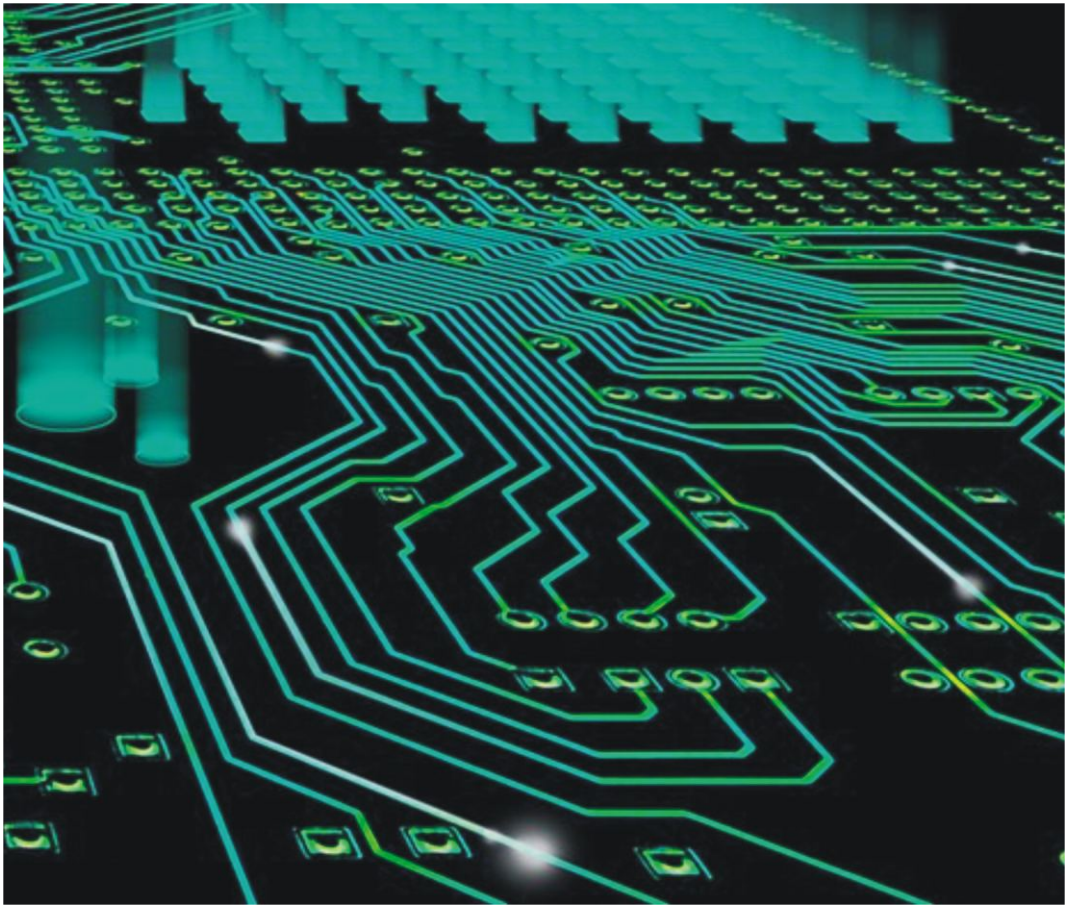
Martha Salomé López de la Fuente

Integrated Circuit Design

Using Open Cores and Design Tools

Achieving integrated circuit designs, by using open source hardware blocks and royalty free design tools, is now available for electronic engineers with a general purpose computer and internet access. Keep up with the trend!



SciencePG
Science Publishing Group

Integrated Circuit Design Using Open Cores and Design Tools

Martha Salomé López de la Fuente

SciencePG
Science Publishing Group

Science Publishing Group

548 Fashion Avenue
New York, NY 10018
www.sciencepublishinggroup.com

Published by Science Publishing Group 2015

First Edition

ISBN: 978-1-940366-44-9

To order additional copies of this book, please contact:
Science Publishing Group
book@sciencepublishinggroup.com
www.sciencepublishinggroup.com

Printed and bound in India

Preface

This book presents and explains the design of Integrated Circuits using open cores and open source design tools. It covers design aspects for all of the circuit elements: a processor (the Open RISC 1200 or OR1200), program memory, data memory, external address and data buses, communication port, interrupt controller, internal bus, clock, reset, and GPIO ports. For this purpose, all the hardware cores are open source and the fabrication technology is low cost. Detailed aspects of the design process are explained, such as application software optimization, small memory usage, memory intensive algorithms versus computation intensive algorithms. Also, an analysis of several application and research fields is presented, so the designed and implemented circuit used in this book as an example, can be used in other applications, with little or no modifications. Besides, a detailed design flow is explained, showing calculations for every design stage; the design flow covers synthesis process, area optimization, power and speed calculations, IO ring definition, and Place&route of the components and conections. Finally, two different implementations are presented: low-cost high volume, and medium-cost low volume: a) Technical data of the Integrated circuit implementation is presented and explained. b) An alternate implementation is also presented, using a development board with an ARM processor, especially useful for one-shot implementations. We hope that this books can help those electronic engineers with innovative ideas that can be implemented in an integrated circuit, without needing a big brand behind them.

Contents

Chapter 1

Introduction

1.1 Integrated Circuits

Integrated circuits are based on transistor logic, which means that logic gates (such as AND, NOT, NOR, EXOR, etc), resistors and other components are connected and packed together to achieve a specific function. This way we've been achieving, for a long time, small, for a long time, small circuits to build more complex circuitry using simple functions to build more complex systems.

For learning purposes students use those small circuits to build more complex systems. An easy example is to use several And Not gates to build an Adder or a Multiplier (Figure 1.1).

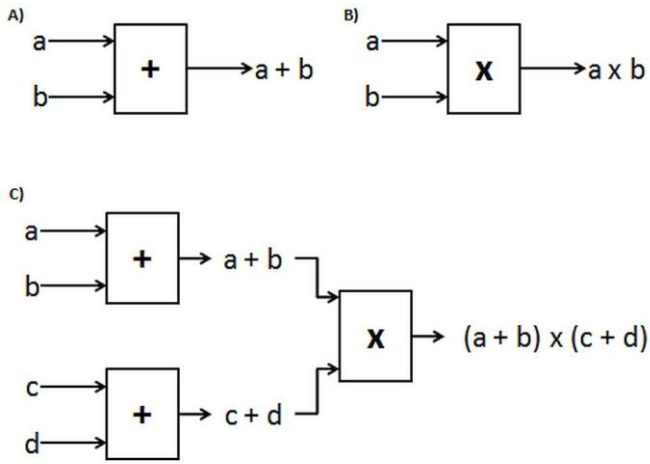


Figure 1.1 *Examples of: A) Adder, B) Multiplier, C) Building bigger blocks: Adder and Multiplier.*

In the process of building more complex circuit based on simpler ones, several questions arise: "Can all the circuits needed for a specific function be integrated into one single integrated circuit?" And, "is there a limit, in either size or technology, to build bigger and more complex integrated circuits?"

Well, the answer is yes. You can implement any mix of simple circuits to develop a more complex function, if you find it useful for a potential application. And, no, there is no limit for a design system, as long as you are willing to pay the design price for power consumption, circuit area, and speed.

The original purpose of the integrated circuits was to replace the bulky and energy hungry bulbs, so the initial functions were oriented to represent logic decisions, for example "if the door is open then activate the alarm", or "if the motor speed is greater than 100, close the valve". At this point, logic circuits have their size and energy consumption, so it was easier to have smaller and cooler control rooms making decisions over production processes. Then, what we can call the steady era, during the 80's and 90's, when the adders came along, multiplexers, encoders, memories and micro controllers. Everyone was fascinated about what they were able to achieve with those circuits. No one would have predicted what this area could become later, when information technologies and the internet modified all previous concepts about computing and processing. With the Information technology boom, came the data processing boom, in a way that more and more complex operations were needed, as well as communication protocols and the need for faster processing.

After these rapid growing technologies it is not easy to predict what will come next, but some approaches can be made. Silicon and transistor technologies are reaching their limit about minimum size and power consumption, which leads to other technologies being, explored (Figure 1.2). Maybe a totally new technology for circuits could come soon, and all of what we've known now should be reconsidered again. But that is what evolution is about.

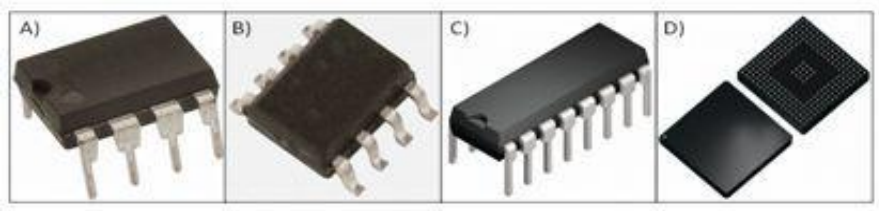


Figure 1.2 *A) Memory EEPROM series 93C56C-I/P, B) Multiplexor AD8180ARZ, C) Encoders Priority SN74LS148N and D) Processors TMS320VC5510AGGWA2.*

1.2 Digital and Analog Components

There is another classification for circuits according to the kind of signals and voltages it manages. Digital circuits are those which work on only two voltage levels, one level to represent a binary 1 and the other to represent a binary 0. All data and information in these circuits is represented, operated and stored, using 0's and 1's. Any value can be represented by a combination of 1's and 0's, and be interpreted back as the original value. Analog circuits contain signals in a wider and continuous range of voltages and currents, so physical variables such as temperature, pressure, volume, and speed can be represented by a scaled value. For example, a Temperature of 50 Celsius can be stated as a value of 10 if a scale of 0 to 50 is used for the range of 0 to 250 Celsius.

In this book most of the signals will be digital, except for some sensors and actuators that can work with analog signals. Each exception will be stated so there is no confusion.

1.3 Combinational and Sequential Circuits

Another circuit classification is based on how outputs are produced. According to the way a circuit delivers an output, either it is a signal, data, or a result; they are classified as Combinational and Sequential circuits.

Combinational circuit outputs do not depend on one another for the next output or result. This means that the moment when a function is performed does not change the result, because it does not depend on previous results or data from previous Actions.

Sequential circuits are time dependant; this means the result of their operation may be different depending on the moment a situation occurs. Specifically, a circuit that contains a program in the memory executes the program in a sequential way, and the outputs produced are taken into account for further operations.

1.4 Clocked or Timed Circuits

A lot of circuits require synchronization between systems or circuits; it means that several, or all, of the circuits involved in a bigger system should perform their operations at the same pace or rhythm, since they depend on each other’s results in order to know what should be done next. In this circuit a clock marks the pace for the execution. A complex system may have one or several clocks, just as long as they remain synchronized for the intended purpose.

1.5 Circuit Size

Integrated circuits come in all sizes. The more gates or transistors are inside the circuit, the more complex the function would be, the larger the circuit looks on the outside, and the more pins are available on the external packaging. A general way to classify the circuits by size is as follows, having no precise boundary between sizes, but: SSI for Small Scale integration, MSI for Medium Scale Integration, LSI for Large Scale Integration, and VLSI for Very Large Scale Integration.

Something interesting happens when circuits become bigger and bigger. When you notice, as a circuit designer, that your circuit and functionality are getting more complex, you will probably conclude that you need a processor instead of individual circuits. After that thought, you lead to this next one: if I already have a processor in my design, could the same circuit be used in other applications other than the originally intended? The exciting answer is yes! A circuit, when it contains a processor, can be easily adapted to perform other functions, with small additions or modifications. Here is where circuit design becomes more interesting: Once you know how to design and implement a complex function using a processor within your integrated circuit, you are ready to implement any function you can imagine.

1.6 Design Process

Designing an integrated circuit starts as any project does: with the idea of what you want to develop. So you start saying something kind of like this: I want to have a circuit that takes anyone’s age, weight and height, then it measures his temperature and blood pressure, and as a result it can predict how long the person

will live. Once you are able to state your intention in known words and clear intentions, you are ready to start designing it. The next step is to elaborate what you will need to measure, capture, calculate and process. For this example you will need: I will need a temperature sensor, a blood pressure sensor, a keyboard so the person can type his age and weight, a processor to run the calculations, a memory to store the calculation program, and a display to show the user the result. At this point you are describing your project in a very known and used way in this field: Inputs, functions, outputs. It means that you have acknowledged what data you need as input, what functions the system will be performing, in any case, everything will be easier if you can think of your idea in these terms: for each variable that my project needs, a sensor should be connected to it; the more complex the functions are, the longer the program will be; the more outputs the project will deliver, the more complex the display or interface will be. At this point, the clearer or more specific you can be on those three aspects (inputs, functions, outputs), the better. You will start from there until you get to the complete circuit, connections and elements.

A normal design process include iterations between what you want, what you achieve, what users say they need, and so on. Usually, the final implementation differs a lot from the initial circuit idea. We will discuss in detail the design process later.

1.7 Simulation Process

Once you have completed your design in paper, and your hand calculations show that it works as you want it to, a simulation is needed. You need a computer aided design tool to prove that what you are hoping to happen will happen. As in any other engineering field, there are many tools to learn and use.

For the purpose of this book, we will classify tools in Open source and licensed tools. Many universities pay licenses to software companies so their students have access to complex and professional tools. If that is your case, you can check with the system administrators what software they have for circuit design. If that is not your case and you are an independent designer, you can rely on open source code and simulators, since licensed software is never cheap and not worth it for a single design.

1.8 Implementation

Many enthusiast designers often risk their time and money by implementing their idea without the certainty that it will work properly, it means, without simulating it. It is up to you if you want to buy circuit components, sensors, and such, relying only on your hand calculations. The more complex your design is, the greater the possibility that it won't work as you expect it to work. After a successful simulation you have several options for implementing your design: If it's not intended for mass production you can use either the proto board version, which is cheap and good for prototyping, but only for small to medium circuits, or the FPGA version (Field Programmable Gate Array) which lets you know in a more accurate way the size, power consumption, transistor count, and speed of your circuit, when it becomes an integrated circuit. The development board version, allows you to store and run your program, check the output, connect your sensors, and test your program over and over until you are satisfied with the result. In the market you can find development boards for as low as 40 US dollars. The last option for implementing your circuit is to pack everything it needs into one single integrated circuit. This is what this book is all about: how to get a whole idea into one single integrated circuit. The best option for mass

production is an integrated circuit that contains everything it needs to perform its function. As you may notice, an integrated circuit is a final version of an idea, which you cannot modify it, has been fabricated. Nevertheless the circuitry and connections cannot be modified, but remember that if you put a memory inside your integrated circuit, and the memory has a program, and you were careful enough to consider a programming interface then you can modify the program in your circuit. Even it can serve as with different purposes if your design is made open enough.

1.9 Fabrication Process

Once you've finished your integrated circuit design you need to have the design in standard format files, so you can send these files to a fabrication company and then receive from them your shiny and brand new integrated circuits. Fabrication processes uses silicon wafers, and stamp your circuit on the silicon, so transistors and connections are made as you specified. Then the small piece of silicon is packed to protect it, adding pins for you to connect the circuit to whatever other system, sensors, or components it will be connected. Most of the well-known fabricants have University programs, in which they charge less for university projects, or let you join teams in order to split the fabrication costs. The fabrication process (Figure 1.3) involves "clean rooms" and very expensive equipment, as any dust particle, even if it is as small as a few nanometers, can get into the circuit, and produce a malfunction.

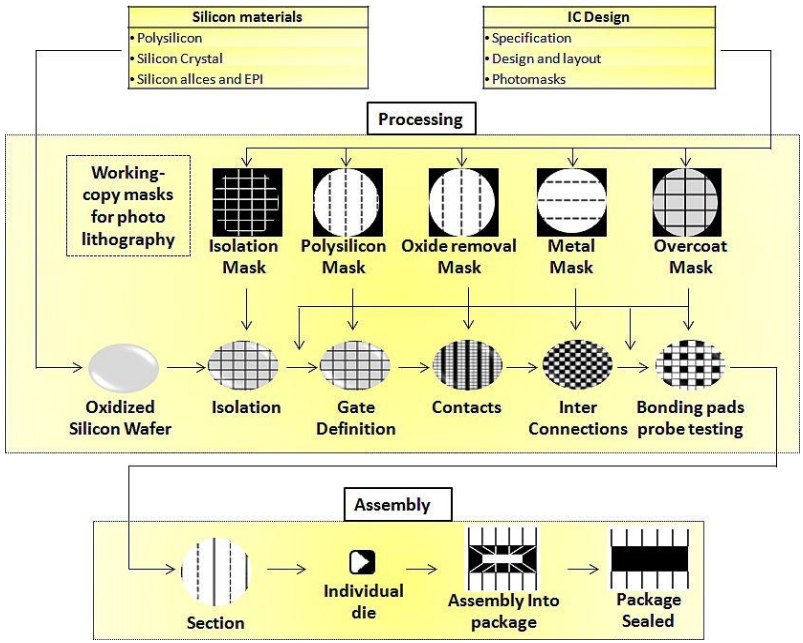


Figure 1.3 *Fabrication process.*

1.10 Marketing Process

In the beginning you intended your circuit for one purpose, so by now you probably know if it will be the main module of a stand-alone device, such as a microwave oven or a dishwasher machine. Having stated that integrated circuits that perform a specific function are intended for a specific device, you will need to get your circuit to the market, maybe not directly to the consumer, but to the device manufacturer.

Chapter 2

Processor Based Integrated Circuits

You may start as circuit designer using gates, adders, multiplexers and so, but soon you will find that it is better to include a processor and a memory in your design, so additions in functionality are easy to integrate. Here is where you notice the hardware versus software implementation advantages. Let's explain how hardware and software are involved in your design decisions: Any project idea can be implemented completely using hardware, it means that every single decision is made by a transistor or set of transistors, connected in a way that will result in a voltage indicating what will happen with an output signal. If your project has many decisions to make, data to store, operations to make, you can use the simulator, estimate how many transistors the circuit need. On the other hand, if all the decisions your project is making are translated into a programming language, you can estimate how much memory the program will need to be stored. And the memory size can be easily translated into transistor count. This way you can compare if your hardware based version is smaller –using less transistors- than your software based version. We will get into this subject with more detail ahead.

2.1 Relevance and Potential Uses

Stimulation systems provide signals and test patterns to be used in a variety of applications. Potential uses and applications for stimulation systems constantly increase as existing tests and lab procedures are desired to be miniaturized or new tests are conceived, whether they are for Lab analysis, prosthetic testing, pollution analysis, or point-of-care.

Current experiments need a stimulation system so tests can be repeatedly performed in order to store results, perform analysis, and obtain statistics and finally report state of the art conclusions and results. On-going experiments

whose focus is to obtain novel results on a specific research area should be supported by a stimulation system that eases the experiment and allows researchers to define and change the stimulation patterns and tests.

Once stimulation systems are found useful in a specific application, the next natural step is to add intelligence to the system, so it can precisely reproduce test procedures, improve performance by learning from previous results, and evolve according to upcoming needs. Potential uses for this stimulation system include a wide range of experiments, from detecting pathogenic cells in fluid samples, bacteria or viruses in blood and urine droplets, microbes or fungi in food items and water, and also target agents in the environment, the human body, or industrial processes.

Specifically, particle manipulation experiments are expected to become part of the everyday life, so usual lab tests can be performed by a miniature device, in site, and by non-specialized personnel. In characterization efforts, all kind of particles and cells are separately stimulated in order to determine their characteristics so manipulation and test procedures become known and can be used in future tests.

Current research is also going to automated tests using stimulation systems, where prosthetic devices are analyzed to check if they react as their human counterpart does; a set or sequence of signals, similar to those generated by the brain in order to control or to sense that body part, is applied to the prosthetic part to determine if proper behavior has been achieved and the body part is ready to use.

Besides, when a stimulation system is configurable as the one presented here, its use may extend to related applications, such as cell disruption, embryo viability tests, DNA manipulation, and serial/automated medical lab tests.

2.2 Design Variables

As circuit designers learn along their experience, there are trade-offs between design variables. The most important of these variables are: circuit speed, circuit area, and power consumption. Secondary variables are pin-out and time to market. Circuit speed involves how fast the used processor and peripherals will run; area is the space the circuit will take in the silicon wafer; power consumption refers to how much battery power will take to fully and continuously execute the application program. All these variables are discussed in detail ahead.

2.3 Chips and Intelligent Systems

Early designs and implementations for portable labs are initially prototyped on development boards, printed circuits, or FPGAs, and there are some design efforts to produce a miniature device which may eventually lead to low-power Lab-on-chips and portable labs. Along with miniaturization efforts, intelligent testing goes its own way on current research work; it will become part of the future fully automated lab processes and tests, so it has to be defined in general terms and be able to be programmed for complex future tests.

About miniaturization and intelligence current developments, several works are referenced here: reviews of stimulation experiments using proposed or designed integrated systems, automation of effective and programmable particle

manipulation using MEMS and a bio-cell processor, DEP filters which could continuously eliminate cells suspended in water, and so on.

An early chip proposal was the engine for a micro-fluidic Lab-on-Chip system; it was presented by Gascoigne as a high voltage integrated circuit which transports droplets on programmable paths; it creates forces over multiple droplets while varying electrode excitation voltage and frequency. Electrodes are driven with a $100V_{pp}$ periodic waveform; the maximum waveform frequency is about 200Hz. This prototype chip has a 32x32 array of 100V electrode drivers. Fabricated in a 130V SOI CMOS technology dissipates 1.87W max, in a 10.4x 8.2 mm^2. The chip is programmable: the routes of multiple droplets may be set arbitrarily within the bounds of the electrode array and the stimulation waveform amplitude, phase, and frequency may be adjusted.

Newer proposals present designs for Lab-on-a-chip integrating one or several sub-systems: Delizia proposes a large array of capacitor sensors for detecting dielectric permittivity variation. It uses an 11-bit resolution ADC at a sampling rate of 100 Kilo-samples/sec; it is implemented in 0.35 μm CMOS technology. The noise coupled to the signal at the chip pad is reduced by using an on-chip analog-to-digital converter. Simulation results show a SNR=65.7 dB and an ENOB value of 10.6b. Its power consumption is about 150 mW. Readout chain is implemented in 0.35 μm CMOS technology with a 3.3 V supply voltage.

Keilman presents a proposal of a bio-analysis system that may be part of future low-power bio-analysis platforms. The analysis technique uses the electro kinetic phenomenon for noninvasive biological cell manipulation. This work generalizes the concept of test micro-structures using standard CMOS process by providing a generic electrode structure, which, when integrated with a

processor, is capable of generating an arbitrary electric field shape, thus facilitating a programmable sequence of different cell manipulations.

Shih et al proposes an adaptive biochip integrating DEP traps and a programmable array for the multi-sorting applications of bio-molecules. The magnitude and direction of the DEP force are controlled via the distribution of time-variant non-uniform electric fields. The voltage on each individual electrode of the multi-sorting array is programmable.

When a programmable or configurable system is desired, a user interface comes in hand for operation, since it allows repeatable and reliable setting of test parameters. There is on-going work on programmable and configurable testing, although it does not come together with miniaturization efforts. A device presented by Manaresi is a 64 mm^2 chip implemented in a two-poly three-metal 0.35 μm CMOS technology, featuring an array of 320x320 actuation electrodes, 20μm x20μm micro sites, including addressing logic, an embedded memory for electrode programming, and an optical sensor. The chip enables software-controlled displacement of living cells, and the manipulation does not damage the viability of the cells.

Similarly, Jungyul Park presents an integrated MEMS-based bio-cell processor; the purpose is the automation of transporting, isolating and immobilizing individual embryo cells for effective manipulation.

An interesting topic on SoC is that modular designs should be able to integrate between them by using standard existing interfaces so a complex system is built by connecting several simple functional blocks. New developments of digital blocks or cores should take integrated systems for particle manipulation to the next level: future designs should include in one design the stimulation system, the

fluidic device, the actuating elements, the sensing circuitry, the data collecting system, the analysis system and the storage device.

As an early example, there is a software configurable architecture able to implement a variety of AC electro-kinetic techniques. The architecture is developed as a flexible IP block and in conjunction with integrated micro fluidic devices and other third-party IP blocks, form the analysis function. This design is basically a two dimensional randomly addressable electrode array being driven by one of four sinusoidal analog signals. The so called Lexel™ array and supporting circuitry are designed on a single chip using a standard 0.18μm CMOS process.

Table 2.1 *Referenced works on Intelligent Labs-on-Chip and Bio-Chips.*

Year	Category	Application	Focus	Integration	Intelligence	Implementation Specifications
2003	Proof of concept/ DEP processor	Droplet manipulation	Droplet based chemistry	No, all external elements	Application dependent	Proposal, fluidic processor versatile platform
2004	Proposal/ Particle manipulation	Diagnostic instrument		Stimulation, circuit, electrodes array	No	No
2004	Design / Bio-cell processor	MEMS, embryo cell	Manipulation automation, MEMS based bio-cell processor	Processor, DEP valves	Automated tests	MEMS based bio cell processor
2005	Implementation/ Low power bio-analysis platform	Bio Analysis		Stimulation system, fluidic device, 2D electrode array.	SW configurable, IP modularity, four output channels	IC
2007	Design/ Stimulation chip	Lab-on-chip	Stimulation systems for electrode arrays	Electrode array, excitation circuit, drivers,	Programmable droplet routes and waveform parameters. Expandable architecture	Fout=200 Hz; demo chip in a 130-V 1.0 μm SOI CMOS. 1.87 W, 10.4 x 8.2 mm^2
2007	Design/ Programmable Bio chip	Bio-molecules multi-sorting		DEP traps, programmable array.	Programmable stimulation	IC

Year	Category	Application	Focus	Integration	Intelligence	Implementation Specifications
2008	Design/ Stimulation and read-out Chip	Lab-on-Chip	Capacitor sensors and actuators array	Sensors array, ADC, Amplifiers, Readout chain	Programmable gain	Simulation for actuators. Implemented in 0.35 μm CMOS
2009	Design/ Field array micro-system	Bio-medical		Sensors, actuators	No	Integrated circuit (IC)

Table 2.1 summarizes the work done by referenced research works that go on the line of Systems-on-a-chip and Lab-on-a-chip. Scope refers to the level achieved in that work: a novel proposal, a detailed design, or a finished and tested implementation. Category refers to the target element in a Lab-on-chip structure; it can be a fluidic device, a stimulation chip, an actuator/sensor set, etc. Focus summarizes de orientation of the work so it shows that it is specific for a particular experimental environment. Intelligence refers to the capabilities of the system to be considered intelligent: programmable functions, uses a processor, configurable operation, includes user interface. Integration refers to the elements covered by the design and the possibility to integrate it into other existing modular designs: a fluidic device, sensors and actuators, stimulation circuitry, a standard user interface, and modularity or IP blocks usage. Implementation (intended or developed) for that proposal or design: printed circuit, integrated circuit, or simulation only. Application refers to the expected or target application, such as air and water pollution, lab test and analysis, medical treatment, or generic particle manipulation.

2.4 Opportunity Areas

The detailed analysis of the state of the art on this area allows us to determine the need of a system like the one presented in this work. A wide range of

applications are using electrical signals to stimulate a fluidic device for experimentation on particle manipulation. The majority of those tests are performed manually controlling the parameters of the applied signals. Also, most of the experiments are specific for a certain type of particle, using certain waveform within a narrow frequency range. This design tackles the need for automated test procedures, configurable operation, miniaturization of the design and a modular design style to ease integration of this system into existing or future designs.

Existing stimulation systems are about using limited logic to synthesize a desired frequency and deliver it to an experimental device; from there, a specific and non-configurable signal or pattern is obtained, and it can be used only for that specific purpose.

The system in this work is a processor based design that can execute a variety of application programs, a memory system that is optimally used to contain program and data while delivering a variety of signals and patterns in a wide frequency range, and the configuration capabilities to allow users to adapt it to specific tests and applications with no modifications to the hardware or software.

The automation of testing and stimulation procedures obtained from this system can speed up current research work by providing a reliable way of repeating, configuring and adapting the system to a specific application, whether it is used as an autonomous system or integrated to an existing Lab-on-a-chip.

2.5 Systems and Labs on a Chip

The integration in one chip of all the components needed for an application, known as System on a Chip (SoC), has been the optimal implementation for

many embedded systems. In functionality it can go as far as designers' dream of it, from containing a little logic up to a processor and peripherals that can be programmed to perform multiple functions.

A SoC containing a set of components like processor, memory system, peripherals and configurable application software can lead to a design that is reliable, modular, programmable and easy to integrate into other designs.

A generic diagram for a design can be presented as in Figure 2.1: a user interface configures, programs, and operates a stimulation system; this system delivers selected electrical signals and patterns to a fluidic device containing the sample and particles to be manipulated; a sensing system can collect info from the stimulation effect and, either go directly back to the interface, or pass through a characterization system where it can be useful for identifying a specific type of particles.

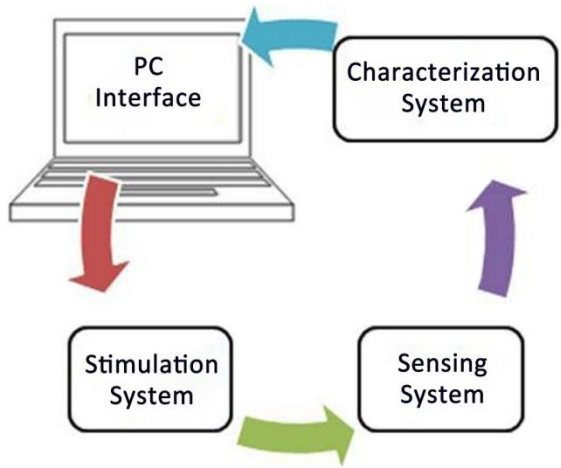


Figure 2.1 *The electric stimulation system in a particle manipulation environment.*

Chapter 3

System Design

3.1 System Definition and Specifications

Based on referenced research works, the definition of what a stimulation system should be and how should it be implemented was achieved, in order to cover particle manipulation tests and procedures over a wide range of applications. Research areas include:

Frequency range: Based on the state of the art in particle manipulation, define and justify a frequency range eligible for a wide range of applications in manipulation procedures and tests.

Frequency synthesis methodology: Explore existing implementations, examine their applicability to this work and decide if the output frequency to system clock ratio can be achieved with them or if a novel methodology is needed to generate data for single and superimposed frequencies.

System design: Explore the design options for the system in this work and justify the selection, from logic-only, programmable array based, and processor based implementation.

Optimization and modularity: Analyze the trade-offs of the selected design scheme about performance (output frequency to clock frequency ratio), circuit size, and power consumption considering portable applications as the target. Explore the trends on intelligent systems about modularity, re-usability, integration and interconnection capabilities. Explore the core-based design methodology.

Configurability: Define and justify the parameters that should be open and configurable in order to obtain an electric stimulation system that covers the majority of electro-kinetically driven micro-fluidic devices.

Prototype implementation: Define a feasible prototype hardware implementation to run the application program so functional specifications and frequency synthesis methodology can be evaluated.

3.1.1 Motivation

Existing systems for manipulation and separation of particles depend on previously known information about the type of target particles or by experimenting on them; such experiments consist of controlling and changing or repeating electrical stimulation, analyzing response and sweeping signal parameters until desired results are achieved. An automated, programmable, configurable system is needed where reliable stimulation is needed for efficient and faster advances on research work about particle manipulation. Advantages of an automated, programmable, intelligent manipulation system:

- Multiple tests can be done and repeated by programming test sequences.
- Previously programmed test parameters for a known test sequence can be stored, accessed, and repeated.
- More reliable data results are obtained due to precise reproduction of test parameters.
- User interface allows rapidly configuring and operating the system for new tests and procedures.

- An intelligent design targets future Lab-on-chip implementations and portable Lab devices.
- A programmable system allows to run original application or to load a new one.
- A scalable design provides interconnection and communication channels so it can be integrated to other systems.

3.1.2 Statement of the Problem

There are current problems and limitations in particle manipulation procedures and research works, so present needs should be detected and solved; overcoming the state of the art and anticipate for future needs in stimulation systems would allow researchers to speed up experiments and results.

The trends show that experiments need more controlled testing environments by using more complex electric stimulation, which only programmable systems can deliver, like signal composition, dual frequency signals, traveling wave fields, mixing sine with square and triangle signals, and what may come in the future.

Besides, if frequency range of output signals could cover a wide spectrum of particle types, sizes, and shapes, research work would be more efficient and might reveal results from previously unknown experimental circumstances.

Last but not least, current implementation schemes for digital frequency systems should take as primary goals a low power, minimum size, and high performance design.

Figure 3.1 illustrates how the research work in all the related disciplines and the corresponding tasks lead to specific outcomes and contributions of this work in each of the four related research areas: the effect of electric fields in electro-kinetically driven fluidic devices, frequency synthesis methodologies, Lab-on-Chip systems, and System-on-Chip design.

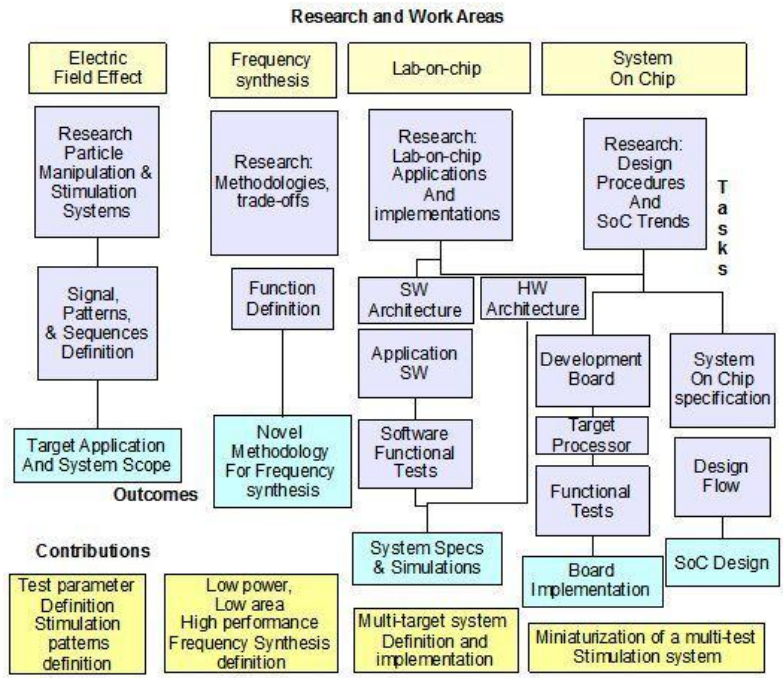


Figure 3.1 *Performed tasks, achieved outcomes and contributions made in the four research areas.*

The overall goal is to specify, define, design, and implement an open processor-based system that allows users from different areas to configure and automate their tests over a specific target particles or cells to obtain reliable and repeatable results in order to achieve the desired mobility effect. It is also desirable to have configurable system variables and test parameters that can be selected or programmed before the experiment or test is executed.

3.1.3 Proposed System

This work presents a processor-based stimulation system to generate signals and configurable tests for stimulation of micro-fluidic devices. It delivers multiple waveforms and patterns to cover a wide range of experiments and applications.

Specific tests or sequence of tests that should be made on specific particles may not be known by publication time since this is a developing area, so this system is designed to be configurable and to deliver a variety of signal patterns and combinations within a frequency range. The design consists of a set of cores integrated as a System-on-Chip (SoC) to configure, operate, and execute a stimulation system which delivers desired data.

This stimulation system includes user interface capabilities for configuration and operation, a memory system to upload and contain the application software for frequency synthesis, a processor to execute the program, and output ports to deliver data from synthesized frequency as shown in Figure 3.2.

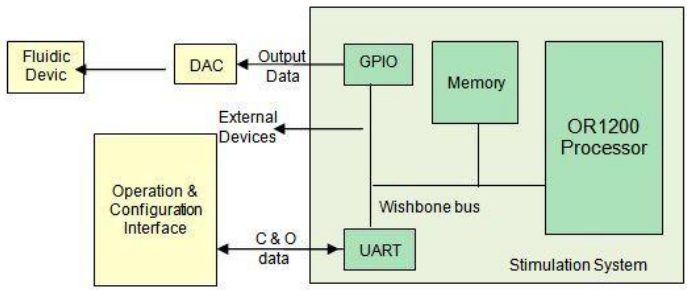


Figure 3.2 *Proposed systems.*

This system is also designed to favor an easy integration to existing or on-going designs of Lab on a chip: it provides input/output Wishbone buses for data and instructions so the system can be application independent by using a

ROM based Bios that loads the selected application software depending on the desired use. Existing or proposed Lab-on-chip systems to be connected to this stimulation system, which is using another standard communication bus like AMBA, can use a converting bridge for interconnection without changing the current design.

One of the possible implementations presented, shows how a different application program can be uploaded before operation, so modifications and additions to the program can be made and tested outside the system and later uploaded to an in-chip memory for operation. This capability, besides making the system adaptable to future applications, makes possible its integration to existing systems.

The user interface allows configuring the system, select mode of operation, select desired type of signal, selecting single or superimposed frequencies, and visualizing data being delivered. User interface interacts with the system via a standard serial port.

The memory system consists of a ROM to contain the boot-loader which uploads the application software at the beginning of operation and Harvard architecture of RAM to contain the program and the data for operation. The processor selected for the SoC design, the OR1200, is the best option from the available open source cores, and its corresponding instruction set covers the needs for this application software.

The on-chip communication is achieved using the Wishbone bus, which is the standard bus for open source cores, and allows a smooth integration of all the components in the system. Two Wishbone buses, one for data and one for instruction, are taken outside the chip so it can be integrated to other systems.

The hardware architecture for the chip is oriented to low power, low area, and low execution times, and by using open source cores this is a design that can be completed with no licensing cost during design and fabrication stages.

The application software implements the novel frequency synthesis methodology designed during this work, so it optimizes hardware resources such as memory map, instruction set, and system clock, in order to achieve maximum output-frequency/system-clock rate in output signals. The software can be tested on development boards based on the same or similar processor.

This system is also designed on a modular basis so it can be integrated, as is, into Lab-on-Chip systems or by adding new driver cores and the application software is designed in an open-source style so it can be configured or extended for future applications.

3.1.4 Frequency Synthesis

The novel frequency synthesis methodology developed for this system integrates the advantages of both, memory intensive and computation intensive approaches into one new synthesis methodology while keeping a low implementation area, low power, and high performance design.

For this system a look-up table is used to store base sine sampled data for a complete sine cycle. As any digital design the best tradeoff between hardware and software implementation should be selected: hardware is used for data storage and software for data processing computation. A software implemented algorithm is defined to select data from look-up table for target frequencies and store it in temporary tables; process data from temporary tables to get single or dual frequency data samples, and store them in output buffer tables.

Finally, and most important, a computation-free algorithm loads data from output buffer table and stores it in one or more output ports depending on the operation mode. An external conditioning circuitry including a DAC, a current to voltage converter and a voltage amplifier converts sampled data into the finally delivered analog signal.

3.1.5 Comprehensive System

As state of the art, research shows highly controlled experiment environments can be achieved when using more complex stimulation, so this system needs to deliver configurable multi-waveform, dual-frequency signals to speed-up research work on multi-particle manipulation tests. The system architecture has the foundation for control purposes, data storage and signal processing; it can be customized to achieve particular control and operation purposes of stimulation systems.

The mix of single-frequency signal generation along with signal superposition and system configuration capabilities, allows a wide control range on stimulation tests. Besides, with minimal modifications other waveforms and patterns can be obtained. A frequency range sweep can be run as a sequence of several user selected exposure times to analyze results under several stimulation conditions on the same experiment.

Execution times are lowered to its minimum so maximum output frequency is dependent only on the processor specification. Besides, a size optimized routine does not change for different generation modes or for different waveforms, so memory usage is kept at a minimum regardless the operation mode.

Memory architecture is designed for minimum area: data samples for sine, triangle and saw tooth waveforms are stored in base data tables using them, temporary tables are constructed based on selected output frequency and desired number of voltage steps; output or buffer tables are finally calculated after a time match operation depending of the number of channels to be updated simultaneously.

Data pre-processing and table preparation reduces computation instructions during signal generation achieving maximum output frequency to clock frequency ratio. The system can generate any periodic waveform as long as it is stored in memory data tables.

For signal updating, multiple simultaneous writes to output port are made so no loss in output frequency occurs when two or more signals are being delivered simultaneously. This port partitioning scheme is particularly convenient for this application.

The proposed system combines efficient use of hardware and software resources: minimum generation code, no computation during synthesis, and minimum memory access times.

3.1.6 Scope and Limitations

About research, the commitment is to review the state of the art on the four areas mentioned to identify the common ground for electric stimulation of fluidic devices, to keep-up with trends, and to anticipate to future stimulation needs. About new developments the challenge is to deliver a flexible and programmable system which runs a novel signal generation methodology and to prove its functionality.

About system design: the goal is to identify the system requirements, to define its functional specifications, for System-on-Chip -standard functionality version- and for development board -extended functionality version-, and to use available software and hardware resources to implement the design.

Limitations are related to time and to available resources: time because design decisions for this system are made based on what current research work shows and what can be identified as a trend; and resources are related to budget dependencies and access to licensed or open CAD tools to achieve the intended design.

Implementation on development board is limited to available processor, instruction set and communication ports specifications for that board. For chip implementation, system specifications like circuit area, power consumption and maximum output frequency are defined and limited by three factors: fabrication technology, physical libraries available, and efficiency of application software; the first two are resources dependent and the third is designer dependent.

Signal waveforms to be delivered are sine, triangle, and square and saw tooth wave. For dual superimposed frequencies the ratio between frequencies define the memory size needed for temporary and buffer tables: if frequencies are not exact multiples a hyper-cycle for resulting signal is not possible or very large, and that leads to unfeasible, large or infinite, memory needs.

3.2 Software Architecture

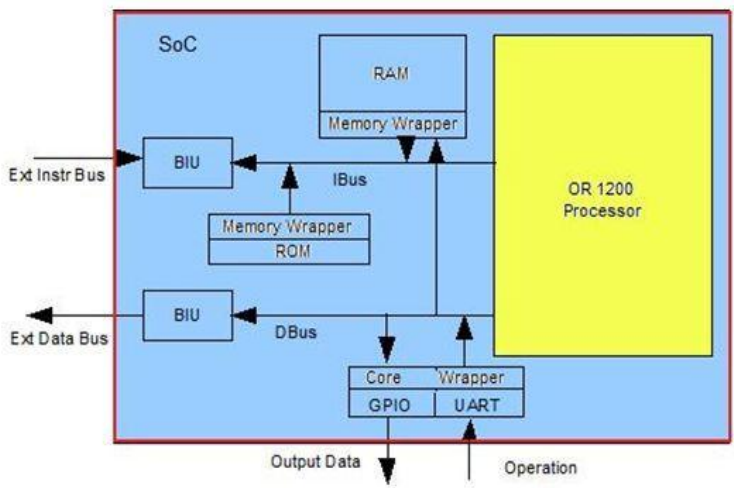


Figure 3.3 *System-on-Chip block diagram for the platform based design.*

The hardware architecture of the SoC is a platform and bus based architecture; it uses a selected set of open source cores and the Wishbone on-chip communication bus. The selection of the OR1200 processor is selected due to its previously demonstrated implementations in FPGAs and to its open instruction set. The on-chip memory array may contain the application software developed specifically for stimulating fluidic devices or a boot-loader to up-load different applications. In chip data, memory contains a complete cycle of the three base waveforms in 256 samples of 8-bit data each. A UART port is used to configure operation, to program sequence tests, to control operation and to visualize data during execution. The four 8-bit GPIO ports deliver processed data points for output signals, which may be, according to selection made in configuration: sine, triangle, or saw tooth and presenting single, dual or superimposed frequencies. Figure 3.3 shows interconnection between in-chip blocks; processor and Bus Interface Units connect directly to Wishbone

instruction and data buses, and memories peripherals connect via wrappers. Table 3.1 details the function of each primary block in the SoC.

Table 3.1 *Function description for primary system blocks.*

Element	Description
CPU (OR1200)	RISC CPU, Harvard architecture, cache memory for data and instructions, operates at 250 MHz max using 180 nm standard cells TSMC technology.
Wishbone Bus	On-chip bus for cache, main memories and interface peripherals
GPIO	Grouped in four I/O 8-bit ports, from open source cores: used as inputs for configuration and operation, as outputs for data
Clock and Reset	Receives clock from crystal oscillator, generates clock and reset signals for system operation, base clock for processor blocks and ¼ base clock for Wishbone bus.
UART	Serial port controller provides connection for external configuration and operation device. Open core source is used.
RAM and ROM	Memory blocks built with Artisan memory generators for verilog, vclef and gdsII views.
Interrupt controller	Exceptions handler from open cores included in OR1200 architecture.

3.2.1 Processor Based Implementation

This particular implementation for the processor was based on the open source files of the OR1200. The Open RISC 1200 is a synthesizable CPU core from Open Cores.org; it is a configurable open source Verilog implementation of the Open RISC 1000 architecture. The OR1200 is intended to be used in a variety of embedded applications. Some open source software, such as Linux, has been ported over to the OR1200 platform.

The GNU tool chain, including GCC, has also been ported to the architecture to aid in software development. The clock cycle for the OR1200 is 250 MHz at a 0.18 μm, 6ML fabrication process.

Estimated power consumption of this processor running at 250 MHz and implemented in 0.18μm technology is less than 1W at full throttle.

Available libraries: 180nm from TSMC will be used for this design. System specifications will meet application software requirements for standard version.

Components: Processor has been selected from available open source cores. Memories have been obtained through memory generators, Peripherals have been selected from open source cores.

Design constraints: Constraints are considered in this order: Performance, Area and Power. Target clock frequency is around 250MHz for an 180nm technology implementation. Processor area budget is less than $2mm^2$; on-chip memory area is less than $3mm^2$. Power budget for OR1200 processor in this technology is 1W at full throttle, added blocks should not exceed that by more than 20%.

3.2.2 SoC Components

The OR1200 is a RISC, Harvard Architecture processor with basic DSP capabilities. As an open source, customizable, core it is not optimized for power or size. This particular implementation for the processor was based on the open source files of the OR1200.

The Open RISC 1200 is a synthesizable CPU core from OpenCores.org; it is a configurable open source Verilog implementation of the Open RISC 1000 architecture.

It specifies a Central CPU/DSP block, Direct mapped data cache, Direct mapped instruction cache, Data MMU based on hash-based DTLB (Translation Lookaside Buffer), Instruction MMU based on hash-based ITLB, Power management unit and power management interface, Tick timer, Debug unit and development interface, Interrupt controller and interrupt interface, Instruction and Data WISHBONE interfaces, and a MAC unit. Peripherals and a memory

subsystem may be added using the implementation of a standardized 32-bit Wishbone bus interface.

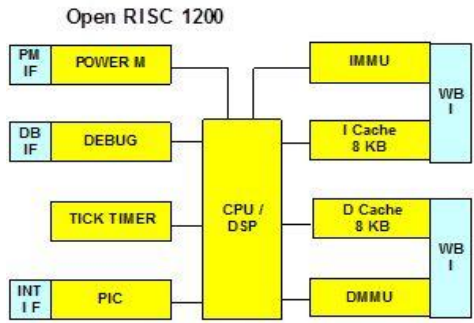


Figure 3.4 *OR1200 Architecture.*

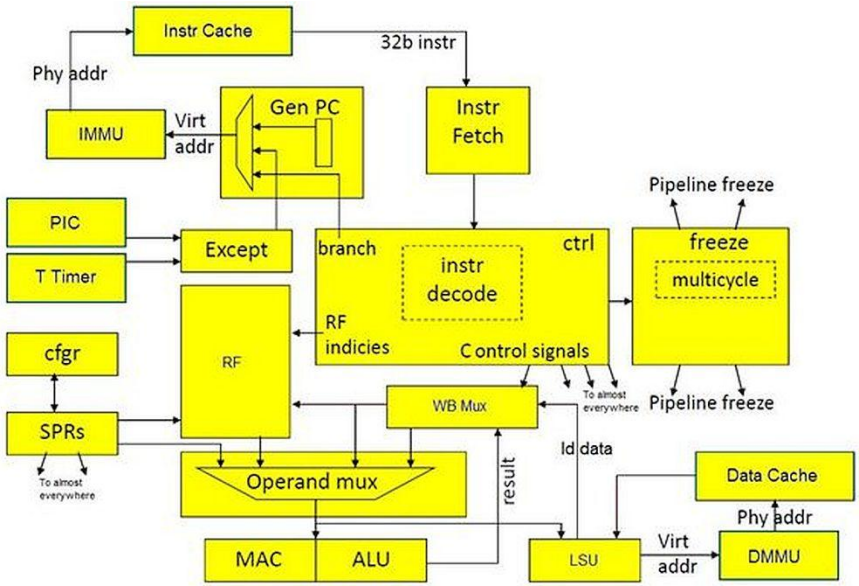


Figure 3.5 *OR1200 internal cores.*

The CPU is an implementation of the 32-bit ORBIS32 Instruction Set Architecture. It has five instruction formats and supports two addressing modes; it has a single-issue 5-stage pipeline, single cycle execution on most instructions.

It uses a Harvard architecture with separate MMUs for data and instruction memories, with support for virtual memory, a hash-based 1-way direct-mapped TLB with page size of 8 KB and a default size of 64 entries, and one-way direct-mapped D-Cache and I-Cache, 8 KB each.

Table 3.2 *Specifications of the OR1200.*

Concept	Specification
Architecture	32-bit scalar RISC, Harvard
Pipeline	5 stage, 32-bit integer instructions
DSP	Basic capabilities
Caches	Separate instruction and data, 1-way direct mapped. Configurable to 1, 2, 4, 8KB
Virtual memory	64-entry hash based 1-way direct mapped TLB for data and instruction MMU
Speed	Worst case: 150 dhrystone 2.1 MIPS at 150MHz (typical corner 250MHz) Best case for 180nm implementation: 300 dhrystone, 2.1MIPS at 300MHz
Size	Default configuration about 40K ASIC gates, 1M transistors
RTL status	Not optimized for speed or area
Instruction Set	Instruction unit handles only ORBIS32 instruction class. ORFPX32/64 and ORVDX64 instruction classes are not supported.
Configurable	Major characteristics can be set by user (see Core HW configuration table)
Communication bus	Wishbone, internal and for SoC interconnection
GPRs	General Purpose Register file is implemented as two synchronous dual-port memories, 32 words, and 32 bits per word.
Exceptions	Transparent to user software, same mechanism to handle all types of exceptions, control is transferred to an exception handler. See Exceptions table.
Interrupt controller	Direct enabled Int0 and Int1, Masked Int [31:2]
Tick timer	Clocked by RISC clock, re-start able, mask interrupt, Max count 2^32
Power management	Dynamically activated modes: slow and idle, Doze, Sleep, and Clock gating
Debug unit	Basic debugging; No watch points, break points or program flow control registers.
Clock & Reset	Core has several clock inputs: clk_cpu, clk_dc, clk_ic, clk_dmmu, clk_immu, clk_tt; all clocks must be in phase and as low skew as possible. Reset signal rest reset all flip flops when asserted high; when not asserted reset exception start.
Wishbone interface	Rev. B compliant, 32-bit bus width may connect to external peripherals and external memory subsystem.

Table 3.2 and 3.3 summarizes the specifications of the OR1200 selected for the design; Figures 3.4 and 3.5 shows the processor architecture.

Table 3.3 *List and description of the processor cores and the peripherals for the System-on-Chip.*

Core	Block description	Core	Block description
OR1200_alu	Arithmetic and Logic Unit	OR1200_lsu	Load and Storage Unit
OR1200_cfgr	Configuration Registers	OR1200_mult_mac	Multiply and MAC Unit
OR1200_ctrl	Control Unit	OR1200_operandmuxes	Operand Mixes
OR1200_dc_top	Data Cache	OR1200_pic	Programmable Interrupt Controller
OR1200_dmmu_top	Data Memory Management Unit	OR1200_pm	Power Management Unit
OR1200_du	Debug Unit	OR1200_rf	Register File
OR1200_except	Exceptions Unit	OR1200_sb	Store Buffer
OR1200_freeze	Freeze Unit	OR1200_sprs	Special Purpose Registers
OR1200_genpc	General Program Counter	OR1200_tt	Tick Timer
OR1200_gpio	General Purpose Input Output	OR1200_uart	Universal Asynchronous Rec/Trans
OR1200_ic_top	Instruction Cache	OR1200_wb_biu	Wishbone Bus Interface Unit
OR1200_if	Instruction Fetch	OR1200_wbmux	Wishbone Mux
OR1200_immu_top	Instruction Memory Management Unit	-	-

Communication Bus. The wishbone bus has become the standard communication bus for the open source cores. It serves as the in-chip bus and as the interface bus for the SoC with the external world, and as for Harvard architecture there are separated buses for data and for instructions.

Instruction interface is used to connect OR1200 core to memory subsystem for purpose of fetching instructions or instruction cache lines. Data interface is used to connect OR1200 core to external peripherals and memory subsystem for

purpose of reading and writing data or data cache lines. Table 3.4 lists signals for instruction lines (data lines are named dwb_xxx).

Table 3.4 *The Wishbone instruction bus.*

Signal	Width	I/O	Description
iwb_CLK_I	1	I	Clock input
iwb_RST_I	1	I	Reset input
iwb_CYC_O	1	O	Indicates valid bus cycle (core select)
iwb_ADR_O	32	O	Address outputs
iwb_DAT_I	32	I	Data inputs
iwb_DAT_O	32	O	Data outputs
iwb_SEL_O	4	O	Indicates valid bytes of data bus (during valid cycle it must be 0xf)
iwb_ACK_I	1	I	Acknowledgment input (normal transaction termination)
iwb_ERR_I	1	I	Error acknowledgment input (abnormal transaction termination)
iwb_RTY_I	1	I	In OR1200 treated same way as iwb_ERR_I.
iwb_WE_O	1	O	Write transaction when asserted high
iwb_STB_O	1	O	Indicates valid data transfer cycle

Memory System: As Harvard architecture, Instruction memory and Data memory are kept separated. Instruction Memory stores the application program for System configuration, System operation, Frequency Synthesis, and Output data delivering. Data Memory stores data samples for required waveforms: look-up tables containing sine, triangle, and saw tooth wave signal data; Temporary tables for processed data, and Buffer output tables for final processed data.

Possible implementations are explored: storing application software in on-chip ROM adds circuit area and restrain functionality to what's stored; loading application software to RAM using a boot-loader reduces memory area and allows additions to application software to be made and debugged in development board before being loaded into the chip.

A similar concept was developed for waveform data tables: data can be stored in on-chip ROM with fixed data width and samples per waveform cycle, this increases in-chip data memory space and reduces flexibility to change data size and samples per cycle. Otherwise sample data can be modified outside the chip, as well as the number of data samples per cycle, and then uploaded to RAM using the boot-loader. This shows the trade-off between fully customized designs versus a configurable programmable circuit design.

User Interface. A UART port has been selected for system configuration and operation. It was selected over a USB port since it takes less circuit area and speed is not important during configuration and operation setting. For extended interfacing capabilities, Wishbone data and instruction buses have been added, they can be used for external memory access and to connect this system to external peripheral systems, drivers or bridges.

User interface functionality has been summarized in Table 3.5 according to the system configuration and operation needs:

Table 3.5 *User interface: primary functions.*

Function	Description
Load	Load application software from available selections
Select	Define waveform to be used for next experiment
Configure	Select waveform, samples per cycle, output frequencies, operation mode, and exposure time for signal generation.
Operate	Start signal generation disabling other functionalities to maximize output frequency. Keep continuous operation until exposure time finishes or stop request is received.
Monitor	Displays feedback info from operation and data being sent to output port. This function can be disabled to eliminate execution time for monitoring and maximize output frequency.

Output Ports: Parallel ports have been used as output channels: a 32 bit (4x8 bit) GPIO port is used to deliver output data. Standard data is 8 bit wide, so up to four channels are available. Output data is delivered in one of this

forms depending on operation mode: in mode 1, 8-bit data samples containing one single frequency are delivered; in mode 2, 16-bit data samples containing two separate frequencies are delivered in two 8-bit channels, 8-bit each; in mode 3, 8-bit data samples containing two superimposed frequencies are delivered in one 8-bit channel.

3.3 Challenges for Variable Optimization

Here are shown the major challenges to be faced in the definition and design stages of the system; also is presented the decision to be analyzed and justified at each concept.

To obtain the maximum output frequency from a base operation frequency on the value you need. Variable optimization: processor selection, design and fabrication models availability.

1. To minimize execution code when in signal generation routine, so nominal output frequency is not reduced. Decision: Base data tables storage scheme, output memory buffer use, only one executing thread when running signal generation.

2. To define a data selection algorithm for waveform construction to reduce harmonic addition. Decision: Define an algorithm to select a set of data that minimizes gap between voltage steps.

3. To optimize code for minimum execution time on procedures like:

    - Select data from tables. Variable optimization: Equally time separated data or equally voltage separated data for harmonic reduction.

- Addressing memory and load data from memory. Decision: schemes for storing and addressing original waveform data, in cache or external memory.

- Buffer memory table construction. Variable optimization: separate, continuous, adjacent or superimposed memory segments when generating more than 1 signal, to reduce access times and maximize output frequency.

4. To keep power consumption low. Variable optimization: Consider the system as a whole or by operating mode, since most of the time the system will be in stand-by or configuring mode.

- Software related consumption. Application routines should be minimized on code size and execution time, in all operating modes: stand-by mode, configuring mode, pre-processing mode, and signal generation mode.

- Hardware related consumption. Determine power consumption for:

- Processor, in all operating modes.

- Memory, in read and write access.

- For every core include activity factor in power estimation.

3.4 System-on-Chip Specifications

The bus based architecture has been defined, and four implementation options have been explored, in order to compare performance and circuit size for each of those implementations. Since main impact in circuit area is due to instruction and data memory, this has been the parameter to be set first, keeping the configurable

and programmable capabilities in focus. Option 4 from Table 3.6 has been selected for the SoC implementation; for the development board implementation there were less memory restrictions and load-store-execute flow was defined by software development tools.

1. Parameters:

   - Instruction and Data Cache size: 2 K-bytes blocks, up to 8 K-bytes total.
   - Instruction on-chip RAM: 2 K-bytes blocks, up to 8 K-bytes total.
   - Instruction on-chip ROM: 256 bytes for boot-loader or 2 K-bytes blocks, up to 8 K-bytes total for on-chip application.

2. Application Software functionality.

   - Standard implementation on SoC. Program size < 4Kb. Data size < 1Kb. Output data: two output channels; Operation: resolution from coarse 8bit data. Output patterns: sine, triangle and saw tooth for single or superimposed frequencies.
   - Extended implementation on development board. Program size < 8Kb. Data size < 4Kb. Number of output channels is board dependent. Output patterns: sinusoidal, triangle, and saw tooth, for single or any mix of two superimposed waveforms of different frequencies. A sequence of user defined test with different time exposure.

3. Variables

   - Circuit Area.
   - Processor area: ALU, Registers, MMU, Exceptions, Control.

- On-chip memory area: Instruction and Data RAM, Instruction ROM.
- Total chip area: processor, memories, peripherals.

4. Performance

- Pre- processing times: time to store processed and buffer data tables.
- Execution times: for each functional module in application.
- Time between samples: maximum time between stores to GPIO port, minimum output frequency.

5. Hardware configurations. The way to load and store application software, along with the selected functionality for it, can fit into several architectures. Application software can be stored and modified on external flash or EEPROM memory, and loaded into in-chip RAM using a small boot-loader.

6. Possible implementations: select block size for Instruction cache memory, Data RAM, and ROM.

7. Operation Flow. Application software may be stored in on-chip ROM with no further modification or debugging capabilities after fabrication, or stored in external memory for debugging, modifications, and up-grade test in development board, to be loaded into chip during boot-load. A top-level operation flow is shown in Table 3.6 for four possible implementations.

Table 3.6 *Possible implementations, shown at top-level operation.*

	Application Software stored in in-chip ROM	**Application Software loaded from external Flash**
In-chip RAM size is smaller than application software size	Option 1 Go to program Start instruction. Load pre-processing program from in-chip ROM into I Cache.	Option 3 Start boot loader. Load external pre-processing program into in-chip RAM.

	Application Software stored in in-chip ROM	Application Software loaded from external Flash
In-chip RAM size is larger than application software size	Read operation parameters from interface. Load data from in-chip ROM into D Cache. Generate and store temporary and buffer data tables in D Cache. Load data generating program from in-chip ROM into I Cache. Store data in buffer tables from D Cache to output port.	Read operation parameters from interface. Load external data into in-chip RAM. Generate and store temporary and buffer data tables in D Cache. Load external data generating program into in-chip RAM. Generate and store temporary and buffer data tables in D Cache. Store data in buffer tables from D Cache to output port.
	Option 2	Option 4
	Go to program Start instruction. Load program from in-chip ROM into I Cache. Read operation parameters from interface. Load data from in-chip ROM into D Cache. Generate and store temporary and buffer data tables in D Cache. Store data in buffer tables from D Cache to output port.	Start boot loader. Load external program into in-chip RAM or I Cache. Read operation parameters from interface. Load external data into in-chip RAM or D-Cache. Generate and store temporary and buffer data tables in D Cache. Store data in buffer tables from D Cache to output port.

3.5 Signal Generation

A goal of this work is to achieve the maximum output frequency for a given architecture, processor speed, and memory access times. The key tasks for this achievement have been: a) data pre-processing, b) the signal generation scheme, and c) the memory access routine.

The equation for output frequency in the selected waveform is:

$$F_0 = \frac{1}{tbs * spc}$$

Where F_o is the output frequency, *tbs* is time between samples (the time between two consecutive data samples to be sent to the output port), and *spc* is

the number of samples per cycle (the number of data samples used to build a complete cycle of the selected waveform).

The value of tbs depends on the system clock cycle and the instructions that take for the signal generation routine to get a new data sample from the output table and send it to the output port:

$$tbs = (\#\ of\ \ instructions)(clock\ \ period)$$

For example, for a system clock of 100 MHz, using 12 samples per sine cycle, and a simple load-store routine of 6 instructions, the cycle period would be 1/100E+6 = 10 ns and the Output frequency:

$$F_0 = \frac{1}{(6)(10E-9)(12)} = 1.388MHz$$

As can be seen from (3), to get higher frequencies a tradeoff can be made by, e.g., using fewer samples per waveform cycle, eliminate check-for-stop during generation or change instruction counting for timer operation.

A precise count for clock cycles between output data samples, and therefore maximum output frequency, can be calculated after compiling the application software for the target processor, the OR1200, based on execution cycles per instruction type shown in Table 3.7.

Table 3.7 *Instruction Set Architecture execution times.*

Instruction type	Cycles to execute
Load	2, if cache hit
Store	1, if cache hit
Integer arithmetic	1
Multiply	3
Compare, logical	1
Rotate, Shift	1

Another way to look at output frequency is to calculate from processor speed and from cycles per instruction in the signal generation cycle:

$$T_{min} = \frac{Cycles\ between\ samples}{Pr\,ocessor\ speed}$$

For OR1200 running @ 200MHz:

$$T_{min} = \frac{10}{200\text{MHz}} = 0.05\mu\text{s}$$

If 10 data samples for the sinusoidal signal are desired, then:

$$T\,sin = (10)(0.05\mu\text{s}) = 0.5\mu\text{s}$$

$$F_0 = \frac{1}{T\,sin} = \frac{1}{0.4\mu s} = 2\ MHz$$

Which, according to the "Nyquist–Shannon sampling theorem", states that perfect reconstruction of a signal is possible when the sampling frequency is greater than twice the maximum frequency of the signal being sampled. In this case minimum execution time is 0.05 μs, so the system could generate, at most, 10 MHz signals. At this sampling rate an external filter will be needed to improve spectral purity.

3.5.1 Frequency Synthesis Methodology

Frequency synthesis has been historically achieved using several analog and digital approaches. The digital approach favors miniaturization and additional functionalities such as data storage and data processing. Actual devices which deliver output signals for a wide frequency range can be found already in wireless communications, but they are application specific and do not allow additional

functions. A variety of signals and patterns have been found to be useful in the mentioned range of applications, where researchers currently work with manual procedures using regular equipment as signal generators or oscillators. A more controlled experiment setting is desired so research results can speed up, and it can be achieved by having complex electric stimulation and varying signal parameters such as frequency, waveform, superimposed patterns, etc.

Besides the problem of delivering complex signal patterns, electric stimulation has to be implemented in a small size device because typical applications demand portable stimulation and test instruments, our approach integrates frequency synthesis, with multi-waveform generation, multi-waveform superposition, and operation configurability, so the implementation can be customized for multiple applications and processes. Besides, as a modular processor based solution is implemented, the methodology takes advantage of it by remaining generic and open to modifications, additions, and upgrades.

At the end, a small, low power implementation was achieved. Original digital frequency synthesis schemes calculate sine values on the fly (computation intensive scheme) or access pre-stored values from memory (memory intensive scheme). None of both schemes are useful by themselves in this system due to the goal of maximizing output frequency to clock frequency ratio, and to the memory and speed optimization tradeoffs between both schemes.

For performance optimization in this system the main intention of pre-processing the base sine data sample is to reduce computation during signal generation (i.e. while sending processed data to the output port); this way pre-processing times does not impact maximum output frequency. Computation intensive versus Memory intensive schemes are shown in Figure 3.6:

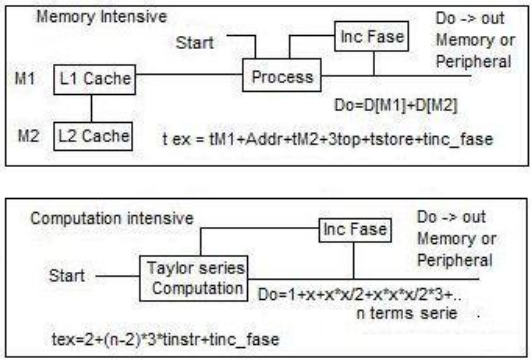


Figure 3.6 *DDFS: Memory intensive versus Computation intensive methodologies.*

3.5.2 Output Data

Available signal waveforms for board implementations are sine, triangle, and saw tooth. Preprocessing base data into temporary and into output buffer tables allows application software to do computation intensive tasks before operation and simple address-load-store operations during signal generation: Data input for pre-processing algorithm comes from base data samples tables. Selected samples are extracted by translating temporal spacing into memory spacing to achieve target signal frequency. Number of data samples per cycle is user defined.

When superimposed frequencies are desired pre-processing is executed twice with correspondent time-space parameters. Intermediate data is stored in separate temporary tables for each processed frequency.

Final processed data is stored in one output buffer table for simple access, low execution times.

Figure 3.7 shows data processing for one single frequency, and 9b for two single frequency outputs – the last step would be add for superposition and concatenate for separate frequencies-. See appendix A2 for the complete data

tables of base waveforms and an example of data processing for superimposing two frequencies.

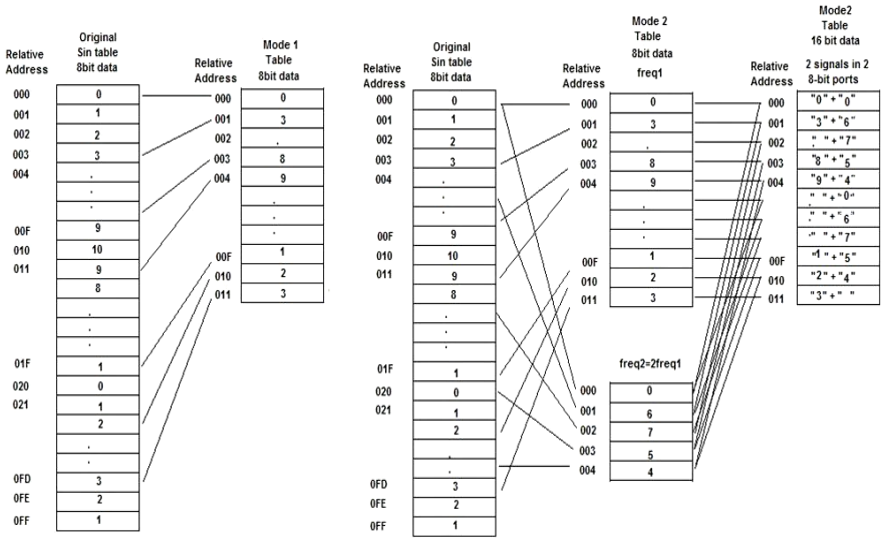


Figure 3.7 *a) Data processing for one single frequency. b) Data processing for two single frequency outputs. Last processing step is Add for frequency superposition or Concatenate for separate frequencies.*

3.5.3 Frequency Sweep and Superposition

Preprocessed data from temporary tables are superimposed by time matching of data samples from both separate frequencies. Data samples are added when time matches among samples and time holes due to frequency difference is filled with last data. Output data is stored into buffer tables. If there is no common factor between frequencies, buffer table size can grow indefinitely.

A set of sequenced tests can be programmed to be executed when characterization test need to go through a frequency range to identify particle's properties or behavior. These sequenced tests can sweep a desired frequency

range in user defined steps, for example, a 1 MHz signal is delivered for 30 seconds and a 10 MHz signal is delivered for 60 seconds after the first one, and so on. This frequency sweeps are useful when the particle's behavior is unknown or when running a characterization experiment.

3.5.4 Methodology Software Architecture

Application software is developed to perform four main tasks: Define and store waveform data samples, get configuration and operation parameters from user, process data to obtain selected output signals, and execute frequency synthesis to deliver processed output data samples.

Waveform data samples for a complete cycle are pre-calculated and stored as integer numbers in a 0-255 scale, being 0 the lowest peak value of the waveform (-V), 127 the mid-value (0), and 255 the highest peak value (+V). Values are stored in data RAM to reduce ROM needs. For the three stated waveforms 3 x 256 bytes of data space is needed. As these values are stored into RAM along with the program code uploading, new data tables with different waveforms can be included in the source files of the application and the software architecture remains unchanged.

Configuration and operation parameters are for user to select the type of waveform desired, the operation mode to be executed, and the time period to deliver the outputs.

When operation mode users select 1 out of 3: one output signal with one single frequency, two separate output signals with two different frequencies, or one output signal with two superimposed frequencies.

Data processing takes base waveform data to generate a temporary table containing selected data to form the desired frequency. Temporary tables for two different frequencies are generated for modes 2 and 3. Temporary tables in modes 2 and 3 are processed to form one output table containing two separate or superimposed frequencies. Values for temporary and output tables are stored in RAM during processing.

Output data delivering, to complete frequency synthesis process, takes data from processed output table and sends it to output port, using 8 bits for modes 1 and 3, or 16 bits for operation mode 2. Figure 3.8 shows application software flow:

Application functionality is achieved by independent tasks. Table 3.8 presents function description for detailing software execution.

Table 3.8 *Application Software, Function Description.*

Function	Description
Get operation parameters. Configure operation.	Gets operation parameters: frequency for output signals, samples per cycle, operation mode
Generate data samples for waveforms	Calculates and stores base sine data into memory table.
Calculate time and space between samples to construct desired frequency, mode 1.	Calculates time and space intervals to extract data from base table to construct one desired frequency.
Mode 1: generate output table for selected frequency	Process data from base table and store into temporary table.
Calculate time and space between samples for two frequencies, modes 2 and 3.	Calculates time and space intervals to extract data from base table to construct two different desired frequencies.
Modes 2 and 3: generate separate temporary tables for each frequency	Process data from base table and store into two separate temporary tables.
Mode 2: generate output table for selected concatenated frequencies	Process data from temporary table and store into one output table containing two frequencies in two separate signals.
Mode 3: generate output table for selected superimposed frequencies	Process data from base table and store into one output table containing two frequencies in one signal.
Send data from output table to output port	Take processed data from output table and store it into output port. 8 bits for modes 1 and 3, 16 bits for mode 2.

Function	Description
Format data to 8 bits in base waveform tables	Scale base sine data from -1 to 1 to 0-255 (8bits).
Continuous cycle of signal generation	Deliver output data continuously until stop requested or sequence time finished.
Non-multiple frequencies protection	Eliminates remaining cycle data for superimposed frequencies when no hyper-cycle is possible.
Variable size protection during data processing	Scale to 8 bits intermediate data resulting from operations to fit temporary and output tables.
Transmit output data for record and re-use	Transmit delivered data to serial port to be displayed or stored by user interface for visualization, record,, or future use.
Scale data, modes 1 and 3	Scale output data in modes 1 and 3 to 8 bits
Scale data, mode 2	Scale output data in mode 2 to 16 bits

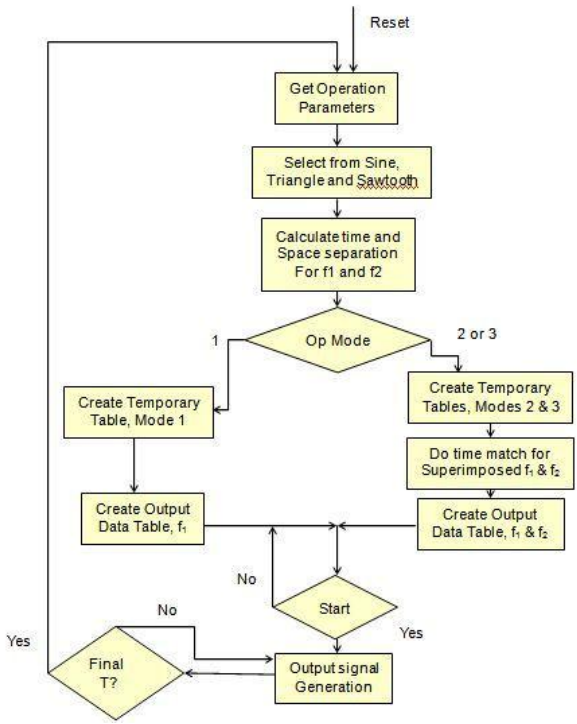


Figure 3.8 *Application Software Flow.*

3.6 ASIC Design Flow

In this section are presented a description of the design methodology, the CAD tools used in the chip design flow, and the procedures and results on the clusters involved in the design: synthesis, timing, memory blocks generation, place and route of the system cores, power consumption analysis, IO ring design, and integration at chip level.

3.6.1 Design Methodology

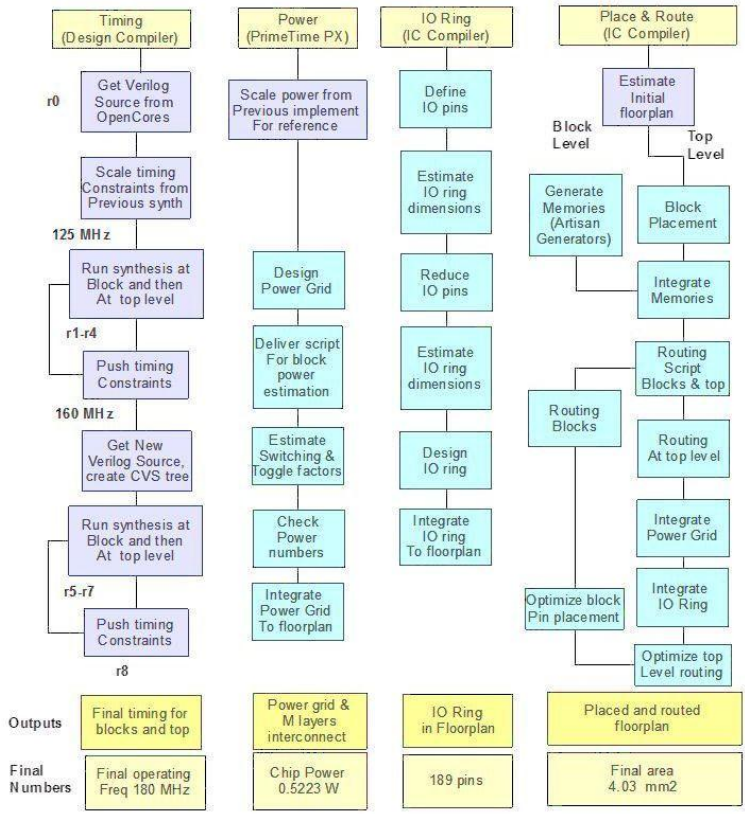


Figure 3.9 *Chip design flow on four areas: Timing, Power, IO pins, Area.*

Open source cores have been used to integrate a modular architecture over an open source bus. The design flow has been followed for optimal performance, minimum chip area and minimum power consumption, in that priority order when a compromise was needed. Work on all the areas relevant to physical design has been done: timing optimization, power estimation and grid design, IO pin selection and ring design, block placement and pin optimization for routing, and layout generation for minimal area.

3.6.2 CAD Tools

During the design flow, CAD tools have been used and the design process is not seamless due, mostly, to input-output file formats and compatibility between tools. As a reference, a tutorial on design flow was reviewed. Table 3.9 shows the tools used for each of the main design tasks. Tasks have been performed sequentially in the first design steps; in the later design steps they were performed in parallel and iteratively to work on the trade-offs of the design: chip area, power consumption, and system timing/performance. Figure 3.9 illustrates the design flow.

Table 3.9 *CAD Tools Used in design flow.*

Concept	Tasks	Tools
Timing	Synthesis process at block level and at processor level. Set initial time constraints at block level. Iteratively push timing constraints at block and chip level. Deliver block and global timing analysis.	Synopsys, Design Compiler (DC)
Place and Route	Initial area estimations at block level. Deliver an initial floor plan for routing. Placement and routing at block level. Global routing. Placement optimizations.	Synopsys, Design Compiler (DC) Synopsys, Integrated Circuit Compiler (ICC)
Memories	Generate logical and physical views for RAM and ROM blocks. Generate different aspect ratio implementations for placement optimization. Create memory wrappers for integration.	Artisan, Memory Generators Synopsys, Design Compiler (DC)
Power	Initial power estimations at block level. Iterative Power Grid Design.	Synopsys, Design Compiler Synopsys, PrimeTime PX

Concept	Tasks	Tools
IO Ring	Iterative IO Ring design.	Synopsys, Integrated Circuit Compiler (ICC)
Clock	Clock tree synthesis at chip level.	Synopsys, Integrated Circuit Compiler (ICC)
Integration	Integrate Power grid to floor plan. Integrate IO ring to floor plan. Integrate memory blocks at chip level. Integrate clock tree at chip level.	Synopsys, Integrated Circuit Compiler (ICC) JupiterXT, Synopsys
Verification	Functional verification at block and processor level	Mentor Graphics, ModelSim

3.6.3 Synthesis and Timing

During the synthesis process the local and global timing has been optimized and initial area and power estimations have been done. Eight rounds of synthesis have been performed until timing convergence and closure have been achieved. Synthesis rounds to get minimum slack time, area estimations, and power estimations have been done with Design Compiler from Synopsys. Selected processor cores from Open Cores have been used for bus compatibility and minimum edition of source codes. The available versions of these open source cores are not optimized for performance or area, so optimization in these areas have been done during this design process.

A simulation for functionality, at processor level, has been done before starting synthesis process, using ModelSim from Mentor Graphics.

The tasks performed and the obtained results from the rounds of synthesis were:

Round 0: Estimate initial timing constraints for each core based on gate count, estimated size and worst case data path.

1st to 4th first rounds: Run synthesis at block level pushing timing constraints; consider adjacent blocks in data path to push constraints for required arrival times and arrival times.

5th and 6th rounds: Run synthesis at top level. Identify critical paths. Push timing constraints on critical paths. Use new version of open core source repeat from round 5.

7th and final 8th rounds: Update CVS tree (source version control) to match last rounds. Round 8 was the final round for pushing the block and global timing constraints.

The global path delay at top level synthesis has been calculated by:

Global path delay= Arrival time + Pass-through + (cycle time – Required arrival time) + interconnect delay

Where the first three terms depend on both, block level synthesis and top level synthesis, and the last term depends on floor-plan and time of flight for metal 3 and metal 4.

Final results show that the minimum clock cycle is the requested 4 ns plus the worst slack time achieved of -0.978 ns, which leads to a minimum required clock cycle of 4.978 ns, equivalent to a processor frequency of 200.8 MHz; Figure 3.10 shows the occurrence distribution of the single and double cycle paths in the architecture, being the ones on the left the worst slack times observed.

Table 3.10 *Results from the rounds.*

Round	Slack	Action taken	Improvement
1	-7.9ns		
2	-6.4ns	Iterate timing constraints at block level	1.5ns
3	-4.5ns	Consider pin info in constraints	2.1ns
4	-4,3ns		
5v1	-2.3ns	Push synthesis effort	2ns
5v2	-1.2ns	Identify multi-cycle paths	1.5ns
5v3	-0.75ns		
6	-1.0ns	Source files change	-0.25ns

Round	Slack	Action taken	Improvement
7	-0.82ns	Source files change	0.18ns
8	-1.1ns		-0.28ns
Final	-0.978ns	Consider interconnect delay	0.12ns

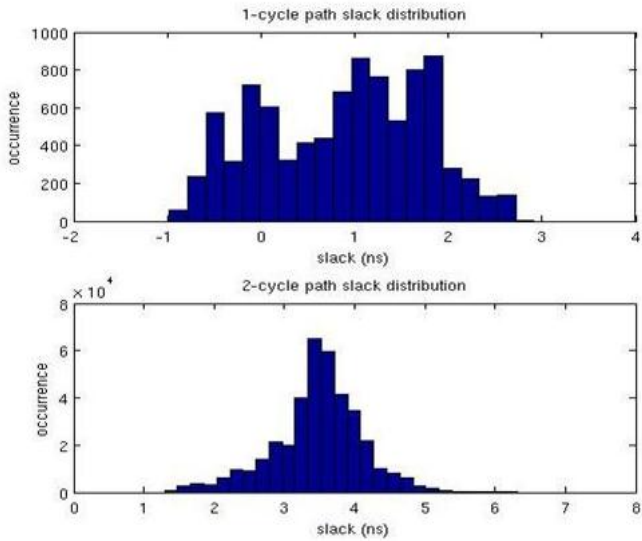


Figure 3.10 *Occurrence distributions of the single and double cycle paths in the architecture.*

Memory blocks.

Memories for this system (instruction cache, data cache, in-chip RAM and in-chip ROM) have been generated using a Memory Generator tool from Artisan which takes as input an abstract description of the memory blocks and produces several memory formats suitable for various tools and purposes. Using a memory generator instead of synthesizing a memory can optimize speed, for density and for power, can control the memory blocks aspect ratio for efficient floor planning, deliver timing and power models for integrate to other design tools, allow configurable word-write mask and redundancy options.

Memory blocks of 2 K bytes, 4 K bytes and 8 K bytes have been generated to facilitate block placement and routing in chip.

A set of views can be generated: PostScript data sheet, ASCII data table, GDSII layout file, LVS netlist, Synopsys model, PrimeTime models, TLF models, VCLEF footprint, Verilog model, and VHDL model. The Relative footprint shows how the aspect ratio of the memory changes as the words, bits, and Mux parameters are varied. The instance and the power ring are included in the footprint.

RAM architecture, timing specifications, and physical characteristics:

1. Synchronous Random Access Memory is triggered by the CLK rising edge.
2. Pins: CLK, CEN, WEN, OEN, A[m-1:0], D[n-1:0], output Q [n-1:0].
3. Memory blocks are cut in symmetrical sides to easy clock distribution and layout.
4. Dual port memories provide dual ports for all, input and output signals.
5. Power rings. Power rings can be generated around the SRAM, size them properly. Size depends on the chip-level power distribution, the number, width, and placement of supply wire connections to the power rings, and the current consumption. Recommendation: supply current evenly at the edge of the instance where the pins are located.
6. Top metal layer: metal1 to metal4 are used in the design and blocked for routing. Layers above m4 can be routed over the memory.

7. I/O pins are located along the bottom edge of the memory block on any of the metal layers, and they are large enough to accommodate a pre-determined on-grid width wire connection.

8. Verification: The views produced by the generator can be verified with standard tools.

ROM architecture, timing specifications, and physical characteristics:

1. Synchronous Read Only Memory is triggered by the CLK rising edge.

2. Pins: CLK, CEN, A[m-1:0], Q [n-1:0]. If CEN is high then memory is in standby mode and Q has last data, if CEN is low memory is in read mode and Q has data from address A.

3. Memory blocks include Row and column decoders, Clock generator, Memory array and Amplifiers/IO buffers for the outputs.

4. Power rings: multiple, evenly spaced connections have been used from core V_{dd} and V_{ss} to the rings around the instance on the side where the I/O pins are located.

5. I/O pins are located along the bottom edge of the memory block on any of the metal layers.

6. ROM code File. An Artisan format ROM code file must be provided for each generated instance.

   - Format: Code file contain only 0s and 1s.

   - The line number in the file is equivalent to (address-1).

- Each character of a line corresponds to the bits of a word. Character in column 1 of a line is the most significant bit.
- Address goes from m to 0, bits and columns go from n to 0.
- This file is needed for behavioral or physical views such as Verilog, VHDL, Tests can, Sunrise, GDSII and LVS Netlist.

Tool Verification. Views and files generated have been verified with Synopsys Design Compiler.

Before Place and Route, a FRAM view has been created for all memories in the design by importing the VCLEF and running the Blockage, Pin and Via (BPV) to create the FRAM.

3.6.4 Place and Route

P&R, the process of placing each individual block within the top level design, has a major impact in chip area: it must use the area optimized netlist for each block and use the interconnect area efficiently for routing.

Several iterations for P&R have been made due to changes in synthesis, pin placement and block aspect ratio impact placement and routing results. Integrated Circuit Compiler (ICC), from Synopsys, has been used to route signals within blocks, to place blocks within layout, to route signals between blocks, and to optimize block and pin placement for optimal routing.

A preliminary floor-plan has been delivered for power grid design. Block placement re-runs have been made with different aspect ratios for each block, until better area utilization is achieved.

About 30% of space is left between blocks for interconnect routing, clock tree and power grid.

3.6.5 Power Analysis

To estimate and analyze the power consumption for this system, a sequence of iterative tasks have been performed: estimate each block power using Design Compiler from Synopsys, include activity factors in power calculations, calculate Switching Power, Cell Internal Power and Cell Leakage power for each block, design a power grid using Prime Time PX, and integrate power grid to routed layout.

Power estimations for individual blocks are made initially from block size and gate count. Activity factors have been integrated using a code benchmark. Later, a preliminary floor-plan from ICC has been used as the base for power grid.

Vertical power lines go on M5, Horizontal on M6. No power ring has been added to ease integration with power in M3 and M4 and to power pins in IO ring. Power grid includes V_{dd}, clk, rst and V_{ss} lines, with spacing and widths: 5λ, 2λ, 2λ, 2λ, 2λ, 2λ, 5λ, 2λ respectively. Total spacing between two V_{dd} will be 22λ, which should be multiple of power grid spacing in M3 and M4 for interconnection.

Power estimation for each block has remained similar, regardless the changes in source code occurring during synthesis and integration. Power grid at M5/M6 considers block placement and individual block consumption, while power grid inside blocks on M3/M4 will be done automatically.

A verification run for obtaining activity factors by block from simulation instead that from estimations has also been made.

Dependencies for Power IR drop analysis, which is the voltage drop due to the resistance of interconnects in power network are illustrated:

- Additional code lines at top level are needed for power grid.
- IO ring needs to be hooked up to the top level design.
- All blocks should place their pins only in M3 and M4.
- The IO pads have to be hooked up to the power rail, to get info about external source of power rails.
- Modules should be power routed in M4 and hooked up to the power grid using via M4 and M5.
- De-coupling capacitor filler cells must be inserted in empty spaces: within individual modules and in the full chip level between modules.
- Design Rule Check should be executed after hooking up power routes.
- Design decision to make for each individual block: where to put the power pins for minimum route to power grid.

Results from final run of power estimation are shown in Table 3.11.

Table 3.11 *Power Consumption Values.*

	Block Activity *(Net Switching Power + Cell Internal Power) + Cell Leakage power					
	Block Activity	**Net Switching Power**	**Cell Internal Power**	**Cell Leakage Power**	**Total Power**	**Weighted value**
alu	0.4	4.8500E−03	6.1020E−03	1.0290E−07	1.1000E−02	4.3809E−03
cfgr	0.02	2.4420E−04	3.0160E−04	1.0400E−08	5.4580E−04	1.0926E−05
ctrl	0.4	1.4490E−03	3.0220E−03	7.0550E−08	4.4710E−03	1.7884E−03
dc_top	0.5	3.0410E−03	5.9600E−02	1.0080E−05	6.2600E−02	3.1330E−02
dmmu_top	0.5	1.2410E−03	9.6630E−03	6.0270E−06	1.0900E−02	5.4580E−03
du	0.01	1.7200E−02	1.9800E−02	6.5830E−07	3.7100E−02	3.7065E−04
except	0.4	3.1590E−03	1.1200E−02	1.9230E−07	1.4300E−02	5.7437E−03
freeze	0.4	2.5380E−05	6.8790E−05	1.9090E−09	9.4170E−05	3.7669E−05
genpc	0.5	3.0450E−03	5.7010E−03	1.2740E−07	8.7470E−03	4.3731E−03
gpio	0.5	7.4720E−04	2.3390E−03	1.9830E−07	3.0860E−03	1.5432E−03
ic_top	0.5	1.3880E−03	5.7500E−02	1.0040E−05	5.8900E−02	2.9454E−02
if	0.5	7.3330E−04	2.2270E−03	3.9420E−08	2.9600E−03	1.4801E−03
immu_top	0.5	2.3510E−03	1.0600E−02	6.0780E−06	1.3000E−02	6.4815E−03
iwb_biu	0.1	7.1960E−04	1.4030E−03	8.6690E−08	2.1220E−03	2.1234E−04
lsu	0.6	5.7150E−03	4.5360E−03	9.4160E−08	1.0300E−02	6.1506E−03
mult_mac	0.1	6.9460E−03	2.8100E−02	5.1140E−07	3.5000E−02	3.5051E−03
operandmux	0.4	2.2710E−03	2.6400E−03	6.4490E−08	4.9110E−03	1.9644E−03
pic	0.01	5.3930E−04	1.1050E−03	3.0240E−08	1.6450E−03	1.6473E−05
pm	0.01	2.3360E−04	4.0410E−04	9.5910E−09	6.3770E−04	6.3865E−06
rf	0.5	2.9220E−03	2.6000E−02	7.3280E−07	2.9000E−02	1.4461E−02
sb	0.2	5.6610E−04	7.5770E−04	1.5890E−08	1.3240E−03	2.6477E−04
sprs	0.02	5.5530E−03	5.2600E−03	1.3510E−07	1.0800E−02	2.1639E−04
tt	0.01	9.8770E−04	1.7000E−03	4.6980E−08	2.6880E−03	2.6923E−05
uart	0.5	4.8040E−04	2.8240E−03	2.8800E−07	3.3050E−03	1.6524E−03
wb_biu	0.1	5.0040E−04	9.5720E−04	4.9520E−08	1.4580E−03	1.4580E−04
wbmux	0.4	2.3900E−03	3.6520E−03	7.5520E−08	6.0420E−03	2.4168E−03
Total		6.9298E−02	2.6746E−01	3.5766E−05	3.3693E−01	1.2349E−01

The power consumed by electronic devices has been on a downward path for many years as a result of the hard work and creativity of talented engineers. Despite the obvious gains, the creation of lower power designs continues to be a major concern of modern engineering. There are two facets to this engineering problem. One is simply the desire to consume less power; to extend battery life and to make wall-powered devices cheaper to operate and ecologically friendlier. The other, perhaps less obvious problem, is that all power consumed must also be dissipated. Power dissipation has become more difficult as devices have become more complex yet smaller. Of course, the best way to help the dissipation problem is to consume less power in the first place. This course looks at the fundamentals of achieving the low power operation needed with nearly all of today's leading-edge chip designs.

3.6.6 IO Ring

The Input-Output pad ring on a chip acts as a communication link between the chip core and the outside world. IO pad ring is a collection of open metal areas usually located at the periphery of the chip. When a chip is being packaged a mechanical wire bonder connects the open metal surface of an IO pad with the corresponding package pin.

The circuit functions of an IO pad ring are listed below:

1. ESD protection – The IO pad ring has diode protection circuitry which protects the gates connected to the pads from any external electrostatic discharge.

2. Buffering the output signal – Usually, digital output pads have buffers to allow driving huge external world capacitances of the order of 30 pF.

3. Buffer the input signal – Digital input pads can have buffers to isolate the external input from the signals inside the core chip. A digital input pad can also have a noise tolerant functionality which removes any noise that might have coupled to the external input. A Schmitt trigger circuit is used to perform this function. The circuit generates a cleaner signal, mitigating any effect that noise might have on the circuit performance. A tradeoff of using Schmitt trigger circuits is the fact that they are power hungry and are slow.

4. Mixed voltage interface – An IO pad ring usually provides a mixed voltage interface. The external IO pads are usually running at higher voltage while the cores chip inside run at a smaller voltage, to minimize power. The IO pad ring contains Level shifter circuits that perform this function.

Due to the limitation in resolution of the mechanical wire-bonding tool, a minimum open metal surface area and a minimum pad pitch need has been maintained. Maintaining a minimum pad pitch resulted in the limitation of the number of input/output pins possible for a chip to the minimum presented ahead. Table 3.12 shows the final IO pins for the designed SoC.

Table 3.12 *SoC IO Pin List.*

Signal	From block	Signal	From block	Signal	From block
clk_i	RISC 250MHz	iwb_dat_i(31:0)	Wishbone	pm_ic_gate_o	Power
rst_i	RISC rst	iwb_cyc_o	Wishbone	pm_dmmu_gate_o	Power
clmode_i	RISC clock control	iwb_adr_o(31:0)	Wishbone	pm_immu_gate_o	Power
pic_ints_i(3:0)	PPIC interrupts	iwb_stb_o	Wishbone	pm_tt_gate_o	Power
iwb_clk_i	Wishbone	iwb_we_o	Wishbone	pm_wakeup_o	Power
iwb_rst_i	Wishbone	iwb_sel_o	Wishbone	pm_cpu_gate_o	Power
iwb_ack_i	Wishbone	GPIO(31:0)	GPIO	pm_1volt_o	Power
iwb_err_i	Wishbone	pm_cpustall_i	Power	pm_clk_sd_o (3:0)	Power
iwb_rty_i	Wishbone	pm_dc_gate_o	Power		

The design of the IO ring for this system has been made based on the needed pins for this specific application and has been integrated later to the complete and routed layout. These pads are available ready-made in TSMC's digital IO pad library. These pads have been piled together in a rectangle to create the pad ring. The open metal area surface of the pad has a 50um length. The pad length itself is 70um. To keep sufficient spacing between two metal area surfaces - where the wire-bonder would come to attach the bonding wires-, a spacer of 10um was inserted between each pad. This increased the pad pitch to 80um.

The input pads were chosen without Schmitt trigger functionality (PDIDGZ) because Schmitt trigger circuits are power hungry and slower. Only general purpose IO pads (PRU08SDGZ) had schmitt trigger circuitry inside them. They also had control enable based input/output configuration functionality. The output pads were chosen based on the current driving capability required from the pad. The average current was estimated for the pad assuming a 30pF load and a 25% rise time at 62.5MHz. Based on the average current calculation for some of the pads, the average current requirement was 4.8mA. An output pad (PDO16CGZ) with a current capability of 16mA was therefore chosen to safely meet the current requirements. 3.11 shows a close up to the physical IO ring.

To find out whether the area was pad limited or core chip limited, a very conservative area estimate was made by adding the block areas and multiplied it by double to account for wire routing overheads. The area assuming a square came out to be 0.583976mm^2 (0.7mm x 0.7mm) – which denoted severely pad limited die size.

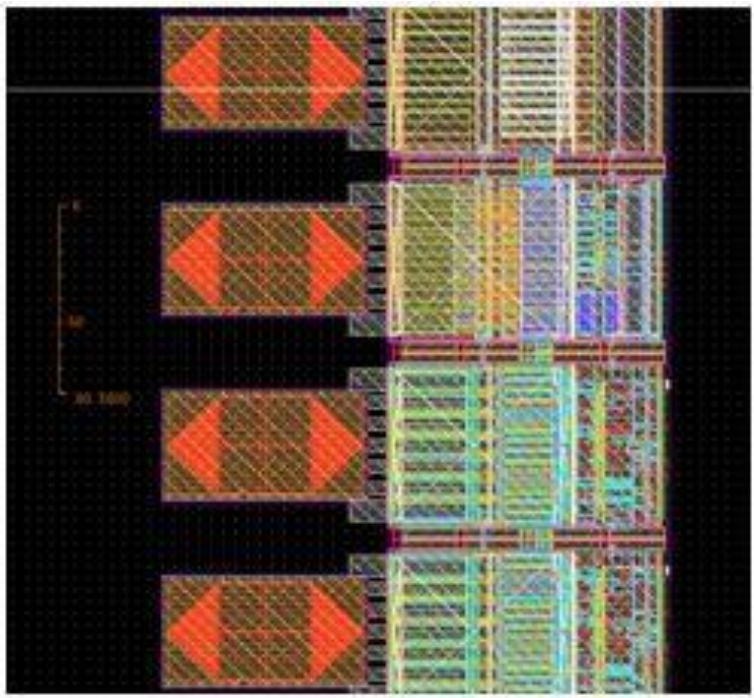


Figure 3.11 *Close-up view of the IO pad ring, pad pitch is 80nm.*

Number of V_{dd} - V_{ss} pair pin calculation:

There are two power supply voltages on the chip. The first is 3.3V volt which is used for input/output signals from the external world to the IO pads. The other voltage is 1.8V which is the V_{dd} for the core chip. The IO pad ring performs the task of converting the voltage levels.

An adequate amount of V_{dd} - V_{ss} pins is needed to allow sufficient current source and sink-in capability. The more the number of IO pads the larger the number of V_{dd} - V_{ss} pins. For this design a 3.3V V_{dd} - V_{ss} pair for every 8 IO pins was chosen. While two 1.8V V_{dd} - V_{ss} pairs were placed on each side of the die.

There are more 3.3V V_{dd} - V_{ss} pairs due to the amount of power dissipation of the IO pads needed to drive the external 30pF buffers. The core chip itself requires less power so fewer IO pins were dedicated to it.

3.6.7 Clock Tree Synthesis

Clock tree synthesis (CTS) is a separate design process which consists on building a balanced buffer tree from clock input pin to all clock sinks in the design blocks.

Clock design includes clock generation, clock regeneration and clock distribution. Tree design, leafs, sinks and location have been set according to this chip needs. The input files needed for the clock design are top level DEF file and top level Verilog net-lists.

To do clock tree synthesis SOC Encounter from Cadence has been used. When starting from a placed net-list, the flow is to perform CTS, do global routing and block level routing.

Clock Tree design is critical for system synchronization: if clock does not arrive on time to each block depending on its location within the data-path, all instruction flow gets wrong.

Clock distribution and Clock pin placement are design placement dependent, so every new place and route run requires a new clock tree design.

Figure 3.12 shows a sample run of the clock tree generation:

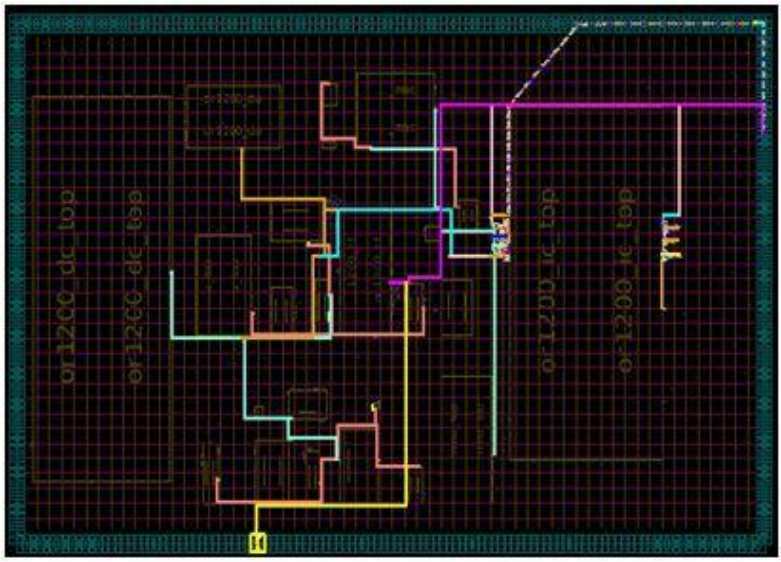


Figure 3.12 *Sample run of the clock tree generation.*

Hardware design in high-performance applications such as communications, wireless infrastructure, servers, broadcast video, and test and measurement equipment is becoming increasingly complex as systems integrate more functionality and require ever-increasing levels of performance. This trend extends to the board-level clock tree that provides reference timing for the system. A "one size fits all" strategy does not apply when it comes to clock tree design. Optimizing the clock tree to meet both performance and cost requirements depends on a number of factors, including the system architecture, integrated circuit (IC) timing requirements (frequencies, signal formats, etc.) and the jitter requirements of the end application.

3.6.8 Integration

Integration work is an inter-dependent task, since results from power, timing, placement, and routing affect other results, so iterations have been done until

satisfactory results in all areas have been reached. First designs for power grid and clock tree have been done based on the first preliminary floor-plan delivered. As part of integration, different versions of definers' file, one from verification, one from Timing and one from Place & Route have been merged into one common file to check and eliminate inconsistencies.

When cores have not been completely compatible with the bus based architecture, wrappers have been needed, since cores are generic open source code and customization is needed, especially for hardware integration and in-chip communication. A Wishbone compliant wrapper has been added for: an 8K SRAM for the memory module, a 4x8 bit GPIO core, and a standard UART. When synthesizing at top level some signal and bus inconsistencies arose: missing pins, incompatible bus widths, and unreferenced instantiations, among others.

Most Verilog sources have gone under editing and current control version is maintained for code consistency since it is critical for integration.

When integrating at chip level, hardware hierarchy has been redefined: block level is the open source for each individual block; CPU level is the unit build by interconnecting the individual blocks; OR1200 level is the processor built with CPU, memories, debug, and system units; SoC level is the system built with OR1200 processor, SRMA, ROM, GPIO, UART, power grid, and clock tree.

Tables 3.13 and 3.14 show details of the tasks performed and results obtained during the last two rounds of integration.

Table 3.13 *Tasks and results, 7th round of synthesis and integration.*

Cluster	Tasks/Problems/Results
P&R	Preliminary floor-plan has been delivered for power grid design. To be final it needs: resize IMMU cache, consider 30% of interleaving space for routing, clock and power lines. Do block placement re-runs with different aspect ratio for each block, until better area utilization is achieved.
Timing	Worst slack time at -800ps over a 4ns period. That leads to a frequency of 208 MHz.
Top level synthesis	Synthesis script and netlist for OR1200 top level: ready. Synthesized OR1200_top.v, net-list generated. To do: re-synthesize to be consistent with updated defines file, due to inconsistencies in source files.
Integration	Different versions, one from verification, one from Timing and one from Place & Route of definers' file should merge into one common file. Example inconsistencies: "OR1200_ARTISAN_SSP" has been commented out, `define OR1200_ASIC is commented out in one version, FPU related macros to be removed.
Verification	Verification is complete with the new Verilog files, re-do when definers' file is common to all clusters. Use SAIF from Synopsys (1st option, for tool compatibility) or VCD from Model-Sim to write out activity factor, for a given timing window of simulation. SAIF: forward_saif file required for Model-Sim to generate a backward_saif, the required output file
Verification for Power	VCD: To write out VCD: Read DC synthesized net-list into Model-Sim. Write out a VCD from Model-Sim from the designed test bench using time window which power numbers are going to be generated; zip VCD: "zip -r DESIGN.vcd.zip DESIGN.vcd" To convert between formats: Invoke Synopsys DC and use "vcd2saif" command
Source	Wrappers are written for SRMA, GPIO, and UART. These blocks, along with OR1200, will build the SoC top level. UART and GPIO connect to the data wishbone of the processor. New pins created for GPIO wrapper: *aux_i, ext_pad_i, ext_pad_o, ext_padoe_o, clk_pad_i,* For UART: i*nt_o, stx_pad_o, srx_pad_i, rts_pad_o, cts_pad_i, dtr_pad_o, dsr_pad_i, ri_pad_i, dcd_pad_i* SRAM: Wishbone compliant wrapped 8K SRAM memory module. No syntax errors. Needs functional verification, run design compiler to get the gate level Verilog.

Table 3.14 *Tasks and results, 8th round of synthesis and integration.*

Cluster	Tasks/Problems/Results
P&R	Re-run the global P&R scripts, since the floor-plan has changed significantly. All of the objects are placed. Rebuild each using synopsis/icc/2010 and make the OR1200_top library.
Power	As Verilog source files change, new SDC files for the new net-lists are needed for each block.
	After generating sdc files for each block, check for consistency among the gate.v net-list and new sdc file. Upload new files in the corresponding folder because existing scripts take them from there. Results from last run of power analysis are shown in a table below.
	Needs from Global P&R for Power IR drop analysis - Due to the resistance of the interconnects in power network, there is a voltage drop -:
	Add code lines for power grid
	IOs still not hooked up in the milky way
	Check all blocks place pins in M3 and M4.
	Hook up the IO pads to the power rail, to get info about external source of power rails
	Modules should be power routed in M4 and hooked up to the power grid using via 4-5
	Insert de-coupling capacitor filler cells in empty spaces: within individual modules and in the full chip level between modules
	Design Rule Check should be executed after hooking up power routes
Timing	Final run: squeeze out as much extra timing as possible. Reference to: For each pass-through in a critical path, a log has been used to show the average path slack, average time for the pass-through, and the number of times that pass-through appears in a critical path. Locate the pass-through routes in each block that occupy a good amount of time in a critical path to tighten while also locating a path with slack to give that you can relax. Due to changes in defines file between r6 and r7, the timing became a little worse, now at 200MHz. Some pins of debug unit and control unit are causing the problem. These pins pop up only in this release, probably due to changing of the define file.
Integration	Design decision to make: where and how to put the power grid on blocks?
Source	Defines file at processor top level was modified to meet requirements from: cache memory blocks (`define OR1200_ASIC and `define OR1200_ARTISAN_SSP 0), register file block (Type of register file RAM: `define OR1200_RFRAM_GENERIC), wishbone bus (OR1200_CLKDIV_4_SUPPORTED, This will allow us to use 50 MHz for the external wishbone bus.), Power management unit (`define OR1200_PM_IMPLEMENTED), and eliminated references to floating process unit since it is not implemented.

3.7 Design Evaluation

This system design was completed through the stages of the design. The hardware architecture was synthesized with Synopsys Design Compiler using TSMC Physical Libraries for 180μm technology. RAM and ROM arrays were

built with Artisan Memory Generators. Place and Route, Clock Tree synthesis and IO Ring design were realized with Synopsys IC Compiler. Power analysis and power grid design was made with Prime Time-Px.

Source cores -Processor, Peripherals, and on-chip Communication Bus- are synthesizable cores from OpenCores.org.

Here are presented the resulting design parameters:

Timing Analysis was made during synthesis process. Timing parameters like system clock frequency, bus clock frequency, and port clock frequency are implementation dependent only. Output update rate considers n data samples per waveform cycle. Table 3.15 shows operating frequencies per block group.

Table 3.15 *Operating Frequencies.*

Concept	Operating Frequency
System Clock	190 MHz
Wishbone bus clock	47.5 MHz
GPIO, 8 bit data L&S	5.94 MHz
Output data update rate	(48/n) MHz

Execution times were obtained from simulations of the application software and from actual execution times on a development board –the LM3S6965-. Application functions are grouped to present their execution times: Parameter set up function gets the operation parameters from user, and generates related data; Data processing function creates temporary and output table generation processing two waveforms -frequencies f_1 and f_2 -, 32 samples each.

An example where $f_2 = 4f_1$ is used. Output update function refers to signal generation - sending waveform data samples to output port-, and corresponds to 1 cycle for addressing, 1 cycle for load from memory, and two cycles for store

in parallel port. Execution times marked with * do not impact maximum output frequency since they are executed during system configuration or data processing, i.e. before signal generation begins.

Table 3.16 *Execution Times.*

Function	Clock cycles	Exec time (ns)
Parameters setup, 8 parameters, 4 cycles each	4 x 8	168
Data processing for temporary tables	2 (4 x 32)	1347
Data processing for output table	$5(32)(f_2/f_1)$	3368
Output update	4	21

Delivered Signals The system can deliver any mix of sine, triangle and saw-tooth waveforms with different frequencies in single or superimposed patterns.

Power analysis Power analysis was made for individual blocks then grouped for simplicity. Power values in Table 3.17 consider full throttle operation for that group of blocks and % of total power is calculated considering activity factor.

Table 3.17 *Power Consumption By Block Group.*

Block	Power, Watts	% of total power
Processor	0.176965	52.3
Memories	0.145400	42.97
Peripherals	0.006385	1.887
Bus	0.009620	2.843

Table 3.18 *Area Use By Block Group.*

Block	Area, (mm^2)	% of chip area
Processor	1.4866	25.73
8K IC RAM	0.9824	17.01
8K DC RAM	0.9736	16.85
256 bytes ROM	0.0212	0.37
Peripherals	0.0522	0.90

Bus & Interconnect	1.4694	25.44
IO Ring	0.7912	13.70

Areas The minimum total and interconnect areas were achieved by varying the aspect ratio of major blocks: memory and processor cores. Area use per block group is shown in Table 3.18.

3.8 Application Software

Based on the hardware specifications, the application software is developed including start up and load procedure, sine/triangle/saw tooth waveform's data storage, configuration and operation setting, and frequency synthesis process using the novel methodology created for this application.

Separate application software versions for development board and IC are kept due to differences in hardware resources.

Board version is an extended functionality version where hardware resources are only limited by the board features.

The SoC version is called the standard version: it has less functionality than the board's due to area budget, processor, and open source restrictions.

3.8.1 Program Flow

Table 3.19 shows a simplification of the application flow by listing only the main tasks; secondary tasks as data protection, data scaling and frequency synthesis details are not shown.

Table 3.19 *Application Program Steps.*

1. Configure General Purpose I/O port
2. Get configuration and operation parameters
3. Calculate time separation between data and shift factor for data extraction
4. Prepare temporary and output tables for requested mode
4.1 Mode 1: 1 8-bit table, single data, for 1 frequency signal output
4.2 Mode 2: 1 16-bit table, inter lapped double data, for two frequency signal outputs
4.3 Mode 3: 1 8-bit table, superimposed data, for two frequencies on 1 signal output
5. Configure four 8-bit GPIO port for output signals.
6. Send data samples for sine signal to output port, with a load-from-table/store-to-port cycle
7. Update output port continuously while waiting for stop signal
8. Restart operation to get new configuration parameters

3.8.2 Standard Version

A standard version of the application program has been developed to run on OR1200 based architecture; this version is the foundation for the SoC design and implementation. The three waveforms can be delivered to output channels and dual superimposed frequencies are included if frequencies are exact multiples, due to in-chip memory limitations. To be interrupted during operation using master reset only. Maximum output frequency is limited by the timing constraints of the available physical libraries for implementation and the timing optimization of the set of processor source files. Standard version takes less than 1 Kbytes of data RAM and less than 2 Kbytes of instruction RAM.

Table 3.20 shows a description of the routines in the standard version.

Table 3.20 *Routine List and Description, Standard Version.*

Routine	Description
Calculate Data Separation	Desired output frequency and number of voltage steps determine how many data points will be extracted from the original sine table in order to construct desired output signal. Two data separation parameters are needed for operation modes 2 and 3.
Calculate base time	According to desired frequency and number of voltage steps, there is a base times that indicates the time between data points are sent to output port.

Routine	Description
Create Table-Mode1	Extract data from original sine table needed to construct 1 output signal, 1 single frequency: each 8-bit data from original sine table is stored as the 8 least significant bits of the 32-bit output port.
Create Table-Mode2	Extract data from original sine table needed to construct two output signals, two separated frequencies: if two 8-bit output ports can be stored at the same time with 32-bit data, two data points from original sine tables must be concatenated before stored in buffer memory table.
Create Table-Mode3	Extract data from original sine table needed to construct 1 output signal, two superimposed frequencies: data points for different frequencies should be added to achieve superimposition.
Output Signal Generation	Continuous, uninterrupted loop, for loading data from buffer memory and storing it on output ports. No memory other than buffer is read, no instructions other than those for signal generation are executed.
Start/Stop external interrupt	Start/Stop button is enabled as an external interrupt in two execution moments: at startup to be ready for accepting configuration and operation parameters, and during Output Signal Generation routine to stop signal.
Timer interrupt generation	Within Output Signal Generation routine: When low frequency output is desired, a timer is used to update data to outputs at a base time determined by Calculate Base routine. For high frequencies time is achieved by cycle and instruction count.

See Appendix A1 for Application software C code, standard version.

3.8.3 Extended Version

The extended version of the application program was developed to run on an ARM9 or Cortex-M3 based development board. A functional implementation of this version is presented in chapter 5. Extended functionality is added, such as delivering data via an USB port for further analysis of monitored or stored data, mixing different waveforms in a superimposed signal for more controlled experimental environments and an interactive user interface for configuration and operation. Data tables for the three signal wave forms (sine, triangle and saw tooth) can be displayed at start-up for demonstration purposes of the novel frequency synthesis methodology to show the frequency superposition effect. Extended version takes less than 4 Kbytes of data RAM - 1 Kbytes for base data and 3 Kbytes for temporary and final data-, and less than 3 Kbytes of instruction

RAM. Although the Extended version has additional functionality it fits in the original SoC design which has separated RAM blocks of 8KB of data RAM and 8KB of instruction RAM.

See Appendix A2 for Application software in C code, extended version.

Chapter 4

The Open Source Design Tools

4.1 Chip Design Flow

As long as the technology and processes involved in circuit design remain within the current boundaries, the design variables will remain the same: Circuit area, execution speed, and power consumption. If a new and completely different technology and materials arrive, this could change. Meantime, the design teams will dependently develop their areas, iteratively delivering new versions of them until the team achieves a final functional design within the time-to-Market frame.

The trade-offs. The design team keeps in mind that when you modify the area of a circuit it also impacts the power consumption and the execution time. Although it is not possible to gain in those three variables: you gain in one, then loose in the other two. That is why in these projects the design team needs to focus on the final goal, rather than its own specifications, and still, optimize the individual parameters to the maximum achievable.

Time to market. The circuit design field is a fast moving one, with new circuits being released every day. Professional design teams always work on a very tight time schedule, in order to complete their design, containing what will be a novelty in the market, only if it arrives on time to it. At this point is important to note that maximum optimization always goes as far as time schedule allows, it means that you will see that if you would have more time to work on your design it can be smaller, or faster, or may consume less energy. But there is no more time for that, or the market will no longer find your design useful. So you need to declare your design done, in order to get into the market on time. Of course this applies when you are designing for a market, but if you design as a hobby or as part of an open cores community, you can have more time.

4.2 Open Source Design Tools

Using open cores in designing independent integrated circuits is a growing trend between electronic engineers, and there are large communities focusing on open source development, intended for electronic hardware. Designing hardware cores by programming them starting from scratch, is not easy, you need to know about electronics, programming, and design tools. For instance, many steps need to be done to ensure a design can be synthesized and translated to an FPGA or a Silicon wafer, through several verification steps. Knowledge on FPGAs and standard libraries is needed, and you need to be good at HDL programming.

4.3 Open Source EDA Tools

There are plenty of good EDA tools that are available as open source. The use of such tools makes it easier for you to take advantage of the resources and open cores available in related sites and forums. The larger and most used site for this purpose is Opencores.org. You can access there IP cores and scripts for an open source HDL simulator.

Here is a description of the most used terms and tools you will need to know. Of course, as any practical tool, there is no other way to be a master than using it and practicing.

Icarus Verilog Simulator. Icarus Verilog is a Verilog simulation and synthesis tool. It works like a compiler: when you compile source code written in Verilog, you can deliver different formats. For synthesis (the process of generating a circuit design from a description language), the compiler generates net lists. These,

and other compilers, elaborate design descriptions according to IEEE standards. You can surf the internet to download the Icarus Verilog simulator.

Verilator. Verilator is a free Verilog HDL simulator. It compiles synthesizable Verilog into an executable format and wraps it into a SystemC model. Keep in mind that the resulting circuit after compilation greatly depends on how you programmed it, so, the execution speed of the resulting model can widely vary. Since Verilator has been used to simulate many very large multi-million gate designs with thousands of independent modules, it is often chose as part of several verification environments. You can also surf the web to find the site for Verilator.

GUI-based design tools. For those not used to code by lines, there are GUI tools (Graphic User Interface). Of course, the more easy and graphic the tool is, the less control you have on the final representation of your design. A sample of a GIU design tool is Fizzim, but there are several more. The advantages of using a GUI tool are that they run in Windows or Apple, or anything with Java.

4.4 Open Cores Library

There are several internet sites where you can find circuit cores developed by experienced designers. One of the most popular sites is www.OpenCores.org, where you can find from the simple circuits, as adders and multipliers, to complex designs as processors and memories. Even more, you can find complete systems of hardware IP cores that you can download and use as open source. You can find them in several Hardware description languages, such as VHDL, Verilog, Verilog, SystemC, Bluespec, and C/C++. The developing stage or status of each core is indicated, so you know how trust worthy or reliable a core is; the stages or

versions of these projects can be Planning, Mature, Alpha, Beta and Stable. Within the open source community, there are different licenses that apply to each product; the licenses you will find are GPL, LGPL, BSD, among others.

A list of example core and projects is presented here, but it is very dynamic so you can check recent cores on the website. The cores are grouped by purpose, for instance: Arithmetic, Processors, Memories, Systems-on-a-Chip, and so on.

a) Arithmetic cores:

- Anti- Logarithm (square-root), base-2, single-cycle
- Discrete Cosine Transform core
- Elliptic Curve Group
- Floating Point Adder and Multiplier
- Gaussian Noise Generator
- Random number generator
- Maximum/Minimum binary tree finder
- Signed integer divider
- Sine and Cosine Table
- Trigonometric functions (degrees) in double fpu

b) Processors:

- ARM-compatible cores
- R6502 Processor
- Educational 16-bit MIPS Processor
- FORTH processor with Java compiler

- HIVE- 32 bit, 8 thread, 4 register
- Leros: A Tiny Microcontroller for FPGAs
- 8051 compatible CPU
- MCPU- A minimal CPU
- Wishbone High Performance Z80
- ZPU- the world's smallest 32 bit CPU with GCC toolchain

c) Memory cores:

- 8/16/32 bit SDRAM Controller
- Functional simulation models for commercially available RAMs
- High Performance Dynamic Memory Controller
- High Speed SDRAM Controller with Adaptive Bank Management and Command Pipeline
- Parametrized FIFO based on SRL16E
- Single Port ASRAM
- Synchronous reset fifo memory
- Wishbone Flash Interface for Parallel FLASH

d) Communication controllers:

- 10, 100, 1000 Mbps Ethernet MAC
- 8b10b Encoder/Decoder
- A VHDL CAN Protocol Controller
- Ethernet MAC 10/100 Mbps

- Ethernet Switch on Configurable Logic
- USB Device Core
- Wishbone SD Card Controller
- Space Wire Light
- Serial Port Interface Flash controller

e) System-on-Chip:

- Embedded FPGA Core
- Arm core
- RFID Transponder
- I2C Controller
- Real-time image processing unit
- OR1200 SoC
- Opens ARC-based SoC
- Soft Multiprocessor on FPGA
- MP3 Decoder
- NoC Network on chip

f) Other cores:

- 16x2 LCD controller
- 8254 Timer
- Alternative Oscilloscope
- Adjustable Frequency Divider

- Keypad scanner
- DDS Signal generator
- Date time
- FM Receiver
- General purpose IO
- Sound Encoder
- PWM controller
- Multiple Switch Debouncer
- OpenRisc 1200 Graphic Configuration Tool
- Programmable Interval Timer

Licenses. Although the open source code is free software, there are differences among the different license agreements that you accept when downloading and using it. Here is a brief explanation of several licenses, and you should check extensively the kind of license you are agreeing to. According to the Open Source Initiative, an Open source license "shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale. The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is

not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed".

LGPL. The GNU Lesser General Public License (LGPL) is a free software license published by the Free Software Foundation (FSF) that allows developers and companies to use and integrate LGPL software into their own software without being required by the terms of a strong license to release the source code of their own software-parts. For proprietary software, LGPL-parts are in the form of a shared library so that there is a clear separation between the proprietary and LGPL parts. The LGPL is primarily used for software libraries.

GPL. The GNU General Public License is a free license mostly used for software and it is intended to guarantee your freedom to share and change all versions of a program, to make sure it remains free software for all its users. So, you need to check carefully the version you are using, since it can come from a developer that has made a lot of changes to the original version.

BSD. The BSD license is a simple and liberal license for software. The restrictions to users are that if they redistribute such software in any form, with or without modifying it, they must include the original copyright notice, a list of restrictions, and a disclaimer of liability. The restrictions are: one should not claim that they wrote the software and should not sue the developer if the software does not function as expected.

Chapter 5

Sample Implementation

A functional implementation has been developed as a prototype of the stimulation system by integrating a user interface, the application software running in a processor based board, and a signal conditioning circuitry that delivers to the fluidic device signals, patterns and sequences selected by the user. The user interface allows the selection of signal and operation parameters, the based board system runs the extended version of the application software and shows the functionality of the multi-frequency synthesis methodology, and the conditioning circuitry allows the system to deliver analog voltages in a range that is needed in the majority of AC based electro-kinetics in micro fluidic devices. This prototype implementation include all the configurable parameters for a flexible setting that meets the functional requirements described in the standard extended versions, and is also a portable prototype that can be easily moved to different places or labs.

This chapter details the functionality of the extended version of the application software, defines an experiment to be performed with this prototype, show simulation results for a specific type of particles being manipulated, describe and illustrate the experimental environment and, most important, present the potential of this system in referenced research works about experiments and devices where this stimulation system could be used, as a stand-alone stimulation module or as a block to be integrated at chip level.

5.1 The Running Application Program

The extended version of the application program, developed to run on an arm9 or Cortex-M3 based development board –the LM3S6965-. Extended functionality is added, such as delivering data via an USB port for further analysis of monitored or stored data, mixing different waveforms in a

superimposed signal for more controlled experimental environments and an interactive user interface for configuration and operation.

Data tables for the three signal wave forms (sine, triangle and saw tooth) are displayed at start-up for demonstration purposes of the novel frequency synthesis methodology to show the frequency superposition effect. A requirement for lower frequencies was fulfilled by adding a new routine for frequencies smaller than 400 Hz, where a counter creates wait cycles so the time between samples in the output port is extended. A cyclic delivering of same or different signal patterns are delivered for a specific and individual period of time, is also available in the extended version: multiple tests can be done with sequenced stimulation patterns where each pattern may have different parameters such as frequency, samples per cycle, and exposure time. Additionally, when a particular stimulus pattern is found useful, it can be stored and re-used later so the experiment is repeated without having to set the operation parameters.

Table 5.1 presents a routine list and description of the extended version. The appendix A2 presents a documented version of the program code.

Table 5.1 *Routine List and Description, Extended Version.*

Routine	Description
Store waveform data tables	Stores 256 8-bit values for each of the three waveforms: Sine, Triangle and Saw tooth. See Appendixes A for table content.
Get operation parameters	Get operations parameters via the user interface. Displays parameter list and gets input values.
Validate input values	Check for frequency multiplicity in operation mode 2, check for 2-multiple number of data samples, check for frequency out of range.
Check frequency range	Separate operation into low and frequencies at 400 Hz. Low frequencies use up to 256 data samples per waveform cycle; high frequencies use up to 32.
Calculate Data Separation	Desired output frequency and number of voltage steps determine how many data points will be extracted from the original sine table in order to construct desired output signal. Two data separation parameters are needed for operation modes 2 and 3.

Routine	Description
Calculate base time	According to desired frequency and number of voltage steps, there is a base times that indicates the time between data points are sent to output port.
Create Table-Mode1	Extract data from original sine table needed to construct 1 output signal, 1 single frequency: each 8-bit data from original sine table is stored as the 8 least significant bits of the 32-bit output port.
Create Table-Mode2	Extract data from original sine table needed to construct two output signals, two separated frequencies: if two 8-bit output ports can be stored at the same time with 32-bit data, two data points from original sine tables must be concatenated before stored in buffer memory table.
Create Table-Mode3	Extract data from original sine table needed to construct 1 output signal, two superimposed frequencies: data points for different frequencies should be added to achieve superimposition.
Time match	When in superposition mode, does a time match between data samples for frequencies f_1 and f_2, since period and time between samples are different. See appendix A.
Clock set	Set clock for system and parallel port control from board main oscillator.
Configure port	Enable and configure parallel port A as output, as 8-bit set, output driving current.
Off-line monitor	When in off-line mode, data in final output table is displayed to monitor frequency superposition methodology.
Output Signal Generation, High frequencies.	Continuous, uninterrupted loop, for loading data from buffer memory and storing it on output ports. No memory other than buffer is read, no instructions other than those for signal generation are executed.
Output Signal Generation, Low frequencies.	Loading data from buffer memory and storing it on output ports uses a counter to create wait cycles and extend the time between samples. In this mode of slow output frequencies, the restriction of no computation during synthesis is not necessary.
Generate multiple sequences	When selected on user interface, cyclic delivering of same or different signal patterns is delivered for a specific and individual period of time. Multiple tests can be done with sequenced stimulation patterns.
Start/Stop external interrupt	Start/Stop button is enabled as an external interrupt in two execution moments: at startup to be ready for accepting configuration and operation parameters, and during Output Signal Generation routine to stop signal.

5.2 Experiment Definition

A Carbon-DEP fluidic device is used for these experiments. The fluidic device has 3-dimensional carbon electrodes above a comb-like planar array of electrodes in a chess board arrange. This device was fabricated by pyrolysis of

SU-8 structures defined by a two-step photolithography process following standard C-MEMS techniques.

The electrodes are used to apply an electric potential to the micro-channel in order to produce a non-uniform electric field distribution that will generate DEP traps. The 3-Dimensional carbon structures are 40 μm high with a 12.5 μm radius and a center to center separation of 45 μm and 100 μm in the X and Y axis, respectively.

Deionized water with K_2HPO_4 as buffer solution with a final conductivity of 21 μS/cm was employed. Conductivity was measured with a multi-parameter bench meter, Model HI 255 from Hanna Instruments.

Fluid sample preparation. *Saccharomyces cerevisiae*, 24858 yeast cells from ATCC - a global nonprofit bio-resource center and research organization that provides biological products, technical services and educational programs to industries and labs- were growth in Yeast Malt Broth at 30 °C for 18 hours until late log phase. Cells were then centrifuged and re-suspended with deionized water to remove the excess of culture media within the cells to a final concentration of $6x10^7$ cells/mL. Cells were labeled with Syto® 9 fluorescent (490/520) green stain. For the non-viable yeast cells, a sample of cells from the culture media is centrifuged and washed with deionized water, later to be heated up to 80 °C for 20 minutes. Non-viable cells are then labeled with propidium iodide fluorescent (490/635) color red. Carboxylated fluorescent polystyrene particles with a diameter of 10.14 and Dragon green color (480/520) were employed in this work. Particles were prepared in the buffer solution to a concentration of 2x106 spheres/mL.

Two mixtures, the first containing viable and non-viable yeast cells for experiment 1, and the second containing 10.14 μm polystyrene particles and viable yeast cells for experiment 2, were employed to evaluate the performance of the signal excitation source.

The sample mixtures were introduced into the fluidic micro device using a micropipette. The micro device was mounted under an inverted epifluorescence video microscope for micro fluidics SVM340 from Lab Smith. A personal computer was employed to manipulate the communication and operation of the microscope.

5.3 Simulations

Simulation of crossover frequency spectra for different experimental settings was performed using MATLAB. This allowed for the selection of the best suspending medium conductivity, as well as for the selection of the most adequate AC frequencies to be used on the experiments. Dielectric properties for yeast cells were extracted from and from for polystyrene particles. To compute the equivalent complex permittivity of yeast, the multi-shell model presented in was used.

Finite element method based simulations were carried out using COMSOL Multiphysics in order to obtain predictions of the experimental results. An array of 4x5 electrode posts was considered on a plane located at 30 μm above the channel floor. At this height the effect of the planar electrodes located at the bottom of the channel are negligible. The channel geometry is shown in Figure 5.1. Boundary conditions were set to electric insulation at the channel

walls, and uniform AC electric potential at the electrode posts. The mesh for this geometry consisted of 14,208 elements.

Two different experiments were planned: separation of live and dead yeast cells using an AC signal of $V_{peak\ to\ peak}$ with a frequency of 100 Hz, and separation of live yeast cells and polystyrene beads using an AC signal of 5 $V_{peak\ to\ peak}$ with a frequency of 28 kHz. The geometry section from which this curves were obtained is represented by the red line plotted on Figure 5.1.

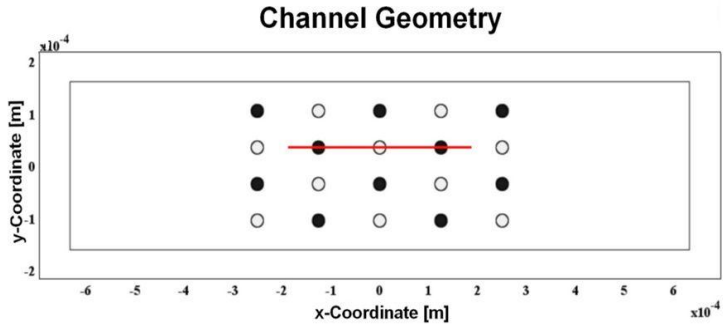


Figure 5.1 *Geometry section of the fluidic device.*

Simulations were performed and estimations of the experimental results are shown in Figure 5.2 where it can be observed that dead cells will experience a positive DEP force, causing the dead cell population to be attracted to the electrode posts. On the other side, live cells will experience a negative repulsive force. However, since the magnitude of this negative DEP force is low, live cells are expected to be found near the posts but not in touch with them. For the second experimental setup, live cells will now experience a strong positive DEP force, while polystyrene beads are expected to be repelled from the posts.

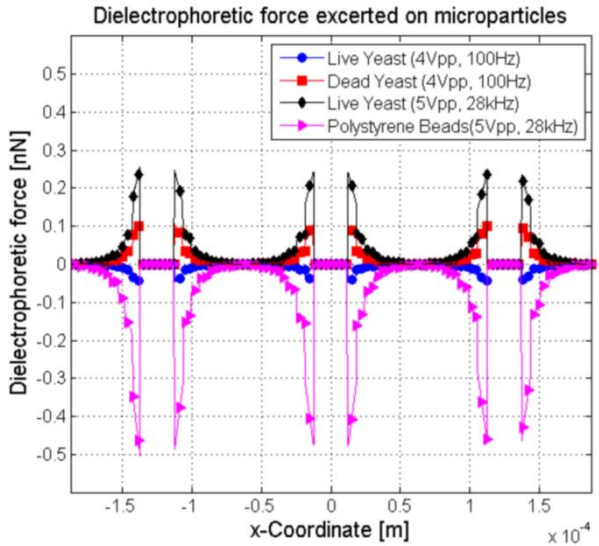


Figure 5.2 *Simulation of dielectrophoretic forces.*

5.4 Experimental Environment

To define reliable experiments the user started by defining test parameters from a previous known base used in manual experiments, like the frequency value known to be effective for a particular manipulation experiment on a specific type of particle. From there, the user can modify parameters such as waveform, frequency, exposure time, or sequence of patterns. These parameters can be changed one at a time or as a set for each test run. Once a set of parameters is found to be effective for a specific manipulation experiment, that test can be precisely repeated with no manual intervention.

This implementation shows a configurable system which delivers single, dual and superimposed $30V_{pp}$ output signals with sinusoidal, saw-tooth and triangle waveforms on frequencies going from 0.01 kHz to 40 kHz. The design is an original application specific architecture which implements a programmable and

configurable dual-frequency and multi-waveform signal generation system. The instrument presented was implemented as a set of components: An application specific user interface, a processor based prototyping board, and a signal conditioning circuit. The C language user interface program was developed to configure the experiment and to control the operation; the processor based development board –the LM3S6965 with an ARM Cortex-M3® processor – runs the application program that generates the electric stimulation signals; the conditioning circuit takes digital data and finally delivers analog signals to a fluidic device.

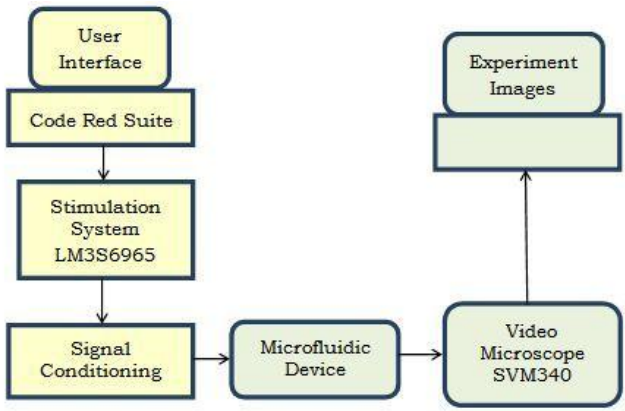


Figure 5.3 *Elements of the board based implementation.*

The core of this instrument is the application software that has been designed specifically for electrical stimulation purposes and has configuration capabilities that allow users to adapt the system to specific tests and applications with no modifications to the hardware or the software. This design can be used as an autonomous stimulation system or can be integrated into Lab-on-Chip designs. Figure 5.3 shows how this stimulation system fits into a particle manipulation setting: the stimulation system running on the LM3S6965 board takes operation

parameters from the User Interface, delivers sine, triangle, and saw tooth wave digital data to the signal conditioning circuitry, which sends analog signals to the micro-fluidic device.

A description of the instrument components is presented:

User interface. Has been developed to define the experiment environment by selecting several operation parameters: select the operation mode between three output options (One single frequency, Two separated frequencies, and Two superimposed frequencies); set the frequencies (base and superimposed) for the experiment; select the number of data samples desired for each frequency; select the exposure time for the test, and start operation when ready for the experiment. Figure 5.4 shows the options for setting operation parameters in the user interface:

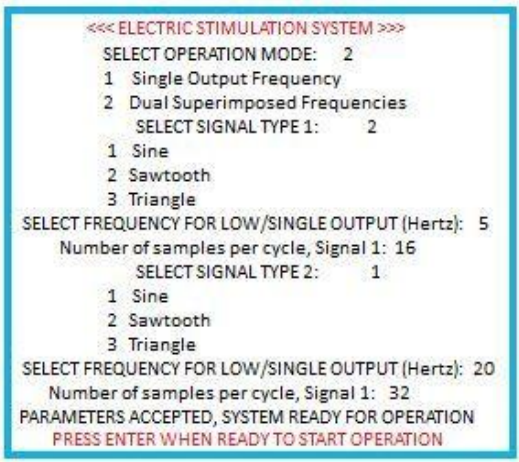


Figure 5.4 *User Interface for the board based implementation.*

Development board. The LM3S6965 - an ARM Cortex-M3® processor based board, shown in figure 5.5 - runs on a 50 MHz clock, has a 256Kb flash memory, 64Kb of SRAM, and up to 42 general purpose output bits grouped in 8-bit output channels. The LM3S6965 stores and runs the application program,

stores processed data needed for the signal generation, and delivers final data to output ports. It has a USB port for the user interface and for re-programming the board. Parallel 8-bit GPIO ports are used to deliver waveform ́s data to the signal conditioning circuitry. The whole system operation is done through the user interface so no manual operation is regularly needed. An emergency start/reset button can be used if an experiment needs to be interrupted before normal operation finishes (Figure 5.5 d).

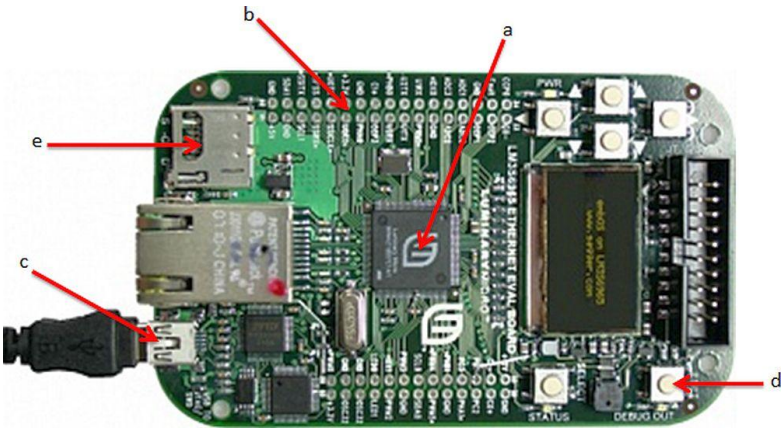


Figure 5.5 *The LM3S6965 prototyping board: a) ARM® Cortex-M3 Processor, b) GPIO port for output digital data, c) USB port for system programming and debugging, and for connecting User interface when in running mode, d) Reset button, e) Memory card slot for extra data and program storage.*

Application Software. Designed for this flexible stimulation environment, it includes the program code and the data tables for the three waveforms. The program contains a frequency synthesis methodology specifically designed for this system, so it can deliver single and dual frequency signals for a more controlled test environment. The program executes operation according to the set of parameters defined by the user and pre-stored data defines the selection of available waveforms. Although, program and data can be modified according to

new users or new needs. Figure 5.6 shows a screenshot of the software development environment, Red code suite ®.

Signal Conditioning. Consists of a digital to analog converter -DAC902- and two AD811 as current-to-voltage converter and voltage amplifiers. Digital data coming from the LM3S6965 board represents single or dual frequency waveforms and are finally converted into a +/-15V analog signal to meet most requirements of current test procedures. The system delivers 2-line analog signals to be applied to the electrodes or stimulation spots in the micro fluidic device.

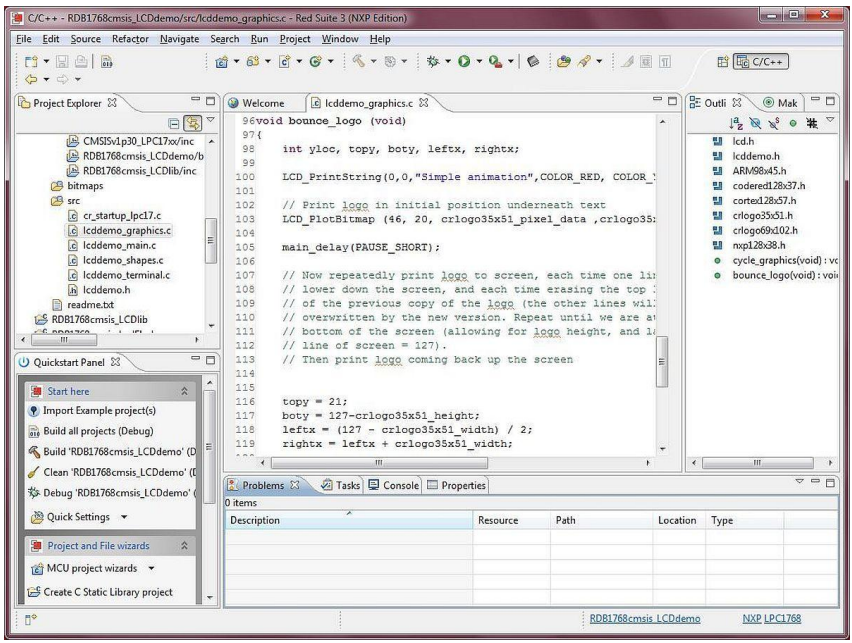


Figure 5.6 *The user interface allows programming, running and debugging the application.*

This system has automated operation: stimulation parameters are selected once, no intervention is needed during execution, and operation can be automatically repeated. This is a programmable implementation since modified

or new applications can be loaded into it. The advantages of this automated, programmable, and intelligent manipulation system are: a) User interface allows to configure and to operate the system for new tests and procedures, b) Previously programmed test parameters for a known test sequence can be stored and accessed later, c) More reliable data results are obtained due to precise reproduction of test parameters, d) Multiple tests can be done and repeated by programming test sequences, e) It can run the current application with the current waveforms or to load and run a different program, and f) It can be integrated to Lab-on-chip implementations or to portable Lab devices.

For experiment continuity, a relevant set of operation parameters can remain loaded in the system for future use: the last set of parameters used for a stimulation experiment is stored in flash memory so the system will perform the last stimulation pattern the next time the system is used, even if it is turned off.

5.5 System Potential

Besides the simulations and experimental results presented, this stimulation system has the potential to be used in a variety of particle manipulation systems.

The flexibility of its operation allows users from different application areas to define a specific stimulation pattern by selecting signal parameters such as frequency, waveform type, superposition, samples per cycle, time of exposure, and sequence.

To show the potential of this system, possible applications have been organized in four types:

- Electric stimulation already in use on particular experiments. Specific experiments from research work that currently use manual or limited electric stimulation show the type of signals used to achieve a particular manipulation effect over a specific type of particles; it is shown here how this system can substitute their stimulation means and improve their research procedures.

- Integrated electric stimulation that can be used in devices currently at proposal or design level. Published particle manipulation systems presented as proposed or demonstration designs that integrate electric stimulation and expose the need of automated stimulation; it is shown here how this system can fulfill those needs by selecting the appropriate set of operation parameters.

- Stimulation presented by theory on ACEK (Alternate Current Electro Kinetics). Existing theory about electric stimulation for particle manipulation presents the possible applications in a variety of fields by using simulations or theoretical demonstrations; it is shown here how this system can be used to comply with almost every application area that needs electric stimulation.

- Applications which use electric stimulation, even if it is not directly related to fluidic systems. Such potential applications go from impedance spectrometry for cell characterization, dielectrophoretic characterization, signal generation for DNA hybridization, or electro-rotation based systems, to completely different research areas such as implantable prosthetics, where new designs of prosthetic devices need to be tested with electric stimulus similar to those received from a live nervous system.

Tables 5.2, 5.3 and 5.4 show referenced research works that could use this stimulation system. For each experiment or device, it can be seen in the tables whether the system can be used as it is or if modifications in the program application are needed to fit in that particular experimental setting.

Experiments. Experiments extracted from the reference list expose a specific manipulation purpose over a particular type of cells or particles, so a specific frequency value or a limited frequency range is used for a particular experiment. Table 5.2 presents the electric stimulation used in actual experiments to show that the stimulation system presented in this work can be used as the stimulation module instead of generic signal generators and manual procedures. Regarding the Voltage amplitude all the experiments require values within the range delivered by this system. Even if higher voltages were needed they could be achieved with additional amplifying stages in the signal conditioning circuitry without modifying the SoC or the platform based design.

Table 5.2 *Particle manipulation experiments.*

Reference	Experiment	Electric stimulation used	Purpose
1992	Experiment: Dual-frequency dielectrophoretic levitation of Canola protoplasts	Sine, f < 1kHz	Compare Single vs. Dual Frequency effect.
2003	Separation of bioparticles using the travelling wave dielectrophoresis with multiple frequencies.	+-6V, 200kHz, 2 superimposed frequencies	Separate red cells from lymphocytes T in a blood sample, simultaneous PDEP and NDEP.
2005	Study of two-frequency dielectrophoresis effects on a linear array.	2 superimposed frequencies: 10Kz+500 Hz, 10- 20 V_{pp}. Large particles NDEP & small particles PDEP. F1<100kHz, f2>300kHz	Main f+low f broaden frequency range for particle separation. T from red blood cells.
2008	Real-time continuous dielectrophoretic separation of malignant cells.	7Vrms, 30-50 kHz	Separate MD231 breast cancer cells in blood.
2008	Micro fluidic Device for DEP Manipulation and Electro-disruption of Respiratory Pathogen Bordetella pertussis.	Sine, 10V, 1 MHz AC and DC sequence patterns.	Manipulation of respiratory pathogens.

Devices. As seen from the State of the Art section in chapter 1, published research works that involve a manipulation device, even if the work is at proposal or design stage, they do not integrate the electric stimulation circuitry in the design, or they talk about a limited integration proposal or a partially configurable demonstration chip. From those device proposals a set of functionality and stimulation parameters were extracted and presented in Table 5.3 to determine if the system presented here can apply to those proposed devices. It was found that the electric stimulation conformed by the signals and patterns delivered by this system are useful in most of the proposed devices. In those cases where V and –V are needed for the device an external high frequency inverter can be used to obtain –V from the original +V.

Table 5.3 *Particle manipulation Devices.*

Reference	Device	Electric Stimulation	Purpose
2000	Micro fabricated multi-frequency particle impedance characterization system	100kHz-10MHz, Sine signal, frequency sweep.	Resistive and reactive impedance measure for characterization of particles and cells
2003	A CMOS Chip for Individual Cell Manipulation and Detection.	2 Sine voltages, phase and counter-phase v1=-v2. Stop and go stimulation for grab & drag. 3.3-9.9V_{pp}, in kHz range.	Detect and manipulate Eukaryotic cells 20-30 μm.
2003	A SoC bio-analysis platform for real-time biological cell analysis-on-a-chip.	Typical DEP stimulation within a frequency range	Multi Bio-analysis.
2003	A programmable dielectrophoretic fluid processor for droplet-based chemistry.	Up to 180V_{pp}, 5-500kHz, varying voltage and frequency	Manipulate contaminants, chemical reagents, virus, and cells.
2005	All CMOS Low Power Platform for Dielectrophoresis Bio-Analysis.	one of four sine signals, 8 different phases, frequency sweep 1kHz-5MHz	Show effect on poly-styrene beads.
2007	A High-Voltage SOI CMOS Exciter Chip for a Programmable Fluidic Processor System Current.	100V_{pp}, up to 200Hz, sine, 0 & 180 °phase. Varying phase, amplitude, and frequency	Use multiple droplets, set a particle route for different particles.

Reference	Device	Electric Stimulation	Purpose
2007	A Programmable Biochip for the Applications of Trapping and Adaptive Multi-sorting.	Sine, $8V_{pp}$, 1MHz, V_1=-V_2, Lab-view controlled	Multi-sorting of proteins and DNA
2009	A robust electrical micro-cytometer with 3-dimensional hydro-focusing.	Sine, 4Vrms, 50kHz	Electrical impedance sensing to detect T cells in blood, for HIV diagnosis.

Theory. From the early research works on particle manipulation to recent publications about more complex stimulation for highly controlled environments, a summary of the AC signals and patterns needed for stimulation are shown in Table 5.4 to illustrate that the mathematical proof and simulations also lead to signals and frequency ranges be covered by this system. Travelling Wave Dielectrophoresis is a specific sequence of Sine signals synchronized to form a travelling electric field that produces a drag effect on the particles within a sample.

Table 5.4 *Particle manipulation Based on Background Theory.*

Reference	Device	Electric Stimulation	Purpose
2003	AC Electro-kinetics—Colloids and Nanoparticles.	Single frequency DEP, TWD, 4 phase Sine	Show mobility effects of AC stimulation
2004	Dielectrophoresis-based programmable fluidic processors.	40V-100V, 1kHz, 2KHZ. 0 and 180 °phase to neighboring electrodes.	Titrating, moving and mixing polar and non-polar, conductive or not, droplets
2004	Sample handling in general-purpose programmable diagnostic instrument.	4 de-phased sine to repel & attract, TWD to concentrate particles in a spiral electrode array.	If f>200kHz all viable cells can be Trapped
2007	Interactions of electrical fields with fluids: laboratory-on-a-chip applications.	2.2Vrms, 100 Hz, 500Hz, 1kHz, $35V_{pp}$ @ 100kHz, $24V_{pp}$ @ 1kHz	Describes ACEK experiments: ACEO, ACDEP, & ACET.
2010	Controlled micro-particle manipulation employing low frequency alternating electric fields.	0.2-1.25Hz, 750V	Show the potential of manipulation using AC fields.

Following Table 5.5 allows visualizing a variety of experimental settings. An experimental setting is defined by selecting the operation parameters for a

specific manipulation purpose. Note that changes in one or several parameters define a whole new experiment setting and purpose.

Parameters that can be changed:

Operation mode: Single or Dual frequency. Waveform type: Sine, Triangle, Saw tooth. Separate or superimposed frequencies. Frequency values for one or two signals. Number of data samples per waveform cycle. Waveform selection for superimposed frequencies.

Table 5.5 *Samples of Experimental Settings.*

Setting	Output channels	Operation mode	Frequencies	Waveform
1	1	Single	50 Hz	Sine
2	1	Dual	Superposition, 10 Hz + 200 Hz	sine over sine
3	2	Dual	Separate, 5 Hz, 400 Hz	Sine and sine
4	1	Single	300 kHz	Triangle
5	1	Dual	Superposition, 2kHz, 10kHz	Saw tooth over saw tooth
6	1	Single	8 kHz	Sine
7	1	Dual	Superposition, 1 kHz + 5 kHz	Triangle over sine
8	2	Dual	Separate, 5 kHz, 10 Hz	Sine, sine
9	1	Single	15 kHz	Saw tooth
10	1	Dual	Superposition, 1kHz, 7kHz	Triangle over saw tooth

Set an experiment through the user interface.

Defining a specific experiment consists of selecting the appropriate set of operation parameters in the user interface. Parameter values can be known from previous experiments or from simulations. At least an idea of the convenient frequency and voltage range is needed. Once the fluid sample has been prepared and put in the fluidic device, the terminals that will carry the electric stimulation

have to be connected to the device. Microscope and camera set have to be ready too.

Here is presented the parameter selection, accessed through the user interface:

Operation Mode. 1 For Single frequency output, 2 for Dual superimposed frequencies, and 3 for Dual separate frequencies.

Waveform for Signal 1. Choose between sine, triangle, and saw-tooth.

Frequency for Signal 1, f_1. Signal 1 is the low frequency signal for the superposition mode.

Number of samples for Signal 1, n_1. The higher the number the less harmonic components are found in the output signal and the lower output frequency can be achieved.

Waveform for Signal 2, requested only if operation mode = 2 or 3.

Frequency for Signal 2, f_2. Is the high frequency to be superimposed on the low frequency Signal 1.

Number of samples for Signal 2, n_2. The amount of memory space needed for the final output table containing sample with both superimposed frequencies could increase significantly if $f_2 >> f_1$. The amount of needed memory space for output buffer table:

$$M_{size} = \left[\left(\frac{f_2}{f_1} \right) n_2 \right] bytes$$

It is recommended that $n_2 < n_1$ to prevent that.

The exposure time for the stimulation, t_{exp}. The exposure time is achieved by defining the number of waveform cycles to be delivered, N. Since f_1 is the base frequency in the case of superimposed frequencies, then

$$N = f_1\left(t_{exp}\right)$$

If a sequence of a different stimulation signal is needed, a similar set of parameters has to be provided.

If the same test or sequence has to be repeated, the same set of parameters is automatically used by the system if re-run.

A set of experiments is described to show the flexibility of this stimulation system.

A specific manipulation experiment.

As shown in tables above, some experiments have already defined the particular set of parameters needed to manipulate a specific type of particles or cells. This set of parameters is introduced once in the user interface and execution are repeated over new fluid samples without changing the parameter set. Another scenario is that the user has an idea about the parameters to be used in an experiment, but not the exact values. In this case user can play with the parameters until the appropriate set of parameters is found.

Once the exact set of parameters is known by achieving the desired manipulation effect, the values can be stored and accessed later to precisely reproduce the experiment.

A particle characterization experiment.

Particle characterization experiments may need a frequency sweep in x10 steps to first determine a smaller range to work. A set of sequences can be defined, and a special case were $f_{new}= 10*f_{old}$ can be defined in program to cover all the frequency range, going through 0.1 Hz, 1 Hz, 10 Hz, 100Hz, 1kHz, 10kHz, …up to the maximum output frequency.

For example, in a sensor measures resistive and reactive impedance of circulating particles. Particle impedance is measured at three or more frequencies simultaneously, enabling the derivation of multiple particle parameters such as blood granulocyte radius, membrane capacitance, and cytoplasmic conductivity.

A frequency sweep experiment.

Some experiments require observing the mobility effect under different frequencies. In those cases the whole frequency range delivered by this system can be swept in user defined steps. An initial f_i frequency is selected, a frequency step f_s is defined, and a time period t_r for each repetition is introduced. This way each following t_r a new frequency $f_{i+1=}f_i+f_s$ is delivered during t_r seconds. In a feasible procedure that uses DEP phenomenon as a method of separation of the abnormal cells from the blood stream is presented.

Negative and positive DEP (NDEP and PDEP) forces generated by a non-uniform electric field are engaged to separate the normal blood cells from the malignant ones. By fine tuning the parameters of the electric field different types of abnormal cells are isolated. It is noticed that at a frequency of 30 kHz all blood cells and the cancer cells experienced a PDEP, and the cells started accumulating in the area of low electric field. Increasing the AC frequency to 50 kHz, the cancer cells experienced PDEP and gathered over the tip of the electrodes array

where the maximum electric filed is present. At the same time the blood cell still with from the electric field. To perform a similar procedure for different cell types within a blood sample, a frequency sweep experiment can be used.

A dual frequency experiment.

If two different particles are present in the same fluid sample, they can be separated by applying two frequencies simultaneously. Particles can be different in type, size, or of the same type but different because one are alive and the others are dead, or because they present a different development stage. In there is an analysis for a mixture of two different types of particles: they choose an angular frequency, w, such that the real parts of the Clausius Mossotti function at U (or Re[G(jw)]) of the two different types of particles have different signs). Then an electric field produces time-averaged dielectric forces in such that the particles with Re[G(jw)]> 0 get attracted to the maximum points of the field, and the particles with Re[G(jw)]< 0 get repelled away from those points.

In a similar analysis they consider an example where the goal is to separate two types of latex balls with a very small difference between both cross-over frequencies, so that the electric field of single frequency is not effective.

A saw-tooth waveform experiment.

It has been shown in previous table that saw-tooth waveforms are useful in a drag-trap effect; during the linear voltage rise the particles are moved to a certain point, and once the voltage exceeds certain level they remain trapped. An interesting effect can be achieved when a saw-tooth over a sine signal is used, because two different types of particles are manipulated, and the more distant the two frequencies, the more different the particles.

Chapter 6

Integrated Circuits for Intelligent Systems

The term Intelligent System is being used in a more extensive and inclusive way. It covers systems that perform intelligent functions, understanding by intelligent that it seems to think and decide, based on information it has and information it takes from the environment. An easy example to understand this concept is a Smartphone's. They are called smart because they read the environment and take actions based on what they find: they detect if Wi-Fi networks are available, they use the GPS to know where they are, and so on. In consequence of those readings it can tell the user what to do or use. The decision making process involves hardware and software within the smart device. A robot is also smart, in the way its software and hardware allows him to be. The more sensors it has, and the more complex its program is, it will be more intelligent and can be able to control more output elements and perform more actions.

Nowadays people are more familiar with intelligent systems and, by now, people born in this decade cannot imagine a functioning world without them. We all now count and rely on intelligent systems for everyday activities such as electronic banking, automated parking, electronic document sharing, and permanent communication capabilities, among others.

More recently, connectivity capabilities are becoming the more important feature of an intelligent device or system. Specific functionalities of each device can still grow and innovate, but it is more and more important that a device can connect with other, similar or different, devices and systems.

This is how the concept of smart cities, smart grid, and smart cars has been defined. A smart system consists now on a set of interconnected devices, either they are of the same nature and purpose, or not.

The following examples, although its outside is known by everyone, show that all systems are the same inside: they have a processor to execute the instructions, a memory to store that instructions, sensors to detect what they need from the environment, actuators to perform the functions, and communication capabilities to connect with others.

Smart house: The control system includes light sensors, motion sensors, proximity sensors, temperature sensors, timers, in order to operate the lighting network, the alarm system, the air conditioning system, the access doors, and so on. The more elements it has, the smarter it looks.

Smart building: Same idea than the smart house, and additionally it may include collective access control, separated areas air conditioning, access record, access reports, energy efficiency programs, personnel data base. You can notice that sensor and actuators sound similar than those in smart houses, but processing and storage capabilities need to be larger.

Smart parking: Commercial centers, Corporate buildings and Residential complexes use to have access control, assigned placement, and space optimization. For this, they need a smart control, like the ones mentioned in previous smart systems: sensors, actuators, program, data management and storage, user interface. The administration of space, maintenance cost, users and rates are now common elements in parking systems. And what if you need to know in advance, prior to your arrival to the parking lot, if they have spaces available and what is the current rate? The system should be available through a web page or an app, so users do not discard this parking from their options by not having that information available anytime and anywhere.

Smart grid: As the green movement becomes more important and global impact on energy resources is a regular element in business decisions, also is becoming important the smart grid concept in large cities. The grid that supplies energy to the city (cables, stations, transformers, and measuring devices) can be aware of the user's consumption habits and needs. The energy demand of a city, and of every city district or area, depends on the season, the time of the day, and the day of the week, among other factors. It is useful for the energy provider to know the demand patterns so they can manage the energy distribution, maintenance tasks, rates, and so. A smart greed consists of a regular energy grid plus the needed sensors and measurement devices to know and predict the energy patterns and take decisions for energy optimization and use.

Smart cars: People still use to name the smart system of a car as the "car computer". It was a proper name when the concept began, because the first cars used to have one system that received signals from simple sensors like rpm readers, impact sensors, and proximity detectors. With this signals and a simple program, a central computer decided things like activate the airbags when an impact occurred, activate the ABS when the regulars breaks were not enough for an efficient speed reduction, and to activate a bip signal if the car was too close to the next car, the sidewalk, or an object behind. But, as processes became more complex, the need for independent controls arose and cars had more than one computer. For instance, a state of the art car, these days, has more than 60 independent intelligent controls or "computers": one for the fuel injection system, one for ABS, one for the security tasks, one to collect and store all the info needed for the maintenance procedures (have you noticed that today's mechanics is not about checking under the car to find out what the malfunction comes from, but to download the computer information to analyze the sensor's measurements over time, and what the system is concluding the

problem is), one for entertainment, and the list of new needs will never stop. And of course, all the systems need to know what is going on with the other sub-systems, as their decisions depend on the other's decisions. At this point, the need for a local network between subsystems is needed, so all of them require communication or interconnection capabilities, and a central system to coordinate the operation between them.

Smart city: What will happen next, after many of the systems living in a city have their own intelligence? The obvious next step is to connect them all together and see what additional intelligence can result from that. The smart grid can know the energy consumption patterns from the smart houses, smart buildings and smart parking. The smart cars can take advantage from the traffic information collected from the City Traffic System. All the smart systems in the city can be accessed using a data center, so any store, service center, weather center, manufacturer and user, can access information in real time and take smart decisions.

Smart manufacturing: When a company already has stable in quality control schemes and lean manufacturing, decides to move towards smart Manufacturing.

We are accustomed to using terms like Smartphone, Smart TV and Smart Cars; soon it will be Smart House, Intelligent Building, Smart City and finally, intelligent planet, meaning that an intelligent system uses its resources to create, manage, and use information to help you make decisions and actions wisely.

Referring to the context of manufacturing, it covers to have information in real time, ensuring its flow and access, and maintain integrated and scalable on which to base all business decisions. This will fundamentally change the way products are invented, manufactured, transported and sold.

When a company decides to join the Smart Manufacturing trend, it will inevitably find other issues when integrating intelligence: Safety and interoperability of data, modeling of production, simulation market, Sustainable Production, Integrated processes, Sensor Networks, Knowledge Management, Zero emissions and, of course, cloud Computing. These terms are not new, but now they must be integrated into a system that includes planning, production, operation, and vision of the company.

Now, if we understood the idea of system intelligence, and we were convinced that we must make the transition, we can classify the next steps into 3 phases:

Integrate into one system all the information from all lines, processes and products of the company. It will take time but it is essential. Since IT resources, sensors, motors, automatic controls, and software to manage production, but each is an efficient island.

Make models and simulations that allow flexible manufacturing, demand production and product customization when markets change rapidly.

When the previous phases progress, create scenarios for innovation, and manage to break the paradigms of today's markets. These breaks are generated by innovative technologies in processes and products. This phase will reverse the traditional chain where the consumer was forced to buy what it was mass produced.

6.1 The Smart Systems and the Integrated Circuits

Having said that, a question arises: What’s the link between Smart systems and Integrated circuit design? The answer is that the core of an intelligent is,

mostly for sure, an integrated circuit. Not a generic or over-the-counter circuit, but an Application Specific Integrated Circuit.

If you were able to open a smart phone, or the car computer, or the robot brain, you will find, at the end, an integrated circuit specifically designed for that purpose. More often than not, it will be only one integrated circuits that includes processor, memory, communication ports, and even sensors and actuators.

What is important about this book is that the design procedures described here are universal and non-dependant of the application or need you want to solve.

ASICs for commercial products. ASIC stands for Application Specific Integrated Circuit, so it means somebody detected an opportunity to develop an original circuit to attend a specific need, and then developed a circuit specifically for that purpose. Examples of this are: A commercial brand for refrigerators decides, for the first time, that it will be useful to have internet connection available in its refrigerators, so the customer can connect from any place and check what's in the fridge, or to send a list to the store every weekend of what is missing and have it delivered to your home. The circuit designer starts from the previous circuit (not from scratch) and adds the needed circuitry to complete the monitoring and detection tasks. If successfully designed, it will lead to what is called an intelligent system, because it apparently understood what happened in your refrigerator, decided that you needed more bananas and milk, and ordered them for you. In this example, you design a circuit specifically intended for that application. The understanding and decision making was made by the carefully designed combination of circuit and program. By the way, if you take this same circuit and connect it to your microwave oven it will not know what to do or will do something wrong. So, this is an Application Specific Integrated Circuit.

It may have sounded casual, but the fact that you should not start a design from scratch, but from what has been already designed before, is one of the main rules of circuit design: You should never start a new design assuming nothing like it has ever been done before. If you start from zero, it will take you more time than other designers to get your idea implemented, and by then you will be out of the competition for your idea's market.

In this time and age where there is a solution for almost everything, you may think: what can I design if everything is already done. Nothing more wrong than that. The more complex our environment is, the more opportunities for ideas we have.

6.2 ASIC for Customized Applications

There are design opportunities that may find broader fields of application, meaning that you want to design a circuit that has some of the functionality defined and limited by the cores it has inside, but other functionalities can be defined by the final user. Which is final by design is the hardware, of course, but you can load a small operative system or a program application that allows the user to program his own application, purpose specific, and can load it in the memory Space, you as the designer, left available for that.

In the same sense, a designer can provide external access to the modules inside the integrated circuit, by having the in-chip address bus, available on external pins. This will increase the pin out of the IC, the packaging and will move the place-and-route, but the benefit of an open system supersedes the design difficulties. The sample application presented in this book shows how this can be done. It illustrates concepts like:

Open architecture: the integrated circuit has the address bus available in the external pin out, so other devices using the same bus can be connected to it. As long as address assignment stays compatible, the interconnection of other devices has no limit through this bus.

Programmable: a memory space is reserved to download a different application program, or additional functions to those already loaded.

Configurable: Many functions are preloaded in the memory chip, by design, but in the user interface of the application some of those functions can be disabled, so they do not take execution time.

6.3 Design and Market Trends

The integrated circuit market have been revolving around developing faster, smaller, and less power consuming components, and it will continue to do so unless a completely different technology is developed.

Each of the 3 variables depends on two factors: the technology and tools available at the moment, and the designer technical capabilities and knowledge. None of them completely compensates the other, so both, the technology and the developer, need to be good to achieve a usable product.

Faster: As silicon transistor based technologies became smaller, they are also faster. That way we went from tenths of nanometers to a single digit figure. As for edition time of this book, smaller has always been more expensive to fabricate, so older machines working with larger transistor sizes are cheaper but no big companies want cheaper and slower circuits. That way, the cheaper fabrication options are good for beginners or universities on small budgets. The

developer competences come in play when designing the HDL program: a designer needs to keep in mind that loops, variable assignment, variable sizes, data transfers and so, will finally translate into circuits. And circuits can be efficient in their implementation or not. A simple example to understand this idea is to think on a simple adder being implemented in a protoboard by 2 students; each of them can have a different idea on how to do it, then use different gates or array of gates, or use multiplexers. At the end, the two circuits will be different in appearance, in size, in gate count and, consequently, in response time. This illustrates how your programming style impacts your design size and speed. Besides, your programming language is also a factor: designers who prefer to program in C see this idea clearly: when a C program is transferred to HDL the sizes are completely different meaning there is no optimization possible when you program in a high level language that does not allow you to see how your program will look when in the final language.

Smaller: As mentioned in the paragraph above, silicon based technologies became smaller over the years. There is a size limit as connections and transistors need to transport electrons, and electrons have dimensions. Under this idea, the smallest a wire or transistor can be, is related to the electron size, so it can freely transit through it without reducing its speed or overheating the wire. About the designer skills, core size and placement are the main issues in circuit size. Core size comes from the synthesis process, where HDL design is translated into circuits, and the programming style impacts the resulting circuit. Once each core is size optimized, the pin placement determines where each core is going to be placed within the integrated circuit; space between cores is needed for interconnections, so you want to be safe and leave more space than needed, but external pin-out may demand that cores be placed differently than interconnections suggest. There is no single solution for a good route and

placement, which is one of the more careful design processes, other than the cores design.

Less power: Power consumption by an integrated circuit is separated in two types: active and passive consumption. Passive is the power it takes to keep the circuit ON even if it is not running the application or any part of the program; this state could be named as Stand-by. Active consumption is when the circuit is operating or running. Of course active consumption is larger than passive, but active is not a fixed or constant figure: Not all parts of the hardware and software architecture are being used in every function of the system, so the power consumption rate depends on the function currently executed or performed by the circuit. In a simple integrated circuit, as a 4-NAND gate circuit, it is easy to estimate the amount of power being consumed if one, or two, or four gates are ON. For an integrated circuit that has a running processor executing a complex application program, a power simulation is needed to determine the low and high consumption peaks.

Modularity is also a trend, meaning that a complete system is built over interchangeable blocks that can define capabilities and functionality using a common platform. Modular systems are upgradeable by definition, as the user can change the processor, the memory or storage capabilities, the communication components, and so on.

Modularity in hardware, for circuit designers, is a constant in any design, as no one starts a new design from scratch, but from the previous product or from something similar. Design teams work by developing independent and coherent blocks that will finally complete the hardware architecture. But for the final user of a product, like a cell phone, a computer or a tablet, the product is a closed

hardware architecture where the user chooses the architecture when buying a product, and can do nothing or just a little to add or upgrade circuits or blocks in it.

Modularity in software has been here for a long time, as it is the concept behind Apps: the main operative system is the foundation, and the apps are the added blocks to complete a different software system based on the preferences of each user.

Figures 6.1 to 6.4 show examples of applications for embedded and intelligent systems, in every day uses.

Figure 6.1 *Example of embedded systems in automotive.*

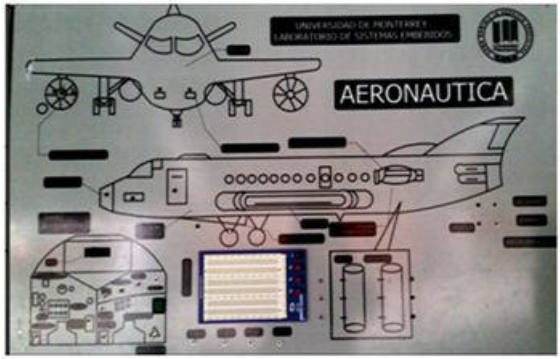


Figure 6.2 *Example of embedded systems in aeronautics.*

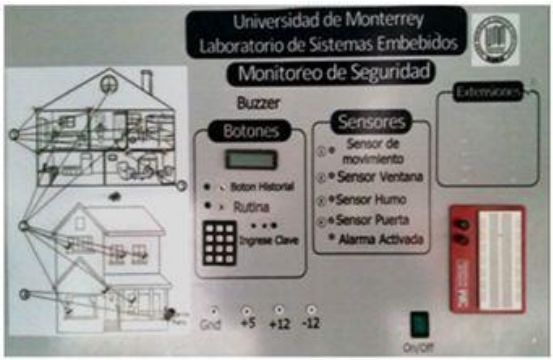


Figure 6.3 *Example of embedded systems in safety monitoring.*

Figure 6.4 *Example of embedded systems in home appliances.*

Glossary

ASIC: An integrated circuit designed for one particular use, such as substituting many small integrated circuits with a larger but specific one.

Address bus: A unidirectional set of signals used by a processor to point to memory locations in which it interested, in a certain device or circuit.

Analog: A continuous value that most closely resembles the real world and can be as precise as the measuring technique allows.

Analog circuit: A collection of components used to process or generate analog signals.

Bit: A zero or one value or representation in the binary language of computers.

Byte: a package of 8 bits.

Clock tree: This refers to the way in which a clock signal is routed throughout a chip. This structure is used to ensure that all of the flip flops see the clock signal as close together as possible.

Custom circuit: An Integrated circuit designed and manufactured for a particular customer.

Data Bus: A bidirectional set of signals used by a computer to convey information from a memory location to the central processing unit and vice versa.

Design flow: Design flows are the explicit and graphic combination of electronic design automation tools and representation to accomplish the design of an integrated circuit.

Die: The small piece of the wafer on which an individual semiconductor device has been formed.

Digital Circuit: A collection of logic gates used to process or generate digital signals.

Diode: A two terminal device that conducts electricity in only one direction.

EDA: Electronic design automation is a category of software tools for designing electronic system such as printed circuit boards and integrated circuits.

Hardware: Generally understood to refer to any of the physical portions constituting an electronic system, circuit boards, power supplies and monitors.

Hertz: Unit of frequency. One hertz equals one cycle or one oscillation per second.

IC layout: Also known as mask design, it is the representation of an IC in terms of planar geometric, so components can be visualized and placed.

Integrated circuit: A complete electronic circuits composed of interconnected diodes and transistors on a single semiconductor substrate.

IP Core: Reusable unit of cell or chip layout. IP cores are used as building blocks within chip designs.

Micros: A micrometer, or one-millionth of a meter.

RAM: A data storage device from which data can be read out and into, which new data can be written on.

Semiconductor: A material (silicon or germanium) that has four electrons in its outer ring and is a poor conductor of electricity.

Silicon: The basic material used to make the majority of semiconductor wafer.

SRAM: A type of RAM that has self contained memory circuitry. Memories are categorized by speed and by storage capacity.

Transistor: A three terminal semiconductor device used mainly to amplify.

Via: A hole filled or lined with a conducting material, which is used to link two or more conducting layers in a substrate.

Wafer: A thin disk, from 3 to 8 inches in diameter from silicon or other semiconductor material. The same or different integrated circuits can be printed in one wafer.

References

[1] Lopez M, Gerstlauer A, Avila A, & Martinez-Chapa S. (2011). A programmable and configurable multi port System on Chip for stimulating electrokinetically drive microfluidic devices (8361-8364). IEE Conference publications.

[2] Lopez M, Hernandez M. Gonzales H. Martinez S. (2013, March). An electric stimulation system for electrokinetic particle manipulation in microfluidic devices. Review of Scientific Instruments.

[3] Jupiter, XT. (2006). Synopsys, Top-Down Hierarchical Flow, User Guide, Version Y-2006.06.

[4] Gascoyne P, & Vykonkal J. (Jan 2004). Dielectrophoresis-based sample handling in general-purpose programmable diagnostic instrument (Vol 92, pp 22-42). IEEE Proceedings.

[5] Manaresi N, Romani A, Medoro G, Altomare L, Leonardi A, Tartagni M, & R Guerrieri. (2003). A CMOS Chip for Individual Cell Manipulation and Detection. (Vol. 38, pp. 2297-2305). IEEE Journal of Solid-State Circuits

[6] Tierney J, Rader C & Gold B. (1971). A digital frequency synthesizer. (Vol 19, pp 48-57). IEEE Transactions on Audio and Electroacustics.

[7] Choi Y, Kim Y, Im M, Kim B, Yun K & Yoon E. (2006). Three Dimensional Electrode Structure Controlled by Dielectrophoresis for Flow-Through Micro Electroporation System. (pp 466-469). IEEE International Conference on Micro Electro Mechanical Systems.

[8] Chang F, Lee Y & Chiu Ch. (2008). Multiple Electrodes Arrayed Dielectrophoretic Chip with Application on Micro-Bead Manipulation. IEEE Proceedings.

[9] Ibrahim M, Elsayed F, Ghallab Y & Badawy W. (2009). An Electric Field Array Microsystem for Lab-on-Chip and Biomedical Analysis. (pp 89-92). IEEE Conference on Biomedical Circuits and Systems.

[10] Standard library SRAM Generator, from Artisan, User Manuel, revision ug_2004q1v0.

[11] Standard library 0.13um - 0.25 um ROM Generator, from Artisan, User Manuel, revision ug_2004q3v1.

[12] ASIC Design Flow Tutorial, Nano-Electronics and Computing Research Center, San Francisco State University.

[13] OpenCores. (2015). OpenCores. Recuperado el 2015, de http://opencores.org/

[14] Design Compiler User Guide, Synopsys, Version C-2009.06.

[15] Library Data Preparation for IC Compiler, User Guide, Synopsys, Version D-2010.03.

[16] Li H, Yanan Z, Akin D & Bashir R. (2005). Characterization and modeling of a microfluidic dielectrophoresis filter for biological species. (Vol 14. pp 103-112). IEEE Journal of Microelectromechanical Systems.

[17] Yuk K, Mc Conaghy C, Gascoyne P, Schwartz J, Vykoukal J & Andrews C. (2007). A High-Voltage SOI CMOS Exciter Chip for a Programmable Fluidic Processor System Current", K.W.; Biomedical Circuits and Systems. (Vol 1. pp 105-115). IEEE Transactions on Biomedical Circuits and Systems.

[18] Rosa C, Tilley P, Fox J & Kaler K. (October 2008). Microfluidic Device for Dielectrophoresis Manipulation and Electrodisruption of Respiratory Pathogen Bordetella pertussis. (pp 2426-2432). IEEE Transactions on Biomedical Engineering.

[19] Villemejane J, Mottet G, Francais O, Pioufle B, Woytasik M & Dufour-Gergam E. (2010). Nanomanipulation of Living Cells on a Chip Using Electric Field. (pp 229-232). IEEE International Symposium in Electronic Design, Test and Application.

Appendixes

This section has been constructed for easy access to the most relevant information about the developed work. Here is found the application program for the standard and extended versions, an illustration of the signal superposition methodology, the content of the base, temporary, and output data tables, a summary of the user interface, a compact description of the SoC design flow, and the final SoC parameters.

A.1 Application Program: Standard Version

The standard version has been developed for the SoC design. It can be stored in in-chip ROM or uploaded to chip RAM at boot time.

/* Name: boardv2.c

Author: Martha Lopez

Version: Board_Extended_v2, 256 data sine samples, buffer table OK, all frequencies, three operation modes

Copyright: (C) Copyright

Description: Board version, standard functionality, sine, saw-tooth, triangle */

// include files

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

//definitions and declarations

#define Pi 3.14159265358979323846264338327

static unsigned int sinedatint[256] =

{ 127, 130, 133, 136, 139, 142, 145, 148, 151, 154, 157, 160, 163, 166, 169, 172, 175, 178, 181, 184, 186, 189, 192, 194, 197, 200, 202, 205, 207, 209, 212, 214, 216, 218, 221, 223, 225, 227, 229, 230, 232, 234, 235, 237, 239, 240, 241, 243, 244, 245, 246, 247, 248, 249, 250, 250, 251, 252, 252, 253, 253, 253, 253, 253, 254, 253, 253, 253, 253, 253, 252, 252, 251, 250, 250, 249, 248, 247, 246, 245, 244, 243, 241, 240, 239, 237, 235, 234, 232, 230, 229, 227, 225, 223, 221, 218, 216, 214, 212, 209, 207, 205, 202, 200, 197, 194, 192, 189, 186, 184, 181, 178, 175, 172, 169, 166, 163, 160, 157, 154, 151, 148, 145, 142, 139, 136, 133, 130, 127, 123, 120, 117, 114, 111, 108, 105, 102, 99, 96, 93, 90, 87, 84, 81, 78, 75, 72, 69, 67, 64, 61, 59, 56, 53, 51, 48, 46, 44, 41, 39, 37, 35, 32, 30, 28, 26, 24, 23, 21, 19, 18, 16, 14, 13, 12, 10, 9, 8, 7, 6, 5, 4, 3, 3, 2, 1, 1, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 1, 1, 2, 3, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16, 18, 19, 21, 23, 24, 26, 28, 30, 32, 35, 37, 39, 41, 44, 46, 48, 51, 53, 56, 59, 61, 64, 67, 69, 72, 75,78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 114, 117, 120, 123 };

static unsigned int toothsawdat[256] =

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 142, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233,

234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 252, 252, 253, 254, 255 };

static unsigned int triangdat[256] =

{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190, 192, 194, 196, 198, 200, 202, 204, 206, 208, 210, 212, 214, 216, 218, 220, 222, 224, 226, 228, 230, 232, 234, 236, 238, 240, 242, 244, 246, 248, 250, 252, 254, 255, 254, 252, 250, 248, 246, 244, 242, 240, 238, 236, 234, 232, 230, 228, 226, 224, 222, 220, 218, 216, 214, 212, 210, 208, 206, 204, 202, 200, 198, 196, 194, 192, 190, 188, 186, 184, 182, 180, 178, 176, 174, 172, 170, 168, 166, 164, 162, 160, 158, 156, 154, 152, 150, 148, 146, 144, 142, 140, 138, 136, 134, 132, 130, 128, 126, 124, 122, 120, 118, 116, 114, 112, 110, 108, 106, 104, 102, 100, 98, 96, 94, 92, 90, 88, 86, 84, 82, 80, 78, 76, 74, 72, 70, 68, 66, 64, 62, 60, 58, 56, 54, 52, 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4, 2 };

unsigned int timeindex1int[256], timeindex2int[256], timeindexbuffint [256];

unsigned int TempTable1int[256], TempTable2int[256]; // Temporary tables //modes 2 & 3, scale 0 to 255

unsigned int BuffTable[256]; // Output table, data to port

double trunc(**double** arg);

float tbs, tbs1, tbs2; // time between samples, signal 1 and 2

int dat_samples_buff1, dat_samples_temp1, dat_samples_temp2; //number of //samples in output table

int opmode; //operation mode: 1 single signlal, 2 superimposed, 3 separate //signals

```
int signaltype; //signal type: 1 sine, 2 saw-tooth, 3 triangle
float freq1, freq2; //selected frequency for outputs 1 and 2
int N, n1, n2; //samples per waveform cycle, signal 1 and 2
void GetOperParam()
{
printf ("Operation mode: 1, 2 or 3:\n ");
scanf ("%d", &opmode);
//printf ("Selected Operation mode = %d\n", opmode);
printf ("Signal type, 1 sine, 2 tooth, 3 triang:\n ");
scanf ("%d", &signaltype);
//printf ("Selected signal type = %d\n", signaltype);
printf ("Output single/low frequency in KiloHertz:\n ");
scanf ("%f", &freq1);
printf ("Samples per cycle: \n");
scanf ("%d", &n1);
if (n1<9) n1=8;
if (n1>8 & n1< 17) n1=16;
if (n1>16 & n1< 33) n1=32;
if (n1>32 & n1< 65) n1=64;
if (n1>64 & n1< 129) n1=128;
```

```
if (n1>128) n1=256;
printf(" %d\n", n1);
if (opmode>1)
{
printf ("Output high frequency2 in KiloHertz:\n ");
scanf ("%f", &freq2);
printf ("Samples per cycle 2: \n");
scanf ("%d", &n2);
if (n2<9) n2=8;
if (n2>8 & n2< 17) n2=16;
if (n2>16 & n2< 33) n2=32;
if (n2>32 & n2< 65) n2=64;
if (n2>64 & n2< 129) n2=128;
if (n2>128) n2=256;
printf(" %d\n", n2);
printf ("Leaving function GetOperParam\n");
}
}
void SineDisplay(int N)
{
```

```
unsigned int iter;
printf ("Entering function SineDisplay ORIGINAL SINE TABLE \n");
for (iter = 0; iter < N; iter++)
{
printf("[%d] ", iter);
printf(" %d\n", sinedatint[iter]);
}
}
void BuffTableGen(int N, int n)
{
unsigned int dsepi;
double dsepd;
unsigned int iter;
unsigned int i;
unsigned int j;
printf ("Entering function BuffTableGen mode 1\n");
i=0;
dsepd=N/n;
dsepi=dsepd;
j=dsepi;
```

```
printf("data separation int, mode1 = %d\n ", dsepi);
for (iter = 0; iter < N; iter=iter+dsepi)
{
/*printf("Data number = %d\n ", iter);*/
if (signaltype == 1) BuffTable[i]=sinedatint[iter];
if (signaltype == 2) BuffTable[i]=toothsawdat[iter];
if (signaltype == 3) BuffTable[i]=triangdat[iter];
/*printf("Original data = %.6ef\n ", sinedat[iter]);*/
printf("%d\n ", BuffTable[i]);
dat_samples_buff1=i;
i=i+1;
}
printf("Amount of data samples in buffer table = %d\n ", dat_samples_buff1);
printf ("Leaving function BuffTableGen mode 1\n");
}
void TempTable1Calc(int N, int n1)//prepare temp table signal 1, modes 2 & 3
{
unsigned int dsepi;
double dsepd;
unsigned int iter;
```

```
unsigned int i;
float t;
printf ("Entering function TempTable1Calc, modes 2 & 3\n");
i=0;
dsepd=N/n1;
dsepi=dsepd;
if ((dsepd-dsepi)>0.495)
{dsepi++; //if separation is 12.5 round up to 13
printf ("Table 1 separation %d \n", dsepi);
}
//printf("data separation in TEMPORARY TABLE 1 = %d\n ", dsepi);
printf ("it time out data\n");
for (iter = 0; iter < N; iter=iter+dsepi)
{
t=tbs1*i;
printf("[%d] >", iter);
printf("[%d] ", i);
//printf("%.3ef ", t);
if (signaltype==1) TempTable1int[i]=sinedatint[iter];
if (signaltype==2) TempTable1int[i]=toothsawdat[iter];
```

```
if (signaltype==3) TempTable1int[i]=triangdat[iter];

timeindex1int[i]=100000*t;

printf("t=%d ", timeindex1int[i]);

printf("%d ", TempTable1int[i]);

printf("%x\n ", TempTable1int[i]);

dat_samples_temp1=i;

i=i+1;

}

//printf("Data samples in temporary table1 = %d\n ",

// ( at_samples_temp1+1));

//printf ("Leaving function TempTable1Calc, modes 2 & 3\n");

}

void TempTable2Calc(int N, int n2)//prepare temp table signal 2, modes 2&3

{

unsigned int dsepi;

double dsepd;

unsigned int iter;

unsigned int i;

float t;

printf ("Entering function TempTable2Calc, modes 2 & 3\n");
```

```
i=0;
dsepd=N/n2;
printf(" %.2ef ", dsepd);
dsepi=dsepd;
if ((dsepd-dsepi)>0.495)
{dsepi++;
printf ("Table 2 separation %d \n", dsepi);
}
//printf("data separation in TEMPORARY TABLE 2 = %d\n ", dsepi);
printf ("it time out data\n");
for (iter = 0; iter < N; iter=iter+dsepi)
{
t=tbs2*i;
printf("[%d] >", iter);
printf("[%d] ", i);
//printf("%.3ef ", t);
if (signaltype==1) TempTable2int[i]=sinedatint[iter];
if (signaltype==2) TempTable2int[i]=toothsawdat[iter];
if (signaltype==3) TempTable2int[i]=triangdat[iter];
timeindex2int[i]=100000*t;
```

```
printf("t=%d ", timeindex2int[i]);
printf("%d ", TempTable2int[i]);
printf("%x\n ", TempTable2int[i]);
dat_samples_temp2=i;
i=i+1;
}
}
void BuffTableSuperposition()////prepare output table, mode 2
{
unsigned int i, j, k, l, m, dato1, dato2;
float t, tbsmin, tmax;
unsigned int tint, auxt1, auxt2;
printf ("Entering function BuffTableSuperposition, mode 2\n");
if (tbs1<tbs2)
tbsmin=tbs1;
else
tbsmin=tbs2;
if (freq1<freq2)
tmax=1/freq1;
else
```

```
tmax=1/freq2;
t=0; i=0; j=0; k=0;
dato1=TempTable1int[i];
dato2=TempTable2int[i];
BuffTable[k]=dato1+dato2;
printf("%d ", k);
printf(" %.2ef ", t);
printf(" %x ", dato2);
printf("+ %x ", dato1);
printf("= %d ", BuffTable[k]);
printf("= %x\n ", BuffTable[k]);
do{
t=t+tbsmin;
tint=t*100000+1;
auxt1=timeindex1int[i+1];
auxt2=timeindex2int[j+1];
m=0;
if (tint<auxt1)
l=0;
else{
```

```
m=1;
i=i+1;
//printf("new data table111, index %d \n", i);
dato1=TempTable1int[i];
}
//printf("m value after checking table1 %d \n", m);
if (tint<auxt2)
l=0;
else{
m=2;
j=j+1;
if (j==n2) j=0; // return to begin of temp table for low frequency
dato2=TempTable2int[j];
}
if(m>0)
{
k++;
if (opmode==2) BuffTable[k]=(dato1+dato2)/2;
if (opmode==3) BuffTable[k]=dato1+dato2*256;
timeindexbuffint[k]=tint;
```

```
printf("[%d] ", k);

printf("t= %d ", tint);

printf(" %x ", dato2);

printf("+ %x ", dato1);

printf("= %d ", BuffTable[k]);

printf("= %x\n ", BuffTable[k]);

}

}

while (t<tmax);

dat_samples_buff1=k;

printf("Amount of data samples in BUFFER TABLE = %d\n", k);

}

void tbsCalc(float freq, float n)//calculate time for requested frequency //
and number of samples

{

printf ("Entering function tbsCalc\n");

tbs=1/(freq*n);

}

void WriteToOut()//load data from output table, write to output port

{

unsigned int i;
```

```
unsigned int j;

unsigned int k;

float t;

printf ("Entering function WriteToOut\n");

printf("Time running between samples buffer1 = %.2ef\n", tbs1);

printf ("it time out data\n");

for (i = 0; i < dat_samples_buff1-1; i=i+1)

{

for (j = 0; j < tbs1*1e+5; j=j+1)// 1e+5 proportional to time // between samples

k=k+1;

t=tbs1*i;

printf("[%d] ", i);

printf(" %d ", timeindexbuffint[i]);

printf(" %x\n ", BuffTable[i]);

}

printf ("Leaving function WriteToOut\n");

}

int main(void)

{

N=256;
```

```
GetOperParam(); //Get operation parameters
//SineDisplay(N); //Display data samples for sine waveform
if (opmode==1)// operation mode = 1?
{
tbsCalc(freq1, n1); //calculate separation between samples
tbs1=tbs;
BuffTableGen(N, n1); //generate buffer table extracting samples
}
if (opmode>1)
{// operation mode= 2 or 3?
tbsCalc(freq1, n1); //calculate separation between samples, signal 1
tbs1=tbs;
TempTable1Calc(N, n1); //generate temp table for signal 1
tbsCalc(freq2, n2); //calculate separation between samples, signal 2
tbs2=tbs;
TempTable2Calc(N, n2); // generate temp table for signal 2
BuffTableSuperposition(); // generate buffer table modes 2 & 3
}
WriteToOut(); //write to output port
return 0 ;
```

```
}
```

A.2 Application Program: Extended Version

The extended version of the application program has been developed for the board based prototype implementation. It has added functionality compared to the SoC based design. Additional functions were defined and implemented according to experimental needs and developing research work in the particle manipulation area.

```
// Uses Luminary Driverlib for parallel port use
// Version date: Feb the 3rd, 2011
// Details: separates frequency ranges in low (<400 Hz) and high (>400Hz)
// Delivers superimposed frequencies in any mix of available waveforms
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "hw_memmap.h"
#include "hw_types.h"
#include "sysctl.h"
#include "hw_sysctl.h"
#include "gpio.h"
#include "hw_gpio.h"
```

```
//#define PORT_DATA(GPIO_PIN_0 |GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3 | //GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7)

#ifdef DEBUG

void

__error__(char *pcFilename, unsigned long ulLine)

{

}

#endif

#define Pi 3.14159265358979323846264338327

static unsigned int sinedatint[256] =

{ 127, 130, 133, 136, 139, 142, 145, 148, 151, 154, 157, 160, 163, 166, 169, 172, 175, 178, 181, 184, 186, 189, 192, 194, 197, 200, 202, 205, 207, 209, 212, 214, 216, 218, 221, 223, 225, 227, 229, 230, 232, 234, 235, 237, 239, 240, 241, 243, 244, 245, 246, 247, 248, 249, 250, 250, 251, 252, 252, 253, 253, 253, 253, 253, 254, 253, 253, 253, 253, 253, 252, 252, 251, 250, 250, 249, 248, 247, 246, 245, 244, 243, 241, 240, 239, 237, 235, 234, 232, 230, 229, 227, 225, 223, 221, 218, 216, 214, 212, 209, 207, 205, 202, 200, 197, 194, 192, 189, 186, 184, 181, 178, 175, 172, 169, 166, 163, 160, 157, 154, 151, 148, 145, 142, 139, 136, 133, 130, 127, 123, 120, 117, 114, 111, 108, 105, 102, 99, 96, 93, 90, 87, 84, 81, 78, 75, 72, 69, 67, 64, 61, 59, 56, 53, 51, 48, 46, 44, 41, 39, 37, 35, 32, 30, 28, 26, 24, 23, 21, 19, 18, 16, 14, 13, 12, 10, 9, 8, 7, 6, 5, 4, 3, 3, 2, 1, 1, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 1, 1, 2, 3, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16, 18, 19, 21, 23, 24, 26, 28, 30, 32, 35, 37, 39, 41, 44, 46, 48, 51, 53, 56, 59, 61, 64, 67, 69, 72, 75,78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 114, 117, 120, 123 };

static unsigned int toothsawdat[256] =
```

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 142, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 252, 252, 253, 254, 255 };

static unsigned int triangdat[256] =

{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190, 192, 194, 196, 198, 200, 202, 204, 206, 208, 210, 212, 214, 216, 218, 220, 222, 224, 226, 228, 230, 232, 234, 236, 238, 240, 242, 244, 246, 248, 250, 252, 254, 255, 254, 252, 250, 248, 246, 244, 242, 240, 238, 236, 234, 232, 230, 228, 226, 224, 222, 220, 218, 216, 214, 212, 210, 208, 206, 204, 202, 200, 198, 196, 194, 192, 190, 188, 186, 184, 182, 180, 178, 176, 174, 172, 170, 168, 166, 164, 162, 160, 158, 156, 154, 152, 150, 148, 146, 144, 142, 140, 138, 136, 134, 132, 130, 128, 126, 124, 122, 120, 118, 116, 114, 112, 110, 108, 106, 104, 102, 100, 98, 96, 94, 92, 90, 88, 86, 84, 82, 80, 78, 76, 74, 72, 70, 68, 66, 64, 62, 60, 58, 56, 54, 52, 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4, 2 };

unsigned int timeindex1int[256], timeindex2int[256], timeindexbuffint[256];

```
    unsigned int TempTable1int[256], TempTable2int[256]; // Temporary tables,
//modes 2 & 3, scale 0 to 255

    unsigned int BuffTable[4096]; // output table, all modes

    double trunc(double arg);

    float tbs, tbs1, tbs2; // time between samples, para senal 1 y 2

    int dat_samples_buff1, dat_samples_temp1, dat_samples_temp2; //data
samples

    int opmode; //operation modes: 1 single signal, 2 superposition, 3 separate
//signals

    int signaltype, signaltype2; //signal type: 1 sine, 2 saw-tooth, 3 triangle

    float freq1, freq2; //frecuency for output signals 1 & 2

    int N, n1, n2; //samples per waveform cycle

    void GetOperParam()

    {

    //printf ("Operation mode: 1, 2 or 3:\n ");

    // scanf ("%d", &opmode);

    opmode=2;

    //printf ("Selected Operation mode = %d\n", opmode);

    //printf ("Signal type, 1 sine, 2 tooth, 3 triang:\n ");

    //scanf ("%d", &signaltype);

    signaltype2=2;

    signaltype=2;
```

```
//printf ("Selected signal type = %d\n", signaltype);
//printf ("Output single/low frequency in KiloHertz:\n ");
//scanf ("%f", &freq1);
freq1=500;
//printf ("Samples per cycle: \n");
//scanf ("%d", &n1);
//if (freq1>399) {
//n1=240000/freq1; //write to port, high frequencies
//n2=16;
//}
//if (freq1<400) {
//n1=64;
//n2=64; //write to port, low frequencies
//}
n1=64;
if (n1<9) n1=16;
if ((n1>8) & (n1< 17)) n1=16;
if ((n1>16) & (n1< 33)) n1=32;
if ((n1>32) & (n1< 65)) n1=64;
if ((n1>64) & (n1< 129)) n1=128;
```

```
if (n1>128) n1=256;
//printf(" %d\n", n1);
if (opmode>1)
{
//printf ("Output high frequency2 in KiloHertz:\n ");
//scanf ("%f", &freq2);
freq2=5000;
//printf ("Samples per cycle 2: \n");
//scanf ("%d", &n2);
n2=64;
if (n2<9) n2=8;
if ((n2>8) & (n2< 17)) n2=16;
if ((n2>16) & (n2< 33)) n2=32;
if ((n2>32) & (n2< 65)) n2=64;
if ((n2>64) & (n2< 129)) n2=128;
if (n2>128) n2=256;
//printf(" %d\n", n2);
//printf ("Leaving function GetOperParam\n");
}
}
```

```
void SineDisplay(int N)
{
unsigned int iter;
// printf ("Entering function SineDisplay ORIGINAL SINE TABLE\n");
for (iter = 0; iter < N; iter++)
{
//printf("[%d] ", iter);
//printf(" %d\n", sinedatint[iter]);
}
}
void BuffTableGen(int N, int n)
{
unsigned int dsepi;
double dsepd;
unsigned int iter;
unsigned int i;
unsigned int j;
//printf ("Entering function BuffTableGen mode 1\n");
i=0;
dsepd=N/n;
```

```
dsepi=dsepd;

j=dsepi;

// printf("data separation int, modo1 = %d\n ", dsepi);

for (iter = 0; iter < N; iter=iter+dsepi)

{

// /*printf("Data number = %d\n ", iter);*/

if (signaltype == 1) BuffTable[i]=sinedatint[iter];

if (signaltype == 2) BuffTable[i]=toothsawdat[iter];

if (signaltype == 3) BuffTable[i]=triangdat[iter];

// /*printf("Dato original = %.6ef\n ", sinedat[iter]);*/

// printf("%d\n ", BuffTable[i]);

dat_samples_buff1=i;

i=i+1;

}

// printf("Amount of data samples in buffer table = %d\n ", //dat_samples_buff1);

//printf ("Leaving function BuffTableGen, mode 1\n");

}

void TempTable1Calc(int N, int n1)//prepare temp table, signal 1, modes 2&3

{
```

```
unsigned int dsepi;

double dsepd;

unsigned int iter;

unsigned int i;

float t;

// printf ("Entering function TempTable1Calc, modes 2 & 3\n");

i=0;

dsepd=N/n1;

dsepi=dsepd;

if ((dsepd-dsepi)>0.495)

{dsepi++; //if separation es >. 495 round up to next integer

// printf ("separation table 1 %d \n", dsepi);

}

//printf("data separation in TEMPORARY TABLE 1 = %d\n ", dsepi);

//printf ("it time out data\n");

for (iter = 0; iter < N+1; iter=iter+dsepi)

{

t=tbs1*i;

// printf("[%d] >", iter);

//printf("[%d] ", i);
```

```
//printf("%.3ef ", t);
if (signaltype==1) TempTable1int[i]=sinedatint[iter];
if (signaltype==2) TempTable1int[i]=toothsawdat[iter];
if (signaltype==3) TempTable1int[i]=triangdat[iter];
timeindex1int[i]=100000*t;
//printf("t=%d ", timeindex1int[i]);
//printf("%d ", TempTable1int[i]);
//printf("%x\n ", TempTable1int[i]);
dat_samples_temp1=i;
i=i+1;
}
//printf("Data samples in temporary table1 = %d\n ", (dat_samples_temp1+1));
//printf ("Leaving function TempTable1Calc, modes 2 & 3\n");
}
void TempTable2Calc(int N, int n2)//prepare temp table signal 2, modes 2&3
{
unsigned int dsepi;
double dsepd;
unsigned int iter;
```

```
unsigned int i;
float t;
//printf ("Entering function TempTable2Calc, modes 2 & 3\n");
i=0;
dsepd=N/n2;
//printf(" %.2ef ", dsepd);
dsepi=dsepd;
if ((dsepd-dsepi)>0.495)
{dsepi++;
// printf ("separation table 2 %d \n", dsepi);
}
//printf("data separation in TEMPORARY TABLE 2 = %d\n ", dsepi);
//printf ("it time out data\n");
for (iter = 0; iter < N; iter=iter+dsepi)
{
t=tbs2*i;
// printf("[%d] >", iter);
//printf("[%d] ", i);
//printf("%.3ef ", t);
if (signaltype2==1) TempTable2int[i]=sinedatint[iter];
```

```
if (signaltype2==2) TempTable2int[i]=toothsawdat[iter];
if (signaltype2==3) TempTable2int[i]=triangdat[iter];
timeindex2int[i]=100000*t;
//printf("t=%d ", timeindex2int[i]);
//printf("%d ", TempTable2int[i]);
//printf("%x\n ", TempTable2int[i]);
dat_samples_temp2=i;
i=i+1;
}
}
void BuffTableSuperposition()////prepare output table mode 2
{
unsigned int i, j, k, l, m, dato1, dato2;
float t, tbsmin, tmax;
unsigned int tint, auxt1, auxt2;
//printf ("Entering function BuffTableSuperposition, mode 2\n");
if (tbs1<tbs2)
tbsmin=tbs1;
else
tbsmin=tbs2;
```

```
if (freq1<freq2)
tmax=1/freq1;
else
tmax=1/freq2;
t=0; i=0; j=0; k=0;
dato1=TempTable1int[i];
dato2=TempTable2int[i];
BuffTable[k]=(dato1+dato2)/2;
//printf("%d ", k);
//printf(" %.2ef ", t);
//printf(" %x ", data2);
//printf("+ %x ", data1);
//printf("= %d ", BuffTable[k]);
//printf("= %x\n ", BuffTable[k]);
do{
t=t+tbsmin;
tint=t*100000+1;
auxt1=timeindex1int[i+1];
auxt2=timeindex2int[j+1];
m=0;
```

```
if (tint<auxt1)
l=0;
else{
m=1;
i=i+1;
//printf("new data table111, index %d \n", i);
dato1=TempTable1int[i];
}
//printf("m value after checking table1 %d \n", m);
if (tint<auxt2)
l=0;
else{
m=2;
j=j+1;
if (j==n2) j=0; //returns to beginning of temp table, low frequency
dato2=TempTable2int[j];
}
if(m>0)
{
k++;
```

```
if (opmode==2) BuffTable[k]=(dato1+dato2)/2;

if (opmode==3) BuffTable[k]=dato1+dato2*256;

timeindexbuffint[k]=tint;

// printf("[%d] ", k);

// printf("t= %d ", tint);

// printf(" %x ", dato2);

//printf("+ %x ", dato1);

//printf("= %d ", BuffTable[k]);

//printf("= %x\n ", BuffTable[k]);

}

}

while (t<tmax);

dat_samples_buff1=k;

//printf("Amount of data samples in BUFFER TABLE = %d\n", k);

}

void tbsCalc(float freq, float n)//calculates time between samples

{

// printf ("Entering function tbsCalc\n");

tbs=1/(freq*n);

}
```

```
void WriteToOutLow()//load data from buffer table, write to output port
{
unsigned int i;
unsigned int j;
unsigned int k;
//float t;
// printf ("Entering function WriteToOut\n");
// printf("Time running between samples buffer1 = %.2ef\n", tbs1);
//printf ("it time out data\n");
// delayed cycle for slow signal generation
k=11000/freq1; // k in inverse proportion of desired frequency
for (;;)
for (i = 0; i < dat_samples_buff1+1; i=i+1)
{
for (j = 0; j < k; j=j+1) {} // generates time between //samples for slow frequencies running k wait cycles between output updates
// printf("[%d] ", i);
// printf(" %d ", timeindexbuffint[i]);
// printf(" %x\n ", BuffTable[i]);
GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_0 |GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7, BuffTable[i]);
```

```
}
// printf ("Leaving function WriteToOut\n");
}
void WriteToOutHigh()//load data from buffer table, write to output port
{
unsigned int i;
//unsigned int j;
// printf("Time running between samples buffer1 = %.2ef\n", tbs1);
//printf ("it time out data\n");
for (;;)
for (i = 0; i < dat_samples_buff1+1; i=i+1)
{
//for (j = 0; j < tbs1*12; j=j+1)//con 1e+5 is time between //samples
// t is accumulated time in cycle, last values is period T of //waveform
//t=tbs1*i;
// printf("[%d] ", i);
// printf(" %d ", timeindexbuffint[i]);
// printf(" %x\n ", BuffTable[i]);
GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_0 |GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7, BuffTable[i]);
```

```
}
// printf ("Leaving function WriteToOut\n");
}
int main(void)
{
//
// If running on Rev A2 silicon, turn the LDO voltage up to 2.75V. //This is a workaround to allow the PLL to operate reliably.
//
if(DEVICE_IS_REVA2)
{
SysCtlLDOSet(SYSCTL_LDO_2_75V);
}
//
// Set the clocking to run directly from the crystal.
// Default values assume an external crystal of 6MHz. See Luminary
// driverlib documentation for other values.
// Clock source can be:
// 'SYSCTL_USE_OSC | SYSCTL_OSC_MAIN' - use the xtal without PLL
// 'SYSCTL_USE_PLL | SYSCTL_OSC_MAIN' - use the xtal with PLL
// If using the PLL, the oscillator runs at 200MHz and then you select
```

```
// a division of this frequency to clock the core. Otherwise the //divider just

//directly divides the XTAL frequency.

// Use a 6MHz external XTAL directly with no division

//printf ("Entering sysctlclokset \n");

SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN | SYSCTL_XTAL_6MHZ);

// Use the XTAL directly to clock the PLL with division by 4

//SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |

// SYSCTL_XTAL_6MHZ);

// printf ("Entering sysctlperipheralenable \n");

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

//printf ("Entering define gpio as output \n");

GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_0 |GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7);

//printf ("Entering write to port \n");

GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_0 |GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7,0x55);

// printf ("Leaving write to port \n");

// Use the XTAL directly to clock the PLL with division by 4
```

```
// SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
// SYSCTL_XTAL_6MHZ);
N=256;
GetOperParam(); //Get operation parameters
//SineDisplay(N); //Display data samples for sine waveform
if (opmode==1)// operation mode = 1?
{
tbsCalc(freq1, n1); //calculate separation between samples
tbs1=tbs;
BuffTableGen(N, n1); //generate buffer table extracting samples
}
if (opmode>1)
{// operation mode= 2 or 3?
tbsCalc(freq1, n1); //calculate separation between samples, signal 1
tbs1=tbs;
TempTable1Calc(N, n1); //generate temp table for signal 1
tbsCalc(freq2, n2); //calculate separation between samples, signal 2
tbs2=tbs;
TempTable2Calc(N, n2); // generate temp table for signal 2
BuffTableSuperposition(); // generate buffer table modes 2 & 3
```

```
}
if (freq1>399) WriteToOutHigh(); // write to output port, high //frequencies
if (freq1<400) WriteToOutLow(); //write to output port, low //frequencies
while(1);
```

Martha Salomé López de la Fuente works at Universidad de Monterrey, Mexico, as a research professor for the intelligent systems and robotics programs. Her contributions in the research and education areas relate to digital systems design, application specific integrated circuits, embedded systems applications, intelligent systems, and service robots. Since 1994 she has been teaching courses on microprocessors, digital electronics, embedded systems, integrated circuit design, and hardware architectures for service robots. Martha has presented her research work in forums such as Engineering in Medicine and Biology Conference, CERMA Electronics Robotics and Automotive Mechanics Conference, Eurasian Multidisciplinary Forum, and World Forum on Internet of Things. She has also published scientific articles in the IEEE Xplore digital library, the Review of Scientific Instruments journal, and the European Scientific Journal.

ISBN: 978-1-940366-44-9
9 781940 366449 >
Price: US $80

To order additional copies of this book, please contact:
Science Publishing Group
book@sciencepublishinggroup.com
www.sciencepublishinggroup.com