A LOW-POWER DSP ARCHITECTURE FOR A FULLY IMPLANTABLE COCHLEAR IMPLANT SYSTEM-ON-A-CHIP

by

Eric David Marsman

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in The University of Michigan
2012

Doctoral Committee:

Professor Richard B. Brown, Co-Chair, University of Utah
Professor Dennis M. Sylvester, Co-Chair
Professor Daryl R. Kipke
Professor Kensall D. Wise

© Eric David Marsman 2012
All Rights Reserved

For my amazing wife, Jennifer,
and our wonderful children, Victoria and Fiona.

ACKNOWLEDGEMENTS

I would like to thank God for all of the blessings and strength He has given me throughout my life. I thank Him for my family, opportunities, and gifts that I have received. I pray that my achievements and future accomplishments are pleasing to Him.

I would like to thank Dr. Richard Brown for his valuable advice and thoughtful insights. I am deeply grateful not only for his technical support and wisdom, but also his patience with my dissertation completion. He truly made my unorthodox path through graduate school possible with foresight, motivation, and understanding.

Along those lines, I would like to thank the rest of my dissertation committee. This work would certainly not have achieved the same level of quality without their feedback. In particular, Dr. Sylvester for co-chairing the committee and allowing me to be his GSI for multiple semesters. Thank you to Dr. Kipke and Dr. Wise for their time and valuable insights.

I'm also grateful to the rest of Dr. Brown's research group. In particular, I'd like to recognize Rob Senger, whom I worked very closely with on the WIMS project. He is an extremely hard worker that is always full of ideas and open for discussion. Thanks to Michael McCorquodale, Matt Guthaus, Fadi Gebara, and Steve Martin for their valuable contributions to the multiple versions of the WIMS MCU chips and advice throughout my time at Michigan. Alan Drake, Rahul Rao, Jay Sivagnaname, Chie-Wei Yew, Koushik Das, and Rob Franklin were also meaningful contributors to my time and experience in Dr. Brown's group. Lastly, Nathaniel Gaskin and Spencer Kellis for boldly carrying on the WIMS work at Utah. Additionally, Dr. Scott Mahlke, Rajiv Ravindran, and Ganesh Dasika are due gratitude for their development of the WIMS Compiler and contributions to the improvements in the ISA. Thanks also to Joe Potkay, Pamela Bhatti, and several other WIMS students I had the pleasure of working with.

The WIMS center and EECS department could not be successful without the many people working hard everyday to make it run smoothly. I cannot list them all here, but in particular I would like to thank: Dr. Wise and Dr. Najafi for their leadership of WIMS, Dr. Nayda Santiago and her students and UPRM, Catharine June, Paulette Ream, Karen Richardson, Bonnie Fox, Vicki Jensen, Beth Stalnaker, Gordy Carichner and the rest of the DCO staff, Robert Gordenker, and Brendan Casey.

Research is a difficult task without funding, and so I would like to thank the NSF for funding the WIMS ERC, MOSIS for fabrication of devices, and all the CAD tool and equipment vendors for contributing to educational institutions.

Early on in the project we made a valuable trip to visit Dr. Blake Wilson and Reinhold Schatzer at the Research Triangle Institute. Dr. Wilson is a pioneer in Cochlear Implants and his team down there took the time to show us around his lab and teach us all they could about implants. It was very kind of them to share their time and experience.

Mobius Microsystems has been an enjoyable place for me and so I would like to thank them for their work on WIMS as well as enjoyment of life after WIMS: Michael, Scott, Gordy, Justin, Sundus, Jon, and Nathaniel. Also, thanks to IDT for their funding of the continuing education program.

I'm extremely grateful to my parents, Carl and Mara. I was born and raised in Michigan and grew up watching Michigan sports. It is the culmination of a lifelong dream to receive this final degree from the University of Michigan, and it wouldn't have been possible without the emotional and financial support of my parents. I also want to thank my brother, Paul, for always pushing me and giving me the competitive drive to achieve all that I have. My mother- and father-in-law are due gratitude for all their hospitality while I was an out-of-town student "visiting" Ann Arbor. Judy and Bob's generosity is greatly appreciated.

Finally, thanks to my wonderful wife, Jennifer, for all that you do. You are an inspiration and have given me so much. You are a great partner and there is nothing that we can't do together. I also need to thank you for our wonderful children, Victoria and Fiona, who provide me happiness on a daily basis.

TABLE OF CONTENTS

Dedication ii

Acknowledgements iii

List of Tables viii

List of Figures ix

Chapter I
Introduction 1

1.1 WIMS Overview 2
1.1.1 Environmental Monitor Testbed 3
1.1.2 Cochlear Prosthesis Testbed 5
1.2 Dissertation Thesis Overview 6

Chapter II
Cochlear Implants 8

2.1 The Human Ear 9
2.2 Implant Electronics 11
2.2.1 Microphone 12
2.2.2 Signal Processor 12
2.2.3 Communication Interface 13
2.2.4 Electrodes 13
2.2.5 Batteries 14
2.3 Patient Fitting Procedure 15
2.4 Sound Processing Strategies 15
2.4.1 Waveform Strategies 16
2.4.2 Interleaved Strategies 17
2.4.3 Feature Extraction and Spectral Information Strategies 19
2.5 Implementation Methods 20
2.5.1 Software Programmable DSP 20
2.5.2 Analog Signal Processing 21
2.5.3 Integrated SoC 22
2.6 Cochlear Implant Improvements 23
2.7 Conclusions 23

Chapter III
WIMS Microsystem 24

3.1 Microsystem Architecture 24
3.1.1 Instruction Set Architecture 25
3.2 Microcontroller 26
3.2.1 Memory Architecture 26
3.2.2 Pipeline 28
3.2.3 Peripherals 29
3.2.4 Testability 30
3.2.5 Summary 31
3.3 Digital Signal Processor 31
3.3.1 Architecture 32
3.3.2 Modes of Operation 34
3.3.3 Interfaces 35
3.3.3.1 Microcontroller 35
3.3.3.2 Analog to Digital Converter 35
3.3.3.3 Electrode Array 36
3.3.4 Conclusions 36
3.4 Clocking Scheme 36
3.4.1 Dynamic Frequency Scaling 37
3.4.2 Mobius Microsystems' IP 40
3.4.3 Conclusions 42
3.5 Summary 42

Chapter IV
Microsystem Design Methodology 43

4.1 Design Methodology 43
4.1.1 Design Trends and Challenges with Microsystems Technology 44
4.1.2 Design Methodology Comparison 46
4.1.2.1 Typical Design Methodology: Bottom-Up 46
4.1.2.2 New Design Methodology: Top-Down 48
4.1.3 Top-Down Microsystem Design Flow 49
4.1.3.1 Hardware Design Flow 49
4.1.3.2 Software Design Flow 52
4.1.3.3 Digital Design and Verification 53
4.2 Conclusions 54

Chapter V
Microsystem Results 56

5.1 Microsystem Measured Results 56
5.1.1 Post-fabrication Testing 56
5.1.2 Microsystem Results 57
5.1.3 Microcontroller Results 59
5.1.3.1 Scratchpad Memory 59
5.1.4 Energy Per Instruction 59
5.1.4.1 Experimental Hardware Methodology 63
5.1.4.2 Hardware Energy Per Instruction Results 67
5.1.5 Digital Signal Processor 69

5.1.6 Dynamic Frequency Scaling 70
5.1.7 Performance Comparison 72
5.2 Conclusions 73

Chapter VI
Conclusion and Future Directions 75

6.1 Platform Comparison 75
6.2 System Demonstrations 76
6.3 Achievements 79
6.4 Future Research Topics 81
6.5 Conclusion 82

Appendix A. WIMS Microcontroller User Manual 84

Bibliography 156

LIST OF TABLES

Chapter II **8**

Table 2.1: Commercially available CI products and their available speech processing algorithms...16

Table 2.2: Summary of programmable DSP options. 21

Table 2.3: Summary of analog signal processing options. 22

Chapter III **24**

Table 3.1: ISA Summary. 27

Chapter V **56**

Table 5.1: Measured power for different operating conditions. 58

Table 5.2: Chip statistics breakdown. 59

Table 5.3: Energy per instruction. 68

Table 5.4: Energy memory correction factors. 69

Table 5.5: Comparison of commercially available cores with the Gen-2 WIMS MCU. 72

Table 5.6: Detailed comparison of Gen-2 WIMS core with ARM7TDMI. 73

Chapter VI **75**

Table 6.1: Comparison of Cochlear Implant signal processing platforms. 76

LIST OF FIGURES

Chapter I **1**

Figure 1.1: Concept diagram of WIMS µGC layout [9]. 4

Figure 1.2: Original µGC data (left); Compressed µGC data (right). 5

Figure 1.3: Concept drawing of the WIMS Cochlear Implant assembly [9]. 6

Chapter II **8**

Figure 2.1: Generic cochlear implant block diagram. 8

Figure 2.2: The human ear [14]. 10

Figure 2.3: Cross section through a cochlea turn [15]. 11

Figure 2.4: The basilar membrane frequency mapping. 12

Figure 2.5: Michigan cochlear electrode array [16]. 14

Figure 2.6: Block diagram of the Compressed Analog signal processing algorithm used in the Ineraid device by Symbion Inc. 16

Figure 2.7: Block diagram of an n channel CIS signal processing algorithm. 18

Chapter III **24**

Figure 3.1: Complete microsystem architecture. 25

Figure 3.2: Normalized power and area trade-offs for possible memory configurations. 28

Figure 3.3: Pipeline block diagram. 29

Figure 3.4: CIS DSP architecture block diagram. 33

Figure 3.5: HDL synthesizable glitch-free dynamic frequency controller [57]. 38

Figure 3.6: Glitch suppression on f_{MCU} during frequency transitions. 39

Figure 3.7: Self-referenced hybrid clock synthesizer. 41

Chapter IV **43**

Figure 4.1: The anatomy of a generalized wireless integrated microsystem. Key technologies and associated development tools are shown. 44

Figure 4.2: Typical ad-hoc and bottom-up microsystems design methodology. 47

Figure 4.3: The top-down microsystems design methodology. 50

Figure 4.4: Matlab Simulink DSP model input waveform. 54

Figure 4.5: Matlab Simulink DSP model channel seven output waveform. 54

Chapter V 56

Figure 5.1: Die micrograph of fabricated microsystem. 57

Figure 5.2: HP82000 test setup. 58

Figure 5.3: Power versus VDD scaling for the MCU and Memory. 60

Figure 5.4: Power savings utilizing scratchpad memory or loop cache. 61

Figure 5.5: Sample assembly loop for energy per instruction measurements. 64

Figure 5.5: Power versus VDD scaling for the DSP. 70

Figure 5.6: Oscilloscope traces showing low-latency dynamic frequency scaling of (top) the TC-LCO clock and (bottom) the TC-LCO and ring clock. 71

Chapter VI 75

Figure 6.1: WIMS Environmental Testbed demo board. 77

Figure 6.2: Complete assembly of the WIMS Cochlear Prosthesis demonstration. 77

Figure 6.3: University of Utah Cochlear Demonstration Board. 78

Figure 6.4: Logic Analyzer traces of ADC (SPI1) and Electrode Array (SPI0) communicating with the MCU. 79

CHAPTER I

INTRODUCTION

Deafness affects a large number of people throughout the world. Studies from the Center for Disease Control (CDC) National Health Interview Survey (NHIS) show that as of 2005, over 600,000 people across all age groups suffer from deafness in the U.S. alone [1]. The combined efforts of a countless number of scientists from the fields of otolaryngology, physiology, bioengineering, signal processing, and integrated circuit (IC) technology have enabled persons suffering from deafness to receive a cochlear implant (CI). These devices allow patients to have a good understanding of speech without the aid of other techniques, such as lip-reading. The first CI device was approved by the Food and Drug Administration (FDA) in 1984 [2] and many improvements have been made since then. As of 2010, according to the National Institute on Deafness and Other Communication Disorders (NIDCD), there are over 219,000 CIs currently implanted worldwide including 42,600 adults and 28,400 children in the U.S. [3]. This document describes a further improvement to existing implant technology by including all of the computational and communication capability required for a CI in a system-on-a-chip (SoC) to enable a fully-implantable Cochlear Prosthesis.

Performance improvements and technology scaling have allowed the IC industry to create devices and systems unimaginable to previous generations. In the recent past, most design effort and improvement have been used to develop high performance systems. However, in an ever more mobile society, low-power portable devices are moving to the forefront of research and design efforts. These new efforts have allowed the design and manufacturing of SoCs that enable the integration of digital, analog, mechanical, biologi-

cal, chemical, and other design domains. The CI industry has yet to combine all of the functionality required into a single chip. The research discussed here will enable that transition by integrating the digital signal processing (DSP) functionality required into an SoC. This work is part of research conducted as part of the National Science Foundation (NSF) Engineering Research Center (ERC) for Wireless Integrated Microsystems (WIMS) for the cochlear prostheses testbed [4].

Increased integration of components into a single SoC will enable performance improvements in remote sensing and biomedical applications. The microsystem described here combines an energy efficient microcontroller unit (MCU), a low-power DSP core, and a monolithic hybrid LC (inducutor-capacitor) and ring oscillator clock reference into a single bulk CMOS SoC intended for use in CIs. By merging all of these components on a single substrate, the area and power consumption of the system can be greatly reduced without sacrificing performance compared to present CI systems. While this is a significant step towards a fully-implantable CI system, there is still further integration to be done. This work will not include an analog to digital converter (ADC) or a radio frequency (RF) communication interface on a single silicon substrate. However, the system presented here will support interfaces for them and demonstration vehicles will show the full system in operation with the ADC and RF interface as separate discrete components. This move towards a fully-implantable CI is not possible with traditional CI approaches due to size and power requirements. Although this work was primarily targeted towards integration into a CI, the same SoC platform or standalone MCU was utilized in remote environmental sensors [5] at the University of Michigan as part of the WIMS ERC.

1.1 WIMS Overview

WIMS represent the cutting edge of ultra low-power, embedded sensor-system research. The WIMS Center at the University of Michigan has developed two testbeds to demonstrate and integrate different aspects of WIMS technology that are being developed by various research groups. These testbeds bridge the gap between individual research ideas and system-level implementation of those ideas. Each testbed requires a microcontroller for control and data processing. However, performance and interfacing requirements between the two testbeds vary slightly. The MCU is the most active component in the

system so power must be kept to an absolute minimum to sustain sufficient battery life, while providing sufficient processing power as required by the testbeds.

A goal of the WIMS project is to develop low-power electronic components that could be used in a wide variety of microsystems. The WIMS testbeds fall into two categories: remote environmental sensors and biomedical implants. They will be described briefly in the next sections to give the reader a feeling for the breadth of applications that have been addressed. The WIMS microcontroller was designed with these target applications in mind. Three design generations (Gen-0, Gen-1, and Gen-2) of the WIMS MCU were designed and tested. Gen-0 was a platform test vehicle and will not be discussed here. Gen-1 will be described in certain situations as it was used heavily in the environmental testbed evaluation. Gen-2 is the focus of the work described here and contains several improvements over the Gen-1 prototype [6] - [8]. Unless otherwise stated, all data and figures discuss the Gen-2 system.

1.1.1 Environmental Monitor Testbed

The original detailed goals of the Environmental Testbed are described in [9] and are summarized here. The differentiation and determination of complex mixtures of toxic gases and vapors remains a challenging analytical problem of significant importance in assessing human exposures, monitoring industrial emissions, biomedical surveillance and diagnosis, and homeland security. To address this challenge, the WIMS ERC launched the Environmental Monitor Testbed to develop a wireless, Micro Gas Chromatograph (μGC) using Micro-Electro-Mechanical Systems (MEMS). The μGC, shown in Fig. 1.1, is comprised of a sample inlet with particulate filter, on-board calibration-vapor source, multi-stage pre-concentrator and focuser, dual-column separation module with pressure- and temperature-programmed separation tuning, an array of microsensors for analyte recognition and quantification, system pressure and temperature sensors, and a pump and valves to direct sample flow. The goal is a fully operational microinstrument that occupies forty cubic centimeters, operates on a few milliwatt average battery power, provides simultaneous determinations of approximately thirty vapors at low- or sub-part-per-billion (ppb) levels in a few minutes, has an embedded controller, and can be remotely controlled and monitored through a wireless communication link. In early tests of the μGC, the design

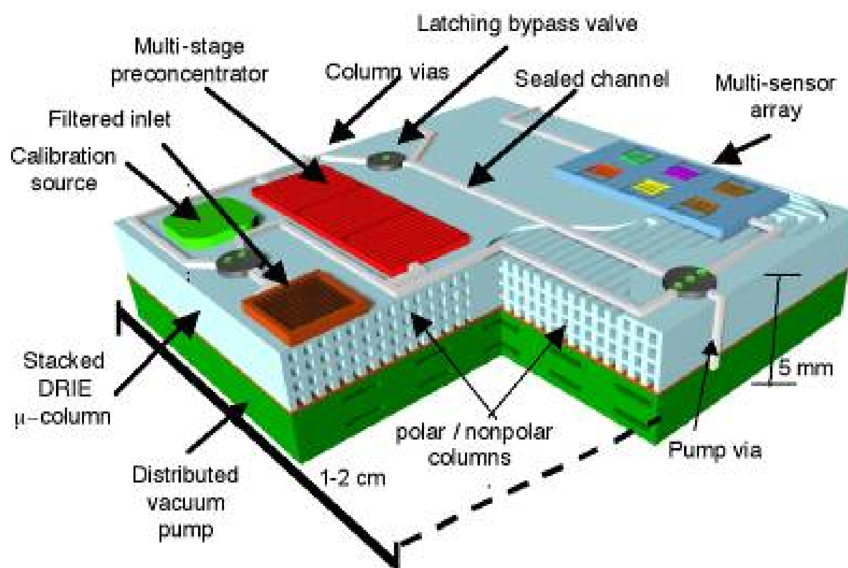


Figure 1.1: Concept diagram of WIMS μGC layout [9].

team determined the components of mixtures of more than ten vapors in less than twelve minutes with low-ppb detection limits [5].

The Gen-1 WIMS MCU was designed to provide all control and data processing required by the μGC system while minimizing the system's total power dissipation. There was a delicate balance between processing and compressing μGC data on-chip versus transmitting that data over the wireless link to a host computer for processing and analysis. Unfortunately, the WIMS Gen-1 MCU was never interfaced with the final μGC system due to the timing of completion of the different projects. However, progress was made in the software development of the μGC software using the Gen-1 MCU.

With the help of WIMS researchers under Professor Nayda Santiago at the University of Puerto Rico in Mayaguez (UPRM), software algorithms were developed to extract and compress critical data points that appear as spikes or peaks in the μGC's voltage output. Results show that up to 36x data compression is possible using UPRM's peak detection algorithm [10]. Fig. 1.2 shows the original μGC data on the left with the compressed μGC data on the right. What were originally 4,341 data points have been compressed into 120 points while maintaining important peak characteristics such as height, width, and area. Although the peak detection algorithm is a software optimization, it affects the hardware design because the level of data compression determines the amount of SRAM required for

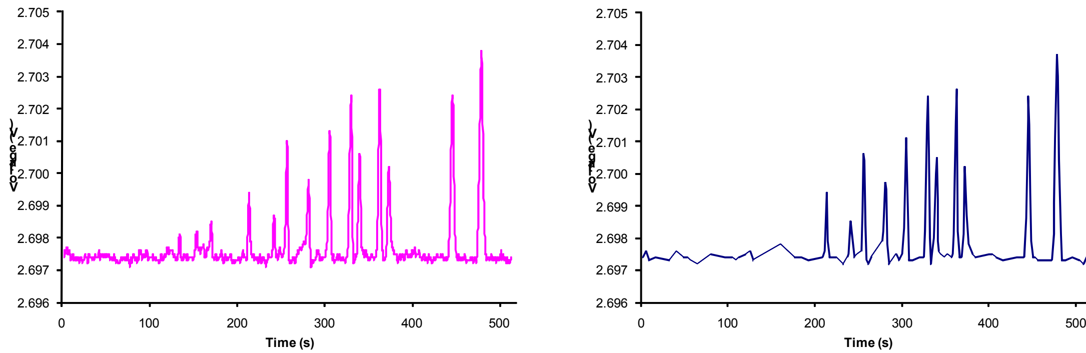


Figure 1.2: Original μGC data (left); Compressed μGC data (right).

data storage on the MCU. By reducing SRAM size, memory power and silicon area are minimized. From a system perspective, data compression also reduces wireless transmitter power because less data needs to be transmitted. This is just one example of how hardware and software co-design is important for minimizing system power. Ideally, the MCU would accomplish all data processing on-chip by performing both peak detection and component analysis so that only the final results are transmitted to the host computer.

1.1.2 Cochlear Prosthesis Testbed

The Cochlear Testbed is described in [9] and summarized here. Improving medical interfaces to the human body promises significant enhancements in quality of life for millions of people by providing diagnostics and therapies to improve function impaired by deafness, paralysis, seizures, blindness, and more. Realizable biomedical devices require meaningful technological advances on several fronts before they reach their full clinical potential. The WIMS approach brings together key components by developing smart neural microprobes and wireless, low-power, reconfigurable microsystems. The Cochlear Prosthesis Testbed integrates a high-density electrode array, a surgeon controllable articulated positioner, on-board circuit self-diagnostics, embedded MCU, and a bidirectional wireless link. The objectives of the prototype, shown in Fig. 1.3, are to demonstrate a fully implantable auditory prosthesis with untethered communication using rechargeable thin film batteries. Tests with WIMS probes demonstrate the ability to provide position feedback to allow precise positioning of electrodes.

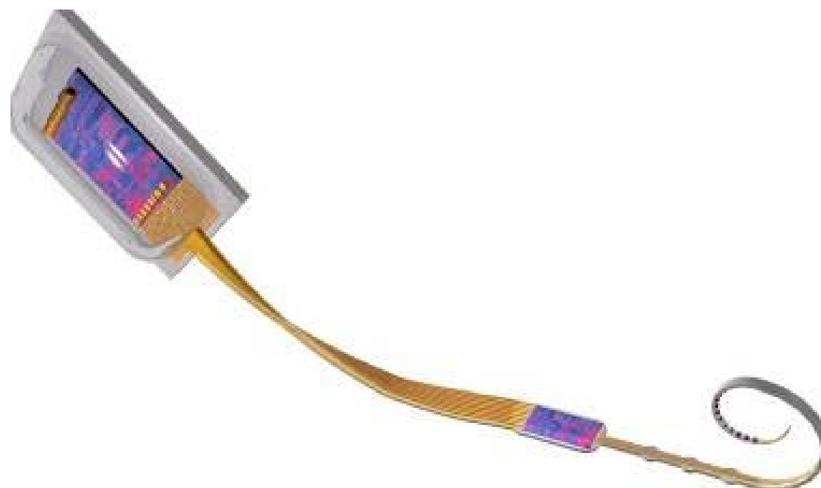


Figure 1.3: Concept drawing of the WIMS Cochlear Implant assembly [9].

As in the Environmental Testbed, the WIMS MCU serves as the primary control module for the cochlear implant system. A low-power DSP core was integrated with the Gen-2 WIMS MCU to provide the data processing capability required by the Continuous Interleaved Sampling (CIS) CI speech processing algorithm (see Section 2.4). The CIS DSP processes microphone data without assistance from the MCU, allowing the microcontroller to perform other tasks, or enter into sleep mode. Integrating the CIS DSP with the WIMS MCU allows the DSP to take advantage of the existing peripheral communication circuitry on the MCU [11].

1.2 Dissertation Thesis Overview

This introductory chapter gives a brief overview of this dissertation. Chapter II gives background on the physiology of the human ear and describes the function and components of state-of-the-art Cochlear Implants including various signal processing algorithms, including CIS. Chapter III describes the WIMS Microcontroller in great detail. The MCU instruction set architecture is outlined including motivation for many power saving features. Peripheral components are described including communication interfaces, memory architecture, and clock generation and distribution. The DSP implementation is reported as a separate subsystem. Chapter IV gives the mixed signal design methodology used to create the MCU. Silicon measured results are given for all previously discussed components in Chapter V. Chapter VI compares the WIMS CI to other commercial and

academic CI systems and describes the achievements of the WIMS Cochlear Prosthesis testbed demonstration vehicle. Finally, future research direction ideas are presented. The user manual for the WIMS MCU is give in Appendix A as a reference.

CHAPTER II

COCHLEAR IMPLANTS

CIs are medical devices implanted in persons who have a degradation or absence of sensory hair cells in the inner ear. In a properly functioning ear, the hair cells in the inner ear generate electrical signals that can be sensed by the neurons and sent to the brain for processing. The purpose of a CI is to perform the function of the hair cells if they are not working properly. Fig. 2.1 shows a schematic diagram of a generic CI and the intercommunication of the different components. A microphone receives the sound and converts it to electrical energy where the automatic gain control (AGC) compresses the dynamic range of the signal. Next, a speech processor performs computations on the signal to generate the cochlear stimulation profile. Data is transmitted through a radio frequency (RF) transcutaneous link to the internal electronics where it is received and processed. Electrical current is delivered to the cochlea to perform the nerve stimulation.

Two types of CIs available today: body-worn (BW) and behind-the-ear (BTE). In a BW device, the batteries and speech processor are worn on the waist, torso, or elsewhere in a pager-size housing. Cables run between the microphone, which is worn behind the ear, and the speech processor. A small cable also runs from the speech processor to the trans-

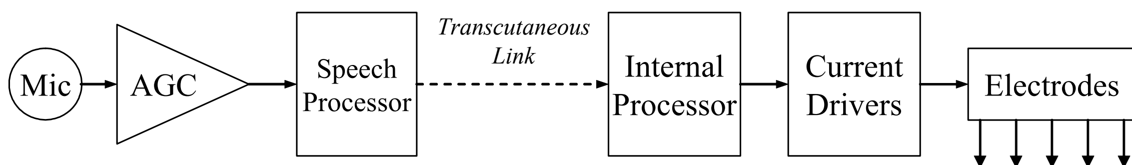


Figure 2.1: Generic cochlear implant block diagram.

mitter, which is located above and behind the ear. The receiver and electrodes are implanted inside the patient's inner ear.

With advances in IC and electronic technology, the state of the art is moving towards BTE devices. In a BTE device, the battery and speech processor are moved from the body-worn pack into the BTE housing along with the microphone. The transmitter, receiver, and electrodes remain unaffected. BTEs have the advantage of being less cumbersome for the active patient. However, BW implants will continue to be available because many children find them more comfortable and a hip-worn pack is more concealable in some situations, and therefore, preferred by some adults [2].

CIs also differ in the number of channels that the sound information is divided into before the signal processing begins. One of the first CIs was a single-channel implant called the House/3M implant and was first implanted in the 1970's [12]. This device contained only one band of signal information and only one electrode was implanted in the cochlea. Today, all implants contain multiple channels of signal information and multiple electrodes are used for stimulation. Researchers still debate which are the best methods for processing the sound and driving the electrodes, and what is the optimal number of electrodes.

A majority of patients with cochlear implants today receive a CI in only one ear. As our understanding of implants and how they are received by patients has grown in recent years, much research has gone into bilateral implants: having implant circuitry and processors for both ears. Many effects of a bilateral implant are still being studied and will not be discussed here, but bilateral implantation is a promising approach that appears to provide better speech reception and sound localization abilities for many of its users [13].

The next sections give more detail on the human ear and the workings of each of the components that make up CI electronics.

2.1 The Human Ear

The human ear consists of the three main sections shown in Fig. 2.2: the outer, middle, and inner ear. The outer ear is composed of the auricle and auditory canal. Their function is to funnel sound into the middle ear region, made up of the eardrum and ossicles. The eardrum converts the pressure wave from the auditory canal into vibrations which the oss-

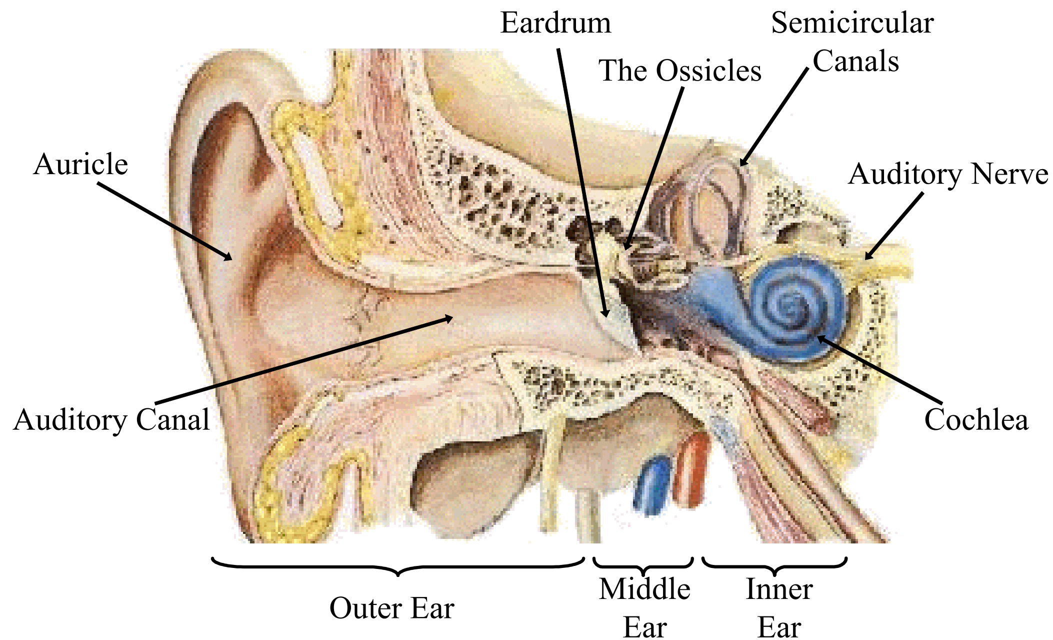


Figure 2.2: The human ear [14].

icles then amplify. The ossicles are three interconnected bones (hammer, anvil, stirrup) which amplify the vibrations of the eardrum and deliver them to the liquid-filled inner ear. The inner ear contains the cochlea, semicircular canals, and auditory nerve. The semicircular canals are not involved in the perception of sound, but rather, they act like accelerometers and assist in balance. Fig. 2.3 shows a cross section through a turn of the cochlea. The cochlea is a helical-shaped liquid-filled organ made up of the scala vestibuli (SV) at the apex, scala media (SM) or cochlear duct in the middle, and scala tympani (ST) at the base (opening). Reissner's membrane separates the SV and SM and the basilar membrane separates the SM and ST. The basilar membrane, shown in Fig. 2.4, contains the hair cells that convert the mechanical energy into electrical energy for the neurons. Each hair cell is sensitive to particular frequencies; the hair cells are arranged logarithmically by pitch in the basilar membrane with the lower tones located at the apex and the higher tones located at the base. The function of the frequency-sensitive hair cells is to release a small electrical current, corresponding to the vibrations they receive, which is then sent to the brain through the auditory nerve.

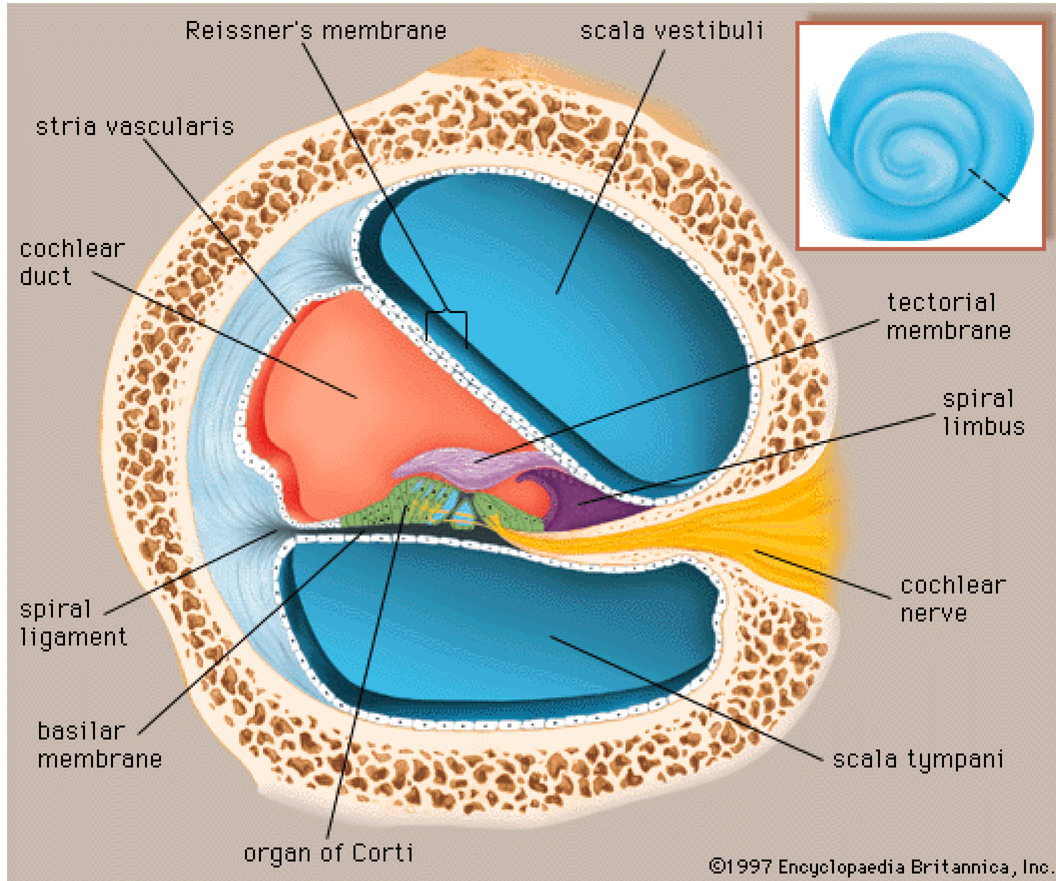


Figure 2.3: Cross section through a cochlea turn [15].

2.2 Implant Electronics

CIs are made up of the following major components: microphone, speech processor, receiver and transmitter for the transcutaneous link, electrodes, and batteries. In various CIs, these components are implemented in vastly different ways. This section gives a brief description of the components and their operation in the system that is generic enough to apply to any CI implementation. It also describes the more common variations. However, because a design decision made for one component can greatly affect the capabilities and requirements for another component, this is not intended to be an exhaustive discussion of all the possible CI configurations.

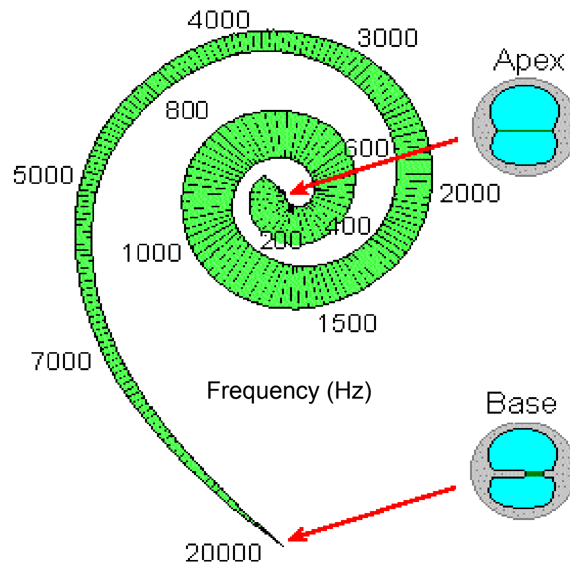


Figure 2.4: The basilar membrane frequency mapping.

2.2.1 Microphone

The audible range for humans is usually considered 20Hz to 20KHz, but most CIs concentrate on frequencies between 300Hz and 10KHz due to electrode limitations and other factors. These electrode limitations will be discussed further in Section 2.2.4. A microphone with a wide input dynamic range is required. Also, because most implants are monolateral, a broad directional microphone is best suited for CIs. Some implants available today have optional focused directional microphones that can be plugged into the implant in order to aid the patient in receiving sounds from one person when in a crowded room, or when there is a lot of background noise.

2.2.2 Signal Processor

The purpose of the signal processor is to convert the electrical signals from the microphone into waveforms that can drive current into the electrodes in a manner that enables the patient to perceive the sounds like a person with normal hearing. This is a complex process that is the focus of numerous researchers. New speech processing algorithms are being developed and tested at a phenomenal rate. The speech processing algorithms that

have been successful to date, along with a few of the more promising ones on the horizon, will be discussed in Section 2.4

2.2.3 Communication Interface

In a conventional CI, the receiver and transmitter perform the task of getting the information and power from the speech processor into the electrodes inside the cochlea. The information transmitted across this link can vary depending upon the type of algorithm the CI is running. Also, most CIs allow for a bidirectional link in which information about electrode performance can be fed back from the electrodes so it can be obtained by an audiologist. Typically, the information transmitted to the electrodes includes an address for the electrode site(s), the amplitude of the pulse, and the pulse duration. Pulse rate is another important variable; depending on the capabilities of the electrodes and the transmission procedures, pulse rate may be sent as data or be determined by the frequency of transmissions.

CIs with transcutaneous communication links have no physical wire connecting the internal and external components whereas devices using percutaneous links use a wire running through the skin to connect the internal and external components. Transcutaneous links are limited by the capabilities of the RF transmission link. Percutaneous links offer much better signal integrity and put fewer limits on the type of stimuli sent to the electrodes. While these are useful in testing new algorithms or electrodes, patients and doctors prefer to have transcutaneous links for aesthetic and health reasons. Transcutaneous links are utilized in all commercial CIs today.

2.2.4 Electrodes

The electrodes and electrode carrier are implanted inside the cochlea by a surgeon. They must be bio-compatible and mechanically stable in order to sustain vibrations from head movement. Their purpose is to stimulate the cochlea with electrical current. Electrode stimulation can happen with either an analog or pulsatile signal, depending on the type of speech processing algorithm used. Pulsatile stimulation can happen in one of several ways: monopolar, bipolar, or dual electrode configurations. Monopolar, or common ground, stim-

ulation fires one electrode at a time with reference to a common ground electrode for any stimulating site. In bipolar stimulation, each firing electrode has a nearby ground electrode to provide a shorter path for current and a more localized stimulation. For dual electrode stimulation, two electrodes in close proximity are fired simultaneously in order to stimulate nerves around both electrodes. The choice of stimulation methodology depends on the speech processing algorithm used, the electrode array capabilities, and the specific patient's remaining hearing function. Newer CI systems are being made with multiple options.

Commercial electrode arrays are presently limited to between eight and thirty-one electrodes. Increasing the density of the electrodes in the array is one area that is being investigated in order to improve CI performance [16]. The Michigan Cochlear Electrode Array is a 128-site array. A 32-site prototype array is shown in Fig. 2.5. Depending on the speech processing algorithm, not all of the electrodes are used for stimulation at all times. In addition, some electrodes may be useless to the patient for a variety of reasons. These electrodes are detected and noted as unusable during the CI fitting procedure. The implants are then programmed to not use these electrodes for stimulation purposes.

2.2.5 Batteries

As with any low-power system, the battery is one of the key elements as it determines the amount of time the system is able to sustain functionality. For a BTE implant, a standard hearing aid battery is suitable. For BW implants, standard AA batteries are used. Typical battery lifetimes in a CI range from nine hours to three to five days in today's systems [18], [19].

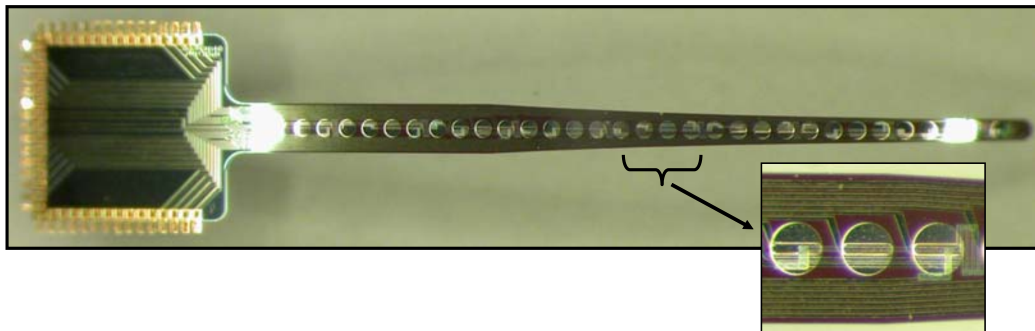


Figure 2.5: Michigan cochlear electrode array [16].

2.3 Patient Fitting Procedure

Upon receiving an implant, the patient must undergo a fitting procedure with an audiologist in order to obtain the best performance with their CI. This procedure takes place four to six weeks after the implant surgery to allow for healing of the inner ear. The fitting procedure consists of the audiologist turning on each electrode and the patient responding when they perceive sound, to establish the minimum current level for each electrode. The audiologist also finds the maximum comfortable current level for each electrode. This enables the audiologist to program the device to use only electrodes that have good contact to underlying nerves and to balance the sound between electrodes representing different frequencies. The speech processor has many features that can be programmed specifically for the patient [17]. Several of these features will be discussed in the following sections.

The combination of having numerous attributes available for programming and the high degree of variability from patient to patient makes for a sometimes lengthy fitting procedure. Often, patients will come back several times in the first few months in order to modify settings. Checkups are performed annually, or bi-annually, after the first satisfactory settings are found.

2.4 Sound Processing Strategies

Most of the programmable features discussed previously occur in the speech processing algorithm and, apart from the electrode to cochlea interface, the speech processor has the greatest potential to either improve or hinder the patient's speech perception performance. A brief overview of several of the possible strategies are presented here, with emphasis on the newer algorithms that are common in CIs today, along with a few that are being examined for possible future use. Table 2.1 shows the algorithms provided on some commercially available CI models from Cochlear Corporation (CC) [20], Med-El (ME) [18], and Advanced Bionics (AB) [19].

The relevant algorithms can be broken into three major categories: Waveform (Compressed Analog: CA, Simultaneous Analog Stimulation: SAS), Interleaved (Contin-

Table 2.1: Commercially available CI products and their available speech processing algorithms.

Product	Algorithms					
	Waveform		Interleaved		Feature Extraction	
	CA	SAS	CIS	HiRes	SPEAK	ACE
CC Freedom	No	No	Yes	No	Yes	Yes
CC ESPirt 3G	No	No	Yes	No	Yes	Yes
ME TEMPO+	No	No	Yes	No	No	No
AB Platinum	No	Yes	Yes	Yes	No	No
AB Auria	No	No	Yes	Yes	No	No

uous Interleaved Sampling: CIS, HiResolution: HiRes), and Feature Extraction (Spectral Peak: SPEAK, Advanced Combination Encoders: ACE). Each category is discussed below, along with some possible variations in their implementation.

2.4.1 Waveform Strategies

CA was the first algorithm instituted in multichannel CIs by Symbion Inc. in a device called the Ineraid [21]. Fig. 2.6 shows the block diagram of the Ineraid device. The signal is taken from the microphone, compressed with the AGC, and split into four bands in the speech range as shown in the diagram. Compression is performed in order to reduce the dynamic range of the input signal so that it is on the order of the electrical stimulation

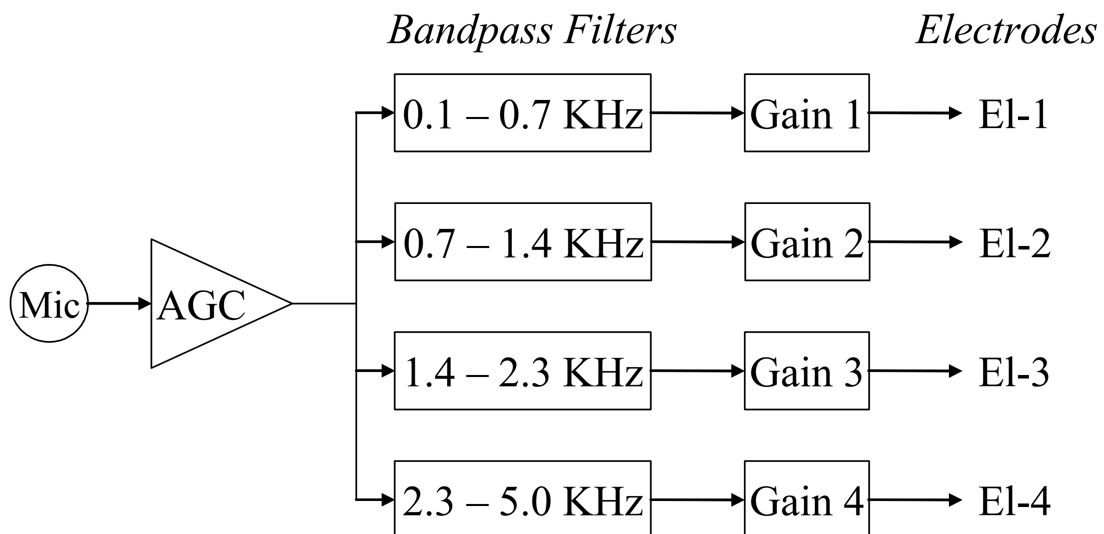


Figure 2.6: Block diagram of the Compressed Analog signal processing algorithm used in the Ineraid device by Symbion Inc.

that will happen at the electrode array. It is also possible to move the compression function to the back end of the circuit as is typically done in implants today.

After the signal is split into its channels, different gains are applied to each channel to ensure that the signals do not surpass the dynamic range of the electrodes and to emphasize the higher frequency bands. Stimulation is then provided to all four electrodes simultaneously in an analog format.

A version of the CA algorithm, called SAS, was developed by Advanced Bionics. The AGC was moved so that it follows the bandpass filters, and it used a non-linear compression function. In addition, the device was expanded to contain eight channels instead of four, mostly due to advances in electrode technology, not speech processing technology [22].

2.4.2 Interleaved Strategies

CIS, developed by Dr. Blake Wilson at the Research Triangle Institute (RTI), is by far the most common algorithm used by cochlear speech processors today. CIS has the best speech comprehension performance of any commercially available algorithm with existing electrodes [23]. Fig. 2.7 is a block diagram for an *n*-channel CIS processor. It has obvious similarities to the CA approach, as both are multi-channel algorithms.

The addition of the pre-emphasis high pass filter (HPF) attenuates frequencies below 1.2kHz in order to help the lower frequency weak consonant sounds to compete with stronger vowel sounds that are typically above 1.2kHz. Next, the sound is split and filtered into several different channels by the logarithmically-spaced bandpass filters (BPFs). Envelope detection by the low pass filter (LPF) and full- or half-wave rectifier is next, followed by the non-linear compression to reduce the dynamic range of the signal. Next is the volume control, which is managed by the patient and is the same for all channels.

The final stage, before transmission to the electrodes, is the pulse modulation. The output of each channel is modulated with non-overlapping bi-phasic pulses. The pulse amplitude, both positive and negative phases, is determined by the amplitude of the signal in the channel. The pulse rate (pulses delivered per second), pulse width, and channel to

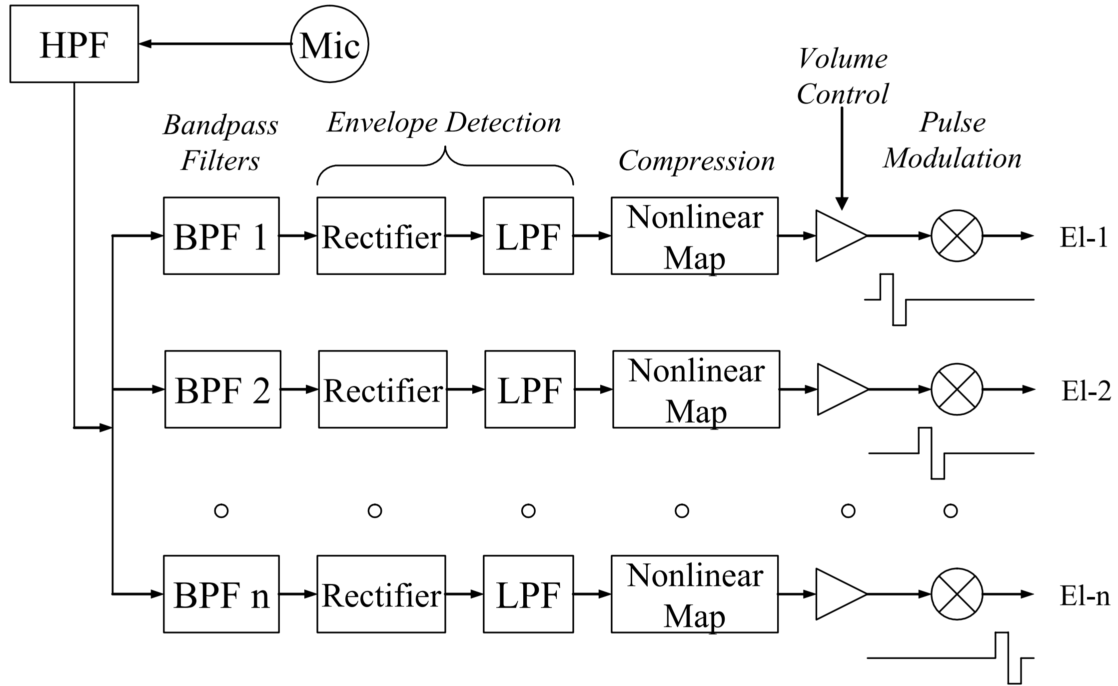


Figure 2.7: Block diagram of an *n* channel CIS signal processing algorithm.

electrode assignment are all factors that are patient specific and are programmed by the audiologist. These pulses are delivered at a high rate of typically 1,000 pulses per second (pps) per electrode. It is important to ensure that the pulse rate is higher than twice the cutoff frequency of the LPF envelope detector in order to prevent aliasing effects. Also, the positive and negative halves of the bi-phasic current pulses must be balanced because any charge imbalance can cause damage to the inner ear tissue. Other programmable features in the CIS algorithm normally include the BPF frequency ranges and compression function. The compression function will be discussed in more detail in Section 3.3.

HiResolution (HiRes) from Advanced Bionics is the newest commercially available speech processing strategy. Implants using this algorithm were approved for implantation by the FDA in 2000. The differences between HiRes and CIS include a wider input dynamic range in the input ADC and front-end, a higher frequency cutoff in the envelope detection LPF, and higher pulse rates with the capability for paired stimulation [24]. Recent research has shown that fine structure temporal cues in speech are above the standard 200 to 400Hz typical LPF in CIS. HiRes increases this LPF up to 2800Hz in order to obtain a sharper envelope and replicate timing information faster than a standard CIS implemena-

tion. Faster pulse rates (up to 89,600 pps) are employed; this is achieved more with increased electrode performance than increased signal processing capability. These small modifications to a standard CIS algorithm appear to help patient perception in noisy environments.

2.4.3 Feature Extraction and Spectral Information Strategies

Spectral information strategies began with the development of feature extraction strategies. Feature extraction strategies, such as F0/F2, F0/F1/F2, and Multipeak (MPEAK) differ from CA or CIS in that they try to present special features of the incoming sound to the cochlea instead of trying to represent the full waveform of information to the ear in either waveform or pulsatile format. They do this by utilizing methods such as zero-detection to extract features of the fundamental frequency (F0) and the formants (F1, F2) of the sound. This information would then be presented to the cochlea by stimulating the proper electrodes corresponding to the frequencies extracted. This approach has worked well under ideal circumstances, but background noise often causes errors in the zero detection circuitry, leading to frequencies being generated in the output that are not present in the input.

The next algorithm developed was called Spectral Maxima Sound Processor (SMSP). SMSP is similar to the later-developed CIS algorithm in that it compresses the input, splits it into different bands, and extracts the envelope. The differences lie in a few of the cutoff frequencies for the filters, and in that SMSP uses sixteen channels as opposed to the eight that are generally used in CIS. Also, SMSP employs an *n*-of-*m* strategy whereby, for each input time slice, it chooses to stimulate *n* of the *m* available electrodes corresponding to the *n* largest amplitude signals. The first SMSP implementation by Cochlear Corporation stimulated six electrodes out of sixteen channels. SMSP stimulates at a slower rate than CIS processors. Spectral Peak (SPEAK) was developed by incorporating a few small changes into the SMSP algorithm. First, it was expanded to twenty channels and the *n* stimulated electrodes were variable from patient to patient depending on pre-programmed thresholds and the energy distribution across the frequency ranges. Advanced Combination Encoders (ACE) has since been developed, and is often described as a combination of SPEAK and CIS. It takes the fast stimulation rates from CIS and the *n*-of-*m* fea-

tures from SPEAK [22]. It is also possible to expand CIS to more than eight channels as well as employ an *n*-of-*m* strategy among the channels and electrodes.

2.5 Implementation Methods

There are several options available when deciding how to implement a specific speech processing algorithm in integrated circuit (IC) technology.

2.5.1 Software Programmable DSP

The programmable DSP market was worth $12 Billion in 2009 and is expected to grow to $15 Billion by 2012 [25]. This has created a competitive market with several options available for the consumers, in this case, CI manufacturers. The variety, flexibility, and availability of these off-the-shelf products has made them highly feasible in CI systems. Texas Instruments has developed the TMS320C5402, with help from Advanced Bionics Corporation, specifically for use in the Clarion and other CI systems [26], [27]. Motorola also has DSP chips currently being used in both research and commercial implant processors [28] - [30]. Table 2.2 gives a summary of some of the available DSP choices available in both academic and commercial CI work.

It is important to note that the Champ-LP processor is actually an array of sixteen custom designed, low frequency DSP chips. This allows for the chips to take the performance penalty and power savings of running at a lower voltage than is typical for circuitry in that process. However, it is not clear what algorithm(s) Champ-LP is capable of executing or how much flexibility exists in fitting the algorithm to the patient's needs.

One drawback to designing a full CI system which utilizes an off-the-shelf programmable DSP chip is that other circuitry is necessary to facilitate system control, electrode stimulation, data communication, and patient fitting. The Sharp system requires adding a ROM, three serial ports, one parallel port, and an ADC to the existing DSP chip. This greatly increases the system size and power consumed performing inter-chip communication.

Table 2.2: Summary of programmable DSP options.

Processor	Technology (μm)	Area (mm^2)	Power Supply (V)	Freq. (MHz)	Current (mA)	Standby Current (mA)
5402[a]	0.18	100[b]	1.8	100	45	2
Champ-LP[c]	0.35	48	0.85	N/A	7.1	N/A
Sharp[d]	N/A	400[b]	3.3	100	48.2	32.3

a. Texas Instruments TMS320C5402 [26].

b. Area includes standard QFP package

c. [31].

d. Uses Motorola DSP56309 [28] with other off-the-shelf components [29]

2.5.2 Analog Signal Processing

Several attempts have been made to design analog circuitry that can perform the same tasks as the programmable DSP chips, and the results have been favorable. Table 2.3 shows a summary of some analog implementations that have been developed. It is obvious that the power savings for these analog implementations over programmable DSPs are great. However, they do not have the same sleep or standby mode advantages that the digital implementations realize. The speech processing algorithm is constantly running to continuously process sound and deliver it to the patient's cochlea, but there is not always information in every channel to be processed. Digital circuitry can easily shut off particular units or entire channels when inputs to that channel are below a certain threshold. The analog counterparts must remain awake and keep bias currents running in case the channel is required to start computation again. The "wake-up time" for analog circuitry is usually greater than that of digital circuits and therefore sleep modes are not as prevalent in analog circuitry.

As with programmable DSP implementations, other components must be added to perform the system's required tasks. Consider also that the analog systems must communicate with several pieces of digital equipment and software during the patient fitting process. While both analog implementations above tout programmable filters and channel variables, it is unclear how easily they will integrate with the audiologist's equipment.

Table 2.3: Summary of analog signal processing options.

Processor	Technology (μm)	Area (mm^2)	Power Supply (V)	Power (μW)
Toumazou[a]	0.8	N/A	5.0	sub 1000
Sarpeshakar[b]	1.5	22[c]	2.8	400[d]

a. [32]

b. [34], [35]

c. Area is for a 2-channel signal processing chip

d. Power is estimated for a 32-channel version not actually implemented

2.5.3 Integrated SoC

A third approach to implementing DSP functionality for a CI is to integrate the DSP functionality onto the same silicon that performs other tasks for the CI system such as clock generation, analog to digital conversion, RF communication, electrode telemetry feedback, and system control. Only in recent years has this become possible with the advances in IC technology.

A digital microcontroller that processes instructions and other mixed-signal components are needed in order to perform the system tasks previously described. This is typically done by a small, low-power microcontroller with several peripheral units closely integrated. Several product lines of microcontrollers are available from industry, including Motorola, Intel, ARM, and Texas Instruments. In order to add DSP capability to the existing microcontroller, one of two approaches can be utilized.

Instructions can be added to the microcontroller instruction set architecture (ISA) that perform DSP-like tasks utilizing high performance computational units. While this has the advantage of being the easiest approach to implement, it can be difficult to achieve the required performance from this method because, without greatly increased complexity, the microcontroller cannot perform its two tasks concurrently. It can be either controlling components of the system or processing incoming sound signals.

A second approach would be to add a DSP module to the microcontroller that could independently process speech. The DSP block could be treated like any other peripheral component by the microcontroller. It would just need to be initialized, and could then process speech signals by itself. Examples of this approach are an ARM core combined with

a programmable DSP [36] and Texas Instruments' TMS320DM310, which includes a microcontroller, peripherals, and a programmable DSP core. While these implementations have the advantage of being a fully integrated SOC, if the application is known prior to chip design and development, the DSP core could be tuned to the CI speech processing algorithms in order to increase performance in all areas.

2.6 Cochlear Implant Improvements

With improvements in the electrode array density and performance, research is progressing towards improving the quality of hearing for patients with CIs [37]. Perhaps even making listening to music enjoyable for CI patients. Improving on existing sound processing algorithms as well as developing new algorithms will assist in obtaining this achievement. As an aesthetic improvement, making a device that is fully implantable would benefit patients concerned with hiding the visible components of a CI. This will require further improvements in battery technology, especially charging techniques, and implementing a microphone system that allows for good sound pick up as well as minimal data loss upon transmission to the signal processing unit. The science, engineering, medical, and psychological professionals working on all of the above improvements are making progress towards making persons with a CI virtually indistinguishable from healthy hearing individuals.

2.7 Conclusions

In the past two decades, CI devices have improved the life and function of people who otherwise would be forced to rely on others for assistance to accomplish everyday tasks. It is exciting to see how beneficial current CIs are to the people who have them, and to work on technologies that will make them even more effective and convenient for future patients. Through continuing research, testing, and exploration, one can only hope that deafness will become a thing of the past.

The next chapter will describe the work done in order to improve the efficiency and reduce the footprint of Cochlear Implant signal processing, control, and communication as part of the WIMS MCU.

CHAPTER III

WIMS MICROSYSTEM

Primary design considerations for a battery powered bio-medical implant system are patient safety followed by power consumption and volume or area. The WIMS Microcontroller was designed and built with these targets in mind. The first generation of the WIMS Microcontroller was designed for use in an Environmental Testbed [5]. Due to the increased computational demands for sound processing in CIs, the second generation WIMS processor added a DSP and made some improvements to the WIMS ISA and performance. The first generation processor will not be discussed in detail here as it was covered in detail in [38].

3.1 Microsystem Architecture

The complete microsystem is shown in Fig. 3.1 as part of the WIMS CI System. The electrode array is described in [16] and its communication interface has been specifically designed to work with the DSP and Universal Synchronous-Asynchronous Receive-Transmit (USART) port on the WIMS Microcontroller. The telemetry coil and RF interface also utilize a USART port on the Microcontroller in order to receive instructions and send back data. The ADC is included to convert sound samples for the DSP. A mock up of this complete CI will be discussed in Section 6.2. The current section will deal with the WIMS microcontroller in detail.

The main components of the microcontroller are the 16-bit datapath, DSP, memory system, communication peripherals, and a hybrid clock source. The WIMS Microcontroller User Manual in Appendix A provides detailed information for system integrators, software

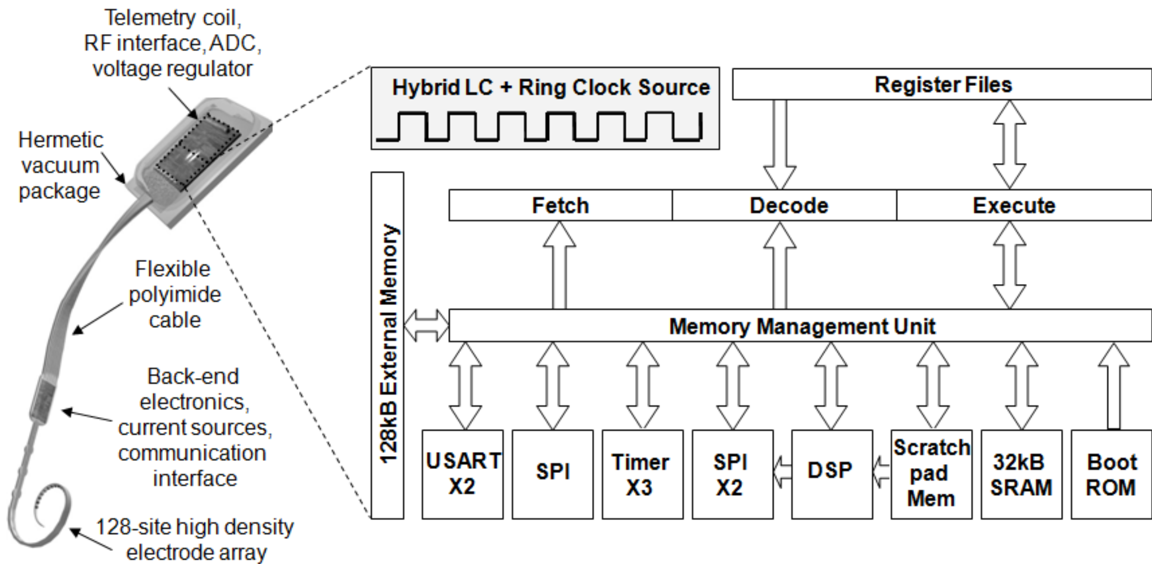


Figure 3.1: Complete microsystem architecture.

programmers, and users. It provides a full description of the processor organization and instruction set architecture (ISA). The ISA is summarized in the next section.

3.1.1 Instruction Set Architecture

The primarily load-store ISA[1] contains eighty-five instructions supporting eight addressing modes and single- and multi-word arithmetic, shift, logical, and control-flow operations. Instructions in the custom ISA were carefully chosen to minimize decode complexity and power without sacrificing functionality. One level of interrupt and subroutine support is available in hardware. Nested interrupts and subroutines are enabled through software control of the hardware stack and frame pointer.

The 16-bit MCU has a 24-bit unified data and instruction address space and can access sixteen data registers and fourteen address registers. The data registers are split evenly into two register windows. A separate address register window is included with six address registers in each window, with the stack and frame pointers available in both windows. The data and address windows are separated in order to allow compiler optimization of window switching. Register windows can be utilized to achieve up to 19% reduction in

1. The author wishes to thank Matt Guthaus for his contributions to the original WIMS ISA specification and Rob Senger and Rajiv Ravindran for their assistance in optimizing future ISA versions.

power consumption and 30% improvement in performance when compared to a non-windowed architecture [39].

Table 3.1 shows a detailed breakdown of the available instructions by type. There is additional support in the ISA for Direct Memory Access (DMA) instructions, but these instructions were not implemented on the Gen-2 WIMS Microcontroller due to time and area constraints. The DMA instructions would provide more efficient spilling of registers, subroutine calls, and loading of the scratchpad memory with data. These instructions were implemented in a subsequent version of the WIMS processor by Dr. Brown's students (Spencer Kellis, Nathaniel Gaskin, Bennion Redd, and Jeff Campell) at the University of Utah [40], [41].

A custom C compiler for the WIMS ISA was developed by Rajiv Ravindran, Ganesh Dasika, and Professor Scott Mahlke. The compiler, based off Trimaran [42], utilizes the existing WIMS assembler to map assembly language to binary code. Scripts are then available to load the binary code into the MCU via one of the external interfaces. The compiler has shown how to utilize the unique features of the WIMS ISA in order to yield efficient code. Register windowing schemes and DMA instructions were analyzed in [43]. Performance improvements utilizing the scratchpad memory for efficient use of commonly executed loop code was shown in [44]. The WIMS compiler group was also helpful in analyzing and optimizing the Gen-1 architecture and specifying the Gen-2 machine.

3.2 Microcontroller

Next, this section will discuss the main components of the MCU along with the testability features that have been included. The memory architecture will be discussed first, followed by the pipeline and peripheral components. Last, the testability features will be described.

3.2.1 Memory Architecture

Previous work has shown that large SRAMs consume more power than small SRAMs [45]. By subdividing the memory structure into blocks, at the cost of extra area for duplicated sense amps and other peripheral circuitry, one can obtain a memory structure that dissipates less power. The Artisan SRAM compiler was used to implement memories

Table 3.1: ISA Summary.

Type and Number	Instruction Category						
Load	Byte	Word	Absolute	Indirect	Update	Imm	Register
13	8	5	2	8	4	7	4
Store	Byte	Word	Absolute	Indirect	Update	Imm	Register
10	5	5	2	8	4	4	4
Arithmetic	Add	Sub	Mult	Div	Compare	Shift	Multi Op[a]
23	3	2	4	4	3	6	11
Logical	And	Or	Xor	Shift	Rotate	Imm	Register
13	2	2	2	3	2	6	6
Bit Ops	Set	Reset	Toggle	Imm	Register	—	—
5	2	2	1	3	2	—	—
Non-Windowed	Data	Address	—	—	—	—	
8	4	4	—	—	—	—	
Control	Jump	Branch	Return	Interrupt	Absolute	Relative	—
9	5	1	1	2	6	3	—
Test	Please see Appendix A for more details on the test instructions						
4							

a. Built-in support for multi-word operations

in the TSMC 0.18μm process. Based upon analysis of various sizes of memory blocks, and considering both area and power dissipation, the optimal configuration for 32KB of on-chip memory was found to be four banks of 8KB each as shown in Fig. 3.2. With this topology, all single-port memory banks that are not being accessed can be disabled on a cycle-by-cycle basis. It also allows instruction and data accesses to different banks of memory on the same cycle without stalling the pipeline. Dual ported SRAMs were not used because they occupied approximately twice the area of the single ported SRAMs of the same storage capacity. A dedicated memory management unit (MMU) in the core routes data from the correct bank to the requesting unit and disables inactive banks of memory. The memory speed is sufficiently fast to allow all accesses to complete within one cycle without the need for caches.

As a power saving feature, a modified scratchpad memory as shown in [45] - [47] was added to our chip. The scratchpad memory, also known as a loop cache, is a small, low-power 512-byte memory that is pre-loaded with the most commonly executed instructions (typically loop code) or commonly accessed data as determined through compiler profiling. This will greatly reduce the power consumption of the controller considering that embed-

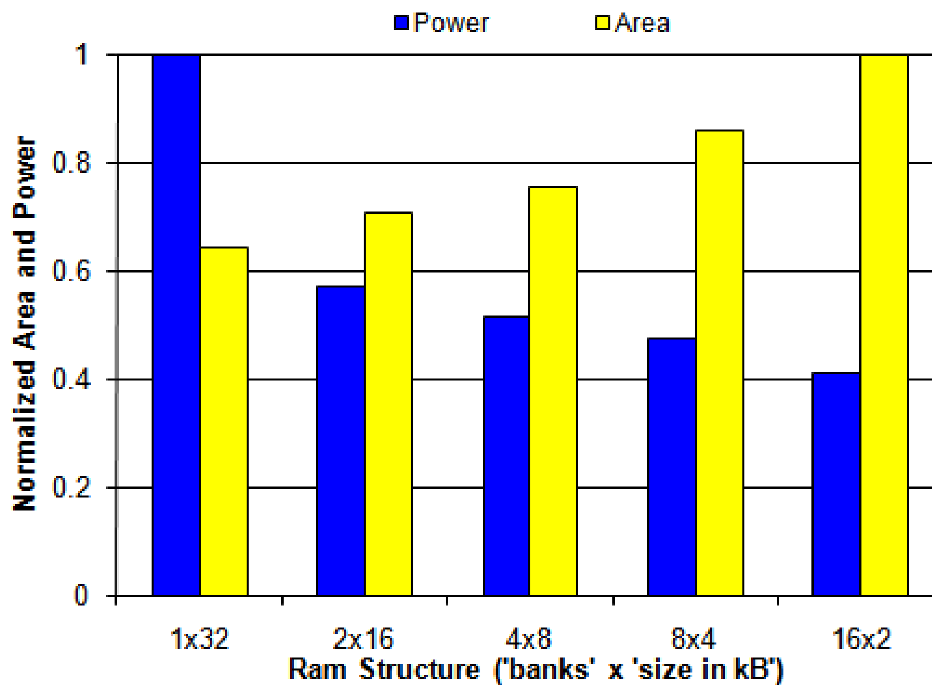


Figure 3.2: Normalized power and area trade-offs for possible memory configurations.

ded controllers typically run the same software throughout their lifetime and much of that time is spent executing loop code. The DSP is also given direct control over the scratchpad in order to store the non-linear compression look-up-table (LUT). An additional interface to up to 128KB of external memory is available for applications that may require more storage.

3.2.2 Pipeline

At the heart of any MCU is the pipeline to process instructions. Fig. 3.3 shows a block diagram of the WIMS MCU pipeline. The Instruction Fetch (IF), Instruction Decode (ID), and Execute (EX) stage compromise the 3-stage pipeline which efficiently implements the WIMS ISA. The Program Counter (PC) determines the instruction fetch address and can auto increment, take an interrupt address, or a branch/jump address from the EX stage. The sixteen data registers and fourteen address registers are split across independent windows. Memory and peripheral access are all handled through the memory management unit (MMU). All peripheral units' control and data registers are memory mapped.

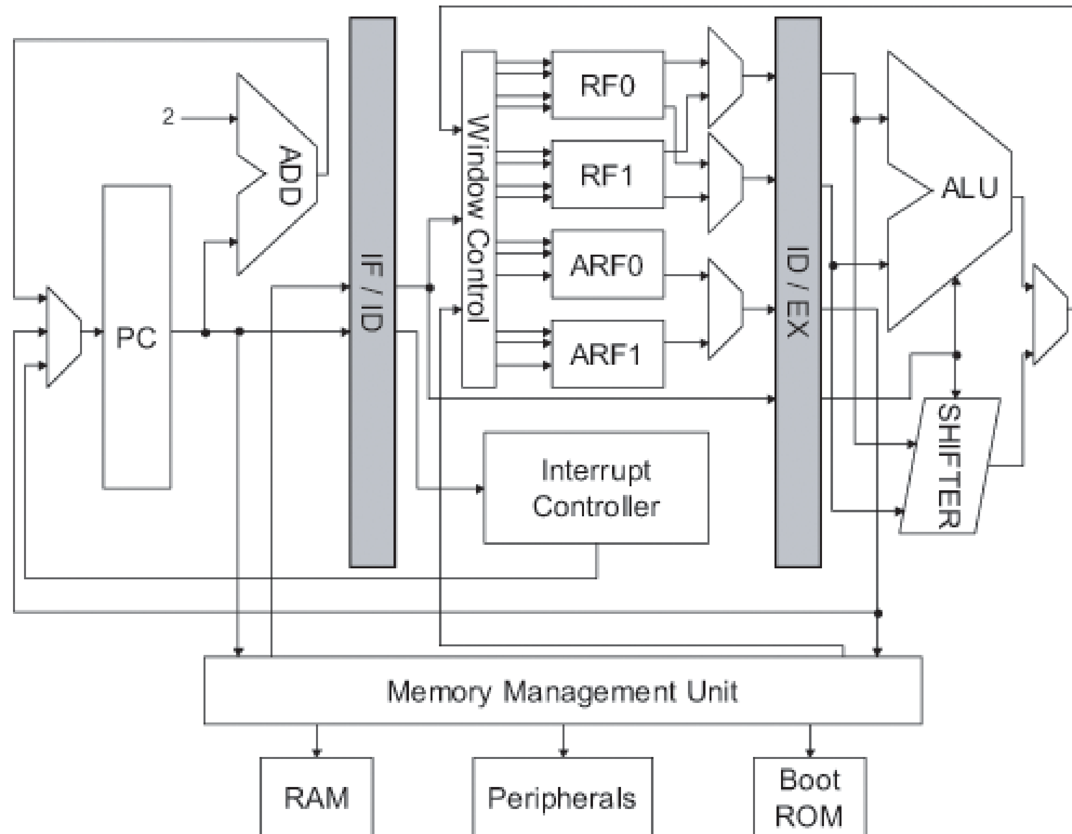


Figure 3.3: Pipeline block diagram.

The boot ROM assists with loading of the on-chip memory and setting the default states for all of the peripheral components. The WIMS MCU supports several startup scenarios based on external interrupts triggering specific interrupt code in the boot ROM. Code can be executed directly from the external memory, loaded from the external memory into the on-chip memory, or be loaded into the on-chip memory from the USART port. A flow chart for the boot up scenarios is shown in Appendix A.

3.2.3 Peripherals

Available communication interfaces include two USART ports and three Serial Peripheral Interface (SPI) ports. There are also three programmable timer interfaces which can interrupt the core or provide waveforms directly to external pins.

In order to support the cochlear DSP, two of the SPI interfaces have shared control between the MCU and the DSP. The controlling unit is selected via software. The controlling unit determines the clock domain for each SPI interface and where data and commands

are routed. In addition, the SPI interfaces have multiple operating modes to support several different external components. For backwards compatibility, there is an environmental mode to support the UMSI chip interfaces [48].There is also a cochlear mode to interface directly with the custom Michigan Electrode Array [16].

3.2.4 Testability

Several features are included in the MCU to facilitate post production testing. When choosing the test features to include, minimal overhead was the key metric. Therefore, trying to reuse existing features and hardware was usually the method chosen. There is support in the ISA for accessing data in test mode as well as hardware support for reusing existing pins for monitoring key registers on chip.

The ISA includes stop, start, and single-step instructions that can be included in any program to hold the MCU at particular points of program execution. Breakpoint modes are also included in the ISA to facilitate finding out what portion of a program is causing a particular failure. The MCU can be told to break on an address, data, or interrupt level match. Once a match is encountered, the MCU will halt execution by inserting a stop instruction after the instruction that caused the data match. At that point, the built in test interface (TI) can be used to interactively interrogate the registers and ascertain the cause of the failure.

The TI is wrapper around one of the existing USART peripherals; it is used only for debugging. When the TI is activated via a special test pin being driven high, it is given the highest priority interrupt level so that it can always insert an instruction into the pipeline when the MCU is in a stop state. If the TI is not activated via the test pin, it can be used as a standard USART port. The TI can also output data, and is given special access to any of the address, data, or status registers on the chip to output them for debugging. In the Gen-1 chip, all of the pipeline registers were specially mapped and available for the TI to interrogate and output. However, this access was deemed too resource intensive for the added visibility it provided so it was not included in the Gen-2 processor. An additional test mode feature is that all of the external memory interface pins are reused to output the value of the PC and EX stage write data (to memory or registers). This helps in monitoring the state of the MCU while sending in debugging instructions.

In addition to the software and ISA support, hardware support is included for testing certain features. An external clock source for the MCU can be provided and chosen via an external pin. This will bypass the on-chip clock generation in case there are any problems with the on-chip clock and allows for testing performance of the MCU outside of clock rates that can be generated on-chip. Three pads are also multiplexed to simplify testing of the on-chip clock generation. The three pins create a shift register out of the thirty-two control bits in the clock generation IP block. The output is routed directly to a bonding pad for monitoring.

3.2.5 Summary

The MCU is a highly flexible processor specifically targeted at remote, low-power applications. The ISA provides enough support for on-chip data processing, debugging applications in the field, and support for compiler optimizations. Efficient implementation of several peripheral communication interfaces allow for adaptation to many possible applications and external interfaces. 32KB of on-chip SRAM, 128KB of available off-chip SRAM, and a 512B on-chip scratchpad memory provide a good balance of power, performance, and chip real estate for several applications.

3.3 Digital Signal Processor

A DSP module was added to the WIMS Microsystem in order to increase the processing capability while at the same time reducing the power consumption required to perform the CI sound processing [49]. A low-power implementation was the first goal, but the system also needed to be flexible enough to allow for patient-specific programming comparable to that of commercial CIs. The CIS algorithm was chosen due to its popularity in commercial devices and high speech recognition rates among patients [23].

At a high level, the DSP performing the CIS algorithm takes input samples from an ADC and splits the sound into several channels based on frequency. Each channel is processed further to find the envelope and compress the dynamic range. Volume adjustments can be made before sequencing through each channel and passing its data to a corresponding electrode to provide stimulation. Additionally, test and programming features need to

be included in the DSP for safety and further research along with the patient fitting procedure.

Integrating the DSP core was done by taking advantage of the existing processing capabilities and communication interfaces of the MCU. The DSP expects the MCU to perform all of the system setup and can therefore eliminate many components required in a standalone DSP implementation. In order to communicate with the ADC and electrode array, the MCU allows the DSP to take control of two SPI peripheral units by setting control bits in the peripheral units. The DSP also expects the MCU to put the SPIs into the correct modes before giving control over to the DSP. Read access to the 512-byte on-chip scratchpad memory is given to the DSP. This allows the DSP to use it for efficient storage of the dynamic range compression via a look up table (LUT).

The parallel nature of the CIS algorithm provides for a significant reduction in hardware by pipelining the datapath and allowing all channels to share the same hardware for filters, LUT, volume, and pulse modulation. Control circuitry is also simplified by allowing finite state machines (FSMs) in the control unit to reuse states for each channel.

3.3.1 Architecture

Fig. 3.4 shows a block diagram of the fully-synthesizable signed-magnitude fixed-point DSP core. The highpass filter (HPF), bandpass filters (BPFs), and lowpass filters (LPFs) are implemented as cascaded infinite impulse response (IIR) stages due to the low memory requirements and simplicity of the hardware. They are 1st, 6th, and 4th order, respectively. Equation 3.1 through 3.3 show the filter equations for the HPF, BPFs, and LPF in terms of their coefficients (a_n, b_n) and the input samples (z^n).

All filter coefficients are programmable by the MCU to provide the required flexibility for patient fitting procedures. All filters realize a vast reduction in area, due to the multiply-intensive nature of filters, by sharing the same cascade stage hardware. Register storage for the filter coefficients and output data from each datapath stage is a significant portion of the architecture. Clock gating is performed on these registers to reduce the clock tree loading, and therefore power consumption.

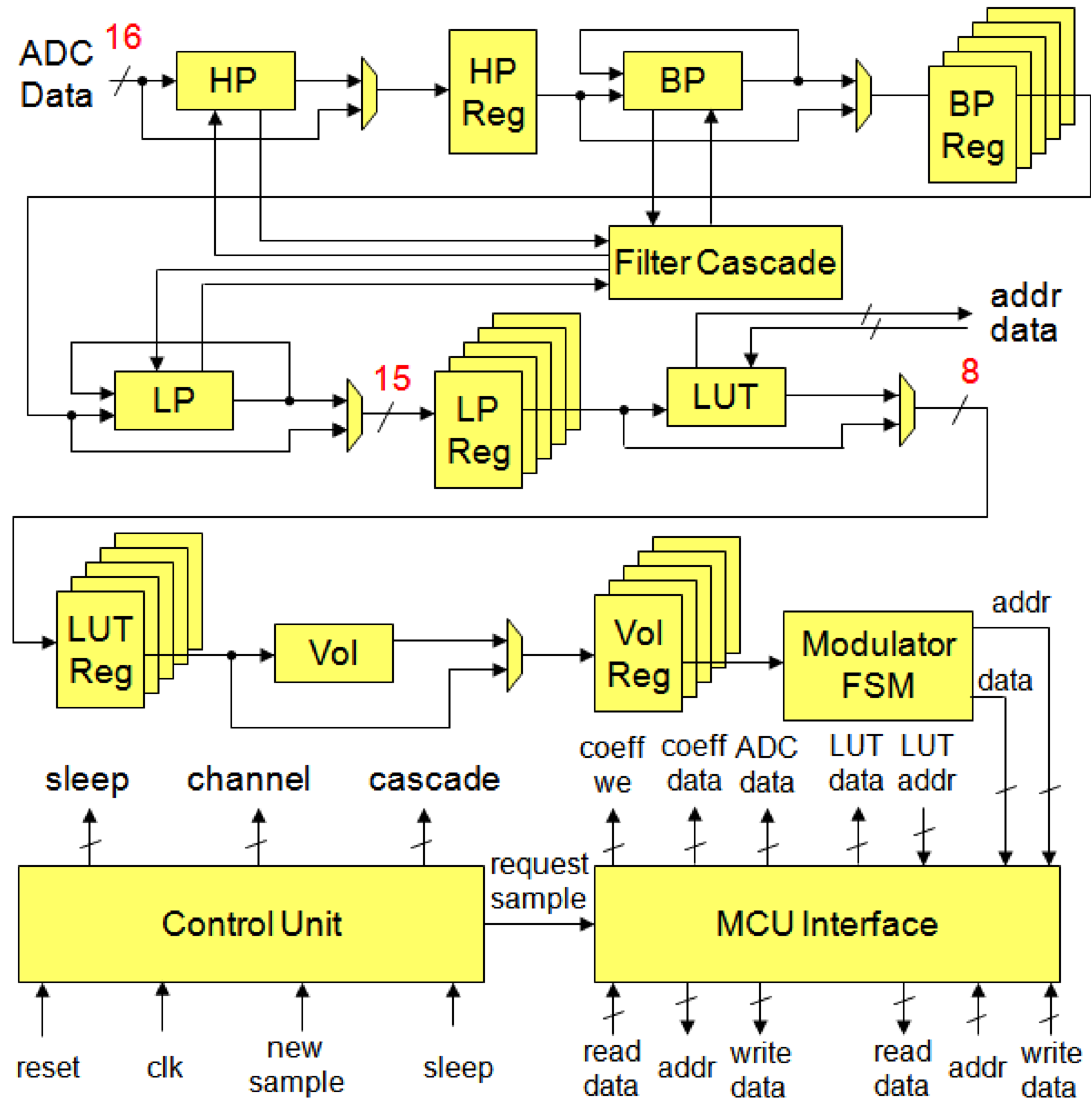


Figure 3.4: CIS DSP architecture block diagram.

Compressing the dynamic range, and therefore the number of bits required to encode the signal amplitude, saves power and area by reducing the datapath width from sixteen to eight bits for all downstream calculations and storage. This logarithmic compression is done using the low-power scratchpad memory to store the LUT data. By allowing the MCU, and therefore the software, to control the data in the 512-entry LUT, the patient can have the compression function fitted to achieve their best performance.

Similarly, the patient can use the volume control to set the volume gain for each channel independently. Equation 3.4 shows the volume equations where THR is the

patient's minimum hearing *THR*eshold and MCL is the patient's *M*ost *C*omfortable *L*evel. Both THR and MCL are measured by the audiologist and are channel dependent.

$$\frac{y[n]}{x[n]} = \frac{b_0 + b_1 z^{-1}}{1 + a_0 z^{-1}} \tag{3.1}$$

$$\frac{y[n]}{x[n]} = \frac{(b_{00} + b_{01}z^{-1} + b_{00}z^{-2})(b_{10} + b_{11}z^{-1} + b_{10}z^{-2})(b_{20} + b_{21}z^{-1} + b_{20}z^{-2})}{(1 + a_{00}z^{-1} + a_{01}z^{-2})(1 + a_{10}z^{-1} + a_{11}z^{-2})(1 + a_{20}z^{-1} + a_{21}z^{-2})} \tag{3.2}$$

$$\frac{y[n]}{x[n]} = \frac{(b_{00} + b_{01}z^{-1} + b_{00}z^{-2})(b_{10} + b_{11}z^{-1} + b_{10}z^{-2})}{(1 + a_{00}z^{-1} + a_{01}z^{-2})(1 + a_{10}z^{-1} + a_{11}z^{-2})} \tag{3.3}$$

$$y = volA \bullet x + volB \qquad volA \in [0, 1] \qquad volB \in [THR, MCL] \tag{3.4}$$

Pulse rate, channel to electrode assignment, and pulse duration are all programmable in the modulator FSM via the MCU interface. The current implementation allows for any bi-phasic pulse stimulation information to be sent to the electrodes through one of the system's SPI interfaces. The maximum pulse rate is 3,000 pulses per second. If more complex stimulation profiles are desired, the MCU can read the data coming out of the DSP volume stage and perform computations to calculate the appropriate pulse characteristics. It can then take control of the SPI interface and send information to the implanted electrodes, bypassing the DSP modulator FSM. This makes the microsystem a useful tool to evaluate experimental stimulation profiles.

3.3.2 Modes of Operation

The DSP core has four operating modes: stimulation, programming, test, and sleep. For typical operation, the DSP will be in stimulation mode and will process samples and generate stimulation pulses automatically. The MCU can be in standby mode during this time. Input data is received from the external ADC through one of the shared SPI interfaces. Programming mode will allow the MCU to set up all filter coefficients, LUT data, and the stimulation profile. Test mode allows for any of the datapath stages to be bypassed via the multiplexors at each output node to provide observability and controllability over each component in the DSP. Sleep mode allows all DSP components to be shut down in order to

conserve power. While in stimulation mode, any unused DSP datapath stages are shut down through the control unit by utilizing the existing sleep mode circuitry.

Assuming a 22kHz front-end ADC, which is standard for speech processing within the human audible range of 0 to 10kHz, the DSP must operate at 3MHz to provide adequate output data for high pulse rate stimulation. This calculation is based on the DSP processing time of a single sample and the data transmission rate to the electrodes.

One key benefit of this DSP architecture is important to point out here. Since all processing units are used in series, all that is required to increase the number of channels is to increase the storage for new channel coefficients and increase the operating frequency of the DSP. There is margin available to run the 3MHz DSP much faster in the TSMC 0.18µm process. Thus, scaleability is a large benefit of this DSP architecture.

3.3.3 Interfaces

The DSP has to communicate with several different components in different modes of operation. This section outlines the operation of the interfaces to the MCU, ADC, and electrode array.

3.3.3.1 Microcontroller

The DSP can be considered a slave to the MCU. The MCU can program all variables and coefficients for the DSP as well as the clock domains and access control for the shared SPI units and scratchpad memory. The DSP can then access and send all information via the memory management unit (MMU). This master/slave relationship between the MCU and DSP allows for efficient reuse of system resources with minimal overhead.

This master/slave organization allows the MCU to control either SPI interface and to inject data samples into the DSP or send data directly to the electrode array. While this mode of operation would be used only for experiments, it is useful for debugging or experimental algorithm investigation.

3.3.3.2 Analog to Digital Converter

The SPI interface supports interfacing to the AD7708 [50], AD7888 [51], or compatible ADC. The interrupt from the SPI interface tells the DSP control unit that a new sample is available and ready for processing. The DSP control unit will then present the

sample to all channels at the next cycle of channel processing. This ensures that all channels process the same data.

3.3.3.3 Electrode Array

The Michigan Electrode Array has an SPI interface for receiving data that will drive the electrodes. A custom data transmission protocol is used by the DSP to send the electrode address and the amplitude of stimulation pulses for each channel. The channels are serially sequenced and the delay between pulses is programmable. Since the pulses per second delivered to the electrodes are set by the transmission rate through the SPI interface, this interface has priority over all other DSP operations. The DSP channel to electrode assignment is also programmable via the MCU setup routine.

3.3.4 Conclusions

The custom DSP presented here is an efficient implementation of the CIS sound processing algorithm. It converts sound samples into signals that can be understood by the electrode array in order to deliver stimulation to the cochlea and allow a CI patient the perception of sound and ability to comprehend spoken language. The four modes of operation allow for system integration and setup. The DSP is flexible and expandable for future developments in electrode technology or signal processing. Modifying the DSP presented here to support the HiResolution algorithm described in Section 2.4 would only require changing the LPF cutoff frequencies and increasing the speed of the modulator FSM.

3.4 Clocking Scheme

The ability to dynamically scale CPU clock frequency with workload has become an important technique for reducing active and standby power consumption in nanoscale embedded systems. Dynamic frequency scaling (DFS) has been used successfully to reduce power in portable embedded applications such as PDAs and cell phones. Recently, even high-performance microprocessors such as Intel's dual-core Montecito have adopted complex dynamic voltage and frequency scaling (DVFS) control circuits to maintain power and temperature within acceptable limits [52].

Many DFS circuits are implemented with a phase locked loop (PLL), which is used to multiply a low frequency reference signal that is typically derived from an external crys-

tal oscillator (XO). The PLL prescaler can be changed to generate a new clock frequency, however, even relatively fast-locking PLLs incur a delay on the order of microseconds to regain lock after the prescaler has been changed [53], [54]. During this time, the system clock is typically unusable, which can easily translate into thousands of missed CPU cycles. To avoid stopping the CPU, dual-PLL DFS architectures such as [55] use one PLL to produce a usable system clock while the second PLL's prescaler is changed. After the second PLL has locked onto the new frequency, the system clock can be switched. This approach is problematic when transitioning from low to high frequency to handle sudden, unexpected increases in workload as might occur in interrupt-driven embedded systems. For real-time applications, the delay to lock the second PLL on a higher frequency can result in unacceptable performance degradation. A two PLL system is also significantly extra area compared to single PLL systems. DFS architectures proposed in [53], [56] avoid this problem by running a single PLL at high frequency and dividing the clock down. Using a frequency divider on the output allows for nearly instantaneous frequency switching without incurring the lock penalty associated with changing the PLL prescaler. However, by maintaining the PLL at a high frequency and using clock division, the PLL dissipates more power than if the prescaler were reduced during times of low CPU activity.

3.4.1 Dynamic Frequency Scaling

Most DFS circuits require full-custom design with careful transistor level simulation to avoid glitches on the clock during frequency transition. In this work, we present an HDL-synthesizable, low-latency, glitch-free dynamic clock frequency controller that switches frequencies without halting the CPU. Rather than a traditional bottom-up approach using a low frequency external XO reference that is multiplied by an on-chip PLL, our implementation employs an all-silicon hybrid temperature compensated LC oscillator (TC-LCO) to eliminate the need for both the PLL and external reference. The TC-LCO produces a high-accuracy, low-drift, and low-jitter 1GHz sinusoidal reference that is frequency divided and turned into a square wave. A ring oscillator is also provided for low power standby mode.

Fig. 3.5 shows the novel DFS circuit. A simple flip-flop chain divides the monolithic clock reference ($2f_0$) to produce eight frequencies ranging from f_0=100MHz to

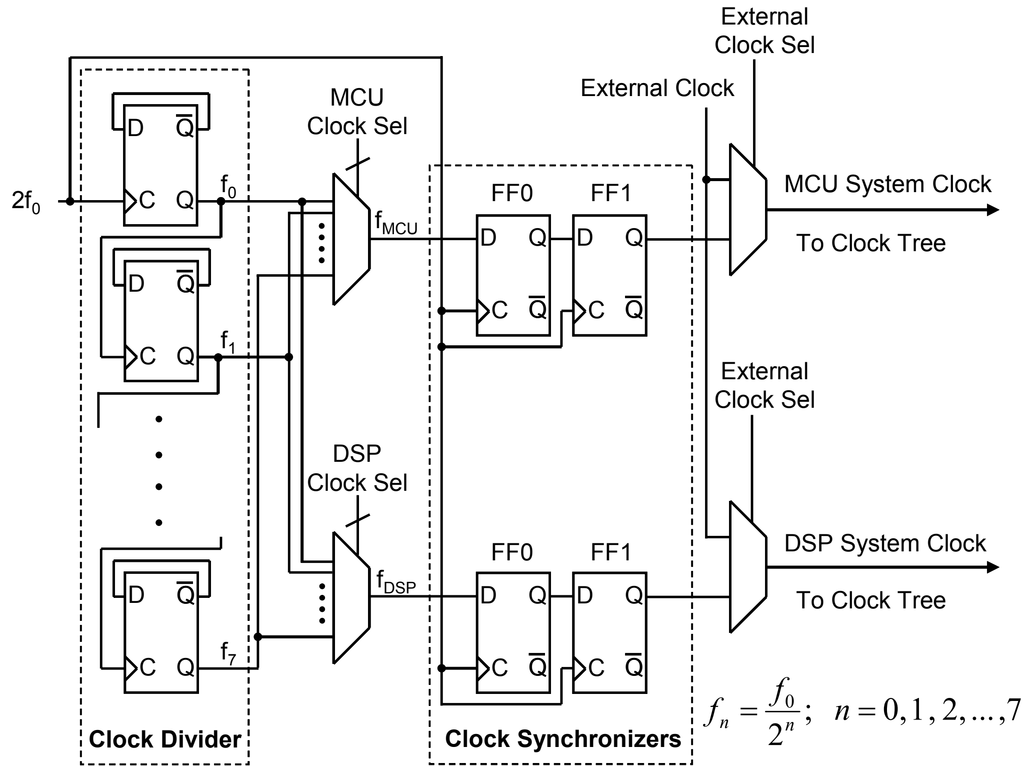


Figure 3.5: HDL synthesizable glitch-free dynamic frequency controller [57].

f_7=781kHz if the TC-LCO is enabled and from f_0=10MHz to f_7=78.1kHz if the ring oscillator is enabled. The eight selectable frequencies supply two software controlled 8-to-1 multiplexers that provide separate clocks to the MCU and DSP cores. The multiplexers' outputs (f_{MCU}, f_{DSP}) are not hazard-free and require parallel chains of synchronizing flip-flops (*FF0*, *FF1*) to remove glitches and eliminate metastability that might occur when switching between frequencies. Optional 2-to-1 multiplexers select between the on-chip clocks and an external clock; however, these multiplexers cannot be changed dynamically. This DFS circuit can easily be expanded to provide additional, independently selectable clocks if required.

Glitches on the clock signal can result in logic failure through two primary mechanisms. If the glitch is small enough that some downstream clocked elements treat the glitch as if it were a clock pulse and some do not, this can cause a failure. Alternatively, if the glitch occurs too close to an adjacent clock edge, this can lead to a timing failure. By clocking the synchronization flip-flops with $2f_0$, the delay between consecutive rising (falling) and falling (rising) edges on *FF0.Q* is forced to be greater than or equal to the half-period

of f_0. Assuming the digital logic is designed to operate at a maximum frequency f_0, this implementation will prevent both of the aforementioned glitch-related timing errors even in latch-based designs. Fig. 3.6 shows examples of how the circuit suppresses glitches on the f_{MCU} multiplexer output during frequency transitions. As the software dynamically switches the clock selection multiplexer between f_0, f_2, and f_1, glitches occur on f_{MCU} at the instant of the switch. These glitches are restricted to frequency f_0 by *FF0* and should not cause logic malfunction in the MCU core.

Although only *FF0* is required for glitch suppression on the selected clock, a metastable value could be latched into *FF0* if f_{MCU} were sampled while in the process of transitioning. This is possible if any of the eight paths through the clock divider chain violate setup time on *FF0*. Metastability could also occur if the MCU clock multiplexer selected a new frequency as *FF0* sampled f_{MCU}. These timing problems can be avoided through careful design and simulation of the various timing paths through the DFS circuit. However, in a fully synthesized design where there is limited control over the synthesized result, it can

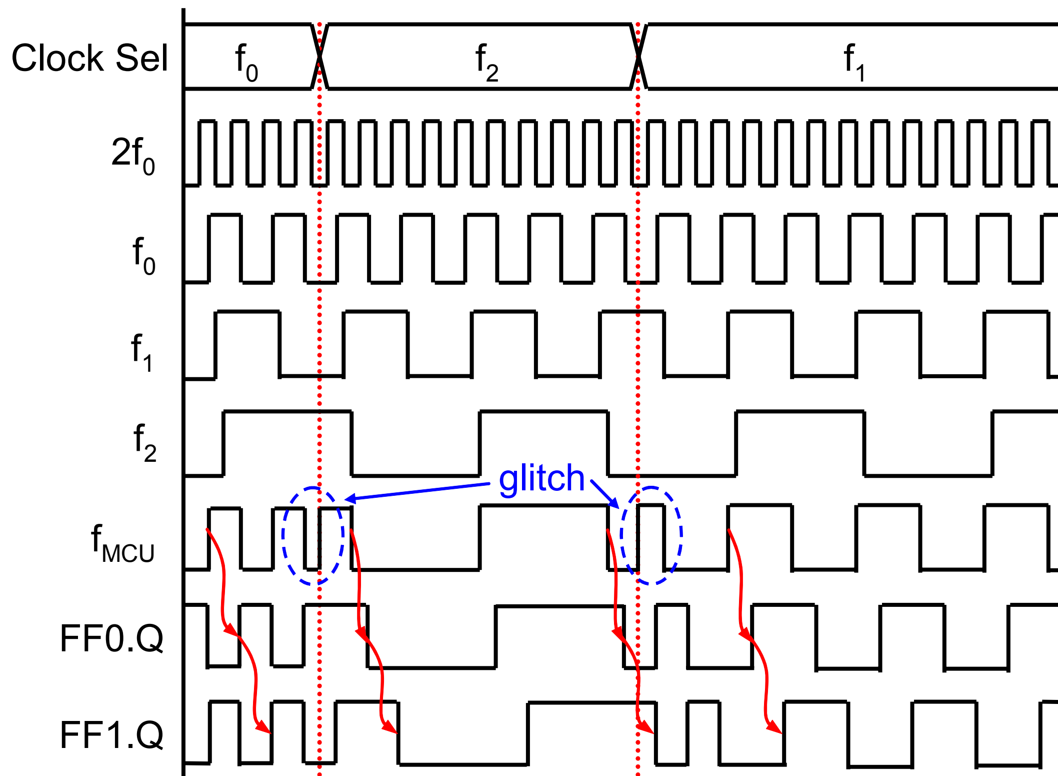


Figure 3.6: Glitch suppression on f_{MCU} during frequency transitions.

be difficult and time consuming to guarantee the necessary timing accuracy to completely avoid metastability in *FF0*. As a relatively simple solution to this problem, *FF1* was added to minimize the probability of metastability on the clock output. To approximate the Mean Time Between Failures (MTBF) caused by a metastable output from *FF1*, the equations from [58] can be used.

From Equation 3.5, f_{clk} is the clock frequency currently selected, $2f_0$ is the synchronization flip-flops' sampling frequency, t_a is the flip flop aperture time, τ_s is the regeneration time constant, and n is the number of synchronization flip flops. Both t_a and τ_s can be approximated at 200ps for this analysis in a 180nm process. For the current configuration where n=2 flip-flops, the worst case MTBF occurs when $f_{clk}=f_0$=100MHz.

$$\mathrm{MTBF} = \left(t_a f_{clk} 2 f_0 \exp\left(-\frac{t_w}{\tau_s}\right)\right)^{-1}$$
$$t_w = \frac{n-1}{2f_0} \tag{3.5}$$

At this speed, the DFS circuit would be expected to have a metastability failure once in 150 years. Adding a third flip flop to the synchronization chain would increase the MTBF to $2.85\text{x}10^{18}$ years at the expense of another cycle of clock switching latency.

The proposed DFS circuit is particularly well suited for integration with high frequency TC-LCOs because the $2f_0$ required for synchronization is already generated when using a FF chain to divide the LCO's high oscillation frequency. LCOs must operate at high frequencies to minimize inductor area [59]. An additional benefit of dividing a high frequency reference clock by N is that the relative period jitter for the divided clock is reduced by a factor of $N^{1/2}$. In contrast, when a low-jitter XO reference is multiplied N times using a PLL, the relative period jitter is increased by $N^{1/2}$ [60]. It is worth noting that our DFS circuit could be used with a PLL as the clock source, however power consumption would increase in order to produce $2f_0$.

3.4.2 Mobius Microsystems' IP

The self-referenced hybrid clock synthesizer[1], shown in Fig. 3.7, includes a free-running RF LCO, a low power ring oscillator, a temperature-compensated bias circuit, and

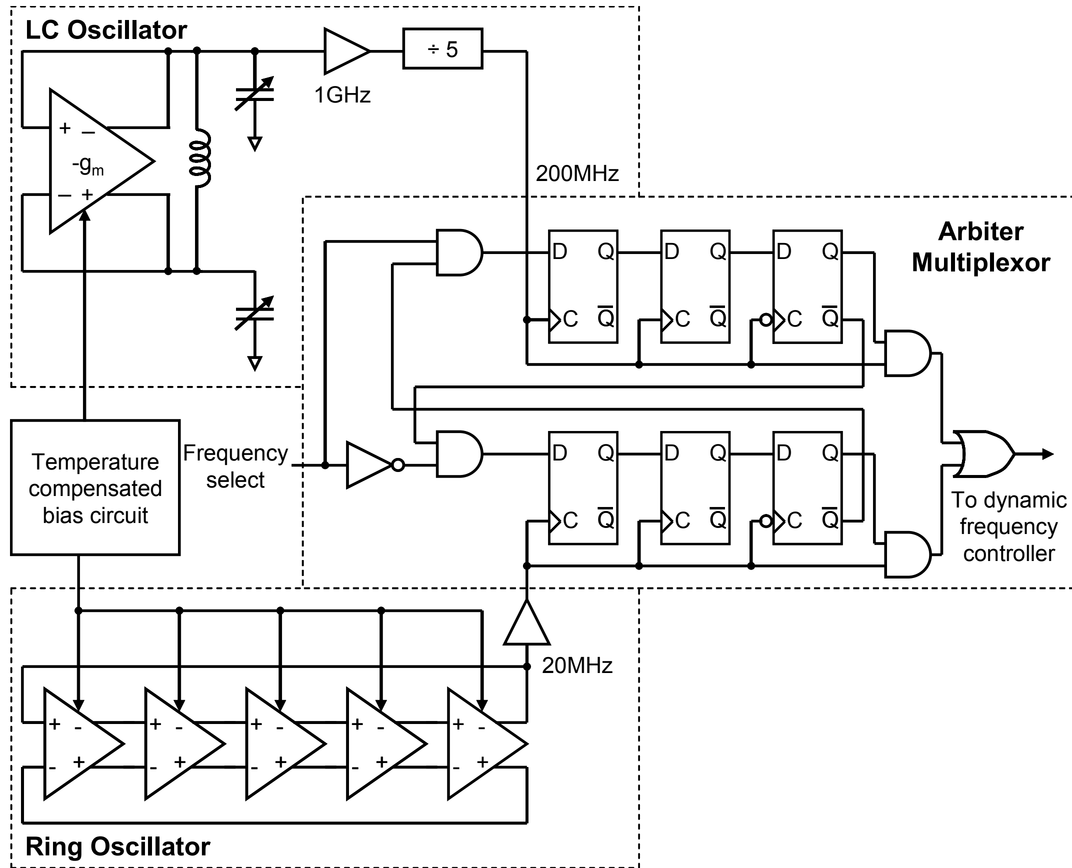


Figure 3.7: Self-referenced hybrid clock synthesizer.

an arbiter [61] for asynchronous glitch-free switching between the two oscillators. The synthesizer supports a reduced power standby mode in which the TC-LCO is powered down while the system operates from the low power, low frequency ring oscillator. The entire clock synthesizer occupies 0.25mm^2 of silicon area.

The RF LCO includes a complimentary and cross-coupled negative-transconductance sustaining amplifier, a single differential inductor, and a bank of switched capacitors in parallel with the LC tank. The LCO generates a 1GHz reference signal that is followed by a frequency divide-by-5 circuit. Frequency deviation due to process variation can be corrected by trimming the capacitance in the LC tank using an 8-bit calibration byte. The measured calibration range is ±10.75%, giving an initial calibrated accuracy of ±420ppm at

1. The author wishes to thank the group at Mobius Microsystems for their development of this IP: Michael McCorquodale, Scott Pernia, Justin O’Day, Gordy Carichner, and Sundus Kubba.

25°C. The ring oscillator contains five differential stages and nominally outputs a 20MHz signal that can be calibrated via the digital interface to account for process variation. The LCO dissipates 9.62mW and the ring oscillator dissipates 0.82mW at 1.8V.

3.4.3 Conclusions

The flexibility provided by both the hybrid clock synthesizer and the DFS circuit is an excellent feature for a general purpose MCU. The merging of these two components to provide a wide range of accuracy and current consumption for both the MCU and DSP allows adaptation under software control to the required computing performance.

3.5 Summary

This chapter has described the WIMS Gen-2 Microsystem in detail. The microsystem architecture was described and each component was discussed further. The MCU ISA was explained and the core of the WIMS microcontroller was described. The peripheral communication capabilities were outlined. The distinct features of the microsystem were reported: LCO, DFS, DSP, and scratchpad memory. Special attention was paid to the DSP architecture and its application to the CIS algorithm and CI component interfaces.

The next chapter will describe the design methodology used to implement the WIMS Gen-2 Microsystem.

CHAPTER IV

MICROSYSTEM DESIGN METHODOLOGY

Once a microsystem has been described in technical detail as in Chapter III and Appendix A, the implementation of said microsystem is a significant challenge. It requires contributions from a large group of people all working together. This chapter describes the methodology used by the members of Dr. Brown's research group[1] to fully implement the WIMS Gen-2 Microsystem.

4.1 Design Methodology

Microsystems are defined as intelligent miniaturized systems comprised of sensing, processing and/or actuating functions where two or more of the following technologies are combined onto a single or multi-chip hybrid: electrical, magnetic, mechanical, optical, chemical, or biological [62]. The primary motivations driving microsystem development are the need to increase functionality and performance while reducing system size, cost, integration complexity, and power dissipation. The proliferation of portable electronic devices in recent years has hinted at a bright future for microsystems, especially when one considers the interest in remote sensor devices and biomedical implants enabled by advances in low-power design and nanoscale integration.

Building a complete microsystem involves several challenges, as these designs unite not only analog and digital circuit domains, but also the magnetic, mechanical, bio-

1. Specifically, the author would like to thank Robert Senger, Michael McCorquodale, Matt Guthaus, Fadi Gebara, and Steve Martin for the contributions to the various versions of the WIMS Microsystems.

logical, chemical, or electrical domains. Digital microcontrollers are manufactured almost exclusively in CMOS technology, but many of the desired peripheral components are not compatible with CMOS and must be fabricated separately or adapted to use CMOS. Moreover, the design constraints associated with these systems can be as specific as the material properties of a layer that defines a microstructure, and as broad as an abstraction of the embedded processor that supports the firmware for the microsystem. A successful microsystem design requires integration and cross-boundary communication from designers working at the device level up to the application software engineers. A variety of tools for the support of such designs exists, but as yet, there is no complete end-to-end framework for microsystem development. This work leveraged advances in integrated circuit (IC) computer aided design (CAD) tools, applying them to microelectromechanical systems (MEMS) and mixed-signal circuits to address the challenges of microsystem development by integrating hardware and software domains and identifying gaps that require new design automation.

4.1.1 Design Trends and Challenges with Microsystems Technology

Fig. 4.1 illustrates a generalized end-to-end wireless integrated microsystem. The figure shows a variety of technologies that might be included, along with the wide range of design tools that might be needed for their design. For example, MEMS components are

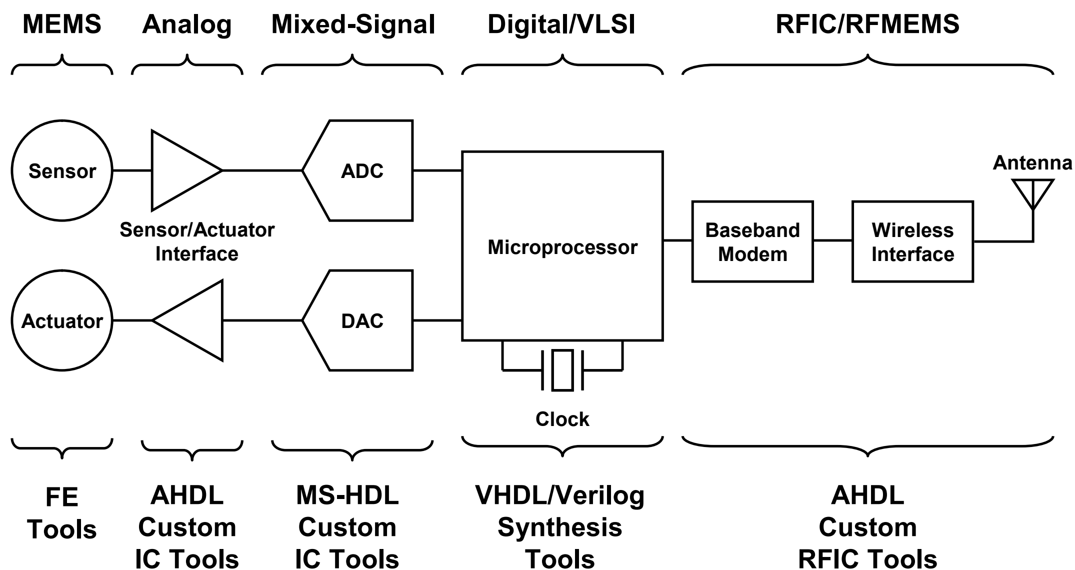


Figure 4.1: The anatomy of a generalized wireless integrated microsystem. Key technologies and associated development tools are shown.

often developed with finite element (FE) tools that simulate a mechanical response to an applied stimulus [63], while the microprocessor section would typically be synthesized from a hardware description language (HDL). Digital IC design tools are now ubiquitous and offer the designer tremendous flexibility through system abstraction. Only recently have such trends developed in the analog and mechanical domains.

Several MEMS technologies warrant integration with integrated circuitry. Indeed, a great deal of research has been underway in this field, including activities in monolithic MEMS-based oscillators [64], accelerometers [65], and switches [66], for example. Only recently have such subsystems been developed, so gaps in the related CAD framework are not surprising. Clearly, a design flow that supports the convergence of these technologies is required if complete systems are to incorporate these research breakthroughs. Past MEMS development has typically been ad-hoc and bottom-up in nature. This design approach is an outgrowth of both the disparate nature of MEMS technologies and the fact that most MEMS work to date has been focused on device development. After device optimization, supporting electronics are added incrementally. These devices are now appearing in larger systems and the typical development strategy is to partition sections of the microsystem into the mechanical, analog, or digital domains. With initial efforts focused on the hardware, software development is often neglected at this early stage of the design. Design activities can become disjoint and ad-hoc, with each subsystem being designed with a separate tool suite and with little, if any, cross-domain verification.

As discussed previously, MEMS technology has been designed almost exclusively with finite element tools; however, the majority of these simulators do not support an interface with a standard IC framework. Therefore, in almost all applications, some level of model extraction and abstraction is required for simulation of the MEMS component with the supporting analog electronics. Often, this extraction is tailored to each component and it must be completed by the designer without the aid of design automation. Additionally, some of these systems require logic for programming or trimming, and in many applications, a complete embedded processor is required to support the system. Here the standard tool suites allow designers to synthesize digital logic and physical design from a hardware

description language, but chip verification with the integrated analog and MEMS devices is typically not performed.

As Fig. 4.1 and the previous discussion illustrate, a variety of CAD tools are required for the development of microsystems. A specific design challenge is system verification across these various design platforms. Clearly, the complexity of microsystem design is significant. Attention to design methodology has become increasingly important in order to develop systems efficiently and close the gap between manufacturing and design capabilities [67].

4.1.2 Design Methodology Comparison

4.1.2.1 Typical Design Methodology: Bottom-Up

A bottom-up design methodology involves the development of each block from the device to system level. Devices are combined to form blocks, which are then combined to complete and verify the system. In [68], the problems associated with a bottom-up methodology are addressed. They include lack of architectural study and optimization, costly redesign effort associated with iteration through the flow, and significant processing time for system-level simulation, if it is even possible. Software development of the compiler and application code typically occur independently of the hardware design flow. This discontinuity between the software and hardware flows leads to non-optimal system performance as a result of functionality or resource limitations and power or performance bottlenecks that inevitably manifest during the application deployment and system integration stages.

Fig. 4.2 illustrates this typical design methodology as applied to microsystems technology. Here a system specification is partitioned into one software domain and three hardware domains: digital, analog, and mechanical. The software flow is relatively straightforward. Assembler and compiler development can begin once the instruction set architecture (ISA) has been formulated. Then the compiler and application code can be written and iteratively debugged. Hardware design activities progress from the device to block level and from the block to system level. A macro is delivered from each domain, and the system is assembled with an automatic place and route (APR) tool. The system is then verified and only at this point are system-wide integration problems identified. Time-con-

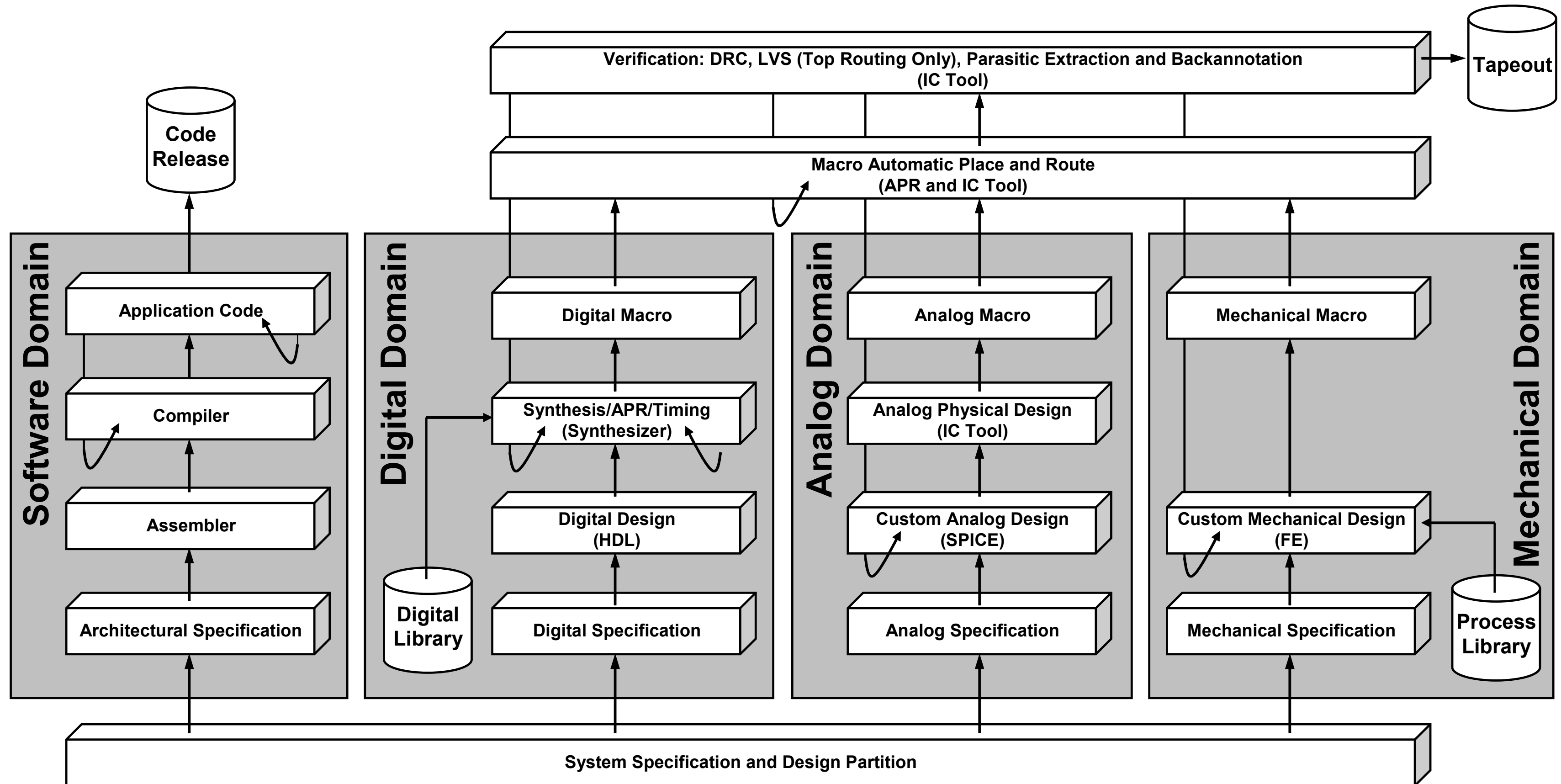


Figure 4.2: Typical ad-hoc and bottom-up microsystems design methodology.

suming redesign is often required at lower levels along with APR iterations to optimize macro placement.

Although this methodology has been employed in the past, it is clearly insufficient for complex microsystems. As the field matures, microsystems will likely contain several, if not hundreds, of magnetic, mechanical, optical, chemical, or biological components along with the supporting analog and digital electronics. A proper, efficient, and exhaustive design methodology and framework are required to support increased levels of integration. Hardware-software codesign methodologies must be adopted early in the design flow to ensure that software runs efficiently with minimal power.

4.1.2.2 New Design Methodology: Top-Down

In the top-down approach [69], hardware development would proceed from the system to device level. The system hardware could be studied and optimized with a mixed-signal hardware description language (MS-HDL) from which the abstract circuit blocks would be derived. Device-level designs would then be completed, and achieved performance could be benchmarked against the original specification using the abstract blocks and system model. Software design should proceed in parallel with the hardware, but with cross-domain performance evaluations to pinpoint architectural shortcomings.

A major goal of this top-down approach was to achieve hardware abstraction and cross-domain simulation for MEMS, analog, and digital electronics at every level. The environment also needed to support simulation of abstract hardware with device primitives in order to accurately model digital programming of analog and mechanical components before synthesis of the digital circuit blocks. A model that could be modified easily for system verification based on the realized subsystem performance was desirable. The complete tool suite had to support low-level simulation including FE and basic transistor-level analysis, as well as non-linear RF and noise analysis. Support for HDL synthesis, timing verification, and automatic place-and-route was mandatory for digital design and final chip assembly.

4.1.3 Top-Down Microsystem Design Flow

4.1.3.1 Hardware Design Flow

Although no single design framework addresses all the integration challenges presented by this complex embedded system, the Cadence AMS, or Analog Mixed Signal, environment is well-suited to achieving many of these goals for system-level development of microsystems hardware. The Cadence AMS environment supports Verilog-AMS, an analog and digital HDL which is a superset of the Verilog and Verilog-A languages. Verilog-AMS is also able to perform behavioral modeling of mechanical devices. Prior to the emergence of Verilog-A, many MEMS engineers had used device-level models, including primitives, for MEMS component modeling. Clearly, the Verilog-A language is a significant improvement over this technique, as it provides added modeling flexibility while minimizing complexity. Additional tools used in this work included Spectre for analog subsystem and transistorlevel design, Coventorware for FE analysis of MEMS components, Synopsys for digital synthesis and timing analysis, Cadence First Encounter for automatic place and route (APR), and Mentor Graphics Calibre for design rule check (DRC) and layout versus schematic (LVS). The requirement of such an extensive and disparate tool suite is a significant challenge faced in the development of microsystems technology.

The detailed design methodology that was developed to build this microsystem is illustrated in Fig. 4.3. Verilog-AMS was employed to realize the system specification. MEMS and analog components were modeled in Verilog-A, while the microprocessor core and digital peripherals were modeled in Verilog. From this system model, a natural partition of top-down subsystem design activities followed. Each block was specified with an abstraction for the hardware. In parallel with behavioral verification of the digital section, the blocks in the mechanical and analog domains were developed and performance metrics were determined. Updated Verilog-A was developed to model achieved performance from FE simulation in the mechanical domain, while device-level design and analysis using Spectre led to achieving the analog specification. A complete behavioral description of the digital electronic hardware was realized. At this point, the first cross-domain verification of the system was performed. Once the HDL from each domain had been updated with the achieved performance, verification of the system model was trivial. In the Cadence AMS

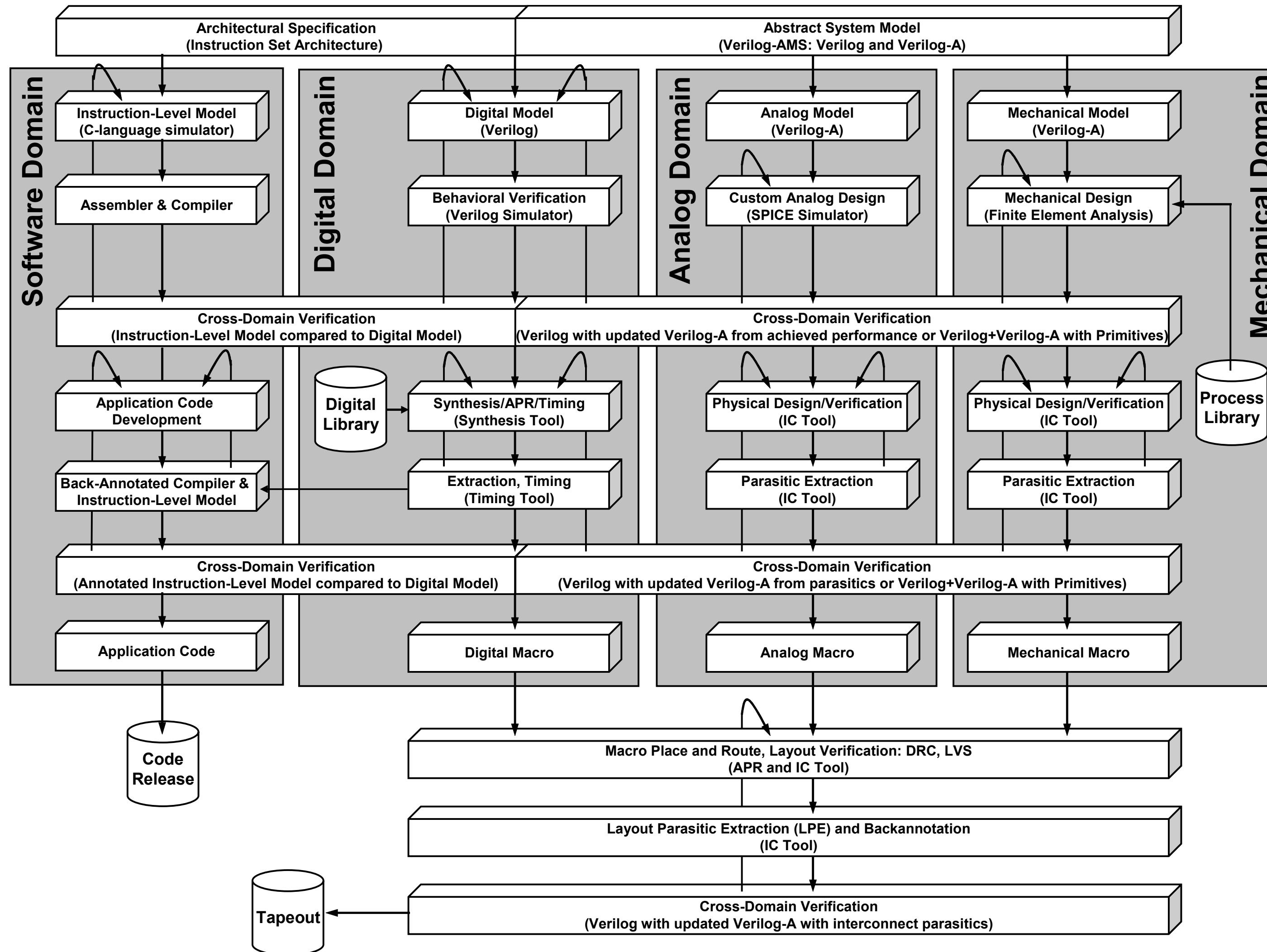


Figure 4.3: The top-down microsystems design methodology.

environment, HDL and primitives may be mixed, and critical subsystem performance metrics can be determined quickly with a detailed model for the subsystem and an abstract model for the remainder of the system. This was particularly significant when considering analog and MEMS device-level performance that required digital programming which was described only in HDL.

A system-wide simulation was completed and iteration in the mechanical and analog design activities continued, as required by system performance specifications. This first cross-domain simulation offered significant benefits over the bottom-up methodology described previously. First, design effort had not been expended synthesizing the digital electronics. Second, iteration in the design of the MEMS and analog circuits occurred early in the design flow. Finally, the system simulation was fast, as it was described in behavioral HDL, rather than by a device-level netlist. Simulation was also timely in the case of a primitive-level subsystem simulation, as the remainder of the system was described by HDL and only the critical blocks were modeled at the device-level.

System development continued with a typical physical design flow. The digital sections were synthesized and the mechanical and analog sections were custom designed. Timing information from the synthesis tool was used in iteration to achieve digital timing closure. Similarly, parasitic extraction and back-annotation afforded an iterative process for completing the mechanical and analog sections. Once timing closure was satisfied in each domain, a second cross-domain simulation was executed for system verification based on physical design. Again, the HDL for the subsystems was updated, and system simulation was timely and accurate. Physical design iteration continued until timing closure was achieved for the complete system. The domain-specific design activities were completed with the delivery of a hard macro.

The final hardware development activities included APR, physical design verification (DRC, LVS), layout parasitic extraction (LPE), and back-annotation. A final cross-domain verification was completed following parasitic extraction. APR iteration was also necessary.

4.1.3.2 Software Design Flow

To reduce software design time, prevent costly hardware revisions, and achieve the best possible microsystem performance and power dissipation, software development should proceed in a parallel, tightly coupled fashion with the hardware design flow. Fig. 4.3 shows that software and digital hardware development began with an architectural specification in the form of an ISA. This laid forth the general machine architecture and the proposed instruction set for the microsystem. The ISA should be a joint effort of the hardware designers, compiler developers, and software engineers. Careful consideration was given to the final microsystem application(s) to ensure that the necessary instruction and hardware support were provided. Compiler input was essential at this early stage to avoid inefficiencies in the instruction set.

A C-language instruction-level model of the microsystem was developed to provide the compiler and application developers with a convenient platform to evaluate their software. This C-simulator modeled microsystem behavior at a higher level of abstraction than the behavioral Verilog model used in the digital domain. With the C-model as a development platform, both an assembler and compiler were written to support the WIMS ISA. At this stage, the first cross-domain verification took place between the software and digital domains. Focused and random test cases were run on both the C-model and Verilog model and any discrepancies in machine state were resolved. Thus, the C-model served to verify functionality of the behavioral Verilog prior to synthesis. By annotating the C-model with preliminary performance and power estimates, compiler developers were able to suggest architectural enhancements or add new instructions to the ISA in order to improve software performance and reduce power. Iteration of both the C-model and Verilog model were required to implement changes as a result of this cross-domain verification.

Following the initial cross-domain verification step, the machine architecture was frozen and application software development efforts began. Once the hardware extraction step was completed in the digital domain, the extracted parasitics were used in conjunction with Synopsys Nanosim simulations to estimate an average energy-per-instruction (see Section 5.1.4). The average instruction energies were annotated into the C-model and used by the compiler to further optimize compiled code. Iteration of the application code and compiler was required to achieve the minimum application power. For the second cross-

domain verification, the annotated C-model was compared against the annotated structural Verilog to ensure that the Verilog machine state still matched the C-model. Test code was run on both the C-model and the Verilog model to correlate the power estimates. Using the C-model, application development was able to continue until the hardware had been fabricated and tested.

4.1.3.3 Digital Design and Verification

An extensive verification environment was designed using Perl scripts to facilitate functional verification of the digital core. Focused assembly language test cases were used to test the basic operation of each instruction in the ISA, as well as anticipated corner cases. Each test was assembled and run on our cycle-based behavioral Verilog model using Cadence-XL's Verilog simulator. The same test case was executed on an instruction based C-simulator model of the processor. Important register values were dumped by each simulator for every instruction executed, along with the data stored in memory at the end of the simulation. By comparing model dumps, bugs could be isolated and corrected.

A random assembly code generator was designed and used to generate millions of lines of random test cases. These detected functional bugs that might have been missed in the focused test cases, particularly any unexpected interdependencies between instructions as they progressed down the pipeline. Approximately 30 million lines of assembly code were executed on each of the simulators for functional verification of the Verilog. Additional test cases were written with the sole purpose of verifying interrupts and the timing of peripherals such as the USART, SPI, and timers.

The same level of verification mentioned above was completed after logic synthesis in Synopsys and again after APR in Cadence Silicon Ensemble, effectively guaranteeing that functional bugs were not introduced by the synthesis and APR tools. PrimeTime was used for static timing analysis with back-annotated parasitics to ensure that all paths met timing specifications. DRC and LVS verification were performed for each sub-block and at the top level using Calibre.

Matlab Simulink was used to model the DSP filters and data processing for both verification of the CIS algorithm and comparison to the DSP RTL code. A model of the ADC converted audio files into samples for processing and a cycle-by-cycle output state

from each stage was compared to the RTL simulator output. The Matlab model read the same filter coefficients used by the RTL. It also had the capability to generate filter coefficients for a set of particular desired filter characteristics. Fig. 4.4 and Fig. 4.5 show a sample input waveform and the corresponding filter output for a single channel.

Similar to the C model for the MCU, a C model of the CIS DSP was written to confirm operation of the RTL. The C model was needed in addition to the Matlab model because it was much simpler to implement the control and stimulation components of the DSP system in C than in Matlab. The Matlab and C filter functions were confirmed to be identical throughout the design and verification process.

4.2 Conclusions

A full chip mixed signal design methodology has been outlined and described using the WIMS SoC as an example design. It is a many faceted problem that utilizes many tools

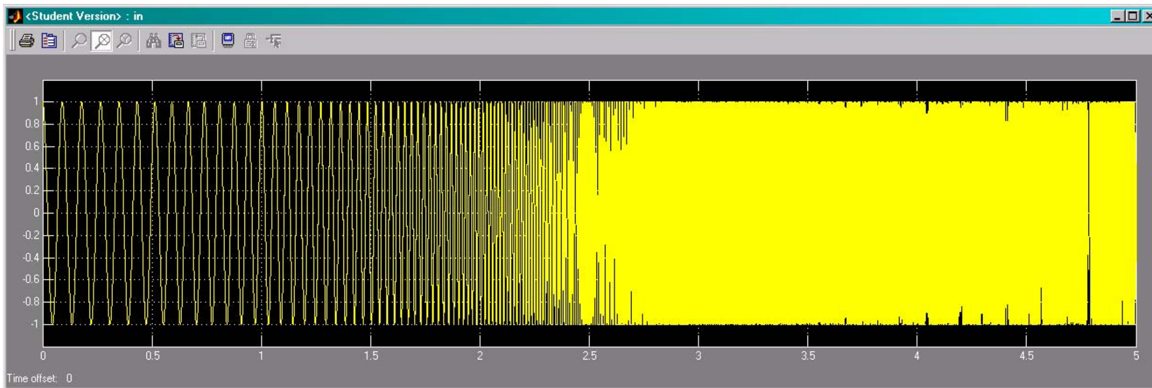


Figure 4.4: Matlab Simulink DSP model input waveform.

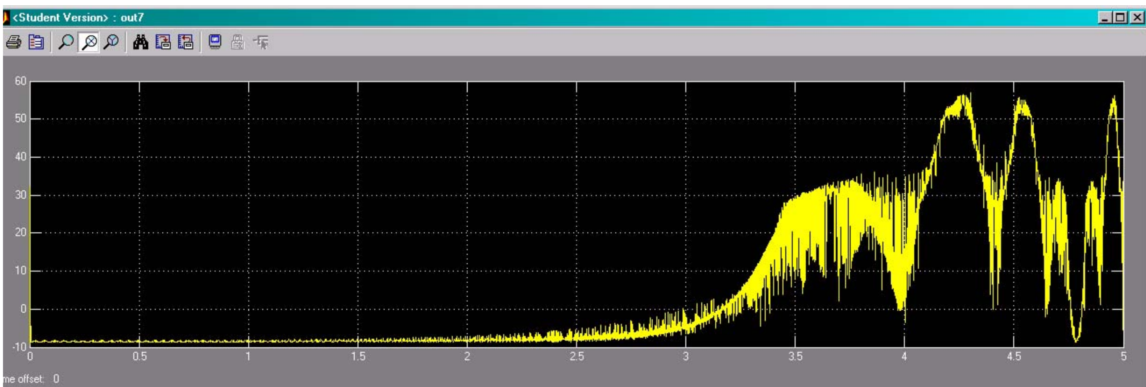


Figure 4.5: Matlab Simulink DSP model channel seven output waveform.

and languages to cover the design, implementation, verification, and fabrication of the design. By leveraging advances in mixed-signal and IC design tools, we developed an efficient top-down design flow that promotes hardware-software codesign. The design methodology has been employed in the development of a complete microsystem with a dedicated power-aware compiler as part of a larger application framework.

During the development of this design methodology and SoCs an intellectual property (IP) repository was created in order to hold and distribute design data, flows, and IP among researchers [70], [71]. The site has successfully distributed IP to dozens of researchers at several institutions. The time and effort to develop a repository such as this is more than made up for by the time saved using the contents.

Chapter V will present the post-fabrication measured silicon results for the WIMS Microsystem that was built using the methodology described in this chapter.

CHAPTER V

MICROSYSTEM RESULTS

After completing the design and implementation of a Microsystem as complex as the one described here, seeing a functioning piece of silicon that works as expected is a noteworthy achievement. It is a difficult task and required some iterations to get correct, but is satisfying nonetheless. This chapter describes the measured results and completed milestones obtained by this work[1].

5.1 Microsystem Measured Results

Fig. 5.1 shows the microsystem fabricated in TSMC 0.18μm mixed-mode bulk CMOS containing 2.3 million transistors and occupying 9.18mm^2. The dies were packaged in 128 pin PGA packages. The major system level blocks are outlined in the die micrograph. A significant portion of the silicon area is occupied by the on-chip memory, as is typical in digital chips. The DSP is the next largest component due to the many parallel channels and storage required for filter coefficients. The pipeline and peripherals are followed by the clock generation IP in terms of area consumed. This section details the measured results.

5.1.1 Post-fabrication Testing

Functional testing of the digital core was performed using a 400/200MHz in-house HP82000 digital tester equipped with a customizable Design Under Test (DUT) board

---

1. The author would like thank Rob Senger and Alan Drake for all their assistance and mentoring with the HP82000 equipment.

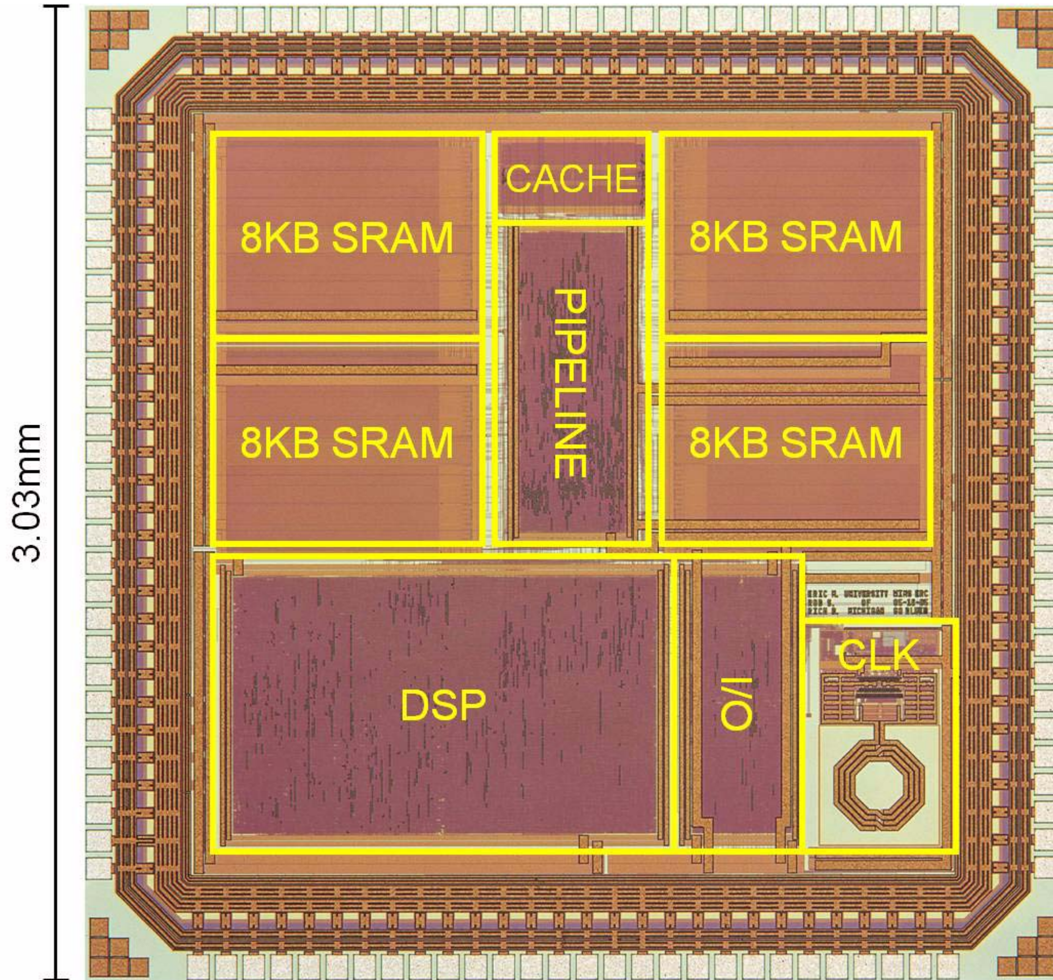


Figure 5.1: Die micrograph of fabricated microsystem.

shown in Fig. 5.2. Verification of the design was done by generating test vectors from Verilog simulations and running them on the tester. The same Verilog simulations used in the design validation phase were used in the post production testing phase, along with other custom built test cases to measure silicon performance. Instructions were loaded both from the USART and external memory interface. Performance characteristics were measured using the other external equipment shown in Fig. 5.2.

5.1.2 Microsystem Results

Table 5.1 shows the measured power consumption for different components of the Microsystem under different operating conditions. This data is taken after a focused ion beam (FIB) fix to some memory control signals in order to save current on inactive memory

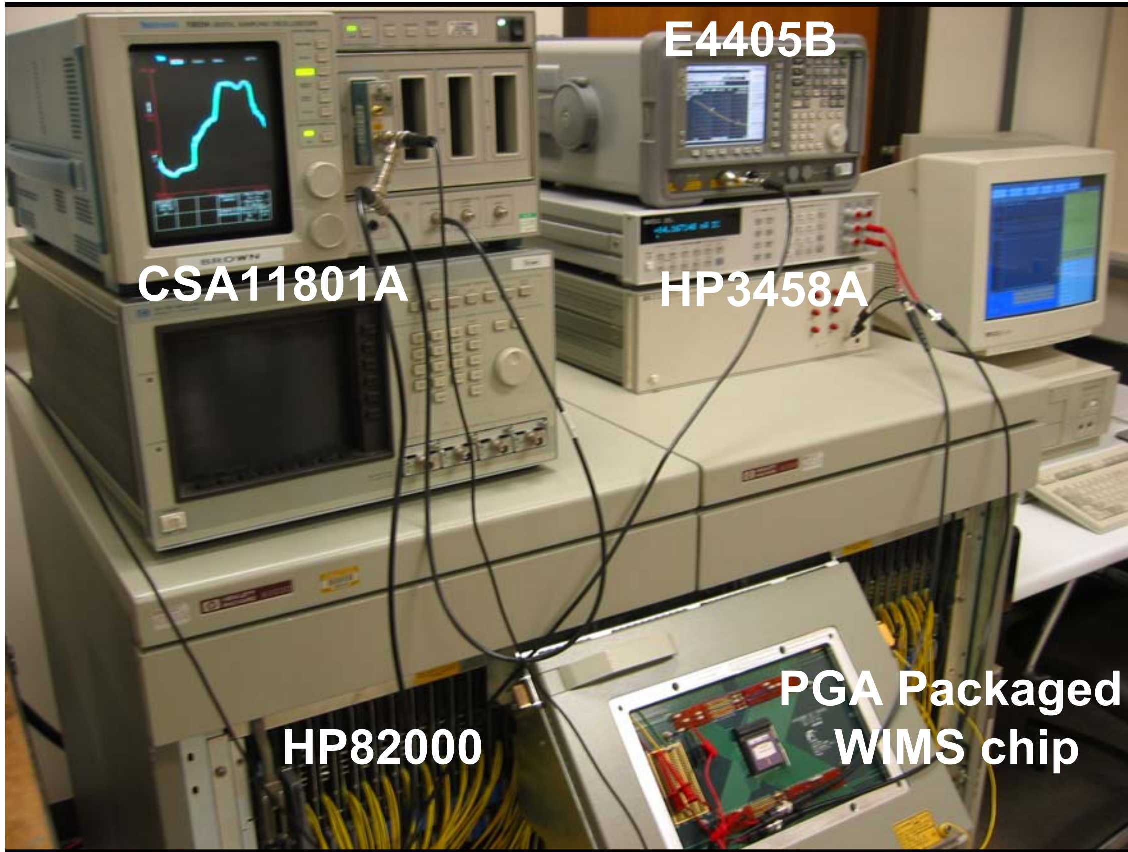


Figure 5.2: HP82000 test setup.

banks. It shows the wide operating capabilities of the MCU from 36.60mW at 100MHz and 1.8V to 1.67mW at 1.2V with 330µW standby power consumption. A significant portion

Table 5.1: Measured power for different operating conditions.

Component	V_{DD} = 1.8V			V_{DD} = 1.2V		
	100MHz	DSP Mode[a]	Standby	1MHz	DSP Mode[a]	Standby
MCU Core (mW)	16.69	1.63	0.54	0.31	0.44	0.10
Memory (mW)	7.83	0.12	0.12	0.04	0.03	0.03
DSP (mW)	2.46[a]	2.46	0.27	1.14[a]	1.14	0.06
Clock (mW)	9.62[b]	0.76[c]	0.76[c]	0.18[c]	0.18[c]	0.18[c]
Total (mW)	36.60	4.97	1.69	1.67	1.79	0.33

a. DSP operating frequency is 3MHz. Other components operating at speed necessary to support DSP functionality.

b. *LC* oscillator is operating, ring oscillator is off.

c. Ring oscillator is operating, *LC* oscillator is off.

of the standby power consumption is from the on-chip clock source. The expected power consumption for the DSP running at the maximum stimulation rate is 1.79mW.

Each major block of the microsystem is shown in Table 5.2 and important statistics are called out. The IO is not called out specifically, but makes up the remainder of the total numbers. The memory and DSP make up more than 1/3 of the chip area.

5.1.3 Microcontroller Results

The microcontroller and peripherals perform system control and a majority of the processing when not in DSP mode. Fig. 5.3 shows power versus VDD curves at several different operating frequencies. The core will not operate at 100MHz below 1.45V. Significant system level power savings can be obtained by using the on-chip selectable clock sources in order to optimize the power versus frequency trade-offs based on the workload requirements. It is up to the software to detect when higher workloads are required based on interrupts from external interfaces or commands from these interfaces.

5.1.3.1 Scratchpad Memory

Fig. 5.4 shows the power savings obtained by the compiler taking advantage of the scratchpad memory for several benchmark programs [72], [73] running at 100MHz and utilizing the scratchpad memory for data or instruction storage [38]. A single access to the scratchpad consumes 45% of the energy that an access to SRAM consumes. As expected, a higher percentage of scratchpad accesses yields a higher energy savings. A total savings of anywhere from 4 to 20% can be obtained.

5.1.4 Energy Per Instruction

As battery-operated embedded systems proliferate, the need for good system level power modeling increases. Modeling the power dissipated by an arbitrary software pro-

Table 5.2: Chip statistics breakdown.

	Pipeline	Peripherals	DSP	CLK	Memory	Total
Area (μm^2)	439,365	366,887	1,274,374	251,940	2,250,148	9,138,529
Transistor Count	102,527	93,048	376,903	634	1,700,126	2,279,617
Decoupling Cap (pF)	153[a]	153[a]	418.2	382.5	724.2	-

a. The Pipeline and Peripherals (I/O) share a power supply, and therefore the 153pF.

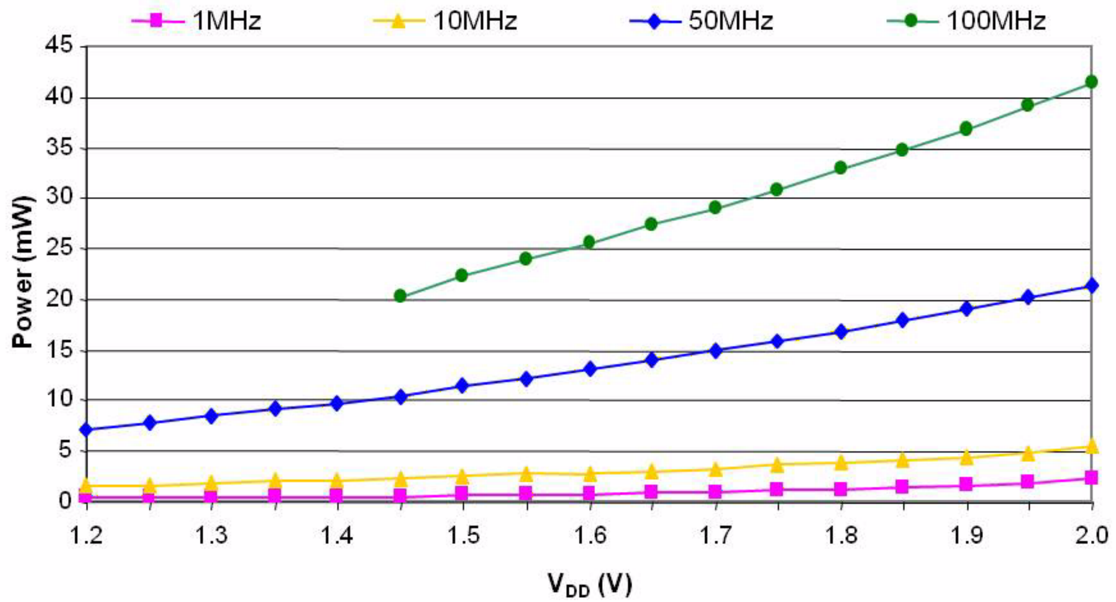


Figure 5.3: Power versus VDD scaling for the MCU and Memory.

gram running on system hardware is not challenging. Various high-level architectural modeling tools exist, but as is typical of high-level models, they lack accuracy. Lower-level power modeling tools, typically built on parasitic annotated netlists and switching stimuli, are more accurate, but are too slow for modeling large chips, and totally impractical for comparing different software optimizations. To quickly estimate the energy consumed in executing a given software program with enough accuracy to support compiler optimizations, a new approach was needed.

Instruction level power modeling is a method of calculating a program's total energy by summing the energies of each individual instruction. Because instruction energies are typically calculated using DC current measurements of actual hardware, the predicted program energy is more accurate than anything from high-level simulators. Because the instruction energies are simply summed to produce the final program energy estimate, instruction level power simulators are fast. Inevitably, architectural factors such as pipelining (stalls) and memory hierarchy complicate the measurement of instruction energies. Switching activity, which varies with data and with instruction ordering, will also affect the energy. Prior work on instruction level modeling has derived instruction energies from actual chip power measurements. While this has worked well if hardware is available, it cannot be used to guide the hardware design phase. As part of the WIMS microcontroller

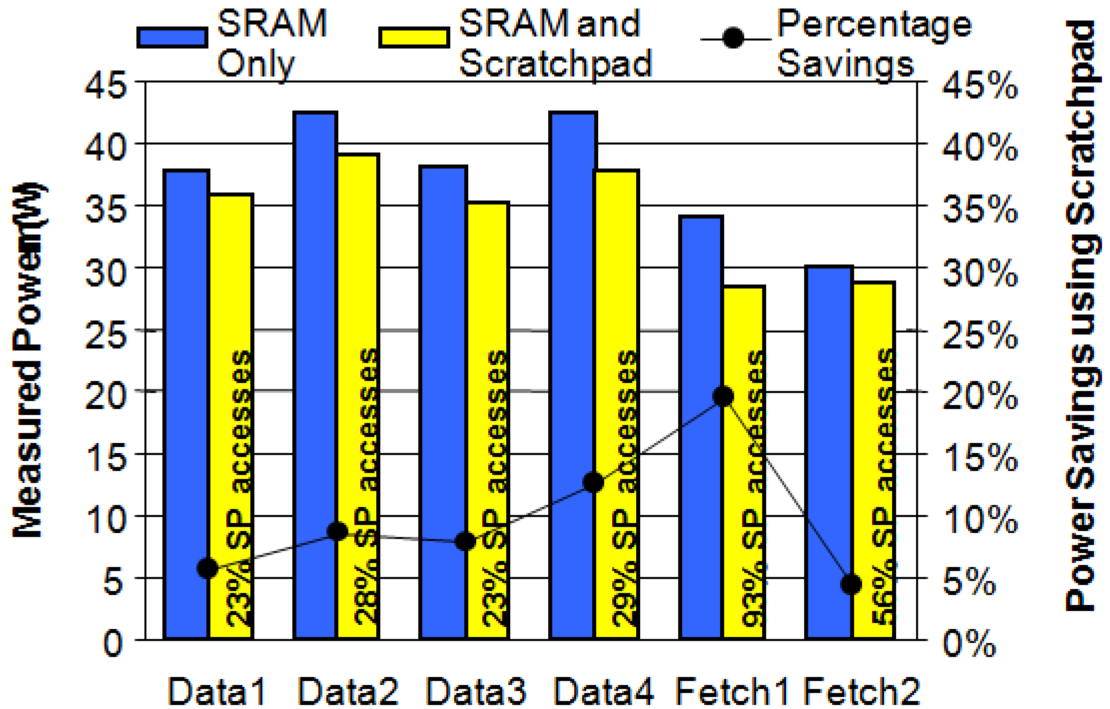


Figure 5.4: Power savings utilizing scratchpad memory or loop cache.

project, a simple instruction-level power model was developed and demonstrated in both simulation (pre-fabrication) and in hardware (post-fabrication) [74].

A complete framework for instruction level power modeling was first developed by Tiwari, Malik and Wolfe in [75]. They validated their proposed methodology using two off-the-shelf microprocessors, the Intel 486DX2 and Fujitsu SPARClite 934 [76]. Tiwari asserts that instruction power can be broken down into base cost, circuit state overhead, stall cost, and cache miss cost. Instruction base cost can be defined as the minimal energy dissipated by an instruction during execution. To calculate base cost, an instruction is run in an infinite loop and the current is measured. However, running the same instruction repeatedly does not accurately model typical program execution because it generates very little processor switching activity. Tiwari refers to this switching current as circuit state overhead, and considers it separately from the base cost. Similarly, any additional current contributed by pipeline stalls or cache misses is measured separately and added to the base cost to produce the total average current required to execute each instruction.

Circuit state overhead, also commonly known as dynamic switching power, is added to the base cost to model the increased switching activity that occurs when two different instructions execute sequentially. To calculate this extra power required by pairs of dissimilar instructions (or data), Tiwari ran test cases with various instruction sequence pairs and determined that the circuit state overhead was typically 15mA for the Intel 486DX2 processor. Thus, totaling the base costs for each instruction plus 15mA was usually sufficient to account for circuit switching overhead. The 15mA number was obtained after extensive measurements of instruction pair combinations, and must be determined separately for each processor type. In [77], Tiwari presents a more detailed analysis of the SPARClite 934, with extensive simulations of instruction pairs to better model circuit state overhead. Although the method yields good results with typically less than 3% error, the number of measurements required is prohibitive for large instruction sets. Complexity can be reduced by grouping similar instructions. For an instruction set with n instructions (or instruction groups), n test cases are required for base cost and $n!/(2(n-2)!)$ test cases are required to measure circuit state overhead for every possible instruction pair combination. Thus, $(n^2+n)/2$ test cases must be written and the current measured for each run. For n=30 instruction groups, which is a relatively small number for modern processors, that is 465 separate test cases. We developed a more efficient estimation methodology that requires only $O(n)$ test cases while still providing a reasonable accounting for circuit switching energy.

In [78], Klass proposes a NOP model to reduce the number of test cases required to estimate the circuit state overhead from $O(n^2)$ to $O(2n)$. The NOP model is based on the assumption that the overhead energy for an instruction is not strongly dependent on the adjacent instruction, but depends on whether the adjacent instructions are the same or different from each other. Similar to Tiwari, Klass' NOP model calculates a particular instruction's energy as either the base energy or the base plus the overhead energy, depending on whether the previous instruction was the same or different than the current instruction. The NOP model differs in that only one overhead energy is calculated for each instruction instead of separate energies for every combination of instructions. To do this, Klass alternates the target instruction with ‘no-ops’ in the program loop. Instructions are not grouped according to similar function, so n test cases are required for the base energy and another n

test cases for the overhead energy. Demonstrating his method in simulation only, Klass reports an error of less than 8% in modeling the transition energy (circuit state overhead) between any two instructions.

5.1.4.1 Experimental Hardware Methodology

Similar to Tiwari's method, our instruction level power model can be derived using current measurements taken from a fabricated microprocessor core. However, there are several important differences between the methodologies that will be highlighted in this section. Our method calculates the energy per instruction (EPI), rather than an instruction's average current, which was the primary metric used for analysis in [75]. EPI is a more illustrative metric for comparison than average current as many multi-cycle instructions demonstrate relatively low average current draw but take multiple CPU cycles to execute, and thus require more energy than single-cycle instructions which might have higher average current, but for only one cycle.

To measure the instruction energies, a laboratory ammeter, power supply, digital tester, and chip evaluation board are required. The ammeter must be connected in series with the test chip's core power supply pins. A separate ammeter may be used for the I/O power supply (or any additional power domains), if desired. This work focuses on core power only. To achieve accurate current measurements, test cases must be looped long enough for the ammeter to provide a stable reading.

Perhaps the most difficult and time-consuming part of measuring instruction power is developing the suite of test cases. To avoid writing a separate test case for each instruction, instructions can be organized into groups based on the functional units they exercise. For example, in many processors, 'add' and 'sub' instructions use almost identical hardware and can be grouped together. Intelligent grouping is critical to minimize the number of test cases, especially for complex processors having hundreds of instructions. However, knowledge of a processor's architecture is necessary to properly group instructions.

Unlike [75], which repeats the same instruction type consecutively within the program loop, our method inserts ‘no-ops’ between each instruction contained in the program loop. An example program is shown in Fig. 5.5, with Part B containing the instruction-no-op sequence. By inserting ‘no-ops’ between each instruction, switching activity is

```
loop_start:
        // Part A: initialize registers and memory data
        ldbi r0, 0xff
        ldbi r1, 0xa0
        . . .
        . . .
        // Part B: measure current for instruction group
        add r0, r1
        noop
        add r1, r3
        noop
        sub r2, r1
        noop
        sub r4, r0
        noop
        . . .
        . . .
        jmp loop_start  // Part C: infinitely loop test case
```

Figure 5.5: Sample assembly loop for energy per instruction measurements.

increased to a level that more accurately parallels switching activity generated by a normal program. [75] neglects switching activity in the base instruction cost, and models it by adding a circuit state correction factor. To get accurate circuit state correction factors, all possible combinations of instruction groups must be executed and the current measured. As derived previously, for an *n* group instruction set, $(n^2+n)/2$ test cases would be required to calculate the base cost and circuit state overhead for every instruction combination. This method builds the switching energy directly into the instruction base cost by using interleaved 'no-ops' to force switching activity. The result is that only *n* test cases are required, thus greatly reducing the time required to estimate instruction energy. Another instruction could probably be used in place of 'no-ops'; however, 'no-ops' are an obvious choice, as they avoid unwanted data hazards that would result in pipeline stalls. It is important to avoid pipeline stalls and interrupts when measuring average instruction energy. Separate test cases can be written to measure the energy dissipated by stalls, and this can be factored in later.

Another difference between Tiwari's method and the method used in this work involves the program loop. Tiwari suggests that the program loop contain enough instruc-

tions so that the jump at the end of the loop will have minimal effect on the measured instruction current. In the WIMS power estimator, this potential source of error was eliminated by creating a boot-up case that is essentially a shell of the main program loop for each instruction test case. It includes Parts A and C from Fig. 5.5, but omits Part B. The boot-up case is looped repeatedly and the energy is measured and subtracted from the measured energy of the corresponding instruction test case, which has Parts A, B, and C. This has the effect of subtracting the jump instruction's energy from the total loop energy. Also, setup instructions can be included in the program loop (Part A) and subtracted out using the boot-up case. This is helpful when setting up data values to be used by the looped instructions. Each time through the loop, the data is re-initialized so the same values are used consistently, resulting in more stable current measurements. Creating the boot-up test cases from the individual instruction test cases is trivial because all it requires is deleting Part B from the program loop. Through our experiments, we found that many test cases can share the same boot-up case. Even using a boot-up test case, experiments show that Part B should contain at least 50 instructions (excluding ‘no-ops’) to ensure a sufficiently diverse assortment of data and address switching.

To calculate the energy required per instruction group, the test case's average current must first be converted into energy by multiplying the CPU time required to execute the loop, t_{loop}, by the measured current value, I_{loop}, and by the supply voltage, V_{DD}. The result, E_{loop}, is the total energy, as shown in Equation 5.1, required to execute the entire program loop.

$$E_{loop} = I_{loop} t_{loop} V_{DD} \tag{5.1}$$

However, to calculate the EPI as shown in Equation 5.2, the energy from Parts A and C must be subtracted from the total. This is accomplished by converting the corresponding boot-up test case's measured current into energy, E_{init}, and subtracting. The remainder, E_{partB}, is the energy dissipated by only Part B of the loop, which is composed of no-op energy plus the energy dissipated by the instruction of interest.

$$E_{partB} = E_{loop} - E_{init} \tag{5.2}$$

To calculate ‘no-op’ energy, a separate ‘noop' test case is written that contains hundreds of random instructions. ‘No-ops’ are inserted at random locations in the test case, and the test case is executed and current measured and energy calculated both with, $E_{randnoop}$, and without, E_{rand}, ‘no-ops’ inserted. The difference divided by the number of ‘no-op’ instructions, $n_{randnoop}$, that were executed gives the average energy per ‘no-op’, E_{noop}, in Equation 5.3.

$$E_{noop} = \frac{E_{randnoop} - E_{rand}}{n_{randnoop}} \tag{5.3}$$

Equation 5.4 shows that the ‘no-op’ energy can then be multiplied by the number of ‘no-ops’ in Part B, $n_{partBnoop}$, of the program loop and subtracted. Dividing the result by the number of non ‘no-op’ instructions in Part B, n_{target}, gives the EPI with average switching included, E_{target}.

$$E_{target} = \frac{E_{partB} - E_{noop} n_{partBnoop}}{n_{target}} \tag{5.4}$$

This is different from [75], [77] which estimated switching power separately from the base instruction cost.

Depending on machine architecture, additional measurements might need to be performed. Conditional branch instructions exhibit different energies depending on whether the branch is taken or not. Separate test cases must be written to measure the branch-taken and not-taken energies. Hierarchical and banked memory architectures add an additional layer of complexity to the energy model. The same instruction, when fetched from different size memory banks or caches, will dissipate different energies. For this reason, a memory correction factor is required to either add or subtract energy from the base instruction cost. The memory energies can be determined using separate test cases that exercise the different memory banks. For cached memory hierarchies, the cache-hit energy and cache-miss energy must both be determined.

A similar methodology can be implemented using simulation platforms, like NanoSim and UltraSim, in order to estimate the EPI of components that are in the design phase. However, these energies will only be as accurate as the simulators. The benefit is that data

can be gathered sooner and used to make design decisions as well as compiler optimizations to improve the hardware-software co-design process.

5.1.4.2 Hardware Energy Per Instruction Results

Each test case was looped infinitely on the digital core, and current measurements were recorded. Based on these measurements, the energy per instruction group was calculated, as shown in Table 5.3. The energy values include the energy required to fetch the instruction from the on-chip main memory banks. The right column contains the effective execution time at 100MHz for each instruction, which is not the same as the time an instruction spends in the pipeline. Because a pipeline has multiple concurrently executing instructions, the time column should be calculated by taking the ideal clock cycles per instruction (CPI) of the instruction being executed (assuming no data hazards), and multiplying by the clock period. Although an 'add' instruction takes 30ns to run through the three-stage WIMS pipeline at 100MHz (assuming no stalls), the ideal CPI is 1, and thus the effective execution time assigned to the 'add' for the purposes of energy calculation is 10ns.

The results in Table 5.3 provide several interesting insights into the power efficiency of the WIMS ISA. Instructions using absolute addressing require significantly more energy than relative addressing because of an extra instruction fetch to retrieve the full 24-bit address. Memory bit manipulation instructions such as 'test-and-set bit' are costly at 1.10nJ, because of two memory accesses; however, they are more efficient than breaking the instruction into its subcomponents. A separate 'load-relative', bit mask (boolean), and 'store-relative' instruction sequence would require 0.66+0.38+0.55=1.59nJ. Although multiply and divide instructions consume almost 5nJ of energy, they are significantly more energy efficient than full 16-bit multiply or divide emulation using 'add', 'sub', 'shift', 'compare', and 'branch' instructions. However, if one of the multiplicands is known in advance and is close to a power of two, it would require less energy to decompose the multiply into shifts and adds. For example, to multiply a number n by a known constant 10 ($10=2^3+1+1$), a shift left of n by 3, followed by two summations of n costs only 1.21nJ compared to 4.99nJ for a multiply instruction. With knowledge of these energy numbers, a compiler can make appropriate decisions about when to use multiply or divide instructions versus shifts and addition or subtraction. ISA designers will observe that for some com-

Table 5.3: Energy per instruction.

Instruction Group	Energy (nJ)	Time (ns)
add-sub	0.43	10
shift	0.35	10
boolean	0.38	10
compare	0.37	10
multiply	4.99	180
divide	4.89	180
copy	0.38	10
bit	1.10	20
load abs	0.94	20
load rel	0.66	10
store abs	0.80	20
store rel	0.55	10

Instruction Group	Energy (nJ)	Time (ns)
win swap	0.33	10
load imm	0.35	10
branch-nt	0.31	10
branch-t	1.03	30
jmp abs	0.97	30
jmp rel	0.72	20
jmp abs sub	1.02	30
jmp rel sub	0.63	20
return	0.67	20
swi	1.01	30
--	--	--
no-op	0.35	10

monly executed operations, dedicated complex instructions can save enough energy to justify the additional hardware overhead.

Table 5.4 shows the measured memory energy correction factors for different memory blocks on the WIMS MCU. They were determined by modifying test cases used for Table 5.3 to access different memory blocks. These numbers are negative because they are calculated relative to the access energy of the higher-power on-chip SRAM. The first column shows the different types of memory accesses that were measured. Only the first row, instruction fetch, contains fetch energy (per word of data); the remaining rows measure the energy for data (load/store) accesses only. The numbers in the second column do not include the energy required to access the external memory itself, but only to drive the values through the external memory bus and associated control logic. External memory energy will depend mostly on the size and configuration of the external memory modules. The third column shows the energy to access the on-chip scratchpad memory. The fourth column is for memory-mapped register accesses (MMR), which only apply to load/store data transfers, not fetches. The final column shows the energy required to fetch instructions from the boot ROM.

All instruction energy measurements from Table 5.3 were performed using the on-chip SRAM for both instruction and data memory. To account for different access energies for the other memory blocks, the correction factors from Table 5.4 are used to adjust the

Table 5.4: Energy memory correction factors.

Memory Access	Ext Mem (nJ)[a]	Scratchpad (nJ)	MMR (nJ)	Boot ROM (nJ)
instruction fetch	-0.11	-0.10	N/A	-0.08
mem bit set/rst[b]	-0.30	-0.30	-0.34	N/A
load absolute[b]	-0.18	-0.18	-0.16	N/A
load relative[b]	-0.19	-0.19	-0.20	N/A
store absolute[b]	-0.07	-0.08	-0.08	N/A
store relative[b]	-0.09	-0.11	-0.10	N/A

a. Excludes memory access energy as this is memory dependent

b. Fetch energy counted separately

values in Table 5.3. For example, to find the energy of a ‘store-relative’ instruction that was fetched from the scratchpad memory and stored data back to the scratchpad, 0.55nJ is taken from Table 5.3 as the base instruction cost. The instruction fetch correction factor is read from Table 5.4 in the scratchpad column along with the store relative correction factor. The three energies are summed to give 0.55 - 0.09 - 0.11 = 0.35nJ for the ‘store-relative’. This number is 36% less than 0.55nJ for ‘store-relative’ to use main memory, and clearly illustrates the benefits of having low power memory structures for commonly accessed code. A major benefit of using memory energy correction factors is that only a few of the test cases from Table 5.3 need to be modified and re-run to generate the data in Table 5.4.

To validate the accuracy of the energy model proposed in this paper, the C-language instruction level simulator of the WIMS processor was modified to include the calculated instruction energies and memory energy correction factors presented here. Six test cases were written that use all of the instruction groups listed in Table 5.3, along with an assortment of different memory banks. The C-simulator was run on each of these test cases and the total energy was predicted. The same test cases were looped on the WIMS processor hardware and the energy was calculated from the measured current. The error between the predicted and measured energies was less than 4% for all six test cases.

5.1.5 Digital Signal Processor

When running in DSP mode, the MCU core and memory are clocked at the lowest possible frequency to support DSP operation while the DSP is clocked at 3MHz to provide the required data throughput to stimulate the cochlear probes. At 1.2V in DSP mode, the

DSP dissipates 1.14mW of the total 1.79mW system power. 1.79mW is the lowest reported active power consumption for a CI-specific DSP. This clearly demonstrates the benefits of designing dedicated hardware accelerators, such as the CIS DSP, for power constrained designs that run highly parallel algorithms.

Fig. 5.5 shows VDD and frequency scaling possibilities for the DSP core using the on-chip DFS circuit. Power consumption can get below 600μW for the DSP at 1.2V and 1.5MHz if the full processing and stimulation of 3MHz is not required. This on-chip, software selectable performance is something that is not typically available in other application specific signal processing chips.

5.1.6 Dynamic Frequency Scaling

The WIMS DFS circuit from Section 3.4.1 was described entirely in behavioral Verilog HDL, synthesized with Synopsys Design Compiler, and placed-and-routed using Cadence Silicon Ensemble with Artisan standard cells. The DFS unit is one of many macros contained within the peripheral block labelled I/O in Fig. 5.1. No custom design, custom layout, or transistor-level simulation was required to ensure a glitch-free clock when multiplexing between frequencies. This distinguishes the WIMS design from other DFS implementations proposed to date [55], [64], [79] - [80], and makes it ideal for ASIC design cycles constrained by time and manpower. Although it does not provide the range

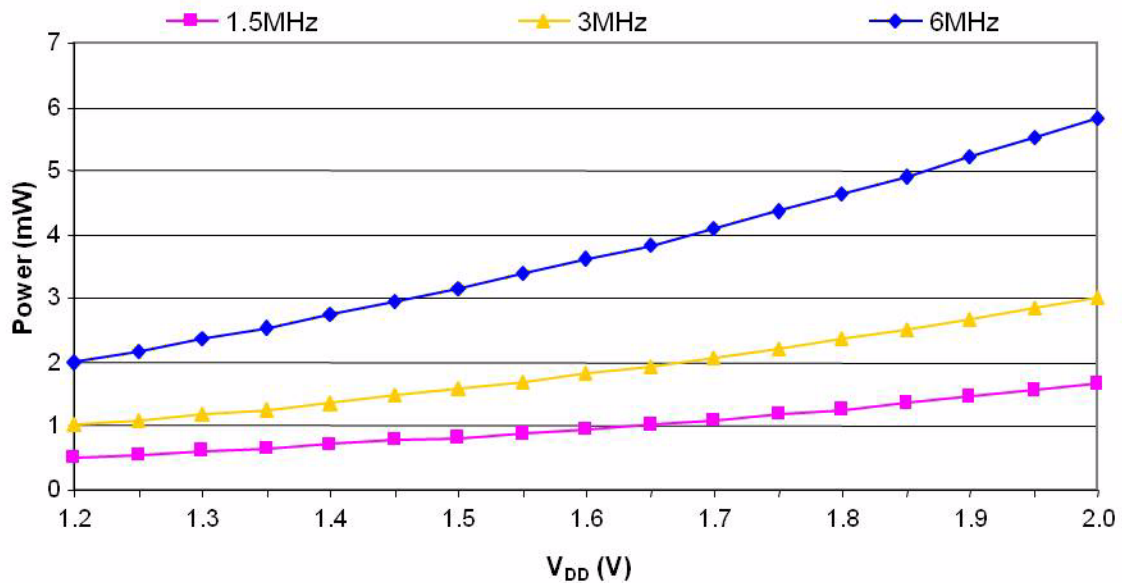


Figure 5.5: Power versus VDD scaling for the DSP.

of frequency and phase options as do spread spectrum clock synthesizers such as the 'flying adder' [80], the WIMS design is much simpler, lower-power, fully HDL-synthesizable, provides ample frequency selections for many DFS applications, and is not restricted to ring oscillator based VCO/PLL clock architectures as is [80].

Fig. 5.6 shows oscilloscope traces of the DFS unit switching frequencies while the MCU core is actively running. From the instant that software changes the clock multiplexer select line, there is a latency of only $n/2f_0$ plus a mux delay until the new frequency has propagated from the multiplexer through the synchronization chain (n=2 flip-flops) and to the clock tree. When the LCO is operating, f_0=100MHz, the latency is about 11ns including a 1ns multiplexer delay. When using the low-power ring oscillator, f_0=10MHz, the total latency increases to about 101ns. The power dissipated by our DFS unit is a relatively con-

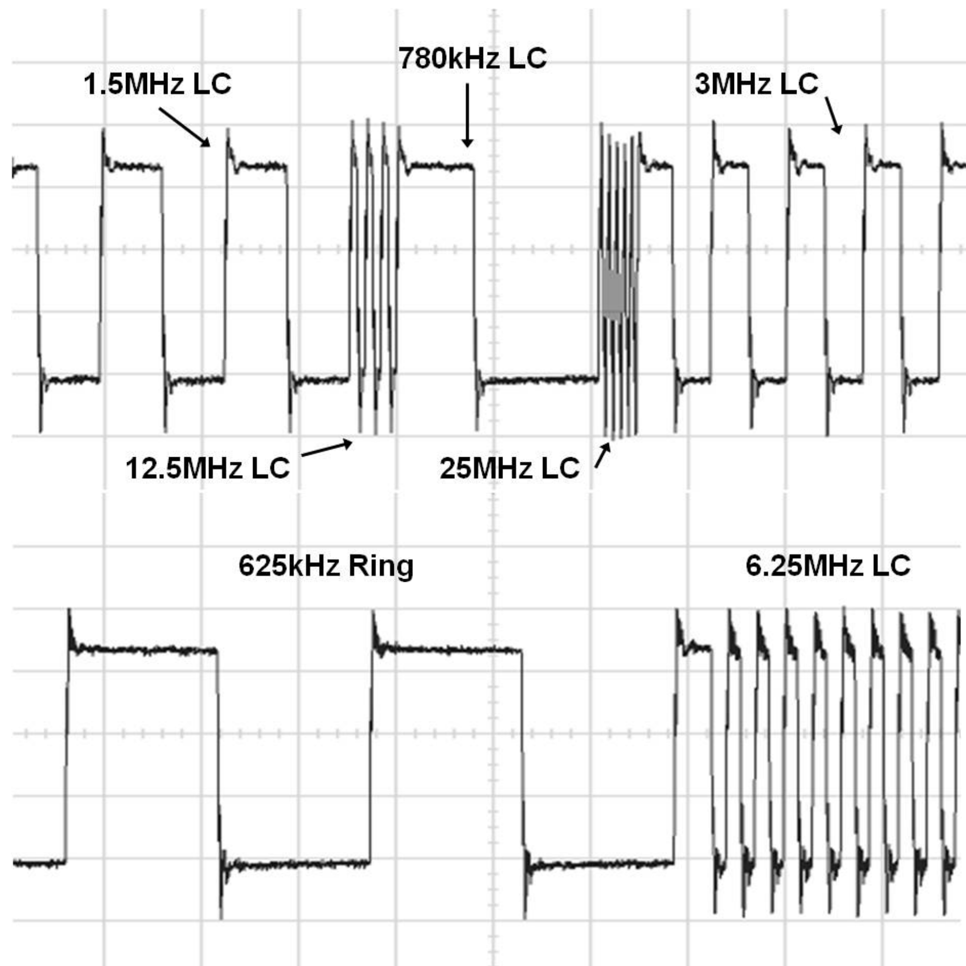


Figure 5.6: Oscilloscope traces showing low-latency dynamic frequency scaling of (top) the TC-LCO clock and (bottom) the TC-LCO and ring clock.

stant 480μW at 1.8V and is much lower than the 150mW reported by [80]. The LCO dissipates 9.62mW and the ring oscillator dissipates 0.82mW at 1.8V. The entire clock synthesizer occupies 0.25mm^2 of silicon.

5.1.7 Performance Comparison

This section compares the WIMS MCU to other commercially available microprocessors as well as the WIMS DSP capabilities and performance to other cochlear implant processing components. Table 5.5 compares the Gen-2 WIMS core against commercially available processor cores in similar processing technologies. It is difficult to do an accurate comparison of different processors due to the many factors involved, so this table is intended to provide only a rough comparison. Table 5.6 examines the WIMS core and the ARM7TDMI to provide a more detailed comparison. The ARM7TDMI offers 32-bit instructions and data with a 16-bit Thumb instruction mode for more compact code. Although the WIMS datapath is only 16-bits wide, the ALU does support multi-word (16b+) signed integer arithmetic for the few applications that might require that level of precision. The WIMS core offers a wide variety of peripherals not present on the ARM core and is 20% more power efficient. The WIMS MCU's large selection of peripherals almost doubles the size of the core, which is one reason why the ARM core is smaller. The WIMS DSP is far and away the lowest power consumption of any of the other DSPs listed. Although this analysis may be too simplified to definitively claim that the WIMS core is

Table 5.5: Comparison of commercially available cores with the Gen-2 WIMS MCU.

Processor	Process (μm)	Voltage (V)	Frequency (MHz)	No. Bits	Active Power (mW)	Standby Power (μW)
TI MSP430F21x1	0.18	1.8	16	16	11.5	126
National CP3000	0.25	2.5	24	16	30.0	2250
Infinion C166S	0.18	1.8	80	16	160.0	990
ARM7TDMI	0.18	1.8	115	32	24.2	-
TI C5402 DSP	0.18	1.8	100	24	81.0	36
Tensilica Xtensa[a]	0.18	1.8	200	32	80	-
WIMS Core	0.18	1.8	100	16	16.7	170
WIMS DSP	0.18	1.8	3	16	1.8	330

a. [81]

more power efficient than commercially available cores, certainly the WIMS core is very competitive with the best commercial offerings.

5.2 Conclusions

This chapter has described the performance achieved by the WIMS Microsystem. The main components include the MCU, DSP, DFS, and scratchpad memory. A methodology and results were presented for calculating the EPI for the WIMS MCU to help with design optimization of the core as well as compiler optimization of code performance for power consumption. These metrics directly impact performance and battery life for embedded applications. The WIMS Microsystem compared favorably to commercially available microcontrollers.

While not described in detail, the LCO IP is a significant advantage in microsystem capabilities when combined with the DFS circuit and compared to commercially available microprocessors. It provides, with minimal overhead cost, a wide operating range for application and system specific performance while eliminating the need for any other clock reference on board.

Table 5.6: Detailed comparison of Gen-2 WIMS core with ARM7TDMI.

Parameter	ARM7TDMI	WIMS
Instruction Width	32, 16b	16b
Data Width	32, 16, 8b	16+, 8b
Address Space	32b	24b
Pipeline	3-stage	3-stage
Interrupt Levels	2	32
Peripherals	Coprocessor & ETM interfaces, JTAG, real-time debug unit	USART (2x), SPI (3x), Timer (3x), external memory interface
Process	0.18μm	0.18μm
Voltage	1.8V	1.8V
Area	0.59mm^2	0.80mm^2
Frequency	115MHz	100MHz
μW/MHz	210	167

The DSP has been shown to be sufficient in processing capability and flexibility for the CIS algorithm across patient parameters. The DSP performance will be compared to commercial CIs in more detail in the next chapter.

CHAPTER VI

CONCLUSION AND FUTURE DIRECTIONS

This chapter begins by comparing the results achieved by the WIMS Cochlear Prosthesis to other Cochlear Implant platforms found in both commercial and academic implementations. Next, demonstration vehicles that utilize the WIMS MCU features are presented. Lastly, significant achievements realized by this work are detailed, and recommendations for future directions are made.

6.1 Platform Comparison

Several approaches have been taken to obtain a functioning cochlear implant system. Keeping patient safety at the forefront, researchers have been looking for ways to improve hearing performance, battery life, and implant aesthetics. This section will compare different methods taken to implement the signal processing in both commercial and academic investigations.

Commercial cochlear implants almost always choose a traditional software programmable DSP for the processing requirements. Academic researchers have investigated more application-specific analog and digital signal processors to improve cochlear implant performance. Table 6.1 compares the work presented here to other work discussed in Section 2.5. The WIMS DSP is the lowest reported area for any implementation even when adding the packaging options for WIMS. It also has the lowest power consumption of all competitors except for Toumazou [34] and Sarpeshkar [35], the two analog signal processors. While the analog signal processors do have slight advantages in power consumption, they sorely lack in configurability for patient-specific tuning of CIs. Also, they are not as

Table 6.1: Comparison of Cochlear Implant signal processing platforms.

Processor	Technology (μm)	Area (mm^2)	Supply (V)	Frequency (MHz)	Power (mW)	Sleep Power (μW)
WIMS	0.18	9.18	1.2	3	1.79	330
5402[a]	0.18	100[b]	1.8	100	45	3,600
Champ-LP[c]	0.35	48	0.85	N/A	7.1	N/A
Sharp[d]	N/A	400[b]	3.3	100	48.2	106,590
Toumazou[e]	0.8	21	5	N/A	0.15	1000
Sarpeshkar[f]	1.5	88	2.8	N/A	0.2	400[g]

a. Texas Instruments TMS320C5402 [26]

b. Area includes standard QFP footprint

c. [31]

d. Uses Motorola DSP56309 [28] with other off-the-shelf components [29]

e. [32], [33]

f. [34]

g. Power is estimated for 32-channel version not actually implemented

flexible as WIMS with regards to expanding to more channels and updates or improvements to the signal processing algorithms and process scaling [82]. Finally, WIMS has the added capability of performing other system control and communication functions that would require extra circuitry if choosing either of the analog signal processing components for a CI system. They are also less flexible than the WIMS solution with regard to expanding the number of channels and the ability to update the signal processing algorithm.

6.2 System Demonstrations

The WIMS Microcontroller SoC has been used in demonstrations for the WIMS ERC for the Environmental and Cochlear testbed. Fig. 6.1 shows the board assembled for the Environmental Testbed demonstration. The MCU is on the lower level board and controls the other system components including the micro gas chromatograph, antenna, and converters. The demonstration vehicle can successfully interface with a host computer over the wireless interface. The on-board MCU can process and transmit data to the host computer. Future advances will improve the volume and power consumption of the system.

Fig. 6.2 shows the final assembly of the WIMS Cochlear Prosthesis demonstration vehicle. The multi-chip package on the right contains the MCU, telemetry chip, RF inter-

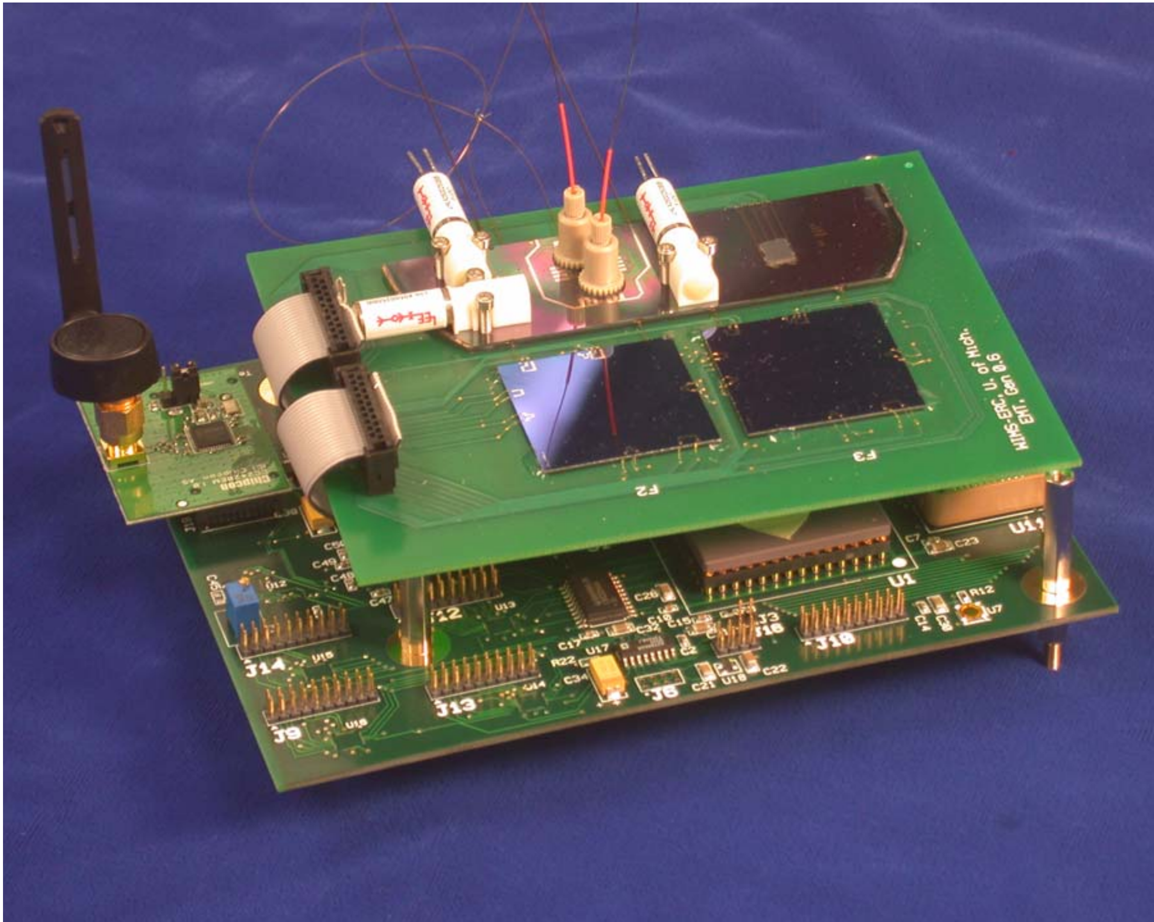


Figure 6.1: WIMS Environmental Testbed demo board.

face, ADC, and voltage regulator with the flexible positioning cable connecting them to the 32-site electrode array on the left side. The electrodes in this system has successfully generated current pulses when directed by the MCU and electrode driving circuitry.

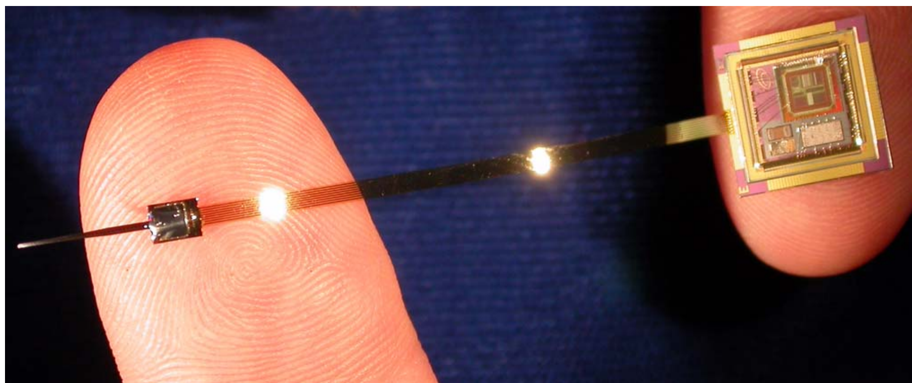


Figure 6.2: Complete assembly of the WIMS Cochlear Prosthesis demonstration.

Students at the University of Utah have continued work on the demonstration of the MCU capabilities. The board in Fig. 6.3 contains the telemetry chip and an ADC under the control of the MCU. LabVIEW software was written in order to load assembly code onto the MCU and execute it while providing sound samples to the ADC and monitoring the telemetry chip output. This made the process of writing C-code and compiling, assembling, loading onto the MCU, and executing that code a push button process. Fig. 6.4 shows logic analyzer traces of the Serial Peripheral Interfaces (SPIs) between the MCU and the ADC and between the MCU and the Cochlear Electrode Array. This successful demonstration of system components communicating with each other is a significant achievement that not all academic research typically achieves. It is a testament to all of the people who worked on each individual component and system-level specifications.

The wide variety of applications that can easily integrate the WIMS MCU shows its flexibility and usefulness. For a fully implantable CI, the integration of several components into a single package is critical due to size constraints, as shown in Fig. 6.2. The WIMS MCU developed in this dissertation provides that integration by implementing the control system, DSP capability, and clock sources necessary for all system functions in a single die, having a small area and extremely low power dissipation.

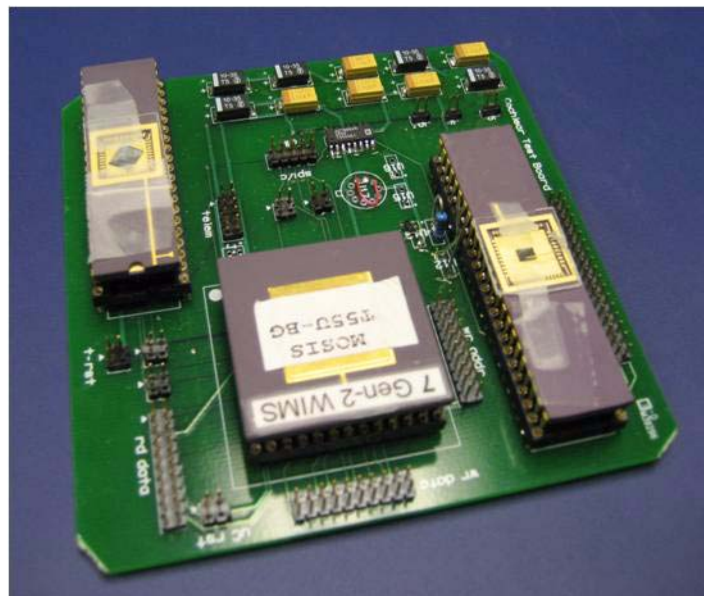


Figure 6.3: University of Utah Cochlear Demonstration Board.

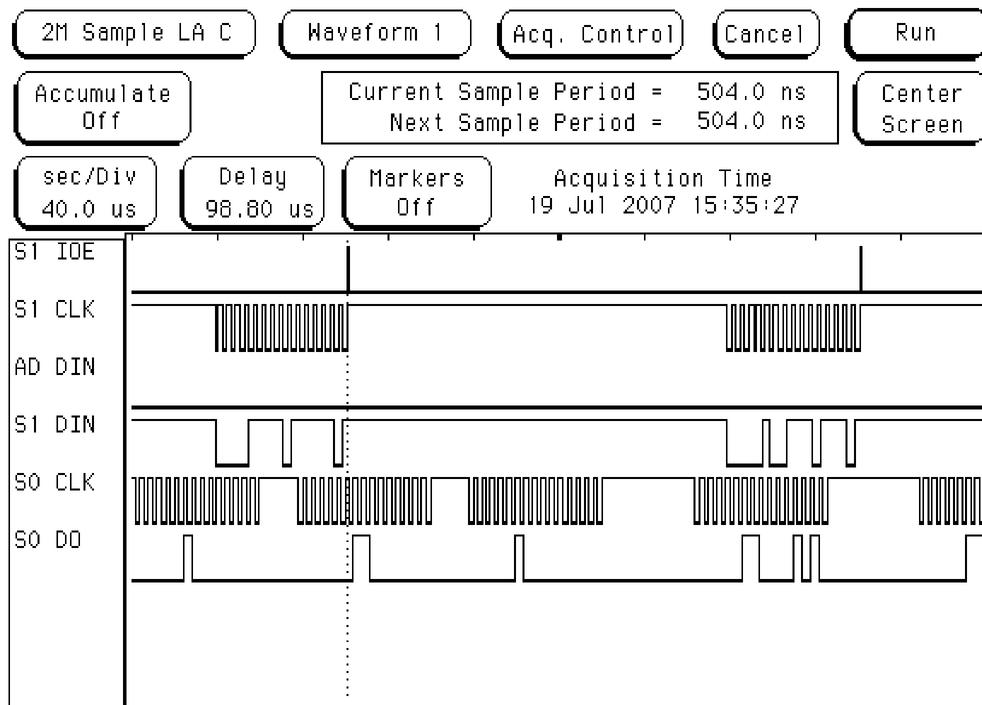


Figure 6.4: Logic Analyzer traces of ADC (SPI1) and Electrode Array (SPI0) communicating with the MCU.

6.3 Achievements

This section describes the noteworthy achievements from this work.

- A microcontroller has been specified, implemented, and tested that satisfies the requirements of WIMS ERC Environmental Monitor and Cochlear Prosthesis testbeds. The aggressive integration of sophisticated components into a small area and with low power consumption is made possible by the storage, processing, and communication capabilities included in the MCU. The 16-bit MCU implements a custom ISA and includes on-chip SRAM, a low-power loop cache or scratchpad memory, and several communication peripherals. The entire WIMS core consumes 16.7mW while operating at 100MHz.

- The 9.18mm^2 WIMS Gen-2 MCU with DSP functionality specifically tuned for the CIS Cochlear Implant processing algorithm achieves 1.79mW total power consumption from 1.2V, including clock generation. This is the lowest reported power for a digital cochlear prosthesis signal processor. Analog signal processors for cochlear implants

have achieved slightly less power consumption, but lack the configurability shown here and are much larger. While in sleep mode, the WIMS MCU power consumption is only 330μW.

- The 16-channel CIS DSP architecture takes advantage of existing MCU components to optimize the complete system. The parallel processing performed by the DSP allows for a low-area implementation by reusing the same processing units (filters) among each of the channels. Dynamic range compression of sound data is done by using the low-power scratchpad memory as a look-up table. The DSP allows for an easy patient fitting procedure by having separate stimulation and programming modes that can all be managed by MCU software. During stimulation mode, processed data is sent to the electrode array through an SPI interface. As an experimental feature, the MCU can set the stimulation profile for each channel or send data directly to the electrodes. The DSP architecture is easily scalable to more channels, higher pulses per second, or other algorithm modifications if required. These features make the WIMS DSP a valuable tool for research as well as a fully-implantable CI device.

- The synthesizable, glitch-free dynamic frequency scaling (DFS) circuit consumes only 480μW with a switching latency of 11ns to 101ns. The DFS circuit is a significant piece of the completely monolithic clocking architecture of the MCU. An on-chip hybrid LCO and ring oscillator provides an accurate 100MHz operating frequency or a low-power 10MHz operating frequency. Both modes are completely software selectable for a total operating range of 100MHz to below 1MHz. Total power consumption is 9.62mW in LCO mode or 0.82mW in low-power mode. All is achieved in 0.25mm^2 of silicon area.

- An on-chip scratchpad memory is included as a power-saving feature. The custom WIMS compiler takes advantage of the scratchpad memory to obtain an average energy savings of 18% across benchmarks. The custom compiler was also utilized to make optimizations in the ISA, specifically in the areas of register windowing, memory access modes, and interrupt support. Application-specific C software for the WIMS testbeds has been written, compiled, and executed on the MCU.

- An *O(n)* energy per instruction (EPI) methodology is presented. It allows detailed analysis and power optimization of software running on the MCU. The methodology allows for both pre- and post-fabrication estimations with minimal overhead compared to similar methodologies. Results achieve less than 4% error and have been annotated into the WIMS MCU simulators for easy software analysis and reporting.
- A thorough analysis of mixed signal design methodologies is presented in Chapter IV, including hardware and software co-design and analysis of existing CAD tool capabilities. A top-down methodology is presented in detail using the WIMS SoC as an example design. Modelling concepts and simulation methodologies are presented for all design domains.

6.4 Future Research Topics

While this work is a significant advancement in the state of the art for MCUs and CI signal processing, there is room for further improvement. Several ideas came about during this work and were not pursued due to a lack of time or resources. They are described here for possible further development.

Integration of an audible-spectrum 16-bit ADC would increase the capabilities of the WIMS SoC as a CI component. The additional area taken up on the WIMS SoC would be minimal compared to the reduction in system area by removing the external ADC from the board. Combining the ADC with a bone-conduction microphone will be the next step toward a fully-implantable CI.

Parkinson's Disease and Epilepsy are topics of great current research interest. Researchers would like to develop systems that can predict the onset of a seizure or control tremors with the use of feedback. These systems, like CIs, require processing of parallel channels of data and system control of sensors or medication delivery vehicles [83] - [85]. While it may be difficult to design a single SoC to perform the control and processing required for all applications, the WIMS platform provides an ideal starting point for adding a custom DSP for each system. Building upon the flexible, low-power features of the WIMS MCU, researchers could efficiently add the capabilities needed to address other embedded medical microsystem applications.

As optimizations to the WIMS MCU, two features could provide significant additions to the processing effectiveness. The first would be standard branch prediction. Currently, taken branches incur a two cycle penalty since they are not resolved until the EX stage of the pipeline, and all branches are assumed to be not-taken. The extra complexity of the branch prediction hardware would need to be analyzed to ensure that it does not add too much area or power to the existing implementation. The power-aware compiler must also take branches into account when optimizing the software. A second feature to be added to the MCU would be the Direct Memory Access (DMA) instructions already outlined in Appendix A. This instruction group would provide a powerful tool for the compiler to optimize data memory location and order.

The final addition to be recommended is to migrate the entire WIMS SoC to a smaller feature size process or Silicon-on-Insulator (SOI). This will have the obvious benefit of reducing the area and the active power of the system. Care should be taken to ensure that leakage power does not come to dominate the sleep mode power of the system. System-level architectural optimizations would most likely be required to further reduce the standby power below the 330μW currently reported. Detailed analysis of the gains to be made is not done here, but standard scaling factor gains [86] can be applied to the current design based on the characteristics of available silicon processes.

6.5 Conclusion

This dissertation has presented an advancement to Cochlear Prostheses moving towards a fully-implantable Cochlear Implant as part of the WIMS NSF ERC. The results of this project could help people with profound deafness to more inconspicuously and conveniently gain the ability to understand speech, allowing them to lead more active lives. The Gen-2 WIMS MCU has the lowest power DSP core that has implemented the CIS algorithm. It has been demonstrated as part of the complete Cochlear Prosthesis testbed, pushing the limits of low-power and low-volume system integration. Self-contained frequency generation and dynamic frequency scaling provide key flexibility to the system. A method of energy estimation was presented to assist in software optimization. A top-down design methodology was presented using the WIMS SOC as a demonstration vehicle. Research advancements such as these can only come from the efforts of large groups of people work-

ing toward a common goal and individual researchers working hard to complete their portion of the system.

Appendix A. WIMS Microcontroller User Manual

Chapter 1. Processor Organization

Introduction

The WIMS processor is an embedded microcontroller for Wireless Integrated Microsystems (WIMS). It is specifically designed to be low-power yet maintain adequate performance in embedded sensor applications. It has a 16-bit integer datapath and 24-bit unified data and instruction address space. This 24-bit address space provides for $2^{24} = 16$ MB of addressable memory. The 16-bit datapath allows efficient manipulation of audio data (typically ~12 bits) or sensor data (12-16 bits) from an analog-to-digital converter. For applications that require greater precision, multi-word arithmetic and shift instructions are provided to enable fixed-decimal-point emulation in software.

Data Registers (DRs)

There are sixteen 16-bit data registers that are referenced by using the shorthand notation r0, r1,..., r15. These are generally used for temporary data storage during instruction computation, however, they can also be used for computing addresses or for loading address registers. Only half of these DRs are accessible by any given instruction. If the Data Register Window (DRW) bit in the MSR is clear, then r0 through r7 are accessible. If the DRW bit in the MSR is set, then r8 through r15 are accessible. The exception to this rule is the Copy Data Register (cdr) instruction, which allows any data register to be copied to any other data register regardless of the DRW bit. This instruction is included to facilitate data transfers between windows.

Address Registers (ARs)

There are fourteen 24-bit address registers that use the notation SP, FP, AR00, AR01,..., AR05, AR10, AR11,..., AR15. These are used for indirect addressing modes as the base register. The 24-bit stack pointer (SP) is for stack operations and compiler support. The 24-bit frame pointer (FP) is used by the compiler to specify the beginning of a stack frame for the compiler. Like the DRs, ARs are also windowed using the Address Register Window (ARW) bit so that either AR00-AR05 or AR10-AR15 are accessible by any given instruction, however, the same SP and FP registers are accessible in both address windows. For example, if the ARW bit is clear then SP, FP, and AR00-AR05 are accessible. If the ARW bit is set, then SP, FP, and AR10-AR15 are accessible. Notice that for the numerical AR*wn* indices, the first digit (*w*) determines the window and the second digit (*n*) determines the register number in that window. There are only twelve ARs, not sixteen. The Copy Address Register (car) instruction allows any address register to be copied to any other address register regardless of the Address Register Window bit.

Link Register (LR)

There is a 24-bit link register to support subroutine calls. During a subroutine call, the next Program Counter (PC) address is automatically loaded into the LR. When returning from a procedure, the LR is loaded back into the PC. The LR is accessible through the non-windowed (NWIN) instructions.

Shift/Divide Auxiliary Register (SDAR)

This register is used to facilitate multi-word shifts, divides, and sign-extension. See instruction examples for how it is used for each type. This register is memory mapped so that it can be saved for interrupt or subroutine support.

Machine State Register (MSR)

There is a 16-bit Machine State Register (MSR) that records most of the important state in the processor. It contains information on Arithmetic Logic Unit (ALU) operations, the window bits, external memory, and the interrupt state of the processor. The MSR is memory mapped so that it is addressable in memory. Besides the PC and interrupt priority

registers, it is one of the few registers to be reset at startup. The MSR resets to 0x0000 so that the INT_LEVEL has the lowest interrupt priority and the remaining MSR fields are cleared. Please refer to Fig. 1 for a diagram of the MSR and see Table 1 for a description of each of the MSR bits.

Figure 1: MSR - Machine State Register

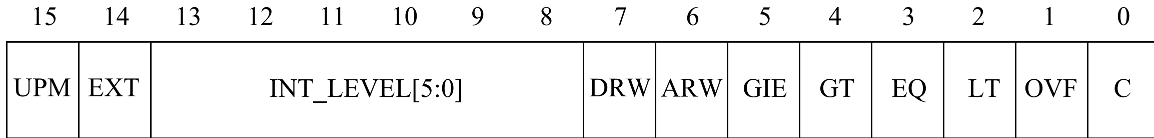


Table 1: Description of MSR Bits

Bit(s)	Mnemonic	Description
0	C	ALU carry output
1	OVF	ALU overflow
2	LT	ALU less-than
3	EQ	ALU equal-to
4	GT	ALU greater-than
5	GIE	Global interrupt enable
6	ARW	Address Register Window
7	DRW	Data Register Window
13:8	INT_LEVEL[5:0]	Current interrupt priority level
14	EXT	External memory enable
15	UPM	Address register update mode

Chapter 2. Addressing Modes

Register

When the Data Register Window bit in the MSR is clear (DRW=0), a 16-bit register can be addressed by r0 to r7. For example,

add r1,r2

loads the sum of r1 and r2 into r1. If the Data Register Window bit is set (DRW=1), then registers r8-r15 can be addressed in the same manner. For example,

add r8,r9

loads the sum of r8 and r9 into r8. If you attempt to address a register that is outside of your current window then the corresponding register in your current window will be used. For example, if you tried the following instruction when DRW=0

add r8,r9

the instruction will be executed as though it were

add r0,r1

because r8 and r9 are outside of the current window and the corresponding registers are r0 and r1. Please note that the register windowing scheme does not apply to the non-windowed instructions (NWIN).

Immediate

An immediate value can be used in either decimal or hexadecimal format. For example,

add r1,-3

loads the sum of r1 and -3 (decimal) into r1. Hexadecimal format is preceded by a “0x”. For example,

add r1,0x2f

loads the sum of r1 and 2f (hexadecimal) into r1. Most instructions that take immediates use signed 8-bit values, however, some use unsigned values or support different sized immediates. Pay careful attention to the description of each instruction when determining the size of the immediate field and whether it is signed or unsigned.

Bit

A single bit in memory can be addressed by an unsigned immediate or unsigned register value. Only the lower 4 bits of the register/immediate are used to select bit[0] through bit[15] of the memory word. The remaining bits of the register/immediate are ignored. For example,

seti 0xe,AR00

would set bit 0xe (0xe = 0b1110 = bit[14]) of the memory addressed by the location stored in AR00.

Register Pair

The 16-bit multiply instruction requires a double precision value (32-bit) for the destination and the divide instruction requires a double precision value for the dividend operand. This is accomplished using a register pair. Only the address of an even register can be used to specify the register pair and the consecutive odd register is implied. For example,

mul r0,r2

multiplies the value in r0 by the value in r2 and stores the two word (32-bit) result in the register pair r0,r1 with r0 holding the most significant word of the result.

Base Register Indirect

Any of the address registers (FP, SP, AR00-AR05, AR10-AR15) can be used as base registers in register indirect addressing. For example,

set r0,(AR00)

sets the bit specified by r0 of the memory addressed by the location stored in AR00. From now on, the shorthand notation memory[AR00] will be used to represent the “memory addressed by the location stored in AR00”.

Base Register Indirect + Immediate Offset

Any of the address registers (FP, SP, AR00-AR05, AR10-AR15) can be used as base registers in register indirect addressing with a signed immediate offset value. For example,

lo r0,-2(AR00)
loads r0 with memory[AR00-2].

Base Register Indirect + Index Register Offset

Any of the address registers (FP, SP, AR00-AR05, AR10-AR15) can be used as base registers in register indirect addressing with an index register offset. The index register is a 16-bit signed number stored in a DR. For example,
ld r0,r1(AR00)
loads r0 with memory[AR00+r1].

Base Register Update

The WIMS Microcontroller supports both post-update and pre-update addressing modes to provide AR manipulation. This minimizes the need for special address manipulating and stack instructions. The update mode is specified by the address register update mode (UPM) bit in the MSR. If UPM is clear (UPM=0) then post-update will be used. If UPM is set (UPM=1) then pre-update will be used. Please note that post-update (UPM=0) is the default mode after the MSR has been reset during microcontroller startup.
During a post-update instruction, the original AR value is used in the current operation and then the AR is updated to its new value. For example,
lou r0,-2(AR00)
will first load r0 with memory[AR00] and then update AR00 to AR00-2.
During a pre-update instruction, the new AR value is calculated and used in both the current operation and to update the AR. For example,
lou r0,-2(AR00)
will load r0 with memory[AR00-2] while updating AR00 to AR00-2.

Direct Memory Access (DMA) - DMAs were not implemented in Gen-2

The WIMS Microcontroller has special DMA instructions to improve the performance of spilling registers, subroutine calls, and setting up the loop cache (LC). The register spilling DMA instructions utilize Base Register (BR) + Index Register Offset or Base Register + Immediate Offset addressing modes to determine the address of the spill. Both offsets are signed values. The BR is selected by the 4-bit value stored in the memory mapped DMA Base Register (DBR). The available settings for DBR are shown in the 'dddd ssss' column of Table 4. For example, if DBR = 0b0110 (which is SP) then:
dmst 0x01, r2
will store memory[SP+r2] with AR*w*1, where *w*=ARW.

For multiple register DMA accesses, an internal 'index' is used with the BR and the offset to cycle through memory. If set, the 1-bit, memory-mapped DMA Increment (DINC) register selects (BR + offset + index), otherwise (BR + offset - index) is used. An example of a multiple register DMA with DBR = 0b1000 (which is AR10) and INC = 1 is:
dmldi 0x09, 0x02
which loads r0<-memory[AR10+2+0], r1<-memory[AR10+2+2], r2<-memory[AR10+2+4], r3<-memory[AR10+2+6], r4<-memory[AR10+2+8], r5<-memory[AR10+2+10], r6<-memory[AR10+2+12], r7<-memory[AR10+2+14]. If DRW=1, then it loads r8-r15.

The loop cache DMA instructions are designed to make filling the loop cache easier and more power efficient. The 512-byte loop cache is divided into thirty-two, 16-byte data 'chunks' that are encoded using a 5-bit value. The memory-mapped DMA Target Chunk ID (DTCID) register points to the starting chunk for a loop cache DMA access (load or store). The memory-mapped DMA Number of Chunks (DNC) specifies how many consecutive chunks to access starting at the DTCID address. Both registers are 5-bits. As shown above, the memory address for the DMA loop cache access is calculated as (BR + offset +/- index) by looking at the DBR and DINC registers.
dmld 0x0f, r0
will load the loop cache at Chunk DTCID with memory[BR + r0 +/- index], and repeat for the number of chunks in DNC.

Chapter 3. Instructions

There are 10 store, 10 load, 3 register initialization, 4 direct memory access, 17 arithmetic, 7 logical, 12 shift/rotate, 5 bit operation, 8 non-windowed, 9 control-of-flow, and 4 miscellaneous instructions. There are 85 instructions total so we will use a 5-bit primary opcode in conjunction with 8 secondary opcodes (CF, BIT, ARITH, SHIFT, STIDX, LDIDX, NWIN, and MISC). Each of these instructions is detailed in the section below with a brief description and an example.

Store/Load Operations

There are four modifiers for the load/store instructions:

1. B specifies whether the instruction accesses a byte (8 bits) or word (16 bits) of memory.
2. A specifies whether the address mode uses a 24-bit absolute immediate.
3. O specifies whether the address mode uses a signed 5-bit immediate offset (-15:15).
4. U specifies whether the address mode is post/pre-update (UPM=0/1 respectively).

Note that all register offsets are treated as signed values.

STO : Store Word Indirect with Immediate Offset
sto r0,-2(AR00)
[AR00-2] <- r0

STOU : Store Word Indirect with Immediate Offset, Update AR
stou r0,-2(AR00)
[AR00] <- r0 (post-update mode)
[AR00-2] <- r0 (pre-update mode)
AR00 <- AR00-2

STOB : Store Byte Indirect with Immediate Offset
stob r0,2(AR00)
[AR00+2] <- r0
Only the lower 8-bits of the register are written to memory.

STOBU : Store Byte Indirect with Immediate Offset, Update AR
stobu r0,2(AR00)
[AR00] <- r0 (post-update mode)
[AR00+2] <- r0 (pre-update mode)
AR00 <- AR00+2
Only the lower 8-bits of the register are written to memory.

STA : Store Word Absolute
sta r0,0x123456
[0x123456] <- r0

STAB : Store Byte Absolute
stab r0,0x123456
[0x123456] <- r0
Only the lower 8-bits of the register are written to memory.

ST : Store Word Indirect with Register Offset
st r0,r1(AR00)
[AR00+r1] <- r0

STU : Store Word Indirect with Register Offset, Update AR
stu r0,r1(AR00)
[AR00] <- r0 (post-update mode)

[AR00+r1] <- r0 (pre-update mode)
AR00 <- AR00+r1

STB : Store Byte Indirect with Register Offset
stb r0,r1(AR00)
[AR00+r1] <- r0
Only the lower 8-bits of the register are written to memory.

STBU : Store Byte Indirect with Register Offset, Update AR
stbu r0,r1(AR00)
[AR00] <- r0 (post-update mode)
[AR00+r1] <- r0 (pre-update mode)
AR00 <- AR00+r1
Only the lower 8-bits of the register are written to memory.

LO : Load Word Indirect with Immediate Offset
lo r0,-2(AR00)
r0 <- [AR00-2]

LOU : Load Word Indirect with Immediate Offset, Update AR
lou r0,-2(AR00)
r0 <- [AR00] (post-update mode)
r0 <- [AR00-2] (pre-update mode)
AR00 <- AR00-2

LOB : Load Byte Indirect with Immediate Offset
lob r0,2(AR00)
r0 <- [AR00+2]
The upper 8-bits of the register are loaded with 0x00.

LOBU : Load Byte Indirect with Immediate Offset, Update AR
lobu r0,2(AR00)
r0 <- [AR00] (post-update mode)
r0 <- [AR00+2] (pre-update mode)
AR00 <- AR00+2
The upper 8-bits of the register are loaded with 0x00.

LA : Load Word Absolute
la r0,0x123456
r0 <- [0x123456]

LAB : Load Byte Absolute
lab r0,0x123456
r0 <- [0x123456]
The upper 8-bits of the register are loaded with 0x00.

LD : Load Word Indirect with Register Offset
ld r0,r1(AR00)
r0 <- [AR00+r1]

LDU : Load Word Indirect with Register Offset, Update AR
ldu r0,r1(AR00)
r0 <- [AR00] (post-update mode)
r0 <- [AR00+r1] (pre-update mode)

AR00 <- AR00+r1

LDB : Load Byte Indirect with Register Offset
ldb r0,r1(AR00)
r0 <- [AR00+r1]
The upper 8-bits of the register are loaded with 0x00.

LDBU : Load Byte Indirect with Register Offset, Update AR
ldbu r0,r1(AR00)
r0 <- [AR00] (post-update mode)
r0 <- [AR00+r1] (pre-update mode)
AR00 <- AR00+r1
The upper 8-bits of the register are loaded with 0x00.

Direct Memory Access Instructions - DMAs were not implemented in Gen-2

These instructions offer an efficient way to fill the loop cache or load/store individual Data or Address Registers or groups of registers from/to the memory stack. The memory location is determined by adding the signed immediate or register offset to the base Address Register. The Data and/or Address Registers to be loaded/stored are determined by the DMT (Direct Memory Access Type) defined in Table 4. BR below is the Base Register determined by the value in the DBR memory mapped register.

When loading/storing a group of DMT registers, the base Address Register and the offset value are left untouched, but the address being stored to is internally incremented according to the number of words being loaded/stored.

DMST : Direct Memory Access Store
dmst DMT,r0
[BR+r0] <- DMT Registers

DMSTI : Direct Memory Access Store Immediate
dmsti DMT,0x04
[BR+0x04] <- DMT Registers
The immediate is sign extended.

DMLD : Direct Memory Access Load
dmld DMT,r0
DMT Registers <- [BR+r0]

DMLDI : Direct Memory Access Load Immediate
dmldi DMT,0x04
DMT Registers <- [BR+0x04]
The immediate is sign extended.

Register Initialization Operations

These instructions load 8-bit immediate values into registers but do not access memory. ldlbi is useful for clearing data registers because the immediate value is zero extended.

LDLBI : Load Low Byte Immediate
ldlbi r0,0x04
r0 <- 0x0004
The upper 8-bits of the immediate are zero extended.

LDHBI : Load High Byte Immediate
ldhbi r0,0xC6
r0 <- {0xC6, r0[7:0]}

The lower 8-bits of the register remain unchanged.

LDABI : Load Address Register Upper Byte Immediate
ldabi AR00,0x0C
AR00<- {0x0C, AR00[15:0]}
The lower word of the Address Register remains unchanged.

Arithmetic Operations

For all arithmetic operations (except compare instructions), the resulting data is compared to zero and the corresponding bit of the MSR is set: LT, EQ, GT. All 8-bit immediate values are sign extended to 16-bits, except for muli, mulsi, divi, and divsi which already take 16-bit signed immediates.

ADD : Add
add r0,r1
r0 <- r1+r0
C <- carry
OVF <- overflow

ADDC : Add with Carry
addc r0,r1
r0 <- r1+r0+C (Carry bit from MSR)
C <- carry
OVF <- overflow
if (result == 0)
    Does not modify flags
else
    Set LT, GT accordingly

ADDI : Add Immediate (Negate immediate to perform SUBI)
addi r0,0x04
r0 <- r0+0x0004
C <- carry
OVF <- overflow

SUB : Subtract
sub r0,r1
r0 <- r0-r1
C <- borrow
OVF <- overflow

SUBC : Subtract with Carry
subc r0,r1
r0 <- r0-r1-C (Carry bit from MSR)
C <- borrow
OVF <- overflow
if (result == 0)
    Does not modify flags
else
    Set LT, GT accordingly

MUL : Multiply
mul r0,r2
(r0,r1) <- r0*r2
C <- 0

OVF <- 0 (overflow can never occur)

MULS : Multiply Single Word
muls r0,r2
r0 <- r0*r2
C <- 0
OVF <- overflow

MULI : Multiply Immediate
muli r0,0x0004
(r0,r1) <- r0*0x0004
C <- 0
OVF <- 0 (overflow can never occur)

MULSI : Multiply Single Word Immediate
mulsi r0,0x0004
r0 <- r0*0x0004
C <- 0
OVF <- overflow

DIV : Divide
div r0,r2
r0 <- (r0,r1) / r2
C <- 0
OVF <- overflow
SDAR <- remainder

DIVS : Divide Single Word
divs r0,r2
r0 <- r0 / r2
C <- 0
OVF <- overflow (only for 8000/ffff = 8000)
SDAR <- remainder

DIVI : Divide Immediate
divi r0,0x0004
r0 <- (r0,r1) / 0x0004
C <- 0
OVF <- overflow
SDAR <- remainder

DIVSI : Divide Single Word Immediate
divsi r0,0x0004
r0 <- r0 / 0x0004
C <- 0
OVF <- overflow (only for 8000/ffff = 8000)
SDAR <- remainder

SEXTB : Sign Extend Byte
sextb r0, r1
r0 <- {8{r1[7]}, r1[7:0]}
C <- 0
OVF <- 0
SDAR[15:0] <- 16{r1[7]}

The above {} notation is used by Verilog to denote copying r1's bit[7] (the sign bit) 8 times into the upper 8 bits. This SEXTB instruction is useful when loading a signed byte from memory and turning it into a signed 16-bit word. To perform 32-bit sign extension, do a SEXTB on the original byte, followed by a load absolute on the SDAR. Load the SDAR into the register(s) to contain the sign-extended upper word(s).

CMP : Compare

cmp r0,r1

```
if (r0 < r1)
    LT <- 1, EQ <- 0, GT <- 0
else if (r0 == r1)
    LT <- 0, EQ <- 1, GT <- 0
else
    LT <- 0, EQ <- 0, GT <- 1
```

CMPI : Compare Immediate

cmpi r0,0x04

```
if (r0 < 0x0004)
    LT <- 1, EQ <- 0, GT <- 0
else if (r0 == 0x0004)
    LT <- 0, EQ <- 1, GT <- 0
else
    LT <- 0, EQ <- 0, GT <- 1
```

The immediate is sign extended.

CMPM : Compare Multiple Unsigned

cmpm r1, r3

```
if (EQ == 1)
    if (r1 < r3)
        LT <- 1, EQ <- 0, GT <- 0
    else if (r1 == r3)
        LT <- 0, EQ <- 1, GT <- 0
    else
        LT <- 0, EQ <- 0, GT <- 1
else
    Does not modify LT, EQ, GT flags.
```

The compare multiple instruction has a slightly different semantic than cmp. It acts as a compare *unsigned* instruction, but only writes flags if the EQ bit was set by a prior instruction. Its purpose is to allow efficient multi-word signed comparisons without the need for intermediate branch instructions. A signed multi-word comparison is initiated with a signed comparison (cmp) on the most significant word, followed by unsigned cmpm instructions for the remaining words of the number. The flags will be set correctly when done. The following instruction sequence shows how two 32-bit signed numbers stored in (r0,r1) and (r2,r3) can be compared:

cmp r0,r2
cmpm r1,r3
br gt, label

Logical Operations

For all logical instructions, the resulting data is compared to zero and the EQ bit is set if all bits of the result are zero. The GT and LT bits are cleared. The C and OVF flags are unaffected. All 8-bit immediate values are zero extended.

AND : Bitwise AND
and r0,r1
r0 <- r0 & r1

ANDI : Bitwise AND Immediate
andi r0,0x04
r0 <- r0 & 0x0004

OR : Bitwise OR
or r0,r1
r0 <- r0 | r1

ORI : Bitwise OR Immediate
ori r0,0x04
r0 <- r0 | 0x0004

XOR : Bitwise Exclusive OR
xor r0,r1
r0 <- r0 ^ r1

XORI : Bitwise Exclusive OR Immediate
xori r0,0x04
r0 <- r0 ^ 0x0004

NOT : Bitwise Not
not r0, r1
r0 <- ~r1

Shift/Rotate Operations

The shift/rotate instructions use only bits[3:0] of immediate or register values to determine the shift amount, which ranges from 0-15 (0x0 - 0xF). All shift instructions set the LT, EQ, and GT flags with the exception of shift with carries, which set only certain flags, and rotates, which don't set any flags. The OVF flag is modified only by shift left instructions when the most significant bit (MSB) flips while in the process of being shifted. Refer to the individual shift instructions to see which flags are set by each instruction. The flag setting behavior is to enable shifts to be used instead mul/div when the multiplying/dividing by factors of 2. The final bit shifted out by left and right shifts is put in the Carry bit (C) of the MSR. All bits shifted out for all shift, but not rotate, instructions are saved in the SDAR (SHIFT/DIV Auxiliary Register) to facilitate multi-word shift operations. The SDAR value written is the logarithmically encoded shift-out value from the logarithmic shifter and is not easily verified by visual inspection. The shift with carry operations use the SDAR as the bits shifted in, so using a shift with carry is the easiest way to decode the SDAR value. Only bits[0-14] of the SDAR are used and bit[15] is set to 0. Multi-word shift operations can be performed with the shift with carry operations. This can be useful for emulating fixed-point multiply and divide. Logical instructions shift in 0's while arithmetic shifts preserve the sign bit when shifting. A "rotate left" instruction can be emulated by subtracting the desired "rotate left" value from 16 and using this amount with rr or rri.

SLL : Shift Left Logical
sll r0,r1
r0 <- r0 << r1
C <- shifted out bit (or 0 in case of shift by 0)
OVF <- set if MSB flips while shifting
SDAR <- {0, encoded shifted out bits}
Shifts in 0's.

SLLI : Shift Left Logical by Immediate
slli r0,0xC

r0 <- r0 << 0xC
C <- shifted out bit (or 0 in case of shift by 0)
OVF <- set if MSB flips while shifting
SDAR <- {0, encoded shifted out bits}
Shifts in 0's.

SLC : Shift Left with Carry
slc r0,r1
r0 <- r0 << r1

if (r0 < 0)
LT <- 1, EQ <- 0, GT <- 0
else if (r0 > 0)
LT <- 0, EQ <- 0, GT <- 1
else
Does not modify LT, EQ, GT flags.

C <- shifted out bit (or 0 in case of shift by 0)
OVF <- set if MSB flips while shifting
SDAR <- {0, encoded shifted out bits}
Shifts in SDAR.

SLIC : Shift Left by Immediate with Carry
slic r0,0xC
r0 <- r0 << 0xC

if (r0 < 0)
LT <- 1, EQ <- 0, GT <- 0
else if (r0 > 0)
LT <- 0, EQ <- 0, GT <- 1
else
Does not modify LT, EQ, GT flags.

C <- shifted out bit (or 0 in case of shift by 0)
OVF <- set if MSB flips while shifting
SDAR <- {0, encoded shifted out bits}
Shifts in SDAR.

SRA : Shift Right Arithmetic
sra r0,r1
r0 <- r0 >> r1
C <- shifted out bit (or 0 in case of shift by 0)
SDAR <- {0, encoded shifted out bits}
Sign bit is preserved. EQ, GT, and LT are set accordingly.

SRAI : Shift Right Arithmetic by Immediate
srai r0,0xC
r0 <- r0 >> 0xC
C <- shifted out bit (or 0 in case of shift by 0)
SDAR <- {0, encoded shifted out bits}
Sign bit is preserved. EQ, GT, and LT are set accordingly.

SRL : Shift Right Logical
srl r0,r1

r0 <- r0 >> r1
C <- shifted out bit (or 0 in case of shift by 0)
SDAR <- {0, encoded shifted out bits}
Shifts in 0's.

SRLI : Shift Right Logical by Immediate
srli r0,0xC
r0 <- r0 >> 0xC
C <- shifted out bit (or 0 in case of shift by 0)
SDAR <- {0, encoded shifted out bits}
Shifts in 0's.

SRC : Shift Right with Carry
src r0,r1
r0 <- r0 >> r1

if (EQ == 1)
if (r0 > 0)
LT <- 0, EQ <- 0, GT <- 1
else
Does not modify LT, EQ, GT flags.
else
Does not modify LT, EQ, GT flags.

C <- shifted out bit (or 0 in case of shift by 0)
SDAR <- {0, encoded shifted out bits}
Shifts in SDAR.

SRIC : Shift Right by Immediate with Carry
sric r0,0xC
r0 <- r0 >> 0xC

if (EQ == 1)
if (r0 > 0)
LT <- 0, EQ <- 0, GT <- 1
else
Does not modify LT, EQ, GT flags.
else
Does not modify LT, EQ, GT flags.

C <- shifted out bit (or 0 in case of shift by 0)
SDAR <- {0, encoded shifted out bits}
Shifts in SDAR.

RR : Rotate Right
rr r0,r1
This rotates r0 by r1 bits and stores it in r0.

RRI : Rotate Right by Immediate
rri r0,0xC
This rotates r0 by 12 bits and stores it in r0.

Memory Bit Operations

The bit operations modify a single bit in memory at the word location specified using the base register indirect addressing mode. The base register address must be word aligned and bit[15] - bit[0] of the target word are referenced using bits[3:0] of either unsigned immediate or unsigned registered values. If any bit operations, except tmsri, are done on the MSR, the memory write trumps the setting/resetting of the EQ bit. This is to prevent stalling for extra write cycles to the MSR.

TAS : Test-and-Set Bit

tas r0,AR00

This sets the EQ flag in the MSR equal to the bit of [AR00] specified by r0 and then sets the bit.

TASI : Test-and-Set Immediate Bit

tasi 0xf,AR00

This sets the EQ flag in the MSR equal to bit[0xf] of [AR00] and then sets the bit.

TAR : Test-and-Reset Bit

tar r0,AR00

This sets the EQ flag in the MSR equal to the bit of [AR00] specified by r0 and then resets the bit.

TARI : Test-and-Reset Immediate Bit

tari 0xf,AR00

This sets the EQ flag in the MSR equal to bit[0xf] of [AR00] and then resets the bit.

TMSRI : Toggle Machine State Register (MSR) Immediate Bit

tmsri 0x5

This toggles (inverts) bit[0x5] of the MSR (which happens to be the GIE bit). This instruction can be used to efficiently toggle any of the sixteen MSR bits and was designed specifically for easy toggling of the GIE, DRW, ARW, EXT, and UPM bits. If the immediate equals 0x10 (bit[17], which is non-existent) both DRW and ARW bits will be toggled and the lower four bits are ignored. For convenience, this instruction may also be invoked with many of the MSR bit mnemonics (UPM,EXT,DRW,ARW,GIE,GT,EQ,LT,OVF,C) as these mnemonics are #defined in the boot_rom.asm file. For example:

tmsri GIE

Non-Windowed Operations

Non-windowed operations have access to registers in both windows, regardless of the Window bit setting in the MSR.

CAR : Copy Address Register

car SP,AR10

This copies address register AR10 to SP. The Address Register Window bit in the MSR is not used by this operation so that all ARs may be copied to or from.

CDR : Copy Data Register

cdr r0,r8

This copies data register r8 to r0. The Data Register Window bit in the MSR is not used by this operation so that all DRs may be copied to or from.

CARW : Copy Address Register and Toggle Address Register Window bit

carw SP,AR01

This copies address register AR01 to SP and toggles the Address Register Window (ARW) bit. The Address Register Window bit in the MSR is not used by this operation so that all ARs may be copied to or from.

CDRW : Copy Data Register and Toggle Data Register Window bit

cdrw r0,r8

This copies data register r8 to r0 and toggles the Data Register Window (DRW) bit. The Data Register Window bit in the MSR is not used by this operation so that all DRs may be copied to or from.

CADR : Copy Address to Data Register
cadr r0,AR00
This copies the lower 16-bits of address register AR00 to data register r0. The Address Register Window and Data Register Window bits in the MSR are not used by this operation so that all ARs and DRs may be copied to or from.

CDAR : Copy Data to Address Register
cdar AR00,r8
This copies data register r8 to the lower 16-bits of address register AR00. The upper 8 bits of the AR are left unchanged. The Data Register Window and Address Register Window bits in the MSR are not used by this operation so that all ARs and DRs may be copied to or from.

CADRB : Copy Address to Data Register Byte
cadrb r0,AR00
This copies the upper 8-bits of address register AR00 to the lower 8-bits of data register r0. The upper 8 bits of the DR are cleared. The Address Register Window and Data Register Window bits in the MSR are not used by this operation so that all ARs and DRs may be copied to or from.

CDARB : Copy Data to Address Register Byte
cdarb AR00,r8
This copies the lower 8-bits of data register r8 to the upper 8-bits of address register AR00. The lower 16-bits of the AR are left unchanged. The Data Register Window and Address Register Window bits in the MSR are not used by this operation so that all ARs and DRs may be copied to or from.

Control of Flow Operations

All of the control of flow instructions (with the exception of swi, reti, and ret) support labels as the destination address.

JR : Jump Relative
jr -8
jr label
The jr instruction can jump forward or backward up to 127 *words*.

JMP : Jump Absolute
jmp 0x123456
jmp label
The jmp instruction can jump anywhere in memory (however odd addresses will generate an alignment exception).

BR : Branch Relative
br ne,label
label: br lte,-127
There are six supported branch conditions: *less-than* (**lt**), *equal-to* (**eq**), *not-equal-to* (**ne**), *greater-than* (**gt**), *less-than-or-equal-to* (**lte**), and *greater-than-or-equal-to* (**gte**). The branch condition is evaluated by checking the LT, EQ, and GT flags in the MSR. The br instruction can conditionally jump forward or backward up to 127 *words* and supports labels for convenience.

Interrupts are supported in the instruction set with the swi and reti instructions:
SWI : Software Interrupt
swi
The swi instruction causes a software interrupt and invokes the corresponding interrupt handler.

RETI : Return from Interrupt

reti

The reti instruction loads the IPC into the PC and the IMSR into the MSR.

Subroutine calls are supported in the instruction set with the following four instructions:

JSR : Jump Subroutine

jsr 0x123456

jsr label

The jsr instruction can jump to a subroutine anywhere in memory (however odd addresses will generate an alignment exception). The next PC is loaded into the LR.

JSRW : Jump Subroutine and Toggle Window Bits

jsrw 0x123456

jsrw label

The jsrw instruction can jump to a subroutine anywhere in memory (however odd addresses will generate an alignment exception). The next PC is loaded into the LR. Both Address and Data Window bits (DRW, ARW) in the MSR are inverted.

JRSR : Jump Relative Subroutine

jrsr -8

jrsr label

The jrsr instruction can jump forward or backward up to 127 words for a subroutine call. The next PC is loaded into the LR.

RET : Return from Subroutine

ret

The ret instruction loads the LR into the PC.

Miscellaneous/Test Instructions

Test instructions such as sstep, stop and start are normally received over the test interface. Stop is also useful for software breakpoints in normal program code.

STOP : Stop Execution

stop

This instruction halts the processor and freezes execution from the PC until a start instruction or an interrupt is received. It allows execution from the test interface, however.

START : Start Execution

start

This instruction exits stop mode.

SSTEP : Single Step

sstep

When in stop mode, this executes a single instruction and then remains in stop mode. When not in stop mode, this instruction has no effect.

NOOP : No Operation

noop

This instruction performs no operation.

Chapter 4. Opcode Formats

Table 2 shows the various opcode formats that used by the WIMS Assembly instructions. Each different format is assigned a unique format number (left). The physical breakdown of each bit in the instruction is shown in the Word 1 and Word 2 columns with the bit field descriptions given in Table 3. Table 4 provides some additional detail on the exact bit encodings of some of the bit fields shown in Tables 2 and 3. The branch conditions (ccc) are shown as well as various windowed and non-windowed register encodings and DMA options. It should be noted that the 'dddd' and 'ssss' fields can be used to encode either address registers or general purpose registers depending on the instruction.

Table 2: WIMS Instruction Formats

Instruction Format	Number of Opcodes		Word 1	Word 2	Usage
	Primary	Secondary			
1	1	3	ooooo qqq uuuu uuuu		Control flow (swi, ret, reti)
2		2	ooooo qqq jjjj jjjj		Jump relative (jr, jrsr)
3		3	ooooo qqq iiii iiii	iiii iiii iiii iiii	Jump absolute (jmp, jsr, jsrw)
4	1	2	ooooo qqq xxx uu sss		Reg/bit operations (tas, tar)
5		2	ooooo qqq xxx u iiii		Imm/bit operations (tasi, tari)
6		1	ooooo qqq uuu i iiii		Toggle bit operations (tmsri)
7	1	4	ooooo qqq uuuu uuuu		Misc/test instructions (stop, start, sstep, noop)
8		2	ooooo qqq sss u hhhh		DMA reg (dmst, dmld)
9	2	0	ooooo jjj jjjj hhhh		DMA imm (dmsti, dmldi)
10	1	8	ooooo qqq dddd ssss		Non-windowed operations (car, cdr, etc.)
11	1	7	ooooo ddd qqqq u sss		Reg/reg shift operations (sll, srl, sra, rri, etc.)
12		7	ooooo ddd qqqq iiii		Reg/imm shift operations (slli, srli, srai, rri, etc.)
13	1	15	ooooo ddd qqqqq sss		Reg/reg arith operations (add, sub, etc.)
14		4	ooooo ddd qqqqq uuu	iiii iiii iiii iiii	Reg/imm arith operations (muli, divi, etc.)
15	1	4	ooooo sss xxx qq ttt		Store indirect w/index reg offset (st, stu, etc.)
16	1	4	ooooo ddd xxx qq ttt		Load indirect w/index reg offset (ld, ldu, etc.)
17	2	0	ooooo ddd jjjj jjjj		Reg/imm arith operations (addi, cmpi)
18	4	0	ooooo ddd kkkk kkkk		Reg/imm operations (ldlbi, andi, etc.)
19	2	0	ooooo ddd iiii iiii		Reg/imm operations (ldhbi, ldabi)
20	1	0	ooooo ccc jjjj jjjj		Branch relative (br)
21	4	0	ooooo sss xxx jjjjj		Store indirect w/imm offset (sto, stou, etc.)
22	4	0	ooooo ddd xxx jjjjj		Load indirect w/imm offset (lo, lou, etc.)
23	2	0	ooooo sss iiii iiii	iiii iiii iiii iiii	Store absolute (sta, stab)
24	2	0	ooooo ddd iiii iiii	iiii iiii iiii iiii	Load absolute (la, lab)

Table 3: Opcode Field Interpretations

Opcode Field	Windowed Field	Description	Opcode Field	Windowed Field	Description
ooooo	No	Primary opcode	ttt	Yes	Offset register
q...q	No	Secondary opcode	ccc	No	Branch condition code
dddd	No	Destination register	i...i	No	Immediate, no extension[1]
ddd	Yes	Destination register	j...j	No	Immediate, sign extension
ssss	No	Source register	k...k	No	Immediate, zero extension
sss	Yes	Source register	hhhh	Yes	Direct Memory Type[2]
xxx	Yes	Index register	u...u	No	Unused bits

1. See tmsri description in Chapter 3 for details of fifth bit.
2. See Table 4 for details.

Table 4: Important Opcode Field Mappings

ccc	Branch Condition	ARW	xxx	Address Register	dddd ssss	Address Register	Data Register	DRW	ddd sss	Data Register	hhhh	Direct Memory Type (DMT)
000	eq	0	000	AR00	0000	AR00	r0	0	000	r0	0000	AR*w*0[1]
001	ne	0	001	AR01	0001	AR01	r1	0	001	r1	0001	AR*w*1[1]
010	lt	0	010	AR02	0010	AR02	r2	0	010	r2	0010	AR*w*2[1]
011	gt	0	011	AR03	0011	AR03	r3	0	011	r3	0011	AR*w*3[1]
100	Unused	0	100	AR04	0100	AR04	r4	0	100	r4	0100	AR*w*4[1]
101	Unused	0	101	AR05	0101	AR05	r5	0	101	r5	0101	AR*w*5[1]
110	lte	0	110	SP	0110	SP	r6	0	110	r6	0110	SP
111	gte	0	111	FP	0111	FP	r7	0	111	r7	0111	FP
		1	000	AR10	1000	AR10	r8	1	000	r8	1000	AR*w*0-5[1], SP, FP
		1	001	AR11	1001	AR11	r9	1	001	r9	1001	r0-r7[2]
		1	010	AR12	1010	AR12	r10	1	010	r10	1010	LR
		1	011	AR13	1011	AR13	r11	1	011	r11	1011	IMSR, IPC
		1	100	AR14	1100	AR14	r12	1	100	r12	1100	AR*w*0-5[1], r0-r7[2], LR, SP, FP
		1	101	AR15	1101	AR15	r13	1	101	r13	1101	AR*w*0-5[1], r0-r7[2], LR, SP, FP, IMSR, IPC
		1	110	SP	1110	LR	r14	1	110	r14	1110	SDAR
		1	111	FP	1111	IPC	r15	1	111	r15	1111	Loop Cache[3]

1. *w* is the ARW bit in the MSR, either 0 or 1.
2. Saves/restores r0-r7 if DRW = 0, and r8-15 if DRW = 1
3. This is a block copy to or from the Loop Cache. See Chapter 2 for more information

Tables 5 and 6 show all of the WIMS Assembly instructions and their associated primary and secondary opcodes. Any instructions that modify MSR flags or the Shift/Divide Auxiliary Register (SDAR) are denoted on the right, where X means the flag/register is modified (either set to 1 or reset to 0 depending on the condition) and 0 means the flag is cleared (set to 0). If the box is empty then the flag/register is left unmodified. The instruction formats (from Table 4) used by each instruction are also shown.

Table 5: Primary Opcodes

Primary Opcode	Primary Mnemonic	Instruction	MSR Flags Modified									SDAR	Instruction Format(s)
			DRW	ARW	UPM	GIE	GT	EQ	LT	OVF	C		
00000	CF	Control Flow Operations	See Table 6										1, 2, 3
00001	BIT	Bit Operations	See Table 6										4, 5, 6
00010	MISC	Miscellaneous and Test Operations	See Table 6										7, 8
00011	NWIN	Non-Windowed Operations	See Table 6										10
00100	SHIFT	Shift Operations	See Table 6										11, 12
00101	ARITH	Register/Register Operations	See Table 6										13, 14
00110	STIDX	Store Indirect with Index Register Offset	See Table 6										15
00111	LDIDX	Load Indirect with Index Register Offset	See Table 6										16
01000	addi	Add Immediate					X	X	X	X	X		17
01001		Unused											
01010	andi	Bitwise AND Immediate					0	X	0				18
01011	ori	Bitwise OR Immediate					0	X	0				18
01100	xori	Bitwise Exclusive OR Immediate					0	X	0				18
01101	ldlbi	Load Low Byte Immediate											18
01110	ldhbi	Load High Byte Immediate											19
01111	ldabi	Load Address Register Upper Byte Immediate											19
10000	br	Branch Relative											20
10001	cmpi	Compare Immediate					X	X	X				17
10010	sto	Store Word Indirect with Immediate Offset											21
10011	stou	Store Word Indirect with Immediate Offset, Update AR											21
10100	stob	Store Byte Indirect with Immediate Offset											21
10101	stobu	Store Byte Indirect with Immediate Offset, Update AR											21
10110	sta	Store Word Absolute											23
10111	stab	Store Byte Absolute											23
11000	dmsti	Direct Memory Access Store Immediate											9
11001	dmldi	Direct Memory Access Load Immediate											9
11010	lo	Load Word Indirect with Immediate Offset											22
11011	lou	Load Word Indirect with Immediate Offset, Update AR											22
11100	lob	Load Byte Indirect with Immediate Offset											22
11101	lobu	Load Byte Indirect with Immediate Offset, Update AR											22
11110	la	Load Word Absolute											24
11111	lab	Load Byte Absolute											24

Table 6: Secondary Opcodes

Primary Mnemonic	Secondary Opcode	Mnemonic	Instruction	MSR Flags Modified									SDAR	Instruction Format
				DRW	ARW	UPM	GIE	GT	EQ	LT	OVF	C		
CF (00000)	000	swi	Software Interrupt	If taken IMSR<=MSR										1
	001	ret	Return from Subroutine											1
	010	reti	Return from Interrupt	MSR<=IMSR										1
	011	jr	Jump Relative											2
	100	jrsr	Jump Relative Subroutine											2
	101	jsr	Jump Subroutine											3
	110	jmp	Jump Absolute											3
	111	jsrw	Jump Subroutine and Toggle Window Bits	X	X									3
BIT (00001)	000	tas	Test-and-Set Bit						X					4
	001	tar	Test-and-Reset Bit						X					4
	010		Unused											
	011		Unused											
	100	tasi	Test-and-Set Immediate Bit						X					5
	101	tari	Test-and-Reset Immediate Bit						X					5
	110		Unused											
	111	tmsri	Toggle Machine State Register Immediate Bit	X	X	X	X	X	X	X	X	X		6
MISC (00010)	000		Unused											
	001		Unused											
	010	stop	Stop Execution											7
	011	start	Start Execution											7
	100	sstep	Single Step											7
	101	dmst	Direct Memory Access Store											8
	110	dmld	Direct Memory Access Load											8
	111	noop	No Operation											7
NWIN (00011)	000	car	Copy Address Register											10
	001	cdr	Copy Data Register											10
	010	carw	Copy Address Register and Toggle ARW		X									10
	011	cdrw	Copy Data Register and Toggle DRW	X										10
	100	cadr	Copy Address to Data Register											10
	101	cdar	Copy Data to Address Register											10
	110	cadrb	Copy Address to Data Register Byte											10
	111	cdarb	Copy Data to Address Register Byte											10
SHIFT (00100)	0000	sll	Shift Left Logical					X	X	X	X	X	X	11
	0001	sra	Shift Right Arithmetic					X	X	X		X	X	11
	0010	srl	Shift Right Logical					X	X	X		X	X	11
	0011	rr	Rotate Right											11
	0100	slli	Shift Left Logical by Immediate					X	X	X	X	X	X	12
	0101	srai	Shift Right Arithmetic by Immediate					X	X	X		X	X	12
	0110	srli	Shift Right Logical by Immediate					X	X	X		X	X	12
	0111	rri	Rotate Right by Immediate											12
	1000	slc	Shift Left with Carry[1]					X	X	X	X	X	X	11
	1001	src	Shift Right with Carry[1]					X	X	X		X	X	11
	1010		Unused											
	1011		Unused											
	1100	slic	Shift Left by Immediate with Carry[1]					X	X	X	X	X	X	12
	1101	sric	Shift Right by Immediate with Carry[1]					X	X	X		X	X	12
	1110		Unused											
	1111		Unused											

Table 6: Secondary Opcodes

Primary Mnemonic	Secondary Opcode	Mnemonic	Instruction	MSR Flags Modified									SDAR	Instruction Format
				DRW	ARW	UPM	GIE	GT	EQ	LT	OVF	C		
ARITH (00101)	00000	add	Add					X	X	X	X	X		13
	00001	sub	Subtract					X	X	X	X	X		13
	00010	mul	Multiply					X	X	X	0	0		13
	00011	div	Divide					X	X	X	X	0	X	13
	00100	addc	Add w/ Carry[1]					X		X	X	X		13
	00101	subc	Subtract w/ Carry[1]					X		X	X	X		13
	00110	muli	Multiply Immediate					X	X	X	0	0		14
	00111	divi	Divide Immediate					X	X	X	X	0	X	14
	01000		Unused											
	01001		Unused											
	01010	muls	Multiply Single Word					X	X	X	X	0		13
	01011	divs	Divide Single Word					X	X	X	X	0	X	13
	01100		Unused											
	01101		Unused											
	01110	mulsi	Multiply Single Word Immediate					X	X	X	X	0		14
	01111	divsi	Divide Single Word Immediate					X	X	X	X	0	X	14
	10000	sextb	Sign Extend Byte					X	X	X	0	0	X	13
	10001	cmp	Compare					X	X	X				13
	10010		Unused											
	10011		Unused											
	10100		Unused											
	10101	cmpm	Compare Multiple Unsigned[1]					X	X	X				13
	10110		Unused											
	10111		Unused											
	11000		Unused											
	11001		Unused											
	11010	and	Bitwise AND					0	X	0				13
	11011	or	Bitwise OR					0	X	0				13
	11100	xor	Bitwise Exclusive OR					0	X	0				13
	11101	not	Bitwise NOT					0	X	0				13
	11110		Unused											
	11111		Unused											
STIDX (00110)	00	st	Store Word Indirect, Reg. Offset											15
	01	stu	Store Word Indirect, Reg. Offset, Update AR											15
	10	stb	Store Byte Indirect, Reg. Offset											15
	11	stbu	Store Byte Indirect, Reg. Offset, Update AR											15
LDIDX (00111)	00	ld	Load Word Indirect, Reg. Offset											16
	01	ldu	Load Word Indirect, Reg. Offset, Update AR											16
	10	ldb	Load Byte Indirect, Reg. Offset											16
	11	ldbu	Load Byte Indirect, Reg. Offset, Update AR											16

1. See the entry for this instruction in Chapter 3 for exact MSR flag setting behavior.

Chapter 5. Interrupts

The WIMS Microcontroller provides hardware support for *one* level of interrupts/exceptions. The registers used to provide interrupt support are detailed below:

Machine Status Register (MSR)

Already described in Chapter 1, refer to Fig. 1 and Table 1 for details.

Global Interrupt Enable (GIE) - controls whether maskable interrupts are taken or ignored. This bit is automatically reset to 0 when an interrupt is taken and set back to its previous value when an interrupt handler returns via the 'reti' instruction. If desired, GIE can be manually changed in software by toggling the MSR GIE bit with 'tmsri GIE' or by storing to the MSR memory location.

Interrupt Priority Level (INT_LEVEL[5:0]) - The current interrupt priority level is specified in this 6-bit field and is automatically set when the microcontroller detects and handles an interrupt. INT_LEVEL is set back to its previous value when an interrupt handler returns via the 'reti' instruction. Only the lower 5 bits (INT_LEVEL[4:0]) are needed to represent the 32 interrupt priority levels supported in hardware. The upper bit (INT_LEVEL[5]), is not set by the hardware and can be set in software to position the interrupt vector table (IVT) in memory. Table 7 summarizes the 32 possible levels of interrupts/exceptions. Additional details on the peripheral interrupts can be found in the relevant sections of this manual. If desired, INT_LEVEL can be manually changed in software by storing to the MSR memory location.

Interrupt Machine State Register (IMSR)

Automatically stores a copy of the 16-bit MSR immediately before an interrupt is handled. The value is copied back into the MSR when the interrupt returns via a 'reti' instruction. The IMSR is memory-mapped.

Interrupt Program Counter (IPC)

Automatically stores a copy of the 24-bit PC immediately before an interrupt is handled. This value is copied back into the PC and execution continues from this address when the interrupt returns via a 'reti' instruction. The IPC is accessible through the non-windowed (NWIN) copy instructions.

Interrupt Vector Table (IVT) Register

This 16-bit memory mapped register specifies the upper 16-bits of the interrupt vector table location. The lower 8-bits of the location are obtained by concatenating the 6-bit interrupt priority level (INT_LEVEL[5:0]) from the MSR with 0b00. The interrupt vector table must be created by the software (compiler) and the upper 16-bits of the vector table's starting address must be stored into the IVT register. Each entry in the interrupt vector table represents a particular interrupt priority and is allocated 4 bytes so that an absolute jump instruction (size = 4 bytes) can jump to the location of the particular interrupt handler routine. The interrupt vector table can hold up to 32 absolute jump instructions (one for each priority level) at 4 bytes each for a total size of 128 bytes. For this reason, the interrupt vector table must be aligned on a 128 byte boundary that is specified by {IVT[15:0], INT_LEVEL[5]}. Similarly, each jump instruction within the IVT that jumps to an interrupt handler must be aligned on a 4 byte boundary so that the bottom 2 address bits are 0b00. The boot ROM contains its own interrupt vector table and sets the IVT register to point to its vector table during the boot sequence. When the boot ROM is finished, the application software should create its own vector table and update the IVT so that it points to the program's vector table instead of to the boot ROM's vector table.

Interrupt Priority Registers (IPRs) - IPRs are not implemented in Gen-2

Each peripheral unit has a programmable interrupt priority register to set the peripheral's priority level. The priority registers are memory-mapped in Table 11 and can be programmed by software to re-order the priority in which peripheral interrupts are handled. Each external interrupt has a 5-bit IPR shown in Table 7. Some peripherals have

two interrupts but only one IPR. In this case, the IPR is 4-bits and the 5th, least significant bit, is appended in hardware by using the default ordering of the peripheral's interrupts. For example, if the USART's IPR was set to 0b1000 then the RX interrupt priority would be {4'b1000, 1'b1} and the TX priority would be {4'b1000, 1'b0}. The TX could never have a higher priority than RX. At boot up, the hardware resets the IPRs with the default priorities shown in Table 7. In the case that multiple peripherals interrupt at the same time and their IPRs contain the same interrupt priority level, it is up to the interrupt handler to decide how to handle the interrupts. If an IPR is set to a priority level in the non-maskable interrupt range, then maskable interrupts using that IPR will be ignored. This behavior can be used to turn off specific maskable interrupts with more granularity than allowed by the single GIE bit.

Implementation Details

If a maskable interrupt occurs while in the middle of executing a multi-cycle instruction such as mul(s)/div(s)/tas/tar, etc., the interrupt will be handled after the multi-cycle instruction completes. Non-maskable interrupts are typically handled right away as they indicate a critical error condition. When the microcontroller decides to handle an interrupt/exception, the current values of the PC and MSR are first copied into the IPC and IMSR registers and then maskable interrupts are disabled by automatically clearing the GIE bit of the MSR. The microcontroller then prepares to jump to the interrupt vector table and assembles its 24-bit jump destination address as {IVT register[15:0], MSR INT_LEVEL[5:0], 00}. This destination is somewhere in the interrupt vector table, and should contain a 'jmp' instruction to the appropriate interrupt handler. The last instruction in the interrupt handler should usually call 'reti' so that the IPC and IMSR are re-loaded into the PC and MSR and execution resumes from the pre-interrupt PC location.

If the GIE bit is set, then the maskable interrupts (priorities 1-24) are handled, otherwise they are ignored. The exceptions/interrupts with priorities 25-31 are non-maskable and thus are always handled, unless an interrupt with a higher priority is already being handled. In this situation, the machine behavior is unpredictable because there is an unhandled catastrophic exception, however, in most cases the instruction causing the exception will act like a noop and will not alter machine state. If, for some reason, the programmer wants to ignore a particular exception/interrupt, just use a 'reti' in the corresponding interrupt vector table location and the interrupt will return immediately after it is taken. Interrupts are always handled according to their priority with INT_LEVEL = 31 having the highest priority and INT_LEVEL = 0 having the lowest priority. If two interrupts occur at the same time then the higher priority interrupt will be handled first. If an interrupt (maskable or non-maskable) is already in the process of being handled and a non-maskable interrupt of higher priority occurs, then the higher priority interrupt will be immediately handled and this forced interrupt nesting would result in the original IPC and IMSR values being lost. This behavior is acceptable because all of the non-maskable interrupts (with the exception of 'swi', which the programmer controls) represent a catastrophic error that needs to be handled appropriately (e.g. restart the program or reset the microcontroller). In contrast, if a maskable interrupt is already in the process of being handled and a maskable interrupt of higher priority occurs, then the higher priority interrupt will have to wait until the lower priority interrupt finishes and GIE is automatically re-enabled. To avoid this case where high priority maskable interrupts get stuck waiting for low priority maskable interrupts to exit their handling routines, the programmer should keep the maskable interrupt handling routines short so that GIE is not disabled for too long. Also, because maskable interrupts are not latched (with the exception of USART and timer interrupts), it is possible to miss them if they occur while the program is stuck handling another interrupt and they disappear before the program re-enables the GIE bit.

Nested Interrupts

If nested interrupts are desired, it is the responsibility of the software to store the IPC and IMSR before manually re-enabling interrupts/exceptions by setting the GIE bit and adjusting INT_LEVEL.

Table 7: Supported Interrupts/Exceptions

Default Priority	Default INT_LEVEL	Mask-able	Priority Register	Name	Description
0	0b00000	N/A		Default	Normal operational mode (no interrupt)
1	0b00001	Yes		Unused	
2	0b00010	Yes		Unused	

Table 7: Supported Interrupts/Exceptions

Default Priority	Default INT_LEVEL	Mask-able	Priority Register	Name	Description
3	0b00011	Yes		Unused	
4	0b00100	Yes		Unused	
5	0b00101	Yes		Unused	
6	0b00110	Yes	~~IPR10~~	TI transmit (TX) interrupt	The Test Interface transmit FIFO is empty *Interrupt asserted until cleared (See Chapter 10)*
7	0b00111	Yes		TI receive (RX) interrupt	The Test Interface receive FIFO is full *Interrupt asserted until cleared (See Chapter 10)*
8	0b01000	Yes	~~IPR9~~	USART transmit (TX) interrupt	The USART transmit FIFO is empty *Interrupt asserted until cleared (See Chapter 10)*
9	0b01001	Yes		USART receive (RX) interrupt	The USART receive FIFO is full *Interrupt asserted until cleared (See Chapter 10)*
10	0b01010	Yes	~~IPR8~~	SPI2 transmit (TX) interrupt	The SPI2 has completed the last transmission
11	0b01011	Yes		SPI2 NACK interrupt	The SPI2 did not receive acknowledge
12	0b01100	Yes	~~IPR7~~	SPI1 transmit (TX) interrupt	The SPI1 has completed the last transmission
13	0b01101	Yes		SPI1 NACK interrupt	The SPI1 did not receive acknowledge
14	0b01110	Yes	~~IPR6~~	SPI0 transmit (TX) interrupt	The SPI0 has completed the last transmission
15	0b01111	Yes		SPI0 NACK interrupt	The SPI0 did not receive acknowledge
16	0b10000	Yes	~~IPR5~~	Timer2 CU1 interrupt	Timer2 Counting Unit 1 interrupt *Interrupt asserted until cleared (See Chapter 12)*
17	0b10001	Yes		Timer2 CU0 interrupt	Timer2 Counting Unit 0 interrupt *Interrupt asserted until cleared (See Chapter 12)*
18	0b10010	Yes	~~IPR4~~	Timer1 CU1 interrupt	Timer1 Counting Unit 1 interrupt *Interrupt asserted until cleared (See Chapter 12)*
19	0b10011	Yes		Timer1 CU0 interrupt	Timer1 Counting Unit 0 interrupt *Interrupt asserted until cleared (See Chapter 12)*
20	0b10100	Yes	~~IPR3~~	Timer0 CU1 interrupt	Timer0 Counting Unit 1 interrupt *Interrupt asserted until cleared (See Chapter 12)*
21	0b10101	Yes		Timer0 CU0 interrupt	Timer0 Counting Unit 0 interrupt *Interrupt asserted until cleared (See Chapter 12)*
22	0b10110	Yes	~~IPR2~~	External interrupt 2	Active *low* interrupt for use by external devices
23	0b10111	Yes	~~IPR1~~	External interrupt 1	Active *low* interrupt for use by external devices
24	0b11000	Yes	~~IPR0~~	External interrupt 0	Active *low* interrupt for use by external devices
25	0b11001	No		System call/ software interrupt	Software interrupt (swi) instruction
26	0b11010	No		Divide by zero exception	The divisor operand for the div instruction is zero
27	0b11011	No		Invalid memory exception in IF stage	Instruction fetch (IF) stage is attempting to fetch from an invalid (unused) memory location
28	0b11100	No		Invalid memory exception in EX stage	Load/store instruction in the execute (EX) stage is accessing an invalid (unused) memory location
29	0b11101	No		Alignment exception	Mul/div operand not aligned correctly or a word load/store is from/to an odd byte
30	0b11110	No		Invalid instruction exception	Unrecognized instruction
31	0b11111	No		Breakpoint exception	Break on address, data, or interrupt value

Chapter 6. Processor Execution States

The WIMS Microcontroller has three modes of operation: normal, stop, and break. Normal mode is the standard operational mode and is the default mode following system startup/reset. Stop mode can be entered only by using the stop instruction and normal mode can be re-entered after stop by using a start instruction. Break mode can be entered only when the hardware breakpoint condition is met (See Chapter 7).

TODO: Clock Control ... be able to stop clocks to unused peripherals.

TODO: Sleep Mode... be able to shut down all clocks. The chip should be woken up by external interrupts or a watch-dog timer.

Chapter 7. Test Support

Test Interface

The WIMS Microcontroller contains a serial Test Interface (TI) that is tightly integrated with the USART peripheral. The TI can be activated by driving a special test_mode pin high. The purpose of the TI is to simplify chip testing while incurring minimal power and pin count overhead. Because it uses a standard serial protocol, the TI is ideal for remote, in-the-field testing of WIMS applications. When the TI is active, if the processor is in normal mode and fetching instructions from memory, then whenever a word of data is received on the USART, it is injected into the decode stage of the pipeline after the current instruction. If it is a multi-word instruction, then the control logic will stall the pipeline and wait for the second word. After the complete instruction is received, it is allowed to proceed in the pipeline as normal. If the processor is in stop mode then instructions are not fetched from program memory. Instead, only instructions from the test interface are executed in a manner like that described above. When the TI is selected, the USART's RX interrupt is automatically disabled and instruction injection occurs automatically. The TX interrupt is still enabled to facilitate the TI sending data out to the tester.

Using the existing bus architecture, the TI can easily access all DR's, AR's, and the memory mapped registers in Tables 11 and 12 for testing purposes. A variety of important system registers and pipeline registers (PLR's) have also been memory mapped so the TI can read and write their contents. All 24-bit registers have been split in order to accommodate the 16-bit nature of the USART when running in TI mode. The Pipeline Register (PLR) mappings are shown in Table 8. IF, ID, and EX refer to the Instruction Fetch, Instruction Decode, and Execute stages of the three-stage pipeline. PLRs were omitted from the Gen-2 chip.

Table 8: Pipeline Register (PLR) Mappings

Address Range		Mnemonic	Description
Start	End		
~~0xff0080~~			~~Reads 00~~
~~0xff0081~~	~~0xff0083~~	~~PLR0~~	~~IF: PC~~
~~0xff0084~~			~~Reads 00~~
~~0xff0085~~	~~0xff0087~~	~~PLR1~~	~~IF/ID: next PC~~
~~0xff0088~~			~~Reads 00~~
~~0xff0089~~	~~0xff008b~~	~~PLR2~~	~~ID/EX: memory address~~
~~0xff008c~~	~~0xff008d~~	~~PLR3~~	~~IF/ID: instruction~~
~~0xff008e~~	~~0xff008f~~	~~PLR4~~	~~IF/ID: word two~~
~~0xff0090~~	~~0xff0091~~	~~PLR5~~	~~IF/ID: {use TI, two word fsm, bit fsm, 2'b11?, ext int[2:0], 1'b1?, IF inv mem, stop fsm, sstep fsm, brkpt_stall, jump/brach taken fsm, 2'b0}~~
~~0xff0092~~	~~0xff0093~~	~~PLR6~~	~~ID/EX: operand a~~
~~0xff0094~~	~~0xff0095~~	~~PLR7~~	~~ID/EX: operand b~~
~~0xff0096~~	~~0xff0097~~	~~PLR8~~	~~ID/EX: bit operand data~~
~~0xff0098~~	~~0xff0099~~	~~PLR9~~	~~ID/EX: {cmd[4:0], mem rd, mem wr, byte mem, alu/shift update MRF, mem update MRF, 6'b0}~~
~~0xff009a~~	~~0xff009b~~	~~PLR10~~	~~ID/EX: {BR cond[2:0], branch, cmpm, 1'b0?, tas, tar?, tmsri, bit, reti, stall, dest reg[3:0]}~~
~~0xff009c~~	~~0xff009d~~	~~PLR11~~	~~ID/EX: {MSR wen[7:0], int, int lvl[5:0], 1'b0}~~

Breakpoints

In addition to the test interface, there is hardware support for three breakpoint conditions shown in Table 9. These conditions can be: a data match, a memory address, or an interrupt priority level. These breakpoints are to assist the test interface software in stopping the microcontroller accurately. The additional hardware is a 24-bit breakpoint register (BPR), a 2-bit mode select (BPM), and a 24-bit comparator. The four possible breakpoint modes are selected by changing the 2-bit BPM register in the MSR. Interrupt Level will only match when the interrupt level in the MSR is set via a taken interrupt (tmsri and stores to the MSR will not trigger a breakpoint on interrupt level). Unlike other

interrupts, breakpoint exceptions are handled the instruction following the instruction that causes a match because the match is not detected until after the instruction has completed.

Table 9: Breakpoint Mode (BPM) Encoding

BPM[1:0]	Description	BPR Bits Used
0b00	Off (default)	Unused
0b01	Break on data match	BPR[15:0]
0b10	Break on address match	BPR[23:0]
0b11	Break on interrupt priority level	BPR[4:0]

Chapter 8. Boot Up Procedure

After powering on, the microcontroller's SRAM contains junk data that must be initialized if the pipeline is to fetch instructions from the SRAM. This section describes the different ways to load data into the on-chip memory during boot-up. The default method used by the boot ROM is to receive data over the USART and store it into memory. This method is described below in detail. To avoid the rather slow process of loading through the USART, external interrupts can be asserted to enable special features built into the boot ROM. Asserting external interrupt 0 during boot-up will skip the boot ROM's USART memory loading process and jump directly to the first on-chip SRAM address (0x000004). This assumes the program has already been loaded into SRAM and is useful for simulation purposes because simulated loading of the memory over the USART can take a long time. The Verilog simulator has other methods to initialize memory. Asserting external interrupt 1 during boot-up jumps directly to the first address in external memory and starts fetching instructions from that location. This useful feature allows instructions to be fetched from off-chip memory in the event that problems exist with the on-chip memory or with the boot ROM's automatic memory loading processes. Asserting external interrupt 2 during boot-up will load instructions from external memory using the second method described below.

Loading SRAM from the USART

The pseudo-coded algorithm below shows how the boot ROM loads the on-chip SRAM by reading from the USART peripheral running in asynchronous mode. Although slow, this method requires only 1 pad (USART RX) and can be driven by a standard serial port. The notation USART_DATA[7:0] implies the boot ROM is waiting for the USART to receive a byte of data and interrupt the core. The USART receive (RX) interrupt handler in the boot ROM then reads the data from the USART.

```
num_blocks[15:0] = {USART_DATA[7:0] : USART_DATA[7:0]};
for (i = 0; i < num_blocks; i++) {
    chunks_in_block[15:0] = {USART_DATA[7:0] : USART_DATA[7:0]};
    address[23:0] = {USART_DATA[7:0] : USART_DATA[7:0] : USART_DATA[7:0]};
    for (j = 0; j < chunks_in_block; j++) {
        word0 = {USART_DATA[7:0] : USART_DATA[7:0]}; // 1 chunk = 1 word
        mem[address] = word0;
        address = address + 2;
    }
}
jump program_start;
```

The first word of data received by the USART during boot-up is the number of blocks (num_blocks) to be stored into on-chip memory. Each block has its own 16-bit size (block_size) and 24-bit starting address (address) that must be sent prior to sending any data chunks. The block_size is the number of data 'chunks' in the block. After loading these necessary setup values, the data chunks are received and stored into memory. For the USART, data chunks are only 16-bits (1 word) because the having larger chunks does not improve the loading speed. The loading speed is restricted by the baud rate of the USART which defaults to $f_{clk}/4$ in the boot ROM where f_{clk} is the core clock frequency. It should be noted that if the on-chip LC clock source is selected at boot-up, the default core clock frequency should be 50MHz with the possibility of minor variance due to process variation or model mis-match.

Loading SRAM from External Memory

The pseudo-coded algorithm below shows how the boot ROM fills the on-chip SRAM by loading from the 16-bit external memory read data input bus (EXTMEMRDDATA[15:0]). The EXTMEMRDDATA bus can be driven by an external memory chip or by a digital tester. This method is much faster than loading over the USART because it loads 16 bits in parallel at the same clock frequency as the pipeline. To enable loading from the external memory bus, external interrupt 2 must be asserted during the boot-up process.

```
num_blocks[15:0] = EXTMEMRDDATA[15:0];
for (i = 0; i < num_blocks; i++) {
```

```
        chunks_in_block[15:0] = EXTMEMRDDATA[15:0];
        address[23:16] = EXTMEMRDDATA[7:0];
        address[15:0] = EXTMEMRDDATA[15:0];
        for (j = 0; j < chunks_in_block; j++) {
            word0 = EXTMEMRDDATA[15:0]; // 1 chunk = 8 words
            mem[address] = word0;
            word1 = EXTMEMRDDATA[15:0];
            mem[address + 2] = word1;
            word2 = EXTMEMRDDATA[15:0];
            mem[address + 4] = word2;
            word3 = EXTMEMRDDATA[15:0];
            mem[address + 6] = word3;
            word4 = EXTMEMRDDATA[15:0];
            mem[address + 8] = word4;
            word5 = EXTMEMRDDATA[15:0];
            mem[address + 10] = word5;
            word6 = EXTMEMRDDATA[15:0];
            mem[address + 12] = word6;
            word7 = EXTMEMRDDATA[15:0];
            mem[address + 14] = word7;
            address = address + 16;
        }
    }
    jump program_start;
```

Please note that when loading from the external memory bus, the data chunk size is 8 words to reduce loading time.

Non-Default Boot-up Values

The boot ROM provides the option to override the default values for various settings by using the external memory read data bus. The boot ROM knows to override default values if it reads a 16-bit, non-zero value from address 0x05fffe-f, which resides in external memory. At boot-up, the boot ROM writes 0x0000 to the external memory address 0x05fffe and subsequently reads a 16-bit data value from the same external memory address. If external SRAM is hooked up to the WIMS Microcontroller's external memory read bus, then the boot ROM should read the 0x0000 that it just wrote and will proceed normally. If a flash memory is hooked up, then the write of 0x0000 should have no effect and the boot ROM will read whatever value was previously programmed at 0x05fffe. Program 0x0000 if you want the boot ROM to proceed normally. If no memory is hooked up, then the user has the option of manually tying the external memory read data bus' pins to specific values which the boot ROM then uses to override certain default settings. The boot ROM uses the following pin mappings to override the settings specified in

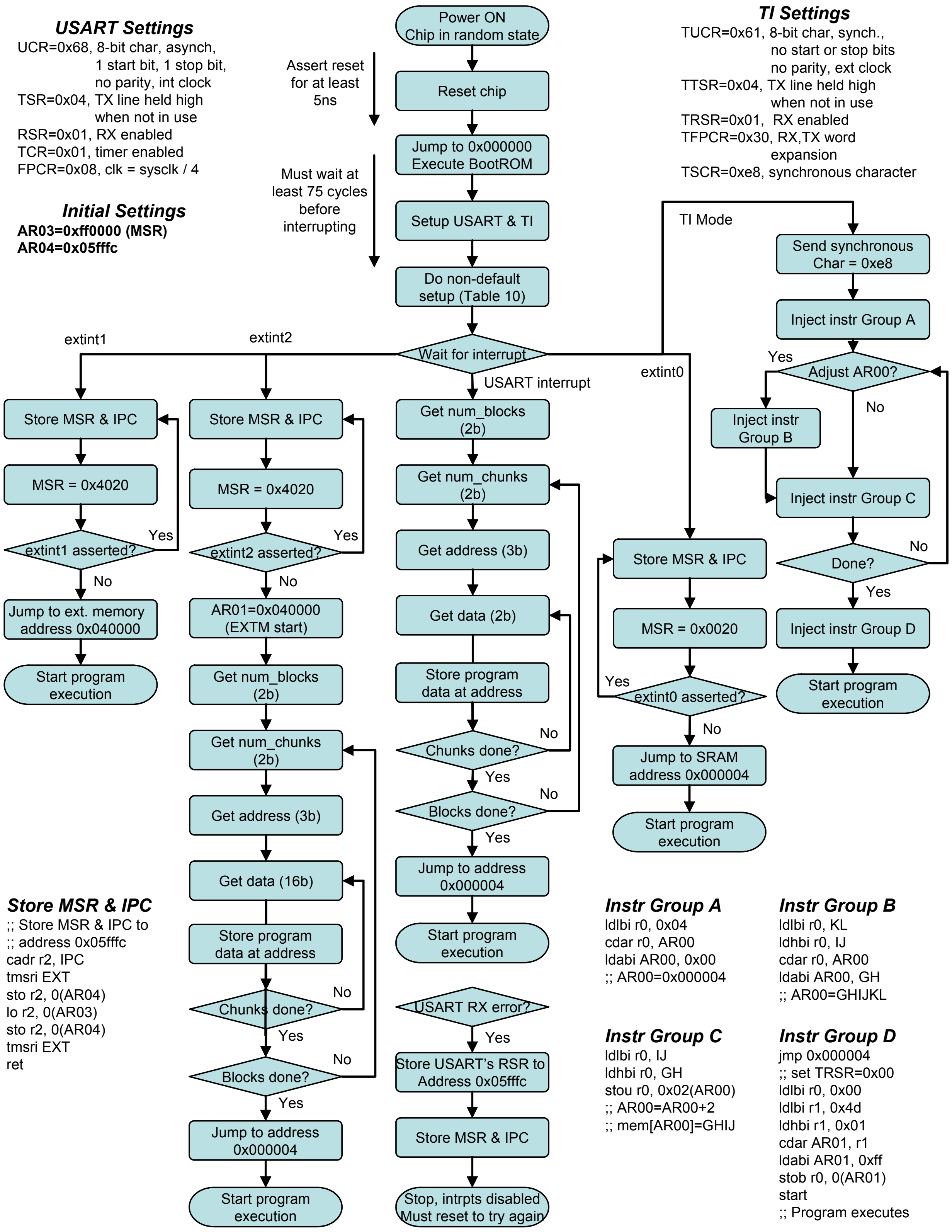
Table 10: Pin Mappings for Non-Default Boot-Up Values

extmemrddataPadMMU_BPin bits	Mapped to value/setting
[15:12]	On-chip clock's LCCal[7:4]
[11:9]	On-chip clock's CSR[2:0]
[8]	Unused
[7:5]	USART: FPCR's PSC bits
[4]	USART: UCR's CLKS bit - off-chip (1)/on-chip (0)
[3]	USART: UCR's ST bits - asynch(0)/synch(1) mode
[2:0]	Unused

TODO: Insert table of registers and startup values (PC, MSR, pipeline regs, etc.). The following modules use reset: alu, clkgen, ex, id, if, mmrf, mmu, peripherals.

Fig. 2 is a flow chart of possible boot up scenarios. It contains start-up values for several peripheral registers as well as well as some instruction groups inserted by the simulation scripts.

Figure 2: Boot Up Flow Chart



Chapter 9. Memory

The WIMS Microcontroller supports a 24-bit fully unified address space that is allocated according to mapping in Table 11.

SRAM

The on-chip SRAM is subdivided into single-port memory banks that can be accessed (loaded from or stored to) in a single cycle. Due to the fact that each of the SRAMs have only one port, the programmer should attempt to restrict data memory accesses and program memory accesses to *different* RAMs so as to prevent unnecessary pipeline stalling and reduced performance. For example, if a store instruction in the execute stage of the pipeline attempts to store to address 0x001000 while the program is attempting to fetch the next instruction from address 0x00500 then the fetch will be stalled while the store proceeds. This is because they are both attempting to access the same memory bank (BANK_0), and the current executing instruction must be given preference over the fetch. Both the store and the fetch could have proceeded simultaneously had the programmer stored to a different memory bank.

Boot ROM

The Boot ROM is split into two parts, as shown in Table 11. This was done to avoid wasting BANK_0's available address space. When the chip is reset, the PC is set to 0x000000 and it reads from BOOTROM1 (which contains only a jmp 0xff0200 instruction) which then jumps to BOOTROM2 and the remainder of the Boot ROM code is executed. By splitting the Boot ROM in such a manner, only 4 bytes of RAM BANK_0 is wasted (unmapped) instead of hundreds of bytes if we had put the entire Boot ROM starting at 0x000000. It was deemed most important that each of the RAM banks be aligned on 8KB address boundaries.

External Memory

The BANK_EXT address range maps to the external memory pins which can be used to connect to an external memory bank. The EXT bit in the MSR must be set in order to use the external memory bus, otherwise load or store accesses to the external memory address range will throw an invalid memory exception. If the extmemsel input pad is not set, then the 16 external memory write data output pins will be connected to the general purpose digital output register that is memory-mapped in Table 12. The general purpose output port (GPOP) can be read or written with any load or store operation to the appropriate address. As a test feature, if extmemtestsel is set, the external memory write data will be the execute stage write data and the external memory address and external memory write enable will be a portion of the current PC value.

Loop Cache

The Loop Cache is not really a typical cache, but a very small bank of memory that consumes much less power than an access to a regular memory bank. To produce the most power efficient code, the compiler will profile code and put the most commonly executed instructions or most commonly accessed data in this area of memory. An access to the loop cache consumes slightly less than 50% of the energy that an access to the SRAM consumes.

Memory Mapped Registers and Peripherals

The remainder of the Table 11 is fairly self-explanatory, with most of the mappings corresponding to the various memory mapped address registers or control registers. The lower 2-bytes of the 3-byte registers are even aligned so that word load/stores can be used to read/write these registers without generating an alignment exception. This resulted in several of the unused odd addresses being mapped to '00' as shown. Stores to these addresses have no effect and loads read the value '00'. Load/stores to the unused portions of memory denoted in Table 11 will result in an invalid memory exception as indicated. The mappings for the microcontroller's peripherals are detailed in Table 12.

Table 11: Core Memory Map

Address Range		Mnemonic	Description
Start	End		
0x000000	0x000003	BOOTROM1	Boot ROM (jump to BOOTROM2)
0x000004	0x001fff	BANK_0	7.996KB memory bank 0
0x002000	0x003fff	BANK_1	8KB memory bank 1
0x004000	0x005fff	BANK_2	8KB memory bank 2
0x006000	0x007fff	BANK_3	8KB memory bank 3
0x008000	0x0081ff	LOOP	512 Byte Loop Cache - DSP LUT
0x008200	0x0083f3	DSP	See Chapter 14
0x0083f4	0x03ffff		Unused: Generates invalid memory exception
0x040000	0x05ffff	BANK_EXT	128KB external memory
0x060000	0xfeffff		Unused: Generates invalid memory exception
0xff0000	0xff0001	MSR[15:0]	Machine State Register
0xff0002	0xff0003	IMSR[15:0]	Interrupt Machine State Register
0xff0004	0xff0005	IVT[15:0]	Interrupt Vector Table
0xff0006		BPM[1:0]	Break Point Mode (See Table 9)
0xff0007	0xff0009	BPR[23:0]	Breakpoint Register
0xff000a	0xff000b	SDAR[15:0]	Shift/Divide Auxiliary Register
0xff000c	0xff000d	GPOP[15:0]	General Purpose Output Port
~~0xff000e~~		~~DTCID[4:0]~~	~~DMA Target Chunk ID #~~
~~0xff000f~~		~~DNC[4:0]~~	~~DMA Number of Chunks~~
~~0xff0010~~		~~DBR[3:0]~~	~~DMA Base Register~~
~~0xff0011~~		~~DINC[0]~~	~~DMA Increment/Decrement Index~~
~~0xff0012~~		~~IPR0[4:0]~~	~~Interrupt Priority Register 0 (ExtInt0)~~
~~0xff0013~~		~~IPR1[4:0]~~	~~Interrupt Priority Register 1 (ExtInt1)~~
~~0xff0014~~		~~IPR2[4:0]~~	~~Interrupt Priority Register 2 (ExtInt2)~~
~~0xff0015~~		~~IPR3[3:0]~~	~~Interrupt Priority Register 3 (Timer0)~~
~~0xff0016~~		~~IPR4[3:0]~~	~~Interrupt Priority Register 4 (Timer1)~~
~~0xff0017~~		~~IPR5[3:0]~~	~~Interrupt Priority Register 5 (Timer2)~~
~~0xff0018~~		~~IPR6[3:0]~~	~~Interrupt Priority Register 6 (SPI0)~~
~~0xff0019~~		~~IPR7[3:0]~~	~~Interrupt Priority Register 7 (SPI1)~~
~~0xff001a~~		~~IPR8[3:0]~~	~~Interrupt Priority Register 8 (SPI2)~~
~~0xff001b~~		~~IPR9[3:0]~~	~~Interrupt Priority Register 9 (USART)~~
~~0xff001c~~		~~IPR10[3:0]~~	~~Interrupt Priority Register 10 (TI)~~
~~0xff001d~~			~~Unused: Reads 00~~
0xff001e	0xff007f		Unused: Generates invalid memory exception
~~0xff0080~~	~~0xff009d~~	~~PLRs~~	~~Pipeline Registers memory map (See Table 8)~~
0xff009e	0xff00ff		Unused: Generates invalid memory exception
0xff0100	0xff01ff	PER	Peripherals' memory map (See Table 12)
0xff0200	0xff0fff	BOOTROM2	Boot ROM
0xff1000	0xffffff		Unused: Generates invalid memory exception

The registers that have strike-through were not implemented in Gen-2 and are invalid memory locations.

Table 12: Peripheral Memory Map

Address Range		Mnemonic	Description
Start	End		
0xff0100	0xff0101	TM0CR[15:0]	Timer0 Control Register
0xff0102	0xff0103	TM0I0[15:0]	Timer0 Image Register Zero
0xff0104	0xff0105	TM0I1[15:0]	Timer0 Image Register One
0xff0106	0xff0107	TM0C0[15:0]	Timer0 Counting Register Zero
0xff0108	0xff0109	TM0C1[15:0]	Timer0 Counting Register One
0xff010a	0xff010b	TM0PS[7:0], TM0SR[7:0]	Timer0 Prescale, Status (Read) Register
0xff010c	0xff010d	TM1CR[15:0]	Timer1 Control Register
0xff010e	0xff010f	TM1I0[15:0]	Timer1 Image Register Zero
0xff0110	0xff0111	TM1I1[15:0]	Timer1 Image Register One
0xff0112	0xff0113	TM1C0[15:0]	Timer1 Counting Register Zero
0xff0114	0xff0115	TM1C1[15:0]	Timer1 Counting Register One
0xff0116	0xff0117	TM1PS[7:0], TM1SR[7:0]	Timer1 Prescale, Status (Read) Register
0xff0118	0xff0119	TM2CR[15:0]	Timer2 Control Register
0xff011a	0xff011b	TM2I0[15:0]	Timer2 Image Register Zero
0xff011c	0xff011d	TM2I1[15:0]	Timer2 Image Register One
0xff011e	0xff011f	TM2C0[15:0]	Timer2 Counting Register Zero
0xff0120	0xff0121	TM2C1[15:0]	Timer2 Counting Register One
0xff0122	0xff0123	TM2PS[7:0], TM2SR[7:0]	Timer2 Prescale, Status (Read) Register
0xff0124	0xff0125	SPI0CR[15:0]	SPI0 Control Register
0xff0126	0xff0129	SPI0DO[31:0]	SPI0 Dataout Register
0xff012a	0xff012d	SPI0DI[31:0]	SPI0 Datain Register
0xff012e	0xff012f	SPI1CR[15:0]	SPI1 Control Register
0xff0130	0xff0133	SPI1DO[31:0]	SPI1 Dataout Register
0xff0134	0xff0137	SPI1DI[31:0]	SPI1 Datain Register
0xff0138	0xff0139	SPI2CR[15:0]	SPI2 Control Register
0xff013a	0xff013d	SPI2DO[31:0]	SPI2 Dataout Register
0xff013e	0xff0141	SPI2DI[31:0]	SPI2 Datain Register
0xff0142		SCR[7:0]	USART Synchronous Character Register
0xff0143		UCR[7:0]	USART Control Register
0xff0144		TSR[7:0]	USART Transmit Status Register
0xff0145		RSR[7:0]	USART Receive Status Register
0xff0146		RDFR[7:0]	USART Rx Data (Read)/Tx FIFO Register (Write)
0xff0147		TDFR[7:0]	USART Tx Data (Write)/Rx FIFO Register (Read)
0xff0148		FPCR[7:0]	USART FIFO & Prescale Control Register
0xff0149		TCR[7:0]	USART Timer Control Register
0xff014a		TSCR[7:0]	TI Synchronous Character Register
0xff014b		TUCR[7:0]	TI Control Register
0xff014c		TTSR[7:0]	TI Transmit Status Register
0xff014d		TRSR[7:0]	TI Receive Status Register
0xff014e		TRDFR[7:0]	TI Rx Data (Read)/Tx FIFO Register (Write)
0xff014f		TTDFR[7:0]	TI Tx Data (Write)/Rx FIFO Register (Read)
0xff0150		TFPCR[7:0]	TI FIFO & Prescale Control Register
0xff0151		TTCR[7:0]	TI Timer Control Register
0xff0152	0xff0153	CSR[15:0]	Clock Select Register
0xff0154	0xff0155	CUC[15:0]	Clock User Control
0xff0156	0xff0157	CDC[15:0]	Clock Debug Control
0xff0158	0xff01ff		Unused: Generates invalid memory exception

Chapter 10. Universal Synchronous/Asynchronous Receiver-Transmitter

Introduction

The WIMS Universal Synchronous Asynchronous Receiver-Transmitter (USART) is a full-duplex serial channel with double-buffered receiver and transmitter modules. The design is functionally modeled after the Motorola 68HC901 microcontroller USART peripheral, a highly-programmable full-featured USART. There are separate transmit (Tx) and receive (Rx) clocks, interrupt channels, data bytes, and status registers. There exists a single interrupt channel for each Rx/Tx section, simultaneously indicating both normal and error event conditions. The USART can also be placed in a 16-bit word mode, where the Rx/Tx buffers expand to allow 16-bit loads and stores of serial data.

Character Protocols

The USART supports both asynchronous and synchronous character formats. These formats are selected independently of the divide-by-one and divide-by-16 clock input modes to each Tx/Rx block, although they are based upon the RS-232 asynchronous protocol and the synchronous character protocol. This feature allows one to clock databits into the USART synchronously while using RS-232 start and stop bits to frame transmissions. In such a mode, all the asynchronous format rules apply.

When divide-by-one clock mode is selected, synchronization must be accomplished externally. The Receiver samples data on the rising edge of the (receiver) clock. In divide-by-16 mode, the Rx input clock frequency is 16 times greater than the data clock cycle, allowing mid-databit sampling after eight Rx clock edges. This increases transient noise rejection.

Asynchronous Format

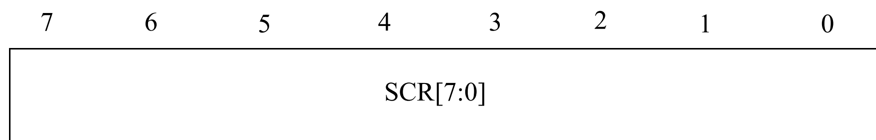
The asynchronous character format follows the RS-232 protocol in most respects, providing software controllable character length, one to two stop bits, and multiple parity options. Character lengths of 5, 6, 7, and 8-bits are selectable by the programmer through the control register. Stop bit lengths of one or two bits help ensure compatibility with other UARTs and protocols. Additionally, error checking in the form of even or odd parity can be enabled if desired.

While in the asynchronous mode, start bit detection is always enabled. This means that an inactive Rx/Tx line should be held high, and that no new data will be shifted in until a one-to-zero start bit transition is detected. As of WIMS tapeout 1, there is no false start bit detection. Later revisions, however, validate a start bit only if it remains zero for the first half of the shift cycle, or the first 8 receiver clock edges.

Synchronous Format (TODO: Expand on this more)

Once the synchronous character format has been selected, received serial data is constantly compared against a preloaded 8-bit synchronous character register (SCR). Synchronization occurs once the incoming data matches the SCR. The SCR compares against the full 8-bits, but smaller character lengths will match if the SCR is loaded with the unused significant bits zeroed out. Upon synchronization, data is clocked into the receiver as character-sized binary units. All incoming characters are valid data unless the user enables SCR stripping, in which case incoming characters matching the SCR are removed from the data stream. During an underrun condition, the transmitter continuously sends the SCR until data becomes available.

Figure 3: SCR - Synchronous Character Register



All bits of the SCR are read and writable and reset low. The SCR should only be written after the character length is selected, and any unnecessary significant bits should be cleared beforehand. In contrast, the parity enable can be tog-

gled at any time, as SCR parity is dynamically calculated. Once parity has been enabled, however, the total character length increases by one bit, making a synchronous word equal to the character length plus one.

The USART Control Register (UCR)

The UCR provides the central control signals for both the Tx and Rx units. Both the clock division mode, and the clock source can be selected by writing this register. Character format, length, and error correction are also assigned through this register. All bits of the UCR are read and writable and reset low.

Figure 4: UCR - USART Control Register

7	6	5	4	3	2	1	0
CLKD	CL[1:0]		ST[1:0]		PE	E/O	CLKS

CLKD - Clock Division Mode (R/W)

- 1 = Data shifted at 1/16th the frequency of the Rx/Tx clock input
- 0 = Data shifted at the frequency of the Rx/Tx clock input

CL1:0 - Character Length (R/W)

This bit pair specifies the number of bits to be added the minimum character length of five.

- 00, 01, 10, 11 = 5, 6, 7, 8-bit words

ST1:0 - Start/Stop Bit Selection (R/W)

ST0 selects asynchronous format when set, synchronous mode when cleared

ST1 selects two stop bits when set, but only when in asynchronous mode

Table 13: Format Control - Start/Stop Bit Selection

ST1	ST0	Start Bits	Stop Bits	Format
0	0	0	0	Synchronous
0	1	1	1	Asynchronous
1	0	0	0	Synchronous
1	1	1	2	Asynchronous

PE - Parity Bit Enable (R/W)

When enabled: Tx inserts/Rx expects an additional “parity bit” after the MSB of a data word.

- 1 = Rx checks parity bit for all received characters, Tx calculates and inserts parity bit for transmission
- 0 = No parity bit insertion or parity checking

E/O - Even or Odd Parity (R/W)

When parity is enabled, setting this bit implies even parity for all data words.

- 1 = Even Parity - all words should have an even number of high bits, including the parity bit
- 0 = Odd Parity - all words should have an odd number of high bits, including the parity bit

CLKS - Clock Select

Selects which clock signal determines the baud rate or synchronous clock edges

- 1 = The off-chip input clock
- 0 = The on-chip generated clock signal

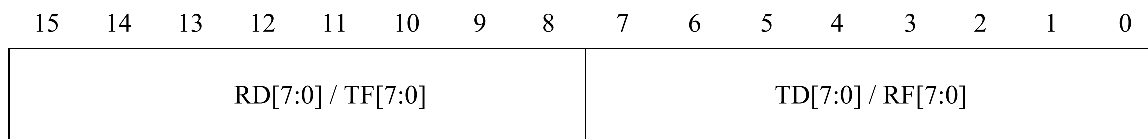
The USART Data Registers (UDR[15:0] = RDFR[7:0] + TDFR[7:0])

These registers provide read access to the data word in the receiver buffer or data word write access to the transmit buffer. The USART allows the user to define the desired R/W size as either 8- or 16-bits; therefore two bytes of Rx/Tx data are made to be addressable individually, or simultaneously.

Each data byte register is actually the union of two separate registers, one for writing to the transmitter, the other for reading from the receiver. The high data byte is referred to as the Receiver Data and Transmitter FIFO Register

(RDFR), where one can read the contents of the receive data buffer or write to the transmitter FIFO. The low data byte register is referred to as the Transmitter Data and Receiver FIFO Register (TDFR), where one can read the contents of the receive FIFO register or write to the transmitter data buffer register. In the previous explanations, it is understood that a Rx/Tx data buffer register acts as the primary link to the Rx/Tx shift register, and the FIFO registers provide an additional byte for either byte-FIFO or 8-to-16-bit word expansion purposes. Both registers reset low.

Figure 5: RDFR and TDFR



The RDFR byte register comprises the most significant (address/databus) byte of the two data registers. In 16-bit word mode, this is significant because the most significant byte is the last byte transmitted, but the first byte received. This aligns bytes transmitted according to the LSB to MSB transmission convention most USARTs assume. In single byte mode, which is the mode most USARTs operate in, only RDFR is utilized for buffering incoming data, and only TDFR can be successfully written to for data transmission. All bits of the RDFR and TDFR reset low.

RD[7:0], RF[7:0] -- Receiver Data (first byte received, last byte received) (R/W)
- 1 = Data bit set
- 0 = Data bit cleared

TF[7:0], TD[7:0] -- Transmit Data (last byte sent, first byte sent) (R/W)
- 1 = Data bit set
- 0 = Data bit cleared

Clock Sources

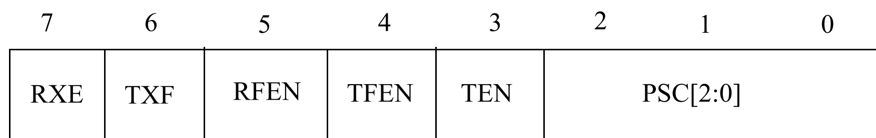
The USART has two primary clock domains. The host system supplies a clock for control and data interfacing purposes, which is referred to as the control clock domain (*sys_clk)*. The second clock domain determines the serial communication rates of the receiver (Rx) and transmission (Tx) modules, the frequency of which is commonly known as the baud rate, measured in bits per second (bps). The baud rate is in turn determined by both the clock inputs to the Rx/Tx modules and the clock multiplier setting for the modules (1x or 16x). The Rx/Tx clock source itself can be an external clock synchronization pin or internally generated from the system clock via the USARTs prescaler and programmable 8-bit timer.

To program the transmission rates one must sequentially choose the transmission format, Rx/Tx clock division mode, and finally program both the prescale (FPCR) and timer (TCR) control registers to appropriate division settings. Instructions for determining an appropriate clock setting can be found in the following two sections.

FIFO and Prescaler Control Register (FPCR)

The FIFO and Prescale Control Register, FPCR, allows the user to select the data word size for either transmission, reception, or both. It also provides additional FIFO status signals, not provided by the Rx/Tx status registers, when the byte-to-word expansion have been enabled. Finally, the lower nibble controls the USART clock generation circuitry. All bits reset low except RXE and all are read and writable except for RXE and TXF. Resets to 0x80.

Figure 6: FPCR - FIFO and Prescale Control Register



RXE -- Receiver word-expansion FIFO Empty (R)

- 1 = Rx FIFO byte empty
- 0 = Rx FIFO byte not empty (TODO: does RXE reset low? if so, how)

TXF -- Transmitter word-expansion FIFO Full (R)
- 1 = Tx FIFO byte full
- 0 = Tx FIFO byte not full

RFEN -- Receiver word-expansion FIFO Enable (R/W)
When the user enables the receive FIFO, the primary receive data buffer (RDFR) overflows into the TDFR register when a second data byte has been received. Only once both the Rx data buffer and the RX FIFO are full does the Rx service request cause an interrupt. If polling is necessary, the software must check FPCR[7] (RXE) instead of the RSR[7] (BF) bit in order to determine whether a 16-bit word has been assembled.
- 1 = Rx FIFO enabled; $\overline{RFE}$ used as "buffer full" interrupt source
- 0 = Rx FIFO disabled

TFEN -- Transmitter word-expansion FIFO Enable (R/W)
When the user enables the transmit FIFO, bytes written to the RDFR shift into the transmit buffer (TDFR) before they are written to the serial transmit shift register. Therefore overwriting TDFR before the Tx Buffer Empty flag asserts not only jeopardizes the original TDFR data, but the RDFR FIFO data byte as well. Because of this arrangement, the Transmit Buffer Empty flag source does not change from the TDFR in either byte or 16-bit word mode.
- 1 = Tx FIFO enabled
- 0 = Tx FIFO disabled

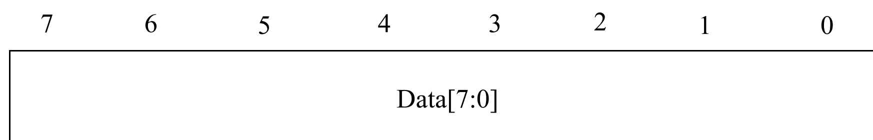
TEN -- Clock generation timer enable (R/W)
Setting this bit enables a programmable 8-bit timer that generates the USART internal Rx/Tx clock source. The 8-bit prescaler output feeds the timer's input.
- 1 = enable programmable timer
- 0 = disable timer

PSC[2:0] -- Low-power 8-bit prescaler control (R/W)
These three bits can set the prescaler to simply pass the clock, or divide the incoming system clock by 2 to the N power. N is equal to the PSC setting plus two, not including the case where all bits are cleared. Clearing the prescale bits to zero disables the prescaler and passes the system clock directly to the 8-bit timer.

Timer Control Register (TCR)

The Timer Control Register, TCR, holds the count length settings. All bits are read and writable and reset low. To enable the clock generation timer one must set the TEN bit in the FIFO and Prescale Control Register, FPCR[3]. The count length itself determines the period of the USART internal clock, which always has a 50% duty cycle. In order to determine the final Rx/Tx input clock frequency, divide the prescale clock output by twice the sum of the count length plus one. All bits reset low. E.G. The prescaler outputs a 10 MHz clock and the TCR is 0x0f, so the Rx/Tx clock input is 312.5 kHz

Figure 7: TCR - Timer Control Register



Data[7:0] -- Timer Setting Data (R/W)
See Table 14 equations.

Table 14: Baud Rate Equations

Prescaler Only	Timer Only
$(f_{clk}) / (8 \cdot \mathrm{PSC}[2{:}0])$	$(f_{clk}) / (2 \cdot (\mathrm{TCR}[7{:}0] + 1))$

The equations for calculating the baud rate can be found in Table 14. One must also divide the expression by the currently selected clock mode multiplier (UCR[7]), 1x or 16x. Additionally, the Prescaler and Timer expressions should be multiplied together if both the prescaler and TCR are enabled. However, the factor of 16, if enabled, does get multiplied into the new clock frequency twice.

Table 15 below illustrates the baud rate ranges achievable for each prescale setting versus sample system clock frequencies, the exact baud depending on the USART timer setting. Note that the baud rate must be below the system clock frequency divided by four, so that maximum frequencies in the first row are always (*sys_clk* / 4).

Table 15: Allowable Clock Generation Baud Rate Ranges for all Clock Modes

PSC	Sys Clk Divider	Allowable Baud Rate Ranges in Kbps (MAX/MIN) vs. System Clock Speeds*											
		1MHz		10MHz		50MHz		100MHz		150MHz		200MHz	
0x0	1	$0.25e^3$	0.122	$2.5e^3$	1.221	$12.5e^3$	6.104	$25e^3$	12.207	$37.5e^3$	18.311	$50e^3$	24.414
0x1	8	125	0.015	$1.25e^3$	0.153	$6.25e^3$	0.763	$12.5e^3$	1.526	$18.75e^3$	2.289	$25e^3$	3.052
0x2	16	62.5	0.008	625	0.076	3125	0.381	6250	0.763	9375	1.144	12500	1.526
0x3	32	31.25	0.004	312.5	0.038	1562.5	0.191	3125	0.381	4687.5	0.572	6250	0.763
0x4	64	15.625	0.002	156.25	0.019	781.25	0.095	1562.5	0.191	2343.7	0.286	3125	0.381
0x5	128	7.813	0.001	78.125	0.010	390.63	0.048	781.25	0.095	1171.9	0.143	1562.5	0.191
0x6	256	3.906	0	39.063	0.005	195.31	0.024	390.63	0.048	585.94	0.072	781.25	0.095
0x7	512	1.953	0	19.531	0.002	97.656	0.0123	195.31	0.036	292.97	0.072	390.63	0.048
*Exact BPS depends upon the 8-bit timer setting and the clock mode factor													

Receiver (Rx Module)

Data received on the serial input line (Rx) is essentially shifted into an internal 8-bit shift register clocked at a frequency (baud rate) equal to the Rx clock multiplied by the division mode (UCR[7]). Once the programmed bit length is received, the character will be transferred to the receive data buffer if the last character stored to the receive buffer has been read (buffer not full). Transferring a character to the receive buffer sets the buffer full bit of the receiver status register, and will produce a interrupt service request to the processor if interrupts are enabled.

Reading the receive buffer clears the buffer full bit, satisfies the buffer full condition, and allows new characters to be transferred to the receive buffer. To read the receive buffer, one accesses the RDFR register. When the Rx word-expansion FIFO has been enabled, one accesses the Rx FIFO register by reading the TDFR If the word-expansion FIFOs are enabled, one can still access the only the RDFR to satisfy the buffer full condition, but in this case it is not always recommended. This is because the buffer full *condition* implies that both the receive buffer and the receive FIFO register (TDFR) are full, and so the TDFR contains the oldest (first) character. Therefore it is more common to either access the RDFR and TDFR together as a word, or to access the TDFR (and optionally the RDFR afterwards,) so as to mimic a two-deep byte-width FIFO. It is important to understand that setting the RSR buffer full bit only produces a so-called buffer full condition when the Rx FIFO is not enabled, because a buffer full condition implies that both the receive buffer and receive FIFO register have been filled. When interrupts are enabled, only a buffer full condition or a (single character) error condition can trigger the Rx service request interrupt.

Whenever a character is transferred to the receive buffer, status information is stored in the RSR. The RSR is not updated again until data in the receive buffer has been read, with two exceptions. In asynchronous format the real-time character in progress (CIP) bit can change at any system clock edge. Secondly, when the Rx FIFO register is enabled (FPCR[5]), the last character of two to be transferred to the RDFR does not require the current RDFR data to be read as long as the Rx FIFO register can accept the receive buffer data (Rx FIFO empty condition), and the current RSR does not indicate an error condition was incurred during the first character's reception. Finally, note that when the buffer full condition exists, a programmer should always read the RSR before reading either of the data registers. Following this rule ensures that the RSR flags will always correspond to the data being read; otherwise there is the possibility that the RSR flags could change in between reading the data register(s) and subsequently reading the RSR.

The Receiver Status Register (RSR)

As described above, the RSR contains the receiver enable bit, receive buffer full flag, synchronous strip enable, and a number of status bits indicating the state of the Rx module. With the exception of the M/CIP bit, and the case where the Rx FIFO has been enabled, the RSR status bits will not change unless the new data word has been read. All bits reset low and see below for read and writability.

Figure 8: RSR - Receiver Status Register

7	6	5	4	3	2	1	0
BF	OE	PE	FE	F/S-B	M-CIP	SS	RE

BF -- Buffer Full (R)
Receiver character has been assembled and subsequently loaded into the receive buffer. The receive buffer register can be read by reading the RDFR data register. Accessing this register clears BF.

- 1 = Character transferred to receive buffer register
- 0 = RDFR has been read

OE -- Overrun Error (R)
Overrun errors occur when the internal shifter has completed shifting an incoming character, but the receive buffer is full. During the occurrence of an OE, the receive buffer and RSR are not overwritten, so as to provide linear data integrity. However, the OE bit will be set once the buffer full condition has been satisfied (reading RDFR). The OE bit is cleared and the error is acknowledged by reading the RSR. New data words are not assembled while the OE bit is set.

- 1 = Receive buffer full and incoming character received
- 0 = RSR has been read

PE -- Parity Error (R)
Parity is enabled and the incoming character's parity is incorrect.

- 1 = Parity error detected on character transferred to receive buffer
- 0 = No parity error detected on incoming character

FE -- Frame Error (R)
Frame errors occur in the asynchronous format when a non-zero data character does not have the correct stop bit alignment.

- 1 = Frame error detected on character transferred to receive buffer
- 0 = No frame error detected on incoming character

F/S or B -- Found / Search condition (synchronous), Break Detected (asynchronous) (FS is R/W, B is R)
The **F**ound / **S**earch conditions are used in the synchronous format, and correspond to whether the initial synchronization character has been found. When this bit is cleared, the receiver enters search mode, and does not return to a high state until an incoming character matches the SCR. Once synchronized, the bit remains high until manually cleared.

- 1 = An incoming character has matched the SCR (enables the character length counter)
- 0 = Search mode, shifter contents dynamically compared to SCR

The **B**reak bit is used to indicate an asynchronous format break condition. A zero-character with no stop-bits is sufficient to trigger a break condition, which will continue until a non-zero data-bit is received.
- 1 = A break has been received and is still waiting service.
- 0 = No break has been received or break has been received and read already.

M or CIP -- Match (sync), Character in Progress (async) (R)
The **M**atch bit indicates a synchronous character has been received while in the synchronous format.
- 1 = Incoming character transferred to receive buffer matched the SCR
- 0 = Incoming character transferred to receive buffer did not match the SCR

The **CIP** bit indicates that a character is currently being assembled (shifted into Rx) while in asynchronous format.
- 1 = Start bit has been detected, character length counter active
- 0 = Last character's stop bit(s) have been received, character length counter cleared

SS -- Synchronous Character Strip Enable (synchronous format only) (R/W)
- 1 = Incoming characters matching the SCR will not be loaded to the receive buffer (synchronous format)
- 0 = Incoming characters matching the SCR will be loaded to the receive buffer

RE -- Receiver Enable (R/W)
Do not set this bit until the receiver clock source has been selected and programmed if necessary.
- 1 = Receiver operation is enabled
- 0 = Receiver operations are disabled

Receiver Conditions of Note

According to the above descriptions, the user will find that there are certain circumstances relating to the overrun and break conditions that merit further specification. The two examples below can occur, and are resolved similarly to the Motorola MC689HC01:

1. A break can be received during a receive buffer full condition, but does not induce an overrun condition when the RDFR is next accessed. Instead, only the break detect bit B is set.

2. A new character is received during a buffer full condition, and subsequently a break is also received. As the break was received before the receive buffer was read, but cannot set the B bit until the buffer full condition is satisfied, both the OE and B flags will be set once the RDFR has next been accessed.

Transmitter (Tx Module)

Writing the TDFR data register loads the transmission buffer register. The data character in the TDFR will be transferred to the internal Tx shift register once the last character has finished transmission, and the END flag of the Transmitter Status Register (TSR) has been asserted. This transfer will produce a buffer empty condition if either the Tx FIFO register has not been enabled, or the Tx FIFO register cannot load the Tx buffer register because it is empty. If the transmitter completes transmission of the character currently in the shifter before a new character has been written to the transmit buffer register, an underrun error will occur. In the synchronous format this necessitates that the synchronous character be continuously transmitted until the transmit buffer has been written. The asynchronous protocol sends a mark (line held high) until the transmit buffer has been written.

When the transmitter has been disabled and a character is being transmitted, the character will continue until assembly is complete. However, while one can load the transmit buffer while the transmitter is disabled, no characters will begin transmission while the TSR enable bit is cleared. If no characters are in the process of being transmitted, disabling the module will cause operations to cease on the next internal Tx clock edge.

In the asynchronous mode, the transmitter can be programmed to send a break by writing to the Tx status and control register. Like a normal data character, a break will not be sent until any characters currently being transmitted finish. Unlike the convoluted implementation of the Motorola MC689HC01 USART, the timing of the break transmission is fully internally controlled and requires no interrupt services. The transmitter also more strictly adheres to industry (RS-232) definitions for break timing, meaning that a break minimally consists of 2N + 3 zero data bits, where N is

the programmed character length. Additionally improved is the fact that the break will complete transmission in its entirety even if the transmitter is disabled during transmission, preventing serial miscommunications. Resets to 0x90.

Figure 9: TSR - Transmission Status Register

7	6	5	4	3	2	1	0
BE	UE	Res	END	B	H	L	TE

BE -- Buffer Empty (R)
Transmit character has been assembled and subsequently loaded into the receive buffer TODO: shouldn't this read 'has been loaded into the TX shifter?'. The transmit buffer register can be directly loaded by writing the TDFR data register, or indirectly loaded by writing the RDFR register while the Tx FIFO is enabled. In the latter case, the TDFR will be loaded with the RDFR character once the transmit buffer's (TDFR) contents have been transferred to the Tx shifter for transmission.

- 1 = Character in transmit buffer transferred to the internal shift register (TODO: default after reset?)
- 0 = Transmit buffer (TDFR) has been reloaded

UE -- Underrun Error (R)
Underrun errors occur when the internal Tx shifter has transmitted an outgoing character but the transmit buffer has not yet been loaded with a new character.

- 1 = Transmit buffer empty and outgoing character has completed transmission
- 0 = TSR has been read (will remain high even if TSR disabled, but not read) (TODO: default after reset?)

Res -- Reserved, produces a zero when read (TODO: 'or written'?)

END -- End of Transmission (R)
This flag asserts whenever a character transmission has been completed, but a new character has not yet been transferred to the Tx shifter from the transmit buffer so as to continue transmission. Unlike the underrun condition, this is not an error, as this bit will assert EOT even after the transmitter was disabled while a character was undergoing transmission. If the transmit buffer is full and a character was just transmitted, END is set on the next positive system clock edge. The END flag is cleared on the system clock cycle following the beginning of the next character transmission.

- 1 = The transmitter has completed sending the last character, but has not yet begun a new transmission (TODO: default after reset)
- 0 = Character currently being shifted out

B -- Send(ing) Break (Asynchronous format only) (R/W)
When B is set, BE will not be set nor a new character begin transmission until the break condition has ended. A standard break consists of 2N + 3 zero bits, where N equals the character length (5 + UCR[6:5]). If written while the transmitter is disabled, the break character will be the first character transmitted if/when the transmitter is later enabled.

- 1 = Break Condition, break character is next in transmission priority
- 0 = No break condition, normal transmission enabled

H,L -- Tx High/Low Output State (R/W)
The High and Low settings configure the transmitter serial output line (TX) while the transmitter is in a disabled state. The bits can be set at any time, but new settings will take effect only when the transmitter has been disabled and not currently transmitting. Loopback mode internally connects the Tx output to the Rx input, and will maintain the connection once the transmitter is re-enabled. The internal loopback configuration is useful for testing the USART.

Table 16: Transmitter Output Settings

H	L	Format
0	0	High Z
0	1	Low
1	0	High
1	1	Loopback

TE -- Transmitter Enable (R/W)
Once the transmitter is enabled, the output will continue to be driven by the H/L bits until transmission begins.

- 1 = Transmit unit enabled (TODO: default after reset?)
- 0 = Transmit unit disabled (TODO: default after reset?)

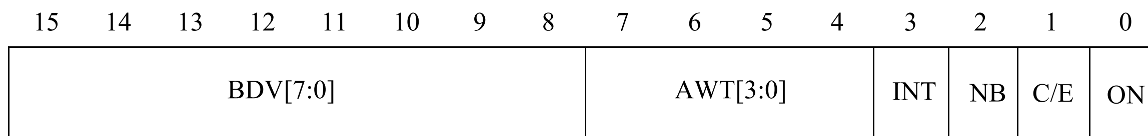
USART Interrupt Channels

The USART has two interrupt channels, the minimal number for bypassing the need for software polling. The assertion of a USART interrupt line indicates that either the receiver or transmitter requires service. In the case of the receiver, a buffer full condition and/or an error condition will trigger an interrupt service request, asserting the first of the two interrupt lines. The transmitter triggers an interrupt service request if either the BE or UE flag is set while the transmitter is enabled. Unlike the MC689HC01, the TSR END bit does not (currently) trigger an interrupt, in part because interrupts are only generated when the Rx or Tx units are enabled. The MC689HC01 utilizes the END bit only while the transmitter is disabled, as opposed to indicating transmission status while enabled as well. For some host systems generating an interrupt while a unit is disabled could cause problems, especially if there is not a high level of selective interrupt control in place.

Chapter 11. Serial Peripheral Interface (SPI)

The SPI block performs the communication protocols for the Cochlear, Neural, and Environmental Testbeds. Fig. 10 shows the different parts of the control register which are described here.

Figure 10: SPICR - SPI Control Register



BDV -- Baud Division Value (R/W)

The BDV defines how many microcontroller cycles is equal to one cycle on the SPI DCLK. Assuming a 40MHz microcontroller clock, it is capable of transmitting between 10MHz and 3.2KHz. Table 17 shows the equation for the DCLK rate.

AWT -- Acknowledge Wait Time (R/W)

The AWT defines the number of SPI DCLK cycles to wait for an acknowledge (NACK) from the slave device before generating an SPI NACK interrupt. 0x0 implies that you are ignoring the NACK signal and no interrupt will be generated. This is useful when communicating with the off-chip ADC. 0x1 implies wait one cycle and 0xf implies wait 16 cycles, and if the NACK signal has not arrived by the end of the specified acknowledge wait time then an SPI NACK interrupt will be generated.

INT -- Interrupt Enable (R/W)

Controls if the SPI unit will send an interrupt to the core when a transmission is complete.

- 1 = Interrupt will be sent
- 0 = Interrupt will not be sent

NB -- Number of Bytes (R/W)

NB defines how many bytes are sent in one transfer (how many bytes of the DOUT register are read and how many bytes are sent into the DIN register). In Cochlear mode, it selects between 2 or 4 bytes. In Environmental mode it chooses between 3 or 4 bytes.

- 1 = Higher of two options
- 0 = Lower of two options

C/E -- Cochlear / Environmental (R/W)

The selection between Cochlear and Environmental protocols is done by the C/E bit. High implies Environmental mode.

- 1 = Environmental mode - 3 or 4 bytes
- 0 = Cochlear mode - 2 or 4 bytes

ON -- SPI On (R/W)

The SPI block is activated (a transmission started) by setting the ON bit. This bit is automatically reset by the hardware when the proper number of bytes is sent or when an interrupt occurs.

- 1 = Transmission started, in progress
- 0 = SPI off

Table 17: SPI Clock Rates

DCLK
$(0.5 \bullet f_{clk}) / (2 + BDV)$

When writing the SPICR, it is important to note that you cannot change the C/E bit and the ON bit with the same write. There is currently no hardware to protect against this. The reason this restriction exists is due to the nature of the two different specifications this block uses. The differences in signal polarity require that the hardware has one cycle to set itself up after the C/E bit has been changed before it can start a transmission. Otherwise the device listening to the SPI block may loose the first transmission. All bits in the SPICR reset low.

Data should be written to the DOUT register before a transmission is started and read from the DIN register, if necessary, after transmission is completed. The DIN register is read only. All other SPI registers are read/write. In the case of less than four byte transfers, DOUT is sent starting with bit[31]. DIN is received into the least significant bit so the higher order bits will be unknown. The DIN and DOUT registers are shown in Fig. 11 and Fig. 12.

Figure 11: DIN - Data In Register

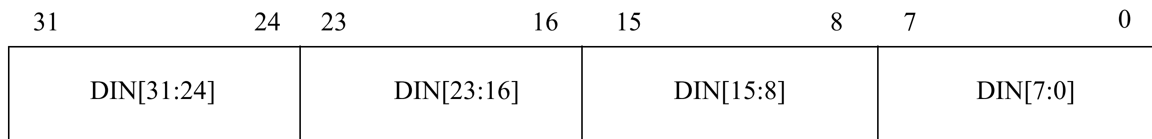


Figure 12: DOUT - Data Out Register

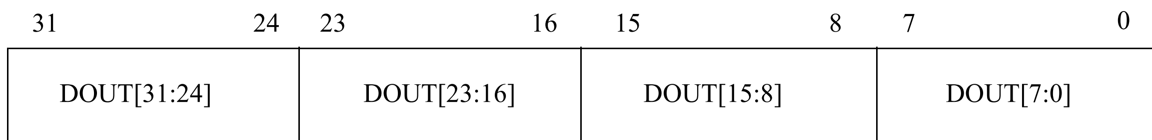


Table 18 shows the signal names used in this document, the cochlear interface, environmental interface, and the AD7705 interface. Each row corresponds to how to hook up one device to the other.

Table 18: SPI Interface Signal Names

SPI	Cochlear	Environmental	ADC
DCLK	DCLK	CLK	SCLK
DIN	DOUT	N/A	DOUT
DOUT	DIN	Data	DIN
NIOE	NIOE	STB	CS_BAR
NACK	NACK	DV	N/A

Fig. 13-15 show signal timing diagrams from simulations for all three types of connections. These are taken from Verilog simulations.

Figure 13: Cochlear Mode Transmission

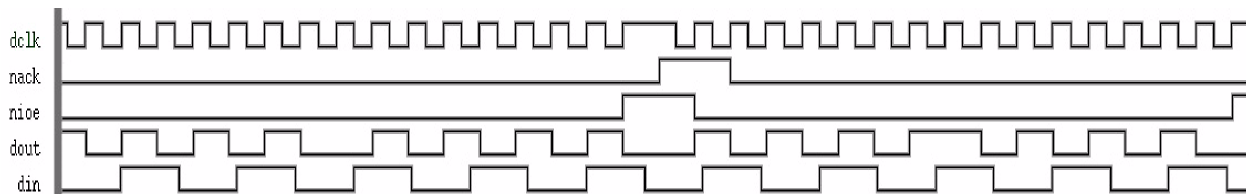


Figure 14: Environmental Mode Transmission

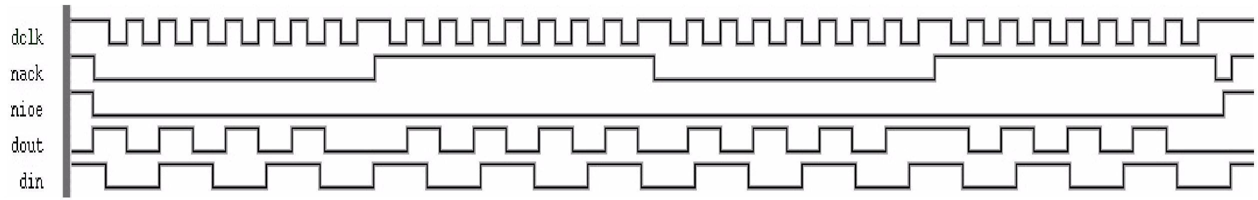
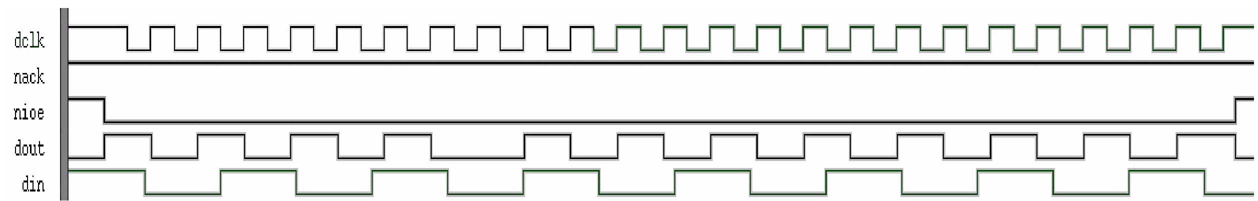


Figure 15: Environmental Mode with AWT = 0x0 (ADC Mode)



Chapter 12. Multifunction Programmable Timer

Introduction

The Multifunction Timer (MT) contains a programmable pair of two counters/timers, each of which can be controlled by software or external pin transitions. The counter/timers aid a programmer in counting external events, timing accurate delays, and generating precise interrupts. Each counting unit (CU) is made up of a command register, a counter, and an image register that facilitates loading and storing timing data. The counting units can be operated individually as counters/timers, or they can be used together to generate pulse-width modulated waveforms. The maximum resolution of the MT is the input clock divided by four.

Each counting unit can be controlled by an input pin (external event) or by software loading/storing to control registers. A single output pin is currently provided for sending a timer output signal off-chip. However, all of the counting units' terminal count outputs are available to the processor for enabling interrupt generation and other timing needs.

Modes of Operation

The Multifunction Timer can be programmed to operate in any one of four separate modes: pulse-width modulation, pulse mode, event counting mode, and time capture mode. Interrupts only occur while in Pulse Mode.

Pulse-Width Modulation (Waveform Mode)

In waveform mode, the timer can produce various pulse-width modulated (PWM) signals. The waveform mode is realized through the combination of both 16-bit counting units. The waveform period is the sum of the counts between the counting units, while the duty cycle depends on the length of the high and low counts relative to each other. The output polarity setting determines which counter will be active high and active low, and therefore which counter's pulse duration will be considered as $Count_{HIGH}$ in equation 2. Once enabled, each counter will load its count duration from the respective image register, and then one CU counts down at a time. Upon finishing the count, a termination signal from the first unit triggers the second unit to load from its image register and begin counting its pulse duration. The duty cycle and waveform itself can be altered simply by loading a new value into the corresponding image register. The image register data will subsequently be loaded into the corresponding counter once a terminal count has been asserted.

$$\text{Period} = \text{Count}_{HIGH} + \text{Count}_{LOW} \quad \text{(eqn 1)}$$

$$\text{Duty} = \text{Count}_{HIGH} / (\text{Count}_{HIGH} + \text{Count}_{LOW}) \quad \text{(eqn 2)}$$

If enabled, the processor can use the terminal count of either counting unit as an interrupt that will be asserted until cleared by the core. Performing a write operation to the controlling TMCR clears the interrupt. Please note that system writes to the image register can only occur after one has disabled the register via the CU's command register. One should re-enable it once the load has been completed.

The contents of an image register are loaded into its corresponding counter under any of the following conditions:

- Terminal Count of CU0 pulses to transfer active status to CU1, and vice versa.
- The input pin pulses (if enabled)
- A command register bit is written to by the processor

Pulse Mode

In Pulse mode, any counting unit is capable of generating a singular pulse. The pulse width is defined by the value loaded into the associated image register of the counting unit. Once the counting unit is activated, the contents of the corresponding counter will be overwritten by the data in the image register. If the CU's output is multiplexed to the output pin, the pulse itself will be sent to the pin. The terminal count, which has a pulse width of one timer clock cycle, is not sent to the output pin, but is used for interrupt generation. When the CU reaches its terminal count, the interrupt will be asserted until cleared by the core. Performing a write operation to the byte of the TMCR that controls the interrupting timer unit will clear the interrupt.

A pulse is triggered and the contents of an image register are loaded to the counter in any of the following events:
An external input pin transition (if enabled)
A command register bit is written to by the processor

Event Counter Mode

In this mode, a CU can be used to count a number of events. An event is defined as a transition on the timer input pin, as defined by the input pin polarity configuration. In this mode, the image register stores the contents of the counter upon the rising edge of any valid load/store trigger signal. (As opposed to the previous two modes where the counter was loaded from the image register.)

The only valid load/store signal in this mode is an input pin event. All counter and image registers must be assigned an initial value. The processor can read the image register value once the image register has been disabled and is therefore stable. While the image register is disabled, the counter will continue to count events unless it too is disabled.

Time Capture Mode

In Time Capture mode, the counting unit can measure the time between events. This is done by counting clock pulses. All counters and image registers should be cleared during initialization. The counting unit can be enabled only by software, after which the CU will continuously count. Each load/store pulse will load the counter's contents to the image register. Valid load/store trigger signals include:

The input pin transitions (if enabled)
A command register bit is written to by the processor

When reading the image register data, disable the register first. Note that the two CU's in time capture mode can be used to capture the rising and falling edges of a pulse, and thereby measure pulse width. Also note that the time between consecutive edges of Time Capture must be greater than one timer clock cycle in order to be captured.

Timer Control Register (TMCR)

The TMCR register(s) provide the control signals for each of the two Counter Units.

Figure 16: TMCR - Timer Control Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TE1	C1IP	OP	DC1	IM1	CR1	C1PM	C1MS	TE0	C0IP	OS	DC0	IM0	CR0	MS[1:0]	

TE0 (TE1) – Pin Trigger Enable 0 (1) (R/W)

There are two sources of control trigger events for each of the four modes:

1. Software load/store, Command register bit written
2. Transition at the input pin (if enabled by software)

- 0 – Only software control trigger
- 1 – Enable external pin trigger events

C0IP (C1IP) – CU0 (CU1) Input Polarity (R/W)

Counter Unit 0 (1) signal input polarity: (TODO: explain this better)

- 0 – active high (input unchanged)
- 1 – active low (input inverted)

OP – Pin Output Polarity (R/W)

Select the polarity of the output pin.

- 0 – output signal unchanged

- 1 – output inverted

DC0 (DC1) – Pseudo Down-Count 0 (1) (R/W)

Forces Counter 0 (1) to decrement rather than increment its value.

Note: The physical implementation does not have Counter 0 (1) decrement, but rather loads/stores a logically inverted binary value of the appropriate datum. I.E. Incrementing inverted values can produce an equivalent timing result as decrementing non-inverted values, all while reducing logic complexity.

- 0 – Increment
- 1 – Decrement via inverted binary values

IM0 (IM1) – Image Register 0 (1) Enable (R/W)

Clearing this bit effectively freezes Image Register 0(1), IMG0 (IMG1), from loading/storing data from/to its respective counter. This is useful when the software wants to read stable data while in event count or time capture mode

- 0 – Disable/Freeze Image Register 0 (1)
- 1 – Enable Image Register 0 (1)

CR0 (CR1) – Counter Register 0 (1) Enable (R/W)

Setting this bit enables Counter 0 (1), CNT0 (CNT1), to begin/resume counting according to the mode selected.

C1PM – CU1 Pulse Mode Enable (R/W)

This bit puts CU1 in Pulse Mode when set.

- 0 – Disable Pulse Mode
- 1 – Enable Pulse Mode

C1MS – CU1 Mode Select (R/W)

This bit determines CU1's operating mode if PWM or Pulse mode has not been enabled via TMCR[9:8] or TMCR[6]. TODO: explain this better

- 0 – Event Count
- 1 – Time Capture

OS – Pin Output Select (R/W)

Select which counter unit's output will be multiplexed to a single timer output pin. This does not prevent the counters' terminal count signals from being used for processor interrupt control, or other internal applications.

- 0 – CU0 output
- 1 – CU1 output

MS[1:0] – Mode Select Bits (R/W)

These bits determine what mode CU0 operates in. As CU0 is the master control CU0 (TODO: typo?) when the timer operates in PWM mode, setting PWM mode in this register causes CU1 to enter PWM mode in default.

- 00 – Event Count
- 01 – Time Capture
- 10 – Pulse Generation
- 11 – Pulse Width Modulation

Timer Image Registers (TMI0, TMI1)

The key purpose of the 16-bit image registers is to facilitate processor access to and control of a counter without asynchronously interrupting the counting function. An image register executes more specific functions depending on the mode of operation. In Waveform and Pulse modes, the image register loads the counter with data upon an appropriate trigger event. In Event Count and Timer Capture modes, the counter value is stored in the image register at the appropriate intervals.

Figure 17: TMI0 - Timer Image Register 0

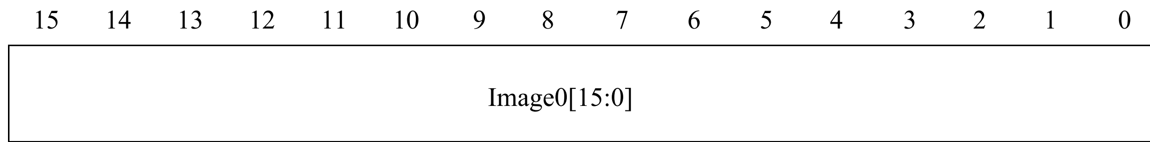
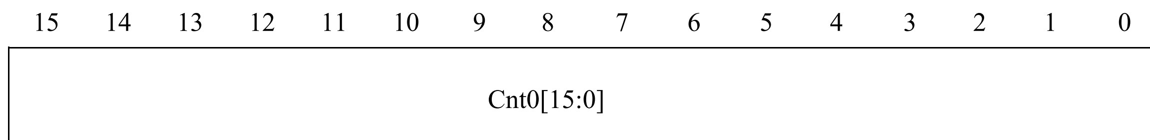


Image Register One follows the same format. Before initializing the command register for Pulse and PWM modes, a program should first (disable and) initialize the image and counter registers to appropriate values. The Image registers reset low and are read and writable.

Timer Counter Registers (TMC0, TMC1)

The 16-bit counters form the core of each counting unit (CU), which is made up of an image register and a counter. Each counter can be initialized, and even loaded in mid operation, although we do not recommend ever doing this outside of initialization routines. A counter will increment or decrement depending on its command register settings, while the mode and trigger event selection determines when and how counting is performed.

Figure 18: TMC0 - Timer Counter Register 0

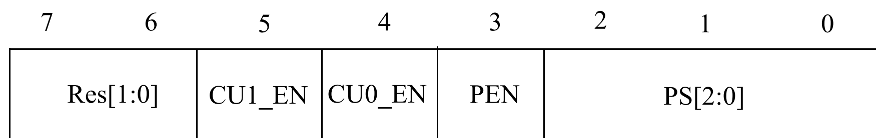


Counter Register One follows the same format. The Counter registers reset low and are read and writable.

Timer Input Clock Prescaler Register (TMPS)

The WIMS Timer has the ability to scale the input clock frequency before it is distributed to the counting, image, and control units. Three control bits determine the amount of clock division, and the prescale enable determines if prescaling is on or off. The active high signals CU0_EN and CU1_EN enable interrupts for each counting unit. The rest of the bits are unused (reserved).

Figure 19: TMPS - Timer Prescale Register



Interrupts are available for Pulse-Width Modulation and Pulse modes only. Event Counter and Time Capture do not have interrupts because they interface with external pins and those pins can be hooked to external interrupt pins if interrupts for these modes are desired. If the counting unit interrupt is enabled and the counting unit is in either of the first two modes, interrupts will be asserted on the terminal count of the counting unit and held until cleared by the core. Clearing the interrupt happens when a write to the proper byte of the TMCR occurs.

This prescaling mechanism is similar to that of the WIMS USART, in that it uses an asynchronous binary toggle counter to continuously scale an input clock. The difference here is the range of the prescaler, which spans 2x division to 256x. Because of the choice of low-activity counter, the divisor is always a power of two between 2-256.

Keep in mind that the system clock is divided by 4 whether or not the prescaler is enabled, which defines the maximum resolution of the timer. One must multiply 4 by the prescaler setting to determine the system clock divisor.

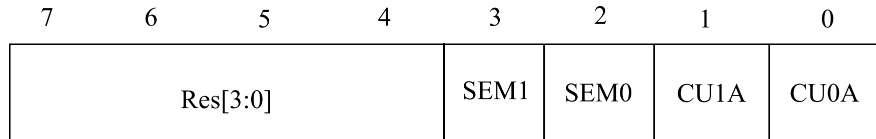
Table 19: Prescale Clock Division Range

Prescale Bits	0XXX	1000	1001	1010	1011	1100	1101	1110	1111
Clock Divisor	4	8	16	32	64	128	256	512	1028

Timer Status Register (TMSR)

The WIMS Timer has this status register for the core to observe the current state of the timer unit.

Figure 20: TMSR - Timer Status Register



CU0A (CU1A) – Counter Unit 0 (1) Active (R)

If this bit is set, it implies that the counting unit is currently counting.

SEM0 (SEM1) – Status Even Miss 0 (1) (R)

If this bit is set, a terminal count or input event occurred that the timer could not handle. The core may need to reprogram or restart the timer, depending on the situation.

Chapter 13. Clock Generation and Clock Source Selection

The WIMS Microcontroller has two possible sources for the clock signal: the on-chip, hybrid LC and ring oscillator clock reference or through the off-chip input pin. The MCU and DSP clock select input pins choose between the off-chip or on-chip sources. The default (high) selects the off-chip clock source. The next section describes the on-chip clock reference and frequency selection.

On-Chip Clock Reference and Frequency Division

The on-chip clock reference generates a signal of $2f_0$, which is given as the input to the circuit in Fig. 21. The $2f_0$ reference clock, can be either 200MHz or 20MHz, depending on whether the LC or ring oscillator is selected. The lower three bits of the CSR (Core_sel in Fig. 22) determine the frequency f_{MCU} for the rest of the microcontroller core. Bits 10:8 in the CSR determine the DSP clock frequency, f_{DSP}. The delay from the time the CSR is set until the new clock frequency reaches the core is $n/2f_0$ + the mux delay, or about 11ns when f_0 = 100MHz. Refer to Appendix Appendix C. for details on the on-chip clock reference.

Figure 21: Clock Selection Circuitry

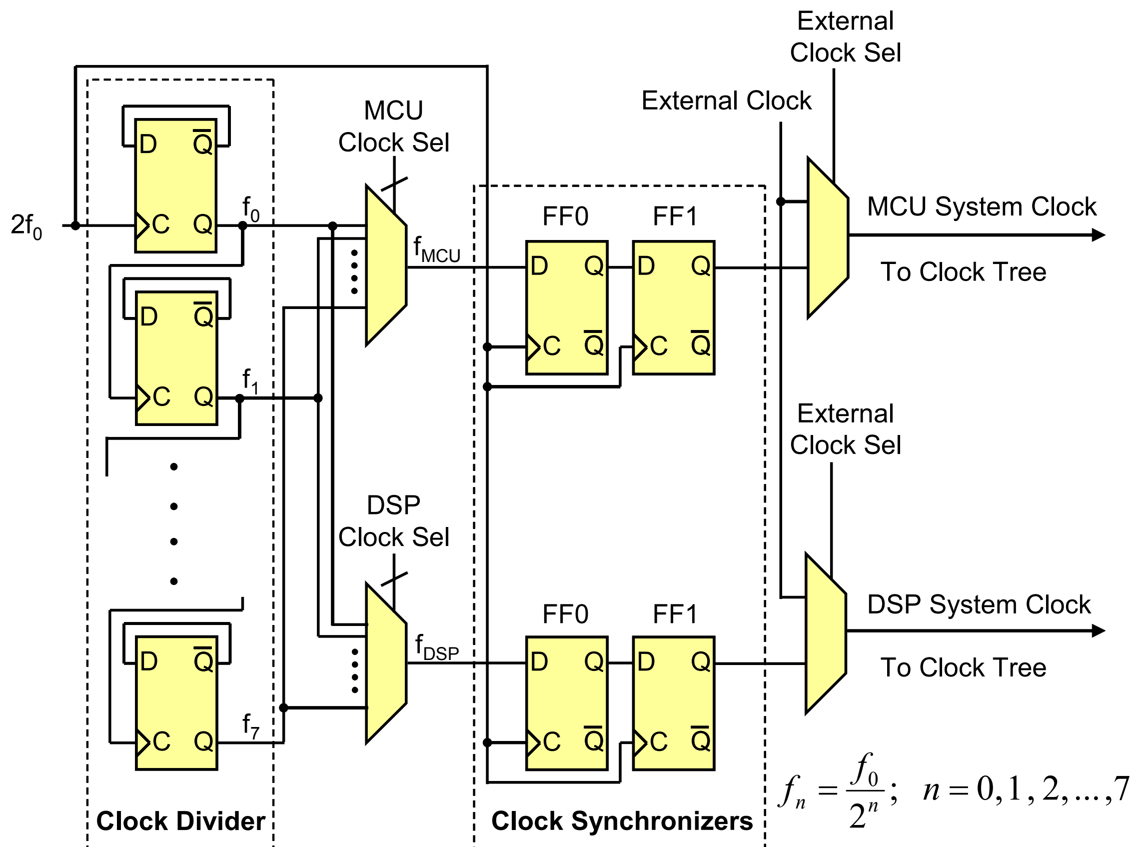


Figure 22: CSR - Clock Select Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res[4:0]					DSP_sel[2:0]			Synch[1:0]		Padsel	Res[1:0]		Core_sel[2:0]		

Core_sel[2:0] –Core clock frequency select (R/W)

The CSR resets to the value 0x1, or $f_{clk}=f_0/2$=50MHz. The other values for Clock_Sel are shown in Table 20.

DSP_sel[2:0] –DSP clock frequency select (R/W)

The CSR resets to the value 0x5, or $f_{clk}=f_0/32$=3.125MHz. The other values for Clock_Sel are shown in Table 20.

Synch[1:0]– Synchronization select (R/W)

The synch[1:0] determines which synchronization clock is used for the clock synchronizer. The other synchronizer inputs aren't shown in the figure and are for testing purposes only. synch[1:0] should be left 0x0.

- 00 – $2f_0$
- 01 – f_0
- 10 – Off chip clk input
- 11 – Off chip clk input - same as 10.

Pad_sel –Pad clock frequency select (R/W)

Pad_sel selects which clock is sent to the pad.

- 0 – clk1 - core clock
- 1 – clk2 - dsp clock

Fig. 23 and Fig. 24 show the mappings for the Clock user and debug control. The details are given in Appendix C.

Figure 23: CUC - Clock User Control

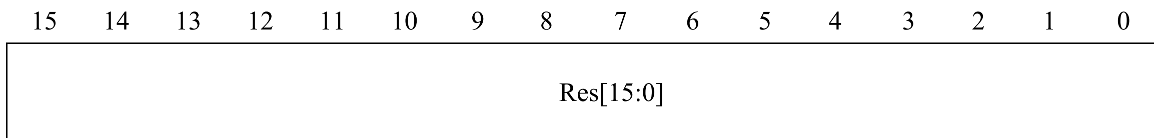


Figure 24: CDC - Clock Debug Control

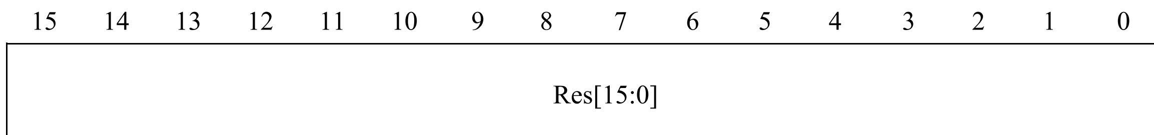


Table 20 gives the f_{clk} values for all possible input values of the CSR. The left assumes LCsel = 1 (LC Oscillator) and the right half assumes LCsel = 0 (Ring Oscillator).

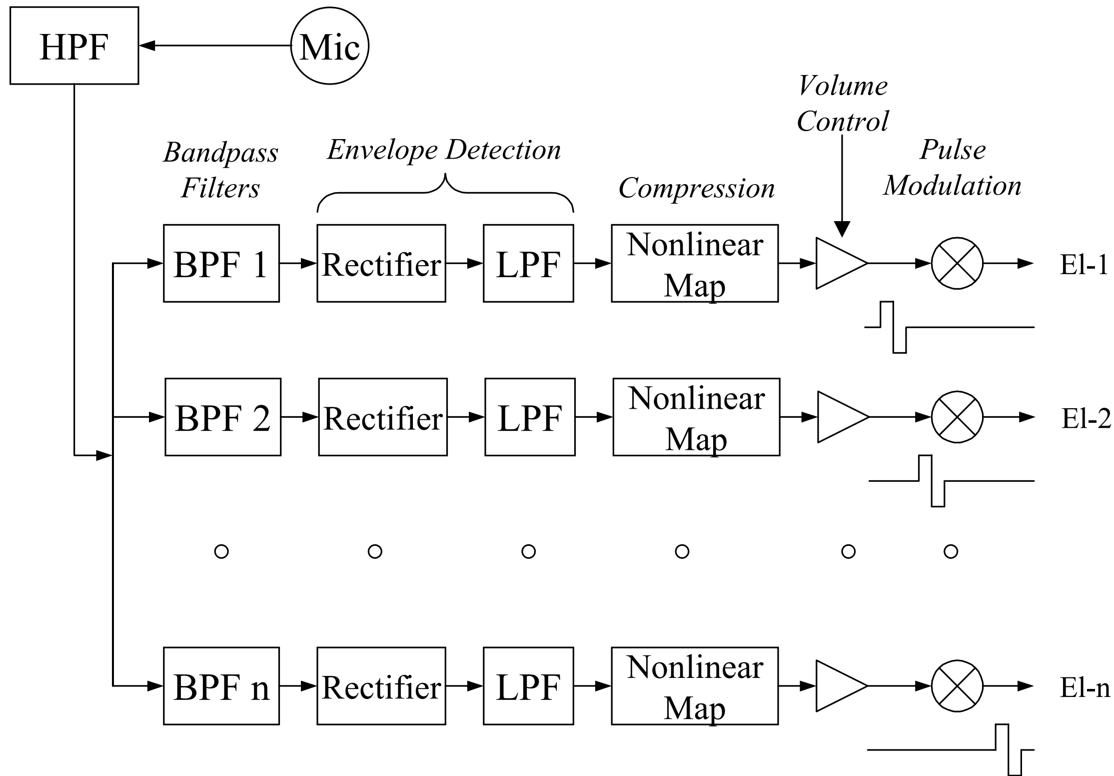
Table 20: CSR Frequency Values

CSR[2:0]	f_{clk} (MHz)	CSR[2:0]	f_{clk} (MHz)	CSR[2:0]	f_{clk} (MHz)	CSR[2:0]	f_{clk} (KHz)
0x0	100	0x4	6.25	0x0	10	0x4	625
0x1	50	0x5	3.13	0x1	5	0x5	312
0x2	25	0x6	1.56	0x2	2.5	0x6	156
0x3	12.5	0x7	0.78	0x3	1.25	0x7	78

Chapter 14. Digital Signal Processor

The DSP included with the WIMS Microcontroller is design to perform the Continuous Interleaved Sampling algorithm (CIS) shown in Fig. 25.

Figure 25: CIS Algorithm



The equations for the IIR Filters are shown in Fig. 26 - Fig. 28.

Figure 26: Highpass Filter Equation

$$\frac{y[n]}{x[n]} = \frac{b_0 + b_1 z^{-1}}{1 + a_0 z^{-1}}$$

Figure 27: Bandpass Filter Equation

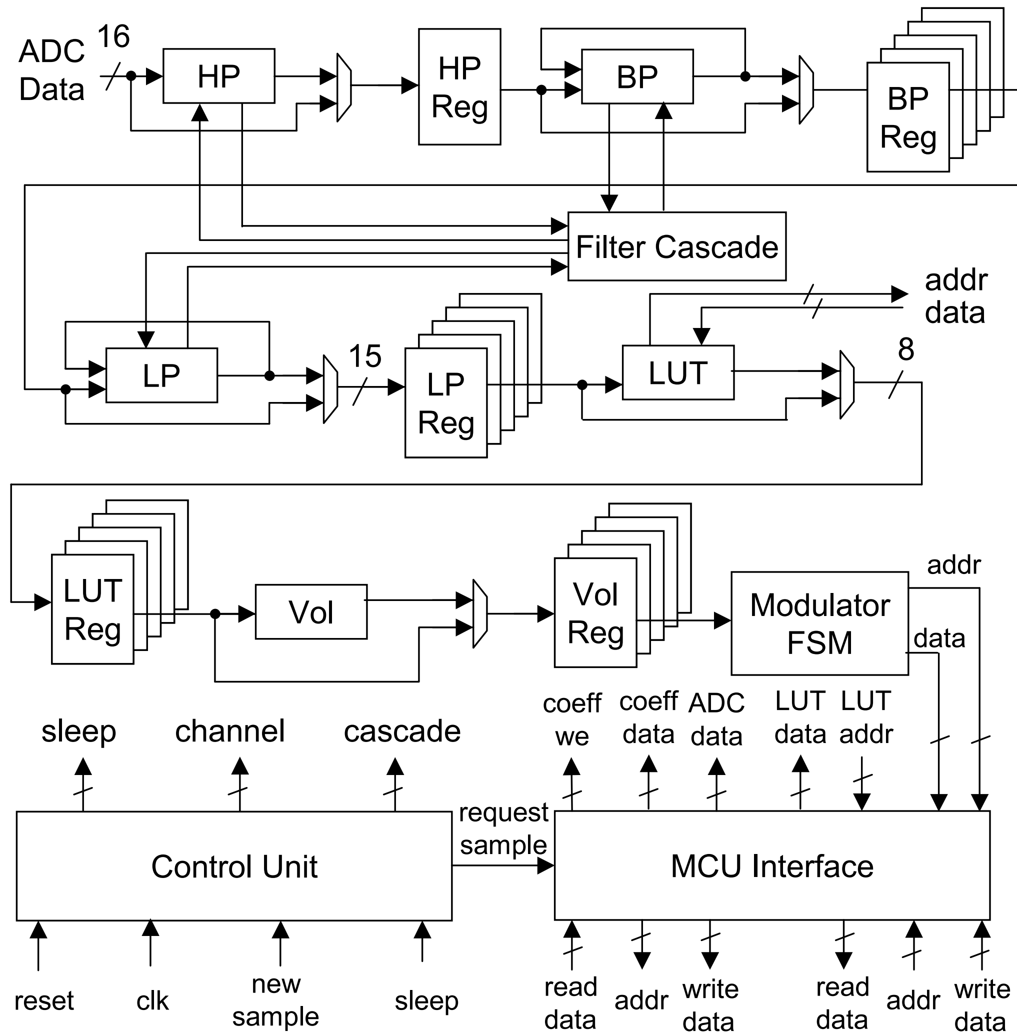
$$\frac{y[n]}{x[n]} = \frac{(b_{00} + b_{01}z^{-1} + b_{00}z^{-2})(b_{10} + b_{11}z^{-1} + b_{10}z^{-2})(b_{20} + b_{21}z^{-1} + b_{20}z^{-2})}{(1 + a_{00}z^{-1} + a_{01}z^{-2})(1 + a_{10}z^{-1} + a_{11}z^{-2})(1 + a_{20}z^{-1} + a_{21}z^{-2})}$$

Figure 28: Lowpass Filter Equation

$$\frac{y[n]}{x[n]} = \frac{(b_{00} + b_{01}z^{-1} + b_{00}z^{-2})(b_{10} + b_{11}z^{-1} + b_{10}z^{-2})}{(1 + a_{00}z^{-1} + a_{01}z^{-2})(1 + a_{10}z^{-1} + a_{11}z^{-2})}$$

The implementation is a dedicated hardware datapath with several programmable features. Fig. 29 shows the implementation chosen. The DSP shares control of the Loop Cache (LUT), SPI0, and SPI1 with the Microcontroller.

Figure 29: CIS Datapath Block Diagram



The volume equations are shown in Fig. 30, where the *THR* is the channel Threshold and *MCL* is the channel Most Comfortable Level.

Figure 30: Volume Equations

$$y = volA \bullet x + volB \qquad volA \varepsilon [0, 1] \qquad volB \varepsilon [THR, MCL]$$

Table 21 shows the memory mapping for the filter coefficients and control registers. Please note that many of the registers used to setup-up the DSP controls are write-only.

Table 21: DSP Memory Map

Address Range		Mnemonic	Description
Start	End		
0x008000	0x0081ff	LUT[7:0]	Look Up Table
0x008200	0x008201	HP_a0[15:0]	Highpass coefficient a0
0x008202	0x008205	HP_b0[15:0], HP_b1[15:0]	Highpass coefficient b0, b1
0x008206	0x008209	BP0_b00[15:0], BP0_b01[15:0]	Channel 0 Bandpass coefficient b00, b01
0x00820a	0x00820e	BP0_a00[15:0], BP0_a01[15:0]	Channel 0 Bandpass coefficient a00, a01
0x00820e	0x008211	BP0_b10[15:0], BP0_b11[15:0]	Channel 0 Bandpass coefficient b10, b11
0x008212	0x008215	BP0_a10[15:0], BP0_a11[15:0]	Channel 0 Bandpass coefficient a10, a11
0x008216	0x008219	BP0_b20[15:0], BP0_b21[15:0]	Channel 0 Bandpass coefficient b20, b21
0x00821a	0x00821d	BP0_a20[15:0], BP0_a21[15:0]	Channel 0 Bandpass coefficient a20, a21
0x00821e	0x008221	BP1_b00[15:0], BP1_b01[15:0]	Channel 1Bandpass coefficient b00, b01
0x008222	0x008225	BP1_a00[15:0], BP1_a01[15:0]	Channel 1 Bandpass coefficient a00, a01
0x008226	0x008229	BP1_b10[15:0], BP1_b11[15:0]	Channel 1 Bandpass coefficient b10, b11
0x00822a	0x00822d	BP1_a10[15:0], BP1_a11[15:0]	Channel 1 Bandpass coefficient a10, a11
0x00822e	0x008231	BP1_b20[15:0], BP1_b21[15:0]	Channel 1 Bandpass coefficient b20, b21
0x008232	0x008235	BP1_a20[15:0], BP1_a21[15:0]	Channel 1 Bandpass coefficient a20, a21
0x008236	0x008239	BP2_b00[15:0], BP2_b01[15:0]	Channel 2 Bandpass coefficient b00, b01
0x00823a	0x00823e	BP2_a00[15:0], BP2_a01[15:0]	Channel 2 Bandpass coefficient a00, a01
0x00823e	0x008241	BP2_b10[15:0], BP2_b11[15:0]	Channel 2 Bandpass coefficient b10, b11
0x008242	0x008245	BP2_a10[15:0], BP2_a11[15:0]	Channel 2 Bandpass coefficient a10, a11
0x008246	0x008249	BP2_b20[15:0], BP2_b21[15:0]	Channel 2 Bandpass coefficient b20, b21
0x00824a	0x00824d	BP2_a20[15:0], BP2_a21[15:0]	Channel 2 Bandpass coefficient a20, a21
0x00824e	0x008251	BP3_b00[15:0], BP3_b01[15:0]	Channel 3 Bandpass coefficient b00, b01
0x008252	0x008255	BP3_a00[15:0], BP3_a01[15:0]	Channel 3 Bandpass coefficient a00, a01
0x008256	0x008259	BP3_b10[15:0], BP3_b11[15:0]	Channel 3 Bandpass coefficient b10, b11
0x00825a	0x00825d	BP3_a10[15:0], BP3_a11[15:0]	Channel 3 Bandpass coefficient a10, a11
0x00825e	0x008261	BP3_b20[15:0], BP3_b21[15:0]	Channel 3 Bandpass coefficient b20, b21
0x008262	0x008265	BP3_a20[15:0], BP3_a21[15:0]	Channel 3 Bandpass coefficient a20, a21
0x008266	0x008269	BP4_b00[15:0], BP4_b01[15:0]	Channel 4 Bandpass coefficient b00, b01
0x00826a	0x00826e	BP4_a00[15:0], BP4_a01[15:0]	Channel 4 Bandpass coefficient a00, a01
0x00826e	0x008271	BP4_b10[15:0], BP4_b11[15:0]	Channel 4 Bandpass coefficient b10, b11
0x008272	0x008275	BP4_a10[15:0], BP4_a11[15:0]	Channel 4 Bandpass coefficient a10, a11
0x008276	0x008279	BP4_b20[15:0], BP4_b21[15:0]	Channel 4 Bandpass coefficient b20, b21
0x00827a	0x00827d	BP4_a20[15:0], BP4_a21[15:0]	Channel 4 Bandpass coefficient a20, a21
0x00827e	0x008281	BP5_b00[15:0], BP5_b01[15:0]	Channel 5 Bandpass coefficient b00, b01
0x008282	0x008285	BP5_a00[15:0], BP5_a01[15:0]	Channel 5 Bandpass coefficient a00, a01
0x008286	0x008289	BP5_b10[15:0], BP5_b11[15:0]	Channel 5 Bandpass coefficient b10, b11
0x00828a	0x00828d	BP5_a10[15:0], BP5_a11[15:0]	Channel 5 Bandpass coefficient a10, a11
0x00828e	0x008291	BP5_b20[15:0], BP5_b21[15:0]	Channel 5 Bandpass coefficient b20, b21
0x008292	0x008295	BP5_a20[15:0], BP5_a21[15:0]	Channel 5 Bandpass coefficient a20, a21
0x008296	0x008299	BP6_b00[15:0], BP6_b01[15:0]	Channel 6 Bandpass coefficient b00, b01
0x00829a	0x00829e	BP6_a00[15:0], BP6_a01[15:0]	Channel 6 Bandpass coefficient a00, a01
0x00829e	0x0082a1	BP6_b10[15:0], BP6_b11[15:0]	Channel 6 Bandpass coefficient b10, b11
0x0082a2	0x0082a5	BP6_a10[15:0], BP6_a11[15:0]	Channel 6 Bandpass coefficient a10, a11
0x0082a6	0x0082a9	BP6_b20[15:0], BP6_b21[15:0]	Channel 6 Bandpass coefficient b20, b21
0x0082aa	0x0082ad	BP6_a20[15:0], BP6_a21[15:0]	Channel 6 Bandpass coefficient a20, a21
0x0082ae	0x0082b1	BP7_b00[15:0], BP7_b01[15:0]	Channel 7 Bandpass coefficient b00, b01
0x0082b2	0x0082b5	BP7_a00[15:0], BP7_a01[15:0]	Channel 7 Bandpass coefficient a00, a01
0x0082b6	0x0082b9	BP7_b10[15:0], BP7_b11[15:0]	Channel 7 Bandpass coefficient b10, b11
0x0082ba	0x0082bd	BP7_a10[15:0], BP7_a11[15:0]	Channel 7 Bandpass coefficient a10, a11
0x0082be	0x0082c1	BP7_b20[15:0], BP7_b21[15:0]	Channel 7 Bandpass coefficient b20, b21

Table 21: DSP Memory Map

Address Range		Mnemonic	Description
Start	End		
0x0082c2	0x0082c5	BP7_a20[15:0], BP7_a21[15:0]	Channel 7 Bandpass coefficient a20, a21
0x0082c6	0x0082c9	BP8_b00[15:0], BP8_b01[15:0]	Channel 8 Bandpass coefficient b00, b01
0x0082ca	0x0082ce	BP8_a00[15:0], BP8_a01[15:0]	Channel 8 Bandpass coefficient a00, a01
0x0082ce	0x0082d1	BP8_b10[15:0], BP8_b11[15:0]	Channel 8 Bandpass coefficient b10, b11
0x0082d2	0x0082d5	BP8_a10[15:0], BP8_a11[15:0]	Channel 8 Bandpass coefficient a10, a11
0x0082d6	0x0082d9	BP8_b20[15:0], BP8_b21[15:0]	Channel 8 Bandpass coefficient b20, b21
0x0082da	0x0082dd	BP8_a20[15:0], BP8_a21[15:0]	Channel 8 Bandpass coefficient a20, a21
0x0082de	0x0082e1	BP9_b00[15:0], BP9_b01[15:0]	Channel 9 Bandpass coefficient b00, b01
0x0082e2	0x0082e5	BP9_a00[15:0], BP9_a01[15:0]	Channel 9 Bandpass coefficient a00, a01
0x0082e6	0x0082e9	BP9_b10[15:0], BP9_b11[15:0]	Channel 9 Bandpass coefficient b10, b11
0x0082ea	0x0082ed	BP9_a10[15:0], BP9_a11[15:0]	Channel 9 Bandpass coefficient a10, a11
0x0082ee	0x0082f1	BP9_b20[15:0], BP9_b21[15:0]	Channel 9 Bandpass coefficient b20, b21
0x0082f2	0x0082f5	BP9_a20[15:0], BP9_a21[15:0]	Channel 9 Bandpass coefficient a20, a21
0x0082f6	0x0082f9	BP10_b00[15:0], BP10_b01[15:0]	Channel 10 Bandpass coefficient b00, b01
0x0082fa	0x0082fd	BP10_a00[15:0], BP10_a01[15:0]	Channel 10 Bandpass coefficient a00, a01
0x0082fe	0x008301	BP10_b10[15:0], BP10_b11[15:0]	Channel 10 Bandpass coefficient b10, b11
0x008302	0x008305	BP10_a10[15:0], BP10_a11[15:0]	Channel 10 Bandpass coefficient a10, a11
0x008306	0x008309	BP10_b20[15:0], BP10_b21[15:0]	Channel 10 Bandpass coefficient b20, b21
0x00830a	0x00830d	BP10_a20[15:0], BP10_a21[15:0]	Channel 10 Bandpass coefficient a20, a21
0x00830e	0x008311	BP11_b00[15:0], BP11_b01[15:0]	Channel 11 Bandpass coefficient b00, b01
0x008312	0x008315	BP11_a00[15:0], BP11_a01[15:0]	Channel 11 Bandpass coefficient a00, a01
0x008316	0x008319	BP11_b10[15:0], BP11_b11[15:0]	Channel 11 Bandpass coefficient b10, b11
0x00831a	0x00831d	BP11_a10[15:0], BP11_a11[15:0]	Channel 11 Bandpass coefficient a10, a11
0x00831e	0x008321	BP11_b20[15:0], BP11_b21[15:0]	Channel 11 Bandpass coefficient b20, b21
0x008322	0x008325	BP11_a20[15:0], BP11_a21[15:0]	Channel 11 Bandpass coefficient a20, a21
0x008326	0x008329	BP12_b00[15:0], BP12_b01[15:0]	Channel 12 Bandpass coefficient b00, b01
0x00832a	0x00832e	BP12_a00[15:0], BP12_a01[15:0]	Channel 12 Bandpass coefficient a00, a01
0x00832e	0x008331	BP12_b10[15:0], BP12_b11[15:0]	Channel 12 Bandpass coefficient b10, b11
0x008332	0x008335	BP12_a10[15:0], BP12_a11[15:0]	Channel 12 Bandpass coefficient a10, a11
0x008336	0x008339	BP12_b20[15:0], BP12_b21[15:0]	Channel 12 Bandpass coefficient b20, b21
0x00833a	0x00833d	BP12_a20[15:0], BP12_a21[15:0]	Channel 12 Bandpass coefficient a20, a21
0x00833e	0x008341	BP13_b00[15:0], BP13_b01[15:0]	Channel 13 Bandpass coefficient b00, b01
0x008342	0x008345	BP13_a00[15:0], BP13_a01[15:0]	Channel 13 Bandpass coefficient a00, a01
0x008346	0x008349	BP13_b10[15:0], BP13_b11[15:0]	Channel 13 Bandpass coefficient b10, b11
0x00834a	0x00834d	BP13_a10[15:0], BP13_a11[15:0]	Channel 13 Bandpass coefficient a10, a11
0x00834e	0x008351	BP13_b20[15:0], BP13_b21[15:0]	Channel 13 Bandpass coefficient b20, b21
0x008352	0x008355	BP13_a20[15:0], BP13_a21[15:0]	Channel 13 Bandpass coefficient a20, a21
0x008356	0x008359	BP14_b00[15:0], BP14_b01[15:0]	Channel 14 Bandpass coefficient b00, b01
0x00835a	0x00835e	BP14_a00[15:0], BP14_a01[15:0]	Channel 14 Bandpass coefficient a00, a01
0x00835e	0x008361	BP14_b10[15:0], BP14_b11[15:0]	Channel 14 Bandpass coefficient b10, b11
0x008362	0x008365	BP14_a10[15:0], BP14_a11[15:0]	Channel 14 Bandpass coefficient a10, a11
0x008366	0x008369	BP14_b20[15:0], BP14_b21[15:0]	Channel 14 Bandpass coefficient b20, b21
0x00836a	0x00836d	BP14_a20[15:0], BP14_a21[15:0]	Channel 14 Bandpass coefficient a20, a21
0x00836e	0x008371	BP15_b00[15:0], BP15_b01[15:0]	Channel 15 Bandpass coefficient b00, b01
0x008372	0x008375	BP15_a00[15:0], BP15_a01[15:0]	Channel 15 Bandpass coefficient a00, a01
0x008376	0x008379	BP15_b10[15:0], BP15_b11[15:0]	Channel 15 Bandpass coefficient b10, b11
0x00837a	0x00837d	BP15_a10[15:0], BP15_a11[15:0]	Channel 15 Bandpass coefficient a10, a11
0x00837e	0x008381	BP15_b20[15:0], BP15_b21[15:0]	Channel 15 Bandpass coefficient b20, b21
0x008382	0x008385	BP15_a20[15:0], BP15_a21[15:0]	Channel 15 Bandpass coefficient a20, a21
0x008386	0x008389	LP_b00[15:0], LP_b01[15:0]	Lowpass coefficient b00, b01
0x00838a	0x00838d	LP_a00[15:0], LP_a01[15:0]	Lowpass coefficient a00, a01
0x00838e	0x008391	LP_b10[15:0], LP_b11[15:0]	Lowpass coefficient b10, b11

Table 21: DSP Memory Map

Address Range		Mnemonic	Description
Start	End		
0x008392	0x008395	LP_a10[15:0], LP_a11[15:0]	Lowpass coefficient a10, a11
0x008396	0x008397	ingain[15:0]	LUT gain multiplier
0x008398	0x008399	A0[7:0], B0[7:0]	Channel 0 MCL, THR
0x00839a	0x00839b	A1[7:0], B1[7:0]	Channel 1 MCL, THR
0x00839c	0x00839d	A2[7:0], B2[7:0]	Channel 2 MCL, THR
0x00839e	0x00839f	A3[7:0], B3[7:0]	Channel 3 MCL, THR
0x0083a0	0x0083a1	A4[7:0], B4[7:0]	Channel 4 MCL, THR
0x0083a2	0x0083a3	A5[7:0], B5[7:0]	Channel 5 MCL, THR
0x0083a4	0x0083a5	A6[7:0], B6[7:0]	Channel 6 MCL, THR
0x0083a6	0x0083a7	A7[7:0], B7[7:0]	Channel 7 MCL, THR
0x0083a8	0x0083a9	A8[7:0], B8[7:0]	Channel 8 MCL, THR
0x0083aa	0x0083ab	A9[7:0], B9[7:0]	Channel 9 MCL, THR
0x0083ac	0x0083ad	A10[7:0], B10[7:0]	Channel 10 MCL, THR
0x0083ae	0x0083af	A11[7:0], B11[7:0]	Channel 11 MCL, THR
0x0083b0	0x0083b1	A12[7:0], B12[7:0]	Channel 12 MCL, THR
0x0083b2	0x0083b3	A13[7:0], B13[7:0]	Channel 13 MCL, THR
0x0083b4	0x0083b5	A14[7:0], B14[7:0]	Channel 14 MCL, THR
0x0083b6	0x0083b7	A15[7:0], B15[7:0]	Channel 15 MCL, THR
0x0083b8	0x0083b9	volA[7:0], volB[7:0]	Volume A, Volume B
0x0083ba	0x0083bb	PW0[7:0], PW1[7:0]	Channel 0, 1 pulse width
0x0083bc	0x0083bd	PW2[7:0], PW3[7:0]	Channel 2, 3 pulse width
0x0083be	0x0083bf	PW4[7:0], PW5[7:0]	Channel 4, 5 pulse width
0x0083c0	0x0083c1	PW6[7:0], PW7[7:0]	Channel 6, 7 pulse width
0x0083c2	0x0083c3	PW8[7:0], PW9[7:0]	Channel 8, 9 pulse width
0x0083c4	0x0083c5	PW10[7:0], PW11[7:0]	Channel 10, 11 pulse width
0x0083c6	0x0083c7	PW12[7:0], PW13[7:0]	Channel 12, 13 pulse width
0x0083c8	0x0083c9	PW14[7:0], PW15[7:0]	Channel 14, 15 pulse width
0x0083ca	0x0083cb	elec0[7:0], elec1[7:0]	Channel 0, 1 electrode address
0x0083cc	0x0083cd	elec2[7:0], elec3[7:0]	Channel 2, 3 electrode address
0x0083ce	0x0083cf	elec4[7:0], elec5[7:0]	Channel 4, 5 electrode address
0x0083d0	0x0083d1	elec6[7:0], elec7[7:0]	Channel 6, 7 electrode address
0x0083d2	0x0083d3	elec8[7:0], elec9[7:0]	Channel 8, 9 electrode address
0x0083d4	0x0083d5	elec10[7:0], elec11[7:0]	Channel 10, 11 electrode address
0x0083d6	0x0083d7	elec12[7:0], elec13[7:0]	Channel 12, 13 electrode address
0x0083d8	0x0083d9	elec14[7:0], elec15[7:0]	Channel 14, 15 electrode address
0x0083da	0x0083db	num_zeros[15:0]	Number of cycles between pulses
0x0083dc	0x0083dd	SPI0_CTRL[15:0]	SPI0 Control reg
0x0083de	0x0083df	SPI1_CTRL[15:0]	SPI1 Control reg
0x0083e0	0x0083e1	SPI1_INT[15:0]	SPI1 Number of Interrupt Cycles
0x0083e2	0x0083e3	DSP_CTRL[15:0]	DSP Control reg
0x0083e4	0x0083e5	dout0[7:0], dout1[7:0]	Channel 0, 1 data out - read only
0x0083e6	0x0083e7	dout2[7:0], dout3[7:0]	Channel 2, 3 data out - read only
0x0083e8	0x0083e9	dout4[7:0], dout5[7:0]	Channel 4, 5 data out - read only
0x0083ea	0x0083eb	dout6[7:0], dout7[7:0]	Channel 6, 7 data out - read only
0x0083ec	0x0083ed	dout8[7:0], dout9[7:0]	Channel 8, 9 data out - read only
0x0083ee	0x0083ef	dout10[7:0], dout11[7:0]	Channel 10, 11 data out - read only
0x0083f0	0x0083f1	dout12[7:0], dout13[7:0]	Channel 12, 13 data out - read only
0x0083f2	0x0083f3	dout14[7:0], dout15[7:0]	Channel 14, 15 data out - read only

Fig. 31 shows the bit breakdown for the DSP control register. This register resets to 0x0001.

Figure 31: DSP Control Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC_out[7:0]								Pulse	Test_addr[2:0]			Cache	SPI1	SPI0	Sleep

Sleep –DSP Sleep (R/W)
- 0 - The DSP is enabled and processing data.
- 1 - The DSP is disabled.

SPI0 –SPI0 Control (R/W)
- 0 - The MCU has control over the SPI0 interface.
- 1 - The DSP has control over the SPI0 interface.

SPI1 –SPI1 Control (R/W)
- 0 - The MCU has control over the SPI1 interface.
- 1 - The DSP has control over the SPI1 interface.

Cache –Loop Cache Control (R/W)
- 0 - The MCU has control over the loop cache interface.
- 1 - The DSP has control over the loop cache interface.

Test_addr[2:0] –Test address (R/W)
- 000 - No test.
- 001 - Highpass filter test.
- 010 - Bandpass filter test.
- 011 - Lowpass filter test.
- 100 - LUT test.
- 101 - Volume test.
- 11x - No test.

Pulse –Pulse polarity (R/W)
- 0 - The positive half of the biphasic pulse comes first.
- 1 - The negative half of the biphasic pulse comes first.

ADC_out[7:0] –ADC out(R/W)

This is the first word sent through SPI1 when requesting a new sample from the ADC hooked up to this port. This is designed to work with the AD7708/7718. This will be written to the ADC control register.

The SPI0 and SPI1 control registers are written to the corresponding SPI port before each transmission. The SPI1_INT is the number of DSP clock cycles to wait before requesting the next ADC sample.

Appendix A. List of Publications

The following publications may useful when looking for additional information on the WIMS Microcontroller.

[1] Michael S. McCorquodale, Fadi H. Gebara, Keith L. Kraver, Eric D. Marsman, Robert M. Senger, Richard B. Brown, “A Top-Down Microsystems Design Methodology and Associated Challenges," *DATE Designers' Forum*, pp. 292-296, March 2003.

[2] Steven M. Martin, Robert M. Senger, Eric D. Marsman, Fadi H. Gebara, Michael S. McCorquodale, Keith L. Kraver, Matthew R. Guthaus, Richard B. Brown, “A Low-Power Microinstrument for Chemical Analysis of Remote Environments,” *11th NASA Symp. on VLSI Design*, pp. 1-4, May 2003.

[3] Robert M. Senger, Eric D. Marsman, Michael S. McCorquodale, Fadi H. Gebara, Keith L. Kraver, Matthew R. Guthaus, Richard B. Brown, “A 16-Bit Mixed-Signal Microsystem with Integrated CMOS-MEMS Clock Reference,” *Design Automation Conference (DAC)*, pp 520-525, June 2003.

[4] Michael S. McCorquodale, Eric D. Marsman, Robert M. Senger, Fadi H. Gebara, Matthew R. Guthaus, Daniel J. Burke, Richard B. Brown, “Microsystem and SoC design with UMIPS,” *IFIP International Conference on VLSI SOC*, pp. 324-329, Dec. 2003.

[5] Rajiv A. Ravindran, Robert M. Senger, Eric D. Marsman, Ganesh S. Dasika, Matthew R. Guthaus, Scott A. Mahlke, Richard B. Brown, “Increasing the Number of Effective Registers in a Low Power Processor Using a Windowed Register File,” *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2003.

[6] Rajiv A. Ravindran, Pracheeti D. Nagarkar, Ganesh S. Dasika, Eric D. Marsman, Robert M. Senger, Scott A. Mahlke, Richard B. Brown, “Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache,” *International Symposium on Code Generation and Optimization (CGO)*, pp. 179-190, March 2005.

[7] Eric D. Marsman, Robert M. Senger, Michael S. McCorquodale, Matthew R. Guthaus, Rajiv A. Ravindran, Ganesh S. Dasika, Scoot A. Mahlke, Richard B. Brown, “A 16-bit Low-Power Microcontroller with Monolithic MEMS-*LC* Clocking,” *Intl. Symp. on Circuits and Systems (ISCAS)*, pp. 624-627, May 2005.

[8] Rajiv A. Ravindran, Robert M. Senger, Eric D. Marsman, Ganesh S. Dasika, Matthew R. Guthaus, Scott A. Mahlke, Richard B. Brown, “Partitioning Variables across Register Windows to Reduce Spill Code in a Low-Power Processor,” *IEEE Transactions on Computers*, Vol. 54, pp. 998-1012, Aug. 2005.

[9] Eric D. Marsman, Robert M. Senger, Gordon A. Carichner, Sundus Kubba, Michael S. McCorquodale, Richard B. Brown, “A DSP architecture for cochlear implants,” *Intl. Symp. on Circuits and Systems (ISCAS)*, pp. 657-660, May 2006.

[10] Robert M. Senger, Eric D. Marsman, Michael S. McCorquodale, Richard B. Brown, “A 16-bit, low-power microsystem with monolithic MEMS-*LC* clocking,” *Asia-South Pacific Design Automation Conference - Student Design Contest (ASP-DAC)*, pp. 94-95, Jan. 2006.

[11] Robert M. Senger, Eric D. Marsman, Gordon A. Carichner, Sundus Kubba, M. McCorquodale, Richard B. Brown, “Low-Latency, HDL-Synthesizable Dynamic Clock Frequency Controller with Hybrid *LC* Clocking,” *Intl. Symp. on Circuits and Systems (ISCAS)*, pp. 775-778, May 2006.

[12] Robert M. Senger, Eric D. Marsman, Spencer S. Kellis, and Richard B. Brown, “Methodology for Instruction Level Power Estimation in Pipelined Microsystems,” *Austin Conf. on Integrated Systems and Circuits (ACISC)*, May 2007.

Appendix B. WIMS Gen-2 Microsystem Physical Specifications

This section gives the test results, pinout, and physical specifications for the WIMS Gen-2 Microsystem taped out in May 2005 through the MOSIS Educational Program in TSMC 0.18μm Mixed Mode/RF process. Fig. 32 is a layout shot of the WIMS Microsystem with all the subsystems labelled. The packaged chips are in a 121 pin PGA package. The die measures 3.023mm x 3.023mm and is 250μm thick and contains 2.28 million transistors. Further statistics are presented in Table 22. The next sections describe the test results, pinout, and other physical specifications.

Figure 32: Gen-2 WIMS Microsystem Layout Shot

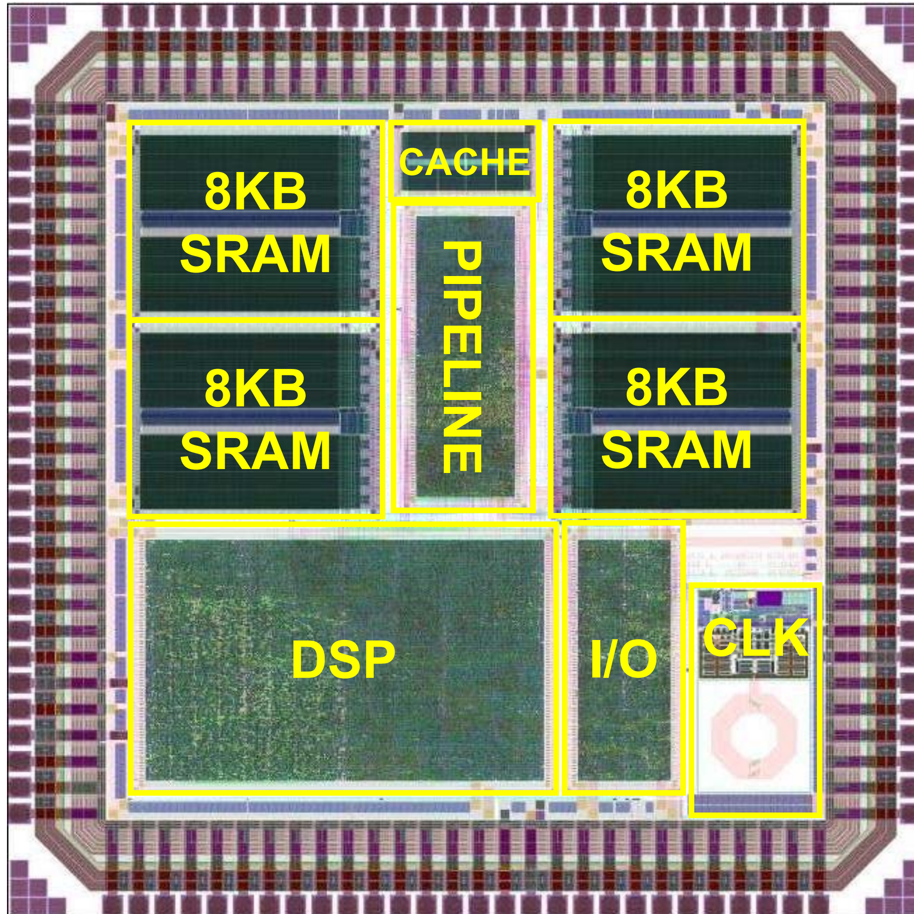


Table 22: Chip Statistics Breakdown

	Pipeline	Peripherals	DSP	CLK	Memory	Total
Area (μm^2)	439,365	366,887	1,274,374	251,940	2,250,148	9,138,529
Transistor Count	102,527	93,048	376,903	634	1,700,126	2,279,617
Decoupling Cap (pF)	153[1]	153[1]	418.2	382.5	724.2	-

1. The Pipeline and Peripherals (I/O) share a power supply, and therefore the 153pF.

Test Results

Fig. 33 is a table of several different operating modes of the complete microsystem at maximum and minimum voltage points. Standby mode consumes 330µW. Maximum power consumption is 46mW. The pads consume about 30mW from a 3.3V supply.

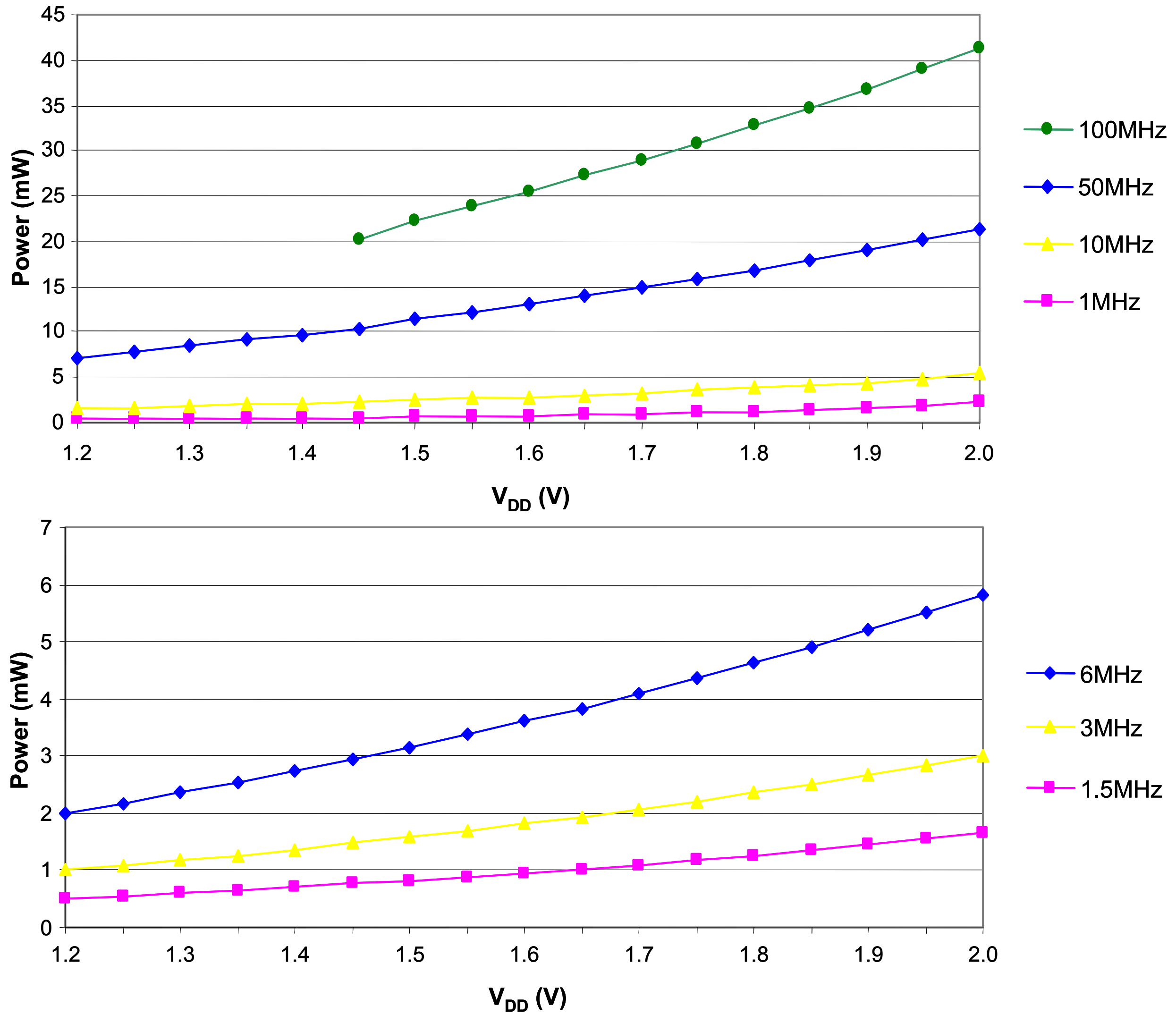
Figure 33: Microsystem measured performance at several operating conditions

Operating Condition / Component	V_{DD} = 1.8V			V_{DD} = 1.2V		
	100MHz	DSP Mode[a]	Standby	1MHz	DSP Mode[a]	Standby
Core (mW)	25.73	1.63	0.17	0.31	0.44	0.06
Memory (mW)	7.83	0.12	0.12	0.04	0.03	0.03
DSP (mW)	2.46[a]	2.46	0.27	1.14[a]	1.14	0.06
Clock (mW)	9.62[b]	0.76[c]	0.76[c]	0.18[c]	0.18[c]	0.18[c]
Total (mW)	45.64	4.97	1.32	1.67	1.79	0.33

a. DSP is operating at 3MHz. Other components operating at speed necessary to support DSP function.
b. LC oscillator is operating, ring oscillator is off.
c. Ring oscillator is operating, LC oscillator is off.

Fig. 34 shows the different operating points for the core and DSP across voltages and frequency ranges.

Figure 34: Power versus VDD scaling across different frequencies for the MCU plus memory (top) and DSP (bottom).



For SPICE parameters and other process parameters, please see the MOSIS web site (http://www.mosis.org) or the UofM Course Tools (https://ctools.umich.edu/) group WIMS Micropower. The fabrication run was T55U-BG (MM_NON-EPI_THK-MTL) in May 2005.

Power per Instruction

The following tables give results of the energy per instruction for the different instructions in the WIMS ISA. For more details on their measurement method, please see Appendix Appendix A..

.

Table 23: Measured energy for each WIMS instruction group fetched from main memory at 100MHz and 1.8V.

Instruction Group	Energy (nJ)	Time (ns)
add-sub	0.43	10
shift	0.35	10
boolean	0.38	10
compare	0.37	10
multiply	4.99	180
divide	4.89	180
copy	0.38	10
bit	1.10	20
load abs	0.94	20
load rel	0.66	10
store abs	0.80	20
store rel	0.55	10

Instruction Group	Energy (nJ)	Time (ns)
win swap	0.33	10
load imm	0.35	10
branch-nt	0.31	10
branch-t	1.03	30
jmp abs	0.97	30
jmp rel	0.72	20
jmp abs sub	1.02	30
jmp rel sub	0.63	20
return	0.67	20
swi	1.01	30
no-op	0.35	10

Table 24: Memory energy correction factor measured at 100MHz and 1.8V.

Memory Access	Ext Mem (nJ)[1]	Loop Cache (nJ)	MMR (nJ)	Boot ROM (nJ)
instruction fetch	-0.11	-0.10	N/A	-0.08
mem bit set/rst[2]	-0.30	-0.30	-0.34	N/A
load absolute[2]	-0.18	-0.18	-0.16	N/A
load relative[2]	-0.19	-0.19	-0.20	N/A
store absolute[2]	-0.07	-0.08	-0.08	N/A
store relative[2]	-0.09	-0.11	-0.10	N/A

1. Excludes memory access energy as this is memory dependent
2. Fetch energy counted separately

Pinout

Fig. 35 shows the bonding diagram for the PGA. For more information on the PGA121M package, see the MOSIS web site (http://www.mosis.org) or the UofM Course Tools (https://ctools.umich.edu/) group WIMS Micropower. Table 25 gives the pin assignments for all of the pins on the chip. The pin numbers are the ones given in Fig. 35 (Bonding Diagram), but can also be used for the bare die by noticing that 2 PGA pins are unused on each side of the chip.

The naming convention for the signals in Table 25 is (*signal)(Source)(Destination)_(W/B)(P/N/A)(in/out)* where *signal* is the net name, *Source* and *Destination* are the source and destination modules, *W* is wire, *B* is bus, *A* is analog, *P* is positive (active high), *N* is negative (active low), and *in*put or *out*put signal.

The Pin Type lists what input or output pad is used for the signal. The physical specifications (loads, drive currents, etc.) are give in the next section.

Figure 35: PGA Bonding Diagram

Table 25: Pin Assignments

Pin	Name	Level (V)	Pin Type	Description
1	-	-	-	No Connection
2	VDD33	3.3	PVDD2DGZ	Pad ring power
3	VSS33	0	PVSS2DGZ	Pad ring ground
4	VSS_mem	0	PVSS1DGZ	Digital memory ground
5	VDD_mem	1.8	PVDD1DGZ	Digital memory power
6	rstPadChip_WNin	0-3.3	PDISDGZ	Global reset
7	extmemtestselPadMMU_WPin	0-3.3	PDISDGZ	External memory test select[1]
8	gpopenPadMMU_WPin	0-3.3	PDISDGZ	GPOP enable[2]
9	extIntPadId_BNin[2]	0-3.3	PDISDGZ	External interrupt - active low
10	extIntPadId_BNin[1]	0-3.3	PDISDGZ	External interrupt - active low
11	extIntPadId_BNin[0]	0-3.3	PDISDGZ	External interrupt - active low
12	extmemwrdataMMUPad_BPout[0]	0-3.3	PDO08CDG	External memory write data[1,2]
13	extmemwrdataMMUPad_BPout[1]	0-3.3	PDO08CDG	External memory write data[1,2]
14	extmemwrdataMMUPad_BPout[2]	0-3.3	PDO08CDG	External memory write data[1,2]
15	extmemwrdataMMUPad_BPout[3]	0-3.3	PDO08CDG	External memory write data[1,2]
16	extmemwrdataMMUPad_BPout[4]	0-3.3	PDO08CDG	External memory write data[1,2]
17	extmemwrdataMMUPad_BPout[5]	0-3.3	PDO08CDG	External memory write data[1,2]
18	extmemwrdataMMUPad_BPout[6]	0-3.3	PDO08CDG	External memory write data[1,2]
19	extmemwrdataMMUPad_BPout[7]	0-3.3	PDO08CDG	External memory write data[1,2]
20	extmemwrdataMMUPad_BPout[8]	0-3.3	PDO08CDG	External memory write data[1,2]
21	extmemwrdataMMUPad_BPout[9]	0-3.3	PDO08CDG	External memory write data[1,2]
22	extmemwrdataMMUPad_BPout[10]	0-3.3	PDO08CDG	External memory write data[1,2]
23	extmemwrdataMMUPad_BPout[11]	0-3.3	PDO08CDG	External memory write data[1,2]
24	extmemwrdataMMUPad_BPout[12]	0-3.3	PDO08CDG	External memory write data[1,2]
25	extmemwrdataMMUPad_BPout[13]	0-3.3	PDO08CDG	External memory write data[1,2]
26	extmemwrdataMMUPad_BPout[14]	0-3.3	PDO08CDG	External memory write data[1,2]
27	extmemwrdataMMUPad_BPout[15]	0-3.3	PDO08CDG	External memory write data[1,2]
28	extmemceMMUPad_WNout	0-3.3	PDO08CDG	External memory chip enable[1]
29	extmemoeMMUPad_WNout	0-3.3	PDO08CDG	External memory output enable[1]
30	-	-	-	No Connection
31	-	-	-	No Connection
32	VSS_mem	0	PVSS1DGZ	Digital memory ground
33	VDD_mem	1.8	PVDD1DGZ	Digital memory power
34	extmemweMMUPad_BNout[0]	0-3.3	PDO08CDG	External memory write enable[1,2]
35	extmemweMMUPad_BNout[1]	0-3.3	PDO08CDG	External memory write enable[1,2]
36	extmemaddrMMUPad_BPout[0]	0-3.3	PDO08CDG	External memory address[1]
37	extmemaddrMMUPad_BPout[1]	0-3.3	PDO08CDG	External memory address[1]
38	extmemaddrMMUPad_BPout[2]	0-3.3	PDO08CDG	External memory address[1]
39	extmemaddrMMUPad_BPout[3]	0-3.3	PDO08CDG	External memory address[1]

Table 25: Pin Assignments

Pin	Name	Level (V)	Pin Type	Description
40	extmemaddrMMUPad_BPout[4]	0-3.3	PDO08CDG	External memory address[1]
41	extmemaddrMMUPad_BPout[5]	0-3.3	PDO08CDG	External memory address[1]
42	extmemaddrMMUPad_BPout[6]	0-3.3	PDO08CDG	External memory address[1]
43	extmemaddrMMUPad_BPout[7]	0-3.3	PDO08CDG	External memory address[1]
44	extmemaddrMMUPad_BPout[8]	0-3.3	PDO08CDG	External memory address[1]
45	extmemaddrMMUPad_BPout[9]	0-3.3	PDO08CDG	External memory address[1]
46	extmemaddrMMUPad_BPout[10]	0-3.3	PDO08CDG	External memory address[1]
47	extmemaddrMMUPad_BPout[11]	0-3.3	PDO08CDG	External memory address[1]
48	extmemaddrMMUPad_BPout[12]	0-3.3	PDO08CDG	External memory address[1]
49	extmemaddrMMUPad_BPout[13]	0-3.3	PDO08CDG	External memory address[1]
50	extmemaddrMMUPad_BPout[14]	0-3.3	PDO08CDG	External memory address[1]
51	extmemaddrMMUPad_BPout[15]	0-3.3	PDO08CDG	External memory address[1]
52	dspenPadCore_WPin	0-3.3	PDISDGZ	DSP enable
53	VDD_dsp	1.8	PVDD1DGZ	Digital DSP power
54	VSS_dsp	0	PVSS1DGZ	Digital DSP ground
55	VDD_dsp	1.8	PVDD1DGZ	Digital DSP power
56	VSS_dsp	0	PVSS1DGZ	Digital DSP ground
57	VDD33	3.3	PVDD2DGZ	Pad ring power
58	VSS33	0	PVSS2DGZ	Pad ring ground
59	VDD33	3.3	PVDD2DGZ	Pad ring power
60	-	-	-	No Connection
61	-	-	-	No Connection
62	dclkSPI2Pad_WNout	0-3.3	PDO08CDG	Data clock SPI2
63	nioeSPI2Pad_WNout	0-3.3	PDO08CDG	I/O enable SPI2
64	doutSPI2Pad_WPout	0-3.3	PDO08CDG	Data out SPI2
65	nackPadSPI2_WPin	0-3.3	PDISDGZ	Acknowledge SPI2
66	dinPadSPI2_WPin	0-3.3	PDISDGZ	Data in SPI2
67	dclkSPI1Pad_WNout	0-3.3	PDO08CDG	Data clock SPI1
68	nioeSPI1Pad_WNout	0-3.3	PDO08CDG	I/O enable SPI1
69	doutSPI1Pad_WPout	0-3.3	PDO08CDG	Data out SPI1
70	nackPadSPI1_WPin	0-3.3	PDISDGZ	Acknowledge SPI1
71	dinPadSPI1_WPin	0-3.3	PDISDGZ	Data in SPI1
72	dclkSPI0Pad_WNout	0-3.3	PDO08CDG	Data clock SPI0
73	nioeSPI0Pad_WNout	0-3.3	PDO08CDG	I/O enable SPI0
74	doutSPI0Pad_WPout	0-3.3	PDO08CDG	Data out SPI0
75	nackPadSPI0_WPin	0-3.3	PDISDGZ	Acknowledge SPI0
76	dinPadSPI0_WPin	0-3.3	PDISDGZ	Data in SPI0
77	VDD_core	1.8	PVDD1DGZ	Digital core power
78	VSS_core	0	PVSS1DGZ	Digital core ground
79	extclkPadCK_WPin	0-3.3	PDISDGZ	External clock input

Table 25: Pin Assignments

Pin	Name	Level (V)	Pin Type	Description
80	extclkselPadCK_WPin	0-3.3	PDISDGZ	Selects off chip clock source if set
81	clkpadenPad_WPin	0-3.3	PDISDGZ	Digital core clock output enable
82	TxTIPad_WPout	0-3.3	PDO08CDG	Transmit pin Test Interface
83	RxPadTI_WPin	0-3.3	PDISDGZ	Receive pin Test Interface
84	clksyncTIPad_WPout	0-3.3	PDO08CDG	Synchronization clock out Test Interface
85	clksyncPadTI_WPin	0-3.3	PDISDGZ	Synchronization clock in Test Interface
86	TxUsartPad_WPout	0-3.3	PDO08CDG	Transmit pin USART
87	RxPadUsart_WPin	0-3.3	PDISDGZ	Receive pin USART
88	clksyncUsartPad_WPout	0-3.3	PDO08CDG	Synchronization clock out USART
89	clksyncPadUsart_WPin	0-3.3	PDISDGZ	Synchronization clock in USART
90	-	-	-	No Connection
91	-	-	-	No Connection
92	timeroutTimer0Pad_WPout	0-3.3	PDO08CDG	Timer out Timer0
93	timeroutTimer1Pad_WPout	0-3.3	PDO08CDG	Timer out Timer1
94	timeroutTimer2Pad_WPout	0-3.3	PDO08CDG	Timer out Timer2
95	timerinPadTimer0_WPin	0-3.3	PDISDGZ	Timer in Timer0
96	timerinPadTimer1_WPin	0-3.3	PDISDGZ	Timer in Timer1
97	timerinPadTimer2_WPin	0-3.3	PDISDGZ	Timer in Timer2
98	VSS_clk	0	PVSS1DGZ	MEMS clock generator ground
99	VDD_clk	1.8	PVDD1DGZ	MEMS clock generator power
100	calenPadCK_WPin	0-3.3	PDISDGZ	Calibration mode enable
101	clkCKPad_WPout	0-3.3	PDO16CDG	Digital core clock output
102	extmemrddataPadMMU_BPin[0]	0-3.3	PDISDGZ	External memory read data (serclk)[3]
103	extmemrddataPadMMU_BPin[1]	0-3.3	PDISDGZ	External memory read data (serload)[3]
104	extmemrddataPadMMU_BPin[2]	0-3.3	PDISDGZ	External memory read data (serdata)[3]
105	VDD_core	1.8	PVDD1DGZ	Digital core power
106	VSS_core	0	PVSS1DGZ	Digital core ground
107	extmemrddataPadMMU_BPin[3]	0-3.3	PDISDGZ	External memory read data
108	extmemrddataPadMMU_BPin[4]	0-3.3	PDISDGZ	External memory read data
109	extmemrddataPadMMU_BPin[5]	0-3.3	PDISDGZ	External memory read data
110	extmemrddataPadMMU_BPin[6]	0-3.3	PDISDGZ	External memory read data
111	extmemrddataPadMMU_BPin[7]	0-3.3	PDISDGZ	External memory read data
112	extmemrddataPadMMU_BPin[8]	0-3.3	PDISDGZ	External memory read data
113	extmemrddataPadMMU_BPin[9]	0-3.3	PDISDGZ	External memory read data
114	extmemrddataPadMMU_BPin[10]	0-3.3	PDISDGZ	External memory read data
115	extmemrddataPadMMU_BPin[11]	0-3.3	PDISDGZ	External memory read data
116	extmemrddataPadMMU_BPin[12]	0-3.3	PDISDGZ	External memory read data
117	extmemrddataPadMMU_BPin[13]	0-3.3	PDISDGZ	External memory read data
118	extmemrddataPadMMU_BPin[14]	0-3.3	PDISDGZ	External memory read data
119	extmemrddataPadMMU_BPin[15]	0-3.3	PDISDGZ	External memory read data
120	-	-	-	No Connection

1. extmemtestsel is used only for test mode. If set high, extmemaddr[15:0] outputs the PC[15:0], extmemwe[1:0] outputs PC[17:16], extmemce outputs PC[18], extmemoe outputs PC[19], and extmemwrdata[15:0] outputs EX stage memwr-data[15:0]. If set low, the external memory output buses operate normally.
2. gpopen controls the GPOP (general purpose output port). If set, extmemwrdata[15:0] outputs the GPOP and extmemwe[1:0] is disabled, otherwise extmemwrdata[15:0] operates normally.
3. These signals are also used in calibration mode for the Copernicus clock. See Appendix Appendix C. for more information.

Physical Specifications

There are several types of input and output pads that are used. This section gives a physical specification for each pad type. This information comes from TSMC through Artisan Components Inc. Further details can be obtained through their web sites.

Table 26: Power Pad Characteristics

Cell	Max. Current (mA)	Pin Load (pF)
PVDD1DGZ	29	5.551
PVDD2DGZ	32	5.059
PVSS1DGZ	29	2.958
PVSS2DGZ	136	5.059

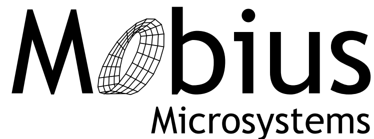
Table 27: I/O Pad Characteristics

Cell	Power (μW/MHz)	Drive Strength (mA)	Pin Load (pF)	Rise Delay (ns)	Fall Delay (ns)
PDISDGZ	4.33	N/A	3.341	0.628+0.150 C_{ld}[1]	0.852+0.158 C_{ld}[1]
PDO08CDG	106.32	8.00	3.219	1.219+0.045 C_{ld}[1]	1.186+0.047 C_{ld}[1]
PDO16CDG	146.29	16.00	2.796	1.618+0.024 C_{ld}[1]	1.493+0.026 C_{ld}[1]

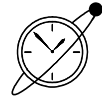
1. C_{ld} is in pF

The bonding pads are 81μm by 71μm with a 86μm center-to-center pitch.

Appendix C. Copernicus On-chip Clock Generator



Copernicus Clock Macro
For UM WIMS Processor



RELEASE 1.2

Design Specification

Features

- 200 MHz LC clock with real-time temperature compensation
- 20 MHz low-power ring clock
- LC clock shutdown for low-power
- Glitch-free switching between LC and ring clock
- 32-bit control interface for calibration of LC/ring frequencies and temperature compensation

Block Diagram

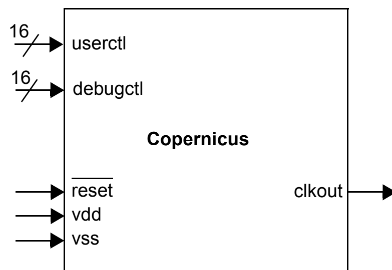


Figure 1. Block diagram of Copernicus Macro

Signal Descriptions

Table 1: Copernicus Signal Descriptions

Name	voltage levels	Description
vdd	1.8V	macro power (input)
vss	0V	macro ground (input)
$\overline{\text{reset}}$	V_H=1.8V, V_L=0V	macro reset, active low (input)
userctl[15:0]	V_H=1.8V, V_L=0V	user control and calibration (input)
debugctl[15:0]	V_H=1.8V, V_L=0V	miscellaneous debug control (input)
clkout	V_H=1.8V, V_L=0V	200MHz/1MHz clock (output)

vdd: Copernicus 1.8V power supply

This will be driven by a dedicated supply pin of the WIMS processor, allowing for better noise isolation and a simple means of measuring macro current.

vss: Copernicus ground

This will be driven by a dedicated supply pin of the WIMS processor, allowing for better noise isolation and a simple means of measuring macro current.

$\overline{\text{reset}}$: asynchronous reset

This active low input initializes the arbiter to allow the LC clock to drive clkout during and upon exiting reset. In turn, the ring clock is barred from driving clkout during and upon exiting reset. The integrator must therefore ensure, via initialization of the userctl inputs, the LC clock is enabled and selected upon reset.

userctl[7:0]: LCCal - LC Calibration

This 8-bit value allows for calibration of the LC clock frequency. Process variations will lead to die-to-die differences in the center frequency. The correct value for LCCal is provided by the external Automatic Frequency Calibration Macro (AFCM). See the AFCM section for details on interfacing with this macro.

The Copernicus clock macro is designed to produce a 200MHz clock output with LCCal = $80, where "$" is used to signify hex radix. Increasing values of LCCal will reduce the frequency whereas decreasing values will result in higher frequencies. A value of $FF will produce a clock with frequency 0.92X the resulting center frequency out of the fab. $00 will give a frequency of 1.08X the frequency out of the fab. LCCal bits are weighted in a binary manner as shown in Table 2.

For maximum flexibility in calibration, it is recommended that LCCal be initialized to $FF upon reset, producing the lowest possible frequency.

Table 2: LCCal Weighting

bit	weight
7	-8%
6	-4%
5	-2%
4	-1%
3	-0.5%
2	-0.25%
1	-0.125%
0	-0.0625

userctl[8]: LCSel - LC Clock Select

This bit selects which clock source drives the clkout output as described in Table 3. This bit will be reset to 1.

Table 3: LCSel Description

value	description
0	clkout = ring oscillator output
1	clkout = LC oscillator output

userctl[9]: LCEn - LC Clock Enable

This bit allows the user to shut down the LC clock for low-power operation as described in Table 4. This bit should be reset to 1.

Table 4: LCEn Description

value	description
0	LC clock disabled and producing no output
1	LC clock running and producing 200MHz

userctl[10]: RingEn - Ring Clock Enable

This bit allows the user to shut down the ring clock for low-power operation as described in Table 5. This bit should be reset to 0.

Table 5: RingEn Description

value	description
0	Ring clock disabled and producing no output
1	Ring clock running and producing 1MHz output

userctl[11]: BiasEn - Bias Enable

This bit allows the user to shut down the bias circuit for low-power operation as described in Table 6. If BiasEn is de-asserted, neither LC nor ring clocks will produce an output. This bit should be reset to 1.

Table 6: BiasEn Description

value	description
0	Bias powered down. Neither clock source can operate.
1	Bias powered up allowing either clock source to operate (if enabled)

userctl[13:12]: RingCal - Ring Calibration

These two bits allow for coarse control of ring oscillator frequency. $3 gives the lowest ring clock frequency while $0 give the highest. Integrator may determine reset value.

userctl[15:14]: TCRes - Resistor TC Calibration

These two bits allow for calibration of resistor bank temperature coefficient. This is a process-dependent value and will typically not change once the correct value is determined in initial silicon evaluation. These bits should be reset to 0.

debugctl[15:0]: Miscellaneous Debug Control

These bits are for debug of the initial silicon. These bits should be reset to $02A8.

clkout: Copernicus Output Clock

This output is driven by either the LC oscillator output or the ring oscillator output depending on the state of LCSel. An arbiter is included in the Copernicus macro to allow for low-latency, glitch-free switching between the two clock sources.

External AFCM Interface

This section describes the interface to the external Automatic Frequency Calibration Macro (AFCM). The AFCM is implemented in an off-chip FPGA which supports multiples means of interfacing to chips containing the Copernicus macro.

The external AFCM compares a reference clock to an output of the Copernicus clock. If the Copernicus clock is found to be running faster (slower) than the reference, the AFCM will generate a new LCCal value for Copernicus to decrease (increase) the frequency of the LC oscillator. The AFCM then compares the reference clock to the new Copernicus clock frequency once

again. This process is repeated until the best value of LCCal is found.

Figure 2 shows a block diagram of the external AFCM. Both the reference clock and Copernicus clock inputs are 50MHz, *not* 200MHz clocks. Note that this is a preliminary diagram and can be changed per requirements.

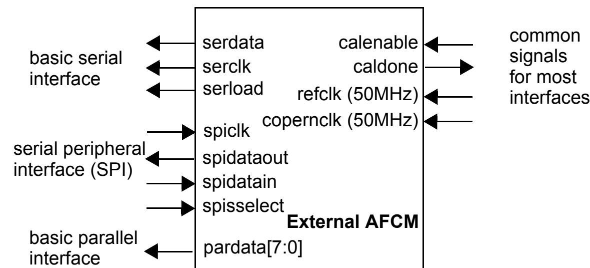


Figure 2. Block diagram of external AFCM

The external AFCM supports various standard interfaces but only two will be relied on for the WIMS processor. The first and most direct method utilizes the basic serial interface of the external AFCM. The SPI interface will serve as a backup to the basic serial interface and will also be used for demonstration purposes.

In addition to calibration, the AFCM will also serve as a direct means of controlling all inputs to the Copernicus macro for the purpose of initial silicon evaluation. In particular, the registers driving debugctl[15:0] and userctl[15:0] will be under direct external control when the basic serial interface is used.

Basic Serial Interface

This interface requires one dedicated pin, calenable, and three dual-purpose pins. The three dual-purpose pins can be any pins that do not serve a critical purpose when the calibration routine is running. For example, GPIO pins would be a good choice here but reset or external clock signals would not be a good choice. When calenable is asserted these pins would drive serdata, serclk and serload to the register logic driving the Copernicus control inputs. Figure 3 shows a diagram of the registers and associated logic for driving userctl[15:0] and debugctl[15:0]. When calenable is asserted, the two 16-bit registers are configured as one 32-bit shift register with input data coming from serdata and the shift clock coming from serclk. Note that no serial output data is required. After the processor boot completes, a 50MHz clock will be driven to an external pin. This frequency can be modified (e.g. to 25MHz) by code running on the core processor.

When calenable is not asserted, the registers are loaded using the normal, system-mode clock and databus (called periphdb and system clock in the diagram).

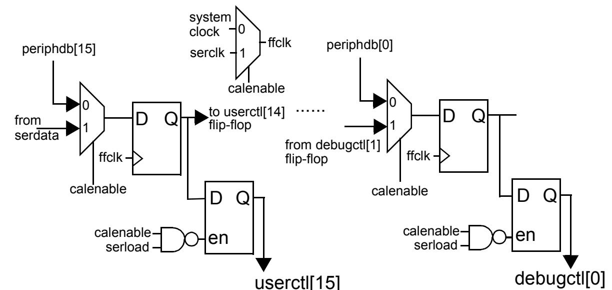


Figure 3. Basic Serial Interface Diagram

A latch controlled by !(calenable*serload) is shown in Figure 3. This latch is included to avoid changing the macro inputs when shifting new values in during calibration mode. If latches are to be avoided, these can be replaced with flip-flops loading on ffclk with a feedback mux that holds data steady when calenable and serload are asserted simultaneously.

Figure 4 shows example waveforms for two 32-bit transfers during calibration or debug.

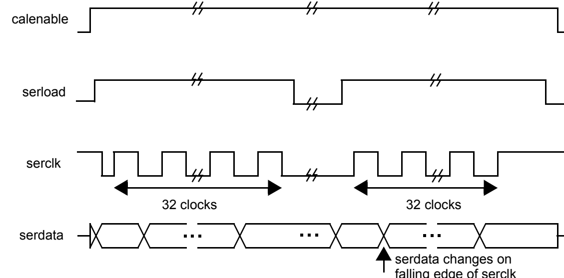


Figure 4. Basic Serial Interface Timing

Serial Peripheral Interface (SPI)

The WIMS SPI can be used as a backup interface for calibration. In addition, Mobius will use this interface as part of a demonstration vehicle for the external AFCM.

Figure 5 shows the pin correspondence between external AFCM and WIMS SPI port.

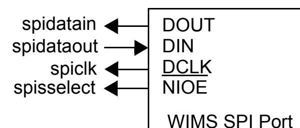


Figure 5. Serial Peripheral Interface

While the external AFCM will perform the actual frequency calibration, code running in the WIMS processor will need to be developed to perform the transfers between the external AFCM and the

WIMS register driving userctl. Figure 6 shows an example flow diagram of this code.

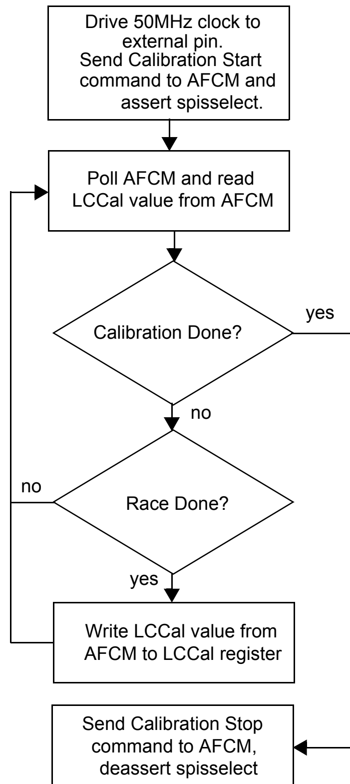


Figure 6. SPI Calibration Flow Diagram

The external AFCM itself is under development and the command syntax is not yet available.

BIBLIOGRAPHY

[1] CDC NHIS, http://www.cdc.gov/nchs/nhis.htm, 2005

[2] John K. Niparko, Karen I. Kirk, Nancy K. Mellon, Amy McConkey Robbins, Debara L. Tucci, and Blake S. Wilson, Eds., *Cochlear Implants: Principles & Practices*, Lippincott Williams & Wilkins, Philadelphia, 2000.

[3] NIDCD, http://www.nidcd.nih.gov/health/hearing/coch.asp, 2010

[4] Kensall D. Wise and Khalil Najafi, "Fully-implantable auditory prostheses: restoring hearing to the profoundly deaf," International Electron Devices Meeting, pps. 499 - 502, Dec. 8-11, 2002.

[5] K. Wise, K. Najafi, R. Sacks, E. Zellers, "A Wireless Integrated Microsystem For Environmental Monitoring", ISSCC Dig. Tech. Papers, pp. 434–435, Feb. 2004.

[6] R. Senger, E. Marsman, M. McCorquodale, F. Gebara, K. Kraver, M. Guthaus, R. Brown, "A 16-Bit Mixed-Signal Microsystem with Integrated CMOS-MEMS Clock Reference," *Design Automation Conf.*, pp. 520–525, June 2003.

[7] R. Senger, "Micropower Digital Design Techniques for the Nanometer Realm," Ph.D. Dissertation, University of Michigan, 2008.

[8] S. Martin, R. Senger, E. Marsman, M. McCorquodale, F. Gebara, K. Kraver, M. Guthaus, R. Brown "A Low-Power Microinstrument for Chemical Analysis of Remote Environments," *11th NASA Symp. on VLSI Design*, pp. 1–4, May 2003.

[9] K. Wise, et al., "WIMS ERC Annual Report 2009," University of Michigan, 2009.

[10] R. Mercado-Reyes, "Development of a Smoothing Process for the Wireless Integrated MicroSystems Microcontroller on the Micro Gas Chromatograph," University of Michigan REU Final Report, Aug. 2004.

[11] E. Marsman, "A Low-Power Digital Signal Processor Architecture for a Cochlear Implant System-on-a-Chip," University of Michigan Thesis Proposal, Oct. 2003.

[12] William F. House, "Cochlear Implants: My Perspective", http://www.allhear.com/monographs/m-95-htm.html, 1997.

[13] Blake S. Wilson, "The RTI's Perspective on Bilateral Cochlear Implantation," Wullstein Symposium - 3rd International Conference on Bilateral Cochlear Implants and Binaural Signal Processing, Dec. 12-15, 2002.

[14] Alec N. Salt, "Inner Ear Anatomy," Washington University Cochlear Fluids Lab, http://oto.wustl.edu/cochlea/intro1.htm.

[15] Cochlea: cross section. [Art]. In Encyclopedia Britannica. Retrieved from http://www.britannica.com/EBchecked/media/534/A-cross-section-through-one-of-the-turns-of-the, 1997.

[16] Pamela T. Bhatti, Ben Y. Arcand, Jinbai Wang, N.V. Butala, Craig R. Friedrich, and Kensall D. Wise, “A high-density electrode array for a cochlear prosthesis,” *12th Intl. Conf. on Solid-State Sensors, Actuators, and Microsystems (TRANSDUCERS)*, Vol. 2, pps. 1750-1753, Jun 8-12, 2003.

[17] B. S. Wilson, et al. “Importance of patient and processor variables in determining outcomes with cochlear implants,” *J. Speech and Hearing Research*, vol. 36, pp. 373–379, 1993.

[18] Med-El, “Products,” http://www.medel.com/professionals/int/medel_prof_blank.html, 2003.

[19] Advanced Bionics Corporation, “Products,” http://www.cochlearimplant.com/products/products.html, 2003.

[20] Cochlear Corporation, “Nucleus Product Data,” http://www.cochlearamericas.com/Professional/141.asp, 2003.

[21] D. K. Eddington, “Speech Discrimination in Deaf Subjects with Cochlear Implants,” Journal of the Acoustical Society of America 68, 885-891, 1980.

[22] P. Loizou, "Signal-processing techniques for cochlear implants". IEEE Engineering in Medicine and Biology Magazine, 18(3), 34-46, 1999.

[23] P. Loizou, "Mimicking the Human Ear: An Overview of Signal Processing Techniques Used for Cochlear Implants". IEEE Signal Processing Magazine, 15(5), 101-130, 1998.

[24] Jill B. Firszt, “HiResolution Sound Processing,” Whitepaper, http://www.advancedbionics.com/content/dam/ab/Global/en_ce/documents/libraries/Professional Library/AB Technical Reports/Sound Processing/HiResolution_Sound_Processing.pdf, 2006.

[25] M. Hames, “DSP”. Texas Instruments Investor Presentation. Retrieved from: http://www.bloobble.com/broadband-presentations/presentations?itemid=692, 2009.

[26] Texas Instruments, “TMS320C5402 Fixed-Point Digital Signal Processor Reference Manual,” 2000.

[27] Texas Instruments. TMS320C54X DSP Reference Set, http://www-s.ti.com/sc/psheets/spru131g/spru131g.pdf, Mar. 2001.

[28] Motorola “Technical Data Advance Information DSP56309/D 24-Bit Digital Signal-Processor,” 2003.

[29] J.A. Zakis, H.J. McDermott, and M. Fisher, “A High Performance Digital Hearing Aid for Advanced Sound Processing Research,” IEEE Engineering in Medicine andBiology Society Conference, 2001.

[30] Motorola. CPU12 Reference Manual, http://e-www.motorola.com/brdata/PDFDB/docs/CPU12RM.pdf, June 2003.

[31] Karolien De Baere, et al., “Development of a dedicated parallel DSP core system for hearing implant applications,” Phillips DSP Conference, 2003.

[32] J. Georgiou and C. Toumazou, “A Micropower Cochlear Prosthesis System,” 14th International Conference on Digital Signal Processing, 2002.

[33] J. Georgiou, C. Toumazou, "A 126-μW cochlear chip for a totally implantable system," Solid-State Circuits, IEEE Journal of , vol.40, no.2, pp. 430- 443, Feb. 2005

[34] T.K.-T. Lu, et al., “A micropower analog VLSI processing channel for bionic ears and speech-recognition front ends,” Proceedings of the 2003 International Symposium on Circuits and Systems, Volume 5 , May 25-28, 41 -44, 2003.

[35] R. Sarpeshkar, et al., “An analog bionic ear processor with zero-crossing detection,” *ISSCC Dig. Tech. Papers*, pp. 78–79, Feb. 2005.

[36] H. H. Hellmich, A. T. Erdogan, and T. Arslan, “Re-Usable Low Power DSP IP embedded in an ARM based SoC Architecture,” IEE Colloquium on Intellectual Property, Edinburgh, UK, July 2000.

[37] O. Poroy and P. Loizou, “Development of a speech processor for laboratory experiments with cochlear implant patients,” *Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 3626–3629, June 2000.

[38] E. Marsman, R. Senger, M. McCorquodale, R. Brown, “A 16-Bit, Low-Power Microcontroller with Monolithic MEMS-LC Clocking,” *ISCAS*, pp 624–627, May 2005.

[39] R. Ravindran, R. Senger, E. Marsman, G. Dasika, M. Guthaus, S. Mahlke, R. Brown, “Increasing the Number of Effective Registers in a Low Power Processor Using a Windowed Register File,” International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2003.

[40] S. Kellis, et al., “Energy Profile of a Microcontroller for Neural Prosthetic Application,” ISCAS, pp 3841-3844, Aug, 2010.

[41] S. Kellis, P. House, E. Marsman, R. Senger, N. Gaskin., K. Thomson, B. Greger, R. Brown, "An Embedded System for Neural Prosthetics," IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC), Florianópolis , Brazil, 2009.

[42] Trimaran. An infrastructure for research in ILP, http://www.trimaran.org, 2000.

[43] R. Ravindran, R. Senger, E. Marsman, G. Dasika, M. Guthaus, S. Mahlke, R. Brown, “Partitioning Variables across Register Windows to Reduce Spill Code in a Low-Power Processor,” *Trans. on Computers*, Vol. 54, pp. 998–1012, Aug. 2005.

[44] R. Ravindran, P. Nagarkar, G. Dasika, E. Marsman, R. Senger, S. Mahlke, R. Brown, “Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache,” *Intl. Symp. on Code Generation and Optimization*, pp. 179–190, March 2005.

[45] N. Bellas, et al., “Performance Improvements in Microprocessor Design Using a Loop Cache,” *International Conference on Computer Design*, 1999.

[46] B. Redd, S. Kellis, N. Gaskin, R. Brown, "Scratchpad memories in the context of process scaling," 54th Midwest Symposium on Circuits and Systems, 2011.

[47] S. Steinke, et al., "Assigning program and data objects to scratchpad for energy reduction," DATE Conf. and Exhibition, Paris, France, 2002, pp. 409-415.

[48] A. Mason and J. Zhou, “Communications Buses and Protocols for Sensor Networks,” Sensors, Vol. 2, 2002.

[49] E. Marsman, R. Senger, G. Carichner, S. Kubba, M. McCorquodale, R. Brown, "A DSP architecture for cochlear implants," *ISCAS*, pp. 657–660, May 2006.

[50] Analog Devices. AD7708 Datasheet, http://www.analog.com/en/analog-to-digital-converters/ad-converters/ad7708/products/product.html, 2001.

[51] Analog Devices. AD7888 Datasheet, http://www.analog.com/en/analog-to-digital-converters/ad-converters/ad7888/products/product.html, 2004.

[52] S. Naffziger, B. Stackhouse, and T. Grutkowski, "The implementation of a 2-core multi-threaded Itanium-family processor," ISSCC Dig. Tech. Papers, pp. 182-183, Feb. 2005.

[53] K. Nowka, et al., "A 32-bit PowerPC system-on-a-chip with support for dynamic voltage scaling and dynamic frequency scaling," J. Solid State Circuits, vol. 37, pp. 1441-1447, Nov. 2002.

[54] W. R. Dasssch, C. H. Lim, and G. Cai, "Design of VLSI CMOS circuits under thermal constraint," IEEE Trans. on Circuits and Systems, vol. 49, no. 8, pp 589-593, Aug. 2002.

[55] S. Geissler, et al., "A low-power RISC microprocessor using dual PLLs in a 0.13mm SOI technology with copper interconnect and lowk BEOL dielectric," ISSCC Dig. Tech. Papers, pp.148-149, Feb. 2002.

[56] S. Akui, et al., "Dynamic voltage and frequency management for a low-power embedded microprocessor," ISSCC Dig. Tech. Papers, pp. 64-65, Feb. 2004.

[57] R. Senger, E. Marsman, G. Carichner, S. Kubba, M. McCorquodale, R. Brown, "Low-latency, HDL-synthesizable dynamic clock frequency controller with hybrid LC clocking," *Intl. Symp. on Circuits and Systems (ISCAS)*, pp. 775–778, May 2006.

[58] W.J. Dally and J.W. Poulton, Digital Systems Engineering, 1st Ed., Cambridge University Press, 2000.

[59] K. Yamamoto, M. Fujishima, "4.3GHz 44mW CMOS frequency divider," ISSCC Dig. Tech. Papers, pp. 104-105, Feb. 2004.

[60] M. McCorquodale, M. Ding, and R. Brown, "Top-down and bottomup approaches to stable clock synthesis", Intl. Conf. on Electronic Circuits and Systems, pp. 575-578, Dec. 2003.

[61] R. Mahmud, "Techniques to make clock switching glitch free," [Online]. Available: http://www.eetimes.com/.

[62] European Union 4th Framework, ESPRIT Workprogramme, http://www.cordis.lu/esprit/src/wp96.htm, Sept. 1996.

[63] M. McCorquodale, "Monolithic and Top-Down Clock Synthesis with Micromachined RF Reference," Ph.D. Dissertation, University of Michigan, 2004.

[64] D. J. Young, V. Malba, J. J. Ou, A. F. Bernhardt, and B. E. Boser, "A low-noise RF voltage-controlled oscillator using on-chip high-Q threedimensional coil inductor and micromachined variable capacitor," in Proc. Solid-State Sensor and Actuator Workshop, 1998, pp. 128-131.

[65] S. Lee, G. J. Nam, J. Chae, H. Kim, and A. J. Drake, "Two-dimensional position detection system with MEMS accelerometer for mouse applications," in Proc. Design Automation Conf., 2001, pp. 852-857.

[66] G. M. Rebeiz and J. B. Muldavin, "RF MEMS switches and switch circuits," Microwave Magazine, vol. 2, no. 4, Dec. 2001, pp. 59-71.

[67] K. Kundert, et al., "Design of mixed-signal systems-on-a-chip," IEEE Trans. on CAD, vol. 19, no. 12, Dec. 2000, pp. 1561-1572.

[68] K. Kundert, “A formal top-down design process for mixed-signal circuits,” Advances in Analog Circuit Design, Apr. 2000.

[69] M. McCorquodale, F. Gebara, K. Kraver, E. Marsman, R. Senger, and R. Brown, “A top-down microsystems design methodology and associated challenges,” in DATE Designers’ Forum Proc., Mar. 2003, pp. 292-296.

[70] M. McCorquodale, E. Marsman, R. Senger, F. Gebara, M. Guthaus, D. Burke, and R. Brown, "Microsystem and SoC Design with UMIPS," *IFIP Intl. Conf. on VLSI SOC*, pp. 324–329, 2003.

[71] University of Michigan Intellectual Property Source (UMIPS), http://www.eecs.umich.edu/umips/, 2004.

[72] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, “MiBench: A free, commercially representative embedded benchmark suite,” IEEE Workshop on Workload Characterization, pp. 10–22, Dec. 2001.

[73] C. Lee, M. Potkonjak, and W. Mangione-Smith, “Mediabench: A tool for evaluating and synthesizing multimedia and communications systems,” Intl. Symp. on Microarchitecture, pp. 330–335, 1997.

[74] R. Senger, E. Marsman, S. Kellis, and R. Brown, “Methodology for Instruction Level Power Estimation in Pipelined Microsystems,” Austin Conf. on Integrated Systems and Circuits (ACISC), May 2007.

[75] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," Trans. on VLSI Systems, vol. 2, no. 4, pp. 437–445, Dec. 1994.

[76] SPARC International Inc. The SPARC Architecture Manual, Version 8, www.sparc.com/standards/V8.pdf, 1992.

[77] V. Tiwari and M. T.-C. Lee, "Power analysis of a 32-bit embedded microcontroller," Asia and South Pacific Design Automation Conf., pp. 141–148, Aug. 1995.

[78] B. Klass, D. Thomas, H. Schmidt, and D. Nagle, "Modeling Inter-Instruction Energy Effects in a Digital Signal Processor," Power Driven Microarchitecture Workshop in ISCA'98, 1998.

[79] Q. Wu, P. Juang, M. Martonosi, and D. Clark, "Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors," HPCA, pp. 178–189, Feb. 2005.

[80] H. Mair and L. Xiu, "An architecture of high-performance frequency and phase synthesis," J. Solid-State Circuits, vol. 35, pp. 835–846, June 2000.

[81] Tensilica Inc. Xtensa Architecture and Performance, http://www.tensilica.com/xtensa arch white paper.pdf, Sept. 2002.

[82] A. J. Annema, “Analog circuit performance and process scaling,” *Trans. on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 46, pp. 711–725, June 1999.

[83] K Jerger, et al., “Early Seizure Detection“, *Journal of Clinical Neurophysiology,* Volume 18, Issue 1, pp 259-268, 2001.

[84] U. Kramer, et al., “A Novel Portable Seizure Detection Alarm System: Preliminary Results“, *Journal of Clinical Neurophysiology*, Feb. 2011, Volume 28, Issue 1, pp 36-38, Feb 2001.

[85] Shriram Raghunathan et al, “The design and hardware implementation of a low-power real-time seizure detection algorithm” *Journal of Neural Engineering*, Vol. 6, Issue 5, Oct. 2009

[86] Haensch, W., et al., “Silicon CMOS devices beyond scaling“, *IBM Journal of Research and Development,* Vol 50, Issue 4.5, pp 339-361, July 2006.