in[i],
          reg_data_out[i],
          cfg_data_out[i]
        )
    end -- for || i in 0 .. M 1
end -- procedure regs
```

Listing A.14: reg1cfgw2r.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]

procedure reg1cfgw2r (
  -- Input(s)
  sync reg_r;
  sync cfg_r;
  input cfg_data_in : regdata;
  -- Output(s)
  output reg_data_out : regdata;
  output cfg_data_out : regdata
) is
    -- Variable(s)
    variable reg : regdata
    -- Channel(s)
    channel ch_r : bit
begin
       loop
       arbitrate
            ch_r then
                case ch_r of
                      0 then reg_data_out <- reg |
                      1 then cfg_data_out <- reg
                end
        |
            cfg_data_in then
                  reg := cfg_data_in
            end
       end
      ||
        loop
        arbitrate reg_r then ch_r <- 0
        |         cfg_r then ch_r <- 1
        end
    end
end
```

Listing A.15: reg1regw2r.balsa

```
-- Import libraries
import [balsa.types.basic]
import [params]

procedure reg1regw2r (
  -- Input(s)
  sync reg_r;
  sync cfg_r;
  input reg_data_in : regdata;
  -- Output(s)
  output reg_data_out : regdata;
  output cfg_data_out : regdata
) is
    -- Variable(s)
    variable reg : regdata
    -- Channel(s)
    channel ch_r : bit
begin
    [
      loop
          arbitrate
          ch_r then
              case ch_r of
                  0 then reg_data_out <- reg |
                  1 then cfg_data_out <- reg
              end
          |
              reg_data_in then reg := reg_data_in
          end -- arbitrate
      end -- loop
    ]
 ||
    [
      loop
          arbitrate reg_r then ch_r <- 0
          |          cfg_r then ch_r <- 1
          end -- arbitrate reg_r
      end -- loop
    ]
end
```

A.3 Verification Tests

Listing A.16: med_tb.balsa

```
-- Threshold Comparator Testbench
--import [balsa.types.builtin] -- Functions and type necessary for
    balsa-c functionality.
import [balsa.types.basic] -- Type comprehension functions.
import [balsa.sim.string] -- Other String handling functions.
import [balsa.sim.fileio] -- File I/O.
import [balsa.sim.memory] -- Functions and types to implement
    memory models.
import [balsa.sim.portio] -- Port file/console I/O used by balsa-
    make-test.
import [balsa.sim.sim] -- Simulator specific operations such as
    time and command line argument access.
import [med]
import [inputGen]
import [outputComp]

procedure med_tb (
  input filename_i0 : String;
  input filename_i1 : String;
  input filename_o : String
) is

channel ch_input_vect : array 3 of regdata
channel ch_output_vect : regdata
```

```

variable file_i0, file_i1 : File
variable file_o : File

begin
    filename_i0, filename_i1, filename_o -> then
        [
          -- Open file
          file_i0 := FileOpen (filename_i0, read)
        ;
          -- Generate input vector(s)
          inputGen(3, regdata, array 3 of regdata, "Input vector(s)
              ", <-file_i0, ch_input_vect)
        ]
      ||
        -- DUT
        med(ch_input_vect, ch_output_vect)
      ||
        [
          -- Open file(s)
          file_i1 := FileOpen (filename_i1, read)
        ;
          file_o := FileOpen (filename_o, write)
        ;
          outputComp(regdata, "Output vector", <- file_i1, <-
              file_o, ch_output_vect)
        ]
    end
end
```

Listing A.17: ema_tb.balsa

```
-- Threshold Comparator Testbench
--import [balsa.types.builtin] -- Functions and type necessary for
    balsa-c functionality.
import [balsa.types.basic] -- Type comprehension functions.
import [balsa.sim.string] -- Other String handling functions.
import [balsa.sim.fileio] -- File I/O.
import [balsa.sim.memory] -- Functions and types to implement
    memory models.
import [balsa.sim.portio] -- Port file/console I/O used by balsa-
    make-test.
import [balsa.sim.sim] -- Simulator specific operations such as
    time and command line argument access.
import [ema]
import [inputGen]
import [outputComp]

procedure ema_tb (
  input filename_i0 : String;
  input filename_i1 : String;
  input filename_o : String
) is

channel ch_input_vect : array 3 of regdata
channel ch_output_vect : regdata

variable file_i0, file_i1 : File
variable file_o : File

```

```
begin
    filename_i0, filename_i1, filename_o -> then
        [
          -- Open file
          file_i0 := FileOpen (filename_i0, read)
        ;
          -- Generate input vector(s)
          inputGen(3, regdata, array 3 of regdata, "Input vector(s)
              ", <-file_i0, ch_input_vect)
        ]
      ||
        -- DUT
        ema(ch_input_vect, ch_output_vect)
      ||
        [
          -- Open file(s)
          file_i1 := FileOpen (filename_i1, read)
        ;
          file_o := FileOpen (filename_o, write)
        ;
          outputComp(regdata, "Output vector", <- file_i1, <-
              file_o, ch_output_vect)
        ]
    end
end
```

Listing A.18: thcomp_tb.balsa

```
-- Threshold Comparator Testbench
import [balsa.types.basic] -- Type comprehension functions.
import [balsa.sim.string] -- Other String handling functions.
import [balsa.sim.fileio] -- File I/O.
import [balsa.sim.memory] -- Functions and types to implement
    memory models.
import [balsa.sim.portio] -- Port file/console I/O used by balsa-
    make-test.
import [balsa.sim.sim] -- Simulator specific operations such as
    time and command line argument access.
import [thcomp]
import [inputGen]
import [outputComp]

procedure thcomp_tb (
  input filename_i0 : String;
  input filename_i1 : String;
  input filename_o : String
) is
-- Channel(s)
channel ch_input_vect : thcomptop_thcomp_bd
channel ch_output_vect : bit
-- Variable(s)
variable file_i0, file_i1 : File
variable file_o : File
begin
    filename_i0, filename_i1, filename_o -> then
        [
          -- Open file
          file_i0 := FileOpen (filename_i0, read)
        ;
          -- Generate input vector(s)
          inputGen(regdata, thcomptop_thcomp_bd, "Input vector(s)",
               <-file_i0, ch_input_vect)
        ]
     ||
        -- DUT
        thcomp(ch_input_vect, ch_output_vect)
     ||
        [
          -- Open file(s)
          file_i1 := FileOpen (filename_i1, read)
        ;
          file_o := FileOpen (filename_o, write)
        ;
          outputComp(bit, "Output vector", <- file_i1, <- file_o,
              ch_output_vect)
        ]
    end
end
```

Appendix B

Verilog Code

The following sections show Verilog code listings. Emacs autoassignments have been removed for better readability.

B.1 Parameters

Listing B.1: sync_params.v

```
localparam
NUM_REGS = 10,
MSB_REGS_ADDRESS = $clog2(NUM_REGS)-1,
LENGTH = 8,
MSB = LENGTH - 1,
EMA_FRACBITS = LENGTH,
MSB_MULT = MSB,
MSB_MULTI = EMA_FRACBITS-1,
NUMSAMPLESREG = 4'b0000,
SUBVALUEREG = 4'b0001,
MEDREG0 = 4'b0010,
MEDREG1 = 4'b0011,
MEDREG2 = 4'b0100,
EMAREG0 = 4'b0101,
EMAREG1 = 4'b0110,
ALPHAREG = 4'b0111,
THCOMPREG = 4'b1000,
THRESHOLDREG = 4'b1001;
```

B.2 Synchronous Modules

Listing B.2: top.v

```verilog
module top (/*AUTOARG*/);
`include "sync_params.v"
  //-------------Input Ports-----------------------------
  input clk;
  input clk_sampler;
  input rst_n;
  input top_ctrltop_start;
  input [MSB:0] top_ctrltop_cfg_data_in;
  input [MSB_REGS_ADDRESS:0] top_ctrltop_cfg_addr;
  input                      top_ctrltop_cfg_r;
  input                      top_ctrltop_cfg_w;
  input                      sense_in;
  //-------------Output Ports----------------------------
  output                     ctrltop_top_start;
  output                     ctrltop_top_start_data;
  output [MSB:0]             ctrltop_top_data;
  output                     sense_oe;
  output                     sense_out;
  output                     drive_oe;
  output                     drive_out;
  /*AUTOWIRE*/
  wire                  rst_n;
  wire                  clk;
  wire                  clk_sampler;
  wire                  top_ctrltop_start;
  wire                  ctrltop_top_start;
  wire [MSB:0]          ctrltop_top_data;
  wire                  sense_in;
  wire                  sense_oe;
  wire                  sense_out;
  wire                  drive_oe;
  wire                  drive_out;
  wire                  thcomptop_ctrltop_finish;
  samplertop SAMPLERTOP (/*AUTOINST*/);
  medtop MEDTOP (/*AUTOINST*/);
  ematop EMATOP (/*AUTOINST*/);
  thcomptop THCOMPTOP (/*AUTOINST*/);
  ctrltop CTRLTOP (/*AUTOINST*/);
endmodule
```

Listing B.3: ctrltop.v

```verilog
module ctrltop (/*AUTOARG*/);
`include "sync_params.v"
  //-------------Input Ports-----------------------------
  input clk;
  input rst_n;
  input top_ctrltop_start;
  input top_ctrltop_cfg_w;
  input top_ctrltop_cfg_r;
  input [MSB_REGS_ADDRESS:0] top_ctrltop_cfg_addr;
  input [MSB:0]              top_ctrltop_cfg_data_in;
  input [MSB:0]              samplerregs_ctrltop_cfg_data_out0;
  input [MSB:0]              samplerregs_ctrltop_cfg_data_out1;
  input [MSB:0]              medregs_ctrltop_cfg_data_out0;
  input [MSB:0]              medregs_ctrltop_cfg_data_out1;
  input [MSB:0]              medregs_ctrltop_cfg_data_out2;
  input [MSB:0]              emaregs_ctrltop_cfg_data_out0;
  input [MSB:0]              emaregs_ctrltop_cfg_data_out1;
  input [MSB:0]              emaregs_ctrltop_cfg_data_out2;
  input [MSB:0]              thcompregs_ctrltop_cfg_data_out0;
  input [MSB:0]              thcompregs_ctrltop_cfg_data_out1;
  input                      thcomptop_ctrltop_data;
  input                      thcomptop_ctrltop_finish;
  //-------------Output Ports----------------------------
  output reg                 ctrltop_top_start;
  output reg                 ctrltop_top_start_data;
  output reg                 ctrltop_samplertop_start;
  output reg [MSB:0]         ctrltop_top_data;
  output reg                 cfg_we;
  output reg [MSB:0]         cfg_data_in;
  output reg [MSB_REGS_ADDRESS:0] cfg_addr;
  //-------------Registers-------------------------------
  reg                             state_r;
  reg                             state_nxt;
  //-------------Parameters------------------------------
  parameter
    IDLE=1'b0,
    MEASURING=1'b1;
  // FSM, Combinatorial logic
  always @* begin
    state_nxt = state_r;
    /*AUTORESET*/
    ctrltop_top_start = 1'b0;
    ctrltop_top_start_data = 1'b0;
    ctrltop_samplertop_start = 1'b0;
    cfg_we = 1'b0;
    cfg_addr = 0;
    cfg_data_in = 0;
    if (top_ctrltop_cfg_r)
      case (top_ctrltop_cfg_addr)
        NUMSAMPLESREG: begin
          ctrltop_top_data = samplerregs_ctrltop_cfg_data_out0;
        end
        SUBVALUEREG: begin
          ctrltop_top_data = samplerregs_ctrltop_cfg_data_out1;
        end
        MEDREG0: begin
          ctrltop_top_data = medregs_ctrltop_cfg_data_out0;
        end
        MEDREG1: begin
          ctrltop_top_data = medregs_ctrltop_cfg_data_out1;
        end
        MEDREG2: begin
          ctrltop_top_data = medregs_ctrltop_cfg_data_out2;
        end
        EMAREG0: begin
          ctrltop_top_data = emaregs_ctrltop_cfg_data_out0;
        end
        EMAREG1: begin
          ctrltop_top_data = emaregs_ctrltop_cfg_data_out1;
        end
        ALPHAREG: begin
          ctrltop_top_data = emaregs_ctrltop_cfg_data_out2;
        end
```

```
        THCOMPREG: begin
          ctrltop_top_data = thcompregs_ctrltop_cfg_data_out0;
        end
        THRESHOLDREG: begin
          ctrltop_top_data = thcompregs_ctrltop_cfg_data_out1;
        end
      endcase
    else if (top_ctrltop_cfg_w) begin
      cfg_we = 1'b1;
      cfg_addr = top_ctrltop_cfg_addr;
      cfg_data_in = top_ctrltop_cfg_data_in;
    end
    case (state_r)
      IDLE: begin
        if(top_ctrltop_start) begin
          ctrltop_samplertop_start = 1'b1;
          state_nxt = MEASURING;
        end
      end
      MEASURING: begin
        if(thcomptop_ctrltop_finish) begin
     ctrltop_top_start = 1'b1;
            ctrltop_top_start_data = thcomptop_ctrltop_data;
     state_nxt = IDLE;
        end
      end
    endcase
  end
  // Sequential logic
  always @ (posedge clk or negedge rst_n)
    if (!rst_n)
      state_r <= IDLE;
    else
      state_r <= state_nxt;
endmodule
```

Listing B.4: medtop.v

```
module medtop (/*AUTOARG*/);
`include "sync_params.v"
  //-------------Parameters---------------------
  parameter
    IDLE=2'b00,
    START=2'b01,
    FIN=2'b10;
  //-------------Input Ports--------------------
  input clk;
  input rst_n;
  input samplertop_medtop_start;
  input [MSB:0] samplertop_medtop_data;
  //-------------Output Ports-------------------
  output reg    medtop_ematop_start;
  output reg [MSB:0] medtop_ematop_data;
  output [MSB:0]     medregs_ctrltop_cfg_data_out0;
  output [MSB:0]     medregs_ctrltop_cfg_data_out1;
  output [MSB:0]     medregs_ctrltop_cfg_data_out2;
  //-------------Registers---------------------
  reg                medtop_med_start;
  reg [MSB:0]        medtop_medregs_reg_data_in0;
  reg [MSB:0]        medtop_medregs_reg_data_in1;
```

```verilog
reg [MSB:0]         medtop_medregs_reg_data_in2;
reg                 medtop_medregs_reg_we0;
reg                 medtop_medregs_reg_we1;
reg                 medtop_medregs_reg_we2;
reg [1:0]           state_r;
reg [1:0]           state_nxt;
reg [1:0]           old_r;
reg [1:0]           old_nxt;
//-------------Wires-------------------------
/*AUTOWIRE*/
always @ (posedge clk or negedge rst_n)
  if (!rst_n) begin
    state_r <= IDLE;
    old_r <= 2'b00;
  end
  else begin
    state_r <= state_nxt;
    old_r <= old_nxt;
  end
always @* begin
  state_nxt = state_r;
  old_nxt = old_r;
  medtop_med_start = 1'b0;
  medtop_medregs_reg_we0 = 1'b0;
  medtop_medregs_reg_we1 = 1'b0;
  medtop_medregs_reg_we2 = 1'b0;
  medtop_medregs_reg_data_in0 = 0;
  medtop_medregs_reg_data_in1 = 0;
  medtop_medregs_reg_data_in2 = 0;
  medtop_ematop_start = 1'b0;
  medtop_ematop_data = 0;
  case (state_r)
    IDLE: begin
      if (samplertop_medtop_start) begin
          case (old_r)
            2'b00: begin
              medtop_medregs_reg_we0 = 1'b1;
              medtop_medregs_reg_data_in0 =
                  samplertop_medtop_data;
              old_nxt = 2'b01;
            end
            2'b01: begin
              medtop_medregs_reg_we1 = 1'b1;
              medtop_medregs_reg_data_in1 =
                  samplertop_medtop_data;
              old_nxt = 2'b10;
            end
            2'b10: begin
              medtop_medregs_reg_we2 = 1'b1;
              medtop_medregs_reg_data_in2 =
                  samplertop_medtop_data;
              old_nxt = 2'b00;
            end
            2'b11: begin
            end
          endcase
        state_nxt = START;
      end
    end
    START: begin
```

```
        medtop_med_start = 1'b1;
        state_nxt = FIN;
      end
      FIN: begin
        if (med_medtop_finish) begin
          medtop_ematop_start = 1'b1;
          medtop_ematop_data = med_medtop_data;
          state_nxt = IDLE;
        end
      end
    endcase
  end
  medregs MEDREGS(/*AUTOINST*/);
  med MED (/*AUTOINST*/);
endmodule
```

Listing B.5: med.v

```
module med (/*AUTOARG*/);
`include "sync_params.v"
  //-------------Parameters------------------------------
  parameter
    IDLE=3'b000,
    LOAD1=3'b001,
    CMP1=3'b010,
    LOAD2=3'b011,
    CMP2=3'b100,
    LOAD3=3'b101,
    CMP3=3'b110,
    FIN=3'b111;
   parameter MSB_STATE = 2;
   parameter MSB_RES = 2;
  //-------------Input Ports-----------------------------
  input clk;
  input rst_n;
  input medtop_med_start;
  input [MSB:0] medregs_med_reg_data_out0;
  input [MSB:0] medregs_med_reg_data_out1;
  input [MSB:0] medregs_med_reg_data_out2;
  //-------------Output Ports----------------------------
  output reg    med_medtop_finish;
  output reg [MSB:0] med_medtop_data;
  //-------------Wires-----------------------------------
  wire                clk;
  wire                rst_n;
  wire                medtop_med_start;
  wire [MSB:0]        medregs_med_reg_data_out0;
  wire [MSB:0]        medregs_med_reg_data_out1;
  wire [MSB:0]        medregs_med_reg_data_out2;
  //-------------Registers-------------------------------
  reg [MSB_STATE:0]          state_r;
  reg [MSB_STATE:0]          state_nxt;
  reg [MSB:0]         cmp_r0;
  reg [MSB:0]         cmp_r1;
  reg [MSB:0] cmp_nxt0;
  reg [MSB:0] cmp_nxt1;
  reg [MSB_RES:0] res_r;
  reg [MSB_RES:0] res_nxt;
  always @ (posedge clk or negedge rst_n)
    if (!rst_n) begin
```

```
      state_r <= IDLE;
      res_r <= 0;
      cmp_r0 <= 0;
      cmp_r1 <= 0;
    end
    else begin
      state_r <= state_nxt;
      res_r <= res_nxt;
      cmp_r0 <= cmp_nxt0;
      cmp_r1 <= cmp_nxt1;
    end
  always @*
    begin
      state_nxt = state_r;
      res_nxt = res_r;
      cmp_nxt0 = cmp_r0;
      cmp_nxt1 = cmp_r1;
      med_medtop_finish = 1'b0;
      med_medtop_data = 0;
      case (state_r)
IDLE: begin
          if (medtop_med_start) begin
     state_nxt = LOAD1;
          end
end
LOAD1: begin
  cmp_nxt0 = medregs_med_reg_data_out0;
  cmp_nxt1 = medregs_med_reg_data_out1;
          state_nxt = CMP1;
end
CMP1:
  begin
    if (cmp_r0 > cmp_r1) begin
              res_nxt[0] = 1'b1;
            end
    else begin
              res_nxt[0] = 1'b0;
            end
            state_nxt = LOAD2;
  end
LOAD2: begin
  cmp_nxt1 = medregs_med_reg_data_out2;
  if (res_r[0]) begin
    cmp_nxt0 = medregs_med_reg_data_out0;
          end
  else begin
    cmp_nxt0 = medregs_med_reg_data_out1;
          end
          state_nxt = CMP2;
end
CMP2: begin
  if (cmp_r0 > cmp_r1)
    res_nxt[1] = 1'b1;
  else
    res_nxt[1] = 1'b0;
          state_nxt = LOAD3;
        end
LOAD3: begin
  if (res_r[0])
    cmp_nxt0 = medregs_med_reg_data_out1;
    else
      cmp_nxt0 = medregs_med_reg_data_out0;
    if (res_r[1])
      cmp_nxt1 = medregs_med_reg_data_out2;
    else
      if (res_r[0])
        cmp_nxt1 = medregs_med_reg_data_out0;
      else
        cmp_nxt1 = medregs_med_reg_data_out1;
           state_nxt = CMP3;
         end
  CMP3: begin
    if (cmp_r0 > cmp_r1)
      res_nxt[2] = 1'b1;
    else
      res_nxt[2] = 1'b0;
           state_nxt = FIN;
         end
  FIN: begin
    med_medtop_finish = 1'b1;
    case ({res_r[0],res_r[1],res_r[2]})
      3'b011, 3'b001, 3'b100 : med_medtop_data =
          medregs_med_reg_data_out0;
      3'b101, 3'b111, 3'b000 : med_medtop_data =
          medregs_med_reg_data_out1;
      3'b010, 3'b110 : med_medtop_data = medregs_med_reg_data_out2;
    endcase
           state_nxt = IDLE;
  end
       endcase
     end
endmodule
```

Listing B.6: ematop.v

```
module ematop (/*AUTOARG*/);
`include "sync_params.v"
  //-------------Parameters---------------------
  parameter
    IDLE=2'b00,
    START=2'b01,
    FIN=2'b10;
  //-------------Input Ports--------------------
  input clk;
  input rst_n;
  input medtop_ematop_start;
  input [MSB:0] medtop_ematop_data;
  input cfg_we;
  input [MSB:0] cfg_data_in;
  input [MSB_REGS_ADDRESS:0] cfg_addr;
  //-------------Output Ports-------------------
  output reg    ematop_thcomptop_start;
  output reg [MSB:0] ematop_thcomptop_data;
  output [MSB:0] emaregs_ctrltop_cfg_data_out0;
  output [MSB:0] emaregs_ctrltop_cfg_data_out1;
  output [MSB:0] emaregs_ctrltop_cfg_data_out2;
  //-------------Registers---------------------
  reg [MSB:0]        ematop_emaregs_reg_data_in0;
  reg [MSB:0]        ematop_emaregs_reg_data_in1;
  reg                ematop_emaregs_reg_we0;
  reg                  ematop_emaregs_reg_we1;
  reg                  ematop_emaregs_reg_we2;
  reg  [1:0]           state_r;
  reg  [1:0]           state_nxt;
  reg                  ematop_ema_start;
  //-------------Wires-------------------------
  /*AUTOWIRE*/
  always @ (posedge clk or negedge rst_n)
    if (!rst_n) begin
      state_r <= IDLE;
    end
    else begin
      state_r <= state_nxt;
    end
  always @* begin
    state_nxt = state_r;
    ematop_ema_start = 1'b0;
    ematop_emaregs_reg_we0 = 1'b0;
    ematop_emaregs_reg_we1 = 1'b0;
    ematop_emaregs_reg_data_in0 = 0;
    ematop_emaregs_reg_data_in1 = 0;
    ematop_thcomptop_start = 1'b0;
    ematop_thcomptop_data = 0;
    case (state_r)
      IDLE: begin
        if (medtop_ematop_start) begin
            ematop_emaregs_reg_we0 = 1'b1; // Write signal high
            ematop_emaregs_reg_data_in0 = medtop_ematop_data; //
                Overwrite sample
          state_nxt = START;
        end
      end
      START: begin
        ematop_ema_start = 1'b1;
        state_nxt = FIN;
      end
      FIN: begin
        if (ema_ematop_finish) begin
          ematop_thcomptop_start = 1'b1;
          ematop_thcomptop_data = ema_ematop_data;
           ematop_emaregs_reg_we1 = 1'b1; // Write signal high
           ematop_emaregs_reg_data_in1 = ema_ematop_data; //
               Overwrite EMA_i-1 with EMA_i
          state_nxt = IDLE;
        end
      end
    endcase
  end
  emaregs emaregs (/*AUTOINST*/);
  ema ema (/*AUTOINST*/);
endmodule
```

Listing B.7: ema.v

```
module ema (/*AUTOARG*/);
`include "sync_params.v"
  //-------------Parameter(s)----------------------------
  parameter
    IDLE=4'b000,
    SUB=4'b001,
```

```verilog
    MULT=4'b010,
    TRUNK=4'b011,
    ADD=4'b100,
    FIN=4'b101;
  parameter MSB_STATE = 2;
  //-------------Input Port(s)----------------------------
  input clk;
  input rst_n;
  input ematop_ema_start;
  input [MSB:0] emaregs_ema_reg_data_out0;
  input [MSB:0] emaregs_ema_reg_data_out1;
  input [MSB:0] emaregs_ema_reg_data_out2;
  //-------------Output Port(s)---------------------------
  output reg    ema_ematop_finish;
  output reg [MSB:0] ema_ematop_data;
  //-------------Wire(s)----------------------------------
  wire                clk;
  wire                rst_n;
  wire [MSB:0]        emaregs_ema_reg_data_out0;
  wire [MSB:0]        emaregs_ema_reg_data_out1;
  wire [MSB:0]        emaregs_ema_reg_data_out2;
  //-------------Register(s)------------------------------
  reg [MSB_STATE:0]  state_r;
  reg [MSB_STATE:0]  state_nxt;
  reg signed [(2*MSB_MULT)+2:0] result_r;
  reg signed [(2*MSB_MULT)+2:0] result_nxt;
  reg                        ema_multshiftadd_start;
  reg [MSB_MULT:0]           ema_multshiftadd_multiplicand;
  reg [MSB_MULTI:0]          ema_multshiftadd_multiplier;
  always @ (posedge clk or negedge rst_n)
    if (!rst_n) begin
      state_r <= IDLE;
      result_r <= 0;
    end
    else begin
      state_r <= state_nxt;
      result_r <= result_nxt;
    end
  always @*
    begin
      state_nxt = state_r;
      result_nxt = result_r;
      ema_ematop_finish = 1'b0;
      ema_ematop_data = 0;
      case (state_r)
 IDLE: begin
   if (ematop_ema_start)
     state_nxt = SUB;
 end
        SUB:
   begin
             result_nxt = emaregs_ema_reg_data_out0 + (~
                 emaregs_ema_reg_data_out1 + 1'b1); //Unsigned to
                 signed subtraction(addition)
             state_nxt = MULT;
   end
  MULT:
    begin
              result_nxt = result_r * $signed({1'b0,
                  emaregs_ema_reg_data_out2}); //Signed
                  multiplication
              state_nxt = TRUNK;
      end
 TRUNK:
      begin
             result_nxt = result_r >>> LENGTH; //Signed rightshift
             state_nxt = ADD;
     end
 ADD:
     begin
       result_nxt = result_r + $signed({emaregs_ema_reg_data_out1[MSB
           ],emaregs_ema_reg_data_out1}); //Signed addition
             state_nxt = FIN;
     end
 FIN:
     begin
       ema_ematop_finish = 1'b1;
       ema_ematop_data = result_r; //Signed to unsigned
             state_nxt = IDLE;
     end
        endcase
      end
endmodule
```

Listing B.8: thcomptop.v

```
module thcomptop (/*AUTOARG*/);
`include "sync_params.v"
  //-------------Parameters-------------------------------
  parameter
    IDLE=2'b00,
    START=2'b01,
    CMP=2'b10;
  parameter MSB_STATE = 1;
  //-------------Input Ports-----------------------------
  input clk;
  input rst_n;
  input ematop_thcomptop_start;
  input [MSB:0] ematop_thcomptop_data;
  input cfg_we;
  input [MSB:0] cfg_data_in;
  input [MSB_REGS_ADDRESS:0] cfg_addr;
  //-------------Output Ports----------------------------
  output reg  thcomptop_ctrltop_data;
  output reg         thcomptop_ctrltop_finish;
  output [MSB:0]     thcompregs_ctrltop_cfg_data_out0;
  output [MSB:0]     thcompregs_ctrltop_cfg_data_out1;
  //-------------Registers--------------------------------
  reg         thcomptop_thcompregs_we0;
  reg [MSB:0]          thcomptop_thcompregs_reg_data_in0;
  reg           thcomptop_thcomp_start;
  reg [MSB_STATE:0] state_r;
  reg [MSB_STATE:0] state_nxt;
  /*AUTOWIRE*/

  thcomp thcomp (/*AUTOINST*/);
  thcompregs THCOMPREGS (/*AUTOINST*/);

  // Sequential logic
  always @(posedge clk or negedge rst_n)
```

```
    if (!rst_n) begin
      state_r <= IDLE;
    end
    else begin
      state_r <= state_nxt;
    end

  // FSM Combinatorial logic
  always @* begin
    state_nxt = state_r;
    thcomptop_thcompregs_we0 = 1'b0;
    thcomptop_thcompregs_reg_data_in0 = 0;
    thcomptop_thcomp_start = 1'b0;
    thcomptop_ctrltop_finish = 1'b0;
    thcomptop_ctrltop_data = 0;
    case (state_r)
      IDLE: begin
        if (ematop_thcomptop_start) begin
           thcomptop_thcompregs_we0 = 1'b1;
            thcomptop_thcompregs_reg_data_in0 =
                ematop_thcomptop_data;
        state_nxt = START;
        end
      end
      START: begin
        thcomptop_thcomp_start = 1'b1;
        state_nxt = CMP;
      end
      CMP: begin
        if (thcomp_thcomptop_finish) begin
          thcomptop_ctrltop_data = thcomp_thcomptop_data;
          thcomptop_ctrltop_finish = 1'b1;
          state_nxt = IDLE;
        end
      end
    endcase
  end
endmodule
```

Listing B.9: thcomp.v

```
module thcomp (/*AUTOARG*/);
`include "sync_params.v"
   //-------------Parameters------------------------------
   parameter
     IDLE=1'b0,
     CMP=1'b1;
   //-------------Input Ports-----------------------------
   input clk;
   input rst_n;
   input thcomptop_thcomp_start;
   input [MSB:0] thcompregs_thcomp_reg_data_out0;
   input [MSB:0] thcompregs_thcomp_reg_data_out1;
   //-------------Output Ports----------------------------
   output reg    thcomp_thcomptop_finish;
   output reg    thcomp_thcomptop_data;
   //-------------Wires-----------------------------------
   wire          thcomptop_thcomp_start;
   //-------------Registers-------------------------------
   reg           state_r;
```

```
  reg           state_nxt;

  always @* begin
     state_nxt = state_r;
     thcomp_thcomptop_finish = 1'b0;
     thcomp_thcomptop_data = 1'b0;
     case (state_r)
 IDLE: begin
     if (thcomptop_thcomp_start)
       state_nxt = CMP;
 end
 CMP: begin
           if (thcompregs_thcomp_reg_data_out0 >
               thcompregs_thcomp_reg_data_out1) begin
              thcomp_thcomptop_data = 1'b1;
           end
     else begin
              thcomp_thcomptop_data = 1'b0;
           end
     thcomp_thcomptop_finish = 1'b1;
           state_nxt = IDLE;
 end
       endcase
  end

  always @ (posedge clk or negedge rst_n)
     if (!rst_n) begin
 state_r <= IDLE;
     end
     else begin
 state_r <= state_nxt;
     end
endmodule
```

B.3 Verification Tests

Listing B.10: med_tb.sv

```
module median3_tb();
`include "sync_params.v"
  localparam MAX_LINE_LENGTH = 11; // 10 bits + newline
  task run();
     reg [MSB:0]                 DataOutExpected;
     integer                     file_in,
       file_out,
       file_mon;
     integer                     return_in,
       return_out,
       return_mon;
     integer                     success_counter;
     integer                     run;
     integer                     match_counter,
       error_counter;
     reg [MAX_LINE_LENGTH*8-1:0]  str;
     begin
   run = 1;
   match_counter = 0;
   error_counter = 0;
   tb.start = 0;
   tb.data_i0 = 0;
   tb.data_i1 = 0;
   tb.data_i2 = 0;
   DataOutExpected = 0;
   file_in = $fopen("../../../standalone/sim/tests/median3/_input.
     dat","r"); //Open file in read mode.
   file_out = $fopen("../../../standalone/sim/tests/median3/_output.
     dat","r"); //Open file in read mode.
   file_mon = $fopen("../../../standalone/sim/tests/median3/_monitor
     .dat","w"); //Open file in write mode.
   $fwrite(file_mon, "median3_tb test started.\n");
   $display("median3_tb test started.");
   while(run) begin
      //Load input data
      return_in = $fgets(str, file_in);
      if(!return_in)
       run = 0;
      else begin
         success_counter = $sscanf(str, "%b", tb.data_i0);
         return_in = $fgets(str, file_in);
         success_counter = $sscanf(str, "%b", tb.data_i1);
         return_in = $fgets(str, file_in);
         success_counter = $sscanf(str, "%b", tb.data_i2);
         @(negedge tb.clk);

         //Initiate filter sequence
         @(negedge tb.clk);
         tb.start = 1;
         @(negedge tb.clk);
         tb.start = 0;

         //Wait for filter sequence finished
         @(posedge tb.finish);

         //Compare output data
         return_out = $fgets(str, file_out);
         success_counter = $sscanf(str, "%b", DataOutExpected);
         @(negedge tb.clk);
         if(DataOutExpected !== tb.data_o) begin
     $display("%0dns ERROR : Output wrong",$time);
     $display("       Got  %b", tb.data_o);
     $display("       Exp  %b", DataOutExpected);
     error_counter = error_counter + 1;
         end
         else begin
     $display("%0dns MATCH : Output correct",$time);
     $display("       Got  %b", tb.data_o);
     $display("       Exp  %b", DataOutExpected);
     match_counter = match_counter + 1;
         end
         @(posedge tb.clk);
      end
   end // while (run)
   $fwrite(file_mon, "#Matches : %d\n",match_counter);
   $fwrite(file_mon, "#Errors  : %d\n",error_counter);
   $fwrite(file_mon, "median3_tb test finished.\n");
   $display("#Matches : %d",match_counter);
   $display("#Errors  : %d",error_counter);
   $display("median3_tb test finished.");
   $fclose(file_in);
   $fclose(file_out);
   $fclose(file_mon);
       end
    endtask
 endmodule
```

Listing B.11: ema_tb.sv

```
module ema_tb();
`include "sync_params.v"
   localparam MAX_LINE_LENGTH = 5;
   task run();
      reg [MSB:0]                  DataOutExpected;
      integer       file_in,
       file_out,
       file_mon;
      integer       return_in,
       return_out,
       return_mon;
      integer       success_counter;
      integer       run;
      integer       match_counter,
       error_counter;
      reg [MAX_LINE_LENGTH*8-1:0]  str;
      begin
  run = 1;
  match_counter = 0;
  error_counter = 0;
  tb.ematop_ema_start = 0;
  tb.emaregs_ema_reg_data_out0 = 0;
  tb.emaregs_ema_reg_data_out1 = 0;
  tb.emaregs_ema_reg_data_out2 = 0;
  DataOutExpected = 0;
  //Open file(s) in read/write mode.
  file_in = $fopen("../../../standalone/sim/tests/ema/_input.dat","
      r");
  file_out = $fopen("../../../standalone/sim/tests/ema/_output.dat"
      ,"r");
  file_mon = $fopen("../../../standalone/sim/tests/ema/_monitor.dat
      ","w");
        $fwrite(file_mon, "ema_tb test started.");
  $display("ema_tb test started.");
  while(run) begin
     //Load input data
     return_in = $fgets(str, file_in);
     if(!return_in)
       run = 0;
     else begin
        success_counter = $sscanf(str, "%b", tb.
            emaregs_ema_reg_data_out0);
        return_in = $fgets(str, file_in);
        success_counter = $sscanf(str, "%b", tb.
            emaregs_ema_reg_data_out1);
        return_in = $fgets(str, file_in);
        success_counter = $sscanf(str, "%b", tb.
            emaregs_ema_reg_data_out2);
        @(posedge tb.clk);
        //Initiate filter sequence
        @(posedge tb.clk);
        tb.ematop_ema_start = 1'b1;
        @(posedge tb.clk);
        tb.ematop_ema_start = 1'b0;
        //Wait for filter sequence finished
        @(posedge tb.ema_ematop_finish);
        //Compare output data
        return_out = $fgets(str, file_out);
        success_counter = $sscanf(str, "%b", DataOutExpected);
        @(negedge tb.clk);
        if(DataOutExpected !== tb.ema_ematop_data) begin
    $display("%0dns ERROR : Output wrong",$time);
    $display("       Got  %b", tb.ema_ematop_data);
    $display("       Exp  %b", DataOutExpected);
    error_counter = error_counter + 1;
        end
        else begin
    $display("%0dns MATCH : Output correct",$time);
    $display("       Got  %b", tb.ema_ematop_data);
    $display("       Exp  %b", DataOutExpected);
    match_counter = match_counter + 1;
        end
        @(posedge tb.clk);
      end
  end // while (run)
         $fwrite(file_mon, "#Matches : %d\n", match_counter);
         $fwrite(file_mon, "#Errors  : %d\n", error_counter);
         $fwrite(file_mon, "ema_tb test finished.\n");
  $display("#Matches : %d", match_counter);
  $display("#Errors  : %d", error_counter);
  $display("ema_tb test finished.");
  $fclose(file_in);
  $fclose(file_out);
  $fclose(file_mon);
     end
   endtask
endmodule
```

Listing B.12: thcomp_tb.sv

```
module thcomp_tb();
`include "sync_params.v"
   localparam MAX_LINE_LENGTH_I = 11; // 10 bits + newline
   localparam MAX_LINE_LENGTH_O = 2;  // 1 bit + newline
   task run();
      reg   DataOutExpected;
      integer file_in,
       file_out,
       file_mon;
      integer return_in,
       return_out,
       return_mon;
      integer success_counter;
      integer run;
      integer match_counter,
       error_counter;
      reg [MAX_LINE_LENGTH_I*8-1:0] str_i;
      reg [MAX_LINE_LENGTH_O*8-1:0] str_o;
      begin
  run = 1;
```

```
match_counter = 0;
error_counter = 0;
tb.thcomp_start = 0;
tb.thcomp_i = 0;
tb.threshold = 0;
DataOutExpected = 0;
//Open file(s) in read/write mode.
file_in = $fopen("../../../standalone/sim/tests/thcomp/_input.dat","r");
file_out = $fopen("../../../standalone/sim/tests/thcomp/_output.dat","r");
file_mon = $fopen("../../../standalone/sim/tests/thcomp/_monitor.dat","w");
       $fwrite(file_mon, "thcomp_tb test started.\n");
$display("thcomp_tb test started.");
 while(run) begin
   //Load input data
   return_in = $fgets(str_i, file_in);
   if(!return_in)
     run = 0;
   else begin
      success_counter = $sscanf(str_i, "%b", tb.thcomp_i);
      return_in = $fgets(str_i, file_in);
      success_counter = $sscanf(str_i, "%b", tb.threshold);
      @(negedge tb.clk);
      //Initiate comparator sequence
      @(negedge tb.clk);
      tb.thcomp_start = 1;
      @(negedge tb.clk);
      tb.thcomp_start = 0;
      //Wait for comparator sequence finished
      @(posedge tb.thcomp_finish);
      //Compare output data
      return_out = $fgets(str_o, file_out);
      success_counter = $sscanf(str_o, "%b", DataOutExpected);
      @(negedge tb.clk);
      if(DataOutExpected !== tb.thcomp_o) begin
  $display("%0dns ERROR : Output wrong",$time);
  $display("       Got  %b", tb.thcomp_o);
  $display("       Exp  %b", DataOutExpected);
  error_counter = error_counter + 1;
      end
      else begin
  $display("%0dns MATCH : Output correct",$time);
  $display("       Got  %b", tb.thcomp_o);
  $display("       Exp  %b", DataOutExpected);
  match_counter = match_counter + 1;
      end
      @(posedge tb.clk);
   end
end // while (run)
       $fwrite(file_mon, "#Matches : %d\n",match_counter);
       $fwrite(file_mon, "#Errors  : %d\n",error_counter);
       $fwrite(file_mon, "thcomp_tb test finished.\n");
$display("#Matches : %d",match_counter);
$display("#Errors  : %d",error_counter);
$display("thcomp_tb test finished.");
$fclose(file_in);
$fclose(file_out);
$fclose(file_mon);
```

```
    end
  endtask
endmodule
```

Listing B.13: top_tb.sv

```
module top_tb();
`include "sync_params.v"
  localparam MAX_LINE_LENGTH = LENGTH+1; // 10 bits + newline
  localparam FREQ = 16;
  localparam MAX_NAME_LENGTH = 10;
  realtime start_period = 1e9/FREQ;
  reg      enabled;
  initial enabled = 1'b0;
  task run();
    integer                       file_in;
    integer     file_out;
    integer     file_mon;
    integer                       return_in;
    integer                       return_out;
    integer                       return_mon;
    integer                       success_counter;
    integer                       run_counter;
    integer                       match_counter;
    integer                       error_counter;
    integer                       start_time;
    integer                       stop_time;
    integer                       diff_time;
    integer                       sleep_time;
    reg [MAX_LINE_LENGTH*8-1:0]   str;
    reg [MSB:0]                   DataOutExpected;
    begin
      enabled = 1'b1;
      match_counter = 0;
      error_counter = 0;
      tb.top_ctrltop_start = 1'b0;
      tb.top_ctrltop_cfg_addr = 0;
      tb.top_ctrltop_cfg_w = 1'b0;
      tb.top_ctrltop_cfg_r = 1'b0;
      tb.top_ctrltop_cfg_data_in = 0;
      //tb.single_extres_model.C = 10e-12; //Change to emulate
          capacitance changes/'touch'
      DataOutExpected = 0;
      @(posedge tb.rst_n);
      //Open file in write mode.
      file_mon = $fopen("../../../standalone/sim/tests/top/_monitor
          .dat","w");
      $fwrite(file_mon, "top_tb test started.\n");
      $display("top_tb test started.");
      //Module configuration:
      //Write config data to numsampesreg
      write_cfg_data(NUMSAMPLESREG, "numsamples", {4{2'b11}});
      //Read config data from numsamplesreg
      read_cfg_data(NUMSAMPLESREG, "numsamples");
      //Write config data to subvaluereg
      write_cfg_data(SUBVALUEREG, "subvalue", {8{1'b0}});
      //Read config data from subvaluereg
      read_cfg_data(SUBVALUEREG, "subvalue");
      //Write config data to alphareg
      write_cfg_data(ALPHAREG, "alpha", {4{2'b10}});
```

```verilog
      //Read config data from alphareg
      read_cfg_data(ALPHAREG, "alpha");
      //Write config data to thresholdreg
      write_cfg_data(THRESHOLDREG, "threshold", {4{2'b11}});
      //Read config data from thresholdreg
      read_cfg_data(THRESHOLDREG, "threshold");

      run_counter = 16;
      while(run_counter) begin
        tb.clk_en = 1'b1;
        start_time = $realtime;
        $display("start time : ", start_time);
        start_circuit();
        run_counter = run_counter - 1;
        stop_time = $realtime;
        $display("stop time : ", stop_time);
        diff_time = stop_time - start_time;
        $display("diff time : ", diff_time);
        sleep_time =  start_period - diff_time;
        $display("sleep time : ", sleep_time);
        tb.clk_en = 1'b0;
        #(sleep_time);
      end
      $display("top_tb test finished.");
      $fclose(file_in);
      $fclose(file_out);
      $fclose(file_mon);
    end
  endtask
  // Task for writing config data
  task write_cfg_data;
    input [MSB_REGS_ADDRESS:0] addr;
    input [8*MAX_NAME_LENGTH-1:0] name;
    input [MSB:0]                 data;
    begin
      $display("Writing %0d to %0sreg", data, name);
      @(posedge tb.clk_dly);
      tb.top_ctrltop_cfg_addr = addr;
      tb.top_ctrltop_cfg_w = 1'b1;
      tb.top_ctrltop_cfg_data_in = data;
      @(posedge tb.clk_dly);
      tb.top_ctrltop_cfg_addr = 16'hx;
      tb.top_ctrltop_cfg_w = 1'b0;
      tb.top_ctrltop_cfg_data_in = 16'hx;
      $display("Finished writing to %0sreg.", name);
    end
  endtask
  // Task for reading config data
  task read_cfg_data;
    input [MSB_REGS_ADDRESS:0] addr;
    input [8*MAX_NAME_LENGTH-1:0] name;
    begin
      $display("Reading from %0sreg", name);
      @(posedge tb.clk_dly);
      tb.top_ctrltop_cfg_addr = addr;
      tb.top_ctrltop_cfg_r = 1'b1;
      @(posedge tb.clk_dly);
      tb.top_ctrltop_cfg_addr = 0;
      tb.top_ctrltop_cfg_r = 1'b0;
      $display("Read value: %0d", tb.ctrltop_top_data);
      $display("Finished reading from %0sreg.", name);
    end
  endtask
  // Tast for performing a sample and filter sequence
  task start_circuit;
    begin
      //Initiate filter sequence
      $display("Starting filter sequence");
      @(posedge tb.clk_dly);
      tb.top_ctrltop_start = 1'b1;
      @(posedge tb.clk_dly);
      tb.top_ctrltop_start = 1'b0;
       //Wait for filter sequence finished
      wait(tb.ctrltop_top_start);
    end
  endtask
  // Probe samplertop_medtop data signal
`define SM_START tb.U_DUT.MEDTOP.samplertop_medtop_start
`define SM_DATA tb.U_DUT.MEDTOP.samplertop_medtop_data
  initial begin
    wait(enabled);
    forever begin
      @(negedge `SM_START);
      $display("samplermedtop data %d", `SM_DATA);
    end
  end
  // Probe medtop_ematop data signal
`define ME_START tb.U_DUT.EMATOP.medtop_ematop_start
`define ME_DATA tb.U_DUT.EMATOP.medtop_ematop_data
  initial begin
    wait(enabled);
    forever begin
      @(negedge `ME_START);
      $display("medtopematop data %d", `ME_DATA);
    end
  end
  // Probe ematop_thcomptop data signal
`define ET_START tb.U_DUT.THCOMPTOP.ematop_thcomptop_start
`define ET_DATA tb.U_DUT.THCOMPTOP.ematop_thcomptop_data
  initial begin
    wait(enabled);
    forever begin
      @(negedge `ET_START);
      $display("ematopthtop data %d", `ET_DATA);
    end
  end
endmodule
```

B.4 Pad/RC Circuit Model

Listing B.14: single_extres_model.sv

```
module single_extres_model(/*AUTOARG*/
  // Outputs
  sense_in,
  // Inputs
  sense_oe, sense_out, drive_oe, drive_out
  );
  input sense_oe, sense_out;
  input drive_oe, drive_out;
  output reg sense_in;

  real C = 10e-12;
  real vcc = 3.3;
  real vc = 0;
  real vc_drive = 0;
  real Rsens = 50;
  real Rext = 1e6;
  real Rstrong = 50;
  real delta_t = 1e-10;
  real delta;

  always @* begin
    delta_t = Rsens*C/50;
    if (delta_t > 500e-9) delta_t = 500e-9;
  end

  // Hysteresis thresholding:
  initial sense_in = 1'b0;
  always @*
    if (sense_in)
      sense_in = vc > 0.3*vcc;
    else
      sense_in = vc > 0.7*vcc;

  always @*
    casez ({sense_oe, sense_out, drive_oe, drive_out})
      4'b0?0?: begin // Both pins floating.
        vc_drive = vc;
      end
      4'b0?10: begin // Drive driven low, sensing on sense.
        Rsens = Rext;
        vc_drive = 0;
      end
      4'b0?11: begin // Drive driven high, sensing on sense.
        Rsens = Rext;
        vc_drive = vcc;
      end
      4'b10??: begin // Sense driven low.
        Rsens = Rstrong;
        vc_drive = 0;
      end
      4'b11??: begin // Sense driven high.
        Rsens = Rstrong;
        vc_drive = vcc;
      end
    endcase

  // RC computer:
  always begin
    delta = vc_drive - vc;
    if ((delta>0?delta:-delta) > 0.01) begin
      vc = vc + delta_t*delta/(Rsens*C);
      // Neat implementation of a cancellable delay (note the
          join_any):
      fork
        #(delta_t*1e9);
        @(vc_drive or delta_t);
      join_any
    end
    else begin
      // We have converged. Wait until vc_drive changes for
          recomputation.
      @(vc_drive);
      #(0.01);
    end
  end
endmodule
```

Appendix C

Python Code

Listing C.1: med.py

```python
#Python script for generating test data for median-3 filter simulation.
import random  # For generating random numbers.
import time    # For timing each sort function with "time.clock()".

#Function for converting integer to binary.
def int2bin(n, count=128):
 return "".join([str((n >> y) & 1) for y in range(count-1, -1, -1)])

length = 8  #Number of bits, length of testvectors.
X = 255     #Max value.
N = 100     #Number of testcases.
M = 3     #Number of input vectors.
list = []

#Erase file content.
w = open('_input.dat', 'w')
w.close()
w = open('_output.dat', 'w')
w.close()

#Open files.
input_file = open('_input.dat', 'a+')
output_file = open('_output.dat', 'a+')

#Generate N input and output test vectors sets.
for k in range(0, N):
 #Append M random input test vectors to list.
 for i in range(0, M):
  list.append(random.randint(0, X-1))

 #Write input data in binary to "input_file".
 for j in range(0, M):
  input_file.write(int2bin(list[j], length))
  input_file.write("\n")

 #Compute median value.
 list.sort()
 median = list[1]

 #Write output test vector in binary to "output_file".
 output_file.write(int2bin(median, length))
 output_file.write("\n")

 #Clean list.
 for j in range(0, M):
  list.pop()

#Close files.
input_file.close()
output_file.close()
```

Listing C.2: ema.py

```
#Python script for generating test vectors for EMA filter
    simulation.
import random  #For generating random numbers.
import time    #For timing each sort function with "time.clock()".

#Function for converting integer to binary.
def int2bin(n, count=128):
 return "".join([str((n >> y) & 1) for y in range(count-1, -1, -1)
     ])

length = 8  #Number of bits, length of testvectors.
X = 255     #Max value.
N = 100     #Number of testcases.
M = 3     #Number of input vectors.
list = []

#Erase file content.
w = open('_input.dat', 'w')
w.close()
w = open('_output.dat', 'w')
w.close()

#Open files.
input_file = open('_input.dat', 'a+')
output_file = open('_output.dat', 'a+')

#Generate N input and output test vectors sets.
for k in range(0, N):
 #Append M random input test vectors (SAM_i EMA_{i-1} ALPHA)to list
     .
 for i in range(0, M):
  list.append(random.randint(0, X-1))

 #Write input test vectors in binary to "input_file".
 for j in range(0, M):
  input_file.write(int2bin(list[j], length))
  input_file.write("\n")

 #Compute "EMA_i".
 sam = list[0]
 emaold = list[1]
 alpha = list[2]
 tmp = sam - emaold
 tmp = alpha*tmp
 tmp = tmp >> length
 tmp = emaold + tmp
 emanew = tmp

 #Write output test vectors in binary to "output_file".
 output_file.write(int2bin(emanew, length))
 output_file.write("\n")

 #Clean list.
 for j in range(0, M):
  list.pop()

#Close files.
input_file.close()
output_file.close()
```

Listing C.3: thcomp.py

```
#Python script for generating test data for threshold comparator
    simulation.
import random  #For generating random numbers.
import time    #For timing each sort function with "time.clock()".

#Function for converting integer to binary.
def int2bin(n, count=128):
 return "".join([str((n >> y) & 1) for y in range(count-1, -1, -1)
     ])

length = 8  #Number of bits, length of testvectors.
X = 255     #Max value.
N = 100     #Number of testcases.
M = 2     #Number of input vectors.
list = []

#Erase file content.
w = open('_input.dat', 'w')
w.close()
w = open('_output.dat', 'w')
w.close()

#Open files.
input_file = open('_input.dat', 'a+')
output_file = open('_output.dat', 'a+')

#Generate N input and output test vectors sets.
for k in range(0, N):
 for j in range(0, M):
     #Append random input test vectors to list.
            list.append(random.randint(0, X-1))

            #Write input data in binary to "input_file".
            input_file.write(int2bin(list[j], length))
            input_file.write("\n")

 #Compute threhold value.
        if list[0] > list[1] :
            result = 1
        else :
            result = 0

 #Write output comparison data in binary to "output_file".
 output_file.write(int2bin(result, 1))
 output_file.write("\n")

 #Clean list.
        for i in range (0, M):
            list.pop()

#Close files.
input_file.close()
output_file.close()
```

Appendix D

Tools

The following sections list the version information for the tools used in this thesis, so that the results can be replicated.

D.1 Common Tools

Common tools are:

```
                Synopsys Verilog Simulator (VCS)

                Discovery Visualization Environment
                      Version C-2009.06
                    Platform Linux RH 4.0
               DVE Build Date: May 19 2009 23:36:07
             Copyright 2007 Synopsys, Incorporated.
                      ALL RIGHTS RESERVED

                         VCD+ Writer
                      Version C-2009.06
                  Copyright 2005 Synopsys Inc.

                        PrimeTime (R)
                       PrimeTime (R) SI
                       PrimeTime (R) PX
         Version E-2010.12-SP3 for linux -- Apr 14, 2011
            Copyright (c) 1988-2011 by Synopsys, Inc.
                      ALL RIGHTS RESERVED
```

D.2 Synchronous Flow Tools

Tools for synchronous design flow:

```
           Design Compiler Graphical
                 DC Ultra (TM)
                  DFTMAX (TM)
              Power Compiler (TM)
                DesignWare (R)
                DC Expert (TM)
              Design Vision (TM)
               HDL Compiler (TM)
              VHDL Compiler (TM)
                 DFT Compiler
             Library Compiler (TM)
              Design Compiler(R)
Version E-2010.12-SP5 for linux -- Jul 17, 2011
    Copyright (c) 1988-2011 Synopsys, Inc.
```

D.3 Asynchronous Flow Tools

Tools for asynchronous design flow:

```
|_  _ |  _ _     _  [ balsa-c: Balsa -> Breeze Compiler ]
|_)(_\|_/ (_\ - (_  (C) 1995-2009, The University of Manchester

version 4.0
                                          _
|_  _ |  _ _    ._  _ |_ _    ._  _ |_ _ [..| _   [ balsa-make-makefile: Makefile generator ]
|_)(_\|_/ (_\ - |||(_\|\(-' - |||(_\|\(-'| ||(-'  (C) 2003-2008, The University of Manchester

version 4.0

|_ ._  _  _ __  _ ')._   _  [ breeze2ps: Breeze -> Postscript Converter ]
|_)| `(-'(-' /_(-'/_|_)_/   (C) 2000-2008, The University of Manchester
                    |
version 4.0

|_ ._  _  _ __  _     _  _   _|_  [ breeze-cost: Breeze cost estimation ]
|_)| `(-'(-' /_(-' - (_.(_)_/ |_  (C) 1998-2008, The University of Manchester

version 4.0

|_ ._  _  _ __  _      _.._   [ breeze-sim: Breeze simulator ]
|_)| `(-'(-' /_(-' - _/ ||||  (C) 2003, The University of Manchester
```