INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

A Stimulation Controller Microcircuit for use in a Cochlear Implant System

by

Colin Loiselle, B.E.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of

Master of Engineering

Department of Electronics
Ottawa-Carleton Institute for Electrical Engineering
Carleton University
Ottawa, Ontario

April 30, 2000

©copyright
2000, Colin Loiselle

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-57731-7

Canada

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

**“A Stimulation Controller Microcircuit for use in a
Cochlear Implant System”**

submitted by Colin Loiselle, B.E.
in partial fulfilment of the requirements
for the degree of
Master of Engineering

Dr. T. A. Kwasniewski
Thesis Supervisor

Dr. N. G. Tarr
Acting Chair, Department of Electronics

Carleton University
July 24, 2000

Abstract

An ultra low power, programmable microcircuit has been designed for use in a cochlear implant system. The implantable hardware is capable of supporting a wide variety of electrical stimulation and speech processing strategies. Following a brief overview of existing cochlear implant designs, the programmable stimulation controller is described in detail. The microcircuit is capable of simultaneously driving ten bipolar channels as traditional bipolar pairs or in a number of common ground or cross-channel stimulation configurations. A finite state machine (FSM) based control unit and a small set of function specific registers provide the core circuitry and interface between the ten independent channel drivers and an on-chip RAM. A small set of commands have been defined to facilitate the upload of auditory stimulation information from an external signal processor. The circuit has been coded in VHDL, simulated, and synthesized to arrive at a power consumption estimate of 1.6 mW.

Acknowledgements

I would like to thank my supervisor, Dr. T. A. Kwasniewski, for his support and guidance throughout the duration of this project. As well, I would like to express my deepest appreciation to Professor Gary Tarr, Professor Robert Harrison, and Professor David Walkey—each of whom served as graduate advisor at one point in my term—and the Faculty of Graduate Studies, for their patience in continuing to extend my student status and allowing me to pursue this work.

To the office staff in the Department of Electronics, I offer my admiration of their administrative skill and infinite patience, especially when handling those last minute extension requests.

I would like to thank the Natural Sciences and Engineering Council of Canada for their funding, and Nortel Networks for the use of their resources. A special thanks goes out to my managers and co-workers at Nortel Networks, most notably Cliff Townsend, Andre Brissette, Sandra Sousa, and Ingrid Renda, for their advice, support, and freely given technical expertise.

A very special thanks goes out to my parents and all my family for a lifetime of love and guidance.

To my wife Teresa and my daughter Julia: you are my joy and inspiration. Teresa, I would never have come this far without your steadfast love and support. Thank you.

Contents

Abstract.....	iii
Acknowledgements.....	iv
List of Tables	vii
List of Figures.....	viii
1.0 Introduction	1
1.1 Motivation	1
1.2 Objectives	3
1.3 Thesis Organization.....	4
2.0 Background	6
2.1 Physiology	6
2.2 Electrical Stimulation of the Ear	10
2.3 Modern Cochlear Implants	13
2.3.1 General Device Description	13
2.3.2 Nucleus (Cochlear Limited).....	18
2.3.3 Clarion.....	24
2.3.4 University of Utah/Ineraid	27
2.3.5 Digisonic	29
2.3.6 Multichannel Urinary Prosthesis.....	30
2.4 Summary.....	32
3.0 A Proposed Receiver-Stimulator Architecture.....	34
3.1 Introduction	34
3.2 Design Requirements.....	36
3.2.1 Safety	36
3.2.2 Power	37
3.2.3 Flexibility	38
3.3 Desired Stimulus.....	40
3.4 Data Format	40
3.4.1 Stimulation Frame.....	40
3.4.2 Pulse Description	42
3.4.3 Transmission Protocol.....	44
3.5 Processor Architecture.....	47
3.5.1 Block Diagram	48
3.5.2 Data Unit	50
3.5.3 Memory	56
3.5.4 Logic Unit	57
3.5.5 Control Unit	59
3.5.6 Channel Driver.....	59
3.6 Operation	68

3.6.1	Programming Mode	68
3.6.2	Stimulation Mode.....	71
4.0	Circuit Design and Synthesis	75
4.1	Design Synthesis.....	75
4.2	MPS Register Transfer Level Description.....	79
4.2.1	Design Partitioning	79
4.2.2	Top Level	82
4.2.3	Data Unit	83
4.2.4	Logic Unit	85
4.2.5	Control Unit	86
4.2.6	Channel Driver	88
4.2.7	MUX/DEMUX.....	90
4.2.8	Other VHDL Entities	90
5.0	Conclusion	92
Appendix A	MPS VHDL Description.....	95
A.1	Top Level.....	95
A.2	Data Unit.....	99
A.3	Logic Unit.....	107
A.4	Control Unit.....	109
A.5	Channel Driver	119
A.5.1	Channel Driver Data Unit	121
A.5.2	Channel Driver Control Unit.....	124
A.6	Driver Control/Status Signal Mux/Demux	128
A.7	RAM Collar	130
A.8	Behavioural RAM.....	131
A.9	Misc Package	132
A.10	Top Level Test Bench	132
Appendix B	PERL Pulse Compiler	134
B.1	Example Pulse Description.....	134
B.2	PERL Pulse Compiler.....	134
B.3	Generated Output Vectors	140
	References.....	142

List of Tables

Table 1: Data Unit Register Descriptions54

Table 2: MPS Combinational Logic57

Table 3: Channel Driver State Machine Output.....66

Table 3: Design Description Hierarchy [AG93]76

List of Figures

Figure 1: The Human Ear [Pic88]	7
Figure 2: Cross Section of the Cochlea [Pic88]	9
Figure 3: Nucleus Speech Processor – Formant Extraction Techniques.....	21
Figure 4: Nucleus Spectral Maxima Sound Processor [Loi98]	22
Figure 5: Nucleus Spectra 22 Speech Processor	23
Figure 6: Clarion Speech Processor – Compressed Analog (CA) Approach.....	25
Figure 7: Clarion – Continuous Interleaved Sampling (CIS) Approach	26
Figure 8: University of Utah/Ineraid Signal Processing Block Diagram	28
Figure 9: Multichannel Urinary Prosthesis Microprocessor Block Diagram [SDH+92]	32
Figure 10: Proposed Cochlear Implant System [Wit92]	35
Figure 11: Stimulation Waveform Examples	41
Figure 12: Instruction Pulse Data Word.....	42
Figure 13: Instruction Pulse Control Word	43
Figure 14: Single word command	45
Figure 15: Block LOAD command	45
Figure 16: Block download structure	47
Figure 17: MPS Top Level Block Diagram	49
Figure 18: Data Unit Block Diagram	51
Figure 19: Channel Status Register	53
Figure 20: MPS Memory Partitioning.....	56
Figure 21: Finite State Machine Architecture	60
Figure 22: Channel Driver Block Diagram	61
Figure 23: Channel Driver State Diagram.....	66
Figure 24: Channel Driver Timing Diagram Example.....	67
Figure 25: Control Unit State Diagram: Programming Mode.....	70
Figure 26: Control Unit State Diagram: Stimulation Mode	74
Figure 27: MPS RTL Hierarchical View	80

Chapter 1

Introduction

1.1 Motivation

Profoundly deaf individuals suffer from more than 80 to 90 dB hearing loss in both ears [Par85]. In general, these people do not receive any significant benefit from the use of a standard acoustic hearing aid. Mechanical deafness—that is, deafness due to injury or disease of the eardrum or the bones of the middle ear—can in most cases be treated surgically [Loe90]. The remaining types of deafness are sensorineural in nature [Loe90]. Sensorineural deafness can be caused by a failure of the hair cells in the inner ear (sensory), the auditory nerve itself (neural), or both [Par85]; however, it has been suggested that the source of sensorineural hearing loss lies more towards a sensory nature than neural [Par85]. In most cases, some elements of the auditory nerve in deaf individuals still function [Loe90]. Thus, while the hair cells are no longer capable of triggering the auditory nerve fibres, the fibres themselves can be electrically stimulated to produce auditory sensations. It is this remaining functionality that can be exploited by a cochlear implant.

A cochlear implant is a prosthetic device used to restore hearing in patients suffering from profound sensorineural deafness. The implant bypasses the normal function of the middle and inner ear by applying an electrical stimulus directly to the auditory nerve. If the nerve fibre has not degenerated, it can be activated by an externally applied electrical current of the proper strength, duration, and orientation. These externally evoked impulses are indistinguishable from those generated by the natural response to acoustic signals in a healthy ear. They are interpreted by the brain as sound [Loe85a]. The cochlear

implant is now the preferred method of rehabilitation for post-lingually deafened adults [GTK+88] and may hold even more promise for pre-lingually deafened children [Loe90].

There are numerous different designs in varying stages of development, with over 7000 individuals worldwide currently wearing some form of cochlear implant [Zim93]. The basic structure of the device commonly consists of four main components: a microphone, an external processor, the implanted receiver-stimulator, and the cochlear electrode array [HvdPD+87]. The microphone gathers the incoming sound and passes it to the speech/signal processing unit. Typically, the goal of the signal processor is to extract some specific information from the incoming auditory signal and then groom this information for transmission to the stimulating unit. The receiver-stimulator generates the stimulation waveforms as specified by the external processor and drives the implanted electrode(s).

While cochlear implants have shown much promise, there have been varying levels of success among cochlear implant patients. Some subjects have experienced little more than a dull buzzing while others have achieved full open set speech discrimination and are capable of conversation without any visual cues [Loe90][Zim93]. In fact, a wide range in performance can often be seen among patients using the same type of implant [GTK+88]. Studies intended to predict performance prior to implantation (pre-operative evaluation) have generally been unsuccessful [HBM87]. This patient variability has been called “the single largest hindrance to the widespread clinical application and commercial success of cochlear prosthetics” [Loe90].

It has been shown that a larger number of channels can provide a better representation of the speech signal to the auditory nerve. Experiments suggest that at least six channels are necessary for intelligibility [Loe85a], covering an overall frequency range of 350 Hz to 5500 Hz [LWZF99]. One sizable challenge of driving multiple channels is to

provide functional separation between the electrodes. The conductive perilymph fluid and surrounding tissues in the cochlea can provide a path for the signal from one electrode to nearby neurons in the same and even adjacent turns of the cochlear spiral, interfering with the signal from other electrodes. Radially bipolar electrode pairs provide better performance than a longitudinally bipolar approach [Loe90] but are still not immune to interference between electrode pairs if enough electrodes are placed for intelligibility. Improvements in electrode design as well as processing strategies to compensate for electrode interference should provide better control in stimulation at selected sites [GC87]. An optimal cochlear implant system should have the flexibility to allow tailoring of the processing and stimulation strategy to the individual user and support future updates in processing and stimulation strategies.

1.2 Objectives

The purpose of this thesis is to explore the feasibility of a programmable, low power, semi-custom circuit capable of supporting a wide range of existing and future speech processing and electrical stimulation strategies. The scope of this document covers the design and analysis of the digital core of the receiver-stimulator. The analog portions of the implanted circuitry—an efficient, low power receiver circuit and the digital to analog converters for driving the electrode array—will be designed in future work. Much of this design is based upon initial concepts outlined by Withey, 1992, including the name (used with permission): Multipulse Programmable Stimulator, or MPS. Other architectural decisions stem from a review of the state of the industry, covered in Chapter 2, and attempt to address the strengths and weaknesses of various existing implant systems. This review helps define the design goals and constraints for the circuit.

The implantable receiver-stimulator is intended for use with an external signal processor and an intra-cochlear electrode array of up to ten bipolar channels. The ten channels should support both bipolar and monopolar stimulation strategies. A stimulation rate of up to 3250 pulses/second (pps) would be necessary to support the processing strategies of existing implants. A faster rate may be desirable in order to support future advances. Stimulation pulse widths between 0 and 500 μs would match those of the current implants, and support for a wider range of pulse widths would provide additional flexibility. In addition to pulsatile stimulation, the MPS should also support analog waveforms. The amplitude of a pulse should range between 0 and 2500 μA .

A compact and yet versatile digital architecture has been defined that allows the user (via the external processing unit) to fully describe a stimulating pulse: its amplitude, duration, and polarity. A pulse train can be created from a series of these pulses, allowing a wide variety of waveforms to be rendered. It is this flexibility that allows the circuit to support the existing processing and stimulation strategies and should allow future approaches to be incorporated. Based on the design specifications, the hardware description language source code for the MPS digital core was written. The chip has been fully simulated and synthesized to arrive at a gate level description, and circuit power estimates have been derived from the gate description.

1.3 Thesis Organization

Chapter 1 is the introduction. It highlights the motivation and objectives of the thesis and provides an outline of the document structure. Chapter 2 provides an introduction to the physiology of hearing and a brief overview of the history of cochlear implants. This is followed by a review of four currently available cochlear implant systems. A fifth cir-

cuit included in the review is an implanted stimulation controller which may support other neuro-stimulator applications but was initially intended for use in a urinary prosthesis. In Chapter 3 the design requirements for a programmable stimulation controller are outlined. Based on the high level requirements, the circuit operation is defined and followed by a detailed design description. Chapter 4 outlines the translation of the design description specified in Chapter 3 into synthesizable VHDL (Very high speed integrated circuit Hardware Description Language) code. The rationale behind the VHDL coding style is described in detail, with an emphasis on design for simulation and synthesis. The VHDL code itself is listed in Appendix A. Conclusions and observations are made in Chapter 5 based on the circuit simulation and synthesis results.

Chapter 2

Background

2.1 Physiology

In order to understand the general operation of a cochlear implant system, some background in hearing physiology is necessary. The following description of the operation of the ear is based on Pickles' *An Introduction to the Physiology of Hearing* [Pic88].

The human ear (Figure 1) can be divided into three distinct regions commonly referred to as the outer ear, middle ear, and inner ear (cochlea). The entire system acts as a transducer to convert sound waves, which are simple pressure variations in the surrounding air, into neural (i.e., electro-chemical) impulses to be relayed along the auditory nerve to the brain.

The outer ear consists of the pinna—the visible portion on the side of the head commonly referred to as the “ear”—the ear canal, and the tympanic membrane (eardrum). The primary purpose of the outer ear is to direct sound waves from the external medium to the eardrum and into the middle ear. In doing so, the resonances of the outer ear also provide 10-20 dB of gain in the frequency range of 1.5 kHz to 8 kHz. As well as enhancing sensitivity, this amplification is directionally dependent. Thus the outer ear also provides valuable clues as to the direction of the source of the sound.

The middle ear couples sound waves from the tympanic membrane to the inner ear. The tympanic membrane converts the sound waves into mechanical motion and relays the signal through three small bones—the malleus (hammer), the incus (anvil), and the stapes (stirrup), known collectively as the ossicles—to the inner ear. The stapes, the

Figure 1: The Human Ear [Pic88]

last bone in the chain, is connected to a thin membrane covering a small opening in the cochlea called the oval window. Movement of this membrane causes motion of the fluid in the cochlea and couples the acoustic wave into the inner ear.

The middle ear acts as an impedance transformer between the external environment (air) and the fluid-filled cochlea. The tympanic membrane and the ossicles convert the relatively high amplitude, low pressure air waves to lower amplitude, high pressure vibrations at the round window. Without this transformation, only about 1% of the incident energy of the auditory signal would be transmitted to the cochlea.

The responsibility of the inner ear is to convert the mechanical impulses from the stapes into electrical nerve impulses. The inner ear, or cochlea, is a bony, fluid filled tube—about 35 mm long in humans—which is coiled two and a half turns into a spiral resembling a snail's shell (Figure 2). The cochlear interior is divided into three fluid filled chambers by two membranes known as Reissner's membrane and the basilar membrane. The chambers are called the scala vestibuli, the scala media, and the scala tympani. Resting on the basilar membrane within the scala media is the organ of Corti. The organ of Corti contains the hair cells that trigger the auditory nerve.

The auditory stimulus is coupled into the cochlea by the stapes vibrating on the oval window. The motion of the oval window sends a pressure wave propagating through the perilymph and induces a wavelike displacement of the basilar membrane. The organ of Corti moves with the basilar membrane, pushing the stereocilia atop the hair cells against the tectorial membrane above. This causes a deflection of the stereocilia, which in turn releases a neurotransmitter and induces the auditory nerve to fire.

Figure 2: Cross Section of the Cochlea [Pic88]

Place Pitch Theory

The basilar membrane is tapered and has resonant frequencies along its length, from 20 kHz at the base by the oval window, to about 20 Hz at the apex. A stimulus of a specific frequency will induce a large displacement in the basilar membrane at one location, decreasing in amplitude as one travels away from this point. The nerve fibres at each point are dedicated to the resonant frequency of that portion of the basilar membrane. The basilar membrane thus breaks up the incoming wave into its component frequencies, and this information is sent to the brain through the auditory nerve. This method of conveying information regarding the frequency content of the sound is referred to as the place pitch theory and is fundamental to the design of many cochlear implants.

The other manner in which frequency information is sent to the brain is called the volley theory of hearing. Nerve cells are limited by their refractory period to a maximum individual firing rate of roughly 300 pulses per second [Loe85a], much less than the 20 Hz to 20 kHz bandwidth of the human auditory system. Fortunately, the human cochlea is innervated with about 30 000 sensory neurons. Thus, while no single auditory neuron can be stimulated faster than 300 Hz, the parallel array of low frequency channels can respond to high rates of stimulation.

2.2 Electrical Stimulation of the Ear

The history of the modern cochlear implant, as outlined by Simmons [Sim66], begins with the first attempts at electrically stimulating the ear. In 1800, Alessandro Volta was the first to stimulate the auditory system by connecting two electrodes across 50 volts and placing them in each ear canal. The results were a shock to the head followed a few moments later by a bubbling sound like “the boiling of thick soup”. Additional attempts

by Volta and others over the next 50 years produced similar momentary sensations but nothing with any tonal quality.

Guillaume Benjamin Amand Duchenne of Boulogne, France in 1855 used an alternating current source to the electrodes. Since sound was an alternating disturbance, it was thought that a direct current source would not provide a satisfactory response. Unfortunately, the results were unsatisfactory, and he reported sounds like “the beating of a fly's wings between a pane of glass and a curtain”.

In 1868, Brenner released a study examining the effects of altering the polarity, rate and intensity of the stimulus, and the physical placement of the inserted electrodes. He also was one of the first to use a bipolar electrode, placing one electrode immersed in saline in the external ear canal, and the other on a distant part of the body.

In 1930, Wever and Bray reported that the response of the auditory nerve in cats to electrical stimulation resembled that due to natural auditory stimulation. Studies by Stevens and Jones (1939) and Jones et al. (1940) indicated that there were three mechanisms responsible for an auditory response to electrical stimulation. The first, known as the eletrophonic effect, results from the transduction of the electrical signal into mechanical motion by the middle ear, with further propagation of the sound wave through the middle and inner ear as normal. The second method was seen in patients in which the middle ear bones were not present. The basilar membrane in the inner ear would vibrate at a point determined by the frequency of the electrical stimulus and activate the corresponding hair cells. Finally, a fairly crude perception of sound was heard by patients with little or no hearing. This third mechanism for producing an auditory response was probably a result of direct electrical stimulation of the auditory nerve.

The first two mechanisms required that a certain amount of residual cochlear function remain, and in general are of little use to profoundly deaf individuals. The third method, direct stimulating of the auditory nerve, only required that the auditory nerve remain intact. However, a fairly widespread electrical field in the region of the inner ear produced a crude sound and would not be sufficient to correct for total perceptive deafness. More localized stimulation of the auditory nerve would be required.

One of the first attempts at stimulation of the auditory nerve was made by Lundberg in 1950 with little more than noise produced. Djournio and Eyries in 1957 were more successful. Through better control of stimulus parameters the patient was able to discriminate differences in pitch of 100 pulses per second and understand a few simple words such as “allo” and “papa”. It was also during this study that they determined that frequencies above about 1 kHz could not be conveyed due to the refractory period of the auditory nerve fibres.

Doyle et al in 1964 attempted to exploit the place pitch theory of frequency coding by inserting an array of four electrode and limiting the spread of the electric field of each. Extensive studies by Simmons (1966) and Simmons et al (1965) described the operation of a six electrode array. Both monopolar and bipolar electrodes were tested, as were the affects of altering stimulation frequency and intensity. Results showed that changes in pitch up to 300 pulses per second could be detected. Also, the pitch perceived varied with the electrode location, and a rough frequency scale along the length of the basilar membrane could be seen. When speech signals were used as the stimulus waveform, the sound was recognized as speech although the speech itself could not be understood. Attempts were made to process the speech into frequency bands, but again this produced no intelligibility.

2.3 Modern Cochlear Implants

Early cochlear prostheses provided important lipreading cues, but the majority of implantees were generally unsuccessful at discriminating speech without additional visual aids [GTK+88][Loi98]. Two single channel systems, the 3M House and Vienna/3M devices, achieved moderate success in the early 1970s and '80s respectively [FF85][Loi98]. These and other single channel devices, however, are unable to take advantage of the tonotopic mapping of the auditory nerve fibers along the basilar membrane. They cannot convey enough frequency information through a single electrode to support sound-only speech discrimination except in the cases of a few exceptional patients [GTK+88][Tyl88]. It was discovered early on that multiple channels, at least four to eight, would be required to provide a high level of speech intelligibility [GTK+88].

Following a general overview of the predominant characteristics of a cochlear implant system, four multichannel devices are reviewed with respect to their speech processing and electrical stimulation strategies, electrode design, and coupling methodology. While far from being a comprehensive review of the state of the industry, this section attempts to highlight what features are to be desired in an “ideal” cochlear implant system. Two of these systems (Nucleus and Clarion) have been approved for commercial use in the United States by the American Food and Drug Administration.

2.3.1 General Device Description

Most cochlear implants systems generally consist of four main components: a microphone, an external processor, the implanted receiver-stimulator, and the cochlear electrode array [HvdPD+87]. The current cochlear implant designs vary in their electrode

geometry and placement, signal processing strategy, stimulation strategy, and coupling methodology.

Electrode Geometry and Placement

The electrode design and its placement within the ear have a profound effect on the performance of a cochlear implant. It also dictates, to some extent, the form of signal processing and method of stimulation that will be used on the implant. Electrode design can be separated into two broad categories: single channel and multichannel. Single channel devices use two electrodes, one source and one return electrode. These implants provide only a broad sensation of pitch and then only at frequencies of a few hundred hertz, but they do give the user a good sense of the rhythm and loudness of sounds, including their own speech, and have been a valuable aid in lip reading and learning to speak [Loe85a]. Single channel implantees have very poor word understanding and are generally unable to converse without visual aids [GTK+88][SK92].

Multichannel devices use an array on the order of eight to twenty-two electrodes. They attempt to use the frequency mapping of the auditory nerve by placing multiple, independently controlled electrodes along the length of the cochlea. Each electrode thus attempts to elicit a response in the specific frequency range addressed by the auditory nerve fibres near each electrode. While each electrode does not produce the desired pure tone, the sounds can be ranked by pitch related to the tonotopic map of the cochlea [Loe90].

Multichannel devices can be further classified as monopolar or bipolar. Monopolar devices use multiple source electrodes with a common return path placed at a remote site in the surgical area [Par85]. Bipolar electrodes allow stimulation between pairs of electrodes both placed at the interface site. Bipolar stimulation results in much less current

spread than monopolar stimulation and thus greater control of the frequency specific neurons at each site [Par85]. The increased current spread with monopolar electrodes also gives rise to increased interference between electrodes [GC87][Loe90]. However, bipolar electrodes require significantly higher stimulation current than monopolar because a good deal of current flows directly between the electrode pair rather than longitudinally along the nerve cell axon. The current density in the tissue surrounding the electrode is much higher with bipolar stimulation, increasing the risk of producing toxic products [Par85].

In the design of bipolar electrodes, the orientation of the two electrodes is important. Radial bipolar electrodes are arranged such that the source and return electrodes are aligned perpendicular to the long axis of the cochlear spiral and parallel to the auditory nerve fibres. Longitudinal bipolar electrodes are aligned perpendicular to the auditory nerve fibres. Electrodes arranged somewhere between these two extremes are termed offset radial bipolar. Theoretical and practical studies have shown that better spatial selectivity is obtained using radial bipolar electrodes [FWW89][MW77].

Both multichannel and single channel designs have had the contacts placed either outside or inside the cochlea [Loe90]. Extra-cochlear designs usually place the electrode(s) in the middle ear, on the round window. Intracochlear electrodes are usually inserted through the round window and along the tympanic canal (scala tympani), although a few designs have placed the contacts directly on the auditory nerve itself. Intracochlear electrodes have the advantage of being in closer proximity to the auditory nerve and have a greater chance of utilizing the tonotopic map of the auditory nerve fibres along the basilar membrane. However, the implantation of the electrode along the tympanic canal usually results in the loss of any form of acoustic hearing [Loe90] as well as the risk of irreparable damage to the surviving auditory nerve fibres [Loe85a]. Extra cochlear elec-

trodes have less spatial selectivity because of the increased distance to the auditory nerve. However, the reduced risk of damage and infection from the surgical implantation often make extra-cochlear electrodes a safer alternative to intra-cochlear electrodes.

Signal Processing

In general, processing of the acoustic signal by the cochlear implant attempts to recreate at least part of the signal processing that would occur in a healthy ear up to the level of the prosthesis interface. This includes some straightforward acoustic filtering in the outer and middle ear plus the processing that occurs in the cochlea. In addition to the mechanical bandpass filtering by the basilar membrane, there is some neural sharpening of this band pass filter and half wave rectification by the hair cell, as well as a 1 kHz low pass filter and a neural adaptation effect at the hair cell/neural interface [Par85].

One of three approaches is generally used to obtain a stimulus waveform from the acoustic signal. First generation devices used the actual acoustic waveform itself, suitably band filtered and compressed in its dynamic range, as the stimulation waveform. It was hoped that the brain would be able to make use of some of the information carried in the analog waveform. These devices provide no speech discrimination [Par85], and the temporal-spatial patterning resulting from this method of stimulation has been termed “highly unphysiological” [Loe90]. In spite of this, these devices have proven to be valuable aides to lip reading.

A second approach achieves improved speech comprehension by converting the output from a set of analog filters into a set of short pulses. These pulses are then delivered to the auditory nerve fibres in accordance with the place pitch theory.

The third approach makes use of the fact that the primary goal of a prosthetic hearing device is speech recognition. Speech feature extraction techniques such as formant tracking, linear predictive coding, and cepstrum analysis are used in an attempt to transmit information important in speech discrimination to the neural elements [Won91][MB89][Ged90].

Stimulation

The manner in which the stimulation current is delivered to the surviving auditory nerve fibres is an important factor in achieving speech comprehension. In general, the stimulating current attempts to evoke as natural a response as possible in order to simulate the firing patterns of a healthy ear. However, the complexity of the auditory system often forces the implant designer to use a less natural approach and provide just enough information to the central nervous system (CNS) to allow speech comprehension. Additional training is required in order for the patient to discriminate these new sounds to achieve speech comprehension. Most of the current implant systems use this strategy, some achieving modest speech comprehension without the aid of lip reading [GTK+88].

The electrical stimulus is usually an analog form of the original acoustic stimulus or a series of short pulses depending upon the form of speech processing chosen. The dynamic range of the auditory nerve fibres to electrical stimulation is generally between 10 and 20 dB, from the threshold of hearing to the threshold of pain. For safety reasons, the waveform must consist of biphasic, charge balanced pulses, with no DC component. This prevents the production of toxic products as a result of electrolysis. As well, the current density must be well controlled so that any electrolysis products from the first half of the waveform will be recaptured by the second half of the biphasic pulse.

Coupling Methodology

Because of the size constraints on the implantable portion of the cochlear prosthesis, most of the signal processing is done outside the body. Two methods of transmitting the stimulation waveforms to the implanted receiver-stimulator are generally used: direct connection through a percutaneous plug affixed to the skull and passing through the scalp, or transcutaneously using inductive coupling of a radio frequency (RF) signal. The percutaneous method offers increased simplicity of the implanted electronics in that all that is required is the implanted plug and the electrode array. The percutaneous connection also allows easier monitoring of the implanted electronics. The disadvantages involve the irritation from and risk of infection due to the implanted plug, the chance of disconnection, as well as the overall aesthetics of the device. The transcutaneous coupling method requires a more complex implanted receiver-stimulator to decode the transmitted RF signal and prevents detailed monitoring of the implanted circuitry. As well, if a magnet is used to align the external and implanted receiver, compression of the tissues between the two magnets can lead to discomfort. Improved aesthetics and a reduced risk of infection over a percutaneous link are the advantages of a transcutaneous design.

2.3.2 Nucleus (Cochlear Limited)

The Nucleus cochlear implant was developed at the University of Melbourne by Clark and associates and first implanted in 1978. Since then, there have been more than 1500 recipients of the device. As of 1990, it is the only cochlear prosthesis approved for marketing by the American Food and Drug Administration [Loe90]. Information on this design was compiled from a number of sources [C+87][Cla93][CTP90][Won91].

The design uses a 22 channel, longitudinally bipolar electrode array. The electrode array consists of 22 platinum bands supported along the distal 17 mm of a flexible silicon

rubber carrier. It is designed to support bipolar and common ground stimulation. Generally, two adjacent electrodes will be used as the source/sink pair in an attempt to stimulate a distinct subset of auditory nerve fibres, but any two electrodes can act as a bipolar stimulating pair. A common ground configuration is available by using a single electrode as the source and connecting the remaining electrodes together to provide a return path.

The Nucleus implant has undergone a number of changes, most notably in the speech processing strategy, since it was first implanted in 1978. Initially, the Nucleus speech processor incorporated a formant tracking strategy. In this approach, the speech processor attempts to identify the broad spectral peaks in the received signal. These peaks are called formants, and they carry some information about the speech signal. The first three formants—F1, F2, and F3—actually contain enough information to identify voiced sounds such as vowels and have also been found to play a role in the identification of unvoiced sounds, e.g., consonants [Loi98].

The first version of the Nucleus implant was based on formant extraction and attempted to code pitch by both rate and place of stimulation. A block diagram of earlier implementations of the Nucleus filtering strategy is shown in Figure 3. An automatic gain control (AGC) block compresses the speech signal to reduce its dynamic range to between 10 dB and 20 dB, suitable for electrical stimulation. The fundamental frequency (F0) and the second formant frequency (F2) and amplitude (A2) are extracted. F2 is used to provide the place of stimulation along the electrode array. The pulse amplitude is set proportional to A2. The electrode is stimulated at a rate of F0 pulses per second for periodic (voiced) sounds, or at a pseudo-random frequency between 200 Hz and 300 Hz for aperiodic (unvoiced) sounds. In this manner it was hoped that pitch information from both F0 and F2 could be delivered simultaneously to a single electrode pair.

This processing strategy was later modified with the addition of information from the first formant (F1). The frequency information from this formant was used to select the stimulation location of a second electrode near the apex of the cochlea. As shown in Figure 3, F0 was again used as the stimulation frequency for voiced sounds, this time for two electrodes. The electrical pulses to each electrode are spaced in time to minimize inter-channel interactions. The addition of F1 resulted in improved word recognition and sentence comprehension but could not improve on the recognition of consonants which are generally comprised of more information at higher frequencies [Cla93].

In an attempt to improve the recognition of higher frequency sounds, Nucleus added three more bandpass filters in the higher frequencies and called the new processing strategy MPEAK. The new speech processor used the outputs of the high frequency bandpass filters to drive three electrodes—electrodes one, four and seven—near the base of the cochlea. The different formant extraction techniques used by Nucleus are shown in Figure 3.

The formant tracking techniques were relatively successful, but were prone to errors in formant identification, especially in noisy situations [Cla93]. As well, a formant based system attempts to provide information relating specifically to speech intelligibility and is not geared toward eliciting a natural neural response patterns [Won91]. A new processor called the Spectra Maxima Sound Processor (SMPS) was developed in the early 1990s. The SMPS uses a filterbank approach, passing the speech signal through a bank of 16 bandpass filters with center frequencies from 250 Hz to 5400 Hz. The filter outputs are mapped to the first 16 (apical) electrodes of the 22 electrode array. The six largest filter outputs are selected from the filterbank every 4 ms. These six channels are stimulated in an interleaved pattern at a fixed pulse rate of 250 pulses per second (pps) and with an

* AGC: Automatic Gain Control

** F2 extraction filter bandwidth = F0/F2: 300-1000 Hz
 WSP: 1000-3000 Hz
 MPEAK: 800-4000 Hz

Figure 3: Nucleus Speech Processor – Formant Extraction Techniques

Figure 4: Nucleus Spectral Maxima Sound Processor [Loi98]

amplitude proportional to the filter outputs. The performance of this new processing and stimulation strategy again improved upon the previous designs [Loi98].

The most recent update to the Nucleus system incorporates a signal processor which can be programmed with either the formant extraction techniques of the earlier implants (F0/F1/F2, MPEAK) or an expanded version of the SMPS filter-bank approach called the Spectral Peak (SPEAK) speech strategy. The Nucleus Spectra 22 processor con-

Figure 5: Nucleus Spectra 22 Speech Processor

sists of two custom integrated circuits (ICs) that contain the circuitry to perform feature extraction on the incoming speech, a programmable filter bank to implement the SPEAK processing strategy, an encoder to encode either the spectral maxima or the speech features to the proper electrodes in preparation for transmission, as well as circuitry to accommodate the necessary analog pre-processing.

The Nucleus receiver-stimulator was designed to allow a certain amount of flexibility in the stimulus parameters, as a speech processing strategy had not been determined prior to the initial trials. The stimulating current pulse can be adjusted in amplitude from $25\ \mu\text{A}$ to $1.5\ \text{mA}$ in 3% steps and in width from $20\ \mu\text{s}$ to $400\ \mu\text{s}$ per phase in steps of $0.4\ \mu\text{s}$. Stimulation rates in excess of 1000 pulses per second are available. The receiver-stimulator is implemented on a single custom IC containing digital and analog circuitry. It

is encased in a welded titanium package with the receiver coil placed externally around the package.

A transcutaneous, inductive coupling was chosen as the method of transmitting information from the external processor to the implanted circuitry. Digital transmission of the stimulation data was chosen. This allows for a single transmission path and provides additional noise immunity over analog transmission. A carrier frequency of 2.5 MHz is used.

Although 22 bipolar channels are available, the Nucleus device was designed to support non-simultaneous stimulation of multiple channels. The reason for this is that variations in loudness with two or more simultaneous stimulus channels cannot yet be adequately predicted [C+87].

2.3.3 Clarion

The Clarion Multichannel Cochlear Implant is a product of the collaborative efforts of the University of California at San Francisco (UCSF), the Research Triangle Institute (RTI), the Research Triangle Park, North Carolina, and Advanced Bionics Corporation of California (formerly MiniMed Technologies) [SK92]. The Clarion prosthesis is a relatively new device based on earlier designs developed by Storz Medical Instruments and UCSF. Clinical trials for the Clarion implant began in 1991, and the device has recently been approved for commercial use by the American Food and Drug Administration (FDA) [Loi98].

The Clarion implant uses a 16 channel, intracochlear electrode array. The contacts consist of 16 platinum-iridium (90%/10%) ball electrodes oriented as eight near-bipolar pairs imbedded along 25 mm of silicone rubber carrier.

Figure 6: Clarion Speech Processor – Compressed Analog (CA) Approach

The external speech processor consists of a wide band (250 Hz to 5500 Hz) automatic gain control (AGC) circuit followed by a bank of programmable band-pass filters. The implanted circuitry is capable of driving eight independent channels at stimulation levels from 1 μA to 2500 μA , in 255 increments. The eight channels are updated in basal to apical order at a rate of up to 13 000 samples/second. A single channel can be updated maximum rate of 91 000 samples/second. Monopolar and bipolar modes of stimulation are available and the waveform can be analog or pulsatile.

Figure 7: Clarion – Continuous Interleaved Sampling (CIS) Approach

The system hardware allows for a variety of speech processing and stimulation strategies. The processor can support the compressed analog (CA) stimulation strategy of the original UCSF/Storz design [WFF+88] or a more sophisticated continuous interleaved sampling (CIS) strategy. Using the CA approach, the input sound is compressed, passed through a bank of eight filters, and then delivered simultaneously to the implanted electrode pairs. This strategy retains the temporal information of the sound but sacrifices spatial selectivity of the stimulating pulses due to interactions between electrodes.

CIS attempts to reduce the inter-electrode interference by delivering interleaved biphasic pulses to the electrode pairs. During stimulation the channels are updated in a sequential fashion such that only one electrode pair is activated at any one time. The other eight electrode pairs are in a high-impedance state. As shown in Figure 7 the sound is passed through eight bandpass filters and the envelope extracted by full-wave rectification and low-pass filtering. The envelope outputs are compressed to meet the proper dynamic range for electrical stimulation and then used to modulate the interleaved biphasic pulses. The pulses are delivered to the electrodes at a rate up to 833 pulse/second.

Transmission of data and power to the internal circuitry is accomplished through an inductive link with a carrier frequency of 49 MHz. A handshaking routine between the internal and external unit ensures that the internal circuitry is functional. Telemetry from the internal unit provides the means to monitor the internal circuitry. The electrode voltages, active or passive, provide information regarding electrode impedance and electrical field strength respectively. As well, device functionality can be partially monitored through the transmission of internal reference voltages. The telemetry unit sends information via a 6-bit analog to digital converter (ADC) on a carrier of 10.7 MHz.

2.3.4 University of Utah/Ineraid

Over 100 patients have been implanted with the Ineraid cochlear prostheses, initially developed at the University of Utah [Loe90]. It has been approved by the United States Food and Drug Administration as an investigational implant for use at approved centres [Loe90]. The following description of the device was based on two main sources [Ric][CTP90].

Figure 8: University of Utah/Ineraid Signal Processing Block Diagram

The Ineraid design uses six monopolar electrodes, spaced 4 mm apart, inserted along the length of the scala tympani. Two additional electrodes are implanted in the temporalis muscle outside the ear, one of which acts as a ground. The most apical electrode, representing the lowest frequencies, lies 22 mm from the round window. Four to six weeks following the implantation of the device, the four channels offering the best performance are selected as the stimulating channels, based on evaluations of stimulation thresholds, dynamic range, and tonotopicality.

As in the Clarion devices, the Ineraid signal processing scheme is based on a compressed analog (CA) approach (Figure 8). A combination of manual and automatic gain

controls compresses the incoming sound to a dynamic range more suitable for electrical stimulation (10 dB to 20 dB). The signal is filtered into four bandpass channels centered at 500 Hz, 1000 Hz, 2000 Hz, and 3300 Hz. These signals are passed through voltage to current converters and applied simultaneously to the four active implanted electrodes. All eight electrodes are connected to the external unit through a percutaneous connection.

In a University of Iowa study the Utah/Ineraid implant showed slightly better patient performance than the Nucleus system, although the difference was statistically insignificant [GTK+88]. In particular, Ineraid patients had better success in a noisy environment than those fitted with the Nucleus device [GTK+88]. It has been suggested that electrode interactions resulting from the broad current spread of the monopolar electrodes may actually be advantageous in a noisy environment, although the reason for this is unclear [CTP90].

2.3.5 Digisonic

The Digisonic cochlear implant is a multichannel, monopolar device developed by a group at Hôpital Saint-Antoine, Paris [COM+95]. It is based on the earlier Chorimac implant which began development in 1973.

The prosthesis uses 15 monopolar (common ground) platinum-iridium electrodes implanted within the cochlea. The electrodes are designed to provide a fairly large stimulation area to minimize current density and yet are shaped to minimize electrode interactions.

The speech processing strategy uses a 128-point Fast Fourier Transform (FFT) to calculate signal energy at 64 points between 100 Hz and 7800 Hz. The 64 values are then

distributed amongst up to 15 electrode channels. The number of frequency bands analyzed and the mapping of these bands to individual electrodes are flexible.

A sequential stimulation strategy is adopted. Starting at the basal most electrode biphasic current pulses are applied to each electrode. The charge density applied is proportional to the energy in the corresponding frequency band. The width of the stimulating pulses can vary from $5\ \mu\text{s}$ to $310\ \mu\text{s}$, in $5\ \mu\text{s}$ steps. The maximum and minimum current amplitudes are set by the audiologist and the current pulse amplitude can vary within this range. The rate of stimulation can either follow the fundamental formant frequency (F_0) or be set manually to a fixed value between 125 Hz and 400 Hz. This stimulation frequency can be increased up to a maximum of 1 kHz if the number of channels addressed is reduced. Undesirable crosstalk between electrodes can be minimized by sending an opposite charge pulse to electrodes neighbouring the point of stimulation.

Transmission of power and data to the implanted electrodes is accomplished transcutaneously over a 3 MHz carrier. Stimulation information is conveyed using a series of 16 pulses of varying widths. The first pulse width encodes the current amplitude to be applied to the 15 stimulating electrodes. The next 15 pulses describe the pulse duration for each of the 15 electrodes. Stimulation information is updated every 3.3 ms.

2.3.6 Multichannel Urinary Prosthesis

Researchers at the Groupe de Recherche en Appareillage Médical de Sherbrooke (GRAMS), École Polytechnique, Université de Sherbrooke, and McGill University, Québec, have developed an experimental multichannel urinary prosthesis [SDH+92]. However, the system can be modified for use in other neurostimulator applications. The custom receiver-stimulator integrated circuit in particular was designed to be flexible in

order to support a wide range of stimulation algorithms. It is appropriate that such a device be included in this review.

Eight monopolar or four bipolar stimulation channels can be controlled. Transmission of the stimulus control signals to the implanted unit is accomplished using a transcutaneous inductive link with a carrier frequency of 20 MHz. The proposed system is initially intended for experimental use only, and is thus not portable. A microcomputer is used as the external speech processor. The data interface stage converts the microcomputer output to an amplitude modulated (AM) signal for transmission to the implanted circuitry. The receiver recovers the transmitted signal—as well as power for the implanted electronics—and sends it to the custom microprocessor. The system is intended to be flexible such that a wide variety of stimuli can be generated. Implementation of the unit as a cochlear implant could be accomplished by updating the software processing algorithms and some passive elements—including the electrodes—on the implanted circuitry.

The microprocessor is fabricated in a 4 μm CMOS gate array. The block diagram of the circuit is shown in Figure 9. Twenty-four bit command words are received from the external unit and provide the stimulus parameters—the shape, frequency, and amplitude of the stimulus current waveforms—for the control units controlling each of the eight channel. Each control unit generates a biphasic waveform. At the output stage of each channel is a 6-bit digital to analog converter (DAC) providing 64 discrete levels of stimulation. An external resistor limits the maximum current level of each channel. Stimulation frequencies up to 4.5 kHz are available. The chip is comprised of 2000 gates and operates at a clock frequency of 1 kHz, consuming a maximum power of 8 mW.

Figure 9: Multichannel Urinary Prosthesis Microprocessor Block Diagram [SDH+92]

2.4 Summary

To provide maximum benefit to the patient, a cochlear implant should be able to support a wide range of speech processing and electrical stimulation strategies. This flexibility places increased pressure on the hardware. The implanted circuitry must be capable of generating stimulating current pulses of varying amplitude, duration, and waveshape. The frequency and location of stimulation should also be flexible. These features should be software programmable at the external unit. Trauma to the patient can be reduced if the implanted hardware can accommodate changes to the speech processing or stimulation

strategies without additional surgery. Power requirements for the implanted unit should be kept to a minimum for safety as well as practical reasons. As well, the size of the implanted electronics is limited by the area behind the ear.

The above requirements suggest the need for a low power, programmable architecture designed specifically for use in cochlear neuro-stimulation applications. A custom stimulation controller has the advantage that it can be designed to perform the desired tasks in the most efficient manner possible. Minimizing the complexity while optimizing performance of the circuit for the desired application decreases the power and area cost. In the next chapter a custom stimulation controller which can be used in a programmable cochlear implant system is introduced.

Chapter 3

A Proposed Receiver-Stimulator Architecture

3.1 Introduction

As mentioned in Chapter 2, a cochlear implant should have the capability to support a wide variety of speech processing and electrical stimulation strategies. A flexible cochlear implant system design has been proposed by Withey [Wit92]. The block diagram in Figure 10 outlines the main components of this system. In the external unit, a DSP based speech processor is programmed with the desired speech processing strategy to extract relevant information from the incoming speech signal. The processing strategy could be based on one of the existing approaches described in Chapter 2 (CIS, formant extraction, etc.), or it could be a completely new technique. The stimulation information is digitally coded and transmitted through a transdermal electromagnetic link to the implanted electronics. Ten radially bipolar electrodes, implanted along the scala tympani, are driven by D/A converters, which in turn are controlled by a programmable receiver-stimulator. The digitally coded information for each channel is multiplexed over a single channel for serial transmission. The implanted receiver-stimulator stores the stimulation information in memory and executes the stimulation in a program-like fashion until the loaded program is complete. Power for the implanted circuitry is derived from the electromagnetic carrier.

A key element of this system is the programmable implanted receiver-stimulator. This chapter outlines the architecture and operation of a receiver-stimulator ASIC (application specific integrated circuit) for a cochlear implant system designed to support a wide range of stimulus formats. The concept is based on the Multipulse Programmable Stimula-

Figure 10: Proposed Cochlear Implant System [Wit92]

tor (MPS) outlined by Withey [Wit92]. The circuit is designed to accommodate existing speech processing and stimulation strategies and is intended to be flexible enough to adapt to a patient's needs over time.

The processor controls ten bipolar channels, two electrodes per channel. Each electrode can be individually programmed as a current source or a sink, or disabled altogether. Most applications will likely implement the electrodes as ten bipolar pairs, but other configurations such as common ground or perhaps a more exotic cross-channel stimulation technique may be accommodated.

The MPS can support an external unit that processes frames of speech ranging from tenths of a millisecond to seconds in length. In each speech frame the spectral content of the speech signal can be considered constant, if the frame length is short enough. A frame length of 30 ms is typical. However, the spectral content will often change from frame to frame, involving a different subset of electrodes to be activated in each stimulation frame. The MPS is reprogrammed at the beginning of each frame. This allows the system to adjust to the changes in spectral content.

3.2 Design Requirements

3.2.1 Safety

A patient's health and safety is always the number one consideration in the design of a medical prosthesis. Transcutaneous transmission of the electrical stimulation waveforms from the external speech processing unit to the implanted receiver/stimulator avoids the risk of infection surrounding the use of percutaneous leads [McD89][GMW75]. The resulting transmission protocol, however, is slightly more complex, and the implanted electronics can no longer draw power through a direct connection to the external unit.

Any electrical stimulation in living tissue must be extremely well controlled. Excessive charge density over the surface of the electrodes during stimulation can cause electrolysis in the fluid surrounding the electrodes. Damage to the surrounding tissue can result from the gas produced as well as from pH changes near the electrodes [BT77]. Also, the charge pulses must be biphasic in nature to avoid a net DC current, which will release toxic oxidation products in biological tissue [BT77].

It has been shown that two or more electrodes activated simultaneously show strong coupling effects due to current spread resulting from non-ideal electrical field generation at the electrodes and the conductive saline solution in which the electrodes are immersed [GC87]. This coupling is often difficult to predict, and the neural populations affected may not be what was anticipated. The primary result is that the desired perception of sound is not achieved, but the electrode interactions can also produce stimulus levels higher than desired [McD89]. It is necessary to ensure that the stimulus does not reach dangerous levels.

The implanted electronics must be hermetically sealed in a biocompatible case to reduce the risk of infection or contamination of the surrounding tissue and to protect the circuitry itself. The entire package must fit in a cavity created in the mastoid process behind the ear. Typically, a maximum volume of $3.5 \times 3.5 \times 0.75$ cm is available [GMW75].

3.2.2 Power

The power consumption of a cochlear implant must be kept low to allow for long term battery operation. Morris and Barszczewski [MB89] have stated that if power con-

sumption of the entire cochlear implant system is less than 1 W, three rechargeable lithium “AAA” cells would be sufficient for continuous daily use with overnight recharging.

3.2.3 Flexibility

As discussed earlier, a cochlear implant system should be flexible enough to tailor its operation to the needs of each individual patient. This flexibility would allow for changes in the speech processing and electrical stimulation strategies to be made without removing the implant, which requires further surgery. Predicting a patient’s success with a particular cochlear implant system is often not possible before the patient has been fitted with the device. By supporting modifications after surgery, the cochlear implant provides a speech processing and electrical stimulation strategy optimized for each specific patient. As well, future improvements in speech processing and electrical stimulation techniques can be incorporated into the device at a later time. Recent developments such as the effect of asymmetric biphasic pulses on the probability of neural activation [WCM89], and the concept of preconditioning—using a long, low amplitude pulse to randomize the firing state of the auditory nerve fibers and achieve a more natural neural response pattern [Wit92]—have shown that a great deal about the electrical stimulation of the auditory nerve and the intelligibility of these “sounds” as they reach the brain is still being discovered.

Flexibility implies programmability. Different degrees of programmability already exist in a number of cochlear implant systems, but it is generally not possible to use an arbitrary electrical stimulation waveform in any current implant. For maximum flexibility, a cochlear implant should provide the means to explicitly define all aspects of the stimulating waveform; that is, the duration and amplitude of each phase of the electrical pulse, as well as the duration between each biphasic pulse, must be programmable. In addition, a

flexible stimulation strategy would allow the user to control the individual electrodes in a way to accommodate location and method of stimulation along the basilar membrane. Some degree of freedom can be accomplished by controlling the direction of current flow in the electrodes and allowing individual electrodes to be enabled or disabled, even the individual electrodes in the bipolar pairs. By adjusting the stimulation strategy as such, some compensation can be made for the variations in the damage to the hair cells along the basilar membrane between individual patients, most of which is not known until after the device has been implanted.

Trade-offs exists between a device's functionality and programmability and its resulting power consumption, speed of operation, and size. Some design constraints must be set at the expense of some circuit flexibility. Loeb has remarked that ten channels should provide sufficient resolution for speech recognition, stating that if recognition is not achieved with ten channels, little improvement will be had by increasing the number of channels [Loe85a]. The transcutaneous data transmission rate required for ten channels of information, each carrying a complete description of the stimulating waveform, would be prohibitively high [Wit92]. Some sacrifices to the flexibility of the device must be made. It has been noted that for a single frame of speech—on average about 30 ms long—most stimuli are composed of repetitive subsections [Wit92]. Sending a description of the composition of each subsection accompanied by a repeat directive can reduce the required bandwidth considerably. Some flexibility in the ability to generate an arbitrary waveform is sacrificed, as the implant is now more suited to sourcing a repetitive pulse train.

3.3 Desired Stimulus

A wide range of stimulation waveforms can be generated by explicitly defining the stimulus pulse amplitude, duration, and polarity. Some examples of a number of waveforms that could be described in this manner are shown in Figure 11. This manner of pulse description can be used to provide simple biphasic pulsatile stimulation (Figure 11 [a-d]), with more complex stimulation waveforms also supported (Figure 11 [e]).

A concern with this method of pulse definition is that potentially dangerous stimulation waveforms can be programmed if the MPS does not confirm whether the programmed waveform is biphasic or not. Because the MPS can support such a wide range of stimulus waveforms, it is up to the processing algorithm to ensure that the stimulation waveform definition falls within the safety guidelines discussed earlier.

3.4 Data Format

A small set of 16-bit instructions describes the pulse programming of the MPS as well as the data transmission format. The MPS instructions are based on Withey's MPS design [Wit92].

3.4.1 Stimulation Frame

The cochlear implant system processes the incoming sound in frames of finite length such that the spectral content within a single frame remains relatively constant. A typical frame would be of the order of 30 ms, but in the MPS system the frame length is flexible and can vary from frame to frame.

The speech frame is processed by the external unit, and it is the processing algorithm that defines which channels will be active and what the stimulating waveform will

Figure 11: Stimulation Waveform Examples

be for each channel. The waveform is broken up into a series of pulses, as discussed in the previous section. The pulse descriptions are transmitted to the implanted processor in a programming mode and then executed in a stimulation mode. Once finished the MPS waits for further instructions to be downloaded.

3.4.2 Pulse Description

The MPS stimulation instruction for each pulse is comprised of two 16-bit words: a data word and a control word. The data word uses eight bits to describe the pulse amplitude and eight bits to describe the pulse duration, as shown in Figure 13. The pulse amplitude that can be specified ranges from zero to 2.55 mA with a granularity of 10 μ A per bit. Similarly, the pulse duration can be set between 10 μ s and 2.56 ms, in 10 μ s steps, with longer pulses available by concatenating a series of two or more individual pulses.

Associated with every data word is a 16-bit control word. The control word has fields to describe pulse polarity and contains information to control pulse repetition patterns. The most significant four bits in the control word are allocated for current direction control, two bits for each electrode in the bipolar pair (labeled P and N). The first bit in each pair indicates whether the electrode is to be driven (ON) or placed in a high impedance state (OFF). Individual electrodes can be actively driven or safely deactivated to eliminate the risk of dangerous leakage current. The second bit controls the direction of current through the electrode, i.e., whether it acts to source or sink current. To describe

Figure 12: Instruction Pulse Data Word

Figure 13: Instruction Pulse Control Word

one of the simple biphasic pulses shown in Figure 11, the current direction control bits in the first control word would be 1110, a positive phase pulse with current sourced in the P electrode returning to the N electrode. The second, negative pulse would be described with current direction control bits set to 1011.

The flexibility to independently control the current in each electrode of the bipolar pair also allows more complex stimulation schemes. Cross channel stimulation, in which current sourced at one location along the implanted array returns at another location, could be executed by setting the P and N electrodes of channel one to source current, and setting the bipolar pair of channel three to sink current, for example. The electrode geometry can thus be optimized post-operatively for each patient, even from one stimulation frame to the next.

Bits nine and ten of the control word, the opcode, along with the repeat count (bits 7-0) can be used to indicate that a series of pulses is to be repeated. Setting bit ten high indicates that this control word and its associated data word are the first in a series of pulses to be repeated $N+1$ times, where N is the value indicated in the repeat count. Setting bit nine of the control word high flags the last word in a repeat sequence. A single stimulation frame can have more than one repeat sequence, but nested repeats are not supported by the MPS.

Bit eight of the control word is the LAST INSTRUCTION indicator. This bit, normally a one, is set to zero for the last instruction of a channel's stimulation frame. Following the execution of this pulse—providing that all repeat loops have been fully executed—the MPS will disable the channel until the next stimulation frame has been initiated.

3.4.3 Transmission Protocol

In order to transmit the stimulation description data from the external processor to the MPS a transmission protocol must be defined. The transmission protocol is based on three commands: RESET, LOAD, and START STIMULATION. The RESET and START STIMULATION commands are single word commands, while the LOAD command is used to download a block of stimulation instructions.

The command format is based on a 16-bit word. The structure for a single word command (RESET, START STIMULATION) is slightly different than that of the LOAD command. The two command formats are shown in Figure 14 and Figure 15, respectively. The received word is identified as a command by the four bit field of bits eight to eleven. The command is defined by the opcode in bits six and seven: 00 = RESET.

Figure 14: Single word command

Figure 15: Block LOAD command

01 = START STIMULATION, 10 = LOAD. The RESET command is used to reset all of the programmed registers in the MPS and restart the programming process. The START STIMULATION command is sent to the MPS after all the stimulation information has been transmitted and instructs the MPS to enter stimulation mode. An even parity bit is included in the single word command to help detect transmission errors.

LOAD is sent to the MPS to indicate that a block of stimulation pulse description words (control and data) are to be transmitted. The format for the LOAD command is shown in Figure 15. Included in the LOAD command is a four bit field (bits 12 to 15) that indicates which channel (from zero (0000) to nine (1001)) the set of instructions is to be applied. The six bit COUNT field (bits 0 to 5) indicates the number of words to be loaded for this channel.

Immediately following the LOAD command, the pulse description word pairs for the stimulation frame are transmitted, alternating between the control and data words until all word pairs—up to a maximum of ten for the channel—have been downloaded. Following the block of pulse description words, a checksum word is transmitted to help detect transmission errors. The checksum is generated by the external processor by consecutively XORing all words in the block command, including the LOAD command word. The MPS performs a similar operation at the receiving end. The MPS compares the calculated checksum to that received from the external processor. If the two do not agree, the received block for that channel is not processed during stimulation.

Thus, a typical transmission sequence for a stimulation frame would begin with a RESET command to clear the MPS registers (not required, but recommended), followed by one to ten LOAD blocks, depending on how many channels are to be programmed. A START STIMULATION command is transmitted to end the programming sequence and put the MPS into stimulation mode.

* with LAST INSTRUCTION flag set

Figure 16: Block download structure

3.5 Processor Architecture

The following sections define an architecture that supports the data format described earlier. The MPS must be able to receive the transmitted data from the external processor, store the pulse description instructions in memory, and generate current pulses as described by the instruction control and data words. It must be able to simultaneously control the pulse generation for ten independent channels while minimizing the amount of hardware required. By defining two distinct modes of operation—a programming mode and a stimulation mode—much of the circuitry can be common to both modes, and the size and power consumption of the ASIC can be minimized.

The following sections describe the MPS architecture in detail and provide some rationale for the architectural decisions being made. As we will see, the MPS is broken up into a number of different blocks in a top-down hierarchical design approach. The reasons for this are two-fold. The first is that partitioning the design simplifies the design process. It is easier to define the functionality of smaller circuits, and then define the interaction of the different blocks. The next step in the design process—elaborating the block diagrams and circuit description into HDL code—is also simplified. Simulating and debugging is much more easily performed on small circuits.

However, it is not simply ease of design that encourages a hierarchical approach. As will be discussed in the next chapter, partitioning of the circuit can play a critical role during automated synthesis. Most notable are the partitioning of sequential (clocked) and combinational elements, and the choice of state machine architecture in the design. Additionally there are the design targets for the circuit to keep in mind. Is the intent to minimize die size, or is low power the primary concern? Architectural decisions made at this stage in the design should simplify the design process and allow for efficient design synthesis while meeting the original design goals.

3.5.1 Block Diagram

The block diagram in Figure 17 outlines the top level structure of the MPS. The MPS consists of ten independent stimulation channel drivers, a block of memory to store stimulation instructions, and a microcontroller core comprised of control, data, and logic units. In programming mode the microcontroller core receives the commands from the external processor and stores the stimulation information in the on-chip RAM. In stimulation mode the core is responsible for retrieving the pulse descriptions from memory and routing the stimulation commands to the appropriate channel drivers at the desired time.

Figure 17: MPS Top Level Block Diagram

The core monitors the status of each channel driver and updates the pulse information as each pulse is executed. A logic unit within the core provides status signals to the control unit based on the current data unit register values.

The ten channel drivers control the pulse amplitude and direction as described by the instruction control and data words for each channel (See section 3.4.2 “Pulse Description”.) The control signals from the micro core are time multiplexed amongst the ten channel drivers. Data (i.e., the pulse amplitude and direction descriptions) is transferred to each driver through a common data bus.

3.5.2 Data Unit

The sequential elements of the micro core—registers, counters, and flip-flops—are grouped together in the data unit along with some combinational logic to support data transfers. A block diagram of the data unit is shown in Figure 18, and a register summary is given in Table 1 on page 54. All registers are synchronously clocked; thus all data transfers occur at a single clock edge. Each register has a number of control signal inputs direct the flow of data to, from, and within the Data Unit. Each These control signals (RESET, LOAD, INCrement, etc.) come from the Control Unit and change on the opposite clock edge to maximize setup and hold time. Most of the Data Unit register values are brought out to the Logic Unit in order to monitor the status of the MPS during programming and stimulation.

Data Flow.

During programming, all data from the external processor arrives at the receive register (REC) via the clock/data recovery circuitry. The 16-bit input is clocked into the REC when the word received (WR) signal is high. REC is monitored by the Logic Unit to

Figure 18: Data Unit Block Diagram

verify that the word received is a valid command. The stimulation pulse control and data words are routed to the Pulse Control Register (PCR) and the Pulse Data Register (PDR) respectively. The two registers together provide a bidirectional 32-bit interface to the on-chip RAM. The value in PCR is also brought to the Logic unit.

A four bit up counter contains the Channel address (CH). CH can be loaded from the top four bits of REC to select the channel to be programmed during a block download. It is also used during stimulation to indicate which stimulation channel is currently being serviced. In both cases CH provides the top four bits of the memory address. The Channel address is routed to the mux and demux of the Channel Status Register array and the Stimulation Status bits to direct the routing of data. CH is a four bit up counter, but it wraps at binary 1001 (i.e., channel 9), providing access to ten stimulation channels.

A second four bit counter contains the Instruction address (INS). INS addresses the 16 stimulation instructions (maximum) for a channel and thus represents the bottom four bits of the memory address. During programming INS is incremented after each control and data word pair have been received, and during stimulation INS is incremented as each pulse is executed for a channel. INS has a bidirectional interface with the Channel Status Register (CSR) array for temporary storage, and with the BRANCH register.

The BRANCH address register provides the means to support looped stimulation instructions by storing the starting Instruction address of a repeat loop. As the pulse description instructions for a channel are read from memory the Pulse Control Word is examined for repeat directives. When the start of a repeat loop is indicated, the value in INS is loaded into BRANCH. When the end of the loop is reached, the BRANCH value is loaded into INS to begin the loop anew, providing that the loop count has not yet reached zero. BRANCH also interfaces with the CSR array.

COUNT is an eight-bit up counter. The counter is used to track the word count during block downloads in programming mode, and it tracks the number of times through a repeat cycle in a stimulation repeat loop. COUNT can be loaded from the Receive register bits zero to six, and it also stores data to and receives data from the CSR array.

Since the micro core is processing information for one channel at a time, an array of Channel Status Registers (CSR) is used to store information about a channel while it is not being serviced. Specifically, the CSR stores the Instruction address (INS), the Branch address (BRANCH), and the loop repeat count (COUNT) for each channel. After the

Figure 19: Channel Status Register

pulse description is read from memory and transferred to the proper channel driver, INS, BRANCH, and COUNT are saved to the CSR array. The CSR interface mux/demux uses the current Channel address to select the proper Channel Status Register. In this manner the micro core can time multiplex through the ten stimulation channels. The hardware required for the micro core is reduced at the expense of increased Control Unit complexity and reduced processor speed.

The STIM_STAT register is a collection of ten SR flip-flops with access to the SET and RESET controls to each bit. The stimulation status for each channel is indicated by the corresponding bit in STIM_STAT. When pulse description data for a channel is

loaded and stored in memory, the STIM_STAT bit is set to indicate that valid stimulation data is available for that channel. During stimulation, only channels with a STIM_STAT bit set high are serviced. When the last pulse description for a channel is executed, the corresponding STIM_STAT bit is reset. When all the stimulation instructions have been processed all ten bits will have been reset, notifying the Control Unit that stimulation has ended and the MPS should enter programming mode.

A single flip-flop is used to store the stimulation enable flag, STIME. This signal is used to synchronize the start of the ten channel drivers. It is set high by the Control Unit once all stimulation information has been received and the channel drivers have been properly programmed.

Table 1: Data Unit Register Descriptions

Symbol	Size/Type	Description/Use
REC	16-bit register	Input receive register: <ul style="list-style-type: none"> receives data from clock/data recovery circuit
ACC	16-bit accumulator	Accumulator: <ul style="list-style-type: none"> stores checksum during data download.
PCR	16-bit register	Pulse Control register: <ul style="list-style-type: none"> programming commands stimulation pulse direction, repeat instructions interfaces with memory
PDR	16-bit register	Pulse Data register <ul style="list-style-type: none"> stimulation pulse amplitude and duration interfaces with memory
COUNT	8-bit down counter	Down counter: <ul style="list-style-type: none"> tracks word count during programming tracks pulse repeat count during stimulation

Table 1: Data Unit Register Descriptions (Continued)

Symbol	Size/Type	Description/Use
CH	4-bit up counter (wrap at 9)	Channel address register: <ul style="list-style-type: none"> • tracks current channel being serviced during programming or stimulation • top nibble of memory address
INS	4-bit register	Instruction address register <ul style="list-style-type: none"> • tracks current instruction in memory • bottom nibble of memory address
BRANCH	4-bit register	Branch address register: <ul style="list-style-type: none"> • saves the instruction address to branch to in a pulse repeat loop
CSR(0.9)	16-bits x 10 register array	Channel status registers: <ul style="list-style-type: none"> • saves channel status (instruction/branch address, repeat count) as other channels are serviced
STIM_STAT	10-bit register with 10 SET/RST inputs	Channel stimulation status flags: <ul style="list-style-type: none"> • indicates if channel has been programmed and if stimulation is complete

3.5.3 Memory

Memory on the MPS consists of a simple block of single port, synchronous dynamic RAM, 32-bits wide by 160 words. 16 words of memory are assigned for each channel. Thus, the MPS is limited to a maximum of 16 different pulse descriptions in a stimulation frame.

Figure 20: MPS Memory Partitioning

3.5.4 Logic Unit

Within the Logic Unit is most of the combinational logic. The bitwise XOR of the receive register and the accumulator and some multiplexing/demultiplexing are performed in the Data Unit. The inputs to the Logic Unit are a number of register values from the Data Unit plus some status signals from the Channel Drivers. The output status signals are used by the Control Unit to direct its state transitions. Table 2 outlines the status signals generated by the logic unit and their purpose.

Table 2: MPS Combinational Logic

Status Signal	Logic High When:	Purpose
COMMAND DETECTED	REC(11:8)='1110'	In programming mode, indicates that the received word is a command word.
RESET	REC(7:6)='00'	In programming mode, indicates that a RESET command has been received.
START STIMULATION	REC(7:6)='01'	In programming mode, indicates that the START STIMULATION command has been received.
LOAD	REC(7:6)='10'	In programming mode, indicates that a LOAD command has been received.
START REPEAT	PCR(10)='1'	In stimulation mode, indicates that the pulse control word is the beginning of a repeat block.
END REPEAT	PCR(9)='1'	In stimulation mode, indicates that the pulse control word is the end of a repeat block.

Table 2: MPS Combinational Logic (Continued)

Status Signal	Logic High When:	Purpose
LAST INSTRUCTION	PCR(8)='0'	In stimulation mode, indicates that this is the last pulse control word for this channel.
LOOP-IN-PROGRESS	COUNT≠'00000000'	If the counter has not reached zero then the pulse repeat sequence has not finished.
COUNT INVALID	COUNT(0)='1'	Used to indicate, during block downloads, if a valid (i.e., even) word count has been sent.
LAST CHANNEL	CHANNEL='1001'	Indicates that channel driver number nine is currently being serviced.
END STIMULATION	DREQ='1' for all channel drivers AND STIM_STAT='0000000000'	If the STIM_STAT bits for all ten channels have been reset, and all channel drivers have finished stimulating (DREQ='0'), then the stimulation frame is finished.
COUNT ZERO	COUNT='00000000'	Used in a number of states to indicate that the counter is zero.
ACC ZERO	ACC='00000000 00000000'	During programming, if the accumulator is zero following a block download the checksum has been verified.

3.5.5 Control Unit

The control unit is responsible for directing data transfers within the MPS. It generates the load, increment and decrement, set and reset signals for the data unit registers, channel drivers, and memory. The core of the control unit consists of a finite state machine (FSM), the basic structure of which is shown in Figure 21. The purpose of an FSM is to generate a set of control signals based on the current state of the machine. The current state vector is stored in state memory, i.e., a group of flip-flops. The transition from one state to the next is based on the current state and a set of input status signals. Output control signals are defined for each state by hardwired combinational logic at the output of the state memory.

Two approaches to the design are available, as shown in Figure 21. A Moore device generates output signals based only on the current state of the device. A Mealy machine uses one or more input values in addition to the state vector to generate output signals. A Mealy device will generally requires fewer states to run through an algorithm, and is thus usually more efficient, i.e., faster. A Moore approach may not be as efficient but is often simpler to implement and easier to design and modify.

The Control Unit of the MPS provides the control signals to the Data Unit registers and the Channel Drivers. The state machine transitions are based on the Logic Unit status signals and the existing state. Please refer to section 3.6 “Operation” on page 68 for the state diagrams and detailed description of the Control Unit operation.

3.5.6 Channel Driver

The MPS core dispenses pulse description and stimulation control data to the ten Channel Drivers. Each Channel Driver is essentially a finite state machine consisting

Figure 21: Finite State Machine Architecture

of a small control unit and set of data registers. A FSM dedicated to the control of each bipolar electrode pair provides the flexibility to independently control the stimulation waveform for each electrode pair.

Data Registers

As shown in Figure 22 two registers in the Channel Driver control the electrode drivers. **AMP** is an eight-bit register that receives the electrode pulse amplitude from the core's Pulse Data Register. The **AMP** output is fed into the external DAC controlling the current amplitude through the electrode pair.

The **DIR** register contains the four-bit current direction control as received from the Pulse Control Register. **DIR** drives the four control signals to the electrode pair: $P_ON/\overline{OFF}$, $P_SOURCE/\overline{SINK}$, $N_ON/\overline{OFF}$, and $N_SOURCE/\overline{SINK}$.

Figure 22: Channel Driver Block Diagram

The pulse width is initially stored in DUR, a 12-bit down counter. This value is decremented until it reaches zero, at which point the next pulse description (AMP, DIR, and DUR) is loaded into the channel driver registers. The top eight-bits of DUR are downloaded from the Pulse Data Register in the MPS core, via a buffer register. The bottom four bits are initially stuffed with four ones. A pulse width of zero loaded from the MPS core corresponds to the fundamental pulse width of 16 clock cycles. Essentially a four-bit shift left, this operation provides a stimulation pulse granularity of 16-times the MPS core clock period. In other words, it allows the core to process 16-times faster than the stimulation pulses. This is necessary in order for the MPS to be able to fully service ten individual channel drivers.

DUR, AMP, and DIR are each buffered from the MPS databus by an additional register. The buffer registers hold the next set of pulse description instructions for the channel driver. Once a pulse is completed (i.e., DUR= x000) the channel driver control unit is able to immediately load the next pulse description into its registers.

In addition to the primary registers and the three buffers, there is one flip-flop to store the data request (DREQ) flag and one flip flop for the registered DAC enable signal (DAC_EN_reg). DREQ signals to the MPS core that the channel driver is ready to accept new stimulation information and is set high by the channel driver control unit when the buffered data is loaded into the primary registers (DUR, AMP, and DIR). DREQ is reset by the MPS core when new data is loaded into the channel driver buffers. In this manner, the DREQ register facilitates a simple handshaking protocol between the MPS core and the channel driver.

The second flip-flop is used to re-time the control signal DAC_EN with the other registered data before sending it to the external DAC. This ensures that the DAC_EN is synchronized with the amplitude and current direction controls.

Control Signals

The channel driver control unit generates three control signals for the channel driver block. DAC_EN (DAC enable) is an output to the external digital-to-analog converter. A high signal enables the DAC output driving the stimulation electrodes. A low signal tri-states the output of the DAC. The DAC_EN signal, retimed by the rising edge of the MPS core clock, is used to synchronize the stimulation waveform with the MPS clock, and provides a method of safely disabling the circuitry driving the electrodes.

The control signal wr_REGS is used to load stimulation information from the channel driver buffers into the primary registers. wr_REGS goes high when a pulse has been completed and a new pulse description is available in the buffers.

The last control signal, dec_DUR, controls the channel driver pulse duration counter, DUR. When dec_DUR is high, the value in DUR is decremented on the rising edge of the core clock. dec_DUR is asserted high when the control unit wishes to proceed with the generation of an output pulse.

Status Signals

The inputs to the control unit are four status signals. STIME comes from the MPS core. It is used to signal to each of the ten channel drivers that programming has been successful and that stimulation may begin.

STIM_STAT is another status signal from the MPS core. It is the current value represented by a particular channel driver's bit in the STIM_STAT register. A high signal

indicates to that channel driver that valid data is available (as stored in memory) for this channel. If the particular channel has not been programmed, or all stimulation data has been processed, the bit is set low by the MPS core.

The third status signal is the data request line, DREQ, from the channel driver itself. In addition to its role as a signal to the MPS core for more data, DREQ is used by the channel driver control unit to indicate when valid data is available in the buffer registers. It is used as a data acknowledge signal ($DREQ = \overline{DACK}$). DREQ='0' acknowledges that valid data has been loaded into the buffers.

The final status signal is DUR_ZERO. DUR_ZERO goes high when the duration counter in the channel driver reaches zero, indicating to the channel driver control unit that the current pulse has been completed.

Control Unit

The channel driver control unit is implemented as a Mealy type finite state machine. The reason for this architectural decision is that the state diagram and transition logic can be greatly simplified by using input status signals in the logic that generates the output control signals. The inputs (status signals) and outputs (control signals) are as described in the previous section. The state diagram of the channel driver control unit illustrates the transitions between states and the logical conditions that control them. The following description is based upon the channel driver state diagram shown in Figure 23. The timing diagram shown in Figure 24 is also used to help illustrate the state transitions.

The ten channel drivers are all initialized in the STOP state upon power-up or following a hardware reset. The channel drivers remain in the STOP state when the MPS is in programming mode. In this state all of the control signals (DAC_EN, wr_REGS, and

dec_DUR) are set to zero, effectively disabling the driver. The FSM will remain in the STOP state until valid data appears at the buffers (DREQ=0). If the primary channel driver registers are empty and valid data has been loaded into the buffers (DUR_ZERO=1 AND - DREQ=0) then the FSM will enter the LOAD state. In the LOAD state the FSM pulls wr_REGS high to transfer data from the buffers to the primary registers, setting DREQ=1 in the process. If STIME has been set high by the MPS core at this time, the channel driver FSM will progress into the state RUN and begin generating a pulse. Otherwise, the FSM goes back to STOP to wait for stimulation mode to begin.

The MPS core signals to the channel drivers that stimulation may begin by pulling STIME high. If data has been loaded into the channel driver (DUR_ZERO=0) the channel driver will proceed with pulse generation by going into the START state in which the DAC_EN signal is set high. The FSM then goes directly to RUN. It is in this state that the duration down-counter is enabled by setting dec_DUR high.

In the RUN state the duration counter is decremented until it reaches zero, setting DUR_ZERO high. If at this time there is data available in the buffers, the channel driver will enter the LOAD state. The next pulse description is loaded into the primary registers and applied to the electrodes. The control unit returns to the RUN state as long as STIME is still high. The control signals DAC_EN and dec_DUR are kept at their previous values to maintain uninterrupted stimulation.

If a pulse has been completed and no data has been loaded into the buffers, i.e. at the end of a stimulation frame, the FSM will return to the STOP state and wait for new data. Also, if at any time STIME goes low, the FSM (in fact, all ten channel driver state machines) automatically enters STOP as the next state.

Figure 23: Channel Driver State Diagram

Table 3: Channel Driver State Machine Output

State	State Machine Outputs		
	DAC_EN	wr_REGS	dec_DUR
STOP	0	0	0
LOAD	0	1	0
START	1	0	0
RUN	1	0	1

- ① MPS core loads data into buffers
- ② MPS sets STIME high after programming all registers
- ③ MPS core loads second set of data into buffers
- ④ End of first pulse detected as DUR goes to zero
- ⑤ Second pulse loaded into primary channel driver registers
- ⑥ End of second pulse detected

Figure 24: Channel Driver Timing Diagram Example

3.6 Operation

Although the MPS control unit is comprised of a single finite state machine, for design purposes the operation of the circuit is broken into two modes of operation: programming mode and stimulation mode. The MPS starts in programming mode on power up. In this mode the MPS is initialized by clearing the registers and channel drivers. The MPS then waits for programming information to be received from the external processor via the clock and data recovery circuitry. Stimulation information for the frame is stored in memory one channel at a time. Parity and checksum calculations are performed to verify the integrity of the transmitted information. The MPS exits programming mode when the stimulation command is received from the external processor.

In stimulation mode the processor begins by loading the first pulse descriptions into the active channel drivers. When the channel drivers have been initialized stimulation is initiated. The processor then sequentially polls each channel, monitoring the status of each channel driver. When a channel driver completes a pulse, a new one is loaded from memory by the processor core. When all pulse descriptions for a frame have been processed by a channel, that driver is disabled. When all ten channels have completed stimulation the MPS returns to programming mode and waits for the signal to load the next frame.

The following sections describe in detail the operation and state transitions in each mode of operation. A detailed state transition diagram accompanies each description.

3.6.1 Programming Mode

Programming mode is entered upon power-up of the MPS and at the completion of a stimulation cycle. The MPS is initialized (ProgInit) by clearing the channel status bit

register `STIM_STAT` and resetting the `STIME` flag. The MPS advances to the next state (`WaitForCommand`) and waits for a valid command word (`WR=1` and `COMMAND=1`) from the external processor. As outlined in section 3.4.3 “Transmission Protocol” on page 44, the MPS can receive one of three commands from the external processor: `RESET`, `STIM`, and `LOAD`. In the `CommandDecode` state the MPS verifies that one of the three valid command words has been received; otherwise, the FSM goes back to `WaitforCommand`.

If a `RESET` command has been received, the control unit is sent back to the `Prog-Init` state. `STIM`, which is usually sent after all programming has been completed, directs the MPS to enter stimulation mode.

When a valid `LOAD` command is received, the MPS enters a loop in which a control word (`WaitForPCR/LoadPCR`) and data word (`WaitForPDR/LoadPDR`) pair are received and stored in memory (`WriteMemory`). As each word is received it is XORed with the value in the accumulator to generate the block checksum. Additionally, the counter is decremented for each received word. If the counter has not reached zero, the instruction address is incremented (`IncInstructionAddress`), and the loop continues.

When the counter has reached zero, indicating that the stimulation information for this channel has been fully transmitted, the MPS waits for the checksum to be transmitted from the external unit (`WaitForChecksum`). It is compared with the generated checksum by XORing it with the value in the accumulator (`VerifyChecksum`). If the result is zero, the transmitted block has been verified, and the stimulation status bit for the current channel is set (`SetStim`). If not, a transmission error may have occurred. The stimulation status bit is not set, and the stimulation information will not be executed for that channel.

Figure 25: Control Unit State Diagram: Programming Mode

After the checksum verification, the control unit returns to the WaitForCommand state. Eventually, all channels for this stimulation frame will be programmed, and the control unit will receive the STIM command. At this time control will be passed to the second half of the control unit, stimulation mode.

3.6.2 Stimulation Mode

When the MPS has received the STIM command from the external processor, the control unit enters the StimInit state. In this state the control unit initializes the MPS for stimulation by first ensuring that the stimulation enable bit, STIME, is set low. The channel address is initialized to channel zero, and the ten channel status registers are cleared. After the MPS has been initialized for stimulation, the FSM enters the stimulation mode control loop. In this loop the control unit monitors the status of each stimulating channel, loading a new stimulation pulse when necessary, and updating the channel status as each stimulation pulse is completed. In the TestChannel state, the control unit examines the DREQ and STIM_STAT flags for the current channel. If the channel driver is requesting data (DREQ=1) and valid data exists in memory for this channel (STIM_STAT=1) the control unit will proceed to LoadStatus and begin loading the next pulse into the channel driver. If stimulation has been completed for all channels (DONE=1) then the control unit returns to programming mode.

If neither of these two conditions exist and the channel driver is currently executing a pulse, then the channel address will be incremented (IncChannelAddress) and the next channel tested. When all ten channels have been assessed for the first time the STIME flag will be set (StartStim), starting the stimulation pulses on the next clock edge.

While the channel drivers are running, the control unit alternates between the TestChannel and IncChannelAddress states. When it gets to a channel requesting data the LoadStatus state is entered. In this state the INS, BRANCH, and COUNT registers are loaded from the channel status registers. This information is used to indicate the next memory address from which to read the next pulse, and the repeat count and repeat branch address if the channel is currently repeating a stimulation pulse.

In the next state (ReadMemory) the next stimulation word is read from memory into PCR and PDR. The current direction control nibble is sent to the channel driver in state LoadDIR. If the other control information in PCR indicates the start of a new loop then the control unit will advance to the StartRepeat state. In StartRepeat the repeat count is initialized from the value in PCR, and BRANCH is loaded from INS to indicate address to branch to at the beginning of the loop.

The pulse amplitude and duration are then transferred from the PDR register to the channel driver in LoadAmpDur. If this is the last instruction for the channel—indicated by the LAST_INS flag in PCR—and all stimulation loops have been completed, then the control unit advances to DisableChannel and resets the stimulation status flag for this channel.

If this is not the last instruction for the channel, the control unit will either increment the instruction address (IncInstructionAddress) or load INS from the BRANCH address (GoToBranch) if the channel is currently in a loop. After this, the MPS will save the channel status—INS, BRANCH, and COUNT—to the channel status register (SaveChannelStatus).

After loading the pulse description into the channel driver the control unit will then increment the channel address and return to polling the channels, alternating between the

TestChannel and IncChannelAddress states. Eventually, all of the channel drivers will have exhausted their supply of pulse descriptions, the ten channel status bits will all have been reset (DONE=1), and the control unit will return to programming mode.

Figure 26: Control Unit State Diagram: Stimulation Mode

Chapter 4

Circuit Design and Synthesis

4.1 Design Synthesis

When describing a digital system there exists an abstraction hierarchy which is often used by designers. This hierarchy provides the means to represent a design in varying levels of detail. Ideally, the designer wants to provide enough detail to accurately describe the circuit, but working at a level with excessive detail can make the design description excessively complex and make the design process lengthy and expensive.

Armstrong and Gray [AG93] have a design hierarchy based on six levels of detail. These are shown in Table 3, “Design Description Hierarchy [AG93],”. The level of detail increases as one moves down the table. Additionally, at each level a model can be described in the behavioural or the structural domain. In the structural domain, a component is described as an interconnection of more primitive components. In the behavioural domain a component is described by its input/output response, i.e., its actual behaviour, with no regard to its internal physical composition.

The lowest level in the design hierarchy is the layout. At this level the design is described by a set of geometrical shapes that represent actual physical entities on silicon: areas of diffusion, polysilicon, and metal. A design at this level can be directly fabricated to create an actual circuit. As the design is described only by the geometrical shapes on silicon, description of the component’s behaviour at this level is governed by the laws of electro-magnetics (Maxwell’s equations) and solid state physics.

Table 3: Design Description Hierarchy [AG93]

Level of Detail	Behavioural Domain Representation	Structural Domain Primitives
System	Performance specifications (English)	Computer, disk unit, radar
Chip	Algorithm	Microprocessor, RAM, UART, interrupt controller
Register	Data flow	Register, ALU, counter, MUX
Gate	Boolean equations	AND, OR, NOT, XOR, flip-flop
Circuit	Differential equations	Transistor, diode, R, L, C
Layout (silicon)	Field equations	Geometrical shapes

Moving up one level is the circuit level description. At this level the design is represented by the traditional electrical circuit elements: resistors, capacitors, transistors, diodes. The circuit functionality can be described using differential equations governing voltage and current levels over time.

The next level up is the gate level. The structural primitives at this level are the elements typically used in digital logic design: logic gates such as AND, OR, NOT, XOR, and storage elements like latches and flip-flops. The operation of these elements, and the behavioural description of a design at this level, are governed by the rules of Boolean logic.

Next is the register level or register transfer level (RTL). At this level the design is composed of functional blocks such as registers, counters, arithmetic logic units (ALU), multiplexers, or memory elements. VLSI macros are often included in a design at this level. Functionality of a design modeled at this level is governed by the flow of binary data, and is usually represented by state diagrams or truth tables.

One level above the register level is the chip level. Now the designer is describing fairly large blocks of logic which one would typically see on a single integrated circuit: microprocessor, memory, communication controllers. The behaviour of a chip level model is described by a number of algorithms controlling the transformation of the input and output signals. A description at this level would not have any reference to the internal workings of the design (i.e., registers, logic).

Finally, at the top is the system level model. At this level one can model a fairly complex structure, one which often interfaces with the real world. A computer, a disk drive controller, or a RADAR unit are a few examples of structural primitives which could be modeled at this level. The behavioural representation of a device modeled at this level is governed by a set of performance specifications such as the processing speed of a computer in MIPS, or the bandwidth of a drive controller. Because the design often interfaces with real world inputs and outputs, the designer will often incorporate higher level data types—such as frequency or temperature—at this level of modelling.

A design can be modelled at any one of these levels. The circuit can also be described using a combination of two or more levels. For example, a microprocessor can be modelled using primarily gate level primitives but may also include a chip level embedded RAM block. Eventually, if a design is to be implemented the circuit description must somehow be translated down to a level that can be physically fabricated, i.e., the silicon layout level. This transformation from one level to another on or more levels lower is referred to as design synthesis. Synthesis can also refer to the translation from the behavioural to the structural domain on the same level.

Typically, the design cycle consists of a series of transformations. A designer will generally start at the system level with a set of performance specifications. To be fabri-

cated in silicon, the circuit must eventually be represented at the layout level. In translating from the system level to the layout level, the design description will pass through some or all of the intermediate levels, increasing in complexity at each step. At times, the hardware designer will manually translate a design from one level to another, while at others automated synthesis will be performed with the aid of a software tool.

One of the choices presented to a designer is the level at which to describe the design. There is a trade off to be made. A design represented at a higher level will be simpler to describe and simulate, requiring less time to complete. However, the resulting lower level implementation may not be exactly what the designer desired and may not perform as well, with respect to power, speed, and area targets, as if the circuit had been designed at the gate, circuit, or even silicon level. Much of this depends on how the design is described at the upper level and the performance of the automatic synthesis engine. Increasing circuit size and complexity and the extremely tight time constraints of typical a design cycle push designers to implement a circuit at the highest level possible and have driven the design automation industry to develop more powerful synthesis tools.

What is required is a method of describing a design in the behavioural domain—a hardware description language (HDL). An HDL serves two purposes: documenting a design and modelling it [AG93]. A model is intended to represent hardware for simulation and synthesis. The desire to simulate complex circuits and systems is a driving force behind the development of a means of accurately modelling hardware. Prototyping a design is costly and time consuming. The designer can validate a design through HDL simulation and hopefully minimize the cost and time to market.

The MPS detailed description was done in the VHSIC (very high speed integrated circuit) hardware description language, or VHDL. The development of VHDL was begun

in 1983 by the United States Department of Defense, but has involved input from many sources in the computer and microelectronics industry. The IEEE became involved in the development of VHDL in 1985, and it was adopted as a standard in 1987.

4.2 MPS Register Transfer Level Description

Chapter 3 represents the high level design specification for the MPS, with performance specifications, a detailed description of the processor's operation, and a high level hardware structural description and block diagram. The next step in the design cycle is to generate the synthesizable RTL description for the MPS. The RTL was coded in VHDL based on guidelines outlined in Armstrong and Gray [AG93] and the Synopsys Design Compiler and VHDL Compiler reference manuals [Syn]. The following sections outline the RTL description and outline each block of code. The main features of each block and the rationale behind the style of coding are outlined. Please refer to Appendix A for the full RTL listings for each block.

4.2.1 Design Partitioning

The MPS top level block diagram shown in Chapter 3 (Figure 17) provides a model for the functional partitioning of the ASIC. This functional partitioning provides a good starting point in partitioning the ASIC for RTL coding. This breakdown of the circuit must now be evaluated for its effect on simulation and synthesis.

The control unit, as a finite state machine (FSM), is a special block and is instantiated as a unique module. The gate level compiler can employ specific strategies in compiling FSMs: minimizing the number of states, reducing the size of the state vector and thus the number of flip-flops required by the FSM, or ensuring that the generated gate level

Figure 27: MPS RTL Hierarchical View

description does not provide the means for the FSM to enter an invalid and potentially fatal state [Syn].

The RAM block must be partitioned as a separate module. During synthesis, any storage element will be mapped to a flip-flop or latch. This is fine for most registers, but it becomes impractical—due to size and power restrictions—when a large block of memory is required. For this reason, memory is generally not synthesized but is inserted into the design post-synthesis as a VLSI macro. The memory block, however, needs to be present

during RTL simulation gate level verification. A behavioural model based on the operation of the RAM macro is created for this purpose.

The data unit, logic unit, and mux_demux blocks provide the remainder of the MPS core circuitry. The data unit describes the operation of the set of registers outlined in Table 1, primarily sequential elements. The logic unit and the mux_demux are both strictly combinational logic. These portions of the circuit are partitioned primarily to simplify the simulation and debugging of the circuit. The ability to directly drive the logic unit inputs with whatever pattern of signals desired simplifies simulation of the RTL, as well as verification and debugging of the generated gate level description. The same is true with the mux_demux. The mux_demux is separate from the logic unit because there is no common circuitry between the two and the partitioning reduces the I/O signals for both blocks, again simplifying the simulation process.

The channel driver is a natural choice to be partitioned as a discrete module. The operation of the channel drivers is quite independent from the rest of MPS. As well, the block can be simulated, synthesized, optimized, and verified once and then copied ten times. By doing so we can reduce the synthesis time and ensure that the ten individual channels will operate in an identical fashion. Another reason for partitioning the channel driver from the rest of the circuit is that the channel driver will likely have different synthesis goals than the data unit or the control unit. For the control unit and data unit, minimizing the power and area will be critical, whereas more attention will be paid to the actual operation of the channel driver to ensure that the driver amplitude and direction controls are operating synchronously. This is an example of partitioning for different synthesis goals.

The channel driver itself is a structural block partitioned into a control unit and a data unit. The control unit is again implemented and synthesized as a finite-state-machine. The data unit houses the registers and combinational logic of the channel driver.

4.2.2 Top Level

The top level is a structural module in which the lower level blocks are instantiated. A top level module target for synthesis should be strictly structural, as emphasized in the Synopsys design for synthesis guidelines [Syn]. The inputs and outputs to the top level block are the “real-world” inputs to the MPS circuitry. The CLK signal is the reference to all the sequential elements within the MPS and is routed to most of the sub-modules. Similarly, RESET is provided as an input to the top level and routed to the sub-modules to initialize all sequential elements. The other two inputs are the 16-bit DATA_in, the input from the external clock and data recovery unit, and the word received flag (WR) used to signal the arrival of valid data at DATA_in.

The outputs of the top level are the electrode control signals for each channel: amplitude, current direction, and DAC enable signal. The electrode control signals from each channel driver are contained in a record (DRV_RECORD), as described in section 4.2.6 “Channel Driver” on page 88. The outputs from the ten channels are grouped into an array of records as the variable ELECTRODE(0..9). During simulation, the individual fields of each electrode control signal can be accessed by referring to ELECTRODE(1).AMP, ELECTRODE(2).DAC_EN, etc.

Within the architecture of the top level is the declaration and instantiation of the lower level modules: DATA_UNIT, CONTROL, LOGIC_UNIT, DRV_MUX, DRIVER, and the behavioural RAM SPRAM160x32. Ten copies of the channel driver module are

instantiated. A number of signals internal to the top level are declared and used to map the connectivity between the sub-modules. As with the ELECTRODE output, most of the signals associated with the channel drivers (DREQ_all, wr_AMP_all) are grouped into ten-bit arrays and indexed in the port mapping as each driver module is instantiated.

4.2.3 Data Unit

The data unit VHDL description is an RTL implementation of the data unit block diagram outlined in Chapter 3 (Figure 18). The entity description outlines the input ports, output ports, and the bidirectional memory interface. The CLK input is the clock from the top level that synchronizes the operation of the storage elements in the data unit with the rest of the MPS sequential circuitry. RESET is an active high, synchronous reset to all the registers. DATA_in is the 16-bit received data word, and is valid when WR is high. The control signal inputs are grouped together in the CTRL bus. Later, within the architectural description, CTRL is mapped to more meaningful signal names.

The outputs from the data unit are primarily the register values used to generate status signals for the control unit. These are included in the port description as output signals. The other output signal is DRVBUS, which is routed to the ten channel drivers in the top level. The data unit places the PCR and PDR values on this bus to load pulse description data into the drivers.

The memory interface completes the port description. An eight-bit address comes from the channel (CH) and instruction (INS) address registers. The 32-bit data port is a bidirectional interface between the PCR/PDR registers and the memory block.

The architectural implementation of the data unit contains the description of each register to be instantiated. One can see that the register descriptions follow a similar struc-

ture. To create the synchronous elements, the CLK signal is monitored, and nothing occurs unless a rising edge (CLK'event and CLK='1') is observed. This is then followed with a test of the common RESET signal. Most registers also have a register-specific reset signal (e.g., rst_COUNT) which is ORed with the global RESET. Either of these resets will force the register into a default state, usually all zeros, regardless of the status of the other control signals. If no reset signal is active, the process generally proceeds with one or more “if...then...else” structures to test the status of relevant control signals in order to load a value into the register, increment its current value, or perform some other task. When synthesized, this process structure will be inferred as one or more flip-flops—in accordance with the size of the signal—with a synchronous reset and whatever additional logic needed to handle features specific to that register (i.e., load, increment, decrement).

This basic structure is used for all of the data unit registers. The channel status register is handled as an array (ten words x 16-bits) in one process, with separate logic to choose the proper value from the array during reads. The channel address register requires some unique logic to force a wrap around after 1001 to correspond to the ten electrode pairs.

In addition to the register descriptions and the mapping of the CTRL vector to meaningful names, the data unit architectural description contains logic to direct the transfer of data between registers. These “if...then...else” statements will generate multiplexers at the register inputs to control what data is loaded.

Most signals in the data unit, and many on the top level, follow a specific naming convention. The registered signals are of the form:

REGISTERNAME_reg

The control signals generally follow the form:

`action_REGISTERNAME`

where `action` can be one of the following: `rd` (read from), `wr` (write to), `rst` (reset the register or flip flop to '0'), `set` (set the register or flip flop to '1'), `dec` (decrement), or `inc` (increment). The register name will be the same as the respective registered signal. Sometimes the register name is followed by `_all`. This is used to differentiate between an individual status bit, such as `STIM_STAT`, and the grouping of all ten bits together for performing a global reset.

4.2.4 Logic Unit

The logic unit is a simple collection of signal assignments that generates the status vector based on the current state of the data unit and channel driver registers. In the entity declaration, the port description lists the registers used as inputs; the status vector is the lone output. The mapping of the status signals within the status vector is described in the comments of the logic unit.

Within the architecture description are the status signal assignments. The statement used to imply the combinational logic uses the `when...else` structure. This is a compact version of the `if...then...else` structure, and will be inferred in the same manner during compilation. For simple tests the `when...else` statement is usually shorter and easier to read.

Two additional signal assignments select the appropriate bit from the `DREQ_all` and `STIM_STAT_all` using the current channel, indicated by the `CH` register. The function `conv_integer` is used to convert the value in `CH` to an integer value which can then be used to index the vectored signals.

4.2.5 Control Unit

When describing a state machine targeted for synthesis there are a few recommended styles to follow [Gol94]. The manner in which the state machine is described in VHDL will have a profound effect on the hardware that is synthesized. There are two types of FSM implementations: highly encoded and one-hot [Gol94]. The current state of the machine is stored in state memory, a set of flip-flops the outputs of which are called the state vector. The two implementations represent the manner in which the state vector is stored. Each state of the state machine can be assigned a unique code, so that n states requires $n/2$ storage elements, or flip-flops. This is a highly encoded implementation of an FSM. Additional combinational logic is required to decode the current state from the state vector.

Alternatively, a flip-flop can be used for each of the n states. This is the “one-hot” implementation; for a valid state vector, only one bit will be true. No encoding or decoding of the state vector is required as the current state can be readily identified from the state vector.

The advantages of the one-hot implementation often outweigh the additional hardware required, i.e., double the flip-flops [Gol94]. The one-hot implementation is typically faster than an equivalent highly encoded design. There is less delay because no combinational logic is required to decode the state vector, and the speed of the circuit is independent of the number of states. A highly encoded machine will slow down as the number of states and the corresponding combinational logic increases. The area penalty imposed by the increased number of storage elements is typically not that large over a highly encoded design [Gol94]. A one-hot FSM is also easier to design and modify. The HDL code can be

written directly from the state diagram without coding a state table. States can easily be added or removed and the logic altered without affecting the whole design.

Simulating and debugging the one-hot circuit can be easier since illegal state transitions are more readily identified. This also provides more reliable operation of the synthesized hardware. Because the number of valid states (n) is a small fraction of the values expressible by the state vector (2^n) it is easier to identify an illegal state and thus take the proper corrective action. With a highly encoded machine an illegal state transition is more likely to result in a valid state, which then may lead to unpredictable and possibly dangerous behaviour, something that is definitely not desired in any circuit, especially a medical implant.

For these reasons, the control unit of the MPS (as well as the smaller one in the channel driver, as we will see later) is implemented as a one-hot FSM. The programming and stimulation modes described in section 3.6 “Operation” on page 68 are integrated into a single module incorporating 26 states. The control unit is coded from the state diagrams in figures 25 and 26. The status signals are based on the state transition conditions, and the control signals are outputs based on the operations to be performed in each state. The status and control signals are gathered and grouped into two vectors, the mappings of which to their respective vectored signals are described in the logic unit and data unit comments, respectively.

The one-hot implementation used for the MPS control unit is outlined by Golson, 1994. The procedure is to examine the STATE vector one bit at a time, looking for the one that is set to identify the current state. Once the state is identified, the proper output control signals are generated, and the next state transition is computed based on the current

state and the status signals. The next state transition is handled by setting the appropriate bit in the NEXT_STATE vector to '1'. The `others=>'0'` statement in line 54 is used to set all the other bits to zero by default. On the next rising edge of CLK, the STATE vector is loaded with the value in NEXT_STATE and the whole process repeats itself.

In coding the state machine, three separate VHDL processes are used to instantiate the state transition logic, the output control signal generation logic, and the state vector flip-flops themselves. This conveniently matches the partitioning shown in the state machine block diagram in Chapter 3 (Figure 21). The state machine is coded in this fashion to ensure that the synthesized circuit will behave in the same manner as the behavioural model [Gol94]. The first two processes are similar, proceeding through the state vector bit by bit until the current state is reached and then either selecting the next state bit (`GenerateNextState`) or generating the output control signals (`GenerateOutput`). These processes will synthesize the combinational portion of the state machine. The last process follows the familiar form for generating a synchronous storage element. Note the difference from most of the storage elements coded so far, in that an asynchronous reset is used. This provides the means to initialize the control unit state machine to its default state at any time.

4.2.6 Channel Driver

The channel driver top level is primarily a structurally oriented module. The port description outlines the interface between the channel driver and the rest of the MPS circuitry. As with the other MPS blocks, the system clock and reset signal are brought in as inputs. The bus `DRVBUS` brings in the stimulation information from the data unit, and the `wr_AMP` and `wr_DIR` control signals from the MPS control unit are used to latch in this data. The status signals `STIME` and `STIM_STAT` are brought as status signals for the

channel driver control unit. As outputs, the channel driver provides the status signal DREQ for the MPS core, and DRV_OUT, a signal of a custom data type DRV_RECORD. DRV_RECORD is a record that was created to describe the current status of a single channel driver output. The record description is located in the file mps_pack.vhd. The six data fields within the record are as follows:

AMP: 8-bit vector — stimulation signal amplitude

DAC_EN: control bit to the external DAC

ASRC, BSRC: current direction control bits to A and B electrodes

AON, BON: on/off control bit to A and B electrodes.

By grouping these different data elements into a single record we can reduce the number of individual signals coming out of the channel driver. This, in turn, simplifies the circuit test bench and makes it easier to track these signals during simulation. The record data type is supported for synthesis [Syn].

The architecture invokes the channel driver data unit and the channel driver control unit and generates the connections between the two. Also within the architecture is the mapping of the various channel driver outputs into the DRV_OUT record.

Channel Driver Data Unit

The channel driver data unit provides the RTL description of the channel driver registers shown in Figure 22. These registers are instantiated in the same manner as those in the MPS core data unit. The relatively small amount of combinational logic required by the channel driver did not warrant a separate logic unit, so a small combinational logic section is included in the architectural description.

Channel Driver Control Unit

Like the MPS core control unit, the channel driver control unit is a one-hot implementation of a finite state machine. The first process handles the next state decoding, the second generates the output control signals, and the last instantiates the FSM flip flops. The channel driver control and status signals, unlike those in the MPS core, are not grouped into vectors. There are not as many signals to fill up the port declarations so they are left on their own with meaningful, easier to trace names.

4.2.7 MUX/DEMUX

Another small module is created to handle the multiplexing and demultiplexing of control and status signals between the channel drivers and the rest of the MPS. Two control signals, C1 and C2, are routed to the proper channel driver. A single status bit, S1, is selected from a vector of inputs, S1_all. In both cases, the current channel, CH, is used to as the selection control.

For each control signal, an index is increased from zero to nine. When the current value of CH (converted to an integer) is reached, the index is used to copy the control signal into the proper bit of the control bit vector. All other bits within the vector are set to zero. Although the VHDL algorithm operates in a sequential manner, it will be synthesized as combinational logic

To select the status bit from the vector S1_all, CH is converted to an integer and used to index the vector.

4.2.8 Other VHDL Entities

In addition to the RTL described above, a number of other VHDL entities were generated to support the MPS design. These include the behavioural RAM model required

for simulation and a VHDL package which was used to hold custom data structures specific to the MPS code.

Behavioural RAM

As discussed earlier, RAM is not synthesized but added to the circuit following synthesis. A behavioural description is still required to represent the RAM during simulation. For this reason, a behavioural description, not targeted for synthesis, was coded. This block was coded to represent the operation of a specific module in the TI TGC3000 process. The RAM behavioural block is the only vendor specific module. If a new process is targeted for synthesis, the RAM block and its operation must be updated to accommodate the new hardware.

mps_pack

mps_pack is a VHDL package that was created to store any design-level functions or data types. In this case the data record DRV_RECORD was defined in mps_pack for use in the channel driver block and at the top level.

Chapter 5

Conclusion

For profoundly deaf individuals, a cochlear implant provides the opportunity to restore the sensation of hearing that cannot be achieved with traditional amplification based hearing aids. In this study, a number of existing designs have been reviewed, focusing on speech processing techniques, electrical stimulation strategies, electrode geometry, and the trans-dermal coupling technique. These systems have achieved varying levels of success, but a major hurdle remains in that the performance of a particular system or strategy is difficult to predict prior to implantation. For this reason, the Multipulse Programmable Stimulator (MPS) has been designed to support the modification of the speech processing and stimulation strategies post-operatively, allowing a cochlear implant system to be customized for each individual patient. It was hoped that this programmability would accommodate the majority of existing speech processing strategies while supporting future advances. The flexibility of the cochlear implant system, particularly the implanted hardware, is critical. The MPS was developed to accommodate a wide range of pulsatile or composite analog stimulation patterns. With the core clock running at 1.6 MHz, the MPS can simultaneously drive all ten bipolar channels each at a maximum stimulation rate of 8900 pulses/second (pps). This rate can be increased as the number of stimulating channels is reduced, up to a maximum rate of 36 000 pps for a single channel. The Clarion system, which supported the highest stimulation rate of the cochlear implant systems reviewed, is capable of driving its eight channels at 3250 pps. The elementary MPS stimulating pulse width can range between 10 μ s and 2560 μ s, a comparable range to that of the reviewed implants (Clarion: 0 to 500 μ s; Nucleus: 20 to 400 μ s; Digisonic: 5 to 310 μ s). To generate a long, low amplitude preconditioning pulse, longer pulses of up to 653 mS in

length can be generated by seamlessly repeating a single pulse up to 255 times. The MPS pulses amplitude ranges between 0 and 2550 μA , also matching the range of the reviewed systems. The ten bipolar channels can be configured as ten bipolar electrode pairs or in a wide variety of common ground or cross-channel stimulation schemes. A system based on this circuit will be able to support most of the reviewed existing processing strategies. One possible exception is the Nucleus 22-channel system, which may have better selectivity of the stimulation site along the basilar membrane because of the additional electrodes. This flexibility was sacrificed to allow the MPS to support simultaneous stimulation of all ten channels, which the Nucleus device does not support. Future design work could possibly expand the number of channels supported.

The digital core of the MPS, representing the bulk of the stimulation control circuitry, has been coded in VHDL for synthesis. The design is process independent (except for a small on-chip RAM, which is typically compiled for a specific technology) and can thus be targeted to a silicon vendor of choice. The circuit architecture was kept as simple as possible in order to minimize circuit size and power consumption without sacrificing the operational design goals laid out in Chapter 3. For example, the MPS core logic was minimized by using common registers for programming and stimulation operations. The distributed control architecture—a single finite state machine (FSM) for the core control unit, and ten individual FSMs controlling the channel drivers—provides the means for the channel drivers to maintain independent operation and support the wide range of stimulation formats desired. It also helps keep power consumption down by activating the control units of only the individual channel drivers scheduled for stimulation. Partitioning of the circuit for design, simulation, and synthesis was based on the top level circuit description and design for synthesis guidelines. The circuit description has been simulated to verify

that it operates as desired. A compiler was created in PERL (Practical Extraction and Report Language), a high level programming language especially suited for processing text. The compiler was created to generate the binary stimulation description words for the MPS, and a VHDL test bench was created to apply the input and monitor the output. The circuit operation was verified for a number of different stimulation styles.

The RTL description of the MPS circuit was synthesized and mapped into Texas Instrument's TGC4000 series 3.3V CMOS, 0.35 μm gate array technology to arrive at a power estimate for the digital core. Synopsys' DesignPower tool used the synthesized netlist to estimate the maximum dynamic power consumption at 1.61 mW (490 μA at 3.3 V) while running at a clock speed of 1.6 MHz. While this value does not take into account the power consumption of the receiver and channel driver D/A circuitry, it provides a strong measure of confidence that the circuit can be implanted and used in a mobile implant system. A direct comparison of the MPS power performance to that of most other implanted microcircuits is difficult. Detailed power consumption ratings of the implanted circuitry for the four cochlear implants reviewed in Chapter 2 were not available. In fact, Cochlear Limited (Nucleus) and Advanced Bionics Corporation (Clarion) were contacted directly but would not release this information. A comparison can be made with the Chorimac-12, a 12-channel monopolar system based on an implanted circuit with a maximum current draw of 100 μA [GC87]. The supply voltage to this circuit was not indicated, so the power consumption could possible range from 330 μW to 1200 mW (V_{cc} between 3.3 V and 12 V). This is somewhat lower than the estimated power consumption of the MPS. Future design efforts should attempt to explore ways to reduce the MPS power consumption to achieve the full value of an MPS based cochlear implant system.

Appendix A

MPS VHDL Description

Appendix A contains the VHDL register transfer level (RTL) description of the MPS.

A.1 Top Level

```
1| -- top.vhd
2| -- MPS VHDL Description
3| -- top level
4| -- Colin Loisel
5| -- version 2.2 -- 17/04/1999
6| -- Updated top with new RAM_COLLAR block replacing SPRAM.
7| -- Control unit CTRL updated to drive memory enable.
8| -- Data unit updated with memory address output and new CTRL bus width.
9| -- Logic unit updated with new port map ordering.
10| -- version 2.1 -- 05/04/1999
11| -- Renamed DRV_OUTPU to ELECTRODE
12| -- Renamed DATABUS to DRVBUS; DRVBUS no longer bi-dir
13| -- version 2.0 -- 19/08/1997
14| -- TOP v2.0 invokes the updated versions of the control unit and
15| -- data unit, as well as the new channel driver and logic unit.
16| --
17| library IEEE;
18| use IEEE.std_logic_1164.all;
19| use IEEE.std_logic_unsigned.all;
20| use IEEE.std_logic_arith.all;
21|
22| use work.all;
23| use work.mps_pack.all;
24|
25| entity TOP is
26|     port(CLK      : in  std_logic;
27|          RESET    : in  std_logic;
28|          DATA_in  : in  std_logic_vector(15 downto 0);
29|          WR        : in  std_logic;
30|          ELECTRODE : out DRV_RECORD_ARRAY(0 to 9));
31| end TOP;
32|
33| architecture STRUCTURE of TOP is
34|     signal DRVBUS: std_logic_vector(15 downto 0);
35|     signal CTRL: std_logic_vector(0 to 34);
36|     signal STATUS: std_logic_vector(0 to 16);
37|     signal CH: std_logic_vector(3 downto 0);
38|     signal REC: std_logic_vector(15 downto 0);
39|     signal ACC: std_logic_vector(15 downto 0);
```

```

40| signal PCR: std_logic_vector(15 downto 0);
41| signal COUNT: std_logic_vector(7 downto 0);
42| signal STIM_STAT_all: std_logic_vector(0 to 9);
43| signal DREQ: std_logic;
44| signal DREQ_all: std_logic_vector(0 to 9);
45| signal wr_AMP_all: std_logic_vector(0 to 9);
46| signal wr_DIR_all: std_logic_vector(0 to 9);
47| signal STIME: std_logic;
48| signal RAM_DATA: std_logic_vector(31 downto 0);
49| signal RAM_ADDR: std_logic_vector(7 downto 0);
50|
51| component CONTROL
52| port (CLK : in std_logic;
53|       RESET: in std_logic;
54|       I : in std_logic_vector(0 to 16); -- Status inputs
55|       O : out std_logic_vector(0 to 34)); -- Control outputs
56| end component;
57|
58| component DATA_UNIT
59| port (CLK : in std_logic;
60|       RESET : in std_logic;
61|       WR : in std_logic;
62|       CTRL : in std_logic_vector(0 to 34);
63|       DATA_in : in std_logic_vector(15 downto 0);
64|       REC : out std_logic_vector(15 downto 0);
65|       ACC : out std_logic_vector(15 downto 0);
66|       PCR : out std_logic_vector(15 downto 0);
67|       COUNT : out std_logic_vector(7 downto 0);
68|       CH : out std_logic_vector(3 downto 0);
69|       STIM_STAT_all: out std_logic_vector(0 to 9);
70|       STIME : out std_logic;
71|       DRVBUS : out std_logic_vector(15 downto 0);
72|       MEM_ADDR : out std_logic_vector(7 downto 0);
73|       MEM_DATA : inout std_logic_vector(31 downto 0));
74| end component;
75|
76| component LOGIC_UNIT
77| port (REC : in std_logic_vector(15 downto 0);
78|       PCR : in std_logic_vector(15 downto 0);
79|       ACC : in std_logic_vector(15 downto 0);
80|       COUNT : in std_logic_vector(7 downto 0);
81|       CH : in std_logic_vector(3 downto 0);
82|       STIM_STAT_all: in std_logic_vector(0 to 9);
83|       DREQ_all : in std_logic_vector(0 to 9);
84|       WR : in std_logic;
85|       STIME : in std_logic;
86|       STATUS : out std_logic_vector(0 to 16));
87| end component;
88|
89| component DRV_MUX
90| port (CH : in std_logic_vector(3 downto 0); -- channel address
91|       C1 : in std_logic;
92|       C2 : in std_logic;

```

```

93|         S1_all: in  std_logic_vector(0 to 9);
94|         C1_all: out std_logic_vector(0 to 9);
95|         C2_all: out std_logic_vector(0 to 9);
96|         S1      : out std_logic);
97|     end component;
98|
99|     component DRIVER
100|     port(CLK      : in std_logic;
101|          RESET    : in std_logic;
102|          DRVBUS   : in std_logic_vector(15 downto 0);
103|          wr_AMP   : in std_logic;
104|          wr_DIR   : in std_logic;
105|          STIME    : in std_logic;
106|          STIM_STAT : in std_logic;
107|          DREQ     : out std_logic;
108|          DRV_OUT  : out DRV_RECORD);
109|     end component;
110|
111|     component SPRAM_COLLAR
112|     port(CLK      : in std_logic;
113|          ADDR     : in std_logic_vector(7 downto 0);      -- address bus
114|          DATA    : inout std_logic_vector(31 downto 0); -- data bus
115|          en_MEM   : in std_logic;                        -- RAM enable, active high
116|          rd_MEM   : in std_logic;                        -- read enable, active high
117|          wr_MEM   : in std_logic;                        -- write enable, active high
118|     end component;
119|
120| begin
121|
122|     DU: DATA_UNIT port map (CLK, RESET, WR, CTRL, DATA_in, REC, ACC,
123|        PCR, COUNT, CH, STIM_STAT_all, STIME, DRVBUS, RAM_ADDR, RAM_DATA);
124|
125|     CU: CONTROL port map (CLK, RESET, STATUS, CTRL);
126|
127|     LU: LOGIC_UNIT port map (REC, PCR, ACC, COUNT, CH, STIM_STAT_all,
128|        DREQ_all, WR, STIME, STATUS);
129|
130|     MUX: DRV_MUX port map (CH, CTRL(19), CTRL(18), DREQ_all,
131|        wr_AMP_all, wr_DIR_all, DREQ);
132|
133|     DRV0: DRIVER port map (CLK, RESET, DRVBUS, wr_AMP_all(0), wr_DIR_all(0),
134|        STIME, STIM_STAT_all(0), DREQ_all(0), ELECTRODE(0));
135|
136|     DRV1: DRIVER port map (CLK, RESET, DRVBUS, wr_AMP_all(1), wr_DIR_all(1),
137|        STIME, STIM_STAT_all(1), DREQ_all(1), ELECTRODE(1));
138|
139|     DRV2: DRIVER port map (CLK, RESET, DRVBUS, wr_AMP_all(2), wr_DIR_all(2),
140|        STIME, STIM_STAT_all(2), DREQ_all(2), ELECTRODE(2));
141|
142|     DRV3: DRIVER port map (CLK, RESET, DRVBUS, wr_AMP_all(3), wr_DIR_all(3),
143|        STIME, STIM_STAT_all(3), DREQ_all(3), ELECTRODE(3));
144|
145|     DRV4: DRIVER port map (CLK, RESET, DRVBUS, wr_AMP_all(4), wr_DIR_all(4),

```

```

146|     STIME, STIM_STAT_all(4), DREQ_all(4), ELECTRODE(4));
147|
148| DRV5: DRIVER port map (CLK, RESET, DRVBUS, wr_AMP_all(5), wr_DIR_all(5),
149|     STIME, STIM_STAT_all(5), DREQ_all(5), ELECTRODE(5));
150|
151| DRV6: DRIVER port map (CLK, RESET, DRVBUS, wr_AMP_all(6), wr_DIR_all(6),
152|     STIME, STIM_STAT_all(6), DREQ_all(6), ELECTRODE(6));
153|
154| DRV7: DRIVER port map (CLK, RESET, DRVBUS, wr_AMP_all(7), wr_DIR_all(7),
155|     STIME, STIM_STAT_all(7), DREQ_all(7), ELECTRODE(7));
156|
157| DRV8: DRIVER port map (CLK, RESET, DRVBUS, wr_AMP_all(8), wr_DIR_all(8),
158|     STIME, STIM_STAT_all(8), DREQ_all(8), ELECTRODE(8));
159|
160| DRV9: DRIVER port map (CLK, RESET, DRVBUS, wr_AMP_all(9), wr_DIR_all(9),
161|     STIME, STIM_STAT_all(9), DREQ_all(9), ELECTRODE(9));
162|
163| RAM: SPRAM_COLLAR port map (CLK, RAM_ADDR, RAM_DATA, CTRL(34),
164|     CTRL(28), CTRL(17));
165|
166|end STRUCTURE;

```

A.2 Data Unit

```

1| -- dataunit.vhd
2| -- MPS VHDL Description
3| -- data unit
4| -- version 3.3 -- 09/10/1999
5| --   Update DRVBUS input mux to include a default state of all 0's if
6| --   neither rd_PDR nor rd_PCR are '1'.
7| --   Corrected typo in DRVBUS input mux logic.  if rd_PDR <= '1' changed
8| --   to if rd_PDR = '1'                                ^^^
9| -- version 3.2 -- 01/10/1999
10| --   Update COUNT input mux so that only the bottom six bits from REC
11| --   are loaded instead of the bottom eight bits.
12| -- version 3.1 -- 17/04/1999
13| --   Updated CTRL vector to 35-bits wide to match control unit.
14| --   Added MEM_ADDR to port map.
15| -- version 3.0 -- 07/04/1999
16| --   Eliminated internal DATABUS from module.  All register transfers
17| --   use dedicated connections.  Muxes were added to support this new
18| --   configuration.  Signals INS_in, BRANCH_in, COUNT_in, PCR_in, and
19| --   PDR_in added.
20| --   IR and MDR updated to PCR and PDR.
21| --   DATABUS (inout port) renamed DRVBUS (out port)
22| --   D_in renamed DATA_in
23| -- version 2.2 -- 19/08/1997
24| --   Added ACC to port signals.
25| -- version 2.1 -- 09/08/1997
26| --   Commented out INC and DEC functions and replaced with a simple
27| --   "X <= X + 1" type assignment.  Call std_logic_unsigned to
28| --   provide overloaded arithmetic operations.
29| --   Renamed registered signals to XXX_reg in architecture.
30| --   Fixed 55 errors.
31| -- version 2.0 -- 02/02/97
32| --   New version of the data unit to correspond with the v2.0 of the
33| --   control unit.
34| -- Colin Loisselle
35| --
36| --
37| library IEEE;
38| use IEEE.std_logic_1164.all;
39| use IEEE.std_logic_unsigned.all;
40| use IEEE.std_logic_arith.all;
41|
42| entity DATA_UNIT is
43|   port(CLK           : in std_logic;
44|        RESET         : in std_logic;
45|        WR            : in std_logic;
46|        CTRL          : in std_logic_vector(0 to 34);
47|        DATA_in      : in std_logic_vector(15 downto 0);
48|        REC           : out std_logic_vector(15 downto 0);
49|        ACC           : out std_logic_vector(15 downto 0);
50|        PCR           : out std_logic_vector(15 downto 0);

```

```

51|         COUNT          : out std_logic_vector(7 downto 0);
52|         CH              : out std_logic_vector(3 downto 0);
53|         STIM_STAT_all   : out std_logic_vector(0 to 9);
54|         STIME           : out std_logic;
55|         DRVBUS          : out std_logic_vector(15 downto 0);
56|         MEM_ADDR        : out std_logic_vector(7 downto 0);
57|         MEM_DATA        : inout std_logic_vector(31 downto 0));
58| end DATA_UNIT;
59|
60| architecture RTL of DATA_UNIT is
61|
62|     -- Channel Status Register array
63|     type REG_ARRAY is array (0 to 9) of std_logic_vector(15 downto 0);
64|     signal CSR_all: REG_ARRAY;
65|
66|     -- Register inputs and outputs
67|     signal CSR_mux_out      : std_logic_vector(15 downto 0);
68|     signal INS_reg, INS_in  : std_logic_vector(3 downto 0);
69|     signal CH_reg           : std_logic_vector(3 downto 0);
70|     signal COUNT_reg, COUNT_in : std_logic_vector(7 downto 0);
71|     signal BRANCH_reg, BRANCH_in: std_logic_vector(3 downto 0);
72|     signal ACC_reg, ACC_in  : std_logic_vector(15 downto 0);
73|     signal REC_reg         : std_logic_vector(15 downto 0);
74|     signal PCR_reg, PCR_in  : std_logic_vector(15 downto 0);
75|     signal PDR_reg, PDR_in  : std_logic_vector(15 downto 0);
76|
77|     -- Control signals
78|     signal rst_STIM_STAT_all: std_logic;
79|     signal rst_STIM_STAT: std_logic;
80|     signal rst_STIME: std_logic;
81|     signal rst_ACC: std_logic;
82|     signal rst_INS: std_logic;
83|     signal rst_COUNT: std_logic;
84|     signal rst_CH: std_logic;
85|     signal rst_CSR_all: std_logic;
86|     signal set_STIM_STAT: std_logic;
87|     signal set_STIME: std_logic;
88|     signal wr_CH: std_logic;
89|     signal wr_INS: std_logic;
90|     signal wr_BRANCH: std_logic;
91|     signal wr_COUNT: std_logic;
92|     signal wr_PCR: std_logic;
93|     signal wr_PDR: std_logic;
94|     signal wr_CSR: std_logic;
95|     signal wr_MEM: std_logic;
96|     signal wr_DIR: std_logic;
97|     signal wr_AMP: std_logic;
98|     signal wr_DUR: std_logic;
99|     signal rd_REC: std_logic;
100|    signal rd_INS: std_logic;
101|    signal rd_BRANCH: std_logic;
102|    signal rd_COUNT: std_logic;
103|    signal rd_PCR: std_logic;

```

```

104| signal rd_PDR: std_logic;
105| signal rd_CSR: std_logic;
106| signal rd_MEM: std_logic;
107| signal dec_COUNT: std_logic;
108| signal inc_INS: std_logic;
109| signal inc_CH: std_logic;
110| signal en_BRANCH: std_logic;
111| signal xor_ACC: std_logic;
112|
113| begin
114|   -- Map control signals CTRL(0..33) to meaningful names
115|   rst_STIM_STAT_all <= CTRL(0);
116|   rst_STIM_STAT <= CTRL(1);
117|   rst_STIME <= CTRL(2);
118|   rst_ACC <= CTRL(3);
119|   rst_INS <= CTRL(4);
120|   rst_COUNT <= CTRL(5);
121|   rst_CH <= CTRL(6);
122|   rst_CSR_all <= CTRL(7);
123|   set_STIM_STAT <= CTRL(8);
124|   set_STIME <= CTRL(9);
125|   wr_CH <= CTRL(10);
126|   wr_INS <= CTRL(11);
127|   wr_BRANCH <= CTRL(12);
128|   wr_COUNT <= CTRL(13);
129|   wr_PCR <= CTRL(14);
130|   wr_PDR <= CTRL(15);
131|   wr_CSR <= CTRL(16);
132|   wr_MEM <= CTRL(17);
133|   wr_DIR <= CTRL(18);
134|   wr_AMP <= CTRL(19);
135|   wr_DUR <= CTRL(20);
136|   rd_REC <= CTRL(21);
137|   rd_INS <= CTRL(22);
138|   rd_BRANCH <= CTRL(23);
139|   rd_COUNT <= CTRL(24);
140|   rd_PCR <= CTRL(25);
141|   rd_PDR <= CTRL(26);
142|   rd_CSR <= CTRL(27);
143|   rd_MEM <= CTRL(28);
144|   dec_COUNT <= CTRL(29);
145|   inc_INS <= CTRL(30);
146|   inc_CH <= CTRL(31);
147|   en_BRANCH <= CTRL(32);
148|   xor_ACC <= CTRL(33);
149|
150|   -- Port signals
151|   CH <= CH_reg;
152|   COUNT <= COUNT_reg;
153|   PCR <= PCR_reg;
154|   REC <= REC_reg;
155|   ACC <= ACC_reg;
156|   MEM_ADDR <= CH_reg & INS_reg;

```



```

157| -----
158| --*****
159| -- Combinational logic to support data transfers within Data Unit
160| -----
161| process(BRANCH_reg, INS_reg, CSR_mux_out, CSR_all, REC_reg, PCR_reg, PDR_reg,
162|         CH_reg, ACC_reg, MEM_DATA, en_BRANCH, rd_REC, rd_PDR, rd_PCR, wr_MEM,
163|         rd_MEM)
164| begin
165|     -- INS and BRANCH input mux
166|     if en_BRANCH = '1' then
167|         INS_in <= BRANCH_reg;
168|         BRANCH_in <= INS_reg;
169|     else
170|         INS_in <= CSR_mux_out(15 downto 12);
171|         BRANCH_in <= CSR_mux_out(11 downto 8);
172|     end if;
173|
174|     -- COUNT input mux
175|     if rd_REC = '1' then
176|         COUNT_in <= "00" & REC_reg(5 downto 0);
177|     elsif rd_PCR = '1' then
178|         COUNT_in <= PCR_reg(7 downto 0);
179|     else
180|         COUNT_in <= CSR_mux_out(7 downto 0);
181|     end if;
182|
183|     -- PCR and PDR bidirectional interface to MEM_DATA
184|     if rd_MEM = '1' then
185|         PCR_in <= MEM_DATA(31 downto 16);
186|         PDR_in <= MEM_DATA(15 downto 0);
187|     else
188|         PCR_in <= REC_reg;
189|         PDR_in <= REC_reg;
190|     end if;
191|
192|     if wr_MEM = '1' then
193|         MEM_DATA <= PCR_reg & PDR_reg;
194|     else
195|         MEM_DATA <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
196|     end if;
197|
198|     -- PCR and PDR output onto DRVBUS
199|     if rd_PDR = '1' then
200|         DRVBUS <= PDR_reg;
201|     elsif rd_PCR = '1' then
202|         DRVBUS <= PCR_reg;
203|     else
204|         DRVBUS <= "0000000000000000";
205|     end if;
206|
207|     -- CSR: Use CH_reg value to select CSR from CSR_all
208|     CSR_mux_out <= CSR_all(conv_integer(CH_reg));
209|

```

```

210|    -- 16-bit XOR of ACC and REC to calculate checksum
211|    ACC_in <= ACC_reg xor REC_reg;
212| end process;
213|
214|  -----
215|  -- Registers
216|  -----
217|  -- REC_reg: Input data receive register
218|  -- 16-bit register with synchronous reset
219|  process(CLK, RESET, WR, DATA_in)
220|  begin
221|    if (CLK'event and CLK='1') then
222|      if RESET='1' then
223|        REC_reg <= "0000000000000000";
224|      elsif (WR='1') then
225|        REC_reg <= DATA_in;
226|      end if;
227|    end if;
228|  end process;
229|
230|  -- CH_reg: Channel address register
231|  -- 4-bit incrementable register from 0 to 9
232|  -- with overflow and synchronous reset
233|  process(CLK, RESET, rst_CH, wr_CH, inc_CH)
234|  begin
235|    if (CLK'event and CLK='1') then
236|      if (RESET='1' or rst_CH='1') then-- synchronous reset
237|        CH_reg <= "0000";
238|      elsif wr_CH='1' then          -- load CH_reg
239|        CH_reg <= REC_reg(15 downto 12);-- from REC
240|      elsif inc_CH='1' then          -- increment CH_reg
241|        if CH_reg="1001" then        -- wrap CH_reg at 9
242|          CH_reg <= "0000";
243|        else
244|          CH_reg <= CH_reg + "1";
245|        end if;
246|      end if;
247|    end if;
248|  end process;
249|
250|  -- INS_reg: Instruction address register
251|  -- 4-bit incremental register with sync. reset
252|  process(CLK, RESET, rst_INS, wr_INS, inc_INS, INS_in)
253|  begin
254|    if (CLK'event and CLK='1') then
255|      if (RESET='1' or rst_INS='1') then  -- synchronous reset
256|        INS_reg <= "0000";
257|      elsif wr_INS='1' then                -- write INS_reg
258|        INS_reg <= INS_in;
259|      elsif inc_INS='1' then                -- increment INS_reg
260|        INS_reg <= INS_reg + "1";
261|      end if;
262|    end if;

```

```

263| end process;
264|
265| -- BRANCH_reg: Branch address register
266| -- 4-bit register with sync. reset
267| process(CLK, RESET, wr_BRANCH, en_BRANCH, BRANCH_in)
268| begin
269|     if (CLK'event and CLK='1') then
270|         if RESET='1' then -- synchronous reset
271|             BRANCH_reg <= "0000";
272|         elsif wr_BRANCH='1' then -- write BRANCH_reg
273|             BRANCH_reg <= BRANCH_in;
274|         end if;
275|     end if;
276| end process;
277|
278| -- COUNT_reg: Counter
279| -- 8-bit down counter with sync. reset
280| process(CLK, RESET, rst_COUNT, wr_COUNT, dec_COUNT)
281| begin
282|     if (CLK'event and CLK='1') then
283|         if (RESET='1' or rst_COUNT='1') then -- synchronous reset
284|             COUNT_reg <= "00000000";
285|         elsif wr_COUNT='1' then -- load counter
286|             COUNT_reg <= COUNT_in;
287|         elsif dec_COUNT='1' then -- decrement COUNT_reg
288|             COUNT_reg <= COUNT_reg - "1";
289|         end if;
290|     end if;
291| end process;
292|
293| -- PCR_reg: Register
294| -- 16-bit register with synchronous reset
295| -- interfaces between REC, DRVBUS, and memory
296| process(CLK, RESET, wr_PCR, PCR_in)
297| begin
298|     if (CLK'event and CLK='1') then
299|         if RESET='1' then -- synchronous reset
300|             PCR_reg <= "0000000000000000";
301|         elsif wr_PCR='1' then -- load PCR
302|             PCR_reg <= PCR_in;
303|         end if;
304|     end if;
305| end process;
306|
307| -- PDR_reg: Register
308| -- 16-bit register with synchronous reset
309| -- interfaces between REC, DRVBUS, and memory
310| process(CLK, RESET, wr_PDR, PDR_in)
311| begin
312|     if (CLK'event and CLK='1') then
313|         if RESET='1' then -- synchronous reset
314|             PDR_reg <= "0000000000000000";
315|         elsif wr_PDR='1' then -- load from DRVBUS

```

```

316|         PDR_reg <= PDR_in;
317|     end if;
318| end if;
319| end process;
320|
321| -- CSR_all: Channel status register array
322| -- array of 10x16-bit registers
323| -- global synchronous reset
324| --
325| process(CLK,RESET,rst_CSR_all,wr_CSR,CH_reg)
326| begin
327|     if (CLK'event and CLK='1') then
328|         if (RESET='1' or rst_CSR_all='1') then    -- synchronous reset
329|             CSR_all <= ("0000000000000000",
330|                 "0000000000000000",
331|                 "0000000000000000",
332|                 "0000000000000000",
333|                 "0000000000000000",
334|                 "0000000000000000",
335|                 "0000000000000000",
336|                 "0000000000000000",
337|                 "0000000000000000",
338|                 "0000000000000000");
339|
340|             -- write to CSR array is indexed by CH_reg value
341|             elsif wr_CSR='1' then
342|                 CSR_all(conv_integer(CH_reg)) <= (INS_reg & BRANCH_reg & COUNT_reg);
343|             end if;
344|         end if;
345|     end process;
346|
347| -- ACC_reg: 16-bit Accumulator for storing checksum
348| process(CLK,RESET,rst_ACC,xor_ACC,ACC_in)
349| begin
350|     if (CLK'event and CLK='1') then
351|         if (RESET='1' or rst_ACC='1') then    -- synchronous reset
352|             ACC_reg <= "0000000000000000";
353|         elsif xor_ACC='1' then
354|             ACC_reg <= ACC_in;
355|         end if;
356|     end if;
357| end process;
358|
359| -- STIM_STAT_all: Channel stimulation status
360| -- 10-bit status register
361| -- A high bit indicates that the corresponding channel has
362| -- valid stimulation data stored in memory.
363| -- All ten bits can be reset together, and can be set or reset
364| -- individually.
365| process(CLK,RESET,rst_STIM_STAT_all,rst_STIM_STAT,set_STIM_STAT)
366| begin
367|     if (CLK'event and CLK='1') then
368|         if (RESET='1' or rst_STIM_STAT_all='1') then

```

```

369|         STIM_STAT_all <= "0000000000";
370|     elsif (rst_STIM_STAT='1') then
371|         STIM_STAT_all(conv_integer(CH_reg)) <= '0';
372|     elsif (set_STIM_STAT='1') then
373|         STIM_STAT_all(conv_integer(CH_reg)) <= '1';
374|     end if;
375| end if;
376| end process;
377|
378| -- STIME: Stimulus enable bit
379| process(CLK,RESET,rst_STIME,set_STIME)
380| begin
381|     if (CLK'event and CLK='1') then
382|         if (RESET='1' or rst_STIME='1') then
383|             STIME <= '0';
384|         elsif set_STIME = '1' then
385|             STIME <= '1';
386|         end if;
387|     end if;
388| end process;
389|
390| end RTL;

```

A.3 Logic Unit

```

1| -- logic.vhd
2| -- MPS VHDL Description
3| -- data unit
4| -- Colin Loisele
5| -- version 1.2 -- 18/04/1999
6| --     Replaced IR with PCR.
7| -- version 1.1 -- 09/08/1997
8| --     Minor syntax errors corrected.  Successful vhdlan.
9| -- version 1.0 -- 02/16/97
10| --     The logic unit contains the combinational logic to generate
11| --     a status vector for the control unit. Each bit in the status
12| --     vector either describes the current status of a data signal
13| --     or represents a software command decoded from the data unit.
14| --
15| --     The STATUS vector is defined as follows:
16| --     STATUS(0): word received (programming mode status)
17| --     STATUS(1): command detected (programming mode status)
18| --     STATUS(2): reset (programming mode command)
19| --     STATUS(3): load (programming mode command)
20| --     STATUS(4): start stimulation (programming mode command)
21| --     STATUS(5): begin loop (stimulation mode command)
22| --     STATUS(6): end loop (stimulation mode command)
23| --     STATUS(7): last instruction (stimulation mode command)
24| --     STATUS(8): loop in progress (stimulation mode status)
25| --     '1' when count is not zero
26| --     STATUS(9): count is not valid (programming mode status)
27| --     '1' when count is odd, ie COUNT(0)='1'
28| --     STATUS(10): count is zero (both modes, status)
29| --     STATUS(11): last channel, ie CH=9 (both modes, status)
30| --     STATUS(12): data request (stimulation mode status)
31| --     STATUS(13): channel stimulation status (both modes, status)
32| --     STATUS(14): end of stimulation (stimulation mode status)
33| --     all stim status bits are zero (all channels
34| --     have been run) and all channel driver DREQ
35| --     lines are '1' (all pulses have been fully
36| --     processed)
37| --     STATUS(15): stimulation enable (both modes, status)
38| --     STATUS(16): accumulator is zero (programming mode status)
39| --
40| --
41| library IEEE;
42| use IEEE.std_logic_1164.all;
43| use IEEE.std_logic_unsigned.all;
44|
45| entity LOGIC_UNIT is
46|   port(REC      : in  std_logic_vector(15 downto 0);
47|        PCR      : in  std_logic_vector(15 downto 0);
48|        ACC      : in  std_logic_vector(15 downto 0);
49|        COUNT    : in  std_logic_vector(7  downto 0);
50|        CH       : in  std_logic_vector(3  downto 0);

```

```

51|         STIM_STAT_all: in  std_logic_vector(0 to 9);
52|         DREQ_all      : in  std_logic_vector(0 to 9);
53|         WR            : in  std_logic;
54|         STIME         : in  std_logic;
55|         STATUS        : out std_logic_vector(0 to 16));
56| end LOGIC_UNIT;
57|
58| architecture RTL of LOGIC_UNIT is
59|     signal STIM_STAT: std_logic;
60|     signal DREQ: std_logic;
61|
62| begin
63|     -- Demux DREQ_all and STIM_STAT_all.
64|     DREQ <= DREQ_all(conv_integer(CH));
65|     STIM_STAT <= STIM_STAT_all(conv_integer(CH));
66|
67|     -- Status Vector
68|     STATUS(0) <= WR;
69|     STATUS(1) <= '1' when REC(11 downto 8)="1110" else '0';
70|     STATUS(2) <= '1' when REC(7 downto 6)="00" else '0';
71|     STATUS(3) <= '1' when REC(7 downto 6)="10" else '0';
72|     STATUS(4) <= '1' when REC(7 downto 6)="01" else '0';
73|     STATUS(5) <= '1' when PCR(10 downto 9)="10" else '0';
74|     STATUS(6) <= '1' when PCR(10 downto 9)="01" else '0';
75|     STATUS(7) <= '1' when PCR(8)='0' else '0';
76|     STATUS(8) <= '0' when COUNT="00000000" else '1';
77|     STATUS(9) <= '1' when COUNT(0)='1' else '0';
78|     STATUS(10) <= '1' when COUNT="00000000" else '0';
79|     STATUS(11) <= '1' when CH="1001" else '0';
80|     STATUS(12) <= DREQ;
81|     STATUS(13) <= STIM_STAT;
82|     STATUS(14) <= '1' when (DREQ_all="1111111111" and
83|         STIM_STAT_all="0000000000") else '0';
84|     STATUS(15) <= STIME;
85|     STATUS(16) <= '1' when ACC="0000000000000000" else '0';
86| end RTL;

```

A.4 Control Unit

```

1| -- control_onehot.vhd
2| --
3| -- MPS VHDL Description
4| -- control unit
5| -- version 2.3 -- 04/10/1999
6| --     Additional state (LoadPCR&PDR) added between ReadMemory and LoadD=IR
7| --     to allow the memory output to stabilize before loading into PCR
8| --     and PDR.
9| -- version 2.2 -- 17/04/1999
10| --     Added en_MEM, I(34), to control signal outputs.
11| --     Added input/output signal mapping in comments.
12| --     State FFs changes to negative-edge triggered.
13| -- version 2.11 -- 13/03/1999
14| --     Added state titles comments.
15| -- version 2.1 -- 09/08/1997
16| --     Minor syntax changes. control_vec added to package typedef.
17| -- version 2.0 -- 14/01/97
18| --     Replaces the separate prog.vhd and stim.vhd with a single entity
19| --     using the one-hot model.
20| -- Colin Loisel
21| --
22| -- MPS control unit, one-hot state assignment with asynchronous reset
23| --
24| -- The control unit is a Moore-type finite-state-machine (FSM). The
25| -- status inputs are grouped in the input vector I(0:16). The control
26| -- outputs are grouped in the output vector O(0:33). The mapping of
27| -- these vectors to meaningful signal names is commented below:
28| --
29| --     I(0):  WR
30| --     I(1):  COMMAND
31| --     I(2):  RESET
32| --     I(3):  LOAD
33| --     I(4):  STIM
34| --     I(5):  BEGIN_LOOP
35| --     I(6):  END_LOOP
36| --     I(7):  LAST_INSTRUCTION
37| --     I(8):  LOOP
38| --     I(9):  COUNT_VALID
39| --     I(10): COUNT_ZERO
40| --     I(11): LAST_CH
41| --     I(12): DREQ
42| --     I(13): STIM_STAT
43| --     I(14): DONE_STIM
44| --     I(15): STIME
45| --     I(16): ACC_ZERO
46| --
47| --     O(0):  rst_STIM_STAT_all
48| --     O(1):  rst_STIM_STAT
49| --     O(2):  rst_STIME
50| --     O(3):  rst_ACC

```



```

51| --      O(4):  rst_INS
52| --      O(5):  rst_COUNT
53| --      O(6):  rst_CH
54| --      O(7):  rst_CSR_all
55| --      O(8):  set_STIM_STAT
56| --      O(9):  set_STIME
57| --      O(10): wr_CH
58| --      O(11): wr_INS
59| --      O(12): wr_BRANCH
60| --      O(13): wr_COUNT
61| --      O(14): wr_PCR
62| --      O(15): wr_PDR
63| --      O(16): wr_CSR
64| --      O(17): wr_MEM
65| --      O(18): wr_DIR
66| --      O(19): wr_AMP
67| --      O(20): wr_DUR
68| --      O(21): rd_REC
69| --      O(22): rd_INS
70| --      O(23): rd_BRANCH
71| --      O(24): rd_COUNT
72| --      O(25): rd_PCR
73| --      O(26): rd_PDR
74| --      O(27): rd_CSR
75| --      O(28): rd_MEM
76| --      O(29): dec_COUNT
77| --      O(30): inc_INS
78| --      O(31): inc_CH
79| --      O(32): en_BRANCH
80| --      O(33): xor_ACC
81| --      O(34): en_MEM
82|
83| library IEEE;
84| use IEEE.std_logic_1164.all;
85| use IEEE.std_logic_arith.all;
86|
87| package typedef is
88|     subtype state_vec is std_logic_vector(0 to 26);
89|     subtype control_vec is std_logic_vector(0 to 34);
90| end typedef;
91|
92| library IEEE;
93| use IEEE.std_logic_1164.all;
94| use IEEE.std_logic_unsigned.all;
95| use IEEE.std_logic_arith.all;
96| use work.typedef.all;
97|
98| entity CONTROL is
99|     port (CLK      : in  std_logic;
100|          RESET: in  std_logic;
101|          I       : in  std_logic_vector(0 to 16);  -- Status inputs
102|          O       : out std_logic_vector(0 to 34)); -- Control outputs
103| end CONTROL;

```

```

104|
105|architecture ONE_HOT of CONTROL is
106|  signal STATE, NEXT_STATE: state_vec;
107|  signal Iin: bit_vector(0 to 16);
108|
109|begin
110|
111|-- Convert std_logic signals to bit_vectors used in state
112|-- machine
113|process(I)
114|begin
115|  Iin <= to_bitvector(I);
116|end process;
117|
118|-- State machine
119|GenerateNextState: process (Iin, state)
120|begin
121|  -- Default value
122|  NEXT_STATE <= STATE_VEC'(others => '0');
123|
124|-- S0: ProgInit - Initialize programming mode
125|-- reset all status bits
126|-- reset stim enable flag
127|--
128|  if STATE(0) = '1' then
129|    NEXT_STATE(1) <= '1';
130|  end if;
131|
132|-- S1: WaitForCommand
133|  if STATE(1) = '1' then
134|    if (Iin(0) = '1' and Iin(1) = '1') then
135|      NEXT_STATE(2) <= '1';
136|    else
137|      NEXT_STATE(1) <= '1';
138|    end if;
139|  end if;
140|
141|-- S2: DecodeCommand
142|  if STATE(2) = '1' then
143|    if (Iin(2) = '1') then
144|      NEXT_STATE(0) <= '1';
145|    elsif (Iin(3) = '1' and Iin(9) = '0') then
146|      NEXT_STATE(3) <= '1';
147|    elsif (Iin(4) = '1') then
148|      NEXT_STATE(13) <= '1';
149|    else
150|      NEXT_STATE(1) <= '1';
151|    end if;
152|  end if;
153|
154|-- S3: WaitForPCR
155|  if STATE(3) = '1' then
156|    if (Iin(0) = '1') then

```

```

157|     NEXT_STATE(4) <= '1';
158|     else
159|         NEXT_STATE(3) <= '1';
160|     end if;
161| end if;
162|
163|-- S4: LoadPCR
164| if STATE(4) = '1' then
165|     NEXT_STATE(5) <= '1';
166| end if;
167|
168|-- S5: WaitForPDR
169| if STATE(5) = '1' then
170|     if (Iin(0) = '1') then
171|         NEXT_STATE(6) <= '1';
172|     else
173|         NEXT_STATE(5) <= '1';
174|     end if;
175| end if;
176|
177|-- S6: LoadPDR
178| if STATE(6) = '1' then
179|     NEXT_STATE(7) <= '1';
180| end if;
181|
182|-- S7: WriteToMemory
183| if STATE(7) = '1' then
184|     if (Iin(10) = '1') then
185|         NEXT_STATE(9) <= '1';
186|     else
187|         NEXT_STATE(8) <= '1';
188|     end if;
189| end if;
190|
191|-- S8: IncInstructionAddress
192| if STATE(8) = '1' then
193|     NEXT_STATE(3) <= '1';
194| end if;
195|
196|-- S9: WaitForChecksum
197| if STATE(9) = '1' then
198|     if (Iin(0) = '1') then
199|         NEXT_STATE(10) <= '1';
200|     else
201|         NEXT_STATE(9) <= '1';
202|     end if;
203| end if;
204|
205|-- S10: LoadChecksum
206| if STATE(10) = '1' then
207|     NEXT_STATE(11) <= '1';
208| end if;
209|

```

```

210|-- S11: VerifyChecksum
211|  if STATE(11) = '1' then
212|    if (Iin(16) = '1') then
213|      NEXT_STATE(12) <= '1';
214|    else
215|      NEXT_STATE(1) <= '1';
216|    end if;
217|  end if;
218|
219|-- S12: SetStim
220|  if STATE(12) = '1' then
221|    NEXT_STATE(1) <= '1';
222|  end if;
223|
224|-- S13: StimInit
225|  if STATE(13) = '1' then
226|    NEXT_STATE(14) <= '1';
227|  end if;
228|
229|-- S14: TestChannel
230|  if STATE(14) = '1' then
231|    if (Iin(14) = '1') then
232|      NEXT_STATE(0) <= '1';
233|    elsif (Iin(12) = '1' and Iin(13) = '1') then
234|      NEXT_STATE(15) <= '1';
235|    else
236|      NEXT_STATE(25) <= '1';
237|    end if;
238|  end if;
239|
240|-- S15: LoadStatus
241|  if STATE(15) = '1' then
242|    NEXT_STATE(16) <= '1';
243|  end if;
244|
245|-- S16: ReadMemory
246|  if STATE(16) = '1' then
247|    NEXT_STATE(17) <= '1';
248|  end if;
249|
250|-- S17: LoadPCR&PDR
251|  if STATE(17) = '1' then
252|    NEXT_STATE(18) <= '1';
253|  end if;
254|
255|-- S18: LoadDir
256|  if STATE(18) = '1' then
257|    if (Iin(5) = '1' and Iin(8) = '0') then
258|      NEXT_STATE(19) <= '1';
259|    else
260|      NEXT_STATE(20) <= '1';
261|    end if;
262|  end if;

```

```

263|
264|-- S19: StartRepeat
265|   if STATE(19) = '1' then
266|       NEXT_STATE(20) <= '1';
267|   end if;
268|
269|-- S20: LoadAmpDur
270|   if STATE(20) = '1' then
271|       if (Iin(6) = '1' and Iin(8) = '1') then
272|           NEXT_STATE(21) <= '1';
273|       elsif (Iin(7) = '1' and Iin(8) = '0') then
274|           NEXT_STATE(24) <= '1';
275|       else
276|           NEXT_STATE(22) <= '1';
277|       end if;
278|   end if;
279|
280|-- S21: GoToBranch
281|   if STATE(21) = '1' then
282|       NEXT_STATE(23) <= '1';
283|   end if;
284|
285|-- S22: IncInstructionAddress
286|   if STATE(22) = '1' then
287|       NEXT_STATE(23) <= '1';
288|   end if;
289|
290|-- S23: SaveStatus
291|   if STATE(23) = '1' then
292|       NEXT_STATE(25) <= '1';
293|   end if;
294|
295|-- S24: ResetStimStat
296|   if STATE(24) = '1' then
297|       NEXT_STATE(25) <= '1';
298|   end if;
299|
300|-- S25: IncChannelAddress
301|   if STATE(25) = '1' then
302|       if (Iin(15) = '0' and Iin(14) = '0' and Iin(11) = '1') then
303|           NEXT_STATE(26) <= '1';
304|       else
305|           NEXT_STATE(14) <= '1';
306|       end if;
307|   end if;
308|
309|-- S26: SetSTIME
310|   if STATE(26) = '1' then
311|       NEXT_STATE(14) <= '1';
312|   end if;
313|end process GenerateNextState;
314|
315|-- Generate output control signals

```

```

316|GenerateOutput: process(STATE)
317|begin
318|
319|  -- default value is '0'
320|  O <= control_vec'(others => '0');
321|
322|-- S0: ProgInit - Initialize programming mode
323|  if STATE(0) = '1' then
324|    O(0) <= '1';
325|    O(2) <= '1';
326|  end if;
327|
328|-- S1: WaitForCommand
329|  if STATE(1) = '1' then
330|    O(3) <= '1';
331|    O(4) <= '1';
332|    O(5) <= '1';
333|  end if;
334|
335|-- S2: DecodeCommand
336|  if STATE(2) = '1' then
337|    O(33) <= '1';
338|    O(21) <= '1';
339|    O(10) <= '1';
340|    O(13) <= '1';
341|  end if;
342|
343|-- S3: WaitForPCR
344|  if STATE(3) = '1' then
345|  end if;
346|
347|-- S4: LoadPCR
348|  if STATE(4) = '1' then
349|    O(33) <= '1';
350|    O(21) <= '1';
351|    O(14) <= '1';
352|    O(29) <= '1';
353|  end if;
354|
355|-- S5: WaitForPDR
356|  if STATE(5) = '1' then
357|  end if;
358|
359|-- S6: LoadPDR
360|  if STATE(6) = '1' then
361|    O(33) <= '1';
362|    O(21) <= '1';
363|    O(15) <= '1';
364|    O(29) <= '1';
365|  end if;
366|
367|-- S7: WriteToMemory
368|  if STATE(7) = '1' then

```

```

369|    O(17) <= '1';
370|    O(34) <= '1';
371|  end if;
372|
373|-- S8: IncInstructionAddress
374|  if STATE(8) = '1' then
375|    O(30) <= '1';
376|  end if;
377|
378|-- S9: WaitForChecksum
379|  if STATE(9) = '1' then
380|  end if;
381|
382|-- S10: LoadChecksum
383|  if STATE(10) = '1' then
384|    O(33) <= '1';
385|  end if;
386|
387|-- S11: VerifyChecksum
388|  if STATE(11) = '1' then
389|  end if;
390|
391|-- S12: SetStim
392|  if STATE(12) = '1' then
393|    O(8) <= '1';
394|  end if;
395|
396|-- S13: StimInit
397|  if STATE(13) = '1' then
398|    O(2) <= '1';
399|    O(6) <= '1';
400|    O(7) <= '1';
401|  end if;
402|
403|-- S14: TestChannel
404|  if STATE(14) = '1' then
405|  end if;
406|
407|-- S15: LoadStatus
408|  if STATE(15) = '1' then
409|    O(27) <= '1';
410|    O(11) <= '1';
411|    O(12) <= '1';
412|    O(13) <= '1';
413|  end if;
414|
415|-- S16: ReadMemory
416|  if STATE(16) = '1' then
417|    O(28) <= '1';
418|    O(34) <= '1';
419|  end if;
420|
421|-- S17: LoadPCR&PDR

```

```

422|  if STATE(17) = '1' then
423|      O(14) <= '1';
424|      O(15) <= '1';
425|      O(28) <= '1';
426|      O(34) <= '1';
427|  end if;
428|
429|-- S18: LoadDir
430|  if STATE(18) = '1' then
431|      O(25) <= '1';
432|      O(18) <= '1';
433|  end if;
434|
435|-- S19: StartRepeat
436|  if STATE(19) = '1' then
437|      O(25) <= '1';
438|      O(13) <= '1';
439|      O(22) <= '1';
440|      O(12) <= '1';
441|      O(32) <= '1';
442|  end if;
443|
444|-- S20: LoadAmpDur
445|  if STATE(20) = '1' then
446|      O(26) <= '1';          -- rd_PDR <= '1';
447|      O(19) <= '1';          -- wr_AMP <= '1';
448|      O(20) <= '1';          -- wr_DUR <= '1';
449|  end if;
450|
451|-- S21: GoToBranch
452|  if STATE(21) = '1' then
453|      O(23) <= '1';          -- rd_BRANCH <= '1';
454|      O(11) <= '1';          -- wr_INS <= '1';
455|      O(32) <= '1';          -- en_BRANCH <= '1';
456|  end if;
457|
458|-- S22: IncInstructionAddress
459|  if STATE(22) = '1' then
460|      O(30) <= '1';          -- inc_INS <= '1';
461|  end if;
462|
463|-- S23: SaveChannelStatus
464|  if STATE(23) = '1' then
465|      O(22) <= '1';          -- rd_INS <= '1';
466|      O(23) <= '1';          -- rd_BRANCH <= '1';
467|      O(24) <= '1';          -- rd_COUNT <= '1';
468|      O(16) <= '1';          -- wr_CSR <= '1';
469|  end if;
470|
471|-- S24: DisableChannel
472|  if STATE(24) = '1' then
473|      O(1) <= '1';          -- rst_STIM_STAT <= '1';
474|  end if;

```



```

475|
476|-- S25: IncChannelAddress
477|  if STATE(25) = '1' then
478|    O(31) <= '1';          -- inc_CH <= '1';
479|  end if;
480|
481|-- S26: StartStim
482|  if STATE(26) = '1' then
483|    O(9) <= '1';          -- set_STIME <= '1';
484|  end if;
485|end process GenerateOutput;
486|
487|-- build the state flip-flops
488|InstantiateFFs: process(CLK,RESET)
489|begin
490|  if RESET = '1' then
491|    STATE <= state_vec'(others => '0');
492|    STATE(0) <= '1';
493|  elsif (CLK = '0' and CLK'event) then
494|    STATE <= NEXT_STATE;
495|  end if;
496|end process InstantiateFFs;
497|
498|end ONE_HOT;

```

A.5 Channel Driver

```

1| -- chdriver.vhd
2| --
3| -- MPS VHDL Description
4| -- channel driver top level
5| -- Colin Loisel
6| -- version 1.3 -- 04/10/1999
7| -- Updated channel driver structural description to accomodate
8| -- active low DREQN signal from driver data unit.
9| -- version 1.2 -- 20/04/1999
10| -- Changed DATABUS to DRVBUS.
11| -- version 1.1 -- 10/08/1997
12| -- Consolidated output DAC amplitude, direction, and on/off control
13| -- into record type DRV_RECORD, located in mps_pack.
14| -- version 1.0 -- 09/08/1997
15| -- Structural implementation of the channel driver incorporates the
16| -- channel driver control unit and data unit blocks.
17| --
18| --
19| library work;
20| use work.all;
21|
22| library IEEE;
23| use IEEE.std_logic_1164.all;
24| use IEEE.std_logic_unsigned.all;
25| use IEEE.std_logic_arith.all;
26|
27| use work.mps_pack.all;
28|
29| entity DRIVER is
30|     port(CLK      : in std_logic;
31|          RESET    : in std_logic;
32|          DRVBUS   : in std_logic_vector(15 downto 0);
33|          wr_AMP   : in std_logic;
34|          wr_DIR   : in std_logic;
35|          STIME    : in std_logic;
36|          STIM_STAT : in std_logic;
37|          DREQ     : out std_logic;
38|          DRV_OUT  : out DRV_RECORD);
39| end DRIVER;
40|
41| architecture STRUCTURAL of DRIVER is
42|     signal AMP: std_logic_vector(7 downto 0);
43|     signal DIR: std_logic_vector(3 downto 0);
44|     signal DREQN, DREQ_int: std_logic;
45|     signal wr_REGS: std_logic;
46|     signal dec_DUR: std_logic;
47|     signal DAC_EN: std_logic;
48|     signal DAC_EN_reg: std_logic;
49|     signal DUR_ZERO: std_logic;
50|

```

```

51| component DRV_CONTROL
52| port(CLK      : in std_logic;
53|      RESET    : in std_logic;
54|      STIME     : in std_logic;
55|      STIM_STAT: in std_logic;
56|      DREQ      : in std_logic;
57|      DUR_ZERO  : in std_logic;
58|      DAC_EN    : out std_logic;
59|      wr_REGS   : out std_logic;
60|      dec_DUR   : out std_logic);
61| end component;
62|
63| component DRV_DATA
64| port(CLK      : in std_logic;
65|      RESET    : in std_logic;
66|      DAC_EN    : in std_logic;
67|      wr_REGS   : in std_logic;
68|      dec_DUR   : in std_logic;
69|      DRVBUS    : in std_logic_vector(15 downto 0);
70|      wr_AMP    : in std_logic;
71|      wr_DIR    : in std_logic;
72|      DREQN     : out std_logic;
73|      DUR_ZERO  : out std_logic;
74|      AMP       : out std_logic_vector(7 downto 0);
75|      DIR       : out std_logic_vector(3 downto 0);
76|      DAC_EN_reg: out std_logic);
77| end component;
78|
79| begin
80|   CNTRL: DRV_CONTROL port map (CLK, RESET, STIME, STIM_STAT, DREQ_int,
81|     DUR_ZERO, DAC_EN, wr_REGS, dec_DUR);
82|
83|   DATA: DRV_DATA port map (CLK, RESET, DAC_EN, wr_REGS, dec_DUR,
84|     DRVBUS, wr_AMP, wr_DIR, DREQN, DUR_ZERO, AMP, DIR,
85|     DAC_EN_reg => DAC_EN_reg);
86|
87|   DREQ_int <= not DREQN;
88|   DREQ <= DREQ_int;
89|
90|   -- Assign DAC amplitude, DAC ON/OFF, and current direction controls
91|   -- to the driver output record.
92|   DRV_OUT.AMP <= AMP;
93|   DRV_OUT.DAC_EN <= DAC_EN_reg;
94|   DRV_OUT.ASRC <= DIR(0);
95|   DRV_OUT.AON <= DIR(1);
96|   DRV_OUT.BSRC <= DIR(2);
97|   DRV_OUT.BON <= DIR(3);
98|
99| end STRUCTURAL;

```

A.5.1 Channel Driver Data Unit

```

1| -- drvdata.vhd
2| --
3| -- MPS VHDL Description
4| -- channel driver data unit
5| -- Colin Loisel
6| -- version 1.6 -- 11/10/1999
7| --     Fixed duration counter.  It used to increment instead of
8| --     decrement.
9| -- version 1.5 -- 04/10/1999
10| --     Changed DREQ to DREQN (active low) to correct a problem that
11| --     resulted in none of the channel drivers ever requesting data.
12| --     For this reason, the set/reset logic for the DREQN FF was
13| --     updated to reset on wr_REGS and set on wr_AMP.
14| -- version 1.4 -- 20/04/1999
15| --     Corrected DREQ to reset on wr_AMP rather than wr_DIR.
16| --     Changed DATABUS to DRVBUS.
17| -- version 1.3.1 -- 05/04/1999
18| --     DUR_ZERO logic brought to top of architecture from the bottom.
19| -- version 1.3 -- 17/03/1999
20| --     Changed DREQ register to a D flip-flop with asynchronous RESET.
21| -- version 1.2 -- 09/08/1997
22| --     Call up std_logic_unsigned package instead of std_logic_arith
23| --     in order to call up arithmetic operations (+ and *) for std_
24| --     logic_vector data types.  Had trouble with type conversion to
25| --     and from unsigned data types.  Analyzed successfully.
26| -- version 1.11 -- 30/07/1997
27| --     Some syntax changes made after initial pass through vhdlan.
28| --     Still 4 errors present.
29| -- version 1.1
30| --     Removed CTRL(0..3) in port definition and replaced with
31| --     DAC_EN, wr_REGS, and dec_DUR.  Removed corresponding signal
32| --     mapping from architecture.
33| -- version 1.0 -- 05/05/1997
34| --
35| library IEEE;
36| use IEEE.std_logic_1164.all;
37| use IEEE.std_logic_unsigned.all;
38|
39| entity DRV_DATA is
40|     port(CLK      : in std_logic;
41|          RESET    : in std_logic;
42|          DAC_EN   : in std_logic;
43|          wr_REGS  : in std_logic;
44|          dec_DUR  : in std_logic;
45|          DRVBUS   : in std_logic_vector(15 downto 0);
46|          wr_AMP   : in std_logic;
47|          wr_DIR   : in std_logic;
48|          DREQN    : out std_logic;
49|          DUR_ZERO : out std_logic;
50|          AMP      : out std_logic_vector(7 downto 0);

```

```

51|         DIR          : out std_logic_vector(3 downto 0);
52|         DAC_EN_reg: out std_logic);
53| end DRV_DATA;
54|
55| architecture RTL of DRV_DATA is
56|     signal DUR_buf: std_logic_vector(7 downto 0);
57|     signal AMP_buf: std_logic_vector(7 downto 0);
58|     signal DIR_buf: std_logic_vector(3 downto 0);
59|     signal DUR: std_logic_vector(11 downto 0);
60|
61| begin
62|     -- Logic to derive status signals
63|     process(DUR)
64|     begin
65|         if DUR="000000000000" then
66|             DUR_ZERO <= '1';
67|         else
68|             DUR_ZERO <= '0';
69|         end if;
70|     end process;
71|
72|     -- DUR_buf: Buffer to DUR register
73|     -- 8-bit register with sync reset
74|     process(CLK,RESET,wr_AMP,DRVBUS)
75|     begin
76|         if (CLK'event and CLK='1') then
77|             if RESET='1' then
78|                 DUR_buf <= "00000000";
79|             elsif wr_AMP='1' then
80|                 DUR_buf <= DRVBUS(7 downto 0);
81|             end if;
82|         end if;
83|     end process;
84|
85|     -- AMP_buf: Buffer to AMP register
86|     -- 8-bit register with sync reset
87|     process(CLK,RESET,wr_AMP,DRVBUS)
88|     begin
89|         if (CLK'event and CLK='1') then
90|             if RESET='1' then
91|                 AMP_buf <= "00000000";
92|             elsif wr_AMP='1' then
93|                 AMP_buf <= DRVBUS(15 downto 8);
94|             end if;
95|         end if;
96|     end process;
97|
98|     -- DIR_buf: Buffer to DIR register
99|     -- 4-bit register with sync reset
100|     process(CLK,RESET,wr_DIR,DRVBUS)
101|     begin
102|         if (CLK'event and CLK='1') then
103|             if RESET='1' then

```

```

104|         DIR_buf <= "0000";
105|     elsif wr_DIR='1' then
106|         DIR_buf <= DRVBUS(15 downto 12);
107|     end if;
108| end if;
109| end process;
110|
111| -- DUR: Pulse DURation counter
112| -- 8-bit down counter with sync reset
113| process(CLK,RESET,wr_REGS,dec_DUR)
114| -- variable DUR_var: std_logic_vector(11 downto 0);
115| begin
116|     if (CLK'event and CLK='1') then
117|         if RESET='1' then
118|             DUR <= "000000000000";
119|         elsif wr_REGS='1' then
120|             DUR <= DIR_buf & "1111";
121|         elsif dec_DUR='1' then
122|             DUR <= DUR - "1";
123|         end if;
124|     end if;
125| end process;
126|
127| -- AMP: Pulse AMPlitude register
128| -- 8-bit register with sync reset
129| process(CLK,RESET,wr_REGS,AMP_buf)
130| begin
131|     if (CLK'event and CLK='1') then
132|         if RESET='1' then
133|             AMP <= "00000000";
134|         elsif wr_REGS='1' then
135|             AMP <= AMP_buf;
136|         end if;
137|     end if;
138| end process;
139|
140| -- DIR: Pulse DIRection register
141| -- 4-bit register with sync reset
142| process(CLK,RESET,wr_REGS,DIR_buf)
143| begin
144|     if (CLK'event and CLK='1') then
145|         if RESET='1' then
146|             DIR <= "0000";
147|         elsif wr_AMP='1' then
148|             DIR <= DIR_buf;
149|         end if;
150|     end if;
151| end process;
152|
153| -- DREQN: Data REquest (NOT) flag
154| -- Flip-Flop with async RESET
155| process(RESET,wr_AMP,wr_REGS)
156| begin

```

```

157|     if (RESET='1' or wr_REGS='1') then
158|         DREQN <= '0';
159|     elsif (wr_AMP='1') then
160|         DREQN <= '1';
161|     end if;
162| end process;
163|
164| -- DAC_EN_reg: DAC_EN_register
165| -- Use a FF to retime DAC_EN output
166| process(CLK,RESET,DAC_EN)
167| begin
168|     if (CLK'event and CLK='1') then
169|         if RESET='1' then
170|             DAC_EN_reg <= '0';
171|         else
172|             DAC_EN_reg <= DAC_EN;
173|         end if;
174|     end if;
175| end process;
176|
177| end RTL;

```

A.5.2 Channel Driver Control Unit

```

1| -- drvctrl.vhd
2| --
3| -- MPS VHDL Description
4| -- channel driver control unit
5| -- version 1.5 -- 11/10/1999
6| --     Updated process sensitivity list on state machine process to
7| --     include input status signals.
8| --     Update control logic for S3.  Go to S0 if STIME='0', not '1'.
9| -- version 1.4 -- 07/10/1999
10| --     Changed package "typedef" to "drvtypedef" to eliminate confusion
11| --     similar package in core control unit.
12| --     Changed state transition logic in S0 - go to S1 when DREQ='0',
13| --     not '1'.
14| -- version 1.3.1 -- 20/04/1999
15| --     Cleaned up some commenting.
16| -- version 1.3 -- 09/08/1997
17| --     Call up std_logic_unsigned instead of std_logic_arith.
18| -- version 1.2 -- 30/07/1997
19| --     First pass analyzation by vhdlan pointed out a number of syntax
20| --     errors and undeclared signals.  These were corrected.
21| -- version 1.1 -- 27/07/1997
22| --     Replaced status (I(0..3)) and control (O(0..2)) signals with
23| --     meaningful signal names in the port definition and architecture.
24| -- version 1.0 -- 17/04/97
25| --     The channel driver control unit controls the DAC enable output
26| --     bit and the data transfers from the channel driver buffers to
27| --     the registers. The control unit is a Moore state machine,
28| --     one-hot implementation with asynchronous reset.

```

```

29| -- Colin Loiselle
30| --
31| --
32| library IEEE;
33| use IEEE.std_logic_1164.all;
34| use IEEE.std_logic_unsigned.all;
35|
36| package drvtypedef is
37|     subtype state_vec is std_logic_vector (0 to 3);
38| end drvtypedef;
39|
40| library IEEE;
41| use IEEE.std_logic_1164.all;
42| use IEEE.std_logic_arith.all;
43| use work.drvtypedef.all;
44|
45| entity DRV_CONTROL is
46|     port (CLK      : in std_logic;
47|           RESET    : in std_logic;
48|           STIME     : in std_logic;
49|           STIM_STAT : in std_logic;
50|           DREQ      : in std_logic;
51|           DUR_ZERO  : in std_logic;
52|           DAC_EN    : out std_logic;
53|           wr_REGS   : out std_logic;
54|           dec_DUR   : out std_logic);
55| end DRV_CONTROL;
56|
57| architecture ONE_HOT of DRV_CONTROL is
58|     signal STATE, NEXT_STATE: state_vec;
59|
60| begin
61|
62| -- State machine
63| process (state, DREQ, DUR_ZERO, STIM_STAT, STIME)
64| begin
65|     -- Default value
66|     NEXT_STATE <= STATE_VEC'(others => '0');
67|
68| -- S0: STOP
69| -- disable DAC and pulse duration counter
70| -- goto LOAD if buffers have been loaded (DREQ=0) and registers are
71| -- empty (DUR=0)
72| -- goto START if STIME=1, STIM_STAT=1, and DUR_ZERO=0 (ie. valid data
73| -- in registers)
74| if STATE(0) = '1' then
75|     if (DREQ = '0' and DUR_ZERO = '1') then
76|         NEXT_STATE(1) <= '1';
77|     elsif (STIME = '1' and STIM_STAT = '1' and DUR_ZERO = '0') then
78|         NEXT_STATE(2) <= '1';
79|     else
80|         NEXT_STATE(0) <= '1';
81|     end if;

```



```

82|   end if;
83|
84| -- S1: LOAD
85| -- load buffer values into registers
86| -- if STIME=1 then go back to RUNning state, else goto STOP
87|   if STATE(1) = '1' then
88|     if (STIME = '1') then
89|       NEXT_STATE(3) <= '1';
90|     else
91|       NEXT_STATE(0) <= '1';
92|     end if;
93|   end if;
94|
95| -- S2: START
96| -- start stimulation by enebbling the DAC output
97| -- go to RUN to start decrementing the counter
98|   if STATE(2) = '1' then
99|     NEXT_STATE(3) <= '1';
100|   end if;
101|
102| -- S3: RUN
103|   if STATE(3) = '1' then
104|     -- if (STIME = '0') or (DUR_ZERO = '1' and DREQ = '1') then
105|     if (DUR_ZERO = '1' and DREQ = '1') then
106|       NEXT_STATE(0) <= '1';
107|     elsif (DUR_ZERO = '1' and DREQ = '0') then
108|       NEXT_STATE(1) <= '1';
109|     else
110|       NEXT_STATE(3) <= '1';
111|     end if;
112|   end if;
113| end process; -- End of State Machine
114|
115| -- Generate output control signals
116|
117| process(STATE)
118| begin
119|
120|   -- default value is '0'
121|   DAC_EN <= '0';
122|   wr_REGS <= '0';
123|   dec_DUR <= '0';
124|
125|   if STATE(1) = '1' then
126|     wr_REGS <= '1';
127|   end if;
128|
129|   if STATE(2) = '1' then
130|     DAC_EN <= '1';
131|   end if;
132|
133|   if STATE(3) = '1' then
134|     DAC_EN <= '1';

```

```
135|     dec_DUR <= '1';
136|   end if;
137|end process;
138|
139|-- build the state flip-flops
140|process(CLK,RESET)
141|begin
142|   if RESET = '1' then
143|     STATE <= state_vec'(others => '0');
144|     STATE(0) <= '1';
145|   elsif (CLK = '0' and CLK'event) then
146|     STATE <= NEXT_STATE;
147|   end if;
148|end process;
149|
150|end ONE_HOT;
```

A.6 Driver Control/Status Signal Mux/Demux

```

1| -- drv_mux.vhd
2| -- MPS VHDL Description
3| -- channel driver control/status signal mux/dmux
4| -- Colin Loisel
5| -- version 1.1 -- 18/04/1999
6| --   Re-ordered entity ports.
7| -- version 1.0 -- 21/08/1997
8| library IEEE;
9| use IEEE.std_logic_1164.all;
10| use IEEE.std_logic_unsigned.all;
11| use IEEE.std_logic_arith.all;
12|
13| use work.all;
14| use work.mps_pack.all;
15|
16| entity DRV_MUX is
17|   port(CH      : in  std_logic_vector(3 downto 0); -- channel address
18|         C1      : in  std_logic;
19|         C2      : in  std_logic;
20|         S1_all: in  std_logic_vector(0 to 9);
21|         C1_all: out std_logic_vector(0 to 9);
22|         C2_all: out std_logic_vector(0 to 9);
23|         S1      : out std_logic);
24| end DRV_MUX;
25|
26| architecture RTL of DRV_MUX is
27|
28| begin
29|   process(CH,C1)
30|   begin
31|     -- Set selected bit to '1', all others to '0'
32|     for indx in 0 to 9 loop
33|       if indx = conv_integer(CH) then
34|         C1_all(indx) <= C1;
35|       else
36|         C1_all(indx) <= '0';
37|       end if;
38|     end loop;
39|   end process;
40|
41|   process(CH,C2)
42|   begin
43|     -- Set selected bit to '1', all others to '0'
44|     for indx in 0 to 9 loop
45|       if indx = conv_integer(CH) then
46|         C2_all(indx) <= C2;
47|       else
48|         C2_all(indx) <= '0';
49|       end if;
50|     end loop;

```

```
51|   end process;
52|
53|
54|   -- Select bit from vector
55|   S1 <= S1_all(conv_integer(CH));
56| end RTL;
```

A.7 RAM Collar

```

1| -- spram_collar.vhd
2| --
3| -- MPS VHDL Description
4| -- single-port RAM collar
5| -- Colin Loisel
6| -- version 1.0 -- 17/04/1999
7| -- The RAM collar provides an interface between the vendor specific
8| -- SPRAM description and the rest of the MPS top level.
9| --
10| library IEEE;
11| use IEEE.std_logic_1164.all;
12| use IEEE.std_logic_unsigned.all;
13|
14| entity SPRAM_COLLAR is
15|     port(CLK      : in std_logic;
16|          ADDR     : in std_logic_vector(7 downto 0);      -- address bus
17|          DATA    : inout std_logic_vector(31 downto 0); -- data bus
18|          en_MEM   : in std_logic;                        -- RAM enable, active high
19|          rd_MEM   : in std_logic;                        -- read enable, active high
20|          wr_MEM   : in std_logic);                       -- write enable, active high
21| end SPRAM_COLLAR;
22|
23| architecture RTL of SPRAM_COLLAR is
24|     signal en_MEMN, rd_MEMN: std_logic;
25|
26|     component SPRAM160x32
27|         port(A      : in std_logic_vector(7 downto 0); -- address bus
28|              CLK    : in std_logic;
29|              D      : in std_logic_vector(31 downto 0); -- data bus in
30|              EZ     : in std_logic;                    -- RAM enable, active low
31|              WZ     : in std_logic;                    -- write low, read high
32|              GZ     : in std_logic;                    -- output enable, active low
33|              Q      : out std_logic_vector(31 downto 0)); -- data bus out
34|     end component;
35|
36| begin
37|     MEM: SPRAM160x32 port map (ADDR, CLK, DATA, en_MEMN, rd_MEM, rd_MEMN,
38|                               DATA);
39|
40|     en_MEMN <= not en_MEM;
41|     rd_MEMN <= not rd_MEM;
42|
43| end RTL;

```



```

15|
16| component C_TOP
17|     port (CLK          : in std_logic ;
18|           RESET        : in std_logic ;
19|           DATA_in     : in std_logic_vector(15 downto 0) ;
20|           WR           : in std_logic ;
21|           ELECTRODE    : out DRV_RECORD_ARRAY(0 to 9) );
22| end component;
23| for all : C_TOP use entity WORK.TOP (STRUCTURE);
24|
25| signal CLK          : std_logic;
26| signal RESET        : std_logic;
27| signal DATA_in     : std_logic_vector(15 downto 0);
28| signal WR           : std_logic;
29| signal ELECTRODE    : DRV_RECORD_ARRAY(0 to 9);
30| begin
31|
32|     TESTED: C_TOP
33|     port map (
34|         CLK => CLK,
35|         RESET => RESET,
36|         DATA_in => DATA_in,
37|         WR => WR,
38|         ELECTRODE => ELECTRODE);
39|
40| end test_TOP;

```


Appendix B

PERL Pulse Compiler

Appendix B contains PERL script that takes a simple set of stimulation pulse descriptions and generates the 16-bit pulse description vectors with an associated time stamp to be sent to the MPS D_in port.

B.1 Example Pulse Description

```
41| reset
42| load channel 5
43|   repeat 4
44|     pulse 50 111 pos
45|     pulse  0 55 off
46|     pulse 50 111 neg
47|   end_repeat
48|   pulse 100 1000 pos
49|   pulse  50 2000 neg
50| end_channel 5
51| load channel 3
52|   pulse 100 10 neg
53|   pulse 200 20 neg
54|   pulse 100 10 neg
55|   pulse  0 10 off
56|   pulse 100 10 pos
57|   pulse 200 20 pos
58|   pulse 100 10 pos
59| end_channel 3
60| start stimulation
61| load channel 5
62|   repeat 4
63|     pulse 50 111 pos
64|     pulse  0 55 off
65|     pulse 50 111 neg
66|   end_repeat
67|   pulse 100 1000 pos
68|   pulse  50 2000 neg
69| end_channel 5
70| start stimulation
```

B.2 PERL Pulse Compiler

```

1| #!/usr/local/bin/perl -w
2| #
3| ## Documentation here!
4| ##
5| ## This script compiles the stimulation description file and
6| ## creates the binary file to download to the MPS.
7| ##
8| ## arguments: stim_file
9| ##
10| ##
11| ##**** Subroutines to convert between binary and decimal ****##
12| sub bittodec {
13|     unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
14| }
15|
16| sub dectobit {
17|     my $decimal = shift;
18|     my $range = shift;
19|     print "    --- Converting decimal value \"$decimal\" into
20|           $range-bit binary...\n";
21|     if ( 2**$range > $decimal ) {
22|         substr(unpack("B32", pack("N", $decimal)), -$range);
23|     }
24|     else {
25|         die "    *** ERROR: $range bits is too small to represent the
26|             value \"$decimal\".\n";
27|         print "    *** ERROR: $range bits is too small to represent the
28|             value \"$decimal\".\n";
29|     }
30| }
31|
32| sub max {
33|     my $max = shift(@_);
34|     foreach $val (@_) {
35|         $max = $val if ($val > $max);
36|     }
37|     return $max;
38| }
39|
40|
41| ##***** Set up input and output files *****##
42| print "\n";
43| $stim_file_in = shift;
44| @file_tmp = split(/\/\./, $stim_file_in);
45| print "file_root = $file_tmp[0]\n";
46| $stim_file_out = $file_tmp[0] . ".out";
47|
48| print "Processing file: $stim_file_in\n";
49| print "Sending output to: $stim_file_out\n";
50|

```

```

51| open (OUT_FILE, ">$stim_file_out");
52|
53| ## Slurp in stimulation file
54| @stim_desc = split(/\n/, `cat $stim_file_in`);
55|
56| ##### Variable Initialiaztion #####
57| ## Create a hash to correlate polarity to DIR description
58| %dir_options = ('pos', '1110',
59|                'neg', '1011',
60|                'off', '0000');
61| ## Constants
62| $CLK_PER = 10000;
63| $WORD_PER = $CLK_PER * 16;
64| $EOF_WAIT = $CLK_PER * 84;
65| $SETUP_TIME = 100;
66|
67| ## Counters for use in the loop
68| $line_index = 0;          ## used to identify current line number
69| $output_index = 0;        ## used to indicate current line in output array
70| $word_count = 0;          ## word count for block download
71| $channel_time = 0;        ## tracks stimulation time of a channel, in us
72| $frame_time = 0;          ## frame length, in us
73|
74| ## Flags
75| $repeat_in_progress = 0;
76| $channel_in_progress = 0;
77| $repeat_first = "1";
78| $repeat_count = 1;
79| $repeat_count_bit = "00000000";
80|
81| ##### Loop through each line in the file #####
82| foreach $line (@stim_desc) {
83|     ++$line_index;
84|     print "+++ LINE ($line_index): $line\n";
85|
86|     @args = split(/\s/, $line);
87|     $num_args = @args;
88|     print "    --- Number of arguments = $num_args:";
89|     foreach $arg (@args) { print "$arg:"; }
90|     print "\n";
91|
92|     if ( $num_args > 0 ) {
93|
94|         SWITCH: {
95|             ## pulse [amplitude] [duration] [polarity | special bxxxx]
96|             if ( lc($line) =~ /pulse\s+(\d+)\s+(\d+)\s+(\w+)\s*(\w*)/ ) {
97|                 print "    --- $1,$2,$3,$4\n";
98|                 $channel_time += ($repeat_count * $2);
99|
100|                 $pdw_bin = '';
101|                 vec($pdw_bin, 0, 8) = ($1 / 10); ## in 10s of uA
102|                 vec($pdw_bin, 1, 8) = ($2 / 10); ## in 10s of us
103|                 $polarity = $3;

```

```

104|
105|         if ( lc($polarity) eq "special" ) {
106|             $dir_bit_string = $4;
107|         }
108|         else {
109|             $dir_bit_string = $dir_options{$polarity};
110|         }
111|
112|             $prog_word = $dir_bit_string . "0" . $repeat_first .
113|             "01" . $repeat_count_bit;
114|             print "    --- PCW = $prog_word\n";
115|             push(@prog_array, $prog_word);
116|             push(@time_array, $WORD_PER);
117|             ++$word_count;
118|             ++$soutput_index;
119|
120|             $prog_word = unpack("B*", $pdw_bin);
121|             print "    --- PDW = $prog_word\n";
122|             push(@prog_array, $prog_word);
123|             push(@time_array, $WORD_PER);
124|             ++$word_count;
125|             ++$soutput_index;
126|             print "    --- WORD COUNT = $word_count\n";
127|
128|             $repeat_first = 0;
129|             $repeat_count_bit = "00000000";
130|         last SWITCH;
131|     }
132|
133|     ## repeat [pulse_repeat_count (1..)]
134|     if ( lc($line) =~ /repeat\s+(\d+)/ ) {
135|         $repeat_count = $1;
136|         $repeat_count_bit = dectobit($repeat_count, 8);
137|         print "    --- COUNT = $repeat_count_bit\n";
138|         $repeat_first = 1;
139|         $repeat_in_progress = 1;
140|         last SWITCH;
141|     }
142|
143|     ## end_repeat
144|     if ( lc($line) =~ /end_repeat/ ) {
145|         $prog_word = $prog_array[$soutput_index - 2];
146|         print "    --- change $prog_word to ";
147|         $prog_word |= "0000001000000000";
148|         print "$prog_word\n";
149|         $prog_array[$soutput_index - 2] = $prog_word;
150|
151|         ## Reset flags and counters
152|         $repeat_count = 1;
153|         $repeat_count_bit = "00000000";
154|         $repeat_in_progress = 0;
155|         last SWITCH;
156|     }

```

```

157|
158|     ## reset
159|     if ( lc($line) =~ /reset/ ) {
160|         $prog_word = "10001110000000000";
161|         print "    --- RESET = $prog_word\n";
162|         push(@prog_array, $prog_word);
163|         push(@time_array, $WORD_PER);
164|         ++$output_index;
165|         last SWITCH;
166|     }
167|
168|     ## start
169|     if ( lc($line) =~ /start/ ) {
170|         $prog_word = "00001110010000000";
171|         print "    --- START = $prog_word";
172|         $frame_end_pause = $EOF_WAIT + $frame_time*10000;
173|         print ", and wait $frame_end_pause us\n";
174|         push(@prog_array, $prog_word);
175|         push(@time_array, $frame_end_pause);
176|         ++$output_index;
177|         last SWITCH;
178|     }
179|
180|     ## load channel [0..9]
181|     if ( lc($line) =~ /load\s+channel\s+(\d)/ ) {
182|         $channel = $1;
183|         $prog_word = dectobit($channel, 4) . "111010";
184|         print "    --- LOAD = $prog_word" . "xxxxxx\n";
185|
186|         ## Add LOAD CHANNEL command word without instruction count
187|         push(@prog_array, $prog_word);
188|         push(@time_array, $WORD_PER);
189|         ++$output_index;
190|
191|         ## Set flags, counters, and mark beginning of channel
192|         ## in programming array.
193|         $channel_in_progress = 1;
194|         $channel_start_index = $output_index - 1;
195|         $checksum = 0;
196|         $word_count = 0;
197|         $channel_time = 0;
198|         last SWITCH;
199|     }
200|
201|     ## end_channel
202|     if ( lc($line) =~ /end_channel/ ) {
203|
204|         ## Add channel word count to load channel command
205|         print "    --- Prog array index = $output_index; ";
206|         print "Channel word count = $word_count\n";
207|         $prog_word = $prog_array[$channel_start_index];
208|         print "    --- LOAD = $prog_word\n";
209|         $prog_word = $prog_word . dectobit($word_count, 6);

```

```

210|         print "    --- LOAD = Sprog_word\n";
211|         Sprog_array[Schannel_start_index] = Sprog_word;
212|
213|         ## Add LAST INSTRUCTION indicator to previous PCW
214|         Sprog_word = Sprog_array[Soutput_index - 2];
215|         print "    --- change Sprog_word ";
216|         Sprog_word &= "1111111011111111";
217|         print "to Sprog_word\n";
218|         Sprog_array[Soutput_index - 2] = Sprog_word;
219|
220|         ## Calculate checksum
221|         $checksum = 0;
222|         print "    --- Calculating checksum...\n";
223|         for ($i = Schannel_start_index;
224|             $i < Soutput_index; $i++) {
225|             print dectobit($checksum,16) . " xor\n" .
226|                 Sprog_array[$i];
227|             $checksum = $checksum ^
228|                 bittodec(Sprog_array[$i]);
229|             print dectobit($checksum,16) . "\n";
230|             ##print Sprog_array[$i] . "\n";
231|         }
232|
233|         ## Add checksum to array
234|         push(@sprog_array, dectobit($checksum,16));
235|         push(@time_array, SWORD_PER);
236|         ++$soutput_index;
237|         print "    --- Finish channel Schannel;
238|             LOAD = Sprog_array[Schannel_start_index];
239|             WORD COUNT = $sword_count;
240|             CHECKSUM = $checksum\n";
241|
242|         ## Re-initialize flags and counters
243|         $checksum = 0;
244|         $channel_in_progress = 0;
245|         $sword_count = 0;
246|         $sframe_time = max($channel_time, $sframe_time);
247|         $channel_time = 0;
248|
249|         last SWITCH;
250|     }
251|
252|     print "    *** ERROR: Syntax error at line $sline_index: \n
253|         \"$sline\"\n";
254| }
255| }
256| }
257|
258| ## Send to output file
259| $time_stamp = 0;
260| print OUT_FILE "@$time_stamp = 0000000000000000\n";
261| for ($i = 1; $i < $soutput_index; $i++) {
262|     $time_stamp = $time_array[$i-1];

```

```

263|    print OUT_FILE "$time_stamp = Sprog_array[$i]\n";
264|    print "line $i: Sprog_array[$i]\n";
265|};
266|
267|Stime_stamp = 0;
268|Stime_stamp_off = SCLK_PER;
269|print OUT_FILE "\@$time_stamp = 1\nStime_stamp_off = 0\n";
270|for (Si = 1; Si < $output_index; Si++) {
271|    Stime_stamp = $time_array[$i-1] - SCLK_PER;
272|    Stime_stamp_off = SCLK_PER;
273|    print OUT_FILE "$time_stamp = 1\nStime_stamp_off = 0\n";
274|};
275|close( OUT_FILE );

```

B.3 Generated Output Vectors

```

1| @0 = 0000000000000000
2| 160000 = 0101111010001010
3| 160000 = 1110010100000100
4| 160000 = 0000010100001011
5| 160000 = 0000000100000000
6| 160000 = 0000000000000101
7| 160000 = 1011001100000000
8| 160000 = 0000010100001011
9| 160000 = 1110000100000000
10| 160000 = 0000101001100100
11| 160000 = 1011000000000000
12| 160000 = 0000010111001000
13| 160000 = 0101011100100111
14| 160000 = 0011111010001110
15| 160000 = 1011000100000000
16| 160000 = 0000101000000001
17| 160000 = 1011000100000000
18| 160000 = 0001010000000010
19| 160000 = 1011000100000000
20| 160000 = 0000101000000001
21| 160000 = 0000000100000000
22| 160000 = 0000000000000001
23| 160000 = 1110000100000000
24| 160000 = 0000101000000001
25| 160000 = 1110000100000000
26| 160000 = 0001010000000010
27| 160000 = 1110000000000000
28| 160000 = 0000101000000001
29| 160000 = 0110111010001111
30| 160000 = 0000111001000000
31| 4192000 = 0101111010001010
32| 160000 = 1110010100000100
33| 160000 = 0000010100001011
34| 160000 = 0000000100000000
35| 160000 = 0000000000000101
36| 160000 = 1011001100000000

```

```
37| 160000 = 0000010100001011
38| 160000 = 1110000100000000
39| 160000 = 0000101001100100
40| 160000 = 1011000000000000
41| 160000 = 0000010111001000
42| 160000 = 0101011100100111
43| 160000 = 0000111001000000
```


References

- [AG93] J. R. Armstrong and F. G. Gray. *Structured Logic Design with VHDL*. Englewood Cliffs, New Jersey: Prentice-Hall, 1993.
- [AZS97] S.U. Ay, F.G. Zeng, and B.J. Sheu. Hearing with Bionic Ears: Speech Processing Strategies for Cochlear Implant Devices. *IEEE Circuits and Devices Magazine*, 13(3):18-23, 1997.
- [BFM+92] J-M. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL Designer's Reference*. Dordrecht: Kluwer Academic Publishers, 1992.
- [BT77] S. B. Brummer and M. J. Turner. Electrochemical considerations for safe electrical stimulation of the nervous system with platinum electrodes. *IEEE Transactions on Biomedical Engineering*, 24(01):59-63, 1977.
- [C+87] G. M. Clark, et al. The University of Melbourne-Nucleus Multi-Electrode Cochlear Implant. *Advances in Oto-Rhino-Laryngology*, Vol 38, C. R. Pfaltz (Ed.), Karger, Basel, 1987.
- [Cla93] G. M. Clark. The University of Melbourne/Nucleus Multiple-Channel Cochlear Implant. *Acoustics Australia*, 21(3):91-97, 1993.
- [COM+95] C. H. Chouard, M. Ouayoun, B. Meyer, and C. Fugain. Speech coding strategies of the Digisonic fully digitized cochlear implant. *Acta Oto-Laryngologica (Stockh)*, 115(2):264-268, 1995.

- [CTP90] G. M. Clark, Y. C. Tong, J. F. Patrick. *Cochlear Prostheses*. London: Churchill Livingstone, 1990.
- [FF85] R.J. Fretz and R.P. Fravel. Design and Function: A Physical and Electrical Description on the 3M House Cochlear Implant System. *Ear and Hearing*, 6(3): 14S-19S, 1985.
- [FWW89] C. C. Finley, B. S. Wilson, and M. W. White. Models of neural responsiveness to electrical stimulation. In *Models of the Electrically Stimulated Cochlea*, ed. J. M. Miller and F. A. Spelman, New York: Springer-Verlag, 1989.
- [GC87] J. Génin, R. Charachon. Experimental basis and design of a new cochlear prosthesis system. *Annals of Otology, Rhinology, and Laryngology (Suppl. 128)*, 96(1,pt2):76-79, 1987.
- [Ged90] I. J. Gedeon. Cepstral analysis: A speech processing strategy for the cochlear implant. Master's thesis, Carleton University, Ottawa, Canada, 1990.
- [GMW75] T. R. Gheewala, R. D. Melen, and R. L. White. A CMOS implantable multi-electrode auditory stimulator for the deaf. *IEEE Journal of Solid State Circuits*, SC-10(6):472-479, 1975.
- [Gol94] S. Golson, Trilobyte Systems. State Machine Design Techniques for Verilog and VHDL. *Synopsys Online Documentation*, v3.3b, Methodology Notes. 224-272, 1994.

- [GTK+88] B. J. Gantz, R. S. Tyler, J. F. Knutson, G. Woodworth, P. Abbas, B. F. McCabe, J. Hinrichs, N. Tye-Murray, C. Lansing, F. Kuk, and C. Brown. Evaluation of five different cochlear implant designs: Audiologic assessment and predictors of performance. *Laryngoscope*, 98(10):1100-1106, 1988.
- [HBM87] R. Hinojosa, R. R. Blough, and E. E. Mhoon. Profound sensorineural deafness: A histopathological study. *Annals of Otolaryngology, Rhinology, and Laryngology (Suppl. 128)*, 96(1,pt2):43-46, 1987.
- [HvdPD+87] R. V. Harrison, P. van der Puije, F. Duval, H. Kunov, and R. Morris. Technical development of an implantable cochlear prosthesis in Canada. *The Journal of Otolaryngology*, 16(5):311-315, 1987.
- [Lit92] B. J. Lithgow. A comparison of burst and amplitude modulated electrical stimulation of the cochlea. *Journal of Clinical Engineering*, 16(3):251-260, 1992.
- [Lit98] B. J. Lithgow. Relaxing cochlear implant electrode design constraints using high pulse rate stimulation. *Proceedings of the 20th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 20(6):3132-3135, 1998.
- [Loe85a] G. E. Loeb. The functional replacement of the ear. *Scientific American*, 252(2):104-111, 1985.

- [Loe85b] G. E. Loeb. Single and multichannel cochlear prostheses: Rationale, strategies, and potential. In *Cochlear Implants*, ed. R. A. Schindler, M. M. Merzenich, 17-28, New York: Raven Press, 1985.
- [Loe90] G. E. Loeb. Cochlear Prosthetics. *Annu. Rev. Neurosci.* 13:357-371, 1990.
- [Loi98] P. C. Loizou. Mimicking the human ear. *IEEE Signal Processing Magazine*, 15(5):101-130, 1998.
- [LWZF99] D. Lawson, B. Wilson, M. Zerbi, C. Finley. Speech processors for auditory prostheses. *Center for Auditory Prosthesis Research, Fourth Quarterly Progress Report*, 1999.
- [MB89] L. R. Morris and P. Barszczewski. Algorithms, hardware, and software for a digital signal processor microcomputer-based speech processor in a multi-electrode cochlear implant system. *IEEE Transactions on Biomedical Engineering*, 36(6):573-584, 1989.
- [McD87] H. McDermott. Cochlear implant for simultaneous multichannel stimulation. *Annals of Otology, Rhinology, and Laryngology (Suppl. 128)*, 96(1,pt2):67-69, 1987.
- [McD89] H. McDermott. An advanced multiple channel cochlear implant. *IEEE Transactions on Biomedical Engineering*, 36(7):220-221, 1987.

- [MW77] M. M. Merzenich and M. W. White. Cochlear implant. The interface problem. In *Functional Electrical Stimulation: Applications in Neural Prostheses*, ed. F. T. Hambrecht and J. B. Reswick, 321-340, New York: Marcel Dekker, 1977.
- [Par85] C. W. Parkins. The bionic ear: Principles and current status of cochlear prostheses. *Neurosurgery*, 16(6):853- 865, 1985.
- [Par89] C. W. Parkins. Temporal response patterns of auditory nerve fibers to electrical stimulation in deafened squirrel monkeys. *Hearing Research*, 41:137-168, 1989.
- [PC87] C. W. Parkins and J. Colombo. Auditory-nerve single-neuron thresholds to electrical stimulation from scala tympani electrodes. *Hearing Research*, 31:267-286, 1987.
- [Pic88] J. O. Pickles *An Introduction to the Physiology of Hearing*. London: Academic Press, 1988.
- [Ric] Ineraid Cochlear Implant System. Richards Medical Company brochure.
- [SDH+92] M. Sawan, F. Duval, M. M. Hassouna, J. Li, M. M. Elhilali, J. Lachance, M. Leclair, S. Pourmehdi, and J. Mouïne. Computerized transcutaneous control of a multichannel implantable urinary prosthesis. *IEEE Transactions on Biomedical Engineering*, 39(6):600-609, 1992.
- [Sim66] F. B. Simmons. Electrical stimulation of the auditory nerve in man. *Arch. Otol.*, 84:24-76, 1966.

- [SK92] R. A. Schindler and D. K. Kessler. Preliminary results with the Clarion cochlear implant. *Laryngoscope*, 102(9):1006-1013, 1992.
- [Syn] Synopsys Online Documentation, version 3.3b. Synopsys Inc.
- [Tyl88] R. Tyler. Open-set recognition with the 3M/Vienna single-channel cochlear implant. *Archives of Otolaryngology, Head and Neck Surgery*, 14:1123-1126, 1988.
- [WCM89] J. A. Wiler, B. M. Clopton, and M. A. Mikhail. Effect of electrical pulse shape on AVCN unit responses to cochlear stimulation. *Hearing Research*, 39:251-261, 1989.
- [WFF+88] B. S. Wilson, C. C. Finley, J. C. Farmer, Jr., D. T. Lawson, B. A. Weber, R. D. Wolford, P. D. Kenan, M. W. White, M. M. Merzenich, and R. A. Schindler. Comparative studies of speech processing strategies for cochlear implants. *Laryngoscope*, 98(10):1069-1077, 1988.
- [Wit92] D. J. Withey. A cochlear implant stimulation strategy and system. Master's Thesis, Carleton University, Ottawa, Canada, 1992.
- [Won91] J. Wong. The Nucleus Cochlear Implant. *Colloquium of the Institution of Electrical Engineers (Great Britian)*, Digest no. 179, 1991.
- [YS79] E. D. Young and M. B. Sachs. Representation of steady state vowels in the temporal aspects of the discharge patterns of populations of auditory-nerve fibers. *Journal of the Acoustical Society of America*, 66(5): 1381-1403, 1979.
- [Zim93] C. Zimmer. Making Senses. *Discover*, 14(6):79-85, 1993.