Real-Time Digital Signal Processing Using the Motorola DSP5630xEVM

Collected Notes from EE442/EE592

Phillip L. De Leon
New Mexico State University
Klipsch School of Electrical and Computer Engineering
Box 30001, Dept. 3-O
Las Cruces, New Mexico 88003-8001
(575) 646-DSP1
pdeleon@nmsu.edu

©2011 Phillip De Leon. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the author. All rights reserved.

Contents

Preface **1**

1 Introduction to Real-Time DSP **3**
1.1 RTDSP Basics . . . 3
1.1.1 DSP Chip History . . . 3
1.1.2 Applications . . . 4
1.1.3 Real-Time Software . . . 5

2 Introduction to the Motorola DSP56300 Family **7**
2.1 Internal Architecture and Programming Model . . . 7
2.2 DSP56300 Architecture Introduction . . . 9
2.3 Addressing Modes . . . 43
2.4 Parallel Moves . . . 67
2.5 Instruction Set I . . . 83
2.6 Additional DSP5630x Instructions . . . 116
2.6.1 MPYI . . . 116
2.6.2 Multi-bit Shift . . . 116
2.7 DSP5630x Architecture & Programming Model . . . 116
2.8 Motorola DSP56302EVM . . . 133
2.9 Problems . . . 135

3 Development Tools **139**
3.1 Getting Started with the Motorola DSP5630xEVM . . . 139
3.1.1 Background . . . 139
3.1.2 DSP5630xEVM Self-Test . . . 140
3.1.3 Demonstration File . . . 140
3.1.4 Example Test Program . . . 140
3.2 Motorola DSP56300 Assembler . . . 141
3.2.1 Equate (EQU) . . . 141
3.2.2 Originate (ORG) . . . 142
3.2.3 Define Storage (DS) . . . 143
3.2.4 Define Constant (DC) . . . 143
3.2.5 Define Storage Modulo (DSM) . . . 144
3.2.6 Other Examples . . . 145
3.3 Domain Technologies Debug-56K . . . 145
3.3.1 Tips on Debugging . . . 145
3.3.2 In-line Assembly . . . 146
3.4 Problems . . . 148

4 Simple Programs **149**
4.1 Simple 5630x Programs . . . 149
4.1.1 Program 1 . . . 149
4.1.2 Program 2 . . . 150
4.1.3 Program 3 . . . 150
4.2 PASS.ASM–Audio Pass Through . . . 150
4.2.1 PASS.ASM . . . 151
4.2.2 PASS.DAT . . . 153
4.2.3 PROGINIT.ASM . . . 154
4.2.4 PROCSTER.ASM . . . 154
4.2.5 ADA_EQU.ASM . . . 154
4.2.6 ADA_INIT.ASM . . . 155
4.2.7 INT_EQU.ASM . . . 155
4.2.8 IOEQU.ASM . . . 155
4.2.9 VECTORS.ASM . . . 155
4.3 Overview of CS4215 Codec Operation . . . 155
4.4 Real-Time Digital FIR Filters . . . 156
4.4.1 Program 4 . . . 156
4.4.2 Program 5 . . . 157
4.4.3 Program 6 . . . 158
4.4.4 Program 6 Solution . . . 158
4.5 Real-Time Digital IIR Filters . . . 159
4.6 Information Sources for DSP5630x Development . . . 159
4.7 Problems . . . 161

5 Sound Field Simulator **163**
5.1 Background . . . 163
5.2 Concert Hall Acoustics . . . 163
5.2.1 Sound Propagation . . . 163
5.2.2 Direct Sound, Early Reflections, and Reverberation . . . 163
5.2.3 Concert Hall Attributes . . . 166
5.3 Digital Synthesis of the Sound Field . . . 168
5.3.1 Digital Synthesis of Early Reflections . . . 168
5.3.2 Digital Synthesis of Reverberation . . . 169
5.3.3 Comb Filter . . . 169
5.3.4 All-Pass Filters . . . 174
5.4 Implementation . . . 174
5.4.1 Algorithm . . . 176
5.4.2 Memory Layout . . . 178
5.5 Testing . . . 181
5.6 Moorer's Sound Field Simulator . . . 182
5.6.1 Tapped Delay Line . . . 182
5.6.2 Comb Filter . . . 182
5.6.3 All Pass Filter . . . 187
5.6.4 Implementation . . . 187

6 Adaptive Noise Cancellation **189**
6.1 Optimal Filtering . . . 189
6.1.1 Wiener Filter . . . 190
6.2 Adaptive Filtering . . . 192
6.2.1 Applications of Adaptive Filters . . . 192
6.2.2 Adaptive Filter Processes . . . 192
6.2.3 Steepest Descent Algorithm . . . 193

6.2.4 Analysis of the Steepest Descent Algorithm 193
6.2.5 LMS Algorithm 194
6.2.6 NLMS Algorithm 195
6.3 Adaptive Cancellation of Sinusoidal Interference 195
6.3.1 Adaptive Noise Canceler 195
6.4 ANC Implementation 199
6.4.1 Algorithm 200
6.4.2 Layout in Memory 201
6.4.3 Adaptive Line Enhancer 203
6.5 ALE Implementation 207
6.5.1 Algorithm 207
6.5.2 Layout in Memory 208
6.6 Fixed-Point Issues 210
6.6.1 ANC 210
6.6.2 ALE 211
6.7 Code Management 212
6.7.1 Subroutines as Separate Files 212
6.7.2 Macros 213

7 Wavetable Synthesis 217
7.1 Background 217
7.2 Sine Wave Synthesis 217
7.3 Generating the Lookup Table 219
7.4 Sinusoid Synthesis with Integer Delta 220
7.5 Sinusoid Synthesis with Real Delta: Round Down Method 222
7.5.1 Algorithm for Round Down Method 224
7.5.2 Code for Round Down Method 224
7.6 Sinusoid Synthesis with Real Delta: Linear Interpolation Method 226
7.6.1 Algorithm for Linear Interpolation Method 227
7.6.2 Code for Linear Interpolation Method 229
7.7 ADSR Dynamics 231
7.8 Envelope Generation 234
7.9 Synthesizing Music 237
7.10 Implementation 239
7.10.1 Algorithm 239
7.10.2 Layout in Memory 241
7.11 Using Flag Bits for Program Control 249
7.12 Using the DSP56302 Triple Timer Module for Program Control 250

8 Modem 251
8.1 Background 251
8.2 Binary Phase-Shift Keying Modem (Transmitter) 251
8.2.1 Amplitude Shift Keying 252
8.2.2 Frequency Shift Keying 252
8.2.3 Phase Shift Keying 253
8.2.4 Other Comments 253
8.2.5 Combined Modulation Schemes 254
8.3 Modem Design for the Telephone Channel 256
8.4 Design Specifications BPSK Modem (Transmitter) 257
8.5 Implementation–Transmitter 259
8.5.1 Algorithm 259
8.5.2 Layout in Memory 260
8.6 Binary Phase-Shift Keying Modem (Receiver) 262

8.6.1 Maximum-Likelihood Detector 262
8.6.2 Sampled Matched Filter 263
8.6.3 Practical Issues 266
8.7 Implementation–Receiver 268
8.7.1 Algorithm 268

9 Real-Time Frequency-Domain Processing 273
9.1 Background 273
9.2 Discrete Fourier Transform Review 273
9.2.1 Matrix Form of the DFT 273
9.3 Fast Fourier Transform 274
9.3.1 Decimation in Time FFT 275
9.4 FFTs on the Motorola DSP5630x 278
9.4.1 Required Hardware Support for the FFT Calculation 278
9.4.2 DIT Butterfly Kernal 278
9.4.3 DIT FFT Outline 280
9.4.4 Bit Reversing 281
9.4.5 Inverse FFT 282
9.4.6 Scaling Issues 283
9.4.7 Available FFT Macros for the DSP56K 283
9.4.8 Comments 283
9.5 Sample DSP56K Codes for Computing the FFT 285
9.5.1 Example 1: Simple code for an FFT using FFTR2A (non-real-time) 285
9.5.2 Example 2: Simple code for an FFT using FFTR2CN (non-real-time) 286
9.6 Code for the IFFT 287
9.7 Continuous, Real-Time Spectral Processing 288
9.7.1 Single Sample Period FFT 288
9.7.2 Multiple Sample Period FFT 289

A Motorola Pass Pack 293
A.1 PASS.ASM 293
A.2 ADA_INIT.ASM 295
A.3 ADA_EQU.ASM 299
A.4 INTEQU.ASM 300
A.5 IOEQU.ASM 302
A.6 VECTORS.ASM 314

B Modified Pass Pack 321
B.1 PASS.ASM 321
B.2 PASS.DAT 323
B.3 PROGINIT.ASM 325
B.4 PROCSTER.ASM 326
B.5 ADA_INIT.ASM 327
B.6 ADA_EQU.ASM 327
B.7 INTEQU.ASM 327
B.8 IOEQU.ASM 327
B.9 VECTORS.ASM 327

Index 328

Preface

This book is a compilation of course notes written during development of a course in Real-Time Digital Signal Processing (DSP) at New Mexico State University. From the beginning, the course has sought to give students practice in writing real-time DSP applications through a series of two-week projects. The course does not attempt to write a "pass through" code (often referred to as `pass.asm`) which is the skeleton code that performs all initialization and moves samples in and out of the digital signal processor (DSP)[1]. This code is supplied with virtually all evaluation modules (EVMs) and many third party development boards. We do, however, dissect a slightly modified version of `pass.asm` that is supplied with the Motorola DSP5630xEVM so that students understand the workings of a pass through code.[2]

The book is organized so that the first four chapters introduce the reader to the basic concepts involved in programming the DSP5630x and the remaining chapters describe projects and applications. The outline of the text is as follows.

Chapter 1 provides a short history of DSP chips, reasons why DSPs are used in many systems, and some preliminaries of real-time programming.

Chapter 2 introduces Motorola's 3rd generation DSP, the DSP56300. This introduction describes the internal architecture and programming model, addressing modes, parallel moves, and instruction set of the DSP56300 family. In addition, we introduce the DSP5630xEVM.

Chapter 3 describes the use of the development tools namely the Motorola DSP56300 assembler and Domain Technologies Debug-EVM debugger.

Chapter 4 reviews `pass.asm` and illustrates the development process by going through several examples which illustrate real-time FIR and IIR filters.

Chapter 5 describes a "sound-field simulator" project that filters audio through a system whose impulse response resembles that of an large acoustical environment. While the system will probably not sound like Boston Symphony Hall, the project does illustrate basic fixed-point arithmetic techniques and addressing used in digital filtering.

Chapter 6 describes an "adaptive noise canceler" project that cancels a sinusoidal noise component (with unknown frequency) while preserving a signal of interest. This project further exposes the reader to filtering, this time with a filter whose coefficients are time-varying and adjusted with an adaptive algorithm.

Chapter 7 describes a "wavetable synthesizer" project which will modulate a wave (tone) to mimic a synthetic instrument. This project exposes the reader to lookup tables and waveform synthesis as well as the use of counters, flags, and other common programming techniques.

Chapter 8 describes a "BPSK modem" project that will introduce students to DSP-based digital transmitters and receivers. This project consists of two codes: the first code functions as a transmitter and modulates data bits for transmission through an audio-band channel and the second code functions as a receiver and translates sampled waveforms into data bits.

Chapter 9 describes the Fast Fourier Transform (FFT) and its implementation on the DSP56300. In addition, we describe a basic code which is capable of real-time spectral processing.

[1]The use of DSP to mean either Digital Signal Processing or Digital Signal Processor will be clear from context.

[2]Throughout this text, the "x" in DSP5030x will be an integer from 0–9 and will denote the particular member of the DSP56300 family.

The four projects provide a good overview of the techniques used in developing real-time, DSP-based applications. The goal of the text is that with adequate exposure to the techniques described in the projects, the student is capable of taking virtually any DSP algorithm and transforming it into working code.

Chapter 1

Introduction to Real-Time DSP

1.1 RTDSP Basics

This section gives a brief history of digital signal processors and their typical applications. We also describe real-time software. We first begin with two definitions.

> ***Definition:*** *Digital Signal Processing* is an operation or transformation of a signal on a computer or other special-purpose digital hardware.

> ***Definition:*** *Real-Time Digital Signal Processing* is an operation or transformation of a signal on a computer in synchronization with events occurring in the physical system such as sampling.

1.1.1 DSP Chip History

The Intel 2920 (1979) was the first commercially-available, general-purpose single chip DSP. The device was not fully appreciated at the time, possibly because there were few engineers conversant with DSP theory and applications. We note that the text *Digital Signal Processing* by A. Oppenheim and R. Schafer was first published in 1975. Although, the 2921 appeared a year or so later, Intel soon discontinued manufacturing DSP devices. In the 1990s, Intel released the i960 microprocessor which is used in DSP applications and added multimedia/DSP capabilities (MMX technology) to the Pentium family.

Several years later (1982) other DSP chips were designed including the NEC μPD7720 and Texas Instruments TMS32010. The chips were capable of 16-bit integer arithmetic at the rate of 5 million instructions per second (MIPS) and had limited random access memory (RAM), read-only memory (ROM), and input/output (I/O) capabilities. These processors, however, were commercially successful and paved the way for future generations of DSPs.

Another milestone in DSP history occurred in 1986 when AT&T (now Lucent Technologies) introduced the first commercially available floating-point DSP, the DSP32. A floating-point DSP

frees the engineer from signal scaling issues such as under- or over-flowing, however, they are often more expensive and consume more power thereby limiting their application. In fact, most commercial products utilize fixed-point DSPs while floating point DSPs are mainly used in image-processing applications. All members of the Motorola DSP56300 family are fixed-point devices while those in the DSP96000 family are floating-point.

1.1.2 Applications

DSPs differ from ordinary general-purpose microprocessors and microcontrollers in that they are specially designed to perform DSP operations rapidly. Such operations include fast Fourier transforms (FFTs) and inner products (sums of products or multiply-accumulates)

$$\begin{aligned} y[n] &= \sum_{k=0}^{N-1} h[k]x[n-k] \\ &= \mathbf{h}^T \mathbf{x}[n] \end{aligned} \tag{1.1}$$

where $\mathbf{h} = [h[0], h[1], \ldots, h[N-1]]^T$, $\mathbf{x}[n] = [x[n], x[n-1], \ldots, x[n-N+1]]^T$, and ^{T} indicates matrix/vector transpose. It is not surprising that as microprocessors have become more multimedia-oriented, many features of DSPs have become integrated in the general microprocessor designs.

Widespread use of DSPs beginning in the early 1980s has caused a revolution in product design. Since their introduction, DSPs have been used in many applications which have fueled the information age. These include audio/video (A/V) applications such as MPEG 1 Layer 3 (MP3) players, surround sound processing, high-definition television (HDTV), and digital video disc (DVD) players; cordless telephones, cellular telephones, and pagers; and modems, FAX machines, ethernet cards, routers, and hubs. The major DSP manufacturers are Motorola (DSP56K, DSP96K), Texas Instruments (TMS320), Lucent Technologies (DSP32), and Analog Devices (ADSP2100).

This revolution has been fueled by several advantages that DSP techniques have over conventional, analog methods. These advantages include

- FLEXIBILITY: Complicated signal processing when performed on a DSP is done in software. Modifications to the signal processing is achieved by modifying *software*. These changes can be rapidly deployed in the manufacturing line by changing a few lines in a ROM or erasable programmable read-only memory (EPROM) or even delivered over the Internet. Changes in analog electronics can often be much more difficult.

- RELIABILITY: Integrated digital circuits are very reliable and can be automatically inserted in boards. Once the code is perfected, the chip function does not change with age or temperature as is sometimes the case with analog electronics.

- ECONOMICS: Low-cost DSPs have made it more economical to digitally implement signal processing applications, particularly audio-band applications like modems. DSPs have made

it possible to manufacture products with tremendous complexity and sophistication at prices ordinary consumers can afford. For fixed cost, DSP performance has doubled every two years for the past 20 years.

The DSP approach is not always the best solution for the problem even if DSP can accomplish the task. For example, a commercial AM radio signal (carrier frequency on the order of 1MHz) could be demodulated with a codec [integrated analog-to-digital converter (A/D) and digital-to-analog converter (D/A)] and a few lines of DSP code running on a sufficiently powerful DSP—a so called "software radio." However, the AM signal can also be demodulated with a simple envelope detector consisting of a diode, resistor, and capacitor. The latter solution is several orders of magnitude cheaper and simpler than the former.

1.1.3 Real-Time Software

While it is expected that the reader has some prior programming experience (not necessarily in assembly language), it is not assumed the reader has ever written event-driven or real-time software. Real-Time Digital Signal Processing is done by writing software for a specialized microprocessor called (of course) a digital signal processor. We will be programming the DSP5630x in assembly language so as to learn the architecture of the DSP. C Compilers do exist for many DSPs (including the DSP56300 family), however, most of the computationally-intensive routines in the software are written in assembly for maximum efficiency. The DSP communicates with the analog world through the codec and with a host PC through a serial interface. The codec we will be using is Crystal Semiconductor's CS4215. The processor and codec reside on a printed circuit board (PCB) called the Evaluation Module (EVM). We will be using the Motorola DSP56302EVM although virtually all of this text applies to any EVM in the Motorola DSP56300 family.

There are three tools used in developing the software (code): editor, assembler (or compiler), and debugger. The editor is used to type in code, edit, and comment. There are many available editors such as NotePad, Programmer's File Editor (PFE), vi, emacs, Microsoft Word, and WordPerfect. The assembler translates instructions to object code; we will use the Motorola DSP56300 Assembler. The debugger is used to load the object code into the DSP and initiate execution, trace execution, and examine registers/memory. The latter functions are critical in correcting mistakes (bugs) your code may have; we will use the Domain Technologies Debug-EVM. It may come as no surprise that most of our time will be spent using the debugger.

Pseudo-code for a typical real-time DSP application might look like the following.

```
initialize_DSP

initialize_codec

initialize_memory

main

    wait_for_input_samples

    get_input_samples

    process_samples

    put_output_samples

    goto main
```

Clearly, real-time DSP applications are limited to cases where the required sampling rate is sufficiently less than the DSP's instruction rate so a reasonable number of instructions can be performed between sample periods. Unlike most code you have probably written, real-time code *must* be executed within a certain period of time (usually the sampling period, T) to maintain synchronization. If the code does not execute quickly enough, there is the risk of not properly processing samples. Sometimes output samples may not be passed to the codec and will be "dropped." Since the sample period for audio-band applications is relatively large, DSPs are used extensively in audio applications since many instructions can be performed between samples. These applications include telephony and home theater applications. The following two examples will determine bounds on the amount of instructions per sample that can be performed real-time using common audio-band sampling rates.

Example: Assume that the Motorola DSP5630x is clocked at 66MHz and can perform 66MIPs (note that the instruction rate can be higher if several instructions are performed in parallel). Furthermore, assume a sampling rate of 48kHz which is a standard audio rate used in digital audio tape (DAT). Simple division dictates that we can perform $(66 \times 10^6)/(48 \times 10^3) = 1375$ instructions per sample. This will be the lower bound on the number of instructions we can theoretically execute in a sample period with the highest sample rate available on the DSP5630xEVM. ■

Example: Now, assume a sampling rate of 8kHz which is typical of voice-band and telephony applications. Again simple division dictates that we can perform $(66 \times 10^6)/(8 \times 10^3) = 8250$ instructions per sample. This will be the upper bound on the number of instructions we can theoretically execute in a sample period with the lowest sample rate available on the DSP5630xEVM. ■

Thus depending on the sample rate, the number of instructions which can be executed in a sample period on the DSP5630x is bounded by

$$1375 < \#\text{instructions}/T < 8250. \tag{1.2}$$

You will therefore be challenged not only to produce working code but code that is sufficiently compact (tight) enough to execute real-time.

Chapter 2

Introduction to the Motorola DSP56300 Family

2.1 Internal Architecture and Programming Model

There are two general types of architectures for microprocessors: Harvard and Princeton. In the Harvard architecture (Figure 2.1) there are separate memories and busses for program and data while in the Princeton architecture (Figure 2.2) there is a single memory and bus.

All DSP5630x devices are based on the DSP56300 core (see block diagrams on pp. 14 and 15) which is described in Chapter 1 of the *DSP56300 Family Manual*. The individual family members are designed with different peripherals and memory configurations so as to better suit the target application. The DSP56300 core is based on a modified Harvard architecture which includes an additional data memory and associated data bus. This architecture incorporates the important feature of duality which facilitates many DSP applications involving complex-valued numbers consisting of real and imaginary components and audio applications which require left and right channel processing. The parallelism of the bus structure permits fast and efficient data throughput. Furthermore

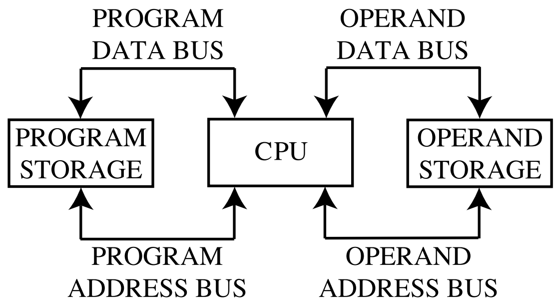


Figure 2.1: Harvard Architecture.

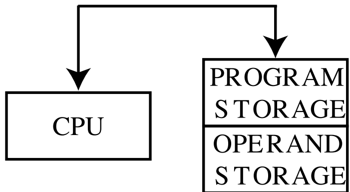


Figure 2.2: Princeton Architecture.

the DSP56300 core is capable of executing an instruction every clock cycle. The major components of the DSP5630*x* core are:

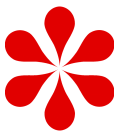
- Three independent execution units
  - Data Arithmetic Logic Unit (ALU)
  - Address Generation Unit (AGU)
  - Program Control Unit (PCU)
- Four independent 24-bit data buses
  - X Data Bus (XDB)
  - Y Data Bus (YDB)
  - Program Data Bus (PDB)
  - Global Data Bus (GDB)
- Three independent 24-bit address buses (capable of accessing 16M of memory)
  - X address bus (XAB)
  - Y address bus (YAB)
  - Program address bus (PAB)
- Highly parallel instruction set with unique DSP addressing modes

These components will be discussed in further detail in the next sections.

2.2 DSP56300 Architecture Introduction

This section is a reproduction of Chapter 1 of the *DSP563xx/6xx Family Digital Signal Processor Training Notes*[1]. The DSP56300 architecture will also be discussed in more detail in Section 2.7.

[1]Copyright of Motorola, Used with Permission.



DSP56300 Architecture Intro

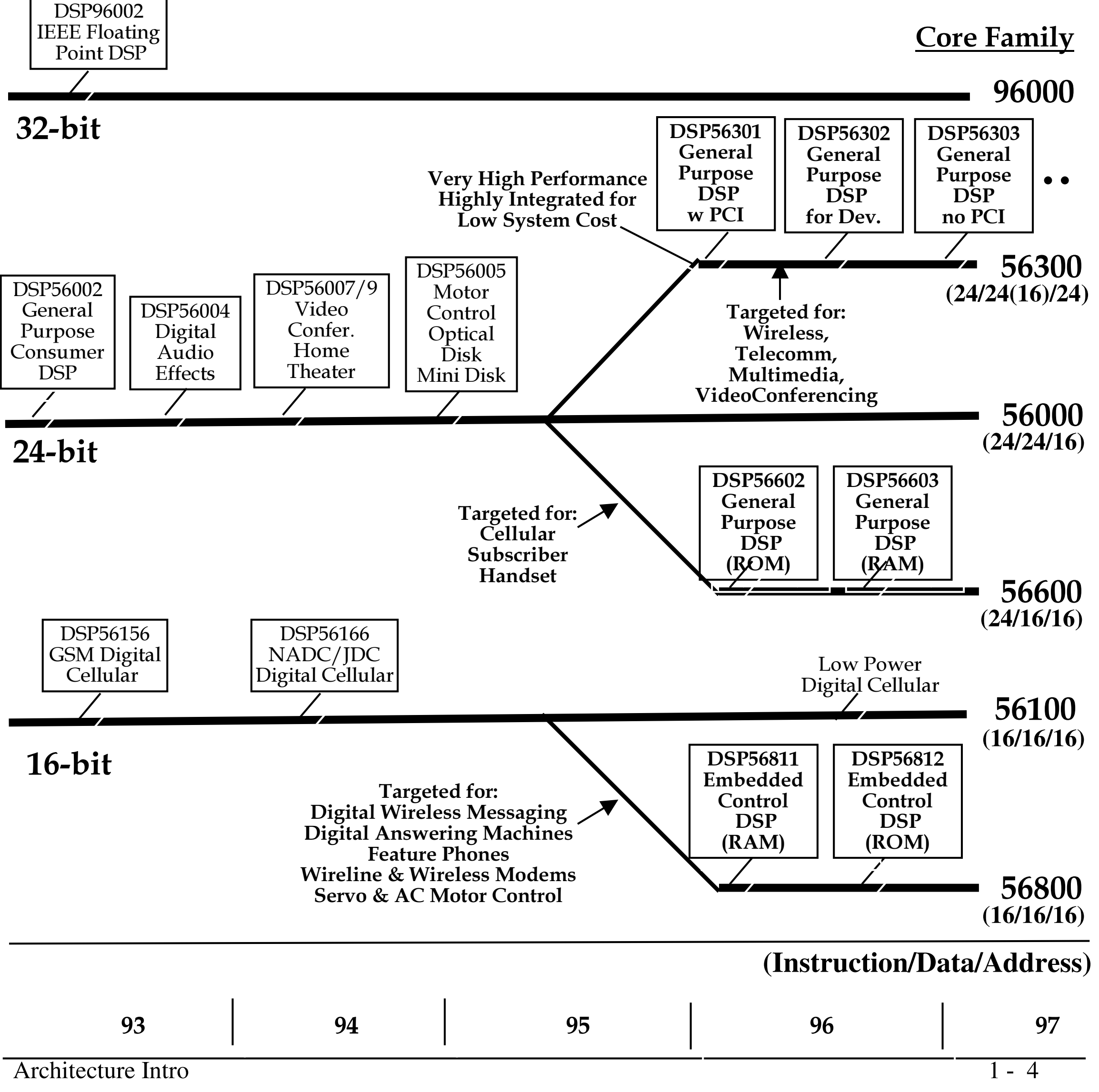
Learn how to:

- Differentiate among DSP56300 family devices
- Use your architectural knowledge to maximize concurrency for execution speed
- Convergently round numbers
- Use the Status Register to:
  - Scale data
  - Detect data growth
  - Normalize numbers
  - Detect out of range numbers
  - Adjust global interrupt mask priority

Table of Contents

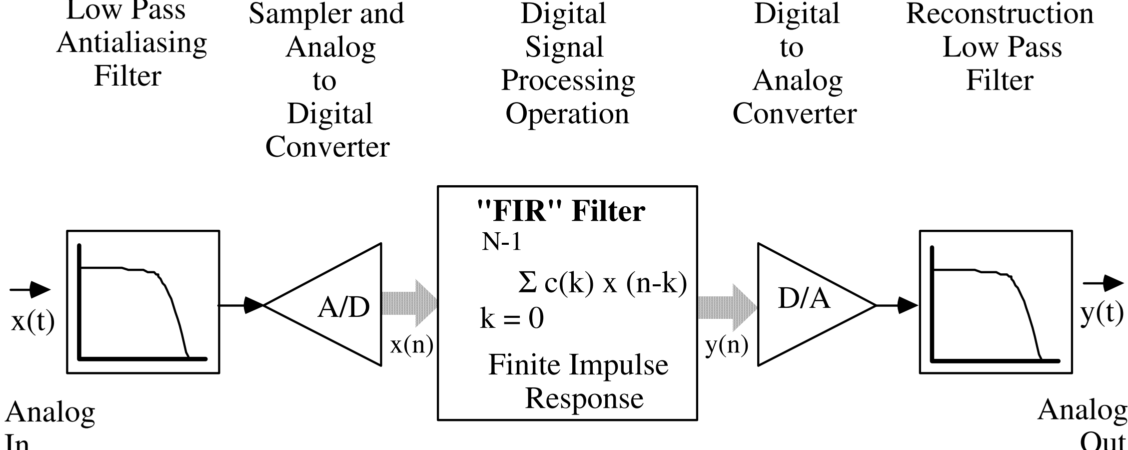
4....DSP Families
5....Basic DSP Operations
6....DSP56301 Block Diagram
7....DSP563xx Derivatives
8....DSP56301 Memory Map
9....FIR Filter Algorithm
10..Operand and Memory Types
11..Word Operand (24 bits)
12..XY Memory
13..Longword Operand (48 bits)
14..Accumulator Operand (56 bits)
15..Programming Model
16..Program Controller
17..Hardware Stack
18..Status & Loop Registers
19..No Overhead Looping
20..Data ALU
21..Data ALU Registers
22..Data Transfers
23..Data Transfer Examples
24..Fractional vs Integer
25..Saturation Arithmetic Limiting
26..X & Y Limiting
27..Convergent Rounding
28..Condition codes Z, N, V & C
29..Extension & Unnormalized bits
30..Scaling bits
31..Scaling Effects
32..Architecture Exercises (1 of 2)
33..Architecture Exercises (2 of 2)

DSP Families



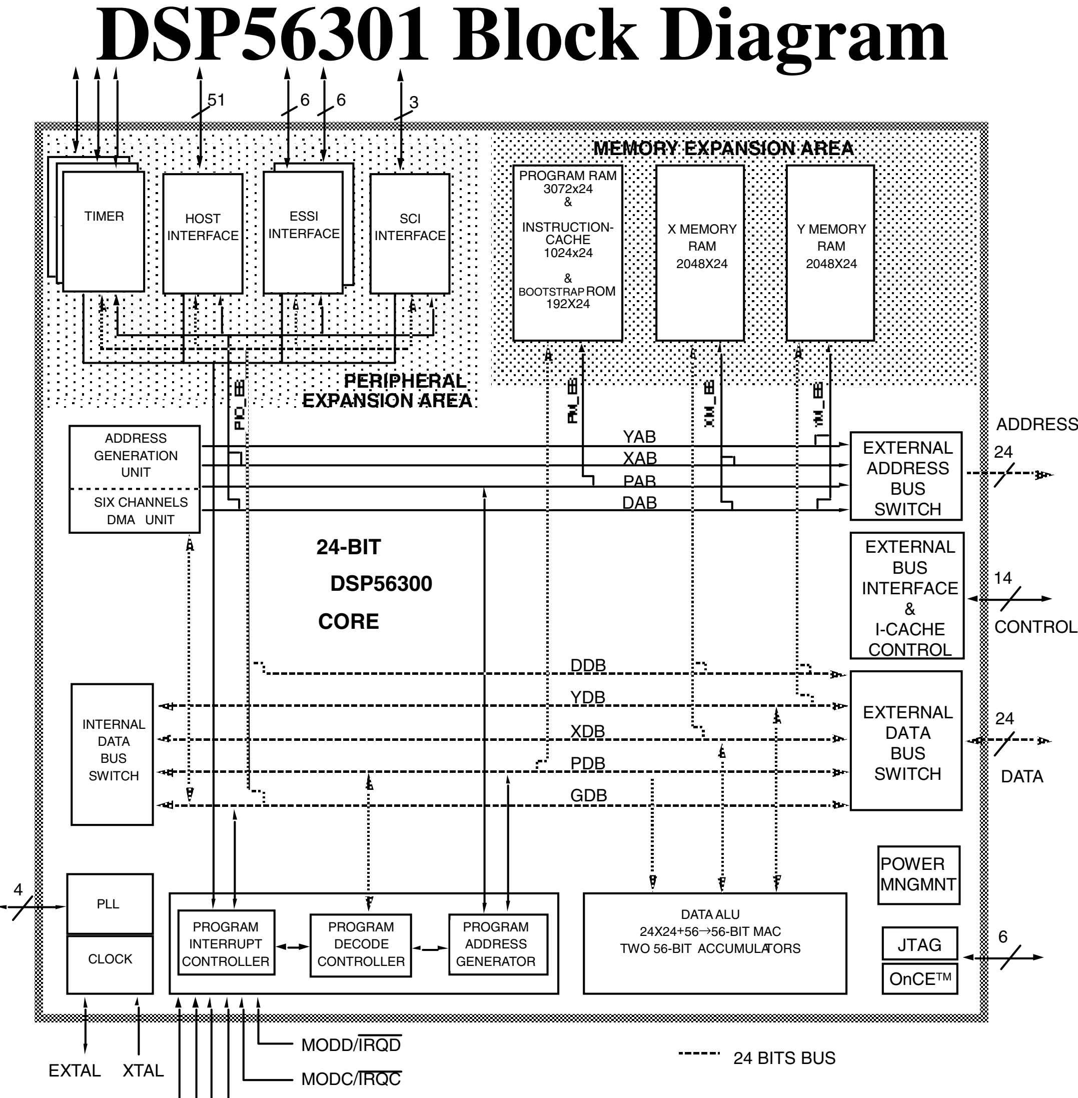
Basic DSP Operations

Digital Filter



- **DSP Applications need very high execution speed in a real-time I/O intensive environment.**
- **Dual-natured**
- **Sophisticated I/O**
- **MAC H/W**
- **No overhead loops**
- **Special DSP addressing**

DSP56301 Block Diagram



- UDR2 0.5u geometry
- 66/80/100 MHz clock frequency
- 208 pin low-cost TQFP package
- Functional down to 1.8 V power supply
- Peripherals and interrupt pins are 5-volt-tolerant
- WAIT and STOP modes
- Full-CMOS/fully-static design - 66/80 MHz down to DC
- Intelligent Power Management:
  Automatically powers down unused memory modules, peripherals and core logic on a per clock basis

- 1.7 million transistors
- 3.3 V power supply
- Low-power dissipation:
  less than 1.4mA/MIPS @ 3.3V (66/80 MIPS)
  less than 1.1mA/MIPS @ 2.7V (55/66 MIPS)
  less than 0.75mA/MIPS @ 1.8V (27/33 MIPS)

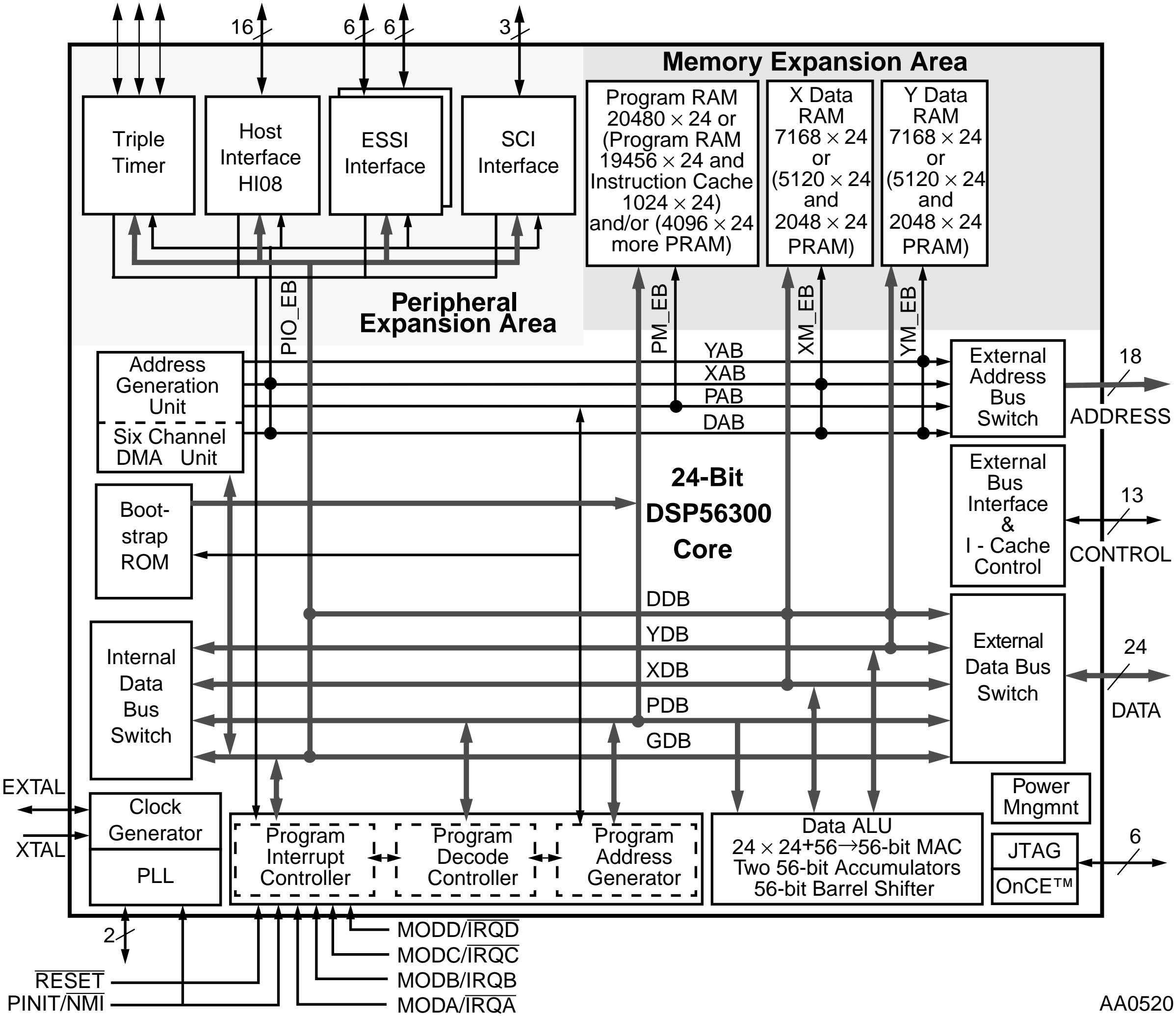
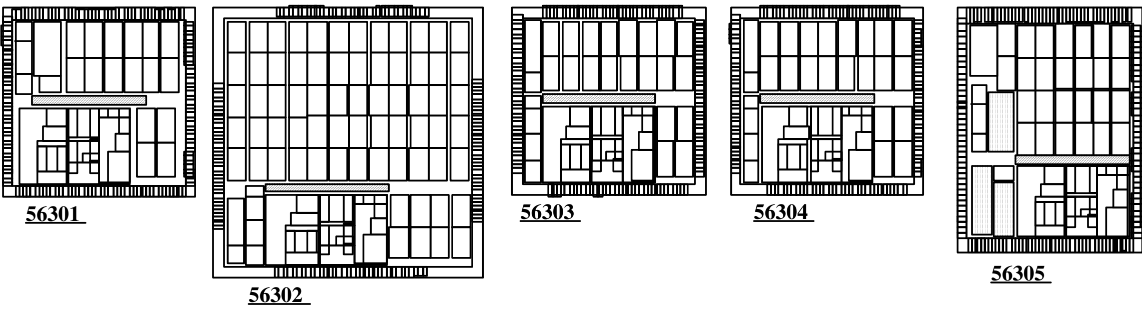


Figure 2.3: DSP56302 Block Diagram from *DSP56302 User's Manual.*

DSP563xx Derivatives



	56301	56302	56303	56304	56305
Package	208 TQFP 252 PBGA	144 TQFP	144 TQFP 196PBGA	144 TQFP 196PBGA	208 TQFP 252PBGA
Total memory	8K	34K	8K	57K	21.25K
PRAM	4K (2K)	20K (24K)	4K (2K)	1K (3.5K)	6.5K (7.5K)
PROM	-	-	-	33K	6K
DRAM	2K + 2K (3K+3K)	7K + 7K (5K+5K)	2K + 2K (3K+3K)	2.5K + 2.5K (1.25K+1.25K)	3.75K X + 2K Y (2.75K) X
DROM	-	-	-	9K + 9K	3K Y
Host interface	HI32	HI08	HI08	HI08	HI32
Serials	2 ESSI + 1 SCI	2 ESSI + 1 SCI	2 ESSI + 1 SCI	2 ESSI + 1 SCI	2 ESSI + 1 SCI
Timers	3	3	3	3	3
HW Accelerators	-	-	-	-	VCOP CCOP FCOP

DRAM = Data RAM
PRAM = Program RAM
DROM = Data ROM
PROM = Program ROM

DSP56301 Memory Map

PROGRAM

Address	Region
$FFFFFF	INTERNAL - RESERVED
$FF00C0	BOOTSTRAP ROM (192 words)
$FF0000	EXTERNAL
$000FFF	1K INTERNAL I_Cache/PRAM
$000BFF	3K INTERNAL PRAM
$000000	

X DATA

Address	Region
$FFFFFF	INTERNAL I/O (128 words)
$FFFF80	EXTERNAL
$FFF000	INTERNAL - RESERVED
$FF0000	EXTERNAL
$0007FF	2K INTERNAL RAM
$000000	

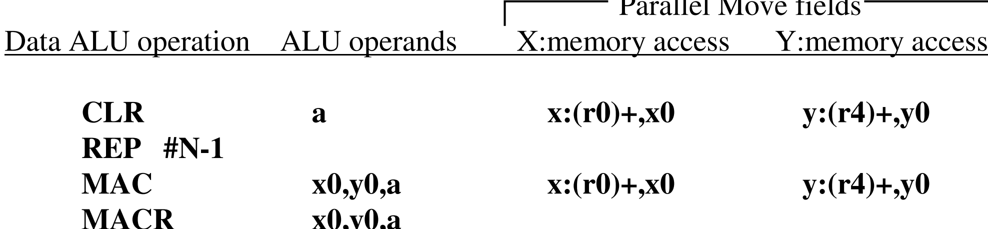
Y DATA

Address	Region
$FFFFFF	EXTERNAL I/O (128 words)
$FFFF80	EXTERNAL
$FFF000	INTERNAL - RESERVED
$FF0000	EXTERNAL
$0007FF	2K INTERNAL RAM
$000000	

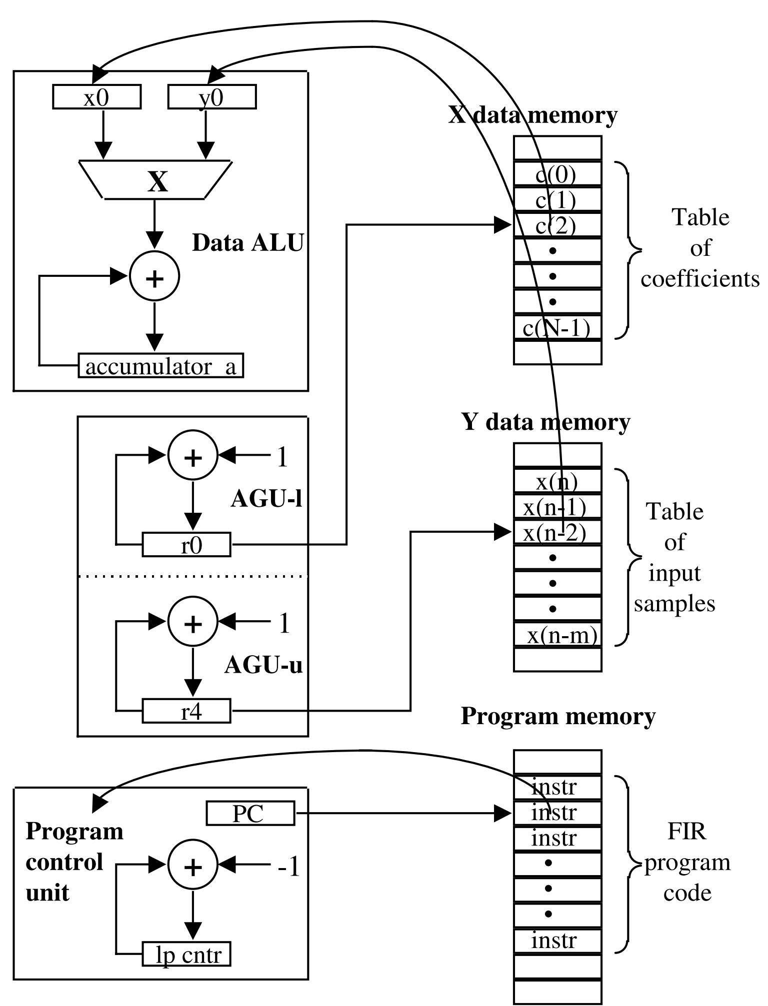
- Three independent concurrently accessed 16M word memory spaces:
  - Program (P:)
  - X data (X:)
  - Y data (Y:)
- On-chip peripherals are memory mapped in top 128 words of X data memory
- Off-chip peripherals are memory mapped in top 128 words of Y data memory
- 2K words of on-chip X data RAM (8 banks of 256 words each)
- 2K words of on-chip Y data RAM (8 banks of 256 words each)
- 3K words of on-chip program RAM and 1K words Instruction cache (CE=1), or 4K words of on-chip program RAM and no Instruction cache (CE=0) organized as 12 or 16 banks of 256 words each.
- Glueless interface for off-chip memory expansion

FIR Filter Algorithm

Main instructions:

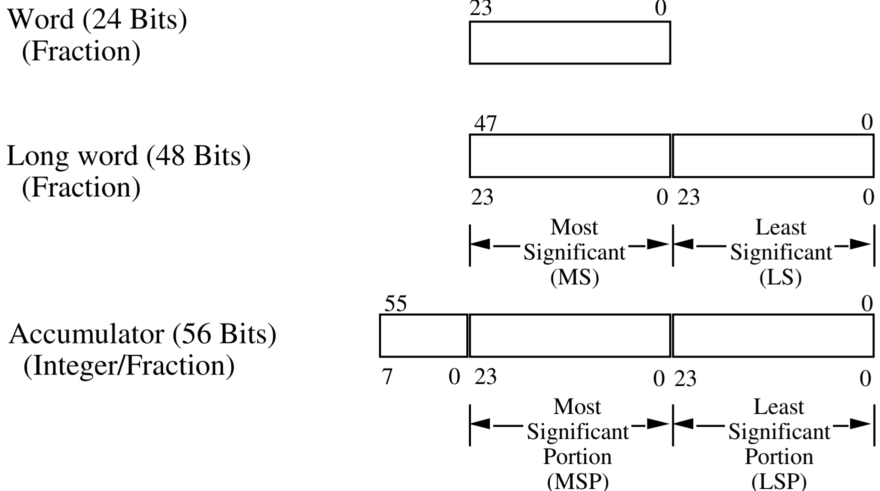


Data ALU operation	ALU operands	Parallel Move fields: X:memory access	Parallel Move fields: Y:memory access
CLR	**a**	**x:(r0)+,x0**	**y:(r4)+,y0**
REP #N-1			
MAC	**x0,y0,a**	**x:(r0)+,x0**	**y:(r4)+,y0**
MACR	**x0,y0,a**		

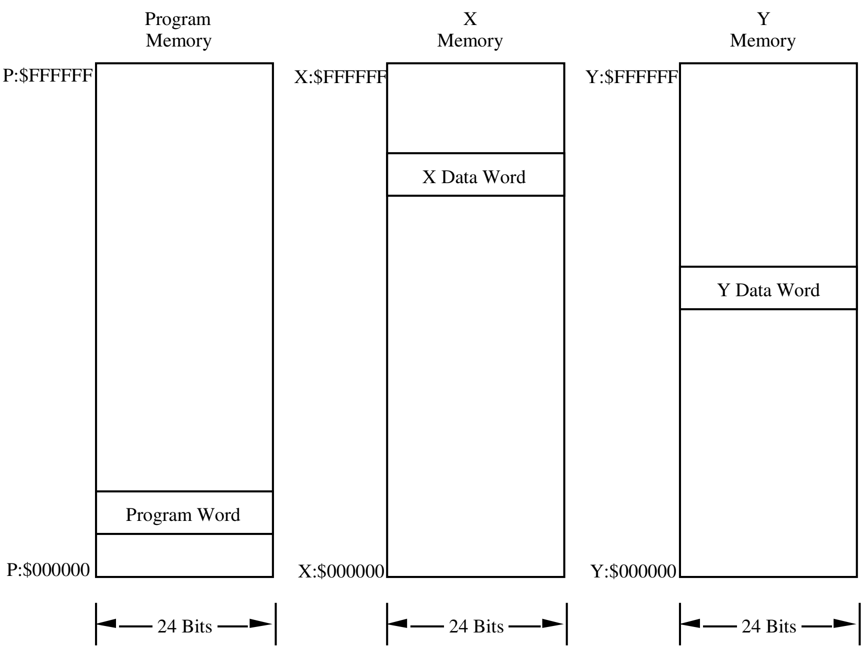


Operand and Memory Types

Operand Types:

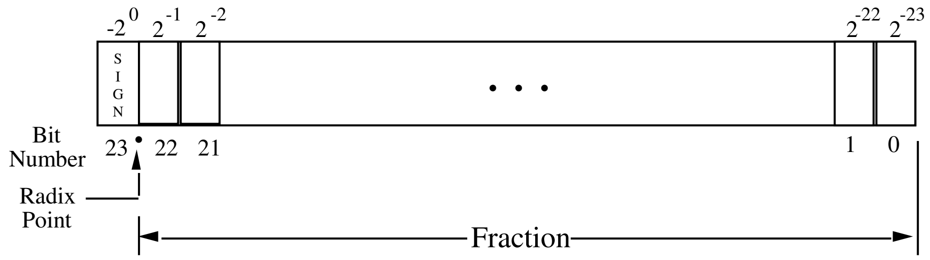


Memory Types:



Three 16 M x 24 Bit Memory Spaces

Word Operand (24 bits)



Largest Value	Smallest Value
$1 - 2^{-23}$ = \$7FFFFF	-1 = \$800000

Dynamic Range

$144\text{dB} = 20\log_{10}(2^{24})$

Storage Locations

Program memory and X or Y data memory
X1, X0, Y1, Y0 Input Registers
A1, A0, B1, B0 Accumulators

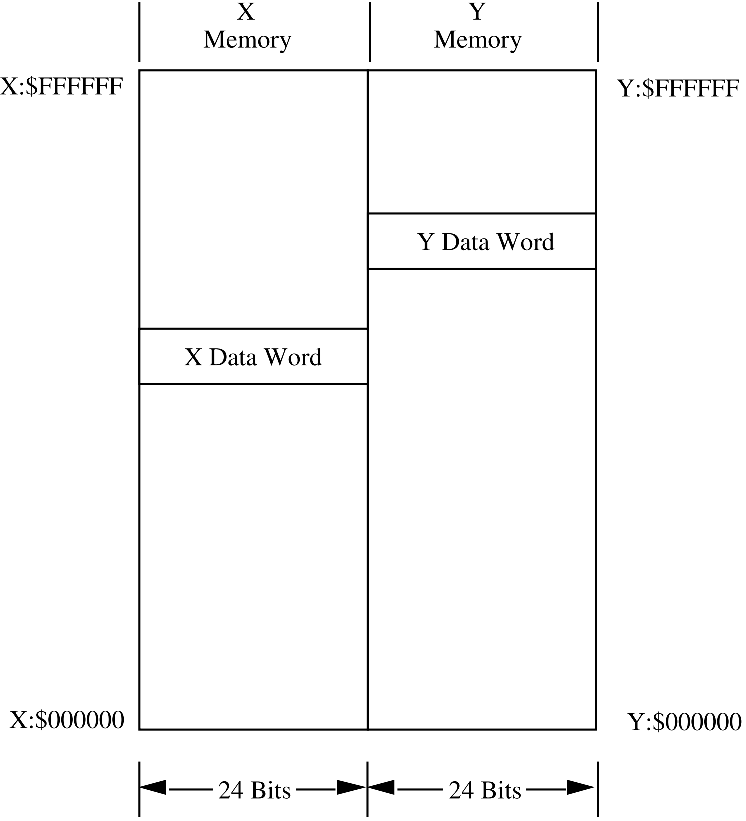
- the 24-bit fractional number representation has the same degree of accuracy as the 32-bit IEEE single precision floating point representation.

Fractional examples:

Decimal Fraction	24-Bit Binary Value	Hexadecimal Coefficient
0.750	0.1100000 00000000 00000000	$600000
0.500	0.1000000 00000000 00000000	$400000
0.250	0.0100000 00000000 00000000	$200000
0.125	0.0010000 00000000 00000000	$100000
-0.125	1.1110000 00000000 00000000	$F00000
-0.750	1.0100000 00000000 00000000	$A00000

Decimal Fraction	Hexadecimal Coefficient
0.00784313678741455	$010101
-0.00784313678741455	$FEFEFF

XY Memory

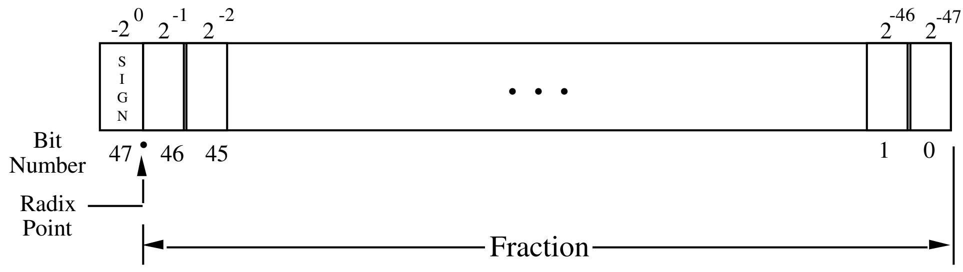


- Two independent addresses are used to access two word operands concurrently.
  - one word operand is in X memory space.
  - the other word operand is in Y memory space.

- Each effective address provides independent read/write control for its memory space. Data may be read from memory to a register or from a register to memory.

Longword Operand (48 bits)

Long Operand:

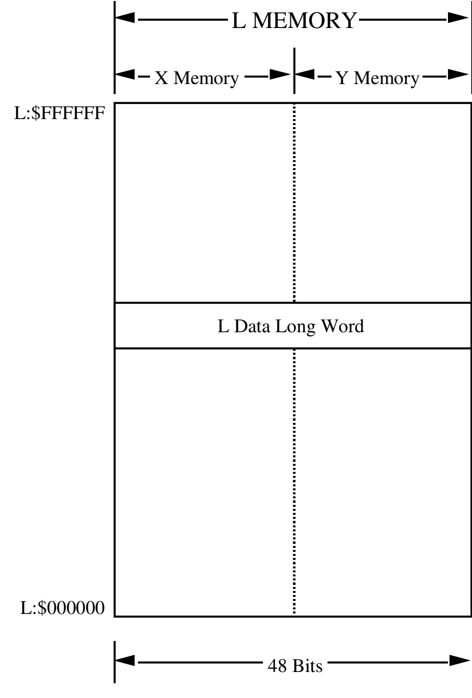


Largest Value	Smallest Value
$1-2^{-47}$ = $7FFFFFFFFFFF	-1 = $800000000000

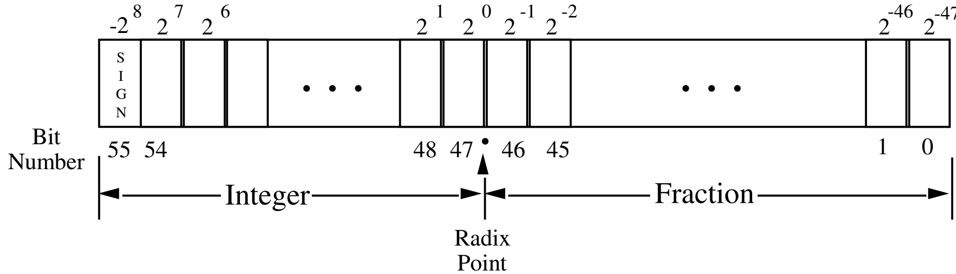
Dynamic Range: 289dB = $20\log_{10}(2^{48})$

Storage Locations: L Memory, Registers A10 and B10
Input registers X and Y

L Memory:



Accumulator Operand (56 bits)



Largest Value	Smallest Value
$256-2^{-47}$ = $7FFFFFFFFFFFFF	-256 = $80000000000000

Dynamic Range: 336dB = $20\log_{10}(2^{56})$

Storage Locations: Accumulators A & B

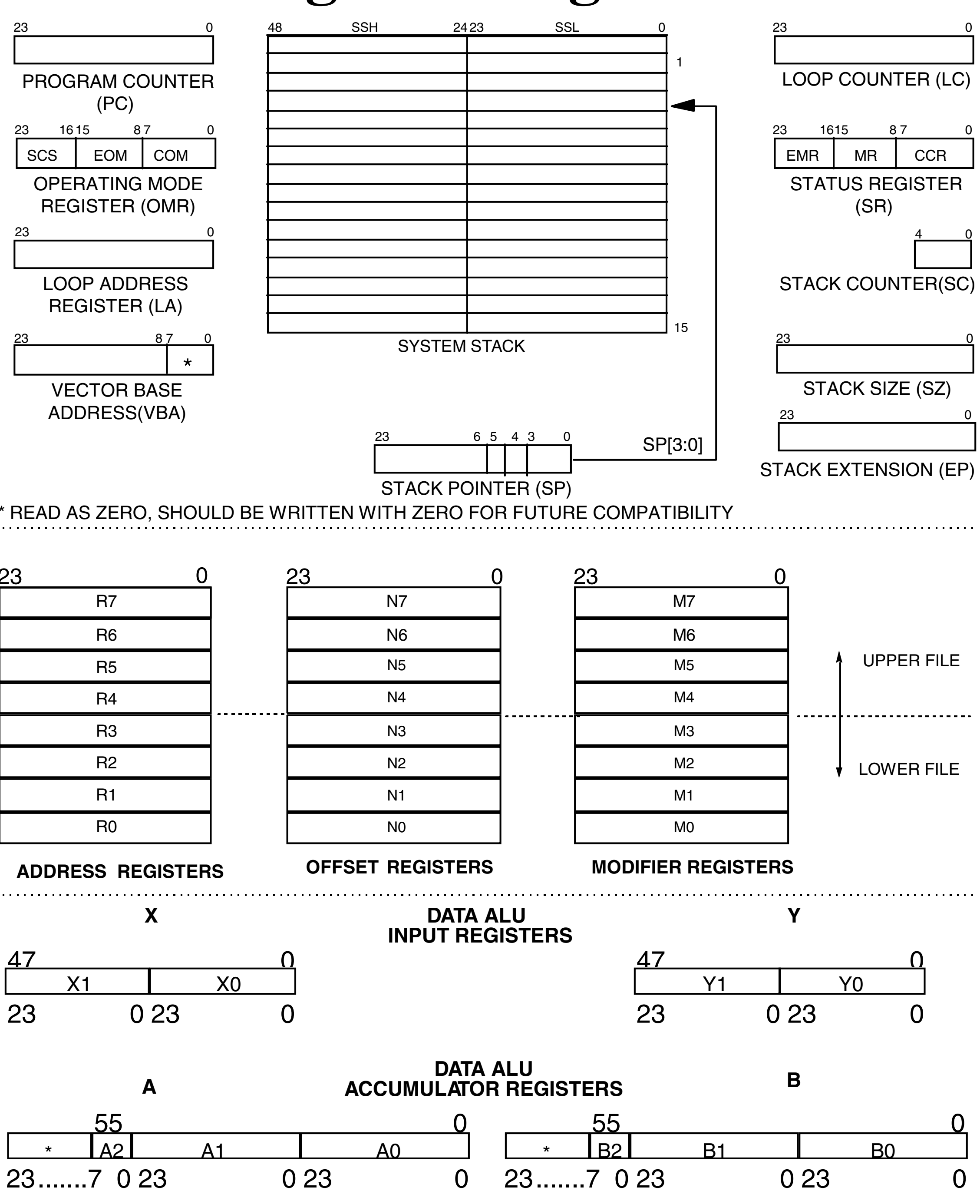
Accumulator Operand Examples:

Decimal Fraction	56-Bit Binary Value	Hexadecimal Coefficient
12.5	000001100.1000000 • • • 0000	$06400000000000
1.5	000000001.1000000 • • • 0000	$00C00000000000
0.5	000000000.1000000 • • • 0000	$00400000000000
-0.5	111111111.1000000 • • • 0000	$FFC00000000000
-1.5	111111110.1000000 • • • 0000	$FF400000000000
-12.5	111110011.1000000 • • • 0000	$F9C00000000000

Decimal Fraction	Hexadecimal Coefficient
2.007843137254902	$01010101010101
-30.117647058823540	$F0F0F0F0F0F0F0

NOTE: **RADIX Point Does not align on a hex Boundary**

Programming Model



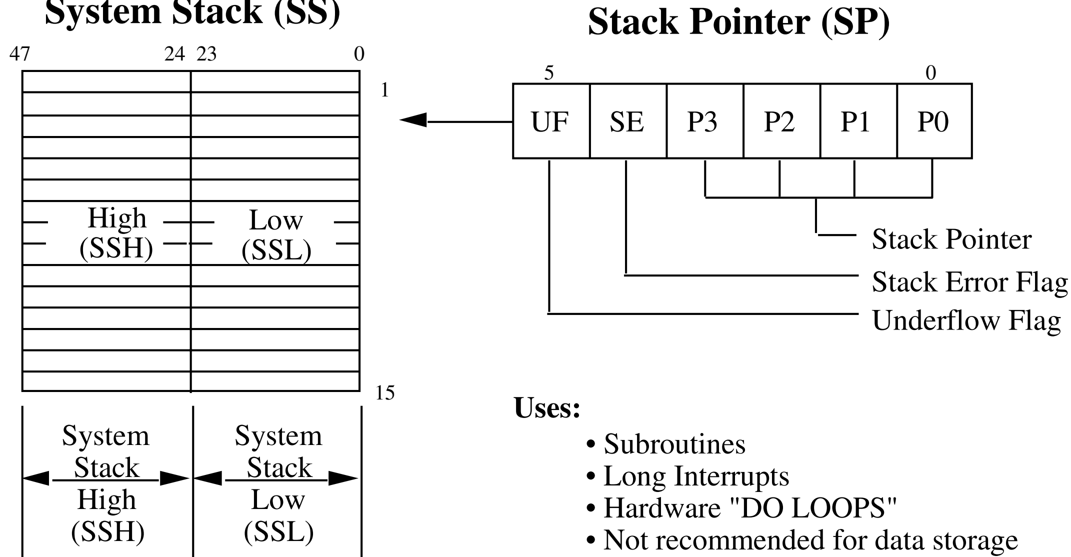
Program Controller

Address
Data
24
24
Clock
Interrupts
Control
Program Counter
Loop Address
Loop Count
Stack Pointer
Status Register
OMR
48 X 15
Hardware
Stack
24
Global Data Bus

Program Space
Address Data
X Data Space
Address Data
Y Data Space
Address Data
MUX
Data ALU
X0
X1
Y0
Y1
Address
Generation
Unit
TRIPLE MUX
Clock
Interrupts
Control
Program
Controller
N0 M0 R0 R4 M4 N4
N1 M1 R1 R5 M5 N5
N2 M2 R2 R6 M6 N6
N3 M3 R3 R7 M7 N7
Address
ALU
Address
ALU
MAC
Multiplier,
Accumulator,
Rounding,
& Data ALU
Shifter
A (56)
B (56)
Shifter/Limiter
PC
LA
LC
SP
SR
OMR
48 X 15
Stack
Global Data Bus

CORE PROCESSOR

Hardware Stack

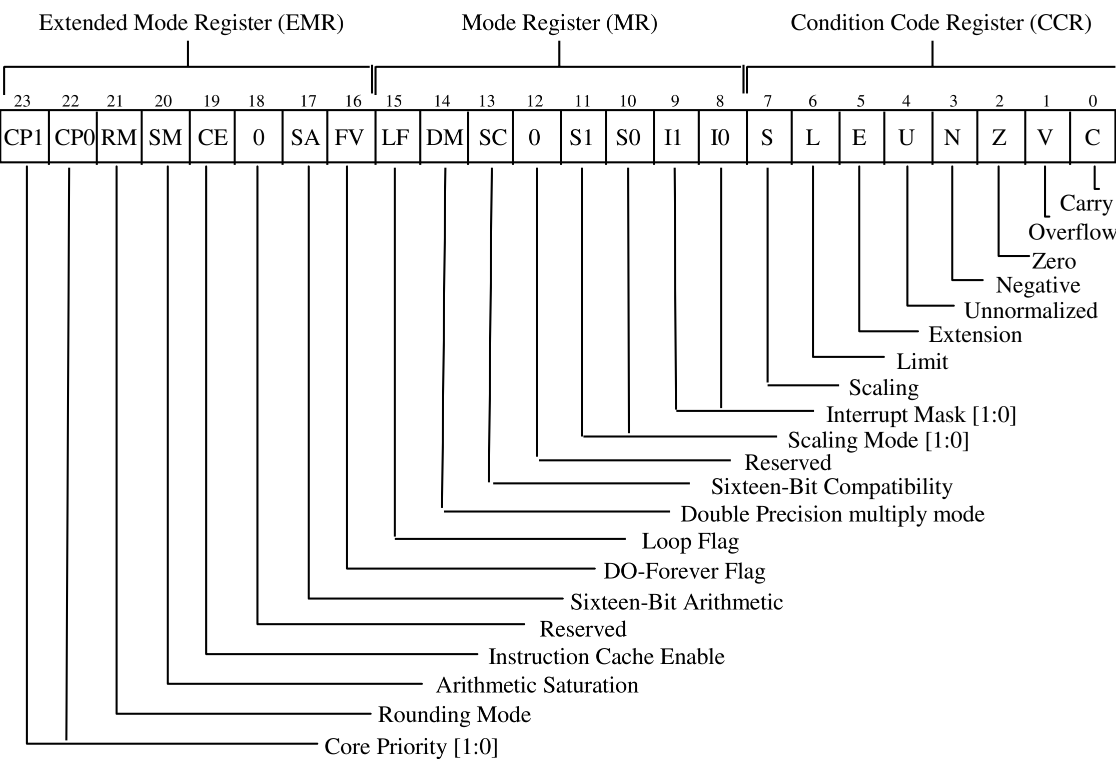


Stack Pointer Values:

UF	SE	P3	P2	P1	P0		
1	1	1	1	1	0	← Stack Underflow condition after double pull	Stack Error Exception
1	1	1	1	1	1	← Stack Underflow condition	
0	0	0	0	0	0	← Stack Empty (reset). Pull causes underflow.	
0	0	0	0	0	1	← Stack location 1	
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	
0	0	1	1	1	0	← Stack location 14.	
0	0	1	1	1	1	← Stack location 15. Push causes overflow	
0	1	0	0	0	0	← Stack Overflow condition	Stack Error Exception
0	1	0	0	0	1	← Stack Overflow condition after double push	

Status & Loop Registers

Status Register:



Loop Registers:

- Used by Hardware DO-Loops.

Loop Address
Where to end the Loop

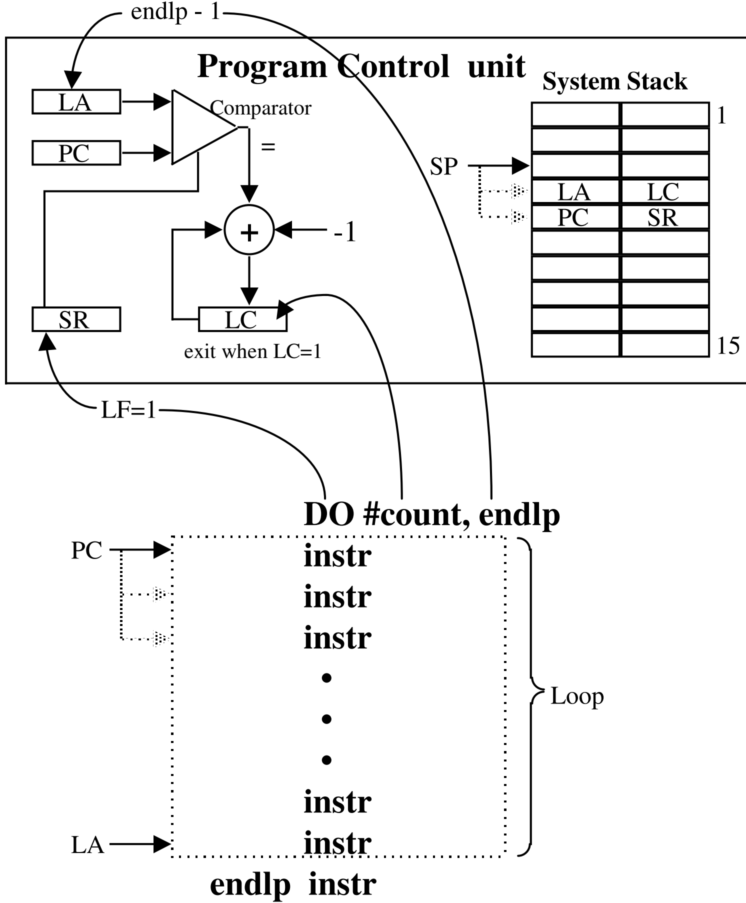
23 0

Loop Count
Number of Iteration

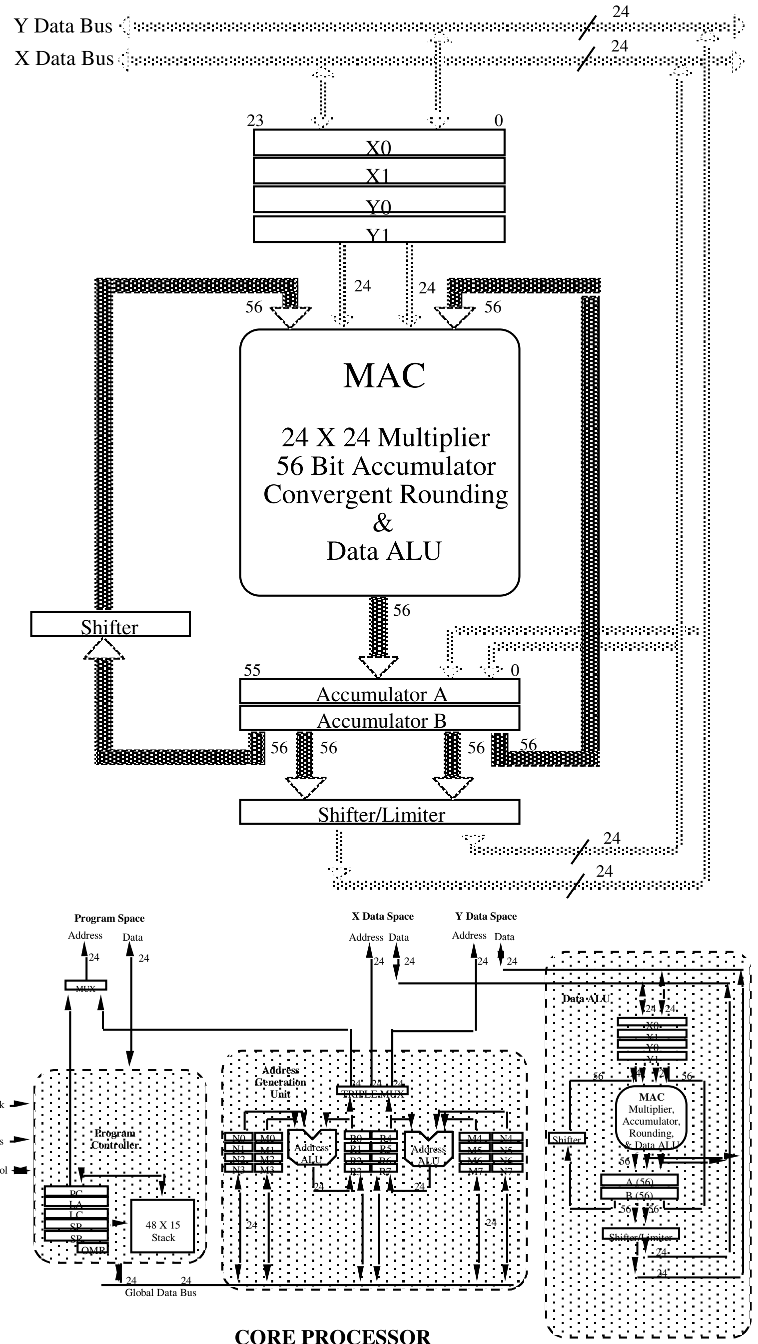
23 0

- the REPeat Instruction only uses the Loop Count Register

No Overhead Looping

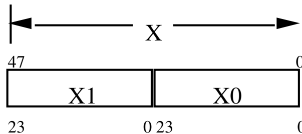


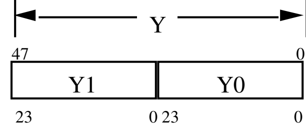
Data ALU



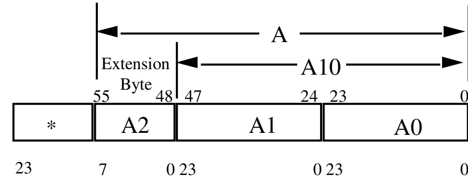
Data ALU Registers

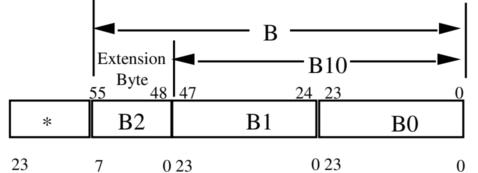
Data ALU Input Registers





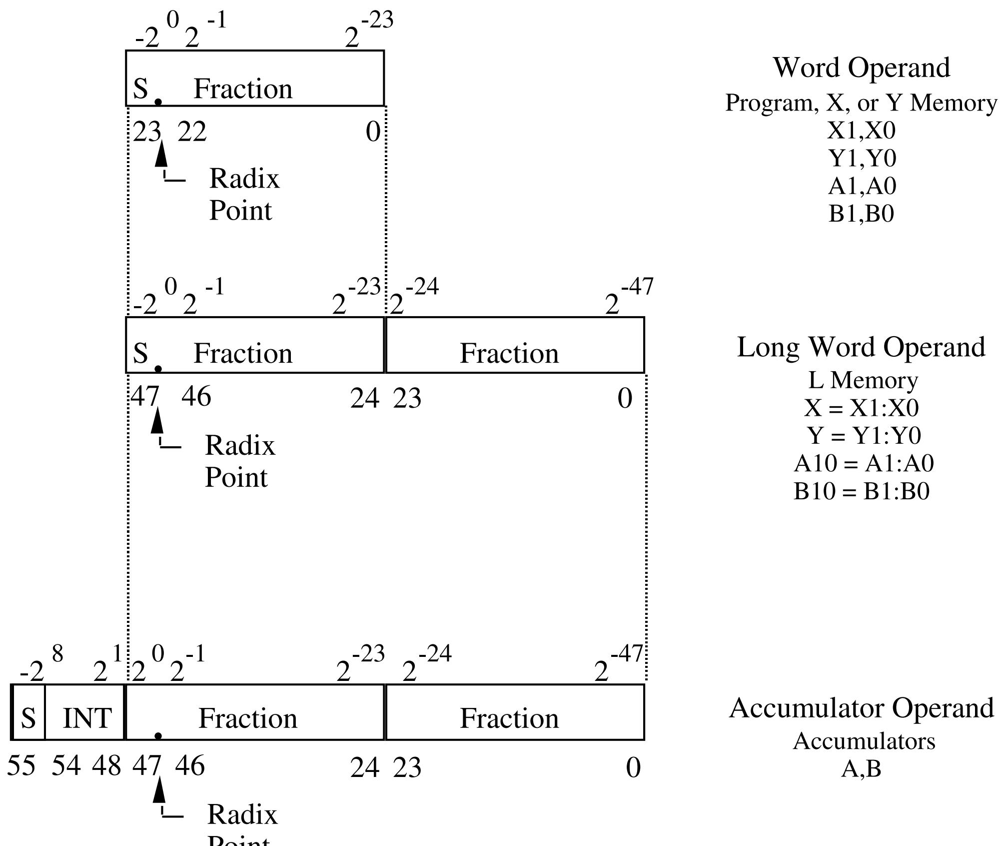
Data ALU Accumulators





* Read as sign extension bits, written as don't care.

Data Transfers

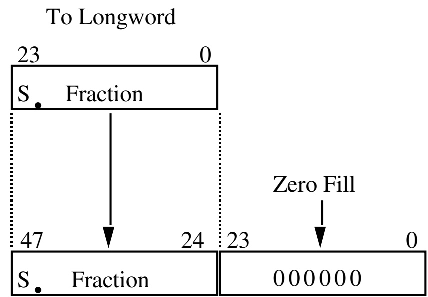


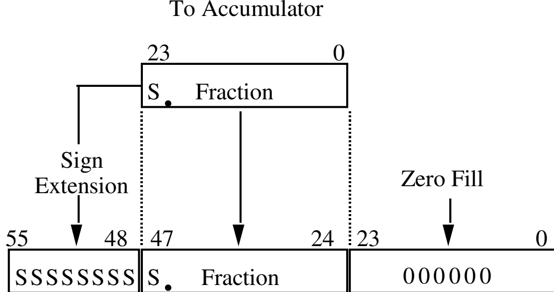
- Because numbers are fractional, numbers are left justified rather than right justified.

- To maintain alignment of the binary point when a word operand is written to a longword location or accumulator register the operand is written to the most significant 24 bits of the destination (A1, B1, X1 or Y1). The most significant bit is automatically sign extended through an accumulator extension register (A2 or B2). The least significant word of the destination is automatically cleared.

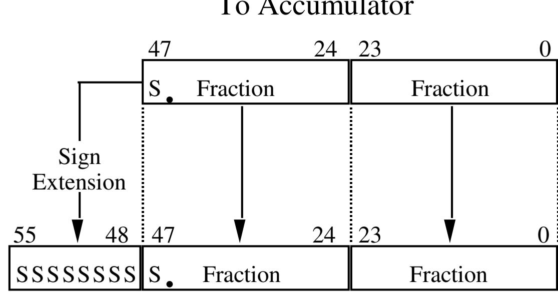
Data Transfer Examples

WORD Transfers:



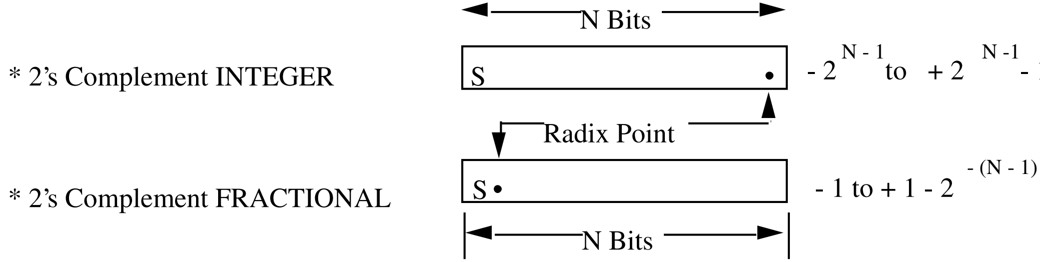


LONG WORD Transfers:

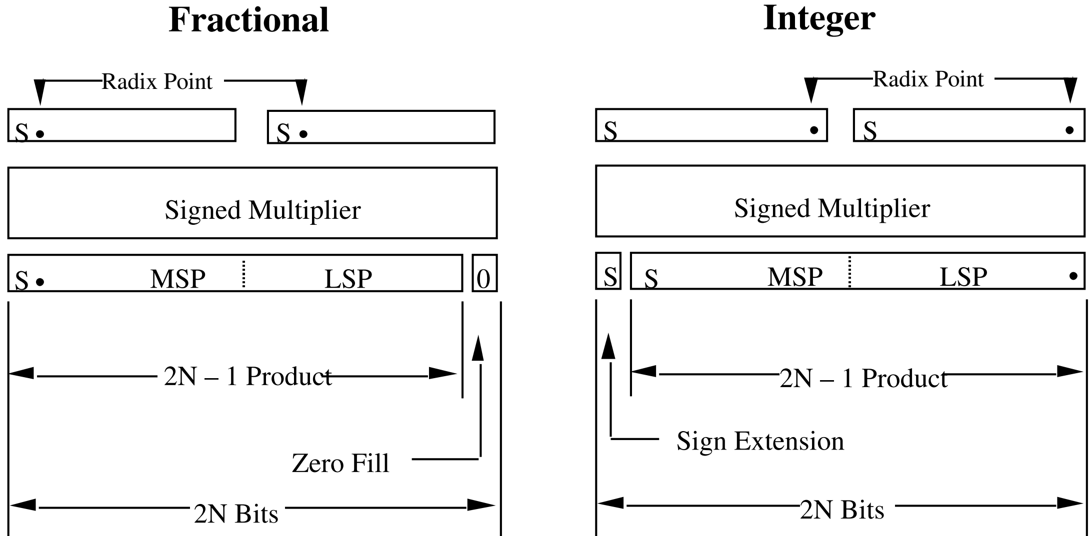


- eg. moving a 48-bit fractional value from L memory to an accumulator

Fractional vs Integer



Fractional vs. Integer Multiplication:

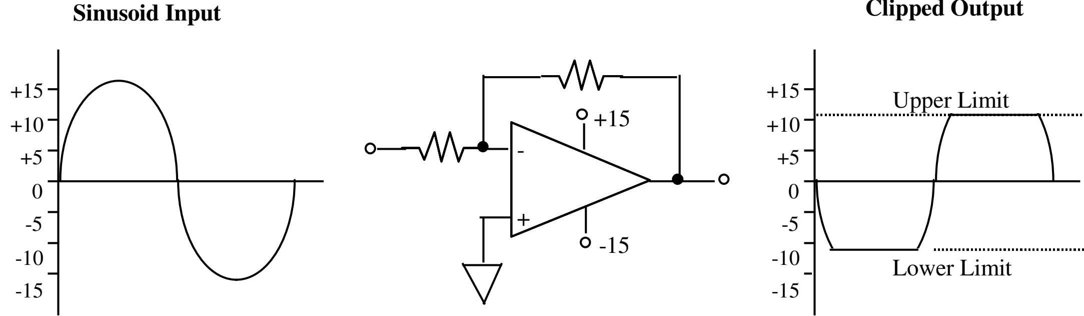


NOTE: Fractional numbers tend to be self-normalizing.

Saturation Arithmetic

Limiting

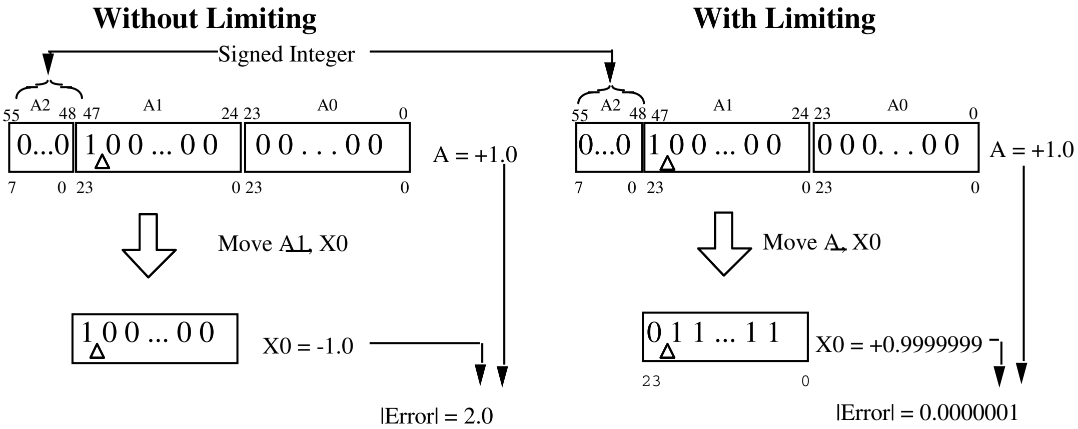
HITTING THE RAIL (Analog:)



The OP-AMP has been driven into

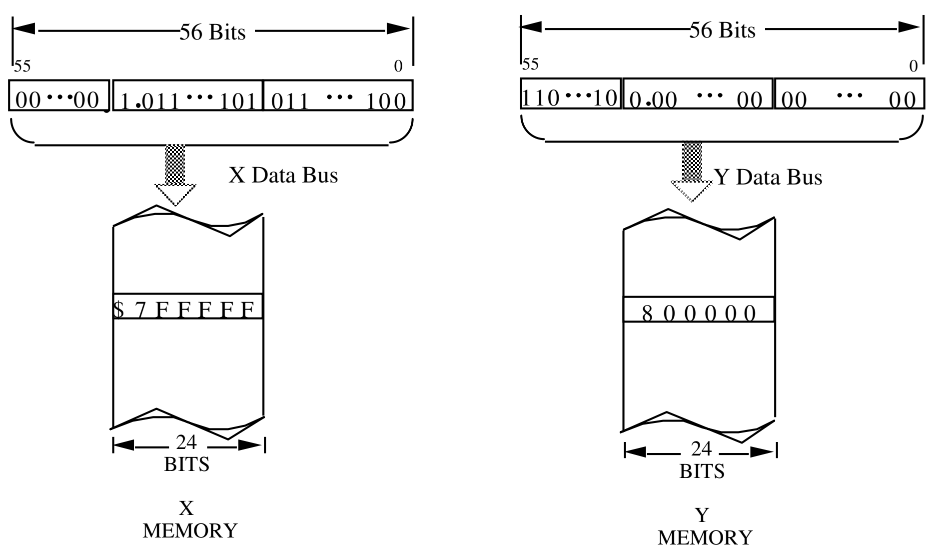
"SATURATION"

HITTING THE RAIL (Digital:)



- Minimizes overflow errors when reading Accumulator Data.
- Limiting occurs automatically when transferring from Accumulator A or B into 24 or 48 bit destination.
- Latched in Condition Code Register.
- Contents of A or B are not changed.

X & Y Limiting



Limiting Cases:

Destination Memory Reference	Source Operand	Accumulator Sign	Limited Value (Hex)		Type of access
			X Data Bus	Y Data Bus	
X	X:A X:B	+ -	7FFFFF 800000	---- ----	one 24 bit word
Y	Y:A Y:B	+ -	---- ----	7FFFFF 800000	one 24 bit word
X and Y	X:A Y:A X:A Y:B X:B Y:A X:B Y:B L:AB L:BA	+ -	7FFFFF 800000	7FFFFF 800000	two 24 bit words
L(X:Y)	L:A L:B	+ -	7FFFFF 800000	FFFFFF 000000	one 48 bit long word

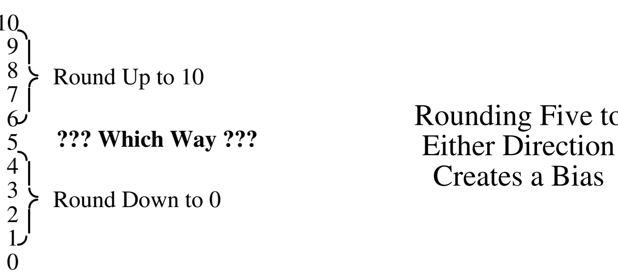
If Limiting or Overflow occurs then
Limit Bit is set and latched.
Must be cleared by software.

Condition Code Register

S	1	E	U	N	Z	V	C

Convergent Rounding

Rounding: The Problem:

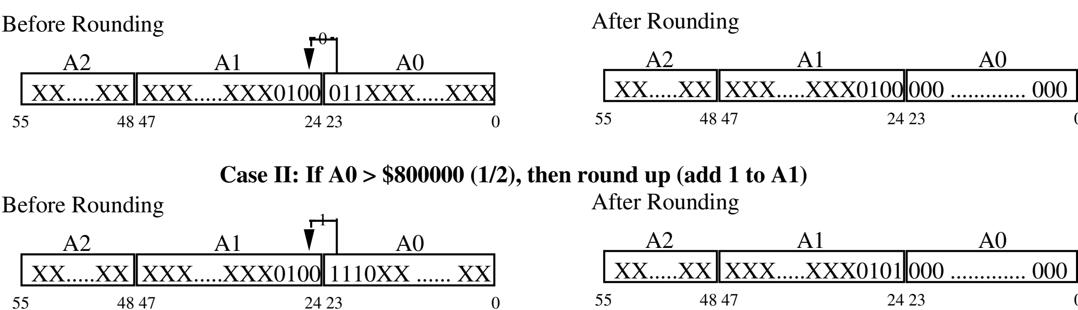


Rounding Five to
Either Direction
Creates a Bias

The Solution: Convergent Rounding or Round to Nearest Even.

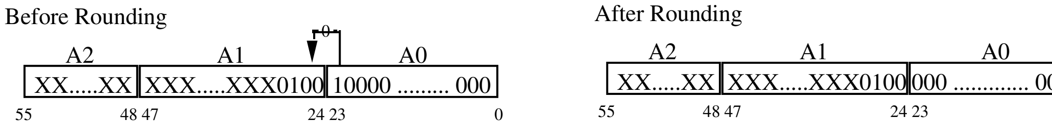
Convergent Rounding Examples:

Case I: If A0 < $800000 (1/2), then round down (add nothing)

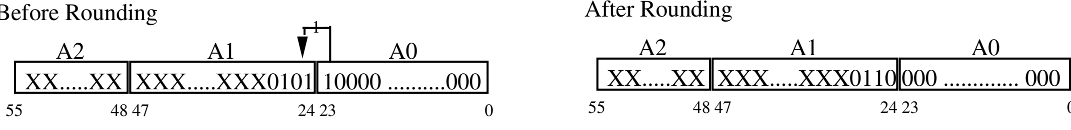


Case II: If A0 > $800000 (1/2), then round up (add 1 to A1)

Case III: If A0 = $800000 (1/2), and the least significant bit of A1 = 0, then round down (add nothing)



Case IV: If A0 = $8000000 (1/2), and the least significant bit of A1 = 1, then round up (add one)



- A0 is always cleared.
- Performed During RND, MPYR, MACR instructions only
- Convergent rounding is particularly important for applications such as IIR Filters and FFTs, which round results after every iteration

Condition codes Z, N, V & C

A or B
Accumulator

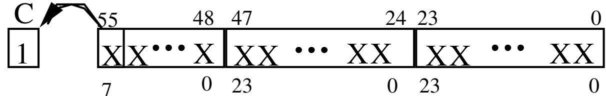
- Zero Z = 1

55 48 47 24 23 0
0 ••• 0 | 00 ••• 00 | 00 ••• 00
7 0 23 0 23 0

- Negative N = 1

55 48 47 24 23 0
1 | X••• X | XX ••• XX | XX ••• XX
7 0 23 0 23 0

- Carry (Borrow) C = 1



- Z, N, and C are cleared otherwise

V- Overflow bit:

- The Result Cannot be Represented in 56 Bits

$ 0 0	0 0 0 0 0 0	0 0 0 0 0 1

+

$ 7 F	F F F F F F	F F F F F F

=

- OVERFLOW V = 1

$ 8 0	0 0 0 0 0 0	0 0 0 0 0 0

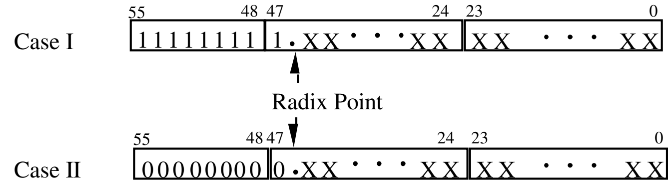
- OVERFLOW also Sets and Latches the Limit "L" Bit.

Extension & Unnormalized bits

E -Extension bit:

- E = 0 If all the Bits of the Signed Integer Portion are the same.

The number (n) is a fraction: +1 > **n** ≥ -1

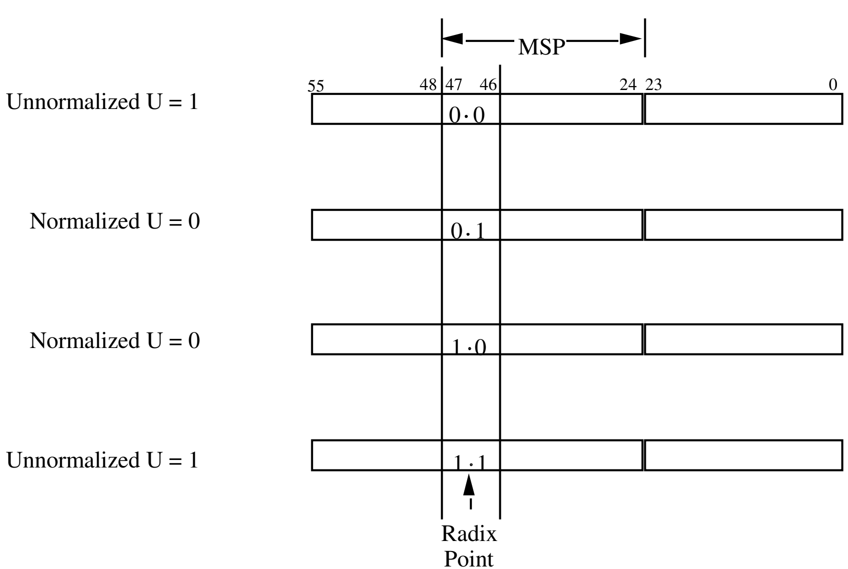


- E = 1 For all other cases, the number (n) is an Integer plus Fraction:
  -256 ≤ n < -1 OR +**1** ≤ n < +256

U -Unnormalized bit:

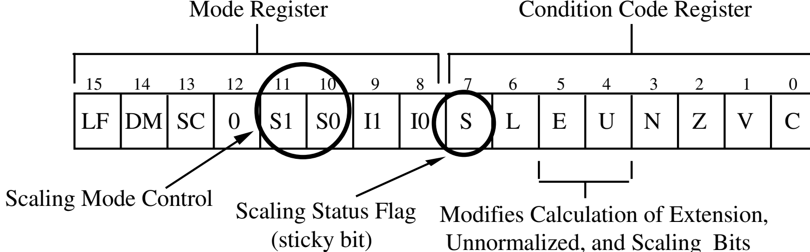
- U = 1 If the two most significant bits of the MSP are the same.

Cleared otherwise



Scaling bits

Status Register

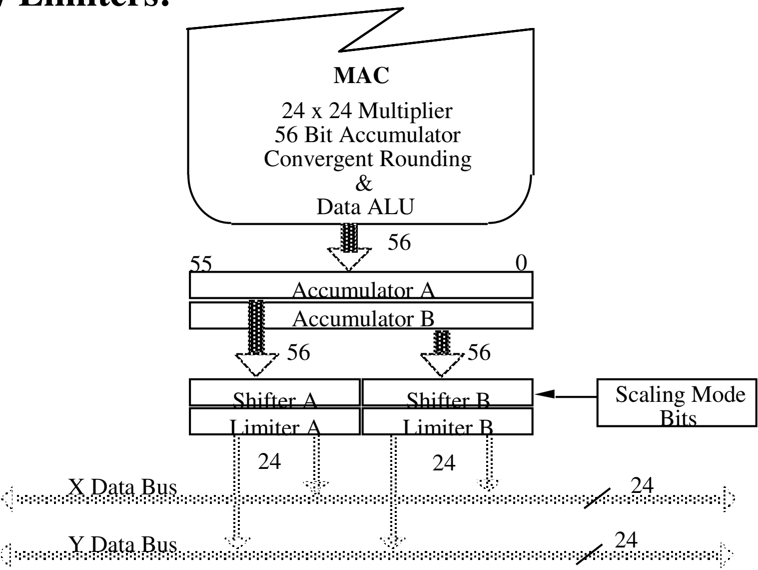


S1	S0	Rounding bit	Scaling Mode	Scaling Status Flag
0	0	23	No Scaling	S = Acc46 xor Acc45
0	1	24	Scale Down (1 bit arithmetic right shift)	S = Acc47 xor Acc46
1	0	22	Scale Up (1 bit arithmetic left shift)	S = Acc45 xor Acc44
1	1	--	Reserved for future expansion	--

NOTE: Acc is a bit in Accumulator A or B

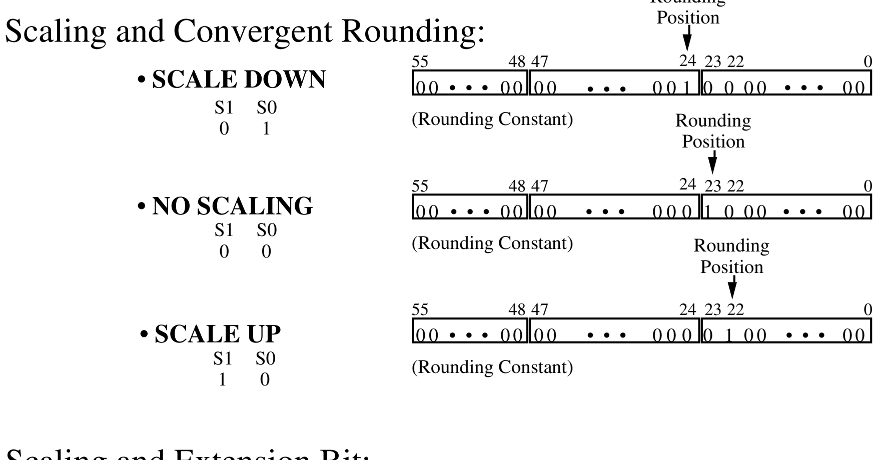
- Scaling Mode bits are cleared during reset and long interrupt service routine.
- Affects Convergent Rounding
- Scaling Status Flag is computed as shown when Accumulator A or B is moved to memory.
- Scaling Status Flag is a sticky bit and is cleared only by an instruction that specifically clears it.

Shifter / Limiters:

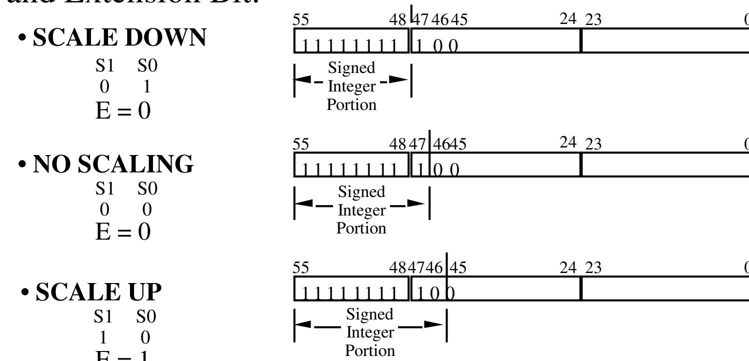


- SHIFTER Allows Dynamic Scaling without Changing Program Code.
- LIMITING Follows the Scaling Function.
- SHIFT/LIMITER Affects 24 or 48 Bit Data Read from A or B Accumulators onto the X or Y Data Buses
- Contents of Accumulators A or B are Not Changed.

Scaling Effects



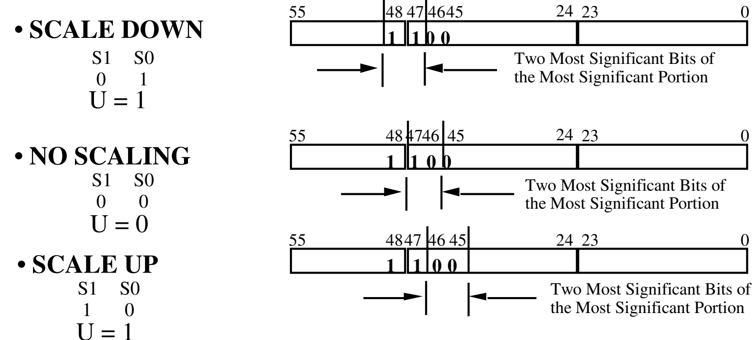
Scaling and Extension Bit:



• E = 0 If all the bits in the signed integer portion of the accumulator are the same, else E = 1

• Scaling also affects X and Y data bus limiting when transferring data from an accumulator

Scaling and Unnormalize Bit:



Architecture Exercises (1 of 2)

Complete these exercises. Multiple choice questions may have more than one correct answer.

1. The Y register is the concatenation of registers ____ and ____.
2. Matching exercise:

x0	y0	x1	y1	A	B	C bit
E bit	N bit	V bit	L bit	U bit	S bit	Z bit

a) This status bit indicates when the accumulator result no longer within the range $-1 \leq Acc < +1$____
b) This status bit indicates the accumulator result is in maximum resolution position.____
c) This status bit indicates the accumulator value just moved is within the range .25 to .5.____
d) This status bit indicates the sign of the accumulator result is incorrect.____
e) These two status flags are sticky bits.________
f) This status bit contains part of the accumulator operand.____
g) The destination for data arithmetic results.________
h) Multiplier and Multiplicands can be in any of these, for the MAC instructions.______________

3. The programmer can implement an algorithm to do dynamic scaling. This involves testing the _____ status flag and adjusting the ______ control bits, in the status register.
4. When moving the value in A2 to X or Y memory, that memory location then has the value:
A. A2 in the least significant byte, the most significant 2 bytes are the sign extension of the ls byte.
B. A2 in the least significant byte, the most significant 2 bytes are all zeroes.
C. A2 is triplicated, is the same in all three bytes.
D. A2 in the most significant byte, the least significant 2 bytes are all zeroes.
5.When the status register I1 and I0 bits are 10:
A. Interrupt requests below level 3 are ignored.
B. Only interrupts at level 2 are taken.
C. Interrupt requests below level 2 are ignored.
D. Interrupt requests below level 1 are ignored.
6. One program memory access and two data accesses can complete in one clock if:
A. All memory accesses are to on-chip resources.
B. Only one of the data memory accesses is to no-wait state external memory and the others are internal.
C. No more than two of the memory accesses are to no-wait state external memory.
D. The data accesses are to internal resources and the program access is external.
E. None of the above
7. The 15 entry on-chip H/W stack is intended to be used for:
A. Subroutine calls and returns.
B. DO loop operations.
C. Temporary storage for variables.
D. Parameters passed into a subroutine.
E. None of the above.
8. Signed fractional data has its radix point between bits:
A. 23 and 22 in X0, Y0, X1, or Y1
B. 47 and 46 in X or Y registers
C. 47 and 46 in A or B accumulators
D. 55 and 54 in A or B accumulators
E. 23 and 22 on the X memory or Y memory data buses.
9. Signed fractional data has its sign bit location in bit:
A. 23 in X0, Y0, X1, or Y1
B. 47 in X or Y registers
C. 47 in A or B accumulators
D. 55 in A or B accumulators
10. When moving data from X0 to accumulator A, the data from X0 will be found in bits
A. 23 to 0, bits 24 to 55 are not affected.
B. 47 to 24, bits 48 to 55 are sign-extended from bit 47, and bits 23 to 0 are zeroes.
C. 23 to 0, bits 24 to 55 are undefined.
D. 47 to 24, bits 48 to 55 are zeroes, and bits 23 to 0 are zeroes.
E. None of the above.

Architecture Exercises (2 of 2)

Complete these exercises. Multiple choice questions may have more than one correct answer.

11. Matching exercise: The values below in the 56-bit accumulators are convergently rounded and moved to a 24-bit destination. Indicate the resultant destination value of each.

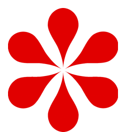
$7FFFFF $800000 $123456 $012345 $012346 $123457

a) $00 123456 700000 _________
b) $00 012345 800000 _________
c) $01 012346 800000 _________
d) $92 012345 800000 _________
e) $00 012345 800110 _________

2.3 Addressing Modes

This section is a reproduction of Chapter 2 of the *DSP563xx/6xx Family Digital Signal Processor Training Notes*[2].

[2]Copyright of Motorola, Used with Permission.



Addressing Modes

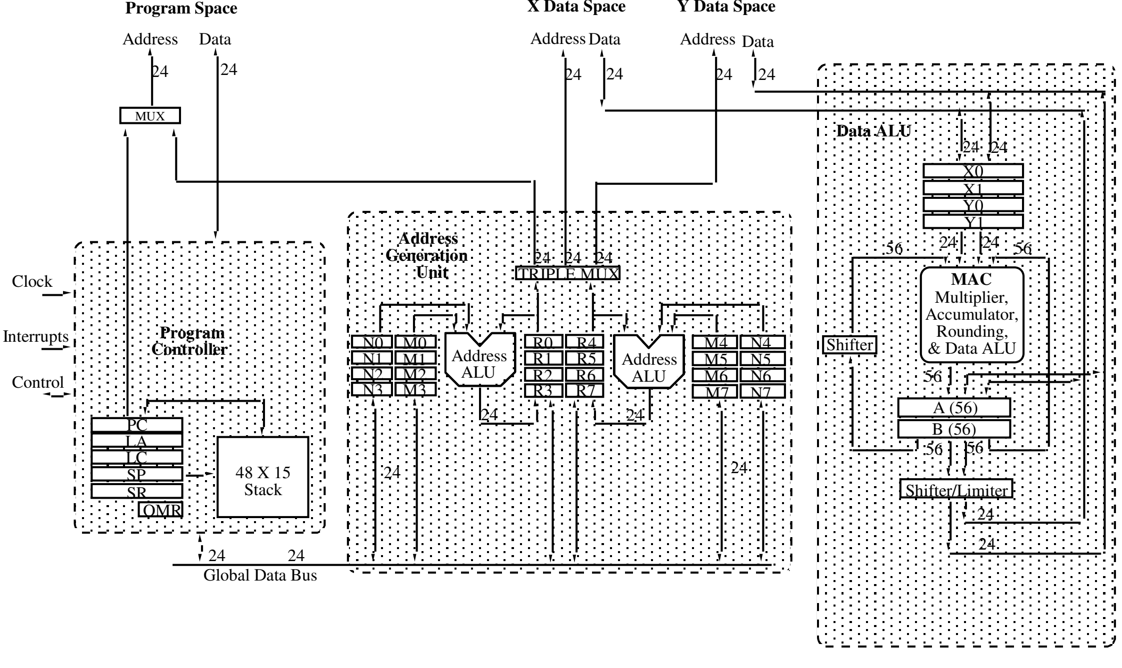
Learn how to:

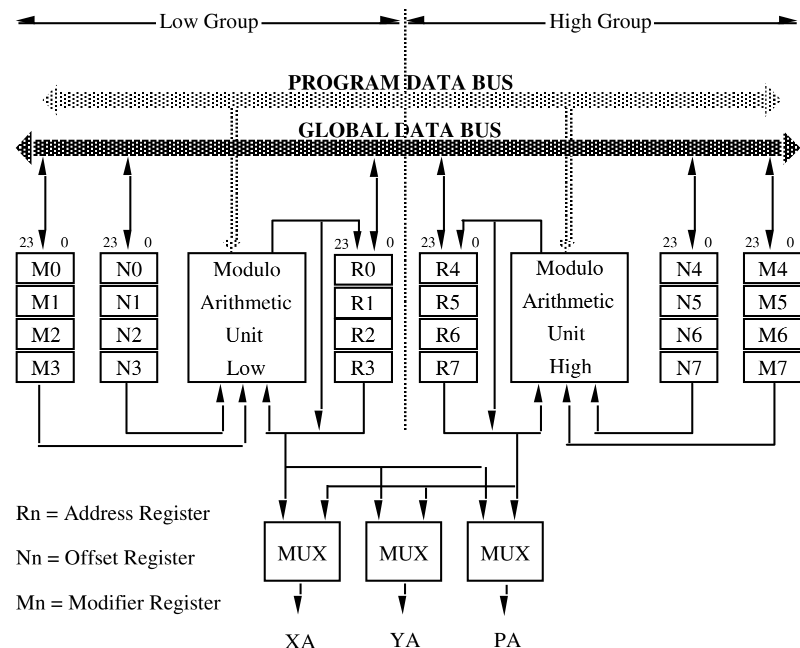
- Determine allowed address modes for various operands
- Load data into ALU registers as unsigned integer or signed fraction
- Access the highest 128 words of data memory most efficiently
- Stack and unstack registers to/from data memory
- Locate modulo and reverse carry buffers at a legal boundary
- Access successive modulo buffer entries
- Access FFT buffer results in reverse carry order
- Configure AGU registers for modulo operation
- Configure AGU registers for reverse carry operation

Table of Contents

4....Core Processor & AGU
5....AGU Program Model
6....Addressing Mode Capabilities
7....Addressing Mode Categories
8....Move Instruction
9....Immediate Data
10..Immediate Short
11..Absolute Addressing
12..Short Addressing
13..Address Register Indirect (ARI)
14..Post Inc & Dec by One
15..Post Inc & Dec by Offset
16..Predecrement & Indexed
17..Data Stack Exercise
18..Modifier Registers
19..Modifier Register Values
20..Modulo Addressing
21..Modulo addressing Exercise
22..FFT Transform
23..Bit Reversed Addressing
24..Bit Reverse Addressing Exercise

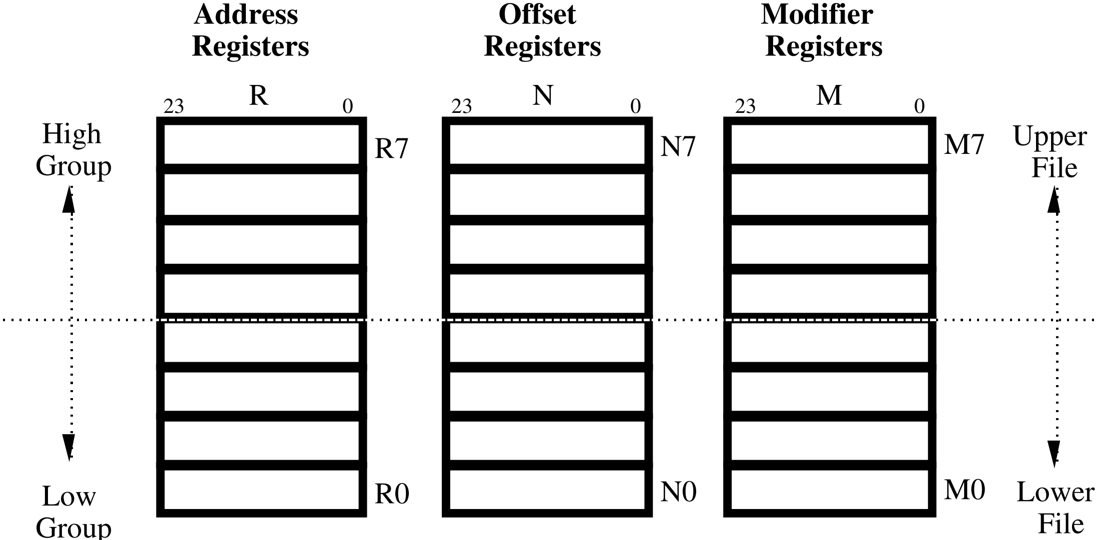
Core Processor & AGU





- Generates up to two independent addresses each Instruction Cycle.
- Performs effective Address calculations.

AGU Program Model



- Provides storage for 24 bit Addresses, Counters, and Scratch Pad Data
- ALU Registers work in groups of three having the same register number n.
- Address Register Rn, uses Offset Register Nn, and is modified by Register Mn.
- When Calculating Two Effective Addresses one must come from the lower file and the second must come from the upper file.

Addressing Mode Capabilities

Addressing Mode	Uses Mn Modifier	Operand Reference									Assembler Syntax
		S	C	D	A	P	X	Y	L	XY	
Register											**Register Name**
Data or Control Register	No	X	X	X							
Address Register Rn	No				X						Rn
Address Offset Register Nn	No				X						Nn
Address Modifier Register Mn	No				X						Mn
Address Register Indirect											
No Update	Yes					X	X	X	X	X	(Rn)
Postincrement by 1	Yes					X	X	X	X	X	(Rn)+
Postdecrement by 1	Yes					X	X	X	X	X	(Rn)-
Postincrement by Offset Nn	Yes					X	X	X	X	X	(Rn)+Nn
Postdecrement by Offset Nn	Yes					X	X	X	X		(Rn)-Nn
Indexed by Offset Nn	Yes					X	X	X	X		(Rn+Nn)
Predecrement by 1	Yes					X	X	X	X		-(Rn)
Short/Long Displacement	Yes						X	X	X		(Rn+displ)
PC Relative											
Short/Long Displacement	No					X					(PC+displ)
Address Register	No					X					(PC+Rn)
Special											
Immediate Data (24 Bits)	No					X					#xxxxxx
Absolute Address (24 Bits)	No					X	X	X	X		xxxxxx
Immediate Short Data (8 Bits)	No					X					#xx
Short Jump Address (12 Bits)	No					X					xxx
Absolute Short Address (6 Bits, zero extended)	No					X	X	X	X		aa
I/O Short Address (6 Bits, ones extended)	No						X	X			pp
Implicit	No	X	X			X					

S = System Stack Reference
C = Program Controller Register Reference
D = Data ALU Register Reference
A = Address ALU Register Reference
P = Program Memory Reference
X = X Memory Reference
Y = Y Memory Reference
L = L Memory Reference
XY = XY Memory Reference

Addressing Mode Categories

Addressing Mode	Mode MMM	Reg RRR	Addressing Categories U	P	M	A	Assembler Syntax
Register Direct							
Data or Control Register	–	–				X	
Address Register	–	–				X	Rn
Address Offset Register	–	–				X	Nn
Address Modifier Register	–	–				X	Mn
Address Register Indirect							
No Update	100	Rn		X	X	X	(Rn)
Postincrement by 1	011	Rn	X	X	X	X	(Rn)+
Postdecrement by 1	010	Rn	X	X	X	X	(Rn)-
Postincrement by Offset Nn	001	Rn	X	X	X	X	(Rn)+Nn
Postdecrement by Offset Nn	000	Rn	X		X	X	(Rn)-Nn
Indexed by Offset Nn	101	Rn			X	X	(Rn+Nn)
Predecrement by 1	111	Rn			X	X	-(Rn)
Short/Long Displacement	-	Rn			X	X	(Rn+displ)
PC Relative							
Short/Long Displacement	-	PC			X		(PC+displ)
Address Register	-	PC			X		(PC+Rn)
Special							
Immediate Data	110	100			X		#xxxxxx
Absolute Address	110	000			X	X	xxxxxx
Immediate Short Data	–	–					#xx
Short Jump Address	–	–				X	xxx
Absolute Short Address	–	–				X	aa
I/O Short Address	–	–				X	pp
Implicit	–	–				X	

Update Mode (U) The Update Addressing mode is used to modify address registers without any associated data move.

Parallel Mode (P) The Parallel Addressing mode is used in instructions where two effective addresses are required.

Memory Mode (M) The Memory Addressing mode is used to refer to operands in memory using an effective addressing field.

Alterable Mode (A) The Alterable Addressing mode is used to refer to alterable or writable registers or memory.

Move Instruction

MOVE Source 1, Destination 1 [Source 2, Destination 2]

23 — 0
MOVE Instruction
(Optional Extension Word)

Source →	**Destination**
Register	Register
Register	Memory
Memory	Register

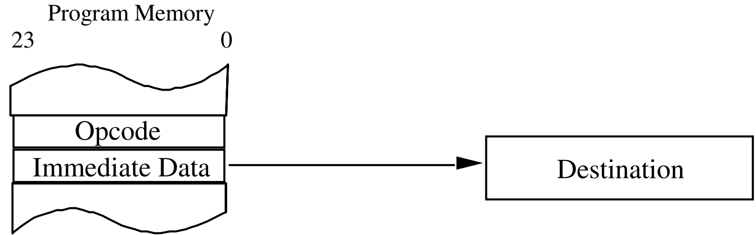
Registers	**Memory Space**
X1,X0	X
Y1,Y0	Y
A,A2,A1,A0	
B,B2,B1,B0	
R0-R7	
N0-N7	

- MEMORY to MEMORY MOVES not supported
  (Except for MOVEP)

Immediate Data

Assembler Syntax: #xxxxxx

Operands Referenced: P Memory



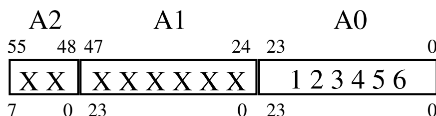
Immediate into 24 bit destination:

Example: MOVE #$123456,A0

Before Execution

A2 (55–48)	A1 (47–24)	A0 (23–0)
X X	X X X X X X	X X X X X X
7 0	23 0	23 0

After Execution



Immediate into 56 bit Accumulator:

Example: MOVE #$123456,A

Before Execution

A2 (55–48)	A1 (47–24)	A0 (23–0)
X X	X X X X X X	X X X X X X
7 0	23 0	23 0

After Execution

A2 (55–48)	A1 (47–24)	A0 (23–0)
0 0	1 2 3 4 5 6	0 0 0 0 0 0
7 0	23 0	23 0

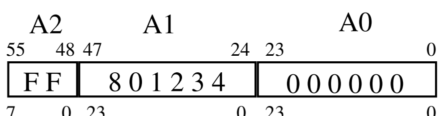
Negative Immediate into 56 bit Accumulator:

Example: MOVE #$801234,A

Before Execution

A2 (55–48)	A1 (47–24)	A0 (23–0)
X X	X X X X X X	X X X X X X
7 0	23 0	23 0

After Execution



Immediate Short

Eight Bit Data
Assembler Syntax: #xx
Operands Referenced: P Memory

If Destination is A2, A1, A0, B2, B1, B0, R0-R7, N0-N7
then 8 bit Immediate Short Operand is treated as Unsigned Integer:

Example: MOVE #$FF,A1

Before Execution

A2 A1 A0
55 48 47 24 23 0
XX XXXXXX XXXXXX
7 0 23 0 23 0

After Execution

A2 A1 A0
55 48 47 24 23 0
XX 0000FF XXXXXX
7 0 23 0 23 0

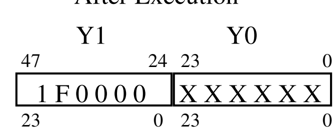
If Destination is X1, X0, Y1, Y0, A,B
then 8 bit Immediate Short Operand is treated as Signed Fraction:

Positive Immediate into 24 bit register
Example: MOVE #$1F,Y1

Before Execution

Y1 Y0
47 24 23 0
XXXXXX XXXXXX
23 0 23 0

After Execution

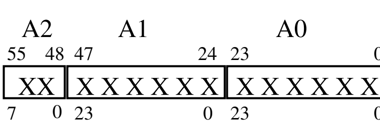


8 Bits, Sign Extended, Zero Filled

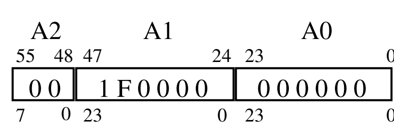
Positive Immediate into 56 bit accumulator:

Example: MOVE #$1F,A

Before Execution



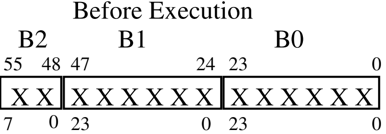
After Execution



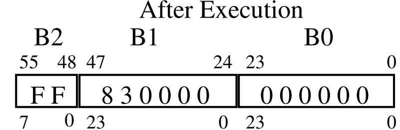
Negative Immediate into 56 bit accumulator:

Example: MOVE #$83,B

Before Execution



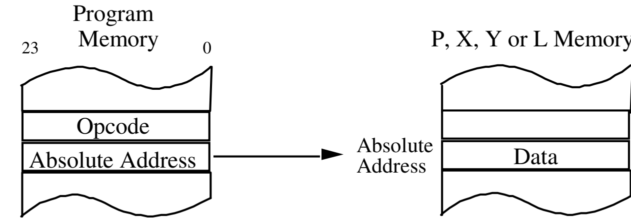
After Execution



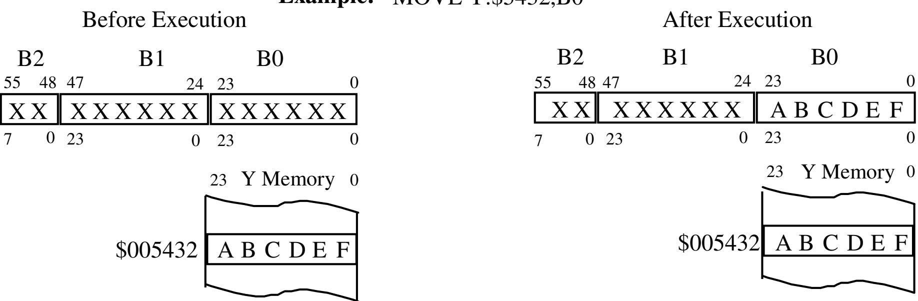
Absolute Addressing

24 Bit Address:

Assembler Syntax: xxxxxx
Operands Referenced: P, X, Y, L Memories



Example: MOVE Y:$5432,B0

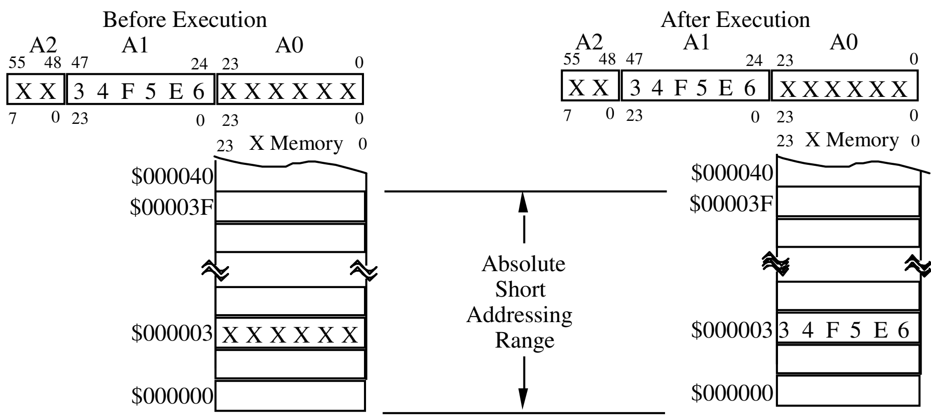


6 Bits, Zero Extended:

Assembler Syntax: aa

Operands Referenced: P, X, Y, L

Example: MOVE A1,X:3



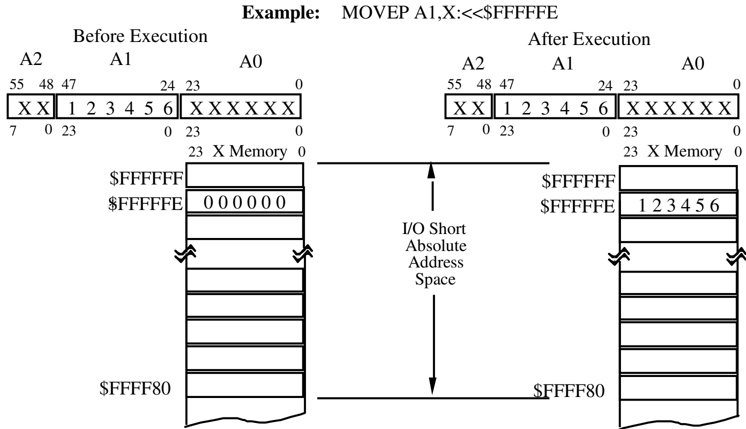
- Addresses Lowest 64 words
- Interrupt Vectors
- X, Y, L Data RAM
- No Extension Word
- One Word Instruction

Short Addressing

6 Bits, Ones Extended:

Assembler Syntax: pp or qq
Operands Referenced: X and Y, I/O Space

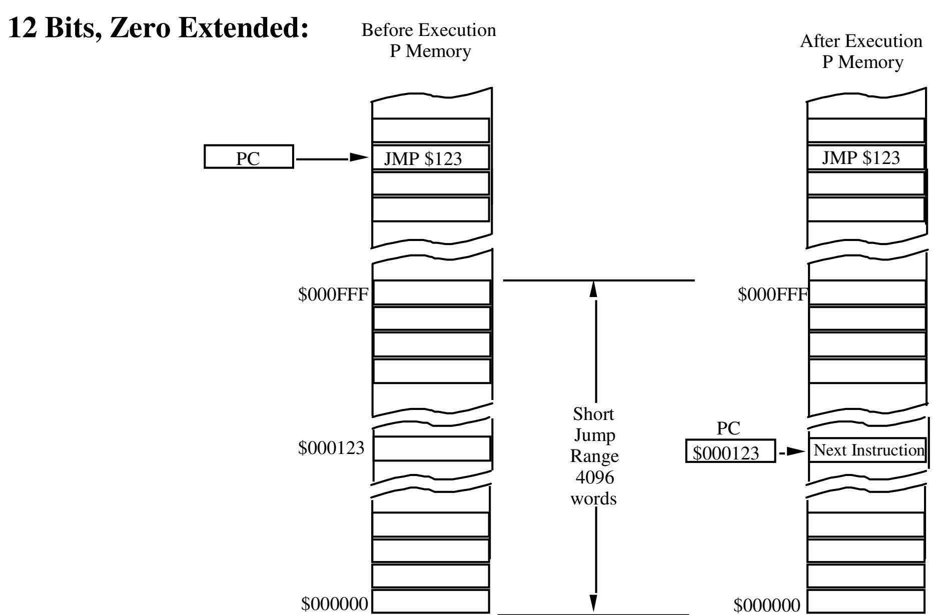
Example: MOVEP A1,X:<<$FFFFFE



* Contents of Interrupt Priority Register for Peripherals after reset

- Upper 128 Words of X or Y Memory
- << indicates I/O short addressing

12 Bits, Zero Extended:



- Used by Conditional JUMPS and JUMP to Subroutines

Address Register Indirect
(ARI)

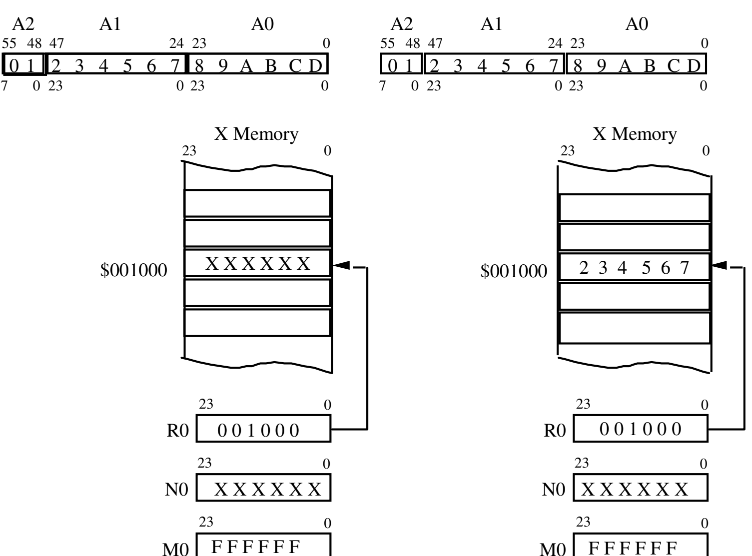
ARI Modes:

- No Update
- Postincrement by One
- Postincrement by Nn
- Postdecrement by One
- Postdecrement by Nn
- Indexed by Offset Nn
- Predecrement by 1
- All Modes of Address Register Indirect use the Mn Modifier Registers.

No Update:

- Assembler Syntax: (Rn)
- Operands referenced: P, X, Y, XY, L memories.

Example: MOVE A1, X:(R0)

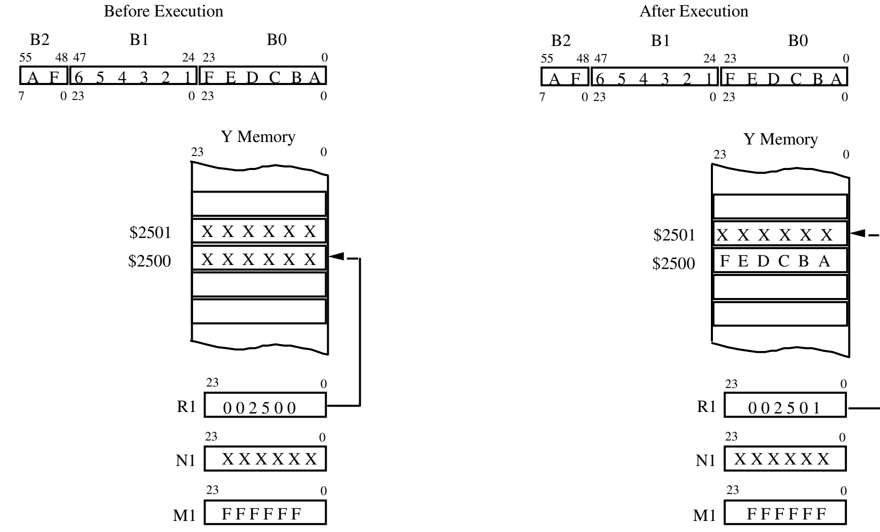


Post Inc & Dec by One

Postincrement by one:

- Assembler Syntax: (Rn) +
- Operands referenced: P, X, Y, XY, L memories.

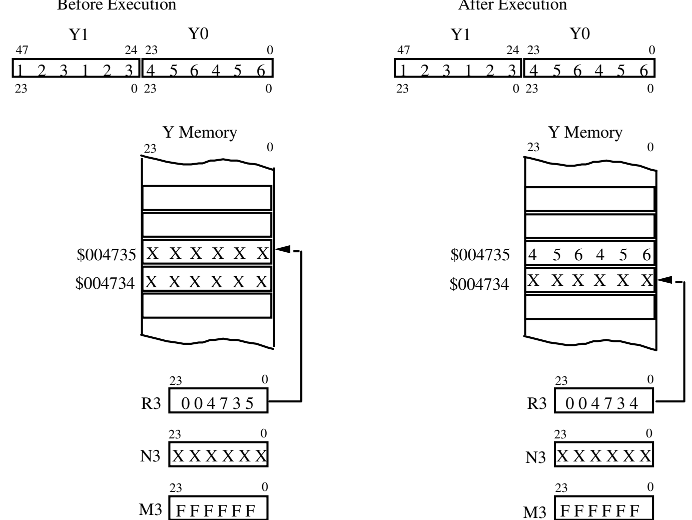
Example: MOVE B0,Y:(R1) +



Postdecrement by one:

- Assembler Syntax: (Rn) -
- Operands referenced: P, X, Y, XY, L memories.

Example: MOVE Y0,Y:(R3) -

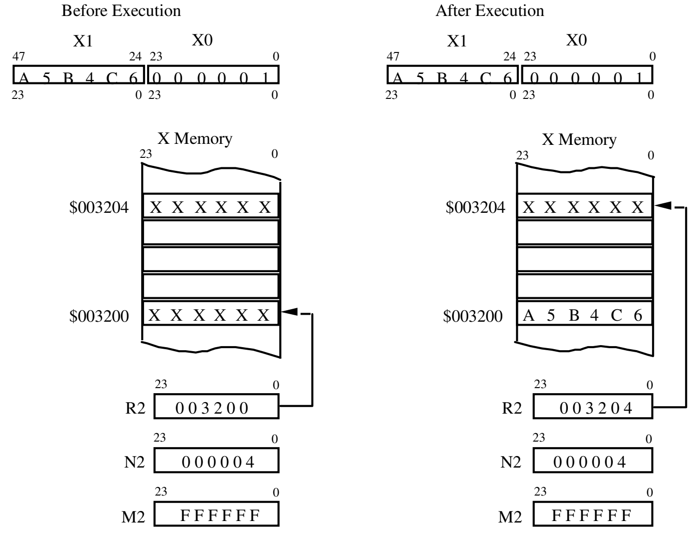


Post Inc & Dec by Offset

Postincrement by Offset:

- Assembler Syntax: (Rn) + Nn
- Operands referenced: P, X, Y, L, XY memories.

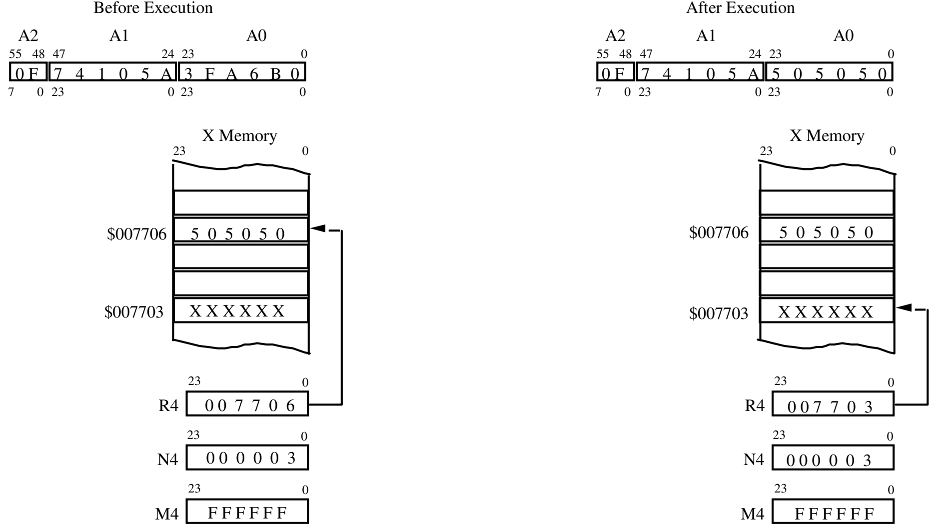
Example: MOVE X1, X:(R2) + N2



Postdecrement by Offset:

- Assembler Syntax: (Rn) - Nn
- Operands referenced: P, X, Y, L memories.

Example: MOVE X:(R4) - N4, A0

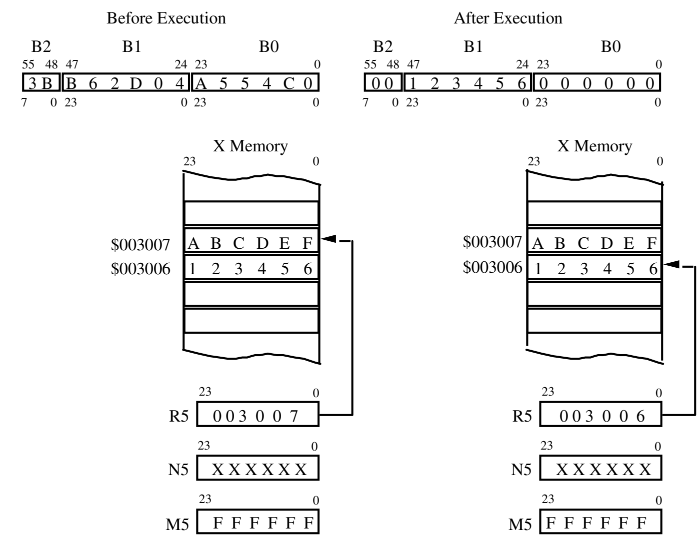


Predecrement & Indexed

Predecrement by one:

- Assembler Syntax: -(Rn)
- Operands referenced: P, X, Y, L memories.

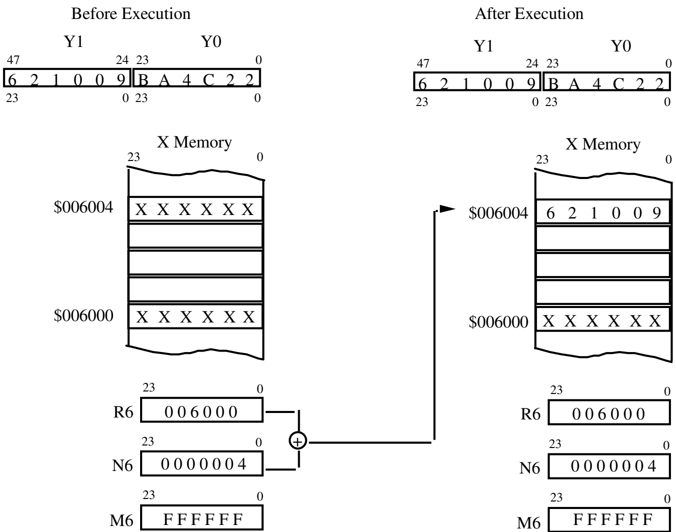
Example: MOVE X:- (R5), B



Indexed by Offset:

- Assembler Syntax: (Rn + Nn)
- Operands referenced: P, X, Y, L memories.

Example: MOVE Y1,X:(R6+N6)



Data Stack

Exercise

Using the address register indirect address modes, create a data stack. Use r7 as your stack pointer; assume m7 is set up for linear addressing. Initialize your stack pointer so that your stack starts at the lowest address of on-chip X RAM. Then save X0 on the stack and restore X0 from the stack.

Put your code here:

Suggested program steps:

1. Initialize r7 to the bottom of on-chip X memory

2.Save X0 on the stack

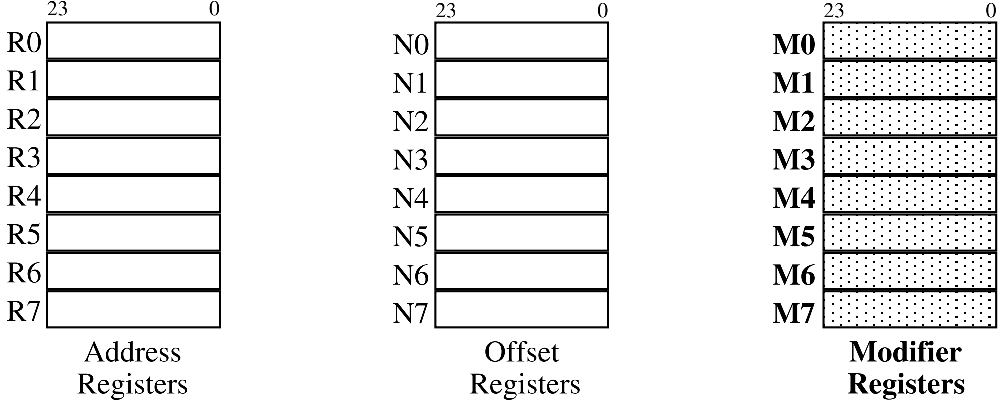
3. Restore X0 from the stack

X memory

$0007FF

$000000

Modifier Registers



USED BY ALL FORMS OF ADDRESS REGISTER INDIRECT

No Update	(Rn)
Postincrement by 1	(Rn)+
Postdecrement by 1	(Rn)-
Postincrement by Offset Nn	(Rn)+Nn
Postdecrement by Offset Nn	(Rn)-Nn
Indexed by Offset Nn	(Rn+Nn)
Predecrement by 1	-(Rn)
Indexed by Displacement	(Rn+displacement)

• Modifier register Mn determines the effective address calculation that is derived from Rn and Nn:
- If Mn = 0, ea calculation is Bit Reverse
- If Mn =1-$XX7FFF, ea calculation is Modulo
- If Mn = $XX8001-$00BFFF, ea calculation is multiple wrap modulo
- If Mn = $XXFFFF, ea calculation is Linear (Modulo 2^{24})
(XX means don't care)

Modifier Register Values

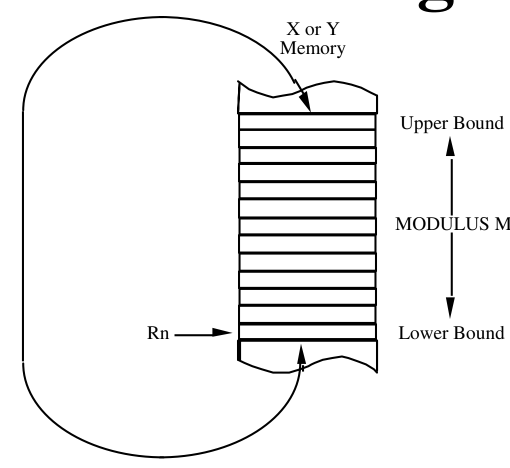
- Modulo can be any value from 1 to 32767
- Multiple-wrap Modulo is a value $8001-$BFFF and must be 2^n-1, where "n" is 0-14

Modifier Mn Value	Addressing Mode Arithmetic
0	Reverse Carry (Bit Reverse)
1	Modulo 2
2	Modulo 3
•	•
•	•
7FFE	Modulo 32767
7FFF	Modulo 32768
8000	Reserved
8001	Multiple Wrap-Around Modulo 2
8002	Reserved
8003	Multiple Wrap-Around Modulo 4
8004-8006	Reserved
8007	Multiple Wrap-Around Modulo 8
8008-800E	Reserved
800F	Multiple Wrap-Around Modulo 2^4
8010-801E	Reserved
801F	Multiple Wrap-Around Modulo 2^5
8020-803E	Reserved
803F	Multiple Wrap-Around Modulo 2^6
8040-807E	Reserved
807F	Multiple Wrap-Around Modulo 2^7
8080-80FE	Reserved
80FF	Multiple Wrap-Around Modulo 2^8
8100-81FE	Reserved
81FF	Multiple Wrap-Around Modulo 2^9
8200-83FE	Reserved
83FF	Multiple Wrap-Around Modulo 2^{10}
8400-87FE	Reserved
87FF	Multiple Wrap-Around Modulo 2^{11}
8800-8FFE	Reserved
8FFF	Multiple Wrap-Around Modulo 2^{12}
9000-9FFE	Reserved
9FFF	Multiple Wrap-Around Modulo 2^{13}
B000-BFFE	Reserved
BFFF	Multiple Wrap-Around Modulo 2^{14}
C000-FFFE	Reserved
FFFF	Linear (Modulo 2^{24})

Bits 23-16 of the modifier registers are don't care.

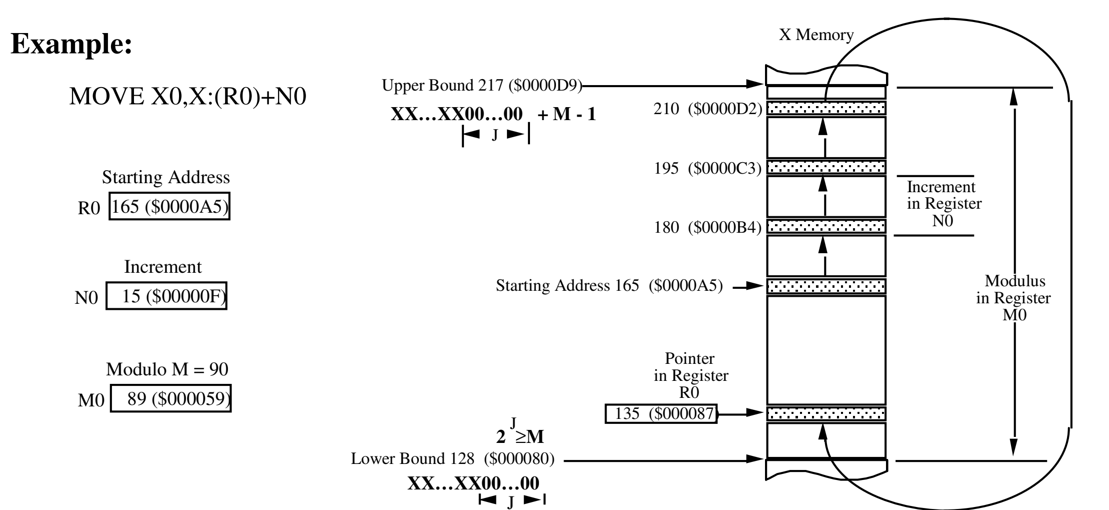
Modulo Addressing

F.I.F.O.'s
Circular Queues
Delay Lines
Shift Registers
Sample Buffers
Finite Sum of Products



Modulo Register setup:

1. Modifier Register Mn = M-1. — MODULUS =M
2. Lower Bound = **XX...XX00...00** (bits 23 … 0, lowest J bits zero) where $2^J \geq M$ — Beginning of Table (Minimize J)
3. Upper Bound = **XX...XX00...00 + M - 1** (bits 23 … 0, lowest J bits zero) — End of Table
4. Lower Bound ≤ Address Register Rn ≤ Upper Bound — Starting Point within Table
5. Offset Register Nn = Increment ≤ M — Desired Increment (if any)



Modulo addressing

Exercise

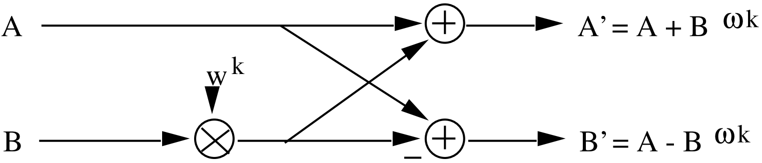
Write code to initialize R0 to point at samples and R4 to point at coefficients using modulo addressing, modulo 8. Also, move the first sample into register X0 and the first coefficient into register Y0, post incrementing both address registers. The 8 samples are stored in RAM beginning at X:$40 and the 8 coefficients are stored in RAM beginning at Y:$80.

Write your program here:	**Suggested program steps:**
`npts     equ 8`	Number of points in each buffer
`         org X:$40`	Originate sample table at X:$40
`samples  dsm npts`	Define storage for modulo 8 table
`         org Y:$80`	Originate coefficient table at Y:$80
`coeff    dc .9,-.1,-.2,.5,.5,-.2,-.1,.9`	Define constant coefficients
`         org P:$100`	Originate program at P:$100
	1. Initialize r0 and r4
	2. Initialize M registers for Modulo 8
	3. Move first sample and coefficient into X0 and Y0.

Note: please read Sec. 3.2 before starting this exercise.

FFT Transform

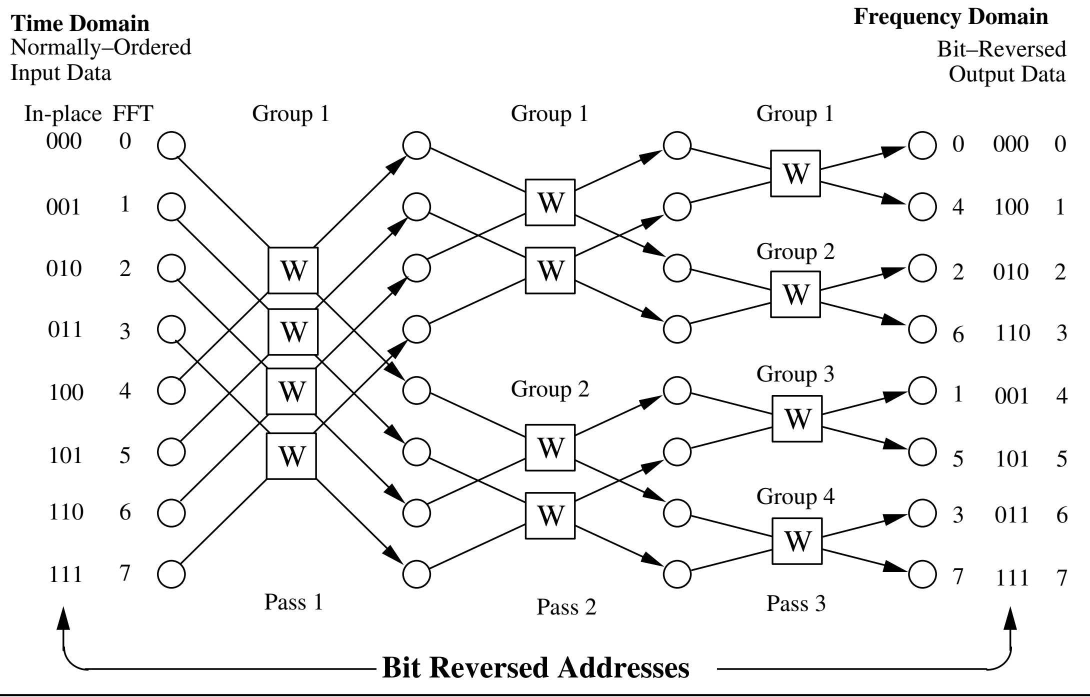
Radix 2, decimation in time, complex FFT butterfly



Schematic Symbol

A
B
W
A'
B'
= Radix 2 Butterfly

8 POINT FFT TRANSFORM



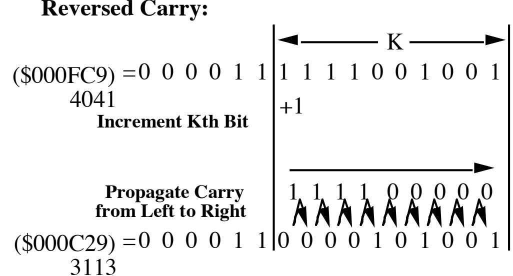
Bit Reversed Addressing

Bit Reverse/Reverse Carry Algorithm:

N = 1024 ($000400) Lower Bound = 3 x N = 3 x 1024 = 3072 ($000C00)

K =10 ($A) Upper Bound = 3 x N + 2^K ñ 1 = 3072 + 1024 ñ 1 = 4095 ($000FFF)

Reversed Carry:



Bit Reverse Register setup:

1. Modifier Register Mn = $0000 — N = Table Size; Select Bit Reverse Addressing Mode
2. Lower Bound = **XX . . . XX00 . . . . . 00** (bits 23 … 0, lower K bits zero) where $2^K \geq N$ — Beginning of Table (Minimize K)
3. Upper Bound = **XX . . . XX00 . . . . . 00 + 2^K - 1** (bits 23 … 0, lower K bits) — End of Table
4. Lower Bound ≤ Address Register Rn ≤ Upper Bound — Starting Point within Table
5. Offset Register Nn = 2^{K-1} — Initialize Offset Register

Bit Reversed Example:

	Starting Address	Array Offset		Increment		Bit Reverse
R0	3072 ($000C00)	(0)	N0	512 ($000200)	M0	0 ($000000)

MOVE X:(R0)+N0,X0

R0 3584 ($000E00) (512)

MOVE X:(R0)+N0,X0

R0 3328 ($000D00) (256)

MOVE X:(R0)+N0,X0

R0 3840 ($000F00) (768)

X Memory

Upper Boundary 4095 ($000FFF) **XX . . . XX11 . . . . . 11** (K)

$2^K \geq N$

N=1024 Word FFT Buffer

Lower Boundary 3072 ($000C00) **XX . . . XX00 . . . . . 00** (K)

Bit Reverse Addressing

Exercise

Write initialization code to use an FFT buffer of the minimum number of points greater than 100 and move data from the buffer, in bit reverse order, into register X0. Use r0 as a pointer to the buffer. Locate the buffer in internal X data RAM, between address $10 and address $100. **Fill in the blanks**: N(number of points in buffer)= ; Lower bound= ($); Upper bound= ($)

Write your program here:

```
pts      equ ________
         org X:______
samples dsr  pts
         org P:$100
```

Suggested program steps:

1. Number of points in FFT
2. Originate table in X memory
   Define storage for reverse carry, N point table
   Originate program at P:$100
3. Initialize r0
4. Initialize n0
5. Initialize m0
6. Move instruction that will be repeated for each point.

2.4 Parallel Moves

This section is a reproduction of Chapter 3 of the *DSP563xx/6xx Family Digital Signal Processor Training Notes*[3].

[3]Copyright of Motorola, Used with Permission.

Parallel Moves

Learn how to:

- Do arithmetic or logic operations in the data ALU concurrent with the
  - move of data from register to register, or
  - move of an immediate data value into a register, or
  - move of the results of up to two accumulators to data memory, or
  - move of data from up to two data memories into data ALU input registers, or
  - move of data to/from memory & a register, & move data register to register, or
  - move of long data to/from memory and a register, or
  - update of an address register from an address calculation

Table of Contents

4....Instruction Set
5....Parallel Move Instructions
6....Immediate Short Data Move
7....Register to Register Data Move
8....Address Register Update
9....X Memory Data Move
10..X Mem. & Register Data Move
11..Y Memory Data Move
12..Register Y Mem. & Data Move
13..Long Memory Data Move
14..XY Memory Data Move
15..Parallel Move Summary
16..Parallel Move Exercises

Instruction Set

- MPU like
- Makes pipeline invisible
- 6 groups
  - Move instructions
  - Arithmetic instructions†
  - Logical instructions †
  - Bit manipulation instructions
  - Program control instructions
  - Loop instructions
- 30 instructions allow parallel moves

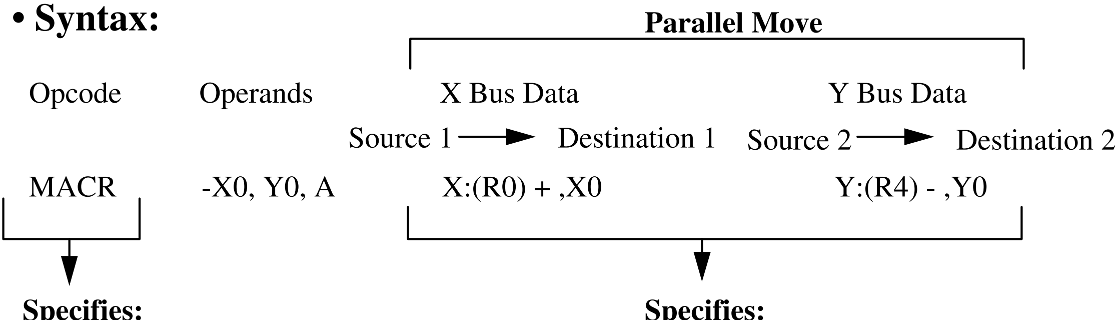
† Most arithmetic and logical instructions allow parallel data moves.

Parallel Move Types:

- No Parallel Move
- Immediate Short Data Move
- Register to Register Data Move
- Address Register Update
- X Memory Data Move
- X Memory and Register Data Move
- Y Memory Data Move
- Register and Y Memory Data Move
- Long Memory Data Move
- XY Memory Data Move

Parallel Move Instructions

• Syntax:



Specifies:

- Data ALU Operation
- Logical Operation
- Convergent Rounding

Supports:

- Condition Code Bits 0 through 5

Specifies:

- Up to two optional data transfers
- Two different addressing modes
- Address space qualifiers [X:, Y:, L:, XY:]

Supports:

- Scaling
- Limiting
- Sign extension and least significant zero fill
- Duplicate sources
- Duplicate destinations not allowed

Example:

Parallel Data Move

MACR -X0, Y0, A

1. Negate X0
2. 24 x 24 Bit Multiply X0 x Y0
3. 56 Bit Sum into Accumulator A
4. Convergently Round A
5. Update Condition Code Register

A,X:(R0)+N0

6. MOVE A to X Memory
7. Scale
8. Limit (If Necessary)
9. Calculate Next Address (R0)+ N0
10. Use Modifier Register M0 For
    – Linear Addressing
    – Modulo M
    – Bit Reverse

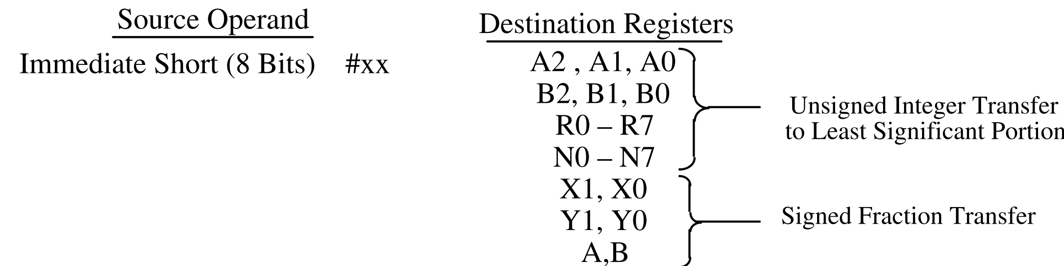
Y:(R4)-,B

11. MOVE Y Memory to B
12. Sign extend and Zero fill
13. Calculate Next Address (R4) -
14. Use Modifier Register M4 For
    – Linear Addressing
    – Modulo M
    – Bit Reverse

Immediate Short Data Move

Assembler Syntax:		PARALLEL MOVE
Opcode	Operand	#xx, Destination

Addressing Modes



Unsigned Integer

	Opcode	Operand	PARALLEL MOVE
Example 1:	ADD	A,B	#$25,A0

Before Execution

A	01	000001	000001
B	06	FFFFFE	FFFFFE

After Execution

A	01	000001	000025
B	07	FFFFFF	FFFFFF

Signed Fraction

	Opcode	Operand	PARALLEL MOVE
Example 2:	ADD	B,A	#$C7,B

Before Execution

A	01	000001	000001
B	01	000001	000001

After Execution

A	02	000002	000002
B	FF	C70000	000000

Register to Register Data Move

Assembler Syntax:		PARALLEL MOVE
Opcode	Operand	Source, Destination

Addressing Modes

Source Registers		Destination Registers
X1, X0	→	X1, X0
Y1, Y0		Y1, Y0
A, A2 , A1, A0		A, A2 , A1, A0
B, B2, B1, B0		B, B2, B1, B0
R0 – R7		R0 – R7
N0 – N7		N0 – N7

Example 1:

Opcode	Operand	PARALLEL MOVE
ADD	B,A	A,N7

Before Execution

A	0 0	1 2 3 4 5 6	A B C D E F
B	1 2	6 5 4 3 2 1	9 A B C D E
N7			0 0 0 0 0 0

After Execution

A	1 2	7 7 7 7 7 8	4 6 8 A C D
B	1 2	6 5 4 3 2 1	9 A B C D E
N7			1 2 3 4 5 6

Example 2:

Opcode	Operand	PARALLEL MOVE
ADD	B,A	X0,B

Before Execution

A	F E	D C B A 9 8	7 6 5 4 3 2
B	1 2	3 4 5 6 7 8	9 A B C D E
X		0 1 0 1 0 1	1 2 3 4 5 6

After Execution

A	1 1	1 1 1 1 1 1	1 1 1 1 1 0
B	0 0	1 2 3 4 5 6	0 0 0 0 0 0
X		0 1 0 1 0 1	1 2 3 4 5 6

Address Register Update

Assembler Syntax:		PARALLEL MOVE
Opcode	Operand	<Effective Address>

Addressing Modes

Effective Address:		**Address Register**
Postincrement by 1	(Rn)+	R0-R7
Postdecrement by 1	(Rn)-	
Postincrement by Offset Nn	(Rn)+Nn	
Postdecrement by Offset Nn	(Rn)-Nn	

Example 1:

Opcode	Operand	PARALLEL MOVE
ADD	A,B	(R1)+N1

Before Execution

A	1 1	2 22 22 2 2	3 3 3 3 3 3
B	4 4	5 5 5 5 5 5	6 6 6 6 6 6
R1	0 0 2 0 0 0		
N1	0 0 F 0 0 0		

After Execution

A	1 1	2 2 2 2 2 2	3 3 3 3 3 3
B	5 5	7 7 7 7 7 7	9 9 9 9 9 9
R1	0 1 1 0 0 0		
N1	0 0 F 0 0 0		

Example 2:

Opcode	Operand	PARALLEL MOVE
ADD	A,B	(R0)-

Before Execution

A	1 2	3 4 5 6 7 8	9 A B C D E
B	1 2	3 4 5 6 7 8	9 A B C D E
R0	0 0 2 0 0 0		

After Execution

A	1 2	3 4 5 6 7 8	9 A B C D E
B	2 4	6 8 A C F 1	3 5 7 9 B C
R0	0 0 1 F F F		

X Memory Data Move

Assembler Syntax: PARALLEL MOVE

Opcode	Operand	X:<Effective Address>,Destination
Opcode	Operand	Source,X:<Effective Address>
Opcode	Operand	#xxxxxx, Destination

Effective Address:

Absolute Short Address	aa
Absolute Address	xxxx
Address Register Indirect with;	
No Update	(Rn)
Postincrement by 1	(Rn)+
Postdecrement by 1	(Rn)-
Postincrement by Offset Nn	(Rn)+Nn
Postdecrement by Offset Nn	(Rn)-Nn
Indexed by Offset Nn	(Rn+Nn)
Predecrement by 1	-(Rn)
Indexed by Displacement	(Rn+displ)

Source or Destination

X1,X0
Y1,Y0
A,A2,A1,A0
B,B2,B1,B0
R0-R7
N0-N7

Example 1:

Opcode	Operand	PARALLEL MOVE
ADD	Y0,B	A,X:$1000

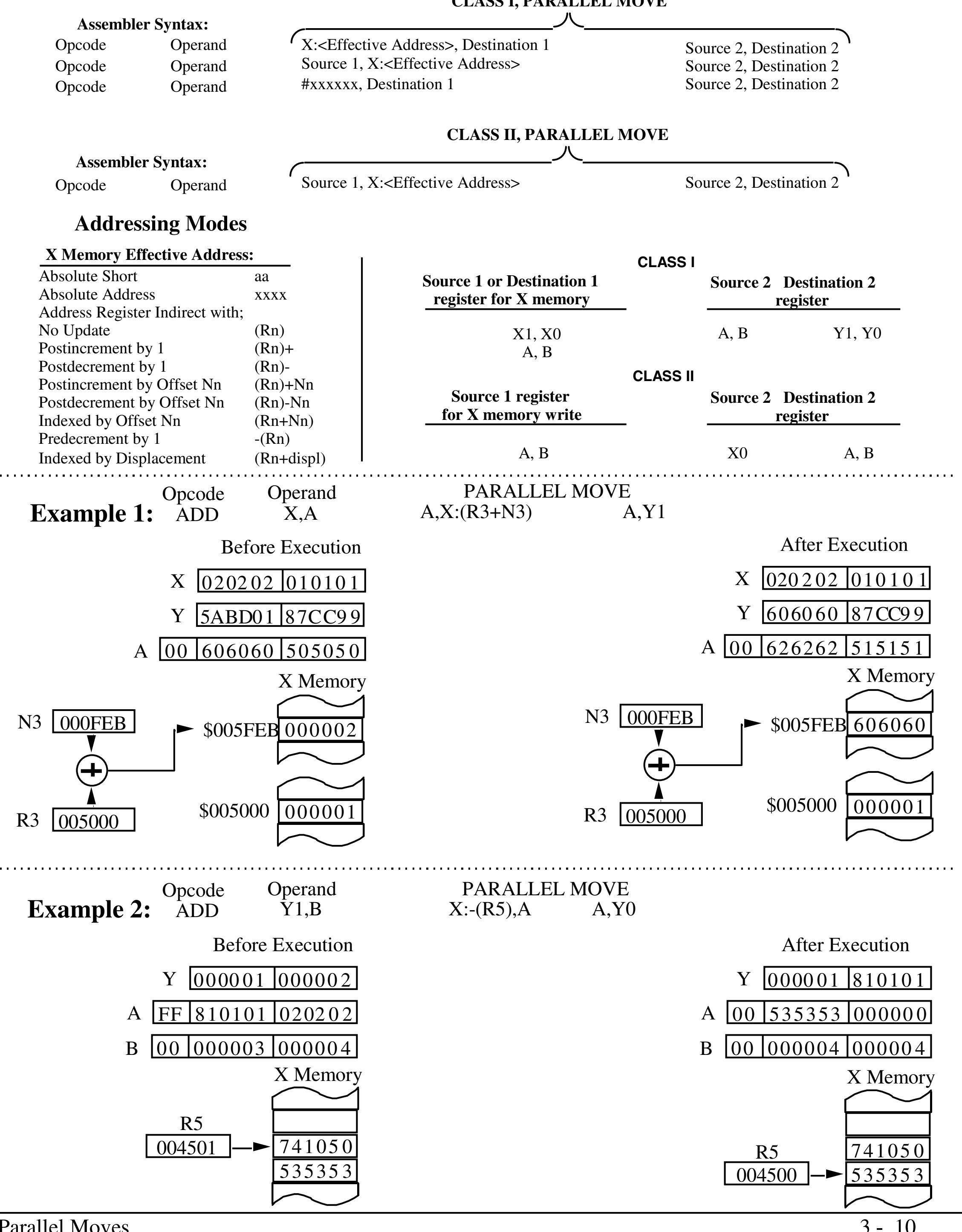
	Before Execution	After Execution
Y	777777 000023	777777 000023
A	00 123456 789ABC	00 123456 789ABC
B	00 000052 666666	00 000075 666666
X Memory $001000	000002	123456

Example 2:

Opcode	Operand	PARALLEL MOVE
ADD	X,A	X:-(R6), B

	Before Execution	After Execution
X	800000 000000	800000 000000
A	00 800000 000000	00 000000 000000
B	00 111111 222222	FF C03B9F 000000
R6	0051F9	0051F8
X Memory	6B44F1 (R6 points here), C03B9F	6B44F1, C03B9F (R6 points here)

X Mem. & Register Data Move



CLASS I, PARALLEL MOVE

Assembler Syntax:

Opcode	Operand	X:<Effective Address>, Destination 1	Source 2, Destination 2
Opcode	Operand	Source 1, X:<Effective Address>	Source 2, Destination 2
Opcode	Operand	#xxxxxx, Destination 1	Source 2, Destination 2

CLASS II, PARALLEL MOVE

Assembler Syntax:

Opcode	Operand	Source 1, X:<Effective Address>	Source 2, Destination 2

Addressing Modes

X Memory Effective Address:	
Absolute Short	aa
Absolute Address	xxxx
Address Register Indirect with;	
No Update	(Rn)
Postincrement by 1	(Rn)+
Postdecrement by 1	(Rn)-
Postincrement by Offset Nn	(Rn)+Nn
Postdecrement by Offset Nn	(Rn)-Nn
Indexed by Offset Nn	(Rn+Nn)
Predecrement by 1	-(Rn)
Indexed by Displacement	(Rn+displ)

CLASS I

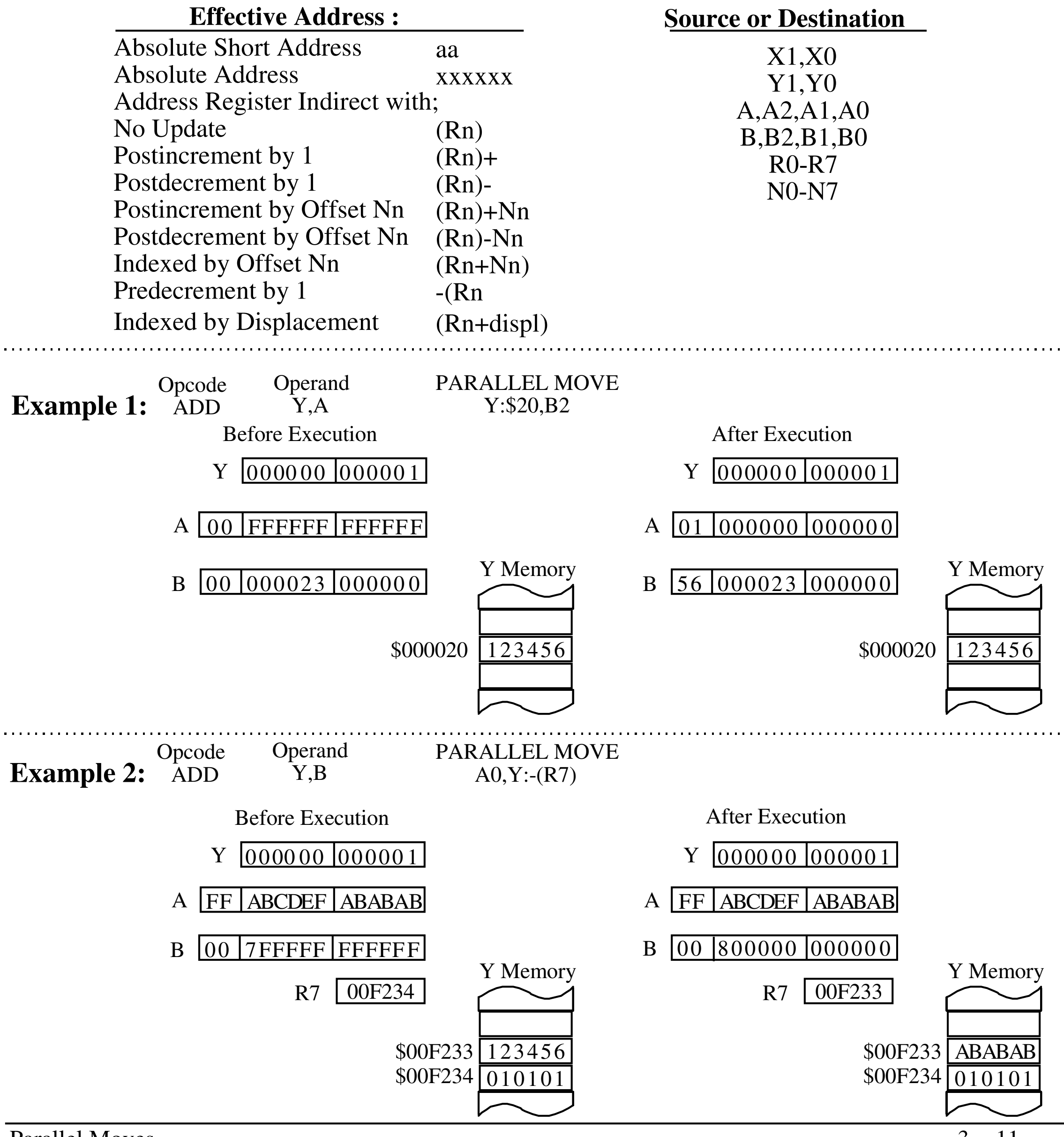
Source 1 or Destination 1 register for X memory	Source 2	Destination 2 register
X1, X0 A, B	A, B	Y1, Y0

CLASS II

Source 1 register for X memory write	Source 2	Destination 2 register
A, B	X0	A, B

Example 1: Opcode ADD, Operand X,A, PARALLEL MOVE A,X:(R3+N3) A,Y1

Example 2: Opcode ADD, Operand Y1,B, PARALLEL MOVE X:-(R5),A A,Y0



Y Memory Data Move

Assembler Syntax: **PARALLEL MOVE**

Opcode	Operand	Y:<Effective Address>,Destination
Opcode	Operand	Source,Y:<Effective Address>
Opcode	Operand	#xxxxxx, Destination

Addressing Modes

Effective Address :	
Absolute Short Address	aa
Absolute Address	xxxxxx
Address Register Indirect with;	
No Update	(Rn)
Postincrement by 1	(Rn)+
Postdecrement by 1	(Rn)-
Postincrement by Offset Nn	(Rn)+Nn
Postdecrement by Offset Nn	(Rn)-Nn
Indexed by Offset Nn	(Rn+Nn)
Predecrement by 1	-(Rn
Indexed by Displacement	(Rn+displ)

Source or Destination
X1,X0
Y1,Y0
A,A2,A1,A0
B,B2,B1,B0
R0-R7
N0-N7

Example 1:

Opcode	Operand	PARALLEL MOVE
ADD	Y,A	Y:$20,B2

Before Execution

Y 000000 000001

A 00 FFFFFF FFFFFF

B 00 000023 000000

Y Memory

$000020 123456

After Execution

Y 000000 000001

A 01 000000 000000

B 56 000023 000000

Y Memory

$000020 123456

Example 2:

Opcode	Operand	PARALLEL MOVE
ADD	Y,B	A0,Y:-(R7)

Before Execution

Y 000000 000001

A FF ABCDEF ABABAB

B 00 7FFFFF FFFFFF

R7 00F234

Y Memory

$00F233 123456

$00F234 010101

After Execution

Y 000000 000001

A FF ABCDEF ABABAB

B 00 800000 000000

R7 00F233

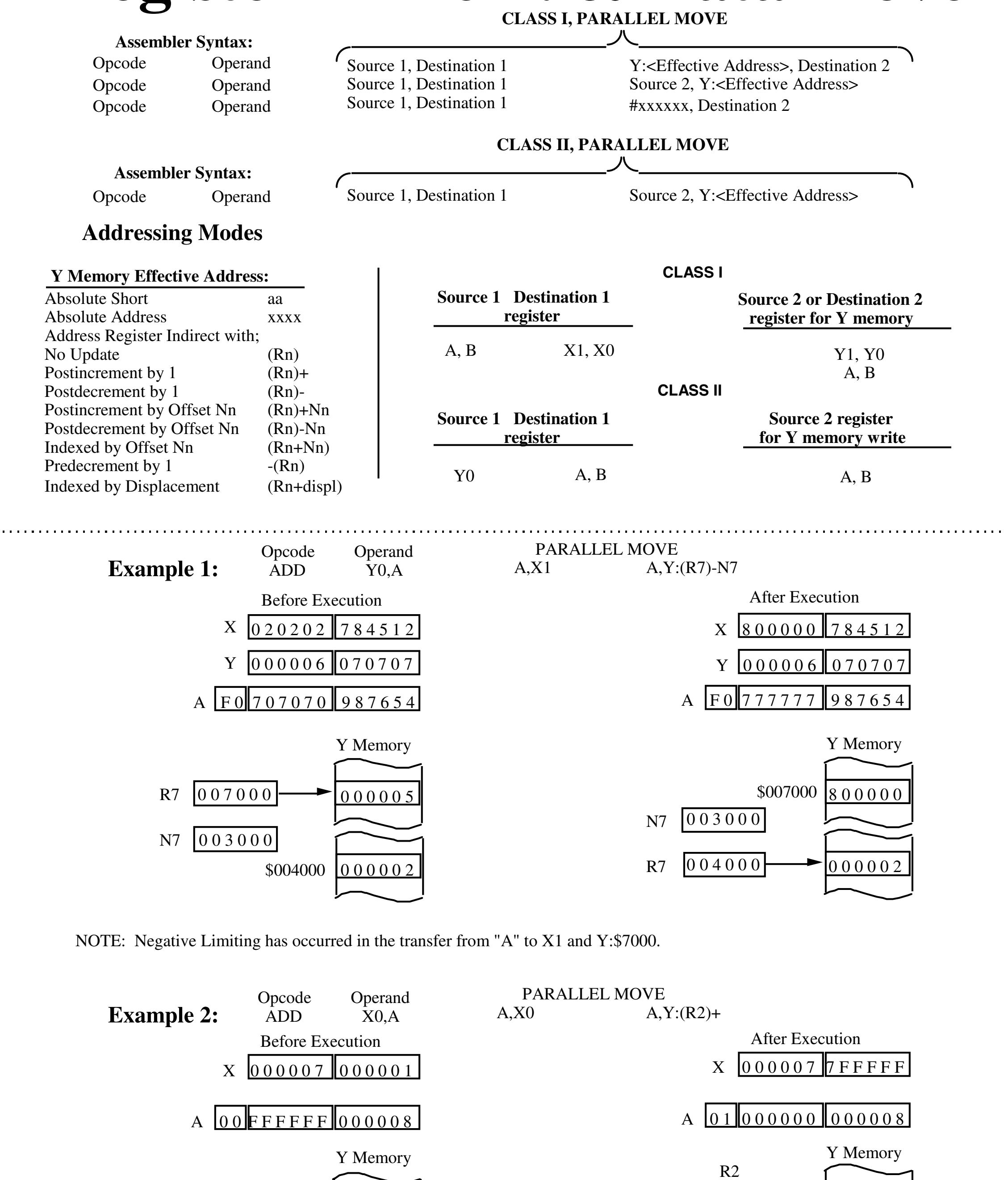
Y Memory

$00F233 ABABAB

$00F234 010101

Parallel Moves 3 - 11

Register Y Mem. & Data Move



NOTE: Positive Limiting has occurred in the transfer from "A" to Y:$913D.

Long Memory Data Move

Assembler Syntax:		PARALLEL MOVE
Opcode	Operand	L:<Effective Address>,Destination
Opcode	Operand	Source,L:<Effective Address>

Addressing Modes

L Effective Address:	
Absolute Short	aa
Absolute Address	xxxx
Address Register Indirect with;	
No Update	(Rn)
Postincrement by 1	(Rn)+
Postdecrement by 1	(Rn)-
Postincrement by Offset Nn	(Rn)+Nn
Postdecrement by Offset Nn	(Rn)-Nn
Indexed by Offset Nn	(Rn+Nn)
Predecrement by 1	-(Rn)
Indexed by Displacement	(Rn+displ)

Source or Destination	
X,Y	One 48 Bit Move
A,A10	One 48 Bit Move
B,B10	One 48 Bit Move
AB,BA	Two 24 Bit Moves

- The Only Parallel Move that References A10, B10, X, Y, AB, and BA.
- Useful for:
  - Double Precision, 48 Bit Long Word Transfers.
  - Complex Number (Real:Imaginary), Two Word Transfers.

Example 1:

Opcode	Operand	PARALLEL MOVE
ADD	Y,B	B10, L:$3000

Before Execution

Y 505050 606060

B 00 030303 040404

X Memory: 123456 ◄ Absolute Location $3000 ► Y Memory: ABCDEF

After Execution

Y 505050 606060

B 00 535353 646464

X Memory: 030303 ◄ Absolute Location $3000 ► Y Memory: 040404

Example 2:

Opcode	Operand	PARALLEL MOVE
ADD	Y1,B	BA,L:(R3)

Before Execution

Y 505050 606060

A 00 010101 020202

B 00 030303 040404

X Memory: 000000 ◄ R3 002000 ► Y Memory: 000000

After Execution

Y 505050 606060

A 00 010101 020202

B 00 535353 040404

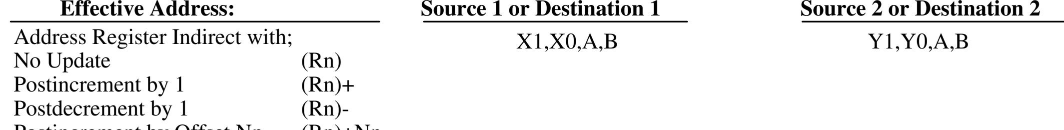
X Memory: 030303 ◄ R3 002000 ► Y Memory: 010101

XY Memory Data Move

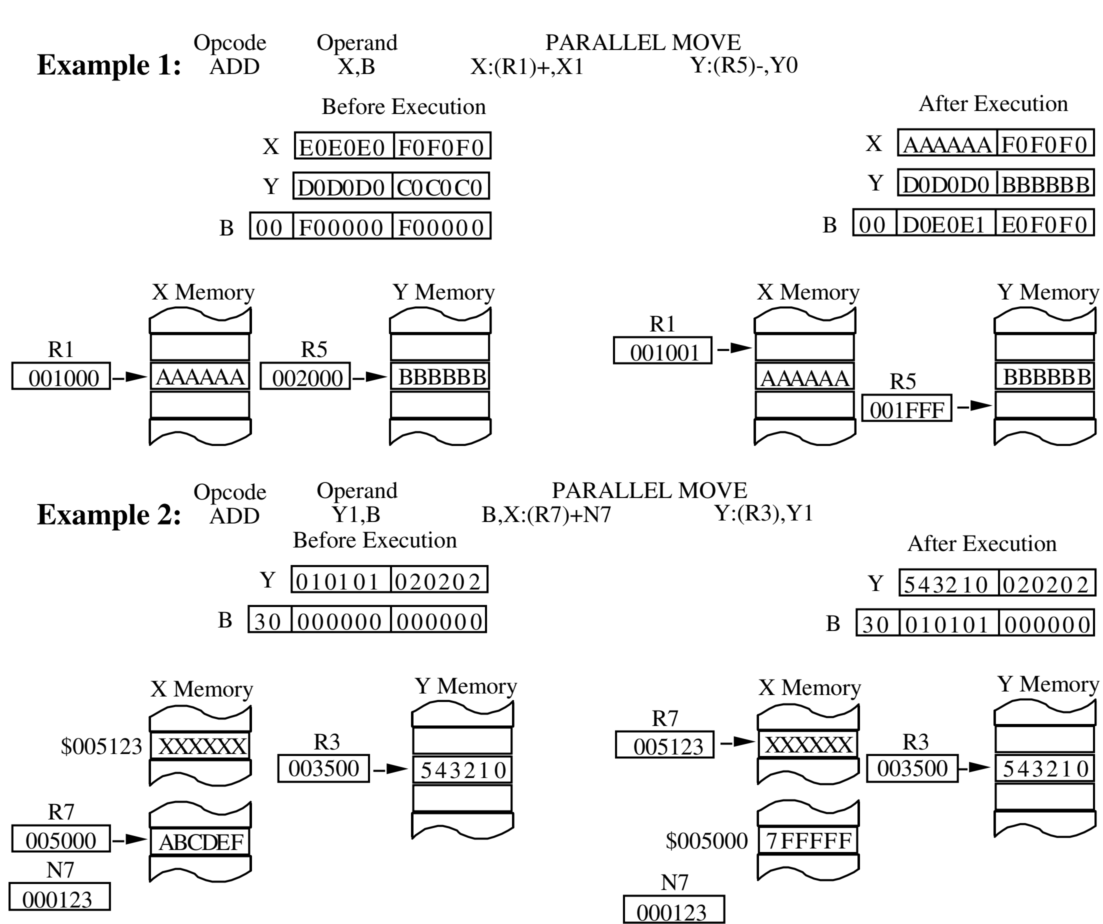
Assembler Syntax: **PARALLEL MOVE**

Opcode	Operand	X:<Effective Address>,Destination 1	Y:<Effective Address>,Destination 2
Opcode	Operand	X:<Effective Address>,Destination 1	Source 2,Y:<Effective Address>
Opcode	Operand	Source 1,X:<Effective Address>	Y:<Effective Address>,Destination 2
Opcode	Operand	Source 1,X:<Effective Address>	Source 2,Y:<Effective Address>

Addressing Modes



Effective Address:		Source 1 or Destination 1	Source 2 or Destination 2
Address Register Indirect with;		X1,X0,A,B	Y1,Y0,A,B
No Update	(Rn)		
Postincrement by 1	(Rn)+		
Postdecrement by 1	(Rn)-		
Postincrement by Offset Nn	(Rn)+Nn		



Parallel Move Summary

Mnemonic	Operation	Assembler Syntax
MOVE	Move Data Registers S → D	MOVE S,D

Operand Combinations

Source Destination
All Parallel Move Combinations

• Equivalent to a Data ALU NOP.

	Opcode/Operands	Parallel Move Examples	
Immediate Short Data	ADD X0,A	#$05,Y1	
Register to Register	ADD X0,A	A1,Y0	
Address Register Update	ADD X0,A	(R0)+N0	
X Memory	ADD X0,A	X0,X:(R3)+	
X Memory plus Register	ADD X0,A	X:(R4)–,X1	A,Y0
Y Memory	ADD X0,A	Y:(R6)+N6,X0	
Y Memory plus Register	ADD X0,A	A,X0	B,Y:(R0)
L Memory	ADD X0,A	L:-(R2),AB	
XY Memory	ADD X0,A	X1,X:(R1) +N1	Y1,Y:(R5)+N5

Parallel Move Exercises

1. Rewrite the following program fragment for maximum concurrency of operation:

Write your Solution below:

```
      clr a
      move x:(r0)+,x0
      move y:(r4)+,y0
      do #ntaps-1,endlp
      mac x0,y0,a
      move x:(r0)+,x0
      move y:(r4)+,y0
endlp macr x0,y0,a
      move (r0)-
```

2. Matching exercise. Some of the following instructions are illegal. Match the reason the instruction is illegal with the illegal instruction or indicate OK if the instruction is legal:

```
A. add x0,a   #$123456,a     a,y0            _____
B. add a,b    #$123456,x0    b,y1            _____
C. add a,b    x1,x:-(r7)     y:(r0),y1       _____
D. add a,b    x0,x:(r4)+n4   y0,y:(r7)+n7    _____
E. add a,b    y:(r3),y                       _____
F. add a,b    l:(r3)+,x                      _____
G. add a,b    -(r0)                          _____
```

a. Both address registers cannot be from the same bank.
b. Addressing mode not allowed.
c. Duplicate destinations are not allowed.
d. Not allowed as a destination for this form of parallel move.

3. Which statement(s) is/are true about this instruction?: `add a,b    #$123456,x0     b,y1`

A. The add operation completes first to allow the sum to be copied into y1.
B. The original value in b is copied into y1 while the add operation completes.
C. This is a one-word instruction.
D. This is a two-word instruction.

2.5 Instruction Set I

This section is a reproduction of Chapter 4 of the *DSP563xx/6xx Family Digital Signal Processor Training Notes*[4].

[4]Copyright of Motorola, Used with Permission.

Instruction Set I

Learn how to:

- Write program loops
- Write Subroutines
- Calculate a butterfly
- Implement a FIR Filter Algorithm
- Test and/or change bits in peripheral registers
- Find a maximum or minimum value in a data stream
- Pack/Unpack data to/from 8-bit peripheral
- Call a subroutine conditionally
- Change program flow conditionally
- Enter low power operation and wait for an interrupt
- Multiply double precision values
- Normalize a number

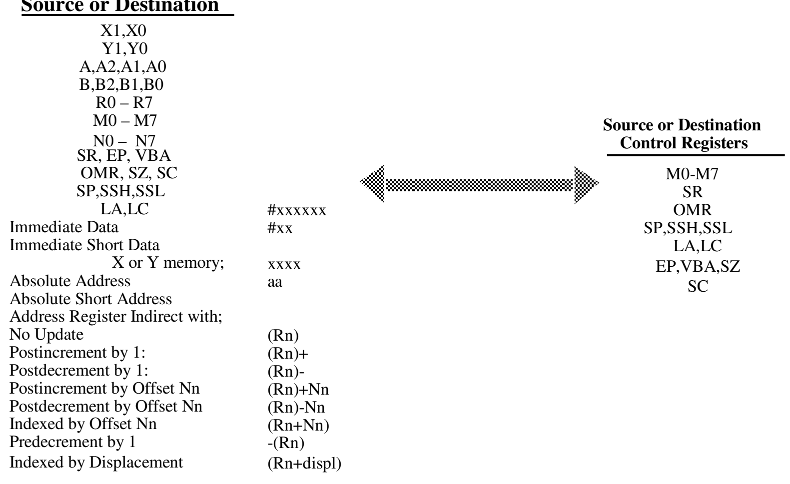
Table of Contents

4....Move Instructions (1 of 2)
5....Move Instructions (2 of 2)
6....Load Updated Address
7....Add & Subtract Instructions
8....Shift & Add or Sub. Instructions
9....Butterfly Calculation Exercise
10..Multiply Instructions
11..Multiply & Acc. Instructions
12..Arithmetic Shift Instructions
13..Compare Instructions
14..Transfer Instructions
15..Transfer Example
16..Arithmetic Instructions
17..Logical Instructions
18..Immediate Logical Instructions
19..Shift & Rotate Instructions
20..Repeat Next Instruction
21..Repeat Characteristics
22..Hardware DO Loops
23..End current DO Loop
24..Don't DO's
25..Program Control Instructions
26..FIR Filter Exercise
27..Bit Manipulation Instructions
28..Bit Field Program Control
29..Divide Iteration (1 of 2)
30..Divide Iteration (2 of 2)
31..Double Precision Multiply Mode
32..Miscellaneous Instructions
33..Example Programs

Move Instructions (1 of 2)

Mnemonic	Operation	Assembler Syntax
MOVEC	Move Data Registers S → D	MOVE(C) S,D

Operand Combinations

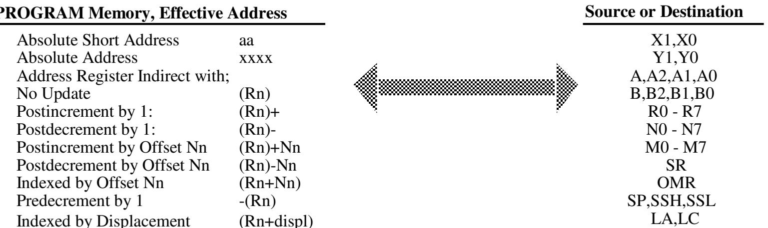


Example: MOVE SSL,LC

• Refer to Instruction Set Details for Restrictions.

Mnemonic	Operation	Assembler Syntax
MOVEM	Move Program Memory S → P:<ea> P:<ea> → D	MOVE(M) S,P:<ea> MOVE(M) P:<ea>,D

Operand Combinations

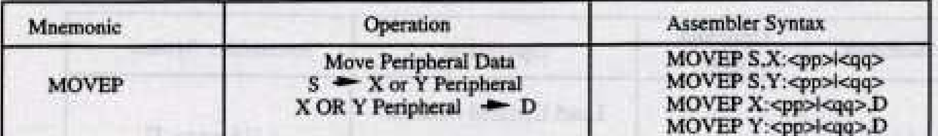


Example 1: MOVE M3, P:(R3)-N3

Example 2: MOVE P:$0, LC

• Refer to Instruction Set Details for Restrictions.

Move Instructions (2 of 2)

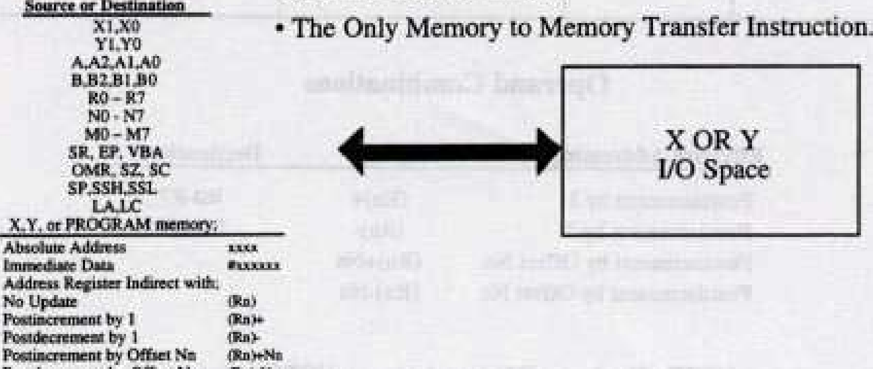


Mnemonic	Operation	Assembler Syntax

MOVEP	Move Peripheral Data
	S → X or Y Peripheral
	X OR Y Peripheral → D
		MOVEP S,X:<pp>I<qq>
		MOVEP S,Y:<pp>I<qq>
		MOVEP X:<pp>I<qq>,D
		MOVEP Y:<pp>I<qq>,D

Operand Combinations

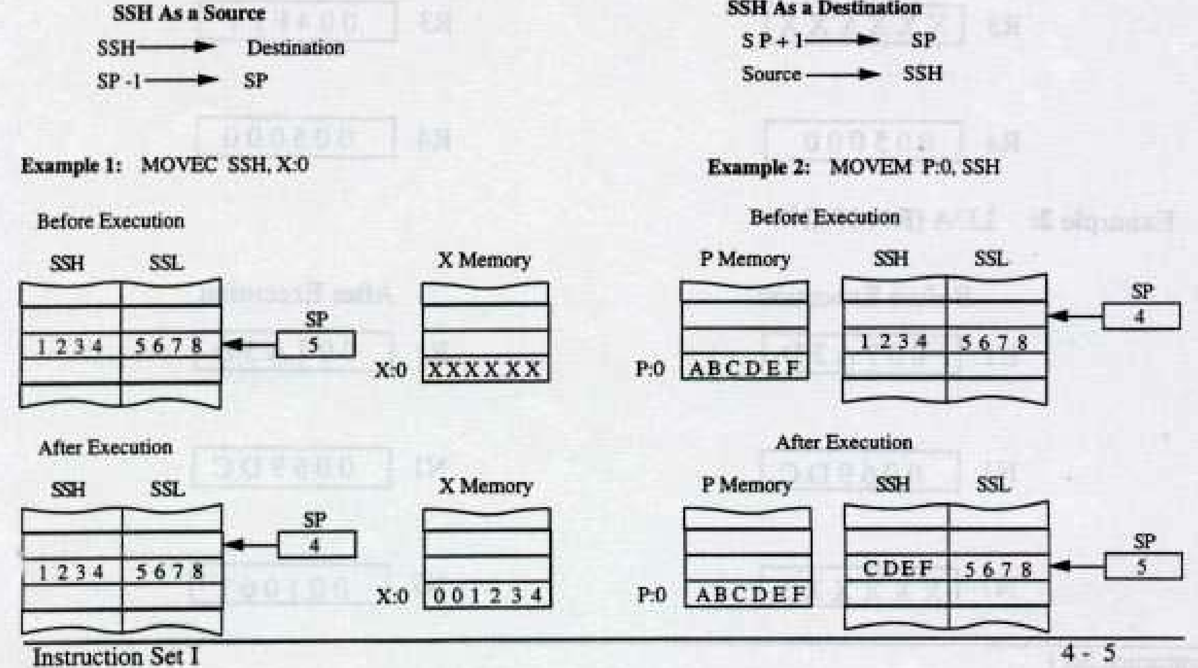
- The Only Memory to Memory Transfer Instruction.



Example 1: MOVEP X:(R3+N3), Y:$FFFFC0

Example 2: MOVEP X:$FFFFFF, A

MOVEC, MOVEM, & MOVEP with SSH:



Load Updated Address

Mnemonic	Operand Size	Operation	Assembler Syntax
LUA	24	Load Updated Address <ea> ► D	LUA <ea>, D

Operand Combinations

Effective Addressing Modes	<ea>	Destinations
Postincrement by 1	(Rn)+	R0-R7
Postdecrement by 1	(Rn)-	N0-N7
Postincrement by Offset Nn	(Rn)+Nn	
Postdecrement by Offset Nn	(Rn)-Nn	

NOTE: The Source Effective Address is NOT Updated.

Example 1: LUA (R4)-,R3

	Before Execution		After Execution
R3	X X X X X X	R3	0 0 4 F F F
R4	0 0 5 0 0 0	R4	0 0 5 0 0 0

Example 2: LUA (R1)-N1,N7

	Before Execution		After Execution
R1	0 0 7 A 3 D	R1	0 0 7 A 3 D
N1	0 0 6 9 D C	N1	0 0 6 9 D C
N7	X X X X X X	N7	0 0 1 0 6 1

Add & Subtract Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
ADD	24,48,56	Addition S + D ➡ D	ADD S,D [Parallel Move]
SUB	24,48,56	Subtraction D - S ➡ D	SUB S,D [Parallel Move]

Operand Combinations

A,B
B,A
X,A or B
X0,A or B
X1,A or B

Y, A or B
Y0,A or B
Y1,A or B

NOTE: Adding or Subtracting an Accumulator from itself is NOT Allowed.

Mnemonic	Operand Size	Operation	Assembler Syntax
ADC	48,56	Addition S + D + C ➡ D	ADC S,D [Parallel Move]
SBC	48,56	Subtraction D - S - C ➡ D	SBC S,D [Parallel Move]

Operand Combinations

X,A
Y,A

X,B
Y,B

Example 1: SUB X,B [Optional Parallel Move]

Before Execution

X | 7FFFFF | FFFFFF |

B | FF | FFFFFF | FFFFFF |

After Execution

X | 7FFFFF | FFFFFF |

B | FF | 800000 | 000000 |

Example 2: ADC Y,A [Optional Parallel Move]

Before Execution

Carry | 1 |

Y | 7FFFFF | FFFFFF |

A | 0F | 800000 | 000000 |

After Execution

Carry | 0 |

Y | 7FFFFF | FFFFFF |

A | 10 | 000000 | 000000 |

Shift & Add or Sub. Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
ADDR	56	Arithmetic Shift Right and Add Accumulators S + D/2 ► D	ADDR S,D [Parallel Move]
SUBR	56	Arithmetic Shift Right and Subtract Accumulators D/2 - S ► D	SUBR S,D [Parallel Move]
ADDL	56	Shift Left and Add Accumulators S + 2 x D ► D	ADDL S,D [Parallel Move]
SUBL	56	Shift Left and Subtract Accumulators 2 x D - S ► D	SUBL S,D [Parallel Move]

Operand Combinations

Source	Destination
A	B
B	A

- The SHIFT Occurs Prior to the Arithmetic Operation.
- These Instructions are useful for Efficient Divide and Decimation in Time (DIT) Fast Fourier Transform (FFT) Algorithms.

Example 1: ADDR A,B [Optional Parallel Move]

	Before Execution			After Execution		
A	00	13579B	000000	00	13579B	000000
B	C0	000000	2468AC	E0	13579B	123456

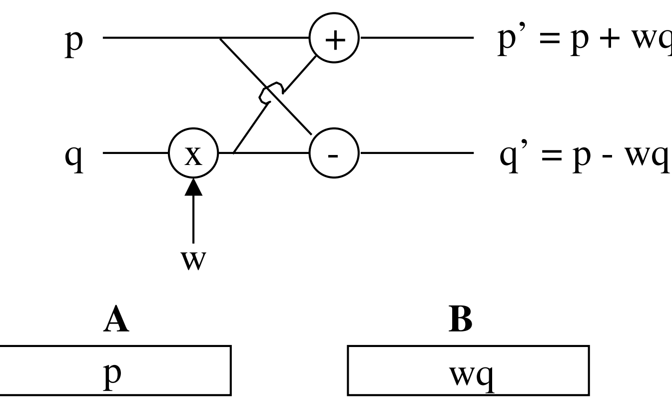
Example 2: SUBL A,B [Optional Parallel Move]

	Before Execution			After Execution		
A	00	004000	000000	00	004000	000000
B	00	005000	000001	00	006000	000002

Butterfly Calculation
Exercise

Using arithmetic and the "shift and add" or "shift and subtract" instructions, calculate a butterfly (for FFT). Accumulator A already has been loaded with "p" and accumulator B already has "wq". Afterward, accumulator A should contain "p - wq" and accumulator B should contain "p + wq".

Butterfly for FFT



Put your code here:

Suggested program steps:

1. A = p - wq

2. B = p + wq (2wq + (p-wq) = p + wq)

Multiply Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
MPY	24, 56	Signed Multiply ± (S1 * S2) ►D	MPY (±) S1, S2, D [Parallel Move]
MPYR	24, 56	Signed Multiply and Round ± (S1 * S2) + r ►D	MPYR (±) S1, S2, D [Parallel Move]

Operand Combinations

Source 1	Source 2	Destination
X0	X0	A or B
Y0	Y0	
X1	X0	
Y1	Y0	
X0	Y1	
Y0	X0	
X1	Y0	
Y1	X1	

Only these combinations of Source1 and Source2 are valid.
X1 $*$ X1 and Y1 $*$ Y1 are not valid.

Example 1: MPY -X0,X0,A [Optional Parallel Move]

Before Execution

X | A B C D E F | 4 0 0 0 0 0 | X0 = 1/2

A | 1 2 | 3 4 5 6 7 8 | 9 A B C D E |

After Execution

X | A B C D E F | 4 0 0 0 0 0 |

A | F F | E 0 0 0 0 0 | 0 0 0 0 0 0 | A = -1/4

Example 2: MPY Y1, Y0, B [Optional Parallel Move]

Before Execution

Y | 4 0 0 0 0 0 | 0 0 0 0 0 7 |

Y1 = 1/2

B | 1 2 | 3 4 5 6 7 8 | A B C D E F |

After Execution

Y | 4 0 0 0 0 0 | 0 0 0 0 0 7 |

B | 0 0 | 0 0 0 0 0 3 | 8 0 0 0 0 0 |

Example 3: MPYR Y1, Y0, B [Optional Parallel Move]

Before Execution

Y | 4 0 0 0 0 0 | 0 0 0 0 0 7 |

Y1 = 1/2

B | 1 2 | 3 4 5 6 7 8 | A B C D E F |

After Execution

Y | 4 0 0 0 0 0 | 0 0 0 0 0 7 |

B | 0 0 | 0 0 0 0 0 4 | 0 0 0 0 0 0 |

Multiply & Acc. Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
MAC	24, 56	D ± (S1 * S2) → D	MAC ± S1, S2, D [Parallel Move]
MACR	24, 56	D ± (S1 * S2) + r → D	MACR ± S1, S2, D [Parallel Move]

Operand Combinations

Source 1	Source 2	Destination
X0	X0	A or B
Y0	Y0	
X1	X0	
Y1	Y0	
X0	Y1	
Y0	X0	
X1	Y0	
Y1	X1	

- Only These Combinations of Source 1 and Source 2 are valid.
  X1 * X1 and Y1 * Y1 are not valid.

- These Instructions are the Basis for All Digital Filtering Algorithms!

Example 1: MAC X1, Y1, A [Optional Parallel Move]

Before Execution

X		0 0 0 0 0 2	0 0 0 0 0 0
Y		0 0 0 0 0 3	0 0 0 0 0 0
A	0 0	0 0 0 0 0 4	0 0 0 0 0 0

After Execution

X		0 0 0 0 0 2	0 0 0 0 0 0
Y		0 0 0 0 0 3	0 0 0 0 0 0
A	0 0	0 0 0 0 0 4	0 0 0 0 0 C

Example 2: MACR X0, Y0, B [Optional Parallel Move]

Before Execution

X		2 0 0 0 0 0	4 0 0 0 0 0
			X0 = 1/2
Y		0 1 0 0 0 0	0 0 0 0 0 3
B	0 0	0 0 0 0 0 4	0 0 0 0 0 0

After Execution

X		2 0 0 0 0 0	4 0 0 0 0 0
Y		0 1 0 0 0 0	0 0 0 0 0 3
B	0 0	0 0 0 0 0 6	0 0 0 0 0 0

Arithmetic Shift Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
ASR	56	Arithmetic Shift Right (55, 47, 23, 0 → C)	ASR D [Parallel Move]
ASL	56	Arithmetic Shift Left (C ← 55, 47, 23, 0 ← 0)	ASL D [Parallel Move]

Destination

A,B

Example 1: ASR A

Before Execution: A | 8 8 | 8 8 8 8 8 8 | 1 1 1 1 1 1 | Carry 0

After Execution: A | C 4 | 4 4 4 4 4 4 | 0 8 8 8 8 8 | Carry 1

Example 2: ASL B

Before Execution: Carry 0 B | C 0 | F 1 1 1 1 1 | 8 8 8 8 8 8 |

After Execution: Carry 1 B | 8 1 | E 2 2 2 2 3 | 1 1 1 1 1 0 |

Compare Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
CMP	24, 56	Compare D - S	CMP S,D [Parallel Move]
CMPM	24, 56	Compare Magnitude |D| - |S|	CMPM S,D [Parallel Move]
TST	56	Test Accumulator D - 0	TST D [Parallel Move]

Operand Combinations

Source	Destination
A	B
B	A
X1,X0	A,B
Y1,Y0	A,B

• The result from the subtract is not stored, the condition code register is updated.

Instruction Syntax:

Opcode Operand — Condition Code Register

Parallel Move — Limit Bit

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C

CCR

L - Set if limiting (parallel move) or overflow has occured in result (Latched Bit)

E - Set if the signed integer portion of result is in use

U - Set if result is unnormalized

N - Set if bit 55 of result is set

Z - Set if result equals zero

V - Set if overflow has occured in result

C - Set if a carry (or borrow) occurs from bit 55 of result

• The definition of the E and U bits varies according to the scaling mode being used.

Transfer Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
TFR	24, 56	Transfer Data ALU Register S► D	TFR S,D [Parallel Move]
Tcc	24, 56	Conditionally Transfer Data ALU Registers If cc Then S1 ►D1 or If cc Then S1 ► D1 and S2 ►D2	Tcc S1,D1 [S2,D2]

Operand Combinations

Source1	Destination1	Source 2	Destination 2
A	B	R0-R7	R0-R7
B	A		
X1,X0	A,B		
Y1,Y0	A,B		

- Performs transfer without shifting or limiting.
- Uses internal ALU data paths.
- Does not affect Condition Code bits.

Tcc Test Conditions

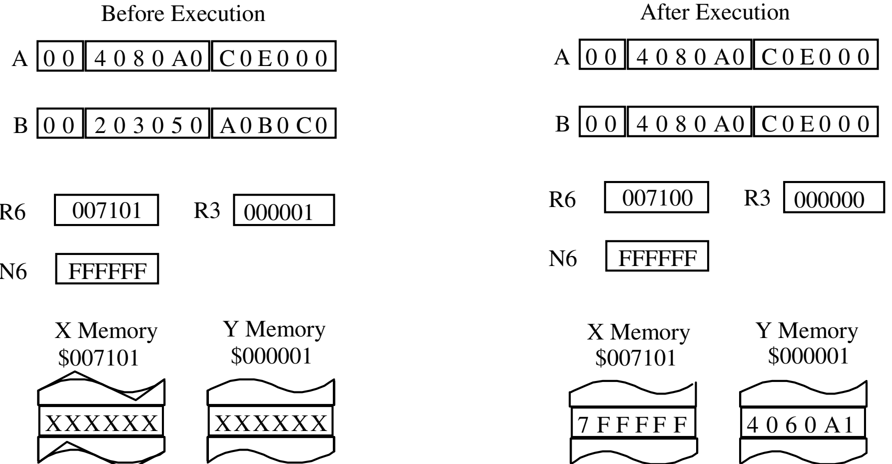
	"cc" Mnemonic	Condition
CC (HS) -	carry clear (higher or same)	C=0
CS (LO) -	carry set (lower)	C=1
EC -	extension clear	E=0
EQ -	equal	Z=1
ES -	extension set	E=1
GE -	greater than or equal	$N \oplus V = 0$
GT -	greater than	$Z+(N \oplus V) = 0$
LC -	limit clear	L=0
LE -	less than or equal	$Z+(N \oplus V) = 1$
LS -	limit set	L=1
LT -	less than	$N \oplus V = 1$
MI -	minus	N= 1
NE -	not equal	Z= 0
NR -	normalized	$Z+(\bar{U} \cdot \bar{E}) = 1$
PL -	plus	N= 0
NN -	not normalized	$Z+(\bar{U} \cdot \bar{E}) = 0$

Transfer Example

Example: TFR A, B A, X: (R6) + N6 B, Y:(R3)-

Status Register (bits 15 … 0): LF 0 T 0 1 0 I1 I0 | 0 L E U N Z V C

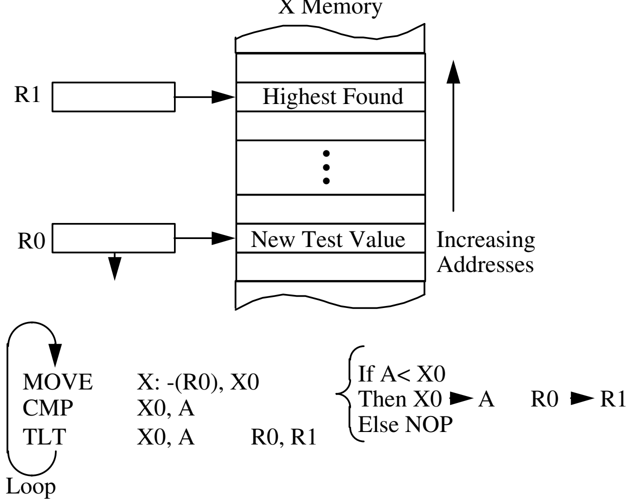
S1, S0 – Scale Up



- Limiting has occurred on the transfer to X Memory.
- TFR using the Parallel MOVE option creates a three Move Instruction.

- Tcc used after CMP or CMPM can perform many useful functions like selectively transfer.
  - Maximum Value
  - Minimum Value
  - Maximum Absolute Value
  - Minimum Absolute Value

Example: Scan Memory to find the largest value in an array.



Arithmetic Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
NEG	56	Negate 0 - D ► D	NEG D [Parallel Move]
NORM	24, 56	Normalize Accumulator Iteration If $\bar{E} \bullet U \bullet \bar{Z} = 1$, then ASL D and Rn - 1 ► Rn Else If E = 1, then ASR D and Rn + 1 ► Rn Else NOP	NORM Rn,D Parallel Move not allowed
RND	56	Round Accumulator D + r ► D	RND D [Parallel Move]
ABS	56	Absolute Value |D| ► D	ABS D [Parallel Move]
CLR	56	Clear Accumulator 0 ► D	CLR D [Parallel Move]

Operand Combinations:

Destination	Register Rn
A,B	R0-R7

Example 1: NEG A [Optional Parallel Move]

Before Execution

A | 0 0 | 0 0 1 2 F 0 | 0 0 0 0 0 0 |

After Execution

A | F F | F F E D 1 0 | 0 0 0 0 0 0 |

Example 2: Normalize Accumulator A.

	R0	A
	0 0 0 0 0 0	0 0 | 1 0 0 0 0 0 | 0 0 1 2 4 8
NORM R0, A		
	F F F F F F	0 0 | 2 0 0 0 0 0 | 0 0 2 4 9 0
NORM R0, A		
	F F F F F E	0 0 | 4 0 0 0 0 0 | 0 0 4 9 2 0
NORM R0, A		
	F F F F F E	0 0 | 4 0 0 0 0 0 | 0 0 4 9 2 0

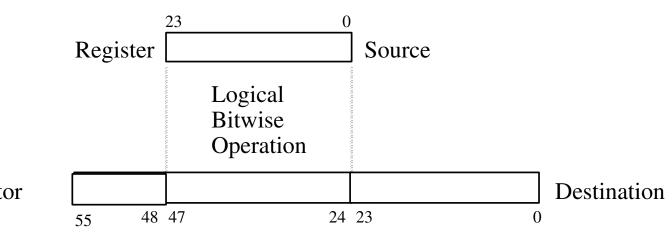
- R0 Contains the Number of Bits shifted to Normalize the Accumulator
  - If R0<0 then the Shift was to the left.
  - If R0>0 then the Shift was to the right.
- Only does one bit of normalization at a time.
  Maximum to normalization is 47.

Logical Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
AND	24	Logical AND S • D [47:24] ► D [47:24]	AND S, D [Parallel Move]
OR	24	Logical Inclusive OR S + D [47:24] ► D [47:24]	OR S, D [Parallel Move]
EOR	24	Logical Exclusive OR S ⊕ D [47:24] ► D [47:24]	EOR S, D [Parallel Move]
NOT	24	Logical Complement $\overline{D}$ [47:24] ► D [47:24]	NOT D [Parallel Move]

Operand Combinations

Source	Destination
X1, X0 Y1, Y0	A, B



Example 1: AND Y0, B

Before Execution

Y | 1 2 3 4 5 6 | F 8 4 2 1 0 |

B | F F | 0 F F F F F | F F F F F F |

After Execution

Y | 1 2 3 4 5 6 | F 8 4 2 1 0 |

B | F F | 0 8 4 2 1 0 | F F F F F F |

Example 2: EOR X1, A

Before Execution

X | F 8 4 2 1 0 | 1 2 3 4 5 6 |

A | F F | F F F F F F | F F F F F F |

After Execution

X | F 8 4 2 1 0 | 1 2 3 4 5 6 |

A | F F | 0 7 B D E F | F F F F F F |

Immediate Logical Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
ANDI	8	AND Immediate with Control Register #xx • D ➤ D	AND(I) #xx, D
ORI	8	OR Immediate with Control Register #xx + D ➤ D	OR(I) #xx, D

Operands
Destinations

MR
CCR
COM
EOM

- Does not support parallel moves.
- If destination is the MR or OMR, then the CCR is not affected.
- If the CCR is the destination, then bits 0 through 6 are set or cleared according to the operation.

Example 1: ANDI #%11111100, COM

- Puts processor in Mode 0
- Does not change other bits

Example 2: ORI #%00000010, COM

- Sets MB Bit, all others are unchanged

Example 3: ORI #%00001111, CCR

- Sets N, Z, V, and C bits in the CCR.
- The L, E, and U bits are unchanged.

Shift & Rotate Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
LSL	24	Logical Shift Accumulator Left (C ← [47 ← 24] ← 0)	LSL D [Parallel Move]
ROL	24	Rotate Accumulator Left (C ← [47 ← 24] ← C)	ROL D [Parallel Move]
LSR	24	Logical Shift Accumulator Right (0 → [47 → 24] → C)	LSR D [Parallel Move]
ROR	24	Rotate Accumulator Right (C → [47 → 24] → C)	ROR D [Parallel Move]

Operands Destination
A, B

Example 1: LSR A

Before Execution

A	8 0	8 2 2 2 2 2	1 1 1 1 1 1

Carry 1

After Execution

A	8 0	4 1 1 1 1 1	1 1 1 1 1 1

Carry 0

Example 2: ROL B

Before Execution

B	F 0	0 0 0 0 0 E	F 0 0 0 0 F

Carry 1

After Execution

B	F 0	0 0 0 0 1 D	F 0 0 0 0 F

Carry 0

Repeat Next Instruction

Mnemonic	Operation	Assembler Syntax
REP	LC → TEMP; #xxx → LC Repeat Next Instruction Until LC = 1 TEMP → LC	REP #xxx REP X:<ea> REP Y:<ea> REP S

#xxx Static Loop Count

Immediate Short Data #xxx
(12 Bits, Zero Extended)

X or Y Dynamic Loop Count Effective Address MODES:

Absolute Short Address	aa
Address Register Indirect with;	
No Update	(Rn)
Postincrement by 1	(Rn)+
Postdecrement by 1	(Rn)-
Postincrement by Offset Nn	(Rn)+Nn
Postdecrement by Offset Nn	(Rn)-Nn
Indexed by Offset Nn	(Rn+Nn)
Predecrement by 1	-(Rn)

S Dynamic Loop Count Operands:

X1,X0
Y1,Y0
A,A2,A1,A0
B,B2,B1,B0
SR, EP, VBA
OMR, SC, SZ
SP,SSH,SSL
LA,LC
R0-R7
N0-N7
M0-M7

Example 1:

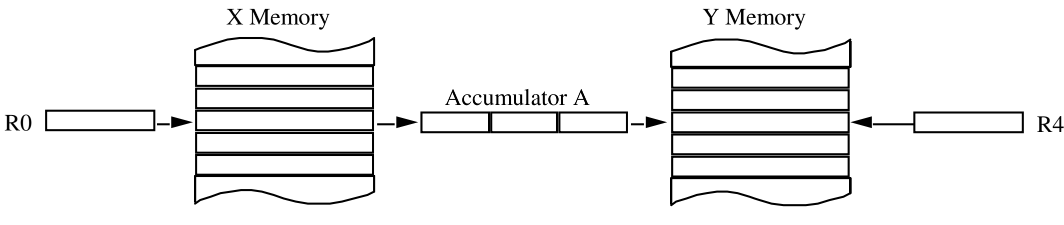
```
REP   X0                              ; Repeat the number of times in X0
MAC   X1,Y1,A  X:(R1)+,X1  Y:(R4)+,Y1 ; X1*Y1+A → A, update X1,Y1
```

Example 2: Multiple Bit Shifts

```
REP #N
ASR   A                ; N is the number of right shifts
```

Example 3: Memory to Memory Block Move

```
REP #N                     ; N is the size of the block moved.
MOVE  X:(R0)+,A    A, Y:(R4)+ ;
```



• One I_{cyc} for each 24 bit Transfer

Repeat Characteristics

- Only single word instructions are repeatable.
- REP and the following instruction are fetched once and stored internally.
- REP is NOT Interruptable!
- LC = 0 skips the following instruction (behaves like NOP).
- If LC = SSH, then SSH is read and the SP is postdecremented.
- REP cannot be the last instruction of a DO loop. i.e. at LA
- The following instructions are NOT Repeatable:

DO
Jcc
JCLR
JMP
JSET
JScc
JSCLR
JSR
JSSET
REP
RTI
RTS
STOP
SWI
WAIT

Hardware DO Loops

Mnemonic	Operation	Assembler Syntax
DO	**Start of Loop** SP+1 ► SP; LA ► SSH; LC ► SSL; #xxx ► LC SP+1 ► SP; PC ► SSH; SR ► SSL; Expr-1 ► LA 1 ► LF **End of Loop** If LC°1 then LC-1 ► LC; SSH ► PC Else SSL(LF) ► SR; SP-1 ► SP SSH ► LA; SSL ► LC; SP-1 ► SP	**Static** DO #xxx, Expr **Dynamic** DO X: <ea>,Expr DO Y: <ea>, Expr DO S, Expr

#xxx Static Loop Count	X or Y Dynamic Loop Count Effective Address MODES:		S Dynamic Loop Count Operands:
Immediate Short Data #xxx	Absolute Short Address	aa	X1, X0
(12 bits, Zero Extended)	Address Register Indirect with;		Y1, Y0
	No Update	(Rn)	A,A2,A1,A0
	Postincrement by 1	(Rn)+	B,B2,B1,B0
	Postdecrement by 1	(Rn)-	SR, EP, VBA
	Postincrement by Offset Nn	(Rn)+Nn	OMR, SC, SZ
	Postdecrement by Offset Nn	(Rn)-Nn	SP,SSL
	Indexed by Offset Nn	(Rn+Nn)	LA,LC
	Predecrement by 1	-(Rn)	R0-R7
	Indexed by Displacement	(Rn+displ)	N0-N7
			M0-M7

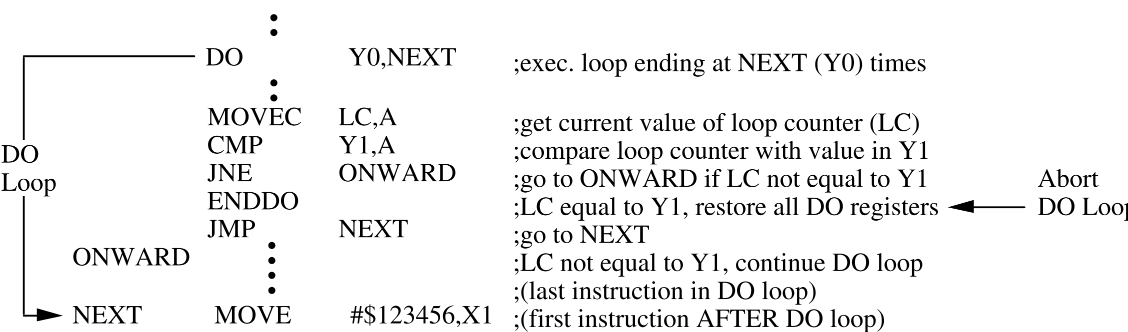
DO LOOP EXAMPLE

```
        DO      #35,END1          ;BEGIN LOOP 1
                   :
        DO      X:(R1)+,END2      ;BEGIN LOOP 2
                   :
        DO      Y:$1EA,END3       ;BEGIN LOOP 3
                   :
        DO      B1,END4           ;BEGIN LOOP4
                   :
        MAC     X0,Y0,A           ;LAST INSTRUCTION IN LOOP 4
END4               :              ;FIRST INSTRUCTION AFTER LOOP 4
        ADD     B,A  X:(R1)+,X0   ;LAST INSTRUCTION IN LOOP 3
END3               :              ;FIRST INSTRUCTION AFTER LOOP 3
        MOVE    X:(R7)+N7,A       ;LAST INSTRUCTION IN LOOP 2
END2               :              ;FIRST INSTRUCTION AFTER LOOP 2
        MOVE    B,Y:-(R0)         ;LAST INSTRUCTON IN LOOP1
END1                              ;FIRST INSTRUCTION AFTER LOOP 1
```

End current DO Loop

Mnemonic	Operation	Assembler Syntax
ENDDO	SSL(LF) → SR; SP-1 → SP SSH → LA; → SSL → LC, SP-1 → SP	ENDDO

END DO EXAMPLE



Don't DO's

Immediately before DO:
MOVEC to LA, LC, SSH, SSL OR SP
MOVEM to LA, LC, SSH, SSL, or SP
MOVEP to LA, LC, SSH, SSL or SP
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH

At LA-2, LA-1 and LA:
DO
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH
MOVEC to LA, LC, SR, SP, SSH or SSL
MOVEM to LA, LC, SR, SP, SSH or SSL
MOVEP to LA, LC, SR, SP, SSH or SSL
ANDI MR
ORI MR
any two-word instruction which reads LC, SP, or SSL

At LA-1:
any one-word instruction (except REP) which reads LC, SP, or SSL
JCLR, JSET, two-word JMP, or two-word Jcc

At LA:
any two-word instruction*
Jcc
JCLR
JMP
JScc
JSET
JSR
REP
RESET
RTI
RTS
STOP
WAIT

Other Restrictions:
DO SSH,xxxx
JSR to (LA) whenever the Loop Flag (LF)is set
JScc to (LA) whenever the Loop Flag (LF) is set
JSCLR to (LA) whenever the Loop Flag (LF) is set
JSSET to (LA) whenever the Loop Flag (LF) is set
A DO instruction cannot be repeated using the REP instruction.

* This restriction applies to the situation in which the DSP56300 Simulator's single line assembler is used to change the last instruction in a DO loop from a one-word instruction to a two-word instruction.

Program Control Instructions

Mnemonic	Operation	Assembler Syntax
JMP	Jump <ea>► PC	JMP <ea>
Jcc	Conditional Jump If cc then <ea>► PC Else PC+1► PC	Jcc <ea>
JSR	Jump to Subroutine SP + 1► SP PC► SSH SR► SSL <ea>► PC	JSR <ea>

JScc	Conditional Jump to Subroutine
	If cc then
	SP + 1► SP
	PC► SSH
	SR► SSL
	<ea>► PC
	Else
	PC + 1► PC
		JScc <ea>

RTS	Return From Subroutine SSH► PC SP - 1► SP	RTS
RTI	Return From Interrupt SSH► PC SSL► SR SP - 1 ► SP	RTI

Jump Effective Address MODES:

Absolute Address	xxxx
Short Jump Address	xxx
Address Register Indirect with;	
No Update	(Rn)
Postincrement by 1	(Rn)+
Postdecrement by 1	(Rn)-
Postincrement by Offset Nn	(Rn)+Nn
Postdecrement by Offset Nn	(Rn)-Nn
Indexed by Offset Nn	(Rn+Nn)
Predecrement by 1	-(Rn)
Indexed by Displacement	(Rn+displ)

Jump Conditions

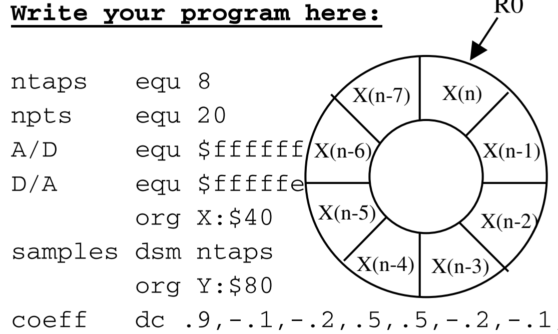
"cc" Mnemonic		Condition
CC (HS)	- carry clear (higher or same)	C=0
CS (LO)	- carry set (lower)	C=1
EC	- extension clear	E=0
EQ	- equal	Z=1
ES	- extension set	E=1
GE	- greater than or equal	N ⊕ V=0
GT	- greater than	Z+(N ⊕V)=0
LC	- limit clear	L=0
LE	- less than or equal	Z+(N ⊕V)=1
LS	- limit set	L=1
LT	- less than	N ⊕ V=1
MI	- minus	N=1
NE	- not equal	Z=0
NR	- normalized	Z+(Ū • Ē)=1
PL	- plus	N=0
NN	- not normalized	Z+(Ū • Ē)=0

FIR Filter

Exercise

Write code to initialize R0 to point at samples and R4 to point at coefficients using modulo addressing, and generate 20 output points of an 8 tap FIR filter. The 8 samples are stored in RAM beginning at X:$40 and the 8 coefficients are stored in RAM beginning at Y:$80.

FIR equation: $Y(n) = \sum_{i=0}^{N-1} C(i)\,X(n-i)$



Write your program here:

```
ntaps    equ 8
npts     equ 20
A/D      equ $ffffff
D/A      equ $fffffe
         org X:$40
samples  dsm ntaps
         org Y:$80
coeff    dc .9,-.1,-.2,.5,.5,-.2,-.1,.9

       org P:$100
```

Suggested program steps:

Number of taps in filter
Number of points to process
Address of input device (A/D)
Address of output device (D/A)
Originate sample table at X:$40
Define storage for modulo 8 table
Originate coefficient table at Y:$80
Define constant coefficients

Originate program at P:$100

1. Initialize r0 and r4
2. Initialize M registers for Modulo 8
3. Do following instructions 20 times.
4. Move new data sample from A/D (input device) in Y memory to X0
5. Clear accum A, save new sample, and load Y0 for first multiplication.
6. Do 7 multiply and accumulates, post incrementing r0 and r4.
7. Do final multiply and accumulate with rounding and adjust r0 to the location of the oldest sample.
8. Output sample to D/A in Y memory

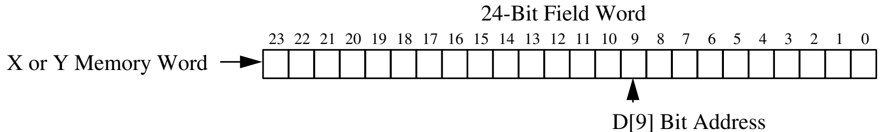
Bit Manipulation Instructions

Mnemonic	Operand Size	Operation	Assembler Syntax
BCLR *	24	Bit Test and Clear ** D[n] → C 0 → D[n]	BCLR #n, D
BSET *	24	Bit Test and Set ** D[n] → C 1 → D[n]	BSET #n,D
BCHG *	24	Bit Test and Change ** D[n] → C $\overline{D[n]}$ → D[n]	BCHG #n,D
BTST	24	Bit Test ** D[n] → C	BTST #n,D

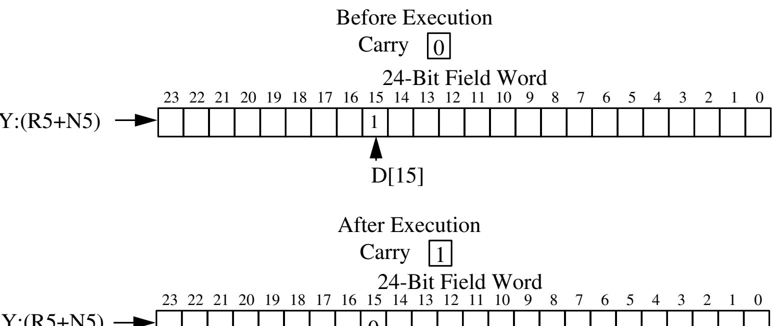
Destination Operands:
All Data ALU, AGU, and Program Controller Registers
or the following X or Y Effective Address MODES

Absolute Address	xxxx
Absolute Short Address	aa
I/O Short Address	pp
Address Register Indirect with;	
No Update	(Rn)
Postincrement by 1	(Rn)+
Postdecrement by 1	(Rn)-
Postincrement by Offset Nn	(Rn)+Nn
Postdecrement by Offset Nn	(Rn)-Nn
Indexed by Offset Nn	(Rn+Nn)
Predecrement by 1	-(Rn)
Indexed by Displacement	(Rn+displ)

* Generates Indivisible Read-Modify-Write Bus Cycle for memory operand accesses
** D[n] is the nth bit of the 24 bit destination operand field. 0 Š n Š 23



Example: BCHG #15, Y:(R5+N5)



Bit Field Program Control

Mnemonic	Operation	Assembler Syntax
JSET	Jump if Bit Set If S[n] =1 Then xxxx ► PC Else PC+1 ► PC	JSET #n,D,xxxxxx
JCLR	Jump if Bit Clear If S[n] =0 Then xxxx ► PC Else PC+1 ► PC	JCLR #n,D,xxxxxx
JSSET	Jump to Subroutine if Bit Set If S[n] =1 Then SP+1 ► SP PC ► SSH, SR ► SSL xxxx ► PC Else PC+1 ► PC	JSSET #n,D,xxxxxx
JSCLR	Jump to Subroutine if Bit Clear If S[n] =0 Then SP+1 ► SP PC ► SSH, SR ► SSL xxxx ► PC Else PC+1 ► PC	JSCLR #n,D,xxxxxx

Destination Operands:
All Data ALU, AGU, and Program Controller Registers or the following X or Y Effective Address MODES

Absolute Short Address	aa
I/O Short Address	pp
Address Register Indirect with;	
No Update	(Rn)
Postincrement by 1	(Rn)+
Postdecrement by 1	(Rn)-
Postincrement by Offset Nn	(Rn)+Nn
Postdecrement by Offset Nn	(Rn)-Nn
Predecrement by 1	-(Rn)
Indexed by Offset Nn	(Rn+Nn)
Indexed by Displacement	(Rn+displ)

Note: Rn is always updated, independent of the results of the bit test (applies to (Rn)+, (Rn)-, (Rn)+Nn, (Rn)-Nn, -(Rn))

- Jump target is specified by the 24-bit absolute address extension

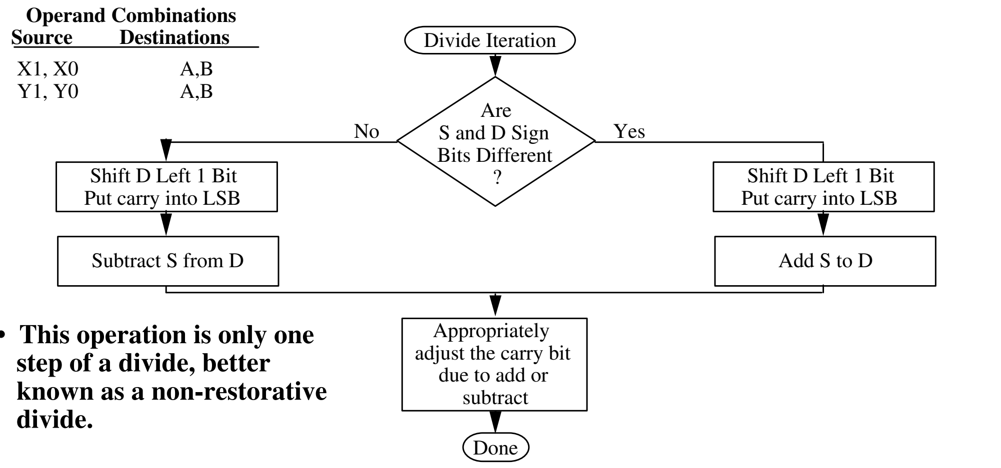
Software Polled I/O Service

Divide Iteration (1 of 2)

MNEMONIC	OPERAND SIZE	OPERATION	ASSEMBLER SYNTAX
DIV	24,56	Divide Iteration If D[55] ⊕ S[23] = 1, then (55 47 23 0) ← ← ← ← C + S ← D Destination Accumulator D else (55 47 23 0) ← ← ← ← C - S ← D Destination Accumulator D	DIV S,D

Operand Combinations

Source	Destinations
X1, X0	A,B
Y1, Y0	A,B



- **This operation is only one step of a divide, better known as a non-restorative divide.**

Example: 24-bit unsigned quotient, 48-bit unsigned remainder

```
AND #$FE,CCR        ;clear carry bit C (quotient sign bit)
REP #24             ;form a 24-bit quotient and remainder
DIV X0,A            ;form quotient in A0, remainder in A1
ADD X0,A            ;restore remainder in A1
```

Before Execution

X | X X X X X X | 6 0 0 0 0 0 | A | 0 0 | 4 0 0 0 0 0 | 0 0 0 0 0 0 |

X0 = 3/4 = 0.75 A = 1/2 = 0.5

After Execution

$$\frac{A}{X0} = \frac{\frac{1}{2}}{\frac{3}{4}} = \frac{2}{3} = 0.666666\ldots$$

X | X X X X X X | 6 0 0 0 0 0 | A | 0 0 | 4 0 0 0 0 0 (L.S.P. Remainder) | 5 5 5 5 5 5 (Quotient) |

$0.0 0 0 0 0 0 4 0 0 0 0 0

Quotient = $0.555555 = 0.666666626930237

Remainder = $0.000000400000 = 0.000000029802322

Divide Iteration (2 of 2)

Example: (24-bit signed quotient, 48-bit signed remainder) (Four-Quadrant)

```
          ABS A       A,B               ;make dividend positive, copy A1 to B1
          EOR X0,B B,X:$10              ;save rem. sign in X:$10, quo. sign in N
          AND #$FE,CCR                  ;clear carry bit C (quotient sign bit)
          REP #24                       ;form a 24-bit quotient
          DIV X0,A                      ;form quotient in A0, remainder in A1
          TFR A,B                       ;save quotient and remainder in B1,B0
          JPL SAVEQUO                   ;go to SAVEQUO if quotient is positive
          NEG B                         ;complement quotient if N bit set
SAVEQUO   TFR X0,B B0,X1                ;save quo. in X1, get signed divisor
          ABS B                         ;get absolute value of signed divisor
          ADD A,B                       ;restore remainder in B1
          JCLR #23,X:$10,DONE           ;go to DONE if remainder is positive
          MOVE #$0,B0                   ;clear LS 24 bits of B
          NEG B                         ;complement remainder if negative
DONE      .........
```

Before Execution

X		000000	123456 (S)
A	00	0E66D7	F2832C
		D (spans A1:A0)	
B	00	000000	000000

After Execution

X		654321 (Quotient)	123456
A	FF	EDCCAA	654321
B	00	000100 (L.S.P. Remainder)	654321

- The 48 bit dividend (destination) must be a positive fraction, sign extended to 56 bits.
- Valid results are obtained only when the operands are interpreted as fractions and |D| < |S|.
- Each DIV operation calculates one bit of quotient using a non restoring fractional division algorithm.
- To calculate a 24-bit quotient, DIV must be executed 24 times.
- In DSP, there is not a real need for a general purpose divide, other than dissemination in time.
- Instead, DSP is a series of summations, delays, and multiplies.

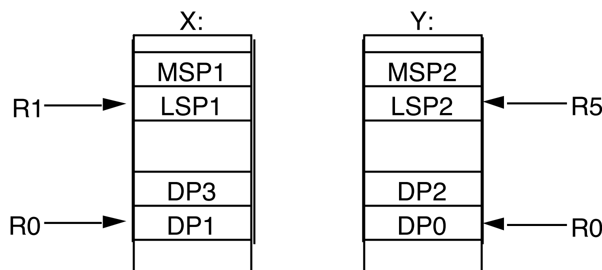
For more information

Theory and Application of Digital Signal Processing, by Rabner and Gold (Prentice-Hall, 1975) pp. 524-530.

Computer Architecture and Organization, by John Hayes, (McGraw Hill, 1978) pp. 190-199.

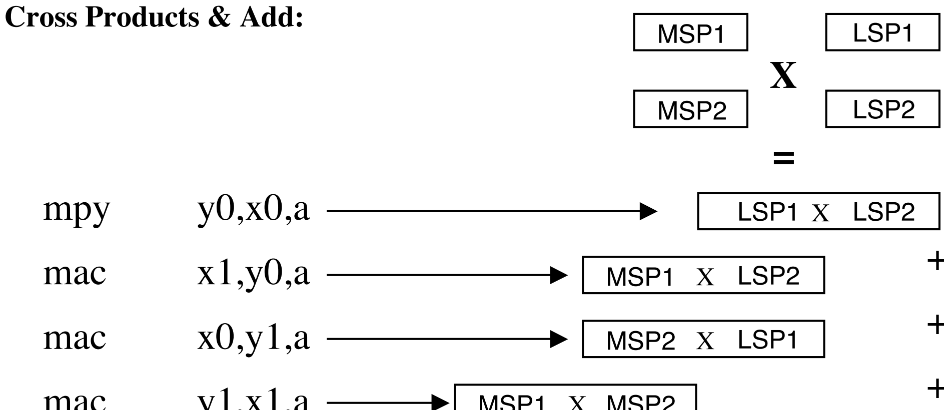
Computer Arithmetic: Principals, Architecture, and Design, by Kai Hwang
(John Wiley and Sons, 1979), pp. 213-223.

Double Precision Multiply Mode



DP3_DP2_DP1_DP0 = MSP1_LSP1 x MSP2_LSP2

Cross Products & Add:



Full Double Precision Multiply Algorithm:

Mode Register (bits 7 6 ... 0; bit 6 = DM)

```
ori      #$40,mr                              ;enter mode
move     x:(r1)+,x0      y:(r5)+,y0           ;load operands
mpy      y0,x0,a  x:(r1)+,x1  y:(r5)+,y1      ;LSP*LSP -> a
mac      x1,y0,a         a0,y:(r0)            ;shifted(a)+
                                              ;   MSP*LSP -> a
mac      x0,y1,a                              ;a+LSP*MSP -> a
mac      y1,x1,a         a0,x:(r0)+           ;shifted(a)+
                                              ;   MSP*MSP -> a
move     a,l:(r0)+
andi     #$bf,mr                              ;exit mode
non-Data ALU operation                        ;pipeline delay
```

Miscellaneous Instructions

Mnemonic	Operation	Assembler Syntax
NOP	No Operation PC+1 → PC	NOP
RESET	Reset on Chip Peripheral Devices 1 → I1 1 → I0 0 → IPR-C & IPR-P PC+1 → PC	RESET
WAIT	Disable Clocks to Processor Core and Enter the Wait Processing State	WAIT
STOP	Enter the Stop Processing State and Stop Clock to on-chip ckts	STOP
SWI	Software Interrupt Begin SWI Exception Processing	SWI

Mnemonic	Operand Size	Operation	Assembler Syntax
INC	56	Addition 1 + D → D	INC D
DEC	56	Subtraction D - 1 → D	DEC D

Destinations Operands

A or B

Operation: One is added or subtrated to/from the LSB of D.

Example : INC B

Before Execution

B | 00 | 7FFFFF | FFFFFF |

After Execution

B | 00 | 800000 | 000000 |

Mnemonic	Operand Size	Operation	Assembler Syntax
DEBUG	-	Enter debug mode & wait for OnCE commands	DEBUG
DEBUGcc	-	If cc enter debug mode & wait for OnCE commands	DEBUGcc

• These instructions will be discussed in more detail in the OnCE topic.

Example Programs

```
        org     p:$100
start
        clr     b         #>1,x1        ;clear count, load increment
        move              #0,r0         ;pointer to memory
        move              #-1,m0        ;linear addressing
        do      #64,_endtest
        tfr     b,a                     ;value to convert
;
;   This converts a binary value in register A1 to a GRAY coded
;   number in A1.  Register X0 is used as a temporary register.
;
        lsr     a         a1,x0         ;shift bits, copy a
        eor     x0,a                    ;xor adjacent bits to do gray code
        add     x1,b      a1,x:(r0)+    ;inc b, save result in memory
_endtest
        end

            display x:0#64
            X:$0000 000000 000001 000003 000002 000006 000007 000005 000004
            X:$0008 00000C 00000D 00000F 00000E 00000A 00000B 000009 000008
            X:$0010 000018 000019 00001B 00001A 00001E 00001F 00001D 00001C
            X:$0018 000014 000015 000017 000016 000012 000013 000011 000010
            X:$0020 000030 000031 000033 000032 000036 000037 000035 000034
            X:$0028 00003C 00003D 00003F 00003E 00003A 00003B 000039 000038
            X:$0030 000028 000029 00002B 00002A 00002E 00002F 00002D 00002C
            X:$0038 000024 000025 000027 000026 000022 000023 000021 000020
```

```
;
;       Does a square root by polynomial aproximation, 10 bit accuracy.
;                   c2              c1          c0
;       sqrt(x) = -.1985987*x*x + 0.8803385*x + 0.3185231
;
;       valid for:  .5<= x < 1.0
;       input value in x0, output in register A.
;       r1 initially points to the coefficients in y memory.
;
        org     y:0
datin   ds      1                       ;location in y memory of input
datout  ds      1                       ;location in y memory of output
pcoef   dc      .8803385,-.1985987,.3185231     ;c1, c2, c0
;                                       ^
        org     p:$100                  ;|
start   move            #pcoef,r1               ;r1, point to polynomial coefficients
        move                    y:datin,x0  ;get input value
;   polynomial evaluation
        mpyr    x0,x0,a         y:(r1)+,y0  ;a = x**2, c1 -> y0
        mpy     x0,y0,a   a,x1      y:(r1)+,y0  ;a = c1*x, x**2 -> x1, c2 -> y0
        macr    x1,y0,a         y:(r1)+,y0  ;a = c2*(x**2) + c1*x, c0 -> y0
        add     y0,a                    ;add c0
;
        move                    a,y:datout  ;write result to y memory
        end

change y:0 .75
display y:0#5
Y:$0000 600000 000000 70AEEF E69451 28C55D
evaluate y:0
 Hex:600000 Dec:6291456 Fract:0.7500000 Bin:011000000000000000000000

        (Execute Square Root Program)

display y:0#5
Y:$0000 600000 6EFBFE 70AEEF E69451 28C55D
evaluate y:1
 Hex:6EFBFE Dec:7273470 Fract:0.8670652 Bin:011011101111101111111110
```

2.6 Additional DSP5630x Instructions

This section reviews several useful instructions available on the DSP5630x.

2.6.1 MPYI

`mpyi #g,x0,a` or `mpy #g,x0,a`

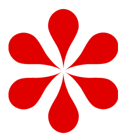
2.6.2 Multi-bit Shift

`asr #N,a,a`

2.7 DSP5630x Architecture & Programming Model

This section is a reproduction of Chapter 6 of the *DSP563xx/6xx Family Digital Signal Processor Training Notes*[5].

[5]Copyright of Motorola, Used with Permission.



DSP5630x Architecture & Programming Model

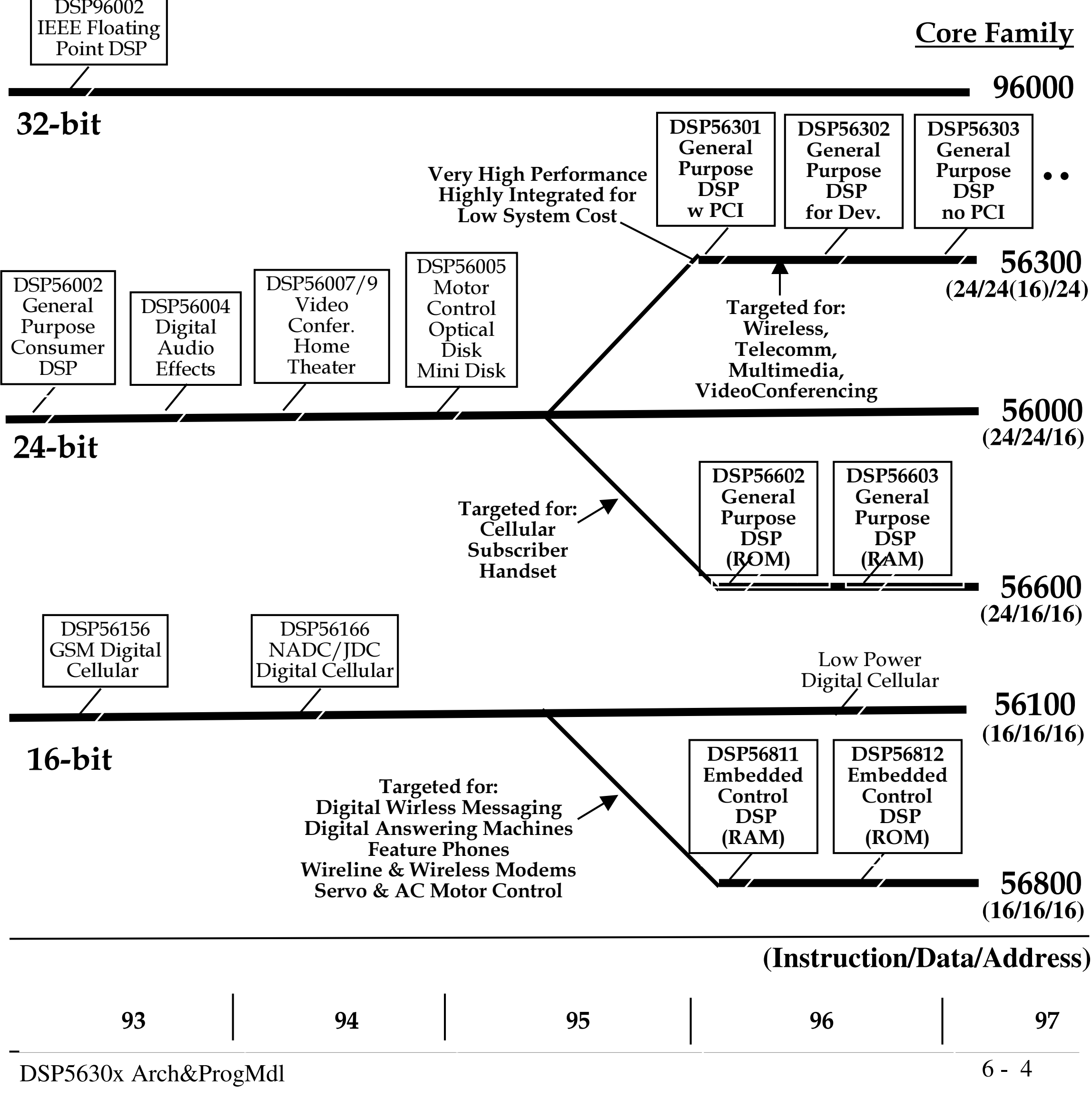
Learn how to:

- Differentiate among DSP56300 family devices
- Extend system stack into data memory
- Use your architectural knowledge to maximize concurrency for execution speed
- Use the Status Register to:
    - Scale data
    - Detect data growth
    - Normalize numbers
    - Detect out of range numbers
    - Adjust global interrupt mask priority
    - Enable the Cache
    - Set compatibility modes

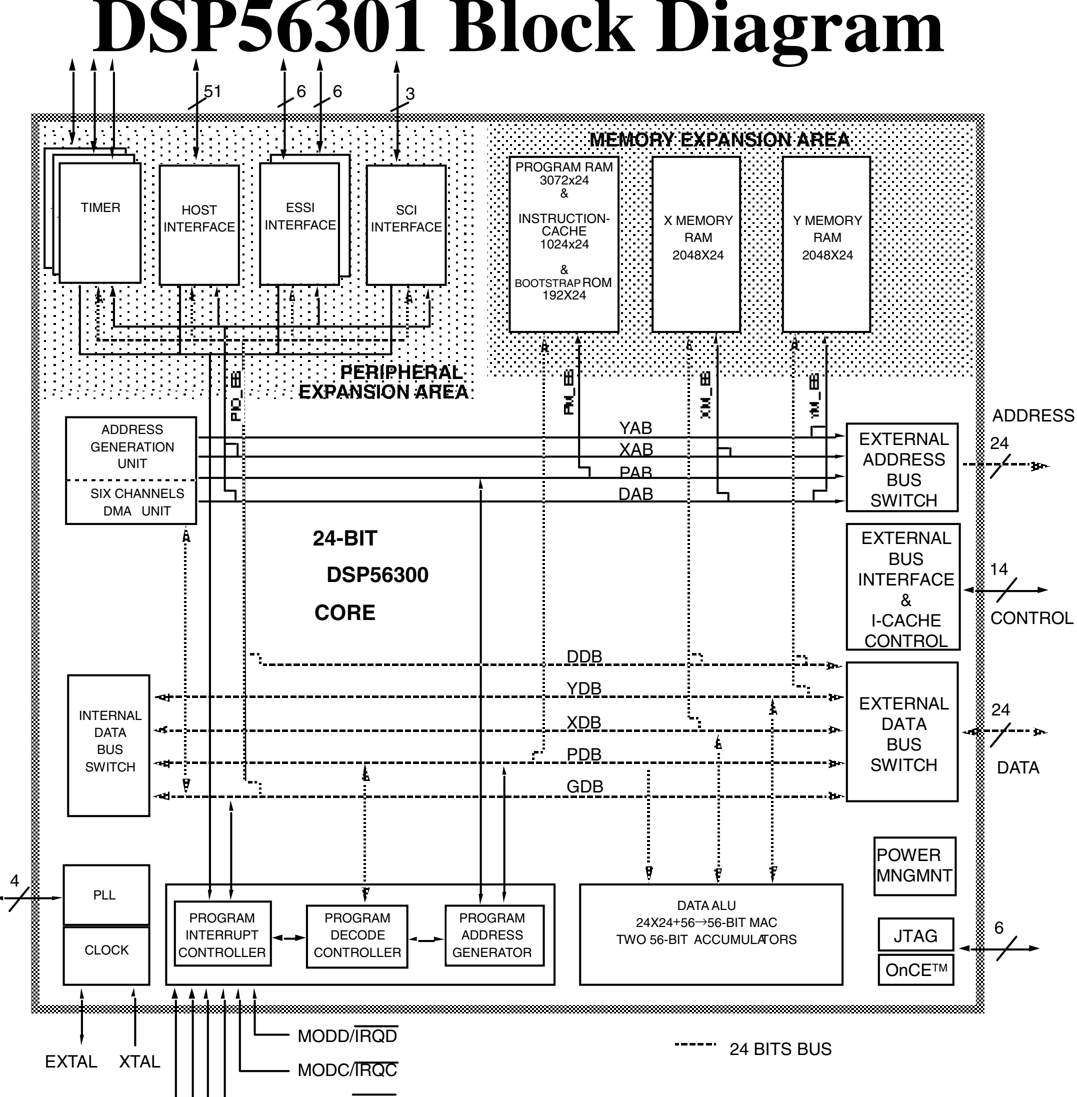
Table of Contents

4....DSP Families
5....DSP56301 Block Diagram
6....DSP563xx Derivatives
7....DSP56301 Memory Map
8....Programming Model
9....Data ALU Block Diagram
10..Address Generation Unit
11..Program Control Unit
12..New SR Functions
13..Map & Registers in 16-bit Modes
14..New OMR Functions
15..Stack Extension Operation
16..Stack Extension Exercise
17..Stack Extension Delays

DSP Families

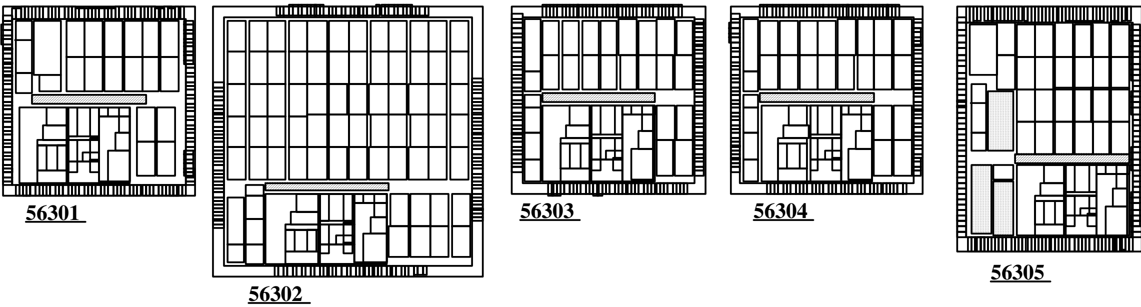


DSP56301 Block Diagram



- UDR2 0.5u geometry
- 66/80/100 MHz clock frequency
- 208 pin low-cost TQFP package
- Functional down to 1.8 V power supply
- Peripherals and interrupt pins are 5-volt-tolerant
- WAIT and STOP modes
- Full-CMOS/fully-static design - 66/80 MHz down to DC
- Intelligent Power Management:
  Automatically powers down unused memory modules, peripherals and core logic on a per clock basis
- 1.7 million transistors
- 3.3 V power supply
- Low-power dissipation:
  less than 1.4mA/MIPS @ 3.3V (66/80 MIPS)
  less than 1.1mA/MIPS @ 2.7V (55/66 MIPS)
  less than 0.75mA/MIPS @ 1.8V (27/33 MIPS)

DSP563xx Derivatives



	56301	56302	56303	56304	56305
Package	208 TQFP 252 PBGA	144 TQFP	144 TQFP 196PBGA	144 TQFP 196PBGA	208 TQFP 252PBGA
Total memory	8K	34K	8K	57K	21.25K
PRAM	4K (2K)	20K (24K)	4K (2K)	1K (3.5K)	6.5K (7.5K)
PROM	-	-	-	33K	6K
DRAM	2K + 2K (3K+3K)	7K + 7K (5K+5K)	2K + 2K (3K+3K)	2.5K + 2.5K (1.25K+1.25K)	3.75K X + 2K Y (2.75K) X
DROM	-	-	-	9K + 9K	3K Y
Host interface	HI32	HI08	HI08	HI08	HI32
Serials	2 ESSI + 1 SCI	2 ESSI + 1 SCI	2 ESSI + 1 SCI	2 ESSI + 1 SCI	2 ESSI + 1 SCI
Timers	3	3	3	3	3
HW Accelerators	-	-	-	-	VCOP CCOP FCOP

DRAM = Data RAM
PRAM = Program RAM
DROM = Data ROM
PROM = Program ROM

DSP56301 Memory Map

PROGRAM

Address	Region
$FFFFFF	INTERNAL - RESERVED
$FF00C0	BOOTSTRAP ROM (192 words)
$FF0000	EXTERNAL
$000FFF	1K INTERNAL I_Cache/PRAM
$000BFF	3K INTERNAL PRAM
$000000	

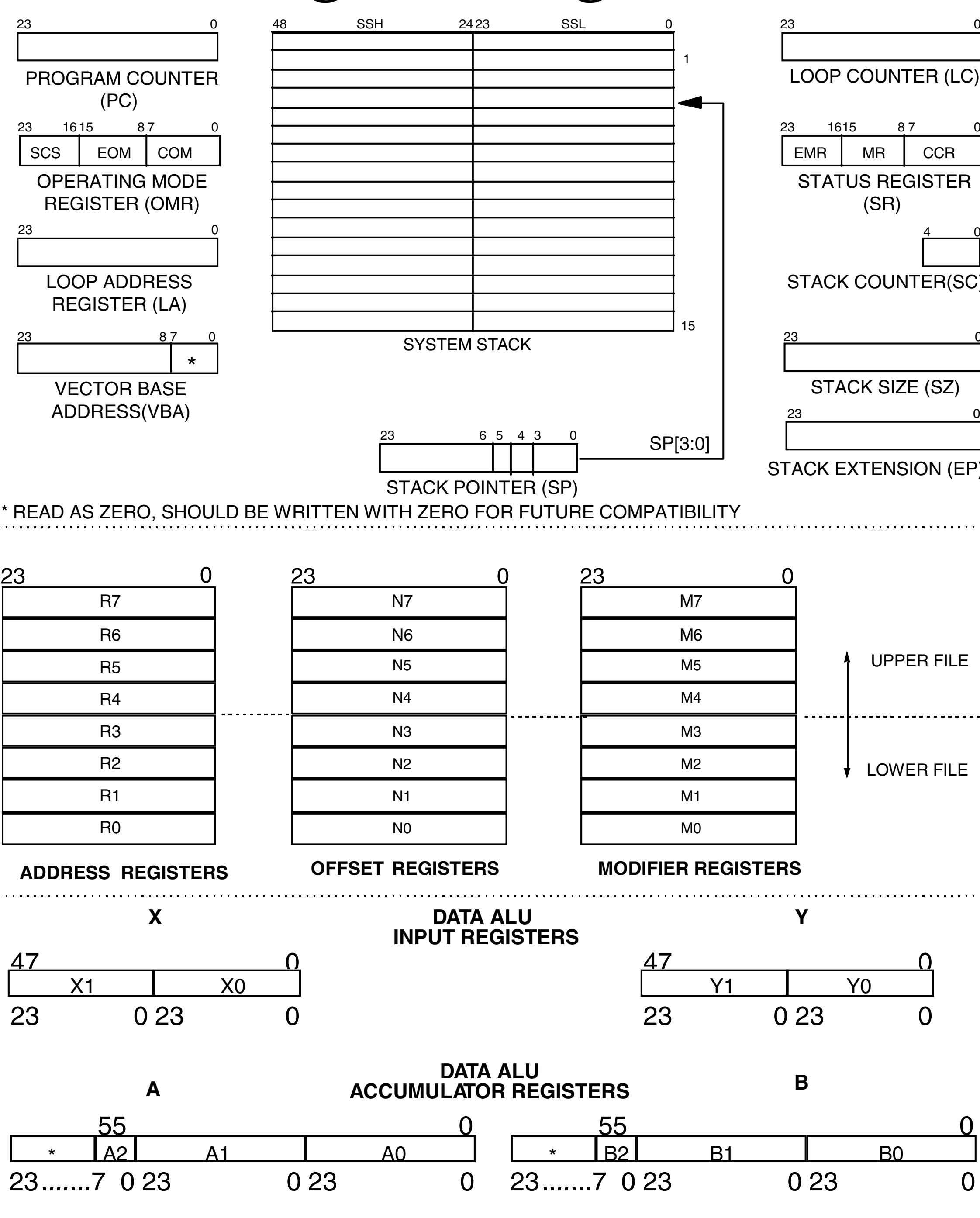
X DATA

Address	Region
$FFFFFF	INTERNAL I/O (128 words)
$FFFF80	EXTERNAL
$FFF000	INTERNAL - RESERVED
$FF0000	EXTERNAL
$0007FF	2K INTERNAL RAM
$000000	

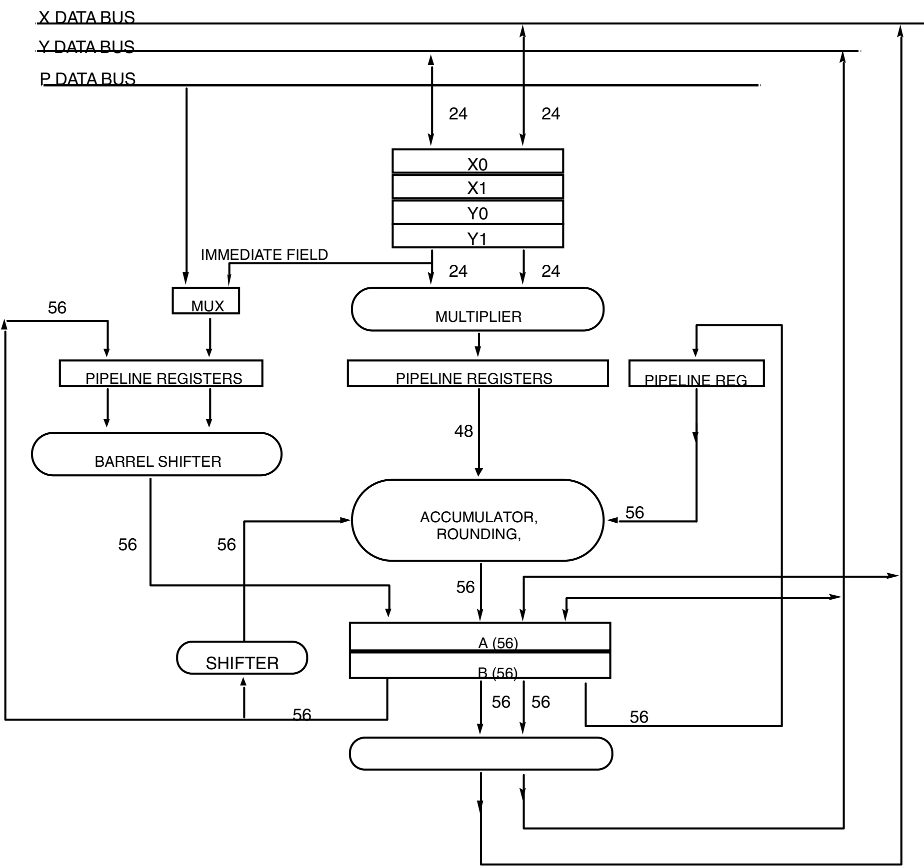
Y DATA

Address	Region
$FFFFFF	EXTERNAL I/O (128 words)
$FFFF80	EXTERNAL
$FFF000	INTERNAL - RESERVED
$FF0000	EXTERNAL
$0007FF	2K INTERNAL RAM
$000000	

- Three independent concurrently accessed 16M word memory spaces:
  - Program (P:)
  - X data (X:)
  - Y data (Y:)
- On-chip peripherals are memory mapped in top 128 words of X data memory
- Off-chip peripherals are memory mapped in top 128 words of Y data memory
- 2K words of on-chip X data RAM (8 banks of 256 words each)
- 2K words of on-chip Y data RAM (8 banks of 256 words each)
- 3K words of on-chip program RAM and 1K words Instruction cache (CE=1), or
  4K words of on-chip program RAM and no Instruction cache (CE=0)
  organized as 12 or 16 banks of 256 words each.
- Glueless interface for off-chip memory expansion

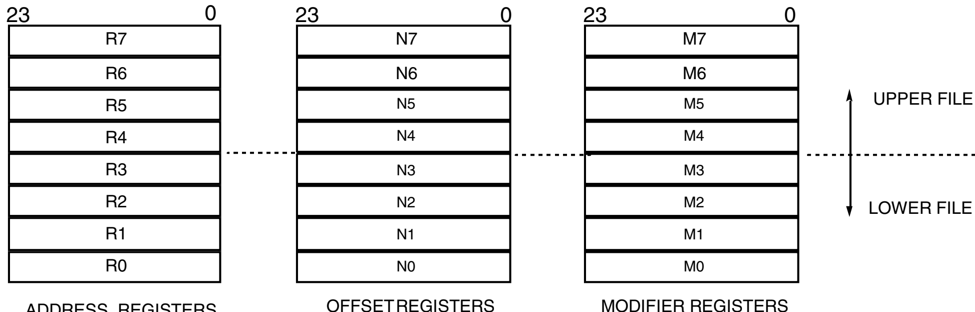

Programming Model
PROGRAM COUNTER (PC)
SCS EOM COM
OPERATING MODE REGISTER (OMR)
LOOP ADDRESS REGISTER (LA)
VECTOR BASE ADDRESS(VBA)
SSH SSL
SYSTEM STACK
STACK POINTER (SP)
SP[3:0]
LOOP COUNTER (LC)
EMR MR CCR
STATUS REGISTER (SR)
STACK COUNTER(SC)
STACK SIZE (SZ)
STACK EXTENSION (EP)
* READ AS ZERO, SHOULD BE WRITTEN WITH ZERO FOR FUTURE COMPATIBILITY
R7 R6 R5 R4 R3 R2 R1 R0
N7 N6 N5 N4 N3 N2 N1 N0
M7 M6 M5 M4 M3 M2 M1 M0
UPPER FILE
LOWER FILE
ADDRESS REGISTERS
OFFSET REGISTERS
MODIFIER REGISTERS
X
DATA ALU
INPUT REGISTERS
Y
X1 X0
Y1 Y0
A
DATA ALU
ACCUMULATOR REGISTERS
B
* A2 A1 A0
* B2 B1 B0
*Read as sign extension bits, written as don't care.

Data ALU Block Diagram



- 100% upward compatible with the 56K Core
- Fully pipelined 24 x 24 Bit Parallel Multiplier-Accumulator
- One MAC instruction per clock execution rate
- 56 Bit Parallel Barrel Shifter
- Performs multibit shift instructions
- Performs bit-field insert/extract for efficient stream parsing
- Performs fast (2 instructions) normalization
- 16 Bit Arithmetic Support
- Permits efficient implementation of standard algorithms (e.g. GSM, ADPCM etc.)
- Permits 16 Bit Data Path Derivative
- Arithmetic Saturation Mode and 2'C Rounding
- Permits implementation of bit-exact standard algorithms
- Multiprecision Arithmetic Support (unsigned and mixed)
- Conditional ALU Instructions
- Saves the overhead of test-and-branch instructions

Address Generation Unit

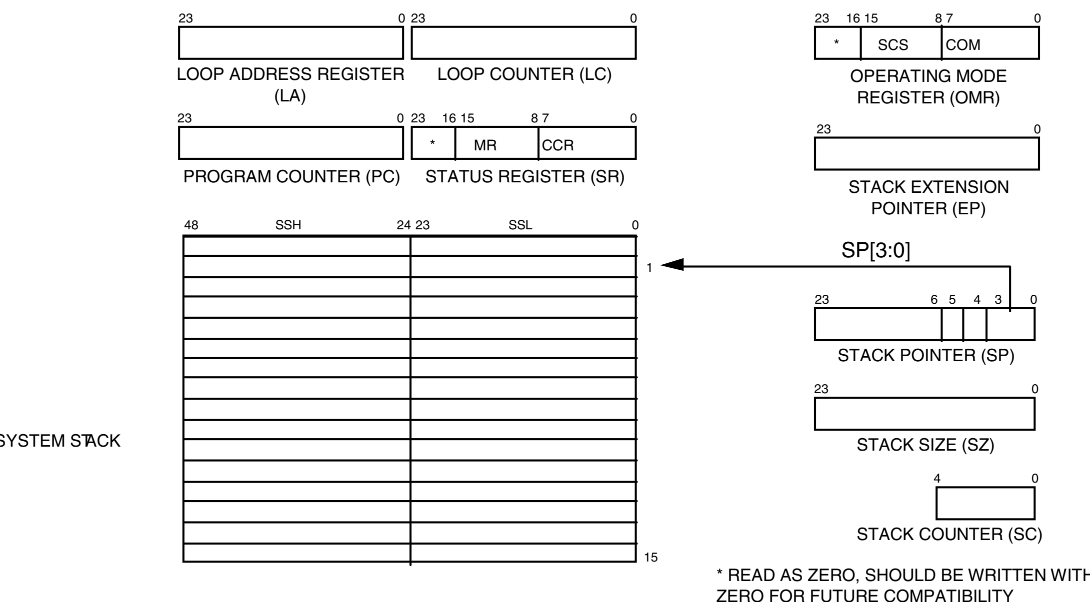


- 100% upward compatible with the 56K Core
- 24 Bit Addresses
- Allows up to 16M words of program and data spaces
- New Addressing Modes:
  - Absolute short/long PC relative jumps - for position independent code and operating systems
  - Address register PC relative (PC + Rn)
  - Short/long Immediate offset (Rn + aa) - for efficient parameter passing in C programs
- Sixteen bit Compatibility mode. When turned on, the following occur in the AGU:
  - Move operations to/from any of the AGU registers (R0-R7, N0-N7 and M0-M7) clear the 8 MSBits of the destination.
  - The 8 MSBits of any AGU address calculation result are cleared.
  - The sign bit of the selected N register is bit15 instead of bit 23.
  - The 8 MSBits of the address are ignored in the calculations of memory regions.

- Address modes:

Syntax	Operation
(Rn)	Access memory pointed to by Rn (no change of Rn)
(Rn)+	Access memory pointed to by Rn and then increment Rn
(Rn)-	Access memory pointed to by Rn and then decrement Rn
(Rn)+Nn	Access memory pointed to by Rn and then increment Rn by value in Nn
(Rn)-Nn	Access memory pointed to by Rn and then decrement Rn by value in Nn
(Rn+Nn)	Access memory pointed to by Rn+Nn (no change of Rn)
-(Rn)	Decrement Rn, then access memory pointed to by decremented value of Rn
(Rn+aa)	Access memory pointed to by Rn+aa (constant offset, no change of Rn)
(PC+displ)	Access memory pointed to by PC+displ (constant offset, no change of PC)
(PC+Rn)	Access memory pointed to by PC+Rn (variable offset, no change of PC)

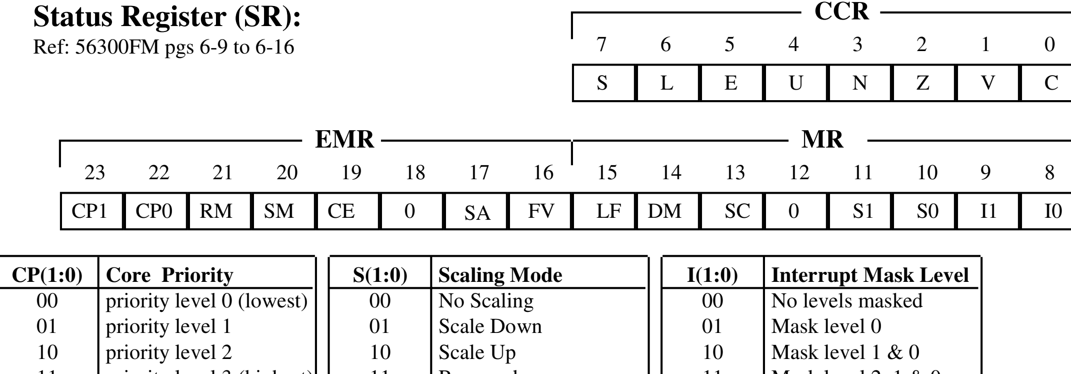
Program Control Unit



- 100% upward compatible with the 56K Core
- Transparent Pipeline
- Conditional jumps/branches optimized for the “taken” path
- Hardware Stack Expandable In Data Memory
- Prevents unrecoverable stack overflow errors
- Permits unrestricted DO loop constructs in C compiler
- Stack Extension Pointer (EP) points to the stack extension in data memory
- Stack Size Register (SZ) determines the total number of entries allocated (2 words each, H/W stack & data memory combined) in the extended mode.
- Stack Counter Register (SC) is used to monitor how many entries of the hardware stack are in use.
- Vector Base Address Register (VBA) points to the base of the vector table.

New SR Functions

Status Register (SR):
Ref: 56300FM pgs 6-9 to 6-16



CP(1:0)	Core Priority
00	priority level 0 (lowest)
01	priority level 1
10	priority level 2
11	priority level 3 (highest)

S(1:0)	Scaling Mode
00	No Scaling
01	Scale Down
10	Scale Up
11	Reserved

I(1:0)	Interrupt Mask Level
00	No levels masked
01	Mask level 0
10	Mask level 1 & 0
11	Mask level 2, 1 & 0

RM	Rounding Mode
0	Convergent Rounding
1	2's Complement Rounding

SC	16 bit Compatibility Mode
0	Disabled
1	Moves of AGU/PCU regs are 16 bit.

SM	Arith. Saturation Mode
0	Extension byte used
1	Saturate on 48 bit results

CE	Cache Enable
0	Cache Disabled
1	Cache Enabled

SA	Sixteen-bit Arith. Mode
0	24 bit arithmetic precision
1	16 bit arithmetic precision

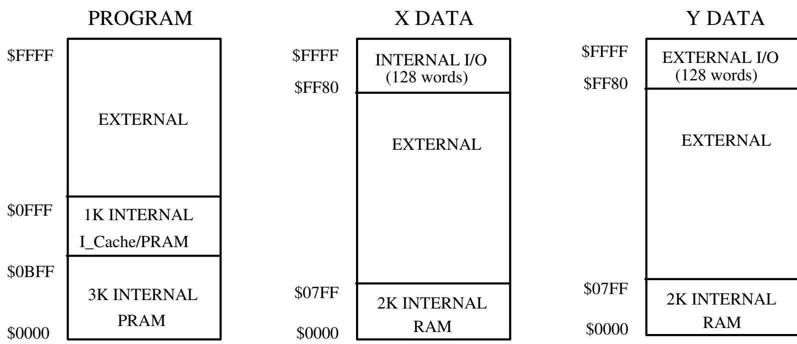
- CP1:0 specify the priority of Core accesses to external memory, relative to DMA.
- RM - for bit exact standard code (e.g. GSM) that used 2's cmpl rounding
- SM - for bit exact standard code (e.g. GSM) that did not make use of extension byte
- SC - for backward compatibility with DSP56k 16-bit addressing.
- SA - for efficient implementation of standard algorithms; e.g. GSM, ADPCM etc.

Saturation Mode operation:

EXT[7]	EXT[0]	MSP[23]	result in accumulator	
0	0	0	unchanged	
0	0	1	$00 7FFFFF FFFFFF	Limit bit sets
0	1	0	$00 7FFFFF FFFFFF	
0	1	1	$00 7FFFFF FFFFFF	
1	0	0	$FF 800000 000000	Limit bit sets
1	0	1	$FF 800000 000000	
1	1	0	$FF 800000 000000	
1	1	1	unchanged	

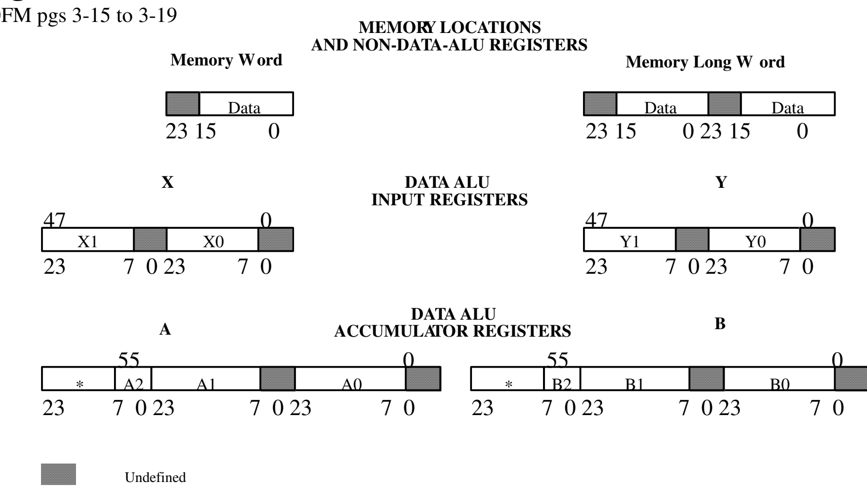
Map & Registers in 16 bit Modes

Memory Map in 16-bit Compatibility mode:



Data Organization in 16-bit Arithmetic mode:

Ref: 56300FM pgs 3-15 to 3-19



- Numerical results have 40 bits of accuracy: 16-bit LSP & MSP, 8-bit Ext.
- Moves from XDB or YDB into Acc A or B are put into bits 32-47, bits 8-23 are cleared, and the extension byte is loaded with the sign extension. Other bits undefined.
- Moves from XDB or YDB into 24-bit ALU registers are put into bits 8-23, other bits undefined.
- Moves from 24-bit register to memory the 16 MS bits are moved to the 16 LS bits of memory and the 8 MS bits of memory are cleared.
- Moves from Acc A or B, scaling and limiting are in effect, the 16 MS bits are moved to the 16 LS bits of memory and the 8 MS bits of memory are the sign extension.
- When switching to and from 16-bit arith. mode, for two instruction cycles, no arith. or move instrs.

New OMR Functions

Operating Mode Register (OMR):
Ref: 56300FM pgs 6-16 to 6-20

COM							
7	6	5	4	3	2	1	0
MS	SD	0	EBD	MD	MC	MB	MA

SCS								EOM							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
0	0	0	SEN	WRP	EOV	EUN	XYS	0	0	0	BRT	TAS	BE	CDP1	CDP0

SEN - Stack Extension Enable
WRP- Extended Stack Wrap Flag
EOV - Extended Stack Overflow Flag
EUN - Extended Stack Underflow Flag
XYS - Stack Extension Space Select

BRT - Bus Release Timing
TAS - TA Synchronize Select
BE - Burst Mode Enable
MS - Memory Switch Mode
EBD - External Bus Disable

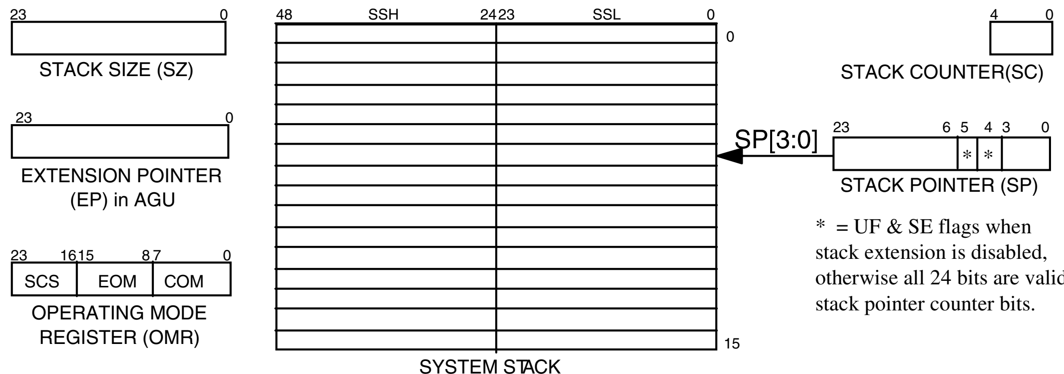
CDP(1:0)	Core-DMA Priority
00	determined by comparing CP1:0 with the active DMA channel priority
01	DMA accesses have higher priority than Core accesses
10	DMA accesses have the same priority as the Core accesses
11	DMA accesses have lower priority than the Core accesses

MOD(D:C:B:A)	Reset Vector	DSP56301 Operating Modes
0000	$C00000	Expanded Mode
x001	$FF0000	Bootstrap from byte-wide memory
x010	$FF0000	Bootstrap thru SCI
x011	-	Reserved
x100	$FF0000	Host Bootstrap PCI Mode (32-bit wide)
x101	$FF0000	Host Bootstrap 16-bit wide UB Mode (ISA)
x110	$FF0000	Host Bootstrap 8-bit wide UB Mode in dbl strb pin config
x111	$FF0000	Host Bootstrap 8-bit wide UB Mode in sgl strb pin config
1000	$008000	Expanded Mode

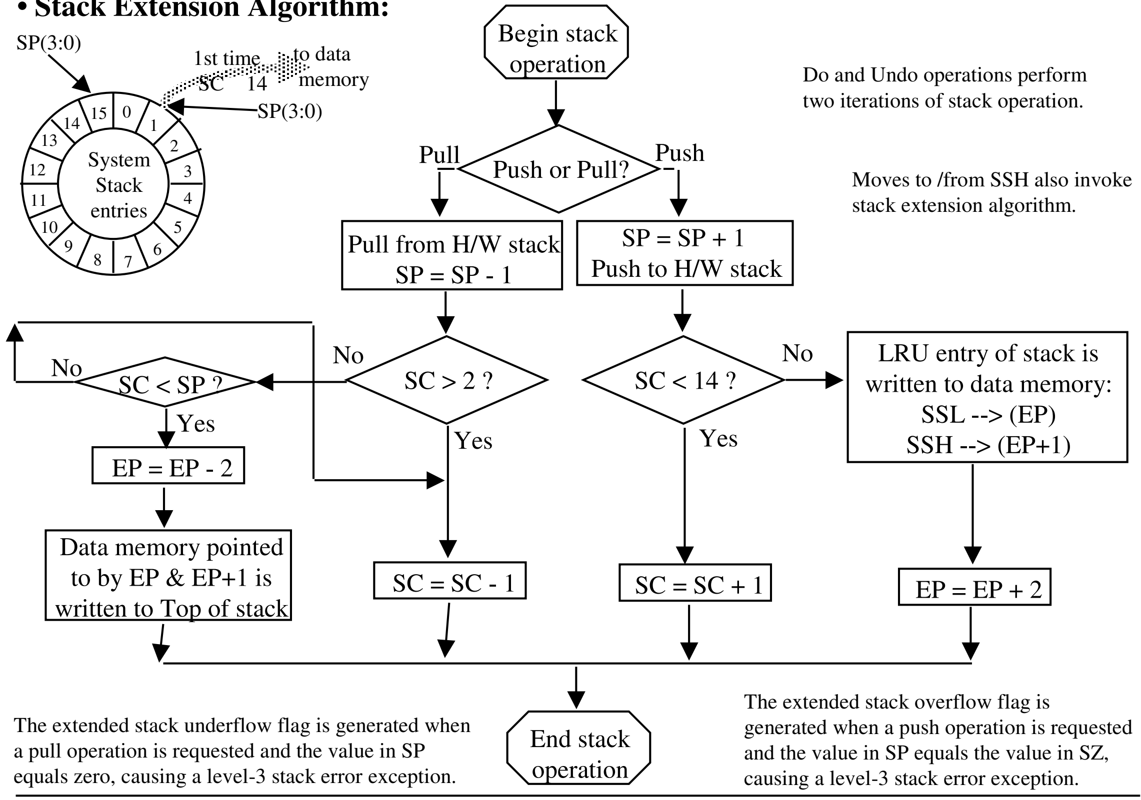
x = don’t care (0 or 1)

• WRP may be used during debug phase of the S/W as a means of evaluating & increasing the speed of the S/W algorithm. WRP is sticky & is set when stack extension memory is first needed.
• BRT selects fast or slow bus release (for fast - BB is not guaranteed to be the last tri-stated pin)
• TAS selects whether the H/W designer must, or the CPU synchronizes TA to the CLKOUT pin.
• EBD disables the external bus (for lower power consumption when no memory expansion needed)

Stack Extension Operation



- The System Stack (SS) is a separate 16x48-bit internal memory divided into two banks: System Stack High (SSH) and System Stack Low (SSL). One location is unusable when the stack extension is disabled.
  - Storing return address (SSH) and chip status (SSL) for subroutine calls
  - Storing LA and LC (loop values), PC and SR (set LF) for hardware DO loops
- Up to 15 long interrupts, 7 DO loops or 15 JSRs can be handled by the SS (with stack extension disabled)
- When the SS limit is exceeded and Stack Entension is disabled a nonmaskable stack error interrupt occurs
- When Stack Extension is enabled the barrier of 15 nesting levels can be changed to any desired value (SZ).
- **Stack Extension Algorithm:**



The extended stack underflow flag is generated when a pull operation is requested and the value in SP equals zero, causing a level-3 stack error exception.

The extended stack overflow flag is generated when a push operation is requested and the value in SP equals the value in SZ, causing a level-3 stack error exception.

Stack Extension Exercise

Write a program that enables stack extension in Y data memory from location $0007F0 to $0007FF.

Write your program here:

Suggested program steps:
(the number of program steps may not match the number of instructions required)

1. Initialize EP

2. Initialize SZ

3. Enable stack extension in Y memory

Stack Extension Delays

• Stack Extension Inserts Delays into the program flow
  - In cases of Stack full or Stack empty
  - Stack-full or stack-empty states are defined by the contents of the SC (Stack Counter)
  - The stack is full when SC = 14, and additional push operations are required
  - The stack is empty when SC = 2, and additional pop operations are required
  - The stack counter equals 2, it means that the on-chip hardware stack has only 2 full entries (4 words) inside. The stack is declared as stack-empty, and any additional pop operation will activate the stack extension mechanism.

Stack Extension Delays:

Instruction Examples	**Stack Full Condition (+ clock cycles)**	**Stack Empty Condition (+ clock cycles)**
JSR, JSSET, BRCLR	2	-
RTI,RTS	-	3
LC=1, PAB=LA	-	5
DO,DOR	4	-
ENDDO	-	5
MOVE R0,SSH	2	-
MOVE SSH,Y1	-	3

2.8 Motorola DSP56302EVM

MOTOROLA
SEMICONDUCTOR PRODUCT INFORMATION

Order this document by:
DSP56302EVMP/D

DSP56302

Advance Information
DSP56302 Evaluation Module

The DSP56302 Evaluation Module (DSP56302EVM) is designed as a low-cost platform for developing real-time software and hardware products to support a new generation of applications in wireless, telecommunications, and multimedia products using multi-line voice/data/fax processing, videoconferencing, audio applications, control, and general digital signal processing. The user can download software to on-chip or on-board RAM, then run and debug it. The user can also connect hardware, such as external memories and A/D or D/A converters, for product development. The 24-bit precision of the DSP56302 Digital Signal Processor (DSP) combined with the on-board 32 K of external SRAM and Crystal Semiconductor's CS4215 stereo, CD-quality, audio codec makes the DSP56302EVM ideal for implementing and demonstrating many communications and audio processing algorithms, as well as for learning the architecture and instruction set of the DSP56302 processor. **Figure 1** shows the functional block diagram for the DSP56302EVM.

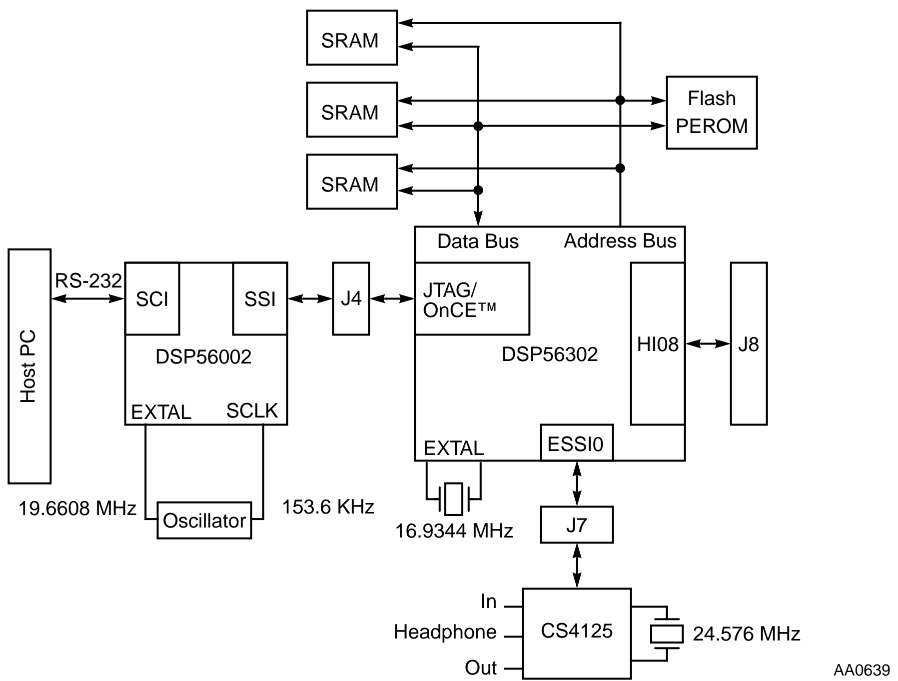


Figure 1 DSP56302EVM Functional Block Diagram

This document contains information on a new product. Specifications and information herein are subject to change without notice.

Preliminary Information

©1996 MOTOROLA, INC.

MOTOROLA

Figure 2.4: Motorola DSP56302EVM Product Brief p. 1.

2.9 Problems

1. Read Chapter 1 Core Description of *Motorola DSP56300 Family Manual*.
2. Read Chapter 3 Data Arithmetic Logic Unit of *Motorola DSP56300 Family Manual*.
3. Read Chapter 4 Address Generation Unit of *Motorola DSP56300 Family Manual*.
4. Read Chapter 6 Program Control Unit of *Motorola DSP56300 Family Manual*.
5. Read Chapter 12 Operating Modes and Memory Spaces of *Motorola DSP56300 Family Manual*.

Features

Hardware

- 24-bit DSP56302 Digital Signal Processor
  - High Performance DSP56300 Core
    - 66/80 Million Instructions Per Second (MIPS) with a 66/80 MHz clock
    - Object-code compatible with the DSP56000 core
    - Highly parallel instruction set
    - Fully pipelined 24 × 24-bit parallel multiplier-accumulator
    - 56-bit parallel barrel shifter
    - 24-bit or 16-bit arithmetic support under software control
    - Position Independent Code (PIC) Support
    - Unique DSP addressing modes
    - On-chip memory-expandable hardware stack
    - Nested hardware DO loops
    - Fast auto-return interrupts
    - On-chip concurrent six-channel DMA controller
    - On-chip Phase Lock Loop (PLL)
    - On-Chip Emulation (OnCE™) module
    - JTAG port
    - Address tracing mode reflects internal program RAM accesses at external port
  - On-Chip Memories
    - Program RAM, Instruction Cache, X data RAM, and Y data RAM size is programmable:

Instruction Cache	Switch Mode	Program RAM Size	Instruction Cache Size	X Data RAM Size	Y Data RAM Size
disabled	disabled	20480 × 24-bit	0	7168 × 24-bit	7168 × 24-bit
enabled	disabled	19456 × 24-bit	1024 × 24-bit	7168 × 24-bit	7168 × 24-bit
disabled	enabled	24576 × 24-bit	0	5120 × 24-bit	5120 × 24-bit
enabled	enabled	23552 × 24-bit	1024 × 24-bit	5120 × 24-bit	5120 × 24-bit

    - 192 × 24-bit bootstrap ROM

Preliminary Information

Figure 2.5: Motorola DSP56302EVM Product Brief p. 2.

- Off-Chip Memory Expansion
  - Data memory expansion to two memory spaces of 256 K × 24-bit words
  - Program memory expansion to one memory space of 256 K × 24-bit words
  - External memory expansion port
  - Four chip-select logic lines for glueless interface to SRAMs and SSRAMs
  - On-chip DRAM controller for glueless interface to DRAMs
- On-Chip Peripherals
  - Enhanced DSP56000-like 8-bit parallel Host Interface (HI08)
  - Two Enhanced Synchronous Serial Interfaces (ESSI)
  - Serial Communications Interface (SCI) with baud rate generator
  - Triple timer module
  - Up to thirty-four programmable General Purpose Input/Output (GPIO) pins, depending on which peripherals are enabled
- Reduced Power Dissipation
  - Very low power CMOS design
  - Wait and Stop low power standby modes
  - Fully-static logic, operation from the device maximum frequency down to DC

- 32 K × 24-bit fast Static RAM for expansion memory
- 64 K × 8-bit Flash PEROM for stand-alone operation
- 16-bit CD-quality audio codec
  - Two channels of 16-bit Analog-to-Digital (A/D) conversion
  - Two channels of 16-bit Digital-to-Analog (D/A) conversion
  - Software-selectable 8-bit and 16-bit data formats, including μ-law and A-law companding
  - Stereo jacks for audio input, output, and headphones
- Command Converter
  - DSP56002 for high-speed OnCE/JTAG command conversion software
  - JTAG connector for use with the Application Development System (ADS) command converter card
- Connectors
  - Host-to-ISA bus connector
  - Port A connector
  - ESSI0, ESSI1, and SCI connector

Preliminary Information

MOTOROLA **DSP56302EVM Product Brief** **3**

Figure 2.6: Motorola DSP56302EVM Product Brief p. 3.

Software

- Motorola's DSP56xxx cross assembler
  - Produces DSP56302 binary code from source code using labels, sections, and macro definitions incorporating the DSP's complete instruction set, all addressing modes, and all memory spaces
  - Offers macros, expression evaluation, and functions for strings, data conversion, and transcendentals
  - Creates reports for cross-references, instruction cycle count, and memory usage
  - Provides extensive error checking and reporting
- Domain Technologies debug software with Windows-based user interface
  - Symbolic debugging
  - Windows for data, code, DSP registers, commands, peripherals, etc.
  - Data and registers displayed in fractional, decimal, or hexadecimal format
  - Graphical display of memory segments
  - Up to eight simultaneous software breakpoints
  - Built-in-line assembler and disassembler
- Demonstration software and example pass-through code
- Self-test files—executable and source code (Flash PEROM is preprogrammed with self-test and audio echo software.)

User Requirements

The user must provide the following:

- Power supply (7–9 V AC or DC with 2.1 mm power connector)
- RS-232 cable (DB9 male to DB9 female)
- Audio source, headphones, and a cable with 1/8-inch stereo plugs
- IBM PC compatible computer (386 class or higher) running Windows 3.1 (or higher) with an RS-232 serial port capable of 9,600–57,600 bit-per-second operation, 4 Mbytes RAM, 3-1/2 inch diskette drive, hard drive with 4 Mbyte of free disk space, and a mouse

Preliminary Information

4 DSP56302EVM Product Brief MOTOROLA

Figure 2.7: Motorola DSP56302EVM Product Brief p. 4.

Chapter 3

Development Tools

The main objective of this chapter is to familiarize yourself with the software tools by going through the complete development cycle with a few demo programs.

3.1 Getting Started with the Motorola DSP5630xEVM

3.1.1 Background

The purpose of this section is to step you through the software development tools and to practice debugging techniques. Motorola provides the assembler (`asm56300.exe`) and linker/librarian (`dsplnk.exe`) while Domain Technologies provides the debugger (`evm56kw.exe`). The assembler converts your ASCII text file into a relocatable object code. The linker function of the linker/librarian converts the relocatable object into an absolute object file which can be downloaded and executed (run) on the DSP5630x. The debugger is used for analyzing the operation of the DSP5630x running your algorithm. It allows you to step through code, examine the contents of registers and memory, and set breakpoints (markers in your code where execution is halted). These capabilities help determine why a program is not doing what you expect. The debugger is also used to download your object code to the EVM.

Debugging DSP programs is more complicated than normal programs. There are three complications. First, the numerical orientation of DSP programs complicates evaluation. When you write a C program to print "Hello, world" but you get "Goodbye, Mars", it's obvious that something is wrong. With DSP programs, you may have to examine long lists of numbers (samples) to figure out what's going on. Second, our DSP programs involve multiprocessing—the processor executing the DSP program (DSP5630x) is different from the host processor running the Windows operating system (Pentium). This bifurcation requires special tools (debugger) for controlling and monitoring the DSP5630x from the host. You cannot insert `printf` in your DSP code because `printf` does

not exist in that environment. The third complication has to do with the real-time nature of the DSP program. To produce audio output, the EVM must emit a sample every sample time. Most attempts to monitor the activity of the DSP5630x will interfere with the real-time requirements. So you have an example of the Heisenberg Uncertainty Principle in that any attempt to observe the operation of the program changes the operation.

Rule 1: Always turn down the volume BEFORE running any program the first time. It's amazing how loud and noisy (and potentially damaging to the ears) buggy code can be! ■

3.1.2 DSP5630xEVM Self-Test

You may test the integrity of the EVM at any time by running the DSP5630xEVM self-test. This can be used during the debugging process to rule out EVM failure (as if one's code could *never* be the problem!). Follow the instructions in *DSP5630xEVM User's Manual* (Section 1.3.4 of *DSP56302EVM User's Manual*).

3.1.3 Demonstration File

The first demonstration of the EVM takes an input signal, digitizes it, and adds a 60Hz tone to simulate the noise that is generated by a 60Hz AC power line. The signal is then notch filtered using three different sets of filter coefficients to produce three different results. Follow the instructions in *DSP5630xEVM User's Manual* (Section 2 of *DSP56302EVM User's Manual*) for running the demonstration file.

3.1.4 Example Test Program

The example test program found in *DSP5630xEVM User's Manual* (Section 3 of *DSP56302EVM User's Manual*) will demonstrate the form of assembly programs, give instructions on how to assemble programs, and show how to use the debugger to verify the operation of programs. The DSP56300 Assembler will be discussed in detail the next section.

You can type in the example program which calculates an inner product using any editor. One popular editor is the freeware *Programmer's File Editor*. Once you've typed the example program, it must be assembled into object code which can be downloaded to the EVM. The *DSP5630xEVM User's Manual* describes the format of the assembler command including assembler options and gives a list of assembler special characters and directives. Assembler directives are instructions given to the assembler to perform auxiliary actions. These directives are different from instructions given to the DSP5630x. For now do not be too concerned with the assembler options and directives. Once you have typed in the program, name it `example.asm`.

Next, assemble the program with the following DOS command (you will have to open a DOS window from within the Windows OS)

```
asm56300 -a -b -l -g example.asm
```

You may wish to create a batch file (`asm.bat`) with the above command so that you need only type

```
asm example.asm
```

and the full call (including options) to the assembler will be made. The assembler will alert you to any typing errors you may have in `example.asm`. These errors (bugs) need to be corrected before complete assembly can take place. Once `example.asm` has been fully debugged and assembled, two files will be created: `example.cld` and `example.lst`. The CLD file is the absolute executable file which will be downloaded (via the debugger) to the DSP5630x and executed. The LST file is the listing file which gives full details of where the program and data will reside in the DSP5630x memory. You may wish to open the LST file in the editor and examine it.

The Motorola DSP linker is a program that processes relocatable object code produced by the assembler into an absolute executable file which can be downloaded to the DSP5630x. Since we specified the `-a` option when we assembled, the linker was automatically invoked to produce an absolute object file.

The final step is to download the program to the DSP5630x and execute it—both of which are accomplished by the debugger. After you start the debugger, various windows will be displayed and these may be slightly different from those in *DSP5630xEVM User's Manual*. Run the program according to the instructions given in *DSP5630xEVM User's Manual*.

3.2 Motorola DSP56300 Assembler

As mentioned earlier, the Motorola DSP56300 Assembler converts your ASCII text file into an object file (assuming you've invoked the linker option) that the DSP5630x can execute. Assembly programs can contain *assembler directives* which specify auxiliary actions to be performed by the assembler. These directives are different from the DSP5630x instruction set and can greatly aid in the development process. The assembler directives are defined in *DSP5630xEVM User's Manual* (Section 3.3.3 and Appendix C of *DSP56302EVM User's Manual*). As a matter of style, we will place all assembler directives in the `pass.dat` file (which will be discussed shortly). A few useful assembler directives and examples are given here.

3.2.1 Equate (EQU)

```
<label>     EQU     [{X: | Y: | L: | P: | E:}]<expression>
```

The Equate (EQU) directive assigns the value and memory space attribute of `<expression>` to the symbol `<label>`. If `<expression>` has a memory space attribute of None, then it can optionally be preceded by any of the indicated memory space qualifiers to force a memory space attribute. An error will occur if the expression has a memory space attribute other than None and it is different than the forcing memory space attribute. The optional forcing memory space attribute is useful to assign a memory space attribute to an expression that consists only of constants but is intended to refer to a fixed address in a memory space.

The EQU directive is one of the directives that assigns a value other than the program counter to the label. The label cannot be redefined anywhere else in the program (or section, if SECTION directives are being used). The `<expression>` may be relative or absolute, but cannot include a symbol that is not yet defined (no forward references are allowed).

Example:

```
N     equ     16
```

3.2.2 Originate (ORG)

```
ORG     <rms>[<rlc>][<rmp>]:[<expr1>][,<lms>[<llc>][<lmp>]:[<exp2>]]
ORG     <rms>[<rmp>][(<rce>)]:[<expr1>][,<lms>[<lmp>][(<lce>)]:[<exp2>]]
```

The ORG directive is used to specify addresses and to indicate memory space and mapping changes. It can also designate an implicit counter mode switch in the assembler and serves as a mechanism for initiating overlays. Some of the parameters used with the ORG directive are:

- `<rms>`—which memory space (X, Y, L, P, or E) will be used as the runtime memory space. If the memory space is L, any allocated datum with a value greater than the target word size will be extended to two words; otherwise it is truncated. If the memory space is E, then depending on the memory space qualifier, any generated words will be split into bytes, one byte per word, or a 16/8-bit combination.
- `<rlc>`—which runtime location counter H, L, or default (if neither H or L is specified) that is associated with the `<rms>` will be used as the runtime location counter.
- `<exp1>`—initial value assigned to the runtime counter used as the `<rlc>`. If `<exp1>` is a relative expression the assembler uses the relative location counter. If `<exp1>` is an absolute expression the assembler uses the absolute location counter. If `<exp1>` is not specified, then the last value and mode that the counter had will be used.

If the last half of the operand field in an ORG directive dealing with the load memory space (LMS), load location counter (LLC), and load physical mapping (LMP) is not specified, then the assembler will

assume that the load memory space and load location counter are the same as the runtime memory space and runtime location counter. In this case, object code is being assembled to be loaded into the address and memory space where it will be when the program is run, and is not an overlay.

Example:

```
ORG     x:$00000a     ;lay things out in x-memory starting at $000a
```

3.2.3 Define Storage (DS)

```
<label>     DS     <expression>
```

The Define Storage (DS) directive reserves a block of memory the length of which in words is equal to the value of `<expression>`. This directive causes the runtime location counter to be advanced by the value of the absolute integer expression in the operand field. `<expression>` can have any memory space attribute. The block of memory reserved is not initialized to any value. The expression must be an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined).

`<label>`, if present, will be assigned the value of the runtime location counter at the start of the directive processing.

Example:

```
ARRAY     ds     12     ;12 words of memory allocated for array
```

3.2.4 Define Constant (DC)

```
<label>     DC     <arg>[,<arg>,...,<arg>]
```

The Define Constant (DC) directive allocates and initializes a word of memory for each argument, `<arg>`. Arguments may be a numeric constant, a single or multiple character string constant, a symbol, and/or an expression. The DC directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zero(s). If DC is used in L memory, the arguments will be evaluated and stored as long word quantities. Otherwise, an error will be occur if the evaluated argument is too large to represent in a single DSP word. `<label>`, if present, will be assigned the value of the runtime location counter at the start of the directive processing. Integer arguments are stored "as is;" floating point numbers are converted to binary values. Single and multiple character strings are handled in the following manner:

- Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character.
- Multiple character strings represent words whose bytes are composed of concatenated sequences of the ASCII representation of the characters in the string (unless the `NOPS` option is specified; see the `OPT` directive). If the number of characters is not an even multiple of the number of bytes per DSP word, then the last word will have the remaining characters left aligned and the rest of the word will be zero-filled. If the `NOPS` option is given, each character in the string is stored in a word whose lower seven bits represent the ASCII value of the character.

Example:

```
COEFS    dc    0.1,0.2,0.3,0.4,0.5    ;coefficients stored in memory
```

3.2.5 Define Storage Modulo (DSM)

```
<label>    DSM    <expression>
```

The DSM directive reserves a block of memory the length of which in words is equal to the value in `<expression>`. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where

$$2^k \geq \texttt{<expression>}.$$

An error will be issued if there is insufficient memory remaining to establish a valid base address. Next the runtime location counter is advanced by the value of the integer expression in the operand field. `<expression>` can have any memory space attribute. The block of memory reserved is not initialized to any given value. The result of `<expression>` must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). The expression also must fall within range

$$2 \leq \texttt{<expression>} \leq m$$

where m is the maximum address of the target DSP. `<label>`, if present, will be assigned the value of the runtime location counter at the start of the directive processing.

Example:

```
CIRCULAR_BUFFER    dsm    16    ;16 words allocated for circ. buffer
```

3.2.6 Other Examples

In programming FIR filters, we will need to allocate modulo storage for input samples and allocate memory for filter coefficients. While the latter does not have to be modulo, doing so will simplify pointer reset to the first coefficient. The following assembler directives will prepare memory for the FIR filter.

```
N equ 5                     ;filter length
 ORG x:$00000a              ;set RLC to x:$00000a
input dsm N                 ;allocate modulo storage for samples
 ORG y:$000000              ;set RLC to y:$000000
coefs dsm N                 ;allocate modulo storage for coefficients
 ORG y:coefs                ;reset RLC to beginning of coefficient storage
 dc     0.2,0.2,0.2,0.2,0.2 ;initialize coefficient memory
```

We note the DSM directive allocating memory for `coefs` advances the runtime location counter by `N`. Therefore we must reset the location counter back to `y:coefs` and then begin laying down the coefficients in memory. The DC directive will also advance the location counter by as many coefficients as we put in memory. If the number of coefficients we lay down is less than or equal to `N`, the runtime location counter will be set properly for the next memory allocation directive. Otherwise we will overwrite our coefficients.

3.3 Domain Technologies Debug-56K

In all likelihood, you will spend the vast majority of your development time using the debugger. Among other things, the debugger will allow you to step through lines of code and examine the states of memory and registers. This examination can provide insight as to why your program does not operate correctly. In any software development environment, a good debugger is critical and the better you know how to use one, the more efficient you will become at the debugging process. Fortunately, the debugger supplied with the EVM is simple to use yet powerful. The *Debug-EVM User's Manual* contains most of the information regarding the use of the debugger. In this section we present some tips on debugging and the use of the in-line assembler feature of the debugger.

3.3.1 Tips on Debugging

Whenever you encounter an architecture for the first time, you should start with a simple program. `pass.asm` is a good choice because the signal processing that it performs is trivial. However, the first time you run `pass` in an untested environment it may not work. To debug it, you might consider setting "breakpoints" to examine register contents at various points in the main routine. When a breakpoint is encountered in your code, the execution is halted allowing you to examine the current state of registers and memory. However, you won't be able to exercise the real-time characteristics

of the program. To debug the program without destroying its real-time operation, you-must apply the first principle of debugging: simplify. Suppose that `pass.asm` didn't work the first time; how would you simplify it to track down the problem? Even a program as simple as `pass.asm` can be simplified. First the operation involves both A/D and D/A conversion. Perhaps the problem has to do with the A/D conversion. Fine. Let's bypass the A/D portion of the PASS program and replace it with a synthesized waveform. Be sure to use a trivial synthesis algorithm—we don't want to confuse our debugging efforts by using buggy analysis. How about a sawtooth waveform: just increment the output sample each pass and let the value wrap around. Remember, the objective is to get something—anything—to come out of the EVM. If we can get the synthesized sound to come out then we know the problem must be in the A/D portion of the code—the problem is cornered.

What if we still can't produce an output? First, be sure that the synthesis code is correct. Then, simplify again. Perhaps the problem has to do with the interrupt-handling mechanism. There are plenty of opportunities for problems when dealing with interrupts. Perhaps execution is not reaching the interrupt service routine (ISR). You might be able to set a breakpoint in the ISR, but the most compelling analytical technique would be to avoid interrupts altogether. Try moving the sawtooth synthesizer to the main routine and just loop around it without waiting for an interrupt. The sample rate will be wrong, but something should come out. If this doesn't work, your synthesis routine is probably wrong.

The watchword in debugging DSP programs is to be *creative*. In normal programs, debugging is often a mechanical process. With DSP programs, debugging is anything but mechanical and may require as much creativity and insight as developing the code.

3.3.2 In-line Assembly

The debugger has a built-in assembler which allows you to type in DSP5630x instructions and run them. This feature is extremely useful to the developer in that you can "try out" instructions and see how they work.

Example: In this example, we will place an `ADD` instruction at `P:$000100`, execute it, and examine the effects. First, place the assembled instruction in memory. In the Command Window type

```
asm P:$000100 add x0,a
```

Next in the Registers Window, set `x0` equal to 0.25 and `a` equal 0.5 so you can have something to add. Next, change the Program Counter (PC) to `$000100` so the DSP knows where to begin execution. In the Command Window type

```
change pc $000100
```

Finally, execute the instruction by typing

```
step
```

Note in the Registers Window the updated value of `a` which should be equal to 0.75.

3.4 Problems

1. Go through the tutorial in the user's manual for `Debug-56K`.

Chapter 4

Simple Programs

In this chapter, we examine several basic programs. These include some non-real-time programs which perform simple arithmetic as well as a program for real-time DSP known as `pass.asm`. The pass code will be the basis all of our real-time DSP codes.

4.1 Simple 5630x Programs

4.1.1 Program 1

The purpose of Program 1 is to add two numbers. Type the following code into a file called `program1.asm`, assemble, and run. Stop the program (`FORCE BREAK`) and verify that indeed it does add the two numbers. As an exercise, go through and comment each line of code.

```
;**************************************************************
;PROGRAM #1--This Program will add two numbers
;**************************************************************
;X Memory
   org x:$0
input1 dc $123

;Y MEMORY
   org y:$0
input2 dc $234
result ds 1

;Program
   org p:$0

   move x:input1,x0
   move y:input2,a
   add  x0,a
   move a,y:result
   jmp  *  ;just stay here until user halts
```

```
;End of Program #1
```

4.1.2 Program 2

Using Program 1, modify it so that it multiplies two numbers.

4.1.3 Program 3

Using Programs 1 and 2, write a program that will compute the following

$$\$001234 \times \$002345 + \$003456 \times \$004567.$$

4.2 PASS.ASM–Audio Pass Through

The starting point for programming DSPs will be a program typically called `pass.asm`. This program, which simply passes samples from the codec to the DSP and back, is included with virtually all DSP evaluation modules and development boards. The `pass.asm` program is our starting point since all sample processing code will lie somewhere between the code which passes samples from the codec to the DSP and the code which passes samples from the DSP back to the codec.

For the DSP5630xEVM, this program is actually a collection of several files which we collectively call "Motorola Pass Pack." The Motorola Pass Pack for the DSP56302EVM is comprised of the following files: `pass.asm`, `ada_init.asm`, `ada_equ.asm`, `intequ.asm`, `ioequ.asm`, and `vectors.asm` which are listed in Appendix A. In addition, these files can be found at

`http://www.ece.nmsu.edu/~pdeleon/Teaching/EE592/SampleMotorolaCodes.html`

You can confirm operation of evaluation module by first assembling `pass.asm` and downloading it to the EVM. To conduct a loop back test, feed an audio signal into the EVM's audio-in jack (left jack). Listen to the output by monitoring the audio-out jack (right jack) with amplified speakers or the headphone jack (middle jack) with headphones.

In order to make the Motorola Pass Pack more flexible for our needs, we have made several modifications to the code included with the EVM. The resulting files (`pass.asm`, `ada_init.asm`, `ada_equ.asm`, `intequ.asm`, `ioequ.asm`, `vectors.asm`, `pass.dat`, `proginit.asm` and `procster.asm`) are collectively called the "Modified Pass Pack" (MPP) and listed in Appendices A and B. In addition, they can be found at

`http://www.ece.nmsu.edu/~pdeleon/Teaching/EE592/SampleMotorolaCodes.html`

Rule 2: *All* real-time DSP codes in this text will use the Modified Pass Pack (MPP). ■

In next sections we provide a general overview of the files comprising the MPP for the DSP56302EVM beginning with `pass.asm` .

4.2.1 PASS.ASM

Lines 1–23 At the top of `pass.asm` we place a descriptive header which is to be filled out for each project. In this header is an area to define variables and associated addresses. You are strongly encouraged to carefully document your code in the header.

Line 26 includes the file `ioequ.asm` which contains equate statements for the DSP56302's I/O registers and ports. All of the mnemonics and values can be found in the *DSP56302 User's Manual*.

Line 27 includes the file `intequ.asm` which contains equate statements for the DSP56302 interrupts. All of the mnemonics and values can be found in the *DSP56302 User's Manual*. This file also establishes the base address for the interrupt vector table, `I_VEC`.

Line 28 includes the file `ada_equ.asm` which initializes constants to facilitate initialization of the codec. Complete information regarding these constants can be found in the *Crystal Semiconductor CS4215 Data Sheet*.

Line 29 includes the file `vectors.asm` which lists instructions in the vector table to execute upon interrupt. We will be mostly concerned with interrupts generated with putting/getting samples to/from the codec. Instructions associated with codec interrupts and found on lines 84–94 in `vectors.asm` are

```
jsr ssi_rx_isr   ;- ESSI0 Receive Data
jsr ssi_rxe_isr  ;- ESSI0 Receive Data w/ Exception Status
jsr ssi_rxls_isr ;- ESSI0 Receive last slot
jsr ssi_tx_isr   ;- ESSI0 Transmit Data
jsr ssi_txe_isr  ;- ESSI0 Transmit Data w/ Exception Status
jsr ssi_txls_isr ;- ESSI0 Transmit last slot
```

Code for the interrupt service (sub)routines (ISRs), `ssi_rx_isr`, `ssi_tx_isr`, etc... can be found in `ada_init.asm` which we will discuss shortly.

Lines 35 We include the file `pass.dat` which is described shortly.

Line 40 instructs the assembler to place our code in program memory beginning at address `p:$100`.

Lines 39–48 establish several basic operational parameters of the DSP including clock speed, memory wait states, interrupt masks, initialization of the hardware stack, operating modes, and software stack pointer.

External Memory In order to access external memory some steps must be taken. These steps are outlined in Section 4.3 in the*DSP56302EVM User's Manual* and assume mask identification numbers (middle line on the DSP56302 part) of F90S and higher. These parts include an address priority disable (APD). We define Address Attribute Registers (AARs) for areas 0 and 3 in `pass.dat` so that the external memory is partitioned into two 16K banks for `x` and `y` memory and the Address

Attribute 3 (AA3) pin acts as a switch, toggling between the upper and lower halves of external memory. These partitions each have a base address of `$010000`. This implies that the only available memory is on-chip between addresses `$000000`–`$001BFF` (assuming the default memory map `SC=0`, `MS=0`, and `CE=0` in the `OMR`) and off-chip between addresses `$010000`–`$013FFF`. Memory addresses between `$001C00`–`$00FFFF` are unavailable. Figure 4.1 illustrates the memory map.

DSP56302EVM Memory Map

Address	X-Memory	Y-Memory
`$013FFF` `$010000`	off-chip (16K)	off-chip (16K)
`$00FFFF` `$001C00`	Unavailable	Unavailable
`$001BFF` `$00000A`	on-chip (7K)	on-chip (7K)
`$000009` `$000000`	Reserved	on-chip

Figure 4.1: Memory map for DSP56302EVM.

To complete the external memory configuration we set `OMR` bits to 0 except for bit 14 which is set to 1. Among other things, this configuration sets the default on-chip memory map to 20K of program RAM (`p` memory), 7K of X data RAM (`x` memory), and 7K of Y data RAM (`y` memory). Bit 14 is set to enable the address priority disable. Finally, we must physically set the jumper, J9 on the DSP56302EVM to connect pins 1 and 2.

Software Stack We note that the address register, `r6` is used to point to the software stack. The ISRs use the stack in their operation and thus you must *never* use `r6` except for stack operations. Misusing `r6` will cause the ISRs to not perform correctly.

Line 50 executes the subroutine `ada_init` (contained in the included file `ada_init.asm`). This subroutine initializes the codec using some of the control words defined in `pass.dat`.

Lines 52–54 Depending on the value of `ADA_OFF` line 53, which masks interrupts and disables the codec, is conditionally assembled into the code. This can be useful for debugging code. `ADA_OFF` is defined in `pass.dat`.

Line 56 We initialize our program with the subroutine contained in `proginit.asm`.

Lines 58–64 Final initialization of the codec includes moving additional control words to the codec as well as clearing storage for received sample values.

Line 66–77 form the main event loop. We begin with the label, `main_loop` defining the beginning of the main event loop. In this loop we receive right and left samples from the codec, execute the

sample processing subroutine `process_stereo`, and transmit right left output samples. We then return to the beginning of the main event loop.

The first two instructions in `main_loop`, `jset` determine the beginning of a new sample period. While waiting for a "frame sync to pass," interrupts are generated to get/put samples and control words from/to the codec. The ISRs are executed in response to these various interrupts.

Line 79–81 We include three files called `ada_init.asm`, `proginit.asm` and `procster.asm`. The file `ada_init.asm` contains the codec initialization routine executed in Line 50 as well as the ISRs. The file `proginit.asm` contains the initialization routine executed in Line 56 and the file `procster.asm` contains the student-written signal processing subroutine called in Line 73.

Note that you may place code (subroutines) in separate files to aid in code organization. In addition there may be several subroutines within any file. If written properly, these files can be reused in other projects. Normally these files are placed after line 81.

DSP5630*x* Clock Speed

The DSP56300 core features a phase-locked loop (PLL) clock oscillator in its central processing module. The PLL allows the processor to operate at a high internal clock frequency using a low frequency clock input. This feature offers two immediate benefits: lower frequency clock input reduces the overall electromagnetic interference generated by the system and the ability to oscillate at different frequencies reduces costs by eliminating the need to add additional oscillators to a system.

The DSP56302EVM has a 16MHz clock which can be multiplied internally to operate the DSP at rates higher than 16MHz. The DSP56302 on the EVM is rated at 66MHz which implies that we need to multiply the PLL by a factor of 4. Overclocking the DSP beyond its rated clock speed could result in serious damage to the part. Line 39 in `pass.asm`

```
movep #$040003,x:M_PCTL ; set PLL for MPY of 4X
```

performs this PLL multiplication function. The PLL control register (PCTL) directs the operation of the on-chip PLL. The PCTL control bits are defined in Section 9.2.3 of the *DSP56300 Family Manual*. Several of these control bits are the multiplication factor bits which will multiply the input clock frequency according to Table 9-1 in *DSP56300 Family Manual*.

4.2.2 PASS.DAT

The `pass.dat` file defines labels used in the program and lays out data memory.

Line 8 establishes the `ADA_OFF` variable used to disable the codec. When this variable is set to 1, code (lines 71–73 in `pass.dat`) which masks interrupts is conditionally assembled.

Lines 9–10 define the AARs required for our configuration of external memory.

Line 41 defines the starting point where the user can lay out 7K of internal `x` memory. We note the lower address of `$00000a` since the first ten (`$000000`–`$000009`) words in `x` memory are used by the code and are thus reserved (see Figure 4.1). The upper address for internal `x` memory is `$001BFF`.

Line 59 defines the starting point where the user can lay out 16K of external `x` memory. We note a lower address of `$010000` and an upper address of `$013FFF`. Addresses `$001C00`–`$00FFFF` are unavailable.

Line 65 defines the starting point where the user can lay out 7K of internal `y` memory. We note a lower address of `$000000` and an upper address of `$001BFF`.

Line 78 defines the starting point where the user can lay out 16K of external `y` memory. We note a lower address of `$010000` and an upper address of `$013FFF`. Addresses `$001C00`–`$00FFFF` are unavailable.

4.2.3 PROGINIT.ASM

The `proginit.asm` file contains developer initialization code such as that used to clear or initialize memory.

4.2.4 PROCSTER.ASM

The `procster.asm` file contains the code for the actual signal processing.

4.2.5 ADA_EQU.ASM

The `ada_equ.asm` file contains labels to facilitate codec initialization and operation. One modification that developers may wish to make is the sample rate of the codec. Control words for the various sample rates that the codec is capable of are defined in this file. These lines read

```
SAMP_RATE_48 equ $003000
⋮
SAMP_RATE_8 equ $000000
```

In order to change the sample rate using one of the defined control words, we need to edit the line

```
CTRL_WD_12 equ NO_PREAMP+HI_PASS_FILT+SAMP_RATE_48+STEREO+DATA_16 ;CLB=0
```

by replacing `SAMP_RATE_48` (f_s = 48kHz) with another sample rate control word. The available control words are `SAMP_RATE_32`(f_s = 32kHz), `SAMP_RATE_27` (f_s = 27kHz), `SAMP_RATE_16` (f_s = 16kHz), `SAMP_RATE_9` (f_s = 9.6kHz), and `SAMP_RATE_8` (f_s = 8kHz). Then assemble, download, and run.

Get familiar with the sounds as you alter code. Clearly the lower the sampling rate, the lower the bandwidth of the signal (according to Nyquist theory). One of the most important lessons you will learn from this course is evaluating code from the sounds it produces.

4.2.6 ADA_INIT.ASM

To be completed.

4.2.7 INT_EQU.ASM

To be completed.

4.2.8 IOEQU.ASM

To be completed.

4.2.9 VECTORS.ASM

To be completed.

4.3 Overview of CS4215 Codec Operation

The CS4215 operates in either a Control Mode or Data Mode. Furthermore, data is moved to/from the codec from/to the SSI (Synchronous Serial Interface) of the DSP56302 in 64 bit frames. Each frame is partitioned into eight 8-bit time slots (TS). TS descriptions for each mode are given on p. 24 of the CS4215 Data Sheet. Entering control mode for the CS4215 allows us to configure the chip by building the frame in the DSP and then transmitting it to the codec. Information contained in the frame (p. 18–20 in the CS4215 Data Sheet) includes such things as output level, quantizer information, and sample rate. After the CS4215 is configured, it enters the Data Mode. Once again, data is communicated to and from the codec in a frame composed of eight 8-bit TS. The description of the frame in data mode is given on p. 21–24 of the CS4215 Data Sheet.

During each sample period, four interrupts are generated by the codec after each frame sync. After each of these interrupts, two TSs are moved to/from the codec from/to the DSP. The frame is built by reserving four contiguous 24 bit words of memory and writing data into memory, two TS at a time and zero padding the remaining 8 LSBs. A pointer to this memory is established and each word is read off one interrupt at a time followed by a pointer increment.

4.4 Real-Time Digital FIR Filters

The purpose of this section is to introduce you to real-time DSP code. In this section you will code several FIR filter programs.

4.4.1 Program 4

At the heart of many DSP functions such as difference equations is the inner product defined in (1.1). The inner product when used in DSP is typically formed between a vector of coefficients (feedforward or feedback) and a vector of samples (input or output). The $N \times 1$ vector of input samples usually contains the current and previous $N - 1$ input samples and is updated each sample period. One can employ a circular queue (buffer) to store the samples and use address pointers to navigate through the queue. We can setup a pair of circular queues (left channel and right channel) in `pass.asm` as follows.

1. In `pass.dat`, add the following lines in the appropriate places

```
N equ 128    ;length of circular queue
 org x:$00000a
RQUEUE dsm N ;alloc and init N words for right channel modulo queue
 org y:$000000
LQUEUE dsm N ;alloc and init N words for left channel modulo queue
```

2. Replace the `NOP` in `proginit.asm` with

```
;Initialize pointers to queues and set up modulo addressing
move #RQUEUE,r0 ;point to right channel input sample queue
move #N-1,m0    ;set up modulo addressing for right queue
move #LQUEUE,r4 ;point to left channel input sample queue
move #N-1,m4    ;set up modulo addressing for left queue
move #$0,x0     ;now clear out queues
rep #N
move x0,x:(r0)+
rep #N
move x0,y:(r4)+
```

3. Assuming the output samples are stored in accumulators `a` and `b`, make the following changes to the associated lines in the main event loop

```
move a,x:TX_BUFF_BASE   ;transmit right
move b,x:TX_BUFF_BASE+1 ;transmit left
```

4. Replace the `NOP` in `procster.asm` with

```
move x:(r0),a   ;copy oldest right channel sample to a
move y:(r4),b   ;copy oldest left channel sample to b
move x0,x:(r0)+ ;store current right channel sample in right queue
move y0,y:(r4)+ ;store current left channel sample in left queue
```

Run the program and make sure it passes audio on both channels. Note that the samples are delayed by N sample periods before being passed to the codec for reconstruction. However, you cannot hear this delay since there is no absolute reference.

You could form a simple echo by adding the oldest sample with the newest:

```
move x:(r0),a   ;copy oldest right channel sample to a
move y:(r4),b   ;copy oldest left channel sample to b
add x0,a        ;add current right channel sample to oldest right channel sample
add y0,b        ;add current left channel sample to oldest left channel sample
asr a           ;scale in case we've overflowed
asr b           ;scale in case we've overflowed
move x0,x:(r0)+ ;store current right channel sample in right queue
move y0,y:(r4)+ ;store current left channel sample in left queue
```

4.4.2 Program 5

This program filters the right channel with the FIR filter defined below.

1. In `pass.dat`, add the following lines in the appropriate places

```
N equ 10       ;length of circular queue
 org x:$00000a
RQUEUE dsm N   ;allocate N words for right channel modulo queue
 org y:$000000
RCOEFS dsm N   ;allocate N words for right channel coefficients
 org y:RCOEFS
 dc 0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1 ;simple MA filter
```

2. Replace the `NOP` in `proginit.asm` with

```
;Initialize pointers to queues and FIR coefficients and set up modulo addressing
move #RQUEUE,r0 ;point to right channel input sample queue
move #N-1,m0    ;set up modulo addressing for right queue
move #RCOEFS,r5 ;point to right FIR coefficients
move #N-1,m5    ;set up modulo addressing for right coefficients
move #$0,x0     ;now clear out input queue
rep #N
move x0,x:(r0)+
```

3. Assuming the right filter output is stored in accumulator `a` and the unfiltered left sample is passed through, make the following changes to the associated lines in the main event loop

```
move a,x:TX_BUFF_BASE    ;transmit right
move y0,x:TX_BUFF_BASE+1 ;transmit left
```

4. Replace the `NOP` in `procster.asm` with FIR filter code

```
clr  a  x0,x:(r0)+  y:(r5)+,y1 ;replace oldest right sample w/current, get 1st coef
rep #N-1
mac  x0,y1,a   x:(r0)+,x0   y:(r5)+,y1
macr x0,y1,a   (r0)-
```

Run the program and make sure it passes audio on both channels. Listen for slight high frequency attenuation on the right channel. You may wish to design a better FIR filter and use those coefficients instead of the MA coefficients.

4.4.3 Program 6

Modify Program 5 to filter both left and right channels. Consider laying variables out in memory as follows

```
N equ 10 ;length of circular queue (should be <= 256 or modify ORG:$0100)
 org x:$000a
RQUEUE dsm N ;allocate N words for right channel circular queue
LCOEFS dsm N ;allocate N words for left channel coefficients
 org x:LCOEFS
 dc 0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1 ;simple MA filter

 org y:$0000
LQUEUE dsm N ;allocate N words for left channel circular queue
RCOEFS dsm N ;allocate N words for right channel coefficients
 org y:RCOEFS
 dc 0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1 ;simple MA filter
```

Now use `r5`, `r0` to point to right coefficients, right queue, respectively and modify Steps 2, 3, and 4 of the previous program for the stereo case. How many instruction cycles does your `process_stereo` routine take?

4.4.4 Program 6 Solution

1. In `pass.dat`, we have the following lines

```
N equ 10 ;circular queue length
 org x:$00000a
RQUEUE dsm N ;allocate N words for right channel circular queue
LCOEFS dsm N ;allocate N words for left channel coefficients
 org x:LCOEFS
 dc 0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1 ;simple MA filter

 org y:$000000
LQUEUE dsm N ;allocate N words for left channel circular queue
RCOEFS dsm N ;allocate N words for right channel coefficients
 org y:RCOEFS
 dc 0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1 ;simple MA filter
```

2. Replace the `NOP` in `proginit.asm` with

```
 ;Initialize pointers to queues and FIR coefficients and set up modulo addressing
 move #RQUEUE,r0 ;point to right channel sample queue
 move #N-1,m0    ;set up modulo addressing for right queue
 move #RCOEFS,r5 ;point to right FIR coefficients
```

```
move #N-1,m5    ;set up modulo addressing for right coefs

move #LQUEUE,r4 ;point to left channel sample queue
move #N-1,m4    ;set up modulo addressing for left queue
move #LCOEFS,r1 ;point to left FIR coeffients
move #N-1,m1    ;set up modulo addressing for left coefs

move #$0,x0     ;now clear out input queues
move #$0,y0
rep #N
move x0,x:(r0)+ y0,y:(r4)+
```

3. Assuming the right, left filter output is stored in accumulator `a`, `b` respectively, make the following changes to the associated lines in the main event loop

```
move a,x:TX_BUFF_BASE   ;transmit right
move b,x:TX_BUFF_BASE+1 ;transmit left
```

4. Replace the `NOP` in `procster.asm` with

```
 clr a   x0,x:(r0)+   y:(r5)+,y1   ;right channel
 clr b   y0,y:(r4)+   x:(r1)+,x1   ;left channel

 ;FIR filter loop
 do #N-1,filter_loop
 mac  x0,y1,a   x:(r0)+,x0   y:(r5)+,y1
 mac  y0,x1,b   y:(r4)+,y0   x:(r1)+,x1
filter_loop
 macr x0,y1,a   (r0)-
 macr y0,x1,b   (r4)-
```

Run the program and make sure it passes audio on both channels. Listen for high frequency attenuation on both channels. Note that `process_stereo` takes $2N + 2$ instruction cycles which is about as fast as you can get for dual channel FIR filtering when you include clearing of accumulators, memory moves, and rounding.

4.5 Real-Time Digital IIR Filters

The following sample codes implement an IIR filter in Cascade Form. We choose this realization due to the fixed-point nature of the DSP5630x.
(To be completed)

4.6 Information Sources for DSP5630x Development

Most real-time DSP applications are constructed by taking portions of other codes and modifying them for your needs. Many sources of codes and other information are available on the Internet.

Some useful sources are given below.

- Motorola's Dr. BuB contains a wealth of useful subroutines compatible with the DSP5630x processor

  `ftp://ftp.funet.fi/pub/ham/dsp/dr.bub/`

- The USENET group `comp.dsp` has many threads on DSP hardware, development tools, and code resources.

4.7 Problems

1. Go through the tutorial in the user's manual for `Debug-56K`.

2. Read details on other assembler directives `Assembler 56300 Reference Manual`.

3. Read instruction set descriptions in Appendix A of `DSP56300 Family Manual`. Highlight all the descriptions and place bookmarks on `MOVE`, `Jcc`, as well as the status register (SR) format.

Chapter 5

Sound Field Simulator

5.1 Background

This chapter discusses the theory and algorithm for the implementation of a sound field simulator. The sound field simulator processes audio signals so as to give the acoustic impression of a much larger and acoustically richer environment such as a concert hall.

5.2 Concert Hall Acoustics

5.2.1 Sound Propagation

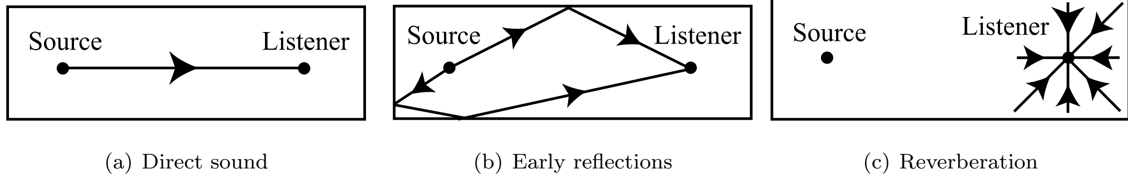
> ***Definition:*** An environment in which the sound intensity varies as $1/r^2$ (r is the distance from the source) is called a *free field*.

Example: A free field exists when a source of sound is very small (a point source) and is located outdoors away from reflecting objects. ■

Sound waves travel away from the source in all directions and hence we have spherical wave fronts. In a free field, the pressure is halved when the distance r doubles (except at high frequencies since air is a nonlinear medium). Free field conditions rarely occur indoors, except in reflection-free anechoic rooms. Indoors, sound travels only short distances before encountering walls and other obstacles. These obstacles reflect and absorb sound in ways that largely determine the acoustic properties of the hall.

5.2.2 Direct Sound, Early Reflections, and Reverberation

The speed at which sound travels is a function of air pressure and frequency of the sound wave. Under typical conditions, sound travels at $c = 344$m/s or about 1000ft/s.



(a) Direct sound (b) Early reflections (c) Reverberation

Figure 5.1: Sound propagation in an enclosed environment consists of (a) direct sound, (b) early reflections, and (c) reverberation.

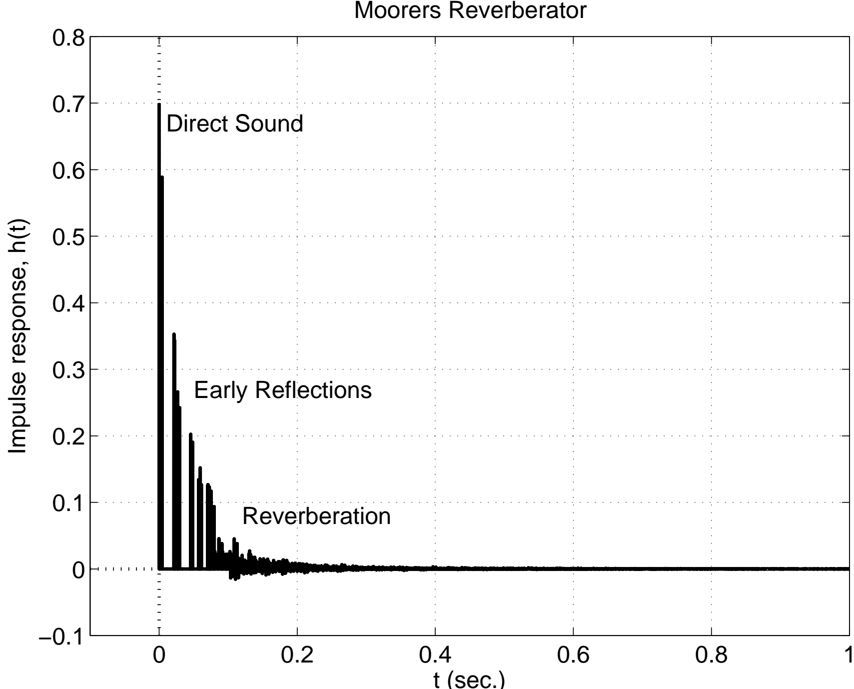


Figure 5.2: Room impulse response.

The direct sound, illustrated in Figure 5.1(a), takes the straight path (approximately 7–70m) from source to listener and reaches the listener in 20–200ms (depending on the distance) after initiation. A short time later, the same sound will reach the listener after reflecting from walls and ceiling. The first group of these reflections to arrive at the listener within 50ms of the direct sound are referred to as the "early reflections." These are illustrated in Figure 5.1(b). After the early reflections, reflected sounds arrive "thick and fast" from all directions as in Figure 5.1(c). These reflections become smaller and closer together, merging into what is called "reverberant sound" which can last up to a couple of seconds. By carefully studying the direct sound, early reflections, and reverberation, a simple but often accurate analysis of the acoustics of a hall can be obtained. A sample impulse response for such an acoustic environment is given in Figure 5.2.

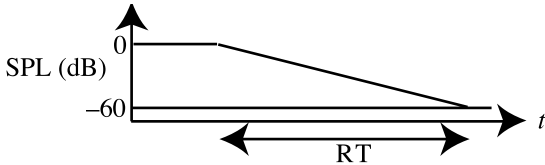


Figure 5.3: SPL vs. Time.

Direct Sound and Early Reflections

> ***Definition:*** A concert hall is considered *intimate* (one of several subjective attributes) if the delay between the direct and first reflections is less than 20ms. Typically a small hall or chamber is acoustically intimate.

If the hall has the traditional rectangular (shoebox) shape, the first reflections will come from the nearest side wall (although those in the center may receive first reflections from the ceiling). If these early reflections do come from the lateral directions, they appear to broaden the source and thus increase the apparent source width (ASW). Studies have shown a high preference for concert halls with ceilings sufficiently high so that the lateral reflections reach the listener before the overhead reflections.

Lateral reflections are easy to achieve at listeners' positions in the center of the main floor of a "shoebox" (parallel side walls and a horizontal ceiling) hall. In a fan-shaped hall the lateral reflections are directed to the rear seats. One measure of ASW is called the "interaural cross-correlation coefficient" (IACC). This is a measure of the degree of dissimilarity of the musical sound that reach the two ears. The less similar the musical sound, the lower the IACC and the greater the ASW.

> ***Definition:*** If the total energy from lateral reflections is greater than the total energy from overhead reflections, the hall takes on a desirable *spatial impression* (SI). SI is another subjective attribute.

Due to economic reasons, fan-shaped halls are predominant since they allow more seats. However, this design has been shown to significantly reduce the SI. One proposal is the reverse fan-shaped hall which computer models have shown to have SI similar to the shoebox hall.

Reverberation

> ***Definition:*** The reverberation time (RT) is defined as the time required for the sound pressure level (at 500Hz) to decrease by one millionth or 60dB of its initial value (Figure 5.3).

Reverberant sound is most pleasant if the listener hears it coming from all directions. This is sometimes referred to as "listener envelopment" (LEV). Before the 19th century music tended to be written for the performance in existing rooms. For example

- Organ music (e.g. Bach) was written for churches which, being large and lacking soft furnishings, had very long RTs
- Chamber music (e.g. Mozart) was written for performances in well furnished private houses and emphasis could be placed on clarity (another subjective attribute defined below).

5.2.3 Concert Hall Attributes

Modern concert halls are now designed to suit the style of music for which it is intended. One trend (although not terribly perfect) is concert halls with adjustable acoustics. A number of studies have been made of various concert halls. Beranek found 18 subjective attributes of musical-acoustic quality that can be related to concert hall acoustics. Three—intimacy, spacial impression, and clarity—have been given. The others are listed below.

Liveliness—this is related primarily to the RT for middle and high frequencies (500Hz and higher). The optimum RT depends on size and function. A hall with insufficient RT is termed "dry."

Warmth—this is related to liveliness and fullness of bass tone. RT time at 250Hz and below should be somewhat longer than at middle and high frequencies. A "warmth" measurement is meaningful only when the hall is fully occupied because an audience will absorb high frequencies more than low frequencies.

Loudness of direct sound—the hall should be designed so that no listener is seated too far from the sound source. If the hall is too large, sound amplification may be necessary.

Reverberant sound level—the level of the reverberant sound, which will be the same throughout the hall, depends on the power of the source and the RT. It is defined as the total sound energy that reaches the listener.

Definition or clarity—the level of the early plus direct sound should be greater than the reverberant sound level at all locations.

Diffusion or uniformity—good spatial distribution of the sound is achieved by diffuse or irregular reflecting surfaces, and by the avoidance of focused sound of sound shadows.

Balance and blend—this depends on the stage design. If the stage is wider than 15m, the ceiling should be low ($\leq$ 10m) and irregular in shape.

If the sound energy were uniformly distributed throughout the room, the decay rate would be exponential. Reverberation time at mid-frequency (500Hz) gives a fair indication of the "liveliness" of the hall. Too great a reverberation may result in a loss of clarity. The sound energy stored in the room depends on the power of the source and the volume of the room. The rate at which that energy is absorbed depends on the area of all surfaces and objects in the room and their absorption coefficients. The total absorption may be computed as

$$A = \sum_i S_i a_i \tag{5.1}$$

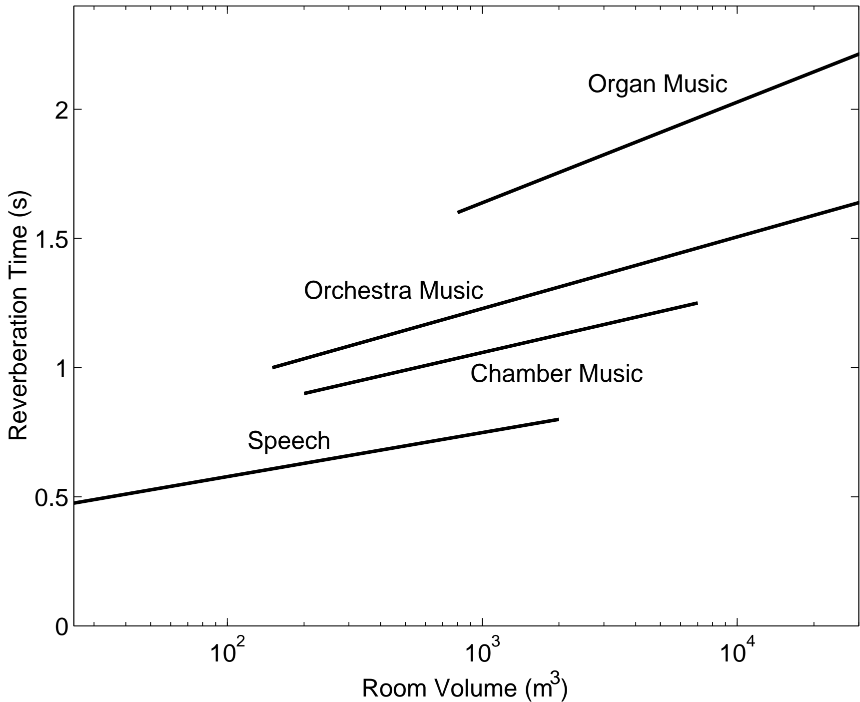


Figure 5.4: Reverberation times as a function of room volume.

where S_i is the area of the reflecting surface in m^2 and a_i is the absorption coefficient. Absorption coefficients are frequency dependent.

Example: Unpainted concrete at 500Hz has $a = 0.31$ while an audience in upholstered seats has $a = 0.56$. ■

> ***Definition:*** RT is given by
>
> $$RT = KV/A \tag{5.2}$$
>
> where $K = 0.161$ s/m, V is the volume in cubic meters, and A is the total absorption given by (5.1).

In general, larger halls have longer RTs than smaller halls. The optimum RT is a compromise between clarity (short RTs), sound intensity (high reverberant levels), and liveliness (long RTs). Sample RTs as a function of Room Volume are illustrated in Figure 5.4.

Example: Symphony Hall in Boston, MA. built 1900 and considered the finest hall in the U.S. has

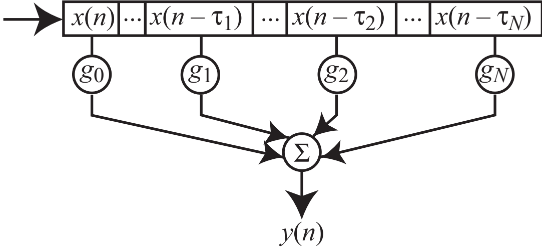


Figure 5.5: Tapped delay line.

the following parameters

$$\begin{aligned} V &= 18,740m^3 \\ A &= 1550m^2 \\ \text{Typical delay for direct sound} &= 15ms \\ \text{Number of Seats} &= 2630 \\ RT(125Hz) &= 2.2s \\ RT(500Hz) &= 1.8s \\ RT(2000Hz) &= 1.7s \end{aligned}$$

■

5.3 Digital Synthesis of the Sound Field

In recordings, we loose all directional information. Therefore we wish to somehow create and add in the experience of being in the concert hall. Since the characteristic of the hall can be divided into the direct sound, early reflections, and reverberation, we'll examine each individually to see how we can digitally synthesize these.

5.3.1 Digital Synthesis of Early Reflections

The early reflections are simply delayed and attenuated versions of the direct sound. Since there are only a few early reflections after the direct sound, we have the possibility of using something similar to an FIR filter. A tapped delay line (TDL) is a special type of FIR filter in that it assumes most of the FIR coefficients are zero. In this case, we view the TDL as in Figure 5.5 where τ_k are the tap spacings, g_k are the tap gains, and (assuming $\tau_0 = 0$)

$$y[n] = \sum_{k=0}^{N} g_k x[n - \tau_k]. \tag{5.3}$$

Tap spacings and gains for a reasonable-sounding seven tap delay line are given in Table 5.1. Note that the gain values in Table 5.1 would have to be normalized for use on a fixed-point processor.

Table 5.1: Tapped delay line parameters.

Tap	Delay (ms)	Gain
0	0.0	1.0
1	19.9	1.02
2	35.4	0.818
3	38.9	0.635
4	41.4	0.719
5	69.9	0.267
6	79.6	0.242

5.3.2 Digital Synthesis of Reverberation

The reverberation in a full-size concert hall is usually in the range of 1–3 seconds. In principle, one could extend the TDL out to this length with appropriate gains to create the reverberation. However, there are usually hundreds or even thousands of reflections during the reverberation time.

Example 1: Storage for two seconds of signal at $f_s = 48$kHz would require 96K of memory for each channel. The DSP5630xEVM has a total of 32K external memory. Internal memory varies according to the particular DSP56300 family member. For example, the DSP56302 has 7K+7K on-chip X- and Y-data memories. ■

Example 2: If we assume 96,000 reflections (tap gains) at $f_s = 48$kHz, we require a minimum of 96,000 `MAC`s per channel. Recall the upper computational bound of 1,250 instructions within a sample period. Thus the computation required for the FIR approach to reverberation is unreasonable. ■

An IIR filter provides a computationally reasonable approach to creating reverberation. The infinitely long, gradual decaying characteristic of reverberation qualitatively matches the impulse response of an IIR filter.

5.3.3 Comb Filter

In order to simulate the reverberation caused by two parallel walls, we consider a Comb Filter (CF) which is IIR. Figure 5.6 illustrates the direct form II or canonical realization for a CF whose system function is given by

$$H(z) = \frac{z^{-m}}{1 - gz^{-m}} = \frac{1}{z^m - g}. \tag{5.4}$$

(**XXX Update filter implementation with *transposed* DF II for better numerical behavior (Lyons)**)

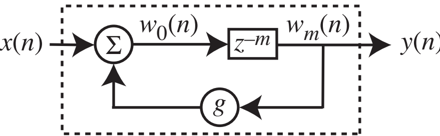


Figure 5.6: Direct form II realization of a comb filter.

From the figure, the state equations are given by

$$w_0[n] = x[n] + g w_m[n] \tag{5.5}$$

$$y[n] = w_m[n] \tag{5.6}$$

$$w_k[n+1] = w_{k-1}[n], \quad k = m, m-1, ..., 1 \tag{5.7}$$

The queue for the comb filter's internal states is illustrated in Figure 5.7. A pointer is used to access the m-th state at time n [Figure 5.7(a)] as needed in (5.5) and (5.6). We update the internal states by replacing $w_m[n]$ (since it is no longer needed) with $w_0[n] = w_1[n+1]$ and adjusting the pointer so it points to the m-th state at time $n+1$ [Figure 5.7(b)]. Since the internal states will be stored in a circular queue, implementation of (5.7) requires only that the pointer be decremented.

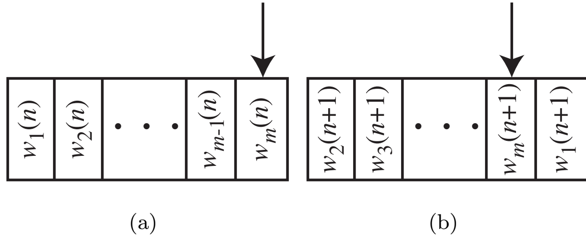


Figure 5.7: Queue for comb filter's internal states at time (a) n and (b) $n+1$.

The magnitude response plot for an example comb filter is given in Figure 5.8 (the plot also explains the reason for the name *comb filter*). From the system function, we see m poles with locations given by

$$\begin{aligned} z^m - g &= 0 \\ z_i &= \sqrt[m]{g} e^{j2\pi i/m}. \end{aligned} \tag{5.8}$$

An example pole/zero plot with $g = 0.9$ and $m = 5$ is shown in Figure 5.9. Taking the inverse z-transform of the system function in (5.4) results in the causal difference equation for the comb filter

$$y[n] = x[n-m] + g y[n-m] \tag{5.9}$$

which leads to an impulse response (assuming zero initial conditions),

$$h[n] = \delta[n-m] + g\delta[n-2m] + g^2\delta[n-3m] + \ldots. \tag{5.10}$$

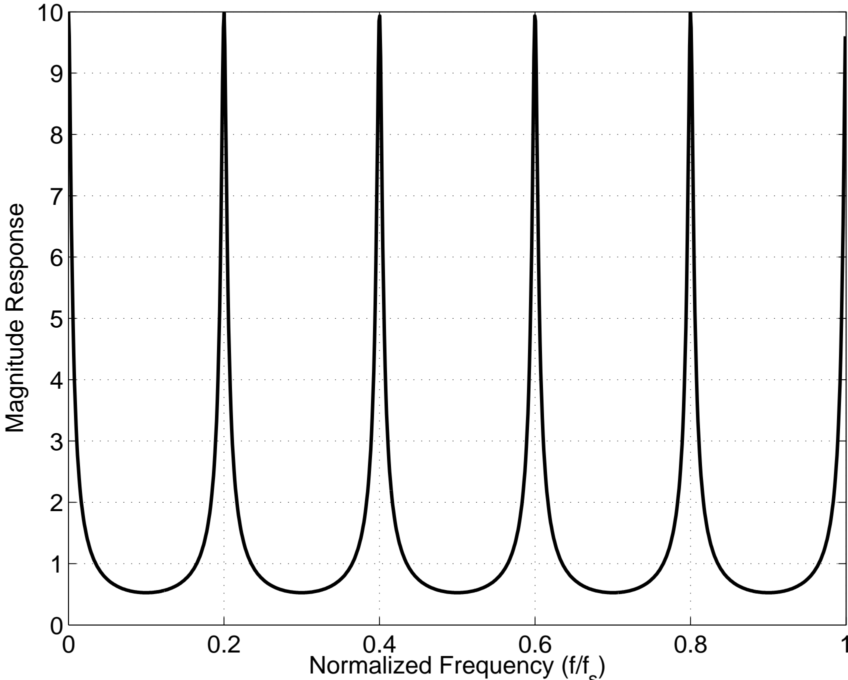


Figure 5.8: Magnitude response of comb filter with $g = 0.9$ and $m = 5$.

The comb filter has an impulse response (Figure 5.10) similar to the response between two parallel walls [(Figure 5.11(a)].

Unfortunately, the impulse response of this IIR filter, a sequence of gradually decaying, regular-spaced impulses, is too regular. Real reverberation has an echo density that increases with the square of time. The echo density of this "unit reverberator" is constant. To use an IIR filter for reverberation, we must increase the density of its impulse response. We get a better effect by including another set of parallel walls [(Figure 5.11(b)] or in the implementation, using a pair of CFs in parallel [(Figure 5.12(a)].

If we add in the floor and ceiling, in the implementation we require three comb filters in parallel [Figure 5.12(b)]. If we add in yet another CF (four total) for better "echo density," we get an impulse response which is closer to a hall's but still not good enough. A common approach to increasing the density of the impulse response is to combine several unit-reverberators. The parallel unit-reverberators should have delay times that are incommensurate with each other so that the resonances do not coincide. Comb filter parameters (delays and gains) are given in Table 5.2. Note the comb filter gains are computed assuming a 48 kHz sample rate.

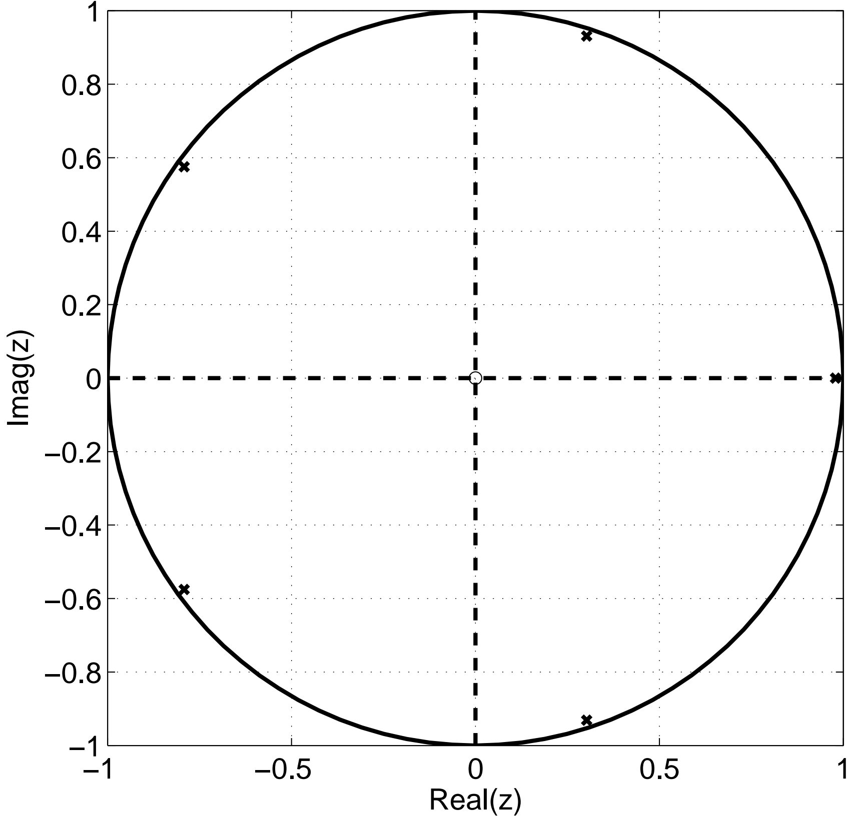


Figure 5.9: Pole and zero locations of comb filter with $g = 0.9$ and $m = 5$.

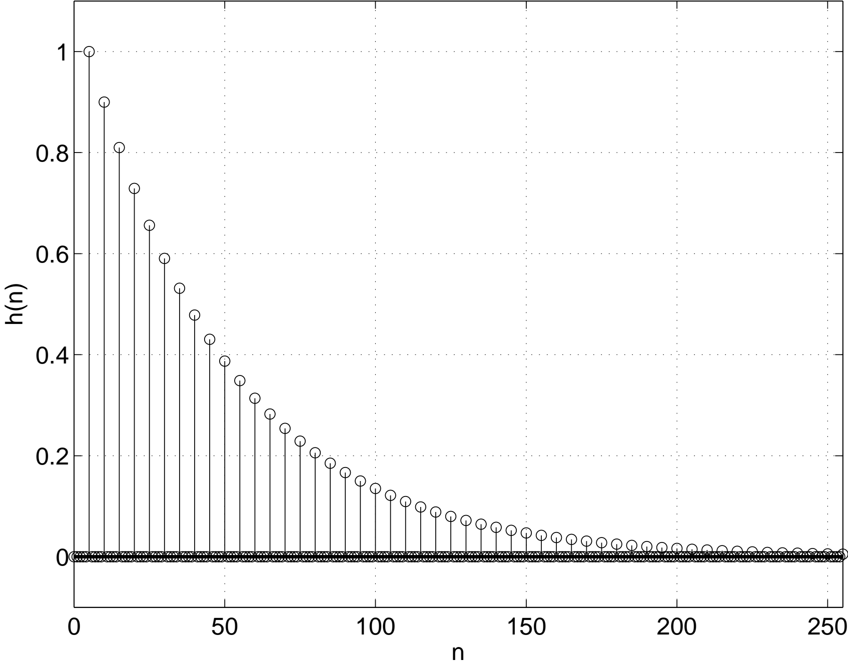
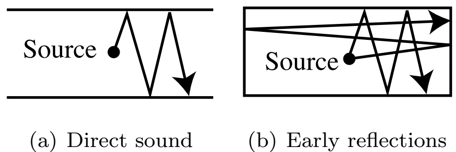
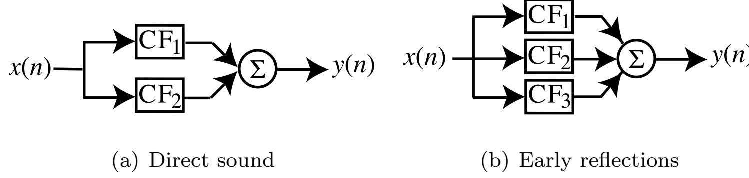


Figure 5.10: Impulse response of comb filter with $g = 0.9$ and $m = 5$.



(a) Direct sound (b) Early reflections

Figure 5.11: Resonance between (a) two parallel walls and (b) two pairs of parallel walls.



(a) Direct sound (b) Early reflections

Figure 5.12: (a) Two parallel comb filters (CFs) and (b) three parallel CFs.

Table 5.2: Comb filter parameters.

Comb Filter	Delay (ms)	g
1	50	0.4635
2	61	0.4316
3	72	0.4077
4	78	0.3918

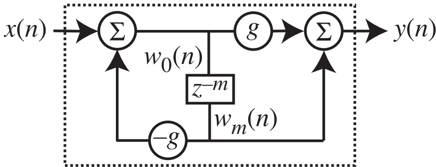


Figure 5.13: All-Pass Filter.

5.3.4 All-Pass Filters

In order to further increase echo density, we feed the output of the reverberators into a pair of all-pass filters (APFs), arranged in series. A single APF (canonical realization) is shown in Figure 5.13. The system function of the APF is given by

$$H(z) = \frac{g + z^{-m}}{1 + gz^{-m}}. \tag{5.11}$$

From the figure, the state equations are given by

$$w_0[n] = x[n] - gw_m[n] \tag{5.12}$$

$$y[n] = gw_0[n] + w_m[n] \tag{5.13}$$

$$w_k[n+1] = w_{k-1}[n], \quad k = m, m-1, ..., 1. \tag{5.14}$$

Taking the inverse z-transform of the system function in (5.11) results in the causal difference equation for the all-pass filter

$$y[n] = gx[n] + x[n-m] - gy[n-m] \tag{5.15}$$

which leads to a causal impulse response (assuming zero initial conditions),

$$\begin{aligned} h[n] = \; & g\delta[n] + (1-g^2)\delta[n-m] - g(1-g^2)\delta[n-2m] + \\ & g^2(1-g^2)\delta[n-3m] - g^3(1-g^2)\delta[n-4m] \ldots . \end{aligned} \tag{5.16}$$

An example impulse response of an all-pass filter is shown in Figure 5.14. The APF will not introduce coloration (since it has a unity-gain magnitude response) but will increase echo density by virtue of the fact that every impulse entering the APF sets off more (scaled) impulses due to the nature of the impulse response. All-pass filter parameters are 0.7 for both filter gains and 6.4, 6.7 ms filter delay for the first, second all-pass filter, respectively.

Combining the four comb filters in parallel together with the all-pass filers in series results in the reverberator structure shown in Figure 5.15 also known as Schroeder's reverberator.

5.4 Implementation

By combining the building blocks of the TDL and reverberator, we can simulate concert hall acoustics. The block diagram for the sound field simulator is shown in Figure 5.16 and the resulting

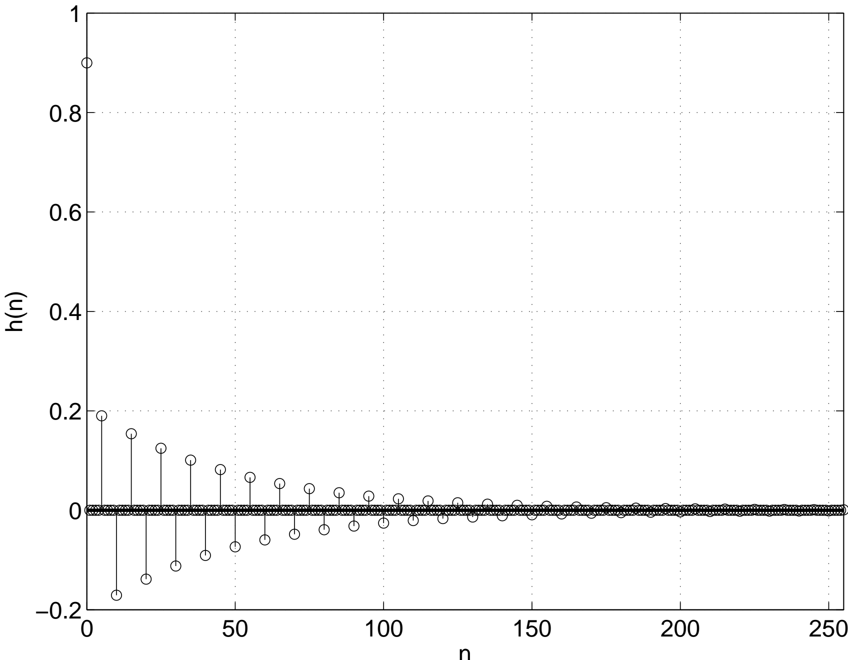


Figure 5.14: Impulse response of all-pass filter with $g = 0.9$ and $m = 5$.

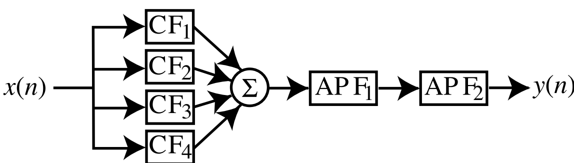


Figure 5.15: Reverberator.

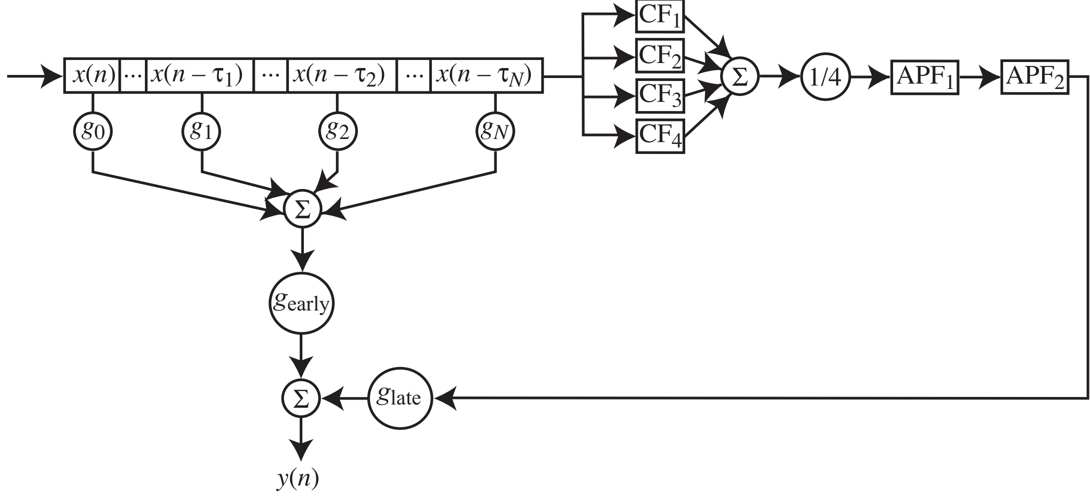


Figure 5.16: Block diagram of sound field simulator.

impulse response shown in Figure 5.2. Note that the oldest sample in the tapped delay line feeds the reverberator. Furthermore, the output of the tapped delay line and reverberator are mixed to form the final output, $y[n]$.

5.4.1 Algorithm

INITIALIZE

Setup pointers (address, modifier registers)

Clear out delay line for TDL and six queues for filter states (four CFs and two APFs)

MAIN (once per sample period)

TAPPED_DELAY_LINE

COMB_FILTER1

COMB_FILTER2

COMB_FILTER3

COMB_FILTER4

APF1_IN = (1/4)(CF1_OUT + CF2_OUT + CF3_OUT + CF4_OUT)

ALL_PASS_FILTER1

ALL_PASS_FILTER2

OUTPUT

Goto ***MAIN***

TAPPED_DELAY_LINE

[Assume Rn points to oldest sample in TDL queue, $x(n - \mathrm{DLEN})$]

CF_IN = $x(n - \mathrm{DLEN})$

Replace oldest sample with $x[n]$

TDL_OUT = 0

For $k = 0$ to NTAPS − 1

Get GAIN(k)

Nn = TIME(k) ;set offset register

Sample = (Rn+Nn) ;get delayed sample

TDL_OUT = TDL_OUT + [GAIN(k)](Sample)

End

Rn− ;point to "new" oldest sample

COMB_FILTER*k (k = 1, 2, 3, 4)*

[Assume Rn points to oldest state in CFk queue, w_m]

OLDEST_STATE = w_m

Replace oldest state with w_0 = CF_IN + (CFkG)(w_m)

CFk_OUT = OLDEST_STATE

Rn− ;point to "new" oldest state

ALL_PASS_FILTER1

[Assume Rn points to oldest state in APF1 queue, w_m]

OLDEST_STATE = w_m

Replace oldest state with w_0 = APF1_IN − (APF1G)(w_m)

APF2_IN = (APF1G)(w_0) + OLDEST_STATE

Rn− ;point to "new" oldest state

ALL_PASS_FILTER2

[Assume Rn points to oldest state in APF2 queue, w_m]

OLDEST_STATE = w_m

Replace oldest state with w_0 = APF2_IN − (APF2G)(w_m)

APF2_OUT = (APF2G)(w_0) + OLDEST_STATE

Rn− ;point to "new" oldest state

OUTPUT

OUT = (G_EARLY)(TDL_OUT) + (GLATE)(APF2_OUT)

5.4.2 Memory Layout

Project 1 requires 7 circular queues for each channel. Since modulo addressing requires special base addresses ($2^j \geq M$) where base address has lower j bits zero), we must carefully lay these out in memory to ensure proper operation. Below is a sample layout for a single channel.

SFS592.DAT

```
1   ;****************************************************************************
2   ;SFS592.DAT: This data file is used with PROJECT.ASM to lay out memory.
3   ;****************************************************************************
4
5   ;********
6   ; Equates
7   ;********
8   ADA_OFF equ     0       ;0 enables codec, 1 disables codec
9   AAR0    equ     $010931 ;split external (off-chip) 32K RAM into 16K X & 16K Y
10  AAR3    equ     $010925 ;beginning at $010000 p.4-6--4-8 DSP56302EVM UM
11
12  SRATE           equ     48000           ;Must correspond to value in ADA_INIT
13  IMP_PERIOD      equ     48000           ;number of samples between impulses
14  IMP_MAG         equ     0.99999999      ;magnitude of impulse
15
16  ;Tapped Delay Line Parameters--see p.24 Table 3a in Moorer
17  NTAPS   equ     7                       ;number of taps in TDL
18  DLEN    equ     @CVI(0.0796*SRATE+1)    ;length for 79.6ms delay (+1 for algorithm)
19  GAIN_SF equ     1.02                    ;scale factor for tap gains
20
21  ;Comb Filter Parameters--see p.18 Table 2 in Moorer
22  ;Delays
23  CF1D    equ     @CVI(0.050*SRATE)       ;length in samples for 50ms delay
24  CF2D    equ     @CVI(0.061*SRATE)       ;length in samples for 61ms delay
25  CF3D    equ     @CVI(0.072*SRATE)       ;length in samples for 72ms delay
26  CF4D    equ     @CVI(0.078*SRATE)       ;length in samples for 78ms delay
27  ;Gains
28  G       equ     0.83
29  CF1G    equ     G*(1.0-0.46*(@CVF(SRATE)/50000.0))      ;gains SRATE-scaled
30  CF2G    equ     G*(1.0-0.50*(@CVF(SRATE)/50000.0))
31  CF3G    equ     G*(1.0-0.53*(@CVF(SRATE)/50000.0))
32  CF4G    equ     G*(1.0-0.55*(@CVF(SRATE)/50000.0))
33
34  ;All Pass Filter Parameters--see p.18 in Moorer
35  ;Delays
36  APF1D   equ     @CVI(0.0064*SRATE)      ;length for 6.4 ms delay
37  APF2D   equ     @CVI(0.0067*SRATE)      ;length for 6.7 ms delay
```

```
38  ;Gains
39  APF1G   equ     0.70
40  APF2G   equ     0.70
41
42  ;Mixing Parameters
43  GEARLY  equ     0.5
44  GLATE   equ     0.9999
45
46  ;************
47  ; Data Memory
48  ;************
49  ;*****************************************************************************
50  ;---Buffer for the CS4215
51  ;       The following lines of code are required for proper on-board audio
52  ;       codec operation.
53  ;*****************************************************************************
54          org x:$000000           ;On-chip X memory $000000 -- $001BFF
55  RX_BUFF_BASE    equ     *
56  RX_data_1_2     ds      1       ;data time slot 1/2 for RX ISR
57  RX_data_3_4     ds      1       ;data time slot 3/4 for RX ISR
58  RX_data_5_6     ds      1       ;data time slot 5/6 for RX ISR
59  RX_data_7_8     ds      1       ;data time slot 7/8 for RX ISR
60
61  TX_BUFF_BASE    equ     *
62  TX_data_1_2     ds      1       ;data time slot 1/2 for TX ISR
63  TX_data_3_4     ds      1       ;data time slot 3/4 for TX ISR
64  TX_data_5_6     ds      1       ;data time slot 5/6 for TX ISR
65  TX_data_7_8     ds      1       ;data time slot 7/8 for TX ISR
66
67  RX_PTR          ds      1       ;Pointer for rx buffer
68  TX_PTR          ds      1       ;Pointer for tx buffer
69
70          org x:$00000a
71  ;*****************************************************************************
72  ;---On-chip X data goes here
73  ;*****************************************************************************
74  IMP_COUNT       dc      IMP_PERIOD
75  TAPSPACE        ds      NTAPS
76          org     x:TAPSPACE
77          dc      @CVI(0*SRATE)           ;time(0)--see p.24 Table 3a in Moorer
78          dc      @CVI(0.0199*SRATE)      ;time(1)
79          dc      @CVI(0.0354*SRATE)      ;time(2)
80          dc      @CVI(0.0389*SRATE)      ;time(3)
81          dc      @CVI(0.0414*SRATE)      ;time(4)
82          dc      @CVI(0.0699*SRATE)      ;time(5)
83          dc      @CVI(0.0796*SRATE)      ;time(6)
84
85  TAPGAIN ds      NTAPS
86          org     x:TAPGAIN
87          dc      1.000/GAIN_SF   ;gain(0)--see p.24 Table 3 in Moorer
88          dc      1.020/GAIN_SF   ;gain(1)
```

```
89          dc      0.818/GAIN_SF   ;gain(2)
90          dc      0.635/GAIN_SF   ;gain(3)
91          dc      0.719/GAIN_SF   ;gain(4)
92          dc      0.267/GAIN_SF   ;gain(5)
93          dc      0.242/GAIN_SF   ;gain(6)
94
95  APF1    dsm     APF1D   ;APF1 queue for filter states (307 words starting at x:$200)
96  APF2    dsm     APF2D   ;APF2 queue for filter states (321 words starting at x:$400)
97  CF1     dsm     CF1D    ;CF1 queue for filter states (2400 words starting at x:$1000)
98
99
100 ;*****************************************************************************
101 ;---Software Stack
102 STACK   equ     *       ;locate stack after last thing in on-chip X memory
103                         ;STACK is used in TXRX_ISR.ASM and must be allowed to grow
104
105 ;You must not write *anything* into x:$001C00 --x:$00FFFF (check .LST file)
106
107
108         org x:$010000   ;Off-chip X memory $010000 -- $013FFF
109 ;*****************************************************************************
110 ;---Off-chip X data goes here
111 ;       If you load actual values in off-chip memory, i.e. "dc" then you must run
112 ;       pass.asm first so that off-chip memory can be configured before the load
113 ;       is attempted.   In this case, Debugger should have Reset on Load (Config
114 ;       menu) unchecked.
115 ;*****************************************************************************
116 CF2     dsm     CF2D    ;CF2 queue for filter states (2928 words starting at x:$10000)
117 CF3     dsm     CF3D    ;CF3 queue for filter states (3456 words starting at x:$11000)
118 CF4     dsm     CF4D    ;CF4 queue for filter states (3744 words starting at x:$12000)
119 TDL     dsm     DLEN    ;TDL queue for filter states (3820 words starting at x:$13000)
120
121
122         org y:$000000   ;On-chip Y memory $000000 -- $001BFF
123 ;*****************************************************************************
124 ;---On-chip Y data goes here
125 ;*****************************************************************************
126
127
128 ;You must not write *anything* into y:$001C00 --y:$00FFFF (check .LST file)
129
130
131         org y:$010000   ;Off-chip Y memory $010000 -- $013FFF
132 ;*****************************************************************************
133 ;---Off-chip Y data goes here
134 ;       If you load actual values in off-chip memory, i.e. "dc" then you must run
135 ;       pass.asm first so that off-chip memory can be configured before the load
136 ;       is attempted.   In this case, Debugger should have Reset on Load (Config
137 ;       menu) unchecked.
138 ;*****************************************************************************
```

5.5 Testing

Although listening to the output of the EVM can give some indication of whether the code is working or not, to be sure we must turn to analytic techniques. One way to rigorously test is to capture and analyze the impulse response of the sound field simulator coming off the EVM. Rather than feed an impulse to the audio input plug (which could potentially damage the EVM), we can replace the input signal coming from the A/D with an *internally* generated an impulse sequence. The audio output of the EVM can then be connected to the audio input of a PC's sound card in order to conveniently record the impulse response. The impulse response can then be plotted and compared with the simulated impulse response in Figure 5.2.

In order to easily generate an impulse train, enter the following code in a file called `impulse.asm`:

IMPULSE.ASM

```
1   impulse move    b2,x:(r6)+      ;save b and x0 on the stack
2           move    b1,x:(r6)+
3           move    b0,x:(r6)+
4           move    x0,x:(r6)+
5           clr     b               ;clear the counter
6           move    x:IMP_COUNT,b0  ;restore to current count
7           dec     b               ;decrement the count
8           jne     no_impulse      ;if not zero, then branch for no impulse
9   do_impulse
10          move    #IMP_MAG,x0
11          move    #>IMP_PERIOD,b0 ;reset count to period
12          jmp     out_impulse     ;go to exit routine
13  no_impulse
14          move    #$0,x0
15  out_impulse
16          move    x0,x:RX_BUFF_BASE
17          move    x0,x:RX_BUFF_BASE+1
18          move    b0,x:IMP_COUNT  ;save current count
19          move    x:-(r6),x0      ;restore b and x0 from stack
20          move    x:-(r6),b0
21          move    x:-(r6),b1
22          move    x:-(r6),b2
23          rts
24
```

Next, before line 69 of `pass.asm` (`move x:RX_BUFF_BASE,x0`) insert the following line:

```
jsr impulse
```

Finally, after line 80 of `pass.asm` (`include 'procster.asm'`) insert the following line:

```
include 'impulse.asm'
```

Note that IMP_COUNT, IMP_MAG, IMP_PERIOD are already defined in the `sfs592.dat` file.

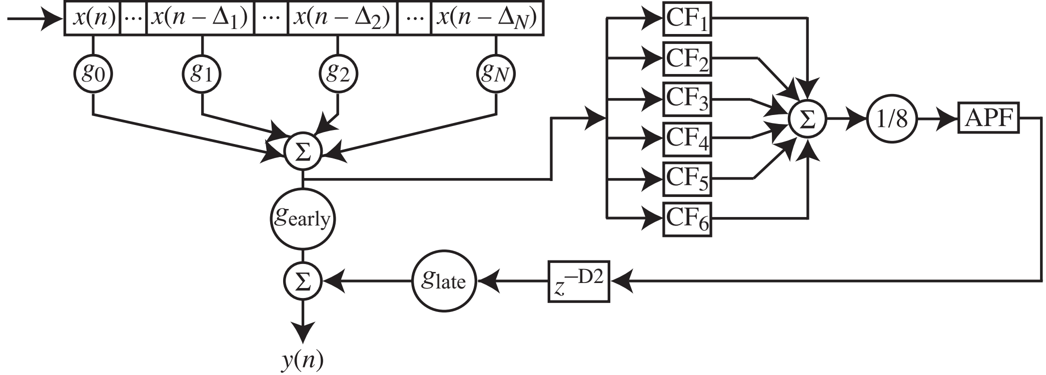


Figure 5.17: Moorer's sound field simulator.

5.6 Moorer's Sound Field Simulator

For completeness, we describe the sound field simulator based on John Moorer's paper although it is not likely the Motorola DSP5630xEVM has sufficient memory for implementations with high sampling rates. This system illustrated in Figure 5.17, is similar to the one described in Section 5.4 with the following differences:

- the tapped delay line has 19 taps
- the output of the tapped delay is fed to the reverbertor (in the previous system, the oldest sample in the delay line fed the reverberator)
- the reverberator consists of a parallel bank of six comb filters followed by a single all-pass filter
- the comb filters in the reverberator have a lowpass filter in the feedback loop
- the output of the reverberator is delayed so that the first pulse of the reverberator roughly coincides with the last pulse of the tapped delay line

5.6.1 Tapped Delay Line

The tapped delay line for Moorer's system is identical to Figure 5.5 and the parameters are given in Table 5.3. These parameters were taken from a highly idealized geometric simulation of the Boston Symphony Hall.

5.6.2 Comb Filter

Figure 5.18(a) illustrates the direct form I realization for the CF used in Moorer's sound field simulator. This CF is identical to the one previously described with the exception of $T(z)$ in the

Table 5.3: Tapped delay line parameters for Moorer's sound field simulator.

Tap	Delay (ms)	Gain	Tap	Delay (ms)	Gain
0	0.0	1.0	10	58.7	0.193
1	4.3	0.841	11	59.5	0.217
2	21.5	0.504	12	61.2	0.181
3	22.5	0.491	13	70.7	0.180
4	26.8	0.379	14	70.8	0.181
5	27.0	0.380	15	72.6	0.176
6	29.8	0.346	16	74.1	0.142
7	45.8	0.289	17	75.3	0.167
8	48.5	0.272	18	79.7	0.134
9	57.2	0.192			

feedback loop. The purpose of this additional filter in the feedback loop is to simulate the absorption of high frequencies by the air. Thus we take $T(z)$ to be a lowpass filter of the form

$$T(z) = \frac{1}{1 - g_1 z^{-1}}. \tag{5.17}$$

Inserting $T(z)$ into the feedback loop of (5.4), where now g_2 denotes the feedback gain of the comb filter, gives a system function

$$\begin{aligned} H(z) &= \frac{z^{-m}}{1 - g_2 T(z) z^{-m}} \\ &= \frac{z^{-m} - g_1 z^{-(m+1)}}{1 - g_1 z^{-1} - g_2 z^{-m}}. \end{aligned} \tag{5.18}$$

Figure 5.18(b) illustrates the direct form II realization of (5.18).

Taking the inverse z-transform of the system function in (5.18) results in the causal difference equation for the comb filter

$$y[n] = x[n-m] - g_1 x[n-m-1] + g_1 y[n-1] + g_2 y[n-m]. \tag{5.19}$$

From Figure 5.18(b), the state equations are given by

$$w_0[n] = x[n] + g_1 w_1[n] + g_2 w_m[n] \tag{5.20}$$

$$y[n] = w_m[n] - g_1 w_{m+1}[n] \tag{5.21}$$

$$w_k[n+1] = w_{k-1}[n], \quad k = m+1, m, ..., 1 \tag{5.22}$$

The queue for the comb filter's internal states is illustrated in Figure 5.19. A pointer is used to access the m and $(m+1)$-th states at time n [Figure 5.19(a)] as needed in (5.20) and (5.21). We

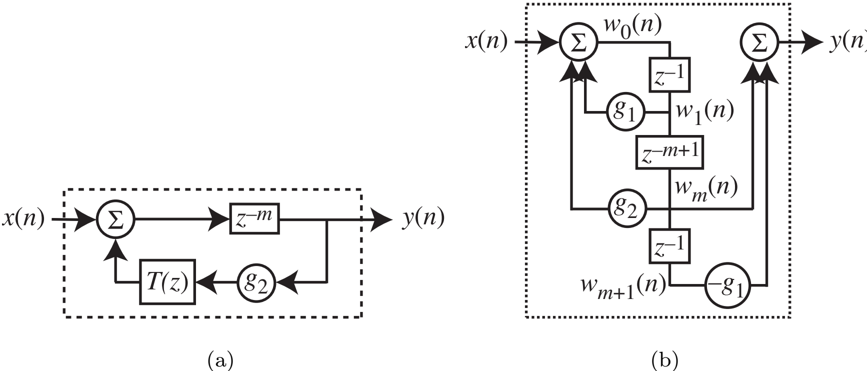


Figure 5.18: Comb filter with lowpass filter $T(z) = 1/(1 - g_1 z^{-1})$ in the feedback loop. (a) Direct form I and (b) direct form II realizations.

update the internal states by replacing $w_{m+1}[n]$ (since it is no longer needed) with $w_0[n] = w_1[n+1]$ and adjusting the pointer so it points to the $(m+1)$-th state at time $n+1$ [Figure 5.19(b)]. Since the internal states will be stored in a circular queue, implementation of (5.22) requires only that the pointer be decremented.

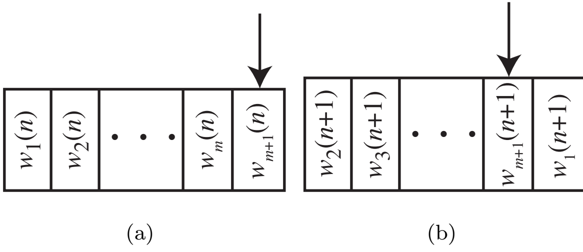


Figure 5.19: Queue for comb filter's internal states at time (a) n and (b) $n+1$.

The magnitude response plot for an example comb filter ($g_1 = 0.3$, $g_2 = 0.6$, and $m = 10$) is given in Figure 5.20 where we note the lowpass nature of the comb. The pole/zero and impulse response plots of the example comb filter are given in Figures 5.21 and 5.22

In order for the comb filter to be stable, we set

$$g_2 \quad = \quad g(1 - g_1) \tag{5.23}$$

where $g = 0.83$ (recommended value). Values for g_1 and g_2 along with the feedback delays are given in Table 5.4. Note the values listed in the table assume a sample rate of 48 kHz. For a different sample rate f_s, g_1 should be scaled (multiplied) by a factor of $f_s/48000$ and g_2 recalculated. Also note that the delay values correspond to the parameter m and not $m+1$. Finally, in order to prevent overflow on a fixed-point processor, the sum of the comb filters should be scaled by 1/8 before being fed to the APF.

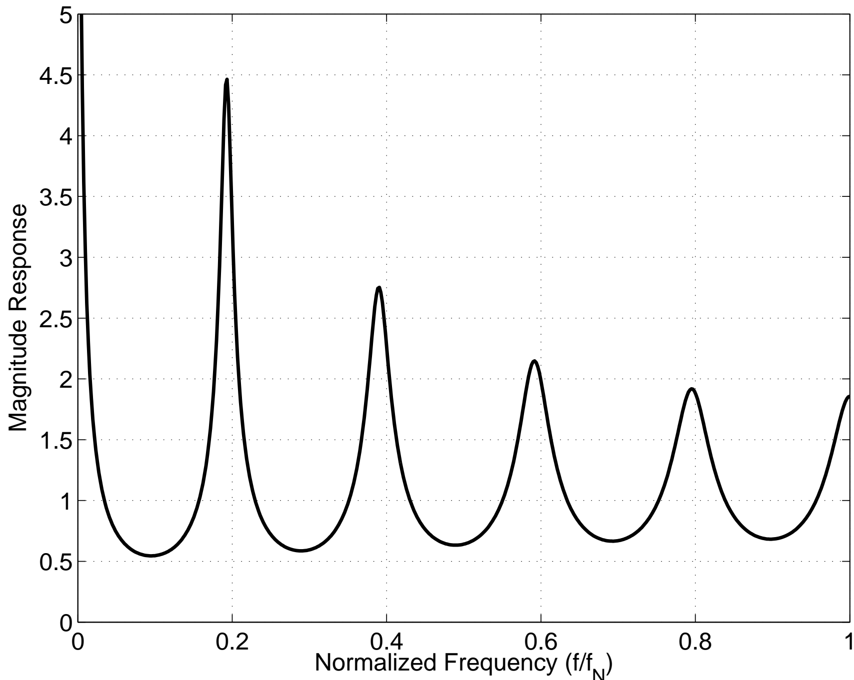


Figure 5.20: Magnitude response of comb filter with a lowpass filter in the feedback loop ($g_1 = 0.3$, $g_2 = 0.6$ and $m = 10$).

Table 5.4: Comb filter parameters for Moorer's sound field simulator.

Comb Filter	Delay (ms)	g_1	g_2
1	50	0.4416	0.4635
2	56	0.4608	0.4475
3	61	0.4800	0.4316
4	68	0.4992	0.4157
5	72	0.5088	0.4077
6	78	0.5280	0.3918

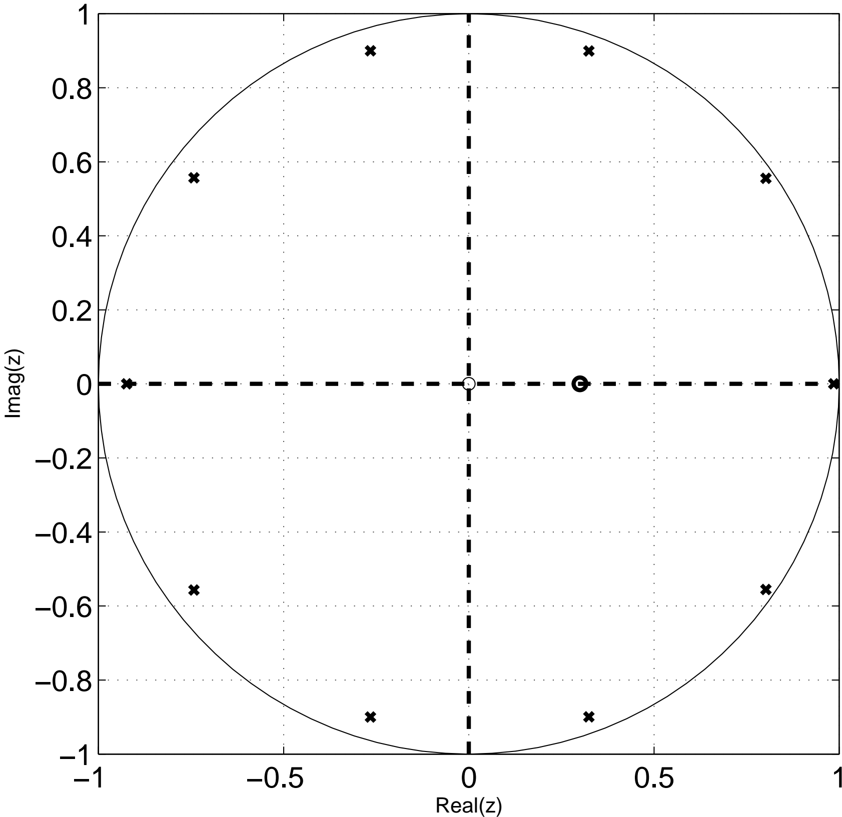


Figure 5.21: Pole and zero locations of comb filter with a lowpass filter in the feedback loop ($g_1 = 0.3$, $g_2 = 0.6$ and $m = 10$).

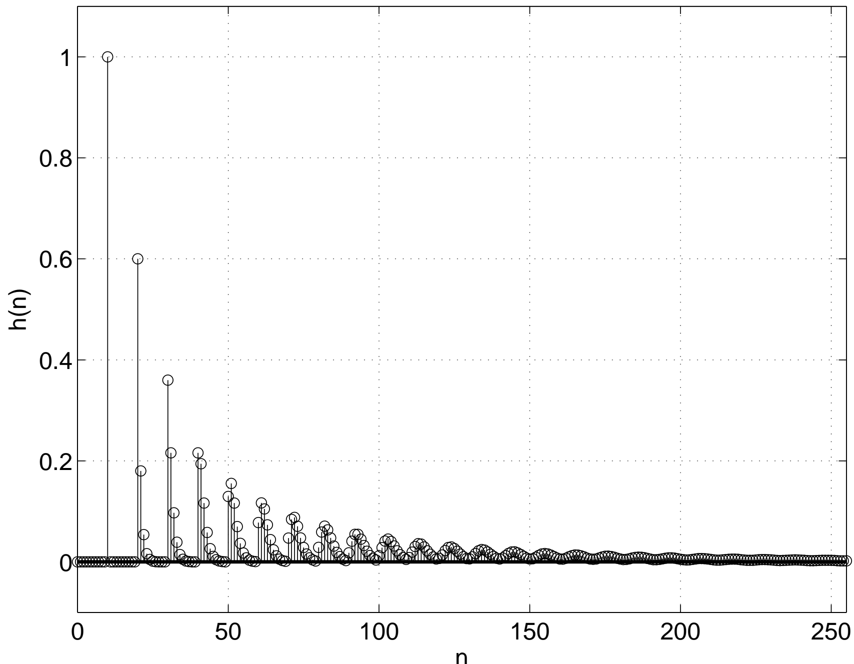


Figure 5.22: Impulse response of comb filter with a lowpass filter in the feedback loop ($g_1 = 0.3$, $g_2 = 0.6$ and $m = 10$).

5.6.3 All Pass Filter

The single APF used in Moorer's sound field simulator is identical to the one described in Section 5.3.4. For this APF, the recommended feedback parameters are $g = 0.7$ and a delay of 6 ms.

5.6.4 Implementation

For the given parameters, the first pulse of the reverberator will emerge *before* the last pulse of the tapped delay line. Normally, reverberation begins after the early reflections. Therefore, we must delay the output of the reverberator in order for the reverberation to follow the tapped delay line output. For the given parameters, the minimum delay in ms will be

$$\mathrm{D} = 79.7 - 50 = 29.7 \tag{5.24}$$

where the last pulse of the TDL occurs 79.7 ms after the initial impulse and the first pulse of the reverberator occurs 50 ms (for CF1) after the same impulse. For our implementation, we insert a delay equal to the above value plus 1 ms,

$$\mathrm{D2} = \mathrm{D} + 1 = 30.7. \tag{5.25}$$

This delay is shown in Figure 5.17. Finally, in order to control reverberation emphasis, we mix the output of the tapped delay line and the reverberator. Figure 5.23 shows the impulse response of the

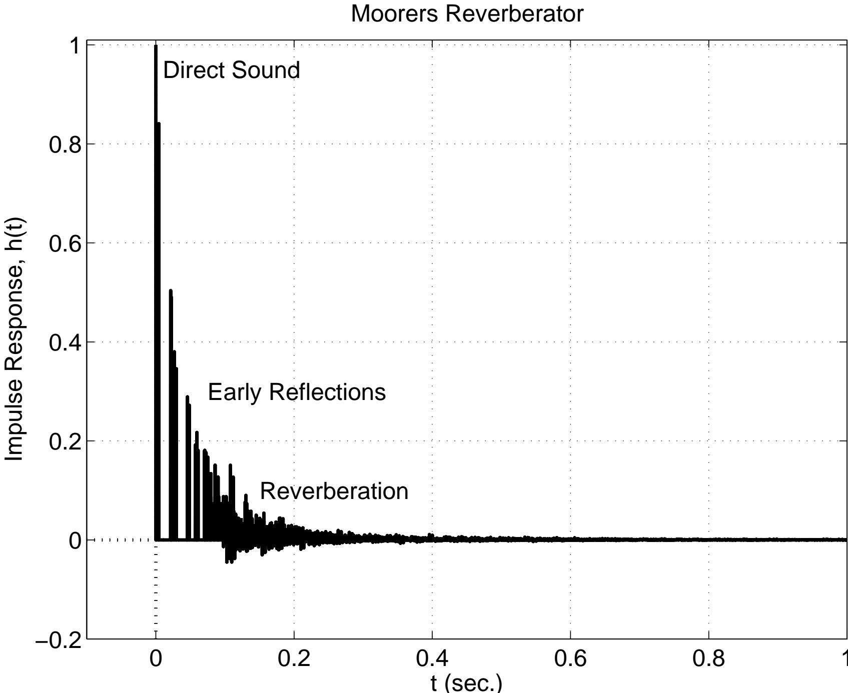


Figure 5.23: Impulse response of Moorer's sound field simulator.

sound field simulator described above.

Chapter 6

Adaptive Noise Cancellation

There are a number of circumstances when a broadband signal is corrupted by a periodic (sinusoidal) interference. Examples include the sensing of seismic signals in the presence of vehicle engine or power line noise. In many cases this periodic noise can be reduced or eliminated. The usual technique is to eliminate the periodic interference with a notch filter at the frequency of the interference. However, should the frequency of the interference change (drift), the fixed notch filter might not be useful in the noise reduction anymore. In this project, we will code an adaptive noise canceler (ANC) which will cancel a periodic interference in a broadband signal even though the frequency of the interference is unknown and changing.

6.1 Optimal Filtering

One common filtering problem is to design an FIR filter $\hat{\mathbf{h}}$, such that when driven by an input $x[n]$, yields an output, $\hat{y}[n]$ that is close in some sense to a desired output, $y[n]$ as in Figure 6.1. One measure of the quality of the filter is how well it minimizes the cost function,

$$J \equiv E\left\{\left[y[n] - \hat{y}[n]\right]^2\right\} \tag{6.1}$$

where E is the statistical expectation operator. This measure leads to the (statistical) least squares filtering problem whose solution is called the *Wiener* filter. Note that for a discrete random variable, X

$$E[X] = \sum_{x:p(x)>0} xp(x) \tag{6.2}$$

which is nothing more than the sum of the outcomes weighted by the probability that the outcome occurs.

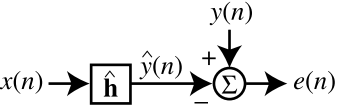


Figure 6.1: Optimal filtering configuration.

Example: Given a fair die (cube with six sides labeled one through six) we can expect the outcome, X to be

$$\begin{aligned} E[X] &= \sum_{x:p(x)>0} xp(x) \\ &= \sum_{k=1}^{6} k\left(\frac{1}{6}\right) \\ &= \frac{7}{2} \end{aligned}$$

■

6.1.1 Wiener Filter

The error signal at sample index n (Figure 6.1) is given by

$$e[n] = y[n] - \hat{y}[n] \tag{6.3}$$

where $y[n]$ is the desired output signal and $\hat{y}[n]$ is the approximation to $y[n]$. The approximation or filter output, $\hat{y}[n]$ is given by

$$\hat{y}[n] = \hat{\mathbf{h}}^T \mathbf{x}[n] \tag{6.4}$$

where $\hat{\mathbf{h}}^T$ is the vector of filter coefficients

$$\hat{\mathbf{h}} = \left[\hat{h}_0, \hat{h}_1, \ldots, \hat{h}_{N-1}\right]^T \tag{6.5}$$

with ^{T} denoting matrix transpose and

$$\mathbf{x}[n] = \left[x[n], x[n-1], \ldots, x[n-N+1]\right]^T \tag{6.6}$$

is the vector containing the current input sample as well as the previous $N-1$ input samples. Next, the mean-squared error (MSE) is defined as

$$\xi \equiv E\left[e[n]^2\right]. \tag{6.7}$$

Substitution of (6.4) into (6.3) and the result into (6.7) yields

$$\begin{aligned} \xi &= E\left[y^2[n]\right] - \hat{\mathbf{h}}^T E\left[\mathbf{x}[n]y[n]\right] - E\left[y[n]\mathbf{x}^T[n]\right]\hat{\mathbf{h}} + \hat{\mathbf{h}}^T E\left[\mathbf{x}[n]\mathbf{x}^T[n]\right]\hat{\mathbf{h}} \\ &= E\left[y^2[n]\right] - \hat{\mathbf{h}}^T \mathbf{p} - \mathbf{p}^T \hat{\mathbf{h}} + \hat{\mathbf{h}}^T \mathbf{R} \hat{\mathbf{h}} \end{aligned} \tag{6.8}$$

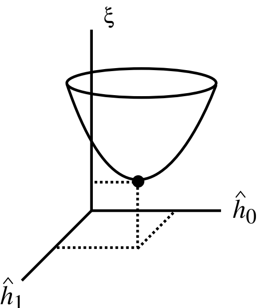


Figure 6.2: MSE Paraboloid for $N = 2$.

where

$$\mathbf{R} \equiv E\Big[\mathbf{x}[n]\mathbf{x}^T[n]\Big] \tag{6.9}$$

is defined as the correlation matrix of $\mathbf{x}[n]$ and

$$\mathbf{p} \equiv E\Big[\mathbf{x}[n]y[n]\Big] \tag{6.10}$$

is defined as the cross-correlation vector between $\mathbf{x}[n]$ and $y[n]$. We complete the square in (6.8) with

$$\begin{aligned}\xi &= E\Big[y^2[n]\Big] - \mathbf{p}^T\mathbf{R}^{-1}\mathbf{p} + \hat{\mathbf{h}}^T\mathbf{R}\hat{\mathbf{h}} - \hat{\mathbf{h}}^T\mathbf{p} - \mathbf{p}^T\hat{\mathbf{h}} + \mathbf{p}^T\mathbf{R}^{-1}\mathbf{p} \\ &= \xi_{min} + \Big(\hat{\mathbf{h}} - \mathbf{R}^{-1}\mathbf{p}\Big)^T\mathbf{R}\Big(\hat{\mathbf{h}} - \mathbf{R}^{-1}\mathbf{p}\Big)\end{aligned} \tag{6.11}$$

where

$$\xi_{min} = E\Big[y^2[n]\Big] - \mathbf{p}^T\mathbf{R}^{-1}\mathbf{p} \tag{6.12}$$

is the minimum mean-squared error (MMSE). Equation (6.11) is a quadratic equation in $\hat{\mathbf{h}}$ and forms an N-dimensional paraboloid as in Figure 6.2. This quadratic is minimized by choosing

$$\begin{aligned}\hat{\mathbf{h}} &= \mathbf{R}^{-1}\mathbf{p} \\ &= \mathbf{h}_{\text{opt}}\end{aligned} \tag{6.13}$$

assuming $\mathbf{R}^{-1}$ exists. The above solution is commonly referred to as the *Wiener*, Least-Squares (LS), or optimal filter.

When $\mathbf{h}_{\text{opt}}$ is driven by $x[n]$, its output $\hat{y}[n]$, is the closest (in the least-squares sense) approximation to $y[n]$. If the input or output statistics change, the filter is no longer optimal and must be recalculated.

In the event $x[n]$ and/or $y[n]$ have time-varying statistics, the Wiener filter will not produce the MMSE at all points in time in which case, $\mathbf{h}_{\text{opt}}$ must be constantly recalculated. Continuous calculation of the Wiener filter can be quite computationally expensive (due to the required matrix inverse), so we turn to adaptive adjustment techniques.

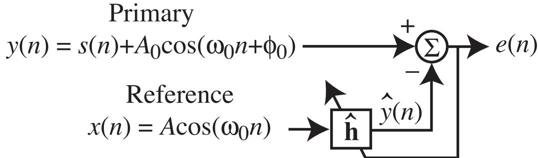


Figure 6.3: Interference canceling configuration.

6.2 Adaptive Filtering

The new problem is then to design a time-varying filter that adjusts or adapts itself to meet some optimality criterion without a priori information about the input or desired signals. Such a filter is called an *adaptive filter*.

6.2.1 Applications of Adaptive Filters

There are four common applications of the adaptive filter: equalization, system identification, prediction, and interference canceling. Interference canceling (Figure 6.3) is used to cancel an unknown sinusoidal interference (noise) that is present in a primary signal which contains an information-bearing signal component. The primary signal serves as the desired response, $y[n]$ for the adaptive filter. A reference signal is employed as the input, $x[n]$ to the adaptive filter. The reference signal is typically located near the noise source so that the interference dominates the signal.

Example: As a first attempt at canceling engine noise in an automobile interior using an adaptive interference canceler, we could obtain a reference signal from a microphone located in the engine compartment, while the primary signal could be obtained from a microphone located in the headrest near the driver's ear. ■

6.2.2 Adaptive Filter Processes

There are two basic processes in adaptive filtering: 1) a filtering process designed to produce an output in response to a sequence of input data and 2) an adaptive process which provides a mechanism for the adaptive control of an adjustable set of parameters used in the filtering process. These processes are illustrated in Figure 6.4.

By far the most popular adaptive filter structure is the FIR filter. The adaptive algorithm used to adjust the set of filter parameters, $\hat{\mathbf{h}}$ given in (6.5), should attempt to give continuous optimal filtering with a *reasonable* amount of computation. Several adaptive filtering algorithms such as the least-mean-square (LMS) and the recursive least-squares (RLS) have been developed with this goal in mind.

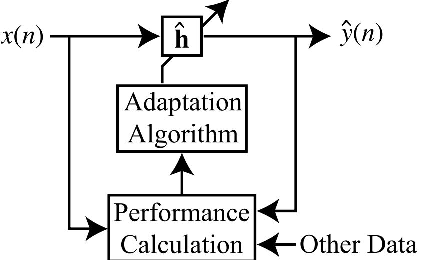


Figure 6.4: Adaptive filter.

6.2.3 Steepest Descent Algorithm

The method of steepest descent (SD) is used to "search" the MSE paraboloid (Figure 6.2) generated from the quadratic in (6.11) for its minimum. This search leads to the MMSE and the associated Wiener filter.

The method begins by initializing $\hat{\mathbf{h}} = \hat{\mathbf{h}}[0]$ where $\hat{\mathbf{h}}[0]$ is the initialization of the adaptive filter coefficients and is simply a guess as to where the MMSE is located on the paraboloid. Usually,

$$\hat{\mathbf{h}}[0] = \Big[0, \ldots, 0\Big]^T. \tag{6.14}$$

Next, the MSE gradient is computed and the subsequent filter is updated by making a change in the direction opposite the gradient. This process is continued until the MMSE is arrived at. In mathematical terms, the update of $\hat{\mathbf{h}}[n]$ to $\hat{\mathbf{h}}[n+1]$ may be expressed as

$$\hat{\mathbf{h}}[n+1] \quad = \quad \hat{\mathbf{h}}[n] + \frac{1}{2}\mu\Big[-\bigtriangledown\xi[n]\Big] \tag{6.15}$$

where μ is the step size that controls the amount of change in the direction opposite the gradient and the 1/2 factor is for mathematical convenience. Figure 6.5 illustrates several steps in a direction opposite the gradient which will lead us to the bottom of a 2D-paraboloid. It is easy to show that

$$\bigtriangledown\xi[n] \quad = \quad -2\mathbf{p} + 2\mathbf{R}\hat{\mathbf{h}}[n]. \tag{6.16}$$

Thus substituting (6.16) into (6.15) yields the SD algorithm for updating the adaptive filter

$$\hat{\mathbf{h}}[n+1] \quad = \quad \hat{\mathbf{h}}[n] + \mu\Big[\mathbf{p} - \mathbf{R}\hat{\mathbf{h}}[n]\Big]. \tag{6.17}$$

We note that for SD, knowledge about the statistics of the input and output samples, i.e. $\mathbf{R}$ and $\mathbf{p}$, is still required; however, the adjustment of the optimal filter can now change as these statistics change. This is different from the Wiener filter which would have to be recalculated.

6.2.4 Analysis of the Steepest Descent Algorithm

The SD algorithm offers a method of searching for $\hat{\mathbf{h}}[n]$ which minimizes the MSE but certainly does not guarantee that $\hat{\mathbf{h}}[\infty] = \mathbf{h}_{\text{opt}}$ or in other words that $\hat{\mathbf{h}}$ will converge to the Wiener solution. A

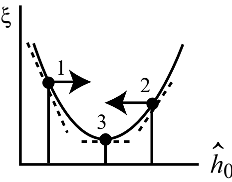


Figure 6.5: Finding the minimum of a quadratic function.

necessary and sufficient condition, however, for convergence of the SD algorithm is

$$0 < \mu < \frac{2}{\lambda_{max}} \tag{6.18}$$

where λ_{max} is the largest eigenvalue of $\mathbf{R}$. The step size bound for convergence of the mean (first moment) of $\hat{\mathbf{h}}[n]$ to $\mathbf{h}_{\text{opt}}$, also guarantees convergence of the mean weight error vector, $E\Big[\hat{\mathbf{h}}[n]-\mathbf{h}_{\text{opt}}\Big]$ to zero.

6.2.5 LMS Algorithm

The limitation of SD is that $\mathbf{R}$ and $\mathbf{p}$ are usually not known *a priori* and must therefore be estimated from the available data. The simplest method of estimating $\mathbf{R}$ and $\mathbf{p}$ is to use the *instantaneous* estimates

$$\hat{\mathbf{R}}[n] = \mathbf{x}[n]\mathbf{x}^T[n] \tag{6.19}$$

and

$$\hat{\mathbf{p}}[n] = \mathbf{x}[n]y[n]. \tag{6.20}$$

Substituting (6.19) and (6.20) for $\mathbf{R}$ and $\mathbf{p}$ in (6.17) yields the LMS algorithm

$$\begin{aligned}
\hat{\mathbf{h}}[n+1] &= \hat{\mathbf{h}}[n] + \mu\Big[\mathbf{x}[n]y[n] - \mathbf{x}[n]\mathbf{x}^T[n]\hat{\mathbf{h}}[n]\Big] \\
&= \hat{\mathbf{h}}[n] + \mu\mathbf{x}[n]\Big[y[n] - \mathbf{x}^T[n]\hat{\mathbf{h}}[n]\Big] \\
&= \hat{\mathbf{h}}[n] + \mu\mathbf{x}[n]e[n].
\end{aligned} \tag{6.21}$$

Using the imperfect estimates of (6.19) and (6.20) to compute the gradient will, however, lead to a "noisy gradient estimate" and consequently a "noisy" adaptive process. One outstanding feature of the LMS algorithm, however, is the simplicity. If x and y are assumed to be real signals and the length of $\hat{\mathbf{h}}$ is N, then the number of `MAC`s per input sample (which is a measure of computational complexity) is

$$C = 2N + 1. \tag{6.22}$$

In practice, we usually choose the step-size according to

$$0 < \mu < \frac{2}{3N\sigma^2} \tag{6.23}$$

where σ^2 is the variance or power of the input signal.

6.2.6 NLMS Algorithm

As can be seen from the LMS algorithm in (6.21), the correction applied to $\hat{\mathbf{h}}[n]$ is proportional to $\mathbf{x}[n]$ and thus if elements of $\mathbf{x}[n]$ are large, the LMS algorithm will experience a "gradient noise amplification." The simple way of managing this is to make μ inversely proportional to the signal power of $\mathbf{x}[n]$ or in effect "normalize" the correction to the input signal power. In this case, the step size is time-varying and given by

$$\mu[n] = \frac{\tilde{\mu}}{\|\mathbf{x}[n]\|^2} \tag{6.24}$$

where

$$\|\mathbf{x}[n]\|^2 = \mathbf{x}[n]^T \mathbf{x}[n] = \sum_{k=0}^{N-1} |x[n-k]|^2 \tag{6.25}$$

and $\tilde{\mu}$ scales the correction. This idea leads directly to the normalized least-mean square (NLMS) algorithm by substituting (6.24) into (6.21)

$$\hat{\mathbf{h}}[n+1] = \hat{\mathbf{h}}[n] + \frac{\tilde{\mu}}{\|\mathbf{x}[n]\|^2}\mathbf{x}[n]e[n]. \tag{6.26}$$

Finally, when $\|\mathbf{x}[n]\|^2$ is a small number there may be numerical difficulties with the division in (6.26) so a modification is made by adding a small offset, $a > 0$

$$\hat{\mathbf{h}}[n+1] = \hat{\mathbf{h}}[n] + \frac{\tilde{\mu}}{a + \|\mathbf{x}[n]\|^2}\mathbf{x}[n]e[n]. \tag{6.27}$$

For stability of the NLMS algorithm we require

$$0 < \tilde{\mu} < 2. \tag{6.28}$$

6.3 Adaptive Cancellation of Sinusoidal Interference

In many environments, a primary signal is available which contains both a signal of interest (information bearing) and an undesired sinusoidal interference. Naturally, if the frequency of the sinusoidal interference was known and fixed, a simple notch filter would suffice. However, if the sinusoidal interference has a frequency which drifts slowly, then a fixed notch filter will not work. In this case we must design a notch filter whose notch follows the frequency of the sinusoidal interference. We independently consider two methods of adaptive cancellation of the sinusoidal interference: adaptive noise canceler and an adaptive line enhancer.

6.3.1 Adaptive Noise Canceler

In the adaptive noise canceler given in Figure 6.3, we assume the availability of both a primary signal and a reference signal. This reference signal contains only the sinusoidal interference and may

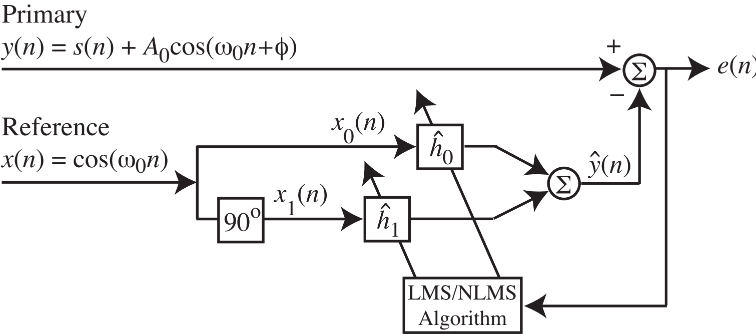


Figure 6.6: Single frequency adaptive notch filter.

be obtained by locating a sensor near the noise source so that the noise source dominates. In the configuration one might initially think that a simple subtraction of the reference from the primary would eliminate the noise. However, we cannot assume that the sinusoid in the primary has the exact amplitude and phase as in the reference. Therefore, simple subtraction will not work and we must construct an adaptive filter to adjust the sinusoid in the reference so that it matches that in the primary. The filter output is then subtracted from the primary leaving only the signal of interest.

The ANC can be constructed with two simple one-coefficient adaptive filters as illustrated in Figure 6.6. Here we note that the reference signal, $x[n]$ is split into two paths: the upper path is multiplied $\hat{h}_0$ and the lower path is first phase shifted by 90° and then multiplied by $\hat{h}_1$. Each filter coefficient is separately adapted using either (6.21) or (6.27).

The 90-degree phase shift can be digitally implemented with a *Hilbert transformer* or *Hilbert filter*. In MATLAB such a filter can be designed with the command (in MATLAB, 'help remez' for more information)

```
hilbert_coefs=firpm(30,[.1 .9],[1 1],'Hilbert');
```

Such a filter has a frequency response as in Figure 6.7. Assuming the length of the Hilbert filter is P (odd) and that the filter has a linear phase, the filter delay will be $(P-1)/2$. In order to keep the two inputs to the adaptive filter time-aligned (coherent), we must add a delay of $(P-1)/2$ samples to the input which directly feeds $\hat{h}_0$. A fully digital version of the adaptive notch filter is illustrated in Figure 6.8.

Simple Analysis of the ANC

In order to understand how ANC in Figure 6.6 operates, we note that the objective is for $\hat{y}(n)$ to be close to $A_0\cos(\omega_0 n+\phi)$ so that $e(n)\approx s(n)$ resulting in cancelation of the sinusoid in the primary. Applying a simple trigonometry relation, we have that

$$A_0\cos(\omega_0 n+\phi) \quad = \quad A_0\cos(\phi)\cos(\omega_0 n) - A_0\sin(\phi)\sin(\omega_0 n). \qquad (6.29)$$

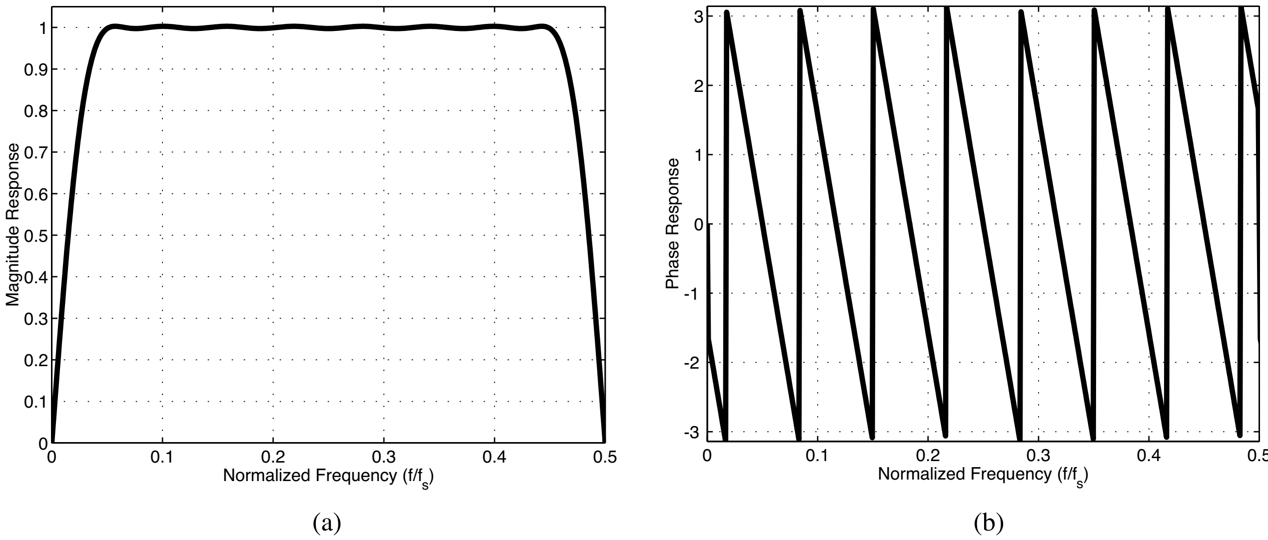


Figure 6.7: (a) Magnitude response and (b) phase response of the example Hilbert filter.

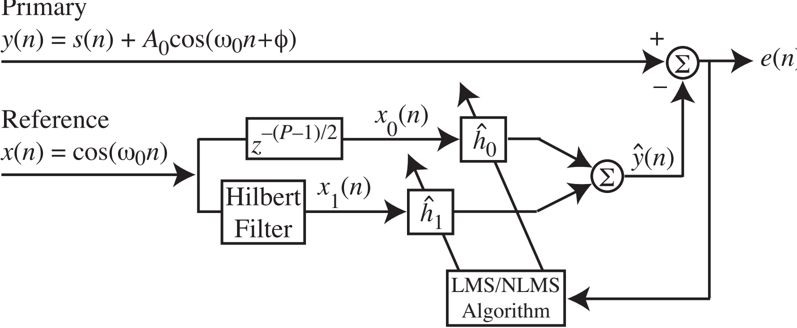


Figure 6.8: Single frequency adaptive notch filter with 90-degree phase shift implemented with a digital Hilbert filter.

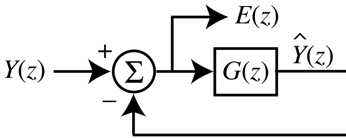


Figure 6.9: Equivalent model of the ANC.

Because of the 90° phase shift,

$$\hat{y}(n) = \hat{h}_0 \cos(\omega_0 n) + \hat{h}_1 \sin(\omega_0 n). \tag{6.30}$$

Thus if the filter coefficients adjust themselves so that $\hat{h}_0 = A_0 \cos(\phi)$ and $\hat{h}_1 = -A_0 \sin(\phi)$, the objective is obtained. The LMS/NLMS algorithm will in fact, adjust the coefficients in such a manner.

Analysis of the ANC

In order to understand the operation of the ANC, we perform a time- and frequency-domain analysis of the ANC. The first step in the analysis is to compute the impulse response, $g[n]$ of the subsystem from the error (ANC output), $e[n]$ back to the filter output $\hat{y}[n]$ while the feedback loop from the filter output to the upper right summing node in Figure 6.8 is broken. The second step is then to close the feedback loop and obtain the overall transfer function of the ANC, $H(z)$ from the primary input, $y[n]$ to the error. The third step is then to examine $H(z)$.

To begin, we restructure the block diagram of the ANC as in Figure 6.9. With this new representation, we lump the reference input, $x[n]$; the adaptive filter, $\hat{\mathbf{h}}$; and the filter update equation (LMS) into a single, open-loop system defined by transfer function,

$$G(z) = \frac{X(z)}{E(z)}. \tag{6.31}$$

Here $X(z)$ and $E(z)$ are the z-transforms of the reference input, $x[n]$ and error, $e[n]$ respectively.

In our first step, we apply an impulse, $\delta[n]$ to $G(z)$ with the feedback loop open. It can be shown that

$$g[n] = 2\mu \cos(\omega_0 n) u[n-1] \tag{6.32}$$

where $u[n]$ is the unit-step function. Taking the z-transform of (6.32) results in

$$G(z) = \frac{2\mu[z\cos(\omega_0) - 1]}{z^2 - 2z\cos(\omega_0) + 1}. \tag{6.33}$$

In our second step, we close the feedback loop and obtain the overall transfer function of the ANC,

$$H(z) = \frac{1}{1 - G(z)}$$

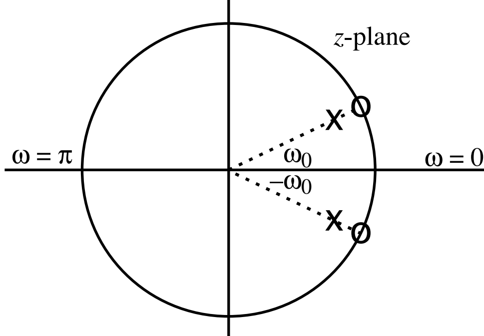


Figure 6.10: Pole zero diagram for the equivalent model of the ANC.

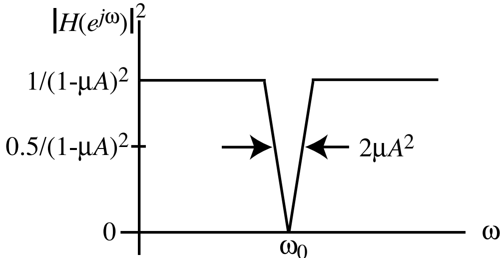


Figure 6.11: Magnitude response of a notch filter.

$$\begin{aligned} &= \frac{z^2 - 2z\cos(\omega_0) + 1}{z^2 - 2(1-\mu)z\cos(\omega_0) + 1 - 2\mu} \\ &= \frac{(z - e^{j\omega_0})(z - e^{-j\omega_0})}{z^2 - 2(1-\mu)z\cos(\omega_0) + 1 - 2\mu}. \end{aligned} \tag{6.34}$$

Finally, in our third step, we recognize that $H(z)$ has a pair of zeros, z_i and a pair of poles, p_i given by

$$\begin{aligned} z_i &= e^{\pm j\omega_0} \\ p_i &= (1-\mu)\cos(\omega_0) \pm j\left[(1-2\mu) - (1-\mu)^2\cos^2(\omega_0)\right]^{1/2}. \end{aligned} \tag{6.35}$$

The zeros are located on the unit circle at an angle of ω_0 and the poles are located just inside the unit circle at a radial distance of $(1-2\mu)^{1/2}$ also at an angle of ω_0 as in Figure 6.10. We therefore conclude that (6.34) and hence our adaptive noise canceler is a notch filter with a notch at $\omega = \omega_0$ as in Figure 6.11. In addition, the 3dB width of the notch filter can be shown to be 2μ thus the smaller μ is the tighter the notch can be made. Of course, the trade off is that with a small μ, the adaptive adjustment algorithm (LMS or NLMS) converges slower.

6.4 ANC Implementation

The ANC given in Figure 6.8 can be implemented on a DSP in a relatively easy manner. We note the basic building blocks include a simple delay (FIFO) which feeds $\hat{h}_0$ and a Hilbert FIR filter

which performs the 90-degree phase shift and feeds $\hat{h}_1$. We also have the adaptive algorithm which individually adjusts $\hat{h}_0$ and $\hat{h}_0$.

6.4.1 Algorithm

INITIALIZE

Clear out DELAY_QUEUE, HILBERT_QUEUE, INPUT, and H_HAT

MAIN (once per sample period)

Get PRIMARY (left channel) and REFERENCE (right channel) from codec

CANCEL_PERIODIC_INTERFERENCE

COMPUTE_MU_ERROR

ADJUST_ADAPTIVE_FILTER

OUTPUT

Goto ***MAIN***

CANCEL_PERIODIC_INTERFERENCE

Get $x_0[n]$ as oldest sample in DELAY_QUEUE

Replace oldest sample in DELAY_QUEUE with REFERENCE

Replace oldest sample in HILBERT_QUEUE with REFERENCE

Compute $x_1[n]$ as inner product between HILBERT_COEFS and HILBERT_QUEUE

Compute FILTER_OUT as inner product between H_HAT = $[\hat{h}_0[n] \;\; \hat{h}_1[n]]^T$ and

INPUT = $[x_0[n] \;\; x_1[n]]^T$

Compute ERROR = PRIMARY − FILTER_OUT

COMPUTE_MU_ERROR (LMS)

Compute MU_ERROR = MU × ERROR

COMPUTE_MU_ERROR (NLMS)

Read in MU_TILDE

Compute INPUT_POWER (sum of squares of samples in INPUT)

Compute MU_ERROR = (MU_TILDE × ERROR) / (OFFSET + INPUT_POWER)

ADJUST_ADAPTIVE_FILTER

Compute H_HAT ← H_HAT + MU_ERROR × INPUT

OUTPUT

Transmit ERROR (noise-canceler output), FILTER_OUTPUT as left, right channels.

6.4.2 Layout in Memory

The ANC requires two circular queues: one for the delay which feeds $\hat{h}_0$ and one for the input to the Hilbert filter which feeds $\hat{h}_1$.

ANC.DAT

```
1   ;****************************************************************************
2   ;ANC.DAT: This data file is used with PROJECT.ASM to lay out memory.
3   ;****************************************************************************
4
5   ;**********
6   ; Equates
7   ;**********
8   ADA_OFF equ     0       ;0 enables codec, 1 disables codec
9   AAR0    equ     $010931 ;split external (off-chip) 32K RAM into 16K X & 16K Y
10  AAR3    equ     $010925 ;beginning at $010000 p.4-6--4-8 DSP56302EVM UM
11
12  MU              equ     0.005   ;LMS step size PLAY WITH THIS!!!
13  A_OFFSET        equ     0.0001  ;offset in NLMS to avoid division by zero
14  MU_TILDE        equ     0.005   ;NLMS step size PLAY WITH THIS!!!
15  P               equ     31      ;Hilbert filter length
16
17  ;************
18  ; Data Memory
19  ;************
20  ;*******************************************************************************
21  ;---Buffer for the CS4215
22  ;       The following lines of code are required for proper on-board audio
23  ;       codec operation.
24  ;*******************************************************************************
25          org     x:0     ;On-chip X memory $000000 -- $001BFF
26  RX_BUFF_BASE    equ     *
27  RX_data_1_2     ds      1       ;data time slot 1/2 for RX ISR
28  RX_data_3_4     ds      1       ;data time slot 3/4 for RX ISR
29  RX_data_5_6     ds      1       ;data time slot 5/6 for RX ISR
30  RX_data_7_8     ds      1       ;data time slot 7/8 for RX ISR
31
32  TX_BUFF_BASE    equ     *
33  TX_data_1_2     ds      1       ;data time slot 1/2 for TX ISR
34  TX_data_3_4     ds      1       ;data time slot 3/4 for TX ISR
35  TX_data_5_6     ds      1       ;data time slot 5/6 for TX ISR
36  TX_data_7_8     ds      1       ;data time slot 7/8 for TX ISR
37
38  RX_PTR          ds      1       ;Pointer for rx buffer
39  TX_PTR          ds      1       ;Pointer for tx buffer
```

```
40
41
42          org x:$00000a
43  ;******************************************************************************
44  ;---On-chip X data goes here
45  ;******************************************************************************
46  HILBERTCOEFS dsm P
47          org x:HILBERTCOEFS      ;MATLAB: g=remez(30,[.1 .9],[1 1],'Hilbert')
48          dc      -0.00419563589035
49          dc      0
50          dc      -0.00928210154880
51          dc      0
52          dc      -0.01883580699771
53          dc      0
54          dc      -0.03440100801933
55          dc      0
56          dc      -0.05955157556970
57          dc      0
58          dc      -0.10303763641989
59          dc      0
60          dc      -0.19683153562364
61          dc      0
62          dc      -0.63135364088220
63          dc      0
64          dc      0.63135364088220
65          dc      0
66          dc      0.19683153562364
67          dc      0
68          dc      0.10303763641989
69          dc      0
70          dc      0.05955157556970
71          dc      0
72          dc      0.03440100801933
73          dc      0
74          dc      0.01883580699771
75          dc      0
76          dc      0.00928210154880
77          dc      0
78          dc      0.00419563589035
79
80  ;******************************************************************************
81  ;---Software Stack
82  STACK   equ     *       ;locate stack after last thing in on-chip X memory
83                          ;STACK is used in TXRX_ISR.ASM and must be allowed to grow
84
85  ;You must not write *anything* into x:$001C00 --x:$00FFFF (check .LST file)
86
87
88          org x:$010000   ;Off-Chip X memory $010000 -- $013FFF
89  ;******************************************************************************
90  ;---Off-chip X data goes here
```

```
91   ;       If you load actual values in off-chip memory, i.e. "dc" then you must run
92   ;       pass.asm first so that off-chip memory can be configured before the load
93   ;       is attempted.  In this case, Debugger should have Reset on Load (Config
94   ;       menu) unchecked.
95   ;*****************************************************************************
96
97
98           org y:$000000   ;On-chip Y memory $000000 -- $001BFF
99   ;*****************************************************************************
100  ;---On-chip Y data goes here
101  ;*****************************************************************************
102  DELAYQ          dsm     @CVI((P-1)/2)
103  HILBERTQ        dsm     P
104
105
106  ;You must not write *anything* into y:$001C00 --y:$00FFFF (check .LST file)
107
108
109          org y:$010000   ;Off-chip Y memory $010000 -- $013FFF
110  ;*****************************************************************************
111  ;---Off-chip Y data goes here
112  ;       If you load actual values in off-chip memory, i.e. "dc" then you must run
113  ;       pass.asm first so that off-chip memory can be configured before the load
114  ;       is attempted.  In this case, Debugger should have Reset on Load (Config
115  ;       menu) unchecked.
116  ;*****************************************************************************
```

6.4.3 Adaptive Line Enhancer

The adaptive line enhancer (ALE) in Figure 6.12 is a device that may be used to detect a periodic signal buried in a broad-band noise background. The ALE is in fact a degenerate form of the ANC in that its reference signal, instead of being derived separately, consists of a delayed version of the primary signal. The delay, Δ is called the prediction depth or decorrelation delay of the ALE. The reference signal to the adaptive filter is thus given by

$$x[n] = y[n-\Delta] \tag{6.36}$$

and is processed to produce an error signal,

$$e[n] = y[n] - \hat{y}[n] \tag{6.37}$$

where the output of the adaptive filter, $\hat{y}[n]$ is given by

$$\begin{aligned} \hat{y}[n] &= \hat{\mathbf{h}}^T[n]\mathbf{x}[n] \\ &= \hat{\mathbf{h}}^T[n]\mathbf{y}[n-\Delta]. \end{aligned} \tag{6.38}$$

The error signal is used in the adjustment of the N coefficients of the adaptive filter.

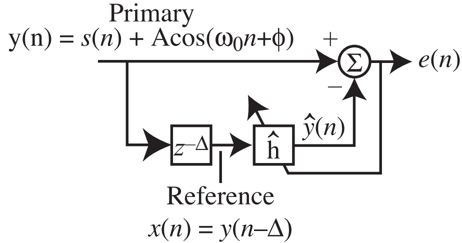


Figure 6.12: Adaptive Line Enhancer.

Time-Domain Analysis of the ALE

Consider a primary signal $y[n]$ consisting of a sinusoidal component, $A\cos(\omega_0 n + \phi)$ buried in wide-band signal of interest (SOI), $s[n]$ as given by

$$y[n] \quad = \quad s[n] + A\cos\left(\omega_0 n + \phi\right) \tag{6.39}$$

where ϕ is an arbitrary phase shift and the SOI $s[n]$ is assumed to have zero mean and variance σ_s^2. The ALE acts as a signal detector by virtue of two actions:

- The prediction depth, Δ is assigned a value large enough to remove the correlation between the SOI, $s[n]$ in the original input signal and the SOI $s(n-\Delta)$ in the reference signal, while a simple phase shift equal to $\omega_0\Delta$ is introduced between the sinusoidal components in these two inputs.
- The adaptive filter coefficients are adjusted by the LMS (or NLMS) algorithm so as to minimize the MSE and thereby compensate for the phase shift $\omega_0\Delta$

The net result of these two actions is the production of an output signal $\hat{y}[n]$ that consists of a scaled sinusoid in zero-mean noise. In particular, when ω_0 is several multiples of π/N away from zero or π, it can be shown that

$$\hat{y}[n] \quad = \quad s_{out}[n] + aA\cos(\omega_0 n + \phi) \tag{6.40}$$

where ϕ denotes a phase shift, and $s_{out}[n]$ denotes the output SOI. The scaling factor a is computed as

$$a \quad = \quad \frac{(N/2)SNR}{1 + (N/2)SNR} \tag{6.41}$$

where the signal-to-noise ratio (SNR) at the input of the ALE is given by

$$SNR \quad = \quad \frac{A^2}{2\sigma_s^2}. \tag{6.42}$$

According to (6.40), we can see that the ALE acts as a "self-tuning filter" whose frequency response exhibits a peak at the angular frequency, ω_0 of the incoming sinusoid, hence the name "spectral line enhancer" or simply "line-enhancer."

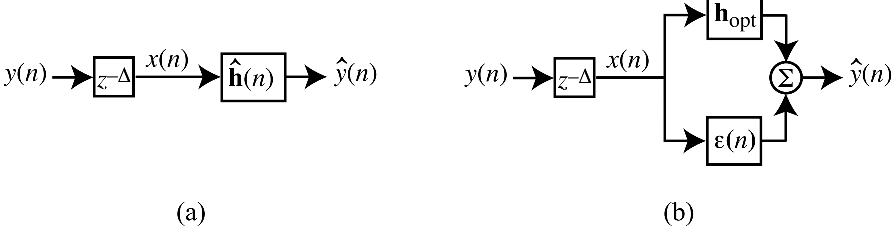


Figure 6.13: Steady-state model of ALE.

Example: Suppose that the ALE has $N = 16$, $A = 1$, and $\sigma_v^2 = 1$. Then,

$$SNR = \frac{1}{2} \tag{6.43}$$

and

$$a = \frac{4}{5}. \tag{6.44}$$

Therefore the error signal (output of the ALE) is given by

$$\begin{aligned} e[n] &= y[n] - \hat{y}[n] \\ &= A\cos(\omega_0 n + \phi) + s[n] - aA\cos(\omega_0 n + \phi) + s_{out}[n] \\ &= (1-a)\cos(\omega_0 n + \phi) + s[n] + s_{out}[n] \\ &= \frac{1}{5}\cos(\omega_0 n + \phi) + s[n] + s_{out}[n]. \end{aligned} \tag{6.45}$$

We can see from this example that the amplitude of the sinusoid is scaled by a factor of 1/5 or −14dB but an output SOI, $s_{out}[n]$ is also added to $e[n]$. ■

Frequency-Domain Analysis of the ALE

The power spectral density (PSD) (how the power is distributed over frequency) or power spectrum is defined as

$$r_{xx}(l) = E\Big[x[n]x(n-l)\Big] \quad \leftrightarrow \quad S_{xx}(\omega) = \sum_{l=-\infty}^{\infty} r_{xx}(l)e^{-j\omega l}. \tag{6.46}$$

The PSD of $\hat{y}[n]$ (assuming the ALE is adjusted by an LMS algorithm) may be expressed as

$$\begin{aligned} S_{\hat{y}\hat{y}}(\omega) &= \frac{\pi A^2}{2}\Big(a^2 + \mu\sigma_s^2 N\Big)\Big[\delta\Big(\omega - \omega_0\Big) + \delta\Big(\omega + \omega_0\Big)\Big] + \mu\sigma_s^4 N + \\ &\quad \frac{a^2\sigma_s^2}{N^2}\left\{\frac{1 - \cos\Big[N\Big(\omega - \omega_0\Big)\Big]}{1 - \cos\Big(\omega - \omega_0\Big)} + \frac{1 + \cos\Big[N\Big(\omega - \omega_0\Big)\Big]}{1 + \cos\Big(\omega - \omega_0\Big)}\right\} \end{aligned} \tag{6.47}$$

for $-\pi < \omega < \pi$. Before we analyze the PSD, we note that the steady-state model of the ALE can be viewed in the following way. The converged adaptive filter, $\hat{\mathbf{h}}[n]$ consists of the Wiener solution $\mathbf{h}_{\text{opt}}$ acting in parallel with a slowly fluctuating random component $\varepsilon[n]$ as in Figure 6.13.

Recognizing that the ALE input consists of two components, a sinusoid of frequency, ω_0 and a wide-band SOI, $s[n]$ of zero mean and variance σ_s^2, we may distinguish four components in the power spectrum.

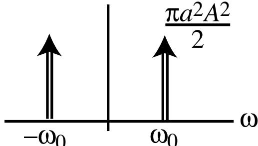


Figure 6.14: Sinusoidal component resulting from processing input sinusoid by Wiener filter.

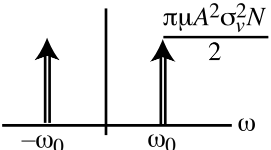


Figure 6.15: Sinusoidal component resulting from processing input sinusoid by stochastic filter $\epsilon[n]$.

- A sinusoidal component of frequency, ω_0 and average power, $\pi a^2 A^2/2$, which is the result of processing the input sinusoid by the Wiener filter, $\mathbf{h}_{\text{opt}}$ as in Figure 6.14.

- A sinusoidal component of frequency, ω_0 and average power, $\pi\mu A^2\sigma_s^2 N/2$, which is due to the stochastic filter represented by the weight vector, $\varepsilon[n]$ acting on the input sinusoid as in Figure 6.15.

- A wide-band noise component of variance, $\mu\sigma_s^4 N$, which is due to the action of the stochastic filter on the SOI $s[n]$ as in Figure 6.16.

- A narrow-band noise component centered on ω_0, which results from processing the SOI $s[n]$ by the Wiener filter as in Figure 6.17.

Thus the power spectrum of the ALE output consists of a sinusoid at the center of a pedestal of narrow-band filtered noise, the combination of which is embedded in a wide-band noise background. Most importantly, when adequate SNR exists at the ALE input, the ALE output is, on average, approximately equal to the sinusoidal component present at the input, thereby providing a simple adaptive device for the detection of a sinusoid in wide-band noise.

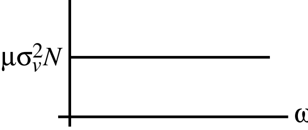


Figure 6.16: Wide band noise.

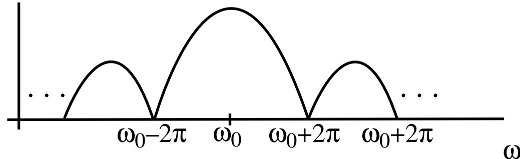


Figure 6.17: Narrow band noise.

6.5 ALE Implementation

The ALE given in Figure 6.12 can be implemented on a DSP in a relatively easy manner. We note the basic building blocks include a simple delay (FIFO) which feeds the FIR filter, $\hat{\mathbf{h}}[n]$. We also have the adaptive algorithm which adjusts $\hat{\mathbf{h}}$. LMS code can be found in Appendix C of the *DSP56300 Family Manual*.

6.5.1 Algorithm

INITIALIZE

Clear out DECORRELATION_QUEUE, INPUT_QUEUE, and H_HAT

MAIN (once per sample period)

Get PRIMARY (left channel) from codec

CANCEL_PERIODIC_INTERFERENCE

GET_MU

ADJUST_ADAPTIVE_FILTER

OUTPUT

Goto ***MAIN***

CANCEL_PERIODIC_INTERFERENCE

Get REFERENCE as oldest sample in DECORRELATION_QUEUE

Replace oldest sample in INPUT_QUEUE with REFERENCE

Replace oldest sample in DECORRELATION_QUEUE with PRIMARY

Compute OUT as inner product between H_HAT and INPUT_QUEUE

Compute ERROR = PRIMARY − OUT

GET_MU (LMS)

Read in MU (fixed)

GET_MU (NLMS)

Compute INPUT_POWER (sum of squares of samples in INPUT_QUEUE)

Read in MU_TILDE

Compute MU = MU_TILDE / (OFFSET + INPUT_POWER)

ADJUST_ADAPTIVE_FILTER

Compute MU_ERR = (MU)(ERROR)

Compute H_HAT = H_HAT + (MU_ERR)(INPUT_QUEUE)

OUTPUT

Transmit PRIMARY (pass-thru) as left channel and ERROR as right channel.

6.5.2 Layout in Memory

The ALE requires three circular queues: one for the decorrelator, one for the input to the adaptive filter, and one for the adaptive filter coefficients.

ALE.DAT

```
1   ;****************************************************************************
2   ;ALE.DAT: This data file is used with PROJECT.ASM to lay out memory.
3   ;****************************************************************************
4
5   ;**********
6   ; Equates
7   ;**********
8   ADA_OFF equ     0       ;0 enables codec, 1 disables codec
9   AAR0    equ     $010931 ;split external (off-chip) 32K RAM into 16K X & 16K Y
10  AAR3    equ     $010925 ;beginning at $010000 p.4-6--4-8 DSP56302EVM UM
11
12  SRATE           equ     48000   ;Must correspond to value in ADA_INIT
13  N               equ     64      ;adaptive filter length PLAY WITH THIS!!!
14  MU              equ     0.05    ;LMS step size PLAY WITH THIS!!!
15  A_OFFSET        equ     0.0001  ;offset in NLMS to avoid division by zero
16  MU_TILDE        equ     0.05    ;NLMS step size PLAY WITH THIS!!!
17  DECORRELATION_DELTA     equ     @cvi(0.005*@cvf(SRATE)) ;5-50ms delay
18
19  ;************
20  ; Data Memory
21  ;************
22  ;*******************************************************************************
```

```
23  ;---Buffer for the CS4215
24  ;       The following lines of code are required for proper on-board audio
25  ;       codec operation.
26  ;******************************************************************************
27          org     x:0     ;On-chip X memory $000000 -- $001BFF
28  RX_BUFF_BASE    equ     *
29  RX_data_1_2     ds      1       ;data time slot 1/2 for RX ISR
30  RX_data_3_4     ds      1       ;data time slot 3/4 for RX ISR
31  RX_data_5_6     ds      1       ;data time slot 5/6 for RX ISR
32  RX_data_7_8     ds      1       ;data time slot 7/8 for RX ISR
33
34  TX_BUFF_BASE    equ     *
35  TX_data_1_2     ds      1       ;data time slot 1/2 for TX ISR
36  TX_data_3_4     ds      1       ;data time slot 3/4 for TX ISR
37  TX_data_5_6     ds      1       ;data time slot 5/6 for TX ISR
38  TX_data_7_8     ds      1       ;data time slot 7/8 for TX ISR
39
40  RX_PTR          ds      1       ;Pointer for rx buffer
41  TX_PTR          ds      1       ;Pointer for tx buffer
42
43
44          org x:$00000a
45  ;******************************************************************************
46  ;---On-chip X data goes here
47  ;******************************************************************************
48
49
50  ;******************************************************************************
51  ;---Software Stack
52  STACK   equ     *       ;locate stack after last thing in on-chip X memory
53                          ;STACK is used in TXRX_ISR.ASM and must be allowed to grow
54
55  ;You must not write *anything* into x:$001C00 --x:$00FFFF (check .LST file)
56
57
58          org x:$010000   ;Off-Chip X memory $010000 -- $013FFF
59  ;******************************************************************************
60  ;---Off-chip X data goes here
61  ;       If you load actual values in off-chip memory, i.e. "dc" then you must run
62  ;       pass.asm first so that off-chip memory can be configured before the load
63  ;       is attempted.  In this case, Debugger should have Reset on Load (Config
64  ;       menu) unchecked.
65  ;******************************************************************************
66
67
68          org y:$000000   ;On-chip Y memory $000000 -- $001BFF
69  ;******************************************************************************
70  ;---On-chip Y data goes here
71  ;******************************************************************************
72
73
```

```
74   ;You must not write *anything* into y:$001C00 --y:$00FFFF (check .LST file)
75
76
77           org y:$010000   ;Off-chip Y memory $010000 -- $013FFF
78   ;******************************************************************************
79   ;---Off-chip Y data goes here
80   ;       If you load actual values in off-chip memory, i.e. "dc" then you must run
81   ;       pass.asm first so that off-chip memory can be configured before the load
82   ;       is attempted.  In this case, Debugger should have Reset on Load (Config
83   ;       menu) unchecked.
84   ;******************************************************************************
```

6.6 Fixed-Point Issues

6.6.1 ANC

For the ANC in Figure 6.8, we have

$$\hat{\mathbf{h}}[n] = \left[\begin{array}{cc} \hat{h}_0[n] & \hat{h}_1[n] \end{array} \right]^T \tag{6.48}$$

and

$$\mathbf{x}[n] = \left[\begin{array}{cc} x_0[n] & x_1[n] \end{array} \right]^T \tag{6.49}$$

where the filter output is given by

$$\hat{y}[n] = \hat{\mathbf{h}}^T[n]\mathbf{x}[n]. \tag{6.50}$$

In the NLMS adaptive filter update given in (6.27), we must be careful with the division since on a fixed point DSP such as the DSP5630x, we require (for the multiply with $\mathbf{x}[n]$)

$$-1 \leq \frac{\tilde{\mu}e[n]}{a + \|\mathbf{x}[n]\|^2} < 1 \tag{6.51}$$

where $\|\mathbf{x}[n]\|^2 = x_0^2[n] + x_1^2[n]$. These bounds may be exceeded since we know only that

$$-2 \leq e[n] < 2 \tag{6.52}$$

where $e[n] = y[n] - \hat{y}[n]$ and

$$0 \leq \|\mathbf{x}[n]\|^2 = A^2 \leq 1. \tag{6.53}$$

Here A is the amplitude of the $x_0[n]$ and $x_1[n]$ tones which are 90° out of phase with each other. There are two cases that we need to examine in order to carry out the division.

CASE 1: If $a + \|\mathbf{x}[n]\|^2 \leq \tilde{\mu}|e[n]|$, then strictly speaking the division in (6.51) will result in a value greater in magnitude than 1.0. So we simply set

$$\frac{\tilde{\mu}e[n]}{a + \|\mathbf{x}[n]\|^2} = \mathrm{sgn}[e[n]] \tag{6.54}$$

where

$$\text{sgn}[e[n]] = \begin{cases} 0.999999, & e[n] \geq 0 \\ -1.0, & e[n] < 0 \end{cases} \tag{6.55}$$

Note that whether $a + \|\mathbf{x}[n]\|^2 \leq \tilde{\mu}|e[n]|$ is true or not can be determined by employing one of the condition codes such as `LE` (see *DSP56300FM* p. A-250).

CASE 2: If $a + \|\mathbf{x}[n]\|^2 > \tilde{\mu}|e[n]|$ and we assume $A < 1 - a$, then

$$a + \|\mathbf{x}[n]\|^2 < 1 \tag{6.56}$$

and we can carry out the division in (6.51). Note that one may use either the signed (four quadrant) division or, realizing that the denominator is always positive, carry out an unsigned division using $|e[n]|$ and then change sign according to $\text{sgn}[e[n]]$.

6.6.2 ALE

For the ALE in Figure 6.12, we have

$$\hat{\mathbf{h}}[n] = \begin{bmatrix} \hat{h}_0[n] & \hat{h}_1[n] & \cdots & \hat{h}_{N-1}[n] \end{bmatrix}^T \tag{6.57}$$

with the input vector, $\mathbf{x}[n]$ given by (6.6) and filter output by (6.50). The division in the NLMS algorithm in (6.27) also represents a problem due to the fixed-point nature of the DSP. In the ALE structure, the adaptive filter is assumed to be length N and thus we know only that

$$0 < \|\mathbf{x}[n]\|^2 < N. \tag{6.58}$$

We present a simple solution to the division problem assuming $a \ll \tilde{\mu}$.

Example (ALE): Let $N = 48$ and define L to be the smallest power of two which is greater than N. In this example $L = 64$. We require

$$\frac{\tilde{\mu}e[n]}{a + \|\mathbf{x}[n]\|^2} < 1 \tag{6.59}$$

so that we may properly multiply with $x[n-k]$ for the filter update. We consider the following cases. ■

CASE 1: If $a + \|\mathbf{x}[n]\|^2 \leq \tilde{\mu}|e[n]|$, then strictly speaking the resulting division in (6.51) will result in a value greater in magnitude than 1.0. So we simply set

$$\frac{\tilde{\mu}e[n]}{a + \|\mathbf{x}[n]\|^2} = \text{sgn}[e[n]] \tag{6.60}$$

where $\text{sgn}[e[n]]$ is given in (6.55). Note that whether $a + \|\mathbf{x}[n]\|^2 \leq \tilde{\mu}|e[n]|$ is true or not can be determined by employing one of the condition codes such as `LE` (see *DSP56300FM* p. A-250).

CASE 2a: If $a + \|\mathbf{x}[n]\|^2 > \tilde{\mu}|e[n]|$ and $a + \|\mathbf{x}[n]\|^2 < 1.0$, then we can immediately compute the division

$$\frac{\tilde{\mu}e[n]}{a + \|\mathbf{x}[n]\|^2}. \tag{6.61}$$

Note that whether $a + \|\mathbf{x}[n]\|^2 < 1.0$ is true can be determined by examining the extension bit of the status register (SR) or employing one of the condition codes such as `ES` (see *DSP56300FM* p. A-250).

CASE 2b: If $a + \|\mathbf{x}[n]\|^2 > \tilde{\mu}|e[n]|$ and $a + \|\mathbf{x}[n]\|^2 \geq 1.0$, then scale $\left[a + \|\mathbf{x}[n]\|^2\right]$ by $1/L$, i.e. `ASR #6,a,a` in this example and $\tilde{\mu}e[n]$ by $1/L$. This scaling will guarantee us a denominator value smaller than 1.0 but greater than the numerator and allow us to do a fixed-point division. Thus with proper scaling, we can now use the divide iteration and related `DIV` instruction. Note that one may use either the signed (four quadrant) division or, realizing that the denominator is always positive, carry out an unsigned division using $|e[n]|$ and then change sign according to $\text{sgn}[e[n]]$.

6.7 Code Management

This section will examine two methods to ease large coding projects. These methods include storing subroutines as separate files and writing macros.

6.7.1 Subroutines as Separate Files

If a program has many subroutines it may be easier to manage (and debug) these routines if they reside in their own files. The format for such a technique is as follows. First, code for the subroutine must be placed in a file. This file must include a label for the subroutine and an `RTS` to return control back to the main program. Next, the file must be included in the main program. Finally, all calls to the subroutine are made with the usual `JSR` instruction or variations on the `JSR` such as `JSSET`, `JCLR`, etc.... We illustrate the technique with an example. Consider the FIR filtering code in the file, `fir.asm`.

```
;--------------------------------------------------------------------------
;Purpose: This routine computes an inner product between an input sample
;  vector and a coefficient vector.
;Assumptions:
;  x0 = current input
;  r0 points to oldest input sample
;  r4 points to first FIR coefficient
;  a = output value
;Alters: x0, y0, a, r0, r4
;--------------------------------------------------------------------------
```

```
FIR_FILTER
        clr a      x0,x:(r0)+    y:(r4)+,y0
        do #N-1,FIR_FILTER_END
        mac        x0,y0,a       x:(r0)+,x0    y:(r4)+,y0
FIR_FILTER_END
        macr       x0,y0,a (r0)-
        rts
```

and the assembler directives in the file `project.dat`

```
N equ 10
        org x:$00000a
INPUT dsm N
        org y:$000000
COEFS dsm N
        org y:COEFS
        dc 0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1
```

Now we use the subroutine as follows (the following code is assumed to be in a file called `project.asm`).

```
       .
       .
       .
       move #INPUT,r0
       move #N-1,m0
       move #COEFS,r4
       move #N-1,m4
        .
        .
        .
       jsr FIR_FILTER
        .
        .
        .
       include 'fir.asm'
```

Note that the last include is usually at the bottom of `project.asm`.

6.7.2 Macros

The above FIR filter subroutine assumes an input vector pointed to by `r0` and a coefficient vector pointed to by `r4`. With any other address register configuration, the above subroutine is obsolete. Macros and their ability to pass arguments will allow us to make code segments more flexible and hence avoid the above obsolescence.

Introduction

Programming applications frequently involve the coding of a repeated pattern of a group of instructions. Some patterns contain variable entries which change for each repetition of the pattern. Macros provide a shorthand notation for handling these instruction patterns.

Once the iterated pattern and variable parameters are determined, the programmer can designate selected fields of any statement as variable. Thereafter, by invoking a macro, the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

Definition

The definition of a macro consists of three parts:

1. header which assigns a name to the macro and defines dummy arguments
2. body which contains DSP5630x instructions
3. terminator which ends the macro

The header is the `MACRO` assembler directive, its label, and the dummy argument list:

```
<label> MACRO [<dummy argument list>] [<comment>]
```

The label is the name by which the macro will be called. The dummy arguments are symbolic names that the macro processor will replace with arguments when the macro is called. Each dummy argument must obey the same rules as global symbol names. Will illustrate the use of macros with the following example taken from the *Motorola Assembler Reference*.

Macro Example

Assume the FIR filter macro is stored in a file called `fir.asm`.

```
FIR_FILTER MACRO X,H,N,Y
; This macro computes the inner product between the vector X and the vector H
; to yield output y according to
;
;   y[n] = x[n]h[0] + x[n-1]h(1) +... + x(n-N+1)h(N-1)
;
; X = base address of input vector in X memory
; H = base address of coefficient vector in Y memory
; N = number of coefficients
; Y = address of output in X memory
;
; Note: Assumes r6 is stack pointer.

        move r0,x:(r6)+ ;backup registers on stack
        move r4,x:(r6)+
        move a2,x:(r6)+
        move a1,x:(r6)+
        move a0,x:(r6)+
        move x0,x:(r6)+
        move y0,x:(r6)+
```

```
        move #X,r0
        move #H,r4
        clr a
        move x:(r0)+,x0 y:(r4)+,y0
        do #N-1,_end
        mac x0,y0,a x:(r0)+,x0 y:(r4)+,y0
_end    macr x0,y0,a
        move a,x:#Y

        move x:-(r6),y0 ;restore registers from stack
        move x:-(r6),x0
        move x:-(r6),a0
        move x:-(r6),a1
        move x:-(r6),a2
        move x:-(r6),r4
        move x:-(r6),r0
endm
```

The underscore as the first character in a label indicates that the label is local to the macro. All local labels within a macro must be unique. You do not always have to backup/restore registers before macro execution—only if you wish to preserve their pre-macro-call state. A macro is specified within a program by a macro call statement. The macro call statement is made up of three fields:

1. an optional label field which will correspond to the value of the location counter at the start of the macro
2. operation field contains the macro name
3. operand field which contains substitutable arguments

Within the operand field, each calling argument of a macro call corresponds one-to-one with a dummy argument of the macro definition. We call the macro as follows.

```
       .
       .
       .
       include 'fir.asm'
       move #LCOEFS,r0
       move #N-1,m0
       move #LINPUT,r4
       move #N-1,m4

       move #RCOEFS,r1
       move #N-1,m1
       move #RINPUT,r5
       move #N-1,m5
       .
       .
       .
```

```
move x0,x:(r0)  ;move left sample into left buffer
move y0,x:(r1)  ;move right sample into right buffer
FIR_FILTER r0,r4,N,OUTPUT
move (r0)-      ;point to oldest sample
move x:OUTPUT,a ;move left output to a
FIR_FILTER r1,r5,N,OUTPUT
move (r1)-      ;point to oldest sample
move x:OUTPUT,b ;move right output to b
.
.
.
```

Chapter 7

Wavetable Synthesis

7.1 Background

In this project, we will code a synthesizer capable of producing sounds similar to a musical instrument. The technique we will use is called "wavetable synthesis" and forms the basis behind virtually all electronic music synthesizers, PC sound cards, and arbitrary waveform generators (AWGs). In order to do this we will need to understand how to synthesize or generate waveforms (signals) with the DSP and the mechanics of sound generation (why certain instruments sound the way they do).

7.2 Sine Wave Synthesis

The generation of sinusoids (a.k.a. tones or waves) is a widely used function of DSPs in audio, communications, and control applications. High-speed, high-precision DSPs are capable of synthesizing stable and low distortion sinusoids of any frequency. These sinusoids can be produced digitally using either a truncated Taylor series or "Lookup Tables." We will examine the latter.

Suppose we calculate the values of L evenly-spaced points on a sinusoid and store them in memory to form a lookup table as in Figures 7.1 and 7.2. The frequency of the sinusoid produced at the D/A then depends upon the length of the lookup table, L; the sampling period, T; and the table increment, Δ between successive table accesses. We will discuss the table increment in more detail shortly.

Assume the sampling rate is $f_s = 1/T$ and the lookup table has L points. The fundamental table frequency (FTF) of the sinusoid generated from reading one table entry each sample period is simply

$$\begin{aligned} f_{fund} &= \frac{1 \text{ cycle}}{L \text{ samples}} \times \frac{1 \text{ sample}}{T \text{ seconds}} \\ &= \frac{1 \text{ cycle}}{LT \text{ seconds}} \end{aligned}$$

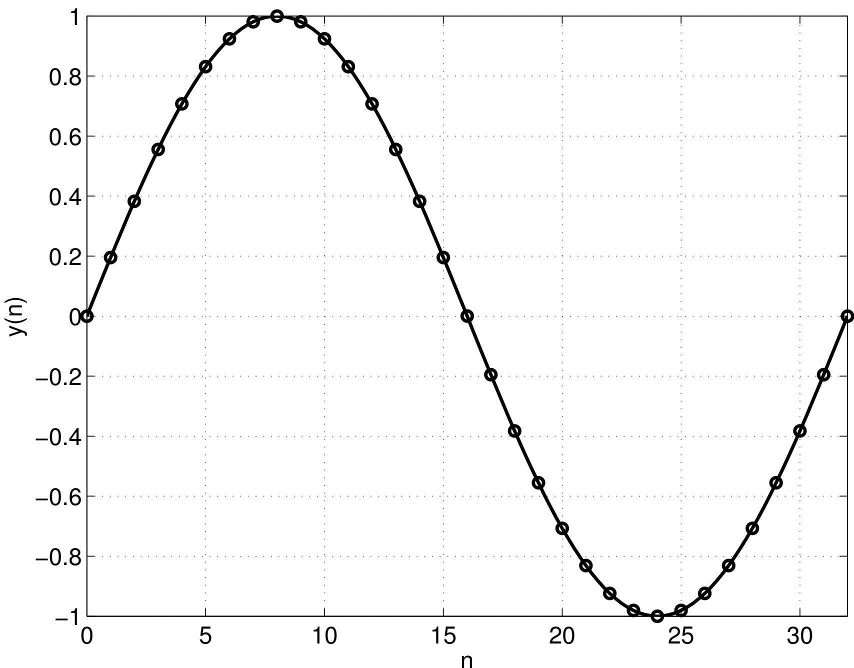


Figure 7.1: Sinusoid samples.

Address	Value
Base Address	0
Base Address + 1	$\sin(2\pi/L)$
Base Address + 2	$\sin(4\pi/L)$
	.
	.
	.
Base Address + $L - 1$	$\sin[2\pi(L-1)/L]$

Figure 7.2: Sinusoid samples stored in a length L lookup table.

$$= \frac{f_s}{L} \text{ (Hz)}. \quad (7.1)$$

We can also define the fundamental table period (FTP) of the sinusoid as

$$\begin{aligned} T_{fund} &= \frac{1}{f_{fund}} \\ &= \frac{L}{f_s}. \end{aligned} \quad (7.2)$$

Example: Assume the sampling rate is $f_s = 16,000$ samples/second and the lookup table has $L = 256$ entries. Then

$$\begin{aligned} f_{fund} &= 16,000/256 \\ &= 62.5\text{Hz}. \end{aligned} \quad (7.3)$$

and

$$\begin{aligned} T_{fund} &= 256/16,000 \\ &= 16 \times 10^{-3}\text{s}. \end{aligned} \quad (7.4)$$

■

If the table increment, Δ equals one, the table entries are read consecutively; if Δ equals two, every other table entry is read; if Δ equals three, every third table entry is read; and so on. In general, we cycle through the lookup table Δ times faster and thus the frequency of the sinusoid will be

$$\begin{aligned} f &= \frac{\Delta \text{ cycles}}{L \text{ samples}} \frac{1 \text{ sample}}{T \text{ seconds}} \\ &= \frac{\Delta}{LT} \\ &= \Delta f_{fund}. \end{aligned} \quad (7.5)$$

The maximum value Δ can assume is $L/2$ since at least two samples per cycle are required to synthesize a sinusoid without aliasing as a result of Nyquist theory. The total harmonic distortion (THD) of the synthesized sinusoid depends on the length of the table, L and the accuracy (number of bits of precision) of the data stored in the lookup table. Clearly, if Δ is an integer, the only frequencies permitted are integer multiples of the fundamental as in Table 7.1.

7.3 Generating the Lookup Table

The first step in synthesizing sinusoids is to create a lookup table containing the values of the sinusoid using

$$\text{Base_Address} + n = \sin(2\pi n/L) \quad (7.6)$$

where Base_Address is the base address of the lookup table. The following macro stored as `sinelut.asm` and called before the main event loop, will load the values defined by (7.6) into memory.

Table 7.1: Integer multiples of the fundamental table frequency.

Δ	f ($f_s = 16$ kHz)
1	62.5 Hz
2	125.0 Hz
$\vdots$	$\vdots$
$L/2$	8,000.0 Hz

SINELUT.ASM

```
1   sinelut  macro   table_length,table_base_address
2   ; sinelut      - macro name to generate sine lookup table
3   ; table_length - Length of Sine table
4   ; table_base_address - base address in Y memory of sine lookup table
5
6   pi      equu     3.141592654
7   freq    equ     2.0*pi/@cvf(table_length)
8
9           org y:table_base_address
10  count   set     0
11          dup     table_length
12          dc      @sin(@cvf(count)*freq)
13  count   set     count+1
14          endm
15          endm
16
```

Synthesizing the sinusoid from the lookup table can be accomplished in several ways as we will see in the next sections.

7.4 Sinusoid Synthesis with Integer Delta

The first step in the sinusoid synthesis with integer Δ is to define the lookup table length and allocate memory for the circular queue to store the table values. This is done in the `pass_sin.dat` file:

PASS_SIN.DAT

```
LUT_LENGTH   equ   256   ;LookUp Table length
.
.
.
     org y:$000000
SINE    dsm   LUT_LENGTH
DELTA   dc    8          ;integer delta
```

Next, in `pass_sin.asm` we insert lines to include the `sinelut.asm` macro file and call the macro in order to generate table values. We also include the `singenid.asm` file which generates the sinusoid sample:

PASS_SIN.ASM

```
.
.
.
        include 'pass_sin.dat'      ;include user data

        include 'sinelut.asm'
        sinelut  LUT_LENGTH,SINE    ;build sine lookup table with macro

        org     p:$100
.
.
.
        include 'ada_init.asm'
        include 'proginit.asm'
        include 'procster.asm'
        include 'singenid.asm'      ;integer delta
echo
        end
```

Next, we include lines to initialize a pointer into the table with the correct modifier and offset values in the `proginit.asm` file:

PROGINIT.ASM

```
1   proginit
2           move   #SINE,r4         ;initialize r4
3           move   #LUT_LENGTH-1,m4 ;initialize m4
4           bset   #0,x:FLAG
5           rts
6
```

Next, in `singenid.asm` we place our `sinusoid` subroutine in order to read the lookup table:

SINGENID.ASM

```
1   sinusoid
2           move    y:(r4)+n4,x0     ;read sample from table, post increment by delta
3           rts
4
```

Finally, we include a line to call the sinusoid subroutine in the `procster.asm` file:

PROCSTER.ASM

```
1   process_stereo
```

```
2           jbclr   #0,x:FLAG,HERE
3           move    y:DELTA,n4
4           bclr    #0,x:FLAG
5   HERE    jsr     sinusoid
6           rts
7
```

Note that the above code generates a wave with a fixed frequency by initializing the offset register `N4` with Δ in `proginit.asm`. If the developer wishes to change the frequency while the code is executing, `N4` would have to be reset with a new Δ (frequency) at an appropriate place in the main event loop.

If Δ is restricted to an integer, the sinusoid generator is limited to integer multiples of the FTF as given in (7.5). To generate arbitrary frequencies, we must allow for the possibility of a real-valued Δ, that is Δ composed of an integer and fractional part. When fractional values of Δ are used, required table values between table entries must be estimated. In the next two sections, we describe these estimation methods.

7.5 Sinusoid Synthesis with Real Delta: Round Down Method

The most straightforward way to estimating a table value between two table entries is to use the first of the two entries. This results in a truncation or rounding down of Δ.

Example: In Figure 7.3, we illustrate the round down method using $L = 8$, $\Delta = 2.5$. Here we note the first desired output coincides with a table entry (denoted by "+") so we simply use this value as indicated by "1". However, the second desired output (denoted by the first "o") lies between two table entries and so we use the entry indicated with the "2". This process continues for the next output point in a similar fashion. ■

Example: The following code segment illustrates how we can repeatedly add $\Delta = 2.5$ on the fixed-point DSP and extract the integer portion. The integer portion is kept in the upper half of the accumulator (`B1`) while the unsigned fractional portion is kept in the lower half of the accumulator (`B0`). ■

```
clr     b
move    #0.5,b0     ;b1 = $000000, b0 = $400000
asl     b           ;shift out sign bit b0 = $800000
move    #$2,b1      ;b1 = $000002, b0 = $800000
clr     a           ;a1 = $000000, a0 = $000000 (i.e. 0.0)
add     b,a         ;a1 = $000002, a0 = $800000 (i.e. 2.5)
add     b,a         ;a1 = $000005, a0 = $000000 (i.e. 5.0)
add     b,a         ;a1 = $000007, a0 = $800000 (i.e. 7.5)
```

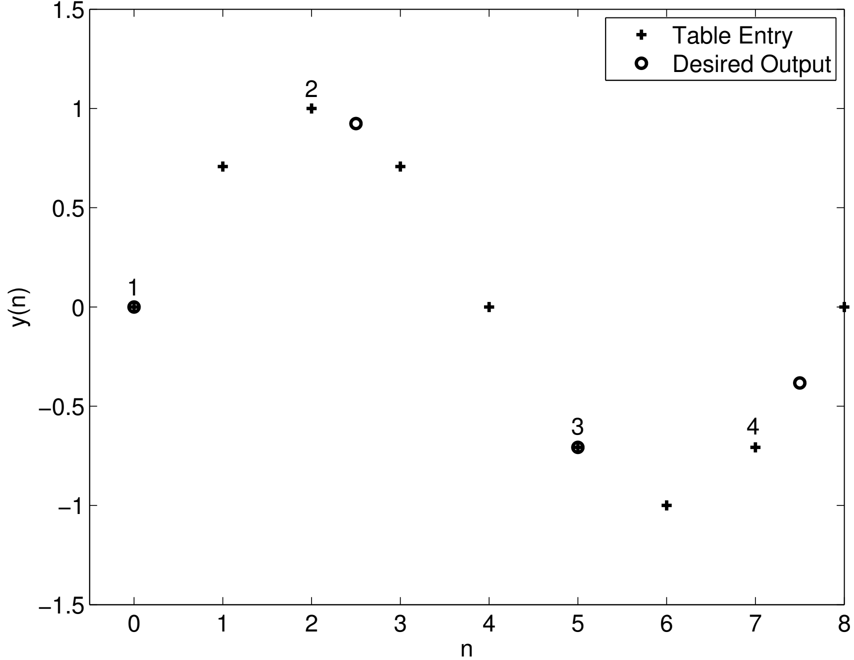


Figure 7.3: Sinusoid synthesis using the round down method.

Example: The following code segment illustrates how we can compute the addition given in the previous example modulo 8. We must add modulo 8 so as to properly wrap back around the lookup table and not go beyond its end. The code below requires that the table length be a power of 2. ■

```
add      b,a          ;a1 = $00000a, a0=$000000 (i.e. 10.0)
move     #>$7,x1      ;x1 = $000007 = %0000 ... 0000 0111
and      x1,a         ;a1 = $000002, a0 = $000000(i.e. 2.0)
```

7.5.1 Algorithm for Round Down Method

STORE_DELTA

Store integer portion of Δ in B1 and the fractional portion in B0. (We must make sure the fractional portion is stored properly in B0. When we move the fractional value in it is interpreted as a signed fraction. We must eliminate the sign bit with a left shift.)

CALCULATE_OFFSET

Calculate the positive integer-value used to load the offset register by adding the previous offset (integer and fractional parts) with Δ. (A1 will contain the integer portion of the offset and A0 will contain the unsigned fractional portion of the offset. Incrementing A by Δ is done on both the integer and fractional parts, i.e. A1 and A0.)

WRAP_POINTER

Wrap the pointer, A1 around when it is incremented past the length of the table, i.e. take A modulo L by masking (`AND`) A with $L-1$ where L is a power of two.

7.5.2 Code for Round Down Method

The first step in the round down method is to define the lookup table length and allocate memory for the circular queue to store the table values. This is done in the `pass_sin.dat` file:

PASS_SIN.DAT

```
LUT_LENGTH   equ   256   ;LookUp Table length
.
.
.
     org x:$00000a
DELTAI   dc   6          ;Integer portion of delta
.
.
.
     org y:$000000
SINE     dsm    LUT_LENGTH
DELTAF   dc     0.5      ;Fractional portion of delta
```

Next, in `pass_sin.asm` we insert lines to include the `sinelut.asm` macro file and call the macro in order to generate table values. We also include the `singenrd.asm` file which generates the sinusoid sample:

PASS_SIN.ASM

```
.
.
```

```
.
        include 'pass_sin.dat'      ;include user data

        include 'sinelut.asm'
        sinelut  LUT_LENGTH,SINE    ;build sine lookup table with macro

        org     p:$100
.
.
.
        include 'ada_init.asm'
        include 'proginit.asm'
        include 'procster.asm'
        include 'singenrd.asm'      ;round down
echo
        end
```

Next, we include lines to initialize pointers into the table and other registers as required in the round down method in the `proginit.asm` file:

PROGINIT.ASM

```
1   proginit
2           clr     b               ;get int, unsigned frac parts of delta in b1, b0...
3           move    y:DELTAF,b0     ;fractional part of delta in b0
4           asl     b               ;shift off sign bit of fractional part
5           move    x:DELTAI,b1     ;integer part of delta in b1
6           clr     a               ;a used to accumulate delta values
7           move    #SINE,r4        ;initialize r4
8           move    #0,n4           ;initialize n4
9           rts
10
```

Next, in `singenrd.asm` we place our `sinusoid` subroutine in order to read the lookup table using the round down method. Note that both accumulators, A and B as well as data register X1 must be preserved after initialization if they will be used elsewhere in a code. Otherwise, this routine will not function properly.

SINGENRD.ASM

```
1   sinusoid
2           add     b,a     y:(r4+n4),x0    ;add delta to a, read table value
3           and     #>(TABLE_LENGTH-1),a    ;wrap table pointer
4           nop                             ;stall
5           move    a1,n4                   ;update offset register with integer part,
6                                           ;i.e. round down for next time
7           rts
```

Finally, we include a line to call the sinusoid subroutine in the `procster.asm` file:

PROCSTER.ASM

```
1   process_stereo
2           jbclr   #0,x:FLAG,HERE
3           move    y:DELTA,n4
4           bclr    #0,x:FLAG
5   HERE    jsr     sinusoid
6           rts
7
```

Note that the above code generates a wave with a fixed frequency by initializing accumulator `B` with Δ in `proginit.asm`. If the developer wishes to change the frequency while the code is executing, `B` would have to be reset with a new Δ (frequency) at an appropriate place in the main event loop.

While the round down method provides a simple technique for estimating table values between entries, distortion in the sinusoid results from the rounding. An alternative which provides lower distortion is described next.

7.6 Sinusoid Synthesis with Real Delta: Linear Interpolation Method

In order to synthesize a sinusoid of any frequency with low distortion, an interpolation method must be used together with table lookup. By using interpolation, sinusoid values between table entries can be represented more accurately as in Figure 7.4. The simplest interpolation method is linear interpolation

$$y = mx + b \tag{7.7}$$

where

$$m = (\text{Base_Address} + l + 1) - (\text{Base_Address} + l) \tag{7.8}$$

$$b = (\text{Base_Address} + l) \tag{7.9}$$

$$x = \text{fractional part of accumulated } \Delta,\ 0 < x < 1.0 \tag{7.10}$$

$$y = \text{interpolated sample, } (\text{Base_Address} + l + x) \tag{7.11}$$

and (Base_Address+l) and (Base_Address+l+1) are the table entries on each side of the interpolated sample.

Example: Let the parameters for the sinusoid synthesis be $L = 8$, $\Delta = 2.5$, and Base_Address = 0. With the samples as illustrated in Figure 7.5, for $l = 2$ we have

$$m = 0.7071 - 1.0 = -0.2929$$
$$b = 1.0$$

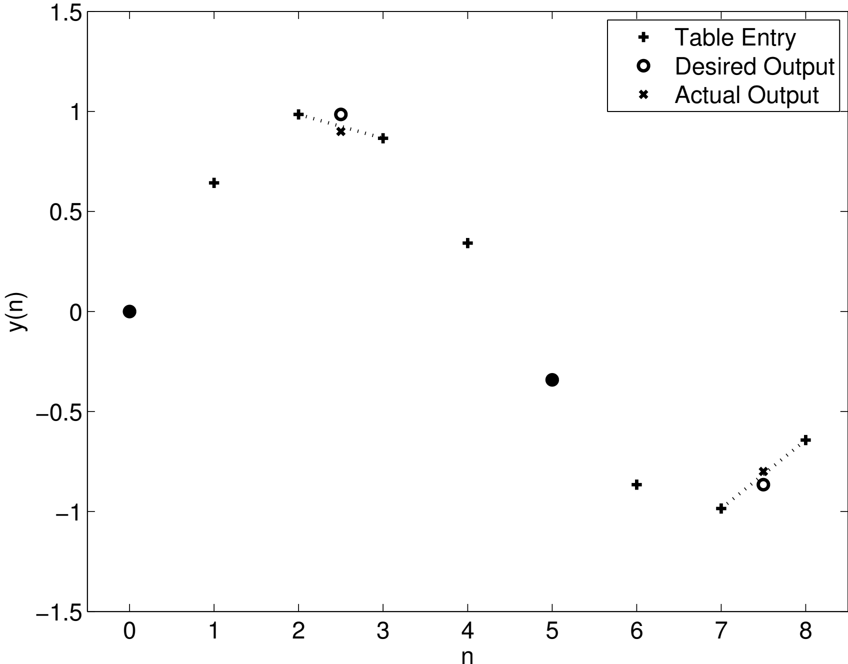


Figure 7.4: Sinusoid synthesis using the linear interpolation method.

$$\begin{aligned} x &= 0.5 \\ y &= mx + b = 0.5(-0.2929) + 1.0 = 0.8536. \end{aligned} \tag{7.12}$$

■

7.6.1 Algorithm for Linear Interpolation Method

We point `R4`, `R5` to the wavetable's Base_Address, Base_Address+1 respectively as in Figure 7.6; both offset registers, `N4` and `N5` will equal l in the figure. Δ will be stored in `Y1.Y0` with the integer part of Δ stored in `Y1` and the unsigned fractional part of Δ stored in `Y0`. The accumulated Δ, denoted $\sum\Delta$, taken modulo L will be stored in `A1.A0` with the integer part of $\sum\Delta$ stored in `A1` and the unsigned part of $\sum\Delta$ stored in `A0`. The value in `A1` will be taken as the offset to the base address of the table, l.

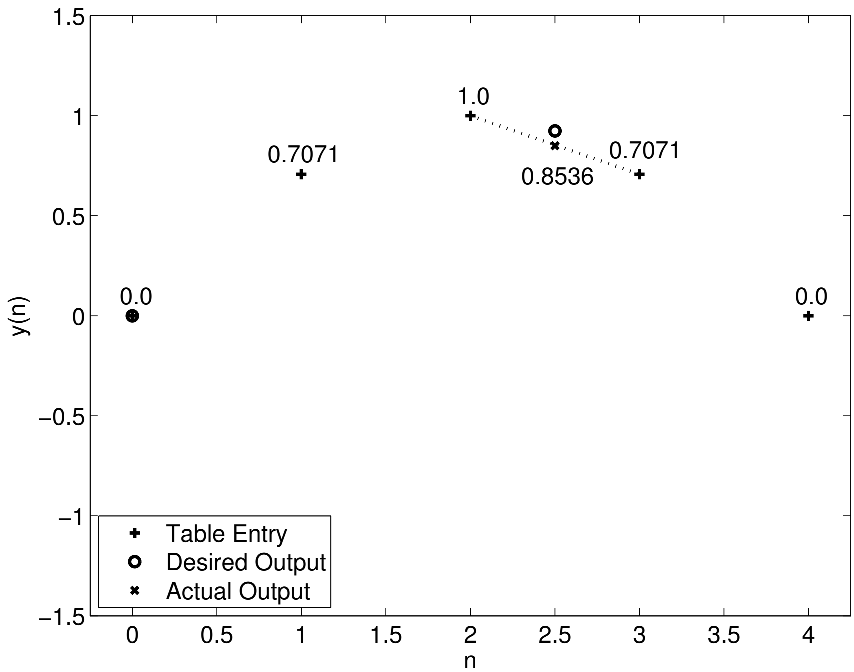


Figure 7.5: Sinusoid synthesis example using the linear interpolation method.

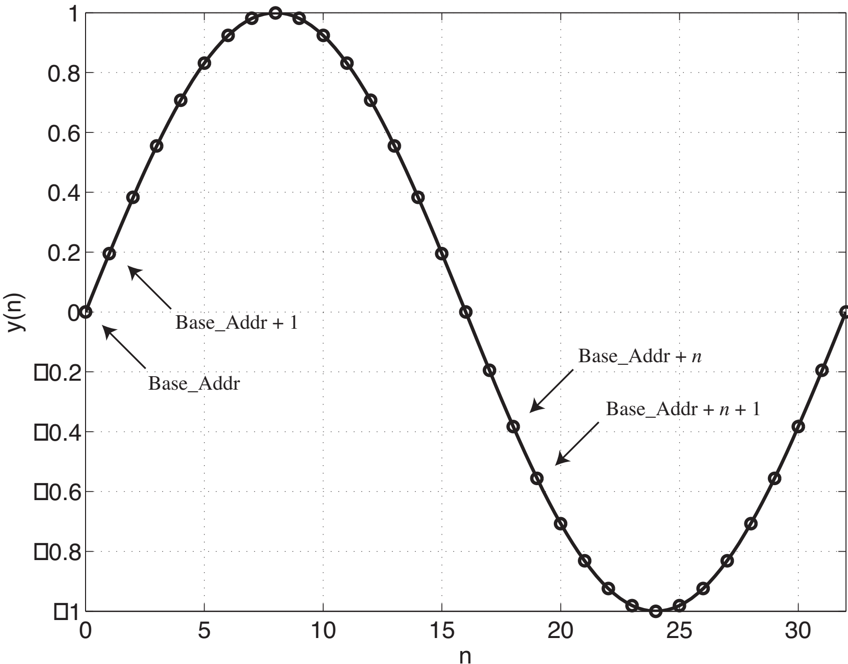


Figure 7.6: Addressing wavetable for linear interpolation algorithm.

INITIALIZE

Initialize `R4`, `R5` to Base_Address, Base_Address+1 respectively; `N4` and `N5` to zero; `M4` and `M5` to $L-1$; `Y1`, `Y0` to integer part, unsigned fractional part of Δ, respectively; and accumulator `A` (which will accumulate Δ) to zero.

CALCULATE_INTERPOLATED_VALUE

Interpolated sample value is calculated using (7.8) – (7.11). Note that x is taken to be `A0`, the signed fractional part of the accumulated Δ.

UPDATE_OFFSET_REGISTERS

Accumulate Δ modulo L in `A1.A0` where the integer part of $\sum\Delta$ stored in `A1` and the unsigned part of $\sum\Delta$ stored in `A0`. Update offset registers `N4` and `N5` with the value in `A1`.

7.6.2 Code for Linear Interpolation Method

The first step in the linear interpolation method is to define the lookup table length and allocate memory for the circular queue to store the table values. This is done in the `pass_sin.dat` file:

PASS_SIN.DAT

```
LUT_LENGTH   equ   256   ;LookUp Table length
.
.
.
     org x:$00000a
DELTAI   dc   6          ;Integer portion of delta
TMP      ds   1          ;temporary storage
.
.
.
     org y:$000000
SINE     dsm    LUT_LENGTH
DELTAF   dc     0.5      ;Fractional portion of delta
```

Next, in `pass_sin.asm` we insert lines to include the `sinelut.asm` macro file and call the macro in order to generate table values. We also include the `singenli.asm` file which generates the sinusoid sample:

PASS_SIN.ASM

```
.
.
.
        include 'pass_sin.dat'      ;include user data

        include 'sinelut.asm'
        sinelut  LUT_LENGTH,SINE    ;build sine lookup table with macro
```

```
        org     p:$100
.
.
.
        jsr     proginit            ;program initialization
        move    y0,x:TMP            ;backup y0 (for singenli.asm)
.
.
.
        move    x:TMP,y0            ;restore y0 (for singenli.asm)
main_loop
.
.
.
        include 'ada_init.asm'
        include 'proginit.asm'
        include 'procster.asm'
        include 'singenli.asm'      ;linear interpolation
echo
        end
```

Next, we include lines to initialize pointers into the table and other registers as required in the linear interpolation method in the `proginit.asm` file:

PROGINIT.ASM

```
1   proginit
2           move   x:DELTAI,y1      ;integer part of delta in y1
3           clr    b                ;get unsigned fractional part of delta in y0...
4           move   y:DELTAF,b0      ;fractional part of delta in b0
5           asl    b                ;shift off sign bit of fractional part
6           move   b0,y0            ;unsigned fractional part of delta in y0
7           clr     a               ;a used to accumulate delta values
8           move   #SINE,r4         ;initialize r4
9           move   #SINE+1,r5       ;initialize r5
10          move   #LUT_LENGTH-1,m4 ;initialize m4
11          move   #LUT_LENGTH-1,m5 ;initialize m5
12          move   #0,n4            ;initialize n4
13          move   #0,n5            ;initialize n5
14          rts
15
```

Next, in `singenli.asm` we place our `sinusoid` subroutine in order to read the lookup table using the linear interpolation method. Note that accumulator A as well as data registers Y0 and Y1 must be preserved after initialization if they will be used elsewhere in a code. Otherwise, this routine will not function properly.

SINGENLI.ASM

```
1   sinusoid
```

```
2           move    a0,b                 ;b1=fraction part of ptr, b0=0
3           lsr     b       y:(r4+n4),x0 ;b1=signed frac. delta, x0=prev. sine value
4           nop                          ;stall
5           move    b1,x1                ;x1=signed fraction
6           move    y:(r5+n5),b          ;b1=current sine value
7           sub     x0,b                 ;subtract sine values to obtain slope
8           nop                          ;stall
9           move    b,x0    y:(r4+n4),b  ;x0=slope value, b1=sine
10          macr    x0,x1,b              ;b=output sample approximation (m*x+b)
11          add     y,a                  ;update table pointer
12          and     #>(TABLE_LENGTH-1),a ;wrap table pointer around
13          move    b,x0                 ;output sample in x0
14          move    a1,n4                ;update offset register n4
15          move    a1,n5                ;update offset register n5
16          rts
17
```

Finally, we include a line to call the sinusoid subroutine in the `procster.asm` file:

PROCSTER.ASM

```
1   process_stereo
2           jbclr   #0,x:FLAG,HERE
3           move    y:DELTA,n4
4           bclr    #0,x:FLAG
5   HERE    jsr     sinusoid
6           rts
7
```

Note that the above code generates a wave with a fixed frequency by initializing registers `Y1`, `Y0` and with the integer, unsigned fractional part of Δ respectively in `proginit.asm`. If the developer wishes to change the frequency while the code is executing, `Y1` and `Y0` would have to be reset with a new Δ (frequency) at an appropriate place in the main event loop.

7.7 ADSR Dynamics

The sound output of musical instruments does not immediately build up to its full intensity (volume) nor does the sound fall to zero intensity instantaneously. It takes a certain amount of time for the sound to build up in intensity and a certain amount of time for the sound to die away.

The period of time during which a musical tone (sinusoid) is building up to some maximum amplitude is called the “attack time” and the time required for the tone’s intensity to partially die away is called its “decay time.” The time for final attenuation is called the “release time.” Many instruments allow the user to hold the tone for a period of time called the “sustain time.” The amplitude of the tones can “fit” inside a curve often called the Attack-Decay-Sustain-Release (ADSR) envelope, illustrated in Figure 7.7.

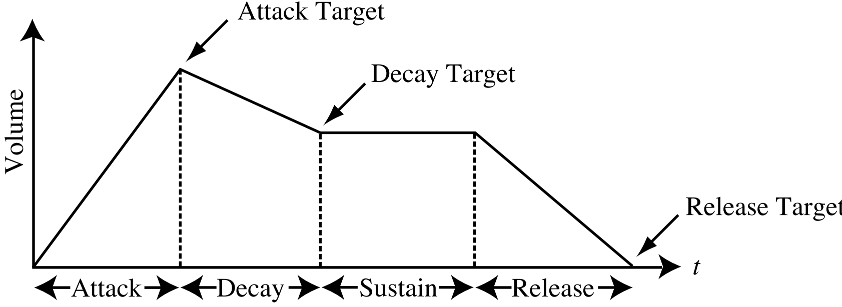


Figure 7.7: Classic ADSR envelope.

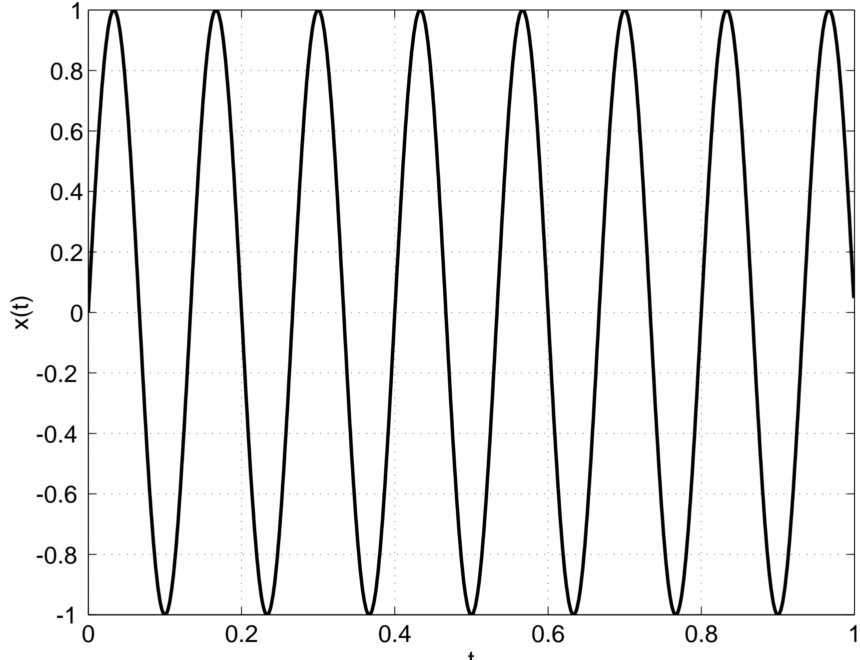


Figure 7.8: Basic sinusoid or tone.

A synthesizer duplicates the intensity variation of the tone by multiplying (modulating) the amplitude of the sinusoid (Figure 7.8) with a scale factor dictated by the ADSR envelope (Figure 7.9). The resulting waveform is shown in Figure 7.10 and the general technique is referred to as *amplitude modulation* (AM). The changes in sound intensity represented by ADSR are called "ADSR dynamics."

The shape of the envelope in Figure 7.9 will form a set of values used to scale the samples sequentially. Figures 7.11 and 7.12 illustrate basic ADSR envelopes for guitar and piano although in reality the envelopes and sound generation is far more complex. For each portion of the envelope (A, D, S, or R) there are two parameters needed for control: 1) target value, $\hat{a}$ and 2) rise/decay rate, g.

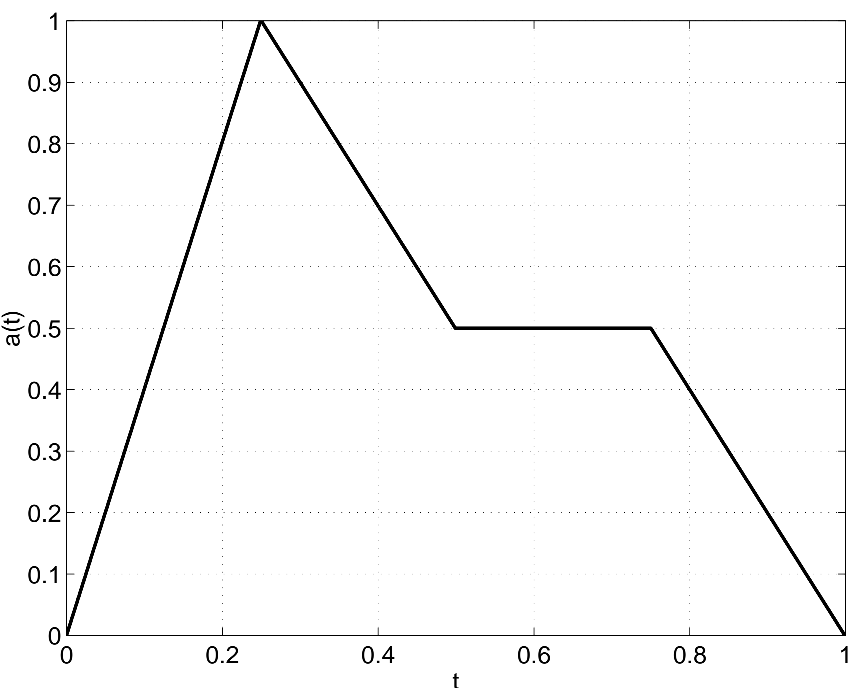


Figure 7.9: ADSR envlope used in modulating basic sinusoid.

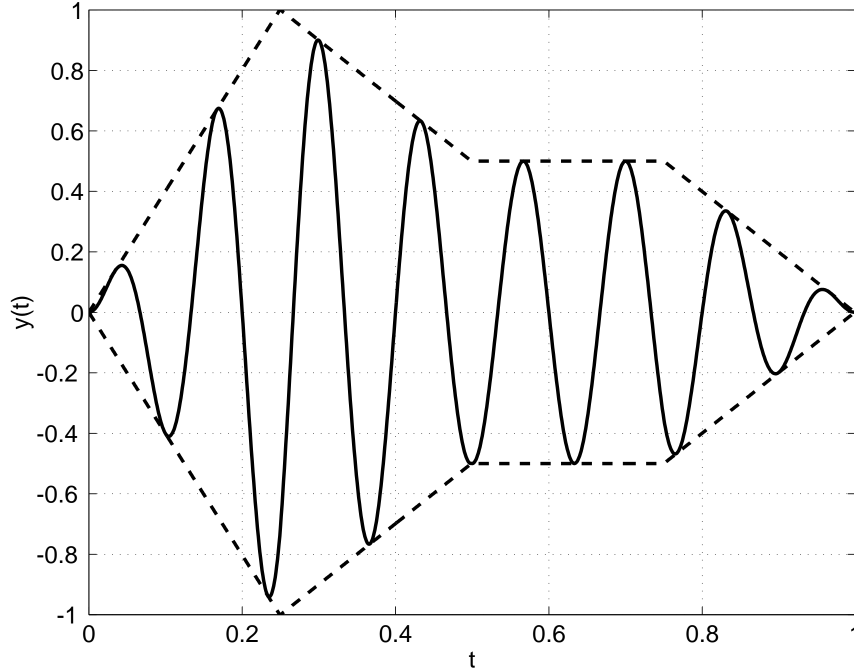


Figure 7.10: Modulated sinusoid using the ADSR envelope.

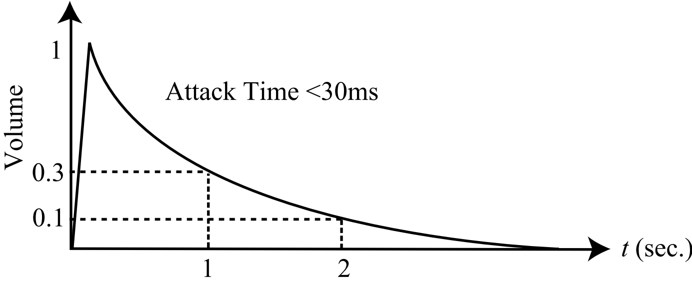


Figure 7.11: Guitar ADSR envelope.

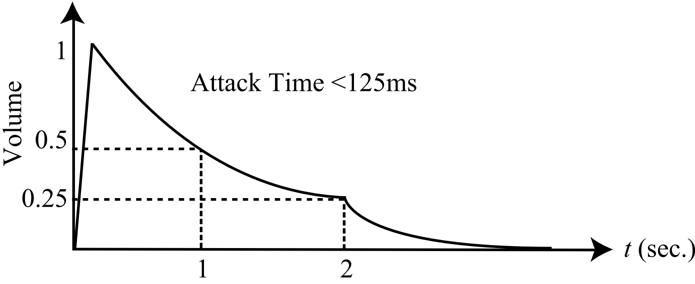


Figure 7.12: Piano ADSR envelope.

7.8 Envelope Generation

In order to scale samples with the correct ADSR values we either must store the ADSR values in memory (lookup table) or compute them "on-the-fly" by following some model of the ADSR.

Example: If we assume that the longest note we synthesize lasts one second and $f_s = 16$kHz, then we would require 16,000 words of storage for the ADSR envelope values. This would be impractical on the DSP56302EVM. ■

The alternative to the ADSR lookup table would be to compute ADSR values "on-the-fly." The computation for each portion of the envelope requires the use of an IIR filter whose difference equation is given by

$$a[n] \quad = \quad \hat{a}g + (1-g)a[n-1] \tag{7.13}$$

and which has a realization shown in Figure 7.13.

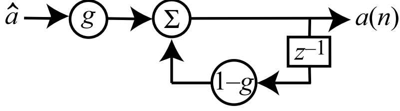


Figure 7.13: IIR filter for envelope generation.

Table 7.2: Attack phase values.

n	$a[n]$
0	0.9
1	0.99
2	0.999
$\vdots$	$\vdots$

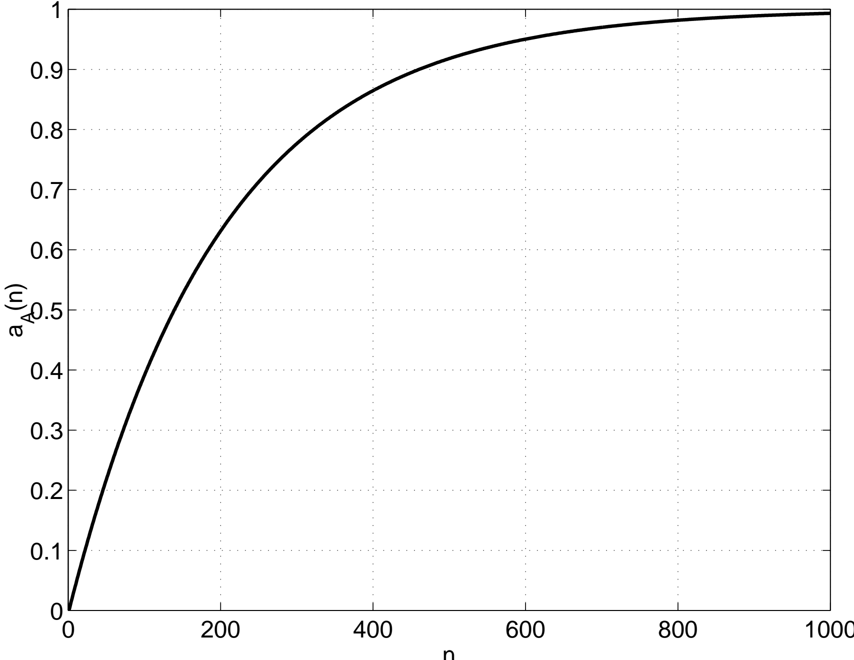


Figure 7.14: Sample attack portion of the ADSR envelope with exponential rise.

Example: Let the target value for the attack be $\hat{a}_A = 1.0$ and the rise/decay rate be $g = 0.9$. Assume $a[-1] = 0$. The first few envelope values (scale factors) for the attack portion are listed in Table 7.2. A sample attack signal (using different parameters) is illustrated in Figure 7.14. ■

Example: Let the target value for the release be $\hat{a}_R = 0.0$ and the rise/decay rate be $g = 0.9$. Assume $a[0] = 1.0$. The first few envelope values (scale factors) for the release portion are listed in Table 7.3. A sample release signal (using different parameters) is illustrated in Figure 7.15. ■

We note that the smaller g is, the longer it takes to reach the target value. When g approaches one, we reach the target value almost instantly.

Table 7.3: Release phase values.

n	$a[n]$
0	1.0
1	0.1
2	0.01
⋮	⋮

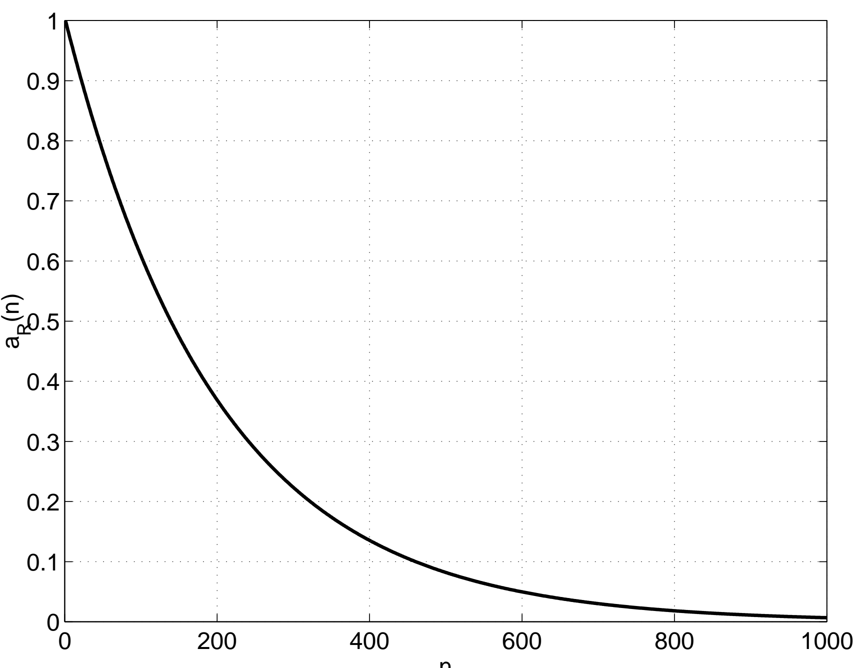


Figure 7.15: Sample release portion of the ADSR envelope with exponential decay.

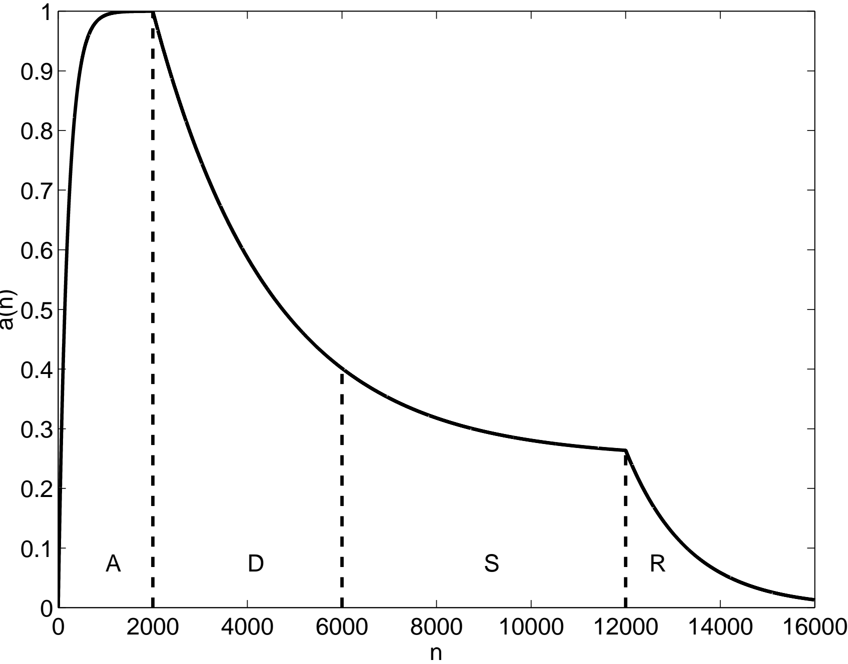


Figure 7.16: ADSR envelope with exponential rise/decay phases.

We will build the ADSR envelope by assembling rising/decaying exponentials as shown in Figure 7.16. Each portion of the envelope requires an initial value and a duration for how long to generate the values in each portion. Usually the final value of the previous portion becomes the initial value for the next portion so that we can have smooth transitions through the envelope. In addition we require rise/decay rates and target values. As described above, the envelope values will modulate the sinusoid. For applications involving variable note durations, usually the attack and release phases are fixed and the decay and sustain phases are lengthened or shortened depending on the note duration.

7.9 Synthesizing Music

A musical note is specified by three parameters: frequency, duration, and loudness. We will assume the middle A note is 440Hz (usual calibration for piano). Notes in the scale of C begin with middle C, D, E, F, G, A, B, and C where the final C is one octave higher than the middle-C are scaled with a doubling, halving in frequency for each octave increase (next scale), decrease (previous scale) respectively. Notations for scale and octave are shown in Figures 7.17 and 7.18 and frequency data for each note is given in Figure 7.19.

We will generate notes using the following steps.

1. The desired frequency will dictate Δ (offset) used in sinusoid synthesis

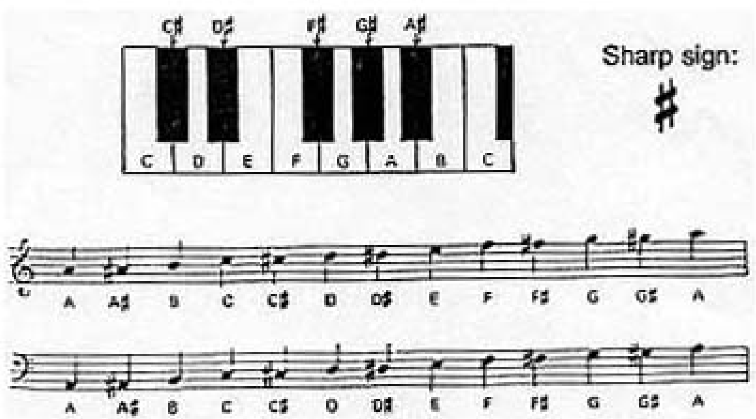


Figure 7.17: Musical notation.

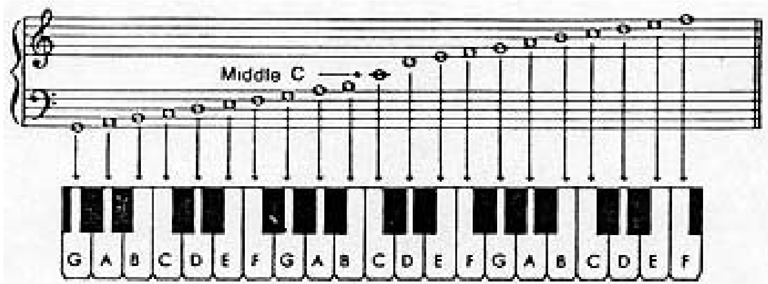


Figure 7.18: Musical scales. A C-scale would begin at middle C then play D, E, F, G, A, B, and C (one octave higher than middle C).

	OCTAVE NUMBER								
	0	1	2	3	4	5	6	7	8
C	16.3516	32.7032	65.4064	130.813	261.626	523.251	1046.50	2093.00	4186.01
C#	17.3239	34.6478	69.2957	138.591	277.183	554.365	1108.73	2217.46	4434.92
D	18.3540	36.7081	73.4162	146.832	293.665	587.330	1174.66	2349.32	4698.64
D#	19.4454	38.8909	77.7817	155.563	311.127	622.254	1244.51	2489.02	4978.03
E	20.6017	41.2034	82.4069	164.814	329.628	659.255	1318.51	2637.02	5274.04
F	21.8268	43.6536	87.3071	174.614	349.228	698.456	1396.91	2793.83	5587.65
F#	23.1247	46.2493	92.4986	184.997	369.994	739.989	1479.98	2959.96	5919.91
G	24.4997	48.9994	97.9989	195.998	391.995	783.991	1567.98	3135.96	6271.93
G#	25.9565	51.9131	103.826	207.652	415.305	830.609	1661.22	3322.44	6644.88
A	27.5000	55.0000	110.000	220.000	440.000	880.000	1760.00	3520.00	7040.00
A#	29.1352	58.2705	116.541	233.082	466.164	932.328	1864.66	3729.31	7458.62
B	30.8671	61.7354	123.471	246.942	493.883	987.767	1975.53	3951.07	7902.13

Figure 7.19: Note frequencies.

2. We will assume a duration of one second for all notes (in real music this is obviously not true)

3. An additional scale factor, loudness, will be used to further scale the synthesized tone for periods of low or high intensity

7.10 Implementation

To simplify the envelope processing a bit, we will merge the decay and sustain portions of the ADSR envelope into a single portion. Also we make use of flag bits (discussed in the next section) for timing control.

7.10.1 Algorithm

INITIALIZE

Set NEW_NOTE_FLAG, A_FLAG; Clear DS_FLAG, R_FLAG

Set ADSR_COUNTER = A_DURATION

MAIN (once per sample period)

If NEW_NOTE_FLAG

GET_NOTE_DATA

End

SYNTHESIZE_NOTE

CHECK_ADSR_COUNTER

CHECK_NOTE_COUNTER

Goto ***MAIN***

GET_NOTE_DATA

Get NOTE_DELTA (integer and fractional portions)

Set registers (`y0,y1,n4,n5`) for new NOTE_DELTA (see `proginit.asm`)

Get NOTE_DURATION

Set NOTE_COUNTER = NOTE_DURATION

Get NOTE_LOUDNESS

Clear NEW_NOTE_FLAG

SYNTHESIZE_NOTE

Based on DELTA, ***GET_SINUSOID_VALUE***

COMPUTE_ADSR_WEIGHT

TMP = (SINUSOID_VALUE)(ADSR_WEIGHT)

OUTPUT = (TMP)(NOTE_LOUDNESS)

CHECK_ADSR_COUNTER

Decrement ADSR_COUNTER (tracks progress through portions of envelope)

If ADSR_COUNTER = 0 (new ADSR portion)

If A_FLAG

Clear A_FLAG, set DS_FLAG, ADSR_COUNTER = DS_DURATION

Elseif DS_FLAG

Clear DS_FLAG, set R_FLAG, ADSR_COUNTER = R_DURATION

Else (must be R_FLAG)

Clear R_FLAG, set A_FLAG, ADSR_COUNTER = A_DURATION

End

End

CHECK_NOTE_COUNTER

Decrement NOTE_COUNTER (tracks progress through note)

If NOTE_COUNTER = 0

Set NEW_NOTE_FLAG

End

GET_SINUSOID_VALUE

Generate sinusoid value using the linear interpolation method

COMPUTE_ADSR_WEIGHT

If A_FLAG (in the attack portion of the ADSR envelope)

Get filter parameters, A_HAT and G for attack phase

Else If DS_FLAG (in the decay/sustain portion of the ADSR envelope)

Get filter parameters, A_HAT and G for decay/sustain phase

Else (in the release portion of the ADSR envelope)

Get filter parameters, A_HAT and G for release phase

End

ADSR_WEIGHT = (A_HAT)(G) + (1−G)(ADSR_WEIGHT)

7.10.2 Layout in Memory

The `wavesyn.dat` file is listed below. Note that the DAT file requires note and frequency definitions as well as corresponding delta values for the LUT. These definitions and values are contained in the `note_equ.dat` file also listed below

WAVESYN.DAT

```
1  ;***************************************************************************
2  ;WAVESYN.DAT: This data file is used with PROJECT.ASM to lay out memory.
3  ;***************************************************************************
4
5  ;**********
6  ; Equates
7  ;**********
8  ADA_OFF   equ   0         ;0 enables codec, 1 disables codec
9  AAR0      equ   $010931   ;split external (off-chip) 32K RAM into 16K X and 16K Y
10 AAR3      equ   $010925   ;beginning at $010000 p.4-6--4-8 DSP56302EVM UM
11
12 SRATE        equ  16000          ;Must correspond to value in ADA_EQU.ASM
13 TABLE_LENGTH equ  256            ;LookUp Table length
14
15 ;Note/Song Information
16 NUMBER_NOTES   equ   8           ;number of notes to be played (8 notes in key of C)
17 MAX_LOUDNESS   equ   0.9999999   ;maximum loudness scale factor
18 WHOLE          equ   SRATE          ;whole note length in samples (1 second)
19
20 ;ADSR Envelope Data (1 second envelope to go with 1 second whole note)
21 A_TARGET      equ   0.999999   ;attack target
22 DS_TARGET     equ   0.01       ;decay/sustain target
23 R_TARGET      equ   0.0        ;release target
24 A_G           equ   0.025      ;attack gain of IIR
25 DS_G          equ   0.00045    ;decay/sustain gain of IIR
26 R_G           equ   0.0005     ;release gain of IIR
27 A_TIME        equ   30                      ;attack duration in ms
```

```
28  R_TIME        equ   30                        ;release duration in ms
29  DS_TIME       equ   1000-A_TIME-R_TIME        ;decay/sustain duration in ms for whole note
30  A_DURATION    equ   @CVI(A_TIME*SRATE/1000)   ;attack duration in samples
31  DS_DURATION   equ   @CVI(DS_TIME*SRATE/1000) ;decay/sustain duration in samples for whole note
32  R_DURATION    equ   @CVI(R_TIME*SRATE/1000)   ;release duration in samples
33
34  ;*********
35  ;Note Data
36  ;*********
37     include 'note_equ.dat'   ; frequency definitions and delta values for notes
38
39  ;************
40  ; Data Memory
41  ;************
42  ;******************************************************************************
43  ;---Buffer for the CS4215
44  ;       The following lines of code are required for proper on-board audio
45  ;       codec operation.
46  ;******************************************************************************
47          org     x:$000000       ;On-chip X memory $000000 -- $001BFF
48  RX_BUFF_BASE    equ     *
49  RX_data_1_2     ds      1       ;data time slot 1/2 for RX ISR
50  RX_data_3_4     ds      1       ;data time slot 3/4 for RX ISR
51  RX_data_5_6     ds      1       ;data time slot 5/6 for RX ISR
52  RX_data_7_8     ds      1       ;data time slot 7/8 for RX ISR
53
54  TX_BUFF_BASE    equ     *
55  TX_data_1_2     ds      1       ;data time slot 1/2 for TX ISR
56  TX_data_3_4     ds      1       ;data time slot 3/4 for TX ISR
57  TX_data_5_6     ds      1       ;data time slot 5/6 for TX ISR
58  TX_data_7_8     ds      1       ;data time slot 7/8 for TX ISR
59
60  RX_PTR          ds      1       ;Pointer for rx buffer
61  TX_PTR          ds      1       ;Pointer for tx buffer
62
63          org x:$00000a
64  ;******************************************************************************
65  ;---On-chip X data goes here
66  ;******************************************************************************
67
68
69  ;******************************************************************************
70  ;---Software Stack
71  STACK   equ     * ;locate stack after last thing in on-chip X memory
72                    ;STACK is used in TXRX_ISR.ASM and must be allowed to grow
73
74  ;You must not write *anything* into x:$001C00 --x:$00FFFF (check .LST file)
75
76
77          org x:$010000   ;Off-chip X memory $010000 -- $013FFF
78  ;******************************************************************************
```

```
79   ;---Off-chip X data goes here
80   ;     If you load actual values in off-chip memory, i.e. "dc" then you must run
81   ;     pass.asm first so that off-chip memory can be configured before the load
82   ;     is attempted.  In this case, Debugger should have Reset on Load (Config
83   ;     menu) unchecked.
84   ;*****************************************************************************
85   NOTESI  dsm 3*NUMBER_NOTES
86       org x:NOTESI
87       dc  @CVI(C4),WHOLE,MAX_LOUDNESS ;integer delta note information, note duration, loudness
88       dc  @CVI(D4),WHOLE,MAX_LOUDNESS
89       dc  @CVI(E4),WHOLE,MAX_LOUDNESS
90       dc  @CVI(F4),WHOLE,MAX_LOUDNESS
91       dc  @CVI(G4),WHOLE,MAX_LOUDNESS
92       dc  @CVI(A4),WHOLE,MAX_LOUDNESS
93       dc  @CVI(B4),WHOLE,MAX_LOUDNESS
94       dc  @CVI(C5),WHOLE,MAX_LOUDNESS
95
96
97           org y:$000000   ;On-chip Y memory $000000 -- $001BFF
98   ;*****************************************************************************
99   ;---On-chip Y data goes here
100  ;*****************************************************************************
101  TABLE   dsm  TABLE_LENGTH
102
103  ;You must not write *anything* into y:$001C00 --y:$00FFFF (check .LST file)
104
105
106          org y:$010000   ;Off-chip Y memory $010000 -- $013FFF
107  ;*****************************************************************************
108  ;---Off-chip Y data goes here
109  ;     If you load actual values in off-chip memory, i.e. "dc" then you must run
110  ;     pass.asm first so that off-chip memory can be configured before the load
111  ;     is attempted.  In this case, Debugger should have Reset on Load (Config
112  ;     menu) unchecked.
113  ;*****************************************************************************
114  NOTESF  dsm NUMBER_NOTES
115      org y:NOTESF
116      dc  (C4-@CVI(C4))   ;fractional delta note information
117      dc  (D4-@CVI(D4))
118      dc  (E4-@CVI(E4))
119      dc  (F4-@CVI(F4))
120      dc  (G4-@CVI(G4))
121      dc  (A4-@CVI(A4))
122      dc  (B4-@CVI(B4))
123      dc  (C5-@CVI(C5))
```

NOTE_EQU.DAT

```
1   ;****************************************************************************
2   ;NOTE_EQU (WS): This file defines note frequencies and LUT delta values.
```

```
3   ;*************************************************************************
4
5   ;*********************
6   ;Frequency Definitions
7   ;*********************
8   ;Octave 0 data
9   C0_f    equ     16.3516 ;Octave 0 C
10  Cs0_f   equ     17.3239 ;Octave 0 C# (C Sharp)
11  D0_f    equ     18.354  ;Octave 0 D
12  Ds0_f   equ     19.4454 ;Octave 0 D# (D Sharp)
13  E0_f    equ     20.6017 ;Octave 0 E
14  F0_f    equ     21.8268 ;Octave 0 F
15  Fs0_f   equ     23.1247 ;Octave 0 F# (F Sharp)
16  G0_f    equ     24.4997 ;Octave 0 G
17  Gs0_f   equ     25.9565 ;Octave 0 G# (G Sharp)
18  A0_f    equ     27.5    ;Octave 0 A
19  As0_f   equ     29.1352 ;Octave 0 A# (A Sharp)
20  B0_f    equ     30.8671 ;Octave 0 B
21
22  ;Octave 1 data
23  C1_f    equ     2*C0_f
24  Cs1_f   equ     2*Cs0_f
25  D1_f    equ     2*D0_f
26  Ds1_f   equ     2*Ds0_f
27  E1_f    equ     2*E0_f
28  F1_f    equ     2*F0_f
29  Fs1_f   equ     2*Fs0_f
30  G1_f    equ     2*G0_f
31  Gs1_f   equ     2*Gs0_f
32  A1_f    equ     2*A0_f
33  As1_f   equ     2*As0_f
34  B1_f    equ     2*B0_f
35
36  ;Octave 2 data
37  C2_f    equ     2*C1_f
38  Cs2_f   equ     2*Cs1_f
39  D2_f    equ     2*D1_f
40  Ds2_f   equ     2*Ds1_f
41  E2_f    equ     2*E1_f
42  F2_f    equ     2*F1_f
43  Fs2_f   equ     2*Fs1_f
44  G2_f    equ     2*G1_f
45  Gs2_f   equ     2*Gs1_f
46  A2_f    equ     2*A1_f
47  As2_f   equ     2*As1_f
48  B2_f    equ     2*B1_f
49
50  ;Octave 3 data
51  C3_f    equ     2*C2_f
52  Cs3_f   equ     2*Cs2_f
53  D3_f    equ     2*D2_f
```

```
 54  Ds3_f   equ     2*Ds2_f
 55  E3_f    equ     2*E2_f
 56  F3_f    equ     2*F2_f
 57  Fs3_f   equ     2*Fs2_f
 58  G3_f    equ     2*G2_f
 59  Gs3_f   equ     2*Gs2_f
 60  A3_f    equ     2*A2_f
 61  As3_f   equ     2*As2_f
 62  B3_f    equ     2*B2_f
 63
 64  ;Octave 4 data
 65  C4_f    equ     2*C3_f
 66  Cs4_f   equ     2*Cs3_f
 67  D4_f    equ     2*D3_f
 68  Ds4_f   equ     2*Ds3_f
 69  E4_f    equ     2*E3_f
 70  F4_f    equ     2*F3_f
 71  Fs4_f   equ     2*Fs3_f
 72  G4_f    equ     2*G3_f
 73  Gs4_f   equ     2*Gs3_f
 74  A4_f    equ     2*A3_f
 75  As4_f   equ     2*As3_f
 76  B4_f    equ     2*B3_f
 77
 78  ;Octave 5 data
 79  C5_f    equ     2*C4_f
 80  Cs5_f   equ     2*Cs4_f
 81  D5_f    equ     2*D4_f
 82  Ds5_f   equ     2*Ds4_f
 83  E5_f    equ     2*E4_f
 84  F5_f    equ     2*F4_f
 85  Fs5_f   equ     2*Fs4_f
 86  G5_f    equ     2*G4_f
 87  Gs5_f   equ     2*Gs4_f
 88  A5_f    equ     2*A4_f
 89  As5_f   equ     2*As4_f
 90  B5_f    equ     2*B4_f
 91
 92  ;Octave 6 data
 93  C6_f    equ     2*C5_f
 94  Cs6_f   equ     2*Cs5_f
 95  D6_f    equ     2*D5_f
 96  Ds6_f   equ     2*Ds5_f
 97  E6_f    equ     2*E5_f
 98  F6_f    equ     2*F5_f
 99  Fs6_f   equ     2*Fs5_f
100  G6_f    equ     2*G5_f
101  Gs6_f   equ     2*Gs5_f
102  A6_f    equ     2*A5_f
103  As6_f   equ     2*As5_f
104  B6_f    equ     2*B5_f
```

```
105
106  ;Octave 7 data
107  C7_f    equ     2*C6_f
108  Cs7_f   equ     2*Cs6_f
109  D7_f    equ     2*D6_f
110  Ds7_f   equ     2*Ds6_f
111  E7_f    equ     2*E6_f
112  F7_f    equ     2*F6_f
113  Fs7_f   equ     2*Fs6_f
114  G7_f    equ     2*G6_f
115  Gs7_f   equ     2*Gs6_f
116  A7_f    equ     2*A6_f
117  As7_f   equ     2*As6_f
118  B7_f    equ     2*B6_f
119
120  ;Octave 8 data
121  C8_f    equ     2*C7_f
122  Cs8_f   equ     2*Cs7_f
123  D8_f    equ     2*D7_f
124  Ds8_f   equ     2*Ds7_f
125  E8_f    equ     2*E7_f
126  F8_f    equ     2*F7_f
127  Fs8_f   equ     2*Fs7_f
128  G8_f    equ     2*G7_f
129  Gs8_f   equ     2*Gs7_f
130  A8_f    equ     2*A7_f
131  As8_f   equ     2*As7_f
132  B8_f    equ     2*B7_f
133
134  ;*************************
135  ;Lookup Table Delta values
136  ;*************************
137  ;Octave 0 data
138  C0      equ     @CVF(C0_f*TABLE_LENGTH/SRATE)
139  Cs0     equ     @CVF(Cs0_f*TABLE_LENGTH/SRATE)
140  D0      equ     @CVF(D0_f*TABLE_LENGTH/SRATE)
141  Ds0     equ     @CVF(Ds0_f*TABLE_LENGTH/SRATE)
142  E0      equ     @CVF(E0_f*TABLE_LENGTH/SRATE)
143  F0      equ     @CVF(F0_f*TABLE_LENGTH/SRATE)
144  Fs0     equ     @CVF(Fs0_f*TABLE_LENGTH/SRATE)
145  G0      equ     @CVF(G0_f*TABLE_LENGTH/SRATE)
146  Gs0     equ     @CVF(Gs0_f*TABLE_LENGTH/SRATE)
147  A_0     equ     @CVF(A0_f*TABLE_LENGTH/SRATE)
148  As0     equ     @CVF(As0_f*TABLE_LENGTH/SRATE)
149  B_0     equ     @CVF(B0_f*TABLE_LENGTH/SRATE)
150
151  ;Octave 1 data
152  C1      equ     @CVF(C1_f*TABLE_LENGTH/SRATE)
153  Cs1     equ     @CVF(Cs1_f*TABLE_LENGTH/SRATE)
154  D1      equ     @CVF(D1_f*TABLE_LENGTH/SRATE)
155  Ds1     equ     @CVF(Ds1_f*TABLE_LENGTH/SRATE)
```

```
156  E1      equ     @CVF(E1_f*TABLE_LENGTH/SRATE)
157  F1      equ     @CVF(F1_f*TABLE_LENGTH/SRATE)
158  Fs1     equ     @CVF(Fs1_f*TABLE_LENGTH/SRATE)
159  G1      equ     @CVF(G1_f*TABLE_LENGTH/SRATE)
160  Gs1     equ     @CVF(Gs1_f*TABLE_LENGTH/SRATE)
161  A_1     equ     @CVF(A1_f*TABLE_LENGTH/SRATE)
162  As1     equ     @CVF(As1_f*TABLE_LENGTH/SRATE)
163  B_1     equ     @CVF(B1_f*TABLE_LENGTH/SRATE)
164
165  ;Octave 2 data
166  C2      equ     @CVF(C2_f*TABLE_LENGTH/SRATE)
167  Cs2     equ     @CVF(Cs2_f*TABLE_LENGTH/SRATE)
168  D2      equ     @CVF(D2_f*TABLE_LENGTH/SRATE)
169  Ds2     equ     @CVF(Ds2_f*TABLE_LENGTH/SRATE)
170  E2      equ     @CVF(E2_f*TABLE_LENGTH/SRATE)
171  F2      equ     @CVF(F2_f*TABLE_LENGTH/SRATE)
172  Fs2     equ     @CVF(Fs2_f*TABLE_LENGTH/SRATE)
173  G2      equ     @CVF(G2_f*TABLE_LENGTH/SRATE)
174  Gs2     equ     @CVF(Gs2_f*TABLE_LENGTH/SRATE)
175  A_2     equ     @CVF(A2_f*TABLE_LENGTH/SRATE)
176  As2     equ     @CVF(As2_f*TABLE_LENGTH/SRATE)
177  B_2     equ     @CVF(B2_f*TABLE_LENGTH/SRATE)
178
179  ;Octave 3 data
180  C3      equ     @CVF(C3_f*TABLE_LENGTH/SRATE)
181  Cs3     equ     @CVF(Cs3_f*TABLE_LENGTH/SRATE)
182  D3      equ     @CVF(D3_f*TABLE_LENGTH/SRATE)
183  Ds3     equ     @CVF(Ds3_f*TABLE_LENGTH/SRATE)
184  E3      equ     @CVF(E3_f*TABLE_LENGTH/SRATE)
185  F3      equ     @CVF(F3_f*TABLE_LENGTH/SRATE)
186  Fs3     equ     @CVF(Fs3_f*TABLE_LENGTH/SRATE)
187  G3      equ     @CVF(G3_f*TABLE_LENGTH/SRATE)
188  Gs3     equ     @CVF(Gs3_f*TABLE_LENGTH/SRATE)
189  A3      equ     @CVF(A3_f*TABLE_LENGTH/SRATE)
190  As3     equ     @CVF(As3_f*TABLE_LENGTH/SRATE)
191  B3      equ     @CVF(B3_f*TABLE_LENGTH/SRATE)
192
193  ;Octave 4 data
194  C4      equ     @CVF(C4_f*TABLE_LENGTH/SRATE)
195  Cs4     equ     @CVF(Cs4_f*TABLE_LENGTH/SRATE)
196  D4      equ     @CVF(D4_f*TABLE_LENGTH/SRATE)
197  Ds4     equ     @CVF(Ds4_f*TABLE_LENGTH/SRATE)
198  E4      equ     @CVF(E4_f*TABLE_LENGTH/SRATE)
199  F4      equ     @CVF(F4_f*TABLE_LENGTH/SRATE)
200  Fs4     equ     @CVF(Fs4_f*TABLE_LENGTH/SRATE)
201  G4      equ     @CVF(G4_f*TABLE_LENGTH/SRATE)
202  Gs4     equ     @CVF(Gs4_f*TABLE_LENGTH/SRATE)
203  A4      equ     @CVF(A4_f*TABLE_LENGTH/SRATE)
204  As4     equ     @CVF(As4_f*TABLE_LENGTH/SRATE)
205  B4      equ     @CVF(B4_f*TABLE_LENGTH/SRATE)
206
```

```
207   ;Octave 5 data
208   C5      equ     @CVF(C5_f*TABLE_LENGTH/SRATE)
209   Cs5     equ     @CVF(Cs5_f*TABLE_LENGTH/SRATE)
210   D5      equ     @CVF(D5_f*TABLE_LENGTH/SRATE)
211   Ds5     equ     @CVF(Ds5_f*TABLE_LENGTH/SRATE)
212   E5      equ     @CVF(E5_f*TABLE_LENGTH/SRATE)
213   F5      equ     @CVF(F5_f*TABLE_LENGTH/SRATE)
214   Fs5     equ     @CVF(Fs5_f*TABLE_LENGTH/SRATE)
215   G5      equ     @CVF(G5_f*TABLE_LENGTH/SRATE)
216   Gs5     equ     @CVF(Gs5_f*TABLE_LENGTH/SRATE)
217   A5      equ     @CVF(A5_f*TABLE_LENGTH/SRATE)
218   As5     equ     @CVF(As5_f*TABLE_LENGTH/SRATE)
219   B5      equ     @CVF(B5_f*TABLE_LENGTH/SRATE)
220
221   ;Octave 6 data
222   C6      equ     @CVF(C6_f*TABLE_LENGTH/SRATE)
223   Cs6     equ     @CVF(Cs6_f*TABLE_LENGTH/SRATE)
224   D6      equ     @CVF(D6_f*TABLE_LENGTH/SRATE)
225   Ds6     equ     @CVF(Ds6_f*TABLE_LENGTH/SRATE)
226   E6      equ     @CVF(E6_f*TABLE_LENGTH/SRATE)
227   F6      equ     @CVF(F6_f*TABLE_LENGTH/SRATE)
228   Fs6     equ     @CVF(Fs6_f*TABLE_LENGTH/SRATE)
229   G6      equ     @CVF(G6_f*TABLE_LENGTH/SRATE)
230   Gs6     equ     @CVF(Gs6_f*TABLE_LENGTH/SRATE)
231   A6      equ     @CVF(A6_f*TABLE_LENGTH/SRATE)
232   As6     equ     @CVF(As6_f*TABLE_LENGTH/SRATE)
233   B6      equ     @CVF(B6_f*TABLE_LENGTH/SRATE)
234
235   ;Octave 7 data
236   C7      equ     @CVF(C7_f*TABLE_LENGTH/SRATE)
237   Cs7     equ     @CVF(Cs7_f*TABLE_LENGTH/SRATE)
238   D7      equ     @CVF(D7_f*TABLE_LENGTH/SRATE)
239   Ds7     equ     @CVF(Ds7_f*TABLE_LENGTH/SRATE)
240   E7      equ     @CVF(E7_f*TABLE_LENGTH/SRATE)
241   F7      equ     @CVF(F7_f*TABLE_LENGTH/SRATE)
242   Fs7     equ     @CVF(Fs7_f*TABLE_LENGTH/SRATE)
243   G7      equ     @CVF(G7_f*TABLE_LENGTH/SRATE)
244   Gs7     equ     @CVF(Gs7_f*TABLE_LENGTH/SRATE)
245   A7      equ     @CVF(A7_f*TABLE_LENGTH/SRATE)
246   As7     equ     @CVF(As7_f*TABLE_LENGTH/SRATE)
247   B7      equ     @CVF(B7_f*TABLE_LENGTH/SRATE)
248
249   ;Octave 8 data
250   C8      equ     @CVF(C8_f*TABLE_LENGTH/SRATE)
251   Cs8     equ     @CVF(Cs8_f*TABLE_LENGTH/SRATE)
252   D8      equ     @CVF(D8_f*TABLE_LENGTH/SRATE)
253   Ds8     equ     @CVF(Ds8_f*TABLE_LENGTH/SRATE)
254   E8      equ     @CVF(E8_f*TABLE_LENGTH/SRATE)
255   F8      equ     @CVF(F8_f*TABLE_LENGTH/SRATE)
256   Fs8     equ     @CVF(Fs8_f*TABLE_LENGTH/SRATE)
257   G8      equ     @CVF(G8_f*TABLE_LENGTH/SRATE)
```

```
258   Gs8    equ    @CVF(Gs8_f*TABLE_LENGTH/SRATE)
259   A8     equ    @CVF(A8_f*TABLE_LENGTH/SRATE)
260   As8    equ    @CVF(As8_f*TABLE_LENGTH/SRATE)
261   B8     equ    @CVF(B8_f*TABLE_LENGTH/SRATE)
```

7.11 Using Flag Bits for Program Control

One common technique used in program control makes use of "flag bits." The idea here is that based on whether a bit or "flag" is set (1) or clear [0], we execute a set of instructions. On the DSP5630x, each word of memory can hold up to 24 flags. The use of flags for program control will aid in executing code particular to the current phase in the ADSR envelope. We illustrate the use of flag bits with an example.

Suppose we have three subroutines A, B, and C that need to be executed in the following way. First, subroutine A should be executed 10 times, then subroutine B should be executed 20 times, then subroutine C should be executed 30 times. We then repeat the entire execution sequence. We define the following in `pass.dat`.

```
A_EXECUTE   equ   0   ;flag bit positions
B_EXECUTE   equ   1
C_EXECUTE   equ   2

A_COUNT     equ   10  ;establish count for each code segment
B_COUNT     equ   20
C_COUNT     equ   30

   org x:
FLAG        dc   0   ;allocate one word for flag bits, all flags cleared

A_COUNTER   ds   1   ;allocate memory for counters
B_COUNTER   ds   1
C_COUNTER   ds   1
```

Next, we include lines to initialize the flag bits and counters in the `proginit.asm` file:

```
proginit
 bset #A_EXECUTE,x:FLAG ;set the A_EXECUTE FLAG
 bclr #B_EXECUTE,x:FLAG ;clear the B_EXECUTE FLAG
 bclr #C_EXECUTE,x:FLAG ;clear the C_EXECUTE FLAG
 move #>A_COUNT,x0
 move x0,x:A_COUNTER
 move #>B_COUNT,x0,
 move x0,x:B_COUNTER
 move #>C_COUNT,x0,
 move x0,x:C_COUNTER
 rts
```

Finally in `procster.asm` we include the following lines:

```
process_stereo
 jset #A_EXECUTE,x:FLAG,A_ROUTINE
 jset #B_EXECUTE,x:FLAG,B_ROUTINE
 jset #C_EXECUTE,x:FLAG,C_ROUTINE

A_ROUTINE
 <YOUR CODE HERE>
 clr a
 move x:A_COUNTER,a0
 dec a
 jne A_RETURN
 bclr #A_EXECUTE,x:FLAG
 bset #B_EXECUTE,x:FLAG
 move #>A_COUNT,a0            ;reset counter for next time
A_RETURN move a0,x:A_COUNTER
 jmp ROUTINE_RETURN


B_ROUTINE
 <YOUR CODE HERE>
 clr a
 move x:B_COUNTER,a0
 dec a
 jne B_RETURN
 bcl #B_EXECUTE,x:FLAG
 bset #C_EXECUTE,x:FLAG
 move #>B_COUNT,a0            ;reset counter for next time
B_RETURN move a0,x:B_COUNTER
 jmp ROUTINE_RETURN


C_ROUTINE
 <YOUR CODE HERE>
 clr a
 move x:C_COUNTER,a0
 dec a
 jne C_RETURN
 bclr #C_EXECUTE,x:FLAG
 bset #A_EXECUTE,x:FLAG
 move #>C_COUNT,a0            ;reset counter for next time
C_RETURN move a0,x:C_COUNTER
 jmp ROUTINE_RETURN

ROUTINE_RETURN rts            ;return to main event loop
```

7.12 Using the DSP56302 Triple Timer Module for Program Control

(To be completed at a later date)

Chapter 8

Modem

8.1 Background

In this project, we will build a crude 600 bits per second (bps) modem which will allow us to transmit and receive binary data. While not following a particular Consultative Committee for International Telephone and Telegraph (CCITT) standard, our modem will employ many of the techniques used in real modem design including BPSK modulation, asynchronous operation, match filtering, frame synchronization, start/stop bits, parity bits, etc.

8.2 Binary Phase-Shift Keying Modem (Transmitter)

> ***Definition:*** A *symbol* is a group of k bits. The symbol's size is the number of bits in the group.
>
> ***Definition:*** For $k = 1$ (symbol size of 1) the system is called *binary*.
>
> ***Definition:*** The set of all $M = 2^k$ symbols is called the *symbol set* or *alphabet*.

Digital modulation is the process by which symbols are transformed into waveforms or signals that are compatible with the characteristics of a channel, such as a telephone line. Typically, the desired information signal (data) modulates a carrier (sinusoid). A device that can *mo*dulate and *dem*odulate is called a *modem*. In general, modems can be categorized by their modulation type usually amplitude modulation (AM), frequency modulation (FM), or phase modulation (PM) or in some cases, a combination of these types. In digital communications, these modulation types are also known as amplitude shift keying (ASK), frequency shift keying (FSK), and phase shift keying (PSK) where the term "keying" is a remnant of telegraph transmission terminology.

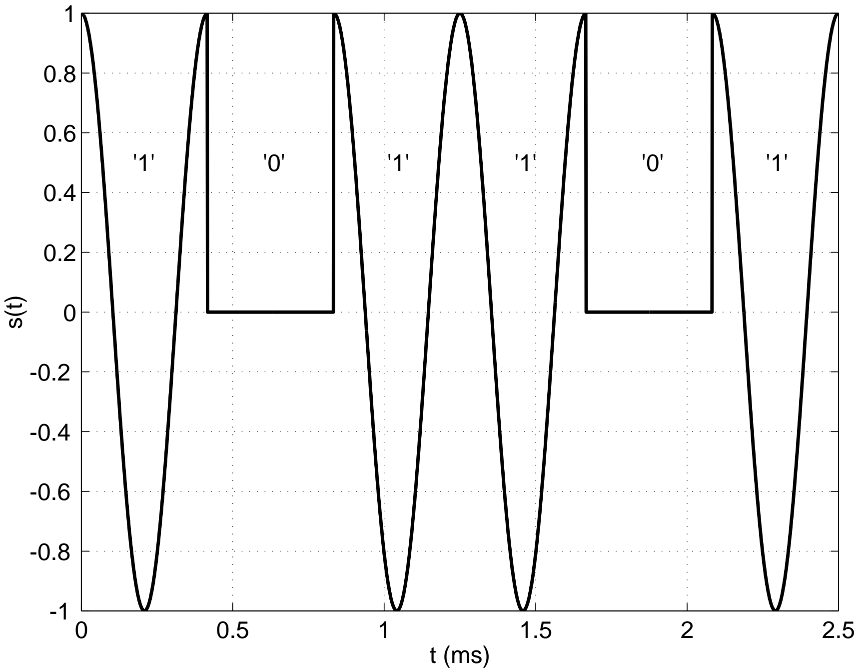


Figure 8.1: Amplitude Shift Keying signal.

8.2.1 Amplitude Shift Keying

The general form of amplitude modulation is

$$s(t) \quad = \quad a_i \cos(2\pi f t + \phi) \tag{8.1}$$

where the frequency, f and phase, ϕ are fixed and the amplitude, a_i has a different value for each symbol being transmitted. As an example in Figure 8.1, we choose a symbol set, $M = 2$ with $a_0 = 0$, $a_1 = 1$, $f = 2400$Hz, and $\phi = 0$. If the symbol takes on only one of two possible values (as in Figure 8.1)

$$a_i \quad \in \quad \{a_0, a_1\} \tag{8.2}$$

then this form of binary ASK is known as On-Off Keying (OOK).

8.2.2 Frequency Shift Keying

The general form of frequency modulation is

$$s(t) \quad = \quad a \cos(2\pi f_i t + \phi) \tag{8.3}$$

where the amplitude, a and phase, ϕ are fixed and the frequency, f_i has a different value for each symbol being transmitted. As an example in Figure 8.2, we choose a symbol set, $M = 2$ with $f_0 = 1200$Hz, $f_1 = 2400$Hz, $a = 1$, and $\phi = 0$.

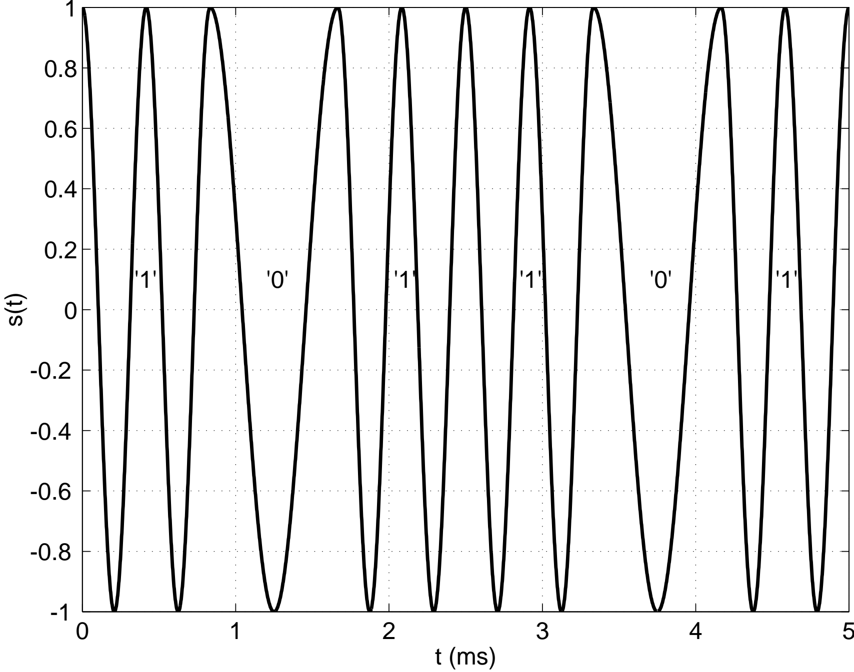


Figure 8.2: Frequency Shift Keying signal.

8.2.3 Phase Shift Keying

The general form of phase modulation is

$$s(t) = a\cos(2\pi f t + \phi_i) \tag{8.4}$$

where amplitude, a and frequency, f are fixed and phase, ϕ_i has a different value for each symbol being transmitted. As an example in Figure 8.3, we choose a symbol set, $M = 2$ with $\phi_0 = 0$, $\phi_1 = \pi$ radians, $a = 1$, and $f = 2400$Hz.

8.2.4 Other Comments

Figures 8.1, 8.2, and 8.3 illustrate the various forms of modulation for binary systems and are known as binary ASK (BASK) or OOK, binary FSK (BFSK), and binary PSK (BPSK) respectively.

> ***Definition:*** The *baud rate* is defined as the number of transmitted symbols per second.
>
> ***Definition:*** The *data rate*, R_b in bits per second (bps) is related to the baud rate by
>
> $$R_b = \text{(baud rate)(symbol size)}. \tag{8.5}$$

Note that if the symbol size is one (binary system) then the baud rate is the same as the data rate in bps. Unfortunately, as larger symbol sets have evolved, some sources still refer to the data rate as the baud rate which is incorrect.

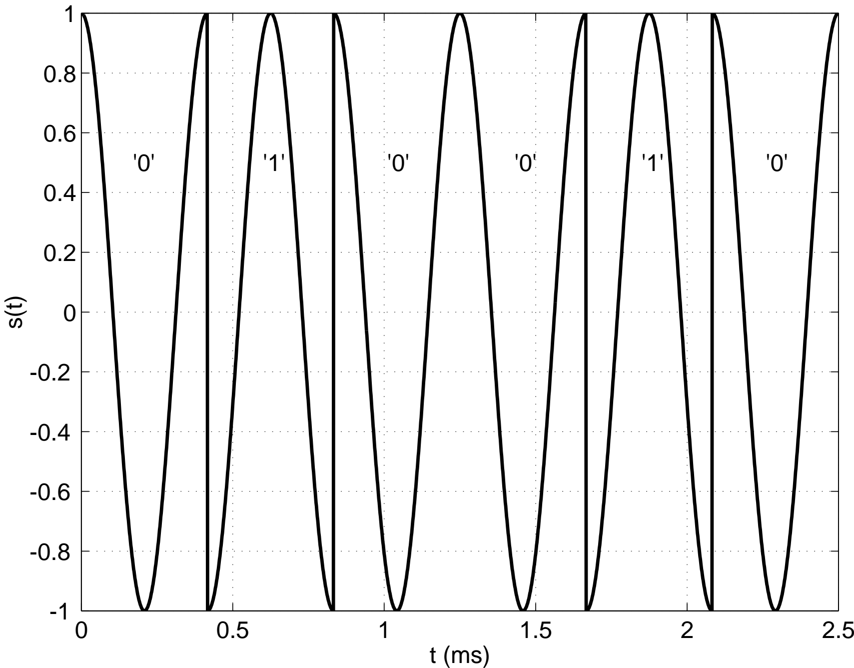


Figure 8.3: Phase Shift Keying signal.

Example: The Bell Type 103 modem was one of the first commercially-available modems in the marketplace. It consisted of a FSK modulator/demodulator transmitting 300 symbols per second (300 baud) with 1 bit per symbol for a data rate of 300bps. The standard consisted of

$f_0 = 1070$Hz ('0' or 'space') and $f_1 = 1270$Hz ('1' or 'mark') for the originating modem

$f_0 = 2025$Hz ('0' or 'space') and $f_1 = 2225$Hz ('1' or 'mark') for the answering modem.

Since the tones are in the audio band, this is sometimes referred to as Audio FSK (AFSK). Finally, the FSK modem is sometimes referred to as an FM modem. ■

8.2.5 Combined Modulation Schemes

One way to increase the data rate (for fixed symbol rate) is by increasing the symbol size through a combination of modulation schemes.

Example: The combination of BASK and BPSK also known as amplitude phase keying (APK) is described by

$$s(t) = a_i \cos(2\pi f t + \phi_j) \tag{8.6}$$

where f is fixed and a_i and ϕ_j have a different value for each bit being transmitted. For this case, a_i and ϕ_j each take on one value corresponding to a bit and so each pair can represent one of four possible two bit (dibit) combinations listed in Table 8.1. A sample APK signal is illustrated in Figure 8.4. If the baud rate were 2400 symbols/s and our symbol size, $M = 4$ (two bits/symbol) then the data rate would be 4800bps since each symbol represents two bits. ■

Table 8.1: Symbol table for APK.

Symbol	Amplitude Scaling, a_i	Phase Shift, ϕ_j
00	1.0	0
01	1.0	π
10	0.5	0
11	0.5	π

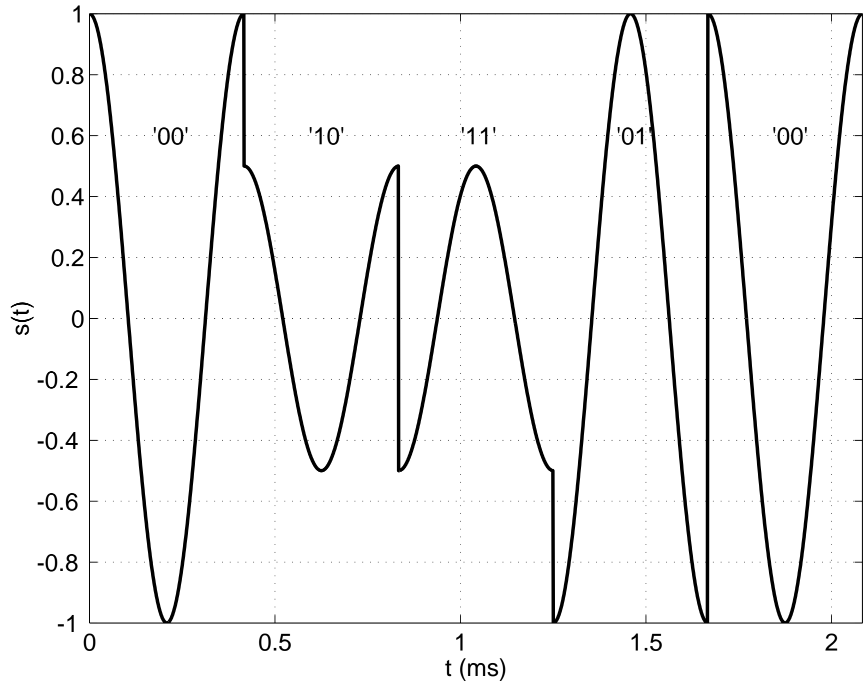


Figure 8.4: Amplitude Phase Keying signal.

8.3 Modem Design for the Telephone Channel

We begin with Shannon's *fundamental* Information Capacity Theorem:

Theorem: The information capacity [in bits per second (bps)], C of a continuous channel of bandwidth B hertz, perturbed by additive white Gaussian noise of power spectral density (PSD) $N_0/2$ and limited in bandwidth to B is given by

$$C = B \log_2 \left(1 + \frac{P}{N_0 B} \right) \tag{8.7}$$

where P is the average transmitted power. ■

A few comments are in order regarding the information capacity theorem. First, we see that the channel capacity is linearly proportional to the bandwidth, B, and logarithmically proportional to the SNR $[P/(N_0B)]$. Second, the above formula implies that for a given average transmitted power, P and channel bandwidth B, we can transmit information at a rate of C bits per second with an arbitrarily small probability of error. It is not possible to transmit at a rate higher than C bits per second without a definite probability of error. The channel capacity theorem defines the fundamental limit on the rate of error-free transmission for a power-limited, band-limited Gaussian channel. To approach this limit, however, the transmitted signal must have statistical properties approximating those of white Gaussian noise.

We note that the available bandwidth of the telephone channel (voice band) is from 300–3,400Hz and given typical SNR and transmitted power (regulated by the FCC), today's 56K modems (v.90) are near the capacity limit. New digital communications methods such as the Digital Subscriber Line (DSL) use out-of-band frequencies for higher data rates over twisted pair (TP) but require special terminating hardware at the telephone companys central office and home.

> ***Definition:*** A *simplex* connection is a one-way link, i.e. transmissions are made from terminal A to terminal B.
>
> ***Definition:*** A *half-duplex* connection is a link whereby transmission may be made in either direction but not simultaneously.
>
> ***Definition:*** A *full-duplex* connection is a two-way link where transmissions may proceed in both directions simultaneously.
>
> ***Definition:*** An *asynchronous* modem does not require synchronization (global clock) between the transmitter and receiver.

In an asynchronous modem, in order to let the receiver know that information is about to be transmitted, a "start" bit is first sent. Then the actual data bits are sent followed by a parity bit (used for error detection/control) and one or two "stop" bits. This group of bits is sometimes called a "character." The parity bit takes on a value of zero or one as needed to insure that the summation

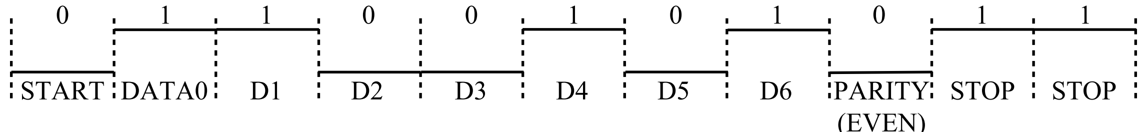


Figure 8.5: Sample 11 bit character frame for the letter "S", ASCII value 83 or %1010011

(using modulo 2 arithmetic) of all the bits in the data word plus the parity bit yields a zero (even parity) or one (odd parity). In this way, if an error occurs in transmission of data bits, it can be detected (error-detection) by the receiver since the parity will not be correct. If the added parity bit is designed to yield an even result, the method is termed *even parity*, and if designed to yield an odd result, it is termed *odd parity*.

In order to illustrate what the serial bit stream would look like at the input to the originating modem, consider the 7-bit American Standard Code for Information Exchange (ASCII) code for the letter "S." The ASCII code for "S" is 83 (decimal) or %1010011 in binary form and forms the value for data bits D0-D6 as in Figure 8.5. These data bits are embedded in an 11 bit frame which includes a start bit of '0', two stop bits of '1'-'1', and an even parity bit of '0' as in Figure 8.5. Finally, ASCII tables and calculators are widely available on the Internet to aid in standard conversion of letters to ASCII code.

8.4 Design Specifications BPSK Modem (Transmitter)

In the transmitter portion of this project, we will assume an asynchronous, simplex modem operating at a data rate of 600bps using a BPSK modulation. The carrier (sinusoid) will have a frequency of 2400Hz and be digitally synthesized using a 9600Hz sampling rate. If we employ a 256-point sinusoid lookup table, this implies an integer-valued table increment, $\Delta = 64$ and hence we may use the `singenid.asm` subroutine. With these parameters, we have

$$\frac{9600 \text{ samples}}{\text{s}} \times \frac{\text{s}}{600 \text{ bits}} = 16 \, \frac{\text{samples}}{\text{bit}} \tag{8.8}$$

$$\frac{9600 \text{ samples}}{\text{s}} \times \frac{\text{s}}{2400 \text{ cycles}} = 4 \, \frac{\text{samples}}{\text{cycle}} \tag{8.9}$$

$$\frac{16 \text{ samples}}{\text{bit}} \times \frac{\text{cycle}}{4 \text{ samples}} = 4 \, \frac{\text{cycles}}{\text{bit}} \tag{8.10}$$

We will assume the data word to be transmitted is a 7-bit ASCII character. Combined with a single start bit of '0', two stop bits of '1'-'1', and an even parity bit. The transmitted signal for the character frame in Figure 8.5 is illustrated in Figure 8.6. Note that this figure shows only one cycle per bit but as mentioned above, there are actually four cycles per bit with the given parameters.

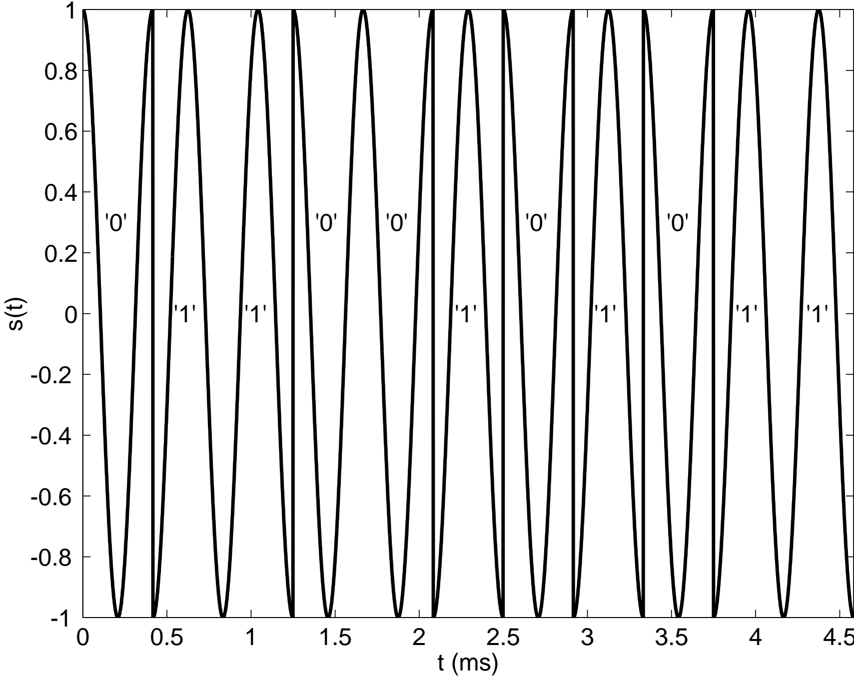


Figure 8.6: Transmitted signal corresponding to the character frame of Figure 8.5

8.5 Implementation–Transmitter

8.5.1 Algorithm

MAIN1 (once per sample period; for 5 seconds)

    SYNTHESIZE_WAVEFORM *for BIT '1'*

    Goto ***MAIN1***

INITIALIZE

    Set DATA_FLAG

MAIN2 (once per sample period)

    If DATA_FLAG

        GET_ASCII_CODE

        CALCULATE_PARITY_BIT

        BUILD_FRAME (see Figure 8.5)

        Set BIT_FLAG

    End

    If BIT_FLAG

        GET_BIT

    End

    SYNTHESIZE_WAVEFORM *for current BIT*

    Decrement SAMP_COUNTER

    CHECK_SAMP_COUNTER

    CHECK_BIT_COUNTER

    Goto ***MAIN2***

GET_ASCII_CODE

    Read next ASCII code

    Clear DATA_FLAG

    BIT_COUNTER = 11

GET_BIT

Read next BIT in frame

Clear BIT_FLAG

SAMP_COUNTER = SAMPS_PER_BIT

SYNTHESIZE_WAVEFORM

COMPUTE_SINUSOID_SAMPLE (use `sinwgid.asm`), x

If (DATA_BIT == '1')

$x = -x$ (phase change)

End

CHECK_SAMP_COUNTER

If (SAMP_COUNTER == 0)

Set BIT_FLAG

Decrement BIT_COUNTER

End

CHECK_BIT_COUNTER

If (BIT_COUNTER == 0)

Set DATA_FLAG

End

Note that the digital synthesis of the carrier should maintain continuous phase. Thus you should not reset lookup table pointer to base address of the table for each new transmitted bit. You must maintain coherency. In addition, while we have expressed the various modulations schemes using cos(), the modulation can also use sin() for which the lookup table macro was given in Chapter 7.

8.5.2 Layout in Memory

BPSKTX.DAT

```
1  ;***************************************************************************
2  ;BPSKTX.DAT: This data file is used with PROJECT.ASM to lay out memory.
3  ;***************************************************************************
4
5  ;**********
6  ; Equates
7  ;**********
```

```
8   ADA_OFF equ     0       ;0 enables codec, 1 disables codec
9   AAR0    equ     $010931 ;split external (off-chip) 32K RAM into 16K X & 16K Y
10  AAR3    equ     $010925 ;beginning at $010000 p.4-6--4-8 DSP56302EVM UM
11
12  LUT_LENGTH      equ     4                ;lookup table length
13  Fs              equ     9600             ;must correspond to value in ADA_INIT.ASM
14  Fc              equ     2400             ;carrier frequency
15  DELTA           equ     Fc*LUT_LENGTH/Fs ;DELTA should be an integer
16  N               equ     Fs/fc            ;matched filter length
17  BPS             equ     300              ;modem date rate (bits per second)
18  SAMPS_BIT       equ     Fs/BPS           ;number of samples per bit
19  THRESHOLD       equ     0.1              ;threshold for bit decision
20  NUM_CHARS       equ     10
21
22  ;************
23  ; Data Memory
24  ;************
25  ;*****************************************************************************
26  ;---Buffer for the CS4215
27  ;       The following lines of code are required for proper on-board audio
28  ;       codec operation.
29  ;*****************************************************************************
30          org x:0 ;On-chip X memory $000000 -- $001BFF
31  RX_BUFF_BASE    equ     *
32  RX_data_1_2     ds      1       ;data time slot 1/2 for RX ISR
33  RX_data_3_4     ds      1       ;data time slot 3/4 for RX ISR
34  RX_data_5_6     ds      1       ;data time slot 5/6 for RX ISR
35  RX_data_7_8     ds      1       ;data time slot 7/8 for RX ISR
36
37  TX_BUFF_BASE    equ     *
38  TX_data_1_2     ds      1       ;data time slot 1/2 for TX ISR
39  TX_data_3_4     ds      1       ;data time slot 3/4 for TX ISR
40  TX_data_5_6     ds      1       ;data time slot 5/6 for TX ISR
41  TX_data_7_8     ds      1       ;data time slot 7/8 for TX ISR
42
43  RX_PTR          ds      1       ;Pointer for rx buffer
44  TX_PTR          ds      1       ;Pointer for tx buffer
45
46
47          org x:$00000a
48  ;*****************************************************************************
49  ;---On-chip X data goes here
50  ;*****************************************************************************
51  CHARS   dsm     NUM_CHARS
52          org x:CHARS
53          dc      'A','B','C','D','E','F','G','H','I','J'
54
55
56  ;*****************************************************************************
57  ;---Software Stack
58  STACK   equ     *       ;locate stack after last thing in on-chip X memory
```

```
59                              ;STACK is used in TXRX_ISR.ASM and must be allowed to grow
60
61   ;You must not write *anything* into x:$001C00 --x:$00FFFF (check .LST file)
62
63
64           org x:$010000   ;Off-Chip X memory $010000 -- $013FFF
65   ;*****************************************************************************
66   ;---Off-chip X data goes here
67   ;       If you load actual values in off-chip memory, i.e. "dc" then you must run
68   ;       pass.asm first so that off-chip memory can be configured before the load
69   ;       is attempted.  In this case, Debugger should have Reset on Load (Config
70   ;       menu) unchecked.
71   ;*****************************************************************************
72
73
74           org y:$000000   ;On-chip Y memory $000000 -- $001BFF
75   ;*****************************************************************************
76   ;---On-chip Y data goes here
77   ;*****************************************************************************
78
79
80   ;You must not write *anything* into y:$001C00 --y:$00FFFF (check .LST file)
81
82
83           org y:$010000   ;Off-chip Y memory $010000 -- $013FFF
84   ;*****************************************************************************
85   ;---Off-chip Y data goes here
86   ;       If you load actual values in off-chip memory, i.e. "dc" then you must run
87   ;       pass.asm first so that off-chip memory can be configured before the load
88   ;       is attempted.  In this case, Debugger should have Reset on Load (Config
89   ;       menu) unchecked.
90   ;*****************************************************************************
91
```

8.6 Binary Phase-Shift Keying Modem (Receiver)

8.6.1 Maximum-Likelihood Detector

Coherent detection of any received digital communications signal can be accomplished with a correlating detector also referred to as a maximum likelihood (ML) detector and illustrated in Figure 8.7. Here the received signal is denoted $r(t)$ and the BPSK waveform for bit i (0 or 1) is given by

$$s_i(t) = \sqrt{2E/T}\cos(\Omega_c t + \phi_i) \tag{8.11}$$

where E is the signal energy per symbol, T is the symbol duration, Ω_c is the carrier frequency, and $\phi_0 = 0$ or $\phi_1 = \pi$. It can be shown that in the binary symbol set case

$$E\{z_0|s_0\} \quad = \quad \sqrt{E}$$

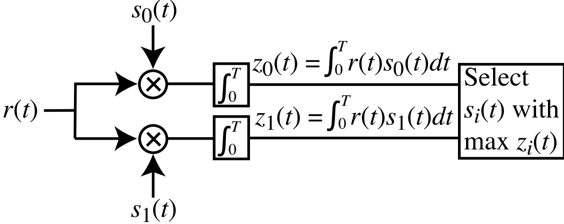


Figure 8.7: Maximum likelihood detector.

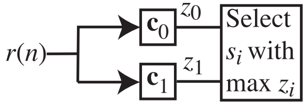


Figure 8.8: Digital matched filters.

$$\begin{aligned} E\{z_1|s_0\} &= -\sqrt{E} \\ E\{z_0|s_1\} &= -\sqrt{E} \\ E\{z_1|s_1\} &= \sqrt{E} \end{aligned} \tag{8.12}$$

where $E\{z_i|s_j\}$ is read as "the expected value of z_i given s_j was transmitted." The decision stage then selects (decides) which symbol (or bit in the binary case) was transmitted given the largest value of $z_i(nT)$, i.e.

$$\text{symbol} = \begin{cases} s_0, & z_0 > 0 \\ s_1, & z_1 > 0 \end{cases} . \tag{8.13}$$

Note that in the binary case, the outputs of the ML detectors will always be opposite in sign as in (8.12). Thus we need only one detector since we can base the symbol decision simply on the sign of either $z_0(t)$ or $z_1(t)$.

8.6.2 Sampled Matched Filter

Discrete-time matched filters can be used to provide the correlator outputs, z_i used in the ML detector as in Figure 8.8. In the discrete-time case, the N coefficients of the ith matched filter, $\mathbf{c}_i$ are taken as the time-reversed (and shifted by $N-1$ samples for causality) samples of $\mathbf{s}_i$

$$c_{i,k} = \begin{cases} s_i[N-1-k], & 0 \le k \le N-1 \\ 0, & \text{otherwise.} \end{cases} \tag{8.14}$$

It can be shown that the matched filter is optimal in the sense of maximizing the SNR of the received signal, $r(t)$. For the BPSK receiver, we need only build one matched filter (matched to the signal corresponding to '1' for example) in the detector since coefficients matched to the other signal (corresponding to '0' for example) and hence filter output, will be the negative of the first. We will

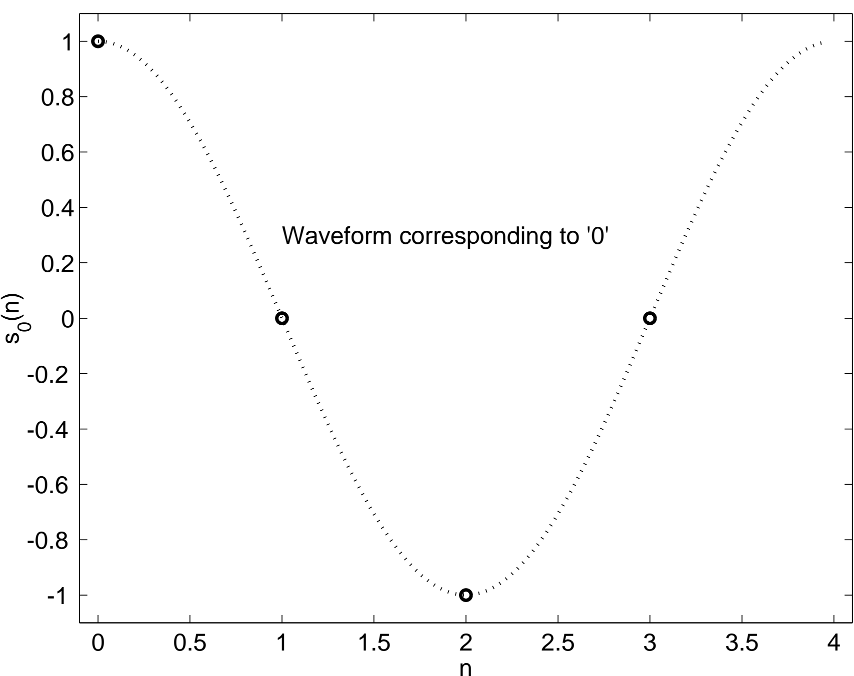


Figure 8.9: s_0 waveform corresponds to bit '0'.

indicate this single matched filter as $\mathbf{c}$. Note that we do not make bit decisions every sample but rather every symbol period. The next few examples illustrate the process of matched filtering.

Example (The BPSK Signal): Assume for the BPSK signals we have

$$\begin{aligned} s_0[n] &= \cos(\omega_c n) \\ s_1[n] &= \cos(\omega_c n + \pi) = -\cos(\omega_c n) \end{aligned} \tag{8.15}$$

where $\omega_c = \pi/2$ (as in the case of $f_s = 9600$Hz and $f_c = 2400$Hz, i.e. four samples per cycle). Here $\omega_c = 2\pi f_c/f_s$ is taken as the discrete-time frequency variable. Signals for $\mathbf{s}_0[n]$ and $\mathbf{s}_1[n]$ are illustrated in Figures 8.9 and 8.10 where

$$\mathbf{s}_i[n] = \left[s_i[n], s_i[n-1], \ldots, s_i[n-N+1]\right]^T . \tag{8.16}$$

In Figure 8.10 we have

$$\mathbf{s}_1[3] = [0, 1, 0, -1]^T . \tag{8.17}$$

■

Example (The Matched Filter): Let us initialize our length $N = 4$ matched filter

$$\mathbf{c} = [c_0, c_1, \ldots, c_{N-1}]^T \tag{8.18}$$

to match against $\mathbf{s}_1[n]$ by time reversing $\mathbf{s}_1[N-1]$ and shifting by $N-1$ samples

$$\mathbf{c} = [0, 1, 0, -1]^T . \tag{8.19}$$

We note that $\mathbf{c} = \mathbf{s}_1[N-1]$ after time reversing and shifting as shown in Figure 8.11. ■

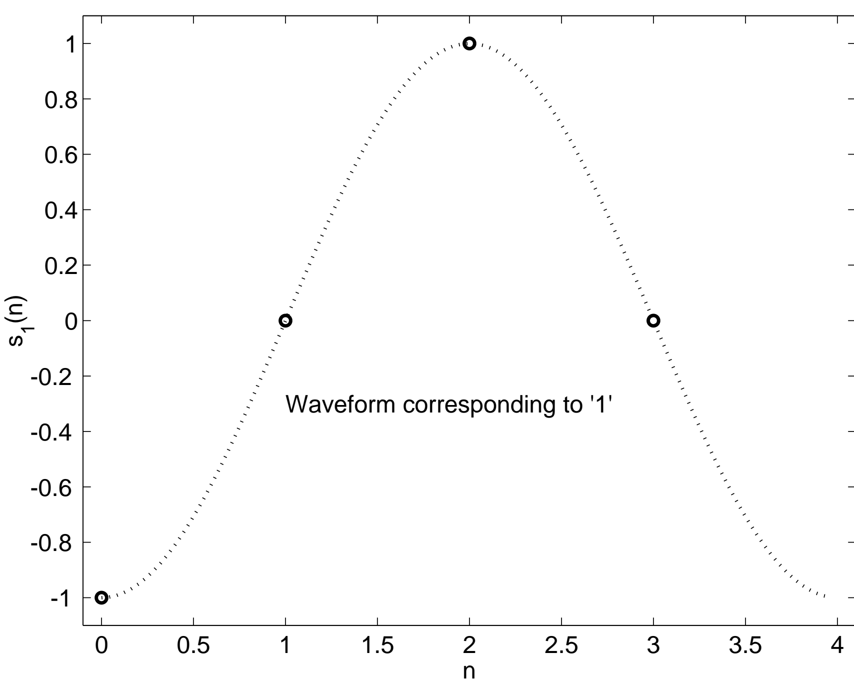


Figure 8.10: s_1 waveform corresponds to bit '1'.

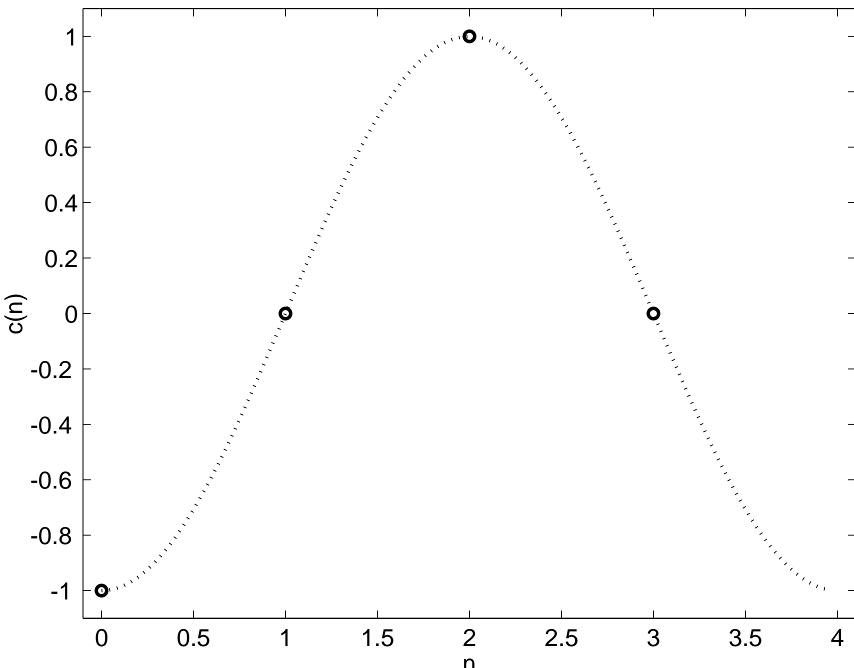


Figure 8.11: Sample matched filter, $\mathbf{c}$ matched against $\mathbf{s}_1[n]$.

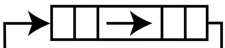


Figure 8.12: Rotating filter coefficients.

Example (The Detection–Part 1): Now suppose we transmit a bit of '1'. Then our received signal at $n = 3$ (in the absence of noise) is given by Figure 8.13 as

$$\begin{aligned} \mathbf{r}[3] &= [s[3], s[2], s[1], s[0]]^T \\ &= [0, 1, 0, -1]^T . \end{aligned} \tag{8.20}$$

The output of the matched filter is given by the convolution

$$\begin{aligned} z[3] &= \mathbf{c}^T \mathbf{r}[3] \\ &= \sum_{k=0}^{N-1} c_k r_k[3] \\ &= 2 \end{aligned} \tag{8.21}$$

where $r_k[3]$ denotes the k-th element of $\mathbf{r}[3]$. Based on the output of the matched filter, we observe a positive match to $\mathbf{s}_1[n]$ and as in (8.13), decide that a '1' was transmitted. ■

Example (The Detection–Part 2): Now suppose we transmit a '0' during the next symbol period. Then our received signal at $n = 7$ (in the absence of noise) is given by Figure 8.13 as

$$\begin{aligned} \mathbf{r}[7] &= [s[7], s[6], s[5], s[4]]^T \\ &= [0, -1, 0, 1]^T . \end{aligned} \tag{8.22}$$

The output of the matched filter is given by

$$\begin{aligned} z[7] &= \mathbf{c}^T \mathbf{r}[7] \\ &= \sum_{k=0}^{N-1} c_k r_k[7] \\ &= -2. \end{aligned} \tag{8.23}$$

Based on the output of the matched filter, we observe a negative match to $\mathbf{s}_1[n]$ and as in (8.13), decide that a '0' was transmitted. ■

Note that in the presence of additive noise, the matched filter will maximize the ratio of $\mathbf{s}_0[n]$ or $\mathbf{s}_1[n]$ to the noise.

8.6.3 Practical Issues

As new samples of the received signal are shifted into $\mathbf{r}[n]$, we must rotate the matched filter coefficients in order to maintain match with the '1' bit as in Figure 8.12.

Example: Let us use the signals from the detection examples to transmit (and receive) a '1' then a '0' as in Figure 8.13. We assume the matched filter, $\mathbf{c}$ has been matched to $\mathbf{s}_1[n]$. The matched filter, received signal vector, and matched filter output at $n = 3$ are given by

$$\mathbf{c}[3] = [0, 1, 0, -1]^T$$

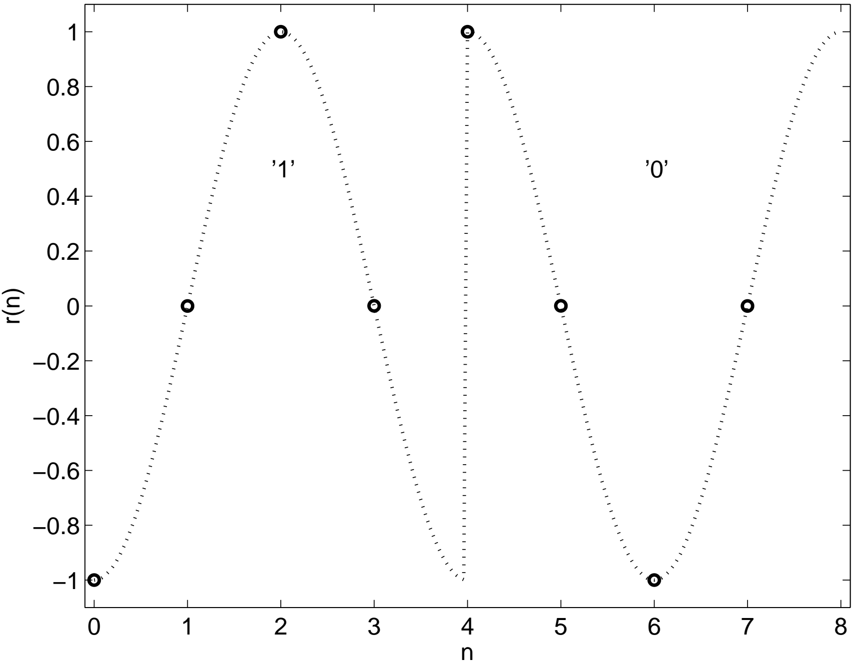


Figure 8.13: Received signal for consecutive '1' and '0'.

$$\begin{aligned} \mathbf{r}[3] &= [0, 1, 0, -1]^T \\ z[3] &= 2. \end{aligned} \tag{8.24}$$

At this point, we decide a '1' was transmitted since we are positively matched to $\mathbf{s}_1[n]$. Now suppose at the transmitter we begin to transmit a '0'. The matched filter, received signal vector, and matched filter output at $n = 4$ and $n = 5$ are given by

$$\begin{aligned} \mathbf{c}[4] &= [-1, 0, 1, 0]^T \\ \mathbf{r}[4] &= [1, 0, 1, 0]^T \\ z[4] &= 0 \end{aligned} \tag{8.25}$$

and

$$\begin{aligned} \mathbf{c}[5] &= [0, -1, 0, 1]^T \\ \mathbf{r}[5] &= [0, 1, 0, 1]^T \\ z[5] &= 0. \end{aligned} \tag{8.26}$$

The small-valued (zero) filter outputs indicate a bit transition is occurring. We continue filtering the received signal and the matched filter, received signal vector, and matched filter output at $n = 6$ and $n = 7$ are given by

$$\begin{aligned} \mathbf{c}[6] &= [1, 0, -1, 0]^T \\ \mathbf{r}[6] &= [-1, 0, 1, 0]^T \\ z[6] &= -2 \end{aligned} \tag{8.27}$$

and

$$\begin{aligned} \mathbf{c}[7] &= [0, 1, 0, -1]^T \\ \mathbf{r}[7] &= [0, -1, 0, 1]^T \\ z[7] &= -2. \end{aligned} \tag{8.28}$$

At this point, we decide a '0' was transmitted since we are negatively matched to $\mathbf{s}_1[n]$. Note that we do not make a decision on every sample period since many samples may represent a single bit. We typically make our decision once we have transitioned completely over to a new bit. ■

Example: Suppose we have $f_s = 9600$Hz, $f_c = 2400$Hz, and a 600bps data rate. Then each bit duration is 1/600s and at $f_s = 9600$Hz we would have 16 samples per bit period. In this case we would need to decide on a bit every 16 samples. Note also that we have 4 samples/cycle and 4 cycles/bit. ■

Obviously symbol decisions occur at a rate equal to the data rate. The example previous to the above illustrates that there is a transition period between symbols. Therefore, we want to base our decisions when we are in the "middle" of the symbol interval rather than at the beginning or end since we may be in transition.

Due to inaccuracies in the clocks of both transmitter and receiver, we expect a slight mismatch to occur between the filter coefficients and reference signal after a period of time. Although we plan on making symbol decisions when we are located in the middle of the symbol interval, this slight mismatch will eventually lead us away from the middle of the interval. The easy way around this is to periodically resynchronize the receiver with the transmitter. Since the filter is matched to the '1' signal it will be convenient to reset the coefficients to match on the samples corresponding to the '1' signal after every stop bit.

8.7 Implementation–Receiver

8.7.1 Algorithm

We assume transmitter emits '1' for 5.0s.

```
INITIALIZE
    Set SYNCHRONIZE_FLAG , STARTUP_FLAG
    STARTUP_COUNTER = SAMPS_BIT, BIT_COUNTER = 11
MAIN (For each sample period)
    If SYNCHRONIZE_FLAG (need to synchronize)
        If STARTUP_FLAG
            STARTUP
        Elseif WAIT_FLAG
            WAIT_FOR_FIRST_START_BIT
        Elseif ADVANCE_FLAG
            ADVANCE_HALFWAY_THROUGH_BIT
        Else (DETERMINE_FLAG)
            UPDATE_VECTORS
            DETERMINE_BIT (Detect first start bit of first frame)
            If (BIT == '1') (synchronization failure—START_BIT should have been '0')
                STOP
            Else (synchronized)
                Clear SYNCHRONIZE_FLAG
            End
        End
    Else (synchronized)
        UPDATE_VECTORS
        Decrement SAMP_COUNTER
        If (SAMP_COUNTER == 0)
            DETERMINE_BIT
        End
        If (BIT_COUNTER == 0)
            PROCESS_FRAME
            BIT_COUNTER = 11
            RESYNCHRONIZE
        End
    End
    Goto MAIN
```

STARTUP (Assume transmitter is sending waveform for BIT = '1')

Shift samples into MF_COEFS and INPUT vectors

Decrement STARTUP_COUNTER

If STARTUP_COUNTER == 0

Clear STARTUP_FLAG, set WAIT_FLAG

End

WAIT_FOR_FIRST_START_BIT

UPDATE_VECTORS

COMPUTE_FILTER_OUTPUT

If $z < -$THRESHOLD (transition detected–hopefully 1st start bit)

Clear WAIT_FLAG, set ADVANCE_FLAG

ADVANCE_COUNTER = SAMPS_BIT/2 $- N + 1$

End

ADVANCE_HALFWAY_THROUGH_BIT_INTERVAL

(By the time $z < -$THRESHOLD, we're already $N-1$ samples into 1st start bit, so SAMPS_BIT/2$-$ $N + 1$ more samples to move halfway into bit interval. Once we're halfway through 1st start bit, making decisions every SAMPS_BITS samples guarantees us to be in the middle of all bits)

UPDATE_VECTORS

Decrement ADVANCE_COUNTER

If ADVANCE_COUNTER == 0

Clear ADVANCE_FLAG, set DETERMINE_FLAG

End

UPDATE_VECTORS

Rotate MF_COEFS around

Shift in new sample into INPUT vector

COMPUTE_FILTER_OUTPUT

z = MF_COEFS^T INPUT (inner product)

DETERMINE_BIT

Clear DETERMINE_FLAG

COMPUTE_FILTER_OUTPUT

If $z >$ THRESHOLD

BIT = '1'

Else (assume $z < -$THRESHOLD)

BIT = '0'

End

SAMP_COUNTER = SAMPS_BIT

Decrement BIT_COUNTER

RESYNCHRONIZE

MF_COEFS = INPUT

PROCESS_FRAME

Write ASCII code to memory (D0–D6 in Figure 8.5)

Chapter 9

Real-Time Frequency-Domain Processing

9.1 Background

In this project, we will provide details on how to transform time-domain signals into the frequency-domain and vice-versa. While this project will not involve any kind of actual processing in the frequency-domain (we just move in and out of the frequency domain), we focus on a step-by-step method of structuring the code so that simple frequency-domain processing can easily be added.

9.2 Discrete Fourier Transform Review

In this section we review the fundamentals of the Discrete Fourier Transform (DFT).

9.2.1 Matrix Form of the DFT

Consider the N-point DFT of a length L signal

$$X(\omega_k) = \sum_{n=0}^{L-1} x[n]e^{-j\omega_k n} \tag{9.1}$$

where

$$\omega_k = 2\pi k/N \tag{9.2}$$

is the kth DFT frequency for $0 \leq k \leq N-1$. Substitution of (9.2) into (9.3) yields

$$X(\omega_k) = \sum_{n=0}^{L-1} x[n]e^{-j2\pi kn/N}. \tag{9.3}$$

For convenience let the Nth root of unity (often called a twiddle factor) be denoted as

$$W_N = e^{-j2\pi/N}. \tag{9.4}$$

Example:
$W_2 = e^{-j2\pi/2} = e^{-j\pi} = -1$
$W_4 = e^{-j2\pi/4} = e^{-j\pi/2} = -j$
$W_8 = e^{-j2\pi/8} = e^{-j\pi/4} = \frac{1-j}{\sqrt{2}}$ ■

Substituting (9.4) into (9.3), the DFT can now be rewritten as (replacing ω_k simply with index k)

$$X[k] = \sum_{n=0}^{L-1} x[n] W_N^{kn}, \ 0 \leq k \leq N-1. \tag{9.5}$$

We can consolidate the above N equations in (9.5) into a matrix equation

$$\begin{bmatrix} X[0] \\ X[1] \\ \vdots \\ X[N-1] \end{bmatrix} = \begin{bmatrix} W_N^{(0)(0)} & W_N^{(0)(1)} & \cdots & W_N^{(0)(L-1)} \\ W_N^{(1)(0]} & W_N^{(1)(1)} & \cdots & W_N^{(1)(L-1)} \\ \vdots & \vdots & \ddots & \vdots \\ W_N^{(N-1)(0)} & W_N^{(N-1)(1)} & \cdots & W_N^{(N-1)(L-1)} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[L-1] \end{bmatrix} \tag{9.6}$$

or

$$\begin{bmatrix} X[0] \\ X[1] \\ \vdots \\ X[N-1] \end{bmatrix} = \begin{bmatrix} W_N^0 & W_N^0 & \cdots & W_N^0 \\ W_N^0 & W_N^1 & \cdots & W_N^{L-1} \\ \vdots & \vdots & \ddots & \vdots \\ W_N^0 & W_N^{N-1} & \cdots & W_N^{(N-1)(L-1)} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[L-1] \end{bmatrix} \tag{9.7}$$

or

$$\mathbf{X} = \mathbf{W}\mathbf{x}. \tag{9.8}$$

The N-point DFT can be thought of as a linear (unitary) transformation (using $\mathbf{W}$, the DFT matrix) of the L-dimensional vector of time samples into an N-dimensional vector of frequency samples. In the frequency domain (space), we use sinusoidal basis functions.

9.3 Fast Fourier Transform

The basic idea in the FFT is to decompose the DFT into smaller transforms, perform these smaller transforms and combine the results, i.e. "divide and conquer." The decomposition exploits symmetries in the DFT matrix.

We first assume that the length of the signal vector, $L = N$ where N is the number of evaluation points in the DFT. We can achieve $L = N$ by zero padding or modulo N wrapping. We therefore have

$$X[k] = \sum_{n=0}^{N-1} x[n]W_N^{kn},\ 0 \le k \le N-1. \tag{9.9}$$

We'll also assume for convenience that N is a power of 2. We usually have enough flexibility to do this.

9.3.1 Decimation in Time FFT

We begin by first partitioning the DFT sum in (9.9) into two sums: the first taken over the even indices and the second taken over the odd indices

$$\begin{aligned} X[k] &= \sum_{n=0}^{N-1} x[n]W_N^{kn} \\ &= \sum_{n=0,\ n \text{ even}}^{N-2} x[n]W_N^{kn} + \sum_{n=0,\ n \text{ odd}}^{N-1} x[n]W_N^{kn} \\ &= \sum_{r=0}^{\frac{N}{2}-1} x[2r]W_N^{k(2r)} + \sum_{r=0}^{\frac{N}{2}-1} x[2r+1]W_N^{k(2r+1)}. \end{aligned} \tag{9.10}$$

Next we note that

$$\begin{aligned} W_N^{k(2r)} &= e^{-j\frac{2\pi}{N}k(2r)} = e^{-j\frac{2\pi}{N/2}kr} = W_{N/2}^{kr} \\ W_N^{k(2r+1)} &= e^{-j\frac{2\pi}{N}k(2r+1)} = e^{-j\frac{2\pi}{N/2}kr}e^{-j2\pi k/N} = W_N^k W_{N/2}^{kr}. \end{aligned} \tag{9.11}$$

Substituting (9.11) into (9.10) we have

$$\begin{aligned} X[k] &= \sum_{r=0}^{\frac{N}{2}-1} x[2r]W_{N/2}^{kr} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} x[2r+1]W_{N/2}^{kr},\ 0 \le k \le N-1 \\ &= G[k] + W_N^k H[k] \end{aligned} \tag{9.12}$$

where $G[k]$, $H[k]$ are the $N/2$-point DFTs of the length $N/2$ sequences $x[2r]$, $x[2r+1]$ respectively

$$G[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r]W_{N/2}^{kr},\ 0 \le k \le \frac{N}{2} - 1 \tag{9.13}$$

$$H[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r+1]W_{N/2}^{kr},\ 0 \le k \le \frac{N}{2} - 1. \tag{9.14}$$

Note that $G[k]$ and $H[k]$ are periodic with period $N/2$, i.e.

$$G[k + N/2] = \sum_{r=0}^{\frac{N}{2}-1} x[2r]W_{N/2}^{(k+N/2)r}$$

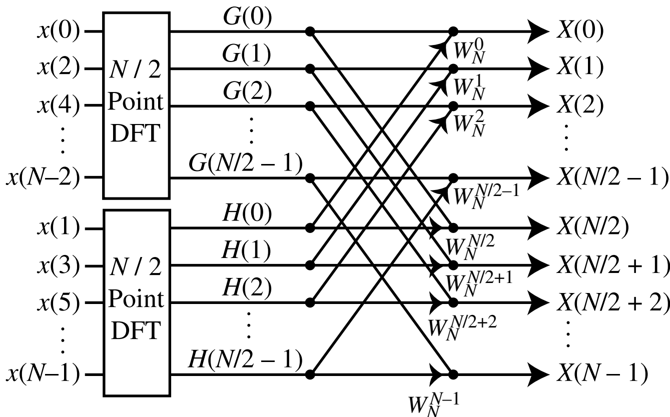


Figure 9.1: N-pt DFT as two N/2-pt DFTs.

$$= \sum_{r=0}^{\frac{N}{2}-1} x[2r] W_{N/2}^{kr}$$
$$= G[k]. \tag{9.15}$$

We can see from (9.12) that the N-point DFT can be computed with a pair of $N/2$-point DFTs as defined in (9.13) and (9.14) and by exploiting the fact the $G[k]$ and $H[k]$ are periodic with a period of $N/2$. The process of DFTs and recombination can be seen graphically in Figure 9.1 We see at this point the computational complexity for an N-point DFT using this approach is

- (2) N / 2-point DFTs each requiring $\left(\frac{N}{2}\right)^2$ complex MACs
- (N) complex MACs for recombination necessary to calculate final $X[k]$s

This yields $\frac{N^2}{2} + N$ complex MACs versus N^2 complex MACs for the conventional DFT calculation in (9.9).

Example: Let $N = 1024$. For a DFT implemented in conventional form we require 1,048,576 complex multiples while that implemented with a divide and conquer approach requires 525,312 complex multiples. The latter approach reduces computation by about 50%. ■

As in this first stage, we can decompose (or decimate) the $N/2$-point DFTs into two $N/4$-point DFTs and another recombination scheme. This can be continued until the problem has been decomposed down to a series of 2-point DFT [a total of $\log_2(N)$ stages] and a bunch of recombination schemes as illustrated in Figure 9.2.

We notice several things in this example and from Figure 9.2

- In order to get the $X[k]$s in order, we must scramble the $x[n]$s. The scrambling scheme is determined as follows. The $x[n]$ required for position m is determined by representing m in

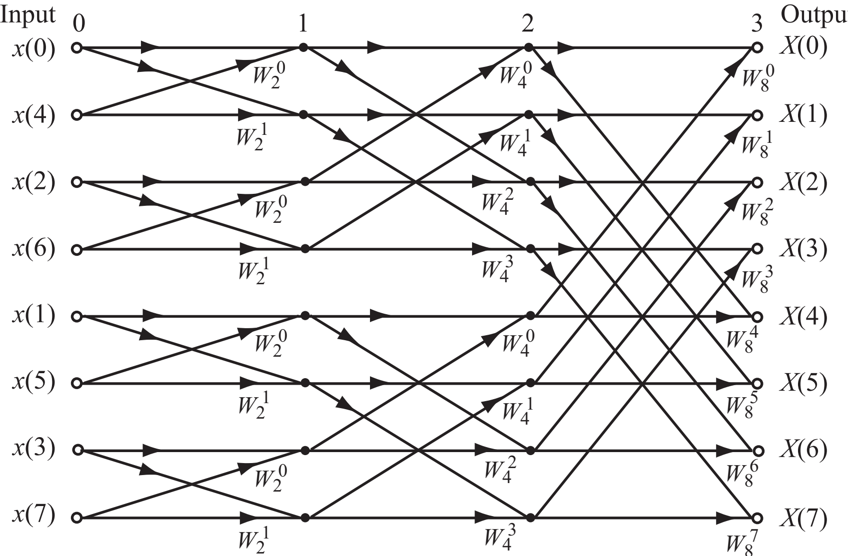


Figure 9.2: FFT Flowgraph.

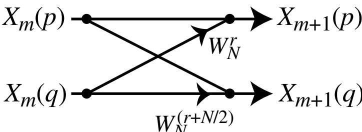


Figure 9.3: FFT Butterfly.

binary form. The bits representing m are reversed and n is obtained by converting these bits back to decimal form. Therefore, $x[n]$ must first be placed in bit-reversed order prior to FFT. On a DSP, bit-reversed addressing modes are typically done in hardware.

- The basic computational block in the FFT is called the "butterfly" illustrated in Figure 9.3 where the I/O equations are given by

$$X_{m+1}[p] = X_m[p] + W_N^r X_m[q] \tag{9.16}$$

$$X_{m+1}[q] = X_m[p] + W_N^{r+N/2} X_m[q]. \tag{9.17}$$

Note that the butterfly outputs, $X_{m+1}[p]$ and $X_{m+1}[q]$ depend only on the input to the butterfly $X_m[p]$ and $X_m[q]$. Therefore memory locations storing $X_m[p]$ and $X_m[q]$ are easily updated with $X_{m+1}[p]$ and $X_{m+1}[q]$. This kind of computation is referred to as an "in-place" calculation. Fast memory reads/writes (or even dual reads/writes in one clock cycle) and MACs are essential to fast butterfly computations and are also optimized in VLSI.

It can be shown that the Decimation-in-Time FFT requires $\frac{N}{2}\log_2(N)$ complex multiplications. Order $N\log(N)$, denoted $\mathcal{O}(N\log N)$, algorithms are often called "fast" hence the name Fast Fourier Transform.

Example: If we compute the DFT of a 1024-point sequence, we require 1,048,576 complex MACs using the conventional DFT and 5120 complex MACs using the FFT. This relates to over a 200-fold reduction in computation! ■

9.4 FFTs on the Motorola DSP5630x

Information regarding the computation of the FFT on the DSP5630x can be found in Appendix C of the *DSP56300 Family Manual* p. C-15—C-16 as well as in Application Note APR4. Codes for various FFT routines are available on the Motorola Web site (Dr. BuB) and the course web page.

9.4.1 Required Hardware Support for the FFT Calculation

- *Multiply accumulate instruction* The basic building block of the DIT FFT is the butterfly computation. Consequently, the architecture and instruction set of a DSP should allow efficient computation of this basic butterfly. Since the butterfly consists of additions and multiplications, a hardware adder/subtractor and multiplier is crucial. Such hardware support is available on the DSP56K with the ADD, SUB, MPY, and MAC instructions.

- *Dual memory read and write in one instruction cycle* Since the butterfly calculation requires complex data, the architecture must easily support complex arithmetic. The I/O to the butterflies are moved between the processors arithmetic unit and memory. Consequently, efficient moves are needed. Since the DSP56K features dual memory spaces, X and Y, the real and imaginary components can be naturally separated into these two spaces. Since the DSP56K can perform parallel reads and writes in one instruction cycle, complex numbers can be easily handled.

- *Bit-reversed addressing mode* In most applications, time and frequency data are used in normal, sequential order even though the standard FFT delivers the output data in bit-reversed order. Thus an efficient method for bit-reversed addressing is needed. The DSP56K provides a bit-reversed addressing mode (hardware).

9.4.2 DIT Butterfly Kernal

The DIT Butterfly kernal is illustrated in Figure 9.4. The DIT butterfly equations are programmed on the DSP56K as

This figure is needed.

Figure 9.4: DIT butterfly kernal.

$$
\begin{aligned}
A_{r'} &= A_r + B_r C_r - B_i C_i && (9.18)\\
A_{i'} &= A_i + B_i C_r + B_r C_i && (9.19)\\
B_{r'} &= A_r - B_r C_r + B_i C_i && (9.20)\\
&= 2A_r - A_{r'} \\
B_{i'} &= A_i - B_i C_r - B_r C_i && (9.21)\\
&= 2A_i - A_{i'}
\end{aligned}
$$

where r represents the real component, i represents the imaginary component, ‘ symbolizes output items, and the twiddle factor, C is defined as

$$
\begin{aligned}
C_N^{-kn} &= e^{-j2\pi kn/N} \\
&= \cos\left(\frac{2\pi kn}{N}\right) - j\sin\left(\frac{2\pi kn}{N}\right) \\
&= C_r - jC_i. && (9.22)
\end{aligned}
$$

The assembly language for the butterfly core is given by (taken from Appendix C of the *DSP56300 Family Manual*)

```
(Preloaded: x1=Br; y0=Ci; a=Ar+BrCr-BiCi; b=Ai)

mac     y0,x1,b   x:(r6)+n6,x0   y:(r1)+,y1  ;b=Ai+BrCi; x0=Cr; y1=Bi
macr    x0,y1,b   a,x:(r5)+      y:(r0),a    ;b=Ai+BrCi+BiCr
                                             ;Br'=Ar-BrCr+BiCi; a=Ai
subl    b,a                                  ;a=Ai-BrCi-BiCr
move    x:(r0),b  b,y:(r4)                   ;b=Ar; Ai'=Ai+BrCi+BiCr
mac     x0,x1,b   x:(r0)+,a      a,y:(r5)    ;b=Ar+BrCr; a=Ar
                                             ;Bi'=Ai-BrCi-BiCr
macr    -y0,y1,b  x:(r1),x1      y:(r6),y0   ;b=Ar+BrCr-BiCi; x1=Br; y0=Ci
subl    b,a       b,x:(r4)+      y:(r0),b    ;a=Ar+BrCr-BiCi
```

The butterfly core uses an amazingly small 7 programs words and executes in 8 instruction cycles.

9.4.3 DIT FFT Outline

In order to combine all the butterflies into a complete program, we first divide the overall algorithm into FFT passes or stages. On each pass, the data is fetched from memory, the butterfly calculations are done in-place, and the results moved back out into memory. It can be shown that there are $\log_2 N$ passes in an N-point FFT.

Within each pass, the butterflies cluster in groups. From one pass to the next, the number of groups doubles, while the number of butterflies per group is halved. Note the twiddle factors are the same for all butterflies within each group, and that the order of the twiddle factors from one group to the next is bit-reversed. The twiddle factors are stored in lookup tables using the supplied `sincos.asm` macro.

All this gives rise to the simple, triple-nested DO loop program, `FFTR2A.ASM`. The outer DO loop steps through the passes, the middle loop goes through all of the groups within a pass, and the inner loop cycles through all of the butterflies inside a group. The resulting program takes 40 words in program memory. The version listed here is a slightly modified version of the original FFTR2A suited for the DSP5630x. Here the modification is a slight rearrangement in instructions from that previously listed in the butterfly core.

```
fft      move    #points/2,n0      ;initialize butterflies per group
         move    #1,n2             ;initialize groups per pass
         move    #points/4,n6      ;initialize C pointer offset
         move    #-1,m0            ;initialize A and B address modifiers
         move    m0,m1             ;for linear addressing
         move    m0,m4
         move    m0,m5
         move    #0,m6             ;initialize C address modifier for
                                   ;reverse carry (bit-reversed) addressing
;
; Perform all FFT passes with triple nested DO loop
;
         do      #@cvi(@log(points)/@log(2)+0.5),_end_pass
         move    #data,r0          ;initialize A input pointer
         move    r0,r4             ;initialize A output pointer
         move    #coef,r6          ;initialize C input pointer
         lua     (r0)+n0,r1        ;initialize B input pointer
         move    n0,n1             ;initialize pointer offsets
         move    n0,n4
         move    n0,n5
         nop
         lua     (r1)-,r5          ;initialize B output pointer

         do      n2,_end_grp
         move    x:(r1),x1  y:(r6),y0        ;lookup -sine and
                                             ; -cosine values
         move    x:(r5),a   y:(r0),b         ;preload data
         move    x:(r6)+n6,x0                ;update C pointer
```

Table 9.1: FFT memory layout

Address	X	Y
brdata	Real	Imag
⋮	⋮	⋮
brdata + points - 1	Real	Imag
nodata	Real	Imag
⋮	⋮	⋮
nodata + points - 1	Real	Imag

```
         do      n0,_end_bfy
         mac     x1,y0,b    y:(r1)+,y1       ;Radix 2 DIT butterfly kernel
         macr    -x0,y1,b   a,x:(r5)+    y:(r0),a
         subl    b,a        x:(r0),b     b,y:(r4)
         mac     -x1,x0,b   x:(r1),x1
         macr    -y1,y0,b   x:(r0)+,a    a,y:(r5)
         subl    b,a        b,x:(r4)+    y:(r0),b
_end_bfy
         move    x:(r0)+n0,x1   y:(r4)+n4,y1
         move    a,x:(r5)+n5    y:(r1)+n1,y1 ;update A and B pointers
_end_grp
         move    n0,b1
         lsr     b   n2,a1                ;divide butterflies per group by two
         lsl     a                        ;multiply groups per pass by two
         move   b1,n0
         move    a1,n2
_end_pass
```

9.4.4 Bit Reversing

Listed below is code to take the bit-reversed data from memory and store it as normally-ordered data in memory. We assume the data is stored in memory as illustrated in Table 9.1.

```
move #brdata,r1      ;pointers to bit-reversed data
move r1,r6
move #nordata,r0     ;pointers to normal order data
move r0,r5
move #0,m1 ;m1 and m6 for bit-reversed addressing
move m1,m6
move #-1,m0          ;m0 and m5 for linear addressing
move m0,m5
move #points/2,n1 ;initialize offset for bit reversing
move n1,n6
```

```
do #points,end_reverse
move    x:(r1)+n1,a     y:(r6)+n6,b
move    a,x:(r0)+       b,y:(r5)+
end_reverse
```

9.4.5 Inverse FFT

The inverse FFT (IFFT) is defined as

$$x[n] = \sum_{k=0}^{N-1} X[k] W_N^{-kn}, \; 0 \le n \le N-1. \tag{9.23}$$

The differences between the FFT (9.9) and the IFFT (9.23) are in the scaling factor $1/N$ and conjugated twiddle factors. We can exploit the similarity in the equations to compute an IFFT using an FFT.

We can compute an IFFT by doing an FFT on $X^*[k]$, conjugating the result, and scaling by $1/N$:

$$\begin{aligned}
\mathbf{x}[k] &= [\mathbf{x}[k]^*]^* \\
&= \left\{ \left[\frac{1}{N} \sum_{k=0}^{N-1} \mathbf{X}[k] W_N^{-kn} \right] \right\}^* \\
&= \left[\frac{1}{N} \sum_{k=0}^{N-1} \mathbf{X}^*[k] W_N^{kn} \right]^* \\
&= \frac{1}{N} \left\{ \text{FFT}\left[X^*[k] \right] \right\}^*
\end{aligned} \tag{9.24}$$

This could be accomplished by flipping the sign bit of $\text{Im}\left[X[k]\right]$ before doing the FFT, then flipping the sign bit of the result of $\text{FFT}[X^*[k]]$. Final scaling by $1/N$ of the result is necessary for the correct answer unless we have already scaled the original input as described in Section 1. The BCHG instruction is useful for this method.

We may alternatively compute the IFFT using an FFT routine as follows

1. swap real and imaginary parts of $X[k]$
2. do the FFT on 1)
3. swap real and imaginary parts of 2)
4. scale by $1/N$

To see why this works we consider the IFFT given by the synthesis equation in (9.23) as

$$\begin{aligned}
x[n] &= \frac{1}{N} \sum_{k=0}^{N-1} X[k] C_N^{kn} \\
&= \frac{1}{N} \sum_{k=0}^{N-1} \left[X_r[k] + jX_i[k] \right] \left[C_r + jC_i \right]
\end{aligned}$$

$$= \frac{1}{N}\sum_{k=0}^{N-1}\left[X_r[k]C_r - X_i[k]C_i\right] + j\left[X_i[k]C_r + X_r[k]C_i\right] \quad (9.25)$$

where C is defined in (9.22). Now suppose we apply the FFT (analysis equation) to $X[k]$ with its real and imaginary parts swapped

$$\begin{aligned} x[n] &= \sum_{k=0}^{N-1} X_{SWAPPED}[k]C_N^{-kn} \\ &= \sum_{k=0}^{N-1}\left[X_i[k] + jX_r[k]\right]\left[C_r - jC_i\right] \\ &= \sum_{k=0}^{N-1}\left[X_i[k]C_r + X_r[k]C_i\right] + j\left[X_r[k]C_r - X_i[k]C_i\right]. \end{aligned} \quad (9.26)$$

Clearly, (9.26) is the same as (9.25) with two exceptions: 1) the real and imaginary parts are swapped and 2) the $1/N$ scale factor needs to still be done.

9.4.6 Scaling Issues

Due to the fractional operation of the DSP56K, we must examine the potential for overflow when computing the FFT. First we examine the analysis equation (9.9) to determine the upper bound on any $|X[k]|$ under the assumption that every $x[n]$ is upper bounded in magnitude by unity.

Clearly, $\text{Re}\,[X[k]]$ or $\text{Im}\,[X[k]] = N$ is achieved when $x[n] = W_N^{-kn}$ or $x[n] = jW_N^{-kn}$. Since N is always less than or equal to one, we must scale $x[n]$ to prevent overflow. If we scale $x[n]$ by $1/N$, we guarantee that $\max\{\text{Re}\,[X[k]]\}$ or $\max\{\text{Im}\,[X[k]]\} = 1$. Therefore, prior to any FFT on the DSP56K you should either scale all $x[n]$s by $1/N$ (using an ASR instruction since N is a power of 2) or put the DSP into an automatic scaling mode where you scale at each stage. Remember that if you scale the input signal, the output values are to be interpreted as being $1/N$ of the correct value. By scaling we lose 10/24 bits of precision for a 1024 point FFT. This implies a lower dynamic range.

9.4.7 Available FFT Macros for the DSP56K

FFT macros for the Motorola DSP56K are widely available. These macros provide various tradeoffs between memory and speed and some provide automatic bit-reversing (normal order). These macros are listed in Table 9.2

9.4.8 Comments

On an 80MHz Motorola 5630x DSP (80mips), a 1024-point DFT computed using an FFT requires 12 program words and 8N+9 clock cycles or about 0.102ms.

Table 9.2: DSP56K FFT Routines

ROUTINE	COMMENT
`fftr2a.asm`	The FFTR2A macro requires about 40 words of program memory for any size FFT from 2-32768 points. This is the most compact FFT routine available for the DSP56000.
`fftr2b.asm`	The FFTR2B macro is slightly faster than the FFTR2A library macro.
`fftr2c.asm`	The FFTR2C macro is faster than the FFTR2A and FFTR2B library macros.
`fftr2cc.asm`	The FFTR2CC macro is faster than the FFTR2A and FFTR2B library macros.
`fftr2cn.asm`	The FFTR2CN macro is identical to the FFTR2C macro except for the storage of the output data. Using the reverse-carry address modifier and a separate output data buffer, the output data is stored in normal order.
`fftr2d.asm`	The FFTR2D macro requires about 40 words of program memory for any size FFT from 2-32768 points. The main advantage of this FFT routine is its use of a full-cycle sinewave table for coefficients. This sinewave table can often be shared with other functions in the system and is available in the DSP56001 Y Data ROM.
`fftr2e.asm`	This FFT macro performs a 1024 point complex FFT on external data using the Radix 2, Decimation in Time, Cooley-Tukey FFT algorithm.
`fftr2en.asm`	This FFT macro performs a 1024 point complex FFT on external data using the Radix 2, Decimation in Time, Cooley-Tukey FFT algorithm. The input and output data are assumed to be in normal (time-sequential) order.

9.5 Sample DSP56K Codes for Computing the FFT

In this section we examine several sample codes for computing the FFT. The first set of codes computes the FFT on values contained in an input file and is therefore not real-time, while the other set computes the FFT on values acquired through sampling.

9.5.1 Example 1: Simple code for an FFT using FFTR2A (non-real-time)

For the first FFT code, we use the macro `FFTR2A.ASM` which takes an FFT of a normally-ordered input and returns a bit-reversed output. For illustration purposes, the FFT is taken on the points contained in the file `SIGNAL.DAT`. These points are stored in memory using the `DC` assembler directive.

```
include 'fftr2a'        ;fft with I/O in normal,bit-reversed order

points   equ     1024       ;length of signal

org l:$0000
data1    dsm     points     ;Re,Im parts of signal; bit-reversed FFT(signal)
ndata1   dsm     points     ;normally ordered FFT(signal)
coef     dsm     points     ;twiddle factors

org x:data1
include 'SIGNAL.DAT'

include 'sincos'        ;macro to...
sincos  points,coef     ;...build twiddle factor tables

org p:$0
jmp begin
org p:$100
begin

;------
; SCALE
;------
move     #data1,r0
move     #points-1,m0
do #points,scale
move     l:(r0),ab
asr #@cvi(@log(points)/@log(2)),a,a     ;scale real(signal) by 1/points
asr #@cvi(@log(points)/@log(2)),b,b     ;scale imag(signal) by 1/points
move     ab,l:(r0)+
scale

;---
;FFT
;---
fftr2a points,data1,coef ;data1 = bit-reversed FFT(data1)
```

```
;----------------------------------------------------------------------
; Convert bit-reverse FFT(signal) into normal order (p. 294 El-Sharkway)
;----------------------------------------------------------------------
move #data1,r1 ;pointers to bit-reversed data
move r1,r6
move #ndata1,r0 ;pointers to normally-ordered data
move r0,r5
move #0,m1 ;m1 and m6 = bit reverse addressing
move m1,m6
move #-1,m0 ;m0 and m5 = linear addressing
move m0,m5
move #points/2,n1 ;n1 and n1 for bit-reversed addressing
move n1,n6

do #points,end_reverse
move x:(r1)+n1,a y:(r6)+n6,b
move a,x:(r0)+ b,y:(r5)+
end_reverse
```

9.5.2 Example 2: Simple code for an FFT using FFTR2CN (non-real-time)

For the second FFT code, we use the macro `FFTR2CN.ASM` which takes an FFT of a normally-ordered input and returns a normally-ordered output.

```
include 'fftr2cn';fft with input,output in normal,normal (respectively) order

points equ 1024 ;length of signal

org l:$0000
data1     dsm     points     ;real,imag parts of signal
odata1    dsm     points     ;real,imag parts of FFT(signal)
coef      dsm     points     ;twiddle factors

    org x:data1             ;put real(signal) in x memory
    include 'RESIGNAL.DAT'

    org y:data1             ;put imag(signal) in x memory
    include 'IMSIGNAL.DAT'

include 'sincos'        ;macro to...
sincos  points,coef     ;...build twiddle factor tables

org p:$0
jmp begin
org p:$40
begin

;-------
; SCALE - scale real(signal), imag(signal) to avoid overflow problems.
```

```
;-------
move     #data1,r0
move     #points-1,m0
do #points,scale
move     l:(r0),ab
asr #@cvi(@log(points)/@log(2)),a,a     ;scale real(signal) by 1/points
asr #@cvi(@log(points)/@log(2)),b,b     ;scale imag(signal) by 1/points
move     ab,l:(r0)+
scale

;----
;FFT - do the FFT on data1 and store results in odata1.  Note that the
;      results are scaled by 1/points
;----
fftr2cn  points,data1,odata1,coef     ;odata1 = FFT(data1)
```

9.6 Code for the IFFT

For the IFFT code, we use the macro `FFTR2CN.ASM` and the real, imaginary swapping technique described in the previous section. We assume odata1 has already been scaled by $1/N = 1/points$.

```
     org l:*
data2     dsm     points     ;imag,real (swapped) parts of FFT(signal)
odata2    dsm     points     ;real,imag parts of IFFT[FFT(signal)]

;------
; SWAP - in order to IFFT, swap real and imag parts
;------
move     #odata1,r0
move     #points-1,m0
move     #data2,r1
move     #points-1,m1
do     #points,swap1
move     l:(r0)+,ab     ;real->a, imag->b
move     ba,l:(r1)+     ;b->real, a->imag
swap1

;-----
;IFFT - do IFFT on data2 and store results in odata2
;-----
fftr2cn  points,data2,odata2,coef

;------
; SWAP - finally, swap real and imag parts to get correct IFFT
;------
move     #odata2,r0
move     #points-1,m0
do     #points,swap2
move     l:(r0),ab     ;imag->a, real->b
move     ba,l:(r0)+    ;a->real, b->imag
```

```
swap2

jmp *     ;stop
```

9.7 Continuous, Real-Time Spectral Processing

When computing FFTs on the EVM, we may use `pass.asm` provided the FFT can be computed in a single sample period. Otherwise, care must be taken to buffer any samples which are acquired during the FFT computation. In the latter case, we rely on two buffers: the first on which the FFT is computed and the second used to buffer samples from the codec while computation takes place. Once the FFT is completed, we begin FFT computation on the filled buffer and use the other for buffering new, incoming samples. We continue this “ping-pong buffering” to achieve FFTs on contiguous sample blocks. If we perform spectral processing and an IFFT, we require a third buffer to store output samples which are then unloaded to the codec.

9.7.1 Single Sample Period FFT

If the FFT can be computed in a single sample period, a sample code segment would be:

```
    .
    .
    .
    clr a
move     #points,a0     ;initialize buffer counter

move     #left_input_buffer,r0     ;initialize I/O buffer pointers
move     #points-1,m0
move     #left_output_buffer,r1
move     #points-1,m1

loop_1
jset     #2,x:SSISR,*     ;wait for frame sync to pass
jclr     #2,x:SSISR,*     ;wait for frame sync

move     x:RX_BUFF_BASE,x0   ;get left input
move     x0,x:(r0)+          ;buffer input
move     x:(r1)+,x1          ;get left output from buffer
move     x1,TX_RX_BUFF_BASE  ;transmit output

dec a
jne loop_1     ;input buffer full and output buffer empty?

jsr swap_buffers
jsr scale_samples
jsr fft
jsr process_spectrum
jsr ifft
```

```
clr a     ;reset buffer counter
move     #points,a0
jmp loop_1
    .
    .
    .
```

9.7.2 Multiple Sample Period FFT

If the FFT cannot be computed in a single sample period, perhaps due to a high sample rate, long FFT, and/or complex spectral processing, we must move the buffering of I/O out of the main event loop and into the interrupt service routine (ISR). Of course, we now assume that all processing in the main even loop executes within N sample periods where N is the buffer size. Sample code segments are as follows.

We first modify the `SSI_RX_ISR`:

```
ssi_rx_isr
move r0,x:(r6)+ ;Save r0 to the stack.
move m0,x:(r6)+ ;Save m0 to the stack.
move x0,x:(r6)+ ;Save x0 to the stack.
move a2,x:(r6)+ ;Save a
move a1,x:(r6)+
move a0,x:(r6)+

move #3,m0 ;Modulo 4 buffer.
move x:RX_PTR,r0 ;Load the pointer to the rx buffer.
jclr #3,x:SSISR,next_rx ;If not fr. syc, jump to receive data.
move #RX_BUFF_BASE,r0 ;If frame sync, reset base pointer.
bset #BUFFER_SAMPLE,x:FLAG ;buffer incoming sample
next_rx
movep x:SSIDR,x:(r0)+ ;Read out received data to buffer.
move r0,x:RX_PTR ;Update rx buffer pointer.
jclr #BUFFER_SAMPLE,x:FLAG,SKIP_BUFFER_SAMPLE

move x:RX_BUFF_BASE,x0 ;get incoming sample
move x0,x:(r0)+ ;buffer incoming sample
bclr #BUFFER_SAMPLE,x:FLAG
clr a
move x:BUFFER_COUNTER,a0
dec a
jne SKIP_BUFFER_SAMPLE
bset #BUFFER_FULL,x:FLAG
SKIP_BUFFER_SAMPLE
move a0,x:BUFFER_COUNTER

move x:-(r6),a0 ;Restore a.
move x:-(r6),a1
move x:-(r6),a2
```

```
move x:-(r6),x0 ;Restore x0.
move x:-(r6),m0 ;Restore m0.
move x:-(r6),r0 ;Restore r0.
rti
```

We next modify the SSI_TX_ISR:

```
ssi_tx_isr
move r0,x:(r6)+ ;Save r0 to the stack.
move m0,x:(r6)+ ;Save m0 to the stack.
move x0,x:(r6)+ ;Save x0 to the stack.

jset #XMIT_CTRL_WORDS,x:FLAG,MOVE_IT_OUT ;let control words pass freely
bset #XMIT_CTRL_WORDS,x:FLAG
move x:(r1)+,x0 ;Get output sample from buffer
move x0,x:TX_BUFF_BASE ;Move output sample for TX
MOVE_IT_OUT
move #3,m0 ; Modulus 4 buffer.
move x:TX_PTR,r0 ; Load the pointer to the tx buffer.
jclr #2,x:SSISR,next_tx ;If not frame sync, jump to transmit data.
move #TX_BUFF_BASE+1,r0 ; If frame sync, reset pointer.
bclr #XMIT_CTRL_WORDS,x:FLAG
next_tx
movep x:(r0)+,x:SSIDR ;SSI transfer data register.
move r0,x:TX_PTR ; Update tx buffer pointer.

move x:-(r6),x0 ;Restore x0.
move x:-(r6),m0 ;Restore m0.
move x:-(r6),r0 ;Restore r0.
rti
```

Finally, the main event loop for this setup would be something like the following:

```
.
.
.
        clr a
        move #points,a0 ;initialize buffer counter

        move #input_buffer,r0           ;initialize I/O/P buffer pointers
        move #points-1,m0
        move #output_buffer,r1
        move #points-1,m1
        move #processing_buffer,r2
        move #points-1,m2

loop_1
        jclr #BUFFER_FULL,x:FLAG,loop_1 ;buzz until BUFFER_FULL FLAG set
        move #points,x0                 ;reset BUFFER_COUNTER
        move x0,x:BUFFER_COUNTER

        jsr P_buff_to_O_buff            ;copy processing_buffer to output_buffer
```

```
        jsr I_buff_to_P_buff            ;copy input_buffer to processing_buffer
        jsr scale_samples               ;process processing_buffer
        jsr fft
        jsr process_spectrum
        jsr ifft                        ;processing_buffer now has next output block

        jmp loop_1
.
.
.
```

Appendix A

Motorola Pass Pack

A.1 PASS.ASM

```
1   ;*************************************************************************
2   ;QS_ST.ASM : WILL PASS AUDIO STRAIGHT THROUGH
3   ;*************************************************************************
4         nolist
5         include 'ioequ.asm'
6         include 'intequ.asm'
7         include 'ada_equ.asm'
8         include 'vectors.asm'
9         list
10
11  ;****************************************************************************
12
13  ;---Buffer for talking to the CS4215
14
15         org    x:0
16  RX_BUFF_BASE    equ     *
17  RX_data_1_2     ds      1       ;data time slot 1/2 for RX ISR
18  RX_data_3_4     ds      1       ;data time slot 3/4 for RX ISR
19  RX_data_5_6     ds      1       ;data time slot 5/6 for RX ISR
20  RX_data_7_8     ds      1       ;data time slot 7/8 for RX ISR
21
22  TX_BUFF_BASE    equ     *
23  TX_data_1_2     ds      1       ;data time slot 1/2 for TX ISR
24  TX_data_3_4     ds      1       ;data time slot 3/4 for TX ISR
25  TX_data_5_6     ds      1       ;data time slot 5/6 for TX ISR
26  TX_data_7_8     ds      1       ;data time slot 7/8 for TX ISR
27
28  RX_PTR          ds      1       ; Pointer for rx buffer
29  TX_PTR          ds      1       ; Pointer for tx buffer
30
31  TONE_OUTPUT     EQU     HEADPHONE_EN+LINEOUT_EN+(4*LEFT_ATTN)+(4*RIGHT_ATTN)
32  TONE_INPUT      EQU     MIC_IN_SELECT+(15*MONITOR_ATTN)
33  CTRL_WD_12      equ     NO_PREAMP+HI_PASS_FILT+SAMP_RATE_48+STEREO+DATA_16   ;CLB=0
```

```
34  CTRL_WD_34      equ     IMMED_3STATE+XTAL1_SELECT+BITS_64+CODEC_MASTER
35  CTRL_WD_56      equ     $000000
36  CTRL_WD_78      equ     $000000
37
38
39
40          org     p:$100
41  START
42  main
43          movep   #$040003,x:M_PCTL  ; set PLL for MPY of 4X
44          movep   #$012421,x:M_BCR   ; set up one ext. wait state for all AAR areas
45          ori     #3,mr             ;mask interrupts
46          movec   #0,sp             ;clear hardware stack pointer
47          move    #0,omr            ;operating mode 0
48          move    #$40,r6            ; initialise stack pointer
49          move    #-1,m6             ; linear addressing
50          jsr     ada_init           ; initialize codec
51
52
53  loop_1
54
55          jset    #2,x:M_SSISR0,*    ;wait for frame sync to pass
56          jclr    #2,x:M_SSISR0,*    ;wait for frame sync
57
58          move    x:RX_BUFF_BASE,a        ;receive left
59          move    x:RX_BUFF_BASE+1,b      ;receive right
60          jsr     process_stereo
61          move    a,x:TX_BUFF_BASE        ;transmit left
62          move    b,x:TX_BUFF_BASE+1      ;transmit right
63
64          move    #TONE_OUTPUT,y0 ;set up control words
65          move    y0,x:TX_BUFF_BASE+2
66          move    #TONE_INPUT,y0
67          move    y0,x:TX_BUFF_BASE+3
68
69          jmp     loop_1
70
71  process_stereo
72          nop
73          nop
74          nop
75          rts
76
77          include 'ada_init.asm'
78  echo
79          end
```

A.2 ADA_INIT.ASM

```
1           page    132,60
2   ;****************************************************************************
3   ;       ADA_INIT.ASM    Ver.2.0
4   ;       Example program to initialize the CS4215
5   ;
6   ;       Copyright (c) MOTOROLA 1995, 1996
7   ;                     Semiconductor Products Sector
8   ;                     Digital Signal Processing Division
9   ;
10  ;       History:
11  ;               14 June 1996:  RLR/LJD - ver.1.0
12  ;***************************************************************************
13
14
15  ;**************************************************************************
16  ;
17  ;      portc usage:
18  ;   bit8: SSI TX (from DSP to Codec)
19  ;   bit7:
20  ;   bit6:
21  ;   bit5:
22  ;   bit4: codec reset (from DSP to Codec)
23  ;   bit3:
24  ;        bit2: data/control bar
25  ;             0=control
26  ;             1=data
27  ;
28  ;***************************************************************************
29  ;*****          initialize the CS4215 codec                    *****
30  ;***************************************************************************
31  ;
32  ;
33  ;  PROGRAM OUTLINE:
34  ;
35  ;1 program fsync and sclk == output
36  ;2 write pc0 = 0 (control mode)
37  ;3 send 64 bit frame x times, with dcb bit = 0, keep doing until read back as 0
38  ;4 send 64 bit frame x times, with dcb bit = 1, keep doing until read back as 1
39  ;5 re-program fsync and sclk == input
40  ;6 write pc0 = 1 (data mode)
41  ;7 receive/send data (echo slots 1,2,3,4; slots 5,6,7,8 == constants)
42  ;
43  ;****************************************************************************
44  ;
45  ;      initialize ssi -- fsync and sclk ==> outputs
46  ;
47          org     p:
48  ada_init
49          movep   #$0000,x:M_PCRC         ; turn off ESSI0 port (for now)
50          movep   #$103807,x:M_CRA0       ; 40MHz/16 = 2.5MHz SCLK, WL=16 bits, 4W/F
```

```
51          movep   #$ff313C,x:M_CRB0       ; RIE,TIE,RLIE,TLIE,RE,TE,sc2/sck outputs
52          movep   #$0003,x:M_PRRC         ; setup pd0 and pd1 as gpio output
53          movep   #$0,x:M_PDRC            ; send out a 0 on DC~ and RST_CODEC~
54
55             ;----reset delay for codec ----
56          do      #1000,_delay_loop
57          rep     #2000                   ; 100 us delay (assuming 40MHz VCO)
58          nop
59  _delay_loop
60
61          bset    #0,x:M_PDRC             ; sends out a 1 on pd0 (rst_codec=1)
62          movep   #$000C,x:M_IPRP         ; set interrupt priority level for ESSI0 to 3
63          andi    #$FC,mr                 ; enable interrupts
64
65  ;****************************************************************************
66  ; The following data sets up the CS4215 control mode data:
67  ;    (CTS = Control Time Slot, U/LN = upper/lower Nibble)
68  ;
69  ;       +------ CTS1-UN:   0      0      1     MLB      0 0 0 0
70  ;       |+----- CTS1-LN:  OLB    CLB     X      X       0 0 0 0
71  ;       ||+---- CTS2-UN:  HPF     X     DFR2   DFR1     0 0 1 0
72  ;       |||+--- CTS2-LN:  DFR0   ST     DF1    DF0      1 1 0 0
73  ; x0 = $002Cxx
74  ;
75  ;       +------ CTS3-UN:  ITS    MCK2   MCK1   MCK0     1 0 0 0
76  ;       |+----- CTS3-LN:  BSEL1  BSEL0  XCLK   XEN      1 0 0 0
77  ;       ||+---- CTS4-UN:  TEST   TEST   TEST   TEST     (TEST MUST BE 0)
78  ;       |||+--- CTS4-LN:  TEST   TEST   ENL    DAD      0 0 0 0
79  ; x0 = $8800xx
80  ;****************************************************************************
81
82  ;--- set up buffer with control mode data
83          move    #CTRL_WD_12,x0
84          move            x0,x:TX_BUFF_BASE
85          move    #CTRL_WD_34,x0
86          move            x0,x:TX_BUFF_BASE+1
87          move    #CTRL_WD_56,x0
88          move            x0,x:TX_BUFF_BASE+2
89          move    #CTRL_WD_78,x0
90          move            x0,x:TX_BUFF_BASE+3
91
92          movep   #$003C,x:M_PCRC         ;turn on ESSI0 except for sc0 and sc2
93
94  ;
95  ; CLB == 0
96  ;
97          jclr    #3,x:M_SSISR0,*         ; wait until rx frame bit==1
98          jset    #3,x:M_SSISR0,*         ; wait until rx frame bit==0
99          jclr    #3,x:M_SSISR0,*         ; wait until rx frame bit==1
100         jset    #18,x:RX_BUFF_BASE,*    ; loop until CLB set
101
```

```
102   ;
103   ; CLB == 1
104   ;
105           bset    #18,x:TX_BUFF_BASE      ;set CLB
106           do      #4,_init_loopB
107           jclr    #2,x:M_SSISR0,*         ; wait until tx frame bit==1
108           jset    #2,x:M_SSISR0,*         ; wait until tx frame bit==0
109   _init_loopB
110           movep   #$0000,x:M_PCRC         ; disable ESSI0
111
112   ;****************************************************************************
113   ;    now CLB should be 1 -- re-program fsync and sclk direction (i/p) -- also,
114   ;  circular buffer pointers for echoing data r0=current, r1=old data to send
115   ;  1 frame later
116   ;
117           movep   #$103807,x:M_CRA0       ; 40MHz/16 = 2.5MHz SCLK, WL=16 bits, 4W/F
118           movep   #$FF310C,x:M_CRB0       ; sckd and fsync (sc02) as inputs
119           movep   #$0003,x:M_PDRC         ; D/C~ pin = 1 ==> data mode
120           movep   #$003C,x:M_PCRC         ; turn on ESSI0 except for sc0 and sc2
121           rts
122
123   ;***************************************************************************
124   ;       SSI0_ISR.ASM    Ver.2.0
125   ;       Example program to handle interrupts through
126   ;       the 56303 SSI0 to move audio through the CS4215
127   ;
128   ;       Copyright (c) MOTOROLA 1995, 1996
129   ;                     Semiconductor Products Sector
130   ;                     Digital Signal Processing Division
131   ;
132   ;       upon entry:
133   ;               R6 must be the stack pointer
134   ;       corrupts:
135   ;               R6
136   ;
137   ;       History:
138   ;               14 June 1996: RLR/LJD - ver 1.0
139   ;*****************************************************************************
140
141
142   ;----the actual interrupt service routines (ISRs)  follow:
143
144   ;************************ SSI TRANSMIT ISR *********************************
145   ssi_txe_isr
146           bclr    #4,x:M_SSISR0           ; Read SSISR to clear exception flag
147                                           ; explicitly clears underrun flag
148   ssi_tx_isr
149           move    r0,x:(r6)+              ; Save r0 to the stack.
150           move    m0,x:(r6)+              ; Save m0 to the stack.
151           move    #3,m0                   ; Modulus 4 buffer.
152           move    x:TX_PTR,r0             ; Load the pointer to the tx buffer.
```

```
153         nop
154         movep   x:(r0)+,x:M_TX00        ; SSI transfer data register.
155         move    r0,x:TX_PTR             ; Update tx buffer pointer.
156         move    x:-(r6),m0              ; Restore m0.
157         move    x:-(r6),r0              ; Restore r0.
158         rti
159
160 ;********************* SSI TRANSMIT LAST SLOT ISR *************************
161 ssi_txls_isr
162         move    r0,x:(r6)+              ; Save r0 to the stack.
163         move    #TX_BUFF_BASE,r0        ; Reset pointer.
164         move    r0,x:TX_PTR             ; Reset tx buffer pointer just in
165                                         ; case it was corrupted.
166         move    x:-(r6),r0              ; Restore r0.
167         rti
168
169 ;************************** SSI receive ISR ********************************
170 ssi_rxe_isr
171         bclr    #5,x:M_SSISR0           ; Read SSISR to clear exception flag
172                                         ; explicitly clears overrun flag
173 ssi_rx_isr
174         move    r0,x:(r6)+              ; Save r0 to the stack.
175         move    m0,x:(r6)+              ; Save m0 to the stack.
176         move    #3,m0                   ; Modulo 4 buffer.
177         move    x:RX_PTR,r0             ; Load the pointer to the rx buffer.
178         nop
179         movep   x:M_RX0,x:(r0)+         ; Read out received data to buffer.
180         move    r0,x:RX_PTR             ; Update rx buffer pointer.
181         move    x:-(r6),m0              ; Restore m0.
182         move    x:-(r6),r0              ; Restore r0.
183         rti
184
185 ;********************** SSI receive last slot ISR *************************
186 ssi_rxls_isr
187         move    r0,x:(r6)+              ; Save r0 to the stack.
188         move    #RX_BUFF_BASE,r0        ; Reset rx buffer pointer just in
189                                         ; case it was corrupted.
190         move    r0,x:RX_PTR             ; Update rx buffer pointer.
191         move    x:-(r6),r0              ; Restore r0.
192         rti
```

A.3 ADA_EQU.ASM

```
1          page    132,60
2   ;***************************************************************************
3   ;   ADA_EQU.ASM
4   ;   Initialization constants to facilitate initialization of the CS4215
5   ;
6   ;   Copyright (c) MOTOROLA 1996
7   ;            Semiconductor Products Sector
8   ;            Digital Signal Processing Division
9   ;
10  ;***************************************************************************
11  ;
12  NO_PREAMP       equ     $100000
13  LO_OUT_DRV      equ     $080000
14  HI_PASS_FILT    equ     $008000
15  SAMP_RATE_9     equ     $003800         ; 9.6 kHz sample rate
16  SAMP_RATE_48    equ     $003000         ; 48  kHz sample rate
17  SAMP_RATE_32    equ     $001800         ; 32  kHz sample rate
18  SAMP_RATE_27    equ     $001000
19  SAMP_RATE_16    equ     $000800
20  SAMP_RATE_8     equ     $000000
21  STEREO          equ     $000400
22  DATA_8LIN       equ     $200300
23  DATA_8A         equ     $200200
24  DATA_8U         equ     $200100
25  DATA_16         equ     $200000
26  IMMED_3STATE    equ     $800000
27  XTAL1_SELECT    equ     $100000         ; 24.576 MHz
28  XTAL2_SELECT    equ     $200000         ; 16.9344 MHz
29  BITS_64         equ     $000000
30  BITS_128        equ     $040000
31  BITS_256        equ     $080000
32  CODEC_MASTER    equ     $020000
33  CODEC_TX_OFF    equ     $010000
34
35  CTRL_WD_12      equ     NO_PREAMP+HI_PASS_FILT+SAMP_RATE_48+STEREO+DATA_16   ;CLB=0
36  CTRL_WD_34      equ     IMMED_3STATE+XTAL1_SELECT+BITS_64+CODEC_MASTER
37  CTRL_WD_56      equ     $000000
38  CTRL_WD_78      equ     $000000
39
40  HEADPHONE_EN    equ     $800000
41  LINEOUT_EN      equ     $400000
42  SPEAKER_EN      equ     $004000
43  MIC_IN_SELECT   equ     $100000
44  LEFT_ATTN       equ     $010000 ;63*LEFT_ATTN   = -94.5 dB, 1.5 dB steps
45  RIGHT_ATTN      equ     $000100 ;63*RIGHT_ATTN  = -94.5 dB, 1.5 dB steps
46  LEFT_GAIN       equ     $010000 ;15*LEFT_GAIN    = 22.5 dB, 1.5 dB steps
47  RIGHT_GAIN      equ     $000100 ;15*RIGHT_GAIN   = 22.5 dB, 1.5 dB steps
48  MONITOR_ATTN    equ     $001000 ;15*MONITOR_ATTN = mute,    6   dB steps
49  ;OUTPUT_SET     equ     HEADPHONE_EN+LINEOUT_EN+(LEFT_ATTN*4)
50  ;INPUT_SET      equ     MIC_IN_SELECT+(15*MONITOR_ATTN)+(RIGHT_ATTN*4)
```

A.4 INTEQU.ASM

```
1   ;********************************************************************************
2   ;
3   ;       EQUATES for ONYXE 56302 interrupts
4   ;
5   ;       Last update: June 11 1995
6   ;
7   ;********************************************************************************
8
9           page    132,55,0,0,0
10          opt     mex
11  intequ  ident   1,0
12
13          if      @DEF(I_VEC)
14          ;leave user definition as is.
15          else
16  I_VEC   equ     $0
17          endif
18
19  ;------------------------------------------------------------------------
20  ;Non-Maskable interrupts
21  ;------------------------------------------------------------------------
22  I_RESET equ     I_VEC+$00       ;Hardware RESET
23  I_STACK equ     I_VEC+$02       ;Stack Error
24  I_ILL   equ     I_VEC+$04       ;Illegal Instruction
25  I_DBG   equ     I_VEC+$06       ;Debug Request
26  I_TRAP  equ     I_VEC+$08       ;Trap
27  I_NMI   equ     I_VEC+$0A       ;Non Maskable Interrupt
28
29  ;------------------------------------------------------------------------
30  ;Interrupt Request Pins
31  ;------------------------------------------------------------------------
32  I_IRQA  equ     I_VEC+$10       ;IRQA
33  I_IRQB  equ     I_VEC+$12       ;IRQB
34  I_IRQC  equ     I_VEC+$14       ;IRQC
35  I_IRQD  equ     I_VEC+$16       ;IRQD
36
37  ;------------------------------------------------------------------------
38  ;DMA Interrupts
39  ;------------------------------------------------------------------------
40  I_DMA0  equ     I_VEC+$18       ;DMA Channel 0
41  I_DMA1  equ     I_VEC+$1A       ;DMA Channel 1
42  I_DMA2  equ     I_VEC+$1C       ;DMA Channel 2
43  I_DMA3  equ     I_VEC+$1E       ;DMA Channel 3
44  I_DMA4  equ     I_VEC+$20       ;DMA Channel 4
45  I_DMA5  equ     I_VEC+$22       ;DMA Channel 5
46
47  ;------------------------------------------------------------------------
48  ;Timer Interrupts
49  ;------------------------------------------------------------------------
50  I_TIM0C equ     I_VEC+$24       ;TIMER 0 compare
```

```
51  I_TIM0OF        equu    I_VEC+$26       ;TIMER 0 overflow
52  I_TIM1C         equ     I_VEC+$28       ;TIMER 1 compare
53  I_TIM1OF        equ     I_VEC+$2A       ;TIMER 1 overflow
54  I_TIM2C         equ     I_VEC+$2C       ;TIMER 2 compare
55  I_TIM2OF        equ     I_VEC+$2E       ;TIMER 2 overflow
56
57  ;------------------------------------------------------------------------
58  ;ESSI Interrupts
59  ;------------------------------------------------------------------------
60  I_SI0RD         equ     I_VEC+$30       ;ESSI0 Receive Data
61  I_SI0RDE        equ     I_VEC+$32       ;ESSI0 Receive Data With Exception Status
62  I_SI0RLS        equ     I_VEC+$34       ;ESSI0 Receive last slot
63  I_SI0TD         equ     I_VEC+$36       ;ESSI0 Transmit data
64  I_SI0TDE        equ     I_VEC+$38       ;ESSI0 Transmit Data With Exception Status
65  I_SI0TLS        equ     I_VEC+$3A       ;ESSI0 Transmit last slot
66  I_SI1RD         equ     I_VEC+$40       ;ESSI1 Receive Data
67  I_SI1RDE        equ     I_VEC+$42       ;ESSI1 Receive Data With Exception Status
68  I_SI1RLS        equ     I_VEC+$44       ;ESSI1 Receive last slot
69  I_SI1TD         equ     I_VEC+$46       ;ESSI1 Transmit data
70  I_SI1TDE        equ     I_VEC+$48       ;ESSI1 Transmit Data With Exception Status
71  I_SI1TLS        equ     I_VEC+$4A       ;ESSI1 Transmit last slot
72
73  ;------------------------------------------------------------------------
74  ;SCI Interrupts
75  ;------------------------------------------------------------------------
76  I_SCIRD         equ     I_VEC+$50       ;SCI Receive Data
77  I_SCIRDE        equ     I_VEC+$52       ;SCI Receive Data With Exception Status
78  I_SCITD         equ     I_VEC+$54       ;SCI Transmit Data
79  I_SCIIL         equ     I_VEC+$56       ;SCI Idle Line
80  I_SCITM         equ     I_VEC+$58       ;SCI Timer
81
82  ;------------------------------------------------------------------------
83  ;HOST Interrupts
84  ;------------------------------------------------------------------------
85  I_HRDF          equ     I_VEC+$60       ;Host Receive Data Full
86  I_HTDE          equ     I_VEC+$62       ;Host Transmit Data Empty
87  I_HC            equ     I_VEC+$64       ;Default Host Command
88
89  ;------------------------------------------------------------------------
90  ;INTERRUPT ENDING ADDRESS
91  ;------------------------------------------------------------------------
92  I_INTEND        equ     I_VEC+$FF       ;last address of interrupt vector space
93          LIST
```

A.5 IOEQU.ASM

```
1   ;*****************************************************************************
2   ;
3   ;       equates for ONYXE 56302 I/O registers and ports
4   ;
5   ;       Last update: June 11 1995
6   ;
7   ;*****************************************************************************
8
9           page    132,55,0,0,0
10          opt     mex
11
12  ioequ   ident   1,0
13
14  ;------------------------------------------------------------------------
15  ;
16  ;       equates for I/O Port Programming
17  ;
18  ;------------------------------------------------------------------------
19
20  ;       Register Addresses
21
22  M_HDR   equ     $FFFFC9 ;PS- Host port GPIO data Register
23  M_HDDR  equ     $FFFFC8 ;PS- Host port GPIO direction Register
24  M_PCRC  equ     $FFFFBF ;Port C Control Register
25  M_PRRC  equ     $FFFFBE ;Port C Direction Register
26  M_PDRC  equ     $FFFFBD ;Port C GPIO Data Register
27  M_PCRD  equ     $FFFFAF ;Port D Control register
28  M_PRRD  equ     $FFFFAE ;Port D Direction Data Register
29  M_PDRD  equ     $FFFFAD ;Port D GPIO Data Register
30  M_PCRE  equ     $FFFF9F ;Port E Control register
31  M_PRRE  equ     $FFFF9E ;Port E Direction Register
32  M_PDRE  equ     $FFFF9D ;Port E Data Register
33  M_OGDB  equ     $FFFFFC ;OnCE GDB Register
34
35
36  ;------------------------------------------------------------------------
37  ;
38  ;       equates for Host Interface
39  ;
40  ;------------------------------------------------------------------------
41
42  ;       Register Addresses
43
44  M_HCR   equ     $FFFFC2 ;Host Control Register
45  M_HSR   equ     $FFFFC3 ;Host Status Rgister
46  M_HPCR  equ     $FFFFC4 ;Host Polarity Control Register
47  M_HBAR  equ     $FFFFC5 ;Host Base Address Register
48  M_HRX   equ     $FFFFC6 ;Host Receive Register
49  M_HTX   equ     $FFFFC7 ;Host Transmit Register
50
```

```
 51  ;       HCR bits definition
 52  M_HRIE  equ     $0      ;Host Receive interrupts Enable
 53  M_HTIE  equ     $1      ;Host Transmit Interrupt Enable
 54  M_HCIE  equ     $2      ;Host Command Interrupt Enable
 55  M_HF2   equ     $3      ;Host Flag 2
 56  M_HF3   equ     $4      ;Host Flag 3
 57
 58  ;       HSR bits definition
 59  M_HRDF  equ     $0      ;Host Receive Data Full
 60  M_HTDE  equ     $1      ;Host Receive Data Emptiy
 61  M_HCP   equ     $2      ;Host Command Pending
 62  M_HF0   equ     $3      ;Host Flag 0
 63  M_HF1   equ     $4      ;Host Flag 1
 64
 65
 66  ;       HPCR bits definition
 67  M_HGEN  equ     $0      ;Host Port GPIO Enable
 68  M_HA8EN equ     $1      ;Host Address 8 Enable
 69  M_HA9EN equ     $2      ;Host Address 9 Enable
 70  M_HCSEN equ     $3      ;Host Chip Select Enable
 71  M_HREN  equ     $4      ;Host Request Enable
 72  M_HAEN  equ     $5      ;Host Acknowledge Enable
 73  M_HEN   equ     $6      ;Host Enable
 74  M_HOD   equ     $8      ;Host Request Open Drain mode
 75  M_HDSP  equ     $9      ;Host Data Strobe Polarity
 76  M_HASP  equ     $A      ;Host Address Strobe Polarity
 77  M_HMUX  equ     $B      ;Host Multiplexed bus select
 78  M_HD_HS equ     $C      ;Host Double/Single Strobe select
 79  M_HCSP  equ     $D      ;Host Chip Select Polarity
 80  M_HRP   equ     $E      ;Host Request PolarityPolarity
 81  M_HAP   equ     $F      ;Host Acknowledge Polarity
 82
 83
 84  ;-----------------------------------------------------------------------
 85  ;
 86  ;       equates for Serial Communications Interface (SCI)
 87  ;
 88  ;-----------------------------------------------------------------------
 89
 90  ;       Register Addresses
 91
 92  M_STXH  equ     $FFFF97 ;SCI Transmit Data Register (high)
 93  M_STXM  equ     $FFFF96 ;SCI Transmit Data Register (middle)
 94  M_STXL  equ     $FFFF95 ;SCI Transmit Data Register (low)
 95  M_SRXH  equ     $FFFF9A ;SCI Receive Data Register (high)
 96  M_SRXM  equ     $FFFF99 ;SCI Receive Data Register (middle)
 97  M_SRXL  equ     $FFFF98 ;SCI Receive Data Register (low)
 98  M_STXA  equ     $FFFF94 ;SCI Transmit Address Register
 99  M_SCR   equ     $FFFF9C ;SCI Control Register
100  M_SSR   equ     $FFFF93 ;SCI Status Register
101  M_SCCR  equ     $FFFF9B ;SCI Clock Control Register
```

```
102
103   ;       SCI Control Register Bit Flags
104
105   M_WDS   equ     $7      ;Word Select Mask (WDS0-WDS3)
106   M_WDS0  equ     0       ;Word Select 0
107   M_WDS1  equ     1       ;Word Select 1
108   M_WDS2  equ     2       ;Word Select 2
109   M_SSFTD equ     3       ;SCI Shift Direction
110   M_SBK   equ     4       ;Send Break
111   M_WAKE  equ     5       ;Wakeup Mode Select
112   M_RWU   equ     6       ;Receiver Wakeup Enable
113   M_WOMS  equ     7       ;Wired-OR Mode Select
114   M_SCRE  equ     8       ;SCI Receiver Enable
115   M_SCTE  equ     9       ;SCI Transmitter Enable
116   M_ILIE  equ     10      ;Idle Line Interrupt Enable
117   M_SCRIE equ     11      ;SCI Receive Interrupt Enable
118   M_SCTIE equ     12      ;SCI Transmit Interrupt Enable
119   M_TMIE  equ     13      ;Timer Interrupt Enable
120   M_TIR   equ     14      ;Timer Interrupt Rate
121   M_SCKP  equ     15      ;SCI Clock Polarity
122   M_REIE  equ     16      ;SCI Error Interrupt Enable (REIE)
123
124   ;       SCI Status Register Bit Flags
125
126   M_TRNE  equ     0       ;Transmitter Empty
127   M_TDRE  equ     1       ;Transmit Data Register Empty
128   M_RDRF  equ     2       ;Receive Data Register Full
129   M_IDLE  equ     3       ;Idle Line Flag
130   M_OR    equ     4       ;Overrun Error Flag
131   M_PE    equ     5       ;Parity Error
132   M_FE    equ     6       ;Framing Error Flag
133   M_R8    equ     7       ;Received Bit 8 (R8) Address
134
135   ;       SCI Clock Control Register
136
137   M_CD    equ     $FFF    ;Clock Divider Mask (CD0-CD11)
138   M_COD   equ     12      ;Clock Out Divider
139   M_SCP   equ     13      ;Clock Prescaler
140   M_RCM   equ     14      ;Receive Clock Mode Source Bit
141   M_TCM   equ     15      ;Transmit Clock Source Bit
142
143   ;-----------------------------------------------------------------------
144   ;
145   ;       equates for Synchronous Serial Interface (SSI)
146   ;
147   ;-----------------------------------------------------------------------
148
149   ;
150   ;       Register Addresses Of SSI0
151   M_TX0   equ     $FFFFBC ;SSI0 Transmit Data Register 0
152   M_TX00  equ     $FFFFBC ;SSI0 Transmit Data Register 0
```

```
153  M_TX01  equ     $FFFFBB ;SSI0 Transmit Data Register 1
154  M_TX02  equ     $FFFFBA ;SSI0 Transmit Data Register 2
155  M_TSR0  equ     $FFFFB9 ;SSI0 Time Slot Register
156  M_RX0   equ     $FFFFB8 ;SSI0 Receive Data Register
157  M_SSISR0        equ     $FFFFB7 ;SSI0 Status Register
158  M_CRB0  equ     $FFFFB6 ;SSI0 Control Register B
159  M_CRA0  equ     $FFFFB5 ;SSI0 Control Register A
160  M_TSMA0 equ     $FFFFB4 ;SSI0 Transmit Slot Mask Register A
161  M_TSMB0 equ     $FFFFB3 ;SSI0 Transmit Slot Mask Register B
162  M_RSMA0 equ     $FFFFB2 ;SSI0 Receive Slot Mask Register A
163  M_RSMB0 equ     $FFFFB1 ;SSI0 Receive Slot Mask Register B
164
165  ;       Register Addresses Of SSI1
166  M_TX1   equ     $FFFFAC ;SSI1 Transmit Data Register 0
167  M_TX10  equ     $FFFFAC ;SSI1 Transmit Data Register 0
168  M_TX11  equ     $FFFFAB ;SSI1 Transmit Data Register 1
169  M_TX12  equ     $FFFFAA ;SSI1 Transmit Data Register 2
170  M_TSR1  equ     $FFFFA9 ;SSI1 Time Slot Register
171  M_RX1   equ     $FFFFA8 ;SSI1 Receive Data Register
172  M_SSISR1        equ     $FFFFA7 ;SSI1 Status Register
173  M_CRB1  equ     $FFFFA6 ;SSI1 Control Register B
174  M_CRA1  equ     $FFFFA5 ;SSI1 Control Register A
175  M_TSMA1 equ     $FFFFA4 ;SSI1 Transmit Slot Mask Register A
176  M_TSMB1 equ     $FFFFA3 ;SSI1 Transmit Slot Mask Register B
177  M_RSMA1 equ     $FFFFA2 ;SSI1 Receive Slot Mask Register A
178  M_RSMB1 equ     $FFFFA1 ;SSI1 Receive Slot Mask Register B
179
180  ;       SSI Control Register A Bit Flags
181
182  M_PM    equ     $FF     ;Prescale Modulus Select Mask (PM0-PM7)
183  M_PSR   equ     11      ;Prescaler Range
184  M_DC    equ     $1F000  ;Frame Rate Divider Control Mask (DC0-DC7)
185  M_ALC   equ     18      ;Alignment Control (ALC)
186  M_WL    equ     $380000 ;Word Length Control Mask (WL0-WL7)
187  M_SSC1  equ     22      ;Select SC1 as TR #0 drive enable (SSC1)
188
189  ;       SSI Control Register B Bit Flags
190
191  M_OF    equ     $3      ;Serial Output Flag Mask
192  M_OF0   equ     0       ;Serial Output Flag 0
193  M_OF1   equ     1       ;Serial Output Flag 1
194  M_SCD   equ     $1C     ;Serial Control Direction Mask
195  M_SCD0  equ     2       ;Serial Control 0 Direction
196  M_SCD1  equ     3       ;Serial Control 1 Direction
197  M_SCD2  equ     4       ;Serial Control 2 Direction
198  M_SCKD  equ     5       ;Clock Source Direction
199  M_SHFD  equ     6       ;Shift Direction
200  M_FSL   equ     $180    ;Frame Sync Length Mask (FSL0-FSL1)
201  M_FSL0  equ     7       ;Frame Sync Length 0
202  M_FSL1  equ     8       ;Frame Sync Length 1
203  M_FSR   equ     9       ;Frame Sync Relative Timing
```

```
204   M_FSP   equ     10      ;Frame Sync Polarity
205   M_CKP   equ     11      ;Clock Polarity
206   M_SYN   equ     12      ;Sync/Async Control
207   M_MOD   equ     13      ;SSI Mode Select
208   ;NOTE: REMOVE THIS NEXT LINE FOR COMPATIBILITY
209   M_SSTE  equ     16      ;SSI Transmit #0 Enable
210   ;M_SSTE equ     $1C000  ;SSI Transmit enable Mask
211   M_SSTE2 equ     14      ;SSI Transmit #2 Enable
212   M_SSTE1 equ     15      ;SSI Transmit #1 Enable
213   M_SSTE0 equ     16      ;SSI Transmit #0 Enable
214   M_SSRE  equ     17      ;SSI Receive Enable
215   M_SSTIE equ     18      ;SSI Transmit Interrupt Enable
216   M_SSRIE equ     19      ;SSI Receive Interrupt Enable
217   M_STLIE equ     20      ;SSI Transmit Last Slot Interrupt Enable
218   M_SRLIE equ     21      ;SSI Receive Last Slot Interrupt Enable
219   M_STEIE equ     22      ;SSI Transmit Error Interrupt Enable
220   M_SREIE equ     23      ;SSI Receive Error Interrupt Enable
221
222   ;       SSI Status Register Bit Flags
223
224   M_IF    equ     $3      ;Serial Input Flag Mask
225   M_IF0   equ     0       ;Serial Input Flag 0
226   M_IF1   equ     1       ;Serial Input Flag 1
227   M_TFS   equ     2       ;Transmit Frame Sync Flag
228   M_RFS   equ     3       ;Receive Frame Sync Flag
229   M_TUE   equ     4       ;Transmitter Underrun Error FLag
230   M_ROE   equ     5       ;Receiver Overrun Error Flag
231   M_TDE   equ     6       ;Transmit Data Register Empty
232   M_RDF   equ     7       ;Receive Data Register Full
233
234   ;       SSI Transmit Slot Mask Register A
235
236   M_SSTSA equ     $FFFF   ;SSI Transmit Slot Bits Mask A (TS0-TS15)
237
238   ;       SSI Transmit Slot Mask Register B
239
240   M_SSTSB equ     $FFFF   ;SSI Transmit Slot Bits Mask B (TS16-TS31)
241
242   ;       SSI Receive Slot Mask Register A
243
244   M_SSRSA equ     $FFFF   ;SSI Receive Slot Bits Mask A (RS0-RS15)
245
246   ;       SSI Receive Slot Mask Register B
247
248   M_SSRSB equ     $FFFF   ;SSI Receive Slot Bits Mask B (RS16-RS31)
249
250
251
252   ;----------------------------------------------------------------------
253   ;
254   ;       equates for Exception Processing
```

```
255   ;
256   ;------------------------------------------------------------------------
257
258
259   ;       Register Addresses
260
261   M_IPRC  equ     $FFFFFF ;Interrupt Priority Register Core
262   M_IPRP  equ     $FFFFFE ;Interrupt Priority Register Peripheral
263
264   ;       Interrupt Priority Register Core (IPRC)
265
266   M_IAL   equ     $7      ;IRQA Mode Mask
267   M_IAL0  equ     0       ;IRQA Mode Interrupt Priority Level (low)
268   M_IAL1  equ     1       ;IRQA Mode Interrupt Priority Level (high)
269   M_IAL2  equ     2       ;IRQA Mode Trigger Mode
270   M_IBL   equ     $38     ;IRQB Mode Mask
271   M_IBL0  equ     3       ;IRQB Mode Interrupt Priority Level (low)
272   M_IBL1  equ     4       ;IRQB Mode Interrupt Priority Level (high)
273   M_IBL2  equ     5       ;IRQB Mode Trigger Mode
274   M_ICL   equ     $1C0    ;IRQC Mode Mask
275   M_ICL0  equ     6       ;IRQC Mode Interrupt Priority Level (low)
276   M_ICL1  equ     7       ;IRQC Mode Interrupt Priority Level (high)
277   M_ICL2  equ     8       ;IRQC Mode Trigger Mode
278   M_IDL   equ     $E00    ;IRQD Mode Mask
279   M_IDL0  equ     9       ;IRQD Mode Interrupt Priority Level (low)
280   M_IDL1  equ     10      ;IRQD Mode Interrupt Priority Level (high)
281   M_IDL2  equ     11      ;IRQD Mode Trigger Mode
282   M_D0L   equ     $3000   ;DMA0 Interrupt priority Level Mask
283   M_D0L0  equ     12      ;DMA0 Interrupt Priority Level (low)
284   M_D0L1  equ     13      ;DMA0 Interrupt Priority Level (high)
285   M_D1L   equ     $C000   ;DMA1 Interrupt Priority Level Mask
286   M_D1L0  equ     14      ;DMA1 Interrupt Priority Level (low)
287   M_D1L1  equ     15      ;DMA1 Interrupt Priority Level (high)
288   M_D2L   equ     $30000  ;DMA2 Interrupt priority Level Mask
289   M_D2L0  equ     16      ;DMA2 Interrupt Priority Level (low)
290   M_D2L1  equ     17      ;DMA2 Interrupt Priority Level (high)
291   M_D3L   equ     $C0000  ;DMA3 Interrupt Priority Level Mask
292   M_D3L0  equ     18      ;DMA3 Interrupt Priority Level (low)
293   M_D3L1  equ     19      ;DMA3 Interrupt Priority Level (high)
294   M_D4L   equ     $300000 ;DMA4 Interrupt priority Level Mask
295   M_D4L0  equ     20      ;DMA4 Interrupt Priority Level (low)
296   M_D4L1  equ     21      ;DMA4 Interrupt Priority Level (high)
297   M_D5L   equ     $C00000 ;DMA5 Interrupt priority Level Mask
298   M_D5L0  equ     22      ;DMA5 Interrupt Priority Level (low)
299   M_D5L1  equ     23      ;DMA5 Interrupt Priority Level (high)
300
301
302   ;       Interrupt Priority Register Peripheral (IPRP)
303
304   M_HPL   equ     $3      ;Host Interrupt Priority Level Mask
305   M_HPL0  equ     0       ;Host Interrupt Priority Level (low)
```

```
306  M_HPL1  equ     1       ;Host Interrupt Priority Level (high)
307  M_S0L   equ     $C      ;SSI0 Interrupt Priority Level Mask
308  M_S0L0  equ     2       ;SSI0 Interrupt Priority Level (low)
309  M_S0L1  equ     3       ;SSI0 Interrupt Priority Level (high)
310  M_S1L   equ     $30     ;SSI1 Interrupt Priority Level Mask
311  M_S1L0  equ     4       ;SSI1 Interrupt Priority Level (low)
312  M_S1L1  equ     5       ;SSI1 Interrupt Priority Level (high)
313  M_SCL   equ     $C0     ;SCI Interrupt Priority Level Mask
314  M_SCL0  equ     6       ;SCI Interrupt Priority Level (low)
315  M_SCL1  equ     7       ;SCI Interrupt Priority Level (high)
316  M_T0L   equ     $300    ;TIMER Interrupt Priority Level Mask
317  M_T0L0  equ     8       ;TIMER Interrupt Priority Level (low)
318  M_T0L1  equ     9       ;TIMER Interrupt Priority Level (high)
319
320
321  ;------------------------------------------------------------------------
322  ;
323  ;       equates for TIMER
324  ;
325  ;------------------------------------------------------------------------
326
327  ;       Register Addresses Of TIMER0
328
329  M_TCSR0 equ     $FFFF8F ;TIMER0 Control/Status Register
330  M_TLR0  equ     $FFFF8E ;TIMER0 Load Reg
331  M_TCPR0 equ     $FFFF8D ;TIMER0 Compare Register
332  M_TCR0  equ     $FFFF8C ;TIMER0 Count Register
333
334  ;       Register Addresses Of TIMER1
335
336  M_TCSR1 equ     $FFFF8B ;TIMER1 Control/Status Register
337  M_TLR1  equ     $FFFF8A ;TIMER1 Load Reg
338  M_TCPR1 equ     $FFFF89 ;TIMER1 Compare Register
339  M_TCR1  equ     $FFFF88 ;TIMER1 Count Register
340
341
342  ;       Register Addresses Of TIMER2
343
344  M_TCSR2 equ     $FFFF87 ;TIMER2 Control/Status Register
345  M_TLR2  equ     $FFFF86 ;TIMER2 Load Reg
346  M_TCPR2 equ     $FFFF85 ;TIMER2 Compare Register
347  M_TCR2  equ     $FFFF84 ;TIMER2 Count Register
348  M_TPLR  equ     $FFFF83 ;TIMER Prescaler Load Register
349  M_TPCR  equ     $FFFF82 ;TIMER Prescalar Count Register
350
351
352  ;       Timer Control/Status Register Bit Flags
353
354  M_TE    equ     0       ;Timer Enable
355  M_TOIE  equ     1       ;Timer Overflow Interrupt Enable
356  M_TCIE  equ     2       ;Timer Compare Interrupt Enable
```

```
357  M_TC    equ     $F0     ;Timer Control Mask (TC0-TC3)
358  M_INV   equ     8       ;Inverter Bit
359  M_TRM   equ     9       ;Timer Restart Mode
360  M_DIR   equ     11      ;Direction Bit
361  M_DI    equ     12      ;Data Input
362  M_DO    equ     13      ;Data Output
363  M_PCE   equ     15      ;Prescaled Clock Enable
364  M_TOF   equ     20      ;Timer Overflow Flag
365  M_TCF   equ     21      ;Timer Compare Flag
366
367  ;       Timer Prescaler Register Bit Flags
368
369  M_PS    equ     $600000 ;Prescaler Source Mask
370  M_PS0   equ     21
371  M_PS1   equ     22
372
373  ;       Timer Control Bits
374  M_TC0   equ     4       ;Timer Control 0
375  M_TC1   equ     5       ;Timer Control 1
376  M_TC2   equ     6       ;Timer Control 2
377  M_TC3   equ     7       ;Timer Control 3
378
379
380  ;------------------------------------------------------------------------
381  ;
382  ;       equates for Direct Memory Access (DMA)
383  ;
384  ;------------------------------------------------------------------------
385
386  ;       Register Addresses Of DMA
387  M_DSTR  equ     $FFFFF4 ;DMA Status Register
388  M_DOR0  equ     $FFFFF3 ;DMA Offset Register 0
389  M_DOR1  equ     $FFFFF2 ;DMA Offset Register 1
390  M_DOR2  equ     $FFFFF1 ;DMA Offset Register 2
391  M_DOR3  equ     $FFFFF0 ;DMA Offset Register 3
392
393
394  ;       Register Addresses Of DMA0
395
396  M_DSR0  equ     $FFFFEF ;DMA0 Source Address Register
397  M_DDR0  equ     $FFFFEE ;DMA0 Destination Address Register
398  M_DCO0  equ     $FFFFED ;DMA0 Counter
399  M_DCR0  equ     $FFFFEC ;DMA0 Control Register
400
401  ;       Register Addresses Of DMA1
402
403  M_DSR1  equ     $FFFFEB ;DMA1 Source Address Register
404  M_DDR1  equ     $FFFFEA ;DMA1 Destination Address Register
405  M_DCO1  equ     $FFFFE9 ;DMA1 Counter
406  M_DCR1  equ     $FFFFE8 ;DMA1 Control Register
407
```

```
408   ;       Register Addresses Of DMA2
409
410   M_DSR2  equ     $FFFFE7 ;DMA2 Source Address Register
411   M_DDR2  equ     $FFFFE6 ;DMA2 Destination Address Register
412   M_DCO2  equ     $FFFFE5 ;DMA2 Counter
413   M_DCR2  equ     $FFFFE4 ;DMA2 Control Register
414
415   ;       Register Addresses Of DMA4
416
417   M_DSR3  equ     $FFFFE3 ;DMA3 Source Address Register
418   M_DDR3  equ     $FFFFE2 ;DMA3 Destination Address Register
419   M_DCO3  equ     $FFFFE1 ;DMA3 Counter
420   M_DCR3  equ     $FFFFE0 ;DMA3 Control Register
421
422   ;       Register Addresses Of DMA4
423
424
425   M_DSR4  equ     $FFFFDF ;DMA4 Source Address Register
426   M_DDR4  equ     $FFFFDE ;DMA4 Destination Address Register
427   M_DCO4  equ     $FFFFDD ;DMA4 Counter
428   M_DCR4  equ     $FFFFDC ;DMA4 Control Register
429
430   ;       Register Addresses Of DMA5
431
432   M_DSR5  equ     $FFFFDB ;DMA5 Source Address Register
433   M_DDR5  equ     $FFFFDA ;DMA5 Destination Address Register
434   M_DCO5  equ     $FFFFD9 ;DMA5 Counter
435   M_DCR5  equ     $FFFFD8 ;DMA5 Control Register
436
437   ;       DMA Control Register
438
439   M_DSS   equ     $3      ;DMA Source Space Mask (DSS0-Dss1)
440   M_DSS0  equ     0       ;DMA Source Memory space 0
441   M_DSS1  equ     1       ;DMA Source Memory space 1
442   M_DDS   equ     $C      ;DMA Destination Space Mask (DDS-DDS1)
443   M_DDS0  equ     2       ;DMA Destination Memory Space 0
444   M_DDS1  equ     3       ;DMA Destination Memory Space 1
445   M_DAM   equ     $3f0    ;DMA Address Mode Mask (DAM5-DAM0)
446   M_DAM0  equ     4       ;DMA Address Mode 0
447   M_DAM1  equ     5       ;DMA Address Mode 1
448   M_DAM2  equ     6       ;DMA Address Mode 2
449   M_DAM3  equ     7       ;DMA Address Mode 3
450   M_DAM4  equ     8       ;DMA Address Mode 4
451   M_DAM5  equ     9       ;DMA Address Mode 5
452   M_D3D   equ     10      ;DMA Three Dimensional Mode
453   M_DRS   equ     $F800   ;DMA Request Source Mask (DRS0-DRS4)
454   M_DCON  equ     16      ;DMA Continuous Mode
455   M_DPR   equ     $60000  ;DMA Channel Priority
456   M_DPR0  equ     17      ;DMA Channel Priority Level (low)
457   M_DPR1  equ     18      ;DMA Channel Priority Level (high)
458   M_DTM   equ     $380000 ;DMA Transfer Mode Mask (DTM2-DTM0)
```

```
459  M_DTM0  equ     19      ;DMA Transfer Mode 0
460  M_DTM1  equ     20      ;DMA Transfer Mode 1
461  M_DTM2  equ     21      ;DMA Transfer Mode 2
462  M_DIE   equ     22      ;DMA Interrupt Enable bit
463  M_DE    equ     23      ;DMA Channel Enable bit
464
465  ;       DMA Status Register
466
467  M_DTD   equ     $3F     ;Channel Transfer Done Status MASK (DTD0-DTD5)
468  M_DTD0  equ     0       ;DMA Channel Transfer Done Status 0
469  M_DTD1  equ     1       ;DMA Channel Transfer Done Status 1
470  M_DTD2  equ     2       ;DMA Channel Transfer Done Status 2
471  M_DTD3  equ     3       ;DMA Channel Transfer Done Status 3
472  M_DTD4  equ     4       ;DMA Channel Transfer Done Status 4
473  M_DTD5  equ     5       ;DMA Channel Transfer Done Status 5
474  M_DACT  equ     8       ;DMA Active State
475  M_DCH   equ     $E00    ;DMA Active Channel Mask (DCH0-DCH2)
476  M_DCH0  equ     9       ;DMA Active Channel 0
477  M_DCH1  equ     10      ;DMA Active Channel 1
478  M_DCH2  equ     11      ;DMA Active Channel 2
479
480
481  ;------------------------------------------------------------------------
482  ;
483  ;       equates for Phase Locked Loop (PLL)
484  ;
485  ;------------------------------------------------------------------------
486
487  ;       Register Addresses Of PLL
488
489  M_PCTL  equ     $FFFFFD ;PLL Control Register
490
491  ;       PLL Control Register
492
493  M_MF    equ     $FFF    ;Multiplication Factor Bits Mask (MF0-MF11)
494  M_DF    equ     $7000   ;Division Factor Bits Mask (DF0-DF2)
495  M_XTLR  equ     15      ;XTAL Range select bit
496  M_XTLD  equ     16      ;XTAL Disable Bit
497  M_PSTP  equ     17      ;STOP Processing State Bit
498  M_PEN   equ     18      ;PLL Enable Bit
499  M_PCOD  equ     19      ;PLL Clock Output Disable Bit
500  M_PD    equ     $F00000 ;PreDivider Factor Bits Mask (PD0-PD3)
501
502
503  ;------------------------------------------------------------------------
504  ;
505  ;       equates for BIU
506  ;
507  ;------------------------------------------------------------------------
508
509  ;       Register Addresses Of BIU
```

```
510
511
512   M_BCR   equ     $FFFFFB ;Bus Control Register
513   M_DCR   equ     $FFFFFA ;DRAM Control Register
514   M_AAR0  equ     $FFFFF9 ;Address Attribute Register 0
515   M_AAR1  equ     $FFFFF8 ;Address Attribute Register 1
516   M_AAR2  equ     $FFFFF7 ;Address Attribute Register 2
517   M_AAR3  equ     $FFFFF6 ;Address Attribute Register 3
518   M_IDR   equ     $FFFFF5 ;ID Register
519
520   ;       Bus Control Register
521
522   M_BA0W  equ     $1F     ;Area 0 Wait Control Mask (BA0W0-BA0W4)
523   M_BA1W  equ     $3E0    ;Area 1 Wait Control Mask (BA1W0-BA14)
524   M_BA2W  equ     $1C00   ;Area 2 Wait Control Mask (BA2W0-BA2W2)
525   M_BA3W  equ     $E000   ;Area 3 Wait Control Mask (BA3W0-BA3W3)
526   M_BDFW  equ     $1F0000 ;Default Area Wait Control Mask (BDFW0-BDFW4)
527   M_BBS   equ     21      ;Bus State
528   M_BLH   equ     22      ;Bus Lock Hold
529   M_BRH   equ     23      ;Bus Request Hold
530
531   ;       DRAM Control Register
532
533   M_BCW   equ     $3      ;In Page Wait States Bits Mask (BCW0-BCW1)
534   M_BRW   equ     $C      ;Out Of Page Wait States Bits Mask (BRW0-BRW1)
535   M_BPS   equ     $300    ;DRAM Page Size Bits Mask (BPS0-BPS1)
536   M_BPLE  equ     11      ;Page Logic Enable
537   M_BME   equ     12      ;Mastership Enable
538   M_BRE   equ     13      ;Refresh Enable
539   M_BSTR  equ     14      ;Software Triggered Refresh
540   M_BRF   equ     $7F8000 ;Refresh Rate Bits Mask (BRF0-BRF7)
541   M_BRP   equ     23      ;Refresh prescaler
542
543   ;       Address Attribute Registers
544
545   M_BAT   equ     $3      ;External Access Type and Pin Definition Bits Mask (BAT0-BAT1)
546   M_BAAP  equ     2       ;Address Attribute Pin Polarity
547   M_BPEN  equ     3       ;Program Space Enable
548   M_BXEN  equ     4       ;X Data Space Enable
549   M_BYEN  equ     5       ;Y Data Space Enable
550   M_BAM   equ     6       ;Address Muxing
551   M_BPAC  equ     7       ;Packing Enable
552   M_BNC   equ     $F00    ;Number of Address Bits to Compare Mask (BNC0-BNC3)
553   M_BAC   equ     $FFF000 ;Address to Compare Bits Mask (BAC0-BAC11)
554
555   ;       control and status bits in SR
556
557   M_CP    equ     $c00000 ;mask for CORE-DMA priority bits in SR
558   M_CA    equ     0       ;Carry
559   M_V     equ     1       ;Overflow
560   M_Z     equ     2       ;Zero
```

```
561  M_N     equ     3       ;Negative
562  M_U     equ     4       ;Unnormalized
563  M_E     equ     5       ;Extension
564  M_L     equ     6       ;Limit
565  M_S     equ     7       ;Scaling Bit
566  M_I0    equ     8       ;Interupt Mask Bit 0
567  M_I1    equ     9       ;Interupt Mask Bit 1
568  M_S0    equ     10      ;Scaling Mode Bit 0
569  M_S1    equ     11      ;Scaling Mode Bit 1
570  M_SC    equ     13      ;Sixteen_Bit Compatibility
571  M_DM    equ     14      ;Double Precision Multiply
572  M_LF    equ     15      ;DO-Loop Flag
573  M_FV    equ     16      ;DO-Forever Flag
574  M_SA    equ     17      ;Sixteen-Bit Arithmetic
575  M_CE    equ     19      ;Instruction Cache Enable
576  M_SM    equ     20      ;Arithmetic Saturation
577  M_RM    equ     21      ;Rounding Mode
578  M_CP0   equ     22      ;bit 0 of priority bits in SR
579  M_CP1   equ     23      ;bit 1 of priority bits in SR
580
581  ;       control and status bits in OMR
582  M_CDP   equ     $300    ;mask for CORE-DMA priority bits in OMR
583  M_MA    equ     0       ;Operating Mode A
584  M_MB    equ     1       ;Operating Mode B
585  M_MC    equ     2       ;Operating Mode C
586  M_MD    equ     3       ;Operating Mode D
587  M_EBD   equ     4       ;External Bus Disable bit in OMR
588  M_SD    equ     6       ;Stop Delay
589  M_MS    equ     7       ;Memory Switch bit in OMR
590  M_CDP0  equ     8       ;bit 0 of priority bits in OMR
591  M_CDP1  equ     9       ;bit 1 of priority bits in OMR
592  M_BEN   equ     10      ;Burst Enable
593  M_TAS   equ     11      ;TA Synchronize Select
594  M_BRT   equ     12      ;Bus Release Timing
595  M_ATE   equ     15      ;Address Tracing Enable bit in OMR.
596  M_XYS   equ     16      ;Stack Extension space select bit in OMR.
597  M_EUN   equ     17      ;Extensed stack UNderflow flag in OMR.
598  M_EOV   equ     18      ;Extended stack OVerflow flag in OMR.
599  M_WRP   equ     19      ;Extended WRaP flag in OMR.
600  M_SEN   equ     20      ;Stack Extension Enable bit in OMR.
```

A.6 VECTORS.ASM

```
1   ;
2           page    132,60
3   ;***************************************************************************
4   ;VECTORS.ASM
5   ;Vector table for the 56303
6   ;
7   ;Copyright (c) MOTOROLA 1996
8   ;Semiconductor Products Sector
9   ;Digital Signal Processing Division
10  ;
11  ;***************************************************************************
12  ;
13          org p:0
14  ;
15  vectors jmp     START   ;Hardware RESET
16  ;
17          jmp     *
18          nop     ;Stack Error
19
20          jmp     *
21          nop     ;- Debug Request Interrupt
22  ;
23          jmp     *
24          nop     ;- Debug Request Interrupt
25  ;
26          jmp     *
27          nop     ;- Trap
28  ;
29          jmp     *
30          nop     ;- NMI
31  ;
32          nop     ;- Reserved
33          nop
34  ;
35          nop     ;- Reserved
36          nop
37  ;
38          jsr     main    ;- IRQA
39  ;
40          jmp     *
41          nop     ;- IRQB
42  ;
43          jmp     *
44          nop     ;- IRQC
45  ;
46          jsr     echo    ;- IRQD
47  ;
48          jmp     *
49          nop     ;- DMA Channel 0
50  ;
```

```
51          jmp     *
52          nop     ;- DMA Channel 1
53    ;
54          jmp     *
55          nop     ;- DMA Channel 2
56    ;
57          jmp     *
58          nop     ;- DMA Channel 3
59    ;
60          jmp     *
61          nop     ;- DMA Channel 4
62    ;
63          jmp     *
64          nop     ;- DMA Channel 5
65    ;
66          jmp     *
67          nop     ;- Timer 0 Compare
68    ;
69          jmp     *
70          nop     ;- Timer 0 Overflow
71    ;
72          jmp     *
73          nop     ;- Timer 1 Compare
74    ;
75          jmp     *
76          nop     ;- Timer 1 Overflow
77    ;
78          jmp     *
79          nop     ;- Timer 2 Compare
80    ;
81          jmp     *
82          nop     ;- Timer 2 Overflow
83    ;
84          jsr     ssi_rx_isr      ;- ESSI0 Receive Data
85    ;
86          jsr     ssi_rxe_isr     ;- ESSI0 Receive Data w/ Exception Status
87    ;
88          jsr     ssi_rxls_isr    ;- ESSI0 Receive last slot
89    ;
90          jsr     ssi_tx_isr      ;- ESSI0 Transmit Data
91    ;
92          jsr     ssi_txe_isr     ;- ESSI0 Transmit Data w/ Exception Status
93    ;
94          jsr     ssi_txls_isr    ;- ESSI0 Transmit last slot
95    ;
96          nop     ;- Reserved
97          nop
98    ;
99          nop     ;- Reserved
100         nop
101   ;
```

```
102           jmp     *
103           nop     ;- ESSI1 Receive Data
104     ;
105           jmp     *
106           nop     ;- ESSI1 Receive Data w/ Exception Status
107     ;
108           jmp     *
109           nop     ;- ESSI1 Receive last slot
110     ;
111           jmp     *
112           nop     ;- ESSI1 Transmit Data
113     ;
114           jmp     *
115           nop     ;- ESSI1 Transmit Data w/ Exception Status
116     ;
117           jmp     *
118           nop     ;- ESSI1 Transmit last slot
119     ;
120
121           nop     ;- Reserved
122           nop
123     ;
124           nop     ;- Reserved
125           nop
126     ;
127           jmp     *
128           nop     ;- SCI Receive Data
129     ;
130           jmp     *
131           nop     ;- SCI Receive Data w/ Exception Status
132     ;
133           jmp     *
134           nop     ;- SCI Transmit Data
135     ;
136           jmp     *
137           nop     ;- SCI Idle Line
138     ;
139           jmp     *
140           nop     ;- SCI Timer
141     ;
142           nop     ;- Reserved
143           nop
144     ;
145           nop     ;- Reserved
146           nop
147     ;
148           nop     ;- Reserved
149           nop
150     ;
151     ;
152           jmp     *
```

```
153          nop     ;Host receive data full
154    ;
155    ;
156          jmp     *
157          nop     ;- Host transmit data empty
158    ;
159          jmp     *
160          nop     ;Available for Host Command
161          jmp     *
162          nop     ;Available for Host Command
163          jmp     *
164          nop     ;Available for Host Command
165          jmp     *
166          nop     ;Available for Host Command
167          jmp     *
168          nop     ;Available for Host Command
169          jmp     *
170          nop     ;Available for Host Command
171          jmp     *
172          nop     ;Available for Host Command
173          jmp     *
174          nop     ;Available for Host Command
175          jmp     *
176          nop     ;Available for Host Command
177          jmp     *
178          nop     ;Available for Host Command
179          jmp     *
180          nop     ;Available for Host Command
181          jmp     *
182          nop     ;Available for Host Command
183          jmp     *
184          nop     ;Available for Host Command
185          jmp     *
186          nop     ;Available for Host Command
187          jmp     *
188          nop     ;Available for Host Command
189          jmp     *
190          nop     ;Available for Host Command
191          jmp     *
192          nop     ;Available for Host Command
193          jmp     *
194          nop     ;Available for Host Command
195          jmp     *
196          nop     ;Available for Host Command
197          jmp     *
198          nop     ;Available for Host Command
199          jmp     *
200          nop     ;Available for Host Command
201          jmp     *
202          nop     ;Available for Host Command
203          jmp     *
```

```
204        nop     ;Available for Host Command
205        jmp     *
206        nop     ;Available for Host Command
207        jmp     *
208        nop     ;Available for Host Command
209        jmp     *
210        nop     ;Available for Host Command
211        jmp     *
212        nop     ;Available for Host Command
213        jmp     *
214        nop     ;Available for Host Command
215        jmp     *
216        nop     ;Available for Host Command
217        jmp     *
218        nop     ;Available for Host Command
219        jmp     *
220
221        nop     ;Available for Host Command
222        jmp     *
223        nop     ;Available for Host Command
224        jmp     *
225        nop     ;Available for Host Command
226        jmp     *
227        nop     ;Available for Host Command
228        jmp     *
229        nop     ;Available for Host Command
230        jmp     *
231        nop     ;Available for Host Command
232        jmp     *
233        nop     ;Available for Host Command
234        jmp     *
235        nop     ;Available for Host Command
236        jmp     *
237        nop     ;Available for Host Command
238        jmp     *
239        nop     ;Available for Host Command
240        jmp     *
241        nop     ;Available for Host Command
242        jmp     *
243        nop     ;Available for Host Command
244        jmp     *
245        nop     ;Available for Host Command
246        jmp     *
247        nop     ;Available for Host Command
248        jmp     *
249        nop     ;Available for Host Command
250        jmp     *
251        nop     ;Available for Host Command
252        jmp     *
253        nop     ;Available for Host Command
254        jmp     *
```

```
255         nop     ;Available for Host Command
256         jmp     *
257         nop     ;Available for Host Command
258         jmp     *
259         nop     ;Available for Host Command
260         jmp     *
261         nop     ;Available for Host Command
262         jmp     *
263         nop     ;Available for Host Command
264         jmp     *
265         nop     ;Available for Host Command
266         jmp     *
267         nop     ;Available for Host Command
268         jmp     *
269         nop     ;Available for Host Command
270         jmp     *
271         nop     ;Available for Host Command
272         jmp     *
273         nop     ;Available for Host Command
274         jmp     *
275         nop     ;Available for Host Command
276         jmp     *
277         nop     ;Available for Host Command
278         jmp     *
279         nop     ;Available for Host Command
280         jmp     *
281         nop     ;Available for Host Command
282         jmp     *
283
284
285         nop     ;Available for Host Command
286         jmp     *
287         nop     ;Available for Host Command
288         jmp     *
289         nop     ;Available for Host Command
290         jmp     *
291         nop     ;Available for Host Command
292         jmp     *
293         nop     ;Available for Host Command
294         jmp     *
295         nop     ;Available for Host Command
296         jmp     *
297         nop     ;Available for Host Command
298         jmp     *
299         nop     ;Available for Host Command
300         jmp     *
301         nop     ;Available for Host Command
302         jmp     *
303         nop     ;Available for Host Command
304         jmp     *
305         nop     ;Available for Host Command
```

```
306         jmp     *
307         nop     ;Available for Host Command
308         jmp     *
309         nop     ;Available for Host Command
310         jmp     *
311         nop     ;Available for Host Command
312         jmp     *
313         nop     ;Available for Host Command
314         jmp     *
315         nop     ;Available for Host Command
316         jmp     *
317         nop     ;Available for Host Command
318   ;
319   ;
```

Appendix B

Modified Pass Pack

B.1 PASS.ASM

```
1   ;******************************************************************************
2   ; NAME : pass.asm v1.7b
3   ;
4   ; DESCRITION : Base pass code for Motorola 56302EVM.
5   ;
6   ; INPUTS :
7   ;
8   ; OUTPUTS :
9   ;
10  ; NOTES :
11  ;  (Print with Courier 10pt and set 1 tab = 8 spaces)
12  ;  1) Set ADA_OFF = 1 (in .DAT file) to disable codec (used in debugging)
13  ;  2) Do not use r6 since this is reserved for software stack (used by ISRs)
14  ;
15  ;       (Additional notes indicating use, restrictions, and assorted reminders)
16  ;
17  ; FILES: pass.dat, proginit.asm, procster.asm, ada_init.asm, ada_equ.asm,
18  ;       intequ.asm, ioequ.asm, vectors.asm.
19  ;       (Please list additional required program files.)
20  ;
21  ; CHANGES:
22  ; 2009-02-19 (PLD) minor update with improved documentation and labeling
23  ;***************************************************************************
24
25          nolist
26          include 'ioequ.asm'
27          include 'intequ.asm'
28          include 'ada_equ.asm'
29          include 'vectors.asm'
30          list
31
32  ;******************************************************************************
33  ; pass.dat includes additional memory layout infomation.
```

```
34  ;****************************************************************************
35          include 'pass.dat'               ;include user data
36
37          org p:$100
38  START
39          movep   #$040003,x:M_PCTL       ;set PLL for MPY of 4X (16MHz clk => 64Mhz DSP)
40          movep   #$012422,x:M_BCR        ;set up one ext. wait state for all AAR areas
41          movep   #AAR0,x:M_AAR0          ;memory partition stuff (see p.4-6--4-8 in
42          movep   #AAR3,x:M_AAR3          ;DSP56302EVM UM)
43          movep   #$012421,x:M_BCR        ;set up one ext. wait state for all AAR areas
44          ori     #3,mr                   ;mask interrupts
45          movec   #0,sp                   ;clear hardware stack pointer
46          move    #$004000,omr            ;operating mode 0 (b14 set for ext mem access)
47          move    #STACK,r6               ;initialise stack pointer
48          move    #$FFFF,m6               ;linear addressing of stack
49
50          jsr     ada_init                ;initialize codec (see ada_init.asm)
51
52          IF      (ADA_OFF==1)            ;used to enable/disable codec (see pass.dat)
53          ori     #$03,mr                 ;Mask all interrupts until needed.
54          ENDIF
55
56          jsr     proginit                ;program initialization (see proginit.asm)
57
58          move    #TONE_OUTPUT,y0         ;set up control words and store
59          move    y0,x:TX_BUFF_BASE+2     ;these may be moved into main_loop for various...
60          move    #TONE_INPUT,y0          ;...output control options such as AGC
61          move    y0,x:TX_BUFF_BASE+3
62          move    #0,y0
63          move    y0,x:RX_BUFF_BASE       ;clear out receive right
64          move    y0,x:RX_BUFF_BASE+1     ;clear out receive left
65
66  main_loop
67          jset    #2,x:M_SSISR0,*         ;wait for frame sync to pass
68          jclr    #2,x:M_SSISR0,*         ;wait for frame sync
69
70          move    x:RX_BUFF_BASE,x0       ;receive right sample
71          move    x:RX_BUFF_BASE+1,y0     ;receive left sample
72
73          jsr     process_stereo          ;stereo signal processing (see procster.asm)
74
75          move    x0,x:TX_BUFF_BASE       ;transmit right sample
76          move    y0,x:TX_BUFF_BASE+1     ;transmit left sample
77          jmp     main_loop
78
79          include 'ada_init.asm'
80          include 'proginit.asm'
81          include 'procster.asm'
82  echo
83          end
```

B.2 PASS.DAT

```
1   ;****************************************************************************
2   ;PASS.DAT: This data file is used with PASS.ASM to lay out memory.
3   ;****************************************************************************
4
5   ;**********
6   ; Equates
7   ;**********
8   ADA_OFF equ     0       ;0 enables codec, 1 disables codec
9   AAR0    equ     $010931 ;split external (off-chip) 32K RAM into 16K X & 16K Y
10  AAR3    equ     $010925 ;beginning at $010000 p.4-6--4-8 DSP56302EVM UM
11
12  ;************
13  ; Data Memory
14  ;************
15  ;*******************************************************************************
16  ;---Buffer for the CS4215
17  ;       The following lines of code are required for proper on-board audio
18  ;       codec operation.
19  ;*******************************************************************************
20          org x:$000000   ;On-chip X memory $000000 -- $001BFF
21  RX_BUFF_BASE    equ     *
22  RX_data_1_2     ds      1       ;data time slot 1/2 for RX ISR
23  RX_data_3_4     ds      1       ;data time slot 3/4 for RX ISR
24  RX_data_5_6     ds      1       ;data time slot 5/6 for RX ISR
25  RX_data_7_8     ds      1       ;data time slot 7/8 for RX ISR
26
27  TX_BUFF_BASE    equ     *
28  TX_data_1_2     ds      1       ;data time slot 1/2 for TX ISR
29  TX_data_3_4     ds      1       ;data time slot 3/4 for TX ISR
30  TX_data_5_6     ds      1       ;data time slot 5/6 for TX ISR
31  TX_data_7_8     ds      1       ;data time slot 7/8 for TX ISR
32
33  RX_PTR          ds      1       ;Pointer for rx buffer
34  TX_PTR          ds      1       ;Pointer for tx buffer
35
36
37          org x:$00000a
38  ;*******************************************************************************
39  ;---On-chip X data goes here
40  ;*******************************************************************************
41
42
43  ;*******************************************************************************
44  ;---Software Stack
45  STACK   equ     *       ;locate stack after last thing in on-chip X memory
46                          ;STACK is used in TXRX_ISR.ASM and must be allowed to grow
47
48  ;You must not write *anything* into x:$001C00 --x:$00FFFF (check .LST file)
49
50
```

```
51          org x:$010000   ;Off-Chip X memory $010000 -- $013FFF
52   ;******************************************************************************
53   ;---Off-chip X data goes here
54   ;       If you load actual values in off-chip memory, i.e. "dc" then you must run
55   ;       pass.asm first so that off-chip memory can be configured before the load
56   ;       is attempted.  In this case, Debugger should have Reset on Load (Config
57   ;       menu) unchecked.
58   ;******************************************************************************
59
60
61          org y:$000000   ;On-chip Y memory $000000 -- $001BFF
62   ;******************************************************************************
63   ;---On-chip Y data goes here
64   ;******************************************************************************
65
66
67   ;You must not write *anything* into y:$001C00 --y:$00FFFF (check .LST file)
68
69
70          org y:$010000   ;Off-chip Y memory $010000 -- $013FFF
71   ;******************************************************************************
72   ;---Off-chip Y data goes here
73   ;       If you load actual values in off-chip memory, i.e. "dc" then you must run
74   ;       pass.asm first so that off-chip memory can be configured before the load
75   ;       is attempted.  In this case, Debugger should have Reset on Load (Config
76   ;       menu) unchecked.
77   ;******************************************************************************
```

B.3 PROGINIT.ASM

```
1   ;***************************************************************************
2   ;PROGINIT.ASM: This file contains the program initialization subroutine.
3   ;***************************************************************************
4
5   proginit
6           nop
7
8           rts
```

B.4 PROCSTER.ASM

```
1   ;***************************************************************************
2   ;PROCSTER.ASM: This file contains the process stereo subroutine.
3   ;***************************************************************************
4
5   process_stereo
6           nop
7
8           rts
```

B.5 ADA_INIT.ASM

This file remains unchanged from Motorola Pass Pack except for two additional NOPs located before line 153 and two additional NOPs located before line 178. The addition of these NOPs prevent (harmless) pipeline stall warning messages from the assembler.

B.6 ADA_EQU.ASM

This file remains unchanged from Motorola Pass Pack except that equates for TONE_OUTPUT and TONE_INPUT are moved from Motorola PASS.ASM to this file.

B.7 INTEQU.ASM

This file remains unchanged from Motorola Pass Pack.

B.8 IOEQU.ASM

This file remains unchanged from Motorola Pass Pack.

B.9 VECTORS.ASM

Line 38 is changed from

```
jsr main ;- IRQA
```

to

```
jsr main _loop;- IRQA
```

Index

ABS, 98
Absorption Coefficient, 167
Accumulator, 23
ada_equ.asm, 150
ada_init.asm, 150
Adaptive Filter, 192
Adaptive Line Enhancer, 203
ADD, 89
ADDL, 90
ADDR, 90
Address Bus
  Program-, 8
  X-, 8
  Y-, 8
Address Generation Unit, 8, 46, 125
Addressing
  Absolute, 53
  Bit Reversed, 65
  Indexed by Offset, 58
  Modulo, 62
  Postdecrement, 56
  Postdecrement by Offset, 57
  Postincrement, 56
  Postincrement by Offset, 57
  Predecrement, 58
  Register Indirect, 55
  Short, 54
Addressing Modes, 44
  Bit-reversed, 277, 281
  Capabilities, 48
  Categories, 49
ADSR, 231
  Guitar Envelope, 232
  Piano Envelope, 232
AGU, *see* Address Generation Unit
All-Pass Filter, 174
Alphabet, 251
ALU, *see* Arithmetic Logic Unit
Amplitude Modulation, 232
AND, 99
ANDI, 100
Apparent Source Width, 165
ARI, *see* Address Register Indirect
Arithmetic Logic Unit, 8, 29, 30, 124
ASCII, 257
ASL, 94
ASR, 94
Assembler, 5, 139, 141
  In-line, 146
Asynchronous, 256

Balance, 166
Baud Rate, 253
BCHG, 109
BCLR, 109
Breakpoints, 145
BSET, 109
BTST, 109

Carrier, 251
CCR, *see* Condition Code Register

Character, 256
Clarity, 166
CLD file, 141
CLR, 98
CMP, 95
CMPM, 95
Codec, 5, 155
Comb Filter, 169
Condition Code Register, 27, 37, 127
Correlation Matrix, 191
Cross Correlation Vector, 191

Data Bus
    Global-, 8
    Program-, 8
    X-, 8
    Y-, 8
Data Rate, R_b, 253
Data Transfers, 31
    Examples, 32
DC, 143
DEBUG, 114
DEBUGcc, 114
Debugger, 5, 139
Debugging Tips, 145
DEC, 114
Decorrelation Delay, 203
Definition, *see* Carity166
Diffusion, 166
Digital Modulation, 251
    APK, 254
    ASK, 251, 252
    FSK, 251, 252
    OOK, 252
    PSK, 251, 253
Digital Signal Processing, 3
Direct Sound, 164
Directives, 140
Discrete Fourier Transform (DFT), 273
DIV, 111
DO, 104
DS, 143
DSM, 144
DSP56300
    Clock Speed, 153
    Core, 46
    Derivatives, 15, 121
    Family, 12, 119
DSP56301
    Block Diagram, 14
    Memory Map, 17
    Block Diagram, 120
    Memory Map, 122
DSP56302
    Block Diagram, 15
    Evaluation Module, 133

Early Reflections, 164
Editor, 5
EMR, *see* Extended Mode Register
ENDDO, 105
EOR, 99
EQU, 142
Evaluation Module, 5
    Self Test, 140
EVM, *see* Evaluation Module
Expectation, 189
Extended Mode Register, 27, 127
Extension Bit, 38

Fast Fourier Transform (FFT), 274
    Decimation in Time, 275
    Butterfly, 277, 278
FFT, 64

FIR Filter, 18, 108, 145, 156, 212
Flag Bit, 249
Free Field, 163
Frequency Modulation, 251
Full-duplex, 256
Fundamental Table Frequency, 217
Fundamental Table Period, 219

GDB, *see* Data Bus

Half-duplex, 256
Hardware Stack, 26
Harvard architecture, 7

In-place Calculation, 277
INC, 114
Inner product, 4
intequ.asm, 150
Interaural Cross-Correlation Coefficient, 165
Intimate, 165
Inverse Fast Fourier Transform (IFFT), 282
ioequ.asm, 150

Jcc, 107
JCLR, 110
JMP, 107
JScc, 107
JSCLR, 110
JSET, 110
JSR, 107
JSSET, 110

Least Mean-Square Algorithm, 194
Least-Squares Filter, 191
Limiting, 35
Linear Interpolation, 226
Linker, 139
Listener Envelopment, 165
Liveliness, 166
LMS, *see* Least Mean-Square
Lookup Table, 217
    Fundamental Table Frequency, 217
    Fundamental Table Period, 219
    Generation, 219
    sinelut.asm, 219
    singenid.asm, 221
    singenli.asm, 230
    singenrd.asm, 225
    Table Increment, 217
Loop, 28, 104
Loop Register, 27
Loudness, 166
LR, *see* Loop Register
LSL, 101
LSR, 101
LST file, 141
LUA, 88

MAC, 93
MACR, 93
Macro, 213
    Definition, 214
Matched Filter, 263
Maximum Likelihood Detector, 262
Mean-Squared Error, 190
Memory
    X-, Y-, 21
Memory Types, 19
Minimum Mean-Squared Error, 191
Mode Register, 27, 127
Modem, 251
    Bell Type 103, 254
Modified Pass Pack, 150
Modifier Registers, 60
    Address Registers, 60
    Offset Registers, 60

Values, 61
Motorola Pass Pack, 150
MOVE, 50, 86
Immediate Data, 51
Short, 52
MPY, 92
MPYR, 92
MR, *see* Mode Register

NEG, 98
NLMS, *see* Normalized Least Mean-Square
NOP, 114
NORM, 98
Normalized Least Mean-Square Algorithm, 195
NOT, 99
Notch Filter, 189
Note, Musical, 237
Frequencies, 237

Operand, 19
Accumulator, 23
Longword, 22
Word, 20
Operating Mode Register, 129
Optimal Filter, 191
OR, 99
ORG, 142
ORI, 100

PAB, *see* Address Bus
Parallel Move, 69
Address Register Update, 74
Immediate Short Data, 72
Instruction Set, 70
Long Memory, 79
Register to Register, 73, 76, 78
Summary, 81
Syntax, 71
X Memory, 75, 76
XY Memory, 80
Y Memory, 77, 78
Parity Bit, 256
pass.asm, 150
PC, *see* Program Controller
PCU, *see* Program Control Unit
PDB, *see* Data Bus
Phase Modulation, 251
Power Spectral Density, 205
Prediction Depth, 203
Princeton architecture, 7
Program Control Unit, 8, 126
Program Controller, 25
Programming Model, 24, 123

Real-Time Digital Signal Processing, 3
REP, 102
RESET, 114
Reverberation, 164
Reverberation Time, 165, 167
RND, 98
ROL, 101
ROR, 101
Rotate, 101
Rounding, 36
RTI, 107
RTS, 107

Sample Rate, 154
Scaling, 39
Effects, 40
Shift, 101
Simplex, 256
Spatial Impression, 165
SR, *see* Status Register
Stack

Software, 151
Start Bit, 256
Status Register, 27, 127
Steepest Descent, 193
STOP, 114
Stop Bit, 256
SUB, 89
SUBL, 90
SUBR, 90
Subroutines, 212
SWI, 114
Symbol, 251

Tapped Delay Line, 168
Tcc, 96
TFR, 96
Transfer, *see* TR96
TST, 95

Unit Reverberator, 171
Unnormalized Bit, 38

vectors.asm, 150

WAIT, 114
Warmth, 166
Wiener Filter, 189

XAB, *see* Address Bus
XDB, *see* Data Bus

YAB, *see* Address Bus
YDB, *see* Data Bus