REAL TIME IMPLEMENTATION OF DES ALGORITHM BY USING TMS3206713 DSK.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology

in

Telematics & Signal Processing

By

B.Rajesh

Department of Electronics and Communication Engineering

National Institute Of Technology

Rourkela

2008

REAL TIME IMPLEMENTATION OF DES ALGORITHM BY USING TMS3206713 DSK.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology

in

Telematics & Signal Processing

By

B.Rajesh

Under the Guidance of

Prof. G.S.Rath.

Department of Electronics and Communication Engineering

National Institute Of Technology

Rourkela

2008

National Institute Of Technology
Rourkela

CERTIFICATE

This is to certify that the thesis entitled, "**Real Time Implementation of DES Algorithm by using TMS320C6713 DSK**" submitted by **B.Rajesh** in partial fulfillment of the requirements for the award of Master of Technology Degree in **Electronics & communication Engineering** with specialization in "**Telematics & Signal Processing**" at the National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University / Institute for the award of any Degree or Diploma.

<div align="right">

Prof. G.S.Rath.
Dept. of Electronics & Communication Engg.
National Institute of Technology
Rourkela-769008

</div>

Date: 29-05-2008.

Acknowledgement

First of all, I would like to express my deep sense of respect and gratitude towards my advisor and guide   **Prof. G. S. Rath**, who has been the guiding force behind this work. I am greatly indebted to him for his constant encouragement, invaluable advice and for propelling me further in every aspect of my academic life. His presence and optimism have provided an invaluable influence on my career and outlook for the future. I consider it my good fortune to have got an opportunity to work with such a wonderful person.

Next, I want to express my respects to **Prof. G. Panda, Prof. K. K. Mahapatra, Prof. S.K. Patra** and **Dr. S. Meher** for teaching me and also helping me how to learn. They have been great sources of inspiration to me and I thank them from the bottom of my heart.

I would like to thank all faculty members and staff of the Department of Electronics and Communication Engineering, N.I.T. Rourkela for their generous help in various ways for the completion of this thesis.

I would also like to mention the names of **madhu** and **arun** for helping me a lot during the thesis period.

I would like to thank all my friends and especially my classmates for all the thoughtful and mind stimulating discussions we had, which prompted us to think beyond the obvious. I've enjoyed their companionship so much during my stay at NIT, Rourkela.

I am especially indebted to my parents for their love, sacrifice, and support. They are my first teachers after I came to this world and have set great examples for me about how to live, study, and work.

Rajesh.B

Roll No: 20607025

Dept of ECE, NIT, Rourkela

Contents

ABSTRACT

The data encryption standard (DES) is an algorithm that was formerly considered to be the most popular method for private key encryption. DES is still appropriate for moderately secured communication. In this project I have implemented DES algorithm for voice data encryption by using the Texas Instruments TMS320C6713 dsp processor. TMS320C6713 is a 32-bit floating point dsp processor which is one of the Texas TMS320C6x family. Digital signal processors such as the TMS320C6x (C6x) family of processors are like fast special-purpose microprocessors with a specialized type of architecture and an instruction set appropriate for signal processing. The architecture of the C6x digital signal processor is very well suited for numerically intensive calculations. Based on a very-long-instruction-word (VLIW) architecture, the C6x is considered to be TI's most powerful processor.

List of figures:

List of tables:

INTRODUCTION

Cryptography is the art of communicating with secret data. In voice communication, cryptography refers to the encrypting and decrypting of voice data through a possibly insecure data line. The goal is to prevent anyone who does not have a "key" from receiving and understanding a transmitted message.

The data encryption standard (DES) is an algorithm that was formerly considered to be the most popular method for private key encryption. DES is still appropriate for moderately secured communication.

IN this project I have implemented DES algorithm to voice data by using the TMS320C6713 dsp processor. TMS320C6713 is a 32-bit floating point dsp processor which is one of the Texas TMS320C6x family. Digital signal processors such as the TMS320C6x (C6x) family of processors are like fast special-purpose microprocessors with a specialized type of architecture and an instruction set appropriate for signal processing. The architecture of the C6x digital signal processor is very well suited for numerically intensive calculations. Based on a very-long-instruction-word (VLIW) architecture, the C6x is considered to be TI's most powerful processor.

Digital signal processors are used for a wide range of applications, from communications and controls to speech and image processing. The general-purpose digital signal processor is dominated by applications in communications (cellular). Applications embedded digital signal processors are dominated by consumer products. They are found in cellular phones, fax/modems, disk drives, radio, printers, hearing aids, MP3 players, high-definition television (HDTV), digital cameras, and so on.

DES is a bit-manipulation technique with a 64-bit block cipher that uses an effective key of 56 bits. It is an iterated Feistel-type cipher with 16 rounds. The general model of DES has three main components for: (1) initial permutation; (2) encryption—the core iteration/*f*-function (16 rounds); and (3) final permutation.

This thesis consists of five chapters. For processing this project first there is a need of knowing about the TMS320C6713 dsp processor that means we want to know about its architecture and how its take inputs and how its produce outputs etc. that's why in second chapter I have explained about the total architecture of TMS320C67 processor and in third chapter I have explained how this processor takes inputs and how its produce outputs and in fourth chapter I have explained about the architecture and instruction set of TMS320C67 processor. In fourth chapter I have explained about DES algorithm and the procedure to implement the DES algorithm for voice signal by using TMS320C67 processor.

CHAPTER-1

Data Encryption Standard Algorithm

1.1 DES Model

Cryptography is the art of communicating with secret data. In voice communication, cryptography refers to the encrypting and decrypting of voice data through a possibly insecure data line. The goal is to prevent anyone who does not have a "key" from receiving and understanding a transmitted message.

The data encryption standard (DES) is an algorithm that was formerly considered to be the most popular method for private key encryption. DES is still appropriate for moderately secured communication.

DES is a bit-manipulation technique with a 64-bit block cipher that uses an effective key of 56 bits. It is an iterated Feistel-type cipher with 16 rounds. The general model of DES has three main components for (see Figure 4.1): (1) initial permutation; (2) encryption—the core iteration/f-function (16 rounds); and (3) final permutation. X and Y are the input and output data streams in 64-bit block segments, respectively, and $K1$ through $K16$ are distinct keys used in the encryption algorithm. The initial permutation is based on the predefined Table 1.1. The value at each position is used to scramble the input before the encryption routine. For example, the 58th data to be moved into the first position
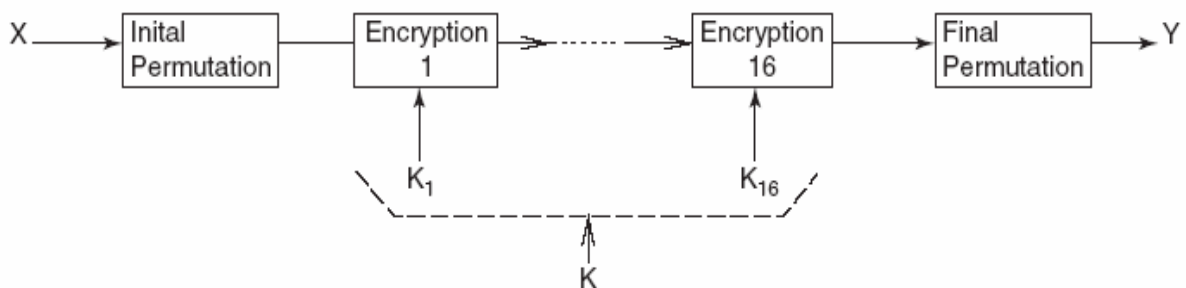
Fig 1.1: DES Model

Table 1.1: Initial permutation table:

IP							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

of a 64-bit array, the 50th bit into position 2, and so on. The input stream is permutated using a nonrepetitive random table of 64 integers (1–64) that corresponds to a new position of each bit in the 64-bit data block. The final permutation is the reverse of the initial permutation to reorder the samples into the correct original formation. The initial permutation is followed by the actual encryption. The permutated 64-bit block is divided into a left and a right block of 32 bits each.

1.2 Sixteen Rounds of DES

Sixteen rounds take place, each undergoing a similar procedure, as illustrated in Figure 1.3. The right block is placed into the left block of the next round, and the left block is combined with an encoded version of the right block and placed into the right block of the next round, or

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \text{ XOR } f(R_{i-1}, k_i)$$

where L_{i-1} and R_{i-1} are the left and right blocks, respectively, each with 32 bits, and k_i is the distinct key for the particular round of encryption. The original key is sent through a

key scheduler that alters the key for each round of encryption. The left block is not utilized until the very end, when it is XORed with the encrypted right block.

The *f*-function operating on a 32-bit quantity expands these 32 bits into 48 bits using the expansion table (see Table 1.2). This expansion table performs a permutation while duplicating 16 of the bits (the rightmost two columns). For example, the first integer is 32, so that the first bit in the output block will be bit 32; the second integer is 1, so that the second bit in the output block will be bit 1; and so on.
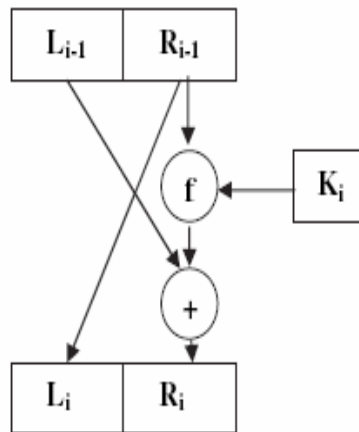
Fig 1.2: Encryption process—one round.

Table 1.2: Expansion of 32 bits to 48.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Table 1.3: S-Box example,S_1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	14

The 48-bit key transformations are XORed with these expanded data, and the results are used as the input to eight different *S*-boxes. Each *S*-box takes 6 consecutive bits and outputs only 4 bits. The 4 output bits are taken directly from the numbers found in a corresponding *S*-box table. This process is similar to that of a decoder where the 6 bits act as a table address and the output is a binary representation of the value at that address. The zeroth and fifth bits determine the row of the *S*-box, and the first through fourth bits determine which column the number is located in. For example, 110100 points to the third row (10) and 10th column (1010). The first 6 bits of data correspond to the first of eight *S*-box tables, shown in Table 1.3. The 32 bits of output from the *S*-boxes are permutated according to the *P*-box shown in Table 1.4, and then output from the *f*-function shown in Figure 1.3. For example, from Table1.4, bits 1 and 2 from the input block will be moved to bits 16 and 7 in the output, respectively. After the 16 rounds of encryption, a final permutation occurs, which reverses the initial permutation, yielding an encrypted data signal.

Table 1.4: P-BOX

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

Fig 1.3: Core f-function of DES

CHAPTER-2

INTRODUCTION TO TMS320C6713 DSPKIT

2.1 INTRODUCTION

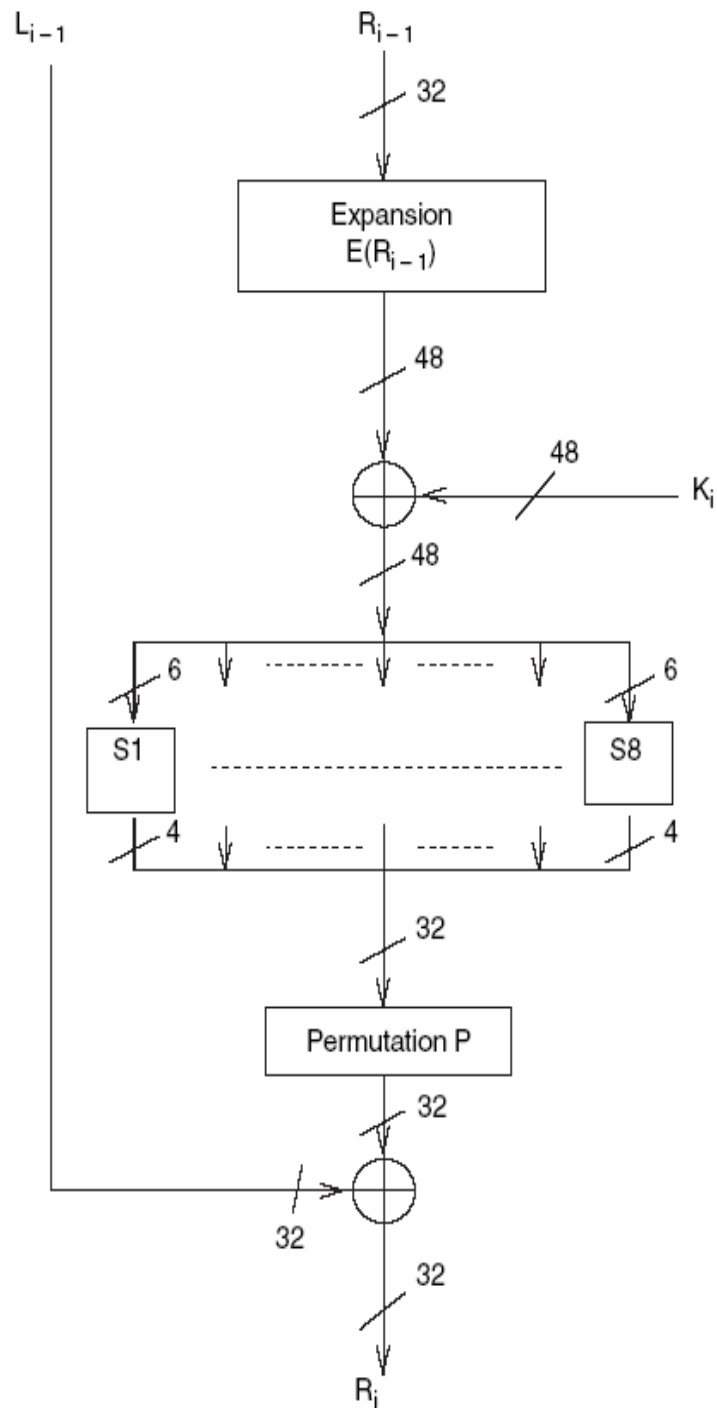Digital signal processors such as the TMS320C6x (C6x) family of processors are like fast special-purpose microprocessors with a specialized type of architecture and an instruction set appropriate for signal processing. The C6x notation is used to designate a member of Texas Instruments' (TI) TMS320C6000 family of digital signal processors. The architecture of the C6x digital signal processor is very well suited for numerically intensive calculations. Based on a very-long-instruction-word (VLIW) architecture, the C6x is considered to be TI's most powerful processor.

Digital signal processors are used for a wide range of applications, from communications and controls to speech and image processing. The general-purpose digital signal processor is dominated by applications in communications (cellular). Applications embedded digital signal processors are dominated by consumer products. They are found in cellular phones, fax/modems, disk drives, radio, printers, hearing aids, MP3 players, high-definition television (HDTV), digital cameras, and so on. These processors have become the products of choice for a number of consumer applications, since they have become very cost-effective. They can handle different tasks, since they can be reprogrammed readily for a different application. DSP techniques have been very successful because of the development of low-cost software and hardware support. For Example, modems and speech recognition can be less expensive using DSP techniques.

DSP processors are concerned primarily with real-time signal processing. Real-time processing requires the processing to keep pace with some external event, whereas non-real-time processing has no such timing constraint. The external event to keep pace with is usually the analog input. Whereas analog-based systems with discrete electronic components such as resistors can be more sensitive to temperature changes, DSP-based systems are less affected by environmental conditions. DSP processors enjoy the advantages of microprocessors. They are easy to use, flexible, and economical.

The basic system consists of an analog-to-digital converter (ADC) to capture an input signal. The resulting digital representation of the captured signal is then processed by a digital signal processor such as the C6x and then output through a digital-to-analog converter (DAC). Also included within the basic system are a special input filter for anti-aliasing to eliminate erroneous signals and an output filter to smooth or reconstruct the processed output signal.

2.2 DSK SUPPORT TOOLS

Most of the work presented in this book involves the design of a program to implement a DSP application. To perform the experiments, the following tools are used:

1. *TI's DSP starter kit (DSK)*. The DSK package includes:

  (a) *Code Composer Studio* (CCS), which provides the necessary software
     Support tools. CCS provides an integrated development environment
     (IDE), bringing together the C compiler, assembler, linker, debugger, and
     so on.

  (b) A board, shown in Figure 1.1 that contains the TMS320C6713 (C6713)
     Floating-point digital signal processor as well as a 32-bit stereo codec for
     Input and output (I/O) support.

  (c) A universal synchronous bus (USB) cable that connects the DSK board
     to a PC.

  (d) A 5V power supply for the DSK board.

2. *An IBM-compatible PC*. The DSK board connects to the USB port of the PC
   Through the USB cable included with the DSK package.

3. *An oscilloscope, signal generator, and speakers*. A signal/spectrum analyzer is optical. Shareware utilities are available that utilize the PC and a sound card to create a virtual instrument such as an oscilloscope, a function generator, or a spectrum analyzer.

2.2.1 DSK Board

The DSK package is powerful, yet relatively inexpensive ($395), with the necessary hardware and software support tools for real-time signal processing. It is a complete DSP system. The DSK board, with an approximate size of 5 * 8 in., includes the C6713 floating-point digital signal processor and a 32-bit stereo codec TLV320AIC23 (AIC23) for input and output.

The onboard codec AIC23 uses a sigma–delta technology that provides ADC and DAC. It connects to a 12-MHz system clock. Variable sampling rates from 8 to 96 kHz can be set readily.

A daughter card expansion is also provided on the DSK board. Two 80-pin connectors provide for external peripheral and external memory interfaces. Two project examples in Chapter 10 illustrate the use of the external memory interface (EMIF) with light-emitting diodes (LEDs) and liquid-crystal displays (LCDs) for spectrum display.

The DSK board includes 16MB (megabytes) of synchronous dynamic random access memory (SDRAM) and 256kB (kilobytes) of flash memory. Four connectors on the board provide input and output: MIC IN for microphone input, LINE IN for line input, LINE OUT for line output, and HEADPHONE for a headphone output (multiplexed with line output). The status of the four user dip switches on the DSK board can be read from a program and provides the user with a feedback control interface. The DSK operates at 225MHz.Also onboard the DSK are voltage regulators that provide 1.26 V for the C6713 core and 3.3 V for its memory and peripherals.
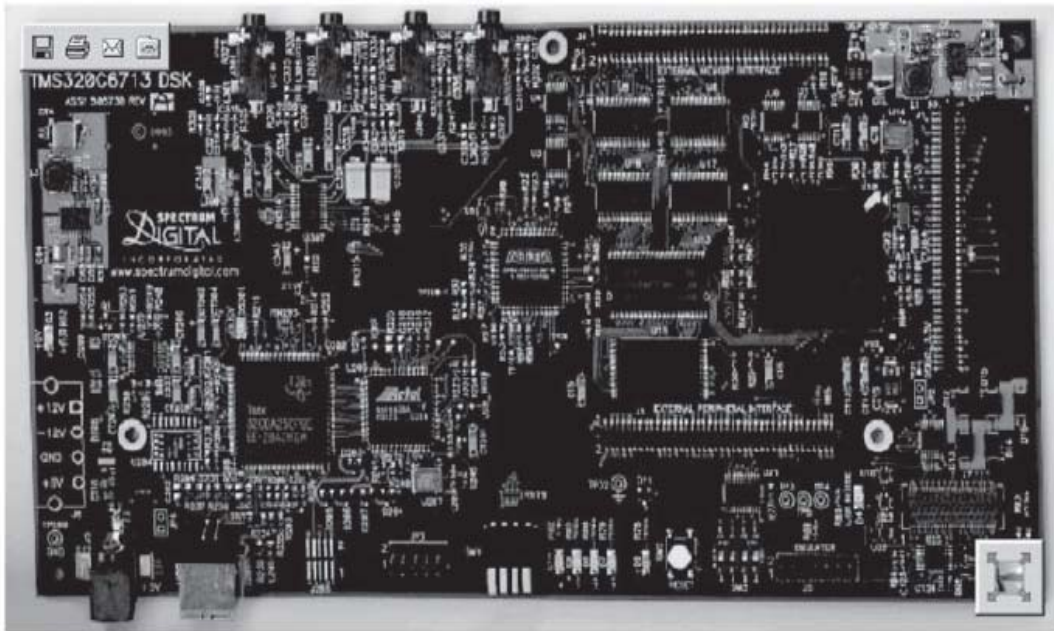
(a)
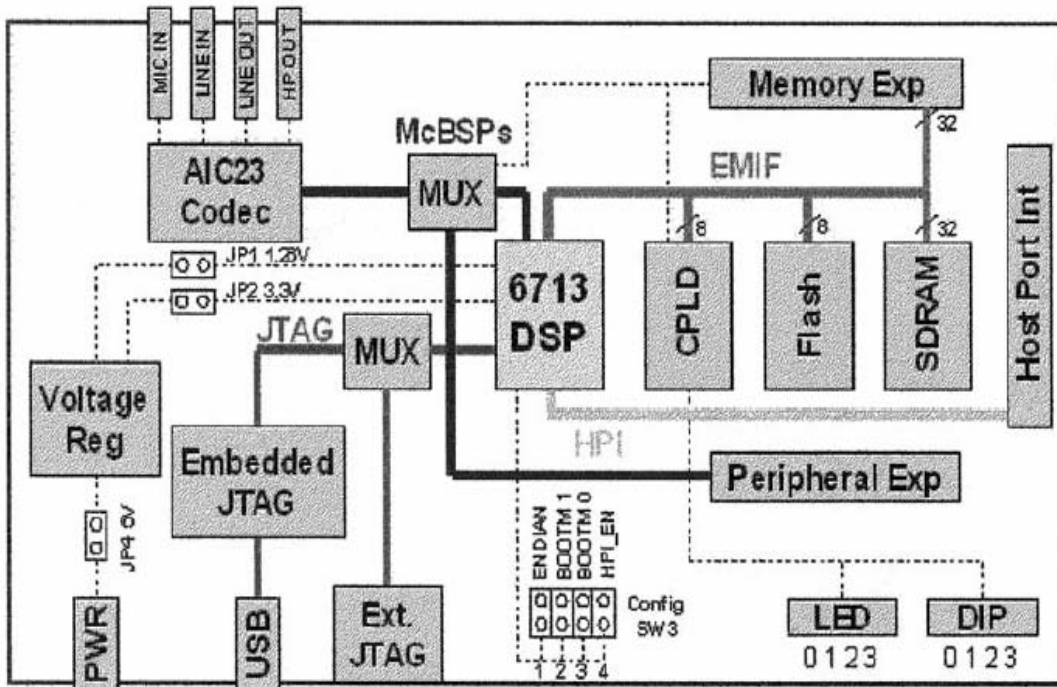
(b)

FIGURE 2.0. TMS320C6713-based DSK board: (*a*) board; (*b*) diagram. (Courtesy of Texas Instruments)

2.2.2 TMS320C6713 Digital Signal Processor

The TMS320C6713 (C6713) is based on the VLIW architecture, which is very well suited for numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. For example, with a clock rate of 225MHz, the C6713 is capable of fetching eight 32-bit instructions every 1/(225 MHz) or 4.44 ns.

Features of the C6713 include 264 kB of internal memory (8kB as L1P and L1D Cache and 256kB as L2 memory shared between program and data space), eight functional or execution units composed of six arithmetic-logic units (ALUs) and two multiplier units, a 32-bit address bus to address 4 GB (gigabytes), and two sets of 32-bit general-purpose registers.

The C67xx (such as the C6701, C6711, and C6713) belong to the family of the C6x floating-point processors, whereas the C62xx and C64xx belong to the family of the C6x fixed-point processors. The C6713 is capable of both fixed- and floating-point processing.

2.3 CODE COMPOSER STUDIO

CCS provides an IDE to incorporate the software tools. CCS includes tools for code generation, such as a C compiler, an assembler, and a linker. It has graphical capabilities and supports real-time debugging. It provides an easy-to-use software tool to build and debug programs.

The C compiler compiles a C source program with extension .c to produce an assembly source file with extension.*asm*.The assembler assembles an.*asm* source file to produce a machine language object file with extension.*obj*.The linker combines object files and object libraries as input to produce an executable file with extension.*out*. This executable file represents a linked common object file format (COFF), popular in Unix-

based systems and adopted by several makers of digital signal processors. This executable file can be loaded and run directly on theC6713 processor.

To create an application project, one can "add" the appropriate files to the project. Compiler/linker options can readily be specified. A number of debugging features are available, including setting breakpoints and watching variables; viewing memory, registers, and mixed C and assembly code; graphing results; and monitoring execution time. One can step through a program in different ways (step into, over, or out).

Real-time analysis can be performed using real-time data exchange (RTDX) . RTDX allows for data exchange between the host PC and the target DSK, as well as analysis in real time without stopping the target. Key statistics and performance can be monitored in real time. Through the joint team action group (JTAG), communication with on-chip emulation support occurs to control and monitor program execution. The C6713 DSK board includes a JTAG interface through the USB port.

2.4(a) QUICK TEST OF DSK

1. On power, a program *post.c* (Power On Self Test), stored in onboard flash
memory, uses the board support library (BSL) to test the DSK. It tests the
internal, external, and flash memories, the two multichannel buffered serial
ports (McBSP), direct memory access (DMA), the onboard codec, and the
LEDs. If all tests are successful, all four LEDs blink three times and stop
(with all LEDs on). During the testing of the codec, a 1-kHz tone is generated
for 1 sec.

2. Launch CCS from the icon on the desktop. A USB enumeration process
Takes place. Then CCS will be opened and the LEDs will turn off. Press GEL
Æ Check DSK Æ Quick Test. The Quick Test can be used for confirmation
Of correct operation and installation. The following message is then
displayed:

Switches: 15

Board Revision: 1

CPLD Revision: 2

This assumes that the four dip switches

This assumes that the four dip switches (0, 1, 2, 3) are all in the up position. Change the switches to (1110)2 so that the first three switches (0, 1, 2) are up and press the fourth switch (3) down. Repeat the procedure to select GEL Æ Check DSK Æ Quick Test and verify that the value of the switches is now 7 (with the display "Switches: 7").You can set the value of the four user switches from 0 to 15.Within your program you can then direct the execution of your code based on these 16 values.

2.4(b) Alternative Quick Test of DSK

1. Open/launch CCS from the icon on the desktop if this has not been done already. Select File ÆLoad Program. Click on the folder *sine8_LED\Debug* within myprojects to load the file *sine8_LED.out*.This loads the executable file *sine8_LED.out* into the C6713 processor. This assumes that you have already copied all the folders on the accompanying CD into your folder: *c:\c6713\myprojects*.

2. Select Debug Æ Run. Press the dip switch #0, which should light LED #0 on and generate a 1-kHz tone. Connect the LINE OUT (or the HEADPHONE) on the DSK board to a speaker or to an oscilloscope and verify the generation of the 1-kHz tone. The four connectors on the DSK board for I/O (MIC, LINE IN, LINE OUT, and HEADPHONE) use a 3.5-mm jack audio cable.

CHAPTER-3

INPUT AND OUTPUT TO THE DSK

3.1 INTRODUCTION

Typical applications using DSP techniques require at least the basic system shown in Figure 2.1, consisting of analog input and output. Along the input path is an antialiasing filter for eliminating frequencies above the *Nyquist frequency*, defined as one-half of the sampling frequency F_s Otherwise, aliasing occurs, in which case a signal with a frequency higher than one-half F_s *is* used as a signal with a lower frequency. The sampling theorem tells us that the sampling frequency must be at least twice the highest-frequency component f in a signal, so that

$$F_s > 2$$

Which s also

$$1/T_s > 2(1/T)$$

Where T_s is the sampling period. Or

$$T_s < T/2$$

The sampling period T_s must be less than one-half the period of the signal. For example, if we assume that the ear cannot detect frequencies above 20 kHz, we can use a lowpass input filter with a bandwidth or cutoff frequency at 20 kHz to avoid aliasing. We can then sample a music signal at $F_s > 40$ kHz (typically, 44.1 or 48 kHz) and remove frequency components higher than 20 kHz. Figure 2.2 illustrates an aliased signal. Let the sampling frequency $F_s = 4$ kHz, or a sampling period of $T_s = 0.25$ms. It is impossible to determine whether it is the 5- or 1-kHz signal that is represented by the sequence (0, 1, 0, -1). A 5-kHz signal will appear as a 1-kHz signal; hence, the 1-kHz signal is an aliased signal. Similarly, a 9-kHz signal would also appear as a 1-kHz aliased signal.
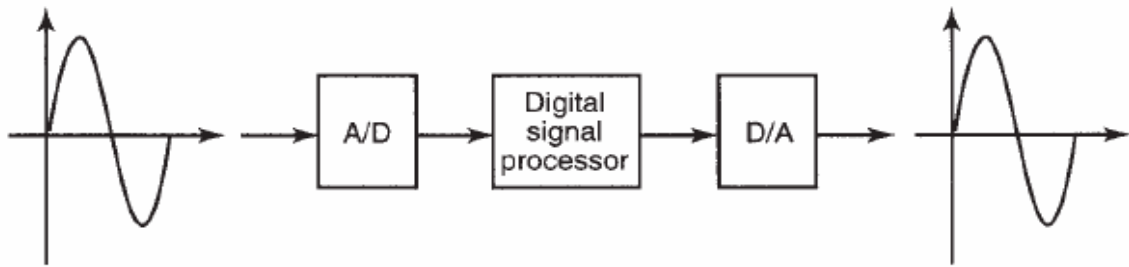
FIGURE 2.1. DSP system with input and output.
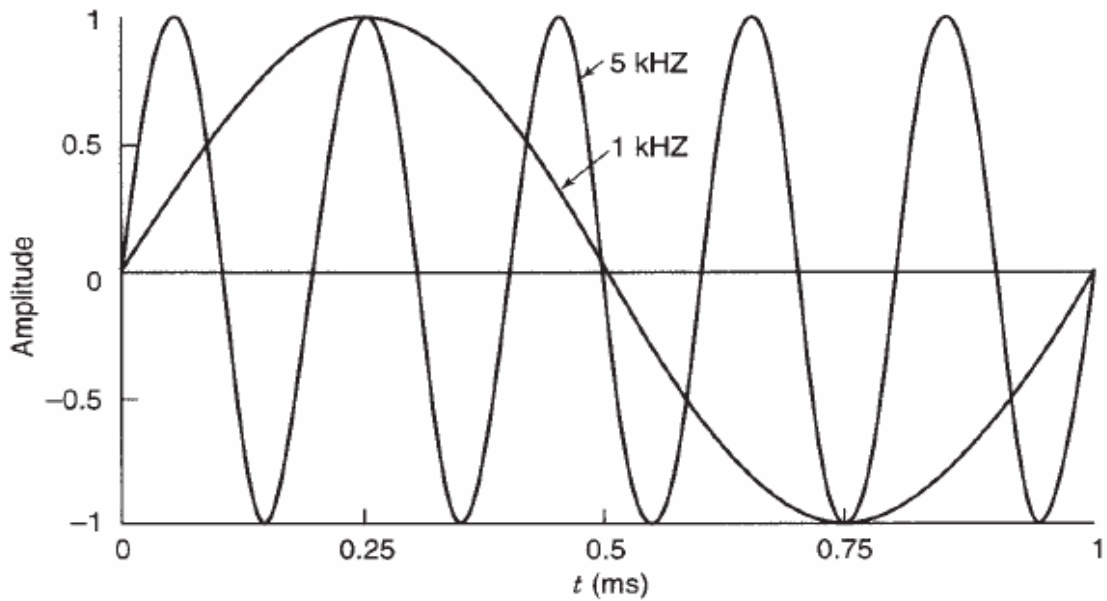
FIGURE 2.2. Aliased sinusoidal signal.

3.2 TLV320AIC23 (AIC23) ONBOARD STEREO CODEC FOR INPUT AND OUTPUT

The DSK board includes the TLV320AIC23 (AIC23) codec for input and output. The ADC circuitry on the codec converts the input analog signal to a digital representation to be processed by the DSP. The maximum level of the input signal to be converted is determined by the specific ADC circuitry on the codec, which is 6V p-p with the onboard codec. After the captured signal is processed, the result needs to be sent to the outside world. Along the output path in Figure 2.1 is a DAC, which performs the reverse

operation of the ADC. An output filter smoothes out or reconstructs the output signal. ADC, DAC, and all required filtering functions are performed by the single-chip codec AIC23 on board the DSK.

The AIC23 is a stereo audio codec based on sigma–delta technology. The functional block diagram of the AIC23 codec is shown in Figure 2.3. It performs all the functions required for ADC and DAC, lowpass filtering, oversampling, and so on. The AIC23 codec contains specifications for data transfer of words with length 16, 20, 24, and 32 bits. A diagram of the AIC23 codec interfaced to the C6713 DSK is shown in *6713_dsk_schem.pdf*, included with the CCS package.

Sigma–delta converters can achieve high resolution with high oversampling ratios but with lower sampling rates. They belong to a category in which the sampling rate can be much higher than the Nyquist rate. Sample rates of 8, 16, 24, 32, 44.1, 48, and 96 kHz are supported and can be readily set in the program.

A digital interpolation filter produces the oversampling. The quantization noise power in such devices is independent of the sampling rate. A modulator is included to shape the noise so that it is spread beyond the range of interest. The noise spectrum is distributed between 0 and $F_s/2$, so that only a small amount of noise is within the signal frequency band. Therefore, within the actual band of interest, the noise power is considerably lower. A digital filter is also included to remove the out-ofband noise.

A 12-MHz crystal supplies the clocking to the AIC23 codec (as well as to the DSP and the USB interface). Using this 12-MHz master clock, with oversampling rates of $250F_s$ and $272F_s$, an exact audio sample rate of 48 kHz (12MHz/250) and a CD rate of 44.1kHz (12MHz/272) can be obtained. The sampling rate is set by the codec's register SAMPLERATE.

The ADC converts an input signal into discrete output digital words in a 2'scomplement format that corresponds to the analog signal value. The DAC includes an

interpolation filter and a digital modulator. A decimation filter reduces the digital data rate to the sampling rate.The DAC's output is first passed through an internal lowpass reconstruction filter to produce an output analog signal. Low noise performance for both ADC and DAC is achieved using oversampling techniques with noise shaping provided by sigma–delta modulators.

Communication with the AIC23 codec for input and output uses two multichannel buffered serial ports McBSPs on the C6713. McBSP0 is used as a unidirectional channel to send a 16-bit control word to the AIC23. McBSP1 is used as a bidirectional channel to send and receive audio data.

Alternative I/O daughter cards can be used for input and output. Such cards can plug into the DSK through the external peripheral interface 80-pin connector J3 on the DSK board.

3.3 PROGRAMMING EXAMPLES USING C CODE

Several examples follow to illustrate input and output with the DSK. They are included to familiarize you with both the hardware and software tools and provide some background to implement a specific application. The example *sine2sliders* illustrates the use of two sliders, an echo example demonstrates the effects of a variable-length buffer on an echo, a noise generator example is used in Chapter 4 as the input to a digital filter, an example illustrates the use of onboard flash memory, and so on.
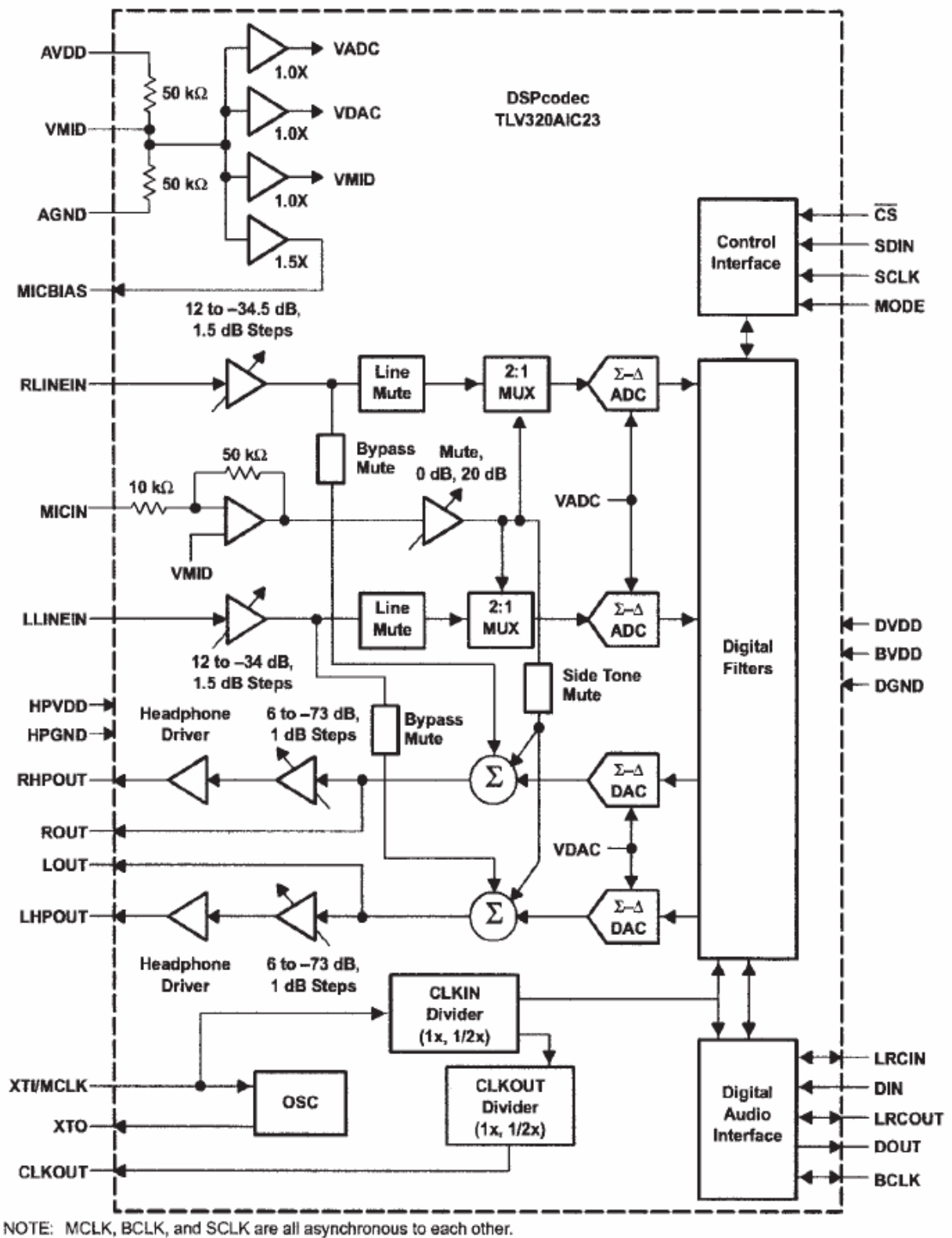
FIGURE 2.3. TLV320AIC23 codec block diagram (Courtesy of Texas Instruments).

NOTE: MCLK, BCLK, and SCLK are all asynchronous to each other.

31

Example 3.1: Loop Program Using Interrupt (loop_intr)

This example illustrates input and output with the AIC23 codec. Figure 2.4 shows the C source program *loop_intr.c*, which implements the loop program. It is interrupt-driven using INT11.

This program example is very important since it can be used as a base program to build on. For example, to implement a digital filter, one would need to insert the appropriate algorithm between the input and output functions. The two functions input_sample and output_sample, as well as the function comm_intr, are included in the communication support file *C6713dskinit.c*.This is done so that the C source program is kept as small as possible.

After the initialization and selection/enabling of an interrupt, execution waits within the infinite while loop until an interrupt occurs. Upon interrupt, execution proceeds to the ISR c_int11, as specified in the vector file vectors_intr.asm. An interrupt occurs every sample period $T_s = 1/F_s = 1/(8 \text{ kHz}) = 0.125\text{ms}$, at which time an input sample value is read from the codec's ADC and then sent as output to the codec's DAC.

Execution returns from interrupt to the while(1) statement waiting for a subsequent interrupt. [Note that in lieu of waiting within the while(1) infinite loop, one could be processing code.] Upon interrupt, execution proceeds to ISR, "services" the necessary task dictated by ISR, then returns to the calling function waiting for the occurrence of a subsequent interrupt.

```
//Loop_intr.c Loop program using interrupt.Output=delayed input
#include "dsk6713_aic23.h" //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

interrupt void c_int11() //interrupt service routine
{
```

```
short sample_data;

sample_data = input_sample(); //input data
output_sample(sample_data); //output data
return;
}

void main()
{
comm_intr(); //init DSK, codec, McBSP
while(1); //infinite loop
}
```
FIGURE 2.4. Loop program using interrupt (loop_intr.c).

1. Within the function *output_sample*, support functions from the BSL are
included to write data using the two serial ports: McBSP0 for control and
McBSP1 for data transfer (*MCBSP_write*). Most of the programs in the book
will output using 16 bits. In this fashion, output_sample is made to default
to the left 16-bit channel and no adapter need be used (see the comments in
C6713dskinit.c). Otherwise, one would need to use output_right_sample.

2. Within the function *comm_intr*, the following tasks are performed.

  (a) Initialize the DSK.

  (b) Configure/select INT11.

  (c) Enable the specific interrupt.

  (d) Enable the global enable interrupt (GIE) bit and the nonmaskable
interrupt.

  (e) Initiate communication.

The interrupt functions called for the tasks above are within the board and chip support files included with CCS. Create and build this project as **loop_intr**. The main C source file is in the folder *loop_intr*. Use the same support files as in Example 1.2: the vector file for the interrupt-driven and linker command file located in the folder *support*, and the runtime support, board support, and chip support library files that can be added with the building option for the linker.

Input a sinusoidal waveform to the LINE IN connector on the DSK, with amplitude of approximately 2 V p-p and a frequency between approximately 1 and 3 kHz. Connect the output of the DSK, LINE OUT to a speaker or to an oscilloscope and verify a tone of the same input frequency, but attenuated to approximately 0.8 V p-p. Using an oscilloscope, the output is a delayed version of the input signal.

Increase the amplitude of the input sinusoidal waveform beyond 6V p-p and observe that the output signal becomes distorted.

Input with Gain

To adjust the gain of the left line-input channel, the corresponding header support file *c6713dskinit.h* of the communication/init "black box" file needs to be modified slightly. First, copy this header file AND *c6713dskinit.c from* the *support* folder into the folder *loop_intr* so that you do not modify the original header file. Remove the init file from the project and replace it with the one in the folder *loop_intr*.This will keep the original init support file unchanged in the folder *support*. Modify the setup register 0, which controls the left input volume, from 0x0017 to 0x001c in order to increase the left line-input volume.

Rebuild the project, making sure that you are adding *c6713dskinit.c* from the folder *loop_intr* (and not from the folder *support*). In this fashion, the corresponding header file *c6713dskinit.h* that will be included will come from that same folder. Load/run the executable file *loop_intr.out*, and verify that the output amplitude is not attenuated and is

the same as the input amplitude of 2V p-p. Values for the set-up register 0 from 0x0018 to 0x001c will cause the output amplitude to increase from 0.8 to 2V p-p.

The left input channel was selected since *input_sample* and *output_sample* default to the left channel. Otherwise, if the right line-input volume is to be increased by modifying the set-up register 1, an adapter/connector with two inputs and one single-ended output connections would be needed. See Example 2.3 (*loop_stereo/ sine_stereo*).

Input from a Microphone

To select an input from a microphone in lieu of line input, modify the header file set-up register 4 from 0x0011 to 0x0015 (third LSB as a 1) so that the ADC gets its input from MIC IN. The microphone input and line input are multiplexed, and only one is active at a time. Rebuild the project to verify your output, with the input to the MIC IN connector.

Example 3.2: Loop Program Using Polling (loop_poll)

This example implements a polling-based loop program to illustrate the input and output of a sample value every sample period *Ts*. Note that the program loop_intr.c in Example 2.1 is an interrupt-driven program. The C source program loop_poll.c shown in Figure 2.5 implements this loop program. The polling technique uses a continuous procedure of testing when the data are ready. Although it is simpler than the interrupt technique, it is less efficient since the input and output data need to be continuously tested to determine when they are ready to be received or transmitted.

//**loop_poll.c** Loop program using polling.Output=delayed input

```
#include "DSK6713_AIC23.h" //codec-DSK file support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

void main()
```

```
{
short sample_data;
comm_poll(); //init DSK, codec, McBSP
while(1) //infinite loop
{
sample_data = input_sample(); //input sample
output_sample(sample_data); //output sample
}
}
```
FIGURE 2.5. Loop program using polling (loop_poll.c).

1. The input to the ADC is from the data receive register (DRR) of the McBSP1.
Since this is a polling-driven program, the SPCR bit 1, which is the receive
ready register (RRDY), is first tested to determine if it is a 1 or enabled (see
Figure B.8).Within input_sample, execution of the statement

   While (!MCBSP_rrdy())

remains in an infinite loop until RRDY becomes 1 or enabled. Execution then
proceeds to read/receive the data.

2. Within the function output_sample, the MCBSP1 *writes* the output from
the DAC to the data transmit register (DXR) of McBSP1. Since this is a
polling-driven program, the transmit ready register (XRDY) bit 17 of SPCR
is first tested to see if it is a 1 or enabled. Within output_sample, execution
of the statement

   While (!MCBSP_xrdy())

remains in an infinite loop until the transmit ready register becomes 1 or
enabled. Execution then proceeds to transmit/write the data.

The same support files as in Example 1.1 are used: the "black box" communication/ init file *c6713dskinit.c*, the vector file vectors_poll.asm, the linker command file c6713dsk.cmd (all three from the folder support), and the three library-support files.

Create and build this project as **loop_poll**. Use the same input as in Example 2.1 and verify the same results.

Example 3.3: Stereo Input and Stereo Output (loop_stereo/sine_stereo)

Loop Program with Stereo Input and Stereo Output (loop_stereo)

This example demonstrates input and output using the stereo capability of the onboard AIC23 codec. It requires the use of an adapter with two inputs and one output that connects to the DSK. Such an adapter has one input connector white (or silver) that represents the left channel and another input connector red (or gold) that represents the right channel. This adapter becomes essential for some of the examples on adaptive filtering that require two separate input signals, processing each input separately. Figure 2.6 shows the loop program *loop_stereo* to illustrate.

//**Loop_stereo.c** Stereo input and output with both channels

```
#include "dsk6713_aic23.h" //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

#define LEFT 0 //reversed in init file
#define RIGHT 1
union {Uint32 combo; short channel[2];} AIC23_data;

interrupt void c_int11() //interrupt service routine
{
AIC23_data.combo = input_sample(); //input 32-bit sample
output_left_sample(AIC23_data.channel[LEFT]);//left channels for I/O
return;
}
void main() //main function
{
comm_intr(); //init DSK, codec, McBSP
while(1); //infinite loop
}
```

FIGURE 2.6. Loop program with stereo input and output (loop_stereo.c).

Within the function input_sample, support functions from the BSL are included to read a 32-bit data. The function *input_sample* captures 32-bit data, 16 bits from the left input channel and 16 bits from the right input channel.The *union* statement is used to process each channel independently.The union of *AIC23_data* and *combo* contains these 32-bit input data.The line of code for output is from the left channel (by default) to output 16-bit data from the left input channel.

Build and run this project as **loop_stereo** using the support files as in Example 1.2 for an interrupt-driven program. The main C source file *loop_stereo.c* is contained in the folder loop_stereo. Connect a 1 kHz (with approximate amplitude of 2 V p-p) sine wave into the left input channel and a 2-kHz sine wave into the right input channel. Verify that the left (default) output channel has the same input signal frequency of 1kHz, but reduced in amplitude (as expected).You do not need a second adapter for the output side since the output defaults to the left channel. Change the output line of code to

output_left_sample(AIC23_data.channel[RIGHT]);

and verify that the output is the 2-KHz sine wave from the right input channel.With the line of code
output_right_sample(AIC23_data.channel[RIGHT]);

//**Sine_stereo.c** Sine generation with output to both channels

#include "dsk6713_aic23.h" //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

#define LEFT 0 //reversed in init file
#define RIGHT 1
union {Uint32 combo; short channel[2];} AIC23_data;

```
short loop = 0, gain = 10;
short sine_table[8]={0,707,1000,707,0,-707,-1000,-707};//sine values

interrupt void c_int11() //interrupt service routine
{
AIC23_data.channel[RIGHT]=sine_table[loop]*gain; //for right channel
AIC23_data.channel[LEFT]=sine_table[loop]*gain; //for left channel
output_sample(AIC23_data.combo); //output to both channels

if (++loop > 7) loop = 0; //reint index if @ end of table
}
void main()
{
comm_intr(); //init DSK, codec, McBSP
while(1); //infinite loop
}
```

FIGURE 2.7. Sine generation with stereo outputs (sine_stereo.c).

two adapters are required to verify that the output from the right channel is the 2-kHz sine wave from the right input channel. You can also use one adapter at the input side to capture the two different signals and one stereo cable at the output side.

Experiment with this project, inputting different signals into each channel and outputting from each channel using adapters and stereo cable. Verify that you can select each input and output channel independently.

Sine Generation with Stereo Output (sine_stereo)

Figure 2.7 shows the C source file *sine_stereo.c*, included in the folder *sine_stereo*, to illustrate further the codec as a stereo device. Build and run this project as **sine_stereo**. Verify that the generated 1 kHz sinusoid is through both output channels, using an adapter or stereo cable at the output side of the DSK.

Example 3.4: Sine Generation with Two Sliders for Amplitude and Frequency Control (sine2sliders)

The polling-based program *sine2sliders.c* in Figure 2.8 generates a sine wave. Two sliders are used to vary both the amplitude (gain) and the frequency of the sinusoid

//**Sine2sliders.c** Sine generation with different # of points

```
#include "DSK6713_AIC23.h" //codec-DSK interface support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
short loop = 0;
short sine_table[32]={0,195,383,556,707,831,924,981,1000,
981,924,831,707,556,383,195,0,-195,-383,-556,-707,-831,-924,
-981,-1000,-981,-924,-831,-707,-556,-383,-195}; //sine data
short gain = 1; //for gain slider
short frequency = 2; //for frequency slider
void main()
{
comm_poll(); //init DSK,codec,McBSP
while(1) //infinite loop
{
output_sample(sine_table[loop]*gain);//output scaled value
loop += frequency; //incr frequency index
loop = loop % 32; //modulo 32 to reinit index
}
}
```

FIGURE 2.8. Sine generation making use of two sliders to control the amplitude and frequency of the sine wave generated (sine2sliders.c).

/***Sine2sliders.gel** Two sliders to vary gain and frequency*/

menuitem "Sine Parameters"

slider Gain(1,8,1,1,gain_parameter) /*incr by 1,up to 8*/

{

gain = gain_parameter; /*vary gain*/

}

slider Frequency(2,8,2,2,frequency_parameter) /*incr by 2,up to 8*/

{

frequency = frequency_parameter; /*vary frequency*/

}

FIGURE 2.9. GEL file with two slider functions to control the amplitude and frequency of the sine wave generated (sine2sliders.gel).

generated. Using a lookup table with 32 points, the variable frequency is obtained by selecting a different number of points per cycle. The gain slider scales the volume/amplitude of the waveform signal. The appropriate GEL file *sine2sliders.gel* is shown in Figure 2.9.

The 32 sine data values in the table or buffer correspond to $\sin(t)$, where $t = 0,11.25$, 22.5, 33.75, 45, . . . , 348.75 degrees (scaled by 1000). The frequency slider takes on a value from 2 to 8, incremented by 2. The modulo operator is used to test when the end of the buffer that contains the sine data values is reached. When the loop index reaches 32, it is reinitialized to zero. For example, with the frequency slider at position 2, the loop or frequency index steps through every other value in the table. This corresponds to 16 data values within one cycle.

Build this project as **sine2sliders**. Use the appropriate support files for a pollingdriven program. The main C source file *sine2sliders.c* is contained in the folder **sine2sliders**. Verify that the frequency generated is $f = Fs/16 = 500$Hz. Increase the slider position (the use of a slider was introduced in Example 1.1) to 4, 6, 8 and verify that the signal

frequencies generated are 1000, 1500, and 2000Hz, respectively. Note that when the slider is at position 4, the loop or frequency index steps through the table selecting the eight values (per cycle): sin[0], sin[4], sin[8], . . . , sin[28] that correspond to the data values 0, 707, 1000, 707, 0, -707, -1000, and -707.The resulting frequency generated is then $f = Fs/8 = 1$ kHz (as in Example 1.1).

Example 3.5: Loop Program with Input Data Stored in Memory (loop_store)

The program *loop_store.c* in Figure 2.10 is an interrupt-based program and is included in the folder **loop_store**. Each time an interrupt INT11 occurs, a sample is read from the codec's ADC and written to the codec's DAC. Furthermore, each sample is written to a 512-element circular buffer implemented using an array *buffer* and an index i that is incremented after each sample is stored. The index is reset to zero when it reaches the end of the buffer. Consequently, the array always contains the 512 most recent sample values.

//**Loop_store.c** Data acquisition.Input data stored also into buffer

```
#include "DSK6713_AIC23.h" //codec-DSK interface support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
#define BUFFER_SIZE 512 //buffer size
short buffer[BUFFER_SIZE]; //buffer where data is stored
int i = 0;
interrupt void c_int11() //interrupt service routine
{
output_sample((short)input_sample());//output acquired data
buffer[i] =((short)input_sample());//store input data into buffer
i++; //increment buffer index
if (i==BUFFER_SIZE) i = 0; //reinit index if buffer full
return; //return from ISR
```

```
}
void main()
{
comm_intr(); //init DSK, codec, McBSP
while(1); //infinite loop
}
```

FIGURE 2.10. Loop program with input data stored in memory (loop_store.c).
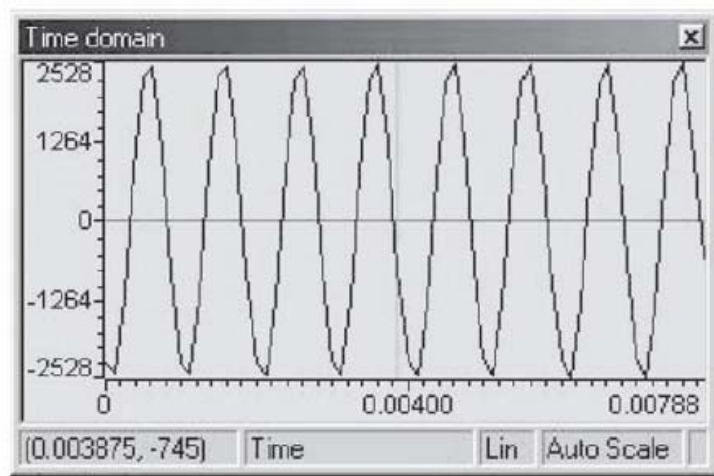
(*a*)
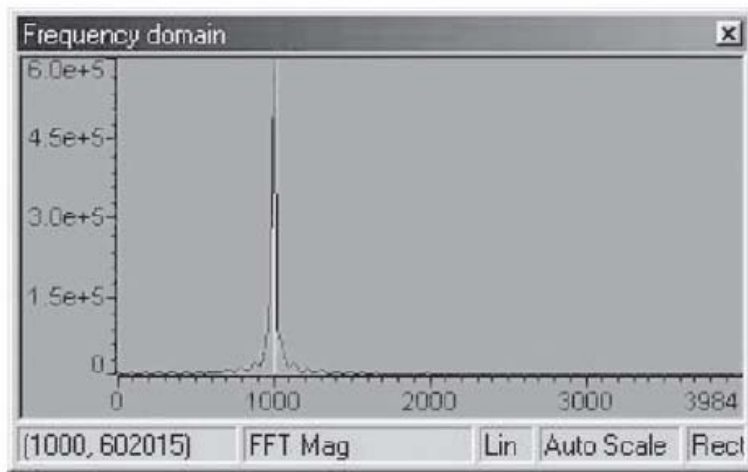
(*b*)

FIGURE 2.11. CCS graphs with the *loop_store* program: (*a*) time-domain plot of stored input data representing a 1-kHz sine wave; (*b*) FFT magnitude of stored data representing a 1-kHz sine wave.

Build this project as **loop_store**. Input a sinusoidal signal with amplitude of approximately 1/2 V p-p and a frequency of 1 kHz. Run and verify your output results.

Use CCS to plot the stored input data in both the time and frequency domains (see also Example 1.2). Select View Æ Graph Æ Time/Frequency. For the timedomain plot, specify a starting address "buffer," 512 points for the acquisition buffer size, 64 points for the data size display (for a clearer plot), a 16-bit signed integer for the data type, and 8000 for the sampling rate.Verify the 1-kHz time-domain sinewave plot within CCS, as shown in Figure 2.11*a*.

Select View Æ Graph Æ Time/Frequency again and FFT magnitude for display to obtain a frequency-domain plot of the stored input data. Specify a display data size of 512 with an FFT order of $M = 9$, where $2M = 512$.The spike at 1 kHz in Figure 2.11*b* represents the 1-kHz sine wave plot within CCS.

//**Loop_print.c** Data acquisition.Loop with data printed to a file

```
#include "DSK6713_AIC23.h" //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
#include <stdio.h>
#define BUFFER_SIZE 64 //buffer size
int i=0, j=0;
short buffer[BUFFER_SIZE]; //buffer for data
FILE *fptr; //file pointer
interrupt void c_int11() //ISR
{
buffer[i]=((short)input_sample()); //store data in buffer
```

```c
i++; //increment buffer count
if (i==BUFFER_SIZE - 1) //if buffer full
{
fptr = fopen("sine.dat","w"); //create output data file
for (j=0; j<BUFFER_SIZE; j++)
fprintf(fptr,"%d\n",buffer[j]);//write buffer data to file
fclose(fptr); //close file
i = 0; //initialize buffer count
puts("done"); //finished storing to file
}
output_sample((short)input_sample()); //output data
return; //return from ISR
}
void main()
{
comm_intr(); //init DSK, codec, McBSP
puts("start\n"); //print "start" indicator
while(1); //infinite loop
}
```

FIGURE 2.12. Loop program to store I/O data in memory and in a file (loop_print.c).

Example 3.6: Loop with Data in a Buffer Printed to a File (loop_print)

This example extends the preceding loop program so that the acquired input data are stored in a memory buffer and then printed to a file. Figure 2.12 shows the C source program *loop_print.c* (included in the folder **loop_print**) that implements this example. It takes a long time (more than 3000 cycles) to execute the printf statement in the program (see Example 1.3).This can be reduced to about 30 cycles using DSP/BIOS.

After initialization of the DSK, the *puts* statement prints the word start as an indicator within the CCS command window; then execution proceeds to the infinite while loop. Upon each interrupt, execution proceeds to ISR, and a newly acquired data value is stored in a buffer of size 64.

The buffer index i is incremented to store each new sampled data value. When the end of the buffer is reached, indicating that the buffer is full, a file sine.dat is "opened" and the contents of the buffer are written into that file. Then the indicator done is printed within the CCS window. This process is repeated continuously so that a new set of 64 data points is acquired, and the done indicator is again displayed (after each set of data fills the buffer and is written to sine.dat).

Build and run this project as ***loop_print***. Input a sine-wave signal of approximately 1/2 V p-p with a 1-kHz frequency. Halt execution after the indicator done is displayed. The buffer of 64 input data representing the sine wave can be retrieved from the file *sine.dat* in the same folder loop_print\Debug. Note that the third set of 64 points will be stored in the buffer and printed in the file *sine.dat* if execution of the program is halted after the third done indicator. A plot program or MATLAB can be used to plot sine.*dat* and verify a 1-kHz sine wave.You can also verify your results by plotting the content of the buffer within CCS, as in the previous example. Note that the output is not displayed appropriately in real time due to the slow execution of the print statement. You can comment the section of code that is associated with printing the input data into a file to verify that a loop program is also implemented.

Example 3.7: Square-Wave Generation Using a Lookup Table (squarewave)

This example generates a square wave using a lookup table. Figure 2.13 shows a listing of the program *squarewave.c* (located in the folder **squarewave**) that implements this project example. A buffer of size 64 is created.Within *main*, the buffer table is loaded with data: the first half with (215 - 1) = 32,767 and the second half with -215 = -32,768. Upon each interrupt that occurs every sample period T_s, one data value from the buffer is

sent for output. After each data value from the. table is output, execution returns to the infinite while loop, waiting for the next interrupt to occur and output the subsequent value in the table. When the end of the buffer (table) is reached, the buffer index is reinitialized to the beginning of the buffer.

Build and run this project as **squarewave**. Verify a square-wave output signal of approximately 3V p-p. Note that the valid input data to the codec are between -215 and (215 - 1) or between -32,768 and 32,767. Change the values in the first half of the table using 0x8000 = 32,768 in lieu of 0x7FFF = 32,767. Rebuild/run and verify that the square-wave signal is no longer generated.

Note that increasing the number of points in the table produces a more pronounced charging/discharging effect (since it is AC coupled) due to the output capacitor (see the block diagram of the AIC23 codec). For example, with 64 points, the fundamental frequency is at 8 kHz/64 = 125Hz. Doubling the number of points will double the period of the square wave, and the discharging effect will be more pronounced (time constant reduced relative to one-half of the period of the square wave). Change the sampling frequency to 16 or 24 kHz and verify that the charching/ discharging effect of the capacitor is less pronounced.

//**Squarewave.c** Generates a squarewave using a look-up table

```
#include "dsk6713_aic23.h" //codec-DSK interface support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
#define table_size (int)0x40 //size of table=64
short data_table[table_size]; //data table array
int i;
interrupt void c_int11() //interrupt service routine
{
output_sample(data_table[i]); //output value each Ts
if (i < table_size) ++i; //if table size is reached
```

```
else i = 0; //reinitialize counter

return; //return from interrupt

}

main()

{

for(i=0; i<table_size/2; i++) //set 1st half of buffer

data_table[i] = 0x7FFF; //with max value (2^15)-1

for(i=table_size/2;i<table_size;i++) //set 2nd half of buffer

data_table[i] = -0x8000; //with -(2^15)

i = 0; //reinit counter

comm_intr(); //init DSK, codec, McBSP

while (1); //infinite loop

}
```

FIGURE 2.13. Square-wave generation program (squarewave.c).

CHAPTER-4

ARCHITECTURE AND INSTRUCTION SET OF THE C6713 PROCESSOR

4.1 INTRODUCTION

Texas Instruments introduced the first-generation TMS32010 DSP in 1982, the TMS320C25 in 1986, and the TMS320C50 in 1991. Several versions of each of these processors—C1x, C2x, and C5x—are available with different features, such as faster execution speed. These 16-bit processors are all fixed-point processors and are code-compatible.

In a von Neumann architecture, program instructions and data are stored in a single memory space. A processor with a von Neumann architecture can make a read or a write to memory during each instruction cycle. Typical DSP applications require several accesses to memory within one instruction cycle. The fixed-point processors C1x, C2x, and C5x are based on a modified Harvard architecture with separate memory spaces for data and instructions that allow concurrent accesses.

Quantization error or round-off noise from an ADC is a concern with a fixedpoint processor. An ADC uses only a best-estimate digital value to represent an input. For example, consider an ADC with a word length of 8 bits and an input range of ±1.5V.The steps represented by the ADC are: input range/28 = 3/256 = 11.72mV. This produces errors that can be up to $\pm(11.72\text{mV})/2 = \pm5.86$mV. Only a best estimate can be used by the ADC to represent input values that are not multiples of 11.72mV.With an 8-bit ADC, 28 or 256 different levels can represent the input signal. An ADC with a larger word length, such as a 16-bit ADC (or larger, currently very common), can reduce the quantization error, yielding a higher resolution.The morebits an ADC has, the better it can represent an input signal.

The TMS320C30 floating-point processor was introduced in the late 1980s. The C31, the C32, and the more recent C33 are all members of the C3x family of floating-point processors. The C4x floating-point processors, introduced subsequently , are code-compatible with the C3x processors and are based on the modified Harvard architecture. The C62x is not code-compatible with the previous generation of fixed-point processors. Subsequently, the TMS320C6701 (C67x) floating-point processor was introduced as

another member of the C6x family of processors. The instruction set of the C62x fixed-point processor is a subset of the instruction set of the C67x processor. Appendix A contains a list of instructions available on the C6x processors.

An application-specific integrated circuit (ASIC) has a DSP core with customized circuitry for a specific application. A C6x processor can be used as a standard general-purpose DSp programmed for a specific application. Specific-purpose digital signal processors are the modem, echo canceler, and others.

A fixed-point processor is better for devices that use batteries, such as cellular phones, since it uses less power than does an equivalent floating-point processor. The fixed-point processors, C1x, C2x, and C5x, are 16-bit processors with limited dynamic range and precision. The C6x fixed-point processor is a 32-bit processor with improved dynamic range and precision. In a fixed-point processor, it is necessary to scale the data. Overflow, which occurs when an operation such as the addition of two numbers produces a result with more bits than can fit within a processor's register, becomes a concern.

A floating-point processor is generally more expensive since it has more "real estate" or is a larger chip because of additional circuitry necessary to handle integer as well as floating-point arithmetic. Several factors, such as cost, power consumption, and speed, come into play when choosing a specific DSp. The C6x processors are particularly useful for applications requiring intensive computations. Family members of the C6x include both fixed-point (e.g., C62x, C64x) and floating-point (e.g., C67x) processors. Other DSp's are also available from companies such as Motorola and Analog Devices.

Other architectures include the Super Scalar, which requires special hardware to determine which instructions are executed in parallel. The burden is then on the processor more than on the programmer, as in the VLIW architecture. It does not necessarily execute the same group of instructions, and as a result, it is difficult to time. Thus, it is rarely used in DSP.

4.2 TMS320C6x ARCHITECTURE

The TMS320C6713 onboard the DSK is a floating-point processor based on the VLIW architecture. Internal memory includes a two-level cache architecture with 4 kB of level 1 program cache (L1P), 4 kB of level 1 data cache (L1D), and 256 kB of level 2 memory shared between program and data space. It has a glueless (direct) interface to both synchronous memories (SDRAM and SBSRAM) and asynchronous memories (SRAM and EPROM). Synchronous memory requires clocking but provides a compromise between static SRAM and dynamic DRAM, with SRAM being faster but more expensive than DRAM.

On-chip peripherals include two McBSPs, two timers, a host port interface (HPI), and a 32-bit EMIF. It requires 3.3 V for I/O and 1.26 V for the core (internal). Internal buses include a 32-bit program address bus, a 256-bit program data bus to accommodate eight 32-bit instructions, two 32-bit data address buses, two 64-bit data buses, and two 64-bit store data buses. With a 32-bit address bus, the total memory space is 232 = 4GB, including four external memory spaces: CE0, CE1, CE2, and CE3. Figure 3.1 shows a functional block diagram of the C6713 processor included with CCS.

Independent memory banks on the C6x allow for two memory accesses within one instruction cycle. Two independent memory banks can be accessed using two independent buses. Since internal memory is organized into memory banks, two loads or two stores of instructions can be performed in parallel. No conflict results if the data accessed are in different memory banks. Separate buses for program, data, and direct memory access (DMA) allow the C6x to perform concurrent program fetches, data read and write, and DMA operations. With data and instructions residing in separate memory spaces, concurrent memory accesses are possible. The C6x has a byte-addressable memory space. Internal memory is organized as separate program and data memory spaces, with two 32-bit internal ports (two 64- bit ports with the C64x) to access internal memory.

The C6713 on the DSK includes 264kB of internal memory, which starts at 0x00000000, and 16MB of external SDRAM, mapped through CE0 starting at 0x80000000.The DSK also includes 512 kB of Flash memory (256 kB readily available to the user), mapped through CE1 starting at 0x90000000. Figure 3.2 shows the L2 internal memory configuration, included with CCS. Table 3.1 shows the memory map, also included with CCS.

With the DSK operating at 225MHz, one can ideally achieve two multiplies and accumulates per cycle, for a total of 450 million multiplies and accumulates (MACs) per second. With six of the eight functional units in Figure 3.1 (not the .D units described below) capable of handling floating-point operations, it is possible to perform 1350 million floating-point operations per second (MFLOPS). Operating at 225MHz, this translates into 1800 million instructions per second (MIPS) with a 4.44-ns instruction cycle time.

4.3 FUNCTIONAL UNITS

The CPU consists of eight independent functional units divided into two data paths, A and B, as shown in Figure 3.1. Each path has a unit for multiply operations (.M), for logical and arithmetic operations (.L), for branch, bit manipulation, and arithmetic operations (.S), and for loading/storing and arithmetic operations (.D). The .S and .L units are for arithmetic, logical, and branch instructions. All data transfers make use of the .D units.

The arithmetic operations, such as subtract or add (SUB or ADD), can be performed by all the units, except the .M units (one from each data path). The eight functional units consist of four floating/fixed-point ALUs (two .L and two .S), two fixed-point ALUs (.D units), and two floating/fixed-point multipliers (.M units). Each functional unit can read directly from or write directly to the register file within its own path. Each path includes a set of sixteen 32-bit registers, A0 through A15 and B0 through B15. Units ending in 1 write to register file A, and units ending in 2 write to register file B.
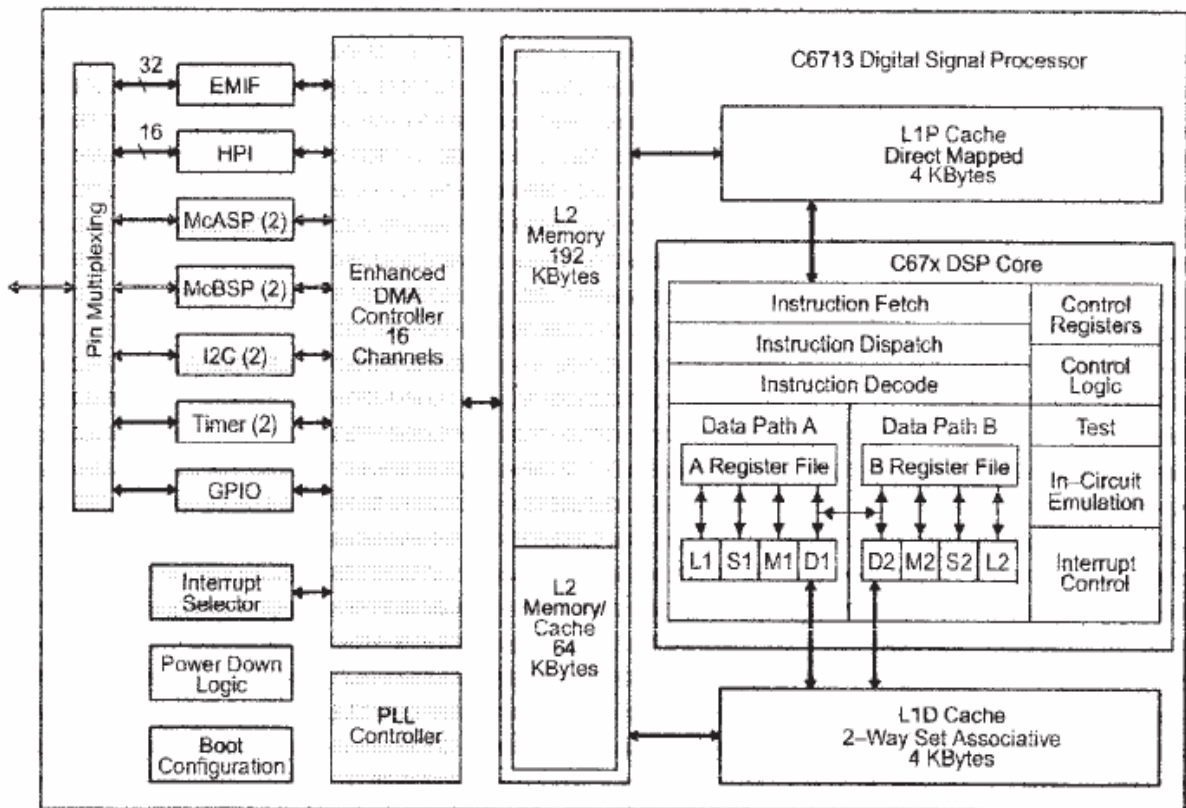
FIGURE 3.1. Functional block diagram of TMS320C6713 (Courtesy of Texas Instruments).
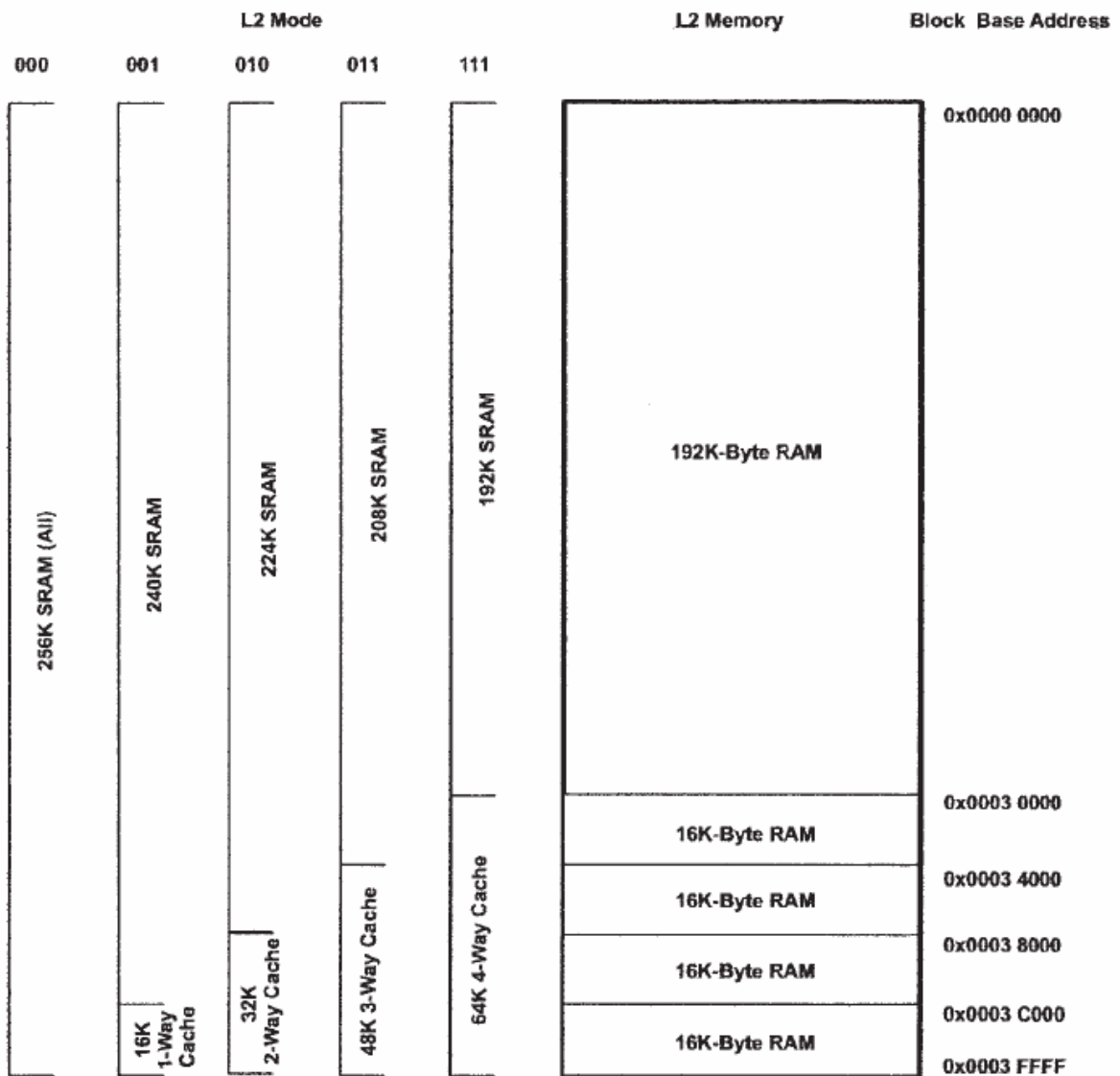
FIGURE 3.2. Internal memory configuration of L2 (Courtesy of Texas Instruments).

TABLE 3.1 Memory Map

Memory Block Description	Block Size (Bytes)	Hex Address Range
Internal RAM (L2)	192 K	0000 0000–0002 FFFF
Internal RAM/cache	64 K	0003 0000–0003 FFFF
Reserved	24 M–256 K	0004 0000–017F FFFF
External memory interface (EMIF) registers	256 K	0180 0000–0183 FFFF
L2 registers	128 K	0184 0000–0185 FFFF
Reserved	128 K	0186 0000–0187 FFFF
HPI registers	256 K	0188 0000–018B FFFF
McBSP 0 registers	256 K	018C 0000–018F FFFF
McBSP 1 registers	256 K	0190 0000–0193 FFFF
Timer 0 registers	256 K	0194 0000–0197 FFFF
Timer 1 registers	256 K	0198 0000–019B FFFF
Interrupt selector registers	512	019C 0000–019C 01FF
Device configuration registers	4	019C 0200–019C 0203
Reserved	256 K–516	091C 0204–019F FFFF
EDMA RAM and EDMA registers	256 K	01A0 0000–01A3 FFFF
Reserved	768 K	01A4 0000–01AF FFFF
GPIO registers	16 K	01B0 0000–01B0 3FFF
Reserved	240 K	01B0 4000–01B3 FFFF
I2C0 registers	16 K	01B4 0000–01B4 3FFF
I2C1 registers	16 K	01B4 4000–01B4 7FFF
Reserved	16 K	01B4 8000–01B4 BFFF
McASP0 registers	16 K	01B4 C000–01B4 FFFF
McASP1 registers	16 K	01B5 0000–01B5 3FFF
Reserved	160 K	01B5 4000–01B7 BFFF
PLL registers	8 K	01B7 C000–01B7 DFFF
Reserved	264 K	01B7 E000–01BB FFFF
Emulation registers	256 K	01BC 0000–01BF FFFF
Reserved	4 M	01C0 0000–01FF FFFF
QDMA registers	52	0200 0000–0200 0033
Reserved	16 M–52	0200 0034–02FF FFFF
Reserved	720 M	0300 0000–2FFF FFFF
McBSP0 data port	64 M	3000 0000–33FF FFFF
McBSP1 data port	64 M	3400 0000–37FF FFFF
Reserved	64 M	3800 0000–3BFF FFFF
McASP0 data port	1 M	3C00 0000–3C0F FFFF
McASP1 data port	1 M	3C10 0000–3C1F FFFF
Reserved	1 G + 62 M	3C20 0000–7FFF FFFF
EMIF CE0*	256 M	8000 0000–8FFF FFFF
EMIF CE1*	256 M	9000 0000–9FFF FFFF
EMIF CE2*	256 M	A000 0000–AFFF FFFF
EMIF CE3*	256 M	B000 0000–BFFF FFFF
Reserved	1 G	C000 0000–FFFF FFFF

* The number of EMIF address pins (EA[21:2]) limits the maximum addressable memory (SDRAM) to 128 MB per CE space.

Source: Courtesy of Texas Instruments.

Two cross-paths (1x and 2x) allow functional units from one data path to access a 32-bit operand from the register file on the opposite side. There can be a maximum of two cross-path source reads per cycle. Each functional unit side can access data from the registers on the opposite side using a cross-path (i.e., the functional units on one side can access the register set from the other side). There are 32 generalpurpose registers, but some of them are reserved for specific addressing or are used for conditional instructions.
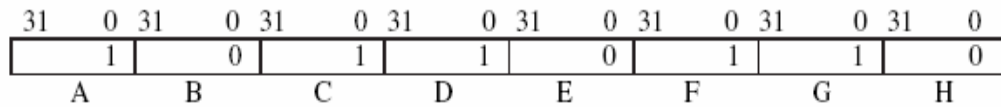
FIGURE 3.3. One FP with three EPs showing the "p" bit of each instruction.

4.4 FETCH AND EXECUTE PACKETS

The architecture VELOCITI, introduced by TI, is derived from the VLIW architecture. An execute packet (EP) consists of a group of instructions that can be executed in parallel within the same cycle time. The number of EPs within a fetch packet (FP) can vary from one (with eight parallel instructions) to eight (with no parallel instructions). The VLIW architecture was modified to allow more than one EP to be included within an FP.

The least significant bit of every 32-bit instruction is used to determine if the next or subsequent instruction belongs in the same EP (if 1) or is part of the next EP (if 0). Consider an FP with three EPs: EP1, with two parallel instructions, and EP2 and EP3, each with three parallel instructions, as follows:

Instruction A

|| Instruction B

Instruction C

|| Instruction D

57
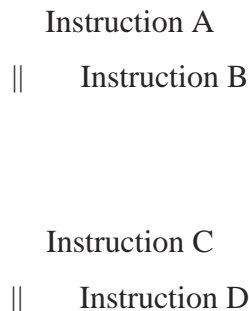
|| Instruction E

Instruction F
|| Instruction G
|| Instruction H

EP1 contains the two parallel instructions A and B; EP2 contains the three parallel instructions C, D, and E; and EP3 contains the three parallel instructions F, G, and H. The FP would be as shown in Figure 3.3. Bit 0 (LSB) of each 32-bit instruction contains a "p" bit that signals whether it is in parallel with a subsequent instruction. For example, the "p" bit of instruction B is zero, denoting that it is not within the same EP as the subsequent instruction C. Similarly, instruction E is not within the same EP as instruction F.

4.5 PIPELINING

Pipelining is a key feature in a DSp to get parallel instructions working properly, requiring careful timing. There are three stages of pipelining: program fetch, decode, and execute.

1. The *program fetch stage* is composed of four phases:
    (a) *PG*: program address generate (in the CPU) to fetch an address
    (b) *PS*: program address send (to memory) to send the address
    (c) *PW*: program address ready wait (memory read) to wait for data
    (d) *PR*: program fetch packet receive (at the CPU) to read opcode from memory
2. The *decode stage* is composed of two phases:
    (a) *DP*: to dispatch all the instructions within an FP to the appropriate functional units.
    (b) *DC*: instruction decode
3. The *execute stage* is composed of 6 phases (with fixed point) to 10 phases (with floating point) due to delays (latencies) associated with the following functional units.
    (a) Multiply instruction, which consists of two phases due to one delay

(b) Load instruction, which consists of five phases due to four delays

(c) Branch instruction, which consists of six phases due to five delays

Table 3.2 shows the pipeline phases, and Table 3.3 shows the pipelining effects. The first row in Table 3.3 represents cycle 1, 2, . . . , 12. Each subsequent row represents an FP. The rows represented PG, PS, . . . illustrate the phases associated with each FP. The program generate (PG) of the first FP starts in cycle 1, and the PG of the second FP starts in cycle 2, and so on. Each FP takes four phases for program fetch and two phases for decoding. However, the execution phase can take from 1 to 10 phases (not all execution phases are shown in Table 3.3).We are assuming that each FP contains one EP.

TABLE 3.2   Pipeline Phases

Program Fetch				Decode		Execute
PG	PS	PW	PR	DP	DC	E1–E6 (E1–E10 for double precision)

TABLE 3.3   Pipelining Effects

					Clock Cycle						
1	2	3	4	5	6	7	8	9	10	11	12
PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
			PG	PS	PW	PR	DP	DC	E1	E2	E3
				PG	PS	PW	PR	DP	DC	E1	E2
					PG	PS	PW	PR	DP	DC	E1
						PG	PS	PW	PR	DP	DC

For example, at cycle 7, while the instructions in the first FP are in the first execution phase E1 (which may be the only one), the instructions in the second FP are in the decoding phase, the instructions in the third FP are in the dispatching phase, and so on. All seven instructions are proceeding through the various phases. Therefore, at cycle 7, "the pipeline is full."

Most instructions have one execute phase. Instructions such as multiply (MPY), load (LDH/LDW), and branch (B) take two, five, and six phases, respectively. Additional execute phases are associated with floating-point and double-precision types of instructions, which can take up to 10 phases. For example, the double-precision multiply operation (MPYDP), available on the C67x, has nine delay slots, so that the execution phase takes a total of 10 phases.

The *functional unit latency*, which represents the number of cycles that an instruction ties up a functional unit, is 1 for all instructions except double-precision instructions, available with the floating-point C67x. Functional unit latency is different from a delay slot. For example, the instruction MPYDP has four functional unit latencies but nine delay slots. This implies that no other instruction can use the associated multiply functional unit for four cycles. A store has no delay slot but finishes its execution in the third execution phase of the pipeline.

If the outcome of a multiply instruction such as MPY is used by a subsequent instruction, a NOP (no operation) must be inserted after the MPY instruction for the pipelining to operate properly.Four or five NOPs are to be inserted in case an instruction uses the outcome of a load or a branch instruction, respectively.

4.6 REGISTERS

Two sets of register files, each set with 16 registers, are available: register file A (A0 through A15) and register file B (B0 through B15). Registers A0, A1, B0, B1, and B2 are used as conditional registers. Registers A4 through A7 and B4 through B7 are used for circular addressing. Registers A0 through A9 and B0 through B9 (except B3) are temporary registers. Any of the registers A10 through A15 and B10 through B15 used are saved and later restored before returning from a subroutine.

A 40-bit data value can be contained across a register pair. The 32 least signifi- cant bits (LSBs) are stored in the even register (e.g., A2), and the remaining 8 bits are stored in the 8 LSBs of the next-upper (odd) register (A3). A similar scheme is used to hold a 64-bit double-precision value within a pair of registers (even and odd).

These 32 registers are considered general-purpose registers. Several specialpurpose registers are also available for control and interrupts: for example, the address mode register (AMR) used for circular addressing and interrupt control registers.

4.7 LINEAR AND CIRCULAR ADDRESSING MODES

Addressing modes determine how one accesses memory. They specify how data are accessed, such as retrieving an operand indirectly from a memory location. Both linear and circular modes of addressing are supported. The most commonly used mode is the indirect addressing of memory.

4.7.1 Indirect Addressing

Indirect addressing can be used with or without displacement. Register R represents one of the 32 registers A0 through A15 and B0 through B15 that can specify or point to memory addresses. As such, these registers are pointers. Indirect addressing mode uses a "*" in conjunction with one of the 32 registers. To illustrate, consider R as an address register.

1. *R. Register R contains the address of a memory location where a data value
   is stored.
2. *R++(d). Register R contains the memory address (location). After the
   memory address is used, R is postincremented (modified) such that the new
   address is the current address offset by the displacement value d. If d = 1 (by
   default), the new address is R + 1, or R is incremented to the next higher
   address in memory. A double minus (--) instead of a double plus would
   update or postdecrement the address to R - d.

3. **++R(d).* The address is preincremented or offset by d, such that the current address is R + d. A double minus would predecrement the memory address so that the current address is R - d.

4. **+R(d).* The address is preincremented by d, such that the current address is R + d (as with the preceding case). However, in this case, R preincrements without modification. Unlike the previous case, R is not updated or modified.

4.7.2 Circular Addressing

Circular addressing is used to create a circular buffer. This buffer is created in hardware and is very useful in several DSP algorithms, such as in digital filtering or correlation algorithms where data need to be updated.

The C6x has dedicated hardware to allow a circular type of addressing. This addressing mode can be used in conjunction with a circular buffer to update samples by shifting data without the overhead created by shifting data directly. As a pointer reaches the end or "bottom" location of a circular buffer that contains the last element in the buffer, and is then incremented, the pointer is automatically wrapped around or points to the beginning or "top" location of the buffer that contains the first element.

Two independent circular buffers are available using BK0 and BK1 within the AMR. The eight registers A4 through A7 and B4 through B7, in conjunction with the two .D units, can be used as pointers (all registers can be used for linear addressing).The following code segment illustrates the use of a circular buffer using register B2 (only side B can be used) to set the appropriate values within AMR:

```
MVKL .S2 0x0004,B2 ;lower 16 bits to B2. Select A5 as pointer
MVKH .S2 0x0005,B2 ;upper 16 bits to B2. Select BK0, set N = 5
MVC .S2 B2,AMR ;move 32 bits of B2 to AMR
```

The two move instructions MVKL and MVKH (using the .S unit) move 0x0004 into the 16 LSBs of register B2 and 0x0005 into the 16 most significant bits (MSBs) of B2. The MVC (move constant) instruction is the only instruction that can access the AMR and the other control registers (shown in Appendix B) and executes only on the B side in conjunction with the functional units and registers on side B. A 32- bit value is created in B2, which is then transferred to AMR with the instruction MVC to access AMR.

The value 0x0004 = (0100)b into the 16 LSBs of AMR sets bit 2 (the third bit) to 1 and all other bits to 0. This sets the mode to 01 and selects register A5 as the pointer to a circular buffer using block BK0.

Table 3.4 shows the modes associated with registers A4 through A7 and B4 through B7.The value 0x0005 = (0101)b into the 16MSBs of AMR sets bits 16 and 18 to 1 (other bits to 0). This corresponds to the value of N used to select the size of the buffer as $2N+1$ = 64 bytes using BK0. For example, if a buffer size of 128 is desired using BK0, the upper 16 bits of AMR are set to (0110)b = 0x0006. If assembly code is used for the circular buffer, as execution returns to a calling C function, AMR needs to be reinitialized to the default linear mode. Hence the pointer's address must be saved.

TABLE 3.4   AMR Mode and Description

Mode	Description
0 0	For linear addressing (default on reset)
0 1	For circular addressing using BK0
1 0	For circular addressing using BK1
1 1	Reserved

4.8 TMS320C6x INSTRUCTION SET

4.8.1 Assembly Code Format

An assembly code format is represented by the field

Label || [ ] Instruction Unit Operands ;comments

A label, if present, represents a specific address or memory location that contains an instruction or data. The label must be in the first column. The parallel bars (||) are there if the instruction is being executed in parallel with the previous instruction. The subsequent field is optional to make the associated instruction conditional. Five of the registers—A1, A2, B0, B1, and B2—are available to use as conditional registers. For example, [A2] specifies that the associated instruction executes if A2 is not zero. On the other hand, with [!A2], the associated instruction executes if A2 is zero. All C6x instructions can be made conditional with the registers A1, A2, B0, B1, and B2 by determining when the conditional register is zero. The instruction field can be either an assembler directive or a mnemonic. An assembler directive is a command for the assembler. For example,

.word value

reserves 32 bits in memory and fill with the specified *value*. A mnemonic is an actual instruction that executes at run time. The instruction (mnemonic or assembler directive) cannot start in column 1. The Unit field, which can be one of the eight CPU units, is optional. Comments starting in column 1 can begin with either an asterisk or a semicolon, whereas comments starting in any other columns must begin with a semicolon.

   Code for the floating-point processors C3x/C4x is not compatible with code for the fixed-point processors C1x, C2x, and C5x/C54x. However, the code for the fixed-point processors C62x is compatible with the code for the floating-point C67x. C62x code is actually a subset of C67x code. Additional instructions to handle double-precision and floating-point operations are available only on the C67x processor. Also, some additional instructions are available only on the fixed-point C64x processor.

Several code segments are presented to illustrate the C6x instruction set. Assembly code for the C6x processors is very similar to C3x/C4x code. Single-task types of instructions available for the C6x make it easier to program than either the previous generation of fixed- or floating-point processors. This contributes to an efficient compiler. Additional instructions available on the C64x (but not on the C62x) resemble the multitask types of instructions for C3x/C4x processors. It is very instructive to read the comments in the programs discussed in this book. Appendix A contains a list of the instructions for the C62x/C67x processors.

4.8.2 Types of Instructions

The following illustrates some of the syntax of assembly code. It is optional to specify the eight functional units, although this can be useful during debugging and for code efficiency and optimization.

1. *Add/Subtract/Multiply*
  (a) The instruction

   ADD .L1 A3,A7,A7 ;add A3 + A7 ÆA7 (accum in A7)

   adds the values in registers A3 and A7 and places the result in register
   A7. The unit .L1 is optional. If the destination or result is in B7, the unit
   would be .L2.
 (b). The instruction

   SUB .S1 A1,1,A1 ;subtract 1 from A1

    subtracts 1 from A1 to decrement it using the .S unit.
(c). The parallel instructions

MPY .M2 A7,B7,B6 ;multiply 16 LSBs of A7, B7 Æ B6

|| MPYH .M1 A7,B7,A6 ;multiply 16MSBs of A7, B7 ÆA6

multiplies the lower or least significant 16 bits (LSBs) of both A7 and B and places the product in B6, in parallel (concurrently within the sameexecution packet) with a second instruction that multiplies the higher or most significant 16 bits (MSBs) of A7 and B7 and places the result in A6. In this fashion, two MAC operations can be executed within a single instruction cycle. This can be used to decompose a sum of products into two sets of sum of products: one set using the lower 16 bits to operate on the first, third, fifth, . . . number and another set using the higher 16 bits to operate on the second, fourth, sixth, . . . number. Note that the parallel symbol is not in column 1.

2. *Load/Store*

  (a) The instruction

        LDH .D2 *B2++,B7 ;load (B2) Æ B7, increment B2

|| LDH .D1 *A2++,A7 ;load (A2) Æ A7, increment A2

  loads into B7 the half-word (16 bits) whose address in memory is speci-
  fied/pointed to by B2.Then register B2 is incremented (postincremented)
  to point at the next higher memory address. In parallel is another indirect
  addressing mode instruction to load into A7 the content in memory whose
  address is specified by A2. Then A2 is incremented to point at the next
  higher memory address.

  The instruction LDW loads a 32-bit word. Two paths using .D1 and .D2
  allow for the loading of data from memory to registers A and B using the
  instruction LDW.The double-word load floating-point instruction LDDW on
  the C6713 can simultaneously load two 32-bit registers into side A and
  two 32-bit registers into side B.

(b) The instruction

      STW .D2 A1,*+A4[20] ;store A1Æ(A4) offset by 20

stores the 32-bit word A1 in memory whose address is specified by A4 offset by 20 words (32 bits) or 80 bytes. The address register A4 is preincremented with offset but it is not modified (two plus signs are used if A4 is to be modified).

3. *Branch/Move*. The following code segment illustrates branching and data transfer:

      Loop MVKL .S1 x,A4 ;move 16 LSBs of x address ÆA4
      MVKH .S1 x,A4 ;move 16 MSBs of x address ÆA4
      .
      .
      .
      SUB .S1 A1,1,A1 ;decrement A1
      [A1] B .S2 Loop ;branch to Loop if A1 # 0
      NOP 5 ;five no-operation instructions
      STW .D1 A3,*A7 ;store A3 into (A7)

The first instruction moves the lower 16 bits (LSBs) of address x into register A4. The second instruction moves the higher 16 bits (MSBs) of address x into A4, which now contains the full 32-bit address of x. One must use the instructions MVKL/MVKH in order to get a 32-bit constant into a register.

Register A1 is used as a loop counter. After it is decremented with the SUB instruction, it is tested for a conditional branch. Execution branches to the label or address Loop if A1 is not zero. If A1 = 0, execution continues and data in register A3 are stored in memory whose address is specified (pointed) by A7.

CHAPTER-5

REAL TIME IMPLEMENTATION OF THE DES ENCRYPTION ALGORITHM BY USING TMS320C6713 DSP PROCESSOR

5.1 Implementation of speech signal encryption by using DES

In this project I have used 32-bit floating point TMS320C6713 for real time implementation of DES encryption algorithm. The reason for choosing TMS320C6713 Processor is

1. High speed arithmetic.

2. Robust in data transfer to and from real world.

3. Multiple access memory structure.

4. Less power and cheap.

Features of TMS320C6713 DSK processor

1. Floating point device.

2. Operating frequency 225MHz.

3. VLIW architecture.

4. 264Kb internal cache, 16Mb SDRAM, 512 Kb flash.

5. 1800 MIPS or 1350 MFLOPS per Second.

6. TLV320AIC23 (AIC23) Onboard Stereo Codec for I/O operations (range 8KHz-96KHz)

7. +5v universal power supply.

In this project I have implemented DES algorithm on TMS320C6713. For testing purposes, the first three onboard switches were utilized: *sw*0 for selecting different keys; *sw*1 to enable encryption only, or both encryption and decryption; and *sw*2 as an on/off switch (a loop program).

In this project I have replaced each sample of voice by 15 binary bits. Here I have taken four samples at a time and converted them into 60 binary digits. For example take 4 samples of speech as follows

Amplitude of sample 1 is: '4'
It is replaced as follows
0000000000000100

In this 1st bit represents the sign of the sample if it is positive then it is '0' else it is '1'. Like this I have obtained 16 binary bits.

Now take second sample as '-4'
It is replaced as follows
1000000000000100

Here 1st bit '1' represents that the amplitude of sample is negative.

Like this I have replaced amplitude of 1st four samples by 64 binary bits and then I have applied DES algorithm for these 64 bits with the given key. After this I have taken next four samples and applied DES algorithm like wise this process continues for all samples of given speech signal.

5.1 MATLAB Results for DES Algorithm:

In this project first I have write a matlab code for the DES algorithm. Here I have taken 64-bit data and 56-bit key for encryption.

Here I have taken the input data 'x' as
x = 1111000011110000111111110000111100000000111111111111100000000
This is the 64-bit input data.

And key 'k' is

k =0101100111101101011100111011100111101100111101100111

I have got the cipher text output as

Cipher out =

0110011101100111011011000110110001100111011001110110110110001101100

And then I have got the decipher output same as given data in put

Decipher output=

11110000111100001111111100001110000000011111111111100000000

So here I have got the output decipher data same as the given input data which means I am recollecting the same data after decryption.

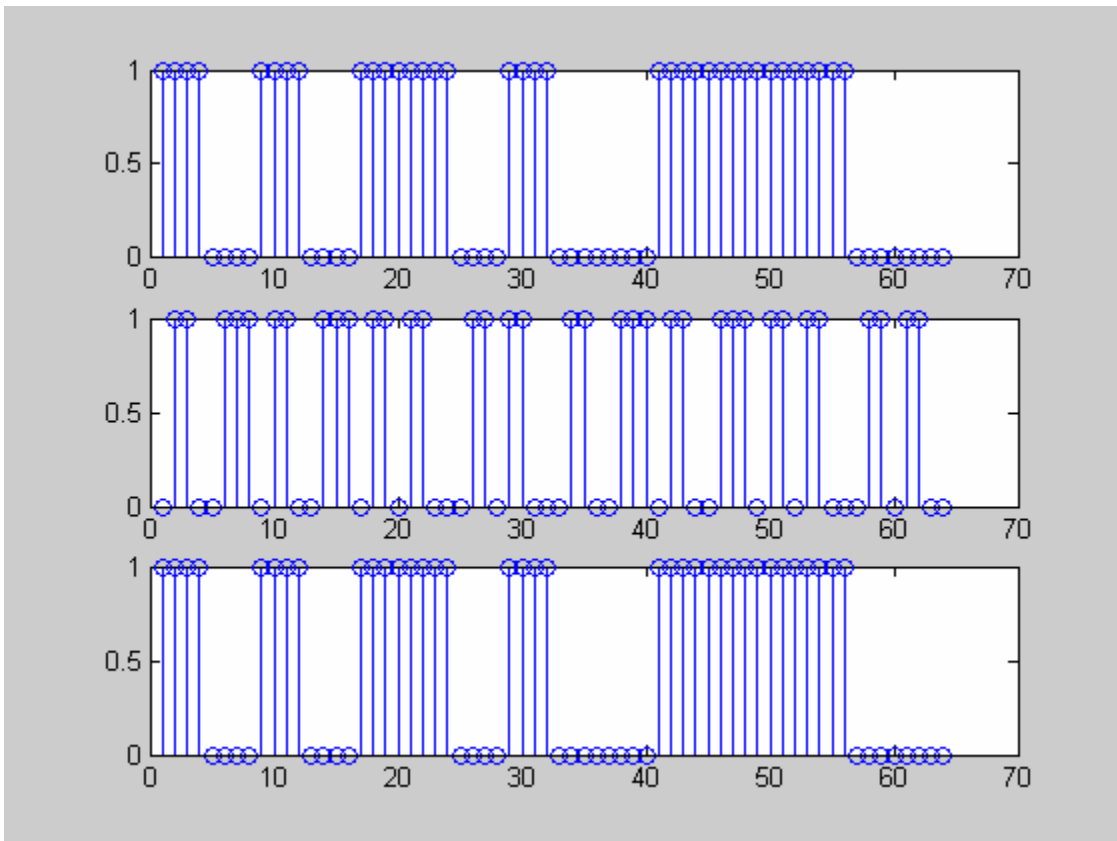The matlab plots for DES algorithm are given below

Fig 5.1: Matlab results for DES algorithm.

In the above plot first plot the is plot of input 64-bit data 'x'. Second plot is the is corresponding cipher output and third one is the decipher output which is same as input.

5.2 Code composer studio Results for DES Algorithm:

For implementing the DES algorithm to voice data I have used TMS320C6713 DSK and I have used the CCS 3.1 version for simulation. The simulation output of CCS 3.1 studio for thousand samples of voice data is given by
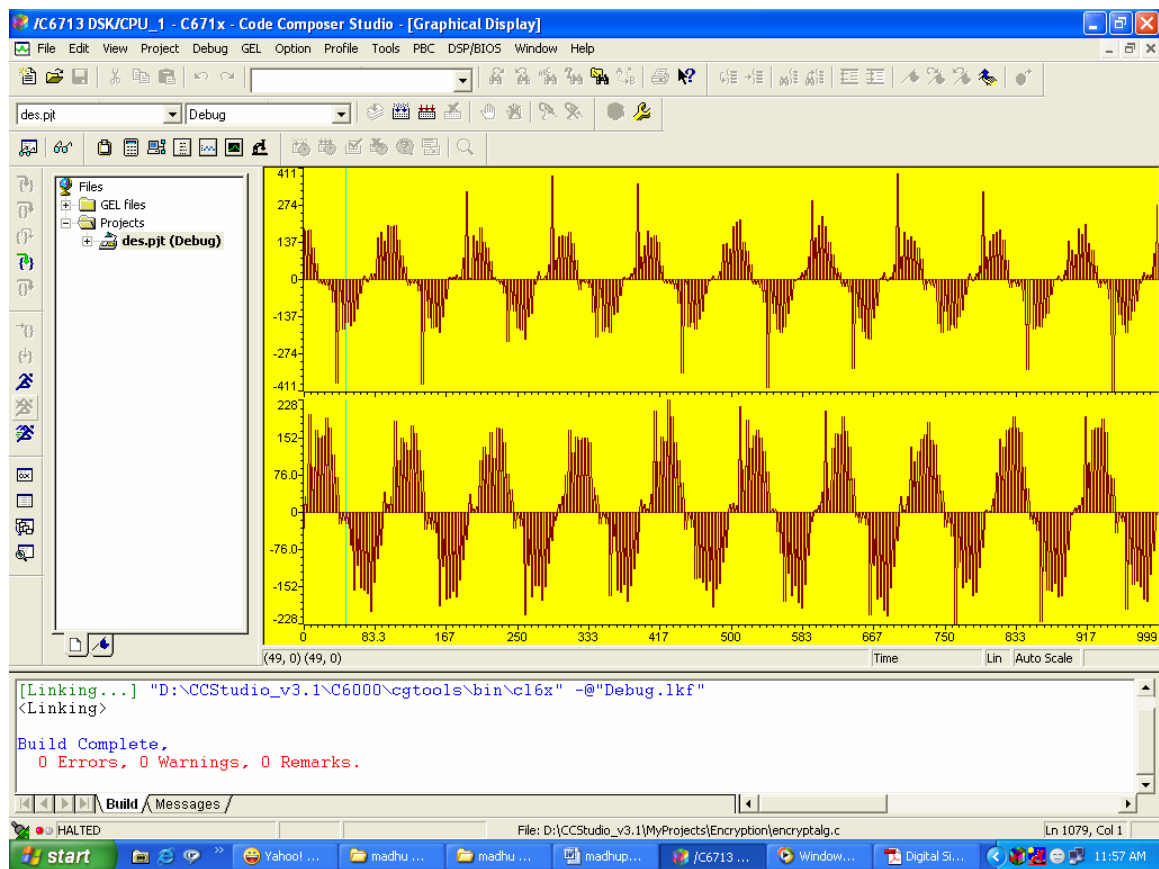

Fig 5.2: CCS output for DES algorithm

This is the final output I have got. In the above figure 1st plot is for the input voice data. Here I have taken only thousand samples. The second plot shows the decipher output which is same as the input data means I am recollecting the same voice data after decryption. There is slight difference between two figures in the above plot it is due to the delay means the program for this DES algorithm is very complex that's why its taking

72

time to execute the program. But in the speaker I am recognizing thee same voice with less disturbance. This problem may be avoided by using the two DSK's one for encryption and another for decryption but maintaining synchronism between two DSK's is important.

CONCLUSION

Now a days there are many situations to keep voice secret in voice communication. This can be achieved by voice encryption by using different encryption algorithms. In this project I have implemented DES algorithm on TMS320C6713 DSK for voice encryption. DES algorithm is a symmetric key algorithm means it uses the same key for encryption and decryption. In this DES algorithm input data is 64-bit length and key is 56-bits. In this project I have done simulations in both matlab and in the code composer studio version 3.1. In this project I have used TMS 320C6713 DSK for real time implementation. It is a 32-bit floating point processor. In this thesis I have presented the results in matlab and in CCS 3.1.

References:

1. Digital signal processing and applications with the C6713 and C6416 DSK by Rulph Chassaing, 2005.

2. Improved Data Encryption by Seung-Jo Han, Heang-Soo oh, Jongan Park, ieee 2001.

3. Energy consumption of Encryption schemes by Sohail Harani, April 9, 2003.

4. Willam Stallings "cryptography and network security" , fourth edition, 2003.

5. Implementation of DES algorithm on TMS320C6713 by Rulph Chassaing, 2004.

6. ABRA87 Abrams, M., and Podell, H. Computer and Network Security. Los Alamitos, CA: IEEE Computer Society Press,1997.

7. Implementation approaches for the Advanced Encryption Standard algorithm, Xinmiao Zhang; Parhi, K.K.; IEEE Circuits and Systems Magazine, Volume: 2 Issue: 4 Fourth Quarter 2002 Page(s): 24 –46.

8. IEEE P1363a and IEEE 1363: Standard Specification for Public key Cryptography.

9. IJCSNS International Journal of Computer Science and Network Security, VOL.6 No.1B, January 2006 "Cryptanalysis of Simplified Data Encryption Standard via Optimisation Heuristics".

10. Clark A and Dawson Ed, "Optimization Heuristics for the Automated Cryptanalysis of Classical Ciphers", Journal of Combinatorial Mathematics and Combinatorial Computing, Vol. 28,pp. 63-86, 1998.