Demonstration System for a Low-Power Classification Processor

by

David J. Rowe

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2000

Author ....
Department of Electrical Engineering and Computer Science
February 4, 2000

Certified by..
Anantha Chandrakasan
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Demonstration System for a Low-Power Classification Processor

by

David J. Rowe

Submitted to the Department of Electrical Engineering and Computer Science
on February 4, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

Abstract

The purpose of this thesis was to develop a self-contained digital system using an ultra low power classification processor for biomedical sensing applications. The initial application for the system was heartbeat detection.

The system consists of an analog input from an amplified microphone, a digital test data input, a programming interface to the processor, and a data output interface. The completed system demonstrates that a useful digital signal processing algorithm can be implemented at the desired low power constraints. Further work will show that these power levels can be achieved using an ambient vibrational source.

Thesis Supervisor: Anantha Chandrakasan
Title: Associate Professor

Acknowledgments

First and foremost, I would like to thank Professor Anantha Chandrakasan for allowing me to work on this project. His enduring patience and encouragement have been greatly appreciated during the course of this project. I am inspired by the example of his success, but more-so by his dedication to his students. I cannot thank him enough for his support.

This thesis is merely a continuation of the innovation by Raj Amirtharajah. All aspects of the technical ingenuity within this thesis can be entirely attributed to Raj. I very much appreciated the extra time he spent answering questions and helping out in tough situations. I would like to thank him for putting up with my incessant phone calls and emails. I feel privileged to have had to opportunity to work on his project.

The technical and philosophical advice of Manish Bhardwaj and Jim Goodman has been a live-saver during the course of this project. Their modesty would not permit them to recognize their contribution, but their help has been unmeasurable. I would also like to thank the other members of Anantha's group for making the lab a friendly and welcoming place to work, Alice Wang, Rex Min, Eugene Shih, Seong-Hwan Cho, Amit Sinha, Travis Furrer, Charatpong (Boo) Chotigavanich, Wendi Heinzelman, and Vadim Gutnik. A great many thanks to Margaret Flaherty for her assistance during my time in the group.

My most meaningful experience during my tenure at MIT has been being a TA for 6.111. I have learned more through a year of teaching than in four years of undergraduate studies. Thanks to Professor Donald Troxel and Professor James Kirtley for giving me the opportunity to work with them in 6.111. I would also like to thank my fellow TA's, Everest Huang, Rob Jagnow, Alex Ihler, Theresa Huang, Josef Brandriss, Jessica Forbess, and Ali Tariq. Thank you also to Danny Seth and Syed Alam for being good friends and classmates.

The individuals who have inspired me the most during my academic career are Professor Hamid Nawab and Professor Thomas Kincaid from Boston University. Their devotion to teaching is unsurpassed. They are very much responsible for the success I have obtained.

Despite my absence from the usual circle of friends during my thesis work, I very much appreciate the support and friendship of my close friends, Jack Livingston, John Arrigo, Jen Saleem, Man Leung, Vishal Goklani, Mike Chipolone, and Vince Leslie.

Without the love and support of my family, my achievements at MIT and BU could never have been accomplished. My parents, Jack and Linda Rowe, my grandmother, Mary Ruth Ross, my brother, Stephen Rowe, and my cousin, Tim Gumto, have be an ever present source of love, support, and encouragement. I can never thank them enough for all they have done for me. I promise to continue to do my best to make them proud. I love you all very much.

Finally, I would like to thank my love, Kelly Dugan, for her warmth, compassion, and love. She has been there at every moment when I needed her. I can only hope that I will someday be able to repay her for all she has done for me. I love you, Kelly.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, there has been an ever expanding demand for portable devices for such applications as wireless communication and hand-held computers. These systems rely heavily on batteries as their source of energy. Investigation of alternative sources for power generation and storage is a growing endeavor as the limits of device physics are reached with increasing clock rates and lower supply voltages. A potential alternative power generating technique is to convert ambient vibrational energy into useful electrical energy as proposed by Amirtharajah [1][2][3].

One application of such a system is biomedical sensing, for example, heartbeat monitoring. The feasibility of such a system imposes certain constraints on the type of application. In order to attain the low power consumption required by the limitations of the ambient energy converter, the digital system must employ a variety of low power VLSI (Very Large Scale Integration) design techniques. Many of these techniques present a trade-off between speed and power dissipation. A human heartbeat can be measured with a microphone, and be converted into a digital signal that can be analyzed and classified with a digital signal processor (DSP). The classification processor developed by Amirtharajah takes advantage of a multitude of low-power design techniques to implement a heartbeat detection algorithm, while consuming approximately 500nW at 1.5V with a 1 kHz clock frequency.

1.1 Background

1.1.1 Matched Filtering

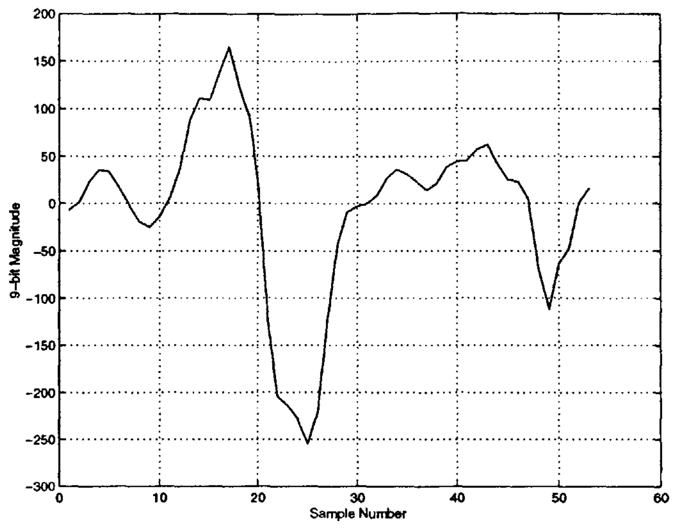


Figure 1-1: Matched Filter Template

Matched filtering is a signal processing technique used to detect a specific pattern in the input signal. A system is designed to have an impulse response, $h[k]$, that is a replica of the desired input signal [4]. A typical matched filter template for a human heartbeat is shown in Figure 1-1. The result of the matched filtering process is a signal, $y[k]$, as shown in Figure 1-2. The signal, $y[k]$, has several characteristics that can be measured, ranked, and classified. [1, pp. 105-125]

These signal features are summarized below:

1. peak correlation output value
2. first valley after peak correlation
3. first valley before peak correlation
4. first peak after peak correlation
5. first peak before peak correlation
6. matched filter output energy
7. peak correlation output width

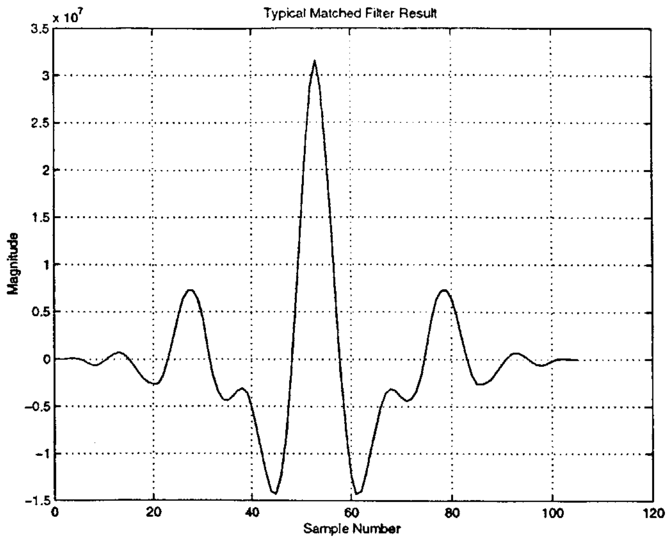


Figure 1-2: Typical Matched Filter Result

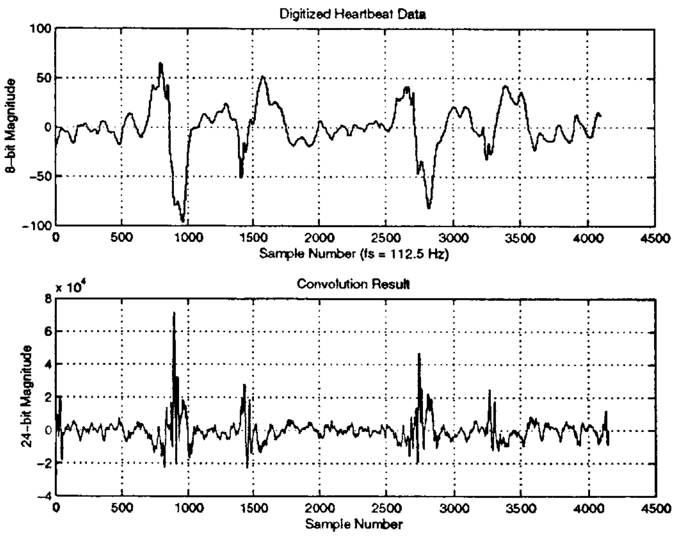


Figure 1-3: Heartbeat Waveform and Matched Filter Result

Figure 1-3 shows a heartbeat signal recorded via an analog acoustic sensor as well as the resultant signal when the heartbeat is passed through the matched filter.

1.1.2 Distributed Arithmetic

Distributed Arithmetic (DA) is a bit-serial operation that computes the inner product of two vectors (one of which is a constant) in parallel without the use of a multiplier [5][6]. The constant vector, with regards to linear filtering, is the impulse response of the filter. The distributed arithmetic calculation is implemented using a read-only memory (ROM) lookup table, an input shift register, and an output accumulator. An advantage of the distributed arithmetic approach, is that an approximate result can be obtained using a lower quantization of the input signal. Using a smaller bit-width yields a trade-off between power reduction and detection accuracy.

Figure 1-4 show a detailed example of a distributed arithmetic computation. The structure shown computes the dot product of a 4-element vector X and a constant vector A. All 16 possible linear combinations of the constant vector elements (A_i) are stored in the ROM. The variable vector X is repackaged to form the ROM address most significant bit first. We have assumed that the X_i elements are 4-bits 2's complement (bit 3 is the sign bit) binary numbers. Every clock cycle, the RESULT register adds $2\times$ its previous value (reset to zero) to the current ROM contents. Moreover, after each cycle the 4 registers that hold the four elements of the X vector are shifted to the right. The sign timing pulse, T_s, is activated when the ROM is addressed by bit 3 of the vector elements (sign). In this case, the adder subtracts the current ROM contents from the accumulator state. After 4 cycles, (4 is the bit-width of the X_i elements) the dot product has been produced within the RESULT register [1, p. 119].

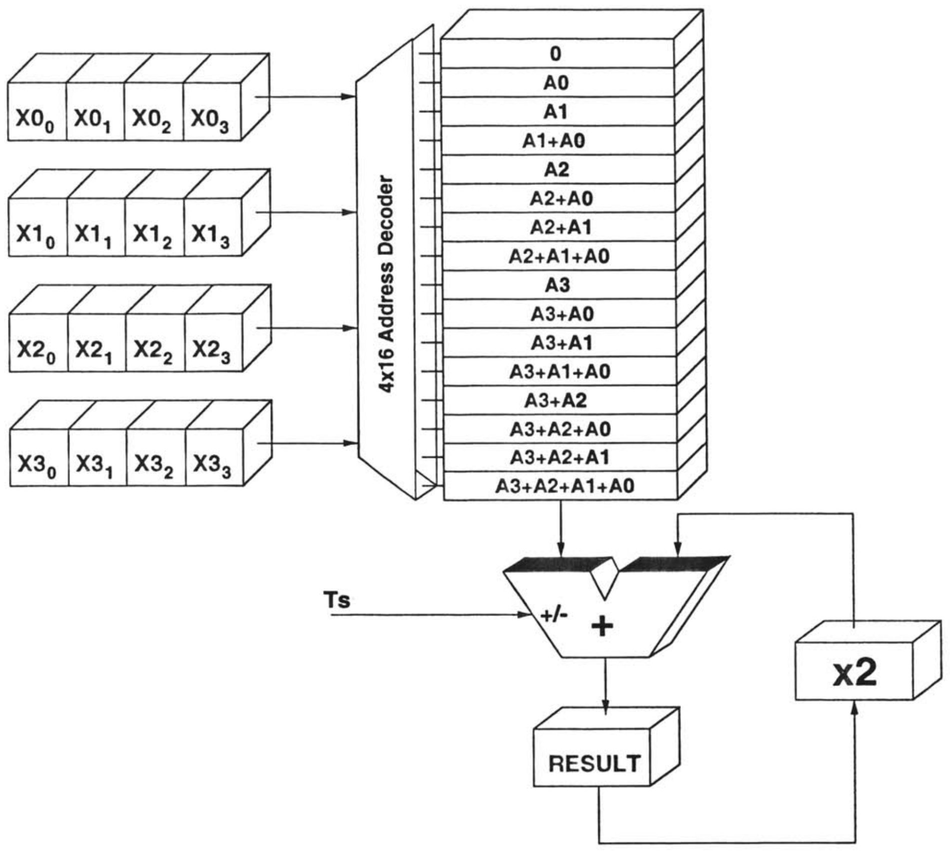


Figure 1-4: Distributed Arithmetic ROM and Accumulator Structure [1, p. 119]

1.2 Overview

The goal of this project was to design and develop a complete, self-enclosed system that uses an ultra low power digital signal processor (DSP) to implement a heartbeat detection algorithm. The heart of the system is a DSP designed by Amirtharajah [1], herein referred to as the SensorDSP chip. A subsequent goal of the project is to prove that a useful digital signal processing system can be implemented at power levels low enough to use ambient vibrational energy as the source of power. The following is a list of requirements for the system:

1. **Programming Interface** - The SensorDSP chip is programmed using a serial boundary-scan register. Matched filter coefficients, micro-controller instructions, and other control words are generated using a computer program and then stored on a programmable read-only memory (PROM). The programming interface transmits data from the PROM into the SensorDSP chip.

2. **Analog Sensor Interface** - Provided with the SensorDSP system is a an analog sensor that the patient would wear on the neck or chest. The analog sensor was provided by the Army Research Laboratory. The analog sensor interface samples data from the analog sensor and sends the data serially into the SensorDSP chip.

3. **Test Data Input** - In addition to an analog data input, the system allows for test data to be stored into a PROM. This data can be serially input into the SensorDSP instead of data from the analog sensor. Although this feature is primarily meant for debugging purposes, it is also useful when devising algorithms for other applications since the algorithms can be tested with known input data.

4. **Detection Logic** - Due to a limited number of pins the chip has a limited data output capability. The SensorDSP has an 8 to 1 multiplexed 12-bit output bus. The detection logic controls the select inputs to the multiplexer and reads the data from the bus. This data can be output to a hexadecimal LED (light

emitting diode) display, or in the case of heartbeat detection, a single LED is flashed when a detection occurs.

5. **Power Measurement/Monitoring** - The system provides two methods of measuring the power consumption of the SensorDSP chip. First, a connection on the circuit board is allocated for an external ammeter to be connected for an accurate power measurement. Second, a power measurement circuit on the board provides a bar-graph display of the power consumption.

6. **Clock Generation** - Although the heartbeat detection system operates at a fixed optimal system clock frequency, the SensorDSP system provides the capability to operate at a wide range of speeds. The clock generation subsystem enables the user to easily change clock speeds.

7. **Battery** - The system runs on a single 9V battery supply. Voltage regulation circuits provide the specific voltages for the components on the board.

This thesis is written such that the reader can gain an understanding of the detection algorithm, the processor architecture, and the system level design, as well as be able to easily setup and use the system. Chapter 2 is a description of the SensorDSP chip architecture. The concentration in this chapter is on the flow of data through the processor. The context of the chapter is within the heartbeat detection algorithm. Chapter 3 is a detailed description of the board level design. A significant amount of detail is provided to enable a user the ability to take advantage of the system's versatility. Detailed information regarding the nuances of the SensorDSP interface is essential in case the user desires to make changes to the board to meet the requirements of a specific application. Chapter 4 contains a step by step set of instructions for the operation of the SensorDSP system. The chapter covers topics ranging from programming the system, to extracting data, to troubleshooting.

Chapter 2

SensorDSP Chip Architecture

The architecture of the SensorDSP chip is tailored to detection and classification algorithms that are applicable to a range of digital signal processing problems [1, p. 129]. The approach is to employ a matched filter implemented with a dedicated linear filter unit. This unit takes advantage of the distributed arithmetic technique discussed in Section 1.1.2. The input signal $x[k]$ is a discrete time waveform that is fed serially into the chip. The linear filter continually computes the filter output $y[k]$. A segmentation procedure is executed by the chip's micro-controller unit. Each segment is long enough to contain an entire heartbeat, but short enough to not contain two heartbeats. Each segment is overlapped to ensure that an error is not made if the crucial portion of the waveform occurred at the edge of a segment. After a segment is filtered through the linear filter, the useful features of the result $y[k]$ are extracted and stored by the micro-controller unit. These features are subsequently ranked and compared against set thresholds to yield a "yes or no" classification of the presence of a heartbeat in the given segment. Figure 2-1 is a detailed block diagram of the typical signal processing that must occur for classification.

The architecture of the chip consists of three major components: a linear filter, a non-linear/short-linear filter, and a feature extraction and classification micro-controller. These subsystems will be discussed in detail in the following sections. Figure 2-2 is a block diagram of the architecture of the SensorDSP chip.

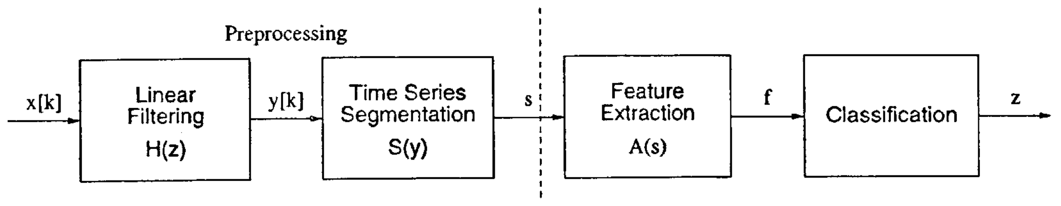
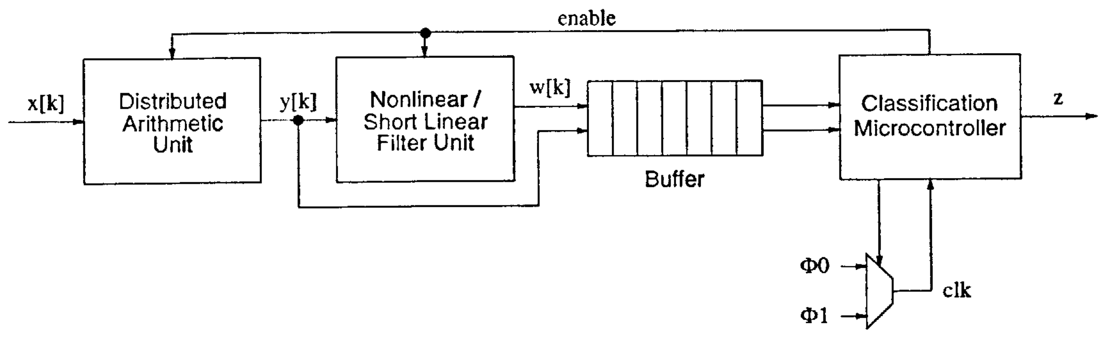


Figure 2-1: Classification and Signal Processing Block Diagram [1, p. 128]



enable
x[k]
Distributed Arithmetic Unit
y[k]
Nonlinear / Short Linear Filter Unit
w[k]
Buffer
Classification Microcontroller
z
Φ0
Φ1
clk

Figure 2-2: Signal Processing Chip Architecture [1, p. 128]

2.1 Distributed Arithmetic Matched Filter

The linear filter is implemented based on a Distributed Arithmetic (DA) technique as discussed in section 1.1.2. Figure 2-3 is a block diagram of the linear filter. By allowing variable input bit-widths of 8, 4, 2, and 1, an approximate filter result can be obtained with less power consumption. For bit-widths that are less than the filter length, using a distributed arithmetic approach is faster than a multiply/accumulate approach. Using distributed arithmetic to compute a matched filter output of an N-bit sample of data, a valid result is obtained every N clock cycles.

The Distributed Arithmetic Unit (DA) consists of 16 distributed arithmetic tables that are addressed via a shift register. This shift register stores up to four samples

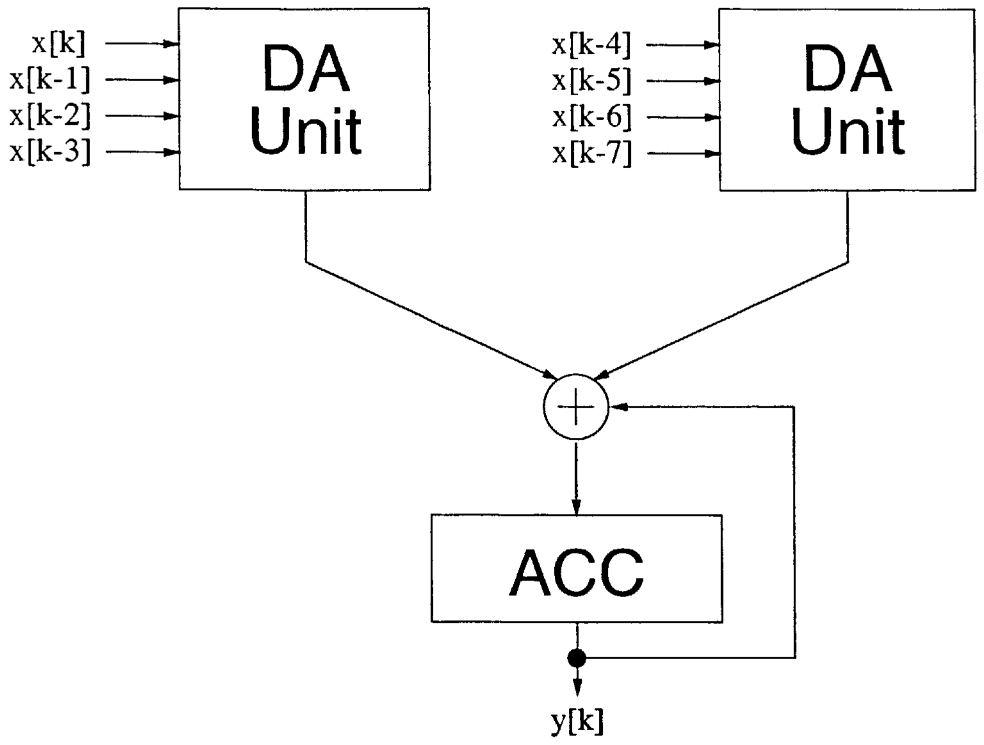


Figure 2-3: Linear Filter Implementation Architecture [1, p. 130]

of the input data. To allow for a variable bit-width, the length of the register is controlled by configuration word provided during device programming. The clock signals to unused portions of the register are gated to reduce power consumption. Figure 2-4 is a diagram of the shift register implementation. The output of the DA tables is delivered to an accumulator. After N clock cycles, where N is the number of bits of the input sample, the accumulator will contain a valid result of the linear filter. The maximum bit-width of the filter coefficients themselves is 9 bits. Since the table is addressed by at most four samples, the DA table will consist of pre-calculated values that are any possible linear combination of four 9-bit numbers. Therefore, each table entry is at most 11 bits.

The accumulator is 24 bits wide to accommodate the partial products of the result. The post-processing data-path of the micro-controller is only 12-bits, therefore the result of the linear filter must be truncated to 12-bits. It is not necessarily true that the high order bits of the filter result are the most crucial. The SensorDSP chip has the capability to select the truncation window during device programming. Since the linear filter is constantly outputting data at a fixed rate, this data needs to be buffered in memory before post-processing. The filter buffer is a reserved portion of the micro-controller unit's data memory.

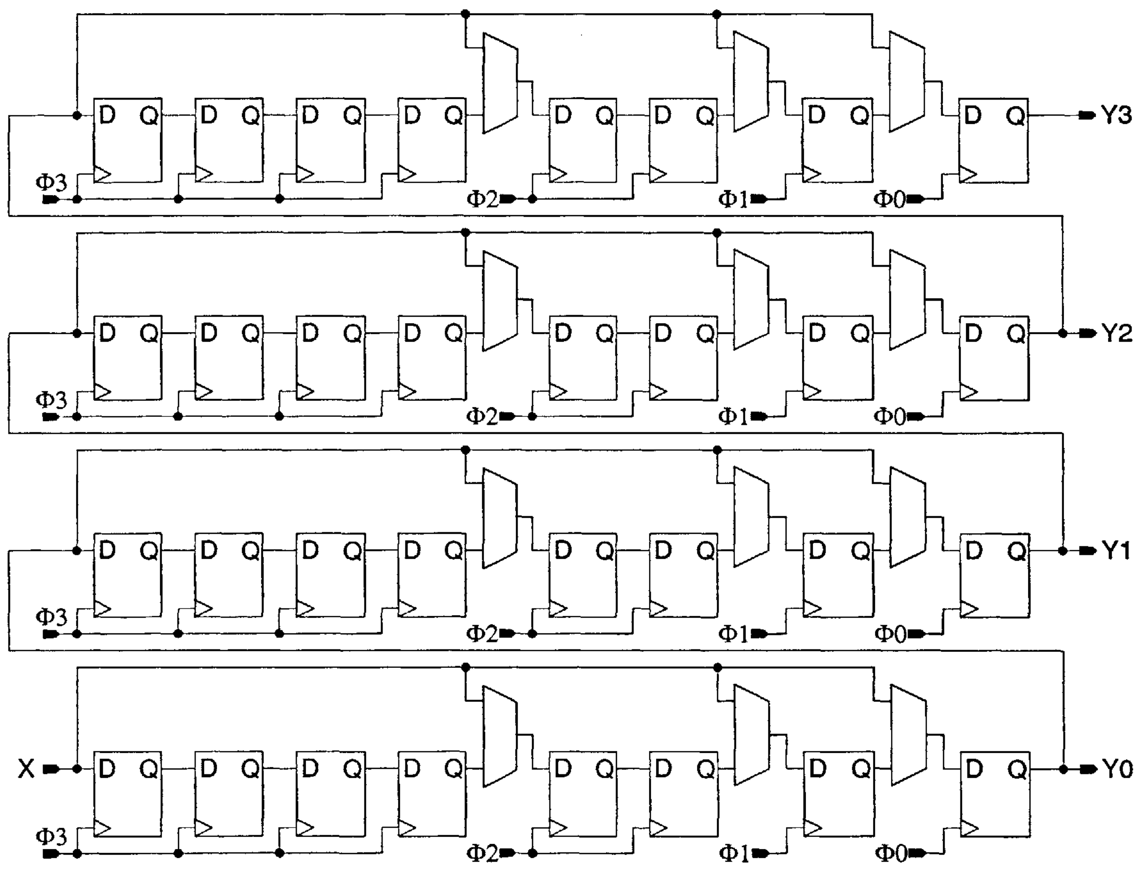


Figure 2-4: DA Unit Shift Register Implementation [1, p. 131]

2.2 Non-Linear/Short Linear Filter

For filters where the bit-width is larger than the filter length, a multiply/accumulate approach is more advantageous. The non-linear/short-linear (NLSL) filter is implemented with a multiply/accumulate architecture. In order to maintain the same clock rate as the DA linear filter, a very long instruction word (VLIW) [7] approach is used within the NLSL unit in order to execute multiple instructions in parallel.[1, p. 130] The maximum number of instructions per sample performed by the NLSL unit is 8 to ensure that it remains synchronized with the output of the matched filter. For smaller bit-widths, the NLSL unit must operate with fewer instructions. Fortunately, the parallel architecture of this unit allows for useful processing in as little as 2 instructions.

The NLSL unit consists of a Square and Accumulate Unit (SAC), a Multiply and Accumulate Unit (MAC), and a Load/Store Unit (LSU). Each unit operates in parallel, with a single instruction execution per clock cycle. Figure 2-5 shows a block diagram of the NLSL unit. The NLSL unit is useful for computing the energy of the matched filter result and for performing additional filtering with a filter that has a short impulse response.

Both the SAC and MAC accumulators are 24 bits wide. Again, since the post-processing data-path is only 12-bits, the results of the NLSL filter must be truncated. The truncation window is selected during device programming. The 12-bit results of the SAC and MAC units are stored in the filter buffer. The micro-controller can then access these values during the feature extraction process.

2.3 Micro-controller Unit

The feature extraction and classification micro-controller is implemented with a standard load/store type of computer architecture [7]. Figure 2-6 is a block diagram of the micro-controller architecture.

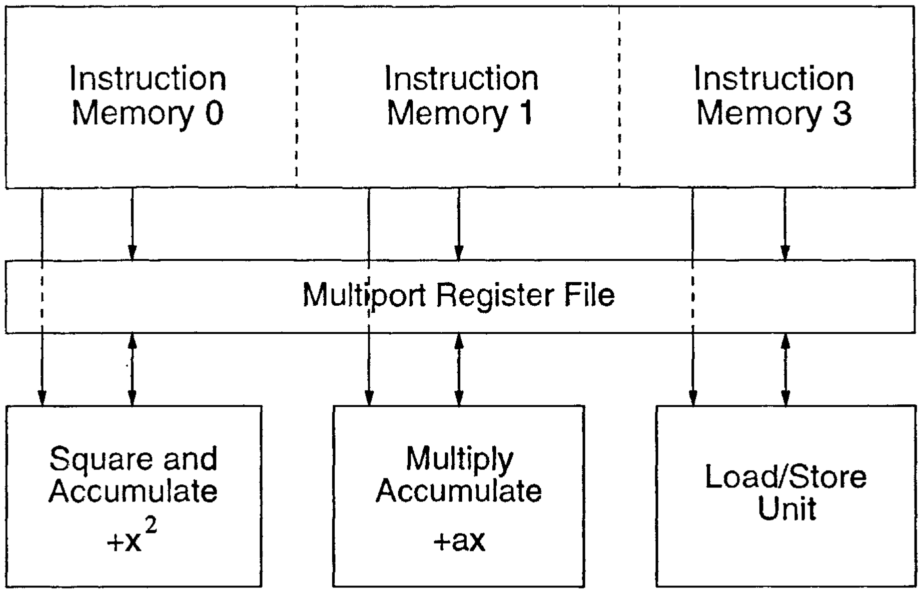


Figure 2-5: NLSL Filter Implementation Architecture [1, p. 131]

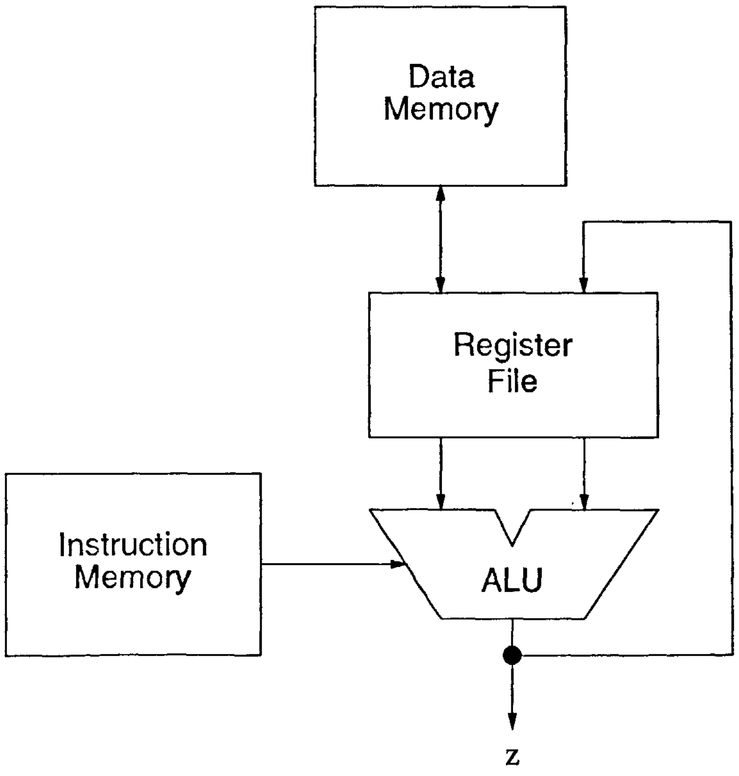


Figure 2-6: Micro-controller Architecture [1, p. 132]

The micro-controller unit consists of a 24-bit accumulator, an 8-bit program counter, a register file, and a data memory block. The 8-bit program counter allows for only 256 possible instructions. This is more than sufficient to perform the desired signal processing application. The register file has eight special purpose registers and 8 general purpose registers. The register file has a dual read port and a single write port.

Unlike the accumulators for the filters, there is no truncation involved with the micro-controller accumulator. Instead, the micro-controller interfaces the accumulator as two 12-bit accumulators. It is up to the user to round or truncate the data as part of the assembly code.

The data memory is accessed via a 12-bit address bus, however only 256 address locations are available for reading and writing. There is an additional 256 locations allocated to the filter buffer.

The purpose of the filter buffer is to store the results of the filter units until the micro-controller can execute the feature extraction and classification algorithm. The filter buffer can only be read by the micro-controller. Writes to the filter buffer are executed automatically with each result from the filters. A dedicated register, BUFPTR, indexes the filter buffer.

It is the micro-controller's responsibility to wait for an entire segment of data from the linear filter to be filled into the filter buffer. When a segment of valid data is available, the micro-controller disables both filter units and switches to a higher clock rate. The reason for the increased clock rate is two-fold. First, the micro-controller must be able to perform the feature extraction and classification algorithm before the next sample of data is received to ensure that samples are not ignored. In actuality, some samples may be lost during the classification process since the duration of the process is data dependent. This does not pose a significant problem since the data segments are over-lapped. Second, since the majority of the processing occurs in the linear and NLSL filters, maintaining a slow clock for the micro-controller reduces the power consumption.

Chapter 3

System Design Description

The system was designed to provide a simple user-friendly interface to the SensorDSP chip. There are three main components to the design: the programming interface, the analog input interface, and the heartbeat indicator. Other supporting components are the clock generation and power measurement circuits. The digital logic for the system is implemented in complex programmable logic devices (CPLD) [8]. A block diagram of the system is shown in Figure 3-1.

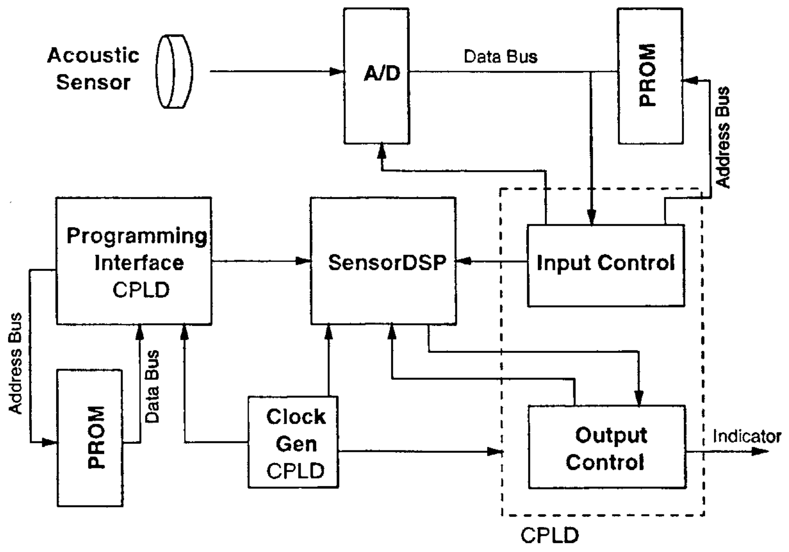


Figure 3-1: System Block Diagram

3.1 Programming Interface

The programming interface provides the distributed arithmetic tables for the linear filter, the instruction memory for the NLSL filter, the instruction memory for the micro-controller, and configuration/control codes. The programming interface consists of a 64K$\times$8 programmable read-only memory (PROM) [9], a parallel to serial shift register, and a finite state machine (FSM) controller.

3.1.1 JTAG Interface

Formally known as IEEE/ANSI standard 1149.1-1990, JTAG is a set of rules formulated by the Joint Test Action Group (JTAG). When applied at the chip level the standard helps reduce the cost of designing, testing and producing integrated circuits [10]. The JTAG standard makes use of a serial Test Access Port (TAP) to supply and extract information from an integrated circuit (IC). This information is used to program, test, and verify the operation of a given IC. In the application of the SensorDSP chip, the TAP is used to supply the chip with distributed arithmetic table entries for the linear filter, instructions for the non-linear/short-linear filter, instructions for the micro-controller, and miscellaneous configuration words. Due to the variety of the information that needs to be provided to the chip, the JTAG standard allows for a simple interface that requires a minimal number of external pins.

The TAP consists of a boundary/scan register and a finite state machine (FSM) controller. The state transition diagram of the FSM is shown in Figure 3-2. A test mode select (TMS) signal causes the TAP FSM to transition from state to state. The boundary/scan register is a shift register that receives its serial input from the "test data input" (TDI) signal. The output of the boundary/scan register is the "test data output" (TDO) signal. The clock signal to the FSM and register is called "test clock" (TCK). A TAP controller has an optional "test reset" (TRST) signal that restores the FSM to the "RUN-TEST/IDLE" state. The FSM changes state on the rising edge of TCK. The control signals TMS and TDI are required to be stable during the rising edge of TCK. To avoid violating setup and hold times of the state machine

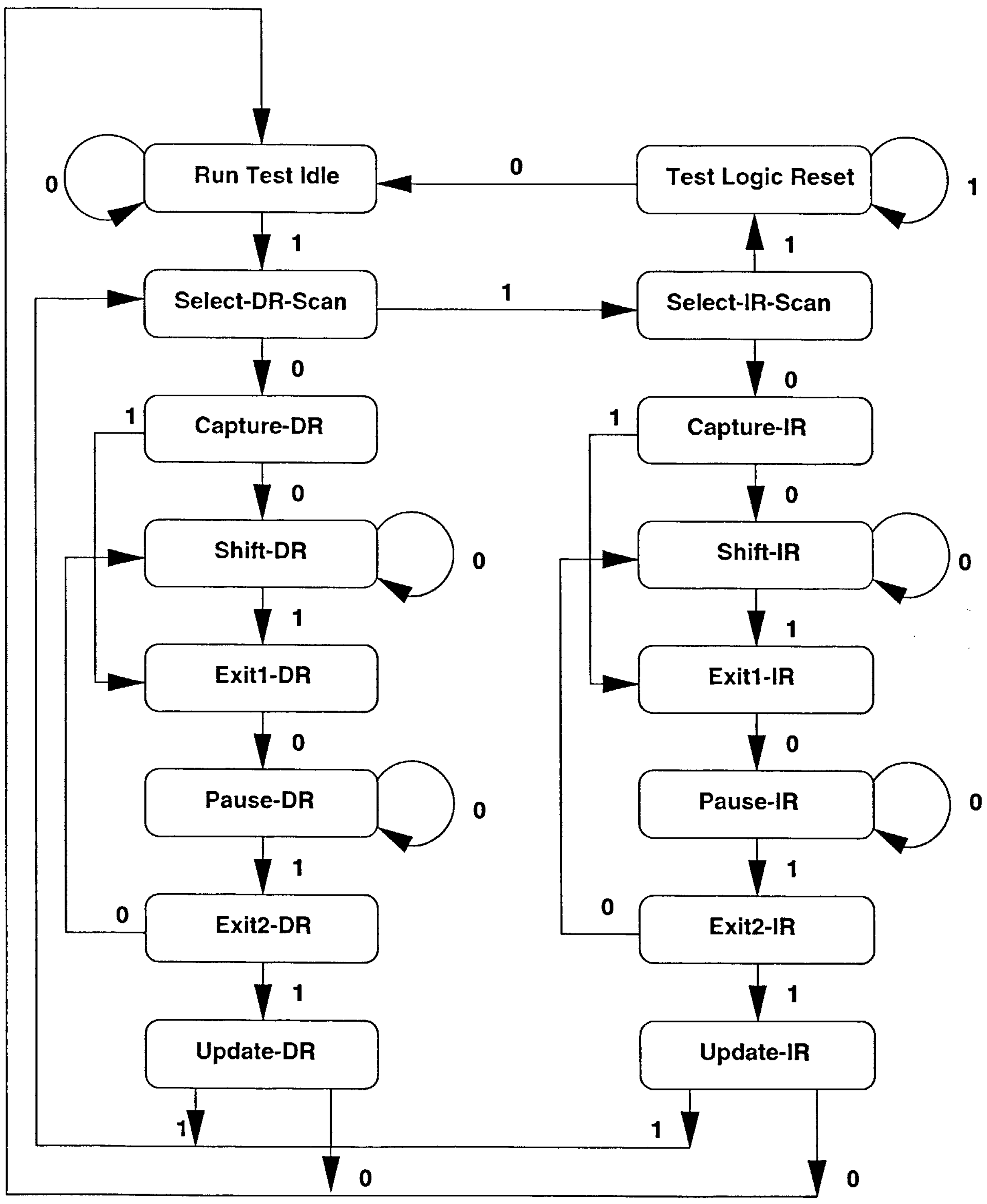


Figure 3-2: JTAG TAP State Diagram

circuitry, it is mandated by the JTAG standard that TMS and TDI only change on the falling edge of TCK.

The operation of the TAP controller is separated into two major functions for the purposes of this project: "Instruction Register Scan" (IR-Scan) and "Data Register Scan" (DR-Scan). While TMS is low, the TAP controller remains in the "Run-Test/Idle" state. When in the "Run-Test/Idle" state, the TAP controller will enter the "IR-Scan" state when TMS is held high for two rising edges of TCK followed by TMS being held low for two rising edges of TCK. An instruction can then be serially input to the chip. At the time when the last bit is clocked into the scan-register, TMS is brought high for two rising edges of TCK. This updates the instruction into the instruction register and returns the controller to the "Run-Test/Idle" state. In order to enter data into the data register, TMS is held high for one rising edge of TCK and then brought low. When the last bit is being input to the chip, TMS is brought high for two rising edges of TCK.

The TAP controller embedded in the SensorDSP chip meets the standards set by the IEEE 1149.1-1990 document [10]. In order to properly program the SensorDSP chip, a similar FSM controller was needed to generate the proper timing for the TMS signal. Figure 3-3 shows the state transition diagram of this FSM.

When the "IR_DR" signal is high, the FSM programs the instruction register. When "IR_DR" is low, the data register is programmed. The FSM outputs a "shift" signal when data is being shifted into the chip. It also outputs a "done" signal when it has finished shifting. These signals are passed to the program control FSM, which will be described in the next section.

3.1.2 Programming Interface Description

The FSM described in the previous section provides the necessary timing of the TMS signal. Since the bit-width of the serial input varies from 1 bit to 37 bits, the program control FSM is needed to keep track of which type of data is being programmed at any given time. A state transition diagram of the FSM is shown in Figure 3-4.

There are 7 different instructions that are programmed into the SensorDSP. They

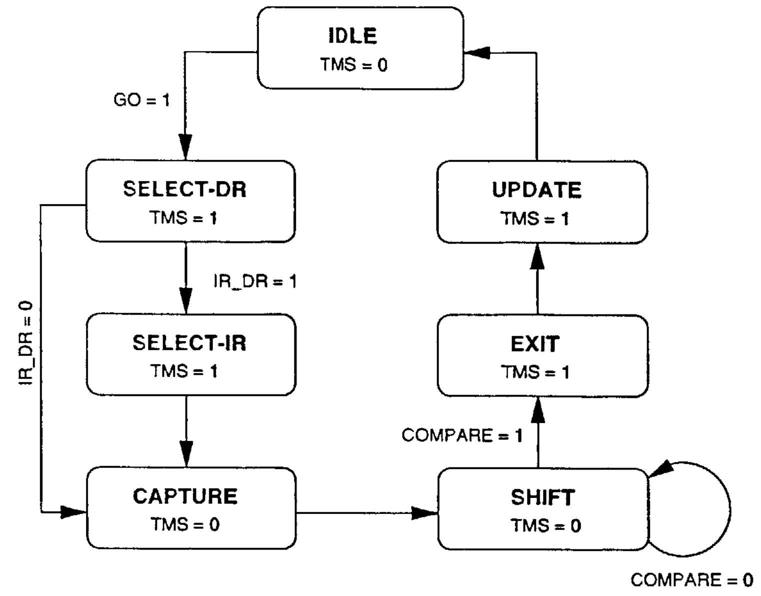


Figure 3-3: JTAG Controller State Diagram

are summarized in Table 3.1. Each instruction is a 7-bit word. Each type of TAP instruction specified in Table 3.1 has an associated bit-width. A list of these bit-widths is found in Table 3.2.

When loading distributed arithmetic table values, the first step is to supply the DA_INSTR1 instruction to the chip. Then the address that specifies a particular DA table entry is supplied to the data register of the TAP controller. Recall from Chapter 2 that there are 16 distributed arithmetic tables and each table has 8 entries. The addressing scheme uses "one-hot encoding" for the 16 most significant bits

Mnemonic	Binary Value	Description
MUCTRL_INSTR0	0000001	Sets TAP to accept micro-controller instructions
NLSL_INSTR0	0000010	Sets TAP to accept NLSL instructions
NLSL_INSTR1	0000100	Sets TAP to accept NLSL configuration settings
DA_INSTR0	0001000	Sets TAP to accept write enable bit for all units
DA_INSTR1	0010000	Sets TAP to accept DA table addresses
DA_INSTR2	0100000	Sets TAP to accept DA table values
DA_INSTR3	1000000	Sets TAP to accept DA configuration settings

Table 3.1: TAP Instruction Words

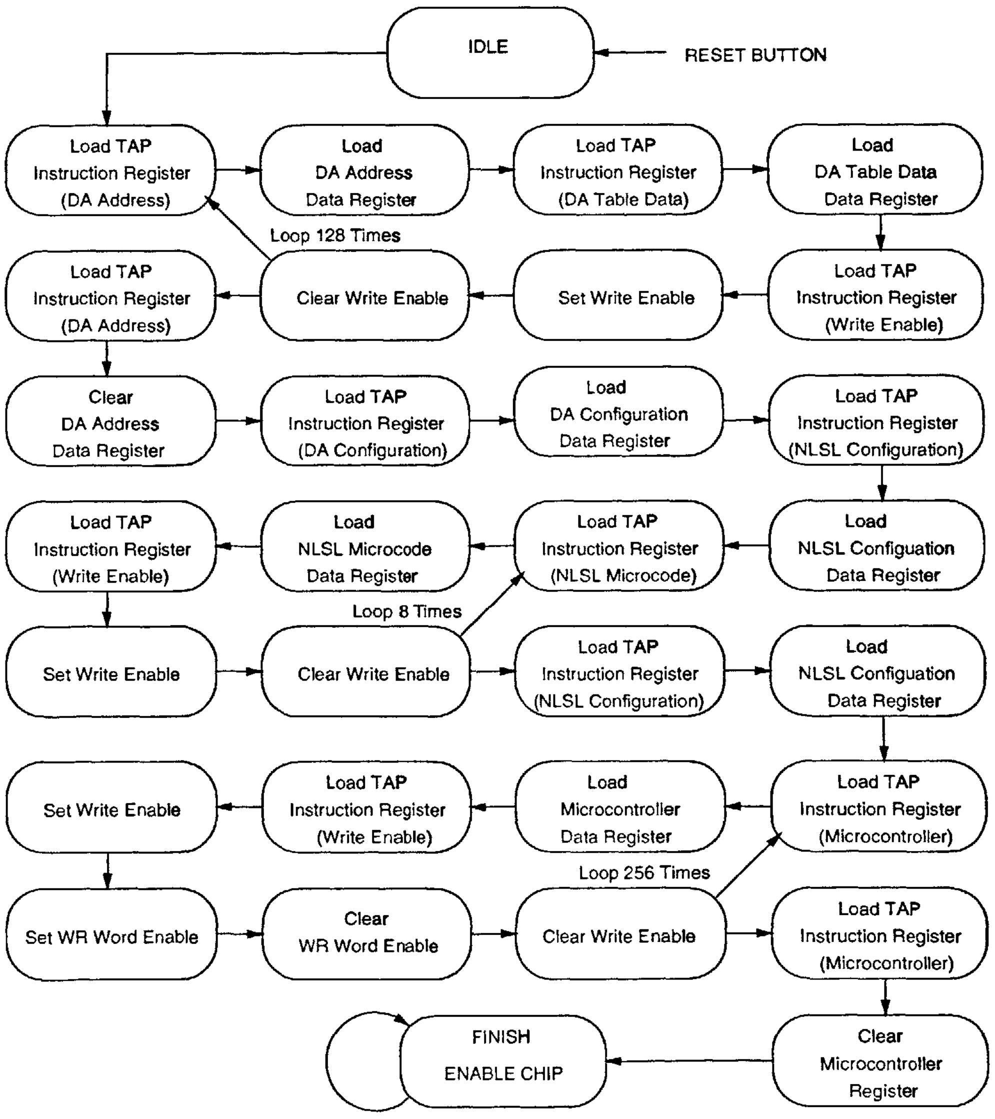

IDLE
RESET BUTTON
Load TAP Instruction Register (DA Address)
Load DA Address Data Register
Load TAP Instruction Register (DA Table Data)
Load DA Table Data Data Register
Loop 128 Times
Load TAP Instruction Register (DA Address)
Clear Write Enable
Set Write Enable
Load TAP Instruction Register (Write Enable)
Clear DA Address Data Register
Load TAP Instruction Register (DA Configuration)
Load DA Configuration Data Register
Load TAP Instruction Register (NLSL Configuration)
Load TAP Instruction Register (Write Enable)
Load NLSL Microcode Data Register
Load TAP Instruction Register (NLSL Microcode)
Load NLSL Configuation Data Register
Loop 8 Times
Set Write Enable
Clear Write Enable
Load TAP Instruction Register (NLSL Configuration)
Load NLSL Configuation Data Register
Set Write Enable
Load TAP Instruction Register (Write Enable)
Load Microcontroller Data Register
Load TAP Instruction Register (Microcontroller)
Loop 256 Times
Set WR Word Enable
Clear WR Word Enable
Clear Write Enable
Load TAP Instruction Register (Microcontroller)
FINISH ENABLE CHIP
Clear Microcontroller Register

Figure 3-4: Programming Controller FSM State Diagram

Instruction Type	Bit-width
TAP Instruction	7
Micro-controller Instruction	31
Micro-controller Write Enable	2
NLSL Filter Instruction	37
NLSL Configuration	9
NLSL Write Enable	1
DA Table Address	19
DA Table Value	11
DA Table Write Enable	1

Table 3.2: Program Instruction Bit-sizes

(MSBs) of the address to select one of the 16 tables. The 3 least significant bits (LSBs) represent the binary number of which table entry is selected. After each table value is loaded into the boundary scan register, the write enable is toggled. Toggling the write enable is accomplished in the same manner as loading addresses or data into the scan register. The process of loading table addresses and table data values is repeated 128 times since there are 8 table entries in 16 tables. After all tables have been programmed, the DA address register must be cleared. Clearing the address register allows data from the input shift register to access the DA tables instead of the scan register.

After programming the DA table values, the DA configuation register must be programmed. The DA table configuration setting is a 34 bit word. The 16 MSBs are enable bits for each DA table. The next 3 bits select the input data bit-width. The next 4 bits are the truncation setting for the linear filter accumulator. The binary values for the bit-width and truncation settings are given in Table 3.3. The remaining 11 bits are the initial condition of the linear filter. The instruction format of the DA table programming words is shown in Figure 3-5.

Once the programming for the DA unit is complete, the NLSL instructions must be programmed. This process is begun by initializing the NLSL configuration register. Then the 8 NLSL instructions can be programmed. The final step is to set the configuration register to the appropriate truncations settings for the SAC and MAC

Truncation Settings

Binary	Truncation
0001	Output = bits[11:0]
0010	Output = bits[15:4]
0100	Output = bits[19:8]
1000	Output = bits[23:12]

Bit-width Settings

Binary	Bit-width
001	1
010	2
100	4
000	8

Table 3.3: Truncation and Bit-width Configuration Settings

accumulators. The binary encoding of the truncation setting is identical to the DA truncation settings shown in Table 3.3. The NLSL instruction format can also be seen in Figure 3-5. The first 3 MSBs are the instruction address. The next 7 bits are the Square/Accumulate instruction. The next 11 bits are the Multiply/Accumulate instruction. The remaining 16 bits are the Load/Store instruction. The NLSL configuration word is 9 bits wide. The MSB is the the filter enable. The next 4 bits are the truncation setting for the SAC (ACC0). The remaining 4 bits are the truncation setting for the MAC (ACC1).

The last remaining sets of information to be programmed into the chip are the micro-controller instructions. The micro-controller programming word is 31 bits wide. The MSB is an enable bit and should always be set to 1. The next 8 MSBs are the address of the instruction to be entered. An empty (zero) bit is placed after the address. The remaining 21 bits are the micro-controller instruction. The micro-controller instruction memory load is repeated for each of the 256 instructions that can be loaded into the SensorDSP chip.

All of the programming data is stored in a 64K×8 bit programmable read-only memory (PROM). Each byte is loaded from the PROM into a shift register and shifted (LSB first) into the SensorDSP chip through the TDI pin. Since the programming information can be as much as 37 bits or as little as 1 bit, the programming control

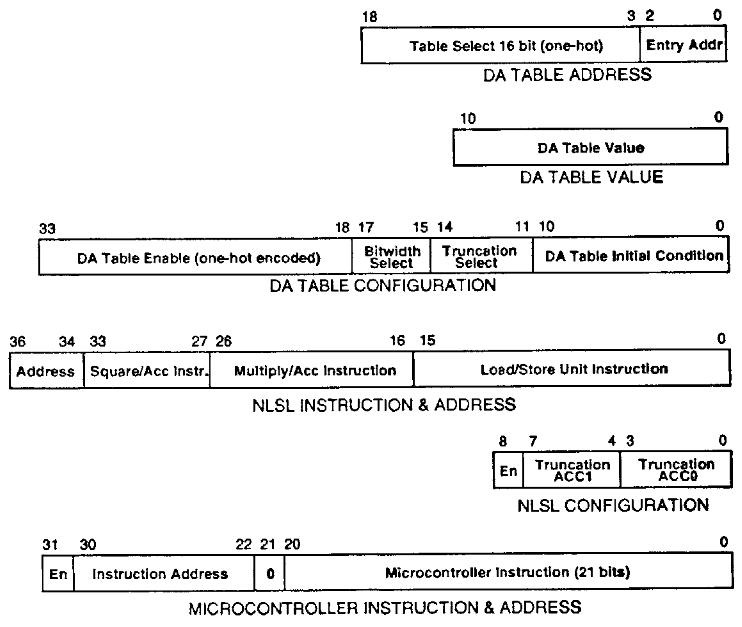


Figure 3-5: Programming Instruction Formats

FSM is responsible for controlling the TMS signal, the load signal and the shift signal to the shift register. The programming controller maintains a count of how many bits have been shifted, a count of how many times a particular sequence has been repeated, and the current address for the PROM. A block diagram of the programming controller is shown in Figure 3-6. For words longer than 8 bits, the word must be contained in multiple addresses of the PROM. For words less than 8 bits, the remaining bits of the PROM address are left unused.

The parallel to serial shift register and address counters for the PROM, as well as, the finite state machine are implemented in a single complex programmable logic device (CPLD).

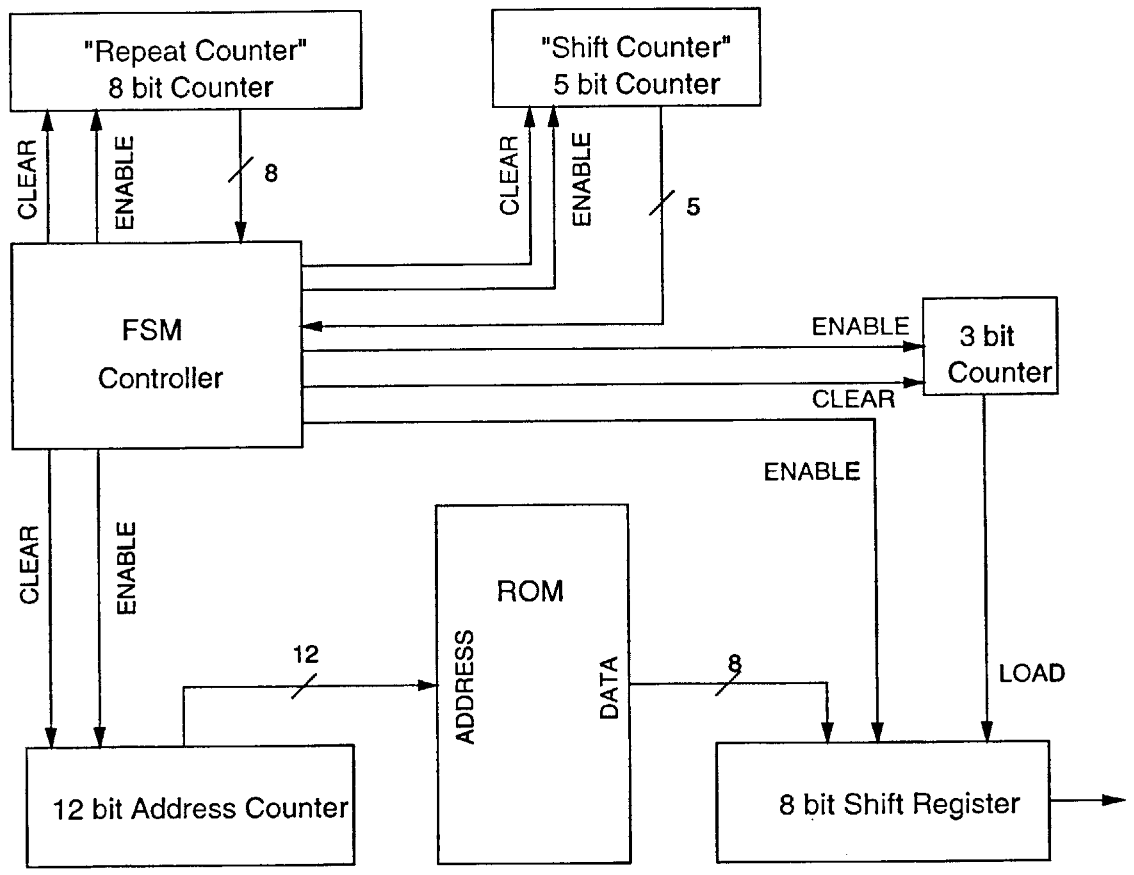


Figure 3-6: Programming Interface Block Diagram

3.2 Analog Sensor Interface

The analog acoustic sensor is a microphone and amplifier contained in a small package that can be placed against the neck or chest of an individual to sense and amplify. The sensor was provided my the Army Research Laboratory. The amplifier has 4 gain settings (20, 10, 3, 1). The output of the analog sensor is sampled with an 8 bit analog to digital converter (A/D). The A/D converter is an Analog Devices AD670 successive approximation converter [11]. The AD670 was chosen because it has the capability to operate in a bipolar or unipolar analog input. It has an adjustable analog input range and a selectable output data format. The A/D is set to operate in bipolar mode with an analog input range of +1.28V to -1.28V. The output format is set to be 2's complement.

One potential problem with the analog interface is the conversion time of the A/D. The AD670 has a maximum conversion time of 10μs. During conversion, the status bit of the A/D is a logic high. For the intended application, the sample rate is less than 1kHz, therefore the conversion time is not a limiting factor.

Since the SensorDSP chip expects a serial input data stream, the parallel output of the AD670 must be converted to serial. This is done with a shift register within a CPLD similar to the programming interface. A strobe from the I/O controller is sent to the A/D initiating a conversion. The data is then loaded into the shift register. During the next N clock cycles (where N is the desired bit-width of the input data) data is shifted (MSB first) into the SensorDSP chip. At the end of N clock cycles the A/D receives another strobe and begins conversion again. A timing diagram of the A/D interface is shown in Figure 3-7.

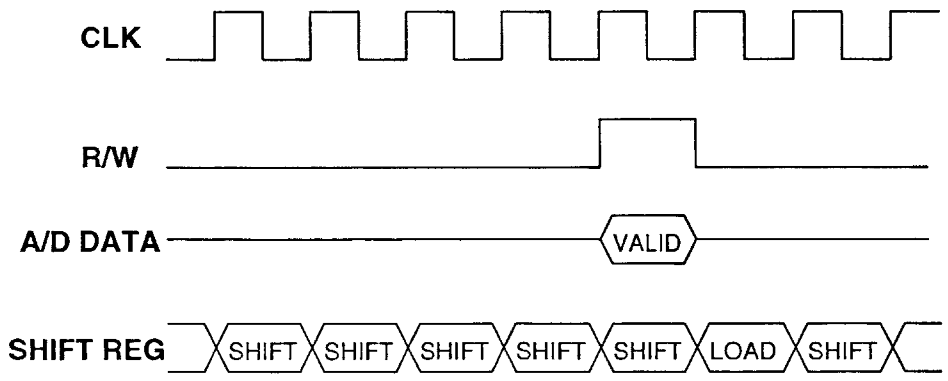


Figure 3-7: A/D Converter Timing

A significant part of properly using the system is choosing an appropriate matched filter for the desired application. As such it is useful to have a controlled input data stream to the chip, for testing and verification purposes. The SensorDSP system provides the capability to insert a PROM containing test data that can be supplied to the chip instead of data from the A/D converter. This is done by placing the

PROM's data lines on the bus with the A/D data. The I/O controller deactivates (sets the outputs to tri-state) either the A/D or the PROM based on a selection switch on the board. The I/O controller increments the PROM address at the same rate it triggers the A/D, thus simulating the presence of real data. A block diagram of the Analog Sensor Interface is shown in Figure 3-8.

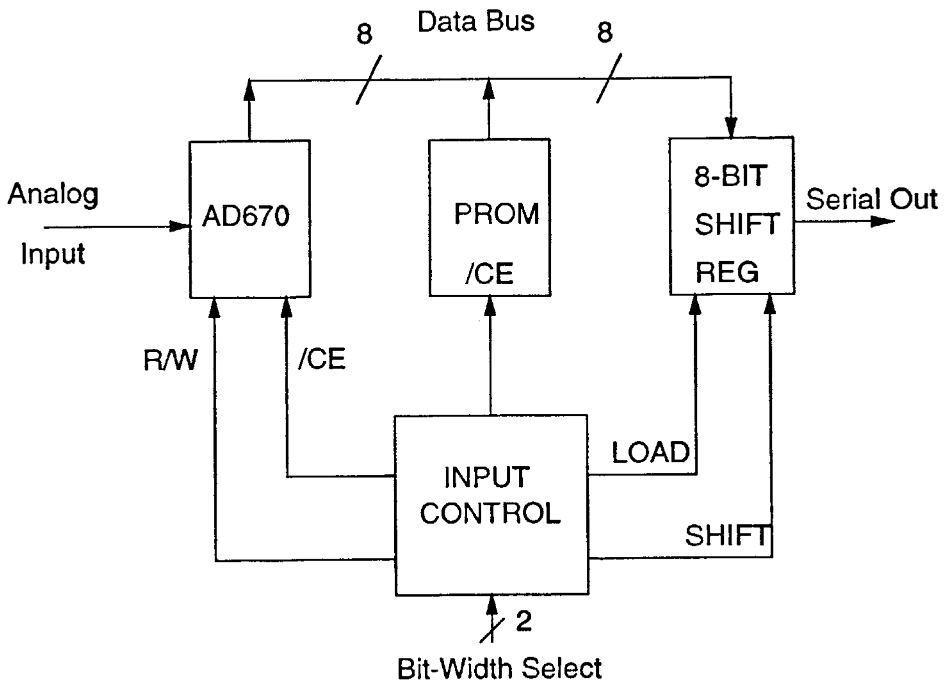


Figure 3-8: Analog Sensor Interface Block Diagram

3.3 Heartbeat Indicator

The SensorDSP chip has a limited output capability. Due to constraints on the number of pins, the chip only has a 12-bit output bus. To increase functionality and testability, this bus is 8-way multiplexed via a 3-bit select input. The data available from this bus are listed in Table 3.4.

The 3-bit selection input is supplied from the output controller. The output controller is contained within a complex programmable logic device (CPLD). The most straightforward way to indicate when the chip has made a heartbeat detection

Binary Select	Test Bus Output
000	DA Filter Output
001	NLSL Acc0 (SAC)
010	NLSL Acc1 (MAC)
011	DA Filter Buffer Output
100	NLSL Acc0 (SAC) Buffer Output
101	NLSL Acc1 (MAC) Buffer Output
110	Micro-controller Memory Data Bus
111	Micro-controller Program Counter

Table 3.4: SensorDSP Output Bus Selection

is to watch the program counter. The assembly code that implements the heartbeat detection algorithm is written such that the program jumps to a unique location of the instruction memory when a heartbeat is detected. By default, the output controller sets the SensorDSP output bus multiplexer to the program counter.

Eight external switches on the system board allow the user to enter which instruction memory address the controller will watch for. When the program counter matches the value on the input switches, an indicator light emitting diode (LED) is lit. The LED is held lit for a fixed amount of time to ensure that it is visible to the human eye. In addition to a single LED indicator, three hexadecimal numeric displays are available to display information extracted from the SensorDSP chip. For the given application, the hex displays provide an approximate "beats per minute" value. A block diagram of the heartbeat indicator circuit is shown in Figure 3-9.

The output controller CPLD can easily be reprogrammed to extract other useful information from the chip such as filtering results or micro-controller calculations.

3.4 Clock Generation

As mentioned in Chapter 1, the SensorDSP chip operates on a slow clock and a fast clock. The SensorDSP has an embedded clock generation module. This module takes in a clock reference (Φ_{ref}) that is twice the desired fast clock frequency. Based on the value of the clock configuration bits (CCONF switches), the Φ_{ref} clock is divided by a

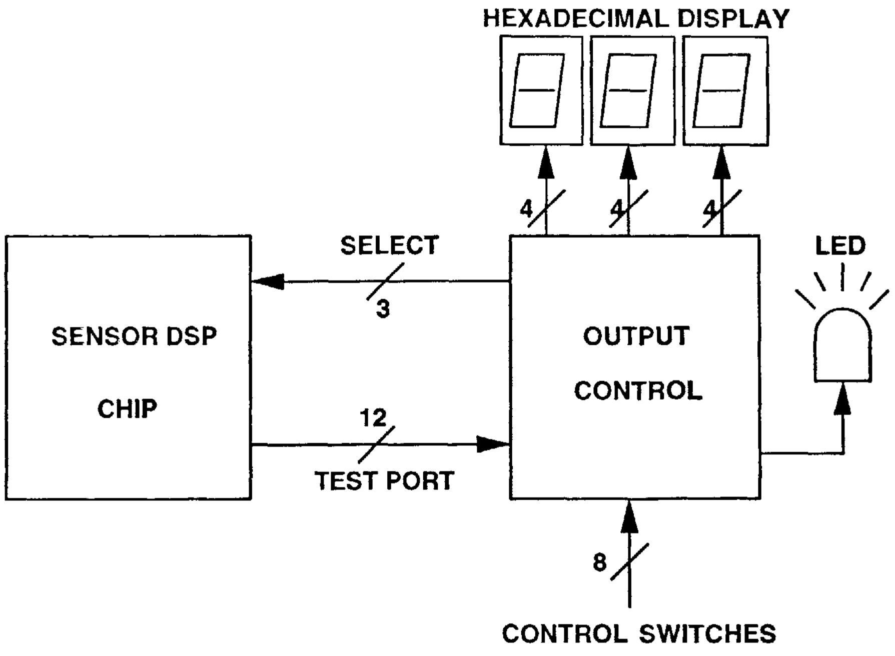


Figure 3-9: Heartbeat Indicator Block Diagram

scaling factor to produce the slow clock. Equation 3.1 calculates the clock frequency based on the frequency of the Φ_{ref} signal and the value of the CCONF switches. The value of Φ_{ref} for the system is 230.4 kHz.

$$f_{slowclk} = \frac{\Phi_{ref}}{2^{C+2}} \tag{3.1}$$

where C is the decimal value of CCONF

The clock generation module outputs the desired clock as well as the read trigger signal (RD_TRIG). The RD_TRIG signal is a clock that has a 45 degree phase shift with reference to the system clock. This signal is used for the timing of memory accesses within the chip. Figure 3-10 shows the timing of this signal.

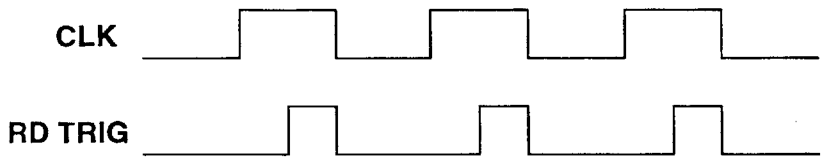


Figure 3-10: Read Trigger Timing

It was the designer's choice to isolate the clock generator module from the chip architecture. Subsequently, the "CLK" and "RD_TRIG" signals must be externally fed back into the chip. As such, the user can opt to use the internal clock generator module or the external clock generator module. Since there is combinational logic that enables the chip to switch clock frequencies there is a potential for glitches or "runt pulses" to appear on the clock signal that could cause the system to execute an incorrect instruction or skip an instruction. Because of the overall low speed operation of the system, the potential for this error is small. Additional precautions have been taken to avoid this problem by adding NOP instructions in the microcode at the locations where this glitch could occur. This is easy to do since the switching is controlled by a microcode instruction. For almost all applications, using the internal clock module is recommended.

In addition to generating the clock for the SensorDSP chip, the external clock module generates the clock for all other components in the system. The input/output controller receives the same slow clock as does the SensorDSP chip. The programming controller receives a separate 14.4 kHz clock. Figure 3-11 depicts a block diagram of the clock generator subsystem.

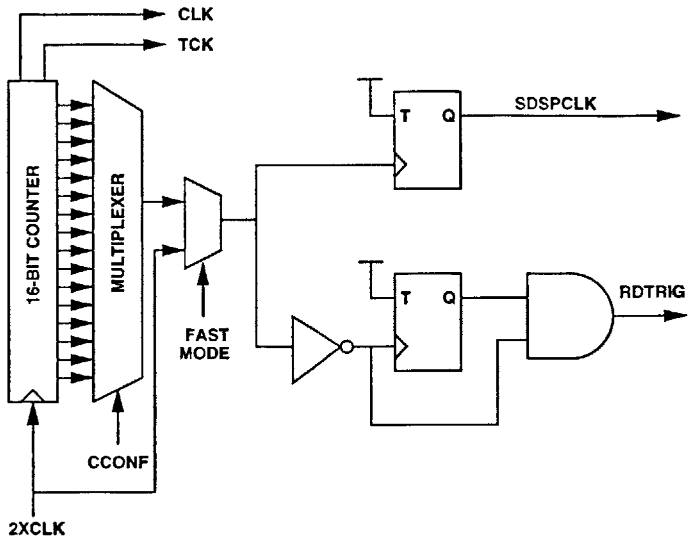


Figure 3-11: Clock Generation Circuitry

3.5 Power Measurement

The overall goal of the system is to demonstrate the functionality of the signal processing application at ultra low power levels. First, an ammeter connection is provided on the board to accurately measure the current being drawn from the power supply. Second, a bar-graph LED display in embedded in the system to allow for an approximate power measurement. A schematic of the power measurement circuitry is shown in Figure 3-12. This circuit utilizes a bank of eight comparators and a resistor voltage divider network. A resistor is placed in series between V_{DD} and the chip. The comparator array detects the voltage drop across the resistor and lights the appropriate LEDs of the bar-graph. Since the current drawn by the chip is on the order of 1μW, or less the resistance of the series resistor must be on the order of 200 kΩ in order for there to be a noticeable voltage drop. Because of this voltage drop, the external V_{DD} will need to be increased to maintain the necessary 1.5V delivered to the chip. The default setting for power measurement yields approximately 0.5μA per LED, however, a potentiometer can be adjusted to calibrate the measurement.

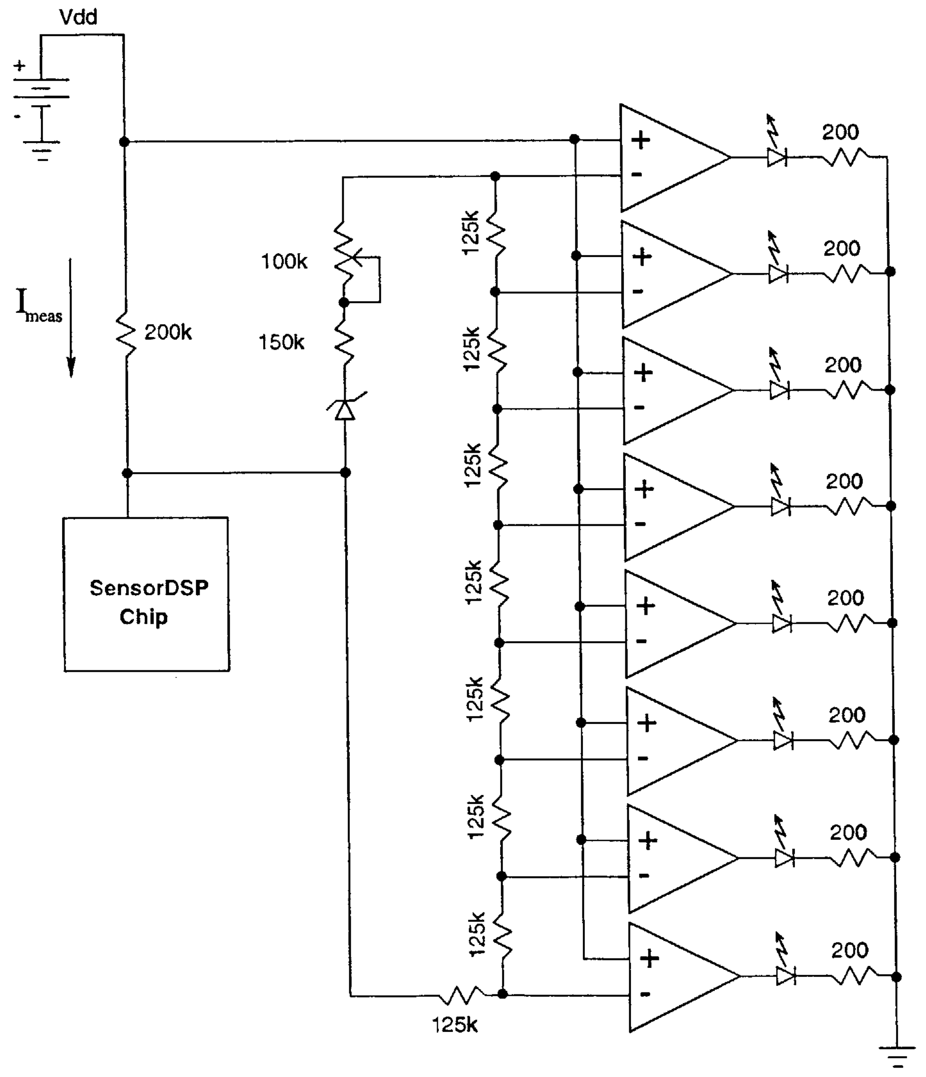


Figure 3-12: Power Measurement Schematic

3.6 Battery Vs. External Supply

The option to utilize battery power instead of an external power supply is provided in the system. The internal power supply generation is provided by a 9V battery and two voltage regulators. One is a fixed, 5V voltage regulator [14] that supplies power to all the support electronics on the board. The second is an adjustable voltage regulator [13] that supplies the V_{DD} of the SensorDSP chip. The schematic for the internal supply is shown in Figure 3-13. Jumper wires on the board are placed by the user to select external or internal mode.

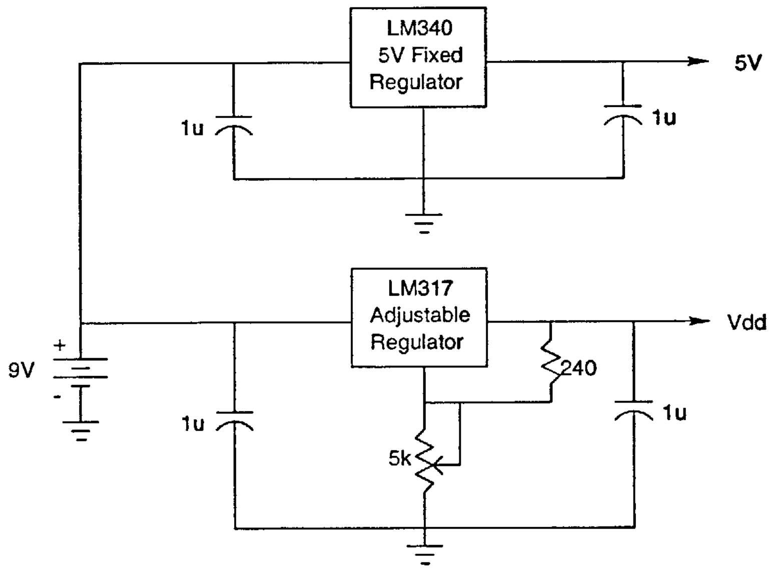


Figure 3-13: Power Regulation Schematic

3.7 Test Ports

The SensorDSP board has 7 test ports that can be used to monitor the operation of the board. Table 3.5 lists all signals available from the test ports. Figure 3-14 shows the pin diagram of a standard test port. Each test port is a 16 pin dual-in-line header pin. Eight of the pins are signal pins; the remaining eight are connected to ground.

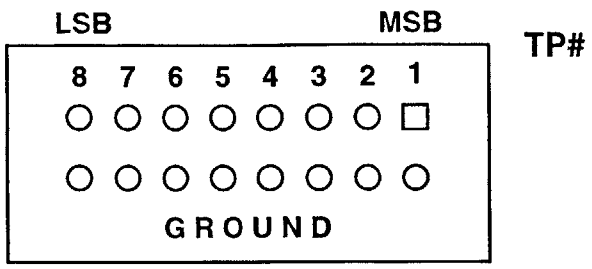


Figure 3-14: Test Point Pin Diagram

TP#	Pin #'s	Signal Name
1	1 - 8	Program PROM Data Bus
2	1 - 8	Program PROM Address Bus (7:0)
3	1	SensorDSP TDI
3	2	SensorDSP TMS
3	3	SensorDSP TRST
3	4	Program IDLE
3	5 - 8	Program PROM Address Bus (11:8)
4	1 - 8	SensorDSP Test Port (11:4)
5	1 - 4	SensorDSP Test Port (3:0)
5	5	PRE
5	6	WORD_EN
5	7	DLATCH
5	8	BUF_WEN
6	1 - 8	Test Data PROM Data Bus
7	1	FASTMODE
7	2	XOUT
7	3	SensorDSP TDO
7	4	RD_TRIG
7	5	CLKIN
7	6	LED_DETECT
7	7	SensorDSP TCK
7	8	XIN

Table 3.5: Test Port Signals

Chapter 4

System Details

"Using The SensorDSP System"

This chapter contains instructions for using the SensorDSP system. The following is a list of required components in order to properly use the system.

1. The SensorDSP board
2. The Heart Sensor Amplifier and Filter or compatible analog signal receiver
3. Dual Source Adjustable Power Supply
4. Cypress Warp2ISR Programmable Logic Kit for PC (VHDL or Verilog) Flash370i Series
5. The SensorDSP Initialization Program (for Unix)
6. A Programmable Read-Only Memory Programmer (compatible with Intel MCS-86 Hexadecimal Format)

There are many steps that must be followed in order to successfully program and use the SensorDSP system. This chapter is a step-by-step "walk-through" of the usage of the system. The topics that are covered in this chapter are:

- Compiling Program Data
- Running the system
- Using the test data input
- Extracting data from the board
- Additional Features
- Troubleshooting

Figure 4-1 is a floor-plan of the system board. All parts, switches, ports, and displays can easily be identified by referring to Figure 4-1 and Table 4.1, which lists the parts of the board with brief descriptions.

4.1 Compiling Program Data

The compilation of program data is accomplished by running the "SensorDSP Initialization Program". The program enables the user to enter linear filter coefficients, NLSL instructions, and micro-controller instructions. The program then generates an Intel MCS-86 hexadecimal formatted file (filename.ntl). The PROM can then be programmed and inserted into the system board. Before running the "SensorDSP Initialization Program" there are several steps that must be performed.

1. Choose coefficients of the matched filter. The filter coefficients should be arranged in a text file with a carriage return separating each number. There should be a maximum of 64 coefficients and each coefficient must be in the range from -255 to +255.

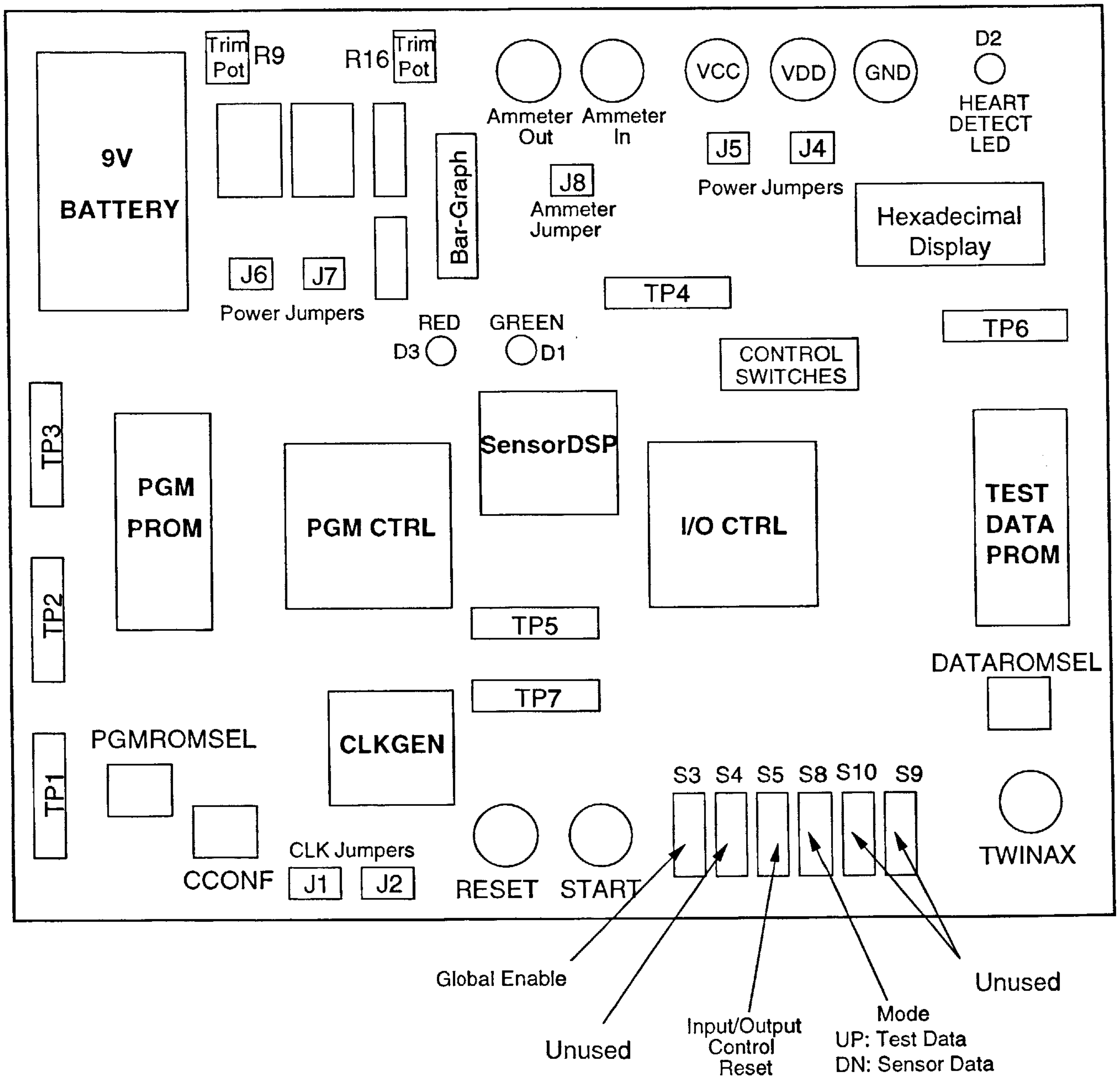


Figure 4-1: SensorDSP System Board Floor-plan

Components		
Ref#	**Nomenclature**	**Description**
U1	SensorDSP	Ultra Low-Power Classification Processor
U2	I/O CTRL	Input/Output Control Logic (CPLD)
U3	PGMROM	Program Data ROM
U4	DATAROM	Test Data ROM
U5	AD670	Analog to Digital Converter
U6	PGM Control	Programming Control Logic/FSM (CPLD)
U7	LM340	Fixed 5V Voltage Regulator
U8	LM317	Adjustable Voltage Regulator
U9	CLKGEN	Clock Generator Logic (CPLD)
U10	MAX924	Quad Comparator
U11	MAX924	Quad Comparator
U12	74LS393	Ripple Counter
Switches		
S1	CONTROL	Select Value for Program Counter Comparison
S2	CCONF	Select Clock Speed
S3	GLB EN	SensorDSP Global Enable (Debugging Only)
S4	Unused	–
S5	I/O RST	Disables I/O Control and Stops Program Counter
S6	DATAROMSEL	Select 1 of 16 Test Data Blocks in ROM
S7	PGMROMSEL	Select 1 of 16 Program Data Blocks in ROM
S8	MODE	Select I/O Mode (UP: Test Data, DN: Sensor Data)
S9	Unused	–
S10	Unused	–

Table 4.1: Component Reference Designations

2. Write a program for the NLSL unit. A separate file should be created containing the instructions for the SAC, MAC, and LSU units. Each file must have less than 8 instructions. Refer to Appendix C for the details of the NLSL instruction set. Sample files are also contained in Appendices D.2, D.3, and D.4.

3. Write a program for the micro-controller unit. Refer to Appendix B for details regarding the micro-controller instruction set. The heartbeat detection assembly code is contained in Appendix D.1. Section 4.7 of this chapter contains some additional details necessary for writing assembly code.

4. Choose input data bitwidth size. Using 8-bit data is recommended.

5. Choose the truncation settings for the DA and NLSL units. This can be done by simulating a matched filter with recorded data to predict the peak output values. In general, it is best to chose the highest truncation setting (i.e. lowest order bits are truncated) and then examine the result of the filter. If the output is zero, then a lower truncation setting should be set.

Once all the above files have been created, the "SensorDSP Initialization Program" can be executed. Start by typing: SensorDSP.

The opening screen is shown below:

```
*********************************************************
*        SENSOR DSP INITIALIZATION PROGRAM              *
*                                                       *
* 1. Assemble Microcontroller Code                      *
* 2. Assemble Non-Linear/Short-Linear Filter Code       *
* 3. Generated Distributed Arithmetic Tables            *
* 4. Generate PROM file for SensorDSP board             *
* 5. Exit Program                                       *
*                                                       *
* Please enter the appropriate number and press return  *
*********************************************************
```

Proceed by selecting option 1 and following the instructions indicated by the program. The program will display any errors that occur during the complilation of the micro-controller assembly code. If there are no errors, then proceed to option 2 and then 3. Once options 1,2, and 3 are successfully completed without errors, choose option 4 to compile the data into a .ntl file. The .ntl file will be placed in the working directory. This file can now be used to program the PROM in a standard PROM programmer.

The following section describes how to properly set up and use the board.

4.2 Running the System

This section describes how to initialize and run the system.

4.2.1 Initial Settings

Before turning the power on to the board, the following steps must be completed.

1. Set jumpers J1 and J2 to internal clock (INT).
2. Place jumper on J3.
3. Set GLB EN switch (S3) to OFF (DOWN).
4. Set I/O RST switch (S5) to ON (UP).
5. Set PGMROMSEL and DATAROMSEL to 0000 (all DOWN).
6. Set CCONF to 0110 (DN, UP, UP, DN). Refer to Section 4.6.2 for alternate clock settings.
7. Set CONTROL switch (S1) to the Program Counter value to be compared.
8. If an external power supply is being used: Set Vcc to 5V. Set Vdd to 1.5V - 3V. Place jumpers on J4 and J5. Remove jumpers from J6 and J7.
9. If the battery supply is being used: Place jumpers on J6 and J7. Remove jumpers from J4 and J5.
10. If no power measurement is needed, place a jumper on J8. If power measurement is needed, the jumper on J8 must be removed.
11. Insert Program ROM in U3 ZIF socket (left). Insert Test Data ROM in U4 ZIF socket (right).

4.2.2 Operating the System

Once the above steps have been completed, power can be turned on. NOTE: When using the battery, power can be turned off by removing jumpers J6 and J7. The following is a list of steps to operate the board:

1. Press the RESET (BLACK) Button. The Green LED (D1) should be ON and the Red LED (D3) should be OFF. The system is in the IDLE state

2. Press the START (RED) Button. The Red LED (D3) should turn ON and the Green LED (D1) should turn OFF. The system is in the PROGRAMMING state. While the Red LED is ON, the SensorDSP chip is being programmed.

3. When programming is complete, both LEDs will be ON. The system is in the RUN state.

4. Select the MODE setting: UP = test data from PROM, DN = data from analog sensor.

5. Switch I/O RST DOWN. This enables the SensorDSP chip. The DETECT LED (D2) will flash when the chip makes a positive classification.

The system can be halted by switching the I/O RST switch UP. To restart the system, the procedure outlined above must be repeated.

4.3 Analog Signal Requirements

The analog to digital converter used in this system expects a bipolar analog signal in the voltage range from -1.28V to +1.28V. The signal processing capability of the system is limited to medium to low throughput applications. It is highly recommended that the analog signal be low pass filtered prior to sampling to eliminate any high frequency noise components. For heartbeat signals, a suggested low pass filter bandwidth is 500 Hz.

Gain (dB)	Switch Position
20	0
10	1
3	2
1	4

Table 4.2: ARL Sensor Gain Settings

To allow the SensorDSP system to sample analog data, the switch labeled "MODE" must be in the "DOWN" position. More information regarding this switch is discussed in section 4.4. The serial data input can be disabled entirely by setting the "I/O_RST" switch to the "UP" position.

4.3.1 Using the ARL Sensor

The analog sensor used in the system was provided by the Army Research Laboratory. The sensor consists of a microphone, amplifier, and low pass filter ($f_{corner} = 490Hz$). Acoustic vibration is coupled to the microphone via an acoustically conductive fluid contained within the packaging of the sensor. The amplifier has a variable gain setting that can be adjusted via an 8 position rotary switch. The rotary switch can be accessed through a small hole in the under-side of the sensor package using a precision screwdriver. Table 4.2 shows available gain setting and their corresponding switch setting. The sensor connects to the SensorDSP board via the "TWINAX" connector indicated in Figure 4-1. On the sensor connector, the plug is the signal out, the pin is the +5V power supply, and the body (shield) is the ground. The sensor is capable of a -2.5V to +2.5V voltage output. The recommended gain setting for heartbeat detection is 20dB (setting 0).

4.4 Test Data Input

Digital test data can be sent to the SensorDSP chip in place of the sampled data from the A/D. This digital test data is stored in a PROM. Similar to the program data, the test data is stored in 4KB blocks in the PROM. The address to the test data PROM is incremented at the same interval that the A/D would sample. To operate the SensorDSP board in test mode, set the switch labeled "MODE" to the "UP" position. The "MODE" switch is also referred to as "S8". The test data must be in two's complement format. The data can be stored in the PROM by whatever means are the easiest for the user. The recommended method is to create a file containing the two's complement data in hexadecimal format and convert the file to an Intel MCS-86 Hexadecimal format using the "DAT2NTL" program supplied with the system.

4.5 Extracting Data

As mentioned in Chapter 2, the SensorDSP chip has a limited data output capability. The internal data-path of the chip is multiplexed to a single 12-bit test port. As such, additional control logic was designed to acquire and analyze data from the chip. This additional control is contained within the "I/O CTRL" CPLD. This CPLD can be re-programmed to perform a different function than the one described in this section. The subsequent sections describe the three methods for extracting information from the SensorDSP chip.

4.5.1 Heartbeat Detection Indicator

A single indicator light (LED) is provided to display when the chip has made a positive classification of a heartbeat. As part of the micro-controller assembly code, a unique location in the program memory is jumped to when a heartbeat is detected. The I/O CTRL logic sets the test multiplexer of the chip to select the program counter as the output. When a specific value of the program counter is reached the LED is lit. The LED remains lit for a fixed period of time, yet cleared before another potential detection is made. The switches labeled "CONTROL" in Figure 4-1 allow the user to select which program counter value the I/O CTRL logic triggers on.

4.5.2 Hexadecimal Display

The hexadecimal display is a set of 3 hexadecimal numeric displays that are connected to the I/O CTRL CPLD. For heartbeat detection the hexadecimal display is used to display the number of beats per minute that are being detected. The I/O CTRL CPLD can be re-programmed to use the hexadecimal display for other application specific purposes.

4.5.3 Using the Test Ports

There are 7 test ports on the SensorDSP board that allow access to the digital signals that pass into and out of the SensorDSP chip. A logic analyzer can be connected to these test ports to acquire data for analysis and debugging purposes. Refer to Table 3.5 and Figure 3-14 in Chapter 3 for the pin configuration of the test ports.

In order to extract information from the chip it will most likely be necessary to re-program the I/O CTRL Logic CPLD.

4.6 Additional Features

The following a a brief description of the additional features implemented in the SensorDSP system.

4.6.1 Multiple Programs

The total amount of memory utilized in the PROM to store the programming information is less than 4KB. As such, as many as 16 different programs can be stored on a single 64K$\times$8 PROM. Programs can be stored in higher address spaces in the PROM by selecting an appropriate offset address when programming the PROM. After the device is inserted in the board, the alternative programs can be selected using the switches labeled "PGMROMSEL" as indicated in Figure 4-1. By default, the starting address of the program data is 0, thus, the PGMROMSEL switches should be set to 0.

The block size of data stored in the "Test Data" PROM is also 4KB. Thus, there is space within the PROM to store 16 different test samples. The samples stored in the higher address spaces of the PROM can be accessed by appropriately setting the switches labeled "DATAROMSEL" as indicated in Figure 4-1.

4.6.2 Changing Clock Speeds

As described in Section 3.4, the SensorDSP system is capable of varying the speed at which the signal processing algorithm is executed. Equation 4.1 calculates the clock speed given a value on the CCONF switches.

$$f_{slowclk} = \frac{\Phi_{ref}}{2^{C+2}} \tag{4.1}$$

where C is the decimal value of CCONF

4.7 Troubleshooting

When testing the SensorDSP board the following steps are recommended:

1. Use an external power supply instead of the battery supply.
2. Use 8-bit input data bit-width. Lower bit-widths yield less accurate results.
3. Use test data stored in the "Test Data" PROM. With known input data, it is significantly easier to debug the system's output.
4. Use the default clock speed settings as described in Section 4.2.1.

The most common problem that will occur with the use of the SensorDSP system is having an incorrect truncation setting. If the truncation is set such that valid high order bits are truncated, the micro-controller will see overflow in the data. If the truncation is set such that the low order bits containing valid data are truncated, the micro-controller will see all zero, or negigible results. Picking the correct truncation setting, is in most cases, a trial and error procedure.

A recommended test of the system is to supply the impulse response to the chip via the test data input. By re-programming the Input/Output CPLD, the test port of the SensorDSP chip can be accessed. The outputs of the DA and NLSL units can then be monitored from the TP4 and TP5 test ports.

When writing micro-code for the feature extraction and classification processor, the following preamble should always be used:

```
CONFIG FAST_MODE CLR ;
CONFIG DATA_VAL CLR ;
NOP ;
NOP ;
NOP ;
NOP ;
NOP ;
CONFIG  ALL_ENABLE SET ;
```

In order to maintain synchronization of the input data, the ALL_ENABLE SET instruction must occur exactly 8 clock cycles after the beginning of the program execution. Refer to Appendix D.1 for more examples of SensorDSP microcode.

The the end characater for the assembly language is \$. The microcode will not compile correctly if the end character is missing.

When programming the CPLDs, it is imperative that an external power supply be used. The In-System-Reprogrammable (ISR) cable presents a significant load to the 5V power supply and subsequently, drains the battery very quickly.

Chapter 5

Conclusions

A photograph of the SensorDSP system board is shown in Figure 5-1. The SensorDSP chip is located in the center of the board. The acoustic sensor is place to the right of the board.

The system is a self-contained re-programmable system that can be used for a variety of low-power signal processing applications. The heartbeat detection algorithm was demonstrated using the system.

Future work with the SensorDSP system will reveal that a useful signal processing device can be power by ambient vibrational energy. A future iteration of the SensorDSP could incorporate the sensor, amplifier, and A/D converter into a single mixed-signal DSP chip.

All files pertaining to the operation of the SensorDSP board are stored in /obiwan/daverowe/. The files are arranged in the following subdirectories: (C, DA, muctrl, nlsl, VHDL, data, verilog). README files are provided in each directory to describe the contents.

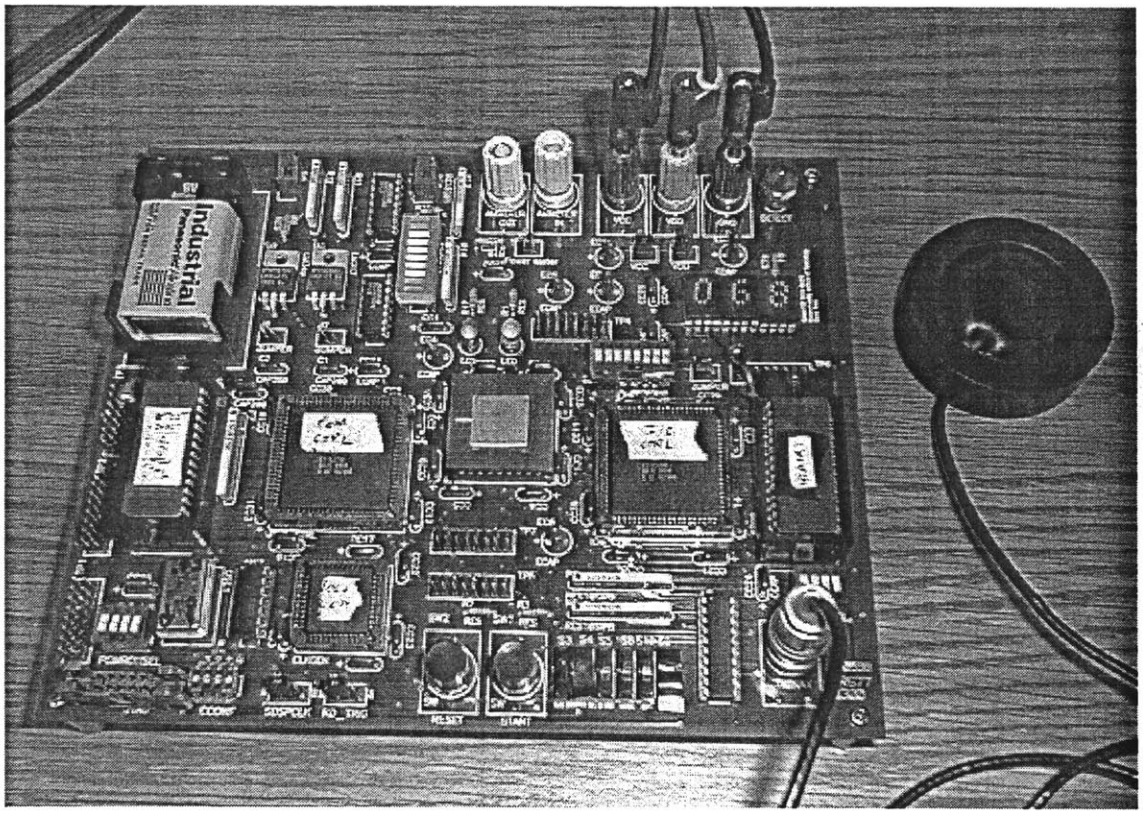


Figure 5-1: SensorDSP Demonstration System Board

Appendix A

SensorDSP Chip Pinouts

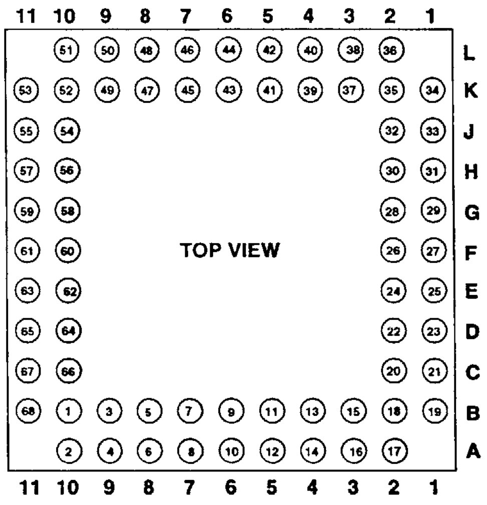


Figure A-1: SensorDSP Chip Footprint (Top View) [1]

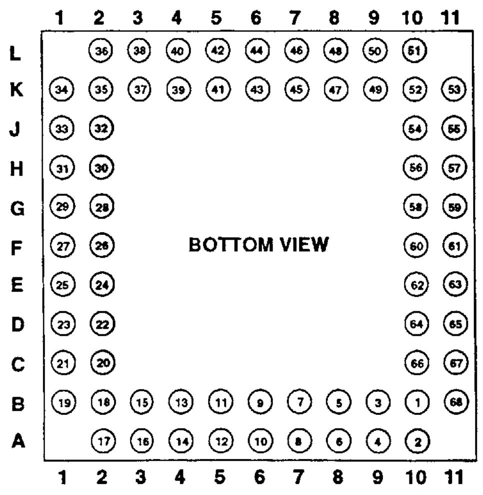


Figure A-2: SensorDSP Chip Footprint (Bottom View) [1]

Signal Pins			
Pin#	Signal Name	Type	Description
1	PRE	Output	SRAM bit-line precharge
2	WORD_EN	Output	SRAM Word Enable
3	DLATCH	Output	SRAM Data Latch
4	BUF_WEN	Output	SRAM Write Enable
6	Xin	Input	Serial Data Input
10	TDI	Input	Test Data Input (Programming)
11	TMS	Input	Test Mode Select (Programming)
12	TCK	Input	Test Clock (Programming)
13	$\overline{TRST}$	Input	Test Reset (Programming)
14	TDO	Output	Test Data Output (Programming)
15	Xout	Output	Output of Serial Shift Register
22	GLB_EN	Input	Global Enable (Forces All units ON)
28	DATA_VAL_SET	Input	Filter buffer data valid bit alway set
29	MUCTRL_EN	Input	Micro-controller Enable
34	$\overline{RST}$	Input	Reset
37	FAST_MODE	Output	Fast mode indicator
38	RD_TRIG_OUT	Output	Internal clk generator module output
39	CLK_OUT	Output	Internal clk generator module output
40	RD_TRIG	Input	SRAM Read Trigger Clock
41	CLK_IN	Input	System Clock Input
42	CLK_RST	Input	Internal clk generator module reset
43	Φ_{ref}	Input	Reference clock for clk gen module
46	CCONF0	Input	Clock speed select
47	CCONF1	Input	Clock speed select
48	CCONF2	Input	Clock speed select
49	CCONF3	Input	Clock speed select
53	YTEST_SEL0	Input	Test Port Multiplexer Select
54	YTEST_SEL1	Input	Test Port Multiplexer Select
55	YTEST_SEL2	Input	Test Port Multiplexer Select
56-67	YTEST[0:11]	Output	12-bit Test Port Bus

Power Pins	
Pin#	Name
7, 9, 16, 18, 20, 23, 25	Ground
26, 30, 32, 36, 44, 50, 52	Ground
17, 19, 27, 35, 51, 68	VHH, Chip Pad Power
8, 21, 24, 31, 33, 45	VDD, Chip Core Power

Table A.1: SensorDSP Chip Pinout

Appendix B

Sensor DSP Microcontroller Assembly Language [1]

The backend processing of the Sensor DSP chip is done on a microcontroller with a custom instruction set architecture. The microcontroller is a load/store, single pipeline stage architecture. This appendix is a user guide and documentation for the microcontroller ISA.

B.1 Instruction Set Overview

The programming model for the Sensor DSP microcontroller is shown in Figure 2-6 and repeated in Figure B-1 for convenience. It is a straightforward single instruction word RISC architecture. Both the filter buffer and the microcontroller data memory share the same address space, as will be discussed below.

Instruction Types					
⟨OPCODE⟩	0000	0000	00000000		no operand
⟨OPCODE⟩	0000		⟨ADDRESS⟩		unconditional jump
⟨OPCODE⟩	⟨DEST⟩		⟨ADDRESS⟩		direct memory instruction
⟨OPCODE⟩	⟨CC BIT⟩		⟨ADDRESS⟩		conditional jump
⟨OPCODE⟩	⟨DEST⟩		⟨CONSTANT⟩		load constant
⟨OPCODE⟩	⟨DEST⟩	⟨SRC 0⟩	00000000		2 operand w/o constant indirect memory instruction 2 operand with implicit destination
⟨OPCODE⟩	⟨DEST⟩	⟨SRC 0⟩	⟨CONSTANT⟩		2 operand w/ arithmetic constant
⟨OPCODE⟩	⟨DEST⟩	⟨SRC 0⟩	⟨SRC 1⟩	0000	3 operand w/ constant field padding

Table B.1: Microcontroller instruction types.

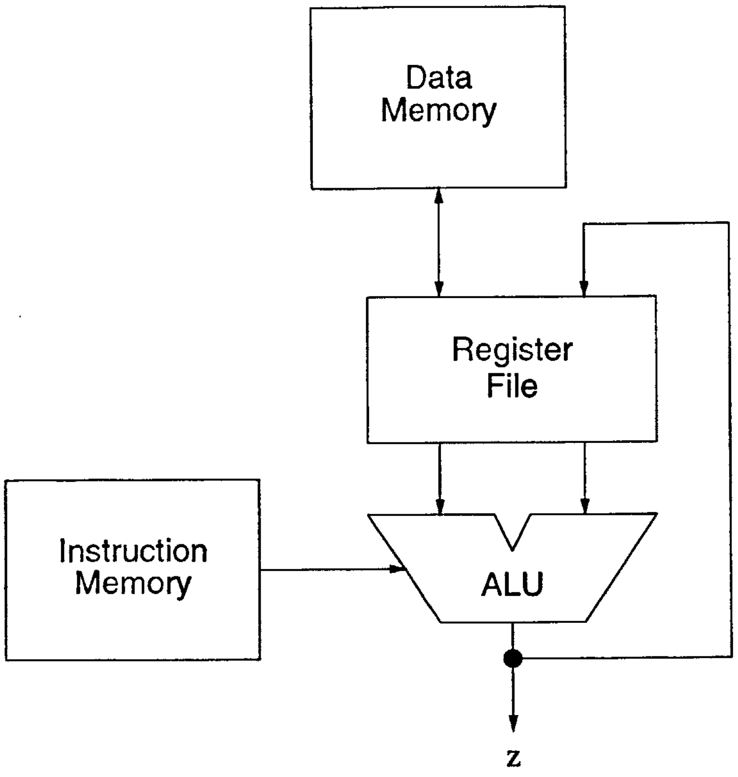


Figure B-1: Sensor DSP microcontroller architecture.

The Sensor DSP microcontroller is a twelve bit data load/store architecture with only one pipeline stage. Instructions are fetched from the instruction memory, decoded, operands are read from the register file, an operation is performed, and the result written back all in one cycle. Consequently, there are no concerns about pipeline bubbles. All instructions take exactly one clock cycle to complete so there is no possibility of out-of-order execution or issuing instructions while previous instructions are still "live". All of this makes programming the machine fairly simple.

The instruction types that the microcontroller opcode decoder recognizes are shown in Figure B.1. The actual instruction memory locations must be padded with extra 0's when a field will not be interpreted. There are 32 instructions for a five bit opcode. There are also 16 registers so register specifiers require 4 bits each. Loads and stores to the register file are done explicitly and must be performed before the operands are used in any subsequent arithmetic operations. Writes of arithmetic operation results only go back to the register file. Arithmetic operations have either two sources and one destination register or one source and one destination register. Conditional jumps require a field specifier which accesses one bit of the Condition Code (CC) register. Jump targets are twelve bits wide as well for compatibility reasons: the instruction memory on-chip is only 256 instructions deep. There are twelve address bits for data memory accesses as well. There are two kinds of constant operations: explicit loads take full twelve bit constants while some arithmetic operations have implicit four bit constants.

B.2 Registers and Other State

There are two sets of state that may be written by the microcontroller program. The first set is the set of registers that store the temporary values for the computation. The second is a group of bits that control the configuration of the front-end processing units and chip clocks, thus making the microcontroller the master functional unit on the chip.

Register Specifiers

Mnemonic	Opcode	Notes
Special Registers		
DA_ACC	0000	distributed arithmetic result
ACC0	0001	NLSL SAC ALU result
ACC1	0010	NLSL MAC ALU result
BUFPTR	0011	segment buffer pointer
CC	0100	condition code
MUACCL	0101	microcontroller accumulator, low order word
MUACCH	0110	microcontroller accumulator, high order word
Other Registers		
R7-R15	0111-1111	

Table B.2: Microcontroller registers.

Table B.2 shows the register specifiers for the various registers in the architecture. There are numerous special registers that interface with the preprocessing units (the Distributed Arithmetic Unit and the Nonlinear/Short Linear Filter Unit). The **DA_ACC** register is implicitly loaded with the results of the DA operation from the filter buffer. Similarly, the **ACC0** and **ACC1** registers store the results of the two filter ALUs from the NLSL Unit, also via the filter buffer. **BUFPTR** points to the location of the current sample in the filter buffer. The condition code register **CC** implicitly stores several bits from the datapath that are useful for conditional operations. Its use will be described in detail below. Lastly, **MUACCH** and **MUACCL** are the high low twelve bit words of the twenty-four bit wide microcontroller accumulator. The results of certain arithmetic operations have twice as many bits as the datapath operands so it is necessary to have this register store the full result. The user is then free to implement any roundings or truncations he or she sees fit. Registers **R7** through **R15** are general purpose registers and are never implicitly written.

Table B.3 shows the various specifiers for the bits of the condition code register. Conditional instructions branch based on the value of the one bit specified using these mnemonics. These specifiers and their use will be described in more detail below in the discussion of control flow instructions in Section B.3.7. Note that in addition to

CC Register Field Specifiers

CC Field	Mnemonic	Opcode	Notes
CC[0]	SIGN12	0000	ALU 12 bit result sign bit
CC[1]	OFLW12	0001	ALU 12 bit result overflow bit
CC[2]	WNOR12	0010	wired NOR of ALU 12 bit result bits
CC[3]	SIGN24	0011	24 bit accumulator sign bit
CC[4]	OFLW24	0100	24 bit accumulator overflow
CC[5]	LOW12	0101	ALU 12 bit result low bit
CC[6]	WOR12	0110	wired OR of ALU 12 bit result bits
CC[7]	PCSEL	0111	jump select bit result
CC[8]	WHIGH	1000	wired high
CC[9]	WLOW	1001	wired low
CC[10]	DATA	1010	new buffer data valid bit
CC[11]	FAST	1011	fast mode enable bit

Table B.3: Condition code register field specifiers for conditional jumps.

the mnemonics, the strings `CC[i]` may also be used as register field specifiers. `CC[0]` is equivalent to `SIGN12` to the instruction decoder.

Configuration Bit Specifiers

Mnemonic	Opcode	Notes
DA_ENABLE	0000	distributed arithmetic unit enable
NLSL_ENABLE	0001	nonlinear/short linear filter unit enable
DATA_VAL	0010	filter buffer valid data bit
BUF_ENABLE	0011	filter buffer enable
ALL_ENABLE	0100	enable all units
FAST_MODE	0101	fast mode select

Table B.4: Configuration state bit specifiers for configuration instructions.

The Sensor DSP microcontroller is also responsible for software enables of the preprocessing filter units and selection of the appropriate clock frequency for real-time operation. This is accomplished using the configuration bit specifiers shown in Table B.4. See Section B.3.9 for instructions and examples of their use.

B.3 Instruction Descriptions

In this section, we describe in detail the commands that the assembler understands. In addition to explicit instructions implemented in the hardware, there are several macros which the assembler expands into one line assembly instructions. All macros are also implemented in one cycle. Some additional commands for naming branch targets and aliasing constants and memory locations are also described.

Note that the assembler program `muctrlasm` is case-insensitive, even though the majority of the example code is uppercase.

B.3.1 RTL Description

Instruction Syntax and RTL Description

Mnemonic	Oprnd	Oprnd	Oprnd	Notes
NOP				
MACC	[R1]	[R2]		⟨MUACC⟩ ← ⟨MUACC⟩ + ⟨R1⟩*⟨R2⟩
MDEC	[R1]	[R2]		⟨MUACC⟩ ← ⟨MUACC⟩ - ⟨R1⟩*⟨R2⟩
MULT	[R1]	[R2]		⟨MUACC⟩ ← ⟨R1⟩ * ⟨R2⟩
ADD	[R1]	[R2]	[R3]	⟨R1⟩ ← ⟨R2⟩ + ⟨R3⟩
SUB	[R1]	[R2]	[R3]	⟨R1⟩ ← ⟨R2⟩ - ⟨R3⟩
ADDC	[R1]	[R2]	[const]	⟨R1⟩ ← ⟨R2⟩ + [const]
SUBFC	[R1]	[R2]	[const]	⟨R1⟩ ← [const] - ⟨R2⟩
NOT	[R1]	[R2]		⟨R1⟩ ← ⟨R2⟩ (bitwise NOT)
NAND	[R1]	[R2]	[R3]	⟨R1⟩ ← (⟨R2⟩ * ⟨R3⟩) (bitwise NAND)
AND	[R1]	[R2]	[R3]	⟨R1⟩ ← (⟨R2⟩ * ⟨R3⟩) (bitwise AND)
NOR	[R1]	[R2]	[R3]	⟨R1⟩ ← (⟨R2⟩ + ⟨R3⟩) (bitwise NOR)
OR	[R1]	[R2]	[R3]	⟨R1⟩ ← (⟨R2⟩ + ⟨R3⟩) (bitwise OR)
XOR	[R1]	[R2]	[R3]	⟨R1⟩ ← (⟨R2⟩ (x) ⟨R3⟩) (bitwise XOR)
EQ	[R1]	[R2]	[R3]	⟨R1⟩ ← (⟨R2⟩ (x) ⟨R3⟩) (bitwise EQ)
LRSHFT	[R1]	[R2]		⟨R1⟩ ← { 0, ⟨R2⟩[11:1] }
LLSHFT	[R1]	[R2]		⟨R1⟩ ← { ⟨R2⟩[10:0], 0 }
ARSHFT	[R1]	[R2]		⟨R1⟩ ← { ⟨R2⟩[11], ⟨R2⟩[11:1] }
LCIRC	[R1]	[R2]		⟨R1⟩ ← { ⟨R2⟩[0], ⟨R2⟩[11:1] }
RCIRC	[R1]	[R2]		⟨R1⟩ ← { ⟨R2⟩[10:0], ⟨R2⟩[11] }
JUMP	[addr]			⟨MUCTRLPC⟩ ← [addr]
CTJUMP	[CC][i]	[addr]		if ⟨CC[i]⟩ then ⟨MUCTRLPC⟩ ← [addr]
				else ⟨MUCTRLPC⟩ ← ⟨MUCTRLPC⟩ + 1
CFJUMP	[CC][i]	[addr]		if ⟨CC[i]⟩ then ⟨MUCTRLPC⟩ ← [addr]
				else ⟨MUCTRLPC⟩ ← ⟨MUCTRLPC⟩ + 1
LOAD	[R1]	[addr]		⟨R1⟩ ← ⟨DMEM[addr]⟩
STOR	[R1]	[addr]		⟨DMEM[addr]⟩ ← ⟨R1⟩
LDI	[R1]	[R2]		⟨R1⟩ ← ⟨DMEM[⟨R2⟩]⟩
STI	[R1]	[R2]		⟨DMEM[⟨R2⟩]⟩ ← ⟨R1⟩
LDC	[R1]	[const]		⟨R1⟩ ← [const]
CONFIG	[CNF][i]	[state]		⟨CNF[i]⟩ ← [state]

Table B.5: Microcontroller instruction syntax and RTL level description.

The instruction syntax and RTL description for the explicitly implemented instructions are shown in Table B.5. Macros and other abstracted instructions get their operands from the same fields as their underlying implementations, which are described in various tables throughout this manual. Refer to Table B.5 to determine which fields operands are coming from and results are going to within a program.

B.3.2 Miscellaneous Instructions

Miscellaneous Instructions

Mnemonic	Opcode	Notes
NOP	00000	separate instruction to disable accesses and save power

Table B.6: Arithmetic instructions and opcodes.

Table B.6 summarizes the miscellaneous and utility instructions for the architecture. Each of these will be described in turn below.

NOP

This is the null operation instruction. It is implemented by simply turning off the write enables to the microcontroller register file.

Example: `NOP`.

B.3.3 Arithmetic Instructions

Arithmetic Instructions

Mnemonic	Opcode	Notes
MACC	00001	multiply-accumulate
MDEC	00010	multiply-decumulate
MULT	00011	multiply
ADD	00100	add
SUB	00101	subtract
ADDC	00110	add constant
SUBFC	00111	subtract from constant

Table B.7: Arithmetic instructions and opcodes.

Table B.7 summarizes the arithmetic instructions for the architecture. Each of these will be described in turn below.

MACC

MACC is the multiply-accumulate instruction. All multiplication related instructions implicitly send their results to the multiplier accumulator register MUACC and so require only two source operands to be specified instead of two sources and a destination. **MACC** takes the product of its operands and adds them to the contents of the MUACC register.

Example: `MACC ACC0 ACC1`

MDEC

The multiply-decumulate instruction is **MDEC**. It works similarly to the multiply and accumulate instruction, except it *subtracts* the product of its operands from the MUACC register instead of adding them.

Example: `MDEC CC MUACCH`

MULT

MULT multiplies its operand together and overwrites the value in the MUACC register with the resultant product. It also has an implicit destination like the previous instructions.

Example: `MULT R7 R8`

ADD

ADD performs arithmetic addition. It computes the sum of its operands and stores the result in the destination register.

Example: `ADD R9 R10 R11`

SUB

Arithmetic difference is computed by **SUB**. It writes the value of subtracting its second operand from its first into the destination register. Integers are represented in two's complement notation so negative values are allowed.

Example: `SUB R12 R13 R14`

ADDC

ADDC is similar to **ADD** above, except that one of its arguments is an eight bit two's complement integer constant. This constant is sign extended to twelve bits when the actual addition is performed.

Example: `ADDC R15 R15 17`

SUBFC

SUBFC (**SUB**tract From Constant) computes a difference like **SUB** above except that one of its arguments is an eight bit constant. Also, the order of the operands in the computation is reversed (see Table B.5) in that the value from the register is subtracted *from* the sign extended version of the eight bit constant.

Example: `SUBFC R7 R9 -1`

In the example above, the contents of register R9 would be subtracted from -1 and stored into register R7.

B.3.4 Arithmetic Macros

Arithmetic Macros

NEG	SUBFC	[R1]	[R2]	0	arithmetic negation
COPY	ADDC	[R1]	[R2]	0	register to register copy
INC	ADDC	[R1]	[R1]	1	increment
DEC	ADDC	[R1]	[R1]	-1	decrement
SUBC	ADDC	[R1]	[R2]	-[const]	subtract constant, negation by assembler
ZERO	SUB	[R1]	[R1]	[R1]	zero a register

Table B.8: Arithmetic macros and implementations.

Table B.8 shows the definitions of various arithmetic macros that may be used exactly like real instructions in an assembly program. The assembler handles the expansion and any other preprocessing before the final translation into binary.

NEG

NEG computes the additive inverse in two's complement notation of the contents of register R2 and stores the results in register R1.

Example: `NEG R7 R8`

COPY

COPY copies the contents of register R2 and stores them into register R1.

Example: `COPY R9 R10`

INC

Incrementing macro INC adds 1 to the contents of register R1 and stores the result in the same place, overwriting its previous value.

Example: `INC R11`

DEC

Decrementing instruction macro DEC works just like INC above, except that the contents of R1 are replaced with the sum of R1 and -1.

Example: `DEC R12`

SUBC

SUBC subtracts an explicit constant from the contents of R2 and stores the value into destination register R1. Thus, the order of its operation is the same as the **SUB** instruction above and the reverse of the **SUBFC** instruction.

Example: `SUBC R9 R10 16`

ZERO

ZERO zeroes out a register by subtracting its contents from itself.

Example: `ZERO R6`

B.3.5 Relational Macros

Relational Macros

GT	SUB	CC	[R2]	[R1]	greater than
LT	SUB	CC	[R1]	[R2]	less than
EQUAL	SUB	CC	[R1]	[R2]	arithmetic equivalence
LTC	ADDC	CC	[R1]	-[const]	less than constant, constant negated by assembler
GTC	SUBFC	CC	[R1]	[const]	greater than constant
EQC	SUBFC	CC	[R1]	[const]	equal to constant

Table B.9: Relational macros and implementations.

Table B.9 shows the definitions of various relational operator macros that may be also used exactly like real instructions in an assembly program. These instructions are implemented as macros to conserve opcodes in the instruction set. All relational instructions *implicitly* write the condition code register using the same data from implicit writes in typical arithmetic and logical operations even though the macro expansion explicitly uses the CC register specifier as the destination. This implicit write is implemented within the instruction decoding ROM of the microcontroller.

GT

GT (Greater Than) determines whether the value in register R1 is greater than in register R2 and stores the result bit implicitly in the CC register field SIGN12.

Example: `GT R7 R8`

In the example above, the **GT** instruction sets the SIGN12 bit in the CC register true if the value in R7 is greater than the value in R8. This is done by subtracting the value in R7 from the value in R8. If this difference is negative, the sign bit of the result will be 1 and the relation will be true.

LT

LT (Less Than) sets the SIGN12 field of the CC register true if the value in register R1 is less than the value in register R2.

Example: `LT R9 R10`

EQUAL

EQUAL sets the WNOR12 field of the CC register true if the value in register R1 is equal to the value in register R2.

Example: `EQUAL R11 R8`

LTC

LTC (Less Than Constant) compares the value in register R1 to an explicit constant. It sets the result bit in the SIGN12 field of the CC register.

Example: `LTC R12 9`

GTC

GTC (Greater Than Constant) is the complement of **LTC** above: it compares the values of register R1 and an explicit constant and stores the result bit into the SIGN12 field of the CC register.

Example: `GTC R9 25`

EQC

Comparing equality with a constant is the function of the **EQC** macro. If register R1's value is equal to the explicit constant, then the WNOR12 field of the CC register is set to true.

Example: `EQC R6 -33`

B.3.6 Logical and Shift Instructions

In addition to the arithmetic instructions and macros supported above, the micro-controller architecture also implements a set of logical and shift operations. These are summarized in Table B.10 and described in detail in the following section.

NOT

The bitwise negation instruction **NOT** inverts the bits of the contents of its argument register and stores the result into the destination register.

Example: `NOT R7 R8`

Logic and Shift Instructions

Mnemonic	Opcode	Notes
NOT	01000	logical negation
NAND	01001	logical nand
AND	01010	logical and
NOR	01011	logical nor
OR	01100	logical or
XOR	01101	logical xor
EQ	01110	logical equivalence
LRSHFT	01111	logical right shift
LLSHFT	10000	logical left shift/arithmetic left shift
ARSHFT	10001	arithmetic right shift
LCIRC	10010	circular left shift
RCIRC	10011	circular right shift

Table B.10: Logical and shift instructions and opcodes.

NAND

NAND performs a bitwise NAND operation between the values stored in its operand registers and writes the result into the destination.

Example: `NAND R6 R10 R11`

AND

AND computes the bitwise AND of the values in its argument registers. The results are written to the destination. **AND** is the bitwise inverse of **NAND** above.

Example: `AND R6 R10 R11`

NOR

NOR computes the bitwise NOR of the arguments and writes the result to the destination.

Example: `NOR da_acc r8 r9`

OR

The bitwise OR of the operands is performed by the **OR** instruction, which is the bitwise inverse of **NOR** above. Results are written to the destination register.

Example: `OR da_acc acc0 muaccl`

XOR

XOR computes the bitwise XOR of the two operands and writes the result to the third register.

Example: `XOR bufptr cc acc1`

EQ

EQ is the bitwise inverse of **XOR** above and computes the bitwise equivalence of the two operands and writes the result to the destination.

Example: `EQ r6 r7 r10`

LRSHFT

LRSHFT (Logical **R**ight **SHIFT**) takes a single operand and bit shifts it to the right, shifting in a 0 into the most significant bit. The result is written to the destination register.

Example: `LRSHFT r11 r12`

LLSHFT

LRSHFT (Logical **L**eft **SHIFT**) takes a single operand and bit shifts it to the left, shifting in a 0 into the least significant bit. The result is written to the destination register.

Example: `LLSHFT r13 r14`

ARSHFT

ARSHFT (**A**rithmetic **R**ight **SHIFT**) takes a single operand and bit shifts it to the right, shifting in a 0 into the most significant bit if the previous MSB is 0, and a 1 if the MSB was previously 1. This preserves the sign of the binary value if it is interpreted as a two's complement number. The result is written to the destination register. Note that there is no complementary instruction since the arithmetic left shift is equivalent to the logical left shift.

Example: `ARSHFT r15 r15`

LCIRC

LCIRC (**L**eft **CIRC**ular shift) takes the value stored in its operand register, shifts it to the left one bit, and copies the previous most significant bit into the new least significant bit position. The result is written to the destination register.

Example: `LCIRC r7 da_acc`

RCIRC

RCIRC (**R**ight **CIRC**ular shift) is the complement of the **LCIRC** instruction above. It takes the value stored in its operand register, shifts it to the right one bit, and copies the previous least significant bit into the new most significant bit position. The result is written to the destination register.

Example: `RCIRC r10 muacch`

Control Flow Instructions

Mnemonic	Opcode	Notes
JUMP	10100	jump
CTJUMP	10101	conditional jump true
CFJUMP	10110	conditional jump false

Table B.11: Control flow instructions and opcodes.

B.3.7 Control Flow Instructions

Table B.11 lists the control flow instructions available to the microcontroller programmer. These consist of one unconditional and two conditional jumps. Because of the limited instruction set size and instruction memory on-chip, there are no provisions for subroutines.

JUMP

JUMP is the unconditional jump instruction. It loads the program counter (PC) register with its argument, an explicit address or an address label.

First Example: `JUMP 0x1AF`

In the first example, the target for the jump is the hexadecimal address `0x1AF`.

Second Example: `JUMP LOOP1:   ; jump forward`

The second example shows the use of an address label. The assembler makes two passes over the source code and resolves labels like `LOOP1:` into absolute addresses like in the first example. The program example in Section B.4 shows an example of the use of address labels in a loop structure.

CTJUMP

CTJUMP (Conditional True **JUMP**) replaces the program counter with its address argument if the boolean argument is true. The boolean argument is the bit of the condition code register specified with one of the condition code field specifiers listed in Table B.3.

Example: `CTJUMP CC[0] LOOP0:`

The example will write the program counter with the address referred to by the address label `LOOP0:` if the value of the `CC[0]` field of the condition code register is true.

CFJUMP

CFJUMP (Conditional False **JUMP**) is the complement of **CTJUMP** above. It replaces the program counter with its address argument if the boolean argument is *false*. The boolean argument is the bit of the condition code register specified with one of the condition code field specifiers listed in Table B.3.

Example: `CFJUMP sign24 0x232`

The example will write the program counter with the address referred to by the address label 0x232 if the value of the sign24 field of the condition code register is 0 (false).

B.3.8 Memory Instructions

Memory Instructions

Mnemonic	Opcode	Notes
LOAD	11000	load register from memory
STOR	11001	store register to memory
LDI	11010	indirect load register from memory
STI	11011	indirect store register to memory
LDC	11100	load 12 bit constant to register

Table B.12: Memory instructions and opcodes.

Table B.12 shows the memory interface instructions of the microcontroller. The Sensor DSP microcontroller is a load/store architecture, so memory accesses use only the register file as a destination. It can use either the register file (for storing variables) or the instruction stream (for storing constants) as a source. Constants can only be written to the register file. The instruction set supports both direct and indirect operations. Only the data memory is visible to the processor; the instruction memory can only be loaded through the JTAG programming interface. Consequently, self-modifying code is not allowed.

LOAD

LOAD is the direct load from memory. It retrieves the data stored in the data memory (or filter buffer, since the processor sees a flat address space) and stores it in the destination register.

Example: `LOAD R7 0x1FF`

The example loads the value at memory location 0x1FF and stores it in the register R7.

STOR

The direct memory write instruction is **STOR**. This operation takes the data stored in the source register and writes it to the memory location specified as a target.

Example: `STOR R11 0x9AC`

The data stored in register R11 will be written to data memory location 0x9AC in the above example.

LDI

LDI is the indirect load instruction. It takes two register specifiers as arguments and reads the contents of the memory location addressed by the operand register value and writes this value into the destination register.

Example: `LDI R12 R13`

In the example, the data memory location pointed to by the value of register `R13` is written to the destination register `R12`.

STI

STI is the indirect store instruction; it is the complement to **LDI** above. It takes the contents of the operand register and writes the value to the memory location addressed by the contents of the destination register.

Example: `STI R6 R15`

The above example reads the contents of register `R6` and writes the value to the data memory location pointed to by the value stored in register `R15`. Note that this convention is the reverse of the argument convention for **LDI**

LDC

LDC (LOAD Constant) is the register constant load instruction. It takes the explicit 12 bit integer constant specified as an argument and writes it to the register location also specified in its arguments.

Example: `LDC R9 -3`

The example writes the constant `-3` (in two's complement representation) into register `R9`.

B.3.9 Filter Interface Instructions

Filter Interface Instructions

Mnemonic	Opcode	Notes
CONFIG	11110	set configuration state bits

Table B.13: Filter interface instructions and opcodes.

The front-end filtering units of the Sensor DSP chip are under software control of the microcontroller using the configuration instructions and state bit specifiers shown in Tables B.13 and B.14 respectively.

CONFIG

CONFIG is the sole configuration instruction available to the microcontroller. In conjunction with the state specifiers shown in Table B.14 and the configuration bit

Configuration State Specifiers

Mnemonic	Code	Notes
CLR	00	set bit to 0
SET	01	set bit to 1
FLIP	10	invert bit
HOLD	11	maintain state (basically a NOP)

Table B.14: Configuration state specifiers and codes.

specifiers shown in Table B.4 it configures the operation of the entire processor chip. The state of each configuration bit can be modified using the state specifiers shown in Table B.14. An example will clarify the operation:

Example: `CONFIG DA_ENABLE CLR`

The example shows that the configuration bit `DA_ENABLE` will be cleared (set to 0) by the instruction. The bit could also be set to 1, inverted, or held in the same state (a null operation).

B.3.10 Unused Opcodes

Unused Opcodes

Opcode
10111
11101
11111

Table B.15: Unused opcodes.

Table B.15 lists the opcodes that are unused in the ISA specification. These codes are not output by the assembler and should not be used if the processor is programmed directly in binary. They actually map to the **NOP** instruction in the opcode decoder.

B.3.11 Miscellaneous Reserved Words

There are three other strings that may be used in the assembly language program and which are similar to preprocessor commands in C. The first is the .DEF construct which is used to alias constants. Any constant in decimal or hexadecimal notation can be referred to by an alphanumeric string. The mapping is done using the .DEF construct.

Example: `.DEF FOO 0x03E`

```
;; Fibonacci number sequence - definitions
;; allocate array label
.DEF          A_LOW                    800                      ; array of Fibonacci numbers
.DEF          NUM                      16                       ; number of Fibonacci numbers
;;; Fibonacci number sequence - initialization
              LDC        R7            1                        ; initialize R7 to 1
              LDC        R8            0                        ; initialize R8 to 0
              LDC        R10           0                        ; initialize counter to 0
              LDC        R11           A_LOW                    ; initialize array pointer base address
              ADD        R12           R11            R10       ; initialize array pointer
;;; Fibonacci number sequence - computation
LOOP:         ADD        R8            R7             R8        ; compute Fibonacci number
              COPY       R9            R8                       ; copy to R9 (output register for now)
              STI        R9            R12                      ; store data to memory
              INC        R10                                    ; update index
              ADD        R12           R11            R10       ; point to next array location
              ADD        R7            R7             R8        ; compute Fibonacci number
              COPY       R9            R7                       ; copy to R9
              STI        R9            R12                      ; store data to memory
              INC        R10                                    ; update index
              ADD        R12           R11            R10       ; point to next array location
              EQC        R10           NUM                      ; if we've computed NUM numbers,
              CTJUMP     CC[2]         READ_BACK:               ; jump to read back numbers
              JUMP       LOOP:                                  ; loop back
;;; Fibonacci number sequence - read back
READ_BACK:    DEC        R12                                    ; decrement pointer
              LDI        R13           R12                      ; read last Fibonacci number generated
              EQUAL      R12           R11                      ; if we're back to array base address,
              CTJUMP     CC[2]         END:                     ; jump to end
              JUMP       READ_BACK:                             ; loop back
END:          NOP
              NOP
              NOP
              $
```

Table B.16: Microcontroller instruction types.

In the example, FOO will be replaced by the binary equivalent of 0x03E in the assembler output. This works for positive and negative decimal constants, where the binary representation is 12 bit two's complement.

Labels are specified with colons and are mapped to instruction addresses by the assembler during its first pass. During the second pass, the labels are resolved to the instruction addresses they refer to.

Example: `LOOP1:  ZERO R8`

The example labels the address of the instruction `ZERO R8` with the label `LOOP1:`. Control flow instructions upstream or downstream in the code can then refer to the label as a target for branches.

Finally, comments are specified using the semicolon ;. Any characters appearing to the right of a semicolon are ignored by the assembler until a newline is reached.

B.4 Example Program: Fibonacci Numbers

Table B.16 is a summary listing of an example program which computes the first few numbers of the Fibonacci sequence. It begins by using the .DEF construct to alias the address of an array in data memory and the number of Fibonacci numbers to compute. Several constants are loaded to initialize the first few numbers of the series. A loop then computes the numbers in the series. A second loop reads these back in reverse order. Then the program terminates. Note the end of program character **$** to force the assembler to stop. This character is necessary for all programs.

Appendix C

Sensor DSP Nonlinear/Short Linear Filter Assembly Language [1]

Nonlinear and short linear filters are implemented using the VLIW architecture shown in Figure 2-5 and repeated in Figure C-1. Each functional unit, the Square-Accumulate Unit (SAC), the Multiply-Accumulate Unit (MAC), and the Load/Store Unit (LSU) implements its own small instruction set. To synchronize with the operation of the Distributed Arithmetic Unit, the total number of instruction slots for each of these units is only 8. This processor is a load/store architecture like the microcontroller unit, so the arithmetic operations use only registers as sources and destinations. The only memory interaction is with the filter buffer which acts as a delay line and implements relative addressing. This means that any address specified to the LSU is really an offset from the current buffer pointer backward in time to an earlier output sample.

Since the NLSL Unit simply does filtering operations, there is no notion of control flow in the instruction set architecture. All instructions are executed through every iteration of the program. The program is repeated every eight clock cycles.

C.1 Registers

A total of 16 registers are available to the NLSL program, of which three are special registers which access the current results in the accumulators of the different functional units. All values are 12 bit two's complement integers. Intermediate values are 24 bits, but these are truncated to 12 bits by specifying some configuration bits through the JTAG interface.

Table C.1 lists the register specifiers and codes for the registers available to the NLSL program. The **DA_ACC** register is loaded every clock cycle from the output of the distributed arithmetic unit. **ACC0** and **ACC1** are also loaded every clock cycle with the new truncated values from the SAC and MAC functional unit accumulators, respectively. The other registers are general purpose.

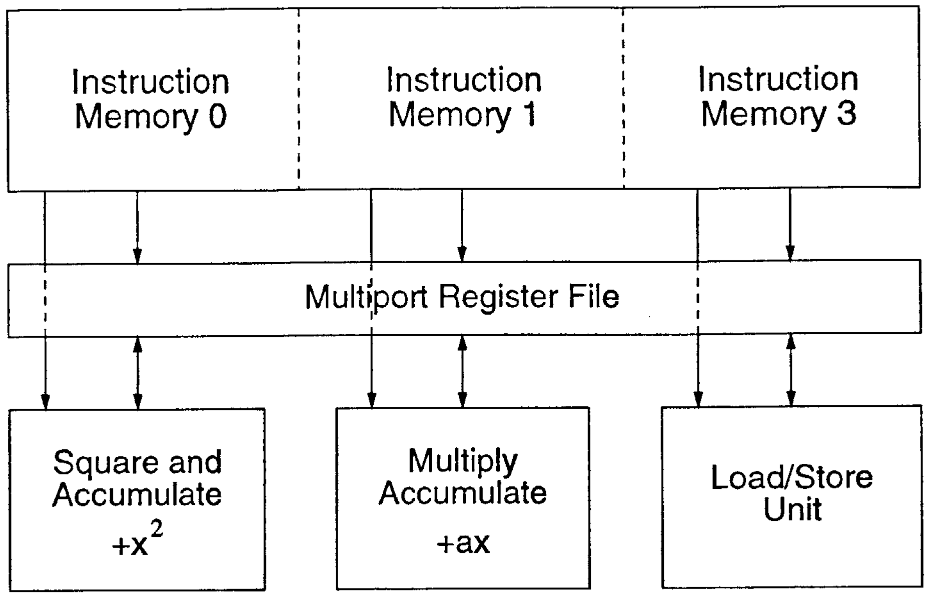


Figure C-1: Short linear and nonlinear filter implementation architecture.

Register Specifiers

Mnemonic	Opcode	Notes
Special Registers		
DA_ACC	0000	distributed arithmetic result
ACC0	0001	NLSL SAC ALU result
ACC1	0010	NLSL MAC ALU result
Other Registers		
R3-R15	0111-1111	

Table C.1: NLSL Unit registers.

C.2 Instruction Descriptions

This section describes in detail the instruction sets available to each of the functional units. Each instruction takes one clock cycle to execute. None of the units are pipelined so there are no pipeline hazards or bubbles. All writes are implicitly to the functional unit's accumulator except for loads and stores which require a destination register specifier.

C.2.1 SAC Instructions

Table C.2 summarizes the SAC Unit instructions, each of which will be described in detail below. All SAC unit arithmetic instructions take only one register specifier, the source register for the operand.

SAC Unit Instructions

Mnemonic	Opcode	Notes
SAC_NOP	000	null operation
SAC_SQAC	001	square-accumulate
SAC_SQRE	010	square
SAC_SQSU	011	square-subtract from accumulator
SAC_ADD	100	add to accumulator
SAC_SUB	101	subtract from accumulator
	110	unused
	111	unused

Table C.2: SAC Unit instructions and opcodes.

Example: `SAC_SQAC DA_ACC`
In the example, the register contents of **DA_ACC** are squared and added to the accumulator value. All arithmetic operations have the same syntax as the example.

SAC_NOP

SAC_NOP is the null operation. No writes occur to the SAC unit accumulator for this instruction.

SAC_SQAC

SAC_SQAC is the square-accumulate instruction. It adds the square of the contents of its argument to the SAC unit accumulator, leaving the result in the accumulator.

SAC_SQRE

SAC_SQRE is the square instruction. It reads the value of is argument register, squares it, and overwrites the contents of the accumulator with the result.

SAC_SQSU

The square-subtract instruction is **SAC_SQSU**. This operation subtracts the square of its operand from the current accumulator value and writes the result to the accumulator.

SAC_ADD

SAC_ADD simply adds the value of its operand to the contents of the accumulator. The result is written to the accumulator.

SAC_SUB

SAC_SUB is th complement of the SAC_ADD instruction above. It subtracts the value of its operand from the accumulator value, leaving the new result in the accumulator.

C.2.2 MAC Instructions

MAC Unit Instructions

Mnemonic	Opcode	Notes
MAC_NOP	000	null operation
MAC_MACC	001	multiply-accumulate
MAC_MDEC	010	multiply-subtract from accumulator
MAC_MULT	011	multiply
MAC_ADD	100	add
MAC_SUB	101	subtract
MAC_ACC	110	add to accumulator
MAC_DEC	111	subtract from accumulator

Table C.3: MAC Unit instructions and opcodes.

Table C.3 lists the instructions available to the MAC unit program. These instructions require either one or two register specifiers for operands and implicitly write their results to the MAC unit accumulator.

Example: `MAC_MULT ACC1 R3`

In the example, the contents of registers `ACC1` and `R3` are multiplied together and their product is written to the MAC unit accumulator. All MAC arithmetic instructions share the same syntax.

MAC_NOP

MAC_NOP is the null instruction. No writes are performed to the MAC accumulator when this instruction is executed.

MAC_MACC

MAC_MACC is the multiply-accumulate instruction. It computes the product of the values stored in its operand registers and adds the result to the accumulator value. The new result is written to the accumulator.

MAC_MDEC

MAC_MDEC is the complement of the MAC_MDEC instruction above. It subtracts the product of the values stored in its operands from the accumulator value, leaving the result of the subtraction in the accumulator.

MAC_MULT

The multiplication instruction is **MAC_MULT**. It simply computes the product of its argument values and overwrites the accumulator result with the product.

MAC_ADD

MAC_ADD is the addition instruction. It sums the values of its operand registers and replaces the accumulator value with the value of the sum.

MAC_SUB

MAC_SUB is the complementary subtraction instruction to the **MAC_ADD** instruction above. It computes the difference between its first and second operands. The result is written to the accumulator.

MAC_ACC

MAC_ACC is the accumulate instruction. It requires only one argument and adds the value in its operand register to the accumulator contents, leaving the result in the accumulator.

Example: `MAC_ACC R7`

In the example above, the contents of register `R7` are added to the current value of the MAC unit accumulator.

MAC_DEC

MAC_DEC subtracts the value stored in its lone argument register from the value stored in the accumulator. The result is written to the accumulator. The instruction syntax for this operation is the same as in the example above for **MAC_ACC**.

C.2.3 LSU Instructions

LSU Unit Instructions

Mnemonic	Opcode	Notes
LSU_NOP	000	null operation
LSU_LDC	001	load constant to register
LSU_LOAD	010	load direct from memory to register
LSU_LDI	011	load indirect from memory to register
LSU_COPY	100	copy from one register to another
	101	unused
	110	unused
	111	unused

Table C.4: LSU Unit instructions and opcodes.

Table C.4 lists the instructions available to the LSU Unit. These instructions are responsible for loading data to the registers from the instruction stream (loading constants) or previous filter outputs stored in the filter buffer. Note that there is no store instruction as stores are done implicitly to the filter buffer every eight cycles. Each LSU instruction requires two arguments: a destination register and a source register, constant, or memory address.

Example: `LSU_LOAD R3 0x010000010 ; load R3 with ACC0[k-2]`

In the example, register `R3` is loaded with the data value from the filter buffer corresponding to the output of the `ACC0` accumulator at time $k-2$, where k (the current time) corresponds to the current value of the filter buffer pointer. There are three specifiers corresponding to the buffers after each of the filtering units. These specifiers precede the 7 bit address corresponding to the delay from the current sample: `0x00` corresponds to results from the distributed arithmetic unit, the `DA_ACC` register; `0x01` corresponds to the results from the SAC unit accumulator `ACC0`; finally, `0x10` corresponds to results from the MAC unit accumulator `ACC1`.

LSU_NOP

LSU_NOP is the null operation for the LSU unit. No results are written to any of the registers for this instruction.

LSU_LDC

LSU_LDC is the load constant instruction. It takes a constant specified in the instruction stream and writes it to the register specified as its destination.

LSU_LOAD

LSU_LOAD is the direct memory load instruction. It accesses the filter buffer value corresponding to the appropriate filter unit result and time offset into the previous samples and writes this value into the destination register.

LSU_LDI

LSU_LDI is the indirect memory load instruction. Instead of using an explicit offset into the filter buffer, it uses the value stored in its second register argument as the filter buffer address. It takes the value stored in the filter buffer and writes it into the destination register.

LSU_COPY

LSU_COPY is the register to register copy instruction. It takes two register specifiers as arguments and copies the value from the source register to the destination register.

Appendix D

SensorDSP Sample Microcode

D.1 Micro-controller Heart Detection Code

```
;;;* Last edited: Mar 18 19:56 1998 (mirth)
;; -----------------------------------------------------------------------
;; Initialization Code for Heartbeat Detection
;; -----------------------------------------------------------------------
;; Need register storage for 4 constants:
;; NOTE: Make the constants as small as possible to avoid overflow!
;;
.DEF THR_SCALE   20                             ; threshold scaling factor
.DEF SEG_ON_THR     90                             ; segment on threshold
.DEF SEG_OFF_THR        85                             ; segment off threshold
.DEF AVG_LEN            40                             ; energy averaging window length
.DEF ENERGY_SCALE 16 ; premultiply scaling
.DEF TRUE 1 ; boolean true
.DEF FALSE 0 ; boolean false
.DEF SEGMENT_ON_WAIT 20 ; number of initial samples to wait
.DEF SEGMENT_WAIT 23 ; samples from segment beginning
;; ; one less then C code because of
;; ; timing
;;
;; Memory mapping for filter outputs:
.DEF Yk_BASE 0x000 ; correlation filter base address
.DEF Ek_BASE 0x080 ; energy base address
.DEF uEk_BASE 0x100 ; mean energy base address
;;
;; Assign registers to values - saves cycles on this tight loop
;; DA_ACC : correlation filter output y[k]
;; ACC0   : energy E[k]
;; ACC1   : mean energy uE[k]
;; R8 : input sample count
;; R9 : segment beginning offset counter
;; R13    : SEG_ON_THR
;; R14    : SEG_OFF_THR
;; R15    : ENERGY_SCALE
;;
;; NOTE: ZERO macro won't work with Verilog's unknown value (X)
;;
;; Timing works as follows: Once the DA unit is enabled,
;; it takes eight cycles after that instruction issues before
;; valid data is ready for the NLSL unit.  After that, it takes
;; another eight cycles before valid data is read into the
;; segment buffer.  Thus, from the time the DA unit is enabled
;; there are 16 cycles before the MUCTRL unit must act on the
;; incoming data.
;;
INIT_PROG:      CONFIG  FAST_MODE      CLR ; clear fast mode
```

```
                CONFIG  DATA_VAL        CLR ; clear valid data bit
;;
;; Initialize variables for first data pass
;;
                LDC     R13             SEG_ON_THR ; initialize values
                LDC     R14             SEG_OFF_THR
                LDC     R15             ENERGY_SCALE
                LDC     R8              SEGMENT_ON_WAIT ; initialize input sample count
                LDC     R9              SEGMENT_WAIT ; initialize segment counter
                CONFIG  ALL_ENABLE      SET ; activate all units
;;
;; Ignore initial samples while average energy is computed.
;;
SKIP_DATA:      CFJUMP  DATA            SKIP_DATA: ; if no data, keep checking
                CONFIG  DATA_VAL        CLR ; clear valid bit
                DEC     R8 ; decrement sample count
                EQC     R8              0 ; if not on last sample,
                CTJUMP  SIGN12      SKIP_DATA: ; continue data skip
;;                                             ; otw, start segmenting
;;
;; NOTE: The code preceding this point should only run ONCE per
;; activation of this
;; algorithm. There should not be any looping back to before this point...
;;
;; ------------------------------------------------------------------------
;; Segmentation Code for Heartbeat Detection: < 8 Cycle Timing
;; ------------------------------------------------------------------------
;; R10   : segment lower bound
;; R11   : segment upper bound
;;
;; Look for segment beginning:
;; AVG_LEN*THR_SCALE*E[k] - SEG_ON_THR*uE[k] >= 0 ->
;; AVG_LEN*THR_SCALE*E[k] >= SEG_ON_THR*uE[k] -> sign bit of difference
;;                                              should be 0
;;
SEG_FALSE:      CFJUMP  DATA            SEG_FALSE: ; if no data, keep checking
                CONFIG  DATA_VAL        CLR  ; clear valid bit
                MULT    ACC0    R15          ; AVG_LEN*THR_SCALE*E[k]
                MDEC    ACC1    R13          ; - SEG_ON_THR*uE[k]
                CTJUMP  SIGN24          SEG_FALSE: ; if not, keep looking
;;                                             ; otw, we're in a segment
                COPY    R10     BUFPTR         ; copy segment lower bound
;;
;; Wait for some segment cycles to pass:
;;
SEG_WAIT:       CFJUMP  DATA            SEG_WAIT: ; if no data, keep checking
                CONFIG  DATA_VAL        CLR ; clear valid bit
                DEC     R9                  ; decrement segment counter
                EQC     R9              0   ; while less than delay
                CTJUMP  SIGN12          SEG_WAIT: ; keep looping
;;
;; Look for segment ending:
;; SEG_OFF_THR*uE[k] - AVG_LEN*THR_SCALE*E[k] >= 0 ->
;; SEG_OFF_THR*uE[k] >= AVG_LEN*THR_SCALE*E[k] -> sign bit of
;;                                            difference should be 0
;;
SEG_TRUE:       CFJUMP  DATA            SEG_TRUE: ; if no data, keep checking
                CONFIG  DATA_VAL        CLR       ; clear valid bit
                MULT    ACC1    R14               ; SEG_OFF_THR*uE[k]
                MDEC    ACC0    R15               ; - AVG_LEN*THR_SCALE*E[k]
                CTJUMP  SIGN24          SEG_TRUE: ; if not, keep looking
;;                                          ; otw, start segment processing
                COPY    R11     BUFPTR      ; copy segment upper bound
                CONFIG  ALL_ENABLE      CLR ; disable filters
                CONFIG  FAST_MODE       SET ; set fast mode
;;
;; ------------------------------------------------------------------------
;; Feature Extraction
;; ------------------------------------------------------------------------
;; Feature 1: Look for peak value of matched filter output and
;; its location in the buffer.
;; R7 : peak output
;; R8 : peak output location
;; R10 : segment lower bound
;; R11 : segment upper bound
;; R12 : segment memory offset
;; R13 : segment memory pointer
;; R14 : peak energy of filter output
;; R15 : temp
;;
;; Allocate feature vector storage:
; .DEF FEATURES[7] 0x300
.DEF FEATURES[7] 0x180
;;
;; Feature offsets (for 160Hz sample rate -
;; see /homes/mirth/C/medclassifier/heartdetect.h):
;; NOTE: The signs change since in the C program the buffer is
;; implemented as a shift
;; register while in Verilog it is a memory with a pointer.
.DEF FST_VLY_AFT  -11
.DEF FST_VLY_BEF   11
.DEF FST_PEK_AFT  -22
.DEF FST_PEK_BEF   22
.DEF THRESH_SCALE  100
.DEF PEAKWIDTH     50
.DEF LOW_ADDR    127
.DEF BUF_DEPTH    128
;;
FEAT_EXT:       COPY    R12     R10             ; initialize offset to 0
                LDC     R13             Yk_BASE ; load y[k] base address
                ADD     R13     R13     R12     ; add within segment offset
                LDI     R7      R13             ; load first value of y[k]
                COPY    R8      R13             ; load location of first value
                LDC     R13             Ek_BASE ; load E[k] base address
                ADD     R13     R13     R12     ; add within segment offset
                LDI     R14     R13             ; load first value of E[k]
FEAT1_LOOP:     INC     R12             ; increment within segment offset
                LDC     R9              LOW_ADDR ; load address bitmask
                AND     R12     R12     R9       ; mask off address
                EQUAL   R12     R11              ; outside segment?
                CTJUMP  WNOR12          SAVE_FEAT1: ; if yes, store features
                LDC     R13             Yk_BASE  ; load y[k] base address
                ADD     R13     R13     R12      ; add within segment offset
                LDI     R15     R13              ; load next value of y[k]
                GT      R15     R7               ; y[k] > ymax
                CFJUMP  SIGN12          ENERGY_FEAT: ; if true, swap
                COPY    R7      R15              ; ymax = y[k]
                COPY    R8      R13              ; kmax = k
ENERGY_FEAT:    LDC     R13             Ek_BASE  ; load E[k] base address
                ADD     R13     R13     R12      ; add within segment offset
                LDI     R15     R13              ; load next value of E[k]
                GT      R15     R14              ; E[k] > Emax
                CFJUMP  SIGN12          FEAT1_LOOP: ; if true, swap
                COPY    R14     R15                 ; Emax = E[k]
                JUMP                    FEAT1_LOOP: ; keep looping
SAVE_FEAT1:     STOR    R7              FEATURES[0] ; peak output feature
                ADDC    R13     R8      FST_VLY_AFT ; first valley after
                                                    ;peak location
                LDC     R9              LOW_ADDR    ; load address bitmask
                AND     R13     R13     R9          ; mask off address
BOUND1_OK:      LDI     R15     R13                 ; get value
SAVE_FEAT2:     STOR    R15             FEATURES[1] ; store to array
                ADDC    R13     R8      FST_VLY_BEF ; first valley before
                                                    ; peak location
                LDC     R9              LOW_ADDR    ; load address bitmask
                AND     R13     R13     R9          ; mask off address
```

```
BOUND2_OK:     LDI    R15    R13                 ; get value
SAVE_FEAT3:    STOR   R15           FEATURES[2]  ; store to array
               ADDC   R13    R8     FST_PEK_AFT  ; first peak after
                                                 ; peak location
               LDC    R9     LOW_ADDR            ; load address bitmask
               AND    R13    R13    R9           ; mask off address
BOUND3_OK:     LDI    R15    R13                 ; get value
SAVE_FEAT4:    STOR   R15    FEATURES[3]         ; store to array
               ADDC   R13    R8     FST_PEK_BEF  ; first peak before
                                                 ; peak location
               LDC    R9            LOW_ADDR     ; load address bitmask
               AND    R13    R13    R9           ; mask off address
BOUND4_OK:     LDI    R15    R13                 ; get value
SAVE_FEAT5:    STOR   R15           FEATURES[4]  ; store to array
               STOR   R14           FEATURES[5]  ; peak energy feature
               LDC    R14           THRESH_SCALE ; load threshold
                                                 ; scaling for width
               COPY   R13    R8                  ; copy peak location
L_LOC:         DEC    R13                        ; decrement pointer
               LDC    R9            LOW_ADDR     ; load address bitmask
               AND    R13    R13    R9           ; mask off address
               LDI    R12    R13                 ; load y[k]
               MULT   R12    R14                 ; THRESH_SCALE*y[k]
               LDC    R9     PEAKWIDTH           ; load peak width scaling
               MDEC   R7     R9       ; THRESH_SCALE*y[k]-PEAKWIDTH*ymax
               CFJUMP SIGN24        L_LOC:       ; if not, keep looking
               COPY   R15    R13                 ; otw, copy pointer
               COPY   R13    R8                  ; copy peak location
U_LOC:         INC    R13                        ; increment pointer
               LDC    R9            LOW_ADDR     ; load address bitmask
               AND    R13    R13    R9           ; mask off address
               LDI    R12    R13                 ; load y[k]
               MULT   R12    R14                 ; THRESH_SCALE*y[k]
               LDC    R9     PEAKWIDTH           ; load peak width scaling
               MDEC   R7     R9       ; THRESH_SCALE*y[k]-PEAKWIDTH*ymax
               CFJUMP SIGN24        U_LOC:       ; if not, keep looking
               SUB    R15    R13    R15          ; otw, compute peak width
               CFJUMP SIGN12   SAVE_FEAT6: ; if positive, store features[6]
               LDC    R9       BUF_DEPTH ; load buffer depth constant
               ADD    R15    R15    R9      ; otw, compute relative value
SAVE_FEAT6:    STOR   R15           FEATURES[6] ; store to array
;;
;; ---------------------------------------------------------------------
;; Classification
;; ---------------------------------------------------------------------
;;
;; Define means and covariance matrices as constants (from C program).
;; HEART CLASS:
.DEF H_MEANS0  376
.DEF H_MEANS1 -204
.DEF H_MEANS2 -232
.DEF H_MEANS3  108
.DEF H_MEANS4  136
.DEF H_MEANS5  402
.DEF H_MEANS6    7
;;
;; NOTE: Many elements of the matrix for D8F9S8 are 0!
.DEF H_COV00     1
.DEF H_COV06    29
.DEF H_COV11     1
.DEF H_COV16    -3
.DEF H_COV26     8
.DEF H_COV36    -5
.DEF H_COV46     7
.DEF H_COV56    -7
.DEF H_COV60    29
.DEF H_COV61    -3
.DEF H_COV62     8
```

```
.DEF H_COV63    -5
.DEF H_COV64     7
.DEF H_COV65    -7
.DEF H_COV66  1862
;;
;; NONHEART CLASS:
.DEF NH_MEANS0  149
.DEF NH_MEANS1  -81
.DEF NH_MEANS2  -87
.DEF NH_MEANS3   24
.DEF NH_MEANS4   34
.DEF NH_MEANS5  109
.DEF NH_MEANS6    7
;;
;; NOTE: Many elements of the matrix for D8F9S8 are 0!
.DEF NH_COV00     1
.DEF NH_COV06     7
.DEF NH_COV16     3
.DEF NH_COV26     2
.DEF NH_COV36     1
.DEF NH_COV46    -3
.DEF NH_COV56    -1
.DEF NH_COV60     7
.DEF NH_COV61     3
.DEF NH_COV62     2
.DEF NH_COV63     1
.DEF NH_COV64    -3
.DEF NH_COV65    -1
.DEF NH_COV66   260
;;
;; Precomputed Classification Parameters:
;;
.DEF PRE_COV11    0
;; example: MACC R7 R7 ; + x[1]*PRE_COV11*x[1]
.DEF PRE_COV06   26
.DEF PRE_COV16   -8
.DEF PRE_COV26    6
.DEF PRE_COV36   -7
.DEF PRE_COV46   11
.DEF PRE_COV56  -10
.DEF PRE_COV66 1553
;;
.DEF PRE_COV06X2   52
.DEF PRE_COV16X2  -16
.DEF PRE_COV26X2   12
.DEF PRE_COV36X2  -14
.DEF PRE_COV46X2   22
.DEF PRE_COV56X2  -20
;;
.DEF PRE_MU0   -840
.DEF PRE_MU1    360
.DEF PRE_MU2    -84
.DEF PRE_MU3     98
.DEF PRE_MU4   -154
.DEF PRE_MU5    140
.DEF PRE_MU6LO 0x34E
.DEF PRE_MU6HI 0xFF7
;;
.DEF PRE_DIFLO 0x709
.DEF PRE_DIFHI 0x051
;;
;; Classes and Output:
.DEF HEART_CLASS    0
.DEF NONHEART_CLASS 1
.DEF CLASS_OUT     0xFFF
;;
;; The matrix multiplications require quite a number of cycles to compute because
;; of the 24 bit data that must be shuttled around.
```

```
;;
;; Scaled feature means difference:
      LOAD     R7          FEATURES[0]    ; get x[0]
      LDC      R8          PRE_MU0        ; load PRE_MU0
      MULT     R7      R8                 ; x[0]*PRE_MU0
      LOAD     R7          FEATURES[1]    ; get x[1]
      LDC      R8          PRE_MU1        ; load PRE_MU1
      MACC     R7      R8                 ; + x[1]*PRE_MU1
      LOAD     R7          FEATURES[2]    ; get x[2]
      LDC      R8          PRE_MU2        ; load PRE_MU2
      MACC     R7      R8                 ; + x[2]*PRE_MU2
      LOAD     R7          FEATURES[3]    ; get x[3]
      LDC      R8          PRE_MU3        ; load PRE_MU3
      MACC     R7      R8                 ; + x[3]*PRE_MU3
      LOAD     R7          FEATURES[4]    ; get x[4]
      LDC      R8          PRE_MU4        ; load PRE_MU4
      MACC     R7      R8                 ; + x[4]*PRE_MU4
      LOAD     R7          FEATURES[5]    ; get x[5]
      LDC      R8          PRE_MU5        ; load PRE_MU5
      MACC     R7      R8                 ; + x[5]*PRE_MU5
      COPY     R14     MUACCH             ; save high word
      COPY     R15     MUACCL             ; save low word
      LOAD     R7      FEATURES[6]        ; get x[6]
      LDC      R8      PRE_MU6HI          ; get PRE_MU6 high word
      MULT     R7      R8                 ; x[6]*PRE_MU6 high
      LDC      R9            1
      MACC     R14     R9                 ; add previous sum high
      LDC      R8          PRE_DIFHI      ; get PRE_DIFHI
      MACC     R8      R9                 ; add PRE_DIFHI
      COPY     MUACCH  MUACCL             ; shift by 12
      LDC      MUACCL        0
      LDC      R8          PRE_MU6LO      ; get PRE_MU6 low word
      MACC     R7      R8                 ; + x[6]*PRE_MU6 low
      MACC     R15     R9                 ; add previous sum low
      LDC      R8          PRE_DIFLO      ; get PRE_DIFLO
      MACC     R8      R9                 ; add PRE_DIFLO to sum
      COPY     R14     MUACCH             ; save high word
      COPY     R15     MUACCL             ; save low word
;; Covariance difference matrix:
      LOAD     R7          FEATURES[0] ; get x[0]
      LDC      R8          PRE_COV06X2 ; load 2*PRE_COV06
      MULT     R7      R8              ; x[0]*2*PRE_COV06
      LOAD     R7          FEATURES[1] ; get x[1]
      LDC      R8          PRE_COV16X2 ; load 2*PRE_COV16
      MACC     R7      R8              ; + x[1]*2*PRE_COV16
      LOAD     R7          FEATURES[2] ; get x[2]
      LDC      R8          PRE_COV26X2 ; load 2*PRE_COV26
      MACC     R7      R8              ; + x[2]*2*PRE_COV26
      LOAD     R7          FEATURES[3] ; get x[3]
      LDC      R8          PRE_COV36X2 ; load 2*PRE_COV36
      MACC     R7      R8              ; + x[3]*2*PRE_COV36
      LOAD     R7          FEATURES[4] ; get x[4]
      LDC      R8          PRE_COV46X2 ; load 2*PRE_COV46
      MACC     R7      R8 ; + x[4]*2*PRE_COV46
      LOAD     R7          FEATURES[5] ; get x[5]
      LDC      R8          PRE_COV56X2 ; load 2*PRE_COV56
      MACC     R7      R8 ; + x[5]*2*PRE_COV56
      LOAD     R7          FEATURES[6] ; get x[6]
      LDC      R8          PRE_COV66 ; load PRE_COV66
      MACC     R7      R8 ; + x[6]*PRE_COV66
      COPY     R12     MUACCH ; copy high word
      COPY     R13     MUACCL ; copy low word
      MULT     R7      R12 ; multiply by previous high
      MACC     R14     R9 ; add difference high
      COPY     MUACCH  MUACCL ; shift by 12
      LDC      MUACCL      0
      MACC     R7      R13 ; mulitply by previous low
      MACC     R15     R9 ; add difference low
```

```
       CFJUMP   SIGN24         NONHEART: ; if not heart, skip
       LDC      R8             HEART_CLASS ; otw, write class
       JUMP                    SEG_LOOP_INIT: ; start over
NONHEART:       LDC     R8     NONHEART_CLASS ; otw, write class
;;
;; -----------------------------------------------------------------------
;; Segmentation Code for Heartbeat Detection: Buffer Catch-Up Timing
;; -----------------------------------------------------------------------
;;
;; Initialize variables again for further segmentation
;;
SEG_LOOP_INIT:  STOR    R8             CLASS_OUT
;;
;; Initialize variables for first data pass
;;
                LDC     R13         SEG_ON_THR ; initialize values
                LDC     R14         SEG_OFF_THR
                LDC     R15         ENERGY_SCALE
                LDC     R9          SEGMENT_WAIT ; initialize segment counter
                CONFIG  FAST_MODE       CLR ; clear fast mode
CONFIG DATA_VAL CLR ; clear valid data bit
                CONFIG  ALL_ENABLE      SET     ; activate all units
                JUMP                SEG_FALSE: ; start segmentation again
;;
;; -----------------------------------------------------------------------
;; Test Structures: Stubs, Drivers, and Old Code
;; -----------------------------------------------------------------------
;; Dump registers: Dumps register outputs through ALU so we can see the contents in
;; Verilog.
;; ADDC R7 R7 0 ; dump register
;; ADDC R8 R8 0 ; dump register
;; ADDC R9 R9 0 ; dump register
;; ADDC R10 R10 0 ; dump register
;; ADDC R11 R11 0 ; dump register
;; ADDC R12 R12 0 ; dump register
;; ADDC R13 R13 0 ; dump register
;; ADDC R14 R14 0 ; dump register
;; ADDC R15 R15 0 ; dump register
\$                                ; end of file marker
```

D.2 NLSL SAC Heart Detection Code

```
SAC_SQAC DA_ACC ; square and accumulate DA filter output
SAC_SQSU R3 ; substract square of previous DA output
SAC_NOP
SAC_NOP
SAC_NOP
SAC_NOP
SAC_NOP
SAC_NOP
```

D.3 NLSL MAC Heart Detection Code

```
MAC_NOP
MAC_NOP
MAC_ACC ACC0 ; add current energy to energy sum
MAC_DEC R4   ; subtract previous energy
MAC_NOP
MAC_NOP
MAC_NOP
MAC_NOP
```

D.4 NLSL LSU Heart Detection Code

```
LSU_LOAD R3 0x000110100 ; load previous DA filter output
LSU_LOAD R4 0x010100111 ; load previous energy output
LSU_LOAD
LSU_LOAD
LSU_LOAD
LSU_LOAD
LSU_LOAD
LSU_LOAD
```

References

[1] R. Amirtharajah, *Design of Low Power VLSI Systems Powered by Ambient Mechanical Vibration*, Ph.D. Thesis, Massachusetts Institute of Technology, May 1999.

[2] R. Amirtharajah and A. Chandrakasan, "Self-powered signal processing using vibration-based power generation," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 5, pp. 687-695, May 1998.

[3] S. Meninger, "A Low Power Controller for a MEMS Based Energy Converter," M.S. Thesis, Massachusetts Institute of Technology, May 1999.

[4] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1989.

[5] A. Peled and B. Liu, "A new hardware realization of digital filters," *IEEE Trans. ASSP,* vol. ASSP-22, no. 6, pp. 456-462, December 1974.

[6] S. A. White, "Applications of distributed arithmetic to digital signal processing: A tutorial review," *IEEE ASSP Magazine,* pp. 4-19, July 1989.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1st. edition, 1990.

[8] Cypress Semiconductor Corp., "CY7C374i 128-Macrocell Flash CPLD Data Sheet", July 1998.

[9] Fairchild Semiconductor Corp., "NM27C512 64K$\times$8 High Performance CMOS EPROM Data Sheet", July 1998.

[10] Standards Committee IEEE, "IEEE standard test access port and boundary scan architecture," IEEE Standard 1149.1-1990, 1990.

[11] Analog Devices, Inc.,"Low Cost Signal Conditioning 8-Bit Analog to Digital Converter Data Sheet", April 1989.

[12] Maxim Integrated Products, "Ultra Low-Power, Single/Dual Supply Comparators Data Sheet", March 1995.

[13] National Semiconductor Corp., "LM317 Adjustable Voltage Regulator Data Sheet", May 1999.

[14] National Semiconductor Corp., "LM340 Positive Fixed Voltage Regulator Data Sheet", May 1999.

3400 - 68